Universidad ORT Uruguay

Facultad de Ingeniería

Programación basada en invariantes: un enfoque didáctico

Entregado como requisito para la obtención del título de Ingeniero en Sistemas

Alejandro Milieris – 150251

Eitan Fogel – 174501

Tutor: Álvaro Tasistro

2016

Declaración de autoría

Nosotros, Alejandro Milieris y Eitan Fogel, declaramos que el trabajo que se presenta en esa obra es de nuestra propia mano. Podemos asegurar que:

• La obra fue producida en su totalidad mientras realizábamos el proyecto final de carrera;

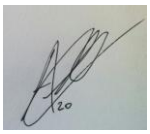
• Cuando hemos consultado el trabajo publicado por otros, lo hemos atribuido con claridad;

• Cuando hemos citado obras de otros, hemos indicado las fuentes. Con excepción de estas citas, la obra es enteramente nuestra;

• En la obra, hemos acusado recibo de las ayudas recibidas;

• Cuando la obra se basa en trabajo realizado conjuntamente con otros, hemos explicado claramente qué fue contribuido por otros, y qué fue contribuido por nosotros;

• Ninguna parte de este trabajo ha sido publicada previamente a su entrega, excepto donde se han realizado las aclaraciones correspondientes.



Alejandro Milieris

Eitan Fogel

Fecha: 03 de Marzo de 2016

Agradecimientos

En primer lugar queremos agradecer a nuestra familia y amigos, por su apoyo incondicional a lo largo del proyecto y de toda la carrera.

A nuestro tutor Álvaro Tasistro, que siempre cumplió a la perfección su rol como tutor y nos asistió con todo lo que necesitamos, y nos guió durante toda la duración del proyecto.

Por último, a Carlos Luna, Daniel Zingaro, Ralph Back y Javier Blanco, por dedicarnos su tiempo y responder a nuestras inquietudes.

Abstract

Se presenta una metodología para la derivación de algoritmos en estilo imperativo a partir de pre y postcondiciones, basada en el uso de invariantes.

Concretamente se propone:

- Una notación para expresar las especificaciones y los algoritmos anotados con sus invariantes y variantes.
- Métodos de resolución de problemas mediante iteración.
- Técnicas de derivación concreta de invariantes a partir de postcondiciones.

El fin del trabajo es servir como medio didáctico para esta disciplina de programación. Esto se desarrolla a través de una lista extensiva de ejemplos.

También se efectúa un relevamiento del estado del arte del uso de la programación basada en invariantes en educación terciaria.

Palabras clave

Invariantes, Metodología de la Programación, Didáctica de la Programación, Programación basada en invariantes.

Índice

1. Introducción

“Mi código es correcto: compila perfectamente y lo he probado con varios casos”. Cualquier profesional o estudiante del área ha dicho u oído esta frase antes. Sin embargo son pocas las personas que pueden explicar por qué el código que acaban de escribir funciona. A la hora de enfrentarse a un problema es común en estudiantes sentarse frente al ordenador e intentar resolverlo una y otra vez, hasta que finalmente logran dar con una solución. Esta tarea si bien puede ser eficaz, no siempre involucra un uso eficiente del tiempo del estudiante, ni suele dar con soluciones óptimas. Peor aún, no siempre el programador puede explicar cómo o por qué su programa funciona.

Saber programar no es una habilidad valiosa en sí misma. El valor está en la capacidad del programador en solucionar los problemas a los que se enfrenta así como en el diseño de la propia solución. Cuando un programador tiene una dificultad en la solución de un problema, ésta suele ser una dificultad en el diseño, y no en la codificación de la misma. Esto entra en contradicción con el enfoque que a veces se da en la educación terciaria en el rubro. Cada vez es más común que los cursos se enfoquen en el aprendizaje de la tecnología de turno, en sus características técnicas y en su correcto uso [1].

Si bien consideramos muy importante aprender y estar al tanto de las últimas tecnologías de la industria, también creemos que hay conocimientos que se deben tener incorporados que serán esenciales en cualquiera de estas tecnologías, así como contar con una metodología formal de solución de problemas, intentando independizarse al máximo de las características de los distintos lenguajes de programación.

Buscamos evitar la “programación artesanal”, aquella que surge de momentos de inspiración o de la aplicación sucesiva de la prueba y el error, y sustituirla por “programación metódica”, es decir, la aplicación sistemática de ciertos pasos que produzcan primero un profundo entendimiento del problema en la mente del desarrollador para luego llevar a una solución eficaz, *performante*, y de manera que su autor pueda comprender y explicar qué fue lo que hizo, por qué lo hizo y en qué consiste su código.

Conceptos como la correcta especificación de un problema, cuándo introducir una nueva variable, cómo construir un *loop* con su correcta inicialización y condición de salida, entre otros, son los que intentaremos transmitir mediante la metodología que presentamos.

El objetivo de esta tesis es generar un material que pueda servir como base para la elaboración y planificación de un curso universitario enfocado en la derivación de soluciones a problemas recurrentes en la computación mediante el uso de la programación basada en invariantes, que busque formar a los estudiantes con recursos y herramientas para asegurar una correcta especificación de los problemas y los algoritmos desarrollados, la simplicidad y eficiencia de su código, así como la comprensión del problema y la solución derivada. En resumen, ayudar a los estudiantes a mejorar la calidad del diseño e implementación de las soluciones a los problemas a los que se enfrentan, y así convertirse en mejores profesionales.

Este material busca aportar un valor didáctico para estudiantes que estén transitando sus primeras experiencias en la programación y apunta a que estos estudiantes puedan comprender los contenidos aquí presentados por sus propios medios. Para esto, asumimos que el lector cuenta con ciertos conocimientos elementales, a saber:

- Noción de **memoria y su estado**, determinado por el contenido de sus **variables**.
- Los **tipos de datos entero, booleano, caracteres, cadenas de texto (*string*) y *array*** con sus respectivas operaciones básicas.
- Cómo se ejecutan las distintas **formas de instrucciones**, es decir:
  a) la **asignación**, como instrucción elemental,
  b) la **secuenciación**, la **bifurcación** o análisis de casos (*if*) y la **repetición** (*loop*, iteración), como formas de organizar la ejecución de instrucciones elementales, y la idea de **método**, su firma y su cuerpo, pudiendo distinguir y reconocer sus parámetros, su retorno y su nombre.

Mediante investigación de la bibliografía existente y el contacto directo con sus autores buscamos obtener una noción del estado del arte a nivel regional y mundial en el uso de invariantes y la introducción de técnicas formales de derivación de algoritmos en la educación terciaria. Concretamente buscamos cubrir un espectro significativo del material publicado acerca del tema, de experiencias y aprendizajes adquiridos de la introducción de estos conocimientos en la docencia y la respuesta brindada por los alumnos. Esta información resulta fundamental para construir a partir de ella y evitar toparse con las mismas dificultades e inconvenientes con las que ha tenido que lidiar la comunidad científica previamente, así como imitar y aprender de las experiencias exitosas.

Existe mucha bibliografía y textos de referencia basados en métodos formales. Los más destacados son los publicados por Dijkstra [2] [3] y Kaldewaij [4]. Estos documentos presentan explicaciones en términos estrictamente formales -es decir, en lógica de primer orden- , utilizando una notación convencional y brindando demostraciones completas desde el punto de vista lógico. Sin embargo, no es abundante la bibliografía acerca de invariantes, y es aún menor la cantidad de textos que la presentan de forma accesible para un estudiante que esté dando sus primeros pasos en las ciencias de la computación. Por su parte Daniel Zingaro [5] utiliza un enfoque mucho más didáctico, en lenguaje natural y brindando explicaciones comprensivas para el público al que apuntamos. Por más que sus demostraciones no sean completas desde el punto de vista estrictamente formal, logra aportar desde la aplicación de conocimientos a ámbitos fácilmente identificables por cualquier programador.

El enfoque que utilizan Kaldewaij [4] y Dijkstra [2] [3], y que catalogamos como “estrictamente formal” tiene la virtud de permitir el desarrollo de lenguajes tales que, si los programas compilan, es porque funcionan: son correctos y terminan. Si bien esto es sumamente valioso, es muy costoso en términos de aprendizaje y de aplicación. Un ejemplo de lenguajes donde el compilador cumple el rol de verificador es *Dafny* [6], desarrollado por Microsoft en los últimos años. El programador debe proveer especificaciones e invariantes completas, cumpliendo con la sintaxis que exige la tecnología. En muchos casos también se debe proveer demostraciones formales de propiedades que el compilador no puede verificar por su cuenta. Esto implica un doble trabajo

para el programador, ya que debe desarrollar el código "ejecutable" y el código "lógico-matemático" para soportar el primero.

El costo de aplicación de este tipo de programación y el del aprendizaje correspondiente son altos. Sin embargo creemos que los métodos subyacentes, basados en invariantes, pueden aprenderse y aplicarse de manera semi-formal, es decir, sin emplear un lenguaje formal de lógica que exija el máximo detalle en cada demostración, y aun así proveerán ventajas considerables en la formación del programador.

La intención en este trabajo es proveer una metodología que permita al programador resolver problemas con mayor facilidad, convencerse de la corrección de su diseño y poder explicarlo. Para ello no es necesario expresar todos los detalles de especificaciones y demostraciones en un lenguaje totalmente formal, al igual que en Matemática esto último tampoco es necesario para entender los teoremas.

Para cumplir el objetivo de evitar la programación artesanal buscamos desarrollar un orden de razonamiento que, de seguirse en forma metódica, nos ayuda a acercarnos a la solución de un amplio abanico de problemas, que luego desarrollamos a lo largo del trabajo. Para facilitar esto hemos clasificado las soluciones a los problemas en dos **métodos** de solución de problemas, y hemos compilado cuatro **técnicas** de derivación de invariantes.

Los métodos representan el algoritmo en su forma más abstracta. Son más intuitivos que las técnicas, y son la forma más fácil de comenzar a razonar sobre un problema: pensar qué hacer, para luego decidir cómo. Las técnicas fueron tomadas de la bibliografía existente [4] [7] [8] y cumplen un rol mucho más sintáctico. El objetivo de las técnicas es facilitar la tarea de derivar la invariante. A lo largo del trabajo mostramos aplicaciones de cada una de las técnicas, así como una explicación de las mismas en base a diversos ejemplos.

La mayor parte de los textos disponibles apoyan sus ejemplos con implementaciones de los algoritmos desarrollados en lenguajes industriales, o utilizando la notación convencional de lógica formal. Para reforzar la idea de que es posible abstraerse de las tecnologías específicas a la hora de adquirir los conocimientos planteamos una notación propia para resolver los problemas que se explica en el documento y se utiliza a lo largo del mismo.

El cuerpo del trabajo está compuesto por siete capítulos, que se conforman de la siguiente manera:
En el primero de ellos presentamos la notación que utilizaremos para resolver los problemas, definimos los conceptos de pre y postcondición, invariante y variante, e introducimos el proceso de derivación (es decir, los pasos a seguir) para resolver un problema. Mientras vamos presentando estos conceptos los utilizamos para resolver dos problemas clásicos como lo son la pertenencia de un elemento a un *array* y la búsqueda del menor divisor de un natural, aplicando para cada uno de ellos un método diferente de resolución de problemas.
En el segundo capítulo presentamos otros problemas clásicos sobre *arrays* y enteros, y a medida que los resolvemos vamos introduciendo las diferentes técnicas de derivación de invariantes que son utilizadas a lo largo del trabajo. El siguiente apartado presenta algoritmos de búsqueda, en los que tratamos tanto búsqueda lineal como binaria, y la comparación de la eficiencia entre una y

otra. Luego continuamos con algoritmos de *sorting*, entre los que destacamos *quick sort, selection sort, insertion sort y flag sort.* Posteriormente exponemos algoritmos más avanzados entre los que resaltan la sucesión de Fibonacci, el cálculo del cubo de un número solamente utilizando sumas, y la Criba de Eratóstenes. La sexta sección se enfoca en Programación dinámica, y finalmente en el último capítulo presentamos una serie de juegos y acertijos entretenidos, para exponer el uso de invariantes en un contexto que se sale de lo tradicional. Al término de estos siete capítulos, exhibimos la bibliografía que más nos ha aportado e influido para la confección de nuestro trabajo y las experiencias del uso de invariantes en la educación, para luego cerrar con conclusiones.

2.Ejemplos introductorios

Este capítulo ilustra dos ejemplos que resultan sencillos para cualquier programador, incluso principiante. Estos ejemplos ayudarán a introducir los conceptos que son relevantes a la hora de aplicar la programación basada en invariantes, mostrar cómo se pueden construir algoritmos usando invariantes y asegurarnos que sean correctos, e introducir la notación y el esquema de derivación que usaremos en todos los problemas. Los conceptos se explican a medida que aparecen para facilitar la comprensión al lector. En este capítulo presentaremos los dos métodos de soluciones de problemas que utilizaremos a lo largo del trabajo.

Para no estar atados a las particularidades de cada lenguaje, y para dejar claro que los conceptos no dependen de la tecnología de turno usaremos una notación propia para resolver los problemas. Lo más importante a destacar de esta notación es que buscamos seguir la misma línea de razonamiento que ocurre en la cabeza del programador a la hora de pensar las soluciones a los problemas. Esto facilita la derivación de algoritmos de una forma más sistemática y metódica. La mayoría de los lenguajes contemporáneos imponen un orden de razonamiento imperativo, es decir, el orden en el que se escribe el programa es el mismo que posteriormente se ejecutará.

2.1.Pertenencia a un array

El primer problema consiste en: dado un *array* y un elemento que recibimos por parámetro, determinar si este elemento pertenece o no al *array*. Para no asumir nada sobre el tipo de dato del elemento a buscar y de los elementos del *array*, decimos que son de tipo T, un tipo de datos genérico para el que están definidas las operaciones de comparación (==, ≠, <, >, ≤, ≥).

Pre y postcondiciones
Todo problema al que nos enfrentamos tiene un conjunto de parámetros, un punto de partida y un punto final. El primer paso para resolver un algoritmo, es entender de dónde partimos y a dónde queremos llegar, es decir, especificar el problema en términos de pre y postcondiciones.

La especificación del problema es una de las partes más importantes del desarrollo de la solución del mismo. La experiencia recopilada a partir de textos [9], entrevistas con autores como Carlos Luna e intercambios privados con Daniel Zingaro y Ralph-Johan Back ha demostrado que esta es la parte más desafiante para los alumnos debido a la carencia de un estándar y una nomenclatura, sumado a que el estudiante no suele ver la utilidad de esta etapa [9]. Éstas varían según el contexto y la naturaleza del dominio del problema.

La correcta comprensión del problema nos dará la especificación exacta que buscamos. Dos errores muy comunes a la hora de especificar son los de agregar condiciones innecesarias (sobre especificar), lo que puede llevar a no tomar en cuenta soluciones válidas, y no incluir condiciones necesarias (sub especificar), de manera que podríamos admitir soluciones que no resuelven el problema en cuestión.

Al punto de partida lo denominamos precondición, y son las características que podemos asumir que cumplen los parámetros que recibimos. No es nuestro trabajo a la hora de diseñar un algoritmo asegurar que se cumpla esto, sino que simplemente asumimos que es así. En este caso en particular no hay ninguna precondición que sea estrictamente necesaria, pero veremos problemas que sí lo requieren. Las precondiciones, cuando existen, las expresaremos con la palabra *requires* seguida de la condición que se debe cumplir.

Por otra parte tenemos el concepto de postcondición. Una postcondición es una condición que debe cumplirse luego de la ejecución del algoritmo. En otras palabras: es lo que debe hacer el algoritmo. Todos los algoritmos cuentan con una postcondición. Si no tuvieran postcondición no harían nada. La mayoría de las veces, como en este caso, representa información acerca de las variables que se retornan. Las expresamos de forma similar a las precondiciones, usando la palabra *ensures*, seguido de la propia condición. Por definición del problema nuestra postcondición es que esta es *true* si y sólo si el elemento e se encuentra en el *array* a, y *false* en caso contrario.

Postcondición: esta es true si y sólo si el elemento e se encuentra en el array a

El algoritmo parcialmente toma la siguiente forma:

method miembroDel*Array* (T [ ] a, T e): boolean esta {
requires: -
ensures: esta es true si y sólo si el elemento e se encuentra en el *array* a.

}

La palabra clave *method* indica que se trata de un método. Todos los algoritmos que veremos los expresaremos en forma de métodos. Seguido de la palabra clave va el nombre del método, que permite identificar al mismo, por lo que se busca un nombre mnemotécnico, al igual que para las variables. Luego vienen los parámetros. Éstos se expresan entre paréntesis y separados por comas. Para cada parámetro, primero ubicamos el tipo de datos, seguido del nombre de la variable. Luego de la firma ubicamos los dos puntos seguido del retorno de la función. En este caso retornaremos una variable de tipo booleano y nombre esta.

Formulando la invariante.
Una vez que hemos definido claramente cuáles son las pre y pos condiciones, es decir, de dónde partimos y a dónde queremos llegar, tenemos que encontrar la forma de llegar de una a otra.
En este caso sabemos que vamos a tener que iterar todo el *array* que hemos recibido para poder determinar si el elemento forma parte del mismo. Pero solamente saber que vamos a recorrer un *array* no es suficiente para poder resolver este problema. ¿Cómo lo haremos? ¿Dónde comenzamos? ¿Cuándo terminamos? ¿Qué haremos en cada iteración? Todas estas preguntas forman parte del diseño y la derivación de un algoritmo.

En la mayoría de las iteraciones contamos con un índice al que llamamos *i* que oscila dentro de un rango definido. Cuando esta variable toma el valor del último índice del *array* sabemos que

hemos terminado la recorrida. Además contamos con la variable *esta* que nos dice si el elemento pertenece al *array*.

La mejor manera de comenzar a resolver un problema es identificar qué propiedades se cumplen siempre en este contexto. Podríamos usar sentencias del estilo N es igual al largo del *array*, o incluso una trivialidad como dos más dos es cuatro. Estas propiedades, si bien son ciertas, no aportan nada para acercarnos a la solución de nuestro problema. También podríamos buscar alguna propiedad más fuerte como por ejemplo: "el valor de esta no ha cambiado". Esta propiedad aporta información útil al problema, pero nada nos asegura que sea cierta siempre. De hecho, fácilmente se puede anticipar que la variable va a cambiar de estado en el momento que encontremos el elemento.

Una propiedad útil en este caso sería esta es verdadero si y sólo si el elemento e se encuentra en la porción ya recorrida del *array*.

Gráficamente podemos verlo en la imagen 2-1:

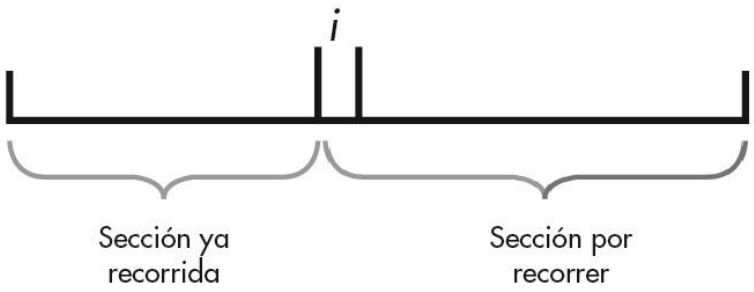


Imagen 2-1: Ejemplo de recorrida en un array

La posición *i* representa la próxima posición a visitar en el *array*. La sección de la izquierda representa lo que ya hemos recorrido, y la sección de la derecha lo que resta por recorrer. Nuestra condición se cumple para todos los elementos del *array* a que se encuentren en posiciones menores que el índice i, pero no sabemos qué ocurre para índices mayores. A medida que *i* avanza esa sección conocida se va haciendo cada vez más grande, y la sección de la derecha es cada vez más pequeña hasta que eventualmente comprenda toda la estructura y habremos concluido la ejecución del algoritmo.

Esta propiedad especial que hemos elegido es lo que llamamos una **invariante**. Definimos una invariante como una **propiedad del programa que se mantiene cierta a lo largo del alcance donde se define**. Puede referirse al estado de una variable en un momento dado de la ejecución, al contenido de alguna estructura de datos o clase, o cualquier dato que sea de utilidad. Es particularmente interesante en situaciones de *loop*, ya que definen una condición o relación que se mantendrá a lo largo de las distintas iteraciones.

Para estos casos tener la garantía de que una condición se cumple en un contexto de incertidumbre (en un *loop* no sabemos el estado de las variables a partir de observar el código) se vuelve muy valioso ya que nos va a servir de base para diseñar el *loop*.

Invariante: esta es verdadero si y sólo si el elemento e se encuentra en el array a[0..i-1]

Para que una invariante se verifique debemos asegurarnos que se cumplan dos condiciones:

1. Cualquier ejecución de las instrucciones del *loop* preserva la invariante.
2. La invariante se cumple antes de entrar al *loop*.

Al establecer que una invariante se cumple luego de cualquier ejecución del *loop*, implícitamente estamos diciendo que se cumple al salir del *loop*, ya que la última ejecución también es "una ejecución cualquiera", por simple reducción matemática.

Además de esta invariante tenemos otra pieza de información muy valiosa. Dado que vamos a recorrer el *array* de izquierda a derecha, sabemos que el índice *i* tendrá como valor mínimo el 0 (que es la primera posición del *array*) y como valor máximo el largo del *array* que recorremos, al que llamamos N. Esto quiere decir que al terminar la recorrida se dará que *i* valdrá lo mismo que N, porque efectivamente ha terminado. De no ser así seguiría en el *loop*.

Por lo tanto al finalizar la ejecución sabemos dos cosas:

1. La invariante se cumple
2. i == N

A sabiendas de estas dos propiedades, podemos reemplazar en la invariante la variable *i* con el largo del *array*, obteniendo: esta es verdadero si y sólo si el elemento e se encuentra en el *array* a[0..N-1]. El *array* a[0..N-1] es el *array* en su totalidad, por lo que cuando se cumpla esta condición habremos arribado a la postcondición, y por lo tanto resuelto el problema.

En nuestra notación expresamos los bucles con la palabra clave *loop*, seguido de llaves donde va el código y las anotaciones que corresponden al alcance del mismo. La condición de terminación del *loop* la expresamos usando la palabra *termination*, seguido de la condición que debe cumplirse para que finalice la ejecución del bucle. Es importante aclarar que si bien la mayoría de los lenguajes de programación manejan el *while* y sus variantes (*for*, *foreach*, etc.), nosotros nos vamos a abstraer de estas particularidades de los distintos lenguajes y mantenernos con iteraciones en su término más general, entendiendo las otras estructuras como casos particulares del *loop*.

Para expresar invariantes utilizaremos la palabra *Invariant*, seguida de la sentencia que define la misma. Esta sentencia se ubica dentro del *loop* porque es el alcance donde aplica la misma. En este caso con la invariante que ya identificamos el algoritmo toma la siguiente forma:

method miembroDelArray (T [ ] a, T e): boolean esta {
    requires: -

ensures: esta es true si y sólo si el elemento e se encuentra en el array a
loop{
invariant: esta es true si y solo si e pertenece a a[0..i-1]
termination: *i* ==N
}
}

Demostrando la corrección
Hasta este momento no sabemos qué sentencias se ejecutarán en cada iteración del *loop*, qué variables usaremos ni cómo se inicializan. Para completar nuestro algoritmo debemos lograr que la invariante se verifique al momento de la inicialización, y luego de un paso cualquiera del *loop*, ya que como vimos, de hacerlo estaríamos arribando a una solución del problema.

Comenzaremos diseñando el paso del *loop* de forma que podamos demostrar que la invariante se mantiene después de una iteración cualquiera del mismo. El paso del *loop* es la porción de código que se va a ejecutar en cada iteración.

Supongamos que estamos en una iteración cualquiera, con lo cual el estado de las variables es incierto pero sabemos que se cumple la invariante. La variable *i* representa la próxima posición del *array* a visitar, todas las posiciones anteriores a *i* ya fueron analizadas y nos resta a[i] y todas las posteriores. Para esta parte de la demostración podemos asumir que la invariante es cierta y podemos usar la información brindada por la misma a nuestro favor, pero siempre debemos asegurarnos que al finalizar la iteración se siga verificando la invariante. Usando la información brindada por la invariante sabemos que la variable esta será cierta si y sólo si el elemento e se encuentra en la sección ya recorrida del *array*.

Analicemos primero el caso en el que el elemento a[i] no contiene el elemento e. El estado de la variable esta no debería cambiar, por lo que si el elemento no había sido encontrado previamente se mantendrá igual, y la invariante seguirá siendo correcta. Si la variable valía true, el elemento ya había sido encontrado, por lo que tampoco deberíamos cambiar su valor. Esto tiene sentido ya que si un elemento se encuentra en el *array* a[0..i-1] también estará en el *array* a[0..i].

En cambio, si el elemento a[i] es igual al elemento e, hemos encontrado el elemento que buscamos, por lo que la variable esta debería ser true a partir de este momento.
Asumiendo que el elemento no había sido encontrado, el estado de la variable esta debe cambiar a true. Previamente la condición decía que esta era false, y el elemento e no pertenecía al *array* a[0..i-1]. Como el elemento e se encuentra en la posición a[i], la condición debe cambiar y la variable esta pasa a estado true. Al finalizar esta iteración, podemos asegurar que el elemento e pertenece al *array* a[0..i]. Si el elemento ya hubiese sido encontrado el comportamiento del algoritmo sería el mismo, simplemente deberíamos sobrescribir la variable esta, asignándole nuevamente el mismo estado. Entonces nuestra condición sería que esta es true y el elemento e sigue perteneciendo al *array*. Nótese que esta situación sólo puede ocurrir en caso de que el *array* acepte elementos repetidos. Por último, en todos los casos debemos incrementar la variable *i* para que el algoritmo avance y continúe la ejecución hasta eventualmente terminar. Así demostramos que este paso del *loop* preserva la invariante.

Paso: Si a[i] es igual al elemento, se asigna esta a true. Independientemente de esto, aumentamos i en una unidad.

Luego de esta explicación es muy fácil predecir cómo será el código resultante. Simplemente un if de la forma:

```
if(a[i]==e) esta = true;
        i = i+1;
```

Para simplificar el código, podemos reemplazar el if por la siguiente sentencia que es equivalente:

```
esta = esta || a[i] == e;
```

Siempre que sea posible es recomendable usar estas pequeñas ventajas. Terminaremos teniendo un código más compacto y elegante.

Para expresar el progreso del algoritmo en nuestra notación utilizaremos la palabra *progress*, seguida de la apertura de llaves para introducir el código de lo que se ejecutará posteriormente. A diferencia de lenguajes como *Dafny*, que soportan el uso de invariantes, no se exige que las demostraciones formen parte del código. Añadiendo el paso del *loop* que verifica la invariante el código se tomará la siguiente forma:

```
method miembroDelArray (T  [ ] a, T e): boolean esta {
        requires: -
        ensures: esta es true si y sólo si el elemento e se encuentra en el array a
        loop{
                invariant: esta es true si y sólo si e pertenece a a[0..i-1]
                termination: i == N
                progress{
                        esta, i = esta || a[i] == e, i+1;
                }
        }
}
```

Hemos demostrado que nuestra invariante se mantiene para una iteración cualquiera del *loop*. Esto era una de las condiciones, pero aún nos falta la segunda: demostrar que antes de entrar al *loop* la invariante era válida. Aquí es donde vemos la importancia de la inicialización de las variables.

Antes de entrar al *loop* debemos inicializar la variable *i* en 0, y *esta* en false. Esto verifica la invariante ya que el segmento a[0..-1] es vacío, y no hay ningún elemento en el conjunto vacío. Por lo tanto esta no es verdadero y el elemento e no está en el *array* definido, con lo que cumplimos la condición de las invariantes que nos faltaba.

La inicialización de las variables se hace utilizando la palabra *Initialization*, seguida de las variables y la asignación de sus valores iniciales.

La notación propuesta admite la asignación simultánea de las variables. La sintaxis es muy sencilla: a la izquierda del signo de igual anotamos todas las variables a asignar separadas por comas, y a la derecha sus respectivos valores también separados por comas, de manera que cada valor se asigna a la variable que aparece en la misma posición. Por ejemplo, si la asignación es 'a, b, c = 1, 2, 3;' la variable a tomará el valor 1, b valdrá 2 y a c se le asignará 3.

Inicialización: i, esta = 0, false;

```
method miembroDelArray (T [ ] a, T e): boolean esta {
        requires: -
        ensures: esta es true si y solo si el elemento e se encuentra en el array a.
        loop{
                invariant: esta es true si y solo si e pertenece a a[0..i-1]
                initialization: esta, i = false, 0;
                termination: i == N
                progress{
                        esta, i = esta || a[i] == e, i+1;
                }
        }
}
```

Hemos entonces demostrado informalmente que la invariante definida es válida previo al ingreso de *loop* y que cualquier iteración del mismo lo mantiene, con lo cual nuestra invariante es correcta. Además hemos demostrado que la invariante junto con la condición de terminación nos lleva a la postcondición, lo que hace que nuestro algoritmo sea correcto y resuelve el problema en cuestión.

Tipos de Invariantes

Toda invariante se puede agrupar dentro de la siguiente clasificación: invariantes esenciales y limitantes [7]. Estas son mutuamente excluyentes.

La invariante que acabamos de derivar entra en la categoría de las invariantes esenciales.
Las invariantes esenciales, como su nombre lo dice, son aquellas que nos aportan información esencial sobre qué es lo que estamos haciendo en cada paso del *loop*. Las más comunes son propiedades del dominio del problema en cuestión, y reducciones o debilitaciones de la postcondición. Nos ayudan a demostrar que en cada paso hacemos lo correcto, y que nos estamos aproximando a la postcondición, y por consiguiente nos ayudan a demostrar la corrección del algoritmo.

Por otro lado tenemos las invariantes limitantes. Prácticamente cualquier problema que desee resolverse mediante invariantes, llevará la introducción de una o más variables. Así como las invariantes esenciales nos dicen qué está pasando con estas variables, las invariantes limitantes establecen los valores entre los que oscilan estas variables. Una invariante limitante puede tener cota superior, inferior, o ambas, como sucede en la mayoría de los casos. Nos ayudan a demostrar

que en cada paso nos estamos acercando a la condición de terminación del *loop*, y están íntimamente ligadas al concepto que veremos a continuación: las **variantes**.
En este caso, sabemos que el menor valor que toma *i* es 0 y el mayor es N, por lo que la invariante limitante establecerá: $0 \leq i \leq N$.

Variante
Por más que pareciera que hemos terminado el algoritmo aún nos falta una demostración más. Nos falta poder demostrar que el programa efectivamente termina. Aquí es donde entra el concepto de variante.

Una **variante** es una **expresión que nos ayuda a demostrar que un programa termina**. Para que una variante sea útil debe representar un entero con una cota inferior y en cada iteración del *loop* debe tomar un valor menor a lo que era previamente. Lo más común es que la variante comience siendo un entero positivo y que su cota inferior sea cero, aunque nada impide que se tomen otros valores.

En este caso tomaremos como variante N-i. Al inicializar *i* en 0 la variante comienza valiendo N, y luego en cada iteración *i* aumenta su valor en uno, al tiempo que N permanece constante, hasta que al salir del *loop* la variante valdrá 0, ya que *i* habrá tomado el valor de N. Sabemos los rangos entre los que se mueve *i* debido a nuestra invariante limitante.

Variante: N - i

A medida que el *loop* avanza, la invariante limitante nos ayuda a verificar que las variables están tomando los valores esperados y que el algoritmo está avanzando, y la variante nos muestra que el algoritmo finaliza. Esta relación conceptual entre variantes e invariantes también se ve reflejada en código: la variante indefectiblemente refiere a una o más variables que también son referidas en alguna de las invariantes del problema.

Para expresar la variante usamos la expresión *variant* seguido de la expresión de la misma.

```
method miembroDelArray (T [ ] a, T e): boolean esta {
        requires: -
        ensures: esta es true si y sólo si el elemento e se encuentra en el array a.
        loop{
                invariant: esta es true si y sólo si e pertenece a a[0..i-1]
                invariant: 0 ≤ i ≤ N
                initialization: esta, i = false, 0;
                termination: i == N;
                variant: N – i;
                progress{
                        esta, i = esta || a[i] == e, i+1;
                }
        }
}
```

Primeras impresiones
Hemos derivado nuestro primer algoritmo y demostrado su corrección. En los siguientes capítulos veremos distintos tipos de problemas resueltos mediante esta técnica y utilizando el mismo proceso de derivación. Además veremos distintas técnicas para derivar las invariantes.

Existen lenguajes que incluyen notaciones para representar invariantes, variantes, pre y postcondiciones, y demás. Lenguajes como *Dafny* [6] incluso tienen el poder de verificar la corrección de lo expresado en tiempo de compilación. Sin embargo, la mayoría de los lenguajes no lo tienen, por lo que debemos expresar estas nociones como comentarios incrustados en el código. En este caso los programadores somos los responsables de la corrección de las anotaciones que proponemos.

Proceso de derivación
Es necesario admitir que parece haber un poco de 'magia' en este proceso. ¿De dónde sale la noción de que necesitaremos recorrer el *array* con un índice y usar una variable auxiliar? Es muy trivial para un caso como éste, pero ¿Qué ocurre en general? ¿Se supone que las ideas caerán del cielo?

La respuesta es que no. No caerán del cielo. Ni por arte de magia. Lo que acabamos de hacer es una derivación de un problema siguiendo un patrón, o línea de razonamiento. El mismo razonamiento se puede aplicar sistemáticamente para otro tipo de problemas, pero hay una parte que es propia del dominio de cada uno.

Los pasos son:
1) Analizar el dominio del problema y especificar en términos de pre y post condiciones.
2) Definir las invariantes.
3) Definir la inicialización de las variables, la condición de terminación del *loop*, el progreso y la variante.

Estos pasos pueden ser iterados de ser necesario. Quizás la invariante elegida no aporta nada para acercarnos a la solución del problema, o quizás hemos especificado mal el problema. En ese caso es correcto ir hacia atrás y volver a pensar cada paso.

Para facilitar este proceso de resolución de problemas es que presentamos **métodos de solución de problema**, y **técnicas de derivación de invariantes**. Lo que hacemos en el tercer paso está fuertemente ligado a las invariantes que derivamos en el paso 2: cada invariante tiene asociadas la condición de terminación, la inicialización y el progreso, que son los que le dan sentido. Pueden diseñarse en cualquier orden, pero cada uno cumple un rol específico: la condición de terminación nos asegura que al final del *loop* alcanzamos la postcondición, la inicialización nos asegura que podemos verificar las invariantes antes de comenzar el *loop*, y el progreso nos asegura que avanzamos, mientras seguimos verificando las invariantes.

En la programación basada en invariantes la verificación de que el código preserve las invariantes se hace continuamente y forma parte del proceso de codificación, a diferencia de la programación tradicional donde el aseguramiento del cumplimiento de algunas restricciones forma parte del proceso de *testing*. Por lo tanto la tarea de las verificaciones no forma parte de ningún paso en

específico, sino que se da a lo largo de todo el proceso en el que siempre debemos asegurarnos que se cumplen estas restricciones [9]. Si bien derivar las invariantes puede implicar un esfuerzo adicional, su aplicación ayuda a una mayor comprensión del problema, lo que deriva en una mayor confianza en el código desarrollado.

El proceso que hemos mostrado es una buena forma de acercarse a la solución de un problema en situaciones donde no sabemos cómo encarar el mismo.

En los ejemplos siguientes vamos a repetir esta metodología para demostrar de forma práctica cómo efectivamente se pueden llegar a soluciones de algoritmos siguiendo estos pasos. Además contamos con dos métodos para solucionar problemas y una serie de técnicas para derivar invariantes a partir de las postcondiciones. El método nos dirá qué es lo que estamos haciendo, mientras que la técnica nos dirá el cómo.

Updating

Al método utilizado para resolver el problema recién visto lo llamamos ***Updating***. Este método consiste en iterar sobre algún tipo de estructura de datos o conjunto de valores, llevando un acumulador o resultado parcial que se va actualizando en cada paso del *loop* hasta que se agota la estructura a recorrer. El acumulador es en ese momento el resultado final. Le llamamos ***Updating*** porque lo que hacemos es calcular el resultado parcial y almacenarlo de forma temporal, por lo que debemos mantenerlo actualizado en cada paso. En general el resultado parcial representa lo que denominamos una "postcondición debilitada", es decir, expresa que la postcondición se cumple, pero sólo en la porción ya analizada de la estructura.

La esencia del método consiste en avanzar en la recorrida de la estructura o el conjunto de datos y procesar una parte del mismo a medida que avanzamos. Este método aplica muy bien para las recorridas en estructuras como *array*s (que representan la mayoría de las que veremos), pero el concepto aplica para conjuntos de enteros también. Nuestro resultado final se obtiene en el momento que hemos completado la recorrida de toda la estructura o conjunto de datos.

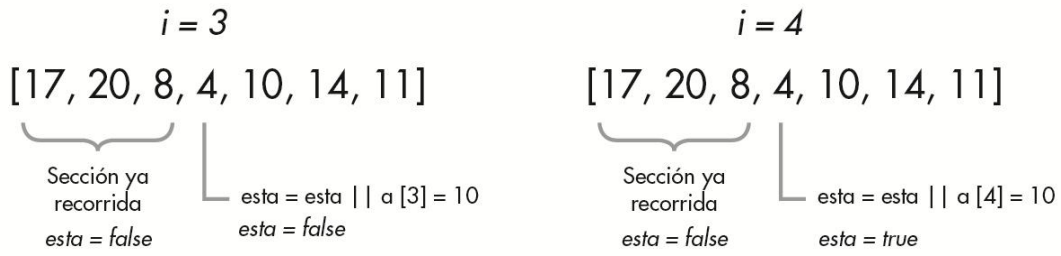


Imagen 2-2: Aplicación de Updating buscando el número 10

En este caso la variable esta representa el resultado parcial que nos dice si el elemento había sido encontrado hasta el momento o no, y es el resultado final del problema a resolver al finalizar la recorrida.

2.2. Menor Divisor

El problema que presentamos a continuación es: dado un número natural n mayor o igual a 2, encontrar el menor número que lo divida.

Precondición: $n \geq 2$

Postcondición: i es el menor natural, mayor o igual que 2, que divide a n

Para encontrar la solución al problema vamos a recorrer los números naturales empezando desde 2 hasta hallar un natural que divida a n (parámetro). Por ejemplo, asumamos que recibimos el número 25. Empezaremos desde el número 2. 2 no divide a 25, porque al hacer la división nos queda un resto. Ahora intentamos con 3, tampoco resulta. Luego con 4, aún sin éxito. Luego el 5, que sí divide a 25, porque dividir 25 entre 5 resulta 5, sin resto alguno, por lo que hemos encontrado la solución al problema.

Para derivar la invariante vamos a descomponer la postcondición, de la siguiente manera:

1. Ningún número en el rango [2..i-1] divide a n.

2. i divide a n. En términos matemáticos: n % i==0

El rango [2..i-1] representa los elementos ya recorridos y descartados.

Separar la postcondición en partes nos permite utilizar una de ellas como invariante, y la otra como condición de terminación del *loop*. En este caso la primera de las condiciones es la invariante y la segunda la condición de terminación.

El *loop* termina cuando se verifica la condición de terminación, y además sabemos que la invariante se cumplirá, si hemos seguido las reglas correctamente. Por lo tanto con ambas condiciones cumpliéndose sabemos que hemos arribado a la postcondición y que el algoritmo es correcto. La inicialización es trivial (asignamos i=2), y verifica la invariante, al igual que el paso del *loop*, en el que incrementamos *i* en una unidad, ya que seguir en el *loop* significa que aún no hemos encontrado el primer divisor de n.

Hasta aquí tenemos:

Invariante 1: ningún número en el rango [2..i-1] divide a n

Inicialización: i = 2;

Terminación: n % i == 0;

Progreso: i = i + 1;

Aún nos falta la invariante limitante y la variante.
La variable *i* puede tener como valor mínimo 2 y como máximo n, de aquí podemos desprender que:

Invariante 2: $2 \leq i \leq n$

Variante: n-i

Sabemos que la variante disminuye en cada iteración ya que n permanece constante, al tiempo que *i* aumenta. La cota inferior de la variante es 0, valor que alcanzará solamente si n es un número primo (sólo es divisible entre 1 y sí mismo). En caso contrario, existe un divisor entre 2 y n-1, por lo que el algoritmo terminará antes.

La implementación de este algoritmo es:

```
method primerDivisor(nat n) : nat i {
        requires: n≥2
        ensures: i es el menor número, mayor o igual a 2, que divide a n
        loop{
                Invariant: ningún número en el rango [2..i-1] divide a n
                Invariant: 2 ≤ i ≤ n
                Initialization: i=2;
                Variant: n - i
                Termination: n % i == 0
                progress{
                        i = i+1;
                }
        }
}
```

Reducción

Lo que acabamos de hacer es la primera aplicación de nuestro segundo método: **Reducción**. El objetivo es reducir el cálculo final requerido por la postcondición al cálculo en una porción en el conjunto de valores posibles. En otras palabras, a medida que avanzamos en la ejecución sabemos que el resultado está en la parte que nos falta procesar, y que no está en lo ya procesado, lo que nos permite ir descartando valores del conjunto original de posibles soluciones hasta que eventualmente damos con la solución al problema o el conjunto que analizamos se agota. Esto nos permite finalizar la ejecución al momento de hallar la solución y no tener que seguir procesando.

3.Algunos algoritmos fundamentales

En este capítulo veremos algunas soluciones de problemas muy elementales y recurrentes. Buscamos brindar los primeros ejemplos de aplicación de la programación basada en invariantes utilizando siempre el proceso propuesto en la sección introductoria de forma metódica. Además introducimos las técnicas de derivación de invariantes que usaremos a lo largo del trabajo.

3.1. Suma de *Array*

Este problema consiste en hallar la suma de todos los elementos de un *array* que recibimos como parámetro. Lo único que sabemos es que el *array* está compuesto por enteros. Fácilmente podemos darnos cuenta que la derivación del problema será bastante intuitiva y similar al ejemplo de la pertenencia al *array* que vimos anteriormente.

La postcondición del problema es que el retorno (llamémosle *suma*) tiene que ser la suma de todos los elementos de a, y no hay ninguna precondición.

Postcondición: suma es la suma de todos los elementos de a

De la misma forma que el problema de la pertenencia a un *array* lo vamos a resolver mediante ***Updating***. Lo que haremos es utilizar la variable *suma*, como resultado parcial del *array* a medida que lo recorremos y una variable *i* que actúa como índice para recorrer el *array* y que representa la siguiente posición del *array* a visitar. Esta será además nuestra invariante esencial.

Invariante 1: suma es la suma de todos los elementos de a[0..i-1]

También debemos definir los valores que puede tomar el índice para poder recorrer el *array*. Esta será nuestra segunda invariante, que es del tipo limitante.

Invariante 2: $0 \leq i \leq N$

Sabemos que la variable *suma* contiene la suma de todos los elementos de a[0..i-1], por lo tanto si sumamos el contenido de a[i] esa variable contendrá el valor de la suma de todos los elementos de a[0..i]. Luego podemos aumentar *i* y mantener la invariante 1. En cada iteración del *loop* sumaremos el contenido de a[i] a la variable *suma* y aumentamos el índice en uno, hasta que eventualmente llegaremos a un resultado para todo el *array*.

El índice *i* comenzará siendo 0 y la variable *suma* también, ya que este es el resultado de la suma de los elementos del *array* vacío y por lo tanto verifica las invariantes antes de ingresar.

Initialization: i ,suma = 0, 0;

Termination: i == N;

progreso: suma, i = suma + a[i], i+1;

El algoritmo terminará cuando el índice ha tomado el valor equivalente al largo del *array*. Por lo tanto si i==N no se ejecutarán las instrucciones del *loop* y el valor de *i* no podrá aumentar más, verificando así la invariante 2.

Dado que la variable *i* incrementa en cada ejecución de las instrucciones del *loop* podemos derivar una variante de la forma:

Variant: N - i;

El algoritmo resultante sería algo del estilo:

```
method sumaArray(int[ ] a) : int suma {
        ensures: suma es la suma de todos los elementos de a
        loop{
                invariant: suma es la suma de todos los elementos de a[0..i-1]
                invariant: 0 ≤ i ≤ N
                initialization: i ,suma = 0, 0;
                termination: i == N;
                progress{
                        suma, i = suma + a[i], i+1;
                }
        }
}
```

Acabamos de ver una aplicación de la primera **técnica** para derivar invariantes: **Relajación de constantes**. Esta técnica consiste en reemplazar una constante *N* que no será alterada durante el transcurso del algoritmo por una variable *i* que irá cambiando de valor en cada paso del *loop*, hasta alcanzar i==N como condición de terminación [4] [7].

Por lo tanto, para el caso de un *array*, si tenemos una postcondición que nos exige que se cumpla una condición S para un *array* A, la transformamos en una invariante que nos dice que S se cumple para el *array*[0..i] y otra que nos dice entre que valores se mueve *i* (normalmente entre 0 y N). Cuando *i* alcance el valor de N, S se cumplirá para el *array*[0..N-1], es decir, S se cumple para el *array* A en su totalidad.

En este caso hemos reemplazado la expresión "*suma* es la suma de todos los elementos de a" por "*suma* es la suma de todos los elementos de a[0..i-1]" y el algoritmo terminará cuando *i* alcance

el valor de N. En ese momento la variable *suma* representará la suma de todos los elementos del *array*.

Al aplicar esta técnica obtenemos, al menos, una invariante limitante y una esencial. En este caso la invariante esencial es la 1 y la limitante la 2. Adicionalmente, en base a la invariante limitante pudimos derivar también nuestra variante.

En este ejemplo podemos apreciar las diferencias entre métodos y técnicas. El método ***Updating*** es más abstracto, y con él llegamos a la idea de llevar un acumulador que almacene sumas parciales, y con la técnica **Relajación de constantes** derivamos las invariantes del problema a partir de la postcondición.

Se puede notar una gran similitud con el ejemplo introductorio de pertenencia a un *array*. Es que precisamente es la misma técnica que usamos en aquel ejemplo. En ese caso el estado de la variable esta representaba el resultado parcial que nos dice si el elemento había sido encontrado hasta el momento o no, que es la invariante esencial, y el rango en el cuál se mueve el índice estaba representado en la invariante segunda invariante: la limitante.

Ahora usted podría preguntarse: ¿Por qué si iteramos sobre *i* la invariante esencial (*suma* es la suma de todos los elementos de a[0..i-1]) se establece sobre i-1? Esto se debe a la aplicación de un ajuste llamado ***Aging*** [7] que aplicamos para forzar la validez de la invariante al inicializar el *loop*. Dado que inicializamos las variables *i* y *suma* en 0, si la invariante se estableciera sobre a[0..i], al entrar al *loop* por primera vez estaríamos diciendo que *suma* es igual a la suma de todos los elementos de a[0..0], ósea que es igual a a[0], pero no sabemos si esto es cierto. Este pequeño desajuste puede arreglarse aplicando ***Aging*** como lo hicimos, o también podríamos inicializar *suma*= a[0], pero esto nos obligaría a agregar una precondición que diga que el *array* tiene al menos un elemento, por lo que estaríamos sobre especificando el problema, y además dentro del *loop* deberíamos cuidar de no sumar el primer elemento otra vez.

3.2. Producto de *Array*

Ahora imaginemos que queremos usar la misma estructura que ya sabemos es correcta para calcular el **producto** de todos los elementos de un *array*. Fácil: cambiemos la operación suma por producto y el nombre de la variable *suma* por *prod*.

Sin embargo, este algoritmo no es correcto. El problema está en la inicialización de las variables.

Estamos utilizando la operación producto, por lo que si alguno de los dos operandos es cero el resultado será también cero. Si inicializamos la variable *prod* en cero el resultado de todas las operaciones que se realizan será cero, así como el resultado final del algoritmo.

Lo correcto sería inicializar el acumulador en 1, debido a que es la identidad de la operación producto y verifica la invariante, ya que por convención el producto del conjunto vacío es 1. Este ejemplo sirve para ilustrar la importancia de la inicialización de las variables para asegurar la corrección de un algoritmo. Para evitar este tipo de problemas es importante que al verificar la

corrección de las invariantes que hemos definido lo hagamos teniendo en cuenta las inicializaciones tanto cuando validamos las invariantes antes de entrar en el *loop* como en una iteración cualquiera.

No daremos una implementación de este algoritmo sino que lo dejamos como tarea para el lector.

3.3.Multiplicación usando suma

Recordemos como nos enseñaron a multiplicar cuando éramos niños: "Multiplicar cinco por cuatro es sumar cuatro veces cinco. Si lo sumamos una vez, tenemos cinco. Sumamos por segunda vez, tenemos 10. Tres veces, 15. Sumamos por cuarta vez y tenemos 20. Luego, cuatro por cinco es 20".

Este cuento que nos hicieron para aprender a multiplicar lo vamos a replicar en este algoritmo, que consiste en multiplicar un entero *a* por un natural *b* sin utilizar el signo de multiplicación.

Por lo tanto nuestra postcondición es: $r == a \times b$

Este enfoque de la multiplicación es un caso más de ***Updating*** donde el dominio del problema no es un *array*, sino operaciones matemáticas sobre enteros, y la variable *r* actúa como acumulador. Basándonos en el ejemplo, si cortáramos la ejecución del algoritmo en la segunda iteración tendríamos el resultado de computar 5 por 2. Es decir, el acumulador funciona como una solución parcial del producto entre a y b.

Para comenzar vamos a aplicar nuevamente la técnica de **Relajación de constantes** para derivar la invariante. Reemplazamos entonces la constante *b* por una variable *x*, cuya función será llevar la cuenta de cuántas veces hemos sumado *a*, y la variable irá incrementando hasta que alcance el valor de *b*, momento en el cual habremos alcanzado la postcondición. Esto nos llevará a la siguiente invariante esencial:

$$Invariante\ 1:\ r == a \times x$$

y la limitante:

$$Invariante\ 2:\ 0 \leq x \leq b$$

Dado que al comenzar el *loop* no habremos computado nada aún, debemos inicializar en 0 tanto *r* (resultado acumulado), como *x* (cantidad de veces que hemos sumado *a*).

$$Inicialización:\ r,\ x=0,\ 0;$$

La inicialización de las variables debe verificar las invariantes del problema, y vemos que efectivamente lo hace. La invariante 1 verifica, ya que $0 = a \times 0$. La segunda invariante también verifica ya que con $x=0$ cumple con su cota inferior y *b*, al ser un número natural, va a ser mayor o igual a 0.

En el paso del *loop*, debemos incrementar el resultado acumulado sumándole el valor de *a*, y también incrementar x en uno para actualizar la cuenta y validar las invariantes.

Paso del loop: x, r= x+1, r+a;

Veremos si con este paso se siguen verificando las invariantes: si al entrar al *loop r* valía $a \times x$ y *x* se incrementa en 1, el nuevo valor de r debe ser $a \times (x+1)$, lo que es igual a $a \times x + a$, y como $a \times x$ era el valor original de *r*, la nueva asignación de *r* se corresponde a $r + a$. Así vemos que se verifica la invariante esencial.

Debemos repetir este paso *b* veces, por lo que la condición de terminación del *loop* será:

Terminación: x == b

En cuanto a la invariante limitante, sabemos que si estamos dentro del *loop* es porque *x* es menor que *b*, por lo que con un incremento en uno, se mantendrá dentro de su cota superior: $x \leq b$ **.**

La variante es *b - x*, que decrece en cada iteración ya que *b* permanece constante y *x* se incrementa en una unidad en cada paso, y su cota inferior es 0, ya que el *loop* termina cuando *x == b*.

Variante: b - x

Dado que el algoritmo termina (lo demostramos mediante la variante), y que lo hace cuando x == b, podemos reemplazar x por b en la invariante esencial y verificamos que r == a × b.

```
method ProductoUsandoSuma(int a, nat b) : int r
{
        ensures r == a × b;
        loop{
                invariant: r = a × x;
                invariant: 0 ≤ x ≤ b
                initialization:x, r= 0, 0;
                termination: x == b;
                variant: b - x
                progress{
                        x, r= x+1, r+a;
```

```
            }
        }
}
```

3.4.Potencia

Luego de haber introducido el método ***Updating*** y la técnica **Relajación de constante,** y haberla utilizado en varias ocasiones, confiamos que el lector ya tiene la capacidad para derivar invariantes mediante dicha técnica, por lo que dejamos como ejercicio el siguiente problema: Dados dos enteros a y b, desarrollar una función utilizando la misma nomenclatura de los problemas anteriores, que retorne el resultado de a^b. Se debe utilizar el método ***Updating*** y se recomienda la utilización de la técnica de **Relajación de constantes**. Se deben incluir todas las invariantes que sean necesarias, tanto esenciales como limitantes.

3.5.División entera

El objetivo del problema es, dados dos naturales *m* y *n* (con $n > 0$), hallar el cociente *q* y el resto *r* a partir de dividir *m* sobre *n*. De la misma forma que el problema de la multiplicación usando suma la complejidad está en resolverlo sin usar el operador de división o resto, sino que solamente podemos usar sumas y restas.

Las postcondiciones del problema son 3, que salen de la definición de división entera:

$$1)\ \ m == q \times n + r$$

$$2)\ \ 0 \leq r$$

$$3)\ \ r < n$$

La primera postcondición establece la definición de la división entera, y las siguientes marcan propiedades que debe cumplir el resto de la división: no puede ser negativo y debe ser menor que el divisor. La segunda condición la vamos a introducir en el retorno, estableciendo que el resto debe ser natural.

La única precondición es que n no puede ser cero, porque como todos sabemos, no se puede dividir entre ese número. Por lo tanto:

$$Precondición: n>0$$

Este clásico problema de la aritmética nos sirve para aplicar nuevamente el segundo método: **Reducción**. A diferencia de ***Updating***, en este tipo de soluciones sabemos que el resultado está en la parte aún por computar, y que el resultado de lo ya computado no tendrá valor matemático alguno. El espacio a recorrer son los parejas $\{q, r\}$ que verifican $m == q \times n + r$.

Si en algún momento cortamos la ejecución del algoritmo antes de llegar a la condición de terminación, *q* y *r (*cociente y resto*)* no van a tener ningún significado, es decir, no van a representar el cociente ni el resto de ninguna operación parcial del problema, a diferencia de lo que sucede con el método ***Updating*** donde si detenemos la ejecución el resultado parcial tiene un significado: normalmente es la misma postcondición debilitada, es decir, aplicada a una porción del conjunto de los valores a analizar.

Para resolver este problema vamos a derivar las invariantes mediante una técnica denominada **Eliminación de una condición**. Consiste en comenzar con un "proyecto de solución" o "solución a medias", que cumple con parte de las postcondiciones pero no con todas, e ir transformando esta solución parcial hasta cumplir con todas las postcondiciones [4] [7].

Concretamente lo que hacemos es tomar algunas de las postcondiciones y utilizarlas como invariantes, y el resto como condiciones de terminación del *loop*. En cada paso del *loop* debemos preservar las condiciones que establecimos como invariantes (además de validarlas al momento de la inicialización), y además acercarnos a las condiciones restantes que establecimos como terminación del *loop*. Al salir del *loop* se cumplirán tanto las invariantes como las condiciones de terminación, y éstas en conjunto componen la postcondición. Es evidente que para aplicar esta técnica debemos estar frente a un problema con más de una postcondición, o una postcondición que pueda ser descompuesta en varias condiciones.

Utilizaremos las condiciones 1 y 2 como invariantes y la 3 como condición de terminación del *loop*:

$$\textit{Invariante 1: } m == q \times n + r$$

$$\textit{Invariante 2: } 0 \leq r$$

$$\textit{Termination : } r < n$$

Establecemos como valores de inicialización q=0 y r=m, por lo que se verifican las invariantes, si sustituimos las variables por los valores iniciales.

$$\textit{Inicialización: } q, r = 0, m;$$

Supongamos que no se cumple aún la condición de terminación, y estamos en un paso cualquiera del *loop*. En cada paso de la ejecución del *loop* el cociente *q* se incrementará en una unidad al tiempo que el resto *r* es actualizado por su nuevo valor *r-n*, hasta que *r* sea menor que *n*, que es la condición de terminación. En ese momento, el resto tendrá un valor válido, y *q* tendrá el valor de la cantidad de veces que "entra" *n* en *m*.

Para una iteración cualquiera del *loop* la invariante esencial ($m == q \times n + r$) se preserva ya que al entrar al *loop* era válida, y en cada paso del *loop* q aumenta en una unidad, por lo que el nuevo valor de m pasa a ser:

$$m = (q+1).\ n + r$$

$$m = q.n + n + r$$

Para mantener m de manera de preservar la invariante, el nuevo valor de r debe ser r-n, y así tendríamos:

$$m = q.n + n+(r - n)$$

$$m = q.n + r$$

Así demostramos la expresión $q \times n + r$ se mantiene constante, y es igual a *m*. En cuanto a la invariante limitante, al entrar al *loop* necesariamente *r* era mayor o igual a *n*, por lo que al restarle el valor de *n*, valdrá como mínimo 0, preservando así la invariante $0 \leq r$.

Paso del loop: q, r:=q+1, r-n;

La variante será el resto *r*, que disminuye en cada iteración, como podemos apreciar en el paso del *loop* (*r = r-n*), ya que *n>0* (precondición), y sus cotas superior e inferior son *n-1* y *0* respectivamente.

Variante: r

Hemos mantenido dos de las tres postcondiciones mediante invariantes, y hemos llegado a la tercera al finalizar el *loop*. Además al demostrar que el algoritmo termina tenemos nuestro algoritmo de división entera completo y correcto. Veámoslo:

```
method DivisionEntera (nat m, nat n): nat q, nat r
        requires n>0
        ensures m == q×n + r
        ensures 0 ≤ r
        ensures r<n
        loop{
                invariant m == q×n + r
                invariant r ≥ 0
                initialization: q, r = 0, m;
                termination: r<n
                variant: r
                progress{
                        q, r = q+1, r-n;
                }
        }
}
```

3.6.Raíz cuadrada

Vamos a calcular la parte entera de la raíz cuadrada de un número natural *n*. ¿Qué quiere decir esto? Para poner un ejemplo: si la raíz cuadrada de 10 es aproximadamente 3,162, su parte entera es 3, y éste es el valor que queremos retornar.

En términos aritméticos, dado un natural *n*, queremos hallar otro natural m tal que:

$$m^2 \leq n < (m+1)^2$$

Esa es la postcondición del problema, al tiempo no hay precondiciones ya que la raíz cuadrada existe para cualquier número natural.

Este problema también lo vamos a resolver mediante **Reducción**. Conceptualmente se resuelve de forma similar al problema del resto y el cociente que acabamos de ver, donde no obtendremos resultados parciales semánticamente significativos sino que en cada paso vamos descartando valores, y la solución al problema tendrá validez recién una vez finalizada su ejecución.

Dado que las postcondiciones son dos (descomponiendo la desigualdad), vamos a aplicar la técnica **Eliminación de una condición**, tomando como invariante $n < (m+1)^2$ y como condición de terminación del *loop* $m^2 \leq n$. De esta manera sabremos que al salir del *loop* (siempre y cuando se mantengan las invariantes correctamente) vamos a cumplir con ambas postcondiciones y habremos resuelto el problema.

Invariant: $n < (m+1)^2$

Termination: $m^2 \leq n$

Ahora pensemos un valor apropiado para inicializar m, tal que de antes de entrar al *loop* se establezca la invariante. Aquí vamos a separar dos casos: por un lado los casos particulares (cuando n vale 0 o 1) y por otro el resto de los casos.

Comenzando por los casos particulares, probemos inicializando *m=n*. Tanto para 0 como para 1 se verifica la invariante. Para *n=0* obtenemos *0 < 1*, y para *n = 1* la invariante establece $1 < 2^2$. En estos casos el *loop* no va a ejecutarse ni una vez ya que de entrada se estaría verificando la condición de terminación, y el retorno sería igual a *n*, lo que es correcto ya que la raíz cuadrada de 0 es 0, y de 1 es 1.

Ahora pensemos un valor apropiado para inicializar *m* cuando n es mayor o igual a 2. Ya que el menor valor posible de *n* es 2, probemos inicializar $m = \frac{n}{2}$. En el caso que n valga 2, *m* valdrá 1, el

algoritmo no va a entrar al *loop* ya que se cumpliría la condición de terminación inicialmente, y el retorno sería 1, lo que equivale a la parte entera de la raíz cuadrada de 2.
Podemos observar que la inicialización m = $\frac{n}{2}$ (división entera) es válida ya que la raíz cuadrada de cualquier natural es menor o igual que su mitad: la parte entera de la raíz cuadrada de 2 y de 3 es 1, de un natural entre 4 y 8 es 2, etc. Para este caso podríamos empezar también con *m=n,* pero sabiendo esta propiedad, nos ahorramos n/2 iteraciones innecesarias.

Inicialización: if (n<2) m=n else m = $\frac{n}{2}$

Esta inicialización, ¿establece la invariante? Veamos:

Invariante: $n < (m+1)^2$

$$n < (\frac{n}{2}+1)^2$$

$$n < (\frac{n+2}{2})^2$$

$$n < \frac{n^2}{4} + n + 1$$

Vemos que efectivamente se verifica la invariante.

En cada paso iremos disminuyendo m en una unidad hasta alcanzar la condición de terminación. La variante será *m* (podemos ver explícitamente que decrece en cada paso), y su cota inferior será justamente la parte entera de la raíz cuadrada de *n*. Sabemos que el algoritmo termina porque la raíz cuadrada existe para cualquier natural.

Variante: m

Paso del loop: m = m-1;

Analicemos si el paso del *loop* verifica la invariante. Al entrar al *loop* sabemos por la invariante que n < $(m+1)^2$ y como no se ha alcanzado la condición de terminación también sabemos que n < m^2.
Al decrementar m en 1, la expresión de la derecha en la invariante ($(m+1)^2$) pasa a valer m^2, que como vimos por la segunda desigualdad es mayor a n, luego $(m+1)^2$ también será mayor que n verificando así la invariante.

```
method RaizCuadrada(nat n): nat m
{
        ensures m² ≤ n < (m + 1)²
```

```
        loop{
                invariant n < (m + 1)^2
                initialization: if (n < 2) m=n else m = n/2
                termination m^2 ≤ n
                variant m
                progress{
                        m = m-1;
                }
        }
}
```

3.7. Logaritmo

Pasemos a otro problema aritmético: el cálculo del logaritmo.
Recordemos la definición de esta operación:
Si los números a y b son positivos, b (la base) diferente a 1, se dirá que el logaritmo de a en base b es h si se cumple que

$$log_b\, a = h \leftrightarrow b^h = a$$

Ya que estamos trabajando con enteros, vamos a exigir que la base del logaritmo sea un número entero, y el resultado también será la parte entera del logaritmo, como hicimos con la raíz cuadrada, por lo que la postcondición será:

$$b^h \leq a < b^{h+1}$$

Por ejemplo, si quisiéramos hallar el logaritmo de 9 en base 3, el resultado sería 2, ya que $3^2 = 9$, y si quisiéramos hallar el logaritmo de 10 en base 3, también obtendremos 2 como resultado, ya que:

$$3^2 \leq 10 < 3^3$$

Exigiremos que la base sea mayor a 1 y *a* mayor a cero por definición de logaritmo, por lo que las precondiciones son:

Precondición: b >1

Precondición: a >0

De la misma forma que los dos ejemplos anteriores vamos a utilizar **Reducción** para resolver el problema. La derivación de la invariante es muy sencilla y a la vez muy útil para reforzar la técnica de **Eliminación de una condición,** que aplicamos recientemente para resolver el problema de la división entera y raíz cuadrada. Dado que la postcondición puede ser descompuesta en dos condiciones:

$$1)\ b^h \leq a$$

$$2)\ b^{h+1} > a$$

Vamos a tomar la primera de ellas como invariante del *loop*, y la segunda como condición de terminación:

$$Invariante\ 1:\ b^h \leq a$$

$$Terminación:\ b^{h+1} > a$$

La invariante $b^h \leq a$ es la invariante esencial de este problema.
La invariante limitante establece la cota inferior de h en 0, que es el menor valor que esta variable puede tomar:

$$Invariante\ 2:\ h \geq 0$$

Vamos a inicializar *h* en 0, su valor mínimo, ya que cualquier número elevado a la 0 es 1, y *a* debe valer al menos 1, con lo que se verifican las invariantes.

$$Inicialización:\ h=0;$$

Luego, iremos incrementando *h* en cada iteración, hasta que el programa verifique que con un incremento más nos estaríamos pasando del valor deseado:

$$Paso\ del\ loop:\ h= h+1;$$

Debemos comprobar que el paso del *loop* verifica las invariantes. Obviemos la limitante que es demasiado trivial. Veamos la esencial. Al entrar a un paso del *loop* sabemos que $b^h \leq a$, y como no se cumple aún la condición de terminación también deducimos que : $b^{h+1} > a$. Luego, al incrementar h en uno, b^h pasa a valer b^{h+1}, que como vimos es menor o igual que a, verificando así la invariante esencial.

La variante es a - b^h. Sabemos que esta expresión decrece a cada iteración dado que *h* se incrementa en 1 en cada paso mientras a y b permanecen constantes. Su cota inferior es 0, que será alcanzada en caso que *h* sea exactamente el logaritmo de a en base b.

$$Variante: a - b^h$$

```
method LogaritmoEntero (int a, int b): int h
{
        requires b >1
        requires a >0
        ensures b^h ≤ a <  b^(h+1)
        loop
        {
                invariant: b^h ≤ a
                invariant: h ≥ 0
                initialization: h = 0;
                termination: b^(h+1) > a
                variant: a - b^h
                progress
                {
                        h= h+1;
                }
        }
}
```

3.8.Máximo común divisor

Dados dos naturales positivos *a* y *b*, el Máximo común divisor es el mayor entero que divide a ambos.

Para resolver este problema podríamos iterar sobre los enteros, comenzando en el 1 y terminando como máximo en *a* o *b*, el mínimo de ellos. Adicionalmente llevaríamos una variable que guarda el resultado parcial de la porción de enteros ya recorridos, y al encontrar un entero que sea común divisor asignaremos este valor al resultado parcial. Esto sería un típico acercamiento mediando *Updating*, que no es la solución más eficiente, por lo que vamos a aplicar una estrategia distinta.

Para resolver el problema de forma más eficiente vamos a utilizar el método de **Reducción** gracias a la implementación iterativa del algoritmo de *Euclides*. Este algoritmo nos dice que:

1) Dados dos naturales x e y, tal que x > y, se cumple que mcd (x,y) = mcd (x-y, y).

Lo que haremos es reducir el problema de calcular el MCD entre a y b al cálculo del MCD entre r y s, para r y s menores o iguales que a y b respectivamente.

Para resolver este problema nos vamos a basar en esta propiedad, y en otra que establece que el máximo común divisor entre un número y sí mismo, es este mismo número:

2) mcd (r,r) = r.

La técnica de derivación que vamos a utilizar se llama ***Decoupling***, o **Desacoplamiento**, que consiste en reemplazar una variable (normalmente el retorno) por otras dos variables (retorno y *x*), utilizando la igualdad entre ellas como condición de terminación del *loop* o como parte de la misma [7]. En este problema podemos aplicar *esta* técnica gracias a la propiedad 2 del máximo común divisor.

Basándonos en estas dos propiedades matemáticas y aplicando **Desacoplamiento**, el camino que vamos a tomar para resolver el problema será el siguiente: utilizaremos dos variables auxiliares *r* y *x* tales que verifiquen la invariante mcd (r,x) = mcd (a,b).

Invariante: mcd (r,x) = mcd (a,b)

Para ello vamos a inicializar *r* y *x* con los valores de *a* y *b* respectivamente, y así validamos la invariante antes de ingresar al *loop*. El *loop* debe terminar necesariamente cuando *r* sea igual a *x*, ya que allí, gracias a la propiedad 2 y a la verificación de la invariante, vamos a poder decir que *r* es el máximo común divisor entre *a* y *b*.

Inicialización: r, x = a, b;

Terminación: r == x;

En cada paso del *loop* aplicaremos el Algoritmo de Euclides para así poder acercarnos a la solución y al mismo tiempo que la invariante se siga cumpliendo:

Paso del loop: if (r > x) r = r - x;

else x = x - r;

Ya tenemos la invariante esencial, la inicialización de las variables, la condición de salida, y el paso del *loop*. Aún nos falta una variante para demostrar que el *loop* termina.
Veamos: en cada paso del *loop*, o bien decrece *r*, o lo hace *x*. La que decrece es la máxima entre ambas variables en cada caso, por lo tanto la variante es:

Variante: max (r,x)

Su cota inferior es efectivamente el máximo común divisor de a y b, que es el valor que tendrán finalmente tanto *r* como *x*.

```
method MaximoComunDivisor(nat a, nat b): nat r
{
        ensures r = mcd (a,b)
```

```
        requires a > 0
        requires b > 0
        loop{
                invariant: mcd (r,x) = mcd (a,b)
                invariant: r > 0
                invariant: x > 0
                initialization: r, x = a, b
                termination: r == x
                variant: max (r,x)
                progress{
                        if (r > x) r = r - x;
                        else x = x - r;
                }
        }
}
```

Este es, quizás el ejemplo más claro de cómo mediante el método de Reducción reducimos el resultado final (*mcd(a,b)*) al resultado de la estructura remanente *mcd(r,x).*

4.Búsquedas

Las búsquedas en una estructura son de los conceptos más importantes que se deben adquirir. Constantemente es necesario realizar búsquedas, y optimizarlas es una de las tareas más desafiantes. Veremos a continuación los dos tipos de búsquedas más básicos: la **búsqueda lineal** y la **búsqueda binaria**. Aplicaremos la metodología definida para derivar los algoritmos así como demostrar su corrección e idealmente el uso de las invariantes facilitará la comprensión de los mismos.

Cuando hablamos de una búsqueda nos referimos a encontrar un elemento que cumple determinada condición dentro de un conjunto. Es similar al primer ejemplo, pero en ese caso estábamos determinando si había algún elemento en el *array* que cumpla la condición definida: pertenecer a esa estructura. En este caso lo que queremos saber es la posición del elemento que cumple la condición, por lo que asumimos que el elemento pertenece al *array*. Por lo tanto nuestra precondición es:

$$\textit{Precondición: } e \in a$$

Independientemente del método de búsqueda el objetivo es el mismo: encontrar la posición del elemento en la estructura. Por lo tanto la postcondición es:

$$a\,[i] == e$$

Los algoritmos de búsqueda pertenecen al método **Reducción**, ya que vamos a ejecutar hasta encontrar el elemento buscado, por lo que siempre el resultado va a estar en lo que resta por computar. A medida que avanzamos descartamos posiciones de la estructura de datos que sabemos que no contienen el elemento que buscamos.

Nuevamente utilizaremos un tipo de datos genérico ya que con esto construimos soluciones generalizadas que pueden ser replicadas en diversas situaciones.

4.1.Búsqueda lineal

En términos cotidianos la búsqueda lineal no es más que recorrer '*de uno en uno*' todos los elementos del conjunto que estamos analizando y buscar la posición del primer elemento que cumple la condición que definimos. Esto ya nos indica que necesitaremos un índice para realizar la recorrida. Para la búsqueda lineal vamos a buscar la primera aparición de un elemento en una *array*. Por lo que nuestra postcondición cambia levemente cuando aplicamos este algoritmo. Podemos entonces escribirla de la forma:

Postcondición: i es la posición del array a que contiene la primer aparición del elemento e

La postcondición puede ser descompuesta en dos condiciones:

1) a[i] == e

2) para todo j menor que i, a[j] es distinto de e.

Esta descomposición nos da la ventaja de poder aplicar la técnica de **Eliminación de una condición** para derivar la invariante, donde usaremos la condición 1 como condición de terminación y la otra como invariante, ya que de la otra forma no tendría sentido.

Invariante 1: para todo j menor que i, a[j] es distinto de e.

Terminación: a[i] == e

La invariante limitante es similar a la que obtuvimos en el primer ejemplo, es decir, *i* oscila entre 0 y el largo del *array*.

Invariante 2: $0 \leq i \leq N$

La inicialización de la variable *i* es bastante trivial, ya que vamos a realizar la recorrida de izquierda a derecha, es decir, comenzando por el cero hasta llegar al largo del *array*. Si inicializamos *i* en 0 la invariante limitante la verifica la propia expresión y la esencial también, ya que los únicos enteros menores que el 0 son los números negativos, que no pertenecen al *array* y por eso sabemos que sus posiciones en el *array* son distintas de e.

Inicialización: i = 0;

El paso del *loop* simplemente incrementa el índice. Para una ejecución cualquiera del *loop* sabemos que el elemento no fue encontrado en las posiciones anteriores por nuestra invariante 1, y sabemos que tampoco está en la posición *a[i]* porque de ser así la ejecución del algoritmo no habría entrado al *loop*. Por lo tanto sabemos que para todo valor de j menor que *i+1*, *a[j]* es distinto de e y aumentar *i* en 1 sigue cumpliendo la invariante esencial. La limitante también se cumple, ya que si *i* antes de comenzar era mayor que 0 también lo será luego de cualquier iteración, pues lo único que se hace es incrementar *i*. La cota superior también se cumple porque la postcondición nos dice que el elemento *e* está presente en el *array*, por lo que en el peor escenario está en la última posición. De ser así *i* habrá tomado el valor N, por lo que está dentro del rango aceptado.

Paso del loop: i = i+1;

Por último nos falta la variante para poder asegurar que el algoritmo termina. De la misma forma que las otras recorridas de *arrays* tiene sentido definirla como *N-i*, ya que *i* aumenta en cada iteración al tiempo que *N* es constante. Además nuestra invariante limitante nos dice que el máximo valor que puede tomar *i* es *N* con lo que la variante presenta una cota inferior.

Variante: N-i

Por lo tanto el algoritmo se vería de la siguiente forma:

```
method busquedaEnArray(T[ ]: a, T e): nat i
{
        requires: el elemento e pertenece al array a.
        ensures: a[i] == e;
        ensures: para todo j menor que i, a[j] es distinto de e.
        loop{
                invariant: para todo j menor que i, a[j] es distinto de e.
                invariant: 0<=i<=N;
                initialization: i = 0;
                termination: a[i]==e;
                variant: N - i;
                progress{
                        i = i+1;
                }
        }
}
```

El algoritmo de la búsqueda lineal puede ser generalizado. En este caso puntual la condición que buscamos es simplemente la ubicación de un elemento determinado, pero nada impide que pueda ser usado para alguna otra condición, como por ejemplo, buscar el primer elemento en un *array* de enteros que sea un número primo. En casi cualquier escenario donde estamos buscado la primera ocurrencia de alguna condición se puede aplicar un algoritmo de búsqueda lineal, siempre y cuando estemos seguros de que el elemento pertenece al conjunto de posibles soluciones. Si la naturaleza del problema no nos lo asegura debemos forzarlo mediante una precondición.

El algoritmo generalizado es:

```
method busquedaGeneralizada( T[ ] a ) : int i
{
        requires: existe al menos un elemento en a que cumple la condición
        ensures: a[i] es el primer elemento que cumple la condición
        loop{
```

```
            invariant: ningún elemento menor que i cumple la condición
            invariant: 0<=i<=N
            initialization: i = 0;
            termination: a[i]-cumple-condición
            variant: N - i;
            progress{
                  i = i+1;
            }
      }
}
```

Este *template* de solución es muy eficaz para los problemas que cumplen con el esquema definido. Cada caso tiene su propia invariante dependiendo del dominio del problema, aunque en general se basan en la aplicación de la técnica de relajación de constantes, de donde se derivan dos invariantes: la limitante, que delimita el rango en el que oscila *i*, y la esencial, que está relacionada con la condición de parada definida en la guarda del *loop*. También es un caso de aplicación de la técnica de ajuste ***Aging***, donde aseguramos que la invariante se cumple hasta el elemento previo al que nos encontramos. Como en todo problema de **Reducción** la solución del problema se encuentra en la estructura que nos queda por procesar.

4.2. Búsqueda binaria

La búsqueda binaria consiste en buscar un elemento en una estructura ordenada. Esto nos va a facilitar mucho el desarrollo de problemas de búsqueda así como también reducirá significativamente el orden del tiempo de ejecución de los algoritmos.

Adivinando
Para comenzar a comprender de qué se trata la búsqueda binaria, y cómo puede ayudarnos con problemas de búsqueda, comencemos con este sencillo ejemplo.

Suponga que un amigo le propone el siguiente juego: él pensará un número del 1 al 100, y usted tratará de adivinarlo. Por cada intento fallido, su amigo le dirá si el número que él piensa es mayor o menor que el que usted dijo.

Si quisiéramos resolver este problema mediante búsqueda lineal, en la práctica tendríamos una conversación similar a esta:

- Comencemos. ¿En qué número estoy pensando?
- Uno
- No. Mayor.
- Dos.
- No. Mayor.
- Tres.
- No. Mayor.

Y así sucesivamente, hasta que nuestro amigo decida retirarse porque el juego ya lo estaba aburriendo, o, expresado en términos algorítmicos, nuestra manera de resolver el problema sería muy poco eficiente, y esto se debe a que con búsqueda lineal, solamente descartamos un posible valor con cada intento fallido que realizamos. Luego, en promedio, esperaríamos 50 resultados fallidos antes de acertar el número correcto.
Si pudiéramos descartar más de un valor cada vez que fallamos, ya lo estaríamos haciendo mejor. Si el lector ya ha jugado a este juego alguna vez, sabrá que una buena táctica es escoger un valor intermedio en el rango de los valores candidatos. Esta táctica descarta la mitad de los valores posibles en cada intento. Si al comenzar el juego, escogemos el valor 50, y nos dicen que el número que buscamos es mayor, luego ya descartamos todos los números entre el 1 y el 50. En el segundo intento, descartamos 25 valores más, y así sucesivamente hasta que tengamos éxito. De esto se trata búsqueda binaria.

Búsqueda Binaria en general
Vamos a elaborar un algoritmo para encontrar un elemento dado en un *array* ordenado mediante búsqueda binaria.

La precondición del problema es que el *array* de entrada está ordenado de manera creciente y que el elemento que buscamos pertenece al *array*, y la postcondición es que el retorno corresponde al índice del elemento que buscamos.

***Precondiciones**: a está ordenado de manera creciente. value ∈ a*

***Postcondición**: a [mid] == value*

Vamos a aplicar la técnica **Relajación de constante** (doble), por lo que sabiendo que el valor pertenece al *array a*, podemos traducir esto a que el valor se encuentra en el segmento *a[0..N-1]*, y relajando las constantes *0* y *N-1* por *i* y *j* obtenemos las siguientes invariantes:

Invariante: value a[i .. j];
Invariante: $0 \leq i \leq j < N$

Inicializamos las variables *i* y *j* con los valores *0* y *N-1* respectivamente y de esta manera verificamos inicialmente las invariantes.

Inicialización: i, j = 0, N-1;

Luego el algoritmo va a funcionar así: vamos achicando el conjunto de valores posibles, incrementando *i* o disminuyendo *j* según corresponda, hasta que *i* sea igual a *j*, y así sabremos que encontramos el elemento. Además, *mid* también debe ser igual a estas variables, ya que la postcondición dice que *a [mid] == value*.

Terminación: i == j == mid

En cada iteración buscaremos un valor intermedio *mid* entre *i* y *j*, y preguntaremos por este valor. Aquí tenemos tres posibilidades: si encontramos el valor deseado, entonces asignamos el valor de *mid* tanto a *i* como a *j*. La invariante esencial se mantiene en este caso, ya que el valor buscado va a ser igual a *i* y a *j*, y éste se encuentra en el segmento *[i .. j]*. De hecho es el único elemento de este segmento.
Si a[*mid]*es menor que el valor buscado, luego todos los valores en posiciones anteriores a *mid* también son menores que el valor buscado, y ya podemos descartarlos. En este caso concluimos que el valor que buscamos se encuentra entre *mid + 1* y *j*, y por lo tanto asignamos *i = mid + 1*.
Si a[*mid]*es mayor que el valor buscado, estamos ante el caso simétrico, y asignamos *j = mid - 1*.
Sabemos que cualquier valor de *mid* entre *i* y *j* es suficiente para probar la validez de las invariantes y corrección del algoritmo, por lo tanto elegiremos *mid* =$\frac{i+j}{2}$.

Paso del loop:

mid = (i+j)/2;

if (a[mid] == value) i ,j = mid, mid

else if (a[mid] < value) i = mid + ;

else j = mid − 1

La variante será *j - i*, ya que en cada paso del *loop*, o bien incrementamos *i*, o bien decrementamos *j*, y su cota inferior es *0*, ya que el *loop* termina cuando *i* == j.

Variante: j - i

A continuación, la implementación del algoritmo de búsqueda binaria generalizado:

```
method busquedaBinaria (T [ ] a, T value): nat mid
{
        requires  a está ordenado de manera creciente. value ∈ a
        ensures a[mid] == value
        loop
        {
                invariant value ∈ a [i..j];
                invariant  0 ≤ i ≤ j<N
                initialization: i, j, mid = 0, N -1, 0;
                termination: i == j == mid
                variant: j - i
                progress{
                        mid = (i + j) / 2;
```

```
                    if (a[mid] == value) i ,j = mid, mid;
                    else if (a[mid] < value) i = mid + 1;
                    else j = mid − 1;
                }
        }
}
```

Luego del problema de la adivinanza, el lector debe haberse convencido que la búsqueda binaria es mucho más rápida que la búsqueda lineal. Particularmente para dicho problema, no nos tomaría más de siete intentos "adivinar" un número entre 1 y 100, mientras que el peor caso en búsqueda lineal consta de 100 intentos.

Entonces, ¿qué tan rápida y efectiva es la búsqueda binaria? Lo importante aquí es estudiar cuántas veces se ejecutaría el *loop* para un *array* de largo N. Para hacerse una idea, luego de una iteración con búsqueda binaria, reducimos nuestros valores posibles a la mitad de los elementos. De estos n/2 elementos, en la próxima iteración eliminaremos nuevamente la mitad, por lo que nos quedarían n/4 elementos posibles para buscar. Este proceso continúa hasta que el espacio de búsqueda nos queda conformado por un elemento y por lo tanto terminamos la ejecución. Obviamente, podemos tener suerte y encontrar el elemento buscado en el primer intento, pero la eficacia de un algoritmo se mide por el peor caso posible.

Luego de *i* iteraciones, nos quedan $\frac{N}{2^i}$ elementos sobre los que buscar. Cuando $\frac{N}{2^i}$ valga 1, el *loop* habrá finalizado, y ¿cuánto tiempo nos llevó llegar a este punto? Queremos hallar *i*, y sabiendo que $\frac{N}{2^i} = 1$, obtenemos que $2^i = N$, por lo tanto $i = \log_2 N$.

De aquí obtenemos que, en el peor caso, nuestro algoritmo de búsqueda binaria requiere un tiempo proporcional al logaritmo en base 2 del tamaño del *array* a considerar. Si jugamos al juego de la adivinanza con valores de hasta un millón, estamos en condiciones de adivinar con un máximo de 20 intentos, y si fuera hasta un billón, no sobrepasaremos los 30 intentos.

5.Algoritmos de sorting

Una de las áreas que más se ha trabajado en la programación es la ordenación de un conjunto de elementos. Desde los comienzos de la algoritmia se ha buscado la forma más eficiente de ordenar un conjunto de datos, en este caso de manera ascendente. El conjunto lo vamos a representar como un *array*. Existen diversas estrategias para resolver el problema de ordenar un *array*, pero todas tienen el mismo objetivo, es decir, las mismas postcondiciones. En muchos problemas, vamos a utilizar una función llamada intercambiar, que recibe como parámetros un array y dos índices válidos del mismo, e intercambia los elementos que ocupan esas posiciones entre sí, sin modificar el resto de la estructura.

Postcondiciones:

El array resultante es una permutación del original, lo que significa que contienen los mismos elementos pero no necesariamente el mismo orden. Lo representamos como: unmodified(a)

El array resultado está ordenado ascendentemente. Formalmente se puede expresar como: $\forall\, i \in [0..N-2]$ *:a [i] ≤ a [i+1], pero lo simplificamos representándolo como sorted(a).*

5.1.Insertion sort

El primer algoritmo de *sorting* que vamos a analizar es conocido como ***Insertion Sort***.

Postcondición: sorted (a) && unmodified (a)

El método para resolver este problema es ***Updating,*** donde en cada paso sabemos que la parte ya recorrida del *array* está ordenada, por lo que al terminar la recorrida habremos ordenado toda la estructura.

Inicialmente contamos con un solo elemento (el primero del *array*), que obviamente es un conjunto ordenado. Luego, para cada elemento del *array*, recorremos desde su posición hacia atrás hasta encontrar la posición en la que debe ser insertado y que mantenga el conjunto de forma ordenada. En otras palabras, lo sacamos de la parte desordenada del *array* y lo insertamos en su posición correspondiente de la parte ordenada.

Podemos deducir que la implementación de este algoritmo requiere dos *loops* anidados: el primero recorre todo el *array* de izquierda a derecha, y para cada elemento ejecuta un método que llamaremos *Insert*, cuya función es insertar el elemento manteniendo ordenada la porción ya recorrida del *array*.

Comencemos por el *loop* principal. Este ejecuta una recorrida del *array* de izquierda a derecha, y asegura que lo ya recorrido está ordenado. Aplicaremos la tradicional **Relajación de constantes,** junto con el ajuste ***Aging*** (ya que el *array* podría ser vacío) para derivar las invariantes. Además

agregamos la invariante *unmodified(a)* estableciendo que los elementos del *array* no se modifican. Por lo tanto obtendremos las siguientes invariantes:

Invariante 1: sorted(a[0..i-1])

Invariante 2: 0 <= i <= N

Invariante 3: unmodified(a)

Este *loop* lo inicializamos con *i=0* y terminamos luego de recorrer toda la estructura.

Inicialización: i = 0;

Terminación: i == N;

Esta inicialización verifica las invariantes porque: no modifica los valores del *array*, el *subarray a[0..-1]* es vacío por lo que está ordenado, y también se verifica la invariante 2 (limitante) porque cumple la sustitución de valores.

Al entrar al paso del *loop* sabemos que el *subarray a[0..i-1]* está ordenado. Luego llamamos al método *Insert*, que insertará *a[i]* en su posición correcta, e incrementamos *i* en una unidad, por lo que podremos saber que *a* va a estar ordenado ahora hasta el último elemento insertado, el de la posición *i-1*, verificando así la invariante luego del paso. Por el momento asumimos que *Insert* se comporta como esperamos. Luego desarrollaremos el método con sus correspondientes demostraciones.

Por último, sabiendo que *N* es constante y que *i* incrementa en cada paso, podemos concluir que la variante del *loop* será *N - i*.

Variante: N - i;

De esta manera, el algoritmo de *Insertion sort* será de la siguiente manera:

```
method InsertionSort (T[ ] a): void
{
        ensures sorted (a) && unmodified (a)
        loop
        {
                invariant: 0 ≤ i ≤ N
                invariant: unmodified(a)
                invariant: sorted(a[0..i-1])
                initialization: i = 0;
                termination: i == N;
```

```
            variant: N - i;
            progress
            {
                v = a[i];
                Insert (a,i,v);
                i = i+1;
            }
        }
}
```

Ahora pasemos a la esencia de este algoritmo: el método *Insert*.

Este método recibe por parámetro el *array*, el elemento que debe ser insertado, al que llamamos *v*, y su índice en el *array*. La precondición del método es que el *array* está ordenado hasta una posición antes que el elemento actual, y la postcondición es que está ordenado hasta el índice que pasamos y que incluye a *v*, además de que no se modifican los elementos del *array* a.

Precondición: sorted (a [0.. i-1])

Postcondición: sorted (a [0..i]) && $\boldsymbol{v} \in \boldsymbol{a}\,[\mathbf{0}..\boldsymbol{i}]$

Postcondición: unmodified (a)

Este método lo resolveremos mediante **Reducción**: recorriendo desde el elemento *a[i]* hacia la izquierda, descartando posiciones donde podríamos insertar el elemento, y a medida que lo hacemos desplazamos cada elemento que visitamos hacia la derecha. Cuando encontramos un elemento menor que *v* no seguimos recorriendo más hacia la izquierda ya que allí habremos encontrado la posición donde insertar *v*. La otra posibilidad que puede darse es recorrer toda la estructura hasta *a[0]* sin encontrar elementos menores que *v*. En ese caso, *v* es el menor elemento en *a[0..i]*, por lo que ocupará la primera posición.

Para derivar las invariantes aplicamos **Relajación de constantes** al parámetro *i*, reemplazándolo por *j* que oscilará entre *i-1* (comenzamos a recorrer desde una posición a la izquierda), y como mínimo llegará a valer *-1*, en caso que *v* deba ser insertado al comienzo del *array*.

Invariante 1: $-1 \leq j < i \leq N$

Además, en cada paso del *loop* sabemos que *v* es menor o igual que los elementos que ya visitamos, ya que en caso contrario hubiéramos detenido la recorrida, y también que el segmento del *array* que ya analizamos está ordenado.

Invariante 2: $v \leq a\,[j + 1..i\,]$

Invariante 3: sorted (a [j + 1..i])

Como ya hemos explicado, *j* se inicializa en *i-1*, y la recorrida se ejecuta hasta encontrar un elemento menor que *v*, o eventualmente luego de haber recorrido todo el *array*.

Inicialización: j = i - 1;

Terminación: j < 0 || a[j] < v

Sabemos que si estamos ejecutando el paso del *loop* es porque *a[j]* es mayor o igual a *v*, por lo tanto desplazamos *a[j]* un lugar hacia la derecha. Luego disminuimos *j* para seguir avanzando hacia la izquierda en el *array*, manteniendo así las invariantes que dicen que el elemento *v* debe ser menor o igual que la sección ya recorrida, y que la misma está ordenada.

Paso del loop : a[j+1] , j = a[j], j-1;

Al salir del *loop* es porque habremos encontrado la posición donde insertar *v* y eso es lo que hacemos.

La variante será *j*, ya que disminuye en cada paso, y su cota inferior es *-1*.
Ejecutando este método para todos los elementos del *array*, alcanzaremos el orden ascendente que buscamos en la estructura.

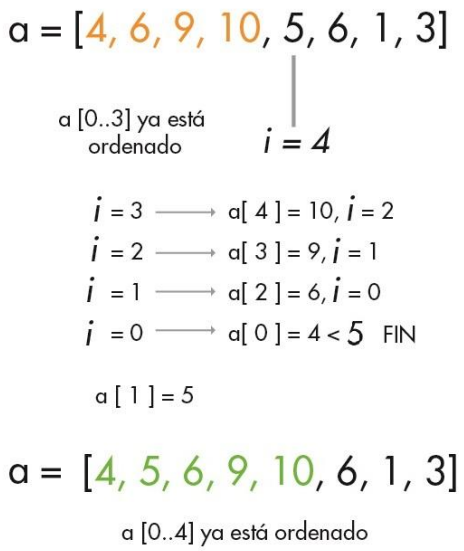


Imagen 5-1: Ejemplo de Insert

Implementación:

method Insert (T [ ] a, nat i, T v): void

```
{
        requires sorted (a [0.. i-1])
        ensures sorted (a [0..i]) &&  v ∈ a [0..i]
        ensures unmodified (a)
        loop
        {
                invariant:  -1 ≤ j < i ≤ n
                invariant: unmodified (a)
                invariant: sorted (a [ j + 1..i ])
                invariant: v ≤ a [j + 1..i ]
                initialization: j = i - 1;
                termination: j < 0 || a[j] < v
                variant: j;
                progress
                {
                        a[j+1] = a[j];
                        j = j - 1;
                }
        }
                a[j+1] = v;
}
```

5.2.Selection sort

Este algoritmo está basado en una idea sencilla de ordenamiento: encontrar el menor elemento del *array*, insertarlo en la primera posición, y luego repetir este proceso desde la segunda posición hasta el final. El algoritmo de *Selection sort* lo resolvemos mediante el método ***Updating***. A medida que avanzamos en la ejecución la sección del *array* ordenada aumenta cada vez más hasta que finalmente abarca todo el *array*.

Este problema no tiene precondiciones, y las postcondiciones son las mismas que para todos los algoritmos de *sorting*.

Postcondición: sorted (a) && unmodified (a)

Como la idea del algoritmo es ir ordenando una porción del *array* a medida que avanzamos vamos a aplicar la técnica **Relajación de Constantes** junto con ***Aging*** (ya que el array podría ser vacío) para derivar las invariantes. Además no vamos a modificar el contenido del *array* en ningún momento, ya que para estar ordenado el *array* debe contener los mismos elementos que en su versión original como nos dice una parte de la postcondición. Por lo tanto vamos a agregar esta misma condición como invariante. Las invariantes quedan de la forma:

Invariante 1: sorted (a [0..i-1])

Invariante 2: $0 \leq i \leq N - 1$

Invariante 3: unmodified(a)

Vamos a añadir una invariante más, específica del propio algoritmo y que necesitamos para poder avanzar. Ya que vamos a seleccionar el menor de la sección del *array a[i..N-1]* en cada paso, sabemos además que todos los elementos de esa sección no necesariamente están ordenados pero todos ellos son mayores que cualquier elemento del *subarray a[0.. i-1]*, la sección ya ordenada.

Invariante 4: todos los elementos de a[i..N-1] son mayores que todos los de a[0..i-1]

Vamos a recorrer el *array* de izquierda a derecha, por lo que vamos a inicializar el índice *i* en 0. Gracias a la invariante 4 sabemos que no será necesario recorrer hasta el último elemento, sino que yendo hasta uno antes será suficiente, ya que cuando *i* valga *N-1*, todos los elementos a su izquierda estarán ordenados y serán menores que él, por lo tanto este es el mayor del *array* y es correcto que se ubique en la última posición.
De aquí podríamos desprender que el *loop* debe terminar cuando *i* sea igual a *N-1*. Esto es correcto en el caso que el *array* contenga elementos, pero no abarca el caso para un *array* vacío. Por esto, hacemos que la condición de terminación sea $i \geq N - 1$. De esta forma, si el *array* es vacío el algoritmo no entra al loop.

Inicialización: i = 0;

Terminación: $i \geq N - 1$;

Al inicializar *i* en 0 podemos observar que se cumple la invariante 1 porque *a[0..-1]* es vacío, misma razón por la que se establece la invariante 4. La verificación de la invariante 2 es trivial. La invariante 3 se cumple porque no hemos hecho nada aún.

En cada paso del *loop* vamos a buscar el menor elemento del segmento *a[i..N-1]*. Gracias a la invariante 4 sabemos que todos los elementos en el segmento *a[0..i-1]* son menores que el menor de *a[i..N-1]*, entonces podemos intercambiar el menor de esta porción por el de la posición *a[i]* y luego aumentar el índice *i*. La invariante 1 se cumple porque el elemento que agregamos a la sección conocida del *array* es mayor que los anteriores, por lo que sigue estando ordenado. La invariante 2 se sigue cumpliendo porque la condición de terminación impide que *i* entre con un valor igual a *N-1*, por lo que nunca podría pasar su valor. La invariante 3 se mantiene porque sólo intercambiamos elementos, no agregamos ni eliminamos los mismos. Por último la invariante 4 se sigue manteniendo porque al tomar el menor elemento de la sección desordenada y pasarlo a la sección ordenada los restantes seguirán siendo mayores que cualquier elemento de los que ya fueron ordenados.

La variante será *N - i*, de forma similar a lo que hicimos en las otras recorridas de *arrays*.

Variante: N - i

method SelectionSort(T[ ] a): void

```
{
        ensures unmodified(a) && sorted(a)
        loop
        {
                invariant: unmodified (a)
                invariant: 0 ≤ i ≤ N - 1
                invariant: sorted (a[0..i-1])
                invariant: todos los elementos de a[0..i-1] son menores que cualquiera de
        a[i..N-1]
                initialization: i = 0;
                termination: i ≥ N-1
                variant: N - i
                progress:
                {
                        m = minIndex(a,i,N-1);
                        intercambiar (a,i,m);
                        i=i+1;
                }
        }
}

method minIndex(T[ ] a, nat from,nat to): int pos
{
        ensures: pos es el indice del menor elemento de a[from..to]
        requires 0 ≤ from ≤  to <N
        loop{
                invariant: pos es el indice del menor elemento de a[from..i-1]
                invariant: from ≤ i ≤ to
                initialization: i, pos = from, from
                termination: i== to
                variant: to - i;
                progress:{
                        i = i+1;
                        if(a[i]<a[pos]) pos = i;
                }
        }
}
```

5.3. Quick-sort

El algoritmo de *Quick-sort* es implementado recursivamente mediante sucesivas elecciones de elementos a los que llamamos *pivot* (*pivot* debe pertenecer al *array*) y llamadas al método *Partition* con el *array* y el *pivot* escogido. Ya que en este trabajo no tratamos sobre soluciones recursivas a problemas, vamos a enfocarnos en el método *Partition*, que es la parte central de *quick-sort*.

Este método consiste en, a partir de un *array* de entrada y un elemento del mismo que llamamos "*pivot*", colocar el *pivot* en su lugar del *array* ordenado. Es decir, que todos los elementos a la izquierda del *pivot* son menores a él, y a la derecha son mayores, asumiendo que no hay elementos repetidos.

Precondiciones: El array a no contiene elementos repetidos y pivot pertenece al array.

Postcondición: El pivot se encuentra en su posición ordenada.

Este problema lo vamos a resolver mediante el método ***Updating***. Vamos a crear dos *arrays* auxiliares, a los que llamaremos *lowValues* y *highValues*, y vamos a recorrer todo el *array* original insertando en *lowValues* los elementos que sean menores que el *pivot*, y en *highValues* los mayores. Con esta idea, y aplicando **Relajación de constantes** para la constante *N* (largo del *array*), con el ajuste de ***Aging*** llegamos a las siguientes invariantes:

Invariante 1: $0 \leq i \leq N$

Invariante 2: unmodified (a)

Invariante 3: Si a[j] <pivot, a[j] pertenece a lowValues, $j, 0 \leq j < i$

Invariante 4: Si a[j]>pivot, a[j] pertenece a highValues, $j, 0 \leq j < i$

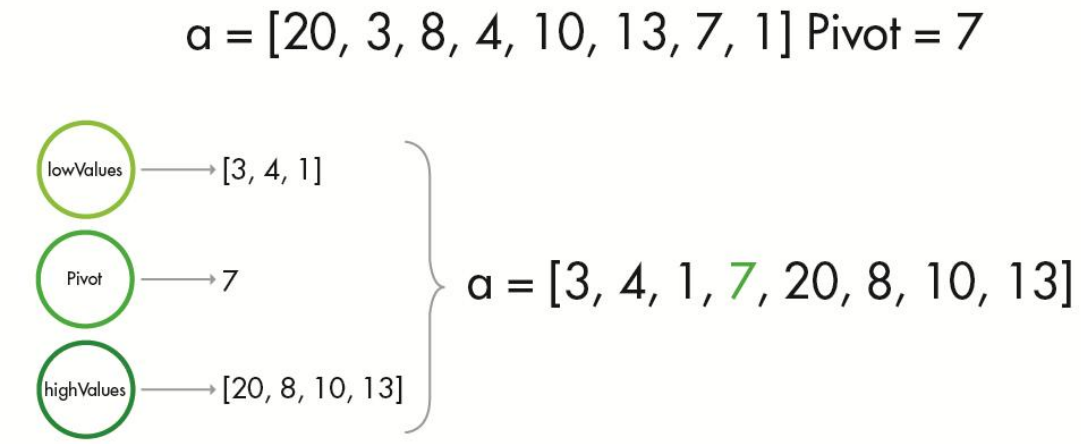
Inicialmente *i* vale 0, la primera posición del *array*, y los *subarrays lowValues* y *highValues* comienzan vacíos, validando así todas las invariantes. Además, vamos a recorrer toda la estructura, por lo que la condición de terminación es *i == N*.

Inicialización: i, lowValues, highValues = 0, [], [];

Terminación: i == N

Luego en cada paso, preguntamos por el elemento *a[i]*: si este es menor que el *pivot* lo insertamos en *lowValues*, y si es mayor hacemos lo propio en *highValues*, y posteriormente incrementamos *i* en una unidad. Esto valida todas las invariantes, y se puede comprobar muy fácilmente por la trivialidad del paso.

Al finalizar el *loop*, concatenamos (con el operador '++') el *array lowValues*, seguido de *pivot* y luego de *highValues*, cumpliendo así con la postcondición del problema: el *pivot* se encontrará en su posición ordenada del *array*. Si luego de hacer esto llamamos recursivamente al método *Partition* con cada *subarray*, finalmente habremos ordenado todo el *array*.



Imágen 5-2:Ejemplo de Partition

Finalmente, la variante es la habitual en problemas donde recorremos una estructura de izquierda a derecha con un índice.

Variante: N - i

A continuación implementamos el método *Partition*.

```
method Partition (T [ ] a, T pivot):
{
        requires a no contiene elementos repetidos
        requires pivot pertenece a a
        ensures El pivot se encuentra ordenado en a
        loop
        {
                invariant:  0 ≤ i ≤ N
                invariant: unmodified (a)
                invariant: en a[0..i-1] si a[ j ] < pivot, a[ j ] pertenece a lowValues
                invariant: en a[0..i-1] si a[ j ] > pivot, a[ j ] pertenece a highValues
                initialization: i, lowValues, highValues = 0, [ ], [ ];
                termination: i == N
                variant: N - i;
                progress
                {
                        if (a[i] < pivot) lowValues.add(a[i]);
                        else if (a[i] > pivot) highValues.add(a[i]);
                        i = i + 1;
                }
        }
        a = lowValues ++ pivot ++ highValues;
}
```

5.4.Flag Sorting

Flag sorting consiste en ordenar un *array* sabiendo de antemano cuántos valores distintos tiene. El objetivo es hacerlo utilizando sólo una iteración, logrando un algoritmo de orden *n* (siendo *n* la cantidad de elementos del *array*).

5.4.1 Dos valores

Comencemos con un *array* que solamente contiene dos valores distintos: 0 y 1. El objetivo es retornar un *array* que contenga los mismos elementos que el *array* original, pero ordenado, es decir, con todos los 0's al principio y seguido de los 1's hasta el final. Es importante señalar que debemos mantener la cantidad de elementos del *array*. Por ejemplo, si nuestro *array* original es {0,1,1,0,1,1,0}, debemos retornar {0,0,0,1,1,1,1}.

La postcondición es la misma que la de las otras ordenaciones. Se debe cumplir *unmodified*(a) y *sorted*(a). En este caso, al haber solamente dos elementos, el *array* de retorno será una serie de 0's seguido de una serie de 1's. Esta vez tenemos una precondición que nos dice los elementos que pueden formar parte del *array*.

Precondición: a está compuesto solamente por 1s y 0s

Postcondición: sorted(a) && unmodified(a)

Nuevamente para resolverlo vamos a utilizar el método ***Updating*** recorriendo toda la estructura de izquierda a derecha. Lo primero que hacemos es aplicar **Relajación de Constantes**, con el ajuste ***Aging***, para derivar una invariante esencial y una limitante. A su vez, vamos a mantener siempre la condición *unmodified*(a) porque nunca vamos a quitar ni agregar elementos, por lo que lo vamos a expresar también en forma de invariante.

Invariante 1: sorted (a [0..i-1])

Invariante 2: $0 \leq i \leq N$

Invariante 3: unmodified(a)

Ahora debemos definir el paso del *loop*. De la invariante 1 sabemos que el *array a[0..i-1]* está ordenado ascendentemente. Esto es equivalente a decir que tenemos algunos 0s seguido de algunos 1s. Si *a[i]* es 1, no es necesaria ninguna modificación, ya que éste 1 aparecería después de la cadena de 1's, así que solamente incrementamos *i* y seguimos. Y si *a[i]* es 0? Sabemos que este 0 debe encontrar su lugar antes de la cadena de 1's, pero no sabemos dónde se encuentra la división entre 0's y 1's en la sección ya ordenada. Evidentemente, tenemos que guardar el índice del primer 1 de la parte del *array* que ya fue ordenado. Para guardar la posición del último elemento debemos modificar la invariante esencial para que nos asegure que esta posición es efectivamente la que queremos que sea. La invariante 1 podemos expresarla diciendo que todos

los elementos de *a[0..j-1]* son 0 y que todos los elementos de *a[j..i-1]* son 1. Por lo tanto podemos reescribir las invariantes como:

Invariante 1: a[0..j-1] son todos 0

Invariante 2: a[j..i-1] son todos 1

Invariante 3: $0 \leq j \leq i \leq N$

Invariante 4: unmodified (a)

La recorrida será de izquierda a derecha, por lo que vamos a inicializar los índices en 0 y el algoritmo va a terminar cuando *i* sea igual al largo del *array*.

Inicialización: i, j = 0, 0;

Terminación: $i == N$

Fácilmente podemos notar que la inicialización verifica la invariante 3 y dado que el conjunto *a[0..-1]* es el conjunto vacío también se verifican las invariantes 1 y 2. Como aún no hemos hecho nada también se verifica la invariante 4.

Ahora que sabemos que la posición *j* guarda la ubicación del primer 1 del *array* en la parte ordenada sabemos qué hacer en caso de que *a[i] = 0*. En ese caso debemos intercambiar *a[i]* con *a[j]* y aumentar tanto *j* como *i* en uno. De esta forma añadimos un cero antes que comiencen los 1s y agregamos el 1 que estaba en esa posición al final del *array* ya ordenado.

El paso del *loop* cumple la invariante 3 por la condición de terminación y porque *j* nunca puede tomar un valor mayor que *i*, ya que aumentamos ambos en el caso de que *a[i]=0* o solamente *i* cuando *a[i]=1*. Las invariantes 1 y 2 se siguen manteniendo porque seguimos teniendo un grupo de ceros seguido de un grupo de unos delimitados por *j* e *i*, respectivamente. En uno de los escenarios simplemente aumentamos el índice porque el elemento a agregar es un 1 y en el otro caso añadimos un 0 después de una secuencia de ceros, por lo que sigue estando ordenado. Como solamente intercambiamos elementos también se cumplirá la condición 4.

Por último la variante será $N - i$.

El algoritmo resultante sería entonces:

```
method flagSorting(T[ ] a): void{
        ensures: sorted(a);
        ensures: unmodified(a);
        requires: a está compuesto por 0s y 1s únicamente
```

```
        loop{
                invariant: sorted(a[0..i-1])
                invariant: 0 ≤ i ≤ N
                invariant: unmodified(a)
                invariant: a[0..j-1] son todos 0
                invariant: a[j..i-1] son todos 1
                invariant: 0 ≤ j ≤ i
                initialization: i, j = 0, 0;
                termination i == N
                variant: N-i;
                progress{
                        if(a[i]==0){
                                intercambio(a,j,i);
                                j = j+1;
                        }
                i=i+1;
                }
        }
}
```

5.4.2 Tres valores

Este problema originalmente fue formulado por Dijkstra y denominado "El problema de la bandera holandesa" [2], ya que los elementos de su *array* eran los colores rojo, blanco y azul. El objetivo es ordenar los elementos para que los azules queden al principio del *array*, seguidos de los blancos y luego los rojos, como la bandera holandesa de abajo hacia arriba.

Nuestro algoritmo será idéntico, solamente que para los valores 0, 1 y 2. En realidad, la elección de los tres elementos es irrelevante para el algoritmo. Recién vimos que para dos posibles valores, dividimos nuestro *array* en tres tramos: los 0's, los 1's, y los que quedan por procesar. Para un valor más, simplemente se agrega un tramo más para fraccionar el *array*. Por lo tanto nuestras invariantes quedan de la forma:

Invariante 1: Los elementos en a[0..i-1] son 0's;

Invariante 2: Los elementos en a[i..j-1] son 1's;

Invariante 3:Los elementos en a[j..k-1] son 2's;

$$Invariante\ 4: 0 \leq i \leq j \leq k \leq N$$

Invariante 5: unmodified (a);

La introducción de la variable *k* determina el final de la sección de 2's. Esta variable es la que actúa como índice durante la recorrida. Por lo tanto podemos definir las condiciones de inicialización y terminación del *loop*:

Inicialización: i, j, k = 0, 0, 0;

Terminación: k == N

Hay tres casos que debemos considerar en cada iteración, uno por cada posible valor de *a[k]*. Si *a[k]* es 2, la situación es análoga a cuando *a[i]* valía 1 en nuestro último algoritmo. En ese caso, simplemente incrementamos *k*, extendiendo así la secuencia de 2's que comienza en *a[j]*. Si *a[k]* es 1, la situación es análoga a cuando *a[i]* era 0 en el algoritmo recién visto, por lo que intercambiamos el valor del *array* en la posición *j* por la posición *k* y aumentamos ambos índices.

El novedoso caso, y que nos llevará más trabajo, es cuando *a[k]* sea 0. Queremos que este elemento siga a cualquier 0 presente en el *array*, así que podríamos pensar en colocarlo en *a[i]* e incrementar *i* en 1. Esto sería intercambiar *a[k]* que es el 0 que acabamos de encontrar por *a[i]*, y luego incrementar ambas variables, *i* y *k*. Lamentablemente, ¡esto no funciona! ¿Por qué?

El problema es que el valor de *a[i]* es intercambiado por *a[k]*, y no podemos garantizar que hemos colocado un 2 en *a[k]*. Luego de haber procesado algunos 1's, *a[i]* será 1, por lo que terminaríamos colocando un 1 en *a[k]*. Luego, al incrementar *k*, estaríamos incumpliendo la parte de la invariante que establece que el segmento *a[j.. k - 1]* está conformado por 2's, por lo tanto no podemos aumentar *k*. Lo que hacemos entonces es mantener su valor para que la posición *k* vuelva a ser procesada por el *loop* y se coloque este elemento en su lugar correspondiente.

El primer caso todavía nos trae problemas, ya que podría incumplir la invariante limitante. Por ejemplo, supongamos que tenemos el *array* [0 , 2]. En nuestra primera iteración, tenemos *i=j=k=0*. Preguntamos por *a[k]*, encontramos un 0, por lo tanto ejecutamos la condición del primer *if*, resultando en un intercambio y un incremento de *i*. Esto nos lleva a que *i* sea mayor que *j*, cuando la invariante limitante dice que $i \leq j$.

La única forma de que *i* sea mayor que *j* es si son iguales antes de incrementar *i*. Previo al incremento de *i*, el rango *a[i..j−1]* es vacío, por lo que debemos mantenerlo vacío luego de la iteración. Esto significa que si llegamos al punto en que el valor de *i* supera a *j*, debemos incrementar *j*.
Simétricamente, tenemos la misma situación entre *j* y *k*, ya que podríamos llegar a incrementar *j* en una iteración sin el correspondiente incremento de *k*, cayendo en el mismo error. No podemos permitir que *j* supere a *k*, así que debemos preguntar por esta condición, y ajustar el valor de *k* cuando sea necesario.

En cada paso del *loop* se incrementan alguna de las variables auxiliares (*i, j, k*), al tiempo que *N* permanece constante, por lo que una variante válida es *N - k - i - j*. El valor máximo que pueden tomar estas variables es *N*, por lo que la cota inferior que puede tomar la variante es *-2N*.

Variante: N - k - i - j

Habiendo analizado todo esto, estamos en condiciones de resolver el problema de la bandera de Holanda:

```
method flagSortingTresValores(int[ ] a) : void{
        requires: a está compuesto solamente por 0's, 1's y 2's.
        ensures: unmodified(a)
        ensures: a queda compuesto por todos sus 0´s seguido por todos sus 1's seguido por
todos sus 2's
        loop{
                invariant: Todos los elementos en a[0..i-1] son 0
                invariant: Todos los elementos en a[i..j-1] son 1
                invariant: Todos los elementos en a[j..k-1] son 2
                invariant: 0 ≤ i ≤ j ≤ k ≤ N
                initialization: i, j, k = 0, 0, 0;
                termination: k == N;
                variant: N - k - j - i;
                progress{
                        if (a[k] == 0){
                                intercambiar (a, i, k);
                                i=i+1;
                                if(i>j)j=j+1;
                                if(j>k)k=k+1;
                        }
                        else if (a[k] == 1) {
                                intercambiar (a, j, k);
                                j = j+1;
                                k = k+1;
                        }
                        else k = k+1;
                }
        }
}
```

6.Algoritmos avanzados

En esta sección veremos problemas más avanzados sobre *arrays*, operaciones provenientes de la programación funcional, la Criba de Eratóstenes, problemas como la secuencia de Fibonacci o el cálculo de un cubo utilizando solamente sumas. Todos los algoritmos involucran el uso tanto de invariantes, como la metodología que venimos utilizando a lo largo del texto. Adicionalmente presentaremos una nueva técnica llamada **Fortalecimiento de Invariantes**.

Las mayores dificultades de estos algoritmos están en la correcta especificación de los problemas utilizando conocimientos proveniente del dominio de los mismos, así como en el desarrollo del paso del *loop* para hacerlo de forma eficiente, *performante* y elegante, por lo que no nos detendremos tan minuciosamente en detalles que ya han sido cubiertos y son triviales.

6.1. La meseta más larga

Supongamos que tenemos un *array* de enteros ordenado en forma creciente, por ejemplo:

$$a = \{1,1,2,2,2,3,3,3,3,4,5,5\}$$

Consideramos una meseta a un rango de números consecutivos iguales. Por ejemplo, {1,1} es una meseta, al igual que {2,2,2}, {2,2} (incluso hay dos posibilidades que se solapan, y ambas son consideradas mesetas), y {3,3,3,3}, entre otras. La meseta más larga es la secuencia de números consecutivos iguales más larga, en este caso es la que forman los 3's. El objetivo es desarrollar un algoritmo que retorne el largo de la meseta más larga para un *array* a de tipo genérico.

Precondición: el array a está ordenado crecientemente

Postcondición: m es el largo de la meseta más larga

Vamos a resolver este problema aplicando el método ***Updating,*** recorriendo todo el *array* de izquierda a derecha, y almacenando temporalmente el largo de la meseta más larga que hemos encontrado hasta el momento.

Para esto vamos a aplicar la técnica de **Relajación de constantes**, junto con el ajuste de ***Aging***, reemplazando la última posición del *array* por la variable *i*. Aplicando esta técnica junto con el ajuste obtenemos la siguiente invariante esencial:

Invariante 1: m es el largo de la meseta más grande de a[0..i-1]

Así como también una invariante limitante:

$$Invariante\ 2:\ 0 \leq i \leq N$$

Inicializamos *i* y *m* en 0, por lo que se verifica tanto la invariante limitante como la esencial debido a que el *array a[0..-1]* es vacío, por lo que es correcto que *m*, que representa el largo de la meseta más larga hasta el momento sea 0.

Inicialización: i, m = 0, 0;

Debemos recorrer todo el *array* porque para obtener el resultado es necesario el cómputo de toda la estructura. Por lo tanto la condición de terminación del *loop* será:

Terminación: i==N;

El desafío de este ejercicio está en encontrar una forma inteligente de desarrollar el cuerpo del bucle para que se sigan cumpliendo las invariantes y que no sea necesario tener dos *loops* anidados.

La invariante nos asegura que m es la meseta más larga hasta el momento. Esta variable solamente va a cambiar en caso de que estemos parados sobre un elemento que conforma una meseta más larga que la que representa el valor de *m*.

Por ejemplo, si nos encontramos en la posición 5 del *array* del ejemplo, ya hemos recorrido el *array* conformado por: {1,1,2,2,2} y el valor de m es 3, que es el largo de la meseta {2,2,2}. Si el siguiente elemento a investigar fuera otro 2, el valor de m aumentaría en uno, ya que agregaría un elemento a la meseta más larga. De no ser así el valor de m permanecería igual debido a que la meseta más larga seguiría siendo el segmento {2,2,2}.

Aquí podemos ver que m solo podría aumentar su valor en una unidad, si es que estamos parados sobre un elemento que conforma la nueva meseta más larga.

En cada paso del *loop*, estamos parados en *a[i]*. Si *a [i-m]* vale lo mismo que *a[i]*, significa que todos los elementos entre *a[i-m]* y *a[i]* tienen el mismo valor debido a la pre condición que establece que el *array* a está ordenado crecientemente. Por lo tanto tenemos una nueva meseta más larga, ya que la conformada por estos elementos es de largo *m+1*. En el caso que *a [i-m]* fuese distinto de *a[i]*, *m* conserva su valor original.

La variante se constituye de la misma forma que hemos visto en varios ejemplos del mismo estilo y se verifica gracias a nuestra invariante esencial y al paso del *loop*, donde aumenta *i* manteniéndose constante *N* (largo del *array*).

Variante: N-i

El algoritmo resultante es:

```
method mesetaMasLarga (T [ ] a): nat m
{
        requires: a no es nulo y está ordenado crecientemente.
        ensures: m es la meseta más larga del array a
        loop{
                invariant: m es el largo de la meseta más grande de a[0..i-1]
```

```
            invariant: 0 ≤ i ≤ N
            initialization: i, m = 0,0;
            termination: i == N;
            variant: n-i;
            progress{
                if(a[i] == a[i-m]) m = m+1;
                i = i+1;
            }
        }
}
```

6.2.Mayor segmento de suma

Un segmento de suma de un *array* a es el resultado de sumar todos los elementos de un *subarray* de *a*. Este algoritmo busca el mayor segmento de suma de un *array* de enteros recibido por parámetro. Evidentemente si el *array* solamente está conformado por naturales el mayor segmento de suma es aquel resultante de todo el *array*. Lo interesante es lo que ocurre cuando tenemos también números negativos. Por ejemplo, en el *array* {4, -3, 9, -5} el segmento de suma más largo es el resultante de 4 + (-3) + 9 = 10. Buscamos realizar un algoritmo que nos permita encontrar el valor del mayor segmento de suma de un *array* de enteros.

Postcondición: s es el valor del mayor segmento de suma en el array a

Como podemos ver, la estructura general del problema se asemeja mucho a la del problema anterior, donde buscamos la meseta más larga. Esta vez también vamos a aplicar el método de ***Updating***, utilizando un acumulador para resultados parciales, y la técnica de **Relajación de constantes** para derivar invariantes junto con el ajuste de ***Aging***.

Aplicando esta técnica obtenemos estas dos invariantes (esencial y limitante).

Invariante 1: s es el valor del mayor segmento de suma en el array a[0..i-1].

Invariante 2: $0 \leq i \leq N$.

Como vamos a recorrer de izquierda a derecha nuestra estructura es fácil predecir la inicialización de las variables y la condición de terminación del *loop*:

Inicialización 1: s, i = 0, 0;

Terminación: i == N;

Nuevamente el desafío está en buscar el paso del *loop* que conserve las invariantes mientras nos vamos acercando a la condición de terminación. También buscamos que el orden del algoritmo sea lineal, es decir, que no tenga iteraciones anidadas.

Sumar el elemento sobre el que estamos posicionados al resultado acumulado no garantiza aumentar el mayor segmento de suma encontrado hasta el momento (debido a que puede haber elementos negativos que disminuirían el resultado acumulado), por esto es que debemos introducir una variable adicional denominada fin. El objetivo de esta variable es guardar el resultado del máximo segmento de suma del *array* que termina en la última posición visitada (*a[i-1]*). Esta es la invariante esencial que indica el comportamiento de la variable *fin.* Además, podemos saber que *fin* oscila entre *0* y *s* ya que *s* es por definición el máximo segmento de suma, y *0* representa el segmento de suma vacío.

Invariante 3: fin es el resultado del máximo segmento de suma del array que termina en a[i-1]

Invariante 4: $0 \leq fin \leq s$

Tenemos que añadir la inicialización de la variable fin.

Inicialización 2: fin = 0;

Al inicializar fin en 0 se cumple la invariante 3 porque el segmento [0..-1] es vacío y 0 es su segmento de suma. Por sustitución de variables se cumple la invariante 4.

Lo que hacemos entonces es sumarle *a[i]* a la variable fin. Si *fin* supera a *s* entonces tenemos un nuevo candidato al mayor segmento de suma, y en ese caso *s* toma este nuevo valor. En el caso de que *fin* sea un número negativo le asignamos el 0, ya que la suma del conjunto vacío (que forma parte de todo *array*) es 0, y es superior al que había sido computado. Esto verifica la invariante 3 y 4. Además verificamos la invariante 1 porque *s* sigue siendo el máximo segmento de suma del *array,* y la invariante 2 por la condición de terminación.

Para finalizar, nuevamente la variante es igual a la de muchos de los problemas que resolvemos mediante relajación de constantes:

Variante: N-i

```
method segmentoDeSuma(int[ ] a) : nat s
{
        ensures: s es el valor del mayor segmento de suma en el array a
        loop{
                invariant: s es el valor del mayor segmento de suma en el array a[0..i-1]
                invariant: 0 ≤ i ≤ N
                invariant: fin es el resultado del máximo segmento de suma del array que
termina en a[i-1]
                invariant: 0 ≤ fin ≤ s
                initialization: i , s, fin = 0,0,0;
```

```
            termination: i == N
            variant: N-i
            progress{
                fin = fin + a[i];
                if(fin < 0) fin = 0;
                if(fin > s) s = fin;
                i = i+1;
            }
        }
}
```

6.3.Rotación de Array

Supongamos que tenemos el siguiente *array* de enteros:

{3,1,6,2,15,6,18}

Imaginemos que el *array* en lugar de una estructura fija es una especie de cinta. Es decir, si estamos en la última posición y avanzamos un lugar volvemos al inicio. Si rotamos el *array* una posición lo que estamos haciendo es mover cada elemento un lugar hacia la izquierda, y el primer elemento lo colocamos en la última posición. Quedaría entonces de la forma:

{1,6,2,15,6,18,3}

Si rotamos el *array* original dos posiciones cada elemento se movería dos lugares hacia la izquierda y los primeros dos elementos ocuparían las dos últimas posiciones, lo que también sería el equivalente a ejecutar la función anterior dos veces consecutivas. El *array* quedaría entonces como:

{6,2,15,6,18,3,1}

Definido esto buscamos la función que permita ejecutar la rotación para cualquier *array* x, una cantidad *k* de posiciones y nos retorna un *array* y que es justamente el *array* x luego de sufrir la rotación.

Postcondición: y es el resultado de aplicar una rotación de k posiciones a la izquierda el array x.

Si prestamos atención podremos notar que la cantidad de posiciones que queremos rotar el *array* coincide con la posición en el *array* original del primer elemento del nuevo *array*. Es decir, si tenemos que rotar 3 veces el *array*, en la tercera posición del *array* original está el elemento que será el primero del nuevo *array* que estamos generando. En la cuarta posición está el elemento que será el segundo del *array* resultante y así sucesivamente.

Array original:
{3,1,6,2,15,6,18}

Array luego de rotación con *k=3:*
{2,15,6,18,3,1,6}

Por lo tanto cada posición del *array* resultante está en el *array* original en la misma posición más *k* posiciones. Una vez que se acaba el *array*, y por lo tanto la posición en y sumado a *k* es mayor que el largo del *array*, alcanza con restarle el largo del *array* a esta operación y nos dará su ubicación en el *array* original.
Por lo tanto podemos redefinir la postcondición de la siguiente forma:

Postcondición: para todo i, tal que 0<=i<N se cumple que y[i]=x[i+k] en caso de que i+k < N y y[i]=x[i+k-N] si i+k>=N

Para resolver este problema lo haremos de la misma forma que los dos anteriores, es decir, utilizando el método de ***Updating***, y **Relajación de constantes** y el ajuste ***Aging*** para derivar las invariantes.

Invariante 1: para todo 0<=j<i se cumple que y[j]=x[j+k] en caso de que j+k < N y y[j]=x[j+k-N] si j+k>=N

Invariante 2: $0 \leq i \leq N$

Para permitir que el parámetro *k* sea cualquier valor, independientemente del largo del *array* vamos a utilizar la función módulo. Por ejemplo, si recibimos un *array* de 10 elementos, y *k* valiera 13, podríamos deducir que rotándolo 10 veces volvería a su estado inicial, por lo que a efectos del *array* resultante, realizar 13 rotaciones es equivalente a realizar solo 3 (13 % 10 = 3). Esta transformación la expresaremos en la inicialización de las variables.

De igual forma que los ejemplos anteriores vamos a definir la inicialización y la terminación de las demás variables que hemos introducido:

Inicialización: k, y, i = k %N, [ ], 0;

Terminación: i == N

Al inicializar *i* en 0 el *array x[0..-1]* es vacío, entonces todos sus elementos cumplen la condición definida en la invariante y ambas invariantes se verifican en su inicialización.

En caso de que *i + k* (la cantidad de posiciones que rotamos el *array*) sea menor que el largo del *array* copiamos en la posición *i* del *array* a retornar el valor del *array* original en la posición *i+k* ya que entra en el *array*.

Si *i* + *k* es mayor o igual al largo del *array* nos vamos a salir del rango permitido, por lo que copiamos en la posición *i+k-N* del *array* resultante el valor del *array* original en la posición *i*. De esta forma la invariante que antes se cumplía para todo valor de *j* menor que *i*, ahora se cumple también para *i* + 1. Esto nos permite aumentar en uno *i* para seguir recorriendo, y que se siga verificando la invariante.

Paso del loop:

if(i+k<N) y[i] = x[i+k];

else y[i] = x[i+k-N];

i = i+1;

Como en las otras recorridas la variante también será:

Variante: N-i;

El algoritmo queda entonces, de la forma:

```
method rotacion (T [ ] x, nat k) : T[ ] y {
        ensures:  y es el resultado de aplicar una rotación de k posiciones a la izquierda el array x
        loop{
                invariant: para todo 0<=j<i se cumple que y[j]=x[j+k] en caso de que j+k < N y y[j]=x[j+k-N] si j+k>=N
                invariant: 0 <= i <= N
                initialization: k,y,i = k%N,[],0;
                termination: i == N;
                variant: N – i;
                progress{
                        if(i+k<N) y[i] = x[i+k];
                        else y[i] = x[i+k-N];
                        i = i+1;
                }
        }
}
```

6.4.Reverso de Array

El objetivo del siguiente problema es realizar el reverso de una porción de un *array* sin crear un *array* auxiliar para resolverlo.
Lo que haremos es definir un método que recibe tres parámetros: un *array* a, y dos enteros, *i* y *d*. El algoritmo que vamos a construir revierte los elementos del *array* a que se encuentren entre las posiciones *i* y *d* del *array*.

Precondición: $0 \leq i \leq d < N$

Postcondición: El array x[i..d] fue revertido.

El reverso de un *array* equivale a leerlo de derecha a izquierda. Por ejemplo, el reverso del *array* {3,10,8,20,5,6} es {6,5,20,8,10,3}, y el reverso de {3,10,8,20,5,6} entre las posiciones 1 y 4 es {3,5,20,8,10,6}. Podemos observar que revertir una porción del *array* es lo mismo que intercambiar elementos comenzando desde los límites definidos, avanzando hacia el punto medio de ellos. Es decir, en el *array* original revertir desde el índice 1 hasta el 4 es equivalente a intercambiar el 10 (posición 1) por el 5 (posición 4) y el 20 (posición 2) por el 8 (posición 3). Si hubiese más elementos seguiríamos ejecutando estos intercambios hasta que los extremos se encuentren o se solapen.

Es fácil anticipar que vamos a resolver este método mediante ***Updating***, porque en cada paso vamos insertando cada elemento en la posición donde corresponde luego de realizada la rotación. Usando la técnica de **Relajación de Constantes** con ***Aging*** podemos sustituir las constantes *i* y *d* por variables (*j* y *k*) que van desde los propios valores originales hasta el valor medio, que es avanzar hasta que sean iguales. Esto deriva en dos invariantes de la forma:

Invariante 1: Las porciones x[i..j-1] y x[k+1..d] fueron revertidas

Invariante 2: $i \leq j \leq k \leq d$

Esto nos permite asumir que todos los elementos del *array* a excepción de los del intervalo *a[j..k]* ya cumplen con la condición, y solamente nos falta recorrer los elementos desde *j* hasta la mitad, y su parte simétrica del otro lado de la mitad, es decir decreciendo desde *k* hasta el valor medio.

Inicialización: j, k = i, d;

Al inicializar las variables en los valores recibidos por parámetros sabemos que la invariante se cumple porque el *array a[i..j-1]* y *a[j+1..j]* serían *subarrays* vacíos, por lo que cumplen el reverso.

Tiene sentido seguir ejecutando el algoritmo siempre y cuando *j* sea menor que *k*. En el momento que ambos se "encuentran" hemos terminado la ejecución y no tiene sentido continuar. Esto se cumple perfectamente si el intervalo que vamos a revertir tiene largo impar. En el caso que el largo sea par eventualmente *j* pasará a ser mayor que *k* sin que ambos sean iguales. Para ese generalizar ambos casos en una misma condición de terminación podemos plantear la condición de la forma:

Terminación: $j \geq k$

Dentro del *loop* simplemente intercambiamos el contenido que hay en cada posición de los índices en la estructura. Previamente la invariante se cumplía para el intervalo *a[i..j-1]* y *a[k+1..d]*. Si intercambiamos el contenido de *k* y *j*, la condición se cumplirá para *a[i..j]* y *a[k..d]* por lo que podemos disminuir *j* y aumentar *k* y que se siga manteniendo la invariante.

Como variante podemos utilizar la variable *k*, que se reduce en cada iteración y tiene una cota inferior definida por la invariante limitante.

Variante: k

```
method reverso (T[ ] a, nat i, nat d)
{
        requires: 0 < i  ≤ d < a.length
        ensures: El array x[i..d] fue revertido
        loop{
                invariant: El array x[i..j-1] y el array x[k+1..d] fueron revertidos.
                invariant: i ≤ j ≤ k ≤ d
                initialization j, k = i, d;
                termination: j ≥ k;
                variant: k;
                progress{
                        intercambiar(a,j,k);
                        j , k = j+1, k-1;
                }
        }
}
```

6.5. ¿Cuál falta?

Supongamos que somos profesores de una clase de 25 alumnos, donde cada uno tiene asignado un número entre el 1 y el 25, a los que pedimos realizar cierta tarea. Las tareas van llegando en el correr de la semana, y nosotros, como nos gusta la programación, cada vez que recibimos una tarea, insertamos el número del estudiante que la entregó en un *array*.
Así, luego de que cinco estudiantes entregaron, nuestro *array* va tomando, por ejemplo, la siguiente forma: {14, 9, 3, 7, 18}. Al llegar la fecha límite, miramos nuestro *array* y vemos que tiene 24 elementos. Lamentablemente, esto significa que un alumno no ha entregado la tarea.

Tenemos 25 números de estudiante, y un *array* de largo 24, por lo tanto sabemos que la tarea de un estudiante cuyo número asignado está en el rango [1..25] está faltando. Vamos a derivar un algoritmo que nos diga quién no ha entregado la tarea que debería.

El algoritmo que vamos a construir recibe como parámetros un *array* con los números de estudiantes que han entregado la tarea, y dos índices *low* y *high*. *Low* y *high* indican el límite inferior y superior del rango de posibles valores de los elementos del *array*. En el caso del problema de los estudiantes, pasaríamos como parámetros *low* = 1 y *high* = 25, porque todos los números de los estudiantes se encuentran comprendidos en ese rango.

Las pre-condiciones del problema son que el *array* de entrada contiene un elemento menos que los contenidos en el intervalo cerrado entre *low* y *high*, que todos los elementos de este *array*

pertenecen al segmento [*low*..*high*] y que el *array* no puede contener elementos repetidos. La postcondición nos dice que el valor devuelto no pertenece al *array*.

Precondición: N == high - low.

Precondición: Todos los elementos de a pertenecen al segmento [low..high]

Precondición: no puede haber elementos repetidos en a

Postcondición: low no pertenece al array a

Siguiendo la idea de búsqueda binaria, queremos cortar el rango [*low*..*high*] (para esta ejemplo es [1..25]) a la mitad, y así poder concluir que nuestro estudiante faltante se encuentra, por ejemplo, en el rango [1..13]. De la misma forma que la búsqueda binaria vamos a resolver este problema con el método de **Reducción** donde achicamos continuamente el rango de valores posibles hasta que solamente encontremos el elemento buscado o solamente quede un elemento que será la solución.

Para derivar las invariantes aplicaremos la técnica **Relajación de constantes**: nuestra invariante esencial establecerá que falta un valor en el intervalo [*low*..*high*], y la limitante establece que ambos se encuentran dentro de los límites del *array* de entrada, y además que el valor de *low* nunca supera a *high*.

Invariante 1: En [low..high] hay un valor que no pertenece al array a

Invariante 2: $0 \leq low \leq high < N$

Utilizando la idea de reducir el rango [1..25] a la mitad, calculamos $\frac{1+25}{2} = 13$. Podemos pensar en nuestro *array* compuesto de dos partes: uno con valores entre 1 y 13, y otro con valores entre 14 y 25. Sabemos que a uno de estos *subarrays* les falta un elemento. Si supiéramos a cuál, continuaríamos buscando en esa porción del *array*, reduciendo así nuestros valores posibles a la mitad.

Si la primera porción del *array* tiene menos de 13 elementos, sabremos que es en esta porción que falta un estudiante, y seguiremos buscando en ella. En caso contrario, si en esta porción tenemos exactamente 13 elementos, sabremos que el estudiante que no entregó la tarea tiene un número entre 14 y 25, y seguiremos buscando solamente en esta porción, ignorando el resto. Haremos estas búsquedas iterativamente hasta que el segmento [*low*..*high*] contenga un sólo elemento, por lo que el algoritmo termina cuando *low* y *high* valen lo mismo, y este valor es el que falta en el *array*.

Terminación: low == high

$$\textit{Inicialización: } mid = \frac{low + high}{2}$$

Aún tenemos pendiente resolver cómo vamos a separar, en cada iteración, los estudiantes cuyo número se encuentre entre 1 y 13 de aquellos cuyo número se halle entre 14 y 25. La idea más simple para esto es crear dos *arrays* auxiliares que van a contener estos elementos una vez que recorremos el *array* de entrada. Durante el recorrido, para cada elemento preguntamos si este es menor o igual al valor medio, y lo agregamos en el *array* que corresponda. Luego, podemos comparar la longitud de estos *arrays*, para determinar en cual *subarray* se encuentra el estudiante faltante, y así en la próxima iteración buscar sólo en dicha porción. De aquí podemos desprender que necesitamos un *loop* dentro del *loop* principal, que recorra los posibles valores y los inserte en el *subarray* correspondiente.

En el *loop* interno creamos dos *arrays*, a los que llamaremos *first* y *second*, y de los que podemos saber que *first* contiene todos los elementos menores o iguales que *mid* que ya hemos recorrido, y *second* el resto de los ya recorridos. No vamos a ahondar en demostraciones que sean evidentes ya que se vuelven sumamente tediosas para *loops* anidados. Con la explicación dada y la ilustración de la corrida a mano de la imagen 6-1 será suficiente para comprender la corrección del algoritmo.

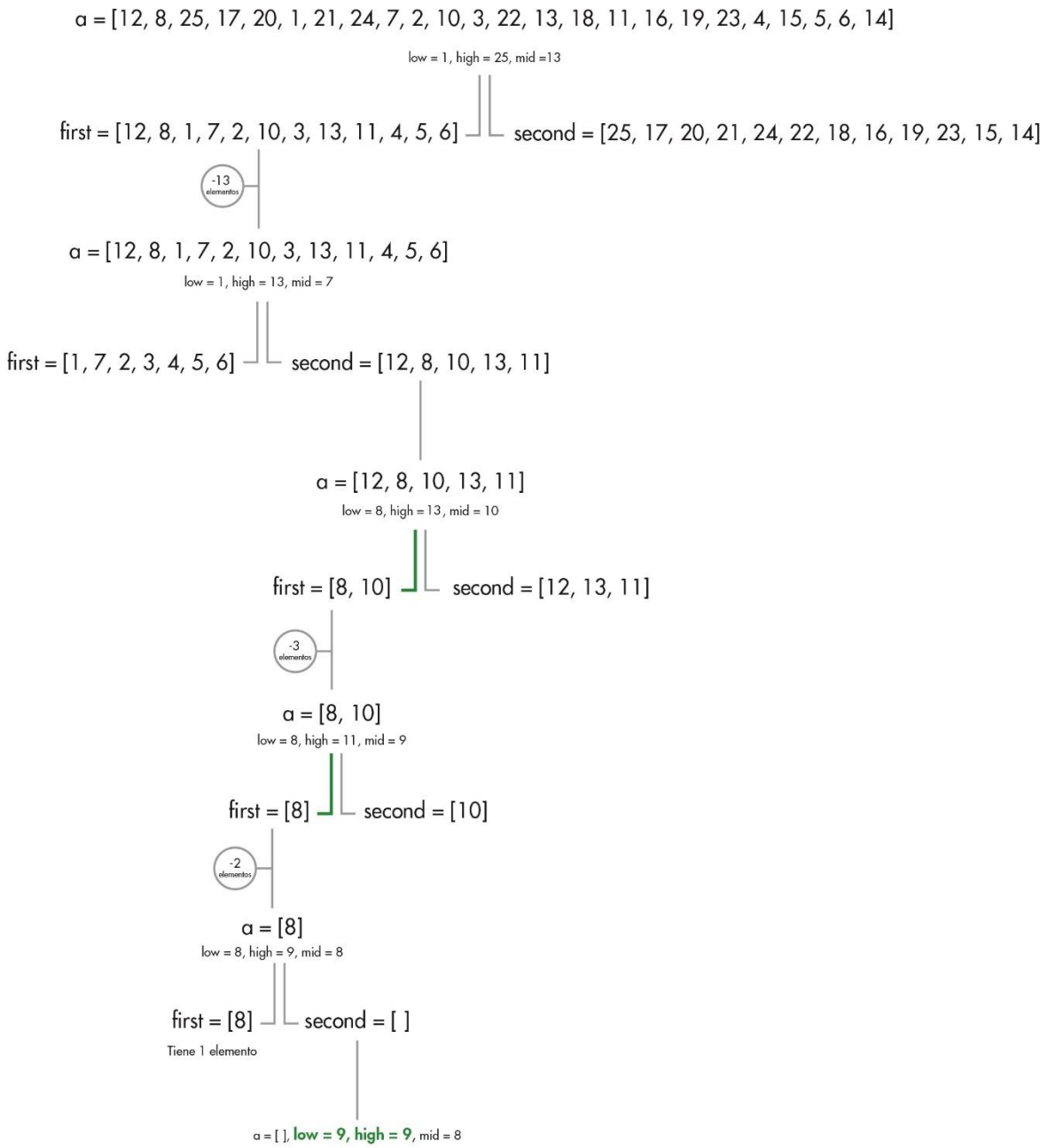


Imagen 6-1: Recorrida a mano para buscar el alumno que no hizo la tarea

La variante del *loop* principal es *high - low*. De la invariante y del hecho que no se cumpla la condición de terminación, sabemos que cada iteración está asociada a un valor positivo de esta variante y que tiene una cota inferior (0, cuando *low==high*, es decir cuando se cumple la condición de terminación), pero esto no es suficiente. La variante, ¿Decrece en cada iteración? Sabemos que *mid* < *high* porque es calculado de la misma manera que en búsqueda binaria general. Luego, si asignamos *high = mid*, aseguramos el decremento de la variable. El otro caso

es análogo: sabiendo que *mid* ≥ *low*, si asignamos *low* = *mid* + 1, *low* está aumentando, por lo que la variante está disminuyendo.

¡Implementemos el algoritmo para averiguar el perro de qué alumno se ha comido la tarea!

```
method cualFalta (nat [ ] a, nat low, nat high): nat low
{
        requires: N == high - low.
        requires: Todos los elementos de a pertenecen al segmento [low..high]
        requires: a no contiene elementos repetidos
        ensures low no pertenece a a
        loop{
                invariant En [low .. high] hay un valor que no pertenece al array a;
                invariant 0 ≤ low ≤ high < a.length
                initialization: mid = (low + high) / 2;
                termination: low == high
                variant: high - low
                progress{
                        loop{
                                initialization:  firstSize, secondSize, i, first, second = 0, 0, 0, [],
                                [];
                                termination: i == N;
                                invariant first [0..firstSize -1] contiene los elementos de a[0..i-1]
                                menores o iguales que mid
                                invariant second[0..secondSize -1] contiene los elementos de
                                a[0..i-1] mayores que mid
                                variant N- i;
                                progress{
                                        if (a[i] <= mid) {
                                                first[firstSize] = a[i];
                                                firstSize = firstSize+1;
                                        }
                                        else {
                                                second[secondSize] = a[i];
                                                secondSize = secondSize + 1;
                                        }
                                        i = i + 1;
                                }
                        }

                if (firstSize < (mid − low) + 1) {
                        high = mid;
                        a = first;
                } else {
                        low = mid + 1;
                        a=second;
```

```
            }
            mid = (low + high) / 2;

            }
        }

}
```

6.6. Programación funcional

Las siguientes funciones provienen de la **Programación funcional**. La programación funcional es un paradigma de programación, como lo indica su nombre, basado en funciones, y no tanto en sentencias imperativas. Una de las características que se busca es crear programas muy compactos, es decir, usando muy pocas líneas de código.

Vamos a desarrollar estas funciones en nuestra notación y analizaremos el uso de invariantes para desarrollar los algoritmos, probar la corrección de estas operaciones y generalizar su comportamiento al máximo posible.

6.6.1. Map

Imaginémonos que somos los dueños de una exitosa empresa. Debido al éxito de la compañía en los últimos tiempos hemos decidido aumentarle el sueldo a nuestra planilla de empleados, pues les estamos agradecidos por su arduo trabajo. Sin embargo queremos realizar el aumento en proporción a su sueldo existente. Aquellos empleados que perciban menos de $1000 (considere la unidad monetaria que desee a efectos del ejemplo) recibirán un 20% de aumento. Aquellos que reciban entre $1000 y $5000 recibirán un 15% de aumento, y aquellos que ganen más de 5000 recibirán un 5% de aumento. A su vez los sueldos son almacenados en una *array* donde cada posición contiene el sueldo que percibe el empleado identificado con ese número. Para simplificar el problema asumimos que todos los empleados tienen un identificador entre 0 y el largo del *array* y que para cada posición del *array* hay un empleado con ese número. Por lo tanto si asumimos la siguiente planilla de empleados:

Identificador	Nombre	Sueldo
0	Federico	$3.500
1	Marcelo	$10.000
2	Carolina	$7.500
3	Damián	$330
4	Sofía	$1200

Tabla 6-1: Planilla de empleados

Nuestro *array* tendrá largo 5, con la siguiente información:

{3500, 10000, 7500, 330,1200}

Fácilmente se podría hacer un algoritmo que recorra todo el *array*, efectúe la función definida y almacene el resultado en otro *array*, que será el final (en este caso sería el *array* con los sueldos ajustados según el aumento definido).
Ahora bien, ¡Qué bueno sería poder aplicar una función que haga todo esto por nosotros! Es decir, simplemente decir que función es la que queremos ejecutar y que nos retorna un *array* con el resultado de haber aplicado esta función para cada elemento del *array* original.
Afortunadamente, esto es lo que hace la función *map*.

La función *map* recibe dos parámetros: una función *func* y el *array* a sobre el que ejecutaremos la función y retorna un *array mapped* con el resultado de la ejecución de la función para los distintos valores.

La postcondición del problema es que todos los elementos del *array mapped* (el *array* que retornamos) deben ser el resultado de ejecutar la función *func* al elemento de igual posición en el *array* a.

Postcondición: mapped está compuesto por todos los elementos de a luego de aplicarles la función func.

Para que esto tenga sentido necesitamos que la función *func* reciba como parámetro un elemento del mismo tipo de datos del que está compuesto el *array* a, y a su vez retorna un elemento de ése mismo tipo de datos, que representa el resultado de aplicar la función dada al valor original. Es decir, si a es un *array* de enteros, la función *func* debe recibir un entero y devolver un entero.

El algoritmo en sí es sencillo y consiste en recorrer el *array* a, ejecutar la función *func* a cada elemento y almacenar el resultado en la misma posición pero en el *array* que se retornará luego.

Nuevamente estamos frente a un algoritmo de ***Updating***, y observando la postcondición podríamos deducir que debemos aplicar la técnica de **Relajación de constantes** con ***Aging*** para derivar el algoritmo.

La invariante esencial nos dice entonces que todos los elementos del *array mapped*[0..i-1] son el resultado de ejecutar la función *func* al elemento de igual posición en el *array* a.

Invariante 1: todos los elementos del array mapped[0..i-1] son el resultado de ejecutar la función f al elemento de igual posición en el array a.

La invariante limitante es:

Invariante 2: $0 \leq i \leq N$

Nos resta definir el paso que haremos dentro de la iteración, la inicialización y la condición de terminación del *loop*. Luego de tantos ejemplos similares es bastante trivial darse cuenta que inicializamos *i* en cero y que irá aumentando su valor en uno, y la condición de terminación es que *i* sea igual al largo del *array*.

Inicialización: i = 0

Terminación: i == N

El paso consiste simplemente en ejecutar la función para el elemento del *array* a en la posición *i* y almacenar el resultado en la posición *i* del *array mapped*, que es el que vamos a retornar. Luego aumentamos la variable *i* en uno. Es bastante trivial darse cuenta que esto sigue verificando las invariantes, por lo que no entraremos en detalles en esta demostración.

Para asegurarnos que el mismo termina debemos definir una variante, que en este caso será *N - i*.

Variante: N - i

Por lo tanto el algoritmo, en nuestra notación será de la forma:

```
method map(T[ ] a, func <T, T>): T[ ] mapped{
    ensures: mapped está compuesto por todos los elementos de a luego de aplicarles la
    función func.
    loop{
        invariant: mapped[0..i-1] está compuesto por todos los elementos de a luego
        de haber pasado por la función func
        invariant: 0 ≤ i ≤ N
        initialization: i, mapped = 0, [];
        termination: i == N;
        variant: N - i
        progress{
            mapped[i], i = func(a[i]), i+1;
        }
    }
}
```

6.6.2. Filter

¿Es usted un apasionado del fútbol? Suponga que el poderoso *Manchester United*, luego de una lamentable temporada, lo contrata a usted como asistente técnico por los conocimientos que posee en el área de la programación informática.
La idea del Gerente Deportivo del club es que usted forme parte del proceso de formación del plantel para el siguiente año, y como en el último año los resultados no fueron los esperados, su primer tarea es determinar qué jugadores deben seguir en el equipo, y cuáles deberán salir al mercado en busca de un nuevo equipo, basándose pura y exclusivamente en las estadísticas de cada jugador.

La base de datos del plantel del *Manchester United* de la última temporada contiene la siguiente información en los primeros 10 registros, como se ilustra en la tabla 6-2.

Nombre	Posición	Goles anotados
Ander Herrera	Defensa	2
Phil Jones	Defensa	1
Luke Shaw	Defensa	3
Michael Carrick	Medio	4
Antonio Valencia	Medio	5
Ashley Young	Medio	3
Angel Di María	Medio	2
Wayne Rooney	Delantero	15
Robin Van Persie	Delantero	11
Radamel Falcao	Delantero	6

Tabla 6-2: Plantel del Manchester United de la última temporada

La persona que lo contrató, espera que usted escriba un algoritmo que determine qué jugadores deben permanecer en el club, según el siguiente criterio:
Si es defensa, debe haber anotado al menos 2 goles en la pasada temporada.
Si es mediocampista, 4 goles al menos.
Si es delantero, 10 goles al menos.

Teniendo en cuenta estos criterios, al finalizar el filtrado de estos 10 jugadores, usted debería entregar la siguiente lista de jugadores que ilustra la tabla 6-3 para que su contrato sea renovado:

Nombre	Posición	Goles anotados
Ander Herrera	Defensa	2
Luke Shaw	Defensa	3
Michael Carrick	Medio	4
Antonio Valencia	Medio	5

Wayne Rooney	Delantero	15
Robin Van Persie	Delantero	11

Tabla 6-3: Plantel del Manchester United para la próxima temporada

Ahora bien, pensemos el proceso sistemático que hemos aplicado a esta lista de jugadores para luego poder traducirlo en forma de algoritmo.

Comenzamos lógicamente por el comienzo de la lista, y nos encontramos con el jugador Ander Herrera. Dado que Ander Herrera es defensa, verificamos si convirtió más de dos goles. Como esta condición se cumple, lo agregamos a una nueva lista de jugadores, inicialmente vacía, que será en definitiva la lista que vamos a entregar en el informe detallando los jugadores que deben continuar en el club. Seguimos recorriendo la lista original, y nos encontramos con Phil Jones: como defensa, se esperaba de él un mínimo de dos goles en el año, pero como convirtió uno solo, no lo incluimos en la lista para el próximo año. Lo sentimos mucho Phil. De esta manera hemos ido depurando la lista inicial de diez jugadores para llegar a la lista final de seis jugadores.

La implementación de este algoritmo se puede hacer utilizando la función *filter*. Esta función recibe cómo parámetros un conjunto de valores de un tipo y una función *func*, que recibe como parámetro un elemento del mismo tipo de datos que los elementos del conjunto y retorna un booleano. El retorno, es otro conjunto con el mismo tipo de objetos que el inicial (jugadores en el ejemplo), que contendrá todos los elementos del *array* original que pasen el filtro, es decir que si invocamos a la función recibida con este parámetro el resultado será verdadero. Además los elementos deben estar en el mismo orden que se encontraban en el *array* original. Llamaremos a este conjunto "*filtered*".

Postcondición: filtered está compuesto por todos los elementos de a que pasan (true) la función func.

Como su nombre lo indica esta función actúa como un filtro, dejando pasar únicamente los elementos que cumplen la condición definida.

Analicemos la aplicación de esta función para nuestro ejemplo.
Inicialmente recibimos un *array* de jugadores, donde cada jugador es un objeto que tiene las propiedades Nombre, Posición y Goles anotados.
Además recibimos las condiciones del filtrado que debemos hacer. Estas condiciones de filtrado son una función que recibe un objeto del tipo Jugador, le aplica los criterios establecidos, y según su retorno determinaremos si el jugador debe continuar en la plantilla o no. Recibimos un jugador como podríamos recibir cualquier objeto, pero lo importante es que la función del filtrado recibe un objeto del mismo tipo de los datos de los elementos de la lista, y devuelve un valor booleano.
La recorrida la vamos a hacer con el método **Updating**: recorremos el *array* original, y para cada elemento, si este "pasa" el filtro lo agregamos al *array* que retornaremos, y de lo contrario no hacemos nada. Repetimos esto hasta finalizar todo el conjunto, y en ese momento el *array* auxiliar se convierte en nuestro retorno.

Vamos a aplicar la técnica **Relajación de constante** para iterar a lo largo del *array*, junto con el ajuste ***Aging*** para validar la invariante al inicializar.

Invariante 1: filtered está compuesto por todos los elementos de a[0..i-1] que pasan (true) la función func.

Invariante 2: $0 \leq i \leq N$

Al igual que en las otras recorridas vamos a inicializar *i* en 0, y el *array filtered* lo inicializamos vacío. De esta forma estamos validando la invariante esencial ya que *a[0..-1]* es vacío.

Inicialización: filtered, i = [ ], 0;

El *loop* termina cuando hemos recorrido todo el *array*:

Terminación: i==N

Debemos demostrar que para cualquier iteración del *loop* seguimos manteniendo las invariantes. En cada iteración del *loop* vamos a preguntar si *a[i]* cumple con la función de filtro, y en caso afirmativo lo agregamos a la lista filtrada. Esto valida nuestra invariante esencial ya que se cumplía para todos los miembros menores a *i*, por lo que se cumplirá para *i* también. Además como simplemente lo agregamos al final del *array* no modificamos el orden en el que aparecen aquellos elementos que cumplen con el filtro. Luego aumentamos *i* en uno.

La variante es *N - i*, al igual que en casi todas las recorridas que realizamos con la técnica relajación de constantes.

Variant: N - i

Pasemos a la implementación de *filter*:

```
method filter(T [ ] a, func <T, boolean>) : T[ ] filtered{
        ensures: filtered está compuesto por todos los elementos de a que pasan (true) la
        función func.
        loop{
                invariant: filtered está compuesto por todos los elementos de a[0..i-1] que
                pasan (true) la función func.
                invariant: 0 ≤ i ≤ N
                initialization: i, filtered = 0, [ ];
                termination: i == N;
                variant: N - i
                progress{
                        if(func(a[i])) filtered.Add(a[i]);
                        i = i+1;
                }
        }
}
```

6.7. Cálculo del cubo con sumas

Este problema consiste en calcular el cubo de un número natural solamente utilizando sumas. La postcondición del problema es: $x = N^3$.
Vamos a resolver este problema con el método **Updating**, llevando un acumulador parcial del cubo. Aplicando la técnica **Relajación de constantes** para la constante *N* (parámetro), obtenemos la siguiente invariante esencial: $x = n^3$, y la limitante $0 \leq n \leq N$, teniendo como condición de terminación del *loop n == N* y la variante *N - n*. Asignamos como valores iniciales *n=0* y *x=0* validando así las invariantes previo a ingresar al *loop*.

Hasta aquí todo parece un simple problema que se resuelve por relajación de constante igual que muchos de los que ya vimos. Pero el truco aparece en el paso del *loop*. En cada paso, vamos a incrementar el valor de *n* en una unidad, como lo hacemos habitualmente en relajación de constante, y *x* digamos que toma el valor de una expresión *E*, que vamos a calcular. Si previo al paso del *loop x* valía n^3 (por la invariante esencial), al aumentar *n*, *x* debe valer $(n+1)^3$, lo que equivale a $E = n^3 + 3n^2 + 3n + 1$. Es decir, para obtener el valor de *x* en cada paso, debemos sumar $3n^2 + 3n + 1$ a su antiguo valor. Esta expresión no puede ser calculada solamente en términos de *n* y *x* utilizando solo sumas, como exige el problema. Por ello, introducimos una nueva variable que representa este valor y fortalecemos el invariante agregando un término que mantenga la invariancia de la nueva variable: $y = 3n^2 + 3n + 1$es nuestra nueva invariante.

En este momento contamos con tres invariantes: las dos que dedujimos aplicando relajación de constante, y la recién obtenida. Dado que y es una nueva variable de nuestro problema, debemos determinar su inicialización, al igual que con todas las variables, y su evolución en cada paso del *loop*. La inicialización es trivial: *y=1*, dado que el valor inicial de *n* es 0. ¿Y qué valor toma y en cada paso del *loop*? De manera análoga a lo que hicimos con *x*, determinamos que *y* tomará el valor de una expresión *F*. Si previo al paso del *loop*, *y* valía $3n^2 + 3n + 1$, y en el paso *n* se incrementa en 1, ahora y debe valer $3(n+1)^2 + 3(n+1) + 1$, expresión que si desarrollamos obtenemos que vale $3n^2 + 6n + 3 + 3n + 1$ 3n2 + 6n +3 + 3n + 3 +1, lo que es equivalente a: $y + 6n + 6$.
Aplicando nuevamente la misma estrategia, introducimos z = 6n + 6 y fortalecemos el invariante, agregando la invariancia de *z* a la misma. Podemos observar que el valor inicial de *z* debe ser 6. Ahora, ¿Qué valor toma *z* en cada paso? Apliquemos el mismo razonamiento que para las variables *x* e *y*. Sabemos que previo al paso del *loop z* valía *6n+6*, y en el paso *n* se incrementa en 1, por lo que *z* pasa a valer *6 (n+1)+6*, lo que equivale a *6n + 6 +6*, o también a *z+6*.

Acabamos de aplicar una técnica llamada **Fortalecimiento de invariantes** [4] [8] para completar el proceso de derivación comenzado con Relajación de constantes. Esta técnica consiste en plantear la forma del ciclo del *loop* dejando algunas expresiones sin resolver, lo cual puede pensarse como ecuaciones con algunas variables como incógnitas. Luego calculamos para "despejar" las incógnitas. Este proceso puede dar lugar a nuevas expresiones las cuales no admitan una expresión simple en términos de las variables de programa. Una forma de resolver esta situación es introducir nuevas variables que se mantengan invariantemente igual a algunas de estas sub expresiones. Esto resuelve el problema al tiempo que aparece la necesidad de mantener las nuevas invariantes.

Con las invariantes derivadas inicialmente sumadas a las que dedujimos en este proceso de fortalecimiento, estamos en condiciones de escribir el algoritmo que resuelve este problema.

```
method CuboConSumas (nat N): nat x
{
        ensures x = N^3
        loop{
        invariant x = n^3
        invariant 0 ≤ n ≤ N
        invariant y= 3n^2 + 3n + 1
        invariant z = 6n + 6
        initialization:n, x, y, z = 0, 0, 1, 6;
        termination: n == N
        variant: N - n
                progress{
                        n, x, y, z = n+1 , x+y , y+z , z+6;
                }
        }
}
```

6.8. Secuencia de Fibonacci

La definición de la secuencia de Fibonacci es:

$$fib\ (0) = 0$$

$$fib\ (1) = 1$$

$$fib\ (n+2) = fib(n) + fib(n+1),\ para\ cualquier\ n\ natural.$$

Vamos a derivar un algoritmo que dado un número natural, calcule su valor en la secuencia Fibonacci. La postcondición será $x = fib\ (N)$. Nuevamente el método de resolución que aplicaremos será ***Updating***.

Derivamos las invariantes del algoritmo mediante la aplicación de la técnica **Relajación de constante**, para la constante *N*, obteniendo las siguientes invariantes:

$$Invariante\ 1:\ x = fib(n)$$
$$Invariante\ 2:\ 0 \leq n \leq N$$

Teniendo como condición de salida del *loop* $n == N$, además de la habitual variante $N - n$.
La inicialización es trivial: $n=0$ y $x=0$ de manera de validar las invariantes antes de ejecutar la primera iteración del *loop*.

En el paso del *loop*, incrementamos *n* en una unidad, por lo tanto si *x* valía *fib(n)*, ahora debe valer *fib(n+1)*, la cual no puede expresarse fácilmente en términos de *r* y *n*. Por ello, seguimos el mismo camino que en el cálculo del cubo, aplicando **Fortalecimiento de invariantes**, y creamos una nueva variable equivalente a *fib(n+1)*, y la agregamos a la invariante:

Invariante 3: y = fib(n+1).

El valor inicial de la nueva variable *y* es 1, ya que *n* inicialmente vale 0, y *fib(1)=1* por definición.

Ahora debemos determinar qué sucede con y en cada iteración. Veamos, si *y* valía *fib(n+1)* antes de entrar al *loop*, con el incremento de *n* debería valer *fib(n+2)*. Por definición de la secuencia de Fibonacci, *fib (n+2) = fib(n) + fib(n+1)*, lo que es igual a *x+y*.
Por lo tanto, la asignación simultánea: *x, y, n= y, x+y, n+1* preserva las tres invariantes y hace válido nuestro algoritmo:

```
method Fibonacci(nat N): nat x
{
        ensures x = fib(N)
        loop{
                invariant x = fib(n)
                invariant 0 ≤ n ≤ N
                invariant y=fib(n+1)
                initialization:n, x, y = 0, 0, 1;
                termination: n == N
                variant: N - n
                progress{
                        x, y, n= y, x+y, n+1;
                }
        }
}
```

6.9. Criba de Eratóstenes

Vamos a comenzar definiendo algunas cuestiones útiles referentes a los números primos. Lo haremos de manera informal y aceptaremos algunos conceptos que son de público conocimiento.

El primer concepto a definir es el de un número primo.

Definición 1: i es primo si y sólo si es mayor o igual a 2 y los únicos números que lo dividen son 1 e i.

Para que esté completa la definición tenemos que especificar a qué nos referimos cuando decimos que un número divide a otro.

Definición 2: a divide a b si y sólo si existe un c natural tal que a×c = b

El algoritmo de la Criba de Eratóstenes recibe un entero *n* y retorna todos los números primos entre 2 y *n*.

Precondición: n debe ser mayor a 2.

Postcondición: el array a retornar contiene todos los números primos entre 2 y n

El *array* a retornar lo vamos a representar como un *array* de booleanos, de tamaño *n*, donde el valor en cada posición es *true* si el valor correspondiente es primo y *false* en caso contrario.

Podemos reformular la postcondición usando las definiciones que acabamos de introducir para que nos ayude a derivar la invariante.

$$\textit{Postcondición: primos[k] es true} \leftrightarrow k \textit{ es primo } \forall k,\ 0 \leq k \leq n$$

Utilizando la definición 1:

$$\textit{Postcondición: primos[k] es true} \leftrightarrow k \geq 2 \textit{ y los únicos números que lo dividen son 1 y } k\ \forall k,\ 0 \leq k \leq n$$

Utilizando la definición 2:

$$\textit{Postcondición: primos[k] es true} \leftrightarrow k \geq 2 \textit{ y } \nexists j,a/\ j \times a = k,\ 1 < j < k,\ 1 < a < k,\ \forall k/\ 0 \leq k \leq n$$

Lo vamos a resolver mediante ***Updating***, ya que buscamos el conjunto de todos los primos en ese rango, y no uno en particular. Si el algoritmo exigiera un número primo en particular podríamos intentar un acercamiento mediante reducción, pero no es el caso.

Vamos a crear un *array* con todos los números entre 2 y *n*, que es el que vamos a retornar posteriormente. Luego recorremos este *array* desde el 2, y multiplicamos cada elemento por todos los enteros desde 2 en adelante mientras el resultado del producto no supere a *n*. En las posiciones de los resultados obtenidos de cada producto asignamos *false*, debido a que sabemos que no son primos, hasta que finalmente hayamos recorrido todo el conjunto. En resumen, lo que hacemos es eliminar todos los que no son primos de la tabla, y el resultado son los números primos.

Pongamos como ejemplo el número $n = 20$. Sabemos que el 1 no es primo, por lo que este vale *false*. Luego comenzamos la ejecución del algoritmo desde el 2. $2 \times 2 = 4$ por lo que el tachamos el 4. $2 \times 3 = 6$por lo que también tachamos este número, así como $2 \times 4 = 8$, ósea que este número también estará cruzado, y así sucesivamente hasta llegar al 20.

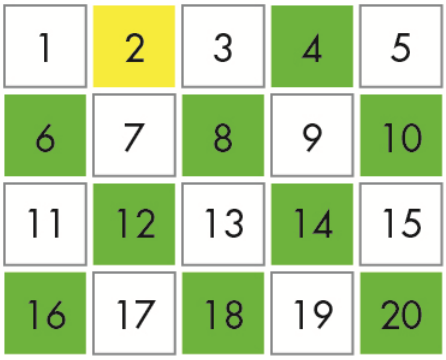


Imagen 6-2: Criba de Eratóstenes. El amarillo representa la posición que estamos analizando y el verde los elementos que ya fueron marcados como compuestos.

Luego haremos el mismo procedimiento para el 3.

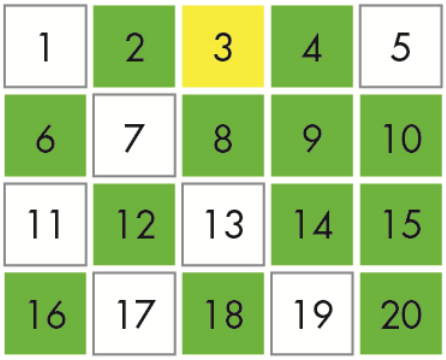


Imagen 6-3: Criba de Eratóstenes

Completaremos el procedimiento para los números restantes hasta arribar al 20. Ahí es donde finalmente tendremos la criba completa, y los números que restan son primos.

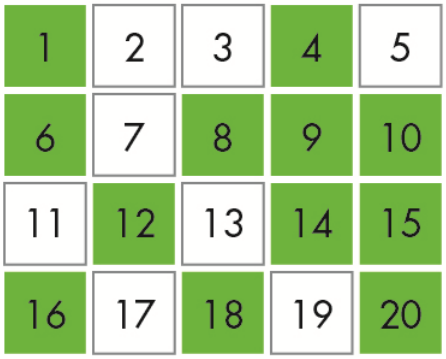


Imagen 6-4: Criba de Eratóstenes completa. Los números en verde son compuestos, por lo que los que están en blanco son primos.

Vamos a aplicar **Relajación de Constantes** a partir de la postcondición que hemos derivado:

$$Invariante\ 1: primos[i]\ es\ true \leftrightarrow \nexists j,a/\ j \times a = i,\ 1<j<i,\ 1<a<i$$

$$Invariante\ 2: 1< i \leq n$$

Además de las condiciones obtenidas a partir de la relajación de constantes vamos a agregar una invariante más.

Invariante 3: $\forall\ a, b\ /\ 1<a<i,\ 1<b<i,\ primos\ [a \times b]=false$

Esta invariante nos aporta información muy valiosa, ya que nos dice que al llegar a un elemento *i* todos los números anteriores (***Aging***) ya fueron multiplicados entre sí, por lo que nos asegura que si el elemento está en true es porque es primo, ya que si no habríamos encontrado dos números cuyo producto es *i*.

Vamos a inicializar *i* en 2, ya que es el primer número posible según la precondición. Sabemos que el 0 y 1 no son primos por lo que vamos a inicializar sus valores en *false*. Además vamos a asumir que todos los números que aún no recorrimos (los mayores que *i*) son primos, hasta que encontremos evidencia que indique lo contrario.

Inicialización 1: i, primos, primos[0], primos[1], primos[2..n] = 2,[ ],false,false,true

Podemos optimizar nuestra condición de terminación. Debido a que estamos multiplicando números menores que *n* entre sí solamente tenemos que ejecutar hasta $\sqrt{n}$ ya que a partir de ahí todos los productos estarán fuera del *array* y su resultado no interesa. Esto también nos modifica nuestra invariante 2.

Terminación 1: $\sqrt{n}$

Invariante 2: $1<i \leq \sqrt{n}$

Al llegar a una posición cualquiera sabemos que el elemento va a ser primo solamente si está en true, por lo que no debemos hacer nada con el elemento *primos[i]*. Esto no quita que debamos mantener las invariantes. La invariante 3, que hemos introducido, nos asegura que todos los números menores a *i* ya fueron multiplicados entre sí y su resultado fue indicado en la tabla.

Por lo tanto lo que debemos hacer es multiplicar *i* por todos los valores mayores a 1 y menores que $\frac{n}{i}$, y marcarlo asignando *false* en la ubicación correspondiente al producto en el *array* resultante. Para esto debemos agregar una segunda recorrida, que la hacemos con un índice *j* que comienza en 2 y se ejecuta hasta $\frac{n}{i}$ porque $\frac{n}{i} \times i$=n. En cada paso de esta segunda iteración sabemos que todos los índices menores ya fueron multiplicados entre sí. Por lo que ya hemos definido lo siguiente acerca del *loop* interior.

Invariante 4: $1<j \leq \frac{n}{i}$

Invariante 5: $\forall m < j,\ primos\ [m \times j]=false$

Inicialización 2: j=2;

Terminación 2: $j== \frac{n}{i}$

En cada paso multiplicamos *i* por *j* y aumentamos *j* en uno, para asegurarnos de seguir manteniendo la invariante.

Paso del loop: primos[i×j],j = false, j+1;

Como aumentamos *j* en cada paso y esta tiene una cota superior definida por la condición de terminación 2 podemos deducir que nuestra variante para el *loop* interno será.

Variante 1: $\frac{n}{i} - j$

Una vez terminado el *loop* interno podemos aumentar la variable *i* y aun así mantener la invariante, porque hemos multiplicado *i* con todos sus predecesores. Ya que aumentamos *i* en cada paso y éste también tiene una cota superior derivamos fácilmente nuestra variante que será:

Variante 2: $\sqrt{n} - i$

method cribaDeEratostenes (int: n) : boolean[ ] primos{
requires: n>2
ensures: primos[k] es true ↔ k ≥2 ∧ ∄j,a/ j×a=k, 1<j<k, 1<a<k, ∀k/ 0 ≤ k ≤ n
***loop*{**
invariant: primos[i] es true ↔ ∄j,a/ j×a=i, 1<j<i, 1<a<i
invariant: 1<i≤$\sqrt{n}$
invariant: ∀ $a, b / a < i, b < i$, primos[a×b]=false
initialization: i, primos, primos[0], primos[1], primos[2..n] = 2,[ ],false,false,true
termination: i==$\sqrt{n}$
variant: $\sqrt{n} - i$
progress{
***loop*{**
initialization: j=2;
invariant: 1 < j ≤ $\frac{n}{i}$
invariant: ∀ $m < j$, primos[m×j]=false
termination: j==$\frac{n}{i}$
variant: $\frac{n}{i} - j$
progress{
primos[i×j], j = false, j+1;
}
}
i = i+1
}
}
}

7.Programación Dinámica

La Programación Dinámica es una técnica para la derivación de algoritmos que permite expresar algunos algoritmos que típicamente son recursivos de forma iterativa. Esto mejora significativamente los órdenes de los mismos, ya que no generan árboles de decisión. Para poder hacerlo, este tipo de algoritmos se apoyan en soluciones de problemas ya calculados previamente para construir la solución final. Este tipo de razonamientos se asemeja mucho a lo que hemos visto con las invariantes, por lo que éstas son muy útiles para resolver problemas mediante la programación dinámica. Generalmente utilizamos invariantes que nos den información acerca de lo computado previamente que nos sirve luego para construir nuestra solución final. Es por esto que las invariantes resultan muy útiles para derivar este tipo de algoritmos.

7.1.Segmento de Ceros

Supongamos que tenemos un *array* de enteros. Consideramos un segmento de ceros como un conjunto de ceros contiguos en el *array* original.

Por ejemplo, si nuestro *array* es {0,0,0,1,3,0,0,4}, un segmento de ceros sería el *array* {0,0,0} , conformado por las tres primeras posiciones del *array* original. También lo serían las primeras dos posiciones (incluidas las combinaciones de ellos), y cualquier *array* conformado por alguno de los ceros.

En este caso estamos buscando el mayor segmento de ceros. El uso de la programación dinámica en este caso nos brinda dos ventajas claras:
Nos permite calcular el máximo segmento de ceros comenzando en cada una de las posiciones.
Nos permite hacerlo en tiempo lineal, es decir, con una única recorrida.

El objetivo es obtener el mayor segmento de ceros de un *array* recibido por parámetro. La solución al problema consistirá en un *array*, que indique donde finaliza el mayor segmento de ceros que comienza en cada una de las posiciones del *array*. Para el *array* propuesto, el mayor segmento de ceros comenzando en la posición 0 termina en la posición 2 (consiste en los primeros tres ceros). Para la posición 1 y 2 el resultado es el mismo (corresponden a los segundos dos ceros, y el último cero respectivamente). Para las siguientes posiciones no hay segmento de ceros, ya que ellos mismos no son cero, por lo que indicamos esto con -1 en esa posición. Asumimos que el *array x* es el original, y devolveremos un *array* y con las soluciones para todas las posiciones.

Por ejemplo, para el ejemplo planteado el *array* resultante es:

Posición	0	1	2	3	4	5	6	7
Array x	0	0	0	1	3	0	0	4
Array y	2	2	2	-1	-1	6	6	-1

Tabla 7-1: Ejemplo se segmento de ceros

Poscondición: y[i]=k si x[k] es el último 0 del segmento que comienza en x[i].

Si x[i] != 0, y[i]= -1, para todo i, 0 ≤ i < N

Este problema se resolverá mediante ***Updating***, porque tenemos que recorrer toda la estructura y acumular un resultado parcial que iremos actualizando (el *array y*).

Vamos a derivar la invariante mediante **Relajación de constantes**, con ***Aging***, pero con una pequeña modificación. Vamos a recorrer de derecha a izquierda del *array* y nos apoyaremos en resultados ya calculados. Recorremos en el sentido opuesto al que lo hacemos tradicionalmente porque la información que nos interesa está a la derecha del elemento que estamos analizando (nos interesa la ubicación del elemento que termina un segmento de ceros), por lo que necesitamos que dichos valores ya hayan sido computados.

Invariante 1: y[j]=k si x[k] es el último 0 del segmento que comienza en x[j].

Si x[j] != 0, y[j]= -1, para todo j, i < j ≤ N-1

Invariante 2: -1 ≤ i ≤ N - 1

Debido a que vamos a recorrer de derecha a izquierda del *array* la inicialización y terminación del algoritmo es ligeramente distinto a los anteriores que hemos derivado usando esta técnica.

Inicialización: i = N-1;

Terminación: i < 0;

Como inicializamos *i* en *N-1* la invariante se verifica para el segmento *y[N..N-1]* que es un segmento vacío del *array* por lo que cumple la invariante. Como estamos recorriendo el *array* de derecha a izquierda el último valor que puede tomar *i* es 0, que es el elemento que está más a la izquierda en el *array*. Por lo que cuando el índice sea menor que cero no es necesario recorrer más.

Supongamos que el elemento de *x* que estamos investigando es el cero que se encuentra en la posición 5 del *array* de ejemplo. Como el elemento a su derecha también es cero, ambos pertenecen al mismo segmento de ceros, ya que son contiguos. Por lo tanto sabemos que ambos finalizan su segmento en la misma posición. Debido a la invariante esencial sabemos que ya hemos calculado el valor para el elemento de la posición 6, así que simplemente debemos copiarlo.
Si el elemento a la derecha del que estamos investigando no es cero estamos frente al final de algún segmento de ceros, por lo que asignamos *y[i] = i* indicando que aquí es donde termina el segmento. En caso de que el elemento no sea cero simplemente asignamos *y[i] = -1* para indicar esto. Por último para un caso borde, si el elemento que estamos analizando es el último, no existe una posición a su derecha, con lo cual de ser cero debemos marcar que aquí termina este segmento indefectiblemente.

Todos los casos mantienen la invariante, ya que se cumplía antes de ingresar al *loop*, y luego de ejecutar las instrucciones sabemos dónde termina el segmento de ceros que comienza en *x[i]*, en caso de que sea cero. En todos los casos debemos disminuir el valor de *i* en uno para avanzar el algoritmo, por lo que *i* será también nuestra variante.

Variante: i

```
method segmentoDeCeros ( int [ ] x ): int [ ] y{
        ensures: y[i]=k si x[k] es el último 0 del segmento que comienza en x[i]. Si x[i] != 0,
        y[i]= -1, para todo i, 0 i < N
        loop{
                invariant: y[j]=k si x[k] es el último 0 del segmento que comienza en x[j]. Si
                x[j] != 0, y[j]= -1, para todo j, i < j ≤ N-1
                invariant: -1 ≤ i ≤ N - 1
                initialization: i = N - 1;
                termination: i < 0
                variant: i
                progress{
                        if(x[i] != 0) y[i] = -1;
                        else if((x[i]==0)&&(i==N -1)) y[i] = i;
                        else if ((x[i] == 0)&&(x[i+1] != 0)) y[i] = i;
                        else y[i] = y[i+1];
                        i = i-1;
                }
        }
}
```

7.2.La meseta más larga: Versión 2

Previamente vimos el ejemplo de la meseta más larga, donde buscamos el mayor segmento de un *array* que contiene elementos iguales. Vamos a resolverlo nuevamente, pero esta vez mediante programación dinámica. Utilizaremos el mismo *array* de ejemplo para graficar el problema:

a = {1,1,2,2,2,3,3,3,3,4,5,5}

Análogamente a lo que acabamos de hacer en el problema del segmento de ceros, lo que buscamos es devolver un *array* en el que cada posición *i* indique el índice del *array* original en el cual termina la mayor meseta que comienza en *x[i]*, siendo x el *array* original. En nuestro ejemplo el *array* resultante sería como observamos en la tabla 7-2:

Posición	0	1	2	3	4	5	6	7	8	9	10	11
***Array* x**	1	1	2	2	2	3	3	3	3	4	5	5
***Array* y**	1	1	4	4	4	8	8	8	8	9	11	11

Tabla 7-2: Ejemplo de la meseta más larga

Usando el conocimiento del dominio de nuestro problema nuestra postcondición queda de la forma:

Postcondición: y[j] = k si y solo si x[k] es el final de la meseta que comienza en x[j], para todo j, $0 \leq j < N$.

En cuanto a la precondición, es la misma que en el problema de la meseta más larga visto previamente.

Precondición: x está ordenado de manera ascendente.

Nuevamente vamos a resolverlo utilizando el método ***Updating*****.** Usando la técnica de **Relajación de constantes,** junto con ***Aging*** derivamos las invariantes:

Invariante 1: y[j] = k si y solo si x[k] termina la meseta que comienza en x[j], para todo j, $i < j \leq N - 1$

Invariante 2: $-1 \leq i \leq N - 1$

La inicialización de i, y la condición de terminación del *loop* serán las mismas que en el ejemplo anterior ya que aquí también recorremos de derecha a izquierda.

Inicialización: i = N-1;

Terminación: i < 0;

Tomemos por ejemplo la posición 7 del *array* original. El elemento que está a su derecha contiene el mismo valor, que es 3. Por lo tanto ambos pertenecen a la misma meseta, y la posición donde termina la meseta que comienza en estos valores es la misma.
La meseta que finaliza en la posición a la derecha del elemento que estamos analizando ya la sabemos por la invariante esencial, por lo tanto copiamos su valor en el *array* y. Si el valor es distinto al de su derecha, la posición que estamos analizando finaliza una meseta, por lo tanto tomamos esa posición como la que finaliza la meseta que comienza en *x[i]*. Por lo tanto asignamos *y[i]=i*. Si el elemento que estamos analizando es el último, podemos afirmar que este elemento representa el fin de una meseta, ya que a su derecha no hay nada por analizar.

Esto preserva la invariante, ya que en todos los casos sabremos en qué posición termina la meseta que comienza en *x[i]*, por lo que podemos avanzar en la recorrida.

Aprovechando que recorremos de derecha a izquierda podemos usar el propio índice como variante, sabiendo que este decrece en cada paso y que la invariante limitante nos indica su cota inferior.

Variante: i

method mesetaDinamica(T[ ] x): int[ ] y{
requires: x debe estar ordenada ascendentemente
ensures: para todo j, 0 ≤ j < N, y[j] contiene el indice de x donde termina la meseta más larga que comienza en el elemento x[j]

```
        loop{
                invariant: para todo i < j < N – 1, y[j] contiene el indice de x donde termina la
                meseta más larga que comienza en el elemento x[j]
                invariant: - 1 ≤ i ≤ N - 1
                initialization: i == N - 1;
                termination: i < 0;
                variant: i;
                progress{
                        if(x[i]==N-1) y[i]=i;
                        else if (x[i]==x[i+1]) y[i]=x[i+1];
                        else y[i]=i;
                        i = i-1;
                }
        }
}
```

7.3.Distancia de Levenshtein

La distancia de *Levenshtein* es una forma de medir la distancia entre dos palabras, o cadenas de texto. Es un número que indica la cantidad mínima de movimientos necesarios (inserción, borrado o sustitución) para que ambas cadenas sean iguales. Por ejemplo la palabra "gato" y "pato" están a una distancia de *Levenshtein* de 1: solamente se necesita un movimiento que es cambiar la 'g' por la 'p' para que las palabras sean iguales. En cambio las palabras 'libro' y 'librería' están a una distancia *Levenshtein* de 4: primero cambiamos la 'o' por la 'e', y luego agregamos las tres letras que faltan, que son la 'r', la 'i', y la 'a'. Si comparamos una cadena cualquiera con una cadena vacía la distancia de *Levenshtein* será el propio largo de la cadena, ya que debemos insertar todos los caracteres a la otra cadena.

Hay 3 operaciones permitidas:

- eliminar un caracter
- insertar un caracter
- reemplazar un caracter por otro

El algoritmo que calcula la distancia de *Levenshtein* se puede realizar de forma recursiva, pero hacerlo de esta forma es sumamente ineficiente, por lo que lo haremos usando programación dinámica.

El algoritmo que vamos a construir debe computar la distancia de *Levenshtein* entre dos cadenas de texto dadas. Esa es la postcondición del problema.

Postcondición: d es la distancia Levenshtein entre s y r

Para resolverlo vamos a implementar el algoritmo de Wagner-Fischer [10].

En primer lugar vamos a representar cada una de las cadenas de texto como *array*s de caracteres, donde la primera posición del *array* representa el caracter vacío, ya que se puede comparar una cadena contra un texto vacío. Adicionalmente tendremos una matriz m donde vamos comparando los caracteres de una palabra con los de la otra y calculamos la distancia para las sub cadenas de texto. Por ejemplo la posición *m[4][7]* contiene la distancia de *Levenshtein* entre *s[0..4]* y *r[0..7]*. Ya que la matriz almacena las distancias parciales, sabemos que en la última posición es donde encontraremos el resultado de comparar las dos cadenas completas. Por lo tanto podemos reescribir nuestra postcondición de la forma:

Postcondición: m[N-1][M-1] es la distancia de Levenshtein de s y r.

N representa el largo de la cadena s, mientras que M representa el largo de la cadena r.

Vamos a derivar el algoritmo mediante el método ***Updating***, donde iremos avanzando en la recorrida de la matriz hasta completarla y si cortamos la recorrida en la mitad, tenemos el resultado de la distancia de *Levenshtein* entre una parte de las palabras por lo que el resultado parcial tiene significado.

Para realizar la recorrida debemos hacer dos *loop*s anidados, que los vamos a recorrer usando los índices *i* y j, que recorren las cadenas *s* y *r* respectivamente. Para lograr esto vamos a aplicar **Relajación de constantes** con ***Aging***. Al ser una matriz debemos implementar una solución con dos *loop*s anidados, por lo que al aplicar la técnica nos va a derivar cuatro invariantes, una esencial y una limitante para cada *loop*. Para el *loop* exterior las invariantes serán:

Invariante 1: m[0..i-1][0..M] contiene la distancia de Levenshtein proveniente de comparar s[0..i-1] y r[0..M]

Invariante 2: $0 \leq i \leq N$

y para el *loop* interno serán:

Invariante 3: m[i][0..j-1] es la distancia de Levenshtein proveniente de comparar s[0..i] y r[0..j-1]

Invariante 4: $0 \leq j \leq M$

Como las dos recorridas son de izquierda a derecha es fácil determinar las condiciones de inicialización y terminación del algoritmo.

Inicialización 1: i = 0;

Terminación 1: i==N

Inicialización 2: j = 0;

Terminación 2: j==M

Gracias a las invariantes 1 y 3 sabemos que todas las posiciones anteriores al elemento que estamos buscando contienen la distancia de *Levenshtein* para los elementos ya recorridos. Particularmente nos interesan 3 de estas posiciones:

- *m[i-1][j]*
- *m[i][j-1]*
- *m[i-1][j-1]*

Por ejemplo, tomemos las palabras “camara” y “cangrejo” como s y r respectivamente, como en la imagen 7-1.

		0	1	2	3	4	5	6	7	8
		-	C	A	N	G	R	E	J	O
0	-									
1	C									
2	A									
3	M									
4	A									
5	R									
6	A									

Imágen 7-1: Distancia de Levenshtein antes de comenzar el algoritmo

Supongamos que estamos comparando las sub cadenas “cama” y “ca”, como en la imagen 7-2, por lo que *i* toma el valor 4 y *j* es 2. Debido a que *s[i]* es igual a *r[j]* no es necesario realizar ninguna operación para convertirlas. La cantidad de operaciones que serán necesaria es equivalente a las necesarias para convertir “cam” en “c”. El resultado de esta operación ya fue calculado y se encuentra en la posición *m[i-1][j-1]*, por lo que en los casos en los que los caracteres que analizamos son iguales copiamos el valor en su diagonal.

		0	1	2	3	4	5	6	7	8
		-	C	A	N	G	R	E	J	O
0	-	0	1	2	3	4	5	6	7	8
1	C	1	0	1	2	3	4	5	6	7
2	A	2	1	0	1	2	3	4	5	6
3	M	3	2	1	1	2	3	4	5	6
4	A	4	3							
5	R									
6	A									

Imagen 7-2:Distancia de Levenshtein entre "Cama" y "Ca"

Asumamos ahora que estamos comparando las sub cadenas “cam” y “cangre”, como en la imagen 7-3, por lo que *i* vale 3 y *j* vale 6. Ya que ‘m’ es distinto a ‘e’ sabemos que tenemos que realizar alguna operación. La primera opción sería sacar una letra de “cam”, por lo que estaríamos comparando “ca” con “cangre” y le agregamos una unidad por haber eliminado la letra ‘m’. El resultado de esta comparación es conocido y se encuentra en *m[i-1][j]*. La segunda opción es agregar una letra en esa posición. Al agregar la letra correspondiente s se transformaría en “camE”. Eso es equivalente a buscar la distancia de *Levenshtein* entre “cam” y “cangr” y agregar una unidad debido a la inserción que realizamos. El resultado de la distancia se encuentra en *m[i][j-1]* que también lo sabemos porque nos lo dice la invariante. La tercera opción es reemplazar el valor de *s[i]* por el valor de *r[j]*. En ese caso es equivalente a la comparación de “ca” con “cangr” y sumarle uno más por la operación realizada. Al igual que en los otros casos, gracias a la invariante sabemos que esta información la tenemos, y que se encuentra en la diagonal, es decir, en *m[i-1][j-1]*.

		0	1	2	3	4	5	6	7	8
		-	C	A	N	G	R	E	J	O
0	-	0	1	2	3	4	5	6	7	8
1	C	1	0	1	2	3	4	5	6	7
2	A	2	1	0	1	2	3	4	5	6
3	M	3	2	1	1	2	3			
4	A									
5	R									
6	A									

Eliminar

Agregar

Sustituir

Imágen 7-3:Distancia de Levenshtein entre "Cam" y "Cangre"

Lo que nos resta por saber es ¿Cuál de las tres opciones deberíamos elegir? La respuesta es muy simple: la menor. Como mencionamos previamente la distancia de *Levenshtein* es la distancia mínima que separa ambas cadenas por lo que tomaremos la mínima de las tres y le agregamos una unidad por la acción que vamos a realizar. De esta forma llenamos el casillero sobre el que estamos parados y podemos aumentar el índice, manteniendo así la invariante.

Terminaremos cuando hemos recorrido ambas palabras, y en la última posición del *array* tendremos la distancia de *Levenshtein* de estas dos palabras. En este caso necesitamos dos variantes, una por cada *loop*.

Variante 1: N - i

Variante 2: M - j

El algoritmo queda entonces de la siguiente forma:

```
method Levenshtein (string s, string r) : int m[N-1][M-1]
{
        ensures: m[N-1][M-1] es la distancia Levenshtein de s y r
```

```
        loop{
                invariant: m[0..i-1][0..M-1] es la distancia de Levenshtein de s[0..i-1] y r[0..M-
1]
                invariant: 0<=i<=N
                initialization: i= 0;
                termination: i==N
                variant: N-i;
                progress{
                        loop{
                                invariant: m[i][0..j-1] es la distancia de Levenshtein de s[0..i] y
                        r[0..j-1]
                                invariant: 0<=j<=M
                                initialization: j = 0;
                                termination: j==M
                                variant: n – j;
                                progress{
                                        if(i==0){
                                                m[0][j] = 0;
                                        }
                                        elseif(j==0){
                                                m[i][j] = 0;
                                        }
                                        elseif(s[i]==r[j]){
                                                m[i][j] = m[i-1][j-1];
                                        }
                                        else{
                                                m[i][j] = Min(m[i][j-1],m[i-1][j],m[i-1][j-1]) + 1;
                                        }
                                        j=j+1;
                                }
                        }
                        i=i+1;
                }
        }
}
```

7.4.El camino a la riqueza

El siguiente problema nos presenta un tablero de $n \times n$, donde cada casillero se asocia con una cantidad (positiva) de dinero. Comenzando desde algún casillero de la fila inferior, el juego consiste en avanzar a otro casillero de la fila siguiente hacia arriba hasta llegar al final. En cada paso hay que escoger uno de los siguientes tres movimientos posibles: moverse hacia el casillero que se encuentra directamente arriba del actual, moverse hacia arriba y una columna hacia la izquierda (siempre y cuando no nos encontremos en la primer columna), o moverse hacia arriba y una columna hacia la derecha, siempre y cuando no nos encontremos en la última columna. La

imagen 7-4 ilustra los posibles movimientos. El jugador recibe la cantidad de dinero correspondiente a cada casillero que ocupa durante el juego. El objetivo es maximizar la cantidad de dinero recolectada.

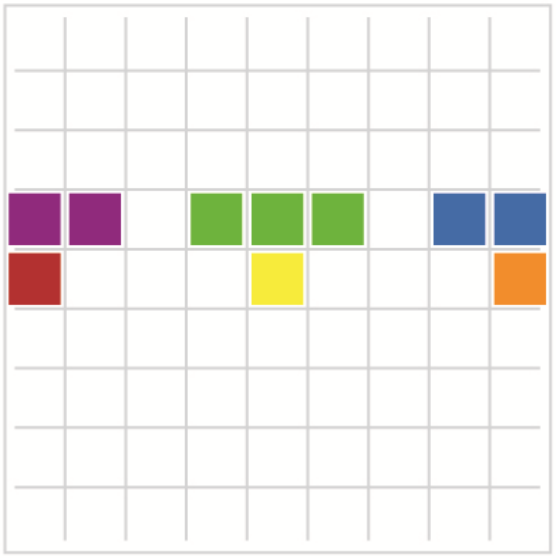


Imagen 7-4: Posibles movimientos. Si la posición actual es el casillero amarillo, se puede moverse hacia cualquiera de los casilleros verdes. Si nos encontramos en el casillero rojo podemos desplazarnos hacia los violetas, y si estamos en la casilla naranja podemos

Supongamos que jugamos en un tablero como el de la imagen 7-5:

2	2	1	1
3	12	1	5
0	0	5	9
8	0	5	4

Imagen 7-5: Tablero de camino a la riqueza

Dado que podemos comenzar en cualquier casillero de la primera fila, vamos a elegir empezar por la primera columna, sumando así 8 unidades de dinero, que representaremos con el símbolo '$'. Ahora tenemos dos movimientos posibles: hacia arriba (recibiendo $0), o hacia arriba y a la derecha (también recibiendo $0). Optemos por la segunda opción. Ahora tenemos tres movimientos posibles. Sin dudarlo vamos a avanzar hacia arriba y sumar $12 a nuestra cuenta, llegando así a $20 acumulados. Luego llegando al final podemos sumar un peso más o dos, y obviamente vamos a elegir el mayor valor, por lo que cerramos el juego con $22 en nuestro haber.

La postcondición del juego es "*m* es la máxima cantidad de dinero que podemos acumular comenzando en la fila de abajo y finalizando en la de arriba, siguiendo las reglas del juego", y no hay precondiciones, ya que el tipo *nat* del tablero establece que en él sólo habrán valores mayores o iguales a cero.

Postcondición: max es el mayor valor que se puede obtener siguiendo las reglas del juego

Como estamos buscando el máximo valor computable dentro de una estructura debemos utilizar el método de ***Updating*** para construir una solución parcial que se vaya actualizando hasta agotar la estructura y entonces llegar a una solución completa.

Podríamos vernos tentados a aplicar **Relajación de constante**, y así decir que la invariante esencial diría algo como "*m* es la máxima cantidad de dinero que podemos acumular comenzando en la fila de abajo y finalizando en la fila *i*, siguiendo los movimientos que permite el juego", pero de hacerlo estaríamos cayendo en un error. ¿Por qué no es correcta esta invariante?

Llamémosle *N-1* a la fila de más abajo, *N-2* a la que la sigue hacia arriba, y así sucesivamente hasta que la primera fila es la fila 0. Podemos hacer válida esta invariante antes de entrar al *loop*, asignando *i=n-1* y m tomando el máximo valor de la fila inferior (8 en nuestro ejemplo).

Hasta aquí vamos bien. Ahora, ¿cómo podemos extender la invariante cuando pasamos a la próxima fila? Podríamos pensar que avanzar hacia el casillero con el valor más alto mantendrá la invariante, pero esto no es cierto. En el ejemplo recién visto, los dos movimientos posibles nos dan un total de $8 (8 + 0), y aparentemente esto es lo máximo que podemos obtener de las primeras dos filas. Sin embargo, si decidiéramos comenzar por la última columna de la última fila (ángulo inferior derecho), podríamos obtener $13 en las primeras dos filas ($4 y después $9). Por lo tanto, es inválido decir que "*m* es la máxima cantidad de dinero que podemos acumular comenzando en la fila de abajo y finalizando en la fila *i*, siguiendo los movimientos que permite el juego". Además, escoger la esquina inferior izquierda como punto de partida, no nos permite extender la invariante ni siquiera a dos filas, por lo que deberíamos comenzar en otro lugar.

Ya que no podemos determinar el camino que maximice nuestra ganancia desde la última fila hasta la primera, una buena alternativa podría ser determinar la mayor cantidad de dinero que podemos recolectar terminando en cada uno de los casilleros del tablero. Si logramos conseguir esta información para todos los casilleros, la máxima ganancia que podremos obtener será el mayor valor que observamos en la primera fila.

Para esto vamos a construir una matriz, llamémosla y, con las mismas dimensiones del tablero del juego, de manera que *y[a][b]* contenga la mayor cantidad de dinero que podemos obtener comenzando en algún lugar de la última fila y terminando en la fila *a* y columna *b*. De aquí podemos desprender que la invariante debe ser que los casilleros de y que ya fueron completados representan el valor máximo que se puede obtener por terminar la recorrida en ese casillero.

Invariante: Todos los elementos entre las filas n-1 y i-1 representan valores máximos por terminar en ese casillero

Para comenzar, podemos calcular los valores de *y* para los casilleros de la última fila, simplemente copiando el valor asociado a estos casilleros. Esto significa que la mayor cantidad de dinero que podemos recolectar comenzando en la última fila y terminando en un casillero específico de la última fila implica comenzar en este casillero.

Para calcular los otros elementos de y, necesitamos dos *loop*s anidados, de la misma forma que en el problema de la distancia de *Levenshtein*. Sabemos que para llegar al casillero en la fila *a* y columna *b*, debemos venir de uno de estos tres casilleros: el que se encuentra debajo de este, el de abajo a la izquierda, o el de abajo a la derecha. Podemos establecer que la mayor cantidad de dinero que obtenemos al llegar a *tablero[a][b]* es el mayor valor que teníamos entre estos tres casilleros, sumado al dinero que obtenemos por pasar por *tablero[a][b]*. Guardamos este valor en *y[a][b]*, y así verificamos la invariante: los casilleros ya completados de la matriz *y* guardan el valor máximo por terminar el juego en dicho casillero del tablero.

Iremos completando la matriz y comenzando desde la última fila (de índice *n-1*), hasta llegar a la primera fila (de índice 0) y así completar toda la matriz. Finalmente procederemos a recorrer la fila 0 para seleccionar el mayor valor en ella, y este valor será la máxima ganancia que se puede obtener al jugar al camino de la riqueza.

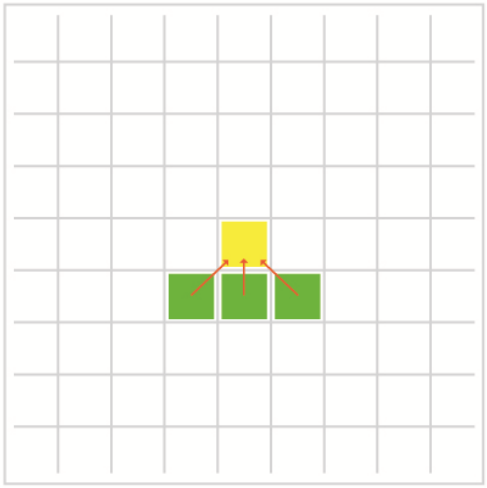


Imagen 7-6: Cambio de paradigma. En lugar de pensar a qué casillero podemos ir, invertimos el problema y pensamos desde qué casillero venimos.

A continuación pasaremos a implementar el algoritmo del juego. La matriz *tablero* representa el tablero del juego donde cada casillero contiene el valor de unidades de dinero que se obtienen por pasar por allí. La matriz *y*, como dijimos, contendrá en cada casillero la mayor cantidad de dinero que podemos obtener empezando en la última fila y llegando hasta este casillero.

```
method CaminoAlaRiqueza(nat [ ] [ ] tablero): nat max{
        ensures: max es el mayor valor que se puede obtener siguiendo las reglas del juego

        //Llenamos la informacion de la ultima fila
loop{
                invariant:y[n-1][i] es la mayor ganancia posible terminando alli
```

```
            initialization: i, j, y=0,0 [ ][ ];
            termination: i >= n
            variant: n - i
            progress{
                y[n−1][i] = tablero[n−1][i]; i++;
            }
        }

        //Llenamos desde la fila n-2 fila por fila hasta la fila 0
        loop{
            invariant: Todos los elementos entre las filas n-1 y i-1 representan valores
máximos por terminar en ese casillero
            initialization: i=n-2;
            termination: i < 0
            variant: i
            progress{
                loop{
                    invariant: y[i][j] es el valor maximo por terminar en [i][j]
                    initialization: j, opcion1, opcion2, opcion3 = 0, 0, 0, 0;
                    termination: j >= n;
                    variant: n - j
                    progress{
                        opcion1 =  y[i+1][j] + tablero[i][j];
                        if (j > 0) opcion2 = y[i+1][j−1] + tablero[i][j];
                        if (j < n − 1) opcion3 = y[i+1][j+1] + tablero[i][j];
                        y [ i ] [ j ] = max(opcion1, opcion2, opcion3);
                        j = j+1;
                    }
                }
                i = i+1;
            }

        }

//Recorremos la fila 0 para obtener el maximo valor de la misma
loop{
            invariant: max es el maximo valor entre y[0][0..k-1]
            initialization: k, max = 0, y [0][0];
            termination: k >= n
            variant: n -k
            progress{
                if (y[0][k] > max) max = y[0][k];
            }
        }

}
```

8.Acertijos y juegos

En esta sección presentaremos juegos y acertijos resueltos mediante la aplicación de invariantes. Para derivar las invariantes que nos ayuden a resolver los mismos no aplicaremos las técnicas antes vistas (Relajación de constante, Eliminación de una condición, etc.), sino que utilizaremos una técnica de abstracción [11] propia de este tipo de ejercicios.

Dado que los acertijos que presentaremos consisten en un estado inicial y una repetición de pasos hasta llegar a cierto estado final, el objetivo es conseguir la abstracción del acertijo a variables matemáticas, y con ellas poder representar el estado inicial, cada paso del acertijo, y el estado final. En cada paso del acertijo debe encontrarse una invariante que nos ayudará a resolverlo.

8.1.Barra de chocolate

Comencemos con un ejemplo muy sencillo. Supongamos que tenemos una barra de chocolate rectangular tradicional, marcada con líneas horizontales y verticales.

El acertijo consiste en encontrar el número de cortes que tenemos que hacerle a la barra de chocolate para que esta quede dividida en los rectángulos más pequeños. Vale aclarar que cada corte se realiza por alguna de las líneas marcadas en la barra y no de manera aleatoria sobre la misma.

En este acertijo la abstracción consiste en olvidarse de la barra de chocolate y modelar esta realidad con variables: El objetivo es encontrar la cantidad de cortes en relación a la cantidad de rectángulos pequeños que conforman la barra. Dado que tenemos que hallar el número de cortes, representaremos la cantidad de cortes realizados en un momento dado con la variable *c*, que inicializamos en 0 ya que inicialmente la barra no tiene ningún corte. La otra variable que debemos modelar representa la cantidad de pedazos en los que está dividida la barra en un momento dado: esta variable *p* la inicializamos en *1* ya que inicialmente la barra es un gran rectángulo.

Inicialización: c, p = 0, 1;

Ya hemos abstraído el problema a dos variables y ya tenemos su inicialización. Ahora debemos modelar cada corte de la barra con estas dos variables. Cada vez que realizamos un corte, la cantidad de pedazos aumenta en uno, al igual que la cantidad de cortes obviamente. Luego, cada corte se podría representar como:

Paso: c, p = c+1, p+ 1;

De aquí podemos deducir que la diferencia entre *p* y *c* se mantiene constante a lo largo de todo el acertijo, por lo tanto *p - c* es la invariante, y vale 1, ya que inicialmente *p - c= 1*.

Invariante: p - c= 1

El juego termina cuando la cantidad de pedazos es igual a la cantidad de rectángulos pequeños que conforman la barra. Si le llamamos *r* a esta cantidad, podemos decir que el acertijo concluye cuando *p == r*.

Terminación: p == r

Si reemplazamos el valor final de p en la invariante tenemos que:
r - c= 1, y por lo tanto *c = r - 1*.

De esta manera hemos llegado a la conclusión del ejercicio y podemos afirmar que la cantidad de cortes que debemos realizar a la barra para reducirla a solamente los rectángulos más pequeños es igual a la cantidad de estos rectángulos menos uno.

Este sencillo ejercicio nos sirve para introducirnos en esta manera de razonar que aplicaremos para los siguientes acertijos.

8.2.Torneo de tenis

La técnica que acabamos de aplicar, y que utilizaremos en estos acertijos consiste en los siguientes pasos:

1) Identificar qué es lo que sabemos y qué no del estado final.
2) Introducir variables, que utilizadas en conjunto modelan el estado del problema en un momento dado.
3) Definir la inicialización de estas variables de manera que modelen el acertijo en el momento inicial.
4) Modelar el paso como una nueva asignación a estas variables.
5) Identificar la invariante del paso.
6) Combinar los pasos anteriores para averiguar lo que no sabemos del estado final, es decir, la solución del problema.

Pasemos al acertijo: un campeonato de tenis se juega con modalidad play-off, es decir: en cada partido juega un jugador contra otro, el ganador sigue en carrera y el perdedor se despide del campeonato. Si en un torneo juegan 2234 jugadores, ¿cuántos partidos se deben jugar hasta que uno se consagre campeón?

¿Qué sabemos y qué debemos averiguar del estado final? Sabemos que queda sólo un jugador en carrera, y debemos averiguar cuántos partidos se han disputado.
Introducimos las variables *p* y *j* que representan la cantidad de partidos que se han disputado y la cantidad de jugadores que siguen en carrera respectivamente.
Antes de iniciar el torneo quedan 2234 jugadores y se han jugado cero partidos.

Inicialización:p, j = 0, 2234;

El paso equivale a la disputa de un partido, y luego del mismo se incrementa en uno la cantidad de partidos, y disminuye en uno la cantidad de jugadores en carrera.

$$Paso: p, j = p+1 , j-1;$$

De aquí podemos deducir que la invariante es:

$$Invariante: p = j_0 - j, \text{ donde } j_0 = 2234,$$

es decir que la cantidad de partidos disputados en un momento dado es igual a la cantidad de jugadores iniciales menos los jugadores que quedan en ese momento.
El torneo termina cuando queda un solo jugador, lo que podemos escribir como:

$$Terminación: j = 1$$

Si reemplazamos el valor final *j* en la invariante, tenemos que:

$$p = j_0 - 1 = 2234 - 1 = 2233$$

Es lógico que si en cada partido hay un eliminado, la cantidad de partidos debe ser igual que la cantidad de eliminados: ósea, la cantidad de jugadores iniciales menos uno. Pero ahora lo hemos demostrado utilizando invariantes.

8.3. Cajas vacías

El acertijo de las cajas vacías plantea lo siguiente: inicialmente tenemos 11 cajas grandes vacías. En un momento dado, se colocan 8 cajas medianas en algunas de estas cajas grandes. No sabemos cuántas cajas grandes fueron llenadas con cajas medianas, pero sabemos que si se colocaron cajas, fueron 8. A su vez, en algunas de estas cajas medianas se colocaron otras cajas chicas, pero nuevamente no sabemos cuántas cajas, ni cuáles fueron rellenadas, aunque sabemos que de haberse rellenado una caja mediana se hizo con 8 cajas chicas exactamente. Luego de estos movimientos quedan 102 cajas vacías (una caja está vacía solamente si no tiene otra caja dentro). El acertijo consiste en hallar cuántas cajas hay en total.

Comencemos por determinar qué sabemos del estado final: sabemos que hay 102 cajas vacías, y queremos averiguar cuántas cajas hay en total. Para modelar esta situación en variables, vamos a necesitar una variable que represente la cantidad de cajas vacías en un momento dado -llamémosla *v* -, y otra que represente la cantidad de cajas totales, a la que llamaremos *t*.

El problema comienza con un total de 11 cajas, todas ellas vacías.

$$Inicialización: t=11, 11;$$

A la hora de analizar el paso, una parte importante de la abstracción de la realidad a las variables, es olvidarse de las cajas grandes, medianas y pequeñas, y sólo pensar en términos de las variables que manejamos, es decir, cajas totales y cajas vacías.
Un paso de este acertijo consiste en colocar 8 cajas dentro de una de ellas, por lo que la cantidad de cajas totales aumenta en 8, y las vacías en 7, ya que se agregaron 8 cajas vacías pero una de las que estaba vacía deja de estarlo.

Paso: t ,v=t+8, v+7;

Debemos hallar la invariante del paso. En los ejemplos de la barra de chocolate y el torneo de tenis se deducía muy fácilmente, pero aquí hay que aplicar un razonamiento matemático un poco más avanzado.
Analicemos las asignaciones a las variables *t* y *v: t* comienza valiendo t_0 (vale 11 pero no viene al caso), luego de un paso valdrá t_0+8, un paso después t_0+ 2 x 8, luego t_0 + 3 x 8, y así sucesivamente. Algo similar sucede con la asignación de *v*. En términos matemáticos, podemos decir que la sucesión de valores de *t* son combinaciones lineales de t_0 y 8, y lo mismo para *v* con v_0 y 7. De aquí podemos deducir que la invariante del paso será también una combinación lineal de *t* y *v*. Esto significa que para ciertos *M* y *N* que debemos hallar, se cumple que *M.t + N.v* permanece invariante en cada paso: es decir, si en la expresión *M.t + N.v* sustituimos *t* y *v* por *t+8* y *v+7* respectivamente (los valores que toman en el paso), esto tiene que seguir valiendo *M.t + N.v*, por definición de invariante:

$$M(t+8) + N(v + 7) = Mt + N\,v$$

$$Factorizando: (Mt + N\,v) + (8M + 7N) = Mt + N\,v$$

$$(M.t + N.v) + (8M + 7N) = (M.t + N.v)$$

$$8M + 7N = 0$$

$$M=7,\ N= -8$$

Habiendo obtenido estos valores, llegamos a que *7t - 8v* es la invariante del paso. Inicialmente tanto *t* como *v* valen 11 (cajas totales y cajas vacías), por lo tanto *7t - 8v* vale -11, y este valor permanece invariante.

Invariante: 7t - 8v = -11

Al finalizar el problema sabemos que contamos con 102 cajas vacías, por lo que:

Terminación: v = 102;

Si reemplazamos el valor final de *v* en la invariante, tendremos que:

$$7t - 8102 = -11 \quad t = 115$$

Y llegamos a que finalmente hay 115 cajas, 102 de ellas vacías.

Cajas vacías: Generalización
Este ejercicio sirve para comprender y repasar los conceptos recién aplicados.

Vamos a repetir el acertijo de las cajas vacías, pero esta vez generalizando las variables: Inicialmente hay *m* cajas, y en cada paso se colocan *k* cajas en algunas de ellas. Finalmente hay *n* cajas vacías. ¿Cuántas cajas hay en total?

Repitiendo el razonamiento que hicimos recién:

Inicialización:v, t=m, m;

Paso: t ,v= t+k, v+k -1;

Invariante: (k-1)t - kv = -m;

Terminación: v = n;

Solución: (k-1)t – k.n = -m

$$t = \frac{k.n-m}{k-1}$$

Para que el problema esté bien formulado *t* debe ser un número entero, por lo que *k .n* debe ser mayor que *m, k* debe valer al menos 2, y *k.n - m* debe ser divisible entre *k-1*.

Otro tipo de juegos
Los acertijos recientemente vistos fueron resueltos mediante la aplicación de algoritmos deterministas. Un algoritmo determinista es aquel que es completamente predictivo si se conocen sus entradas. Dicho de otra forma, si se conocen las entradas del algoritmo siempre producirá la misma salida, y la máquina interna pasará por la misma secuencia de estados. En cada paso de los juegos presentados ya sabíamos lo que iba a suceder y las consecuencias que iba a implicar este paso (un partido de tenis, un corte en la barra de chocolate), y sabíamos que en cada uno de estos pasos nos estábamos acercando un poco más a la solución del problema.

Pasemos a otro tipo de juegos, más elaborados e interactivos. Estudiaremos juegos con turnos, donde en cada turno se puede elegir entre al menos dos opciones, y según la opción elegida y el estado en el que nos encontrábamos previamente, cambiará el avance del algoritmo. Esto es el no determinismo.

8.4.Palitos Ganadores

El juego trata de dos jugadores y una pila de fósforos [5]. Cada jugador se alterna para sacar una cantidad de fósforos de la pila. Esta cantidad puede ser uno o dos fósforos. El jugador que saca el último palito es el ganador.

Asumamos que solamente hay 1 fósforo en la pila y es nuestro turno. Si sacamos el fósforo tenemos certeza que hemos ganado la partida, porque el contrincante no puede hacer ningún movimiento y entonces pierde. Llamamos a este estado como una **posición ganadora.**

Si quedan dos fósforos y es nuestro turno tenemos dos opciones válidas: sacar uno o dos fósforos. Si sacamos dos fósforos no queda ninguno en la pila, y el contrincante no tiene movimientos válidos. Por lo tanto estamos nuevamente frente a una posición ganadora. Esta vez hay una particularidad, porque si decidimos quitar una fósforo en lugar de dos, no habremos ganado (en realidad perdemos, porque el rival tiene la posición ganadora). Entonces definimos una posición ganadora como un estado en el que si tomamos las decisiones correctas es seguro que ganaremos. Si en cambio tenemos tres fósforos en la pila nos enfrentamos a una **posición perdedora**. Si tomamos dos fósforos nuestro rival tendrá uno en la pila en su turno, una posición ganadora. Si

tomamos un fósforo nuestro rival tendrá dos en la pila, nuevamente una posición ganadora (aunque en este escenario todavía tenemos la esperanza que el rival se equivoque). Ósea que la posición perdedora es aquella donde cualquier acción nuestra deriva en una posición ganadora del contrincante y nuestra única alternativa es esperar a que se equivoque si queremos ganar.

Si tuviéramos cuatro fósforos (siendo el último caso particular a analizar) y elegimos tomar un solo fosforo nuestro adversario queda en posición perdedora. Si tomáramos dos quedaría en posición ganadora. Por lo tanto tener cuatro fósforos restantes y siendo nuestro turno es un posición ganadora, porque la victoria depende únicamente de nuestros aciertos. Si continuamos analizando más casos veremos un patrón emergente. Si en nuestro turno hay una cantidad de fósforos en la pila divisible entre tres estamos ante posición perdedora, de lo contrario estamos frente a una posición ganadora.

Derivar el algoritmo no resulta tan difícil con esta información, que es propia del dominio del problema. Lo que tenemos que lograr es que nuestro rival se enfrente a una cantidad divisible entre 3 (una posición perdedora). Por lo tanto si el número de palitos restantes no es divisible entre 3, vamos a sacar la cantidad que lo hagan serlo. Por ejemplo, si quedan 40 fósforos, sabemos que de la división sobra 1. Si sacamos un fósforo nuestro rival se enfrenta a 39 fósforos, número que es divisible entre 3 y no tiene manera de ponernos a nosotros en posición perdedora, porque solamente puede sacar 1 o 2 palitos, dejando el remanente en un número no-divisible entre 3.

A continuación presentamos la implementación de este juego. En cada turno, el jugador 1 tendrá la posibilidad de elegir cuántos palitos desea sacar, por lo que lo llamamos el jugador humano, mientras que el jugador 2 jugará de manera inteligente, intentando siempre quedar en posición ganadora, por lo que lo llamaremos la computadora. Programaremos el juego para que gane la computadora, por lo que exigimos que la cantidad inicial de palitos sea múltiplo de tres y comienza jugando el humano.

Esta es la única precondición, y las postcondiciones son que al final la cantidad de palitos restantes será 0 y sea el turno del humano, lo que significa que ha perdido.

La variable que modela la cantidad de palitos restantes será también la variante, que como veremos explícitamente, decrece en cada turno hasta llegar a 0.
A su vez las invariantes nos aseguran que si estamos en posición ganadora es porque le toca jugar a la computadora, y viceversa.

```
method SticksAndStones(int restantes) : int ganador
{
        requires restantes % 3 == 0
        ensures restantes == 0;
        loop{
                invariant: restantes % 3 == 0  ↔ turno == 1;
                invariant: restantes % 3 ≠ 0  ↔ turno == 2
                initialization: turno = 1;
                termination: restantes == 0;
```

```
            variant: restantes
            progress{
                int tomados;
                if(turno==1){
/*El usuario ingresa cuantos palitos quiere tomar. Los unicos valores posibles son 1 y 2. El valor ingresado se almacena en la variable tomados */
                    tomados = valor ingresado;
                    restantes = restantes - tomados;
                    turno = 2;
                }
                else{
                    tomados = restantes%3;
                    restantes = restantes - tomados;
                    turno = 1;
                }
            }
        }
}
```

8.5. Bidones de agua

Siguiendo con la modalidad de juegos, pasemos a otro juego clásico: Tenemos dos bidones vacíos, uno con cinco litros de capacidad y otro pequeño con tres, y una fuente de agua infinita [5]. Este caso tiene la particularidad que lo que hacemos es diseñar un algoritmo que plantea el juego en sí, y no la solución.

Hay seis movimientos posibles:

1) Vaciar el bidón pequeño volcando su contenido en el suelo.
2) Vaciar el bidón grande volcando su contenido en el suelo.
3) Llenar el bidón pequeño desde la fuente de agua.
4) Llenar el bidón grande desde la fuente de agua.
5) Volcar el contenido del bidón grande en el pequeño.
6) Volcar el contenido del bidón pequeño en el grande.

En cualquiera de los movimientos no podemos desbordar los bidones, y el objetivo del juego es medir exactamente cuatro litros de agua. El algoritmo de este juego funcionará con un simple *loop* que ejecutará solicitando la elección de uno de los seis movimientos, hasta que hayamos conseguido medir los cuatro litros, y así sabemos que cuando el *loop* finaliza, hemos solucionado el problema.

Hay que tener en cuenta que en ningún momento el bidón pequeño puede tener menos de cero litros ni más de tres, y el grande no puede tener nunca menos de cero litros ni más de cinco. Estas condiciones lógicamente serán nuestras invariantes.

Ahora bien, ¿cualquiera de los seis movimientos preserva las invariantes? Veamos:

Los primeros dos movimientos definitivamente las preservan, ya que cero es una medida válida para ambos bidones. El tercero y el cuarto también ya que se llena uno de los bidones a tope sin modificar el contenido del otro. La dificultad parece estar cuando volcamos el contenido de un bidón a otro. Comencemos por el movimiento N°5, en el que volcamos el contenido del bidón grande al pequeño. En este caso, para no desbordar el bidón pequeño, comenzamos calculando la cantidad total de agua que hay en los bidones. Luego la cantidad de agua que quede en el bidón pequeño será el mínimo valor entre tres litros, y el total calculado. Posteriormente actualizamos el valor del bidón grande con lo que había menos lo que volcamos en el pequeño.

Para el sexto movimiento la verificación para impedir que se desborde el bidón grande es análoga: calculamos la cantidad total de agua entre ambos bidones, y establecemos que el bidón grande pasa a tener el mínimo valor entre cinco litros y el total calculado, y luego actualizamos el valor del bidón pequeño según cuánto hemos volcado al grande.

Veamos la implementación de este algoritmo. ¡Intenta descifrar una solución!

```
method BaldesDeAgua( ) : void
{
        loop{
                invariant: 0 ≤ baldeTresLitros ≤ 3 ;
                invariant: 0 ≤ baldeCincoLitros ≤ 5
                initialization: baldeTresLitros , baldeCincoLitros = 0, 0;
                termination: baldeCincoLitros == 4;
                progress{
                        int opcion, combinados, valorAnterior;
                        //El jugador ingresa el número de opción seleccionada, entre la 1 y la 6.
                        if(opcion ==1) baldeTresLitros= 0;
                        else if(opcion ==2) baldeCincoLitros = 0;
                        else if(opcion ==3) baldeTresLitros= 3;
                        else if(opcion ==4) baldeCincoLitros = 5;
                        else if(opcion ==5){
                                combinados = baldeTresLitros+ baldeCincoLitros;
                                valorAnterior = baldeTresLitros;
                                baldeTresLitros= Min (3, combinados);
                                baldeCincoLitros= Max (0, baldeCincoLitros− (3 −
                                valorAnterior ));
                        }
                        else if(opcion ==6){
                                combinados = baldeTresLitros+ baldeCincoLitros;
                                valorAnterior = baldeCincoLitros;
                                baldeCincoLitros= Min (5, combinados);
                                baldeTresLitros= Max (0, baldeTresLitros− (5 − valorAnterior
                        ));
                        }

                }
```

```
        }
}
```

9. Trabajos relacionados y experiencias en la educación

9.1. Trabajos relacionados

A lo largo de este año de trabajo hemos investigado sobre diversos materiales que tratan sobre metodologías formales y programación basada en invariantes, producidos en diferentes partes del mundo. Muchos de estos materiales nos han sido de gran apoyo, ya sea como generadores de ideas, por observar conceptos sobre los que nos interesó profundizar, como también para descartar pensamientos o enfoques que no nos parecieron adecuados para nuestro trabajo.

En esta sección presentaremos los más relevantes de estos trabajos, contando brevemente de qué trata cada uno, y por qué decidimos profundizar o descartar algunas ideas, así como dar una noción del estado de la investigación actual del tema.

Invariants: a Generative Approach to Programming [5]
Daniel Zingaro

Daniel Zingaro introduce este libro contando sus primeras experiencias como estudiante de programación. Relata que para un examen estudió todos los algoritmos que habían construido durante el curso, pero a la hora del examen, le pidieron que desarrolle un nuevo algoritmo y no supo qué hacer. Allí nace su deseo de estudiar el diseño de algoritmos y algunos años después escribe este libro. Este ejemplo ilustra la problemática que estamos intentando resolver, por lo que sentimos que fue un excelente material en el cual basarnos.

Lo primero que nos llamó la atención y decidimos adoptar fue el tono con el que está escrito el texto. Es amigable e informal con el lector, utiliza ejemplos para ilustrar muchos conceptos y es muy agradable de leer. Intentamos adquirir estas cualidades para nuestro trabajo. Adoptamos también el nivel de informalidad en las demostraciones. Si bien son completas y correctas, tanto Zingaro como nosotros, priorizamos que el lector comprenda lo que intentamos decir por sobre la corrección formal. Muchos de los problemas que analizamos y profundizamos pueden ser encontrados en este texto, por lo que en ese aspecto también fue una inspiración para nosotros.

Sin embargo, existe una diferencia en el objetivo de cada uno. El foco de Zingaro está puesto principalmente en garantizar la corrección de los algoritmos presentados mediante invariantes, sin profundizar demasiado en cómo se llega a obtener estas invariantes. Nuestro objetivo, en cambio, está en el cómo llegar a la derivación del algoritmo, sin perder de vista que el mismo debe ser correcto. Se podría decir que utilizamos un estilo similar en las demostraciones pero intentando ser más metódicos en la derivación de algoritmos. Otra diferencia es en el lenguaje de programación. Los problemas planteados en el libro de Zingaro son resueltos en el lenguaje Java, en el que las invariantes son ingresadas como comentarios. Nosotros, en cambio, proponemos una notación propia. Si bien tenemos la desventaja de no poder ejecutar los algoritmos para

observar los resultados, nuestra notación acompaña la forma natural en la que pensamos las soluciones a los problemas y las invariantes tienen un papel protagónico.

***Loop* invariants: analysis, classification and examples** [7]
Carlo A. Furia, Bertrand Meyer, Sergey Velder

Probablemente, junto con el de *Zingaro*, sea el texto que nos ha sido de mayor utilidad a lo largo de nuestro trabajo.

En él se desarrolla una identificación, validación y clasificación sistemática de invariantes aplicados a diversos algoritmos de problemas conocidos. Además analiza patrones para determinar cómo derivar invariantes a partir de las postcondiciones de un problema.

Hemos estudiado y profundizado en varios problemas y técnicas de derivación de este libro, cambiando el foco principalmente de las invariantes, que aquí son expresadas y demostradas en lógica y matemática proposicional, mientras que nosotros las presentamos y explicamos en lenguaje natural. Gran parte de la clasificación de las técnicas de derivación de invariantes fueron tomadas de este texto.

Otro concepto muy útil que extrajimos de este y aplicamos en nuestro trabajo fue el de clasificación de invariantes en esenciales (*essential*) y limitantes (*bounding*).

Este trabajo fue escrito utilizando un lenguaje más formal que el utilizado en el libro de *Zingaro* o en el presente trabajo, pero los conceptos son igualmente válidos.

Una de las principales diferencias de nuestro trabajo con este texto, así como los otros que plantean las técnicas de derivación es que para los autores las propias técnicas son la derivación del algoritmo. Desde nuestra perspectiva las técnicas son una forma muy concreta de derivar invariantes cuando el contexto permite su aplicación. En cambio la derivación conceptual del algoritmo la planteamos cuando definimos el método de solución de problema que usamos para encarar la solución.

Programming: The derivation of Algorithms [4]
Anne Kaldewaij

Kaldewaij es, junto con *Dijkstra*, uno de los referentes en cuanto a la derivación de Algoritmos utilizando métodos y demostraciones formales, y utiliza las invariantes a lo largo de su trabajo. Le encontramos una especial utilidad al capítulo 4: "*General Programming Techniques*".

Aquí el autor explora distintas técnicas para la derivación de algoritmos, que forman también parte de la base e inspiración que usamos para compilar las que presentamos. Él denomina "Tomar conyuntos como invariantes" (traducido del inglés "*Taking conjuncts as invariant*") a lo que nosotros a posteriori llamamos "Eliminación de una condición", "Reemplazar constantes por variables"("*Replacing constantes by variables*") es similar a "relajación de constantes" y por último analiza "Fortalecimiento de Invariantes"("*Strengthening invariants*"), de la misma forma que el ajuste que nosotros tratamos. Analiza también las Invariantes de Cola ("*Tail invariants*"),

pero decidimos no incluirlo porque excede el alcance de nuestro trabajo. Esta última técnica es en realidad, desde nuestra perspectiva, una forma del método de reducción.

Cálculo de Programas [8]
Javier Blanco, Silvina Smith y Damián Barsotti
Universidad Nacional de Córdoba

Este libro cubre una gran cantidad de problemas de la programación, orientados desde la construcción sistemática de programas imperativos y la programación funcional. Es un material muy completo que comienza desde problemas muy elementales como resolución de ecuaciones, presentación de lógica y cálculo proposicional, inducción y recursión, y demás, hasta llegar al capítulo que más nos interesó: **Técnicas para determinar invariantes.**
En dicho capítulo se presentan cuatro técnicas con una pequeña serie de problemas resueltos aplicando cada una de ellas. Lo que notamos de este trabajo fue el uso en exceso de sentencias de lógica proposicional para construir y demostrar los programas, lo que consideramos – y validamos con expertos de la materia- puede ser contraproducente para estudiantes iniciales.
Las técnicas que presenta son:
1. Tomar términos de una conjunción: Es análoga a la que nosotros llamamos Eliminación de una condición.
2. Reemplazo de constantes por variables: la más tradicional de las técnicas.
3. Fortalecimiento de invariantes: Análoga a la que denominamos de la misma manera.
4. Problemas con los bordes: Similar a la que nosotros llamamos Aging. Nosotros la vemos como un ajuste de índices para complementar reemplazo de constantes por variables cuando es necesario. No la consideramos una técnica por sí misma.

Este material comenzó a utilizarse en el año 2001 en la Universidad de Río Cuarto, y en 2003 en la Universidad Nacional de Rosario

Una propuesta de integración de nociones lógico-matemáticas en la enseñanza de la Programación [1]
Ariel Ferreira Szpiniak, Carlos Luna y Ricardo Medel
Universidad Nacional de Río Cuarto

Este artículo presenta una propuesta de integración de contenidos de la matemática y la lógica en la asignatura Programación Avanzada de la Universidad de Río Cuarto. Dicha asignatura se dicta al comienzo del segundo año de las carreras de Analista en Computación, Profesorado en Ciencias de la Computación y Licenciatura en Ciencias de la Computación, luego de haber cursado en el primer año un curso de Programación imperativa, un curso de Lógica y uno de Álgebra elemental.
Lo que nos interesó particularmente de esta propuesta es el racional de la misma, que busca *"inculcar la noción de que los programas son objetos matemáticos plausibles de ser tratados como herramientas lógico-matemáticas"*, lo que tiene una conexión muy marcada con el enfoque de nuestro trabajo.
En cuanto al estudio de Invariantes, este nuevo plan de estudios de Programación Avanzada incluía la introducción a predicados invariantes, la construcción de invariantes, y la construcción de ciclos a partir de invariantes.

Invariant based programming: basic approach and teaching experiences [9]
Ralph-Johan Back

Esta publicación es una de las primeras en reportar experiencias en la enseñanza sistemática de un método basado en invariantes. En él se describe la técnica y hay una interesante comparación entre los principales aspectos que diferencian la programación basada en invariantes de la programación tradicional. Back argumenta que siguiendo este acercamiento "la verificación de programas se vuelve un complemento de la codificación en lugar de un complemento del *testing*". Este enfoque que utiliza nos fue muy útil como guía e inspiración para apuntar nuestra investigación y la forma de resolver los distintos algoritmos en esa dirección.

Adicionalmente Back introduce los "diagramas anidados" para expresar el algoritmo, sus pre y postcondiciones, sus invariantes, variantes y el progreso del *loop*. Los objetivos que busca con los diagramas son similares a lo que nosotros buscamos con la notación que utilizamos, obviamente con la diferencia de que él lo expresa de forma gráfica.

Le encontramos un especial valor a la documentación de las experiencias obtenidas en el dictado de cursos basados en estos conocimientos que desarrollaremos en la siguiente sección.

9.2.Experiencias en la Educación

Nos hemos contactado con diversos docentes y autores de los textos que utilizamos para saber acerca de los resultados obtenidos luego de aplicar estos conceptos en la educación y sus conclusiones. Estas experiencias son fundamentales para entender qué se ha encontrado como valioso por parte de los estudiantes y con qué dificultades se han topado.

Daniel *Zingaro* nos transmitió las experiencias surgidas de utilizar su libro en cursos universitarios: en el 2008 fue utilizado en el segundo curso de Ciencias de la Computación de la Universidad de Toronto. En el curso se programaba en *Python* (lenguaje que no admite el uso de invariantes en su sintaxis), por lo que *Zingaro* introdujo una manera de razonar basada en invariantes. Él expresa que no sintió que sus alumnos se vean beneficiados por sus presentaciones sobre invariantes, ya que debido a que el nivel de los problemas a resolver no era muy complejo, los estudiantes los resolvían sin el uso de invariantes y sin usar los métodos de razonamiento que el docente intentaba transmitir. *Zingaro* reconoce que debió poner más énfasis en motivar a los alumnos a aplicar estas herramientas de razonamiento, explicando y convenciendo a los mismos de la importancia del uso de estos mecanismos.

Luego volvió a utilizar este libro en el año 2014, en un curso de segundo año de Programación. Recuerda esta experiencia de forma positiva desde el punto de vista del impacto que causó en los estudiantes, que destacaban que la naturaleza informal de este libro era un contraste muy interesante y un gran recurso de aprendizaje comparado con el enfoque formal que se presentaba habitualmente en clase.

Por su parte Carlos Luna nos contó que en sus experiencias como docente aplicando técnicas de construcción de programas demostrados mediante lógica proposicional, notó como gran ventaja

que los estudiantes lograban analizar los problemas de manera rigurosa como se deseaba, pero a la vez no lograban comprender las demostraciones con lenguajes formales como la lógica y matemática proposicional, llegando muchas veces a demostrar la corrección de programas incorrectos. Por ello es que nuestro trabajo explica cómo construir y demostrar programas mediante derivación y aplicación de invariantes, pero explicados con lenguaje natural de manera que los estudiantes puedan comprenderlo y poder explicar por qué estos programas son correctos, o en caso contrario poder demostrar por qué no lo son.

En cuanto a los resultados de esta propuesta, Luna nos contó que observaba cómo para aquellos alumnos con más habilidades lógico-matemáticas el curso presentado de esta manera se hacía más fácil y llegaban a aprobar con nota máxima, mientras que a los estudiantes con menores habilidades les costaba más comprender la asignatura que con la manera tradicional. Sin embargo, el objetivo de esta propuesta no era que los estudiantes aprueben con una mayor nota, sino que apuntaba a objetivos a largo plazo, es decir, formar estudiantes capaces de analizar y construir programas rigurosa y metódicamente, en lugar de hacerlo a prueba y error, para luego obtener mejores resultados hasta el final de la carrera y luego en su ejercicio como profesionales de la programación.

Tuvimos la oportunidad de entrevistar a Javier Blanco, co-autor del libro “Cálculo de Programas”. En dicha entrevista nos comentó sobre los resultados que observó al enseñar programación mediante técnicas formales de construcción de programas en la Universidad de Córdoba. Entre otras cosas Blanco expresa la importancia de enseñar estas técnicas en los primeros cursos de programación, ya que según su experiencia, si el alumno aprende programación sin técnicas formales, se le dificultará mucho aprenderlas y aplicarlas más adelante. Es por ello que en la Universidad de Córdoba decidió enseñar estas técnicas con este libro en el primer curso de programación de la carrera, el cual se dicta en el segundo semestre, luego de un curso de Lógica.

La principal desventaja que destaca Blanco es que algunos estudiantes se enfocan tanto en comprender los formalismos para construir programas que terminan por no comprender el propio programa que construyen.

Sin embargo, también observa una gran demanda por egresados de la Universidad de Córdoba en la industria del software, y luego en sus respectivos empleos destacan las cualidades de las técnicas que manejan. En cuanto a la parte técnica, él observa que a largo plazo los alumnos no necesariamente aplican las técnicas aprendidas a rajatabla, sino que adquieren y aplican conceptos más profundos tales como la construcción de programas más simples y concisos, el no introducir variables si no cumplen un rol en las invariantes, entre otros.

Back, en su publicación cuenta acerca de distintas experiencias de la programación basada en invariantes en la educación en las que participó. Las experiencias consistieron en sesiones de dos programadores trabajando en conjunto en la solución de problemas utilizando programación basada en invariantes y en dos cursos dictados, el primero en el 2005 para alumnos de doctorado cuya área de investigación estaba relacionada con métodos formales y el segundo en el año 2007 como un curso ordinario de ciencias de la computación.

Fueron 15 las sesiones de programación dictadas, todas a grupos de dos estudiantes. Alrededor de la mitad de los mismos tenían conocimientos sobre métodos formales pero no sobre programación basada en invariantes, mientras que los restantes no tenían ningún tipo de conocimientos del rubro. Prácticamente todos tenían sólidos conocimientos de programación. Se utilizaron grupos de dos programadores para que hubiese un diálogo entre ellos que pudiese ser observable. La tarea de los programadores consistía en la especificación del problema en términos de pre y pos condiciones, la construcción de la solución del programa y la verificación de la propia solución. El nivel de los problemas no era demasiado alto. En general se utilizaron problemas del nivel del problema de la bandera holandesa de Dijkstra (*Flag sorting*) en los programadores sin conocimientos de métodos formales y problemas más complejos para aquellos familiarizados con el rubro.

Todas las sesiones fueron exitosas. Todos los grupos lograron aprender el método y cumplir con el cometido en un lapso de entre 2 horas y media y 3 horas. Según Back la dificultad no estaba en la formulación de la invariante, sino en la especificación del problema en términos de pre y pos condiciones. Formular la naturaleza del dominio del problema en éstos términos no es sencillo, ya que cada dominio tiene sus particularidades, tanto en su naturaleza como en la notación utilizada. En promedio la mitad del tiempo disponible para solucionar el problema fue destinado a la formulación de la postcondición.

En cuanto a los cursos, aquel dirigido a estudiantes de doctorado no demostró dificultades. Los alumnos aprendieron los conceptos sin problemas. El curso brindado para estudiantes de grado de ciencias de la computación fue ofrecido como optativo, y contó con la presencia de 16 estudiantes, aproximadamente la mitad de primer año, y la mayoría contaban con algún conocimiento de lógica previo. El curso consistió en 32 horas de clases, 12 de las cuales fueron destinadas a la solución de ejercicios.

Los resultados del curso fueron catalogados como "*alentadores*" [9]. La mayor parte de los estudiantes comprendieron el enfoque de manera apropiada y fueron capaces de construir y verificar programas por su cuenta. En cuanto a la devolución por parte de los alumnos en los cuestionarios entregados, revelan una actitud positiva en cuanto al curso y el material utilizado. En promedio el curso fue "*útil, interesante, e incluso divertido*" [9]. Los estudiantes consideraron que la programación basada en invariantes "*era fácil de aprender (...) y que el método se podría utilizar en la práctica*" [9]. La mayor dificultad estuvo en la demostración de las condiciones de verificación, formulando invariantes para programas complejos y la falta de un estándar para la sintaxis y la demostración. Uno de los aspectos mejor valorados por los estudiantes es, según el documento, la capacidad de pensar en términos generales acerca de cómo funciona un programa independientemente de un lenguaje de programación en específico.

Como lecciones aprendidas Back destaca cuatro aspectos que resultan desafiantes, no sin antes desbancar el mito de que es difícil encontrar las invariantes para un programa. Adherimos a su opinión de que la dificultad está en la correcta y completa especificación del problema. El primer aspecto que considera delicado está a la hora de construir la teoría del dominio para la aplicación. Al no ser estándar, cada dominio tiene su propia notación. Otro aspecto es que se necesitan sólidos conocimientos de lógica, principalmente para la formulación de las restricciones y para poder razonar sobre un problema. Lógica de primer orden suele ser suficiente. El tercer aspecto

es la falta de herramientas. No existen demasiadas herramientas para apoyar y la introducción temprana de las mismas resulta contraproducente. Por último él considera importante la ilustración de buenas figuras para las transiciones. Esto es específico para su publicación ya que trata estos conceptos y consideramos que no aplica para nuestro trabajo.

En un intercambio privado Back expresa la importancia de los métodos formales, haciendo una distinción entre las Ciencias de la Computación y la Ingeniería en Computación, entendiendo a la primera como una disciplina científica que necesita una sólida base matemática. Back ha vuelto a afirmar que el principal problema está en que los estudiantes no tienen un lenguaje suficientemente poderoso que permita expresar las invariantes.

10. Conclusiones

En las primeras semanas del mes de abril de 2015, planteamos como principal objetivo generar un material que pueda actuar como base para la creación de un curso que apunte a la derivación de soluciones basadas en invariantes para un cúmulo de problemas habituales en la programación. Este material debe ser leído e interpretado como una guía para un eventual curso universitario para alumnos de los primeros semestres de carreras afines a la Programación. A su vez buscamos que este trabajo colabore a que los estudiantes puedan producir soluciones más simples y *performante*s cuando se enfrenten a un problema y que sean capaces de demostrar la corrección de su código y explicarlo a terceros.

Si bien este material aún no se ha utilizado en la docencia aun, debido a su reciente elaboración, tenemos como siguiente objetivo a corto o mediano plazo crear un curso universitario a partir de este trabajo, o incorporarlo a un curso existente. Creemos, basándonos en nuestra experiencia (cinco años de estudios universitarios y tres trabajando en la industria de la tecnología), y en las opiniones de otros profesionales, que en ese momento vamos a estar en condiciones de afirmar que hemos cumplido con todos nuestros objetivos.

Además, a lo largo de este año de trabajo hemos cumplido con el objetivo de tener una noción sobre la bibliografía existente acerca de la Programación basada en invariantes, así como también de la utilización de estos materiales para el dictado de cursos y los resultados obtenidos en los mismos. Esto nos ha sido de gran ayuda e inspiración para la generación de nuestro trabajo, en el que procuramos imitar aquello que ha ayudado a los estudiantes que experimentaron con estos materiales y eliminar lo que se observó cómo desventajas. Sobre esto hemos profundizado en la sección Trabajos relacionados y experiencias en la educación.

Nuestra principal contribución es en el planteo de una metodología de resolución de problemas planteada en la primera sección y utilizada a lo largo de todo el documento para resolver los problemas. Consideramos que la aplicación de esta metodología aportará a nuestro objetivo de que los desarrolladores eviten lo que denominamos “programación artesanal”.

Para ayudar al programador en su tarea de diseñar la solución a los problemas, presentamos dos métodos: ***Updating*** y **Reducción**. El primero de ellos consiste en iterar sobre algún tipo de estructura de datos o conjunto de valores, llevando un acumulador o resultado parcial que se va actualizando en cada paso del *loop* hasta que se agota el intervalo, momento en el que generalmente el acumulador se convierte en el resultado final. El segundo, Reducción, tal como dice el nombre, busca reducir el cálculo final requerido por la postcondición al cálculo en una porción menor de la estructura de datos o de los posibles valores.

Luego, para facilitar la tarea de hallar invariantes para el problema, presentamos una recopilación de técnicas para su derivación, así como algunos ajustes que son necesarios en ciertas situaciones.

Las técnicas ya estaban identificadas en muchos de los textos que citamos a lo largo del trabajo [7] [4] [8]. Hemos tomado lo que creímos más útil de cada una de ellas. El planteo que proponemos de elegir el método y luego derivar invariantes mediante las técnicas tiene la ventaja de que los métodos son más intuitivos, mientras que las técnicas son más sintácticas.

La primera de estas técnicas y la más usada es **Relajación de constante**, técnica que consiste en reemplazar una constante N que no será alterada durante el transcurso del algoritmo por una variable *i* que irá cambiando de valor en cada paso del *loop*, hasta alcanzar i=N como condición de terminación, garantizando que hemos recorrido toda la estructura o los posibles valores.
Luego planteamos la técnica **Eliminación de condición**, que se basa en tomar algunas de las postcondiciones del problema y utilizarlas como invariantes, y el resto de las postcondiciones como condiciones de terminación del *loop*, para así al salir del *loop* saber que hemos cumplido con el objetivo del problema.
Más adelante introducimos la técnica ***Decoupling*** que consta en reemplazar una variable (normalmente el retorno) por dos variables (retorno y *x*), utilizando la igualdad entre ellas como condición de terminación del *loop* o como parte de la misma.
Por último presentamos la técnica **Fortalecimiento de invariantes**. Esta técnica consiste en plantear la forma del ciclo del *loop* dejando algunas expresiones sin resolver, lo cual puede pensarse como ecuaciones con algunas variables como incógnitas. Luego calculamos para "despejar" las incógnitas.
En algunos casos estas técnicas son aplicadas junto con ajustes, a los que llamamos ***Aging.*** Para poder realizar esto de forma coherente con los objetivos planteados hemos desarrollado una notación propia que se adecúa al hilo de pensamiento que queremos que ocurra en la mente del desarrollador. Esta notación toma ideas del lenguaje de programación *Dafny* [6] y de la notación utilizada en los textos de métodos formales que hemos utilizado como referencia.

Nos parece oportuno señalar que la clasificación propuesta de métodos y técnicas de derivación aplica para una amplia variedad de casos, pero con ella no estamos estableciendo que sirva para resolver todos los problemas del universo de la Programación. Esta sería una afirmación muy tajante. Sin embargo, lo que buscamos transmitir es que adoptando la metodología que proponemos, la tarea de enfrentarse a cualquier problema se hará más sencilla y se generará una mejor solución.

Luego de haber aprendido sobre programación basada en invariantes y métodos formales de derivación de algoritmos hemos notado que la certeza de la corrección y la capacidad de explicación del programa obtenida por parte del programador, así como la calidad del código producido es superior. Éste código suele ser más corto, comprensible y *performante* que el producido en base a la prueba y error. Vemos como principal valor agregado la forma de pensar obtenida de la aplicación de estas técnicas. Conceptos como la correcta especificación de un problema y tener un conocimiento de lo que sabemos que se cumple en un momento determinado son los que luego permiten un salto de calidad y una diferenciación en la solución de un problema. El estudio de las invariantes es una buena forma de adquirir estos hábitos que son extrapolables a muchos ámbitos de las ciencias de la computación.

También hemos de reconocer que ninguno de los problemas presentados representa una dificultad tal que no puedan ser resueltos sin la aplicación de invariantes. Creemos que cualquier

programador los hubiese podido resolver eventualmente, aunque quizás sin dar con la mejor solución. En ese sentido adherimos a la opinión de Daniel Zingaro, que en un intercambio privado vía correo electrónico nos trasmitió que en su experiencia "*la mayoría de los problemas (...) pueden ser 'hackeados' sin invariantes, por lo que los alumnos no ven una razón para usarlas*". Esto resulta problemático para el docente que intenta transmitir estos conocimientos, ya que tiene una doble tarea a la hora de motivar a sus alumnos en el uso de las invariantes.

Creemos que el momento en el cual estos conocimientos son transmitidos es particularmente delicado. Si se ubica como un curso demasiado avanzado en la carrera el estudiante contará con suficientes recursos para solucionar los problemas y no se verá motivado por aprender nuevas técnicas. En cambio si se ubica demasiado temprano el estudiante tendrá problemas para comprender los conceptos. Creemos que el mejor momento es en el primer curso de diseño de algoritmos, que no necesariamente coincide con el primer curso de programación, ya que este suele estar dedicado a introducir conceptos elementales como lo son las variables, clases, métodos, asignaciones e instrucciones.

Afortunadamente estos conceptos fueron reafirmados por autores como Carlos Luna, Javier Blanco y Daniel Zingaro en testimonios recogidos a lo largo del año.

Nuestra experiencia nos ha demostrado que no existen demasiadas herramientas para brindar apoyo a la programación basada en invariantes. Para extender los conceptos aquí presentados y su aplicabilidad a la docencia es imperioso el desarrollo de herramientas que la respalden. Consideramos que las existentes presentan un nivel de dificultad y una curva de aprendizaje que no son óptimas para el público aquí apuntado. En concreto consideramos que el desarrollo de un editor de código o IDE que se adapte a la notación presentada es un primer paso. Un segundo paso es el desarrollo de un intérprete que logre traducir la sintaxis y pueda ser compilada a lenguajes como *Dafny*. Creemos que estos objetivos son tanto útiles, como alcanzables para profundizar en la aplicación de estos conocimientos en la educación.

11. Referencias bibliográficas

[1] Szpinak, A. Ferreira, C. D. Luna y R. H. Medel, «Una propuesta de integración de nociones lógico-matemáticas en la enseñanza de la programación,» Córdoba, Argentina, 1997.

[2] Dijkstra, A Discipline of Programming, Prentice Hall , 1976.

[3] Dijkstra, W. H. J. Feijen y J. Sterringa, A Method of Programming, Addison-Wesley, 1988.

[4] A. Kaldewaij, Programming: The Derivation Of Algorithms, Prentice Hall, 1990.

[5] D. Zingaro, Invariants: a Generative Approach to Programming, College Publications, 2008.

[6] «Dafny: a language and program verifier for functional correctness,» Microsoft. [En línea].

[7] A. Furia, M. B. y Velder, «Loop invariants: analysis, classification, and examples,» 2012.

[8] J. Blanco, S. Smith y D. Barsotti, «Cálculo de Programas,» Córdoba, Argentina, 2008.

[9] R.-J. Back, «Invariant based programming: basic approach and teaching experiences,» Turku, Finland, 2008.

[10] R. Wagner y M. Fischer, The string-to-string correction problem, 1974.

[11] R. C. Backhouse, Algorithmic Problem Solving, 2011.

12. Anexos

12.1. Técnicas de derivación

Relajación de constantes
Es la técnica más popular, sirve para una gran variedad de problemas, y probablemente sea la más sencilla de comprender y aplicar. Consiste en reemplazar una constante *N* que no será alterada durante el algoritmo por una variable *i* que irá cambiando de valor en cada paso del *loop* hasta alcanzar la condición de salida [7]. El valor de la variable tendrá una de sus cotas en la constante que reemplaza.

Si la postcondición nos dice que la condición S se cumple para Q, de la forma:

$$S \rightarrow Q$$

Aplicando esta técnica, derivamos como invariante:

$$S \rightarrow Qi$$

Esto significa que la condición se cumple para la sección que culmina en la variable i.

Una vez que hemos finalizado la iteración de la variable i, S será válida para toda la estructura, lo que es equivalente a:

$$S \rightarrow Q$$

Para ilustrar esto tomemos como ejemplo la sumatoria de enteros.

La postcondición del problema es:

$$x = \sum_{i=0}^{N} i$$

Aplicando la técnica obtenemos como invariante:

$$x = \sum_{j=0}^{i} j$$

La condición de terminación será $i == N$. Por ello, al terminar la ejecución podemos realizar *esta* sustitución:

$$x = \sum_{j=0}^{N} j$$

Por lo tanto hemos arribado a la postcondición [4].

Eliminación de una condición

Esta técnica es aplicable cuando el problema tiene más de una postcondición. Consiste en tomar una de las postcondiciones como condición de terminación del *loop* y las otras como invariantes del problema.

Supongamos que tenemos una postcondición que nos dice que al terminar la ejecución se debe cumplir:

$$S \wedge Q \wedge R$$

Luego elegimos una de ellas como condición de terminación del *loop*, y las otras dos como invariantes. Supongamos:

Condición de Terminación: S
Invariante 1: Q
Invariante 2: R

Si logramos mantener las invariantes a lo largo del *loop*, y de probar la terminación del mismo, entonces habremos llegado a la corrección del algoritmo, ya que el *loop* termina cuando su condición de terminación se satisface, y dado que la condición de terminación es una de las postcondiciones y las invariantes son las otras, estaremos cumpliendo con el total de la postcondición a la salida del *loop*.

Desacoplamiento

Esta técnica consiste en reemplazar una propiedad de una variable utilizada en la postcondición por una propiedad de dos variables que será utilizada en la invariante.
Normalmente se suele reemplazar una variable A, generalmente el retorno o resultado del método por dos variables: esta misma, y una nueva variable *x,* usando la igualdad entre estas como parte de la condición de salida del *loop,* o como la condición de salida misma.

Al igual que en relajación de constante, cuando aplicamos la técnica de desacoplamiento, obtenemos al menos una invariante esencial y una limitante.

Esta técnica no es de aplicación inmediata como las anteriores, ya que suele requerir un conocimiento avanzado sobre el dominio del problema para poder aplicarla, como en el caso del máximo común divisor. El conocimiento del algoritmo de Euclides fue el que permitió la aplicación de esta técnica.

Fortalecimiento de Invariantes

Esta técnica consiste en plantear la forma del ciclo del *loop* dejando algunas expresiones sin resolver, lo cual puede pensarse como ecuaciones con algunas variables como incógnitas. Luego calculamos para "despejar" las incógnitas. Este proceso puede dar lugar a nuevas expresiones que

no admiten una expresión simple en términos de las variables de programa. Una forma de resolver esta situación es introducir nuevas variables que se mantengan invariantemente igual a algunas de estas sub expresiones. Esto resuelve el problema al tiempo que aparece la necesidad de mantener las nuevas invariantes [8].

El ejemplo más representativo del uso de esta técnica lo vimos en el cálculo del cubo utilizando únicamente el operador de la suma. La postcondición nos dice:

$$x = N^3$$

Esto lo derivamos y llegamos a la expresión $E = n^3 + 3n^2 + 3n + 1$. Como no podemos calcular esta expresión ya que solamente podemos utilizar el operador suma, realizamos un cambio de variable $y = n^3 + 3n^2 + 3n + 1$. Aquí es donde introducimos la variable para resolver la expresión que no hubiésemos podido de otra forma. En ejemplo completo puede encontrarse en la sección Algoritmos Avanzados.
Ésta técnica, al igual que *Decoupling*, no puede ser utilizada en todos los escenarios, sino que el dominio del problema debe permitirlo. Adicionalmente suele utilizarse en conjunto con otras técnicas. En el caso del cubo utilizando sumas lo hicimos mediante relajación de constantes en conjunto con el fortalecimiento.

Aging
Reemplaza una variable o expresión por una expresión que representa el valor que la variable tuvo en iteraciones previas del *loop*, habitualmente en la última iteración.
Estas invariantes sirven para resolver desfasaje de una unidad entre cuando la variable es evaluada en la invariante y cuando es actualizada en el paso del *loop*. Al entrar en la iteración sabemos que la invariante se cumple para el conjunto que incluye el elemento anterior, pero no el que representa el índice sobre el que estamos parados. Usualmente nos aseguramos que se cumpla para el elemento actual y recién después actualizamos el valor de la variable. Ésta técnica es menos sintáctica que las otras, porque su aplicación no nos deriva una invariante directamente, sino que ajusta una ya existente. Por esto la consideramos un ajuste y no una técnica propiamente dicha.

Ejemplos: *Quick-sort, selection sort, insertion sort.*

Selection sort: en el *loop* interno primero incrementamos *i* y luego entramos al *loop* interno, y lo que sabemos es que el *array* está ordenado hasta i-1. Si no hiciéramos *aging* en ese caso estaríamos diciendo que el *array* ya está ordenado hasta una posición que no sabemos.

12.2. Problemas resueltos

Algoritmo	Método de Solución de Problema	Técnicas utilizadas para la derivación	Capítulo
Pertenencia a un array	Updating	Relajación de Constantes/Aging	2.1

Menor divisor de un entero	Reducción	Eliminación de Condición	2.2
Suma de array	Updating	Relajación de Constantes/Aging	3.1
Producto de array	Updating	Relajación de Constantes/Aging	3.2
Multiplicación usando suma	Updating	Relajación de Constantes	3.3
Potencia	Updating	Relajación de Constantes	3.4
División entera	Reducción	Eliminación de Condición	3.5
Raiz cuadrada	Reducción	Eliminación de Condición	3.6
Logaritmo	Reducción	Eliminación de Condición	3.7
Máximo común divisor	Reducción	Decoupling	3.8
Búsqueda lineal	Reducción	Eliminación de Condición	4.1
Búsqueda binaria	Reducción	Relajación de Constantes	4.2
Insertion Sort	Updating	Relajación de Constantes	5.1
Inserción ordenada de elemento (Insertion Sort)	Reducción	Relajación de Constantes	5.1
Selection Sort	Updating	Relajación de Constantes	5.2
Elección del pivot (Quick Sort)	Updating	Relajación de Constantes	5.3
Flag Sorting (dos valores)	Updating	Relajación de Constantes/Aging	5.4.1
Problema de la bandera holandesa (Flag Sorting con tres valores)	Updating	Relajación de Constantes/Aging	5.4.2
La meseta más larga	Updating	Relajación de Constantes/Aging	6.1
Mayor segmento de suma	Updating	Relajación de Constantes/Aging	6.2
Rotación de array	Updating	Relajación de Constantes/Aging	6.3
Reverso de array	Updating	Relajación de Constantes/Aging	6.4
Búsqueda del elemento faltante (¿Cuál falta?)	Reducción	Relajación de Constantes	6.5
Map	Updating	Relajación de Constantes/Aging	6.6.1
Filter	Updating	Relajación de Constantes/Aging	6.6.2
Cálculo del cubo con sumas	Updating	Relajación de Constantes/Fortalecimiento de Invariantes	6.7
Secuencia de fibonacci	Updating	Relajación de Constantes/Fortalecimiento de Invariantes	6.8

Criba de eratóstenes	Updating	Relajación de Constantes/Aging	8.9
Segmento de Ceros	Updating	Relajación de Constantes/Aging	7.1
La meseta más larga (Programación Dinámica)	Updating	Relajación de Constantes/Aging	7.2
Distancia de Levenshtein	Updating	Relajación de Constantes/Aging	7.3
El camino a la riqueza	Updating	Relajación de Constantes/Aging	7.4
Cortes en la barra de chocolate	Juegos	Ninguna	8.1
Partidos en torneo de tenis	Juegos	Ninguna	8.2
Problema de las cajas vacías	Juegos	Ninguna	8.3
Palitos Ganadores	n/a	Ninguna	8.4
Bidones de agua	n/a	Ninguna	8.5

Tabla 12-0-1:Problemas resueltos con sus respectivos métodos y técnicas aplicadas.