Universidad ORT Uruguay
Facultad de Ingeniería

Metodología de Programación con Dafny y KeY

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

Matias Hernández - 169236
Gianfranco Drago - 198490

Tutor: Álvaro Tasistro

2022

Declaración de autoría

Nosotros, Matías Hernández y Gianfranco Drago, 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 de la carrera Ingeniería en Sistemas;
- 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.

Matías Hernández

Gianfranco Drago

28-09-2022

Agradecimientos

Agradecemos al Dr. Álvaro Tasistro, quien aceptó ser tutor de nuestra tesis. Nos ayudó a encontrar marco teórico para utilizar como base en la elaboración de la metodología. También nos brindó opiniones, revisiones y conocimiento que fueron fundamentales durante todo el proceso. En particular, aportó considerablemente en la verificación formal de algoritmos complejos que se tenían que probar.

Abstract

En esta investigación se presenta el diseño de una metodología para la especificación y derivación de algoritmos iterativos expresados en código imperativo, y se utilizan las herramientas mecánicas *KeY* y *Dafny* para sus respectivas implementaciones. Creemos que el enfoque metodológico que se introduce, ayuda didácticamente a lectores que están recién comenzando en el área de la verificación formal, a tener un pensamiento orientado a invariantes al momento de realizar algoritmos. A lo largo del cuerpo de la tesis, se muestran diversos algoritmos utilizando la metodología diseñada y su correspondiente implementación en las dos herramientas mencionadas anteriormente. La evaluación concluye que la metodología diseñada puede ser implementada en la materia "Estructuras de Datos y Algoritmos 2"de la carrera de Ingeniería en Sistemas de Universidad ORT Uruguay.

Palabras clave

especificación formal; verificación automática; invariantes; key; dafny

Índice general

1 Introducción

1.1 Objetivos

Los objetivos del presente trabajo de iniciación en la investigación han sido:

1. Establecer una metodología para la especificación y derivación de algoritmos iterativos expresados en código imperativo.

   Con base en el estudio de metodologías existentes y la utilización de técnicas de especificación formal que involucran la abstracción, establecer una metodología de derivación sistemática de código a partir de dichas especificaciones. Realizar los algoritmos contenidos en la tesis utilizando dicha metodología y concluir sobre la conveniencia en la utilización de la misma.

2. Estudiar el valor didáctico de utilizar herramientas mecánicas para la verificación de la derivación totalmente formal de algoritmos.

   Mediante el uso de herramientas de verificación formal conocidas por la industria, como *KeY* y *Dafny*, determinar el valor didáctico que tiene la realización de los ejemplos de la tesis utilizando ambas herramientas. Además, realizar una comparación entre ambas implementaciones.

3. Concluir de los dos análisis anteriores (aspecto informal y formal), la aplicabilidad de alguno de ellos, en alguno de los cursos en la carrera de Ingeniería en Sistemas de Universidad ORT Uruguay.

1.2 Estado del arte

En esta sección se comentan documentos que contemplan el contexto actual sobre metodologías de verificación formal, así como también herramientas mecánicas para la verificación de la derivación totalmente formal de algoritmos.

1.2.1 Metodologías

Invariants: A Generative Approach to Programming [1]

El autor del libro, Daniel Zingaro, introduce al lector en el pensamiento orientado a invariantes al momento de especificar formalmente algoritmos. Se utilizan algoritmos desarrollados en *Java* para demostrar su metodología y además cuenta con varios ejercicios propuestos para que el lector pueda practicar.

Loop invariants: Analysis, classification and examples [2]

Los autores clasifican algoritmos según criterios que exponen en el libro y muestran algunas técnicas para la especificación formal de los mismos. Se realizan implementaciones en lenguaje *Eiffel*.

(In-)Formal Methods: The Lost Art [3]

El autor del libro, Caroll Morgan, detalla que el texto está pensando para aprender a programar más rápido, efectivamente, y con mejores resultados. Se especifican algunas técnicas de codificación, que se pueden utilizar para probar el funcionamiento de algoritmos.

Curso Lógica de la Programación de Universidad ORT Uruguay

El profesor, Dr. Álvaro Tasistro, realiza un curso que corresponde a una electiva sobre lógica de la programación. El curso contiene una introducción a metodologías informales para la derivación de invariantes, a partir de la especificación formal. Además, se enseña el lenguaje *Dafny* y se realizan diversas implementaciones de algoritmos conocidos.

1.2.2 Herramientas

Coq [4]

Coq es un sistema de código abierto desarrollado por Inria, un instituto de ciencia de la computación en Francia, que se utiliza para realizar pruebas sobre programas. Provee un lenguaje formal para escribir definiciones matemáticas, así como también ejecutar algoritmos y teoremas.

Why3 [5]

Why3 es una plataforma para la verificación deductiva de programas. Proporciona un lenguaje para especificaciones y programación denominado *WhyML*, que se basa en verificadores de teoremas externos a la herramienta. Por defecto, viene con una librería de teoremas lógicos y estructuras de datos básicas de programación.

Isabelle/HOL [6]

Isabelle es un asistente genérico de pruebas que permite que las fórmulas matemáticas se expresen en un lenguaje formal y además proporciona herramientas para probar dichas fórmulas en un cálculo lógico. Originalmente se desarrolló en la Universidad de Cambridge y la Universidad de Múnich, pero ahora existen numerosas instituciones que contribuyen al proyecto.

Frama-C [7]

Frama-C es una plataforma que permite verificar requerimientos y garantizar especificaciones de programas en lenguaje *C*. La plataforma puede realizar análisis estático y dinámico gracias al uso de extensiones colaborativas.

SLAM [8]

SLAM es un proyecto que verifica que los programas satisfagan ciertas propiedades de comportamiento. La finalidad es ayudar a asegurar la confianza y correcto funcionamiento de dichos programas.

The Pi Verifying Compiler (πVC) [9]

Es un proyecto desarrollado en Stanford, que está compuesto por un compilador que verifica programas especificados para asegurar su correcto funcionamiento. Se utiliza un lenguaje de programación propietario denominado *Pi* (*Prove it*).

KeY [10]

KeY es un proyecto de largo alcance que comenzó en 1998 por Reiner Hähnle, Wolfram Menzel y Peter Schmitt en la Universidad de Karlsruhe. Es un programa que permite la especificación formal en lenguaje *JML* utilizado a través de comentarios sobre código *Java*.

Esta es una de las herramientas elegidas para las implementaciones de los algoritmos que se realizan en la tesis, ya que utiliza un lenguaje de programación ampliamente conocido por la industria (*Java*) y además utiliza un estándar de especificación como lo es *JML*.

Dafny [11]

Dafny es un lenguaje de programación que se utiliza para la verificación formal de algoritmos. Fue creado por el grupo *Research in Software Engineering (RiSE)* de *Microsoft Research* y diseñado por K. Rustan M. Leino.

Dafny es la otra herramienta elegida para las implementaciones de los algoritmos. La principal razón es porque su verificador puede realizar verificaciones en tiempo real, a medida que se van escribiendo las especificaciones. Otra razón es que parte del equipo ya tenía conocimiento previo [12] en la herramienta.

1.3 Estructura de la tesis

La estructura del resto de la tesis es la siguiente:

Capítulo 2

Se presenta el diseño de una metodología de derivación formal de algoritmos iterativos expresados en código imperativo, que se va a utilizar en cada uno de los algoritmos contenidos en la tesis.

Capítulo 3 y 4

Se presenta una introducción a *Dafny* y *KeY* como herramientas de verificación de algoritmos mediante la especificación formal. Se detallan las cláusulas que se utilizan con cada una de las herramientas en el cuerpo de la tesis y detalles fundamentales del procesamiento de la especificación.

Capítulo 5, 6 y 7

Se presentan algoritmos aritméticos, de búsqueda y de ordenamiento utilizando la metodología de derivación formal presentada anteriormente e implementadas en ambas herramientas.

Capítulo 8

Contiene las respectivas conclusiones a los objetivos de la tesis.

Repositorio de códigos

Los códigos utilizados están disponibles públicamente en el siguiente enlace:

`https://github.com/matiashrnndz/programming-logic-with-key-n-dafny`

2 Metodología

La verificación completa de algoritmos expresados en código imperativo generalmente requiere, como paso crítico, equipar cada iteración con alguna invariante. Además de su rol en la verificación, las invariantes ayudan en la comprensión del programa al proporcionar conocimientos sobre la naturaleza de los algoritmos. El reto está en encontrar invariantes que representen correctamente dicha naturaleza y sean útiles en la demostración. [2]

Una apreciación con respecto a las invariantes, es que desde que se aprende a hacer una iteración en programación, uno está acostumbrado a pensar en invariantes, sin ni siquiera tener conocimiento formal de ellas. Por ejemplo, al estar haciendo una búsqueda lineal, el desarrollador da por entendido que las posiciones que ya fueron recorridas, no tienen al elemento buscado. Indirectamente está pensando en invariantes para el desarrollo de la iteración. Lo que se propone es que las invariantes tomen un rol más importante y explícito, y se logre entender la naturaleza de un algoritmo gracias a ellas.

2.1 Alcance de la metodología

Dado un contexto donde se debe resolver un algoritmo iterativo expresado en código imperativo y se tienen las pre y post-condiciones del mismo. La metodología, propone establecer un marco de trabajo para la derivación de invariantes desde las pre y post-condiciones que el algoritmo debe satisfacer para cumplir con sus especificaciones.

2.2 Sintaxis de la metodología

Para especificar la metodología, se decidió utilizar como base la sintaxis del lenguaje *Eiffel* [13], realizando una adaptación a nuestra metodología:

```
AlgorithmName (Parámetros): (Salida)
  require
    -- Pre-condiciones
  ensure
    -- Post-condiciones
  local
    -- Variables locales
  do
    invariant
      -- Invariantes
    init
      -- Inicialización de variables
    until
      -- Condición de terminación
    step
      -- Declaraciones
    variant
      -- Variantes
    adjust
      -- Ajustes
    end
  end
```

La metodología comienza desde un punto inicial denominado *require* que son las pre-condiciones que tiene el programa. También, se conoce el punto final denominado *ensure* que son las post-condiciones.

Generalmente, es necesaria la realización de iteraciones para lograr que se cumplen las post-condiciones, ya que la complejidad es suficientemente alta. Con normalidad se suelen utilizar estructuras de datos y operaciones sobre las mismas. La metodología está pensada principalmente para estos casos complejos.

El cuerpo de la función es el nexo entre la pre-condición y la post-condición del programa y se utiliza alguna declaración iterativa para poder recorrer una estructura de datos determinada. Este cuerpo se denomina con la cláusula *do*.

Se utilizan variables auxiliares, que pueden ser acumuladoras, índices, punteros, entre otras, que se deben inicializar para operar dentro de dicha iteración. Este proceso es denominado *init* y son condiciones iniciales que tiene el programa junto a los parámetros de entrada.

Las declaraciones iterativas, cuenta con una condición de terminación en la iteración por parte de las variables que lo componen. Estas condiciones se denominan con la cláusula *until*.

Por otra parte, dentro de cada iteración, existe una lógica inherente a la naturaleza del algoritmo y que además en cada paso, debe ayudar al verificador a estar más cerca de alcanzar la condición de terminación de la iteración y mantener las invariantes. A esta cláusula se la denomina *step*.

En este punto, se pueden detectar dos tipos de predicados que marcan el comportamiento de cada *step*. Por un lado, se tienen los predicados que van variando en cada uno de los pasos de la iteración, denominados *variant*. Estos predicados tienen la intención de marcar un decrecimiento que ayude a probar la terminación del algoritmo. Las variantes son funciones de las variables, están acotadas inferiormente y deben decrecer en cada paso.

Por otro lado, se tienen predicados que no varían en ningún paso de la iteración, a los que se le denomina *invariant*.

Como se comenta anteriormente, para ciertos casos, no alcanza con una iteración, sino que existen iteraciones anidadas o secuenciales. Además, también puede suceder que se necesite un ajuste (se denomina *adjust*) para que bajo cierta condición se ajuste el resultado final de tal forma que se cumplan las post-condiciones del mismo.

Con esta explicación, el problema de diseño de algoritmos se establece como el de elegir sabiamente las invariantes que debe tener cada una de las iteraciones y luego adaptar el resto de las cláusulas según dichas invariantes.

Existe una relación bastante estrecha entre las invariantes y las variantes en las iteraciones, ya que se podrían entender como dos sub-conjuntos complementarios, que unidos determinan el estado actual del algoritmo en una determinada iteración. Generalmente, a medida que la variante reduce su rango de variación, las invariantes van a estar más cerca de tender a ser la post-condición.

2.3 Tipos de invariantes

En la mayoría de los casos, se pueden definir dos tipos de invariantes fundamentales:

1. Invariantes de acotamiento:

   Las invariantes de acotamiento son las que acotan dentro de un rango la flexibilidad que tienen los índices o valores de una variable.

2. Invariantes esenciales:

   Son las invariantes que generalmente terminan derivando en la post-condición que quieren ayudar a probar. En muchas ocasiones cumplen la misma propiedad de la post-condición, pero de manera parcial, de tal forma que una vez terminada la iteración (y quizás un posible ajuste) se derivan en la post-condición.

2.4 Métodos para la derivación de invariantes

A continuación, se describen los métodos (técnicas) utilizados para derivar invariantes en base a las pre y post-condiciones, principalmente basados en el curso "Lógica de la Programación" de Universidad ORT Uruguay:

1. Sustituir constante por variable.

   Las invariantes se deducen reemplazando una constante (que en algunas ocasiones se recibe por parámetro) con una variable, de tal forma que se pueda verificar que se mantiene determinado resultado de forma parcial en cada paso, hasta que llega a converger en la post-condición.

   Por ejemplo, si se tiene una post-condición R a cumplirse dentro de un rango [0..N), entonces luego de la sustitución, la invariante está dada por la propiedad en un rango [0..k) con (k $\leq$ N).

   $$R([0..N)) \longmapsto R([0..k))$$

   Cuando $k = N$, la invariante converge a la post-condición R.

2. Fortalecimiento de invariante.

   En algunos casos, se tiene una invariante para una determinada posición k y se necesita tener conocimiento también del estado en la posición $k + 1$ (agrega una variable). Para esos casos, es necesario declarar una nueva invariante que "fortalezca" a la invariante anterior.

   Un ejemplo claro para esta metodología es el algoritmo Fibonacci, que utiliza el cálculo de elementos contiguos para evaluar al próximo elemento. En ese caso, se necesita una invariante esencial para la posición k, que eventualmente debe llegar a la posición n solicitada y otra invariante que fortalezca a la primera con la posición $k + 1$.

3. Invariante de cola

   Las invariantes de cola son más débiles que las invariantes comunes y representan lo que falta por computar hasta llegar al estado final. En cada cómputo, se aplica una determinada función a alguna variable. [14]

   Si el valor final a calcular se representa con $f(x)$, r es el acumulador, $f(x_i)$ es el cálculo en la posición parcial x_i y $\oplus$ la operación a realizar, entonces:

$$f(x) = r \oplus f(x_i)$$

es un invariante de cola.

4. Seleccionar condiciones (conyuntos)

   Sea una post-condición R, consistente en una conjunción de condiciones. Algunas de ellas pueden ser transformadas en invariantes y las otras en la condición de terminación.

   $$R : R_1 \wedge R_2 \wedge R_3 \wedge .. \wedge R_n$$

   En este caso, se podría elegir a las condiciones [1..n-1] para ser probadas utilizando invariantes:

   $$R_1 \wedge R_2 \wedge R_3 \wedge .. \wedge R_{n-1}$$

   y la condición n para ser la condición de terminación.

   $$R_n.$$

2.5 Procedimiento

1. Invariantes:

   Se aplican métodos de derivación de invariantes sobre la especificación (pre y post-condiciones) para poder deducir las invariantes.

2. Terminación:

   Se determina la condición de terminación del algoritmo y se define la variante.

3. Inicialización:

   Se definen las inicializaciones para poder ingresar a las invariantes desde la pre-condición.

4. Paso:

   Se define como avanzar hacia la condición de terminación, decreciendo la variante y manteniendo las invariantes.

5. Ajuste:

   Es la manera (generalmente trivial) de producir la post-condición desde la invariante, sabiendo que se ha alcanzado también la condición de terminación.

2.6 Operaciones personalizadas

A continuación se comentan algunas operaciones personalizadas que se utilizan en las especificaciones de los algoritmos, que facilitan su lectura:

1. Largo de un *array*:

   Cuando el algoritmo recibe un *array* por parámetro, siempre se utiliza N como el largo del mismo.

   $$N := arr.length$$

2. Definición de un rango:

   El rango admite ser abierto o cerrado en cada extremo.

   $$[0..N)$$

   $$(2..4)$$

3. Sub-conjuntos de elementos de un *array*:

   Se permite realizar un sub-conjunto de elementos desde una posición k hasta una posición i de un *array*, de la siguiente manera:

   $$arr[k..i]$$

   Esta notación admite también intervalos abiertos y cerrados. Si no se desea incluir a i, se podría escribir de la siguiente manera:

   $$arr[k..i)$$

4. Operador lógico que recibe dos conjuntos de elementos por parámetro:

   Utilizando cualquier operador lógico de comparación y dos conjuntos a comparar, la notación representa a la aplicación del comparador sobre cada elemento del conjunto izquierdo con el conjunto derecho.

   $$arr[0..i) \leq arr[i..N)$$

En este ejemplo, todos los elementos que tiene el *array* en las posiciones [0..i) son menores o iguales a los elementos que tiene el *array* en las posiciones [i..N).

5. Predicado *perm*:

   El predicado *perm* es verdadero cuando los dos conjuntos de elementos que recibe por parámetro, son permutaciones entre ellos.

   perm(arr[..], old(arr[..]))

   En este ejemplo, *perm* será verdadero, sólo si los elementos actuales del *array* son una permutación con los elementos iniciales del *array*.

6. Predicado *sorted_increasing*:

   Es un predicado que es verdadero cuando el conjunto de elementos que recibe por parámetro está ordenado ascendentemente.

   sorted_increasing(arr[..])

   En este ejemplo, el conjunto de elementos que se recibe por parámetro son todos los elementos del *array*.

3 Dafny

Dafny es un lenguaje de programación fuertemente tipado, que soporta tanto programación imperativa como declarativa. Fue creado por el grupo *Research in Software Engineering (RiSE)* de *Microsoft Research* y diseñado por K. Rustan M. Leino. [12]

El usuario escribe tanto la implementación del algoritmo, como su especificación formal. Desde el punto de vista de la implementación, Dafny soporta conceptos tradicionales como clases, herencias, tipos de datos inductivos, asignaciones dinámicas, entre otros. Con respecto a la especificación formal, Dafny admite la especificación de pre y post-condiciones, declaración de invariantes y métricas de terminación de los algoritmos para poder verificar la corrección total de los mismos.

La verificación de programas funciona compilando el algoritmo en Dafny a un lenguaje de verificación intermedio denominado Boogie, de forma tal que si Boogie asegura la corrección de un programa, también estaría asegurando la corrección del programa hecho en Dafny. La herramienta Boogie es utilizada para generar verificaciones de las condiciones utilizando su motor de razonamiento lógico. [15] [12]

Cuenta con una extensión de *Visual Studio Code*, que permite al verificador de Dafny estar revisando el código en tiempo real, marcando errores, mostrando contraejemplos y verificando si las especificaciones declaradas se cumplen. [11]

A continuación, se detallan las cláusulas más relevantes que se utilizan en la implementación de los algoritmos contenidos en la tesis. Como aclaración, no abarca el alcance total que tiene cada herramienta, sino que la intención es que el lector pueda entender los algoritmos sin necesitar conocimiento previo de Dafny.

3.1 Tipos de datos básicos

Los tipos básicos que se utilizan en la tesis son *bool*, *char*, *int* y *nat*.

```
var a: bool := false  // Init a bool type
var a: char := 'A'    // Init a char type
var a: int := -5      // Init an int type
var a: nat := 5       // Init a nat type
```

3.2 Estructuras de datos

Las estructuras de datos que se utilizan son *array*, *set*, *multiset* y *seq*.

Arrays

Los *arrays* son estructuras de datos de una dimensión, que son accedidos como referencia.

```
a := new T[N]                // Default values with length N
a := new int[N](i => i + 1)  // Init values using expression with length N
```

Las operaciones que se utilizan son las siguientes:

```
a.Length  // Length of array
a[i]      // Value on position i
a[i] := b // Assignment
a[lo..hi] // Subarray conversion to sequence
a[lo..]   // Drop
a[..hi]   // Take
a[..]     // Array conversion to sequence
```

Sets

Son conjuntos de expresiones anotadas entre *brackets* que pueden estar vacío y sus elementos no tienen orden ni pueden estar duplicados. [15]

```
{}               // Empty set
{2, 7, 5, 3}     // Set of type int
{4+2, 1+9, a*b}  // Set of type int asigned as an aritmetic expression
```

Además, admite las siguientes operaciones relacionales:

```
a  ==  b  // Equality
a  !=  b  // Disequality
a  <   b  // Proper subset
a  <=  b  // Subset
a  >=  b  // Superset
a  >   b  // Proper superset
```

```
a  !!  b  // Disjointness
a  +   b  // Union
a  -   b  // Difference
a  *   b  // Intersection
a  in  b  // Membership
a !in  b  // Non-membership
|a|       // Cardinality
```

Multisets

Son similares a los *set* pero tienen en cuenta la multiplicidad de cada elemento, no sólo su presencia o ausencia como sucede en los *sets*. [15]

```
multiset{}                  // Empty multiset
multiset{0, 1, 1, 2, 3, 5}  // Multiset of type int
multiset{4+2, 1+5, a*b}     // Multiset of type int asigned as an aritmetic expression
```

Las operaciones relacionales que admiten los *multisets* son las siguientes:

```
a  ==  b  // Equality
a  !=  b  // Disequality
a  <   b  // Proper multiset subset
a  <=  b  // Subset
a  >=  b  // Superset
a  >   b  // Proper multiset superset
a  !!  b  // Disjointness
a  +   b  // Union
a  -   b  // Difference
a  *   b  // Intersection
a  in  b  // Membership
a !in  b  // Non-membership
|a|       // Cardinality
a[e]      // Multiplicity of e in a
a[e := n] // Change of multiplicity
```

El tipo de datos *multiset* será utilizado principalmente para determinar las propiedades de integridad de diversos algoritmos. Se opta por utilizar esta estructura de datos principalmente porque acepta elementos repetidos. Un ejemplo sería:

```
multiset(arr[..]) == multiset(old(arr[..]))
```

En este caso, se comparan los *multiset* de los elementos del *array* al finalizar el algoritmo, con los elementos del *array* al comienzo del algoritmo (*old*).

Seqs

Son secuencias de elementos de algún tipo T que determinan un mapeo de un conjunto de números naturales denominados índices a valores del tipo T. [15] Se pueden asignar de la siguiente manera:

```
[]                 // Empty seq
[3, 1, 4, 3]       // Seq of int
[4+2, 1+5, a*b]    // Seq of int asigned as an aritmetic expression

seq(5, i => i * i) // Is equivalent to [0, 1, 4, 9, 16]
```

Las operaciones relacionales que aceptan son las siguientes:

```
a  ==  b   // Equality
a  !=  b   // Disequality
a  <   b   // Proper prefix
a  <=  b   // Prefix
a  +   b   // Concatenation
|a|        // Cardinality
a[i]       // Selection, returns a type T
a[i := e]  // Update, returns a seq<T>
a  in  b   // Membership
a !in  b   // Non-membership
a[lo..hi]  // Subsequence
a[lo..]    // Drop
a[..hi]    // Take
a[slices]  // Slice, returns seq<seq<T>>
mutiset(a) // Conversion to multiset
```

3.3 Expresiones cuantificadoras

Se utilizan las expresiones *forall* y *exists* para determinar la mayoría de las invariantes y post-condiciones de las implementaciones.

Son expresiones *booleanas* que especifican que una determinada expresión es verdadera para todos (*forall*) o para alguna (*exists*) combinación de valores de las variables cuantificadas. Dafny infiere el tipo de las variables en caso de que no sean dadas en las declaraciones.

Forall

La expresión cuantificadora *forall* se denota de la siguiente manera, siendo V las variables a utilizar, C la condición de acotamiento de dichas variables e I la expresión *boolean* que deben cumplir todas las variables.

```
forall V :: C ==> I
```

Exists

También existe la expresión cuantificadora *exists* que se denota de la siguiente manera, siendo V las variables a utilizar, C la condición de acotamiento de dichas variables e I la expresión *boolean* que debe cumplir al menos una vez las variables.

```
exists V :: C ==> I
```

3.4 Métodos

Los métodos son fragmentos de código imperativo y ejecutables. Como en la mayoría de los lenguajes imperativos, el cuerpo del método contiene una serie de declaraciones.

Además, también puede tener entre cero y n cláusulas *requires*, *ensures*, *decreases* y *modifies*, que serán explicadas en las siguientes secciones.

```
method Name(n: T) returns (result: T)
  requires A    // Pre-condición
  ensures B     // Post-condición
  decreases C   // Terminación
  modifies D    // Accesibilidad
{
    // Statements
}
```

3.5 Pre-condiciones

Las pre-condiciones son declaradas utilizando la cláusula *requires* y representan las condiciones que deben cumplir los parámetros recibidos en el método para que sean válidos. En caso de no haber pre-condiciones, se puede omitir la cláusula:

```
requires A
```

También se puede expresar la ausencia de pre-condiciones de la siguiente manera:

```
requires true
```

3.6 Post-condiciones

Las post-condiciones son la tesis del algoritmo, es decir, lo que se desea demostrar para verificar formalmente el correcto funcionamiento.

```
ensures A
```

Pueden existir múltiples lineas *ensures* si el algoritmo requiere más de una post-condición.

3.7 Invariantes

Las invariantes son utilizadas en las iteraciones para marcar que ciertas condiciones son mantenidas sin variación en cada paso.

Generalmente son útiles para determinar acotamiento de variables y para ayudar a demostrar las post-condiciones establecidas.

```
while (C)
    invariant 0 <= A <= N  // Invariante de acotamiento
    invariant B            // Invariante de una condición B
{
    // Statements
}
```

3.8 Terminación

La cláusula *decreases* se utiliza para probar la terminación de un programa en la presencia de una recursión o iteración. En otras palabras, para la declaración de *variantes*. La forma de especificarlo en Dafny es la siguiente:

```
decreases A, B
```

Admite una lista de expresiones separadas por coma que demuestran la terminación. En ese caso, el ordenamiento utilizado es el lexicográfico.

Como particularidad, se puede ver condicionales de este tipo:

```
decreases if A then B else C
```

Por otro lado, si no se quiere probar la terminación, se debe expresar la cláusula de la siguiente manera:

```
decreases *
```

3.9 Accesibilidad de datos

En algunos casos también es necesario especificar qué lugares en memoria una iteración tiene permitido modificar. Esto se realiza utilizando la cláusula *modifies*. [15]

Para permitir que se pueda modificar toda una estructura de datos completa A, se podría hacer de la siguiente manera:

```
modifies A
```

En caso de no querer que una estructura de datos sea modificada, pero sí leída, se puede establecer utilizando la cláusula *reads*.

```
reads A
```

3.10 Funciones

El cuerpo de una función contiene una única expresión y no tiene permiso para acceder a la memoria. Es utilizado principalmente para apoyar la especificación de invariantes, pre-condiciones, post-condiciones y terminaciones de algoritmos.

Un ejemplo de una función sería la cantidad de *inversions* que tiene una secuencia de elementos:

```
function inversions (s: seq<int>): set<(nat, nat)>
{
  set i: nat, j :nat | 0 <= i < j < |s| && s[i] > s[j] :: (i, j)
}
```

3.11 Predicados y lemas

Los predicados y lemas son muy utilizados en Dafny para facilitar la lectura de las condiciones que existen en las especificaciones formales.

Se optó por utilizar cuantificadores en lugar de predicados y lemas en Dafny porque en KeY no existen. La intención es que se pueda ver con facilidad una comparación entre KeY y Dafny utilizando una sintaxis lo más parecida posible entre ambos.

Para ver ejemplos del uso de predicados y lemas en Dafny, se puede ver la tesis "Lógica de programación en Dafny"[12] de Universidad ORT Uruguay donde se realiza la demostración formal del algoritmo *Tree Sort* utilizando varios lemas y predicados.

3.12 Instalación

1. Instalar *Visual Studio Code* como IDE.
2. Si es *Mac* o *Linux*, instalar *.NET 6.0*.
3. Instalar la extensión *Dafny* en *Visual Studio Code* que se encarga de instalar todas las dependencias excepto *.NET 6.0*. En la tesis se utiliza la versión *Dafny 3.8*. [16]

4 KeY

KeY es un proyecto que comenzó en 1998 por Reiner Hähnle, Wolfram Menzel y Peter Schmitt en la Universidad de Karlsruhe (ahora denominado *Karlsruhe Institute of Technology*) [17].

El objetivo de *KeY* es integrar metodologías de análisis formal de programas, como la especificación y verificación formal, utilizando un lenguaje común en el desarrollo de programas como es *Java*. [17]

Para verificar formalmente programas en *Java*, *KeY* utiliza el lenguaje *Java Modeling Language (JML)* [18] que es conocido ampliamente dentro de las comunidades de metodología formal y de desarrollo de lenguajes. [17]

Para poder verificar la especificación en *JML* se utiliza el *IDE* de *KeY*, en el cual se carga un archivo que contiene tanto el código java como la especificación *JML*.

A través de ciertas configuraciones para la estrategia de búsqueda de las pruebas (*Proof Search Strategy*), el programa *KeY* empieza a aplicar un conjunto interno de reglas que se van derivando en un árbol de demostración (*Proof Tree*).

Cuando el sistema logra cerrar las hojas de dicho árbol, se concluye que el programa se pudo verificar formalmente. Por otro lado, si el sistema no logra cerrar alguna hoja automáticamente, mostrará una lista de *goals* abiertos que eventualmente el usuario tendrá que ayudar aplicando ciertas reglas para poder cerrarlas.

El usuario también puede crear sus propias reglas y *scripts* (conjunto de reglas y macros) para ayudar al demostrador de *KeY*.

4.1 Tipos de datos

Los tipos de datos que se utilizan en la tesis son *int*, *boolean* y *char*.

```
boolean a = false   // Init a boolean type
char a = 'A'        // Init a char type
int a = -5          // Init an int type
```

4.2 Estructuras de datos

Como estructura de datos se utilizan *arrays*.

```
int[] a = {1, 3, 5}    // Init values
int[] a = new T[N]     // Default values with length N
```

4.3 Expresiones cuantificadoras

Se utilizan expresiones *forall* y *exists* para determinar la mayoría de las invariantes y post-condiciones de las implementaciones.

Son expresiones *booleans* que especifican si una expresión es verdadera para todas (*forall*) o para alguna (*exists*) combinación de valores de las variables cuantificadas.

Forall:

La expresión cuantificadora *forall* utiliza un conjunto de variables V, ciertas condiciones de acotamiento C e I, la expresión *boolean* que debe cumplir las variables.

```
(\forall V;
        C;
        I)
```

Un ejemplo de uso sería verificar si todos los elementos de un *array* son menores a 5:

```
(\forall int i;
        0 <= i && i < arr.length;
        arr[i] < 5);
```

Exists:

La expresión cuantificadora *exists* utiliza un conjunto de variables V, ciertas condiciones de acotamiento C e I, la expresión *boolean* que debe cumplir al menos una de las variables.

```
(\exists V;
        C;
        I)
```

Un ejemplo de uso sería verificar si al menos un elemento de un *array* tiene valor 5:

```
(\exists int i;
        0 <= i && i < arr.length;
        arr[i] == 5);
```

4.4 Métodos

Los métodos son fragmentos de código java imperativo y ejecutable.

En *KeY* las especificaciones en los métodos se deben escribir con un comentario en lenguaje *JML* arriba de la firma del mismo. Pueden contener entre cero y n cláusulas *requires*, *ensures*, *decreases*, *measured_by* y *assignable*.

```
/*@ public normal_behaviour
  @  requires A       // Pre-Condition
  @  ensures B        // Post-Condition
  @  decreases C      // Iteration termination metric
  @  measured_by C    // Recursion termination metric
  @  assignable D     // Accesibility
  @*/
public T' Name(T: a) {
    // Statements
}
```

4.5 Pre-condiciones

Las pre-condiciones son declaradas utilizando la cláusula *requires* y representan las condiciones *boolean* que deben cumplir los parámetros recibidos en el método para que sean válidos. En caso de no haber pre-condiciones, se puede omitir la cláusula.

```
/*@
  @  requires A
  @  requires B
  @*/
```

También se puede expresar la ausencia de pre-condiciones de la siguiente manera:

```
/*@
  @  requires true
  @*/
```

4.6 Post-condiciones

Las post-condiciones son expresiones *boolean* que se declaran utilizando la cláusula *ensures*. Son las condiciones que se deben demostrar como válidas para verificar formalmente el correcto funcionamiento del algoritmo.

```
/*@
  @  ensures A
  @  ensures B
  @*/
```

Pueden existir múltiples lineas *ensures* si el algoritmo requiere más de una post-condición.

4.7 Invariantes

Las invariantes son utilizadas en las iteraciones para marcar que ciertas condiciones son mantenidas sin variación en cada paso. Además, son útiles para determinar que una variable está acotada y para ayudar a demostrar las post-condiciones establecidas.

```
/*@  loop_invariant 0 <= A && A <= N  // Invariante de acotamiento
  @  loop_invariant B                 // Invariante de una condición B
  @*/
while (C) {
    // Statements
}
```

En el siguiente ejemplo se puede observar una invariante de acotamiento, que indica que el índice i está acotado entre cero y el largo del *array*. La otra invariante indica que todos los elementos que están a la izquierda de i tienen distinto valor que cinco.

```
/*@  loop_invariant 0 <= i && i <= arr.length  // Invariante de acotamiento
  @  loop_invariant (\forall int k;
  @                          0 <= k && k < i;
  @                          arr[k] != 5);
  @*/
while (i != arr.length) {
    // Statements
}
```

4.8 Terminación

La cláusula *decreases* se utiliza para probar la terminación de un programa o de una iteración. La forma de especificarlo en KeY es la siguiente.

```
/*@
  @  decreases A, B
  @*/
```

En caso de estar en contexto de recursión, se debe utilizar la cláusula *measured_by* en la especificación, la cual indica cómo va decreciendo determinada condición A en cada recursión.

```
/*@
  @  measured_by A
  @*/
```

Ambas formas admiten una lista de condiciones separadas por una coma que demuestran la terminación. La lista puede contener también un solo elemento.

Si no se quiere probar la terminación, existe otra cláusula denominada *diverges*.

```
/*@
  @  diverges true
  @*/
```

4.9 Accesibilidad de datos

La cláusula *assignable* de JML es utilizada para asignar permisos de escritura y lectura sobre variables en memoria dentro de un contexto.

En algunos casos que aparecen en la tesis, es necesario especificar que cualquier dirección de memoria de un array puede ser modificada.

```
/*@
  @  assignable arr[*]
  @*/
```

Por otro lado, existen casos en los que no se desea permitir que se modifique ninguna posición de memoria de un *array*.

```
/*@
  @  assignable \strictly_nothing
  @*/
```

4.10 Modelos

Los *model* en *KeY* representan un atributo no compilable (que puede ser una función) y se utilizan para facilitar ciertas demostraciones.

Un ejemplo sería declarar una función que representa a la sucesión de Fibonacci:

```
/*@  model_behavior
  @    requires n >= 0;
  @    model int fib(int n) {
  @      return (n == 0 || n == 1) ? 1 : fib(n-1) + fib(n-2);
  @    }
  @*/
```

Este ejemplo de *model* puede ser utilizado para comprobar que una variable A tenga como valor cierta posición en la sucesión de Fibonacci.

```
/*@  loop_invariant a == fib(i)
  @*/
while (C) {
    // Statements
}
```

4.11 JavaDL

Java Dynamic Logic permite expandir la anotación de *JML* utilizada para realizar la especificación formal.

En la tesis se utiliza *JavaDL* para probar las propiedades de integridad de los algoritmos de ordenamiento.

```
/*@ public normal_behavior
  @  ...
  @  ensures \dl_seqPerm(\dl_array2seq(arr), \old(\dl_array2seq(arr)));
  @  ...
  @*/
public void sort(int[] arr) {

    /*@  ...
      @  loop_invariant \dl_seqPerm(\dl_array2seq(arr), \old(\dl_array2seq(arr)))
      @  ...
      @*/
    while(C) {
        // Statements
    }
}
```

Como se puede observar, se utiliza *dl_seqPerm* para verificar que existe una permutación de dos secuencias y se utiliza *dl_array2seq* para convertir el *array* a secuencia.

4.12 Instalación

1. Tener instalado Java versión 8 o más nueva.
2. Descargar el archivo *key-2.10.0-exe.jar* desde la página oficial de *KeY* [10].

4.13 Configuración del IDE

Sobre el *layout*, se puede ver que la pantalla principal está dividida en tres secciones y una barra de herramientas. A continuación se detallan configuraciones básicas del *IDE* para poder reproducir los ejemplos de la tesis.

1. Sección *Source*:

   Aparece el archivo *Java* que se carga en el *IDE*.

2. Sección *Sequent*:

   Es una interfaz gráfica donde aparece el código y la especificación formal en formato *sequent*. Se puede seleccionar con el puntero cada *sequent* y el *IDE* resalta en la sección *Source* las líneas de código vinculadas a dicho *sequent*.

   Principalmente esta sección se utiliza cuando queda algún *goal* pendiente de cerrar, aquí es donde se deben aplicar reglas manualmente para intentar ayudar al verificador a cerrar dichos *goals*.

Para poder entender con mayor facilidad los *sequent* que se muestran, se recomienda seleccionar el *checkbox* de *View* > *User Pretty Syntax*.

3. Sección con configuraciones:

   En la pestaña *Proof* aparece un árbol con las reglas que *KeY* aplica sobre los *sequents*.

   En la pestaña *Goals*, se pueden ver los nodos del árbol que están pendientes de cerrar. Cuando se cierran todos los *goals* implica que se pudo demostrar la especificación formal del algoritmo.

   En la pestaña *Proof Search Strategy* aparecen configuraciones que toma en cuenta el verificador cuando aplica las reglas sobre los *sequent*. Se debe seleccionar un ambiente de configuración predeterminado denominado *Java verif. std.*

4. Barra de herramientas:

   En *Options* > *Show settings* > *Taclet Options* se debe seleccionar *moreSeqRules:on*, ya que se requiere para que el verificador pueda trabajar con las pruebas relacionadas a las permutaciones.

4.14 Scripts

Los *scripts* son archivos *batch* que se pueden cargar en el *IDE* de *KeY* para realizar una verificación de un algoritmo. Generalmente se utiliza cuando el algoritmo no puede ser demostrado automáticamente por el verificador de *KeY*.

Un *script* está compuesto por un archivo *.key* y uno *.script*.

Archivo *.key*

1. *Settings*:

   Contiene configuraciones del *IDE* que se tienen que agregar para poder correr exitosamente el *script*.

```
\settings {
  "[Choice]DefaultChoices=moreSeqRules-moreSeqRules:on
   [Strategy]MaximumNumberOfAutomaticApplications=10000
   [StrategyProperty]OSS_OPTIONS_KEY=OSS_ON
   [StrategyProperty]LOOP_OPTIONS_KEY=LOOP_INVARIANT"
}
```

2. *Java source*:

   Contiene la ruta hacia el archivo *Java* que se tiene que probar. Generalmente está en la misma carpeta, por ese motivo, se utiliza:

```
\javaSource ".";
```

3. *Choose contract*:

   Se elige el contrato que se desea probar, generalmente cuando existen varios métodos en una mismo archivo *Java*, uno puede seleccionar cuál de todas las especificaciones *JML* se desea probar con el *script*.

```
\chooseContract "Quicksort[Quicksort::partition([I,int,int)].JML normal_behavior
      operation contract.0";
```

4. *Proof script*:

   Contiene la ruta del *script* donde están las reglas y macros a aplicar.

```
\proofScript "script 'Partition.script';"
```

Archivo *.script*

Es un archivo *batch*, que se ejecuta por precedencia desde el comienzo del archivo hasta el final y contiene una serie de macros y reglas.

```
macro autopilot-prep;

tryclose;

macro simp-upd;

rule seqPermFromSwap;
rule andRight;
  auto;
```

En este caso en particular, una vez que finaliza el macro del piloto automático, queda un *goal* abierto que el verificador no puede cerrar automáticamente.

Por esa razón, se debe aplicar manualmente un conjunto de reglas a través del comando *rule*.

Más información sobre *scripts* en *KeY* puede ser encontrada en el documento *KeY Developer Documentation* [19], aunque la información está bastante incompleta.

5 Algoritmos aritméticos

Este capítulo aborda algoritmos aritméticos conocidos, de complejidad baja, para poder introducir la metodología de derivación formal propuesta en la tesis y ambas herramientas.

5.1 División entera

Dado un dividendo y un divisor, se retorna el cociente y el resto que da la división entera entre dichos valores, utilizando sólo suma y resta.

5.1.1 Metodología

```
1  IntegerDivision (D: int, d: int): (q: int, r: int)
2    require
3      D >= 0    -- Requerimiento de acotamiento
4      d > 0     -- Requerimiento de acotamiento
5    ensure
6      r in [0..d)      -- Propiedad de acotamiento
7      D = d * q + r    -- Propiedad esencial
8    local
9      r: int
10     q: int
11   do
12     invariant
13       r >= 0           -- Invariante de acotamiento
14       D = d * q + r
15     init
16       r := D
17       q := 0
18     until
19       r < d
20     step
21       r := r - d
22       q := q + 1
23     variant
24       r
25     end
26   end
```

Se recibe por parámetro al dividendo y divisor representados con las variables D y d.

Como pre-condición, el dividendo tiene que ser cero o más, mientras que el divisor tiene que ser mayor a cero.

$$\mathrm{D} \geq 0 \qquad (3 :: \text{require})$$

$$\mathrm{d} > 0 \qquad (4 :: \text{require})$$

El algoritmo tiene dos post-condiciones:

1. El resto está acotado entre [0..d].

$$\mathrm{r} \in [0..\mathrm{d}) \qquad (6 :: \text{ensure})$$

2. Se cumple la propiedad de la división.

$$\mathrm{D} := \mathrm{d} * \mathrm{q} + \mathrm{r} \qquad (7 :: \text{ensure})$$

Las variables q y r, representan al cociente y resto, que son el retorno del algoritmo.

Para probar la post-condición de la propiedad de la división, se utiliza la terminación del algoritmo mientras se mantiene la siguiente invariante en cada paso:

$$\mathrm{D} := \mathrm{d} * \mathrm{q} + \mathrm{r}. \qquad (14 :: \text{invariant})$$

Para probar la post-condición de la propiedad de acotamiento del resto, también se utiliza la terminación del algoritmo, junto con la siguiente invariante:

$$\mathrm{r} \geq 0. \qquad (13 :: \text{invariant})$$

Se utiliza a la variable r inicializada en el dividendo D y el cociente q inicializado en cero.

$$\mathrm{r} := \mathrm{D} \qquad (16 :: \text{init})$$

$$\mathrm{q} := 0. \qquad (17 :: \text{init})$$

La variante del algoritmo es el resto r, ya que será el que va a ir decreciendo.

En cada paso, se resta al resto r una unidad de divisor d. Por ende, para mantener la invariante esencial, se debe incrementar en una unidad el cociente q.

$$r := r - d \qquad (21 :: \text{step})$$

$$q := q + 1. \qquad (22 :: \text{step})$$

Como terminación del algoritmo, el resto debe decrementar hasta ser menor que el divisor *d*.

$$r < d. \qquad (19 :: \text{until})$$

En este caso, esta condición de terminación junto con la invariante de acotamiento del resto, que se mantuvo en cada paso:

$$r \geq 0 \qquad (13 :: \text{invariant})$$

demuestran que la post-condición de acotamiento del resto se cumple:

$$r \in [0..d). \qquad (6 :: \text{ensure})$$

También se logra probar la post-condición de la propiedad de la división, ya que se mantuvo la invariante siguiente en todos los pasos:

$$D := d * q + r, \qquad (14 :: \text{invariant})$$

derivando en la post-condición:

$$D := d * q + r \qquad (7 :: \text{ensure})$$

5.1.2 Implementación en Dafny

```
1  method IntegerDivision (D: int, d: int) returns (q: int, r: int)
2    requires D >= 0
3    requires d > 0
4    ensures 0 <= r < d
5    ensures D == d * q + r
6  {
7    r := D;
8    q := 0;
9
10   while (r >= d)
11     invariant r >= 0
12     invariant D == d * q + r
13     decreases r
14   {
15     r := r - d;
16     q := q + 1;
17   }
18 }
```

5.1.3 Implementación en KeY

```
1  private int q;
2  private int r;
3
4  /*@ public normal_behavior
5    @  requires D >= 0;
6    @  requires d > 0;
7    @  ensures 0 <= r < d;
8    @  ensures D == d * q + r;
9    @*/
10 public void integerDivision(int D, int d) {
11   r = D;
12   q = 0;
13
14   /*@  loop_invariant r >= 0;
15     @  loop_invariant D == d * q + r;
16     @  decreases r;
17     @*/
18   while(r >= d) {
19     r = r - d;
20     q = q + 1;
21   }
22 }
```

5.2 Potenciación

La potenciación es la multiplicación de un número denominado base por sí mismo una cantidad determinada de veces.

5.2.1 Metodología

```
1  Power (base: int, exp: int): (result: int)
2    require
3      exp >= 0    -- Requerimiento de acotamiento
4    ensure
5      result = base^exp    -- Propiedad esencial
6    local
7      e: int
8    do
9      invariant
10       e >= 0                          -- Invariante de acotamiento
11       result * base^e = base^exp      -- Invariante esencial
12     init
13       e := exp
14       result := 1
15     until
16       e = 0
17     step
18       result := result * base
19       e := e - 1
20     variant
21       e
22     end
23   end
```

Se recibe por parámetro la base y el exponente que se desea calcular. La base es el número que se multiplica y el exponente indica la cantidad de veces que se tiene que multiplicar dicha base.

Como requerimiento para este algoritmo, sólo son válidos los exponentes naturales:

$$\text{exp} \geq 0. \qquad (3 :: \text{require})$$

La post-condición implica que se retorne como resultado la potencia solicitada.

$$\text{result} = base^{exp}. \qquad (5 :: \text{ensure})$$

Para demostrar la post-condición, se utiliza el método “sustituir constante por variable”, sustituyendo la constante *exp* por la variable *e*, la cual representa la cantidad de veces que falta multiplicar la base por sí misma.

e. (21 :: variant)

La variable *e*, al representar el exponente, se inicializa en *exp* y por lo tanto, tampoco podrá ser menor a 0. Se deriva en la invariante de acotamiento siguiente:

$$e \geq 0 \qquad (10 :: \text{invariant})$$

$$e := exp. \qquad (13 :: \text{init})$$

Además, se utiliza el método "invariante de cola", que incorpora a la variable acumuladora *result*, junto con su invariante esencial. Se inicializa en 1 por ser el neutro del producto.

$$result * base^{e} = base^{exp} \qquad (11 :: \text{invariant})$$

$$result := 1. \qquad (14 :: \text{init})$$

En cada paso se multiplica la base con el acumulador *result* y se resta una unidad del exponente *e*.

$$result := result * base \qquad (18 :: \text{step})$$

$$e := e - 1 \qquad (19 :: \text{step})$$

La condición de terminación se cumple cuando *e* llega a 0.

$$e = 0 \qquad (16 :: \text{until})$$

En este caso, con $e = 0$, $base^{e} = base^{0} = 1$, por lo tanto, la invariante esencial:

$$result * base^{e} = base^{exp} \qquad (11 :: \text{invariant})$$

deriva en la post-condición:

$$result = base^{exp}. \qquad (5 :: \text{ensure})$$

5.2.2 Implementación en Dafny

```
method Power(base: int, exp: nat) returns (result: int)
  ensures result == Pow(base, exp)
{
  var e := exp;
  result := 1;

  while(e != 0)
    invariant e >= 0
    invariant result * Pow(base, e) == Pow(base, exp)
    decreases e
  {
    result := result * base;
    e := e - 1;
  }
}

function Pow(base: int, exp: nat):int
{
  if (exp == 0) then 1
  else base * Pow(base, exp-1)
}
```

Para poder determinar que la potenciación es correcta, se utiliza una función recursiva denominada *Pow*.

Su resultado equivale a la potenciación aritmética de una base elevada a un exponente natural.

5.2.3 Implementación en KeY

```
/*@ model_behavior
  @  requires exp >= 0;
  @  model int pow(int base, int exp) {
  @    return (exp == 0) ? 1 : base * pow(base, exp-1);
  @  }
  @*/

/*@ public normal_behavior
  @  requires exp >= 0;
  @  ensures \result == pow(base, exp);
  @*/
public int power(int base, int exp) {

  int e = exp;
  int result = 1;

  /*@  loop_invariant e >= 0;
    @  loop_invariant result * pow(base, e) == pow(base, exp);
    @  decreases e;
    @*/
  while(e > 0) {
    result = result * base;
    e = e - 1;
  }

  return result;
}
```

Para poder determinar que la potenciación es correcta, se utiliza un *model* para inicializar una función recursiva denominada *Pow*. Su resultado equivale a la potenciación aritmética de una base elevada a un exponente natural.

5.3 Sucesión de Fibonacci

Es una secuencia de números naturales, que se van sumando de a pares de tal forma que cada número es igual a la suma de los dos anteriores. Este algoritmo retorna el n-ésimo número de la sucesión de Fibonacci.

La definición matemática para la sucesión de Fibonacci es la siguiente:

$$\begin{cases} fibonacci(0) = 1 \\ fibonacci(1) = 1 \\ fibonacci(n+2) = fibonacci(n) + fibonacci(n+1) \end{cases} . \tag{5.1}$$

5.3.1 Metodología

```
1  Fibonacci (n: int): (a: int)
2    require
3      n >= 0    -- Requerimiento de acotamiento
4    ensure
5      a = fib(n)    -- Propiedad esencial
6    local
7      a: int
8      b: int
9      i: int
10     temp: int
11   do
12     invariant
13       i in [0..n]       -- Invariante de acotamiento
14       a = fib(i)        -- Invariante esencial
15       b = fib(i + 1)    -- Invariante esencial
16     init
17       a := 1
18       b := 1
19       i := 0
20     until
21       i = n
22     step
23       temp := a
24       a := b
25       b := temp + b
26       i := i + 1
27     variant
28       n - i
29     end
30   end
```

Al retornar el n-ésimo número de la sucesión de Fibonacci, se requiere que el valor de n recibido por parámetro, sea un número natural.

$$\text{n} \geq 0. \qquad (3 :: \text{require})$$

Como post-condición, se retorna el n-ésimo número de la sucesión de Fibonacci.

$$\text{a} = \text{fib(n)}. \qquad (5 :: \text{ensure})$$

Para demostrar la post-condición, se utiliza el método "sustituir constante por variable" y se sustituye la constante n por la variable i. Determinando la siguiente invariante esencial:

$$\text{a} = \text{fib(i)}. \qquad (14 :: \text{invariant})$$

De cierta forma, la variable a, está representando el i-ésimo número en la sucesión de Fibonacci.

Inherentemente al cómputo que se realiza para calcular el siguiente número en la sucesión de Fibonacci, se necesita tener dos elementos sucesivos (k y k + 1). Por lo tanto, se utiliza el método "fortalecimiento de invariante" que deriva también la siguiente invariante esencial:

$$\text{b} = \text{fib(i + 1)}. \qquad (15 :: \text{invariant})$$

Además, como i representa al i-ésimo número de la sucesión de Fibonacci, entonces i debe estar acotado entre 0 y n.

$$\text{i} \in [0..\text{N}]. \qquad (13 :: \text{invariant})$$

Se inicializa el índice i, que delimita las particiones, en 0.

$$\text{i} := 0. \qquad (19 :: \text{init})$$

Se inicializan las variables que contendrán los resultados parciales de la sucesión de Fibonacci a y b, en los casos base:

$$\text{a} := 1 \qquad (17 :: \text{init})$$

$$\text{b} := 1. \qquad (18 :: \text{init})$$

De la partición que determina las veces que faltan por computar, se puede derivar la variante.

$$\text{n - i}. \qquad (28 :: \text{variant})$$

En cada paso, se realiza el cómputo del siguiente número en la sucesión de Fibonacci.

$$\text{temp} := \text{a} \qquad (23 :: \text{step})$$

$$a := b \qquad (24 :: step)$$

$$b := temp + b. \qquad (25 :: step)$$

De esta forma, se mantienen ambas invariantes esenciales:

$$a = fib(i) \qquad (14 :: invariant)$$

$$b = fib(i + 1) \qquad (15 :: invariant)$$

Antes de finalizar el paso, se incrementa el índice i en una unidad.

$$i := i + 1. \qquad (26 :: step)$$

Cuando el índice i llega a valer n, se finalizan los pasos.

$$i = n. \qquad (21 :: until)$$

En este caso, cuando $i = n$, la invariante esencial determinada por la variable a deriva en la post-condición.

$$a = fib(i) \qquad (14 :: invariant)$$

$$a = fib(n). \qquad (5 :: ensure)$$

5.3.2 Implementación en Dafny

```
1  method FibonacciIterative(n: int) returns (a: int)
2    requires n >= 0
3    ensures a == FibonacciRecursive(n)
4    decreases n
5  {
6    a := 1;
7    var b: int := 1;
8    var i: int := 0;
9
10   while (i != n)
11     invariant 0 <= i <= n
12     invariant a == FibonacciRecursive(i)
13     invariant b == FibonacciRecursive(i+1)
14     decreases n - i
15   {
16     var olda: int := a;
17     a := b;
18     b := olda + b;
19     i := i + 1;
20   }
21 }
```

Se debe utilizar una función recursiva para demostrar el método iterativo:

```
function FibonacciRecursive(n: nat): nat
  decreases n
{
  if (n == 0) then 1 else
  if (n == 1) then 1 else
  FibonacciRecursive(n-2) + FibonacciRecursive(n-1)
}
```

5.3.3 Implementación en KeY

```
/*@ public normal_behavior
  @  requires n >= 0;
  @  ensures \result == fib(n);
  @  decreases n;
  @*/
public int fib_iterative(int n) {
  int a = 1;
  int b = 1;
  int i = 0;

  /*@  loop_invariant 0 <= i && i <= n;
    @  loop_invariant a == fib(i);
    @  loop_invariant b == fib(i+1);
    @  assignable \strictly_nothing;
    @  decreases n - i;
    @*/
  while (i != n) {
    int olda = a;
    a = b;
    b = olda + b;
    i++;
  }
  return a;
}
```

Como se puede observar, se crea un *model* con la definición de la función Fibonacci de manera recursiva. Es necesaria como base para poder probar la función iterativa.

De todas formas, se realiza también una función recursiva en *KeY* para poder compararla con *Dafny*:

```
/*@ model_behavior
  @  requires n >= 0;
  @  model int fib(int n) {
  @    return (n == 0 || n == 1) ? 1 : fib(n-1) + fib(n-2);
  @  }
  @*/

/*@ public normal_behavior
  @  requires n >= 0;
  @  ensures \result == fib(n);
  @  measured_by n;
```

```
12    @  assignable \strictly_nothing;
13    @*/
14  public int fib_recursive(int n) {
15    if (n == 0 || n == 1) {
16      return 1;
17    }
18    return fib_recursive(n-2) + fib_recursive(n-1);
19  }
```

5.4 Suma de elementos de un array

Recorre un *array* linealmente de izquierda a derecha y devuelve un valor que representa la sumatoria de sus elementos.

5.4.1 Metodología

```
1  SumArray (arr: array<int>): (result: int)
2    require
3      true    -- No tiene requerimientos
4    ensure
5      result = sum k when k in [0..N) of arr[k]    -- Propiedad esencial
6    local
7      N: int
8      i: int
9    do
10     invariant
11       i in [0..N]                                  -- Invariante de acotamiento
12       result = sum k when k in [0..i) of arr[k]    -- Invariante esencial
13     init
14       N := arr.length
15       result := 0
16       i := 0
17     until
18       i = N
19     step
20       result := result + arr[i]
21       i := i + 1
22     variant
23       N - i
24     end
25   end
```

El algoritmo recibe como parámetro un *array* donde hay que realizar las operaciones.

Como post-condición, el resultado debe ser la sumatoria de todos los elementos del *array*.

$$\text{result} = \sum_{k=0}^{N-1} arr[k]. \qquad (5 :: \text{ensure})$$

Para poder realizar la demostración, se utiliza el método "invariante de cola", para plantear el resultado final como un cómputo parcial acumulador de resultado, sobre la variable *result*.

Además, también se utiliza el método "sustituir constante por variable", sustituyendo la constante N por la variable i, de tal forma que los elementos que están a la izquierda de i ya fueron computados en el acumulador y los elementos que están a la derecha de i restan por computar.

De la partición computada se deriva la invariante:

$$\text{result} = \sum_{k=0}^{i-1} arr[k]. \qquad (12 :: \text{invariant})$$

De la partición por computar, se deriva la variante:

$$\text{N - i.} \qquad (23 :: \text{variant})$$

El índice i se inicializa en 0 y queda acotado entre [0..N]:

$$\text{i} := 0 \qquad (16 :: \text{init})$$

$$\text{i} \in [0..\text{N}]. \qquad (11 :: \text{invariant})$$

La variable acumuladora *result*, se inicializa en 0 por ser el neutro de la suma.

$$\text{result} := 0 \qquad (15 :: \text{init})$$

En cada paso, se agrega la suma del elemento en la posición i a la acumulación de la variable *result* (se pasa al elemento de la partición por computar, a la partición computada).

$$\text{result} := \text{result} + \text{arr[i]}. \qquad (20 :: \text{step})$$

Para poder continuar con la acumulación, se incrementa el índice i en una unidad.

$$\text{i} := \text{i} + 1. \qquad (21 :: \text{step})$$

La condición de terminación se cumple cuando el índice i termina de recorrer todo el *array*.

$$\text{i} = \text{N}. \qquad (18 :: \text{until})$$

En ese caso, como $i = N$, la invariante esencial que tenía el cálculo parcial de la sumatoria de todos los elementos a la izquierda de la posición i, deriva en la post-condición.

$$\text{result} = \sum_{k=0}^{i-1} arr[k] \qquad (12 :: \text{invariant})$$

$$\text{result} = \sum_{k=0}^{N-1} arr[k]. \qquad (5 :: \text{ensure})$$

5.4.2 Implementación en Dafny

```
method SumArrayIterative(arr: array<int>) returns (result: int)
  ensures result == SumArrayRecursive(arr, arr.Length-1)
{
  var N: nat := arr.Length;
  result := 0;
  var i: nat := 0;

  while (i != N)
    invariant 0 <= i <= N
    invariant result == SumArrayRecursive(arr, i-1);
    decreases N - i;
  {
    result := result + arr[i];
    i := i + 1;
  }
}
```

Para poder demostrar la post-condición marcada en la sección anterior, se debe utilizar una función recursiva que realice la suma parcial hasta determinada posición del *array*:

```
function SumArrayRecursive (arr: array<int>, to: int): int
  requires -1 <= to < arr.Length
  decreases to
  reads arr
{
  if to == -1 then 0
  else arr[to] + SumArrayRecursive(arr, to - 1)
}
```

5.4.3 Implementación en KeY

```
/*@ public normal_behavior
  @  requires arr != null;
  @  ensures \result == (\sum int k;
  @                          0 <= k && k < arr.length;
  @                          arr[k]);
  @  assignable \nothing;
  @*/
public int sumArrayIterative(int[] arr) {
  int N = arr.length;
  int result = 0;
  int i = 0;

  /*@  loop_invariant 0 <= i && i <= N;
    @  loop_invariant result == (\sum int k;
    @                            0 <= k && k < i;
    @                            arr[k]);
    @  assignable \nothing;
    @  decreases N - i;
    @*/
  while (i != N) {
    result = result + arr[i];
    i = i + 1;
  }
```

```
24    return result;
25  }
```

Se utiliza un cuantificador generalizado denominado *sum*, que en este caso, retorna la suma matemática de cada uno de los elementos del *array*. Por esa razón, en la cláusula *ensures*, el cuantificador generalizado es utilizado para probar que el resultado sea el esperado. No es necesario realizar una función recursiva para poder probar el método iterativo.

De todas formas, se realiza la función recursiva para su comparación con la realizada en *Dafny*:

```
1   /*@ public normal_behavior
2     @  ensures \result == (\sum int k;
3     @                          0 <= k && k < arr.length;
4     @                          arr[k]);
5     @  assignable \nothing;
6     @*/
7   public int sumArrayRecursive(int[] arr) {
8     return sumArrayRecursiveAux(arr, 0);
9   }
10
11  /*@ public normal_behavior
12    @  ensures \result == (\sum int k;
13    @                          0 <= k && k < arr.length;
14    @                          arr[k]);
15    @  measured_by arr.length - i;
16    @  assignable \nothing;
17    @*/
18  public int sumArrayRecursiveAux(int[] arr, int i) {
19    if (i == arr.length) {
20      return 0;
21    }
22    return arr[i] + sumArrayRecursiveAux(arr, ++i);
23  }
```

5.5 Producto de elementos de un array

Recorre un *array* de izquierda a derecha y devuelve un valor que representa la productoria de sus elementos.

5.5.1 Metodología

```
1  ProdArray (arr: array<int>): (result: int)
2    require
3      true    -- No tiene requerimientos
4    ensure
5      result = prod k when k in [0..N) of arr[k]    -- Propiedad esencial
6    local
7      N: int
8      i: int
9    do
10     invariant
11       i in [0..N]                                  -- Invariante de acotamiento
12       result = prod k when k in [0..i) of arr[k]   -- Invariante esencial
13     init
14       N := arr.length
15       result := 1
16       i := 0
17     until
18       i = N
19     step
20       result := result * arr[i]
21       i := i + 1
22     variant
23       N - i
24     end
25   end
```

El algoritmo recibe como parámetro un *array* donde hay que realizar las operaciones.

Como post-condición, el resultado debe ser la productoria de todos los elementos del *array*.

$$\text{result} = \prod_{k=0}^{N-1} arr[k]. \qquad (5 :: \text{ensure})$$

Para poder realizar la demostración, se utiliza el método "invariante de cola", para plantear el resultado final como un cómputo parcial acumulador de resultado, sobre la variable *result*.

Además, también se utiliza el método "sustituir constante por variable", sustituyendo la constante N por la variable i, de tal forma que los elementos que están a la izquierda de i ya fueron computados en el acumulador y los elementos que están a la derecha de i restan por computar.

De la partición computada se deriva la invariante:

$$\text{result} = \prod_{k=0}^{i-1} arr[k]. \qquad (12 :: \text{invariant})$$

De la partición por computar, se deriva la variante:

$$\text{N - i.} \qquad (23 :: \text{variant})$$

El índice i se inicializa en 0 y queda acotado entre [0..N]:

$$\text{i} := 0 \qquad (16 :: \text{init})$$

$$\text{i} \in [0..\text{N}]. \qquad (11 :: \text{invariant})$$

La variable acumuladora *result*, se inicializa en 1 por ser el neutro del producto.

$$\text{result} := 1. \qquad (15 :: \text{init})$$

En cada paso, se agrega el producto del elemento en la posición i a la acumulación de la variable *result* (se pasa al elemento de la partición por computar, a la partición computada).

$$\text{result} := \text{result} * \text{arr[i]}. \qquad (20 :: \text{step})$$

Para poder continuar con la acumulación, se incrementa el índice i en una unidad.

$$\text{i} := \text{i} + 1. \qquad (21 :: \text{step})$$

La condición de terminación se cumple cuando el índice i termina de recorrer todo el *array*.

$$\text{i} = \text{N}. \qquad (18 :: \text{until})$$

En ese caso, como $i = N$, la invariante esencial que tenía el cálculo parcial de la productoria de todos los elementos a la izquierda de la posición i, deriva en la post-condición.

$$\text{result} = \prod_{k=0}^{i-1} arr[k] \qquad (12 :: \text{invariant})$$

$$\text{result} = \prod_{k=0}^{N-1} arr[k]. \qquad (5 :: \text{ensure})$$

5.5.2 Implementación en Dafny

```
method ProdArrayIterative(arr: array<int>) returns (result: int)
  ensures result == ProdArrayRecursive(arr, arr.Length-1)
{
  var N: nat := arr.Length;
  result := 1;
  var i: nat := 0;

  while (i != N)
    invariant 0 <= i <= N
    invariant result == ProdArrayRecursive(arr, i-1);
    decreases arr.Length - i;
  {
    result := result * arr[i];
    i := i + 1;
  }
}
```

Para poder demostrar la post-condición marcada en la sección anterior, se debe utilizar una función recursiva que representa la suma parcial hasta determinada posición del *array*:

```
function ProdArrayRecursive (arr: array<int>, to: int): int
  requires -1 <= to < arr.Length
  decreases to
  reads arr
{
  if to == -1 then 1
  else arr[to] * ProdArrayRecursive(arr, to - 1)
}
```

5.5.3 Implementación en KeY

```
/*@ public normal_behavior
  @  ensures \result == (\product int k;
  @                                0 <= k && k < arr.length;
  @                                arr[k]);
  @  assignable \nothing;
  @*/
public int prodArrayIterative(int[] arr) {
  int N = arr.length;
  int result = 1;
  int i = 0;

  /*@  loop_invariant 0 <= i && i <= N;
    @  loop_invariant result == (\product int k;
    @                                    0 <= k && k < i;
    @                                    arr[k]);
    @  assignable \nothing;
    @  decreases N - i;
    @*/
  while (i != N) {
    result = result * arr[i];
    i = i + 1;
  }
  return result;
}
```

Para poder probar que el algoritmo cumple con su especificación, se utiliza un nuevo cuantificador generalizado denominado *product*, que en este caso, retorna el producto matemático de cada uno de los elementos del *array*. Por esa razón, en la cláusula *ensures*, el cuantificador generalizado es utilizado para probar que el resultado sea el esperado. No es necesario realizar una función recursiva para poder probar el método iterativo.

De todas formas, se realiza la función recursiva para su comparación con la realizada en *Dafny*:

```
1  /*@ public normal_behavior
2    @  requires arr != null;
3    @  ensures \result == (\product int k;
4    @                              0 <= k && k < arr.length;
5    @                              arr[k]);
6    @  assignable \nothing;
7    @*/
8    public int prodArrayRecursive(int[] arr) {
9      return prodArrayRecursiveAux(arr, 0);
10   }
11
12 /*@ public normal_behavior
13   @  requires arr != null;
14   @  ensures \result == (\product int k;
15   @                              0 <= k && k < arr.length;
16   @                              arr[k]);
17   @  assignable \nothing;
18   @*/
19 public int prodArrayRecursiveAux(int[] arr, int i) {
20   if (i == arr.length) {
21     return 1;
22   }
23   return arr[i] * prodArrayRecursiveAux(arr, ++i);
24 }
```

6 Algoritmos de búsqueda

6.1 Búsqueda lineal

Es un algoritmo de búsqueda sobre un *array*, que lo recorre de izquierda a derecha buscando el elemento que es pasado por parámetro. Al encontrar una coincidencia, devuelve el índice de dicho elemento en el *array*. En caso de terminar el recorrido sin haber encontrado ninguna coincidencia, devuelve el valor -1.

6.1.1 Metodología

```
1  LinearSearch (arr: array<int>, key: int): (i: int)
2    require
3      true    -- No tiene requerimientos
4    ensure
5      i = -1 then key not in arr[..]    -- Propiedad caso negativo
6      i in [0..N) then key = arr[i]     -- Propiedad caso positivo
7    local
8      N: int
9    do
10     invariant
11       i in [0..N]             -- Invariante de acotamiento
12       key not in arr[0..i)    -- Invariante esencial
13     init
14       i := 0
15       N := arr.length
16     until
17       i = N or arr[i] = key
18     step
19       i := i + 1
20     variant
21       N - i
22     adjust
23       if i = N then -1
24     end
25   end
```

El algoritmo tiene dos post-condiciones:

1. Propiedad que se cumple cuando no se encuentra el elemento buscado:

$$\text{i} = \text{-1} \Rightarrow \text{key} \notin \text{arr}[..]. \qquad (6 :: \text{ensure})$$

2. Propiedad que se cumple cuando se encuentra el elemento buscado:

$$0 \leq \text{i} < \text{N} \Rightarrow \text{key} = \text{arr}[\text{i}]. \qquad (7 :: \text{ensure})$$

Por parámetro, se recibe un *array* de elementos como estructura de datos y un valor a buscar *key*.

Se aplica el método de "seleccionar condiciones" para probar las post-condiciones. De tal forma, que la post-condición positiva se prueba utilizando la condición de terminación y la post-condición negativa utilizando una invariante.

Se determina que existe una partición que tiene los elementos ya recorridos y otra partición que tiene los elementos por recorrer.

De la partición con elementos ya recorridos, se deriva la invariante que mantiene la propiedad de que cada elemento ya recorrido no puede ser el elemento buscado. Esto se realiza con el método "sustituir constante por variable" en la invariante. En este caso, se cambia N por un índice i.

$$\text{key} \notin \text{arr}[0..\text{i}). \qquad (13 :: \text{invariant})$$

El índice, al ser una búsqueda lineal, se inicializa en el primer elemento del *array* y aumenta de a una unidad en cada paso.

$$\text{i} := 0 \qquad (15 :: \text{init})$$

$$\text{i} := \text{i} + 1. \qquad (21 :: \text{step})$$

De la partición con elementos a recorrer, se deriva la variante que determina la cantidad de elementos que faltan por recorrer.

$$\text{N - i}. \qquad (22 :: \text{variant})$$

Para que el algoritmo termine, pueden presentarse dos escenarios. Se encuentra el elemento buscado o se termina de recorrer todo el *array* sin haberlo encontrado. Entonces, se puede derivar la condición de terminación de la siguiente manera:

$$\text{i} = \text{N} \vee \text{arr}[\text{i}] = \text{key}. \qquad (18 :: \text{until})$$

Para el caso afirmativo, al encontrar el elemento buscado, se cumple la post-condición automáticamente, ya que se prueba utilizando la condición de terminación.

Para el caso negativo, al terminar el recorrido en el *array* sin haber encontrado el elemento buscado, la invariante esencial que utiliza el índice i para determinar la partición, avanza hasta que $i = N$.

En este caso, como resultado se devuelve N, pero como se prefiere devolver -1 en el caso negativo, se realiza un ajuste derivando automáticamente en la última post-condición.

$$\text{if i} = \text{N} \Rightarrow \text{-1}. \qquad (24 :: \text{adjust})$$

6.1.2 Implementación en Dafny

```
1  method LinearSearch (arr: array<int>, key: int) returns (i: int)
2    ensures 0 <= i < arr.Length ==> arr[i] == key
3    ensures i == -1 ==> key !in arr[..]
4  {
5    i := 0;
6
7    while (i != arr.Length && arr[i] != key)
8      invariant 0 <= i <= arr.Length
9      invariant forall k :: 0 <= k < i ==> arr[k] != key
10     decreases arr.Length - i
11   {
12     i := i + 1;
13   }
14
15   if (i == arr.Length) {
16     i := -1;
17   }
18 }
```

6.1.3 Implementación en KeY

```
1  /*@ public normal_behavior
2    @ ensures (0 <= \result && \result < arr.length ==> arr[\result] == key);
3    @ ensures (\result == -1 ==> !(\exists int k;
4    @                                  0 <= k && k < arr.length;
5    @                                  arr[k] == key));
6    @ assignable \strictly_nothing;
7    @*/
8  public int linearSearch (int[] arr, int key) {
9    int i = 0;
10
11   /*@ loop_invariant 0 <= i && i <= arr.length;
12     @ loop_invariant (\forall int k;
13     @                         0 <= k && k < i;
14     @                         arr[k] != key);
15     @ assignable \strictly_nothing;
16     @ decreases arr.length - i;
17     @*/
18   while (i != arr.length && arr[i] != key) {
19     i = i + 1;
20   }
21   return i == arr.length ? -1 : i;
22 }
```

6.2 Máximo elemento de un array

Busca el máximo elemento en un *array*, realizando una búsqueda lineal de izquierda a derecha.

6.2.1 Metodología

```
1  MaxSearch (arr: array<int>): (result: int)
2    require
3      N > 0    -- Requerimiento de acotamiento
4    ensure
5      result in [0..N)            -- Propiedad de acotamiento
6      arr[result] >= arr[0..N)    -- Propiedad esencial
7    local
8      N: int
9      i: int
10   do
11     invariant
12       i in [0..N]                 -- Invariante de acotamiento
13       result in [0..N)            -- Invariante de acotamiento
14       arr[result] >= arr[0..i)    -- Invariante esencial
15     init
16       N := arr.length
17       result := 0
18       i := 0
19     until
20       i = N
21     step
22       if arr[i] > arr[result] then result := i
23       i := i + 1
24     variant
25       N - i
26     end
27   end
```

El algoritmo recibe por parámetro un *array* con al menos un elemento.

$$N > 0. \qquad (3 :: \text{require})$$

Se determinan dos post-condiciones:

1. Propiedad de acotamiento del resultado.

$$\text{result} \in [0..N). \qquad (5 :: \text{ensure})$$

2. Propiedad que garantiza que en la posición *result* del array, está el valor máximo.

$$\text{arr[result]} \geq \text{arr}[0..N). \qquad (6 :: \text{ensure})$$

Al ser una búsqueda lineal, existe una partición entre elementos ya recorridos y otra partición con los elementos por recorrer.

Para la partición con elementos ya recorridos, se utiliza el método "sustituir constante por variable" en la invariante, cambiando N por un índice i.

$$\text{arr[result]} \geq \text{arr[0..i)}. \qquad (14 :: \text{invariant})$$

El índice utilizado, se inicializa en el primer elemento del *array* y aumenta de a una unidad en cada paso.

$$\text{i} := 0 \qquad (18 :: \text{init})$$

$$\text{i} := \text{i} + 1. \qquad (23 :: \text{step})$$

Se utiliza un puntero *result*, que debe estar acotado entre las posiciones del *array*, porque apunta a la posición del elemento mayor.

$$\text{result} \in \text{[0..N)}. \qquad (13 :: \text{invariant})$$

Se inicializa en cero como caso base, ya que si existe un único elemento, ese será el mayor.

$$\text{result} := 0. \qquad (17 :: \text{init})$$

En cada paso, se controla si el valor que toma el *array* en la posición del índice i, es mayor al elemento en la posición *result*. En caso de ser así, se actualiza la posición de *result* a la de dicho elemento.

$$\text{if arr[i]} > \text{arr[result]} \Rightarrow \text{result} := \text{i}. \qquad (22 :: \text{step})$$

De la partición con elementos por recorrer, se deriva la variante que determina la cantidad de elementos que faltan por recorrer.

$$\text{N - i}. \qquad (25 :: \text{variant})$$

Para que el algoritmo finalice, el índice i debe haber terminado de recorrer todo el *array*.

$$\text{i} = \text{N}. \qquad (20 :: \text{until})$$

Al finalizar el recorrido, la invariante esencial que utiliza el índice i para determinar la partición, avanza hasta que $i = N$ derivando a la post-condición principal.

$$\text{arr[result]} \geq \text{arr[0..N)}. \qquad (6 :: \text{ensure})$$

6.2.2 Implementación en Dafny

```
method MaxSearch(arr: array<int>) returns (result: int)
  requires arr.Length > 0
  ensures 0 <= result < arr.Length
  ensures forall k :: 0 <= k < arr.Length ==> arr[result] >= arr[k]
{
  var N: nat := arr.Length;
  result := 0;
  var i: nat := 0;

  while (i != N)
    invariant 0 <= i <= N
    invariant 0 <= result < N
    invariant forall k :: 0 <= k < i ==> arr[result] >= arr[k]
    decreases N - i;
  {
    if (arr[i] > arr[result])
    {
      result := i;
    }
    i := i + 1;
  }
}
```

6.2.3 Implementación en KeY

```
/*@ public normal_behavior
  @  requires arr.length > 0;
  @  ensures 0 <= result < arr.length;
  @  ensures (\forall int k;
  @                0 <= k < arr.length;
  @                arr[result] >= arr[k]);
  @  assignable \nothing;
  @*/
public int maxSearch(int[] arr) {
  int N = arr.length;
  int result = 0;
  int i = 0;

  /*@  loop_invariant 0 <= i && i <= N;
    @  loop_invariant 0 <= result && result < N;
    @  loop_invariant (\forall int k;
    @                        0 <= k && k < i;
    @                        arr[result] >= arr[k]);
    @  decreases N - i;
    @  assignable \nothing;
    @*/
  while (i != N) {
    if (arr[i] > arr[result]) {
      result = i;
    }
    i = i + 1;
  }
  return result;
}
```

6.3 Unicidad de elementos en un array

Es un algoritmo propuesto por el libro *Cracking the Code Interview* [20], que evalúa en el *array* que no haya ningún carácter repetido, esto es, que cada carácter sea único.

6.3.1 Metodología

```
1  IsUnique (s: array<char>): (b: bool)
2    require
3      true    -- No tiene requerimientos
4    ensure
5      b <==> forall a, b when a in [0..N) and b in [0..N) and b != a then s[a] != s[b]
6    local                                                    -- Propiedad esencial
7      N: int
8      i: int
9      j: int
10   do
11     invariant
12       i in [0..N]      -- Invariante de acotamiento
13       b <==> forall a, b when a in [0..i) and b in [0..i) and a != b then s[a] != s[b]
14     init                                                   -- Invariante esencial
15       N := s.length
16       b := true
17       i := 0
18     until
19       not b or i = N
20     step
21       invariant
22         j in [0..i]                 -- Invariante de acotamiento
23         s[i] != arr[0..j)           -- Invariante esencial
24       init
25         j := 0
26       until
27         j = i or s[j] = s[i]
28       step
29         j := j + 1;
30       variant
31         i - j
32       end
33       b := j = i
34       i := i + 1
35     variant
36       N - i
37     end
38   end
```

El algoritmo recibe por parámetro un *array* de caracteres y se determina como post-condición la unicidad de sus elementos:

$$b \iff (\forall\ a, b \in [0..i) \land a \neq b \Rightarrow s[a] \neq s[b]).\ (5 :: \text{ensure})$$

Se aplica el método de "seleccionar condiciones" para probar las post-condiciones (si y sólo si) de tal forma que la post-condición negativa se prueba utilizando la condición de terminación y la post-condición positiva utilizando una invariante.

Se pueden realizar dos particiones, una para los elementos ya recorridos y otra para los elementos por recorrer.

Se prueba la post-condición positiva, utilizando una invariante con el método de "sustituir una constante por variable" (N por i).

$$\text{b} \iff (\forall\ \text{a, b} \in [0..\text{i}) \land \text{a} \neq \text{b} \Rightarrow \text{s[a]} \neq \text{s[b]}). \qquad (13 :: \text{invariant})$$

Se inicializa el índice i en el primer elemento del *array* y aumenta de a una unidad en cada paso.

$$\text{i} := 0 \qquad (17 :: \text{init})$$

$$\text{i} := \text{i} + 1. \qquad (34 :: \text{step})$$

Para determinar si se mantiene la propiedad de unicidad de los elementos, se utiliza una variable b, inicializada en *true*.

$$\text{b} := \text{true}. \qquad (16 :: \text{init})$$

De la partición con elementos a recorrer, se deriva la variante que determina la cantidad de elementos que restan por recorrer.

$$\text{N - i}. \qquad (36 :: \text{variant})$$

Para que el algoritmo termine, existen dos casos posibles. Si se encontró un elemento repetido ó si se terminó de recorrer el *array*:

$$\neg\ \text{b} \lor \text{i} = \text{N}. \qquad (19 :: \text{until})$$

En caso de existir al menos un elemento repetido, se retorna b como *false* y se cumple con la post-condición.

En caso de no existir elementos repetidos, la invariante esencial que utiliza al índice i para determinar la partición, avanza hasta que $i = N$, derivando a la post-condición principal.

$$\text{b} \iff (\forall\ \text{a, b} \in [0..\text{i}) \land \text{a} \neq \text{b} \Rightarrow \text{s[a]} \neq \text{s[b]}) \ (5 :: \text{ensure})$$

El problema se reduce en probar cómo se comparan los elementos del *array*, de tal forma que se pueda comparar cada elemento con todos los demás.

Esto se debe evaluar dentro del paso principal y requiere tomar como alcance desde [0..i], con la finalidad de que se pueda mantener la invariante principal comentada anteriormente. Se utiliza un nuevo índice *j* que está acotado en dicho rango.

$$\text{j} \in [0..\text{i}]. \qquad (22 :: \text{invariant})$$

En este punto, se puede volver a aplicar el método de "sustituir constante por variable", generando dos nuevas particiones. Una tendrá los elementos ya recorridos por el índice *j*, en el rango [0..j), que tienen diferente valor al elemento en la posición *i*:

$$\text{arr}[0..\text{j}) \mathrel{!}= \text{s}[\text{i}]. \qquad (23 :: \text{invariant})$$

La otra partición tendrá los elementos entre [j..i), que son los elementos que faltan por recorrer.

$$\text{i - j}. \qquad (31 :: \text{variant})$$

Se inicializa *j* en la posición del primer elemento del *array*, para poder comparar cada uno de los elementos hasta la posición *i*.

$$\text{j} := 0 \qquad (25 :: \text{init})$$

$$\text{j} := \text{j} + 1. \qquad (29 :: \text{step})$$

Existen dos condiciones de terminación:

$$\text{j} = \text{i} \vee \text{s}[\text{j}] = \text{s}[\text{i}]. \qquad (27 :: \text{until})$$

1. Se encuentra algún elemento repetido

En caso de ser así, se terminan los pasos internos y en el paso principal se asigna a *b* como *false*.

$$\text{b} := \text{j} = \text{i}. \qquad (33 :: \text{step})$$

2. El índice *j* recorre todos los elementos hasta la posición de *i*.

En este caso, cuando $j = i$, la invariante esencial que utiliza a *j*, verifica que no existen elementos repetidos en la partición recorrida de [0..i).

6.3.2 Implementación en Dafny

```
method IsUnique(s: array<char>) returns (b: bool)
  ensures b <==> forall a, b :: 0 <= a < s.Length
                             && 0 <= b < s.Length
                             && a != b ==> s[a] != s[b]
{
  var N: nat := s.Length;
  b := true;
  var i: nat := 0;

  while (b && i != N)
    invariant 0 <= i <= N;
    invariant b <==> forall a, b :: 0 <= a < i
                                 && 0 <= b < i
                                 && a != b ==> s[a] != s[b]
    decreases N - i;
  {
    var j: nat := 0;

    while (j != i && s[j] != s[i])
      invariant 0 <= j <= i;
      invariant forall k :: 0 <= k < j ==> s[k] != s[i]
      decreases i - j;
    {
      j := j + 1;
    }
    b := j == i;
    i := i + 1;
  }
}
```

6.3.3 Implementación en KeY

```
/*@ public normal_behavior
  @ ensures (\result <==> (\forall int a, b;
  @                                0 <= a && a < s.length &&
  @                                0 <= b && b < s.length &&
  @                                a != b; s[a] != s[b]));
  @*/
public boolean isUnique(char[] s) {
  int N = s.length;
  boolean b = true;
  int i = 0;

  /*@ loop_invariant 0 <= i && i <= N;
    @ loop_invariant (b <==> (\forall int a, b;
    @                                0 <= a && a < i &&
    @                                0 <= b && b < i &&
    @                                a != b; s[a] != s[b]));
    @ assignable \nothing;
    @ decreases N - i;
    @*/
  while (b && i != N) {
    int j = 0;

    /*@ loop_invariant 0 <= j && j <= i;
      @ loop_invariant (\forall int k; 0 <= k < j; s[k] != s[i]);
      @ assignable \nothing;
      @ decreases i - j;
      @*/
```

```
28      while (j != i && s[j] != s[i]) {
29        j = j + 1;
30      }
31      b = j == i;
32      i = i + 1;
33    }
34    return b;
35  }
```

6.4 Búsqueda binaria

Es un algoritmo eficiente de búsqueda, que requiere una lista ordenada de elementos. Para realizar la búsqueda utiliza dos índices l y r, inicializados al comienzo y al final.

Luego se realiza una serie de iteraciones sobre el *array*, dividiendo el intervalo a la mitad repetitivamente en cada paso, según el valor del elemento buscado y el valor del elemento que se encuentra en la mitad del intervalo.

Si el valor buscado es mayor, se toma la primer mitad del intervalo, de lo contrario se toma la segunda mitad.

Al encontrar una coincidencia, se devuelve el índice del elemento. En caso de terminar el recorrido sin haber encontrado ninguna coincidencia, se devuelve -1.

6.4.1 Metodología

```
1  BinarySearch (arr: array<int>, key: int): (m: int)
2    require
3      sorted_increasing(arr)    -- Requerimiento de ordenamiento
4    ensure
5      m = -1 then key not in arr[..]    -- Propiedad caso negativo
6      0 <= m < N then key = arr[m]      -- Propiedad caso positivo
7    local
8      N: int
9      l: int
10     r: int
11   do
12     invariant
13       0 <= l <= r <= N        -- Invariante de acotamiento
14       m = (l + r) / 2
15       key not in arr[0..l)    -- Invariante esencial
16       key not in arr[r..N)    -- Invariante esencial
17     init
18       N := arr.length
19       l := 0
20       r := N
21       m := (l + r) / 2
22     until
23       l = r or arr[m] = key
24     step
25       if key < arr[m]
26         then r := m
27         else l := m + 1
28       m := (l + r) / 2
29     variant
30       r - l
31     adjust
32       if l = r then m := -1
33     end
34   end
```

Se recibe por parámetro un *array* de elementos ordenado ascendentemente y un elemento *key* a buscar.

$$\text{sorted\_increasing(arr)} \qquad (3 :: \text{require})$$

El algoritmo tiene dos post-condiciones:

1. Propiedad que se cumple cuando no se encuentra el elemento buscado:

$$\text{m} = \text{-1} \Rightarrow \text{key} \notin \text{arr[..]}. \qquad (5 :: \text{ensure})$$

2. Propiedad que se cumple cuando se encuentra el elemento buscado:

$$0 \leq \text{m} < \text{N} \Rightarrow \text{arr[m]} = \text{key}. \qquad (6 :: \text{ensure})$$

Se aplica el método de "seleccionar condiciones" para probar las post-condiciones. De tal forma, que la post-condición positiva se prueba utilizando la condición de terminación y la post-condición negativa utilizando una invariante.

Se determina que existe una partición que tiene los elementos descartados y otra partición que tiene los elementos por recorrer.

De la partición de elementos descartados, se derivan dos invariantes que mantienen la propiedad de que cada elemento ya descartado, no puede ser el elemento buscado. Esto se realiza con el método de "sustituir constante por variable" en las invariantes esenciales.

Para una invariante, se selecciona el rango [0..l) y para la otra el rango [r..N), formando las siguientes invariantes esenciales:

$$\text{key} \notin \text{arr[0..l)} \qquad (15 :: \text{invariant})$$

$$\text{key} \notin \text{arr[r..N)}. \qquad (16 :: \text{invariant})$$

Ambos punteros l y r están acotados dentro de las posiciones del *array*:

$$0 \leq \text{l} \leq \text{r} \leq \text{N}. \qquad (13 :: \text{invariant})$$

El puntero l se inicializa en cero y el puntero r se inicializa en N.

$$\text{l} := 0 \qquad (21 :: \text{init})$$

$$r := N. \qquad (22 :: \text{init})$$

De la partición con elementos por recorrer, se deriva la variante que determina la cantidad de elementos que faltan por recorrer.

$$r - l. \qquad (30 :: \text{variant})$$

Para que el algoritmo termine, pueden suceder dos casos. Se encuentra el elemento buscado o no existen más elementos para evaluar. Entonces, se puede derivar la condición de terminación:

$$l = r \vee arr[m] = key. \qquad (23 :: \text{until})$$

Para el caso afirmativo, al encontrar el elemento buscado, se cumple la post-condición automáticamente, ya que se prueba utilizando la condición de terminación.

Para el caso negativo, implica que no existen más elementos en la partición a recorrer y todos los elementos del *array* pertenecen a la partición descartada. Por esta razón, al ser $l = r$, la unión entre el conjunto arr[0..l) + arr[r..N) = arr[0..N] = arr[..], como aparece en la post-condición del caso negativo.

Para este caso, se realiza un ajuste para no devolver el punto medio m, y se devuelve -1:

$$\text{if } l = r \Rightarrow m := -1. \qquad (34 :: \text{adjust})$$

Resta por explicar cómo se realizan los pasos para ir reduciendo a la mitad, sucesivamente, la partición a recorrer.

En cada paso, se evalúa si el elemento a buscar *key*, es menor al punto medio entre el puntero l y r.

$$\text{if } key < arr[m]. \qquad (25 :: \text{step})$$

En caso de que *key* sea menor, se agregan los elementos arr(m..r] a la partición con elementos descartados.

$$r := m. \qquad (28 :: \text{step})$$

En caso de que *key* sea mayor o igual, se agregan los elementos arr[l..m] a la partición con elementos descartados.

Luego de cualquiera de los dos casos, se debe calcular nuevamente el punto medio m.

$$m = (l + r) \;/\; 2. \qquad (28 :: \text{step})$$

6.4.2 Implementación en Dafny

```
method BinarySearch(arr: array<int>, key: int) returns (m: int)
  requires forall a, b :: 0 <= a <= b < arr.Length ==> arr[a] <= arr[b]
  ensures 0 <= m < arr.Length ==> arr[m] == key
  ensures m == -1 ==> forall k :: 0 <= k < arr.Length ==> arr[k] != key
{
  var N: nat := arr.Length;
  var l: int := 0;
  var r: int := N;
  m := (l + r) / 2;

  while (l != r && key != arr[m])
    invariant 0 <= l <= r <= N
    invariant m == (l + r) / 2
    invariant forall k :: 0 <= k < l ==> arr[k] != key
    invariant forall k :: r <= k < N ==> arr[k] != key
    decreases r - l
  {
    if (key < arr[m]) {
      r := m;
    } else {
      l := m + 1;
    }
    m := (l + r) / 2;
  }

  if (l == r) {
    m := -1;
  }
}
```

6.4.3 Implementación en KeY

```
/*@ public normal_behavior
  @ requires (\forall int a, b;
  @                   0 <= a && a <= b && b < arr.length;
  @                   arr[a] <= arr[b]);
  @ ensures (0 <= \result && \result < arr.length ==> arr[\result] == key);
  @ ensures (\result == -1 ==> (\forall int k;
  @                                   0 <= k && k < arr.length;
  @                                   arr[k] != key));
  @*/
public int binarySearch(int[] arr, int key) {
  int N = arr.length;
  int l = 0;
  int r = N;
  int m = (l + r) / 2;

  /*@ loop_invariant 0 <= l && l <= r && r <= N;
    @ loop_invariant m == (l + r) / 2;
    @ loop_invariant (\forall int k;
    @                         0 <= k < l;
    @                         arr[k] != key);
    @ loop_invariant (\forall int k;
```

```
22      @                          r <= k < N;
23      @                          arr[k] != key);
24      @ assignable \strictly_nothing;
25      @ decreases r - l;
26      @*/
27    while (l != r && key != arr[m]) {
28      if (key < arr[m]) r = m;
29      else l = m + 1;
30      m = (l + r) / 2;
31    }
32    return (l == r ? -1 : m);
33  }
```

7 Algoritmos de ordenamiento

7.1 Insertion Sort

Algoritmo de ordenamiento que recibe un *array* de elementos por parámetro y los ordena de a un elemento a la vez, desde izquierda a derecha.

7.1.1 Metodología

```
1  InsertionSort (arr: array<int>)
2    require
3      N > 0    -- Requerimiento de acotamiento
4    ensure
5      sorted_increasing(arr[..])     -- Propiedad de ordenamiento
6      perm(arr[..], old(arr[..]))    -- Propiedad de integridad
7    local
8      N: int
9      i: int
10     j: int
11   do
12     invariant
13       i in [1..N]                    -- Invariante de acotamiento
14       sorted_increasing(arr[0..i])   -- Invariante esencial
15       perm(arr[..], old(arr[..]))    -- Invariante esencial
16     init
17       N := arr.length
18       i := 1
19     until
20       i = N
21     step
22       invariant
23         j in [0..i]    -- Invariante de acotamiento
24         forall a, b in 0 <= a < b <= i && b != j then arr[a] < arr[b]    -- Invariante
25         prem(arr[..], old(arr[..]))    -- Invariante esencial               esencial
26       init
27         j := i
28       until
29         j = 0 or arr[j - 1] <= arr[j]
30       step
31         swap(j, j - 1)
32         j := j - 1
33       variant
34         j
35       end
36       i := i + 1
```

```
37      variant
38        N - i
39      end
40    end
```

Se recibe por parámetro un *array* a ordenar y tiene dos post-condiciones:

1. Propiedad de ordenamiento

$$\text{sorted\_increasing(arr[..])}. \qquad (5 :: \text{ensure})$$

2. Propiedad de integridad

$$\text{perm(arr[..], old(arr[..]))}. \qquad (6 :: \text{ensure})$$

Propiedad de ordenamiento

Se aplica el método de elegir y probar la post-condición de ordenamiento utilizando una invariante.

Se determina que existe una partición que tiene los elementos ya ordenados y otra con los elementos a ordenar.

De la partición con elementos ordenados, se deriva una invariante utilizando el método de "sustituir constante por variable". En este caso, se sustituye N por una nueva variable i, de tal forma que todos los elementos que estén a la izquierda del índice i pertenezcan a la partición ordenada (arr[0..i)).

$$\text{sorted\_increasing(arr[0..i])}. \qquad (14 :: \text{invariant})$$

Por otra parte, los elementos que están a la derecha de i pertenecen a la partición por ordenar (arr[i..N)). De aquí se puede derivar la variante siguiente:

$$\text{N - i}. \qquad (38 :: \text{variant})$$

Además, en base a las particiones, se puede observar que el índice i se inicializa en 1 y a lo sumo puede valer N.

$$\text{i} := 1 \qquad (18 :: \text{init})$$

$$\text{i} \in [1..\text{N}]. \qquad (13 :: \text{invariant})$$

Para que el algoritmo termine, es necesario que el índice i recorra todos los elementos del *array*, de tal forma que $i = N$. En ese caso, se deriva automáticamente a la post-condición de la propiedad de ordenamiento.

$$\text{sorted\_increasing(arr[..])}. \qquad (5 :: \text{ensure})$$

Resta por explicar los pasos que debe realizar el algoritmo para poder ordenar cada elemento e incrementar sucesivamente de a una unidad, la partición ordenada.

Se define un nuevo alcance que contiene a los elementos de la partición ordenada y a un elemento extra a ordenar (arr[0..i) + arr[i]). Además, se necesita un índice nuevo para recorrer los elementos desde derecha a izquierda dentro de esa partición.

$$\text{j} := \text{i} \qquad (27 :: \text{init})$$

$$\text{j} \in [0..\text{i}]. \qquad (23 :: \text{invariant})$$

Se define la invariante que mantiene la condición de que el *array* está ordenado hasta la posición i, para cada par de elementos, excepto para los que el índice del segundo elemento sea igual a j.

$$\forall\ \text{a, b} \in 0 \leq \text{a} < \text{b} \leq \text{i} \wedge \text{b} \neq \text{j} \Rightarrow \text{arr[a]} < \text{arr[b]}. \qquad (24 :: \text{invariant})$$

En cada paso, se realiza un *swap* entre j y el elemento que está a su izquierda y se decrementa j en una unidad.

$$\text{swap(j, j - 1)} \qquad (31 :: \text{step})$$

$$\text{j} := \text{j - 1}. \qquad (32 :: \text{step})$$

La variante en este caso, sería simplemente el índice j:

$$\text{j}. \qquad (34 :: \text{variant})$$

Se puede probar que el ordenamiento del nuevo elemento termina si se cumple una de las siguientes condiciones:

1. Se llega a la posición 0, indicando que el elemento que estaba en la posición inicial i era el que tenía menor valor con respecto a los que estaban en la partición ordenada.

2. El elemento que está a la izquierda de la posición de j tiene menor valor que el elemento en la posición j, lo que implica que está en el lugar correcto de la partición ordenada.

$$j = 0 \vee arr[j - 1] \leq arr[j]. \qquad (29 :: until)$$

Propiedad de integridad

Para demostrar que los elementos finales sean una permutación de los elementos iniciales, basta con observar que la única operación que se realiza sobre el *array*, son *swaps* entre elementos consecutivos, haciendo que se mantenga la propiedad de integridad en cada uno de los pasos.

$$perm(arr[..], old(arr[..])) \qquad (25 :: invariant)$$

$$perm(arr[..], old(arr[..])). \qquad (15 :: invariant)$$

Automáticamente se puede derivar a la verificación de la propiedad de integridad:

$$perm(arr[..], old(arr[..])). \qquad (6 :: ensure)$$

7.1.2 Implementación en Dafny

```
1  method InsertionSort(arr: array<int>)
2    requires arr.Length > 0;
3    ensures forall a, b :: 0 <= a <= b < arr.Length ==> arr[a] <= arr[b]
4    ensures multiset(arr[..]) == multiset(old(arr[..]))
5    modifies arr
6  {
7    var N: nat := arr.Length;
8    var i: nat := 1;
9
10   while (i != N)
11     invariant 1 <= i <= N;
12     invariant forall a, b :: 0 <= a <= b < N && b < i ==> arr[a] <= arr[b]
13     invariant multiset(arr[..]) == multiset(old(arr[..]))
14     decreases N - i;
15     modifies arr;
16   {
17     var j: nat := i;
18
19     while (j != 0 && arr[j - 1] > arr[j])
20       invariant 0 <= j <= i;
21       invariant forall a, b :: 0 <= a < b <= i && b != j ==> arr[a] <= arr[b]
22       invariant multiset(arr[..]) == multiset(old(arr[..]))
23       decreases j;
24       modifies arr;
25     {
26       var temp: int := arr[j];
27       arr[j] := arr[j-1];
28       arr[j-1] := temp;
29
30       j := j - 1;
31     }
```

```
32      i := i + 1;
33    }
34  }
```

7.1.3 Implementación en KeY

```
1   /*@ public normal_behavior
2     @  requires arr.length > 0;
3     @  ensures (\forall int a, b;
4     @                  0 <= a && a <= b && b < arr.length;
5     @                  arr[a] <= arr[b]);
6     @  ensures \dl_seqPerm(\dl_array2seq(arr), \old(\dl_array2seq(arr)));
7     @  assignable arr[*];
8     @*/
9   public void insertionSort(int[] arr) {
10    int N = arr.length;
11    int i = 1;
12
13    /*@  loop_invariant 1 <= i && i <= N;
14      @  loop_invariant (\forall int a, b;
15      @                          0 <= a && a <= b && b < N && b < i;
16      @                          arr[a] <= arr[b]);
17      @  loop_invariant \dl_seqPerm(\dl_array2seq(arr), \old(\dl_array2seq(arr)));
18      @  assignable arr[*];
19      @  decreases N - i;
20      @*/
21    while (i != N) {
22      int j = i;
23
24      /*@  loop_invariant 0 <= j && j <= i;
25        @  loop_invariant (\forall int a, b;
26        @                           0 <= a && a < b && b <= i && b != j;
27        @                           arr[a] <= arr[b]);
28        @  loop_invariant \dl_seqPerm(\dl_array2seq(arr), \old(\dl_array2seq(arr)));
29        @  assignable arr[*];
30        @  decreases j;
31        @*/
32      while (j != 0 && arr[j - 1] > arr[j]) {
33        int temp = arr[j];
34        arr[j] = arr[j-1];
35        arr[j-1] = temp;
36
37        j = j - 1;
38      }
39      i = i + 1;
40    }
41  }
```

Una vez que se ejecuta el verificador del *IDE KeY v2.10.0*, restan dos *goals* sin cerrar. Ambos *goals* tienen relación con la demostración de la propiedad de integridad. Por esta razón, se utilizan ciertas reglas manualmente para ayudar al verificador a probar dicha post-condición.

Sobre la pestaña *Sequent*, se debe realizar exactamente el mismo procedimiento para ambos *goals* pendientes:

1. Hacer *click* sobre el símbolo implica ($\Rightarrow$).

2. Hacer *click* sobre *Strategy Macros* > *Auto Pilot* > *Finish Symbolic Execution*.

3. Hacer *click* en el *sequent seqPerm*.

4. Hacer *click* sobre la regla *seqPermFromSwap*.

5. Hacer *click* sobre el símbolo implica ($\Rightarrow$).

6. Hacer *click* sobre *Strategy Macros* > *Close Probable Goals Below*.

Una vez realizado estos pasos, queda demostrada correctamente la verificación formal de las post-condiciones del algoritmo.

7.2 Bubble Sort

Algoritmo de ordenamiento que recibe un *array* de elementos por parámetro y utiliza como mecanismo la comparación de pares consecutivos de elementos para ordenar. Si el elemento actual es mayor al elemento siguiente, se intercambian posiciones. Los elementos se van ordenando de derecha a izquierda.

7.2.1 Metodología

```
1  BubbleSort (arr: array<int>)
2    require
3      N > 0    -- Requerimiento de acotamiento
4    ensure
5      sorted_increasing(arr[..])     -- Propiedad de ordenamiento
6      perm(arr[..], old(arr[..]))    -- Propiedad de integridad
7    local
8      N: int
9      i: int
10     j: int
11   do
12     invariant
13       i in [0..N)                      -- Invariante de acotamiento
14       sorted_increasing(arr[i..N))     -- Invariante esencial
15       arr[0..i] <= arr(i..N)           -- Invariante esencial
16       perm(arr[..], old(arr[..]))      -- Invariante esencial
17     init
18       N := arr.length
19       i := N - 1
20     until
21       i = 0
22     step
23       invariant
24         j in [0..i]                      -- Invariante de acotamiento
25         sorted_increasing(arr[i..N))     -- Invariante esencial
26         arr[0..i] <= arr(i..N)           -- Invariante esencial
27         arr[0..j] <= arr[j]              -- Invariante esencial
28         prem(arr[..], old(arr[..]))      -- Invariante esencial
29       init
30         j := 0
31       until
32         j = i
33       step
34         if arr[j] > arr[j + 1] then swap(j, j + 1)
35         j := j + 1
36       variant
37         i - j
38       end
39       i := i - 1
40     variant
41       i
42     end
43   end
```

Se recibe por parámetro un *array* a ordenar y tiene dos post-condiciones:

1. Propiedad de ordenamiento

sorted_increasing(arr[..]). (5 :: ensure)

2. Propiedad de integridad

perm(arr[..], old(arr[..])). (6 :: ensure)

Propiedad de ordenamiento

Se aplica el método de elegir y probar la post-condición de ordenamiento utilizando una invariante.

Se determina que existe una partición que tiene los elementos ya ordenados y otra partición que tiene los elementos a ordenar.

De la partición ordenada, se deriva una invariante utilizando el método de "sustituir constante por variable". En este caso, se sustituye *0* por una nueva variable *i*, de tal forma que todos los elementos que estén a la derecha del índice *i*, pertenezcan a la partición ordenada (arr[i..N)).

sorted_increasing(arr[i..N)). (14 :: invariant)

Por otro lado, la partición que está a la izquierda del índice *i*, que contiene a los elementos a ordenar, se puede derivar la variante:

i. (41 :: variant)

Además, como *Bubble Sort* va ordenando de mayor a menor, desde derecha a izquierda, existe una particularidad con la partición a ordenar, que sus elementos tienen menor valor que los elementos de la partición ordenada. Esto se deriva a la siguiente invariante:

arr[0..i] $\leq$ arr(i..N). (15 :: invariant)

De la unión de ambas particiones, se puede deducir que existe una invariante de acotamiento del índice *i*:

i $\in$ [0..N). (13 :: invariant)

El índice *i* es inicializado en la última posición del *array* y además se va decrementando de a una unidad en cada paso.

i := N - 1 (19 :: init)

i := i - 1. (39 :: step)

Como condición de terminación, se debe recorrer todo el *array*.

$$\mathrm{i} = 0. \qquad (21 :: \text{until})$$

Cuando se termina de recorrer todo el *array*, el índice i vale 0, por lo tanto, la invariante esencial que fue determinada por la partición ordenada, deriva automáticamente a la post-condición de la propiedad de ordenamiento.

Resta por explicar los pasos que debe realizar el algoritmo, para poder ordenar los elementos desde la partición a ordenar, hacia la partición ordenada:

$$0 \leq \mathrm{j} \leq \mathrm{i}. \qquad (21 :: \text{invariant})$$

Los elementos que están a la derecha del índice i seguirán estando ordenados, ya que no se realizan operaciones sobre dichos elementos.

$$\text{sorted\_increasing}(\mathrm{arr[i..N]}). \qquad (14 :: \text{invariant})$$

Por el mismo motivo, se mantiene la invariante que detalla que los elementos que estén en la partición a ordenar, son menores a los elementos que estén en la partición ordenada:

$$\mathrm{arr[0..i]} \leq \mathrm{arr(i..N)}. \qquad (15 :: \text{invariant})$$

Se realiza una nueva partición sobre la partición de elementos a ordenar, utilizando el método de "sustituir constante por variable", sustituyendo i por una nueva variable j, de tal forma que todos los elementos que están a la izquierda del índice j, son menores o iguales al elemento en la posición j.

$$\mathrm{arr[0..j]} \leq \mathrm{arr[j]}. \qquad (27 :: \text{invariant})$$

El índice j es inicializado en cero y recorrerá de izquierda a derecha la nueva partición.

$$\mathrm{j} := 0 \qquad (30 :: \text{init})$$

$$\mathrm{j} \in [0..\mathrm{i}]. \qquad (24 :: \text{invariant})$$

Por otro lado, la partición determinada por los elementos arr(j..i], serán parte de la variante:

$$\text{i - j.} \qquad (37 :: \text{variant})$$

En cada paso se compara el elemento en la posición j con el elemento a su derecha: en caso de que el elemento en la posición j sea mayor entonces se realiza un *swap* y se incrementa j en una unidad. Si no, sólo se incrementa j.

$$\text{if arr[j] > arr[j + 1]} \Rightarrow \text{swap(j, j + 1)} \qquad (34 :: \text{step})$$

$$\text{j := j + 1.} \qquad (35 :: \text{step})$$

Los pasos terminan cuando el índice j, recorre todos los elementos dentro de la partición.

$$\text{j = i.} \qquad (32 :: \text{until})$$

Frente a eso, como $j = i$, todos los elementos que se encuentren a la izquierda de la posición j, serán menores al valor arr[j]. Esto hace que la posición arr[j] y arr[i] cumplan con el criterio para pertenecer en la partición ordenada, derivando en la invariante de la partición ordenada.

Propiedad de integridad

Para demostrar que los elementos finales sean una permutación de los elementos iniciales, basta con observar que la única operación que se realiza sobre el *array*, son *swaps* entre elementos consecutivos, haciendo que se mantenga la propiedad de integridad en cada uno de los pasos.

$$\text{perm(arr[..], old(arr[..]))} \qquad (28 :: \text{invariant})$$

$$\text{perm(arr[..], old(arr[..])).} \qquad (16 :: \text{invariant})$$

Automáticamente se puede derivar a la verificación de la propiedad de integridad.

$$\text{perm(arr[..], old(arr[..])).} \qquad (6 :: \text{ensure})$$

7.2.2 Implementación en Dafny

```
method BubbleSort(arr: array<int>)
  requires arr.Length > 0;
  ensures forall a, b :: 0 <= a <= b < arr.Length ==> arr[a] <= arr[b]
  ensures multiset(arr[..]) == multiset(old(arr[..]))
  modifies arr;
{
  var N: nat := arr.Length;
  var i: nat := N - 1;

  while(i != 0)
    invariant 0 <= i < N
    invariant forall a, b :: i <= a <= b < N ==> arr[a] <= arr[b]
    invariant forall a, b :: 0 <= a <= i < b < N ==> arr[a] <= arr[b]
    invariant multiset(arr[..]) == multiset(old(arr[..]))
    decreases i;
    modifies arr;
  {
    var j: int := 0;

    while(j != i)
      invariant 0 <= j <= i;
      invariant forall a, b :: i <= a <= b < N ==> arr[a] <= arr[b]
      invariant forall a, b :: 0 <= a <= i < b < N ==> arr[a] <= arr[b]
      invariant forall k :: 0 <= k <= j ==> arr[k] <= arr[j]
      invariant multiset(arr[..]) == multiset(old(arr[..]))
      decreases i - j;
      modifies arr;
    {
      if (arr[j] > arr[j + 1]) {
        var temp: int := arr[j + 1];
        arr[j + 1] := arr[j];
        arr[j] := temp;
      }
      j := j + 1;
    }
    i := i - 1;
  }
}
```

7.2.3 Implementación en KeY

```
/*@ public normal_behavior
  @  requires arr.length > 0;
  @  ensures (\forall  int a, b;
  @                    0 <= a && a <= b && b < arr.length;
  @                    arr[a] <= arr[b]);
  @  ensures \dl_seqPerm(\dl_array2seq(arr), \old(\dl_array2seq(arr)));
  @  assignable arr[*];
  @*/
public void bubbleSort(int[] arr) {
  int N = arr.length;
  int i = N - 1;

  /*@  loop_invariant 0 <= i && i < N;
    @  loop_invariant (\forall int a, b;
    @                          i <= a && a <= b && b < N;
    @                          arr[a] <= arr[b]);
    @  loop_invariant (\forall int a, b;
    @                          0 <= a && a <= i && i < b && b < N;
    @                          arr[a] <= arr[b]);
    @  loop_invariant \dl_seqPerm(\dl_array2seq(arr), \old(\dl_array2seq(arr)));
    @  assignable arr[*];
```

```
22      @  decreases i;
23      @*/
24    while(i != 0) {
25      int j = 0;
26
27      /*@  loop_invariant 0 <= j && j <= i;
28        @  loop_invariant (\forall int a, b;
29        @                          i <= a && a <= b && b < N;
30        @                          arr[a] <= arr[b]);
31        @  loop_invariant (\forall int a, b;
32        @                          0 <= a && a <= i && i < b && b < N;
33        @                          arr[a] <= arr[b]);
34        @  loop_invariant (\forall int k;
35        @                          0 <= k && k <= j;
36        @                          arr[k] <= arr[j]);
37        @  loop_invariant \dl_seqPerm(\dl_array2seq(arr), \old(\dl_array2seq(arr)));
38        @  assignable arr[*];
39        @  decreases i-j;
40        @*/
41      while(j != i) {
42        if (arr[j] > arr[j + 1]) {
43          int temp = arr[j + 1];
44          arr[j + 1] = arr[j];
45          arr[j] = temp;
46        }
47        j = j + 1;
48      }
49      i = i - 1;
50    }
51  }
```

Una vez que se ejecuta el verificador del *IDE KeY v2.10.0*, queda un *goal* sin cerrar que tiene relación con la demostración de la propiedad de integridad. Por esta razón, se utilizan ciertas reglas manualmente para ayudar al verificador a probar dicha post-condición.

Sobre la pestaña *Sequent*, se debe realizar el siguiente procedimiento:

1. Hacer *click* sobre el símbolo implica ($\Rightarrow$).
2. Hacer *click* sobre *Strategy Macros* > *Auto Pilot* > *Finish Symbolic Execution*.
3. Hacer *click* en el *sequent seqPerm*.
4. Hacer *click* sobre la regla *seqPermFromSwap*.
5. Hacer *click* sobre el símbolo implica ($\Rightarrow$).
6. Hacer *click* sobre *Strategy Macros* > *Close Probable Goals Below*.

Una vez realizado estos pasos, queda demostrado correctamente la verificación formal de las post-condiciones del algoritmo.

7.3 Selection Sort

Algoritmo de ordenamiento que recibe por parámetro un *array* y ordena sus elementos de izquierda a derecha, seleccionando repetitivamente al menor elemento de la partición no ordenada e insertándolo al final de la partición ordenada.

7.3.1 Metodología

```
1  SelectionSort (arr: array<int>)
2    require
3      N > 0    -- Requerimiento de acotamiento
4    ensure
5      sorted_increasing(arr[..])     -- Propiedad de ordenamiento
6      perm(arr[..], old(arr[..]))    -- Propiedad de integridad
7    local
8      N: int
9      i: int
10     j: int
11     min: int
12   do
13     init
14       N := arr.length
15       i := 0
16     invariant
17       i in [0..N)                      -- Invariante de acotamiento
18       sorted_increasing(arr[0..i])     -- Invariante esencial
19       arr[0..i) <= arr[i..N)           -- Invariante esencial
20       perm(arr[..], old(arr[..]))      -- Invariante esencial
21     until
22       i = N - 1
23     step
24       init
25         j := i + 1
26         min := i
27       invariant
28         j in (i..N]                      -- Invariante de acotamiento
29         min in [i..N)                    -- Invariante de acotamiento
30         arr[min] <= arr[i..j)            -- Invariante esencial
31         prem(arr[..], old(arr[..]))      -- Invariante esencial
32       until
33         j = N
34       step
35         if arr[j] < arr[min] then min := j
36         j := j + 1
37       variant
38         N - j
39       end
40       swap(i, min)
41       i := i + 1
42     variant
43       N - 1 - i
44     end
45   end
```

Se recibe por parámetro un *array* a ordenar y tiene dos post-condiciones:

1. Propiedad de ordenamiento:

sorted_increasing(arr[..]). (5 :: ensure)

2. Propiedad de integridad:

perm(arr[..], old(arr[..])). (6 :: ensure)

Propiedad de ordenamiento

Se aplica el método de elegir y probar la post-condición de ordenamiento utilizando dos invariantes.

Se determina que existe una partición que tiene los elementos ya ordenados y otra partición que tiene los elementos a ordenar.

De la partición ordenada, se deriva una invariante utilizando el método de "sustituir constante por variable". En este caso, se sustituye N por una nueva variable i, de tal forma que todos los elementos que están a la izquierda del índice i están ordenados.

sorted_increasing(arr[0..i]). (18 :: invariant)

Por la naturaleza del *Selection Sort*, siempre va a ordenar de izquierda a derecha, agregando al elemento con menor valor de la partición a ordenar, en la partición ordenada. Este comportamiento implica que todos los elementos pertenecientes a la partición ordenada son de menor valor que los pertenecientes a la partición desordenada.

arr[0..i) $\leq$ arr[i..N]. (19 :: invariant)

Por otro lado, los elementos que están a la derecha del índice i, determinan la variante.

N - 1 - i. (43 :: variant)

El índice i, se inicializa en 0 y queda acotado entre 0 y el largo del *array*:

i := 0 (15 :: init)

i $\in$ [0..N). (17 :: invariant)

En cada paso se realiza la lógica de seleccionar el elemento menor de la partición a ordenar (arr[min]) y moverlo al último lugar de la partición ordenada (arr[i]). Además, se incrementa el índice i en una unidad.

$$\text{swap(i, min)} \qquad (40 :: \text{step})$$

$$\text{i} := \text{i} + 1. \qquad (41 :: \text{step})$$

Como condición de terminación, se debe recorrer todo el *array*.

$$\text{i} = \text{N} - 1. \qquad (22 :: \text{until})$$

Cuando se termina de recorrer todo el *array*, el índice i apunta al último elemento, por lo tanto, la invariante esencial que fue determinada por la partición ordenada, deriva automáticamente a la post-condición de la propiedad de ordenamiento.

$$\text{sorted\_increasing(arr[0..N])}. \qquad (5 :: \text{ensure})$$

Resta por explicar los pasos para seleccionar el menor elemento de la partición a ordenar [i..N).

Para poder realizar esto, es necesario tener un puntero que esté dentro de la partición a ordenar que apunte al elemento de menor valor encontrado hasta el momento. Se inicializa en la posición i y está acotado dentro de la partición a ordenar:

$$\text{min} := \text{i} \qquad (26 :: \text{init})$$

$$\text{min} \in \text{[i..N)}. \qquad (29 :: \text{invariant})$$

Dentro de la partición a ordenar (arr[i..N)) se realiza una sub-partición tal que todos los elementos que están a la izquierda de un nuevo índice j (arr[i..j)) deben ser mayores al valor que toma el *array* en la posición del puntero *min*:

$$\text{arr[min]} \leq \text{arr[i..j)}. \qquad (30 :: \text{invariant})$$

El índice j se inicializa en la posición que está a la derecha de i y está acotado dentro de la partición de elementos a ordenar:

$$\text{j} := \text{i} + 1 \qquad (25 :: \text{init})$$

$$\text{j} \in \text{(i..N]}. \qquad (28 :: \text{invariant})$$

Por otro lado, la otra sub-partición que queda definida dentro de la partición de elementos no ordenados, serían los elementos que están a la derecha de j (arr[j..N)), que serán parte de la variante.

N - j. (38 :: variant)

En cada paso, se compara si el elemento en la posición j es menor al elemento en la posición *min*. En caso de ser así, se actualiza el puntero *min* a la posición de j.

if arr[j] < arr[min] ⇒ min := j. (35 :: step)

Además, se incrementa el índice j en una unidad:

j := j + 1. (36 :: step)

Los pasos terminan cuando el índice j llega al final del *array*.

j = N. (33 :: until)

Bajo esa condición ($j = N$), implica que el valor que toma el *array* en la posición *min* es menor al valor de cada elemento de la partición arr[i..j] = arr[i..N], cumpliendo así la condición para pertenecer a la partición ordenada.

Propiedad de integridad

Para demostrar que los elementos finales sean una permutación de los elementos iniciales, basta con observar que la única operación que se realiza sobre el *array*, son *swaps* entre el elemento de menor valor de la partición a ordenar con el elemento en la posición i. Como ambos elementos siempre están dentro del *array*, se mantiene la propiedad de integridad en cada uno de los pasos.

perm(arr[..], old(arr[..])) (31 :: invariant)

perm(arr[..], old(arr[..])). (20 :: invariant)

Automáticamente se puede derivar a la verificación de la propiedad de integridad.

perm(arr[..], old(arr[..])). (6 :: ensure)

7.3.2 Implementación en Dafny

```
method SelectionSort(arr: array<int>)
  requires arr.Length > 0
  ensures forall a, b :: 0 <= a <= b < arr.Length ==> arr[a] <= arr[b]
  ensures multiset(arr[..]) == multiset(old(arr[..]))
  modifies arr
{
  var N: nat := arr.Length;
  var i: nat := 0;

  while(i != N - 1)
    invariant 0 <= i < N
    invariant forall a, b :: 0 <= a < i <= b < N ==> arr[a] <= arr[b]
    invariant forall a, b :: 0 <= a <= b <= i ==> arr[a] <= arr[b]
    invariant multiset(arr[..]) == multiset(old(arr[..]))
    decreases N - 1 - i
    modifies arr
  {
    var j: int := i + 1;
    var min: int := i;

    while(j != N)
      invariant i < j <= N
      invariant i <= min < N
      invariant forall k :: i <= k < j ==> arr[min] <= arr[k]
      invariant multiset(arr[..]) == multiset(old(arr[..]))
      decreases N - j;
      modifies arr
    {
      if (arr[j] < arr[min])
      {
        min := j;
      }
      j := j + 1;
    }
    var temp: int := arr[min];
    arr[min] := arr[i];
    arr[i] := temp;

    i := i + 1;
  }
}
```

7.3.3 Implementación en KeY

```
/*@ public normal_behavior
  @  requires arr.length > 0;
  @  ensures (\forall  int a, b;
  @                    0 <= a && a <= b && b < arr.length;
  @                    arr[a] <= arr[b]);
  @  ensures \dl_seqPerm(\dl_array2seq(arr), \old(\dl_array2seq(arr)));
  @*/
public void selectionSort(int[] arr) {
  int N = arr.length;
  int i = 0;

  /*@  loop_invariant 0 <= i && i < N;
    @  loop_invariant (\forall int a, b;
    @                          0 <= a && a < i && i <= b && b < N;
    @                          arr[a] <= arr[b]);
    @  loop_invariant (\forall int a, b;
    @                          0 <= a && a <= b && b <= i;
    @                          arr[a] <= arr[b]);
```

```
19      @  loop_invariant \dl_seqPerm(\dl_array2seq(arr), \old(\dl_array2seq(arr)));
20      @  decreases N - 1 - i;
21      @  assignable arr[*];
22      @*/
23    while(i != N - 1) {
24      int j = i + 1;
25      int min = i;
26
27      /*@  loop_invariant i < j && j <= N;
28        @  loop_invariant i <= min && min < N;
29        @  loop_invariant (\forall int k;
30        @                          i <= k && k < j;
31        @                          arr[min] <= arr[k]);
32        @  loop_invariant \dl_seqPerm(\dl_array2seq(arr), \old(\dl_array2seq(arr)));
33        @  decreases N - j;
34        @  assignable \strictly_nothing;
35        @*/
36      while(j != N) {
37        if (arr[j] < arr[min]) {
38          min = j;
39        }
40        j = j + 1;
41      }
42      int temp = arr[min];
43      arr[min] = arr[i];
44      arr[i] = temp;
45
46      i = i + 1;
47    }
48  }
```

Una vez que se ejecuta el verificador del *IDE KeY v2.10.0*, quedan dos *goals* sin cerrar. Ambos *goals* tienen relación con la demostración de la propiedad de integridad. Por esta razón, se utilizan ciertas reglas manualmente para ayudar al verificador a probar dicha post-condición.

Sobre la pestaña *Sequent*, se debe realizar exactamente el mismo procedimiento para ambos *goals* pendientes:

1. Hacer *click* sobre el símbolo implica (⇒).
2. Hacer *click* sobre *Strategy Macros* > *Auto Pilot* > *Finish Symbolic Execution*.
3. Hacer *click* en el *sequent seqPerm*.
4. Hacer *click* sobre la regla *seqPermFromSwap*.
5. Hacer *click* sobre el símbolo implica (⇒).
6. Hacer *click* sobre *Strategy Macros* > *Close Probable Goals Below*.

Una vez realizado estos pasos, se habrá demostrado correctamente la verificación formal de las post-condiciones del algoritmo.

7.4 Comb Sort

Es un algoritmo de ordenamiento que optimiza al *Bubble Sort*, ya que *Bubble Sort* va comparando elementos adyacentes y removiendo las inversiones una por una y *Comb Sort* utiliza comparaciones con un *gap* optimizado y variable. Este *gap* comienza siendo grande y tiende a achicarse en cada iteración.

El factor de decrecimiento que se utiliza para calcular el *gap* en cada iteración, fue estudiado empíricamente y se llegó a la conclusión que 1.3 era el valor más óptimo.

7.4.1 Metodología

Para la realización del algoritmo, se utiliza un *gap* variable que se calcula de la siguiente forma:

```
1  NewGap (prevGap: int, N: int): (gap: int)
2    require
3      prevGap in [1..N]     -- Requerimiento de acotamiento
4    ensure
5      gap in [1..N]                          -- Propiedad de acotamiento
6      prevGap = 1 then gap = prevGap = 1     -- Propiedad esencial
7      prevGap > 1 then gap < prevGap         -- Propiedad esencial
8    do
9      init
10       gap := (prevGap * 10) / 13
11     adjust
12       if gap < 1 then gap := 1
13     end
14   end
```

Recibe como parámetro el *gap* previo y el largo del *array* denominado *N*.

Es necesario que el *gap* previo esté acotado de la siguiente manera:

$$\text{prevGap} \in [1..N]. \qquad (3 :: \text{require})$$

Se realiza el cálculo del nuevo *gap* utilizando el *gap* previo y un factor matemático de optimización.

$$\text{gap} := (\text{prevGap} * 10) \ / \ 13. \qquad (10 :: \text{init})$$

En caso de que el nuevo *gap* calculado sea un valor menor a 1, se le asigna 1 por defecto.

$$\text{if gap} < 1 \Rightarrow \text{gap} := 1. \qquad (12 :: \text{adjust})$$

Como el *gap* previo no puede ser mayor a N y el cálculo del nuevo *gap* se ajusta para que no pueda ser menor a 1, se puede afirmar que el nuevo *gap* está acotado de la siguiente manera:

$$\text{gap} \in [1..\text{N}]. \qquad (5 :: \text{ensure})$$

Además, al ser el factor de multiplicación del nuevo *gap* menor a 1, si el previo era mayor a 1, se puede asegurar que el nuevo *gap* es menor al previo.

$$\text{prevGap} > 1 \Rightarrow \text{gap} < \text{prevGap}. \qquad (7 :: \text{ensure})$$

Por otro lado, si el *gap* previo era 1, entonces por el ajuste que se realiza en el cálculo del *gap* nuevo, también será 1.

$$\text{prevGap} = 1 \Rightarrow \text{gap} = \text{prevGap} = 1. \qquad (6 :: \text{ensure})$$

A continuación se detalla la metodología del *Comb Sort*:

```
1  CombSort (arr: array<int>)
2    require
3      N > 1    -- Propiedad de acotamiento
4    ensure
5      sorted_increasing(arr[..])     -- Propiedad de ordenamiento
6      perm(arr[..], old(arr[..]))    -- Propiedad de integridad
7    local
8      N: int
9      gap: int
10     sorted: bool
11     i: int
12   do
13     invariant
14       gap in [1..N]    -- Invariante de acotamiento
15       sorted = true then forall k in [0..N-gap) then arr[k] <= arr[k+gap] --Invariante
16       perm(arr[..], old(arr[..]))    -- Invariante esencial                  esencial
17     init
18       N := arr.length
19       gap := N
20       sorted := false
21     until
22       gap = 1 and sorted = true
23     step
24       invariant
25         0 <= i < i + gap <= N    -- Invariante de acotamiento
26         sorted = true then forall k in [0..i) then arr[k] <= arr[k+gap] -- Invariante
27         prem(arr[..], old(arr[..]))    -- Invariante esencial               esencial
28       init
29         gap := NewGap(gap, N)
30         sorted := true
31         i := 0
32       until
33         i + gap = N
34       step
35         if arr[i] > arr[i + gap]
36           then swap(i, i + gap); sorted := false;
37         i := i + 1
38       variant
```

```
39          N - (i + gap)
40        end
41      variant
42        if gap > 1 then gap else |inversions(arr[..])|
43      end
44    end
```

Se recibe por parámetro un *array* a ordenar y tiene dos post-condiciones:

1. Propiedad de ordenamiento

$$\text{sorted\_increasing(arr[..])}. \qquad (5 :: \text{ensure})$$

2. Propiedad de integridad

$$\text{perm(arr[..], old(arr[..]))}. \qquad (6 :: \text{ensure})$$

Propiedad de ordenamiento

Se aplica el método de elegir y probar la post-condición de ordenamiento utilizando una invariante.

Se determina que existe una partición que tiene los elementos ya ordenados y otra partición que tiene los elementos a ordenar. El problema para delimitar estas particiones, es que sus elementos pertenecientes no están seguidos, sino que están a un *gap* de distancia entre ellos. Sólo estarán seguidos cuando *gap* sea 1.

Por lo comentado anteriormente, de la partición ordenada, se deriva una invariante utilizando el método "sustituir constante por variable". En este caso, se sustituye N por N-*gap*.

$$\text{sorted} = \text{true} \Rightarrow \forall\ \text{k} \in [0..\text{N-gap}) \Rightarrow \text{arr[k]} \leq \text{arr[k + gap]}. \qquad (15 :: \text{invariant})$$

Por otro lado, los elementos que pertenecen a la partición a ordenar, forman parte de la variante. Mientras que el *gap* sea mayor a 1, la variante será el *gap*. Cuando el *gap* sea 1, se revisa la cantidad de inversiones que existen entre los elementos.

$$\text{if gap} > 1 \Rightarrow \text{gap else } |\text{inversions(arr[..])}|. \qquad (42 :: \text{variant})$$

Además *gap* se inicializa en N y por el comportamiento que tiene el método *NewGap*, se puede derivar a la invariante de acotamiento:

$$\text{gap} := \text{N} \qquad (19 :: \text{init})$$

$$\text{gap} \in [1..N]. \qquad (14 :: \text{invariant})$$

Se utiliza una variable *sorted* inicializado en *false*, que determina si el *array* está ordenado o no.

$$\text{sorted} := \text{false}. \qquad (20 :: \text{init})$$

En cada paso, se calcula el nuevo *gap* y se asume que la variable *sorted* es *true*.

$$\text{gap} := \text{NewGap(gap, N)} \qquad (29 :: \text{init})$$

$$\text{sorted} := \text{true}. \qquad (30 :: \text{init})$$

Además, se realiza la lógica que compara cada elemento de la partición a ordenar con su posición + *gap* y se realiza la lógica de *swaps* en caso de ser necesario.

Como condición de terminación, el *gap* tiene que ser 1 y la variable *sorted* verdadera, implicando que se comparó cada elemento y están ordenados.

$$\text{gap} = 1 \land \text{sorted} = \text{true}. \qquad (22 :: \text{until})$$

Como se puede observar, la variable *N - gap* que fue determinada para especificar la partición ordenada, ahora será *N - 1*. Esto implica que se puede derivar en la post-condición:

$$\forall\ \text{k} \in [0..\text{N-1}) \Rightarrow \text{arr[k]} \leq \text{arr[k + 1]}. \qquad (5 :: \text{ensure})$$

Resta por explicar los pasos para comparar cada elemento de la partición a ordenar, con su posición + *gap* y la lógica de *swaps* que se realiza.

Dentro de la partición a ordenar, se realiza una sub-partición utilizando el método "sustituir constante por variable", sustituyendo *N-gap* por una variable nueva *i*, tal que dentro del rango [0..i), todos los elementos son menores o iguales a los de su posición + *gap*.

$$\text{sorted} = \text{true} \Rightarrow \forall\ \text{k} \in [0..\text{i}) \Rightarrow \text{arr[k]} \leq \text{arr[k + gap]} \qquad (26 :: \text{invariant})$$

El índice *i* se inicializa en la posición 0 y está acotado entre [0..i+gap]:

$$\text{i} := 0 \qquad (31 :: \text{init})$$

$$0 \leq \text{i} < \text{i} + \text{gap} \leq \text{N}. \qquad (25 :: \text{invariant})$$

En cada paso, se controla si el valor que tiene el *array* en la posición *i*, es mayor al valor que tiene el *array* en la posición *i + gap*:

$$\text{if arr[i]} > \text{arr[i + gap]}. \qquad (35 :: \text{step})$$

En caso de ser mayor, se realiza un *swap* entre la posición *i* con *i + gap* y se asigna *sorted* como *false*.

$$\text{then swap(i, i + gap); sorted} := \text{false}; \qquad (36 :: \text{step})$$

Además se incrementa el índice *i* en una unidad:

$$\text{i} := \text{i} + 1. \qquad (37 :: \text{step})$$

También se determina otra partición delimitada por los elementos que están entre *i + gap* y el final del *array*. Esto deriva directamente en la variante:

$$\text{N - (i + gap)}. \qquad (39 :: \text{variant})$$

Por lo tanto, como condición de terminación, se realizan pasos hasta que *i + gap* sea igual a N:

$$\text{i + gap} = \text{N}. \qquad (33 :: \text{until})$$

En este caso, al ser *i + gap = N*, la invariante esencial que está delimitada por [0..i):

$$\text{sorted} = \text{true} \Rightarrow \forall\ \text{k} \in [0..\text{i}) \Rightarrow \text{arr[k]} \leq \text{arr[k + gap]} \qquad (26 :: \text{invariant})$$

deriva en la invariante esencial de la partición ordenada [0..N-gap], quedando estos nuevos elementos como parte de la partición ordenada:

$$\text{sorted} = \text{true} \Rightarrow \forall\ \text{k} \in [0..\text{N-gap}) \Rightarrow \text{arr[k]} \leq \text{arr[k + gap]} \qquad (15 :: \text{invariant})$$

Propiedad de integridad

Para demostrar que los elementos finales sean una permutación de los elementos iniciales, basta con observar que la única operación que se realiza sobre el *array*, son *swaps* entre el elemento de la partición a ordenar. Como ambos elementos siempre están dentro del *array*, se mantiene la propiedad de integridad en cada uno de los pasos.

perm(arr[..], old(arr[..])) (27 :: invariant)

perm(arr[..], old(arr[..])). (16 :: invariant)

Automáticamente se puede derivar a la verificación de la propiedad de integridad.

perm(arr[..], old(arr[..])). (6 :: ensure)

7.4.2 Implementación en Dafny

```
method CombSort(arr: array<int>)
  requires arr.Length > 1;
  ensures forall k :: 0 <= k < arr.Length - 1 ==> arr[k] <= arr[k + 1]
  ensures multiset(arr[..]) == multiset(old(arr[..]))
  modifies arr
  decreases *
{
  var N: nat := arr.Length;
  var gap: nat := N;
  var sorted: bool := false;

  while(gap != 1 || sorted == false)
    invariant 1 <= gap <= N
    invariant sorted == true ==> forall k :: 0 <= k < N-gap ==> arr[k] <= arr[k+gap]
    invariant multiset(arr[..]) == multiset(old(arr[..]))
    decreases *
  {
    gap := NewGap(gap, N);
    sorted := true;
    var i: int := 0;

    while(i + gap != N)
      invariant 0 <= i < i + gap <= N
      invariant sorted == true ==> forall k :: 0 <= k < i ==> arr[k] <= arr[k+gap]
      invariant multiset(arr[..]) == multiset(old(arr[..]))
      decreases N - (i + gap)
    {
      if (arr[i] > arr[i + gap])
      {
        var temp: int := arr[i];
        arr[i] := arr[i + gap];
        arr[i + gap] := temp;

        sorted := false;
      }
      i := i + 1;
    }
  }
}
```

```
method NewGap(prevGap: nat, length: nat) returns (gap: nat)
  requires 1 <= prevGap <= length
  ensures 1 <= gap <= length
  ensures prevGap == 1 ==> gap == prevGap == 1
  ensures prevGap > 1 ==> gap < prevGap
```

```
{
  gap := (prevGap * 10) / 13;
  if (gap < 1)
  {
    gap := 1;
  }
}
```

Se pudo demostrar tanto la propiedad de integridad como la propiedad de ordenamiento del algoritmo.

Con respecto a la terminación, se puede observar en cada paso de la iteración principal, que el *gap* va decreciendo hasta llegar a 1.

En ese momento, el *gap* permanecerá en 1 y lo único que podría determinar la terminación del algoritmo sería controlar el decrecimiento de la cantidad de inversiones.

La condición del *decreases* que habría que poner en la iteración principal sería la siguiente:

```
decreases if gap > 1 then gap else |inversions(arr[..])|
```

Con la función *inversions* de la siguiente manera:

```
function inversions (s: seq<int>): set<(nat, nat)>
{
  set i: nat, j :nat | 0 <= i < j < |s| && s[i] > s[j] :: (i, j)
}
```

Como *Dafny* no puede asegurar que las inversiones vayan a decrecer siempre, se tendría que realizar un conjunto de lemas para demostrar que efectivamente las inversiones terminan decreciendo hasta cero.

Queda fuera del alcance de la tesis la realización de dichos lemas.

7.4.3 Implementación en KeY

```
/*@ public normal_behavior
  @  requires arr.length > 1;
  @  ensures (\forall int k;
  @                  0 <= k && k < arr.length - 1;
  @                  arr[k] <= arr[k + 1]);
  @  ensures \dl_seqPerm(\dl_array2seq(arr), \old(\dl_array2seq(arr)));
  @  diverges true;
  @  assignable arr[*];
  @*/
public void combSort(int[] arr) {
  int N = arr.length;
  int gap = N;
  boolean sorted = false;
```

```
14
15    /*@  loop_invariant 1 <= gap && gap <= N;
16      @  loop_invariant sorted == true ==> (\forall int k;
17      @                                              0 <= k && k < N - gap;
18      @                                              arr[k] <= arr[k + gap]);
19      @  loop_invariant \dl_seqPerm(\dl_array2seq(arr), \old(\dl_array2seq(arr)));
20      @  assignable arr[*];
21      @  diverges true;
22      @*/
23    while(gap != 1 || sorted == false) {
24      gap = newGap(gap, N);
25      sorted = true;
26      int i = 0;
27
28      /*@  loop_invariant 0 <= i && i < i + gap && i + gap <= N;
29        @  loop_invariant sorted == true ==> (\forall int k;
30        @                                              0 <= k && k < i;
31        @                                              arr[k] <= arr[k + gap]);
32        @  loop_invariant \dl_seqPerm(\dl_array2seq(arr), \old(\dl_array2seq(arr)));
33        @  decreases N - (i + gap);
34        @  assignable arr[*];
35        @*/
36      while(i + gap != N) {
37        if (arr[i] > arr[i + gap]) {
38          int temp = arr[i];
39          arr[i] = arr[i + gap];
40          arr[i + gap] = temp;
41
42          sorted = false;
43        }
44        i = i + 1;
45      }
46    }
47  }
```

```
1   /*@ public normal_behavior
2     @  requires 1 <= prevGap && prevGap <= length;
3     @  ensures 1 <= \result && \result <= length;
4     @  ensures prevGap == 1 ==> \result == prevGap && prevGap == 1;
5     @  ensures prevGap > 1 ==> \result < prevGap;
6     @*/
7   private int newGap(int prevGap, int length) {
8     int gap = (prevGap * 10) / 13;
9     if (gap < 1) {
10      gap = 1;
11    }
12    return gap;
13  }
```

De la misma manera que sucede en *Dafny*, en *KeY* también queda excluida la demostración de la terminación del algoritmo, porque sucede exactamente el mismo problema comentando en la sección anterior.

7.5 Quick Sort

Algoritmo de ordenamiento que utiliza la técnica divide para conquistar. Los menores quedan a la izquierda y los mayores a la derecha. Toma un elemento como *pivot* y particiona los elementos hacia ambos lados del mismo. A medida que se van reduciendo dichas particiones, quedan ordenados los elementos.

7.5.1 Metodología

El algoritmo está conformado por un método *wrapper*, un método principal que dado un rango de elementos orquesta las llamadas recursivas y un método auxiliar que se encarga de realizar las particiones, denominado *Partition*.

7.5.1.1 Metodología del método *wrapper*

```
1  QuickSort (arr: array<int>)
2    require
3      N > 0    -- Requerimiento de acotamiento
4    ensure
5      sorted_increasing(arr[..])     -- Propiedad de ordenamiento
6      perm(arr[..], old(arr[..]))    -- Propiedad de integridad
7    local
8      N: int
9      from: int
10     to: int
11   do
12     init
13       N := arr.Length
14       from := 0
15       to := N - 1
16     step
17       QuickSort(arr, from, to)
18   end
```

Es el encargado de inicializar los valores *from* y *to* del método principal, de tal forma que representen respectivamente a la posición del extremo izquierdo y derecho del *array*:

$$\text{from} := 0 \qquad (14 :: \text{init})$$

$$\text{to} := \text{N - 1.} \qquad (15 :: \text{init})$$

Una vez inicializados dichos valores, se realiza la llamada del método principal, que será el encargado de realizar la lógica de ordenamiento.

Quicksort(arr, from, to). (17 :: step)

Se desea probar que se cumple tanto la propiedad de ordenamiento:

sorted_increasing(arr[..]) (5 :: ensure)

como la propiedad de integridad de sus elementos:

perm(arr[..], old(arr[..])). (6 :: ensure)

Para poder demostrar que se cumplen ambas propiedades, se explica primero la metodología del método principal y luego se profundiza con el método *Partition*.

7.5.1.2 Metodología del método principal

```
1  QuickSort (arr: array<int>, from: int, to: int)
2    require
3      from in [0..N]                                  -- Requerimiento de acotamiento
4      to in [-1..N)                                   -- Requerimiento de acotamiento
5      from <= to + 1                                  -- Requerimiento de acotamiento
6      from > 0 then arr[from - 1] <= arr[from..to]    -- Requerimiento de ordenamiento
7      to < N - 1 then arr[to + 1] >= arr[from..to]    -- Requerimiento de ordenamiento
8    ensure
9      from > 0 then arr[from - 1] <= arr[from..to]    -- Propiedad de ordenamiento
10     to < N - 1 then arr[to + 1] >= arr[from..to]    -- Propiedad de ordenamiento
11     sorted_increasing(arr[from..to])                -- Propiedad de ordenamiento
12     perm(arr[..], old(arr[..]))                     -- Propiedad de integridad
13     arr[0..from) = old(arr[0..from))                -- Propiedad de integridad
14     arr(to..N) = old(arr(to..N))                    -- Propiedad de integridad
15   local
16     m: int
17   do
18     init
19       m := Partition(arr, from, to)
20     until
21       from >= to
22     step
23       Quicksort(arr, from, m - 1)
24       Quicksort(arr, m + 1, to)
25     variant
26       to - from
27     end
28   end
```

Se recibe por parámetro un *array* y un rango a ordenar determinado por *from* y *to*.

Se requiere que el parámetro *from*, esté acotado entre el comienzo del *array* y el largo del mismo, porque en la segunda llamada recursiva, se asigna a *from* la posición $m + 1$ pudiendo este ser a lo sumo N:

from $\in$ [0..N]. (3 :: require)

Se requiere que el parámetro *to*, esté acotado entre -1 y el último elemento del *array*. Esto es necesario porque en la primera llamada recursiva se asigna a *to* la posición *m - 1*, pudiendo esta expresión ser como mínimo -1:

$$\text{to} \in [\text{-1..N}]. \qquad (4 :: \text{require})$$

La variable *from* puede valer a lo sumo *to + 1*:

$$\text{from} \leq \text{to} + 1. \qquad (5 :: \text{require})$$

Se controla que la variable *from* sea menor a *to*, en caso de no ser así, se termina la ejecución porque el rango no es válido.

$$\text{from} \geq \text{to}. \qquad (21 :: \text{until})$$

Para los casos con un rango válido (*from* $<$ *to*), se llama a un método auxiliar *Partition* que devuelve una posición del *array* denominada *m* y reordena el *array* de tal forma que todos los elementos menores estén a la izquierda y los mayores a la derecha:

$$\text{m} := \text{Partition(arr, from, to)}. \qquad (25 :: \text{init})$$

Se realiza una llamada recursiva con los elementos que están a la izquierda de *m*:

$$\text{Quicksort(arr, from, m - 1)} \qquad (23 :: \text{step})$$

y otra llamada recursiva con los elementos que están a la derecha de *m*:

$$\text{Quicksort(arr, m + 1, to)}. \qquad (24 :: \text{step})$$

Como se puede observar en este comportamiento, existen múltiples llamadas recursivas que trabajan sobre rangos que no colisionan entre ellos.

Por esta razón, los elementos que estén en un rango a la izquierda del rango actual, tienen menor valor que los elementos actuales:

$$\text{from} > 0 \Rightarrow \text{arr[from - 1]} \leq \text{arr[from..to]}. \qquad (9 :: \text{ensure})$$

De forma análoga, los elementos que están en un rango a la derecha del rango actual, tienen mayor valor que los elementos actuales:

$$\text{to} < \text{N - 1} \Rightarrow \text{arr[to + 1]} \geq \text{arr[from..to]}. \qquad (10 :: \text{ensure})$$

Estas post-condiciones están atadas también a las siguientes pre-condiciones:

$$\text{from} > 0 \Rightarrow \text{arr[from - 1]} \leq \text{arr[from..to]} \quad (6 :: \text{require})$$

$$\text{to} < \text{N - 1} \Rightarrow \text{arr[to + 1]} \geq \text{arr[from..to]}. \quad (7 :: \text{require})$$

El algoritmo termina porque el rango determinado por *from* y *to* decrece en cada llamada:

$$\text{to - from}. \quad (26 :: \text{variant})$$

Cuando ya no existen más llamadas recursivas pendientes, el algoritmo habrá ordenado todo el *array* de elementos:

$$\text{sorted\_increasing(arr[from..to])}. \quad (11 :: \text{ensure})$$

Para demostrar que los elementos iniciales sean los mismos que los finales, se puede observar a continuación que el algoritmo *Partition* mantiene la propiedad de integridad de los elementos.

Por esta razón se puede determinar que existe una permutación entre los elementos actuales y finales del *array*.

$$\text{perm(arr[..], old(arr[..]))}. \quad (12 :: \text{ensure})$$

Además, como el método principal *Quicksort* no realiza operaciones sobre elementos que están afuera del rango *from* y *to*, se puede asegurar que la posición de dichos elementos son exactamente las mismas:

$$\text{arr[0..from)} = \text{old(arr[0..from))} \quad (13 :: \text{ensure})$$

$$\text{arr(to..N)} = \text{old(arr(to..N))}. \quad (14 :: \text{ensure})$$

7.5.1.3 Metodología del método Partition

```
1  Partition (arr: array<int>, from: int, to: int): (l: int)
2    require
3      0 <= from <= to < N                              -- Requerimiento de acotamiento
4      from > 0 then arr[from - 1] <= arr[from..to]     -- Requerimiento personalizado
5      to < N - 1 then arr[to + 1] >= arr[from..to]     -- Requerimiento personalizado
6    ensure
7      l in [from..to]                                  -- Propiedad de acotamiento
8      arr[l] > arr[from..l)                            -- Propiedad de ordenamiento
9      arr[l] <= arr(l..to]                             -- Propiedad de ordenamiento
```

```
10      perm(arr[..], old(arr[..]))                    -- Propiedad de integridad
11      arr[0..from) = old(arr[0..from))               -- Propiedad de integridad
12      arr(to..N) = old(arr(to..N))                   -- Propiedad de integridad
13      from > 0 then arr[from - 1] <= arr[from..to]    -- Propiedad personalizada
14      to < N - 1 then arr[to + 1] >= arr[from..to]    -- Propiedad personalizada
15    local
16      N: int
17      r: int
18      pivot: int
19    do
20      invariant
21        from <= l <= r <= to                          -- Invariante de acotamiento
22        pivot = arr[l]
23        pivot > arr[from..l)                          -- Invariante esencial
24        pivot <= arr(r..to])                          -- Invariante esencial
25        perm(arr[..], old(arr[..]))                   -- Invariante esencial
26        arr[0..from) = old(arr[0..from))              -- Invariante esencial
27        arr(to..N) = old(arr(to..N))                  -- Invariante esencial
28        from > 0 then arr[from - 1] <= arr[from..to]   -- Invariante esencial
29        to < N - 1 then arr[to + 1] >= arr[from..to]   -- Invariante esencial
30      init
31        N := arr.length
32        l := from
33        r := to
34        pivot := arr[from]
35      until
36        l = r
37      step
38        if pivot <= arr[r]
39          then r := r - 1
40          else arr[l] := arr[r]; arr[r] := arr[l + 1]
41               arr[l + 1] := pivot; l := l + 1
42      variant
43        r - l
44      end
45    end
```

Se recibe por parámetro un *array* y un rango a ordenar determinado por *from* y *to*.

Para poder demostrar que el algoritmo *Partition* realiza la lógica adecuadamente, se utilizan 8 post-condiciones:

1. Propiedad de ordenamiento:

El índice resultante l está acotado entre *from* y *to*:

$$\text{l} \in [\text{from..to}]. \qquad (7 :: \text{ensure})$$

Los elementos que están a la derecha de l son mayores al valor que toma el *array* en la posición l:

$$\text{arr[l]} \leq \text{arr[l..to]}. \qquad (8 :: \text{ensure})$$

Los elementos que están a la izquierda de l son menores o iguales al valor que toma el *array* en la posición l:

$$\text{arr[l]} \leq \text{arr[l..to]}. \qquad (9 :: \text{ensure})$$

2. Propiedad de integridad:

El *array* resultante deben ser permutaciones de los elementos que tenía el *array* de entrada:

$$\text{perm(arr[..], old(arr[..])).} \qquad (10 :: \text{ensure})$$

Todos los elementos que están a la izquierda de la posición *from* no forman parte del ordenamiento y por ende conservan sus posiciones iniciales:

$$\text{arr[0..from]} == \text{old(arr[0..from]).} \qquad (11 :: \text{ensure})$$

Todos los elementos que están a la derecha de la posición *to* tampoco forman parte del ordenamiento y por ende también conservan sus posiciones iniciales:

$$\text{arr[to..N]} == \text{old(arr[to..N]).} \qquad (12 :: \text{ensure})$$

3. Propiedad personalizada para el *Quick Sort* recursivo:

Si la partición actual no es la partición más a la izquierda de todas (from $>$ 0), entonces el último elemento de la partición que está inmediatamente a su izquierda, debe ser menor o igual a todos los elementos de la partición actual:

$$\text{from} > 0 \Rightarrow \text{arr[from - 1]} \leq \text{arr[from..to].} \quad (13 :: \text{ensure})$$

Si la partición actual no es la partición más a la derecha de todas (to $<$ N), entonces el primer elemento de la partición que está inmediatamente a la derecha, debe ser mayor o igual a todos los elementos de la partición actual:

$$\text{from} > 0 \Rightarrow \text{arr[from - 1]} \leq \text{arr[from..to].} \quad (14 :: \text{ensure})$$

Propiedad de ordenamiento

Se aplica el método de elegir y probar las post-condiciones de ordenamiento, utilizando invariantes.

El método *Partition*, utiliza un *pivot* para comparar los elementos, se selecciona como *pivot* el valor de *arr[l]*:

$$\text{pivot} = \text{arr[l].} \qquad (22 :: \text{invariant})$$

Se determina que existe una partición a la izquierda de l con los elementos que son de menor valor que *arr[l]* y otra partición a la derecha de l con los elementos que son de mayor valor que *arr[l]*.

De la partición de elementos menores, se deriva una invariante utilizando el método de "sustituir constante por variable". En este caso, se sustituye la constante l resultante por la variable l, de tal forma que todos los elementos que están a la izquierda de l, son menores que el *pivot*:

$$\text{pivot} > \text{arr[from..l]}. \qquad (23 :: \text{invariant})$$

Por otro lado, de la partición con elementos mayores, se deriva otra invariante utilizando el método "sustituir constante por variable". Se sustituye la constante l por una nueva variable r, de tal forma que todos los elementos que están a la derecha de r, son mayores o iguales que el *pivot*:

$$\text{pivot} \leq \text{arr(r..to]}. \qquad (24 :: \text{invariant})$$

Los índices l y r, se inicializan en *from* y *to* respectivamente y están acotados de la siguiente manera:

$$\text{l} := \text{from} \qquad (32 :: \text{init})$$

$$\text{r} := \text{to} \qquad (33 :: \text{init})$$

$$\text{from} \leq \text{l} \leq \text{r} \leq \text{to}. \qquad (21 :: \text{invariant})$$

De las particiones detalladas anteriormente se desprende una partición adicional, comprendida entre [l..r], que son los elementos a recorrer y representan a la variante del algoritmo:

$$\text{r - l}. \qquad (43 :: \text{variant})$$

En cada paso se verifica si el valor que toma el *array* en la posición r es mayor o igual al valor del *pivot*:

$$\text{if pivot} \leq \text{arr[r]}. \qquad (38 :: \text{step})$$

De ser así, se debe agregar el elemento en la posición r a la partición con elementos mayores al *pivot*. Para hacer esto, simplemente se debe decrementar el índice r en una unidad:

$$\text{then r} := \text{r - 1}. \qquad (39 :: \text{step})$$

De no ser así, se deben realizar *swaps* de tal manera que el elemento en la posición de r quede en la partición con elementos menores al *pivot*.

Para que el elemento quede en la partición de valores menores al pivot se asigna el valor de la posición r en la posición l. En la posición r, se asigna el valor de un elemento que aún no haya sido recorrido (l + *1*):

$$\text{else arr[l] := arr[r]; arr[r] := arr[l + 1].} \qquad (40 :: \text{step})$$

Además, en la posición l + *1* se asigna el valor del *pivot*, que era inicialmente en este paso, el valor que tenía el *array* en la posición l. Se incrementa el índice l en una unidad:

$$\text{arr[l + 1] := pivot; l := l + l.} \qquad (41 :: \text{step})$$

Como condición de terminación, se deben juntar los índices l y r, ya que dicho rango contiene a la partición con elementos a recorrer.

$$\text{l = r.} \qquad (36 :: \text{until})$$

En este caso, si $l = r$, las invariantes esenciales derivan a las post-condiciones respectivas.

$$\text{arr[l]} \leq \text{arr[l..to]} \qquad (8 :: \text{ensure})$$

$$\text{arr[l]} \leq \text{arr[l..to].} \qquad (9 :: \text{ensure})$$

Propiedad de integridad

Para demostrar que los elementos finales son una permutación de los elementos iniciales, basta con observar que la única operación que se realiza sobre el *array*, son *swaps* entre elementos que están dentro del rango [from..to]. Por lo tanto, se mantiene la invariante en cada uno de los pasos:

$$\text{perm(arr[..], old(arr[..])).} \qquad (25 :: \text{invariant})$$

Para demostrar que las post-condiciones que aseguran la integridad de los elementos que están a la izquierda de *from* y a la derecha de *to* se cumplen, basta con observar que no existe ninguna operación sobre dichos elementos. Por lo tanto, las invariantes se mantienen:

$$\text{arr[0..from)} = \text{old(arr[0..from))} \qquad (26 :: \text{invariant})$$

$$\text{arr(to..N)} = \text{old(arr(to..N))}. \qquad (27 :: \text{invariant})$$

Una vez finalizado el algoritmo, dichas invariantes derivan automáticamente en sus respectivas post-condiciones:

$$\text{perm(arr[..], old(arr[..]))} \qquad (10 :: \text{ensure})$$

$$\text{arr[0..from]} == \text{old(arr[0..from])} \qquad (11 :: \text{ensure})$$

$$\text{arr[to..N]} == \text{old(arr[to..N])}. \qquad (12 :: \text{ensure})$$

Propiedad personalizada para el Quick Sort recursivo

Las dos post-condiciones que se comentan en esta sección, no son inherentes al comportamiento tradicional del algoritmo *Partition*, sino que son necesarias para ayudar al *Quicksort* recursivo, a poder demostrar su propiedad de ordenamiento.

Si la partición actual no es la partición más a la izquierda de todas (from $>$ 0) entonces el último elemento de la partición que está inmediatamente a su izquierda debe ser menor o igual a todos los elementos de la partición actual:

$$\text{from} > 0 \Rightarrow \text{arr[from - 1]} \leq \text{arr[from..to]}. \quad (13 :: \text{ensure})$$

Si la partición actual no es la partición más a la derecha de todas (to $<$ N) entonces el primer elemento de la partición que está inmediatamente a la derecha debe ser mayor o igual a todos los elementos de la partición actual:

$$\text{to} < \text{N - 1} \Rightarrow \text{arr[to + 1]} \geq \text{arr[from..to]}. \quad (14 :: \text{ensure})$$

Demostrar estas propiedades es muy sencillo, ya que basta con repetirlas exactamente como invariantes y no se realizan operaciones sobre elementos por fuera del rango establecido por *from* y *to*.

$$\text{from} > 0 \Rightarrow \text{arr[from - 1]} \leq \text{arr[from..to]}(28 :: \text{invariant})$$

$$\text{to} < \text{N - 1} \Rightarrow \text{arr[to + 1]} \geq \text{arr[from..to]}.(29 :: \text{invariant})$$

Además, se pide como requerimiento que esa propiedad se cumpla desde la invocación del método *Partition*, ya que es un comportamiento inherente a la lógica del *Quick Sort* cuando se realizan las llamadas recursivas:

$$\text{from} > 0 \Rightarrow \text{arr[from - 1]} \leq \text{arr[from..to]} \quad (4 :: \text{require})$$

$$\text{to} < \text{N - 1} \Rightarrow \text{arr[to + 1]} \geq \text{arr[from..to]}. \quad (5 :: \text{require})$$

7.5.2 Implementación en Dafny

Método wrapper del Quick Sort

```
1  method QuicksortWrapper(arr: array<int>)
2    requires arr.Length > 0
3    ensures forall a, b :: 0 <= a <= b < arr.Length ==> arr[a] <= arr[b]
4    ensures multiset(arr[..]) == multiset(old(arr[..]))
5    modifies arr
6  {
7    var N: nat := arr.Length;
8    Quicksort(arr, 0, N - 1);
9  }
```

Método Quick Sort

```
1  method Quicksort(arr: array<int>, from: int, to: int)
2    requires 0 <= from <= arr.Length
3    requires -1 <= to < arr.Length
4    requires from <= to + 1
5    requires from > 0 ==> forall i :: from <= i <= to ==> arr[from - 1] <= arr[i]
6    requires to < arr.Length-1 ==> forall i :: from <= i <= to ==> arr[i] <= arr[to + 1]
7    ensures forall a, b :: from <= a <= b <= to ==> arr[a] <= arr[b]
8    ensures from > 0 ==> forall i :: from <= i <= to ==> arr[from - 1] <= arr[i]
9    ensures to < arr.Length - 1 ==> forall i :: from <= i <= to ==> arr[i] <= arr[to + 1]
10   ensures multiset(arr[..]) == multiset(old(arr[..]))
11   ensures forall k :: 0 <= k < from ==> arr[k] == old(arr[k])
12   ensures forall k :: to < k < arr.Length ==> arr[k] == old(arr[k])
13   decreases to - from
14   modifies arr
15 {
16   if (from < to)
17   {
18     var m: int := Partition(arr, from, to);
19     Quicksort(arr, from, m - 1);
20     Quicksort(arr, m + 1, to);
21   }
22 }
```

Método Partition

```
1  method Partition(arr: array<int>, from: int, to: int) returns (l: int)
2    requires 0 <= from <= to < arr.Length
3    requires from > 0 ==> forall i :: from <= i <= to ==> arr[from - 1] <= arr[i]
4    requires to < arr.Length-1 ==> forall i :: from <= i <= to ==> arr[i] <= arr[to + 1]
5    ensures from <= l <= to
```

```
6     ensures forall k :: from <= k < l ==> arr[k] < arr[l]
7     ensures forall k :: l < k <= to ==> arr[k] >= arr[l]
8     ensures multiset(arr[..]) == multiset(old(arr[..]))
9     ensures forall k :: 0 <= k < from ==> arr[k] == old(arr[k])
10    ensures forall k :: to < k < arr.Length ==> arr[k] == old(arr[k])
11    ensures from > 0 ==> forall i :: from <= i <= to ==> arr[from - 1] <= arr[i]
12    ensures to < arr.Length - 1 ==> forall i :: from <= i <= to ==> arr[i] <= arr[to + 1]
13    modifies arr
14  {
15    l := from;
16    var r: int := to;
17    var pivot: int := arr[from];
18
19    while (l != r)
20      invariant from <= l <= r <= to
21      invariant pivot == arr[l]
22      invariant forall k :: from <= k < l ==> arr[k] < pivot
23      invariant forall k :: r < k <= to ==> arr[k] >= pivot
24      invariant multiset(arr[..]) == multiset(old(arr[..]))
25      invariant forall k :: 0 <= k < from ==> arr[k] == old(arr[k])
26      invariant forall k :: to < k < arr.Length ==> arr[k] == old(arr[k])
27      invariant from > 0 ==> forall i :: from <= i <= to ==> arr[from - 1] <= arr[i]
28      invariant to < arr.Length-1 ==> forall i :: from <= i <= to ==> arr[i] <= arr[to+1]
29      decreases r - l
30      modifies arr
31    {
32      if (pivot <= arr[r])
33      {
34        r := r - 1;
35      }
36      else
37      {
38        arr[l] := arr[r];
39        arr[r] := arr[l + 1];
40
41        arr[l + 1] := pivot;
42        l := l + 1;
43      }
44    }
45  }
```

En el libro *Quicksort Revisited* [21], se exponen diversas variantes del algoritmo *Quick Sort*, comentando posibles especificaciones formales en lenguaje informal.

El método *Partition* que se especifica es muy diferente al nuestro, ya que utilizan un *pivot* por afuera del rango a ordenar. Este comportamiento hace que cambie mucho la lógica del *Quick Sort* también.

En nuestro caso, al realizar un *Partition* a la medida de nuestro *Quick Sort*, las detalladas al final de la metodología del *Partition*, pudimos evitar la utilización de *asserts* en el código *Dafny*.

Por otro lado, la propuesta del *Partition* del *paper* es muy interesante para tenerla en cuenta, porque permite que el usuario utilice un *pivot* dinámicamente, haciendo que el *Partition* quede agnóstico a la implementación del *Quick Sort*.

7.5.3 Implementación en KeY

Método wrapper del Quick Sort

```
1  /*@ public normal_behaviour
2    @  requires array.length > 0;
3    @  ensures (\forall int a, b;
4    @                  0 <= a && a <= b && b < array.length;
5    @                  array[a] <= array[b]);
6    @  ensures \dl_seqPerm(\dl_array2seq(array), \old(\dl_array2seq(array)));
7    @  assignable array[*];
8    @*/
9  public void quicksortWrapper(int[] array) {
10   int N = array.length;
11   quicksort(array, 0, N - 1);
12 }
```

Método Quick Sort

```
1  /*@ public normal_behavior
2    @  requires 0 <= from && from <= array.length;
3    @  requires -1 <= to && to < array.length;
4    @  requires from <= to + 1;
5    @  requires from > 0 ==> (\forall int i;
6    @                                from <= i && i <= to;
7    @                                array[from - 1] <= array[i]);
8    @  requires to < array.length - 1 ==> (\forall int i;
9    @                                          from <= i && i <= to;
10   @                                          array[i] <= array[to + 1]);
11   @  ensures (\forall int a, b;
12   @                  from <= a && a <= b && b <= to;
13   @                  array[a] <= array[b]);
14   @  ensures from > 0 ==> (\forall int i;
15   @                               from <= i && i <= to;
16   @                               array[from - 1] <= array[i]);
17   @  ensures to < array.length - 1 ==> (\forall int i;
18   @                                         from <= i && i <= to;
19   @                                         array[i] <= array[to + 1]);
20   @  ensures \dl_seqPerm(\dl_array2seq(array), \old(\dl_array2seq(array)));
21   @  measured_by to - from + 1;
22   @  assignable array[from..to];
23   @*/
24 private static void quicksort(int[] array, int from, int to) {
25   if (from < to) {
26     int m = partition(array, from, to);
27     quicksort(array, from, m - 1);
28     quicksort(array, m + 1, to);
29   }
30 }
```

Para que el *IDE KeY v2.10.0* pueda demostrar formalmente las post-condiciones establecidas, es necesario ayudar al verificador con un *script*.

Debido a la complejidad del mismo, se solicitó apoyo al soporte de *KeY* a través de correo electrónico y ellos nos proporcionaron su versión del *Quick Sort* con sus *scripts* de verificación.

Con algunos pequeños cambios en el *script* provisto por el soporte de *KeY*, pudimos demostrar en *KeY* el *Quick Sort* nuestro.

```
\withOptions moreSeqRules:on;

\javaSource ".";

\chooseContract "Quicksort[Quicksort::quicksort([I,int,int)].JML normal_behavior
    operation contract.0";

\proofScript "script 'Quicksort.script';"
```

```
macro autopilot-prep;

#
# proof obligation seqPerm after 3 method calls
select formula="{heapAtPre:=heap || exc:=null || heap:=heapAfter_quicksort_0}
   seqPerm(seqDef{int u;}(0, array.length, any::select(heap, array, arr(u))),
           seqDef{int u;}(0, array.length, any::select(heapAtPre, array, arr(u))))";

macro simp-upd;

let @seqPre="seqDef{int u;}(0, array.length, array[u])"
    @seqPartition="seqDef{int u;}(0, array.length, any::select(heapAfter_partition,
     array, arr(u)))"
    @seqQuicksort="seqDef{int u;}(0, array.length, any::select(heapAfter_quicksort,
     array, arr(u)))"
    @seqQuicksort0="seqDef{int u;}(0, array.length, any::select(heapAfter_quicksort_0,
     array, arr(u)))";

rule seqPermSym
  formula="seqPerm(@seqPartition, @seqPre)";

rule seqPermSym
  formula="seqPerm(@seqQuicksort, @seqPartition)";

rule seqPermSym
  formula="seqPerm(@seqQuicksort0, @seqQuicksort)";

rule seqPermTrans
  formula="seqPerm(@seqPre, @seqPartition)";

rule seqPermTrans
  formula="seqPerm(@seqPre, @seqQuicksort)";

rule seqPermSym
  formula="seqPerm(@seqPre, @seqQuicksort0)";

# Now, the power of autopilot is enough.
# Run another 10000 on each open goal.
tryclose;
```

Método Partition

```
/*@ public normal_behavior
  @  requires 0 <= from && from <= to && to < array.length;
  @  requires from > 0 ==> (\forall int i;
  @                                 from <= i && i <= to;
  @                                 array[from - 1] <= array[i]);
```

```
6   @  requires to < array.length - 1 ==> (\forall int i;
7   @                                          from <= i && i <= to;
8   @                                          array[i] <= array[to + 1]);
9   @  ensures from <= \result && \result <= to;
10  @  ensures (\forall int k;
11  @                  from <= k && k < \result;
12  @                  array[k] < array[\result]);
13  @  ensures (\forall int k;
14  @                  \result < k && k <= to;
15  @                  array[k] >= array[\result]);
16  @  ensures \dl_seqPerm(\dl_array2seq(array), \old(\dl_array2seq(array)));
17  @  ensures (\forall int k;
18  @                  0 <= k && k < from;
19  @                  array[k] == \old(array[k]));
20  @  ensures (\forall int k;
21  @                  to < k && k < array.length;
22  @                  array[k] == \old(array[k]));
23  @  ensures from > 0 ==> (\forall int i;
24  @                               from <= i && i <= to;
25  @                               array[from - 1] <= array[i]);
26  @  ensures to < array.length - 1 ==> (\forall int i;
27  @                                          from <= i && i <= to;
28  @                                          array[i] <= array[to + 1]);
29  @  assignable array[from..to];
30  @*/
31 private static int partition(int[] array, int from, int to) {
32
33   int l = from;
34   int r = to;
35   int pivot = array[from];
36
37   /*@  loop_invariant from <= l && l <= r && r <= to;
38     @  loop_invariant pivot == array[l];
39     @  loop_invariant (\forall int k;
40     @                        from <= k && k < l;
41     @                        array[k] < pivot);
42     @  loop_invariant (\forall int k;
43     @                        r < k && k <= to;
44     @                        array[k] >= pivot);
45     @  loop_invariant \dl_seqPerm(\dl_array2seq(array), \old(\dl_array2seq(array)));
46     @  loop_invariant (\forall int k;
47     @                        0 <= k && k < from;
48     @                        array[k] == \old(array[k]));
49     @  loop_invariant (\forall int k;
50     @                        to < k && k < array.length;
51     @                        array[k] == \old(array[k]));
52     @  loop_invariant from > 0 ==> (\forall int i;
53     @                                     from <= i && i <= to;
54     @                                     array[from - 1] <= array[i]);
55     @  loop_invariant to < array.length - 1 ==> (\forall int i;
56     @                                                 from <= i && i <= to;
57     @                                                 array[i] <= array[to + 1]);
58     @  assignable array[from..to];
59     @  decreases r - l;
60     @*/
61   while (l != r) {
62     if (pivot <= array[r]) {
63       r = r - 1;
64
65     } else {
66       array[l] = array[r];
67       array[r] = array[l + 1];
68
69       array[l + 1] = pivot;
70       l = l + 1;
71     }
72   }
```

```
73    return l;
74  }
```

A diferencia del caso anterior, nuestro método *Partition* variaba mucho en comparación al método *Partition* que nos compartió el soporte de *KeY*, por esa razón, pudimos acomodar el *script* provisto para que solo falte un *goal* por demostrar.

El *goal* pendiente, tiene relación a la propiedad de integridad de los elementos, que se sabe que se cumple porque en *Dafny* quedó demostrado automáticamente por el verificador.

Se logran demostrar las propiedades de ordenamiento del método *Partition*.

```
1  \settings {
2    "[Choice]DefaultChoices=moreSeqRules-moreSeqRules:on
3     [Strategy]MaximumNumberOfAutomaticApplications=10000
4     [StrategyProperty]OSS_OPTIONS_KEY=OSS_ON
5     [StrategyProperty]LOOP_OPTIONS_KEY=LOOP_INVARIANT"
6  }
7
8  \javaSource ".";
9
10 \chooseContract "Quicksort[Quicksort::partition([I,int,int)].JML normal_behavior
       operation contract.0";
11
12 \proofScript "script 'Partition.script';"
```

```
1  macro autopilot-prep;
2
3  # Run another 10000 on open goals
4  tryclose;
5
6  # one goal remain open after autopilot
7  # it  have to do with the seqPerm condition
8
9  macro simp-upd;
10 rule seqPermFromSwap;
11 rule andRight;
12   auto;
```

8 Conclusiones

En este capítulo se harán evaluaciones sobre los objetivos establecidos para la tesis.

8.1 Evaluación sobre la metodología

En esta sección se realizará una evaluación de la metodología diseñada en el capítulo 2 y aplicada en todo el cuerpo de la tesis. Se proponen ciertos aspectos que consideramos esenciales para la evaluación:

Base teórica

Se analizaron artículos y materiales de cursos sobre metodologías de programación orientadas a invariantes, que fueron detallados en el capítulo 1, en la sección de estado del arte.

Utilizando dichos conceptos y con la ayuda de nuestro tutor, se realizó una adaptación que derivó en el diseño de la metodología de la tesis.

Sintaxis

Principalmente, la sintaxis tiene que ser accesible de entender y no debería agregar una capa extra de dificultad, sino que debería aportar en clarificar los conceptos y dar un marco de trabajo para la especificación informal.

En nuestro caso, se optó por elegir la sintaxis del lenguaje de programación *Eiffel* [13] como base de la sintaxis de nuestra metodología, porque consideramos que la estructura general del lenguaje es adecuada.

Por otro lado, realizamos varias adaptaciones sobre la base del lenguaje, porque nuestra intención no es que la metodología informal que se detalla en los algoritmos compile. Entonces se realizaron cambios que consideramos que favorecen la lectura y la especificación.

Técnicas para la derivación de invariantes

Se clasifican las invariantes en dos tipos, haciendo que el lector pueda entender cuáles invariantes son esenciales y cuáles son de acotamiento.

Se detallan cuatro técnicas de derivación de invariantes en base a pre y post-condiciones, que son aplicadas en cada uno de los algoritmos propuestos en la tesis, realizando múltiples ejemplos para cada una de las cuatro técnicas.

Se realiza una explicación de la elección de la técnica y cómo se deriva la invariante, con la finalidad de que el lector pueda seguir la linea de razonamiento.

Aplicabilidad

Consideramos que aplicar la metodología para los algoritmos propuestos en la tesis fue muy positivo, ya que sirvió para modelar de manera metodológica y orientada a invariantes las implementaciones.

Programar de manera metodológica y orientado a invariantes, quizás no parece natural al comienzo (desde el punto de vista de la formación académica que hemos tenido) pero, con el tiempo, nos dimos cuenta que para nosotros fue más natural desde el punto de vista lógico. Se aprende a ir acotando el problema y reduciéndolo, de tal forma que la implementación no es necesaria realizarla para entender la naturaleza del algoritmo.

Además, consideramos que la metodología es totalmente agnóstica a la herramienta que se utiliza para probarla formalmente. Inclusive, se podría utilizar sólo como sustitución del pseudo-código y realizar las implementaciones en un lenguaje de programación tradicional.

Como equipo, notamos que de no haber usado la metodología, muchos de los algoritmos hubieran sido implementados de una manera muy diferente. Además, consideramos que la utilización de la metodología, da mayor seguridad y confianza en lo que se está implementando.

8.2 Evaluación sobre KeY y Dafny

A continuación se evalúa y compara *KeY* con *Dafny* en base a la experiencia con los algoritmos realizados en la tesis:

Documentación

Ambas cuentan con un documento principal de referencia para poder aprender el funcionamiento de la herramienta y utilizarlo como base para realizar ejemplos.

En la búsqueda que realizamos para obtener material de información sobre *Dafny* y *KeY*, encontramos más variedad de recursos académicos y ejemplos en *Dafny* que en *KeY*. Encontrar ejemplos en *KeY* fue muy difícil y nos vimos limitados a los que aparecen publicados en el sitio web de la herramienta. [10].

Con respecto a *KeY*, no encontramos suficiente información sobre como escribir los *scripts* [19] o la realización de reglas personalizadas que se utilizan para extender el verificador. Esto es fundamental cuando se realizan algoritmos que son complejos para que el verificador pueda probarlos formalmente. Por otro lado, *Dafny* cuenta con el uso de *lemmas* y *predicados* que ayudan a extender al verificador de una manera sencilla y natural, además de tener documentación de apoyo.

Comunidad

Basados en nuestra experiencia, consideramos que la comunidad de *KeY* es bastante cerrada y reducida. Esto nos dificultó para obtener información adicional sobre la herramienta.

En particular, nuestro equipo contactó al soporte de *KeY* y luego de unas semanas obtuvimos una respuesta aceptable por parte de ellos. Una vez analizada dicha respuesta, el equipo respondió el correo con nuevas consultas que nunca fueron respondidas a pesar de nuestra insistencia.

En el alcance de la tesis, no se tuvo que contactar con el soporte de *Dafny*, ya que se pudo realizar los algoritmos sin mayores inconvenientes. En una tesis anterior de un integrante del equipo [12], existió un contacto con una comunidad de *Dafny* y se obtuvo respuesta satisfactoria.

Sintaxis

Con respecto a la sintaxis de la funcionalidad del algoritmo, *KeY* utiliza el lenguaje *Java* que es muy popular mundialmente. Esta fue la razón principal por la que el equipo optó por la utilización de *KeY* como una de las dos herramientas para la tesis.

Por otro lado, *Dafny* utiliza un lenguaje propio que si bien es bastante similar a otros lenguajes *.NET*, requiere un poco más de tiempo para acostumbrarse a la sintaxis.

Con respecto a la sintaxis de la especificación formal, *KeY* utiliza el estándar *JML* extendido por sintáxis de *JavaDL* que se escribe como comentario incrustado en el código *Java*.

Dafny al utilizar un lenguaje propio, agrega como parte de la sintaxis de su lenguaje las instrucciones de especificación formal que se utilizan.

Verificador

KeY al utilizar comentarios sobre código Java, no puede realizar una verificación en tiempo real sobre las especificaciones que se van realizando. Por otro lado, *Dafny* puede ir compilando el código en tiempo real y marcando exactamente dónde hay problemas de verificación.

Cuando se realizan errores humanos en la verificación de algoritmos, el verificador de *Dafny* marca automáticamente en tiempo real dónde está el error.

Por otro lado, *KeY* al utilizar comentarios sobre código *Java* para la especificación formal, no puede marcar el error en tiempo real y se debe hacer a través del *IDE KeY v2.10.0*.

El *IDE* de *KeY* no nos fue sencillo de comprender y muestra *goals* pendientes sin indicar claramente dónde está el error, haciendo que se pierda mucho tiempo a la hora de intentar probar formalmente los algoritmos.

Como equipo, cuando nos estancábamos con alguna implementación en *KeY*, terminábamos haciéndola en *Dafny* para entender dónde estaba el error.

Valor didáctico

A pesar que *KeY* utiliza *Java* como base de las implementaciones y esto podría suponer una ventaja para el estudiante, ya que podría enfocarse directamente en aprender la especificación formal en *JML*, sin embargo, según nuestra experiencia, consideramos que la complejidad del verificador genera que la curva de aprendizaje sea demasiado grande y engorrosa.

Por otro lado, si bien *Dafny* utiliza una sintaxis nueva para los estudiantes, consideramos que el verificador es muy comprensible y facilita la implementación de los algoritmos (por lo comentado en el punto anterior).

Pensamos que es importante brindar la opción al estudiante de poder aplicar metodología de lógica de programación de manera formal, para modelar su pensamiento orientado a invariantes y expandir su conocimiento en esta área poco explorada. De las herramientas que incluimos en esta investigación, la que recomendamos utilizar para esto es *Dafny*.

8.3 Aplicabilidad didáctica

Metodología informal

Consideramos importante la introducción a alguna metodología informal de diseño de algoritmos en una etapa inicial de la carrera. Creemos que potenciaría el pensamiento lógico de los estudiantes y los ayudaría a modelar su pensamiento a la hora de resolver un problema algorítmico.

Por la naturaleza de los cursos de la carrera, sentimos que "Estructura de Datos y Algoritmos 2" es el lugar adecuado para introducir el concepto de metodología informal. A esa altura de la carrera, el estudiante tiene una base de programación suficiente para poder complementarla con una abstracción metodológica informal como la que se detalla en esta tesis.

Principalmente, consideramos que sería conveniente poder introducir el concepto de invariantes con el propósito de que los alumnos reflexionen las soluciones a los algoritmos propuestos en el curso, orientado a dicho concepto en vez de utilizar el tradicional pseudo-código. Cabe aclarar, que consideramos fundamental un apoyo gráfico que acompañe la metodología informal propuesta, ya que sería un concepto totalmente nuevo para los estudiantes.

Por otro lado, la aplicabilidad de una herramienta como lo es *KeY*

y *Dafny* en este curso en particular, no lo vemos viable, ya que sería realmente muy pesado aprender tanto la metodología informal como la formal, en un curso con tanto contenido algorítmico.

KeY como herramienta didáctica

Desde nuestra experiencia, consideramos que *KeY* es una herramienta con una curva de aprendizaje muy alta y con documentación insuficiente para aprender de ella. Además, por lo comentado anteriormente, no nos fue sencillo tener una comunicación estable con la comunidad de *KeY*.

Consideramos que estos aspectos hacen que sea muy poco recomendable utilizarla para un curso, inclusive siendo un curso electivo avanzado en la carrera. Sentimos que la complejidad que tiene el aprendizaje de la herramienta haría que el curso fuera realmente muy engorroso para los estudiantes.

Dafny como herramienta didáctica

En Universidad ORT Uruguay existe una materia electiva para alumnos avanzados en la carrera que introduce el concepto de metodología informal y además incluye la realización de diversas implementaciones de algoritmos utilizando *Dafny* como herramienta. Esta materia es denominada "Lógica de la programación" y dictada por el Dr. Álvaro Tasistro.

A continuación, se detallan aspectos que consideramos relevantes sobre *Dafny*, que creemos que hacen que sea una herramienta atractiva para una electiva de Lógica de la programación:

1. Consideramos que la curva de aprendizaje es accesible, si se trata de un estudiante avanzado de la carrera. El alcance del curso sería de un semestre y el estudiante podría aprender y utilizar la herramienta en diversos algoritmos conocidos.

2. Nos fue bastante sencillo acceder a una amplia documentación de *Dafny*, lo cual consideramos fundamental para poder extraerla en material para el curso.

3. Nuestra experiencia nos determina que *Dafny* tiene una comunidad abierta y accesible, esto es fundamental cuando los estudiantes busquen información por su cuenta o quieran profundizar su conocimiento en alguna temática.

4. Se puede utilizar como *IDE Visual Studio Code*, que es una herramienta muy utilizada y gratuita para el desarrollo en *Dafny*.

Esta herramienta cuenta con una extensión de *Dafny*, que es actualizada frecuentemente y compila en tiempo real el código *Dafny* escrito, subrayando en rojo las especificaciones que no pueden ser probadas. Por otro lado, si la especificación se puede probar la marcará con verde.

9 Referencias bibliográficas

[1] D. Zingaro, *Invariants: A Generative Approach to Programming*, ser. Texts in computer science. College Publications, 2008. [Online]. Available: https://books.google.com.uy/books?id=YEKzOAAACAAJ

[2] C. A. Furia, B. Meyer, and S. Velder, "Loop invariants: Analysis, classification, and examples," *ACM Comput. Surv.*, vol. 46, no. 3, jan 2014. [Online]. Available: https://doi.org/10.1145/2506375

[3] C. Morgan, "(in-)formal methods: The lost art," in *Engineering Trustworthy Software Systems*, Z. Liu and Z. Zhang, Eds. Cham: Springer International Publishing, 2016, pp. 1–79.

[4] M. Sozeau, "The coq proof assistant," Inria, accedido el 9 de septiembre, 2022, desde https://coq.inria.fr/.

[5] Toccata, "Why3," Inria Saclay-Ile-de-France, accedido el 9 de septiembre, 2022, desde https://why3.lri.fr/.

[6] L. Paulson, "Isabelle/hol," Universidad de Cambridge and Universidad de Munich, accedido el 9 de septiembre, 2022, desde https://isabelle.in.tum.de/.

[7] M. Alberti and T. Antignac, "Frama-c," CEA LIST and Inria, accedido el 9 de septiembre, 2022, desde https://frama-c.com/.

[8] P. de Halleux, S. Rajamani, T. Ball, and T. Hoare, "Slam," Microsoft Research, accedido el 9 de septiembre, 2022, desde https://www.microsoft.com/en-us/research/project/slam/.

[9] A. Bradley and Z. Manna, "The pi verifying compiler," Stanford, accedido el 9 de septiembre, 2022, desde https://theory.stanford.edu/~arbrad/pivc/index.html.

[10] T. K. Project, "The key project," The Key Project, accedido el 7 de noviembre, 2021, desde https://www.key-project.org/.

[11] K. R. M. Leino, “Dafny,” Dafny, accedido el 9 de septiembre, 2022, desde https://dafny.org/.

[12] M. S. Hernández Lorenzo, *Lógica de programación con Dafny*. Montevideo: Universidad ORT Uruguay, 2021. [Online]. Available: https://sisbibliotecas.ort.edu.uy/cgi-bin/koha/opac-retrieve-file.pl?id=7b293247a6c49ffd72c9e68bf6dd9ae4

[13] E. Organization, “Eiffel,” Eiffel, accedido el 9 de septiembre, 2022, desde https://eiffel.org/.

[14] M. Fokkinga, “Tail invariants,” 1994. [Online]. Available: https://maartenfokkinga.github.io/utwente/mmf94g.pdf

[15] K. R. M. Leino, R. L. Ford, and D. R. Cok, *Dafny Reference Manual*, Microsoft Research, abril 2021. [Online]. Available: https://homepage.cs.uiowa.edu/~tinelli/classes/181/Papers/dafny-reference.pdf

[16] K. R. M. Leino, “Dafny github,” accedido el 10 de septiembre, 2021, desde https://github.com/dafny-lang/dafny.

[17] W. Ahrendt, B. Beckert, R. Bubel, R. Hähnle, P. H. Schmitt, and M. Ulbrich, *Deductive Software Verification - The KeY Book*, The KeY Project, 2016. [Online]. Available: http://www.key-project.org/thebook2/

[18] G. T. Leavens and Y. Cheon, *Design by Contract with JML*, Iowa State University and University of Texas at El Paso, septiembre 2006. [Online]. Available: https://www.cs.ucf.edu/~leavens/JML/jmldbc.pdf

[19] T. K. Project, “Key developer documentation,” The Key Project, accedido el 10 de septiembre, 2021, desde https://www.key-project.org/docs/workbench/.

[20] G. McDowell, *Cracking the Coding Interview: 189 Programming Questions and Solutions*. CareerCup, LLC, 2015. [Online]. Available: https://books.google.com.uy/books?id=jD8iswEACAAJ

[21] R. Certezeanu, S. Drossopoulou, B. Egelund-Muller, K. R. M. Leino, S. Sivarajan, and M. Wheelhouse, *Quicksort Revisited*, 03 2016, vol. 9660, pp. 407–426. [Online]. Available: https://doi.org/10.1007/978-3-319-30734-3_27