Especificación de algoritmos

Isabel Pita

Facultad de Informática - UCM

28 de septiembre de 2017

Índice

Bibliografía Recomendada

- Algoritmos correctos y eficientes: Diseño razonado ilustrado con ejercicios. *Matí-Oliet, N.; Segura Diaz, C. M., Verdejo Lopez, A.*. Ibergarceta Publicaciones, 2012.
- Especificación, Derivación y Análisis de Algoritmos: ejercicios resueltos. *Narciso Martí Oliet, Clara María Segura Díaz y Jose Alberto Verdejo López*. Colección Prentice Práctica, Pearson Prentice-Hall, 2006

Capítulo 1 de ambos libros.

- Ejercicios resueltos: todos menos el 1.5, 1.6, 1.7
- Ejercicios propuestos: todos menos el 1.2.

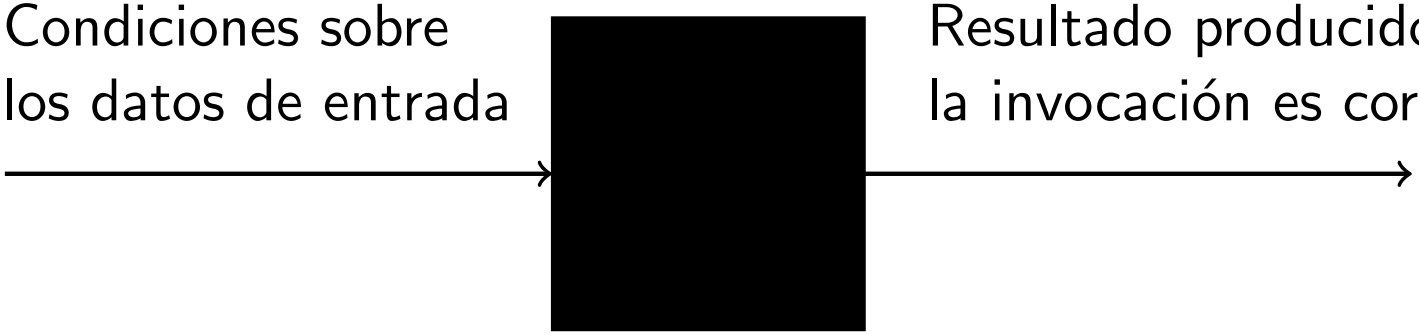
Objetivos

- Diferencia entre especificar e implementar
- Partes de una especificación formal
- Especificar algoritmos en lenguaje Dafny
  - Definición de predicados
  - Definición de funciones

- Especificar un algoritmo.

qué hace el algoritmo.

Precondición
Condiciones sobre los datos de entrada



Postcondición
Resultado producido si la invocación es correcta.

- Implementar,

cómo *se consigue la funcionalidad pretendida*.

Especificación 1: Dado un vector de números enteros a ordenado de menor a mayor y un número entero x, devolver un valor booleano b que indique si alguno de los elementos del vector $a[0], \ldots, a[n-1]$ es igual al valor dado x.

Implementación: Algoritmo de búsqueda secuencial o algoritmo de búsqueda binaria.

Especificación 2: Ordenar de menor a mayor un vector de números enteros a.

Implementación: Ordenación rápida (quicksort), ordenación por mezclas (mergesort), ordenación por inserción (insertion sort)...

Usos de una especificación:

- Los **usuarios** del algoritmo:
  la especificación debe contener la información necesaria para utilizar el algoritmo.
- El **implementador** del algoritmo:
  - la especificación debe definir los requisitos que cualquier implementación válida debe satisfacer.
  - ha de dejar suficiente **libertad** para que se elija la implementación que se estime más adecuada con los recursos disponibles.

Especificación. Propiedades. Tipos.

Propiedades de una buena especificación:

Precisión ha de responder a cualquier pregunta sobre el uso del algoritmo sin tener que acudir a la implementación del mismo. El lenguaje natural no permite la precisión requerida si no es sacrificando la brevedad.

Brevedad ha de ser significativamente más breve que el código que especifica.

Claridad ha de transmitir la intuición de lo que se pretende.

Los lenguajes formales ofrecen a la vez precisión y brevedad. A veces deben ser complementados con explicaciones informales para obtener claridad.

Ventajas de una **especificación formal** frente a una informal:

- Elimina ambigüedades que el usuario y el implementador podrían resolver de formas distintas, dando lugar a errores de uso o de implementación que aparecerían en ejecución.
- Permiten realizar una verificación/derivación formal del algoritmo. Este tema se tratará en los capítulos **3** y **4**, y consiste en razonar sobre la corrección del algoritmo mediante el uso de la **lógica**.
- Permite generar automáticamente un conjunto de casos de prueba que permitirán probar la corrección de la implementación. La especificación formal se usa para **predecir** y **comprobar** el resultado esperado.

Especificación. Propiedades. Tipos.

Especificación informal: dado un vector a de componentes enteras, devolver un valor booleano b que indique si existe algún elemento cuyo valor es igual a la suma de todos los elementos que le preceden en el vector.
Ambigüedades:

- No quedan claras todas las obligaciones del usuario: (1) ¿serían admisibles llamadas con vectores vacíos?; (2) ¿y con un elemento?
- En caso afirmativo, ¿cuál será el valor devuelto por la función?
- Tampoco están claras las obligaciones del implementador. Por ejemplo, si el vector tiene mas de un elemento y $a[0] = 0$ la función, ¿ha de devolver **true** o **false**?

Tratar de explicar en lenguaje natural todas estas situaciones llevaría a una especificación más extensa que el propio código de la función.

Especificación. Propiedades. Tipos.

Este curso escribiremos las especificaciones formales en el lenguaje Dafny

Especificación formal:

```
method esSuma (a : array<int>) returns (b : bool)
  // Precondicion
  requires a != null;
  //Poscondicion
  ensures b == exists w :: 0 <= w < a.Length &&
                           a[w] == Sum(a[..w]);
```

No es ambigua:

1. No son correctas llamadas con vectores sin memoria reservada;
2. sí son correctas llamadas con vectores vacíos, y en ese caso ha de devolver $b = \mathbf{false}$.
3. las llamadas con vectores cuya primera componente es cero, $a[0] = 0$ han de devolver $b = \mathbf{true}$.

Predicados

- La precondición y la postcondición de una especificación formal se expresan mediante predicados lógicos.
- Un predicado es una expresión de tipo **bool**.
- Definición de predicados en Dafny:

  Ejemplos:

```
n < 0
n <= x
n > 2 * s
n1 + n2 >= n3 + n4
n == m (doble signo igual)
0 <= i < n
```

Predicados

Si `P` y `Q` son predicados, también lo son:

1. `!P`, (no `P`). Ejemplo: `!(n < 0)`.
2. `P && Q`, (`P` y `Q`). Ejemplo: `0 <= n && x > 1000`.
3. `P || Q`, (`P` o `Q`). Ejemplo: `0 <= n || x > 1000`.
4. `P ==> Q`, (`P` entonces `Q`) (escrito con dos signos igual)
5. Cuantificación universal: `forall w:: Q(w) ==> P(w)` , (para todo valor perteneciente al rango `Q` se cumple `P`).
   Sobre el rango vacío su valor es cierto
   Ejemplo:
   ```
   forall w :: 0 <= w < a.Length ==> a[w] > 0.
   ```
6. Cuantificación existencial: `exists w:: Q(w) && P(w)`, (existe un valor perteneciente al rango `Q` para el que se cumple `P`) .
   Sobre el rango vacío su valor es falso.
   Ejemplo:
   ```
   exists w :: 0 <= w < a.Length && a[w] > 0.
   ```

Predicados

- Es **muy importante** distinguir los dos tipos de variables que pueden aparecer en un predicado con cuantificadores:

  Variables ligadas Son las que están afectadas por un cuantificador, es decir se encuentran dentro de su ámbito.

  Variables libres Son las variables que no se encuentran afectadas por ningún cuantificador. En una especificación se utilizan para describir variables del programa.

- Por ejemplo en el predicado
  `forall w :: 0 <= w < a.Length ==> a[w] > 0`,
  - a es variable **libre**,
  - w es variable **ligada**.

Predicados

En general:

- Cuando se anidan cuantificadores las variables están ligadas al cuantificador más interno.
- Una variable ligada se puede **renombrar** de forma consistente sin cambiar el significado del predicado.
  Por ejemplo:

```
forall w :: 0 <= w < a.Length ==> a[w] > 0
```

$$\equiv$$

```
forall i :: 0 <= i < a.Length ==> a[i] > 0
```

Para evitar errores al escribir predicados utilizad identificadores diferentes para las variables ligadas. Tambien conviene utilizar identificadores diferentes para las variables libres de los usados en las variables ligadas.

Especificaciones

Los predicados nos permiten definir los valores que toman las variables/parámetros del programa.

- Cabecera de la función.

```
method raizEntera(n : int) returns (r : int)
```

  1. Los parámetros de entrada o entrada/salida se muestran entre paréntesis separados por comas.
  2. Los valores de salida se muestran después de la cláusula `returns`, entre paréntesis y separados por comas. Se debe dar nombre a todos los valores de salida.

- Precondición. Los requisitos sobre los datos de entrada se escriben en la cláusula `requires`.

```
requires n >= 0
```

- Poscondición. Los valores de las variables de salida se describen en la cláusula `ensures` .

```
ensures r >= 0 && r*r <= n <(r+1)*(r+1)
```

Especificaciones

- Comprobar que un vector esta formado por valores positivos

```
method positivo(v : array<int>) returns (r : bool)
  requires v != null
  ensures
    r == forall k::0<=k<v.Length ==> v[k]>0
```

- Todos los valores entre la posición `a` (incluida) y `b` (excluida) son pares.

```
method pares(v : array<int>, a : int, b : int)
              returns (r : bool)
  requires v != null
  requires forall k::0<= k<v.Length ==> v[k]>0
  requires 0<=a<=b<= v.Length
  ensures r == forall k::a<=k<b ==> v[k] %2==0
```

Especificaciones

- Si existe un valor negativo en el vector, `s` es la posición del primer valor negativo desde la izquierda del vector.

```
method existeNeg(v : array<int>)
                 returns (r : bool, s : int)
  requires v != null
  ensures r == exists k::0<=k<v.Length && v[k]<0
  ensures r ==> 0<=s< v.Length && v[s] < 0
  ensures r ==> forall j::0<=j<s ==> v[j]>=0
```

Otras facilidades de Dafny.

- Dar nombre a un predicado:

```
predicate negativo (n : int)
{ n < 0}
```

- Uso de secuencias en la definición de predicados:

```
predicate esta (a : seq<int>, x : int)
{ exists u :: 0 <= u < |a| && a[u] == x }

predicate seqNegativa(a : seq<int>)
{ forall u :: 0 <= u < |a| ==> negativo(a[u]) }
```

- Imponer restricciones sobre los datos de entrada:

```
predicate iguales (a : seq<int>, b: seq<int>)
requires |a| == |b|
{ forall u :: 0 <= u < |a| ==> a[u] == b[u] }
```

Definición de funciones para usarlas en predicados.

- Suma de los valores de una secuencia de enteros.

```
function Sum(s : seq<int>) : int
ensures s == [] ==> Sum(s) == 0
ensures s != [] ==> Sum(s) == s[0] + Sum(s[1..])
{ if s == [] then 0 else s[0] + Sum(s[1..]) }
```

  - Especificar un algoritmo que calcule la suma de los elementos de un vector

```
method sumaVector (v:array<int>) returns (s:int)
 requires v != null
 ensures s == Sum(v[..])
```

  - La secuencia formada por los valores del vector `v` se representa como `v[..]`.
  - La secuencia formada por los valores del vector `v` entre las posiciones `a` (incluida) y `b` (excluida) se representa como `v[a..b]`.

Definición de funciones para usarlas en predicados.

- Suma de los valores de una secuencia que cumplen una propiedad.

```
function SumP1Elem(s:seq<int>, p:(int)->bool):int
  reads p.reads
  requires forall k :: 0 <= k <|s| ==> p.requires(s[k])
  ensures s==[] ==> SumP1Elem(s,p)==0
  ensures s!=[] && p(s[0]) ==>
          SumP1Elem(s,p)==s[0]+SumP1Elem(s[1..], p)
  ensures s!=[] && !p(s[0])==>
          SumP1Elem(s,p)==SumP1Elem(s[1..],  p)
{
  if s == [] then 0
  else if p(s[0]) then s[0] +SumP1Elem(s[1..], p)
  else SumP1Elem(s[1..], p)
}
```

- En el nombre de la función `P1` indica que la propiedad expresada tiene un argumento y `Elem` indica que la propiedad se expresa sobre los elementos de la secuencia.

Definición de funciones para usarlas en predicados.

- Especificar un algoritmo que calcule la suma de los elementos negativos de un vector.

```
method sumaVector (v:array<int>) returns (s:int)
requires v != null
ensures s == SumP1Elem(v[..],negativo)
```

donde `negativo` es el predicado definido anteriormente.

Definición de funciones para usarlas en predicados.

- Número de valores de una secuencia de enteros que cumplen una propiedad.

```
function Count(s:seq<int>, p:(int) -> bool):nat
  reads p.reads
  requires forall k :: 0 <= k <|s| ==> p.requires(s[k])
  ensures s == [] ==> Count(s,p) == 0
  ensures s != [] && p(s[0]) ==>
       Count(s,p) == 1 + Count(s[1..], p)
  ensures s != [] && !p(s[0]) ==>
       Count(s,p) == Count(s[1..],p)
{
  if s == [] then 0
  else if p(s[0]) then 1 + Count(s[1..], p)
  else Count(s[1..], p)
}
```

- Sobre la secuencia vacía su valor es cero.

Definición de funciones para usarlas en predicados.

- Número de valores de una secuencia de enteros que cumplen una propiedad definida sobre la secuencia completa.

```
function CountP1Sec(s:seq<int>, i:int,
     p:(seq<int>, int) -> bool) : nat
  reads p.reads
  requires 0 <= i <= |s|
  requires forall k :: 0 <= k <|s| ==> p.requires(s,k)
  ensures i == 0 ==> CountP1Sec(s,i,p) == 0
  ensures i > 0 && p(s,i-1) ==>
    CountP1Sec(s,i,p) == 1 + CountP1Sec(s, i-1, p)
  ensures i > 0 && !p(s,i-1) ==>
    CountP1Sec(s,i,p) == CountP1Sec(s, i-1, p)
{
  if i == 0 then 0
  else if p(s,i-1) then 1 + CountP1Sec(s, i-1, p)
  else CountP1Sec(s, i-1, p)
}
```

Ejemplos de especificaciones

- Especificar un algoritmo que calcule el cociente por defecto y el resto de la división de naturales. Primer intento:

```
method divide (a : int, b : int)
                        returns (q : int, r : int)
requires a >= 0 && b > 0
ensures a == q * b + r
```

- El especificador ha de imaginar que el implementador es un ser **malévolo** que trata de satisfacer la especificación del modo más simple posible, respetando la "letra" pero no el "espíritu" de la especificación.

Ejemplos de especificaciones

- El siguiente programa es correcto respecto a la especificación dada: `q := 0; r := a;`.
  El problema es que la postcondición es demasiado **débil**.
  Existen valores de las variables que satisfacen la postcondición, pero que no son resultados correctos del programa. Debemos restringir los valores posibles. Para ello añadimos condiciones al predicado.
- Segundo intento:

```
method divide (a : int, b : int)
                              returns (q : int, r : int)
requires a >= 0 && b > 0
ensures a == q * b + r && 0 <= r < b
```

  Por conocimientos elementales de matemáticas sabemos que sólo existen dos números naturales que satisfacen lo que exigimos a q y r.

Ejemplos de especificaciones

- Devolver un número primo mayor o igual que un cierto valor:

```
predicate primo (n : int)
 requires n > 1
 {forall w :: 1 < w < n ==> n % w != 0}

method unPrimo (n : int) returns (p : int)
requires n > 1
ensures p >= n && primo(p)
```

  - ¿Sería correcto devolver $p = 2$?
  - Nótese que la postcondición no determina un único `p`. El implementador tiene la libertad de devolver cualquier primo mayor o igual que `n`.

Ejemplos de especificaciones

- Devolver el **menor** número primo mayor o igual que un cierto valor:

```
method menorPrimo (n : int) returns (p : int)
requires n > 1
ensures p >= n && primo(p)
ensures forall w :: n <= w && primo(w) ==> p <= w
```

- Nótese que un predicado **no** es una implementación. Por tanto no le son aplicables criterios de eficiencia: $primo(x)$ es una propiedad y **no** sugiere que la comprobación haya de hacerse dividiendo por todos los números menores que x.

Ejemplos de especificaciones

- Calcular la posición de un vector en la que se encuentra el máximo de los valores del vector:

```
method posMax (v : array<int>) returns (p : int)
requires v != null && v.Length > 0
ensures 0 <= p < v.Length
ensures forall k::0<=k<v.Length ==> v[k]<=v[p]
```

- Calcular el máximo de los elementos de un vector:

```
method maximo (v : array<int>) returns (m : int)
requires v != null && v.Length > 0
ensures forall k :: 0<=k<v.Length ==> v[k]<=m
ensures exists k :: 0<=k<v.Length && v[k]==m
```

- Existen predicados semejantes para la definición del elemento mínimo.

Ejemplos de especificaciones

- Especificar un procedimiento que *positiviza* un vector. Ello consiste en reemplazar los valores negativos por ceros.
  Primer intento:

```
method positivizar(v : array<int>)
  requires v != null
  ensures forall i :: 0 <= i < v.Length && v[i] < 0
                                        ==>   v[i] == 0
  modifies v
```

- Donde lo que conseguimos es una postcondición equivalente a **false**, ya que si un valor es menor que cero no puede ser cero.
- PROBLEMA: En la postcondición `v[i] < 0` se refiere al vector al comienzo de la función, mientras que en `v[i] == 0` nos referimos al vector al finalizar la función.
- Hay que indicar a que vector nos referimos. `old(v[i])`.
- Observad que el parámetro `v` es de entrada/salida, para indicarlo se utiliza la clausula `modifies`.

Ejemplos de especificaciones

- Segundo intento:

```
method positivizar(v : array<int>)
  requires v != null
  ensures v.Length == old(v.Length)
  ensures forall i :: 0 <= i < old(v.Length) &&
                        old(v[i]) < 0 ==> v[i] == 0
  modifies v
```

- El implementador malévolo podría modificar también el resto de valores o añadir nuevos valores al final del vector.
- Tercer intento:

```
method positivizar(v : array<int>)
  requires v != null
  ensures v.Length == old(v.Length)
  ensures forall i :: 0 <= i < old(v.Length) &&
    old(v[i]) < 0 ==> v[i] ==  0
  ensures forall i :: 0 <= i < old(v.Length) &&
    old(v[i]) >= 0 ==> v[i] == old(v[i])
  modifies v
```

Problema número 399

Las perlas de la condesa

Tiempo máximo: 1,000-3,000 s Memoria máxima: 4096 KiB

http://www.aceptaelreto.com/problem/statement.php?id=399

El collar de perlas de la condesa es conocido por la calidad y la cantidad de sus perlas. Lo luce en las grandes fiestas, dándole tres o cuatro vueltas alrededor de su cuello. La perla del centro es la de mayor tamaño, y según nos desplazamos hacia los extremos son estrictamente más pequeñas. Las perlas están perfectamente seleccionadas y colocadas, lo que hace al collar completamente simétrico respecto a la perla central.

Durante el último baile, se ha roto el cierre y las perlas han rodado por el suelo. Los invitados han intentado recuperarlas todas ensartándolas en un nuevo cordel según las encontraban. Al día siguiente, la condesa ha llamado a su joyero para que las engarce de nuevo en el orden adecuado. Éste mide el diámetro de cada una de ellas y procede a rehacer el collar colocando la perla central y a continuación las siguientes en tamaño hasta llegar a los extremos. A la condesa no le importa si se ha perdido alguna perla, siempre y cuando el collar siga siendo completamente simétrico.

Entrada

Cada línea de la entrada forma un caso de prueba, que consiste en una lista de números positivos separados por espacios. Cada uno representa el diámetro de una de las perlas en el orden en el que las fueron recogiendo los invitados. La perfección del collar es legendaria, por lo que el diámetro se mide con una unidad de medida que muchos consideran infinitesimal, aunque el diámetro será siempre menor que 2^{31}. Cada lista tiene $0 < perlas < 1.000$ y acaba siempre con un cero.

El último caso de prueba, que no deberá procesarse, contiene un collar vacío, representado por una lista con un único cero.

Salida

Para cada caso de prueba, el programa escribirá "`NO`" si no es posible formar un collar simétrico con todas y cada una de las perlas encontradas de forma que la perla de mayor tamaño quede en el centro. En otro caso, se escribirán, separados por espacios, los diámetros de las perlas tal y como han quedado ordenadas en el collar, desde un extremo al otro.

Obtener una especificación a partir de un problema.

Especificamos un función que devuelva cierto si se puede formar el collar y falso en caso contrario.

```
predicate igual(a : seq<int>, n : int, v : int)
 requires |a| > 0 && 0 <= n < |a|
 { a[n] == v}

method perlas (a : array<int>) returns (r : bool)
requires a != null
requires forall k :: 0 <= k < a.Length ==>
         CountPSec2(a[..],a.Length,a[k],igual) <= 2
ensures  r ==
 (forall k :: 0 <= k < a.Length ==>
   (exists k1::0< k1< a.Length&&k1!=k&&a[k1]==a[k]||
    forall k2::0<=k2<a.Length ==> a[k]>a[k2])) &&
 (exists j :: 0 <= j  < a.Length &&
    forall k3 :: 0 <= k3 < a.Length ==> a[j] > a[k3])
```

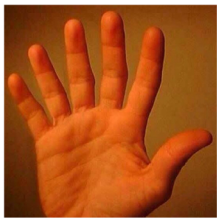
donde la función `CountPSec2` cuenta el número de elementos de la secuencia `a[..]` que son iguales al elemento `a[k]`.

Problema número 346

El hombre con seis dedos

Tiempo máximo: 1,000-2,000 s Memoria máxima: 4096 KiB

http://www.aceptaelreto.com/problem/statement.php?id=346



El otro día me presentaron a un hombre y al darnos la mano noté algo extraño. ¡Luego me di cuenta de que tenía seis dedos! La cosa me sorprendió tanto que he estado investigando y resulta que no es tan extraño que una persona nazca con más de cinco dedos en una mano o un pie. La anomalía genética se denomina *polidactilia* y se estima que 1 de cada 500 bebés nace con ella. Lo que ocurre es que normalmente el dedo extra es muy pequeño y termina siendo extirpado sin más consecuencias. Lo que sí es más extraño es que el dedo se encuentre completo y sea funcional. Eso es lo que le ocurría a este hombre.

Continuando mi investigación he dado con una página web donde se han recopilado los años donde se conoce que nació una persona con algún dedo de más, completo y funcional. Y estoy interesado en construir un programa para averiguar en qué siglo nacieron más personas así. Ya puestos, sería mejor tener un programa más general, que sirviera también para averiguar la decada con más nacimientos, o el lustro, etc.

Entrada

El programa recibirá por la entrada estándar una serie de casos de prueba. Cada caso consta de dos líneas. En la primera aparecen dos números: el número N de nacimientos de los que tenemos constancia (entre 1 y 200.000) y el número A de años del periodo en el que estamos interesados (un número mayor que cero). En la segunda aparecen, separados por espacios y ordenados cronológicamente, los N años de nacimiento (si en un mismo año nació más de una persona, el año aparecerá repetido).

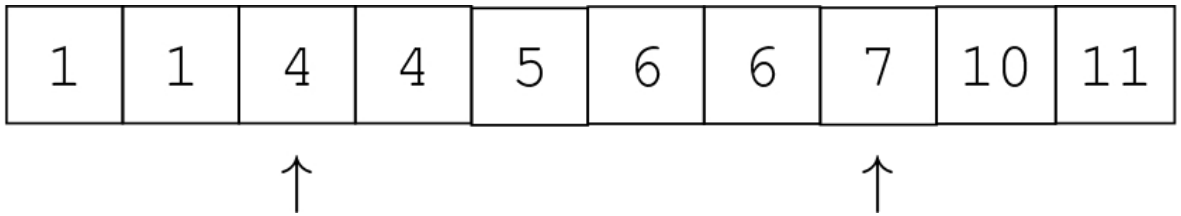
Detrás del último caso aparece una línea con dos ceros.

Salida

Para cada caso de prueba el programa escribirá una línea con el mayor número de personas que ha nacido en un periodo de A años. Ten en cuenta que si un periodo de A años comienza en el año D entonces abarca hasta el año $D + A - 1$, inclusive.

Obtener una especificación a partir de un problema.

Observamos que nos piden el número de elementos del mayor subvector que cumpla que la diferencia entre el valor del último elemento del subvector y el primero sea menor que p. Por ejemplo, en el siguiente vector el subvector de longitud máxima que cumple los requisitos para $p = 5$ es el comprendido entre las dos flechas y su longitud es $r = 6$.



Obtener una especificación a partir de un problema.

Especificación:

```
predicate prop(a : seq<int>, i : int, j : int,p : int)
 requires |a| > 0 && 0 <= i <= j < |a|
 { a[j] - a[i] < p}

method dedos (a : array<int>, p : int) returns (r : int)
requires a != null && p > 0 && a.Length > 0
requires forall k1,k2::0<=k1<k2<a.Length==>a[k1]<=a[k2]
ensures forall i,j::0<=i<=j<a.Length && a[j]-a[i]<p
        ==> r>=j-i+1
ensures exists i,j::0<=i<=j<a.Length && a[j]-a[i]<p
        ==> r == j-i+1
```