Cuaderno de problemas
Fundamentos de algorítmia.

Algoritmos iterativos

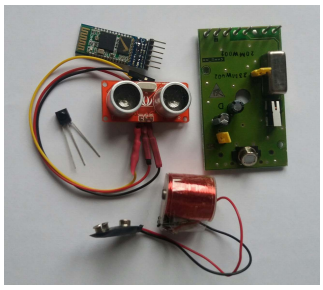
Prof. Isabel Pita

26 de noviembre de 2019

Índice

1. Desplazar elementos de un vector. Sensores defectuosos.

Sensores defectuosos



Ayer compré unos sensores a un precio increible, sin embargo no funcionan tan bien como yo esperaba. Algunas veces el valor que transmiten es absurdo. He intentado devolverlos, pero me dicen que las ofertas no admiten cambios. Después de mucho estudiar los datos que se producen me he dado cuenta que cuando falla el sensor siempre devuelve el mismo valor erroneo. Siendo así, todavía puedo aprovechar los sensores. Lo único que tengo que hacer es eliminar este valor y quedarme con el resto de los datos.

He hecho un programa que elimina todos los valores erróneos, uno por uno. !!Pero que lento es!!. Al comentárselo a mi hermano me ha explicado una forma de implementar la función mucho más eficiente. Aunque debo de tener cuidado de no variar el orden relativo entre los datos correctos.

Requisitos de implementación.

Debe implementarse una función que reciba un vector con todos los datos tomados por el sensor y el valor del dato erróneo, y lo modifique quitándole todos los datos erróneos.

El coste de la función debe ser del orden del número de datos de entrada.

Entrada

La entrada comienza con un valor entero que indica el número de casos de prueba. Cada caso de prueba consta de dos líneas. En la primera se indica el número de medidas tomadas y el valor erróneo. En la segunda se muestran todos los valores tomados por el sensor.

Las medidas tomadas se encuentran en el rango de valores (-2^{63}... 2^{63}).

Salida

Para cada caso de prueba se escriben dos líneas. En la primera se muestra el número de datos correctos tomados por el sensor, en la segunda los valores correctos.

Entrada de ejemplo

```
4
8 -1
5 -1 -1 10 4 -1 10 7
3 -1
3 5 7
4 0
0 0 0 0
7 0
0 10 -23 0 -12 67 0
```

Salida de ejemplo

```
5
5 10 4 10 7
3
3 5 7
0

4
10 -23 -12 67
```

1.1. Objetivos del problema

- Aprender a desplazar varios elementos de un vector recorriendo el vector una única vez.
- Recordar los modificadores de tipo de C++.

1.2. Ideas generales.

- En el problema nos piden eliminar todos los elementos de un vector que cumplen una determinada propiedad. En este caso la propiedad pedida es coincidir con el valor de un parámetro de entrada. El problema se debe resolver recorriendo el vector una única vez y sin utilizar un vector auxiliar.
- Si desplazamos todas las componentes a la derecha del vector por cada elementos que se elimina, el coste de la solución en el caso peor es cuadrático respecto al número de elementos del vector.
- Si se utiliza la función `erase` de la clase `vector` para eliminar los elementos que no cumplen la propiedad, el coste en el caso peor es cuadrático respecto al número de elementos del vector, ya que esta función `erase` tiene coste lineal respecto al número de elementos desplazados.
- Para resolver el problema en tiempo lineal respecto al número de elementos del vector y sin utilizar memoria auxiliar, copiaremos en las primeras componentes del vector los elementos considerados correctos, es decir, que no cumplen la propiedad pedida (se eliminan los elementos que cumplen la propiedad). Antes de devolver el control modificaremos la longitud del vector para ajustarlo al número de valores correctos.

1.3. Ideas detalladas.

- Debemos utilizar las posiciones del vector cuyos valores son erróneos para colocar los valores correctos. Para ello llevaremos un índice: `valoresBuenos` con la posición del vector que cumple que todas las componentes desde el comienzo del vector hasta este índice son correctos.
  - Si la siguiente componente del vector es igual al elemento que queremos eliminar, se pasa a considerar el siguiente elemento.
  - Si la siguiente componente del vector es distinta al valor a eliminar, se copia en la posición indicada por el índice `valoresBuenos` y se avanza al elemento siguiente después de avanzar el índice `valoresBuenos`.
  - La parte del vector entre el índice `valoresBuenos` y el elemento que estemos considerando en esta vuelta del bucle son valores basura que al final del método se desprecian porque quedan a la derecha del índice `valoresBuenos`.

1.4. Algunas cuestiones sobre implementación.

- *Modificadores de tipos. Redefinición de un tipo.*

  En la descripción de los datos de entrada al problema, se dice que los elementos del vector están en el rango $(-2^{63} \dots 2^{63})$. Esto significa que los valores no pueden almacenarse en una variable de tipo `int` sino que deben almacenarse en una variable de tipo `long long int`.

  Podemos redefinir el tipo `long long int` dándole un identificador representativo y más fácil de escribir mediante la instrucción:

  ```
  using lli = long long int;
  ```

- *Cómo modificar el tamaño del vector.*

  La función `resize(n)` modifica el tamaño del vector para dejarlo en longitud `n`. Si el nuevo tamaño es menor que el anterior, como ocurre en este problema, la función elimina los últimos elementos del vector para dejar el tamaño pedido. El coste es lineal en el número de elementos que se eliminan, debido al coste de destruir los elementos.

- *Cómo evitar algunas copias de los elementos del vector.*

  Si las componentes del vector son de un tipo básico, copiaremos el valor en la posición `valoresBuenos`, sin modificar el valor de la componente actual. El valor de la componente actual se perderá, pero no importa porque es un valor a eliminar.

  Por el contrario, si las componentes del vector tienen un tipo que implementa la *semántica de movimiento* se utilizará la función `swap` de la STL que permite intercambiar los elementos sin realizar copias.

1.5. Errores frecuentes.

- Implementar una función con coste cuadrático respecto al número de elementos del vector, bien por utilizar la función `erase` o por desplazar todos los elementos a la derecha cuando se encuentra un valor erroneo.
- Implementar una función que devuelve como resultado un vector con los datos no eliminados. Esta solución utiliza un vector diferente del vector de entrada y por lo tanto no cumple con los requisitos expuestos en el enunciado del problema.

1.6. Coste de la solución

- En la implementación que se muestra al final del ejercicio y que sigue las ideas presentadas anteriormente, el vector se recorre completo una única vez mediante un bucle `for`. En cada iteración en el caso peor se realizarán una asignación, dos incrementos y dos comparaciones. Por lo tanto, siendo n es el número de elementos del vector, el coste es aproximadamente $5n$ y el orden de complejidad es $\mathcal{O}(n)$.

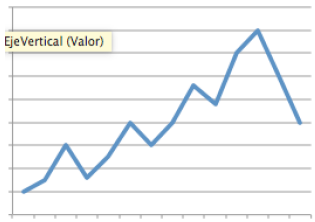
1.7. Implementación en C++.

```
#include <iostream>
#include <fstream>
#include <vector>

using lli = long long int;

// Funcion que resuelve el problema
void resolver (std::vector<lli> & v, int valorErroneo) {
    int valoresBuenos = 0;
    for (int k = 0; k < v.size() ;++k) { // Bucle que recorre el vector
        if (v[k] != valorErroneo){ // Si el dato es bueno
            v[valoresBuenos] = v[k];  // lo trasladamos a la zona correcta
            ++valoresBuenos;  // aumentamos la zona correcta
        }
    }
    v.resize(valoresBuenos);  // Dejamos en el vector unicamente los datos correctos
}
// Lectura de los datos de entrada, llamada a la funcion resolver y salida de datos
void resuelveCaso() {
    // Lectura de los datos
    int numElem, valorErroneo;
    std::cin >> numElem >> valorErroneo;
    std::vector<lli> v(numElem);
    for (lli& n : v) std::cin >> n;
    // Resolver el problema
    resolver(v, valorErroneo);
    // Escribir los datos de salida
    if (v.size() > 0) std::cout << v[0];
    for (int i = 1; i < v.size(); ++i)
        std::cout << ' ' << v[i];
    std::cout << '\n';
```

```
}

int main() {
    // Para la entrada por fichero.
   #ifndef DOMJUDGE
    std::ifstream in("datos.txt");
    auto cinbuf = std::cin.rdbuf(in.rdbuf());
   #endif
   // Entrada con numero de casos
    int numCasos;
    std::cin >> numCasos;
    for (int i = 0; i < numCasos; ++i) resuelveCaso()
        ;

    // Para restablecer entrada.
   #ifndef DOMJUDGE
      std::cin.rdbuf(cinbuf);
      system("PAUSE");
   #endif

    return 0;
}
```

2. Recorrido con acumuladores. Evolución de beneficios.

Evolución de los beneficios.



Nuestra empresa quiere ampliar el negocio y para ello va a comprar una compañía de fabricación de componentes. Antes de realizar la compra queremos comprobar el estado en que se encuentra la empresa de fabricación y para ello analizamos los datos sobre sus ventas y sus gastos.

Nuestro jefe nos ha encargado que obtengamos los años en que las ventas superaron las ventas de todos los años anteriores. Para ello contamos con las ventas de cada año desde que se fundó la empresa. Después de dedicar un rato a mirar los números, he decidido que lo más fácil es realizar un programa que me resuelva el problema.

Requisitos de implementación.

Debe implementarse una función que reciba un vector con todos los datos de las ventas y el año en que empezaron a tomarse los datos, y devuelva en otro vector los años en los que se superaron las ventas de los años anteriores.

El coste de la función debe ser del orden del número de datos de entrada.

Entrada

La entrada comienza con un valor entero que indica el número de casos de prueba. Cada caso de prueba consta de dos líneas. En la primera línea se indica el primer y el último año a los que corresponden los valores de las ventas que nos dan. En la segunda línea se muestran los valores de las ventas en los años indicados.

El número de años considerados es mayor que cero y el intervalo está entre los años 1700 y 4000. Los valores posibles de las ventas son números enteros en el intervalo $(2^{31} + 1..2^{31} - 1)$.

Salida

Para cada caso de prueba se escriben en una línea los años en los que las ventas superan las ventas de los años anteriores.

Entrada de ejemplo

```
5
2000 2017
159 172 181 190 201 213 227 239 243 233 232 229 225 225 227 233 241 250
2005 2012
281 292 304 310 300 308 315 318
2013 2017
-120 -140 -60 -50 -70
1990 1992
20 10 5
3500 3503
25 25 27 40
```

Salida de ejemplo

```
2000 2001 2002 2003 2004 2005 2006 2007 2008 2017
2005 2006 2007 2008 2011 2012
2013 2015 2016
1990
3500 3502 3503
```

2.1. Objetivos del problema

- Practicar a utilizar variables acumuladoras para evitar realizar cálculos acumulativos en cada iteración de un bucle.

2.2. Ideas generales.

- Los valores que se necesita conocer para decidir si el valor de la posición i-esima es mayor que todos los valores de su izquierda son los valores de la izquierda del vector. Por lo tanto se debe recorrer el vector de izquierda a derecha.
- Para evitar comprobar en cada posición si el valor es mayor que todos los de la izquierda, se guarda en una variable auxiliar el valor máximo encontrado hasta el punto en que se está recorriendo el vector.

2.3. Ideas detalladas.

- *Cómo calcular el número de elementos del vector.*

  Conocemos el año correspondiente al primer valor y el año correspondiente al último valor. Como solo se consideran años positivos, el número de datos es el año final menos el año inicial más uno.

- *Inicializar el valor máximo.*

  Como los datos pueden ser todos negativos, el beneficio máximo debe inicializarse al primer elemento del vector. En el enunciado se garantiza que el vector no será nulo.

2.4. Coste de la solución

En la solución del problema se recorrerá el vector una única vez y en cada iteración realizaremos una pregunta y en el caso peor una asignación y una insercción al final del vector. Realizamos 5 instrucciones en cada iteración todas ellas de coste constante, por lo que el coste esta en el orden de $\mathcal{O}(n)$ siendo n el número de elementos del vector.

Añadir un elemento en la última posición de un vector tiene coste amortizado constante. En el caso peor, el vector se llena y hay que hacer una copia de todos sus elementos para poder añadir el nuevo, por lo que en el caso peor el coste es lineal respecto al número de elementos del vector, sin embargo, si al ampliar el vector para añadir un nuevo elemento duplicamos su capacidad, puede demostrarse que si consideramos el coste de realizar varias operaciones éste es constante. Aunque en este curso estudiamos el coste de los algoritmos en el caso peor, cuando el coste amortizado de las operaciones sea constante consideraremos el coste como constante.

Se observa que si para cada elemento calculamos el valor máximo de las componentes anteriores a él en cada iteración, recorremos el vector en cada iteración y por lo tanto el coste será cuadrático en el número de elementos del vector.

2.5. Errores frecuentes.

No se conocen

2.6. Modificaciones al problema.

- Recorrer el vector de derecha a izquierda.
- Acumular la suma en lugar del máximo.

2.7. Implementación en C++.

```
#include <iostream>
#include <fstream>
#include <vector>
```

```
std::vector<int> resolver (std::vector<int> const& v, int inicio) {
    std::vector<int> sol;
    int PIBMax = v[0];
    sol.push_back(inicio); // posición del primer máximo
    for (int k = 1; k < v.size(); ++k) {
        if (v[k] > PIBMax){ // Encontrado nuevo máximo
            PIBMax = v[k];
            sol.push_back(k+inicio);
        }
    }
    return sol;
}

void resuelveCaso() {
    // Lectura de los datos
    int inicio, fin;
    std::cin >> inicio >> fin;
    std::vector<int> v(fin - inicio + 1);
    for (int& n : v) std::cin >> n;
    // Resolver el problema
    std::vector<int> sol = resolver(v, inicio);
    // Escribir los datos de salida
    if (sol.size() > 0) std::cout << sol[0];
    for (int i = 1; i < sol.size(); ++i)
        std::cout << ' ' << sol[i];
    std::cout << '\n';
}

int main() {
    // Para la entrada por fichero.
   #ifndef DOMJUDGE
    std::ifstream in("datos.txt");
    auto cinbuf = std::cin.rdbuf(in.rdbuf());
   #endif
   // Entrada con numero de casos
    int numCasos;
    std::cin >> numCasos;
    for (int i = 0; i < numCasos; ++i) resuelveCaso()
        ;

    // Para restablecer entrada.
   #ifndef DOMJUDGE
      std::cin.rdbuf(cinbuf);
      system("PAUSE");
   #endif

    return 0;
}
```

3. Segmento máximo, la carta más alta.

La carta mas alta.

Pablo está aprendiendo a contar. Para que practique, todas las tardes cuando su padre llega a casa del trabajo, juega con él varias partidas a *La carta mas alta*. El juego consiste en que cada jugador coge una carta del mazo y gana el que tiene la carta más alta. Como Pablo solo sabe contar hasta 7, su padre utiliza únicamente las cartas del 1 al 7 de una baraja española.

A Pablo le gusta ganar. Para animarle a seguir jugando, su padre apunta todos los días el número máximo de partidas seguidas que Pablo ha conseguido ganar.

Entrada

La entrada consta de una serie de casos de prueba. Cada caso es una secuencia de dígitos terminada con el carácter '.'. Los dígitos se escriben uno a continuación del otro, sin caracteres blancos entre ellos, y están entre 1 y 7 ambos incluidos.

Salida

Para cada caso de prueba se escribe en una línea el número máximo de partidas seguidas que ha ganado Pablo.

Entrada de ejemplo

```
425575.
4376.
56763245745241.
.
```

Salida de ejemplo

```
1
2
3
```

3.1. Objetivos del problema

- Calcular el segmento máximo de elementos de un vector que cumplen un propiedad.

3.2. Ideas generales.

- El problema se resuelve con un único bucle. En cada iteración debemos conocer la longitud del segmento máximo que cumple la propiedad encontrado hasta esta iteración y la longitud del segmento que cumple la propiedad y finaliza en la componente que se trata en esta iteración.

3.3. Algunas cuestiones sobre implementación.

- *Se puede utilizar el tipo* `string` *para la lectura de datos..*

  Los datos se dan como una cadena de dígitos sin blancos. Leer la cadena completa en una variable de tipo `string`. Se puede utilizar la instrucción `std::cin << s` siendo `s` una variable de tipo `std::string`. Con esta instrucción se descartan los caracteres blancos, tabuladores o salto de línea hasta encontrar un que no sea uno de los anteriores, a continuación se guardan en `s`, todos los caracteres hasta un blanco, tabulador o salto de línea. También puede utilizarse la instrucción `getline(std::cin, s);`, con esta instrucción se guardan en la variable `s` todos los caracteres hasta el salto de línea incluido.

- *El final de datos.*

  La última línea se marca con el carácter punto. Si la lectura de datos se realiza con una variable de tipo `string`, la comparación para detectar el final de datos debe ser con la cadena de caracteres punto (`"."`). Si se utiliza el carácter punto (`'.'`) se produce un error de compilación.

3.4. Ideas detalladas.

- *El bucle que recorre el vector.*

  La cadena de caracteres tiene en las posiciones pares la jugada del primer jugador y en las posiciones impares la jugada del segundo jugador. Por lo tanto cada vuelta del bucle debe comparar dos valores y avanzar dos posiciones. Debemos verificar que las condiciones de entrada dadas en el enunciado (precondición) garantizan que el número de elementos sea par.

```
for (int i = 1; i < s.size(); i+=2) {...}
```

- *Posibles implementaciones del bucle.*

  La instrucción del bucle debe actualizar el valor de todas las variables, para ello si el valor que se está tratando en la iteración cumple la propiedad pedida hay que incrementar la longitud del último segmento y en caso de que no la cumpla hay que iniciar un nuevo segmento. Además, si el último segmento supera al que tenemos guardado como el máximo hasta el momento hay que actualizar el segmento máximo. Observamos que el segmento actual sólo puede superar al máximo hasta este momento si se incrementa su contador. Por lo tanto sólo es necesario actualizarlo cuando el valor cumple la propiedad.

```
for (int i = 1; i < s.size(); i+=2) {
    if (s[i-1]>s[i]) { // gana primer jugador
        ++cont;
        if (cont > contMax) {
            contMax = cont;
        }
    }
    else cont = 0;
}
```

Con esta solución el número de veces que se modifica el contador máximo coincide con la longitud del segmento máximo.

Otra solución posible consiste en controlar si el segmento actual supera al segmento máximo cuando encontramos un valor que no cumple la propiedad, es decir, cuando termina el segmento válido. Sin embargo, es muy importante ver que en este caso si el último segmento del vector es válido y es el de tamaño máximo, quedará sin actualizar, ya que el segmento terminó con el final del vector en lugar de con un elemento que no cumple la propiedad. Por lo tanto hay que comprobar al terminar el bucle si el último segmento es mayor que el encontrado hasta este momento.

```
for (int i = 1; i < s.size(); i+=2) {
    if (s[i-1]>s[i]) { // gana primer jugador
        ++cont;
    }
    else {
        if (cont > contMax) contMax = cont;
        cont = 0;
    }
}
if (cont > contMax) contMax = cont;
```

3.5. Errores frecuentes.

1. Olvidarse de comprobar el último segmento cuando se modifica el contador al finaliza el segmento.

3.6. Coste de la solución

El bucle recorre la mitad de los elementos del vector. En cada iteración del bucle se consultan dos elementos del vector y se hace un incremento, una comparación y una asignación, o se hace una asignación. Por lo tanto cada iteración tiene coste constante. Como el bucle hace $n/2$ iteraciones, siendo n el número de elementos del vector, el coste del bucle es del orden de $\mathcal{O}(n)$.

3.7. Modificaciones al problema.

- *Obtener los segmentos con longitud máxima.*

  Nos pueden pedir obtener el segmento máximo más a la derecha del vector, o más a la izquierda, u obtenerlos todos. En los dos primeros casos es necesario utilizar una variable auxiliar más en la que se guarda el comienzo del segmento máximo encontrado hasta este momento. En el caso en que nos pidan obtener todos los segmentos máximos es necesario utilizar un vector para almacenar el comienzo de todos los segmentos máximos encontrados.

  El inicio del segmento máximo se actualiza cada vez que se encuentra un segmento mayor. Si nos piden el segmento máximo más a la derecha debemos actualizar el inicio cuando encontramos un intervalo mayor o igual, y en caso de que nos pidan todos los comienzos actualizaremos el vector cuando se encuentre un segmento estrictamente mayor y añadiremos un nuevo inicio cuando se encuentre un intervalo igual.

  Solución del problema *La carta mas alta* para el caso en que se pide el inicio del intervalo más a la izquierda.

```
struct solucion {
    int contMax;
    int iniMax;
};
solucion resolver (std::string const& s) {
    solucion sol;
    int cont = 0; sol.contMax = 0;
    for (int i = 1; i < s.size(); i+=2) {
```

```
            if (s[i-1]>s[i]) { // gana primer jugador
                ++cont;
                if (cont > sol.contMax) {
                    sol.contMax = cont;
                    sol.iniMax = i - 2*cont + 1;
                }
            }
            else cont = 0;
        }
            return sol;
    }
```

- *Obtener el segmento de suma máxima.*

  En este caso llevaremos en una variable la suma máxima encontrada hasta el momento en que estamos ejecutando y en otra la suma del último segmento que estamos tratando. El segmento actual acaba si la suma se hace negativa ya que en este caso conviene comenzar un nuevo segmento en lugar de acumular al segmento anterior.

3.8. Implementación en C++.

```
#include <iostream>
#include <fstream>
#include <vector>

// Intervalo mas largo en el que ha ganado el primer jugador
int resolver (std::string const& s) {
    int cont = 0; int contMax = 0;
    for (int i = 0; i < s.size()-1; i+=2) {
        if (s[i]>s[i+1]) { // gana primer jugador
            ++cont;
            if (cont > contMax) {
                contMax = cont;
            }
        }
        else cont = 0;
    }
    return contMax;
}


bool resuelveCaso() {
    std::string datos;
    std::cin >> datos;
    if (datos = ".") return false;
    std::cout << resolver(datos) << '\n';
    return true;
}

int main() {
    // Para la entrada por fichero.
   #ifndef DOMJUDGE
    std::ifstream in("datos.txt");
    auto cinbuf = std::cin.rdbuf(in.rdbuf());
   #endif

   // Entrada con centinela
    while (resuelveCaso())
        ;
```

```
    // Para restablecer entrada.
   #ifndef DOMJUDGE
      std::cin.rdbuf(cinbuf);
      system("PAUSE");
   #endif

    return 0;
}
```

4. Intervalos, piedras preciosas.

Piedras preciosas

En la novela "Kim de la India", cuando Kim visita al "Curandero de perlas", éste le propone jugar junto a su otro discípulo al "Juego de las joyas". Sobre una bandeja se encuentran una serie de joyas que cada uno de los jugadores debe recordar, después de mirarlas durante un corto espacio de tiempo. Ganará aquel que consiga describir con mayor exactitud las piedras preciosas. La primera vez que juega, Kim no consigue recordar todas ellas y es vencido por el otro chico. Sin embargo, aprende pronto y las siguientes veces consigue hacer la relación exacta del contenido de la bandeja.

Dado que ahora los dos chicos son capaces de recordar las piedras de la bandeja con toda exactitud, el sahib Lurgan ha decidido modificar un poco el juego para hacerlo más complicado. Coloca las piedras preciosas formando una línea y les pregunta cuantas veces aparece una secuencia de longitud 7 que tenga al menos 3 zafiros y 2 rubís. Viendo que el juego capta su interés sigue realizando este tipo de preguntas, ¿Cuantas veces aparece una secuencia de longitud 5 con al menos 3 diamantes y 1 rubí?, o ¿Cuantas veces aparece una secuencia de tamaño 4 con al menos 2 esmeraldas y 2 jades?

Para no tener que comprobar visualmente que discípulo ha respondido de forma correcta, el sahib desarrolla un programa que dada la lista con las piedras calcula cuantas veces aparece la secuencia deseada. De esta forma no tiene miedo de equivocarse al dar el premio.

Entrada

La entrada comienza con el número de casos de prueba. Cada caso tiene dos líneas. En la primera se indica el número de piedras, el tamaño de la secuencia que se busca, el primer tipo de piedra y el número de veces que debe aparecer y el segundo tipo de piedra y el número de veces que debe aparecer. En la segunda línea se indica la lista de piedras preciosas representadas por su inicial en minúsculas.

Las piedras preciosas que se consideran son diamante, rubí, esmeralda, zafiro y jade. Cada una se identifica por el primer carácter de su nombre. Se garantiza que el tamaño de la secuencia es menor o igual que el número de piedras y que la suma del número de veces que debe aparecer la primera y la segunda piedra es menor o igual que la longitud de la secuencia.

Salida

Para cada caso de prueba se escribe en una línea el número de secuencias que cumplen la propiedad pedida.

Entrada de ejemplo

```
4
6 3 d 1 z 1
r d z e d z
7 3 e 2 j 1
e j e r e e j
5 2 r 1 z 0
d z j r e
7 4 z 2 r 1
z r d z z r e
```

Salida de ejemplo

```
4
2
2
4
```

Autor: Isabel Pita.

4.1. Objetivos del problema

- Practicar problemas que buscan intervalos de longitud fija que cumplen una propiedad.

4.2. Ideas generales.

- El problema se resuelve con un bucle inicial que permite inicializar los valores con los primeros elementos del vector y un bucle principal donde se recorre el resto del vector.
- En el bucle inicial se calculan los valores que permiten resolver el problema para el intervalo inicial de valores.
- En el bucle principal se actualizan los valores eliminando el primer elemento del intervalo y añadiendo un nuevo elemento por la derecha.

4.3. Algunas cuestiones sobre implementación.

4.4. Implementación en C++.

```
#include <iostream>
#include <fstream>
#include <vector>

enum piedrasPreciosas {diamante, rubi, esmeralda, zafiro, jade};

// lectura de valores del tipo enumerado
std::istream& operator>> (std::istream& entrada, piedrasPreciosas& p) {
    char num;
    std::cin >> num;
    switch (num) {
        case 'd': p = diamante; break;
        case 'r': p = rubi; break;
        case 'e': p = esmeralda; break;
        case 'z': p = zafiro; break;
        case 'j': p = jade; break;
    }
    return entrada;
}

// Función que resuelve el problema
int resolver(std::vector<piedrasPreciosas> const& v, int p, piedrasPreciosas t1,
             int x, piedrasPreciosas t2, int y){
    // Inicializar los valores con el primer intervalo
    int cont = 0; // número de intervalos que tienen las gemas requeridas
    int num1 = 0; // Número de gemas del primer tipo
    int num2 = 0; // Número de gemas del segundo tipo
    for (int i = 0; i < p; ++i) { // Cuenta las gemas del primer intervalo
        if (v[i] = t1) ++num1;
        else if (v[i] = t2) ++num2;
    }
    if (num1 ≥ x && num2 ≥ y) ++cont; // Si en intervalo tiene las gemas deseadas
    // Bucle principal
    for (int j = p; j < v.size(); ++j) { // avanza el intervalo una posición
    // Elimina la gema de la izquierda
        if (v[j-p] = t1) --num1;
        else if (v[j-p] = t2) --num2;
        // Añade la gema de la derecha
        if (v[j] = t1) ++num1;
        else if (v[j] = t2) ++num2;
        // Comprueba si el intervalo tiene las gemas deseadas
        if (num1 ≥ x && num2 ≥ y) ++cont;
    }
```

```
    return cont;
}

// Resuelve un caso de prueba, leyendo de la entrada la
// configuración, y escribiendo la respuesta
bool resuelveCaso() {
    int numElem;
    std::cin >> numElem;
    if (!std::cin) return false;
    int numpiedras; int numtipo1, numtipo2; piedrasPreciosas tipo1, tipo2;
    std::cin  >> numpiedras  >> tipo1 >> numtipo1 >> tipo2 >> numtipo2;
    std::vector<piedrasPreciosas> v(numElem);
    for (int i = 0; i < numElem; ++i) std::cin >> v[i];
    //for (piedrasPreciosas& i : v) std::cin >> i;
    int cont = resolver(v,numpiedras, tipo1, numtipo1, tipo2, numtipo2);
    std::cout << cont << '\n';
    return true;
}

int main() {
    // Para la entrada por fichero.
   #ifndef DOMJUDGE
    std::ifstream in("datos.txt");
    auto cinbuf = std::cin.rdbuf(in.rdbuf());
   #endif

   // Entrada con centinela
    while (resuelveCaso())
        ;

    // Para restablecer entrada.
   #ifndef DOMJUDGE
      std::cin.rdbuf(cinbuf);
      system("PAUSE");
   #endif

    return 0;
}
```