Cuaderno de problemas
Fundamentos de algorítmia.

Algoritmos iterativos

Prof. Isabel Pita

21 de septiembre de 2022

Índice

1. Vector de acumulados. Reserva de elefantes

Reservas de elefantes

El parque nacional de Chobe, en la región noroeste de Botswana, es famoso por su alta densidad de elefantes. Los encargados del parque llevan muchos años recopilando datos sobre el número de nacimientos de elefantes en la región. Sin embargo, los visitantes del parque no suelen estar interesados en los nacimientos de un determinado año, sino que preguntan por el número de nacimientos entre dos fechas, lo que obliga a los encargados a sumar los datos entre las dos fechas pedidas. Cansados ya de tanto sumar, han decidido que deben mantener las sumas ya realizadas y de esta forma solo tendrán que consultar el dato. Pero mantener todos los intervalos de fechas no resulta viable, por lo que después de pensarlo un poco han decidido que con tener los datos acumulados de los nacimientos de todos los años anteriores es suficiente.

Requisitos de implementación.

Los datos deben prepararse para que resolver cada pregunta tenga un coste constante. La preparación de los datos tiene coste lineal en el número de datos que tiene el parque.

Entrada

La entrada consta de una serie de casos de prueba. Cada caso se escribe en varias líneas. En la primera se muestra el primer y último año (0 < p <= u <= 100.000) en los que se han tomado datos de los nacimientos de elefantes. En la segunda línea se muestran los `n = u-p+1` nacimientos ocurridos durante estos años (0 <= k < 2^{30}). En la línea siguiente se indican el número de preguntas que realizan los visitantes (0 < m <=100.000) y en las `m` líneas siguientes se muestran dos años (0 <=a <= b < 1.000) que representan el intervalo en el que se debe calcular el número de nacimientos de elefantes.

El final de los casos se marca con dos ceros que no deben tratarse.

Salida

Para cada caso de entrada se escriben tantas líneas como preguntas haya en el caso. En cada línea se escribe el número de nacimientos ocurridos durante los años que se indican en la pregunta. Ambos años incluidos.

Cada caso termina con una línea con tres guiones (`---`).

Entrada de ejemplo

```
1950 1955
4 1 2 0 3 6
3
1950 1953
1952 1955
1951 1954
2010 2016
4 2 1 6 4 4 2
2
2014 2014
2010 2016
0 0
```

Salida de ejemplo

```
7
11
6
---
4
23
---
```

Autor: Isabel Pita

1.1. Objetivos del problema

- Crear un vector a partir de otro con sus valores acumulados.
- Obtener información a partir de un vector de valores acumulados.

1.2. Ideas generales.

- En la descripción de la entrada se indica que para los datos obtenidos entre dos fechas se realizarán varias preguntas, y aclara que el número de preguntas puede llegar a 100.000. Este requisito insinúa que la respuesta a cada pregunta se debe realizar en tiempo *casi constante*, ya que el coste de responder cada pregunta se debe multiplicar por el número de preguntas realizadas. Por lo tanto, se deben preparar los datos de entrada de forma que nos permitan responder a las preguntas *rápido*.
- Nos basaremos en: $\sum_{i=x}^{y} a_i = \sum_{i=0}^{y} a_i - \sum_{a=0}^{x} a_i$.
- Se creará un vector cuya componente i-ésima tenga el valor $\sum_{j=0}^{i} a_j$.
- A partir de los datos del vector de acumulados se puede obtener mediante una simple operación aritmética el resultado de cada pregunta.

1.3. Algunas cuestiones sobre implementación.

- El número de valores de entrada se obtiene como la diferencia de los años de fin e inicio más uno.
- El vector de acumulados puede crearse con una componente más que el vector de entrada y fijar el valor de la primera componente a cero. De esta forma al restar el año de inicio de toma de datos del inicio del periodo obtenemos el índice del vector correspondiente a la suma de los valores anteriores al inicio del periodo.

  indice suma valores anteriores al inicio del periodo = año inicio periodo - año inicio datos.

  La suma acumulada de los nacimientos desde el inicio de toma de datos hasta el final del periodo incluido lo obtenemos en la componente siguiente a la resta del inicio de la toma de datos del año de finalización.

  suma desde el inicio de datos hasta final del periodo = v[año fin periodo - año inicio datos].

- Si el vector de acumulados se declara con el mismo número de componentes que el vector de entrada, la suma de los valores hasta el comienzo del periodo (sin incluir) está en la posición

  indice suma valores anteriores al inicio del periodo = año inicio periodo - año inicio datos - 1.

  Por lo tanto hay que diferenciar el caso en que el inicio del periodo coincide con el inicio de los datos para no acceder al vector fuera de rango.

1.4. Ideas detalladas.

- El problema hará uso de una función que calcule el vector de acumulados. Esta función recorre el vector de entrada de izquierda a derecha y cada componente se calcula como la componente anterior acumulada mas el valor de la entrada.

```
acum[0] = 0;
for (int i = 1; i < entrada.size() + 1; ++i)
{
    acum[i] = acum[i-1] + entrada[i-1];
}
```

- Después de calcular el vector de acumulados se procesarán las preguntas. Para cada pregunta se calcula la respuesta haciendo uso del vector y se muestra en consola.

```
    int m; std::cin >> m;
    for (int i = 0; i < m; ++i) {
      int f1, f2;
      std::cin >> f1 >> f2;
      std::cout << acum[f2-a1+1]-acum[f1-a1] << '\n';
    }
```

1.5. Coste de la solución

- El coste de calcular el vector de acumulados es lineal en el número de elementos del vector, que se corresponde con el número de años de los que se han tomado medidas. El algoritmo tiene un bucle que recorre el vector de entrada completo. En cada vuelta del bucle se realiza una asignación, una comparación y dos incrementos, todos ellos de coste constante. Por lo tanto el coste es el número de vueltas del bucle.
- El coste de resolver cada pregunta es constante, ya que se resuelve con dos accesos al vector. Por lo tanto el coste de resolver todas las preguntas es del orden del número de preguntas.
- Como los dos procesos son secuenciales, primero se calcula el vector de acumulados y después se contestan las preguntas. El coste es el máximo entre los dos procesos. Por lo tanto el coste total del problema (incluyendo las preguntas) está en el orden $\mathcal{O}($ max(número de años considerados, número de preguntas realizadas)).

1.6. Errores frecuentes.

- Realizar una implementación de coste cuadrático respecto al número de años considerados y el número de preguntas. Si sumamos el número de nacimientos para cada pregunta que nos hacen, tenemos que el coste es del orden $\mathcal{O}(k * n)$ siendo k el número de preguntas y n el número de años sumados, en el caso peor todos ellos. Cómo el número de años es un valor $0 < p \leq u \leq 100{,}000$, y el número de preguntas es un valor $0 < m \leq 100{,}000$, tenemos en total unas 10^{10} operaciones, lo que no es ejecutable con un algoritmo cuadrático en un tiempo razonable.
- Los datos de entrada indican que el número de nacimientos de cada año es un entero $0 \leq k < 2^{30}$, Si todos los años nace el máximo de animales, como podemos tener hasta 100.000 años, no es posible almacenar la suma de todos los nacimientos en una variable de tipo `int`. Hay que utilizar el tipo `long long int` para guardar la suma de los valores.

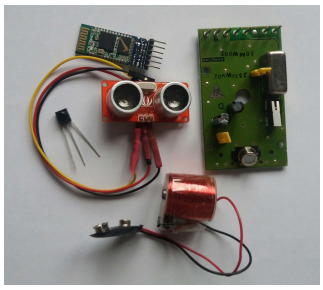
1.7. Implementación en C++.

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

using lli = long long int;

// Calcula un vector con los valores acumulados
// El vector de acumulados tiene una posicion mas que el vector de entrada
// La primera posicion toma el valor cero para facilitar el calculo de las diferencias
void resolver(std::vector<int> const& a, std::vector<lli>& v)
{
    v[0] = 0;
    for (int i = 1; i < v.size(); ++i)
    {
        v[i] = v[i-1] + a[i-1];
    }
}

bool resuelveCaso() {
    // Lee las dos fechas entra las que tenemos datos
```

```
    int a1, a2;
    std::cin >> a1 >> a2;
    if (a1 == 0 && a2 == 0) return false;
    // Declara el vector de entrada de datos y lo lee
    int nElem = a2-a1+1;
    std::vector<int> v(nElem);
    for (int i = 0; i < nElem; ++i) {
        std::cin >> v[i];
    }
    // Calcula el vector de acumulados
    std::vector<lli> salida(v.size()+1);
    resolver(v,salida);
    // Lee las preguntas y mira la respuesta en el vector de acumulados
    int m; std::cin >> m;
    for (int i = 0; i < m; ++i) {
        int f1, f2;
        std::cin >> f1 >> f2;
        std::cout << salida[f2-a1+1]-salida[f1-a1] << '\n';
    }
    std::cout << "---\n";
    return true;

}

int main() {
#ifndef DOMJUDGE
    std::ifstream in("datos.txt");
    auto cinbuf = std::cin.rdbuf(in.rdbuf()); //save old buf and redirect std::cin to casos.txt
#endif

    while (resuelveCaso());

#ifndef DOMJUDGE // para dejar todo como estaba al principio
    std::cin.rdbuf(cinbuf);
    system("PAUSE");
#endif

    return 0;
}
```

2. 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
```

2.1. Objetivos del problema

- Aprender a eliminar varios elementos de un vector en tiempo lineal respecto al número de elementos del vector, sin utilizar memoria auxiliar.
- Recordar los modificadores de tipo de C++.

2.2. 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 la 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.

2.3. 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.

2.4. 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 el índice `valoresBuenos` para pasar al elemento siguiente.
- 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`.

2.5. 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 medidas tomadas por el sensor, el coste es aproximadamente $5n$ y el orden de complejidad es $\mathcal{O}(n)$.

2.6. 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 diferente del de entrada 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.

2.7. 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;
}
```

3. Juego de dados

Juego de dados

Dos amigos han pasado la tarde jugando a los dados. El juego consiste en meter dos dados en un cubilete, luego lanzar los dados y ver cual de los dos ha conseguido una puntuación mayor. Cada uno ha llevado la cuenta de todas sus tiradas, apuntando el valor de los dados. Ahora quieren saber cual es el valor que más veces ha obtenido cada uno.

Requisitos de implementación.

El problema debe resolverse con una función que recibe en un vector los datos de entrada y devuelve un vector con los valores que se repiten el número máximo de veces.

Se utilizará un vector auxiliar para contar las veces que aparece cada número.

Entrada

La entrada consta de una serie de casos de prueba. Cada caso consta de dos líneas, en la primera se muestra el número de veces que se han tirado los dados y el valor máximo que se puede conseguir con las tiradas (hay que tener en cuenta que no todos los dados tienen 6 caras, hay dados con muchas caras, con los que se pueden obtener más de 6 valores diferentes), y en la segunda se muestran los valores obtenidos con los dados. El final de casos se marca con el valor -1.

El máximo número de caras de un dado es 10.000. El número de veces que se tiran los dados es mayor o igual que uno y menor o igual que 200.000.

Salida

Para cada caso de prueba se escribe en una línea el valor de los dados que más se ha repetido en la secuencia. Si hay varios que se repiten el número máximo de veces se muestran todos ellos, ordenados de mayor a menor.

Entrada de ejemplo

```
7 12
5 3 8 5 10 10 5
6 10
5 9 9 2 5 2
1 24
12
-1
```

Salida de ejemplo

```
5
9 5 2
12
```

Autor: Isabel Pita

3.1. Objetivos del problema

- Conocer las diferentes formas de calcular la moda de una colección de elementos. Se conoce como moda al elemento que más veces se repite en la colección.
- Aprender a acumular la información en una estructura de datos (vector) para resolver el problema de forma eficiente.

3.2. Ideas generales.

- El problema pide calcular la moda de una colección de valores. Los valores están acotados y por lo tanto se puede utilizar un vector auxiliar para contar las apariciones de cada elemento.
- Se declara un vector auxiliar con tantas componentes como valores posibles haya en el vector. Si no se conoce el mínimo y el máximo de los valores del vector, se calculará. En el vector se almacena el número de veces que aparece cada valor.
- Observad que si el número de datos diferentes es pequeño pero están muy dispersos puede utilizarse un TAD diccionario que se estudiará en la asignatura de ED.

3.3. Algunas cuestiones sobre implementación.

- El problema pide que se escriban todos los valores que aparecen un número máximo de veces. Estos valores los guardaremos en un vector que será el resultado de la función como pide el enunciado.
- Lo más sencillo es implementar dos bucles. En el primero se calcula el valor máximo y en el segundo se recorre el vector guardando todas aquellas componentes cuyo valor sea igual al máximo.

3.4. Ideas detalladas.

- Se declara un vector auxiliar en el que se contarán el número de veces que aparece cada valor.
- El tamaño del vector será el rango de los valores del vector. En este problema los valores están en el rango *[0..maxValor]*, siendo *maxValor* un dato de entrada.
- En general, buscaremos el valor máximo y mínimo de los datos y declararemos el vector auxiliar de tamaño *maximo - minimo +1*. La información del valor i-ésimo se encontrará en la componente *i-minimo* del vector.
- Se recorre el vector de entrada y se incrementa la componente del vector auxiliar correspondiente al valor del vector de entrada.
- Una vez creado el vector auxiliar lo recorremos buscando el valor máximo.
- Cómo en este problema nos piden que devolvamos todos los valores que tiene un número de apariciones máximo, haremos un nuevo bucle que recorra el vector de apariciones y guarde todos los índices cuyas componentes tengan un valor máximo en un vector que devolveremos como resultado de la función.

3.5. Coste de la solución

- En la solución del problema hay dos partes.
- Primero se crea el vector auxiliar con las apariciones de cada elemento. Se implementa con un bucle que da un número de vueltas igual al número de tiradas de dados que se han realizado. En cada vuelta se modifica una componente de un vector, por lo tanto el coste de cada vuelta es constante. El coste de esta primera parte es lineal en el número de tiradas de dados.
- Después se recorre el vector auxiliar para encontrar el valor máximo y contar el número de veces que aparece. Se hace con dos bucles que tienen el mismo coste. El número de vueltas que da el bucle es el número de posibles valores que se pueden obtener con los dados. El coste de cada vuelta es constante. Por lo tanto el coste de la segunda parte es del orden del número de valores posibles de los dos dados.

- Por lo tanto el coste del algoritmo es el máximo entre el número de tiradas de los dados realizadas y el número de posibles valores que puedan tomar los dados.

3.6. Errores frecuentes.

- No tener en cuenta que el cero y el valor máximo son valores posibles de los dados y por lo tanto el tamaño del vector donde se contarán las apariciones debe ser $maxValor + 1$.
- La salida se pide ordenada de mayor a menor. Para obtenerla así directamente, el vector de apariciones en el que se han contado las apariciones de cada elemento se debe recorrer de derecha a izquierda. De esta forma el primer elemento del vector de salida será el mayor índice.

3.7. Implementación en C++.

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

// Devuelve un vector con los valores que se repiten el numero maximo de veces
std::vector<int> resolver(std::vector<int> const& v, int maximoValores){
    // Cuenta el numero de apariciones
    std::vector<int> apariciones(maximoValores);
    for (int n : v) ++apariciones[n-1]; // Los valores de v comienzan en 1
    // Calcula el maximo del vector apariciones
    int aparicionesMax = apariciones[0];
    for (int n : apariciones) aparicionesMax = std::max(aparicionesMax, n);
    // Anade a sol los valores que aparecen aparicionesMax veces
    std::vector<int>sol;
    for (int i = 0; i < apariciones.size(); ++i)
        if (apariciones[i] == aparicionesMax) sol.push_back(i+1); // se suma 1 porque los valores
    return sol;
}

bool resuelveCaso() {
    int numTiradas, valorMax;
    std::cin >> numTiradas >> valorMax;
    if (numTiradas == -1) return false;
    std::vector<int> v(numTiradas);
    for (int& i : v) std::cin >> i;
    std::vector<int> sol = resolver(v, valorMax);
    std::cout << sol[sol.size()-1];
    for (int i = sol.size()-1; i > 0; --i )
        std::cout << ' ' << sol[i-1];
    std::cout << '\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 concentinela
     while (resuelveCaso()) ;

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

```
    #endif

     return 0;
}
```

4. Segmento máximo, rescate aéreo.

Rescate aereo 1

Después de sufrir un devastador ataque alienígena, el alto mando ha recibido una llamada de rescate de la ciudad de Nueva York. El grupo de supervivientes se encuentra detras de la línea de rascacielos de la ciudad. Al otro lado de los edificios se encuentra vigilando una nave extraterrestre. Para evitar que el transporte aereo encargado del rescate sea detectado por la nave enemiga, hemos encontrado la secuencia más larga de edificios cuya altura es estrictamente mayor que la de nuestro transporte. El alto mando ya ha avisado a los supervivientes para que se dirigan a este punto de encuentro. Suponemos que todos los edificios tienen la misma anchura.

Requisitos de implementación.

Para resolver este problema hay que estudiar primero el problema 3: Segmento máximo, todos con la selección, del cuaderno de problemas que se encuentra en el campus.

Se utilizará la plantilla especial que hay en el campus en la pestaña de "Laboratorio" para este problema.

El coste de la función debe ser lineal en el número de edificios considerados.

Entrada

La entrada comienza con el número de casos de prueba. Cada caso de prueba tiene dos líneas. En la primera se indica el número de edificios de la línea n, seguido de la altura del transporte t. En la segunda se indica la altura de cada edificio de la línea.

Se supone que la altura del transporte es menor que la altura de alguno de los edificios considerados y por lo menos existe un edificio.

Salida

Para cada caso de prueba se escriben en una línea el comienzo y el final del intervalo. En caso de existir dos intervalos iguales se eligirá el de la izquierda.

Entrada de ejemplo

```
2
10 5
3 6 4 8 9 8 7 2 8 9
7 5
8 8 8 2 3 9 9
```

Salida de ejemplo

```
3 6
0 2
```

Autor: Isabel Pita

4.1. Objetivos del problema

- Aprender el esquema de solución de los problemas que piden calcular el segmento máximo de elementos de un vector que cumplen una propiedad.

4.2. Ideas generales.

- En los problemas de segmento máximo se debe guardar información sobre el segmento mas largo que se haya encontrado que cumple la propiedad pedida (*segmento máximo*). Hay tres datos importantes sobre el segmento: su longitud, su punto de comienzo y su punto de final. Sólo es necesario guardar dos de estos datos, ya que el tercero se obtiene a partir de los dos guardados. El elegir unos u otros depende del implementador.

- Se debe también guardar información sobre el último segmento que cumple la propiedad pedida (*segmento actual*). Entendemos por segmento actual el que termina en el valor que se está tratando. Sobre este segmento conocemos su final, porque coincide con el valor que estamos tratando, por lo tanto solo es necesario guardar su longitud o su punto de comienzo. Como en el caso anterior uno de estos datos se obtiene a partir de los otros dos.

- El problema se resuelve con un único bucle que recorre el vector. En cada iteración se actualiza el segmento actual dependiendo de si el dato que se trata en la iteración cumple la propiedad o no. Una vez actualizado el segmento actual se comprueba si debe actualizarse el segmento máximo.

4.3. Ideas detalladas.

- *Posibles implementaciones del bucle.*

  Las instrucciones del bucle deben actualizar el valor de todas las variables: longitud máxima hasta este momento, longitud del segmento actual, y posición del inicio del segmento máximo. 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 longitud. Por lo tanto sólo es necesario actualizarlo cuando el valor cumple la propiedad.

```
for (int i = 0; i < v.size(); ++i) {
  if (alturaNave < v[i]) { // elemento bueno
    ++longActual;
    if (longActual > longSegMax) { // mejora el segmento
      longSegMax = longActual;
      iniSegMax = i - longActual + 1;
    }
  }
  else longActual = 0;
}
```

  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. En esta solución, es muy importante ver que si el último segmento del vector es válido y es el de tamaño máximo, quedará sin actualizar al acabar el bucle, 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 = 0; i < v.size(); ++i) {
  if (alturaNave < v[i])  ++longActual;  // elemento bueno
  else {
    if (longActual > longSegMax) { // mejora el segmento
      longSegMax = longActual;
      iniSegMax = i - longActual + 1;
    }
    longActual = 0;
```

```
    }
  }
  if (longActual > longSegMax) { // mejora el segmento
    longSegMax = longActual;
    iniSegMax = i - longActual + 1;
  }
```

Ambos bucles tienen coste lineal en el número de elementos del vector. Hay que notar que el caso peor es diferente para cada bucle. En el primer bucle el caso peor ocurre cuando todos los edificios tienen una altura mayor que la nave. En este caso se modifica el segmento máximo para cada elemento del vector. En el segundo tipo de bucle el caso peor ocurre cuando la altura de los edificios va alternando, una más alta y a continuación una más baja. Se modifican los valores del segmento máximo cada dos elementos. Por lo tanto la constante multiplicativa es el doble en el primer bucle respecto al segundo. Sin embargo, si el tiempo no es muy crítico se hace notar que es bastante más sencillo probar la corrección del primer bucle que la del segundo.

4.4. Coste de la solución

Vemos el coste de la solución que se muestra al final del problema. En ella se utiliza el primer bucle explicado en el apartado anterior. El bucle recorre todos los elementos del vector. En cada iteración del bucle se consultan un elemento del vector y se hacen dos incrementos, dos comparaciones y tres asignaciones, todas ellas operaciones de coste constante. Por lo tanto cada iteración tiene coste constante. Como el bucle hace n iteraciones, siendo n el número de edificios, el coste del bucle es del orden de $\mathcal{O}(n)$.

4.5. Errores frecuentes.

- Proponer un algoritmo de coste cuadrático respecto al número de elementos.
- En el segundo bucle, olvidarse de comprobar el último segmento después del bucle.

4.6. 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 en la que se guarda el comienzo o final 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 o final de todos los segmentos máximos encontrados.

  El inicio/final del segmento máximo se actualiza cada vez que se encuentra un segmento mayor. Si nos piden el segmento máximo más a la izquierda debemos actualizar el inicio/final cuando encontramos un intervalo estrictamente mayor, si nos piden el segmento más a la derecha actualizamos el inicio/final 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/final cuando se encuentre un intervalo igual.

- *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.

4.7. Implementación en C++.

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

```
// Devuelve la posicion inicial y final del segmento maximo
std::pair<int, int> resolver(std::vector<int> const& v, int alturaNave) {
    int longSegMax = 0, iniSegMax = 0, longActual = 0;
    for (int i = 0; i < v.size(); ++i) {
        if (alturaNave < v[i]) { // elemento bueno
            ++longActual;
            if (longActual > longSegMax) { // mejora el segmento
                longSegMax = longActual;
                iniSegMax = i - longActual + 1;
            }
        }
        else longActual = 0;
    }
    return {iniSegMax, iniSegMax + longSegMax - 1};
}

void resuelveCaso() {
    // Lectura de los datos
    int numElem, alturaNave;
    std::cin >> numElem >> alturaNave;
    std::vector<int> v(numElem);
    for (int& i : v) std::cin >> i;
    // Resolver el problema
    std::pair<int, int> sol = resolver(v, alturaNave);
    // escribir el resultado
    std::cout << sol.first << ' ' << sol.second << "\n";
}

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

    // Entrada por 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;
}
```

5. Segmento máximo, todos con la selección.

Todos con la selección

Si la selección nacional gana el próximo partido en Málaga, habrá ganado 6 partidos seguidos. Hacia bastante tiempo que no tenía una racha ganadora seguida tan larga. Nuestro periodista encargado de seguir el partido del próximo sábado quiere saber cual ha sido la racha ganadora más larga de la selección en todo la historia, para poder contarlo durante el partido.

Para ello recopila los datos y le pide a un amigo informático que le ayude a analizarlos con un programa. Deben obtener el máximo número de partidos seguidos que ha conseguido ganar la selección, si ha ocurrido varias veces que se ganasen este número de partidos, y hace cuantos partidos que finalizó la última racha.

Requisitos de implementación.

Implementar una función que reciba en un vector los datos, y devuelva la información pedida en el problema.

Entrada

La entrada consta de una serie de casos de prueba. Cada caso de prueba tiene dos líneas. En la primera se indica el número de partidos jugados por la selección. En la segunda se indica la diferencia de goles entre los dos equipos. Un valor positivo indica que la selección ganó el partido, un valor cero indica que empató y un valor negativo que perdió.

Salida

Para cada caso de prueba se escribe en una línea el número máximo de partidos seguidos ganados, el número de veces que se ha ganado este número de partidos seguidos y el número de partidos jugados desde que finalizó la última racha ganadora.

Entrada de ejemplo

```
10
2 0 -3 1 1 0 2 1 -1 2
9
-1 3 1 2 0 1 -2 4 3
10
1 1 3 0 1 -1 4 3 1 2
3
-1 0 -1
```

Salida de ejemplo

```
2 2 2
3 1 5
4 1 0
0 0 3
```

Autor: Isabel Pita.

5.1. Objetivos del problema

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

5.2. 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 finaliza en la componente que se trata en esta iteración y cumple la propiedad.
- El número de veces en que se ha conseguido la longitud máxima se guarda en una variable y se actualiza cuando se encuentra una nueva racha con la longitud máxima y cuando se encuentra una racha con una longitud mayor que la que teníamos.
- Para saber hace cuántos partidos que finalizó la última racha debemos guardar la posición en que comienza o finaliza el último segmento máximo encontrado.

5.3. Ideas detalladas.

- *Posibles implementaciones del bucle.*

  Las instrucciones del bucle deben actualizar el valor de todas las variables: longitud máxima hasta este momento, longitud del segmento actual, y posición del final del segmento máximo. 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 = 0; i < v.size(); ++i) {
    if (v[i] > 0) { // El elemento continua la racha
        ++longAct;
        if (longMax < longAct) { // Mejora la racha anterior
            longMax = longAct;
            ultMax = i;
            numVeces = 1;
        }
        else if (longMax == longAct) { // Iguala la racha anterior
            ++numVeces;
            ultMax = i;
        }
    }
    else longAct = 0; // Se rompe la racha
}
```

  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. En esta solución, 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. Es importante también observar que, cuando nos piden contar cuantas veces se encuentra el segmento máximo o encontrar el último segmento tenemos que controlar que ya hemos encontrado al menos un segmento para incrementar el número de segmentos. En otro caso, al estar inicializadas la longitud máxima y la longitud actual ambas a cero, contará como segmentos máximos todos los valores que no cumplen la propiedad.

```
int longMax = 0; int ultMax = 0; // Indica el siguiente al final de la racha
int numVeces = 0; int longAct = 0;
for (int i = 0; i < v.size(); ++i) {
    if (v[i] > 0) { // El elemento continua la racha
```

```
            ++longAct;
        }
        else {
            if (longMax < longAct) { // Mejora la racha anterior
                longMax = longAct;
                ultMax = i;
                numVeces = 1;
            }
            else if (longMax == longAct && longMax > 0) {
             // Iguala la racha anterior y ya se encontro alguna racha
                ++numVeces;
                ultMax = i;
            }
            longAct = 0; // Se rompe la racha
        }
    }
    if (longMax < longAct) { // Mejora la racha anterior
        longMax = longAct;
        ultMax = (int)v.size();
        numVeces = 1;
    }
    else if (longMax == longAct && longMax > 0) { // Iguala la racha anterior
        ++numVeces;
        ultMax = (int)v.size();
    }
```

Ambos bucles tienen coste lineal en el número de elementos del vector. Hay que notar que el caso peor es diferente para cada bucle. En el primer bucle el caso peor ocurre cuando el equipo nacional gana todos los partidos. En este caso se modifica el segmento máximo para cada elemento del vector. En el segundo tipo de bucle el caso peor ocurre cuando la selección va alternando los partidos ganados con los perdidos. Se pierden la mitad de los partidos y en todos ellos se modifican los valores del segmento máximo. Por lo tanto la constante multiplicativa es el doble en el primer bucle respecto al segundo. Sin embargo, si el tiempo no es muy crítico se hace notar que es bastante más sencillo probar la corrección del primer bucle que la del segundo.

5.4. Errores frecuentes.

- Proponer un algoritmo de coste cuadrático respecto al número de elementos.
- En el segundo bucle, olvidarse de comprobar el último segmento después del bucle.

5.5. Coste de la solución

Vemos el coste de la solución que se muestra al final del problema. En ella se utiliza el primer bucle explicado en el apartado anterior. El bucle recorre todos los elementos del vector. En cada iteración del bucle se consultan un elemento del vector y se hacen dos incrementos, dos comparaciones y tres asignaciones, todas ellas operaciones de coste constante. Por lo tanto cada iteración tiene coste constante. Como el bucle hace n iteraciones, siendo n el número de elementos del vector, el coste del bucle es del orden de $\mathcal{O}(n)$.

5.6. 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 en la que se guarda el comienzo o final 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 o final de todos los segmentos máximos encontrados.

El inicio/final del segmento máximo se actualiza cada vez que se encuentra un segmento mayor. Si nos piden el segmento máximo más a la izquierda debemos actualizar el inicio/final cuando encontramos un intervalo estrictamente mayor, si nos piden el segmento más a la derecha actualizamos el inicio/final 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/final cuando se encuentre un intervalo igual.

- *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.

5.7. Implementación en C++.

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

struct tSol {
    int ganados;
    int veces;
    int ultimosPerdidos;
};

// Funcion que resuelve el problema
tSol resolver(std::vector<int> const& v){
    int longMax = 0; int ultMax = -1; int numVeces = 0; int longAct = 0;
    for (int i = 0; i < v.size(); ++i) {
        if (v[i] > 0) { // El elemento continua la racha
            ++longAct;
            if (longMax < longAct) { // Mejora la racha anterior
                longMax = longAct;
                ultMax = i;
                numVeces = 1;
            }
            else if (longMax == longAct) { // Iguala la racha anterior
                ++numVeces;
                ultMax = i;
            }
        }
        else longAct = 0; // Se rompe la racha
    }
    return {longMax, numVeces, (int)v.size() - ultMax - 1};
}

// Resuelve un caso de prueba, lee la entrada  y escribe la respuesta
bool resuelveCaso() {
    // Lectura de los datos de entrada
    int numElem;
    std::cin >> numElem;
    if (!std::cin) return false;
    std::vector<int> v(numElem);
    for (int i = 0; i < numElem; ++i) {
        std::cin  >> v[i];
    }
    // LLamada a la funcion que resuelve el problema
    tSol s = resolver(v);
    // Escribe los resultados
    std::cout << s.ganados << ' ' << s.veces << ' ' << s.ultimosPerdidos << '\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;
}
```

6. Mejorando las carreteras

Mejorando las carreteras

El gobierno está intentando reducir el número de accidentes de carretera. Para ello ha elaborado un ambicioso plan de modernización de las vías en el que se pretende mejorar los tramos más peligrosos. Para detectar estos cuenta con los puntos kilométricos en que han ocurrido accidentes durante los últimos años. Ahora quiere obtener cual es el punto kilométrico en que más accidentes han ocurrido para cada una de las carreteras del país.

Requisitos de implementación.

El problema pide calcular la moda de una colección de valores. En este caso, los valores no están acotados y por lo tanto no podemos utilizar un vector auxiliar para contar las apariciones de cada elemento. En lugar de esto, ordenaremos el vector y buscaremos el segmento máximo con todos los valores iguales.

La función resolver recibirá un vector con los datos leidos de la entrada. Ordenará los elementos y buscará el segmento máximo, devolviendo el valor que se repiten el número máximo de veces. Si existen dos valores que se repiten el número máximo de veces se elije él de más a la izquierda del vector.

Entrada

La entrada consta de una serie de casos de prueba. Cada caso consta de dos líneas, en la primera se muestra el número de accidentes ocurridos, y en la segunda se muestran los puntos kilométricos en que han ocurrido los accidentes. Estos se encuentran ordenados por la fecha en que ocurrió el accidente. El final de casos se marca con el valor -1.

Se garantiza que los valores de la secuencia son números enteros positivos que pueden almacenarse en una variable de tipo `int`. El número de elementos de la secuencia es siempre mayor que cero, pero no está acotado pudiendo ser "muy grande".

Salida

Para cada caso de prueba se escribe en una línea el punto kilométrico en que han ocurrido un mayor número de accidentes. En caso de que haya dos puntos kilométricos con el mismo número de accidentes se elegirá él que esté más cerca del comienzo de la carretera (el menor) ya que resulta más barato arreglar la carretera cuanto más cerca estemos del comienzo y tampoco es cuestión de gastar de más.

Entrada de ejemplo

```
7
5 3 8 5 10 10 5
6
900 50 900 200 50 200
1
12000
-1
```

Salida de ejemplo

```
5
50
12000
```

Autor: Isabel Pita

6.1. Objetivos del problema

- Conocer las diferentes formas de calcular la moda de una colección de elementos. Se conoce como moda al elemento que más veces se repite en la colección.
- Comprobar la importancia de que los elementos de una colección estén ordenados para obtener algoritmos eficientes.

6.2. Ideas generales.

- El problema pide calcular la moda de una colección de valores. Los valores no están acotados y por lo tanto no se puede utilizar un vector auxiliar para contar las apariciones de cada elemento.
- Ordenaremos los valores y después recorreremos el vector ordenado contando cuantas veces aparece cada valor y quedándonos con el máximo.
- Dependiendo de los datos concretos, el uso de un diccionario no ordenado podría dar una solución mejor al problema.

6.3. Algunas cuestiones sobre implementación.

- Se utiliza la función `sort` de la librería `algorithm` para ordenar los datos del vector de menor a mayor:

```
std::sort(v.begin(), v.end());
```

6.4. Ideas detalladas.

- Para calcular el segmento máximo de valores iguales se emplea el algoritmo de segmento máximo. Ver los problemas sobre este algoritmo.
- El vector se recorre con un bucle de izquierda a derecha.
- Se debe guardar información sobre el máximo número de valores consecutivos iguales hasta la posición que se ha recorrido en el vector, asi como del valor. También se guarda información sobre el número de elementos iguales al valor de la posición que se está tratando (valor actual).
- Se recomienda realizar un bucle que en cada vuelta compruebe si el valor actual coincide con el anterior/siguiente. En caso de que coincida, el valor se considera *bueno*, se incrementa el contador de elementos iguales al que se está tratando y se comprueba si este contador ha superado al máximo que se tenía guardado. Si lo ha superado se debe actualizar el número máximo de veces que aparece un valor y el valor de que se trata.
- Si el valor actual no coincide con el valor anterior, el contador de elementos iguales al actual se debe inicializar a 1.

6.5. Coste de la solución

- En la solución del problema hay dos partes.
- Primero se ordena el vector. El algoritmo de ordenación tiene complejidad $\mathcal{O}(n \log n)$ siendo n el número de puntos kilométricos considerados en la entrada de datos.
- Después se ejecuta un bucle para buscar el máximo número de elementos iguales. El bucle da n vueltas, siendo n el número de puntos kilométricos. En cada vuelta se ejecuta un condicional, con asignaciones y comparaciones todas de coste constante. El coste del bucle es $\mathcal{O}(n)$ siendo n el número de puntos kilométricos.
- Por lo tanto el coste del algoritmo es el máximo entre el coste de ordenar y el coste del bucle: $\mathcal{O}(n \log n)$, siendo n el número de puntos kilométricos de la entrada.

6.6. Errores frecuentes.

- No emplear el esquema del algoritmo del segmento máximo propuesto. Si se actualiza el máximo cuando se termina un segmento de valores iguales, debe comprobarse si es necesario actualizarlo cuando se acaba el bucle, ya que puede ocurrir que el segmento máximo de elementos iguales se encuentre al final del vector y en ese caso no se habrá actualizado.

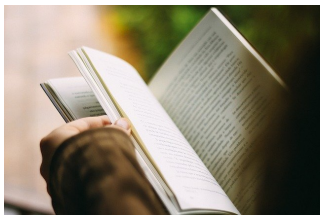
6.7. Implementación en C++.

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

int resolver(std::vector<int> & v){
    // Ordenar el vector
    std::sort(v.begin(), v.end());
    // Buscar el segmento maximo de elementos iguales
    int maxi = 0; int cont = 1; int km = v[0];
    for (int i = 1; i < v.size(); ++i){
        if (v[i] == v[i-1]) {
            ++cont;
            if (cont > maxi) { maxi = cont; km = v[i];}
        }
        else {
            cont = 1;
        }
    }
    return km;
}

bool resuelveCaso() {
    // lectura de los datos
    int numAccidentes;
    std::cin >> numAccidentes;
    if (numAccidentes == -1) return false;
    std::vector<int> v(numAccidentes);
    for (int& i : v) std::cin >> i;
    // Resolver el problema
    int sol = resolver(v);
    // Escribir la solucion
    std::cout << sol << '\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 concentinela
     while (resuelveCaso()) ;

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

```
    return 0;
}
```

7. Intervalos, releyendo un libro.

Releyendo un libro. Versión 1



Cuando tengo un rato libre me gusta coger un libro y releer alguna de las partes que más me gustaron cuando lo leí por primera vez. Normalmente, mientras voy leyendo doy una puntuación entre 0 y 10 a cada página, de forma que cuando quiero volver a leerlo se que páginas me gustaron más. Cero significa que no tengo interés en volver a leer esa página, mientras que un 10 indica que es una de las mejores partes de la obra. Una vez que empiezo a leer en una página, continuo con las páginas siguientes sin saltarme ninguna.

Para asegurarme de que leo la mejor parte he aprendido a estimar el número de páginas que me dará tiempo a leer en el rato que tengo. Siempre busco que las páginas sean lo más interesantes posible. El interés de varias páginas se calcula como la suma del interés de cada página. Si hay varios intervalos igual de interesantes elijo el que esté más avanzado en el libro. Recuerda que las páginas que leo son siempre consecutivas.

Requisitos de implementación.

La función que resuelve el problema debe tener un coste lineal respecto al número de páginas del libro. Puede implementarse más de un bucle, siempre que el coste sea lineal respecto al número de páginas del libro.

Entrada

La entrada consta de una serie de casos de prueba. Cada caso de prueba tiene dos líneas. En la primera se indica el número de páginas del libro (1 < n < 200.000), seguido de la cantidad de páginas que me dará tiempo a leer (1 < L < n). En la segunda línea se indica la puntuación que he dado a cada página del libro (0 <= p <= 10). La entrada finaliza con dos ceros.

Salida

Para cada caso de prueba se escribe la página en que debo empezar a leer el libro para que al leer L páginas empezando en ella, lea siempre al menos una página que tenga la máxima puntuación y además no haya otras L páginas que conteniendo una página con la puntuación máxima me gusten más. Si existen dos bloques de páginas que conteniendo la página que más me gusta, su suma de puntuaciones es igual, elegiré la que se encuentre más avanzada en el libro.

Entrada de ejemplo

```
6 3
0 5 3 4 1 2
7 2
6 4 5 6 0 0 8
5 4
7 7 7 7 7
6 2
5 6 7 6 7 5
0 0
```

Salida de ejemplo

```
1
5
1
3
```

7.1. Objetivos del problema

- Practicar problemas que buscan intervalos de longitud fija que cumplen una propiedad.
- Estudiar el esquema general utilizado en este tipo de problemas y las diferentes variantes que nos podemos encontrar.

7.2. Ideas generales.

- El problema se resuelve con un bucle inicial que recorre el primer intervalo. Este bucle permite inicializar las variable con los valores correspondientes al primer intervalo. Después se implementa un bucle principal donde se recorre el resto del vector. Al avanzar un dato se actualizan los valores de las variables eliminando el primer elemento del intervalo y añadiendo el nuevo elemento.

7.3. Algunas cuestiones sobre implementación.

- El programa debe tener dos bucles en secuencia. En el primero se recorren las primeras L páginas, siendo L la longitud del intervalo que tenemos tiempo de leer. En el segundo se recorren el resto de las páginas.
- En el segundo bucle, los índices deben recorrer todos los posibles intervalos excluyendo el primero. Los intervalos que debe recorrer este bucle son: [1..L+1), [2..L+2), [3..L+3) etc.
- La variable de control del segundo bucle puede ser el extremos izquierdo o el extremo derecho del intervalo. Si elegimos el extremo derecho tendremos el bucle:

```
for (int i = L; i < v.size(); ++i) {
    sumaAct -= v[i-L];
    sumaAct += v[i];
    if (sumaAct >= sumaMax){ // El intervalo es mejor
        posIni = i-L+1;
        sumaMax = sumaAct;
    }
}
```

  Mientras que si elegimos el extremo izquierdo tendremos:

```
for (int i = 0; i < v.size()-L; ++i) {
    sumaAct -= v[i];
    sumaAct += v[i+L];
    if (sumaAct >= sumaMax){ // El intervalo es mejor
        posIni = i+1;
        sumaMax = sumaAct;
    }
}
```

7.4. Ideas detalladas.

- La información necesaria para resolver el problema es la suma del intervalo que estamos considerando y la suma del intervalo máximo que hayamos encontrado, así como su posición inicial.
- En el primer bucle se calcula la suma del primer intervalo.
- A continuación se inicializa la suma máxima con la suma del primer intervalo y la posición inicial a cero.
- En el bucle que recorre el resto del vector restamos el interés de la primera página del intervalo y sumamos el interés de la página siguiente del intervalo.
- Nos piden el intervalo cuya suma sea máxima que se encuentra más a la derecha en el vector. Esto se consigue escogiendo el máximo más a la derecha cuando se pregunta en el bucle principal si la suma actual supera a la máxima ya guardada. Se observa que se modifica el inicio del intervalo si

la suma actual es mayor o igual. Es decir en caso de ser igual se modifica el comienzo del intervalo para que quede guardado el intervalo de más a la derecha.

```
if (sumaAct >= sumaMax){ // El intervalo es mejor
    posIni = i-L+1;
    sumaMax = sumaAct;
}
```

- Si se quisiera el intervalo de más a la izquierda utilizaríamos un menor estricto, de forma que si se encuentra una suma igual no se modifique el comienzo del intervalo.

7.5. Errores frecuentes.

1. No actualizar la suma máxima al acabar el primer bucle.
2. Errores derivados de utilizar un único bucle en el que se mezcla el tratamiento del primer intervalo con el tratamiento general del vector.

7.6. Coste de la solución

El algoritmo tiene dos bucle. El primero se ejecuta tantas veces como el tamaño del intervalo. Las instrucciones del bucle incluyen una comparación, una suma y un incremento, todas ellas de coste constante, por lo tanto el coste de este bucle es el número de vueltas que da el bucle por el coste constante de cada vuelta y está en el orden $\mathcal{O}(p)$, siendo p el tamaño del intervalo.

El segundo bucle recorre todos los elementos del vector desde el final del primer intervalo hasta el final del vector. Por lo tanto el número de vueltas que da es `v.size() - p`. En cada vuelta se realizan 2 comparaciones, 6 operaciones aritméticas y un incremento, todas ellas operaciones de coste constante. Por lo tanto el coste de este bucle está en el orden de $\mathcal{O}(v.size() - p)$, que es equivalente a $\mathcal{O}(v.size())$.

7.7. Modificaciones al problema.

- *Obtener el intervalo de tamaño fijo que incluya algunos valores.*
  - Por ejemplo nos piden el comienzo del intervalo de longitud L más a la izquierda del vector que incluya al menos dos ceros.
  - El esquema general es el visto en este problema.
  - En lugar de llevar calculada la suma del intervalo debemos llevar el número de ceros que tiene el intervalo. Al ir avanzando el intervalo en el segundo bucle comprobaremos si el primer valor del intervalo es un cero y si el nuevo elemento del intervalo es un cero para actualizar el contador del número de ceros.
- *Obtener el intervalo de tamaño fijo que incluya el máximo de los valores del vector.*
  - Por ejemplo nos piden el comienzo del intervalo de longitud L mas a la izquierda del vector que incluya al menos un máximo.
  - El esquema general es el visto en el problema.
  - En primer lugar calcularemos cuál es el valor máximo del vector. El valor máximo puede estar repetido en el vector.
  - LLevaremos información sobre el valor máximo que hemos encontrado y la posición más a la derecha del intervalo en que hay un máximo.
  - En el primer bucle recorreremos el primer intervalo e iremos actualizando la posición del máximo si alguno de los valores del intervalo coincide con él.
  - Una vez finalizado el primer bucle actualizamos los valores del intervalo máximo encontrado. Hay que tener en cuenta que si el intervalo no contiene el valor máximo entonces no se debe actualizar el intervalo máximo.
  - En el bucle general recorremos el resto del vector actualizando la primera y ultima componente del intervalo. Se debe actualizar también la información sobre la posición del valor máximo más a la derecha. Si el intervalo no tiene un máximo entonces no se actualizará la información del intervalo máximo.

7.8. Implementación en C++.

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

int resolver (std::vector<int> const& v, int L){
    // Primer intervalo
    int sumaMax = 0; int sumaAct = 0; int posIni;
    for (int i = 0; i < L; ++i) {
        sumaAct += v[i];
    }
    sumaMax = sumaAct; posIni = 0;}
    // Resto del vector
    for (int i = L; i < v.size(); ++i) {
        sumaAct -= v[i-L];
        sumaAct += v[i];
        if (sumaAct >= sumaMax){ // El intervalo es mejor
            posIni = i-L+1;
            sumaMax = sumaAct;
        }
    }
    return posIni;
}


// Resuelve un caso de prueba.
bool resuelveCaso() {
    // Lectura de los datos
    int numElem, L;
    std::cin >> numElem >> L;
    if (numElem == 0 && L == 0) return false;
    std::vector<int> v(numElem);
    for (int & i : v) std::cin >> i;
    // Resolver el problema
    int pos = resolver(v,L);
    // Escribir la solucion
    std::cout << pos << '\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;
}
```

8. Partición. Viajes a Marte.

Viajes a Marte

Los vuelos a Marte están cada vez más solicitados. Para poder cubrir la demanda, la compañía ha encargado unos vehículos espaciales con más filas de asientos. El constructor ha propuesto disminuir la distancia entre las filas de la parte posterior. De esta forma no será necesario hacer modificaciones en el diseño del aparato. El problema está en que en estas filas sólo podrán sentarse personas *bajitas*. La compañía, sin embargo, no quiere aumentar el presupuesto, por lo que ha decidido pedir a los viajeros su altura cuando compran el billete y a la hora de embarcar llama primero a los más *bajos* para que ocupen las filas posteriores.

Tu misión es dividir a los viajeros entre *bajos* y *altos* en función de la altura que te digan. Si la altura de algún viajero coincide con la altura establecida se le considerará *bajo*. Debes también indicar el número de viajeros *bajos* del vuelo.

Entrada

La entrada consta de una serie de casos de prueba. Cada caso de prueba consta de $n+1$ líneas. En la primera se indica el número n de viajeros y la altura establecida para diferenciarlos. En cada una de las líneas siguientes aparece el nombre de un viajero y su altura.

El número de viajeros es mayor o igual que 0 y menor que 150.000. Los nombres de los viajeros son cadenas de caracteres sin blancos. La altura son números enteros positivos.

Salida

Para cada caso de prueba se escriben dos líneas. La primera comienza con `Bajos:` seguido de los nombres de los viajeros *bajos*. La segunda comienza con `Altos:` seguido de los nombres de los viajeros *altos*.

Para poder comparar tu salida con la del juez los nombres de los viajeros se mostrarán en orden alfabético, aunque esto no debe alterar el algoritmo empleado para resolver el problema, sino solo la salida de los datos.

Entrada de ejemplo

```
5 170
Rosa 150
Olga 175
Maria 180
Lucia 170
Ana 150
4 180
Daniel 190
Jose 200
Antonio 190
Roberto 200
2 150
Beatriz 140
Gonzalo 120
```

Salida de ejemplo

```
Bajos: Ana Lucia Rosa
Altos: Maria Olga
Bajos:
Altos: Antonio Daniel Jose Roberto
Bajos: Beatriz Gonzalo
Altos:
```

Autor: Isabel Pita

8.1. Objetivos del problema

- Practicar el algoritmo de partición cuando los datos del vector se dividen en dos partes.
- Este algoritmo es muy importante ya que se utiliza en el algoritmo de *quick sort* para ordenar secuencias de elementos.

8.2. Ideas generales.

- En el problema nos piden modificar el vector, de forma que en la parte izquierda queden todos los elementos menores o iguales que un valor dado y en la parte derecha todos los mayores. Debemos calcular también la posición que divide al vector.
- LLevaremos dos índices. El primer índice marca las posiciones del vector tales que a la izquierda del índice todos los valores son menores o iguales que el valor dado. El segundo índice marca las posiciones del vector tales que todos los valores a su derecha son mayores que el valor dado. Los valores entre los dos índices son los que todavía no se han tratado y por lo tanto pueden ser mayores o menores que el valor dado.

8.3. Algunas cuestiones sobre implementación.

- El vector se pasa por referencia, ya que se debe modificar en la función.
- En los datos de entrada, los nombres de los viajeros son cadenas de caracteres sin blancos por lo que se pueden leer utilizando `std::cin` en una variable de tipo `std::string`.
- En el vector debemos guardar el nombre y la altura de cada viajero. Esto se puede hacer con:
  - Un vector de pares `std::vector<std::pair<std::string, int>>`. En este caso, al ordenar el vector con la función `sort`, se utiliza el operador menor definido para el tipo `pair`. Este operador ordena los valores según el orden de la primera componente y si el valor de la primera componente es igual por el valor de la segunda.
  - Un `struct` con dos campos que representen los dos valores:

    ```
    struct tInfo {
      std::string nombre;
      int altura;
    };
    ```

    En este caso se debe sobrecargar el operador menor para el tipo `tInfo` definido en el `struct` para que la función `sort` lo pueda utilizar.

    ```
    bool operator< (tInfo d1, tInfo d2) {
      return (d1.nombre < d2.nombre) || (d1.nombre == d2.nombre && d1.altura < d2.altura);
    }
    ```

  - Para que el juez pueda corregir el problema se pide que la salida se muestre ordenada por orden alfabético. Sin embargo esto no debe afectar al algoritmo implementado. Por ello, una vez que la función devuelve el vector con los elementos menores a la izquierda de la posición `p` devuelta por la función y los elementos mayores desde la posición `p` hasta el final del vector se ordenan ambas mitades con la función `sort` de la librería `algorithm`.

    ```
    std::sort(v.begin(), v.begin()+p);
    std::sort(v.begin()+p, v.end());
    ```

8.4. Ideas detalladas.

- Se inicializa un índice a la primera posición del vector y otro a la última posición del vector.
- Se implementará un solo bucle que se ejecuta mientras no se crucen los dos índices.
- En cada vuelta del bucle se comprueba si el elemento correspondiente al primer índice está bien colocado, en caso afirmativo se incrementa este índice, si no, si el elemento correspondiente al segundo índice está bien colocado se decrementa este índice, y si no es que ambos elementos están más colocados. En este caso se intercambian los valores y se mueven los dos índices.

8.5. Errores frecuentes.

No se conocen

8.6. Coste de la solución

- En la implementación que se muestra al final del ejercicio y que sigue las ideas presentadas anteriormente, el bucle principal está controlado por dos índices. En cada vuelta del bucle se modifica uno de los índices o los dos. Por lo tanto el número máximo de vueltas que da el bucle coincide con el número de elementos del vector.
- En cada vuelta del bucle se realizan dos comparaciones, un intercambio de variables y cuatro incrementos o decrementos. Todas estas operaciones tienen coste constante (el intercambio es de dos pares con tipos `std::string` e `int`, el tipo `std::string` se intercambia con coste constante debido a que soporta la semántica de movimiento y el tipo `int` se intercambia tambien con coste constante por ser un tipo básico). Por lo tanto el coste del algoritmo esta en el orden $\mathcal{O}(n)$ siendo n el número de viajeros del vuelo.

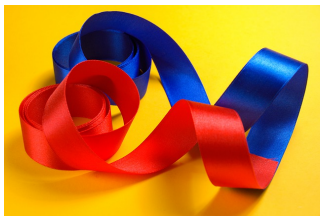
8.7. Implementación en C++.

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

using psi = std::pair<std::string, int>;  // Nombre del viajero y su altura

int particion (std::vector<psi> & v, int altura) {
    int p = 0, q = (int)v.size()-1;
    while (p <= q) {
        if (v[p].second <= altura)  ++p; // elemento del indice izquierdo correcto
        else if (v[q].second > altura) --q;  // elemento del indice derecho correcto
        else {  // Ambos elementos fuera de sitio
            std::swap(v[p],v[q]);
            ++p; --q;
        }
    }
    return p;
}


// Resuelve un caso de prueba, leyendo de la entrada la
// configuracion, y escribiendo la respuesta
bool resuelveCaso() {
    int numViajeros, altura;
    std::cin >> numViajeros;
    if (!std::cin) return false;
    std::cin >> altura;
    std::vector<psi> v(numViajeros); // nombre y altura
    for (psi & n : v) {
        std::cin  >> n.first >> n.second;
    }
    int p = particion(v, altura);
    // Ordena cada parte para la salida de datos
    std::sort(v.begin(), v.begin()+p);
    std::sort(v.begin()+p, v.end());
    // Escribe la primera mitad
    if (p > 0) std::cout << v[0].first;
    for (int i = 1; i < p; ++i){
```

```
        std::cout << ' ' << v[i].first ;
    }
    std::cout << '\n';
    // Escribe la segunda parte
    if (p < v.size()) std::cout << v[p].first;
    for (int i = p+1; i < v.size(); ++i){
        std::cout << ' ' << v[i].first ;
    }
    std::cout << '\n';
    std::cout << "---\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 casos ilimitados
    while (resuelveCaso()) {} //Resolvemos todos los casos

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

    return 0;
}
```

9. Partición. Cintas de colores.

Cintas de colores



Tengo una caja llena de cintas de colores rojas, azules y verdes. Las he estado midiendo y ahora quiero ordenarlas por colores, para poder saber facilmente las longitudes que tengo de cada color.

Dada una lista de todas las cintas con sus longitudes, sin ningún orden, debes ordenarlas de forma que al principio de la lista aparezcan las cintas azules, a continuación las verdes y por último las rojas. Dentro de cada grupo no es necesario que las cintas aparezcan en un orden determinado.

Requisitos de implementación.

Se debe implementar una función que reciba un vector con las cintas identificadas por su color y su longitud y modifique el vector para dejar al principio las cintas azules, en el medio las cintas verdes y por último las cintas rojas. La función devolverá dos índices indicando donde empiezan y terminan las cintas verdes. El coste de la función debe ser lineal en el número de cintas.

Debe emplearse el algoritmo de partición con dos índices.

Entrada

La entrada consta de una serie de casos de prueba. Cada caso de prueba consta de dos líneas. En la primera se indica el número n de cintas. En la siguiente se muestra el color de cada cinta seguido de su longitud.

El número de cintas es mayor o igual que 0 y menor que 300.000. Los colores de las cintas se identifican por su primer carácter: `a` para las cintas azules, `v` para las cintas verdes y `r` para las cintas rojas. Las longitudes son números enteros positivos.

Salida

Para cada caso de prueba se escriben tres líneas. La primera comienza con `Azules:` seguido de las longitudes de las cintas azules. La segunda comienza con `Verdes:` seguido de las longitudes de las cintas verdes. La tercera comienza con `Rojas:` seguido de las longitudes de las cintas rojas.

Para poder comparar tu salida con la del juez las longitudes se mostrarán en orden de menor a mayor, aunque esto no debe alterar el algoritmo empleado para resolver el problema, sino solo la salida de los datos.

Entrada de ejemplo

```
6
v 5 r 1 a 3 a 7 v 3 r 6
3
r 4 r 7 r 1
5
a 6 a 2 v 1 v 3 a 5
4
r 5 r 7 a 9 a 3
```

Salida de ejemplo

```
Azules: 3 7
Verdes: 3 5
Rojas: 1 6
Azules:
Verdes:
Rojas: 1 4 7
Azules: 2 5 6
Verdes: 1 3
Rojas:
Azules: 3 9
Verdes:
Rojas: 5 7
```

Autor: Isabel Pita

9.1. Objetivos del problema

- Practicar el algoritmo de partición cuando los datos del vector se dividen en tres partes.
- Este algoritmo es muy importante ya que se utiliza en el algoritmo de *quick sort* para ordenar secuencias de elementos.

9.2. Ideas generales.

- En el problema nos piden modificar el vector, de forma que en la parte izquierda queden todas las cintas de color azul, en la parte del centro las cintas de color verde y por último las cintas de color rojo. Debemos calcular también las posiciones que separan las diferentes partes
- Al dividir el vector en tres partes es necesario mantener tres índices. El primer índice marca las posiciones del vector tales que a la izquierda del índice todos los valores son cintas de color azul. El segundo índice marca las posiciones del vector tales que todas las cintas desde el primer índice hasta el segundo son de color verde. El tercer índice marca las posiciones del vector tales que todas las cintas a su derecha son de color rojo. Los valores entre el segundo y tercer índice son los que todavía no se han tratado y por lo tanto pueden ser cintas de cualquier color.

9.3. Algunas cuestiones sobre implementación.

- El vector se pasa por referencia, ya que se debe modificar en la función.
- En el vector debemos guardar el color y la longitud de cada cinta. Esto se puede hacer con:
  - Un vector de pares `std::vector<std::pair<char, int>>`. En este caso, al ordenar el vector con la función `sort`, se utiliza el operador menor definido para el tipo `pair`. Este operador ordena los valores según el orden de la primera componente y si el valor de la primera componente es igual por el valor de la segunda.
  - Un `struct` con dos campos que representen los dos valores:

    ```
    struct tInfo {
      char color;
      int longitud;
    };
    ```

    En este caso se debe sobrecargar el operador menor para el tipo `tInfo` definido en el `struct` para que la función `sort` lo pueda utilizar.

    ```
    bool operator< (tInfo d1, tInfo d2) {
      return (d1.longitud < d2.longitud);
    }
    ```

- Para que el juez pueda corregir el problema se pide que la salida se muestre ordenada por la longitud de las cintas. Sin embargo esto no debe afectar al algoritmo implementado. Por ello, una vez que la función devuelve el vector con las cintas azules a la izquierda de la posición `p` (valor de retorno de la función), las cintas verdes entre la posición `p` y la posición `q` (valor de retorno de la función) y las cintas de color rojo desde la posición `q` hasta el final del vector se ordena cada parte con la función `sort` de la librería `algorithm`.

    ```
    std::sort(v.begin(), v.begin()+p);
    std::sort(v.begin()+p, v.begin()+1+q);
    std::sort(v.begin()+1+q, v.end());
    ```

- El índice `q` indica la última componente de color verde. Por ello si todas las cintas son rojas este índice toma el valor -1. Al ordenar la parte verde y la parte roja debemos escribir el final y el principio de la parte a ordenar respectivamente como

    ```
    std::sort(v.begin()+p, v.begin()+1+q);
    std::sort(v.begin()+1+q, v.end());
    ```

  El motivo es que la expresión `v.begin() + q` si el valor de `q` es -1 es erróneo.

9.4. Ideas detalladas.

- Se inicializan dos índices `i` y `k` a la posición cero y un índice `q` a la última posición del vector. Los valores a la izquierda de `i` serán de color azul, los valores entre `i` y `k` serán los de color verde y los valores a la derecha de `q` serán los de color rojo. Los valores entre `k` y `q` son los valores que todavía no se han tratado.
- Se implementará un solo bucle que se ejecuta mientras no se crucen los dos índices `k` y `q`.
- En cada vuelta del bucle se comprueba si el elemento correspondiente al índice `k` es de color verde, en caso afirmativo está bien colocado y se incrementa el índice `k`. Si es de color azul se intercambia con el elemento de la posición `p` y se incrementan los índices `p` y `q`. Si es de color rojo se intercambia con el elemento de la posición `q` y se decrementa el índice `q`. Observar que en este último caso no se incrementa el valor de `k` porque no se ha comprobado el color del elemento que estaba en la posición `q` y por lo tanto no se puede considerar que ya está tratado.

9.5. Errores frecuentes.

- El índice `q` indica la última componente de color verde. Por ello si todas las cintas son rojas este índice toma el valor -1. Al ordenar la parte verde y la parte roja debemos escribir el final y el principio de la parte a ordenar respectivamente como

  ```
  std::sort(v.begin()+p, v.begin()+1+q);
  std::sort(v.begin()+1+q, v.end());
  ```

  El motivo es que la expresión `v.begin() + q` si el valor de `q` es -1 es erróneo, por lo tanto le sumamos primero uno al valor de `v.begin()` y de esta forma luego se puede restar. .

9.6. Coste de la solución

- En la implementación que se muestra al final del ejercicio y que sigue las ideas presentadas anteriormente, el bucle principal está controlado por dos índices `k` y `q`. En cada vuelta del bucle se modifica uno de los índices. Por lo tanto el número máximo de vueltas que da el bucle coincide con el número de elementos del vector.
- En cada vuelta del bucle se realizan comparaciones, intercambio de variables e incrementos o decrementos. Todas estas operaciones tienen coste constante (el intercambio es de dos pares con tipos `std::char` e `int`), por lo tanto el coste del algoritmo esta en el orden $\mathcal{O}(n)$ siendo n el número de cintas.

9.7. Implementación en C++.

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

using psi = std::pair<char, int>;

void particion (std::vector<psi> & v, int &p, int &q) {
    p = 0; q = (int)v.size()-1; int k = 0;
    while (k <= q ) {
        if (v[k].first == 'v') ++k;
        else if ( v[k].first == 'a')  {
            std::swap(v[p],v[k]);
            ++p; ++k;
        }
        else {
            std::swap(v[q],v[k]);
            --q;
        }
```

```
    }
}


bool resuelveCaso() {
    // lectura de los datos de entrada
    int numCintas;
    std::cin >> numCintas;
    if (!std::cin) return false;
    std::vector<psi> v(numCintas); // nombre y altura
    for (psi & n : v) {
        std::cin  >> n.first >> n.second;
    }
    // resuelve el problema
    int p, q;
    particion(v, p, q);
    // Ordena cada parte para la salida de datos
    std::sort(v.begin(), v.begin()+p);
    std::sort(v.begin()+p, v.begin()+1+q); // Debe ser 1+q para evitar v.begin()-1 cuando q vale
    std::sort(v.begin()+1+q, v.end());
    // Escribe las cintas azules
    std::cout << "Azules:";
    for (int i = 0; i < p; ++i){
        std::cout << ' ' << v[i].second ;
    }
    std::cout << '\n';
    // Escribe la segunda parte
    std::cout << "Verdes:";
    for (int i = p; i <= q; ++i){
        std::cout << ' ' << v[i].second ;
    }
    std::cout << '\n';
    // Escribe la tercera parte
    std::cout << "Rojas:";
    for (int i = q+1; i < v.size(); ++i){
        std::cout << ' ' << v[i].second ;
    }
    std::cout << '\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 casos ilimitados
    while (resuelveCaso()) ; //Resolvemos todos los casos

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

    return 0;
}
```