Cuaderno de problemas
Fundamentos de algorítmia.

Algoritmos iterativos

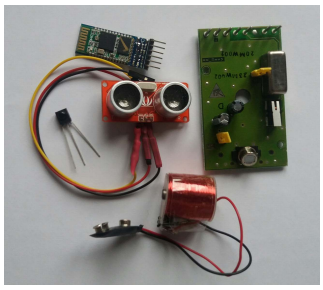
Prof. Isabel Pita

19 de octubre de 2020

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 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.

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 el índice `valoresBuenos` y se pasa 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`.

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 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.

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 $\mathcal{O}(n)$.

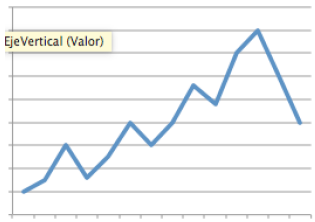
1.7. Implementación en C++.

```
#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.

2.5. Errores frecuentes.

- Realizar una implementación de coste cuadrático respecto al número de elementos del vector. Si para cada elemento calculamos el valor máximo de las componentes anteriores a él, recorremos el vector, hasta la posición que estamos tratando, en cada iteración y por lo tanto el coste será cuadrático en el número de elementos del vector.

2.6. Modificaciones al problema.

- Si la propiedad pedida sobre los datos del vector se refiere a los valores de la derecha del dato, entonces debemos recorrer el vector de derecha a izquierda.
- La propiedad pedida sobre los datos puede referirse al mínimo de los valores anteriores (o posteriores) o a la suma de los valores.

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); // posicion del primer maximo
    for (int k = 1; k < v.size(); ++k) {
        if (v[k] > PIBMax){ // Encontrado nuevo maximo
            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, 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.

3.1. Objetivos del problema

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

3.3. 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.

3.4. 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.

3.5. 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 $\mathcal{O}(n)$.

3.6. 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.

3.7. 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;
}
```

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?. Por ahora las preguntas se refieren siempre a dos tipos de piedras.

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. El número de piedras es al menos uno y como máximo 200.000.

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
```

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 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.

4.3. Algunas cuestiones sobre implementación.

- Para representar las piedras preciosas se puede utilizar un tipo enumerado

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

- Para leer los valores se sobrecarga el operador extractor para el tipo enumerado anterior:

```
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;
}
```

- El vector debe guardar valores del tipo `piedrasPreciosas`. Y la lectura de los valores del vector se hará con el for basado en iteradores.

```
std::vector<piedrasPrecisas> v(numElem);
for (piedrasPreciosas i : v) std::cin >> i;
```

4.4. Ideas detalladas.

- La información necesaria para resolver el problema es : el número de intervalos que tienen las gemas requeridas, y el número de gemas de cada tipo que hay en el intervalo.
- En el primer bucle se calculan las gemas de cada tipo del primer intervalo.
- Si el primer intervalo cumple la propiedad pedida tenemos que actualizar el número de intervalos que se han encontrado.
- En el bucle que recorre el resto del vector, comprobaremos la gema que queda a la izquierda del intervalo y si es una de las pedidas la eliminaremos del número de estas gemas. A continuación comprobaremos la gema que entra a formar parte del intervalo por la parte derecha y si es una de las pedidas aumentaremos el contador de esta gema. Por último se comprueba si el nuevo intervalo cumple la propiedad.

4.5. Errores frecuentes.

1. No incrementar el contador del número de intervalos si el primer intervalo cumple la propiedad.
2. Errores en la inicialización de los valores derivados de utilizar un único bucle en el que se mezcla el tratamiento del primer intervalo con el tratamiento general del vector.

4.6. 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 dos comparaciones y dos incrementos, ambas instrucciones 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 6 comparaciones y 5 incrementos, todos ellos 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 en el caso peor $(p == 1)$ a $\mathcal{O}(v.size())$.

4.7. Modificaciones al problema.

- *Obtener el intervalo de tamaño fijo de suma máxima.*

  La solución es semejante a la descrita en este problema. En lugar de llevar el número de gemas de cada tipo se guarda la suma del intervalo. Al desplazar el intervalo, se resta el valor de la izquierda y se suma el valor de la derecha.

4.8. 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;
}

// Funcion 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; // Numero de intervalos que tienen las gemas requeridas
    int num1 = 0; // Numero de gemas del primer tipo
    int num2 = 0; // Numero 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 posicion
    // Elimina la gema de la izquierda
        if (v[j-p] = t1) --num1;
        else if (v[j-p] = t2) --num2;
        // Anade 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
// configuracion, 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 (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;
}
```

5. 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 300.000. 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

5.1. Objetivos del problema

- Practicar el algoritmo de partición.
- Este algoritmo es muy importante ya que se utiliza en el algoritmo de *quick sort* para ordenar secuencias de elementos.

5.2. Ideas generales.

- En el problema nos piden modificar el vector, de forma que en la parte izquierda queden todos los elementos menores 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 que el 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 dado.

5.3. 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, sino, si el elemento correspondiente al segundo índice está bien colocado se decrementa este índice, y sino es que ambos elementos están más colocados. En este caso se intercambian los valores y se mueven los dos índices.

5.4. 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`.
- 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());
```

5.5. Errores frecuentes.

No se conocen

5.6. 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.

5.7. 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;
}
```