Cuaderno de problemas
Fundamentos de algorítmia.

Divide y vencerás

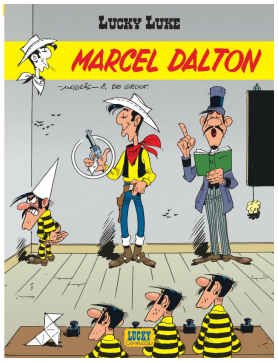
Prof. Isabel Pita

23 de noviembre de 2020

Índice

1. Lucky Lucke en busca del culpable.

Lucky Lucke en busca del culpable



Lucky Lucke es el vaquero más rápido del oeste. Alguien ha robado el oro de la cámara blindada del banco y Lucky Lucke debe encontrar el mejor algoritmo posible para dar con el culpable. Lucky Lucke conoce la altura del bandido porque después de cometer el atraco dejó su figura marcada en la parte superior de la puerta del banco.

El sheriff le ha mandado por mail un fichero con la altura de todos los sospechosos del oeste ordenados de menor a mayor. ¿Puedes ayudarle a encontrar al culpable? .

Requisitos de implementación.

Se implementarán dos funciones recursivas diferentes: La primera *buscarIz* encuentra el primer sospechoso de la lista con la altura pedida, la segunda, *buscarDr* encuentra la posición en la lista del último sospechoso con la altura pedida.

La función *resuelveCaso*, llamará a la función *buscarIz* para calcular la primera posición en que aparece la altura pedida y llamará a la función *buscarDr* para obtener la última posición de la altura pedida en la lista.

Entrada

La entrada consta de una serie de casos de prueba. Cada caso comienza con el número de sospechosos seguido de la altura del bandido. A continuación en una línea aparecen las alturas de todos los sospechosos ordenados de menor a mayor.

El número de sospechosos es un valor 0 <= N <= 1.000.000 y las alturas de los sospechosos son valores enteros positivos menores de 250.

Salida

Para cada caso de prueba se muestra en un línea la posición que ocupa en la lista de sospechosos el primero con la altura buscada seguido de la posición del último sospechoso con la altura dada. Si sólo existe un sospechoso se mostrará solo su posición en la lista y si no exsite ninguno se escribirá *NO EXISTE*.

Entrada de ejemplo

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

Salida de ejemplo

```
3
0 2
NO EXISTE
```

1.1. Objetivos del problema.

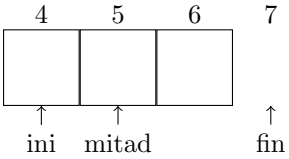
- Aprender a realizar búsquedas eficientes en vectores ordenados.
- Aprender a descartar la mitad del vector al realizar una comparación con el elemento central.

1.2. Ideas generales.

- El problema se resuelve aplicando la técnica divide y vencerás, aprovechando la información que podemos inferir del hecho de que el vector esté ordenado.
- Se compara el elemento buscado con el elemento de la posición central del vector. Hay tres posibilidades:
  - Si el elemento buscado es menor que el elemento central, el elemento sólo se puede encontrar en el lado izquierdo del vector, ya que todos los elementos del lado derecho son mayores que el elemento buscado. En un vector ordenado de menor a mayor todos los elementos que se encuentren a la derecha de uno dado son mayores o iguales que él
  - Si el elemento buscado es mayor que el elemento central, el elemento solo puede estar en el lado derecho del vector ya que los elementos de la parte izquierda del vector son todos menores que el elemento central.
  - Si el elemento buscado es igual al elemento central, el elemento está en el vector.
- El vector del problema puede tener elementos repetidos. Debemos buscar el elemento mas a la izquierda del vector que sea igual al elemento buscado y el elemento más a la derecha del vector igual al elemento buscado. Cada búsqueda se realiza con una función recursiva distinta.
- En el caso de buscar uno de los extremos del intervalo de valores iguales al buscado no podemos considerar el caso en que el elemento central coincida con el valor buscado, ya que este no tiene porque ser el extremo del intervalo. Debemos seguir buscando por el lado pedido sin eliminar el elemento central del vector.

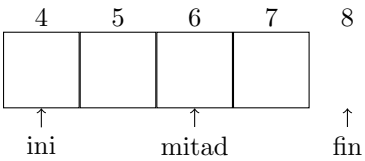
1.3. Algunas cuestiones sobre implementación.

- La función recursiva debe tener dos parámetros que indiquen el inicio y final del vector. Para ser consistentes con las funciones de la librería `STL`, marcaremos el final del vector en la posición siguiente a la última.
- El elemento central se puede calcular como `mitad = (ini + fin) / 2` o como `mitad = (ini + fin - 1) / 2`. Cuando el número de elementos del vector es impar ambas fórmulas producen el mismo elemento central.

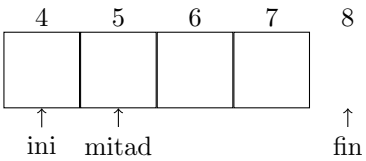


Sin embargo, cuando el número de elementos del vector es par, la primera fórmula nos da la mitad izquierda con un elemento más y la segunda fórmula la mitad derecha con un elemento más. Esto puede tener importancia en algunos problemas.

`mitad = (ini+fin)/2`



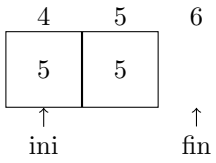
`mitad = (ini+fin-1)/2`



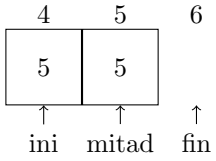
1.4. Errores frecuentes.

- No considerar suficientes casos base y que el algoritmo no termine.
- *Que el vector no disminuya al realizar la llamada recursiva para algún tamaño del vector.*

  Por ejemplo, queremos calcular la posición del elemento mas a la izquierda con valor 5. Hemos ido disminuyendo el tamaño del vector hasta quedar en la situación:



  Si calculamos la mitad del vector con la fórmula: `(ini + fin) / 2`, tendremos:

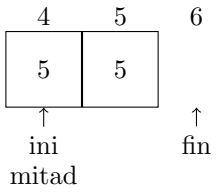


  Como el elemento de la mitad del vector coincide con el elemento buscado, debemos seguir buscando en la parte izquierda del vector y el elemento central debe estar en la parte en que seguimos buscando ya que puede ser el de mas a la izquierda. La llamada recursiva es:

```
binarySearchIz(v, elem,ini, mitad + 1);
```

  El tamaño del vector no disminuye y la recursión no acaba. Veredicto del juez: `TIME LIMIT`.

  Fijémonos que pasa si utilizamos la otra fórmula: `(mitad = (ini + fin - 1) / 2;`. La posición central será:



  Al hacer la llamada recursiva `binarySearchIz(v, elem,ini, mitad + 1);`, el vector tiene únicamente el valor de la posición 4. Por lo tanto disminuye el tamaño y la llamada recursiva acaba correctamente.

1.5. Coste de la solución.

La función `binarySearchIz` tiene dos casos base, ambos con coste constante porque son comparaciones. En el caso recursivo se ejecuta una única llamada con la mitad de los elementos del vector de entrada. Las instrucciones que acompañan a la llamada recursiva tienen coste constante. Por lo tanto la recurrencia es:

$$T(n) = \begin{cases} c_1 & n == 1 \\ T(\frac{n}{2}) + c_2 & n > 1 \end{cases}$$

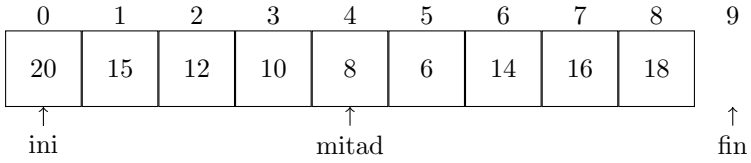
siendo n el número de elementos del vector que viene dado por `fin - ini`.

El despliegue puede consultarse en la parte del cuaderno de problemas correspondiente a la complejidad de algoritmos recursivos.

Se obtiene un coste $\mathcal{O}(\log n)$ siendo n el número de elementos del vector.

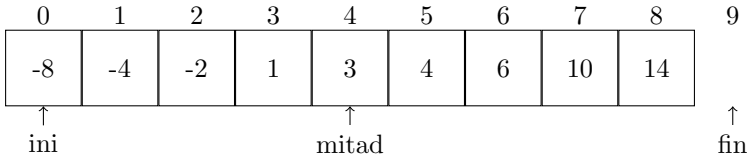
1.6. Modificaciones al problema.

- *Dado un vector cuyos valores son crecientes/decrecientes hasta una determinada posición y a partir de ella son decrecientes/crecientes, obtener la posición en que se produce el cambio.*



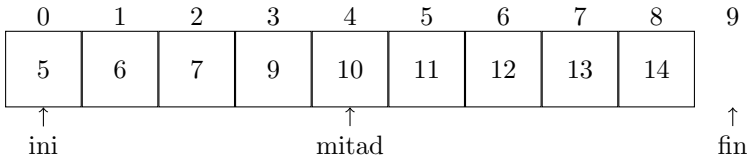
Comprobamos si el valor de la mitad está en la parte creciente o decreciente del vector. Si está en la parte decreciente el valor del mínimo debe estar en la parte derecha del vector ya en la parte izquierda los valores serán más grandes que el de la mitad. Si el valor de la mitad está en la parte creciente de los valores, el mínimo debe estar en la parte izquierda del vector, ya que hacia la derecha los valores serán cada vez más grandes.

- *Dado un vector ordenado de números enteros, todos ellos diferentes, comprobar si algún valor se encuentra en su posición.*



El vector está ordenado. Si el valor de la mitad del vector es mayor que la posición que ocupa, ningún valor en la parte derecha puede estar en su posición, ya que al no haber valores repetidos los valores crecerán como mínimo de uno en uno, igual que las posiciones. Por lo tanto, si el valor del centro era mayor que su posición todos los valores de la derecha serán también mayores que su posición. Si el valor de la mitad del vector es menor que su posición, todos los valores de la izquierda del vector serán menores que su posición, ya que aunque los valores disminuyan de uno en uno, las posiciones también lo hacen.

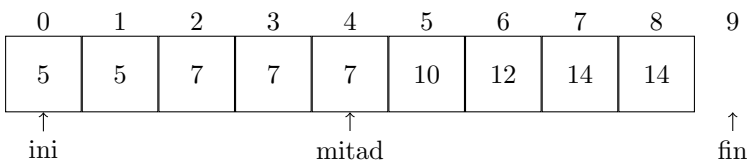
- *Dado un vector ordenado de valores consecutivos entre los que falta un único valor, encontrar el valor que falta.*



Si la distancia entre el valor de la mitad y el valor del inicio, coincide con la distancia entre la posición de la mitad y la posición del inicio no puede faltar ningún valor en la parte izquierda. Si la distancia entre los valores es mayor que la distancia entre las posiciones entonces es que falta el valor.

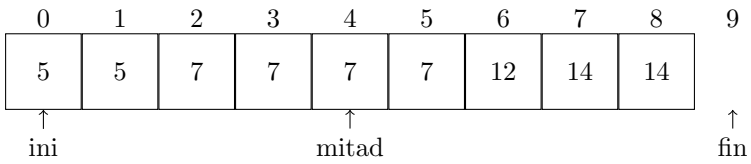
En este problema se facilita la implementación si se garantiza que en las llamadas recursivas el valor que falta siempre será mayor que el inicio y menor que el fin.

- *Contar cuantas veces aparece un valor dentro de una colección ordenada.*



Buscaremos la primera vez que aparece y la última.

- *Contar cuantos valores diferentes hay en un intervalo.*



Este problema no sigue exactamente el esquema de la búsqueda binaria ya que requiere hacer las dos llamadas recursivas una a cada mitad del vector. Se mejora el coste evitando las llamadas recursivas a las partes del vector cuyo inicio y final sea el mismo elemento.

1.7. Implementación.

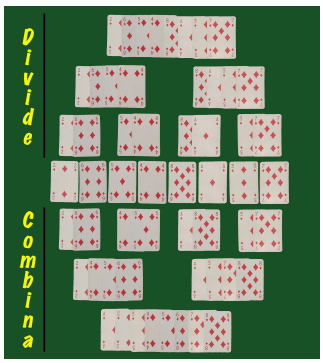
```
// Funcion que busca la posicion mas a la izquierda del elemento
template <class T>
int binarySearchIz (std::vector<T> const& v, T elem, int ini, int fin) {
    if (ini ≥ fin) return ini;  // vector vacio
    else if (ini+1 = fin) return ini; // vector 1 elemento
    else { // Vector de dos o mas elementos
        int mitad = (ini + fin -1) / 2;
        if (v[mitad] < elem) return binarySearchIz(v,elem,mitad+1,fin);
        else return binarySearchIz(v,elem,ini,mitad+1);  // incluye la mitad
    }
}

// Funcion que busca la posicion mas a la derecha del elemento
template <class T>
int binarySearchDr (std::vector<T> const& v, T elem, int ini, int fin) {
    if (ini ≥ fin) return ini; // vector vacio
    else if (ini + 1 = fin) return ini; // vector un elemento
    else { // vector 2 o mas elementos
        int mitad = (ini + fin) / 2;
        if (elem < v[mitad]) return binarySearchDr(v,elem,ini,mitad);
        else return binarySearchDr(v,elem,mitad,fin);  // Incluye el elemento mitad
    }
}

bool resuelveCaso(){
    int numelem, elem;
    std::cin >> numelem;
    if (!std::cin) return false;
    std::cin >> elem;
    std::vector<int> v(numelem);
    for (int& n : v) std::cin >> n;
    // Busca en el lado izquierdo
    int posIz = binarySearchIz(v,elem,0,(int)v.size());
    if (posIz = v.size() || v[posIz] != elem)
        std::cout << "NO EXISTE\n";
    else {
        // Si existe el elemento busca en el lado derecho
        int posDr = binarySearchDr(v,elem,0,(int)v.size());
        if (posIz = posDr) std::cout << posIz << '\n';
        else std::cout << posIz << ' ' << posDr << '\n';
    }
    return true;
}
```

2. Ordenación por mezclas. (*mergesort*)

Ordenar con mergesort



El algoritmo *mergesort* fue inventado por el matemático John von Neumann en 1945. El algoritmo emplea la técnica *divide y vencerás* consiguiendo una complejidad en tiempo del orden de *O(n log(n))* siendo n el número de elementos a ordenar. La complejidad en espacio es del orden *O(n)*. Se puede implementar obteniendo ordenación estable, lo que significa que los valores iguales mantienen el orden en que estuviesen en el vector original.

Comparado con otros algoritmos de ordenación, el algoritmo *mergesort* presenta una complejidad semejante al algoritmo *heapsort* aunque este último tiene una complejidad *O(1)* en espacio, frente a la complejidad *O(n)* que tiene el algoritmo *mergesort*. En el caso peor el algoritmo *Quicksort* tiene una complejidad peor que *mergesort*, aunque en el caso medio consigue mejores resultados. En general, *mergesort* es la mejor opción para ordenar listas enlazadas, ya que en este caso se puede conseguir una complejidad *O(1)* en espacio. Además otros métodos de ordenación más eficientes sobre estructuras de acceso aleatorio no se comportan tan bien sobre este tipo de estructuras.

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 valores a ordenar (0 <= N <= 100.000). En la segunda línea se muestran estos valores (-10^9 <= N <= 10^9).

Salida

Para cada caso de prueba se escribe en una línea los valores del vector ordenados.

Entrada de ejemplo

```
10
4 5 2 7 4 9 2 6 7 3
7
-5 -6 -2 -1 -4 -5 -8
1
1
0

3
6 9 8
2
7 6
```

Salida de ejemplo

```
2 2 3 4 4 5 6 7 7 9
-8 -6 -5 -5 -4 -2 -1
1

6 8 9
6 7
```

Autor: Isabel Pita

2.1. Objetivos del problema.

- Conocer el método de ordenación por mezclas: *mergesort*.
- Practicar el método de resolución de problemas *divide y vencerás*. En concreto, practicar el método de resolución recursiva que divide el vector en dos partes aproximadamente iguales, resuelve cada una de las partes y por último combina los resultados de ambas partes.
- Practicar el uso de índices para referirnos a partes de un vector.

2.2. Ideas generales.

- La solución es recursiva.
- Ordenaremos la parte izquierda y la parte derecha del vector para después mezclar las dos partes ya ordenadas.
- Al ordenar cada parte aplicamos el mismo algoritmo de *mergesort* a un número menor de elementos.
- El algoritmo termina cuando la parte del vector considerada es vacío o tiene un elemento ya que en estos casos el vector ya está ordenado.

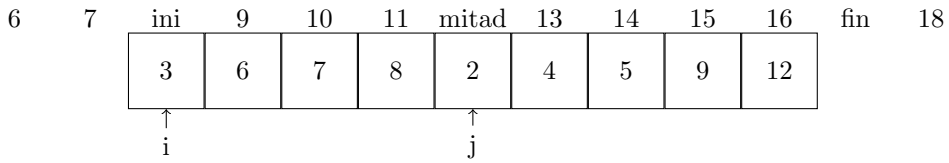
2.3. Ideas detalladas.

- *Cómo resolver el problema.*

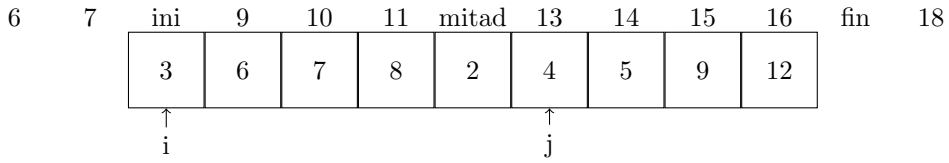
  1. Primero se resuelve el problema para la parte izquierda del vector utilizando el mismo algoritmo de *mergesort*. El resultado es que la parte izquierda del vector está ordenada.
  2. A continuación se resuelve el problema para la parte derecha del vector. El resultado es que la parte derecha del vector está ordenada.
  3. Por último se mezclan las dos partes ya ordenadas, lo que da como resultado el vector completo ordenado.

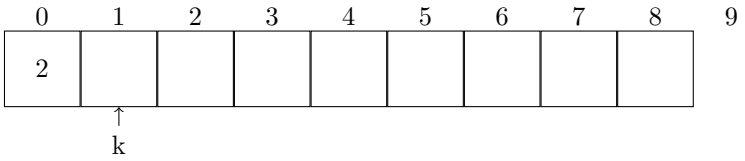
- *Cómo mezclar dos partes ordenadas de un vector para obtener un vector completamente ordenado, con coste lineal en el número de elementos del vector.*

  1. Primera solución: utilizar la función `inplace_merge` de la librería `algorithm`.
  2. Segunda solución: Implementar una función que realice la mezcla:
     a) La mezcla se realiza con un algoritmo iterativo en un vector auxiliar.
     b) Se declara un índice al comienzo de cada parte a mezclar.

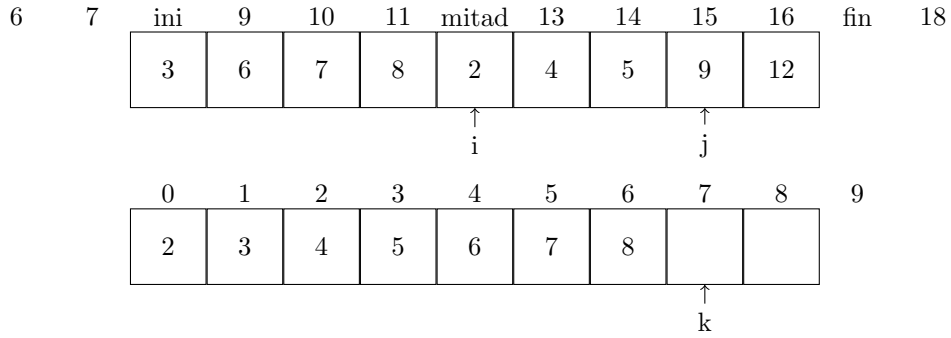


     c) Se recorre el vector hasta que alguno de los dos índices alcanza el final de su parte. Si el elemento de la primera parte es menor o igual que él de la segunda se copia en el vector auxiliar este elemento y se incrementa su índice. En otro caso se copia el elemento de la segunda parte y se incrementa su índice.





d) Cuando termina de mezclarse alguna de las dos partes, se realiza un bucle para copiar la otra parte en el vector auxiliar. Como se ignora que parte terminó se realiza dos bucles, uno para cada parte. Sólo se ejecutará uno de los bucles, en el otro la condición será falsa y se saltará su ejecución.



e) Por último se copia el vector auxiliar en la zona del vector original que se estaba mezclando.

2.4. Algunas cuestiones sobre implementación.

- Cada llamada recursiva se realiza sobre una parte distinta del vector original. Copiar la parte del vector que queremos ordenar en un nuevo vector es muy ineficiente. Por ello la llamada a la función se realiza con el vector completo, que pasamos constante por referencia para evitar realizar la copia y dos índices que indican el principio y final de la parte del vector que estamos tratando.
- Existen varias formas de indicar el principio y final de la parte del vector que estamos considerando. Dependiendo de cúal se utilice varía la instrucción que calcula el punto medio del vector y los parámetros utilizados en las llamadas. Puede también tener efecto sobre los casos base que se deben considerar.

  1. Indicar la posición del primer elemento y la posición siguiente al último elemento que se quiere considerar. Esta es la forma que se utiliza en las funciones de la librería STL que utilizan iteradores. Por ello es la forma que se utilizará en los problemas implementados en este cuaderno.
  2. Indicar la posición del primer elemento y la posición del último elemento que se quiere considerar.
  3. Indicar la posición del primer elemento y el número de elementos que tiene la parte del vector que se considera. Esta forma no se recomienda ya que suele dar lugar a un número mayor de errores en la implementación.

2.5. Errores frecuentes.

No se conocen.

2.6. Coste de la solución.

- Para calcular el coste de la función recursiva `mergesort` planteamos la recurrencia correspondiente a la implementación realizada.
- En la implementación Secc. 2.8, el caso base es vacío por lo que su coste es constante. En el caso recursivo se ejecutan dos llamadas recursivas cada una con la mitad de los elementos y se llama a la función mezclar que tiene coste lineal en el número de elementos del vector. La recurrencia es:

$$T(n) = \begin{cases} c_1 & n == 0 \;||\; n == 1 \\ 2T(\frac{n}{2}) + n & n > 1 \end{cases}$$

siendo n el número de elementos del vector que viene dado por `fin - ini`.

- El coste es del orden de $\mathcal{O}(n \log n)$
- El despliegue puede consultarse en la parte del cuaderno de problemas correspondiente a la complejidad de algoritmos recursivos.

2.7. Modificaciones al problema.

Este problema permite muchas modificaciones.

- Probar que un vector cumple una cierta propiedad recursiva. En estos casos, cada llamada recursiva debe devolver la información necesaria para comprobar la propiedad pedida. Al combinar los resultados, se comprueba la propiedad pedida para la parte del vector correspondiente a una llamada con los datos obtenidos de cada una de las partes.
  - Suficientemente disperso.
  - Degradado de una imagen.
  - Vector parcialmente ordenado.
- Construcción de soluciones desde el caso base
  - Jugando a las canicas
  - Torneo de tenis
- Algoritmos más complejos que necesiten información de las llamadas recursivas y que el vector resultado esté ordenado.
  - BattleStar Galactica.
  - Par de puntos mas cercanos.

2.8. Implementación.

```
// Funcion recursiva que resuelve el problema
// ini - posicion inicial de la parte del vector que se considera
// fin - Posicion siguiente al ultimo elemento de la parte del vector que se considera.
template <typename T, typename Comp = std::less<T>>
void mergesort(std::vector<T> & v, int ini, int fin, Comp ord = Comp()) {
    if (ini + 1 < fin) { // Vector de mas de un elemento
        int mitad = (ini + fin) / 2;
        // Ordena cada parte
        mergesort(v,ini,mitad);
        mergesort(v,mitad,fin);
        // mezcla
        std::inplace_merge(v.begin()+ini,v.begin()+mitad,v.begin()+fin);
    }
}

// Funcion principal
template <typename T, typename Comp = std::less<T>>
void mergesort (std::vector<T> & v, Comp ord = Comp()) {
    mergesort(v,0,(int)v.size());
}

// Para resolver cada caso de entrada
bool resuelveCaso() {
```

```
    // Lectura de los datos
    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];
    // Ordena el vector
    mergesort(v);
    // Escribe los valores ordenados
    if (!v.empty()) std::cout << v[0];
    for (int i = 1; i < numElem; ++i)
        std::cout << ' ' << v[i];
    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()) {} //Resolvemos todos los casos

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

    return 0;
}
```

3. Valores suficientemente dispersos.

Suficientemente disperso

El dueño del casino está muy preocupado por que los jugadores de ruleta puedan llegar a detectar que el crupier controla el número que saca en cada tirada. Todas las noches estudia la lista de números que han salido ese día y comprueba que sean *suficientemente dispersos*.

El dueño se conforma con que la diferencia entre el primer valor que se saca y el último sea mayor o igual que una cantidad K. Además comprueba que los datos que salieron en la primera mitad de la noche y los datos que salieron en la segunda mitad sean tambien suficientemente dispersos, esto es que la diferencia entre el primer valor y el último tanto de la primera mitad como de la segunda sea mayor que K y además cada una de sus mitades sea también suficientemente dispersa.

Estudia únicamente secuencias con un número de elementos que sea potencia de dos para poder dividirlas siempre en dos partes iguales. Considera que un sólo valor siempre es suficientemente disperso.

Entrada

La entrada consta de una serie de casos de prueba. Cada caso de prueba consta de dos líneas. En la primera se muestra el número de tiradas que se considera y el valor de dispersión que se precisa. En la segunda se muestran los números que han salido en cada tirada

El número de valores de cada caso de prueba es una potencia de 2. El valor de dispersión es un entero positivo mayor que cero. Los valores de cada tirada son números entre 0 <= N <= 1.048.576

Salida

Para cada caso de prueba se escribe en una línea *SI* si el vector está suficientemente disperso y *NO* si no lo está.

Entrada de ejemplo

```
4 3
6 1 3 9
4 3
3 10 12 14
8 4
20 2 0 4 14 8 5 10
```

Salida de ejemplo

```
SI
NO
SI
```

3.1. Objetivos del problema.

- Practicar el método de resolución de problemas *divide y vencerás*. En concreto, practicar el método de resolución recursiva que divide el vector en dos partes aproximadamente iguales, resuelve cada una de las partes y por último combina los resultados de ambas partes.

3.2. Ideas generales.

- El problema se resuelve aplicando la técnica divide y vencerás, siguiendo el esquema del algoritmo *mergesort*.
- Comprobar si la mitad izquierda y la mitad derecha del vector son suficientemente dispersas. Para ello realizar llamadas recursivas.
- A continuación se obtiene el resultado de la función, combinando los resultados obtenidos de las llamadas recursivas con la comprobación de que la parte del vector considerada en la llamada a la función cumple la propiedad.
- El algoritmo termina cuando la parte del vector considerada tiene un elemento, ya que en estos casos el vector es suficientemente disperso.

3.3. Ideas detalladas.

Se consideran suficientes las ideas generales de este problema para su resolución.

3.4. Algunas cuestiones sobre implementación.

- *El vector tiene al menos un elemento.* El enunciado del problema indica que el número de valores de cada caso de prueba es una potencia de dos. Por ello el vector tiene al menos un elemento ($2^0 = 1$).

3.5. Errores frecuentes.

- Utilizar la misma variable para guardar el resultado de las dos llamadas recursivas, perdiendo los datos de la primera llamada cuando se ejecuta la segunda.
- No comprobar que cada mitad del vector también debe cumplir la propiedad.

3.6. Coste de la solución.

- Para calcular el coste de la función recursiva `sufDispersa` planteamos la recurrencia correspondiente a la implementación realizada.
- En la implementación Sec. 3.8, la función `sufDisperso` tiene un caso base con coste constante. En el caso recursivo se ejecutan dos llamadas, cada una con la mitad de los elementos del vector de entrada. Las instrucciones que acompañan a la llamada recursiva tienen coste constante. Por lo tanto la recurrencia es:

$$T(n) = \begin{cases} c_1 & n == 1 \\ 2T(\frac{n}{2}) + c_2 & n > 1 \end{cases}$$

  siendo n el número de elementos del vector que viene dado por `fin - ini`.
- El despliegue puede consultarse en la parte del cuaderno de problemas correspondiente a la complejidad de algoritmos recursivos.
- Se obtiene un coste $\mathcal{O}(n)$ siendo n el número de elementos del vector.

3.7. Modificaciones al problema.

- En el problema resuelto, la propiedad que se pide depende de que cada parte del vector sea suficientemente dispersa (resultado de las llamadas recursivas) y de comparar dos valores del vector. Por lo tanto la propiedad se puede calcular en tiempo constante. Sin embargo, en muchos problemas, la propiedad depende, no sólo de que las dos partes cumplan la propiedad sino también de otros valores que se obtienen realizando alguna operación sobre todos los elementos del vector o de alguna parte. En estos casos debemos evitar realizar esta operación, devolviendo los valores como parte del resultado de las llamadas recursivas de cada parte. Por ejemplo, supongamos la propiedad: *la diferencia entre la suma de los elementos de la mitad izquierda del vector y la suma de los elementos de la mitad derecha debe ser mayor que una cierta cantidad constante por el número de elementos considerados*. Debemos evitar realizar la suma de los elementos de cada parte en cada llamada recursiva. Para ello la función recursiva devolverá dos valores: un valor booleano que indique si el vector cumple la propiedad y un valor entero que sea la suma del vector:

  ```
  struct tSol {
     bool cumple;
     int suma;
  };
  ```

  Se considera como caso base el vector de un elemento. En este caso el valor de la suma es el valor del elemento y se considera que se cumple la propiedad.

  En el caso recursivo se realiza una llamada a la función para resolver cada mitad del vector y luego se componen los resultados devueltos por cada llamada, comprobando que cada parte cumple la propiedad y que el vector completo también la cumple.

  ```
  int m = (ini+fin)/2;
  // LLamadas recursivas
  tSol sIz = resolver(v,ini,m);
  tSol sDr = resolver(v,m,fin);
  // Calculo de la solucion a partir de las soluciones de las llamadas recursivas
  tSol solucion;
  solucion.suma = sIz.suma + sDr.suma;
  solucion.cumple = sIz.cumple && sDr.cumple &&
                                  K*(fin-ini) ≤ std::abs(sIz.suma - sDr.suma);
  return solucion;
  ```

  Observad que el número de elementos del vector se calcula como `fin - ini`.

3.8. Implementación.

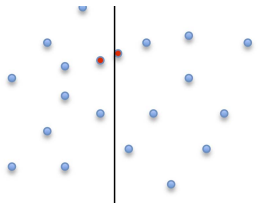
```
// Funcion que resuelve el problema
// v.size() es potencia de dos
bool sufDisperso(std::vector<int> const& v, int ini, int fin, int K) {
    if (ini + 1 = fin) return true; // Un elemento
    else { // Vector de 2 o mas elementos
        int m = (ini+fin)/2;
        bool sIz = sufDisperso(v,ini,m,K);
        bool sDr = sufDisperso(v,m,fin,K);
        return sIz && sDr && K ≤ std::abs(v[ini] - v[fin-1]);
    }
}

// Resuelve un caso de prueba, leyendo de la entrada la
// configuracion, y escribiendo la respuesta
bool resuelveCaso() {
    int num;
    std::cin >> num;
    if (!std::cin) return false;
    int K;
    std::cin >> K;
```

```
    std::vector<int> v(num);
    for (int & n: v) std::cin >> n;
    if (sufDisperso(v,0,(int)v.size(),K)) std::cout << "SI\n";
    else std::cout << "NO\n";
    return true;
}
```

4. Par de puntos mas cercanos.

Par de puntos más cercanos



Dados una serie de puntos en un plano, queremos saber cual es la distancia entre la pareja de puntos que se encuentren más cerca.

Requisitos de implementación.
El problema debe resolverse con la técnica de divide y vencerás.
La distancia debe calcularse en una variable de tipo *float* y no se truncará hasta la instrucción de salida por pantalla, momento en que se hará un casting: *(int) distancia*.

Entrada

La entrada consta de una serie de casos de prueba. Cada caso consta de *N* líneas. En la primera se muestra el número de puntos que se consideran. En las *N-1* restantes se dan las coordenadas x e y de cada punto.
Las coordenadas de los puntos son números enteros.

Salida

Para cada caso de prueba se escribe en una línea la parte entera de la mínima distancia entre todos los puntos.

Entrada de ejemplo

```
2
5 0
2 6
4
5 8
2 7
10 4
7 20
5
7 1
5 9
3 6
7 10
4 2
```

Salida de ejemplo

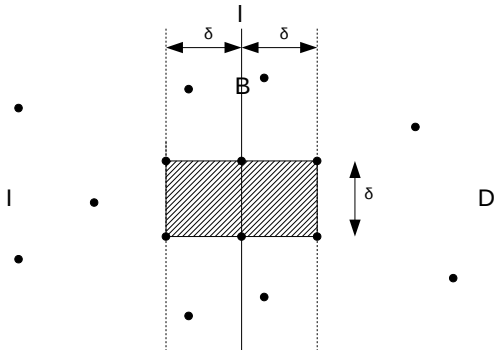
```
6
3
2
```

4.1. Objetivos del problema.

- Practicar la técnica de divide y vencerás para resolver un problema. En concreto practicar el método de resolución recursiva que divide el vector en dos partes aproximadamente iguales, resuelve cada una de las partes y por último combina los resultados de ambas partes.
- Practicar una operación de combinación de los resultados de las llamadas recursivas compleja. Las llamadas recursivas devolverán varios resultados.
- Aprender a devolver un vector ordenado como resultado de dos llamadas recursivas.

4.2. Ideas generales.

- El problema se resuelve aplicando la técnica divide y vencerás, siguiendo el esquema del algoritmo *mergeSort*.
- Se ordenan los puntos del vector por la coordenada x y se divide el vector en dos mitades. Sea m la coordenada x del punto en la mitad del vector. La parte izquierda del vector contiene los puntos de menor coordenada x que m y la parte derecha los puntos de coordenada x mayor que m. Se hace una llamada recursiva con cada mitad del vector que calcula los dos puntos más cercanos de la mitad del vector con que se realiza la llamada. Sean $p11$ y $p12$ los dos puntos más cercanos de la mitad izquierda y $p21$ y $p22$ los dos puntos mas cercanos de la mitad derecha del vector. Seleccionamos el par con menor distancia d.
- Para calcular la distancia entre los puntos de una mitad y los puntos de la otra, se tienen en cuenta sólo los puntos que están a una distancia menor que d de m. Los puntos más alejados no pueden tener una distancia menor que d de los puntos de la otra mitad.
- Según se observa en la figura, de los puntos que están a distancia menor que d de m sólo los 5 puntos con coordenada y mayor que la que estamos considerando pueden estar a una distancia del punto menor que d. Ordenamos los puntos por la coordenada y (antes de filtrarlos) y después calculamos la distancia de cada punto únicamente con los 5 siguientes.



4.3. Ideas detalladas.

- Deben declararse dos casos base. Uno para un vector con dos elementos y otro para un vector con tres elementos. El enunciado garantiza que siempre habrá al menos dos elementos. Cuando se divide un vector de tres elementos da lugar a un vector de dos y un vector de un elemento. Como el caso base de un elemento no tiene sentido consideramos el caso base de tres elementos. Esto es suficiente ya que al dividir el vector de 4 elementos se producen dos vectores de dos, que se considerarán en el caso base. Al dividir un vector de 5 elementos se producen uno de dos y otro de tres que también pueden ser tratados por los caso base.
- La estructura de la parte recursiva de la función es:

  1. Calcular el punto medio.

2. Hacer las dos llamadas recursivas, almacenando sus soluciones en variables distintas.
3. Obtener los dos puntos mas cercanos entre las dos soluciones dadas por las dos llamadas recursivas.
4. Mezclar las dos mitades del vector, ordenando las componentes por la coordenada y.
5. Filtrar la lista obtenida dejando únicamente los puntos que están a una distancia de la coordenada x del punto medio menor que la distancia mínima del punto 3.
6. Calcular la distancia entre cada punto y sus 5 puntos siguientes y quedarse con la mínima.

4.4. Algunas cuestiones sobre implementación.

- Definir un tipo `tPunto` que almacene las coordenadas x e y de un punto. Sobrecargar los operadores de lectura y escritura de un punto, y definir comparadores por la coordenada x y por la coordenada y.

```
struct tPunto {
    int x, y;
};

std::istream& operator>> (std::istream& in, tPunto & p) {
    in >> p.x >> p.y;
    return in;
}

std::ostream& operator<< (std::ostream & out, tPunto const& p) {
    out << "< " << p.x << ',' << p.y << " >";
    return out;
}

struct comparadorx {
    bool operator() (tPunto const& p1, tPunto const& p2) {
        return p1.x < p2.x;
    }
};

struct comparadory {
    bool operator() (tPunto const& p1, tPunto const& p2) {
        return p1.y < p2.y;
    }
};
```

- Para calcular la distancia entre dos puntos, se puede utilizar la función `sqrt` de la librería `cmath`.

```
float distancia (tPunto const& p1, tPunto const& p2) {
    return std::sqrt((p1.x-p2.x)*(p1.x-p2.x)+(p1.y-p2.y)*(p1.y-p2.y));
}
```

- Definir un tipo `tSol` con los valores que debe devolver la función: un par de puntos y la distancia entre ellos.

```
struct tSol {
    std::pair<tPunto,tPunto> puntos;
    float distancia;
};
```

El tipo `pair` se encuentra definido en la librería `utility`. Podemos acceder a la primera componente del par con el atributo `first` y al segundo con `second`.

El siguiente ejemplo crea una variable de tipo `tSol`, con los puntos `<5,3>` y `<6,4>` y una distancia 20 y luego los muestra por pantalla. Debemos tener sobrecargado el operador `<<` para el tipo `tPunto`:

```
tSol s = {{5,3},{6,4},20};
std::cout << s.first << ' ' << s.second << '\n';
```

- Para intercambiar el valor de dos variables se puede utilizar la función `std::swap` de la librería `utility`.

  El siguiente ejemplo intercambia el valor de las componentes `i` y `j` de un vector `v`:

  ```
  std::swap(v[i],v[j]);
  ```

- Para ordenar el vector se puede utilizar la función `sort` de la librería `algorithm`.

  Para ordenar las componentes por la coordenada x, se utiliza el comparador `comparadorx` definido anteriormente.

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

- Para mezclar las dos mitades del vector y que quede ordenado por la componente y, se puede utilizar la función `inplace_merge` de la librería `algorithm` con el comparador `comparadory` definido anteriormente.

- Para filtrar los datos del vector `v` se puede utilizar la función `copy_if`. Esta función recibe el inicio y final del vector a filtrar, el vector en que se copiaran los elementos filtrados y un comparador en el que se define la forma de filtrarlos.

  El comparador debe tener dos atributos, que representan la distancia horizontal al medio y la coordenada x del punto medio. Definimos un constructor para el comparador para dar valor a estos atributos:

  ```
  struct comparadorDist {
      bool operator() (tPunto const& p) {
          return distancia(p,{xmitad,p.y}) ≤ distmin;
      }
      comparadorDist() {}
      comparadorDist(float d, int x) {distmin = d; xmitad = x;}
      float distmin;
      int xmitad;
  };
  ```

  Una vez definido el comparador la llamada a la función `copy_if` es:

  ```
  comparadorDist cc {s.distancia,xmitad};
  std::copy_if(v.begin()+ini, v.begin()+fin, std::back_inserter(filtrado),
                            cc);
  ```

  Donde la función `std::back_inserter` devuelve un iterador a la estructura que se le pase por parámetro que permite insertar elementos por el final de la estructura.

  En lugar de utilizar un objeto comparador se podría utilizar una lambda-expressión. Esto permite utilizar una expresión más sencilla en la llamada a la función `copy_if`.

  ```
  std::copy_if(v.begin()+ini, v.begin()+fin, std::back_inserter(filtrado),
     [s,xmitad](tPunto const& p){return distancia(p,{xmitad,p.y}) ≤ s.distancia;});
  ```

  El último parámetro es una lambda-expressión que nos permite definir una función online. Entre corchetes se indican las variables del programa que se utilizarán en la expresión. En nuestro caso se utilizan dos variables definidas en el programa: `s` que es de tipo `tSol` y se necesita para conocer la mínima distancia hasta este momento y `xmitad` que es un entero con la coordenada x del punto medio. Entre llaves se expresan los parámetros del predicado. En nuestro caso es un único parámetro que define el punto que queremos saber si esta a menor distancia del punto medio. Por último entre llaves se escribe el cuerpo de la función. En nuestro caso se compara la distancia entre el punto dado y el punto medio con la distancia mínima que tiene el programa calculada hasta este momento.

4.5. Errores frecuentes.

- Olvidarse de ordenar los elementos por la coordenada x antes de llamar a la función recursiva.
- No ordenar los elementos por la coordenada y en los casos base. Si no se ordenan, al realizar la mezcla no obtendremos un vector ordenado por la ordenada y, y el argumento de que es suficiente con mirar los 5 puntos siguientes a uno dado en el vector será falso.

4.6. Coste de la solución.

- Para calcular el coste de la función recursiva `parMasCercano` planteamos la recurrencia correspondiente a la implementación realizada.
- En la implementación 4.7, la función `parMasCercano` tiene dos casos base, ambos con coste constante porque utilizan instrucciones de coste constante, y las funciones que utilizan tienen también coste constante. La función `sort` se utiliza sobre un vector de tres elementos por lo que su coste es constante. En el caso recursivo se ejecutan dos llamadas, cada una con la mitad de los elementos del vector de entrada. Las instrucciones que acompañan a la llamada recursiva tienen coste lineal en el número de elementos del vector, ya que incluyen la mezcla de las dos partes del vector, y el filtrado de los datos. Por último, el doble bucle para calcular las distancias entre los elementos filtrados tiene coste líneal en el número de elementos filtrados, porque el bucle interno está acotado por 7 vueltas.

  Por lo tanto la recurrencia es:

$$T(n) = \begin{cases} c_1 & n == 1 \\ 2T(\frac{n}{2}) + n & n > 1 \end{cases}$$

  siendo n el número de elementos del vector que viene dado por `fin - ini`.
- El despliegue puede consultarse en la parte del cuaderno de problemas correspondiente a la complejidad de algoritmos recursivos.
- Se obtiene un coste $\mathcal{O}(n \log(n))$ siendo n el número de elementos del vector.

4.7. Implementación.

```
const int MAX_PUNTOS = 5;

struct tPunto {
    int x, y;
};

// Sobrecarga de los operadores de lectura y escritura del tipo punto
std::istream& operator>> (std::istream& in, tPunto & p) {
    in >> p.x >> p.y;
    return in;
}

std::ostream& operator<< (std::ostream & out, tPunto const& p) {
    out << "< " << p.x << ',' << p.y << " >";
    return out;
}

struct comparadorx {
    bool operator() (tPunto const& p1, tPunto const& p2) {
        return p1.x < p2.x;
    }
};

struct comparadory {
    bool operator() (tPunto const& p1, tPunto const& p2) {
        return p1.y < p2.y;
    }
};

struct tSol {
    std::pair<tPunto,tPunto> puntos;
    float distancia;
```

```
};

float distancia (tPunto const& p1, tPunto const& p2) {
    return std::sqrt((p1.x-p2.x)*(p1.x-p2.x)+(p1.y-p2.y)*(p1.y-p2.y));
}

struct comparadorDist {
    bool operator() (tPunto const& p) {
        return distancia(p,{xmitad,p.y}) ≤ distmin;
    }
    comparadorDist() {}
    comparadorDist(float d, int x) {distmin = d; xmitad = x;}
    float distmin;
    int xmitad;
};

tSol parMasCercano (std::vector<tPunto> & v, int ini, int fin) {
    if (ini+2 = fin) { // dos elementos
        if (v[ini].y > v[ini+1].y) std::swap(v[ini],v[ini+1]);
        return {{v[ini],v[ini+1]},distancia(v[ini],v[ini+1])};
    }
    else if (ini+3 = fin) { // 3 elementos
        std::sort(v.begin()+ini, v.begin()+ini+3, comparadory());
        float d1 = distancia(v[ini],v[ini+1]);
        float d2 = distancia(v[ini+1],v[ini+2]);
        float d3 = distancia(v[ini],v[ini+2]);
        if (d1 < d2 && d1 < d3){
            return {{v[ini],v[ini+1]},d1};
        }
        else if (d2 < d1 && d2 < d3)
            return {{v[ini+1],v[ini+2]},d2};
        else return {{v[ini],v[ini+2]},d3};
    }
    else { // mas de tres elementos
        int mitad = (ini + fin) / 2;
        int xmitad = v[mitad].x;
        // LLamadas recursivas
        tSol iz = parMasCercano(v,ini,mitad);
        tSol dr = parMasCercano(v,mitad,fin);
        tSol s = (iz.distancia < dr.distancia)? iz:dr;
        // mezcla ordenada por y
        std::inplace_merge(v.begin()+ini,v.begin()+mitad,v.begin()+fin,comparadory());
        // filtrar lista
        std::vector<tPunto> filtrado;
        comparadorDist cc {s.distancia,xmitad};
        std::copy_if(v.begin()+ini, v.begin()+fin, std::back_inserter(filtrado), cc);
        // Calcular las distancias de un punto con los 5 siguientes
        int i = 0;
        while (i < filtrado.size()) {
            int count = 0; int j = i+1;
            while (j < filtrado.size() && count < MAX_PUNTOS) {
                if (distancia(filtrado[i],filtrado[j]) < s.distancia) {
                    s.puntos = {filtrado[i],filtrado[j]};
                    s.distancia = distancia(filtrado[i],filtrado[j]);
                }
                ++j; ++count;
            }
            ++i;
        }
        return s;
    }
}
```

```
// Resuelve un caso de prueba, leyendo de la entrada la
// configuracon, y escribiendo la respuesta
bool resuelveCaso() {
    int nElems;
    std::cin >> nElems;
    if (!std::cin) return false;
    std::vector<tPunto> v(nElems);
    for (int i = 0; i < nElems; ++i) {
        std::cin >> v[i];
    }
    std::sort(v.begin(),v.end(),comparadorx());
    tSol s = parMasCercano(v,0,(int)v.size());
    //std::cout << s.puntos.first << ' ' << s.puntos.second << ' ' << s.distancia << '\n';
    std::cout <<  (int)s.distancia << '\n';
    return true;
}
```

5. Ordenación rápida: *quicksort.*

Método de ordenación rápida. Quicksort

El algoritmo de ordenación rápida fue desarrollado por Tony Hoare en 1959. Si se implementa de forma adecuada puede ser 2 o 3 veces más rápido que *mergesort* o *heapsort*.

Se implementa con la técnica de *divide y vencerás*. El vector se divide en dos partes más pequeñas. A continuación se ordena cada una de estas partes empleando el mismo algoritmo. Si las dos partes son aproximadamente del mismo tamaño se consigue una complejidad del orden de $O(n\ log(n))$. En cambio, si una de las partes es muy pequeña la complejidad del algoritmo es del orden de $O(n^2)$.

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 valores a ordenar (0 <= N <= 100.000). En la segunda línea se muestran estos valores (-10^9 <= N <= 10^9).

Salida

Para cada caso de prueba se escribe en una línea los valores del vector ordenados.

Entrada de ejemplo

```
10
4 5 2 7 4 9 2 6 7 3
7
-5 -6 -2 -1 -4 -5 -8
1
1
0

3
6 9 8
2
7 6
```

Salida de ejemplo

```
2 2 3 4 4 5 6 7 7 9
-8 -6 -5 -5 -4 -2 -1
1

6 8 9
6 7
```

Autor: Isabel Pita

5.1. Objetivos del problema.

- Conocer el método de ordenación rápida: *quicksort*.
- Practicar la técnica de resolución de problemas *divide y vencerás*. En concreto practicar el método de resolución recursiva que primero separa los datos del vector en dos partes y luego resuelve cada una de ellas.
- Conocer la diferencia entre la complejidad en el caso medio y en el caso peor.

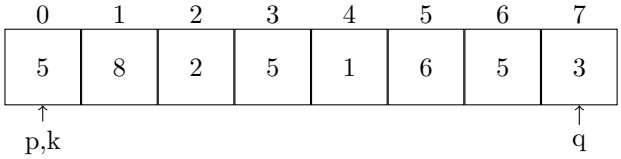
5.2. Ideas generales.

- La solución es recursiva.
- Se preparan los datos colocando en la parte izquierda del vector los datos menores que uno dado, llamado *pivote* y en la parte derecha los datos mayores. El *pivote* quedará en el sitio que le corresponde en la ordenación.
- Por costumbre se suele tomar como *pivote* el primer elemento del vector.
- Por último se ordena la parte izquierda que contiene los datos menores que el pivote y la parte derecha que contiene los datos mayores que el pivote.
- Si aproximadamente la mitad de los datos del vector son menores que el pivote y la otra mitad son mayores el vector se divide aproximadamente por la mitad y la complejidad del algoritmo está en $\mathcal{O}(n \log(n))$ siendo n el número de elementos del vector. Si hay muy pocos elementos menores/mayores que el *pivote* entonces las dos partes del vector estarán muy descompensadas y el coste está en $\mathcal{O}(n^2)$, siendo n el número de elementos del vector.

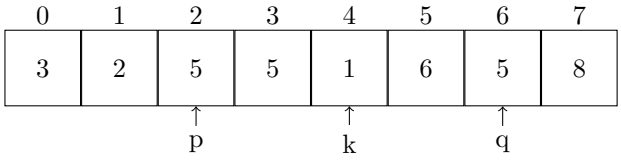
5.3. Ideas detalladas.

- *Cómo partir el vector dejando en el lado izquierdo los elementos menores que el pivote y en el lado derecho los valores mayores.*
  - El algoritmo deja en la parte izquierda del vector los elementos menores que el pivote, en la parte central todos los elementos iguales al pivote y en la parte derecha los elementos mayores que el pivote.
  - Se recorre el vector con tres índices. Los dos primeros recorren el vector de izquierda a derecha. Uno de ellos, p, indica la posición siguiente a los elementos menores que el pivote, otro, k, indica la posición siguiente a los elementos iguales que el pivote. El tercer índice, q recorre el vector desde la derecha e indica la posición anterior a los elementos mayores que el pivote.

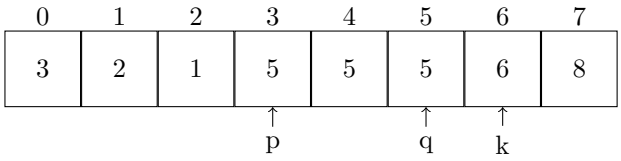
    Posición inicial:



    Posición intermedia



    Posición final



- Este algoritmo es más eficiente que el algoritmo que divide el vector en dos partes, dejando los elementos iguales al pivote en la posición que tuvieran en el vector original, debido a que las llamadas recursivas tendrán menos elementos cuando el vector original tenga muchos elementos repetidos.

5.4. Algunas cuestiones sobre implementación.

Ver Sec. 2.8

5.5. Errores frecuentes.

No se conocen.

5.6. Coste de la solución.

El caso base es vacío por lo que su coste es constante. En el caso recursivo se ejecuta la función partición que tiene coste lineal en el número de elementos del vector, y dos llamadas recursivas. Si el vector se divide aproximadamente por la mitad, cada llamada recursiva se realiza con la mitad de los elementos y la recurrencia es:

$$T(n) = \begin{cases} c_1 & n == 1 \\ 2T(\frac{n}{2}) + n & n > 1 \end{cases}$$

siendo n el número de elementos del vector que viene dado por `fin - ini`.

El despliegue puede consultarse en la parte del cuaderno de problemas correspondiente a la complejidad de algoritmos recursivos.

Se obtiene un coste $\mathcal{O}(n \log(n))$ siendo n el número de elementos del vector.

Sin embargo, si el vector se divide en una parte casi vacía y la otra parte con casi todos los elementos la recurrencia es del tipo:

$$T(n) = \begin{cases} c_1 & n == 1 \\ T(n-1) + n & n > 1 \end{cases}$$

siendo n el número de elementos del vector que viene dado por `fin - ini`.

y el coste $\mathcal{O}(n^2)$

5.7. Modificaciones al problema.

5.8. Implementación.

```
// Divide el vector en tres partes.
// [ini..p) tiene los elementos menores que el pivote
// [p..q] tiene los elementos iguales que el pivote
// (q..b) tiene los elementos mayores que el pivote
template <typename T, typename Comp = std::less<T>>
void particion (std::vector<T> & v, int a, int b, T pivote,
                int& p, int& q ) {
    int k;
    p = a; k = a; q = b-1;
    while (k≤ q ) {
        if (v[k] = pivote) ++k;
        else if ( v[k] < pivote)  {
            std::swap(v[p],v[k]);
            ++p; ++k;
```

```
        }
        else {
            std::swap(v[q],v[k]);
            --q;
        }
    }
}

template <class T>
void quickSort( std::vector<T> & v, int ini, int fin ) {
    int  p, q;
    if ( ini + 1 < fin ) { // mas de un elemento
        particion(v, ini, fin, v[ini], p, q);
        quickSort(v, ini, p);
        quickSort(v, q+1, fin);
    }
}
```

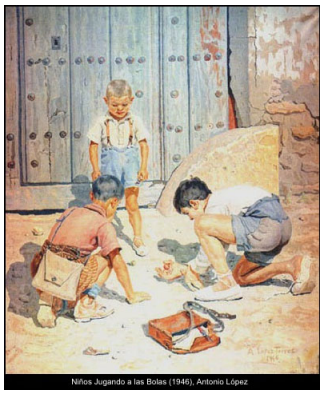
Utilizando la función `partition` de la librería `algorithm`:

```
using tvit = std::vector<int>::iterator;
template <typename T, typename Comp = std::less<T>>
void quickSort( std::vector<T> & v, tvit ini, tvit fin ) {
    if ( ini != fin) { // mas de un elemento
        tvit itMitad = std::partition(ini, fin, [ini](int n){return n < (*ini);});
        quickSort(v, ini, itMitad);
        quickSort(v, itMitad+1, fin);
    }
}

// Funcion principal
template <typename T, typename Comp = std::less<T>>
void quickSort (std::vector<T> & v, Comp ord = Comp()) {
    quickSort(v,v.begin(),v.end());
}
```

6. Jugando a las canicas.

Jugando a las canicas


Niños Jugando a las Bolas (1946), Antonio López

Antes los niños jugaban a las canicas en el patio del colegio o en la calle por la tarde. Bastaba con hacer un agujero en el suelo y un circulo alrededor. Luego se agrupaban los amigos por parejas y se iban enfrentando dos a dos. En cada partida los chicos van tirando las canicas de una en una alternándose. El objetivo es meter el mayor número de canicas en el agujero y para ello pueden golpear las canicas que no entraron, tanto las suyas para intentar que entren como las del amigo para alejarlas y que resulte imposible meterlas posteriormente. El perdedor le entrega al ganador parte de sus canicas.

Normalmente se organizaban torneos con todos los chicos del barrio. Los ganadores de cada ronda pasaban a enfrentarse en la ronda siguiente. En nuestro problema, el chico que ha perdido entrega la mitad de sus canicas al ganador. Al comenzar el torneo cada jugador tiene un número de canicas, que depende de las que le van comprando sus padres y abuelos y del número de partidas que ha ganado. Suponemos que se enfrentan el primer chico con el segundo, el tercero con el cuarto etc. En la siguiente ronda se enfrentan el ganador entre el primero y segundo con el ganador entre el tercero y cuarto etc. En la final se enfrentan el ganador de la primera mitad con el ganador de la segunda mitad. En el problema consideramos que cada ronda la gana el jugador que tiene más canicas. A igualdad de canicas, gana el que se apuntó antes en el torneo (el que está primero en el vector)

Requisitos de implementación.

Implementar una función recursiva que devuelva el nombre del jugador que gana la ronda y el número de canicas que tiene cuando acaba la ronda. En cada ronda se enfrentan el ganador de la mitad izquierda con el ganador de la mitad derecha. Utilizad la función recursiva para obtener el ganador de la mitad izquierda y el ganador de la mitad derecha y el número de canicas de cada uno.

Entrada

La entrada consta de una serie de casos de prueba. Cada caso de prueba tiene n+1 líneas. En la primera se indica el número, n, de elementos del vector. En las n siguientes se indican los nombres de los jugadores y sus canicas.

Se garantiza que todos los chicos tienen al menos una canica inicialmente.

Salida

Para cada caso de prueba se escribe en una línea el nombre del ganador del torneo y el número de canicas que tiene al acabar.

6.1. Objetivos del problema.

- Practicar el método de resolución de problemas *divide y vencerás*. En concreto, practicar la construcción de soluciones desde el caso base.

6.2. Ideas generales.

- La solución es recursiva.
- Para cada ronda obtenemos el ganador en la ronda anterior de la parte izquierda del vector y el número de canicas que tiene al llegar a esta ronda mediante una llamada recursiva a la parte izquierda del vector. Obtenemos el ganador en la ronda anterior de la parte derecha del vector con otra llamada recursiva.
- Combinamos los resultados de las dos llamadas recursivas, obteniendo el ganador de esta ronda y el número de canicas con que termina la ronda.
- El caso base del algoritmo se produce cuando tenemos únicamente dos amigos para enfrentarse.

6.3. Ideas detalladas.

Consultar el problema de la ordenación por mezclas (mergesort) y el problema que calcula si un vector es suficientemente disperso.

6.4. Algunas cuestiones sobre implementación.

Consultar el problema de la ordenación por mezclas (mergesort) y el problema que calcula si un vector es suficientemente disperso.

6.5. Errores frecuentes.

No se conocen.

6.6. Coste de la solución.

- Para calcular el coste de la función recursiva `canicas` planteamos la recurrencia correspondiente a la implementación realizada.
- En la implementación Secc. 6.8, el caso base tiene coste constante ya que solo realiza unas pocas comparaciones, asignaciones y operaciones aritméticas. En el caso recursivo se ejecutan dos llamadas recursivas cada una con la mitad de los elementos. Con los resultados de las llamadas recursivas se realizan únicamente operaciones de coste constante. La recurrencia es:

$$T(n) = \begin{cases} c_0 & n == 2 \\ 2T(\frac{n}{2}) + c_1 & n >= 4 \end{cases}$$

  siendo n el número de elementos del vector que viene dado por `fin - ini`.
- El coste es del orden de $\mathcal{O}(n)$
- El despliegue puede consultarse en la parte del cuaderno de problemas correspondiente a la complejidad de algoritmos recursivos.

6.7. Modificaciones al problema.

- Calcular los participantes de una determinada ronda.
  - Las rondas empiezan a numerarse cuando se enfrentan los primeros amigos. Si tenemos 2^n amigos para jugar, en la primera ronda se juegan 2^{n-1} partidas. La segunda ronda se enfrentan los ganadores de la primera ronda etc.

- Para calcular la ronda en que estamos podemos añadir un valor de retorno de la función que lo indique. En el caso base se devolverá un 1, indicando que finalizó la ronda 1. Cada función devolverá una ronda más. Observad que el valor devuelto por las dos llamadas recursivas será el mismo, por lo que solo es necesario coger uno de ellos.
- Los participantes de la ronda se pueden devolver en un vector al que sólo se le dará valor en la ronda pedida para evitar copias innecesarias de datos. Se puede devolver el índice en que se encuentran los participantes en el vector original, en lugar de copiar sus identificadores.

6.8. Implementación.

```
struct tdatos {
    std::string nombre;
    int canicas;
};
// Sobrecarga operador >> para lectura de datos
std::istream & operator>> (std::istream& entrada, tdatos& d) {
    entrada >> d.nombre >> d.canicas;
    return entrada;
}
// Siempre hay al menos dos elementos
tdatos canicas(std::vector<tdatos> const& v, int ini, int fin){
    tdatos sol;
    if (ini + 2 = fin){  // dos elementos
        if (v[ini].canicas ≥ v[ini+1].canicas) {
            sol.nombre = v[ini].nombre;
            sol.canicas = v[ini].canicas + v[ini+1].canicas/2;
        }
        else {
            sol.nombre = v[ini+1].nombre;
            sol.canicas = v[ini+1].canicas + v[ini].canicas/2;
        }
    }
    else { // 4 o mas elementos. Potencia de dos
        int mitad = (ini + fin) /2;
        tdatos solIz = resuelve(v,ini,mitad);
        tdatos solDr = resuelve(v,mitad,fin);
        if (solIz.canicas ≥ solDr.canicas) {
            sol.nombre = solIz.nombre;
            sol.canicas = solIz.canicas + solDr.canicas / 2;
        }
        else {
            sol.nombre = solDr.nombre;
            sol.canicas = solDr.canicas + solIz.canicas / 2;
        }
    }
    return sol;
}

bool resuelveCaso() {
  // Lectura de datos
  int numElem;
  std::cin >> numElem;
  if (!std::cin) return false;
  std::vector<tdatos> v(numElem);
  for (size_t i = 0; i < numElem; i++){
    std::cin >> v[i];
  }
        // Resolver y escribir los resultados
  tdatos sol = canicas(v, 0, v.size());
        std::cout << sol.nombre << ' ' << sol.canicas << '\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;
}
```