2. Divide y Vencerás (DyV)

2.1 Introducción

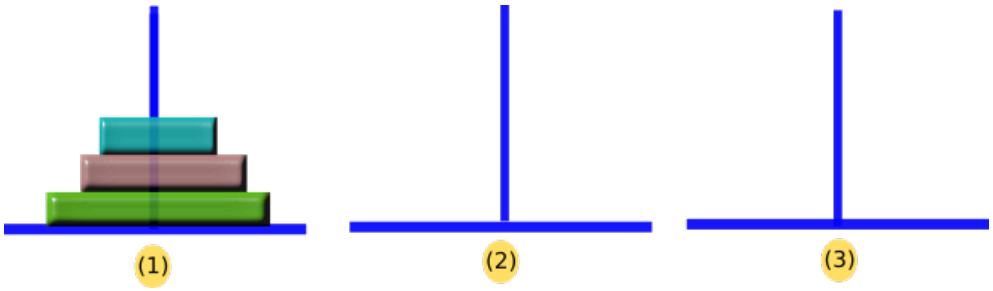
La técnica de diseño Divide y Vencerás, DyV, resuelve un problema a partir de las soluciones de subproblemas del mismo tipo pero de menor tamaño. Como herramienta de diseño usa la recursividad, aunque también se puede plantear el algoritmo basado en la técnica DyV usando un algoritmo iterativo que use alguna estructura que guarde el contexto por donde seguir. El esquema general de DyV es:

```
1  DivideVenceras(p:problema)
2   para i=1,2,...,k
3     si=Resolver(pi)
4   solucion=Combinar(s1,s2,...,sk)
```

Los pasos fundamentales son:

1. Platear el problema en k-subproblemas de menor tamaño. Se divide el problema en k-subproblemas cada uno de un tamaño n_k, tal que $0 \leq n_k < n$
2. Se resuelve de manera independiente los k-subproblemas:
   - De forma directa si es un caso base
   - De forma recursiva
3. Combinar las k soluciones de los subproblemas para obtener la solución al problema original.

Ejemplo 2.1.1
Torres de Hanoi. En el juego de las torres de Hanoi tenemos 3 torres y queremos mover todos los discos que están en una de las torres en otro. La condición que siempre se debe de cumplir es que un disco siempre tiene encima otro disco de menor diámetro.



El planteamiento siguiendo la técnica DyV es Resolver el problema para $n-1$ discos de la torre origen a intermedia, dejando solamente un disco en la torre que, siendo este el que se mueve a la torre destino y a continuación reolvemos para mover los $n-1$ discos de la torre intermedia a la torre destino. El código se puede escribir como sigue:

```
1  void Mover(int n,int T1,int T2){// T1 torre origen, T2 torre destino
2   if (  n==1)
3    cou<<''Moviendo de ''<<T1 << '' a ''<<T2<<endl;
4   else{
5     Mover(n-1,T1,6-T1-T2); // la torre intermedia es 6-T1-T2
6     Mover(1,T1,T2);
7     Mover(n-1,6-T1-T2,T2);
8    }
9 }
```

La eficiencia de este algoritmo se puede planter como:

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

Planteando el caso general tenemos:

$$T(n) = 2T(n-1)+1$$

$$T(n)-2T(n-1) = 1 \Longrightarrow \begin{cases} x = T(n) \\ b = 1 \\ p(m) = 1 \\ d = 0 \end{cases}$$

$$(x-2)(x-1) = 0$$

$$T(n) = c_1 \cdot 2^n + c_2 1^n$$

usando las condiciones iniciales llegamos a que $c_1 = -1$ y $c_2 = 1$

$$T(n) = 2^n - 1^n \Longrightarrow T(n) \in \mathbf{O}(2^n)$$

□

2.2 Requisitos para aplicar DyV

Existe una serie de elementos que son indepensables para constuir un algoritmo basado en DyV:

1. Tener un método básico para resolver los problemas de tamaño pequeño.
2. División del problema original en un conjunto de subproblemas con soluciones más sencillas.
3. Los problemas en los que se dividen deben ser disjuntos e independientes
4. Tener un método para combinar los resultados.

En el caso que tengamos solamente un subproblema hablamos de reducción. Normalmente la técnica DyV intenta dividir el problema en subproblemas de igual tamaño. Y por lo tanto los problemas más frecuentes es dividir el problema en dos subproblemas de igual tamaño. Así el esquema en tal caso sería:

```
1  DivideVenceras(p,q:indice)
2   si (Pequenho(p,q)) entonces
3     solucion = SolucionDirecta(p,q);
4   else
5    m = Dividir (p,q)
6    Solucion = Combinar(DivideVenceras(p,m),DivideVenceras(m+1,q))
```

Ejemplo 2.2.1
Buscar el máximo en un vector entre las posiciones p y q.

```
1  const int Th =3;
2  int Maximo(const int * v, int p, int q){
3    if (q-p<Th){ // Si es pequenho
4        int maximo = v[p];
5        for (int i=p+1;i<q;i++)
6          if (max<v[i])
7            max =v[i];
8        return max;
9    }
10   else {
11    int m= (p+q)/2;
12    int d1= Maximo(v,p,m);
13    int d2= Maximo(v,m,q);
14    return (d1>d2? d1:d2);
15   }
16 }
```

□

Ejemplo 2.2.2

Un ejemplo típico de la técnica DyV que divide el problema en dos subproblemas es la búsqueda binaria. En este caso para resolver el problema original se basa en la solución obtenida para solamente

una de las mitades. Por lo tanto en este caso la combinación de la soluciones no hace nada, ya que solamente consiste en trasmitir la solución obtenida por una de las mitades.

```
1  /*
2   * @brief Busca un elemento en un conjunto de elementos en un vector
3     @param v: vector con los elementos. Deben estar ordenados de forma creciente
4     @param inicio: indice del elemento mas a la izquierda del vector.
5     @param fin: indice del ultimo elemento a la derecha
6     @param x: elemento a buscar
7     @return el indice donde esta x. En el caso que no este devuelve -1
8  */
9  int Busqueda_Binaria(int * v,int inicio, int fin, int x){//← T(fin-inicio+1)=T(n)
10    if (inicio<=fin){//← 1 acotamos todo con 1
11      int mitad = (fin+inicio)/2;
12      if (x==v[mitad]) return mitad;
13    }
14    else{
15      if (v>[mitad])
16        return Busqueda_Binaria(v,mitad+1,fin,x); //← T(n/2)
17      else
18        return Busqueda_Binaria(v,inicio,mitad-1,x); //← T(n/2)
19      else
20        return -1;
21    }
22 }
```

La eficiencia del algoritmo se puede plantear en el caso general como

$$T(n) = T(\frac{n}{2}) + 1 \qquad \text{para } n \geq 1$$

Resolviendo la recurrencia tenemos:

$$T(n) - T(\tfrac{n}{2}) = 1 \Longrightarrow \text{hacemos el cambio de variable } n = 2^m$$

$$T(2^m) - T(2^{m-1}) = 1 \Longrightarrow \begin{cases} x = T(2^m) \\ b = 1 \\ p(m) = 1 \\ d = 0 \end{cases}$$

$$(x-1)^2 = 0$$

$$T(2^m) = c_1 1^m + c_2 \cdot m \cdot 1^m \text{ deshacemos cambio de variable } n = 2^m \text{ y } m = log_2(n)$$

$$T(n) = c_1 + c_2 \cdot log_2(m)$$

Claramente este algoritmo tiene un tiempo de ejecución $T(n) \in \Theta(log_2(n))$ □

A pesar de que existe muchos problemas que se resuelve con la técnica DyV diviendo el problema original en dos mitades, la técnica DyV permite dividir el problema en más de dos mitades. Como ejemplo veamos el algoritmo para buscar un elemento en un vector ahora en ver de dividir en dos partes los dividimos en tres partes y se activa la recursividad par buscar en el primer tercio, segundo o tercero dependiendo del valor del elemento que buscamos.

Ejemplo 2.2.3
Búsqueda Ternaria.- Dado un vector ordenado de forma creciente aplicar la busqueda ternaria para encontrar un elemento x. Para resolver este problema vamos a planter dos soluciones:

SOLUCIÓN 1.- Compara el elemento con la mitad si es igual ya lo hemos encontado. En otro caso realiza la buqueda en el primer tercio si este le devuelve false, realiza la busqueda en el segundo tercio y si este devuelve false realiza la busqueda en el tercer tercio.

```
bool B_Ternaria (const int *v,
                 int n, int x){
if (n==0) return false;
else if (n==1) return v[0]==x;
else {
int mitad = n/2;
if (v[mitad]==x) return true;
int tercio = n/3;
if (B_Ternaria(v,tercio,x))
return true;
else
  if (B_Ternaria(v+tercio,
                 tercio,x))
  return true;
  else
  if (B_ternaria(v+2*tercio,
                  n-2*tercio,x))
    return true;
  else
  return false;
}
}
```

Veamos la eficiencia de este algoritmo:

$$T(n) = \begin{cases} 1 & n == 0 || n == 1 \\ 3T(\frac{n}{3}) + 1 & n \geq 2 \end{cases}$$

Resolviendo la recurrencia:

$$T(n) = 3T(\tfrac{n}{3}) + 1 \Longrightarrow$$
haciendo el cambio de variable $n = 3^m$
$$T(3^m) - 3T(3^{m-1}) = 1 \Longrightarrow \begin{cases} x = T(3^m) \\ b = 1 \\ p(m) = 1 \\ d = 0 \end{cases}$$

$$(x-3)(x-1) = 0$$
$$T(3^m) = c_1 \cdot 3^m + c_2 \cdot 1^m$$
deshaciendo el cambio de variable $n = 3^m$
y $m = log_3(n)$
$$T(n) = c_1 \cdot n + c_2 \in \Theta(n)$$

SOLUCIÓN 2.-Compara con el elemento en 1/3 si es igual lo hemos encontrado. En otro caso se compara con el elemento en 2/3 si es igual termina. En otro caso lanza la busqueda en el tercio que corresponda.

```
bool B_Ternaria (const int *v,
                 int n, int x){
if (n==0) return false;
else if (n==1) return v[0]==x;
  else {
  int tercio = n/3;
  if (v[tercio]==x) return true;
  else if (v[2*tercio]==x)
      return true;
  if (v[tercio]>x)
    return B_Ternaria(v,tercio,x);
  if (v[2*tercio]>x)
   return B_Ternaria(v+tercio,
                     tercio,x);
  else
  return
  B_Ternaria(v+2*tercio,
             n-2*tercio,x);
  }
}
```

Veamos la eficiencia de este algoritmo:

$$T(n) = \begin{cases} 1 & n == 0 || n == 1 \\ T(\frac{n}{3}) + 1 & n \geq 2 \end{cases}$$

Resolviendo la recurrencia:

$$T(n) = T(\tfrac{n}{3}) + 1 \Longrightarrow$$
haciendo el cambio de variable $n = 3^m$
$$T(3^m) - T(3^{m-1}) = 1 \Longrightarrow \begin{cases} x = T(3^m) \\ b = 1 \\ p(m) = 1 \\ d = 0 \end{cases}$$

$$(x-1)(x-1) = 0$$
$$T(3^m) = c_1 \cdot 1^m + c_2 \cdot m \cdot 1^m$$
deshaciendo el cambio de variable $n = 3^m, m = log_3(n)$
$$T(n) = c_1 + c_2 \cdot log_3(n) \in \Theta(log(n))$$

Con la Solución 2 obtenida llegamos a una eficiencia del tipo $O(log(n))$, que es la misma que la eficiencia de la búsqueda binaria (la base de los logaritmos nos importa). Por lo tanto no se mejora la búsqueda por mas particiones que hagamos para concentrar la búsqueda. Vale con dos particiones. □

2.3 Algoritmo de Ordenación basados en DyV

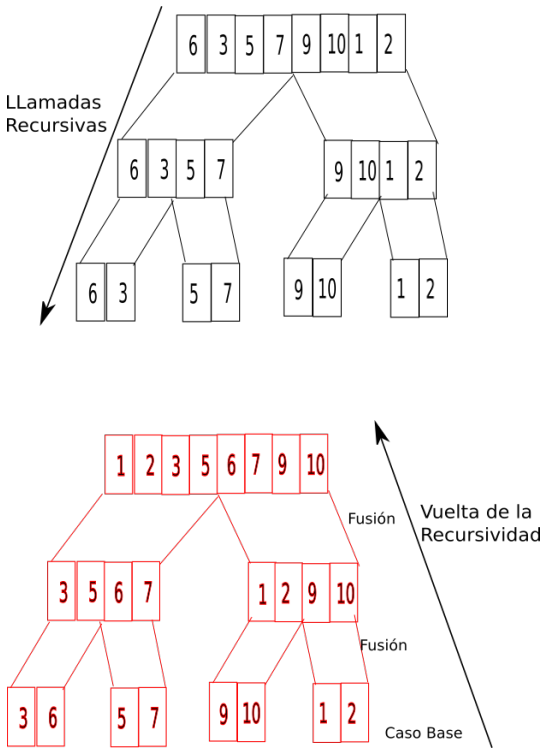
En este sección vamos a analizar dos algoritmos de ordenación basados en DyV:

- Mergesort. Divide en dos mitadas. Una vez ordenadas esas dos mitadas obtiene el vector original ordenado cuando fusiona esas dos mitades.
- Quicksort.- Divide en dos mitades pero en principio no siempre son dos mitades con igual número de elementos. La ordenación del vector original se basa en la ordenación directa de estas dos partes.

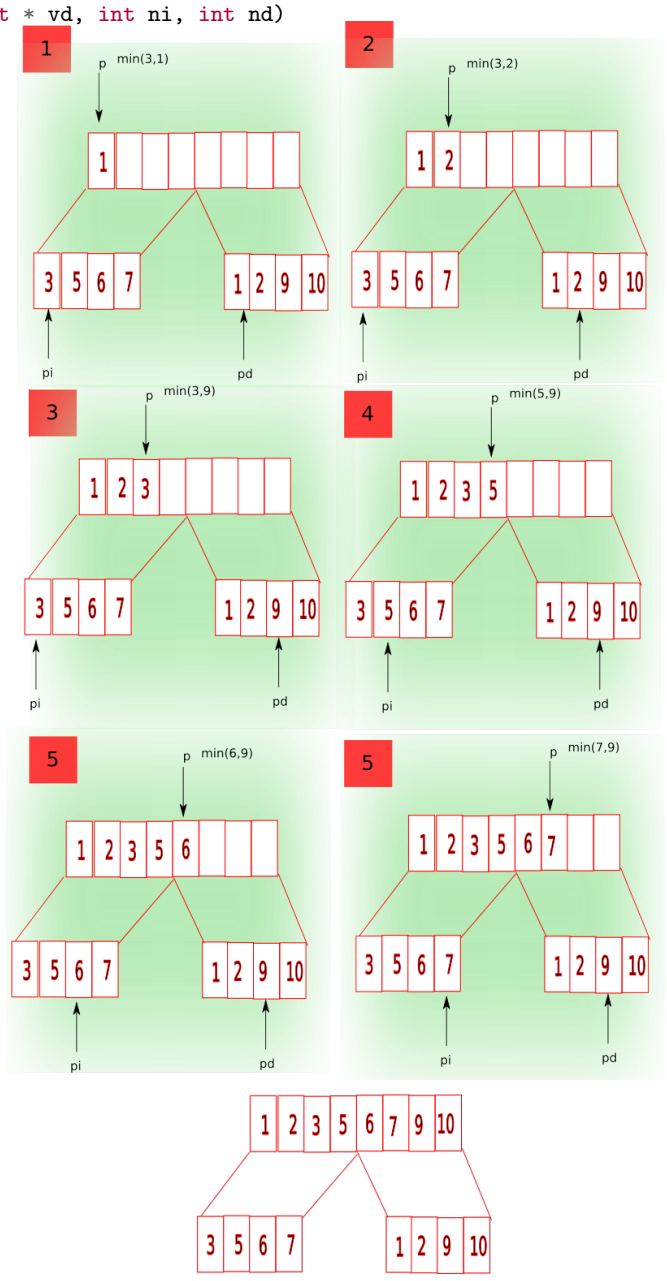
2.3.1 Mergesort

Si tenemos un vector de x elementos y queremos ordenarlos, dividimos el vector por la mitad hasta llegar a un número mínimo de elementos en el que el vector se ordena y luego se fusiona con el nivel superior. Es una función recursiva. Su código sería:

```
1   void Orden_MergeSort (int * v, int n)
2   {
3         if (n==1)
4          return v[0]);
5         else
6         if (n == 2)
7         {
8               if (v[0] > v[1])
9                     Intercambiar (v[0],v[1]);
10        }
11        else if (n>2)
12        {
13              int ni=(n/2), nd=n-(n/2);
14              int * vi=new int [ni];
15              int * vd=new int [nd];
16
17              for(int i=0;i<ni;i++)
18                    vi[i]=v[i];
19              for(int i=0;i<nd;i++)
20                    vd[i]=v[i+(n/2)];
21
22              //Ordenamos a la izquierda
23              Orden_MergeSort (vi,ni);
24
25              //Ordenamos a la derecha
26              Orden_MergeSort (vd,nd);
27
28              //Fusionamos en v, vi y vd
29              Fusion(v,vi,vd,ni,nd);
30              delete [] vi;
31              delete [] vd;
32        }
33  }
```



```
1   void Fusion (int * vout, int * vi, int * vd, int ni, int nd)
2   {
3       int pi=0, pd=0, p=0;
4
5       while (pi < ni && pd < nd)
6       {
7           if (vi[pi] < vd[nd])
8           {
9               vout[p] = vi[pi];
10              pi++;
11          }
12
13          else
14          {
15              vout[p] = vd[pd];
16              pd++;
17          }
18          p++;
19      }
20      //si queda algo en vi
21      while (pi < ni)
22      {
23          vout[p] = vi[pi];
24          p++;
25          pi++;
26      }
27
28      //si queda algo en vd
29      while (pd < nd)
30      {
31          vout[p] = vd[pd];
32          p++;
33          pd++;
34      }
35  }
```



El algoritmo de mergesort tiene un tiempo de ejecución:

$$T(n) = 2T(\frac{n}{2}) + n \quad n > 2$$

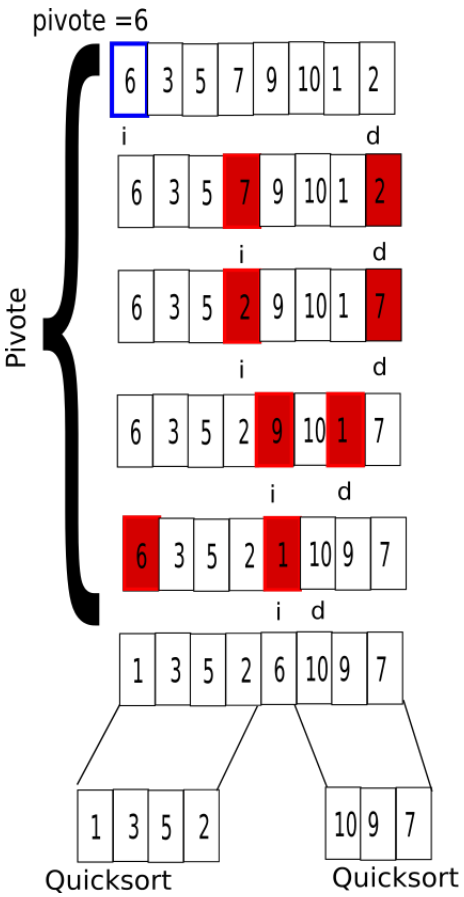
Si resolvemos la recurrencia llegamos a que el algoritmo Mergesort tiene un orden exacto de $\Theta(nlog_2(n))$.

2.3.2 Quicksort

Este método en cada paso ordena los elementos relativos a un pivote (uno de los elemento del vector).

```
1   void QuickSort (int inicio, int final)
2   {
3       int pos_pivote;
4
5       if (inicio < final)
6       {
7           pos_pivote = Pivote (inicio, final);
8           //Ordena primera mitad
9           QuickSort (inicio, pos_pivote-1);
10          //Ordena segunda mitad
11          QuickSort (pos_pivote + 1, final);
12      }
13  }
```

```
1   int Pivote (int primero, int ultimo)
2   {
3       int intercambia, izda, cha;
4       int pivote = v[primero];
5
6       i = primero + 1; //avanza hacia delante
7       d = ultimo; //retrocede hacia atras
8
9       while (i <= d)
10      {
11          while (i <= d && v[i] <= pivote)
12              i++;
13
14          while (i <= d && v[dcha] > pivote)
15              d--;
16
17          if (i < d)
18          {
19              intercambia = v[i];
20              v[i] = v[d];
21              v[d] = intercambia;
22              d--;
23              i++;
24          }
25      }
26      intercambia = v[primero];
27      v[primero] = v[d];
28      v[d] = intercambia;
29
30      return d;
31  }
```



La ordenación relativa al pivote la realiza la función Pivote.Esta ordenación consiste en dejar todos los elementos menores o iguales que el pivote a la izquierda y todos los elementos mayores a la derecha.La

función de ordenación Quicksort se llama recursivamente para el subvector izquierda y subvector derecha. La función Pivote toma un elemento arbitrario del vector (pivote), en nuestro código se toma el elemento primero. A continuación la función Pivote recorre el vector de izquierda a derecha usando el índice i hasta encontrar un elemento apuntado por i tal que v[i] sea mayor que pivote. Después, recorre el vector de derecha a izquierda (con el índice d) hasta encontrar otro elemento tal que v[d] menor que pivote e intercambia los elementos v[i] y v[d]. Se repite el proceso hasta que se cruzan i y d. En este caso se intercambian v[primero] por v[d] y se devuelve d (en v[d] tenemos el pivote) para establecer las dos mitades siguientes. El elemento v[d] ya está colocado y no se volverá a tratar más. Para este algoritmo es interesante analizar los tiempos de ejecución en el mejor, peor y caso promedio. En el caso promedio y mejor la probabilidad de que un numero sea menor o mayor a este tiene una probabilidad de $\frac{1}{2}$. Por lo tanto en promedio tendremos mitades con el mismo número de elementos aproximadamente.Y en este caso el tiempo de ejecución del algoritmo Quicksort se puede plantear como:

$$T(n) = 2T(\frac{n}{2}) + n \quad para \quad n > 2$$

Siendo n la eficiencia de la función Pivote. Resolviendo esta ecuación llegamos a que $T_{\frac{1}{2}}(n) \in \Theta(nlog_2(n))$ y en el mejor de los casos $T_m(n) \in \Omega(nlog_2(n)$. Quicksort como esta formulado tiene como peor caso cuando el vector está ordenado, en cualquiera de las dos direcciones. Ya que cuando se ejecuta la función Pivote si esta ordenado de forma creciente obtiene una división con todos lo elementos situados a la derecha y la otra es vacía. Si esta ordenado de forma decreciente obtiene una división con todos elementos a la izquierda del pivote y la otra parte es vacía. En este caso el tiempo de ejecución en el peor de los casos es:

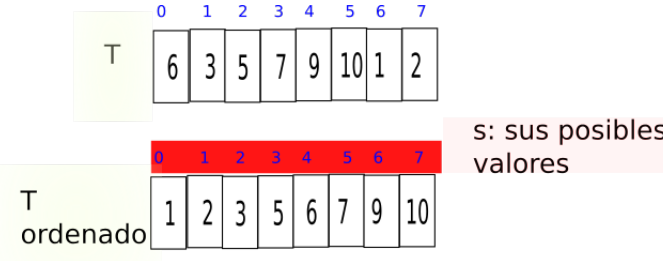
$$T(n) = T(n-1) + n \quad para \quad n > 2$$

Si resolvemos esta ecuación llegamos a que $T_p(n) \in O(n^2)$.

Problema de la Selección

Sea T un vector no ordenado con n posiciones. Y s es un entero enre $0, 1, \cdots, n-1$. El problema de la selección consiste en encontrar el elemento que se encontraría en la posicion s si el vector estuviera ordenado. La solución trivial sería ordenar el vector y consultar $T[s]$. Pero nuestro objetivo es sin tener que ordenar a priori el vector deducir que elemento esá en la posición s cuando el vector estuviera ordenado.

A la derecha tenemos un ejemplo de un vector T con 8 elementos. Si estuviera ordenados los diferentes valores para $s \in 0, 1, \cdots, 7$ obtenemos el vector que se muestra abajo. Por ejemplo si nos pregunta que elemento ocuparía $s = \frac{n}{2} = 4$ nos están preguntado por elemento que es la mediana. En nuetro ejemplo T[4] si estuviera ordenado es 6.



Idea. Usaremos la función *Pivote* del algoritmo de ordenación *Quicksort*. De forma que cada vez que se ejecuta *Pivote* tiene un doble efecto:

- Coloca los elementos a la izquierda que son menores o iguales que el elemento en la posición pivote, y a la derecha mayores.
- El elemento en la posición pivote ya está ordenado. Por lo tanto si al ejecutar la función *Pivote* nos devuelve un índice igual a s, ya sabemos qué elemento estaría en s cuando el vector esté ordenado.

El algoritmo sería el siguiente:

```
int Seleccion(const int *v, int n, int s){
  int i,j,l;
  i=0; j=n-1;
  do{
    l = Pivote(v,i,j);
    if (s<l)
     j = l-1; //debe estar en la mitad izquierda
    else i = l+1;//para la mitad derecha

  }while (l!=s);
  return v[l];
}
```

La eficiencia de este algoritmo es:

- *Caso Promedio y Mejor*. Ocurre cuando la función Pivote devuelve un índice l que está entorno a la mitad. En este caso el tiempo de ejecución sería $T(n) = T(\frac{n}{2}) + n$, que estaría en $\Omega(n)$ y $\Theta(n)$, respectivamente.
- *Caso Peor*. Ocurre cuando el vector está ordenado de partida. En este caso la función Pivote genera un índice que deja en una mitad $n-1$ elementos y en la otra 0 elementos. Por lo tanto el tiempo de ejecución sería $T(n) = T(n-1) + n$, que resolviendo podemos afirmar que se encuentra en $\mathbf{O}(n^2)$.

Ejemplo 2.3.1
Modificar la función Pivote del algoritmo Quicksort, para obtener una ordenación relativa con respecto a un elemento p y la ristra de elementos en el vector que son iguales a p.
Solución.- En primer lugar colocamos los elementos menores a p a la izquierda, y los mayores a la derecha de p. A continuación desde el extremo izquierdo del vector analizamos si el elemento es igual p y lo colocamos lo más proximo a el. El código queda de la siguiente forma:

```
 /*
 @brief: busca la ristra de elementos iguales a p y los coloca
        todos consecutivos desde k a l
 @param v: vector con los elementos
 @param incio: indice inicial del vector
 @param ultimo: indice ultimo valido del vector
 @param p: elemento sobre el que se quiere buscar la ristra
 @param k: indice donde inicia los valores iguales a p
```

```
 @param l: indice donde termina los valores iguales a p
 */
 void Pivote2(int * v, int inicio,int ultimo, int p, int &k,int &l){
  k = inicio; l = ultimo+1;
  //colocamos menores o iguales que p a la izquierda, y mayores a derecha
  do{
    k++;
  }while(v[k]<=p && k<ultimo);
  do{
    l--;
  }while(v[l]>p)
  while (k<l){
    Intercambiar(v[k][v[l]);
    do{ k++; }while (v[k]<=p);
    do{ l--; }while (v[l]>p);
  }
  Intercambiar(v[inicio],v[l]);
  l++; //indice tal que v[l] es diferente a p
  k= inicio-1;
  m=l;
  //buscamos tal que v[k]=p
  do{ k++; }while(v[k]!=p && k<l);
  //buscamos tal que v[m]!=p
  do{ m--; }while (v[m]==p && m>=inicio) ;

  while (k<m){
    Intercambiar(v[k],v[m]);
    do{ k++; } while (v[k]!=p);
    do{ m--; }while (v[m]==p);
  }
}
```

□

Problema: El elemento en su Posición

Sea $a[0,\cdots,n-1]$ un vector ordenado de forma creciente y todos los elementos distintos. Tenemos que implementar un algoritmo de complejidad $\Theta(log_2(n))$ y $O(log_2(n))$ capaz de encontrar un índice i tal que $0 \leq i \leq n-1$ y que cumpla que $a[i] = i$, suponiendo que exista.

Solución. Como idea usar la búsqueda binaria y teniendo en mente que si existe tal elemento debe ser uno de entre 0 y $n-1$, para un vector con n posiciones. Los pasos a seguir son:

1. Obtenemos el elemento que está en la mitad $a[\frac{n}{2}]$ y comprobamos si $a[\frac{n}{2}] = \frac{n}{2}$, si es así lo hemos encontrado. En otro caso se puede dar las siguientes situaciones:
2. $a[\frac{n}{2}] > \frac{n}{2}$. En este caso si existe el elemento debe encontrarse entre los índices 0 y $\frac{n}{2}-1$. Esto es

así porque el rango de valores que pueden haber a la derecha cumplen la condición de que :

- $a[\frac{n}{2}]+1 > \frac{n}{2}+1 \Longrightarrow a[\frac{n}{2}+1] > \frac{n}{2}+1$
- $a[\frac{n}{2}]+2 > \frac{n}{2}+2 \Longrightarrow a[\frac{n}{2}+2] > \frac{n}{2}+2$
- $\cdots$
- $a[\frac{n}{2}]+n-\frac{n}{2}-1 > \frac{n}{2}+n-\frac{n}{2}-1 = n-1 \Longrightarrow a[\frac{n}{2}+n-\frac{n}{2}-1] > n-1$

```
1  int Posicion(const int * v, int n){
2   if (n==0)
3    return -1;
4   else {
5    int mitad = n/2;
6    if (v[mitad]==n/2) return mitad;
7    if (v[mitad>n/2)
8    return Posicion(v,n/2);
9    else return Posicion(v+n/2+1,n-n/2-1);
10  }
11 }
```

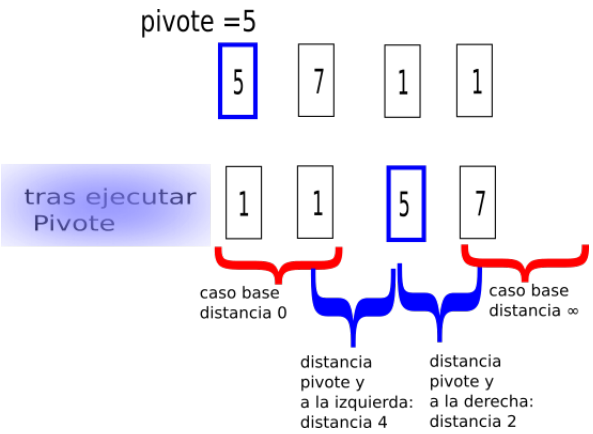
Problema: Distancia mínima entre elementos vecinos

Dados n valores reales, diseñar un algoritmo que calcule el valor de la absoluta diferencia minima de entre los dos valores más cercanos. Es decir:

$$d = min_{0 \leq i \leq n-1} |v_i - v_{i+1}| \tag{2.1}$$

Solución. Para llevar a cabo este algoritmo vamos a usar la función Pivote del algoritmo Quicksort. En el algoritmo, el caso general actua obteniendo la distancia menor en la mitad izquierda, y la distancia menor en la parte derecha. De entre estas dos disntancias nos quedamos con el mínimo. Además obtenemos la distancia entre el pivote y elemento que tiene a su izquierda, y la distancia del privote al elemento que tiene a la derecha. De entre todas estas distancias nos quedamos con la menor y este será el resultado de nuestro algoritmo. Como caso base tendremos que si el vector tiene un elemento la distancia es infinita, y si tiene dos elementos la diferencia entre estos.

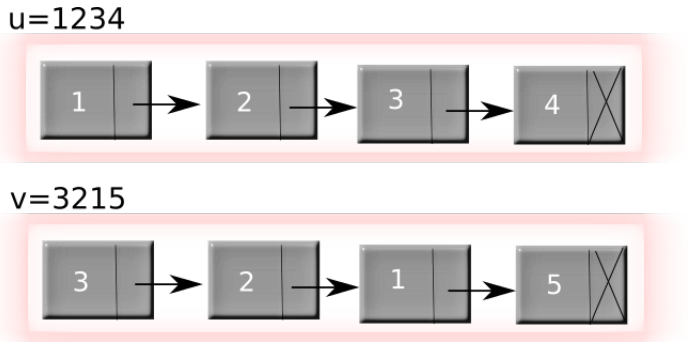
En la derecha se ve los paso a seguir con un vector de cuatro elementos. En primer lugar se ejecuta la función Pivote para dejar a la izquierda los valores menores al pivote que es 5. Y a la derecha los mayores. Recursivamente se aplica buscar la distancia mínima sobre la parte izquierda y la parte derecha. La solución de estas dos mitades se obtienen a través de los casos bases, siendo distancia 0 para la mitad izquierda y distancia infinita para la mitad derecha. Una vez obtenidas estas se calcula la distancia entre el pivote y el elemento a su izquierda , con disntancia 4 y la distancia entre el pivote y el elemento a la derecha con distancia 2. El mínimo por lo tanto entre 0, 4, 2, e infinito es 0. Y sería la distancia mínima que se devuelve.




```
1  float M_Distancia(const float * v, int n){
2   if (n==2)
3    return (abs(v[0]-v[1]));
4   else if (n==1)
5    return numeric_limits<float>::max(); //infinito
6   else{
7     int p= Pivote(v,0,n-1);//p es la posicion del pivote
8     float i=M_Distancia(v,p);//minima distancia en la parte izquierda
9     float d=M_Distancia(v+p+1,n-p-1);//minima distancia en la parte derecha
10    float m_1=min(i,d);
11    float m_2=numeric_limits<float>::max();
12    if (p>0)
13    //distancia pivote y elemento a la izquierda
14      m_2= min(m_2,abs(v[p-1]-v[p]));
15    if (p<n-1)
16    //distancia pivote y elemento a la derecha
17     m_2= min(m_2,abs([v[p]-v[p+1]));
18
19    return (m_1>m_2)?m_2:m_1;
20  }
21 }
```

2.4 Multiplicación de Enteros Largos

Supongamos que vamos a representar enteros muy grandes, y para ellos supongamos que usamos una representación como celdas enlazadas o un vector de dígitos. Por ejemplo en la figura se ve como se representa dos enteros 1234 y 3215 usando un conjunto de celdas enlazadas:



El agoritmo de multiplicación de números que aprendemos es la escuela sigue los siguientes pasos:

1. *resultado* $\leftarrow 0$
2. Para cada cifra de v (cv)
3. Multiplicar todos las cifras de u por cv obteniendo r
4. Sumarle a *resultado* el valor *r* con el desplazamiento correspondiente

Si u y v son de tamño n el algoritmo clásico tiene eficienci $\Theta(n^2)$.

El objetivo en esta sección es obtener un nuevo algoritmo basado en DyV que reduzca el tiempo $\Theta(n^2)$.

Aproximación 1.-. El algoritmo más directo divide los enteros en dos mitadas de tamaño $\frac{n}{2}$. Para cada una de las mitades se resuelve el problema de la multiplicación. Teniendo como caso base cuando los dos enteros tienen tamaño 1 entonces usamos la multiplicación estándar. Una vez obtenidas estas soluciones las combinamos con los desplazamientos correspondientes. Por ejemplo:

- Supongamos que u se descompone en wx y v se descompone en yz
- $u = 10^s w + x$ si $u = 1234 \Longrightarrow w = 12 \quad x = 34$
- $v = 10^s y + z$ si $v = 3215 \Longrightarrow y = 32 \quad z = 15$
- Si $s = \lfloor \frac{n}{2} \rfloor$ tenemos que $u \cdot v = (10^s w + x) \cdot (10^s y + z) = \underbrace{(10^{2s} wy)}_{subproblema\ 1} + \underbrace{10^s (wz + xy)}_{subproblemas\ 2\ y\ 3} + \underbrace{xz}_{subproblema\ 4}$

Suponiendo que las sumas se pueden hacer en $\mathbf{O}(n)$, $T(n)$ se puede formular como:

$$T(n) = 4T(\tfrac{n}{2}) + n \text{ con cambio de variable } n = 2^m$$
$$T(2^m) - 4T(2^{m-1}) = 2^m \Longrightarrow (x-4)(x-2) = 0$$
$$T(2^m) = c_1 \cdot 4^m + c_2 \cdot 2^m$$
$$T(n) = c_1 \cdot n^2 + c_2 \cdot n \in O(n^2) \leftarrow \textbf{NO MEJORAMOS!!}$$

Esta aproximación no mejora al algoritmo clásico de multiplicación. Por lo tanto tenemos que buscar algo en la descomposición que reduzca los 4 subproblemas a 3 subproblemas

Aproximación 2.-. Partimos de los cuatro subproblemas:

$$r = u \cdot v = \underbrace{(10^{2s} wy)}_{subproblema\ 1} + \underbrace{10^s (wz + xy)}_{subproblemas\ 2\ y\ 3} + \underbrace{xz}_{subproblema\ 4}$$

Idea.-. Reusar xz y wy para transformar $(wz+xy)$:

$$\begin{aligned}(wz+xy) &= \overbrace{(wz-wy+wy+xy-xz+xz)}^{\text{sumando y restando } wy \text{ y } xz} \\ &\quad \text{sacamos factor } w \text{ y } x \\ &= (z-y)\cdot w+(y-z)\cdot x+wy+xz \\ &\quad \text{sacamos factor } (z-y) \\ &= \underbrace{(z-y)(w-x)+wy+xz}\end{aligned}$$

Tenemos 2 multiplicaciones hechas
mas una nueva.
Hemos pasado de 4 a 3 multiplicaciones

Esta reducción, fue ideada por **Karatsuba y Ofman**, generando el algoritmo que se especifica como:

1. $r = u\cdot v = 10^{2s}wy+10^{s}((z-y)(w-x)+wy+xz)+xz$
2. Subproblemas : $m_1 = wy$, $m_2 = xz$, $m_3 = (z-y)\cdot(w-x)$

Eficiencia: Algoritmo de Karatsuba y Ofman.- La ecuación de recurrencia de este algoritmo sería:

$$\begin{aligned}T(n) &= 3T(\tfrac{n}{2})+dn \text{ con cambio de variable } n=2^m \\ T(2^m)-3T(2^{m-1}) &= d2^m \Longrightarrow (x-3)(x-2)=0 \\ T(n) &= c_1\cdot 3^{log_2(n)+c_2\cdot n} \\ T(n) &= c_1\cdot n^{log_2(3)}+c_2\cdot n \text{ con } n^{log_2(3)} = n^{1,59} \\ T(n) &\in \Theta(n^{1,59}) \leftarrow \textbf{HEMOS MEJORADO!!}\end{aligned}$$

En la práctica se obtienen beneficio con números mayores a 500 bits. Para ver cuando es mejor usar el método clásico, que gasta $\Theta(n^2)$, frente al método de Karatsuba y Ofman habría que buscar donde se cruzan. Por lo tanto planteamos:

$$c_{11}n^{1,59}+c_{12}n = c_{21}n^2+c_{22}n \Longrightarrow$$

cuando despejamos n obtenemos el límite superior de el tamaño del entero, en bits,con el que es mejor aplicar la multiplicación clásica

```
1  int Dyv_Mult(long int n ,long int v, int n,int base){
2   if (n<=base)// si el tamanho es menor o igual a la base
3     return u*v;
4   else{
5   // Obtenemos 10^s si la base fuese 10
6   int factorbase=1; int s=n/2;
7   for (int i=0;i<s;i++)
8     factorbase=factorbase*base;
9
10  // Obtenemos w, y, x, z ya que u =  10^s w+y  y v=10^s x+z
11  w = u/factorbase; y = u%factorbase;
12  x = v/factorbase; z = v%factorbase;
```

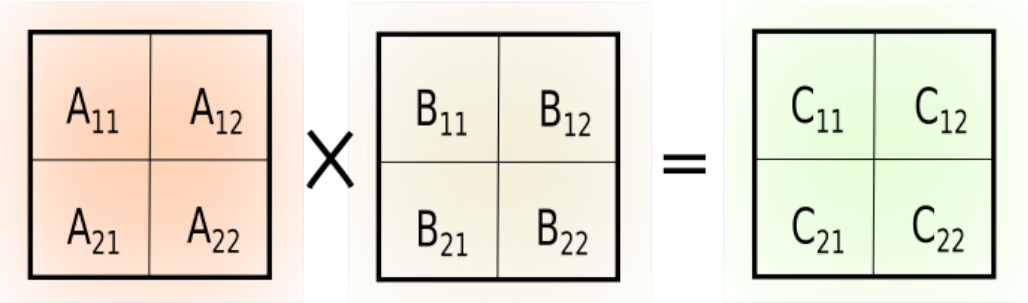
```
13
14   //resolvemos los subproblemas m1,m2,m3
15   m1= DyV_Mult(w,y,n/2,base);
16   m2= DyV_Mult(x,z,n/2,base);
17   m3= DyV_Mult((w-x),(z-y),n/2,base);
18   //Combinamos
19   int r = m1*(factobase*factorbase)+factorbase(m3+m1+m2)+m2
20   return r;
21   }
22  }
```

2.5 Multiplicación Rápida de Matrices

El algoritmo de multiplicación clasico de matrices se puede implementar de la siguiente forma:

```
1   /**
2     * @brief Multiplicacion matricial
3       @param A: matriz izquierda a ser multiplicada
4       @param B: matriz derecha a ser multiplicada
5       @param C: matriz resultante . ES MODIFICADO
6       @pre el numero de columnas de A debe ser igual al numero de filas de B
7   **/
8   void Mult_Matrices(const Matriz & A, const Matriz &B,  Matriz &C){
9       C = Matriz(A.filas(),B.columnas()); // se inicia la matriz
10                                          // con todos los valores a 0.
11      for (int i=0;i<A.filas(); i++)//  n
12        for (int j=0;j<B.columnas(); j++){ // n
13          C(i,j)=0;
14          for (int k=0;k<A.columnas();k++)// n
15            C(i,j)+= A(i,k)*B(k,j);
16         }
17  }
```

Suponiendo que las matrices son cuadradas el tiempo de ejecución de este algoritmo clásico es $\Theta(n^3)$. Por lo tanto nuestro objetivo es usando la técnica DyV obtener un algoritmo que baje ese tiempo de ejecución. Para ellos veamos una posible descomposición de A, B y C:



La primera aproximación para formular un algoritmo basado en la técnica DyV es que si cada matriz de $n \times n$, filas y columnas, la dividimos en cuatro subcuadrantes de tamaño $\frac{n}{2} \times \frac{n}{2} : A_{ij}, B_{ij}$ *y* C_{ij}. Con esta descomposición podemos formular cada cuadrante de C de la siguiente forma:

$$C_{11} = A_{11} \times B_{11} + A_{12} \times B_{21} \quad C_{12} = A_{11} \times B_{12} + A_{12} \times B_{22}$$
$$C_{21} = A_{21} \times B_{11} + A_{22} \times B_{21} \quad C_{22} = A_{21} \times B_{12} + A_{22} \times B_{22}$$

Vemos que tenemos que resolver 8 multiplicaciones matriciales de tamaño $\frac{n}{2}$. La combinación de los resultado requiere un tiempo de $\Theta(n^2)$ ya que debemos colocar los datos en cada casilla de C. Por lo tanto la eficiencia pordemos formularla como:

$$T(n) = 8T(\tfrac{n}{2}) + a \cdot n^2 \text{ haciendo el cambio de variable } n = 2^m$$
$$T(2^m) - 8T(2^{m-1}) = a \cdot 4^m$$
$$(x-8)(x-4) = 0 \Longrightarrow$$
$$T(2^m) = c_1 \cdot 8^m + c_2 \cdot 4^m \Longrightarrow$$
$$T(n) = c_1 \cdot n^3 + c_2 \cdot n^2 \in O(n^3) \Longleftarrow \textbf{NO HEMOS MEJORADO}$$

Idea.-.Podríamos mejorar si en vez de realizar 8 multiplicaciones hiciésemos 7. Esta es la idea del **Algoritmo de Strassen**.

Algoritmo de Strassen

Para realizar las multiplicación de las matrices descompone las matrices en cuatro subcuadrantes pero C_{ij} los define en base a un conjunto de operaciones. Así define:

$$C_{11} = P + S - T + V$$
$$C_{12} = R + T$$
$$C_{21} = Q + S$$
$$C_{22} = P + R - Q + U$$

Y las matrices P, S, T, U *y* V las obtiene como:

$$P = (A_{11} + A_{22}) \times (B_{11} + B_{22})$$
$$Q = (A_{12} + A_{22}) \times B_{11}$$
$$R = A_{11} \times (B_{12} - B_{22})$$
$$S = A_{22} \times (B_{21} - B_{11})$$
$$T = (A_{11} + A_{12}) \times B_{22}$$
$$U = (A_{21} - A_{11}) \times (B_{11} + B_{12})$$
$$V = (A_{12} - A_{22}) \times (B_{21} + B_{22})$$

Ahora tenemos 7 subproblemas de tamaño $\frac{n}{2}$ por lo tanto el tiempo de ejecución se puede formular como:

$$T(n) = 7T(\tfrac{n}{2}) + a \cdot n^2 \text{ haciendo el cambio de variable } n = 2^m$$
$$T(2^m) - 7T(2^{m-1}) = a \cdot 4^m$$
$$(x-7)(x-4) = 0 \Longrightarrow$$
$$T(2^m) = c_1 \cdot 7^m + c_2 \cdot 4^m \Longrightarrow$$
$$T(n) = c_1 \cdot n^{log_2(7)} + c_2 \cdot n^2 \in O(n^{log_2(7)}) \Longleftarrow \textbf{HEMOS MEJORADO}$$

Hemos mejorado ya que $n^{log_2(7)} = n^{2,81}$. No obstante las constantes que multiplican al polinomio an^2 con muchos mayores, ya que tenemos muchas más sumas y restas para calcular P,S,T,U y V . Por lo tanto es mejor aplicar el Algoritmo de Strassen cuando la entrada es muy grande (empíricamente con dimensiones de $n > 120$.

```
1  void Alg_Strassen(Matriz & A, Matriz &B,  Matriz &C){
2    if (A.filas()<Umbral)
3      Mult_Matrices(A, B,C); //Algoritmo clasico
4    else{
5     vector<Matriz> cuadrantesC(4),cuadrantesA(4),cuadrantesB(4);
6     //Iniciamos los cuadrantes
7     int k=0;
8     for (int i=0;i<2;i++)
9      for (int j=0;j<2;j++){
10      cuadrantesC[k]=submatriz(C,i*C.filas()/2,j*C.columnas()/2,
11                               (i+1)*C.filas()/2,(j+1)*C.columas()/2);
12      cuadrantesA[k]=submatriz(A,i*A.filas()/2,j*A.columnas()/2,
13                               (i+1)*A.filas()/2,(j+1)*A.columas()/2);
14      cuadrantesB[k]=submatriz(B,i*B.filas()/2,j*B.columnas()/2,
15                               (i+1)*B.filas()/2,(j+1)*B.columas()/2);
16      k++;
17     }
18
19     Matriz P(C.filas()/2,C.columnas()/2),
20            Q(C.filas()/2,C.columnas()/2),
21            R(C.filas()/2,C.columnas()/2),
22            S(C.filas()/2,C.columnas()/2),
23            T(C.filas()/2,C.columnas()/2),
24            U(C.filas()/2,C.columnas()/2),
25            V(C.filas()/2,C.columnas()/2);
26     P=  Alg_Strassen((cuadranteA[0]+cuadranteA[3]),(cuadranteB[0]+cuadranteB[3]));
27     Q=  Alg_Strassen((cuadranteA[1]+cuadranteA[3]),cuadranteB[0]);
28     R=  Alg_Strassen(cuadranteA[0],(cuadranteB[1]-cuadranteB[3]));
29     S=  Alg_Strassen(cuadranteA[3],(cuadranteB[2]-cuadranteB[0]));
30     T=  Alg_Strassen((cuadranteA[0]+cuadranteA[1]),cuadranteB[3]);
31     U=  Alg_Strassen((cuadranteA[2]-cuadranteA[0]),cuadranteB[0]+cuadranteB[1]);
32     V=  Alg_Strassen((cuadranteA[1]-cuadranteA[3]),cuadranteB[2]+cuadranteB[3]);
33
34     cuadranteC[0]= P+(S-(T+V));//sobrecargadas para matrices suma y resta
35     cuadranteC[1]= R+T;
36     cuadranteC[2]= Q+S;
37     cuadranteC[3]= P+(R-(Q+U));
38     //reconstruimos C
39     k=0;
40     for (int i=0;i<2;i++)
41      for (int j=0;j<2;j++){
42       Asigna_submatriz(C,i*C.filas()/2,j*C.columnas()/2,
43                        (i+1)*C.filas()/2,(j+1)*C.columas()/2,cuadranteC[k]);
44      k++;
45     }
46
47  }
48 }
```

2.6 Otros problemas

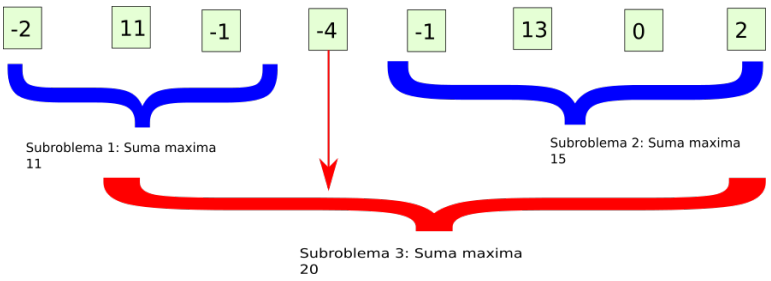
2.6.1 Subsecuencia más larga

Dados n enteros cualesquiera $a_1, a_2, \cdots, a_n$, necesitamos encontrar el valor de la expresión:

$$max_{0 \leq j < n} \left(\sum_{k=i}^{j} a_k \right)$$

que calcula el máximo de las sumas parciales de elementos consecutivos. Por ejemplo si tenemos los siguientes enteros: $(-2, 11, -4, 13, 5, -2)$ la solución al problema es 20 (se consigue sumando desde a_1 hasta a_3). Diseñar un algoritmo DyV con tiempo $nlogn$ que resuelva el problema.
Solución. Se divide el problema en tres subproblemas más pequeños, sobre cuyas soluciones construiremos la solución total. En este caso la subsecuencia que obtiene la suma máxima puede encontrarse en una de las tres partes. Así puede estar en la mitad del vector a la izquierda, o en la mitad del vector a la derecha, o bien contiene al elemento en la mitad del vector y la subsecuencia con suma máxima contiene elementos a la izquierda y derecha del centro. Las tres soluciones se combinan mediante el cálculo de su máximo para obtener la suma pedida. Los dos casos pueden resolverse recursivamente. Respecto al tercero, podemos calcular la subsecuencia de suma máxma en el subvector que se inicia en el ultimo elemento de la subsecuencia maxima de la mitad izquierda y termina donde comienza la subsecuencia máxima de la mitad a la derecha.

En el ejemplo que se ve a la derecha la mitad izquierda devuelve una subsecuencia máxima que suma 11. Se inicia en el elemento 11 y termina en él (solamente contiene a 11). La mitad derecha devuelve una subsecuencia máxima que suma 15, y contiene (13,0,2). El tercer problema tiene que analizar el puntos siguiente donde terminó la subsecuencia izquierda ,siendo en -1, hasta antes donde se inicia la secuencia máxima derecha, en -1. La parte izquierda aporta 11 si añadimos -1 obtenemos 10 si añadimos -4 obtenemos 6 si añadimos -1 obtenemos 5, y ahora finalmente le añadimos el máximo de la derecha que es 15, obteniendo 20. Luego el tercer subproblema contiene (11,-1,-4,-1,13,0,2), y esta es nuestra solución.




```
1  /**
2   @brief Obtiene la subsecuencia de suma maxima de un conjunto de elementos
3   @param datos: el conjunto de elmentos enteros
4   @param inicio: indice donde comienza la secuencia
5   @param fin: indice donde finaliza la secuencia
6   @param bini: indice donde comienza la mejor secuencia que tiene suma maxima. ES MODIFICADO
```

```
7    @param bfin: indice donde termina la mejor secuencia que tiene suma maxima. ES MODIFICADO
8    @return la suma maxima de la subsecuencia definida por bini-bfin
9   */
10  int SubSecuencia(vector<int>& datos,int inicio,int fin, int& bini,int &bfin){
11   if (fin-inicio==1){//Caso base solo un elemento
12        bini=inicio;
13        bfin=inicio+1;
14       return datos[inicio];
15   }
16   else{
17       int mitad=(fin+inicio)/2.0; //indice del punto medio
18       int bi_izq,bf_izq,bi_dr,bf_dr;
19
20       int s_izq,s_dr;
21       s_izq=SubSecuencia (datos,inicio,mitad,bi_izq,bf_izq); //Mitad izquierda
22
23       s_dr=SubSecuencia(datos,mitad,fin,bi_dr,bf_dr);//Mitad derecha
24
25       //El maximo de las dos mitades
26       int max;
27       if (s_izq>s_dr){
28            max=s_izq;
29            bini=bi_izq;
30            bfin=bf_izq;
31       }
32       else {
33            max=s_dr;
34            bini=bi_dr;
35            bfin=bf_dr;
36       }
37       //Desde al centro hacia la izquierda
38       int posi=mitad-1,pi;
39       int suma=0,max_centro=numeric_limits<int>::min();
40       for (; posi>=bf_izq;posi--){
41          suma+=datos[posi];
42          if (max_centro<suma){
43              max_centro=suma;
44              pi=posi;
45          }
46       }
47       //Se le puede anhadir el subvector que se obtuvo en la mitad izquierda
48       if (suma+s_izq>max_centro){
49          pi=bi_izq;
50          max_centro=suma+s_izq;
51       }
52
53       //Desde al centro hacia la derecha
54       int pf=mitad;
55       posi=mitad;
56       suma=max_centro;
57       for (; posi<bi_dr;posi++){
58          suma+=datos[posi];
59          if (max_centro<suma){
```

```
60              max_centro=suma;
61              pf=posi;
62           }
63        }
64        //Se le puede anhadir el subvector que se obtuvo en la mitad derecha
65        if (suma+s_dr>max_centro){
66           pf=bf_dr;
67           max_centro=suma+s_dr;
68
69        }
70
71        if (max_centro>max){
72          max=max_centro;
73          bini=pi;
74          bfin=pf;
75        }
76
77        return max;
78     }
79  }
```

2.6.2 Vector Peinado

Un vector $A[0,\cdots,2n]$ con $2n+1$ elementos está peinado si cumple:

$$A[0] \leq A[1] \geq A[3] \leq A[4] \geq A[5] \cdots \leq A[2n-1] \geq A[2n]$$

Definir un función basada en divide y vencerás para peinar un vector.
Solución. Para realizar este ejercicio usaremos varios algoritmos vistos anteriormente. Así los pasos a seguir son:

1. En primer lugar usaremos el problema que permite descubrir cual es el elemento que se situaría en $\frac{n}{2}$ si el vector estuviera ordenado. Para ello usamos la función Selección (ver el problema de la sección 2.3.2). Así obtenemos la mediana. Esto nos cuesta como máximo $O(nlog_2(n))$.
2. A continuación ordenamos la mitad izquierda usando mergesort y la mitad derecha usando mergesort. Esto nos cuesta como máximo $O(nlog_2(n))$.
3. Y por último intercalamos la parte derecha en la parte izquierda. Esto nos cuesta $O(n)$.

El algoritmo podría quedar como:

```
1  void Peina_Vector( int * v,int n,int s){
2     int mediana = Seleccion(v,n,n/2); //Esta funcion nos deja en v[n/2] la mediana.
3     Mergesort(v,n/2); //ordenamos la parte izquierda
4     Mergesort(v+n/2,n-n/2); //ordenamos la parte derecha
5     int * aux = new int[n];
6     for (int i=0;i<n/2;i+=2){
7      aux[i]= v[i];
8      aux[i+1]= v[i+n/2];
9     }
10    delete[]v;
11    v= aux;
```

```
12
13 }
```

2.6.3 Elimina Repetidos

Dado un vector con n elementos donde se sabe que algunos están repetidos. Diseñar un algoritmo divide y vencerás para eliminar todos los elementos repetidos. Si aparecen elementos repetidos aparecen consecutivos. Tras la eliminacion de los elementos repetidos los elementos deben mantener el mismo orden. Por ejemplo:

$$\{9,9,9,8,8,0,0,4,4\}$$
trás eliminar los elementos obtenemos
$$\{9,8,0,4\}$$

Los pasos a seguir son:

1. Eliminamos los repetidos en la mitad izquierda
2. Eliminamos los repetidos en la mitad derecha
3. Puede que queden elemetos repetidos a la izquierda y derecha del elemento en la mitad. Esto se eliminan

El código queda de la siguiente forma:

```
1  void EliminaRepetidos(int *v,int inicio,int fin){
2     if (fin-inicio>1){
3
4      int mitad = (fin+inicio)/2;
5
6      EliminaRepetidos(v,inicio,mitad);
7      EliminaRepetidos(v,mitad,fin);
8      //miramos si desde el centro a izquierda o derecha hay repetidos
9      int i,k=mitad-1,j;
10     while (k>=0 && v[k]==v[mitad]) k--;
11
12     k++;
13     i=k;//punto de incio a la izquierda iguales a v[mitad]
14
15     //miramos si desde el centro a izquierda o derecha hay repetidos
16     k=mitad+1;
17     while (k<n & v[k]==v[mitad]) k++;
18     k--;
19
20     j=k;//punto de incio a la derecha iguales a v[mitad]
21     if (i<j){
22       int n= (fin-inicio)-(j-i);
23       int * aux = new int[n];
24       for (int l=0;l<i;l++)
```

```
25            aux[l]=v[l];
26          aux[i]=v[i];
27          k=i+1;
28          for (int l=j+1;j<fin;j++,k++)
29             aux[k]= v[l];
30          delete[]v;
31          v=aux;
32       }
33     }
34  }
```

2.6.4 Mediana de dos vectores

Sean X e Y dos vectores de tamaño n, ordenados de forma no decreciente. Se pide implementar un algoritmo divide y vencerás para calcular la mediana de los $2n$ elementos que contiene X e Y. Recordad que la mediana es aquel elemento que una vez ordenado el vector, deja la mitad de los elementos a cada uno de los lados. Como en nuestro caso tenemos $2n$ elementos buscamos el elemento en la posición n de la unión ordenada de X e Y.

Solución.- Para diseñar este algoritmo tenemos que tener en cuenta las siguientes consideraciones:

- El caso base ocurre cuando tenemos dos vectores de un elemento. En este caso la mediana será el mínimo de ambos números.
- En otro caso sea Z el vector resultado al mezclar los dos vectores X e Y. Sean mx, my y mz la mediana de X, Y y Z, respectivamente. Los posibles casos que se puede dar son:
  - Si $mx = my$ entonces $mz = mx = my$. Ya que al unir los vectores dejaran a la izquierda y derecha de mz n elementos.
  - Si $mx < my$ debemos derivar donde esta mz. Si tomamos el rango de valores $[mx, my]$ debemos plantear cuantos elementos nos queda a la izquierda de este rango y cuantos nos queda a la derecha. Pues a la izquierda nos queda $\frac{n}{2}$ y a la derecha $\frac{n}{2}$. Mayores a mx debe haber al menos por el vector X $\frac{n}{2}$ y menores a my otros tantos. Por lo tanto en el rango $[mx, my]$ debe haber al menos n elementos. Esto quiere decir que como la mediana mz debe dejar a sus lados n elementos mz debe estar entre $[mx, my]$. Lo que implica buscar la mediana en la mitad derecha de X y en la mitad izquierda de Y.
  - Si $my < mx$ igualmente debemos razonar donde esta mz. Si tomamos el rango de valores $[my, mx]$ debemos plantear cuantos elementos nos queda a la izquierda de este rango y cuantos nos queda a la derecha. Pues a la izquierda nos queda $\frac{n}{2}$ y a la derecha $\frac{n}{2}$. Mayores a my debe haber al menos por el vector Y $\frac{n}{2}$ y menores a mx otros tantos. Por lo tanto en el rango $[my, mx]$ debe haber al menos n elementos. Esto quiere decir que como la mediana mz debe dejar a sus lados n elementos, mz debe estar entre $[my, mx]$. Lo que implica buscar la mediana en la mitad derecha de Y y en la mitad izquierda de X.

Siguiendo estas consideraciones el algoritmo quedaría como sigue:

```
1  /**
2   @brief Busca la mediana de la fusion de dos vectores ordenados
3          con el mismo numero de elementos
4   @param X: el primer conjunto de elementos ordenados de forma creciente
```

```
 @param Y: el segundo conjunto de elementos ordenados de forma creciente
 @param iniX: indice donde comienza los elementos de X
 @param ultX: indice limite superior de X.
 @param iniY: indice donde comienza los elementos de Y
 @param ultY: indice limite superior de Y.
 @return devuelve la mediana del vector fusion de X e Y.
 */

int Mediana(const int *X, const int *Y,int iniX,int ultX,int iniY,int ultY,int n) {
  if (iniX==ultX && iniY==ultY)
   // se devuelve el menor
     return (X[ultX]>Y[ultY]? Y[ultY]: X[ultX]);
  else{
    if (ultX-iniX+1==2){ // dos elementos
     if (X[ultX]<Y[iniY]) return X[ultX];
     else if (Y[ultY]<X[iniX]) return Y[ultY];
     else return minimo(X[iniX],Y[iniY]);
    }
    else {
      int mitad=n/2;
      int posX= iniX+mitad;
      int posY= iniY+mitad;
      if (X[posX]==Y[posY]) return X[posX];
      else
       if (X[posX]<Y[posY])
        return Mediana(X,Y,ultX-mitad,ultX,iniY,iniY+mitad,mitad);
       else
        return Mediana(X,Y,iniX,iniX+mitad,ultY-mitad,ultY,mitad);
    }
   }
}
```

Ejercicio 2.1
Obtener la eficiencia del problema Mediana de dos vectores □