Tema 4:

"Algoritmos Recursivos de Ordenación y Búsqueda en Vectores"

Algoritmos Recursivos de Ordenación y Búsqueda

Parte I

Algoritmos Recursivos de Búsqueda sobre vectores

1.1.- Búsqueda Secuencial (proceso)

- Es el método más sencillo.
- NO es necesario que el array esté ordenado.
- Consiste en ir comparando el valor buscado con los sucesivos valores del array, hasta que:
  - sean Iguales => elemento encontrado. Devuelvo su posición (i)
  - finalice el array => elemento NO encontrado. Devuelvo -1 o mensaje "elemento no encontrado".

Recordad...

19	5	13	4	7

1.1.- Búsqueda Secuencial (algoritmo recursivo)

19	5	13	4	7

```
int BusquedaSecuencialRecursiva(int v[],int x, int i)
{
  if (i == MAX_ELEM) //elemento no está en el vector
      return -1;
  else
      if (v[i]== x) //elemento encontrado
            return i;
      else
            return BusquedaSecuencialRecursiva(v,x,i+1);  //sigo buscando
}
```


1.2.- Búsqueda Binaria o Dicotómica (proceso)

Recordad...

- Es necesario que el **array** esté **ordenado**.
- El algoritmo consiste en comprobar si el valor es menor o mayor que el elemento **central del array**.
- Si es **menor** indica que el valor buscado se encuentra en la primera mitad del array, aplicándose el mismo método a esa mitad, descartándose la segunda mitad.
- Si es **mayor** indica que el valor buscado se encuentra en la segunda mitad del array y por tanto se aplicará el mismo método a esa segunda mitad, descartándose la primera mitad.
- Las comparaciones con el valor central se hacen hasta que coincida con el valor buscado o hasta que la mitad del array no tenga elementos.

Sea el array de enteros A (-8, 4, 5, 9, 12, 18, 25, 40, 60), buscar la clave, clave = 40.

1.

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]
-8	4	5	9	12	18	25	40	60

bajo = 0
alto = 8

↑ central

$central = \frac{bajo + alto}{2} = \frac{0+8}{2} = 4$

clave (40) > a[4] (12)

2. Buscar en sublista derecha

18	25	40	60

bajo = 5
alto = 8

↑

$central = \frac{bajo + alto}{2} = \frac{5+8}{2} = 6$ (división entera)

clave (40) > a[6] (25)

3. Buscar en sublista derecha

40	60

bajo = 7
alto = 8

↑

$central = \frac{bajo + alto}{2} = \frac{7+8}{2} = 7$

clave (40) = a[7] (40) (búsqueda con éxito)

4. El algoritmo ha requerido 3 comparaciones frente a 8 comparaciones ($n - 1$, $9 - 1 = 8$) que se hubieran realizado con la búsqueda secuencial.

Búsqueda Binaria o Dicotómica (proceso)

medio = central
i = izquierdo = bajo
j = derecho = alto

1.2.- Búsqueda Binaria o Dicotómica (algoritmo recursivo)

```
int BusquedaBinariaRecursiva(int v[],int x, int i, int j)
{
  int medio;

  if (i>j)
    return -1;

  medio = (i+j) / 2;

  if (v[medio] < x)
     return BusquedaBinariaRecursiva(v,x,medio+1,j);
  else
     if (v[medio] > x)
           return BusquedaBinariaRecursiva(v,x,i,medio-1);
     else
           return medio;
}
```

medio = central
i = izquierdo = bajo
j = derecho = alto

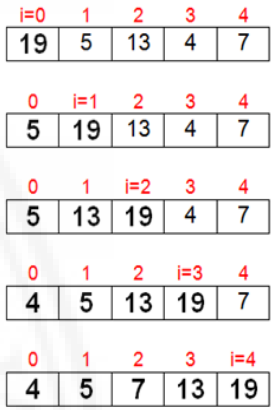
Parte II

Algoritmos Recursivos de Ordenación sobre vectores

2.1.- Inserción Directa (proceso)

Recordad...

- Busca para cada elemento (desde la posic. 0 del vector en adelante), la posición que le corresponde entre los elementos YA ordenados.
- Cuando considera el elemento que ocupa la posición i, los elementos anteriores (posiciones 0.. i-1) ya estarán ordenados.



2.1.- Inserción Directa (algoritmo recursivo)

```
void insercionDirectaRecursivo(int v[],int i)
{
   int elemAInsertar;
   int posIns;

   if ( i < MAX_ELEM) {
      elemAInsertar = v[i];
      posIns = i ;

      for( ; (posIns>0) && (elemAInsertar < v[posIns-1]); posIns-- )
      {
            v[posIns] = v[posIns - 1];
       }
       v[posIns] = elemAInsertar;
       // Ordenada por Insercion v[i]
       //Ordenar las componentes restantes (long del vector - i - 1)
       insercionDirectaRecursivo(v, i + 1);
     }
}
```

¡Abrimos hueco!

2.2.- La estrategia Divide y Vencerás

- Estrategia recursiva eficiente para la resolución de problemas.

1º.- **DIVIDIR** el problema original de talla X en A subproblemas disjuntos de talla equilibrada (ej. dividir un array por el elemento central)

2º.- **VENCER**: Resolver los subproblemas de manera recursiva hasta alcanzar los respectivos casos base.

3º.- **COMBINAR**: las soluciones de los subproblemas para la obtención del problema original.

2.2.- La estrategia Divide y Vencerás

- Pseudocódigo de un algoritmo "Divide y Vencerás"
  - Adaptación del **Esquema General Recursivo**

```
TipoResultado vencer (TipoDatos x){
    TipoResultado resMetodo, resLlamada1, resLlamada2,..., resLlamadaA;

    if (casoBase(X))
       resMetodo = solucionBase(x);
    else
    {
        int c = dividir(x);

        resLlamada1 = vencer(x/c);
        …..
        resLlamadaA = vencer(x/c);
        resMetodo = combinar(x, resLlamada1, resLlamada2,.., resLlamadaA);

        return resMetodo;
    }}
```


2.3.- La estrategia Divide y Vencerás de Ordenación Rápida

- Estrategia Divide y Vencerás:

  1º.- DIVIDIR la talla del problema en dos subproblemas de talla aproximadamente la mitad de la original (c=2).

  2º.- VENCER los dos subproblemas (A = 2).

  3º.- COMBINAR los subproblemas resueltos.

- **Dividir y Combinar** se debe realizar, como máximo, en un coste lineal. **(O(n))**
- Los métodos Quicksort y Mergesort utilizan una estrategia DyV para ordenar arrays con un coste de **$x * \log_2(x)$**

2.3.1.- Mergesort

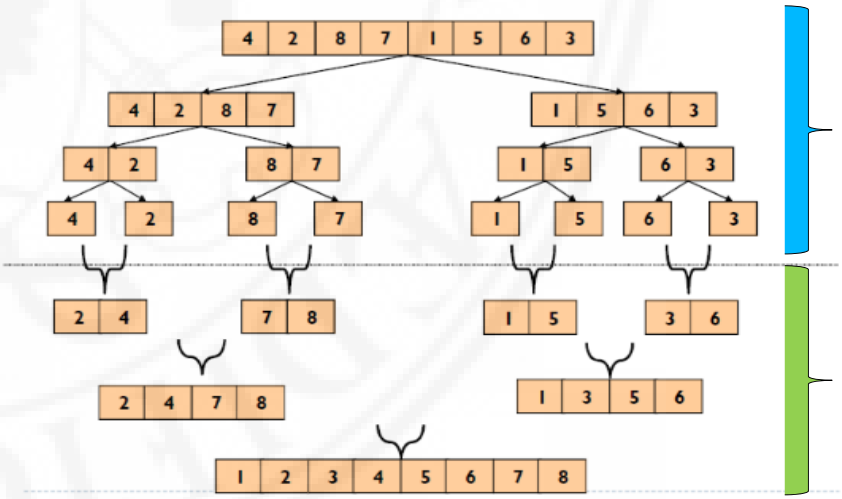
- Se basa en la técnica "Divide y Vencerás"
- **Pasos: (Estrategia de Fusión o Mezcla)**

  1.- DIVIDIR: **Se divide el vector** en dos partes de, aproximadamente, la mitad del tamaño.

  2.- VENCER: **Se ordena por fusión o mezcla cada una de esas partes** utilizando este mismo método (2 llamadas recursivas).

  3.- Tomando las dos partes ordenadas, se las intercala de forma ordenada, para obtener el vector original ordenado. (proceso iterativo) ➔ COMBINAR

Mergesort



Mergesort

```
void merge_sort(int vector[], int inicio, int fin)
{
    int largo = fin - inicio;
    if (largo < 2) {
        return;
    }

    int medio = inicio + (largo / 2);
    merge_sort (vector, inicio, medio);
    merge_sort (vector, medio, fin);
    merge (vector, inicio, medio, fin);
}
```

¡Recursiva!

- La **fusión o Mezcla natural** permite obtener un vector ordenado a partir de dos vectores ordenados, con un coste Lineal con la suma de los tamaños de los vectores.
- Su utilización en MergeSort implica la fusión de dos subvectores almacenados de manera implícita en un mismo array.
- Requiere un **array auxiliar** para almacenar el resultado de la fusión.

Merge

(Fusión o Mezcla natural)

```
void merge (int vector[], int inicio, int medio, int fin)
{
    int pos_1 = inicio;
    int pos_2 = medio;
    int aux[fin-inicio];
    int pos_a = 0;

    // Intercala ordenadamente
    while ( (pos_1 < medio) && (pos_2 < fin) ) {
        if ( vector[pos_1] <= vector[pos_2] ) {
            aux[pos_a] = vector[pos_1];
            pos_a++; pos_1++;
        } else {
            aux[pos_a] = vector[pos_2];
            pos_a++; pos_2++;
        }
    }
    // Copia lo que haya quedado al final del primer vector
    while (pos_1 < medio) {
        aux[pos_a] = vector[pos_1];
        pos_a++; pos_1++;
    }
    // Copia lo que haya quedado al final del segundo vector
    while (pos_2 < fin) {
        aux[pos_a] = vector[pos_2];
        pos_a++; pos_2++;
    }

    // Copia los valores del vector auxiliar al original
    int a = 0;
    int i = inicio;
    while (i < fin) {
        vector[i] = aux[a];
        i++; a++;
    }
}
```

¡Iterativa!

Problemas de Mergesort

- En mezclaDyV (merge()), la fusión de los dos subvectores se hace sobre un **vector auxiliar**.
  - Utiliza espacio adicional lineal con la talla del problema
  - Copia de valores del vector auxiliar sobre el vector principal tras la fusión.
- Para tamaños elevados de vector esta limitación puede suponer un tamaño excesivo de memoria.

2.3.2.- Ordenación Rápida (Quicksort)

(Idea general)

- Es **el más eficiente** de los conocidos.
- Consiste en **elegir un** elemento cualquiera del array, llamado **Pivote**, y en una 1ª pasada, colocar todos los elementos menores que él a su izquierda y los mayores que él a su derecha.

  ***Pivote**: 1er elemento, último elemento o elemento que está en la posición central del array...*
- Se repite el mismo proceso con cada nuevo subvector (izquierdo/derecho) invocando Quicksort de forma **recursiva**.
- El algoritmo finaliza cuando el número de elementos del subvector que se va a procesar es <= 1.

2.3.2.- Ordenación Rápida (Quicksort)

(Proceso)

- 1º Elección de un **elemento pivote** del vector.
- 2º **DIVIDIR**: Realizar una partición del vector v en dos grupos, empleando como mucho un tiempo lineal con la talla del vector.

  Vizqdo = {Elementos menores que el pivote}

  Vderecho = {Elementos mayores que el pivote}

  - El elemento pivote queda en su posición ordenada
- 3º **VENCER**: Aplicar Quicksort recursivamente a Vizquierdo y Vderecho para resolver los subproblemas de talla mitad de la original.
- No hace falta la fase **COMBINAR** porque la realizar la partición se hace implícitamente.
- Al DIVIDIR es posible generar vectores vacíos.

1. *Lista original* 5 2 1 9 3 8 7

   pivote elegido 5

   sublista izquierda1 (elementos menores que 5) 2 1 3

   sublista derecha1 (elementos mayores o iguales a 5) 9 8 7

2. El algoritmo se aplica a la sublista izquierda

   Sublista Izda1 2 1 3

   pivote

   sublista Izda 1
   sublista Dcha 3

   Sublista Izda1 *Izda* *pivote* *Dcha*

   1 2 3

3. El algoritmo se aplica a la sublista derecha

   Sublista Dcha1 9 8 7

   pivote

   sublista Izda 7 8
   sublista Dcha

   Sublista Dcha1 *Izda* *pivote* *Dcha*

   7 8 9

4. *Lista ordenada final*

   Sublista izquierda *pivote* *Sublista derecha*

   1 2 3 5 7 8 9

Ordenación Rápida (Quicksort)

(proceso)

Pivote: 1er elemento de la lista

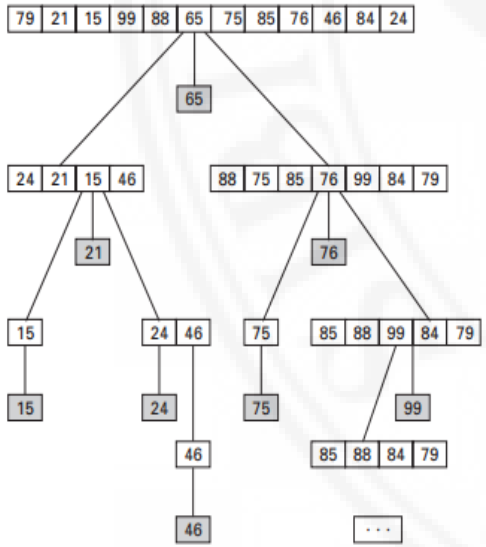


Figura 6.2. Ordenación rápida eligiendo como pivote el elemento central.

Ordenación Rápida (Quicksort)

(proceso)

Pivote: elemento central de la lista

2.3.2.- Ordenación Rápida (Quicksort)

```
#define N 5
void QuickSort(int A[], int Desde, int Hasta);
void main(void)
{
  int A[N] = {19, 5, 13, 4, 7};
  int i;

  QuickSort(A, 0, N-1);
}

void QuickSort(int A[], int Desde, int Hasta)
{
  int Izq, Der, Mitad, Aux;

  Izq = Desde;
  Der = Hasta;
  Mitad = A[(Izq+Der) / 2];
  do
  {
    for ( ;Izq <= Der && A[Izq] < Mitad; Izq++);
    for ( ;Der >= Izq && A[Der] > Mitad; Der--);
    if (Izq <= Der)
    {
      Aux   = A[Izq];      A[Izq] = A[Der];      A[Der] = Aux;
      Izq++; Der--;
    }
  } while (Izq <= Der);

  if (Izq < Hasta)
      QuickSort(A, Izq, Hasta);
  if (Der > Desde)
      QuickSort(A, Desde, Der);
}
```

Estrategias de selección del pivote en Quicksort

- La selección del pivote debe minimizar la posibilidad de obtener una partición desequilibrada, incluso para instancias degeneradas:
  - Vector inicialmente ordenado ascendentemente
  - Vector donde todas sus componentes son iguales
- Diferentes estrategias de selección del pivote (y partición):
  - Primer elemento del vector
  - Elemento central
  - Último elemento del vector
  - Mediana de 3 elementos del vector

FIN

Acerca de Creative Commons

Licencia CC (Creative Commons)

- Este tipo de licencias ofrecen algunos derechos a terceras personas bajo ciertas condiciones.
- Este documento se encuentra bajo una Licencia Creative Commons Atribución-NoComercial-SinDerivadas 3.0 Unported.
- Su significado es el siguiente:
- **Reconocimiento - NoComercial - SinObraDerivada (by-nc-nd): No se permite un uso comercial de la obra original ni la generación de obras derivadas.**
- *Para ver una copia de esta licencia, visite http://creativecommons.org/licenses/by-nc-nd/3.0/ o envie una carta a Creative Commons, 171 Second Street, Suite 300, San Francisco, California 94105, USA.*