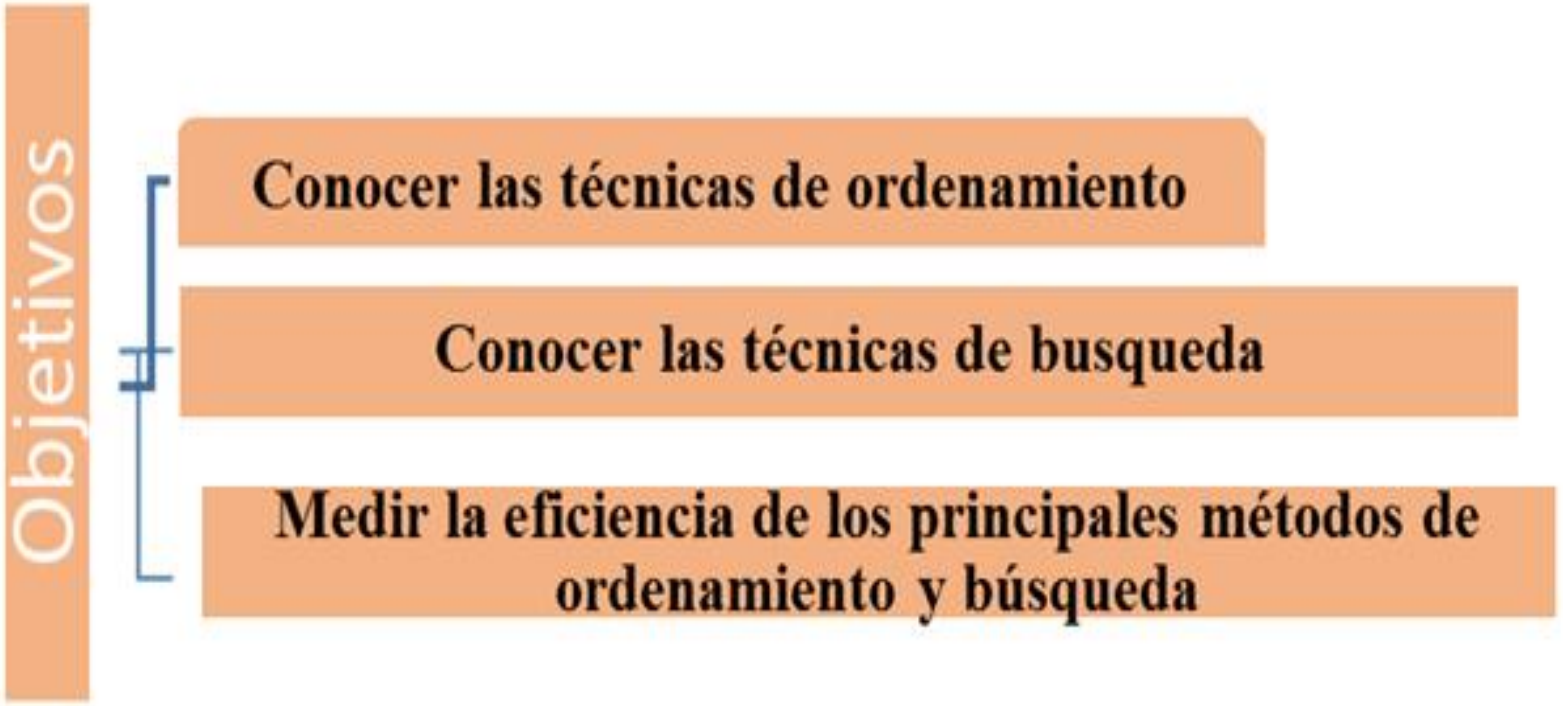
Universidad Nacional Mayor de San Mar os
Facultad de Ingeniería de Sistemas

Algorítmica III

Guía 3
Técnicas de ordenamiento y búsqueda

Mg Augusto Cortez Vásquez


Objetivos
Conocer las técnicas de ordenamiento
Conocer las técnicas de busqueda
Medir la eficiencia de los principales métodos de ordenamiento y búsqueda

En toda discusión, el primero que se incomoda y grita suele ser el que tiene menos razón....
Por eso un necio puede refutar a un sabio en cualquier momento, pero un sabio no puede convencer a un necio nunca.

Li Po

Leonardo Pisano Fibonacci **(1175-1250)**

Al estudiar relaciones de recurrencia inevitablemente tenemos que hacer referencia a la relación de Fibonacci, dada por Leonardo de Pisa (cerca de 1175-1250) en 1202. Leonardo aborda un problema relacionado con el numero de parejas de conejos que se producen durante un año si se parte de una sola pareja que engendra otra pareja al final de cada mes. Cada pareja nueva comienza a reproducirse en forma similar un mes después de su nacimiento; y suponemos que ninguna pareja muere durante ese año. Por tanto, al final del primer mes tenemos dos pares de conejos; tres pares después de dos meses; cinco pares después de tres meses, y así sucesivamente.

Tipos de algoritmos de ordenación□

- **Algoritmos de ordenación interna:**
  En memoria principal (acceso aleatorio)

Algoritmos de ordenación externa:
En memoria secundaria (restricciones de acceso).

Introducción

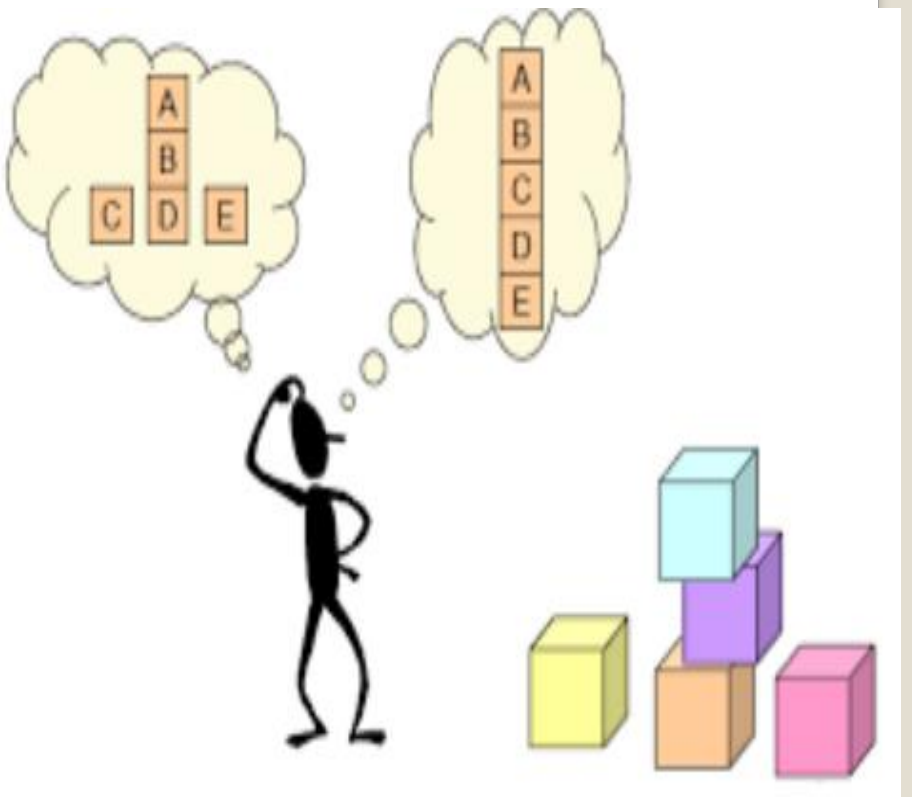
El método de ordenamiento es quizás el método de ordenación más popular entre los estudiantes principiantes de computación por su fácil comprensión y programación, y es probablemente el menos eficiente.

Este algoritmo consiste en comparar (N-1) pares de elementos adyacentes e intercambiarlos entre sí (si conviene) hasta que todos se encuentren ordenados. Se realizan (N-1) pasadas, transportando en cada una de las mismas el mayor elemento, como es en nuestro caso, o viceversa

Todos los algoritmos de ordenamiento requieren comparar dos elementos y decidir si se intercambian de lugar o no, la diferencia entre los algoritmos que se mencionaran es como elegir los dos elementos.

¿Porque es importante la ordenación?

Buscar un elemento en un vector ordenado es mas sencillo que buscar en uno que no lo este. Esto no solo es cierto para algoritmos a ser ejecutados por computadoras, sino también para el proceder cotidiano de las personas. Por ejemplo, encontrar el nombre de alguien en un listado telefónico es fácil, mientras que buscar un teléfono sin conocer el nombre de la persona, es prácticamente imposible.



Con frecuencia muchas de las informaciones que se procesan en un computador se presentan en forma ordenada, esto facilita su manejo,

por ejemplo:

El listado de cursos ofrecidos en la Facultad se encuentran ordenados

La relación de alumnos matriculados en el curso de algorítmica se encuentra ordenado

El periódico, calendario de eventos esta organizado por fechas,

El índice de un libro esta ordenado

Los archivos de un directorio, están generalmente ordenados

Sea A: una secuencia de n elementos A_1, A_2, A_3, ... A_n en memoria.

Ordenar A se refiere a la operación de ordenar los contenidos de A de forma que estén ordenados en orden creciente (numérico o lexicografito),

es decir

$A_1 \leq A_2 \leq A_3 \leq \ ... \leq \ A_n$

Todos los métodos de ordenación eligen dos elementos y los comparan, luego los intercambian si no están en orden. La diferencia entre los métodos esta en como eligen los dos elementos a comparar.

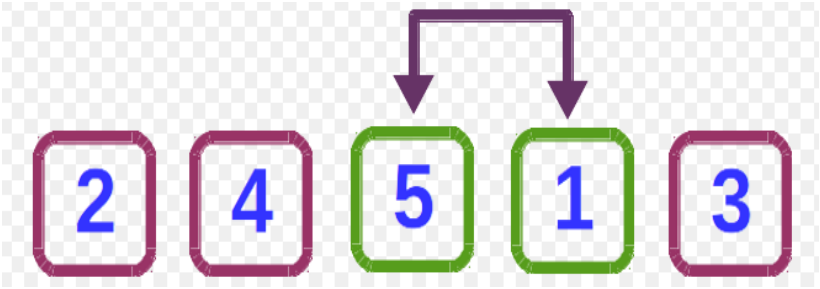
Como A tiene n elementos, existen n! permutaciones de los n elementos

Se presentaran algunos de los principales métodos de ordenación

- Ordenación por burbuja
- Ordenación por Shell
- Ordenación Rápida
- Ordenación por inserción
- Ordenación por selección
- Ordenamiento Merge Sort
- Ordenamiento por árbol en montón
- Ordenamiento por árbol binario de búsqueda

Ordenación por burbuja (Bubblesort)

El algoritmo mas sencillo, sobre para todo los que recién se inician en conocer las técnicas de ordenamiento es algoritmo de burbuja. Consiste en ciclar repetidamente a través de la lista, comparando elementos adyacentes de dos en dos. Si un elemento es mayor que el que está en la siguiente posición se intercambian




```
Acción   CAMBIAR( X,Y)
 Inicio
                    AUX   = X
                     X     =  Y
                     Y     = AUX
 fin_accion
```

Ordenación por burbuja (Bubblesort)

```
Acción  BURBUJA(V,N)
 Inicio
        Para I desde 1 hasta N-1
              Para j desde i+1 hasta N
                  Si (V[i] > V[j])
                        CAMBIAR(V[i] , V[j])
                   FinSi
              FinPara
          FinPara
 fin_accion
```

Ventajas:

Fácil implementación.
No requiere memoria adicional.

Desventajas:

Muy lento.
Realiza numerosas comparaciones.
Realiza numerosos intercambios.
Este algoritmo es uno de los más pobres en rendimiento.
No es recomendable usarlo.

Tan sólo se muestra para ilustrar su sencillez.

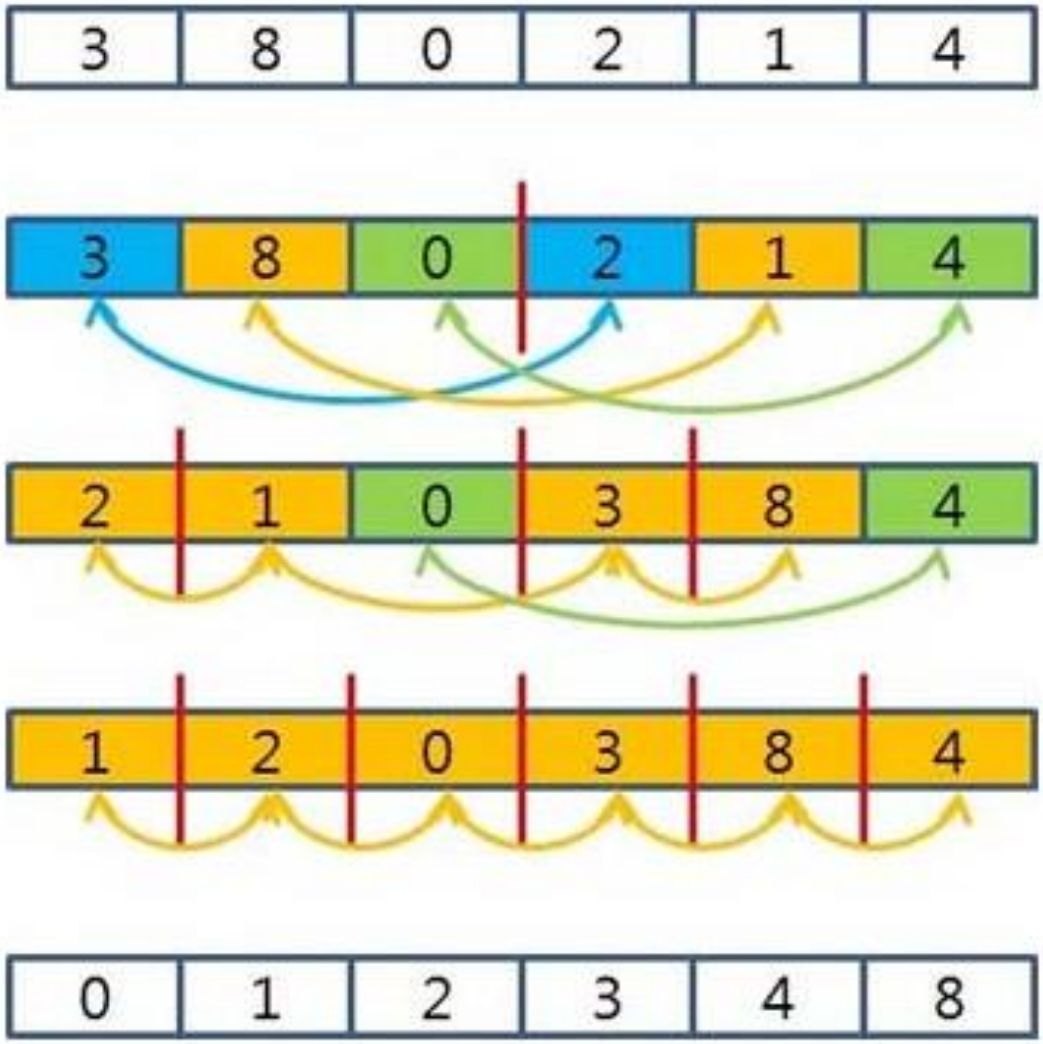
```
void bubbleSort (double v[])
{
   double tmp;
   int i,j;
   int N = v.length;

   for (i = 0; i < N - 1; i++)
        for (j = N - 1; j > i; j--)
             if (v[j] < v[j-1]) {
                 tmp = v[j];
                 v[j] = v[j-1];
                 v[j-1] = tmp;
              }
}
```

Ordenación por SHELL

El **ordenamiento Shell** (**Shell** sort en inglés) es un algoritmo de **ordenamiento**. El método se denomina **Shell** en honor de su inventor Donald **Shell**. Su implementación original, requiere $O(n^2)$ comparaciones e intercambios en el peor caso.

El algoritmo Shell es una mejora de la ordenación por inserción, donde se van comparando elementos distantes, al tiempo que se los intercambian si corresponde.

A medida que se aumentan los pasos, el tamaño de los saltos disminuye; por esto mismo, es útil tanto como si los datos desordenados se encuentran cercanos, o lejanos.
Es bastante adecuado para ordenar listas de tamaño moderado, debido a que su velocidad es aceptable y su codificación es bastante sencilla.

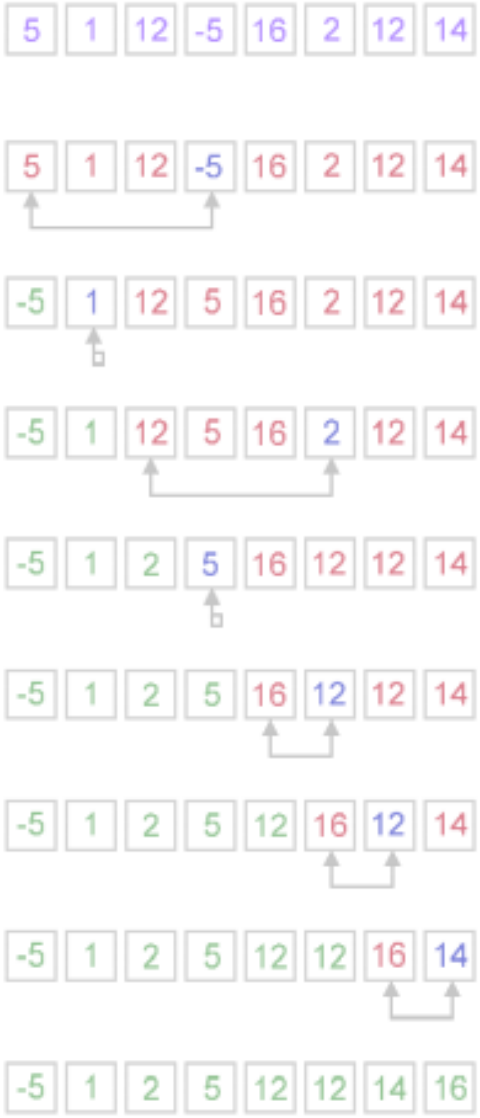

3 8 0 2 1 4
3 8 0 2 1 4
2 1 0 3 8 4
1 2 0 3 8 4
0 1 2 3 4 8

```
int[] ShellSort(int[] array){
    int aux = array.length / 2;
    while (aux > 0) {
        for (int i = 0; i < array.length - aux; i++) {
            int j = i + aux;
            int tmp = array[j];
            while (j >= aux && tmp > array[j - gap]) {//tmp <
array[j-gap] otro orden
                array[j] = array[j - aux];
                j -= aux;
            }
            array[j] = tmp;
        }
        if (aux == 2) {
            aux = 1;
        }else{
            aux /= 2.2;
        }
    }
    return array;
}
```

Ordenación por Selección del menor

```
Accion SELECCION_MENOR( V, N)

Inicio
      Para i desde 1 hasta N-1
                  menor=V[i]
                  k=i
                   Para j desde i+1 hasta  N
                              Si (V[j]< menor)
                                          menor=V[j]
                                          k=j
                              FinSi
                    FinPara
                  V[k]=V[i]
                  V[i]=menor
       FinPara
Fin
```

```
void selectionSort (double v[])
{
  double tmp;
  int i, j, pos_min;
  int N = v.length;

  for (i=0; i<N-1; i++) {

      // Menor elemento del vector v[i..N-1]
      pos_min = i;

      for (j=i+1; j<N; j++)
          if (v[j]<v[pos_min])
             pos_min = j;

      // Coloca el mínimo en v[i]

      tmp = v[i];
      v[i] = v[pos_min];
      v[pos_min] = tmp;
  }
}
```



Ordenación por Inserción

El algoritmo de inserción es un método que se comporta en forma similar a un juego de cartas. Este método consiste en insertar un elemento en el vector en una parte ya ordenada de este vector y comenzar de nuevo con los elementos restantes. Este método es conocido por el nombre de método de la baraja., por ser utilizados generalmente por jugadores de cartas.

Sea **A**: A_1, A_2, A_3, ... A_n en memoria. El método de inserción consiste en examinar A_1 hasta A_2., insertando cada elemento A_k en su lugar adecuado del subarreglo previamente ordenado A_1, A_2, ... A_{k-1}

```
Accion INSERCION(V, N)
Inicio
        Para i desde 2 hasta N
                temp = V[i]
                j = i – 1
                Mientras ( (V[j] > temp) Y (j > 1))
                        V[j+1] = V[j]
                        j = j-1
                FinMientras
                V[j+1] = temp
        FinPara
Fin_accion
```

Análisis del algoritmo ordenamiento por inserción.

Requerimiento de memoria: Una variable adicional para realizar los intercambios.
Tiempo de ejecución: Para una lista de **n** elementos el ciclo externo se ejecuta **n-1** veces. El ciclo interno se ejecuta como máximo una vez en la primera iteración, 2 veces en la segunda, 3 veces en la tercera, etc. Esto produce una complejidad $O(n^2)$.

Ventajas:

Fácil implementación.
Requerimientos mínimos de memoria.

Desventajas:

Lento.
Realiza numerosas comparaciones.

Este también es un algoritmo lento, pero puede ser de utilidad para listas que están ordenadas o semiordenadas, porque en ese caso realiza muy pocos desplazamientos.

```
void insertionSort (double v[])
{
  double tmp;
  int i, j;
  int N = v.length;

  for (i=1; i<N; i++) {

      tmp = v[i];

      for (j=i; (j>0) && (tmp<v[j-1]); j--)
          v[j] = v[j-1];

      v[j] = tmp;
  }
}
```

Método de ordenación rápida (quicksort)

Hoare, construyo un método más eficiente que los métodos de burbuja y shell.

El método se basa en la estrategia típica de “divide y vencerás” (divide and conquer).

Este método consiste en dividir la lista a ordenar en dos sublistas, ordenando cada una de ellas por separado.

Se denomina método de ordenación rápida porque, en general, puede ordenar una lista de datos, mucho más rápidamente que cualquiera de los métodos de ordenación ya estudiados.. La lista de clasificar almacena un vector o array se divide (parte) en dos sublistas: una con todos con todos los valores menores o iguales a un cierto valor específico y otra con todos los valores mayores que ese valor. El valor elegido puede ser cualquier valor arbitrario del vector. En ordenación rápida, se llama a este valor pivote.

```
void quicksort (double v[], int izda, int dcha)
{
   int pivote;   // Posición del pivote

   if (izda<dcha) {

      pivote = partir (v, izda, dcha);

      quicksort (v, izda, pivote-1);

      quicksort (v, pivote+1, dcha);
   }
}
```

```
quicksort (vector, 0, vector.length-1);
```

```
int partir (double v[],
       int primero, int ultimo)
{
  // Valor del pivote
  double pivote = v[primero];

  // Variable auxiliar
  double temporal;

  int izda = primero+1;
  int dcha = ultimo;

  do {  // Pivotear…

    while ((izda<=dcha)
          && (v[izda]<=pivote))
          izda++;
    while ((izda<=dcha)
          && (v[dcha]>pivote))
          dcha--;

    if (izda < dcha) {
       temporal = v[izda];
       v[izda] = v[dcha];
       v[dcha] = temporal;
       dcha--;
       izda++;
    }

  } while (izda <= dcha);

  // Colocar el pivote en su sitio
  temporal = v[primero];
  v[primero] = v[dcha];
  v[dcha] = temporal;

  // Posición del pivote
  return dcha;
}
```

Evaluacion QuickSort

En el peor caso

$T(n) = T(1) + T(n-1) + c \cdot n \in O(n^2)$

Tamaño de los casos equilibrado

(idealmente, usando la mediana como pivote)

$T(n) = 2T(n/2) + c \cdot n \in O(n \log n)$

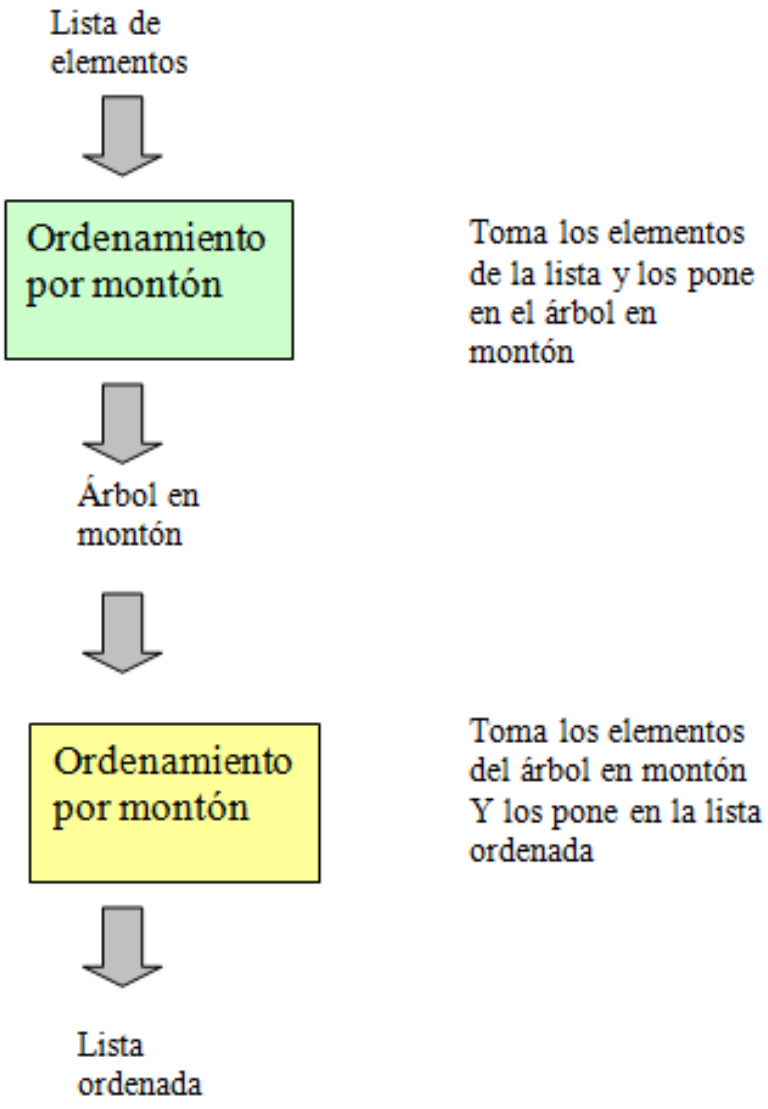
Promedio

$$T(n) = \frac{1}{n}\sum_{i=1}^{n-1}\left[T(n-i)+T(i)\right]+cn = \frac{2}{n}\sum_{i=1}^{n-1}T(i)+cn \in O(n\log n)$$

ÁRBOL DE MONTON

Una técnica alternativa a las convencionales consiste en hacer uso de un árbol de montón. Un árbol de montón se caracteriza porque siempre tiene en la raíz el mayor valor. Por tanto si se quiere ordenar una secuencia de elementos (archivo, lista etc.), primero se insertan estos elementos en un árbol de montón, luego, se elimina la raíz y se coloca su valor en la salida (secuencia ordenada).

El último elemento del árbol (nodo de la derecha del último nivel) se coloca en la raíz. Después de esto posiblemente el árbol no sea de montón por lo que tendrá que llamarse a una rutina que haga descender el valor adicionado hasta que el árbol sea de montón. Nuevamente se elimina la raíz y se adiciona a la salida. Este procedimiento se repite hasta que el árbol queda vació.


Lista de elementos
Ordenamiento por montón
Toma los elementos de la lista y los pone en el árbol en montón
Árbol en montón
Ordenamiento por montón
Toma los elementos del árbol en montón Y los pone en la lista ordenada
Lista ordenada

ORDENAMIENTO EN MONTÓN

Sea M un árbol en montón de dimensión N, lo que quiere decir que el nodo raíz corresponde al mayor valor del árbol. Si extraemos la raíz y lo colocamos en una lista, y reemplazamos la raíz por el último nodo, es posible que el árbol resultante no sea de montón, en este caso tendremos que convertirlo en montón. Una vez que se consiga un árbol de montón, nuevamente eliminamos la raíz y lo adicionamos en la lista, y así proseguimos hasta que el árbol este vacío, y todos sus elementos se hayan pasado a la lista de en orden descendente.

Entrada : árbol en montón

Salida : lista de elementos en orden descendente

Procedimiento :

Mientras el árbol en montón no sea vacío, repetir los pasos:

Eliminar la raíz, y colocar el elemento en la lista.
Colocar como raíz el último nodo del árbol. Si el árbol que resulta no es de montón, entonces se procederá a convertirlo en montón, luego de conseguirlo, se continuara en el paso 1.

Evaluación Ordenamiento Árbol en Montón (AM)

Construcción inicial del AM:

n operaciones de inserción O(log n)
sobre un árbol parcialmente ordenado
O(n log n)

Extracción del menor/mayor elemento del AM

n operaciones de borrado O(log n)
sobre un árbol parcialmente ordenado
O(n log n)

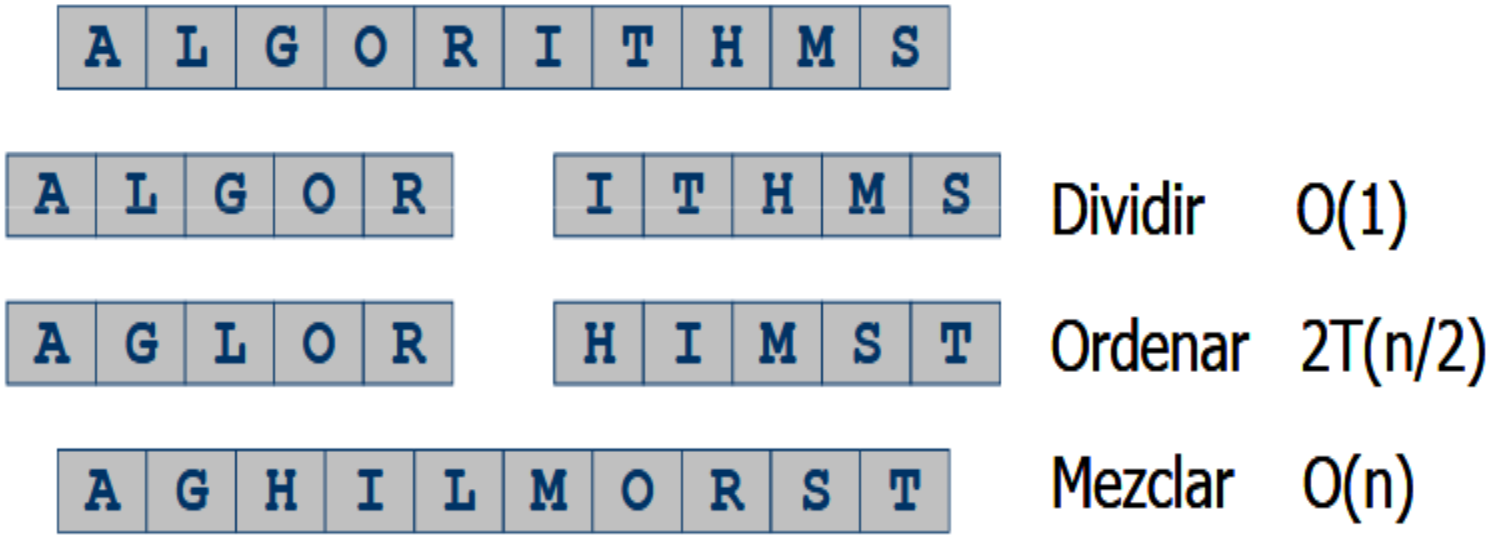
ORDENAMIENTO EN MERGE SORT

Algoritmo de ordenación “divide y vencerás”:

Dividir nuestro conjunto en dos mitades.

Ordenar recursivamente cada mitad.

Combinar las dos mitades ordenadas: O(n).


A L G O R I T H M S
A L G O R
I T H M S
A G L O R
H I M S T
A G H I L M O R S T
Dividir O(1)
Ordenar 2T(n/2)
Mezclar O(n)

```
Accion Merge_Sort(S,i,j)
Inicio
Si i = j  retornar
          M=(i+j)/2
          Merge_Sort(S, i, m)
          Merge_Sort(S , m+1, j)
          Merge(S, i , m, j, C)
          Para k dese i hasta j
                    S_k = C_k
          FinPara
Fin
```

Tenemos que $T(n) = 2T(n/2) + n$ (a)

Sustituimos n por n/2

$2T(n/2) = 2\,(2T(n/4) + n) = 4\,T(n/4) + n$

Remplazamos en (a) $T(n) = 4T(n/4) + 2n$ (b)

Sustituimos n por n/4

$4T(n/4) = 4\,(2T(n/8) + n) = 8\,T(n/8) + n$

Remplazamos en (b)

$T(n) = 8T(n/8) + 3n$

Sucesivamente, tenemos que

$T(n) = 2^k T(n/2^k) + kn$ (c)

Hacemos $k = \log n$ luego $n = 2^k$

En (c)

$T(n) = nT(1) + n \log n = n + n\log n$

Por tanto el algoritmo es de orden $O(n\log n)$

Técnicas de búsqueda

La búsqueda se refiere a la operación de encontrar la posición de un valor dado entre una secuencia de valores.

Los algoritmos de búsqueda al igual que los de ordenamiento son muy utilizados en las aplicaciones informáticas. Las técnicas para buscar un elementos pueden variar dependiendo de la precondición que este ordenado o no el vector. Si no esta ordenado, se tiene que explorar necesariamente todos los elementos del vector. Si esta ordenado solo se explorara hasta que se encuentre el valor buscado, en cuyo caso termina el algoritmo, o hasta que se encuentre un valor mayor al buscado, en este caso como no es posible que el valor buscado se encuentre mas adelante, se terminara la exploración y se responderá que el valor no se encontró. En ambos casos se puede utilizar un algoritmo iterativo o una solución recursiva.

Búsqueda lineal

Una búsqueda lineal es aquella que, para encontrar un elemento que se encuentra en la posición de una lista, debe recorrer todas las posiciones anteriores.

Por tanto, su rendimiento será de **O(n)** en el peor caso, siendo n el tamaño de la estructura. Es decir, el tiempo de búsqueda crece linealmente con el número de elementos.

Por ello, para un numero de elementos enorme (cientos de miles, millones…) es muy lento. Debido a esto, se debería usar solo para colecciones de datos pequeñas y NO ordenadas.

BUSQUEDA SECUENCIAL

```
Accion BUSSEC(A, N, V)
Inicio
        Encontro = falso
        i=1
        Mientras ( i ≤ N)
                Si (A[i] = V)
                        Encontro = verdad
                Sino
                        i= i+1
                FinSi
        FinMientras
        Retornar i
Fin
```

Se quiere buscar un valor V en el arreglo A. Podemos hacer una búsqueda secuencial. Una aproximación evidente consiste en examinar secuencialmente todos los elementos de A hasta que o bien lleguemos al final del vector, o bien encontremos un elemento que sea menor que V. Nótese que cuando se encuentra el valor buscado, el algoritmo sigue explorando el vector innecesariamente. El ciclo debería detenerse cuando se encuentra el valor

```
Accion BUSSEC(A, N, V) // mejorada
Inicio
        Encontro = falso
        i=1

        Mientras ( i ≤ N  ∧ ¬ encontro )
                Si (A[i] = V)
                        Encontro = verdad
                Sino

                        i= i+1

                FinSi
        FinMientras
        Retornar i
Fin
```

```
Accion Buscar()
Inicio
        Leer V
        i = BusSecRecur (A, 1 , N, V)
        Si i= N+1
                Escribir "No existe  valor"
        Sino
                Escribir "Si existe  valor"
        FinSi
Fin
```

```
Accion BusSecRecur(A, i, N, V) // mejorada
Inicio
        Si ( i ≤ N )
                Si(A[i ] = V )
                        Retornar i
                Sino
                        Retornar BusSecRecur
        (A, i+1, N, V)
                        // llamada recursiva
                FinSi
        Sino

                Retornar i

        FinSi

Fin
```

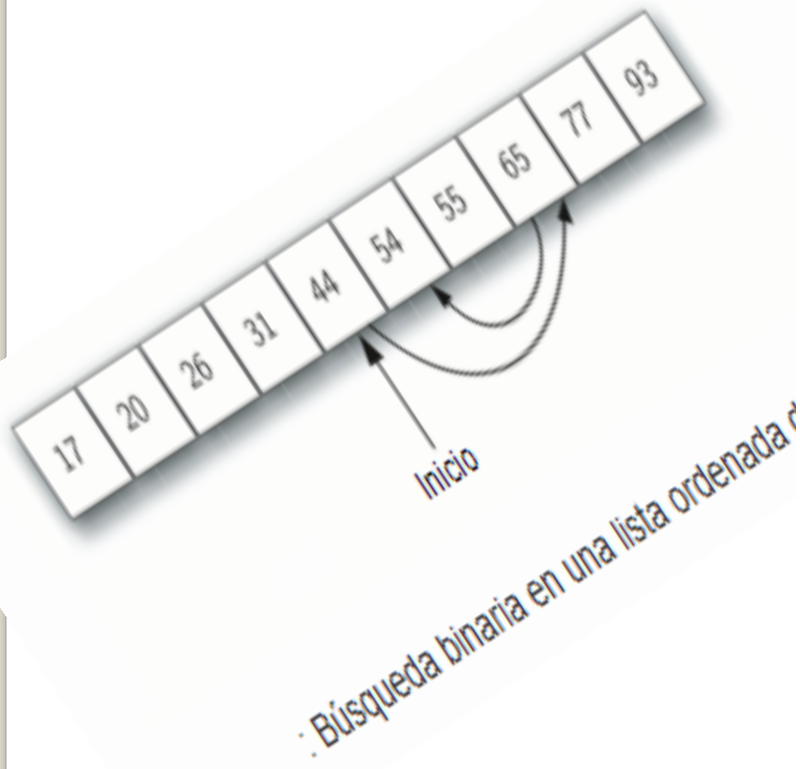
Búsqueda binaria

Una búsqueda binaria es aquella que aprovecha un orden en la estructura de datos para encontrar un elemento de forma dicotomica.

Lo que hace este algoritmo es comparar con el valor que esta en el medio de la lista. Si el elemento que esta en el medio es menor al que se busca, se repite el proceso con la mitad inferior (0, mitad-1) o si es mayor pues con la mitad superior (mitad+1, n), siendo n el tamaño de la lista.

El bucle se repite hasta que se encuentra o hasta que no se pueda dividir mas.

las búsquedas binarias tienen un rendimiento de **O(log(n))** en el peor caso, lo que significa que el tiempo de búsqueda crece muy lentamente respecto al tamaño de la lista.




```
def busquedaBinaria(unaLista, item):
    primero = 0
    ultimo = len(unaLista)-1
    encontrado = False

    while primero<=ultimo and not encontrado:
        puntoMedio = (primero + ultimo)//2
        if unaLista[puntoMedio] == item:
            encontrado = True
        else:
            if item < unaLista[puntoMedio]:
                ultimo = puntoMedio-1
            else:
                primero = puntoMedio+1

    return encontrado
```

```
def busquedaBinaria(unaLista, item):
    if len(unaLista) == 0:
        return False
    else:
        puntoMedio = len(unaLista)//2
        if unaLista[puntoMedio]==item:
          return True
        else:
          if item<unaLista[puntoMedio]:
            return busquedaBinaria(unaLista[:puntoMedio],item)
          else:
            return busquedaBinaria(unaLista[puntoMedio+1:],item)

listaPrueba = [0, 1, 2, 8, 13, 17, 19, 32, 42,]
print(busquedaBinaria(listaPrueba, 3))
print(busquedaBinaria(listaPrueba, 13))
```

Cuando dividimos la lista suficientes veces, terminamos con una lista que tiene un único ítem. Ya sea aquél ítem único el valor que estamos buscando o no lo sea.

En todo caso, habremos terminado. El número de comparaciones necesarias para llegar a este punto es *i*

donde $\boldsymbol{n / 2^i = 1.}$

La solución para *i* nos da $i = \log n$.

El número máximo de comparaciones es logarítmico con respecto al número de ítems de la lista. Por lo tanto, la búsqueda binaria es $O(\log n)$.

Es necesario enfrentar una cuestión de análisis adicional. En la solución recursiva mostrada anteriormente, la llamada recursiva,

Ejercicios propuestos

1. **Construya una aplicación que muestre un menú de técnicas de ordenamiento.**

2. **Desarrolle cada una de los técnicas de ordenamiento**

3. **Evalué cada uno de los algoritmos de ordenamiento**

….Ejercicios propuestos

3 Construya una aplicación que muestre un menú de técnicas de búsqueda.

4 Desarrolle cada una de los técnicas de búsqueda

5 Evalué cada una de las técnicas de búsqueda