Ordenamientos

Uninorte

Temática

- Complejidad
  - Clasificación
  - Estructuras de datos
- Algoritmos de búsqueda
- Algoritmos de ordenamiento

Complejidad: algorítmica

- La complejidad algorítmica es una métrica teórica que se aplica a los algoritmos.
- Esta métrica analiza el comportamiento de los algoritmos con base en el número de instrucciones y la cantidad de datos tratados.
- Representa la cantidad de recursos temporales que necesita el algoritmo para resolverse, y permite determinar su eficiencia.
- En la práctica usualmente se tiene en cuenta: el lenguaje de programación usado, el compilador, hardware y/o software, máquina, entre otras cosas.
- Mejor y peor de los casos suelen ser los más usados.

http://www.lcc.uma.es/~av/Libro/CAP1.pdf

Complejidad: Clasificación

- Lugar:
  - Memoria interna
  - Memoria externa
- Tiempo
- Complejidad

Complejidad: Estructuras de datos

Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	**Θ(1)**	**Θ(n)**	**Θ(n)**	**Θ(n)**	**O(1)**	**O(n)**	**O(n)**	**O(n)**	**O(n)**
Stack	**Θ(n)**	**Θ(n)**	**Θ(1)**	**Θ(1)**	**O(n)**	**O(n)**	**O(1)**	**O(1)**	**O(n)**
Queue	**Θ(n)**	**Θ(n)**	**Θ(1)**	**Θ(1)**	**O(n)**	**O(n)**	**O(1)**	**O(1)**	**O(n)**
Singly-Linked List	**Θ(n)**	**Θ(n)**	**Θ(1)**	**Θ(1)**	**O(n)**	**O(n)**	**O(1)**	**O(1)**	**O(n)**
Doubly-Linked List	**Θ(n)**	**Θ(n)**	**Θ(1)**	**Θ(1)**	**O(n)**	**O(n)**	**O(1)**	**O(1)**	**O(n)**
Skip List	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n log(n))
Hash Table	**N/A**	**Θ(1)**	**Θ(1)**	**Θ(1)**	**N/A**	**O(n)**	**O(n)**	**O(n)**	**O(n)**
Binary Search Tree	**Θ(log(n))**	**Θ(log(n))**	**Θ(log(n))**	**Θ(log(n))**	**O(n)**	**O(n)**	**O(n)**	**O(n)**	**O(n)**
Cartesian Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(n)	O(n)	O(n)	O(n)
B-Tree	**Θ(log(n))**	**Θ(log(n))**	**Θ(log(n))**	**Θ(log(n))**	**O(log(n))**	**O(log(n))**	**O(log(n))**	**O(log(n))**	**O(n)**
Red-Black Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Splay Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(log(n))	O(log(n))	O(log(n))	O(n)
AVL Tree	**Θ(log(n))**	**Θ(log(n))**	**Θ(log(n))**	**Θ(log(n))**	**O(log(n))**	**O(log(n))**	**O(log(n))**	**O(log(n))**	**O(n)**
KD Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)

http://bigocheatsheet.com/

Algoritmos de búsqueda

- Búsqueda lineal o secuencial O(n)
- Búsqueda binaria O(log(n))
- Otras búsquedas
  - Búsqueda ternaria, Fibonacci, Exponencial

Búsqueda secuencial

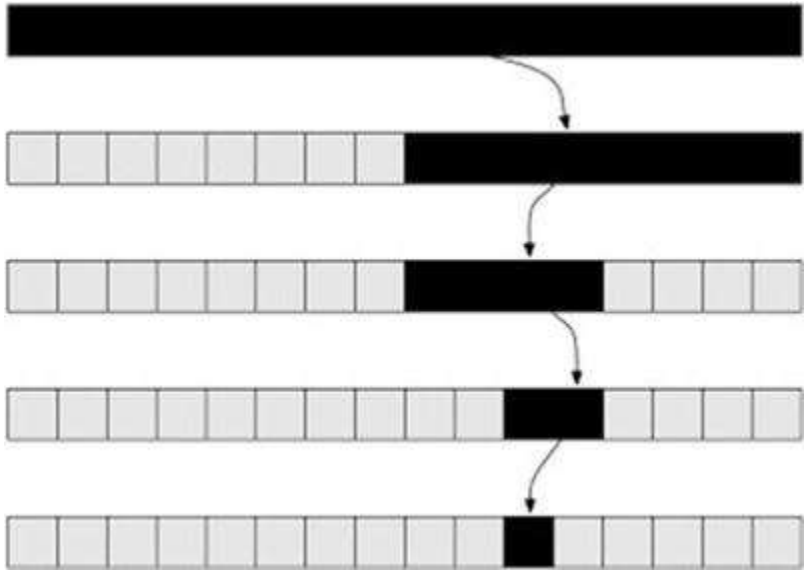
```
Inicio
Entero vector[100], n, dato, i
para i = 1, i<=n, i<- i + 1
        Si (dato = vector[i]) entonces
                Escribir “El dato esta en el vector en la posicion ”+i
finpara
Fin
```

Búsqueda binaria

- La búsqueda binaria es un algoritmo eficiente para encontrar un elemento en una lista ordenada de elementos
- Funciona al dividir repetidamente a la mitad la porción de la lista que podría contener al elemento, hasta reducir las ubicaciones posibles a solo una.

```
Entero n,dato,V(),centro, inf, sup
Logico sw=falso
Lea n
leerVector(V,n)
ordenarVector(V,n)
Lea dato
inf <- 1
sup <- n
HH
centro <- (sup + inf) / 2
  Si V(centro) = dato  entonces
     sw<-verdadero
  Sino
     Si dato < V(centro)
           sup <- centro - 1
     Sino
           inf <- centro + 1
     finSi
  finSi
FinHH (inf > sup o sw=verdadero)
```

Búsqueda binaria



Algoritmos de ordenamiento

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Quicksort	**Ω(n log(n))**	**Θ(n log(n))**	**O(n^2)**	**O(log(n))**
Mergesort	**Ω(n log(n))**	**Θ(n log(n))**	**O(n log(n))**	**O(n)**
Timsort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)
Heapsort	**Ω(n log(n))**	**Θ(n log(n))**	**O(n log(n))**	**O(1)**
Bubble Sort	**Ω(n)**	**Θ(n^2)**	**O(n^2)**	**O(1)**
Insertion Sort	**Ω(n)**	**Θ(n^2)**	**O(n^2)**	**O(1)**
Selection Sort	**Ω(n^2)**	**Θ(n^2)**	**O(n^2)**	**O(1)**

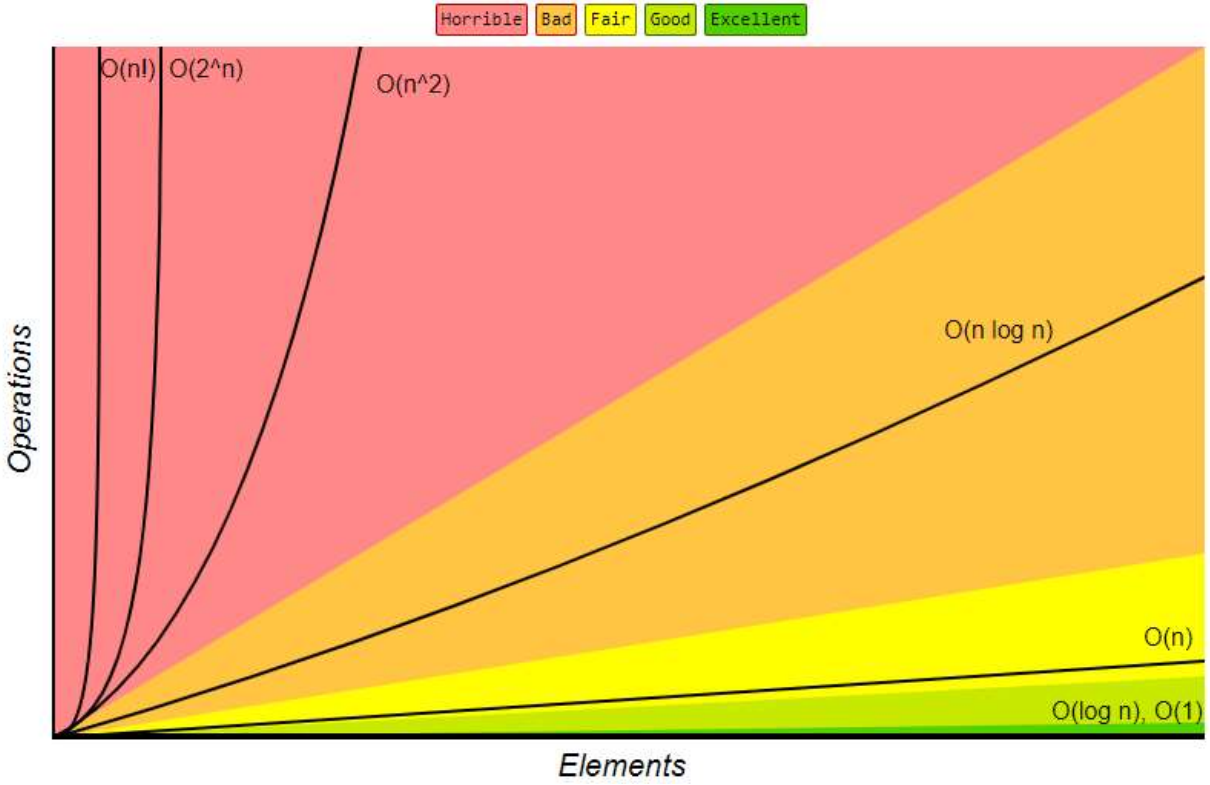
http://bigocheatsheet.com/

Algoritmos de ordenamiento

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Tree Sort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(n)
Shell Sort	**Ω(n log(n))**	**Θ(n(log(n))^2)**	**O(n(log(n))^2)**	**O(1)**
Bucket Sort	Ω(n+k)	Θ(n+k)	O(n^2)	O(n)
Radix Sort	**Ω(nk)**	**Θ(nk)**	**O(nk)**	**O(n+k)**
Counting Sort	Ω(n+k)	Θ(n+k)	O(n+k)	O(k)
Cubesort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)

http://bigocheatsheet.com/

Relación entre operaciones y elementos



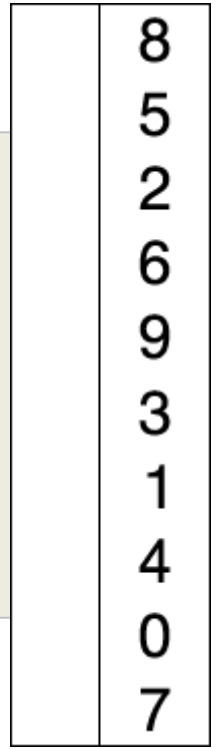
Bubble sort

6 5 3 1 8 7 2 4

Select sort

El método select sort consiste en encontrar el menor de todos los elementos del arreglo e intercambiarlo con el que está en la primera posición. Luego el segundo más pequeño, y así sucesivamente hasta que todo el arreglo este ordenado.

```
Entero V(),n,i,j,tmp, menor
Para (i <- 1,n)
    menor <- i
    Para (j <- i+1,n)
      Si (a[j] < a[menor]) ent
           menor = j
      finSi
    finPara
    Si (menor != i) ent
       intercambia(V,i,menor)
    finSi
finPara
```



Insert sort

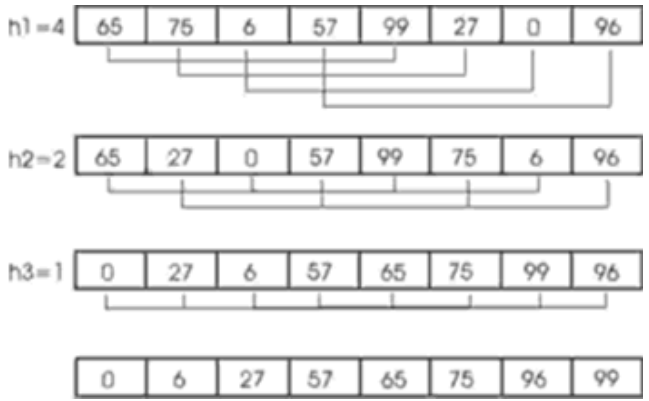
Consta de tomar uno por uno los elementos de un arreglo y recorrerlo hacia su posición con respecto a los anteriormente ordenados. Así empieza con el segundo elemento y lo ordena con respecto al primero. Luego sigue con el tercero y lo coloca en su posición ordenada con respecto a los dos anteriores, así sucesivamente hasta recorrer todas las posiciones del arreglo.

```
Entero V[],n, tmp, i, j
Para (i <- 1, n)
   tmp <- V[i]
   j <- i
   MQ (j>1 y tmp < V[j-1]) haga
       V[j] <- V[j-1]
       j <- j-1
   finMQ
   V[j] <- tmp
finPara
```

Shell sort

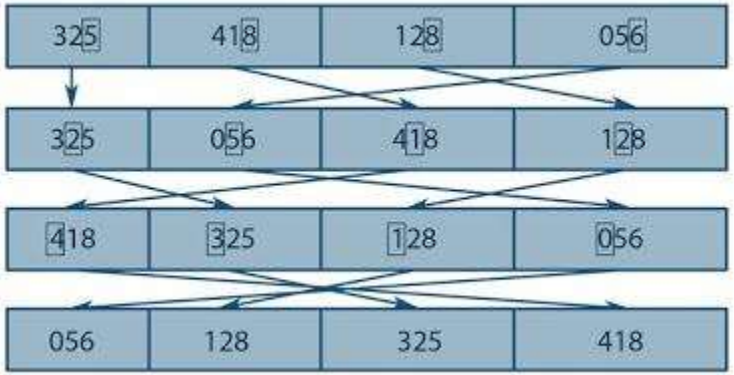
- Este algoritmo mejora Insert sort comparando elementos separados por un espacio de varias posiciones, no sólo una posición.
- Esto permite que un elemento haga “bigger steps” hacia su posición esperada.
- El último paso del Shell sort es un Insert sort, pero los datos del vector están casi ordenados.

```
Entero V[],n, tmp, i, j, inc
inc <- n div 2
MQ(inc >0) haga
    Para(i = inc+1, n)
       j <- i - inc
       MQ(j>0) haga
           k <- j + inc
           Si(V[j]>V[k])
              intercambia(V,j,k)
           Sino
              j <- 0
           finSi
           j <- j - inc;
       fin MQ
    finPara
    inc <- inc div 2;
fin MQ
```



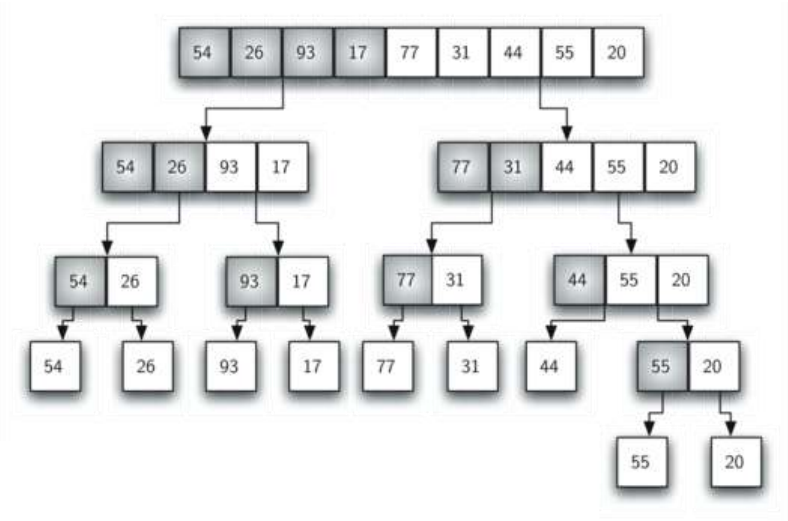
Radix sort

- Primero se va ordenando tomando en consideración el número menos significativo (la unidad) del más pequeño al más grande.
- Luego, a partir de la lista que obtuvimos en el paso anterior, ordenamos los numero de menor a mayor considerando la decena de cada uno de ellos.
- Finalmente comprobamos que la lista fue ordenada satisfactoriamente mediante este procedimiento.



Merge sort

- Dividir la lista al medio, formando dos sublistas de (aproximadamente) el mismo tamaño cada una.
- Ordenar cada una de esas dos sublistas (usando este mismo método)
- Una vez que se ordenaron ambas sublistas, intercalarlas de manera ordenada.



6 5 3 1 8 7 2 4

Heap sort

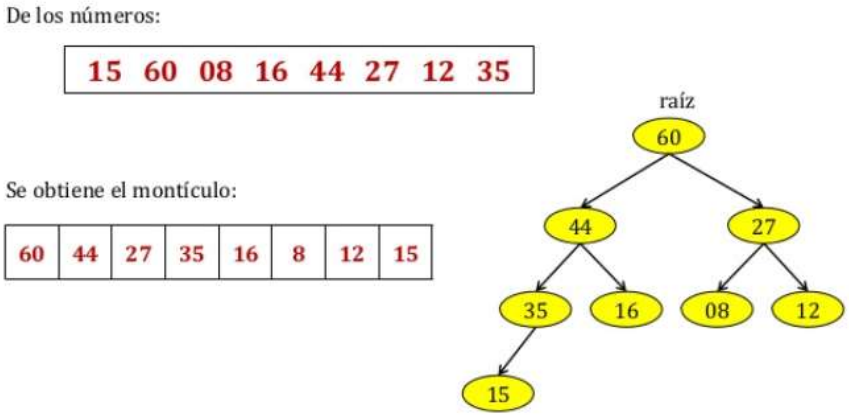
Este algoritmo consiste en almacenar todos los elementos del vector a ordenar en un montículo (heap), luego extraer el nodo que queda como nodo raíz del montículo (cima) en sucesivas iteraciones obteniendo el conjunto ordenado.

Monticulo maximo: Cumple la condición de que todo padre es mayor que sus hijos.

Deben estar balanceados. No es lo mismo que un ABB.

EL padre de cada K es K/2

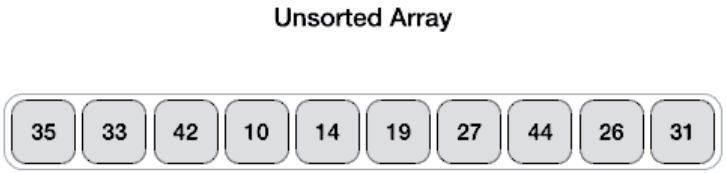
Para cada K, el hijo izquierdo es 2*K y el derecho es 2*K+1



Quicksort

- Se basa en la técnica de “Divide y vencerás” por la que en cada recursión, el problema se divide en subproblemas de menor tamaño y se resuelven por separado (aplicando la misma técnica) para ser unidos de nuevo una vez resueltos.
- Pasos
  - Se elige un elemento *v* de la lista **L** de elementos al que se le llama ***pivote***
  - Se particiona la lista **L** en 2 listas:
    - $\mathbf{L_1}$ - que contiene todos los elementos de **L** menos *v* que sean menores o iguales que *v*
    - $\mathbf{L_2}$ - que contiene todos los elementos de **L** menos *v* que sean mayores o iguales que *v*
  - Se aplica la recursión sobre $\mathbf{L_1}$ y $\mathbf{L_2}$
  - Se unen todas las soluciones que darán forma final a las listas **L** finalmente ordenada.

Quicksort



Referencias

- A. Mancilla, Ebratt, Capacho. Diseño y construcción de algoritmos. Universidad del Norte, 2014. ISBNe 9789587414974

- Joyanes Aguilar, Luis, Castillo Sanz, Andrés, Sánchez García, Lucas. Algoritmos, programación y estructuras de datos. McGraw-Hil, 2005.