

“UNIVERSIDAD NACIONAL SAN CRISTÓBAL DE HUAMANGA”

FACULTAD DE INGENIERÍA DE MINAS, GEOLOGÍA Y CIVIL

ESCUELA DE FORMACIÓN PROFESIONAL DE INGENIERÍA DE SISTEMAS



LABORATORIO DE ESTRUCTURA DE DATOS

“ALGORITMOS DE ORDENAMIENTO Y BÚSQUEDA”

Docente : TERRAZA HUAMAN, Edem Jersson

Estudiantes :

- QUISPE CCAHUANA, Maria Leonela
- SARMIENTO MEDINA, Mijael
- TELLO SAICO, Marcos Jamil
- VALDIVIA CALLE, Roy Yefferson
- VILA CAYO, Nayherly Dianeth

Ayacucho - Perú

2022

ÍNDICE

ALGORITMOS DE BÚSQUEDA:	3
1. Búsqueda Secuencial:	3
2. Búsqueda Binaria:	5
3. Búsqueda Hashing	7
ALGORITMOS DE ORDENAMIENTO:	10
1. Ordenamiento de Burbuja	10
2. Ordenamiento por Inserción	11
3. Ordenamiento por selección	12
4. Ordenamiento Shell	13
5. Ordenamiento binario	14
6. Ordenamiento rápido (Quicksort)	18
7. Ordenamiento por montículos (Heapsort)	21
8. Ordenamiento Radix	24

ALGORITMOS DE BÚSQUEDA:

Un algoritmo de búsqueda es aquel que está diseñado para localizar un elemento concreto dentro de una estructura de datos para entregar una respuesta exacta.

Consiste en solucionar un problema de existencia o no de un elemento determinado en un conjunto finito de elementos, es decir, si el elemento en cuestión pertenece o no a dicho conjunto, además de su localización dentro de éste. Surgen de la necesidad de conocer tanto si un dato se encuentra o no dentro de una colección como de la posición que ocupa.

Con el objetivo de localizar un elemento específico dentro de una estructura de datos. Con el fin de que el programa sea más eficiente de la forma siguiente: Dada una lista de números, tenemos que verificar si un número “n” está dentro de esta lista; además, guardaremos el número de pasos que nos tomó obtener el resultado.

Existen diferentes algoritmos de búsqueda y la elección depende de la forma en que se encuentren organizados los datos: si se encuentran ordenados o si se ignora su disposición o se sabe que están al azar. También depende de si los datos a ordenar pueden ser accedidos de modo aleatorio o deben ser accedidos de modo secuencial.

1. Búsqueda Secuencial:

Es algoritmo búsqueda lineal o la búsqueda secuencial es un método para encontrar un valor objetivo dentro de una lista de búsqueda más simple, menos eficiente y que menos precondiciones requiere: no requiere conocimientos sobre el conjunto de búsqueda ni acceso aleatorio. Consiste en comparar cada elemento del conjunto de búsqueda con el valor deseado hasta que éste sea encontrado o hasta que se termine de leer el conjunto.

Supondremos que los datos están almacenados en un array y se asumirá acceso secuencial.

Se utiliza cuando el contenido del Vector no se encuentra o no puede ser ordenado. Consiste en buscar el elemento comparándolo secuencialmente (de ahí su nombre) con cada elemento del arreglo o conjunto de datos hasta que se encuentre, o hasta que se llegue al final del arreglo. La existencia se puede asegurar desde el momento que el elemento es localizado, pero no podemos asegurar la no existencia hasta no haber analizado todos los elementos del arreglo.

Este es el método de búsqueda más lento, pero si nuestra información se encuentra completamente desordenada es el único que nos podrá ayudar a encontrar el dato que busquemos.

```
1
2 def busqueda_lineal(array, n, x):
3
4     for i in range(0, len(array)): #recorre el array de inicio a fin
5         if (array[i] == x): #condicional de comparacion
6             return i
7     return -1
8
9 array = [10, 8, 3, 1, 2, 7, 0]
10 print("\narray= ",array,"\n")
11 n = len(array)
12 x = int(input("Ingrese el elemento a buscar en el array: "))
13
14 index = busqueda_lineal(array, len(array), x)
15 if(index == -1):
16     print("Elemento no encontrado !!!")
17 else:
18     print("El elemento << {} >> se encuentra en el indice: {}".format(x,index))
19
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

PS D:\LABORATORIO_ESTRUCTURA_DATOS> & C:/Users/USUARIO/AppData/Local/Programs/Python/Python37/python.exe c

array= [10, 8, 3, 1, 2, 7, 0]

Ingrese el elemento a buscar en el array: 7
El elemento << 7 >> se encuentra en el indice: 5

Complejidad.

- **MEJOR CASO:** El algoritmo de búsqueda lineal termina tan pronto como encuentra el elemento buscado en el array. Si tenemos suerte, puede ser que la primera posición examinada contenga el elemento que buscamos, en cuyo caso el algoritmo informará que tuvo éxito después de una sola comparación. Por tanto, la complejidad en este caso será **$O(1)$** .
- **PEOR CASO:** Sucede cuando encontramos X en la última posición del array. Como se requieren **n** ejecuciones del bucle mientras, la cantidad de tiempo es proporcional a la longitud del array **n**, más un cierto tiempo para realizar las instrucciones del bucle mientras y para la llamada al método. Por lo tanto, la cantidad de tiempo es de la forma **$an + b$ (instrucciones del mientras * tamaño del arreglo + llamada al método)** para ciertas constantes a y b, que representan el coste del bucle mientras y el costo de llamar el método respectivamente. Representando esto en notación O, **$O(an+b) = O(n)$** .
- **CASO MEDIO:** Supongamos que cada elemento almacenado en el array es igualmente probable de ser buscado. La media puede calcularse tomando el tiempo total de encontrar todos los elementos y dividiéndolo por **n**:
Total = $a(1 + 2 + \dots + n) + bn = a(n(n+1) / 2) + bn$, a representa el costo constante asociado a la ejecución del ciclo y b el costo constante asociado a la evaluación de la condición. 1, 2, ..., n, representan el costo de encontrar el elemento en la primera, segunda, ..., enésima posición dentro del arreglo.

Media = (Total / n) = $a((n+1) / 2) + b$ que es $O(n)$.

Ventajas de la técnica.

Es el algoritmo más simple de búsqueda y no requiere ningún proceso previo de la tabla, ni ningún conocimiento sobre la distribución de las llaves. La búsqueda secuencial es el área del problema donde previamente existían mejores algoritmos.

Desventajas de la técnica.

Este método de búsqueda es muy lento, pero si los datos no están en orden es el único método que puede emplearse para hacer las búsquedas. Si los valores de la llave no son únicos, para encontrar todos los registros con una llave particular, se requiere buscar en toda la lista.

Si los registros a los que se accede con frecuencia no están al principio del archivo, la cantidad promedio de comparaciones aumenta notablemente dado que se requiere más tiempo para recuperar dichos registros.

Para las aplicaciones interactivas que incluyen peticiones o actualizaciones de registros individuales, los archivos secuenciales ofrecen un rendimiento pobre.

Definitivamente, la búsqueda secuencial es el método menos eficiente; porque se basa en comparar el valor que se desea buscar con cada uno de los valores del archivo.

2. Búsqueda Binaria:

En este caso, este tipo de búsqueda es usado en listas que estén previamente ordenadas, ya que su método de búsqueda es la de dividir los datos en dos grupos, eligiendo el grupo en el cual debería estar el dato buscado (supone que está ordenado alfabéticamente o numéricamente), volviendo a aplicar la división, y así sucesivamente hasta verificar si existe o no existe el dato buscado.

Divide el array en mitades, es por eso el nombre “binaria”, para que la búsqueda sea más sencilla.

La búsqueda binaria en Python es un algoritmo de búsqueda que encuentra la posición de un valor en un array ordenado. Compara el valor con el elemento en el medio del array, si no son iguales, la mitad en la cual el valor no puede estar es eliminada y la búsqueda continúa en la mitad restante hasta que el valor se encuentre.

La Búsqueda Binaria o Búsqueda de Medio Intervalo es un algoritmo utilizado para localizar un valor específico. Examina el valor con el elemento en el medio del arreglo, si no son iguales, la mitad en la cual el valor no puede estar es eliminada y la búsqueda continúa en la mitad restante hasta que el valor se encuentre.

Ventajas:

- Se puede aplicar tanto a datos en listas lineales como en árboles binarios de búsqueda.
- Es el método más eficiente para encontrar elementos en un arreglo ordenado.

Desventaja:

- Este método funciona solamente con arreglos ordenados, por lo cual si nos encontramos con arreglos que no están en orden, este método, no nos ayudaría en nada.

Algoritmo:

1. Verifica que la lista tenga elementos. Si no los hay termina.
2. Compara el elemento a buscar con el número que se encuentra a la mitad de la lista.
3. Si es el que se busca regresa el número y termina.
4. Si no es el número buscado:
 - a. Si el número es menor al elemento de la mitad de la lista realiza la búsqueda binaria con la sublista izquierda.
 - b. Si no realiza la búsqueda binaria con la sublista derecha.

Ejemplo:

```
def binaria(lista, x):  
    if len(lista) <= 0:  
        return "Número no encontrado"  
  
    m = lista[len(lista)//2]  
    if m == x:  
        return "Número encontrado"  
    else:  
        if x < m:  
            return binaria(lista[:len(lista)//2], x)  
        else:  
            return binaria(lista[(len(lista)//2)+1:], x)  
  
lista = [-5, -1, 2, 4, 5, 6, 10, 27]  
print(binaria(lista, 4))
```

➞ Número encontrado

3. Búsqueda Hashing

La función hash, también conocido como hashing o transformación de llaves, es un método que permite el acceso a estos datos sin que los mismos estén ordenados, la cual aumenta la velocidad de búsqueda reduciendo el tiempo de espera

significativamente.

Es una función criptográfica, que se caracteriza porque a partir de un dato de entrada de cualquier extensión, obtiene una salida finita que normalmente es una cadena de longitud fija. A esta salida se le conoce como “resumen” de la información de entrada y su valor siempre va a ser el mismo para la misma entrada.

Se trata de una estructura de datos que asocia claves con valores. Este tipo de estructura permite realizar búsquedas de orden constante.

Funcionan al transformar un valor o contenido con una función hash. Así se puede obtener un número que identifica la posición en la que se colocará el valor.

Función hash:

$$h(k) = k \% m$$

donde:

- k: elemento del que obtendremos su valor hash.
- m: un número igual al tamaño de la tabla hash.

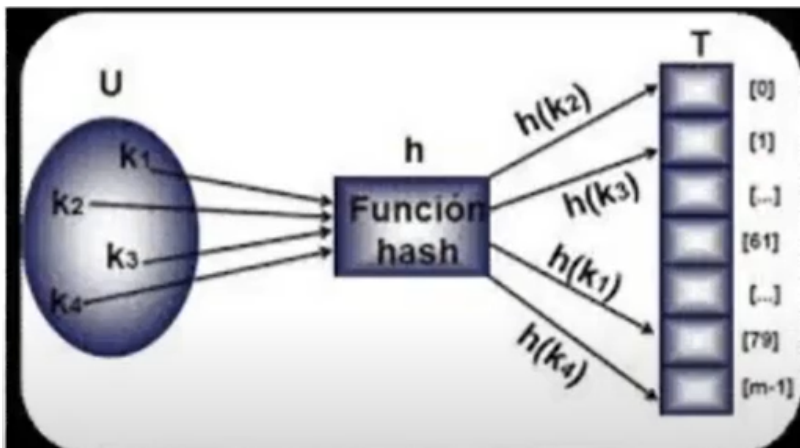


Tabla hash: Es una estructura que asocia claves con valores para que después pase los datos de entrada que se quiere incluir en dicha tabla, por una función hash que nos va a devolver valores entre 0 y el tamaño de la tabla.

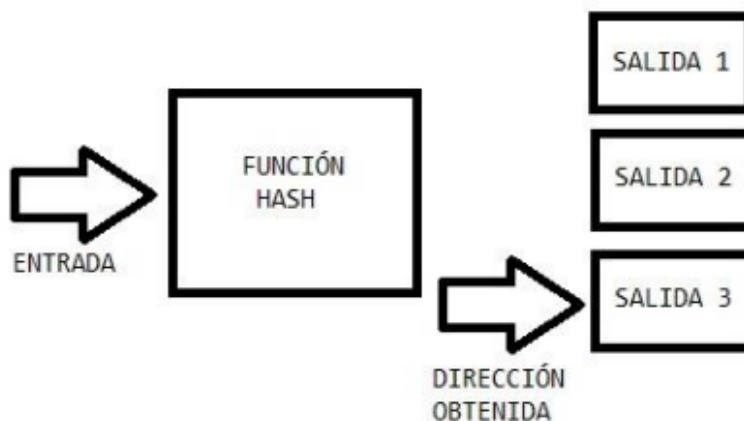


Fig. 1 Las funciones hash. Una vez que la función hash procesa los datos de entrada nos dará una ruta directa a los datos de salida.

05

```
def convCad(cad):
    salida = " "
    for j in cad:
        salida += str(ord(j))
    return int(salida)
def hashM(cad,m):
    i = convCad(cad)
    return int(m*(i * 0.000000000000212324 % 1))
def agregar(cad,ht,m):
    res=hashM(cad,m)
    ht[res].append(cad)
def buscar(cad,ht,m):
    h = hashM(cad,m)
    for i in ht[h]:
        if i == cad:
            return True
    return False
```

```
m=19
ht = [[] for i in range(m)]
agregar("pizarron", ht,m)
agregar("plumon", ht,m)
agregar("borrador", ht,m)
agregar("impresora", ht,m)
agregar("pluma", ht,m)
agregar("cuaderno", ht,m)
print(ht)
print(buscar("pluma", ht,m))
print(buscar("libro", ht,m))
```

```
[[], [], [], [], ['plumon'], [], ['borrador'], ['pluma'], ['impresora'], ['cuaderno'], [], [], [], [], [], [], [], ['pizarron']]
True
False
```


ALGORITMOS DE ORDENAMIENTO:

Los algoritmos de ordenamiento nos permiten, como su nombre lo dice, ordenar información de una manera especial basándonos en un criterio de ordenamiento.

Los algoritmos de ordenación sirven para reorganizar el orden de los elementos de una estructura, como un vector. Muestran de forma estructurada cómo ordenar los elementos que haya dentro de una estructura, como un array.

Una de las características de los algoritmos de ordenación es que se puede ordenar cualquier estructura con elementos que sean ordenables.

- Podemos ordenar números, porque unos son mayores que otros
- Podemos ordenar meses, porque unos vienen antes que otros.

Veamos los siguientes algoritmos de ordenamiento:

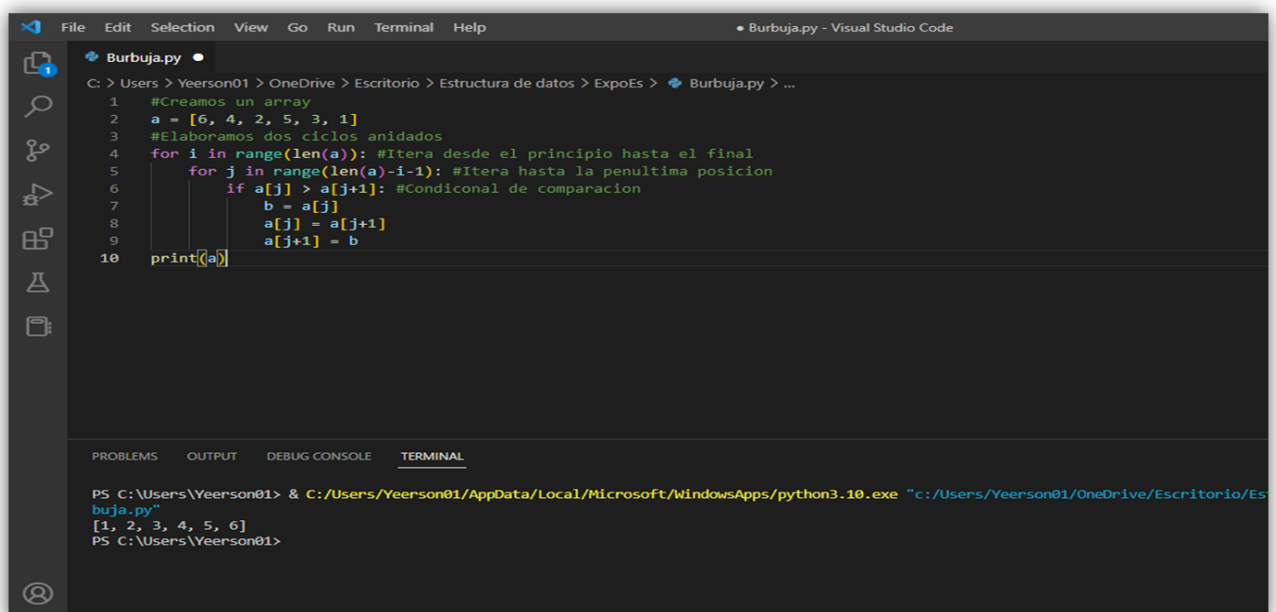
1. Ordenamiento de Burbuja

Este algoritmo de clasificación simple itera sobre la lista de datos, Comparando elementos en pares hasta que los elementos más grandes “Burbujean” hasta el final de la lista y los más pequeños permanecen al principio.

Ventajas: La ventaja principal del ordenamiento de burbuja es que es muy popular y fácil de implementar. Además, en este tipo de ordenamiento, los elementos se intercambian sin utilizar almacenamiento temporal adicional, de modo que el espacio requerido es el mínimo.

Desventajas: No se recomienda utilizarlo ya que consta de un código bastante extenso y no se comporta adecuadamente con una lista que contenga un número grande de elementos. además de que el bucle es muy repetitivo al comparar elementos y ordenados del array.

Véase Gráficamente:



```
File Edit Selection View Go Run Terminal Help
Burbuja.py
C: > Users > Yeerson01 > OneDrive > Escritorio > Estructura de datos > ExpoEs > Burbuja.py > ...
1 #Creamos un array
2 a = [6, 4, 2, 5, 3, 1]
3 #Elaboramos dos ciclos anidados
4 for i in range(len(a)): #Itera desde el principio hasta el final
5     for j in range(len(a)-i-1): #Itera hasta la penultima posicion
6         if a[j] > a[j+1]: #Condicional de comparacion
7             b = a[j]
8             a[j] = a[j+1]
9             a[j+1] = b
10 print(a)
```

```
PS C:\Users\Yeerson01> & C:/Users/Yeerson01/AppData/Local/Microsoft/WindowsApps/python3.10.exe "C:/Users/Yeerson01/OneDrive/Escritorio/Estructura de datos/ExpoEs/Burbuja.py"
[1, 2, 3, 4, 5, 6]
PS C:\Users\Yeerson01>
```

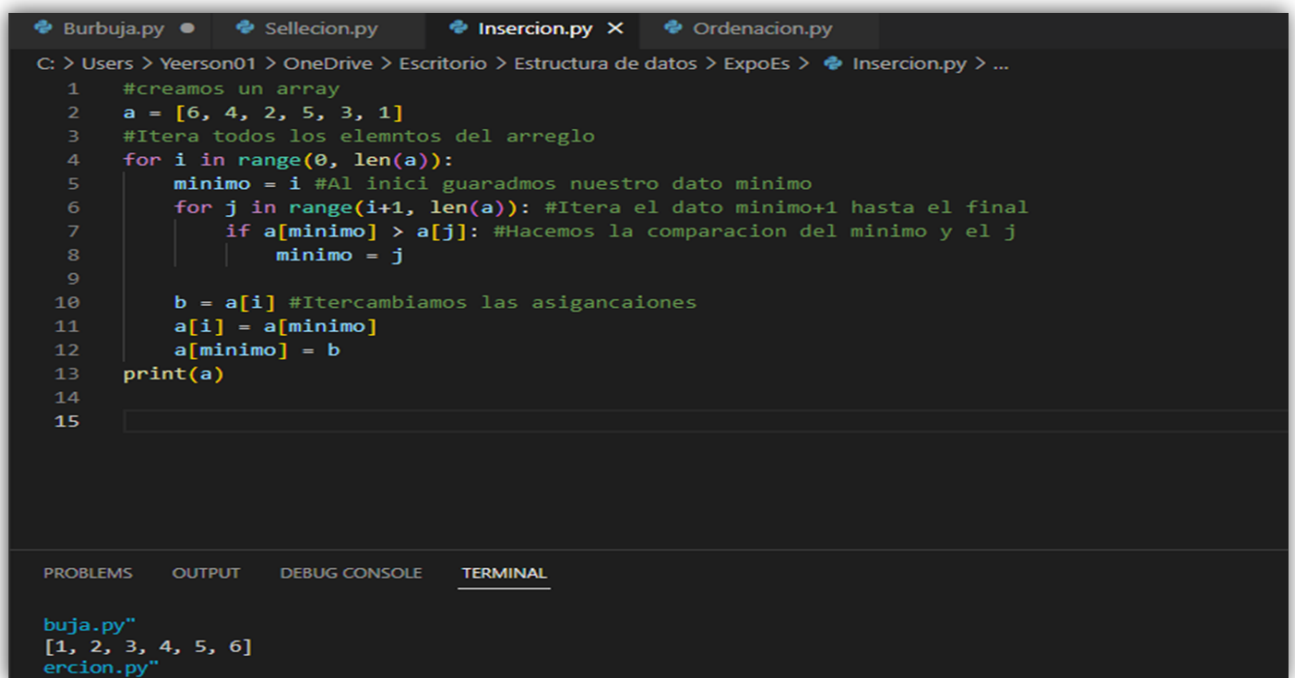
2. Ordenamiento por Inserción

Este algoritmo, consiste en el recorrido por la lista seleccionado en cada iteración un valor como clave y compararlo con el resto insertandolo en el lugar correspondiente

Ventajas: La principal ventaja es su simplicidad y buen rendimiento cuando se trabaja con una pequeña lista.

Desventajas: A diferencia de los demás algoritmos de ordenamiento este no funciona bien con una lista grande. Por lo tanto, este solo es útil cuando se ordena una lista de pocos elementos.

Véase Gráficamente:



```
C: > Users > Yeerson01 > OneDrive > Escritorio > Estructura de datos > ExpoEs > Insercion.py > ...
1  #creamos un array
2  a = [6, 4, 2, 5, 3, 1]
3  #Itera todos los elemntos del arreglo
4  for i in range(0, len(a)):
5      minimo = i #Al inici guaradmos nuestro dato minimo
6      for j in range(i+1, len(a)): #Itera el dato minimo+1 hasta el final
7          if a[minimo] > a[j]: #Hacemos la comparacion del minimo y el j
8              minimo = j
9
10     b = a[i] #Intercambiamos las asignaciones
11     a[i] = a[minimo]
12     a[minimo] = b
13     print(a)
14
15
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
buja.py"
[1, 2, 3, 4, 5, 6]
ercion.py"
```

3. Ordenamiento por selección

Descripción: Mejora con respecto al método de la burbuja. Busca el elemento menor de la lista y se intercambia con el primero, luego se busca el siguiente menor de la lista y se intercambia con el segundo y así sucesivamente.

Ventajas: Funciona bien con una lista pequeña. Además, debido a que es un algoritmo de ordenamiento en el lugar, no hay almacenamiento temporal adicional más allá de lo que se necesita para mantener la lista original.

Desventajas: Su poca eficiencia cuando se trata con una enorme lista de elementos.

Mejor caso: $O(n^2)$

Caso promedio: $O(n^2)$.

Peor caso: $O(n^2)$

Código

```
#Ordenamiento por selección
def ordenSeleccion(lista):

    for i in range(len(lista)-1): #bucle padre inicio i=0
        min = i #primer elemento será el minimo

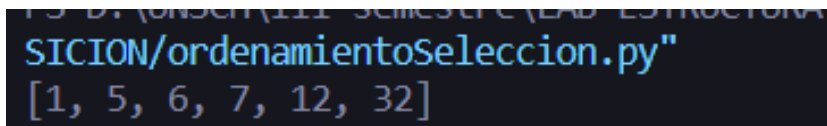
        for j in range(i+1,len(lista)): #bucle hijo el que busca al numero menor
            if lista[j]<lista[min]:
                min=j

        if min != i:
            lista[min], lista[i] = lista[i], lista[min]

def mostrarLista(lista):
    for numero in lista:
        print(numero)

lista = [5,12,1,6,32,7]
ordenSeleccion(lista)
print(lista)
```

Ejecución del código:



```
SICION/ordenamientoSeleccion.py"
[1, 5, 6, 7, 12, 32]
```

4. Ordenamiento Shell

Compara todos los elementos de un arreglo, haciendo saltos entre ellos depende la longitud del arreglo. Generalmente se inicia con un salto inicial = $n/2$, donde n es el número de elementos en la lista, posteriormente el salto se reduce a la mitad.

Ventajas:

No requiere memoria adicional

Fácil implementación

Desventajas:

Realiza numerosas comparaciones e intercambios

Pasos para el ordenamiento Shell

Calcular intervalo $\rightarrow n/2$

Se clasifica cada grupo por separado

Replicar método de inserción

Actualizar intervalo, es decir ($n/4$)

Concluye cuando se alcanza el tamaño de salto 1

Código:

```

def ordenamientoShell(lista):
    contadorSubListas = len(lista)//2
    while contadorSubListas>0:
        for posicionInicio in range(contadorSubListas):
            brechaOrdenamientoInsercion(lista,posicionInicio,contadorSubListas)
        print("Despues del incremento de tamaño: ", contadorSubListas,"la lista es: ", lista)
        contadorSubListas=contadorSubListas//2

    def brechaOrdenamientoInsercion(lista,inicio,brecha):
        for i in range(inicio+brecha,len(lista),brecha):
            valorActual1 = lista[i]
            posicion=i
            while posicion>= brecha and lista[posicion-brecha]>valorActual1:
                lista[posicion]= lista[posicion-brecha]
                posicion = posicion-brecha
            lista[posicion]=valorActual1

    lista=[20,4,36,13,16,19,5,4]
    print (lista)
    ordenamientoShell(lista)
    print("la lista ordenada es: ", lista)

```

Ejecución del código:

```

[20, 4, 36, 13, 16, 19, 5, 4]
Despues del incremento de tamaño: 4 la lista es: [16, 4, 5, 4, 20, 19, 36, 13]
Despues del incremento de tamaño: 2 la lista es: [5, 4, 16, 4, 20, 13, 36, 19]
Despues del incremento de tamaño: 1 la lista es: [4, 4, 5, 13, 16, 19, 20, 36]
la lista ordenada es: [4, 4, 5, 13, 16, 19, 20, 36]

```

5. Ordenamiento binario

El ordenamiento binario es un algoritmo de ordenación de tipo comparación. Es una modificación del algoritmo de ordenamiento por inserción.

En este algoritmo, también mantenemos una submatriz ordenada y otra sin ordenar. La única diferencia es que encontramos la posición correcta de un elemento utilizando la búsqueda binaria en lugar de la búsqueda lineal. Esto ayuda a acelerar el algoritmo de ordenación reduciendo el número de comparaciones necesarias.

Algoritmo de ordenamiento binario (pasos / operaciones):

Supongamos que tenemos un array $A[]$ con " n " elementos. El primer elemento ($A[0]$) ya está ordenado y en la submatriz ordenada.

O1: Marca el primer elemento de la submatriz sin ordenar $A[1]$ como la clave.

O2: Usa la búsqueda binaria para encontrar la posición (p) de $A[1]$ dentro de la submatriz.

O3: Desplaza los elementos de " p " 1 paso hacia la derecha e inserta $A[1]$ en su posición correcta.

O4: Repite los pasos anteriores para todos los elementos de la submatriz sin ordenar.

Ejemplo de ordenación binaria:

Tenemos el array $A = [5, 3, 4, 2, 1, 6]$.

PASO 1: Ordenación por inserción

Subarrayado ordenado	Subarray no ordenado	Array
(5)	(3, 4, 2, 1, 6)	(5, 3, 4, 2, 1, 6)

- Primera iteración

Clave: $A[1] = 3$

PASO 2: Búsqueda binaria

Devuelve la posición 3 como índice 0. Desplazamiento a la derecha de los demás elementos del array ordenado.

Subarrayado ordenado	Subarray no ordenado	Array
(3, 5)	(4, 2, 1, 6)	(3, 5, 4, 2, 1, 6)

- Segunda iteración

Clave: $A[2] = 4$

PASO 3: Repetición de la búsqueda binaria

➤ Posición: 4

Índice: 1

Subarrayas ordenadas	Subarray no ordenado	Array
(3, 4, 5)	(2, 1, 6)	(3, 4, 5, 2, 1, 6)

- Tercera iteración

Clave: $A[3] = 2$

➤ Posición: 2

Índice: 0

Subarrayas ordenadas	Subarray no ordenado	Array
(2, 3, 4, 5)	(1, 6)	(2, 3, 4, 5, 1, 6)

- Cuarta iteración

Clave: $A[4] = 1$

➤ Posición: 1

Índice: 0

Subarrayas ordenadas	Subarray no ordenado	Array
(1, 2, 3, 4, 5)	(6)	(1, 2, 3, 4, 5, 6)

- Quinta iteración

Clave: A[5] = 6

➤ Posición: 6

Índice: 5

Subarrayado ordenado	Subarray no ordenado	Array
(1, 2, 3, 4, 5, 6)	()	(1, 2, 3, 4, 5, 6)

Obteniendo el array ordenado.

Código python:

```
def busqueda_binaria(A, c, low, high):
    if high <= low:
        if c > A[low]:
            return low + 1
        else:
            return low
    mid = int((low+high)/2)
    if c == A[mid]:
        return mid
    if c > A[mid]:
        return busqueda_binaria(A,c,mid+1,high)
    else:
        return busqueda_binaria(A,c,low,mid-1)

def ordenamiento_binario(A,n):
    for i in range (1,n):
        j = i-1
        clave = A[i]
        p = busqueda_binaria(A,clave,0,j)
        while j >= p:
            A[j+1] = A[j]
            j = j-1
        A[j+1] = clave
    print(A)
A = [5,3,4,2,1,6]
print(A)
n=6
ordenamiento_binario(A,n)
```

Ejecución del código:

```
[5, 3, 4, 2, 1, 6]
```

```
[1, 2, 3, 4, 5, 6]
```

Complejidad:

PEOR CASO: $O(n \log n)$

CASO PROMEDIO: $O(n \log n)$

MEJOR CASO: $O(n)$

6. Ordenamiento rápido (Quicksort)

El ordenamiento rápido es un algoritmo de ordenación muy eficiente basado en el principio del algoritmo de dividir y conquistar.

El ordenamiento rápido funciona dividiendo el array en dos partes alrededor de un elemento pivote seleccionado. Mueve los elementos más pequeños a la izquierda del pivote y los más grandes a la derecha. Después de esto, las subpartes izquierda y derecha se ordenan recursivamente para ordenar todo el array.

Se denomina ordenamiento rápido porque es unas 2 o 3 veces más rápido que los algoritmos de ordenación habituales. El ordenamiento rápido se utiliza ampliamente para la búsqueda de información y los cálculos numéricos de las estructuras de datos.

Algoritmo de ordenamiento rápido (pasos / operaciones):

Supongamos que tenemos un array $A[]$ con " n " elementos. Tomamos dos variables "**beg**" y "**end**", y almacenamos el índice del elemento inicial y final.

Partition()

O1: Selecciona el último elemento como elemento pivote.

O2: Inicializa el valor del puntero "**i**" a "**beg-1**" para poder mover los elementos más pequeños que el pivote al inicio del array.

O3: Recorre iterativamente el array.

O4: Si el elemento $A[i] < \text{pivote}$ incrementa "**i**" e intercambia $A[i]$ con $A[j]$ (para cada elemento del array).

O5: Intercambia $A[i]$ con $A[\text{end}]$ para poner el elemento pivote en su posición correcta

O6: Retorna el índice del elemento pivote.

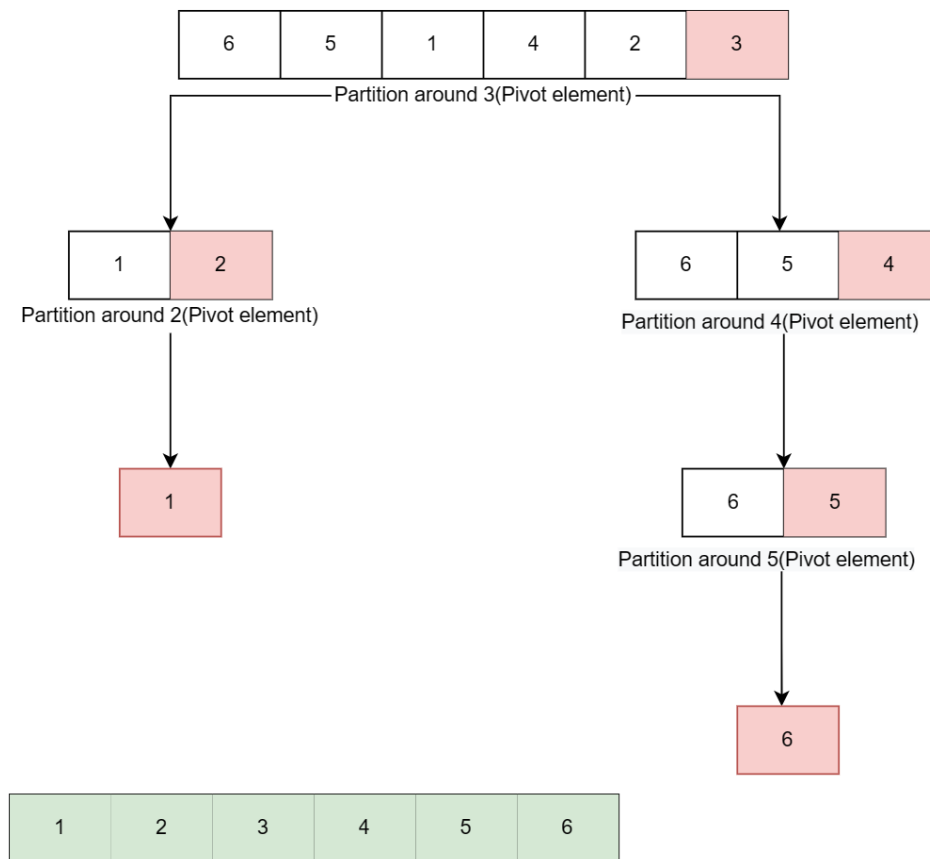
Quicksort()

O1: Selecciona el índice del pivote.

O2: Particionar el array alrededor del índice pivote.

O3: Ordena recursivamente los elementos del lado izquierdo $A = [\text{beg}, \dots, \text{pi}]$ del elemento pivote.

O4: Ordena recursivamente los elementos del lado derecho $A = [\text{pi}+1, \dots, \text{end}]$ del elemento pivote.



Ejemplo de ordenamiento rápido:

Tenemos el array $A = [6, 5, 1, 4, 2, 3]$.

PASO 1:

Seleccionar el elemento pivote

Elemento pivote: 3

Subpartes: $[1, 2]$ y $[6, 5, 4]$

Se coloca el elemento pivote en su posición ordenada y se ordenan recursivamente las subpartes.

PASO 2: Para la subparte $[1, 2]$

Seleccionar el elemento pivote

Elemento pivote: 2

Se coloca el elemento pivote en su posición correcta y se forma la subparte $[1]$ que ya está ordenada.

PASO 3: Para la subparte [6,5,4]

Seleccionar el elemento pivote

Elemento pivote: 4

Se coloca el elemento pivote en su posición y se forma el subarray [6,5]

PASO 4: Para la subparte [6,5]

Seleccionar el elemento pivote

Elemento pivote: 5

Se coloca el elemento pivote en su posición y se forma el subarray [6]

PASO 5:

Obtenemos el array ordenado como:

A = [1,2,3,4,5,6]

Código python:

```
def partition(A,beg,end):
    pivot = A[end]
    i = beg-1
    for j in range (beg,end):
        if A[j] <= pivot:
            i = i + 1
            A[i], A[j] = A[j], A[i]
    A[i+1], A[end] = A[end], A[i+1]
    return i+1

def quicksort(A,beg,end):
    if beg < end:
        pi = partition(A,beg,end)
        quicksort(A,beg,pi-1)
        quicksort(A,pi+1, end)

A = [6,5,1,4,2,3]
n = 6
print(A)
quicksort(A,0,n-1)
print(A)
```

Ejecución del código:

```
[6, 5, 1, 4, 2, 3]
```

```
[1, 2, 3, 4, 5, 6]
```

Complejidad:

PEOR CASO: $O(n^2)$

CASO PROMEDIO: $O(n \log n)$

MEJOR CASO: $O(n \log n)$

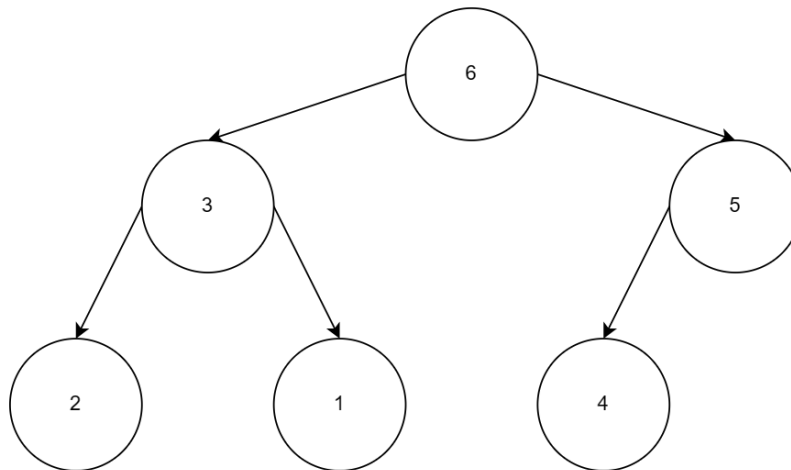
7. Ordenamiento por montículos (Heapsort)

El ordenamiento por montículos es un algoritmo de ordenación basado en la comparación. Su nombre proviene de la estructura de datos del montón utilizada en el algoritmo.

El montón es una estructura de datos especial basada en un árbol binario. Tiene las siguientes propiedades:

- Es un árbol binario completo con todos los niveles llenos excepto el último. El último puede estar parcialmente lleno, pero todos los nodos están lo más a la izquierda posible.
- Todos los nodos padres son más pequeños/más grandes que sus dos nodos hijos. Si son más pequeños, el montón se llama **min-heap**, y si son más grandes, entonces el montón se llama max-heap. Para un índice dado "*i*", el padre está dado por $(i-1)/2$, el hijo izquierdo está dado por $(2*i+1)$ y el hijo derecho está dado por $(2*i+2)$.

El ordenamiento por montículos funciona de forma muy similar a la ordenación por selección. Selecciona el elemento máximo del array usando **max-heap** y lo coloca en su posición final del array. Hace uso de un procedimiento llamado **heapify()** para construir el montón.

**Algoritmo de ordenamiento por montículos (pasos/operaciones):**

Supongamos que tenemos un array **A []** sin ordenar que contiene "**n**" elementos

HeapSort()

O1: Construye un montón máximo con los elementos del array.

O2: Para cada elemento empezando por el último elemento de **A**. Hacer lo siguiente.

O3: El elemento raíz **A[0]** contendrá el elemento máximo.

O4: Reduce el tamaño del heap en uno y **Heapify()** el heap máximo con último elemento eliminado.

Heapify()

O1: Inicializa el índice parent con el índice del elemento actual.

O2: Calcula **leftChild** como $2*i+1$ y **rightChild** como $2*i+2$.

O3: Si el elemento en el **leftChild** es mayor que el valor en el índice parent establece el índice parent a **rightChild**.

O4: Si el elemento **rightChild** es mayor que el valor en el índice parent establece el índice parent a **rightChild**.

O5: Si el valor del índice parent ha cambiado en los dos últimos pasos, entonces intercambia parent con el elemento actual y recursivamente **heapify** el subárbol del índice parent. En caso contrario, la propiedad heap ya está satisfecha.

Ejemplo de ordenamiento por montículos:

Tenemos el array $A = [5,3,4,2,1,6]$

PASO 1: Construir el montón

$A[0] = 6$

$A = [6,3,5,2,1,4]$

PASO 2: Heapify

➤ Primera iteración

<code>Swap(A[5],A[0])</code>	4 3 5 2 1 6
<code>Heapify()</code>	5 3 4 2 1 6

➤ Segunda iteración

<code>Swap(A[4],A[0])</code>	1 3 4 2 5 6
<code>Heapify()</code>	4 3 1 2 5 6

➤ Tercera iteración

<code>Swap(A[3],A[0])</code>	2 3 1 4 5 6
<code>Heapify()</code>	3 2 1 4 5 6

➤ Cuarta iteración

Swap(A[2],A[0])	1 2 3 4 5 6
Heapify()	2 1 3 4 5 6

➤ Quinta iteración

Swap(A[1],A[0])	1 2 3 4 5 6
Heapify()	1 2 3 4 5 6

➤ Sexta iteración

Swap(A[0],A[0])	1 2 3 4 5 6
Heapify()	1 2 3 4 5 6

Obtenemos el array ordenado.

Código python:

```
def heapify (A,n,i):
    parent = i
    leftChild = 2*i+1
    rightChild = 2*i+2
    if leftChild < n and A[leftChild] > A[parent]:
        parent = leftChild
    if rightChild < n and A[rightChild] > A[parent]:
        parent = rightChild
    if parent != i:
        A[i], A[parent] = A[parent], A[i]
        heapify(A,n,parent)

def heapSort(A,n):
    for i in range (int(n/2-1), -1, -1):
        heapify(A,n,i)
    for i in range (n-1, 0, -1):
        A[0], A[i] = A[i], A[0]
        heapify(A,i,0)

A = [5,3,4,2,1,6]
print(A)
n = 6
heapSort(A,n)
print(A)
```

Ejecución del código:

```
[5, 3, 4, 2, 1, 6]
```

```
[1, 2, 3, 4, 5, 6]
```

Complejidad:PEOR CASO: $O(n \log n)$ CASO PROMEDIO: $O(n \log n)$ MEJOR CASO: $O(n \log n)$ **8. Ordenamiento Radix**

El ordenamiento Radix es un algoritmo de ordenamiento no comparativo. Este algoritmo evita las comparaciones insertando elementos en cubos de acuerdo con el radix (base: es el número de dígitos únicos utilizados para representar números). Ordena los elementos basándose en los dígitos de los elementos individuales. Realiza la ordenación por conteo de los dígitos desde el menos significativo hasta el más significativo. También se ha llamado ordenación en cubo o digital y es muy útil en máquinas paralelas.

Algoritmo de ordenamiento Radix (pasos/operaciones):

Supongamos que tenemos un array $A[]$ sin ordenar que contiene " n " elementos

O1: Encuentre el elemento más grande (**maxm**) en el array.

O2: Ordena cada dígito de maxm empezando por el menos significativo utilizando un algoritmo de ordenación estable.

Ejemplo de ordenamiento por montículos:

Tenemos el array $A = [1851, 913, 1214, 312, 111, 23, 41, 9]$

Al ordenarlo por medio de un algoritmo de ordenación radix. Tenemos:

Índice	Matriz de entrada	Primera iteración	Segunda iteración	Tercera iteración	Cuarta iteración
0	1851	1851	0009	0009	0009
1	0913	0111	0111	0023	0023
2	1214	0041	0312	0041	0041
3	0312	0312	0913	0111	0111
4	0111	0913	1214	1214	0312
5	0023	0023	0023	0312	0913
6	0041	1214	0041	1851	1214
7	0009	0009	1851	0913	1851

En la primera iteración, ordenamos según el lugar de la unidad y luego nos movemos hacia los lugares de las decenas, centenas y millares para obtener el array ordenado.

A = [9,23,41,111,312,913,1214,1851]

Código python:

```
import math
class Radix:
    def countingSort(self, A, digit, radix):
        B = [0]*len(A)
        C = [0]*int(radix)
        for i in range(0,len(A)):
            digitAi = (A[i]/radix**digit)%radix
            C[int(digitAi)] = C[int(digitAi)]+1
        for j in range(1,radix):
            C[j] = C[j] + C[j-1]
        for m in range (len(A)-1,-1,-1):
            digitAi = (A[m]/radix**digit)%radix
            C[int(digitAi)] = C[int(digitAi)]-1
            B[C[int(digitAi)]] = A[m]
        return B

    def radixsort(self,A,radix):
        m = max(A)
        output = A
        digits = int(math.floor(math.log(m,radix)+1))
        for i in range(digits):
            output = self.countingSort(output,i,radix)
        return output

a = Radix()
A = [1851,913,1214,312,111,23,41,9]
print(A)
print (a.radixsort(A,10))
```

Ejecución del código:

```
[1851, 913, 1214, 312, 111, 23, 41, 9]
[9, 23, 41, 111, 312, 913, 1214, 1851]
```

Complejidad:

PEOR CASO: O(n)

CASO PROMEDIO: O(n)

MEJOR CASO: O(n)