Universidad Nacional Autónoma de México

Facultad de Estudios Superiores Aragón

Proyecto Primer Parcial

Equipo 2

Integrantes:

Monzón Lucero Miguel Ángel

Robles Leon Cristopher Ruben

Grupo: 1509

Materia:

Diseño y Análisis de Algoritmos

Profesor:

Miguel Angel Sanchez Hernandez

Índice

Introducción

En este trabajo se presenta la documentación de algunos algoritmos vistos en la clase de Diseño y Análisis de Algoritmos, se harán de una forma ordenada y tratando de seguir el mismo formato y procedimiento en cada uno de ellos. Esta documentación incluye aspectos como el objetivo, el modelo, el pseudocódigo, el algoritmo utilizado, las entradas y salidas del algoritmo para finalizar con la obtención de la función de complejidad, que es la forma en que podemos analizar que tan eficiente es un algoritmo, así como gráficas de algunos de estos algoritmos donde se verá de manera más gráfica su eficiencia y diferencia respecto a otros algoritmos; además de este análisis se incluyen “simulaciones” de los algoritmos y los códigos de cada algoritmo como evidencia de su implementación. Al final del documento también se agrega una conclusión donde agregamos de manera crítica en qué aspectos tuvimos deficiencias y en general un análisis de nuestro desempeño a lo largo de la realización del proyecto.

Objetivos

- Como principal objetivo queremos resolver de manera satisfactoria todos y cada uno de los puntos que se solicitan en el proyecto.
- Nuestro siguiente objetivo es poder comprender cada uno de los algoritmos que analizaremos a lo largo del documento, y poder ver que tan eficiente es un algoritmo respecto a otro.
- Como último objetivo queremos poder llegar a aprender cómo es que se diseña un algoritmo y cómo es que influye de una buena manera que lo hagamos de la manera más eficiente posible

Análisis Algoritmo “Predecesor menor”

Objetivo
Buscar el índice del primer elemento que es menor que el anterior en una lista de cadenas de caracteres.

Restricciones
Los elementos de la lista únicamente deben ser cadenas de caracteres

Modelo
Entradas:

Listas de cadenas de caracteres;

Proceso:

1. Lista = asignamos nuestra lista de cadenas de caracteres.
2. Recorremos la Lista comenzando por el índice 1 de la lista
3. Mientras recorremos la lista se compara el elemento en la posición i con el elemento anterior de la lista, y en el momento que el elemento sea menor que el elemento anterior nos regresa ese índice.

Restricción:

Solo funciona con cadenas de caracteres, es indiferente si son mayúsculas o minúsculas, pero se recomienda que se use el mismo formato.

Descripción del Algoritmo
El algoritmo que se diseñó sirve para recorrer una lista de cadenas de caracteres y obtener el índice del primer elemento que tenga menor cantidad de caracteres que el elemento siguiente.

Entrada
La entrada que se le dio al algoritmo es la siguiente:
["DANIELA", "MIGUEL", "RUBEN", "HUGO", "DENISSE", "FERNANDO", "PAULINA", "SANTIAGO", "VALENTINA"]

Salida
La salida que se busca obtener es la siguiente: es el número 3

Pseudocódigo
```
predecesor(Lista){
        for(i=1; i<Lista.longitud; i++){
                if( Lista[i].longitud < Lista[i - 1].longitud){
                        return[i - 1]
        }
}
```

Funcion de Evaluacion
Para comprobar que el modelo es correcto, mi función me dice el índice del primer elemento que es menor que el siguiente.

Función de Complejidad
3(n) = 3n = n
Mejor caso

3(1) = 3
Peor caso
3(n) = 3n = n

Conclusiones
En este problema pudimos observar que se tienen que analizar bien las instrucciones al momento de generar un algoritmo para que se pueda resolver y aplicar de la manera en que se solicita, al inicio estábamos confundidos en que es lo que se solicitaba, pero después de preguntar y analizarlo estábamos interpretando de una manera errónea el problema o lo que se solicitaba. Resuelto esto pudimos resolverlo y aplicarlo a lo que se indicaba en las instrucciones y funciona de manera correcta.

Análisis Algoritmo “Suma máxima de lista consecutivos”

Objetivo
En este caso particular se realizaron 3 versiones de algoritmo que pudieran resolver el objetivo de regresar la suma máxima de valores consecutivos en una lista de números enteros y además que pudieran mostrar los elementos que dan como resultado la suma máxima.

Restricciones
Los elementos de la lista únicamente deben ser números enteros.

Modelo
Entradas:

Lista de números enteros en el orden que sea.

Proceso:

1. sumaMaxima = La suma consecutiva de los números contenidos en una lista.

Restricción:

Únicamente se pueden números enteros, no se permiten cadenas de caracteres ni flotantes o booleanos.

Nota: Estos puntos se aplican para las 3 versiones del algoritmo.

Descripción del Algoritmo

Versión 1
El algoritmo usado en esta versión realiza el armado de una tabla con la cual podemos ordenar los elementos de la lista e ir guardando las sumas entre estos elementos, posteriormente al armado de esta tabla, el algoritmo recorre esta tabla y escoge la suma máxima entre los elementos de la lista. Devuelve la suma máxima y los índices de los elementos entre los que obtuvo la suma máxima.

Versión 2
En esta versión el algoritmo realiza la misma función que la versión anterior, crea una tabla con las sumas de los elementos de la lista, pero a la vez que crea la tabla compara y guarda el valor de la suma más grande, lo que nos ayuda hacer el algoritmo un poco más eficiente. Devuelve la suma máxima y los índices de los elementos entre los que obtuvo la suma máxima.

Versión 3

En esta última versión ya no se crea una tabla con las sumas entre los elementos, simplemente se recorre la lista una vez y a la par se van haciendo comparaciones entre las sumas de los elementos y se va guardando la suma máxima entre estos, esto nos beneficia ya que se vuelve muy eficiente en comparación con las otras versiones. Devuelve únicamente la suma máxima.

Entrada

Para hacer la prueba de estos algoritmos se utilizó la misma lista para las 3 versiones para que la comprobación de su funcionamiento arroje la misma salida.

Lista = [107, 38, 104, -59, -77, 81, -78, -94, 115, -145, -135, -81, 137, -62, 125, 58, 33, -7, -107, -139, 70, -105, 117, -23, 94, -47, -66, -101, 20, 61, 43, 83, 128, 8, 71, 97, -114, -65, -134, 17, 65, 28, 82, -40, -22, 126, -60, 81, 49, -98, -116, 36, -138, 127, -66, 122, 137, 95, 98, 44, -6, 111, -143, 122, 43, 86, -122, 36, 138, -78, 6, 68, -116, -30, -129, 115, -47, 142, 34, 80, -146, -25, -80, -138, 134, 14, 137, -74, 73, -144, -101, -137, -130, -143, -33, 79, 38, 85, 96, -63]

Salida

Según la Lista las salidas que nos debe arrojar son:
Versión 1
(1075, 79, 28)

Versión 2
(1075, 79, 28)

Versión 3
1075

Pseudocódigo

Versión 1

```
maxSuma(lista){
        for(i=0; i<n; i++){
                for(j=0; j=i-1; j++){
                        suma[i][j] = suma[i-1][j]+lista[i]
                }
                suma[i][i] = lista[i]
        }
        max = 0
        for(i=1; i<n; i++){
                for(j=1; j=i-1; j++){
                        if(suma[i][j] > max){
                                max = suma[i][j]
                }
        }
        return max
}
```

Versión 2

```
maxSuma(lista){
	max = 0
	for(i=0; i<n; i++){
		for(j=1; j=i-1; j++){
			suma[i][j] = suma[i-1][j]+lista[i]
			if(suma[i][j] > max){
				max = suma[i][j]
			}
		}
		suma[i][i] = lista[i]
		if(suma[i][j] > max){
			max = suma[i][i]
		}
	}
	return max
}
```

Versión 3

```
maxSuma(lista){
	max = 0
	suma = 0
	for(i=0; i<n; i++){
		if(suma+lista[i] > 0){
			suma = suma+lista[i]
		}
		else{
			suma = 0
		}
		if(suma > max){
			max = suma
		}
	}
	return max
}
```

Funcion de Evaluacion

En las 3 versiones se usa la misma lista de números para que en los 3 casos nos muestre el mismo resultado si es que están funcionando los algoritmos. La forma de comprobarlo es hacer manualmente la suma de los elementos de la lista.
La forma de hacerlo es sumar los números que están entre el primer y segundo índice que nos da el algoritmo.

Para este caso los índices son 28 y 79

Los elementos que debemos sumar son los siguientes:

[20, 61, 43, 83, 128, 8, 71, 97, -114, -65, -134, 17, 65, 28, 82, -40, -22, 126, -60, 81, 49, -98, -116, 36, -138, 127, -66, 122, 137, 95, 98, 44, -6, 111, -143, 122, 43, 86, -122, 36, 138, -78, 6, 68, -116, -30, -129, 115, -47, 142, 34, 80]

El resultado esperado es:

Versión 1
(1075, 79, 28)
Versión 2
(1075, 79, 28)
Versión 3
1075

Función de Complejidad

Versión 1

$(2 + n(2 + n(3 + x) + 3)) + (2 + n(2 + n(3 + x) + 3))$

$(2 + 5n + (3 + x)n^2) + (2 + 5n + (3 + x)n^2) = n^2$

Mejor caso

$(2 + 5(1) + (3 + (2))(1)^2) + (2 + 5(1) + (3 + (2))(1)^2) =$

$(2 + 5 + 5 + 2 + 5 + 5) = 22$

Peor caso

$(2 + n(2 + n(3 + x) + 3)) + (2 + n(2 + n(3 + x) + 3))$

$(2 + 5n + (3 + x)n^2) + (2 + 5n + (3 + x)n^2) = n^2$

Versión 2

$2 + n(2 + n(3 + x) + 3 + 3)$

$2 + 5n + (3 + x)n^2 + 3 = n^2$

Mejor caso

$2 + 5(1) + (3 + (4))(1)^2 + 3 = 2 + 5 + 7 + 3 = 17$

Peor caso

$2 + 5n + (3 + (4))n^2 + 3 = n^2$

Versión 3

$2 + 2 + n(x + 3) = n$

Mejor caso

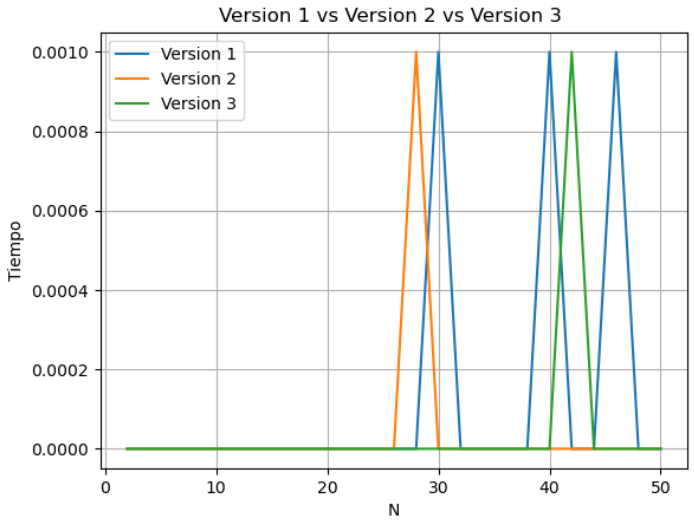
$2 + 2 + (1)((7) + 3) = 2 + 10 = 14$

Peor caso

$2+2 + n((7) + 3) = n$

Conclusiones

En las 3 versiones del algoritmo se llegó a la misma conclusión y resultados en todas las simulaciones que se realizaron, (se anexan los códigos con los que se puede comprobar que con cualquier lista que se proporcione a los algoritmos, los 3 arrojan el mismo resultado) Para tener más claro cómo es que se comportan estos algoritmos decidimos graficar su comportamiento y estos fueron los resultados:



Como podemos observar los 3 algoritmos tienen un comportamiento muy similar con la misma lista, claro está que en el caso de la versión 1 al tener que hacer el recorrido más de una vez, tiene varios picos en los que tarda más que las otras 2 versiones que solo hacen un recorrido de la lista; y en los 3 casos no tienen problemas al hacerlo con cantidades pequeñas de datos, pero al momento de incrementar esta cantidad de datos vemos que las versiones 1 y 2 tienen picos mucho antes que la versión 3.

Análisis Algoritmo “Trayectorias”

Objetivo
Buscar todas las combinaciones de trayectorias posibles entre dos puntos.

Restricciones
Ocupando dos movimientos U=arriba y R=derecha.

Modelo
Entradas:

Dos puntos cualesquiera (x1, y1), (x2, y2)

Proceso:

#R = movimiento a la derecha = x2 - x1

#U = movimientos hacia arriba = y2 - y1

Restricción:

x2 > x1 , y2 > y1

Descripción del Algoritmo
El algoritmo que se ocupó fue el de permutaciones, este algoritmo recibe 2 puntos, a partir de ahí obtiene la cantidad de movimientos que tiene para ir a la derecha y hacia arriba, y busca todas las permutaciones de trayectorias posibles.
Durante el pseudocódigo se mencionan el factorial y swap, estas funciones se programaron como complemento del algoritmo; ya que el factorial es la forma en la que el algoritmo sabe la cantidad de permutaciones que debe hacer.

Entrada
[#U , #D]
Nota: el algoritmo no permite caracteres repetidos por lo tanto cambiamos los datos
2 = U
1 = D
Ejemplo (0,0) (2,3)
#R = 2 - 0 = 2
#U = 3 - 0 = 3
[#U , #D] = [D,D,U,U,U]
cambiamos variables
[1,1,2,2,2]

Salida
Nos debe de mostrar las 10 posibles trayectorias que existen del punto (0,0) al punto (2,3)

Pseudocódigo
```
permutaciones(lista){
        s = 0
        listaF = 0
        for(i=0; i<n; i++){
                s[i] = i
        println(s[i],.....s[lista])
        }
        for(i=1; i<n!; i++){
```

```
                m = longitud.lista - 2
                while(s[m] >= s[m+1]){
                        m = m - 1
                }
                k = longitud.lista - 1
                while(s[m] >= s[k]){
                        k = k - 1
                }
                swap(s[m], s[k])
                p = m + 1
                q = longitud.lista - 1
                while(p < q){
                        swap(s[p], s[q])
                        p = p + 1
                        q = q - 1
                }
                if(s not in listaF){
                        println(listaF[s],.....listaF[lista])
                }
        }
}
```

Funcion de Evaluacion

Para comprobar que el modelo es correcto, mi función me dice el número de permutaciones tomando en cuenta la entrada.

Número de movimientos totales su factorial entre la multiplicación del factorial de los movimientos que se repiten.

$n! \,/\, a!\, b! \ldots z!$

tomando el ejemplo tenemos que $5!\,/2!\,3! \;=\; 10$

Función de Complejidad

$2 + 2 + n(1 + 3) + 1 + 2 + n(27 + 3) = n$

Mejor caso

$2 + 2 + (1)(1 + 3) + 1 + 2 + (1)(27 + 3)$=2+2+4+1+2+30=41

Peor Caso

$2 + 2 + n(1 + 3) + 1 + 2 + n(27 + 3) = n$

Conclusiones

Este algoritmo fue de los más complicados de analizar y aunque lo habíamos hecho ya en conjunto como grupo, aun así se complicó en algunas cosas, aunque una vez que comprendes cómo es que funciona ya es más sencillo de entender y comprobar que lo que hace lo hace de forma correcta. Creemos que es un algoritmo que puede ser de mucha ayuda al momento de querer programar trayectorias para un juego o para saber cómo podríamos llevar un objeto de un punto a otro en un ambiente virtual.

Análisis Algoritmos “Métodos de Ordenamiento”

Los siguientes algoritmos se analizaron de forma independiente pero en algunas partes la información que se proporcionó es la misma en los 4 casos, ya que generamos un análisis en conjunto para poder estudiar cual de estos métodos de ordenamiento es más eficiente como fue el caso de los algoritmos de las sumas.

Burbuja

Objetivo

Ordenar de menor a mayor una lista de números enteros.

Restricciones

Únicamente se permite ingresar listas de números enteros

Modelo

Entradas:

Listas de números enteros en el orden que sea.

Proceso:

Restricción:

Únicamente se permite ingresar una lista de números enteros

Descripción del Algoritmo

Este algoritmo hace múltiples pasadas a lo largo de una lista durante estas pasadas compara los elementos adyacentes e intercambia los que no están en orden, con cada pasada a lo largo de la lista ubica el siguiente valor más grande en su lugar apropiado. En esencia, cada elemento de la lista “burbujea” hasta el lugar al que pertenece.

Entrada

A este algoritmo se le proporcionó la siguiente entrada:

Lista = [107, 38, 104, -59, -77, 81, -78, -94, 115, -145, -135, -81, 137, -62, 125, 58, 33, -7, -107, -139, 70, -105, 117, -23, 94, -47, -66, -101, 20, 61, 43, 83, 128, 8, 71, 97, -114, -65, -134, 17, 65, 28, 82, -40, -22, 126, -60, 81, 49, -98, -116, 36, -138, 127, -66, 122, 137, 95, 98, 44, -6, 111, -143, 122, 43, 86, -122, 36, 138, -78, 6, 68, -116, -30, -129, 115, -47, 142, 34, 80, -146, -25, -80, -138, 134, 14, 137, -74, 73, -144, -101, -137, -130, -143, -33, 79, 38, 85, 96, -63]

NOTA: A este algoritmo se le proporcionó aparte una lista aleatoria para poder medir el tiempo que tardaba en realizar el ordenamiento de la lista, los parámetros que tenía la lista que se generó fue una lista de 15000 elementos en un rango de -500 a 500. Esto con el fin de analizar un poco mejor el algoritmo en las conclusiones.

Salida

La salida que debe dar es la lista ordenada:

[-146, -145, -144, -143, -143, -139, -138, -138, -137, -135, -134, -130, -129, -122, -116, -116, -114, -107, -105, -101, -101, -98, -94, -81, -80, -78, -78, -77, -74, -66, -66, -65, -63, -62, -60, -59, -47, -47, -40, -33, -30, -25, -23, -22, -7, -6, 6, 8, 14, 17, 20, 28, 33, 34, 36, 36, 38, 38, 43,

43, 44, 49, 58, 61, 65, 68, 70, 71, 73, 79, 80, 81, 81, 82, 83, 85, 86, 94, 95, 96, 97, 98, 104, 107, 111, 115, 115, 117, 122, 122, 125, 126, 127, 128, 134, 137, 137, 137, 138, 142]

Pseudocódigo

```
metodo_burbuja(Lista){
        for(i=1; i<Lista.longitud; i++){
                for(j=0; i<Lista.longitud-1; i++){
                        if(Lista[j] > Lista[j+1]){
                                auxiliar = Lista[j]
                                Lista[j] = Lista[j+1]
                                Lista[j+1] = auxiliar
                        }
                }
        }
}
```

Funcion de Evaluacion

La forma de comprobar es verificando que cada elemento esté ordenado del menor al mayor.

Función de Complejidad

$2 + n(2 + n(3 + x) + 3)$

$2 + 5n + (3 + x)n^2 = n^2$

Mejor Caso

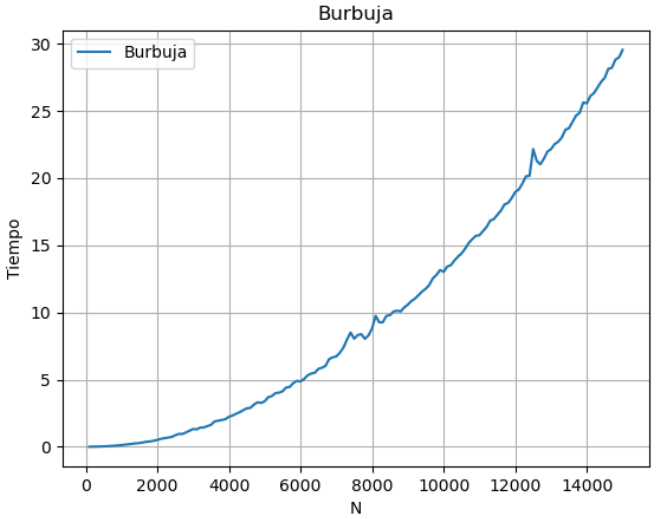
$2 + 5(0) + (3 + 4)(0)^2 = 2 = 2$

Peor Caso

$2 + 5n + (3 + 4)n^2 = n^2$

Conclusiones

Para hacer nuestro análisis realizamos la medición del tiempo que tarda este algoritmo en ordenar una lista tan grande y obtuvimos los siguientes resultados:



Como podemos observar en la gráfica el algoritmo de ordenamiento de burbuja no tiene problemas con pocos datos y es relativamente rápido, pero mientras le damos más datos mucho más tiempo le toma ordenarlos.

NOTA: ESTE ANÁLISIS ES EN BASE A LOS RESULTADOS MOSTRADOS LA GRÁFICA Y TOMANDO EN CUENTA QUE LOS 4 MÉTODOS DE ORDENAMIENTO ORDENARON LA MISMA LISTA DE ELEMENTOS ALEATORIOS.

Selección

Objetivo
Ordenar de menor a mayor una lista de números enteros.

Restricciones
Únicamente se permite ingresar listas de números enteros

Modelo
Entradas: Listas de números enteros en el orden que sea.
Proceso:

Restricción: Únicamente se permite ingresar una lista de números enteros

Descripción del Algoritmo
Este algoritmo busca el elemento que es menor mientras hace el recorrido de la lista, para colocarlo en su posición correcta al finalizar el primer recorrido. Cuando realiza el siguiente recorrido busca el segundo elemento menor para colocarlo en su posición correcta al finalizar ese recorrido, este proceso lo realiza n-1 veces ya que el elemento final debería estar en su lugar después del penúltimo recorrido.

Entrada
A este algoritmo se le proporcionó la siguiente entrada:

Lista = [107, 38, 104, -59, -77, 81, -78, -94, 115, -145, -135, -81, 137, -62, 125, 58, 33, -7, -107, -139, 70, -105, 117, -23, 94, -47, -66, -101, 20, 61, 43, 83, 128, 8, 71, 97, -114, -65, -134, 17, 65, 28, 82, -40, -22, 126, -60, 81, 49, -98, -116, 36, -138, 127, -66, 122, 137, 95, 98, 44, -6, 111, -143, 122, 43, 86, -122, 36, 138, -78, 6, 68, -116, -30, -129, 115, -47, 142, 34, 80, -146, -25, -80, -138, 134, 14, 137, -74, 73, -144, -101, -137, -130, -143, -33, 79, 38, 85, 96, -63]

NOTA: A este algoritmo se le proporcionó aparte una lista aleatoria para poder medir el tiempo que tardaba en realizar el ordenamiento de la lista, los parámetros que tenía la lista que se generó fue una lista de 15000 elementos en un rango de -500 a 500. Esto con el fin de analizar un poco mejor el algoritmo en las conclusiones.

Salida
La salida que debe dar es la lista ordenada:

[-146, -145, -144, -143, -143, -139, -138, -138, -137, -135, -134, -130, -129, -122, -116, -116, -114, -107, -105, -101, -101, -98, -94, -81, -80, -78, -78, -77, -74, -66, -66, -65, -63, -62, -60,

-59, -47, -47, -40, -33, -30, -25, -23, -22, -7, -6, 6, 8, 14, 17, 20, 28, 33, 34, 36, 36, 38, 38, 43, 43, 44, 49, 58, 61, 65, 68, 70, 71, 73, 79, 80, 81, 81, 82, 83, 85, 86, 94, 95, 96, 97, 98, 104, 107, 111, 115, 115, 117, 122, 122, 125, 126, 127, 128, 134, 137, 137, 137, 138, 142]

Pseudocódigo

```
metodo_seleccion(Lista){
        for(i=0; i<Lista.longitud; i++){
                mínimo = i
                for(j=i; j<Lista.longitud; i++){
                        if(Lista[j] < Lista[mínimo]){
                                mínimo = j
                        }
                }
                if(mínimo != i){
                        auxiliar = Lista[i]
                        Lista[i] = Lista[mínimo]
                        Lista[mínimo] = auxiliar
                }
        }
}
```

Funcion de Evaluacion

La forma de comprobar es verificando que cada elemento esté ordenado del menor al mayor.

Función de Complejidad

$2 + n(2 + n(3 + 2) + 3 + 5)$

$2 + 5n + (3 + 2)n^2 + 5 = n^2$

Mejor caso

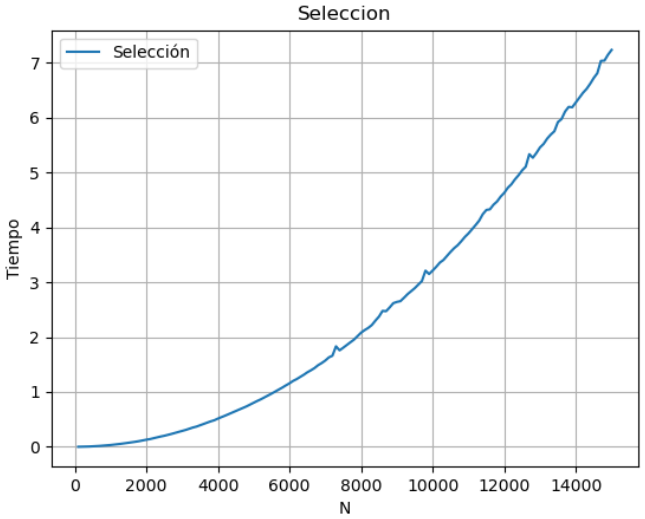
$2 + 5(0) + (3 + 2)(0)^2 + 5 = 2 + 5 + 5 + 5 = 17$

Peor caso

$2 + 5n + (3 + 2)n^2 + 5 = n^2$

Conclusiones

Para hacer nuestro análisis realizamos la medición del tiempo que tarda este algoritmo en ordenar una lista tan grande y obtuvimos los siguientes resultados:



En este caso podemos observar que al igual que con el ordenamiento de burbuja no tiene problema con una cantidad pequeña de datos, pero mientras aumentamos esa cantidad de datos le cuesta un poco más ordenarlos, pero aunque tarda en ordenar muchos datos, es más eficiente que el ordenamiento de burbuja.

NOTA: ESTE ANÁLISIS ES EN BASE A LOS RESULTADOS MOSTRADOS LA GRÁFICA Y TOMANDO EN CUENTA QUE LOS 4 MÉTODOS DE ORDENAMIENTO ORDENARON LA MISMA LISTA DE ELEMENTOS ALEATORIOS.

Inserción

Objetivo
Ordenar de menor a mayor una lista de números enteros.

Restricciones
Únicamente se permite ingresar listas de números enteros

Modelo
Entradas: Listas de números enteros en el orden que sea.
Proceso:

Restricción: Únicamente se permite ingresar una lista de números enteros

Descripción del Algoritmo
En este algoritmo se asume que el elemento en la posición 0 es una sublista ordenada, mientras se va recorriendo la lista principal a partir del índice 1, ese elemento se compara con los elementos de esta sublista y se va ordenando. Con cada pasada los elementos de la lista principal se comparan y ordenan en la sublista, este recorrido lo hacen n-1 veces.

Entrada
A este algoritmo se le proporcionó la siguiente entrada:

Lista = [107, 38, 104, -59, -77, 81, -78, -94, 115, -145, -135, -81, 137, -62, 125, 58, 33, -7, -107, -139, 70, -105, 117, -23, 94, -47, -66, -101, 20, 61, 43, 83, 128, 8, 71, 97, -114, -65,

-134, 17, 65, 28, 82, -40, -22, 126, -60, 81, 49, -98, -116, 36, -138, 127, -66, 122, 137, 95, 98, 44, -6, 111, -143, 122, 43, 86, -122, 36, 138, -78, 6, 68, -116, -30, -129, 115, -47, 142, 34, 80, -146, -25, -80, -138, 134, 14, 137, -74, 73, -144, -101, -137, -130, -143, -33, 79, 38, 85, 96, -63]

NOTA: A este algoritmo se le proporcionó aparte una lista aleatoria para poder medir el tiempo que tardaba en realizar el ordenamiento de la lista, los parámetros que tenía la lista que se generó fue una lista de 15000 elementos en un rango de -500 a 500. Esto con el fin de analizar un poco mejor el algoritmo en las conclusiones.

Salida

La salida que debe dar es la lista ordenada:

[-146, -145, -144, -143, -143, -139, -138, -138, -137, -135, -134, -130, -129, -122, -116, -116, -114, -107, -105, -101, -101, -98, -94, -81, -80, -78, -78, -77, -74, -66, -66, -65, -63, -62, -60, -59, -47, -47, -40, -33, -30, -25, -23, -22, -7, -6, 6, 8, 14, 17, 20, 28, 33, 34, 36, 36, 38, 38, 43, 43, 44, 49, 58, 61, 65, 68, 70, 71, 73, 79, 80, 81, 81, 82, 83, 85, 86, 94, 95, 96, 97, 98, 104, 107, 111, 115, 115, 117, 122, 122, 125, 126, 127, 128, 134, 137, 137, 137, 138, 142]

Pseudocódigo

```
metodo_insercion(Lista){
        for(j=1; j<Lista.longitud; i++){
                i = j-1
                x = Lista[j]
                while(x < Lista[i] and i >= 0){
                        Lista[i+1] = Lista[i]
                        i = i-1
                }
                Lista[i+1] = x
        }
}
```

Funcion de Evaluacion

La forma de comprobar es verificando que cada elemento esté ordenado del menor al mayor.

Función de Complejidad

$2 + n(3 + 3 + n(3 + 3)) = n^2$

Mejor caso

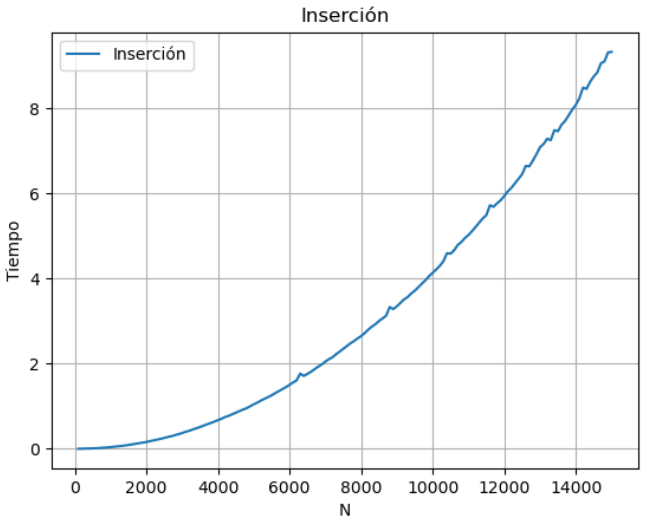
$2 + (1)(3 + 3 + (0)(3 + 3)) = n$

Peor caso

$2 + n(3 + 3 + n(3 + 3)) = n^2$

Conclusiones

Para hacer nuestro análisis realizamos la medición del tiempo que tarda este algoritmo en ordenar una lista tan grande y obtuvimos los siguientes resultados:



Este caso al igual que el ordenamiento de selección son mejores que el ordenamiento de burbuja; nuevamente este algoritmo no tiene complicaciones al ordenar una cantidad pequeña de elementos, pero al momento de agregar más elementos sufre un poco incluso sufre ligeramente más que el ordenamiento de selección.

NOTA: ESTE ANÁLISIS ES EN BASE A LOS RESULTADOS MOSTRADOS LA GRÁFICA Y TOMANDO EN CUENTA QUE LOS 4 MÉTODOS DE ORDENAMIENTO ORDENARON LA MISMA LISTA DE ELEMENTOS ALEATORIOS.

Quicksort

Objetivo
Ordenar de menor a mayor una lista de números enteros.

Restricciones
Únicamente se permite ingresar listas de números enteros

Modelo
Entradas: Listas de números enteros en el orden que sea.
Proceso:

Restricción: Únicamente se permite ingresar una lista de números enteros

Descripción del Algoritmo
Este algoritmo primero selecciona el primer valor de la lista, esto se le conoce como pivote, este pivote nos va ayudar a dividir la lista. Este pivote se agrega a la lista final ordenada para a partir de este hacer las llamadas a la función de QuickSort.

El proceso de partición busca el punto de división y al mismo tiempo mueve los elementos al lado correcto de la lista, ya sea (< pivote) o (> pivote).

Este algoritmo a diferencia de los otros 3 métodos de ordenamiento no es un algoritmo que hace iteraciones, si no se trata de un algoritmo recursivo, por lo que algunas funciones se llamarán a sí mismas para completar la tarea.

Entrada
A este algoritmo se le proporcionó la siguiente entrada:

Lista = [107, 38, 104, -59, -77, 81, -78, -94, 115, -145, -135, -81, 137, -62, 125, 58, 33, -7, -107, -139, 70, -105, 117, -23, 94, -47, -66, -101, 20, 61, 43, 83, 128, 8, 71, 97, -114, -65, -134, 17, 65, 28, 82, -40, -22, 126, -60, 81, 49, -98, -116, 36, -138, 127, -66, 122, 137, 95, 98, 44, -6, 111, -143, 122, 43, 86, -122, 36, 138, -78, 6, 68, -116, -30, -129, 115, -47, 142, 34, 80, -146, -25, -80, -138, 134, 14, 137, -74, 73, -144, -101, -137, -130, -143, -33, 79, 38, 85, 96, -63]

NOTA: A este algoritmo se le proporcionó aparte una lista aleatoria para poder medir el tiempo que tardaba en realizar el ordenamiento de la lista, los parámetros que tenía la lista que se generó fue una lista de 15000 elementos en un rango de -500 a 500. Esto con el fin de analizar un poco mejor el algoritmo en las conclusiones.

Salida
La salida que debe dar es la lista ordenada:

[-146, -145, -144, -143, -143, -139, -138, -138, -137, -135, -134, -130, -129, -122, -116, -116, -114, -107, -105, -101, -101, -98, -94, -81, -80, -78, -78, -77, -74, -66, -66, -65, -63, -62, -60, -59, -47, -47, -40, -33, -30, -25, -23, -22, -7, -6, 6, 8, 14, 17, 20, 28, 33, 34, 36, 36, 38, 38, 43, 43, 44, 49, 58, 61, 65, 68, 70, 71, 73, 79, 80, 81, 81, 82, 83, 85, 86, 94, 95, 96, 97, 98, 104, 107, 111, 115, 115, 117, 122, 122, 125, 126, 127, 128, 134, 137, 137, 137, 138, 142]

Pseudocódigo
```
metodo_quicksort(lista){
	quicksort_auxiliar(Lista,0,Lista.longitud-1)
}

quicksort_auxiliar(lista, primero, último){
	if(primero < ultimo){
		puntoDivision = particion(lista, primero, último)
		quicksort_auxiliar(lista, primero, puntoDivision-1)
		quicksort_auxiliar(lista, puntoDivision+1, último)
	}
}

particion(lista, primero, ultimo){
	pivote = lista[primero]
	marcaIzq = primero+1
	marcaDer = último
	hecho = false
	while(hecho != true){
		while(marcaIzq <= marcaDer and lista[macaIzq] <= pivote){
			marcaIzq = marcaIzq+1
		}
		while(marcaDer >= marcaIzq and lista[macaDer] >= pivote){
			marcaDer = marcaDer-1
```

```
                }
                if(marcaDer < marcaIzq){
                        hecho = true
                }
                else{
                        temp = lista[macaIzq]
                        lista[marcaIzq] = lista[marcaDer]
                        lista[marcaDer] = temp
                }
        }
        temp = lista[primero]
        lista[primero] = lista[marcaDer]
        lista[marcaDer] = temp

        return marcaDer
}
```

Funcion de Evaluacion

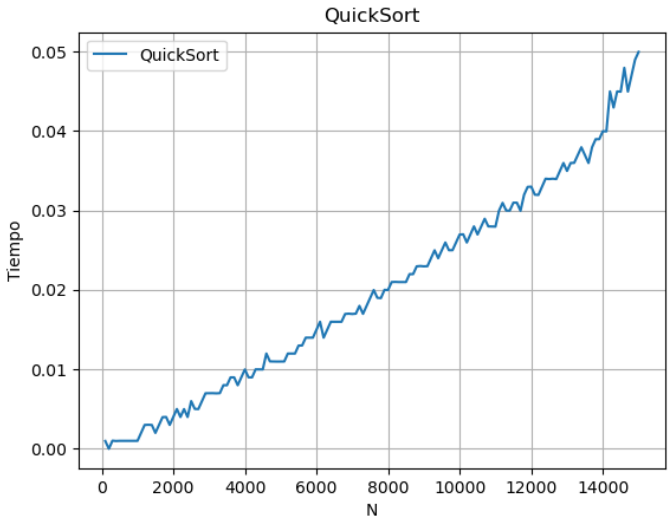
La forma de comprobar es verificando que cada elemento esté ordenado del menor al mayor.

Función de Complejidad

O(n·log n).
Mejor caso
O(n·log n).
Peor caso
O(n²)

Conclusiones

Para hacer nuestro análisis realizamos la medición del tiempo que tarda este algoritmo en ordenar una lista tan grande y obtuvimos los siguientes resultados:



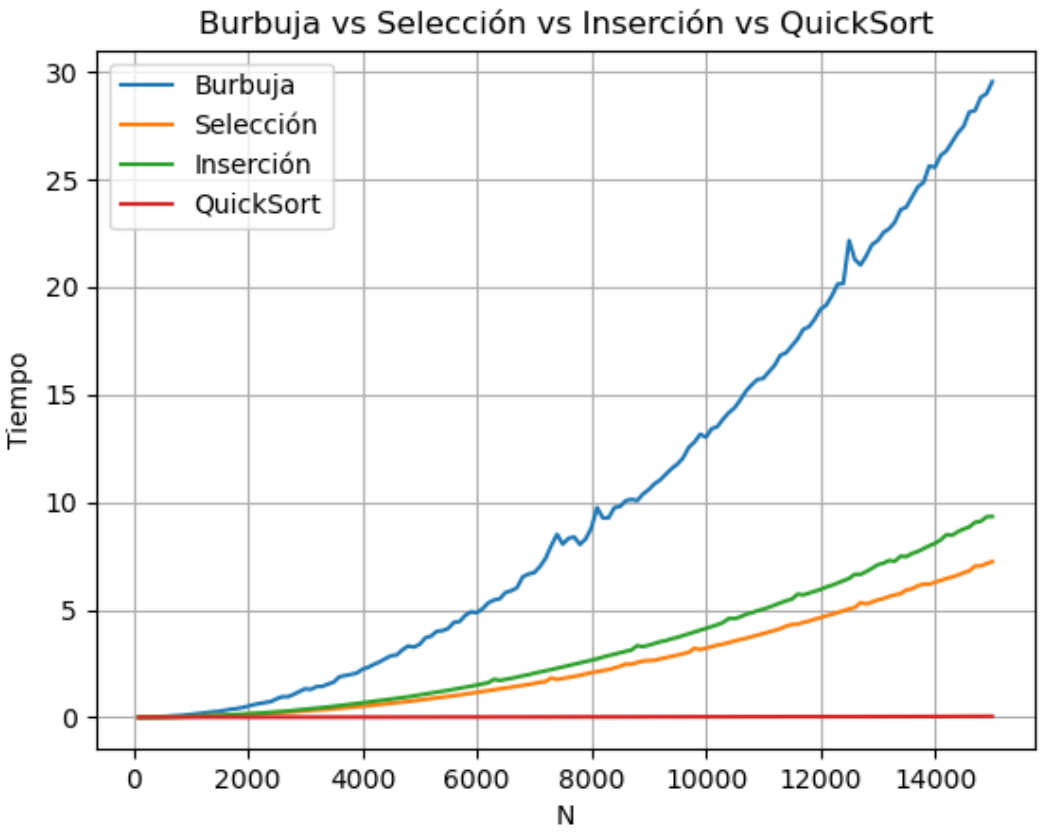
En este método de ordenamiento podemos observar una enorme diferencia respecto a los otros 3 métodos, este algoritmo no tiene ningún problema con pocos o muchos elementos, hace el ordenamiento increíblemente rápido, aunque no es el método más lineal de los 3

pues como podemos notar su gráfica es muy desigual en comparación de a las de los otros 3 ordenamientos que es un poco más lineal; aun así este método es el más eficiente de todos, sobre todos cuando se trata de ordenar una cantidad considerable de elementos.

NOTA: ESTE ANÁLISIS ES EN BASE A LOS RESULTADOS MOSTRADOS LA GRÁFICA Y TOMANDO EN CUENTA QUE LOS 4 MÉTODOS DE ORDENAMIENTO ORDENARON LA MISMA LISTA DE ELEMENTOS ALEATORIOS.

Conclusiones Generales Métodos de Ordenamiento

Para juntar la información que obtuvimos sobre las gráficas de los métodos de ordenamiento, realizamos la gráfica conjunta de los 4 métodos para ver de forma más clara cómo es que unos son más eficientes que otros.

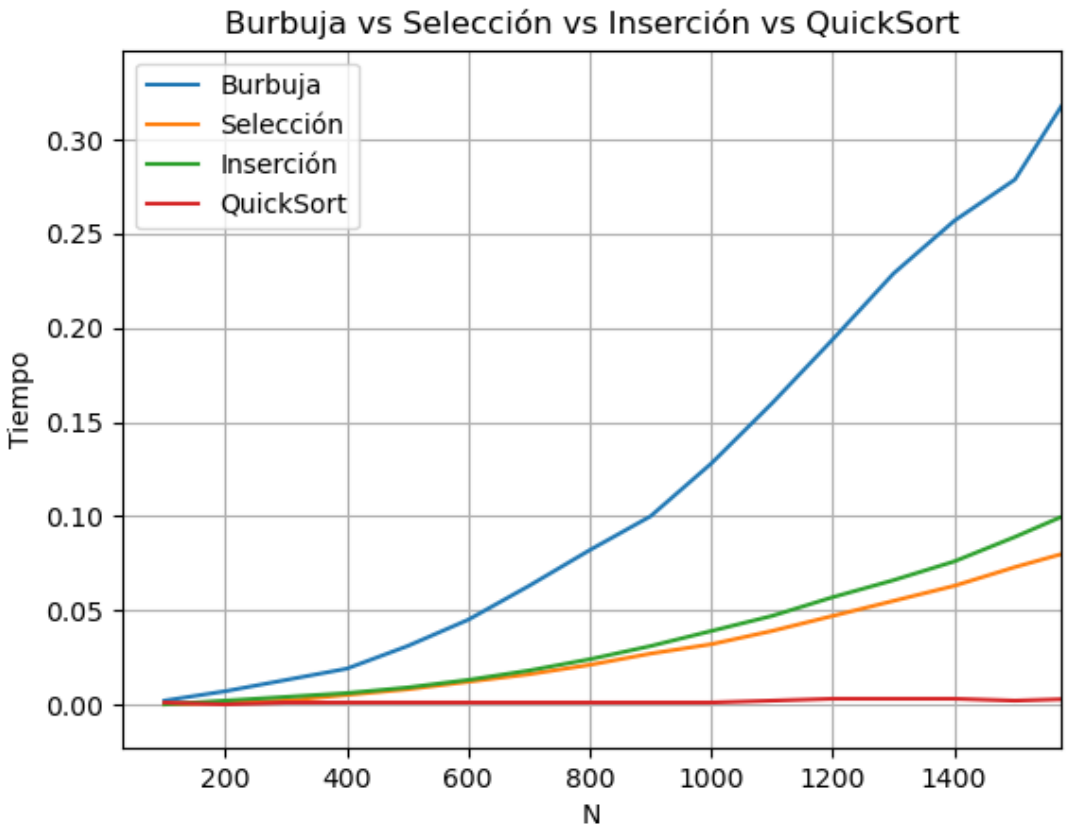


Como se mencionó en las conclusiones individuales de los métodos algunos resultaron ser más eficientes que otros al momento de ordenar una cantidad grande de elementos. Para nuestro caso práctico con 15000 elementos podemos notar la diferencia abismal que existe entre los métodos.

Entre el método burbuja y los otros métodos podemos observar que se trata de un método que con una gran cantidad de elementos es prácticamente inservible debido a su tiempo de ejecución que se representa de forma exponencial.

Entre el método de selección e inserción podemos encontrar que no existe gran diferencia al momento de hacer el ordenamiento de los elementos, van casi de la mano.

Entre el método QuickSort y los otros métodos podemos encontrar que prácticamente ni siquiera se separa del eje x, y aunque ordena la misma cantidad de elementos se mantiene por debajo de los otros 3 métodos.



Si nos fijamos bien en el principio de la gráfica podemos observar que a partir de aproximadamente los 100 elementos el método burbuja comienza a subir exponencialmente.

A diferencia de los métodos de selección e inserción que aproximadamente a partir de los 400 elementos comienzan a irse hacia arriba pero a un ritmo menor que el método burbuja.

En el caso del método QuickSort aun con el zoom que se hizo no se nota la separación del eje x, solo se nota un poco aproximadamente a partir del elemento 1100.

Referencias

- *5.7. El ordenamiento burbuja — Solución de problemas con algoritmos y estructuras de datos*. (s. f.). Runestone. Recuperado 10 de octubre de 2021, de https://runestone.academy/runestone/static/pythoned/SortSearch/ElOrdenamientoBurbuja.html

- *5.8. El ordenamiento por selección — Solución de problemas con algoritmos y estructuras de datos*. (s. f.). Runestone. Recuperado 14 de octubre de 2021, de https://runestone.academy/runestone/static/pythoned/SortSearch/ElOrdenamientoPorSeleccion.html

- 5.9. El ordenamiento por inserción — Solución de problemas con algoritmos y estructuras de datos. (s. f.). Runestone. Recuperado 14 de octubre de 2021, de https://runestone.academy/runestone/static/pythoned/SortSearch/ElOrdenamientoPorInsercion.html

- 5.12. El ordenamiento rápido — Solución de problemas con algoritmos y estructuras de datos. (s. f.). Runestone. Recuperado 14 de octubre de 2021, de https://runestone.academy/runestone/static/pythoned/SortSearch/ElOrdenamientoRapido.html