**Diseño de Experimento sobre Ordenamiento de un Arreglo.**

**Integrantes:**

* Santiago Álvarez
* Nicolás Sanabria
* Juan Felipe Castillo
* Mayumi Tamura

1. **Planeación y realización.**

El objeto de estudio será sobre las diferentes maneras que se puede ordenar un arreglo, se definirán dos tipos de algoritmos de ordenamiento con la misma complejidad temporal y sobre estos se trabajarán los diferentes factores que pueden afectar su proceso.

La variable a considerar será el tiempo que tarde la ejecución del algoritmo, este tiempo será tomado con una herramienta proporcionada por el lenguaje C#.

Se experimentará con varios factores además de la diferencia de algoritmo y se repetirá el tratamiento numerosas veces.

Este experimento se compone de:

* **Unidad experimental:**  Arreglos de números enteros con diferentes condiciones de tamaño y orden inicial.
* **Factor de estudio:** Ordenamiento de arreglos de enteros.
* **Variable de respuesta:** Tiempo de ejecución.
* **Factores controlables:** Algoritmo de ordenamiento. Estado inicial de los valores en el arreglo (en orden aleatorio, orden ascendente y orden descendente). Tamaño del arreglo. Memoria RAM.
* **Factores no controlables:** Temperatura ambientes, desgaste del procesador.

**Niveles y Tratamientos:**

Para este estudio utilizaremos dos algoritmos, el heapsort y el quicksort, ambos con una complejidad de 𝜣(n log n), se varía el tamaño del arreglo en 20, 450, y 1000, se ejecutarán en todos los órdenes (ascendente, aleatorio y descendente) y, además, se realizarán los experimentos con memoria RAM de 8Gb y 12Gb.

|  |  |  |  |
| --- | --- | --- | --- |
| **Algoritmo de ordenamiento** | **Tamaño del arreglo** | **Estado de los valores del arreglo.** | **Memoria RAM** |
| Heapsort | 20 | Ascendente | 8 Gb |
| Heapsort | 20 | Ascendente | 12 Gb |
| Heapsort | 450 | Ascendente | 8 Gb |
| Heapsort | 450 | Ascendente | 12 Gb |
| Heapsort | 1000 | Ascendente | 8 Gb |
| Heapsort | 1000 | Ascendente | 12 Gb |
| Heapsort | 20 | Aleatorio | 8 Gb |
| Heapsort | 20 | Aleatorio | 12 Gb |
| Heapsort | 450 | Aleatorio | 8 Gb |
| Heapsort | 450 | Aleatorio | 12 Gb |
| Heapsort | 1000 | Aleatorio | 8 Gb |
| Heapsort | 1000 | Aleatorio | 12 Gb |
| Heapsort | 20 | Descendente | 8 Gb |
| Heapsort | 20 | Descendente | 12 Gb |
| Heapsort | 450 | Descendente | 8 Gb |
| Heapsort | 450 | Descendente | 12 Gb |
| Heapsort | 1000 | Descendente | 8 Gb |
| Heapsort | 1000 | Descendente | 12 Gb |
| Quicksort | 20 | Ascendente | 8 Gb |
| Quicksort | 20 | Ascendente | 12 Gb |
| Quicksort | 450 | Ascendente | 8 Gb |
| Quicksort | 450 | Ascendente | 12 Gb |
| Quicksort | 1000 | Ascendente | 8 Gb |
| Quicksort | 1000 | Ascendente | 12 Gb |
| Quicksort | 20 | Aleatorio | 8 Gb |
| Quicksort | 20 | Aleatorio | 12 Gb |
| Quicksort | 450 | Aleatorio | 8 Gb |
| Quicksort | 450 | Aleatorio | 12 Gb |
| Quicksort | 1000 | Aleatorio | 8 Gb |
| Quicksort | 1000 | Aleatorio | 12 Gb |
| Quicksort | 20 | Descendente | 8 Gb |
| Quicksort | 20 | Descendente | 12 Gb |
| Quicksort | 450 | Descendente | 8 Gb |
| Quicksort | 450 | Descendente | 12 Gb |
| Quicksort | 1000 | Descendente | 8 Gb |
| Quicksort | 1000 | Descendente | 12 Gb |

**Complejidad temporal HeapSort.**

MAX-HEAPIFY(A, i)

1 l = LEFT.(i)

2 r = RIGHT.(i )

3 if l <= A:heap-size and A[l] > A[i]

4 largest = l

5 else largest = i

6 if r <= A:heap-size and A[r] > A[largest]

7 largest = r

8 if largest != i

9 exchange A[i] with A[largest]

10 MAX-HEAPIFY(A, largest)

La función de este algoritmo es T(n) ≤ T(2n/3) + Θ(1)1

Por teorema del maestro la complejidad es de O(lg n)

BUILD-MAX-HEAP(A)

1 A.heap-size = A.length

2 for i = [A.length/2] downto 1

3 MAX-HEAPIFY(A, i )

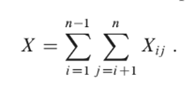
La complejidad del BUILD-MAX-HEAP es Θ(n)[1]

[1] Cormen, T.,Leiserson C.,Rivest, R., & Stein, C.. (2009). Introduction to algorithms. Massachusetts: Mit Pr

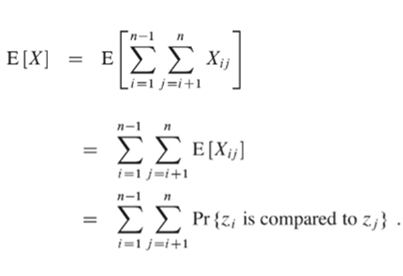
**Complejidad temporal QuickSort.**

El tiempo de ejecución esperado de RANDOMIZED-QUICKSORT es O(n lg n): si, en cada nivel de recursión, la división inducida por RANDOMIZED-PARTITION coloca una fracción constante de los elementos en un lado de la partición, entonces el árbol de recursión tiene profundidad Ө(lg n) y O(n) en cada nivel. Incluso si agregamos algunos niveles nuevos con la división más desequilibrada posible entre estos niveles, el tiempo total sigue siendo O (n lg n).

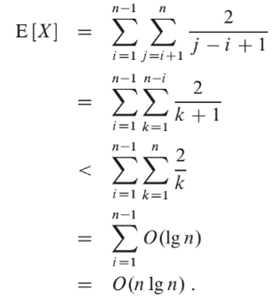
El análisis utiliza variables indicadoras aleatorias. Se define:  donde se considera si la comparación tiene lugar en cualquier momento durante la ejecución del algoritmo, no sólo durante una iteración o una llamada de PARTICIÓN. Como cada par se compara una vez como máximo, se puede caracterizar fácilmente el número total de comparaciones realizadas por el algoritmo:



Tomando las expectativas de ambos lados, y luego utilizando la linealidad de expectativa, se obtiene:



Debido a que el conjunto Zij tiene (j - i + 1) elementos, y debido a que los pivotes se eligen de forma aleatoria e independiente, la probabilidad de que cualquier elemento dado sea el primero elegido como pivote es 1 / (j - i + 1). Así, se obtiene:



Cormen, T.,Leiserson C.,Rivest, R., & Stein, C.. (2009). Introduction to algorithms. Massachusetts: Mit Pr

**Complejidad espacial HeapSort.**

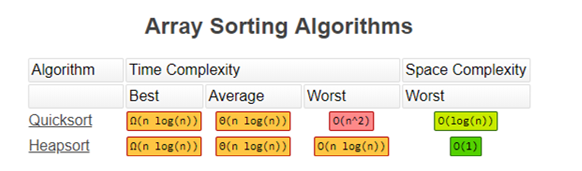
La complejidad espacial del algoritmo HeapSort en el peor de los casos es de O(n) en total y O(1) auxiliar.

**Complejidad espacial QuickSort.**

El espacio utilizado por quicksort depende de la versión utilizada.

La versión en contexto de quicksort tiene una complejidad de espacio de O (log n), incluso en el peor de los casos, cuando se implementa cuidadosamente utilizando las siguientes estrategias:

* Se utiliza la partición en el lugar. Esta partición inestable requiere espacio O (1).
* Después de la partición, la partición con el menor número de elementos se ordena (recursivamente) primero, lo que requiere a lo sumo espacio O (log n). Luego, la otra partición se ordena usando la recursión o iteración de cola, que no se agrega a la pila de llamadas.



HeapSort - <https://en.wikipedia.org/wiki/Heapsort>

QuickSort - <https://en.wikipedia.org/wiki/Quicksort#Space_complexity>

<http://bigocheatsheet.com/>

**2)** **Análisis.**

**Gráfico de Resultados.**

Analizando los datos del experimento se encuentra que claramente hay diferencias entre los resultados debido a lo que se conoce como error experimental

Al comparar dichos resultados en cuanto al tamaño de el arreglo, se observa que este factor influye de manera significativa, ya que el tiempo que tarda el algoritmo está estricta y directamente relacionado con el tamaño del arreglo introducido, a mayor tamaño, mayor es el tiempo de ejecución para ordenarlo.

En cuando a estado del arreglo antes de aplicarle el algoritmo, se puede ver que el que más se demora en general es el que está ordenado descendentemente, seguido de el aleatorio y el ordenado ascendentemente, los cuales varían. A veces el ascendente demora más, otras veces el aleatorio, pero siempre es una diferencia muy corta entre ellos.

Ahora, al revisar el factor memoria, se observa que en el HeapSort es más eficiente cuando se ejecuta con una RAM de 12gb, mientras que el QuickSort es algo indiferente, los valores varían pero con rangos muy cortos.

Por último, se puede decir que el algoritmo QuickSort es el que tiene tiempos más cortos comparado con el HeapSort en cualquier caso experimentado.

**3)** **Interpretación.**

Con base en los datos analizados se puede interpretar que la mejor opción es usar datos ordenados ascendentemente, pero lo que se busca con los algoritmos de ordenamiento es ordenar conjuntos de datos de forma aleatoria, por lo tanto no siempre se van a recibir conjuntos ordenados. Por lo tanto, el mejor tratamiento sería los datos ordenados ascendentemente con el algoritmo de QuickSort, ya que este presenta tiempos de ejecución menores que el HeapSort. Pero en casos reales, los conjuntos de datos van a estar desordenados aleatoriamente, para esto se observa que en la mayoría de casos entre más memoria RAM tenga el equipo, más rápido ejecuta el ordenamiento. También hay que tener en cuenta que la velocidad de ejecución se puede ver afectada por factores no controlables como el desgaste del procesador, ya que se aprecia que en algunos casos, la memoria RAM de mayor capacidad es menos eficiente que una de menor capacidad, lo cual contradice una hipótesis inicialmente intuitiva. Por otro lado, la temperatura ambiente del equipo puede ralentizar este proceso, haciendo que el tiempo de ejecución con unas características sea menor que el otro.

Para finalizar, se obtiene como tratamiento ganador el algoritmo de ordenamiento QuickSort, dado que presenta los tiempo de ejecución más eficientes en términos generales, además de esto, con una memoria RAM de 12gb y entre menor cantidad de datos, más rápido será el tiempo de ejecución.

**4) Conclusión.**

Finalmente, a partir del experimento se puede decidir que el mejor tratamiento para ordenar un conjunto de datos es el algoritmo de QuickSort, con una memoria de 12 Gb y con el conjunto de datos de menor tamaño, debido a que este algoritmo trabaja con árboles y este divide el problema en pequeños subproblemas, haciendo uso del paradigma *divide y vencerás*. Lo anterior con el objetivo de establecer unas medidas a implementar para generalizar el resultado del experimento y garantizar así que las mejoras se mantengan en futuras prácticas. También es posible concluir que es importante tener en cuenta otros factores no controlables como lo son: el desgaste del procesador, ya que de este depende en gran parte la velocidad de ejecución, y la temperatura ambiente, que también puede influir en que el proceso se realice de manera más rápida o más lenta.