Algoritmos Dividir y Conquistar

Para este trabajo compararemos los tiempos de ejecución de diversos tipos de algoritmos dividir y conquistar, para ello se ha creado diversos Datasets mediante código los cuales se encuentran disponibles en el repertorio adjuntado al final de este archivo, los algoritmos que serán contrastados son Selection sort, Mergesort, Quicksort y ….. esto para los casos de ordenamiento de datos mientras que para los casos de multiplicación de matrices contrastaremos iterativo cubico tradicional, iterativo cubico optimizado y Strassen. En los que se espera ver como para el ordenamiento de datos el algoritmo de Mergesort seria el mas eficiente y para multiplicación de matrices seria el de Strassen

**Algoritmos de ordenamiento**

Selection sort: El algoritmo de Selection sort es uno de los más lentos en el ordenamiento de datos debido a su forma de trabajar ya que este compara cada elemento del arreglo con el menor que encuentre en una ejecución y luego lo posiciona al inicio del arreglo para luego seguir al siguiente, en el peor de los casos se recorre el arreglo n veces con n valores lo que nos da una complejidad de Ο () esto sucede en el 100% de los casos debido a que el algoritmo consiste en comparar siempre el numero en ejecución con los demás por ende siempre se recorre el arreglo completo independiente de si esta mas o menos ordenado

|  |  |
| --- | --- |
| Número de datos (enteros) | Tiempo de ejecución (segundos) |
| 6.561 | 0,089 |
| 19.683 | 0,804 |
| 65.536 | 9,206 |
| 117.649 | 29,485 |
| 262.144 | 161,106 |
| 823.543 | 1.624,286 |
| 964.324 | 2.181,890 |
| 1.000.000 | 2.399,789 |
| 2.097.152 | , |
| 4.782.969 | , |

Como se puede apreciar los datos siguen una distribución exponencial, es por esto que se realizaron solo hasta 1.000.000 de datos ya que los tiempos serian muy extensos pero seguirían la misma distribución

Mergesort: El algoritmo mergesort consiste en 2 partes principales, un mergesort que se encarga de dividir el arreglo en subarreglos de la mitad de tamaño hasta llegar al mínimo de subdivisión, luego llama a la función merge que toma los lados derecha e izquierda de cada subarreglo y compara sus elementos para clasificarlos de mayor a menor. Basándonos en la lógica de este código podemos llegar a la deducción de su complejidad la cual es Ο () debido a que en cada iteración se hace la mitad de trabajo recursivo, esto es más evidente viendo el teorema maestro T(n) = a T ( ) + c donde a son la cantidad de subproblemas a resolver en cada ejecución, que en este caso sería a = 2, luego vemos en cuantos se divide los arreglos en cada ejecución, lo que nos da b = 2 luego vemos que d = 1 dándonos que el teorema maestro queda de la siguiente forma T(n) = 2 T ( ) + c

|  |  |
| --- | --- |
| Número de datos (enteros) | Tiempo de ejecución (segundos) |
| 6.561 | 0,011 |
| 19.683 | 0,034 |
| 65.536 | 0,135 |
| 117.649 | 0,202 |
| 262.144 | 0,469 |
| 823.543 | 1,498 |
| 964.324 | 1,690 |
| 1.000.000 | 1,802 |
| 2.097.152 | 3,678 |
| 4.782.969 | 9,346 |

La distribución de los datos es casi lineal ya que en la teoría debería seguir la distribución n log (n) lo cual es casi lineal, y se puede apreciar claramente en los resultado obtenidos

Quicksort: el algoritmo de Quicksort consta de utilizar un pivote el cual es elegido de manera aleatoria para comenzar a comparar este con el resto de los elemento y en cada iteración divide el arreglo a la mitad haciendo que en el mejor de los casos nuestro tiempo de ejecución sea de Ο () esto sucederá siempre que elijamos un buen pivote por otro lado el peor caso sería elegir un pivote que sea el primer o el último dato del arreglo y que este estes ordenado haciendo que nuestro tiempo de ejecución sea de Ο () pero por estadística de probabilidades se puede llegar a demostrar como en el promedio de los caso esto no sucede lo cual se puede apreciar en las pruebas realizadas ya que como se puede ver los tiempo de ejecución fueron menores que de Mergesort el cual también trabaja en tiempo casi lineal pero con un menor rendimiento.

|  |  |
| --- | --- |
| Número de datos (enteros) | Tiempo de ejecución (segundos) |
| 6.561 | 0,002 |
| 19.683 | 0,004 |
| 65.536 | 0,012 |
| 117.649 | 0,025 |
| 262.144 | 0,058 |
| 823.543 | 0,193 |
| 964.324 | 0,218 |
| 1.000.000 | 0,233 |
| 2.097.152 | 0,502 |
| 4.782.969 | 1,291 |

La distribución de los datos es bastante similar a la de Mergesort ya que ambos algoritmo tiene un tiempo de ejecución n log(n) pero Quicksort es mas optimo por ende los tiempos de ejecución son significativamente menores

Función sort: La función sort de la librería standard de c++ utiliza un algoritmo de Introsort el cual es un algoritmo de sorting híbrido ósea que utiliza 3 tipos diferentes de sort, el Quicksort, Heapsort y el Insertion sort lo que significa que en general es el mejor algoritmo de sorting que hay ya que en el peor, mejor y promedio, su tiempo de ejecución es de Ο () esto ya que utiliza lo mejor de cada algoritmo, pero no todo es perfecto ya que no es el algoritmo más rápido, simplemente es el algoritmo mas constante ya que Quicksort sigue siendo más rápido sin embargo sort() soluciona el problema de elegir un mal pivote.

|  |  |
| --- | --- |
| Número de datos (enteros) | Tiempo de ejecución (segundos) |
| 6.561 | 0,001 |
| 19.683 | 0,004 |
| 65.536 | 0,014 |
| 117.649 | 0,029 |
| 262.144 | 0,066 |
| 823.543 | 0,254 |
| 964.324 | 0,274 |
| 1.000.000 | 0,280 |
| 2.097.152 | 0,655 |
| 4.782.969 | 1,579 |

Los dato comprueban lo teórico que es que sort() es mucho mas eficiente que otro algoritmo pero sin llegar a ser mejor que Quicksort, lo que en esquelas cantidades de datos no es significativo pero para volúmenes mayores de datos puede llegar a ser significativo pero si llegar a ser terrible

Multiplicación de matrices: Este algoritmo utiliza el método tradicional de multiplicación de matrices iterando cada fila de la matriz A con cada columna de la matriz B siguiendo las restricciones de tamaños (que compartan tamaño de columnaA con filaB) y guardando el resultado en una nueva matriz en la posición correspondiente, esto implica que el tiempo de ejecución de el algoritmo es de Ο () ya que es necesario recorrer n filas contra n columnas para n datos y dado que realiza siempre el mismo tipo de ejecución el mejor y el peor caso sigue siendo el mismo

|  |  |
| --- | --- |
| Tamaño matriz cuadrada | Tiempo de ejecución (segundos) |
| (1024x1024),(1024x243) | 3,318 |
| (2048x2048),(2048x729) | 44,563 |
| (64x256),(256x2048) | 0,457 |
| (81x512),(512x81) | 0.046 |
| (4096x1000),(1000x16) | 0,817 |
| (49x1296),(1296x81) | 0,064 |
| (729x125),(125x2401) | 2,746 |
| (4096x64),(64x16) | 0,054 |
| (3125x32),(32x3125) | 3,949 |
| (1024x2401),(2401x49) | 1,538 |

Los datos muestran como el crecimiento de ejecución es cubico ya que a medida que crecen los datos los tiempos de ejecución se disparan como lo son de a Donde los tiempos pasan de 3 segundos a 44 según aprox.

Multiplicación de matrices Traspuesta: El algoritmo de multiplicación traspuesta realiza el mismo trabajo que el algoritmo de multiplicación de matrices pero trasponiendo una de las matrices para mejorar el rendimiento de este ya que por la forma en que los procesadores obtienen los datos al transponer la matriz estos quedan mejor posicionados para realizar la multiplicación.

|  |  |
| --- | --- |
| Tamaño matriz cuadrada | Tiempo de ejecución (segundos) |
| (1024x1024),(1024x243) | 3,161 |
| (2048x2048),(2048x729) | 38,960 |
| (64x256),(256x2048) | 0,289 |
| (81x512),(512x81) | 0,041 |
| (4096x1000),(1000x16) | 0,801 |
| (49x1296),(1296x81) | 0,069 |
| (729x125),(125x2401) | 2,706 |
| (4096x64),(64x16) | 0,055 |
| (3125x32),(32x3125) | 3,938 |
| (1024x2401),(2401x49) | 1,510 |

Los datos comprueban como la multiplicación transpuesta es mas optima que la multiplicación normal reduciendo así 6 segundos aprox. Respecto de la misma comparación previa

Strassen: El algoritmo de Strassen particiona a la matiz en 8 partes (4 de cada matriz) y luego se crean 7 ecuaciones las cuales son las que se van a utilizar para calcular los valores de cada uno de los bloques de las matrices, esto como se puede apreciar hace baja en 1 las ecuaciones utilizadas para los cálculos de la matriz con respecto a los cálculos originales, esto nos da una complejidad de Ο () por el teorema maestro el cual nos da los 7 ecuaciones que se utilizan respecto a las 2 particiones de los datos y al trabajo de recuperación que es 2 nos da T(n) = 7 T ( ) + c esto nos sugiere que los tiempos de ejecución respecto de un algoritmo tradicional de multiplicación de matrices debería ser más rápido esto en la teoría sin embargo en la practica esto es demasiado difícil de implementar ya que Strassen comienza a ganar en tiempos de ejecución a partir de un volumen muy alto de números lo cual se puede apreciar en los datos recopilados en los cuales los tiempos de ejecución de la multiplicación tradicional es mucho menor que en Strassen

**Explicación del dataset**

Se crearon 2 códigos de dataset distintos para los problemas, el primero es el utilizado para los algoritmos de ordenamiento de datos el cual fue echo mediante chatGPT y donde es necesitan un tamaño que se debe modificar en el código y luego se crean 10 dataset de manera secuencial los cuales siguen una distribución aleatoria, para los casos de prueba utilizados los tamaños fueron los indicados en las tablas previamente enseñadas, el nombre de los dataset siguen el patrón *“números” + n° de dataset creado*, esto para todos los dataset a partir del 2do creado y con un limite de 10 dataset. Para el caso de los algoritmos de multiplicación de matrices se utilizo un algoritmo similar al de ordenamiento, con la diferencia que este crea 2 matrices en cada ejecución (MatrizA y MatrizB) con tamaños variados e iguales simplemente siguiendo las condiciones de los algoritmos y los nombres siguen otro patrón el cual es *“MatrizA” + n° de dataset creado y “MatrizB” + n° de dataset*, el mismo crea lo mas 9 matrices A y B. Sin embargo, para el algoritmo de Strassen se utilizaron matrices de base 2 para realizar las pruebas ya que de otra manera el algoritmo no funciona de manera optima por ende se volvió a ejecutar el algoritmo de multiplicación de matrices con los mismo dataset de Strassen para comparar los tiempos de ejecución de ambos