Documentación y Análisis de algoritmos

En el presente documento se piensa analizar el funcionamiento de los algoritmos con entradas diferentes. El análisis que vamos a emplear es con lo aprendido en el curso de "Análisis de Algoritmos" de la Escuela de Computación situada en el TEC de San José, Costa Rica.

Créditos a:

- Brenes Maleaño Andrés Ottón
- Fernández Jiménez Axel Fernández
- López Saborío Iván Moisés

Análisis de Algoritmos

Algoritmo de “coin change”

La base de este algoritmo consiste en determinar la menor cantidad de monedas a utilizar para dar un vuelto específico. Utiliza programación dinámica, esto consiste en guardar los datos para luego consultarlos, sin necesidad de calcularlos más de una vez.

Notamos que cuando los valores de las monedas distan mucho del vuelto deseado, el algoritmo se vuelve más lento. Por ejemplo, si se tienen 5 monedas con los siguientes valores: 5,10,25,50 y 100, y se desea un vuelto de 1000000, el algoritmo va a tardar más tiempo. Notamos que lo que determina el tiempo del algoritmo es el vuelto deseado, ya que cuando este es un n pequeño se resuelve bastante rápido.

Entradas utilizadas para probar:

Primera corrida

Valores = [10,25,50,100]

Vuelto = 1000

Segunda corrida

Valores = [10,25,50,100]

Vuelto = 10000000

Algoritmo de “Knapsack”

Este algoritmo consiste en conseguir el mejor valor en un bulto cumpliendo la única condición de no superar el peso que este pueda soportar. Tendremos productos con distintos valores, en nuestro caso son aleatorios, el usuario solo debe introducir la cantidad de elementos y el peso que el bulto soporta.

Con cantidades pequeñas de productos y un soporte alto del bulto, el algoritmo se ejecuta bastante rápido. Cuantos más productos intente elegir el usuario, peor será el rendimiento del algoritmo, a la cantidad de 150 elementos el algoritmo ya se torna pesado, sin embargo, debe también considerarse el soporte del bulto, ya que si tenemos muchos productos pero el peso es bajo, se ejecutará de manera rápida.

Primera corrida

Productos = 10

Vuelto = 100

Segunda corrida

Valores = 150

Vuelto = 10000

Tercera corrida

Valores = 250

Vuelto = 1000000

Algoritmo de “Floyd” & “Dijkstra”

Floyd & Dijkstra son algoritmos que se centran en buscar los caminos más cortos en un grafo, o al menos, los que tienen mejor valor.

Con tamaños de grafo pequeños, no se pueden notar diferencias entre los algoritmos. Conforme vamos aumentando el tamaño del grafo notamos que el algoritmo de Floyd se torna más lento que Dijkstra, esto debido a que Floyd busca la mejor ruta entre todos los nodos y Dijkstra solo de uno a todos los demás.

Primera corrida

Tamaño del grafo = 10

Segunda corrida

Tamaño del grafo = 100

Tercera corrida

Tamaño del grafo = 250

Algoritmo de “Hanoi Towers”

Las Torres de Hanoi tiene complejidad 2^n, siendo n el tamaño de las torres, ¿en qué consiste el algoritmo? La primera torre posee cierta cantidad de discos y hay que llevar estos discos hasta la tercera torre sin poner un disco grande encima de uno más pequeño.

Teniendo en cuenta la complejidad, utilizamos n pequeños para probar el algoritmo. A partir de n = 25 el algoritmo se torna lento y pesado para la máquina.

Primera corrida

Tamaño de las torres = 8

Resultado = 00.00 segundos.

Segunda corrida

Tamaño de las torres = 15

Resultado = 00.0024 segundos.

Tercera corrida

Tamaño de las torres = 25

Resultado = 24.2500 segundos.

Algoritmo de "QuickSort"

El algoritmo Quicksort es el mejor en cuanto a ordenamiento se refiere, el algoritmo fácilmente puede ordenar listas de un millón de elementos en tan solo 6 segundos. Su funcionamiento es el siguiente, utiliza un pivote, una lista de menores creadas a partir del pivote y una lista de mayores con la misma lógica. Se hace lo mismo para las listas creadas hasta que estas tengan un solo elemento y luego se juntan las listas.

Quicksort es muy poderoso, así que lo forzamos a hacer ordenamientos de listas desde 500 mil elementos hasta 1 millón de elementos, se comportó como esperábamos, de la mejor manera.

Primera corrida

Tamaño de la lista = 499999

Resultado = 02.7600 segundos.

Segunda corrida

Tamaño de la lista = 750000

Resultado = 04.3001 segundos.

Tercera corrida

Tamaño de la lista = 999999

Resultado = 06.3537 segundos.

Algoritmo de “HeapSort”

Heapsort ordena utilizando árboles, es un algoritmo bastante eficaz pero que no se equipara a Quicksort.

Utilizamos las mismas entradas que en Quicksort para establecer relaciones entre cada algoritmo, cabe destacar que cada lista fue creada aleatoriamente. Y pudimos notar que Heapsort era más lento que Quicksort, sin embargo, no deja de ser eficiente ordenar medio millón de elementos en 10 segundos.

Primera corrida

Tamaño de la lista = 499999

Resultado = 09.2701 segundos.

Segunda corrida

Tamaño de la lista = 750000

Resultado = 14.5759 segundos.

Algoritmo de “N-Matrix Product”

El algoritmo busca la mejor manera de multiplcar n matrices, busca a su vez, la menor cantidad de operaciones posibles para optimizar la matriz final.

No sabíamos como iba a funcionar el algoritmo con una entrada mayor de 25 matrices, por lo que utilizamos 30 y el programa dejó de funcionar, suponemos que fue mucho peso. Luego empezamos con n pequeños y subimos hasta encontrar el momento donde se complicaba el algoritmo. Notamos que el límite es cercano a 10, y cabe mencionar que las dimensiones de las matrices son asignadas aleatoriamente, por lo que esto afectara el algoritmo.

Primera corrida

Cantidad de matrices = 30

Resultado = Programa caído.

Segunda corrida

Cantidad de matrices = 10

Resultado = Rápida respuesta.