Universidad de Santiago de Chile

Facultad de Ingeniería

Departamento de Ingeniería Informática

Laboratorio N°1: Enumeración exhaustiva – Fuerza bruta

Integrante: Matías Cortés.

Ignacio Villarroel.

Profesor: Cristián Sepúlveda.

Ayudante: Cesar Rivera.

Asignatura: Algoritmos Avanzados.

Tabla de contenidos

Introducción…… 3

Método…… 4

a. Herramientas…….... 4

b. Desarrollo Experimental…… 4

Resultado y Análisis……... 6

a. Problema 1……6

b. Problema 2……6

Discusión……. 7

Conclusión……... 8

Apéndice…….. 8

Referencia…… 9

Introducción

En el desarrollo de softwares, la resolución de problemas mediante la implementación de algoritmos es muy común, desde el momento donde se analiza el problema pasando por el modelado de las posibles soluciones hasta la misma construcción del algoritmo permite entender que existen diversas respuestas para una misma problemática. Existen diversos algoritmos que están enfocados a resolver los mismos problemas, sin embargo, hay algunos que son mejores que otros, en términos del tiempo en la que estos se demoran al encontrar la solución.

En el siguiente informe se pretende implementar un algoritmo de enumeración exhaustiva o de fuerza bruta, como posible solución a los problemas que se abarcarán, al lograr cumplir con esta meta, analizar que tan eficiente es esta solución en términos de su complejidad tomando en consideración el tiempo de demora, dándole cada vez mas número por resolver, de esta forma analizar su comportamiento y tabularlo, para obtener el patrón de complejidad.

Cada problema se abordará de manera distinta en particular, por un lado, en la primera problemática se quiere utilizar un forzamiento en la búsqueda de la solución, calculando todas las posibles soluciones y eligiendo la mejor. Mientras que en la segunda se abarcará por medio de la aplicación de “divide y conquista” para la resolución. Los objetivos principales en el desarrollo de la experiencia son:

- Analizar los algoritmos creados en base sus tiempos de ejecución.
- Obtener la complejidad.
- Identificar si lo obtenido está acorde con lo esperado en la experiencia.
- Ver si los algoritmos creados son mejorables.

Método

a. Herramientas

Para el desarrollo óptimo de la experiencia se utilizaron diversos implementos de tipo software, los cuales fueron:

- Editor de texto ATOM para la programación en C. y uso de la terminal Windows.
- Archivos .txt los cuales fueron obtenidos mediante la plataforma uvirtual para las pruebas de código.
- Google drive para el traspaso de información para la creación del trabajo.

Respecto al hardware utilizado para la compilación y aplicación del código, el equipo que se utilizo fue un computador con:

- Procesador: AMD Ryzen 7 4800H CPU 2.90 GHz - 8 núcleos.
- Ram: 16 Gb.
- Sistema: Windows 10x64.

b. Desarrollo Experimental

Bajo el concepto de una aplicación de división del algoritmo en sub-problemas, el cual ambos problemas se desglosaron viendo que necesitaba para que el código funcionara, es por esto que en primer lugar se desarrollo un pseudocódigo, siendo la idea original para la resolución de la problemática.

```
busquedaMaxima(array X, array V, array P, num Limite, num PTotal, num Comb, num Cantidad, num Posicion, num PosicionFinal):
        Num i <- 1
        Si PosicionFinal < Posicion entonces
                Num AcumuladorP
                Para i <- hacia PosicionFinal
                        AcumuladorP <- P[X[i]-1] + AcumuladorP
                Fin
                #Se ven las sumas de los valores V con respecto a P
                Si PTotal >= Acumulador P
                        Num AcumuladorV
                        Para i <- 1 hacia PosicionFinal
                                AcumuladorV <- V[X[i]-1] + AcumuladorV
                        Fin
                #Se comparan los valores si cumplen lo especificado
                Si AcumuladorV > limite
                        limite = AcumuladorV
        #Se crea la recursividad, hasta que el ciclo de combinaciones encuentre al mejor
        Para i <- Comb hacia Cantidad
                X[Posicion] <- i
                BusquedaMaxima(X,V,P,Limite, PTotal, i+1, Cantidad, Posicion+1, PosicionFinal)
```

Pseudocódigo 1.

En el primer problema, se necesita encontrar las posibles combinaciones dado un subconjunto de elementos dado un ponderado, la idea es encontrar la suma de términos que sea la mas cercana a este. Para esto se utilizan idealmente dos arreglos donde cada uno guardará las combinaciones las ponderaciones, al utilizar fuerza bruta, se hará revisar todas las posibles sumas en el arreglo, recorriéndolo en su totalidad hasta realizarlas todas, luego de hacerlo, se verificará cual cumple la ponderación máxima o mas cercana. Se calcula la complejidad del algoritmos, resultando $0(n) = 2^n$, el cual indica que no es un buen algoritmo, sin embargo resulta útil para bajas cantidades de números.

```
caminoCorto(matriz Mapa, num Lugar. array X, num Final):
        Array Posicion[Lugar]
        Num Desplazamiento <- funcionRecorrer(X, Lugar, Mapa)
        Num i <- 0
        Si Desplazamiento < Final entonces
                Final <- desplazamiento
        #Se verifican los caminos en la matriz
        Para i hacia Lugar
                #Se ve que el lugar actual se menor al que ha caminado
                Si Posicion[i] < i entonces
                        Si i % 2 =! 0 entonces #Se cambia el lugar
                                change(X[Posicion[i]], X[i])
                        #Si el modulo es 0 se ve la cabeza del recorrido
                        Sino
                                change(X[Posicion[0], X[i])
                        #Se debe comprobar si el camino es los eficiente posible
                        Desplazamiento <- funcionRecorrer(X, Lugar, Mapa)
                        Si Desplazamiento < Final entonces
                                Final <- desplazamiento
                        #Se vuelve al inicio para un nuevo camino
                        i=0
                        Posicion[i]=Posicion[i]+1
                #Si no se cumple
                Sino
                        Posicion[i]=0
                        i=i+1
```

Pseudocódigo 2.

Para el segundo caso experimental se aplicó una metodología estudiada anteriormente en la asignatura de métodos de programación, basándose principalmente en el algoritmo “Heapsort”, el cual realiza un ordenamiento en base a permutaciones aplicándose en sentido de la que si es par o no, es decir, si existe que la posición a intercambiar es un número par se intercambiará el primer elemento por el última, mientras que si no es par, el último elemento del arreglo será intercambiado por cada elemento, este último es el que nos permite crear un camino a modo de que se recorrerán los posibles caminos, dando al final el corto. La complejidad de este algoritmo fue de $o(n) = n!$, lo cual nos indica en primera instancia que es una complejidad mala en cuanto al tiempo de ejecución.

Respecto a las limitaciones, al momento de la aplicación de los códigos, al tener complejidades altas, la resolución para todos los archivos .txt no ha sido posible puesto que el computador se demora mucho en resolverlos.

Resultado y análisis

a. Problema 1

Al haber desarrollado el programa, mediante la utilización de time.h, se tabulan los resultados de las soluciones con sus respectos tiempos y se grafican.

Cantidad - Ponderación	Tiempo (s)
6_500	0
8_500	0
10_500	0,001
12_500	0,001
14_500	0,002
16_500	0,005
18_500	0,015
20_500	0,066
24_1000	1,255
28_1000	23,852
32_1000	431,01

Tabla problema 1.

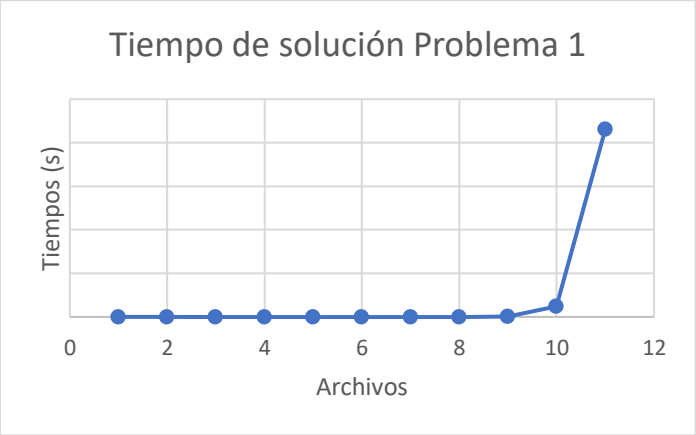

Gráfico Problema 1.

b. Problema 2

Para el segundo problema, se obtuvo los siguientes tiempos que además se graficaron:

Mapa	Tiempo (s)
4_30	0
6_78	0,001
8_249	0,002
10_601	0,120
12_677	18,357

Tabla problema 2.

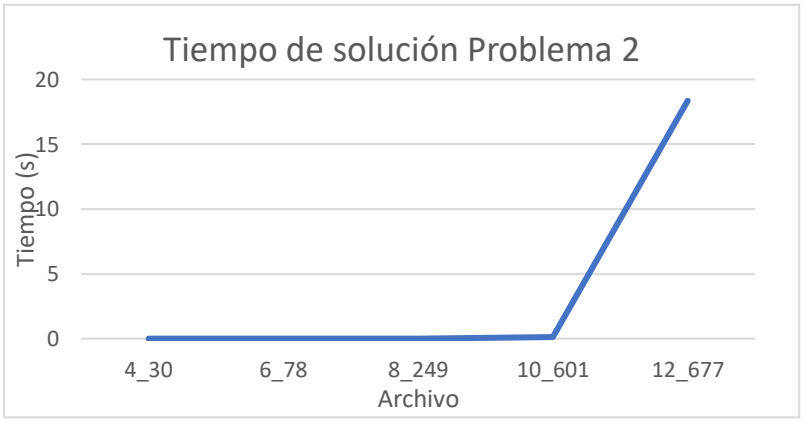

Gráfico Problema 2.

Discusión

Al ver el resultado obtenido, se infiere que a pesar de que al inicio hay un inicio de tiempos bajos, existe un incremente a partir de la décima solución haciendo que la onceava, se dispare en tiempo, indicando de que existe un incremento exponencial en la búsqueda de la solución. Bajo la visión de la *imagen 1* en el apéndice se puede deducir por la forma del gráfico que existe una complejidad parecida a la de $0(n) = 2^n$, eso es parecido en el orden de complejidad obtenido en la creación del pseudocódigo, sin embargo, la valoración de este tipo de código es baja puesto que sus tiempos de ejecución son demasiado altos, por lo que es totalmente mejorable el desarrollo de estos.

En base a los resultados obtenidos, se observa que en este caso, a diferencia de lo visto en el problema 1, el tiempo se inclina casi en su totalidad de manera vertical, de esta manera se intuye en base a la *imagen 1* en el apéndice que se tiene un desarrollo factorial en la búsqueda del camino mas corto, esto se debe a las permutaciones realizadas, mostrando nuevamente una complejidad similar a la vista en el pseudocódigo de $o(n) = n!$. Respecto a la importancia de esto, nos da a entender que es uno de los peores códigos con respecto a tiempo de ejecución existente, donde la resolución incrementa rápidamente entre mas complejo sea el archivo entregado, por lo que a pesar de resolver lo solicitado, si el archivo es grande, puede que se demoré años en resolverlo.

Conclusión

La observación vistas en el desarrollo de la experiencia, si indica que los objetivos que se establecieron al principio de este informe se cumplieron a cabalidad, se logró tabular y graficar los tiempos obtenidos de los archivos que si se pudo calcular sus tiempos, además de obtener sus complejidades respectivas y verificar que se siguió con éxito la creación de los programabas en base a los pseudocódigos, por otro lado, la complejidades obtenidas, no son las mas rápidas en lo que respecta a tiempos de ejecución, tanto para el problema 1 que se obtuvo una complejidad de $0(n) = 2^n$, como para el problema 2 que se calculo que tiene una complejidad de $o(n) = n!$. Esto nos da a entender que la cantidad de pruebas que se pudieron hacer fue baja puesto que los tiempos de ejecución se alargaban mucho de las siguientes. Con toda esta información se concluye que a pesar de lograr resolver la problemática, la complejidad tiene una importancia fundamental a la hora del desarrollo de un software, tomando en cuenta las iteraciones y permutaciones pertinentes para el buen desarrollo del código.

La aplicación de estas problemáticas representó un desafío, en el cual se da cuenta de que los programas creados son mejorables en el ámbito de técnicas de implementación. También nos permite repasar que estas son bases de programación aprendidas anteriormente, como el algoritmos Heapsort estudiado anteriormente, el cual fue retomado para el desarrollo del problema 2.

Apéndice

Se muestra la gráfica de complejidades en apoyo a la comparativa de los gráficos obtenidos.

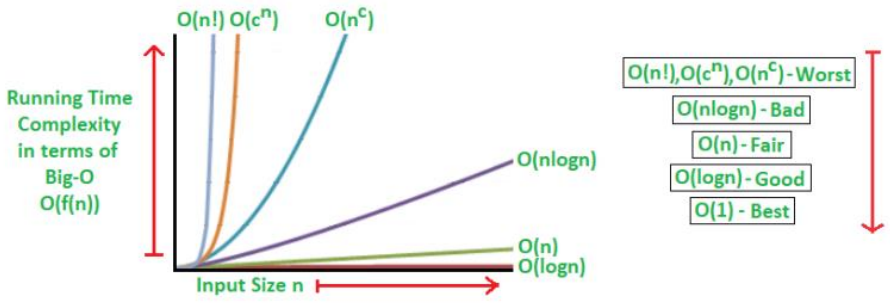

Imagen 1. Complejidades

Referencias

- Martinez, C. (2011). *Algorítmica Heaps y Heapsort*. Algorítmica. Recuperado 13 de abril de 2022, de http://algorithmics.lsi.upc.edu/docs/pqueues.pdf
- *Complejidad (Big-O) - Guías de Make it Real*. (2019). Complejidad (Big-O). Recuperado 12 de abril de 2022, de https://guias.makeitreal.camp/algoritmos/complejidad
- *Introducción al algoritmo de fuerza bruta del lenguaje C - programador clic*. (2018). Introducción al algoritmo de fuerza bruta del lenguaje C. Recuperado 13 de mayo de 2022, de https://programmerclick.com/article/42401809441/