DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Recuperatorio del Trabajo Práctico I

Algoritmos y Estructura de Datos III
Primer Cuatrimestre de 2015

Integrante	LU	Correo electrónico
Iván Arcuschin	678/13	iarcuschin@gmail.com
Martín Jedwabny	885/13	martiniedva@gmail.com
José Massigoge	954/12	jmmassigoge@gmail.com
Lucas Puterman	830/13	lucasputerman@gmail.com

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2160 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (54 11) 4576-3359
http://www.fcen.uba.ar

Índice

1. Ejercicio 2 - Alta frecuencia

1.1. Problema a resolver

Se tienen una lista de frecuencias que transmiten señales, donde cada una tiene un tiempo de inicio y fin y un costo por transmisión. El problema a resolver consiste en dar un algoritmo que minimice los costos, es decir, que para cada unidad de tiempo encuentre la frecuencia con menor costo e indique como ir cambiando de frecuencia acorde pasa el tiempo. A la vez, queremos transmitir todo el tiempo que sea posible.

Tenemos los siguientes parámetros de entrada:

- n = Cantidad de frecuencias
- Luego vienen n líneas con frecuencias, donde F_i representa a la frecuencia en la línea i con $1 \leq i \leq n$ y tiene los siguientes parámetros:
  - c_i = Costo de la frecuencia
  - b_i = Tiempo de inicio de la frecuencia
  - e_i = Tiempo de final de la frecuencia

- Ejemplo: Situación inicial.

```
2
10 1 16
5 6 10
```

  Aquí se presentan dos señales, la primera tiene costo 10, inicio 1 y fin 16. La segunda tiene costo 5, inicio 6 y fin 10.

  En este caso el algoritmo debe elegir la mejor señal para los tiempos 1 al 16.

  En este ejemplo tenemos $n = 2$ y luego n señales:

F_i	c_i	b_i	e_i
F_1	10	1	16
F_2	5	6	10

- Solución del ejemplo:

```
130
1 1 6
2 6 10
1 10 16
```

La solución muestra que lo mejor es usar la primera señal de los tiempos 1 al 6, luego pasar a la segunda del 6 al 10 para, finalmente, retornar a la primer señal del 10 al 16.

El costo total de la transmisión anterior es 130 que es el menor costo posible para el ejemplo.

1.2. Resolución planteada

La solución planteada utiliza la técnica de *Divide and Conquer*. En este caso, se agregan todas las frecuencias recibidas a un vector y luego se llama a una función que se queda con dos problemas de menor complejidad en cada paso. Se llama a la misma recursivamente hasta llegar al caso base y luego se van uniendo las soluciones parciales hasta alcanzar un único vector con la solución final.

Veamos ahora como hacemos para unir dos soluciones parciales. En primer lugar, creo un indice de iteración para cada uno de los dos vectores que contienen los elementos que quiero unir (los llamamos vector izq y der):

```
int izq_it <- 0
int der_it <- 0
```

Mientras ambos iteradores sean menores al tamaño del vector que itera, me fijo cual de los dos elementos a los que apuntan tiene menor principio y llamo a la función auxiliar $mergeAux$ que los irá agregando al resultado según corresponda. Es importante ver cual tiene menor principio porque queremos transmitir todo el tiempo posible.

```
izq y der son Vector<Signal>
    donde Signal es tupla <numero:int, costo:int, principio:int, final:int>
Mientras izq_it < |izq| y der_it < |der| hacer:
    Si izq[izq_it].principio <= der[det_it].principio:
        mergeAux(izq, der, izq_it, der_it, result)
    Sino
        mergeAux(der, izq, der_it, izq_it, result)
    Fin if
Fin ciclo
```

Una vez que sale del ciclo, significa que al menos uno de los vectores fue recorrido completamente, por lo que nos queda agregar los elementos restantes del otro vector al vector solución:

```
Mientras izq_it < |izq| hacer:
    result.agregarAlFinal(izq[izq_it])
    izq_it <- izq_it + 1
Fin ciclo

Mientras der_it < |der| hacer:
    result.agregarAlFinal(der[der_it])
    der_it <- der_it + 1
Fin ciclo
```

Como vimos en el pseudocódigo arriba, la función auxiliar $mergeAux$ necesita comparar dos señales que pertencen una al vector izq y la otra a der. Para hacer esto utiliza los iteradores pasados por parametro.

Pasamos a mostrar cómo la función auxiliar mergeAux compara dos señales dadas. Sean F_1 y F_2 las dos señales pertenecientes a izq y der apuntadas por los iteradores, donde $b_1 < b_2$. Sea también $resultado$ un vector que contendra frecuencias.

Al comparar las dos señales se nos presentan varios casos posibles:

Observacion: Para todos los gráficos siguientes, las señales azules representan las de menor costo y las rojas las de mayor. Además, la que se encuentra más a la izquierda representa a F_1.

- Si $c_1 \leq c_2$:
  - Si $b_2 \geq e_1$, entonces agrego F_1 a $resultado$ y aumento el iterador del vector al que pertenece F_1.

Lo llamaremos caso A

  - Si $b_2 \leq e_1 \wedge e_1 < e_2$. Entonces, agrego F_1 a $resultado$, defino $b_2 = e_1$ y aumento el iterador del vector al que pertenece F_1.

Lo llamaremos caso B

$b_2 = e_1$

  - Si $b_2 \leq e_1 \wedge e_1 \geq e_2$. Entonces desestimo F_2, es decir, aumento el iterador del vector al que pertece F_2.

Lo llamaremos caso C

- Si $c_1 > c_2$:
  - Si $b_2 \leq e_1 \wedge e_1 > e_2$. Entonces defino $F_1' = F_1$ pero con $e_1' = b_2$ y la agrego a $resultado$. Si $e_2 < e_1$, entonces defino $b_1 = e_2$ y no aumento ningún iterador.

Lo llamaremos caso C′

Aquí el primer segmento rojo
es F_1' y el segundo F_1

- Si $b_2 \leq e_1 \wedge e_1 \leq e_2$. Entonces defino $e_1 = b_2$ y agrego F_1 a $resultado$ y aumento el iterador del vector al que pertenece F_1.

Lo llamaremos caso B′

$e_1 = b_2$

- Si $b_2 \geq e_1$. Entonces agrego F_1 a $resultado$ y aumento el iterador del vector al que pertenece F_1.

Lo llamaremos caso A′

Notemos que el único caso donde no se aumenta ninguno de los dos iteradores es el caso C′. Sin embargo, como podemos ver en ese caso, luego de ejecutar ese llamado a $mergeAux$ nos queda nada más y nada menos que el caso A, y aumenta en uno nuestra cantidad de señales.
Por lo tanto al ejecutar un $merge$ entero con una entrada de tamaño k, donde k es la suma de la cantidad de elementos de los dos vectores a unir, necesitamos a lo sumo $2k$ operaciones Y se duplica la cantidad de señales retornadas, que es a lo sumo $2k$.

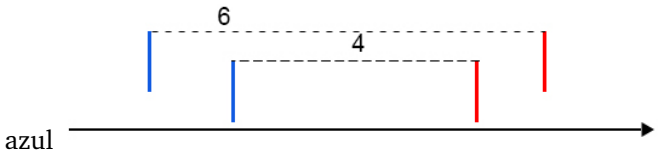
Por lo que podemos decir que $merge$ tiene complejidad lineal en la cantidad de elementos.

Ahora, veamos que si n es la entrada del problema, el vector resultante luego de hacer el algoritmo tendrá a lo sumo tamaño $2n$

Lema 1.1. *Sea n la cantidad de frecuencias recibidas al comienzo del problema. La cantidad de frecuencias resultantes al dar el menor costo de transmisión será a lo sumo $2n$.*

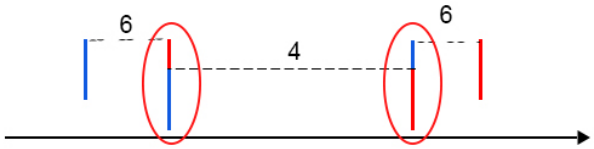
Demostración. Supongamos que puedo graficar en una línea de tiempo todos los t que tienen comienzos y finales de las n señales. De esta forma tendré n comienzos y n finales graficados a lo sumo porque dos señales pueden compartir los mismos comienzos o finales.

Por ejemplo:



Aqui podemos ver 2 frecuencias, por ende 2 comienzos (azul) y 2 finales (rojo)

Como vimos arriba, solo se da en el caso C′ que al comparar dos frecuencias me terminan quedando tres. Pero observemos que siempre que una frecuencia es separada en fragmentos, estos están delimitados por el comienzo o el final de otra frecuencia. Es decir, pondré como final de un fragmento el comienzo de otro, o pondré como comienzo de un fragmento el final de otro.



Aqui podemos ver que aunque aumenta la cantidad de frecuencias, siguen usando los mismos comienzos y finales

Pero como ya dijimos, en una entrada de tamaño n, tengo n comienzos y n finales. Entonces, la cantidad de veces que puedo cortar una frecuencia en dos está acotada por la cantidad de comienzos y finales que yo tenga, y estos son $2n$. □

Por lo tanto, aunque vimos que $merge$ puede retornar para una entrada k algo de a lo sumo tamaño $2k$, donde k es la suma de la cantidad de elementos de los dos vectores a unir. Uno puede pensar que cuando se ejecutan los $merge$ de nuestro algoritmo divide and conquer, se puede duplicar el tamaño de las frecuencias que devuelve en cada paso y aumentar de manera exponencial, esto es falso ya que estas están acotadas por $2n$.

1.3. Complejidad propuesta

Analizamos la complejidad usando Teorema Maestro[1]:

Sean $a > 1$ y $b > 1$ constantes, sea $f(n)$ una función y sea $T(n)$ definido en los enteros no negativos por la recurrencia

$$T(n) = aT(\frac{n}{b}) + f(n)$$

$T(n)$ puede ser acotado asintóticamente de la siguiente manera:

1) Si $f(n) = \mathcal{O}(n^{\log_b a - \epsilon})$ para alguna constante $\epsilon > 0$, entonces, $T(n) = \Theta(n^{\log_b a})$.

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. Introduction to Algorithms. The MIT Press, 2001.

2) Si $f(n) = \mathcal{O}(n^{\log_b a})$ entonces, $T(n) = \Theta(\log_b a \log n)$.

3) Si $f(n) = \Omega(n^{\log_b a+\epsilon})$ para alguna constante $\epsilon > 0$ y si, $af(\frac{n}{b}) \leq cf(n)$ para alguna constante $c < 1$ y todas las n suficientemente grandes, entonces $T(n) = \Omega(f(n))$.

En cada paso nos quedamos con dos subproblemas que tienen la mitad de tamaño que el original, así que $a = 2$ y $c = 2$. Además de la recursión, se unen los resultados. Para esto llamamos a la función merge, que ya vimos que tiene complejidad lineal en la cantidad de elementos del arreglo. Más allá de esto, se hace una cantidad constante de comparaciones y asignaciones, por lo que podemos decir que $f(n) \in \Theta(n)$. No podríamos usar el primer caso del teorema, porque $n^{\log_2 2-\epsilon} = n^{1-\epsilon}$, y no es cierto que $f(n) \in \Theta$ ($n^{1-\epsilon}$). En cambio, podemos usar el segundo caso, porque como ya dijimos $f(n) \in \Theta(n)$. Reemplazando, obtenemos $T(n) \in \Theta$ ($n^{\log_2 2}$ $log(n)$), o sea $T(n) \in \Theta$ (n $log(n)$).

1.4. Implementación en C++

```
#include <utility>
#include <vector>
#include <iostream>
using namespace std;

// Varios typedefs
struct Signal {
    Signal() : numero(0), costo(0), principio(0), fin(0) {};
    Signal(const int n,const int c,const int p, const int f) :
        numero(n), costo(c), principio(p), fin(f) {};

    int numero;
    int costo;
    int principio;
    int fin;
};
typedef vector<Signal> Vec;

// Prototipado de funciones
void mergeSort(vector<Signal>& vec);
void merge(Vec& izq, Vec& der, Vec& result);
void mergeAux(Vec& menor, Vec& mayor, int& i, int& j,Vec& result);
void mostrar(const Vec& v);

// Implementacion
int main() {
    int n;
    cin >> n;

    Vec l;
    l.reserve(n);
    for(int i = 0; i < n; ++i) {
        int j=0;
        int values [3];
        while(j<3) {
            cin >> values[j];
            j++;
        }
        l.push_back(Signal(i+1,values[0],values[1],values[2]));
    }

    mergeSort(l);
    mostrar(l);

    return 0;
}

void mergeSort(vector<Signal>& vec) {
    if(vec.size() == 1) {
        return;
    }
```

```
    Vec::iterator medio = vec.begin() + (vec.size() / 2);

    Vec izq(vec.begin(), medio);
    Vec der(medio, vec.end());
    Vec result;

    /**
    * Se reservan para resultado (2*n)-1 lugares, ya que en peor caso mergeSort
    * devuelve un vector del doble del original menos uno, esto se debe a que se
    * necesitan en peor caso 3 frecuencias en resultado al comparar dos frecuencias.
     */
    result.reserve(2*(izq.size()+der.size())-1); // O(n)

    mergeSort(izq); //T(n/2)
    mergeSort(der); //T(n/2)
    merge(izq, der, result); // O(n)

    vec = result; // O(n)
}

void merge(Vec& izq, Vec& der, Vec& result) {
    int izq_it = 0, der_it = 0;

    while(izq_it < izq.size() && der_it < der.size()) {
        if(izq[izq_it].principio <= der[der_it].principio) {
            mergeAux(izq,der,izq_it,der_it,result);
        } else {
            mergeAux(der,izq,der_it,izq_it,result);
        }
    }

    while(izq_it < izq.size()) {
        result.push_back(izq[izq_it]);
        izq_it++;
    }

    while(der_it < der.size()) {
        result.push_back(der[der_it]);
        der_it++;
    }
}

/**
* Compara dos frecuencias donde primera tiene <= tiempo de inicio que segunda.
* Si es necesario agrega alguna al resultado y aumenta izq_it y der_it.
 */
void mergeAux(Vec& primera, Vec& segunda, int& i, int& j,Vec& result) {
    if(primera[i].costo <= segunda[j].costo) {
        if(segunda[j].principio <= primera[i].fin) {
            segunda[j].principio = primera[i].fin;
        }
        if(segunda[j].principio >= segunda[j].fin) {
            j++;
        }else{
```

```
            result.push_back(primera[i]);
            i++;
        }
    } else {
        int aux = primera[i].fin;
        if(segunda[j].principio <= primera[i].fin) {
            primera[i].fin = segunda[j].principio;
        }
        if(primera[i].fin > primera[i].principio) {
            result.push_back(primera[i]);
        }
        if(segunda[j].fin < aux) {
            primera[i].principio = segunda[j].fin;
            primera[i].fin = aux;
        } else {
            i++;
        }
    }
}

/**
 * Recibe un vector y lo muestra por pantalla
 */
void mostrar(const Vec& v) {
    int n = v.size();
    int finalCost = 0;
    for(int i = 0; i < n; ++i) {
        finalCost += v[i].costo * (v[i].fin - v[i].principio);
    }
    cout << finalCost << endl;

    for(int i = 0; i < n; ++i) {
        cout << v[i].principio;
        cout << "␣" << v[i].fin;
        cout << "␣" << v[i].numero;
        cout << endl;
    }
    cout << -1 << endl;
}
```

1.5. Experimentación computacional

La función que utilizamos para llevar a cabo las mediciones fue `std::clock`[2]. La unidad temporal que utilizamos para este ejercicio fue de nanosegundos. La complejidad teórica calculada es de $\mathcal{O}(n * log(n))$

1.5.1. Experimentación con instancias aleatorias

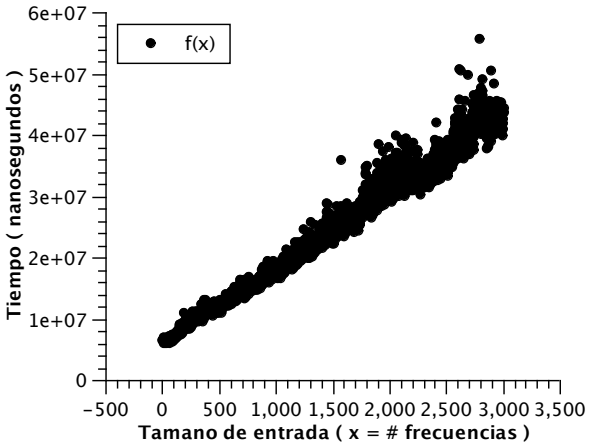
Para generar las instancias aleatorias utilizamos la función `std::rand`[3] con determinados intervalos de valores para la variables, para obtener instancias coherentes. El detalle de intervalos es el siguiente:

- Cantidad de muestras: 100
- Cantidad de frecuencias (n): de 1 a 3000

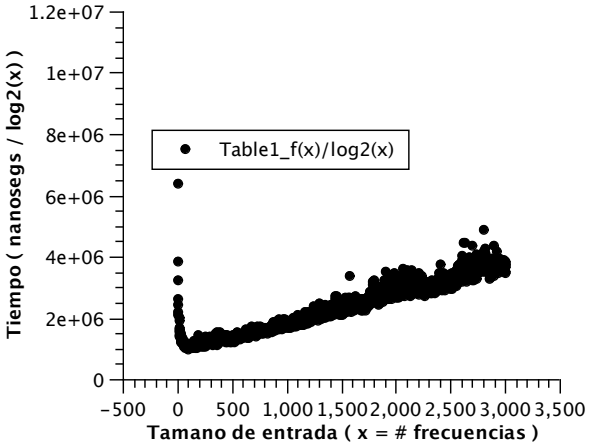
[2]Referencia `http://en.cppreference.com/w/cpp/chrono/c/clock`
[3]Referencia `http://en.cppreference.com/w/cpp/numeric/random/rand`

- Costos (P): $1 \leq P \leq 10.000$
- Para cada frecuencia, se verifico que su tiempo final fuera mayor que su inicio (no se usaron frecuencias invalidas).

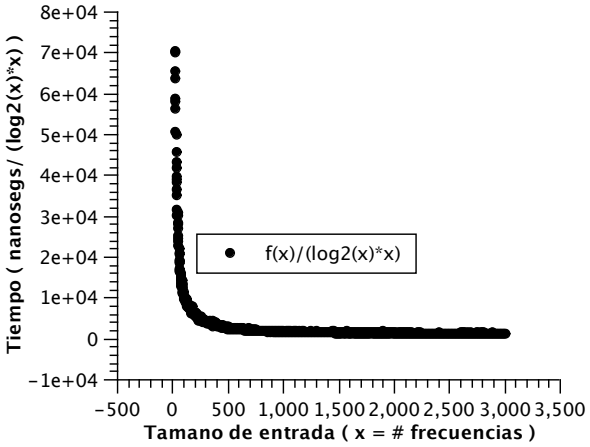
Entonces, para cada n, del 1 al 3000 se generaron 100 instancias aleatorias de tamaño n. Luego, estas fueron promediadas, arrojando los siguientes resultados:



Tiempos sin procesar



Dividiendo a los tiempos por $\log(n)$



Dividiendo a los tiempos por $n \log(n)$

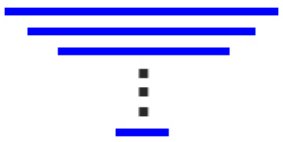
A continuación, adjuntamos una tabla con los ultimos 20 valores obtenidos en este último paso, teniendo en cuenta que los casos fueron previamente ordenados segun el tamaño (n):

Tamano(n)	Tiempo(t)	$t/log(n)$	$t/n*log(n)$
2,980	41,823,717	3,623,894.53913547	1,216.07199299848
2,981	42,205,634	3,656,833.08407709	1,226.713547157695
2,982	41,379,610	3,585,113.377975757	1,202.251300461354
2,983	42,125,330	3,649,569.318887776	1,223.456023763921
2,984	45,341,672	3,928,055.705472136	1,316.372555453129
2,985	40,887,769	3,542,055.278169351	1,186.618183641324
2,986	42,795,296	3,707,146.720857472	1,241.509283609334
2,987	43,971,377	3,808,865.466588621	1,275.147461194717
2,988	41,832,598	3,623,449.707752226	1,212.667238203556
2,989	41,938,501	3,632,470.907368039	1,215.279661213797
2,990	41,182,479	3,566,839.555032387	1,192.922928104477
2,991	43,525,502	3,769,612.695656072	1,260.318520781034
2,992	42,113,000	3,647,127.818332012	1,218.959832330218
2,993	42,931,618	3,717,867.670108657	1,242.187661245792
2,994	43,479,759	3,765,179.403366345	1,257.574951024163
2,995	42,626,427	3,691,130.148415261	1,232.430767417449
2,996	42,621,659	3,690,563.361166947	1,231.830227358794
2,997	40,046,500	3,467,438.573756513	1,156.969827746584
2,998	40,993,661	3,549,300.88989128	1,183.88955633465
2,999	44,504,732	3,853,134.875349259	1,284.806560636632
3,000	43,832,216	3,794,751.699991118	1,264.917233330373

A partir de la información suministrada, podemos observar que, en el primer gráfico las mediciones tienden a algo un poco más grande que lineal. Al dividir los tiempos por logaritmo de n (segundo gráfico), se observa que tienden a ser algo lineal. Por último, en el último gráfico, se dividen además de por logaritmo de n, por n y el gráfico que arroja es constante y mayor a 0. Por lo que podemos concluir que la complejidad de $\mathcal{O}(n*log(n))$ se condice con nuestra predicción de complejidad.

1.5.2. Experimentación con instancias particulares

Como quedo demostrado en la justificación de la complejidad teórica, el peor caso ocurre cuando al comparar dos frecuencias la de menor costo esta completamente contenida dentro de la de mayor costo. En este caso, la salida termina teniendo $2n - 1$ señales.

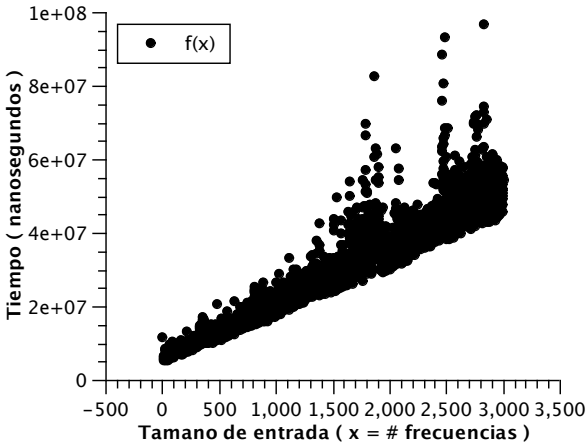


(a) Aquí supongamos que cada señal tiene costo menor que la de arriba

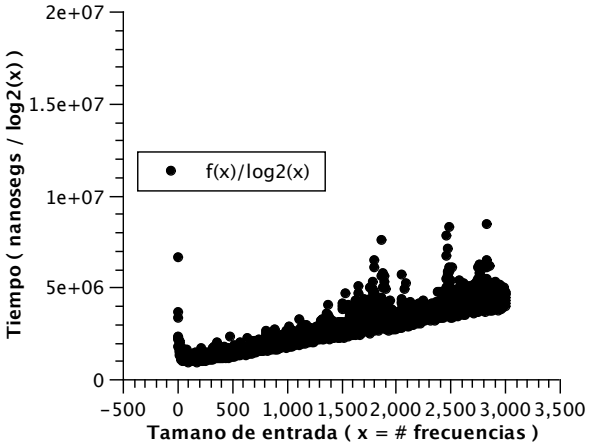
Generamos una muestra de peor caso de tamaños 1 a 3000 de la siguiente manera:

- El costo de la primer frecuencia es 3001
- El comienzo de la primer frecuencia es 1
- El final de la primer frecuencia es 6001

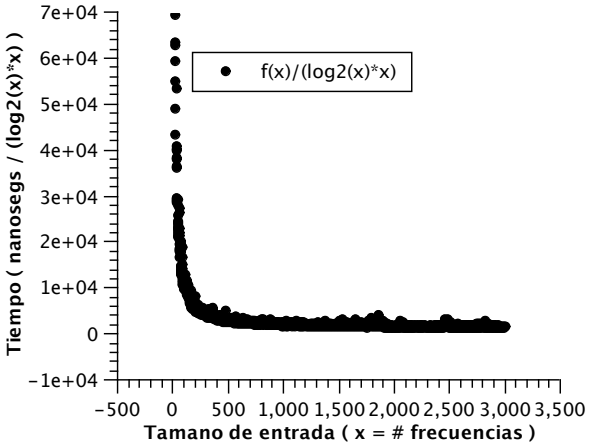
- Cada frecuencia que se agrega disminuye en 1 el costo, aumenta en 1 el comienzo y disminuye en 1 el final



Tiempos sin procesar



Dividiendo a los tiempos por $\log(n)$



Dividiendo a los tiempos por $n \log(n)$

A continuación, adjuntamos una tabla con los ultimos 20 valores obtenidos en este último paso, teniendo en cuenta que los casos fueron previamente ordenados segun el tamaño (n):

Tamano(n)	Tiempo(t)	$t/log(n)$	$t/n*log(n)$
2,980	47,188,711	4,088,754.524179232	1,372.065276570212
2,981	48,174,802	4,174,021.169127874	1,400.208376091202
2,982	47,537,404	4,118,622.264314194	1,381.161054431319
2,983	50,090,561	4,339,644.926021389	1,454.792130748035
2,984	50,262,066	4,354,321.012249329	1,459.222859332885
2,985	50,960,628	4,414,654.205912404	1,478.946132633971
2,986	51,054,247	4,422,579.162716796	1,481.104876998257
2,987	58,162,121	5,038,088.62161512	1,686.671784939779
2,988	53,436,807	4,628,581.344801059	1,549.056674966887
2,989	57,760,600	5,002,889.805053413	1,673.767080981403
2,990	50,334,133	4,359,469.874376941	1,458.016680393625
2,991	52,002,106	4,503,745.849466659	1,505.765914231581
2,992	55,619,141	4,816,805.175903654	1,609.894778042665
2,993	53,321,719	4,617,647.88796729	1,542.815866343899
2,994	49,501,380	4,286,628.55316679	1,431.739663716362
2,995	48,025,112	4,158,615.939924299	1,388.519512495592
2,996	51,954,186	4,498,656.781775029	1,501.554333035724
2,997	51,590,387	4,466,969.595815528	1,490.48034561746
2,998	45,924,717	3,976,240.104930008	1,326.297566687795
2,999	54,525,123	4,720,681.230346652	1,574.085105150601

A partir de la información suministrada, podemos observar que, en el primer gráfico las mediciones tienden a algo un poco más grande que lineal. Al dividir los tiempor por $\log(n)$ (segundo gráfico) se observa que el gráfico tiende a ser lineal. Por último, al dividir los tiempos por $n\log(n)$, el gráfico que arroja es constante y mayor a 0. Aunque este es peor caso, los gráficos siguen mostrando que se cumple la complejidad teórica calculada. Por lo que podemos concluir que la complejidad de $\mathcal{O}(n*log(n))$ se condice con nuestra predicción de complejidad.

1.6. Adicionales

Debido a un aumento en la demanda de nuestro servicio de transmisión de información, hemos duplicado nuestro caudal de datos a transmitir y por tal motivo podemos ahora utilizar dos frecuencias en paralelo en lugar de una sola. Quisiéramos resolver el mismo problema que antes pero ahora queremos transmitir siempre que sea posible en dos frecuencias. En los momentos en que esto no sea posible, seguiremos transmitiendo por una sola frecuencia, y obviamente en los momentos en los que esto último no se pueda no transmitiremos nada. Es decir, el objetivo es transmitir información durante todo el tiempo que sea posible y en la mayor cantidad de frecuencias en paralelo hasta un máximo de dos frecuencias, y hacer esto invirtiendo la menor cantidad de dinero. Se pide desarrollar los siguientes puntos:

1. *Dar una idea de qué cambiaría en su algoritmo para resolver este nuevo problema.*

   Podríamos resolver este problema modificando el algoritmo que ya tenemos de la siguiente forma:

   Ejecutamos el algoritmo normalmente, pero guardando una copia extra del vector original que contiene a las señales, llamémosla $copia$.

   Una vez que se termina de ejecutar el algoritmo original, tendremos en el vector $resultado$ la opción de menor costo para la transmisión de las señales.

   Ahora, lo que se debe hacer es recorrer el vector $resultado$ y, por cada señal en $resultado$, buscar en el vector $copia$ la misma o el fragmento de la misma correspondiente y eliminarla.

Esta tarea la haremos de la siguiente manera. Llamemos R_i a la frecuencia en la posición i de $resultado$. Buscamos en $copia$ la frecuencia correpondiente a R_i, llamemosla C_{R_i}. Actuaremos de la siguiente forma:

Recordemos que dada una frecuencia F_i, c_i, b_i y e_i representan su costo, tiempo de comienzo y tiempo de fin respectivamente.

- Si $b_i = b_{R_i} \wedge e_i = e_{R_i}$ entonces esta señal se usa completamente, por lo que definimos $c_{R_i} = e_{R_i} = b_{R_i} = -1$.
- Si $b_i > b_{R_i} \wedge e_i < e_{R_i}$ entonces me quedan dos fragmentos que no fueron usados, por lo que agrego al final de $copia$ una frecuencia F con $c = c_{R_i}$, $b = b_{R_i}$ y $e = e_i$. Luego modifico a C_{R_i} definiendole $b_{R_i} = b_i$.
- Si $b_i > b_{R_i} \wedge e_i = e_{R_i}$ o $b_i = b_{R_i} \wedge e_i < e_{R_i}$ entonces modifico $b_{R_i} = b_i$ o $e_{R_i} = e_i$ según sea necesario.

Una vez finalizado esto, podríamos tener frecuencias que tengan todos sus atributos en -1 (primer caso arriba), por lo que debemos crear otro vector, llamemoslo v y lo llenaremos con los elementos de $copia$ que no tengan todo en -1. Luego, ejecutamos nuevamente el algoritmo de $mergeSort$ que usamos para resolver el ejercicio original sobre el vector v, obteniendo así las segundas mejores señales.

2. *¿Cómo afectaría este cambio en la complejidad temporal de su algoritmo?*

   Veamos: el agregado que le hacemos a nuestro algoritmo se divide en tres partes, una es crear el vector $copia$, la segunda es seleccionar las señales en $resultado$ y eliminar los fragmentos correspondientes en $copia$ y la tercera es volver a correr el algoritmo $mergeSort$ original sobre $copia$.

   Crear $copia$ no modifica nuestra complejidad temporal ya que lo hacemos al mismo tiempo que creamos el vector original.

   Sabemos que $resultado$ tiene tamaño de a lo sumo $2n-1$. Deberíamos recorrerlo entero y, por cada frecuencia, buscar el correspondiente en $copia$. Esto parece tener complejidad $\mathcal{O}(n^2)$. Sin embargo, podemos hacer algunos cambios en nuestra estructura para poder hacer esto de manera más eficiente.

   Actualmente tenemos la siguiente estructura $Signal$ en la implementación:

```
struct Signal {
    Signal() : numero(0), costo(0), principio(0), fin(0) {};
    Signal(const int n,const int c,const int p, const int f) : numero(n),
    costo(c), principio(p), fin(f) {};

    int numero;
    int costo;
    int principio;
    int fin;
};
```

   Notemos que estamos guardando el número de señal, que es ni más ni menos que el orden con que la señal fue guardada tanto en el vector original como en $copia$. Por esto, cada vez que ejecutando el algoritmo tengamos que buscar una señal de $resultado$ en $copia$, podemos hacerlo en $\mathcal{O}(1)$. Luego tenemos que modificarlo según sea necesario, esto lo hacemos también en $\mathcal{O}(1)$. Por lo que todo el proceso de recorrer $resultado$ y modificar $copia$ podemos hacerlo en $\mathcal{O}(2n) \in \mathcal{O}(n)$ porque vimos que $resultado$ tiene su tamaño acotado por $2n$.

Ahora tenemos que crear el vector v y llenarlo con las señales de $copia$ que cumplen las condiciones que vimos en el punto anterior. Esto esta acotado por el tamaño de $copia$ luego de modificarlo.

Veamos cual es el tamaño de $copia$ viendo cuales son los tres casos posibles al comparar una señal de $resultado$ con una de copia:

Llamemos R_i a la frecuencia en la posición i de resultado. Buscamos en $copia$ la frecuencia correspondiente a R_i, llamemosla C_{R_i}. Actuaremos de la siguiente forma:

Recordemos que dada una frecuencia F_i, c_i, b_i y e_i representan su costo, tiempo de comienzo y tiempo de fin respectivamente.

- Si $b_i = b_{R_i} \wedge e_i = e_{R_i}$ entonces esta señal se usa completamente, por lo que definimos $c_{R_i} = e_{R_i} = b_{R_i} = -1$.

  En este caso, el tamaño de $copia$ se mantiene igual, ya que no se agregan fragmentos.

- Si $b_i > b_{R_i} \wedge e_i < e_{R_i}$ entonces me quedan dos fragmentos que no fueron usados, por lo que agrego al final de $copia$ una frecuencia F con $c = c_{R_i}$, $b = b_{R_i}$ y $e = e_i$. Luego modifico a C_{R_i} definiendole $b_{R_i} = b_i$.

  En este caso, el tamaño de $copia$ aumenta en uno.
  Recordemos que $resultado$ está ordenado cronologicamente por lo tanto mientras lo vaya recorriendo, cada frecuencia R_i tendrá $b_i \geq e_{i-1}$. Entonces, al partir las frecuencias, la que agrego al final de $copia$ tiene que ser la que tenga el menor final ya que nunca volverá a ser buscada, mientras que la otra debe permanecer en su lugar, para que si tiene que volver a ser accedida en el futuro, esto se logre en $\mathcal{O}(1)$.

- Si $b_i > b_{R_i} \wedge e_i = e_{R_i}$ o $b_i = b_{R_i} \wedge e_i < e_{R_i}$ entonces modifico $b_{R_i} = b_i$ o $e_{R_i} = e_i$ según sea necesario.

  En este caso, el tamaño de $copia$ se mantiene igual, ya que no se agregan fragmentos.

Entonces, como podemos ver, el tamaño de $copia$ aumenta a lo sumo en uno por cada iteración. Esto se ejecuta tantas veces como elementos tenga $resultado$ y ya vimos que el tamaño de resultado está acotado por $2n$ por lo que $copia$ finaliza con a lo sumo $3n$ elementos. Por lo tanto, crear v nos cuesta $\mathcal{O}(3n) \in \mathcal{O}(n)$

Por último, solo tenemos que correr nuestro $mergeSort$ con entrada v, como ya demostramos en el ejercicio original esto tiene complejidad $\mathcal{O}(n \log(n))$. Entonces con v quedaría $\mathcal{O}(3n \log(3n)) \in \mathcal{O}(n \log(n))$.

Por lo tanto, no se modifica la complejidad del algoritmo, ya que esta termina siendo igualmente de:

$$\mathcal{O}(n \log(n))$$

1.7. Informe de modificaciones

- Mejoras en descripción del problema.
- La resolución del problema fue modificada completamente.
- Aclaraciones sobre Teorema Maestro en la justificación de la complejidad.

- Aclaraciones sobre como fueron tomadas las muestras.
- Agregados los adicionales