Entrega Proyecto 2

Análisis de Algoritmos
Primer Semestre 2024, Prof. Cecilia Hernández
Ayudantes: Martita Muñoz, Álvaro Guzmán

Integrantes: Diego Venegas Anabalon, Francy Jelvez Jen

Fecha Inicio: Domingo 2 de junio 2024.

Fecha Entrega: Lunes 24 de junio 2024 (23:59 hrs).

Instrucciones de entrega:

- Se debe entregar (a través de Canvas) el informe con sus respuestas en formato pdf y un .zip con los archivos o programas que utilice para sus respuestas. Es decir, en su entrega de Canvas deberá aparecer un archivo pdf (informe) y un archivo zip con los códigos o material que referencie en su informe.
- El informe debe estar escrito en LaTeX.
- Sólo una persona del grupo debe enviar la entrega por Canvas, procurando que los integrantes del grupo estén detallados en su informe.
- No se aceptan grupos de una persona.
- El código fuente deberá ir en el .zip que entregue y deberá estar escrito en C/C++.

Pregunta 1 [1.2 puntos]: Considere un sistema de cloud computing que tiene disponibilidad de N unidades de procesamiento p_i donde cada una tiene una capacidad c_i para atender un requerimiento. Por otro lado, existen N usuarios, u_i donde cada uno de ellos solicita un requerimiento que requiere r_i capacidad de procesamiento. Asuma que el número de unidades de procesamiento es la misma que el número de usuarios, pero no puede preprocesar la información otorgada para las unidades de procesamiento y usuarios.

Ejemplo: Entradas: $N = 7$, arreglo P que contiene capacidades de procesamiento de cada unidad, arreglo U que contiene los requerimientos de cada usuario. Salida: arreglo A_U que indica para cada usuario el identificador de la unidad de procesamiento que lo debe atender.

$$P = [8, 20, 10, 3, 50, 2, 5]$$

$$U = [10, 2, 50, 8, 20, 5, 3]$$

Asignación de usuarios a unidades de procesamiento:

$$A_U = [2, 5, 4, 0, 1, 6, 3]$$

1. Escriba un algoritmo aleatorizado que resuelve el problema de asignación de requerimientos de usuario en unidades de procesamiento.

2. Proporcione el análisis de tiempo esperado.

3. Proporcione la implementación, evaluación y discusión.

Solución:

1. Pseudocódigo:

```
Permutación(P,U,A, n)
    for i to n do
        A[i] <- i
    endfor
    correcto = false
    while correcto := false
        for i to n do
            j <- random(i,n)
            intercambiar A[i] y A[j]
        endfor
        correcto := true
        for i to n do
            if P[A[i]] < U[i] then
            correcto <- false
            endif
        endfor
```

2. Primero, hagamos un breve análisis del código:
   - En el primer ciclo asignamos números del 0 a N-1 a un arreglo de tamaño N, por lo que tenemos un coste de O(N).
   - Seguido, entramos al while, y en este, hacemos una permutación aleatorizada inplace (vista en clases), la cual tiene un coste de O(N).
   - En el mismo while, verificamos con otro ciclo que cada elemento u_i tenga un valor valido de p_i, lo que tiene un coste O(N).

   A continuación, consideraremos p (diferente de p_i) como la probabilidad de obtener una permutacion valida, por lo que $\frac{1}{p}$ será la cantidad de intentos que necesitaremos para obtener al menos un caso correcto. De esta forma podemos plantear el tiempo esperado como cantidad de intentos por el tiempo que tarda la permutación:

   $$E(T) = \frac{1}{p} \cdot O(N)$$

   Teniendo en cuenta esto, nos podemos percatar que el tiempo que tome este algoritmo Las Vegas, dependerá de que tan restringidos esten P y U. ¿A que nos referimos con restrigidos? Dado que P y U no necesariamente tienen que tener los mismos elementos (como en el ejemplo), podemos tener casos en que exista más de una solución correcta, lo que haria que la probabilidad de llegar a una permutacion valida tambien aumente.

   En un caso restringido, con k soluciones, tendremos que la probablididad de llegar a una de estas soluciones sera de $\frac{k}{N!}$, por lo que tendriamos un tiempo esperado de $E(T) = \frac{1}{\frac{k}{N!}} \cdot O(N) = \frac{N!}{k} \cdot O(N)$.

   Este resultado puede parecer alarmante, sobre todo en casos en los que N es muy grande, pero si consideramos que k tambien puedes ser muy grande, el tiempo esperado puede verse reducido en unos pocos intentos de permutaciones, lo que cambiaria nuestra esperanza a $E(T) = O(N)$

3. Implementación en C++ con el ejemplo del ejercicio:

```
int* LasVegas(int N, int* P, int* U){
    // Setear el arreglo de 0 hasta N-1. O(n).
    int* A = new int[N];
    for(int i = 0 ; i < N ; i++){
        A[i] = i;
    }
    bool correcto = false;
    while(!correcto){
        // Algoritmo aleatorizado para permutar inplace. O(n).
        for(int i = 0 ; i < N ; i++){
            // Un numero random del i a N-1.
            int j = rand() % (N - i) + i;
            int temp = A[i];
            A[i] = A[j];
            A[j] = temp;
        }
        // Verificar si la permutacion es correcta. O(n).
        correcto = true;
        for(int i = 0 ; i < N ; i++){
            if(P[A[i]] < U[i]){
                correcto = false;
                break;
            }
        }
    }
    return A;
}
Evaluación: O(n) + O(n) + O(n) = O(n). Para más detalle, ver el análisis del item 2.
```

Nuestra perspectiva de este algoritmo es que si bien es un método que si o si te llevará a una solución, gracias al planteamiento del problema, pueden haber mejores maneras de encontrar una solución. Si se pudieran preprocesar los arrays P y U, se podría correr un algoritmo de Sorting e inmediatamente se encontraría cuales preprocesadores pertenecen a cuales usuarios. O se podría marcar cuáles permutaciones ya se han hecho para evitar la repetición. Pero como algoritmo estocástico, sabemos que teóricamente puede llegar a una solución.

A continuación, unos resultados de algunos experimentos de tiempos, los cuales consideran solamente una permutación correcta:

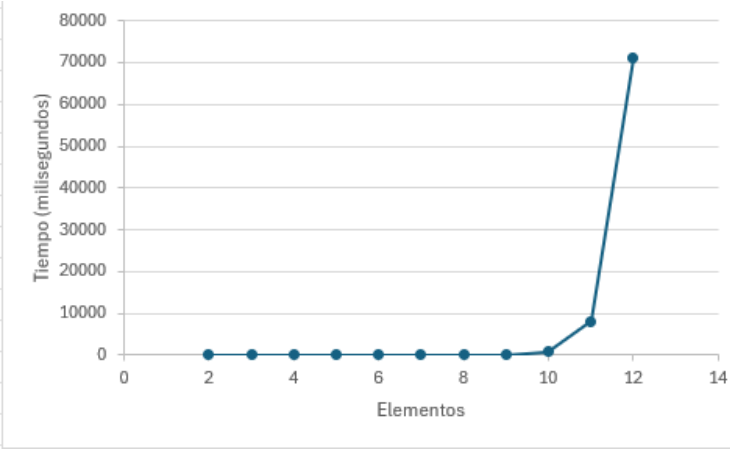

Figura 1: Gráfico de los experimentos

Tamaño	Tiempo (milisegundos)
2	0
3	0
4	0
5	0
6	0
7	0
8	6
9	36
10	766
11	8098
12	70934

Figura 2: Tabla de los experimentos

Los resultados son los esperados, puesto que la probabilidad de lograr la permutacion correcta decrece en orden factorial.

Pregunta 2 [1.2 puntos]: Considere un grupo de n estudiantes universitarios. Se desea construir un arreglo que contenga los promedios de notas de los n estudiantes y que soporte las operaciones de inserción y de búsqueda, para ello se puede mantener un arreglo ordenado, de modo que la búsqueda tomará $O(\log n)$ y la inserción, $O(n)$.

Si bien la búsqueda es rápida, la inserción no lo es tanto. Se le pide diseñar una nueva estructura de modo que la inserción sea más rápida, para ello siga la siguiente estrategia:

i) Se definen los arreglos **ordenados** $A_0, A_1, ..., A_{k-1}$ de modo que A_i es de tamaño 2^i, con $i = 0, 1, ..., k-1$.

ii) Sea $\langle n_{k-1}, n_{k-2}, ..., n_0 \rangle$ la representación binaria de n. Es decir, $n = \sum_{i=0}^{k-1} 2^i n_i$.

iii) Los elementos (los n promedios) se particionan en los arreglos $A_0, A_1, ..., A_{k-1}$ de modo que A_i estará lleno si $n_i = 1$, y estará vacío si $n_i = 0$.

iv) Si bien cada arreglo A_i está ordenado, ellos no tienen ninguna relación entre si.

A modo de ejemplo, observe en la Figura 3 como se puede visualizar esta estructura de datos al agregar los elementos de una lista de promedios dada por [2.7, 4.9, 7, 5.56].

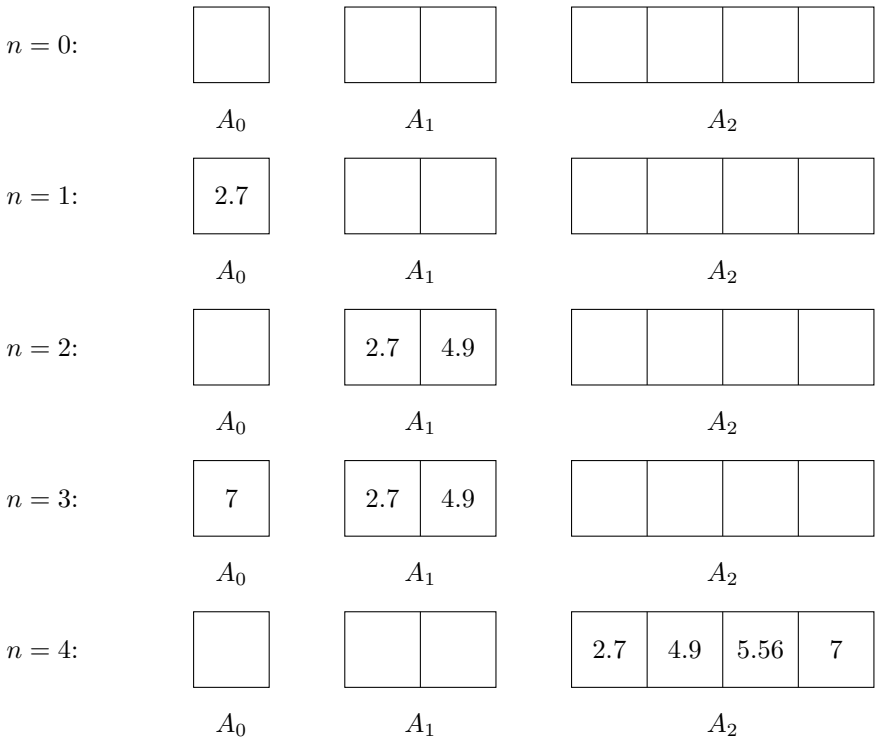

Figura 3: Nueva estructura de datos. Notar que los arreglos A_i vacíos y llenos se corresponden con la representación binaria de n en cada caso, según lo establecido en la parte iii) de la estrategia. Además, estos arreglos siempre están ordenados. Se utilizó $k = 2$ a modo de ilustrar la estructura, pero en general el valor de k depende de n.

Se pide realizar lo siguiente:

1. Dado $n \in \mathbb{N}$, encontrar el menor valor de $k \in \mathbb{N}$ de modo que los n promedios se puedan particionar en los k arreglos siguiendo la estrategia anterior. Justifique. Utilice este valor de k para el resto del problema.

2. Muestre, justificadamente, que la suma de las cantidades de elementos en los k arreglos es igual a n.

3. Definir una operación Buscar para esta estructura de datos (los k arreglos) de modo que el tiempo de ejecución en el peor caso sea de $O(\log^2 n)$. Justifique.

4. Definir una operación Insertar para esta estructura de datos (los k arreglos) de modo que el tiempo de ejecución en el peor caso sea de $O(n)$, pero que el costo amortizado de esta sea de $O(\log n)$. Muestre la complejidad amortizada para M inserciones usando el método de contabilidad. (**Pista:** Puede suponer que unir dos arreglos ordenados de tamaño m toma un tiempo de $2m$). Notar que la operación Insertar debe insertar el nuevo elemento de modo que se mantenga la estructura, es decir, tener k arreglos $A_0, A_1, ..., A_{k-1}$ de modo que todos ellos estén ordenados y estén llenos o vacíos según el punto iii) descrito en la estrategia (puede ver la Figura 3 para guiarse).

5. Escriba un programa que genere listas de tamaño n de promedios (números reales entre 1.0 y 7.0) y utilícelo para validar sus resultados teóricos respecto a las operaciones Buscar y Insertar del item 3. y 4. de forma experimental. Utilice tamaños suficientemente grandes (n=1000, 2000, ...) y obtenga al menos 10 mediciones por experimento. Grafique y discuta sus resultados.

Solución:

1. Dada la representación binaria de n del punto ii $n = \sum_{i=0}^{k-1} 2^i n_i$, donde el k-1 es la cantidad de Arrays y n_i sólo puede tomar valores de 1 o 0. Como queremos buscar el menor k necesario para abarcar n promedios maximizaremos n_i, para asi obtener el valor cubierto por los arreglos, de la siguiente forma:

$$n = \sum_{i=0}^{k-1} 2^i 1 = \sum_{i=0}^{k-1} 2^i = 1 + \sum_{i=1}^{k-1} 2^i = 1 + 2(2^{k-1} - 1) = -1 + 2^k$$

$$\iff n + 1 = 2^k$$

$$\iff log(n+1) = k$$

   Lo que significa que para k arreglos podremos almacenar un total de $log(n+1)$ elementos, pero ¿Que ocurre cuando tomamos una cantidad de elementos diferente de $2^k - 1$? En estos casos, con la anterior formula obtendremos que necesitamos un valor k fraccionario, lo que no es posible, por lo que aproximaremos por exceso a su valor entero mas cercano, lo que tambien es conocido como función techo. De esta forma, tendremos que el minimo k necesario para n promedios es:

$$k = \lceil log(n+1) \rceil$$

2. Sabemos que la representación binaria es capaz de representar cualquier número del 0 hasta el infinito, y en este caso no es la excepción, dado que nuestra estrategia sigue la regla $n = \sum_{i=0}^{k-1} 2^i n_i$, en donde n es la cantidad de elementos almacenados en nuestra estructura, k-1 es la cantidad total de arreglos en nuestra estructura, 2^i la cantidad máxima almacenada por el i-esimo arreglo y n_i representa si el i-esimo arreglo esta en uso con un 1 y con un 0 el caso contrario. Ademas, anteriormente obtuvimos que la cantidad k de arreglos necesarios para almacenar n elementos esta dada por $k = \lceil log(n+1) \rceil$, de lo que ademas, se pudo concluir que solo en los casos en que tenemos una cantidad de elementos igual $2^k - 1$ se estara ocupando la capacidad máxima actual de la estructura.

   Usando inducción matemática, y la sumatoria del punto ii), podemos demostrar el primer principio, con n = 1:

$$n = \sum_{i=0}^{\lceil log(n+1) \rceil - 1} 2^i n_i \iff n = \sum_{i=0}^{\lceil log(2) \rceil - 1} 2^i n_i \iff n = \sum_{i=0}^{0} 2^i n_i \iff n = 2^0 n_0$$

   Sabemos que solo hay espacio para un elemento en el primer arreglo, por lo que n_0 obligatoriamente tendra valor 1, por lo que queda demostrado el caso base.

Se supondra que la proposición $n = \sum_{i=0}^{k-1} 2^i n_i$ es correcta. A continuación, se demostrara que la proposición anterior es válida para n+1 elementos:

Caso sin acarreo:

$$n+1 = \sum_{i=0}^{k-1} 2^i n_{1i}$$

Nos fijamos que no hay un cambio que afecte a la cantidad k de arrays, si no que afecta a los n_i, puesto que ocurren mini acarreos, que cambian los valores de n_i.

Supongamos que n = 5. Su representación binaria es 101 donde:

$$n_2 = 1, n_1 = 0 y n_0 = 1$$

Comparamos con su sucesor, n + 1 = 6, que es 110 en binario :

$$n_2 = 1, n_1 = 1, n_0 = 0$$

La manera de calcular estos dos numeros con la sumatoria entregada es:

$$n = 5 = \sum_{i=0}^{(log(n+1)-1} 2^i n_i \Rightarrow \sum_{i=0}^{2} 2^i n_i \Rightarrow 2^0 n_0 + 2^1 n_1 + 2^2 n_2 \Rightarrow 1 \cdot 1 + 2 \cdot 0 + 4 \cdot 1 = 5$$

$$n = 6 = \sum_{i=0}^{(log(n+1)-1} 2^i n_i \Rightarrow \sum_{i=0}^{2} 2^i n_i \Rightarrow 2^0 n_0 + 2^1 n_1 + 2^2 n_2 \Rightarrow 1 \cdot 0 + 2 \cdot 1 + 4 \cdot 1 = 6$$

Esto demuestra algo.

Caso con acarreo:

$$n+1 = \sum_{i=0}^{(k+1)-1} 2^i n_i$$

$$\iff n+1 = \sum_{i=0}^{k} 2^i n_i$$

$$\iff n+1 = \sum_{i=0}^{k-1} 2^i n_i + \sum_{i=k}^{k} 2^i n_i$$

Como bien sabemos, la representación binaria produce acarreo, que es cuando tenemos un número tal que 011...111, y lo incrementamos en uno, obteniendo 100...000, lo que en nuestro caso hace que $\sum_{i=k}^{k} 2^i n_i = 0$, puesto que en i = 0, 1,..., k-1 tendremos $n_i = 0$, mientras que n_k se hace igual a 1.

$$n+1 = \sum_{i=0}^{k-1} 2^i n_i + \sum_{i=k}^{k} 2^i n_i$$

$$\iff n+1 = 0 + \sum_{i=k}^{k} 2^i n_i$$

$$\iff n+1 = 2^k n_k$$

$$\iff n+1 = 2^k \cdot 1$$

$$\iff n+1 = 2^k$$

$$\iff log(n+1) = k$$

3. Realizamos una implementación de la estructura descrita anteriormente, la cual se encuentre en el archivo zip adjunto. En esta implementación no se utilizaron n_i y k como tal, pero la insercion sigue la misma logica que se pide, a continuación el código de la función Buscar:

```
// Esto es una Búsqueda Binaria recursiva común y corriente.
    double binarySearch(double* array, int size, double x) {
    int left = 0;
    int right = size - 1;
    while (left <= right) {
        int mid = left + (right - left) / 2;
        if (array[mid] == x) {
            return mid;
        }
        if (array[mid] < x) {
            left = mid + 1;
        } else {
            right = mid - 1;
        }
    }
    return -1;
    }

    // Retorno: (i, j) con i es el indice del array y j es el indice del elemento en el array.
    // x: valor a buscar.
    pair<int, int> Buscar(double x) {
        int i = 0;
        while (i < NumArrays) { //NumArrays almacena los arrays que hay en el BinomialArray.
            int iarray = binarySearch(Arrays[i], sizes[i], x); // Realizamos una busqueda binaria en el array i.
            if (iarray != -1) {
                return {i, iarray};
            }
            i++;
        }
        return {-1, -1};
    }
};
```

Nuestra operación Buscar da, en el peor caso, un tiempo de $O(log^2(n))$.

Primero, vemos que esta compuesta por dos funciones: Buscar (la principal) y binarySearch (se usa en la operación buscar). binarySeach, es la búsqueda binaria recursiva típica, la cual, como bien sabemos, toma tiempo O(n) (En este caso, como mucho se hará búsqueda binaria a un arreglo con los $\lfloor \frac{n}{2} \rfloor + 1$ elementos del Binomial Array, por lo que igual es equivalente O(n)). mientras que la función buscar ejecuta k veces binarySearch, y como vimos anteriormente $k = \lceil log(n+1) \rceil$, por lo que se realizan log(n+1) binarySearch.

Por esto, el algoritmo Buscar ejecuta log(n) veces un algoritmo de tiempo de ejecución O(log(n)), osea, que su tiempo de ejecución en el peor caso será $O(log^2(n))$.

4. Para comprender de mejor manera como funciona nuestra inserción, primero se mostraran los elementos que ocuparemos en nuestra estructura de datos, la cual apodamos Binomial Arrays, por una estructura con funcionamiento similar que anteriormente llegamos a trabajar, llamada Binomial Heaps. Además, se debe tener en consideración que los arreglos tienen sus elementos ordenados de menor a mayor, mientras los arreglos estan ordenados acorde a su tamaño, no a lo que almacenan, por lo que no se puede establecer un algoritmo como búsqueda binaria para estos.

```
#include <vector>
#include <algorithm>

struct BinomialArray {
    vector<double*> Arrays; // Vector arrays con promedios.
    vector<int> sizes;  // Tamaños de los arrays.
    int NumArrays; // Número de arrays.
    int size; // Número de elementos.

    BinomialArray() {
        Arrays = vector<double*>();
        sizes = vector<int>();
        NumArrays = 0;
        size = 0;
    }

    void insertar(double promedio) {
    size++;
    double* temp = new double();
    temp[0] = promedio;
    Arrays.push_back(temp);
    sizes.push_back(1);
    NumArrays++;
    for (int i = NumArrays - 1 ; i > 0 ; --i) {
        if (sizes[i] == sizes[i - 1]) {
            double* temp = new double[sizes[i] * 2];
            merge(Arrays[i], Arrays[i] + sizes[i], Arrays[i - 1], Arrays[i - 1] + sizes[i - 1], temp);
            delete[] Arrays[i];
            delete[] Arrays[i - 1];
            Arrays[i - 1] = temp;
            sizes[i - 1] *= 2;
            Arrays.pop_back();
            sizes.pop_back();
            NumArrays--;
        } else {
            break;
        }
    }
}
```

Para que se dé el peor caso en nuestra inserción, se tiene que cumplir la propiedad log(n+1) = k anteriormente vista, la cual nos quiere decir en representación binaria el caso en que n_i almacena solo 1s, puesto que esto llevara a un caso en el que ocurrira acarreo, es decir $k = k + 1$, equivalente a el merge sucesivo de todos los arreglos hasta formar uno solo.

Ahora, haciendo un poco de análisis a los costos que conlleva cada inserción, tenemos lo siguiente:

- Costo por elemento (C_e): Cada vez que insertamos un elemento, este es insertado como array de un solo elemento, por lo que conlleva costo 1.
- Costo total por elementos (C_{te}): Para M inserciones tendremos un costo de $\sum_{i=1}^{M} 1 = M$.
- Costo de merge sucesivos (C_m):
  - Cada 2 inserciones se hace un merge de arreglos de un elemento, lo que conlleva costo 2.
  - Cada 4 inserciones se hace un merge de arreglos de 2 elementos, lo que conlleva costo 4.
  - Cada 8 inserciones se hace un merge de arreglos de 4 elementos, lo que conlleva costo 8.
  - Cada 2^k inserciones se hace un merge de arreglos de $\frac{2^k}{2}$ elementos, lo que conlleva costo 2^k.

Por lo que en la inserción 2^k, los sucesivos merge tendran un costo de:

$$\sum_{i=1}^{k} 2^i = 2(2^k - 1) = 2^{k+1} - 2$$

- Costo total de merges (C_{tm}): Para la 2^k inserción tendremos k inserciones sucesivas, las cuales van aumentando como vimos anteriormente, por lo que tendremos:

$$\sum_{i=1}^{k} 2^{k+1} - 2 = k(2^{k+1} - 1) = 2^{k+1}k - k$$

- Costo por inserción (C_i): Esta dado por la suma de C_e y C_m, tal que: $1 + 2^{k+1} - 2$, Además $k = log(M)$, por lo que tenemos:

$$1 + 2^{k+1} - 2 = 2^{log(M)+1} - 1 = 2M - 1$$

- Costo total acumulado (C_t): El costo total para $k = log(M)$ inserciones es la suma de C_{te} y C_{tm}, tal que:

$$M + 2^{k+1}k - k = M + 2^{log(M)+1}log(M) - log(M) = M + 2Mlog(M) - log(M)$$

De esta forma, el costo estimado ($\hat{C}_i$) o ahorro que tomaremos sera de $m \cdot log(M)$, con m una constante, puesto que para M inserciones el costo sera de:

$$\frac{M + 2Mlog(M) - log(M)}{M} = 1 + 2log(M) - \frac{log(M)}{M} \in O(log(M))$$

Por lo tanto, se demostrara que se cumple la invariante $\sum_{i=1}^{M} \hat{C}_i - \sum_{i=1}^{M} C_i \geq 0$, que nos dice que la sumatoria de todos nuestros ahorros siempre debe ser mayor a la sumatoria de costes hasta la i-esima inserción.

$$\iff \sum_{i=1}^{M} mlog(M) \geq M + 2Mlog(M) - log(M)$$

$$\iff m \cdot Mlog(M) \in O(Mlog(M)) \geq M + 2Mlog(M) - log(M) \in O(Mlog(M))$$

$$\iff m \cdot O(Mlog(M)) \geq O(Mlog(M)))$$

Además, para m podemos tomar el valor de 3, puesto en $2log(M)+1-\frac{log(M)}{M}$, $\frac{log(M)}{M}$ nunca alcanzara valores mayores a 1 y por lo tanto:

$$2log(M) + 1 - \frac{log(M)}{M} \leq 2log(M) + 1 - 0.16 \leq 2log(M) + 0,84 \leq 3log(M)$$

En conclusión el costo amortizado para nuestra función de inserción será de O(log(M)).

i	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
t	1	2	4	4	8	8	8	8	16
C_e	1	1	1	1	1	1	1	1	1
C_m	0	2	0	$2+4=6$	0	0	0	$2+4+6=14$	0
C_i	1	3	1	7	1	3	1	15	1
C_t	1	4	5	12	13	16	17	32	33
$\hat{C}_i$	mlog(M)	mlog(M)	mlog(M)	mlog(M)	mlog(M)	mlog(M)	mlog(M)	mlog(M)	mlog(M)
cuenta	mlog(M)-1	2mlog(M)-4	3mlog(M)-5	4mlog(M)-12	5mlog(M)-13	6mlog(M)- 16	7mlog(M)-17	8mlog(M)-32	9mlog(M)-33

Cuadro 1: Tabla de contabilidad

i	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
t	1	2	4	4	8	8	8	8	16
C_e	1	1	1	1	1	1	1	1	1
C_m	0	2	0	6	0	0	0	14	0
C_i	1	3	1	7	1	3	1	15	1
C_t	1	4	5	12	13	16	17	32	33
$\hat{C}_i$	6,3	6,3	6,3	6,3	6,3	6,3	6,3	6,3	6,3
cuenta	8,5	15,0	23,5	26,0	34,5	41,0	49,6	44,0	52,6

Cuadro 2: Tabla de contabilidad con M=9 y con m=3

5. Los códigos de la prueba experimental están en el archivo .zip; El de insertar es insert-test.cpp y el de buscar es search-test.cpp. En insert-test la cantidad de elementos hacen saltos de 1000 en 1000, desde 1000 hasta 19000, mientras que en search-test se avanza en razón de $2^i - 1$, empezando desde 1023 hasta $(2^{10} - 1)$, y cabe mencionar que nuestro test de search mide el peor caso, buscando un promedio imposible (Un 8.0). Los gráficos se pueden ver en las figuras 2-6.

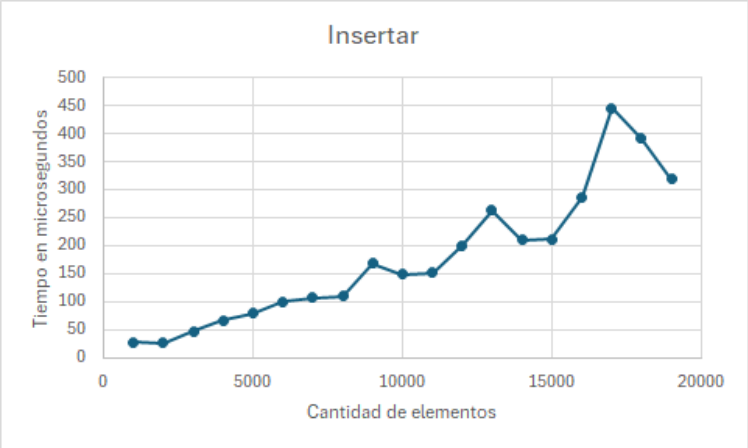

Figura 4: Gráfico test experimental de función insertar, el tiempo es medido en microsegundos

Size	Time
1000	28
2000	26
3000	47
4000	67
5000	79
6000	100
7000	107
8000	110
9000	168
10000	148
11000	152
12000	200
13000	263
14000	210
15000	212
16000	286
17000	445
18000	391
19000	318

Figura 5: Tabla con resultados de test experimental de función insertar

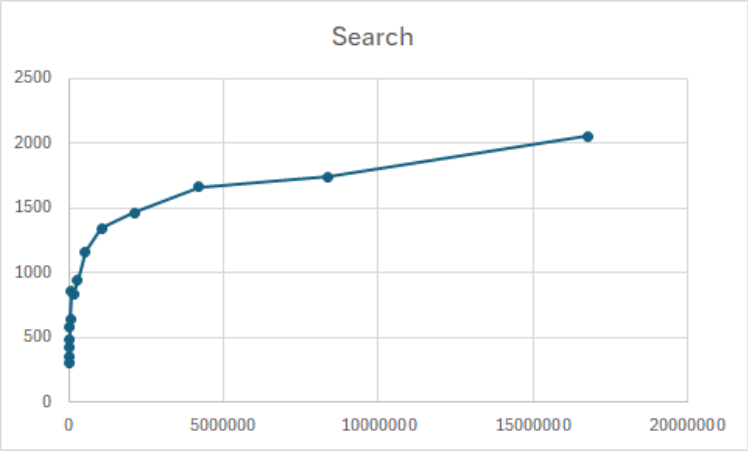

Figura 6: Gráfico test experimental de función buscar, el tiempo medido es en nanosegundos

Size	Time
1023	306
2047	348
4095	423
8191	487
16383	580
32767	639
65535	854
131071	831
262143	937
524287	1161
1048575	1345
2097151	1462
4194303	1661
8388607	1741
16777215	2054

Figura 7: Tabla con resultados de test experimental de función buscar

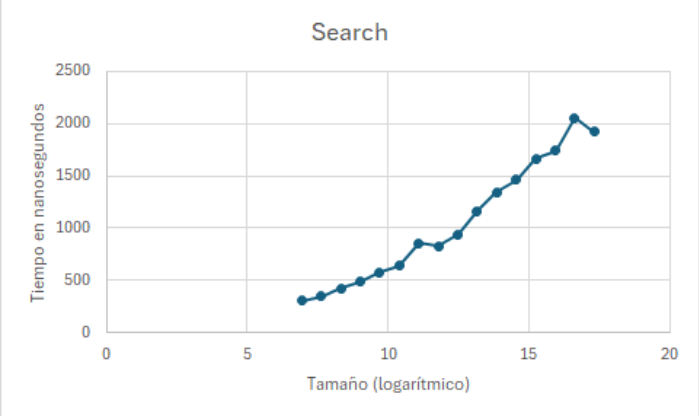

Figura 8: Mismo gráfico de la figura 4, pero el tamaño X es medido por función logarítmica

Podemos ver que la función buscar es mucho más rápida que la función insertar, pues un nanosegundo es el 0,001 de un microsegundo, y también podemos ver que ambas funciones son en su mayoría de proporción directa, osea, que mientras más elementos/promedios hayan en el BinormialArray (Nuestra estructura de elección para resolver el problema), más tiempo se demora.
Nos fijaremos en los tres máximos locales del gráfico de Insertar; cuando son insertados 9000, 13.000 y 17.000 elementos. Lo que hicimos primero fue correr 1000 repeticiones de este test para ver si era un problema de ruido, y pareciera haber corregido estos problemas.

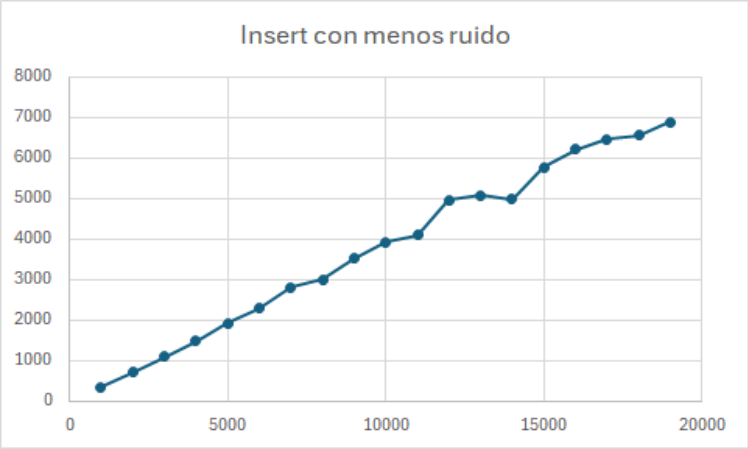

Figura 9: Gráfico test experimental de función insertar, el tiempo es con nanosegundos

Size	Time
1000	342
2000	711
3000	1100
4000	1489
5000	1917
6000	2282
7000	2811
8000	3004
9000	3526
10000	3926
11000	4093
12000	4959
13000	5082
14000	4993
15000	5781
16000	6214
17000	6461
18000	6563
19000	6895

Figura 10: Tabla con resultados de test experimental de función insertar

Se corrigen los máximos globales de 9.000 y 17.000 y sigue apareciendo un máximo local en 13.000, pero es increíblemente mínimo, por lo que nosotros creemos que son los procesos computacionales por detrás los que aumentan el tiempo en el elemento 13.000.

Pregunta 3 [1.2 puntos]: En clases usted estudió una forma de resolver el problema de Vertex-Cover aproximado, llamemos a esta estrategia A. Se propone la siguiente heurística para resolver el mismo problema. Iterativamente se va escogiendo el vértice v de mayor grado (si hay empate no importa el orden) y se eliminan todas las aristas que tengan a v como uno de sus extremos. Llamemos a esta segunda estrategia B.

Se pide lo siguiente:

a) Implemente en C/C++ ambas estrategias A y B.

b) Evalúe su implementación en el siguiente grafo: *Enlace de descarga del grafo*. Encuentre la solución exacta y la solución entregada por ambas estrategias. ¿Cuál estrategia entregó una mejor solución? ¿Se cumple que la estrategia A es 2-aproximación? Justifique.

c) Evalúe su implementación con el siguiente grafo: *Enlace de descarga del grafo*. Discuta cuál estrategia dio mejores resultados. Según lo visto en clases, ¿qué puede decir de la solución real? (No se pide encontrar la solución real)

d) Considere el grafo bipartito presentado en la Figura 11. Calcule la solución exacta y determine el razón de aproximación dado por ambas estrategias, en el caso de la estrategia B, comience por el vértice 6. ¿Qué puede concluir respecto al razón de aproximación para la estrategia B?

Observación: En los archivos *grafo1.txt* y *grafo2.txt* correspondientes a los grafos encontrará información de ellos en la primera línea para que así los pueda leer.

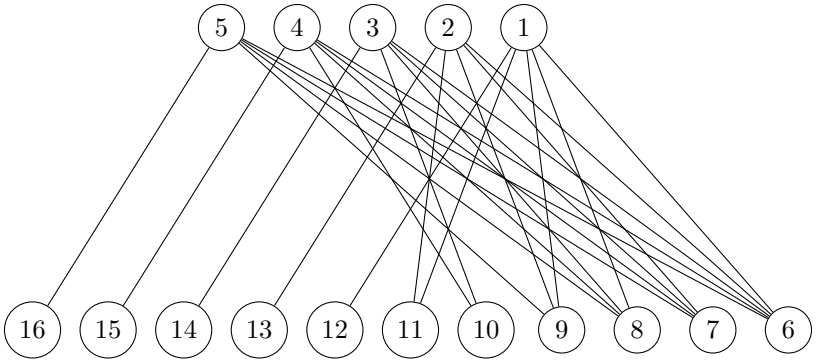

Figura 11: Grafo bipartito. Los grados de los vértices 1, 2, 3, 4 y 5 son iguales a 5. Los grados de los vértices 6, 7, 8, 9, 10, 11, 12, 13, 14, 15 y 16 son 5, 4, 4, 3, 2, 2, 1, 1, 1, 1 y 1, respectivamente.

Solución:

a) En el archivo zip adjunto, dentro de la carpeta Item3 se encuentran los archivos A.cpp y B.cpp con sus respectivas implementaciones.

b) Por medios propios, llegamos a que la solución exacta para este problema no era solo una, y además, involucra un total de 7 vértices, seguido de esto, lo confirmamos con una página llamada ***Visualgo***, que nos arrojo el siguiente resultado según su búsqueda a fuerza bruta:

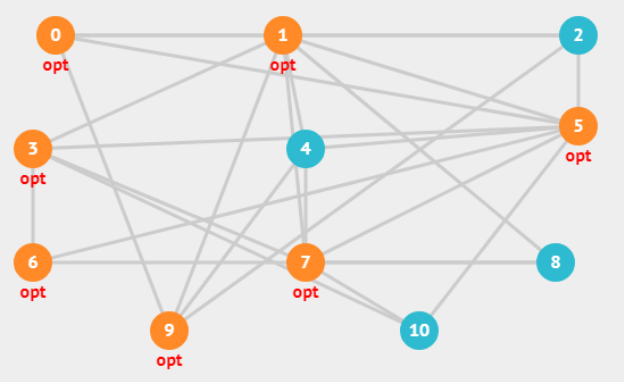

Figura 12: El Vertex Cover es representado por los nodos naranjos

De ambos algoritmos, el algoritmo que dio mejores resultados fue el de la estrategia B, que visita los vértices 1, 5, 7, 9, 3, 0 y 6, siendo un resultado óptimo. Mientras que la estrategia A da los vértices 0, 1, 2, 5, 3, 4, 6 y 7, que son un total de 8, por lo que no es una solución óptima.

En cuanto a la estrategia A, si es 2-aproximación, puesto que dentro de la su descripción, es decir, que puede dar como mucho el doble de lo óptimo, tenemos que $8 \in [7, 14]$.

c) En nuestras implementaciones, llegamos a los siguientes resultados para el grafo propuesto:

```
PS C:\Users\Diego\Desktop\Ada\Item3> ./A
0, 1, 2, 4, 3, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 25, 15, 16, 17, 47, 19, 55, 20, 24,
 21, 23, 22, 40, 26, 45, 27, 41, 28, 66, 30, 31, 32, 33, 34, 35, 37, 38, 39, 42, 43,
56, 46, 102, 48, 49, 50, 58, 51, 52, 53, 76, 59, 60, 61, 86, 62, 63, 64, 65, 67, 103,
 68, 71, 69, 104, 70, 72, 73, 74, 75, 77, 79, 84, 81, 97, 83, 87, 88, 89, 90, 91, 93,
 94, 96, 100,
Tamano: 90
```

Figura 13: Estrategia A.

```
PS C:\Users\Diego\Desktop\Ada\Item3> ./B
8, 12, 84, 3, 72, 30, 66, 47, 73, 40, 11, 71, 58, 9, 10, 75, 6, 74, 86, 100, 13, 31,
35, 64, 99, 4, 20, 23, 65, 76, 94, 5, 26, 33, 34, 59, 7, 19, 41, 57, 91, 102, 104, 0,
 15, 22, 38, 42, 51, 60, 61, 87, 97, 14, 24, 27, 43, 48, 62, 67, 70, 88,
Tamano: 62
```

Figura 14: Estrategia B.

Como se puede observar, la estrategia más eficiente es la B, que llega al Vertex Cover de 62 vértices en vez de los 90 vértices a los que llega la estrategia A. Lo que se puede visualizar en las estrategias es que la Estrategia B llega más rápido a la solución debido a que elimina los vértices con más grados, entonces para el momento en que se llega al vértice n°50, los 50 vértices que tenían más conexiones ya no existen, en vez de los vértices eliminados de la estrategia A que pueden llegar a tener sólo un grado, lo que implica que las operaciones de reducir a 0 los grados de los vértices en el grafo son menores.

d) Dado que el Grafo es bipartito, con tomar el grupo de vertices [1, 2, 3, 4, 5] ya tendriamos nuestro mínimo vertex cover. En nuestras implementaciones, llegamos a los siguientes resultados para el grafo propuesto:

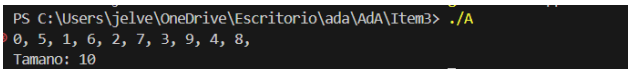

Figura 15: Estrategia A

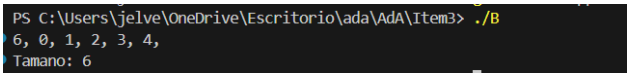

Figura 16: Estrategia B

Para ambas estrategias los resultados están dentro de la 2-aproximación, puesto que como se dijo anteriormente, el mínimo vertex cover es de 5 vértices, y $10, 6 \in [5, 10]$. Además, si la estrategia B no hubiese empezado por el vértice 6, esta hubiese arrojado la solución óptima.

En cuanto a la razón de aproximación de ambos resultados tenemos lo siguiente:

Como este es un problema de minimización se utilizara $0 \leq S.Óptima \leq S.Algoritmo$ y

$\rho(n) = \dfrac{S.Algoritmo}{S.Óptima}$

Para A:

$$\rho(n) = \frac{10}{5} = 2$$

Para B:

$$\rho(n) = \frac{6}{5} = 1,2$$

Pregunta 4 [1.2 puntos]: Una escuela necesita llevar a sus estudiantes, maestros, apoderados e invitados a las Olimpiadas a realizarse en otra ciudad. Para ello, dispone de m autobuses, los cuales tienen P asientos. Las personas que irán forman parte de uno de los n equipos que participarán en las Olimpiadas. Tenga en cuenta que el director desea que cada equipo completo vaya al interior de un autobús, pero puede darse el caso que un autobús se complete, por lo que algunas personas tendrán que ir en otro autobús alejado de su equipo. Su trabajo es asignar los equipos en los autobuses de forma tal que la cantidad de personas alejadas de sus equipos sea mínima. De ser posible, consiga que todos los equipos se encuentren completos al interior de un bus.

Sea w_i la cantidad de miembros pertenecientes al equipo i ($1 \leq i \leq n$), conformados por estudiantes, maestros, apoderados e invitados. Asuma que $\Sigma_{i=1}^{n} w_i \leq mP$, es decir, existen suficientes asientos para todas las personas. Además, asuma que $w_i \leq P$, es decir, que un equipo completo puede caber dentro de un autobús. También asuma que $1 \leq n \leq 2m$.

Ejemplos:

Suponga el siguiente caso: Tiene 4 autobuses con 50 asientos para 6 equipos ($m = 4$, $n = 6$, $P = 50$). La cantidad de miembros de cada equipo son los siguientes: $\{40, 20, 40, 50, 10, 30\}$. En este caso, en el primer autobús iría el equipo de 50, en el segundo iría un equipo de 40, en el tercer autobús iría el otro equipo de 40 y el equipo de 10, y en el cuarto autobús iría el equipo de 20 y el equipo de 30. No existen personas alejadas de sus equipos.

Suponga el siguiente ejemplo: Tiene 4 autobuses con 50 asientos para 6 equipos ($m = 4$, $n = 6$, $P = 50$). La cantidad de miembros de cada equipo son los siguientes: $\{40, 30, 40, 50, 10, 30\}$. En este caso, en el primer autobús iría el equipo de 50, en el segundo iría un equipo de 40, en el tercer autobús iría el otro equipo de 40 y el equipo de 10, y en el cuarto autobús iría un equipó de 30 y 20 miembros del segundo equipo de 30. Quedarían 10 personas alejadas de sus equipos.

Solución:

El problema planteado es nada más ni nada menos que una variante del problema de los contenedores (Bin Packing Problem), un problema de optimización que consiste en asignar una C cantidad en n espacios o 'Contenedores' , cada uno con una determinada capacidad, y el objetivo de este problema es encontrar el mejor orden para distribuir estos equipos de tal manera que estén, en su mayoría, en el mismo bus.

La forma que elegimos para solucionar el problema es a través de BacktrackingDFS, para poder iterar sobre todas las formas de distribución de los equipos, y elegir la que separe menos a los equipos (La solución óptima).

Se logrará esto a través de un sistema de penalización; En el que mientras más miembros de un equipo son separados de un bus, se 'penalizará' a la iteración con un mayor puntaje. El algoritmo determinará cuál es la mejor distribución cuando este encuentre una solución, y luego empezará a comparar cuales tengan el puntaje más pequeño. También podará las ramas que tengan un puntaje mayor a la de la mejor solución encontrada hasta el momento. El algoritmo derechamente se detendrá cuando encuentre una solución de puntaje 0, significando que encontró una solución que no separa ningún equipo, por lo que no tuvo ninguna penalización.

Por comodidad, hicimos una estructura de datos:

```
struct tupla{
    int bus; // Indice del bus.
    int equipo; // Indice del equipo.
    int miembros; // Cantidad de miembros.
};

class Backtracking{
    private:
        int m; // Cantidad de buses.
        vector<tupla> mejorSolucion;
        int puntajeSolucion;
}
```

Con las siguientes funciones (El constructor de Backtracking es el algoritmo principal):

```
        void BacktrackingDFS(vector<tupla> A, int k, int* Buses, int* equipos, int penalizacion, int suma){
            // Cota de poda.
            if(penalizacion >= puntajeSolucion) return;

            // Cota de solución optima.
            if(puntajeSolucion == 0) return;

            // Caso base.
            if(suma == 0){
                if(penalizacion < puntajeSolucion){
                    puntajeSolucion = penalizacion;
                    mejorSolucion.clear();
                    for(int i = 0; i < A.size(); i++){
                        mejorSolucion.push_back(A[i]);
                    }
                }

            // Caso recursivo general.
            } else {
                for(int i = 0 ; i < m; i++){
                    if (Buses[i] > 0){
                        //Caso en que un bus puede dejar entrar un equipo completo.
                        if(Buses[i] >= equipos[k]){
                            int aux1 = equipos[k];
                            int aux2 = Buses[i];
                            A.push_back({i, k, equipos[k]});
                            Buses[i] -= aux1;
                            equipos[k] = 0;
                            BacktrackingDFS(A, k+1, Buses, equipos, penalizacion, suma - aux1);
                            A.pop_back();
                            equipos[k] = aux1;
                            Buses[i] = aux2;
                        //Caso en que se va a dejar a una fracción del equipo en el bus
                        } else {
                            int aux1 = equipos[k];
                            int aux2 = Buses[i];
                            equipos[k] -= aux2;
                            A.push_back({i, k, Buses[i]});
                            Buses[i] = 0;
                            if(equipos[k] > 0) BacktrackingDFS(A, k, Buses, equipos, penalizacion + min(equipos[k], aux2), suma - aux2);
                            else BacktrackingDFS(A, k+1, Buses, equipos, penalizacion, suma - aux2);
                            A.pop_back();
                            equipos[k] = aux1;
                            Buses[i] = aux2;
                        }
                    }
                }
            }
        }

    public:
        Backtracking(int m, int P, int n, int* equipos){
            this->m = m;
            this->puntajeSolucion = INT_MAX;
            vector<tupla> A;
            int* buses = new int[m];
            for(int i = 0; i < m; i++){
                buses[i] = P;
            }
            int suma = 0;
            for(int i = 0; i < n; i++){
                suma += equipos[i];
            }
            BacktrackingDFS(A, 0, buses, equipos, 0, suma);
        }

        vector<tupla> getMejorSolucion(){
            return mejorSolucion;
        }

        int getPuntajeSolucion(){
            return puntajeSolucion;
        }

        ~Backtracking(){}
};
```

Corriendo los ejemplos propuestos, da de output lo siguiente

a) Ejemplo 1: (m = 4, n = 6, P = 50), cantidad de miembros por equipo 40,20,40,50,10,30:

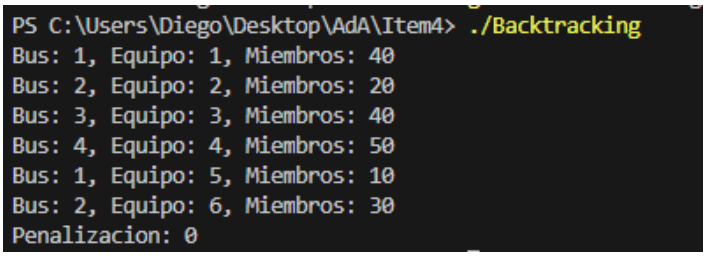

Figura 17: Output de nuestro algoritmo

b) Ejemplo 2: (m = 4, n = 6, P = 50), cantidad de miembros por equipo 40,30,40,50,10,30:

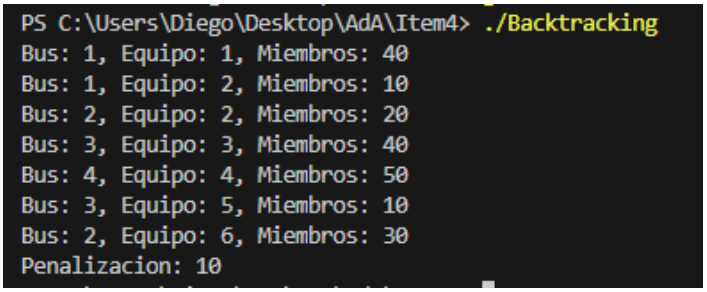

Figura 18: Output de nuestro algoritmo

La implementación y los tests se encuentran en el archivo .zip.

Pregunta 5 [1.2 puntos] Una fábrica de cajas fabrica cajas de todos los materiales y tamaños posibles. La empresa busca guardar sus cajas ocupando el menor espacio posible, guardando las cajas dentro de otras cajas.

Dada n cajas, cada caja c_i tiene dimensiones x_i, y_i y z_i y un peso w_i. Para evitar dañas las cajas, una caja más pesada no puede ser guardada en una caja más ligera. Su trabajo consiste en encontrar la mayor cantidad de cajas que pueden ser guardadas dentro de otras cajas.

Ejemplos:

Suponga el siguiente ejemplo: Tiene 3 cajas disponibles ($n = 3$) con las características (x_i, y_i, z_i, w_i): $\{(40, 50, 30, 3), (30, 25, 40, 1), (41, 49, 30, 2)\}$. Es posible guardar las tres cajas metiendo c_3 dentro de c_1 y c_2 dentro de c_2. La mayor cantidad de cajas que pueden ser guardadas dentro de otras cajas es 3.

Suponga el siguiente ejemplo: Tiene 4 cajas disponibles ($n = 4$) con las características (x_i, y_i, z_i, w_i): $\{(40, 50, 30, 2), (21, 22, 23, 1), (30, 51, 40, 1), (41, 49, 30, 3)\}$. Es posible guardar c_2 en cualquiera de las otras tres cajas. La mayor cantidad de cajas que pueden ser guardadas dentro de otras cajas es 2.