DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico I

Algoritmos y Estructuras de Datos III
Primer Cuatrimestre de 2015

Integrante	LU	Correo electrónico
Aldasoro Agustina	86/13	agusaldasoro@gmail.com
Noriega Francisco	660/12	frannoriega.92@gmail.com
Zimenspitz Ezequiel	155/13	ezeqzim@gmail.com
Zuker Brian	441/13	brianzuker@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

Resumen

Habiéndonos sido dado una serie de tres problemáticas a resolver, se plantean sus respectivas soluciones acorde a los requisitos pedidos. Se adjunta una descripción de cada problema y su solución, conjunto a su análisis de correctitud y de complejidad sumado a su experimentación. El lenguaje elegido para llevar a cabo el trabajo es C++.

Dentro de cada *.cpp* está el comando para compilar cada ejercicio desde la carpeta donde se encuentran los mismos. A continuación se los adjunta. El flag *-std=c++11* debió ser añadido dado que utilizamos la librería <chrono>, la cual nos permitió medir tiempos de ejecución:

1. *g++ -o main Zombieland.cpp -std=c++11*
2. *g++ -o main AltaFrecuencia.cpp -std=c++11*
3. *g++ -o main SenorCaballos.cpp -std=c++11*

Índice

1. Problema 1: ZombieLand

1.1. Descripción de la problemática

En un país con *n* ciudades, se encuentran una determinada cantidad de Zombies y de Soldados por cada una de ellas. El objetivo del problema es exterminar la invasión zombie, para ello es necesario un enfrentamiento *zombies vs soldados* por cada ciudad. Para que el combate sea positivo en una ciudad, es decir se logre matar a todos los zombies de la misma, es necesario que la cantidad de zombies sea, a lo sumo, diez veces más grande que la cantidad de soldados.

Se sabe de antemano cuántos zombies y cuántos soldados se encuentran atrincherados en cada ciudad. Los soldados acuartelados no pueden moverse de la ciudad en la que están, pero sí se cuenta con una dotación de soldados extra que se la puede ubicar en cualquiera de las *n* ciudades para salvarla. La cantidad de soldados extra es ilimitada, mas los recursos para trasladarlos no lo son. El costo del traslado depende de cada ciudad. Siempre que se respete el presupuesto del país, se pueden trasladar todos los soldados necesarios para salvar a cada ciudad.

Debido a que los recursos económicos son finitos, no siempre va a ser posible salvar a las n ciudades. Lo que se desea en este problema es maximizar la cantidad de ciudades salvadas, respetando el presupuesto. Es decir, se deben establecer las cantidades de soldados extras enviados a cada ciudad de modo que la cantidad de ciudades salvadas sea la óptima y gastando un monto por debajo del presupuesto. El algoritmo debe tener una complejidad temporal de $O(n.log(n))$, siendo n la cantidad de ciudades del país.

Aca se podría poner unos dibujitos de soluciones óptimas como para que quede más lindo

1.2. Resolución propuesta y justificación

Para la resolución del problema decidimos utilizar un algoritmo goloso, que salvará en cada paso a la ciudad que más le convenga en ese momento, es decir, la que permita maximar la cantidad de ciudades salvadas.

Como primera instancia, el algoritmo simplemente calcula, para cada ciudad, cuánto sería el costo de salvarla. Para ello, primero se calcula la cantidad de soldados extra necesarios y luego se multiplica por el costo de traslado de cada unidad:

```
soldados_extras_necesarios = redondeo_hacia_arriba((zombies - (soldados_existentes * 10)) /10)

costo_total = costo_unitario * soldados_extras_necesarios
```

Luego de haber obtenido una magnitud con la cual se pueden comparar las ciudades entre sí, se ordenan las ciudades de menor a mayor en base al costo de salvarla para ser recorridas secuencialmente y enviar los ejércitos requeridos hasta que se agote el presupuesto.

Notar que si alguna ciudad no requiere soldados extras para ser salvada, entonces serán las primeras en ser salvadas dado que el costo_total será igual a 0.

Se recorren secuencialmente las ciudades ordenadas por el costo_total, de modo que para cada una se va a comparar el costo de salvarla contra el presupuesto restante en ese momento (presupuesto_actual). Si es factible el salvataje, se resta el costo_total del presupuesto_actual y se envían las tropas necesarias a la ciudad; en caso contrario se la marca como ciudad perdida.

Vale aclarar que el orden impuesto a las ciudades implica que cuando ya no se pueda salvar a una ciudad, no se podrá salvar a ninguna otra de las restantes.

A continuación demostraremos que el algoritmo resuelve efectivamente el problema planteado.

Teorema El algoritmo resuelve el problema planteado, salvando la mayor cantidad de ciudades posibles.

Demostración Para demostrar el teorema enunciado, supondremos que nuestro algoritmo no devuelve una solución óptima, y tomaremos la solución óptima con más ciudades salvadas en común. Sean C_k las ciudades de la solución óptima y D_k las ciudades de la solución dada por el algoritmo, y supondremos que están ordenadas de menor a mayor, según el costo de ser salvadas. Sea C_j la primer ciudad distinta a las ciudades de D, de modo que $C_i = D_i \; \forall i, i < j$.

Entonces, hasta C_{i-1}, el costo total por salvar dichas ciudades es el mismo al de D hasta D_{i-1}. Pero ya que el algoritmo elige siempre la ciudad que (pudiendo salvarse) cueste lo menor posible, eso significa que C_i tiene un costo igual o mayor al de D_i, con lo cual si en C reemplazamos C_i por D_i, seguimos teniendo presupuesto (en particular, mayor que el que se tenía) y se salvan la misma cantidad de ciudades, por lo que debe ser óptimo.

Pero esta nueva solución, es una solución óptima que tiene más ciudades en común que C, por lo que es absurdo, ya que C era la solución óptima que más ciudades en común tenía con D.

El absurdo proviene de suponer que D no es óptimo y que por lo tanto la solución óptima con más ciudades salvadas en común con D, no es D.

Así, D debe ser óptimo, en el sentido de que debe tener la mayor cantidad ciudades que pueden ser salvadas, para el presupuesto y costos para cada ciudad dados.

1.3. Análisis de la complejidad

La complejidad de nuestra solución es $O(n.log(n))$, siendo *n* la cantidad de ciudades del país.

En primera instancia, guardamos los datos de las ciudades pasadas por stdin en structs y los dejamos dentro de un vector, para luego poder utilizarlas de un modo práctico. Como esto se realiza secuencialmente, tiene costo lineal ***O(n)***.

Algorithm 1: zombieland

for each *ciudad* $\in$ *país* **do**
  Calcular la cantidad de soldados extra necesarios y el costo de salvarla.
  Almacenar esta información en un vector *datos* mediante *push_back()*
Ordena al vector *datos* mediante *sort()*
while *pueda salvar* **do**
  if *puede ser salvada* **then**
    Indicar cantidad de soldados extras enviados y actualizar el presupuesto_actual
while *haya ciudades insalvables* **do**
  cantidad de soldados extras enviados = 0
es necesario este cacho? habla del stdout... no me parece necesario en absoluto... yo lo sacaría
for each *ciudad* en vector *datos* **do**
  Insertar en el vector *respuesta*[ciudad.id] la ciudad actual.

En el codigo, yo pondria este reordenamiento dentro de zombieland, ya que hay que considerarlo para medir tiempos.
A mi no me parece que sea necesario para medir tiempos... o sea, arrancar de la instancia pasada por parametro, salvo que la limemos y usemos avls, heaps, etc. no me parece necesario contarlas. lo mismo para escribir, la respuesta está, en el ej2 tenemos que calcular un pedazo de respuesta post algoritmo y lo hacemos y lo consideramos, el std out extra porque deberíamos considerarlo? es algo que nos piden para visualizar, si las cosas andan mal, porque pueden llegar a estar mal

Dijo el ayudante que eso si habia que ponerlo....

El código presenta un **primer ciclo *for*** que calcula el costo de salvar a cada ciudad y las agrega mediante push_back() a un vector. El ciclo recorre linealmente todas las ciudades por lo que tiene complejidad $O(n)$.

El cálculo de salvar a cada ciudad coincide con el descripto en la sección anterior, el cual por ser operaciones aritméticas es $O(1)$. Armar el nuevo struct para insertar dentro del vector *datos* también posee un costo constante $O(1)$. La función push_back() [1] tiene costo $O(1)$ amortizado, lo que implica que cuando no precisa redimensionar el vector cuesta esto, y cuando lo hace, toma tiempo lineal en la cantidad de elementos. Como insertamos durante todo el ciclo tomamos el costo amortizado $O(1)$. Por lo tanto, la complejidad total del primer ciclo for nos da ***O(n)***.

Le sigue **ordenar el vector** con estos datos, para ello usamos sort() [2] cuya complejidad es ***O(n.log(n))***.

A continuación, se realizan dos **último ciclos *while*** el primero salva todas las ciudades que pueda, mientras dure el presupuesto y el segundo deja en *0 soldados enviados* a las ciudades que no pueden ser salvadas. Estos cálculos aritméticos y asignaciones son todos de complejidad constante $O(1)$. El primer ciclo toma $O(ciudades\_salvadas)$ y el segundo $O(n - ciudades\_salvadas)$ dando como resultado un recorrido lineal sobre todas las ciudades, por lo tanto lo hace con complejidad ***O(n)***.

Para mi no va...Dijo que siiiiii

[1] http://www.cplusplus.com/reference/vector/vector/push_back/
[2] http://www.cplusplus.com/reference/algorithm/sort/

Finalmente, se debe **reordenar el vector** obtenido hasta ahora para que quede en orden creciente respecto de su *id*. Como esto se hace recorriendo secuencialmente el primer vector, asignándole uno a uno los elementos al nuevo vector *respuesta* indexados; sólo hace una pasada lineal con costo ***O(n)***.

Como cada paso de los mencionados son secuenciales, las complejidades se suman, obteniendo:

$O(n)$ + $O(n.log(n))$ + $O(n)$ que es igual a ***O(n.log(n))*** por propiedades de O.

1.4. Código fuente

```
struct ciudad{
    int zombies;
    int soldados;
    int costo;
};
```

```
struct ciudad2{
    int numCiudad;
    int soldadosNecesarios;
    int costoTotal;
    bool operator< (const ciudad2& otro) const{
        return costoTotal < otro.costoTotal;
    }
};
```

```
int main(int argc, char const *argv[]){
    chrono::time_point<chrono::system_clock> start, end;
    long int cantCiudades, presupuesto;
    vector<ciudad> pais;
//Leemos informacion del problema
    cin >> cantCiudades;
    cin >> presupuesto;
    for (int i = 0; i < cantCiudades; ++i){
        ciudad alguna;
        cin >> alguna.zombies;
        cin >> alguna.soldados;
        cin >> alguna.costo;
        pais.push_back(alguna);
    }
    long int salvadas = 0;
    start = chrono::system_clock::now();
//Aplicamos el algoritmo
    vector<ciudad2> res = zombieland(cantCiudades, presupuesto, pais, salvadas);
    end = chrono::system_clock::now();
    vector<long int> entregados(cantCiudades);
//Stdout pedido
    for (int i = 0; i < cantCiudades; ++i){
        entregados[res[i].numCiudad] = res[i].soldadosNecesarios;
    }
    cout << salvadas << " ";
    for (int i = 0; i < cantCiudades; ++i){
        cout << entregados[i] << " ";
    }
    cout << endl;
    chrono::duration<double> elapsed_seconds = end-start;
    cout << "Tiempo: " << elapsed_seconds.count() << endl;
    return 0;
}
```

```
const vector<ciudad2> zombieland(long int cantCiudades, long int presupuesto,
const vector<ciudad>& pais, long int& salvadas){
    salvadas = 0;
    vector<ciudad2> datos;
    for (int i = 0; i < cantCiudades; ++i){
        ciudad2 actual;
//ID de la ciudad
        actual.numCiudad = i;
//Calculamos el costo de salvar la ciudad i
        double diferencia = (pais[i].zombies - pais[i].soldados * 10);
        if (diferencia > 0)
            actual.soldadosNecesarios = ceil(diferencia/10);
        else
            actual.soldadosNecesarios = 0;
        actual.costoTotal = actual.soldadosNecesarios * pais[i].costo;
//Lo agregamos al vector
        datos.push_back(actual);
    }
//Ordenamos el vector de menor a mayor costo para salvarlas
    sort(datos.begin(), datos.end());
    long int dif = presupuesto;
//Vemos cuantas salvamos respetando el presupuesto
    int i = 0;
    while(i<cantCiudades && dif >= 0){
        dif = presupuesto - datos[i].costoTotal;
        if (dif>=0){
            salvadas++;
            presupuesto = dif;
            ++i;
        }
    }
//Las que sobran no se salvan
//seteamos los soldados necesarios en 0 para imprimir una respuesta correcta
    while(i<cantCiudades){
        datos[i].soldadosNecesarios = 0;
        ++i;
    }
    return datos;
}
```

1.5. Experimentación

Constrastación Empírica de la complejidad

2. Problema 2: Alta Frecuencia

2.1. Descripción de la problemática

Se quiere transmitir información secuencialmente mediante un enlace el mayor tiempo posible. Los enlaces tienen asociadas distintas frecuencias, con un costo por minuto y un intervalo de tiempo (sin cortes) en el cual funcionan. Se utilizan durante minutos enteros, y es posible cambiar de una frecuencia a otra instantánemente (del minuto 1 al 4 uso la frecuencia A y del 4 al 6 la B). Los datos del precio y e intervalo de tiempo de cada frecuencia son dados. Se desea optimizar este problema para transmitir todo el tiempo que tenga al menos una frecuencia abierta, pero gastando la menor cantidad de dinero. Se debe contar con una complejidad de $O(n.log(n))$.

A continuación se muestran dos casos particulares de este problema. En ambos se ofrecen tres frecuencias, con distintos costos cada una. Se puede ver recuadrado en violeta cuál es la elección que debe hacerse por intervalo de tiempo.

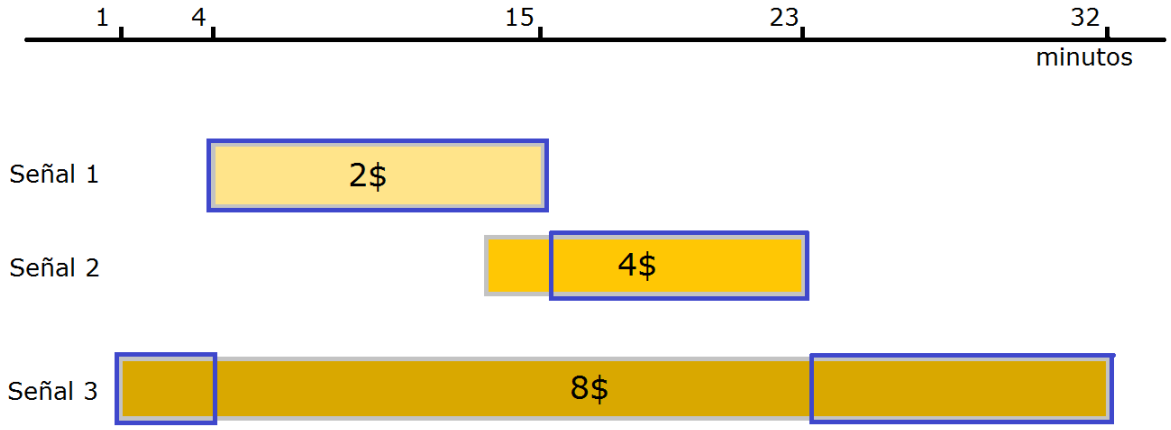


Figura 1: Ejemplo 1

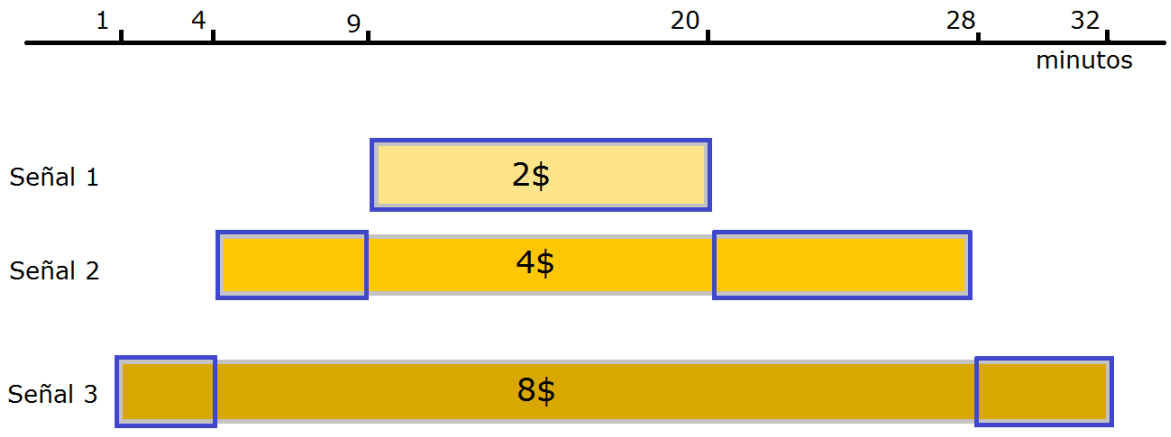


Figura 2: Ejemplo 2

2.2. Resolución propuesta y justificación

El algoritmo que utilizamos pertenece a la familia de *Divide & Conquer*.

Aca pasa lo mismo de la implementacion porque hablamos mucho de vector y bla...

El primer paso consiste en ordenar las frecuencias de menor a mayor en base al costo de cada una.

Luego, se sigue el esquema clásico de Divide & Conquer:

```
divide(conjuntoDeFrecuencias F){
    Si hay mas de un elemento:
        Divido F en mitades A, B.
        intervalosA = divide(A)
        intervalosB = divide(B)
        Devuelvo conquer(intervalosA, intervalosB)
    Si hay un solo elemento:
        Lo devuelvo.
}
```

En palabras, si hay una sola frecuencia, la devolvemos, pues es trivial que su intervalo de duración es el más barato y el de mayor extensión temporal.

Si no, intervalosA será el conjunto de intervalos en la que funcionará cada frecuencia, de modo que el costo de contratar el servicio con este cronograma sea mínimo en el costo y máximo en la cantidad de tiempo de uso. E intervalosB será un conjunto con las mismas caraterísticas, con la diferencia de que el A será el más óptimo de la mitad más barata y el B el más óptimo de la mitad más cara. Esto resulta de haber ordenado las frecuencias por su costo antes de comenzas con este tramo de algoritmo.No se si este parrafo se entiende mucho

Al hacer conquer(intervalosA, intervalosB) se obtiene el conjunto de intervalos con las características mencionadas pero de todas las frecuencias.No habria que explicar mas aca?

Este último paso abusa del invariante: todos los intervalos del conjunto intervalosA deben aparecer en el conjunto solución, y que lo único que debe agregar son los intervalos de intervalosB que o bien aumentan el rango de tiempo para transmitir (uso el servicio desde antes o más tiempo) o bien completan gaps que puedan existir entre las frecuencias más baratas.

2.3. Análisis de la complejidad

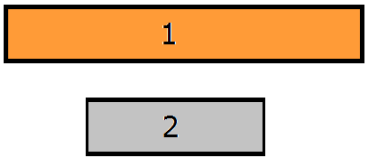
Para realizar este análisis primero es necesario calcular cuántos intervalos contendrá, como máximo, el conjunto solución (desde ahora "*CS*"). Este valor será $2n-1$, siendo n la cantidad de frecuencias dadas como parámetro.

Esto se deduce de analizar las posibles entradas para el algoritmo. Primero vamos a analizar el caso donde no existan dos frecuencias pasadas como parámetro con el mismo valor. De este modo la solución óptima al problema va a ser única.

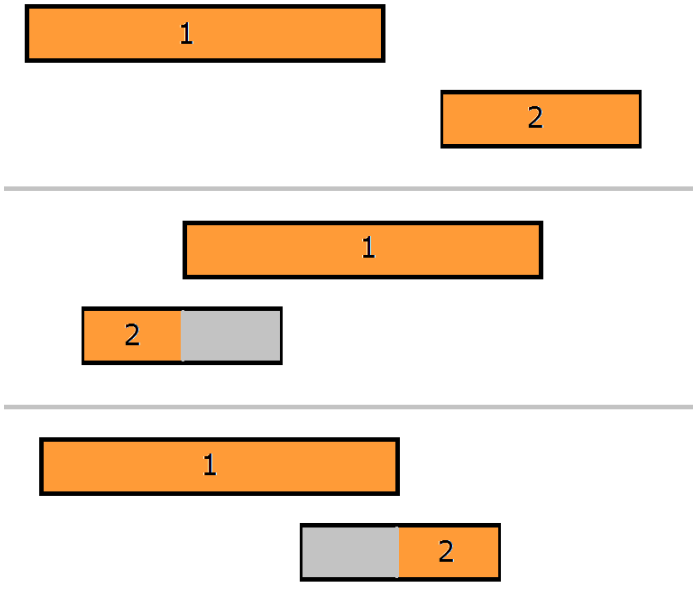
Supongamos **n=1**. El intervalo de la frecuencia pasada como parámetro será el único elemento del *CS* y verifica $2n-1 = 2.1-1 = 1$.

Luego, tomamos **n=2**. De este modo, ambas pueden solaparse o ser disjuntas. Llamamos 1 a la frecuencia más barata y 2 a la más cara, 1 va a estar incluida completa en *CS* y 2 puede formar dos intervalos, uno o ninguno. A continuación, se adjuntan cada uno de los casos, indicando en naranja los intervalos que pertenecerán a *CS*.

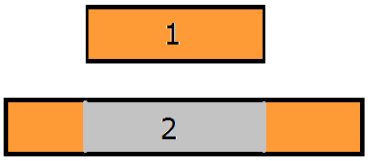
La frecuencia 2 no forma ningún intervalo:



La frecuencia 2 forma un intervalo:

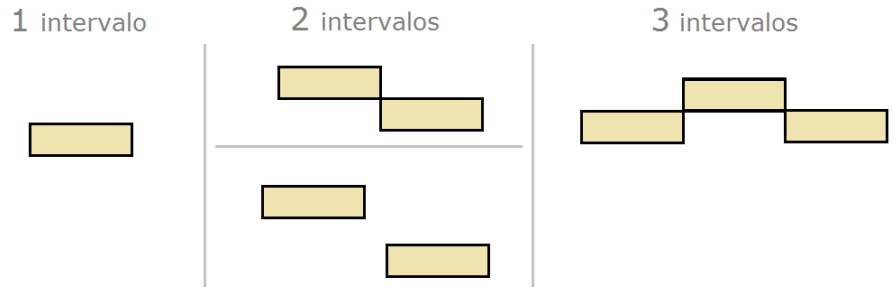


La frecuencia 2 forma dos intervalos:



Es decir, para n=2 la cantidad máxima de intervalos posibles es 3 intervalos, lo cual también cumple $2n - 1 = 2.2 - 1 = 3$.

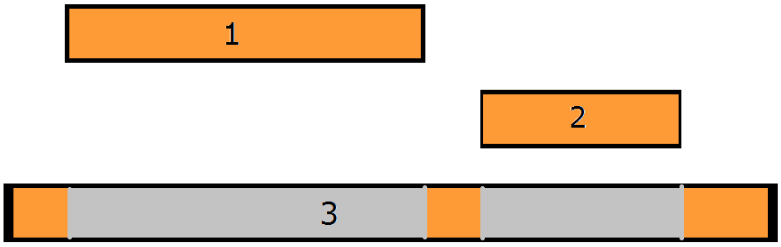
Al momento de tomar **n=3**, lo hacemos considerando las frecuencias 1, 2 y 3. Partimos del caso anterior al que le añadimos la frecuencia 3. Es decir, vamos a contar con a lo sumo 3 intervalos. La distribución de los intervalos se va a dar de la siguiente manera:



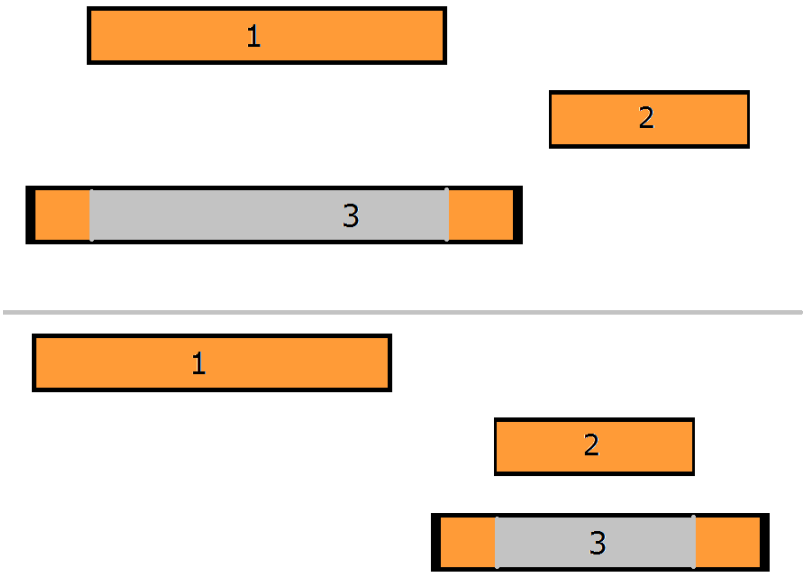
Podemos observar que de las cuatro distribuciones distintas que pueden ocurrir, en sólo una los intervalos son completamente disjuntos, esto es en el caso donde se añadió el intervalo 2 completo. En los demás casos, si bien pueden ser uno, dos o tres intervalos distintos al unirlos conforman un intervalo continuo, por lo cual entre ellos no va a haber gaps, es decir sólo se podrán añadir intervalos en los bordes. Por este motivo, a estos casos los tomamos análogos al caso de $n = 2$ donde se agregarán a lo sumo dos intervalos, lo cual conforma en el caso máximo 3 intervalos pre-existentes y 2 nuevos, dando un total de 5. Esto cumple lo planteado $2n - 1 = 2.3 - 1 = 5$.

Ahora debemos considerar por separado, el caso donde contamos con dos intervalos pre-existentes totalmente disjuntos.

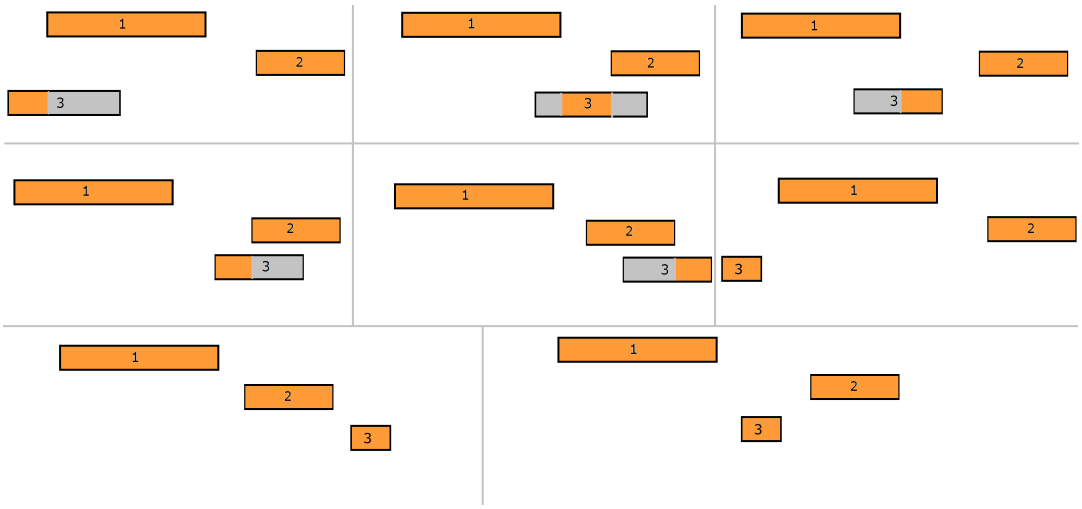
El único caso donde se adicionan tres intervalos es el siguiente:



Los casos donde se adicionan dos intervalos son los mostrados a continuación:



Los siguientes son los casos donde al agregar la frecuencia 3 sólo se adiciona un intervalo:



Por consiguiente, el máximo de intervalos que pueden añadirse son tres, considerando siempre sólo dos pre-existentes, lo cual da un total de 5 que cumple: $2n - 1 = 2.3 - 1 = 5$.

Extender la validez a n!!

Partimos de la hipótesis de que los costos de las frecuencias eran todos distintos, lo cual nos asegura una solución única con una cantidad única de intervalos. Ahora, si consideramos que pueden existir dos (o más) frecuencias, llamemos *A* y *B*, con el mismo precio podemos asumir indistintamente que *A*<*B* o *A*>*B* lo cual podría devolver intervalos distintos, pero siempre su cantidad va a estar acotada por lo mismo probado anteriormente: $2n - 1$.

Una vez que ya acotamos la cantidad final de intervalos por $2n - 1$, podemos proceder al análisis de complejidad del algoritmo de Divide & Conquer.

El algoritmo divide tiene complejidad *T(n)=2T(n/2)+O(conquer)*. Donde conquer va a recorrer, en el

peor caso, dos vectores cuyas longitudes son $(2n-1)/2$ cada uno, y aplicar operaciones que toman $O(1)$ para cada una de ellas. El vector que va construyendo con el CS toma $O(2n-1)$ que es igual a $O(n)$. Estas complejidades se suman y por propiedades de O se obtiene $O(n)$.

Reemplazando en la ecuación: *T(n)=2T(n/2)+O(n)*. Vemos que es la misma ecuación de recurrencia que el algortimo de MergeSort, por Teorema Maestro se deduce que tiene complejidad $O(n.log(n))$, como pretendíamos.

2.4. Código fuente

```
struct frecuencia{
    long int id;
    long int costo;
    long int principio;
    long int fin;
    bool operator< (const frecuencia& otro) const{
        if(costo == otro.costo){
            if(principio == otro.principio)
                return fin > otro.fin;
            return principio < otro.principio;
        }
        return costo < otro.costo;
    }
};
```

```
int main(int argc, char const *argv[]){
    chrono::time_point<chrono::system_clock> start, end;
    long int cantFrec;
    vector<frecuencia> frecuencias;
    //Leemos informacion del problema
    cin >> cantFrec;
    for (int i = 0; i < cantFrec; ++i){
        frecuencia actual;
        actual.id = i;
        cin >> actual.costo;
        cin >> actual.principio;
        cin >> actual.fin;
        if(actual.fin > actual.principio)
        frecuencias.push_back(actual);
    }
    start = chrono::system_clock::now();
    //Aplicamos el algoritmo
    vector<frecuencia> optimas = altaFrecuencia(frecuencias);
    vector<frecuencia>::iterator iter;
    //Calculamos el costo total (lo tuvimos en cuenta en la medicion de tiempos y complejidad)
    long int costoTotal = 0;
    for (iter = optimas.begin(); iter != optimas.end(); iter++){
        costoTotal += (iter->fin - iter->principio) * iter->costo;
    }
    end = chrono::system_clock::now();
    //Stdout pedido
    cout << costoTotal << endl;
    for (iter = optimas.begin(); iter != optimas.end(); iter++){
        cout << iter->principio << " " << iter->fin << " " << iter->id + 1 << endl;
    }
    cout << "-1" << endl;
    chrono::duration<double> elapsed_seconds = end-start;
    cout << "Tiempo: " << elapsed_seconds.count() << endl;
    return 0;
}
```

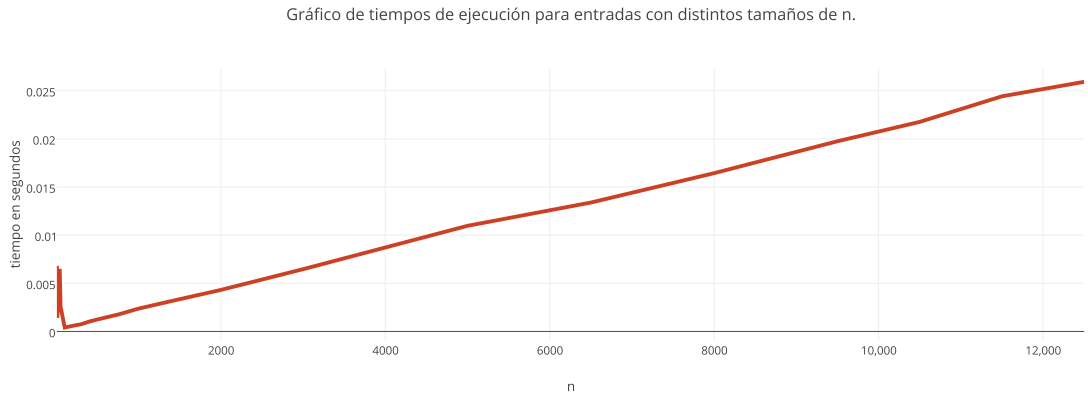
```
vector<frecuencia> altaFrecuencia(vector<frecuencia>& frecuencias){
    //Ordenamos el vector de menor a mayor costo de cada frecuencia
    sort(frecuencias.begin(), frecuencias.end());
    return divide(frecuencias, 0, frecuencias.size()-1);
}
```

```
vector<frecuencia> divide(vector<frecuencia>& frecuencias, long int comienzo, long int final){
    //Si tenemos dos o mas frecuencias, llamos a divide con cada una las mitades
    // y luego combinamos las soluciones
    if(final - comienzo > 0){
        vector<frecuencia> barata = divide(frecuencias, comienzo, (final+comienzo)/2);
        vector<frecuencia> cara = divide(frecuencias, ((final+comienzo)/2)+1, final);
        return conquer(barata, cara);
    }
    //Si solo hay una frecuencia,esta sera la forma mas barata y de maximo tiempo para transmitir
    else{
        vector<frecuencia> res;
        res.push_back(frecuencias[comienzo]);
        return res;
    }
}
```

```
vector<frecuencia> conquer(vector<frecuencia> barata, vector<frecuencia> cara){
    vector<frecuencia>::iterator iterCara = cara.begin(), iterBarata = barata.begin();
    vector<frecuencia> res;
    //mientras tenga frecuencias caras
    while(iterCara != cara.end()){
        // si todavia tengo baratas
        if(iterBarata != barata.end()){
            // si la cara empieza antes que la barata
            if(iterCara->principio < iterBarata->principio){
                // si la cara termina antes de que empiece la barata o al mismo tiempo
                if(iterCara->fin <= iterBarata->principio){
                    // la cara es parte de la solucion
                    res.push_back(*iterCara);
                    iterCara++;
                }
                // sino la cara empieza antes y termina despues del principio de la barata
                else{
                    // agrego el pedazo de cara que es solucion
                    // y le digo que su nuevo principio es el fin de la barata
                    frecuencia antes;
                    antes.id = iterCara->id;
                    antes.costo = iterCara->costo;
                    antes.principio = iterCara->principio;
                    antes.fin = iterBarata->principio;
                    res.push_back(antes);
                    iterCara->principio = iterBarata->fin;
                }
            }
            // sino la barata empieza antes o al mismo tiempo que la cara
            else{
                // si la cara termina despues que la barata
                if(iterCara->fin > iterBarata->fin){
                    // si la cara intersecta con la barata,
                    //le digo que su nuevo principio es el fin de la barata
                    if (iterCara->principio < iterBarata->fin)
                    iterCara->principio = iterBarata->fin;
                    // la barata es parte de la solucion
                    res.push_back(*iterBarata);
                    iterBarata++;
                }
                // si no salteo la cara, hay una o mas frecuencias para el tiempo
                //en que se puede utilizar que son mas baratas
                else
                iterCara++;
            }
        }
        // sino agrego todas las caras restantes a la solucion
        else{
            if(iterCara->principio < iterCara->fin)
            res.push_back(*iterCara);
            iterCara++;
        }
    }
    //si me quede sin caras, agrego las baratas restantes a la solucion
    while(iterBarata != barata.end()){
        res.push_back(*iterBarata);
        iterBarata++;
    }
    return res;
}
```

2.5. Experimentación

Constrastación Empírica de la complejidad



3. Problema 3: El señor de los caballos

3.1. Descripción de la problemática

En este problema, se presenta un tablero de ajedrez de tamaño nxn, el cual cuenta con alguna cantidad de caballos ubicados en una posición aleatoria del tablero. Lo que se quiere lograr es *cubrir* todo el tablero. Un casillero se considera cubierto si hay un caballo en él o bien, si es una posición en la cual algún caballo existente puede moverse con un sólo movimiento. Para lograr este cometido, puede ser necesario agregar nuevas fichas *caballo* al tablero. No existe un límite en la cantidad de caballos para agregar, pero lo que se busca es dar una solución agregando la menor cantidad de caballos posibles.

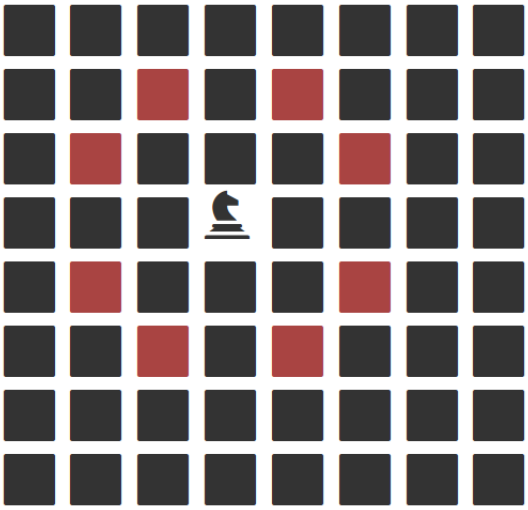


Figura 3: Casillas que *cubre* un caballo

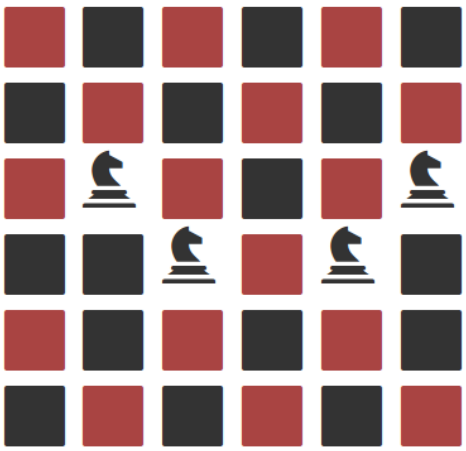


Figura 4: Así se ve un tablero en un posible estado inicial

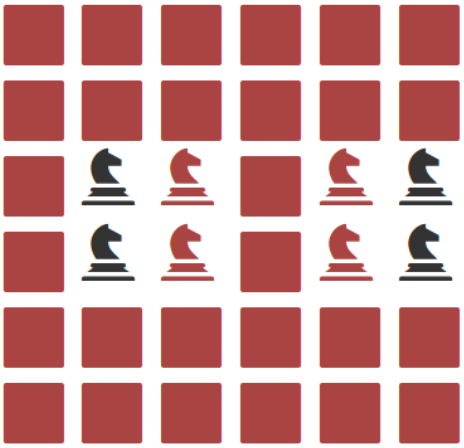


Figura 5: Y esta sería una de las soluciones óptimas donde se el agregan 4 caballos

3.2. Resolución propuesta y justificación

Para la resolución de este ejercicio, se pedía un algoritmo de backtracking. O sea que el algortimo debe, a partir de una subsolución del problema, fijarse si es solución y hacer algo al respecto con ella, o bien si es una subsolución válida, para cada elemento que se pueda agregar a la subsolución, agregarselo y repetir el proceso, o en el último caso, si se llega a una solución no válida, volver hacia atrás y tomar una decisión adecuada. Esto implica que el algortimo debe revisar un extenso árbol de posibilidades, donde una decisión es agregar o no un caballo en la posición i, j del tablero para todo i, j dentro del rango del mismo. Posterior a esto se pedía pensar e implementar una serie de podas y estrategias para no recorrer todo el árbol, sino mirarlo inteligentemente y así reducir los tiempos de ejecución.

El algortimo es muy sencillo, pregunta cuántos caballos hace falta agregar para cubrir el tablero si en la posición i, j agrega un caballo (recursivamente llena todo el tablero) y lo compara con la respuesta de no agregarlo, quedándose siempre con la solución que menos caballos extra le lleva.

El algoritmo resuelve el ejercicio planteado porque recorre todo el árbol de soluciones y se queda con alguna de las más óptimas.

Como que no pude cerrar la idea... AYUDA

3.3. Análisis de la complejidad

Para analizar la complejidad de este algoritmo, hay que tener en cuenta dos situaciones.

Sea n = dimensión del tablero

Si el tablero viene cubierto por los caballos preubicados, entonces el algoritmo chequea esto y devuelve que está completo en tiempo $O(n^2)$, es decir, recorrer la matriz donde está almacenado el tablero, posición por posición.

Si no empieza a trabajar con el backtracking. El primer approach para la resolución del ejercicio fue aplicar "fuerza bruta", dándonos una complejidad de $O(2^{n^2-k})$ siendo k la cantidad de caballos preubicados. Esto es para cada posición sin caballos, ver que pasa si tomo alguna de las dos posibles decisiones, o lo que es lo mismo, recorrer por completo el árbol de soluciones y devolver la de más óptima.

Para disminuir los tiempos de ejecución (en otra sección se hablará sobre esto) se pidió realizar podas al árbol y estrategias de recorrido (determinar si vale la pena o no seguir revisando alguna rama).

Ninguna poda/estrategia disminuye la complejidad teórica exponencial del primer approach, dado que el algoritmo realiza las mismas preguntas para cada posición del tablero, ¿qué pasa si lo dejo sin caballo? ¿qué pasa si le pongo un caballo?. No obstante, los tiempos de ejecución se ven radicalmente afectados, esto sucede porque se poda el árbol de soluciones posibles que se analizan, es decir, en el primer approach, se revisan todas las ramas, sin excepción, y aplicando podas descartamos muchas de estas que ya sabemos que no nos llevarán a un resultado que nos interese.

La primera poda fue la más intuitiva, si tenemos una solución con k caballos extra agregados, y analizando otra rama llegamos a tener que necesitar agregar un caballo a una subsolución de k-1 caballos (o sea que tendría por lo menos k caballos), entonces no nos interesa seguir revisándola, pues tenemos una solución que es igual o más óptima con k caballos.

La segunda estrategia fue plantear si en algún momento sabíamos que no debíamos agregar un caballo en un determinado casillero. Entonces, salteamos las k posiciones de los caballos preubicados y además salteamos aquellas posiciones que, estando atacadas, si le pusieramos un caballo, estarían atancando a casilleros que ya están siendo cubiertos por otros caballos.

3.4. Código fuente

3.5. Experimentación

Constrastación Empírica de la complejidad

-Hacer lo que hicieron en clase