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++.

Estos son los comandos para compilar cada ejercicio (el flag se utilizó para la librería ¡chrono¿para medir tiempos de ejecucion):
g++ -o main Zombieland.cpp -std=c++11
g++ -o main AltaFrecuencia.cpp -std=c++11
g++ -o main SenorCaballos.cpp -std=c++11

Dentro de cada .cpp está el comando para compilar cada ejercicio desde la carpeta donde se encuentran los mismos.

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 (o sea que cada ciudad sabe cuando iniciar el combate para no ser infectados). 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

Ver que habla mucho sobre vectores y dijo que no mezclaramos implementacion con algoritmo

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 no se pueda salvar a una ciudad, no se puede salvar a ninguna otra.

Como en este vector las ciudades no aparecen en orden creciente por su número, debemos reordenarlas. Debido a que el *id* de cada ciudad va a ser único y va a encontrarse en el intervalo [0, n-1], se lo recorre secuencialmente colocando cada ciudad en la posición de su *id* dentro de un nuevo vector *respuesta* . Ver que este parrafo me aprece que no quedo muy claro... es necesario este párrafo? habla del stdout... no me parece necesario en absoluto... yo lo sacaría

El algortimo resuelve el problema salvando la mayor cantidad de ciudades posibles porque FRAN NORIEGA – INSERT FORMAL DEMO

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
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

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...

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)***.

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

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;
    }
};
```

Poner como leemos y escribimos?? y el main? CREO QUE SI... no se, se puede dejar como apendice esto no? directamente lo imprimimos del sublime, toda la paja pasar todos los algoritmos a latex, con el main y las funciones extras y bla bla bla

Algorithm 2: ZombieLand(*out*: vector<ciudad2>; *in*: int cantCiudades, int presupuesto, vector<ciudad>& pais; *in/out*: int& salvadas)

```
salvadas = 0;
vector<ciudad2> datos;
for (int i = 0; i < cantCiudades; ++i) do
    ciudad2 actual;
    actual.numCiudad = i;
    double diferencia = (pais[i].zombies - pais[i].soldados * 10);
    if (diferencia > 0) then
        actual.soldadosNecesarios = ceil(diferencia/10);
    else
        actual.soldadosNecesarios = 0;
    actual.costoTotal = actual.soldadosNecesarios * pais[i].costo;
    datos.push_back(actual);
sort_heap(datos.begin(), datos.end());
for (int i = 0; i < cantCiudades; ++i) do
    int dif = presupuesto - datos[i].costoTotal;
    if (dif>=0) then
        salvadas++;
        presupuesto = dif;
    else
        datos[i].soldadosNecesarios = 0;
return datos;
```

No segui poniendo el algoritmo, por si le tenemos que hacer algun cambio.

1.5. Experimentación

Ver bien como generar ciudades aleatorias

1.5.1. Constrastación Empírica de la complejidad

-Comparar tiempos para ciudades de tamaño 10, 100, 1000, 10000 y 100000 generando aleatoriamente varias de cada una y scar promedio. Hacer lo que hicieron en clase

1.5.2. Modificación del algoritmo

Decidimos hacer una modificación dentro del algoritmo. Creemos que esta misma va a generar una mejora en la cantidad de tiempo.

La misma consiste en: modificar el último *ciclo for* para que cuando encuentre la primer ciudad que no pueda salvar deje de iterar ya que las restantes tampoco podrán ser salvadas (esto ocurre porque están ordenadas por costo).

Consideramos que saliendo antes del ciclo, el tiempo de cómputo para un mismo país debería ser menor que con el algoritmo original.

Experimentamos 10 casos distintos de tamaño 1000(? y para cada uno comparamos sus tiempos de ejecución entre el algoritmo original y el modificado.
Aca va el gráfico de eso que acabo de explicar arriba.

1.5.3. Comparar países de a dos

Consideramos 10?? casos, donde para cada uno tomamos dos paises (A y B) con la misma cantidad de ciudades para cada par de paises, cantidades distintas de ciudades (100, 500, 1000, ... (hasta donde nos de la compu jajaj)). Estos casos los vamos a armar de tal forma que A tenga ya todas sus ciudades salvadas al momento de ser ingresada, en cambio B tiene una relación de cantidad soldados/zombies aleatoria. Corremos el algoritmo para todos los pares de paises.

Dado a que nuestro algoritmo no establece ningún control de filtro sobre casos donde las ciudades ya esten salvadas al momento de ser recibidas como parámetro, consideramos que para cada par de ciudades los tiempos de cómputo no van a diferir mucho.

Aca va el gráfico donde muestra la diferencia entre cada par. Y ver si lo dibujamos linealizado como mostraron en clase tambien.

Aca hay que escribir si coincidió con lo que creiamos o no y dar una pequeña explicación de porque pasa eso

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.

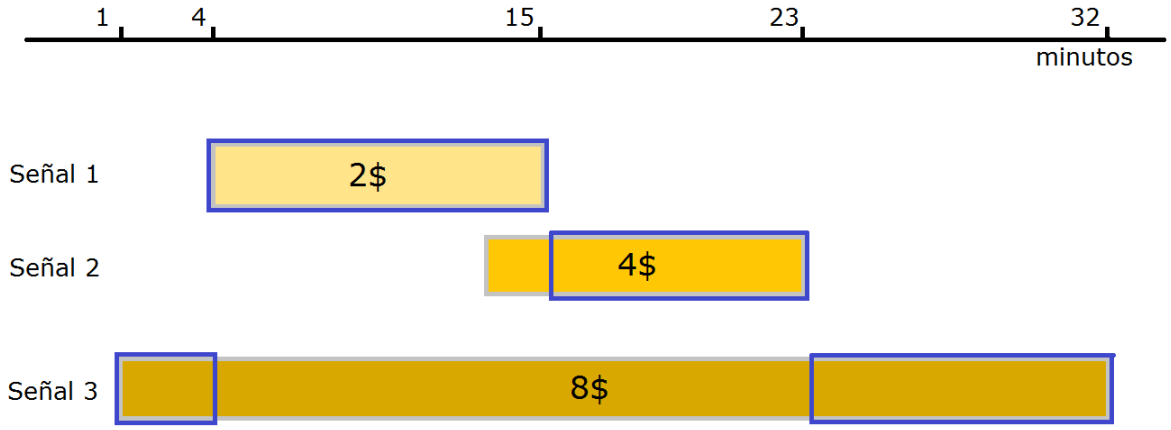

Figura 1: Ejemplo 1

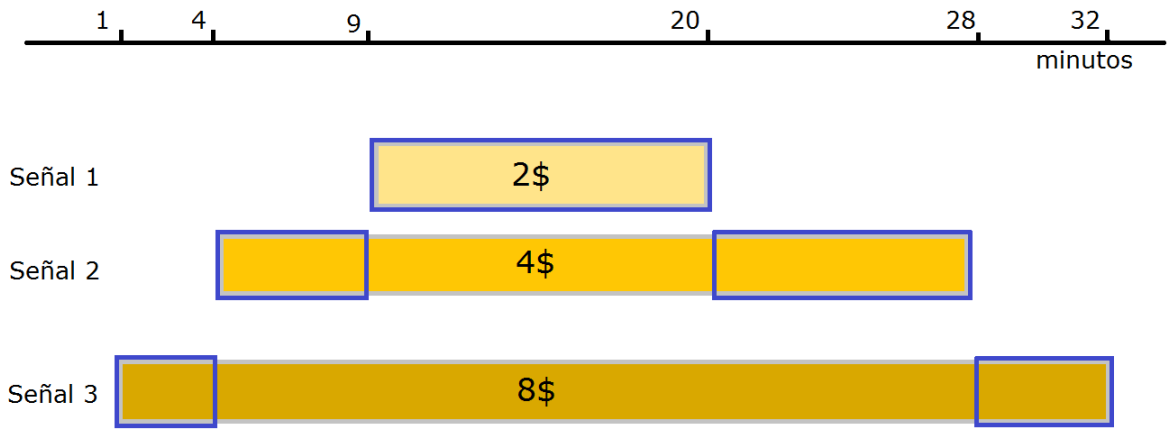

Figura 2: Ejemplo 2

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...

Todas las frecuencias se encuentran almacenadas en un vector. Primero se las ordena en orden creciente respecto del costo, es decir en la primera posición se almacena la más barata y en la última, la más cara. Una vez que contamos con este ordenamiento inicial, se va a proseguir mediante el *divide* dentro de este vector.

Se prosigue de acuerdo a las características de Divide & Conquer. De modo recursivo, se divide al vector pasado por parámetro por la mitad creando dos nuevos a los cuales se les aplica nuestro algoritmo de merge. El caso base se da cuando el vector tiene un sólo elemento, para lo cual se devuelve el vector tal cual entró.

```
divideAndConquer(conjuntoDeFrecuencias F){
    Si hay mas de un elemento:
        Divido F en dos conjuntos A, B.
        divideAndConquer(A)
        divideAndConquer(B)
        Devuelvo conquer(A,B)
    Si hay un solo elemento:
        Lo devuelvo.
}
```

Nuestro algoritmo de Merge (*conquer*) se encarga de elegir entre las frecuencias de dos arreglos pasados como parámetro, de modo que devuelve un sólo vector indicando los intervalos ocupados por las frecuencias elegidas.

Dado el enunciado del problema, para elegir qué frecuencia utilizar en determinado intervalo de tiempo se debe priorizar el precio más barato emitiendo señal siempre que sea posible. Gracias a que en el primer llamado de nuestra función estas se encontraban en orden creciente respecto del costo, en cada paso de *merge* de los dos arreglos de entrada con intervalos se van a priorizar los de la izquierda. Es decir que en cada paso, todos los intervalos pertenecientes al vector de la izquierda (como son los de menor precio) van a pertencer al vector resultado. Mientras que sólo los intervalos que completen tiempo sin señal pertenecientes al vector de la derecha van a ser colocados en el vector resultado. Esto representa un invariante que se va a cumplir en cada paso del algoritmo, lo cual nos permite justificar que la solución obtenida es la deseada. Que alguien lea esto para ver si se entiende, para mi si :)

```
conquer(conjuntoDeFrecuencias A, conjuntoDeFrecuencias B){
vector<frecuencia>::iterator iterCara = cara.begin(), iterBarata = barata.begin();
vector<frecuencia> res;

En el conjunto res voy a tener el resultado.
    Voy recorriendo en orden A y B, mientras haya elementos en A:
    //Llamo A[i] al intervalo actual de A y B[j] al de B.
        Si B[j] comienza antes que A[i]:
            Si B[j] termina antes que A[i]:
                Inserto B[j] en res.
                j++
            Si B[j] termina despues que A[i] empiece



        Si A[i] comienza antes que B[j]:
            Inserto A[i] en res
            Si B[j] se superpone en algun momento con A[i]:
                Al comienzo de B[j] le asigno el final de A[i]




while(iterCara != cara.end()){
if(iterBarata != barata.end()){
if(iterCara->principio < iterBarata->principio){ //la cara empieza antes
if(iterCara->fin <= iterBarata->principio){ //la cara termina antes de que empiece la barata
res.push_back(*iterCara); //meto la cara entera
iterCara++;
}
else{ //la cara empieza antes y termina despues del principio 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;
}
}
else{ //la barata empieza antes (o igual) que la cara
if(iterCara->fin > iterBarata->fin){ //la cara termina despues que la barata
if (iterCara->principio < iterBarata->fin) //este if es nuevo
iterCara->principio = iterBarata->fin;
res.push_back(*iterBarata);
iterBarata++;
}
else
iterCara++;
}
}
else{
if(iterCara->principio < iterCara->fin)
res.push_back(*iterCara);
iterCara++;
}
}
while(iterBarata != barata.end()){
res.push_back(*iterBarata);
iterBarata++;
}
return res;
}
```

Este algoritmo resuelve lo propuesto dando una solución óptima porque del modo en que está definido, primero divide al grupo con todas las frecuencias ofrecidas recursivamente hasta llegar a conjuntos con una sola frecuencia. Luego, al momento de mergear estos grupos de a dos, siempre prioriza al grupo de la izquierda (el más barato).

Esto quiere decir que en el primer paso de este *merge* se van a comparar solamente dos frecuencias entre sí, colocando la frecuencia más barata completa y; si la oferta de intervalos de la frecuencia más cara completa uno o dos intervalos donde no se le había asignado señal, se le otorga a ellos la frecuencia más cara. De este modo, comparamos de a dos, frecuencias consecutivas otorgándole prioridad a la frecuencia más barata. Poner dibujito.

En el siguiente paso del *merge* se van a comparar dos conjuntos de intervalos con dos frecuencias en cada uno. Cada uno de estos conjuntos puede estar conformado por uno, dos o tres intervalos. Poner dibujito Gracias al formato de nuestro algoritmo, en cada paso del merge se preserva el invariante de que el conjunto ubicado a la izquierda tiene las frecuencias más baratas mientras que el de la derecha tiene las más caras. Se prosigue de manera análoga a lo anterior, se preservan todos los intervalos de frecuencias pertenecientes al conjunto de los más baratos y sólo se agregan frecuencias del conjunto caro en caso de que esten disponibles en intervalos de tiempos que hayan quedado vacíos.

Estos pasos se van a reiterar, comparando conjuntos de intervalos entre dos, hasta que se hayan recorrido todas las frecuencias.

Una vez terminados todos los pasos recursivos, vamos a contar con el conjunto de intervalos que completen la mayor cantidad de tiempo con el costo más bajo. Si no hay dos frecuencias con el mismo costo, esta solución (óptima) es única, en caso contrario podría existir más de una solución óptima.

2.3. Análisis de la complejidad

Primero consideramos que nos encontramos en un caso donde la solución óptima es única. Si contamos con la oferta de *n* frecuencias, podemos asegurar que la cantidad de intervalos que va a contener la salida es de, a lo sumo, *2n-1*. Esta cota superior está dada porque, si consideramos agregar de a una las distintas frecuencias (empezando por las de menor costo), lo máximo que puede agregar son dos intervalos (o cubrir huecos de otros que no llegaron a llenar dos). Aca viene Eze y explica bien todo esto :D.

Por lo tanto, al hacer nuestro algoritmo de Divide & Conquer vamos a contar con, a lo sumo, 2n-1 intervalos. Esto nos otorga una complejidad de *O(n.log(n))* por el Teorema de ??????.

2.4. Código fuente

2.5. Experimentación

2.5.1. Constrastación Empírica de la complejidad

-Comparar tiempos para 10, 100, 1000, 10000 y 100000 intervalos generando aleatoriamente varios de cada una y scar promedio. Hacer lo que hicieron en clase

2.5.2. Modificación del algoritmo

Debido a que nuestro algoritmo no considera como caso específico el ordenar distintas frecuencias con el mismo valor, al momento de elegir intervalos se podrían realizar cortes innecesario. Es decir, no siempre se elige la cantidad mínima posible en estos casos.

Como consideramos que esto puede tardar un tiempo no despreciable, modificamos el operador $<$ para frecuencias, de modo que cuando tiene dos frecuencias del mismo valor ingrese primero la que tenga el comienzo antes y, en caso de comenzar en simultáneo, ingrese primero la frecuencia de mayor tamaño.

Aca poner el codigo que se deberia modificar.

Aca va el grafico de esto

Comentar si paso lo que esperabamos o no.

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 con la mínima cantidad de caballos posibles.

En la figura 3 se pueden ver todas las casillas que están cubiertas por un sólo caballo.

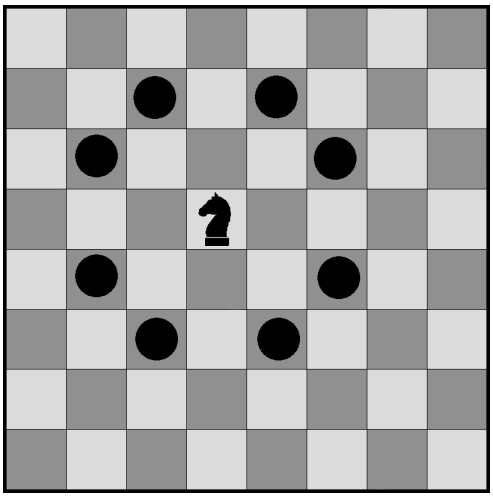

Figura 3: Casillas que *cubre* un caballo

3.2. Resolución propuesta y justificación

3.3. Análisis de la complejidad

3.4. Código fuente

3.5. Experimentación

3.5.1. Constrastación Empírica de la complejidad

-Hacer lo que hicieron en clase