



## 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 SeñorCaballos.cpp -std=c++11*

## Índice

<b>1. Problema 1: ZombieLand</b>	<b>3</b>
1.1. Descripción de la problemática . . . . .	3
1.2. Resolución propuesta y justificación . . . . .	5
1.3. Análisis de la complejidad . . . . .	6
1.4. Código fuente . . . . .	7
1.5. Experimentación . . . . .	9
1.5.1. Constrainación empírica de la complejidad . . . . .	9
<b>2. Problema 2: Alta Frecuencia</b>	<b>12</b>
2.1. Descripción de la problemática . . . . .	12
2.2. Resolución propuesta y justificación . . . . .	13
2.3. Análisis de la complejidad . . . . .	15
2.4. Código fuente . . . . .	19
2.5. Experimentación . . . . .	22
2.5.1. Constrainación Empírica de la complejidad . . . . .	22
<b>3. Problema 3: El señor de los caballos</b>	<b>24</b>
3.1. Descripción de la problemática . . . . .	24
3.2. Resolución propuesta y justificación . . . . .	25
3.3. Análisis de la complejidad . . . . .	26
3.4. Código fuente . . . . .	27
3.5. Experimentación . . . . .	31
3.5.1. Constrainación Empírica de la complejidad . . . . .	31

## 1. Problema 1: ZombieLand

### 1.1. Descripción de la problemática

En un país con  $n$  ciudades, se encuentra una determinada cantidad de Zombies y de Soldados por cada una de ellas; donde el problema es exterminar la invasión zombie. Para ello, es necesario un enfrentamiento *zombies vs soldados* por cada ciudad. Para que el combate sea viable 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 salvarlas. 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 la cantidad de soldados extra enviados a cada ciudad de modo que la cantidad de ciudades salvadas sea la óptima (es decir, la máxima) y gastando un monto por debajo o igual al presupuesto. El algoritmo debe tener una complejidad temporal de  $O(n \cdot \log(n))$ , siendo  $n$  la cantidad de ciudades del país.

A continuación se presenta un ejemplo sobre cómo resolver el problema, dado un caso específico.

Se nos presenta un país con cuatro ciudades ( $A$ ,  $B$ ,  $C$  y  $D$ ), el cual tiene un presupuesto a gastar de 100\$. Las ciudades están caracterizadas de la siguiente manera:



## Sección 1.1 Descripción de la problemática

---

En la siguiente tabla se muestran las diferentes elecciones que se pueden realizar al contratar soldados extra, teniendo en cuenta el presupuesto inicial, indicando cuántas ciudades son salvadas.

Casos	Cantidad de Soldados extra por ciudad				Presupuesto Gastado	Ciudades salvadas
	A	B	C	D		
Caso 1	1	0	0	0	45\$	A y B
Caso 2	2	0	0	0	90\$	A y B
Caso 3	1	0	1	0	65\$	A y B
<b>Caso 4</b>	<b>1</b>	<b>0</b>	<b>2</b>	<b>0</b>	<b>85\$</b>	<b>A, B y C</b>
Caso 5	0	1	0	0	100\$	B
Caso 6	0	0	1	0	20\$	B
Caso 7	0	0	2	0	40\$	B y C
Caso 8	0	0	1	1	100\$	B y D
Caso 9	0	0	0	1	80\$	B y D
Caso 10	0	0	0	0	0\$	B

Se puede observar que la ciudad *B* siempre se encuentra en el conjunto de Ciudades Salvadas, esto se debe a que al momento de ser ingresada la cantidad de zombies es menor (o igual) a diez veces la cantidad de soldados.

En este ejemplo, la solución buscada (la óptima) es la del Caso 4, si bien esta no es la que menor presupuesto gasta, es la que logra salvar la mayor cantidad de ciudades (3).

## 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 maximizar 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 `costo_total` será igual a 0.

Se recorren secuencialmente las ciudades ordenadas por `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 salvar la ciudad, se resta `costo_total` de `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  $O(n \cdot \log(n))$ , 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 ∈ 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
```

---

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()`<sup>1</sup> 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()`<sup>2</sup> cuya complejidad es  $O(n \cdot \log(n))$ .

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  $O(\text{ciudades\_salvadas})$  y el segundo  $O(n - \text{ciudades\_salvadas})$  dando como resultado un recorrido lineal sobre todas las ciudades, por lo tanto lo hace con complejidad  $O(n)$ .

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

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

---

<sup>1</sup><http://www.cplusplus.com/reference/vector/vector/push.back/>

<sup>2</sup><http://www.cplusplus.com/reference/algorithm/sort/>

## 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;
}
```

## Sección 1.4 Código fuente

---

```
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

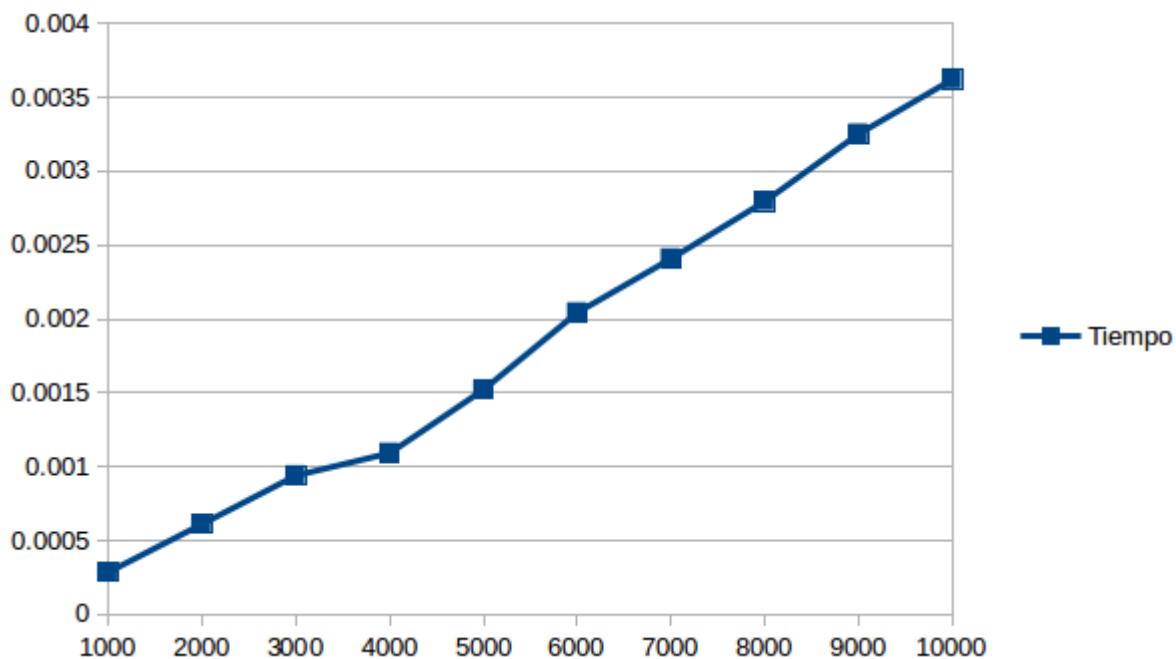
### 1.5.1. Contrastación empírica de la complejidad

Para realizar la contrastación empírica de la complejidad de nuestro algoritmo, decidimos comenzar con un muestreo aleatorio de ciudades y presupuestos, aumentando la cantidad de ciudades en cada prueba.

La siguiente tabla indica cuánto tiempo utilizó el algoritmo para resolver el problema, siendo  $n$  la cantidad de ciudades en cada caso.

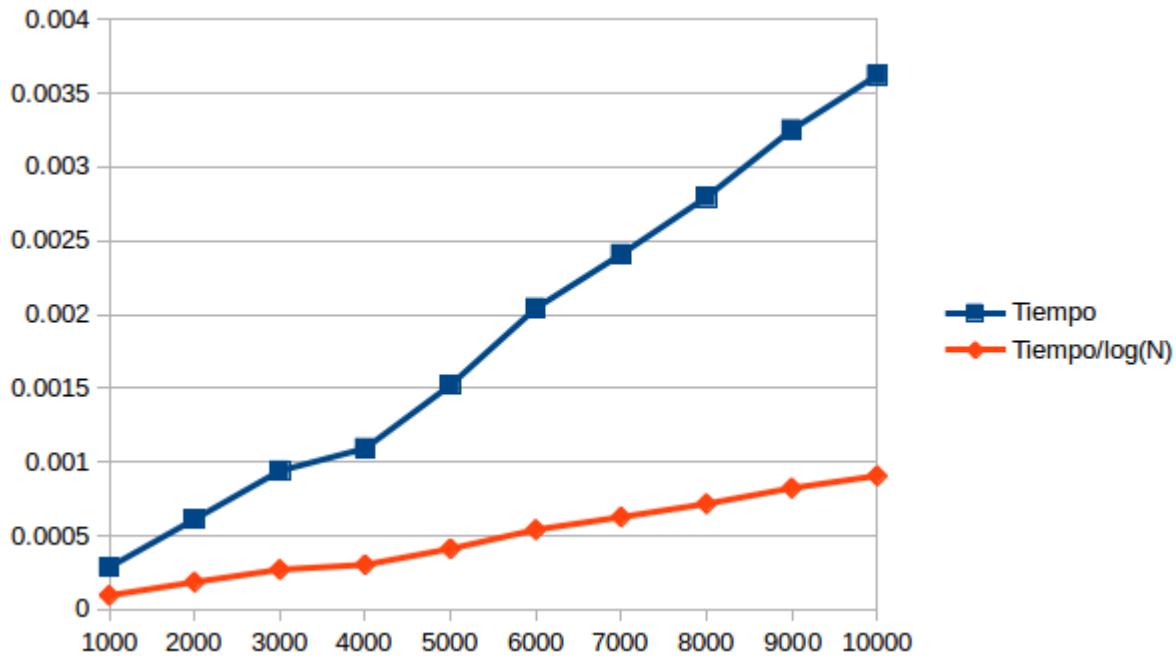
n	Tiempo en segundos
1000	0.0002860962
2000	0.0006116975
3000	0.0009399667
4000	0.0010920646
5000	0.0015213336
6000	0.0020432861
7000	0.0024080363
8000	0.0027960707
9000	0.0032532473
10000	0.0036220356

El gráfico resultante fue el siguiente:



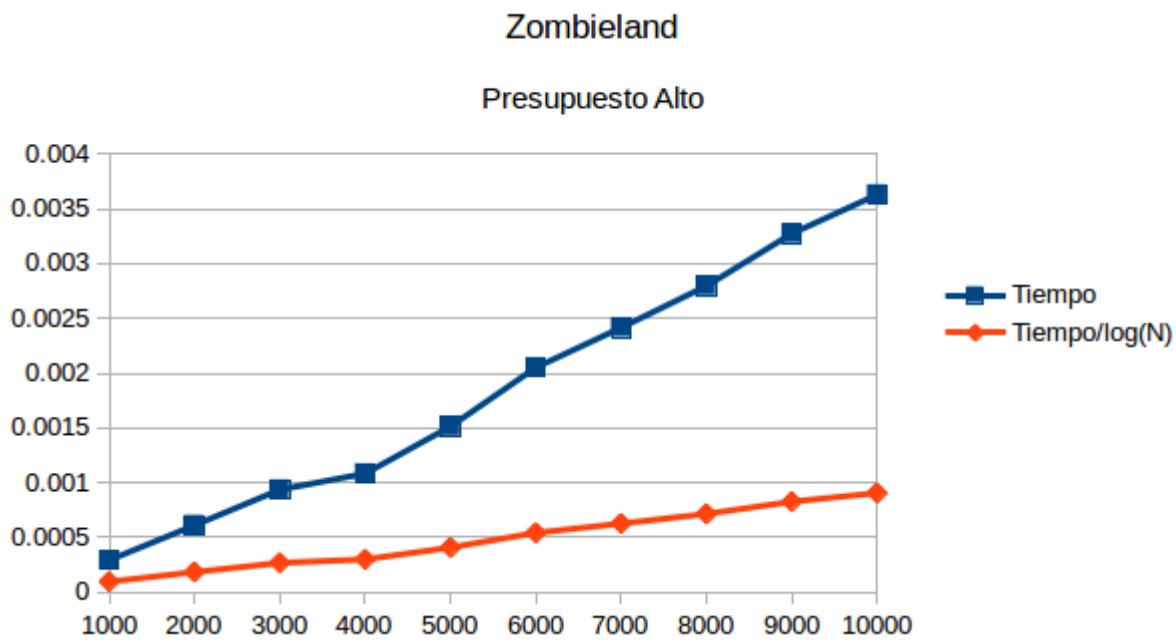
De acuerdo al análisis de este gráfico, podemos ver que aparenta ser una curva con poca inclinación, similar al gráfico de  $O(n * \log(n))$ , que fue la complejidad propuesta.

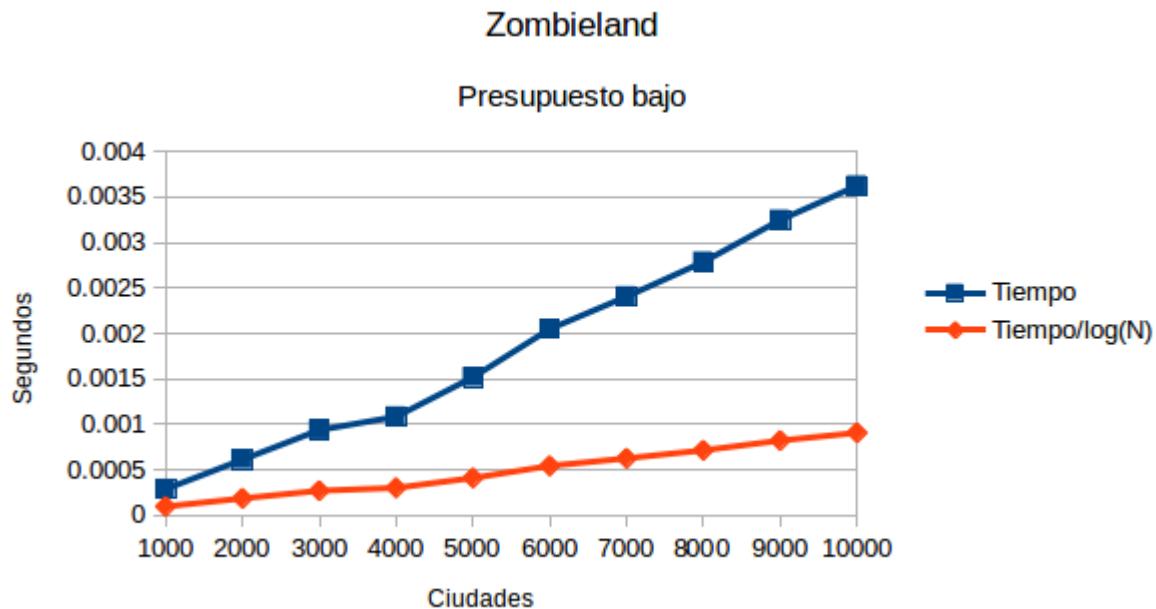
Sin embargo, esto no es suficiente para probar que nuestro algoritmo cumple la complejidad. Por lo tanto, se realizó la linealización, dividiendo los tiempos por  $\log(n)$ .



En esta comparación, podemos ver que al dividir la curva inicial por  $\log(n)$ , se obtiene prácticamente una función lineal, lo cual era esperado.

Luego realizamos una nueva prueba, para comprobar que el algoritmo solamente depende de la cantidad de ciudades, sin importar cuál es el presupuesto asociado. Como en todos los casos vamos a realizar el `sort()`, que tiene complejidad  $O(n * \log(n))$ , el gráfico debiera ser similar.





Efectivamente, utilizando tanto presupuestos altos como bajos, la curva es siempre similar.

## 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 instantáneamente (del minuto 1 al 4 uso la frecuencia A y del 4 al 6 la B). Los datos del precio e intervalo de tiempo de cada frecuencia son dados. Se desea encontrar una solución óptima a este problema, es decir, transmitir todo el tiempo que se tenga al menos una frecuencia abierta, pero gastando la menor cantidad de dinero. El algoritmo que resuelva dicho problema debe tener una complejidad algorítmica 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.

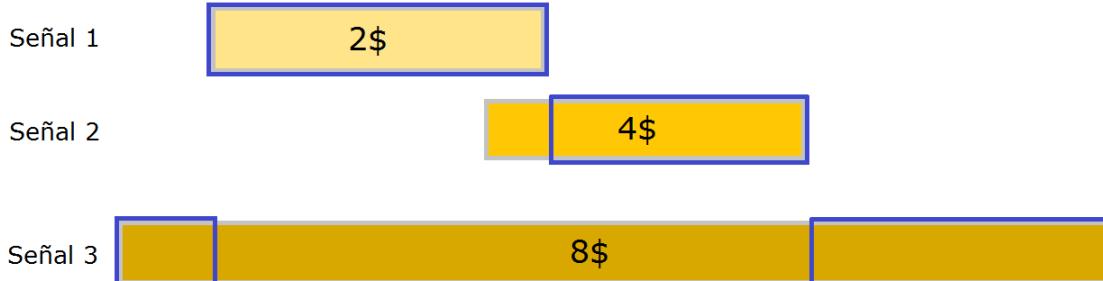
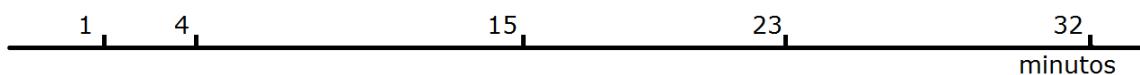


Figura 1: Ejemplo 1

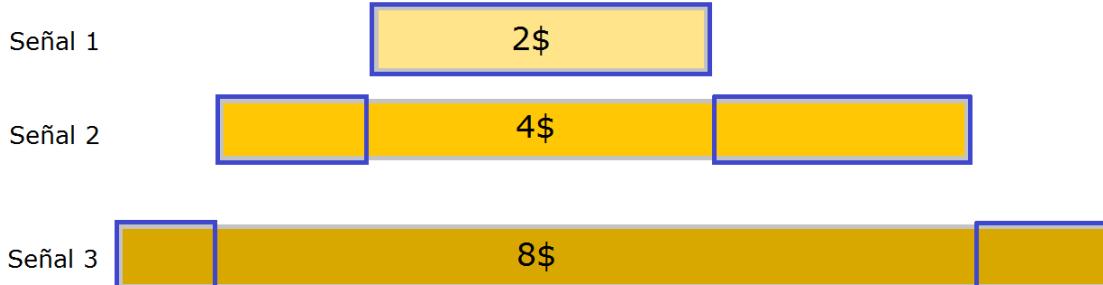
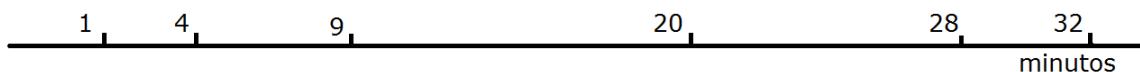


Figura 2: Ejemplo 2

## 2.2. Resolución propuesta y justificación

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

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 esto no ocurre, se seguirá subdividiendo hasta llegar al caso en el que hay solo un intervalo. Es importante aclarar (pues es fundamental para el funcionamiento del algoritmo), que intervalosA tendrá la mitad de frecuencias más baratos, e intervalosB, la mitad de frecuencias más caros. Esto ocurre pues las frecuencias fueron ordenadas por su costo antes de comenzar con este tramo del algoritmo

Si esto no ocurre, 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 características, 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 comenzar con este tramo de algoritmo.

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

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 (es decir, se toman la frecuencia más barata a cada momento, y se hacen los cortes pertinentes a cada frecuencia, según corresponda. Véase *Ejemplo 1* para ver como se recortan las frecuencias).

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 *conquer* 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 pertenecer 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 (es decir, se completarán los vacíos entre frecuencias del primer vector, con frecuencias o intervalos de frecuencias de éste vector). 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.

Este último paso hace uso exhaustivo del invariante: todos los intervalos del conjunto intervalosA deben pertenecer al conjunto solución, y lo único que se 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

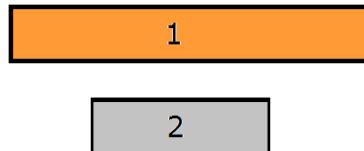
Para realizar este análisis primero es necesario calcular cuántos intervalos contendrá, como máximo, el conjunto solución (desde ahora “CS”). Este valor será  $2n - 1$ , siendo  $n$  la cantidad de frecuencias dadas como parámetro.

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.

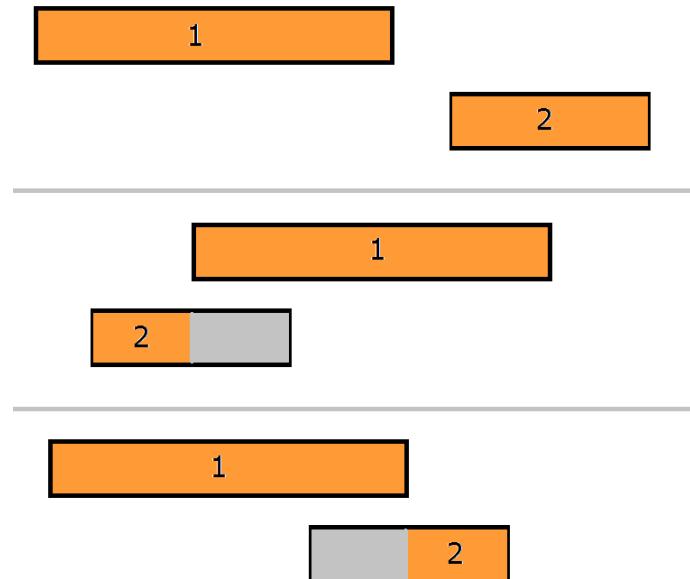
Supongamos **n=1**. El intervalo de la frecuencia pasada como parámetro será el único elemento del CS y verifica  $2n - 1 = 2.1 - 1 = 1$ .

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.

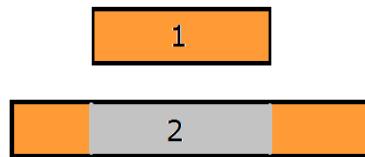
La frecuencia 2 no forma ningún intervalo:



La frecuencia 2 forma un intervalo:

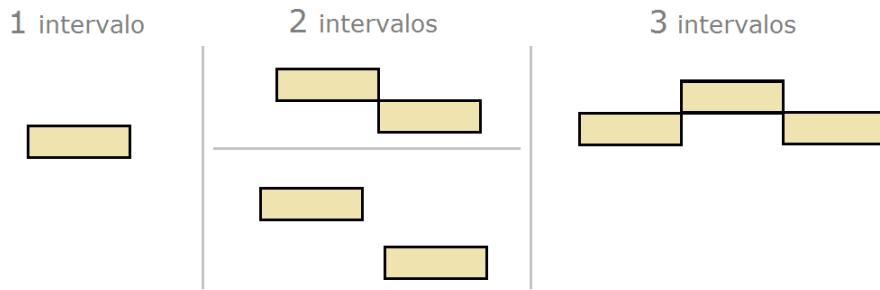


La frecuencia 2 forma dos intervalos:



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

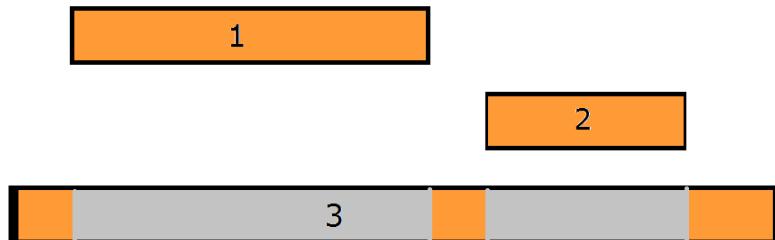
Al momento de tomar  $n=3$ , lo hacemos considerando las frecuencias 1, 2 y 3. Partimos del caso anterior al que le añadimos la frecuencia 3. Es decir, vamos a contar con a lo sumo 3 intervalos. La distribución de los intervalos se va a dar de la siguiente manera:



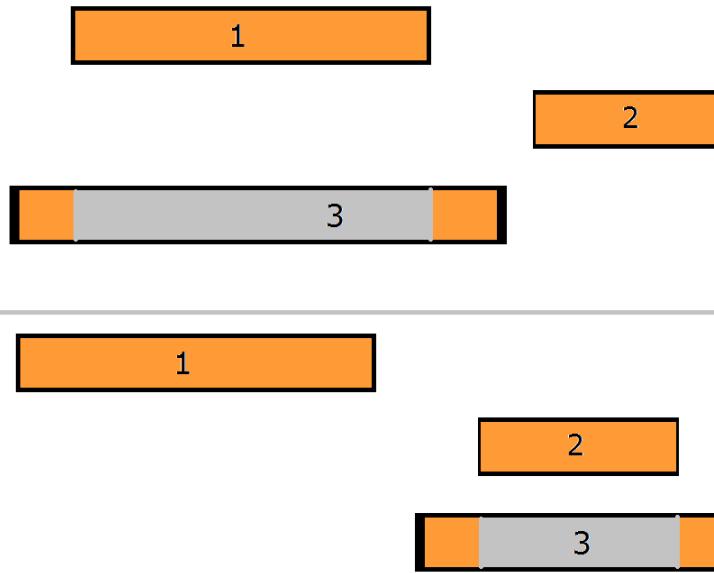
Podemos observar que de las cuatro distribuciones distintas que pueden ocurrir, en sólo una los intervalos son completamente disjuntos, esto es en el caso donde se añadió el intervalo 2 completo. En los demás casos, si bien pueden ser uno, dos o tres intervalos distintos al unirlos conforman un intervalo 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:  $2n - 1 = 2 \cdot 3 - 1 = 5$ .

Si extendemos este cálculo para cualquier  $n$ , vamos a concluir que siempre la cantidad de intervalos va a estar acotada por  $2n - 1$  ya que se puede reducir el problema a cualquiera de las situaciones mencionadas.

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* partitiona el problema en dos subproblemas más chicos, en particular de la mitad de tamaño que el problema original, lo que implica que tiene complejidad  $T(n) = 2T(n/2) + O(\text{conquer})$  (ecuación que surge de llamar recursivamente a la función *divide*), 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 obtenemos:  $T(n) = 2T(n/2) + O(n)$ . Vemos que es la misma ecuación de recurrencia que el algoritmo de MergeSort, que 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, llamamos 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;
}

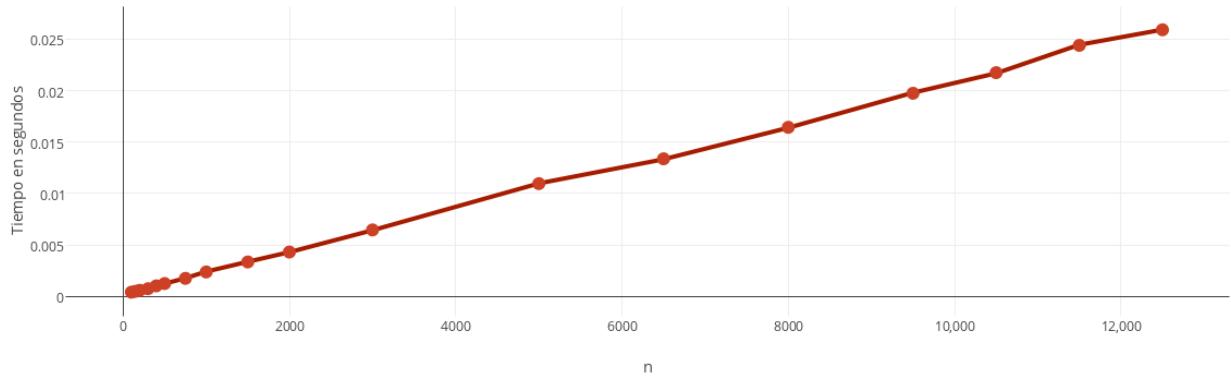
```

## 2.5. Experimentación

### 2.5.1. Contrastación Empírica de la complejidad

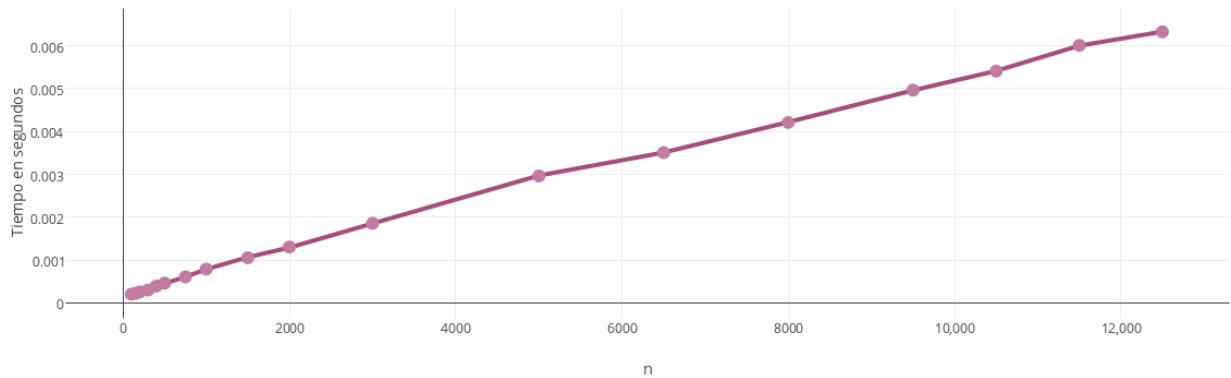
n	Tiempo en segundos
100	0.000415645
150	0.00049622
200	0.000595058
300	0.000752836
400	0.001029706
500	0.001253788
750	0.001757269
1000	0.0023792
1500	0.003352018
2000	0.004313126
3000	0.006464969
5000	0.010964701
6500	0.013377069
8000	0.01643833
9500	0.0197472
10500	0.021750165
11500	0.024404865
12500	0.02591816

Gráfico de tiempos de ejecución para entradas con distintos tamaños de n



n	Tiempo en segundos dividido por $\log_2(n)$
100	0.000207823
150	0.000228033
200	0.000258605
300	0.000303916
400	0.000395727
500	0.000464543
750	0.000611211
1000	0.000793067
1500	0.001055391
2000	0.0013066
3000	0.001859288
5000	0.002964258
6500	0.003508359
8000	0.00421162
9500	0.004964447
10500	0.005408889
11500	0.006010017
12500	0.00632627

Gráfico de tiempos de ejecución linealizados



### 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  $n \times n$ , el cual cuenta con cierta 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.

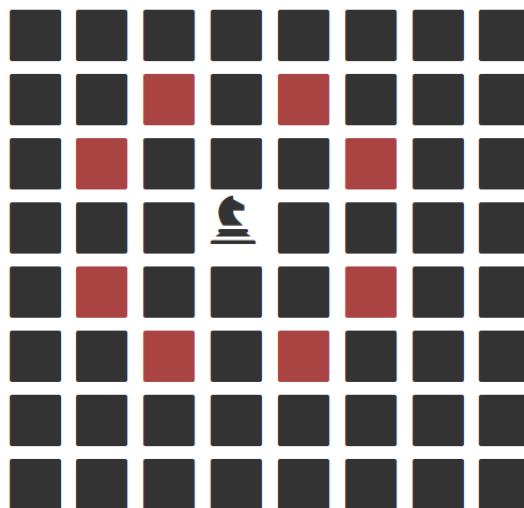


Figura 3: Casillas que *cubre* un caballo

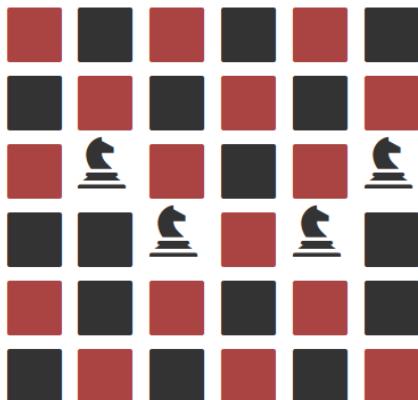


Figura 4: Así se ve un tablero dado un posible estado inicial

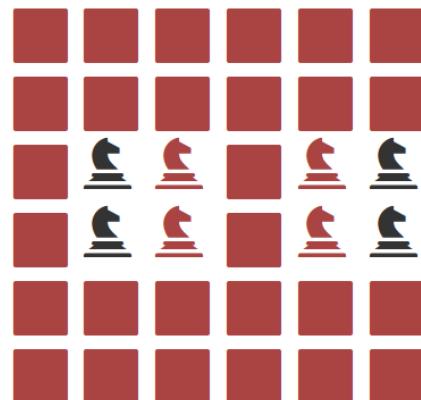


Figura 5: Una de las soluciones óptimas donde se le agregan 4 caballos

### 3.2. Resolución propuesta y justificación

Para la resolución de este ejercicio, se pedía un algoritmo de backtracking. O sea que el algoritmo 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.

Bajo el contexto de un algoritmo de Backtracking se divisan a las distintas estrategias para resolver un determinado problema como distintos caminos desde la raíz hasta una hoja dentro de un árbol de decisiones. En nuestro caso, cada camino va a representar cada una de las distribuciones posibles de fichas *caballo* en las casillas marcadas como *vacías* al comienzo de la ejecución.

Vamos a recorrer desde la rama que no ubica ninguna pieza de caballo, hasta la que ubica caballos en todas las posiciones vacías, secuencialmente.

Cuando llegamos al final de cada camino (una hoja) no tenemos asegurado haber *cubierto* el tablero, es decir no todas las ramas van a ser solución. Pero si lo es, debemos comparar con las soluciones encontradas anteriormente, preservando siempre la que utilizó una menor cantidad de fichas.

Habiendo recorrido todo el árbol, nos aseguramos de encontrar alguna solución, en particular la más óptima. Esto implica que el algoritmo debe revisar un (muy) 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.

Para inicializar la búsqueda, necesitamos guardar una solución del tablero que sea máxima en la cantidad de caballos agregados. Esta es, particularmente, llenar el tablero.

Luego el algoritmo 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, guardando en el contenedor de soluciones antes mencionado, la solución más óptima hasta ese momento.

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

### 3.3. Análisis de la complejidad

El algoritmo tiene una complejidad mínima de  $O(n^2)$  (siendo  $n$  la dimensión del tablero) porque crea un primer vector de solución óptima con “agregar todos los caballos posibles al tablero”

Continuando este algoritmo, hay que tener en cuenta dos situaciones.

#### Primera Situación:

El tablero pasado como parámetro de entrada se encuentra cubierto completamente con las fichas existentes.

Nuestro algoritmo sólo debe chequear que el tablero esté cubierto. Lo realiza en tiempo  $O(n^2)$ , dado que debe recorrer la matriz donde está almacenado el tablero, posición por posición.

Que sumado a la complejidad anterior da como resultado  $O(n^2)$

#### Segunda Situación:

El tablero pasado como parámetro de entrada *no* se encuentra cubierto completamente con las fichas existentes.

Es en este caso donde se comienza a trabajar con el *Backtracking*. La primer resolución llevada a cabo fue aplicar “fuerza bruta”. Esto consistió en recorrer por completo el árbol de soluciones y devolver una que fuera óptima. Para ello, en cada posición sin caballos, debemos ver que pasa si tomo alguna de las dos posibles decisiones (insertar un caballo o no). Este proceso cuenta con una complejidad de  $O(2^{n^2-k})$  siendo  $k$  la cantidad de caballos preubicados. Agregar o quitar un caballo toma tiempo  $O(1)$  y por cada posición del tablero ( $n^2$ ) que esté libre ( $n^2 - k$ ), elige entre dos opciones, o sea  $2 \times 2 \times \dots \times 2 O(n^2 - k)$  veces.

Con el fin de disminuir los tiempos de ejecución, se pidió realizar podas al árbol y estrategias de recorrido (determinar si vale la pena o no seguir revisando alguna rama).

Ninguna poda o estrategia disminuye la complejidad teórica exponencial del primer caso estudiado, dado que el algoritmo realiza las mismas preguntas para cada posición del tablero: ¿qué pasa con el resto del tablero si a esta posición la dejo sin caballo? y ¿qué pasa con el resto del tablero si a esta posición le agrego un caballo?.

No obstante, los tiempos de ejecución se ven radicalmente afectados (ver Sección: 3.5.1), esto sucede porque se poda el árbol de soluciones posibles que se analizan. Es decir, en un primer momento se revisaban todas las ramas, sin excepción; pero aplicando podas, descartamos muchas de estas que ya sabemos que no nos llevarán a un resultado de interés.

La poda fue la más intuitiva: mientras vamos recorriendo el árbol contamos con una solución (preservada) que presenta  $k$  caballos extra agregados. Si analizando otra rama nos encontramos en una posición donde debemos agregar un caballo pero la cantidad de caballos, para este camino, iguala a  $k$  no vamos a estar interesados en proseguir por esta rama ya que, a lo sumo, vamos a encontrar otra solución óptima con  $k$  caballos.

La estrategia fue plantear, para cada posición a analizar, si agregar un caballo cubre alguna casilla que estaba libre (contando la suya). Si no lo hace, podemos establecer que este caballo no se va a encontrar en el Conjunto Solución de esa rama.

Entonces, no solo salteamos las  $k$  posiciones de los caballos preubicados, sino que también salteamos aquellas posiciones que, estando atacadas, si le pusieramos un caballo, estarían atacando a casilleros que ya están siendo cubiertos por otros caballos.

### 3.4. Código fuente

```
struct casillero{
    bool esCaballo;
    long int ataques;
};

struct coordenadas{
    unsigned int fila;
    unsigned int col;
};

int main(int argc, char const *argv[]){
    chrono::time_point<chrono::system_clock> start, end;
    long int cantCaballos;
    cin >> dimension;
    cin >> cantCaballos;
//Creamos un tablero sin caballos
    Tablero tablero(dimension, Vec(dimension));
    for (int i = 0; i < dimension; ++i){
        for (int j = 0; j < dimension; ++j){
            casillero actual;
            actual.esCaballo = false;
            actual.ataques = 0;
            tablero[i][j] = actual;
        }
    }
//Leemos informacion del problema
    int fila;
    int col;
    for (int i = 0; i < cantCaballos; ++i){
        cin >> fila;
        cin >> col;
//Agregamos los caballos y las posiciones a las que ataca
        tablero[fila-1][col-1].esCaballo = true;
        atacame(tablero, fila-1, col-1, 1);
    }
    start = chrono::system_clock::now();
//Aplicamos el algoritmo
    vector<coordenadas> sol = senorCaballos(tablero);
    end = chrono::system_clock::now();
//Stdout pedido
    cout << sol.size() << endl;
    for (int i = 0; i < sol.size(); ++i){
        cout << sol[i].fila+1 << " " << sol[i].col+1 << endl;
    }
    chrono::duration<double> elapsed_seconds = end-start;
    cout << "Tiempo: " << elapsed_seconds.count() << endl;
    return 0;
}
```

```

vector<coordenadas> senorCaballos(Tablero& t){
    //Creamos un vector de n*n, con n = dimension del tablero,
    //la peor manera de cubrir un tablero es colocando un caballo en cada casillero"
    coordenadas aca;
    vector<coordenadas> optimo;
    for (int i = 0; i < dimension; ++i){
        for (int j = 0; j < dimension; ++j){
            aca.fila = i; aca.col = j;
            optimo.push_back(aca);
        }
    }
    //Creamos el vector solucion
    vector<coordenadas> agregados;
    senorCaballosAux(t, 0, 0, agregados, optimo);
    return optimo;
}

```

```

void atacame(Tablero& t, int fila, int col, int ataco){
    //ataco = 1 ataca, ataco = -1 desataca
    if(col-2 >= 0){
        if(fila-1 >= 0) t[fila-1][col-2].ataques += ataco;
        if(fila+1 < dimension) t[fila+1][col-2].ataques += ataco;
    }
    if(col+2 < dimension){
        if(fila-1 >= 0) t[fila-1][col+2].ataques += ataco;
        if(fila+1 < dimension) t[fila+1][col+2].ataques += ataco;
    }
    if(fila-2 >= 0){
        if(col-1 >= 0) t[fila-2][col-1].ataques += ataco;
        if(col+1 < dimension) t[fila-2][col+1].ataques += ataco;
    }
    if(fila+2 < dimension){
        if(col-1 >= 0) t[fila+2][col-1].ataques += ataco;
        if(col+1 < dimension) t[fila+2][col+1].ataques += ataco;
    }
}

```

```

bool chequeo(const Tablero& t){
    for (int i = 0; i < dimension; ++i){
        for (int j = 0; j < dimension; ++j){
            if(!t[i][j].esCaballo && t[i][j].ataques == 0){
                return false;
            }
        }
    }
    return true;
}

```

```

int senorCaballosAux(Tableo& t, int i, int j, vector<coordenadas>& agregados,
vector<coordenadas>& optimo){
//si el tablero esta cubierto, dadas las podas será una solución optima,
//la guardamos y retornamos cuantos caballos agregamos
    if(chequeo(t)){
        optimo.assign(agregados.begin(),agregados.end());
        return optimo.size();
    }
//sino si estoy dentro del tablero
    int siAgrego, siNoAgrego;
    if(i<dimension){
//    si hay un caballo preubicado, salteo el espacio
//    si la posicion esta atacada y de agregar un caballo no cubre ninguna previamente cubierta,
//    no es lo que busco
        if(t[i][j].esCaballo || (!sirveAregar(t, i, j) && t[i][j].ataques > 0)){
            j++;
            if(j==dimension){
                j=0; i++;
            }
            return senorCaballosAux(t, i, j, agregados, optimo);
        }
//    sino backtracking
        else{
//        si tengo un optimo de k caballos, y llegue a una subsolucion de k-1 caballos que aun
//        no cubre todo el tablero, no es lo que busco
            if(agregados.size() >= optimo.size()-1)
                return -1;
//        PRUEBO SIN AGREGAR UN CABALLO
//        si no termine la fila
            if(j<dimension-1)
                siNoAgrego = senorCaballosAux(t, i, j+1, agregados, optimo);
//        si tengo que cambiar de fila
            else
                siNoAgrego = senorCaballosAux(t, i+1, 0, agregados, optimo);
//        PRUEBO AGREGANDO UN CABALLO
            t[i][j].esCaballo = true;
            atacame(t, i, j, 1);
            coordenadas aca; aca.fila = i; aca.col = j;
//        lo agrego a la posible solucion
            agregados.push_back(aca);
//        si no termine la fila
            if(j<dimension-1)
                siAgrego = senorCaballosAux(t, i, j+1, agregados, optimo);
//        si tengo que cambiar de fila
            else
                siAgrego = senorCaballosAux(t, i+1, 0, agregados, optimo);
//        RESTAURO el tablero y el vector con los caballos agregados
            t[i][j].esCaballo = false;
            atacame(t, i, j, -1);
            agregados.pop_back();
        }
//        si no tienen solucion agregando y sin agregar, devuelvo que no hay solucion
        if(siAgrego == -1 && siNoAgrego == -1)
            return -1;
        if(siNoAgrego == -1 || siAgrego < siNoAgrego)
            return siAgrego;
        return siNoAgrego;
    }
//sino devuelvo que no hay solucion
    return -1;
}

```

```

bool sirveAregar(const Tablero& t, int fila, int col){
    if(col-2 >= 0){
        if(fila-1 >= 0 && t[fila-1][col-2].ataques == 0 && !(t[fila-1][col-2].esCaballo))
            return true;
        if(fila+1 < dimension && t[fila+1][col-2].ataques == 0 && !(t[fila+1][col-2].esCaballo))
            return true;
    }
    if(col+2 < dimension){
        if(fila-1 >= 0 && t[fila-1][col+2].ataques == 0 && !(t[fila-1][col+2].esCaballo))
            return true;
        if(fila+1 < dimension && t[fila+1][col+2].ataques == 0 && !(t[fila+1][col+2].esCaballo))
            return true;
    }
    if(fila-2 >= 0){
        if(col-1 >= 0 && t[fila-2][col-1].ataques == 0 && !(t[fila-2][col-1].esCaballo))
            return true;
        if(col+1 < dimension && t[fila-2][col+1].ataques == 0 && !(t[fila-2][col+1].esCaballo))
            return true;
    }
    if(fila+2 < dimension){
        if(col-1 >= 0 && t[fila+2][col-1].ataques == 0 && !(t[fila+2][col-1].esCaballo))
            return true;
        if(col+1 < dimension && t[fila+2][col+1].ataques == 0 && !(t[fila+2][col+1].esCaballo))
            return true;
    }
    return false;
}

```

```

void imprimir(const Tablero& tablero){
    for (int i = 0; i < dimension; ++i){
        for (int j = 0; j < dimension; ++j){
            cout << tablero[i][j].esCaballo << "(" << tablero[i][j].ataques << ")";
        }
        cout << endl << endl;
    }
}

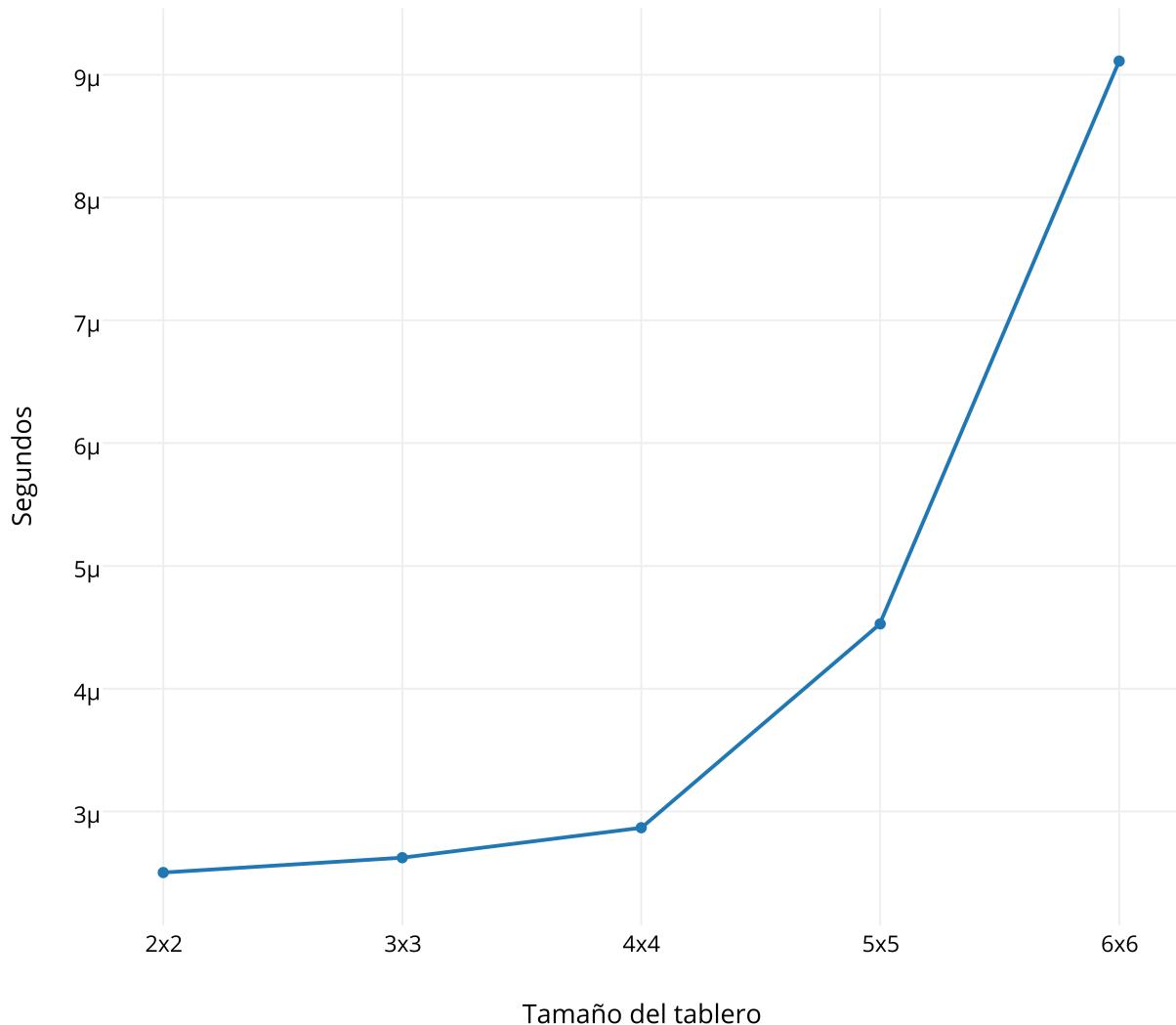
```

### 3.5. Experimentación

#### 3.5.1. Constrainación Empírica de la complejidad

Primero analizaremos el caso en que los tableros vienen cubiertos como entrada al problema

Promedio de tiempos de ejecución para tableros cubiertos



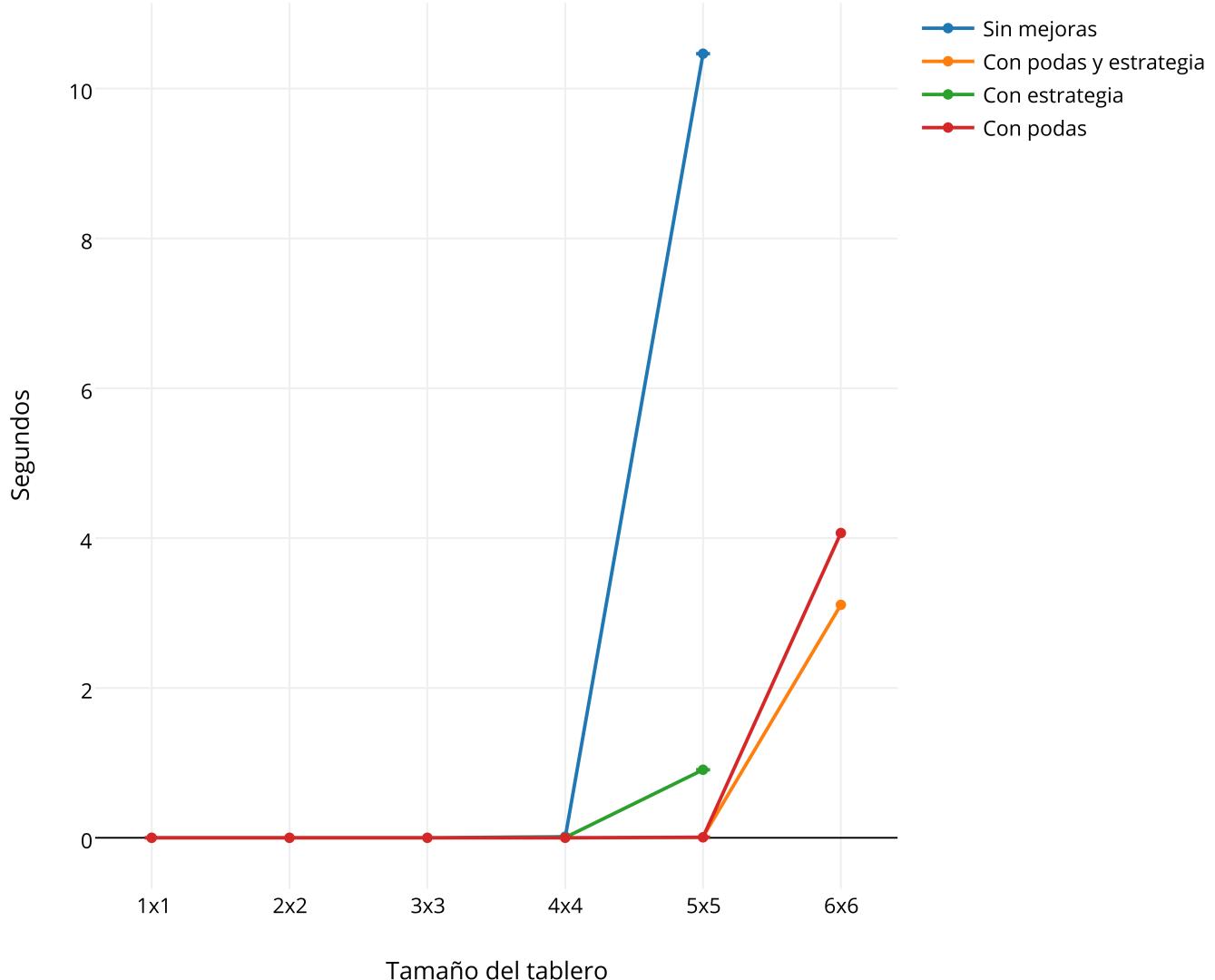
Como se ve en el gráfico, para tableros de mayor dimensión, se incrementan los tiempos de ejecución. Estos aumentos aparentan seguir una curva, que en principio parece la estimada en el análisis de complejidad de  $O(n^2)$ .

En segundo lugar vamos a ver qué sucede con los tiempos de ejecución para tableros con 0, 2 y 4 caballos preubicados, para distintos tamaños de tablero. Cada gráfico corresponde a los tiempos de ejecución del algoritmo bruto, con poda, con estrategia y con ambas situaciones a la vez para las distintas cantidades de caballos preubicados.

Nota: para los tableros de 6x6 no tomamos mediciones en el caso de sin poda y solo estrategia porque resultaban demasiado altos.

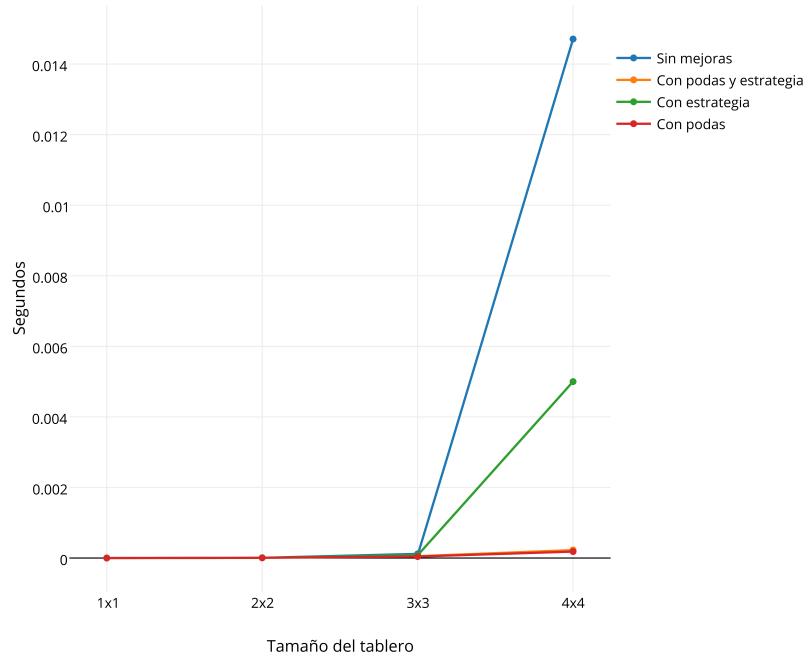
## Tablero vacío

Promedio de tiempos de ejecución para tableros sin caballos



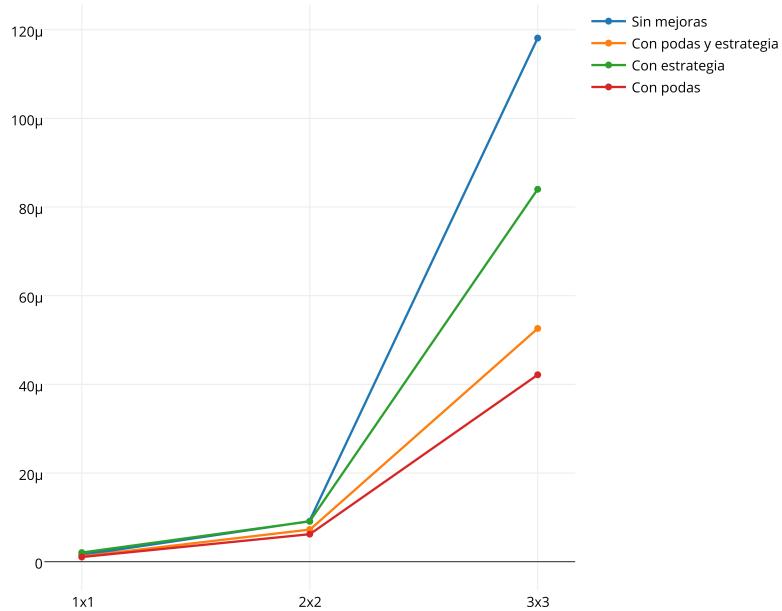
Este gráfico muestra que a medida que aumenta el tamaño del tablero, los tiempos aumentan. No obstante, en los tableros más chicos no se puede apreciar una diferencia significativa en cuanto a sus mediciones de tiempo. Por este motivo, haremos zoom al gráfico.

Ampliación para apreciar los tableros más pequeños



Aún más zoom...

Ampliación para apreciar los tableros más pequeños



Como era de esperar, el algoritmo que realiza el *Backtracking completo* es el que tarda más tiempo en ejecutarse, para cualquier tamaño de tablero.

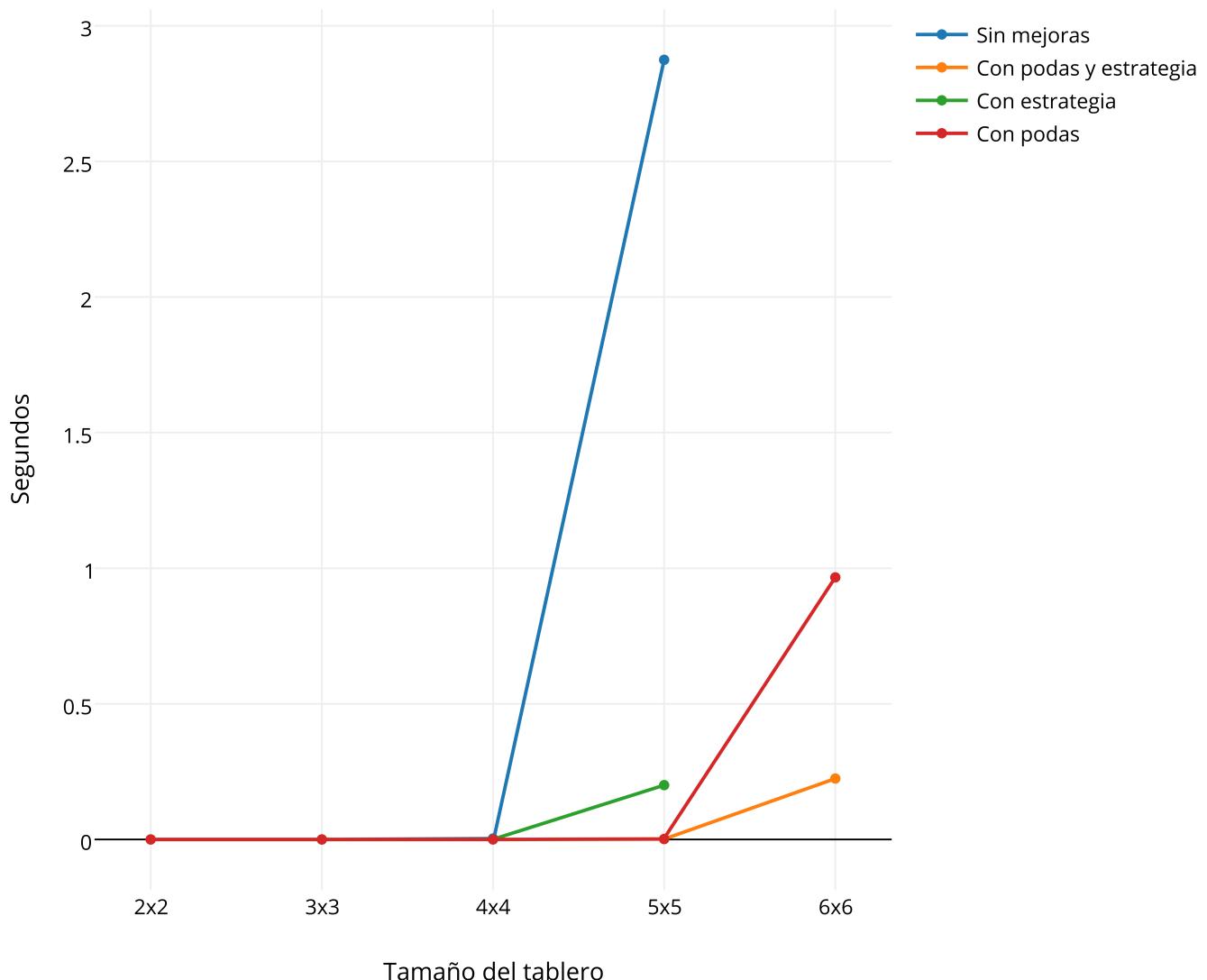
Además podemos ver que aplicar la estrategia (solamente) es menos eficiente que podar el árbol ya que sus mediciones de tiempo se preservan por encima de las dos curvas restantes.

Al momento de comparar los dos casos con podas sucede que para los tableros más pequeños las mediciones de tiempo oscilan entre sí. Esto se debe a que para tableros chicos calcular si agregar un caballo es una buena idea o no (que es el procedimiento que se lleva a cabo en la estrategia) carece de sentido ya que hay poca probabilidad de que un caballo ataque otras posiciones del tablero que no estén cubiertas. En la mayoría de los casos, va a ser mejor agregar un caballo y después ver qué pasa.

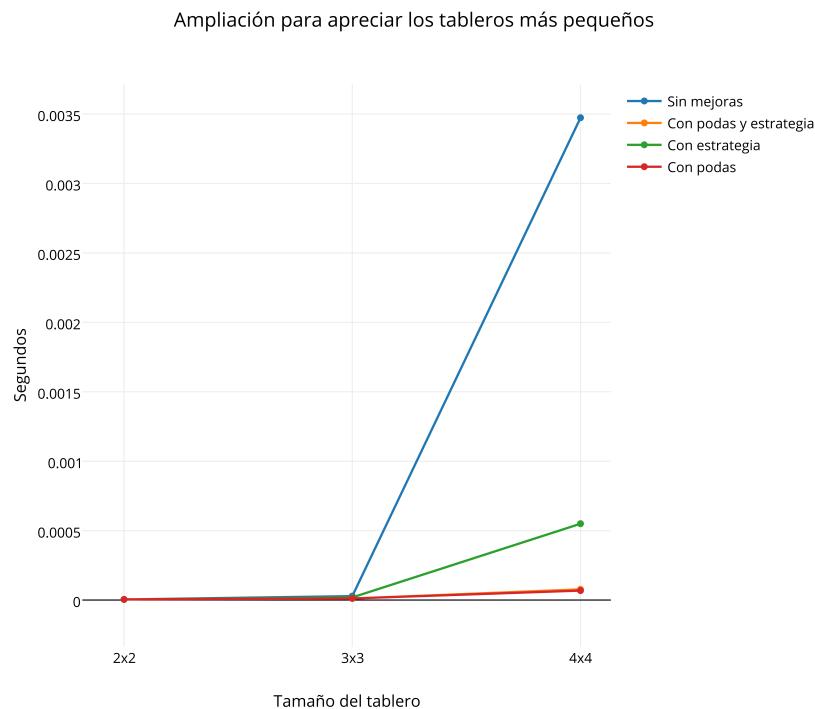
Pero al aumentar el  $n$ , los tiempos de ejecución de realizar podas sin estrategia siempre son mayores. En estos casos lo que ocurre es que llevar a cabo nuestra estrategia conviene ya que se eliminan muchas ramas del árbol de un sólo paso.

## Tablero con dos caballos iniciales

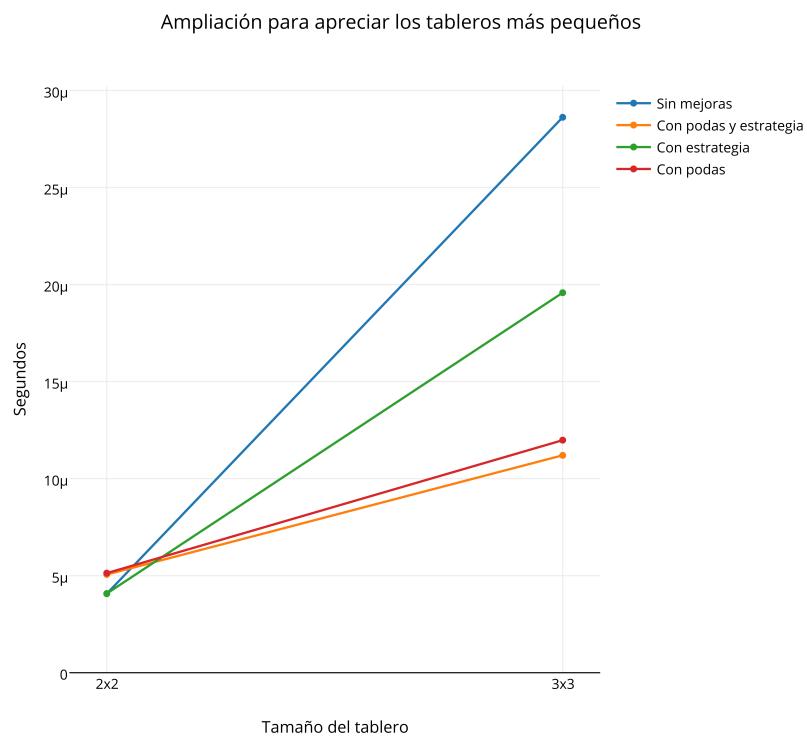
Promedio de tiempos de ejecución para tableros con 2 caballos



Haciendo zoom

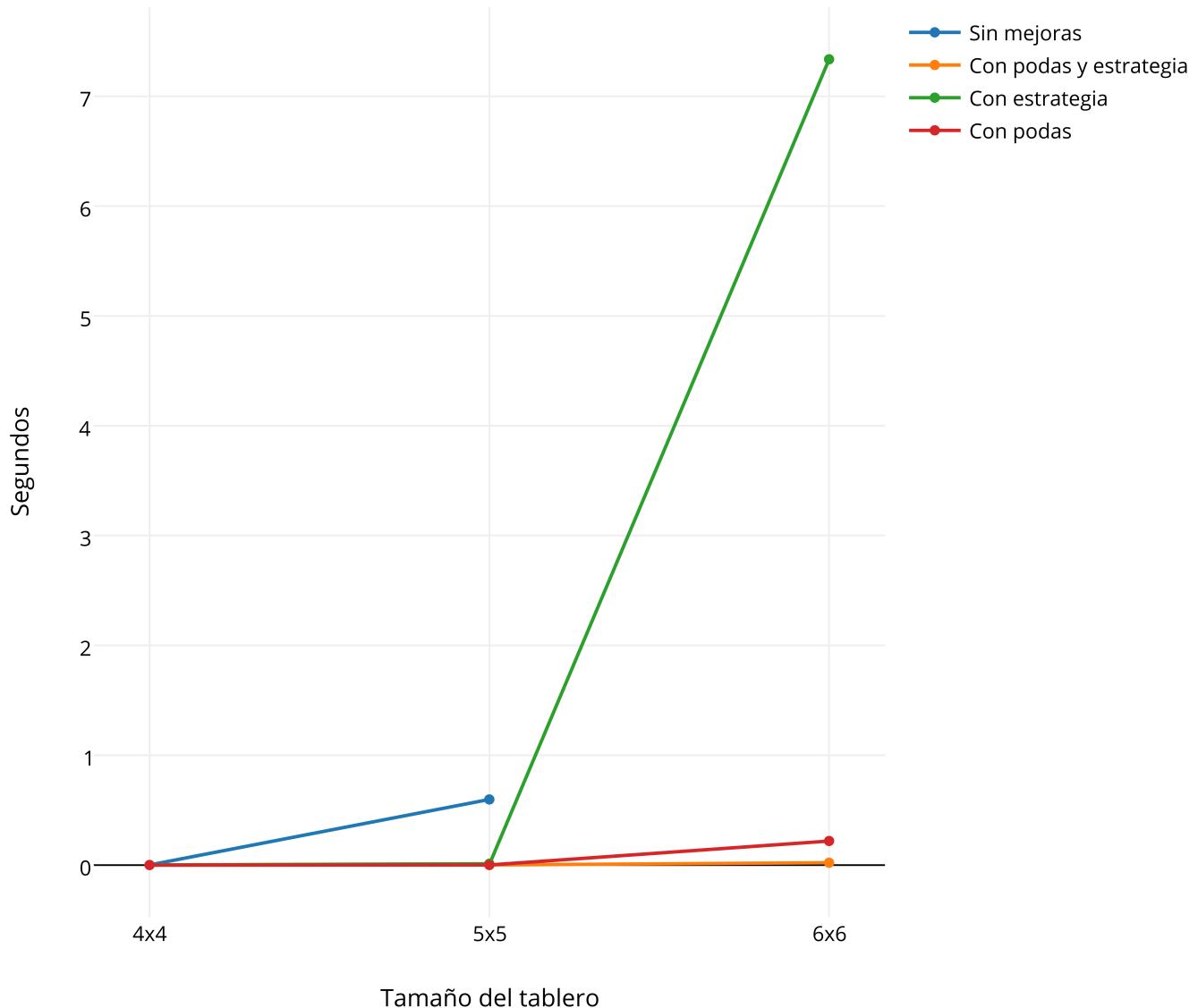


Aún más zoom...



## Tablero con cuatro caballos iniciales

Promedio de tiempos de ejecución para tableros con 4 caballos

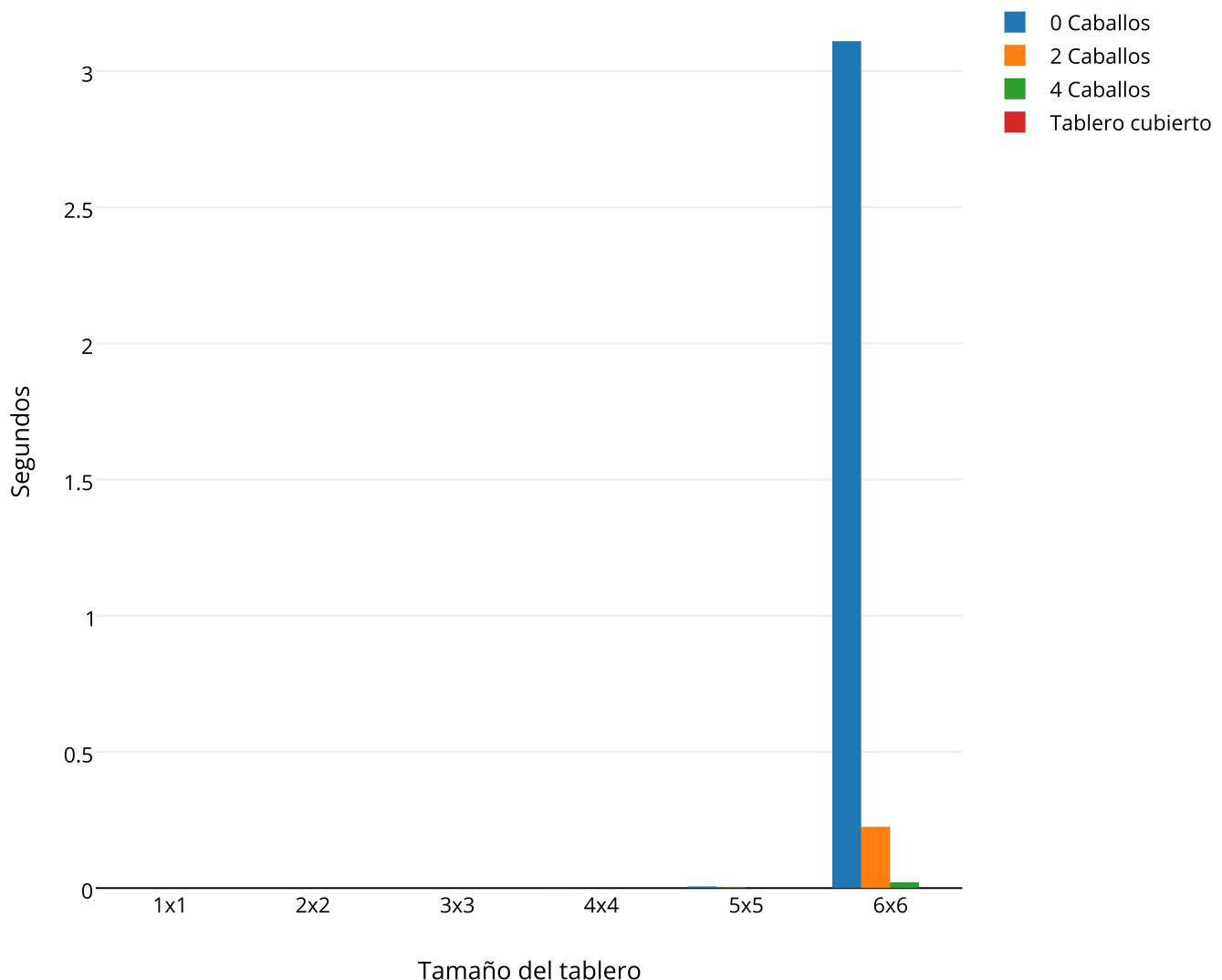


Se puede apreciar que al tratar con un tablero de entrada con dos o cuatro caballos preubicados, sin estar este cubierto, los tiempos de ejecución para cada método preservan la relación analizada en el tablero vacío.

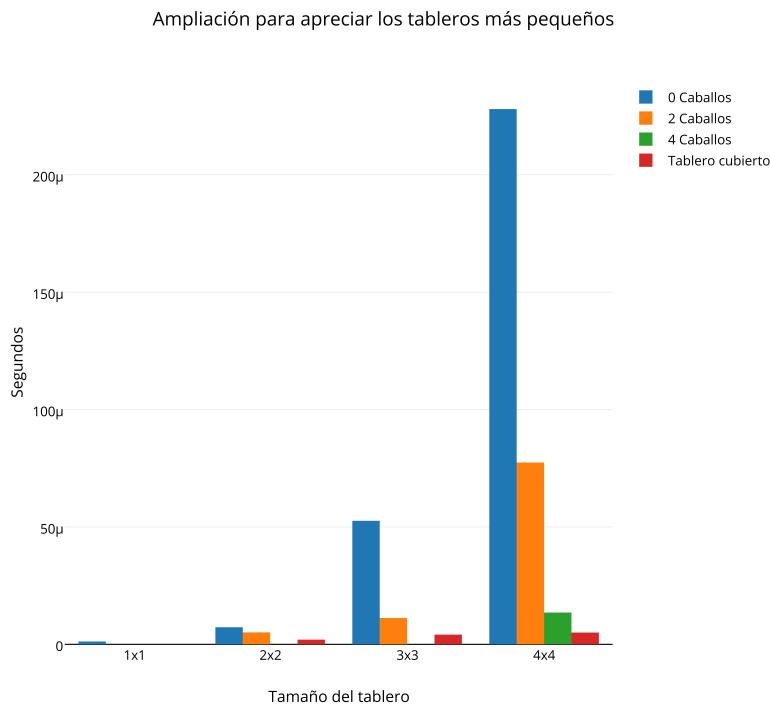
Como último paso, queremos verificar si el  $k$ , perteneciente a nuestra cota de complejidad ( $O(2^{n^2} - k)$ ), ejerce una influencia notoria en los tiempos de ejecución.

Para comprobar esto, hicimos gráficos de barras indicando cuánto tardó el algoritmo con Podas y Estrategia (a la vez) para distintos tamaños de  $n$  comparando la cantidad de caballos inicial.

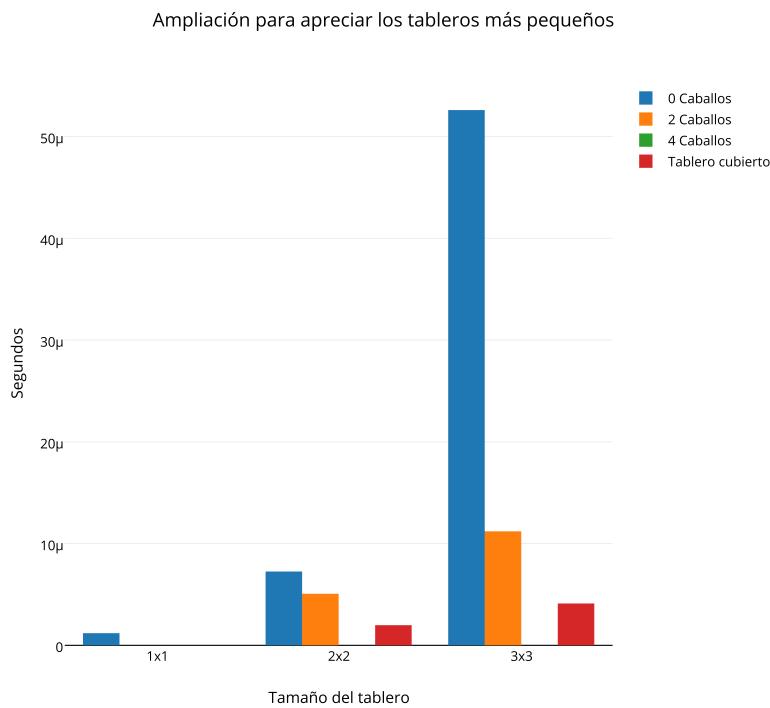
Promedio de tiempos de ejecución según los caballos dados



### Haciendo zoom



Aún más zoom...



Se puede apreciar que, independientemente del tamaño del tablero a utilizar, la relación de tiempos de ejecución es siempre la misma: el caso que menos tiempo tarda es el caso trivial (cuando el tablero está ocupado al inicio) y luego el tiempo de ejecución se incrementa acorde disminuye la cantidad de caballos preubicados.