DEPARTAMENTO DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico I

Algoritmos y Estructura de Datos III

Primer Cuatrimestre de 2015

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

Facultad de Ciencias Exactas y Naturales

Universidad de Buenos Aires

Ciudad Universitaria - (Pabellón I/Planta Baja)

Intendente Güiraldes 2160 - C1428EGA

Ciudad Autónoma de Buenos Aires - Rep. Argentina

Tel/Fax: (54 11) 4576-3359

http://www.fcen.uba.ar

Índice

1. Ejercicio 1 - Zombieland

1.1. Problema a resolver

El problema a resolver consiste en salvar la mayor cantidad de ciudades de la invasión zombie. Para lograr tal fin, se busca lanzar un ataque coordinado, utilizando los soldados que ya estan en cada ciudad y enviando refuerzos a las mismas, cuya cantidad esta limitada por el presupuesto inicial.

Para salvar una ciudad la proporción zombie-soldado debe ser de, al menos, 1 soldado por cada 10 zombis. La cantidad de zombis y soldados de cada ciudad es conocida, como también el costo de enviar un soldado a la misma.

Se busca un algoritmo que, tomando el estado actual de todas las ciudades del país, y un presupuesto, optimize el envio de soldados a cada ciudad para salvar la mayor cantidad de ciudades, detallando cuantas ciudades son salvadas y cuantos soldados se envía a cada ciudad del país.

- Ejemplo: Situación inicial.

```
5 ciudades, $ 600 presupuesto
ciudad 1: 10 zombis 2 soldados $ 500 costo p/soldado
ciudad 2: 20 zombis 2 soldados $ 400 costo p/soldado
ciudad 3: 30 zombis 2 soldados $ 300 costo p/soldado
ciudad 4: 40 zombis 2 soldados $ 200 costo p/soldado
ciudad 5: 50 zombis 2 soldados $ 100 costo p/soldado
```

- Solución del ejemplo:

```
4 ciudades salvadas (todas menos la 4)
ciudad 1: 0 soldados enviados (salvada, $ 0 costo)
ciudad 2: 0 soldados enviados (salvada, $ 0 costo)
ciudad 3: 1 soldados enviados (salvada, $ 300 costo)
ciudad 4: 0 soldados enviados (no es salvada, $ 400 costo)
ciudad 5: 3 soldados enviados (salvada, $ 300 costo)
```

1.2. Resolución planteada

La solución planteada utiliza la `técnica algorítmica golosa`. La idea de la misma es ordenar las ciudades en orden creciente en un vector según el `costo de salvar` cada ciudad e ir recorriendo el vector restando al presupuesto disponible (P) `el costo de salvar` cada ciudad, hasta que el mismo se

agote o todas las ciudades sean salvadas, guardando en un contador la cantidad de ciudades salvadas y en un vector del mismo tamaño que la cantidad de ciudades (n), en el `indice de la ciudad`, los `soldados a enviar` a cada ciudad (este vector es inicializado en 0) . El `costo de salvar` cada ciudad se calcula a partir de $\lceil \frac{z}{10} \rceil$, donde z es la cantidad de zombis en la ciudad, restándole a ese resultado la cantidad de soldados (s) en la ciudad, y por último, multiplicar el resultado previamente obtenido por el costo de enviar refuerzos (c) a la ciudad. En caso de que ese valor sea negativo, se pone en 0. La cantidad de `soldados a enviar` se calcula de la misma manera que el `costo de salvar` cada ciudad, sin hacer la ultima multiplicación. El `indice de la ciudad` se refiere al renglón en donde aparece la ciudad en el input. Por último se imprime por pantalla el contador de ciudades salvadas y el vector que contiene los `soldados a enviar`. En pseudocódigo:

```
Creamos un vector infoCiudad de tamaño igual a n.
Creamos un vector soldadosPorCiudad de tamaño n, inicializado en 0.
Para cada ciudad:
        Tomamos su z, s y c, y
        Hacemos soldados a enviar = parte alta de la división z / 10  - s
        costo de salvar = soldados a enviar * c,
        Si costo de salvar es menor a 0,
                Hacemos soldados a enviar y costo de salvar = 0.
        Insertamos cada ciudad en el vector infoCiudad
Ordenamos el vector infoCiudad segun costo de salvar.
Para todo i desde 1 hasta n:
         Si P es mayor o igual al costo de salvar la i-ésima ciudad
                  Incrementamos el contador de ciudades salvadas
                  Guardamos en soldadosPorCiudad[i] la cantidad de soldados a enviar
                  Hacemos P = P - costo de salvar la i-ésima ciudad
Imprimimos el contador ciudades salvadas y luego el vector soldadosPorCiudad
```

1.2.1. Demostración formal de la solución

Teorema

El algoritmo goloso propuesto encuentra siempre una solución óptima, donde solución óptima se refiere a maximizar la cantidad de ciudades que podrían recuperarse con el presupuesto disponible (P).

Demostración

Sea $\pi_i =$ `costo de salvar` la ciudad i, es decir:

$$\pi_i = \begin{cases} (\lceil \frac{z_i}{10} \rceil - s_i) * c_i & : s_i \leq \lceil \frac{z_i}{10} \rceil \\ 0 & : s_i > \lceil \frac{z_i}{10} \rceil \end{cases}$$

Probamos que dada una solución óptima cualquiera, el algoritmo genera una solución con la misma cantidad de ciudades rescatadas (puede contener ciudades distintas), es decir que la solución generada por el algoritmo goloso es óptima. Para esto, suponemos por el absurdo que la solución dada por el algoritmo goloso no es óptima y tomamos la solución óptima que mas ciudades rescatadas comparte con la solución del algoritmo goloso.

Sean I las ciudades rescatadas por esta solución óptima y J las ciudades rescatadas por el algoritmo goloso. Suponemos que las ciudades están ordenadas por π (de menor a mayor). En caso de empate, se ordenan por su orden de aparición.

Sea I_j la primera ciudad de I distinta a las ciudades de J (existe pues supuse que J no es óptimo), es decir que $I_i = J_i$ para todo $i < j$. Por lo tanto, se cumple que $\pi_{I_j} \geq \pi_{I_{j-1}}$. Pero entonces como también se cumple que $\pi_{J_j} \geq \pi_{J_{j-1}}$, y J_j fue elegido como aquel que tiene el menor π (de las ciudades no incluidas en J_i para todo $i < j$), y $I_{j-1} = J_{j-1}$, debe suceder que J_j sea menor o igual a I_j. Con lo

cual si en I reemplazo I_j por J_j, seguimos salvando la misma o mayor cantidad de ciudades, ya que el costo acumulado es igual o menor al anterior costo acumulado, por lo que debe ser óptimo.

Pero este nuevo conjunto de ciudades, tiene más ciudades en común con J que I, lo cual es absurdo, pues I era el conjunto de ciudades que más elementos en común con J tenía.

El absurdo provino de suponer que J no es óptimo y que por lo tanto el conjunto de ciudades con la mayor cantidad de elementos en común con J no es J. Así, J debe ser óptimo, en el sentido de que debe salvar la mayor cantidad de ciudades con el presupuesto disponible.

1.3. Complejidad propuesta

Como se desprende del pseudocódigo del punto anterior, el algoritmo que proponemos tiene 2 ciclos que se repetirán n veces (siendo n la cantidad de ciudades). En ambos ciclos realizamos operaciones con complejidad $\mathcal{O}(1)$, ya que son operaciones aritméticas, de comparación, asignaciones, o inserciones en vectores (no hay que hacer resizes), por lo tanto el costo de cada ciclo es de $\mathcal{O}(n)$.

Por otro lado, hay 3 operaciones por fuera de los ciclos, cuya complejidad es distinta de $\mathcal{O}(1)$. Estas son, dos creaciones de vector (complejidad $\mathcal{O}(n)$) y un ordenamiento de vector. Para el ordenamiento del vector se estipula una complejidad de $\mathcal{O}(n * log(n))$, ya que existen varios algoritmos de ordenamiento que responden a este requerimiento (HeapSort, MergeSort).

Por lo tanto la complejidad total del algoritmo es $\mathcal{O}(n + n + n + n * log(n) + n)$, lo que es igual a $\mathcal{O}(n * log(n))$.

1.4. Implementación en C++

Nota: se utilizó C++11 cuya implmentación de la función std::sort tiene complejidad $\mathcal{O}(n * log(n))$, a diferencia de la versión C++98 que tiene una función std::sort con complejidad $\mathcal{O}(n^2)$.

```
#include <functional>
#include <algorithm>
#include <vector>
#include <iostream>
using namespace std;

struct datoCiudad {

    datoCiudad() : costoRescate(0), cantSoldados(0), indiceCiudad(0) {}

    datoCiudad(int n1, int n2, int n3) : costoRescate(n1), cantSoldados(n2),
        indiceCiudad(n3) {}

    bool operator<(const struct datoCiudad& other) const {
        //Como se ordena datoCiudad
        return costoRescate < other.costoRescate;
    }

    int costoRescate;
    int cantSoldados;
    int indiceCiudad;
};

// Implementacion. Contiene el cargado de input mas la resolucion del ejercicio.
int main() {
    // Cantidad de Ciudades
    int n;
    cin >> n;
    // Presupuesto
    int P;
    cin >> P;
    // Vector que tiene los datoCiudad de cada ciudad
    vector<datoCiudad> infoCiudad(n);
    // Zombies por Ciudad
    int z;
    // Soldados por Ciudad
    int s;
    // Costo de enviar Soldado por Ciudad
    int c;
    // Cantidad de Ciudades Salvadas
    int ciudadesSalvadas = 0;
    // Soldados enviados por Ciudad
    vector<int> soldadosPorCiudad(n, 0);
    // 1 Soldado = 10 zombies
    int minSoldados = 10;


    for(int i = 0; i < n; i++) {
        cin >> z;
        cin >> s;
```

```
        cin >> c;
        // cantActual = parte alta de z /10 - s
        // (soldados necesarios para salvar ciudad)
        int cantActual = ((z % minSoldados) ? z / minSoldados + 1 :
                z / minSoldados) - s;
        // costoActual = el costo para salvar la ciudad
        int costoActual = cantActual * c;
        // Sino tengo que mandar refuerzo pongo los valores en 0
        if (cantActual < 0) {
            costoActual = 0;
            cantActual = 0;
        }
        // Completo el vector con los nuevos datos de cada ciudad
        datoCiudad dato(costoActual, cantActual, i+1);
        infoCiudad[i] = dato;
    }

    // Ordeno el vector infoCiudad
    sort(infoCiudad.begin(), infoCiudad.end());

    // Veo cuantas ciudades puedo salvar con el presupuesto asignado
    for(int i = 1; i <= n; i++) {
        if (P < infoCiudad[i - 1].costoRescate) {
            break;
        }
        ciudadesSalvadas++;
        soldadosPorCiudad[infoCiudad[i - 1].indiceCiudad - 1] =
                infoCiudad[i - 1].cantSoldados;
        P = P - infoCiudad[i - 1].costoRescate;
    }

    // Genero el output, segun formato solicitado
    cout << ciudadesSalvadas << ' ';
    for (int i = 0; i < n; i++) {
        cout << soldadosPorCiudad[i] << ' ';
    }
    cout << endl;

    return 0;

}
```

1.5. Experimentación computacional

La función que utilizamos para llevar a cabo las mediciones fue `std::clock`[1]. La unidad temporal que utilizamos para este ejercicio fue ciclos de clock. La complejidad teórica calculada es de $\mathcal{O}(n * log(n))$

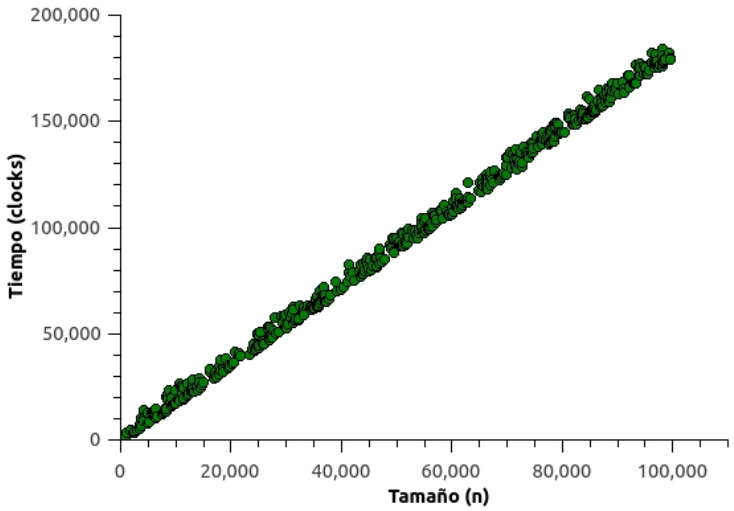
1.5.1. Experimentación con instancias aleatorias

Para generar las instancias aleatorias utilizamos la función `std::rand`[2] con determinados intervalos de valores para la variables, para obtener instancias coherentes. El detalle de intervalos es el siguiente:

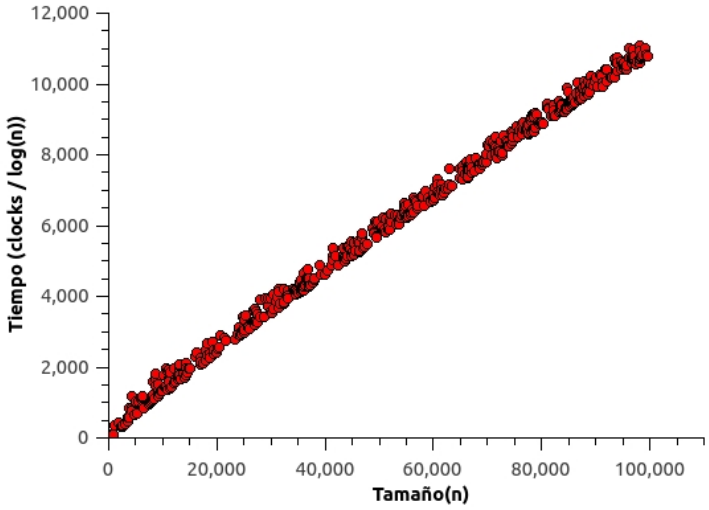
[1] Referencia `http://en.cppreference.com/w/cpp/chrono/c/clock`
[2] Referencia `http://en.cppreference.com/w/cpp/numeric/random/rand`

- Cantidad de ciudades (n): $100 \leq n \leq 100.000$
- Presupuesto disponible (P): $0 \leq P \leq 10.000$
- Cantidad de zombies por ciudad (z): $0 \leq z \leq 10.000$
- Cantidad de soldados por ciudad (s): $0 \leq s \leq 1.000$
- Costo de refuerzos por ciudad (c): $0 \leq c \leq 100$

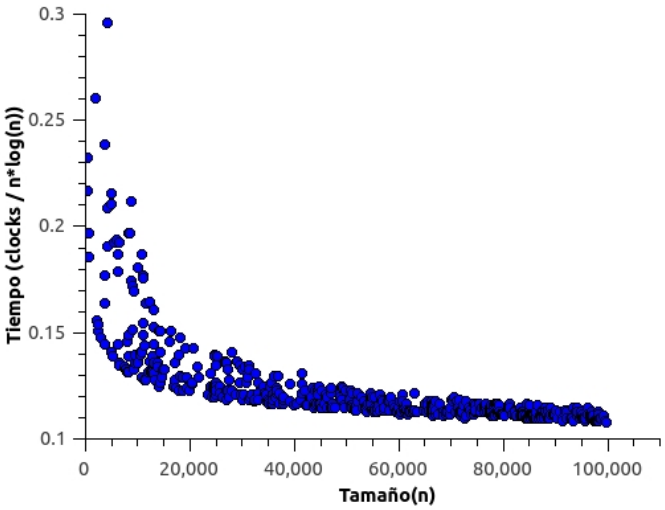
Generamos 500 instancias aleatorias, cuya medición temporal, arroja el siguiente resultado:



Si al tiempo medido lo dividimos por $log(n)$, se obtiene el siguiente resultado:



Por ultimo, dividimos el ultimo resultado obtenido por n, para de esta manera constatar que la complejidad experimental coincide con la calculada teóricamente:



A continuación, adjuntamos una tabla con los últimos 20 valores obtenidos en las instancias aleatorias, teniendo en cuenta que los casos fueron previamente ordenados según el tamaño (n):

Tamaño(n)	Tiempo(t)	$t/log(n)$	$t/n*log(n)$
96,553	178,384	10,772.61	0.112
96,631	176,000	10,627.89	0.110
96,703	181,731	10,973.25	0.113
96,714	176,870	10,679.63	0.110
96,750	178,210	10,760.19	0.111
96,998	177,114	10,691.63	0.110
97,107	179,106	10,810.82	0.111
97,288	175,469	10,589.58	0.109
97,538	176,710	10,662.09	0.109
97,904	176,177	10,626.46	0.109
98,066	184,153	11,105.95	0.113
98,068	177,755	10,720.08	0.109
98,100	181,413	10,940.38	0.112
98,229	178,379	10,756.18	0.110
98,296	178,935	10,789.07	0.110
98,404	180,022	10,853.57	0.110
98,612	179,711	10,832.83	0.110
99,097	182,861	11,018.01	0.111
99,342	179,873	10,835.65	0.109
99,482	179,194	10,793.42	0.108
Promedio			0.110

Promedio de las 500 instancias: 0.125

A partir de la información suministrada, podemos concluir que, en el último resultado, estamos en presencia de una constante cercana a 0, pero distinta a 0. Si bien esta conclusión no es suficiente como demostración matemática de que el limite no tiende a 0, si efectivamente tiende a 0, esto no afecta la conclusión a la que arribamos, ya que la complejidad sigue siendo $\mathcal{O}(n * log(n))$ De este análisis se desprende que la complejidad calculada teóricamente coincide con la complejidad de la experimentación.

1.5.2. Experimentación con instancias particulares

La diferencia entre mejor o peor caso viene dado por el algoritmo de ordenamiento, ya que es la operación con mayor complejidad, $\mathcal{O}(n * log(n))$. En la implementación presentada se utiliza la función `std::sort`[3], que acorde a la documentación disponible[4], la complejidad es siempre la misma. Por lo cual carece de sentido plantear mejor y peor caso, ya que el resultado sería el mismo.

Para demostrar este argumento, planteamos el siguiente experimento:

- Casos donde las ciudades ya están ordenadas desde el input, según el costo de salvar cada ciudad de forma creciente (posible "mejor" caso). Para generar este tipo de input, fijamos la cantidad de soldados y costo de enviar refuerzos de cada ciudad, mientras vamos incrementando de ciudad en ciudad los zombis en las mismas de forma que necesitamos cada vez mas soldados para rescatar la ciudad.

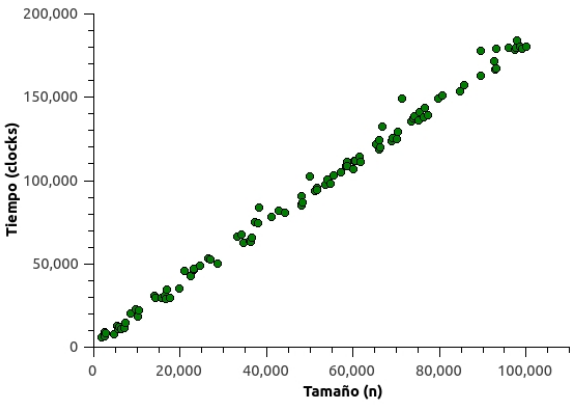
```
Ejemplo:
5 ciudades, $ XX presupuesto (es irrelevante)
ciudad 1: 1000 zombis 100 soldados $ 10 costo p/soldado
ciudad 2: 2000 zombis 100 soldados $ 10 costo p/soldado
ciudad 3: 3000 zombis 100 soldados $ 10 costo p/soldado
ciudad 4: 4000 zombis 100 soldados $ 10 costo p/soldado
ciudad 5: 5000 zombis 100 soldados $ 10 costo p/soldado
```

- Casos donde las ciudades ya están ordenadas desde el input, según el costo de salvar cada ciudad de forma decreciente (posible "peor" caso). Para generar este tipo de input, fijamos la cantidad de soldados y costo de enviar refuerzos de cada ciudad, mientras vamos decrementando de ciudad en ciudad los zombis en las mismas de forma que necesitamos cada vez menos soldados para rescatar la ciudad.

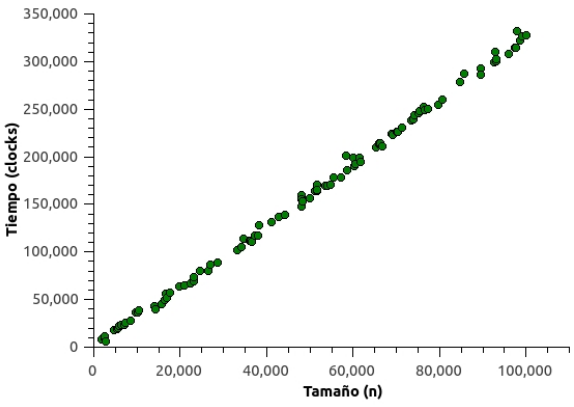
```
Ejemplo:
5 ciudades, $ XX presupuesto (es irrelevante)
ciudad 1: 5000 zombis 100 soldados $ 10 costo p/soldado
ciudad 2: 4000 zombis 100 soldados $ 10 costo p/soldado
ciudad 3: 3000 zombis 100 soldados $ 10 costo p/soldado
ciudad 4: 2000 zombis 100 soldados $ 10 costo p/soldado
ciudad 5: 1000 zombis 100 soldados $ 10 costo p/soldado
```

- 100 instancias de cada tipo.

A continuación exponemos gráficamente el resultado de la experimentación:



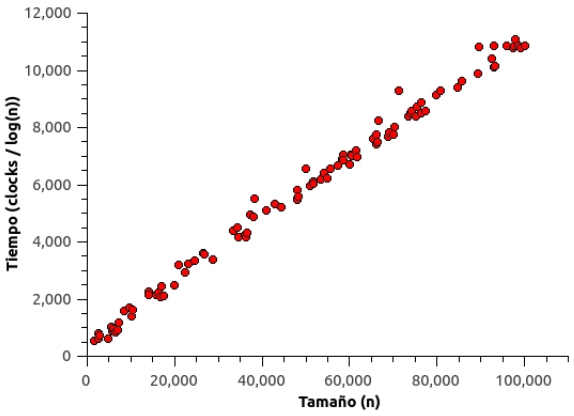
(a) Mejor Caso



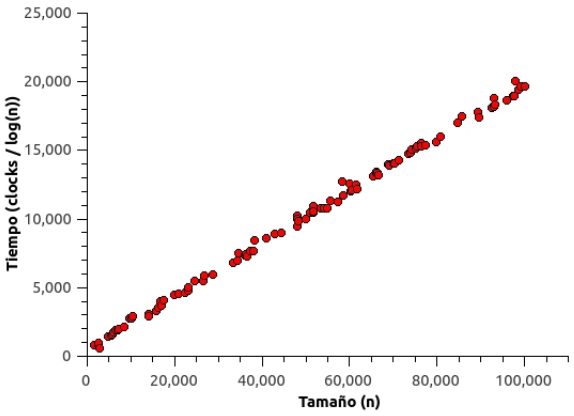
(b) Peor Caso

Al igual que en el caso de las instancias aleatorias, dividimos por $log(n)$ a los tiempos obtenidos:

[3]Referencia: `http://en.cppreference.com/w/cpp/algorithm/sort`
[4]Referencia: `https://gcc.gnu.org/onlinedocs/libstdc++/libstdc++-html-USERS-4.4/a01027.html`

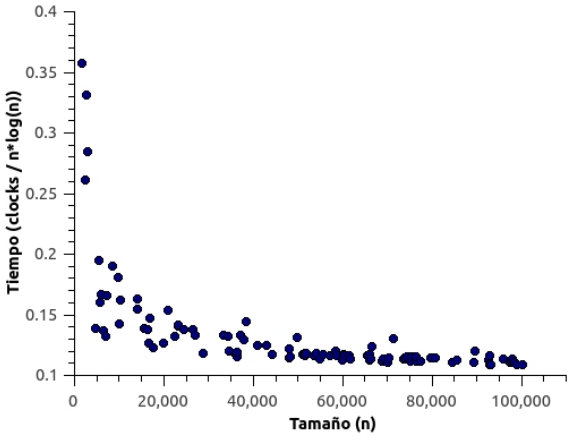


(a) Mejor Caso

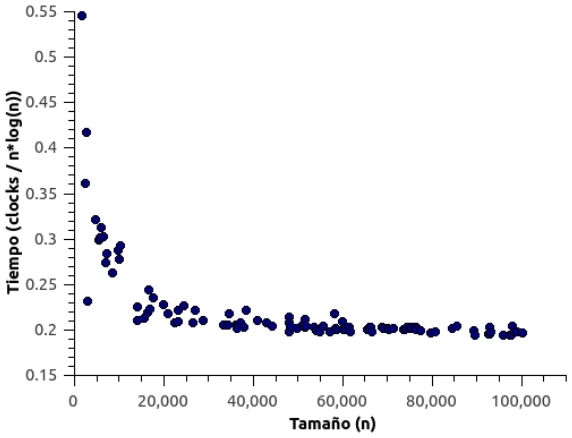


(b) Peor Caso

Por último dividimos el anterior resultado por n:



(a) Mejor Caso



(b) Peor Caso

Visualmente queda en evidencia la similitud entre ambos casos y el resultado obtenido con las instancias aleatorias. De esta forma, demostramos la inexistencia de un mejor o peor caso práctico.

A continuación, adjuntamos una tabla con los últimos 20 valores obtenidos en las instancias consideradas “mejor” caso, teniendo en cuenta que los casos fueron previamente ordenados según el tamaño (n):

Tamaño(n)	Tiempo(t)	$t/log(n)$	$t/n * log(n)$
75,298	141,520	8,735.63	0.116
76,191	138,505	8,540.55	0.112
76,329	144,175	8,888.75	0.116
77,184	139,809	8,611.04	0.112
79,597	149,414	9,177.52	0.115
80,640	151,454	9,292.11	0.115
84,521	154,040	9,411.61	0.111
85,486	157,776	9,630.24	0.113
89,272	163,195	9,923.13	0.111
89,483	178,221	10,834.55	0.121
92,479	171,924	10,421.63	0.113

Tabla 1/2

Tamaño(n)	Tiempo(t)	$t/log(n)$	$t/n * log(n)$
92,854	167,048	10,122.48	0.109
92,877	179,340	10,867.09	0.117
93,021	167,888	10,171.78	0.109
95,888	180,248	10,891.73	0.114
97,401	179,004	10,801.82	0.111
97,567	180,332	10,880.35	0.112
97,863	184,156	11,108.14	0.114
98,415	180,447	10,879.09	0.111
98,914	179,276	10,803.74	0.109
100,032	180,844	10,887.59	0.109
Promedio			0.113

Tabla 2/2

Promedio de las 100 instancias: 0.133

La información de los últimos 20 valores obtenidos en las instancias consideradas “peor” caso son:

Tamaño(n)	Tiempo(t)	$t/log(n)$	$t/n * log(n)$
75,298	248,304	15,327.10	0.204
76,191	252,415	15,564.52	0.204
76,329	249,327	15,371.63	0.201
77,184	250,608	15,435.32	0.200
79,597	254,918	15,657.94	0.197
80,640	261,103	16,019.37	0.199
84,521	279,177	17,057.29	0.202
85,486	287,561	17,551.97	0.205
89,272	293,204	17,828.37	0.200
89,483	286,589	17,422.54	0.195
92,479	299,692	18,166.63	0.196
92,854	310,673	18,825.61	0.203
92,877	300,995	18,238.77	0.196
93,021	303,404	18,382.25	0.198
95,888	309,010	18,672.36	0.195
97,401	314,691	18,989.72	0.195
97,567	315,383	19,028.66	0.195
97,863	332,936	20,082.43	0.205
98,415	322,493	19,443.00	0.198
98,914	327,398	19,730.04	0.199
100,032	327,654	19,726.19	0.197
Promedio			0.199

Promedio de las 100 instancias: 0.223

2. Ejercicio 2 - Alta frecuencia

2.1. Problema a resolver

Se tienen una lista de frecuencias con un tiempo de inicio y fin y un costo por transmisión. El problema a resolver consiste en dar un algoritmo que minimice los costos, es decir, que para cada tiempo encuentre la frecuencia con menor costo e indique como ir cambiando de frecuencia acorde pasa el tiempo.

- Ejemplo: Situación inicial.

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

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

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

- Solución del ejemplo:

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

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

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

2.2. Resolución planteada

La solución planteada utiliza la técnica de *Divide and Conquer*. La idea utilizada está basada en el algoritmo Merge Sort. Se agregan todas las señales recibidas a un vector y luego se llama a una función que se queda con dos problemas de la mitad de tamaño en cada paso, se llama a la misma recursivamente hasta llegar al caso base y luego se unen las soluciones en un único vector con la solución de la siguiente manera:

Creo un iterador para cada uno de los vectores (los llamamos vector izq y der):

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

Mientras alguno de los iteradores sea menor al tamaño del vector que itera me fijo cual de los dos elementos a los que apuntan los iteradores tiene menor principio y llamo a una funcion auxiliar que los ordenará.

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

Una vez que sale del ciclo, significa que al menos uno de los vectores fue recorrido completamente, por eso agrego los elementos restantes al vector solución:

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

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

Pasamos a mostrar como la función auxiliar mergeAux compara dos señales dadas. Llamaremos *primera* a la que tiene menor principio y *segunda* a la que tiene mayor principio. Se contemplan los siguientes casos:

- Si el costo de la *primera* es menor o igual al de la *segunda*:
  - Si el principio de la segunda es mayor al final de la primera, entonces agrego la primera al resultado.
  - Si el principio de la segunda es menor o igual al final de la primera y el final de la primera es menor al final de la segunda. Entonces, agrego la primera al resultado y le pongo como nuevo principio a la segunda el final de la primera.
  - Si el principio de la segunda es menor o igual al final de la primera y el final de la primera es mayor o igual al final de la segunda. Entonces desestimo la segunda señal.
- Si el costo de la primera es mayor al de la segunda: Guardo una variable int auxiliar (de ahora en adelante aux) con el final de la primera
  - Si el principio de la segunda es menor o igual al final de la primera y el final de la primera es mayor al principio, entonces le pongo como fin a la primera el principio de la segunda y la agrego al resultado. Si el fin de la segunda es menor que aux, modifico la primera poniendole como principio el final de la segunda y como final a aux y no aumento el iterador.
  - Si el principio de la segunda es menor o igual al final de la primera y el final de la primera es menor o igual al principio, entonces le pongo como fin a la primera el principio de la segunda y desestimo el resultado.
  - Si el principio de la segunda es mayor al final de la primera, entonces agrego la primera al resultado.

2.3. Complejidad propuesta

Analizamos la complejidad usando Teorema Maestro[5]:. En cada paso nos quedamos con dos subproblemas que tienen la mitad de tamaño que el original, así que a = 2 y c = 2. Además de la recursión, se unen los resultados. Para esto llamamos a la función merge, que realiza como máximo $2n - 1$ operaciones(*) con n el tamaño de la entrada. Esto tiene complejidad lineal en la cantidad de elementos del arreglo. Más allá de esto, se hace una cantidad constante de comparaciones y asignaciones, por lo que podemos decir que f(n) $\in \Theta$ (n). No podríamos usar el primer caso del teorema, porque $n^{\log_2 2-\epsilon} = n^{1-\epsilon}$, y no es cierto que f(n) $\in \Theta$ ($n^{1-\epsilon}$). En cambio, podemos usar el segundo caso, porque como ya dijimos f(n) $\in \Theta$ (n). Reemplazando, obtenemos T(n) $\in \Theta$ ($n^{\log_2 2}\ log(n)$), o sea T(n) $\in \Theta$ ($n\ log(n)$).

(*) Veamos por inducción que el peor caso es 2n-1:

Primero notemos que la función mergeAux solo hace una cantidad de comparaciones y asignaciones constantes, o sea, que tiene complejidad O(1).

- Caso base n=1

  Si n=1 la función merge no entrará en el primer ciclo ya que uno de los dos vectores a comparar tendrá tamaño 0 y tanto izq_it como der_it son 0. Por ende, solo agregará la única señal disponible al resultado realizando una cantidad constante de operaciones. Como 2n-1 = 2-1 = 1 se cumple.

- Paso Inductivo

  Supongamos que vale para n. Veamos que vale para n+1. Si tenemos n+1 señales, entonces la función entra en el ciclo de la función merge al menos una vez. Esto significa que entra en la función mergeAux al menos una vez. Viendo la función mergeAux podemos observar los posibles casos al comparar dos señales:

  Llamaremos *primera* a la que tiene menor principio y *segunda* a la que tiene mayor principio.

  - Que la primera tenga menor costo que la segunda y menor final que el principio de la segunda.
    Ejemplo:
    ```
    1 1 4
    2 5 6
    ```
    En este caso se llamará a mergeAux una vez, se agregará la primera señal al resultado y se aumentará el iterador i, resultando así un vector de tamaño n.
  - Que la primera tenga menor o igual costo que la segunda y que la segunda esté completamente contenida dentro de la primera.
    Ejemplo:
    ```
    1 1 4
    2 2 3
    ```
    En este caso se llamará a mergeAux una vez, se aumentará el iterador j, resultando asi un vector de tamaño n.
  - Que la primera tenga menor o igual costo que la segunda y menor final que la segunda.
    Ejemplo:
    ```
    1 1 4
    2 3 6
    ```
    En este caso se llamará a mergeAux una vez, se le pondrá como comienzo a la segunda el final de la primera. Luego, se agregará la primera al resultado y se aumentará el iterador, resultando así un vector de tamaño n.
  - Que la primera tenga mayor costo que la segunda y mayor final que el final de la segunda.
    Ejemplo:

[5]Teorema Maestro `http://es.wikipedia.org/wiki/Teorema_maestro`

```
2 1 4
1 2 3
```

En este caso se llamará a mergeAux una vez, se guardará el fin de la primera en una variable auxiliar. Luego, se pondrá como fin de la primera el comienzo de la segunda y se agregará la primera al resultado. Después, se le pondrá como principio a la primera el fin de la segunda y como fin el valor guardado en la variable auxiliar. Como no aumentaron los iteradores, se llamará de nuevo a la función. En el ejemplo anterior con los siguientes valores ahora:

```
2 3 4
1 2 3
```

Ahora la función es equivalente al primer ejemplo de la esta lista. Por lo que se agregará una al reultado y se aumentara un iterador resultando así una función de tamaño n. En este caso se llamó dos veces a la función mergeAux.

- Que la primera tenga mayor costo que la segunda y menor final que el principio de la segunda. Ejemplo:

```
2 1 4
1 5 6
```

En este caso se llamará a mergeAux una vez, se agregará la primera al resultado y se aumentará el iterador i, resultando así un vector de tamaño n.

- Que la primera tenga mayor costo que la segunda y mayor final que el principio de la segunda. Ejemplo:

```
2 1 4
1 3 6
```

En este caso se llamará a mergeAux una vez, se le pondrá a la primera como fin el principio de la mayor, se agregará esta al resultado y se aumentará el iterador i, resultando así un vector de tamaño n.

Como podemos ver, en todos los casos anteriores salvo en uno se llama una vez sola a la función mergeAux. En el peor caso esta se llama dos veces o sea, con costo 2*O(1). Veamos entonces que 2(n+1)-1 = 2n+2-1 = 2n+1. En el peor caso se llama a la función dos veces con costo O(1) y luego queda un vector de tamaño n, que por hipotesis inductiva tiene costo 2n-1.

Ahora 2n-1+2 = 2n+1 = 2(n+1)-1. Entonces queda demostrado.

2.4. Implementación en C++

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

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

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

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

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

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

    mergeSort(l);
    mostrar(l);

    return 0;
}

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

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

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

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

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

    vec = result; // O(n)
}

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

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

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

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

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

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

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

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

2.5. Experimentación computacional

La función que utilizamos para llevar a cabo las mediciones fue `std::clock`[6]. La unidad temporal que utilizamos para este ejercicio fue de nanosegundos. La complejidad teórica calculada es de $\mathcal{O}(n * log(n))$

2.5.1. Experimentación con instancias aleatorias

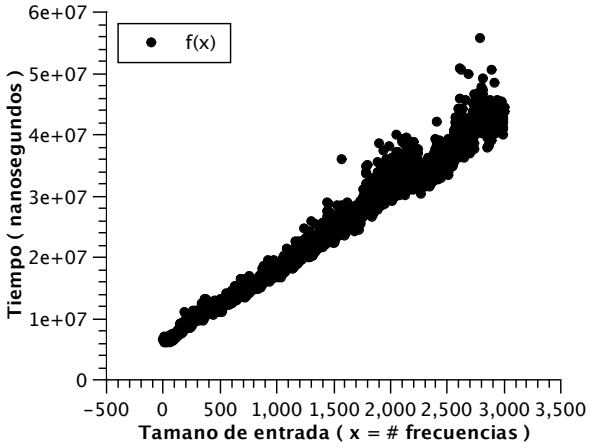
Para generar las instancias aleatorias utilizamos la función `std::rand`[7] con determinados intervalos de valores para la variables, para obtener instancias coherentes. El detalle de intervalos es el siguiente:

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

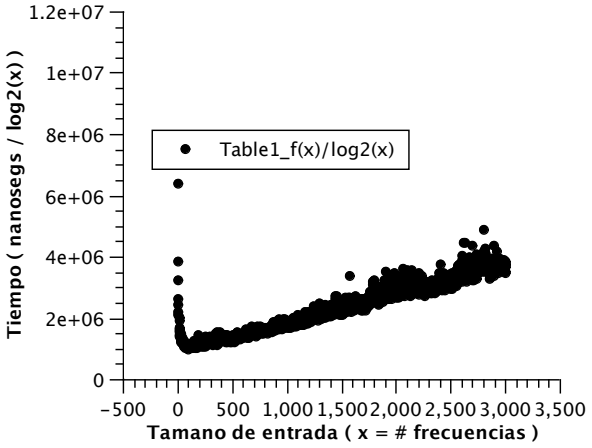
[6]Referencia `http://en.cppreference.com/w/cpp/chrono/c/clock`
[7]Referencia `http://en.cppreference.com/w/cpp/numeric/random/rand`

- Costos (P): $1 \leq P \leq 10.000$
- Para los inicios y finales se verifico que el final sea mayor al inicio en cada frecuencia

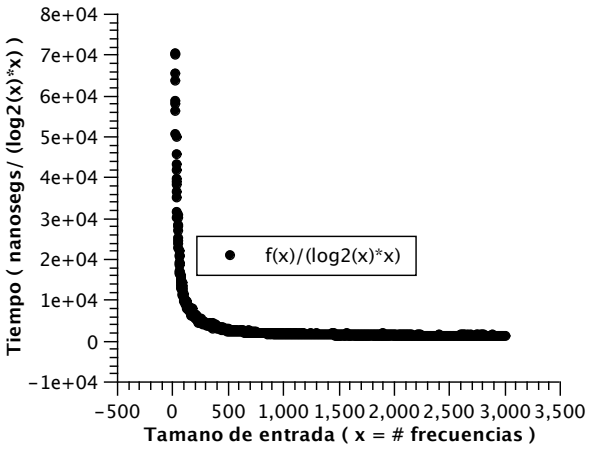
Se generaron 100 muestras. Para cada muestra se generaron instancias aleatorias de tamaños 1 a 3000. Luego se tomo el promedio de tiempos para cada tamaño en las 100 muestras



(a) Tiempos sin procesar



(b) Logaritmo a la figura (a)



(c) Dividiendo por x la figura (b)

Figura 4

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

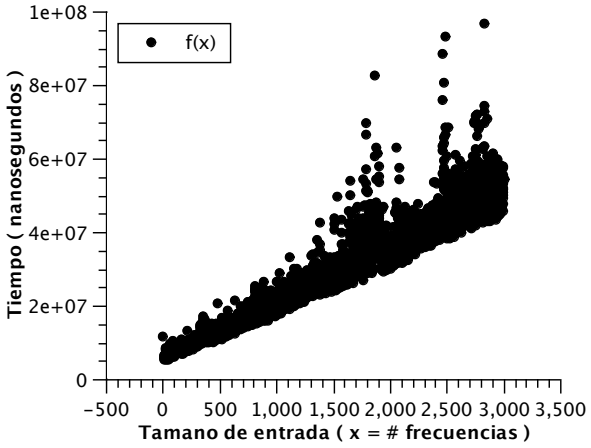
Tamano(n)	Tiempo(t)	$t/log(n)$	$t/n*log(n)$
2,980	41,823,717	3,623,894.53913547	1,216.07199299848
2,981	42,205,634	3,656,833.08407709	1,226.713547157695
2,982	41,379,610	3,585,113.377975757	1,202.251300461354
2,983	42,125,330	3,649,569.318887776	1,223.456023763921
2,984	45,341,672	3,928,055.705472136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

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

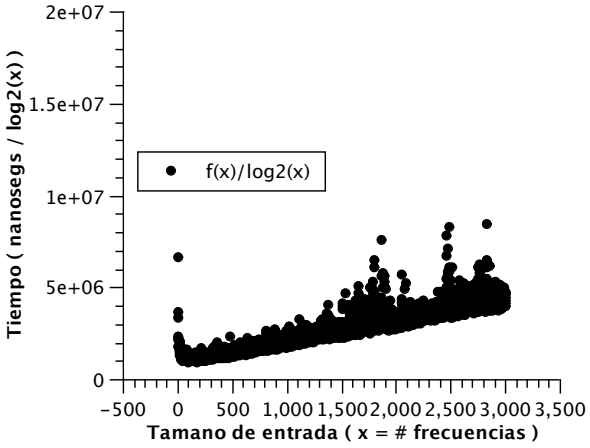
2.5.2. Experimentación con instancias particulares

Como quedo demostrado en la justificación de la complejidad teórica, el peor caso ocurre cuando al comparar dos frecuencias la de menor costo esta completamente contenida dentro de la de mayor costo, en este caso, la salida termina teniendo 2n-1 señales. Generamos una muestra de peor caso de tamaños 1 a 3000 de la siguiente manera:

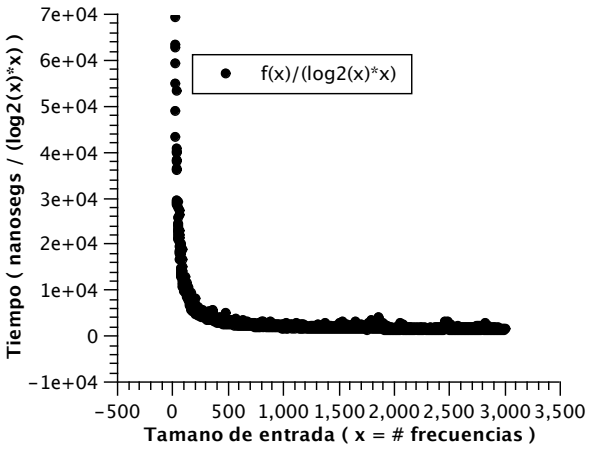
- El costo de la primer frecuencia es 3001
- El comienzo de la primer frecuencia es 1
- El final de la primer frecuencia es 6001
- Cada frecuencia que se agrega disminuye en 1 el costo, aumenta en 1 el comienzo y disminuye en 1 el final



(a) Tiempos sin procesar



(b) Logaritmo a la figura (a)



(c) Dividiendo por x la figura (b)

Figura 5

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

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

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

3. Ejercicio 3 - El señõr de los caballos

3.1. Problema a resolver

El problema a resolver consiste en dar un algoritmo de Backtracking que minimice la cantidad de caballos utilizados para cubrir todos los casilleros de un tablero de n x n.

Decimos que un casillero está cubierto sí está ocupado con un caballo o es amenazado por otro caballo.

A su vez, el tablero inicial puede o no tener caballos ya posicionados, en cuyo caso igual se quiere buscar la solución optima que minimice la cantidad de caballos utilizados.

- Ejemplo 1: Situación inicial.

```
n = 8, sin caballos iniciales
```

- Solución del ejemplo[8]:

```
Cantidad de caballos agregados: 12
Tablero:
0 0 0 0 0 0 0 0
0 0 1 0 0 0 0 0
0 0 1 1 0 1 1 0
0 0 0 0 0 1 0 0
0 0 1 0 0 0 0 0
0 1 1 0 1 1 0 0
0 0 0 0 0 1 0 0
0 0 0 0 0 0 0 0
```

3.2. Resolución planteada

La solución planteada utiliza la técnica algorítmica de backtracking, como se pide en el enunciado. La idea es recorrer el tablero inicial evaluando cada posición vacía (es decir sin caballo. Puede ser vacía y estar amenazada) para determinar si es necesario poner un caballo o, en caso contrario, comparar las mejores soluciones al poner un caballo en la casilla o dejarla como estaba.

[8] Se puede encontrar las soluciones para $3 \leq n \leq 17$ con tableros vacios en `http://home.earthlink.net/~morgenstern/solution/knsols1.htm`

Nuestro modelo consiste en representar el tablero con una matriz de tamaño $n * n$. De esta manera conseguimos acceder a cada dato del tablero en tiempo constante. Dicho esto, nuestro algoritmo va 'procesando' cada casilla sin caballo secuencialmente de izquierda a derecha y de arriba hacia abajo. Es decir que empieza desde la primer casilla sin un caballo (x,y), toma las decisiones correspondientes a la casilla, se fija si la casilla (x,y+1) tiene un caballo, si no lo tiene la 'procesa', en caso contrario se fija la casilla (x, y+2), (x, y+3), ... , (x, n-1) , (x+1, 0) , ... , (x+1, n-1) , (n-1, 0) y asi hasta (n-1, n-1).

Para ayudar a tomar estas decisiones hemos pensado e implementado una serie de *podas* que evitan llegar a soluciones invalidas o (parcialmente) suboptimas. Consideramos que se va a llegar a una solución invalida cuando, al visitar una casilla sin caballo que no esta amenazada, no ponerle un caballo implica que no pueda ser amenazada en el futuro. Es decir que no hay habría manera de colocar un caballo en una casilla a ser visitada en el futuro de manera que aquella casilla quede amenazada. De esto se encargan las podas 'B' y 'C', las cuales explicamos en la próxima sección. Las demás podas se encargan de no considerar decisiones que necesariamente llevarían a soluciones subóptimas (tableros en los cuales todas las casillas tienen un caballo o están amenazadas, pero cuya cantidad de caballos es mayor a la de otra solución).

3.2.1. Poda A

Este caso se da cuando todas las posiciones que amenaza el casillero siendo evaluado tienen un caballo o están amenazadas y alguna al menos tiene un caballo, es decir, la casilla actual esta amenazada. Si pasa esto, entonces el casillero siendo evaluado tiene que permanecer vacío. Porque si pusiera un caballo, entonces me llevaría a una solución sub-optima, ya que estaría poniendo un caballo que no es realmente necesario.

- Notación: 'c' es el casillero siendo evaluado, 'a' son las posiciones amenazadas, '1' son las casillas que tienen un caballo.
- Ejemplo: Situación inicial.

```
Tablero (c es el casillero siendo evaluado):
c 0 a 0
0 0 a 0
0 1 0 0
0 0 0 1
```

- Decisión subóptima:

```
Tablero:
1 0 a 0
0 0 a 0
0 1 0 0
0 0 0 1
```

- Decisión óptima:

```
Tablero:
a 0 a 0
0 0 a 0
0 1 0 0
0 0 0 1
```

3.2.2. Poda B

Este caso se da cuando existe una posición que amenaza el casillero siendo evaluado que ya fue procesada, está vacía y solo la puede amenazar el casillero siendo evaluado (todas las otras posiciones que pueden amenazar a la amenazada ya fueron procesadas y ninguna tiene un caballo). Si pasa esto, entonces el casillero siendo evaluado tiene que tener un caballo sí o sí. Dejarlo vacío me llevaría a una tablero incorrecto, porque sería imposible amenazar el otro casillero en el futuro.

- Ejemplo: Situación inicial.

```
Tablero (c es el casillero siendo evaluado):
0 0 0 0
0 0 a 0
0 c 0 0
0 0 0 1
```

- Decisión incorrecta: en este caso, amenazar la casilla (0,0) se vuelve imposible

```
Tablero:
0 0 0 0
0 0 a 0
0 a c 0
0 0 0 1
```

- Decisión correcta:

```
Tablero:
a 0 a 0
0 0 a a
0 1 c 0
0 0 0 1
```

3.2.3. Poda C

Este caso se da cuando todas las posiciones que amenaza el casillero siendo evaluado ya fueron procesadas y ninguna tiene un caballo. Si pasa esto, entonces el casillero siendo evaluado tiene que tener un caballo sí o sí. Porque si se lo deja vacío, no hay forma de amenazarlo en el futuro.

- Ejemplo: Situación inicial.

```
Tablero (c es el casillero siendo evaluado):
1 1 1
a c a
a a a
```

- Decisión incorrecta:

```
Tablero:
1 1 1
a 0 a
a a a
```

- Decisión correcta:

```
Tablero:
1 1 1
a 1 a
a a a
```

3.2.4. Poda D

Este caso se da cuando existe una posición que el casillero siendo evaluado amenaza que tiene un caballo, y a la vez, todas sus posiciones amenazadas que no son la actual tienen caballo. Además, no se deben cumplir *B* o *C*. Si pasa esto, entonces el casillero siendo evaluado tiene que permanecer vacío, excepto que el caballo amenazado halla sido puesto a través del input. En caso contrario, si pusiera un caballo me llevaría a una solución sub-optima, porque el casillero amenazado, estaría siendo amenazado por todas las posiciones posibles y además tendría un caballo. Casos en los que una posición contiene un caballo y, a la vez, todas sus posiciones amenazadas contienen un caballo llevan a tableros subóptimos, porque poner un caballo allí no cambia el total de casillas amenazadas.

- Ejemplo: Situación inicial.

```
Tablero (c es el casillero siendo evaluado):
c a 0 0
a 0 1 0
1 a a a
0 1 a 1
```

- Solución parcial subóptima:

```
Tablero:
1 a 0 0
a 0 1 0
1 a a a
0 1 a 1
```

- Solución parcial óptima:

```
Tablero:
a a 0 0
a 0 1 0
1 a a a
0 1 a 1
```

3.2.5. Poda G

Esta poda consiste en calcular una cota como primer aproximación a la cantidad de caballos necesarios para la solución óptima usando un *algoritmo goloso*. Si en algún paso del algoritmo del ejercicio la cantidad de caballos supera esta cota, dejamos de calcular dicha solución, porque la solución que nos dio el algoritmo goloso es mejor (esto lo generaliza la poda Z).

La cota en cuestión se calcula antes de empezar el algoritmo de backtracking.

El algoritmo goloso que se utiliza para esta cota buscará en cada paso maximizar la cantidad de posiciones amenazadas al poner un caballo. Es decir, compara la cantidad de nuevas casillas amenazadas entre varios casilleros y se queda con la mejor, y asi recorre cada casilla del tablero una sola vez.

Como claramente este problema no adhiere al *principio de optimalidad*, la solución dada no será óptima en principio, pero en general es una aproximación más interesante que usar como cota n^2.

3.2.6. Poda S

Esta poda funciona de manera muy parecida a lo que hace la *Poda G*. Sin embargo, en este caso la cota a utilizar es la cantidad de caballos en la mejor solución que se conoce para un Tablero de ese tamaño (o sea, para un n dado). Esta información fue recompilada del sitio mencionado en la sección de *Problema a resolver* y tiene dichos valores para $3 \leq n \leq 17$.

De esta forma, sí en algún momento encontramos una solución con cantidad de caballos igual a la cantidad de caballos utilizada en la solución optima, sabemos que no tenemos que seguir buscando y ya podemos dar la solución.

Es necesario notar, que los valores recompilados solo sirven para soluciones en las que el Tablero inicial esté vacío. Es por eso que esta poda además tiene en cuenta la cantidad de caballos iniciales. Sí en algún momento la solución parcial actual tiene más caballos que la solución optima más los caballos iniciales, entonces sabemos que podemos podar dicha solución, ya que no será optima.

3.2.7. Poda Z

Esta poda funciona de manera muy parecida a lo que hace la *Poda G*. Sin embargo, en este caso la cota a utilizar es la cantidad de caballos en la mejor solución encontrada hasta el momento. De esta forma, una vez que nuestro algoritmo encuentre una solución de k caballos, automaticamente empezara a podar todos los subarboles que tengan soluciones parciales con $k + 1$ caballos.

3.2.8. Pseudocódigo

Para hacer el pseudocódigo más sencillo, no vamos a listar las condiciones que tienen que pasar para cada Poda, ya que cada una fue explicada más arriba con ejemplos.

```
main:
        pongo tablero_inicial = un tablero vacio más los caballos pasados por input
        pongo cota_optima = resolver_con_goloso(tablero_inicial)
        result = resolver_aux(cota_optima, (0,0))

resolver_aux(optimo_alcanzado, ultima_celda):
        Si entro en los casos de las podas G, S o Z
                devuelvo la cantidad de caballos utilizados por la solución optima \
                        alcanzada
        pongo siguiente = siguiente celda vacía sin un caballo en el tablero \
                después de ultima_celda
        Si no hay siguiente
                devuelvo el mínimo entre la cantidad de caballos usada por la \
                        solución actual y la solución optima alcanzada
    Si entro en el caso de la poda A
                dejo la celda como esta y llamo a resolver_aux recursivamente
        Si entro en el caso de la poda B
                pongo un caballo en la celda y llamo a resolver_aux recursivamente
        Si entro en el caso de la poda C
                dejo la celda como esta y llamo a resolver_aux recursivamente
        Si entro en el caso de la poda D (y no entre en los casos B ni C)
                pongo un caballo en la celda y llamo a resolver_aux recursivamente
        Si no entre a ningún caso de poda
                pruebo llamando a resolver_aux recursivamente: dejando vacío y \
                        poniendo un caballo en la celda
                devuelvo el mínimo de esos dos

resolver_con_goloso(tablero):
        pongo siguiente = siguiente celda vacia no amenazada en el tablero
    Si no hay siguiente
                devuelvo la cantidad de caballos usada
        pongo posibilidades = [las posibles formas de poner un caballo que haga
                que la celda "siguiente" este cubierta]
        tomo de posibilidades la que maximize la cantidad de celdas cubiertas
        agrego al tablero la decision tomada
        llamo resolver_con_goloso(tablero)
```

3.2.9. Justificación

Como planteamos en nuestra resolución, nuestro algoritmo primero encuentra una solución aproximada usando un algoritmo auxiliar goloso. Luego, procede visitando cada celda sin un caballo secuencialmente. Las podas desechan tableros incorrectos y decisiones que necesariamente llevarían a tableros con más caballos de lo necesario. Finalmente, si ninguna poda aplica, compara las soluciones dadas entre dejar un casillero vacío y poner un caballo en ese lugar.

3.3. Complejidad propuesta

Dado un Tablero de n x n, el algoritmo propuesto evalúa cada celda de forma secuencial una sola vez, y ya que para cada celda hay dos posibilidades, o bien poner un caballo o bien dejarla vacía, el árbol de

decisión tendría 2^M nodos, donde *M* es la cantidad de celdas a evaluar.

Por ejemplo, en un tablero vacío de largo *n*, $M = n^2$. Sin embargo, si el tablero inicial tiene k celdas con caballos ya cargados, entonces tenemos $M = n^2 - k$, pues hay k celdas para las cuales no hay que decidir si tiene o no un caballo.

Ahora, para ver que el algoritmo propuesto tiene complejidad $\mathcal{O}(2^{n^2-k})$, tenemos que ver que las operaciones realizadas en cada nodo (sin contar las llamadas recursivas y guardar un nuevo tablero, explicado a continuación) tienen costo constante. En nuestro caso, realizamos distintas comparaciones de enteros y checkeos para las podas. Estos checkeos, a su vez, consisten en recorrer las posiciones a las cuales amenazaría un caballo colocado en la posición actual. Como la cantidad de posiciones que un caballo puede amenazar es a lo sumo 8, podemos decir que este ciclo se repite una cantidad constante de veces. Dentro del ciclo, solo se realizan operaciones constantes. Por lo tanto, en cada paso, se realizan a lo sumo dos llamadas recursivas y operaciones de costo constante.

Cuando llegamos a soluciones del tablero con menos cantidad de caballos que la mejor solución calculada hasta el momento, queremos guardar el tablero para mostrar las casillas con caballo al final del algoritmo. Si bien esta operación tiene costo n^2, esto no sucede muy frecuentemente. Veamos lo siguiente: el tablero se guarda solo si se encuentra uno nuevo con menos cantidad de caballos que el mejor hasta ahora, pero las posibles cantidades de caballos en el tablero son de 0 a n^2. Por lo tanto, como tablero se guarda como máximo n^2 cantidad de veces y esta operación tiene un costo de n^2, guardar el tablero solo añade n^4 trabajo a todo el algoritmo (contando todo el árbol de recursión). Entonces, estos casos no le añaden complejidad al algoritmo ya que, en particular, n^4 esta en $\mathcal{O}(2^{n^2-k})$.

Además, el algoritmo propuesto utiliza una función auxiliar que con un algoritmo goloso obtiene una primera cota para la solución optima. Entonces, tenemos que ver que la complejidad de esta función es a lo sumo $\mathcal{O}(2^{n^2-k})$. Siguiendo la notación de antes, la función auxiliar tiene *M* celdas a evaluar. Para cada una de ellas checkea si obtiene más celdas cubiertas amenazándola desde alguna celda vecina, o poniendo un caballo en ella, pero solo toma una decisión por cada celda y las evalúa secuencialmente. Como vimos recién, las comparaciones toman tiempo constante ya que cada celda tiene a lo sumo 8 celdas posibles que la pueden amenazar. Por lo tanto, la complejidad de esta función es $\mathcal{O}(M)$ que es estrictamente menor que $\mathcal{O}(2^M)$.

Luego, la complejidad del algoritmo propuesto en el peor caso es:

$$\mathcal{O}(2^{n^2-k})$$

3.4. Implementación en C++

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

// Varios typedefs
typedef vector<int> Vec;
typedef vector<Vec> Tablero;
typedef pair<int, int> Coord;
struct CoordEspecial {
    CoordEspecial(int n, Coord d) : nuevas_amenazadas(n), c(d) {}

    bool operator<(const struct CoordEspecial& other) const {
        //Como ordena el MaxHeap
        return nuevas_amenazadas < other.nuevas_amenazadas;
    }

    int nuevas_amenazadas;
    Coord c;
};

// Prototipado de funciones
int resolver(Tablero& p, int cant_caballos);
int aux_resolver(Tablero& p, Coord anterior, int cant_caballos, Tablero& t_optimo,
        int c_optimo, Tablero& original, int cant_caballos_original);
bool agregar_caballo(Tablero& p, int f, int c);
int cota_goloso(Tablero& p, int caballos);
vector<Coord> coordenadas_amenazadas(Tablero& p, int f, int c);
vector<Coord> nuevas_coordenadas_amenazadas(Tablero& p, int f, int c);
vector<Coord> aux_coordenadas_amenazadas(Tablero& p, int f, int c,
        bool dame_todas);
Coord proxima_coordenada(Tablero& p, Coord actual);
void mostrar(const Tablero& m);

// Implementacion
int main() {
    int n;
    int k;

    cin >> n;
    if(cin.eof()){
        return 1;
    }
    cin >> k;
    if(cin.eof()){
        return 1;
    }

    Tablero p(n, Vec(n, 0));
    int cant_caballos = 0;
    for(int i = 0; i < k; i++){
```

```
            int f, c;
            cin >> f >> c;
            f--;
            c--;
            if (agregar_caballo(p, f, c) == true) {
                cant_caballos++;
            }
        }

        int sol = resolver(p,cant_caballos);
        //cout << endl << "El tablero optimo tiene " << sol << " caballos." << endl;

        return 0;
}

/**
 * Agrega un caballo al Tablero dado.
 *
 * Codigos de celdas:
 * -----------------
 * 0 : no tiene caballo ni esta amenazada
 * 1 : tiene un caballo
 * 2 : no tiene un caballo y esta amenazada
 */
bool agregar_caballo(Tablero& p, int f, int c) {
        int n = p.size();
        //Check si (f,c) invalida o ya tiene un caballo
        if (f < 0 || f >= n || c < 0 || c >= n || p[f][c] == 1)
            return false;
        //Agrego un caballo a (f,c)
        p[f][c] = 1;
        //Marco las posiciones amenazadas
        vector<Coord> cam = nuevas_coordenadas_amenazadas(p, f, c);
        for (int i = 0; i < cam.size(); i++) {
            Coord crd = cam[i];
            if (p[crd.first][crd.second] != 1) // si no tiene un caballo
                p[crd.first][crd.second] = 2; // la marco como amenazada
        }
        return true;
}

/**
 * Devuelve todas las coordenadas que pueden ser amenazadas desde un casillero.
 */
vector<Coord> coordenadas_amenazadas(Tablero& p, int f, int c) {
        return aux_coordenadas_amenazadas(p,f,c,true);
}

/**
 * Devuelve todas las coordenadas que pueden ser amenazadas desde un casillero,
 * y no estan ya amenazadas o tienen un caballo.
 */
vector<Coord> nuevas_coordenadas_amenazadas(Tablero& p, int f, int c) {
        return aux_coordenadas_amenazadas(p,f,c,false);
```

```
}

/**
 * Funcion auxiliar que calcula coordenadas amenazadas.
 */
vector<Coord> aux_coordenadas_amenazadas(Tablero& p, int f, int c, bool dame_todas)
    int n = p.size();
    vector<Coord> solution;
    solution.reserve(8);
    Coord offsets[] = { make_pair(-1, 2),make_pair(2, -1),
        make_pair(-1, -2),make_pair(-2, -1),
        make_pair(2, 1),make_pair(1, 2),
        make_pair(-2, 1),make_pair(1, -2)};
    for(int i = 0; i < 8; i++) {
        int x = offsets[i].first;
        int y = offsets[i].second;
        if (0<=f+x && f+x<n && 0<=c+y && c+y<n && (p[f+x][c+y] == 0 || dame_todas))
            solution.push_back(make_pair(f+x, c+y));
    }
    return solution;
}

/**
 * Dada una coordenada, devuelve la proxima que no vale 1 o una invalida
 */
Coord proxima_coordenada(Tablero& p, Coord actual) {
    int n = p.size();
    bool encontro_proxima = false;
    while (actual.first < n && !encontro_proxima){
        if (actual.second < n-1){
            actual.second++;
        } else {
            actual.second = 0;
            actual.first++;
        }
        if (actual.first < n && p[actual.first][actual.second] != 1)
            encontro_proxima = true;
    }
    return actual;
}

/**
 * Imprime un tablero por stdout.
 */
void mostrar(const Tablero& m) {
    cout << endl;
    int n = m.size();
    for(int i = 0; i < n; ++i) {
        cout << int(m[i][0] == 1);;
        for (int j = 1; j < n; j++) {
            cout << "␣" << int(m[i][j] == 1);
        }
        cout << endl;
    }
```

```
}

/**
 * Encuentra el tablero optimo que minimiza la cantidad de caballos utilizados.
 *
 * param Tablero& p: el tablero inicial
 * param int n: el tamanio del tablero (p.size() = n y p[i].size() == n,
 *      con 0<=i<n)
 * param int cant_caballos: la cantidad de caballos utilizados en el
 *      tablero inicial
 */
int resolver(Tablero& p, int cant_caballos) {
    // calculo una primera cota para el tablero optimo con un algoritmo goloso
    Tablero optimo = p;
    int cota = cota_goloso(optimo, cant_caballos);
    //cout << "Cota algoritmo goloso: " << cota << endl;
    // guardo el tablero original para saber cuales eran los caballos que
    // venian al principio
    Tablero original = p;
    // empiezo evaluando en el (0,0)
    Coord principio(0, 0);
    int n = p.size();
    int sol = aux_resolver(p, principio, cant_caballos, optimo, cota, original,
        cant_caballos);

    //Genero el output
    cout << sol - cant_caballos << endl;
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            if (optimo[i][j] == 1 && original[i][j] != 1) {
                cout << i+1 << "␣" << j+1 << endl;
            }
        }
    }

    return sol;
}

/**
 * Funcion auxiliar que evalua un Tablero y se vuelve a llamar de forma recursiva
 * luego de tomar una decisison con respecto al siguiente casillero a evaluar.
 * La decision puede ser: dejar el casillero vacio (para amenazarlo mas adelante)
 * o poner un caballo en el casillero.
 *
 * Casos de poda:
 * --------------
 * A : todas las posiciones amenazadas por el actual estan en 1 o 2 y
 *      alguna tiene un 1
 * => tengo que dejar la posicion actual en 2
 * B : existe alguna posicion amenazada que ya fue procesada, tiene un 0
 *      y solo la puede amenazar la posicion actual (todas las posiciones que
 *      pueden amenazar a la amenazada ya fueron procesadas y ninguna
 *      tiene un caballo)
 * obs: clave para la correctitud
```

```
 * => tengo que poner un 1 en la posicion actual
 * C : todas las posiciones amenazadas ya fueron procesadas y ninguna
 *      tiene un 1 (ej: centro 3x3)
 * obs: clave para la correctitud
 * => tengo que poner un 1 en la posicion actual
 * D : existe una posicion amenazada que tiene caballo y, a la vez,
 *      todas sus posiciones amenazadas que no son la actual tienen caballo.
 *      Ademas, no se deben cumplir B o C
 * => tengo que dejar la posicion actual en 2
 * G : el primer tablero optimo lo calculamos con un goloso
 * Z: la solucion que esta siendo calculada tiene >= caballos que la
 *      solucion optima actual
 * S: speedup para problemas con Tablero de n <= 10. La solucion que
 *      esta siendo calculada tiene mas caballos que la cantidad de caballos
 *      de la solucion optima + la cantidad de caballos en el tablero inicial,
 *      o tiene exactamente la cantidad de caballos de la solucion optima.
 *
 * param Tablero& p: el tablero a evaluar
 * param Coord actual: el ultimo casillero evaluado antes de la llamada recursiva.
 * param int cant_caballos: la cantidad de caballos utilizados para el
 *      tablero dado.
 * param int c_optimo: la cantidad de caballos de la mejor solucion encontrada
 *      hasta ahora.
 * param Tablero& original: el tablero inicial.
 */
int aux_resolver(Tablero& p, Coord actual, int cant_caballos, Tablero& t_optimo,
        int c_optimo, Tablero& original, int cant_caballos_original) {
    int n = p.size();
    //Poda caso S
    int precomputed_solutions[] = {0,1,4,4,4,5,8,10,12,14,16};
    if (n < 11 && (c_optimo == precomputed_solutions[n] ||
        cant_caballos > precomputed_solutions[n] + cant_caballos_original))
        return c_optimo;

    //Poda caso Z
    if (cant_caballos > c_optimo)
        return c_optimo;

    //Encuentro proxima coordenada a rellenar
    if (actual.first < n && p[actual.first][actual.second] == 1)
        actual = proxima_coordenada(p, actual);

    //Si no encontro un actual, ya no hay mas nada para completar
    if (actual.first == n) {
        if (cant_caballos < c_optimo) {
            t_optimo = p;
            return cant_caballos;
        } else {
            return c_optimo;
        }
    }

    //Sino, veamos...
    vector<Coord> cam = coordenadas_amenazadas(p, actual.first, actual.second);
```

```
    bool caso_A = true, caso_B = false, caso_C = true, caso_D = false;

    for (int i = 0; i < cam.size() && !caso_B; i++) {
        Coord crd = cam[i];
        //check caso A de poda
        if (p[crd.first][crd.second] == 0) {
            caso_A = false;
        }
        //check caso C de poda
        if (crd > actual) {
            caso_C = false;
        }
        //check caso B de poda
        if (crd < actual && p[crd.first][crd.second] == 0) {
                // crd ya fue procesada y tiene un 0
            vector<Coord> cam_crd = coordenadas_amenazadas(p, crd.first,
                crd.second);
            bool solo_la_puede_amenazar_actual = true;
            for (int j = 0; j < cam_crd.size()
                && solo_la_puede_amenazar_actual; j++) {
                Coord crd2 = cam_crd[j];
                if (crd2 > actual)
                    solo_la_puede_amenazar_actual = false;
            }
            if (solo_la_puede_amenazar_actual)
                caso_B = true;
        }
        //check caso D de poda
        if (p[crd.first][crd.second] == 1
                && original[crd.first][crd.second] != 1) {
            vector<Coord> cam_crd = coordenadas_amenazadas(p, crd.first,
                crd.second);
            bool todas_menos_actual_la_amenazan = true;
            for (int j = 0; j < cam_crd.size()
                && todas_menos_actual_la_amenazan; j++) {
                Coord crd2 = cam_crd[j];
                if (crd2 != actual && p[crd2.first][crd2.second] != 1)
                    todas_menos_actual_la_amenazan = false;
            }
            if (todas_menos_actual_la_amenazan)
                caso_D = true;
        }
    }
    //ademas, para los casos A y C de poda, se tiene que cumplir lo siguiente
    caso_A = caso_A && p[actual.first][actual.second] != 0;
    caso_C = caso_C && p[actual.first][actual.second] == 0;

    //Si se cumple alguno de los casos
    if (caso_B || caso_C) {
        // si o si hay que poner un caballo
        int valor_anterior = p[actual.first][actual.second];
        vector<Coord> ncam = nuevas_coordenadas_amenazadas(p, actual.first,
                actual.second);
        p[actual.first][actual.second] = 1;
```

```
        // seteo como amenazadas, las coordenadas que estan en 0 y puedo amenazar
        for (int i = 0; i < ncam.size(); i++) {
            p[ncam[i].first][ncam[i].second] = 2;
        }

        int sol = aux_resolver(p, proxima_coordenada(p,actual), cant_caballos+1,
                t_optimo, c_optimo, original, cant_caballos_original);

        //vuelvo todo a como estaba antes
        for (int i = 0; i < ncam.size(); i++) {
            p[ncam[i].first][ncam[i].second] = 0;
        }
        p[actual.first][actual.second] = valor_anterior;

        return sol;
    }
    if (caso_A || caso_D) {
        // hay que dejar todo como estaba, obs: el caso D requiere que B y C
        // no se cumplan
        return aux_resolver(p, proxima_coordenada(p,actual), cant_caballos,
                t_optimo, c_optimo, original, cant_caballos_original);
    }

    //La coordenada actual puede ser 0 o 2 (porque seguro que no tiene un caballo)
    // y tengo que comparar resoluciones
    vector<Coord> ncam = nuevas_coordenadas_amenazadas(p, actual.first,
        actual.second);
    int valor_anterior = p[actual.first][actual.second];

    //Veo las soluciones dejando la coordenada como estaba
    int sol1 = aux_resolver(p, proxima_coordenada(p,actual), cant_caballos,
        t_optimo, c_optimo, original, cant_caballos_original);

    //Veo las soluciones poniendo un caballo en la coordenada
    p[actual.first][actual.second] = 1;
    for (int i = 0; i < ncam.size(); i++) {
        p[ncam[i].first][ncam[i].second] = 2;
    }
    int sol2 = aux_resolver(p, proxima_coordenada(p,actual), cant_caballos+1,
        t_optimo, sol1, original, cant_caballos_original);
    //obs: el sol1 aca es clave, porque compara las soluciones de los dos casos

    //vuelvo todo a como estaba
    for (int i = 0; i < ncam.size(); i++) {
        p[ncam[i].first][ncam[i].second] = 0;
    }
    p[actual.first][actual.second] = valor_anterior;

    return sol2;
}

/**
 * Funcion auxiliar que evalua un Tablero y calcula una primer cota para la
```

```
 * cantidad de caballos necesarios para armar una solucion.
 * La cota calculada se obtiene utilizando un algoritmo goloso que maximiza a
 * cada paso la cantidad de posiciones amenazadas con el nuevo caballo.
 *
 * param Tablero& p: el tablero a evaluar
 * param int n: el tamanio del tablero (p.size() = n y p[i].size() == n,
 *      con 0<=i<n)
 * param int caballos: la cantidad de caballos utilizados para el tablero dado.
 */
int cota_goloso(Tablero& p, int caballos) {
    int n = p.size();
    // Encuentro proxima coordenada a rellenar
    Coord c = Coord(-1,-1);
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            if (p[i][j] == 0) {
                c = Coord(i,j);
                break;
            }
        }
        if (c.first != -1) {
            break;
        }
    }
    // Si no encontro un actual, ya no hay mas nada para completar
    if (c.first == -1) {
        // devuelvo la cantidad de caballos usados
        return caballos;
    }

    // vamos a ver si conviene poner un caballo en el casillero c,
    // o si conviene amenazarlo desde alguna otra posicion vacia
    vector<Coord> ncam = nuevas_coordenadas_amenazadas(p, c.first, c.second);

    // tengo que evaluar como maximo 9 posibilidades (el casillero actual + 8
    // que la pueden amenazar)
    // voy a usar un heap para ordenar estas posibilidades de mayor a menor
    priority_queue<CoordEspecial> heap;
    heap.push(CoordEspecial(ncam.size(), c));
    for (int i = 0; i < ncam.size(); i++) {
        vector<Coord> ncam_aux = nuevas_coordenadas_amenazadas(p, ncam[i].first,
                ncam[i].second);
        heap.push(CoordEspecial(ncam_aux.size(), ncam[i]));
    }

    // uso la posiblidad que maximiza la cantidad de casilleros vacios amenazados
    CoordEspecial ce = heap.top();
    agregar_caballo(p, ce.c.first, ce.c.second);
    return cota_goloso(p, caballos + 1);
}
```

3.5. Experimentación computacional

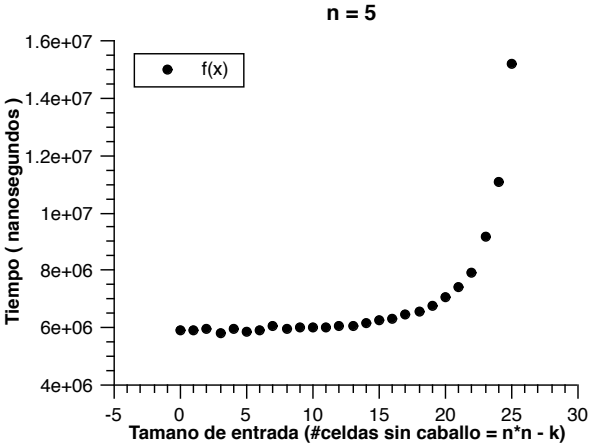
La función utilizada para llevar a cabo las mediciones fue `std::clock`.
La complejidad teórica calculada es de $\mathcal{O}(2^{n^2-k})$.

3.5.1. Experimentación con instancias aleatorias

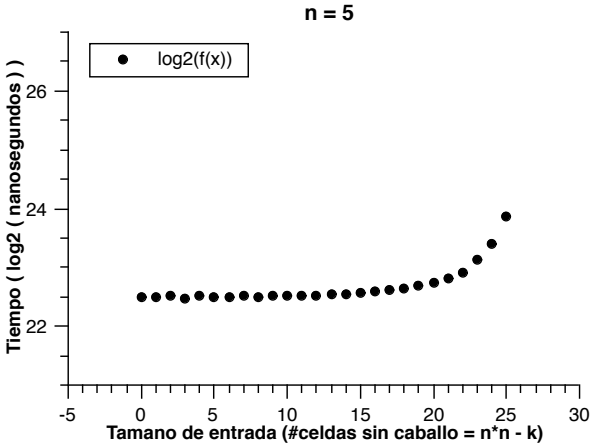
Para generar instancias aleatorias se creo un script que utiliza la función `std::rand`. Los intervalos para los parámetros fueron:

- Tamaño del tablero (n): $3 \leq n \leq 6$
- Cantidad de caballos iniciales (k): $0 \leq k \leq n^2$

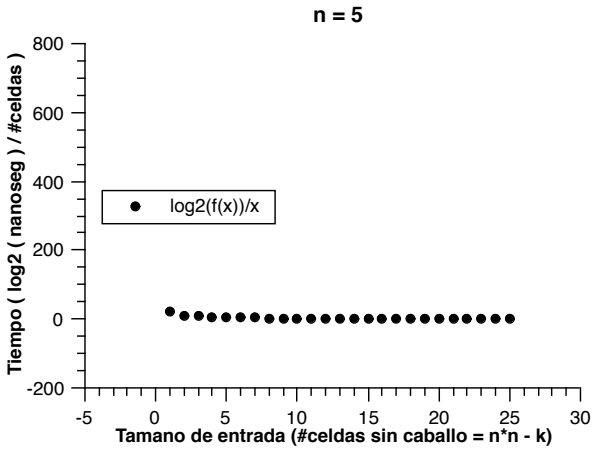
Se generaron 200 muestras para cada combinación de n y k posibles en dichos intervalos, y para cada $x = n^2 - k$ se promediaron dichas muestras. Dichas mediciones arrojaron los siguientes resultados:



(a) Tiempos sin procesar

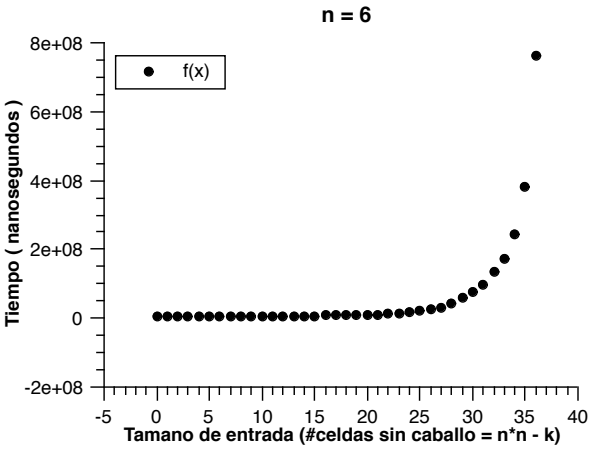


(b) Logaritmo a la figura (a)

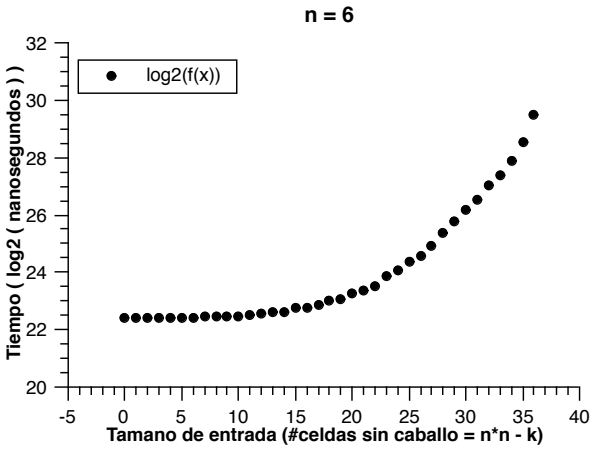


(c) Dividiendo por x la figura (b)

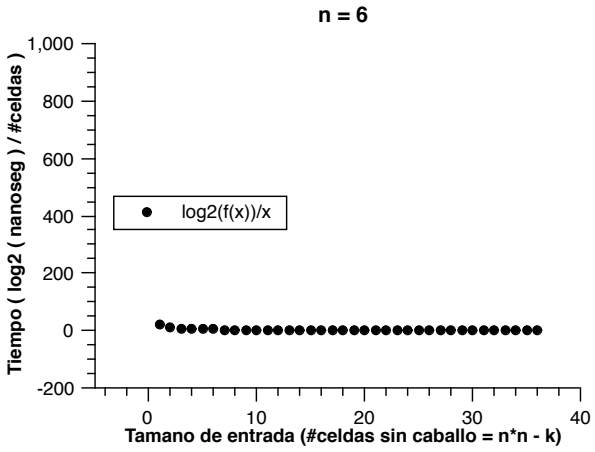
Figura 6: Gráficos para $n = 5$ sin podas G ni S



(a) Tiempos sin procesar

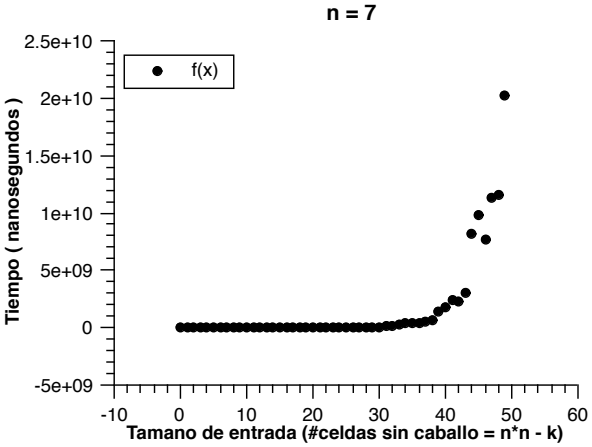


(b) Logaritmo a la figura (a)

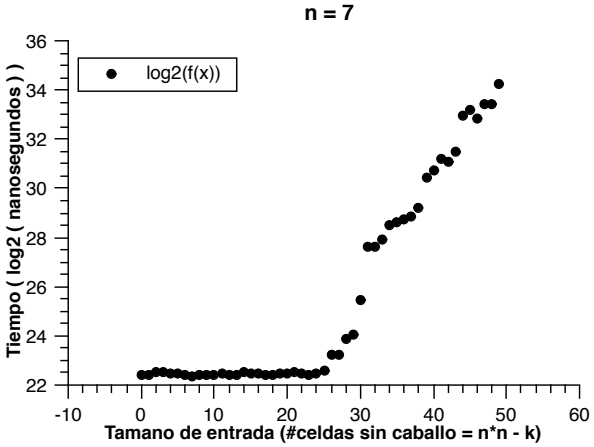


(c) Dividiendo por x la figura (b)

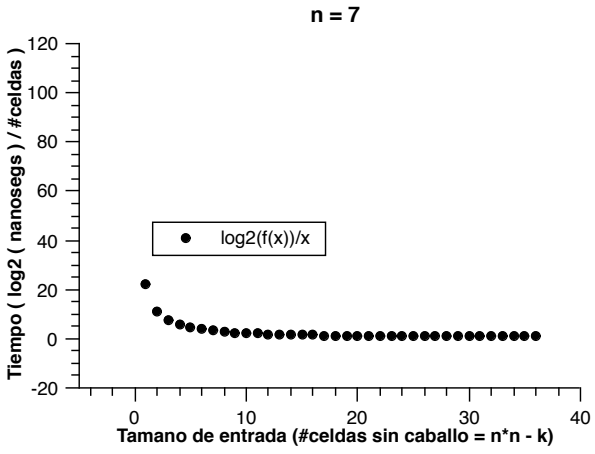
Figura 7: Gráficos para $n = 6$ sin podas G ni S



(a) Tiempos sin procesar



(b) Logaritmo a la figura (a)



(c) Dividiendo por x la figura (b)

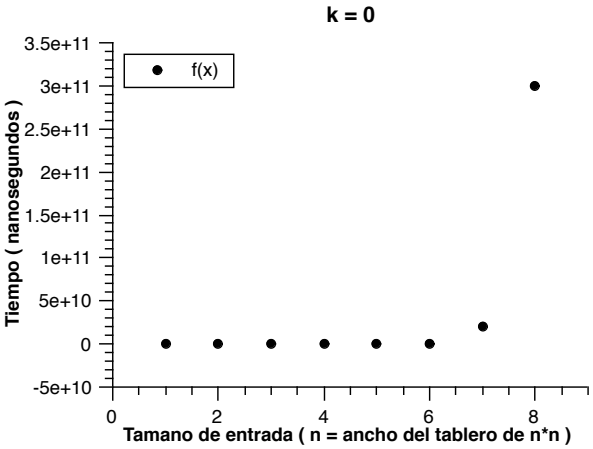
Figura 8: Gráficos para $n = 7$ sin podas G ni S

Con sus respectivas tablas en el Anexo A. *Nota: se quitaron las podas G y S para facilitar el análisis del gráfico.*

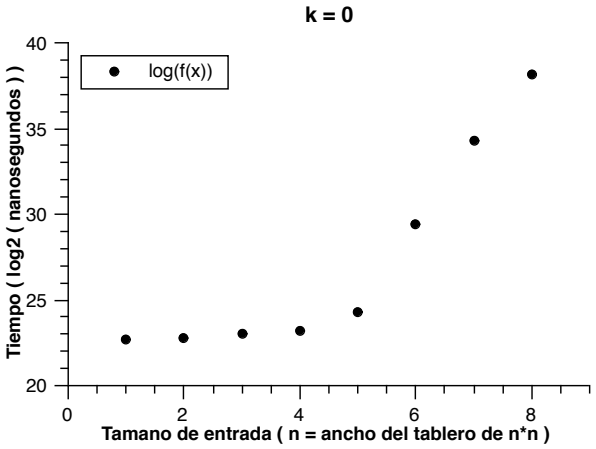
Como podemos ver de los gráficos 1, 2 y 3 y sus respectivas tablas, al aplicarles logaritmo y dividirlos por x, todos tienden a un número constante entre cero y uno (para más detalles, se puede ver la tabla de valores en el anexo). Entonces nuestro algoritmo tendría complejidad $\mathcal{O}(c * 2^{n^2-k})$, donde c es la constante a la cual converge el gráfico. Por lo tanto concluimos que los gráficos se condicen con nuestra predicción de complejidad.

3.5.2. Experimentación con instancias particulares

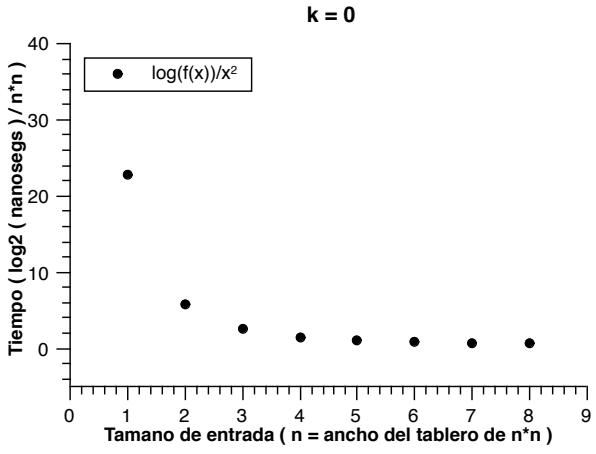
La experimentación con instancias particulares se realizó dejando el k fijo en cero y variando el n de 1 a 8:



(a) Tiempos sin procesar



(b) Logaritmo a la figura (a)



(c) Dividiendo por x la figura (b)

Figura 9: Gráficos para $k = 0$

Y su tabla también puede encontrarse en el Anexo A.

Para este experimento se utilizaron todas las podas detalladas anteriormente. Se deprenden entonces los mismos resultados observados en los gráficos y tablas anteriores (el razonamiento es identico).

3.5.3. Experimentación sin Podas

Por último, se realizaron experimentaciones para comparar las mejoras aportadas por las podas:

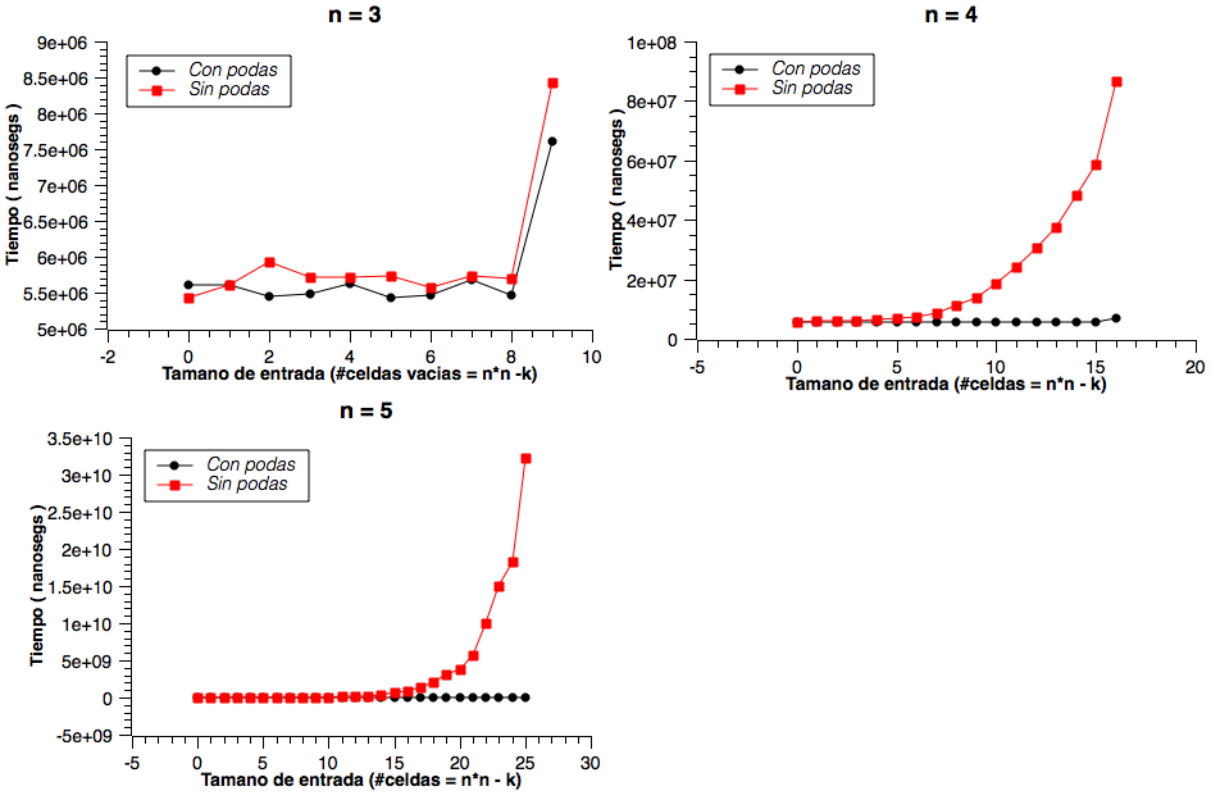


Figura 10: Gráficos para $n = 3,\ 4\ y\ 5$ comparando con podas vs. sin podas

Nota: el grafico sin podas sigue teniendo las podas B y C, ya que estás son las que hacen que el algoritmo devuelva soluciones correctas y no incorrectas.

Y su tabla también puede hallarse en el Anexo A.

A. Tablas Ejercicio 3

$x = n^2 - k$	f(x)	log2(f(x))	log2(f(x))/x
0	5,923,061	22.497912	inf
1	5,900,691	22.492453	22.492453
2	5,959,983	22.506877	11.253438
3	5,826,992	22.474320	7.491440
4	5,973,392	22.510119	5.627530
5	5,857,268	22.481796	4.496359
6	5,890,469	22.489951	3.748325
7	6,034,068	22.524699	3.217814
8	5,934,667	22.500736	2.812592
9	6,012,767	22.519598	2.502178
10	5,993,751	22.515028	2.251503
11	6,021,937	22.521796	2.047436
12	6,034,641	22.524836	1.877070
13	6,079,293	22.535472	1.733498
14	6,160,413	22.554596	1.611043
15	6,246,402	22.574594	1.504973
16	6,332,631	22.594374	1.412148
17	6,439,949	22.618618	1.330507
18	6,580,078	22.649673	1.258315
19	6,762,607	22.689148	1.194166
20	7,056,819	22.750587	1.137529
21	7,393,725	22.817870	1.086565
22	7,896,252	22.912737	1.041488
23	9,165,207	23.127736	1.005554
24	11,082,891	23.401831	0.975076
25	15,192,578	23.856863	0.954275

Tabla para $n = 5$

$x = n^2 - k$	f(x)	log2(f(x))	log2(f(x))/x
0	5,565,381	22.408049	inf
1	5,565,990	22.408207	22.408207
2	5,500,844	22.391221	11.195611
3	5,548,730	22.403726	7.467909
4	5,552,305	22.404655	5.601164
5	5,597,932	22.416462	4.483292
6	5,616,712	22.421294	3.736882
7	5,680,755	22.437651	3.205379
8	5,755,858	22.456600	2.807075
9	5,852,594	22.480645	2.497849
10	5,814,878	22.471317	2.247132
11	5,922,181	22.497697	2.045245
12	6,270,084	22.580053	1.881671
13	6,322,634	22.592094	1.737853
14	6,516,583	22.635684	1.616835
15	7,002,270	22.739391	1.515959
16	7,204,589	22.780485	1.423780
17	7,510,993	22.840572	1.343563
18	8,480,934	23.015792	1.278655
19	8,879,245	23.082006	1.214842
20	10,012,036	23.255232	1.162762
21	10,944,942	23.383761	1.113512
22	12,068,263	23.524715	1.069305
23	15,184,908	23.856135	1.037223
24	17,689,305	24.076374	1.003182
25	21,671,515	24.369297	0.974772
26	24,556,881	24.549624	0.944216
27	31,239,419	24.896864	0.922106
28	43,780,536	25.383786	0.906564
29	58,248,061	25.795707	0.889507
30	75,611,103	26.172095	0.872403
31	96,289,604	26.520877	0.855512
32	136,355,286	27.022795	0.844462
33	173,505,088	27.370403	0.829406
34	244,281,566	27.863970	0.819529
35	382,792,021	28.511986	0.814628
36	762,531,623	29.506222	0.819617

Tabla para $n = 6$

$x = n^2 - k$	f(x)	log2(f(x))	log2(f(x))/x
0	5,658,051	22.431874	inf
1	5,681,202	22.437765	22.437765
2	6,076,304	22.534763	11.267381
3	6,069,770	22.533210	7.511070
4	5,773,935	22.461123	5.615281
5	5,758,758	22.457326	4.491465
6	5,492,207	22.388955	3.731492
7	5,395,753	22.363393	3.194770
8	5,551,311	22.404397	2.800550
9	5,627,470	22.424055	2.491562
10	5,567,587	22.408621	2.240862
11	5,766,603	22.459290	2.041754
12	5,584,103	22.412894	1.867741
13	5,524,277	22.397354	1.722873
14	6,066,478	22.532428	1.609459
15	5,736,076	22.451633	1.496776
16	5,748,738	22.454814	1.403426
17	5,602,114	22.417540	1.318679
18	5,636,143	22.426277	1.245904
19	5,857,809	22.481930	1.183259
20	5,757,360	22.456976	1.122849
21	6,092,406	22.538581	1.073266
22	5,693,843	22.440971	1.020044
23	5,645,738	22.428731	0.975162
24	5,689,501	22.439871	0.934995
25	6,275,472	22.581292	0.903252
26	9,863,721	23.233701	0.893604
27	9,787,981	23.222580	0.860096
28	15,326,389	23.869514	0.852483
29	17,433,639	24.055370	0.829496
30	46,732,384	25.477919	0.849264
31	208,022,561	27.632165	0.891360
32	207,625,305	27.629407	0.863419
33	248,319,145	27.887620	0.845079
34	376,092,555	28.486513	0.837839
35	411,197,407	28.615256	0.817579
36	448,556,370	28.740714	0.798353
37	492,876,908	28.876652	0.780450
38	609,898,842	29.183995	0.768000
39	1,427,423,303	30.410766	0.779763
40	1,758,884,497	30.712014	0.767800
41	2,463,547,271	31.198090	0.760929
42	2,262,938,362	31.075550	0.739894
43	3,067,540,200	31.514435	0.732894
44	8,222,938,242	32.937007	0.748568
45	9,802,502,055	33.190503	0.737567
46	7,629,766,484	32.828992	0.713674
47	11,287,976,621	33.394068	0.710512
48	11,509,510,828	33.422107	0.696294
49	20,280,864,861	34.239400	0.698763

Tabla para $n = 7$

$x = n$	f(x)	log2(f(x))	log2(f(x))/(x*x)
1	6,894,821	22.717082	22.7171
2	7,258,736	22.791287	5.69782
3	8,437,732	23.008424	2.55649
4	9,649,562	23.202032	1.45013
5	20,032,573	24.255844	0.970234
6	696,757,144	29.376081	0.816002
7	20,679,794,169	34.267503	0.699337
8	299,907,350,785	38.125726	0.595714

Tabla para $k = 0$

k	$n = 3$		$n = 4$		$n = 5$	
	con podas	sin podas	con podas	sin podas	con podas	sin podas
0	5,599,710	5,421,781	5,473,768	5,784,018	5,605,871	6,544,455
1	5,612,436	5,604,190	5,510,945	5,947,644	5,536,143	7,107,637
2	5,450,933	5,936,583	5,521,407	6,005,466	5,495,775	8,050,800
3	5,487,297	5,705,466	5,477,040	6,037,971	5,589,136	9,446,413
4	5,617,896	5,717,293	5,509,341	6,381,715	5,553,264	9,799,520
5	5,431,101	5,728,081	5,555,143	6,812,464	5,518,550	11,924,701
6	5,471,542	5,566,319	5,506,907	7,172,540	5,596,573	14,523,360
7	5,672,431	5,727,042	5,533,490	8,411,726	5,486,095	27,464,034
8	5,467,644	5,688,279	5,514,259	11,391,268	5,541,097	39,776,822
9	7,611,266	8,436,007	5,544,664	13,931,410	5,584,355	66,818,916
10			5,565,962	18,574,226	5,509,177	64,198,701
11			5,529,941	24,196,471	5,548,982	100,469,504
12			5,564,854	30,429,761	5,589,544	176,433,434
13			5,551,530	37,589,963	5,615,420	242,246,035
14			5,578,268	48,436,269	5,628,109	377,537,126
15			5,522,458	58,712,605	5,659,103	683,836,846
16			6,956,095	86,716,207	5,767,523	893,203,595
17					5,788,079	1,326,734,757
18					5,974,659	2,031,435,579
19					6,048,999	3,165,594,743
20					6,283,413	3,781,117,314
21					6,259,904	5,603,863,234
22					6,193,765	9,993,456,118
23					6,006,958	14,970,489,604
24					5,502,917	18,238,644,771
25					8,168,827	32,265,934,044

Tabla para $n = 3,\ 4\ y\ 5$ comparando con Podas vs. sin Podas