Algoritmos y Estructuras de Datos III

Trabajo Práctico N°1

Departamento de Computación
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

Integrante	LU	Correo electrónico
Amil, Diego Alejandro	68/09	amildie@gmail.com
Barabas, Ariel	755/11	ariel.baras@gmail.com
Garcia Marset, Matias	356/11	matiasgarciamarset@gmail.com
Orellana, Ignacio	229/11	nacho@foxdev.com.ar

Reservado para la cátedra

Instancia	Docente	Nota
Primera entrega		
Segunda entrega		

Índice

1. Introducción

El presente informe apunta a documentar el desarrollo del trabajo práctico número 1 de la materia Algoritmos y Estructuras de Datos III, cursada correspondiente al primer cuatrimestre del año 2013. El siguiente trabajo práctico propone la resolución de cuatro problemas algorítmicos. Vamos a dividir la resolución de cada uno de estos problemas en los siguientes aspectos:

1. Realizar una descripción del problema en sí; tratando de exponer la naturaleza del ejercicio sin ambiguedades, detallar las particularidades que consideremos pertinentes y proponer una solución.
2. Demostrar formalmente que la solución propuesta es la correcta.
3. Detallar mediante pseudocódigo la implementación realizada para la obtener la solución y justificar la complejidad del algoritmo.
4. Someter a la implementación a una serie de tests para verificar tanto su correctitud como su performance y graficar los resultados.

2. Pautas de Implementación

Adoptaremos a `C++` como lenguaje para realizar las implementaciones. De ser necesario vamos a usar la librería standard, haciendo referencia a la documentación de la misma cuando sea necesario. Cada ejercicio se encuentra en su respectiva carpeta dentro de la carpeta `codigo`, en archivos nomenclados desde `ej1.cpp` a `ej4.cpp`.

Dentro de cada respectiva carpeta se encuentran dos archivos adicionales, un .in, desde donde se van a leer los datos de entrada y un .out, en donde se van a escribir. Todos los códigos se compilan y ejecutan al correr el script `compilar.sh`, ubicado en la carpeta `codigo`. Para simplificar el código decidimos crear una clase que maneje la entrada y salida de datos.

Esta misma clase se encarga de leer la entrada, parsear la información, armar las estructuras necesarias y mostrar los resultados. Tanto en las justificaciones de complejidad como en el pseudocódigo vamos a obviar mencionar estos procedimientos, y consideraremos que los algoritmos comienzan con las estructuras necesarias y sus propiedades configuradas; y terminan al haber completado el procesamiento en si.

3. Resoluciones

3.1. Ejercicio 1

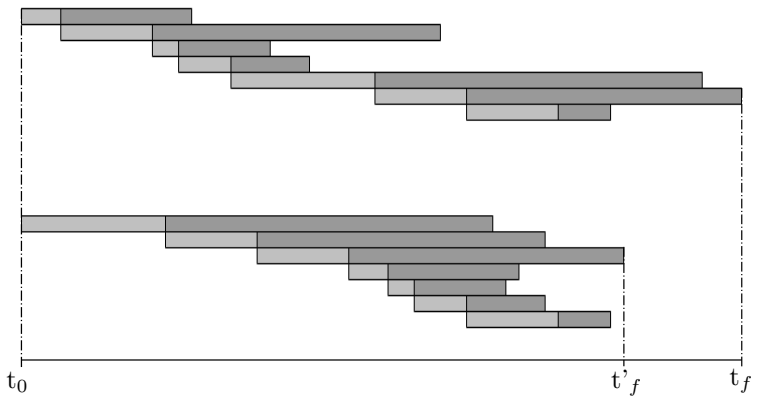
3.1.a. Interpretación del problema

Tenemos un número n de máquinas. Cada una de estas tiene un tiempo t que va a tomarse para realizar la produccion. Si definimos como t_i al instante en el que se comienza a cargar la primera maquina, sea cual sea, y como t_f el instante en el que la última máquina termina su producción, tenemos que buscar un órden de carga que garantice un t_f mínimo. Cabe destacar que este orden puede no ser único, por ejemplo, en el caso en el que todas las máquinas tengan el mismo tiempo de carga y el mismo tiempo de producción, cualquier orden para cargarlas es óptimo.

Ahora bien; intuitivamente pensamos que, en pos de buscar el menor tiempo de producción, es deseable que la mayor cantidad de máquinas trabajen al mismo tiempo para amortizar sus tiempos de producción individuales. Sería interesante evaluar la posibilidad de encenderlas en orden, comenzando por la máquina de mayor tiempo de producción y terminando por la de menor. Hay que tener en cuenta que, sea cual sea el orden en el que sean enciendan, siempre vamos a tener un mismo tiempo de encendido t_e que es la suma del tiempo de encendido de todas las máquinas. Asumimos como trivial que $t_e < t_f$.

A continuación presentamos una ilustración de la comparación del tiempo de producción total al encender las máquinas en un orden aleatorio versus el tiempo de producción total al encenderlas en orden decreciente por sus tiempos de producción.

```
input:
7
3 7 2 4 11 7 7
9 22 7 6 25 21 4
```



Se observa cómo el tiempo de producción total se redujo considerablemente. Vamos a proponer entonces como hipótesis que el orden óptimo para encenderlas es aquel que procede de esta manera y lo demostraremos a continuación.

3.1.b. Demostración de la solución

Supongamos una solucion S. En esta solucion todas las máquinas están ordenadas como proponemos: de mayor tiempo de producción a menor. Supongamos ahora una solución S' en la que las máquinas tienen el mismo orden que S, exepto por un único par de máquinas. Vamos a demostrar que el tiempo que se tarda S' en terminar no puede ser estrictamente menor que S.

Si definimos a M_{max} como la última máquina en terminar, al permutar un par de máquinas pueden pasar 2 cosas: que M_{max} esté en la permutación o que no lo esté.

1 - M_{max} no está en la permutación.

Si sucede esto, M_{max} sigue terminando en su momento original. Cuaquier cambio posible sólo va a poder empeorar el tiempo de S.

2 - M_{max} está en la permutación. Si esto pasa, la máquina va ser cargada y encendida antes o después del momento en el que es encendida en el orden de la solución S. Si es encendida después, va a terminar después, haciendo a la solución S' peor que S. Si es encendida antes, ahora una máquina que tiene un mayor tiempo de producción va a ser encendida en el momento en el que debía ser encendida M_{max}, alargando más el tiempo de producción y, otra vez, haciendo a la solución S' obsoleta.

Luego, no hay permutación posible de 2 elementos que pueda mejorar el tiempo de S, haciendo a esta la solución buscada.

3.1.c. Pseudocódigo

Para simplificar el programa, representaremos las máquinas con el siguiente struct:

```
struct{ int id; int c; int p; } maquina
```

Donde id representa al número de máquina, c su tiempo de carga y p a su tiempo de producción. Obtener cualquiera de los atributos cuesta O(1). También estableceremos la siguiente relación de orden entre dos máquinas cualquesquiera:

$m_i > m_j \iff tiempo\_de\_carga(m_i) > \text{tiempo\_de\_carga}(m_j)\ \forall i, j \in [0, ..., n-1]$

Ya hemos demostrado que el órden óptimo de carga para las máquinas es aquel que las carga comenzando por la que tiene mayor tiempo de ejecución y terminando por la que tiene menor. Aprovechamos la librería standard de C++ para hacer este sort, dejándonos la implementación de este problema únicamente como:

Algorithm 1: Ordenamiento de máquinas - O(n logn)

```
Data: vector<maquina> v
sort(v) // O(n log n)[a]
tiempo_total(v) //O(n)
```

[a] una pagina de cppreference o algo

Donde tiempo_total es una función que calcula el tiempo total de ejecución. Esta función recorre iterativamente el vector de máquinas y guarda en memoria una variable llamada inicio. Esta variable guarda, para una determinada máquina i, el momento en el que esta máquina comienza su ejecución. En cada iteración, calcula el tiempo que va a tardar la máquina i en terminar su producción y va guardando el máximo, que luego devuelve.

Algorithm 2: Función tiempo_total - O(n)

```
Data: vector<maquina> v
Result: int tiempo_total_prod
vector<int> tiempos
int inicio ← 0
int t_max ← tiempos[0]
for (int i = 1, i < longitud(v), i++) do
    inicio ← inicio + v[i − 1].c
    if ( v[i].c + v[i].p + inicio > t_max ) then
        t_max ← v[i].c + v[i].p + inicio
    end
end
return t_max
```

3.2. Ejercicio 2

3.2.a. Interpretación del problema

El problema pide identificar un sensor de acuerdo a las mediciones que estos hacen y a un numero k que determina la k-esima medición en la cual ese sensor fallo. esto se debe obtener a partir de conocer los intervalos en los cuales los sensores dan información y la cantidad de sensores que hay. El problema se resuelve simulando la sucesión de apariciones de las mediciones de acuerdo con los intervalos de aparición y buscando la k-ésima medición y asi obtener el numero de sensor al que pertenece.

La forma que encontramos para resolver el ejercicio fue utilizando una estructura de datos que nos permita insertar elementos de forma ordenada, en este contexto podemos insertar túplas que representen (intervalo, sensor) donde el intervalo es el tiempo en el cual se producirá la próxima medida y el sensor que identifica a cada sensor, entonces en cada paso (hasta llegar a los k pasos) tomamos el menor elemento y le sumamos su intervalo de medición (que esta dado por parámetro), esto va a reubicar el elemento mas adelante en la estructura y así vamos formando la lista de sucesivos elementos hasta llegar al k-ésimo elemento.

La estructura de datos que utilizamos es un Heap , en donde podemos guardar los elementos de forma ordenada en O(log (n)) y podemos tomar el menor elemento en tiempo constate, las operaciones push y pop del Heap tienen un costo temporal de O(log n) donde n es la cantidad de elementos del Heap.

siendo n los sensores y k la k-esima medida. nos toma O(n∗log n) agregar todos los elementos y O(k∗log n) reórdenar y tomar los primeros k elementos.
Caso particular: k <= n , en este caso particular resulta impráctico armar la estructura ya que según el enunciado en el tiempo 0, todos los sensores dan una primera medida , con lo cual tenemos en forma consecutiva de 1 a n las primeras n medidas y la k-ésima medida se encuentra dentro de esa lista, en este contexto alcanza con devolver el valor k que apunta al k-ésimo sensor.

3.2.b. Pseudocódigo

Algorithm 3: Función tiempo_total - O($n * log(n) + k * log(n)$)

```
Data: int sensores, int k, array mediciones[n]
Result: int nro_sensor
Heap h
int i=1
if k <= n then
    devuelvo k
else
    while i<n+1 do
        meter < m[i], i > en el heap
        i++
    end
    while i<k+1 do
        min = sacar elemento de h
        borro elemento de h
        min = < primero(min) + tercero(min), segundo(min), tercero(min) >
        meto min en h
        nro_sensor = segundo(min)
        i++
    end
    devuelvo nro_sensor
end
```

3.2.c. Demostraciones

Para mostrar que la solución es la correcta primero mostremos el invariante de la estructura elegida:

Invariante del Heap:
El elemento mínimo esta unívocamente determinado[1] y es el primero[2]
De esta manera creamos la estrctura donde vamos a guardar los sensores con sus intervalos de proxima medicion, la estructura va a guardar estos elementos ordenados por intervalo de proxima medicion y si hubiera dos mediciones iguales, se desempata por el numero de sensor, siempre tomando el mas chico primero.
La estructura anteriormente nombrada asegura que el primer elemento siempre contiene la próxima medición, esto es porque si hubiese otra medición (es decir, una con medición menor[3]), esta estaría antes, y seria la primera en el heap (por su invariante).
En cada iteracion, tomo el menor elemento que representa el elemento cuyo intervalo de proxima medicion es la mas chica, y le sumo su intervalo y le disminuyo el valor a k en uno. la invariante del heap me garantiza que en cada paso voy a tener el menor elemento ordenado por intervalo de proxima medicion y esto se traduce en el primer sensor que va a producir su medicion en una linea de tiempo. existe el caso particular en el cual la k-esima medicion es menor que la cantidad de sensores que hay, con lo cual unicamente encontramos la solucion buscando el k-esimo sensor ya que todos producen mediciones en el tiempo 0. y lo devuelvo, esto se hace directamente devolviendo el valor de k

[1] Lo que garantiza esto es que no hay túplas iguales en la estructura.

[2] Esto es gracias a las operaciones:
pop(): que devuelve el primer elemento del heap, que es el mínimo de la estructura.
push(x):que inserta el elemento x.
top(): que elimina el primer elemento del heap.
Estas tres operaciones se realizan manteniendo el invariante de la estructura.

[3] Teniendo en cuenta que cada tupla contiene <“próximo momento en que va a medir”, “numero de sensor”> , entonces, EL MENOR “próximo momento en que va a medir” (primera coordenada de la tupla) va a ser efecticamente EL PROXIMO EN MEDIR y de haber dos iguales, se desempata por el numero de sensor (segunda coordenada).

3.3. Ejercicio 3

3.3.a. Interpretación del problema

El problema consiste en encontrar, dado un conjunto de puntos en el plano (que asumimos que son enteros), una caja de tamaño b x h que encierre la mayor cantidad de estos puntos posible. Esta caja no puede rotarse de ninguna manera. Para justificar la solición provista, es necesario que tengamos primero en cuenta las siguientes consideraciones:

- Dado un conjunto de puntos, puede haber más de una posible solución. De hecho, es bastante probable que esto suceda (fig. 1).
- No obstante, algunos conjuntos pueden tener una única solución; por ejemplo aquel cuya distancia horizontal máxima entre dos puntos coincide con b y, a su vez, la distancia vertical máxima con h (fig. 2).

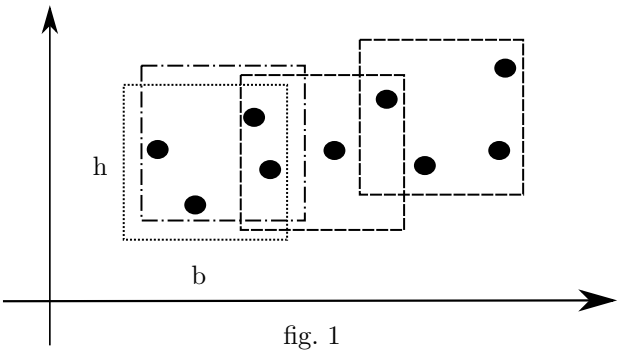


fig. 1

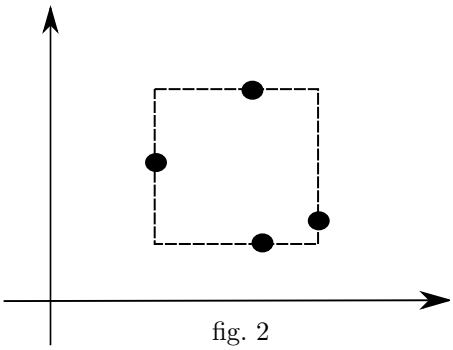


fig. 2

El problema pide escribir un algoritmo que devuelva una de todas estas posibles soluciones, o la única de haber una sola. Si llamamos C al conjunto de soluciones totales, definimos C' al subconjunto de este cuyas soluciones que cumplen la siguiente propiedad: representan aquellos rectángulos con un punto sobre su pared vertical izquierda y un punto sobre su piso. Consideramos elementos de C' a aquellos rectángulos donde el punto se encuentra tanto sobre la pared izquierda como sobre el piso, siendo esto posible únicamente cuando el punto se encuentra sobre la esquina inferior izquierda del rectángulo. La fig. 3 muestra las posibles soluciones al conjunto de puntos indicado en la fig. 1.

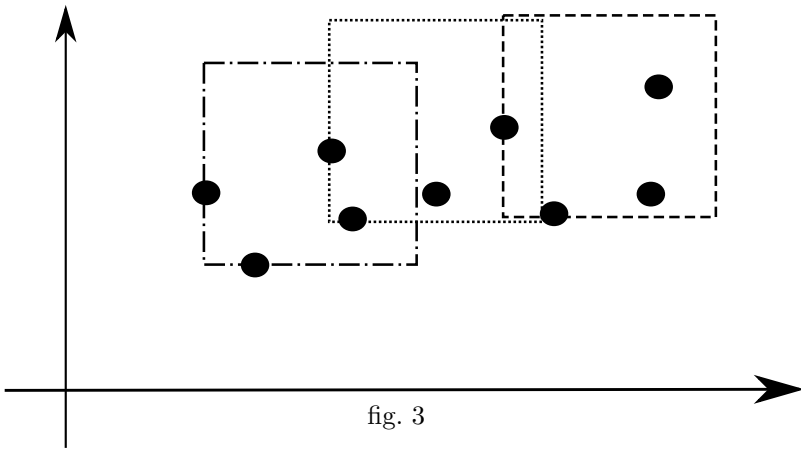


fig. 3

Aprovechando que el enunciado no pide nada más específico que encontrar una solucion cualquiera, para simplificar la implementación vamos a devolver la última solución que encontremos. A continuación vamos a demostrar que cualquier solucion perteneciente a C' es solucion del problema.

3.3.b. Demostraciones

Vamos a probar que, dado un conjunto de puntos en el plano que pueden ser encerrados por un rectángulo C de dimensiones b y h, siempre existe un rectángulo C' que tiene las mismas dimensiones que C y que encierra a los mismos puntos, con la particularidad de que C' tiene un punto tanto sobre su línea inferior como sobre su línea izquierda.

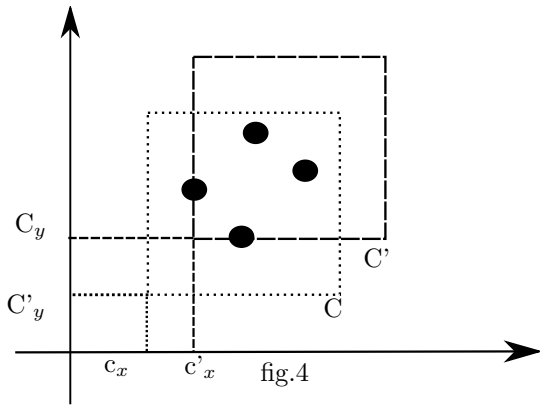


fig.4

Si P es el conjunto de puntos que pueden ser encerrados por un rectángulo de dimensiones b x h, esto significa que para todo par de puntos p y q tal que $p, q \in P$, se cumple que $dist_x(p, q) \leq b$ y que $dist_y(p, q) \leq h$[4].

Esto significa que, si C_x representa la coordenada en x en la que empeiza el rectángulo y C_y la coordenada en y, se cumple que, para todo punto p

$$C_x \leq p.x \leq C_x + b$$
$$C_y \leq p.y \leq C_y + h$$

Luego, el rectángulo propuesto tiene la propiedad de que $C'_x = p.x$ y que $C'_y = p.y$. Es trivial ver que, para todos los puntos p anteriores se cumple que:

$$C'_x = p.x \leq C'_x + b$$
$$C'_y = p.y \leq C'_y + h$$

Que era lo que queríamos verificar inicialmente.

[4]Donde amba funciones devuelven la distancia entre ambos puntos en el eje x y en el y respectivamente.

3.3.c. Pseudocódigo

Para tener una mayor comodidad al trabajar con los datos, vamos a trabajar con el struct punto[5], definido como:

```
struct{ int x; int y; } punto
```

Nuestra implementación utiliza dos funciones. La función global (algoritmo 1) va a iterar en todos los pares de puntos posibles y para cada par, va a hacer un llamado a la función contar_puntos_internos (algoritmo 5), que toma dos puntos, las dimensiones del rectángulo y el vector de puntos y cuenta cuantos de ellos entran en el rectángulo formado con los puntos que recibe. De no poder formar un rectángulo, devuelve cero. La funcion global siempre va a mantener el máximo en memoria, y va a devolver última solucion encontrada al terminar de iterar por todos los pares posibles de puntos.

Algorithm 4: Encontrar rectángulo con máxima cantidad de puntos internos - O(n^3)

```
Data: vector<punto> v, int h, int b
Result: punto coordRectSol
int t_max ← 0
int t ← 0
for int i = 0, i < longitud(v), i++ do
    for int j = 0, j < longitud(v), j++ do
        t ← contar_puntos_internos(v[i], v[j], v, b, h)
        if t > t_max then
            t_max ←t
            coordRectSol ← (v[i].x, v[j].y)
        end
    end
end
return coordRectSol
```

Donde la función contar_puntos_internos está definida como:

Algorithm 5: Función contar_puntos_internos - O(n)

```
Data: punto pIzq, punto pPiso, vector<punto> v, int b, int h
Result: int cantPuntosInternos
if (pPiso.x < pIzq.x ∨ pPiso.y > pIzq.y ∨ pPiso.x - b > pIzq.x ∨ pIzq.y - h > pPiso.y) then
    /* Caso en el que pIzq y pPiso no pueden formar un rectángulo */
    return 0
else
    /* Caso en el que si pueden formarlo, cuento la cantidad de puntos dentro del mismo.*/
    int cantPuntos ← 0for((int i = 0, i < longitud(v), i++))do
        if ( v[i].x ≥ pIzq.x ∧ v[i].x ≤ pIzq.x + b ∧ v[i].y ≥ pPiso.y ∧ v[i].y ≤ pPiso.y + h)
        then
            cantPuntos++
        end
    end
    return cantPuntos
end
```

[5]Obviamos mencionar sus constructores ya que no son relevantes a esta justificación.

3.3.d. Tests de Correctitud

A continuación vamos a someter a nuestro algoritmo a una serie de tests para exponer los resultados en diferentes casos bordes que fuimos identificando. Para cada test vamos a exponer la entrada, la salida y, en caso de que sea necesario, una justificación de la correctitud de la solución.

Test#1
Input: `k = 1, b = 1, h = 1, p = {(1,1)}`
Caracterización: Un único punto en el plano.
Output: Coordenadas de la caja: `(1,1)`, cantidad de puntos en su interior: `1`
Status: OK. Se obtiene una caja que tiene al único punto en su extremo inferior izquierdo.

Test#2
Input: `k = 6, b = 3, h = 4, p = {(1,1), (2,1), (3,1), (4,1), (5,1), (6,1)}`
Caracterización: Una línea recta horizontal de puntos en el plano.
Output: Coordenadas de la caja: `(1,1)`, cantidad de puntos en su interior: `4`
Status: OK. Se obtienen las coordenadas de una caja que contiene a los primeros 4 puntos de la recta.

Test#3
Input: `k = 6, b = 4, h = 3, p = {(1,1), (1,2), (1,3), (1,4), (1,5), (1,6)}`
Caracterización: Una línea recta vertical de puntos en el plano.
Output: Coordenadas de la caja: `(1,1)`, cantidad de puntos en su interior: `4`
Status: OK. Al igual que en el test anterior, se devuelven las coordenadas de la caja que recta contiene a los 4 puntos inferiores de la recta.

Test#4
Input: `k = 5, b = 6, h = 4, p = {(2,4), (6,2), (8,5), (4,6), (5,4)}`
Caracterización: Un conjunto de puntos a una distancia que sólo admite una única solución.
Output: Coordenadas de la caja: `(2,2)`, cantidad de puntos en su interior: `5`
Status: OK. Se devuelven las coordenadas de la única caja posible.

Test#5
Input: `k = 4, b = 5, h = 3, p = {(1, 1), (50,1), (1,50), (50,50)}`
Caracterización: Un conjunto de puntos muy separados.
Output: Coordenadas de la caja: `(1,1)`, cantidad de puntos en su interior: `1`
Status: OK. Se obtienen las coordenadas de la caja que encierra al primer punto. Las otras 3 soluciones son equivalentes.

Test#6
Input: `k = 15, b = 3, h = 2, p = {(1,1), (2,1), (3,1), (4,1), (5,1), (1,2), (2,2), (3,2), (4,2), (5,2), (1,3), (2,3), (3,3), (4,3), (5,3)}`
Caracterización: Un conjunto denso de puntos.
Output: Coordenadas de la caja: `(1,1)`, cantidad de puntos en su interior: `12`
Status: OK. El algoritmo devuelve las coordenadas de una de las posibles cajas que contienen la mayor cantidad de puntos (12) dentro del bloque de puntos.

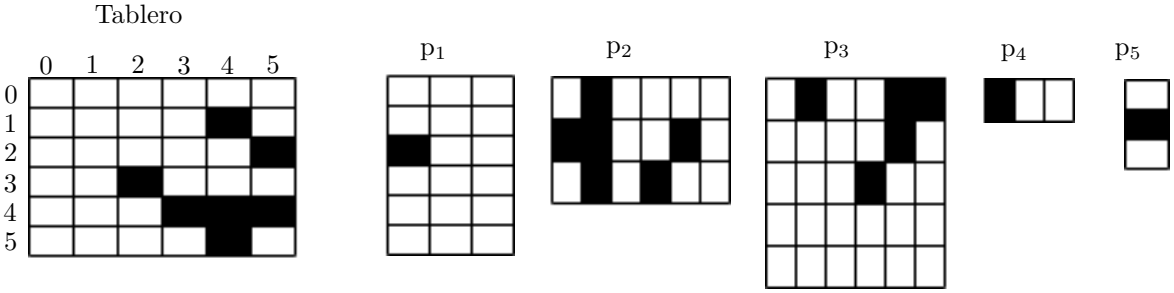
3.4. Ejercicio 4

3.4.a. Interpretación del problema

Dado un tablero de dimensiones n x m, coloreado de manera arbitraria con dos colores diferentes y un conjunto de piezas rotables, también rectangulares, coloreadas de la misma manera que el tablero, el ejercicio pide encontrar una solución óptima para llenar dicho tablero con un subconjunto mínimo de piezas, bajo las siguientes reestricciones:

- Una pieza sólo puede ser ubicada en el tablero si y sólo si la dispocisión de los colores de la rotación de dicha pieza coinicide con la dispocisión de los colores del tablero partiendo de la coordenada en la que se ubicaría.
- Dos piezas no se pueden solapar.
- No pueden haber cuadrados de las piezas que sobresalgan del tablero.

Un ejemplo de tablero seria la siguiente configuración:



Se ve fácilmente que la solución es rotar a p_1 2 veces en sentido horario y ubicarla en la posición $(0, 0)$, y luego rotar a p_2 3 veces en sentido horario y ubicarla en $(3, 0)$. Se puede ver también que es posible obtener otra solución usando las piezas restantes, pero esta solución no nos interesa ya que usa más piezas que la solución anterior.

La forma que proponemos para resolver el ejercicio es básicamente intentar poner cada pieza en el primer lugar de arriba a la izquierda que esté libre, cuando este encuentre una pieza que entra, entonces va a marcar en el tablero que esa pieza fue agregada y va a intentar con la siguiente pieza, en el siguiente lugar libre que este de izquierda a derecha y de arriba a abajo, utilizamos técnicas de backtracking para ramificar el problema en varias instancias donde se prueben diferentes soluciones que constan de intentar poner en el siguiente casillero en blanco todas las piezas restantes no utilizadas, y por cada una volvemos a intentar con todas las piezas restantes, cabe destacar también que hay ramificaciones en las cuales una pieza puede entrar en mas de una rotación, que tambien genera diferentes ramificaciones.

3.4.b. Pseudocódigo

Nuestro algoritmo trabaja de la siguiente manera:

Algorithm 6: Encontra la optima configuracion de piezas para ubicar en el tablero

```
Data: tablero t, piezas p
if (cantidadColoresBlancos(t) > cantidadColoresBlancos(p)∨
t.n * t.m > cantidadCasilleros(p)) then
    return false;
end
piezas pUsadas
for int i = 0, i < longitud(piezas), i++ do
    if (ej4(t, p, i, pUsadas)) then
        return true;
    end
    return false;
end
```

Algorithm 7: Encontra la optima configuracion de piezas para ubicar en el tablero llamado anteriormente

```
Data: tablero t, piezas p, int i, piezas pUsadas
for int b = 0, b < 4, b++ do
    if t.entraPieza(t, p[i]) then
        pintoTablero(t, p[i]);
        guardarPosicionDondeEntroLaFicha(p[i]);
        pUsadas ← p[i];
        if buscoProximaPosicionEnBlanco(t) then
            return componerResultado(pUsadas);
        end
        for int i = 0, i < longitud(piezas), i++ do
            if (ej4(t, sacarPiezaUsada(p,i), i, pUsadas)) then
                return true;
            end
        end
    end
    rotarPieza(p[i]);
end
return false;
```

k = cantidad de piezas, $(n * m)$ = dimensiones del tablero.

`entraPieza()`, `pintoTablero()`, `buscoProximaPosicionEnBlanco()` tienen todas una complejidad de $O(n * m)$, `guardarPosicionDondeEntroLaFicha`, `componerResultado` y `sacarPiezaUsada` tienen complejidad temporal lineal.

La primer funcion de ej4 se fija primero si el tablero tiene solución, contando la cantidad de casilleros blancos y comparandolo con la cantidad de piezas blancas del tablero, esto toma, por cada pieza k, contar su cantidad de casilleros blancos y ya que la mayor pieza puede ser tan grande como el tablero, esta complejidad es $O(k * (n * m))$.

Luego, por cada pieza k, se hace un llamado recursivo a la segunda funcion de ej4, la cual ejecuta una vez las funciones `entraPieza()`, `pintoTablero()`, `buscoProximaPosicionEnBlanco()` y hace un llamado recursivo a la misma función con una pieza menos para todas las piezas restantes, esto quiere decir que primero las 3 funciones tomando una complejidad de $O(3n * m) = O(n * m)$ y la llamda recursiva cuesta $O(k! * costoDeLaFuncion) = O(k! * (n * m))$.

La complejidad para el ejercicio entonces es $O(k * (n * m) + k! * (n * m)) = O(k! * (n * m))$ donde `k` es la cantidad de fichas, `n` es el ancho del tablero y `m` el alto.

Explicacion del funcionamiento del algoritmo.

1. Busco la primer cuadricula que esté libre.
2. Intento poner la primer pieza sin usar ahí, si no entra, pruebo sus rotaciones.
3. Por cada rotación de la pieza que haga, llamo de vuelta a la función con las piezas restantes que no se usaron, y guardando las piezas que use con su cantidad de rotaciones en un resultado parcial"
4. Si la función no encuentra una pieza que pueda poner en la cuadricula libre entonces el problema no tiene solución y se descarta, y se sigue probando con las otras ramificaciones.
5. Una vez que encuentra una solución, devuelve la lista de piezas usadas con la toda la información necesario como la rotación y el x e y de su ubicación en el tablero.

La poda del árbol que genera las multiples soluciones de las piezas se poda cuando, en cada paso, se intenta poner una pieza y esta pieza no entra de ninguna manera, entonces se descarta esa solución por más de que hayan piezas restantes, ya que el caso en que esa pieza no es la siguiente esta contemplado en pasos recursivos anteriores, de esta manera se evita repetir configuraciones de piezas.

3.4.c. Demostraciones

El procedimiento anterior resuelve correctamente el ejercicio ya que arma un arbol con todas las posibles combinaciones de piezas a poner, incluyendo sus rotaciones, si tengo N piezas e intento poner la primera y sus 4 rotaciones, y por cada una intento poner la segunda, como el algoritmo siempre va a agarrar la proxima posicion vacia y por cada posicion va a intentar poner una de las piezas restantes y por cada pieza que ponga va a llamar a la funcion nuevamente con las piezas restantes. Esto cubre efectivamente todas las combinaciones de piezas que pueda poner ya que genera un arbol de soluciones hasta encontrar la solucion correcta. Ademas, notar que no se devuelven soluciones parciales ya que cuando no quedan mas piezas por probar y no se completo el tablero, ya no es considerada esa convinacion de piezas una posible solicion. En otras palabras, la solucion es un conjunto minimal de piezas que forman el tablero.

Ahora veamos que cuando el algoritmo tiene solución, esta es correcta. Para esto usemos el invariante de la funcion **entra()**.
entra(t,p): Dado un tablero T y una pieza p, me asegura que p entra en el proximo lugar[6].
En cada paso vamos buscando la pieza que encaja en el tablero y esto lo realizamos hasta que completamos el tablero o no entra ninguna otra pieza. Esto lo hacemos en cada paso para todas las combinaciones (incluidas sus rotaciones); Entonces sea S=$\{s_1,..,s_n\}$ una solucion donde s_i ($1 \leq i \leq n$) es una pieza, sabemos que todas las piezas unidas en el orden correcto posee la misma configuracion que el Tablero. Supongamos que S no es solución, entonces existe un s_i tal que no cabe en T, es decir, que todas las piezas, menos esa, ya esten ocupando sus respectivos lugares, pero eso es imposible porque la funcion entra() me asegura que la pieza esta en una posicion valida para la posicion que se calculo, entonces, todos los s_i caben en el T y por lo tanto S, de existir, es correcto.

Finalmente sabemos que solo hay soluciones correctas, que no son ni vacias ni parciales, solo resta tomar la solucion que utilice la menor cantidad de piezas ya que esta es la solucion optima que pide el enunciado, esto lo hacemos realizando una poda de forma que en el momento en que encuentre una solucion con K piezas, el resto de las posibles combinaciones que superen el valor K seria descartadas automaticamente.

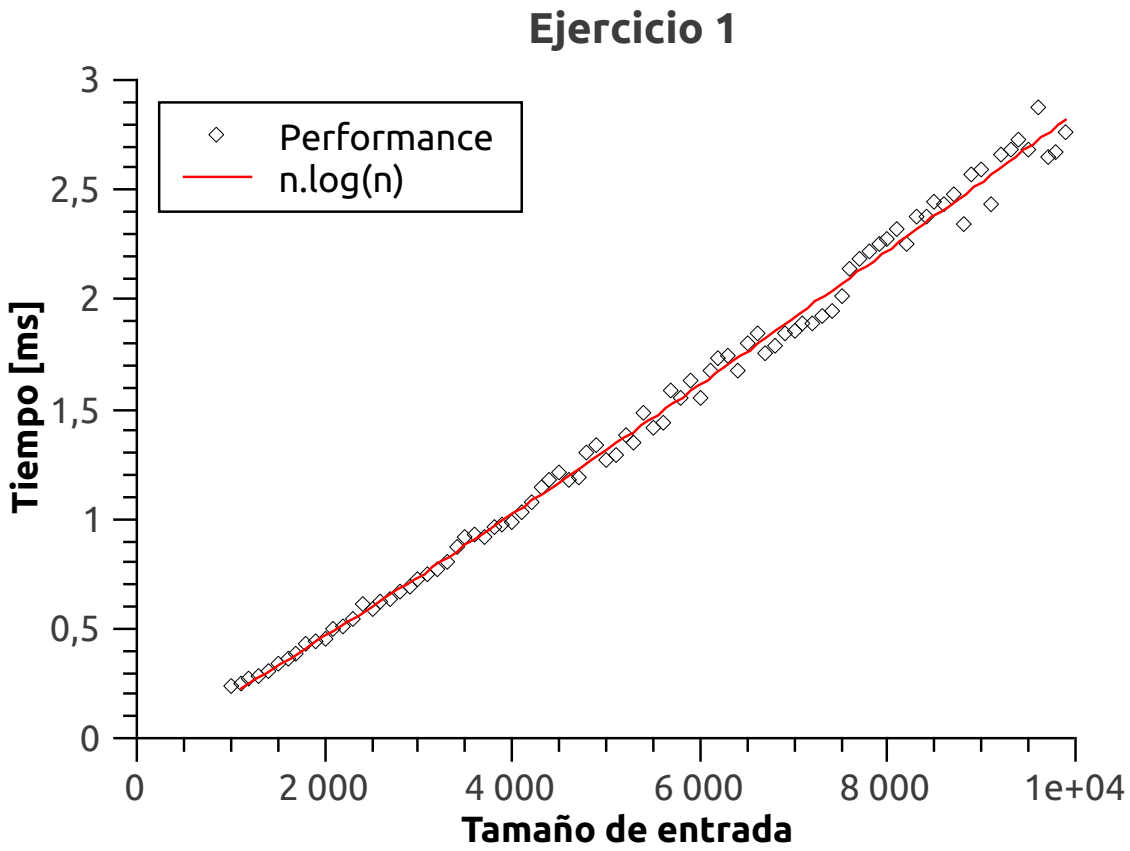
3.4.d. Casos de test

Ademas de test aleatorios, consideramos los siguientes casos que nos parecen interesantes:

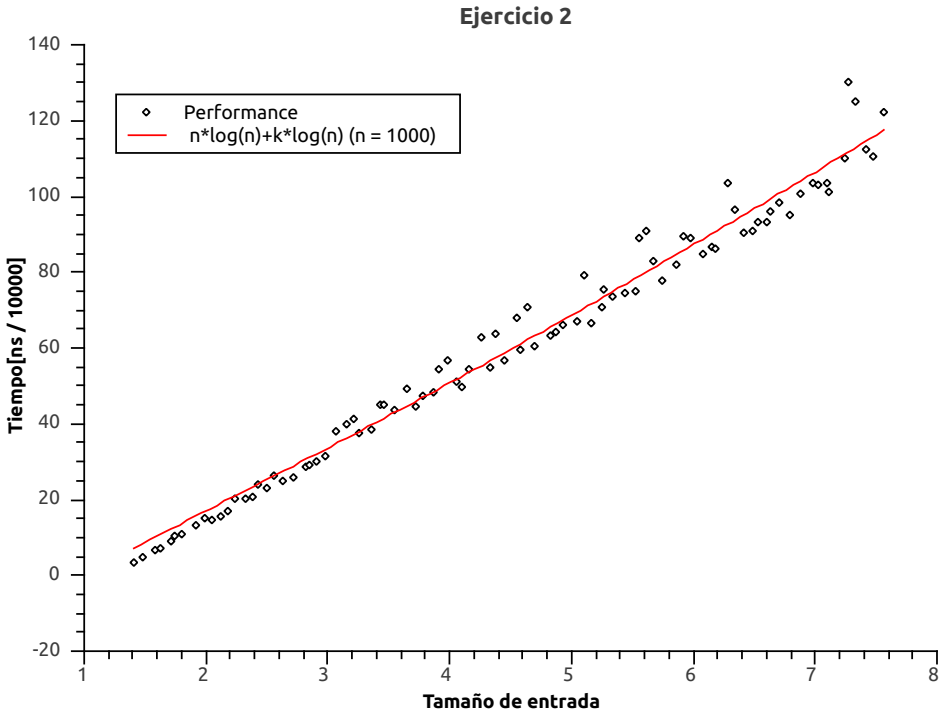
[6] El proximo lugar esta definido por el tablero y es el primero libre de arriba hacia abajo, de izquierda a derecha.

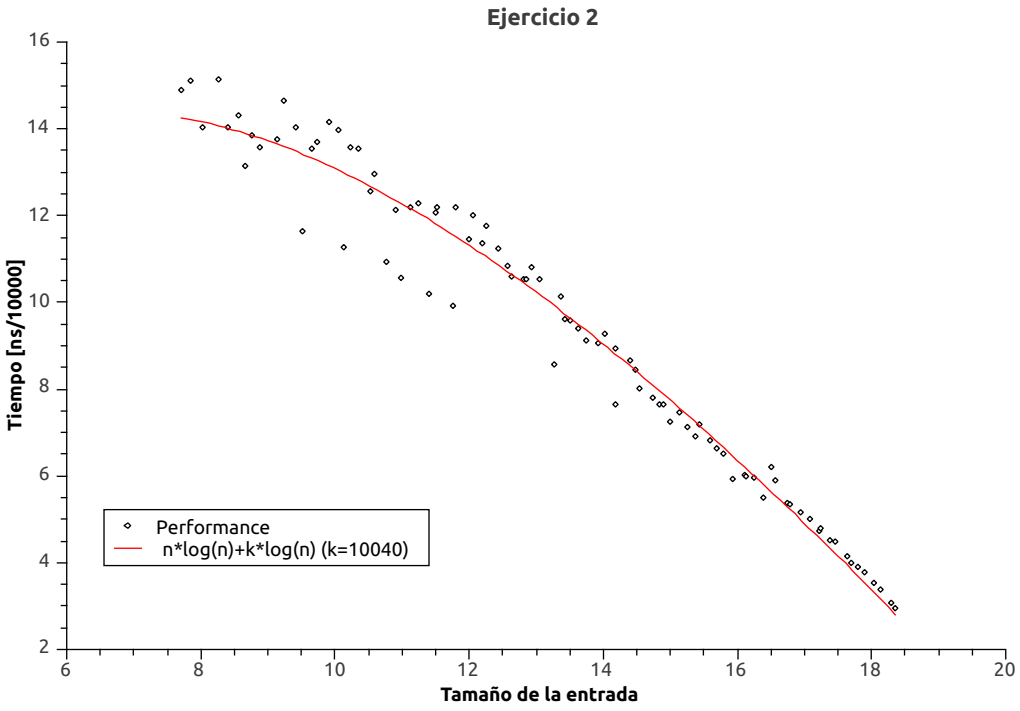
4. Testing de Performance

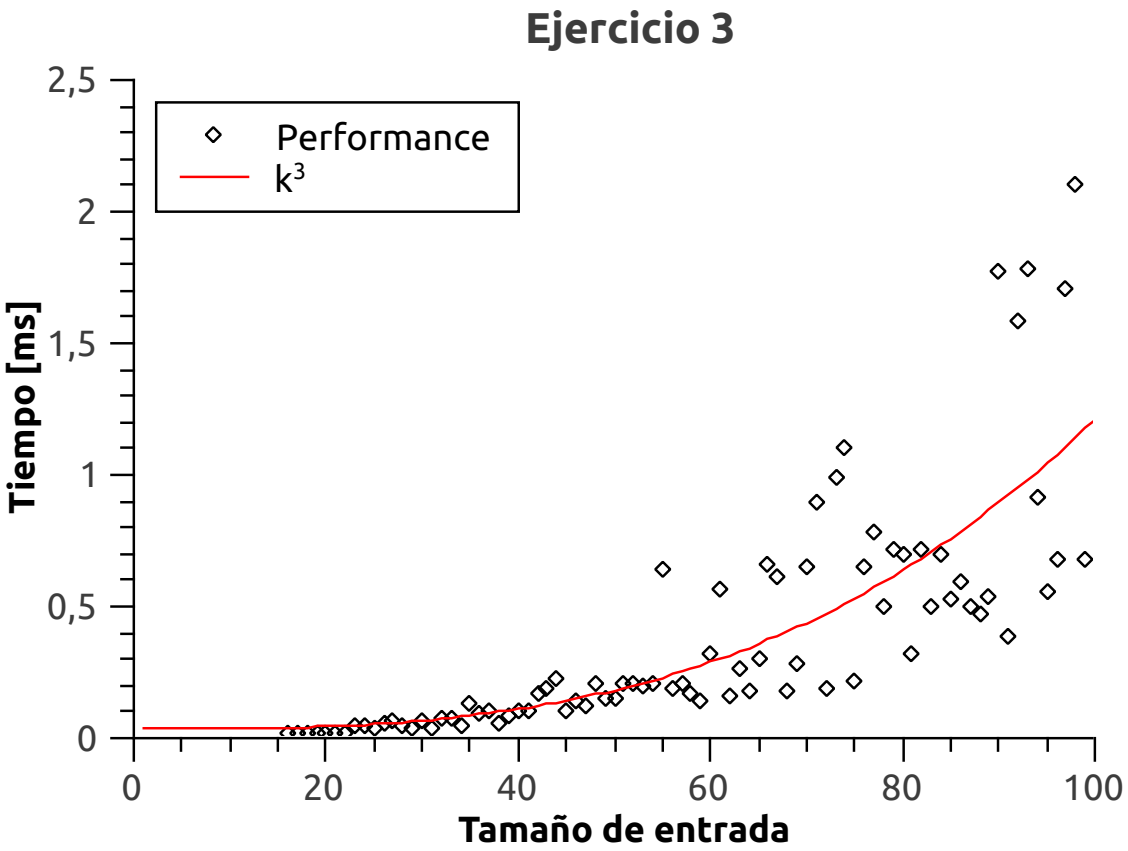
A continuación presentamos los tests que realizamos para medir la performance de nuestras implementaciones. La manera en la que los generamos fue alimentándole al programa una serie de datos de tamaño creciente. Para clarificar la visualización, graficamos también la función que representa la complejidad de peor caso en las implementaciones. Pudimos ver como las sucesivas ejecuciones del programa respetan los valores teóricos esperados y la incrementación lógica en los tiempos de ejecución.


Ejercicio 1
Performance
n.log(n)
Tiempo [ms]
Tamaño de entrada
0
0,5
1
1,5
2
2,5
3
0
2 000
4 000
6 000
8 000
1e+04

En este caso hicimos dos graficos, en uno fijamos la variable n que representa a la cantidad de sensores en 1000 ya que consideramos es un número razonable y en el otro fijamos la variable k que determina la k-esima medición a encontré en 10040 ya que consideramos un número razonable, y ya que los n crecen desde 1000 hasta 10000 elegimos este número para evitar que se den los casos k≤n que son O(1).






Ejercicio 3
Performance
k3
Tiempo [ms]
Tamaño de entrada
2,5
2
1,5
1
0,5
0
0
20
40
60
80
100

5. Conclusiones

Si bien nos quedó pendiente para la segunda entrega el ejercicio 4, pudimos llevar a cabo la resolución de los problemas 1 a 3 junto con sus mediciones de performance. Fue interesante graficar los resultados y comparar los tiempos de ejecución entre todos los programas. Asimismo, dejamos pendientes estos aspectos para el ejercicio restante.