DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

# Trabajo Práctico I

4 de septiembre de 2013 — Algoritmos y Estructuras de Datos III


| Integrante | LU | Correo electrónico |
|---|---|---|
| Amil, Diego Alejandro | 68/09 | amildie@gmail.com |
| Barabas, Ariel | 755/11 | ariel.baras@gmail.com |
| Aleman, Damian Eliel | 377/10 | damianealeman@gmail.com |
| Fernández Gonzalo Pablo | 836/10 | ralo4155@hotmail.com |


| Instancia | Docente | Nota |
|---|---|---|
| Primera entrega | | |
| Segunda entrega | | |




**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. 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 segundo cuatrimestre del año 2013. Este trabajo práctico consiste en la realización de un análisis teórico-experimental de un conjunto de problemas propuestos por la cátedra. Se requiere, para cada uno de los tres problemas, la implementación de un algoritmo que satisfaga criterios tanto de correctitud como de complejidad temporal.

Vamos a exponer, para cada uno de los problemas, los siguientes apartados:

- Una interpretación del enunciado, deallando ejemplos y/o casos particulares.
- Una solución propuesta.
- Un pseudocódigo que implemente dicha solución, junto con una explicación de su correctitud y una justificación de su complejidad.
- Un apartado de testing, tanto de correctitud como de performance.

# 2. Pautas de Implementación

El lenguaje elegido para la implementación de los algoritmos es `C++`. De ser necesario vamos a utilizar la librería standard del mismo y aclarar los costos de las operaciones en cuestión. En el caso de tener que implementar una clase propia para simplificar el código o proveer de cierto encapsulamiento, los costos de los métodos de la misma serán verificados y justificados. La estructura de directorios que utilizaremos para la implementación será la siguiente para todos los ejercicios:

```
\codigo
 timer.h
     \ej1
          ej1.cpp
          ej1.h
          Makefile
          \test
               ej1_test.cpp
               ej1_test.sh
```

El archivo `timer.h` contiene las funciones necesarias pera medir el tiempo de ejecución de nuestros programas. Vamos a usar la función `clock_gettime` de la librería `time.h`. Estas funciones son idénticas para las mediciones en todos los ejercicios. Las funciones pedidas por la cátedra se encuentran en el archivo `ej1.h`, y se incluyen en `ej1.cpp`. Este archivo trabaja con entrada y salida standard de manera que, para ejecutar un programa con su respectivo conjunto de casos, será suficiente con direccionarlo por consola escribiendo `./ej1<ej.in`. Obviaremos mencionar detalles referentes a la carga de datos en las implementaciones.

El directorio `/test` contiene los archivos necesarios para efectuar tests de performance de nuestros programas. Para hacer esto, sólo es necesario correr el programa `ej1_test.cpp` pasandole 3 parámetros, que son: el `n` máximo hasta el cual testear, el salto entre `n` y la cantidad de tests para cada `n`. De esta manera ejecutar, por ejemplo `./ej1_test 10000 500 20` va a generar y correr tests con $500 \leq n \leq 10000$, con un incremento de 500 entre cada `n` y 20 tests distintos para cada `n`. La salida de la ejecución consiste en los tiempos de ejecución medidos para cada `n`.

El script `ej1_test.sh` encapsula la funcionalidad del programa anterior. Llamarlo con los mismos parámetros va a automáticamente compilar todo lo necesario, llamar a al programa para generar y correr los tests y graficar todo, generando un `.pdf` con el gráfico en ese mismo directorio. Este es el mismo proceso que empleamos para generar los gráficos de este informe.

# 3. Ejercicio 1

## 3.1. Interpretación del enunciado

En este ejercicio se habla de un señor llamado Pascual, quien trabaja en un centro de distribución del correo y se encarga de ubicar los paquetes que llegan al centro, en camiones para ser entregados. Cada uno de estos paquetes tiene un peso determinado, por lo tanto un camión solo puede ser cargado con un conjunto de paquetes tales que la suma de sus pesos no exceda el límite de peso del camión, el cual es el mismo para todos los camiones. Pascual tiene una forma particular de hacer su trabajo, nos dicen que intenta ubicar cada paquete que llega en el camión menos cargado de los que están en uso y si no llegara a entrar, usa un nuevo camión de los infinitos que tiene a su disposicion para guardarlo.

Como se trata de pesos, todos estos valores son positivos y cada paquete entra en un camión, así que el peor caso (con respecto a recursos materiales) sería necesitar un camión para cada paquete.

El objetivo es escribir un algoritmo que permita, conociendo el límite de peso de los camiones y los pesos de cada paquete, saber cuántos camiones necesitará Pascual para organizar todos los paquetes y que tan cargado estará cada uno, según su metodología de distribución. Se pide que la complejidad del algoritmo sea estrictamente menor a $O(n^2)$ siendo $n$ la cantidad de paquetes.

## 3.2. Resolución

El método inmediato para resolver un problema de predicción es simular la situación. Se ha desarrollado un algoritmo que reproduce el método de Pascual y, una vez finalizado, obtiene los camiones necesarios y sus respectivas cargas (en peso). A continuación se muestra el pseudocódigo de la solución. La entrada se supone ya leída y guardada en la variable *capacidadMaxima* (la capacidad de cada camión) y en una lista de paquetes (el peso de cada paquete) ordenada según su llegada. Además vamos a usar una cola de prioridad para almacenar los camiones, la cual se va a agrandar tanto como necesitemos.

**Algoritmo 1:** Llenado de camiones

```
   Data: int capacidadMaxima, lista(Paquete) paquetes, cola_prioridad(Camión) camiones
1  foreach Paquete p ∈ paquetes do
2      if peso(camionMenosCargado) + peso(p) ≤ capacidadMaxima then
3          le sumo peso(p) a peso(camionMenosCargado);
4      else
5          agrego otro camión con peso p a camiones;
```

Se asume que la instrucción *ForEach* (línea 1) recorre la lista en orden. Lo que se hace es tomar cada paquete de la lista y ubicarlo en un camión. Tal como lo haría Pascual, si el paquete entra en el camión menos cargado (línea 2) se lo ubica en él. Si no, se lo ubica en un camión vacío (línea 5). *camionMenosCargado* representa al camión cuya carga actual es la menor de entre los camiones utilizados (que ya tienen al menos un paquete). Es el menor elemento de *camiones* (el elemento con mayor prioridad de la cola) y debe ser actualizado tras cada iteración, ya que se actualiza la carga del camión menos cargado (por lo que podría dejar de ser el menos cargado) o la de un camión nuevo (el cual podría pasar a ser el nuevo menos cargado).

## 3.3. Complejidad

Como usamos una cola de prioridad [1] implementada con un heap, obtener el camión menos cargado consiste en tomar el mínimo del heap en tiempo constante. Si se modifica su valor, este debe ser extraído del heap y reinsertado con otro valor (su nuevo peso). Ambas operaciones se realizan en tiempo logarítmico en el tamaño del heap. Si en lugar de modificar el mínimo, se agrega un nuevo elemento, también se debe realizar una inserción en tiempo logarítimico al heap.

[1] Introduction to Algorithms, Thomas H. Cormen. Pagina 163

El tamaño del heap es la cantidad de camiones utilizados, pero está acotada por la cantidad de paquetes, ya que no tendría sentido usar más camiones que paquetes. Finalmente, la complejidad del algoritmo es O($n \log(n)$) con $n$ la cantidad de paquetes, ya que se realizan $n$ iteraciones y en cada una se realizan a lo sumo dos operaciones logarítmicas en la cantidad de paquetes.

## 3.4. Implementación

El conjunto de camiones es implementado con la cola de prioridad[2] de la librería estándar de C++. En lugar de guardar la carga de cada camión, se guarda la capacidad libre (es decir la capacidad máxima menos la suma de los pesos de los paquetes), por lo que el primer elemento no es el de menor valor sino el de mayor. Tanto la estructura como las complejidades son equivalentes. Este heap nos permite acceder en tiempo constante al primer elemento del conjunto y agregar y sacar elementos en O(log(tamaño del heap)). La operación de sumar el peso de un paquete en realidad es sacar un elemento y agregar otro con el peso sumado, lo cual es de complejidad logarítmica según la documentación de la librería.

## 3.5. Demostración de Correctitud

En esta sección demostraremos por el absurdo que la distribución de los paquetes de nuestro algoritmo es igual a la de Pascual.

Supongamos que la distribución de los paquetes de nuestro algoritmo no es la misma a la de Pascual. Entonces debería existir un paquete que no fue asignado por nuestro algoritmo al mismo camión al que lo asignó Pascual. Dado que recorremos los paquetes en el mismo orden que lo hace Pascual (el orden de llegada), sea $P_i$ el primer paquete que no se asigna de la misma manera. Entonces todos los paquetes anteriores (de $P_1$ a $P_{i-1}$) fueron asignados al mismo camión. Por lo tanto hasta ese momento la carga de cada camión es igual para ambos métodos. Existen dos casos posibles:

1. El paquete $P_i$ entra en el camión menos cargado (el peso de $P_i$ es menor o igual a la capacidad libre del camión).
2. El paquete $P_i$ no entra en el camión menos cargado (el peso de $P_i$ es mayor a la capacidad libre del camión).

En el primer caso Pascual lo ubicaría en el camion menos cargado. Como

$$\text{peso}(P_i) \leq \textit{capacidadLibre}(\text{CMC}) \Rightarrow \textit{peso}(\text{CMC}) + \textit{peso}(\text{P}_i) \leq \text{capacidadMaxima}$$

entonces nuestro algoritmo entra por la primera guarda y también ubica el paquete $P_i$ en el camion menos cargado(CMC).

En el otro caso Pascual lo ubicaría en un nuevo camión vacío. Como no se cumple la condición enunciada previamente, el algoritmo entra por la sentencia correspondiente al *else*, lo que significa que agrega un camión vacío y ubica el paquete en el nuevo camión. Como en ambos casos el algoritmo determinó ubicar el paquete en el mismo camión que lo haría Pascual, el paquete $P_i$ se lo asigna de la misma manera, lo cual genera un absurdo con la suposición de que el paquete $P_i$ se asigna de forma distinta.

## 3.6. Testing

**Correctitud**

**Performance**

[2] http://en.cppreference.com/w/cpp/container/priority_queue

amsmath

# 4. Ejercicio 2

## 4.1. Interpretación del enunciado

Se tienen una cantidad $n$ de cursos con sus respectivas fechas de inicio y finalización representadas con números enteros positivos. El objetivo de este ejercicio es desarrollar un algoritmo que, dadas las fechas de inicio y fin de cada curso, encuentre un subconjunto de cursos que no se solapen entre sí, es decir, sean compatibles. Se pide que la complejidad del algoritmo sea estrictamente menor a $O(n^2)$ siendo $n$ la cantidad de cursos.

Para este ejercicio, asumiremos que los cursos no se interrumpen. Es decir, por ejemplo, que el curso que arranca el día 5 y termina el día 8 va a dictarse también los días 6 y 7. Entonces lo que nos queda es un problema de optimización: ¿Cómo elegir la mayor cantidad de cursos posibles de manera que ningún par de cursos dado tenga días en común?. En particular para que se puedan tomar dos cursos, la fecha de inicio de uno de los dos tiene que ser mayor (estricto) que la fecha de finalizacon del otro curso.

Cabe destacar que lo que este problema busca es maximizar la cantidad de cursos disponibles para ser cursados por un único alumno, y no buscar que haya más días cubiertos por un curso. También hay que mencionar que los cursos no tienen ningún valor numérico asignado.

## 4.2. Resolución

Para resolver este ejercicio se desarrolló un algoritmo goloso. Se busca devolver un conjunto de cursos. El algoritmo itera agregando, en caso de ser posible, un curso en cada iteración. Mientras se pueda seguir agregando cursos (no se puede si todos los restantes se solapan con alguno seleccionado), se agrega de los que quedan, el de menor fecha de finalización que no se solape con alguno seleccionado. Luego se devuelven los cursos seleccionados.

A continuación expondremos un pseudocódigo de la solución implementada. Suponemos los datos levantados directamente desde la entrada y almacenados en un vector llamado *entrada*.

**Algoritmo 2:** Resolución Golosa Ejercicio 2

```
   Data: vector(Curso) entrada
1  vector(Curso) cronograma_aceptado ⟵ ∅
2  while puedoAgregarCursos do
3  |  cronograma_aceptado ⟵ cronograma_aceptado ∪
4  |  c ∈ entrada / sonCompatibles(c, cronograma_aceptado) ∧
5  |  fechaFin(c) ≥ fechaFin(c') ∀c' ∈ entrada
6  return cronograma_aceptado
```

El vector de cursos *entrada* contiene todos los cursos de la instancia, mientras que el vector *cronograma_aceptado* contiene los que se van agregando a la solución. La función *sonCompatibles* devuelve si el curso $c$ no se solapa con algún curso del vector de aceptados.

## 4.3. Complejidad

Para obtener en cada iteración del ciclo principal (línea 2) el curso con menor fecha de finalización, se pueden ordenar, antes de entrar al ciclo, los cursos según su fecha de finalización. Ordenar los cursos puede hacerse en $O(nlogn)$ con $n$ la cantidad de elementos a ordenar, en este caso, la cantidad de cursos de la entrada. Luego, con los cursos ordenados, el ciclo debe iterar sobre los $n$ cursos, evaluando si cada uno es compatible con el resto del cronograma aceptado. Esto puede hacerse determinando si se solapa con el anterior (como se agregan en orden, si se solapa con alguno, tiene que ser con el último agregado), lo cual tiene complejidad O(1), ya que es una comparación de enteros (la fecha de finalización del curso que se itera y la del último curso agregado). Entonces,

El ciclo que recorre los $n$ cursos del cronograma de entrada tiene una complejidad de $O(n)$. Luego, la complejidad de la función *resolver* termina siendo de $T(n) = O(nlogn) + O(n) = O(nlogn)$.

## 4.4. Implementación

La implementación de este algoritmo en C++ primero ordena los cursos según su fecha de finalización. Luego recorre todos los cursos en orden y para cada curso, evalúa si puede ser agregado o no. La función de ordenamiento está implementada mediante la función sort dentro de la STL de C++. La misma tiene una complejidad de $O(nlogn)$ [3]. En cada iteración del ciclo que recorre los cursos, no hace más que comparar la fecha de finalización del curso que se itera con la del último curso agregado. Si es mayor estricta, agrega el curso a los cursos aceptados.

## 4.5. Demostración de Correctitud

En esta sección vamos a demostrar que el algoritmo resuelve el problema de encontrar la máxima cantidad de cursos que no se superponen. Para eso veremos que:

1. Sea B un conjunto de cursos que maximiza la cantidad de cursos compatibles, y sea A el conjunto que siempre elige la de menor fecha de finalización que sea compatible.
   Luego #A = #B.

2. Nuestro algoritmo siempre elige la de menor fecha de finalización que sea compatible

**Sea B un conjunto de cursos que maximiza la cantidad de cursos compatibles, y sea A el conjunto que siempre elige la de menor fecha de finalización que sea compatible. Luego #A = #B.**

Para demostrar esto, emplearemos la siguiente notación.
Sea A = $\{A_1, A_2, A_3...A_k\}$ el conjunto de cursos que da nuestro algoritmo, con

$$fechaFin(A_i) < fechaFin(A_j) \quad \forall i, 1 \leq i < j \leq k \tag{1}$$

con $k$ la cantidad de cursos de nuestra solución. Por otro lado definimos B = $\{B_1, B_2, B_3..B_m\}$, con

$$fechaFin(Ai) < fechaFin(A_j) \quad \forall i < j, 1 \leq i < j \leq m \tag{2}$$

con $m$ la cantidad de cursos de la solución óptima. Como B es el conjunto de cursos de la solución óptima vale que $k \leq m$ .

Lema: Primero veremos que $\forall i : 1 \leq i \leq k$ vale: $FechaFin(A_i) \leq FechaFin(B_i)$

DEMOSTRACION DEL LEMA (por inducción)
Queremos ver que:

$$fechaFin(A_i) \leq fechaFin(B_i) \quad \forall i : 1 \leq i \leq k \tag{3}$$

Caso Base: Queremos ver que

$$fechaFin(A_1) \leq fechaFin(B_1) \tag{4}$$

Esto es cierto ya que al inicio del algoritmo, cualquier curso es compatible (ya que el conjunto de cursos seleccionados hasta ese momento es vacío) y además el algoritmo siempre selecciona el curso compatible con menor fecha de finalización. Luego queda probado el caso base.
Paso Inductivo:
Lo que tenemos que demostrar:

$$\forall c : 1 \leq c < k \quad fechaFin(A_c) \leq fechaFin(B_c) \quad \Rightarrow fechaFin(A_{c+1}) \leq fechaFin(B_{c+1}) \tag{5}$$

[3] http://www.cplusplus.com/reference/algorithm/sort/

Como vale la hipótesis inductiva sabemos que $fechaFin(A_c) \leq fechaFin(B_c)$.
También sabemos que la $fechaFin(B_{c+1}) > fechaInicio(B_{c+1})$
y, como B es un conjunto que tiene a cursos compatibles tenemos que $fechaInicio(B_{c+1}) > fechaFin(B_c)$. Luego con la afirmación anterior sumada a la hipótesis inductiva podríamos seleccionar el curso $B_{c+1}$:

$$fechaFin(A_c) \overset{\text{porHI}}{\leq} fechaFin(B_c) \overset{\text{Bconjcompatible}}{<} fechaInicio(B_{c+1}) \tag{6}$$

Ya que nuestro algoritmo selecciona el curso compatible con menor fecha de finalización, se deduce que $fechaFin(A_{c+1}) \leq fechaFin(B_{c+1})$ □

Con el lema podremos probar que vale que $k \geq m$, y como sabíamos que $k \leq m$ tendremos que el conjunto de soluciones dadas por el algoritmo es óptimo en cuanto a maximizar la cantidad de cursos que sean compatibles.

DEMOSTRACION DE $K \geq m$ (por el absurdo):

Sea B el conjunto que maximiza la cantidad de cursos con un cardinal de m, y el conjunto que devuelve nuestro algoritmo un cardinal de k. Supongamos que k<m y lleguemos a un absurdo. Como B es un conjunto que tiene cursos compatibles, el curso $B_{k+1}$ es compatible con el conjunto $\{B_1, B_2, B_3..B_k\}$. Del lema sabemos que $FechaFin(A_k) \leq fechaFin(B_k)$, por lo tanto $B_{k+1}$ también sería compatible con el conjunto $\{A_1, A_2, A_3..A_k\}$, por lo tanto ese curso entraria en el cronograma de cursos acepatados de nuestro algoritmo.
ABSURDO (ya que supusimos que #A = k )□

**Nuestro algoritmo siempre elige la de menor fecha de finalización que sea compatible**

Es preciso notar que nuestro algoritmo ordena en forma creciente de acuerdo a la fecha de finalización y luego recorre desde el de menor fecha, hasta el de mayor fecha de finalización.
Por otro lado siempre elige tomar un curso más, si este es compatible
ya que de la forma que recorremos vale que:

$$fechaFin(curso_{iteracionj}) \geq fechaFin(curso_{iteracioni}) \quad \forall j > i \tag{7}$$

Además el algoritmo en cada iteración comprueba que:

$$fechaInicio(curso_{iteracionj}) > ultimaFechaFin \tag{8}$$

para que no haya solapamientos entre el curso a elegir con los ya elegidos anteriormente.
Esto es válido debido a que la $fechaInicio(curso_{iteracionj}) > ultimaFechaFin$ y
$fechaFin(curso_{iteracionj}) \geq fechaFin(curso_{iteracioni}) \quad \forall j > i$

## 4.6. Testing

**Correctitud**
Los tests expuestos a continuación fueron diseñados con el fin de verificar diferentes casos particulares que pudimos identificar. 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**
**Caracterización:** Varios cursos que no coinciden en ningún dia.
**Input:** 4 1 2 3 4 5 6 7 8

**Output:** 1 2 3 4
**Status:** OK. Los 4 cursos se pueden tomar.

**Test#2**
**Caracterización:** Varios cursos, todos incompatibles entre si.
**Input:** 3 1 5 2 6 3 7
**Output:** 1
**Status:** OK. Si bien elegir cualquiera de los cursos es solución, nuestro algorimo elije el primero analizado.

**Test#3**
**Caracterización:** Prioriza obtener la mayor cantidad de cursos, por sobre la mayor cantidad de horas cursadas.
**Input:** 4 9 15 3 7 21 27 1 31
**Output:** 2 1 3
**Status:** OK. Se descarta el curso que dura desde el 1ro hasta el 31 del mes, para que se puedan dictar los 3 cursos que no se solapan entre si.

**Test#4**
**Caracterización:** Cursos encadenados.
**Input:** 11 1 2 2 3 3 4 4 5 5 6 6 7 7 8 8 9 9 10 10 11 11 12
**Output:** 1 3 5 7 9 11
**Status:** OK. Se preserva la mayor cantidad de cursos, eliminando los necesarios para "desconectar"toda la serie. En estos casos, hay que eliminar un curso de por medio.

**Performance**

# 5. Ejercicio 3

## 5.1. Interpretación del enunciado

En el piso de un museo se desean instalar sensores láser de seguridad. Cada sensor cubre una determinada cantidad de metros cuadrados del piso. Existen dos tipos de sensores, los sensores bidireccionales de $4.000 que emiten dos láseres en direcciones opuestas y los sensores cuatridireccionales de $6.000 que emiten cuatro láseres formando un ángulo recto entre cada par consecutivo de láseres, es decir, forman una cruz. Cada sensor cubre la posición (x,y) del piso sobre la que está situado además de todas las posiciones sobre las que incidan sus láseres. Los láseres bidireccionales pueden ser orientados horizontal o verticalmente. La emisión del láser solo se detiene al alcanzar una pared. Además de las paredes que delimitan el piso, existen otras paredes dentro del mismo. Los pisos son rectangulares con un ancho y alto determinados. En cada posición (x,y) del piso ($x < ancho$ y $y < alto$) puede haber una pared, un lugar vacío o un sensor. No puede haber más de un sensor en la misma posción. Tampoco se puede ubicar un sensor de forma que sea alcanzado por un láser de otro sensor.

Se quiere ubicar alguna cantidad de sensores de forma que toda posición del piso sea vigilada por al menos un sensor. Una posición se considera vigilada si es alcanzada por el láser de algún sensor. Existen además ciertas posiciones importantes, las cuales deben ser vigiladas por dos sensores distintos. Dado lo costosos que son los sensores, se pide también encontrar la forma más barata de vigilar todo el piso. El objetivo de este ejercicio es desarrollar un algoritmo que, dado un piso del museo, determine la forma menos costosa de vigilar cada posición con al menos un sensor (dos las importantes) sin superponer sensores ni que estos se vigilen entre sí. Si bien no hay restricción a la complejidad, se pide que el algoritmo desarrollado utilice la técnica de *backtracking*, y que se implementen podas para reducir el tiempo de ejecución.

## 5.2. Resolución

El piso viene representado como una matriz de enteros de dimensiones conocidas. Cada casillero de la matriz se corresponde a una posición del piso y contiene un 0 si hay una pared en dicha posición del piso, un 1 si es un espacio simple (o vacío) o un 2 si el espacio es importante. El algoritmo desarrollado realiza backtracking iterativo sobre los casilleros en los cuales se pueden ubicar los sensores, es decir los casilleros en principio vacíos. Dado que un casillero importante debe ser vigilado por dos sensores distintos, no tiene sentido ubicar sensores en tales posiciones, ya que tendría que ubicarse otro sensor en otra posición que vigile esa casilla, pero esto provocaría que un sensor vigile a otro, lo cual está prohibido. Ignorar los casilleros importantes como posibles ubicaciones para los sensores es una primera poda.

Para modelar el problema se tomarán los sensores bidireccionales como dos tipos de sensores distintos (aunque con el mismo precio) si se encuentran orientados horizontamente que si se encuentran orientados verticalmente. Entonces se definen tres nuevos posibles valores para cada casillero (uno por cada tipo de sensor). Las posibles soluciones del árbol de soluciones del algoritmo consisten en, para cada casillero en principio vacío, asignarle un valor correspondiente a alguna de las cuatro posibilidades: que siga vacío, que se ubique un sensor horizontal, que se ubique un sensor vertical o que se ubique un sensor cuatridireccional. El total de soluciones es $4^n$,con $n$ la cantidad de casilleros en principio vacíos. Esta $n$ está acotada por el tamaño de entrada (el producto entre el ancho y el alto del piso), aunque se considerará la complejidad en función de dicha $n$. La justificación para tal decisión es que, como recorrer todas las posibilidades tiene complejidad $O(4^n)$(y backtracking básicamente hace eso) un piso muy grande cubierto de paredes salvo por unos pocos casilleros será mucho más fácil de resolver que uno más pequeño pero con mayor proporción de casilleros vacíos sobre paredes.

A continuación se muestra el pseudocódigo correspondiente al backtracking iterativo sobre los casilleros en principio vacíos.

**Algoritmo 3:** Resolución basada en Backtracking Ejercicio 3

```
   Data: Piso piso
1  Piso mejor ⟵ piso
2  costo(mejor) ⟵ costoMaximo + 1
3  while loop do
4  |  if pisoValido(piso) y costo(piso) <costo(mejor) then
5  |  |  mejor ⟵ piso
6  |  Casilla c ⟵primeraCasillaVacia
7  |  overflow ⟵cambiarValor(c)
8  |  while overflow do
9  |  |  if ultimaCasilla(c) then
10 |  |  |  loop = false
11 |  |  |  break
12 |  |  else
13 |  |  |  c ⟵casillaSiguienteVacia(c)
14 |  |  |  overflow ⟵cambiarValor(c)
15 if costo(mejor) >costoMaximo then
16 |  return No hay Solucion
17 else
18 |  return mejor
```

El piso *mejor* es el mejor piso encontrado hasta ahora, es decir, la distribución de sensores más barata. El costo de este piso (la cantidad de dinero necesaria para implementarlo) se inicializa con $costoMaximo + 1$, para que la primera solución válida que encuentre, cualqueira sea su costo, lo sobreescriba. Si al finalizar no se encuentra ninguna solución válida, el costo de la mejor solución seguirá siendo mayor a *costoMaximo*, lo que se utiliza para saber si se debe retornar una solución, o si no se encontró ninguna (líneas 15 a 18). Al empezar cada iteración del ciclo principal (línea 3), *piso* contiene una de las $4^n$ soluciones. Lo primero que se hace es verificar si dicha solución es válida y mejor (más barata) que la óptima conseguida hasta el momento (líneas 4 y 5). La función *pisoValido* recorre todos los casilleros del piso y verifica lo siguiente.

- Si el casillero está vacío, verifica que al menos un sensor vigile su posición.
- Si el casillero contiene un sensor, verifica que ningún otro sensor vigile su posición.
- Si el casillero es importante, verifica que 2 sensores vigiles su posición.

La verificación de cada casillero consiste en recorrer, en el peor caso, toda la fila y toda la columna del casillero, dando una complejidad de O($w$+$h$) con $w$ el ancho y $h$ el alto del piso. Luego la complejidad de la función *pisoValido* tiene una complejidad de O($w$*$h$*($w$+$h$)). El resto del ciclo (líneas 6 a 14) cambian el valor de al menos un casillero, obteniendo otra solución para la siguiente iteración. La función *cambiarValor* toma una casilla y le cambia el valor. El valor puede ser VACIO (no contiene ningún sensor), SENSORH (contiene un sensor horizontal), SENSROV (contiene un sensor vertical) o SENSOR4 (contiene un sensor cuatridireccional) y los recorre en ese orden. Cuando se cambia el valor de SENSOR4 a VACIO orta vez, la función devuelve *true* para notificar que ya se han probado todas las combinaciones de la casilla, y ahora se debe cambiar el valor de otra casilla. Cuando *cambiarValor* devuelve *true* con la última casilla, significa que ya se han probado todas las combinaciones y el algoritmo debe terminar (líneas 9 a 11). Tanto *primeraCasillaVacia* como *casillaSiguienteVacia* recorren la lista de las casillas que en principio están vacías, es decir, incluyendo a las que en ese momento contengan sensores.

Si bien pareciera que el algoritmo funciona a fuerza bruta (probando todas las soluciones posibles), recorre el árbol de soluciones al igual que el método de backtracking recursivo. Si se considera cada posible solución como el conjunto de los casilleros vacíos donde cada uno puede

tomar uno de cuatro valores, entonces cada solución se puede representar con un número de $n$ dígitos en base 4. Si a su vez se representan en el árbol de soluciones de backtracking, se puede establecer una relación uno a uno entre cada hoja del árbol y cada número en base 4. De esta forma se justifica que el algorimto desarrollado recorre las mismas soluciones que el método de backtracking recursivo.

### Podas

Sin podas, este algoritmo recorre las $4^n$ posibilidades. Una primera poda que se implementó, además de la poda trivial de no poner sensores en los lugares importantes, es evaluar, antes de comenzar el ciclo, si algún lugar importante está posicionado de forma tal que no pueda haber 2 sensores vigilándo su posición. Como ya se mencionó, cada lugar importante debe ser vigilado por un sensor en su misma fila y otro en su misma columna. Entonces, si un lugar importante se encuntra de forma tal que no puede haber ningún sensor en su misma fila o en su misma columna, la instancia no tiene solución posible. La implementación de esta poda consiste en recorrer cada casillero importante y, de forma similar a la que se verificaba que un casillero sea valido en *pisoValiso*, recorrer su fila y su columna. Si no se encuentra un casillero vacío antes de toparse con una pared, en alguna dirección, resulta que la instancia no tiene posible solución.

Otra poda implementada consiste en determinar la validez de cada casillero en el momento en que se le asigna determinado valor para reducir la complejidad de *pisoValido*. Por ejemplo, cuando *cambiarValor* le asigna a una casilla el valor SENSORH (coloca en su posición un sensor horizontal) se asegura que este sensor no esté vigilando a otro sensor recorriendo la fila de la casilla. Si es así, intenta asignarle otro valor, en este caso SENSORV, hasta encontrar alguno válido. Esto incrementa la complejidad de *cambiarValor* ya que debe verificar que el valor que se está asignando es válido. Pero al asegurar que solo instancias válidas (sin sensores apuntándose entre sí) llegarán al principio del ciclo, *pisoValido*, solo debe evaluar que los casilleros sin sensores sean vigilados por algún sensor y que los casilleros importantes sean vigilados por 2 sensores.

Una poda similar consiste en que al entrar en la función *cambiarValor*, si la casilla es vigilada por otro sensor, no vale la pena probar ubicar sensores en ella. En estos casos se retorna *true* como notificación de que ya se han evaluado las cuatro posibilidades, aunque no sea así. Finalmente se implementó otra poda para que, si la solución parcial que se está construyendo es más costosa que la solución más barata encontrada hasta el momento, se "retroceda". Es decir, se dejen vacíos todos los casilleros hasta volver a alcanzar un costo temporal menor al de la solución más barata hasta el momento. Es correcto realizar esta poda debido a que el costo de una solución parcial (que necesita más sensores para ser una solución válida) no puede reducirse cuando se alcance una solución definitiva (cuando termine de agregar los sensores necesarios) ya que agregar sensores sólo puede incrementar el costo final.

Las podas implementadas reducen el espacio de soluciones de la misma forma en este algoritmo iterativo como en un backtracking recursivo. Por ejemplo, sea la solución $A = a_1, a_2, ... a_{i-1}, a_i, ... a_n$, con $a_j$ = VACIO $\forall$ j $\in$ (i..n]. Si $a_i$ es VACIO y se va a evaluar asignarle SENSORH (ubicar un sensor horizontal en su casilla), primero se determina mediante la segunda poda enunciada, si tiene sentido hacerlo. Si no es así (si un sensor horizontal en esa posición vigilaría a otro sensor en la misma fila) se deben podar todas las soluciones con el prefijo $A_{1..i} = a_1, a_2, ... a_{i-1}$, SENSORH. La técnica de backtracking recursivo realizaría esta poda al recorrer el nodo en el nivel $i$, antecesor de la hoja correspondiente a la solución A. Dicho nodo es antecesor de todas las hojas cuyas soluciones asociadas tienen el prefijo $A_{1..i}$, entonces al dejar de recorrer esa ramificación, se evitan evaluar todas esas soluciones. El algoritmo desarrollado con backtracking iterativo realiza la misma poda cuando itera sobre la solución A. Al comprobar que no tiene sentido asignarle SENSORH a $a_i$, le asigna en su lugar SENSORV, ignorando así todas las soluciones con el prefijo $A_{1..i}$.

## 5.3. Complejidad

Aún aplicando las podas, la complejidad del algoritmo sigue siendo O($4^n$) ya que en el peor caso, ninguna de las cotas reduciría el conjunto de soluciones y todas ellas deberían evaluarse. Sin

embargo, esperamos que en la mayoría de los casos se vea reducido el tiempo total de ejecución gracias a las podas (es muy común que en un piso haya 2 casilleros vacíos adyacentes, caso en el que no se evaluariá poner sensores cuatridireccionaoles en ambos). La complejidad de *pisoValido* sigue siendo O($w$*$h$*($w$+$h$)), ya que debe recorrer todos los casilleros y, para los vacíos y los importantes (en la primera iteración son todos menos las paredes), recorrer una fila y una columna. La función *cambiarValor* también debe recorrer una fila y una columna pero sólo para un casillero (aunque podría tener que hacerlo 4 veces), por lo que su complejidad es O(4*$w$*$h$). La complejidad final del algoritmo resulta entonces
T(n) = O( 4n * ($w$*$h$*($w$+$h$) + 4*$w$*$h$ ) = O( $4^n$ * $w$*$h$*($w$+$h$+4) ). Si acotamos $n$ por $w$*$h$ y llamamos $s$ a $w$*$h$, queda
T(s) = O( $4^s$ * ( $ws$ + $hs$ + 4$s$ ) = O( $4^s$ * ( $2s^2$ + 4$s$) ) = O( $4^s$ * $s^2$) con s la cantidad de casilleros del piso.

## 5.4. Implementación

Cada piso se representa con una matriz de enteros, donde cada entero representa el contenido de la posición en el piso. Se guardan dos pisos, el *pisoMejor* (almacena el mejor encontrado hasta el momento) y el *pisoActual* (el cual se modifica en cada iteración). La función *cambiarValor* toma en un arreglo los casilleros vacíos y suma 1 al número en base 4 que representa a la solución de *pisoActual*. Ese cambio se ve reflejado en el piso. La función del pseudocódigo *pisoValido* se reemplazó en el código por *evaluarPiso* la cual recorre el *pisoActual* y, si es mejor que el *pisoMejor*, pasa a ser el nuevo *pisoMejor*. Gracias a las podas se sabe que el *pisoActual* que recorre *evaluarPiso* es válido en el sentido que no hay sensores apuntándose entre sí, por lo que sólo se asegura que todos los casilleros estén siendo vigilados por al menos un sensor y que los casilleros importantes estén siendo vigilados por 2 sensores.

La primera poda se ejecuta antes de comenzar el ciclo principal. Para cada casillero importante, se recorren su fila y su columna en busca de alguna posición vacía. Si para alguno no se encuentran, la función principal retorna sin entrar al ciclo. Las otras podas se implementan en *cambiarValor* ya que actúan al momento de asignar valores a los casilleros. Para determinar la validez de los sensores en los casilleros se implementaron las funciones *vigila*(*Posicion* $p$) que devuelve si un sensor en la posición $p$ está vigilando a otro sensor, *vigilada*(*Posicion* $p$) que devuelve si algún sensor está vigilando la posición $p$ y *doble_vigilada*(*Posicion* $p$) que devuelve si 2 sensores están vigilando la posición $p$ (utilizada por *evaluarPiso* para determinar si todas las posiciones importantes están siendo vigiladas por 2 sensores). Estas tres funciones recorren la fila y la columna de la posición $p$ en busca de sensores.

## 5.5. Demostración de Correctitud

...

## 5.6. Testing

### Correctitud

### Performance

# 6. Conclusiones

La realización de este trabajo práctico nos permitió familiarizarnos con el enfoque teórico-experimental que la cátedra espera dentro de los trabajos prácticos. Pudimos recorrer el proceso del análisis formal de un problema, desde su observación inicial hasta las conclusiones prácticas.

Por el lado teórico y formal, nos planteó las dudas sobre que es necesario demostrar para justificar nuestras soluciones propuestas. Por el lado práctico, nos ayudó a complementar las nociones de algoritmos greedy y backtracking que ya habíamos visto en las clases teóricas de la materia.

También nos acercó ciertas nociones preliminares sobre cómo mostrar resultados de la manera más concreta posible y evitar los errores más comunes al hacerlo.

Para finalizar, sentimos que este trabajo nos representó un buen ejercicio en el análisis algorítmico general y nos dejó bien orientados para afrontar los siguientes proyectos dentro de la materia.