# Algoritmos y Estructuras de Datos III

Trabajo Práctico 1 - Técnicas algoritmicas

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

## Informe



| Integrante | LU | Correo electrónico |
|---|---|---|
| Carrasco, Andrés Nicolas | 1905/21 | nicocarrasco703@gmail.com |
| Moraut, Tobias | 1507/21 | tobiasmoraut7@gmail.com |



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

# 1. Introducción

## 1.1. Presentación informal del problema

Tenemos un conjunto de actividades $\mathcal{A}$ de tamaño $n$. Cada actividad se realiza en algún intervalo de tiempo, el cual consiste de un momento inicial y un momento final. Sabemos que siempre el momento inicial es menor estricto al momento final, y que ambos están acotados por $2n$.

Queremos encontrar el subconjunto más grande de $\mathcal{A}$ tal que ninguna actividad se solape con otra en el tiempo. Vale aclarar que una actividad puede comenzar al mismo tiempo que termina la anterior.

Debemos implementar un algoritmo que resuelva este problema, utilizando una estrategia *golosa*.

## 1.2. Enunciado

Dado un conjunto de actividades $\mathcal{A} = \{A_1, \ldots, A_n\}$, queremos encontrar un subconjunto de actividades $\mathcal{S}$ de cardinalidad máxima, tal que ningún par de actividades de $\mathcal{S}$ se solapen en tiempo.

Cada actividad $A_i$ se realiza en algún intervalo de tiempo $(s_i, t_i)$, siendo $s_i \in \mathbb{N}$ su momento inicial y $t_i \in \mathbb{N}$ su momento final. Suponemos que $1 \leq s_i < t_i \leq 2n$, $\forall\ 1 \leq i \leq n$ y que es posible que exista $A_i, A_j \in \mathcal{S}$ tal que $t_i = s_j$.

# 2. Algoritmo

## 2.1. Explicación

**Entrada:** $n$, $s_1$, $t_1$, …, $s_n$, $t_n$

**Salida:** máxima cantidad de actividades $x$, $A_1$, …, $A_x$

Inicialmente, creamos un vector $a$ de tamaño $n$. Utilizamos un ciclo para tomar los valores de entrada y crear pares $(s_i, t_i)$ que luego guardamos en $a$.

Utilizamos la función `paresATriplas`$(a)$, la cual convierte el vector de pares $a$ en un vector de triplas $(s_i, t_i, i)$ donde $i$ es el índice de la actividad en el vector $a$. Esto resulta útil para mantener la posición de las actividades en el vector original.

Luego, llamamos a la función `ordenarTriplas` que utiliza un algoritmo de *Bucket Sort*. Como sabemos que los intervalos están acotados por $2n$, creamos un vector de $2n$ buckets (listas de triplas) dentro de los cuales agregamos las triplas $(s_i, t_i, i)$, clasificándolas según su $t_i$.

Recorriendo en orden los buckets y concatenando las triplas podemos reconstruir nuestro vector con las actividades ordenadas por tiempo de finalización.

A continuación, se construye el conjunto de actividades no solapadas $\mathcal{S}$. El algoritmo comienza seleccionando la actividad que termina más temprano y la agrega al conjunto de actividades no solapadas. Luego, para cada actividad restante, si su tiempo de inicio es posterior o igual al tiempo de finalización de la última actividad en el conjunto $\mathcal{S}$, se agrega. De lo contrario, la actividad se descarta.

Finalmente, el programa imprime la cantidad de actividades en el conjunto no solapado $\mathcal{S}$ y los índices de las actividades originales en el vector de entrada.

## 2.2. Demostración de correctitud

Vamos a demostrar que el algoritmo implementado es correcto. Para ello debemos demostrar que la solución que brinda nuestro algoritmo en cada iteración es válida, y que además, es extensible a una solución óptima.

**Demostración por inducción en $i$**

Caso base:

Consideremos el caso $i = 0$. Podemos notar que en este punto todavía no se incluyo una actividad. Esto significa que al comenzar, tenemos una solución vacía, la cual es posible extender a una solución óptima dado que no tiene actividades que puedan solaparse con otras de cualquier solución óptima.

Paso inductivo:

Supongamos que la solución parcial $B_1, \ldots, B_i$ que se obtiene en la iteración $i$ es un subconjunto de actividades que no se solapan y además, su extensión óptima es $B_1, \ldots, B_i, C_{i+1}, \ldots, C_j$. Llamemos a esta extensión $B*$ ($B_1, \ldots, B_i \subseteq B*$).

En la iteración $i+1$ agregamos $B_{i+1}$ de manera que ahora tenemos la solución $B_1, \ldots, B_i, B_{i+1}$.

Queremos ver que $B_1, \ldots, B_{i+1}$ es extensible a una solución óptima. Para ello debemos demostrar que $B_1, \ldots, B_{i+1}$ sigue incluido en $B*$. Consideramos dos casos:

1. $B_{i+1} = C_{i+1}$: por hipótesis inductiva $B_1, \ldots, B_i, B_{i+1}, \ldots, C_j$ es la extensión óptima de $B_{i+1}$.
2. $B_{i+1} \neq C_{i+1}$: en este caso podemos notar que $B_{i+1}$ es la actividad con el menor tiempo final que no se solapa con $B_i$. Reemplazando $C_{i+1}$ por $B_{i+1}$, este seguirá sin solaparse con $B_i$, ya que de lo contrario el algoritmo no habría elegido a $B_{i+1}$ en primer lugar. Además tampoco se solapa con $C_{i+2}$, puesto que el tiempo final de $B_{i+1}$ es menor o igual al tiempo final de $C_{i+1}$, y, por transitividad, $B_{i+1}$ no se solapa con $C_{i+2}$.

De esta manera, hemos probado que nuestro algoritmo goloso es correcto.

# 3. Complejidad del algoritmo

## 3.1. Justificación

El algoritmo cumple con una complejidad lineal de $O(n)$, siendo $n$ la cantidad de actividades que ingresan por parámetro. El algoritmo tiene esta complejidad ya que ninguna sección del programa la excede.

Primero, podemos ver que al convertir las tuplas que representan la actividad en triplas, la función recorre el vector de tuplas una vez, garantizando una complejidad temporal lineal. Luego, ordenamos el vector utilizando *bucket sort*. Este recorre una vez al vector y agrega a las triplas en su respectivo bucket, a un costo de $O(n)$.

Posteriormente, recorremos el vector donde almacenamos los buckets, la cual tiene $2n+1$ elementos. Nuevamente, es $O(n)$, ya que en total tiene almacenadas $n$ triplas, por lo que estaríamos recorriendo $O((2n+1)+n)$ lugares, es decir, $n$.

Finalmente, en la función principal del programa, realizamos una nueva búsqueda sobre el vector de triplas ya ordenado, para incluir las triplas válidas en el vector que luego retornaremos como solución, quedando finalmente una complejidad de $O(n)$.

## 3.2. Test de performance

Para mostrar empíricamente que la complejidad del algoritmo es $O(n)$, generamos una serie de valores de entrada aleatorios que respetan la precondición del enunciado. Esto es, para toda actividad $B_i$ con $1 \leq i \leq n$, $s_i < t_i \leq 2n$.

Luego de correr todos los valores de entrada, llegamos a los siguientes resultados:

| **Tamaño de entrada** | **Tiempo** |
|---|---|
| 65536 | 0.0666124 |
| 131072 | 0.122436 |
| 262144 | 0.246224 |
| 524288 | 0.505848 |
| 1048576 | 1.03825 |
| 2097152 | 2.13012 |
| 4194304 | 4.43141 |
| 8388608 | 9.42437 |
| 16777216 | 19.5847 |
| 33554432 | 40.6773 |

Como podemos observar, el tiempo de ejecución del algoritmo depende del tamaño de la entrada.

Posteriormente, graficamos los resultados en un eje de coordenadas. Esto lo hicimos en escala logarítmica, ya que los valores de las entradas crecen exponencialmente.

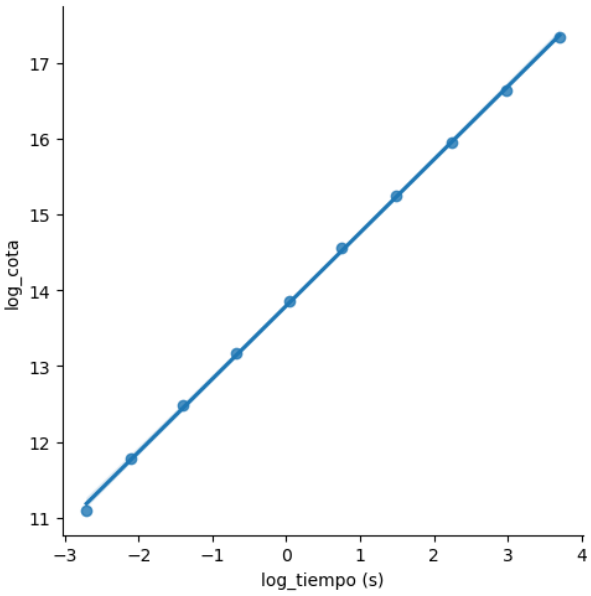


Figura 1: Gráfico a escala logarítmica

Podemos notar que el crecimiento es lineal.

### 3.3. Comparativa entre conjuntos de instancias

Para terminar, y respondiendo a la pregunta de si existen conjuntos de instancias que sean más fáciles o más difíciles de resolver para nuestro algoritmo, realizamos una comparativa entre los valores de entrada previamente utilizados (aleatorios) contra un conjunto de valores de entrada que están ordenados de manera creciente.

Obtuvimos los siguientes resultados:

| **Tamaño de entrada** | **Tiempo** |
|---|---|
| 65536 | 0.0510711 |
| 131072 | 0.0957448 |
| 262144 | 0.184489 |
| 524288 | 0.409753 |
| 1048576 | 0.763463 |
| 2097152 | 1.49925 |
| 4194304 | 3.07001 |
| 8388608 | 6.02821 |
| 16777216 | 12.0863 |
| 33554432 | 24.2218 |

Realizando un gráfico comparativo entre ambos resultados, podemos ver claramente que el conjunto de instancias ordenadas se resuelve más rápido.

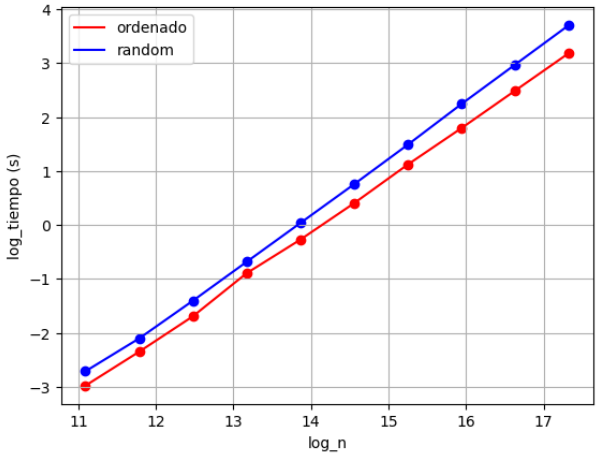


Figura 2: Gráfico comparativo a escala logarítmica

Notamos que existe un $c > 0$ tal que $t_{\text{random}} * c = t_{\text{ordenado}}$. Calculando la diferencia entre las rectas, podemos estimar que $c \approx 1{,}4$.

Esta leve mejoría de eficiencia para este tipo de instancias se debe a que el algoritmo ya no tiene que ordenar las actividades.

Para concluir, no hemos encontrado un conjunto de instancias que sea más difícil de resolver por nuestro algoritmo. Pero si encontramos que para un conjunto de instancias ordenado, el algoritmo es un poco más eficiente.