Teoría de Algoritmos I

75.29 / 95.06

Trabajo Práctico 1

Amigo, Nicolás	105832
Nicolini, Franco	105632
Serrano, Maria Candela	105287
Singer, Joaquín	105854
Primerano Lomba, Franco Alejandro	106004

Profesores:
Víctor Podberezski
Lucas Ludueño
Kevin Untrojb
Jose Sbruzzi
Ernesto Alvarez

10 de Mayo de 2021

Índice general

Parte 1: Tiempo compartido

1.1. Ejecución

Al estar programado en python no se realiza una compilación del programa, simplemente se puede ejecutar mediante la linea: `python3 main.py contrato.py <"archivo"` donde "archivo" debe ser reemplazado por el nombre del archivo a utilizar.

1.2. Análisis sobre la solución actual

La solución que se utiliza actualmente consiste en tomar primero la oferta con menor duración, rechazar las incompatibles con esta, y repetir hasta haber aceptado o rechazado todas. La solución no es óptima porque usando ese criterio de elección, al seleccionar la que dure menos tiempo y desechando aquellas que son incompatibles, se podrían encontrar casos en los que hay combinaciones de ofertas que permiten aceptar más de estas y que desechamos por ser incompatibles con la primera selección. Un ejemplo de esto se muestra en la figura 1.1.

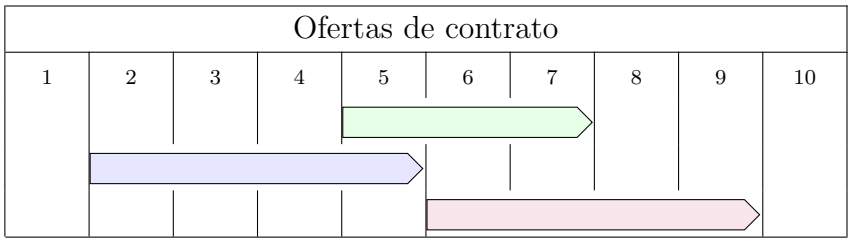

Figura 1.1: Contraejemplo a solución eligiendo el que dura menos tiempo

En este caso, el algoritmo se quedaría solo con la oferta verde, rechazando las otras dos, que son compatibles entre sí.

La solución actual está dada por una elección greedy que a primera vista puede parecer la forma correcta de llegar a una solución óptima global, es más, probablemente es la primera o una de las primeras en las que un humano piense intuitivamente. Esto es así porque al estar buscando maximizar la cantidad de ofertas aceptadas, la idea de seleccionar las de menor duración, dejando el servidor libre la mayor cantidad de tiempo posible, parece el camino correcto. El problema con esta solución queda bastante claro en el contraejemplo brindando, y es que un contrato, por más corto que sea, puede solaparse con otros que sean compatibles entre sí, evidenciando que lo importante no es elegir contratos de corta duración, sino contratos que efectivamente liberen el servidor lo antes posible, para dar lugar a otros.

1.3. Problemas de optimización y algoritmos greedy

Un problema de optimización es aquel en el que dado un conjunto de soluciones posibles, se busca encontrar la mejor de estas bajo un criterio dado, en muchos casos esto consiste en maximizar o minimizar una variable.

Existen diversas formas de abordar la resolución de estos problemas, una de ellas es con un algoritmo de tipo *greedy*. La idea básica detrás de estos algoritmos es subdividir el problema en subproblemas con una jerarquía entre ellos y resolver estos subproblemas de forma miope, sin considerar la instancia general sino el caso particular del subproblema actual. Es decir, tomar una decisión local óptima esperando que esa decisión lleve a una solución global óptima. Para tomar estas decisiones óptimas de utiliza una heurística.

Que un problema pueda ser resuelto de forma óptima utilizando un algoritmo de este tipo, suele revelar algo interesante y útil sobre la estructura del mismo. Para que esto ocurra debe existir una elección greedy y una subestructura óptima, donde esta elección local vaya llevando efectivamente a una solución óptima global.

1.4. Nuestra solución óptima

La solución greedy que proponemos se basa en la analizada en clase y en la bibliografía, para resolver el *Interval Scheduling Problem*. Esta solución toma como elección greedy el intervalo que finalice primero, liberando el recurso lo antes posible.

El problema actual es muy similar al *Interval Scheduling Problem*, la única diferencia que presenta es que el periodo de tiempo en el que transcurre no tiene un principio ni un final, por lo que no se puede aplicar el algoritmo que ya conocemos para obtener una solución óptima, como se puede observar en la figura 1.2.

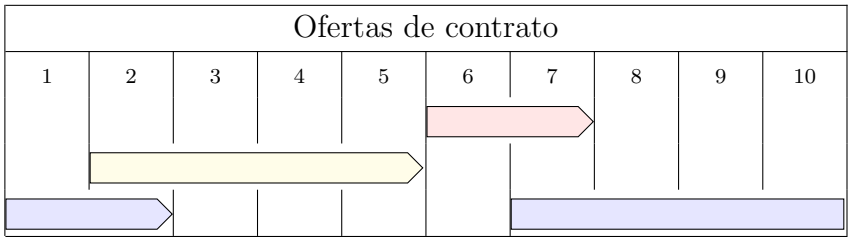

Figura 1.2: Contraejemplo cíclico

En este caso cíclico, eligiendo primero el intervalo que finaliza antes, se descartan los otros dos, que son compatibles entre sí.

Entonces, lo que proponemos es romper estos ciclos para poder aplicar la solución óptima ya conocida. La idea consiste en tomar un intervalo cualquiera (tomamos por facilitar la implementación el que termina primero, pero podría haber sido cualquiera), y resolver el *interval scheduling problem* comenzando por ese intervalo. Luego, resolver el problema para todos los intervalos que se

superpongan con el elegido, y así abordar las opciones que habíamos descartado. Al finalizar la resolución de cada instancia, se entrega como solución final la que más intervalos haya aceptado.

En la solución propuesta, se resuelven una o más instancias del *interval scheduling problem*, y para esto se aplica un algoritmo greedy. Primero se debe observar que se trata de un problema de optimización, donde se quiere maximizar una variable, que es la cantidad de contratos aceptados. Para lograr esto, se toma una elección greedy que consiste en ir eligiendo los contratos que finalicen primero y no se superpongan con uno previamente elegido. Esta elección es categorizada como greedy ya que nunca tiene en cuenta la instancia global del problema, no son relevantes para la elección los contratos que finalizan después. La división en subproblemas viene de ir considerando cada iteración una instancia más avanzada en el periodo de tiempo donde ocurre el problema, hasta llegar al final del mismo habiendo elegido siempre qué oferta aceptar con la elección greedy.

1.5. Análisis temporal y espacial del problema

La complejidad temporal de nuestro problema sera $O(n^2)$ ya que, luego de ordenar por finalización de manera creciente, tendrá que recorrer la lista de elementos desde aquellos que se superpongan con el que finaliza primero. A todos ellos le aplicará la solución conocida del *interval scheduling problem*. En el peor de los casos, el primer elemento seleccionado se superpondrá con todos y por ello aplicara N veces dicha solución, cuya complejidad es $O(n)$ si los elementos ya están previamente ordenados[1]. Por ultimo, y en forma de aclaración, se utiliza las operaciones append y len sobre una lista. Las mismas tienen complejidad $O(1)$[2] por lo que no afecta a la complejidad final.

Para analizar la complejidad espacial involucrada tendremos en cuenta que el crecimiento de ofertas horarias implicará un crecimiento lineal tanto en la lista inicial, como en la lista con mayores ofertas encontrada hasta determinado momento, como en la lista que utiliza y devuelve el interval scheduling problem lineal. Esto es un total de $O(3n)$ y por lo tanto la complejidad espacial utilizada es $O(n)$.

1.6. Análisis temporal y espacial de nuestra solución

En la función `ordenar_contratos` contamos con la función sort que tiene una complejidad $O(nlogn)$[3], la siguiente función llamada `hallar_maximo_subconjunto_de_contratos` tiene una complejidad $O(n)$ ya que realiza un ciclo de rango igual al tamaño de contratos superpuestos, que en el peor de los casos es N. Dentro de esta ultima función está `interval_scheduling_problem` que también cuenta con una complejidad $O(n)$, lo que hará que nuestro problema tenga una complejidad temporal de $O(n^2)$. En comparación con $O(nlogn)$, $O(n^2)$ es mayor, por lo tanto coincide con lo obtenido en el calculo de la complejidad temporal del problema.

1.7. Demostración de que la solución es óptima

Para la demostración de que la solución es óptima nos basamos en el hecho de que el algoritmo de la solución al *interval scheduling problem* ya devuelve una solución óptima Nuestro algoritmo primero convierte un problema de contratos circular a uno lineal que resolvemos utilizando la solución

conocida. Esta conversión implica elegir un elemento y asumir que ya pertenece al conjunto de contratos óptimos y "cortar" la semana de acuerdo a este elemento.

Ejemplo:

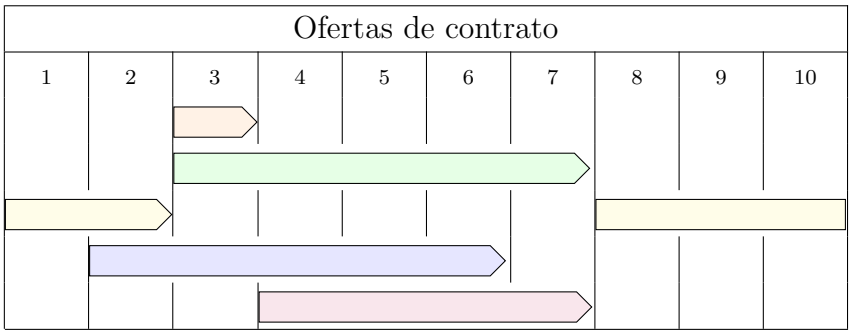

Figura 1.3: Ejemplo de semana

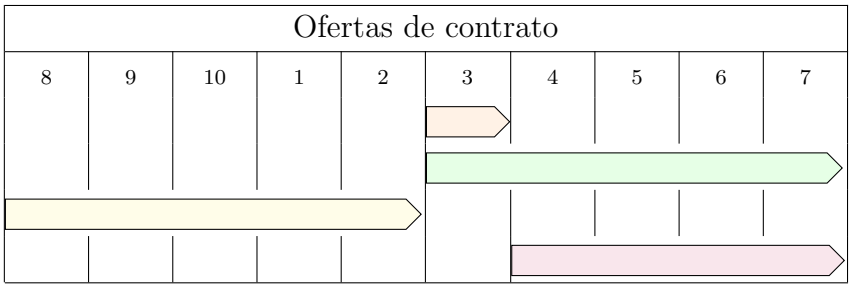

Figura 1.4: Conversión basada en el contrato amarillo

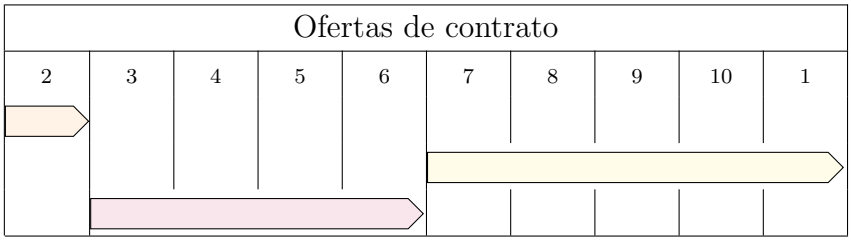

Figura 1.5: Conversión basada en el contrato naranja

Con esto podemos ver que si hacemos la conversión con todos los contratos disponibles, guardándonos la cantidad de la solución óptima de cada uno, llegaremos a la solución óptima para el problema con el intervalo de tiempo circular en tiempo cuadrático. Sin embargo nosotros implementamos adicionalmente una optimización de algoritmo que reduce la cantidad de contratos que hay que probar. Utilizando el siguiente **teorema**:

Sea C un contrato, I(C) un conjunto con los contratos incompatibles a C y ϕ una solución óptima. Entonces

$$\exists x \in \{C\} \cup I(C) / x \in \phi \tag{1.1}$$

Demostración:
Asumo que $\forall x \in \{C\} \cup I(C)/x \notin \phi$. Entonces $\{C\} \cup \phi$ es compatible y es una mejor solución que ϕ. ABSURDO ya que asumimos que ϕ era solución óptima.

Este teorema nos dice que no es necesario hacer la conversión con cada contrato sino que basta con hacerlo con un elemento cualquiera y sus contratos incompatibles.

Este algoritmo sigue siendo $O(n^2)$ ya que en el peor caso pueden ser todos los contratos incompatibles entre si pero reducirá el tiempo del caso promedio donde la cantidad de contratos que son incompatibles entre si sera mucho menor a la cantidad de contratos.

1.8. Pseudocódigo

```
lista_de_contratos es la lista de todas las ofertas

hallar_maximo_subconjunto_de_contratos:
  mejor_encontrada es una lista de contratos vacía
  por cada contrato en la lista:
    si el contrato no es comptabile con el primero de la lista:
      lista_nueva = resolución del interval scheduling problem de esta instancia
      si lista_nueva tiene más contratos que mejor_encontrada:
        mejor_encontrada = lista_nueva
  devolver mejor_encontrada
```

1.9. Pruebas del programa

Para probar su funcionalidad elegimos los archivos “primero.txt” y “segundo.txt”.

En el primero, de 20 personas, el caso óptimo incluye a un contrato que empieza el domingo a las 19hs y finaliza el lunes a las 8hs. Dando un ejemplo de una solución que incluye a un contrato que atraviesa la semana. En el segundo caso, de 56 personas, el caso óptimo no posee contratos que empiecen en la semana y finalicen pasada la medianoche del domingo.

Parte 2: Un rompecabezas cuadrado

2.1. Presentación del algoritmo

El algoritmo en pseudocódigo es:

```
patio es una matriz
n es el tamaño del lado de la matriz patio
posición_ocupada es la posición de la rejilla
llenar patio(patio, n, posición_ocupada)
    if n == 2:
        patio llenar posiciones no ocupadas

    #Divido el patio en 4 cuadrantes
    a1 = top izquierdo patio
    a2 = top derecho patio
    a3 = bottom izquierdo patio
    a4 = bottom derecho patio

    pos_ocupada_n = posición_ocupada si pertenece a a_n

    for a_n if posición_ocupada no pertenece a a_n
        if a_n == a1
            patio llenar posición (n/2 - 1, n/2 -1)
            pos_ocupada1 = (n/2 - 1, n/2 -1)
        if a_n == a2
            patio llenar posición (n/2 - 1, n/2)
            pos_ocupada2 = (n/2 - 1, n/2)
        if a_n == a3
            patio llenar posición (n/2, n/2 - 1)
            pos_ocupada3 = (n/2, n/2 - 1)
        if a_n == a4
            patio llenar posición (n/2, n/2)
            pos_ocupada4 = (n/2, n/2)

    llenar patio(a1, n/2, pos_ocupada1)
    llenar patio(a2, n/2, pos_ocupada2)
    llenar patio(a3, n/2, pos_ocupada3)
    llenar patio(a4, n/2, pos_ocupada4)
```

2.2. Ejemplo del algoritmo en uso

Se comienza con un patio vacío, que solo contiene el sumidero.

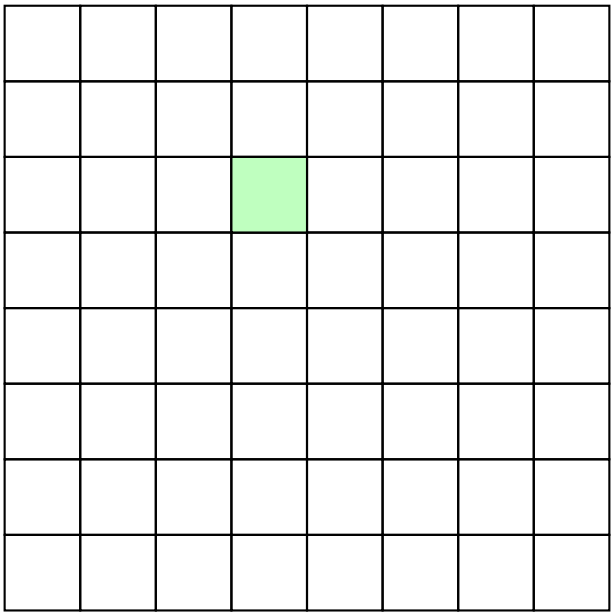

Figura 2.1: Patio antes de comenzar iterar

Luego, se separa el patio en 4 cuadrantes y se coloca una baldosa en el centro, dejando como cuadro que no cubre la L al del cuadrante que contenga el sumidero.

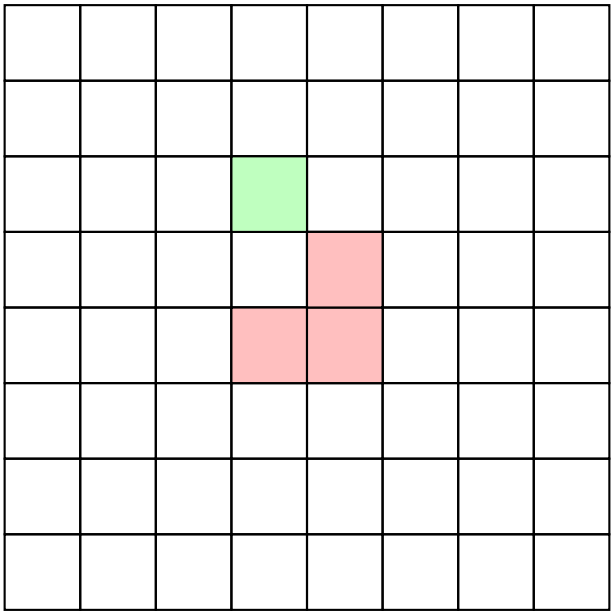

Figura 2.2: Patio después de la primera llamada recursiva

Se repite el paso anterior con cada cuadrante, tomando como el sumidero de cada cuadrante el espacio que fue ocupado en el paso anterior (o el sumidero original para el cuadrante al que pertenece),

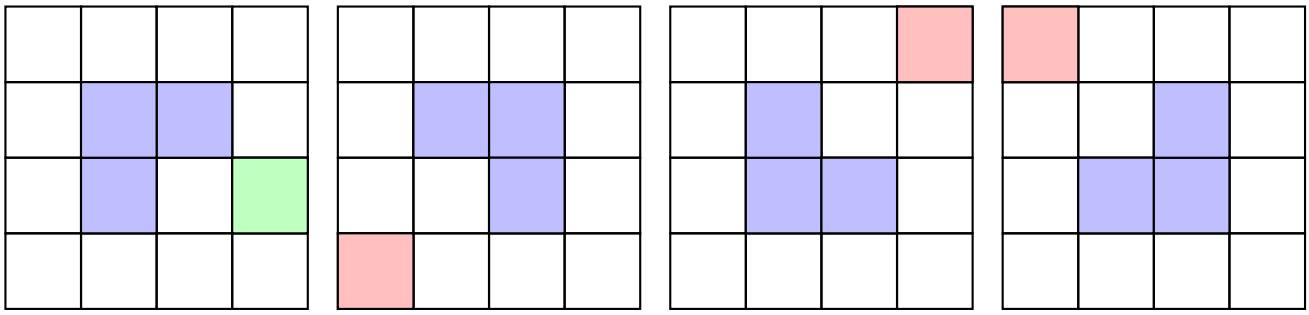

Figura 2.3: Separación en cuadrantes

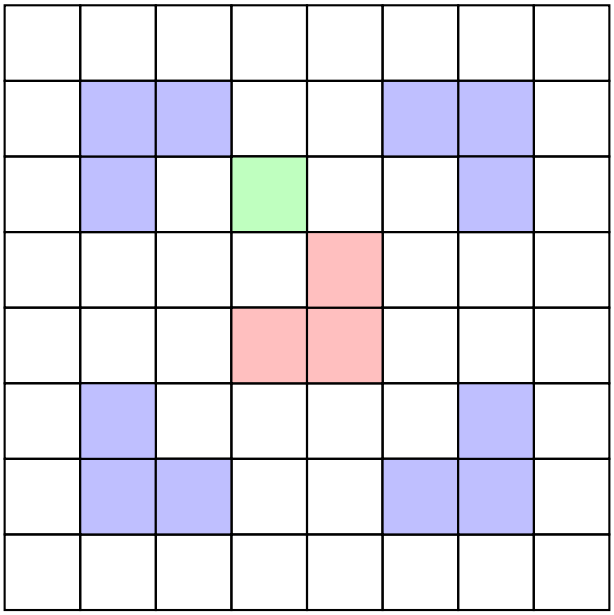

Figura 2.4: Patio después de la segunda llamada recursiva

Luego, se repite lo mismo, hasta que los cuadrantes sean un cuadrado de 2×2 (en el ejemplo se muestra la separación del cuadrante superior izquierdo).

Figura 2.5: Separación en cuadrantes

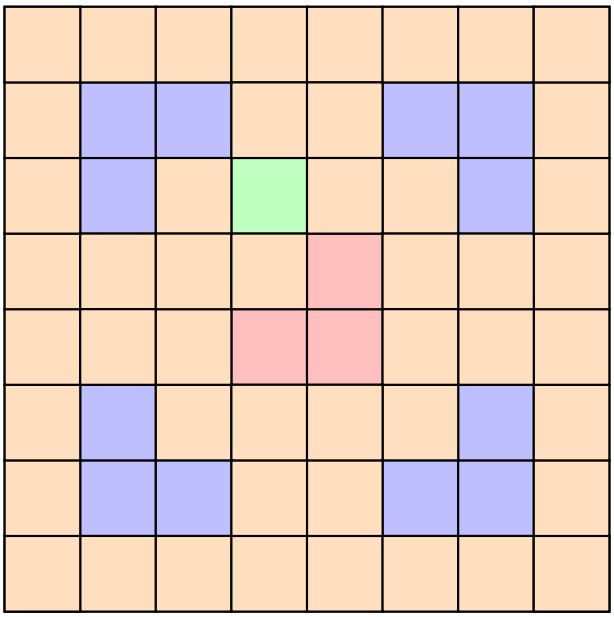

Figura 2.6: Patio después de la tercera llamada recursiva

2.3. Explicación detallada del algoritmo

La solución propuesta consiste en dividir el problema original en cuatro subproblemas similares y resolverlos recursivamente de la misma forma hasta llegar al caso base. Si se nos da un patio con lado n cada subproblema revolverá los cuarto cuadrados de lado $\frac{n}{2}$. Para que cada uno de los subproblemas sean iguales al anterior cada uno de los cuadrantes deberá tener un espacio que ya se encuentra ocupado, como lo es la rejilla en la primera iteración. Para lograr esto se coloca una baldosa de manera que que este contenida entre los tres cuadrantes en los que no se encuentra el espacio ya ocupado. De esta manera los cuatro subproblemas serán idénticos en cuanto su tamaño y la existencia de una posición ya ocupada. Como caso base estableceremos el caso en el lado del cuadrado sea dos y bastara con solo poner una baldosa (habrá solo una manera de poner la baldosa en este caso ya que una de las posiciones ya estará ocupada).

2.4. Relación de recurrencia

Si establecemos la relación de recurrencia del algoritmo llegamos a la expresión:

$$T(n) = 4T(\frac{n}{2}) + O(1)$$

Ya que todas las operaciones menos el llenar patio pueden realizarse en tiempo constante. Y el problema es dividido en 4 subproblemas cuyo tamaño es el original(n^2) dividido en 4, es decir, por ser n una potencia de 2:

$$(\frac{n}{2})$$

2.5. Análisis temporal y espacial

Para realizar el análisis de la complejidad temporal utilizamos el teorema maestro[4], que enuncia lo siguiente:

Dada una ecuación de recurrencia de la forma $T(n) = aT(\frac{n}{b}) + f(n)$:

$$T(n) = \begin{cases} \Theta(n^{\log_b a}) & \text{si } f(n) = O(n^c) \wedge c < \log_b a \\ \Theta(n^{\log_b a} \log^{k+1} n) & \text{si } f(n) = \Theta(n^c \log^k n) \wedge c = \log_b a, \text{ con } k \geq 0 \\ \Theta(f(n)) & \text{si } f(n) = \Omega(n^c) \wedge c > \log_b a \end{cases}$$

en este caso como nuestra ecuación de recurrencia es:

$$T(n) = 4T(\frac{n}{2}) + O(1)$$

utilizamos el primer caso del teorema maestro donde resulta $T(n) = \Theta(n^2)$.

Y para la complejidad espacial espacial:

$$M(n) = O(n^2)$$

ya que se deberá guardar la matriz que representa al patio en memoria.

Bibliografía

[1] Complejidad de la solución al Interval Scheduling Problem conocida

J. Kleinberg, E. Tardos, Algorithm Design, Addison Wesley (2006).

Capítulo 4.1, **Analizando el algoritmo**

[2] Python TimeComplexity
`https://wiki.python.org/moin/TimeComplexity`

[3] On the Worst-Case Complexity of TimSort
`https://drops.dagstuhl.de/opus/volltexte/2018/9467/`

[4] T. Cormen, C. Leiserson, R. Rivest, C. Stein, Introduction to Algorithms (tercera edición), MIT Press (2009).

Capítulo 4.5, **The master method for solving recurrences**