ALGORITMIA

Grado en Ingeniería Informática del Software - Curso 2022-2023

GUIÓN DE LA PRÁCTICA 3

OBJETIVO:

- **Divide y Vencerás: modelos recursivos y ejemplos**

Modelos recursivos básicos

El paquete **alg77777777.p3** contiene diez clases siguientes:

Las clases **Sustraccion1.java** y **Sustraccion2.java** tienen un esquema por *SUSTRACCIÓN* con a=1, lo que lleva consigo un **gran gasto de pila**, de hecho se desborda cuando el tamaño del problema crece hasta algunos miles. Afortunadamente los problemas así solucionados, van a tener una mejor solución iterativa (con bucles) que la solución de este tipo (sustracción con a=1).

La clase **Sustraccion3.java** tiene un esquema por *SUSTRACCIÓN* con a>1, lo que lleva consigo un **gran tiempo de ejecución** (exponencial o no polinómico). Esto supone que para un tamaño del problema de algunas decenas el algoritmo no acaba (*tiempo intratable, NP*). La consecuencia es que debemos procurar no plantear soluciones del tipo *SUSTRACCIÓN* con varias llamadas (a>1)

Las clases **Division1.java**, **Division2.java** y **Division3.java** tienen un esquema por *DIVISIÓN*, siendo la primera del tipo $a<b^k$, la segunda del tipo $a=b^k$ y la última del tipo $a>b^k$ (ver cómo se definen estas constantes en teoría).

SumaVector1.java resuelve de tres formas diferentes el sencillo problema de sumar los elementos de un vector y **SumaVector2.java** mide tiempos de esos tres algoritmos para diferentes tamaños del problema.

Fibonacci1.java resuelve de cuatro formas diferentes el problema de calcular el número de Fibonacci de orden n y **Fibonacci2.java** mide tiempos de esos cuatro algoritmos para diferentes tamaños del problema.

Escribir las siguientes clases recursivas para obtener las siguientes complejidades de ejecución:

- **Sustraccion4.java**, método recursivo POR SUSTRACCION con una complejidad $O(3^{n/2})$
- **Division4.java**, método recursivo POR DIVISIÓN con una complejidad $O(n^2)$ y el número de subproblemas = 4.

TRABAJO PEDIDO

Realizar un análisis de las complejidades y empírico de las 10 clases anteriores. Estudiando el código y ejecutándolo. Escribir el código para Sustracción4.java y Division4.java

Las clases que programe las incluirá dentro del paquete alg<dnipropio>.p31 , si utiliza Eclipse llamar al proyecto prac03_Recursivo<UOpropio>
Se entregará en el campus virtual según el calendario previsto.

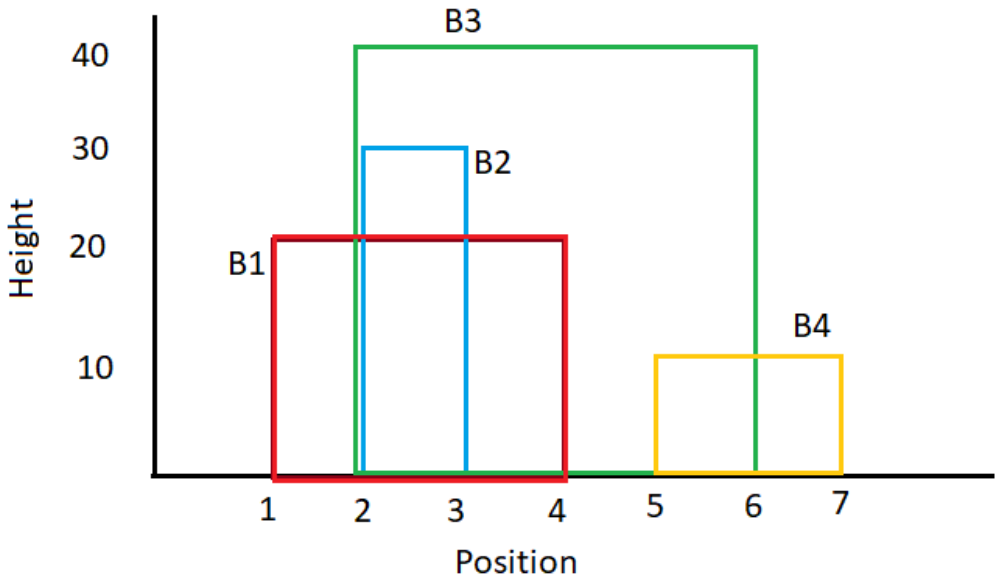
Problema Divide y Vencerás

El skyline de la ciudad

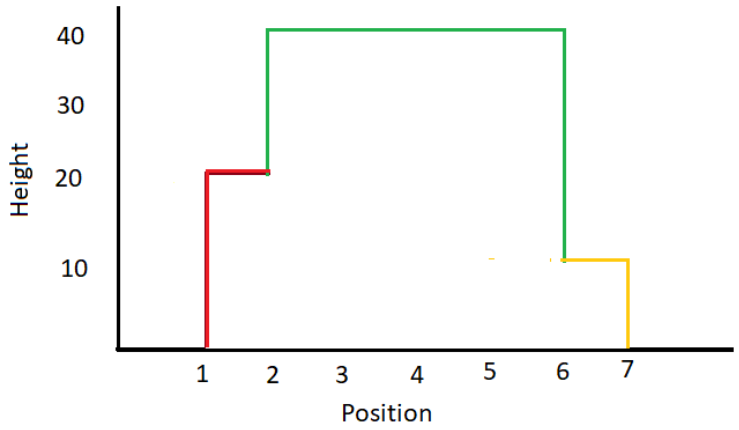
Imaginemos que nuestro cliente quiere realizar diseños con el skyline de cualquier ciudad en base a las dimensiones de sus edificios más representativos. Para ello, nos ofrecen ficheros de texto con las dimensiones de los edificios más representativos de la ciudad. Por ejemplo, el fichero de entrada **city.txt** podría tener los siguientes valores con 4 edificios representativos de Oviedo:

```
1 4 20

5 7 10

2 6 40

2 3 30
```

Cada fila representa un edificio (sin que sigan un orden en particular), siendo el primer valor la posición izquierda del edificio en el skyline, el segundo valor la posición derecha del edificio en el skyline y el tercer valor la altura del edificio. Gráficamente los cuatro edificios del fichero anterior se podrían ver como siguen:



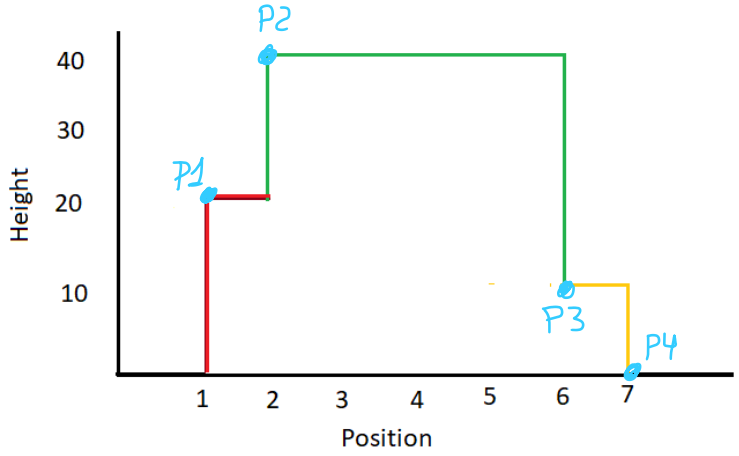
El skyline básicamente hace referencia al contorno que forman los edificios, lo que en el caso anterior sería algo como lo que sigue, formado por la altura máxima de los edificios en cada punto:



Pero nuestro cliente es un poco exigente y no quiere complicarse con todas las dimensiones del contorno. Lo que quiere es lo que llamaremos **puntos clave**. Los puntos clave son el conjunto mínimo de puntos necesarios para ser capaces de representar el contorno del skyline. En la siguiente imagen pueden verse exactamente 4 puntos clave:

- P1 en la coordenada (1, 20).
- P2 en la coordenada (2, 40).
- P3 en la coordenada (6, 10).
- P4 en la coordenada (7, 0).

Básicamente hay que pensar que un punto clave se puede obtener moviéndonos hacía la derecha hasta que se detecta un cambio de altura, altura que será parte de la coordenada del nuevo punto clave. Por lo que al haber 4 cambios de altura, tenemos 4 puntos clave.



Es por todo lo anterior que la salida del programa deberá ser el conjunto de puntos expuesto:

```
x: 1 y: 20

x: 2 y: 40

x: 6 y: 10

x: 7 y: 0
```

Para facilitar el trabajo se ofrece lo siguiente:

- 5 conjuntos de prueba con información de edificios representativos de diferentes ciudades: **city.txt**, **city2.txt**, **city3.txt**, **city4.txt** y **city5.txt**.
- Una clase **Tests.java** con pruebas unitarias basadas en los 5 conjuntos de prueba. Las pruebas deberán ser pasadas sin realizar ninguna modificación en el fichero **Tests.java**.
- Una clase **Times.java** para obtener tiempos realizando dos soluciones diferentes del problema:
  - **solveBruteForce()**: aproximación "sencilla" en la que obtendremos una complejidad polinómica.
  - **solveDivideAndConquer()**: aproximación un poco más elaborada en la que obtendremos una complejidad mejor que la anterior.
- Una clase **Building.java**, que guarda información sobre los edificios y que permite ordenar ascendentemente, en caso de necesidad, los edificios en base al lado derecho del mismo.
- Una clase **KeyPoint.java**, que guarda información sobre los puntos clave que se piden en el problema.
- Una clase **Skyline.java**, que permite crear el skyline (el contorno) mediante dos constructores diferentes:
  - El primero de ellos permite crear el skyline para un único edificio.
  - El segundo de ellos permite generar un skyline mediante la unión de otros dos skylines.
- Un clase **SkylineProblem.java** con los métodos necesarios para implementar la solución final.

Visto lo anterior, se pide:

1. Implementar todo el código en la clase **SkylineProblem.java** y los dos métodos faltantes de la clase **Skyline.java** que permita que la aplicación funcione como se espera.

2. Probar que todo funciona correctamente con la clase **Tests.java**.

3. Medir tiempos con la clase **Times.java** y comparar los resultados de ambas implementaciones rellenando la siguiente tabla:

n	*t Fuerza Bruta*	*T Divide y vencerás*
10	…..	
20	…..	
40	…..	
80	…..	
160	…..	
…	…..	
Oveflow heap		

4. Analizar la complejidad temporal teórica de los algoritmos y comprobar si los tiempos obtenidos en el apartado anterior cumplen o no con esas complejidades.

Trabajo optativo

Como parte extra puede extender el problema paralelizando el algoritmo presentado, en línea a lo visto al respecto en clase de teoría. Como conclusión mida tiempos, para así ver qué mejora temporal se consigue e indique la complejidad temporal del nuevo algoritmo implementado.

Las clases que programe las incluirá dentro del paquete alg<dnipropio>.p32 Si utiliza Eclipse llamar al proyecto prac03_Skyline<UOpropio>
La respuesta a las preguntas planteadas en el enunciado y las tablas con los tiempos se incluirán en un documento aparte.
En el campus virtual se entregarán dos ficheros: ZIP con el código y PDF documento de tiempos, según el calendario previsto.