Trabajo práctico 2

Fecha de entrega: viernes 3 de octubre, hasta las 18:00 hs.

Este trabajo práctico consta de varios problemas y para aprobar el trabajo se requiere aprobar todos los problemas. La nota final del trabajo será un promedio ponderado de las notas finales de los ejercicios y el trabajo práctico se aprobará con una nota de 5 (*cinco*) o superior. De ser necesario (o si el grupo lo desea) el trabajo podrá reentregarse una vez corregido por los docentes y en ese caso la reentrega deberá estar acompañada por un *informe de modificaciones*. Este informe deberá detallar brevemente las diferencias entre las dos entregas, especificando los cambios, agregados y/o partes eliminadas del trabajo. Cualquier cambio que no se encuentre en dicho informe podrá no ser tenido en cuenta en la corrección de la reentrega. Para la reentrega del trabajo **podrían pedirse ejercicios adicionales**. En caso de reentregar, la nota final del trabajo será el 20 % del puntaje otorgado en la primera corrección más el 80 % del puntaje obtenido al recuperar.

Para cada ejercicio se pide encontrar una solución algorítmica al problema propuesto y desarrollar los siguientes puntos:

1. Describir detalladamente el problema a resolver dando ejemplos del mismo y sus soluciones.
2. Explicar de forma clara, sencilla, estructurada y concisa, las ideas desarrolladas para la resolución del problema. Para esto se pide utilizar pseudocódigo y lenguaje coloquial combinando adecuadamente ambas herramientas. Es importante que lo expuesto en este punto sea suficiente para el desarrollo de los puntos subsiguientes, pero no excesivo (**¡no es un código fuente!**).
3. Justificar por qué el procedimiento desarrollado en el punto anterior resuelve efectivamente el problema y demostrar formalmente su correctitud.
4. Deducir una cota de complejidad temporal del algoritmo propuesto (en función de los parámetros que se consideren correctos) y justificar por qué el algoritmo desarrollado para la resolución del problema cumple la cota dada. Utilizar el modelo uniforme salvo que se explicite lo contrario.
5. Dar un código fuente claro que implemente la solución propuesta. El mismo no sólo debe ser correcto sino que además debe seguir las *buenas prácticas de la programación* (comentarios pertinentes, nombres de variables apropiados, estilo de indentación coherente, modularización adecuada, etc.). Se deben incluir las partes relevantes del código como apéndice del informe impreso entregado.
6. Realizar una experimentación computacional para medir la performance del programa implementado. Para ello se debe preparar un conjunto de casos de test que permitan observar los tiempos de ejecución en función de los parámetros de entrada. Deberán desarrollarse tanto experimentos con instancias aleatorias (detallando cómo fueron generadas) como experimentos con instancias particulares (de peor caso en tiempo de ejecución, por ejemplo). Se debe presentar **adecuadamente** en forma gráfica una comparación entre los tiempos medidos y la complejidad teórica calculada y extraer conclusiones de la experimentación.

Respecto de las implementaciones, se acepta cualquier lenguaje que permita el cálculo de complejidades según la forma vista en la materia. Además, debe poder compilarse y ejecutarse correctamente en las máquinas de los laboratorios del Departamento de Computación. La cátedra recomienda el uso de C++ o Java, y se sugiere consultar con los docentes la elección de otros lenguajes para la implementación.

La entrada y salida de los programas **deberá hacerse por medio de la entrada y salida estándar del sistema**. No se deben considerar los tiempos de lectura/escritura al medir los tiempos de ejecución de los programas implementados.

Deberá entregarse un informe impreso que desarrolle los puntos mencionados. Por otro lado, deberá entregarse el mismo informe en formato digital acompañado de los códigos fuentes desarrollados e instrucciones de compilación, de ser necesarias. Estos archivos deberán enviarse a la dirección `algo3.dc@gmail.com` con el asunto *"TP 2: Apellido_1, ..., Apellido_n"*, donde n es la cantidad de integrantes del grupo y *Apellido_i* es el apellido del i-ésimo integrante.

Problema 1: Plan de vuelo

Estamos desarrollando un sitio web de compra de pasajes aéreos llamado *aterrizar.com* (con la idea de competir contra un famoso sitio actual). Nuestro sitio ofrece, entre otras cosas, una opción de búsqueda de itinerarios innovadora. Eligiendo esta opción, el sistema indicará al usuario el itinerario que llega lo antes posible al destino especificado. La solución provista puede eventualmente utilizar la cantidad de tramos que sean necesarios, ya que el único objetivo es optimizar la fecha de llegada al destino.

Nos encontramos entonces frente al desafío de desarrollar un algoritmo que resuelva este problema. Es decir, conociendo las ciudades de origen y destino, A y B respectivamente, y una lista de vuelos disponibles debemos devolver un itinerario saliendo de A y llegando a B, de manera tal que la llegada a B ocurra lo antes posible. Para eso podemos combinar todos los vuelos que hagan falta pasando por cualquier ciudad intermedia en el camino. De cada vuelo de la lista de disponibles conocemos su origen y destino y las fechas y horas de partida y llegada. El itinerario devuelto por el algoritmo debe ser un itinerario factible y para ello deben cumplirse las siguientes condiciones:

- El primer vuelo debe salir de A y el último vuelo debe terminar en B.
- Si el itinerario usa más de un vuelo, la ciudad de llegada de cada vuelo debe coincidir con la ciudad de salida del vuelo siguiente. Además, debe haber al menos 2 horas de diferencia entre la llegada de un vuelo y la partida del vuelo siguiente.

Queremos que *aterrizar.com* sea el portal de vuelos número 1 del mundo, con lo cual necesitamos que el algoritmo desarrollado se pueda ejecutar muy rápido aun cuando la cantidad de vuelos disponibles sea grande. Para ello, se pide que el algoritmo desarrollado tenga una complejidad no peor que $\mathbf{O}(n^2)$, siendo n la cantidad de vuelos disponibles en la instancia a resolver. El algoritmo debe detectar los casos en los que no haya solución. Por otro lado, en caso de haber más de una solución óptima, el algoritmo puede devolver cualquiera de ellas. Se puede asumir que no saldrán dos vuelos al mismo tiempo de la misma ciudad. El item 3 del enunciado (demostración formal de correctitud) **es opcional** para este problema.

Formato de entrada: La entrada contiene una instancia del problema. La primera línea tiene el siguiente formato:

```
A B n
```

donde `A` y `B` son dos *strings* que indican las ciudades de origen y destino de la instancia y `n` es un entero no negativo que indica la cantidad de vuelos en la lista de disponibles. Ambos *strings* contienen sólo carácteres de letras y/o números. A esta línea le siguen `n` líneas, una para cada vuelo disponible, con el siguiente formato:

```
ori des ini fin
```

donde `ori` y `des` son *strings* que indican el origen y destino, respectivamente, del vuelo en cuestión, e `ini` y `fin` representan los momentos de partida y de llegada, respectivamente, del vuelo. Por simplicidad, `ini` y `fin` son enteros no negativos que representan la cantidad de horas pasadas desde el momento en que se hace la consulta, el cual asumimos como hora `0`. Asumimos entonces que todos los vuelos salen en horas "enteras".

Formato de salida: La salida debe contener una única línea. En caso de haber solución, la línea debe tener el siguiente formato:

```
fin k v1 v2 ... vk
```

donde `fin` es la hora de llegada a la ciudad `B` (representada de la misma manera que en la entrada), `k` es la cantidad de vuelos en el itinerario y los valores `v1`, ... , `vk` son enteros no negativos identificando a cada uno de los `k` vuelos del itinerario (en orden cronológico). Los vuelos disponibles se numeran de `1` a `n` por orden de aparición en la entrada. Si hay más de una solución óptima, el programa puede devolver cualquiera de ellas. En caso de que la instancia no tenga solución, la línea de salida deberá tener simplemente la palabra `no`.

Problema 2: Caballos salvajes

Tenemos un juego de mesa (para un jugador) que utiliza un tablero de ajedrez de $n \times n$ casillas e inicialmente se ocupan algunas de las casillas del tablero con caballos blancos. El objetivo del juego es reunir a todos los caballos en una misma casilla, y la idea es hacerlo con la menor cantidad de movimientos totales posible. Esta cantidad es la suma de los movimientos de todos los caballos en el tablero. Los movimientos deben respetar los movimientos permitidos para una pieza de caballo en el ajedrez[1]. Se permite que una casilla esté ocupada por más de un caballo a la vez.

Se pide escribir un algoritmo que tome las dimensiones del tablero, la cantidad de caballos y las posiciones iniciales de éstos e indique cuál es el mejor casillero que se puede elegir para reunir a todos los caballos minimizando la cantidad total de movimientos realizados. El algoritmo debe detectar los casos en los que no haya solución. Dado un tablero de $n \times n$ y k caballos iniciales, el algoritmo debe tener una complejidad temporal no peor que $\mathbf{O}(k \cdot n^2)$.

Formato de entrada: La entrada contiene una instancia del problema. La primera línea consta de dos números enteros positivos (separados por un espacio) n y k, donde n indica la cantidad de filas (y columnas) del tablero y k es la cantidad de caballos ubicados en el mismo. A esta línea le siguen k líneas, una para cada caballo, cada línea conteniendo dos enteros f y c (ambos entre 1 y n y separados por un espacio) indicando la posición (fila y columna, respectivamente) del caballo en cuestión.

Formato de salida: La salida debe contener una única línea con el siguiente formato:

```
f c m
```

donde `f` y `c` representan la casilla elegida como destino (fila y columna respectivamente) y `m` es la cantidad de movimientos totales necesarios para llevar a todos los caballos a esa casilla. Si hay más de una solución óptima, el programa puede devolver cualquiera de ellas. En caso de que la instancia no tenga solución, la línea de salida deberá tener simplemente la palabra `no`.

Problema 3: La comunidad del anillo

Nuestra empresa, *AlgoNET*, se dedica principalmente a brindar soluciones algorítmicas para problemas de redes y en esta ocasión nos enfrentamos al siguiente desafío.

Nuestro cliente quiere ofrecer un servicio particular sobre de una red existente de computadoras. Para ello, debe elegir algunas de estas computadoras para usar como servidores formando un *backbone* para el envío de cierto tipo de información y pretende que este *backbone* siga una topología de red de tipo *anillo* para que el ruteo de paquetes dentro del *backbone* sea rápido y seguro[2]. Además de este anillo de servidores, se requiere que todas las demás computadoras puedan tener acceso a los servidores del anillo. Este acceso puede ser vía un enlace directo a alguno de estos servidores o bien pasando a través de otros equipos en el medio.

La red consta actualmente de conexiones entre algunos pares de equipos, las cuales son utilizadas a diario por muchos servicios además del nuevo servicio que se pretende ofrecer. El nuevo servicio puede utilizar cualquiera de las conexiones existentes para comunicar los equipos entre sí, pero dada la alta demanda de ancho de banda, por cada conexión que utilice el nuevo servicio, se deberá pagar un cierto costo asociado al enlace utilizado. El objetivo de nuestro cliente es elegir el anillo de servidores y todas las demás conexiones necesarias, de maner de minimizar el costo incurrido por el uso de estos enlaces.

Se pide escribir un algoritmo que, dada la configuración de la red actual y los costos de uso por enlace, indique qué servidores y enlaces representarán el anillo y además qué otros enlaces utilizar para que todo equipo quede conectado al anillo de servidores. La solución dada por el algoritmo debe tener un costo mínimo, en la suma de los enlaces utilizados. Siendo n la cantidad de equipos de la red, el algoritmo debe tener una complejidad estrictamente mejor que $\mathbf{O}(n^3)$. El algoritmo debe detectar los casos en los que no haya solución. Por otro lado, en caso de haber más de una solución óptima, el algoritmo puede devolver cualquiera de ellas. Se sabe que entre cada par de equipos existe a lo sumo un único enlace que los comunica directamente.

[1] `http://es.wikipedia.org/wiki/Caballo_(ajedrez)`.
[2] `http://en.wikipedia.org/wiki/Ring_network`

Formato de entrada: La entrada contiene una instancia del problema. La primera línea contiene un entero positivo n que indica la cantidad de equipos de la red (numerados de 1 a n) y un entero no negativo m (separados por un espacio) que corresponde a la cantidad de enlaces disponibles. A esta línea le siguen m líneas, una para cada enlace, con el siguiente formato:

```
e1 e2 c
```

donde `e1` y `e2` representan los equipos en los extremos del enlace en cuestión (ambos enteros entre 1 y n) y `c` es el costo por utilizar dicho enlace.

Formato de salida: En caso de haber solución, la salida debe comenzar con una línea con el siguiente formato:

```
C Ea Er
```

donde `C` es el costo de la solución dada y `Ea` y `Er` son los enlaces utilizados para el anillo y para el resto de la red, respectivamente. A esta línea deben seguirle `Ea` líneas, una para cada enlace utilizado en el anillo y luego `Er` líneas, una para cada enlace utilizado fuera del anillo. Todas estas líneas deberán tener el siguiente formato:

```
e1 e2
```

donde `e1` y `e2` representan los equipos en los extremos del enlace en cuestión (ambos enteros entre 1 y n). Si hay más de una solución óptima, el programa puede devolver cualquiera de ellas. En caso de que la instancia no tenga solución, la salida deberá tener simplemente una línea con la palabra `no`.