UBA - FCEyN - Departamento de Computación

ALGORITMOS Y ESTRUCTURAS DE DATOS III
Trabajo Práctico Nº2

Primera entrega: 09-MAY-2008, hasta las 19:30 horas.
Segunda entrega: 23-MAY-2008, hasta las 19:30 horas.

Ver información general sobre los Trabajos Prácticos en la página de la materia en Internet.

- Desarrollar e implementar algoritmos para los problemas enumerados.
- Utilizando el pseudocódigo, calcular el orden de la complejidad de cada algoritmo, y decir si es constante, logarítmica, lineal, polinomial, exponencial o mayor **en función del tamaño de la entrada**.
- Aplicar los algoritmos a un conjunto de instancias de entrada, midiendo para cada instancia el tiempo de ejecución o la cantidad de operaciones. Elegir las instancias de manera tal que se pueda apreciar el comportamiento de los algoritmos e incluirlas en el soporte digital a entregar.
- Graficar para cada algoritmo el tiempo de ejecución o la cantidad de operaciones en función del *tamaño de la entrada*. **Comparar con la complejidad teórica calculada.**
- Leer los datos de entrada desde un archivo con el nombre `Tp2EjX.in`, donde `X` es el número del problema. Escribir los resultados en un archivo con el nombre `Tp2EjX.out`.
- Respetar los formatos de archivos que se indican en cada caso.
- Ignorar los costos de lectura y escritura de los archivos tanto al medir como al calcular complejidad.

1. **Enunciado** - Torneo de Tenis
   Un torneo de Tenis de eliminación simple consiste en varios partidos donde el perdedor de cada partido es eliminado del torneo y no vuelve a jugar un partido en ese torneo. El fixture del torneo se arma al comienzo del mismo tomando dos jugadores aún no eliminados para cada partido, hasta que quede sólo un jugador no eliminado, que resulta ser el ganador[1]. Con este esquema de fixture no sólo la destreza o el entrenamiento entran en juego para decidir el ganador sino que la *suerte* tiene un papel importante.

   Después de observar el entrenamiento de los participantes hay ciertos partidos de los cuales se puede saber con certeza su resultado, es decir, para ciertos jugadores a,b, se puede asegurar que a le gana a b.

   Diremos que el torneo puede ser *arreglado* para que gane x si existe un fixture de eliminación simple donde se puede asegurar que gane x.

   Encontrar todos los participantes x para los cuales el torneo puede ser *arreglado* para gane x.

   Modelar este problema utilizando grafos. Justificar el modelo.

   El mejor algoritmo que conocemos es de $O(n + m)$.

[1] Un fixture donde jueguen todos contra todos no tendría gracia.

Entrada `Tp2Ej1.in`
Cada instancia de entrada es definida en $1+m$ líneas, siendo m la cantidad de partidos que se conoce con certeza su resultado. La primer línea de la instancia contiene dos números n y m, siendo n la cantidad de participantes. Las siguientes m líneas contienen los m partidos cuyos resultados son conocidos, descriptos cada uno por una línea conteniendo dos enteros a_i y b_i representado la relación *antisimétrica* "a_i le gana a b_i". Los participantes se representan en la entrada como números entre 1 y n inclusive. El archivo conteniendo varias instancias termina con una línea con sólo un "0 0", la cual no debe ser considerada como una instancia.

Salida `Tp2Ej1.out`
Para cada instancia de la entrada, se debe escribir una línea en la salida conteniendo la cantidad de participantes para los cuales el torneo puede ser arreglado, y a continuación la lista de estos en orden creciente representando cada participante por un número, al igual que en la entrada. Todo par de valores consecutivos en una línea debe estar separado por exactamente un espacio en blanco y no debe haber espacios al comienzo o al final de la línea.

Ejemplo

`Tp2Ej1.in`	`Tp2Ej1.out`

`1 0`
`2 0`
`2 1`
`2 1`
`5 5`
`1 2`
`2 3`
`3 4`
`4 5`
`5 1`
`0 0`
	`1 1`
	`0`
	`1 2`
	`5 1 2 3 4 5`

2. **Enunciado** - Ruteo de barcos
Alrededor de un lago se encuentran ubicados n pueblos guerreros, numerados de 1 a n en sentido horario. Entre algunos de esos pueblos hay acuerdos comerciales que permiten intercambiar bienes en bote a través del lago. Los líderes de estos pueblos han decido crear una *ruta comercial* que una todos los pueblos y satisfaga las siguiente condiciones:

   - Comience en un pueblo A y termine en un pueblo B pasando por *todos* los demás pueblos en el medio.
   - Entre cada par de pueblos consecutivos en la ruta haya un acuerdo comercial.
   - La ruta no se cruce a sí misma para evitar choques entre los barcos.

   Desarrollar un algoritmo que decida si existe una ruta comercial para los líderes. **JUSTIFICAR** el algoritmo, explicando por qué el algoritmo planteado resuelve el problema.

   SUGERENCIA: Resolver usando programación dinámica. Considerar el orden circular de las ciudades de 1 a n y plantear la recursión en función de si es posible pasar sólo por todos los pueblos entre i y j (en el orden circular) con una subruta que termine en alguno de estos dos.

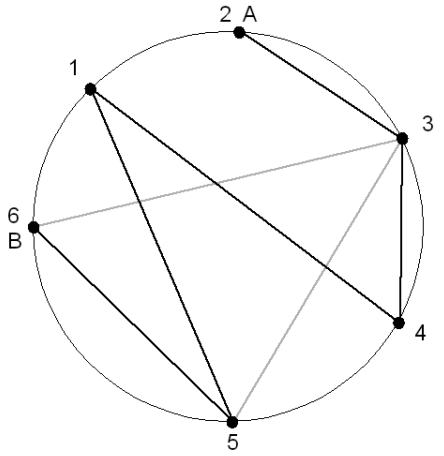
   Entrada `Tp2Ej2.in`
   Cada instancia de entrada es definida en $1 + m$ líneas, siendo m la cantidad total de acuerdos comerciales. La primer línea de la instancia contiene dos números n y m, siendo

n la cantidad de pueblos. Las siguientes m líneas contienen los m acuerdos comerciales, descriptos cada uno por una línea conteniendo un par de enteros no ordenado (a_i, b_i) representado la relación *simétrica* "a_i y b_i tienen un acuerdo comercial". Los pueblos se representan en la entrada como números entre 1 y n inclusive. Cada par no ordenado de pueblos no se repite en la instancia. El archivo conteniendo varias instancias termina con una línea con sólo un "0 0", la cual no debe ser considerada como una instancia.

Salida `Tp2Ej2.out`
Para cada instancia de la entrada, se debe escribir una línea en la salida. En caso que exista al menos una ruta comercial con esas características, se deben escribir las ciudades de 1 a n, ordenadas según alguna de las rutas, cada una separada por exactamente un espacio en blanco. En caso contrario se debe escribir una línea con un "-1". En todo caso no debe haber espacios al comienzo o al final de la línea.

Ejemplo

Tp2Ej2.in	Tp2Ej2.out
6 7	2 3 4 1 5 6
1 4	-1
1 5	
2 3	
6 3	
4 3	
5 6	
5 3	
4 3	
1 2	
1 3	
1 4	
0 0	



Primer instancia del ejemplo, con una ruta marcada entre los pueblos A (2) y B (6).

3. **Enunciado** - Patricia
Un radix tree o PATRICIA es un trie en el cual las cadenas de nodos con un solo hijo son compactadas y transformadas en un solo nodo. Esto permite mejorar el consumo de memoria de la estructura en el caso en que hay pocas cadenas definidas o que muchas cadenas tengan prefijos largos en común.

Esta compactación genera entonces las diferencias básicas entre los radix trees y los tries:

a) Todos los nodos internos de un radix tree tienen como mínimo dos hijos (excepto posiblemente la raiz).

b) Las ramas de un radix tree pueden estar etiquetadas con más de un caracter.

Si además el conjunto (o del conjunto de claves del diccionario) es un conjunto libre de prefijos, sucede que las cadenas (o los valores asociados a las claves) se encuentran solamente en las hojas. Un conjunto libre de prefijos es aquel en el cual ninguno elemento es prefijo de otro.

Implementar un conjunto de cadenas basado en estas ideas que soporte las siguientes operaciones:

a) Agregar una cadena al conjunto.

b) Consultar si una cadena pertenece al conjunto.

c) Sacar una cadena del conjunto.

d) Consultar la cantidad de cadenas del conjunto.

Donde las cadenas forman un conjunto libre de prefijos. Las tres primeras operaciones deben tener complejidad $O(\|s\|)$ donde s es la clave más larga ya definida y $\|s\|$ indica la longitud de s. La última operación debe tener complejidad de orden constante.

El conjunto de caracteres sobre el que se van a definir las cadenas son las 26 letras minúsculas del inglés.

Entrada `Tp2Ej3.in`
Cada instancia de entrada contiene $1 + n$ líneas. La primera línea indica n, y las siguientes n líneas especifican operaciones a realizar, una por línea:

- `agregar <cadena>`
- `pertenece <cadena>`
- `sacar <cadena>`
- `cardinal`

El archivo conteniendo varias instancias termina con una línea con sólo un "0", la cual no debe ser considerada como una instancia.

Salida `Tp2Ej3.out`
Para cada instancia de entrada se debe escribir una línea en la salida. La línea consiste en un dígito binario por cada operación `pertenece <cadena>`, y un natural por cada operación `cardinal`. Para la primera, se escribirá un 0 si la cadena no pertenece y un 1 en caso contrario, para la segunda, la cantidad de elementos; se deberá dejar exactamente un espacio entre cada número y no deberá haber espacios iniciales o finales.

Ejemplo

Tp2Ej3.in	Tp2Ej3.out
6 agregar hola pertenece hola agregar chau cardinal sacar hola pertenece hola 0	1 2 0