Cognoms, Nom Data

Algorísmia QT 2019–2020

Examen parcial **Una solució** **6 de Novembre de 2019**

Durada: 2h

Instruccions generals:

- L'exercici 1 s'ha de resoldre fent servir l'espai reservat per a cada resposta.
- Heu d'argumentar la correctesa i l'eficiència dels algorismes que proposeu. Per això podeu donar una descripció d'alt nivell de l'algorisme amb les explicacions i aclariments oportuns que permetin concloure que l'algorisme és correcte i té el cost indicat.
- Heu de justificar totes les vostres afirmacions, en cas contrari la nota de la pregunta serà 0.
- Podeu fer crides a algorismes que s'han vist a classe, però si la solució és una variació n'haureu de donar els detalls.
- Es valorarà especialment la claredat i concisió de la presentació.
- Entregueu per separat les vostres solucions de cada bloc d'exercicis (Ex 1, Ex 2 i Ex 3).
- La puntuació total d'aquest examen és de **10 punts**.

Exercici 1 (4.5 punts)

(a) (1 punt) Ens donen una taula d'enters positius, on cada enter pot tenir un nombre diferent de dígits, però el nombre total de dígits a la taula és n. Demostreu que l'aplicació directa de l'algorisme RADIX no pot ordenar la taula en temps $O(n)$. Dissenyeu un algorisme que ordene la taula en temps $O(n)$.

Sol. En la entrada podemos tener números con $k = O(n)$ dígitos, por ello RADIX ordenaría en $O(n^2)$ y no en $O(n)$.

Si miramos dos números, si uno de ellos tiene menos dígitos que el otro, es menor. Haremos la ordenación en dos pasos.

- Ordenamos primero los valores por número de dígitos ($\leq n$), utilizando counting sort, en $O(n)$.
- Ordenamos los valores con k dígitos, para cada valor $1 \leq k \leq n$, usando RADIX en $O(k(n+k))$. Como $k \leq n$, en $O(kn)$.

Si n_k es el número de valores con k dígitos, el coste del algoritmo es $O(n + \sum_{k=1}^{n} kn_k)$, pero $\sum_{k=1}^{n} kn_k$ es el número total de dígitos, n. Por lo tanto el coste del algoritmo es $O(n)$.

(b) (0.75 punts) Quin és el cost total d'executar n operacions PUSH, POP i MULTIPOP, sobre una pila (stack), si coneixem que al començament del proces la pila té s_0 elements i al finalitzar les n operacions la pila té s_n elements. Recordeu que MULTIPOP(k) extreu k elements de la pila, si hi han k o més elements, i en cas contrari buida la pila.

Si utilizamos analisis amortizado, partiendo de la pila vacia, por el método del ahorrador es suficiente con pagar 2 en cada PUSH y 0 en cualquiera de las dos operaciones de POP o MULTIPOP. Estas dos unidades se consume 1 al insertar el elemento en la cola y la segunda la ahorramos para cuando el elemento se extraiga. Esto nos da un coste en caso peor de $2n$.

Como la pila inicialmente no está vacía, para simular la hucha, deberíamos iniciar la ejecución con s_0 unidades ahorradas. Además sabemos que acabaremos con al menos s_n ahorrado. La diferencia entre estos dos valores es el coste adicional necesario para cubrir los POPs sin PUSH previo en la secuencia. Esto nos da un coste en caso peor de $2n + s_0 - s_n$.

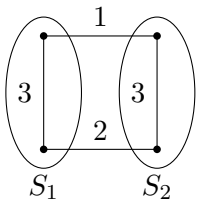
(c) (0.75 punts) Argumenteu si el següent algorisme per a trobar el MST d'un graf donat $G = (V, E)$, que és connex, no dirigit i amb pesos : $E \to \mathbb{Z}^+$, és o no és correcte, i si és correcte doneu la seva complexitat:

i. Particioneu V en dos subconjunts S_1 i S_2 (i.e. $V = S_1 \cup S_2$ i $S_1 \cap S_2 = \emptyset$ tal que cadascun dels subgrafs resultants G_{S_1} i G_{S_2} son connexes.

ii. Recursivament trobeu un MST T_{S_1} per a G_{S_1} i un MST T_{S_2} per a G_{S_2}.

iii. Com G és connex hi hauran arestes entre els vèrtexs de T_{S_1} i de T_{S_2}, escolliu l'atesta amb menys pes per a formar un MST de G.

Solució: FALS a la figura de sota l'algorisme previ tornara un T amb pes 7, mentre que el valor del MST és 6



(d) (1 punt) Considerem un alfabet Σ amb $n = 2^k$ caràcters $(x_0, x_1, \ldots, x_{n-1})$, i considerem un text T amb m caràcters, on cada caràcter de Σ apareix com a mínim un cop.

i. Sigui $S(T)$ el nombre be bits necessaris per a emmagatzemar T utilitzant compressió de longitud fixada, quina serà aquesta longitud?

ii. Sigui $H(T)$ el nombre de bits necessaris per a emmagatzemar T utilitzant Huffman. Definim l'eficiència $E(T) = S(T)/H(T)$. Descriviu, en funció de n i m, com serà un text T pel que $E(T)$ sigui el més petit possible.

iii. Descriviu, en funció de n i m, com serà un text T pel que $E(T)$ sigui el més gran possible. Doneu una expressió de l'eficiència com a funció de n. (Ajut: considereu el text $T = x_0, x_1, \ldots, x_{n-2}\underbrace{x_{n-1} \cdots x_{n-1}}_{m-(n-1)\text{cops}})$

Solució:

(i) Necessitem un codl de $k = \log n$ bits per representar els caràcters. Llavora la longitud serà km o $m \log n$.

(ii) L'eficiència és petita quan tots els símbols a T tenen la mateixa freqüència, el que ens dona $E(T) = 1$

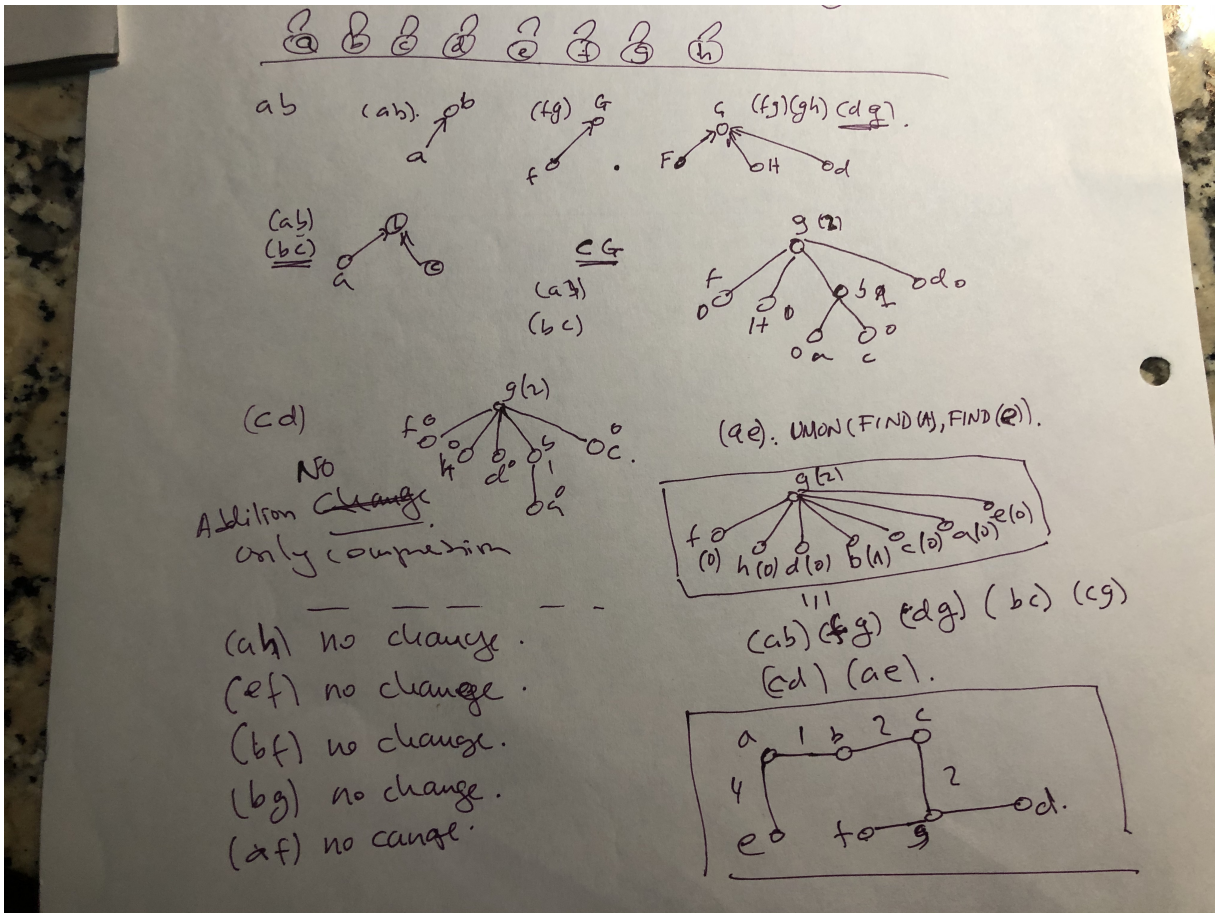
(iii) Com que cada caràcter ha d'aparèixer com a mínim un cop, l'eficiència serà gran quan tenim un caràcter amb freqüència alta. Així un cas pitjor per l'eficiència es el text proposat $T = x_0, \ldots, x_{n-2}, x_{n-1}, \ldots, x_{n-1}$. A T tots els caràcters tenen freqüència 1, menys x_{n-1} que en té $m - (n-1)$, lo que dona $S(T) = m - n - 1 + kn$. Llavors, per $m >> n$, tenim $E(T) = O(k) = O(\log n)$.

(e) (1 punt) Donat una graf connex i no dirigit $G = (V, E)$ amb pesos $w : E \to 10$ pel conjunts $V = \{a, b, c, d, e, f, g, h\}$ i el conjunt d'arestes amb pesos

$$\{(a,b)1, (b,c)2, (c,d)3, (a,e)4, (a,f)8, (e,f)5, (b,f)6, (b,g)6, (f,g)1, (c,g)2, (d,g)1, (g,h)1\},$$

simuleu l'aplicació de Kruskal mostrant com evolucionar UF fins a obtenir un MST de G amb temps $O(m)$.

Solució: FALS a la figura següent l'algorisme previ tornara un T amb pes 7, mentre que el valor del MST és 6


a b c d e f g h
ab
(ab). b a
(fg) G f
G (fg)(gh) (dg).
F H d
(ab)
(bc)
a b c
CG
(ab)
(bc)
g(2)
f H d b a c
(cd)
g(2)
f H d b c a c.
No
Addition Change.
only comparism
(ae). UNION(FIND(A), FIND(E)).
g(2)
f(0) H(0) d(0) b(1) c(0) a(0) e(0)
111
(ab) (fg) (dg) (bc) (cg)
(cd) (ae).
a 1 b 2 c
4
2
d.
e f g
(ah) no change.
(ef) no change.
(bf) no change.
(bg) no change.
(af) no cange.

Exercici 2 (3 punts) (Planificació). Ens donen un conjunt de treballs $S = \{a_1, a_2, \ldots, a_n\}$, a on per a completar el treball a_i es necessiten p_i unitats de temps de processador. Únicament tenim un ordinador amb un sol processador, per tant a cada instant únicament podem processar una treball. Sigui c_i el temps on el processador finalitza de processar a_i, que dependrà dels temps del treballs processats prèviament. Volem minimitzar el temps "mitja"necessari per a processar tots els treballs (el temps amortitzat per treball), es a dir volem minimitzar $\frac{\sum_{i=1}^{n} c_i}{n}$. Per exemple, si tenim dos treballs a_1 i a_2 amb $p_1 = 3, p_2 = 5$, i processem a_2 primer, aleshores el temps mitja per a completar els dos treballs és $(5 + 8)/2 = 6.5$, però si processem primer el treball a_1 i després a_2 el temps mitja per processar els dos treballs serà $(3 + 8)/2 = 2.2$

(a) (1.5 punts) Considerem que la computació de cada treball no es port partir, es a dir quan comença la computació de a_i les properes p_i unitats de temps s'ha de processar a_i. Doneu un algorisme que planifique la computació dels treballs a S de manera que minimitze el temps mitja per a completar tots els treballs. Doneu la complexitat del vostre algorisme i demostreu la seva correctesa.

(b) (1.5 punts) Considereu ara el cas de que no tots els treballs a S estan disponibles des de el començament, es a dir cada a_i porta associat un temps r_i fins al que l'ordinador no pot començar a processar a_i. A més, podem suspendre a mitges el processament d'un treball per a finalitzar més tard. Per exemple si tenim a_i amb $p_i = 6$ i $r_i = 1$, pot començar a temps 1, el processador aturar la seva computació a temps 3 i tornar a computar a temps 10, aturar a temps 11 i finalitzar a partir del temps 15. Doneu un algorisme que planifique la computació dels treballs a S de manera que es minimitze el temps mitja per a completar tots els treballs.

Solució.

Notemos que en la función a optimizar, $\frac{\sum_i c_i}{n}$, el denominador no depende de la planificación. Por lo tanto la planificación con coste mínimo es la del coste medio mínimo y viceversa. Los algoritmos que propondré resuelven el problema de buscar una planificación con coste mínimo.

(a) El algoritmo ordena los trabajos en orden creciente de p_i, y los planifica en ese orden. El coste es el de la ordenación, $O(n \log n)$.

Para ver que es correcto utilizo un argumento de intercambio. Supongamos que la planificación con coste mínimo no sigue el orden creciente de tiempo de procesado. Para simplificar asumo que el orden $a_1, \ldots, a_n$ es el que proporciona coste óptimo y que en él se produce una inversión, es decir $p_i > p_{i+1}$, para algún i.

Tenemos que $c_i = p_1 + \cdots + p_i$, por lo tanto

$$\sum_i c_i = np_1 + (n-1)p_2 + \cdots + (n-i)p_i + \cdots + 1p_n.$$

Si intercambiamos a_i con a_{i+1} solo cambia la contribución al coste de estos dos elementos que pasa de ser $(n-i)p_i + (n-i-1)p_{i+1}$ a ser $(n-i)p_{i+1} + (n-i-1)p_i$. El incremento en coste debido al intercambio es

$$(n-i)p_{i+1} + (n-i-1)p_i - [(n-i)p_i + (n-i-1)p_{i+1}] = p_{i+1} - p_i < 0.$$

Por tanto, la ordenación no es óptima y tenemos una contradicción.

(b) En este segundo apartado tendremos que seguir el criterio del apartado anterior, pero teniendo en cuenta que se incorporarán a lo largo del tiempo nuevos trabajos. La regla voraz del algoritmo es: procesar en cada instante de tiempo el proceso disponible al que le quede menos tiempo por finalizar. Utilizando el mismo argumento de intercambio que en el apartado (a) la regla voraz es correcta.

Tenemos que ir con cuidado en la implementación ya que el número total de instantes de tiempo es $\sum_i t_i$ y este valor puede ser exponencial en el tamaño de la entrada. Sin embargo, los tiempos en los que se para la ejecución de un proceso coinciden con los de disponibilidad de un nuevo proceso. Necesitamos controlar solo los instantes de tiempo en los que finaliza la ejecución de un proceso o en los que un proceso está disponible, un número polinómico.

El algoritmo ordena en orden creciente de r_i los procesos y mantiene una cola de prioridad con los procesos disponibles y no finalizados, utilizando como clave lo que le falta al proceso para finalizar su ejecución.

- Ordenar por r_i;
- Insertar en la cola todos los procesos con $r_k = r_1$ (clave p_k), i = primer proceso no introducido en la cola, $t = r_1$.
- mientras cola no vacia
  - $(j, p) = pop()$, si $t + p \leq r_i$ procesamos lo que queda de a_j, $t = t + p$, y repetimos hasta que la cola quede vacía o $t + p > r_i$.
  - Si $t + p > r_i$, insertamos $(j, t + p - r_i)$, $t = r_i$.
  - Insertamos en la cola todos los procesos con $r_k = r_i$ (clave p_k), i = primer proceso no introducido en la cola.

La implementicón es correcta ya que el conjunto de trabajos disponibles y no finalizados solo se modifican cuando hay un nuevo trabajo disponible o cuando iniciamos el procesamiento de uno de ellos. En el primer caso ese caso actualizamos la cola y el posible trabajo que se estaba ejecutando se interrumpe, y se vuelve a insertar en la cola con el tiempo restante. En el segundo, sacamos al proceso con menor tiempo para finalizar y iniciamos o reiniciamos su ejecución.

El coste de la ordenación es $O(n \log n)$ y el coste de cada inserción en la cola es $O(\log n)$. Para contabilizar el número total de inserciones, notemos que cada proceso se inserta en la cola cuando está disponible, lo que nos da n insercciones. Un proceso puede volver a reinsertarse en la cola varias veces, sin embargo, por cada tiempo de disponibilidad se reinserta un proceso como mucho, esto nos da $\leq n$ inserciones debido a paradas en la ejecución. Sumando todo, el coste del algoritmo es $O(n \log n)$.

Exercici 3 (2.5 punts) (RHEX). El joc de RHEX és una variació del joc de l'HEX inspirada en el Reversi. En aquest joc, hi han 2n fitxes, les fitxes tenen dos cares, una de color blanc i l'altre de color negre. El tauler es una xarxa $n \times n$ de cel.les hexagonals, com la de l'HEX. Hi han dos jugadors, un amb color negre i l'altre amb color blanc, que fan torns. A cada torn el jugador que li toca col·loca una fitxa mostrant el seu color a una posició encara buida. Un cop col·locada una fitxa al tauler, no es pot moure.

Quan es col.loca una fitxa al tauler es considera la *zona dominada*: totes les celles ocupades i connectades amb la nova cella amb un camí continuo de fitxes, normalment hi hauran fitxes de tos dos colors en aquests camins. Dues cel·les es consideren connectades si comparteixen una vora i les dues tenen una fitxa, independentment del color de les fitxes. Si a la zona dominada hi ha menys fitxes del color del jugador que té el torn que del color de l'altre jugador, es giren totes les fitxes a la zona dominada.

La partida acaba quan un dels dos jugadors guanya (aconsegueix tenir més de n fitxes del seu color) o quan s'hagin col·locat les $2n$ fitxes.

(a) (1.5 punts) Descriviu un algorisme eficient per al seguiment d'una partida. L'algorisme, després de cada jugada, ha de permetre mantenir el nombre total de fitxes de cada color al tauler i a més ha d'indicar si el jugador que acaba de jugar ha guanyat el joc de RHEX.

(b) (1 punt) Expliqueu com modificar el vostre algorisme per tal de que a més de mantenir la puntuació podeu actualitzar el tauler.

Solució.

El tablero tiene n^2 posiciones y en total solo puede haber $2n$ fichas. Por ello la estructura de datos que utilizaré solo contendrá información sobre las casillas en las que hay fichas.

Podemos asociar a cada estado del tablero, después de k, jugadas por un grafo $G(k)$ en el que los nodos son las posiciones de las k fichas y donde dos vértices están conectados si las casillas son contiguas.

(a) Observemos que la "zona dominada"es la componente conexa (cc) del grafo $G(k)$ que contiene la la ficha k-ésima.

La estructura eficiente para mantener las componentes conexas en un grafo que se va modifiando con el tiempo es Union Find (UF). Tendremos que modificar UF para que el nodo raíz de una componente guarde dos contadores n_u y b_u con el númeo de fichas negras, blancas, de la componente. Tendremos que añadir a la implementación de UF métodos que nos permitan actualizar los contadores.

Cuando introducimos una ficha de color c en la posición p:

- Makeset(p); actualizamos los contadores de p para que reflejen el color de la ficha.
- Hacemos un FIND para cada una de las posiciones contiguas y obtenemos la lista de hasta 6 identificadores diferentes de sets en la ED, podría darse el caso de que dos posiciones contiguas ya estuviesen en la misma cc.
- Calculamos n y b, sumando los contadores de las diferentes cc vecinas y sumamos 1 al color de la nueva ficha. Si se cumple la condición del problema intercambiamos los contadores. Actualizamos N y B.

- Hacemos UNION de todas las componentes conexas vecinas y del set correspondiente a la nueva ficha. Actualizamos los contadores de la cc para que coincidan con los valores calculados en el paso previo.
- Si N o B son $> n$ acabamos y si no continuamos con la siguiente ficha de color contrario.

El número total de operaciones UF por elemento es constante por lo que el coste del algoritmo es $O(n \log^* n)$.

Por definición el algoritmo mantiene correctamente las cc del tablero después de cada jugada y actualiza los contadores adecuadamente.

(b) Para modificar el tablero de forma eficiente observemos que solo pueden cambiar de color las fichas la cc que contiene la ficha que se acaba de colocar en el tablero. Necesitamos mantener información de los colores de las fichas y de las posiciones que forman cada componente conexa. Para ello añadiremos al nodo representante una lista de pares (posición,color) conteniendo todas las posiciones del tablero incluidas en la cc.

Esta información se puede mantener fácilmente. En un Makeset, añadimos una lista con un solo par. En un UNION, unimos las dos listas en tiempo constante. Con esta implementación podemos cambiar el color de las fichas en tiempo proporcional al número de fichas que cambian de color.