Algoritmi e Strutture Dati – 9/1/2024

Esercizio A1 – Complessità – Punti ≥ 8

Per calcolare la complessità computazionale di fun(), definiamo l'equazione di ricorrenza. L'algoritmo divide la matrice in quattro parti uguali e chiama se stesso ricorsivamente su ciascuna di queste parti. Inoltre, c'è un ciclo **while** che incide sulla complessità.
Il ciclo **while** itera sui limiti della matrice riducendo l'indice i o l'indice j, partendo da e_i ed e_j fino a quando non raggiungeranno s_i o s_j. Questo significa che nel migliore dei casi il ciclo farà n iterazioni (correndo lungo il bordo destro o in basso della matrice), nel peggiore dei casi farà $2n$ interazioni (alternando riduzioni dell'indice i a riduzioni dell'indice j). La complessità del ciclo è quindi $O(n)$.
Sia $T(n)$ la complessità temporale dell'algoritmo. Poiché la dimensione iniziale è una potenza di 2, tutte le sottomatrici hanno dimensioni che sono potenze di 2, e questo semplifica l'analisi. L'algoritmo divide la matrice in quattro parti di dimensione $\frac{n}{2}$ e chiama se stesso su ciascuna di queste parti. Quindi, l'equazione di ricorrenza può essere espressa come:

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

dove $O(n)$ rappresenta la complessità del ciclo **while**.
Utilizzando il Master Theorem, abbiamo che $a = 4$, $b = 2$, $\alpha = \log_2 4 = 2$ e $\beta = 1$. La complessità dell'algoritmo è quindi $\Theta(n^2)$.

Possibili errori Alcuni di voi hanno scritto $T(n) = 4T(n/4)$; il fatto che l'algoritmo venga chiamato su quattro quadranti non significa che la dimensione n venga divisa per 4 (casomai $n^2/4$), ma che la dimensione del lato (che è il valore che utilizziamo per rappresentare la dimensione) è divisa per 2.

Esercizio A2 – Tree Repair Shop – Punti ≥ 10

Il grafo originale è un albero, il che significa che è connesso e aciclico. Sono presenti $n - 1$ archi; aggiungendo uno e un solo arco, si è creato un ciclo e il grafo ha ora n archi. Eliminando uno qualunque degli archi dell'unico ciclo presente, si ripristina la proprietà di essere un albero.
Modifichiamo l'algoritmo per l'individuazione di cicli in grafi non orientati, conservando il meccanismo che evita di tornare sul nodo di provenienza dopo aver attraversato un arco. Quando si individua un nodo già visitato, si cancella l'arco che porta verso quel nodo e si termina la ricorsione restituendo **true**, che viene intercettato nelle chiamate ricorsive e fa terminare precocemente la ricorsione.
Il fatto che si termini la ricorsione quando si trova l'arco incriminato rende il costo della visita pari $O(n + m)$. Il costo della rimozione dell'arco è $O(n)$ nel caso pessimo. Sfruttando poi il fatto che il grafo di input ha n archi, si può dire che la complessità risultante è $O(n)$. Questa versione o un algoritmo equivalente ha preso il 110%, se lo studente ha sottolineato il fatto che la visita dell'intero grafo avviene solo nel caso pessimo.

```
repairTree(GRAPH G)
    boolean[] visited = new boolean[G.size()] = {false}
    rtRec(G, 1, nil, visited)
```

```
boolean rtRec(GRAPH G, NODE curr, NODE prev, boolean[] visited)
    visited[curr] = true
    foreach next ∈ G.adj(curr) − {prev} do
        if visited[next] then                          % Trovato il nodo finale del ciclo
            G.deleteEdge(curr, next)
            return true
        else
            if rtRec(G, next, curr, visited) then
                return true
    return false
```

DFS o BFS con predecessore, senza meccanismo di terminazione precoce – Corretta Molti degli algoritmi proposti risolvevano correttamente il problema, ma visitando l'intero grafo anche dopo che si è trovato l'arco incriminato. L'algoritmo risultante è $\Theta(n)$, non $O(n)$. Comunque è corretto e prende il 100%.

DFS senza predecessore – Errata In molti compiti, la struttura dell'algoritmo era simile a quella presentata, ma non si è tenuto conto del fatto che quando si attraversa un arco (u, v) e si arriva ad un nodo v, si trova immediatamente l'arco (v, u) che porta al nodo u già visitato. L'arco non orientato (u, v) viene quindi cancellato, con l'effetto che vengono cancellati tutti gli archi del grafo.

BFS senza predecessore – Errata Altri studenti hanno preferito utilizzare una BFS. La scelta è ovviamente legittima, ma questo rende più complicato tenere conto del predecessore e si ricade nell'errore precedente. Era comunque possibile utilizzare correttamente una BFS, aggiungendo una coppia (nodo, predecessore) alla coda invece che il solo nodo, oppure memorizzando il nodo di provenienza tramite un vettore *parent*.

Esercizio A3 – k-Ripetizioni – Punti ≥ 12

Soluzione $O(n^4)$ Una soluzione molto costosa considera tutte le possibili sottostringhe contigue, utilizzando due cicli annidati di dimensione $O(n)$; per ognuno dei caratteri della stringa (terzo ciclo annidato di dimensione $O(n)$), conta quante volte questo carattere appare nella sottostringa (quarto ciclo annidato di dimensione $O(n)$). Il costo computazionale è quindi $\Theta(n^4)$. Si può ottimizzare leggermente il terzo ciclo (uscendo quando si trova una frequenza minore di k), ma tanto la sostanza non cambia - la complessità dell'algoritmo resterebbe $O(n^4)$.

```
int[] krepeating(ITEM[] S, int n, int k)
    int maxSoFar = 1
    for start = 1 to n do
        for end = start + 1 to n do
            boolean ok = true
            for i = start to end do
                int count = 0
                for j = start to end do
                    if S[i] == S[j] then
                        count = count + 1
                if count < k then
                    ok = false
            if ok then
                maxSoFar = max(maxSoFar, end − start + 1)
    return maxSoFar
```

Soluzione $O(n^3)$ Una soluzione migliore considera sempre tutte le possibili sottostringhe, ma calcola la frequenza dei caratteri di una sottostringa in tempo $O(n)$ utilizzando un dizionario che associa ad ogni carattere la frequenza nel vettore.

```
int[] krepeating(ITEM[] S, int n, int k)
    int maxSoFar = 1
    for start = 1 to n do
        for end = start to n do
            HASH freq = Hash()
            for i = start to end do
                if freq.lookup(S[i]) == nil) then
                    freq.insert(S[i], 0)
                freq.insert(S[i], freq.lookup(S[i]) + 1)
            boolean ok = true
            foreach c ∈ freq do
                if freq.lookup(S[i]) < k then
                    ok = false
            if ok then
                maxSoFar = max(maxSoFar, end − start + 1)
    return maxSoFar
```

Soluzione $O(n^2L)$ È possibile migliorare la soluzione evitando di resettare la tabella hash tutte le volte che si passa da un sottovettore $A[start \dots end]$ a un sottovettore $A[start \dots end + 1]$; è sufficiente continuare a contare le frequenze tenendo fisso l'estremo *start* fino a quando non si arriva al termine del vettore. Tuttavia, la valutazione se l'insieme di frequenze rispetta il vincolo su k va eseguita tutte le volte che viene considerato un nuovo elemento, e questo porta la complessità a $O(n^2L)$, dove L è il numero di caratteri distinti presenti nel sottovettore. Nel caso pessimo (tutti caratteri distinti), la complessità è comunque $O(n^3)$, ma si ottiene un miglioramento quando i caratteri distinti sono relativamente pochi.

```
int[] krepeating(ITEM[] S, int n, int k)
    int maxSoFar = 0
    for start = 1 to n do
        HASH freq = Hash()
        for end = start to n do
            if freq.lookup(S[i]) == nil) then
                freq.insert(S[i], 0)
            freq.insert(S[i], freq.lookup(S[i]) + 1)
            boolean ok = true
            foreach c ∈ freq do
                if freq.lookup(S[i]) < k then
                    ok = false
            if ok then
                maxSoFar = max(maxSoFar, end − start + 1)
    return maxSoFar
```

Soluzione $O(n^2)$ È possibile ridurre ancora la complessità utilizzando un approccio divide-et-impera. L'idea è la seguente: dato una sottostringa $S[start \dots end]$, si calcola la frequenza di ogni carattere del sottovettore utilizzando un dizionario, come fatto sopra. Questa operazione costa $O(n)$. A questo punto, si verifica se tutti i caratteri del vettore hanno una frequenza maggiore o uguale a k. Se così è, si ritorna la dimensione del sottovettore. Se un carattere $S[i]$ ha una frequenza inferiore a k, è ovvio che non può essere utilizzato in nessuna sotto-sottostringa della sottostringa $S[start \dots end]$; quindi si divide la stringa in due parti, $S[start \dots i-1]$ e $S[i+1 \dots end]$ e si prende ricorsivamente il massimo.
Siccome il carattere con frequenza inferiore si può trovare all'inizio della sottostringa, l'equazione di ricorrenza nel caso pessimo è la seguente:

$$T(n) = \begin{cases} T(n-1) + n & n > 1 \\ 1 & n = 0 \end{cases}$$

e la complessità risultante è $O(n^2)$.

```
int[] krepeating(ITEM[] S, int n, int k)
    return krRec(S, 1, n, k)
```

```
int[] krRec(ITEM[] S, intstart, int end, int k)
    if end < start then
        return 0
    else
        HASH freq = Hash()
        for i = start to end do
            if freq.lookup(S[i]) == nil) then
                freq.insert(S[i], 0)
            freq.insert(S[i], freq.lookup(S[i]) + 1)
        for i = start to end do
            if freq.lookup(S[i]) < k then
                return max(krRec(A, start, i − 1, k), krRec(A, i + 1, end, k))
        return end − start + 1
```

Una soluzione che cerca il carattere con la frequenza minore di k a partire dal centro (andando nelle due dirizioni sinistra-destra in parallelo) anzichè dall'inizio potrebbe dividere la stringa in due parti più equilibrate, portando la complessità più verso $O(n \log n)$; ma sicuramente ci sono dei casi "difficili" che rendono la complessità comunque $O(n^2)$.

Esercizio B1 – Commissioni di laurea – Punti ≥ 8

Il problema può essere risolto tramite una rete di flusso.
Creiamo i seguenti nodi:

- una sorgente s;
- un nodo per ogni professore, $p_1 \dots p_n$;
- due nodi per ogni commissione i, $1 \leq i \leq k$, chiamati c_i^2 e c_i^3; c_i^2 serve a raccogliere i due professori esperti, c_i^3 serve a raccogliere gli altri 3 professori;
- un pozzo t

Creiamo i seguenti archi:

- colleghiamo s a ogni professore, con peso 1 (in quanto può partecipare a una solo commissione) (n archi);
- colleghiamo ogni professore ai nodi c_i^2, dove i è l'area didattica in cui il professore è esperto (n archi), con peso 1
- colleghiamo ogni professore ai nodi c_i^3, per tutte le aree didattiche in cui il professore non è esperto ($n(k-1)$ archi), con peso 1;
- colleghiamo tutti i nodi c_i^2 e c_i^3 al pozzo ($2k$ archi), con peso 2 e 3, rispettivamente.

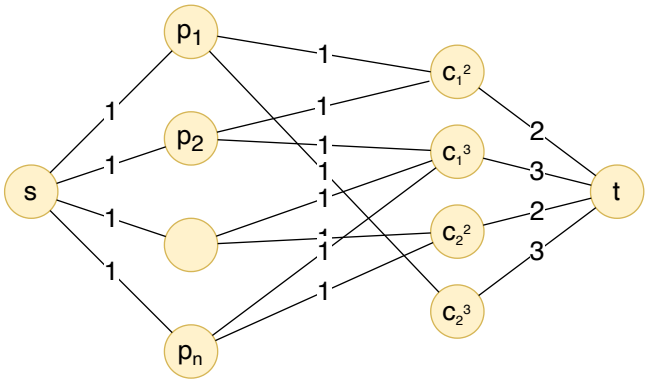
La dimensione di V ed E è la seguente:

$$|V| = 2 + n + 2k$$
$$|E| = n + nk + 2k = n + nk + 2k$$

Utilizzando il limite di Ford-Fulkerson e tenuto conto che il flusso massimo è limitato superiormente da n, il costo totale dell'algoritmo proposto è

$$O(n \cdot (|V| + |E|) == O(n(2 + 2n + nk + 4k)) = O(n^2 k)$$

La figura seguente mostra un piccolo esempio con n professori e due commissioni.



Esercizio B2 – Tutti i cicli! – Punti ≥ 10

La soluzione può essere ottenuta adattando una visita in profondità e utilizzando la tecnica del backtracking. Si noti innanzitutto che ogni ciclo che passa attraverso *start* può essere espresso come un ciclo che parte da *start* e arriva a *start*. Quindi, eseguiamo una visita in profondità, segnando come visitato ogni nodo che si raggiunge e marcandolo come non visitato una volta completata la sua visita.
Viene utilizzato uno stack aggiuntivo, in cui vengono memorizzati i nodi visitati finora. Quando si incontra di nuovo il nodo di partenza, si stampa il contenuto dello stack (assumiamo che venga stampato dal fondo alla cima) e poi si stampa di nuovo il nodo di partenza, per chiudere il ciclo.

```
boolean printCycles(Graph G, Node start)
    Stack stack = Stack()
    boolean[] visited = new boolean[1...G.n] = {false}
    printCyclesRec(G, start, start, stack, visited)
```

```
printCyclesRec(Graph G, Node start, Node curr, Stack stack)
    stack.push(curr)
    visited[curr] = true
    foreach next ∈ G.adj(curr) do
        if next == start then
            print stack, start
        else
            printCycleRec(G, start, next, stack)
    visited[curr] = false
    stack.pop()
```

Analisi complessità, semplice

- Come caso ottimo, consideriamo un albero o addirittura una foresta. Il costo dell'algoritmo sarà lineare nel numero dei nodi (il numero degli archi è inferiore o uguale a $n-1$), perché verrà effettuata una semplice visita e non verrà stampato nulla. Possiamo dire quindi che l'algoritmo ha costo $\Omega(n)$.
- Come caso pessimo, consideriamo un grafo completo, dove tutti i nodi sono connessi a tutti gli altri nodi. Partendo da un nodo, esistono $(n-1)!$ possibili cicli di $n-1$ nodi, che partono da *start*, attraversano una permutazione di $n-1$ nodi e ritornando a *start*. Poichè la stampa dei cicli costa $O(n)$, nel caso pessimo il costo è $T(n) = \Omega(n!)$; il limite è inferiore perché non stiamo considerando cicli di lunghezza inferiore, che dovrebbero essere comunque contati. Tuttavia, come analisi è sufficiente, perché ci dice che l'algoritmo ha costo almeno fattoriale nel caso pessimo.

Analisi complessità, completa (non necessaria) È possibile osservare che il numero di cicli contenenti un nodo in un grafo completo può essere calcolato nel modo seguente:

- $(n-1)$ cicli di lunghezza 2 ($start \to u \to start$, con u, $start$ distinti);
- $(n-1)\cdot(n-2)$ cicli di lunghezza 3 ($start \to u \to v \to start$, con $u, v, start$ distinti);

- ...
- $(n-1)!$ cicli di lunghezza $n-1$ (partendo da *start*, attraversando una permutazione di $n-1$ nodi e ritornando a *start*)

Il numero di cicli effettivamente stampati nel caso di un grafo completo è quindi pari a:

$$C(n) = \sum_{k=1}^{n-1} \frac{(n-1)!}{(n-1-k)!}$$

Il numero di cicli da stampare cresce, partendo da $n = 2$, in questo modo: 1, 4, 15, 64, 325, 1956, 13699, 109600, 986409. Questa sequenza corrisponde alla sequenza https://oeis.org/A007526 registrata nel sito OIS; in effetti, corrisponde al numero di permutazioni di sottoinsiemi non vuoti di $n-1$ elementi. Utilizzando la formula di Joseph K. Horn citata nella pagina, si ottiene una complessità pari a:

$$C(n) = \lfloor e \cdot (n-1)! - 1 \rfloor$$

Possiamo quindi concludere che il costo computazionale nel caso pessimo è $O(n!)$, derivante da $O(n)$ per la stampa dei cicli e $O((n-1)!)$ per il numero di cicli. Combinato con il risultato che $T(n) = \Omega(n!)$ nel caso pessimo, si può concludere che $T(n) = \Theta(n!)$ sempre nel caso pessimo.

Esercizio B3 – Quanti quadrati! – Punti ≥ 12

Questo problema è simile a un problema discusso nelle esercitazioni in aula: determinare la dimensione della più grande sottomatrice quadrata presente nella matrice.
Nel peggiore dei casi, può essere risolto in tempo $O(n^5)$: per ognuna delle n^2 celle (i, j) e per ogni possibile dimensione k delle n dimensioni, si verifica se esiste un quadrato di dimensione $k \times k$ la cui cella in alto a sinistra è in posizione (i, j), in tempo $O(n^2)$. In altre parole, mediante cinque cicli annidati.
È possibile trovare una soluzione in $O(n^4)$ nel modo seguente: per ognuna delle n^2 celle (i, j), si cerca la dimensione k del più grande quadrato la cui cella in alto a sinistra è in posizione (i, j) e si tiene conto del fatto che, se esiste un quadrato di dimensione k, allora esistono anche quadrati di dimensione $k-1, k-2, \ldots, 1$ che iniziano in (i, j). Questo implica l'utilizzo di quattro cicli annidati.
È possibile trovare una soluzione in $O(n^3)$ definendo una relazione di programmazione dinamica $DP[i][j][k]$, che stabilisce se esiste una sottomatrice quadrata di dimensione k il cui angolo in basso a destra si trova in posizione (i, j).
Riutilizziamo il codice discusso nell'esercitazione, che calcolava una matrice $DP[1 \ldots n][1 \ldots n]$ memorizzando in $DP[i][j]$ la dimensione della più grande sottomatrice quadrata il cui angolo in basso a destra si trova nella cella (i, j).
Tutte le celle (i, j) con valore $A[i][j] = 0$ hanno una dimensione massima pari a 0. Per le celle con valore 1, se si trovano sui bordi superiore ($i = 1$) o sinistro ($j = 1$), il loro valore è pari a 1. Altrimenti, per formare un quadrato di dimensione k che termina in (i, j), è necessario che esistano quadrati di dimensione almeno $k-1$ nelle celle "sopra" $(i-1, j)$, "sopra a sinistra" $(i-1, j-1)$ e "a sinistra" $(i, j-1)$; si prende quindi il valore minimo tra queste celle e si somma 1.

$$DP[i][j] = \begin{cases} 0 & \text{se } A[i][j] = 0 \\ 1 & \text{se } A[i][j] = 1 \wedge (i = 1 \vee j = 1) \\ \min\{DP[i-1][j], & \\ \quad DP[i-1][j-1], & \\ \quad DP[i][j-1]\} + 1 & \text{altrimenti} \end{cases}$$

È possibile calcolare DP con due cicli annidati. Una volta stabilita la dimensione massima delle sottomatrici che terminano in ognuna delle celle (i, j), si può utilizzare l'osservazione proposta per passare dalla soluzione $O(n^5)$ alla soluzione $O(n^4)$, ovvero che se esiste una sottomatrice quadrata di dimensione $DP[i][j]$ che termina in posizione (i, j), allora esistono anche sottomatrici quadrate che terminano in posizione (i, j) di dimensione $DP[i][j] - 1$, $DP[i][j] - 2$, ..., 1. Dobbiamo quindi contare $DP[i][j]$ quadrati per ognuna delle celle (i, j). È quindi sufficiente restituire la somma della matrice DP.

```
int maxSquare(boolean[][] A, int n)
    int[][] DP = new int[1...n][1...n]
    for i = 1 to n do
        DP[i][1] = A[i][1]
        DP[1][i] = A[1][i]
    for i = 2 to n do
        for j = 2 to n do
            if A[i][j] == 0 then
                DP[i][j] = 0
            else
                DP[i][j] = min(DP[i − 1][j], DP[i − 1][j − 1], DP[i][j − 1]) + 1
    return sum(A, n)                                  % Return matrix sum, Θ(n^2)
```

La complessità di questo algoritmo è ovviamente $\Theta(n^2)$.