Algoritmi e Strutture Dati – 23/08/2023

Esercizio A1 – Complessità – Punti ≥ 8

Dal codice, è possibile vedere che la matrice di input è quadrata. Consideriamo, per semplicità, il caso in cui n è una potenza di 2. Il risultato è valido anche per dimensioni che non rispettano tale vincolo.

Il codice calcola la somma di tutti i valori, in tempo $O(n^2)$, e poi chiama ricorsivamente se stessa su due porzioni di dimensione $n/2$, che corrispondono al secondo e quarto quadrante se interpretiamo la matrice come un piano cartesiano. Il caso base si verifica quando $x_1 = x_2$ e $y_1 = y_2$; in questo caso, la somma dei valori richiede tempo $O(1)$ (essendoci un valore solo da sommare) e non vengono effettuate chiamate ricorsive.

La funzione di complessità è quindi:

$$T(n) = \begin{cases} 1 & n \leq 1 \\ 2T(n/2) + n^2 & n > 1 \end{cases}$$

Applicando il Master Theorem, otteniamo $\alpha = \log_2 2 = 1$ e $\beta = 2$. Poichè $\beta > \alpha$, per il Master Theorem $T(n) = \Theta(n^2)$.

Esercizio A2 – Diretto o no? – Punti ≥ 10

È possibile risolvere il problema in molti modi diversi, con complessità quali $O(mn)$, $O(n^2)$, $O(m+n)$.
La soluzione più banale consiste nel verificare, per ogni arco (u, v) di G, se u è nel vettore/lista di adiacenza di v. In un grafo completo, questo algoritmo ha complessità $O(mn)$, perchè per ognuno degli m archi è necessario effettuare una ricerca nel vettore/lista di adiacenza del nodo destinazione, con complessità $O(n)$.

```
boolean isUndirected(Graph G)
  foreach x ∈ G.V() do
      foreach y ∈ G.adj(x) do
          if x ∉ G.adj(y) then
              return false
  return true
```

Una soluzione leggermente più efficiente consiste nel costruire una matrice di adiacenza a partire dalle liste di adiacenza e poi verificare se la matrice è simmetrica, con costo $O(n^2)$.

```
boolean isUndirected(Graph G)
  boolean[] M = new int[1...G.n][1...G.n] = {false}
  foreach x ∈ G.V() do
      foreach y ∈ G.adj(x) do
          M[x][y] = true
  for x = 1 to n − 1 do
      for y = x + 1 to n do
          if M[x][y] ≠ M[y][x] then
              return false
  return true
```

Il problema della prima soluzione è che sebbene adj() restituisce un insieme, normalmente diamo per scontato che questo insieme sia implementato come una lista/vettore non ordinato. Questo significa che l'operatore $\in$ abbia costo $O(n)$. Se si assume che l'insieme sia implementato in modo più efficiente (ma generalmente sprecando più memoria), si possono ottenere complessità migliori.
Per la regola che è lecito utilizzare le "API" fornite dal libro e dalle slide, ma se volete che si comportino diversamente da quanto descritto nel libro, dovete re-implementarle, proponiamo una soluzuone che crea un vettore di insiemi basati su tabella hash, uno per ogni nodo x, che contengono i nodi vicini di x. In questo modo, l'operatore $\in$ ha costo $O(1)$ e il costo totale è $O(m+n)$, perché l'istruzione **if** viene eseguita $O(m)$ volte e inizializzazione e scorrimento dei nodi costano $O(n)$.

```
boolean isUndirected(Graph G)
    Set[] neighbors = new Set[1...G.n]
    for x = 1 to n do
        neighbors[x] = Set()
    foreach x ∈ G.V() do
        foreach y ∈ G.adj(x) do
            neighbors[x].insert(y)
    foreach x ∈ G.V() do
        foreach y ∈ G.adj(x) do
            if x ∉ neighbors[y] then
                return false
    return true
```

Esercizio A3 – Prossimità – Punti ≥ 12

La soluzione proposta sfrutta il meccanismo di calcolo della posizione nel vettore degli alberi binari heap. L'idea è che la radice si trova in posizione 1; i figli sinistro e destro del nodo i-esimo si trovano in posizione $2i$ e $2i+1$. La funzione principale richiama la funzione ricorsiva locate(), che cerca un nodo nell'albero e restituisce il suo livello e la sua posizione nell'albero heap.
Se i due nodi hanno lo stesso livello, si calcola la differenza di posizione; visto che si chiede di calcolare il numero di nodi intermedi nel livello, si sottrae 1. Se i due nodi non appartengono allo stesso livello, si ritorna $+\infty$.

```
proximity(Tree T, Tree t1, Tree t2)
    int level1, pos1 = locate(T, t1, 0, 1)
    int level2, pos2 = locate(T, t2, 0, 1)
    if level1 == level2 then
        return |pos2 − pos1| − 1
    return +∞
```

La funzione locate() prende in input la radice t del sottoalbero che si sta esaminando, il nodo x che si sta cercando, il livello *level* della radice e la posizione *pos* della radice nel vettore heap. La funzione restituisce due valori: il livello e la posizione del nodo, se è stato trovato, (-1, -1) se non è stato trovato. La parte ricorsiva esplora i due sottorami sinistro e destro. Se il nodo viene trovato in uno dei due rami, si restituisce il livello corrente e la posizione. Il calcolo del livello e della posizione viene effettuato nel passaggio dei parametri, passando $level+1$ e $pos \cdot 2$ oppure $pos \cdot 2+1$.

```
(int, int) locate(Tree t, Tree x, int level, int pos)
    if t == nil then
        return (−1, −1)
    else if t == x then
        return (level, pos)
    else
        int, int level, pos = locate(t.left, x, level + 1, pos · 2)
        if level ≥ 0 then
            return (level, pos)
        int, int level, pos = locate(t.right, x, level + 1, pos · 2 + 1)
        if level ≥ 0 then
            return (level, pos)
        return (−1, −1)
```

La complessità è ovviamente $O(n)$, in quanto si tratta di una visita in profondità.

Casi particolari Ci sarebbero tantissimi casi particolari da distinguere, che sono stati evitati dall'assunzione che entrambi t_1 e t_2 appartengano all'albero e siano distinti. Ne parliamo qui per illustrare cosa succede, questa discussione non era richiesta dall'esercizio.

- Se t_1 e t_2 sono lo stesso nodo, viene restituito -1, perché il livello è uguale e la differenza di posizione è 0;
- Se uno dei due nodi non appartiene all'albero e l'altro sì, viene restituito $+\infty$, perché uno ha livello -1 e l'altro un livello positivo o nullo;

- Se entrambi i nodi non appartengono all'albero, viene restituito -1

Esercizio B1 – Rete idrica – Punti ≥ 8

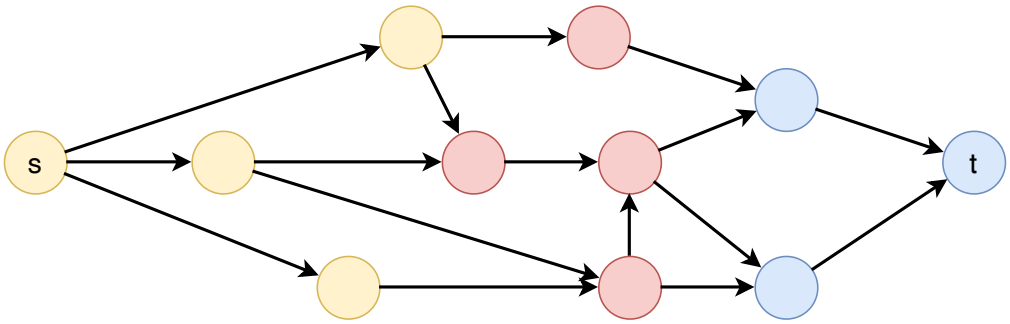
Il problema è risolvibile tramite reti di flusso; si costruisce una rete di flusso $G = (V', E', s, t, c')$ e si modifica più volte per azzerare il flusso attraverso ognuno degli archi $(x, y) \in E_{per}$.
L'insieme dei nodi V' è costituito dai nodi in V più un nodo sorgente s e un nodo pozzo t: $V' = V \cup \{s, t\}$.
L'insieme degli archi E' è costituito da:

- tutti gli archi $(u, v) \in E$, con capacità $c'(u, v) = c(u, v)$;
- un arco da s a ogni nodo v in V_{in}, con capacità $c'(s, v) = in(v)$;
- un arco da ogni nodo in V_{out} a t, con capacità $c'(v, t) = out(v)$;

Detto $tot = \sum_{v \in V_{out}} out(v)$ la richiesta complessiva di acqua dei consumatori, l'esecuzione dell'algoritmo di Ford-Fulkerson ha costo $O(tot \cdot (|V| + |E|))$ (in quanto gli archi aggiunti sono in al più $|V|$ e in nodi aggiunti sono 2).
Per valutare la capacità della rete su ognuno degli archi in E_{per}, si ripete l'esecuzione mettendo temporaneamente a zero la capacità dell'arco testato. Il costo complessivo è quindi:

$$T(n) = |E_{per}| \cdot tot \cdot (|V| + |E|))$$

La figura seguente mostra una possibile rete; i nodi sorgenti sono ocra, i nodi consumatori azzurri e i nodi distributori rosa.



Esercizio A2 – Sub-palindrome – Punti ≥ 10

La soluzione proposta si basa sulla tecnica del backtrack. Il procedimento genera tutte le sottosequenze della stringa in input, stampando soltanto quelle che sono palindrome. Questa condizione viene verificata dalla funzione isPalindrome().

La chiamata ricorsiva utilizza due indici: i, che rappresenta il carattere attualmente in considerazione, e j, che rappresenta il numero di caratteri già inseriti nel vettore di appoggio sub. Quando $i = 0$, tutti i caratteri sono stati presi in considerazione, ed è possibile verificare se la stringa è palindroma e, in caso affermativo, stamparla.

Da notare che, per ridurre il numero di parametri nelle chiamate e per semplicità, l'indice i varia da n a 0, con 0 come condizione di uscita, mentre l'indice j varia da 0 a n. Pertanto, le stringhe vengono memorizzate in sub in ordine inverso; tuttavia, poiché stampiamo solo le stringhe palindrome, il risultato rimane identico.

La complessità temporale della soluzione è $O(n \cdot 2^n)$, situazione che si verifica quando la stringa è composta da un unico carattere ripetuto n volte. In questo caso, ogni sottosequenza viene stampata, con un costo di $O(n)$ per ciascuna. La complessità nel caso peggiore è $\Omega(2^n)$, situazione che si verifica quando tutti i caratteri sono distinti. In questo scenario, vengono comunque generate tutte le 2^n sottosequenze, ma soltanto quella vuota e quelle composte da singoli caratteri vengono stampate.

```
subPalindrome(ITEM[] str, int n)
    ITEM[] sub = new ITEM[1...n]
    palRec(str, sub, n, 0)
```

```
palRec(ITEM[] str, boolean[] sub, int i, int j)
    if i == 0 then
        if isPalindrome(sub, 1, j) then
            print sub[1...j]
    else
        palRec(S, T, i − 1, j)
        sub[j + 1] = str[i]
        palRec(S, T, i − 1, j + 1)
```

```
boolean isPalindrome(ITEM[] sub, int i, int j)
    while i < j do
        if sub[i] == sub[j] then
            i = i + 1
            j = j - 1
        else
            return false
    return true
```

Esercizio B3 – Stringhe eleganti – Punti ≥ 12

È possibile risolvere il problema utilizzando la programmazione dinamica.
Definiamo $DP[i][b]$ come il numero di inversioni necessarie per rendere eleganti i primi i bit della stringa, con b come ultimo bit nella posizione i-esima. Il valore DP può essere calcolato attraverso la seguente relazione ricorsiva:

$$DP[i][b] = \begin{cases} 0 & \text{se } i = 0 \\ DP[i-1][0] + S[i] & \text{se } i > 0 \wedge b = 0 \\ \min\{DP[i-1][0], DP[i-1][1]\} + (1 - S[i]) & \text{se } i > 0 \wedge b = 1 \end{cases}$$

Come caso base utilizziamo $i = 0$ (stringa vuota), in cui il numero di caratteri da invertire è pari a zero. Altrimenti, consideriamo l'ultimo bit e distinguiamo due casi, in base al valore di b:

- Se $b = 0$, il bit precedente deve essere necessariamente 0. Restituiamo quindi il numero di bit per rendere eleganti i primi $i - 1$ bit della stringa, con 0 come ultimo bit, e sommiamo $S[i]$. Questo perché, se $S[i] = 1$, dobbiamo aggiungere 1 inversione per trasformarlo in zero; se $S[i] = 0$, dobbiamo sommare 0 inversioni.
- Se $b = 1$, il bit precedente può essere sia 0 che 1. Restituiamo quindi il minimo fra $DP[i-1][0]$ e $DP[i-1][1]$, e sommiamo $1 - S[i]$. Questo perché, se $S[i] = 0$, dobbiamo aggiungere 1 inversione per trasformarlo in zero; se $S[i] = 1$, dobbiamo sommare 0 inversioni.

Il valore finale potrebbe trovarsi sia in $DP[n][0]$ che in $DP[n][1]$, poiché a volte potrebbe essere più conveniente trasformare la stringa in tutti 0, altre volte in una combinazione di 0 ed 1. Restituiamo quindi il minimo fra i due.
L'algoritmo è il seguente:

```
int minInversions(ITEM[] S, int n)
    int[][] DP = new int[0...n][0...1] = {0}
    for i = 1 to n do
        DP[i][0] = DP[i-1][0] + S[i]
        DP[i][1] = min(DP[i-1][0], DP[i-1][1]) + (1 - S[i])
    return min(DP[n][0], DP[n][1])
```

La complessità è $O(n)$, derivante dal ciclo **for**.

Una soluzione basata su backtrack Una possibile soluzione alternativa consiste nel generare tutte le possibili inversioni, che sono 2^n (ogni bit può essere invertito oppure no) e quindi verificare se la stringa di bit risultante è elegante oppure no; nel caso, si aggiorna una variabile globale che contiene il numero minimo di inversioni incontrate finora. Il costo di questo algoritmo è $\Theta(n2^n)$. Una versione più sofisticata fa anche pruning, cioè dopo aver generato un 1, genera solo 1 successivamente. Questa soluzione non la scrivo perché è abbastanza complicata.

Una soluzione $\Theta(n^2)$ Una possibilità a cui non avevo pensato, suggerita da Luca Demattè, consiste nello scorrere il vettore con un indice i che va da 1 a n, e considerare quell'indice il punto in cui inizia la parte composta da soli 1. In questo caso, tutti i bit 1 nel sottovettore $S[1 \dots i-1]$ vanno portati a 0 e tutti i bit 0 nel sottovettore $S[i \dots n]$ vanno portati a 1. Si prende quindi il numero di bit da cambiare fra tutti gli indici i che sono compresi fra 1 e n. Si considera quindi il minimo numero di bit da cambiare fra tutti gli indici possibili.
La complessità temporale della funzione è $\Theta(n^2)$. Questo perché ci sono due cicli annidati che scorrono il vettore di input: il primo ciclo **for** scorre tutti gli elementi del vettore, e per ciascun elemento, il secondo e il terzo ciclo **for** scorrono ancora gli elementi del vettore. Quindi, il numero totale di operazioni eseguite è proporzionale a n^2.

```
int minInversions(ITEM[] S, int n)
    int minSoFar = +∞
    for i = 1 to n do
        int count_1 = 0
        for j = 1 to i − 1 do
            if S[j] == 1 then
                count_1 = count_1 + 1
        for j = i to n do
            if S[j] == 0 then
                count_0 = count_0 + 1
        minSoFar = min(minSoFar, count_1 + count_0)
    return minSoFar
```

Una soluzione $\Theta(n)$ alternativa A partire dalla soluzione precedente, è possibile pensare ad una soluzione alternativa con costo computazionale $\Theta(n)$, suggerita in forma leggermente diversa da Michele Gementi e Kevin Delugan.
L'idea è basata sul concetto di "prefix sum" (somma dei prefissi). In questo caso, creiamo un vettore $count_1$ di dimensione n, dove $count_1[i]$ contiene il numero di 1 nel sottovettore S[1 ... i]. Una volta che abbiamo questo vettore, possiamo utilizzarlo per calcolare rapidamente il numero di 0 in qualsiasi sottovettore $S[i \dots n]$, senza dover scorrere l'intero vettore ad ogni passo. Precisamente, il numero di 0 in $S[i \dots n]$ è dato da $(n - i - 1) - (count_1[n] - count_1[i - 1])$, dove $(n - i - 1)$ è il numero di bit in $S[i \dots n]$ e $(count_1[n] - count_1[i - 1])$ è il numero di bit 1 in $S[i \dots n]$, ottenuto sottraendo il numero di bit 1 in $S[1 \dots i - 1]$ dal numero totale di bit 1 $S[i \dots n]$ e $count_1[n] - count_1[i - 1]$. Sottraendo il secondo dal primo, possiamo ottenere il numero di bit 1 in $S[1 \dots i - 1]$.

```
int minInversions(ITEM[] S, int n)
    int[] count_1 = new int[1 ... n]
    count_1[i] = S[i]
    for i = 2 to n do
        count_1[i] = count_1[i − 1] + S[i]
    int minSoFar = +∞
    for i = 1 to n do
        minSoFar = min(minSoFar, count_1[i − 1] + (n − i − 1) − (count_1[n] − count_1[i − 1]))
    return minSoFar
```

Una soluzione errata Ho visto diverse varianti di algoritmi che potremmo definire greedy e non funzionano. Ne mostro uno qui:

```
int minInversions(ITEM[] S, int n)
    int left = 1
    int right = n
    while left ≤ n and S[left] == 0 do
        left = left + 1
    while right ≥ 1 and S[right] == 0 do
        right = right − 1
    int zeroInversions = 0
    int oneInversions = 0
    for i = left to right do
        if S[i] == 0 then
            zeroInversions = zeroInversions + 1
        else
            oneInversions = oneInversions + 1
    return min(zeroInversions, oneInversions)
```

L'algoritmo individua il primo bit 1 da sinistra e il primo bit zero da destra (ignorando i bit 0 iniziali e i bit 1 finale, che vanno bene come sono), e conta le inversioni necessarie per trasformare tutti i bit zero a uno e le inversioni necessarie per trasformare tutti i bit uno a zero, restituendo il massimo fra questi due valori.
L'algoritmo non è corretto: nella stringa `10001110`, l'algoritmo restituisce 4, mentre è sufficiente invertire il primo e l'ultimo bit.