Algoritmi e Strutture Dati – 09/02/23

Esercizio A1 – Complessità – Punti ≥ 8

L'algoritmo ricorsivo contiene un ciclo **while** che verifica se i valori del sottovettore considerato sono "palindromi", ovvero se gli estremi sono uguali; finché sono uguali, li somma alla variabile *tot*.
Si consideri il caso ottimo in cui tutti i valori del vettore sono distinti. In questo caso, il ciclo **while** termina subito, alla prima verifica di condizione, e l'equazione di ricorrenza è la seguente:

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

Applicando il Master Theorem, otteniamo $\alpha = \log_2 2 = 1$ e $\beta = 0$; quindi l'algoritmo è $\Omega(n)$.
Nel caso pessimo, tutti i valori del vettore sono uguali. Il ciclo **while** esamina tutti i valori del sottovettore, con un costo pari a $O(n)$. In questo caso, l'equazione di ricorrenza è la seguente:

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

Applicando il Master Theorem, otteniamo $\alpha = \log_2 2 = 1$ e $\beta = 1$; quindi l'algoritmo è $O(n \log n)$. Per semplicità, abbiamo assunto che n sia una potenza di 2.

Esercizio A2 – Broadcast – Punti ≥ 10

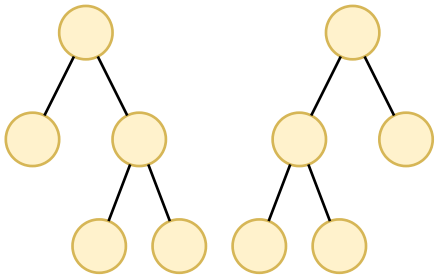
È possibile risolvere il problema utilizzando la tecnica del divide-et-impera. Si consideri la radice T di un qualunque sottoalbero.

- Se T non ha figli, il numero di turni è zero (la radice è già al corrente).
- Se T ha un figlio, il numero di turni è pari al numero di turni necessari per diffondere il messaggio nel sottoalbero radicato nel figlio di T, $+1$ per spedire il messaggio al figlio.
- Se T ha due figli, si calcola il numero di turni necessari per diffondere il messaggio nel sottoalbero sinistro (n_L) e nel sottoalbero destro (n_R). Se $n_L \neq n_R$, saranno necessari un turni per informare il figlio che richiede una quantità maggiore di turni (da cui $+1$); poi parallelamente si informerà l'altro figlio, che richiede un numero di turni inferiore (di 1 turno o più) e quindi completerà la diffusione nello stesso numero di turni dell'altro o meno. Se $n_L = n_R$, i due sottoalberi richiederanno la stessa quantità di tempo, ma uno dei due verrà informato dopo due turni (da cui $+2$).

```
minRounds(Tree T)
if T.left == nil and T.right == nil then
    return 0
else if T.left ≠ nil and T.right == nil then
    return 1 + minRounds(T.left)
else if T.left == nil and T.right ≠ nil then
    return 1 + minRounds(T.right)
else
    int n_L = minRounds(T.left)
    int n_R = minRounds(T.right)
    return max(n_L, n_R) + iif(n_L == N_R, 2, 1)
```

Il costo dell'algoritmo è lineare nel numero di nodi, perché effettua una visita in profondità.

Soluzioni errate : molte soluzioni non hanno considerato che non è possibile scegliere di mandare il messaggio prima a sinistra e poi a destra "a priori", senza capire quandi turni sono necessari per diffondere il messaggio nei due sottaalberi. Questa soluzioni sono basate su BFS o DFS; le soluzioni DFS sono facili da scrivere (manda a sinistra, manda a destra, prendi il massimo fra i due), mentre le soluzioni BFS devono fare i salti mortali per cercare di conteggiare i round necessari; alcune sono anche parzialmente corrette, nel senso che contano effettivamente i turni necessari per una soluzione che manda il messaggio sempre prima a sinistra. Tuttavia, questa strategia "fissa" non è corretta.
Nell'albero seguente a sinistra, per esempio, saranno necessari 4 turni per diffondere il messaggio, se si manda il messaggio prima nel sottoalbero sinistro della radice; mentre saranno sufficienti 3 turni nel caso il messaggio sia mandato prima nel sottoalbero di destra. Nell'albero simmetrico a destra, la situazione è invertita. Un algoritmo che adotta una strategia fissa (prima da un lato, poi dall'altro) non restituirà il valore corretto in uno di questi casi.



Esercizio A3 – Minimo locale – Punti ≥ 12

Prima di iniziare, dimostriamo la proprietà data per vera nel testo.
Sia $A[i \dots j]$ un vettore contenente $n \geq 3$ valori distinti, tale per cui $A[i] > A[i+1]$ e $A[j-1] < A[j]$. Vogliamo dimostrare che $A[i \dots j]$ contiene un minimo locale.

- **Caso base**: Se $n = 3, j = i+2$, $A[i] > A[i+1] < A[i+2]$ e quindi $A[2]$ è un minimo locale.
- **Ipotesi induttiva**: Assumiamo che tutti i vettori di dimensione $3 \leq n' < n$ tali che $A[i] < A[i+1]$ e $A[j-1] < A[j]$ contengano un minimo locale.
- **Passo induttivo**: Sia m l'elemento centrale. Se $A[m-1] > A[m] < A[m+1]$, allora $A[m]$ è un minimo locale. Altrimenti, sono possibili due casi:
  - se $A[m-1] < A[m]$, allora $A[i \dots m]$ è un sottovettore di dimensione inferiore a n tale che $A[i] > A[i+1]$ e $A[j-1] < A[j]$, quindi contiene un minimo locale. Bisogna fare attenzione a un caso particolare, tuttavia: nel caso $n = 4$, m corrisponde a $i+1$ ($m = \lfloor (i+i+3)/2 \rfloor = \lfloor i + 3/2 \rfloor = i+1$). Se questo caso fosse possibile, ci ridurremmo ad un vettore di dimensione 2, che non rispetta le condizioni di partenza. Tuttavia, sappiamo che $A[i] > A[i+1]$ è equivalente a dire che $A[m-1] > A[m]$, quindi siamo necessariamente nell'altro caso.
  - se $A[m] > A[m+1]$, allora $A[m \dots j]$ è un sottovettore di dimensione inferiore a n e superiore o uguale a 3 tale che $A[m] > A[m+1]$ e $A[j-1] < A[j]$, quindi contiene un minimo locale.

Si può utilizzare la dimostrazione per realizzare un algoritmo basato su divide-et-impera.

```
int findLocalMinimum(int[] A, int n)
    return flmRec(A, 1, n)
```

```
int flmRec(int[] A, int i, int j)
    if j == i + 2 then
        return A[i + 1]
    else
        int m = ⌊(i + k)/2⌋
        if A[m − 1] > A[m] and A[m] < A[m + 1] then
            return m
        else if A[m − 1] < A[m] then
            return flmRec(A, 1, m)
        else
            return flmRec(A, m, j)
```

Si noti che il caso base è stato esplicitato, ma non sarebbe necessario: se ci sono tre valori, viene calcolato il mediano e la prima condizione dell'if interno è vera, quindi viene restituito m.
La complessità è ovviamente logaritmica, in quanto ricerca dicotomica.

Esercizio B1 – MCS – Punti ≥ 8

Il problema è una semplice variante di LCS, in cui invece di aggiungere +1 tutte le volte che si seleziona un carattere perché uguale da entrambe le parti, si aggiunge $T[i]$, se questo valore è positivo. Altrimenti, si sceglie il massimo fra $DP[i-1][j]$ e $DP[i][j-1]$, come in LCS.
La formula ricorsiva è la seguente:

$$DP[i][j] = \begin{cases} 0 & i \leq 0 \ \vee j \leq 0 \\ DP[i-1][j-1] + T[i] & i > 0 \wedge j > 0 \wedge T[i] = U[j] \wedge T[i] > 0 \\ max\{DP[i-1][j], DP[i][j-1]\} & i > 0 \wedge j > 0 \wedge T[i] \neq U[j] \end{cases}$$

Il codice per risolvere il problema è il seguente. Nel codice abbiamo gestito il caso in cui due valori negativi sono uguali,

scartandoli e sommando 0.

```
int mcs(Item[] T, Item[] U, int n, int m)
    int[][] DP = new int[0...n][0...m] = {0}
    for i = 1 to n do
        for j = 1 to m do
            if T[i] == U[j] and T[i] > 0 then
                DP[i][j] = DP[i-1][j-1][T[i]] + iif(T[i] > 0, T[i], 0)
            else
                DP[i][j] = max(DP[i-1][j], DP[i][j-1])
    return DP[n][m]
```

La complessità è $\Theta(nm)$, a causa dei due cicli **for**.

Alcune soluzioni proposte costruivano nuovi vettori, scartando (giustamente) tutti i valori negativi. Quest'approccio è leggermente preferibile, perché riduce la dimensione dell'input e quindi della tabella DP, ma in assenza di valori negativi la complessità è la stessa.

Esercizio B2 – Facoltiadi – Punti ≥ 10

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

- una sorgente s;
- un nodo per ogni studente, $p_1 \dots p_n$;
- tre nodi per ogni squadra j, $1 \leq j \leq 14$, chiamati s_j^M, s_j^F, s_j^X; s_j^M serve a raccogliere 7 studenti, s_j^F serve a raccogliere 7 studentesse, s^X serve a raccogliere studenti di qualunque genere, incluso "preferisco non rispondere";
- un pozzo t.

Creiamo i seguenti archi:

- colleghiamo s a ogni studente, senza distinzioni di genere, con peso 1 (in quanto può partecipare a una sola squadra);
- colleghiamo ogni studente come segue:
  - colleghiamo ogni studente maschio i ai nodi $s_{d(i)}^M$, $s_{d'(i)}^M$, con peso 1;
  - colleghiamo ogni studentessa femmina i ai nodi $s_{d(i)}^F$, $s_{d'(i)}^F$, con peso 1;
  - colleghiamo ogni studente, senza distinzioni di genere, ai nodi $s_{d(i)}^X$, $c_{d'(i)}^X$, con peso 1;

  in totale, gli archi di queste tre categorie sono $\leq 4n$, perché ogni studente può essere connesso a due dipartimenti come appartenente al suo genere e come studente generico;
- colleghiamo tutti i nodi s_j^M, s_j^F, s_j^X al pozzo, con peso 7 (42 archi).

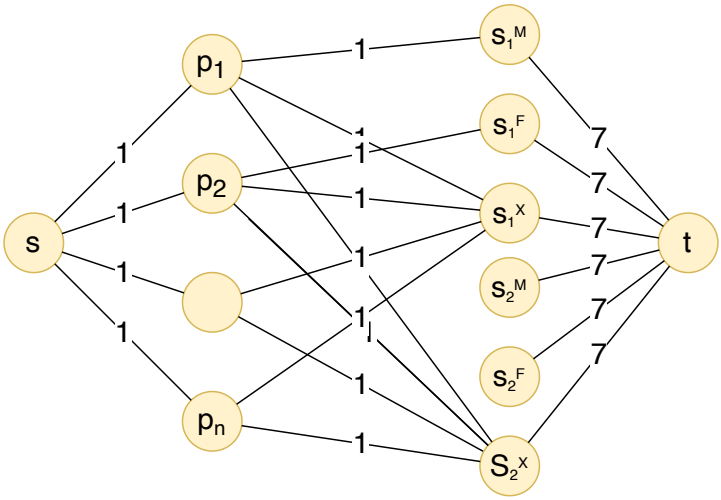
L'idea è la seguente: riusciamo a formare le squadre se riusciamo a selezionare 7 maschi, 7 femmine e altri 7 studenti di qualunque genere (anche non determinato). Ogni studente può partecipare a una sola di queste categorie, perché riceve un valore 1 dalla sorgente. Se il valore totale è pari a 294, è stato possibile fare un'assegnamento correto.
La dimensione di V ed E è la seguente:

$$|V| = 2 + n + 42 = n + 44$$
$$|E| \leq n + 4n + 42 = 5n + 42$$

Utilizzando il limite di Ford-Fulkerson e tenuto conto che il flusso massimo è limitato superiormente da 294, il costo totale dell'algoritmo proposto è

$$O(294 \cdot (|V| + |E|) = O(294 \cdot (6n + 86)) = O(n)$$

La figura seguente mostra un piccolo esempio con n studenti e due squadre.



Esercizio B3 – k-prodotto – Punti ≥ 12

È possibile risolvere il problema tramite programmazione dinamica. Sia $DP[i][k]$ il numero di sottosequenze contenute nel prefisso $A[1 \dots i]$ con prodotto inferiore o uguale a $k > 0$. Tale valore può essere calcolato ricorsivamente nel modo seguente:

$$DP[i][k] = \begin{cases} 1 & i = 0 \\ DP[i-1][k] & i > 0 \wedge A[k] > k \\ DP[i-1][k] + DP[i-1][\lfloor k/A[i] \rfloor] & i > 0 \end{cases}$$

La spiegazione è la seguente:

- se $i = 0$, la sottosequenza è vuota e contiene una sottosequenza (se stessa) con prodotto inferiore a k;
- altrimenti, se $A[i] > k$, il valore $A[i]$ deve essere ignorato e restituiamo $DP[i-1][k]$
- altrimenti, consideriamo due possibilità: se non prendiamo $A[i]$, ci sono $DP[i-1][k]$ sottosequenze; se prendiamo $A[i]$, ci sono $DP[i-1][\lfloor k/A[i] \rfloor]$ sequenze. Questi due insiemi di sottosequenze vanno sommate insieme.

Utilizziamo memoization per tradurre la formula in codice:

```
int productK(int[]A, int n, int m)
    int[][] DP = newint[1...n][1...m] = {−1}
    return pkRec(A, DP, n, m)
```

```
int pkRec(int[]A, int[][] DP, int i, int k)
    if i == 0 then
        return 1
    else
        if DP[i][k] < 0 then
            DP[i][k] = pkRec(DP, i, k)
            if A[i] ≤ k then
                DP[i][k] = DP[i][k] + pkRec(DP, i − 1, [⌊k/A[i]⌋])
        return DP[i][k]
```

La complessità è pari a $O(nm)$, il costo di riempire l'intera tabella.