**Cognome**: **Nome**: **# Matricola**: **Riga**: **Col**:

# *Algoritmi e Strutture Dati – 23/08/2023 – Parte A*

## A1 – Complessità – Punti ≥ 8

Si consideri questo algoritmo:

```
int P(int[][]A, int n)
    return R(A, 1, 1, n, n)
```

```
int R(int[][]A, int x1, int y1, int x2, int y2)
    int tot = 0
    for x = x1 to x2 do
        for y = y1 to y2 do
            tot = tot + A[x][y]
    if x1 ≠ x2 or y1 ≠ y2 then
        int xm = ⌊(x2 + x1)/2⌋
        int ym = ⌊(y2 + y1)/2⌋
        tot = tot + R(A, x1, y1, xm, ym)
        tot = tot + R(A, xm + 1, ym + 1, x2, y2)
    return tot
```

Scrivere l'equazione di ricorrenza associata a questo algoritmo e calcolare la complessità computazionale che ne deriva. Per semplificarvi la vita, assumete che la dimensione $n$ sia una potenza di 2.

## A2 – Diretto o no? – Punti ≥ 10

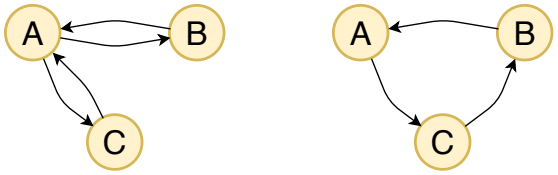
Scrivere un algoritmo

**boolean** isUndirected(GRAPH $G$)

che prenda in input un grafo $G$ rappresentato tramite liste o vettori di adiacenza, e restituisca **true** se $G$ rappresenta un grafo non orientato, cioè per ogni arco $(x, y)$ esiste anche l'arco $(y, x)$, **false** altrimenti.

Discutere informalmente la correttezza dell'algoritmo e la sua complessità computazionale.

Per esempio, l'algoritmo deve restituire **true** per il grafo seguente a sinistra, **false** per il grafo seguente a destra.




## A3 – Prossimità – Punti ≥ 12

In un albero binario, la *prossimità* fra due nodi $t_1$ e $t_2$ appartenenti allo stesso livello $\ell$ è pari al numero di nodi del livello $\ell$ che sarebbero compresi fra $t_1$ e $t_2$ in albero binario perfetto con la stessa altezza; se $t_1$ e $t_2$ non appartengono allo stesso livello, la loro prossimità è pari a $+\infty$.

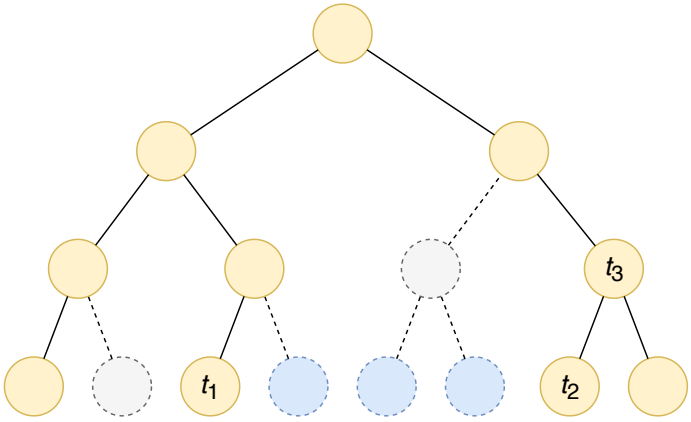
Scrivere un algoritmo

**int** proximity(TREE $T$, TREE $t_1$, TREE $t_2$)

che prenda in input l'albero $T$ e due suoi nodi $t_1, t_2$ e restituisca la prossimità fra $t_1$ e $t_2$ in $T$.

Discutere informalmente la correttezza della soluzione proposta e calcolare la complessità computazionale. Per semplicità, assumete che $t_1$ e $t_2$ siano effettivamente due nodi distinti appartenenti all'albero.

Nell'albero seguente, i nodi ocra rappresentano $T$, mentre i nodi tratteggiati rappresentano i nodi mancanti dell'albero perfetto. La prossimità fra $t_1$ e $t_2$ è pari a 3 (i nodi in azzurro), mentre la prossimità fra $t_1$ e $t_3$ è $+\infty$.



**Cognome**: **Nome**: **# Matricola**: **Riga**: **Col**:

# *Algoritmi e Strutture Dati – 23/08/2023 – Parte B*

## B1 – Rete idrica – Punti $\geq 8$

Siete responsabili di una rete idrica descritta da un grafo $G = (V, E)$. Il sottoinsieme $V_{in} \subseteq V$ contiene i nodi che sono sorgenti di acqua, e ogni nodo $v \in V_{in}$ produce $in(v)$ litri di acqua al secondo. Il sottoinsieme $V_{out} \subseteq V$ contiene i nodi che sono consumatori di acqua, con ogni nodo $v \in V_{out}$ che consuma $out(v)$ litri di acqua al secondo. Questi sottoinsiemi sono disgiunti: $V_{in} \cap V_{out} = \emptyset$. Gli altri nodi in $V$ sono semplici distributori.
I nodi sono collegati da tubazioni descritte dall'insieme $E$; la tubazione che collega un nodo $u$ ad un nodo $v$ ha una portata di $c(u, v)$ litri al secondo. Sfortunatamente, esiste un sottoinsieme $E_{\text{per}} \subseteq E$ di tubazioni che devono essere sostituite.
Per sostituire una tubazione, è necessario chiuderla; quando è chiusa, l'acqua non può più fluire attraverso di essa. Potreste domandarvi cosa succederebbe alla rete se chiudessimo una tubazione alla volta per ripararla. È possibile che alcuni nodi consumatori non ricevano abbastanza acqua?
Descrivere un algoritmo che prenda in input un grafo $G = (V, E)$, gli insiemi $V_{in}$, $V_{out}$ e $E_{\text{per}}$, e restituisca **true** se chiudere una qualsiasi delle tubazioni guaste produce una riduzione di servizio per uno o più consumatori d'acqua, **false** altrimenti.

## B2 – Sub-palindrome – Punti $\geq 10$

Scrivere un algoritmo

$$\text{subPalindrome}(\textsc{Item}[\,]\ S,\ \textbf{int}\ n)$$

che prenda in input una stringa $S$ e stampi tutte le sottosequenze palindrome di $S$.

Discutere informalmente la correttezza dell'algoritmo e la sua complessità computazionale; se possibile, discutere un limite inferiore e superiore alla complessità.

Per esempio, la parola `"casca"` contiene le seguenti sottosequenze palindrome:
`""`, `"c"`, `"a"`, `"s"`, `"c"`, `"a"`, `"cc"`, `"aa"`, `"cac"`, , `"csc"`, `"asa"`, `"aca"`

## B3 – Stringhe eleganti – Punti $\geq 12$

Una stringa binaria è detta *elegante* se è composta da un certo numero di 0 (possibilmente nessuno), seguito da un certo numero di 1 (anche qui, possibilmente nessuno).
Un'*inversione* di un bit della stringa consiste nel cambiare tale bit da 0 a 1 o da 1 a 0.
Scrivere un algoritmo

$$\textbf{int}\ \text{minInversions}(\textsc{Item}[\,]\ S,\ \textbf{int}\ n)$$

che prenda in input una stringa $S$ di dimensione $n$ e restituisca il numero minimo di inversioni necessarie per renderla elegante.

Discutere informalmente la correttezza dell'algoritmo e la sua complessità computazionale.

Per esempio,

- `0000`1`0000` può essere trasformata nella stringa elegante `0000`0`0000` con 1 inversione (ma non con 0), quindi l'algoritmo deve restituire 1;
- `00111`00`11` può essere trasformata nella stringa elegante `00111`11`11` con 2 inversioni (ma non con 0 o 1), quindi l'algoritmo deve restituire 2.
- `111111111` e `000011111` sono già eleganti, quindi l'algoritmo deve restituire 0.