

Grafi – Stessa distanza

- In un grafo orientato G , dati due nodi s e v , si dice che:
 - v è **raggiungibile** da s se esiste un cammino da s a v ;
 - la **distanza** di v da s è **la lunghezza del più breve cammino** da s a v (misurato in numero di archi), oppure $+\infty$ se v non è raggiungibile da s
- Scrivere un algoritmo che prenda in input un grafo orientato $G = (V, E)$ e due nodi $s_1, s_2 \in V$, che restituisca il numero di nodi in V tali che:
 - siano raggiungibili sia da s_1 che da s_2 , e
 - si trovino alla stessa distanza da s_1 e da s_2 .
- Discutere la complessità dell'algoritmo proposto.

Grafi – Grafi bipartiti

- Un grafo non orientato G è **bipartito** se l'insieme dei nodi può essere partizionato in due sottoinsiemi disgiunti tali che nessun arco del grafo connette due nodi appartenenti allo stesso sottoinsieme.
- $G = (V, E)$ è **2-colorabile** se è possibile trovare una **2-colorazione** di esso, ovvero un **assegnamento** $c[u] \in C$ per ogni nodo $u \in V$, dove C è un insieme di "colori" di dimensione 2, tale che:
$$(u, v) \in E \Rightarrow c(u) \neq c(v)$$
- Si dimostri che G è bipartito:
 - se e solo se è 2-colorabile
 - se e solo se non contiene cicli di lunghezza dispari
- Scrivere un algoritmo che prenda in input un grafo bipartito G e restituisca una 2-colorazione di G sull'insieme di colori $C = \{0, 1\}$, espressa come un vettore $c[1 \dots n]$. Discuterne la complessità.

SortinoSort

Il professor Sortino ha inventato un nuovo algoritmo di ordinamento.

- Il vettore di input viene diviso in tre parti, di dimensioni circa $n/3$.
- Vengono ordinati ricorsivamente i primi due terzi, i secondi due terzi, e infine di nuovo i primi due terzi.

```
SortinoSort(int[] A, int i, int j)
```

```
if  $j - i + 1 \leq 6$  then
```

```
    InsertionSort(A, i, j)
```

```
else
```

```
    int  $s = \lceil (j - i + 1) / 3 \rceil$ 
```

```
    SortinoSort(A, i, i + 2s - 1)
```

```
    SortinoSort(A, i + s, j)
```

```
    SortinoSort(A, i, i + 2s - 1)
```

SortinoSort

- ① Qual è la complessità di questo algoritmo? Il Prof. Sortino finirà nella prossima edizione del mio libro?
- ② (Difficile, Opzionale) Dimostrare per induzione che questo algoritmo è corretto. Per comodità, assumete pure che tutti i valori siano distinti.

Spoiler alert!

Stessa distanza

```
int sameDistance(GRAPH  $G$ , NODE  $s_1$ , NODE  $s_2$ )
```

```
int  $dist_1$  = new int[1... $G.n$ ]  
int  $dist_2$  = new int[1... $G.n$ ]  
distance( $G$ ,  $s_1$ ,  $dist_1$ )  
distance( $G$ ,  $s_2$ ,  $dist_2$ )  
int  $counter$  = 0  
foreach  $u \in G.V()$  do  
    if  $dist_1[u] \neq -1$  and  $dist_1[u] == dist_2[u]$  then  
         $counter = counter + 1$   
return  $counter$ 
```

Grafi – Grafi bipartiti

- Se G è bipartito, è 2-colorabile. Diamo colore 0 a tutti i nodi in una partizione, diamo colore 1 a tutti i nodi nell'altra. Non essendoci archi fra i nodi di una partizione, la colorazione è valida.
- Se G è 2-colorabile, non contiene cicli di lunghezza dispari.
Supponiamo per assurdo che esista un ciclo $(v_1, v_2), (v_2, v_3) \dots, (v_{k-1}, v_k), (v_k, v_1)$, con k dispari. Se il nodo v_1 ha colore 0, il nodo v_2 deve avere colore 1; il nodo v_3 deve avere colore 0, e così via fino al nodo v_k , che deve avere colore 0. Poichè v_1 è successore di v_k , v_1 deve avere colore 1, assurdo.

Grafi – Grafi bipartiti

- Se non esistono cicli di lunghezza dispari, il grafo è bipartito.

Dimostriamo questa affermazione costruttivamente. Si prenda un nodo x lo si assegna alla partizione S_1 . Si prendono poi tutti i nodi adiacenti a nodi in S_1 e li si assegna alla partizione S_2 . Si prendono tutti i nodi adiacenti a nodi in S_2 e li si assegna alla partizione S_1 . Questo processo termina quando tutti i nodi appartengono ad una o all'altra partizione. Un nodo può essere assegnato più di una volta se e solo se fa parte di un ciclo. Ma affinché venga assegnato a due colori diversi, deve far parte di un ciclo di lunghezza dispari, e questo non è possibile.

Grafi – Grafi bipartiti

Questa funzione ritorna un vettore di colori se il grafo è 2-colorabile, **nil** altrimenti.

```
int[] color(GRAPH G)
```

```
int[] colors = new int[1 ... G.n]  
for u ∈ G.V() do  
    | colors[u] = -1  
foreach u ∈ G.V() do  
    | if colors[u] < 0 then  
        | if not colorRec(G, u, colors, 0) then  
            | | return nil  
return colors
```

Grafi – Grafi bipartiti

Questa funzione ritorna **true** se il grafo è 2-colorabile, **false** altrimenti.

```
boolean colorRec(GRAPH G, NODE u, int[] colors, int color)
```

```
  colors[u] = color
```

```
foreach v ∈ G.adj(u) do
```

```
  if colors[v] < 0 then
```

```
    if colorRec(G, v, colors, 1 − color) == false then
```

```
      return false
```

```
  else if colors[v] == color then
```

```
    return false
```

```
return true
```

La complessità è rappresentata dalla seguente equazione di ricorrenza:

$$T(n) = \begin{cases} 3T(\frac{2}{3}n) + 1 & n > 6 \\ 1 & n \leq 6 \end{cases}$$

Utilizzando il teorema delle ricorrenze lineari con partizione bilanciata, si ottiene che $a = 3$, $b = 3/2$, da cui $\alpha = \log_{3/2} 3$; inoltre, $\beta = 0$. Siamo quindi nel caso $T(n) = n^\alpha$.

Non avete una calcolatrice e non sapete quanto sia $\log_{3/2} 3$?

La complessità è rappresentata dalla seguente equazione di ricorrenza:

$$T(n) = \begin{cases} 3T(\frac{2}{3}n) + 1 & n > 6 \\ 1 & n \leq 6 \end{cases}$$

Utilizzando il teorema delle ricorrenze lineari con partizione bilanciata, si ottiene che $a = 3$, $b = 3/2$, da cui $\alpha = \log_{3/2} 3$; inoltre, $\beta = 0$. Siamo quindi nel caso $T(n) = n^\alpha$.

Non avete una calcolatrice e non sapete quanto sia $\log_{3/2} 3$?

E' semplice: $(\frac{3}{2})^2 = \frac{9}{4} < 3$, mentre $(\frac{3}{2})^3 = \frac{27}{8} > 3$. Quindi α è compreso fra 2 e 3, e quindi questo algoritmo è addirittura peggiore di Insertion Sort. Il prof. Sortino non finirà nel mio libro.