Domande Orale Primo Modulo

Domande per l'orale del primo modulo di Analisi e Progettazione di Algoritmi (APA).

1. Complessità di problemi, problemi aperti e chiusi

Un problema può avere complessità $O(f(n)), \Omega(f(n))$ o $\Theta(f(n))$. Solitamente O è per indicare il caso peggiore (quindi il limite superiore), Ω per il caso migliore (quindi il limite inferiore) e Θ quando si conoscono il limite inferiore e superiore. Nel caso di algoritmi randomizzati si calcola la media pesata di tutti i tempi di esecuzione per uno stesso input e quello che interessa è il caso peggiore, quindi quello con tempo di esecuzione maggiore.

Un problema si dice **chiuso** quando esiste un algoritmo di complessità $O(f(n))$ e si è dimostrato che qualunque algoritmo risolvente ha complessità $\Omega(f(n))$, ossia non può esistere un algoritmo di complessità inferiore a $\Omega(f(n))$ (in altre parole $O(f(n)) = \Omega(f(n))$), dimostrando così che l'algoritmo è **ottimo** con possibili miglioramenti marginali.

Un problema si dice **aperto** quando il miglior algoritmo risolvente noto è $O(f(n))$ e si è dimostrato che qualunque algoritmo risolvente deve avere complessità $\Omega(g(n))$ con $g \neq f$. In altri termini, si sa risolvere il problema in un tempo $f(n)$ e si sa che non lo si può risolvere in un tempo migliore $g(n)$, dove g 'cresce meno' di f (in altre parole $O(f(n)) \neq \Omega(g(n))$).

Un problema potrebbe avere anche un **gap algoritmico** che può essere chiuso 'dal di sopra' trovando un algoritmo migliore che, se coincide con il limite inferiore, rende l'algoritmo chiuso e ottimo oppure 'dal di sotto' dimostrando che esiste un limite inferiore più alto, similmente a prima se questo coincide con il limite superiore l'algoritmo è chiuso e ottimo.

2. Algoritmo di Dijkstra

Problema: dato un grafo orientato pesato G, con pesi non negativi e dati un nodo di partenza s e un nodo di arrivo t, trovare un cammino minimo tra s e t.

```
Dijkstra(G,s)
    for each (u nodo in G) dist[u] = +inf
    parent[s] = null; dist[s] = 0
    Q = heap vuoto
    for each (u nodo in G) Q.add(u,dist[u])
    while(Q non vuota)
        u = Q.getMin() // nodo u nero
        for each((u,v) arco in G)
            if dist[u] + peso(u,v) < dist[v]
                dist[v] = dist[u] + peso(u,v); parent[v] = u
                Q.changePriority(v,dist[v])
    return dist, parent
```

A parole: si inizializzano i pesi dei nodi a ∞ tranne che per il nodo di partenza che si inizializza a 0. Si crea un heap con i nodi e i loro pesi e, fino a quando l'heap non è vuoto, si estrae il nodo grigio con peso minore e si marca come visitato (nero). Per ogni arco uscente dal nodo si controlla se il peso del nodo di partenza più il peso dell'arco è minore del peso "registrato" nell'heap, in tal caso si aggiorna il peso e si cambia la priorità nell'heap e si marca il nodo v come grigio.

NB: in un heap il padre ha priorità minore dei figli, quindi il nodo con peso minore è in cima all'heap.
Complessità:

- Inizializzazione nodi bianchi: $O(n)$
- n estrazioni da heap: $O(n \log n)$
- ciclo interno dove ogni arco viene percorso una volta e per ogni nodo adiacente si ha eventuale cambio di priorità: $O(m \log n)$

Complessivamente $O(n \log n + m \log n) = O((n + m) \log n)$, nel caso di grafo denso $m = n^2 \rightarrow O(n^2 \log n)$

3. Definizione di minimo albero ricoprente, algoritmi di Prim e Kruskal

Un **minimo albero ricoprente** di G è un albero ricoprente (che contiene tutti i nodi del grafo ed è aciclico) di G in cui la somma degli archi è minima.

```
Prim(G,s)
    for each (u nodo in G) marca u come non visitato
    for each (u nodo in G) dist[u] = inf
    parent[s] = null ; dist[s] = 0
    Q = heap vuoto
    for each (u nodo in G) Q.add(u,dist[u])
    while (Q non vuoto)
        u = Q.getMin()
        marca u come visitato (nero)
        for each((u,v) arco in G)
            if v non visitato e peso(u,v) < dist[v]
                dist[v] = peso(u,v); parent[v] = u
                Q.changePriority(v,dist[v])
```

Complessità: uguale a Dijkstra, $O((n + m) \log n)$

```
Kruskal(G)
    s = sequenza di archi di G in ordine di costo crescente
    T = foresta formata dai nodi di G e nessun arco
    counter = 0
    while(counter < n-1)
        estrai da s il primo elemento (u,v)
        if (u,v non connessi in T) T = T + (u,v) // aggiungi arco
        counter++
    return T
```

Complessità: il problema è controllare se due nodi sono già connessi. Farlo in modo banale con una visita dei due alberi richiede $O(n)$ nel caso peggiore, quindi si ha $O(n \cdot m)$.

```
KruskalUF(G)
    s = sequenza di archi di G in ordine di costo crescente
    T = foresta formata dai nodi di G e nessun arco
    UF = struttura union-find vuota
    for each (u nodo in G) UF.makeSet(u)
    while (s non vuota)
        estrai da s il primo elemento (u,v)
        if(UF.union_by_need(u,v))
        // esegue la union delle radici se UF.find(u) != UF.find(v)
            T = T + (u,v)
    return T
```

Complessità: $O(m \log m)$

4. Definizione di ordinamento topologico, i due algoritmi per calcolarlo

Un ordinamento topologico di un grafo orientato aciclico (DAG) $G = (V, E)$ è un ordine totale stretto su V tale che se $(u, v) \in E$ allora u precede v nell'ordinamento ($u < v$).

```
topologicalsort(G)
    S = insieme vuoto
    Ord = sequenza vuota
    for each (u nodo in G) indegree[u] = indegree di u
    for each (u nodo in G)
        if indegree[u] == 0 S.add(u)
    while (S non vuoto)
        u = S.remove()
        Ord.add(u) // in fondo
        for each ((u,v) arco in G)
            indegree[v]--
            if indegree[v] == 0 S.add(v)
    return Ord
```

Complessità: $O(n + m)$

```
DFS(G)
    for each (u nodo in G) marca u come bianco ; parent[u] = null
    time = 0
    for each (u nodo in G)
        if u bianco DFS-Visit(G,u)

DFS-Visit(G,u,T)
    time++; start[u] = time
    visita u ; marca u come grigio
    for each ((u,v) arco in G)
        if v bianco
            parent[v] = u
            DFS-Visit(G,v)
    time++; end[u] = time
```

Complessità: $O(n + m)$

5. Definizione di componenti fortemente connesse, l'algoritmo per calcolarle

6. Caratteristiche della programmazione dinamica, problema LCS e algoritmo per risolverlo, algoritmo di Floyd-Warshall

7. Definizioni di problema di decisione, astratto e concreto, algoritmo di verifica, classi P, NP e NP-C

8. Nozione di riduzione polinomiale, definizione di classe NP-C, problema P-NP

9. Esempi di problemi NP-completi e riduzioni (SAT, 3SAT, CLIQUE)