Domande Orale Primo Modulo: 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 $m = n^2 \to O(n^2 \log n)$

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

In un grafo orientato G, due nodi u e v si dicono mutualmente raggiungibili, o **fortemente connessi**, se ognuno dei due è raggiungibile dall'altro, ossia se esistono un cammino da u a v e un cammino da v a u. Una **componente fortemente connessa** è un sottografo di G in cui i nodi sono tutti fortemente connessi tra loro.

```
SCC(G)
    DFS(G, Ord)
    GT = grafo trasposto di G
    OrdFC = sequenza vuota // componenti fortemente connesse
    while(Ord non vuota)
        u = ultimo nodo visitato in Ord
        C = insieme di nodi vuoto
        DFS(GT,u,C)
        OrdFC.add(C)
    return OrdFC
```

Complessità: $O(n + m)$

```
CFC(G)
    Ord = DFS(G) // con i time-stamp (solo questa)
```

```
        GT = grafo trasposto di G
        OrdFC = sequenza vuota // componenti fortemente connesse
        while(Ord non vuota)
            u = primo nodo non visitato in Ord
            C = DFS(GT,u)
            OrdFC.add(C)
        return OrdFC
```

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

La programmazione dinamica è vantaggiosa se un sottoproblema viene usato più volte, si basa su def. ricorsiva come divide-et-impera e la correttezza per induzione forte, spesso bottom-up (memorizzazione dei risultati dei sottoproblemi).
LCS problema: date due sequenze trovare una sottosequenza comune di lunghezza massima.
Si costruisce una matrice LCS con m+1 righe e n+1 colonne. La prima riga e la prima colonna sono inizializzate a 0. Si può procedere riga per riga o colonna per colonna, l'ultima casella (angolo in basso a destra) conterrà la soluzione. Se si ha lo stesso carattere sulla riga e sulla colonna, una "freccia diagonale", aumentando di 1 la lunghezza rispetto alla casella puntata dalla freccia; se sulla riga e sulla colonna ci sono due caratteri diversi, una freccia verso la casella con il valore maggiore tra la casella sopra e la casella a sinistra (se i valori sono uguali punto sempre sopra).
Per l'algoritmo abbiamo la matrice L per le lunghezze e la matrice R dei riferimenti, X e Y sono le sequenze.

```
    LCS(L,R,X,Y)
        for(i=0;i<=m;i++) L[i][0] = 0
        for(j=0;j<=n;j++) L[0][j] = 0

        for(i=1;i<=m;i++)
            for(j=1;j<=n;j++)
                if(X[i]==Y[j])
                    L[i,j] = L[i-1,j-1] + 1
                    R[i,j] = "diagonale"
                else if(L[i-1,j] > L[i,j-1])
                    L[i,j] = L[i,j-1]
                    R[i,j] = "sinistra"
                else
                    L[i,j] = L[i-1,j]
                    R[i,j] = "sopra"
```

In corrispondenza di ogni freccia diagonale abbiamo un elemento della sottosequenza comune.
Complessità: $\Theta(mn)$.
Problema: dato un grafo pesato G trovare il cammino minimo tra ogni coppia di nodi. Sono ammessi costi negativi ma non cicli di costo negativo.
Idea: chiamiamo k-vincolato un cammino che passa solo per nodi $1 \dots k$ (esclusi gli estremi), per $k \leq n$, e indichiamo con $d^k(x, y)$ la distanza k-vincolata tra x e y, cioè la lunghezza minima di un cammino k-vincolato.

```
    Floyd-Warshall(G)
        for each (x,y nodi in G)
            D[x,y] = 0 se x=y, peso(x,y) se x!=y, +inf altrimenti
            P[x,y] = x se x!=y e (x,y) in E, null altrimenti
        for (k=1;k<=n;k++)
            for (x,y nodi in G)
```

```
                if (D[x,k]+D[k,y]<D[x,y])
                    D[x,y] = D[x,k]+D[k,y]
                    P[x,y] = P[k,y]
        return D,P
```

Complessità: $O(n^3)$

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

Un problema (astratto) è una relazione $\mathcal{P} \subseteq I \times S$, dove I è l'insieme degli input (o istanze del problema) e S è l'insieme delle (possibili) soluzioni. In generale per ogni istanza la soluzione può non essere unica.

Un **problema (astratto) di decisione** $\mathcal{P}$ è un problema (astratto) in cui ogni input ha come soluzione vero oppure falso, ossia $\mathcal{P} : I \to \{T, F\}$. Dato un problema $\mathcal{P} : I \to \{T, F\}$, diciamo che un algoritmo A risolve $\mathcal{P}$ se $\forall i \in I, A(i) = \mathcal{P}(i)$.

Un **problema concreto** $\mathcal{P}$ è un problema il cui insieme di istanze è l'insieme delle stringhe binarie, ossia $\mathcal{P} : \{0,1\}^* \to \{T, F\}$. Un problema astratto può essere rappresentato in modo concreto tramite una codifica, ossia una funzione iniettiva: $c : I \to \{0,1\}^*$. Il problema $c(\mathcal{P})$ è definito da $c(\mathcal{P})(x) = T$ se e solo se $x = c(i)$ e $\mathcal{P}(i) = T$, ossia assumiamo convenzionalmente che la soluzione sia falso sulle stringhe che non sono codifica di nessun input.

L'**algoritmo di verifica** per un problema (astratto) $\mathcal{P} \subseteq I$ è un algoritmo $A : I \times C \to \{T, F\}$, dove C è un insieme di certificati (un certificato è una "prova" che dimostra la verità della proprietà da verificare), e $A(x, y) = T$ per qualche y se e solo se $x \in \mathcal{P}$.

- Classe P: problemi risolvibili in tempo polinomiale
- Classe NP: problemi per i quali esiste un algoritmo di verifica polinomiale
- Classe NP-C: problemi "più difficili" di NP che sappiamo risolvere solo in tempo esponenziale ma non possiamo escludere che esistano algoritmi in tempo polinomiale per risolverli. Questi problemi godono di un'importante proprietà: se si scoprisse un algoritmo che risolve uno di questi problemi in tempo polinomiale, tutti i problemi di NP-C (e come conseguenza tutti quelli di NP) sarebbero risolvibili in tempo polinomiale dimostrando quindi che $P = NP$

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

Dati due problemi concreti $\mathcal{P}_1$ e $\mathcal{P}_2$, diciamo che $\mathcal{P}_1$ è riducibile polinomialmente a $\mathcal{P}_2$, e scriviamo $\mathcal{P}_1 \leq_{\mathrm{P}} \mathcal{P}_2$, se esiste una funzione $f : \{0,1\}^* \to \{0,1\}^*$, detta funzione di riduzione, calcolabile in tempo polinomiale, tale che, per ogni $x \in \{0,1\}^*, x \in \mathcal{P}_1$ se e solo se $f(x) \in \mathcal{P}_2$.

Definizione di classe NP-C nella domanda (7).

La domanda del problema P-NP è che non si sa se $P = NP$ o $P \subset NP$, ossia se ci sono problemi verificabili in tempo polinomiale che non sono risolvibili in tempo polinomiale. Quello che è intuitivamente chiaro è che $P \subseteq NP$, ossia che se so risolvere un problema in tempo polinomiale so anche verificarlo in tempo polinomiale. Collegamento con definizione di NP-C. Se un qualunque problema NP-C appartiene alla classe P, allora si ha $P = NP$, o, equivalentemente, se è $P \neq NP$, quindi esiste almeno un problema in NP non risolvibile in tempo polinomiale, allora nessun problema NP-C è risolvibile in tempo polinomiale.

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

Prima di iniziare, definiamo CNF:

- $l ::= x|\bar{x}$ letterale
- $c ::= l_1 \vee \ldots \vee l_n$ clausola
- $\phi_{CNF} ::= c_1 \wedge \ldots \wedge c_n$ formula in CNF
- $\phi_Q ::= \phi_{CNF}|\exists x.\phi_Q|\forall x.\phi_Q$

- **SAT**(Boolean **SAT**isfiability): data una formula in forma normale congiuntiva (CNF), determinare se esiste un'assegnazione di valori di verità alle variabili che la renda vera. (Data una formula booleana dire se si riesce a dare un valore (vero o falso) alle variabili che la renda vera). Questo problema è NP-completo: si definisce un algoritmo che, dato un problema $\mathcal{P} \in NP$ e un input x per $\mathcal{P}$, costruisce una formula CNF che descrive un algoritmo non deterministico per $\mathcal{P}$, ossia la formula è soddisfacibile se e solo se l'algoritmo restituisce T.
- **3SAT**: è un problema di soddisfacibilità booleana in cui ogni clausola ha esattamente 3 letterali. È NP-completo: visto che 3SAT è una riduzione di SAT $\rightarrow 3SAT \subseteq SAT \in NP$.
- **CLIQUE**(o cricca): in un grafo non orientato $G = (V, E)$ è un insieme $V' \subseteq V$ di nodi tale che per ogni coppia di essi esiste l'arco che li collega, ossia il sottografo indotto da V' è completo. La dimensione di una clique è il numero dei suoi nodi. Il problema della clique è quello di trovare una clique di dimensione massimo in un grafo. Il corrispondente problema di decisione richiede di determinare se nel grafo esiste una clique di dimensione k. Questo problema è NP-completo: dando una riduzione di 3SAT (dando una formula 3CNF ϕ formata da k clausole), costruiamo il grafo G_ϕ nel modo seguente:
  - Un nodo per ogni occorrenza di letterale in una clausola, quindi $3k$ nodi
  - Un arco tra due occorrenze di letterali se sono in due clausole diverse e non sono uno la negazione dell'altro

  Se ϕ è soddisfacibile allora esiste una clique di dimensione k in G_ϕ. Se G_ϕ contiene una clique di k nodi, sappiamo che contiene esattamente un nodo per ogni clausola (con i corrispondenti letterali $l_1, \ldots, l_n$) scegliamo un'assegnazione di valori alle variabili che risultino veri, ciò è possibile in quanto sappiamo che non ci sono negazioni di letterali, altrimenti non ci sarebbe l'arco tra i due nodi. Con questa assegnazione ogni clausola è soddisfatta, e quindi l'intera formula.