Progettazione di Algoritmi **Anno Accademico 2021–2022**

Note per la Lezione 16

Ugo Vaccaro

Ricordiamo che esistono due approcci per trasformare un inefficiente algoritmo di Divide et Impera in un efficiente algoritmo.

1. Il primo, basato sulla tecnica della Memoization, aggiunge all'algoritmo una tabella in cui vengano memorizzate le soluzioni ai sottoproblemi, la prima volta che essi vengono "incontrati". Prima di ogni chiamata ricorsiva dell'algoritmo su di un particolare sottoproblema, viene effettuato un controllo sulla tabella per verificare se la soluzione a quel sottoproblema è stata già calcolata in precedenza.
2. La seconda tecnica risolve semplicemente tutti i sottoproblemi del problema di partenza, in maniera iterativa ed in modo "bottom-up", ovvero risolvendo prima i sottoproblemi di taglia piccola e poi via via quelli di taglia maggiore fino a risolvere l'intero problema di partenza

Quale approccio è migliore? Dipende... Entrambi hanno i loro meriti.

L'approccio basato sulla memorizzazione preserva la struttura ricorsiva tipica degli algoritmi basati su Divide et Impera (che sono in generale semplici ed eleganti). Per contro, vi è un'aggiunta di lavoro, tipo gestione stack, etc., che in certe situazioni può diventare significativo.

L'approccio iterativo "bottom-up" è in generale efficiente. Tuttavia, gli algoritmi basati su questo approccio tendono a calcolare la soluzione a *tutti* i sottoproblemi del problema originale, anche quelli che potrebbero non concorrere alla soluzione ottima del problema di partenza. Ciò non accade per gli algoritmi basati sul primo approccio, che risolvono solo i sottoproblemi "strettamente necessari", ovvero quelli che effettivamente occorrono nelle chiamate ricorsive a partire dall'istanza di partenza.

Vista l'efficacia della tecnica della Programmazione Dinamica nel produrre algoritmi efficienti, è naturale chiedersi quali sono i problemi algoritmici per cui è possibile applicare tale tecnica. Affinchè la Programmazione Dinamica sia applicabile, è necessario che la seguente proprietà valga:

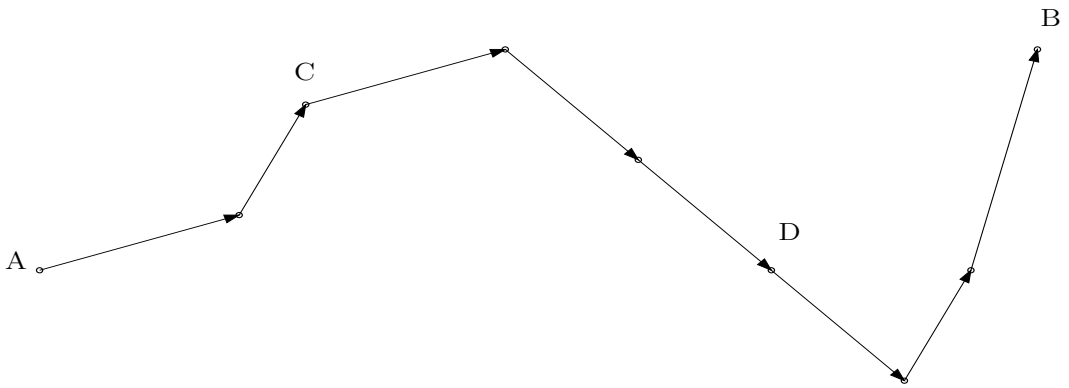
Se S è una soluzione ottima ad un problema di ottimizzazione $\mathcal{P}$,
allora le componenti di S sono soluzioni ottime a sottoproblemi di $\mathcal{P}$.

Esempio 1: Cambio di monete. Se S è un insieme di monete di cardinalità *minima* per ottenere un valore totale V, e se rimuoviamo da S una qualsiasi moneta v di valore d, allora il sottoinsieme $S - \{v\}$ è un insieme di cardinalità *minima* per il sottoproblema di ottenere il valore $V - d$ (ovvero è soluzione ottima per tale sottoproblema).

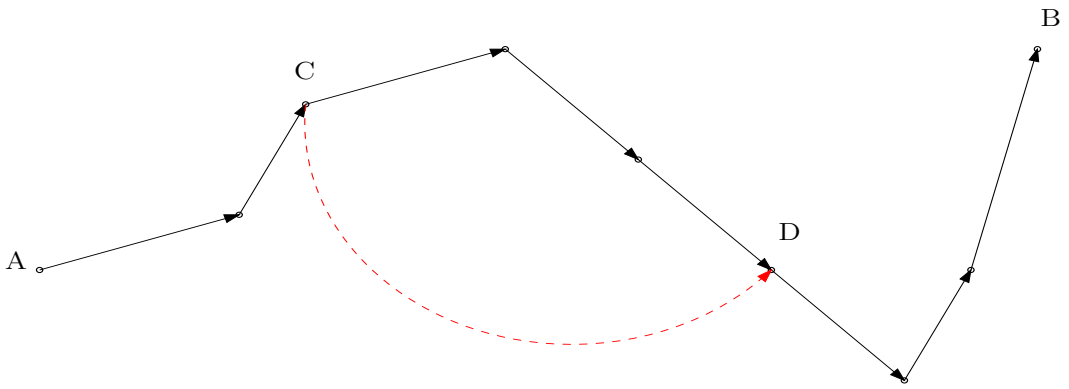
Esempio 2: Scheduling di attività. Se $\mathcal{O}$ è una soluzione ottima al problema dello Scheduling di attività, e $n \in \mathcal{O}$, allora $\mathcal{O} - \{n\}$ è una soluzione ottima al sottoproblema relativo alle attività $\{1, \ldots, p(n)\}$.

La stessa proprietà la si può provare per ciascheduno dei problemi di ottimizzazione che abbiamo risolto con la Programmazione Dinamica. Consideriamo un altro esempio che studieremo nel seguito del corso, ovvero il problema di determinare cammini di lunghezza minimi in grafi. Il problema in questione è quello di trovare, all'interno di un grafo, il più breve percorso (ovvero quello composto dal minor numero di archi) da un dato

vertice `u` ad un dato vertice `v` del grafo (si immagini di voler calcolare, ad esempio con GoogleMaps, il più breve percorso da un indirizzo di partenza ad un indirizzo di destinazione all'interno di una città). Supponiamo che per andare dal vertice A al vertice B la via più corta sia quella descritta in figura



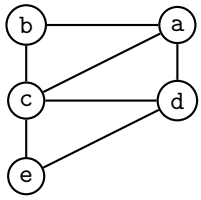
Sosteniamo che il sottopercorso in figura che va dal nodo C al nodo D è *anch'esso il più corto* che va da C a D. Infatti, se non lo fosse sarebbe possibile andare da C a D con un percorso più corto di quello nero, ad esempio quello indicato in rosso nella figura



Ma allora, potremmo andare da A a B effettuando questo cammino: prima andiamo da A a C lungo il percorso nero, poi potremmo andare da C a D lungo il percorso rosso, ed infine andare da D a B lungo il percorso nero. É chiaro che cosí facendo abbiamo trovato un percorso globale che va da A a B di lunghezza *inferiore* a quello totalmente nero, *contro l'ipotesi che quest'ultimo rappresentasse la via più corta per andare da* A a B

Ma allora tutti i problemi di ottimizzazione godono di questa proprietà? No.

Consideriamo il seguente grafo in cui la distanza tra punti è calcolata come il numero di archi che occorre attraversare per andare da un punto all'altro



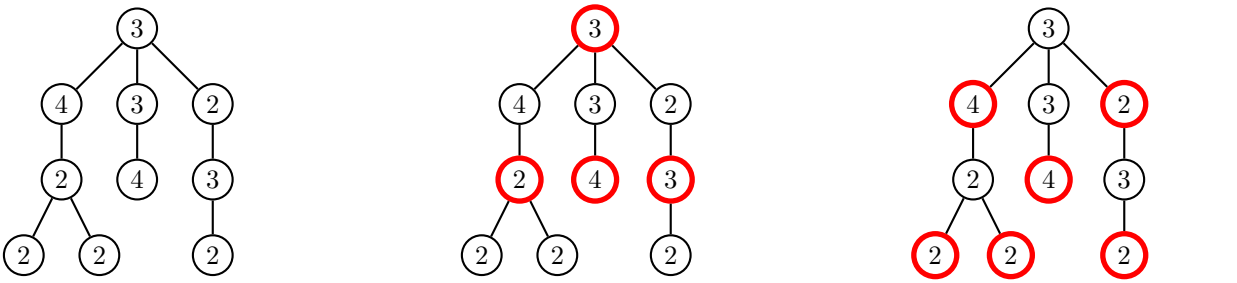
Il cammino più lungo per andare dal punto `a` al punto `e` consiste in `a-b-c-d-e`. Esso passa attraverso `b`, ma il cammino più lungo dal punto `b` al punto `e` non è `b-c-d-e`, bensí è `b-c-a-d-e`. Il fatto che per tale algoritmo non valga la proprietà che una qualsiasi soluzione ottima contenga al suo interno soluzioni ottime a sottoproblemi, ha conseguenze *drammatiche*. Innanzitutto, per il problema di determinare il cammino più lungo all'interno di un arbitrario grafo, dati il vertice di partenza `u` ed il vertice di arrivo `v`, non si può applicare la tecnica di Programmazione Dinamica. In più, a tutt'oggi non è noto nessun algoritmo di complessità polinomiale per tale problema, e vi sono fondati motivi per ritenere che un tale algoritmo non esista.

Vediamo qualche ulteriore esempio di problemi risolubili attraverso la tecnica di Programmazione Dinamica.

Problema: Dovete pianificare una festa con i vostri amici. Ogni vostro amico ha un "coefficiente di simpatia", misurato da un numero intero positivo. Purtroppo, alcuni dei vostri amici non sono tra di loro in buoni rapporti, cosicchè volete invitare solo persone che tra di loro non hanno problemi. Tra tutte le possibili scelte *sotto questo vincolo*, volete invitare il gruppo di amici che *massimizza la somma dei loro coefficienti di simpatia.*

Le relazioni del "non essere in buoni rapporti" sono esprimibili mediante gli archi di un albero.

Esempio:



Somma coefficienti simpatia=12 Somma coefficienti simpatia=16

L'albero di sopra rappresenta lo stato delle relazioni tra le persone. Il numero all'interno di ciascun nodo rappresenta il coefficiente di simpatia della persona corrispondente al nodo. Ad esempio, si vede che la persona rappresentata con la radice (che ha coefficiente di simpatia 3) non è in buoni rapporti con i nodi che nell'albero sono suoi figli. Per cui, se lo invitiamo alla festa, non possiamo invitare nessuno dei suoi figli nell'albero. Più formalmente, abbiamo il seguente problema:

Input: Albero $T = (V, E)$ con radice r, funzione peso $p : V \to \{1, 2, \ldots\}$
Output: Sottoinsieme $A \subseteq V$, di peso totale $p(A) = \sum_{v \in A} p(a)$ **massimo**, tale che $\forall u, v \in A$ vale che $(u, v) \notin E$.

Sia $opt(v)$ il peso totale di una soluzione ottima del problema ristretto al sottoalbero di T di cui il generico nodo v è la radice (noi siamo interessati a $opt(r)$). Abbiamo due possibilità: o il nodo v è un invitato presente nella soluzione ottima di questo problema, o non lo è.

1. Nel primo caso i suoi "figli" nell'albero non avranno la possibilità di partecipare ma tutti gli altri nodi possono, in linea di principio, partecipare. In tali nodi dobbiamo scegliere quelli che non hanno archi tra di loro e che massimizzano la somma dei pesi totale. Poichè i diversi sottoalberi sono disgiunti avremo quindi

$$opt(v) = p(v) + \sum_{u : u \text{ nipote di } v} opt(u)$$

2. Nel secondo caso in cui il nodo v non partecipa alla festa ottima, i suoi figli nell'albero avranno la libertà di partecipare o non partecipare a seconda di cosa sia più conveniente, quindi:

$$opt(v) = \sum_{u:\ u \text{ figlio di } v} opt(u)$$

Quale di queste due eventualità è quella migliore? Non lo sappiamo, e quindi poniamo

$$opt(v) = \max \left\{ p(v) + \sum_{u: u \text{ nipote di } v} opt(u), \sum_{u:\ u \text{figlio di } v} opt(u) \right\}$$

con caso base $opt(f) = p(f)$ per tutti i nodi foglia f nell'albero.

L'algoritmo prende in input l'albero $T = (V, E)$, il vettore dei pesi `p(v)` e la radice `r` dell'albero. Fà uso di un vettore `Opt(v)`, indicizzato dai nodi interni dell'albero T.

```
CalcolaOpt(T,p,r)
1. IF (r è una foglia) {
2.     RETURN p(r)
3.     } ELSE {
4.         IF (Opt(r) non è definito) {
5.         Opt(r)=max(p(r)+∑_{u:u nipote di r} CalcolaOpt(T,p,u), ∑_{u: ufiglio di r} CalcolaOpt(T,p,u))
      }
   }
6. RETURN (Opt(r))
```

Complessità: Per ogni nodo v dell'albero, l'algoritmo calcola il valore $Opt(v)$ una sola volta, pertanto la sua complessità è $\Theta(n)$, dove n è il numero di nodi nell'albero

$\diamond$

Per introdurre il prossimo esempio di applicazione della tecnica Programmazione Dinamica, ricordiamo la definizione di palindromo. Una stringa è palindroma se non cambia leggendola da destra a sinistra e viceversa. Ad esempio **ossesso** ed **ingegni** sono due palindromi.

Data una stringa di n caratteri $S[1] \dots S[n] = a_1 a_2 \dots a_n$ vogliamo calcolare il minimo numero di caratteri che occorre inserire in $S[1] \dots S[n]$ per renderla un palindromo. Ad esempio, la stringa **geni**, può essere trasformata in un palindromo **in**ge**g**ni mediante l'inserzione dei tre caratteri **i**, **n**, e **g**.

Sia $n[i, j]$ il minimo numero di caratteri necessari per trasformare la sottostringa $S[i]S[i+1] \dots S[j-1]S[j]$ di simboli consecutivi di $S[1] \dots S[n]$ in una stringa palindroma. Il caso base consiste in una stringa di un solo carattere, cioè il caso in cui $i = j$, per il quale vale , ovviamente, $n[i, j] = 0$ In generale, se $i < j$, distinguiamo vari casi.

- Se i caratteri $S[i]$ ed $S[j]$ sono uguali allora possiamo semplicemente risolvere il sottoproblema di rendere palindroma la sottostringa $S[i+1] \dots S[j-1]$, e ciò richiederà $n[i+1, j-1]$ caratteri.

- Se $i < j$ ed invece $S[i] \neq S[j]$, dovremo inserire almeno un carattere per rendere la stringa palindroma. Possiamo o inserire $S[i]$ alla fine di $S[i] \dots S[j]$ e risolvere ricorsivamente il sottoproblema di rendere palindroma $S[i+1] \dots S[j]$ (ciò richiederà $n[i+1, j]$ caratteri), oppure inserire $S[j]$ all'inizio di $S[i] \dots S[j]$ e risolvere ricorsivamente il sottoproblema di rendere palindroma $S[i] \dots S[j-1]$ (ciò richiederà $n[i, j-1]$ caratteri), e sceglierci la via migliore.

Avremo quindi

$$n[i,j] = \begin{cases} 0, & \text{se } i = j \\ n[i+1, j-1], & \text{se } i < j \text{ e } S[i] = S[j] \\ \min\{n[i+1, j], n[i, j-1]\} + 1 & \text{se } i < j \text{ e } S[i] \neq S[j] \end{cases}$$

L' algoritmo sarà quindi:

```
Palindromo(S[i]...S[j])
1. IF (i==j) {
2.   RETURN 0
3.   } ELSE {
4.     IF (n[i,j] non è definito) {
5.       IF (S[i]==S[j]) {
6.           n[i,j]=Palindromo(S[i+1]...S[j-1])
7.           } ELSE {
8.           n[i,j]=min(Palindromo(S[i+1]...S[j]), Palindromo(S[i]...S[j-1]))+1
       }
     }
   }
9. RETURN (n[i,j])
```

Complessità: l'algoritmo calcola tutti i $n \times n$ valori della matrice `n[i,j]`. Per ciascuno di essi il tempo richiesto è $\Theta(1)$, per una complessità totale di $\Theta(n^2)$.

$\diamond$

Consideriamo infine il seguente problema: Due compari hanno rubato n oggetti $o_1, \dots, o_n$, di valore rispettivamente $a_1, a_2, \dots, a_n$. Essendo onesti (tra di loro...), vorrebbero suddividere la refurtiva nel modo più equo possibile. Ovvero vorrebbero dividere gli oggetti $o_1, \dots, o_n$ in due sottoinsiemi in modo tale che la somma dei valori degli oggetti in uno dei due sottoinsiemi sia la più simile possibile alla somma dei valori degli oggetti nell'altro dei due sottoinsiemi. Più formalmente, abbiamo il seguente problema algoritmico:

Input: insieme di interi $A = \{a_1, \dots, a_n\}$, $a_i \in \{0, \dots, K\}$ per ogni i.
Output: partizione di A in due insiemi S_1 ed S_2 disgiunti, che minimizzi la quantità $|\sum_{a_i \in S_1} a_i - \sum_{a_j \in S_2} a_j|$.

Sia $S = (\sum_{i=1}^{n} a_i)/2$. Il problema di partizionare A in due insiemi S_1 ed S_2 in modo da minimizzare $|\sum_{a_i \in S_1} a_i - \sum_{a_j \in S_2} a_j|$ è chiaramente equivalente a trovare un insieme S_1 la cui somma di elementi sia $\leq S = (\sum_{i=1}^{n} a_i)/2$ e sia *più vicina possibile* a $S = (\sum_{i=1}^{n} a_i)/2$. Risolviamo quindi quest'ultimo problema. Sia

$$P[i,j] = \begin{cases} 1, & \text{se esiste un sottoinsieme di } \{a_1, \dots, a_i\} \text{ di somma pari a } j \\ 0 & \text{altrimenti} \end{cases}$$

Vale

$$P[i,j] = 1 \Leftrightarrow P[i-1,j] = 1 \vee P[i-1, j-a_i] = 1$$

ovvero

$$P[i,j] = \max\{P[i-1,j], P[i-1,j-a_i]\}.$$

Le condizioni iniziali sono esprimibili come

$$P[1,j] = \begin{cases} 1, & \text{se } a_1 = j \\ 0 & \text{altrimenti} \end{cases}$$

per $j = 1, \ldots,$

Pertanto, tutti i valori i valori $P[i,j]$, per $i = 1, \ldots n$ e $j = 1, \ldots, nK$ si possono calcolare in tempo $O(n^2K)$.

Ritorniamo al nostro problema di partenza che era equivalente a trovare un S_1 che avesse somma $\leq S = (a_1 + a_2 + \ldots + a_n)/2$ e fosse anche *più vicina possibile* a $S = (\sum_{i=1}^{n} a_i)/2$. Ricordando che

$$P[n,j] = \begin{cases} 1, & \text{se qualche sottoinsieme di } \{a_1, \ldots, a_n\} \text{ ha somma } j \\ 0 & \text{altrimenti} \end{cases}$$

possiamo allora semplicemente calcolare

$$\max\{j : j \leq S \text{ e } P[n,j] = 1\}$$

che rappresenta appunto la più grande somma di elementi in qualche sottoinsieme di $\{a_1, \ldots, a_n\}$ che sia di valore pari al più ad $S = (a_1 + a_2 + \ldots + a_n)/2$.

In altri termini, il tutto si riduce al calcolo dei $P(i,j)$, che è agevolmente fatto mediante il seguente algoritmo.

```
1. FOR (j=1, j<nK+1, j=j+1) {
2.   P[1, j]=0
    }
3. FOR (i=1, i<n, i=i+1) {
4.   P[1,a_i]=1
    }
5. CalcolaP(A, 1, nK)
```

```
CalcolaP(A, i, j)
1. IF (i==1) {
2.   RETURN P[i,j]
3.    } ELSE {
4.    IF (P[i,j] non è definito) {
5.     P[i,j]=max(CalcolaP(A,i-1,j), CalcolaP(A,i-1,j-a_i))
      }
6. RETURN P[i,j]
```