TECNICHE ALGORITMICHE: PROGRAMMAZIONE DINAMICA

[Deme, seconda edizione] cap. 10

Sezione 10.3

[Cormen] cap. 15

Riassunto delle tecniche algoritmiche viste

Fin qui, per risolvere problemi complicati, avete visto **3 tecniche generali di base**:

- La «tecnica» di **forza bruta**
- La tecnica **Divide et Impera** (nel corso di Algoritmi 1)
- La tecnica **Greedy**

Queste metodologie a volte possono essere inefficienti, o semplicemente non bastare per affrontare la totalità dei problemi.

Introdurremo quindi un'altra tecnica, chiamata **programmazione dinamica** (attenzione: il termine «programmazione» centra poco con Java o C).

Esempio 1: Numeri di Fibonacci

La **successione di Fibonacci** F_n o Fib(n) è una successione di numeri interi positivi definita come

$$F_n = \begin{cases} 1, & n = 1 \\ 1, & n = 2 \\ F_{n-1} + F_{n-2}, & n > 2 \end{cases}$$

(a volte è definita a partire da 0, ma il tutto non cambia)

I primi numeri della successione sono

1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, …

Calcolare F_n con il metodo Divide et Impera

È abbastanza facile scrivere, a partire dalla definizione, un algoritmo **Divide et Impera** che calcoli F_n.

Fib(n)

if n ≤ 2 **return** 1

else return Fib(n-1) + Fib(n-2)

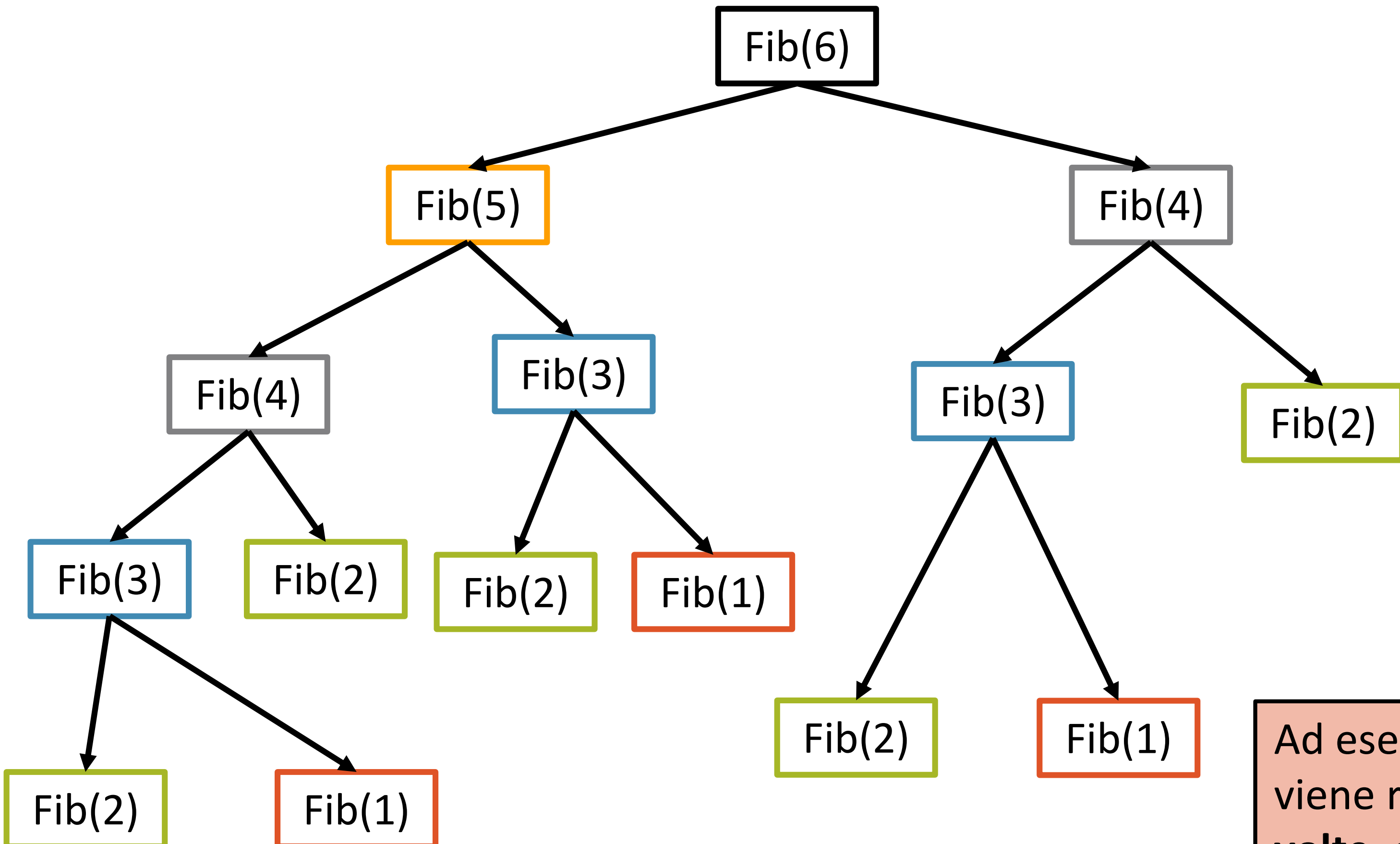
È altresì facile capire che il tempo T(n) impiegato da questo algoritmo, per un certo n, è

T(n) = T(n-1) + T(n-2) + 1

Si dimostra che ha una complessità **esponenziale**, in particolare $\mathbf{T(n) \approx 2^n}$

Il problema principale è che l'algoritmo richiama la funzione Fib(x) **sullo stesso input** x **molte volte**.

Ad esempio con n = 6 le chiamate ricorsive sono



Ad esempio, Fib(3) viene richiamata **3 volte**, e ogni volta devo richiamare Fib(2) e Fib(1).

Ottimizzazione su Fibonacci

Ma è davvero necessario ricalcolare ogni volta Fib(x)?

No, perché **non cambia il suo valore** durante l'esecuzione dell'algoritmo.

IDEA: ogni volta che calcolo un nuovo Fib(x) lo **«metto da parte»** (ad esempio in un vettore), e quando avrò di nuovo bisogno di lui dovrò solo leggere la x-esima posizione del vettore.

```
MemFib(n, F) //F è un array ausiliario di lunghezza almeno n+1
  if n ≤ 2 return 1
  else
    if F[n] == NULL
      F[n] <- MemFib(n-1, F)+MemFib(n-2, F)
  return F[n]
```

Memoizzazione

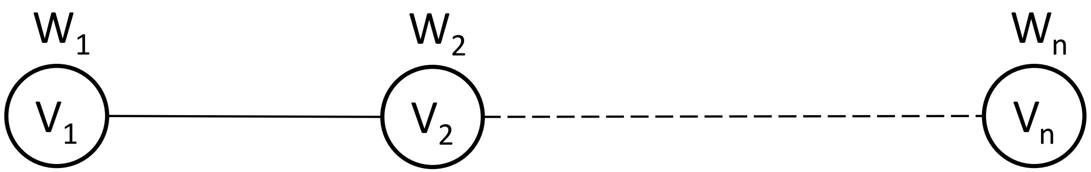
La tecnica usata per ottimizzare la funzione Fib, in cui memorizziamo via via i valori calcolati in una qualche struttura di supporto, si chiama **memoizzazione** (senza la «r»).

È possibile applicare la tecnica della memoizzazione a tutte quelle **funzioni che non cambiano comportamento**, sugli stessi input, nel tempo (in generale, tutte quelle che avete visto finora).

La memoizzazione è una parte fondamentale della programmazione dinamica (infatti, la tecnica si chiama così perché la struttura ausiliaria viene «programmata» dinamicamente, riempiendola via via con i vari valori di output).

Esempio 2:
Massimo Sottoinsieme Indipendente (MSI)

Si consideri un grafo G = (V,E) con **struttura lineare** (**path graph**), ovvero un grafo con n nodi $v_1, \dots, v_n$ e archi (v_i, v_{i+1}), per i = 1, … n − 1. **Ad ogni nodo** v_i è **associato un peso** non negativo w_i.



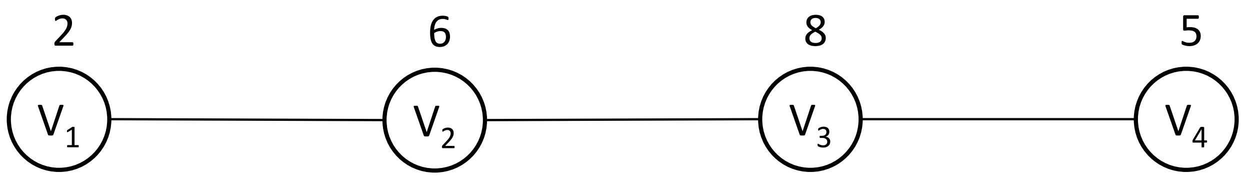
Dato un sottoinsieme S ⊆ V dell'insieme dei nodi del grafo, definiamo **peso di S**, W(S), **la somma dei pesi dei nodi in S**:

$W(S) = \sum_{v_i \in S} w_i$

Un sottoinsieme S ⊆ V dell'insieme dei nodi del grafo si dice **indipendente** se **per ogni coppia u, v** di nodi in S, u e v **non sono adiacenti** in G, ovvero non esiste un arco (u, v) in E.

PROBLEMA. Dato un grafo con struttura lineare G, con un peso non negativo w_i associato a ciascun nodo i, si determini il **sottoinsieme indipendente di nodi del grafo che ha massimo peso**.

Esempio:



Su questo path graph, la soluzione ottima è $\{V_2, V_4\}$ di peso (massimo) 11.

Proviamo 3 possibili approcci:

- Forza bruta
- Greedy
- Divide et impera

Approccio di forza bruta

Dobbiamo provare **tutte le possibili combinazioni** di nodi non adiacenti, e restituire quella con peso massimo.

$\{V_1, V_2\}$ non è indipendente

$\{V_1, V_3\}$ ha peso 10

$\{V_1, V_4\}$ ha peso 7

$\{V_2, V_3\}$ non è indipendente

$\{V_2, V_4\}$ ha peso 11

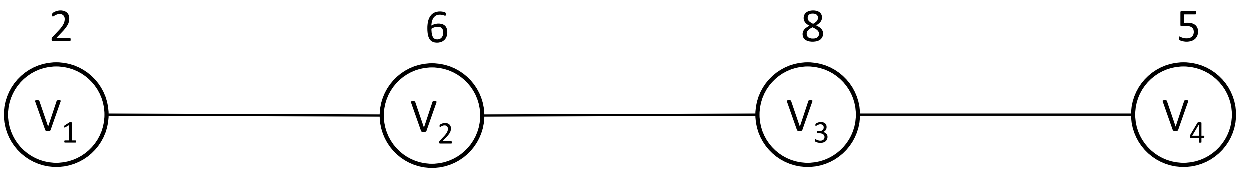
$\{V_3, V_4\}$ non è indipendente

L'algoritmo restituisce la soluzione ottima (ovviamente), ma deve considerare un **numero combinatorio** di casi.

Approccio Greedy

Iterativamente scegliere il **nodo di peso massimo** non ancora selezionato, che **non sia adiacente ad alcun nodo già selezionato**.

Esempio:



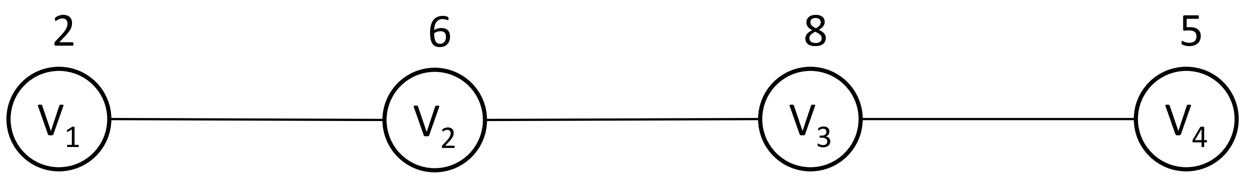
La prima iterazione sceglie V_3 con peso 8. La seconda iterazione deve per forza scegliere V_1. Il risultato ha peso 10, ma $\{V_2, V_4\}$ ha peso 11, quindi $\{V_1, V_3\}$ non è una soluzione ottima.

L'approccio greedy non è quindi corretto.

Approccio Divide et Impera

Si divide il problema in **due sottoproblemi di uguale dimensione** e si risolvono ricorsivamente tali sottoproblemi.

Esempio:



La prima chiamata divide il grafo in 2 parti: $\{V_1, V_2\}$ e $\{V_3, V_4\}$. La chiamata su $\{V_1, V_2\}$ restituisce $\{V_2\}$ e la chiamata su$\{V_3, V_4\}$ restituisce $\{V_3\}$. Ma le due soluzioni parziali non sono combinabili tra loro ($\{V_2, V_3\}$ non è indipendente).

L'approccio Divide et Impera non è corretto.

Quindi?

Notiamo però che una **sottostruttura ottima** (vedi Greedy) esiste.

Sia P_i lo stesso **problema ristretto** ai primi i nodi $\{V_1, V_2, \dots V_i\}$.

Teorema (Sottostruttura Ottima MSI): Se **S_i è una soluzione ottima** per P_i, allora vale una delle seguenti affermazioni:

1. $V_i \notin S_i$ e S_i è anche una soluzione ottima per P_{i-1}
2. $V_i \in S_i$ e $S_i - \{V_i\}$ è una soluzione ottima per P_{i-2}.

DIMOSTRAZIONE. Se S_i è una soluzione ottima per P_i, allora sono possibili due casi (facili):

1. V_i non fa parte di S_i
2. V_i fa parte di S_i

Caso 1: V_i non fa parte di S_i

Nel primo caso, **S_i è** anche **una soluzione ammissibile** (per ora non sappiamo se ottimale) **per il problema P_{i-1}**, visto che non contiene V_i.

Dimostriamo che **è ottima anche per P_{i-1}**:

se non lo fosse, esisterebbe una soluzione **S'_{i-1} per P_{i-1} con peso maggiore di S_i**.

ma tale soluzione sarebbe una soluzione **anche per P_i** e sarebbe migliore (di **peso maggiore**) **di S_i**.

Ma questo **contraddice** l'ipotesi che **S_i** sia la **soluzione ottima** per P_i (**assurdo**).

Quindi, la prima asserzione dell'enunciato (**$V_i \notin S_i$ e S_i è anche una soluzione ottima per P_{i-1}**) è valida.

Caso 2: V_i fa parte di S_i

Nel secondo caso, dimostriamo che

$S' = S_i - \{V_i\}$ è una soluzione ottima per il problema P_{i-2}

V_i appartiene a S_i (per ipotesi), quindi **V_{i-1} non può appartenere a S_i** (e quindi **nemmeno a S'**) per la condizione di **indipendenza sulle soluzioni**.

(Dimostriamo che) per il problema P_{i-2}, **S' è una soluzione ottima**. Se non lo fosse, esisterebbe una soluzione S'' per P_{i-2} di peso maggiore di S'. Ma allora **$S'' \cup \{V_i\}$ sarebbe una soluzione per P_i e avrebbe peso maggiore di S_i**, contrariamente all'ipotesi che S_i sia ottimale per P_i (**assurdo**).

Quindi in questo caso è valida la seconda asserzione dell'enunciato (**$V_i \in S_i$ e $S_i - \{V_i\}$ è una soluzione ottima per P_{i-2}**).

Corollario

Se S_{i-1} è una soluzione ottima per P_{i-1} e S_{i-2} è una soluzione ottima per P_{i-2}, allora **la soluzione ottima per P_i è la soluzione di peso massimo tra S_{i-1} e $S_{i-2} \cup \{V_i\}$**.

Da questo corollario possiamo definire una formula per il calcolo di S_i:

$$S_0 = \emptyset$$
$$S_1 = w_1$$
$$S_i = \begin{cases} S_{i-1} & se\ W(S_{i-1}) > W(S_{i-2}) + w_i \\ S_{i-2} \cup V_i & altrimenti \end{cases}$$

Possiamo calcolare S_i a partire dai **sottoproblemi S_{i-1} e S_{i-2}**.

Memoizzazione per MSI

La soluzione di un problema di tipo MSI, così come definita ora, è molto simile alla definizione data per i numeri di Fibonacci. Possiamo quindi applicare la **memoizzazione**, salvando in **A[i] l'msi per il sottoproblema S_i**, ottenendo il seguente algoritmo:

```
Msi (n, A) //A è un array ausiliario di lunghezza almeno n+1
  if n == 0 return {}          //possiamo anche inizializzare così A[0] e A[1]
  if n == 1 return V_1           semplificando l'algoritmo
  else
    if A[n] == NULL
      A[n] <- max_peso(Msi(n-1, A), Msi(n-2, A)+V_n)
  return A[n]
```

dove max_peso restituisce il sottoinsieme di peso maggiore

Sottostruttura Ottimale

Notiamo che la proprietà della sottostruttura ottimale (**un problema può essere risolto a partire dalle soluzioni dei suoi sottoproblemi**) è fondamentale per poter esprimere le funzioni in una forma che permetta la memoizzazione.

Aggiungiamo quindi la proprietà della **sottostruttura ottimale** alle caratteristiche **necessarie per l'applicabilità** della tecnica della **programmazione dinamica**.

Approccio Bottom-up

È facile notare che in entrambi i casi (Fibonacci e MSI), dato un **problema** di **dimensione n**, **per trovare la soluzione**, dobbiamo **risolvere una volta tutti i sottoproblemi i-esimi, con 0 ≤ i ≤ n**.

Di conseguenza, quando si applica la tecnica della programmazione dinamica, invece che un approccio di tipo Top-down (adottato dalle tecniche Divide et Impera) si preferisce **un approccio Bottom-up**, che parte dal più piccolo sottoproblema fino a risolvere l'n-esimo sottoproblema, che altri non è che il problema di partenza.

Aggiungiamo quindi la caratteristica «**Bottom-up**» alle tecniche di **programmazione dinamica** (oltre alla memoizzazione e alla necessità di avere una sottostruttura ottimale).

Fibonacci – programmazione dinamica

L’algoritmo per il calcolo dell’n-esimo numero di Fibonacci, espresso con la tecnica Bottom-up (e quindi con l’approccio della programmazione dinamica), è il seguente:

```
MemFib(n, F) //F è un array ausiliario di lunghezza almeno n+1
  F[1] <- F[2] <- 1
  for i = 3..n
    F[i] <- F[i-1] + F[i-2]
  return F[n]
```

Nota: applicando l’approccio Bottom-up, non c’è più bisogno della ricorsione.

MSI programmazione dinamica

L’algoritmo per il calcolo del massimo sottoinsieme indipendente espresso con la tecnica della programmazione dinamica è:

```
Msi (n, A) //A è un array ausiliario di lunghezza almeno n+1
  A[0] <- {}
  A[1] <- {V_1}
  for i = 2..n
    A[i] <- max_peso(A[i-1], A[i-2]+V_i)
  return A[n]
```

Riassumendo

La **programmazione dinamica** si applica in problemi in cui è possibile **dividere il problema iniziale in sottoproblemi** (come per la tecnica Divide et Impera e la tecnica Greedy).

È necessario che il problema goda della proprietà della **sottostruttura ottimale** (come per Greedy).

In particolare, è utile applicare la Programmazione Dinamica quando **un sottoproblema viene risolto più volte** (rendendo inefficiente l'approccio Divide et Impera)

Per farlo dobbiamo **riscrivere la funzione in una forma ricorsiva** (permesso dalla sottostruttura ottimale), **individuare i sottoproblemi** ed una struttura per **memoizzare le loro soluzioni**.

In più, possiamo/dobbiamo adottare un approccio di tipo **Bottom-up**.

Cosa devo aver capito fino ad ora

- Cos'è la Programmazione Dinamica
- A che genere di problemi si applica (e perché in tali problemi ha risultati migliori rispetto ad altre tecniche)
- Quali sono le condizioni necessarie per applicarla
- Come si applica
- Cos'è la memoizzazione
- Algoritmo di PD per numeri di Fibonacci
- Algoritmo PD per Massimo Sottoinsieme Indipendente

…se non ho capito qualcosa

- Alzo la mano e chiedo
- Ripasso sul libro
- Chiedo aiuto sul forum
- Chiedo o mando una mail al docente