Programmazione Dinamica

Storia

Il termine è stato inizialmente utilizzato negli anni quaranta dal matematico Richard Bellman per descrivere il processo di soluzione dei problemi *nei quali sia necessario trovare la migliore soluzione, anche per sottoproblemi*. Nel 1953 affinò il termine ad assumere il significato moderno. Questo metodo fu studiato per l'analisi di sistemi e scopi ingegneristici riconosciuti dall'IEEE.

La parola *programmazione* in "programmazione dinamica" non ha alcuna particolare attinenza con la programmazione informatica. Originariamente il termine programmazione dinamica si applicava unicamente alla soluzione di alcuni tipi di problemi operativi al di fuori dell'area informatica, esattamente come per la programmazione lineare. Un programma è semplicemente la pianificazione per l'azione che sarà prodotta. Per esempio, una lista mirata di eventi, nel caso di una esibizione, è talvolta chiamata programma. Programmare, in questo caso, significa trovare un piano d'azione accettabile.

Programmazione dinamica

Lo studio delle tecniche algoritmiche inizia di solito con gli algoritmi «greedy», che in un certo senso sono l'approccio più naturale al disegno degli algoritmi.

Sfortunatamente per molti problemi la difficoltà non è quella di individuare la soluzione greedy più efficiente, ma il fatto che non vi è nessun algoritmo greedy che risolve il problema.

Per questi problemi è importante avere altri approcci.

La tecnica «divide et impera» può a volte essere l'alternativa, ma spesso non e` sufficiente a ridurre il tempo esponenziale della ricerca bruta a tempo polinomiale.

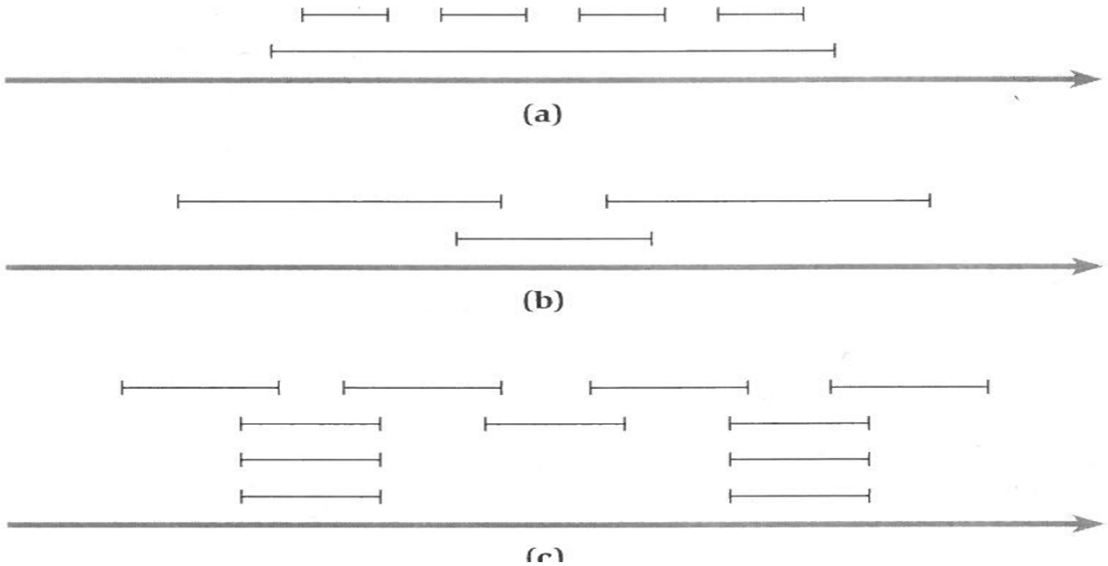
Programmazione dinamica

Una tecnica di disegno più potente e sottile è la «*programmazione dinamica*»: l'idea base viene dall'intuizione che sta sotto alla divide et impera ed è essenzialmente l'opposto della strategia greedy.

Si esplora implicitamente lo spazio di tutte le possibili soluzioni decomponendo i problemi in una serie di sottoproblemi e si costruiscono poi le soluzioni a sottoproblemi di dimensione sempre maggiore. Si può vedere la programmazione dinamica come operare pericolosamente vicino alla ricerca per forza bruta: sebbene però lavori sistematicamente in un insieme esponenziale di possibili soluzioni al problema, lo fa senza esaminarle tutte esplicitamente.

Un algortmo greedy: scheduling di intervalli

Problema: abbiamo una risorsa (ex: tempo CPU, sala seminari, etc.) e tante richieste di usare la risorsa per un certo periodo di tempo. L'obiettivo è quello di soddisfare il massimo numero di richieste in un certo tempo.
Ogni richiesta i ha un tempo di inizio s(i) e un tempo di fine f(i) (ovviamente s(i) < f(i)).
Alcuni esempi:

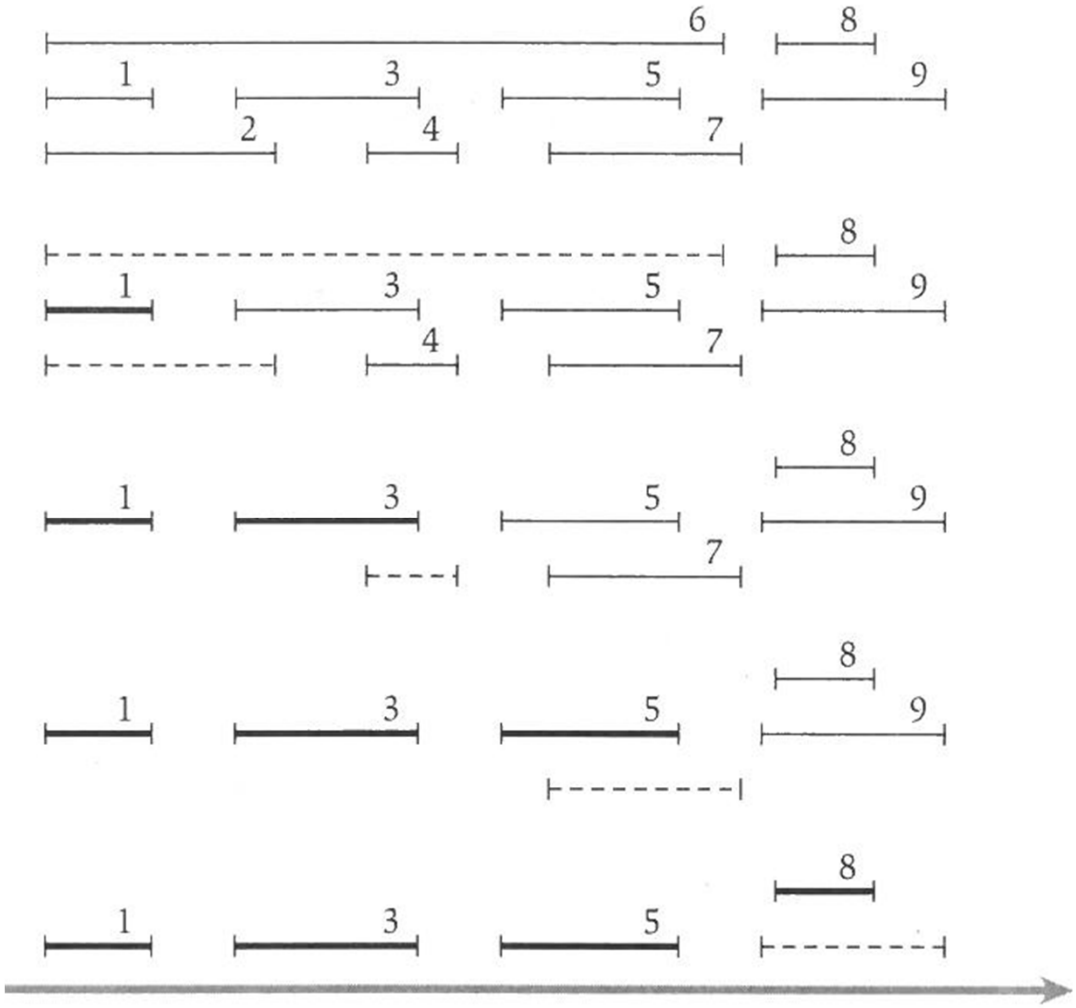


scheduling di intervalli: l'algoritmo

Bisogna prima ordinare gli intervalli *per tempo di fine crescente*. Sia R l'insieme degli intervalli es S l'insieme che conterrà la soluzione

```
Compute-Opt-greedy (n, s, f)
        i ← 1
        S ← emptyset
        while  i <= n  do
                inserisci i in S
                m ← i +1
                while f(i) > s(m)
                        m ← m+1    // risorse scartate //
                i ← m
        return S
```

esempio di esecuzione.



valutazione della complessità

- L'ordinamento delle risorse richiede O(n lg n);
- Il doppio ciclo richiede un numero di confronti eguale ad n. Il numero di istruzioni tra due successivi confronti è limitato da una costante. Quindi complessivamente il tempo di esecuzione sarà O(n);
- In totale il tempo è O(n lg n).

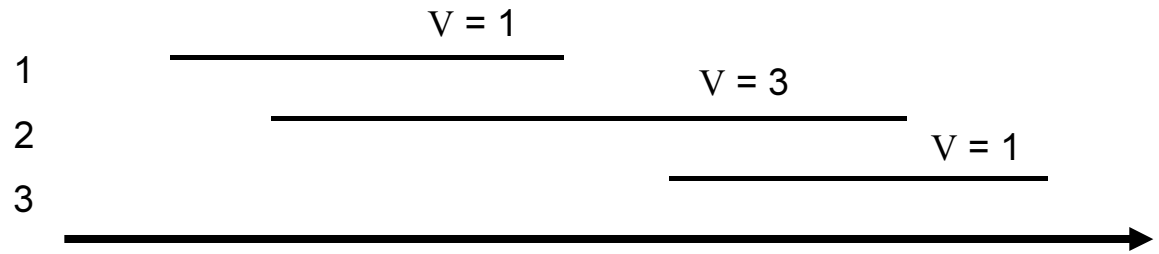
Un esempio più difficile: scheduling di intervalli pesati

Consideriamo il problema seguente:

"Sono dati n intervalli, ognuno specificato da un tempo di inizio (s_i) e da un tempo di fine (f_i) e da un valore v_i. Due intervalli sono compatibili se non si sovrappongono. Selezionare gli intervalli compatibili in modo che il valore complessivo degli intervalli selezionati sia massimo."

Una soluzione ottima a questo problema di scheduling di intervalli, *non* può essere trovata con un algoritmo greedy.

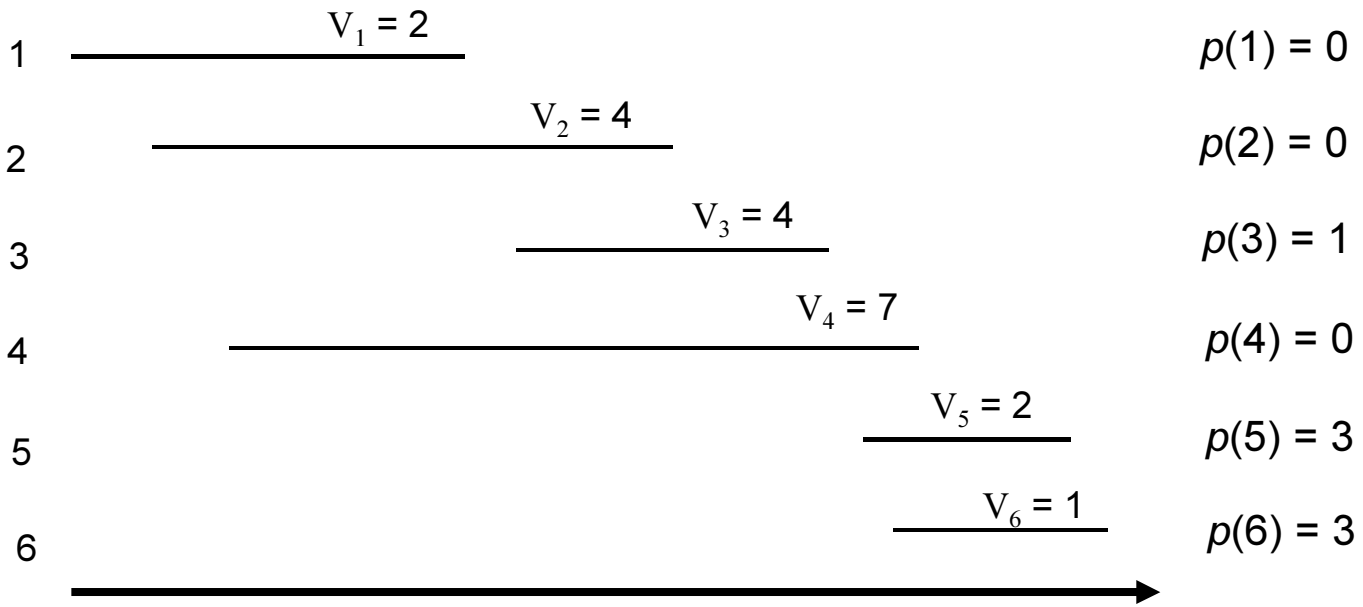
Se gli intervalli hanno un peso v (un valore) e si intende selezionare un sottinsieme $S \subseteq \{1, \ldots, n\}$ di intervalli mutuamente compatibili in modo da massimizzare la somma dei pesi degli intervalli selezionati (un insieme di valore massimo $\sum_{i \in S} v_i$), l'algoritmo greedy non funziona, anzi per questo problema non è noto nessun algoritmo greedy.



Un esempio: scheduling di intervalli pesati

Supponiamo che gli intervalli siano ordinati per tempo di fine non decrescente: $f_1 \leq f_2 \leq \ldots \leq f_n$;

p(j) sia il massimo indice i < j tale che gli intervalli i e j sono disgiunti



Un esempio: scheduling di intervalli pesati

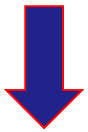
Supponiamo di avere una soluzione ottima.

L'ultimo intervallo, l'ennesimo,

- o appartiene alla soluzione ottima
- o non vi appartiene.

Nel primo caso nessun intervallo tra $p(n)$ e n può appartenere alla soluzione ed essa deve essere ottima per gli intervalli 1, 2, …, $p(n)$.

Nel secondo la soluzione ottima è uguale a quella per gli intervalli 1,…, n-1



Trovare una soluzione ottima per gli intervalli 1, 2, …, n implica cercare soluzioni ottime di problemi più piccoli della forma 1, 2, …, j.

Allora per ogni valore di j tra 1 e n la soluzione ottima e` ottima anche per gli intervalli da 1 a j.

Denotiamo con OPT(j) il valore di questa soluzione e definiamo OPT(0) = 0.

Un esempio: scheduling di intervalli pesati

$$OPT(j) = \max \{v_j + OPT(p(j)), OPT\ (j-1)\}$$

L'intervallo j appartiene a una soluzione ottima sull'insieme {1, 2, …, n} se e solo se

$$v_j + OPT(p(j)) \geq OPT\ (j-1)\}$$

Queste osservazioni sono la prima componente essenziale su cui si basa la programmazione dinamica: una equazione di ricorrenza che esprime la soluzione ottima in termini delle soluzioni ottime a sottoproblemi piu` piccoli.

La definizione di OPT(j) fornisce direttamente un algoritmo ricorsivo per calcolare OPT(n), supponendo di avere gli intervalli ordinati per tempi di fine e di aver calcolato i valori di *p*(j) per ogni j.

Un esempio: scheduling di intervalli pesati

Compute-Opt(j)

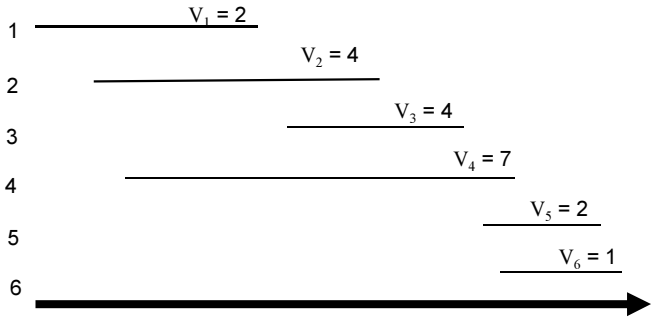
if (j = 0) return 0

else return max(v_j + Compute-Opt(p(j)), Compute-Opt(j-1))

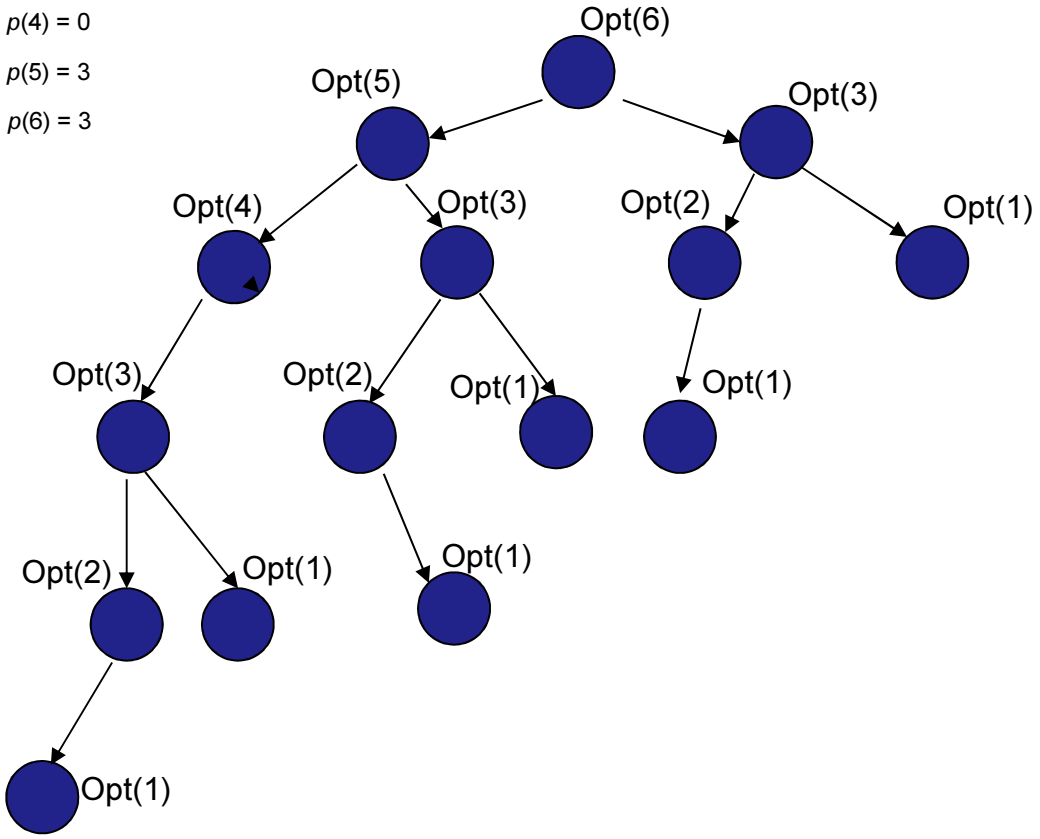
Compute-Opt(j) calcola correttamente OPT(j)
per ogni j = 1, 2, …, n

Sfortunatamente l'algoritmo richiede tempo esponenziale nel caso peggiore, per esempio nel caso prima preso in esame:

Un esempio: scheduling di intervalli pesati



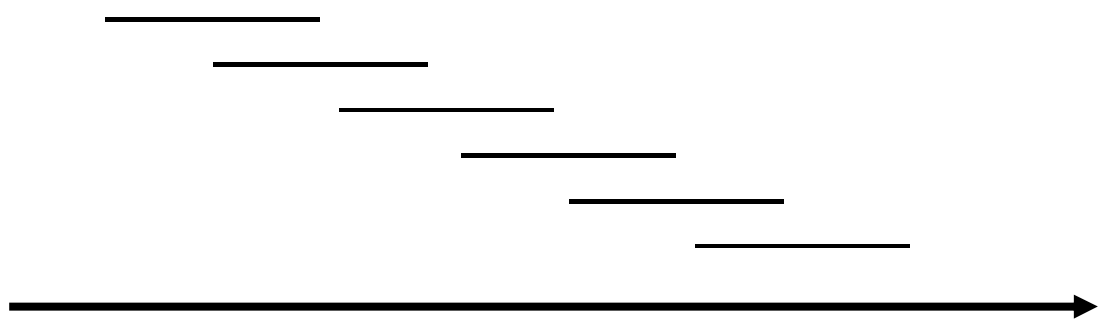
$p(1) = 0$
$p(2) = 0$
$p(3) = 1$
$p(4) = 0$
$p(5) = 3$
$p(6) = 3$



L'albero dei sottoproblemi aumenta molto velocemente

Un esempio: scheduling di intervalli pesati

Un caso ancora peggiore e` quello mostrato nella figura seguente, in cui *p*(j) = j-2 per ogni j = 2, 3, …, n. Compute-Opt(j) genera chiamate ricorsive separate sui problemi j-1 e j-2: il numero di chiamate su questa istanza aumenta esponenzialmente.



Tale aumento esponenziale e` dovuto alla ridondanza del numero di volte in cui ognuna di queste chiamate viene effettuata.

Come eliminare la ridondanza?

«Memoizing» la ricorsione

Strategia migliore: non calcolare due volte lo stesso sottoproblema

Sia M un array globale di dimensione n

```
M-Compute-Opt(j)
    if (j = 0) return 0
    else if (M[j] non e` vuoto) return M[j]
    else
        M[j] ← max(v_j + M-Compute-Opt(p(j)), Compute-Opt(j-1))
    return M[j]
```

Il tempo di esecuzione di M-Compute-Opt(n) e` O(n), supponendo gli intervalli in input ordinati in modo non decrescente per tempi di fine.

Trovare una soluzione

```
Find-solution(j)
        if (j = 0) then output nothing
        else if (v_j + M[p(j)] ≥ M[j-1])
                Find-solution(p(j));
                output j
        else Find-solution(j-1)
```

Dato l'array M dei valori ottimi dei sottoproblemi,
Find-solution genera una soluzione ottima in tempo O(n).

Principi base della programmazione dinamica

Iterare sui sottoproblemi piuttosto che calcolare
le soluzioni ricorsivamente.

Per capire meglio la programmazione dinamica, aiuta riformulare l'algoritmo in una versione essenzialmente equivalente, che cattura esplicitamente l'essenza della tecnica e serve come "template" generale per gli algoritmi che vedremo.

Il punto chiave per l'efficienza dell'algoritmo e` l'array M.

Esso «codifica» il fatto che servono i valori delle soluzioni ottimali dei sottoproblemi sugli intervalli {1, 2 …, j} per ogni j e che per definire il valore di M[j] servono valori già memorizzari nell'array.

Possiamo direttamente calcolare le entry di M con un algoritmo iterativo piuttosto che usare la versione memorized della ricorsione.

Algoritmo iterativo

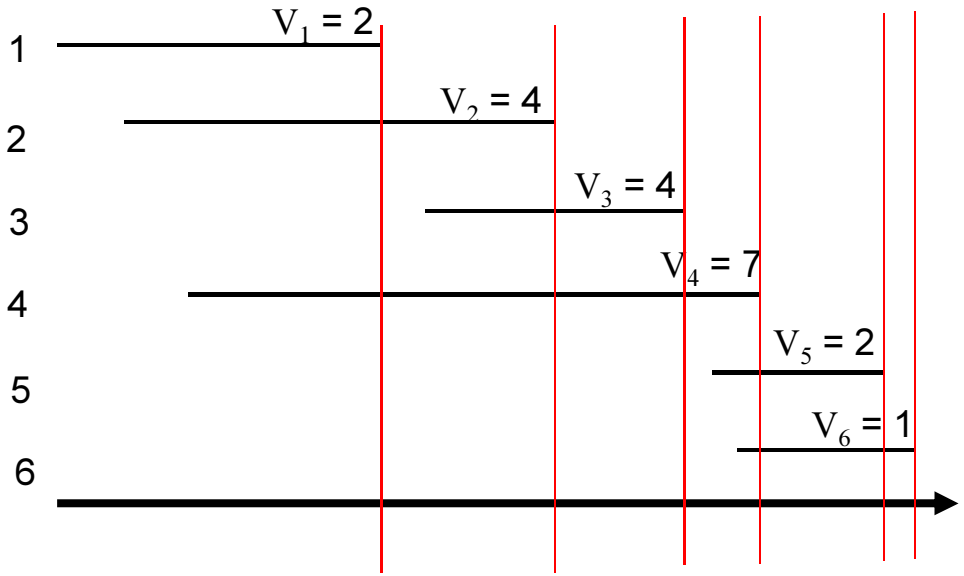
Iterative-Compute-Opt

$\underline{M}[0] \leftarrow 0$

$\underline{for}\ j \leftarrow 1$ to n

$M[j] \leftarrow \max(v_j + M[(p(j))], M[(j-1)])$

L'algoritmo Iterative-Compute-Opt è corretto e il suo tempo di esecuzione e` O(n).



$p(1) = 0$

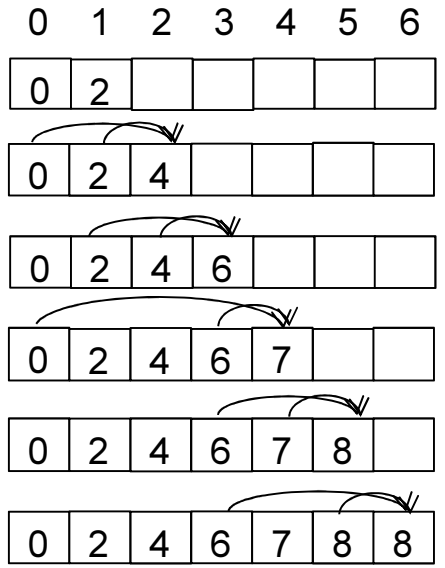
$p(2) = 0$

$p(3) = 1$

$p(4) = 0$

$p(5) = 3$

$p(6) = 3$



Principi base della programmazione dinamica

Che cosa possiamo imparare dall'esempio?

Serve una collezione di sottoproblemi derivati dal problema originale che soddisfa alcune proprietà base:

a) Il numero di sottoproblemi è *polinomiale*
b) La soluzione al problema originale puo` essere facilmente calcolata dalle soluzioni ai sottoproblemi
c) Vi e` un ordinamento naturale sui sottoproblemi dal "piu` piccolo" al "piu` grande" insieme con una ricorrenza facile da calcolare

A volte e` piu` facile iniziare il processo di disegno dell'algoritmo formulando un insieme di sottoproblemi che sembra naturale e poi cercando la ricorrenza che li lega, ma spesso, come nell'esempio visto, puo` essere utile definire la ricorrenza ragionando sulla struttura della soluzione ottima e poi determinare quali sottoproblemi e` necessario risolvere per "srotolare" la ricorrenza.
Questa relazione "chicken-and-egg" fra sottoproblemi e ricorrenza e` un gioco sottile che sta alla base della programmazione dinamica.

Programmazione dinamica

- Strategia sviluppata negli anni ’50 da Bellman
- Ambito: problemi di *ottimizzazione*
- L’obiettivo è trovare la soluzione ottima secondo un “*indice di qualità*” assegnato ad ogni soluzione candidata

- Approccio:
  - Come per la tecnica divide-et-impera l’idea è quella di risolvere un problema combinando le soluzioni di sottoproblemi
  - Ci sono però importanti differenze con divide-et-impera

Programmazione dinamica vs divide et impera

Divide et impera

- Tecnica ricorsiva
- Approccio top-down
- Vantaggiosa solo quando i sottoproblemi sono indipendenti, altrimenti gli stessi sottoproblemi possono essere risolti più volte

Programmazione dinamica

- Tecnica (essenzialmente) iterativa
- Approccio bottom-up
- Vantaggiosa quando ci sono sottoproblemi in comune

Una definizione induttiva di funzione si traduce naturalmente in un algoritmo ricorsivo per il calcolo, *top-down*, della funzione.

Dal punto di vista della *correttezza* questo è del tutto soddisfacente; dal punto di vista dell'*efficienza* non è detto.

In genere è possibile ricavare dalla definizione induttiva anche un algoritmo iterativo. Di solito è un po' più difficile, ma si ottengono spesso dei vantaggi dal punto di vista dell'efficienza.