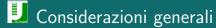
Progettazione di Algoritmi

Salvatore Filippone salvatore.filippone@uniroma2.it

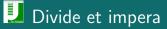


La progettazione degli algoritmi non ammette "ricette" universali, e richiede una discreta inventiva

Fasi di analisi

- Classificazione del problema: decisionale, ricerca, ottimizzazione;
- Caratterizzazione della soluzione;
- Tecnica specifica di progettazione divide et impera, programmazione dinamica, greedy, ricerca locale, backtrack, probabilistica;
- Strutture di dati.

S. Filippone Ing. Alg. 2/21



Suddividere un problema in sottoproblemi, risolverli, e combinare le soluzioni in una soluzione del problema principale

Abbiamo già visto diversi esempi di questa tecnica:

- Quicksort;
- Mergesort;
- Algoritmo di Strassen per la moltiplicazione di matrici.

S. Filippone Ing. Alg. 3/21



Programmazione dinamica

La programmazione dinamica è una tecnica in cui si risolvono tutti i sottoproblemi di un problema (come divide et impera), verso la soluzione del problema principale, in modo da non risolvere mai due volte uno stesso sottoproblema

Il coefficiente binomiale

Algorithm 1: integer C(integer n, k)

```
if n < k then
```

- Risultato 0
- else if n = k or k = 0 then
- Risultato 1

else

Risultato C(n-1, k-1) + C(n-1, k)

4/21

S. Filippone Ing. Alg.



Programmazione dinamica

La programmazione dinamica è una tecnica in cui si risolvono tutti i sottoproblemi di un problema (come divide et impera), verso la soluzione del problema principale, in modo da non risolvere mai due volte uno stesso sottoproblema

Il coefficiente binomiale

Algorithm 3: integer C(integer n, k)**Algorithm** 4: integer tartaglia(integer n, integer[][]C) if n < k then Risultato 0 for $i \leftarrow 0$ to n do else if n = k or k = 0 then $C[i,0] \leftarrow 1;$ Risultato 1 $C[i,i] \leftarrow 1$: else for $i \leftarrow 2$ to n do **Risultato** C(n-1, k-1) + C(n-1, k)for $i \leftarrow 1$ to i - 1 do $C[i,j] \leftarrow C[i-1,j-1] + C[i-1,j]$:

Se le soluzioni vengono memorizzate esplicitamente si parla di *memoization*.



Condizioni di applicabilità

- È possibile combinare soluzioni di sottoproblemi per risolvere un problema più grande;
- 2 Le decisioni ottime di un sottoproblema sono ottime anche quando è parte di un problema più grande;

La programmazione dinamica è conveniente quando divide-et-impera risolve più volte gli stessi sottoproblemi

Condizioni per complessità polinomiale

- Il numero dei sottoproblemi deve essere (non più che) polinomiale;
- Si può usare una tabella per memorizzare tutte le soluzioni dei sottoproblemi;
- 3 Il tempo di ricombinazione dei sottoproblemi è polinomiale.



S. Filippone Ing. Alg. 5 / 21

Esempio — la successione di Fibonacci

Algorithm 5: integer *fibonacci*(**integer** *n*)

if n = 0 or n = 1 then

Risultato 1

else

Risultato fibonacci(n-1) + fibonacci(n-2)

6/21

Esempio — la successione di Fibonacci

Algorithm 8: integer *fibonacci*(**integer** *n*)

```
if n = 0 or n = 1 then
    Risultato 1
```

else

Risultato fibonacci(n-1) + fibonacci(n-2)

Algorithm 9: integer *fibonacci*(**integer** *n*)

integer
$$[] F \leftarrow \text{new integer}[0 \dots n];$$

 $F[0] \leftarrow F[1] \leftarrow 1;$
for $i \leftarrow 2$ to n do
 $| F[i] \leftarrow F[i-1] + F[i-2];$
Risultato $F[n]$

Esempio — la successione di Fibonacci

```
Algorithm 11: integer fibonacci(integer n)

if n = 0 or n = 1 then

Risultato 1

else

Risultato fibonacci(n-1) + fibonacci(n-2)
```

```
Algorithm 12: integer fibonacci (integer n)

integer [] F \leftarrow \text{new integer}[0 \dots n];
F[0] \leftarrow F[1] \leftarrow 1;
for i \leftarrow 2 to n do
 | F[i] \leftarrow F[i-1] + F[i-2];
Risultato F[n]
```

Algorithm 13: integer *fibonacci*(**integer** *n*)

```
 \begin{aligned} &\textbf{integer} \ F_0, F_1, F_2; \\ &F_0 \leftarrow F_1 \leftarrow F_2 \leftarrow 1; \\ &\textbf{for} \ i \leftarrow 2 \, \textbf{to} \, n \, \textbf{do} \\ & \middle| \ F_0 \leftarrow F_1; \\ &F_1 \leftarrow F_2; \\ &F_2 \leftarrow F_0 + F_1; \\ &\textbf{Risultato} \ F_2 \end{aligned}
```



Esempio — moltiplicazione di matrici

Si consideri il problema di calcolare il prodotto di *n* matrici rettangolari

$$A_1 \cdot A_2 \cdot \cdot \cdot A_n$$
;

questo prodotto è ben definito se il numero di colonne c_h della matrice A_h è eguale al numero di righe della matrice A_{h+1} .

La moltiplicazione costa $c_{h-1} \cdot c_h \cdot c_{h+1}$; infatti

$$(A_h \cdot A_{h+1})[i,j] = \sum_{k=1}^{c_h} A_h[i,k] \cdot A_{h+1}[k,j].$$

La moltiplicazione è associativa, quindi il risultato finale è identico (a meno degli arrotondamenti floating-point), ma il costo varia a seconda dell'ordine. Caso particolare di LAMP (Linear Algebra Mapping Problem).

S. Filippone Ing. Alg.



Esempio — moltiplicazione di matrici

Consideriamo

$$A_1(30 \times 5), A_2(5 \times 25), A_3(25 \times 7);$$

il prodotto

$$(A_1 \times (A_2 \times A_3))$$

costa 1925 moltiplicazioni, invece

$$((A_1 \times A_2) \times A_3)$$

ne costa 9000.

Catena di matrici

Date n matrici A_1, A_2, \ldots, A_n , trovare una formulazione (parentesizzazione) che minimizzi il costo del calcolo del prodotto $A_1 \cdot A_2 \cdots A_n$



Possiamo caratterizzare la soluzione ottima con

Parentesizzazione

 $P_{i,j}$ è una parentesizzazione di $A_i \cdot A_{i+1} \cdots A_j$ se:

$$P_{i,j} = \begin{cases} A_i & \text{se } i = j \\ (P_{i,k} \cdot P_{k+1,j}) & \text{per qualche } k \end{cases}$$

Se $P_{i,j}$ è ottima, allora $P_{i,k}$ e $P_{k+1,j}$ sono anch'esse ottime

Il costo di $(P_{i,k} \cdot P_{k+1,j})$ è pari al costo di $P_{i,k}$, più il costo di $P_{k+1,j}$, più il costo di moltiplicare le due matrici risultanti $c_{i-1} \times c_k$ e $c_k \times c_j$. Quindi:

$$M[i,j] = \begin{cases} \min_{i \le k \le j-1} (M[i,k] + M[k+1,j] + c_{i-1}c_kc_j) & \text{se } 1 \le i < j \le n \\ 0 & \text{se } 1 \le i \le j \le n \end{cases}$$

S. Filippone Ing. Alg. 9 / 21

Esempio — moltiplicazione di matrici

Algorithm 14: parentesi(integer[] c, integer[] [] M, integer[] [] S, integer n)

```
\begin{array}{l} \text{for } i \leftarrow 1 \, \text{to } n \, \text{do} \\ \mid M[i,i] \leftarrow 0; \\ \text{for } h \leftarrow 2 \, \text{to } n \, \text{do} \\ \mid \text{for } i \leftarrow 1 \, \text{to } n-h+1 \, \text{do} \\ \mid j \leftarrow i+h-1; \\ M[i,j] \leftarrow \infty; \\ \mid \text{for } k \leftarrow i \, \text{to } j-1 \, \text{do} \\ \mid t \leftarrow M[i,k]+M[k+1,j]+c[i-1] \cdot c[k] \cdot c[j]; \\ \mid \text{if } t < M[i,j] \, \text{then} \\ \mid M[i,j] \leftarrow t; \\ S[i,j] \leftarrow k; \end{array}
```

Algorithm 15: stampaPar(integer[][] *S*, integer *i*, integer *j*)

```
if i = j then | print 'A(i'; print i; print ')'; else | print 'A(i'; print i; print
```

S. Filippone Ing. Alg. 10 / 21



II problema dello zaino — Memoization

Siete in una casa che sta andando a fuoco, avete uno zaino e dovete salvare oggetti del maggior valore possibile.

S. Filippone Ing. Alg. 11/21



II problema dello zaino — Memoization

Siete in una casa che sta andando a fuoco, avete uno zaino e dovete salvare oggetti del maggior valore possibile.

Knapsack (0-1)

Dati n oggetti, ciascuno caratterizzato da un valore p_i e da un volume v_i , ed uno zaino di capacità C (tutti dati interi), trovare un sottoinsime $S \subseteq \{1, \ldots, n\}$ tale che

$$\max_{\text{over all } S} p(S) = \sum_{i \in S} p_i, \qquad v(S) = \left(\sum_{i \in S} v_i\right) \leq C.$$

Definendo S(i,c) la soluzione ottima del problema P(i,c) con i primi i oggetti e capacità c, allora

- $i \in S(i,c)$: Allora $S(i,c) \{i\}$ è la soluzione ottima per $P(i-1,c-v_i)$;
- $i \notin S(i,c)$: Allora S(i,c) è la soluzione ottima per P(i-1,c).

S. Filippone 11/21 Ing. Alg.

II problema dello zaino — Memoization

Algorithm 16: knapsack(integer[] p, integer[] v, integer i, integer c, integer [][] D)

```
if i = 0 or c = 0 then
    Risultato 0
if c < 0 then
    Risultato -\infty
if D[i, c] = \perp then
    D[i, c] \leftarrow \max(knapsack(p, v, i-1, c, D), knapsack(p, v, i-1, c-v[i], D) + p[i]);
Risultato D[i, c]
```

Complessità nel caso peggiore: O(nC), ma potrebbe risolvere in meno passaggi.

L'inizializzazione della tabella D costa comunque O(nC), ma se ci aspettiamo di toccare pochi elementi, potrebbe essere utile usare una tabella hash.

Si noti che C è un dato del problema, che quindi non è polinomiale; ma se $C = O(n^k)$ allora lo ridiventa: si dice problema pseudo-polinomiale.



La più lunga sottosequenza comune (LCS)

In biologia, il DNA è una lunga molecola costituita da una serie di basi (Adenina, Citosina, Guanina, Timina), p.es.

ACCGGTTCGAGTGCGCGGAAGCCGGCCGAA

Le basi vengono "lette" a gruppi di tre e codificano per gli aminoacidi che costituiscono le proteine; questo meccanismo è il fondamento della ereditarietà in tutti gli organismi viventi.

AAA	Lisina	AAC	Asparagina
ACT	Treonina	ATT	Isoleucina
TCT	Serina	TAT	Tirosina

Uno dei problemi fondamentali della biologia è di definire e misurare una distanza tra le sequenze di DNA di due organismi; la caratterizzazione statistica delle differenze tra due sequenze consente poi di ricostruire degli alberi filogenetici che descrivono le relazioni evolutive fra i diversi organismi.

La più lunga sottosequenza comune (LCS)

Teorema

Sottostruttura ottima di LCS Se $X = \langle x_1, x_2, \dots, x_m \rangle$ e $Y = \langle y_1, y_2, \dots, y_n \rangle$ sono due sequenze, e $Z = \langle z_1, z_2, \dots, z_k \rangle$ una LCS di X e Y di lunghezza k, allora:

- Se $x_m = y_n$ allora $z_k = x_m = y_n$ e Z_{k-1} è LCS di X_{m-1} e Y_{n-1} ;
- 2 Se $x_m \neq y_n$ allora $z_k \neq x_m$ implica $Z \in LCS$ di $X_{m-1} \in Y$;
- § Se $x_m \neq y_n$ allora $z_k \neq y_n$ implica $Z \in LCS$ di $X \in Y_{n-1}$:

Dimostrazione.

- ① Se fosse $z_k \neq x_m$, allora si potrebbe accodare $x_m = y_n$ a Z ottenendo una sottosequenza di lunghezza k+1, contro l'ipotesi:
- ② Se $z_k \neq x_m$, allora Z è una sottosequenza di X_{m-1} e Y, e se ce ne fosse una più lunga, sarebbe anche sottosequenza di $X \in Y$, contro l'ipotesi di lunghezza massima;
- § Se $z_k \neq v_n$, vale un argomento simmetrico al precedente.



La più lunga sottosequenza comune (LCS)

Quindi il calcolo della sottoseguenza ottima si basa sulla formulazione

$$c[i,j] = \begin{cases} 0 & \text{se } i = 0 \text{ o } j = 0 \\ c[i-1,j-1] + 1 & \text{se } i,j > 0 \text{ e } x_i = y_j \\ \max(c[i-1,j],c[i,j-1]) & \text{se } i,j > 0 \text{ e } x_i \neq y_j \end{cases}$$

Il calcolo della lunghezza della sottosequenza massima si può eseguire agevolmente con una procedura che visita la tabella c[] in ordine; per ricostruire la sequenza, occorre ripercorrere la tabella "all'indietro", magari con l'uso di una tabella ausiliaria; in ogni caso si ha una complessità $\Theta(mn)$.

S. Filippone Ing. Alg. 15 / 21



La progettazione del software

Gli algoritmi devono essere tradotti in software, espressi in un qualche linguaggio di programmazione; i corsi di ingegneria del software sono dedicati a vari aspetti di questa attività. Qui accenniamo molto brevemente ad alcuni principi fondamentali:

- Separazione delle responsabilità;
- Astrazione:
- Incapsulamento,

che convergono nella attività di *Programmazione Modulare*

S. Filippone Ing. Alg. 16 / 21



Separazione delle responsabilità

Un importante principio nella progettazione è la raccomandazione di decomporre il software stesso in unità:

- Ciascuna unità dovrebbe eseguire una sola funzione (responsabilità);
- Funzionalità avanzate si possono ottenere componendo unità più piccole:
- Ciascuna unità può essere realizzata e collaudata separatamente.

Ad esempio, una regola fondamentale è di separare e segregare in apposite funzioni l'I/O (ossia, il dialogo con il mondo esterno), e di far comunicare le funzioni di calcolo solo attraverso i loro argomenti.

S. Filippone Ing. Alg.



Astrazione ed incapsulamento

Astrazione

- Il processo di semplificazione di un progetto nelle sue componenti principali;
- Il progetto può essere compreso più facilmente nelle sue linee generali. senza essere oberati dai dettagli;
- Lo sviluppatore si può riservare di modificare la implementazione se necessario;

Incapsulamento

- Metodo che tende a nascondere i dettagli della realizzazione interna, presentando all'utente una interfaccia che descrive il comportamento atteso;
- La separazione tra funzionalità e realizzazione è uno strumento essenziale (lo sviluppatore può sempre cambiare l'algoritmo interno)
- L'utente non vede una modifica della funzionalità (ma potrebbe vedere p.es. un miglioramento delle performance)



Programmazione modulare

Criteri:

Decomponibilità Si può decomporre il problema in sottoproblemi?

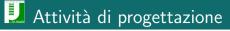
Componibilità Si può risolvere il problema combinando componenti esistenti?

Comprensibilità Si riesce a migliorare la comprensione del programma?

Continuità Un piccolo cambiamento nelle specifiche genera un cambiamento altrettanto piccolo nella realizzazione?

Protezione Un errore in un modulo ha effetto anche altrove?

S. Filippone Ing. Alg. 19 / 21



Progettazione della interfaccia:

- L'interfaccia di ciascun modulo dovrebbe consentire una comunicazione senza ambiguità con gli altri;
- L'interfaccia dovrebbe essere separata dai meccanismi interni, per garantire che la loro modifica non influenzi il funzionamento degli altri moduli;
- Al di là della disciplina del singolo programmatore, esistono strumenti per incoraggiare queste pratiche; la programmazione orientata agli oggetti ne è un esempio.

S. Filippone Ing. Alg. 20 / 21



Progettazionie dei Componenti allocazione delle funzionalità ai componenti, e specifica delle interfacce;

Progettazione delle strutture dati Specifica di dettaglio delle strutture dati;

Progettazione degli algoritmi Specifica di dettagli degli algoritmi

S. Filippone Ing. Alg. 21/21