Algoritmi e Strutture Dati

Tecniche Algoritmiche

Basato su materiale di C. Demetrescu, I. Finocchi, G.F. Italiano

Divide et impera

Divide et impera

- Tecnica top-down:

1. Dividi l'istanza del problema in due o più sottoistanze
2. Risolvi ricorsivamente il problema sulle sottoistanze
3. Ricombina la soluzione dei sottoproblemi allo scopo di ottenere la soluzione globale

- Esempi: ricerca binaria, mergesort, quicksort

Teorema Master:

un strumento generale per risolvere equazioni di ricorrenza

L'equazione di ricorrenza:

$$T(n) = \begin{cases} a\,T(n/b) + f(n) & \text{se } n>1 \\ 1 & \text{se } n=1 \end{cases}$$

ha soluzione:

1. $T(n) = \Theta(n^{\log_b a})$ se $f(n)=O(n^{\log_b a-\varepsilon})$ per $\varepsilon>0$
2. $T(n) = \Theta(n^{\log_b a} \log n)$ se $f(n) = \Theta(n^{\log_b a})$
3. $T(n) = \Theta(f(n))$ se $f(n)=\Omega(n^{\log_b a+\varepsilon})$ per $\varepsilon>0$ e $a\,f(n/b)\leq c\,f(n)$ per $c<1$ e n abbastanza grande

Esempi

1) T(n) = n + 2T(n/2)
   a=2, b=2, f(n)=n=$\Theta(n^{\log_2 2})$ ➡ T(n)=Θ(n log n)
   (caso 2 del teorema master)

2) T(n) = c + 3T(n/9)
   a=3, b=9, f(n)=c=$O(n^{\log_9 3 - \varepsilon})$ ➡ T(n)=$\Theta(\sqrt{n})$
   (caso 1 del teorema master)

3) T(n) = n + 3T(n/9)
   a=3, b=9, f(n)=n=$\Omega(n^{\log_9 3 + \varepsilon})$
   $3(n/9) \leq c\, n$ per $c=1/3$ ➡ T(n)=Θ(n)
   (caso 3 del teorema master)

Moltiplicazione di interi di grandezza arbitraria

- Moltiplicare o sommare due numeri non sempre è un'operazione eseguibile in tempo costante:
  - se ogni numero occupa più di una parola di memoria, anche le operazioni elementari potrebbero richiedere tempo superiore a O(1)
- Moltiplicare due numeri a *n* cifre, con l'algoritmo appreso in scuola elementare, richiede tempo $O(n^2)$
  - possiamo fare di meglio?

Rappresentazione decimale di interi a *n* cifre

$$X = x_{n-1}x_{n-2}\ldots x_1x_0 = \sum_{i=0}^{n-1} x_i \cdot 10^i, \qquad x_i \in \{0,1,\ldots,9\}$$

$$Y = y_{n-1}y_{n-2}\ldots y_1y_0 = \sum_{i=0}^{n-1} y_i \cdot 10^i, \qquad y_i \in \{0,1,\ldots,9\}$$

- Dividiamo *X* e *Y* a metà:
  - X_1 (Y_1) = *n*/2 cifre più significative di *X* (*Y*)
  - X_0 (Y_0) = *n*/2 cifre meno significative di *X* (*Y*)

Divisione del problema

$$X = X_1 \cdot 10^{\frac{n}{2}} + X_0$$

$$Y = Y_1 \cdot 10^{\frac{n}{2}} + Y_0$$

$$\begin{aligned} X \cdot Y &= (X_1 \cdot 10^{\frac{n}{2}} + X_0) \cdot (Y_1 \cdot 10^{\frac{n}{2}} + Y_0) \\ &= (X_1 \cdot Y_1) \cdot 10^n + (X_1 \cdot Y_0 + X_0 \cdot Y_1) \cdot 10^{\frac{n}{2}} + X_0 \cdot Y_0 \end{aligned}$$

Osservazioni

$$X \cdot Y = (X_1 \cdot Y_1) \cdot 10^n + (X_1 \cdot Y_0 + X_0 \cdot Y_1) \cdot 10^{\frac{n}{2}} + X_0 \cdot Y_0$$

- Operazioni semplici
  - $Z \cdot 10^k$ = shift a sinistra delle cifre del numero Z di k posizioni
  - Somma di due numeri di k cifre: tempo $O(k)$
- Moltiplicare due numeri a n cifre si riduce quindi a:
  - **4** moltiplicazioni di numeri a **$n/2$** cifre
  - un numero costante di operazioni implementabili in tempo $O(n)$

Equazione di ricorrenza

$$T(n) = \begin{cases} 4T(n/2) + O(n) & \text{se } n > 1 \\ 1 & \text{se } n = 1 \end{cases}$$

- La soluzione è $\Theta(n^2)$ dal Teorema Master
- Cosa abbiamo guadagnato?

Ridurre il numero di moltiplicazioni

$$X \cdot Y = (X_1 \cdot Y_1) \cdot 10^n + (X_1 \cdot Y_0 + X_0 \cdot Y_1) \cdot 10^{\frac{n}{2}} + X_0 \cdot Y_0$$

- Detti:

$$\begin{aligned} P_1 &= (X_1 + X_0) \cdot (Y_1 + Y_0) \\ P_2 &= (X_1 \cdot Y_1) \\ P_3 &= (X_0 \cdot Y_0) \end{aligned}$$

si ha:

$$(X_1 \cdot Y_0 + X_0 \cdot Y_1) = P_1 - X_1 \cdot Y_1 - X_0 \cdot Y_0$$

$$X \cdot Y = P_2 \cdot 10^n + (P_1 - P_2 - P_3) \cdot 10^{\frac{n}{2}} + P_3$$

Analisi

$$X \cdot Y = P_2 \cdot 10^n + (P_1 - P_2 - P_3) \cdot 10^{\frac{n}{2}} + P_3$$

- Eseguiamo **3** moltiplicazioni e un numero costante di somme e shift

$$T(n) = \begin{cases} 3T(n/2) + O(n) & \text{se } n > 1 \\ 1 & \text{se } n = 1 \end{cases}$$

- La soluzione è $\Theta(n^{\log_2 3}) = \Theta(n^{1.59})$

(dal Teorema Master)

Programmazione dinamica

Programmazione dinamica

Tecnica bottom-up:

1. Identifica dei sottoproblemi del problema originario, procedendo logicamente dai problemi più piccoli verso quelli più grandi
2. Utilizza una tabella per memorizzare le soluzioni dei sottoproblemi risolti:
   quando si incontrerà di nuovo lo stesso sottoproblema, basterà esaminare la tabella
3. Si usa quando i sottoproblemi non sono indipendenti, e lo stesso sottoproblema può apparire più volte

Esempio: numeri di Fibonacci

Distanza tra stringhe

Siano *X* e *Y* due stringhe di lunghezza *m* ed *n*:

$$X = x_1 \cdot x_2 \cdot \cdots \cdot x_m \qquad Y = y_1 \cdot y_2 \cdot \cdots \cdot y_n$$

Vogliamo calcolare la "distanza" tra *X* e *Y:* il minimo numero delle seguenti operazioni elementari che permetta di trasformare X in Y

`inserisci`(a): Inserisci il carattere a nella posizione corrente della stringa.

`cancella`(a): Cancella il carattere a dalla posizione corrente della stringa.

`sostituisci`(a, b): Sostituisci il carattere a con il carattere b nella posizione corrente della stringa.

Esempio

Distanza fra RISOTTO e PRESTO

Azione	Costo	Stringa ottenuta
Inserisco P	1	P|*RISOTTO*
Mantengo R	0	PR|*ISOTTO*
Sostituisco I con E	1	PRE|*SOTTO*
Mantengo S	0	PRES|*OTTO*
Cancello O	1	PRES|*TTO*
Mantengo T	0	PREST|*TO*
Cancello T	1	PREST|*O*
Mantengo O	0	PRESTO|

Approccio

- Denotiamo con $\delta(X,Y)$ la distanza tra X e Y
- Definiamo X_i il prefisso di X fino all'i-esimo carattere, per 0≤i ≤m (X_0 denota la stringa vuota):

$$X_i = x_1 \cdot x_2 \cdot \cdots \cdot x_i \text{ se } i \geq 1$$

- Risolviamo il problema di calcolare $\delta(X,Y)$ calcolando $\delta(X_i,Y_j)$ per ogni i, j tali che $0 \leq i \leq m$ e $0 \leq j \leq n$
- Manterremo le informazioni in una tabella D di dimensione $m \times n$

Inizializzazione della tabella

- Alcuni sottoproblemi sono molto semplici
  - $\delta(X_0,Y_j)=j$ partendo dalla stringa vuota X_0, basta **inserire** uno ad uno i j caratteri di Y_j
  - $\delta(X_i,Y_0)=i$ partendo da X_i, basta **cancellare** uno ad uno gli i caratteri per ottenere Y_0
- Queste soluzioni sono memorizzate rispettivamente nella prima riga e nella prima colonna della tabella D

Avanzamento nella tabella (1/2)

- Se $x_i=y_j$, il minimo costo per trasformare X_i in Y_j è uguale al minimo costo per trasformare X_{i-1} in Y_{i-1}

D [i,j] = D [i-1, j-1]

- Se $x_i \neq y_j$, distinguiamo in base all'ultima operazione usata per trasformare X_i in Y_j in una sequenza ottima di operazioni:
  - inserimento
  - cancellazione
  - sostituzione

Avanzamento nella tabella (2/2)

`inserisci`(y_j): Il minimo costo per trasformare X_i in Y_j è pari al minimo costo per trasformare X_i in Y_{j-1} più 1 per inserire y_j

⇨ D [i,j] = **1** + D [i, j-1]

`cancella`(x_i): Il minimo costo per trasformare X_i in Y_j è pari al minimo costo per trasformare X_{i-1} in Y_j più 1 per cancellare x_i

⇨ D [i,j] = **1** + D [i-1, j]

`sostituisci`(x_i, y_j): Il minimo costo per trasformare X_i in Y_j è pari al minimo costo per trasformare X_{i-1} in Y_{j-1} più 1 per sostituire il carattere x_i con y_j

⇨ D [i,j] = **1** + D [i-1, j-1]

$$D[i,j] = \begin{cases} D[i-1,j-1] & \text{se } x_i = y_j \\ 1 + \min\{D[i,j-1], D[i-1,j], D[i-1,j-1]\} & \text{se } x_i \neq y_j \end{cases}$$

Pseudocodice

algoritmo `distanzaStringhe`$(stringa\ X, stringa\ Y) \rightarrow intero$
  matrice D di $(m+1) \times (n+1)$ interi
  for $i = 0$ **to** m **do** $D[i,0] \leftarrow i$
  for $j = 1$ **to** n **do** $D[0,j] \leftarrow j$
  for $i = 1$ **to** m **do**
    for $j = 1$ **to** n **do**
      if $(x_i \neq y_j)$ **then**
        $D[i,j] \leftarrow 1 + \min\{D[i,j-1], D[i-1,j], D[i-1,j-1]\}$
      else $D[i,j] \leftarrow D[i-1,j-1]$
  return $D[m,n]$

Tempo di esecuzione ed occupazione di memoria: O(m·n)

Esempio

		P	R	E	S	T	O
	0	**1**	2	3	4	5	6
R	1	1	**1**	2	3	4	5
I	2	2	2	**2**	3	4	5
S	3	3	3	3	**2**	3	4
O	4	4	4	4	**3**	3	3
T	5	5	5	5	**4**	**3**	4
T	6	6	6	6	5	**4**	4
O	7	7	7	7	6	5	**4**

La tabella D costruita dall'algoritmo.

In grassetto sono indicate due sequenze di operazioni che permettono di ottenere la distanza tra le stringhe

Tecnica golosa (o greedy)

Tecnica golosa e problemi di ottimizzazione

- La tecnica è usata per risolvere problemi di ottimizzazione
  - Ad ogni problema è associato un costo/valore
  - una soluzione è frutto di una sequenza di scelte, ciascuna delle quali contribuisce a determinare il costo/valore finale
  - si è interessati a trovare una soluzione che abbia un costo/valore ottimo (minimo o massimo)
- Strategia:
  - Ad ogni passo si sceglie la soluzione ottima o "greediest" (più golosa) localmente, senza preoccuparsi degli effetti successivi di tale scelta
  - La soluzione locale scelta può dipendere dalle scelte precedenti.
- Es: "Restituire il minor numero di monete di resto, usando monete da 100, 10, 1 Euro"
  - ad ogni passo viene controllato il resto ancora da dare e si aggiunge la moneta con valore maggiore possibile.
  - per restituire 112 Euro la macchina farà cadere in sequenza una moneta da 100, poi 10, poi 1 e poi 1 Euro (per un totale di 4 monete)

Esempi di problemi di ottimizzazione

- Allocazione di risorse (merci in un magazzino)
- Scheduling (ordinamento temporale)
- Pianificazione di investimenti
- Pattern matching: date due sequenze di simboli:

  AACCGATGTACCT

  CGAACGATACGGTTAC

  trovare la piu' lunga sottosequenza comune
- Distanza minima in reti: dato un insieme di città collegate da strade trovare il percorso minimo che collega ogni coppia di città

Tecnica golosa su insiemi

Vogliamo trovare la migliore soluzione a un problema (e.g., il cammino più corto per andare da Napoli a Torino)

1. Insieme di **`candidati`** (e.g., collegamenti fra coppie di città)
2. Insieme dei candidati già esaminati
3. Funzione **`ammissibile`**: verifica se un insieme di candidati è una soluzione, anche non ottima (un insieme di collegamenti fra città è un cammino?)
4. Funzione **`ottimo`**: verifica se un insieme di candidati è una soluzione ottima (l'insieme di collegamenti è un cammino minimo da Napoli a Torino?)
5. Funzione **`seleziona`**: indica quale dei candidati non ancora esaminati è al momento il più promettente

E' necessario definire una funzione obiettivo da minimizzare o massimizzare (e.g., lunghezza del cammino da Napoli a Torino)

Tecnica golosa su insiemi: schema algoritmico generale

algoritmo `paradigmaGreedy`$(insieme\ di\ candidati\ C) \rightarrow soluzione$
  $S \leftarrow \emptyset$
  while ((**not** `ottimo`(S)) **and** $(C \neq \emptyset)$) **do**
    $x \leftarrow$ `seleziona`(C)
    $C \leftarrow C - \{x\}$
    if (`ammissibile`$(S \cup \{x\})$) **then** $S \leftarrow S \cup \{x\}$
  if (`ottimo`(S)) **then return** S
  else errore non ho trovato soluzioni

Perché goloso?
sceglie sempre il candidato più promettente!

Un problema di sequenziamento

- Un server (e.g., una CPU, o un impiegato dell'ufficio postale) deve servire n clienti
- Il servizio richiesto dall'i-esimo cliente richiede t_i secondi
- Chiamiamo T(i) il tempo di attesa del cliente i
- Vogliamo minimizzare il tempo di attesa medio, i.e.:

$$T_{avg} = \frac{T}{n} = \frac{1}{n}\sum_{i=1}^{n} T(i) \qquad T = \sum_{i=1}^{n} T(i)$$

Un problema di sequenziamento: Esempio

$$t_1 = 50 \text{ msec}, \quad t_2 = 100 \text{ msec}, \quad t_3 = 3 \text{ msec}$$

Dei sei possibili ordinamenti, il migliore è evidenziato in blu

Ordine	T			
1 2 3	$50 + (50 + 100) + (50 + 100 + 3)$	msec	=	353 msec
1 3 2	$50 + (50 + 3) + (50 + 3 + 100)$	msec	=	256 msec
2 1 3	$100 + (100 + 50) + (100 + 50 + 3)$	msec	=	403 msec
2 3 1	$100 + (100 + 3) + (100 + 3 + 50)$	msec	=	356 msec
3 1 2	$3 + (3 + 50) + (3 + 50 + 100)$	msec	=	209 msec
3 2 1	$3 + (3 + 100) + (3 + 100 + 50)$	msec	=	259 msec

Un algoritmo goloso per il sequenziamento

- Il seguente algoritmo genera l'ordine di servizio in maniera incrementale secondo una strategia greedy.
- Supponiamo di aver deciso di sequenziare i clienti i_1, i_2, … i_m. Se decidiamo di servire il cliente j, il tempo totale di servizio diventa:

$$t_{i_1} + t_{i_2} + \ldots + t_{i_m} + t_j$$

- Quindi al generico passo l'algoritmo serve la richiesta più corta tra quelle rimanenti
- Tempo di esecuzione: O(n log n), per ordinare le richieste

Tre tecniche algoritmiche efficienti

- Divide et impera
  - applicazioni nella ricerca binaria, mergesort, quicksort, moltiplicazione di matrici
- Programmazione dinamica
  - applicazioni nel calcolo dei numeri di Fibonacci, prodotto tra matrici, cammini minimi tra tutte le coppie di nodi (algoritmo di Floyd e Warshall)
- Tecnica greedy
  - applicazioni nel distributore automatico di resto, nel calcolo del minimo albero ricoprente (algoritmi di Prim, Kruskal, Boruvka) e dei cammini minimi a sorgente singola (algoritmo di Dijkstra)

Due approcci estremi

- Ricerca esaustiva ("forza bruta")
  - Genera tutte le soluzioni teoricamente possibili fino a che non si trova quella effettivamente corretta
  - Trova sempre la soluzione, ma è dispendiosa
- Backtracking (ritorno all'indietro)
  - Si applica a problemi in cui devono essere soddisfatti dei **vincoli**
  - Si considerano successivamente tutte le possibili soluzioni, scartando man mano quelle che non soddisfano i vincoli.
  - Una tecnica classica consiste nell'esplorazione di strutture ad albero e tenere traccia di tutti i nodi e i rami visitati in precedenza
    - così da poter tornare indietro al più vicino nodo associato ad un cammino ancora inesplorato, se la ricerca nel ramo attuale non ha successo