1 Divide-et-impera

1.1 Risoluzione problemi

Dato un problema, non si hanno metodi generali per risolverlo in modo efficiente, tuttavia è possibile evidenziare 4 fasi:
1. Classificazione del problema;
2. Caratterizzazione della soluzione;
3. Tecnica di progetto;
4. Utilizzo di strutture dati.

Queste fasi non sono necessariamente sequenziali.

1.1.1 Classificazione dei problemi

Problemi decisionali Il dato in ingresso soddisfa una certa proprietà? **Soluzione**: risposta si/no. Un problema di esempio può essere stabilire se un grafo è connesso.

Problemi di ricerca

Problemi in cui si ha:
- **Spazio di ricerca:** insieme di "soluzioni" possibili;
- **Soluzione ammissibile:** soluzione che rispetta certi vincoli;

Ad esempio la posizione di una sotto-stringa in una stringa.

Problemi di ottimizzazione

Problemi in cui ogni soluzione è associata ad una funzione di costo e si vuole la soluzione di costo minimo. ad esempio il cammino più breve tra due nodi (grafi pesati).

1.1.2 Definizione matematica del problema

Una cosa fondamentale da fare è definire bene il problema in modo formale. Spesso la formulazione è banale, ma può suggerire una prima idea di soluzione. Ad esempio, data una sequenza di *n* elementi, una permutazione ordinata è data dal minimo seguito da una permutazione ordinata degli *n - 1* elementi (Selection Sort). La definizione matematica del problema può suggerire una possibile tecnica di soluzione.
Sottostruttura ottima → Programmazione dinamica;
Proprietà greedy → Tecnica greedy;

1.1.3 Tecnica di soluzione dei problemi

Divide-et-impera

Un problema viene suddiviso in sotto-problemi indipendenti, che vengono risolti ricorsivamente (top-down).
Ambito: problemi di decisione, ricerca.

Top-down

La strategia top-down è una strategia in cui un problema viene diviso in parti sempre più piccole fino a che non si ottengono componenti semplici da implementare.
Top-down è diverso da divide-et-impera, dato che quest'ultimo fa uso di top-down, che è una strategia di progettazione, mentre divide-et-impera è una tecnica algoritmica specifica con una struttura ben definita.

Programmazione dinamica

La soluzione viene costituita (bottom-up) a partire da un insieme di sotto-problemi potenzialmente ripetuti.
Ambito: problemi di ottimizzazione.

Bottom-up

Bottom-up, al contrario di top-down, parte da componenti più piccole e riutilizzabili e le combina per ottenere funzionalità più complesse fino ad ottenere un intero sistema.
Come funziona:

1. Si implementano piccole parti indipendenti;
2. Le si mettono insieme per formare parti più grandi;
3. Si continuano a comporre finché non si ottiene l'algoritmo/programma completo.

Memoization (o annotazione)

Versione top-down della programmazione dinamica.

Memoization (o annotazione)

Versione top-down della programmazione dinamica.

Tecnica greedy

Approccio "ingordo": si fa sempre la scelta localmente ottima.

Backtrack

Si procede per "tentativi" e si torna di tanto in tanto sui propri passi.

Ricerca locale

La soluzione ottima viene trovata "migliorando" via via soluzioni esistenti.

Algoritmi probabilistici

Meglio scegliere con giudizio (in maniera costosa) o scegliere a caso "gratuitamente".

1.2 Divide-et-impera

Questa tecnica, applicata alla risoluzione di un problema computazionale, consiste nel partizionare il problema in sotto-problemi più piccoli dello stesso tipo e indipendenti, risolverli ricorsivamente, e successivamente ricombinare, con poco sforzo, le soluzioni parziali per ottenere la soluzione del problema originale.
Lo "schema" di una procedura ricorsiva `divideEtImpera()` per risolvere un problema di *P* di dimensione *n* è il seguente, dove *k* è una costante intera prefissata:

divideEtImpera(*P*, **integer** n)

if $n \leq k$ **then**
    risolvi P direttamente
else
    dividi P in h sottoproblemi $P_1, ..., P_h$ di dimensione $n_1, ..., n_h$
    for i $\leftarrow 1$ **to** h **do** divideEtImpera(P_i,n_i)
    combina i risultati di $P_1, ..., P_h$ per ottenere quello di P
end if

La tecnica *divide-et-impera* permette di provare agevolmente la correttezza dell'algoritmo usando il principio di induzione e di impostarne facilmente le relazioni ricorrenti della funzione *T(n)* di complessità:

$$T(n) = \begin{cases} c, & n \leq k \\ D(n) + C(n) + \sum_{i=1}^{h} T(n_i), & n > k \end{cases}$$

dove c è una costante, *D(n)* è il numero di operazioni per dividere il problema e *C(n)* è il numero di operazioni per combinare i risultati.

Nota

Se i dati sono partizionati in maniera bilanciata, cioè tutti gli n_i sono all'incirca uguali, allora l'algoritmo può risultare molto efficiente.

1.3 Minimo: Divide-et-impera

Si applichi ora il metodo divide-et-impera a un problema di ricerca del valore minimo in un array *A*:

int minrec(**int**[] A, **int** i, **int** j)

if $i == j$ **then**
    return A[i]
else
    m=$\lfloor (i+j)/2 \rfloor$
    return min(minrec(A,i,m), minrec(A, m+1, j)
end if

parte intera inferiore e parte intera superiore

$\lfloor x \rfloor$ rappresenta la parte intera inferiore, spesso chiamata "floor", ovvero se si ha un numero reale esso, se seguito da virgola, viene approssimato per difetto, ad esempio: $\lfloor 3,9 \rfloor = 3$.
Mentre, al contrario $\lceil x \rceil$ rappresenta la parte intera superiore, detta "ceiling", e si approssima per eccesso, ad esempio: $\lfloor 3,1 \rfloor = 4$.

Nell'algoritmo avviene che viene preso un valore medio della posizione dell'array *A*, con *i* che indica l'inizio di *A* e *j* che ne indica la fine, *m* è quindi l'indice che indica la metà dell'array. Ogni volta viene richiamata la funzione fino a quando non si arriva ad avere un singolo elemento nell'array, ovvero la condizione dell'"if".
L'array viene quindi diviso in tanti piccoli pezzi per trovare il minimo avendo una complessità:

$$T(n) = \begin{cases} 2T(n/2) + 1, & n > 1 \\ 1, & n = 1 \end{cases}$$

Analizzando la complessità dell'algoritmo, quindi, si ha un caso base, quando l'array contiene un solo elemento e quindi il costo risulta costante, perché basta restituire quell'elemento, ma nel caso ricorsivo con *n* elementi il costo è dato da $2T(n/2)$, ovvero due chiamate ricorsive, ciascuna su metà dell'input $(n/2)$ e $+1$ che rappresenta il costo costante delle operazioni di divisione (ovvero il calcolo di *m*) e di combinazione (confronto finale tra i due minimi).
Risulta che l'algoritmo divide-et-impera non è conveniente.

1.4 Esempio binary search

```
int binarySearch(int[] S, int v, int i, int j)
  if i > j then
      return 0
  else
      m = ⌊(i + j)/2⌋
      if S[m] == v then
          return m
      else if S[m] < v then
          return binarySearch(S, v, m + 1, j)
      else
          return binarySearch(S, v, i, m − 1)
      end if
  end if
```

La procedura binary search è un particolare tipo di divide-et-impera. Il problema viene suddiviso in 2 sottoproblemi (h=2) approssimativamente uguali, mentre i costi $C(n)$ e $D(n)$ sono costanti. La particolarità sta nel fatto che la chiamata ricorsiva avviene in solo uno dei due sottoproblemi, invece che su entrambi.
Si può considerare un esempio "semplificato" di divide-et-impera, ma permette di enunciare una regola importante in questi casi: **se non tutti i sottoproblemi devono essere analizzati, è buona norma affrontare ricorsivamente i problemi più piccoli possibili**.
La complessità della ricerca può essere ulteriormente abbassata tenendo conto dei valori delle chiavi memorizzate nel vettore e della loro **distribuzione di probabilità**.

Esempio di ricerca: Dizionario

Se si vuole cercare la parola "casa" nel dizionario, non lo si apre a metà ma a circa $\frac{1}{4}$ del numero di pagine.
Si consideri una rappresentazione a vettore, con n chiavi numeriche e distribuite uniformemente nell'intervallo [k_{min}, k_{max}]. Dovendo cercare le chiavi k in $S[1...n]$, si tenta la ricerca in una posizione "ragionevolmente" vicina (non al centro), data da: $n * \frac{k-k_{min}}{k_{max}-k_{min}}$.
Il numeratore fornisce lo scarto tra i valori della chiave da cercare e quella più piccola, mentre il denominatore fornisce lo scarto tra i valori della chiave più grande e quella più piccola. Il loro rapporto indica quanto k si discosta dagli estremi.
Se $k = k_{min}$, allora il rapporto è 0 e conviene quindi cercare nella prima posizione del vettore, se $k = k_{max}$, il rapporto è 1 e allora conviene cercare nell'ultima posizione del vettore. La ricerca binaria "ignora" k, assume che il rapporto sia pari a $\frac{1}{2}$ e tenta in posizione centrale. Il metodo di ricerca risultante è detto di **interpolazione**.
La procedura di ricerca binaria `binarySearch()` può essere trasformata in una di interpolazione semplicemente sostituendo l'istruzione:

- $m \leftarrow \lfloor (i+j)/2 \rfloor$

con

- $m \leftarrow i + \lfloor (k - A[i]) * (j-i)/(A[j] - A[i]) \rfloor$

Infatti la ricerca viene effettuata in generale sulla porzione $A[i...j]$ del vettore, che contiene la più piccola chiave in $A[i]$ e la più grande in $A[j]$. La formula per m si ricava da quella ricerca

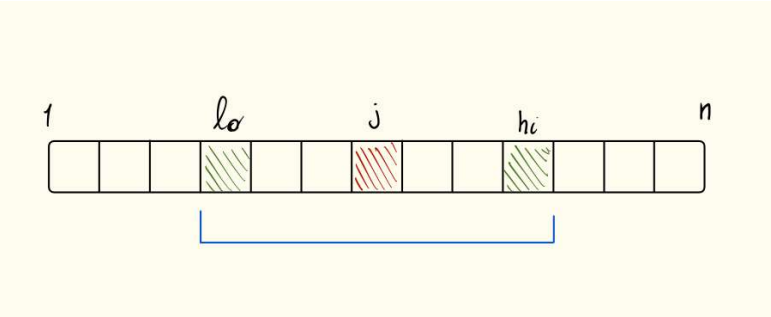
binaria, che era $i + \lfloor (j-i)/2 \rfloor$, sostituendo $(k - A[i]) * (j-i)/(A[j] - A[i])$ ad $\frac{1}{2}$.

Complessità

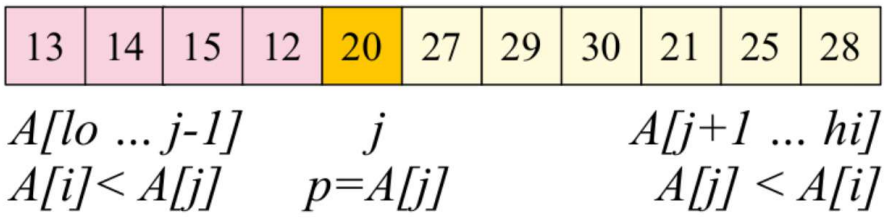
Con distribuzione uniforme delle chiavi, è possibile dimostrare che la complessità è $O(log\, log\, n)$. La funzione $log\, log\, n$ cresce molto lentamente, ma è paragonabile a $log\, n$ per n piccolo. Pertanto, se ci sono poche chiavi, oppure se le chiavi non sono uniformemente distribuite, è più conveniente usare la ricerca binaria. Al contrario, conviene usare l'interpolazione se ci sono tante chiavi oppure se sono uniformemente distribuite.

1.5 Quicksort (Hoare, 1961)

L'algoritmo di QuickSort è l'algoritmo praticamente più efficiente per ordinare gli elementi di un vettore.
Questo è basato sulla tecnica *divide-et-impera*, ma differisce dal "MergeSort" nel modo in cui divide il problema e combina i risultati.
Questo algoritmo ha un caso medio: $O(n\, log\, n)$ e un caso pessimo: $O(n^2)$ che però viene evitato grazie a tecniche "euristiche" (ovvero, in generale, tecniche che non garantiscono di trovare la soluzione ottimale o perfetta, ma progettate per trovare una soluzione soddisfacente in un tempo ragionevole) e spesso è preferito ad altri algoritmi a causa di costanti moltiplicative più basse. Nella scelta, spesso si valuta anche la "stabilità" dell'ordinamento.
Si consideri un vettore $A[1, ..., n]$ con indici lo, hi tali che $1 \leq lo \leq hi \leq n$:

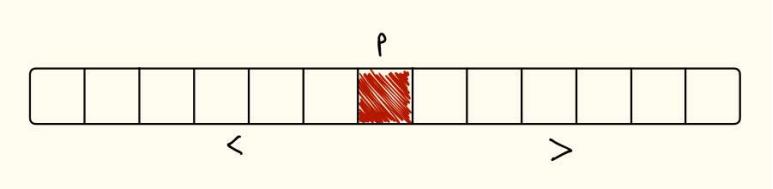


La parte evidenziata in blu rappresenta la parte del vettore che si vuole ordinare.
Essendo un algoritmo di "divide-et-impera" si deve prima dividere, che è la parte più complessa, quindi si identifica un valore $p \in A[lo, ..., hi]$ detto perno (*pivot*).
L'operazione di "divide" sposta tutti gli elementi del sottovettore a cui si fa riferimento in modo che il valore del perno sia poi posizionato in un certo punto del vettore e questo punto si indica con "j".



Tutti gli elementi più piccoli del valore del perno devono essere collocati prima del perno stesso e tutti gli elementi più grandi devono essere collocati dopo. I valori collocati prima e dopo non devono essere interamente ordinati perché si è ancora nella fase di "divide", ovvero

si porta il perno al centro in cui appunto la parte prima contiene solo valori minori di esso.
A questo punto "impera" ordina i due sottovettori $A[lo, ..., j-1]$ e $A[j+1, ..., hi]$, richiamando ricorsivamente il *QuickSort*. Queste due parti potrebbero avere dimensione 0 nel caso "j-1" sia più piccolo di "lo".
La parte del "combina" non fa nulla, questo perché:



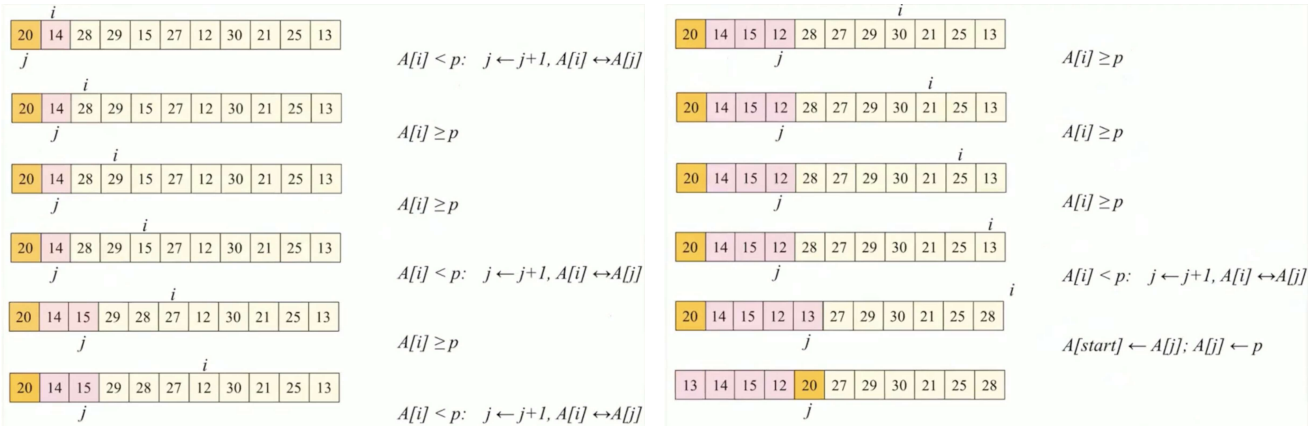
Tutti gli elementi più piccoli del perno stanno prima dello stesso, mentre tutti gli elementi più grandi stanno dopo ed entrambe le parti vengono riordinate tramite chiamate ricorsive sulle stesse parti. Alla fine tutti gli elementi sono ordinati nei sottovettori.

1.6 Pivot

```
int pivot(ITEM[] A, int lo, int hi)
  ITEM pivot = A[lo] int j=lo
  for i do=lo+1 to hi do
      if A[i] < pivot then j=j+1 swap(A, i, j)
      end if
  end for
  A[lo]=A[j]
  A[j]=pivot
  return j
```

```
swap(ITEM[] A, int i, int j)
  ITEM temp = A[i]
  A[i] = A[j]
  A[j] = temp
```

Questa versione del *pivot* sceglie come valore del pivot, appunto, quello che si trova in prima posizione. Il ciclo va dalla seconda casella fino ad arrivare all'ultima per poi uscire e cerca di capire cosa fare del valore in i. La parte del ciclo sposta i valori in modo che tutti quelli più grandi si trovano dopo e quelli piccoli prima, confrontando ogni valore A[i] col pivot. Se trova un valore più piccolo del pivot esso va prima del perno.
La parte dopo il ciclo mette il perno al centro.

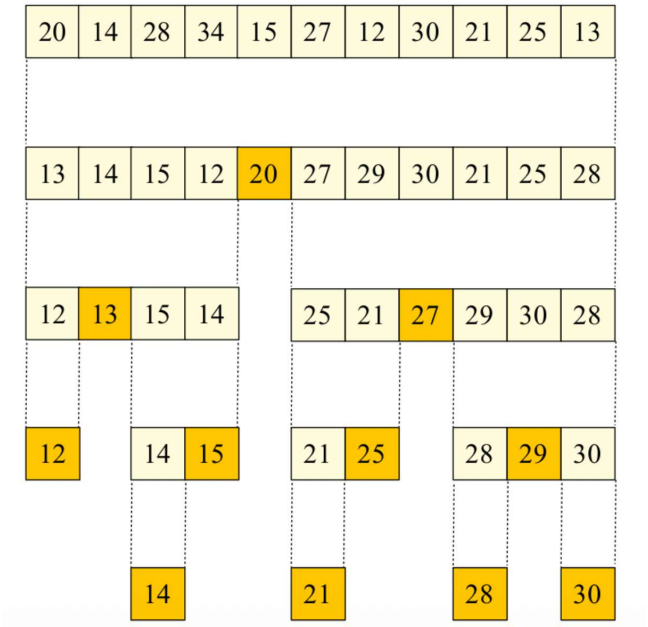


Il costo di *pivot()* è: $\theta(n)$.

1.7 Quicksort: procedura principale

```
QuickSort(ITEM[] A, int lo, int hi)
if lo<hi then
    int j=pivot(A, lo, hi)
    QuickSort(A, lo, j-1)
    QuickSort(A, j+1, hi)
end if
```

Svolgimento ricorsione:



1.7.1 Caso pessimo

Nel caso pessimo il vettore è già ordinato, o in maniera crescente o in maniera decrescente, e si ottiene un costo pari a: $T(n) = T(n-1) + T(0) + \theta(n) = T(n-1) + \theta(n) = \theta(n^2)$

1.7.2 caso ottimo

Se il perno è sempre i valore mediano, ovvero quello che ha un numero di valori più piccoli e più grandi pari, allora la scelta del perno permette di dividere il vettore in due sottoparti più o meno uguali. In questo caso, il vettore con n elementi, viene sempre diviso in due sottoproblemi di dimensione $n/2 \rightarrow T(n) = 2T(n/2) + \theta(n) = \theta(n\,log\,n)$.
Il problema è che il vettore non è sempre ben diviso in due parti. Il partizionamento nel caso medio di *QuickSort* è molto più vicino al caso ottimo che al caso peggiore (perché non ha senso ordinare un vettore già ordinato, quindi se si vuole effettuare un ordinamento ci saranno degli elementi sparsi nel vettore).
Nella realtà il caso medio dipende dall'ordine degli elementi e non dai loro valori, si devono considerare tutte le possibili permutazioni e può risultare difficile dal punto di vista analitico, ma si può arrivare a un'intuizione ovvero che alcuni partizionamenti saranno parzialmente bilanciati, mentre altri saranno pessimi e in media questi si alternano nella sequenza di partizionamento. I partizionamenti parzialmente bilanciati dominano quelli pessimi, ovvero dividono man mano il vettore in parti sempre più piccole.
Se si fa un'analisi probabilistica su tutte le possibili permutazioni si vede che, nel caso medio, il costo è $\theta(n\,log\,n)$, quindi nel caso medio l'algoritmo ha lo stesso costo computazionale del *MergeSort*.
I fattori moltiplicativi, anche considerando le partizioni più sfavorevli, sono comunque in favore del *QuickSort*.

1.8 Moltiplicazione di matrici

Siano A e B due matrici rispettivamente di dimensione $n_i * n_k$ e $n_k * n_j$, con n_k comune, e sia C il prodotto tra A e B, per calcolare $c_{i,j} = \sum_{k=1}^{n_k} a_{i,k} * b_{k,j}$: Per ogni $i \in [1, \ldots n_i]$ e per ogni

matrixProduct(**ITEM**[][] A, B, C, **int** n_i, n_k, n_j)

```
for i do=1 to n_i do
    for j do=1 to n_j do
        C[i,j]=0
        for k do=1 to n_k do
            C[i,j]= C[i,j]+A[i,k]*B[k,j]
        end for
    end for
end for
```

$j \in [1, \ldots n_j]$ si deve tradurre la sommatoria in codice e diventa un ulteriore ciclo che va da 1 a n_k in cui si va ad accumulare la sommatoria.
Avendo un ciclo su n_i, un ciclo su n_j e un ciclo su n_k, la complessità sarà: $T(n) = \theta(n_i * n_k * n_j)$ e se le matrici hanno tutte le stesse dimensioni viene fuori una complessità pari a $\theta(n^3)$.

1.8.1 Algoritmo di Strassen

Le matrici $n * n$ (quindi quadrate, ma si può fare anche con matrici rettangolari, aggiungendo degli zeri per renderle delle stesse dimensioni) vengono suddivise in quattro matrici $n/2 * n/2$.

$$A = \begin{bmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{bmatrix} \qquad B = \begin{bmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{bmatrix}$$

Si può calcolare la matrice come:

$$C = \begin{bmatrix} A_{1,1}B_{1,1} + A_{1,2}B_{2,1} & A_{1,1}B_{1,2} + A_{1,2}B_{2,2} \\ A_{2,1}B_{1,1} + A_{2,2}B_{2,1} & A_{2,1}B_{1,2} + A_{2,2}B_{2,2} \end{bmatrix}$$

Ma ci sono 8 moltiplicazioni matriciali, avendo come equazione di ricorrenza: $8T(n/2) + n^2$ con $n > 1$ e si ha sempre una complessità pari a $\theta(n^3)$.
Strassen ha trovato un modo per ridurre la complessità dividendo in ulteriori 7 sottomatrici:

$$\begin{cases} X_1 = (A_{11} + A_{22}) * (B_{11} + B_{22}) \\ X_2 = (A_{21} + A_{22}) * B_{11} \\ X_3 = A_{11} * (B_{12} - B_{22}) \\ X_4 = A_{22} * (B_{21} - B_{11}) \\ X_5 = (A_{11} + A_{12}) * B_{22} \\ X_6 = (A_{21} - A_1) * (B_{11} + B_{12}) \\ X_7 = (A_{12} - A_{22}) * (B_{21} + B_{22}) \end{cases}$$

Si trova un'equazione di ricorrenza:

$$T(n) = \begin{cases} 7T(n/2) + n^2, & n > 1 \\ 1, & n = 1 \end{cases} \qquad \rightarrow T(n) = \theta(n^{log_2 7}) \approx \theta(n^{2.81}) \tag{1}$$

riducendo la complessità e avendo come calcolo finale:

$$C = \begin{bmatrix} X_1 + X_4 - X_5 + X_7 & X_3 + X_5 \\ X_2 + X_4 & X_1 + X_3 - X_2 + X_6 \end{bmatrix}$$

Strassen è stato il primo a scoprire che era possibile moltiplicare in meno di n^3 moltiplicazioni scalari.

1.9 Conclusioni

Quando applicare *divide-et-impera*?

Quando i costi risultano essere migliori del corrispondente algoritmo iterativo.

Ulteriori vantaggi

È facile parallelizzare e c'è un utilizzo ottimale della cache.