12 Dividi et Impera

12.1 Risoluzione di problemi

Dato un problema non esistono "ricette originali" per risolverlo in modo efficiente; tuttavia è possibile evidenziare quattro fasi:

1. **classificazione del problema**: è il primo passo verso la risoluzione del problema;
2. **caratterizzazione della soluzione**: bisogna caratterizzare matematicamente la soluzione, evitando di evitare soluzioni banali;
3. **tecnica di progetto**: quando è possibile dividere il problema in più sottoproblemi di complessità minore allora la tecnica "dividi et impera" potrebbe essere quella più appropprata (più avanti vedremo delle tecniche più interessanti quali: programmazione dinamica (Capitolo 13), algoritmi ingordi (Capitolo 14) e backtrack (Capitolo 16));
4. **utilizzo di strutture dati**: bisogna scegliere la struttura dati più adatta alla risoluzione del nostro particolare problema (spesso sarà una tabella hash o un albero binario di ricerca, più avanti vedremo delle strutture dati specializzate per risolvere problemi specifici, a differenza di quelle che abbiamo visto fin'ora che sono generiche).

Queste fasi non sono necessariamente sequenziali, dipende da come stiamo affrontando il problema.

12.1.1 Classificazione dei problemi

Ma come possiamo classificare un problema? Le classi di problemi che affronteremo possono essere raggruppate in quattro macro-categorie:

- **problemi decisionali**: consistono nel determinare se il dato in ingresso soddisfa o meno una certa proprietà ed hanno una risposta binaria (si/no, true/false); come ad esempio stabilire se un grafo risulta connesso o meno. Su questo genere di problemi spesso non esistono delle tecniche standard e bisogna creare algoritmi ad-hoc;
- **problemi di ricerca**: consistono nel trovare nello spazio di soluzioni possibili una soluzione ammissibile che rispetti certi vincoli, come ad esempio la ricerca della posizione di una sottostringa in una stringa. In questi problemi la tecnica "dividi et impera" può ricorrere in nostro aiuto;
- **problemi di ottimizzazione**: ad ogni soluzione è associata una funzione di costo e vogliamo trovare quella di costo minimo, come ad esempio il cammino (pesato) più breve fra due nodi. Questa classe di problemi può essere risolta tramite la programmazione dinamica o algoritmi ingordi;
- **problemi di approssimazione**: a volte, trovare la soluzione ottima è computazionalmente impossibile e ci si accontenta di una soluzione approssimata, in questo caso il costo rimane basso ma non sappiamo se è ottimale; un esempio di questo genere di problemi è quello del commesso viaggiatore.

12.1.2 Caratterizzazione della soluzione

É fondamentale definire bene il problema dal punto di vista matematico. La formulazione del problema può suggerire una prima idea, seppur banale, alla risoluzione del problema. Lo si può osservazione nella formulazione del seguente problema: data una sequenza di n elementi, una permutazione ordinata è data dal minimo seguito da una permutazione ordinata dei restanti $n-1$ elementi. Il quale scaturisce l'algoritmo selectionSort. La definizione matematica può suggerire una possibile tecnica, ad esempio:

- se troveremo una *sottostruttura ottima* allora potremmo applicare la tecnica di programmazione dinamica (Capitolo 13);
- se troveremo la *proprietà greedy* allora potremmo applicare la tecnica di programmazione dinamica (Capitolo 14).

Tecniche di soluzione dei problemi

Come vengono affrontati i problemi dalle varie tecniche?

- Nella tecnica dividi-et-impera un problema viene suddiviso in sotto-problemi indipendenti, i quali vengono risolti ricorsivamente (avendo quindi un approccio dall'alto verso il basso, detto *top-down*); Abbiamo già visto diversi esempi dell'applicazione di questa tecnica, provate a pensare all'algoritmo mergeSort: ordinare due sottovettori sono due problemi indipendenti (ordinare il sottovettore di sinistra non richiede conoscere il contenuto del vettore di destra e viceversa);
- Nella programmazione dinamica la soluzione viene costruita (dal basso verso l'altro, *bottom-up*) a partire da un insieme di sotto-problemi potenzialmente ripetuti.
- La tecnica della *memoization* (annotazione) è la versione *top-down* della programmazione dinamica.
- La tecnica *greedy* effettua sempre la scelta localmente ottima (necessita di una dimostrazione).
- Il backtrack procede per "tentativi", tornando ogni tanto sui nostri passi;
- Nella ricerca locale la soluzione ottima viene trovata "migliorando" via via soluzioni esistenti; Negli algoritmi probabilistici si dimostra che talvolta è meglio scegliere casualmente, ma in modo "gratuito", che con giudizio, ma in maniera costosa.

12.2 La tecnica del Dividi-et-Impera

La tecnica del Dividi-et-Impera si suddivide in tre fasi principali:

- **Divide**: divide il problema in sotto-problemi più piccoli è indipendenti;
- **Impera**: risove i sottoproblemi ricorsivamente;
- **Combina**: "unisce" le soluzioni dei sottoproblemi.

Sfortunatamente non esiste una ricetta unica per applicare questa tecnica: ad esempio l'algoritmo mergeSort ha una fase "divide" banale (bastava calcolare il valore mediano) mentre una fase di unione delle soluzioni complessa, diversamente nel quickSort la fase "divide" è complessa ma non esiste una fase "combina". É quindi necessario fare uno sforzo creativo, in quanto la tecnica ci dà una modalità con cui ad arrivare alla soluzione, ma bisogna applicarla caso per caso.

12.2.1 Minimo divide-et-impera

La tecnica "dividi et impera" non è un proiettile d'argento, a volte utilizzarla crea più danni di quanti ne risolva. Ad esempio osserva questo esempio nella quale è presentato un algoritmo di ricerca del minimo con questa tecnica.

Algoritmo 12.1: Algoritmo di ricerca del minimo con tecnica dividi-et-impera

```
minrec(int[ ] A, int i, int j)
    if i == j then
        return A[i]
    else
        m ← ⌊(i+j)/2⌋
        return min(minrec(A, i, m), minrec(A, m + 1, j))
```

Complessità L'algoritmo divide il vettore a metà, cerca il minimo nella metà di sinistra e nella metà di destra, il risultato è il minimo dei due minimi.

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

La complessità ammonta a $\alpha = 1, \beta = 0, T(n) = \Theta(n)$, non ne vale la pena, tanto vale fare la ricerca del minimo come abbiamo spiegato nelle lezioni.

12.3 La torre di Hanoi

La torre di Hanoi è un gioco matematico che consiste di tre pioli e di n dischi di dimensioni diverse. Inizialmente i dischi sono impilati in ordine decrescente nel piolo di sinistra. Lo scopo del gioco è quello di impilare i dischi sul piolo di destra, senza mai impilare un disco più grande su uno più piccolo, muovendo al massimo un disco alla volta ed utilizzando il piolo centrale come appoggio. Questo problema può essere risolto tramite la tecnica "dividi-et-impera".

Algoritmo 12.2: Versione ricorsiva della soluzione al problema della torre di Hanoi

```
hanoi(int n, int src, int dest, int middle)
    if n = 1 then
        stampa src → dest
    else
        hanoi(n − 1, src, middle, dest)  // sposta n − 1 dischi da src a middle
        stampa src → dest  // sposta 1 disco da src a dest
        hanoi(n − 1, middle, dest, src)  // sposta n − 1 dischi da middle a dest
```

Nella prima parte l'algoritmo sposta $n-1$ dischi da *src* a *middle* utilizzando *dest* come punto d'appoggio. Dopodiché sposta l'ultimo disco rimanente dalla *src* alla *dest*. Infine sposta $n-1$ dischi da *src* a *dest* utilizzando *src* come punto d'appoggio.

Complessità L'equazione di ricorrenza prodotta da questo algoritmo è $T = 2T(n-1) + 1 = \Theta(2^n)$. Si può dimostrare che questa soluzione è ottima (non si può fare meglio di così).

12.4 Algoritmo di ordinamento

L'algoritmo di ordinamento quickSort è basato sulla tecnica "dividi et impera", nel caso medio ha una complessità di $\mathcal{O}(n \log n)$, mentre nel caso pesssimo è di $\mathcal{O}(n^2)$. Fino a qualche anno fa era l'algoritmo di eccellenza per l'ordinamento. Infatti presenta molti aspetti a suo favore:

- il fattore costante del quickSort è migliore di quello del mergeSort;
- non utilizza memoria addizionale in quanto svolge i calcoli "in-memory" (a differenza di mergeSort che ha bisogno di un vettore di appoggio);

- esistono delle tecniche "euristiche" per evitare il caso pessimo.

Quindi spesso è preferito ad altri algoritmi. All'interno dell'ultimo capitolo (Capitolo 21) riassumeremo tutti gli algoritmi di ordinamento visti fin'ora e ne vedremo di nuovi, tra questi anche gli algoritmi attualmente utilizzati negli attuali linguaggi di programmazione (c, java, python).

Spiegazione Sono dati in input un vettore $A[1 \dots n]$, gli indici *start*, *end* tali che $1 \leqslant start \leqslant end \leqslant n$, tali indici indicano quale parte del vettore stiamo ordinando, come avviene in mergeSort.

1. la parte del "divide" avviene nel seguente modo:
   - scegliamo un valore $p \in A[start \dots end]$ detto perno (*pivot*);
   - spostiamo gli elementi del vettore $A[start \dots end]$ in modo tale che:
     - $\forall i \in [start \dots j-1] : A[i] \leqslant p$;
     - $\forall i \in [j+1 \dots end] : A[i] \geqslant p$

     l'indice j viene calcolato per rispettare tale condizione;
   - il perno viene messo in posizione $A[j]$.
2. la parte "impera" ordina i due sottovettori $A[start \dots j-1]$ e $A[j+1 \dots end]$ richiamando ricorsivamente quickSort;
3. la parte "combina" non fa nulla.

Algoritmo 12.3: Algoritmo di ordinamento

```
quickSort(ITEM[] A, int primo, int ultimo)
    // su almeno due elementi
    if primo < ultimo then
        int j ← perno(A, primo, ultimo)  // logica dell'algoritmo

        // richiamo l'algoritmo su entrambi i sottovettori
        quickSort(A, primo, j − 1)
        quickSort(A, j + 1, ultimo)

// sposta gli elementi più piccoli a sinistra del perno, i più grandi a destra
int perno(ITEM[] A, int primo, int ultimo)
    ITEM x ← A[primo]  // il perno è il primo elemento
    int j ← primo  // il cursore parte dal primo elemento

    // spostamenti "in-place"
    da i ← primo fino a ultimo fai
        if A[i] < x then // l'elemento è più piccolo del perno
            j++  // sposta il cursore j
            swap(A[i], A[j])  // scambia gli elementi:  i ↔ j

    /* a questo punto tutti gli elementi posizionati prima della posizione j sono più piccoli del
       perno, rimane solo da riposizionare il perno nella sua posizione finale (è ordinato)      */

    // riposiziono il perno
    A[primo] ← A[j]
    A[j] ← x

    // restituisco la posizione del perno
    return j
```

Complessità computazionale Il costo della funzione perno è $\Theta(n)$ (deve guardare $n-1$ valori ed effettuare i confronti). Il costo di quickSort dipende dal partizionamento:

- il partizionamento *peggiore* si verifica quando il perno è l'elemento minimo (o massimo), questo particolare caso accade quando il vettore è ordinato in ordine crescente (decrescente). La complessità risultante è $T(n) = T(n-1) + T(0) + \Theta(n) = T(n-1) + \Theta(n) = \Theta(n^2)$.
- il partizionamento *migliore* avviene quando il vettore di dimensione n viene diviso in due sottoproblemi di dimensione $n/2$. La complessità risultante è $T(n) = 2T(n/2) + \Theta(n) = \Theta(n \log n)$;
- il partizionamento nel *caso medio* è molto più vicino al caso ottimo che al caso peggiore, prendiamo ad esempio il partizionamento 9-a-1:

$$T(n) = T\left(\frac{n}{10}\right) + T\left(\frac{9n}{10}\right) + cn = \Theta(n \log n)$$

Prendiamo un altro esempio, il partizionamento 99-a-1:

$$T(n) = T\left(\frac{n}{10}\right) + T\left(\frac{99n}{10}\right) + cn = \Theta(n \log n)$$

Nota. *In questi esempi, il partizionamento ha proporzionalità limitata e i fattori moltiplicativi possono essere importanti.*

Il costo computazionale dipende dall'ordine degli elementi e non dai loro valori. Dobbiamo quindi considerare tutte le possibili permutazioni, il che è difficile dal punto di vista analitico. Alcuni partizionamenti saranno parzialmente bilanciati, altri pessimi; in media questi si alterneranno nella sequenza di partizionamenti, ma quelli parzialmente bilanciati "dominano" quelli pessimi.

12.5 Moltiplicazione di catena di matrici

Viene fatto un accetto ad un argomento che verrà affrontato in modo più approfondito nel capitolo successivo.

12.6 Conclusioni

La tecnica dividi-et-impera viene applicata quando i passi "divide" e "combina" sono semplici e i costi risultano migliori del corrispondente algoritmo iterativo (quindi, ad esempio, va bene per effettuare l'ordinamento, ma non per effettuare la ricerca del minimo). Ulteriori vantaggi dell'applicazione di questa tecnica sono:

- la facile parallelizzazione: la possibilità di dividere il problema in più sottoproblemi porta ad una naturale divisione dei compiti fra più processori;
- l'utilizzo ottimale della memoria *cache* (*cache oblivious*): tutti i dati con la quale stiamo lavorando sono colocalizzati nella memoria principale.

12.7 Applicazioe della tecnica

Infine vediamo una prima applicazione della tecnica e ne valutiamo le prestazioni.

12.7.1 Gap

In un vettore V contenente $n \geqslant 2$ interi, un gap è un indice i, $1 < i \leqslant n$, tale che $V[i-1] < V[i]$.

- Dimostrare che se $n \geqslant 2$ e $V[1] < V[n]$, allora V contiene almeno un gap;
- Progetta un algoritmo che, dato un vettore V contentente $n \geqslant 2$ interi e tale che $V[1] < V[n]$ (la condizione sopra), restituisca la posizione di un gap nel vettore (questo algoritmo assume che il gap esista).

Dimostrazione per assurdo. Suppponiamo che non ci sia un gap nel vettore. Allora $V[1] \geqslant V[2] \geqslant V[3] \geqslant \cdots \geqslant V[n]$, che contraddice il fatto che $V[1] < V[n]$. □

Proviamo a riformulare la proprietà tenendo conto di due indici:

- sia V un vettore di dimensione n;
- siano i, j due indici tali che $1 \leqslant i < j \leqslant n$ e $V[i] < V[j]$.

In altre parole, ci sono più di due elementi nel sottovettore $V[i \dots j]$ e il primo elemento $V[i]$ è più piccolo dell'ultimo elemento $V[j]$.

Dimostrazione per induzione. Voglia provare per induzione sulla dimensione n del sottovettore che il sottovettore contiene un gap.

- **caso base**: $n = j - i + 1 = 2$, ad esempio $j = i + 1$: $V[i] < V[j]$ implica che $V[i] < V[j]$ implica che $V[i] < V[i+1]$, che è un gap;
- **ipotesi induttiva**: dato un qualunque (sotto)vettore $V[h \dots k]$ di dimensione $n' < n$, tale che $V[h] < V[k]$, allora $V[h \dots k]$ contiene un gap;
- **passo induttivo**: consideriamo un qualunque elemento m tale che $i < m < j$. Almeno uno dei due casi seguenti è vero:
  - se $V[m] < V[j]$, allora esiste un gap in $V[m \dots j]$, per ipotesi induttiva;
  - se $V[i] < V[m]$, allora esiste un gap in $V[i \dots m]$, per ipotesi induttiva.

□

Algoritmo 12.4: Algoritmo che ricerca un intervallo all'interno del vettore

```
// funzione wrapper
gap(int[ ] V, int n)
    // n: dimensione del vettore
    return gapRec(V, 1, n)

gapRec(int[ ] V, int i, int j)
    if j == i + 1 then // ho due elementi
        return j  // ritorno il secondo elemento
    m = ⌊(i+j)/2⌋  // calcolo il mediano
    if V[m] < V[j] then
        return gapRec(V, m, j)  // a destra
    else
        return gapRec(V, i, m)  // a sinistra
```

Complessità L'algoritmo ha complessità

Tabella 1: Valutazione delle prestazioni degli algoritmi scritti in python

n	Iterativa (ms)	Ricorsiva (μs)
10^3	0.06	2.05
10^4	0.61	2.78
10^5	6.11	3.36
10^6	62.44	4.01
10^7	621.69	4.87
10^8	6205.72	5.47