RICORSIONE E TECNICA DIVIDE ET IMPERA

Un *problema* è caratterizzato dai seguenti elementi:

(i) ciò che è noto, cioè i *dati* che costituiscono l'*istanza* del problema che si deve risolvere (ovvero l'*input*);

(ii) ciò che si deve determinare, cioè i *risultati* la cui determinazione fornisce la *soluzione* del problema (ovvero l'*output*);

(iii) le relazioni (*proprietà*) che legano i dati di input ai risultati e costituiscono la *struttura* del problema.

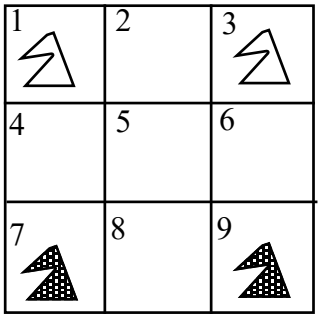
Da un punto di vista strettamente matematico, ad ogni problema può essere associata una funzione definita sullo spazio di tutte le istanze del problema, la quale per ogni istanza restituisce come valore l'insieme delle soluzioni; se tale insieme è vuoto, l'istanza del problema non ammette soluzione; se è un singoletto, l'istanza ha un'unica soluzione. Un problema in cui la soluzione è di tipo booleano e per ogni istanza del quale esiste esattamente un'unica soluzione (*true* o *false*) è chiamato *problema di decisione*.

Un *metodo risolutivo* di un problema è un algoritmo che calcola una delle soluzioni nell'insieme restituito da tale funzione se esso non è vuoto o altrimenti restituisce una opportuna segnalazione che il problema non ha soluzioni. Un problema è *polinomiale* se esiste un algoritmo che lo risolve in tempo polinomiale e *non-polinomiale* (ad esempio, *esponenziale*) altrimenti. Molti problemi non sono classificabili nel senso che non sono noti algoritmi risolutivi polinomiali per essi ma non è escluso che essi possano essere un giorno individuati.

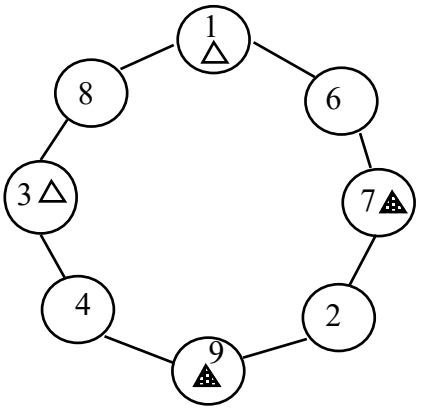
In generale l'individuazione di un algoritmo efficiente (ad esempio polinomiale di grado basso) per la risoluzione di un problema richiede non solo la conoscenza approfondita delle caratteristiche strutturali del problema, ma anche la conoscenze di varie tecniche di progetto di algoritmi. In questo apprenderemo una tecnica fondamentale di progetto di algoritmi, *Divide et Impera*, basata sulla ricorsione, che, a partire da un problema, individua alcuni

sottoproblemi da risolvere ricorsivamente per costruire la soluzione del problema originario. Altre tecniche di progetto di algoritmi saranno studiate in successivi capitoli.

Ovviamente la sola conoscenza delle varie tecniche di progetto di algoritmi non è sufficiente per risolvere un qualsiasi problema. E' fondamentale riflettere sul problema e analizzarlo approfonditamente alla ricerca di qualche analogia, di una caratteristica strutturale che ci permetta di formularlo in maniera tale da poter utilizzare uno dei metodi di risoluzione del nostro repertorio. Questa fase non è automatica e necessita di una certa dose di immaginazione e creatività. Si consideri ad esempio il problema di dover scambiare di posto i due cavalli bianchi in alto con quelli neri in basso nella seguente scacchiera, usando le mosse ad *elle* ammesse per tali pezzi:



Il problema può essere risolto, come si dice in maniera *bruta*, andando a considerare tutti i possibili spostamenti. Tuttavia una soluzione semplice ed efficiente può essere ottenuta introducendo la seguente fantasiosa rappresentazione degli spostamenti possibili per ogni posizione:



Ad esempio, il cavallo bianco in posizione 1 può spostarsi in posizione 6; e se spostiamo il cavallo nero in posizione 7 alla posizione 2, il cavallo bianco può ulteriormente spostarsi in posizione 2. A questo punto la soluzione del problema è banale: basta far circolare i pezzi in senso orario o antiorario in modo che i cavalli bianchi nelle posizioni 3 e 1 passino rispettivamente nelle posizioni 7 e 9 mentre mentre quelli neri nelle posizioni 7 e 9 passino rispettivamente nelle posizioni 3 e 1.

Immaginazione e creatività sono stati sufficienti per risolvere il problema dello scambio di pezzi e il loro ruolo nel progetto di algoritmi è universalmente riconosciuto — addirittura si

parla di *arte della programmazione*. Ma, come accade anche nell'arte, immaginazione e creatività sono importanti ma non sufficienti. Esse vanno combinate con una approfondita conoscenza di tecniche, per evitare di dover inventare ad ogni passo la ruota inaridendo così le proprie capacità progettuali. Ciò tanto più vale nel progetto di algoritmi che, al di là delle connotazioni artistiche, rimane comunque una disciplina ingegneristica, in cui esistono ormai tecniche generali che, se utilizzate opportunamente, possono guidare verso soluzioni valide sia per le prestazioni degli algoritmi che per la facilità di verifica della loro correttezza.

Passiamo ora a descrivere l'organizzazione del capitolo. Nella Sezione 1, presenteremo lo schema generale di un algoritmo di tipo Divide et Impera e valuteremo la complessità per vari casi, a secondo del numero di sottoproblemi generati, la dimensione di essi e il costo di individuarli e ricomporre le loro soluzioni. Presenteremo algoritmi Divide et Impera per la moltiplicazione efficiente di interi e di matrici nella Sezione 2. La tecnica è particolarmente significativa per il caso del prodotto di due matrici in quanto tale problema è la base di molti problemi su matrici, ad esempio la risoluzione di un sistema di equazioni lineari. Mostreremo che, come spesso accade con algoritmi Divide et Impera, le tecniche di moltiplicazione introdotte sono realmente più efficienti di quelle tradizionali solo quando gli interi hanno un numero elevato di cifre e le matrici sono di grandi dimensioni. Nella Sezione 3 presenteremo due ulteriori algoritmi di ordinamento efficienti, l'ordinamento per fusione (*merge sort*) e l'ordinamento rapido (*quick sort*) in aggiunta a quelli (heap sort e radix sort) introdotti nel Capitolo 2. Affronteremo il problema dell'ordinamento parziale nella Sezione 4. In particolare affronteremo prima il problema della determinazione del secondo maggiore (o secondo minore) con il minimo numero possibile di confronti per poi concentrarci sul problema più generale di determinare il k-mo maggiore (o minore). Infine, nella Sezione 5, mostreremo come la tecnica Divide et Impera non sia altro che un caso particolare di ricorsione. Così come la ricorsione può essere sostituita dalla iterazione, discuteremo come realizzare un algoritmo Divide et Impera in maniera iterativa e quando tale tipo di realizzazione risulti più conveniente.

3.1 LA TECNICA DIVIDE ET IMPERA

3.1.1. Algoritmo Tipico

La tecnica *Divide et Impera*, come il nome stesso suggerisce, consiste nel risolvere un problema mediante un'accorta suddivisione di esso in vari sottoproblemi. Più precisamente, si individuano *k* problemi aventi dimensioni più piccole, si risolvono ricorsivamente i *k* sottoproblemi individuati e si utilizzano le loro soluzioni per determinare quella del problema originale; la ricorsione si interrompe allorquando un sottoproblema raggiunge una dimensione così piccola da poter essere risolto direttamente. L'algoritmo tipico ha il seguente schema:

```
Tsoluzione DivideEtImpera ( const Tproblema& p )
{     if ( p.dimensione <= dimMin )
            return risolviDirettamente(p);
      else
      {     card k = individuaNumeroSottoProblemi(p);
            Vettore<Tproblema> P(1,k);
            P = individuaSottoProblemi(p,k);
            Vettore<Tsoluzione> S(1,k);
            for ( Card i = 1; i <= k; i++ )
                  S[i] = DivideEtImpera(P[i]);
            return combinaSoluzioni(S);
      }
}
```

Come esempio, consideriamo il problema di dover ricercare in un vettore ordinato un dato elemento dalla posizione 1 alla posizione *n*. Seguendo la tecnica *Divide et Impera*, individuiamo innanzitutto il numero di sottoproblemi, che in questo caso è 1. Quindi individuiamo l'unico sottoproblema nel seguente modo: se l'elemento centrale del vettore precede l'elemento dato nell'ordinamento allora il sottoproblema è quello di ricercare l'elemento dato dalla posizione *n*/2+1 fino a *n*, altrimenti se l'elemento dato precede l'elemento centrale il sottoproblema è quello di ricercare l'elemento dato dalla posizione 1 alla posizione *n*/2-1. A questo punto si risolve ricorsivamente il sottoproblema; la fase successiva di combinare la soluzione dei sottoproblemi è banale poichè la soluzione del problema originale coincide con quella del sottoproblema. Ovviamente nel caso che il vettore ha una dimensione ridotta, in particolare 1, il problema può essere risolto immediatamente.

Non c'è certamente sfuggito che l'algoritmo appena descritto non è altro che la ben nota *ricerca binaria* (o *ricerca logaritmica*) della quale abbiamo già scritto una versione iterativa nel Capitolo 2. La ricerca binaria può essere riscritta in versione Divide et Impera nel seguente modo:

```
bool ricercaBinaria(int V [], int X, int in, int fin)
{    if ( in >= fin ) // test per risoluzione diretta
          return ((in == fin) && (X==V[in]));
     else
     {    // individuazione dell'unico sottoproblema
          int medio = (in+fin)/2;
          if ( V[medio]<X )
               in = medio+1;
          else
               if (X < V[medio] )
                    fin = medio - 1;
               else
                    in = fin = medio;
          // risoluzione dell'unico sottoproblema e
          // utilizzo della sua soluzione
          return ricercaBinaria(V,X,in,fin);
     }
}
```

Basandosi su una più sofisticata scelta dell'unico sottoproblema, ma solo nel caso siano definiti operatori aritmetici per *T*, possiamo modificare la ricerca binaria nel seguente modo, ottenendo così la cosiddetta *ricerca uniforme*:

```
bool ricercaUniforme(int V [], int X, int in, int fin)
{    if ( in >= fin || X<V[in] || V[fin]<X || V[in]==V[fin] )
          return ((in <= fin) && (X==V[in]));
     else
     {    // individuazione più accurata dell'unico sottoproblema
          int m = in+(fin-in)*(X-V[in])/(V[fin]-V[in]);
          if (V[m]<X)
               in = m+1;
          else
               if (X<V[m])
                    fin = m - 1;
               else
                    in = fin = m;
          // risoluzione dell'unico sottoproblema e
          // utilizzo della sua soluzione
          return ricercaUniforme(V,X,in,fin);
     }
}
```

Il test più sofisticato per verificare se procedere alla risoluzione diretta del problema poteva essere utilizzato anche nella ricerca binaria; nel caso della ricerca uniforme, esso diventa obbligatorio per evitare di avere valori negativi o addirittura divisioni per zero nel calcolo di x. Al contrario della ricerca binaria, la ricerca uniforme è un algoritmo naturale che spesso utilizziamo nel consultare un dizionario o un elenco telefonico. Infatti, se dobbiamo ricercare una parola iniziante con "S", andremo verso la fine dell'elenco mentre, se la parola inizia con "B", andremo verso l'inizio e andremo al centro solo nel caso la parola inizi con una delle lettere centrali dell'alfabeto.

Nelle sezioni che seguono avremo modo di analizzare altre applicazioni tipiche della tecnica Divide et Impera.

3.3.ORDINAMENTI EFFICIENTI

3.3.1.Ordinamento per Fusione (Merge Sort)

Il metodo di *ordinamento per fusione* (*merge sort*) è una semplice e classica applicazione della tecnica Divide et Impera. Si procede nel seguente modo. Dato un vettore di n elementi, si suddivide il vettore in due parti, la prima costituita dai primi $n/2$ elementi e la seconda contenente i rimanenti. A questo punto si ordinano ricorsivamente i due sottovettori e quindi il vettore complessivo viene ordinato per fusione dei due sottovettori ordinati nel seguente modo:

```
void merge( int V[], int in, int fin, int medio)
{    int A[fin-in];
     int i1 = in; int i2 = medio+1;
     int i3=0;
     while((i1 <= medio)&&(i2 <= fin))
     {    if (V[i1]<V[i2] )
               {A[i3] = V[i1]; ++i1;}
          else
               {A[i3] = V[i2]; ++i2;}
          ++i3;
     }
     while (i1 <= medio)
     {    A[i3] = V[i1];
          ++i1;
          ++i3;
     }

     while (i2<=fin)
     {    A[i3] = V[i2];
          ++i2;
          ++i3;
     }

     for ( i3 = 0, i1 = in; i1 <= fin; ++i3, ++i1)
          V[i1] = A[i3];
}
```

La complessità della fusione è ovviamente lineare mentre l'individuazione dei sottoproblemi è addirittura costante. Quindi l'algoritmo Divide et Impera è di tipo 1, caso $a = c = 2$ e $d = 1$ e, dunque la complessità dell'algoritmo è $\Theta(n \log n)$. Come è usuale con gli algoritmi backtracking e, più in generali, con tutti gli algoritmi che usano la ricorsione, i vantaggi reali si hanno quando n supera un certo valore. In questo caso, come suggerito dagli esperimenti, fissiamo tale limite in 20. Sotto tale limite adoperiamo un algoritmo di ordinamento diverso, addirittura quadratico, ma che, a causa del peso delle costanti, ha un comportamento migliore; in particolare, scegliamo l'*ordinamento per inserzione* (*insertion sort*) che ha un ottimo comportamento per valori piccoli di n. Scriviamo di seguito l'algoritmo di ordinamento per fusione:

```
void mergeSort( int V[], int in, int fin)
{    if ( (fin - in) < 20 )
          insertionSort(V,in,fin);
     else
     {    int medio = (in+fin)/2;
          mergeSort(V,in,medio);
          mergeSort(V,medio+1,fin);
          merge(V,in,fin,medio);
     }
}
```

Poichè la ricorsione non può essere arrestata anticipatamente e la funzione "merge" richiede sempre un tempo di esecuzione lineare, la complessità $\Theta(n \log n)$ vale non solo nel caso peggiore ma anche in quello medio e in quello migliore.

ESERCIZIO: adattare la insertionSort per lavorare su un intervallo dell'array.

SOLUZIONE:

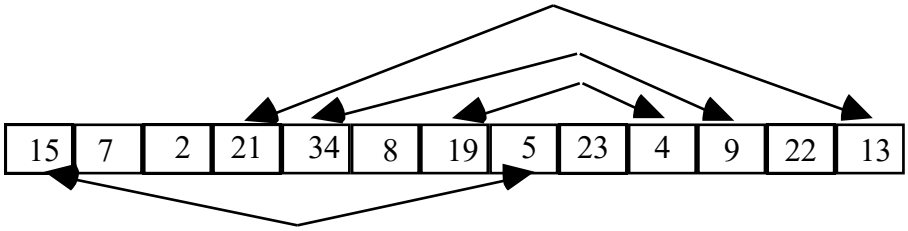
```
void insertionSort(int V[], int in, int fin)
{   int temp, j, i;
     for(i=in;i<=fin;i++)
     {
          temp=V[i];
          j=i-1;
          while((V[j]>temp) && (j>=in)) {
               V[j+1]=V[j];
               j--;
           }
          V[j+1]=temp;
     }
}
```

3.3.2.Ordinamento Rapido (Quick Sort)

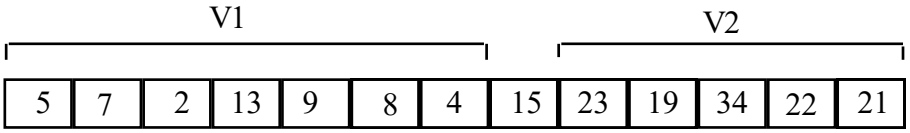
Con il metodo dell'*ordinamento rapido* (*quick sort*), il problema dell'ordinamento viene affrontato come segue: si prende in esame un elemento *x* del vettore *V* da ordinare, ad esempio il primo, e si determina la posizione esatta che gli compete nell'ordinamento. Contemporaneamente si portano alla sua sinistra tutti gli elementi che lo precedono nell'ordinamento e alla sua destra quelli che lo seguono ottenendo così due sottovettori V_1 e V_2. A questo punto i due sottovettori possono essere ordinati con lo stesso metodo secondo la tecnica Divide et Impera. Una volta ordinati i due sottovettori, anche il vettore è ordinato poichè l'elemento *x* di separazione tra i due sottovettori è già nella sua posizione nell'ordinamento e tutti gli elementi in V_1 precedono quelli in V_2. L'algoritmo cruciale è quindi la *partizione* di *V*. A tale scopo si scandisce il vettore V da sinistra a destra finchè non si trova un elemento y che segue *x* nell'ordinamento e da destra a sinistra finchè non si trova un elemento *z* che precede x; a questo punto *y* e *z* vengono scambiati. Le scansioni vengono riprese per trovare eventuali altri due elementi da scambiare e così via fino a quando tutti gli

elementi non siano stati scanditi. Alla fine viene scambiato *x* con il più a destra degli elementi che lo precedono.

Esempio 2. Dato il vettore:



dopo la scansione e gli scambi indicati con le doppie frecce, l'elemento 15 verrà portato nella posizione che gli spetta nell'ordinamento, e tutti gli elementi alla sua sinistra sono più piccoli mentre quelli alla sua destra sono più grandi. Il vettore diventa:



Scriviamo di seguito l'algoritmo di partizionamento:

```
int partiziona(int V[], int in, int fin)
{    int i = in; int j = fin+1;
     while ( i < j )
     {    do j--; while (V[in]<V[j] );
          do i++; while (V[in]>=V[i] && i <j );
          if ( i < j ) scambia(V[i],V[j]);
     }
     scambia(V[in],V[j]);
     return j;
}
```

L'algoritmo di ordinamento rapido può essere quindi scritto come segue utilizzano la funzione *partiziona* e richiamando, come è ormai usuale, per $n < 20$ un ordinamento iterativo quale l'insertion sort :

```
void quickSort( int V[], int in, int fin)
{    if ( (fin - in) < 20 )
          insertionSort(V,in,fin);
     else
     {    int posOrd=   partiziona(V,in,fin);
          quickSort(V,in,posOrd-1);
          quickSort(V,posOrd+1,fin);
     }
}
```

La complessità del quick sort dipende da quella della funzione *partiziona*. Il numero di confronti nella funzione *partiziona* è $\Theta(n)$ nei tre casi peggiore, medio e migliore mentre il numero di scambi è $\Theta(n)$ nel caso peggiore e medio e $\Theta(1)$ nel caso migliore quindi complessivamente la complessità di *partiziona* è $\Theta(n)$ nei tre casi. Un altro punto importante

per la valutazione della complessittà dell'algoritmo di ordinamento è stabilire qual'è la dimensione massima dei due sottovettori: si passa da un estremo in cui un sottovettore è nullo mentre l'altro ha dimensione n-1 all'estremo opposto in cui ambedue i sottovettori hanno dimensione $n/2$ (in effetti, $(n-1)/2$ ma sappiamo ormai trascurare le costanti irrilevanti). Nel primo caso il quick sort è un l'algoritmo Divide et Impera di tipo 2, caso $a = 1$ e $d = 1$, cioè va risolto un unico sottoproblema di dimensione n-1 e il costo per individuare il sottoproblema e utilizzare la sua soluzione per determinare quella del problema originario è lineare; la complessità è quindi $\Theta(n^2)$. Nel secondo caso il quicksort diventa un algoritmo Divide et Impera di tipo 1, caso $a = c = 2$ e $d = 1$ come per il merge sort, per cui la complessità è $\Theta(n \log n)$. Pertanto possiamo senz'altro stabilire che il quick sort ha complessità $\Theta(n^2)$ nel caso peggiore e $\Theta(n \log n)$ nel caso migliore. Facendo la ragionevole ipotesi di equiprobabilità di occorrenza di tutte le possibilità tra i due estremi possibili circa la dimensione dei sottovettori, è possibile dimostrare che la complessità del quicksort è $\Theta(n \log n)$ nel caso medio. Per quanto riguarda il numero di confronti e di scambi abbiamo i seguenti risultati:

	Confronti			**Scambi**		
	MIN	*MEDIO*	*MAX*	*MIN*	*MEDIO*	*MAX*
Quick Sort	$\Theta(n\log n)$	$\Theta(n\log n)$	$\Theta(n^2)$	$\Theta(1)$	$\Theta(n\log n)$	$\Theta(n\log n)$

L'algoritmo di quick sort ha un comportamento particolarmente favorevole nel caso medio, addirittura superiore a quello dello heap sort e del merge sort in termini di costanti, e viene spesso utilizzato nella pratica nonostante la sua complessità quadratica nel caso peggiore. Al contrario del bubble sort, il comportamento peggiore accade quanto più il vettore originario è parzialmente ordinato mentre è tanto più efficiente quanto più il vettore è disordinato. Per migliorare il suo comportamento si può scegliere l'elemento di riferimento per il partizionamento mediante una scelta casuale, che aumenti il disordine, piuttosto che scegliere sempre uno stesso elemento, il primo o un qualsiasi altro — facendo un paragone con le carte da gioco, è come prelevare una qualsiasi carta dal mazzo piuttosto che accettare la prima. Introduciamo questo stratagemma nella funzione *partizione*, ricorrendo alla funzione *rand* della libreria matematica di C++, che ad ogni chiamata restituisce un intero casuale, che viene da noi trasformato in un cardinale casuale nell'intervallo degli indici del vettore:

```
int  partiziona( int V[], int in, int fin)
{    int r = abs(rand());
     int iCasuale = r % (fin-in+1) + in;
     scambia(V[in],V[iCasuale]);
     int i = in;
     int j = fin+1;
     while ( i < j )
     {    do j--; while ( V[in]<V[j] );
          do i++; while ( V[i]>=V[in] && i <j );
          if ( i < j ) scambia(V[i],V[j]);
     }
     scambia(V[in],V[j]);
     return j;
}
```