1 Il problema dell'ordinamento

Input: Una sequenza di n numeri, $(a_1, a_2, \ldots, a_n)$.
Output: Una permutazione $(a'_1, a'_2, \ldots, a'_n)$ della sequenza di input tale che $a'_1 \leq a'_2 \leq \ldots \leq a'_n$.
Ipotesi:

- Ci riferiamo ad un ordinamento crescente di tipo numerico. L'ordinamento inverso si ottiene in modo del tutto ovvio invertendo il senso dei confronti.
- In generale, la distinzione fra numeri e caratteri alfabetici è irrilevante.
- I metodi descritti nel seguito sono applicabili all'ordinamento per chiave di file composti da record (di cui la chiave rappresenta un campo).

1.1 Selection Sort

È forse il più semplice degli algoritmi di ordinamento: si cerca l'elemento più piccolo del vettore e lo si scambia con quello attualmente in prima posizione. Si ripete il procedimento $n-1$ volte relativamente ai sottovettori non ancora ordinati (es.: al passo i si cerca il minimo fra gli elementi che occupano le posizioni dalla i–esima alla n–esima e lo si pone in posizione i–esima).

Algoritmo

```
procedure selection (var a: arrayorder);
    var i, j, min, t: integer;
    begin
        for i := 1 to n - 1 do
            begin
                min := i;
                for j := i + 1 to n do
                    if a[j] < a[min] then min := j;
                t := a[min]; a[min] := a[i]; a[i] := t
            end
    end;
```

Osservazioni:

- L'operazione fondamentale compiuta dall'algoritmo è la seguente: per ogni i, da 1 a $n-1$, viene scambiato il minimo elemento del sottovettore $a[i \ldots n]$ con $\boldsymbol{a[i]}$.
- Al crescere dell'indice i, tutti gli elementi la cui posizione nel vettore è inferiore ad i sono già nella loro posizione finale (cioè in ordine).
- Per ogni i, da 1 a $n-1$, vengono effettuati $\underline{1}$ scambio ed $\underline{n-i}$ confronti per un totale di $n-1$ scambi e $(n-1)+(n-2)+\ldots+2+1 = \dfrac{n(n-1)}{2}$ confronti, indipendentemente dal vettore in input (solo il numero di volte che il valore della variabile *min* viene modificato dipende dall'input).

1.2 Insertion Sort lineare

È il metodo comunemente utilizzato per ordinare una "*mano*" di bridge o di ramino: si inizia con la mano sinistra vuota e quindi si inserisce una carta alla volta nella posizione corretta. Tornando a parlare di numeri, ciascun elemento del vettore da ordinare viene inserito nella posizione che gli compete, spostando

tutti gli elementi maggiori di una posizione verso destra (ovviamente a partire dall'ultimo elemento). Si noti, in particolare, che gli elementi del vettore vengono ordinati in loco, cioè al più un numero costante di elementi è temporaneamente memorizzato *fuori* dall'array.

Esempio: $A = (5, 2, 4, 6, 1, 3)$. Sia j l'indice con il quale si indica la "*carta corrente*", $a[1 \ldots j-1]$ le carte già ordinate e $a[j+1 \ldots n]$ quelle da inserire. Si ottiene la seguente successione di passi:

$$\begin{array}{cccccc} 5 & \underline{2} & 4 & 6 & 1 & 3 \\ 2 & 5 & \underline{4} & 6 & 1 & 3 \\ 2 & 4 & 5 & \underline{6} & 1 & 3 \\ 2 & 4 & 5 & 6 & \underline{1} & 3 \\ 1 & 2 & 4 & 5 & 6 & \underline{3} \\ 1 & 2 & 3 & 4 & 5 & 6 \end{array}$$

Algoritmo

```
procedure insertion (var a:arrayorder);
    var i, j, key: integer;
    begin
       for j := 2 to n do
          begin
             key := a[j];
             i := j;
             (* Si inserisce a[j] nella sequenza ordinata a[1...j-1] *)
             while a[i-1] > key do
                begin
                   a[i] := a[i-1];
                   i := i-1
                end;
                a[i] := key
          end
    end;
```

Osservando attentamente questo algoritmo ci si accorge tuttavia che si potrebbe avere un malfunzionamento, perché il ciclo *while* potrebbe proseguire anche per valori negativi dell'indice i. Per risolvere questo problema occorre mettere una "*sentinella*" nell'elemento $a[0]$ dell'array, che sia minore di tutti gli elementi che devono essere ordinati [1].

Osservazioni

- L'*insertion sort lineare* è un algoritmo di ordinamento **stabile**: elementi uguali all'interno del vettore mantengono le loro posizioni relative dopo il sorting.
- Un possibile metodo per migliorare l'algoritmo di ordinamento è quello di utilizzare una *ricerca binaria* (anziché sequenziale). In questo caso, l'algoritmo noto come *insertion sort binario* opera un numero "*ottimo*" di confronti, anche se il numero di scambi necessari per sistemare l'elemento corrente resta invariato. Pertanto non migliorano le prestazioni generali dell'algoritmo, la cui complessità resta dell'ordine di $\mathcal{O}(n^2)$.

[1] Utilizzando il linguaggio C, si noti che il primo elemento dell'array ha indice 0, caratteristica che, in questo caso, si rivela particolarmente utile. Infatti, gli elementi *effettivi* dell'array da ordinare vanno ad occupare le posizioni indiciate da 1 a n, mentre la sentinella occupa $a[0]$. Ovviamente il vettore ha lunghezza $n+1$.

1.2.1 Analisi computazionale dell'insertion sort

Il tempo impiegato per l'ordinamento dipende dalla dimensione e dal grado di ordinamento dell'input. L'unità di misura più naturale è quindi costituita dalla dimensione dell'input. Il tempo di esecuzione è rappresentato dal numero di operazioni elementari o "*passi*" eseguiti. In tal senso, occorre definire la nozione di passo: possiamo supporre che per eseguire ciascuna linea di pseudo–codice sia richiesto un tempo costante e che ciascuna linea richieda un tempo di esecuzione diverso da quello di tutte le altre. Pertanto si assumerà che ogni esecuzione dell'i–esima riga impiegi un tempo c_i.
Si ottiene la seguente tabella:

	Insertion sort (A)	costo	n° di volte
1	for $j \leftarrow 2$ to $length[A]$	c_1	n
2	do $key \leftarrow A[j]$	c_2	$n-1$
3	*Si inserisce A[j]*	0	$n-1$
4	$i \leftarrow j-1$	c_4	$n-1$
5	while $i > 0$ e $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
6	do $A[i+1] \leftarrow A[i]$	c_6	$\sum_{j=2}^{n}(t_j-1)$
7	$i \leftarrow i-1$	c_7	$\sum_{j=2}^{n}(t_j-1)$
8	$A[i+1] \leftarrow key$	c_8	$n-1$

con t_j numero di volte che il test del ciclo *while* alla linea 5 viene eseguito per quel j.
Pertanto il costo complessivo nel caso generale risulterà essere

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5\sum_{j=2}^{n} t_j + c_6\sum_{j=2}^{n}(t_j-1) + c_7\sum_{j=2}^{n}(t_j-1) + c_8(n-1).$$

Esaminiamo dunque i vari casi. Per l'*insertion sort* il caso migliore si verifica quando il vettore in ingresso è già ordinato, per cui, per $i = 2, 3, \ldots, n$, $A[i] < key$ (linea 5). Sotto questa ipotesi risulta ovviamente $t_j = 1$, da cui:

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) \\ &= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8), \end{aligned}$$

ovvero,

$$T(n) = an + b, \qquad \text{funzione lineare di } n.$$

Se invece l'array è ordinato in senso opposto a quello richiesto, siamo nel caso peggiore: occorre infatti confrontare l'elemento $A[j] = key$ con ogni elemento del sottovettore $A[1 \ldots j-1]$. Pertanto $t_j = j$.
Essendo $\sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1$ e $\sum_{j=2}^{n}(j-1) = \frac{n(n-1)}{2}$, si ha:

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) + c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1) \\ &= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right)n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right)n - (c_2 + c_4 + c_5 + c_8), \end{aligned}$$

ovvero,

$$T(n) = an^2 + bn + c, \qquad \text{funzione quadratica di } n.$$

Infine, nel caso medio, sarà $t_j = \frac{j}{2}$ e, facendo i dovuti calcoli, si vede che la funzione ottenuta è ancora una funzione quadratica della dimensione dell'array da ordinare.

1.3 Bubblesort

Il *bubblesort* ordina un array operando lo scambio di elementi adiacenti, quando questi non si trovino già nell'ordine corretto. L'algoritmo procede per passi successivi fino a che il vettore non risulta completamente ordinato (cioè si fa un passaggio a vuoto). Inoltre, ciascun passo dell'algoritmo pone almeno un elemento nella sua posizione finale. In questo modo, ad ogni passo, il vettore da ordinare ha almeno un elemento in meno rispetto al vettore ordinato al passo precedente.

Algoritmo

```
procedure bubble (var a:arrayorder);
    var i, j, t: integer;
    begin
       for i := n downto 1 do
          for j := 2 to i do
             if a[j − 1] > a[j] then
                begin
                   t := a[j − 1];
                   a[j − 1] := a[j];
                   a[j] := t
                end
    end;
```

Quando si incontra l'elemento più grande, durante la prima passata, questo viene scambiato con ognuno degli elementi che si trovano inizialmente alla sua destra (supponendo una rappresentazione del tipo vettore–riga) fino a che raggiunge la posizione più a destra nel vettore. Si procede quindi in modo analogo relativamente al sottovettore $a[1 \ldots n-1]$ e così via fino a che si ottiene un sottovettore composto da un solo elemento.
Nel caso pessimo – *file ordinato al contrario* – risulta chiaro che la i–esima passata del *bubblesort* richiede $n-i$ confronti e scambi, e quindi complessivamente si hanno $(n-1)+(n-2)+\ldots+2+1 = \frac{n(n-1)}{2}$ confronti–scambi. D'altra parte, le prestazioni del *bubblesort* dipendono dal vettore in input. Per esempio, occorono $n-1$ confronti (una sola passata) e nessuno scambio se il file è già ordinato. Tuttavia è stato provato che la *performance* nel caso medio non è significativamente migliore rispetto al caso pessimo.
Vediamo dunque quali possibili cambiamenti possono essere operati relativamente all'algoritmo *bubblesort* onde poterne migliorare le prestazioni:

- Tenere memoria del fatto che al passo precedente si siano o meno effettuati degli scambi (per poter terminare l'algoritmo).
- Ricordare la posizione dell'ultimo scambio avvenuto (per sapere esattamente quanto è lungo il sottovettore che deve ancora essere ordinato).
- Eliminare l'asimmetria del procedimento.

Esempio

12	18	42	44	55	67	<u>94</u>	6,	1 scambio, 7 passate,
<u>94</u>	6	12	18	42	44	55	67,	7 scambi, 1 passata.

⇒ Una possibile soluzione al problema è quella di alternare il verso di disamina del vettore in passate successive, utilizzando lo **Shakersort**. Infatti per files "*quasi ordinati*" lo *shakersort* può risultare molto efficiente: se k sono gli elementi fuori posto, la complessità in tempo è $\mathcal{O}(kn)$.

Algoritmo

```
procedure shakersort (var a:arrayorder);
    var j, k, l, r, x: integer;
    begin
      l := 2; r := n; k := n;
        repeat
          for j := r downto l do
            if a[j − 1] > a[j] then
              begin x := a[j − 1]; a[j − 1] := a[j];
                a[j] := x; k := j
              end;
          l := k + 1;
          for j := l to r do
            if a[j − 1] > a[j] then
              begin x := a[j − 1]; a[j − 1] := a[j];
                a[j] := x; k := j
              end;
          r := k − 1;
        until l > r
    end;
```

Esempio

$l=2$	$l=3$	$l=3$	$l=4$	$l=4$
$r=8$	$r=8$	$r=7$	$r=7$	$r=4$
44	$\underline{6}$	6	6	6
55	44	44	$\underline{12}$	12
12	55	12	44	$\underline{18}$
42	12	42	18	$\underline{42}$
94	42	55	42	$\underline{44}$
18	94	18	$\underline{55}$	55
6	18	$\underline{67}$	67	67
67	67	$\underline{94}$	94	94

1.4 Quicksort

La nuova tecnica che descriveremo si fonda sul principio che l'efficienza aumenta *effettuando gli scambi su lunghe distanze.* L'idea è la seguente: si scelga un elemento a caso, lo si indichi con x, si percorra il vettore a partire da sinistra fino a trovare un $a_i > x$ e poi da destra fino a trovare un $a_j < x$. Si scambino i due elementi e si continui il processo di scansione e scambio fino a quando le due scansioni si incontrano in un punto interno all'array. Così facendo si partiziona il vettore in due parti: quella a sinistra, che conterrà tutti gli elementi più piccoli di x, e quella a destra, che conterrà invece gli elementi maggiori di x. In questo modo x viene ad occupare la sua posizione definitiva. Il procedimento appena descritto impiega un tempo lineare nella dimensione n dell'array.

Esempio

$\underline{x=11}$	13	15	17	14	2	9	20	18	5	16	3	19	8	6	1	12	7	10	4
4	10	7	1	6	2	9	8	3	5	$\underline{x=11}$	16	19	18	20	14	12	17	15	13

Una volta calcolati i due sottovettori, la procedura deve essere riapplicata ricorsivamente a ciascuno di essi fino ad ottenere un vettore ordinato.

Algoritmo

```
procedure partizioni (var a:arrayorder);
    var i, j, w, x: integer;
    begin i := 1; j := n;
        (* Scegli x a caso. Per esempio x := a[1] *)
        repeat
            while a[i] < x do i := i + 1;
            while x < a[j] do j := j - 1;
            if i <= j then
                begin w := a[i]; a[i] := a[j]; a[j] := w;
                      i := i + 1; j := j - 1
                end
        until i > j
    end;
```

Il procedimento di base esegue tanti confronti quanti sono gli elementi dell'insieme: quando lo si applica a tutti i sottoinsiemi generati al passo i si effettuano ancora (al max) n confronti e spostamenti $\Rightarrow$ Se si procede per k passi la complessità sarà dell'ordine di $\mathcal{O}(kn)$. In particolare, nel caso in cui ad ogni passo il vettore risulti diviso in due parti di uguale lunghezza si avrà $k = \log_2 n$, da cui si ottiene un $\mathcal{O}(n \log_2 n)$ per la complessità in tempo. La procedura completa può essere formulata come segue:

Algoritmo

```
procedure quicksort (var a:arrayorder);
    procedure ordina (var s,d:integer);
        var i, j, w, x: integer;
        begin i := s; j := d;
              x := a[(s + d) div 2];
              repeat
                  while a[i] < x do i := i + 1;
                  while x < a[j] do j := j - 1;
                  if i <= j then
                        begin w := a[i]; a[i] := a[j]; a[j] := w;
                              i := i + 1; j := j - 1
                        end
              until i > j;
              if s < j then ordina(s, j);
              if i < d then ordina(i, d)
        end;
begin
    ordina(1, n)
end;
```

Il *quicksort* (come il *mergesort* che esamineremo in seguito) è basato sul paradigma del *Divide et Impera*. Vediamo in dettaglio cosa questo significhi relativamente al problema particolare dell'ordinamento di un vettore.

- **Divide**: Il vettore $a[p \ldots r]$ è ripartito in due sottovettori non vuoti $a[p \ldots q]$ e $a[q+1 \ldots r]$, in modo tale che ogni elemento di $a[p \ldots q]$ sia minore o uguale di ogni elemento di $a[q+1 \ldots r]$. In pratica, così facendo, si calcola la posizione finale dell'elemento q.
- **Impera**: I due sottovettori sono ordinati tramite chiamate ricorsive della stessa procedura.

- **Ricombinazione**: Poiché i sottovettori sono già ordinati tra loro, non è richiesto alcuno sforzo per ricombinarli a formare il vettore ordinato completo.

Algoritmo

```
procedure quicksort1 (p,r:integer);
    var q: integer;
    begin
       if r > p then
          begin
             q :=partition(p,r);  (* Restituisce la posizione dell'elemento separatore *)
             quicksort1(p, q − 1);
             quicksort1(q + 1, r)
          end
    end;
```

Se $p = 1$ e $r = n$ possiamo scrivere la seguente relazione di ricorrenza per il numero $C(n)$ di confronti.

$$\begin{cases} C(n) = 0 & n = 0, 1 \\ C(n) = P(n) + C(q-1) + C(n-q) & n > 1 \end{cases},$$

dove con $P(n)$ si indica il numero di confronti eseguiti dalla procedura *partition*. In particolare, risulta $P(n) = n + 1$, perché si effettuano $n - 1$ confronti fra x e tutti gli altri elementi dell'array, più due confronti quando i due puntatori si incrociano. Esaminiamo quindi i vari casi che possono verificarsi.

- **Caso pessimo**: $q = 1$

$$\begin{aligned} C(n) &= n + 1 + C(0) + C(n-1) = n + 1 + C(n-1) = (n+1) + n + C(n-2) = \ldots \\ &= (n+1) + n + (n-1) + (n-2) + \ldots + 3 = \sum_{i=3}^{n+1} i = \frac{(n+1)(n+2)}{2} - 3 = \mathcal{O}(n^2) \end{aligned}$$

- **Caso medio**
  Sia $\overline{C}(n)$ il valore medio di $C(n)$. Poiché ogni valore di q è equiprobabilmente distribuito in $(1, n)$ si ha:

$$\overline{C}(n) = n + 1 + \frac{1}{n} \sum_{k=1}^{n} [\overline{C}(k-1) + \overline{C}(n-k)]$$

  Ma $\overline{C}(0) + \overline{C}(1) + \ldots + \overline{C}(n-1) = \overline{C}(n-1) + \ldots + \overline{C}(1) + \overline{C}(0)$, da cui

$$\overline{C}(n) = n + 1 + \frac{2}{n} \sum_{k=1}^{n} \overline{C}(k-1).$$

  Pertanto moltiplicando per n ed $n - 1$ le espressioni relative a $\overline{C}(n)$ e $\overline{C}(n-1)$ rispettivamente, si ottiene:

$$n\overline{C}(n) = (n+1)n + 2\sum_{k=1}^{n} \overline{C}(k-1),$$

$$(n-1)\overline{C}(n-1) = n(n-1) + 2\sum_{k=1}^{n-1} \overline{C}(k-1),$$

  e, sottraendo,

$$n\overline{C}(n) - (n-1)\overline{C}(n-1) = n(n+1) - n(n-1) + 2\overline{C}(n-1),$$

che implica

$$n\overline{C}(n) = (n+1)\overline{C}(n-1) + 2n.$$

Dividendo la relazione appena ricavata per $n(n+1)$ si ottiene, infine

$$\begin{aligned}\frac{\overline{C}(n)}{n+1} &= \frac{\overline{C}(n-1)}{n} + \frac{2}{n+1} = \frac{\overline{C}(n-2)}{n-1} + \frac{2}{n} + \frac{2}{n+1} = \dots \\ &= \frac{\overline{C}(2)}{3} + \dots + \frac{2}{n-1} + \frac{2}{n} + \frac{2}{n+1} = 2\sum_{k=3}^{n+1}\frac{1}{k} = 2\left(H_{n+1} - \frac{3}{2}\right)\end{aligned}$$

dove si è posto

$$\begin{cases} H_n = \displaystyle\sum_{k=1}^{n}\frac{1}{k} \\ H_1 = 1, \quad H_2 = \displaystyle\sum_{k=1}^{2}\frac{1}{k} = 1 + \frac{1}{2} = \frac{3}{2} \end{cases}$$

Concludendo si ottiene:

$$\overline{C}(n) = 2(n+1)\left(H_{n+1} - \frac{3}{2}\right).$$

Notando che:

$$\int_3^{n+2}\frac{1}{x}\,dx < \sum_{k=3}^{n+1}\frac{1}{k} < \int_2^{n+1}\frac{1}{x}\,dx,$$

si ha infine la disuguaglianza

$$\log_e(n+2) - \log_e 3 < \frac{\overline{C}(n)}{2(n+1)} < \log_e(n+1) - \log_e 2$$

per cui, con un semplice cambiamento della base del logaritmo, si ottiene la valutazione della complessità nel caso medio, pari a $\mathcal{O}(n\log_2 n)$. Pertanto la complessità nel caso medio si mantiene, in ordine di grandezza, uguale a quella del caso ottimo.

2 Divide et Impera

Vi sono situazioni caratteristiche in cui un problema si ripropone al suo interno in sottoproblemi uguali all'originale, ma applicati a sottoinsiemi dei dati, e la soluzione globale si ottiene come combinazione delle soluzioni dei sottoproblemi. Un algoritmo organizzato secondo tale metodo, ha la struttura tipo della seguente procedura pseudo–codificata:

```
procedure DIVETIMP(S);
    begin
1.      if |S| < k then
2.          begin *Risolvi direttamente il problema*;
3.              return *Risultato*
            end
                else
4.          begin *Dividi S in sottoinsiemi S1, S2,..., Sh *;
5.              return *Il risultato di una combinazione di DIVETIMP(S1),
                        DIVETIMP(S2),..., DIVETIMP(Sh)*
            end
    end;
```

L'algoritmo opera su un insieme finito S; k è un limite prefissato alla cardinalità di S, al di sotto del quale il problema è risolto con elaborazioni dirette.
Il metodo è spontaneamente formulato in un linguaggio che accetta la ricorsività, il che presenta pregi e difetti. Ricordiamo comunque che non si accresce in potenza il modello computazionale ammettendo che esso includa la ricorsività, poiché è sempre possibile scrivere un programma non ricorsivo che contenga *esplicitamente* tutte le istruzioni eseguite dal corrispondente programma ricorsivo.

Pregi

- Il primo vantaggio offerto dalla ricorsività è la facilità di formulazione del metodo, previa una minima esperienza nella formalizzazione ricorsiva degli algoritmi ⇒ Alta probabilità di non commettere errori.

- Un altro considerevole vantaggio è la semplicità con cui spesso è possibile costruire dimostrazioni convincenti della correttezza di algoritmi ricorsivi. Nel caso di DIVETIMP, tale dimostrazione si basa sul *principio di induzione*: si prova che è corretto il programma se la condizione limite $|S| < k$ è verificata (istruzione (*1.*)), mediante ispezione diretta della soluzione calcolata nell'istruzione (*2.*); quindi supposto corretto il programma per S_1, S_2,..., S_h, si mostra che è corretto per S, mediante l'esame della combinazione dei risultati parziali costruiti nell'istruzione (*5.*).
  È infine possibile esprimere il numero di operazioni che esegue l'algoritmo attraverso semplici relazioni di ricorrenza. Indicando con $T(n)$ il numero di operazioni che si desidera contare (con $n = |S|$, e similmente $n_i = |S_i|$), potremo scrivere:

$$\begin{aligned} T(n) &= c, && \text{per } n < k, \\ T(n) &= D(n) + C(n) + T(n_1) + T(n_2) + \ldots + T(n_h), && \text{per } n \geq k, \end{aligned}$$

  dove la costante c si calcola esaminando le elaborazioni eseguite nell'istruzione (*2.*); $D(n)$ esprime il lavoro di divisione dell'insieme eseguito nell'istruzione (*4.*); $C(n)$ esprime il lavoro di ricombinazione dei risultati ottenuti nelle chiamate ricorsive su S_1, S_2,..., S_h, eseguito nell'istruzione (*5.*).

Difetti

- La semplicità di formulazione dell'algoritmo si paga direttamente in termini di comprensibilità delle operazioni che il programma esegue. Ciò dipende essenzialmente dal fatto che il problema è definito "*dall'alto*", il che è caratteristico della formulazione ricorsiva, mentre le operazioni sui dati hanno inizio "*dal basso*", e cioè dopo che un sufficiente numero di chiamate ricorsive, interne l'una all'altra, hanno condotto a sottoinsiemi elementari ⇒ In caso di errore nel programma è molto difficile ricostruire le operazioni per controllare manualmente il comportamento sui dati più semplici (istruzioni elementari).

- Inoltre i meccanismi che i sistemi di elaborazione impiegano per eseguire programmi ricorsivi sono in genere piuttosto complessi, e possono rendere l'esecuzione dei programmi molto più lunga e laboriosa di quella degli equivalenti programmi iterativi.

2.1 Mergesort

Abbiamo già studiato la complessità computazionale del *quicksort*, concludendone l'ottimalità nel caso medio ($\mathcal{O}(n \log_2 n)$ confronti), e notando che, viceversa, nel caso peggiore, l'algoritmo opera ricorsivamente su sottoinsiemi sbilanciati e richiede ancora un totale di $\mathcal{O}(n^2)$ confronti.
Fino a questo punto non abbiamo individuato un metodo di ordinamento che operi $\mathcal{O}(n \log_2 n)$ confronti anche nel caso peggiore; la possibilità di raggiungere questo risultato appare legata alla capacità di costruire un metodo che lavori *sempre* su partizioni bilanciate dell'insieme. Si arriva così a definire il

metodo di ordinamento *per fusione* o *mergesort*. Per definire l'algoritmo con precisione, utilizziamo una formulazione ricorsiva che pone *mergesort* tra i più eleganti esempi del "*divide et impera*".
Sia dunque A un insieme di n elementi. Nella formulazione ricorsiva, *mergesort* si applica all'intero insieme e lo ordina mediante la fusione di due semi–insiemi, ordinati ricorsivamente nello stesso modo.

Algoritmo

```
procedure merge(A,p,q,r);
1.      begin i ← p; k ← p; j ← q + 1;
            while (i ≤ q) and (j ≤ r) do
2.              begin if A(i) < A(j) then
3.                  begin B(k) ← A(i);
4.                      i ← i + 1
                    end
                        else
5.                  begin B(k) ← A(j);
6.                      j ← j + 1
                    end;
7.                  k ← k + 1
8.              end;
9.      if i ≤ q then {A(k),...,A(r)} ← {A(i),...,A(q)};
        {A(p),...,A(k − 1)} ← {B(p),...,B(k − 1)}
        end;
```

Si noti che l'insieme generato dalla fusione viene memorizzato in un vettore temporaneo di lavoro B, e quindi ricopiato nell'array A. Possiamo ora scrivere l'algoritmo completo, consistente nella procedura *mergesort(A,p,r)*, $1 \leq p < r \leq n$, che ordina la sezione di A compresa tra le posizioni p e r, facendo uso della procedura appena descritta.

Algoritmo

```
procedure mergesort(A,p,r);
    begin
        if p = r then return;
        q ← ⌊(p + r)/2⌋;
        mergesort (A,p,q);
        mergesort (A,q+1,r);
        merge(A,p,q,r)
    end;
```

L'ordinamento dell'insieme si otterrà quindi con una chiamata esterna alla procedura con $p = 1$ e $r = n$.

Esempio: $A = \{40, 25, 10, 50, 32, 15, 55, 18\}$

40 25 10 50 32 15 55 18
25 40 10 50 32 15 55 18
25 40 10 50 32 15 55 18
10 25 40 50 32 15 55 18
10 25 40 50 15 32 55 18
10 25 40 50 15 32 18 55
10 25 40 50 15 18 32 55
10 15 18 25 32 40 50 55

Si noti che le chiamate ricorsive alla procedura si susseguono una interna all'altra finché si raggiungono insiemi elementari su cui iniziano le vere e proprie operazioni di confronto e ordinamento, con la fusione di due sottoinsiemi ciascuno costituito da un solo elemento. Di qui si passa alla fusione dei sottoinsiemi di due elementi, e così via.

Studiamo ora il numero di confronti, $C(n)$, eseguiti da *mergesort* tra gli elementi del vettore da ordinare, come rappresentativo della complessità in tempo dell'algoritmo. Si ricava immediatamente la relazione di ricorrenza

$$\begin{cases} C(1) = 0, \\ C(n) = 2C(n/2) + M(n) + c, \quad \text{per } n > 1, \end{cases}$$

dove con $M(n)$ si indica il numero di confronti richiesti dalla procedura *merge*. In questa procedura i confronti tra $A(i)$ e $A(j)$ si eseguono nell'istruzione (*2.*), e per ogni confronto si elimina un elemento dalla prima sezione di A (istruzione (*3.*)) o dalla seconda sezione di A (istruzione (*5.*)). Il numero di tali confronti è perciò limitato superiormente da $n-1$. Pertanto supponendo, senza perdita di generalità, $n = 2^k$, si ha:

$$\begin{aligned} C(n) &= 2(2C(n/4) + n/2 + c) + n + c = 2(2(2C(n/8) + n/4 + c) + n/2 + c) + n + c = \dots \\ &= 2^k C(1) + n\sum_{i=0}^{k-1} 1 + c\sum_{i=0}^{k-1} 2^i = (C(1) + c)n + n\log_2 n - c = \mathcal{O}(n \log_2 n). \end{aligned}$$

Mergesort è quindi un *algoritmo di ordinamento ottimo anche nel caso pessimo*: il motivo è da ricercarsi ovviamente nel fatto che la partizione operata in questo caso nei due sottoinsiemi delimitati tra p, q e $q+1$, r è sempre bilanciata.

2.2 Heap

Esaminiamo ora una struttura informativa introdotta da Williams (1964) e da questi denominata *heap* (dall'inglese "*mucchio*"). Lo heap ospita gli elementi di un insieme A di cardinalità n, su cui è definita una relazione di ordinamento. Valgono per esso le seguenti due proprietà.

Proprietà 1 *Lo heap è un albero quasi perfettamente bilanciato (fino al livello $k-1$ contiene il numero massimo di nodi, cioè $2^k - 1$ in totale, e al livello k ne contiene un numero compreso fra 1 e 2^k) e i suoi nodi a livello massimo sono tutti "addossati a sinistra".*

Proprietà 2 *Ogni nodo contiene un elemento minore dell'elemento contenuto nel nodo padre.*

Osservazioni

- Noto il valore di n, la forma dell'albero è fissata dalla Proprietà 1; l'allocazione degli elementi nei nodi può invece variare, purché rispetti la Proprietà 2: l'elemento massimo dell'insieme è comunque sempre allocato nella radice. Si noti che nello heap i sottoalberi di ciascun nodo sono ancora heap.
- La Proprietà 1 consente di dare allo heap una semplicissima rappresentazione sequenziale, disponendo gli elementi di A nell'ordine in cui questi si incontrano esaminando l'albero per livelli crescenti, e scorrendo da sinistra verso destra i nodi in ogni livello. Secondo questa allocazione, $A(1)$ è l'elemento nella radice e, per ogni elemento $A(i)$, gli elementi corrispondenti ai figli sinistro e destro, se esistono, sono rispettivamente in $A(2i)$ e $A(2i+1)$: se $2i > n$ e/o $2i+1 > n$, il figlio sinistro e/o destro di $A(i)$ non esiste. Per la precedente Proprietà 2 si ha dunque: $A(2i) < A(i)$, $A(2i+1) < A(i)$, ovunque tali elementi siano definiti.

Esempio
La struttura sequenziale descritta nell'esempio è semplicissima e sarà impiegata per costruire tutti gli algoritmi sullo heap.

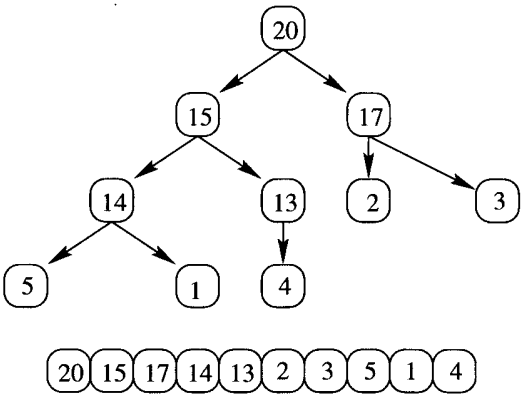

Figura 1: Un esempio di heap $A = \{20, 15, 17, 14, 13, 2, 3, 5, 1, 4\}$.

2.2.1 Code a priorità

Nella sua accezione più semplice la coda permette in ogni istante l'estrazione del primo elemento inserito tra quelli correntemente contenuti nella struttura, con lo stesso meccanismo secondo cui evolve una coda di persone in attesa ad uno sportello.
Il meccanismo di coda si generalizza assegnando ad ogni elemento una priorità e richiedendo in estrazione l'elemento a più alta priorità tra i presenti: la struttura che ne deriva è detta *coda con priorità*. Questa nuova struttura richiede un tipo di allocazione in memoria più complicato della semplice lista con puntatori alla testa (per l'estrazione) ed alla coda (per l'inserzione), poiché per eseguire l'eliminazione si deve conoscere la posizione dell'elemento a più alta priorità; estratto questo, si deve individuare l'elemento immediatamente seguente per la successiva eliminazione, e così via. Se si impiegasse una semplice lista, l'inserzione potrebbe essere eseguita ancora in tempo costante, ma l'eliminazione richiederebbe un tempo proporzionale a $\mathcal{O}(n)$, non prevedibile in linea di principio. Vediamo invece come lo heap ben si adatti alla memorizzazione delle code a priorità.
Nei nodi dello heap si allocano gli elementi della coda, e i valori di ordinamento per lo heap sono quelli di priorità per la coda. Rappresenteremo quindi la priorità con numeri interi e identificheremo, per semplicità, gli elementi con le loro priorità. È evidente che l'elemento massimo, quello da estrarre alla prima richiesta, è immediatamente accessibile nella radice. Vediamo come eseguire l'inserzione di un nuovo elemento, e come estrarre l'elemento massimo, ricostituendo in entrambi i casi la struttura di heap.
L'inserzione di un nuovo elemento W avviene creando una nuova foglia contenente W, nella prima posizione libera dello heap; si confronta, quindi, W con il proprio padre Z, scambiando gli elementi dello heap se $W > Z$ e ripetendo l'operazione finché W diviene minore del padre corrente, o raggiunge la radice.

Esempio: Inserimento e ricostruzione dello heap

20	15	17	14	13	2	3	5	1	4	<u>16</u>
20	15	17	14	16	2	3	5	1	4	13
20	16	17	14	15	2	3	5	1	4	13

Algoritmo

procedure heapinser(A,n,W);
    begin $n \leftarrow n+1$;
        $A(n) \leftarrow W$;
        $j \leftarrow n$;
        while $(j > 1)$ *and* $(A(j) > A(\lfloor j/2 \rfloor))$ *do*
            begin *Scambia $(A(j)$ *con* $A(\lfloor j/2 \rfloor)$*;
                $j \leftarrow \lfloor j/2 \rfloor$
            end
    end;

L'algoritmo ha complessità in tempo pari a $\mathcal{O}(\log_2 n)$ nel caso pessimo, poiché è basato sulle ripetizioni del ciclo *while*, eseguite sui nodi di un percorso ascendente che da una foglia giunge al massimo alla radice dello heap. Benché vi sia una perdita di efficienza rispetto all'allocazione della coda in una lista (inserzione in tempo costante), l'impiego dello heap è giustificato dall'aumento di efficienza dell'operazione di estrazione.
Sappiamo che l'elemento da estrarre è il massimo, e che questo si trova nella radice. Tolto il massimo, si trasferisce al suo posto l'elemento W contenuto nell'ultima foglia, ottenendo così un albero che non necessariamente è un heap, poiché W può essere minore degli elementi X e Y contenuti nei figli. Se ciò accade, si scambiano di posizione W ed il massimo tra X e Y e si procede nell'operazione di scambio fra W e i suoi nuovi figli finché W si mantiene minore.

Esempio: Estrazione e ricostruzione dello heap

<u>20</u>	15	17	14	13	2	3	5	1	4
4	15	17	14	16	2	3	5	1	
17	15	4	14	15	2	3	5	1	

L'estrazione del massimo elemento di uno heap può essere eseguita come segue:
$Z \leftarrow A(1)$;
$A(1) \leftarrow A(n)$;
heapricost(A,1,n − 1);
utilizzando la procedura *heapricost* descritta di seguito.

Algoritmo

procedure heapricost(A,i,j);
    begin
        if $i > j/2$ *then return;*
        Sia $A(k) = \max(A(2i), A(2i+1))$ *se esiste,*
        cioè $k = 2i$ *oppure* $k = 2i+1$ *;
        if $A(k) > A(i)$ *then*
            begin
                *Scambia $A(i)$ e $A(k)$ *;
                heapricost(A,k,j)
            end
    end;

Per valutare la complessità in tempo della procedura *heapricost* nel caso pessimo, notiamo che essa compie il massimo numero di passi quando $k = 2i$ e $A(k) > A(i)$ in ogni chiamata ricorsiva, finché si raggiunge la chiusura $i > j/2$. Tale complessità $T(i,j)$ è funzione di i e j ed è regolata dalla relazione di

ricorrenza

$$T(i,j) = a, \qquad \text{per } i > j/2,$$
$$T(i,j) = T(2i,j) + b, \quad \text{per } i \leq j/2,$$

con a e b costanti, $i \leq j$. Da questa relazione otteniamo per sostituzioni successive:

$$T(i,j) = T(2i,j) + b = T(4i,j) + b + b = \ldots = T(2^h i, j) + bh = a + bh$$

ove h è l'intero tale che $2^h i > j/2$ e $2^{h-1} i \leq j/2$, ovvero $log_2(j/i) - 1 < h \leq log_2(j/i)$, e cioè $h = \lfloor \log_2(j/i) \rfloor$. Abbiamo così $T(i,j) = a + b\lfloor \log_2(j/i) \rfloor$.
Questo risultato non è sorprendente poiché la procedura sopra descritta è basata su una discesa lungo l'albero e $\lfloor \log_2(j/i) \rfloor$ è pari al numero di nodi che si incontrano nel percorso che da $A(i)$ scende fino ad $A(j)$. Dalla relazione ottenuta si deduce che l'estrazione del massimo dello heap, e la sua ricostituzione, può essere eseguita con un costo pari a $\mathcal{O}(\log_2 n)$.
Complessivamente dunque la struttura di heap consente di gestire code con priorità in modo che entrambe le operazioni di inserzione ed estrazione di elementi richiedano, nel caso peggiore, tempo logaritmico.

2.2.2 Costruzione dello heap

Consideriamo ora il problema di costruire uno heap di n elementi. Se procediamo per successive inserzioni, partendo da uno heap vuoto, in n passi abbiamo costruito lo heap cercato: ogni passo realizzato con l'algoritmo di inserzione descritto ha, nel caso peggiore, complessità proporzionale a $\log_2 i$, ove i è il numero di elementi contenuti nello heap, cioè inseriti fino a quel momento. Complessivamente il procedimento richiede un tempo $T(n) = c\sum_{i=1}^{n} \log_2 i$, che è proporzionale a $n \log_2 n$, come segue dalla coppia di disuguaglianze:

$$\frac{n}{2} \log_2 \frac{n}{2} < \sum_{i=1}^{n} \log_2 i < n \log_2 n.$$

Viceversa, lo heap può essere costruito con una tecnica differente, che presenta una complessità proporzionale ad n. Questa costruzione parte da un albero binario, che ha la forma di heap di n nodi, ma con gli elementi allocati in modo casuale, e utilizza n volte la procedura *heapricost* per permutare gli elementi, a partire da quelli al massimo livello, fino a ricostituire interamente uno heap.

Algoritmo

```
procedure heap(A,n);
    begin i ← n;
        while i ≥ 1 do
            begin heapricost(A,i,n);
                i ← i − 1
            end
    end;
```

Calcoliamo la complessità in tempo $H(n)$ della procedura *heap*. Si ha:

$$H(n) = c + \sum_{i=1}^{n} (a + b\lfloor \log_2(n/i) \rfloor + d) = A\sum_{i=1}^{n} \lfloor \log_2(n/i) \rfloor + Bn + C = AS(n) + Bn + C,$$

con A, B, C costanti. Sviluppiamo il calcolo di $S(n)$ e, senza perdita di generalità, supponiamo $n = 2^k - 1$:

$$\begin{aligned} S(n) &= \sum_{i=1}^{n} \lfloor \log_2((2^k - 1)/i) \rfloor \\ &= \sum_{i=2^0}^{2^1-1} \lfloor \log_2((2^k - 1)/i) \rfloor + \sum_{i=2^1}^{2^2-1} \lfloor \log_2((2^k - 1)/i) \rfloor + \ldots + \sum_{i=2^{k-1}}^{2^k-1} \lfloor \log_2((2^k - 1)/i) \rfloor. \end{aligned}$$

Nell'ultima espressione ogni sommatoria calcolata per i valori di i compresi tra 2^{s-1} e $2^s - 1$, con $1 \leq s \leq k$, ha i termini tutti uguali tra loro, a causa dell'approssimazione di questi all'intero inferiore. In conclusione, risulta:

$$S(n) = (k-1) + 2 \times (k-2) + 2^2 \times (k-3) + \ldots + 2^{k-2} \times 1 + 2^{k-1} \times 0 = 2^k - k - 1 = n - \log_2(n+1),$$

come si può provare per induzione. Combinando questo risultato con quanto ottenuto precedentemente per $H(n)$, otteniamo complessivamente $H(n) \in \mathcal{O}(n)$, ovvero si riesce a costruire lo heap in tempo lineare.

2.2.3 Heapsort

Lo heap trova la sua applicazione più elegante nel metodo di ordinamento noto con il nome di *heapsort*. Esso consiste nell'estrarre il massimo dallo heap, salvarlo all'esterno come massimo dell'insieme e ricostituire lo heap; ripetere quindi le operazioni relativamente a tutti gli elementi successivi. Poiché ogni estrazione e ricostituzione dello heap richiede un tempo pari a $\mathcal{O}(\log_2 n')$, se n' è il numero di elementi attualmente contenuti nello heap, l'ordinamento avrà complessità dell'ordine di $\mathcal{O}(n \log_2 n)$. Quindi *heapsort* è ottimo, come *mergesort*, anche nel caso pessimo. Inoltre *heapsort* può essere eseguito sulla struttura sequenziale dello heap.

Algoritmo

```
procedure heapsort(A,n);
    begin heap(A,n);
        i ← n;
        while i ≥ 2 do
            begin *Scambia A(1) con A(i)*;
                heapricost(A,1,i − 1);
                i ← i − 1
            end
    end;
```

La procedura *heapsort* richiede tempo lineare per la costruzione dello heap (chiamata alla procedura *heap(A,n)*), e tempo dell'ordine di $\mathcal{O}(n \log_2 n)$ per il successivo ordinamento, dove il ciclo *while* si ripete $n - 1$ volte, e ciascuna impiega un tempo logaritmico (chiamata alla procedura *heapricost(A,1,i − 1)*).