Teoria della complessità

Analisi di complessità di algoritmi fondamentali

Donato Malerba

Algoritmi di Ordinamento

- L'ordinamento è una delle attività di elaborazione più importanti in quanto si stima che circa il 30% del tempo di calcolo impiegato da un elaboratore è speso in questa attività.
- Gli algoritmi di ordinamento più semplici sono caratterizzati dal fatto che essi richiedono un numero di confronti proporzionale a n^2 per ordinare n elementi. Altri algoritmi meno intuitivi sono invece ottimi in ordine di grandezza. Gli algoritmi migliori sono quelli in grado di scambiare, nelle prime fasi di ordinamento, valori che occupano posizioni molto distanti nella lista.

Algoritmi di Ordinamento

- Infatti è stato dimostrato che in media, per dati casuali, gli elementi devono essere spostati di una distanza pari a n/3 rispetto alla loro posizione originaria. Gli algoritmi più semplici e meno efficienti tendono a spostare solo gli elementi vicini.

Ordinamento per selezione

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

Operazione dominante: confronto

Ordinamento per selezione

- Il numero di confronti eseguiti non dipende dalla configurazione dei dati di ingresso (si tratta di due cicli for).
- La prima volta in cui è eseguito il ciclo più interno si fanno n-1 confronti, la seconda volta n-2, …, infine un solo confronto.
- Quindi il numero di confronti è dato dalla somma di Gauss:

$$(n-1) + (n-2) + \ldots + 1 = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$$

L'algoritmo ha una complessità $\Theta(n^2)$ sotto assunzione di costo uniforme.

Ordinamento per scambi successivi

```
procedure bubblesort(var a: nelements; n: integer);
var i, j, t: integer;
    sorted: boolean
begin
   sorted := false; i :=0;
   while (i<n) and (not sorted) do
   begin
        sorted := true;
        i := i+1;
        for j := 1 to n-i do
                if a[j]>a[j+1] then
                        begin
                           t :=a[j]; a[j] := a[j+1]; a[j+1] :=t; sorted:=false;
                        end;
   end
end
```

Operazione dominante: confronto

Ordinamento per scambi successivi

- Il numero di confronti eseguiti dipende dalla configurazione dei dati di ingresso.
- Se i dati sono ordinati il numero di confronti è n-1:
  $f_{ott}(n)=n-1$
- Se i dati in ordine inverso, la prima volta in cui è eseguito il ciclo più interno si fanno n-1 confronti, la seconda volta n-2, …, infine un solo confronto.
- Quindi il numero di confronti, nel caso pessimo, è dato dalla somma di Gauss:

$(n-1) + (n-2) + \ldots + 1 = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$

Nel caso pessimo, l'algoritmo ha una complessità $\Theta(n^2)$ sotto assunzione di costo uniforme.

Ordinamento per inserzione

```
procedure insertsort(var a: nelements; n: integer);
var i, j, first, p, x: integer;
begin
   first:=a[1]; p :=1;
   for i:=2 to n do
        if a[i]<first then
                begin
                    first := a[i]; p:=i
                end;
        a[p] := a[1]; a[1] := first;
```

Operazione dominante: confronto

Ordinamento per inserzione

```
for i:= 3 to n do
    begin
        x :=a[i]; j := i;
        while x<a[j-1] do
            begin
                a[j] := a[j-1]; j:=j-1;
                end;
        a[j] :=x
end
end
```

Operazione dominante: confronto

Ordinamento per inserzione

- Il numero di confronti eseguiti dipende dalla configurazione dei dati di ingresso.
- Se i dati sono ordinati il numero di confronti è
  - n-1 ← primo ciclo for che inizializza la "sentinella"
  - n-2 ← il ciclo while più interno dev'essere eseguito almeno una volta per ogni valore di i.
  - 2n-3 ← totale

  Quindi $f_{ott}(n)=2n-3$
- Se i dati sono ordinati in ordine inverso, la prima volta in cui è eseguito il ciclo while più interno si fanno 2 confronti, la seconda volta 3, …, infine n-1 confronti.

Ordinamento per inserzione

Quindi il numero massimo di confronti nel secondo ciclo, è dato dalla somma di Gauss:

$$2+3+\ldots+n-1=\sum_{i=2}^{n-1} i=\sum_{i=1}^{n-1} i-1=\frac{n(n-1)}{2}-1$$

Nel caso pessimo, l'algoritmo ha una complessità

$$f_{pess}(n)=n-1+\frac{n^2-n}{2}-1=\frac{n^2+n-4}{2}$$

cioè è un $\Theta(n^2)$ sotto assunzione di costo uniforme.

- Per condurre un'analisi di costo medio, assumiamo che la probabilità che a[i], i=3, 4, …, n venga inserito in posizione k-esima, k=2, 3, …, i, sia la stessa, indipendentemente da i e k. Quindi definito l'evento

E_k : a[i] va inserito in k-esima posizione

Ordinamento per inserzione

La complessità media per inserire correttamente *a[i]*, *i*=3, 4, …, *n* è data da:

$$\sum_{k=2}^{i} \text{costo}(E_k)P(E_k)=\sum_{k=2}^{i}(i-k+1)\frac{1}{i-1}=$$

$$=\frac{1}{i-1}\left[\sum_{k=2}^{i} i-\sum_{k=2}^{i} k+\sum_{k=2}^{i} 1\right]=\frac{1}{i-1}\left\{i(i-1)-\left[\frac{i(i+1)}{2}-1\right]+(i-1)\right\}=$$

$$=\frac{1}{i-1}\left(i^2-i-\frac{i^2}{2}-\frac{i}{2}+1+i-1\right)=\frac{1}{i-1}\frac{i^2-i}{2}=\frac{i}{2}$$

Ordinamento per inserzione

La complessità nel caso medio è quindi data da:

$$f_{med}(n) = n - 1 + \sum_{i=3}^{n} \frac{i}{2} = n - 1 + \frac{1}{2}\left(\frac{n(n+1)}{2} - 3\right) =$$

$$= n - 1 + \frac{n^2 + n - 6}{4} = \frac{n^2 + 5n - 10}{4}$$

cioè è dell'ordine $\Theta(n^2)$.

L'ordinamento per inserzione è di solito considerato il migliore degli algoritmi con complessità $\Theta(n^2)$ per ordinare piccoli insiemi con scarso preordinamento.

Ordinamento a diminuzione di incremento

- La strategia è quella di individuare nell'array inc catene di elementi a distanza inc l'uno dall'altro.
- Ogni catena viene ordinata mediante un algoritmo efficiente.
- La distanza fra gli elementi diminuisce progressivamente fino a quando si giunge a una sola catena di elementi a distanza unitaria.
- In questo modo si ordinano prima gli elementi distanti e poi via via gli elementi più vicini.
- L'ordinamento di catene "più fitte" può avvantaggiarsi di dell'ordinamento pregresso su catene "più rade". Per questo è raccomandabile l'uso di una variante dell'insertsort per ordinare ogni catena.

Ordinamento a diminuzione di incremento

```
procedure shellsort(var a: nelements; n: integer);
var inc, corrente, precedente, j, k, x: integer;
      inserito: boolean;
begin
  inc:=n;
  while inc>1 do
    begin
      inc:=inc div 2;
      for j:=1 to inc do
        begin
          k:= j + inc;
          while k<=n do
                    begin
                        inserito := false; x:=a[k]; corrente := k;
                        precedente := corrente –inc;
```

Ordinamento a diminuzione di incremento

```
                    while precedente>=j and (not inserito) do
                      begin
                          if x<a[precedente] then
                                begin
                                      a[corrente] := a[precedente];
                                      corrente := precedente;
                                      precedente := precedente – inc;
                                end
                          else
                              inserito := true
                      end;
                    a[corrente] := x;
                    k := k+inc
                end
          end
      end
end
```

Operazione dominante: confronto

Ordinamento a diminuzione di incremento

Per questo algoritmo ci limitiamo a dimostrare che un limite superiore alla complessità nel caso pessimo è dato da n^2. Infatti il numero di volte in cui il ciclo più esterno è eseguito è dato da:

$m = \log_2 n$

Allo i-esimo ciclo esterno si avranno $\frac{n}{2^i}$ catene di 2^i elementi.

L'ordinamento di una catena ha un costo massimo dato da:

$$\frac{2^i(2^i-1)}{2}$$

Ordinamento a diminuzione di incremento

Quindi il costo complessivo è dato da:

$$\sum_{i=1}^{m}\left(\frac{n}{2^i}\frac{2^i(2^i-1)}{2}\right) = \frac{n}{2}\sum_{i=1}^{m}(2^i-1) =$$

$$= \frac{n}{2}\sum_{i=1}^{m}2^i - \frac{n}{2}\sum_{i=1}^{m}1 = \frac{n}{2}\frac{1-2^{m+1}}{1-2} - \frac{n}{2}\log_2 n =$$

$$= \frac{n}{2}(2^m 2-1) - \frac{n}{2}\log_2 n = n^2 - \frac{n}{2} - \frac{n}{2}\log_2 n$$

Pertanto $f_{pess}(n) = O(n^2)$.

Ordinamento a diminuzione di incremento

Analizzando l'esempio illustrato precedentemente si può osservare che le due catene di lunghezza $n/4$ devono essere necessariamente parzialmente ordinate in modo crescente in quanto includono quelle ordinate di lunghezza $n/2$.

L'efficienza dello shellsort dipende dalla scelta delle lunghezze delle catene. Si può mostrare che ci sono sequenze di decrementi migliori di:

$$\frac{n}{2}, \frac{n}{4}, \frac{n}{8}, \ldots, 1$$

In particolare, per la sequenza:

2^p-1, …, 31, 15, 7, 3, 1 con $p=\lfloor \log_2 n \rfloor$

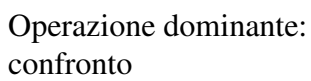
lo shellsort non fa più di $n^{1.5}$ confronti, cioè $f_{pess}(n) = O(n^{1.5})$.

Ordinamento per partizionamento

```
procedure quicksort(var a: nelements; minsin,maxdes: integer);
var maxsin, mindes: integer;
begin
    if minsin<maxdes then
      begin
          partition(a, minsin, maxdes, maxsin, mindes);
          if (maxsin-minsin) < (maxdes-mindes) then
             begin
                   quicksort(a, minsin, maxsin); quicksort(a, mindes, maxdes);
             end
          else
              begin
                   quicksort(a, mindes, maxdes); quicksort(a, minsin, maxsin);
              end
       end
end
```

Ordinamento per partizionamento

```
procedure partition(var a: nelements; l,u: integer; var i,j: integer );
var k, t, x: integer;
begin
   k := (l+u) div 2; x := a[k]; i := l; j := u;
   while a[i] < x do i:=i+1;
   while x < a[j] do j:=j -1;
   while i < j-1 do
      begin
        t:=a[i]; a[i]:=a[j]; a[j]:=t;
        i := i+1; j:= j+1;
        while a[i]<x do i:=i+1;
        while x<a[j] do j:=j-1;
      end;
```



Ordinamento per partizionamento

```
   if i<=j then
      begin
        if i<j then
           begin
                t:=a[i]; a[i]:=a[j]; a[j]:=t;
           end
        i:=i+1; j:=j-1
      end
end
```

Ordinamento per partizionamento

La complessità del quicksort dipende dalla distribuzione dei dati.

Analizziamo il caso medio e consideriamo come operazione dominante il confronto con elementi dell'array **a** (vedi test a[i]<x, x<a[j] nella procedura *partition*).

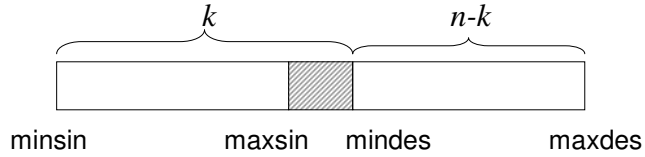
Sia *n=maxdes-minsin* il numero di elementi del segmento di **a** da ordinare. Allora la complessità in tempo per l'esecuzione della procedura *partition* è data da:

$g_{ott}(n)=n$ $g_{med}(n)=n+1$ $g_{pess}(n)=n+2$

in quanto ***x*** dovrà essere confrontato con almeno tutti gli ***n*** elementi del segmento di a da partizionare (si hanno due confronti in più quando gli indici si incrociano, *j<i*).

Ordinamento per partizionamento

Supponiamo che al temine della procedura *partition* abbiamo:



La probabilità che il segmento sia partizionato in

- partizione sinistra di dimensione k-1
- partizione destra di dimensione n-k

è, sotto assunzione di distribuzione uniforme di probabilità, la stessa per ogni k, con $1 \leq k \leq n$, quindi:

$P($dim. partiz. sin $= k-1 \wedge$ dim. partiz. destra $= n-k) = 1/n$

Ordinamento per partizionamento

Allora la complessità dell'algoritmo quicksort può essere espressa mediante la seguente *relazione di ricorrenza*:

$$f_{med}(n) = \sum_{k=1}^{n} \frac{1}{n} \left(g_{med}(n) + f_{med}(k-1) + f_{med}(n-k) \right) =$$

$$= g_{med}(n) + \frac{1}{n} \sum_{k=1}^{n} \left(f_{med}(k-1) + f_{med}(n-k) \right)$$

per n≥2, mentre $f_{\text{med}}(1) = f_{\text{med}}(0) = 0$

Ordinamento per partizionamento

Per semplicità indicheremo di seguito f_{med} con f.
L'equazione ricorrente può essere risolta in pochi passi. Infatti:

$$f(n) = n + 1 + \frac{1}{n} \sum_{k=1}^{n} f(k-1) + \frac{1}{n} \sum_{k=1}^{n} f(n-k)$$

ma $$\sum_{k=1}^{n} f(k-1) = f(0) + f(1) + \ldots + f(n-1) =$$

$$= f(n-1) + f(n-2) + \ldots + f(1) + f(0) = \sum_{k=1}^{n} f(n-k)$$

quindi $$f(n) = n + 1 + \frac{2}{n} \sum_{k=1}^{n} f(k-1)$$

Ordinamento per partizionamento

da cui:

(i) $$nf(n) = (n+1)n + 2\sum_{k=1}^{n} f(k-1)$$

Ma allora dev'essere anche vero che:

(ii) $$(n-1)f(n-1) = n(n-1) + 2\sum_{k=1}^{n-1} f(k-1)$$

così, sottraendo *(ii)* a *(i)* si ha:

$$nf(n) - (n-1)f(n-1) = n(n+1) + 2\sum_{k=1}^{n} f(k-1) - n(n-1) - 2\sum_{k=1}^{n-1} f(k-1)$$

che è equivalente a:

$$nf(n) = (n+1)f(n-1) + 2n$$

Ordinamento per partizionamento

e dividendo ambo i membri per *n(n+1)* si ha:

$$\frac{f(n)}{n+1} = \frac{f(n-1)}{n} + \frac{2}{n+1} = \frac{f(n-2)}{n-1} + \frac{2}{n} + \frac{2}{n+1} =$$

$$= \frac{f(n-3)}{n-2} + \frac{2}{n-1} + \frac{2}{n} + \frac{2}{n+1} = \ldots$$

$$\ldots = \frac{f(2)}{3} + \sum_{k=3}^{n} \frac{2}{k+1} = 1 + 2\sum_{k=3}^{n} \frac{1}{k+1} \approx 2\sum_{k=1}^{n} \frac{1}{k} \approx 2\int_{1}^{n} \frac{1}{x} dx = 2\ln(n)$$

Quindi:

$f_{med}(n) \approx 2n\ln(n)$

Ordinamento per fusioni successive

```
procedure mergesort(var a: nelements; primo,ultimo: integer);
var q: integer;
begin
   if primo<ultimo then
     begin
        q := (primo+ultimo) div 2;
        mergesort(a, primo, q)
        mergesort(a, q+1, ultimo);
        merge(a, primo, ultimo, q)
     end
end
```

Ordinamento per fusioni successive

```
procedure merge(var a: nelements; primo,ultimo,mezzo: integer);
var i, j, k, h: integer;
     b: nelements;
begin
   i := primo; k := primo; j:=mezzo+1;
   while (i<=mezzo) and (j <= ultimo) do
     begin
        if a[i]<a[j] then
           begin
                 b[k] := a[i]; i := i+1
           end
        else
           begin
                 b[k] := a[j]; j := j+1
           end
        k := k+1;
     end;
```

Operazione dominante: confronto

Ordinamento per fusioni successive

```
if i<=mezzo then
   begin
     j := ultimo –k;
     for h:=j downto 0 do a[k+h] := a[i+h]
   end
     for j:=primo to k-1 do a[j] := b[j]
end
```

Ordinamento per fusioni successive

Se il tempo per l'operazione di fusione è proporzionale ad n, la dimensione dell'array da ordinare, allora la complessità computazionale della procedura mergesort è descritta mediante la seguente *relazione di ricorrenza*:

$$f(n) = \begin{cases} a & n = 1, a \text{ costante} \\ 2f(n/2) + cn & n > 1, \text{c costante} \end{cases}$$

Quando n è una potenza di 2, $n=2^k$, possiamo risolvere l'equazione attraverso sostituzioni successive

Ordinamento per fusioni successive

$$f(n) = 2\left(2f(\frac{n}{4}) + c\frac{n}{2}\right) + cn = 4f(\frac{n}{4}) + 2cn =$$

$$= 4\left(2f(\frac{n}{8}) + c\frac{n}{4}\right) + 2cn = 8f(\frac{n}{8}) + 3cn = \ldots$$

$$\ldots = 2^k f(1) + kcn = an + cn\log_2 n$$

Si mostra facilmente che se $2^k \le n \le 2^{k+1}$ allora

$$f(n) \le f(2^{k+1})$$

quindi f(n) è un O($n\log_2 n$).

Mergesort ha una complessità asintotica, in tempo, proporzionale a $n\log_2 n$ anche nel caso pessimo.

Ordinamento per fusioni successive

Che dire della complessità in spazio?

Osserviamo che ad ogni chiamata della procedura MERGE viene allocata una variabile locale ***b*** della dimensione del vettore ***a*** da ordinare.

Poiché abbiamo $\lfloor \log_2 n \rfloor$ chiamate ricorsive di MERGESORT (e quindi di MERGE) la complessità in spazio è pari a $n\log_2 n$ (al contrario degli algoritmi SELECTSORT, BUBBLESORT e INSERTSORT che hanno complessità lineare in spazio).

Tuttavia ad ogni chiamata ricorsiva si usa solo una parte di ***b***, quella relativa ai due segmenti da fondere; al termine della chiamata ricorsiva ***b*** viene perso. Quindi è preferibile allocare ***b*** solo una volta all'atto della chiamata della