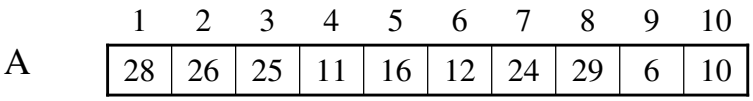
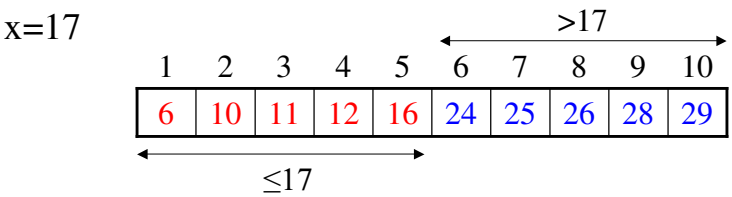
Verifica di correttezza di algoritmi fondamentali

Partizionamento di array

Sia dato un array non ordinato di n elementi: partizionare gli elementi in due sottoinsiemi così che gli elementi ≤x siano in un sottoinsieme e quelli >x nell'altro sottoinsieme.





Partizionamento di array

Un modo sarebbe quello di ordinare l'array in ordine crescente e trovare a questo punto la localizzazione (indice) dell'elemento che separa i due subset.

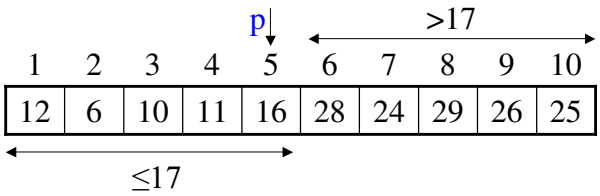
Osservazione: Il problema dell'ordinamento ha una complessità $n \cdot log\ n$, cioè si effettua un numero di confronti proporzionale almeno a $n \cdot log\ n$, dove n è il numero di elementi.

Possiamo trovare una soluzione più efficiente?

In particolare, cerchiamo una soluzione che faccia un numero di confronti proporzionale a n.

Partizionamento di array

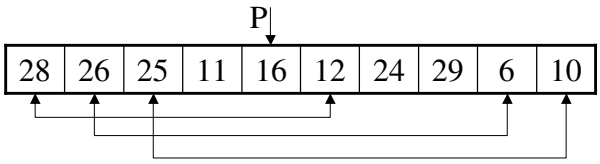
Osserviamo che non è necessario ordinare l'array. Infatti anche questa soluzione "non ordinata" è accettabile:



Per conoscere il valore di p basterà contare gli elementi ≤x. Sapendo il valore di p sappiamo che i valori ≤x a sinistra di p non vanno spostati, mentre ogni volta che incontriamo a sinistra di p un valore >x, questo va spostato a destra di p scambiandolo con un valore a destra di p che è ≤x.

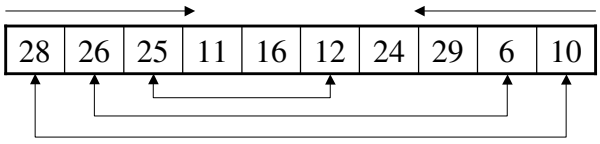
Partizionamento di array

È così necessario scorrere l'array 2 volte.

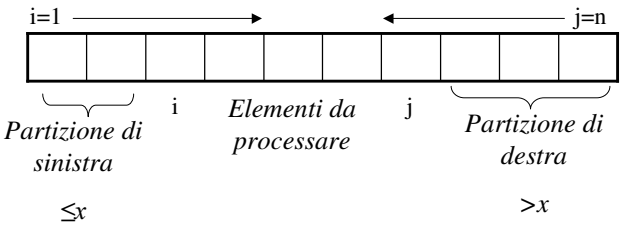


Per economizzare, si può partire dai due estremi e procedendo ad effettuare gli scambi se c'è necessità.

Partizionamento di array



Così si estendono la partizione sinistra e quella destra finchè ci sono ancora elementi da processare.



Partizionamento di array

In questo modo si scorre l'array una sola volta basandosi su un meccanismo del tipo:

MENTRE le due partizioni non si incontrano ESEGUI

a) estendi le partizioni di sinistra e destra scambiando le coppie piazzate male

Per la partizione sinistra si utilizza un indice i che si incrementerà via via, mentre per quella destra un indice j che verrà decrementato.

L'estensione della partizione sinistra avverrà mediante il ciclo

```
while a[i]≤x do i:=i+1;
```

mentre l'estensione della partizione destra avverrà mediante il ciclo

Partizionamento di array

```
while a[j]>x do j:=j-1;
```

i parte da 1 e j da n.

Nel caso sia necessario si effettua lo scambio:

```
t:=a[i];
a[i]:=a[j];
a[j]:=t
```

Vanno fatte alcune considerazioni circa la terminazione del ciclo: l'indice j dev'essere limite superiore per la partizione sinistra.

Quindi possiamo scrivere:

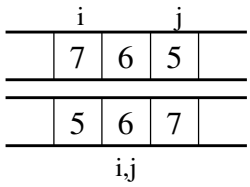
Partizionamento di array

```
i:=1
j:=n
while i<j do
        begin
                while a[i]≤x do i:=i+1;
                while a[j]>x do j:=j-1;
                t:=a[i];
                a[i]:=a[j];
                a[j]:=t;
                i:=i+1; j:=i-1
        end
```

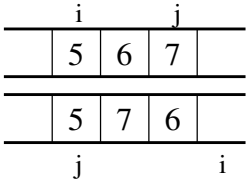

Partizionamento di array

Comunque con questo algoritmo può capitare di fare uno scambio di troppo.

CASO 1

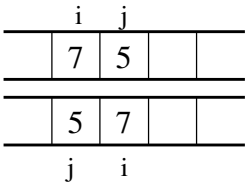


CASO 2

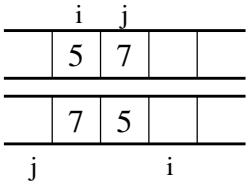


Partizionamento di array

CASO 3



CASO 4



Partizionamento di array

Un modo per prevenire il partizionamento errato sarebbe quello di effettuare uno scambio solo quando i<j: *if i<j then "scambia a[i] e a[j]"*

Tuttavia questo ulteriore controllo è stato introdotto per risolvere giusto due casi particolari. Esiste un modo per aggirare il problema, visto che c'è già la condizione

while i<j do

nel ciclo più esterno? Basterebbe spostare tale condizione immediatamente prima dello scambio.

Partizionamento di array

```
while i<j 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 a[j]>x do j:=j-1
end
```


Partizionamento di array

Tuttavia così facendo c'è sempre uno scambio automatico del primo con l'ultimo elemento. Il modo più chiaro per risolvere il problema è quello di inizializzare i valori di i e j prima di entrare nel ciclo in modo tale che a[i] ed a[j] vadano scambiati.

Per raggiungere questo obiettivo facciamo precedere il ciclo principale dai seguenti controlli:

```
while (a[i]<=x) and (i<j) do i:=i+1;
while (a[j]>x) and (i<j) do j:=j-1;
```

Dal secondo ciclo si può uscire per due ragioni (non mutuamente esclusive): i=j oppure a[j]≤x. Il primo caso si ha quando l'array è già partizionato.

Partizionamento di array

1	2	3	4	5	6	7	8	9	10
12	6	10	11	16	28	24	29	26	25

i=j

In tal caso occorre decrementare j di una unità, affinché la viariabile punti all'ultimo elemento della partizione sinistra:

if a[j]>x then j:=j-1

In questo modo non solo ci assicuriamo una corretta inizializzazione dei valori di i e j prima di entrare nel ciclo ma gestiamo anche la situazione in cui x sia fuori dell'intervallo dei valori assunti dagli elementi dell'array.

Partizionamento di array

Infatti se x è maggiore o uguale a tutti i valori dell'array questi due cicli termineranno con i=j=n e si avrà che

$\forall k \in [1..j]\ \ a[k] \leq x$

Al contrario se x è minore di tutti gli elementi dell'array i due cicli termineranno con i=j=1 e grazie al controllo successivo si avrà che j=0.

(Ricordiamo che j rappresenta il limite inferiore per i valori nella partizione che risultano esser maggiori di x)

Partizionamento di array

DESCRIZIONE DELL'ALGORITMO

1. INPUT: l'array a[1..n] e il valore di partizionamento x
2. Muovere le due partizioni da sinistra e da destra finché non si incontra una coppia di elementi nella posizione errata. Prevedere speciali casi di x fuori dei valori degli elementi dell'array
3. Mentre le due partizioni non si sono incontrate o incrociate esegui:
   (a) Scambia le coppie errate ed estendi le due partizioni di un elemento
   (b) Estendi la partizione di sinistra per elementi ≤x
   (c) Estendi la partizione di destra per elementi >x

4. OUTPUT: l'indice di partizione j e l'array partizionato.

Partizionamento di array

```
procedure xpartition(var a:nelements; n: integer; var p: integer; x: real );
var i {limite superiore corrente per valori nella partizione ≤x},
    j {limite inferiore corrente per valori >x}: integer;
    t {variabile temporanea x gli scambi}: real;
begin
    i:=1; j:=n;
    while (i<j) and (a[i] <= x) do i:=i+1;
    while (i<j) and (x < a[j]) do j:=j -1;
    if a[j]>x then j:=j-1
    while i < j 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;
    p:=j
end
```

Partizionamento di array

DIMOSTRAZIONE DI CORRETTEZZA PARZIALE:

Si dimostra che la procedura "xpartition" è corretta rispetto alle seguenti specifiche:

P_0={n>0} P_F={a[1..p-1] ≤ x, a[p+1..n]>x}

Dalla post-condizione si evince che alla fine non sapremo se a[p]≤x oppure se a[p]>x. Ad ogni modo l'array risulterà partizionato.

Dimostrazione. Si può provare che:

{n>0} i:=1; j:=n {n>0, i=1, j=n}

e che {a[1..i-1] ≤ x, 1 ≤ i ≤ j≤n} è invariante del ciclo

while (i<j) and (a[i] ≤x) do i:=i+1;

Inoltre {a[j+1..n]>x, 1≤i≤j≤n} è invariante del ciclo

while (i<j) and (a[j]>x) do j:=j-1;

Partizionamento di array

Per cui prima di iniziare il ciclo while i<j do valgono le seguenti condizioni:

{a[1..i-1]≤x, a[j+1..n]>x, 0≤j, i≤n+1, i≤j+1} (*)

che è invariante del ciclo stesso, e:

{ a[i]>x, a[j] ≤x}

per effetto della terminazione dei cicli precedenti.

Dimostriamo che (*) è l'invariante del ciclo

while i<j do …:

1. {a[1..i-1]≤x, a[j+1..n]>x, 0≤j, i≤n+1, i≤j+1} ∧ {a[i]>x,,a[j] ≤x} ∧ {i<j}

   t:=a[j]; a[i]:=a[j]; a[j]:=t

   {a[1..i] ≤x, a[j..n]>x, 0<j, i≤n, i+1≤j}

Partizionamento di array

2. {a[1..i] ≤x, a[j..n]>x, 0<j, i≤n, i+1≤j} ⇒
   {a[1..i+1-1] ≤x, a[j-1+1..n]>x, 0 ≤ j-1, i+1≤n+1, i+1≤j-1+1}
3. {a[1..i+1-1] ≤x, a[j-1+1..n]>x, 0 ≤ j-1, i+1≤n+1, i+1≤j-1+1} i:=i+1; j:=j-1
   {a[1..i-1] ≤x, a[j+1..n]>x, 0 ≤ j, i≤n+1, i≤j+1}
4. {a[1..i-1] ≤x, a[j+1..n]>x, 0≤j, i≤n+1, i≤j+1}
   while a[i]≤x do i:=i+1;
   while a[j]>x do j:=j-1
   {a[1..i-1]≤x, a[j+1..n]>x, 0≤j, i≤n+1, i≤j+1}

Quando il ciclo principale termina si ha:

{a[1..i-1]≤x, a[j+1..n]>x, (i=j ∨ i=j+1)}

e dopo l'assegnazione p:=j si ha:

P_F={a[1..p-1] ≤x, a[p+1..n]>x}

Partizionamento di array

DIMOSTRAZIONE DI TERMINAZIONE

Se i<j allora il primo ciclo while termina dopo j-i passi perché j è limite superiore per i ed i viene incrementata ad ogni passo.

Analogamente, se i<j allora il secondo ciclo termina dopo j-i passi perché i è un limite inferiore per j e j viene decrementata ad ogni passo.

Il ciclo while a[i]≤x do i:=i+1 termina perché nella dimostrazione di correttezza parziale abbiamo dimostrato che vale la seguente condizione:

{a[1..i-1]≤x, a[j+1..n]>x, 0≤j, i≤n+1, i≤j+1}

Partizionamento di array

Con un minimo sforzo è anche dimostrabile che: $i-1 \geq 1$ e $j+1 \leq n$.

Ma questo ci assicura che $\exists k \geq i$ $a[k]>x$ e poiché i è incrementata si ha che dopo un numero finito di passi $(k-i)$ il ciclo termina.

Analoghe considerazioni per il ciclo successivo.

Infine il ciclo principale termina perché i cicli interni terminano e inoltre la distanza $i-j$, limitata inferiormente da 0, diminuisce di almeno 2 unità ad ogni passo per via dell'esecuzione di: $i:=i+1$; $j:=j-1$.

Partizionamento di array

ANALISI DELLA COMPLESSITÀ

Prendendo il confronto come operazione dominante si osserva che la complessità è indipendente dall'ordinamento dei dati. Occorrerà effettuare un confronto del tipo $a[i] \leq x$ o $a[j]>x$ fino a quando gli indici i e j non si intrecciano.

Quindi devono essere eseguiti (n+2) confronti e si ha:

$f(n)=n+2$

dove f(n) è rappresenta il numero di confronti (e quindi la complesità rispetto all'operazione dominante) effettuati per partizionare l'array.

Partizionamento di array

Se si valuta invece il numero di scambi allora la complessità dipende dalla distribuzione dei dati, perché mentre per un array già partizionato non sarà necessario effettuare alcuno scambio, per un array ordinato in modo decrescente e tale che x occupa la posizione centrale sono necessari

$$\left\lfloor \frac{n}{2} \right\rfloor$$ = parte intera di n/2

scambi.

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 dispongono gli elementi di un insieme secondo una relazione d'ordine predefinita dipendente dal tipo di informazione.

INFORMAZIONE NUMERICA	Ordinamento secondo la relazione d'ordine $\leq$ su R
INFORMAZIONE ALFANUMERICA	Ordinamento lessicografico

Algoritmi di ordinamento

L'ordinamento può essere crescente o decrescente.

7, 11, 13, 16, 16, 19, 23 ordinamento crescente

tra, su, per, in, fra, con ordinam. lessicografico decr.

Il problema dell'ordinamento ha una complessità $nlog_2n$ il che vuol dire che qualunque algoritmo di ordinamento ha una complessità non inferiore a $nlog_2n$, dove n è il numero di elementi da ordinare.

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 hanno la minore complessità teoricamente ammissibile (proporzionale a $nlog_2n$).

Algoritmi di ordinamento

Gli algoritmi migliori sono quelli in grado di scambiare, nelle prime fasi di ordinamento, valori che occupano posizioni molto distanti nella lista.

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 (select sort)

Dato un insieme di n interi, ordinarlo in ordine crescente.

Delle possibili specifiche per questo problema possono essere:

$P_0=\{n>0, \forall i\in[1,n]\ a[i]\in Z\}$

$P_F=\{\forall i\in[1,n]\ \forall j\in[i+1,n]\ \ a[i]\leq a[j]\}$

Osserviamo che, per definizione di minimo elemento in un insieme, P_F può essere letta così:

$\forall i\in[1,n]\ \ a[i]=\min_{i\leq j\leq n}\{a[j]\}$

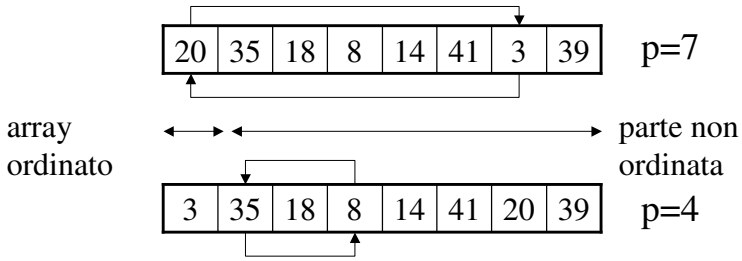
Ordinamento per selezione (select sort)

Questo ci suggerisce che un modo per ordinare l'array sarebbe quello di cercare il minimo elemento e piazzarlo nella prima posizione. Questa operazione potrebbe poi essere ripetuta fino ad avere l'array tutto ordinato.

C'è un problema:

dobbiamo ricordare la posizione del minimo per poi fare lo scambio.

Ordinamento per selezione (select sort)



Dunque, ad ogni passo, sino all'esaurimento dell'array si dovrà:

a) Trovare il minimo elemento a[p] e la posizione p della parte di array non ancora ordinata

Ordinamento per selezione (select sort)

b) scambiare l'elemento in a[p] con quello che si trova nella prima posizione della parte di array non ordinata

Tale meccanismo sarà compreso in un ciclo che ci garantisce l'esplorazione e l'ordinamento dell'intero array.

Ordinamento per selezione (select sort)

DESCRIZIONE DELL'ALGORITMO

Input: array a[1..n] di n elementi

- Mentre ci sono ancora elementi nella parte di a non ordinata esegui:

  a) trova il minimo min e la sua posizione p

  b) scambia il minimo min con il primo elemento della parte di a non ordinata

Output: array a[1..n] ordinato

IMPLEMENTAZIONE IN PASCAL

Ordinamento per selezione (select sort)

```
procedure selectsort(var a: nelement; n: integer);
var i {primo elem. nella parte di array non ordinata},
    j {indice x la parte non ordinata}, p {posiz. del min},
    min {minimo corrente}: 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
```

Ordinamento per selezione (select sort)

ANALISI DELLA COMPLESSITÀ

Considerando il confronto a[j]<min come operazione dominante possiamo subito affermare che la complessità **non** dipende dalla distribuzione dei dati. Più precisamente, il numero di confronti è dato da:

f(n)=n(n-1)/2

Infatti, per i=1 si fanno n-1 confronti,
“ i=2 “ n-2 “ ,
“ i=n-1 “ 1 “
totale n(n-1)/2 = $\sum_{i=1}^{n-1} i$

Ordinamento per selezione (select sort)

Anche il numero degli scambi non dipende dalla distribuzione dei dati in ingresso. Infatti il numero di scambi effettuati è sempre n-1.

ANALISI DELLA TERMINAZIONE

Il programma Pascal termina per via dei cicli ‘for’.

Ordinamento per selezione (select sort)

ESERCIZI

1) Ordinare un insieme di interi in ordine decrescente
2) Implementare un ordinamento per selezione che rimuove i duplicati durante il processo di ordinam.
3) Il numero di confronti richiesti dall'ordinamento per selezione può essere ridotto considerando gli elementi a coppie e trovando contemporaneamente sia il minimo che il massimo. Implementare un algoritmo che incorpori questa idea e determinare il numero di confronti effettuati

Ordinamento per selezione (select sort)

4) Implementare una funzione che esamina un array e stabilisce se questo è ordinato oppure no.
5) Modificare l'algoritmo di ordinamento per selezione cosicché esso ordini tutti i valori minori di un certo x. Variando x casualmente, confrontare questo algoritmo con quello di partizionamento.

Ordinamento per scambi (bubble sort)

Dato un insieme di n interi, ordinarlo in ordine crescente.

Delle possibili specifiche per un algoritmo che risolve il problema possono essere

$P_0=\{n>0, \forall i \in [1,n]\ \ a[i] \in Z\}$

$P_F=\{\forall i \in [1,n-1]\ \ a[i] \leq a[i+1]\}$

P_F ci suggerisce che un modo per ordinare il vettore è quello di considerare gli elementi consecutivi a due a due, facendo in modo da ordinare la coppia.

Ordinamento per scambi (bubble sort)

Esempio

Inizialmente	30	12	18	8	14	41	3	39
2° passo	12	30	18	8	14	41	3	39
3° passo	12	18	30	8	14	41	3	39
…								
Infine	12	18	8	14	30	3	39	41

Come si vede, alla fine otteniamo che l'elemento più grande sia in coda. Questo ultimo è, in effetti, la parte di vettore ordinata.

Ordinamento per scambi (bubble sort)

Il meccanismo:

per tutte le coppie adiacenti del vettore esegui

(a) se la coppia non è in ordine crescente scambia gli elementi

ripetuto nuovamente porterà come effetto che il secondo elemento più grande (39) andrà in penultima posizione. Dunque il meccanismo ripetuto n-1 volte ci darà il vettore ordinato a partire da destra.

Tale algoritmo è particolarmente buono se il vettore è già ordinato, perché si effettuano meno scambi.

Ordinamento per scambi (bubble sort)

Osservazione: quando non si effettuano scambi siamo sicuri che il vettore è ordinato.

Allora si può formulare la seguente

DESCRIZIONE DELL'ALGORITMO

1. Mentre il vettore non è ordinato esegui
   a) metti a true l'indicatore SORTED
   b) per tutte le coppie adiacenti di elementi nella parte non ordinata esegui

Ordinamento per scambi (bubble sort)

b.1) se la coppia corrente non è in ordine allora

1.a) scambia gli elementi

1.b) metti SORTED a false

Output: array a[1..n] ordinato

IMPLEMENTAZIONE IN PASCAL

Ordinamento per scambi (bubble sort)

```
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
```

Ordinamento per scambi (bubble sort)

ANALISI DELLA CORRETTEZZA

Il ciclo for più interno è pensato per scambiare quegli elementi consecutivi che non sono ordinati. Pertanto l'invariante del ciclo sarà:

$P=\{\forall k \in [1, j-1]\ \ a[k] \leq a[j]\}$

Dimostrazione.

P vale prima che si esegua il ciclo per la prima volta, non appena j è inizializzata a 1.

Sia $B=\{j \leq n-i\}$ la condizione verificata durante il ciclo for. Allora vogliamo provare che:

Ordinamento per scambi (bubble sort)

$$\{P \wedge B\} \quad S \quad \{P\}$$

dove S è il blocco di istruzioni eseguite nel ciclo for (compreso l'incremento di j).

Infatti

1. $P \wedge B \wedge \{a[j]>a[j+1]\}$
   begin t:=a[i]; …; end
   $B \wedge \{\forall k \in [1, j]\ \ a[k] \leq a[j+1]\}$
2. $P \wedge B$ if a[j]>a[j+1] … end
   $B \wedge \{\forall k \in [1, j]\ \ a[k] \leq a[j+1]\}$

Ordinamento per scambi (bubble sort)

3. $B \wedge \{\forall k \in [1, j]\ a[k] \leq a[j+1]\}$ $j:=j+1$ P
4. $\{P \wedge B\}$ S $\{P\}$ *(per composizione)*

c.v.d.

Abbiamo così dimostrato che P è l'invariante del ciclo for.

Si esce dal ciclo quando $j=n-i+1$, per cui al termine del ciclo for deve accadere che:

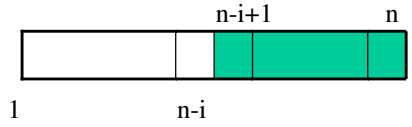
$\forall k \in [1, n-i]\ a[k] \leq a[n-i+1]$

Ordinamento per scambi (bubble sort)

Abbiamo così provato che il ciclo più interno sposta nella posizione n-i+1 il massimo elemento presente nelle posizioni da 1 a n-i+1.

Resta ora da dimostrare l'invariante del ciclo while.

Ricordiamo che allo i-esimo passo gli ultimi i elementi del vettore sono ordinati



quindi $\{\forall k \in [n-i+1, n-1]\ a[k] \leq a[k+1]\}$

Ordinamento per scambi (bubble sort)

Cosa possiamo dire per la parte non ordinata?

Se è vero not sorted allora possiamo solo concludere che {∀k∈[1, n-i] a[k]≤a[n-i+1]}

mentre se è falso allora anche la parte sinistra è ordinata

{∀k∈[1, n-i-1] a[k]≤a[k+1]}

e inoltre

{∀k∈[1, n-i], ∀k'∈[n-i+1, n] a[k]≤a[k']}

ovvero, tutti gli elementi della parte sinistra sono minori degli elementi della parte destra.

Quindi l'invariante del ciclo è data da:

Ordinamento per scambi (bubble sort)

{∀k∈[n-i+1, n-1], a[k]≤a[k+1]} ∧

({not sorted ∧ ∀k∈[1, n-i], a[k]≤a[n-i+1]} ∨

{sorted ∧ ∀k∈[1, n-i-1] a[k]≤a[k+1] ∧ ∀k∈[1, n-i], ∀k'∈[n-i+1, n] a[k]≤a[k']})

La dimostrazione mediante le regole (R1-R5) che questa è effettivamente l'invariante è lasciata per esercizio.

Il ciclo while termina quando i=n oppure sorted=true.

Nel primo caso, la prima condizione dell'invariante diventa:

Ordinamento per scambi (bubble sort)

$\{\forall k \in [1, n-1], a[k] \leq a[k+1]\} = P_F$

che è quanto volevamo dimostrare.

Se invece si termina per sorted=true, allora dev'essere soddisfatta la seconda condizione in OR dell'invariante del while, e quindi si ha:

$\forall k \in [1, n-i-1]\ a[k] \leq a[k+1] \wedge$

$\forall k \in [n-i+1, n-1]\ a[k] \leq a[k+1] \wedge$

$\forall k \in [1, n-i], \forall k' \in [n-i+1, n]\ a[k] \leq a[k']\}$

che è equivalente a dire che il vettore è ordinato.

Ordinamento per scambi (bubble sort)

ANALISI DI TERMINAZIONE

Il ciclo for termina per definizione.

Nel ciclo while ci sono due condizioni in AND.

La seconda potrebbe diventare falsa solo se non ci troviamo nel caso di ordinamento decrescente del vettore dato.

In ogni caso, sicuramente la prima condizione (i<n) diventerà falsa dopo n passi perché la variabile i è incrementata ad ogni passo.

Ordinamento per scambi (bubble sort)

ANALISI DELLA COMPLESSITÀ

Considerando il confronto a[j]>a[j+1] come operazione dominante possiamo subito osservare che la complessità dipende dalla distribuzione dei dati.

Più precisamente, nel caso in cui il vettore sia ordinato in modo crescente il numero di confronti è dato da n-1 perché non si effettuerà nessuno scambio e non si forzerà sorted a false.

Al contrario, quando il vettore è ordinato in senso decrescente si effettua il massimo numero di confronti, dato da n(n-1)/2 .

Ordinamento per scambi (bubble sort)

Infatti, per i=1 si fanno n-1 confronti

" i=2 " n-2 "

…

" i=n-1 " 1 "

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

Quindi:

$f_{ott}(n) = n-1$ $f_{pess}(n) = n(n-1)/2$

Ordinamento per scambi (bubble sort)

Inoltre si dimostra che

$$f_{med}(n) = n(n-1)/4$$

Si osservi che il numero di scambi può essere al più pari al numero di confronti. Quindi nel caso migliore (vettore ordinato in senso crescente) non si effettuano scambi, mentre nel caso peggiore (vettore ordinato in senso decrescente) si effettuano n(n-1)/2 confronti.

NOTA: Mentre gli elementi maggiori sono spostati rapidamente all'estrema destra, gli elementi minori sono spostati lentamente all'estrema sinistra. Questo

Ordinamento per scambi (bubble sort)

comportamento asimmetrico può essere modificato alterando le direzioni in cui si effettuano gli spostamenti nelle varie iterate.

ESERCIZIO

Realizzare un algoritmo che incorpori questa variante.

Ordinamento per inserzione (insertion sort)

Dato un vettore di n interi, ordinarlo in ordine crescente.

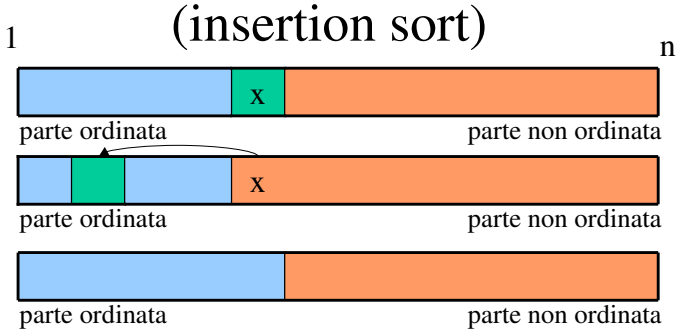
Delle possibili specifiche per un algoritmo che risolve il problema possono essere:

$P_0=\{n>0, \forall i\in[1, n]\ a[i]\in Z\}$

$P_F=\{\forall i\in[2, n]\ \forall j\in[1, i-1]\ a[j]\leq a[i]\}$

ovvero un vettore è ordinato se a[i] è maggiore di ogni elemento che lo precede. Quindi P_F ci suggerisce che la soluzione al problema può essere costruita inserendo un elemento della parte non ordinata del vettore nella parte parzialmente ordinata, estendendola di un elemento.

Ordinamento per inserzione (insertion sort)



Come elemento da scegliere basterà prendere il primo della parte non ordinata.

Inizialmente il vettore sarà disordinato e come "parte ordinata" prenderemo a[1].

Ordinamento per inserzione (insertion sort)

Poi, all'interno di un ciclo che ci assicura lo scorrimento dell'intera struttura, sceglieremo il prossimo elemento per l'inserzione, lo inseriremo nella posizione giusta della parte ordinata del vettore. Per far spazio, ogni volta, ad x dovremo far scorrere di un posto gli elementi maggiori di x.

Partendo con j:=i il ciclo sarà:

```
while x<a[j-1] do
        begin
                a[j]:=a[j-1]; j:=j-1
        end
```


Ordinamento per inserzione (insertion sort)

C'è un problema di terminazione. Se risulta

$\forall j \in [1, i-1]$ a[j]<x

si finirà per referenziare l'elemento a[0], che non è definito!

Per evitare il problema si potrebbe aggiungere un controllo sul valore di j nella condizione del while.

```
while (x<a[j-1]) and j>1 do …
```

In questo modo aumentiamo il numero di confronti effettuati in quanto abbiamo un test più costoso.

Ordinamento per inserzione (insertion sort)

Una soluzione più efficiente consiste nel trovare il minimo elemento e porlo in a[1] prima ancora di entrare nel ciclo while. In questo modo ci assicuriamo che la condizione

$$\forall j \in [1, i-1] \quad a[j] < x$$

non si possa mai verificare.

Ovviamente se a[1] contiene il minimo possiamo considerare ordinata la parte del vettore costituita dai primi due elementi, poiché $a[1] \leq a[2]$. Ciò vuol dire che il primo x da sistemare è il terzo elemento del vettore.

Ordinamento per inserzione (insertion sort)

DESCRIZIONE DELL'ALGORITMO

Input: vettore a[1..n] di n interi

1. Trovare il minimo e piazzarlo al primo posto.
2. Mentre ci sono ancora elementi da inserire, esegui:
   a) seleziona il prossimo x da inserire
   b) mentre x è minore dell'elemento precedente
      b.1) sposta di una posizione l'elemento precedente
      b.2) estendi la ricerca di un elemento indietro
   c) inserisci x nella giusta posizione

Output: vettore a[1..n] ordinato.

Ordinamento per inserzione (insertion sort)

IMPLEMENTAZIONE IN PASCAL

```
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;
```


Ordinamento per inserzione (insertion sort)

```
        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
```

Ordinamento per inserzione (insertion sort)

ANALISI DELLA CORRETTEZZA

Il primo ciclo for è pensato per individuare il minimo elemento del vettore. Pertanto l'invariante del ciclo sarà:

P_1={∀k∈[1, i-1] first≤a[k], first=a[p], 1≤p≤i-1}

La dimostrazione è immediata. Infatti P_1 vale prima che si entri nel ciclo, non appena i è inizializzata a 2.

Sia B_1={i ≤ n} la condizione verificata durante il ciclo for. Allora vogliamo provare che: $\{\mathbf{P_1 \wedge B_1}\}$ **S** $\{\mathbf{P_1}\}$

dove S è il blocco di istruzioni eseguite nel ciclo for compreso l'incremento di i.

Ordinamento per inserzione (insertion sort)

Infatti,

1. $P_1 \wedge B_1 \wedge$ {a[i]<first} begin first:=a[i];…; end
   $B_1 \wedge$ {∀k∈[1, i] first≤a[k], first=a[p], 1≤p≤i}
2. $P_1 \wedge B_1$ if a[i]<first … end
   $B_1 \wedge$ {∀k∈[1, i] first≤a[k], first=a[p], 1≤p≤i}
3. $B_1 \wedge$ {∀k∈[1, j] first≤a[k], first=a[p], 1≤p≤i} i:=i+1 P_1
4. $P_1 \wedge B_1$ S P_1 (per composizione) c.v.d.

Così ci siamo assicurati che prima di eseguire il secondo ciclo for si ha:

∀k∈[1, n] a[1]≤a[k]

Ordinamento per inserzione (insertion sort)

Prima di analizzare l'invariante del secondo ciclo for conviene studiare il ciclo while interno.

Tale ciclo serve per individuare la posizione in cui x va inserito, effettuando i dovuti spostamenti degli a[j]. Pertanto una invariante significativa potrebbe essere:

$P_2=\{\forall k\in[j, i-1]\ a[1]\le x<a[k+1]=a_k,\ 2 \le j \le i,\ x=a_i\}$

dove gli a_k sono i valori presenti negli elementi a[k] prima di iniziare il ciclo.

Infatti, quando j=i la P_2 è vera. Inoltre, detta

$B_2=\{x<a[j-1]\}$

la condizione controllata nel ciclo while, si dimostra facilmente che

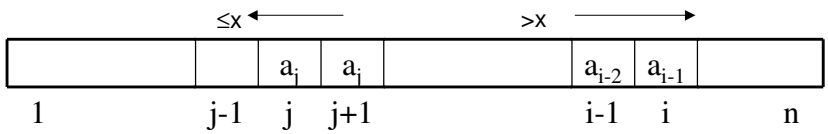
Ordinamento per inserzione (insertion sort)

$P_2 \wedge B_2$ begin a[j]:=a[j-1]; j:=j-1 end P_2

(si osservi che se vale B_2 allora dev'essere j>2)

Al termine del ciclo si avrà:

$\{\forall k\in[j, i-1]\ a[1]\le x<a[k+1]=a_k,\ 2 \le j \le i,\ a[j-1] \le x,\ x=a_i\}$



quindi ponendo x in a[j] si ottiene

$\{\forall k\in[j, i-1]\ a[1] \le a[j-1] \le a_i < a[k+1]=a_k,\ 2 \le j \le i\}$

Ordinamento per inserzione (insertion sort)

Congiungendo questa conclusione con l'ipotesi che

$a_1 \leq a_2 \leq \ldots \leq a_{i-1}$

valida per ipotesi si ha che i primi i elementi del vettore risultano ordinati.

Infine si dimostra che l'invariante del secondo ciclo for è proprio la seguente:

$\{\forall k \in [2, i-1]\ \forall j \in [1, k-1]\ a[j] \leq a[k]\}$

(dim. per esercizio). Da qui segue la specifica P_F quando i=n+1, cioè il ciclo for termina.

Ordinamento per inserzione (insertion sort)

DIMOSTRAZIONE DELLA TERMINAZIONE

I due cicli for terminano sicuramente.

Dimostriamo che il ciclo while termina dopo i-2 passi. È sufficiente osservare che prima dell'inizio del ciclo 0<j=i e che ad ogni esecuzione j è decrementato. Pertanto, dopo aver eseguito il ciclo per i-2 volte sarà j=2 e il test x<a[j-1]=a[1] deve necessariamente fallire per via della condizione invariante enunciata nella dimostrazione di correttezza parziale.

Ordinamento per inserzione (insertion sort)

ANALISI DELLA COMPLESSITÀ

Se guardiamo ai confronti a[i]<first e x<a[j-1] come operazioni dominanti, allora dobbiamo concludere subito che il numero di confronti effettuati dipende dall'ordinamento dei dati.

Il caso migliore è costituito da un vettore già ordinato in modo crescente.

In tal caso il controllo x<a[j-1] fallirà subito e il ciclo while non sarà mai eseguito. Quindi nel caso ottimo si ha:

$$f_{ott}(n) = \underbrace{(n-1)}_{\text{ricerca del minimo da porre in a[1]}} + (n-2) = 2n-3$$

Ordinamento per inserzione (insertion sort)

Al contrario, quando il vettore a[2..n] è ordinato in senso inverso e a[1] contiene il minimo, occorrerà effettuare il maggior numero di confronti.

Più precisamente

n-1 confronti per porre il minimo in a[1]
2 confronti per porre a[3] in 2ª posizione
3 " a[4] in 3ª "
...
i-1 " a[i] in iª "
...
n-1 " a[n] in nª "

Ordinamento per inserzione (insertion sort)

In totale,

$$f_{pess}(n) = n - 1 + \sum_{i=2}^{n-1} i \quad = n - 1 + \sum_{i=1}^{n-1} i \; - 1 =$$

$$= n - 1 + \tfrac{1}{2}(n - 1)n \; - 1 =$$

$$= \frac{2n - 2 + n^2 - n - 2}{2} = \frac{n^2 - n - 4}{2}$$

Pertanto, sia nel caso pessimo, il numero di confronti da effettuare è proporzionale a n^2. Anche se non lo dimostriamo, questo vale anche nel caso medio.

Ordinamento per inserzione (insertion sort)

ESERCIZIO

Valutare empiricamente la complessità media dell'ordinamento per inserzione nel caso di dati generati mediante un generatore di numeri pseudocasuali.

Analoghe conclusioni possono essere tratte nel caso in cui si consideri lo spostamento come operazione dominante.

L'ordinamento per inserzione è di solito considerato il migliore degli algoritmi con complessità quadratica per ordinare piccoli insiemi con scarso preordinamento.

Ordinamento a diminuzione di incremento (Shell's sort)

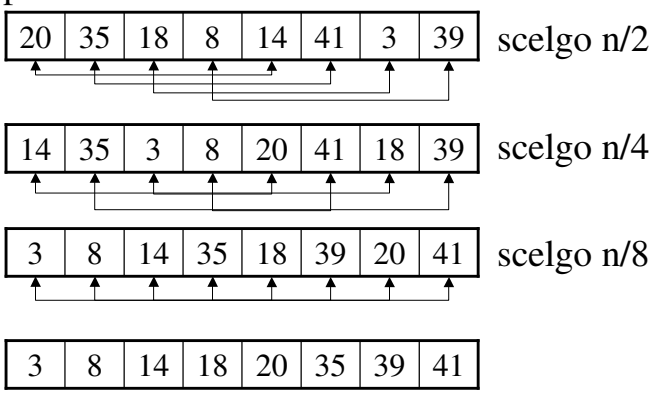
Questo algoritmo è basato sull'osservazione che si può trarre vantaggio dai confronti iniziali piazzando gli elementi il più possibile vicino alla loro posizione finale.

Un metodo consiste nel muovere, inizialmente, elementi su lunghe distanze e poi diminuire le distanze.

Ordinamento a diminuzione di incremento (Shell's sort)

Esempio

n=8



Ordinamento a diminuzione di incremento (Shell's sort)

Il disordine nell'array è piccolo verso la fine dell'ordinamento (per es. quando la distanza è n/8) quindi conviene scegliere un algoritmo efficiente per dati parzialmente ordinati al fine di ordinare ogni singola catena.

L'ordinamento per inserzione sembra il migliore. Ma come applicarlo alle varie catene?

Ordinamento a diminuzione di incremento (Shell's sort)

```
inc:=n;
while inc>1 do
    begin
        (a) inc:=inc div 2;
        (b) applica il sort per inserzione alle catene con incremento inc
    end
```

Altro problema è quello di definire quante catene vanno ordinate per ogni incremento e come accedere ad ogni singola catena per applicare l'ordinamento per inserzione.

Il numero di catene è sempre uguale all'incremento.

Ordinamento a diminuzione di incremento (Shell's sort)

```
inc:=n;
while inc>1 do
    begin
        (a) inc:=inc div 2;
        (b) for j:=1 to inc do
                begin
                    "applica l'ordinamento per inserzione alla
                     catena corrente con incremento inc"
                end
    end
```

Nell'ordinamento per inserzione il primo elemento da considerare per l'inserzione è il secondo elemento nell'array. Sarà dunque il secondo elemento per ogni

Ordinamento a diminuzione di incremento (Shell's sort)

catena il cui indice k è dato da:

k:=j+inc

Gli elementi successivi in ogni catena che inizia con generico j saranno trovati così:

k:=k+inc

La catena termina quando k supera n, il numero di elementi nell'array.

Con questi raffinamenti arriviamo a

Ordinamento a diminuzione di incremento (Shell's sort)

```
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
                             x:=a[k]
                             "trova la posizione corrente per x"
                             a[corrente]:=x; k:=k + inc
                             end
                  end
end
```


Ordinamento a diminuzione di incremento (Shell's sort)

Come meccanismo per garantire la terminazione avevamo, nel caso dell'ordinamento per inserzione, la sentinella (minimo in prima posizione)

Questo approccio non è più adottabile poiché non esiste più un unico elemento, a[1], dove terminano tutte le catene.

Per trovare la posizione giusta di x nella catena si dovrà partire con:

corrente := k

e confrontare x con l'elemento precedente nella catena, con posizione

precedente := corrente - inc

Ordinamento a diminuzione di incremento (Shell's sort)

In seguito gli altri elementi nella catena potranno essere trovati usando:

precedente := precedente - inc

La condizione per fare l'inserzione diventa:

while x<a[precedente] do

Come garantirci la terminazione?

Ordinamento a diminuzione di incremento (Shell's sort)

Se precedente è ripetutamente decrementato di inc allora diventerà minore di j che segna l'inizio della catena corrente.

Dunque basterà fare l'inserzione garantendosi che:

precedente >= j

Saremmo tentati di scrivere:

while (precedente >= j) and (x<a[precedente]) do

ma ci sono problemi con l'implementazione Pascal.

Infatti, anche se precedente >= j non è vero, il test x<a[precedente] verrà effettuato ugualmente, accedendo così a elementi indefiniti per l'array.

Ordinamento a diminuzione di incremento (Shell's sort)

Modifichiamo alora la condizione di fine ciclo così:

while (precedente >= j) and not inserito do

dove inserito è un indicatore di tipo boolean utilizzato per indicare se x<a[precedente] è vero o falso.

Infine l'inserzione vera e propria è realizzata come nell'implementazione dell'ordinamento per inserzione.

DESCRIZIONE DELL'ALGORITMO

Input: vettore a[1..n] di n elementi

1. Inizializzare l'incremento inc a n
2. Mentre l'incremento si mantiene maggiore di 1 esegui
   a) dimezza l'incremento
   b) per tutte le catene (in numero = inc) da ordinare a intervalli di inc esegui
      b.1) determina la posizione k del secondo membro della catena corrente
      b.2) mentre non è ancora raggiunta la fine della catena corrente esegui
         2.a) usa il meccanismo di inserzione per porre x=a[k] al posto giusto
         2.b) passa a considerare l'elemento successivo della catena corrente incrementando k di inc.

Output: vettore a[1..n] ordinato.

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
```

Ordinamento a diminuzione di incremento

ANALISI DELLA CORRETTEZZA

La condizione invariante per il ciclo while più esterno è che, dopo ogni iterazione, tutte le catene i cui elementi sono saparati da inc posti sono state ordinate per inserzione.

Dopo il j-esimo passo del successivo ciclo for, le prime j catene (con incremento inc) sono state ordinate per inserzione.

Per il successivo ciclo while più interno, dopo l'iterazione che coivolge k, l'elemento in posizione k è stato appropriatamente ordinato nella sua catena.

Ordinamento a diminuzione di incremento

Con il ciclo while più interno, dopo l'iterazione che riguarda l'indice precedente, si stabilisce o che l'elemento corrente può essere inserito in posizione corrente o che tutti gli elementi nella catena corrente, incluso a[precedente], sono maggiori di x.

Tutti questi elementi sono stati spostati di un posto in previsione di un'eventuale inserzione di x nella catena.

Ordinamento a diminuzione di incremento

ANALISI DELLA TERMINAZIONE

Il ciclo esterno termina perché inc è ridotto di un fattore 2 ad ogni passo.

Il for termina per definizione. Il while che gestisce la catena corrente termina perché k cresce di inc ad ogni passo.

Il while più interno termina perché precedente è ridotto di inc ad ogni passo.

Ordinamento a diminuzione di incremento

ANALISI DELLA COMPLESSITÀ

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 $f_{pess}(n)$ è proporzionale a 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 dell'ordinamento di Shell 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 Shell's sort non fa più di $n^{1.5}$ confronti, cioè $f_{pess}(n)$ è proporzionale a $n^{1.5}$.

Divide et Impera

Una tecnica di risoluzione dei problemi consiste nella scomposizione di un problema in sottoproblemi, alcuni dei quali dello stesso tipo del problema di partenza ma più semplici, e nella successiva combinazione delle soluzioni dei sottoproblemi in modo da ottenere la soluzione del problema di partenza.

Questo approccio, spesso combinato alla programmazione ricorsiva, consente di trovare algoritmi efficienti.

Lo scheletro di una procedura ricorsiva DIVIDE-ET-IMPERA per risolvere un problema P di dimensione n (la dimensione è una misura del grado di complessità del problema) è il seguente:

Prof. D. Malerba

Divide et Impera

```
procedure DIVIDE-ET-IMPERA(P,n);
begin
        if n ≤k then
                {risolvi P direttamente}
        else
        begin
                {dividi P in h sottoproblemi P1, …, Ph di
                 dimensione n1, …, nh}
                for i := 1 to h do DIVIDE-ET-IMPERA(Pi, ni)
                {combina i risultati di P1, …, Ph in modo da
                 ottenere quello di P}
        end
end
```

Divide et Impera

Il calcolo della complessità di algoritmi progettati secondo la tecnica "divide et impera" è effettuato mediante la seguente funzione ricorsiva:

$f(n) = \text{costante}$ per $n \leq k$

$f(n) = d(n) + c(n) + \Sigma^{h}_{i=1} f(n_i)$ per $n > k$

dove d(n) e c(n) denotano la complessità rispettivamente della divisione del problema e della composizione dei risultati parziali.

Se i dati sono partizionati in maniera bilanciata, cioè se tutti gli n_i sono all'incirca uguali, e se il costo di divisione del problema e composizione dei risultati parziali non è elevato, allora l'algoritmo può risultare molto efficiente.

Divide et impera

Esempi di algoritmi ottenuti mediante la tecnica "divide et impera" sono:

- ordinamento per partizionamento (o quicksort)
- ordinamento per fusioni successive (o mergesort)
- ricerca binaria.

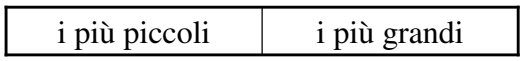
Ordinamento per partizionamento (quicksort)

Dato un vettore di interi, ordinarlo in ordine non decrescente.

Anche questo algoritmo è basato, come quello di Shell, sul concetto di distanza su cui muovere i dati, ed è sviluppato, come il *mergesort*, usando la tecnica di programmazione *dividi et impera*.

L'idea principale è quella di spostare subito, sin dal primo passo, tutti gli elementi più piccoli nelle prime posizioni e tutti gli elementi più grandi in coda al vettore.

Ordinamento per partizionamento (quicksort)



Questo processo è detto **partizionamento**.

Bisognerebbe conoscere l'elemento mediano.

Esempio

20	35	18	8	14	41	3	39

Se scegliessimo 18 ottimizzeremmo la partizione. 20 è nella partizione sbagliata, il che implica che sicuramente ci sarà un altro elemento nella seconda partizione che sarà in posizione errata.

Ordinamento per partizionamento (quicksort)

Dunque, se ci muoviamo contemporaneamente da sinistra e da destra cercando gli elementi rispettivamente più grandi e più piccoli rispetto ad un elemento mediano (18) troviamo facilmente i candidati allo scambio.

L'algoritmo può riassumersi, finora, in:

Ordinamento per partizionamento (quicksort)

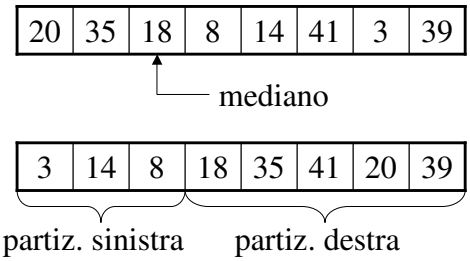
1) estendi le due partizioni in senso inverso fino a trovare elementi sbagliati
2) mentre le due partizioni non sono esaurite
   a) scambia le coppie errate
   b) estendi le due partizioni fino a trovare un'altra coppia errata

A questo punto abbiamo due partizioni che possono essere trattate indipendentemente.

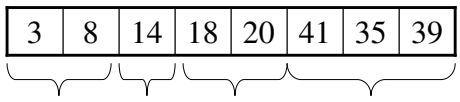
Le due partizioni possono non essere di eguale dimensione ed è possibile ordinare separatamente l'una e l'altra.

Ordinamento per partizionamento (quicksort)

Nel nostro caso:



cui possiamo riapplicare i concetti già esposti.



Ordinamento per partizionamento (quicksort)

Queste partizioni sono ancora parzialmente ordinate, anche se sicuramente vi è ordine tra le partizioni.

Il meccanismo di base è, dunque:

1. Partiziona i dati in partizioni sinistra e destra, ammesso che nell'insieme corrente ci sia più di un elemento
2. Ripeti il processo di partizionamento per la partizione di sinistra
3. Ripeti il processo di partizionamento per la partizione di destra

Ordinamento per partizionamento (quicksort)

Riguardo al processo di partizionamento, ci serviamo di quattro indicatori di indice:

Bisogna trovare un buon metodo per individuare i valori mediani.

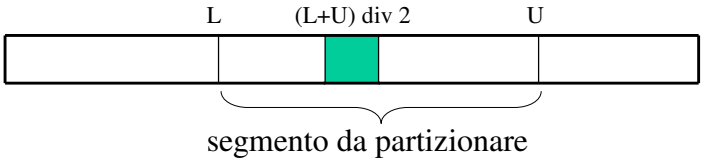
Potremmo scegliere il primo elemento, ma nel caso di vettore già ordinato otterremmo una partizione sinistra di un solo elemento mentre quella destra includerebbe tutti gli altri.

Ordinamento per partizionamento (quicksort)

Poiché, come si vedrà nell'analisi di complessità, l'algoritmo è tanto più veloce quanto più sono bilanciati i partizionamenti, possiamo concludere che nel caso di vettore già ordinato non si avrebbe nessun vantaggio nello scegliere il primo elemento come valore mediano.

Nell'ipotesi di vettore ordinato la scelta migliore sarebbe quella di prendere l'elemento centrale x del segmento da partizionare.

Ordinamento per partizionamento (quicksort)



Poiché nel caso in cui il vettore non sia ordinato non disponiamo di criteri obiettivi per scegliere (una posizione vale l'altra), preferiamo l'elemento centrale perché è la scelta ottima nell'ipotesi di ordinamento.

La procedura di partizionamento è quindi la seguente:

Ordinamento per partizionamento (quicksort)

```
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 (quicksort)

```
        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 (quicksort)

Riepilogando, il quicksort ha come primo step la chiamata alla procedura

partition(a, minsin, maxdes, maxsin, mindes)

Per ripetere il processo di partizionamento per la partizione di sinistra, se la nostra procedura è quicksort, basterà chiamare

quicksort(a, minsin, maxsin)

per la partizione di sinistra e

quicksort(a, mindes, maxdes)

per ordinare la partizione di destra.

Ordinamento per partizionamento (quicksort)

Per la condizione di terminazione, la chiamata ad un segmento di un unico elemento costituirà la terminazione della ricorsione.

Nella chiamata a quicksort sono definiti i limiti inferiore e superiore di ciascun segmento. Dunque, possiamo usare un test del tipo:

se un segmento contiene più di un elemento allora
(a) partiziona in partizione sinistra e destra
(b) quicksort sulla partizione sinistra
(c) quicksort sulla partizione destra

Ordinamento per partizionamento (quicksort)

Più precisamente il test avrà la forma

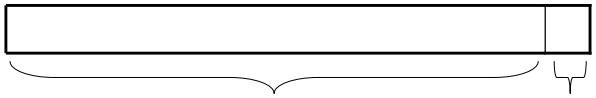
if minsin < maxdes then …

Si deve osservare che ad ogni chiamata ricorsiva si allocherà lo spazio per due nuovi parametri formali, minsin e maxsin (rispettivamente mindes e maxdes).

È pertanto indispensabile analizzare il numero massimo di livelli nella ricorsione per capire quale sia la complessità in spazio dell'algoritmo.

L'ipotesi peggiore è quella di avere un solo elemento nella partizione più piccola, ogni volta.

Ordinamento per partizionamento (quicksort)



In questo caso abbiamo n-1 livelli nella ricorsione, cioè dobbiamo memorizzare i limiti per n-1 partizioni su un vettore di n elementi, con una complessità pari a:

$$n + 2\,(n - 1)$$

array a — limiti delle partizioni

È troppo.

Ordinamento per partizionamento (quicksort)

È più conveniente processare per prima la partizione **più piccola**. In questo caso il numero massimo di livelli di ricorsione è $\log_2 n$ e l'occupazione in spazio è data da: $n + \log_2 n$

Concludendo, la chiamata ricorsiva della procedura quicksort va preceduta da un test che decida quale partizione ordinare.

Ordinamento per partizionamento (quicksort)

DESCRIZIONE DELL'ALGORITMO *QUICKSORT*

Input: vettore di elementi da ordinare e i limiti del segmento corrente da ordinare

1. Se il segmento corrente contiene più di un elemento allora

   a) partiziona il segmento corrente in due segmenti più piccoli così che tutti gli elementi nel segmento di SINISTRA siano minori o eguali a tutti gli elementi nel segmento di DESTRA

Ordinamento per partizionamento (quicksort)

b) se il segmento di SINISTRA è più piccolo del segmento di DESTRA allora

(b.1) quicksort del segmento di SINISTRA

(b.2) quicksort del segmento di DESTRA

altrimenti

(b'.1) quicksort del segmento di DESTRA

(b'.2) quicksort del segmento di SINISTRA

Ordinamento per partizionamento (quicksort)

```
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 (quicksort)

quicksort(20 35 08 18 14 41 03 39) *initial call*

([03 14 08] 18 [35 41 20 39]) *result of 1st call to partition*

- quicksort(3 14 8 18 [35 41 20 39])
  - ([03 08][14] 18 [35 41 20 39]) *result of 2nd call to partition*
  - quicksort([03 08] 14 18 [35 41 20 39]) *recursion terminates*
  - quicksort(03 08 14 18 [35 41 20 39])
    - ([03] [08] 14 18 [35 41 20 39])
    - quicksort([03] 08 14 18 [35 41 20 39]) *recursion terminates*
    - quicksort(03 08 14 18 [35 41 20 39]) *recursion terminates*
- quicksort(03 08 14 18 35 41 20 39)

Ordinamento per partizionamento (quicksort)

(03 08 14 18 [35 39 20] [41])
quicksort(03 08 14 18 [35 39 20] 41) *recursion terminates*
quicksort(03 08 14 18 35 39 20 41)
(03 08 14 18 [35 20] [39] 41)
quicksort(03 08 14 18 [35 20] 39 41) *recursion terminates*
quicksort(03 08 14 18 35 20 39 41) *recursion terminates*
(03 08 14 18 [20] [35] 39 41)
quicksort(03 08 14 18 [20] 35 39 41) *recursion terminates*
quicksort(03 08 14 18 20 35 39 41) *recursion terminates*

Ordinamento per partizionamento (quicksort)

Analisi della complessità

La complessità del quicksort dipende dalla distribuzione dei dati. Prendiamo come operazione dominante il confronto con elementi dell'array a (vedi test a[i]<x, x<a[j] effettuati nella procedura *partition*).

Il caso pessimo è quello in cui ad ogni chiamata del quicksort una delle due partizioni risulta di un solo elemento.

È un evento piuttosto raro, ma quando si verifica la complessità sarà data da

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

Si dimostra che nel caso medio la complessità è $n \log n$.

Ordinamento per partizionamento (quicksort)

Analizziamo il CASO MEDIO e consideriamo come operazione dominante il confronto con elementi dell'array a (vedi test a[i]<x, x<a[j] effettuati 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

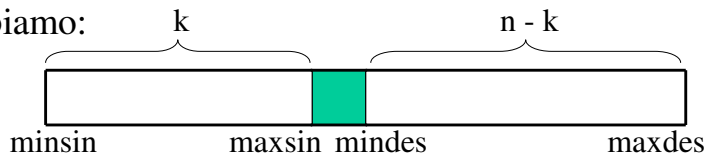
Ordinamento per partizionamento (quicksort)

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).

Supponiamo che al termine della procedura *partition* abbiamo:



Ordinamento per partizionamento (quicksort)

La probabilità che il segmento sia partizionato in partizione sinistra di dimensione k-1 e partizione destra di dimensione n-k è, sotto assunzione di distribuzione di probabilità uniforme, la stessa per ogni k con $1 \leq k \leq n$, quindi:

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

Ordinamento per partizionamento (quicksort)

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

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

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

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

Ordinamento per partizionamento (quicksort)

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

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

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

Ordinamento per partizionamento (quicksort)

quindi

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

da cui

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

Ma allora dev'essere anche vero che:

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

Ordinamento per partizionamento (quicksort)

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:

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

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

Ordinamento per partizionamento (quicksort)

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

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

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

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

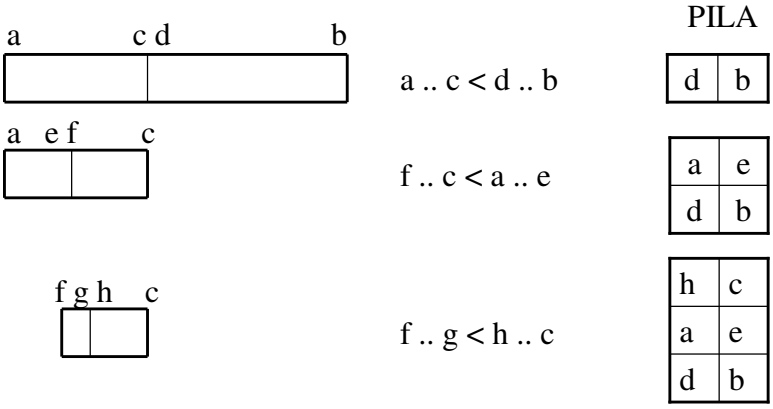
quindi $f_{med}(n) \approx 2n \ln(n)$

Versione iterativa del quicksort

Per abbattere il tempo necessario alla gestione della ricorsione, si può pensare di ricorrere all'utilizzo di una pila in uno schema di composizione iterativo.

Per la precisione salviamo nella pila solo gli estremi di una partizione del vettore da ordinare, e ordiniamo direttamente l'altra partizione.

Versione iterativa del quicksort



Versione iterativa del quicksort

Quando è stata raggiunta una partizione di dimensione 1 non abbiamo altra alternativa che ricominciare il processo di partizionamento sul segmento memorizzato più recentemente.

Per far questo basta rimuovere i limiti della partizione dalla cima (top) della pila (stack). Quando tutte le partizioni avranno raggiunto dimensione 1 non ci saranno più limiti memorizzati nella pila. Questa è una condizione di terminazione del processo.

Versione iterativa del quicksort

Input: vettore a[1..n] da ordinare

1. Crea la pila
2. Piazza i limiti 1..n nella pila
3. Mentre la pila non è vuota esegui

   (a) Leggi ed estrai dalla pila i limiti LEFT e RIGHT del segmento da partizionare

   (b) Mentre il segmento da partizionare ha dimensione maggiore di 1 esegui:

      (b.1) Scegli l'elemento mediano del segmento

      (b.2) Partiziona il segmento in due rispetto al valore mediano

Versione iterativa del quicksort

(b.3) Salva i limiti della partizione più grande nella pila

(b.4) Reinizializza il processo sulla partizione più piccola ridefinendo LEFT e RIGHT.

Versione iterativa del quicksort

```
procedure quicksort(var a:nelements; n:integer);
var  left, right, newleft, newright, medio, mcorr, temp: integer;
     stack: pila;
begin
     creapila(stack);
     inpila(1,stack); inpila(n,stack);
     while not pilavuota(stack) do
       begin
         right:=leggipila(stack); fuoripila(stack);
         left:=leggipila(stack); fuoripila(stack);
         while left<right do
inizio par-   begin
tizionamento     newleft:=left; newright:=right;
                 medio:=(left+right)div2; mcorr:=a[medio];
```

Versione iterativa del quicksort

```
while a[newleft]<mcorr do newleft:=newleft+1;
while mcorr<a[newright] do newright:=newright-1;
while newleft<newright-1 do
  begin
     temp:=a[newleft]; a[newleft]:=a[newright];
     a[newright]:=temp;
     newleft:=newleft+1; newright:=newright-1;
     while a[newleft]<mcorr do newleft:=newleft+1;
     while mcorr<a[newright] do newright:=newright-1;
  end;
if newleft<=newright then
  begin
     if newleft<newright then
```


Versione iterativa del quicksort

```
        begin
           temp:=a[newleft]; a[newleft]:=a[newright];
           a[newright]:=temp
        end;
     newleft:=newleft+1; newright:=newright-1
  end;
if newright<medio then
  begin
     inpila(newleft, stack); inpila(right, stack);
     right:=newright
  end
else
```

Versione iterativa del quicksort

```
            begin
               inpila(left, stack); inpila(newright, stack);
               left:=newleft
            end;
         end;
      end;
   end;
```


Algoritmo di fusione (merge)

Problema: fondere due array di interi, già ordinati secondo il medesimo criterio, in un array unico, ordinato secondo lo stesso criterio.

L'algoritmo dovrebbe trarre vantaggio dall'ordinamento parziale che già esiste sui dati.

Siano a e b gli array ordinati in ordine crescente:

a:	15	18	42	51					m elementi
b:	8	11	16	17	44	58	71	74	n elementi

Algoritmo di fusione (merge)

La soluzione è l'array c così costituito:

	b	*b*	*a*	*b*	*b*	*a*	*a*	*b*	*a*	*b*	*b*	*b*
c:	8	11	15	16	17	18	42	44	51	58	71	74

Osserviamo che:

1) l'array risultante ha m+n elementi
2) è necessario esaminare tutti gli elementi di a e b
3) bisogna formalizzare un metodo di estrazione degli elementi da a e b

Mentre per fondere due array di 1 solo elemento

a: | 15 | b: | 8 |

Algoritmo di fusione (merge)

il procedimento è un banale confronto fra i due, con l'assegnamento a c[1] dell'elemento più piccolo e a c[2] del più grande, il problema si complica col crescere degli elementi. Niente infatti ci assicura che l'elemento posto in c[2] risulti più piccolo anche di altri elementi non ancora esaminati.

a:	15	18	42	…
b:	8	11	16	…
c:	8	…		

Algoritmo di fusione (merge)

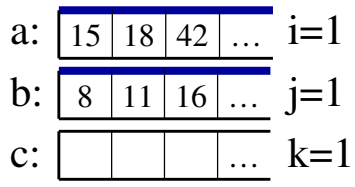
Per piazzare il prossimo elemento in c dobbiamo paragonare l'elemento risultato più grande, ovvero a[1], col successivo elemento di b, b[2].

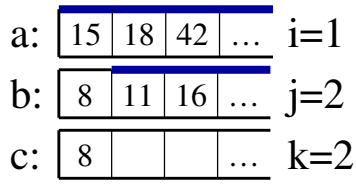
In generale, il prossimo elemento da piazzare in c dev'essere il minore dei primi elementi delle parti non ancora fuse.

Algoritmo di fusione (merge)

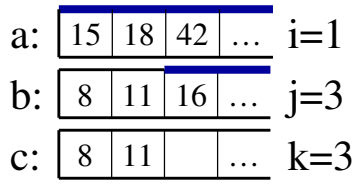
i, j indicano l'inizio del segmento di array da fondere

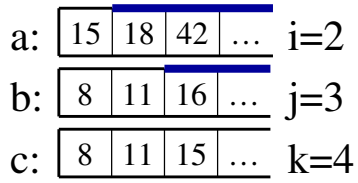
k indica il primo elemento libero di c





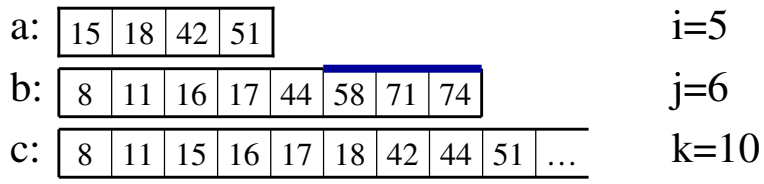
Algoritmo di fusione (merge)





Algoritmo di fusione (merge)

Cosa succede quando si esauriscono gli elementi di un array?



Basterà copiare nell'array risultante gli elementi dell'array più lungo.

Algoritmo di fusione (merge)

1. while (i≤m) and (j≤n) do
   a) Confronta a[i] e b[j] mettendo il più piccolo in c[k]
   b) aggiorna appropriatamente gli indici i, j, k
2. if i<m then
   copia il resto di a in c
   else
   copia il resto di b in c

Algoritmo di fusione (merge)

L'algoritmo potrebbe essere semplificato.

Infatti, confrontando gli ultimi elementi di ambedue gli array, possiamo subito stabilire quale array sarà completato per primo nella fusione e quale per ultimo.

if a[m]≤b[n] then
(a) fondi tutto a con b
(b) copia il resto di b
else
(a') fondi tutto b con a
(b') copia il resto di a

Algoritmo di fusione (merge)

Quindi possiamo definire la seguente

(1) PROCEDURA MERGE

1. definire gli array a[1..m] e b[1..n]
2. se l'ultimo elemento di a è $\leq$ dell'ultimo elemento di b allora
   (a) fondi tutto a con b
   (b) copia il resto di b

   altrimenti
   (a') fondi tutto b con a
   (b') copia il resto di a
3. dà il risultato c[1..m+n]

Algoritmo di fusione (merge)

È possibile pensare ad una sola procedura parametrica MERGECOPY che implementa i passi (a) e (b) (rispettivamente (a') e (b')).

Tale procedura può essere ulteriormente ottimizzata. Infatti, quando l'ultimo elemento dell'array a è minore del primo elemento dell'array b (o viceversa) allora la fusione consiste semplicemente nella copiatura prima di un array e poi dell'altro.

a: | 15 | 18 | 42 | 51 | b: | 58 | 71 | 74 |

Algoritmo di fusione (merge)

(2) **PROCEDURA MERGECOPY**

1. definire a[1..m] e b[1..n] con a[m]≤b[n]
2. se a[m]≤b[1] allora
   a) copia tutto a nei primi m elementi di c
   b) copia tutto b in c partendo dalla posizione m+1

   altrimenti
   a') fondi tutto a con b in c
   b') copia il resto di b in c partendo dalla posizione in cui è finita la fusione

Algoritmo di fusione (merge)

Il passo (a') è realizzato dalla

(3) **PROCEDURA SHORTMERGE**

1. Definire a[1..m], b[1..n] con a[m]≤b[n]
2. Mentre tutto a non è ancora fuso esegui
   (a) Se l'elemento corrente di a è ≤ dell'elemento corrente di b allora
      (a.1) copia l'elemento corrente di a nella posizione corrente di c
      (a.2) avanza alla successiva posizione di a

…

Algoritmo di fusione (merge)

altrimenti

(a'.1) copia l'elemento corrente di b nella posizione corrente di c

(a'.2) avanza alla successiva posizione di b

(b) avanza alla successiva posizione di c

Algoritmo di fusione (merge)

Mentre il passo (b') è realizzato dalla

(3) **PROCEDURA COPY**

1. definire b[1..m], c[1..n+m]
2. definire la posizione j del primo elemento di b da copiare
3. definire la posizione k del primo elemento libero di c
4. mentre non è ancora finito b esegui
   - (a) copia l'elemento j-simo di b nella posizione k-sima di c
   - (b) incrementa j
   - (c) incrementa k

Algoritmo di fusione (merge)

```
procedure copy(var b: nelements; var c: npelements;
          j,n: integer; var k:integer);
var i: integer;
begin
    for i:=j to n do
        begin
            c(k):=b(i); k:=k+1
        end
end;
```


Algoritmo di fusione (merge)

```
procedure shortmerge(var a,b: nelements; var c: npelements;
          m: integer; var i,k:integer);
var i: integer;
begin
    i:=1;
    while i<=n do
        begin
            if a(i)<=b(j) then
                begin
                    c(k):=a(i); i:=i+1
                end
            else
                begin
                    c(k):=b(j); j:=j+1
                end;
            k:=k+1
        end
end;
```

Algoritmo di fusione (merge)

```
procedure mergecopy(var a, b: nelements; var c: npelements;
            m,n: integer);
var i, j, k: integer;
begin
    i:=1; j:=1; k:=1;
    if a(m)<=b(i) then
        begin
            copy(a, c, i, m, k); copy(b, c, j, n, k)
        end
    else
        begin
            shortmerge(a, b, c, m, j, k); copy(b, c, j, n, k)
        end
end;
```


Algoritmo di fusione (merge)

```
procedure merge(var a, b: nelements; var c: npelements;
            m,n: integer);
begin
    if a(m)<=b(n) then
        mergecopy(a, b, c, m, n)
    else
        mergecopy(b, a, c, n, m)
end;
```

Algoritmo di fusione (merge)

```
program fusione(input, output);
const maxdim=50;
        maxdim2=100;

type nelements=array[1..maxdim] of integer;
      npelements=array[1..maxdim2] of integer;

var array1, array2: nelements; merged_array:npelements;
    dim1, dim2: integer;

procedure merge(var a, b: nelements; var c: npelements; m, n: integer);

procedure mergecopy(var a, b: nelements; var c:npelements;
                                m, n: integer);
var i, j, k: integer;
```


Algoritmo di fusione (merge)

```
procedure copy(var b: nelements; var c:npelements; j, n: integer;
                    var k: integer);
var i: integer;
begin
    …
end

procedure shortmerge(var a, b: nelements; var c:npelements;
                                m: integer; var j, k: integer);
var i: integer;
begin
    …
end
```

Algoritmo di fusione (merge)

```
begin {mergecopy}
    …
end {fine procedura mergecopy}

begin {merge}
    …
end {fine procedura merge}

begin {inizio programma fusione}
    lettura di dim1 ed array1
    lettura di dim2 ed array2
    merge(array1, array2, merged_array, dim1, dim2);
    stampa di merged_array
end
```


Algoritmo di fusione (merge)

Analisi della complessità

La complessità dell'algoritmo dipende dalla distribuzione dei dati.

Consideriamo come operazione dominante il confronto di un elemento di a con uno di b.

Si possono presentare due situazioni:

1) $a[m] \leq b[n] \wedge a[m] \leq b[1]$

   (viceversa $b[n] \leq a[m] \wedge b[n] \leq a[1]$)

   In questo caso non si effettuano altri confronti oltre a quelli riportati sopra.

Algoritmo di fusione (merge)

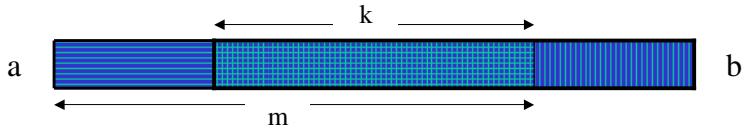
2) $a[m] \leq b[n] \wedge a[m] > b[1]$ (viceversa $b[n] \leq a[m] \wedge b[n] > a[1]$)

In tal caso si possono effettuare da $2+m+1$ a $2+m+(n-1)$ confronti

(rispettivamente da $2+n+1$ a $2+n+(m-1)$ confronti)

Quindi $f_{ott}(m, n) = 2$ $f_{pess}(m, n) = 1 + m + n$

In generale, se $a[m] \leq b[n]$ e per qualche $k \in \{1, 2, \ldots, n\}$ risulta $b[k] < a[m] \leq b[k+1]$



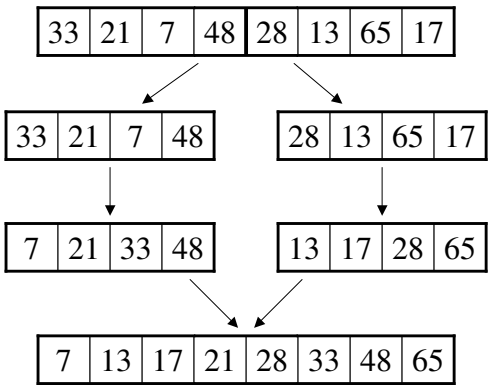
Allora si effettuano $2 + m + k$ confronti.

Ordinamento per fusioni successive (merge sort)

Problema: *ordinare, in ordine crescente, un vettore di interi*

L'idea alla base dell'algoritmo di MERGESORT è che l'ordinamento di un array di n interi può essere ottenuto dividendo l'array in due sequenze di n/2 elementi ciascuno, ordinando singolarmente ogni sequenza e quindi fondendo le due metà ordinate in un'unica sequenza.

Ordinamento per fusioni successive (merge sort)



insieme iniziale

suddivisione in 2 sottoinsiemi

ordinamento di ogni singolo sottoinsieme

combinazione (fusione) dei sottoinsiemi ordinati

Ordinamento per fusioni successive (merge sort)

L'ordinamento di ogni singolo sottoinsieme può essere effettuato secondo la stessa tecnica. In tal caso abbiamo la seguente procedura ricorsiva:

```
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 (merge sort)

Nota: la procedura 'MERGE' non può essere la stessa definita per la fusione di due vettori distinti. Occorre definire una nuova procedura MERGE che fonde segmenti adiacenti di uno stesso vettore.

Ordinamento per fusioni successive (merge sort)

```
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;
```

Ordinamento per fusioni successive (merge sort)

```
    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 (merge sort)

Analisi della complessità

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 dalla seguente relazione di ricorrenza:

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

Ordinamento per fusioni successive (merge sort)

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

$$f(n) = 2\,(2\,f(n/4) + c\,(n/2)) + cn = 4\,f(n/4) + 2\,cn =$$
$$= 4\,(2\,f(n/8) + c\,(n/4)) + 2\,cn = 8\,f(n/8) + 3\,cn =$$
$$\ldots$$
$$= 2^k\,f(1) + kcn = an + cn\log_2 n$$

Ordinamento per fusioni successive (merge sort)

Si dimostra facilmente che se $2^k<n<2^{k+1}$ allora

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

quindi $f(n)$ è un $\mathbf{O(n\log_2 n)}$.

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

Ma che dire della complessità in spazio?

Ordinamento per fusioni successive (merge sort)

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

Poiché abbiamo $\lfloor \log_2 n \rfloor$ chiamate ricorsive di MERGESORT (e quindi di MERGE) si ha che 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).

Ordinamento per fusioni successive (merge sort)

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

```
procedure ordinaperfusioni(var a: nelements; n: integer);
var b: nelements;
    procedure mergesort(var a,b: nelements; primo,ultimo:integer);
    var q: integer;
```

Ordinamento per fusioni successive (merge sort)

```
        procedure merge(var a,b: nelements; primo, ultimo, mezzo:
integer);
        var i,j,k,h: integer;
        begin {merge}    …
        end {merge}
    begin {mergesort}
        if primo<ultimo then
            begin
            q := (primo+ultimo) div 2; mergesort(a, b, primo, q);
            mergesort(a, b, q+1, ultimo); merge(a, b, primo, ultimo)
            end
    end {mergesort}
begin {main}
    mergesort(a,b,1,n)
end
```


La ricerca binaria

*Dato un elemento **x** ed un insieme di dati in ordine numerico strettamente crescente determinare se **x** è presente o meno nell'insieme.*

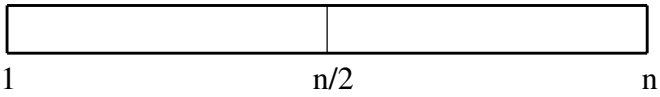
Il problema può essere paragonato a quello di cercare una parola in un dizionario o di trovare un nome in un elenco telefonico.

Quello che si fa in questi casi è scartare il più velocemente possibile grandi porzioni dell'elenco.

Il nostro obiettivo è trovare un algoritmo generale che consenta di compiere questo stesso processo indipendentemente dalla particolare distribuzione dei dati ordinati.

La ricerca binaria

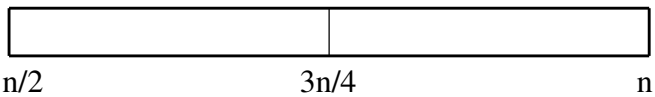
Si comprende facilmente che il valore cercato può essere o nella prima metà o nella seconda metà della lista (potrebbe anche essere il valore centrale).



Poiché la lista è ordinata, si può stabilire rapidamente in quale metà della lista si trovi il valore confrontando questi con il valore centrale. Questo confronto, da solo, consente di scartare la metà delle possibilità iniziali.

La ricerca binaria

Se il valore è nella seconda metà della lista possiamo ripetere lo stesso procedimento:



Ancora una volta, confrontando **x** con l'elemento di posizione 3n/4 è possibile scartare un quarto della lista iniziale dall'insieme delle possibili posizioni di **x**.

Possiamo continuare ad applicare questa idea di *ridurre il problema della metà ad ogni confronto* finchè non si incontra il valore cercato o finché non si stabilisce che **x** non è presente nella lista data.

La ricerca binaria

Strategia generale:

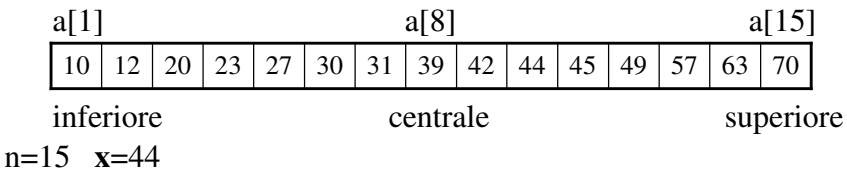
ripeti

> *l'esame del valore centrale dei dati che restano e sulla base del confronto con **x** elimina metà dei dati rimanenti*

*fino a quando non si trova il valore **x** o non si stabilisce che esso non è presente nella lista.*

La ricerca binaria

Esempio:



Se **x** è presente nella lista vogliamo anche conoscerne la posizione.

La posizione del valore centrale è data da:

centrale := (n+1) div 2 ≡ 8

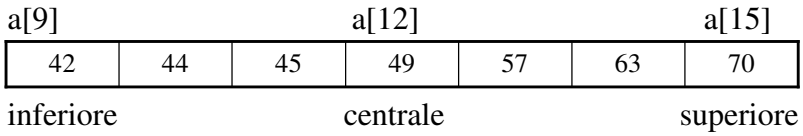
e si ha che a[centrale]=a[8]=39<44 per cui **x** è nelle posizioni a[9]…a[15].

La ricerca binaria

Il limite inferiore dev'essere aggiornato:

inferiore := centrale +1

mentre quello superiore resta invariato.



Il valore centrale è dato da:

centrale := (inferiore + superiore) div 2

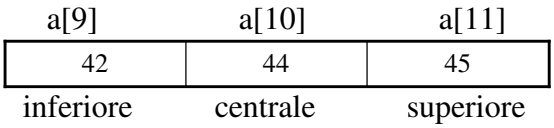
e confrontando a[12] con 44 si scopre che esso è minore di a[12], sicchè **x** deve trovarsi eventualmente in a[9]…a[11].

La ricerca binaria

Questa volta è il limite superiore ad essere cambiato:

superiore := centrale - 1

Quindi si ha:



Con il terzo confronto si trova il valore cercato e la sua posizione nell'array.

Ad ogni confronto, o diminuisce il limite superiore o aumenta quello inferiore.

La ricerca binaria

Quando il valore cercato è presente l'algoritmo termina, e accade che a[centrale] è proprio il valore cercato. Tuttavia questo test non è mai vero quando il valore cercato non è presente nella lista ordinata.

Come gestire anche questa situazione?

Se fosse **x**=43 andremmo avanti e giungeremmo all'incrocio dei due indici, inferiore=10 e superiore=9.

Pertanto il confronto:

inferiore > superiore

unitamente al test di uguaglianza del valore dell'array con il valore cercato può essere utile per la terminazione dell'algoritmo.

La ricerca binaria

Questo doppio controllo funziona anche quando **x** è minore di a[1] e quando **x** è maggiore di a[n].

Algoritmo:

1. Considera il numero **n** di elementi nella lista ed il valore **x** da cercare
2. Considera l'insieme ordinato a[1..n]
3. Stabilisci i limiti inferiore e superiore
4. Ripeti
   a) Calcola la posizione centrale dell'array rimanente

La ricerca binaria

b) Se il valore **x** cercato è maggiore di quello centrale

allora poni il limite inferiore uguale a quello centrale più 1

altrimenti poni il limite superiore uguale a quello centrale meno 1

finchè il valore cercato è trovato o inferiore diventa maggiore di superiore

5. Stabilisci trovato di conseguenza

La ricerca binaria

```
procedure ricerca_binaria(var a: n_elementi; n, x: integer;
                          var centrale: integer; var found: boolean);
var inferiore, superiore: integer;
begin
    inferiore:=1; superiore:=n;
    repeat
        centrale:=(inferiore+superiore) div 2;
        if x>a[centrale] then
            inferiore:=centrale + 1
        else
            superiore:=centrale -1;
    until (a[centrale]=x) or (inferiore>superiore)
    found:=(a[centrale]=x)
end
```

La ricerca binaria

La condizione di terminazione dell'algoritmo è piuttosto rozza. Ci chiediamo se sia possibile trovare una soluzione più semplice ed allo stesso tempo più efficiente.

Da un attento esame si comprende che la condizione

a[centrale] = x

non può essere eliminata.

Osserviamo che se **x** è presente nell'array vorremmo che dopo ogni iterata valga la seguente condizione:

a[inferiore] ≤ **x** ≤ a[superiore]

quindi si devono modificare diversamente gli indici inferiore e superiore.

La ricerca binaria

In particolare se accade che

x > a[centrale]

allora **x** dev'essere compreso fra a[centrale+1] ed a[superiore], quindi l'assegnazione:

inferiore := centrale + 1

è corretta, altrimenti **x** dev'essere compreso fra a[inferiore] ed a[centrale], pertanto si dovrà effettuare l'assegnazione:

superiore := centrale.

Così facendo, la condizione di terminazione diventa:

inferiore = superiore

La ricerca binaria

Questo controllo, che va bene se x è nella lista ordinata, dev'essere vagliato anche per i seguenti casi particolari:
1) c'è solo un elemento nell'array;
2) **x** è minore del primo elemento dell'array;
3) **x** è maggiore dell'ultimo elemento dell'array;
4) **x** è compreso fra a[1] ed a[n] ma non è presente nell'array.

(PER ESERCIZIO)

La ricerca binaria

Algoritmo:
1. Considera il numero **n** di elementi nella lista ed il valore **x** da cercare
2. Considera l'insieme ordinato a[1..n]
3. Assegna alle variabili inferiore e superiore i limiti dell'array
4. Mentre inferiore < superiore
   a) Calcola la posizione centrale del segmento rimanente dell'array su cui cercare

La ricerca binaria

b) Se il valore **x** cercato è maggiore di quello centrale
   allora poni il limite inferiore uguale a quello centrale più 1
   altrimenti poni il limite superiore uguale a quello centrale

5. Se l'elemento dell'array che occupa la posizione inferiore è uguale al valore cercato
   allora restituisci trovato
   altrimenti restituisci non trovato

La ricerca binaria

```
procedure ricerca_binaria(var a: n_elementi; n,x: integer; var centrale:
    integer; var found: boolean);
var inferiore, superiore: integer;
begin
    inferiore := 1; superiore := n;
    while inferiore < superiore do
      begin
       centrale := (inferiore+superiore) div 2;
       if x > a[centrale] then
           inferiore := centrale + 1;
       else
           superiore := centrale
      end;
    found := (a[inferiore]=x)
end
```

La ricerca binaria

Analisi della correttezza

Inizialmente valgono le seguenti condizioni:

$$n>0 \wedge \forall i \in \{1, 2, \ldots, n-1\} \quad a[i] \leq a[i+1]$$

Inoltre possiamo dire che se x è presente allora

$$a[1] \leq x \leq a[n]$$

La condizione invariante del ciclo è:

$$\text{inferiore} \geq 1 \quad \wedge \quad \text{superiore} \leq n \quad \wedge$$
$$\text{se x è presente allora } a[\text{inferiore}] \leq x \leq a[\text{superiore}]$$

La ricerca binaria

Il ciclo termina quando inferiore = superiore (vedi seguito), quindi all'uscita dal ciclo vale la seguente condizione:

$$\text{inferiore} = \text{superiore} \wedge \text{se x è presente allora}$$
$$x = a[\text{inferiore}] = a[\text{superiore}]$$

Quindi found=true se x=a[inferiore].

Verifichiamo la **terminazione** del programma.
Sia distanza=superiore - inferiore, allora la condizione di terminazione può essere riscritta come:

$$\text{while distanza} > 0 \text{ do } \ldots$$

La ricerca binaria

Dimostriamo che ad ogni passo del ciclo distanza viene dimezzata. Infatti, se x > a[centrale] allora si aggiorna inferiore come segue:

inferiore := centrale + 1

Quindi

distanza(passo i+1) =

= superiore(passo i+1) - inferiore(passo i+1) =

= superiore(passo i) - centrale(passo i) - 1 =

= superiore(passo i) -

$$-\left\lfloor \frac{\text{inferiore(passo i)} + \text{superiore(passo i)}}{2} \right\rfloor - 1$$

La ricerca binaria

Ma

$$\left\lfloor \frac{\text{inferiore(passo i)} + \text{superiore(passo i)}}{2} \right\rfloor =$$

$$= \begin{cases} \dfrac{\text{inferiore(passo i)} + \text{superiore(passo i)}}{2} & \text{se distanza è pari} \\ \dfrac{\text{inferiore(passo i)} + \text{superiore(passo i)} - 1}{2} & \text{altrimenti} \end{cases}$$

La ricerca binaria

Quindi

distanza(passo i+1) =

$$\begin{cases} \dfrac{\text{distanza(passo i)}}{2} - 1 & \text{se distanza(passo i) è pari} \\ \dfrac{\text{distanza(passo i)}}{2} - \dfrac{1}{2} & \text{altrimenti} \end{cases}$$

$$\left\lfloor \frac{\text{distanza(passo i)} - 1}{2} \right\rfloor$$

La ricerca binaria

Analogo risultato lo si ottiene nel caso in cui

$x \leq a[\text{centrale}]$

Quindi ad ogni passo del ciclo, distanza viene più che dimezzata. Si dimostra facilmente che

distanza(passo i+1) è pari se distanza(passo i) è dispari, e viceversa.

In particolare la sequenza delle distanze è:

n-1, …, 2, 1, 0

Quindi il ciclo termina perché

inferiore = superiore

La ricerca binaria

Analisi della complessità

L'operazione dominante di questo algoritmo è il confronto di x con un elemento di a.

Ad ogni passo del ciclo è eseguito un confronto. Pertanto il numero di confronti è legato al numero di volte in cui il ciclo è eseguito, k.

Il ciclo è eseguito finchè non resta che un solo elemento da confrontare, ovvero inferiore = superiore.

Allora dev'essere:

$$\frac{n}{2^k} = 1 \Rightarrow k = \lfloor \log_2 n \rfloor$$

La ricerca binaria

Poiché all'uscita dal ciclo si effettua un ulteriore confronto si ha una complessità pari a $\lfloor \log_2 n \rfloor + 1$

La complessità di questo algortimo non dipende dalla distribuzione dei dati.