Complessità degli algoritmi

Complessità degli algoritmi

- L'**efficienza** di un algoritmo si valuta in base all'utilizzo che l'algoritmo fa delle risorse del calcolatore:

CPU	**Memoria**
tempo	spazio

- Algoritmi diversi, pur occupando lo stesso spazio, possono richiedere tempi diversi:
  - due algoritmi richiedono l'utilizzo di un array, ma uno richiede la scansione di tutto l'array, e impiega un tempo t_1 , l'altro richiede la scansione solo di una parte dell'array, e impiega un tempo $t_2<t_1$.

Complessità degli algoritmi

- Si preferisce valutare l'efficienza in base al tempo impiegato per risolvere il problema.
- Come si può **misurare il tempo** impiegato dall'algoritmo? Con un cronometro?
- Ogni linguaggio ammette delle funzioni per calcolare il tempo: in alcuni si accede ad un "orologio" della macchina; spesso viene fornito un tempo calcolato in secondi o millisecondi e viene indicato il tempo trascorso a partire da una data fissa. (Cap. 5)

Complessità degli algoritmi

- In questo caso per calcolare il tempo si effettuano due successive chiamate della funzione tempo e si fa la differenza tra le due misure di tempo ottenute.
- È una **buona misurazione**?
- Lo è solo in parte; non lo è in senso assoluto.

Complessità degli algoritmi

- Infatti il tempo misurato in tale modo dipende:
  - dal **calcolatore** usato
  - dal **linguaggio** usato
  - dai **dati** che sono elaborati in quel momento.
- Non ha perciò senso dire che "l'algoritmo di ricerca impiega 10 sec."; il tempo dipende anche da fattori estranei all'algoritmo.

Complessità degli algoritmi

- Se vogliamo **confrontare** algoritmi che risolvono lo stesso problema misurando il tempo che impiegano, dovremo:
  - usare la **stessa macchina**
  - usare lo **stesso linguaggio**
  - usare gli **stessi dati**.
- Si potrà costruire un array di dati abbastanza grande; tale vettore sarà l'**unico** ingresso per confrontare algoritmi che risolvono quel problema in prove eseguite sullo **stesso** calcolatore e con lo **stesso** linguaggio.

Complessità degli algoritmi

- Come possiamo **stimare il tempo** per l'algoritmo in modo da avere una **misura assoluta**, vale a dire, una misura che dipenda **solo** dall'algoritmo?
- Osserviamo che il tempo di esecuzione è una funzione crescente della dimensione dei dati di ingresso.
- **Esempio**.
- Sommare n numeri dove n può assumere i valori n = 10, 1000, 100000000 .

Complessità degli algoritmi

- Il tempo per calcolare la somma aumenta con n

$$n \to +\infty \quad \Rightarrow \quad T(n) \to +\infty$$

- Non per tutti gli algoritmi è così: nell'algoritmo che calcola il massimo tra due numeri, in quello che calcola l'area del trapezio, nell'algoritmo che esegue delle prove sul troncamento della divisione tra interi, … i **dati** di ingresso sono una **quantità fissa** e non variano di dimensione.

Complessità degli algoritmi

- Nel calcolo della somma e del massimo su **n** dati, nell'algoritmo di ricerca, andiamo ad acquisire il valore di **n** e gli **n** valori dell'array: la **dimensione dei dati varia** con **n**.

- Osserviamo che il numero dei dati di ingresso **fa variare** il numero di **operazioni**:

  n = 10 l'algoritmo esegue 10 somme

  n = 10000 l'algoritmo esegue 10000 somme

Complessità degli algoritmi

- Si chiama **complessità computazionale** la funzione F(n) che calcola il numero di operazioni eseguite dall'algoritmo:

  $n \in \mathbb{N}$ dimensione dei dati di ingresso

  F(n) numero di operazioni

  $n \to +\infty \quad \Rightarrow \quad F(n) \to +\infty$

- Ha interesse sapere **come** la funzione F(n) "cresce", vale a dire quale è il suo **ordine di infinito**.

Complessità degli algoritmi

- Confrontiamo le due funzioni

  $y = x$ e $y = x^2$

  osservando i valori che assumono per x = 5, 10, 100

x	y = x	$y = x^2$
5	5	25
10	10	100
100	100	10000

  intuiamo che $y=x^2$ tende a $+\infty$ più rapidamente di y=x; $y=x^2$ ha un ordine di infinito maggiore di y=x.

Complessità degli algoritmi

- Quali sono le **operazioni** che un algoritmo esegue?
  - operazioni elementari
  - assegnazioni (accesso e assegnamento)
  - confronti
  - lettura/scrittura
  - invocazione di funzioni (passaggio parametri e gruppi di istruzioni all'interno della funzione)
  - strutture di controllo

Complessità degli algoritmi

- Dobbiamo **sommare** i **tempi** delle varie istruzioni, in tale modo troviamo il **tempo totale**.
- Non dobbiamo però essere così precisi: possiamo fare delle **semplificazioni**.
- Le operazioni fondamentali sono:
  confronti e **assegnazioni**

Complessità degli algoritmi

- Si valuta un tempo t_a (assegnazione) e t_c (confronto) indipendentemente dalla semplicità o meno dell'espressione.
- Si può anche calcolare il numero di accessi eseguiti per alcune variabili fondamentali nell'algoritmo: contare quante volte la variabile compare in un confronto o in un assegnazione.

Complessità degli algoritmi

- **Assegnazione**.
  1) a = 25;
  2) a = sqrt(3*sin(x) + cos(y)/7);
- Queste due assegnazioni necessitano di due tempi diversi e sicuramente sarà:
  $t_1 < t_2$ perché l'espressione 2) richiede più calcoli.
- Indichiamo con $\mathbf{t_a}$ il tempo per una assegnazione.

Complessità degli algoritmi

- **Sequenza** di assegnazioni**.**
  $<a_1>$
  $<a_2>$ k costante
  …. il numero di istruzioni
  $<a_k>$ non dipende da n
- **Sommiamo** i tempi:
  $k\, t_a \sim \mathbf{t_a}$ k **non** dipende da n
- Se abbiamo due funzioni $F_1(n)$ $F_2(n)$ con lo stesso ordine di infinito, può essere importante stimare anche il valore di k.

Complessità degli algoritmi

- **Struttura condizionale**.
  se P
  allora <a1>
  altrimenti <a2>
  //finese
  <a1> e <a2> rappresentano gruppi di istruzioni
- Il predicato P potrà essere:
  1) a<b
  2) (a<b) o (non S) e ((a==c) o (c!=h)
- Certamente si avrà $t_1 < t_2$ ma consideriamo t_p il tempo per valutare un predicato.

Complessità degli algoritmi

- Avremo
  $t_p + t_{a1}$
  $t_p + t_{a2}$
- Se <a1> e <a2> non dipendono da **n** (sequenza di istruzioni)
  $t_p + t_{a1} \sim t_p + t_{a2} \sim \mathbf{t_p}$
- abbiamo un numero **costante** di operazioni, come nell'algoritmo del max(a,b).

Complessità degli algoritmi

- Quali sono le strutture che variano con **n**? Quando eseguiamo una **scansione** di un array (lettura, stampa, somma, ricerca, …) consideriamo un numero **n** (n>0) di dati di ingresso.
- **Struttura iterativa**.
- Consideriamo un ciclo con incremento fisso e passo 1:

  per i da 1 a n **eseguire**
  iterazione //indipendente da n
  //fineper
- Indichiamo con $\mathbf{t_p}$ il tempo per il predicato e con $\mathbf{t_i}$ il tempo per l'iterazione (per ora indipendente da **n**):

Complessità degli algoritmi

- Avremo:

  t_p : eseguito n+1 volte
  (n volte con P vero, 1 volta con P falso)

  t_i : eseguito n volte

  $\Rightarrow \quad (n+1)\cdot t_p + n \cdot t_i \sim \mathbf{n \cdot c}$

  c = costante: "costo" dell'iterazione:

$(n+1)\cdot t_p + n\cdot t_i = n\cdot t_p + t_p + n\cdot t_i \le n\cdot t_p + n\cdot t_p + n\cdot t_i = 2n\cdot t_p + n\cdot t_i = n\cdot(2\cdot t_p + t_i) = n \cdot c$

$(n > 0 \quad \text{quindi} \quad n \ge 1)$

Complessità degli algoritmi

- Cosa cambia se anche l'iterazione dipende da **n**? Supponiamo che l'iterazione sia un ciclo dello stesso tipo, che viene eseguito n volte:

  1. **per** i da 1 a n **eseguire**
  2. **per** k da 1 a n **eseguire**
     iterazione
     //fineper
  //fineper

Complessità degli algoritmi

$(n+1)\cdot t_{p1} + ((n+1) \cdot t_{p2} + n\cdot t_i) \cdot n =$
$= (n+1)\cdot t_{p1} + ((n+1) \cdot t_{p2})\cdot n + n^2\cdot t_i \le$
$\le 2n\cdot t_{p1} + 2n\cdot t_{p2}\cdot n + n^2\cdot t_i \le$
$\le n^2 (2\, t_{p1} + 2\, t_{p2} + t_i) = \mathbf{n^2 \cdot c}$

avendo considerato:
$n+1 \le 2n$, $2n \le 2n^2$ con n naturale (positivo)

c= costante : "costo" dell'iterazione

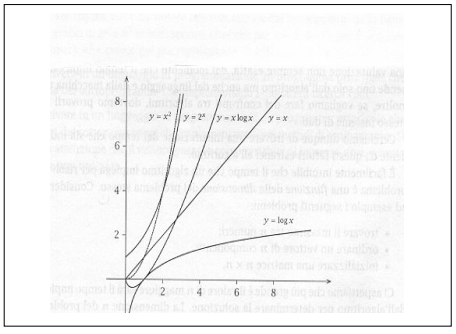
Complessità degli algoritmi

- Consideriamo delle **funzioni di riferimento** e calcoliamo la complessità degli algoritmi confrontandola con queste funzioni.

  $y = \log x$
  $y = x$
  $y = x \log x$
  $y = x^2$
  $y = 2^x$
  $y = e^x$

$y = x^2$ e $y = 2^x$ si incontrano per x=2 e x=4

Complessità degli algoritmi



Complessità degli algoritmi

- Notazione **O** (o-grande), **Ω**, **Θ** .
- Ci interessa sapere come cresce F(n) al crescere di n, sempre nell'ipotesi che

  $F(n) \rightarrow +\infty$ quando $n \rightarrow +\infty$
- Sia f(n) la funzione di complessità che cerchiamo. Si dice che

  $f(n)$ è $\mathbf{O}(g(n))$

  se $\exists\ c, n_0 > 0 \mid \forall\ n \geq n_0 \quad f(n) \leq c \cdot g(n)$
- Come ordine di infinito f "non cresce più di" g.

Complessità degli algoritmi

- Se f(n) = O(g(n)) significa che **g**(n) è una **limitazione superiore** per **f**(n): il calcolo esatto di f(n) è troppo complicato e ci limitiamo a stimarne una limitazione superiore.
- Esempio.

  $n^2 + n$ è $\mathbf{O}(n^2)$

  infatti:

  $n^2 + n \leq n^2 + n^2 = 2 \cdot n^2$

Complessità degli algoritmi

- Sia f(n) la funzione di complessità che cerchiamo. Si dice che

  $f(n)$ è $\mathbf{\Omega}(g(n))$

  se $\exists\ c, n_0 > 0 \mid \forall\ n \geq n_0 \quad c \cdot g(n) \leq f(n)$
- Come ordine di infinito f "cresce almeno quanto" g;

  g(n) è una **limitazione inferiore** per f(n).
- Esempio. $n^2 + n$ è $\mathbf{\Omega}(n)$

  infatti:

  $2n = n + n \leq n^2 + n$

Complessità degli algoritmi

- Sia f(n) la funzione di complessità che cerchiamo. Si dice che

  $f(n)$ è $\mathbf{\Theta}(g(n))$

  se $\exists \quad c_1, c_2, n_0 > 0 \quad \mid \quad \forall\ n \geq n_0$

  $c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$
- Come ordine di infinito f "cresce quanto" g.

  Esempio. n^2, $1000\ n^2$, $1/100\ n^2$ sono tutte $\mathbf{\Theta}(n^2)$

  varia solo la costante moltiplicativa.

Classi di complessità

- Quando scriviamo un algoritmo, vogliamo **stimare** la sua complessità cercando di individuare la funzione g(n) che approssima f(n). In tale modo gli algoritmi vengono suddivisi in **classi di complessità**:

costanti	k	(non dipendono da n)
logaritmo	$\log n$, $\log_2 n$	
lineari	n	
	$n \log n$	
polinomiali	n^k	$k = 2, 3, \ldots$
esponenziali	$n!$, a^n , n^n	$a \neq 0, 1$

Casi di prova per stimare la complessità

- Quando si scrive un algoritmo si deve **sempre** dare una stima della **limitazione superiore**.
- È opportuno dare una stima della limitazione inferiore e sarebbe importante stimare anche un comportamento medio tra le due stime, ma questo spesso è difficile.
- **O**(g) limitazione **superiore**: sono richieste **al più** g(n) operazioni
- **Ω**(g) limitazione **inferiore**: sono richieste **almeno** g(n) operazioni.

Casi di prova per stimare la complessità

- **Casi di prova**.
- Caso **peggiore**: i dati sui quali l'algoritmo richiede il **massimo** numero di operazioni.
- Caso **favorevole**: i dati sui quali l'algoritmo richiede il **minor** numero di operazioni.
- Caso **medio**: i dati che richiedono un numero medio di operazioni.
  - Nel calcolo della complessità non si considerano i cicli per acquisire i dati: sono uguali per tutti gli algoritmi e sono $\Theta(n)$.

Complessità dell'algoritmo di ricerca lineare

- **Caso I**: elementi distinti.

```
i ← 0
trovato ← falso
mentre  i ≠  n  e   non trovato     eseguire
          i ← i+1
          se  a[i] == b
              allora     trovato ← vero
          //fine
 //fine
 se  trovato
          allora   stampa  "trovato al posto " i
          altrimenti  stampa  " elemento non presente"
 //fine
```

Complessità dell'algoritmo di ricerca lineare

- **Caso favorevole**.

  Si ha quando a[1] = b: primo elemento dell'array:

  $2t_a + t_p + t_a + t_c + t_a + t_p + t_c + t_{stampa}$

  inizio V i←i+1 a[i] == b trovato F se risultato

  la funzione **non** dipende da n, quindi è costante:

  $\mathbf{\Omega}(1)$

  è il caso con il minimo numero di operazioni.

Complessità dell'algoritmo di ricerca lineare

- **Caso peggiore**.

  Si ha quando il ciclo viene eseguito fino in fondo.

1) l'elemento non c'è:

$2t_a + (n+1)\,t_p + nt_a + nt_c + t_c + t_{stampa}$

inizio predicato i←i+1 a[i] == b se risultato

la funzione è del tipo $\mathbf{c \cdot n}$, quindi $\mathbf{O}(n)$.

Complessità dell'algoritmo di ricerca lineare

2) l'elemento c'è ed è l'ultimo:

a[n] = b

$2t_a + (n+1)\,t_p + nt_a + nt_c + t_a + t_c + t_{stampa}$

inizio predicato i←i+1 a[i] == b trovato se risultato

la funzione è del tipo $\mathbf{c \cdot n}$, quindi $\mathbf{O}(n)$.

- Il massimo numero di operazioni è dato da una funzione lineare.

Complessità dell'algoritmo di ricerca lineare

- **Caso II**: elementi ripetuti.

```
k ← 0
per i da  1  a   n   eseguire
  se a[i] == b
          allora    k ← k+1
                    posiz[k] ← i
 //finescelta
//fineciclo
se k==0
   allora stampa "elemento non presente"
    altrimenti   stampare i primi   k  valori  dell'array posiz:
                posiz[1], …, posiz[k]
```

Complessità dell'algoritmo di ricerca lineare

- La struttura iterativa for ("per") viene eseguita sempre completamente.
- **Caso peggiore**. Gli elementi sono tutti uguali a b: ogni volta si assegna un valore a `posiz`

  $t_a + (n+1)t_p + nt_c + 2nt_a + t_c + t_{stampa}$ ⇒ **O**(n)
- **Caso favorevole**. L'elemento non c'è: il numero di assegnazioni è 0

  $t_a + (n+1)t_p + nt_c + t_c + t_{stampa}$ ⇒ **Ω**(n)

  Pertanto la complessità è: **Θ**(n)

Complessità dell'algoritmo di ricerca lineare

- **Caso III**. Ricerca su dati ordinati.
- La complessità è:

  Ω(1) nel caso favorevole (primo elemento) ed è **O**(n) nel caso peggiore (ultimo o mancante).
- **Esercizio**.

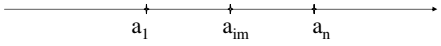
  Dimostrare l'affermazione precedente.
- Con un array ordinato si ha anche un altro algoritmo che non esamina tutti gli elementi.

Ricerca binaria (dicotomica)

- Supponiamo di cercare un nome in un elenco ordinato di nomi: vocabolario, elenco telefonico. Sfruttiamo l'ordine lessicografico (alfabetico) e incominciamo a "dividere" l'elenco in due parti pensando alla iniziale del nome, poi alla lettera successiva, ... non iniziamo dalla prima parola dell'elenco se cerchiamo un nome che inizia per M.
- Analogamente se pensiamo dei numeri appartenenti ad un intervallo sulla retta.

Ricerca binaria (dicotomica)

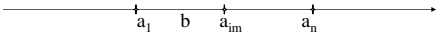
- **Analisi**.
- Consideriamo l'array $a = (a_1, a_2, \ldots a_n)$



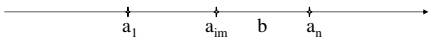
  con im = (1+n)/2 **indice** di mezzo
- Esaminiamo a_{im} : se $b = a_{im}$ allora abbiamo trovato l'elemento, altrimenti vediamo se risulta $b<a_{im}$ oppure $b>a_{im}$ e proseguiamo la ricerca solo nella parte "possibile".

Ricerca binaria (dicotomica)

- Caso **$b < a_{im}$**
- Gli elementi di destra non si guardano



- Caso **$b > a_{im}$**
- Gli elementi di sinistra non si guardano



Ricerca binaria (dicotomica)

- Indichiamo con **is** e **id** gli estremi della parte di array che stiamo guardando: $a_{is} \leq b \leq a_{id}$. Tali valori vengono inizializzati con i valori della prima e dell'ultima componente (prima =0,1; ultima = n-1, n).
- Si confronta b con a_{im} e se risulta $a_{im} \neq b$ cambiamo il valore degli estremi **is** e **id;** questo è il modo per "non guardare più" una parte di array:

  se $b < a_{im}$ cambierà id

  se $b > a_{im}$ cambierà is
- Perciò il valore di **id** diminuisce e quello di **is** aumenta: per proseguire la ricerca dovrà essere sempre is ≤ id.

Ricerca binaria (dicotomica)

- Vediamo con un esempio che questo meccanismo "funziona":

a = (1, 5, 6, 8, 11, 15) n = 6 b = 5

1) is = 1 id = 6 im = (1+6)/2 = 7/2 = 3
   $a_{im} = a_3 = 6$ e $6 \neq 5$ $a_{im} > b$
   id varia e assume il valore im-1: id=2
2) is = 1 id = 2 im = (1+2)/2 = 3/2 = 1
   $a_{im} = a_1 = 1$ e $1 \neq 5$ $a_{im} < b$
   is varia e assume il valore im+1: is=2
3) $a_{im} = a_2 = 5$ e 5=5 trovato= vero

Ricerca binaria (dicotomica)

- Algoritmo ricerca binaria

definizione variabili a array , b intero
                       n, is, id, im intero
                       trovato logico
acquisire e stampare n, a, b
se a[1] <= b e b <= a[n]
  allora //eseguire la ricerca
    is ← 1
    id ← n
    trovato ← falso

Ricerca binaria (dicotomica)

mentre non trovato e is <= id **eseguire**
  im ← (is+id)/2
  se a[im] == b
    allora trovato ← vero
    altrimenti //scegliere dove proseguire
      se b < a[im]
        allora id ← im-1
        altrimenti is ← im+1
      //finese
  //finese
//finementre

Ricerca binaria (dicotomica)

  se trovato
    allora stampa "trovato al posto " im
    altrimenti "non trovato
  //finese
altrimenti stampa "b è esterno ad a"
//finese
//fine algoritmo

Calcoliamo la complessità di questo algoritmo.

Complessità dell'algoritmo di ricerca binaria

- Il primo confronto a[1] <= b e b <= a[n] viene eseguito per verificare se è oppure no possibile eseguire la ricerca: nel caso in cui questo predicato fosse falso, non si esegue l'algoritmo.
- Questo non viene considerato come caso favorevole, perché di fatto l'algoritmo non inizia nemmeno.

Complessità dell'algoritmo di ricerca binaria

- **Caso favorevole**.
- L'elemento viene trovato con un solo confronto; ma non è il primo elemento, è quello centrale: b = a[(1+n)/2]

$$3t_a + t_p + \underbrace{t_a + t_c}_{a[im]==b} + \underbrace{t_a + t_p + t_c}_{trovato\ se} + t_{stampe}$$

numero costante di operazioni: $\Omega(1)$

Complessità dell'algoritmo di ricerca binaria

- **Caso peggiore**.
- Si esegue l'iterazione il massimo numero di volte quando: l'elemento o è l'ultimo "guardato" oppure non è presente.
- Quante operazioni ci sono nel ciclo?

  $t_a + t_c + t_c + t_a$ numero costante
  $\quad\quad a[im]==b \quad a[im]>b \quad is\ o\ id$
- Quante iterazioni? Facciamo una stima del numero di iterazioni eseguite valutando quanti elementi restano ad ogni divisione dell'array (gli elementi sono in numero finito).

Complessità dell'algoritmo di ricerca binaria

- Supponiamo che $n = 2^k$

1^a iterazione restano 2^{k-1} elementi
2^a iterazione restano 2^{k-2} elementi
3^a iterazione restano 2^{k-3} elementi
…..
k-esima iterazione restano $2^{k-k} = 1$ elementi: is=id
(k+1)-esima iterazione: se l'elemento è presente trovato diventa vero, altrimenti trovato è falso e is>id: il ciclo termina:

$\mathbf{O}(k)$ con $k = \log_2 n$: $\mathbf{O(\log_2 n)}$

Si può dimostrare anche per valori di n qualunque ($2^{k-1} \le n \le 2^k$).

Casi di prova

Casi di prova

- Quando si vuole testare un algoritmo si devono **costruire vari casi di prova**. Invece di eseguire il programma più volte si può costruire un file di dati contenente tutti i casi che si vogliono provare.
- **Esempio**.
- Supponiamo di voler provare un algoritmo di ricerca: l'array sarà uno solo e i casi di prova saranno costituiti da diversi valori di b (esterno, presente, presente più volte, ecc.)

Casi di prova

- Lo schema del programma sarà perciò del tipo:

```
//acquisire e stampare l'array
//acquisire la quantità dei casi di prova:
          // numprove
for(int k = 1; k <= numprove; k++){
   //acquisire l'elemento b da cercare
   //chiamare il metodo di ricerca
   //stampa risultati
   cout<<"prova numero "<<k
        <<" elemento cercato = "<<b<<endl;
}
```

Casi di prova

- Il file dei dati avrà pertanto la seguente organizzazione:
  - il valore di **n** e gli **n** elementi per l'array
  - il valore di **numprove**
  - gli elementi da cercare (tanti quanto è il valore di numprove)
- Il programma viene eseguito una sola volta e produce i vari risultati dell'esecuzione dell'algoritmo, ottenuti con i diversi valori da cercare.

Il problema dell'ordinamento

Il problema dell'ordinamento

- Dati **n** elementi sui quali si possa considerare un relazione **d'ordine totale** (**ogni** coppia di elementi è confrontabile) costruire la permutazione ordinata, vale a dire trovare una disposizione degli elementi in modo tale che si abbia:

  $\mathbf{a_1 < a_2 < a_3 < \ldots < a_n}$

- Problema analogo è quello di considerare gli elementi ordinati in ordine decrescente.

  (Capitolo 3)

Il problema dell'ordinamento

- Vedremo metodi diversi e ne calcoleremo la complessità.
- I metodi di ordinamento saranno:
  - ordinamento scansione (o selezione diretta)
  - bubblesort
  - ordinamento per inserimento
  - ordinamenti **ricorsivi**
    - quicksort
    - mergesort
- La **complessità** varia da $\mathbf{O(n^2)}$ a $\mathbf{O(n \cdot \log_2 n)}$

Ordinamento per scansione

- **Analisi**.
- Vogliamo mettere gli elementi nell'ordine:

  $\mathbf{a_1 < a_2 < a_3 < \ldots < a_n}$

- Osserviamo che:

  a_1 è il minimo tra gli elementi $\{a_1, a_2, \ldots a_n\}$

  a_2 è il minimo tra gli elementi $\{a_2, a_3, \ldots a_n\}$

  . . . . .

  a_{n-1} è il minimo tra gli elementi $\{a_{n-1}, a_n\}$

  a_n è al suo posto.

Ordinamento per scansione

- **Progetto**.
- Cerchiamo un modo per avere al **primo** posto il **minimo** tra tutti, al **secondo** posto il **minimo** tra $\mathbf{a_2}$ e i rimanenti, al posto **(n-1)-**esimo il **minimo** tra $\mathbf{a_{n-1}}$ e $\mathbf{a_n}$.
- Avremo una **struttura iterativa** che "mette a posto" gli elementi da **1** a **n-1**, dove "mettere a posto" significa:

  mettere al posto **i**-esimo l'elemento **minimo** tra $\mathbf{a_i}$ e gli elementi rimanenti.

Ordinamento per scansione

mentre ci sono elementi da ordinare **eseguire**

mettere all'**i-esimo** posto il **minimo**

degli elementi $\{a_i, a_{i+1}, \ldots, a_n\}$

//finementre

- La struttura iterativa "mentre ci sono elementi da ordinare" sarà un ciclo del tipo:

  per i da **1** a **n-1** **eseguire**

Ordinamento per scansione

- Cerchiamo un modo per: trovare il **minimo.**
- Abbiamo **due strategie**:
- **I.** considerare gli elementi

  $\mathbf{a_i}$ e $\mathbf{a_k}$ con k = i+1, …, n

  e **scambiarli** se non sono nell'ordine
- **II.** calcolare il **valore** e la **posizione** del minimo ed eseguire un solo scambio tra $\mathbf{a_i}$ e il **minimo** trovato.

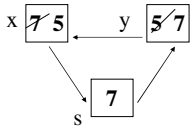
Ordinamento per scansione

- Come si effettua lo **scambio** del valore tra due variabili?
- Consideriamo due variabili x e y, vogliamo scambiare il loro contenuto:

Ordinamento per scansione

- **Non** possiamo fare subito l'assegnazione x=y altrimenti x e y avrebbero lo stesso valore. **Dobbiamo** utilizzare una variabile di supporto, s, dello stesso tipo di x e y, per depositare temporaneamente uno dei due valori:

1) s=x;
2) x=y;
3) y=s;



- Uno scambio si fa con 3 assegnazioni

Ordinamento per scansione

```
//ordinamento per scansione con scambio
per i da 1 a n-1 eseguire
    //mettere in a_i il minimo tra a_i , a_{i+1} , …, a_n
    per k da i+1 a n eseguire
        se a[i] > a[k]
            allora //scambiare a_i con a_k
                   s ← a[i]
                   a[i] ← a[k]
                   a[k] ← s
        //finese
    //fineper
//fineper        //ciclo esterno
```

Ordinamento per scansione

- Calcoliamo la **complessità** dell'algoritmo per scansione.
- L'operazione fondamentale eseguita nella struttura iterativa è il **confronto**: questo viene eseguito **sempre**, mentre lo scambio viene eseguito solo nel caso vero.
- Quanti confronti vengono eseguiti?
- Il ciclo esterno varia da 1 a n-1, il ciclo interno varia da i a n: i non è fisso e quindi il calcolo esatto è un po' più laborioso.

Ordinamento per scansione

n° iterazione	elementi	n° confronti
i=1	(a_1, a_2) (a_1, a_3) … (a_1, a_n)	**n-1**
i=2	(a_2, a_3) … (a_2, a_n)	**n-2**
……		…….
i=n-1	(a_{n-1}, a_n)	**1**

Sommiamo i confronti:

$1 + 2 + 3 + \ldots. + n-2 + n-1 = n \cdot (n-1)/2$

Ordinamento per scansione

- Un modo facile per ricordare la somma (uno dei due numeri interi n o n-1 è pari) è:
  (il primo + l'ultimo) × (metà del numero di elementi presenti nella formula)
- Si può dimostrare la formula per induzione (progressione aritmetica).
- Verifichiamo ad esempio per n=7:
  $1 + 2 + 3 + 4 + 5 + 6 = (1 + 6) \times 6/2 = 7 \times 3 = 21$

Ordinamento per scansione

- **Caso peggiore**.
- All'interno dell'iterazione si fanno tutti i confronti e tutti gli scambi.
- Dati: vettore è ordinato in **ordine inverso**
  a = (8, 6, 5, 3, -2 -7)

n(n-1)/2 confronti + 3n(n-1)/2 assegnazioni
Quindi **O**(n^2 / 2).

Ordinamento per scansione

- **Caso favorevole**.
- Si fanno tutti i confronti e nessuno scambio
- Dati: vettore già ordinato
  a = (2, 4, 5, 8, 20, 34, 51)

n(n-1)/2 confronti + 0 assegnazioni
Quindi **Ω**(n^2/2)

Ordinamento per scansione

- L'algoritmo di **ordinamento lineare** con **scambio** ha complessità:

- **O**(n^2/2) nel caso peggiore (limitazione superiore)
- **Ω**(n^2/2) nel caso favorevole (limitazione inferiore)
- Pertanto
  Θ(n^2 / 2)

Ordinamento per scansione

- Se vogliamo ordinare l'array tenendo conto del valore del minimo e della sua posizione, dobbiamo avere una variabile **min**, per memorizzare il valore minimo, e un indice **imin** per memorizzare la posizione.
- Possiamo anche utilizzare la sola variabile **imin** e accedere al valore del minimo tramite **a[imin]**.
- Con questa seconda scelta si fanno meno assegnazioni.
- Esercizio: verificare l'affermazione.

Ordinamento per scansione

```
//ordinamento per scansione con minimo
  per i da 1 a n-1 eseguire
     //mettere in a_i il minimo tra a_i , a_{i+1} , …, a_n
     imin ← i
     per k da i+1 a n eseguire
          se a[imin] > a[k]
             allora imin ← k
          //finese
     //fineper
```

Ordinamento per scansione

//dopo il ciclo interno si deve mettere a_{imin}
// al posto di a_i : scambiare a_i con a_{imin}
s ← a[i]
a[i] ← a[imin]
a[imin] ← s
//fineper //ciclo esterno

Ordinamento per scansione

- **Caso peggiore**.
- All'interno dell'iterazione il confronto dovrà essere sempre vero. In realtà ciò non si verifica per tutte le iterazioni, perché con lo scambio si ordina anche la seconda parte dell'array.
- Dati: vettore è ordinato in **ordine inverso**

confronti: n (n-1) / 2
assegnazioni:
4(n-1) (ciclo esterno) + n(n-1)/2 (al più nel ciclo interno)

Ordinamento per scansione

- **Caso favorevole**.
- Si fanno tutti i confronti e solo le assegnazioni del ciclo esterno.
- Dati: vettore già ordinato

confronti: n (n-1) / 2
assegnazioni: 4(n-1) (solo nel ciclo esterno)

Ordinamento per scansione

- Le operazioni dei due algoritmi hanno un costo diverso: uno scambio richiede tre assegnazioni; lo scambio è una operazione "costosa".
- Confrontiamo le due versioni:
- Il numero di **confronti** nei due algoritmi è uguale ed è sempre:

n(n-1) / 2

(un confronto è "più costoso" di una assegnazione)

Ordinamento per scansione

- Varia il numero di **assegnazioni**:
  nel caso **favorevole** si ha:
  0 (scambio) **4(n-1)** (minimo)

  nel caso **peggiore** si ha:
  3(n-1)n /2 (scambio) **(n-1)n / 2 + 4(n-1)** (minimo)

- Verificare che per n>4
  3(n-1)n /2 > (n-1)n / 2 + 4(n-1)

Ordinamento bubblesort

- Si eseguono confronti tra elementi **successivi**:
  (a_1, a_2) (a_2, a_3) (a_3, a_4) … (a_{n-1}, a_n)
  e se gli elementi non sono nell'ordine allora si **scambiano**. Con questa tecnica la componente con valore **più grande** va in fondo (al posto di a_n):
  (9, 2, 8, 10, 1, 4) diventa (2, 8, 9, 1, 4, **10**)
- Si ricomincia dell'inizio, dalla coppia (a_1, a_2), e si "mette a posto" a_{n-1}, poi a_{n-2}, … , infine a_2.

Ordinamento bubblesort

- Si può partire dalla fine del vettore, iniziando il confronto dalla coppia (a_{n-1}, a_n) e andare indietro. Con questa tecnica la componente con valore **più piccola** va al posto di a_1:

  (9, 2, 8, 10, 1, 4) diventa (**1**, 9, 2, 8,10, 4)
- In questo caso si ricomincia nuovamente dalla coppia (a_{n-1}, a_n) e si mette in ordine a_2, poi a_3 si continua fino ad a_{n-1} .

Ordinamento bubblesort

- **Progetto.**
- Consideriamo la prima versione: si inizia sempre dalla coppia (a_1, a_2); l'indice del ciclo che esegue i confronti è l'indice della **prima componente** della coppia. Alla prima iterazione varia da 1 a n-1 (si mette in ordine a_n), nella seconda iterazione l'indice varia da 1 a n-2 (si mette in ordine a_{n-1}); nella terza iterazione l'indice varia da 1 a n-3 (si mette in ordine a_{n-2}); ecc. nell'ultima iterazione l'indice varia da 1 a n-(n-1)=1 (si mette in ordine a_2); con l'ultimo confronto è in ordine anche a_1.

Ordinamento bubblesort

```
//ordinamento bubblesort: prima versione
 per  i  da  1  a    n-1    eseguire
        per k  da   1  a   n-i   eseguire
            se   a[k] > a[k+1]
                allora  //scambiare a_k  con  a_{k+1}
                            s   ←   a[k]
                            a[k]  ←   a[k+1]
                            a[k+1] ←   s
            //finese
      //fineper
  //fineper
```

Ordinamento bubblesort

- Quanti confronti?

n° iterazione	elementi	n° confronti
i=1	(a_1, a_2) (a_2, a_3) … (a_{n-1}, a_n)	**n-1**
i=2	(a_1, a_2) … (a_{n-2}, a_{n-1})	**n-2**
	……	…….
i=n-1	(a_1, a_2)	**1**

Ordinamento bubblesort

- I **confronti** vengono **sempre** eseguiti: se il confronto risulta vero si esegue lo scambio, se risulta falso non si esegue lo scambio.
- Sommiamo i confronti:

  $1 + 2 + 3 + \ldots. + n-2 + n-1 = n \cdot (n-1)/2$

  (come per l'ordinamento per scansione)

Ordinamento bubblesort

- Quante assegnazioni?
- **Caso favorevole**.

  0 assegnazioni (già ordinato)
- **Caso peggiore**.

  $3 \cdot n \cdot (n-1)/2$ (ordine inverso)
- Pertanto anche il bubblesort è $\Theta(n^2 / 2)$.

Confronto tra i vari metodi di ordinamento

- Consideriamo il seguente array e vediamo come si muovono gli elementi (alla prima iterazione).

  a = (40, 20, 1, 7, 10, 0) n=6

1) lineare con scambio

40	20	1	7	10	0	scambio 40, 20
20	40	1	7	10	0	scambio 20, 1
1	40	20	7	10	0	scambio 1, 0
0	**40**	**20**	**7**	**10**	**1**	

Confronto tra i vari metodi di ordinamento

2) lineare con minimo

40 20 1 7 10 0

imin ~~1~~ ~~2~~ ~~3~~ 6 scambio 40, 0

0 20 1 7 10 40

3) bubblesort (massimo in fondo)

40	20	1	7	10	0	scambio 40, 20
20	40	1	7	10	0	scambio 40, 1
20	1	40	7	10	0	scambio 40, 7
20	1	7	40	10	0	scambio 40, 10
20	1	7	10	40	0	scambio 40, 0
20	**1**	**7**	**10**	**0**	**40**	

Confronto tra i vari metodi di ordinamento

4) bubblesort (minimo in testa)

40	20	1	7	10	0	scambio 0, 10
20	40	1	7	0	10	scambio 0, 7
20	1	40	0	7	10	scambio 0, 40
20	1	0	40	7	10	scambio 0, 1
20	0	1	40	7	10	scambio 0, 20
0	**20**	**1**	**40**	**7**	**10**	

Varianti del bubblesort

- Osserviamo che nel **bubblesort** si confronta una componente con la successiva, a_k con a_{k+1}; pertanto, **se non ci sono scambi** significa che l'array è ordinato.
- Esempio.

  a= (2, 4, 5, 8, 21, 34, 58, 90)
- Le coppie considerate sono:

  (2, 4) (4, 5) (5, 8) (8, 21) (21, 34)

  (34, 58) (58, 90)

Varianti del bubblesort

- Infatti, per la proprietà transitiva della relazione d'ordine $<$ si ha che:

  se $a_k < a_{k+1}$ e $a_{k+1} < a_{k+2}$ allora $a_k < a_{k+2}$

- **Esercizio1**. Costruire una variante di bubblesort che tenga conto che se non si fanno scambi l'array è ordinato.

  Suggerimento. Utilizzare una variabile logica **`ordinato`** da inizializzare a **`falso`** e da inserire nel predicato del ciclo esterno: se non ci sono scambi **`ordinato`** deve essere **`vero`**.

Varianti del bubblesort

- **Esercizio2**.
- Costruire una variante che tenga conto della posizione dell'**ultimo scambio**.
- Esempio.

  a = (3, 1, 2, 5, 7, 8, 9, 12, 34, 35)

  Nel ciclo interno si effettuano gli scambi:

  (3, 1) diventa (1, 3)

  (3, 2) diventa (2, 3)

  i successivi confronti: (3, 5), (5, 7), (7, 9) , ecc. non comportano altri scambi.

Varianti del bubblesort

- Memorizzando la posizione dell'ultimo scambio non si "guarda" più la parte di array già ordinato.
- Suggerimento. Chiamare: **`inizio`** e **`fine`** gli estremi degli indici che limitano la parte di array da ordinare, **`sc`** una variabile che memorizza la posizione dell'ultimo scambio; inizializzare **`sc`** con il valore di **`inizio.`**
- Quando **`inizio`** = **`fine`** significa che l'array è ordinato.

Complessità delle varianti del bubblesort

- Per entrambe queste varianti la complessità nel caso favorevole è inferiore a quella del caso senza la variante.
- Infatti, se si fornisce un array già ordinato, dopo la prima iterazione il ciclo termina
  - nel primo caso con ordinato=vero
  - nel secondo caso con inizio=fine
- Pertanto il caso **favorevole** è $\Omega(n)$.
- Il caso peggiore è sempre $O(n^2/2)$.

Esercizio

- Si calcoli la complessità dell'algoritmo seguente, considerando i casi a>b, a=b, a<b; specificando il numero di assegnazioni e di confronti nei tre casi.
- Si supponga n>0.
- Quale sarebbe il valore di a e b se si stampassero a e b alla fine della struttura iterativa "mentre"?

Esercizio

```
Algoritmo
definizione variabili   a, b, n, i, k     intero
                        continua  logico
 acquisire valori per a, b, n
 continua ← vero
 i ← 0
 mentre i ≠ n e continua eseguire
    i ← i+1
      se a < b
         allora  a ← a-1
```

Esercizio

```
    altrimenti  se  a == b
                  allora
                     per  k da 1 a n  eseguire
                            a ← a+1
                            b ← b+1
                      //fineper
                   altrimenti  continua ← falso
                //finese
     //finese
//finementre
```

Costanti

L'uso delle costanti

- Questo programma effettua un cambio di valuta

```
int main(){//cambio valuta
     double dollaro = 2.35;
     double euro = dollaro * 0.71;
          cout<<dollaro<<" dollari =
"<<euro
                    <<" euro \n";
    }
```

- Un utente potrebbe chiedersi quale sia il **significato** del numero **0.71** usato nel programma, ma anche se questo **valore** non **varierà** nel tempo.

L'uso delle costanti

- Così come si usano nomi simbolici descrittivi per le variabili, è opportuno assegnare **nomi simbolici** anche alle **costanti** utilizzate nei programmi.
- Si hanno dei vantaggi:
  1. si aumenta la **leggibilità**
  2. si evitano molte **sostituzioni**, se il valore della costante deve cambiare
  3. non lo si **confonde** con altri valori.

L'uso delle costanti

1. Aumenta la **leggibilità**: chiunque usi il nostro programma capisce che **0.71** è il fattore di conversione

```
int main(){
      const double EURO_DOLLARO = 0.71;
      double dollaro = 2.35;
      double euro = dollaro*EURO_DOLLARO;
           cout<<dollaro<<" dollari =
"<<euro
                    <<" euro \n";
   }
```

L'uso delle costanti

2. Si evitano molte **sostituzioni**: il valore può cambiare e lo dobbiamo cambiare in ogni istruzione in cui viene usato:

```
int main(){
      const double EURO_DOLLARO = 0.90;
      double dollaro1 = 2.35;
      double euro1 = dollaro1*EURO_DOLLARO;
      double dollaro2 = 3.45;
      double euro2 = dollaro2*EURO_DOLLARO;
           //cout<<. . .
  }
```

L'uso delle costanti

3. Si può **confondere** con altri valori:

```
int main(){
      const double EURO_DOLLARO = 0.75;
      double dollaro1 = 2.35;
      double euro1 = dollaro1*EURO_DOLLARO;
      double costocaffe = 0.90;
          // cout<<. . .
  }
```

Definizione di costante

- Sintassi.
  (par 13.4.1)

```
const nometipo NOME_COSTANTE =
                    espressione;
```

- **NOME_COSTANTE** è il nome che vogliamo dare alla costante.
- Spesso il nome delle costanti viene scritto usando tutte le lettere maiuscole (es. EURO_DOLLARO componendo due nomi con la sottolineatura); ci si deve ricordare che il linguaggio distingue le maiuscole dalle minuscole.

Definizione di costante

- Le costanti sono identificate dalla parola chiave **`const`.**
- In fase di **definizione** si deve specificare il **tipo** della costante e le si attribuisce un **valore** (espressione): tale valore **non può più essere modificato**.
- Il tentativo di modificare il valore della costante, dopo l'inizializzazione, produce un errore (semantico) durante la compilazione.

Definizione di costante

- Un uso frequente delle costanti si ha nella definizione degli array: quando si definisce un array si deve sempre indicare, tramite una costante, il numero massimo delle componenti: il compilatore riserva in memoria uno spazio per quel numero di componenti. Per aumentare la leggibilità, permettere facili sostituzioni e non confondere con altri valori si può definire l'array nel modo seguente:

```
const int dim = 100;
double a[dim];
```