Programmazione I

Il Linguaggio C

Ordinamento

Daniel Riccio
Università di Napoli, Federico II

24 novembre 2021

Sommario

- Argomenti
  - Ordinamento
  - Metodi di ordinamento
  - Complessità dei metodi di ordinamento

Il problema dell’ordinamento

Quello di ordinare in modo crescente o decrescente dei numeri, o delle parole in ordine alfabetico diretto o inverso, è uno dei problemi più frequenti della programmazione.

Formalmente esso si enuncia in questi termini:

dato il vettore **`a`** *di* **`n`** *componenti* **`a[0], ... , a[n-1]`** *trasformarlo in un vettore ordinato in modo non decrescente, cioè tale che risulti* **`a[i] <= a[i+1]`** *per qualsiasi valore di* **`i`**.

Per ordinare dei dati esistono due tecniche principali, dette rispettivamente ordinamenti interni e ordinamenti esterni.

Gli *ordinamenti interni* si usano quando la lista dei dati non è troppo lunga e può essere memorizzata per intero nella memoria del computer, di solito in un vettore.

Gli *ordinamenti esterni* si usano per insiemi di dati molto grandi, memorizzati in file su dischi esterni o su nastri, che non conviene caricare nella loro interezza nella memoria del computer.

Il problema dell'ordinamento

Il primo caso (ordinamento di un vettore) corrisponde a ordinare le carte di un mazzo disponendole su un tavolo, in modo che siano tutte visibili e utilizzabili contemporaneamente da chi riordina.

Il secondo caso (ordinamento di file) corrisponde invece a ordinare le carte disponendole in mucchietti o pile, in modo che solo la carta in cima a ogni pila sia visibile e utilizzabile.

Funzioni di complessità

La tabella seguente illustra i valori con cui crescono alcune funzioni di **n**.

n	lg n	$n^{7/6}$	n lg n	n^2
1	0	1	0	1
16	4	25	64	256
256	8	645	2,048	65,536
4,096	12	16,384	49,152	16,777,216
65,536	16	416,128	1,048,565	4,294,967,296
1,048,576	20	10,568,983	20,971,520	1,099,511,627,776
16,777,216	24	268,435,456	402,653,183	281,474,976,710,656

Complessità e tempo

Se i valori nella tabella rappresentano microsecondi, allora per processare 1.048.476 elementi,

- un algoritmo **O**(**lg n**) può impiegare 20 microsecondi
- un algoritmo **O**($\mathbf{n}^{1.25}$) può impiegare 33 secondi
- un algoritmo **O**($\mathbf{n}^2$) può impiegare fino a 12 giorni.

Come abbiamo visto, nella ricerca binaria un confronto in più consente di ricercare fra un numero doppio di valori.

Dato che la funzione $\mathbf{lg_2 n}$ cresce di **1** quando **n** raddoppia, la ricerca binaria è un algoritmo di complessità **O**($\mathbf{lg_2 n}$) o anche, come si dice, caratterizzato da un fattore di crescita **O**($\mathbf{lg_2 n}$).

Essa è assai più veloce della ricerca lineare, che ha complessità **O**(**n**).

In seguito vedremo una stima della complessità in termini di tempo per ogni algoritmo, usando la notazione **O**-grande.

Algoritmi di ordinamento

Un'altra classificazione degli algoritmi di ordinamento si basa sulla loro efficienza o economia di tempo. Una buona misura dell'efficienza si ottiene contando numeri di confronti tra chiavi e di movimenti (trasposizioni) necessari per il riordino.

Queste quantità sono funzioni del numero **`n`** di elementi da ordinare. Anche se buoni algoritmi di ordinamento richiedono un numero di confronti dell'ordine di **`n log n`**, vedremo dapprima alcune tecniche semplici e ovvie, chiamate *metodi diretti*, che richiedono un numero di confronti dell'ordine di n^2.

Di seguito vedremo i seguenti algoritmi:

- Selection Sort
- Insertion Sort
- Bubble Sort

Metodi di ordinamento

Metodi diretti:

– Tecniche semplici ed ovvie
– Adatti ad illustrare le caratteristiche dei maggiori principi di ordinamento
– Facili da capire e da realizzare
– Complessità dell'ordine di n^2:
  - Rapidi per n abbastanza piccolo
  - Inutilizzabili per n grande

Metodi avanzati:

– Tecniche più complesse e meno intuitive:
  - A differenza dei metodi diretti, che spostano un elemento di una posizione ad ogni passo elementare, tali metodi si basano sul principio di spostare gli elementi per distanze maggiori con un unico salto

– Richiedono, generalmente, un numero di confronti nell'ordine di $n \cdot \log n$

Metodi di ordinamento

Alcuni dei metodi diretti sono:

–Ordinamento per selezione

- Altresì detto ordinamento per minimi (o massimi) successivi

–Ordinamento per inserimento

- Basato sul concetto dell'*inserzione ordinata*

–Ordinamento per scambi

- Altresì noto come *bubble sort*

Selection sort

Lista originaria →

9	8	4	5	6	2	3	7	1	0

ciclo	min	scambio										
1	0	a_{10}<->a_1	0	8	4	5	6	2	3	7	1	9

ciclo	min	scambio										
2	1	a_9<->a_2	0	1	4	5	6	2	3	7	8	9

ciclo	min	scambio										
3	2	a_6<->a_3	0	1	2	5	6	4	3	7	8	9

....

ciclo	min	scambio										
6	5	a_7<->a_6	0	1	2	3	4	5	6	7	8	9

A questo punto, pur essendo il vettore ordinato, l'algoritmo prosegue fino a:

ciclo	min	scambio										
9	8	nessuno	0	1	2	3	4	5	6	7	8	9

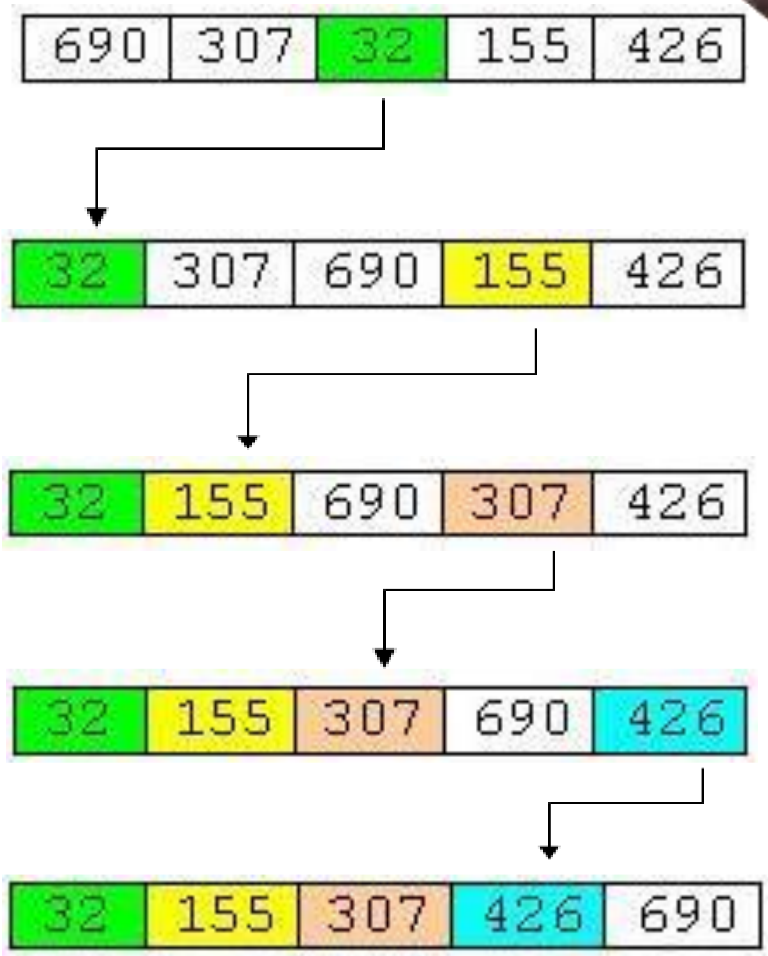
Selection sort

Nel 1° passo si seleziona il numero **32** e lo si scambia con il 1° elemento dell'elenco.

Nel 2° passo si seleziona il numero **155** tra il 2° e il 5° elemento dell'elenco riordinato e lo si scambia con il 2°.

Nel 3° passo si seleziona il numero **307** tra il 3° e il 5° elemento dell'elenco e lo si scambia con il 3°.

Finalmente, nel 4° e ultimo passo si seleziona il restante valore minimo e lo si scambia con il 4° elemento.



Selection sort

Si applica al caso di una lista di **n** elementi memorizzati in un array

Ha per obiettivo il riordinamento degli elementi mediante spostamento fisico

Algoritmo

- Prende in esame la generica sotto-lista **a**[**i**]…**a**[**n**]
- Determina l'elemento minimo **min** e la sua posizione **jmin**
- Esegue lo scambio tra **a**[**i**] ed **a**[**jmin**]
- Il procedimento viene ripetuto per le successive sotto-liste, cioè per i=1..**n**-1

Selection sort

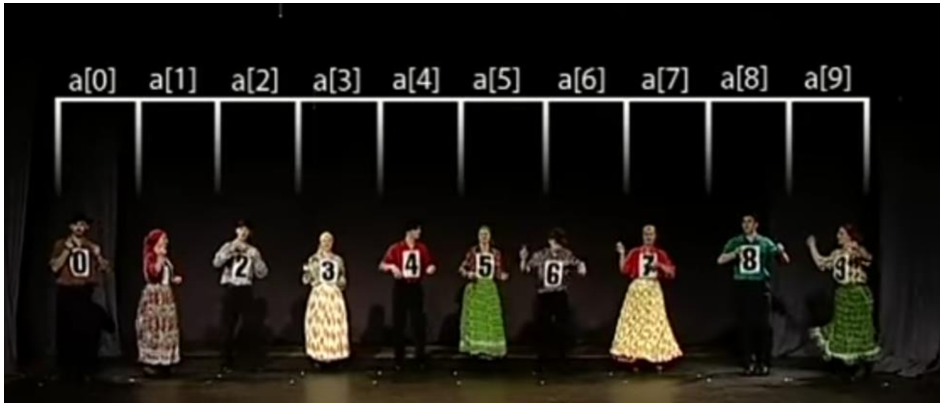
```
#include <stdio.h>

void Swap(int *a, int *b);
void SelectSort(int A[], int n);

int V[] = {72, 23, 12, 5, 0};

int main(int argc, char *argv[])
{
   int i=0;
   int n=5;

   printf("Vettore non ordinato: ");
   for(i=0; i<n; i++)
      printf("%d ", V[i]);

   SelectSort(V, n);

   printf("Vettore ordinato: ");
   for(i=0; i<n; i++)
      printf("%d ", V[i]);

   return 0;
}
```

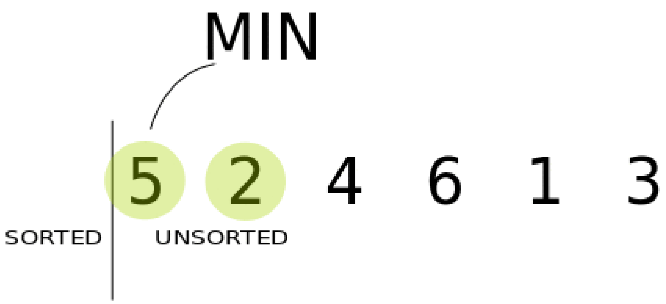
```
void SelectSort(int A[], int n)
{
 int i, j, min, t;

 for(i=0; i<n; i++){
   min = i;

   for (j = i; j < n; j++)
    if (A[j] < A[min])
     min = j;

   Swap(&A[min],&A[i]);
  }
}
```

```
void Swap(int *a, int *b){

   int temp;

   temp = *a;
   *a = *b;
   *b = temp;
}
```

Selection sort

Select-sort with Gypsy folk dance





Link

https://www.youtube.com/watch?v=0-W8OEwLebQ&list=PLOmdoKois7_FK-ySGwHBkltzB11snW7KQ&index=5

Insertion sort

Si applica al caso di una lista di **n** elementi memorizzati in un array

- Ha per obiettivo il riordinamento degli elementi mediante spostamento fisico
- L'algoritmo utilizzato si basa sull'operazione di **push** di un elemento **elem** in un vettore (utilizzata per costruire liste ordinate):
  - Ricerca della posizione **i** del vettore ove inserire **elem** (primo **V**[i]>**elem**)
  - Spostamento in avanti di un posto di tutti gli elementi **V**[j], j=i..**nelem**, **nelem** numero di elementi presenti in **V**
  - Incremento di **nelem** di un'unità
  - Inserzione di **elem** in **V**
  - Tale sequenza di operazioni, qualora applicata ad un array pre-esistente, consente di riordinarlo:
    - Al ciclo **i**-esimo viene prelevato **a**[i], che trova la sua corretta posizione nell'ordinamento in uno dei posti **a**[1],…,**a**[i], mentre gli altri elementi ancora da ordinare restano nella loro posizione iniziale

Insertion sort

Lista originaria →

0	8	1	9	2	3	4	7	6	5

elem	posiz
0	1

0	8	1	9	2	3	4	7	6	5

elem	posiz
8	2

0	8	1	9	2	3	4	7	6	5

elem	posiz
1	2

0	1	8	9	2	3	4	7	6	5

....

elem	posiz
4	5

0	1	2	3	4	8	9	7	6	5

Infine:

elem	posiz
5	6

0	1	2	3	4	5	6	7	8	9

Insertion sort

Array a[0]a[1]..a[n-1]

Algoritmo Caso a) → Inserzione ordinata di elementi:

Si inserisce a[0]
Per i=1..n-1
{
  si cerca il primo elemento a[j]>a[i] nella sottolista a[0]..a[i-1]
  l'elemento **nuovo** va aggiunto in posizione j
  gli elementi a[i-1]..a[j] vanno spostati a destra di una posizione
  l'elemento nuovo va assegnato ad a[j] : a[i] →a[j]
}

Algoritmo Caso b) → Riordinamento di un vettore preesistente

Come nel caso precedente...
Al ciclo i-mo viene prelevato a[i]
a[i] va posizionato al posto giusto in a[0]..a[i]
gli elementi ancora da ordinare restano nella posizione iniziale

Insertion sort

```
#include <stdio.h>

void InsertionSort(int A[], int n);

int V[] = {72, 23, 12, 5, 0};

int main(int argc, char *argv[])
{
   int i=0;
   int n=5;

   printf("Vettore non ordinato: ");
   for(i=0; i<n; i++)
      printf("%d ", V[i]);

   InsertionSort(V, n);

   printf("Vettore ordinato: ");
   for(i=0; i<n; i++)
      printf("%d ", V[i]);

   return 0;
}
```

```
void InsertionSort(int A[], int n) {

   int i,j,k;
   int temp;

   for(i=1;i<n;i++) {
     temp=A[i]; // Salva V[i] in temp

// Ricerca il I elemento > di V[i] in V[0]..V[i-1]
     j=0;
     while((A[j] < A[i]) && (j < i))
       j++;

     // V[i] va inserito in posizione j...
     // Prima, sposta V[j]..V[i-1]in avanti
     for (k=i-1;k>=j;k--)
       A[k+1]=A[k];

     // Inserisci temp in V[j]
     A[j]=temp;
   }
}
```

Insertion sort

Insert-sort with Romanian folk dance

6 5 3 1 8 7 2 4

Link

https://www.youtube.com/watch?v=EdIKIf9mHk0&list=PLOmdoKois7_FK-ySGwHBkltzB11snW7KQ&index=1

Insertion sort - complessità

n-1 inserimenti

– L'**i**-esimo inserimento richiede **i**+1 operazioni (tra inserzione e spostamenti):

- Se inserisco un elemento al posto **k**, avrò **k** confronti e (**i**-**k**) spostamenti, più l'inserimento:
  + # operazioni: **k** + (**i**-**k**) +1 = **i**+1, **i**=2..**n**
  + # medio di operazioni per inserimento: (**min**+**max**)/2 = [3+(**n**+1)]/2=(**n**+4)/2

Complessità:

– **C** = (**n**-1)*(**n**+4)/2

- …dell'ordine di n^2

Bubble sort

Si applica al caso di una lista di **n** elementi memorizzati in un array

Ha per obiettivo il riordinamento degli elementi mediante spostamento fisico

L'algoritmo utilizzato:

– Prevede l'ordinamento attraverso scambi tra coppie successive
  - Si esamina la prima coppia (**a**[1],**a**[2]) e si esegue uno scambio nel caso non sia ordinata
  - Si prosegue in maniera analoga con la coppia successiva (**a**[2],**a**[3]) e così via fino alla coppia (**a**[n-1],**a**[n])
  - Al termine del ciclo, l'elemento massimo occupa la posizione **n**, alla quale sarà pervenuto attraverso un certo numero di scambi (da qui il paragone con la bolla che **gorgoglia** verso l'alto)

– Il procedimento viene ripetuto con riferimento alla sotto-lista **a**[1]…**a**[n-1] e così via fino a pervenire alla sotto-lista **a**[1],**a**[2]

L'ultimo scambio effettuato in un ciclo determina la posizione a partire dalla quale la lista è ordinata:

– Registrando la posizione **u** dove tale scambio è stato effettuato, è possibile limitare il successivo ciclo di scansione alla sotto-lista **a**[1]…**a**[u]

Bubble sort

Lista originaria →

9	8	4	5	6	2	3	7	1	0

Ciclo # 1:

coppia	relazione	scambio									
1	9>8	a_1<->a_2	8	9	4	5	6	2	3	7	1	0

coppia	relazione	scambio									
2	9>4	a_2<->a_3	8	4	9	5	6	2	3	7	1	0

....

coppia	relazione	scambio									
9	9>0	a_9<->a_{10}	8	4	5	6	2	3	7	1	0	9

Bubble sort

Ciclo # 2:

coppia	relazione	scambio										
1	8>4	a_1<->a_2	4	8	5	6	2	3	7	1	0	9

coppia	relazione	scambio										
8	8>0	a_8<->a_9	4	5	6	2	3	7	1	0	8	9

Ciclo # 3:

coppia	relazione	scambio										
3	6>2	a_3<->a_4	4	5	2	6	3	7	1	0	8	9

coppia	relazione	scambio										
4	6>3	a_4<->a_5	4	5	2	3	6	7	1	0	8	9

coppia	relazione	scambio										
6	7>1	a_6<->a_7	4	5	2	3	6	1	7	0	8	9

coppia	relazione	scambio										
7	7>0	a_7<->a_8	4	5	2	3	6	1	0	7	8	9

Bubble sort

Ciclo # 4, al termine:

coppia	relazione	scambio									
		a_6<->a_7	4	2	3	5	1	0	6	7	8	9

....

Ciclo # 8, al termine:

coppia	relazione	scambio									
		a_2<->a_3	1	0	2	3	4	5	6	7	8	9

....

Ciclo # 9

coppia	relazione	scambio									
		a_1<->a_2	0	1	2	3	4	5	6	7	8	9

Bubble sort

Nel caso esaminato, la disposizione iniziale dei dati non è favorevole all'algoritmo:

- Lo zero, che è il minimo ed è situato in ultima posizione, trova la sua collocazione finale solo al termine dell'ultimo ciclo

In generale, nel caso in cui in un determinato ciclo non ci siano scambi, l'algoritmo può terminare precocemente

Bubble sort

Worst-case (lista contro-ordinata):

- (n-1) cicli
- Per ogni ciclo (n-i) confronti, i=1..(n-1)
  - **Min** = n-(n-1)=1
  - **Max** = n-1
  - # medio di confronti: (**min**+**max**)/2 = [1+(n-1)]/2=n/2
- # totale confronti:
  - # cicli * # confronti/ciclo = (n-1)*n/2

Best-case (lista già ordinata):

- (n-1) confronti nell'unico ciclo

Average-case (distribuzione uniforme degli elementi)

- Metà del caso peggiore:
  - [n*(n-1)]/4

Ne deriva che la complessità è dell'ordine di n^2

NB: Il bubble sort impone, in media, un numero di spostamenti (scambi) molto elevato; considerando che lo scambio di due elementi è generalmente un'operazione molto più costosa del confronto tra due chiavi, se ne deduce che tale tecnica è decisamente inferiore all'ordinamento per selezione (è in effetti il metodo meno efficiente)

Bubble sort

Array a[0]a[1]..a[n-1]

Algoritmo

Per i=1..n-1

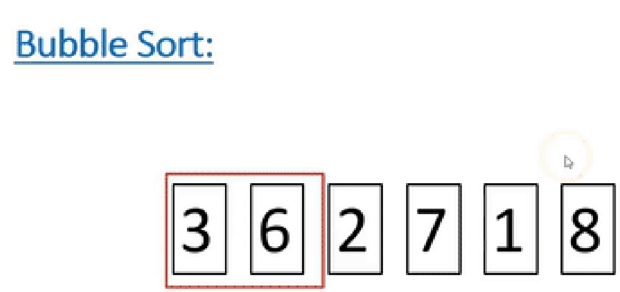
{

si considera la sottolista a[0]..a[n-i]

si effettua una scansione della lista dal primo all'ultimo elemento

se un elemento precede un elemento <, lo si scambia col successivo

}



Bubble sort

Versione iterativa:

```
void BubbleSort(int V[], int n) {
    int i,j;
    int nswaps=0; // numero di scambi
    int ncycles=0; // numero di iterazioni

    for(i=1;i<n;i++) {
        ncycles++; // Analizza il sottoinsieme V[0]..V[n-i]
        for(j=0;j<n-i;j++) {
            if(V[j]>V[j+1]) {
                nswaps++;
                swap(&V[j],&V[j+1]);
            }
        }
    }
    printf("Sono stati eseguiti %d scambi\n", nswaps);
    printf("Sono stati eseguiti %d scambi\n", ncycles);
}
```

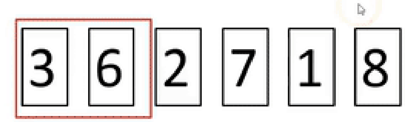
Bubble sort – versione migliorata

```
void BubbleSort(int V[], int n) {

   int i,j;
   int nswaps=0; // numero di scambi
   int ncycles=0; // numero di iterazioni
   boolean scambi=TRUE; // TRUE se è stato effettuato almeno
   i=1; // uno scambio
   while((i<=n-1) && (scambi==TRUE)) {
      scambi=FALSE;
      ncycles++;
      for(j=0;j<n-i;j++){ // Analizza il sottoinsieme V[0]..V[n-i]
         if(V[j]>V[j+1]) {
            nswaps++;
            swap(&V[j],&V[j+1]);
            scambi=TRUE;
         }
      }
      i++;
   }
   printf("Sono stati eseguiti %d scambi\n", nswaps);
   printf("Sono stati eseguiti %d scambi\n", ncycles);
}
```

Bubble sort

Bubble-sort with Hungarian folk dance

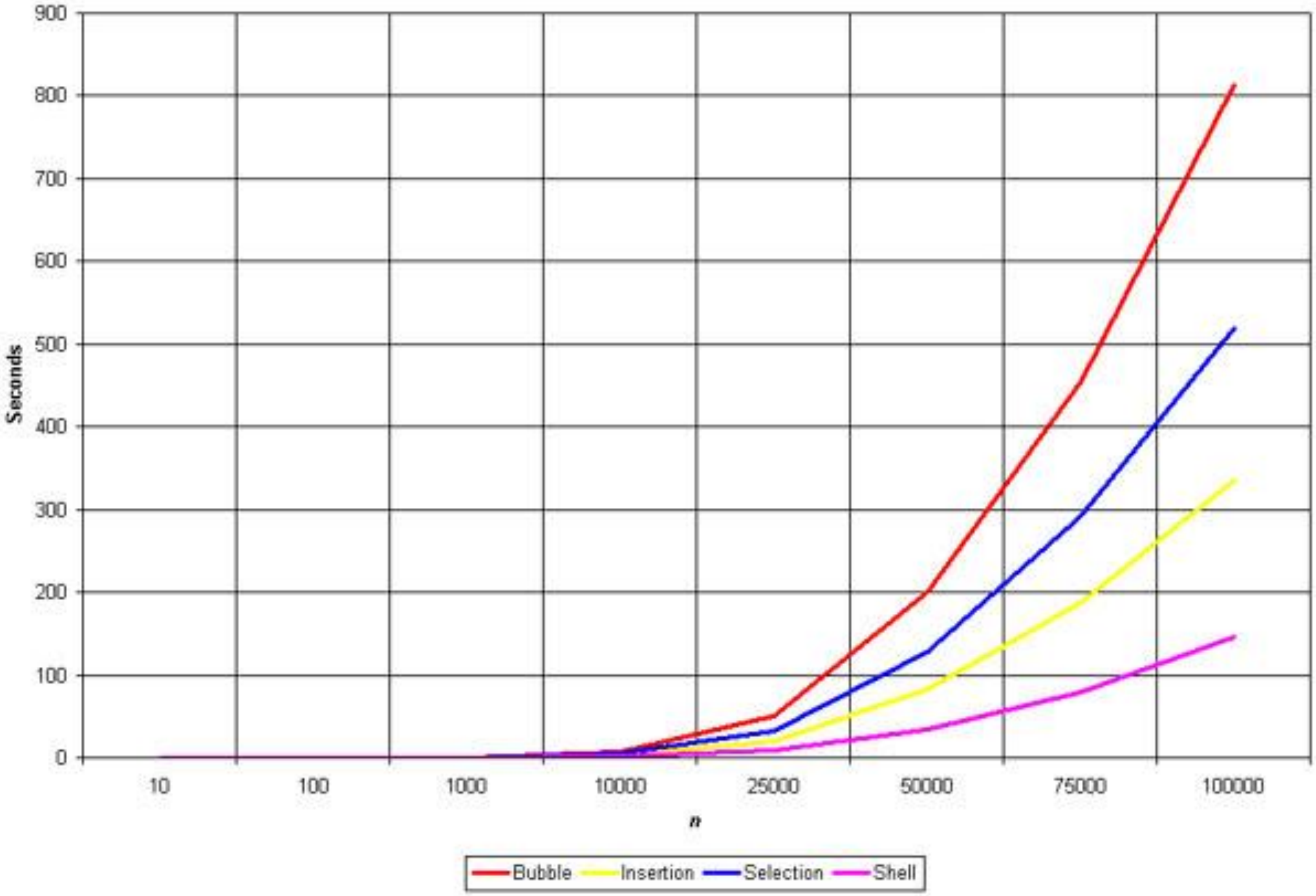
Bubble Sort:



Link

https://www.youtube.com/watch?v=semGJAJ7i74&list=PLOmdoKois7_FK-ySGwHBkltzB11snW7KQ&index=2

Complessità



Problemi di ordinamento

Supponiamo di avere una enorme mole di dati rappresentati da interi (es. codici identificativi di persone). Inoltre, si accede ai dati solo in lettura (i.e. non si aggiungono o cancellano valori) mediante operazioni di ricerca.

Supponiamo di effettuare **M** ricerche su **N** valori

Ricerca Lineare

Numero di operazioni: O(**M** × **N**)

Cerchiamo una soluzione
che richieda al più O(**N** + **M** × log(**N**))

Ordinamento più ricerca binaria

Numero di operazioni:

- Ordinamento O(**N**)
- Ricerca binaria O(**M** × log(**N**))

M ricerche binarie ciascuna di costo log(**N**)

M, **N**	**M** × **N**	**N** + **M** × log(**N**)
M=10, **N**=16	160	56
M=100, **N**=256	25 600	1 056
M=1000 **N**=4096	4 096 000	16 096
M=10000 **N**=65536	655 360 000	225 536

Esercizi

Dato il vettore di interi V[]={34, 12, 65, 21, 89, 3}, si mostrino le diverse configurazioni intermedie del vettore prodotte dall’insertion sort:

t_0: V[] = {34, 12, 65, 21, 89, 3}
t_1: V[] = {12, 34, 65, 21, 89, 3}
t_2: V[] = {12, 34, 65, 21, 89, 3}
t_3: V[] = {12, 21, 34, 65, 89, 3}
t_4: V[] = {12, 21, 34, 65, 89, 3}
t_5: V[] = {3, 12, 21, 34, 65, 89}

Esercizi

Dato il vettore di interi V[]={34, 12, 65, 21, 89, 3}, si mostrino le diverse configurazioni intermedie del vettore prodotte dal selection sort:

t_0: V[] = {34, 12, 65, 21, 89, 3}
t_1: V[] = {3, 12, 65, 21, 89, 34}
t_2: V[] = {3, 12, 65, 21, 89, 34}
t_3: V[] = {3, 12, 21, 65, 89, 34}
t_4: V[] = {3, 12, 21, 34, 89, 65}
t_5: V[] = {3, 12, 21, 34, 65, 89}

Esercizi

Dato il vettore di interi V[]={34, 12, 65, 21, 89, 3}, si mostrino le diverse configurazioni intermedie del vettore prodotte dal bubble sort:

t_0: V[] = { 34, 12, 65, 21, 89, 3 }
t_1: V[] = { 12, 34, 65, 21, 89, 3 }
t_2: V[] = { 12, 34, 65, 21, 89, 3 }
t_3: V[] = { 12, 34, 21, 65, 89, 3 }
t_4: V[] = { 12, 34, 21, 65, 89, 3 }
t_5: V[] = { 12, 34, 21, 65, 3, 89 }
t_6: V[] = { 12, 34, 21, 65, 3, 89 }
t_7: V[] = { 12, 21, 34, 65, 3, 89 }
t_8: V[] = { 12, 21, 34, 65, 3, 89 }
t_9: V[] = { 12, 21, 34, 3, 65, 89 }
t_{10}: V[] = { 12, 21, 34, 3, 65, 89 }
t_{11}: V[] = { 12, 21, 34, 3, 65, 89 }
t_{12}: V[] = { 12, 21, 3, 34, 65, 89 }
t_{13}: V[] = { 12, 21, 3, 34, 65, 89 }
t_{14}: V[] = { 12, 3, 21, 34, 65, 89 }
t_{15}: V[] = { 3, 12, 21, 34, 65, 89 }