Fondamenti di Informatica

Corso di Laurea in Ingegneria Elettronica

Corso di Laurea in Ingegneria Informatica

Algoritmi di Ricerca e Ordinamento: Teoria

La presente dispensa è stata prodotta dal Prof. V. Carchiolo e dal Prof. S. Cavalieri ed è ufficialmente adottata nei corsi di Fondamenti di Informatica per Ingegneria Informatica/Elettronica

Algoritmi di Ricerca e Ordinamento

1. Introduzione 3

2. Algoritmi di Ricerca 4

2.1 RICERCA SEQUENZIALE 5

2.3 RICERCA BINARIA 9

3. Ordinamento 11

3.1 Algoritmo di Ordinamento naïve sort 13

3.2 Algoritmo di Ordinamento bubble sort 15

3.3 Algoritmo di Ordinamento shaker sort 19

3.4 Ordinamento insert sort 20

3. 5 Algoritmo di Ordinamento QuickSort 24

Esempi del QuickSort *26*

3.6 Algoritmo di Ordinamento merge sort 30

4. Confronto fra i diversi algoritmi 33

1. Introduzione

La ricerca di un elemento in una sequenza di informazioni e l'ordinamento di una sequenza di informazioni sono due importanti problemi. La ricerca di un elemento in una sequenza di informazioni di tipo omogeneo consiste nel verificare l'esistenza di un dato elemento in una collezione di elementi mentre l'ordinamento di una sequenza di informazioni consiste nel disporre le stesse informazioni in modo da rispettare una qualche relazione d'ordine di tipo lineare; ad esempio una relazione d'ordine "minore o uguale" dispone le informazioni in modo "non decrescente".
L'ordinamento è molto importante in quanto permette di ridurre notevolmente i tempi di ricerca di una informazione nell'ambito di una sequenza. Nel caso in cui tale sequenza sia ordinata secondo una relazione d'ordine, è possibile sfruttare la stessa relazione d'ordine per effettuare la ricerca.
In genere le informazioni che compongono la sequenza su cui effettuare ordinamenti e ricerche possono essere piuttosto complesse e possono essere pensati come costituiti da una ***chiave*** e da un gruppo di ***dati accessori***.
Ad esempio se consideriamo la sequenza delle informazioni che costituiscono una agenda telefonica ciascuna informazione della sequenza conterrà, ad esempio, le seguenti informazioni:

- nome e cognome
- indirizzo
- numero telefonico
- E_mail

In una tale tipologia di informazione "nome e cognome" possono rivestire il ruolo di chiave mentre le altre informazioni aggiuntive, quali l'indirizzo, rivestono il ruolo di dati accessori.
Nella descrizione degli algoritmi di ricerca e di ordinamento, i dati accessori vengono tipicamente ignorati perché non sono utilizzati nella ricerca e non sono oggetto dell'organizzazione dei dati. La chiave essendo l'insieme di valori che identifica un elemento della sequenza riveste un ruolo fondamentale poiché in un algoritmo di ricerca sarà l'informazione utilizzata per la scoperta dell'elemento e in un algoritmo di ordinamento sarà l'informazione rispetto alla quale ordinare i dati.
La chiave può avere un qualsiasi tipo di dato e può anche essere formata da un insieme di valori: dipende dal tipo di insieme che si vuole rappresentare. La chiave può essere ***univoca*** in tutto l'insieme di elementi, oppure ***multipla*** qualora sia consentito di condividerla tra più elementi distinti. Ad esempio se si considera la sequenza degli studenti che seguono un corso universitario l'informazione "numero di matricola" sarà una chiave univoca (anche detta chiave primaria) poiché non esistono due studenti con lo stesso numero di matricola mentre la chiave "nome e cogniome" potrebbe essere una chiave multipla poiché esiste la possibilità che in uno stesso corso vi siano più studenti con lo stesso nome e cognome.

Le singole informazione della sequenza possono essere memorizzate su un tipo struct composto da differenti campi. Le modalità di ricerca e ordinamento dipendono dalle modalità di allocazione della sequenza. In particolare possiamo distinguere il caso di utilizzo di strutture dati allocate in memoria centrale dell'elaboratore o di strutture allocate in memoria di massa.

2. Algoritmi di Ricerca

Un **algoritmo di ricerca** è un algoritmo che permette di trovare un elemento avente determinate caratteristiche all'interno di un insieme di elementi. Quindi con il termine Ricerca si intende il procedimento per localizzare una particolare informazione in un elenco di dati.

Per esempio:

1. cercare il numero di Giuseppe Pappalardo nell'elenco telefonico
2. cercare il 4 di mazze in un mazzo di carte

Dai precedenti esempi si intuisce che il problema della ricerca dipende da come le informazioni sono organizzate. Il problema della ricerca può quindi essere formulato come

Data una collezione di elementi trovare se esiste l'elemento di valore E

Come già introdotto, gli elementi dell'insieme sono caratterizzati da una *chiave* e da un gruppo di *dati accessori.* Ad esempio se consideriamo il problema "cercare il numero di Giuseppe Pappalardo nell'elenco telefonico", il nome e cognome (Giuseppe Pappalardo) rivestono il ruolo di chiave mentre le altre informazioni aggiuntive, quali l'indirizzo, rivestono il ruolo di dati accessori.

La chiave può avere un qualsiasi tipo di dato e può anche essere formata da un insieme di valori: dipende dal tipo di insieme che si vuole rappresentare. La chiave può essere *univoca* in tutto l'insieme di elementi, oppure *multipla* qualora sia consentito di condividerla tra più elementi distinti. In questo secondo caso è fondamentale specificare il corretto comportamento di un algoritmo di ricerca. Bisogna infatti decidere se sarà restituito il primo elemento dotato di una certa chiave, l'ultimo, uno qualsiasi o anche tutti.

Nel seguito noi tratteremo il problema della ricerca nel caso in cui non ci siano nell'insieme più dati con la stessa chiave (chiave unica) e supporremo di operare con dati allocati in memoria centrale

Per le nostre implementazioni supponiamo di aver definito un generico tipo Tipobase che rappresenterà il tipo degli elementi della sequenza ed è esso stesso tipo della nostra chiave e supporremo, inoltre, che sul tipo Tipobase siano definiti gli operatori di confronto.

Supporremo infine che i nostri dati siano in numero di N e che siano immagazzinati all'interno di un array V.

La generica funzione di ricerca avrà quindi come parametri l'array V contenente i dati su cui effettuare la ricerca (o meglio il puntatore al primo elemento dell'array), un intero N che rappresenta il numero di elementi presenti nell'array e un valore **E** di tipo **Tipobase** che rappresenta l'elemento da ricercare.
Avendo fatto l'ipotesi che gli elementi siano memorizzati su un array la funzione di ricerca ritornerà la posizione dell'elemento cercato sull'array, quindi un valore compreso tra 0 e N-1. Nel caso l'elemento E non fosse presente nell'insieme, la funzione di ricerca tornerà il valore -1.
Affronteremo il problema sia per insiemi di elementi nei quali non è presente alcuna organizzazione sia per insieme nei quali gli elementi sono ordinati rispetto alla chiave secondo una relazione di ordinamento crescente (minore, maggiore).
Per il problema della ricerca esistono numerosi algoritmi risolutivi con complessità polinomiale, noi presenteremo nel seguito i seguenti tre algoritmi:

- ricerca sequenziale
- ricerca sequenziale con sentinella
- ricerca binaria (o dicotomica)

Mentre per gli insiemi ordinati possono essere utilizzati tutti e tre i tipi di algoritmi per gli insiemi su cui non è definita alcuna relazione di ordinamento è possibile utilizzare solo il primo e il secondo dei tre algoritmi. La complessità del primo algoritmo come vedremo è indipendente dall'organizzazione.
Per ciascuno degli algoritmi è possibile una implementazione sia iterativa che ricorsiva.

2.1 RICERCA SEQUENZIALE

Se non abbiamo alcuna informazione circa l'ordine degli elementi nell'insieme un metodo per localizzare un particolare elemento è una ***ricerca lineare*** cioè si parte dal primo elemento e si procede esaminando uno per uno tutti gli elementi fino a quando non sono stati letti tutti gli elementi dell'insieme.
L'algoritmo può essere formalizzato effettuando le seguenti operazioni:

1. lettura dal primo elemento del vettore V;
2. confronto della chiave E con ciascuno degli elementi del vettore;
3. La lettura termina quando la chiave E è stata confrontata con tutti gli elementi del vettore (dell'insieme).

L'implementazione iterativa può essere effettuata secondo il seguente codice nel quale utilizzando un ciclo si esegue il confronto dell'elemento E con tutti gli elementi del vettore:

```
int RicercaSequenziale(tipobase V[], tipoBase E, int N){
 int i, pos=-1;
 for (i = 0; i<N; i++)
      if (V[i]==E) pos=i;
 return pos;
}
```

Come precedentemente discusso nella valutazione di un algoritmo si possono distinguere tre diversi casi:

1. caso migliore
2. caso peggiore
3. caso medio

Solitamente la complessità si valuta sul *caso peggiore.* Tuttavia, poiché esso è di norma assai raro, spesso si considera anche il *caso medio*. Nel caso della ricerca sequenziale non c'è differenza tra i tre casi. Per la *ricerca sequenziale* in un array, il costo non dipende dalla posizione dell'elemento cercato ed è pari a O(N).

Va inoltre evidenziato che la complessità dell'algoritmo di ricerca sequenziale, cosi come la sua implementazione, è indipendente dall'organizzazione dei dati visto che tutti gli elementi vengono confrontati con il valore cercato in modo indipendente dalla presenza di un ordinamento o meno dei dati.

Ad esempio, supponendo che di avere il vettore di elementi 10 di tipo int in figura

2	30	61	12	10	21	26	22	38	3

E che il valore cercato sia il numero 26, cioè E=26. L'algoritmo effettuerà 10 confronti del valore E anche se al 6 confronto essendo il risultato del test E == **V[i]** uguale al valore TRUE viene modificato il valore della variabile pos assegnandogli il valore dell'indice corrente (i). I confronti successivi saranno tutti uguali al valore FALSE ma verranno effettuati comunque anche se inutili.

2.2 RICERCA SEQUENZIALE con SENTINELLA

La ricerca si può effettuare in modo più efficiente sfruttando un meccanismo di limitazione dei confronti ad una parte degli elementi del vettore. Il meccanismo da introdurre per limitare il numero di confronti riflette la caratterizzazione dell'organizzazione dei dati. Pertanto, tale meccanismo potrà essere differente nel caso di insiemi di dati non ordinati o ordinati e quindi le performance di un tale tipo di ricerca sarà differente nei due casi.

L'algoritmo può essere formalizzato effettuando le seguenti operazioni:

1. lettura dal primo elemento del vettore V;
2. confronto della chiave E con ciascuno degli elementi del vettore fino a quando è soddisfatto la condizione di terminazione definita in tabella 1.

	Terminazione con successo	Terminazione senza successo
Insieme non ordinato	*i-esimo elemento* = E	quando si è raggiunto l'ultimo elemento del vettore
Insieme ordinato	*i-esimo elemento* = E	*i-esimo elemento* > E

La ricerca termina **con successo** quando l'elemento V[i] considerato ad un certo passo è proprio uguale a E mentre la ricerca termina **con insuccesso** termina nel caso di insiemi non ordinati quando si è raggiunto l'ultimo elemento del vettore mentre nel caso di vettore ordinato quando la chiave viene confrontata con un valore maggiore.

Nel caso di dati non ordinati, l'implementazione iterativa può essere effettuata secondo il seguente codice nel quale utilizzando un ciclo si esegue il confronto dell'elemento E con tutti gli elementi del vettore:

```
int RicercaSentinella (tipobase V[], tipobase E, int N){
 int i;
 for(i = 0; i<N; i++)
      if (V[i] == E ) return i;
 return -1;
}
```

L'implementazione del caso non ordinato utilizza tre casi:

1. L'insieme è vuoto in questo caso l'algoritmo ritorna il valore -1
2. L'elemento è il primo dell'insieme in questo caso l'algoritmo ritorna la posizione nell'array dell'elemento corrente
3. L'algoritmi ritorna il risultato della ricerca sull'insieme privato del primo elemento

```
int RicercaSentinella (tipobase V[], tipobase E, int primo, int N)
{
     if (primo == N) return -1;
     if (E==V[primo]) return primo;
     return RicercaSentinella (V, E, primo+1, N);
    }
```

Nel caso di dati ordinati, l'implementazione iterativa può essere effettuata secondo il seguente codice nel quale utilizzando un ciclo si esegue il confronto dell'elemento E con tutti gli elementi del vettore:

```
    int RicercaSentinella (tipobase V[], tipobase E, int N){
     int i;
     for (i = 0; i<N&&E<=V[i]; i++)
          if (E == V[i]) return i;
     return -1;
    }
```

L'implementazione ricorsiva utilizza tre casi:

1. L'insieme è vuoto in questo caso l'algoritmo ritorna il valore -1
2. L'elemento è il primo dell'insieme in questo caso l'algoritmo ritorna la posizione nell'array dell'elemento corrente
3. L'algoritmo ritorna il risultato della ricerca sull'insieme privato del primo elemento

```
int RicercaSentinella (tipobase V[], tipobase E, int primo, int N){
          if (primo == N) return -1;
          if (E==V[primo]) return primo;
          if (E < V[primo]) return -1;
          return RicercaSentinella (V, E, primo+1, N);
    }
```

Come precedentemente discusso nella valutazione di un algoritmo si possono distinguere tre diversi casi:

- caso *migliore*

- caso *peggiore*
- caso *medio*

Per la *ricerca sequenziale con sentinella* in un array non ordinato, il costo dipende dalla posizione dell'elemento cercato.

- *Caso migliore*: l'elemento è il primo dell'array → un solo confronto
- *Caso peggiore*: l'elemento è l'ultimo o non è presente → N confronti, costo *lineare O*(N)
- *Caso medio*: l'elemento può con eguale probabilità essere il primo (1 confronto), il secondo (2 confronti), ... o l'ultimo (N confronti)

2.3 RICERCA BINARIA

Se la sequenza è ordinata si può effettuare una ricerca più efficiente che mi permette di individuare l'elemento cercato senza dover scandire tutti gli elementi del vettore. L'algoritmo di ***ricerca binaria*** dopo ogni confronto scarta metà degli elementi del vettore su cui si effettua la ricerca **restringendo** il campo di ricerca.

```
int RicercaBinaria (tipobase V[], tipobase E, int inf, int sup)
{
 int m = (inf+sup)/2;
 while (inf<=sup){
      if (E == V[m]) return m;
      if (E > V[m]) inf = m+1;
      else sup = m-1;
 }
 return -1;
}
```

La funzione ricerca_binaria può essere implementata invocando una funzione ricorsiva:

```
int RicercaBinaria (tipobase V[], tipobase E, int inf, int sup) {
     int m = (inf+sup)/2;

     if (inf> sup) return -1;
     if (E == V[m]) return m;
     if (E > V[m]) return RicercaBinaria(V,E, m+1,sup);
     return ric(V,E,inf, m-1);
    }
```

Esempio: Valore Cercato 26

Si consideri il vettore in figura. La cella Gialla rappresenta l'estremo inferiore del vettore, la rosa il valore dell'estremo superiore.

2	3	6	12	16	21	24	26	28	30	36	41	50

Alla prima esecuzione la chiave E viene confrontata con l'elemento di posizione media (cella verde). Quando l'elemento di posizione media è uguale alla chiave k potremo affermare che la ricerca è terminata con successo. Se la chiave E è minore del valore media vuol dire che se presente l'elemento si troverà nella posizioni precedenti del vettore, al contrario se la chiave E è maggiore del valore media vuol dire che se presente l'elemento si troverà nella posizioni successive del vettore.

Nel caso dell'esempio quindi si passa ad analizzare la seguente parte del vettore.

26	28	30	36	41	50

Poi si considera la porzione del vettore

26	28

Ad ogni iterazione o ricorsione l'insieme è dimezzato, il numero di confronti è pari a quante volte un numero **N** può essere diviso per 2 fino a ridurlo a 0.

Per esempio in un vettore di dimensione **N** con $N = 2^h - 1$ l'algoritmo deve compiere $\mathbf{h = log_2(N+1)}$ passi (e quindi confronti) per la ricerca senza successo, nel caso medio il numero è leggermente

inferiore mentre nel caso migliore è 1. Quindi potremo affermare che la complessità dell'algoritmo e pari a O(**$\log_2 N$**).
Ad esempio se dovessimo cercare un elemento in un insieme di 1.000.000 di elementi, nei casi più sfortunati con l'algoritmo di ***Ricerca Sequenziale*** **con Sentinella** dovremmo eseguire circa 1.000.000 confronti, mentre con la ***Ricerca Binaria*** ne dobbiamo effettuare al massimo solamente 21.

3. Ordinamento

L'ordinamento di una sequenza di informazioni consiste nel disporre le stesse informazioni in modo da rispettare una qualche relazione d'ordine di tipo lineare; ad esempio una relazione d'ordine "minore o uguale" dispone le informazioni in modo "non decrescente".
L'ordinamento è molto importante in quanto permette di ridurre notevolmente i tempi di ricerca di una informazione nell'ambito di una sequenza di informazioni. Nel caso in cui tale sequenza sia ordinata secondo una qualche relazione d'ordine, è possibile sfruttare la stessa relazione d'ordine per effettuare la ricerca. Ad esempio sia data una sequenza di informazioni ordinata secondo una relazione d'ordine del tipo "minore o uguale"; l'algoritmo di ricerca procede scandendo la sequenza di elementi, fino a quando o individua l'elemento cercato o trova una informazione maggiore di quella cercata. In questo ultimo caso la ricerca deve essere interrotta in quanto la relazione d'ordine adottata assicura che gli elementi successivi sono senz'altro superiori a quello cercato.
In genere le informazioni che compongono la sequenza e che devono essere ordinate sono piuttosto complesse. Ad esempio, si pensi ad un vettore di elementi di tipo struct, dove ciascuno struct sia composto da differenti campi; in tal caso l'ordinamento viene applicato ad un numero finito di attributi di ciascuna informazione; tali attributi sono dette chiavi. In tal caso la ricerca può trarre vantaggio dall'ordinamento se la ricerca stessa viene effettuata utilizzando le stesse chiavi sulle quali è stata applicata la relazione d'ordine. Generalmente le chiavi di ordinamento vengono scelte in modo da permettere di distinguere tutti gli elementi dell'insieme da ordinare; ad esempio se si dovesse ordinare un archivio anagrafico, la scelta delle sole chiavi "cognome" e "nome" non permetterebbe di distinguere (sia nella fase di ordinamento sia in quella di ricerca), persone con lo stesso cognome e nome.
È possibile dimostrare che il problema dell'ordinamento non può essere risolto con un algoritmo di complessità asintotica inferiore a quella pseudo-lineare. Pertanto la complessità per un algoritmo di ordinamento nel caso migliore è pari a $O(n \cdot \log_2 n)$.
Esistono diverse categorie di algoritmi di ordinamento. Essi possono essere distinti in due classi:

Algoritmi di Ricerca e Ordinamento

- Algoritmi di Ordinamento Interno. Fanno uso di strutture dati allocate in memoria centrale dell'elaboratore. Per tale motivo hanno il principale limite di non permettere l'ordinamento di grosse sequenze di informazioni.
- Algoritmi di Ordinamento Esterno. Tipicamente fanno uso di strutture file per la memorizzazione e le operazioni di ordinamento della sequenza di informazioni da ordinare.

Nel seguito tratteremo esclusivamente il problema degli algoritmi di ordinamento interni.
Come detto tali algoritmi sono caratterizzati dall'uso di strutture dati allocate in memoria RAM per la memorizzazione e l'ordinamento di sequenze di informazioni. Ad esempio essi fanno uso di vettori.
Esistono diversi modi per classificare gli algoritmi di ordinamento interni. La prima classificazione può essere effettuata in base alla complessità di calcolo. La complessità di calcolo si riferisce soprattutto al numero di operazioni necessarie all'ordinamento (principalmente operazioni di confronto e scambio), in funzione del numero di elementi (n) da ordinare:

1. **Algoritmi Semplici di Ordinamento.** Algoritmi che presentano una complessità proporzionale a n^2, dove n è il numero di informazioni da ordinare. Essi sono generalmente caratterizzati da poche e semplici istruzioni.
2. **Algoritmi Evoluti di Ordinamento.** Algoritmi che offrono una complessità computazionale proporzionale a $n \cdot \log_2 n$ (che è sempre inferiore a n^2) . Tali algoritmi sono molto più complessi e fanno molto spesso uso di ricorsione. La convenienza del loro utilizzo si ha unicamente quando il numero n di informazioni da ordinare è molto elevato.

Oltre che per il loro principio di funzionamento e per la loro efficienza, gli algoritmi di ordinamento possono essere confrontati in base ai seguenti criteri:

- Stabilità: un algoritmo di ordinamento `e stabile se non altera l'ordine relativo di elementi dell'array aventi la stessa chiave. Algoritmi di questo tipo evitano interferenze con ordinamenti pregressi dello stesso array basati su chiavi secondarie.
- Sul posto: un algoritmo di ordinamento opera sul posto se la dimensione delle strutture ausiliarie di cui necessita è indipendente dal numero di elementi dell'array da ordinare. Algoritmi di questo tipo fanno un uso parsimonioso della memoria.

Nel seguito verranno presentati i seguenti algoritmi di ordinamento **Principali algoritmi di ordinamento:**

1. ***naïve sort*** (semplice, intuitivo, poco efficiente)
2. ***bubble sort*** (semplice, un po' più efficiente)
3. ***shaker sort*** (semplice, un po' più efficiente)
4. ***insert sort*** (intuitivo, abbastanza efficiente)

5. ***quick sort*** (non intuitivo, alquanto efficiente)
6. ***merge sort*** (non intuitivo, molto efficiente)

Per "misurare le prestazioni" di un algoritmo, conteremo quante volte viene svolto il ***confronto fra elementi dell'array.***

Come per gli algoritmi di ricerca considereremo la seguente struttura dati

tipobase vettore[n];

dove *tipobase* è il generico tipo che caratterizza ciascun elemento del vettore (ad esempio int, float, char, struct, etc.). In altre parole, si è supposto che la sequenza di elementi da ordinare viene memorizzata in un vettore di dimensione n contenente elementi appartenenti ad un generico tipo, detto tipobase.

Nel caso in cui le informazioni da ordinare sono molto complesse (ad esempio siano costituite da uno struct), l'ordinamento verrà effettuato considerando uno o più attributi (campi) di ciascun elemento (ossia si dovranno scegliere delle chiavi).

Supporremo inoltre che su tipobase sia definita una relazione di ordinamento e che siano definiti gli operatori di confronto (==, !=, >, <, etc).

3.1 Algoritmo di Ordinamento naïve sort

L'algortimo **naive sort** anche chiamato per selezione diretta o **SelectionSort**. Tale algoritmo basato su un processo iterativo ha come obiettivo la selezione dell'elemento della sequenza di origine che contiene il valore maggiore (nel caso di ordinamento crescente) e di scambiare tale valore con il valore contenuto nell'ultima posizione, in modo da ridurre la sequenza di origine di un elemento. Considerando un vettore di n elementi l'algoritmo potrebbe essere descritto dal seguente pseudo codice

```
while (<array non vuoto>) {
 <trova la posizione p del massimo>
 if (p<n-1) <scambia v[n-1] e v[p] >
 /* invariante: v[n-1] contiene il massimo */
 <restringi l'attenzione alle prime n-1 caselle dell' array, ponendo n' = n-1 >
}
```

La codifica in C sarà

```
void scambia(tipobase *a, tipobase *b){
 tipobase z;
 z = *a;
```

```
     *a = *b;
     *b = z;
    }
    int trovaPosMax(tipobase v[], int n){
     int i, posMax=0;
     for (i=1; i<n; i++)
          if (v[posMax]<v[i]) posMax=i;
     return posMax;
    }
    void naiveSort(tipobase v[], int n){
     int p;
     while (n>1) {
          p = trovaPosMax(v,n);
          if (p<n-1) scambia(&v[p],&v[n-1]);
          n--;
     }
    }
```

naiveSort è stabile in quanto nel confronto tra v[j] e valore massimo viene usato < anzichè <= e opera sul posto in quanto usa sempre solo due variabili aggiuntive.

Esempio dato l'array contenente
42 38 11 75 99 23 84 67
esso viene ordinato da naivesort attraverso le seguenti iterazioni (il valore sottolineato è quello da scambiare con il valore massimo della sequenza di origine)

42 38 11 75 99 23 84 67
alla prima iterazione l'ottavo elemento più grande (quello sottolineato) viene posizionato in ultima posizione (l'ottava cioè quella di indice 7) attraverso uno scambio. L'elemento più grande viene trovato dalla funzione **trovaPosMax** che cerca nei primi n elementi del vettore v la posizione occupata del valore più grande
11 38 42 75 67 23 84 99
alla seconda iterazione l'elemento più grande (quello sottolineato) tra i primi 7 viene posizionato in 7 posizione. Essendo già il valore 84 al suo posto non vengono effettuati scambi.
11 38 42 75 67 23 84 99

alla terza iterazione l’elemento più grande (quello sottolineato) tra i primi 6 viene posizionato in 6 posizione effettuando uno scambio.

11 38 42 23 67 75 84 99

alla quarta iterazione l’elemento più grande (quello sottolineato) tra i primi 5 viene posizionato in 5 posizione. Essendo già il valore 67 al suo posto non vengono effettuati scambi.

11 38 42 23 67 75 84 99

alla quinta iterazione l’elemento più grande (quello sottolineato) tra i primi 4 viene posizionato in 4 posizione effettuando uno scambio.

11 38 23 42 67 75 84 99

alla sesta iterazione l’elemento più grande (quello sottolineato) tra i primi 3 viene posizionato in 3 posizione effettuando uno scambio.

11 23 38 42 67 75 84 99

alla settima iterazione, ed ultima iterazione, l’elemento più grande (quello sottolineato) tra i primi 2 viene posizionato in 2 posizione. Essendo già il valore 23 al suo posto non vengono effettuati scambi.
Per valutar la complessità analizziamo il numero di confronti. Il numero di *confronti* necessari vale *sempre*:

$$\mathbf{(N-1) + (N-2) + (N-3) + \ldots + 2 + 1 = N*(N-1)/2 = O(N^2/2)}$$

Nel caso peggiore, il numero di scambi necessari è N-1, infatti viene effettuato al più uno scambio per iterazione. In generale il numero di scambi potrà essere minore.
È Importante notare che la complessità non dipende dai particolari dati di ingresso l’algoritmo fa gli stessi confronti sia per un array disordinato, sia per un array *già ordinato.*

3.2 Algoritmo di Ordinamento bubble sort

L’algoritmo **Bubble sort** tenta di corregge il difetto principale del naïve sort che quello di *non* accorgersi se l’array, a un certo punto, è già ordinato. BubbleSort (letteralmente: ordinamento a bolla) è un semplice algoritmo di ordinamento basato sullo scambio di elementi adiacenti. Ogni coppia di elementi adiacenti viene comparata e invertita di posizione se sono nell'ordine sbagliato. L'algoritmo continua nuovamente a rieseguire questi passaggi per tutta la sequenza di elementi finché non vengono più eseguiti scambi, situazione che indica che la lista è ordinata. L’algoritmo deve il suo nome al modo in cui gli elementi vengono ordinati cioè quelli più piccoli "risalgono" verso un'estremità della lista, così come fanno le bollicine in un bicchiere di spumante; al contrario quelli più grandi "affondano" verso l'estremità opposta della sequenza.
Il bubble sort è un algoritmo iterativo, ossia basato sulla ripetizione di un procedimento fondamentale. La singola iterazione dell'algoritmo prevede che gli elementi dell'array siano confrontati a due a due,

procedendo in un verso stabilito. La scelta del verso non è significativa; d'ora in poi ipotizzeremo che lo si scorra partendo dall'inizio.
Per esempio, saranno confrontati il primo e il secondo elemento, poi il secondo e il terzo, poi il terzo e il quarto, e così via fino al confronto fra il penultimo e l'ultimo elemento. Ad ogni confronto, se i due elementi confrontati non sono ordinati secondo il criterio prescelto, vengono scambiati di posizione. Durante ogni iterazione almeno un valore viene spostato rapidamente fino a raggiungere la sua collocazione definitiva; in particolare, alla prima iterazione il numero più grande raggiunge l'ultima posizione dell'array, alla seconda il secondo numero più grande raggiunge la penultima posizione, e così via.
Il motivo è semplice, e si può illustrare con un esempio. Supponiamo che l'array sia inizialmente disposto come segue:
15 6 4 10 11 2
Inizialmente 15 viene confrontato con 6, ed essendo 15>6, i due numeri vengono scambiati:
6 15 4 10 11 2
A questo punto il 15 viene confrontato col 4, e nuovamente scambiato:
6 4 15 10 11 2
Non è difficile osservare che, essendo 15 il numero massimo, ogni successivo confronto porterà a uno scambio e a un nuovo spostamento di 15, che terminerà nell'ultima cella dell'array. Per motivi analoghi, alla seconda iterazione è garantito che il secondo numero più grande raggiungerà la sua collocazione definitiva nella penultima cella dell'array, e via dicendo. Ne conseguono due considerazioni:

- se i numeri sono in tutto N, dopo N-1 iterazioni si avrà la garanzia che l'array sia ordinato;
- alla iterazione i-esima, le ultime i-1 celle dell'array ospitano i loro valori definitivi, per cui la sequenza di confronti può essere terminata col confronto dei valori alle posizioni N-1-i e N-i.

L'algoritmo operando per *"passate successive"* sull'array , ad a ogni iterazione, considera una ad una *tutte le possibili coppie di elementi adiacenti*, scambiandoli se risultano nell'ordine errato. In questo modo dopo ogni iterazione, l'elemento massimo è in fondo alla parte di array considerata
Quando non si verificano scambi, l'array è ordinato, e l'algoritmo termina.
Ovviamente, a ogni iterazione può accadere che più numeri vengano spostati; per cui, oltre a portare il numero più grande in fondo, ogni singola iterazione può contribuire anche a un riordinamento parziale degli altri valori. Anche per questo motivo, può accadere (e

normalmente accade) che l'array risulti effettivamente ordinato *prima* che si sia raggiunta la N-1esima iterazione.

L'algoritmo utilizza una variabile booleana (in C un int) usata come "flag" che indica se nell'iterazione corrente si è eseguito almeno uno scambio. La variabile viene reimpostata a *false* all'inizio di ogni iterazione, e impostata a *true* solo nel caso in cui si proceda a uno scambio. Se al termine di una iterazione completa il valore della variabile flag è false, l'array è ordinato e l'intero algoritmo viene terminato. Questa tecnica produce una riduzione del tempo *medio* di esecuzione dell'algoritmo, pur con un certo overhead costante(assegnamento e confronto della variabile *flag*).

Di seguito viene la codifica in C della versione originale dell'algoritmo:

```
void scambia(tipobase v[], unsigned long i, unsigned long j)
{
     tipobase tmp=v[i];
     v[i]=v[j];
     v[j]=tmp;
}

void BubbleSort(tipobase v[], unsigned long dim)
{
    short ordinato=0;
    unsigned long i,j;

    for (j=0; j<dim-1 && !ordinato; j++) {
        ordinato=1;
        for (i=dim-1; i>j; i--)
       if (v[i]<v[i-1]) {
                   scambia(v,i,i-1);
                   ordinato=0;
        }
    }
}
```

Esistono numerose varianti del bubblesort, molte delle quali possono essere definite ottimizzazioni, in quanto mirano a ottenere lo stesso risultato finale (l'ordinamento dell'array) eseguendo, in media, meno operazioni, quindi impiegando meno tempo. Una di queste è basata sull'osservazione che (sempre assumendo una scansione dell'array, per esempio, in avanti, e ordinamento crescente) se una data iterazione non sposta nessun elemento di posizione maggiore di un dato valore *i*, allora si può facilmente dimostrare che nessuna iterazione successiva eseguirà scambi in posizioni successive a tale valore *i*. L'algoritmo può dunque essere ottimizzato memorizzando l'indice a cui avviene l'ultimo scambio durante una iterazione, e limitando le iterazioni successive alla scansione dell'array solo fino a tale posizione.

Di seguito viene la codifica in C della versione modificata dell'algoritmo:

```
void scambia(tipobase *a, tipobase *b){
 tipobase tmp;
 tmp = *a;
 *a = *b;
 *b = z;
}
void bubbleSort(tipobase v[], int n){
 int i;
 int ultimoScambiato = n;
 while (n>1 && ultimoScambiato>0){
      ordinato = 1;
      for (i=0; i<n-1; i++)
           if (v[i]>v[i+1]) {
                scambia(&v[i],&v[i+1]);
                ordinato = i; }
      n= ultimoScambiato;
 }
}
```

Il Bubble sort non è un algoritmo efficiente. Il bubble sort effettua all'incirca $N^2/2$ confronti ed $N^2/2$ scambi sia in media che nel caso peggiore. Il tempo di esecuzione dell'algoritmo è $O(N^2)$.

3.3 Algoritmo di Ordinamento shaker sort

Lo Shaker sort, noto anche come Bubble sort bidirezionale è sostanzialmente una variante del bubble sort. Si differenzia da quest'ultimo per l'indice del ciclo più interno che, anziché scorrere dall'inizio alla fine, inverte la sua direzione ad ogni ciclo.

Le prestazioni del bubble sort possono essere leggermente migliorate tenendo conto del fatto che dopo la prima iterazione l'elemento più grande si troverà certamente nell'ultima posizione della lista, quella sua definitiva; alla seconda iterazione il secondo più grande si troverà in penultima posizione, quella sua definitiva, e così via. Ad ogni iterazione il ciclo dei confronti può accorciarsi di un passo rispetto al precedente evitando di scorrere ogni volta tutta la lista fino in fondo: all'n-esima iterazione si può quindi fare a meno di trattare gli ultimi n-1 elementi che ormai si trovano nella loro posizione definitiva.

Anche per ShakerSort si possono considerare due varianti. La prima che rileva solo il fatto che in una coppia di iterazioni (up-down e down-up) non si siano effettuati scambi l seconda in cui si considera la pozione in cui è stato effettuato l’ultimo scambio.

```
void scambia(tipobase *a, tipobase *b){
 tipobase z;
 z = *a;
 *a = *b;
 *b = z;
}
void shakersort(tipobase v[], int n){
 int left = 0, right = n, fine, i;
 do { fine = 1;
      --right;
      for ( i = left; i < right; i++)
           if (v[i] > v[i+1]) {
                scambia(&v[i], &v[i+1]);
                fine = 0; }
      if (fine) return;
      fine = 1;
      for (i = right; i > left; i--)
           if (v[i] < v[i-1]) {
                scambia(&v[i], &v[i-1]);
                fine = 0; }
```

```
          ++left;
     } while (!fine);
    }
```

Mentre se si considera la seconda versione

```
    void scambia(tipobase *a, tipobase *b){
     tipobase z;
     z = *a;
     *a = *b;
     *b = z;
    }
    void shakersort(tipobase v[], int n){
     int left = 0, right = n-1, fine, i;
     do {for ( i = left; i < right; i++)
               if (v[i] > v[i+1]) {
                    scambia(&v[i], &v[i+1]);
                    fine = i; }
          right = fine;
          for (i = right; i > left; i--)
               if (v[i] < v[i-1]) {
                    scambia(&v[i], &v[i-1]);
                    fine = i; }
          left = fine;
     } while(left<right);
    }
```

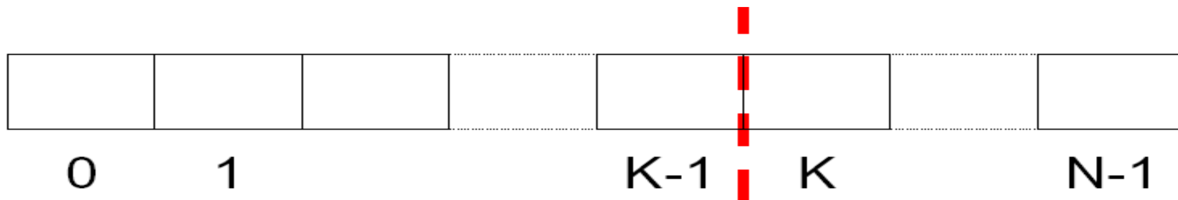
3.4 Ordinamento insert sort

L'algoritmo di ordinamento InsertSort (ordinamento diretto) nasce dall'idea che Per ottenere un array ordinato basta *costruirlo ordinato, inserendo gli elementi al posto giusto fin dall'inizio.*

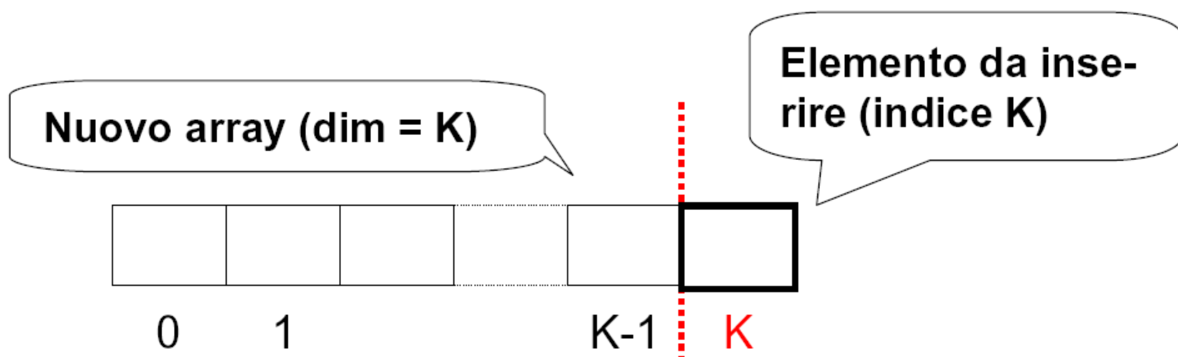
Idealmente, il metodo costruisce un nuovo array, contenente gli stessi elementi del primo, ma ordinato.

In pratica, *non è necessario costruire un secondo array*, in quanto le stesse operazioni possono essere svolte direttamente sull'array originale: così, alla fine esso risulterà ordinato. Insertsort, in realtà, è un algortmo di ordinamento iterativo che al generico passo i vede l'array diviso in una sequenza di destinazione v[0], . . . , v[k − 1] già ordinata e una sequenza di origine v[k], . . . , v[n − 1] ancora da

ordinare. L'obiettivo `e di inserire il valore contenuto in a[k] al posto giusto nella sequenza di destinazione facendolo scivolare a ritroso, in modo da ridurre la sequenza di origine di un elemento.



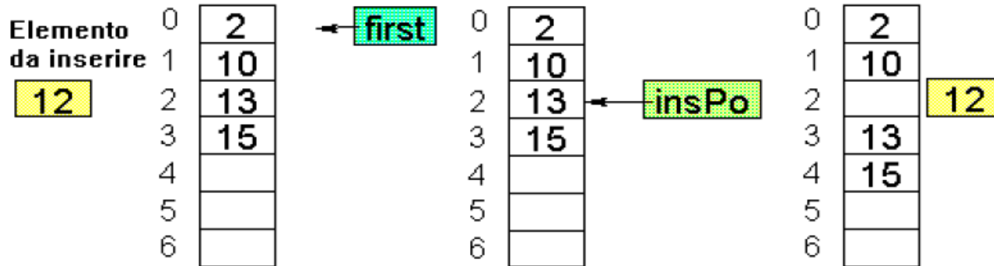
Pertanto "vecchio" e "nuovo" array condividono lo stesso array fisico di N celle (da 0 a N-1) ed in ogni istante, le prime K celle (numerate da 0 a K-1) costituiscono il nuovo array mentre le successive N-K celle costituiscono la parte residua dell'array originale. Come conseguenza della scelta di progetto fatta, in ogni istante il nuovo elemento da inserire si trova nella cella successiva alla fine del nuovo array, cioè la (K+1)-esima (il cui indice è K)



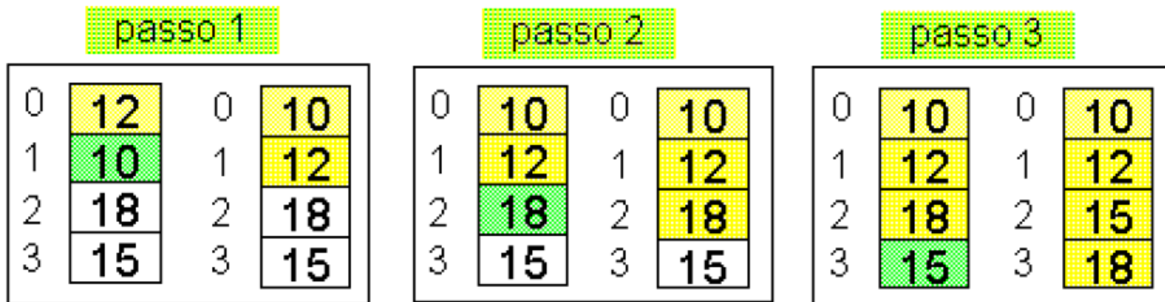

```
Specifica
for (k=1; k<n; k++)
<inserisci alla posizione k-esima del nuovo
array l'elemento minore fra quelli rimasti
nell'array originale>

void insMinore(tipobase v[], int pos){
<determina la posizione in cui va inserito il nuovo elemento>
<crea lo spazio spostando gli altri elementi in avanti di una
posizione>
<inserisci il nuovo elemento alla posizione prevista>
}
```

Esempio

Scelta di progetto: se il nuovo array è lungo K=4 (numerate da 0 a 3) l'elemento da inserire si trova nella cella successiva (di indice K=4).



Esempio



Implementazione

```
void insMinore(tipobase v[], int pos){
 int i = pos-1;
tipobase x = v[pos];
 while (i>=0 && x<v[i])   {
      v[i+1]= v[i]; /* crea lo spazio */
      i--;
 }
 v[i+1]=x; /* inserisce l'elemento */
```

```
}

void insertSort(tipobase v[], int n){
 int k;
 for (k=1; k<n; k++)
 insMinore(v,k);
}
```

Insertsort è stabile in quanto nel confronto tra a[j] e valore ins viene usato > piuttosto che >=. Insertsort opera sul posto in quanto usa soltanto una variabile aggiuntiva (valore ins).
Si osserva che l'implementazione dell'algoritmo esegue al più un numero costante di passi per ogni confronto tra elementi del vettore di ingresso. Possiamo quindi affermare che il tempo di calcolo è dello stesso ordine di grandezza del numero di confronti eseguiti.
Il caso peggiore, quello con il massimo numero di confronti, occorre quando v[1] > v[2] > · · · > v[n]. In questo caso la procedura esegue Σ^{n-1} i = n(n-1)/2 confronti. Di conseguenza il tempo di calcolo richiesto dall'algoritmo su un input di lunghezza n è $\Theta(n^2)$ nel caso peggiore.
Nel caso migliore invece, quando il vettore A è già ordinato, la procedura esegue solo n − 1 confronti e di conseguenza il tempo di calcolo risulta lineare. Osserviamo tuttavia che il caso migliore non è rappresentativo. Infatti, supponendo di avere in input una permutazione casuale (uniformemente distribuita) di elementi distinti, si può dimostrare che il numero medio di confronti risulta n(n-1)/4. Quindi, anche nel caso medio, il tempo di calcolo resta quadratico.
Esempio il vettore
42 38 11 75 99 23 84 67
viene ordinato da insertsort attraverso le seguenti iterazioni (il valore sottolineato `e quello da inserire al posto giusto nella sequenza di destinazione)
42 38 11 75 99 23 84 67
Alla prima iterazione il secondo valore (quello sottolineato) deve essere fatto scorrere fino ad occupare la posizione corretta. Per fare cio sarà eseguito uno scambio
38 42 11 75 99 23 84 67
Alla seconda iterazione il terzo valore (quello sottolineato) deve essere fatto scorrere fino ad occupare la posizione corretta. Per fare cio saranno eseguiti 2 scambi.
11 38 42 75 99 23 84 67

Alla terza iterazione il quarto valore (quello sottolineato) deve essere fatto scorrere fino ad occupare la posizione corretta. Essendo gia in questo caso nella posizione corretta non vengono effettuati scambi

11 38 42 75 <u>99</u> 23 84 67

Alla quarta iterazione il quinto valore (quello sottolineato) deve essere fatto scorrere fino ad occupare la posizione corretta. Essendo già in questo caso nella posizione corretta non vengono effettuati scambi

11 38 42 75 99 <u>23</u> 84 67

Alla quinta iterazione il sesto valore (quello sottolineato) deve essere fatto scorrere fino ad occupare la posizione corretta. Per fare cio saranno eseguiti 4 scambi.

11 23 38 42 75 99 <u>84</u> 67

Alla sesta iterazione il settimo valore (quello sottolineato) deve essere fatto scorrere fino ad occupare la posizione corretta. Per fare cio sarà eseguito uno scambio.

11 23 38 42 75 84 99 <u>67</u>

Alla settima iterazione l'ottavo valore (quello sottolineato) deve essere fatto scorrere fino ad occupare la posizione corretta. Per fare cio saranno eseguiti 4 scambi.

11 23 38 42 67 75 84 99

3. 5 Algoritmo di Ordinamento QuickSort

L'algoritmo di ordinamento QuickSort si basa su un concetto molto semplice. Si consideri un vettore di n elementi e si supponga di ordinarlo secondo algoritmi non evoluti, ossia quelli che presentano un tempo di calcolo proporzionale a n^2, come detto prima. Si supponga, adesso di dividere il vettore da ordinare in due pezzi di n/2 elementi ciascuno e di ordinare le due metà separatamente. Applicando sempre gli algoritmi non evoluti, si avrebbe un tempo di calcolo pari a $(n/2)^2 + (n/2)^2 = n^2/2$. Si supponga adesso di riunire i due vettori ordinati e di poter riottenere il vettore originario ordinato. E' chiaro che se questo ragionamento si potesse applicare anche immaginando una decomposizione del vettore originario in quattro, si avrebbe un tempo di calcolo totale pari a: $(n/4)^2 + (n/4)^2 + (n/4)^2 + (n/4)^2 = n^2/4$. Se si potesse dividere in 8, si avrebbe un tempo ancora più basso, e così via dicendo.

L'unico problema è che in linea di massima il ragionamento descritto prima non funziona sempre. Si consideri ad esempio il vettore:

13	10	1	45	15	12	21	15	29	34
0	1	2	3	4	5	6	7	8	9

Si supponga di divederlo a metà e di ordinare separatamente i due pezzi, ottenendo i due vettori:

1	10	13	15	45
0	1	2	3	4

12	15	21	29	34
5	6	7	8	9

Se i due vettori ordinati vengono riuniti assieme, otteniamo il vettore:

1	10	13	15	45	12	15	21	29	34
0	1	2	3	4	5	6	7	8	9

che ovviamente non è ordinato.

Un modo per superare il problema è quello di operare nel modo seguente:

1. prima di dividere il vettore, gli elementi del vettore vengono spostati in modo che la prima metà (quella a sinistra) contiene elementi tutti più piccoli della seconda metà (quella a destra).
2. il vettore viene decomposto nelle due metà
3. le due metà vengono ordinate separatamente
4. il vettore originario è ordinato mettendo insieme le due metà ordinate separatamente

Il QuickSort segue questa "strategia". In particolare, l'algoritmo QuickSort si compone dei seguenti passi:

1. il vettore da ordinare viene delimitato dall'indice del primo elemento (inf) e dall'indice dell'ultimo elemento (sup).
2. viene scelto un elemento del vettore di indice compreso tra inf e sup, chiamato pivot. Il pivot viene scelto del tutto casualmente. Una scelta potrebbe essere l'elemento che ha come indice (inf+sup)/2.
3. vengono utilizzati due indici i,j, che vengono inizializzati a i=inf e j=sup.
4. l'indice i viene incrementato di 1 fino a quando l'elemento di indice i non è maggiore o uguale al pivot.
5. l'indice j viene decrementato di 1 fino a quando l'elemento di indice j non è minore o uguale al pivot.
6. nel caso in cui i<j, gli elementi di indici i e j vengono scambiati, e poi i viene incrementato e j decrementato (in modo unitario)
7. nel caso in cui i=j, non si effettua lo scambio, ma i viene incrementato di 1 e j viene decrementato di 1.
8. se i>j allora l'algoritmo termina. In questo modo risulta che:
    a) tutti gli elementi di indice appartenente a inf,..,j sono minori o uguali del pivot
    b) tutti gli elementi di indice appartenente a i,..,sup sono maggiori o uguali del pivot

c) tutti gli elementi (se esistono) di indice appartenente a j+1,..,i-1 sono uguali del pivot

9. L'algortimo QuickSort viene applicato ricorsivamente al sottovettore individuato dagli indici (inf,j), se tale sottovettore contiene almeno un elemento, ossia se inf<j
10. l'algortimo QuickSort viene applicato ricorsivamente al sottovettore individuato dagli indici (i, sup), se tale sottovettore contiene almeno un elemento, ossia se i<sup

Esempi del QuickSort

Nel seguito verranno mostrati tre esempi del QuickSort, che si riferiscono a situazioni reali. Nel primo esempio verrà considerato il caso in cui il vettore originario viene decomposto nella prima ricorsione in due sottovettori di dimensione uguale (pari alla metà della dimensione del vettore originario). Nel secondo esempio, il vettore originario viene decomposto in due sottovettori, di cui il primo ha dimensione uguale alla dimensione originaria meno 1, e l'altro ha una dimensione unitaria. I due primi esempi rappresentano il caso migliore e il caso peggiore del comportamento del QuickSort. I due casi sono legati alla scelta del pivot. Come si vedrà negli esempi, nel primo esempio il pivot viene scelto pari al mediano[1] degli elementi contenuti nel vettore, e ciò rappresenta il caso migliore. Nel secondo esempio, invece, il pivot coincide con l'elemento massimo del vettore. Ciò rappresenta il caso peggiore del comportamento del QuickSort.
Il terzo esempio si riferisce ad un comportamento generico (ossia compreso tra il peggiore e il migliore) del QuickSort.

Esempio 1: il Pivot coincide con l'elemento mediano del vettore.

Si consideri il seguente vettore di n=10 elementi:

13	10	1	45	15	12	21	15	29	34
i=inf				m=(inf+sup)/2					J=sup

In tal caso, risulta pivot=vett[m]=15, ossia casualmente il pivot coincide con l'elemento mediano del vettore. Infatti il numero di elementi più piccoli di 15 è 4, mentre il numero di elementi più grandi di lui è 4. L'indice i viene incrementato fino a quando non viene trovato un elemento più grande o uguale al pivot. Nel nostro esempio l'indice i si arresta in corrispondenza dell'elemento 45. Per quanto riguarda l'indice j esso viene spostato fino a quando non si perviene all'elemento 15, uguale al pivot.

13	10	1	45	15	12	21	15	29	34
			i				**j**		

[1] **Per definizione, l'elemento mediano di un insieme è quello tale che il numero di elementi dell'insieme più grandi di lui è all'incirca uguale al numero di elementi dell'insieme che sono più piccoli di lui.**

Gli elementi di indice i e j vengono scambiati, e l'indice i viene incrementato, mentre j viene decrementato, ottenendo:

13	10	1	15	15	12	21	45	29	34
				i		**j**			

L'indice i viene arrestato in quanto esso corrisponde al pivot. L'indice j viene fatto decrementare fino a quando esso perviene all'elemento 12, che è inferiore al pivot. Gli indici sono dunque come mostrato dalla seguente figura:

13	10	1	15	15	12	21	45	29	34
				i	**j**				

Gli elementi di indice i e j vengono scambiati e successivamente i viene incrementato e j viene decrementato, ottenendo:

13	10	1	15	12	15	21	45	29	34
				j	**i**				

La prima ricorsione dell'algoritmo QuickSort si conclude, perché i due indici i e j si sono invertiti. Come si vede alla fine della prima passata di ordinamento risulta:

- tutti gli elementi di indice appartenente a inf,..,j sono minori o uguali del pivot
- tutti gli elementi di indice appartenente a i,..,sup sono maggiori o uguali del pivot
- non ci sono elementi di indice appartenente a j+1,..,i-1 (se ci fossero stati, sarebbero stati uguali al pivot).

Come si vede l'esempio considerato rappresenta il caso migliore perché il vettore originario è stato decomposto in due vettori che hanno entrambi dimensione uguale e pari a metà della dimensione iniziale.
L'algoritmo procede ricorsivamente operando sui vettori delimitati dagli indici (inf,..,j) e (i,..,sup).

Esempio 2: il Pivot coincide con l'elemento più grande del vettore.

Si consideri il seguente vettore di n=10 elementi:

13	20	1	15	34	28	21	14	29	3
i=inf				m=(inf+sup)/2					j=sup

In tal caso, risulta pivot=vett[m]=34, ossia casualmente il pivot coincide con l'elemento massimo del vettore. L'indice i viene incrementato fino a quando non viene trovato un elemento più grande o uguale al pivot. Nel nostro esempio l'indice i si arresta in corrispondenza del pivot. *L'esempio mette in evidenza il motivo di incrementare l'indice i fino a quando si trova un elemento più grande o* ***uguale*** *al pivot. Se non ci fosse la condizione* ***uguale****, nel nostro esempio l'indice i verrebbe*

<u>*continuamente incrementato oltre la dimensione del vettore.*</u> Per quanto riguarda l'indice j esso non viene spostato in quanto l'elemento j-esimo è inferiore al pivot.

13	20	1	15	34	28	21	14	29	3
				i					**j**

Gli elementi di indice i e j vengono scambiati, e l'indice i viene incrementato, mentre j viene decrementato, ottenendo:

13	20	1	15	3	28	21	14	29	34
					i			**J**	

L'indice i viene fatto incrementare fino a quando arriva all'elemento 34, che è pari al pivot. Infatti secondo la regola di incremento dell'indice i, esso viene fatto arrestare perché è stato trovato un elemento uguale al pivot. L'indice j non viene fatto decrementare perché si riferisce ad un elemento già inferiore al pivot. Gli indici sono dunque come mostrato dalla seguente figura:

13	20	1	15	3	28	21	14	29	34
inf								**J**	**i=sup**

Come si vede alla fine della prima passata di ordinamento risulta:

- tutti gli elementi di indice appartenente a inf,..,j sono minori o uguali del pivot
- tutti gli elementi di indice appartenente a i,..,sup sono maggiori o uguali del pivot
- non ci sono elementi di indice appartenente a j+1,..,i-1.

L'algoritmo procede ricorsivamente operando sui vettori delimitati dagli indici (inf,..,j) e (i,..,sup). Come si vede l'esempio considerato rappresenta un caso peggiore, perché il vettore originario è stato decomposto in due vettori di cui il primo (quello compreso tra inf e j) ha dimensione quasi uguale a quella originaria.

Esempio 3: il Pivot non coincide né con il massimo/minimo né con il mediano del vettore.

Si consideri il seguente vettore di n=10 elementi:

13	10	1	45	15	28	21	15	29	34
i=inf				m=(inf+sup)/2					j=sup

In tal caso, risulta pivot=vett[m]=15. Esso non è né il massimo (o il minimo) del vettore, né il mediano, visto che il numero di elementi più piccoli di 15 è 3, mentre il numero di elementi più grandi di lui è 5. L'indice i viene incrementato fino a quando non viene trovato un elemento più grande o uguale al pivot. Nel nostro esempio l'indice i si arresta in corrispondenza dell'elemento 45. Per quanto riguarda l'indice j esso viene spostato fino a quando non si perviene all'elemento 15, uguale al pivot.

13	10	1	45	15	28	21	15	29	34
			i				**j**		

Gli elementi di indice i e j vengono scambiati, e l'indice i viene incrementato, mentre j viene decrementato, ottenendo:

13	10	1	15	15	28	21	45	29	34
				i		**j**			

L'indice i viene arrestato in quanto esso corrisponde al pivot. L'indice j viene fatto decrementare fino a quando esso perviene all'elemento 15, che è uguale al pivot. Gli indici sono dunque come mostrato dalla seguente figura:

13	10	1	15	15	28	21	45	29	34
				i=j					

Gli elementi i e j non vengono scambiati, perché non avrebbe senso visto che i e j coincidono, e successivamente i viene incrementato e j viene decrementato, ottenendo:

13	10	1	15	15	28	21	45	29	34
			j		**i**				

La prima passata dell'algoritmo QuickSort si conclude, perché i due indici i e j si sono invertiti. Come si vede alla fine della prima passata di ordinamento risulta:

- tutti gli elementi di indice appartenente a inf,..,j sono minori o uguali del pivot
- tutti gli elementi di indice appartenente a i,..,sup sono maggiori o uguali del pivot
- vi è un solo elemento di indice appartenente a j+1,..,i-1, uguale al pivot (15).

L'algoritmo procede ricorsivamente operando sui vettori delimitati dagli indici (inf,..,j) e (i,..,sup).
La sequente procedura realizza l'algortimo QuickSort.

```
void scambia (tipobase v[], long i, long j) {
      tipobase tmp=v[i];
      v[i]=v[j];
      v[j]=tmp;
}

void QSort (tipobase v[], long inf, long sup){
 tipobase pivot=v[(inf+sup)/2];
long i=inf,j=sup;
while (i<=j) {
      while (v[i]<pivot) i++;
```

```
        while (v[j]>pivot) j--;
        if (i<j) scambia(v,i,j);
        if (i<=j) {
            i++;
            j--;
            }
    }
    if (inf<j) QSort(v,inf,j);
    if (i<sup) QSort(v,i,sup);
   }
```

Come si vede essa si compone di un ciclo while che termina quando i>j. Dentro questo ciclo due cicli while incrementano i e decrementano j. Si noti che i due cicli terminano sicuramente quando v[i] coincide con il pivot e v[j] coincide con il pivot. Nel caso in cui i<j viene effettuato lo scambio, che non viene realizzato se i=j. Se i≤j, la variabile i viene incrementata e j decrementata.

3.6 Algoritmo di Ordinamento merge sort

Il merge sort è un algoritmo di ordinamento basato su confronti che utilizza un processo di risoluzione ricorsivo, sfruttando la tecnica del Divide et Impera, che consiste nella suddivisione del problema in sottoproblemi della stessa natura di dimensione via via più piccola.

Concettualmente, l'algoritmo funziona nel seguente modo:

Se la sequenza da ordinare ha lunghezza 0 oppure 1, è già ordinata. Altrimenti:

La sequenza viene divisa (divide) in due metà (se la sequenza contiene un numero dispari di elementi, viene divisa in due sottosequenze di cui la prima ha un elemento in più della seconda)

Ognuna di queste sottosequenze viene ordinata, applicando ricorsivamente l'algoritmo(impera)

Le due sottosequenze ordinate vengono fuse (combina). Per fare questo, si estrae ripetutamente il minimo delle due sottosequenze e lo si pone nella sequenza in uscita, che risulterà ordinata

Specifica

```
void mergeSort(int v[], int iniz, int fine, int vout[]) {
if (<array non vuoto>){
<partiziona l'array in due metà>
<richiama mergeSort ricorsivamente sui due sub-array,
se non sono vuoti>
<fondi in vout i due sub-array ordinati>
```

```
}
}
```

Supponendo di dover ordinare la sequenza [9,10,5,6,67,2,1,18,3,4], l'algoritmo procede ricorsivamente dividendola in metà successive, fino ad arrivare ad elementi singoli che vengo quindi fusi in modo ordinato (merge) a coppie (vedi fig. successive)

[9,10] [5,] [6,67] [1,2] [18,] [3,4]

Al passo successivo, si fondono le coppie di array di due elementi:

[9,10] [5,6,67][1,2] [3,4,18]

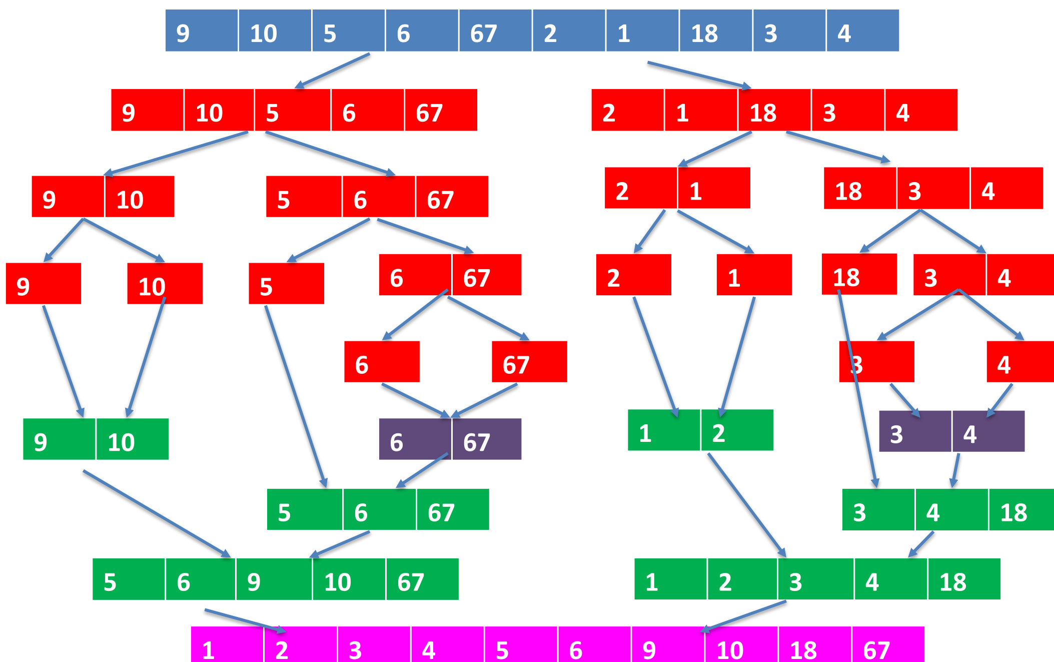
Al passo successivo si fondono le seuqneza

[5,6,9,10,67][1,2,3,4,18]

ed infine si fondono le due ultime sequenza

[1,2,3,4,5,6,9,10,18,67]

L'esecuzione ricorsiva all'interno del calcolatore non avviene nell'ordine descritto sopra. Tuttavia, si è formulato l'esempio in questo modo per renderlo più comprensibile.



La codifica C

```
void merge(int v[], int i1, int i2, int fine, int vout[]){
 int i=i1, j=i2, k=i1;
 while ( i <= i2-1 && j <= fine ) {
      if (v[i] < v[j]) vout[k] = v[i++];
      else vout[k] = v[j++];
      k++;
 }
 while (i<=i2-1) { vout[k] = v[i++]; k++; }
 while (j<=fine) { vout[k] = v[j++]; k++; }
 for (i=i1; i<=fine; i++) v[i] = vout[i];
}


void mergeSort(int v[], int first, int last, int vout[]) {
 int mid;
 if ( first < last ) {
      mid = (last + first) / 2;
      mergeSort(v, first, mid, vout);
      mergeSort(v, mid+1, last, vout);
      merge(v, first, mid+1, last, vout);
 }
}
```

Diversamente dagli altri algoritmi di ordinamento visti sinora, mergesort non opera sul posto in quanto nella fusione ordinata viene usato un array di appoggio il cui numero di elementi è proporzionale al numero di elementi dell’array da ordinare.

La complessità asintotica non dipende dalla disposizione iniziale dei valori negli elementi dell’array.

Merge Sort, per ordinare una sequenza di n oggetti, ha complessità temporale $O(n \log_2 n)$ sia nel caso medio che nel caso pessimo. Infatti:

- la funzione merge qui presentata ha complessità temporale $O(n)$
- mergesort richiama se stessa due volte, e ogni volta su (circa) metà dell sequenza in input

Da questo segue che il tempo di esecuzione dell'algoritmo è dato dalla ricorrenza:

$T(n) = 2T(n/2) + O(n)$ la cui soluzione è $O(n \log_2 n)$,

4. Confronto fra i diversi algoritmi

Nome	**Migliore**	**Medio**	**Peggiore**	**Stabile**	***In place***	**Caratteristiche dell'ordinamento**
Bubble sort	$O(n)$	$O(n^2)$	$O(n^2)$	Sì	Sì	Algoritmo per confronto tramite scambio di elementi
Insertion sort	$O(n)$	$O(n+d)$	$O(n^2)$	Sì	Sì	A. per confronto tramite inserimento
Merge sort	$O(n\log n)$	$O(n\log n)$	$O(n\log n)$	Sì	No	A. per confronto tramite unione di componenti, ottimale e facile da parallelizzare
Quicksort	$O(n\log n)$	$O(n\log n)$	$O(n^2)$	No	Sì	A. per confronto tramite partizionamento. Le sue varianti possono: essere stabili
Selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	No	Sì	A. per confronto tramite selezione di elementi
Shaker sort	$O(n)$	Varia	$O(n^2)$	Sì	Sì	A. per confronto tramite scambio di elementi, miglioria del *Bubble sort*.