9 - Algoritmi Fondamentali

Gli algoritmi fondamentali aiutano un utente a risolvere problemi comuni tramite soluzioni standard, riconosciute come corrette e ottimali.

I più diffusi risolvono task di ordinamento e ricerca dei dati.

Precisiamo una cosa, **non esiste un'unica soluzione che risolve un determinato problema**, ma come si fa a dire quale tra le soluzioni è ottimale? Tramite il criterio della **complessità computazionale**.
Ogni algoritmo ha una complessità ad essere eseguito, di solito gli algoritmi più efficienti sono quelli più difficili da implementare (complessità **implementativa**), mentre quelli più semplici sono i meno efficienti

Complessità di un algoritmo

La complessità computazionale è la **misura di quanto è complesso per un elaboratore eseguire un algoritmo** (ovvero la quantità di risorse usate da quel determinato algoritmo).

Le risorse che si sfruttano per un calcolatore sono:

- **Spazio**: quantità di memoria occupata durante l'esecuzione
- **Tempo**: quantità di tempo impiegata per ottenere la soluzione
  Minori quindi saranno le risorse utilizzate dall'algoritmo, minore sarà la sua complessità computazionale

Complessità temporale

Ma come si misura il tempo necessario per un algoritmo? Per convenzione si calcola il numero di volte in cui viene ripetuta l'operazione principale

Un codice del genere:

```
for(int i = 0; i < n; i++) {
    // operazioni
}
```

ha complessità n.
Quando effettuiamo calcoli di complessità in realtà calcoliamo delle approssimazioni, supponendo che il valore di n sia "grande", la differenza tra n e $n+1$ non è significativa, quindi l'istruzione di complessità 1 si può ignorare.

Tipicamente quando si calcola la complessità si cercano le istruzioni "dominanti", cioè quelle che vengono eseguite più volte. Spesso (non sempre!) la complessità degli algoritmi è

data dal numero di operazioni che vengono effettuate nei cicli.

Livelli di complessità di un algoritmo

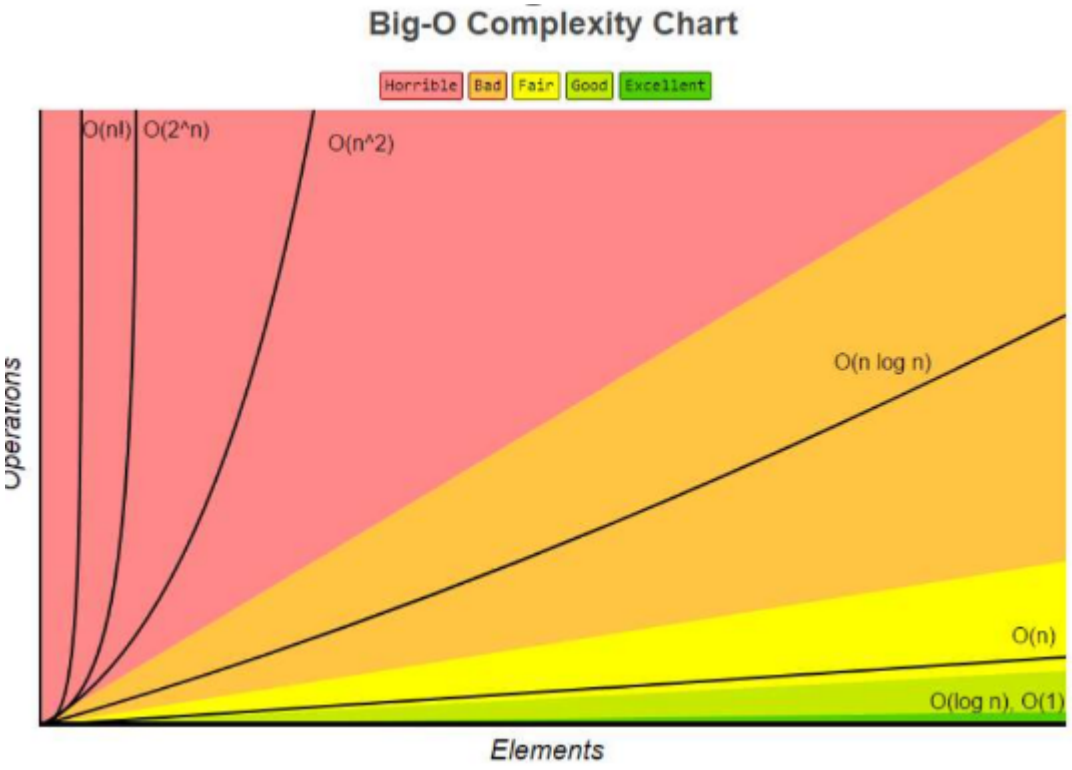
La complessità di un algoritmo è crescente:

- **Complessità crescente:**
  - Costante $O(1)$
  - Logaritmica $O(\log n)$
  - Lineare $O(n)$
  - nlog $O(n \log n)$
  - Polinomiale $O(n^m)$
    - Quadratica $O(n^2)$
    - Cubica $O(n^3)$
    - ...
  - Esponenziale $O(k^n)$ $k > 1$

Un algoritmo è ottimale quando la sua complessità è costante (complessità ottimale).
Le operazioni collegate agli operatori presenti nel linguaggio (es. assegnazione, confronto, etc.) hanno tutti complessità costante, perché basta 1 operazione (1 istruzione) per risolvere il problema.

La notazione $O(n)$ serve appunto a dire che la complessità è «approssimata» ad n
Gli algoritmi più comuni hanno una complessità polinomiale o lineare. Gli algoritmi più efficienti hanno una complessità logaritmica o $nlogO(nlogn)$ che sono vicine
alla complessità lineare, mentre gli algoritmi esponenziali non sono trattabili in modo

efficiente da un elaboratore



Quando parliamo di complessità computazionale dobbiamo distinguere diversi casi:

- **Migliore**: Corrispondente alla configurazione iniziale che comporta il **minimo** numero di esecuzioni dell'operazione principale
- **Peggiore**: Corrispondente alla configurazione iniziale che comporta il **massimo** numero di esecuzioni
- **Medio**

Ad esempio, in un algoritmo di ricerca la complessità computazionale è diversa se l'elemento da trovare è il primo del vettore (caso migliore), l'ultimo del vettore (caso peggiore) o è al centro del vettore (caso medio)
Allo stesso modo, in un algoritmo di ordinamento la complessità computazionale è diversa se il vettore è già ordinato (caso migliore), oppure ordinato in modo opposto (caso peggiore) rispetto a quello che vogliamo.

Algoritmi di ricerca

Poniamo un problema:
Si ricerca un elemento x in un insieme di n dati (come un array), e se è effettivamente presente, si restituisce la posizione.
Se l'elemento non viene trovato, si può restituire un valore speciale (ad esempio, `0` o `-1`).

Ricerca Lineare Esaustiva

Questa tecnica consiste nel **scandire ogni elemento** dell'insieme, memorizzando eventualmente la posizione in cui si trova l'elemento cercato, è utilizzabile anche su insiemi **non ordinati** e non richiede particolari condizioni.

L'algoritmo restituisce **l'ultima occorrenza** dell'elemento, il che è utile quando ci sono duplicati, sopratutto continua comunque a esaminare tutti gli elementi anche dopo aver trovato una corrispondenza.

La complessità di questo metodo è sempre $O(n)$ nel caso migliore, peggiore e medio, poiché scandisce sempre tutti gli elementi.

Esempio

```
int ricercaLineareEsaustiva(int a[], int n, int x) {
    int j = 0;
    int posizione = -1; // -1 indica elemento non trovato

    while (j < n) {
        if (a[j] == x) {
            posizione = j; // aggiorna la posizione ogni volta che trova x
        }
        j++;
    }

    return posizione; // restituisce l'ultima occorrenza o -1
}
```

Ricerca Lineare con Sentinella

Questa variante si interrompe **alla prima occorrenza** trovata, migliorando le prestazioni in casi favorevoli. È particolarmente utile quando si sa che l'elemento, se presente, è **unico**, oppure quando si è interessati **solo alla sua presenza**.
La complessità risulta:

- $O(1)$ nel caso migliore (elemento in prima posizione),
- $O(n)$ nel caso peggiore o se l'elemento è assente,
- $\frac{n+1}{2} \rightarrow O(n)$$in media, considerando una distribuzione casuale.

Esempio

```
int ricercaLineareConSentinella(int a[], int n, int x) {
    int j = 0;
    int posizione = -1; // -1 indica elemento non trovato

    while (j < n && posizione < 0) {
```

```
        if (a[j] == x) {
            posizione = j; // interrompe al primo match
        }
        j++;
    }

    return posizione; // restituisce la prima occorrenza o -1
}
```

Ricerca Binaria

La ricerca binaria è l'algoritmo di ricerca **più efficiente** in termini di complessità, ma può essere applicato **solo su insiemi ordinati**. Richiede dunque che i dati siano preordinati, eventualmente tramite un algoritmo di ordinamento.

L'idea è quella di confrontare l'elemento cercato con quello al centro dell'intervallo considerato e, in base al confronto, **escludere metà dei dati**: se è maggiore, si analizza la metà destra; se è minore, la metà sinistra. Si ripete finché:

- l'elemento viene trovato, oppure
- l'intervallo si riduce a zero.

La **complessità è O(log n)** in tutti i casi:

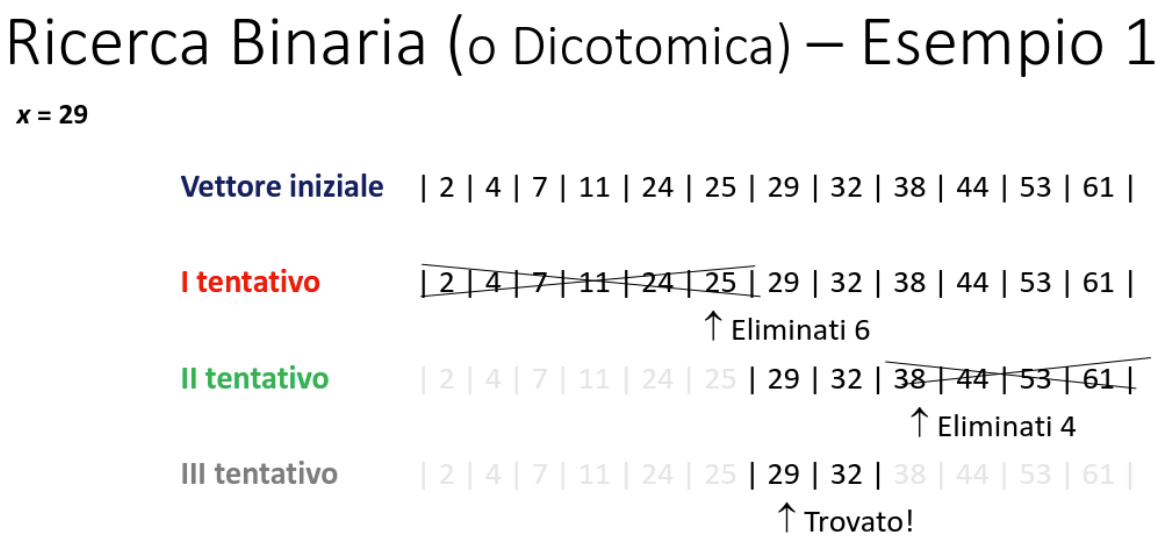
- **O(1)** nel migliore, quando l'elemento è già al centro.
- **O(log n)** nel peggiore e medio, poiché si dimezza ogni volta il numero di elementi da analizzare.
  Ad esempio, in un array di 128 elementi si impiegano al massimo **8 cicli**, dato che `log2(128) = 7` e si aggiunge 1 per il conteggio iniziale.
  (infatti nel primo ciclo abbiamo 128 elementi da esaminare, al termine elementi da esaminare dimezzati, al secondo ciclo diventano 64 elementi da esaminare e via dicendo fino al settimo ciclo dove sono 2 e all'ottavo ciclo 1)

Il codice tipico segue questi passaggi:

1. Inizializza `first = 0` e `last = n - 1`.
2. Calcola il valore mediano `j = (first + last) / 2`.
3. Confronta `x` con `a[j]`:
   - Se uguali, restituisce `j`.
   - Se `x > a[j]`, aggiorna `first = j + 1`.
   - Se `x < a[j]`, aggiorna `last = j - 1`.
     Esempio

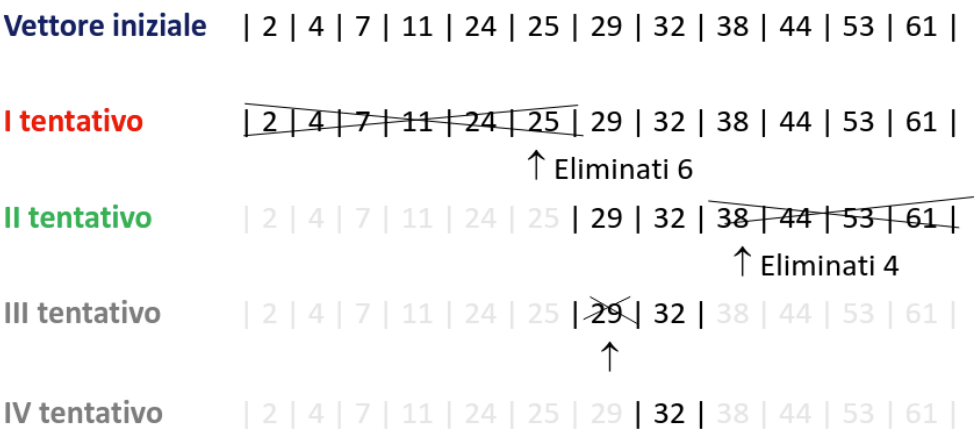
```c
int ricercaBinaria(int a[], int n, int x) {
    int posizione = -1; // -1 indica elemento non trovato
    int first = 0;
    int last = n - 1;

    while (first <= last && posizione == -1) {
        int mid = (first + last) / 2;

        if (a[mid] == x) {
            posizione = mid; // trovato
        } else if (x > a[mid]) {
            first = mid + 1; // cerca nella metà destra
        } else {
            last = mid - 1; // cerca nella metà sinistra
        }
    }

    return posizione; // indice dell'elemento o -1
}
```

Vediamo ora qualche esempio con i numeri



Ricerca Binaria (o Dicotomica) – Esempio 2

$x = 31$



- **X=31:** non trovato in 4 tentativi

Algoritmi di ordinamento

L'obiettivo è disporre gli elementi in una precisa **relazione d'ordine**.
La *CC* si basa sul tipo di dato che abbiamo dichiarato, in base a ciò l'ordinamento può essere **numerico** e **alfanumerico**, e possono essere entrambi **crescenti/decrescenti**.
Anche in questo caso non esiste un **algoritmo migliore in assoluto** ma dipende dal contesto del problema in cui ci ritroviamo.
L'attività di ordinamento occupa in media il 30% del tempo del calcolo dell'elaboratore, per questo è un attività di elaborazione **importante**.
Gli algoritmi di ordinamento si dividono in due operazioni differenti tra gli elementi, **confronti** e **scambi**.
L'ordinamento si dive anche in:

- **Esterni**: usando un array di appoggio, in cui si farà doppia occupazione di memoria e la necessità di copiare il risultato nell'array originale.
- **Interni**: l'ordinamento viene eseguito sullo stesso array da ordinare.
  I più famosi ed utilizzati sono:
- **Per selezione (Selection Sort)**
- **A bolle (Bubble Sort)**
- **Per inserzione (Insert Sort)**

Selection sort

Esso è basato sul concetto di **minimi successivi**, ovvero:

1. trovare il **più piccolo elemento** dell'insieme e porlo in prima posizione
2. trovare il **più piccolo dei rimanenti (n-1)** elementi e sistemarlo in seconda posizione

3. ripetere finché si trovi e collochi il **penultimo elemento**
4. l'ultimo elemento sarà automaticamente sistemato

```
void selectionSort(int a[], int n){
    for(i=0; i<n-1; i++){
        min=a[i];
        p=i; //p=posizione del minimo, min=val.minimo
        for(j=i+1;j<n;j++){ //trovare il minimo
            if(a[j]<min){
                min=a[j];
                p=j;
            }
            a[p]=a[i]; //una volta individuato il minimo
            a[i]=min; //effetto lo scambio tra i valori
        }
    }
}
```

“Pasted image 20250512125846.png” could not be found.

Complessità del Selection Sort

La complessità totale è data dalla complessità dei due cicli.
*Trattando i **confronti***:
Il **ciclo esterno** si ripete n-1 volte, mentre il **ciclo interno** ricerca il minimo nella parte dell'array non ancora ordinata. La complessità è **quadratica** $O(n^2)$
I confronti tra gli elementi nell'array si misurano tramite la formula $\frac{n(n-1)}{2}$
*Trattando gli **scambi***:
Lo scambio viene effettuato solo quando viene trovato il **minimo**, uno per ogni passo di ordinamento del sotto-array.

Ogni ciclo scorre tutta la parte **non ordinata**, non trae quindi vantaggio da un eventuale **pre-ordinamento**. Il suo vantaggio è che vengono effettuati pochi scambi, ogni scambio di complessità $O(1)$ poiché richiedono solo 3 passaggi.

Bubble Sort

Rispetto all'algoritmo precedente, questo algoritmo è **espressamente** basato sugli scambi degli elementi. Gli elementi più piccoli salgono verso l'alto come bolle, ad ogni passo si **ordina** un elemento. Il numero di scambi quindi sarà nettamente maggiore rispetto al

selection sort.

	Inizio	I/1	I/2	I/3	I/4	II
array(1)	4	4	4	4	4	1
array(2)	3	3	3	3	1	4
array(3)	6	6	6	1	3	3
array(4)	1	1	1	6	6	6
array(5)	5	2	2	2	2	2
array(6)	2	5	5	5	5	5

Complessità del Bubble Sort

I vari casi di complessità del bubble sort varia sul:

- **caso migliore**: eseguendo un singolo passo di ciclo quando la **lista è già ordinata**, con complessità $O(n)$
- **caso peggiore**: è l'esatto opposto, ovvero quando nessun elemento è posto in ordine, avendo un numero di passi pari a $n-1$. Vengono eseguiti un numero di confronti in maniera **decrescente**, perché ad ogni passo una parte sarà sempre ordinata, gli *n-i* scambi sono in tutto $(n-1)*n/2-> O(n^2)$, notando un incremento di scambi altamente maggiore della Selezione.
- **caso medio**: ove gli scambi siano pari alla metà dei confronti, $O(n^2)$
  Il bubble sort è chiaramente inferiori agli altri metodi, nel caso peggiore il numero di confronti sarà pari all'ordinamento per selezione, ma con scambi maggiori. E' molto veloce per gli insiemi con alto grado di preordinamento. Per definire uno schema di vantaggi si può dichiarare che:
- **Se l'insieme è pre-ordinato**: $BubbleSort > SelectionSort$
- **Se l'insieme non è pre-ordinato**: $BubbleSort < SelectionSort$

Inserction Sort

Si basa sul metodo eseguito nei giochi di carte per mischiare il mazzo.
Questo algoritmo **ricerca** la giusta posizione d'ordine di ogni elemento rispetto alla parte già ordinata. Gli elementi da ordinare vengono considerati uno per volta, si confronta l'elemento *n* con tutti quelli della parte ordinata e lo si colloca nella giusta posizione, facendo scalare gli altri o di uno a destra o di uno a sinistra. Al primo passo avremo due elementi ordinati, quindi dopo *n-1* passi avremo tutti gli elementi ordinati.
In totale questo algoritmo effettua **n-1 passi**.

	Inizio	I	II	III	IV	V
array(1)	40	30	30	10	10	10
array(2)	30	40	40	30	30	20
array(3)	60	60	60	40	40	30
array(4)	10	10	10	60	50	40
array(5)	50	50	50	50	60	50
array(6)	20	20	20	20	20	60

Ad ogni passo dell'algoritmo una parte sarà chiamata **parte del vettore ordinata** e l'altra zona è la **parte non ordinata**.
La scansione della posizione in cui inserire il nuovo elemento nella parte ordinata è **sequenziale**. Al ciclo *i*, avremo *i+1* **elementi ordinati** e *n-(i+1)* **elementi non ordinati**.
Anche questo algoritmo fa uso di una **sentinella**, per comprendere se l'elemento è stato inserito o meno al posto giusto, se non è stato inserito continua fin quando non lo trova, e se è minore del precedente allora si scalerà il precedente, altrimenti viceversa, e si posizione poi fin quando non trova la posizione giusta. Bisogna anche controllare che se questo elemento analizzato arriva in prima posizione, non dovrà mai essere più spostato da lì, rendendo successivamente vera la sentinella.

Inserct Sort in C:

```c
void inserction_sort(int x[], int n){
    int i,j,app;
    for(i=1;i<n;i++){ //ciclo che scorre tutti gli elementi partendo dal
secondo
        app=x[i]; //salva il valore da posizionare
        j=i-1; //memorizza l'indice dell'ultimo valore ordinato
        while(j>0 && x[j]>app){
            x[j+1]=x[j];
            j--; //se l'elemento è più piccolo di quello da posizionar
        } //ciclo per individuare la posizone corretta
        x[j+1]=app; /* esce dal ciclo quando ha trovato la posizione
    }               * per l'elemento, dunque inserisce il valore lì*/
    return;
}
```

Complessità Inserction Sort

I passi saranno sempre pari a $n-1$, con un numero di scambi che è pari ad ogni confronto effettuato, salvo l'ultimo.

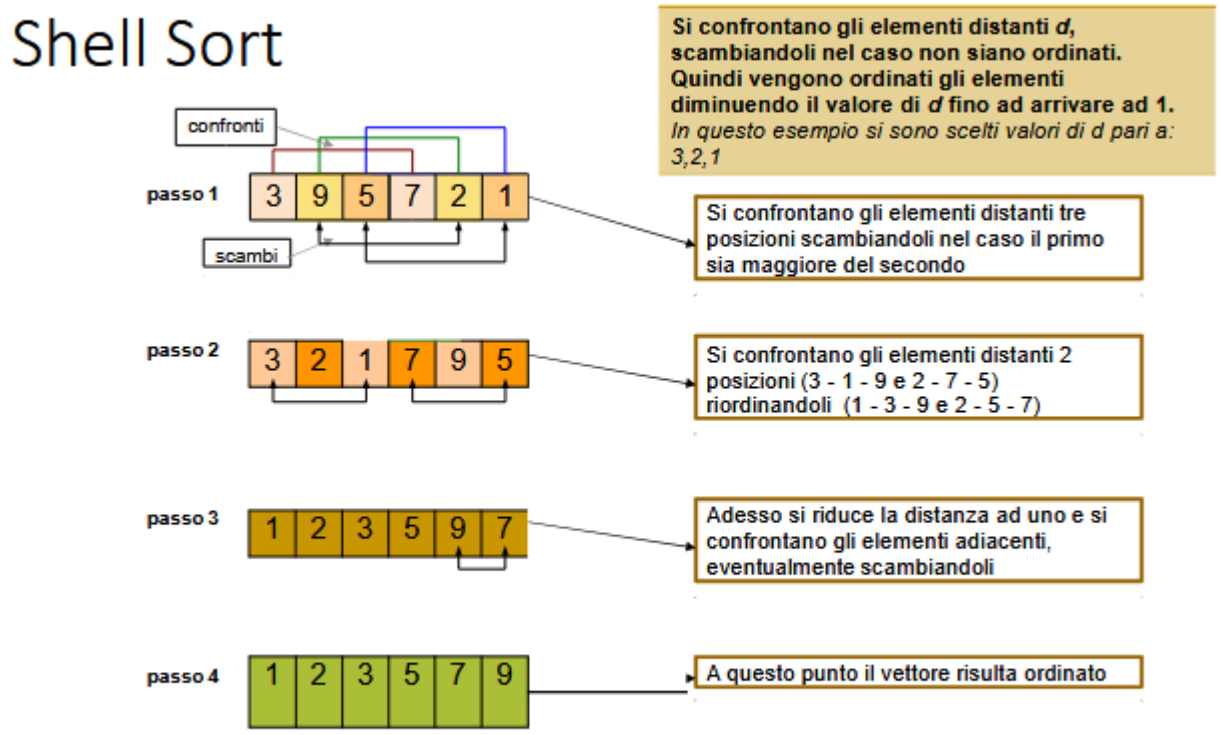
- **Caso ottimo**: la lista è già ordinata, avendo $n - 1$ confronti e 0 scambi, avendo lo stesso risultato del metodo a bolle nel caso ottimale.
- **Caso pessimo**: la lista non è per nulla ordinata, con un totale di confronti e scambi pari a $i - 1$, $\frac{(n-1)*n}{2}$.
- Anche nel **caso medio** la complessità è uguale a quella del caso pessimo, la quale complessità di entrambi è pari a $O(n^2)$.
  Questo algoritmo è **efficace** per piccole sequenze e/o sequenze **parzialmente ordinate**, riducendo così il numero degli scambi, diminuendo anche il tempo di esecuzione, dato il grande peso degli scambi, rispetto al confronto.
  Ad ogni passo la porzione ordinata **cresce di una unità**, differentemente la porzione disordinata **decresce** di una unità.

Algoritmi avanzati di ordinamento

Gli algoritmi di ordinamento di base, come abbiamo notato, hanno un livello di complessità elevata, per risolvere problemi banali della realtà odierna.
E' necessario introdurre degli algoritmi che abbiano una complessità **lineare**.

Shell Sort

Questo è un **algoritmo evoluto**, ed è basato sul concetto di **riduzione degli incrementi**. Si confrontano tutti gli elementi che si trovano ad una **distanza *d*** e si continua **riducendo** il valore di di fino ad arrivare ad elementi adiacenti *d=1*.
Questo algoritmo va a modificare il basico bubble sort, poiché in quest'ultimo si confrontano solo gli elementi adiacenti.



L'ultimo passo sarà identico ad un bubble sort, ma si avrà creato a monte un array pre-ordinato manualmente.

Come si sceglie la distanza?

Valutare il valore di *d* è molto complesso, bisogna tenere a mente che l'ultimo passo deve avere *d* **sempre pari ad 1**.
Le sequenze tipicamente utilizzate sono: 9,5,3,2,1
Per **dogma** si preferisce non usare distanze pari alle potenze di 2.
E' un algoritmo efficiente, poiché **diminuisce** gli scambi da effettuare.

Implementazione Shell Sort

```
void ShellSort(int* vett, int dim){
    int i,j,gap,k;
    int x,a[5]={9,5,3,2,1};
    for(k=0;k<5;k++){
        gap=a[k];
        for(i=gap;i<dim;i++){
            x=vett[i];
            for(j=i-gap;(x<vett[j]) && (j>=0); j=j-gap){
                vett[j+gap]=vett[j];
                vett[j]=x; //scambio elementi
```

```
            }
        }
    }
}
```

Possiamo effettuare più scambi a ciclo, nel primo ciclo si confronta *x* con vett[0], quindi primo elemento con l'elemento pari al gap, ovvero in questo caso di defautl sarà vett[9]. Al ciclo successivo i valori si incrementano, quindi vett[1] con vet[10] (se esiste), ecc...

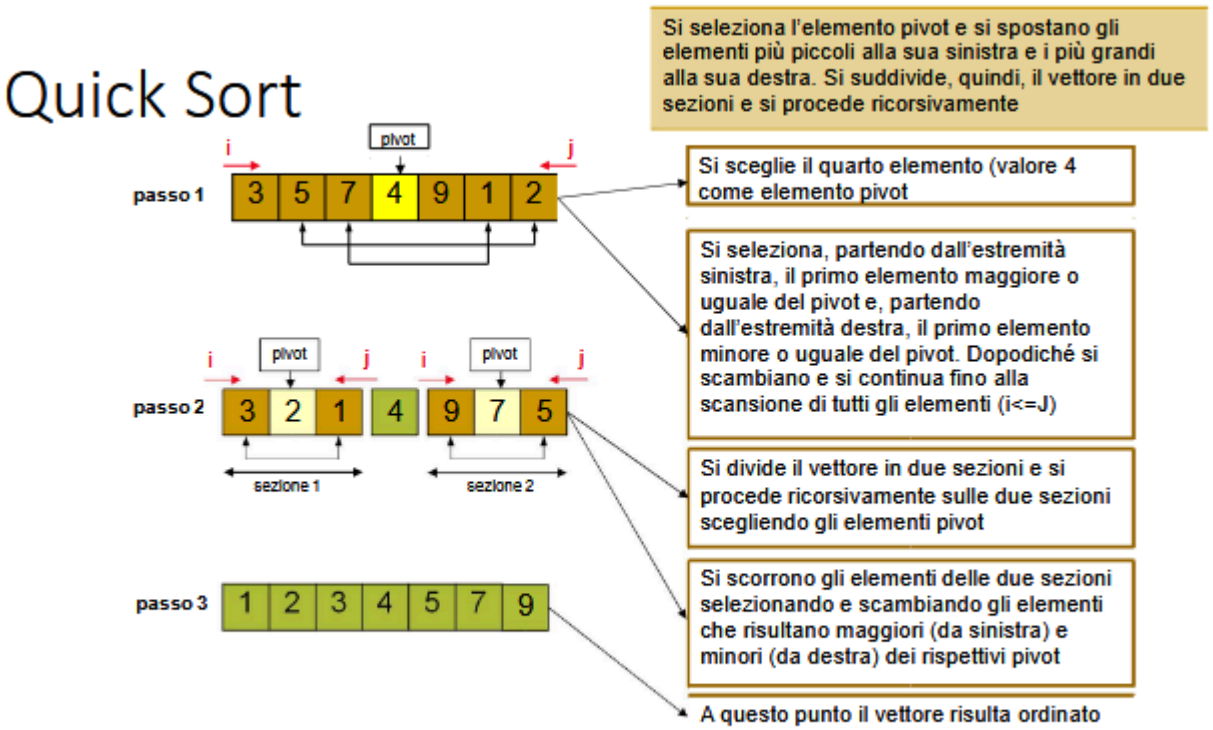
Complessità del shell sort

La complessità media è pari a $O(n \log_2 n)$, ma ciò dipende dalla distribuzione dei dati. La complessità rimanere di questo livello anche nel caso peggiore, quindi tende ad ottenere prestazioni migliori.
Intuitivamente si comprende che gli elementi vengono spostati più **rapidamente**, utilizzando **meno confronti**.

Quick Sort

E' un algoritmo che si basa sulla **ricorsione**, è un algoritmo che richiama sé stesso, sono più semplici ed eleganti.
La sua *CC* è pari a $O(n \log n)$ nel caso ottimo e nel caso medio. Per quanto riguarda il caso peggiore, non si ha una miglioria dagli algoritmi precedenti, ritornando ad un livello esponenziale pari a $O(n^2)$.

Questo algoritmo è definito come un **concetto di partizione**, la procedura generale consiste nella selezione di un valore del vettore analizzato, questo valore viene definito **pivot** e suddividerà il vettore in **due sezioni**, la prima formata da tutti i valori inferiori al pivot e nella seconda sono presenti tutti quelli maggiori.
Questo processo viene ripetuto per ognuna dei settori rimanenti fino all'ordinamento completo.



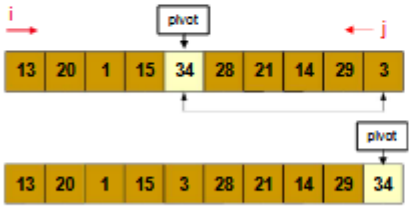
In questo caso ci basiamo sul caso migliore in cui è possibile prendere il pivot (ovvero perfettamente a metà), ma bisogna scegliere con parsimonia se prendere il pivot in base alla posizione (in un array lungo) o in base al valore (in un array breve con valori distinti e non uguali); la scelta del pivot **influisce totalmente** il comportamento del QuickSort.

Analisi del Pivot

Il caso peggiore nella scelta del pivot è la creazione di una scissione formata dalla medesima dimensione del vettore iniziale - 1, mentre l'altro lato ha una dimensione unitaria. Bisogna avere molta **parsimonia** e avvolte anche fortuna e buone conoscenze progettuali per determinare la scelta del pivot. Solo nel caso in cui conoscessimo a priori il vettore possiamo risalire all'**elemento mediano**, ovvero la scelta migliore.

Quick Sort – Caso peggiore

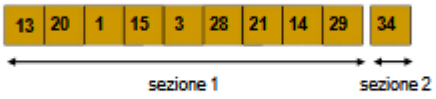
Se scegliamo come pivot l'elemento di posto centrale → indice m=(inf+sup)/2)
Il pivot casualmente coincide con l'elemento massimo del vettore.



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 continuamente incrementato oltre la dimensione del vettore.

Per quanto riguarda l'indice j esso non viene spostato in quanto l'elemento j-esimo è inferiore al pivot.



Siccome **i > j**, la prima passata è finita: otteniamo due sezioni di cui una ha dimensione 1

Il metodo migliore per scegliere il pivot è la **scelta casuale**, selezionando l'elemento che occupa la posizione centrale.
Da questo si evince che non sempre questo algoritmo può essere effettivamente una miglioria, bisogna conoscere e studiare il problema che si sta analizzando per determinare la giusta scelta degli algoritmi da implementare.

QuickSort nel C:

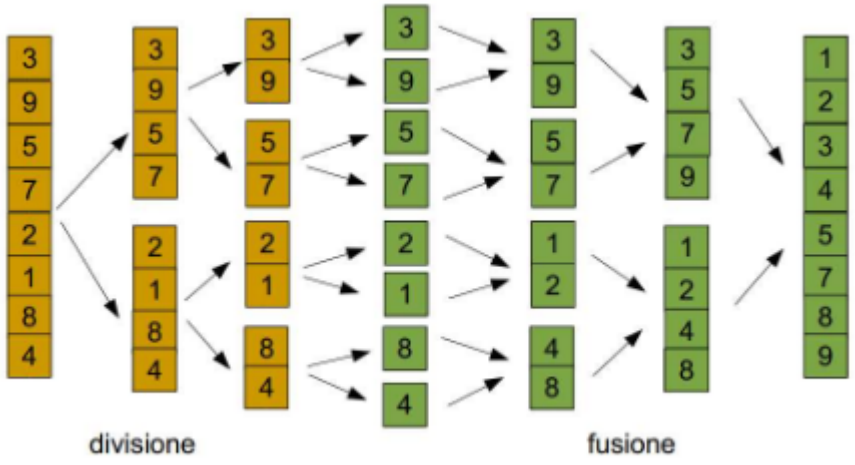
```c
void quickSort(int v[],int l, int r){
    int i;
    if(r<=1) return;
    i=partition(v,l,r);
    quickSort(v,l,i-1);
    quickSort(v,i+1,r);
}

int partition(int v[],int l, int r){
    int x,i,j,temp;
    int p=(l+r)/2;
    x=v[p];
    i=l-1;
    j=r+1;
    while(i<j){
        while(v[--j]>x){
```

```
        while(v[++i]<x){
            if(i<j){
                temp=v[i];
                v[i]=v[j];
                v[j]=temp;
            }
        }
        return j;
    }
}
```

Questa funzione restituisce l'indice *j*, il quale rappresenta il punto in cui termina la metà a sinistra dell'array partizionato, tutti gli elementi da v[0] a v[j] sono minori o uguali a pivot.

Merge Sort

Il merge sort è un algoritmo evoluto di fusione con *CC* pari a $O(n\log(n))$ in tutti i suoi casi. Anche questo come il precedente è un algoritmo ricorsivo, basato sul principio del **divide et impera**, sfruttando il concetto di *merging* degli array ordinati.
Il merge utilizza uno **spazio ausiliario proporzionale a N**, inoltre le risorse di tempo e spazio impiegate **non dipendono dall'ordinamento iniziale** del file di input.



Merge sort nel C:

```
void mergeSort(int a[],int l, int r){
    if(r<=l) return;

    int m=(r+l)/2;
    mergeSort(a,l,m);
    mergeSort(a,m+1,r);

    merge(a,l,m,r);
```

```
}

void merge(int a[],int l, int m, int r){
    int i,j,k,*aux;
    aux=(int*)malloc((r-l+1)*sizeof(int));

    for(i=m+1;i>l;i--){
        aux[i-1]=a[i-1];
            for(j=m;j<r;j++){
                aux[r+m-j]=a[j+1];
                for(k=l;k<=r;k++){
                    if(aux[j]<aux[i]) a[k]=aux[j--];
                    else a[k]=aux[i++];
                }
            }
    }
}
```

Questa implementazione del codice fa uso di un array ausiliario di dimensione proporzionale all'output, per fare ciò il secondo array viene trascritto in maniera inversa alla fine del primo. Nella **prima funzione** si prende un array in input e i due corrispettivi indici dx e sx. La funzione richiama se stessa per poter suddividere in due l'array in base ai casi.

- **Caso base**: se l'indice dx e sx sono uguali o sx è minore di dx, significa che il singolo array è vuoto o contiene un singolo elemento. In questo caso è già ordinato e ritorna immediatamente.
- **Calcolo del punto medio**: il punto medio viene calcolato con $l + r/2$. Questa operazione ci permette di suddividere l'array nelle due metà, da *l* a *m* il primo array e il secondo da *m+1* a *r*.
- La funzione richiama se stessa successivamente per ordinare ricorsivamente le due metà del sotto array.
- Tramite l'uso della funzione merge si andranno ad unire i due nuovi array.
  La **seconda funzione** crea un array di ausilio della stessa dimensione del sotto array. Questo array seve come spazio di lavoro per memorizzare temporaneamente gli elementi durante la fusione.
  Successivamente si andranno a copiare gli elementi da sinistra a destra nel primo ciclo *for*, andando da *m+1* a *l*, copiando nel vettore ausiliare in ordine inverso questi elementi man mano che ci si avvicina a destra. Corrispettivamente nel secondo ciclo *for* andremo a copiare gli elementi da destra a sinistra, sempre in ordine inverso. Infine si ritorna in un ciclo *for* principale che scorre tutta la lunghezza dell'array iniziale, riempiendo l'array

originale con i valori ordinati. Durante questo ciclo vengono confrontati i valori dalla parte sinistra e dalla parte destra del vettore ausiliare.