

Ricerca Lineare

Avendo un Array di 10 elementi faremo in media 5 controlli
(che corrispondono alla metà dell'Array).

La Complessità è lineare per la lunghezza dell'Array ed è $O(n)$

Ricerca Dicotomica

1	2	3	4	5	6	7	8	9	10
---	---	---	---	---	---	---	---	---	----

In modo più semplice veloce e con meno Passaggi è

Se le carte sono disposte in maniera crescente, infatti se sono "ORDINATE" posso

1) Dividere l'Array in due Metà

2) Confrontare con l'elemento centrale:

- se l'elemento cercato è minore ci concentriamo sulla parte Sinistra

- Se l'elemento cercato è maggiore concentriamo l'attenzione sulla parte destra

Codice:

Includiamo le librerie che ci servono

Dopotutto facciamo una funzione che abbia come parametri

l'ARRAY, dimensione, valore da cercare e restituisca un booleano
il booleano lo settiamo a false per vedere se lo trova e iniziamo a scorrere da zero fino alla fine

```
bool ricercaBinaria (int array[], int n, int key)
```

```
{  
    bool found = false;  
    int start = 0;  
    int end = n;
```

Succesivamente serviamo un ciclo while che redete le istruzioni finché il valore è stato trovato, o finché l'inizio
corrisponde alla fine.

Nel ciclo while inizializziamo l'elemento mediano sommando l'inizio alla fine meno l'inizio fatto 2.

Controlliamo se l'elemento mediano corrisponde all'elemento cercato, in quel caso il valore booleano found diventerà true.

```

while (!found && (start != end))
{
    int midpoint = start + ((end - start)/2);
    cout << "start = " << start << ", end = " << end;
    cout << ", midpoint = " << midpoint << endl;
    if (array[midpoint] == key)
        found = true;
}

```

All'ultimo controlloiamo se il valore è minore dell'elemento mediano e il punto di inizio sarà l'elemento mediano o se è maggiore dell'elemento mediano e il punto finale sarà l'elemento mediano

```

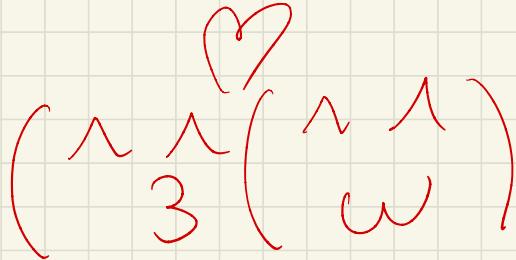
else if (key < array[midpoint]) {end = midpoint;}
else {start = midpoint+1;}
Al fine torneremo l'elemento booleano
return found;

```

```

int main()
{
    int array[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    cout << ricercaBinaria(array, 10, 7);
}

```



SEI IL MIGLIORE

QUALE ELLÈ DI TUTTO B

L' Ordinamento:

Diversi tipi di ordinamento essi si dividono in Algoritmi iterativi e ricorsivi.

Tipi di ordinamento:

ORDINAMENTI ITERATIVI:

Ordinamento per Scambio:

Averno un Array, Confrontero' il primo elemento con il successivo ed effettuo lo scambio se l'elemento successivo è più piccolo.

Selection Sort:

Averno un Array, lo scorro una prima volta dall'inizio alla fine per individuare l'elemento più piccolo che andrò ad inserire in posizione 0.

Successivamente scorrerò l'array incrementando di volta in volta il punto iniziale del confronto.

Insertion Sort:

Considero un sottoarray con cui fare il confronto per riconoscere la posizione del numero da inserire nell'Array più grande. Di volta in volta il sottoArray verrà incrementato di uno.

Codice:

INSERTION SORT

```
void insertionSort ( int v[], int n )  
{
```

```
    for ( int j = 1; j < n; j++ )  
    {
```

```
        int key = v[j];
```

```
        int i = j - 1;
```

```
        while ( i >= 0 && v[i] > key )
```

```
        {
```

```
            v[i + 1] = v[i];
```

```
            i--
```

```
        }
```

```
        v[i + 1] = key
```

```
}
```

Codice:

SELECTION SORT:

```
void selectionSort ( int v[], int n )  
{
```

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

```
        int idx = i;
```

```
        for ( int j = i + 1; j < n; j++ )
```

```
        {
```

```
            if ( v[j] < v[idx] )
```

```
                idx = j;
```

```
        }
```

```
        swap ( v[i], v[idx] );
```

```
}
```

Ordinamento Ricorsivo:

Per gli Ordinamenti Ricorsivi si utilizza la strategia **dividi et impera**

- 1) **Fraziono** l'array in sezioni sempre più piccole fino ad arrivare ad un caso base dato un Array di un solo elemento
- 2) **Considero** l'array di un solo elemento come già ordinato (caso base)
- 3) **Passo fondamentale** nonché "cuore della strategia", la combinazione degli Array

Merge Sort:

Divido l'Array in due metà fino ad ottenere degli array con un singolo elemento, dopodiché presi gli elementi singolarmente li metto in ordine prima per la parte a sinistra, poi per la parte a destra e infine globalmente

Quick Sort:

Considerati tre indici (uno all'inizio, uno che lo precede e uno alla fine dell'Array) e considerato un elemento a caso il (pivot), faccio Scorrere l'indice iniziale e non appena trovo un valore minore del pivot, invertendo il valore in posizione i con il valore in posizione j (valore puntato dall'indice iniziale e valore puntato dall'indice che lo precedeva). Alla fine sostituisco il pivot con il valore in posizione $i+1$.

Ordinamento per Scambio

10	8	7	6	4	9	2	3	5	1
----	---	---	---	---	---	---	---	---	---

- 1) prendo una carta in mano lasciando libera una posizione
- 2) se la carta i -esima è più piccola della carta che ho in mano, allora scambio le posizioni, quindi zero in mano una carta più piccola rispetto a quella che avevo in precedenza
- 3) altrimenti non scambio gli elementi
- 4) quando le carte finiscono prendo la carta successiva e ripetere i passaggi

Codice:

Sono presenti due for che devono essere di volta in volta scelti per considerare i vari elementi. L'indice del primo for sarà inizializzato a 0, mentre l'indice del secondo for sarà inizializzato a $i+1$. All'interno del secondo for sarà presente una condizione per cui se essa è rispettata si effettuerà lo swap
vad ordinamento (int array[], int n){}

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

```
    for(int j=i+1; j < n; j++) {
```

```
        if (array[i] < array[j]) {
```

```
            Swap (array[i], array[j]);
```

```
        }
```

```
    }
```

Insertion Sort

Arr. Disordinato verifichiamo il primo elemento lo mettiamo a sinistra il secondo è più piccolo(?) va shiftato il primo elemento a destra.

- Algoritmo di ordinamento

→ Scorsione della sequenza in input elemento per elemento

→ Inserimento dell'elemento considerato, nella posizione corretta all'interno della sequenza fino a quel momento ORDINATA



↑
- Ora vediamo che il 5 è più piccolo del 12 perché vero inserito prima e poi si avrà un vettore DIM 2 perché non abbiamo altri elementi.
[5 | 12 | 3 | 4] da confrontare nella Sottosequenza

- Ora abbiamo che il 3 è più piccolo di 12 e stiamo in avanti poi altro confronto ed il 3 è più piccolo del 5 e viene messo all'inizio shiftando il 5.



```
void insertionSort (int v[], int n)  
{
```

```
    for (int j = 1; j < n; j++)
```

```
    {
```

```
        int Key = v[j];
```

```
        int i = j - 1;
```

```
        while (i >= 0 && v[i] > key)
```

```
        {
```

```
            v[i + 1] = v[i];
```

```
            i--;
```

```
        }
```

```
        v[i + 1] = key;
```

```
}
```

Complessità

Migliore $O(n)$

Media $O(n^2)$

Peggiorre $O(n^2)$

Selection Sort

Scorreremo vediamo dove è il minimo e swap diamo

- Algoritmo di ordinamento per una sequenza di n elementi, basato sul seguente funzionamento

→ Selezione dell'elemento più piccolo degli n elementi e suo inserimento in prima posizione al posto dell'attuale primo elemento che invece andrà nella posizione rispettivamente precedentemente dell'elemento più piccolo;

→ Applicazione del passo precedente per i restanti n-1 elementi poi sugli n-2 elementi e così via.

EX:

5	8	(1)	4	13
---	---	-----	---	----

- fissiamo l'elemento con i e scorreremo con j fino a trovare un elemento più piccolo se lo troviamo effettuiamo lo swap

- il min è uno piccolo lo scambieremo col 5 (elemento fissato da i)

1	(8)	5	4	3
i	i	j		

• il 3 < il minore, 8 passa più a sinistra

1	3	(5)	(4)	8
			n-2 elementi	

il 4 è più piccolo ordiniamo effettuando lo SWAP ed è ordinato.

1	3	4	5	8
---	---	---	---	---

Codice:

```
void SelectionSort (int v[], int n)
```

{

```
for (int i=0; i<n; i++)
```

{

```
    int idx = i;
```

```
    for (int j=i+1; j<n; j++)
```

{

```
        if (v[j] < v[idx])
```

```
            idx = j;
```

}

```
        Swap (v[i], v[idx]);
```

}

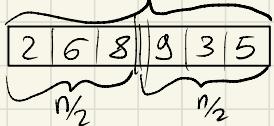
Complessità

$O(n^2)$ SEMPRE

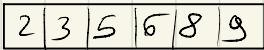
Merge Sort:

Algoritmo di ordinamento su n -elementi basato sulla seguente idea.

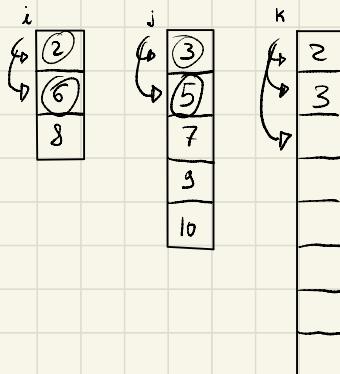
- Dividere gli n elementi in 2 parti di $\frac{n}{2}$ elementi



Considerare che le due parti chuse di $\frac{n}{2}$ elementi, sono ordinate e procedere con la loro fusione



ex



Confrontiamo i e j , chi è minore? In questo caso (i) perciò salviamo il valore su k e incrementiamo su (i) che (j) si effettueranno i confronti e valuteremo chi è minore, in questo caso (j), lo salviamo in k e incrementiamo (j) e (i) e così via fino alla fine degli elementi.

Merge Sort → Divide et Impera

Divide

- 1) Si procede alla suddivisione dei problemi in problemi di dimensione minore

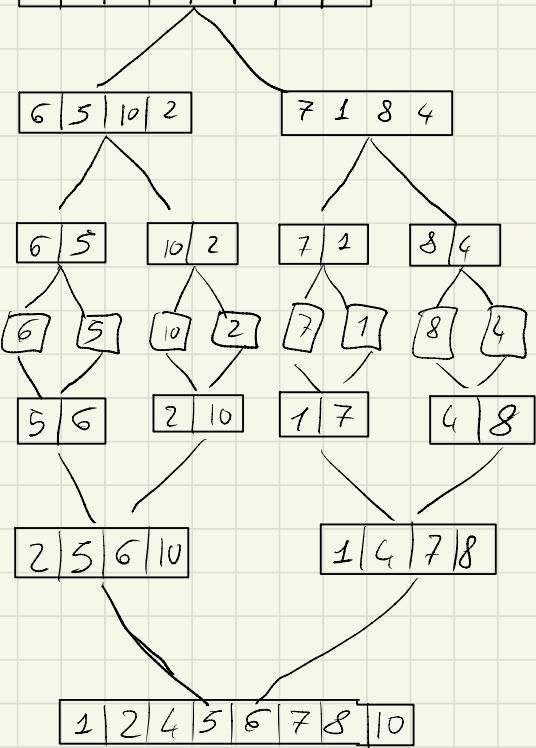
Impero

- 2) → i problemi vengono risolti in maniera ricorsiva. Quando il problema arriva ad avere una dimensione sufficientemente piccola si può usare il caso base

Combina

- 3) → Si combina l'output ottenuto dalle varie chiamate ricorsive alla fine di ottenere il risultato finale

6	5	10	2	7	1	8	4
---	---	----	---	---	---	---	---



- Nel momento della fusione si confrontano i valori e li inseriamo ordinati.

Complessità: $O(n \log n)$

Quick Sort (fuori in place [utilizza un solo Array])

La scelta del pivot influenzava l'efficienza dell'algoritmo.

Algoritmo di ordinamento ricorsivo, basato sul paradigma del Divide et Impera, come il Merge Sort.

Divide → Si sceglie un elemento chiamato "pivot" e si partitiona il vettore in due sottovettori:

il primo con elementi più piccoli del pivot e il secondo con elementi più grandi o uguali al pivot.

Impera → si chiama ricorsivamente l'ordinamento di quickSort prima sul vettore principale e poi man mano nei sottovettori.

Combinazione → non è necessario questo step, il vettore ricombinato sarà già ordinato.

Scegliere:

Per eseguire il quick sort consideriamo tre indici:

- p, avro l'indice iniziale
- q, avro un indice generico compreso tra p e r
- r, avro l'indice finale

Inoltre debbiamo considerare anche altri due indici:

- i, avro l'indice corrispondente a p-1;
- j, avro l'indice corrispondente a p;

Edu infine consideriamo un elemento a caso (soltanmente è considerato l'elemento alla fine dell'array, ma comunque non facciamo considerazioni sul valore effettivo) chiamato PIVOT.

A questo punto procederemo nel seguente modo:

Faccio scorrere l'indice j (dall'inizio alla fine) confrontando di volta in volta l'elemento in posizione con il pivot.

Se l'elemento dovesse essere MINORE del pivot SWAPERO l'elemento in posizione i con l'elemento in posizione j (se sono uguali non li scambiero)

ed incrementerò i. Una volta che avro scambiato j con i, tutto l'array scambio il pivot con l'elemento in posizione i+1.

Alla fine avrò che tutti gli elementi a sinistra del pivot sono minori e tutti quelli a destra sono maggiori.

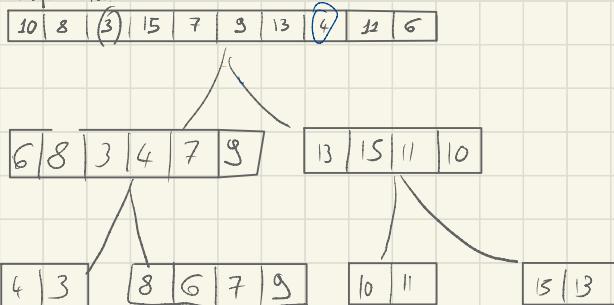
Se il pivot è circa PRIMI ALTA METÀ del valore massimo degli elementi, allora la procedura lavorerà in maniera ottimale e sarà eseguito il procedimento in un tempo pari a $(\log n)$, quindi la complessità sarà di $O(n \log n)$.

Se invece il pivot corrisponde al valore MINIMO o al valore MAXIMO la complessità sarà $O(n^2)$. Questo perché l'operazione che provoca un aumento della complessità di un algoritmo è quello di scambio quindi meno scambi saranno eseguiti e più "leggero" sarà il programma.

Codice:

```
int partition (int vet[], int start, int end) {  
    int pivot = vet[end];  
    int i = start - 1;  
    for (int j = start; j < end; j++) {  
        if (vet[j] <= pivot)  
            Swap (vet[i], vet[j]);  
        i++;  
    }  
    Swap (vet[i+1], vet[end]);  
    return i+1;  
}
```

Esempio GoFico:



Riapplichiamo fino ad avere vettore Dim(1)

Pivot maggiore con keli!

Void quick_sort (int vet[], int start, int end) {

```
if (start < end) {  
    int pivot = partition (vet, start, end);  
    quick_sort (vet, start, pivot-1);  
    quick_sort (vet, pivot+1, end);  
}
```

void quick_sort (int vet[], int n) {

```
// richiamo passando il vettore, l'indice del primo e dell'ultimo elemento  
quick_sort (vet, 0, n-1);
```