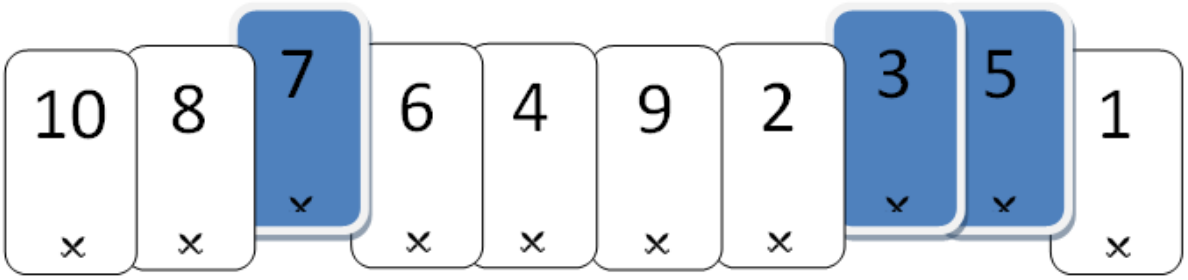
Un gioco con le carte

Prima di parlare di cos'è un ordinamento e di quali sono i tipi di ordinamenti, immaginiamo di voler fare qualche gioco con le carte. Prendiamo in particolare le carte siciliane dello stesso seme (*quindi abbiamo un array di 10 elementi*) e facciamo il seguente gioco

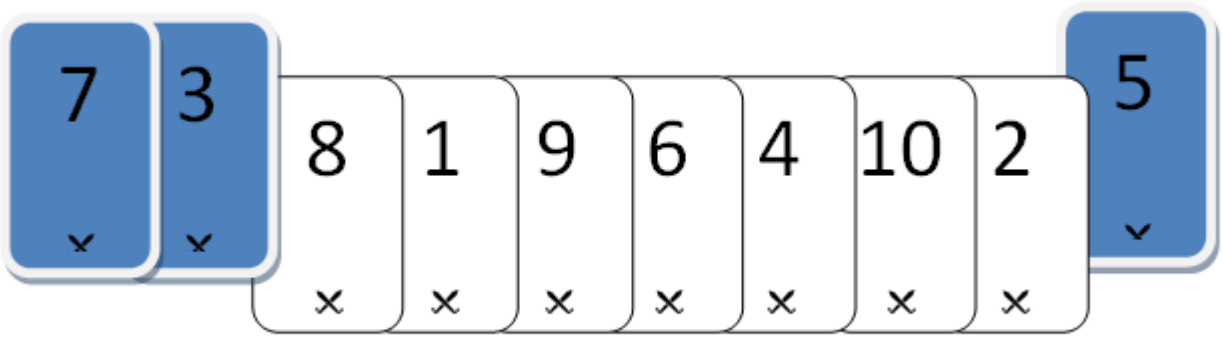
La ricerca lineare

Disposte 10 carte in maniera randomica sul tavolo, si cercano il numero 5, il numero 3 e il numero 7.



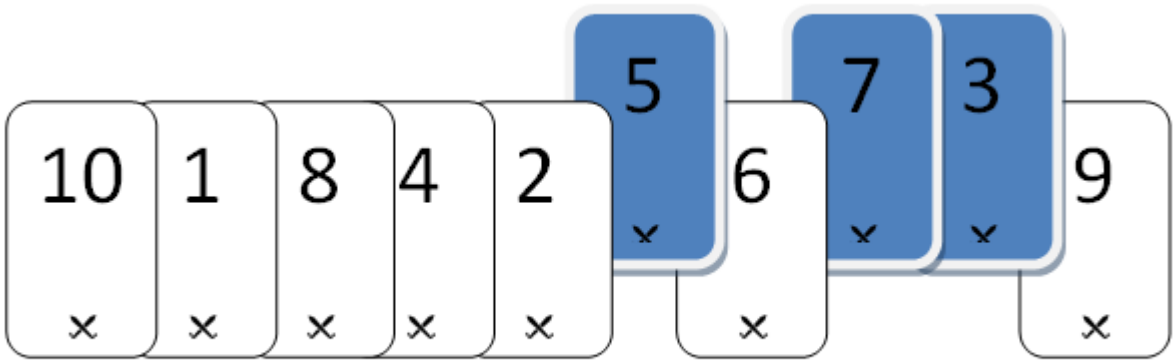
Per trovare il 5 faccio 9 controlli
Per trovare il 3 faccio 8 controlli
Per trovare il 7 faccio 3 controlli

...mescolo le carte e ottengo...



Per trovare il 5 faccio 10 controlli
Per trovare il 3 faccio 2 controlli
Per trovare il 7 faccio 1 controllo

...mescolo le carte e ottengo...



Per trovare il 5 faccio 6 controlli
Per trovare il 3 faccio 9 controlli
Per trovare il 7 faccio 8 controlli

> In media quindi facciamo circa 5 controlli (che corrispondono alla *metà dell'array*).
> La complessità è lineare per la lunghezza dell'array e corrisponde a O(n)

La ricerca dicotomica (*o binaria*)

Un modo più semplice (e veloce, con meno passaggi) per trovare un numero lo si può avere se le carte sono disposte in ordine crescente. Se le carte difatti sono disposte in maniera "ordinata" posso:

1. dividere l'array in due metà
2. confrontare l'elemento cercato e l'elemento centrale:
   - se l'elemento cercato è minore concentro l'attenzione sulla parte a sinistra
   - se l'elemento cercato è maggiore concentro l'attenzione sulla parte a destra
3. valuto l'elemento cercato con i restanti numeri, reiterando la procedura

Codice

La prima cosa da fare è includere le librerie necessarie

```C++
#include<iostream>
using namespace std;
```

Dopodiché facciamo una funzione che abbia come parametri un array, il numero di elementi e il valore cercato e che restituisca un valore booleano corrispondente a true se l'elemento è stato trovato. Inizializziamo bool a false, l'inizio a zero e la fine a n (numero di elementi presenti nell'array)

```C++
bool ricercaBinaria(int array[], int n, int key){
    bool found = false;

    int start = 0;
    int end = n;
```

Successivamente scriviamo un ciclo while che reitera 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 fratto 2. Controlliamo se l'elemento mediano corrisponde all'elemento cercato, in quel caso il valore booleano found diverrà true

```C++
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;
```

Altrimenti controlliamo 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

```C++
else if (key<array[midpoint]) { end = midpoint; }
else {start = midpoint+1; }
```

Alla fine della funzione torneremo il valore booleano

```C++
    return found;
```

Main

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

L'ordinamento

Esistono diversi modi per ordinare una struttura dati (noi consideriamo gli array). Essi si dividono in algoritmi iterativi e ricorsivi. Riprendendo il gioco delle carte, illustriamo brevemente le diverse tipologie per poi andare ad analizzarle in dettaglio.

Tipi di ordinamenti

Ordinamenti iterativi

- **ORDINAMENTO PER SCAMBIO**: avendo un array, confronterò il primo elemento con il successivo ed effettuo lo scambio se *l'elemento successivo è più piccolo*
- **SELECTION SORT**: avendo un array, lo scorro una prima volta dall'inizio alla fine *per individuare l'elemento più piccolo* che andrò a inserire in posizione 0. Successivamente scorrerò l'arrat incrementando di volta in volta il punto iniziale del controllo
- **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

Ordinamenti Ricorsivi

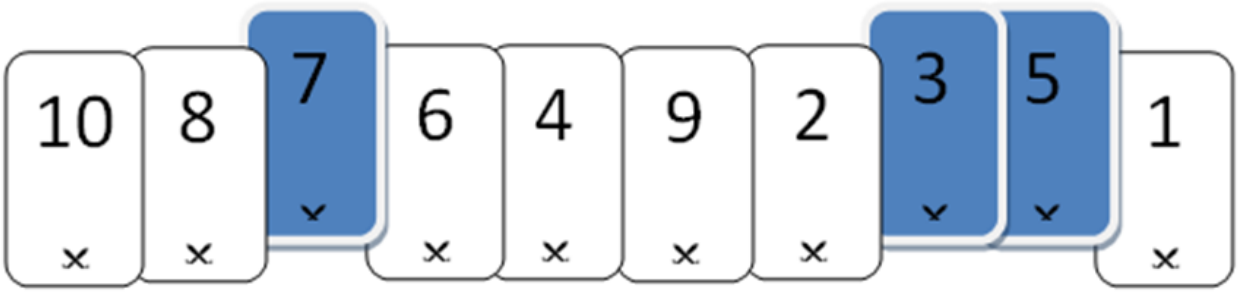
Per gli ordinamenti ricorsivi si utilizza la strategia divide et impera (già più volte citata). Difatti per fare gli ordinamenti farò:

1. Fraziono l'array in sezioni sempre più piccole fino ad arrivare ad un caso base dato da un array di un solo elemento *(suddivisione del problema)*;
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 ed 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, inverto 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



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 avrò 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 reitero i passaggi

In totale si faranno dai 10 ai 90 passaggi.

Codice

Nel codice sono presenti due `for` che devono essere di volta in volta scorsi 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`.

C++

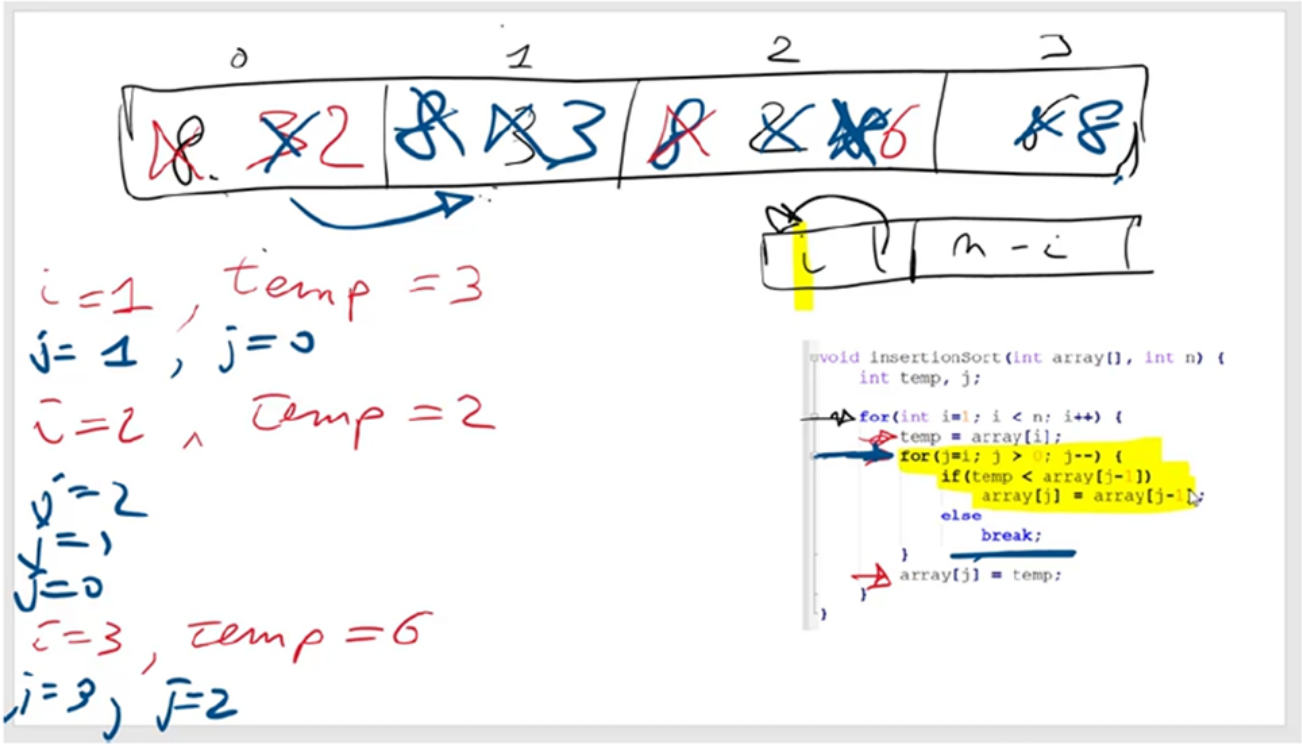
```
void ordinamento (int array[], int n){
    for(int i = 0; i<n; i++){
        for(int j = i+1; j<n; j++){
            if (array[j] < array[i]){
```

```
                swap (array[i], array[j]);
                }
            }
        }
    } ```

__La complessità è pari a O(n2).__


## Selection sort
![[Pasted image 20220711175033.png]]

1. prendo la prima carta e cerco l'elemento minore nei successivi
2. stavolta trovato l'elemento minore non effettuo scambi, ma _tengo "a mente" l'indice_ del valore più piccolo
3. reitero il procedimento considerando l'elemento successivo

Possiamo paragonare il selection sort ad un ordinamento per scambio, ma è preferibile utilizzare il selection poiché il suo utilizzo ci permette di effettuare lo **scambio una singola volta**.

Inoltre, anche il selection sort (come l'ordinamento per scambio) ha una complessità pari a _O(n^2)_; ciononostante i due differiscono per via delle dimensioni minori delle costanti del selection sort.

### Codice
Definisco una funzione alla quale _passo come parametri un `array` e la dimensione di quest'ultimo_. Nella funzione utilizzo `due for` che devono essere di volta in volta scorsi 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**. Nel **primo** for _inizializzo la variabile_ `indexMin` _ad i_. Nel **secondo** for controllo se l'_elemento di posizione j è minore dell'elemento di posizione indexMin_, in questo caso `modifico indexMin`. Alla fine – sempre nel primo for – utilizzerò l'istruzione `swap` col valore di posizione i e quello di posizione indexMin.
```

```cpp
    void selectionSort(int array[], int n){
        int indexMin;
        for(int i = 0; i<n-1;i++){
            indexMin = i;
            for (int j = i+1; j<n; j++){
                if(array[j] < array[indexMin]){
                    indexMin = j;}
            }
        swap(array[i], array[indexMin]);}
    } ```


## Insertion sort


1. pesco una carta e  la metto in posizione 0 (_tenendo a mente che un vettore di un solo elemento è considerato ordinato)_

2. pesco un'altra carta e individuo la suaposizione corretta rispetto al sottoarray preso in considerazione (in cui è presente un'unica carta). Pertanto ad ogni passaggio la dimensione aumenterà di uno

3. pesco la carta successiva e reitero il secondo passaggio, finché le carte non terminano

### Codice
Definisco una **funzione insertion_sort** alla quale _passo come parametri un array e la dimensione di quest'ultimo_. All'interno della funzione inizializzo due interi: `j` e `temp` che saranno utili all'interno dei for successivi. Nella funzione utilizzo `due for` che devono essere di volta in volta scorsi per considerare i vari elementi. L'indice del `primo for` sarà **inizializzato a 0**, mentre l'indice del `secondo for` sarà **inizializzato a i**. Nel **primo for** inizializzo la variabile `temp` con _l'elemento presente nell'array all'indice i_, nel **secondo for** – invece – controllo se la variabile in posizione j-i è maggiore rispetto alla
```

temp e in quel caso inserisco al posto j il valore di posizione j-1.

```cpp
    void insertion_sort(int* vett, int dim){
        int j;
        int temp;

        for(int i = 1; i<dim; i++){
            temp = vett[i];
            for(j = i; j>0; j--){
                if(temp<vett[j-1]){
                    vett[j] = vett[j-1];
                }
                else break;
            }
            vett[j] = temp;
        }
    }
```



La parte del codice evidenziata può essere sostituita con una ricerca binaria (*dicotomica*) dato che la parte a sinistra dell'array che si prende in considerazione è

già ordinata.

Inoltre, anche l'insertion sort ha una complessità pari a O(n^2);

Merge sort

Per eseguire il merge sort consideriamo tre indici:

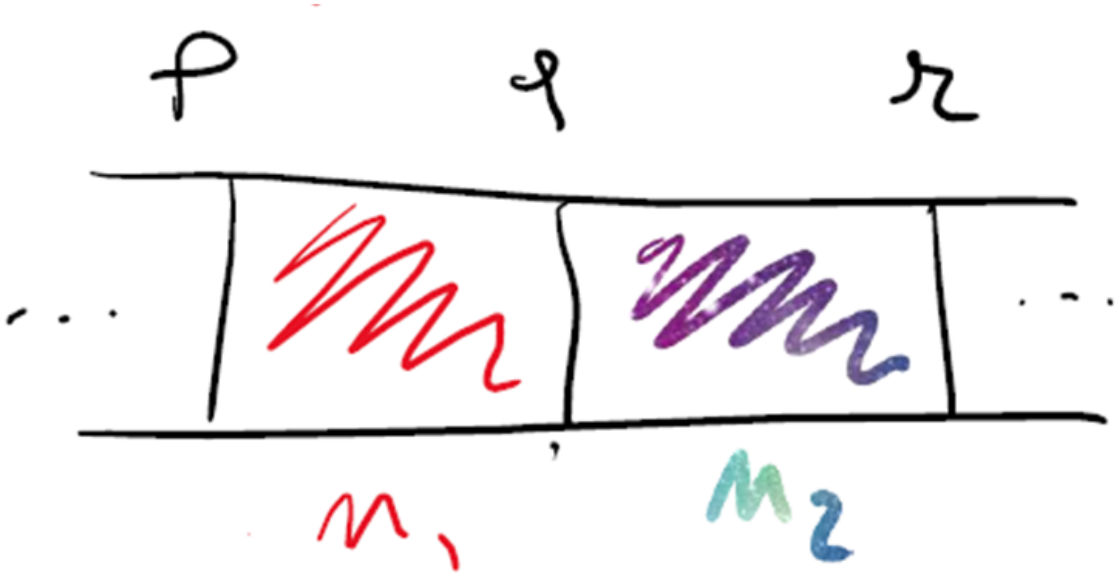
- **p**, ovvero l'indice *iniziale*;
- **q**, ovvero l'indice *mediano*;
- **r**, ovvero l'indice *finale*.

e due array:

- uno a sinistra di dimensione **n1** q-p+1
- uno a destra di dimensione **n2** r-q

i quali quindi:

- **L** sarà composto dagli elementi di A che vanno *da p a q*;
- **R** sarà composto dagli elementi di A che vanno *da q+1 a r*.

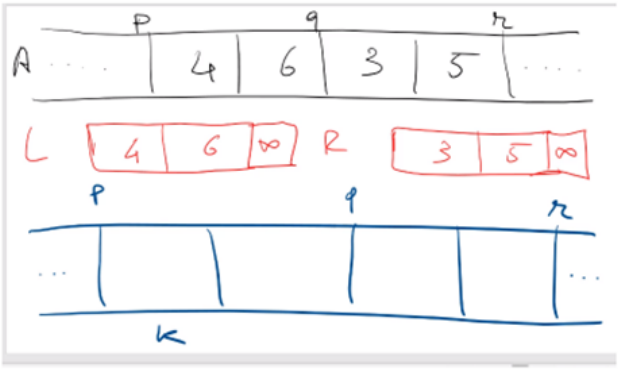


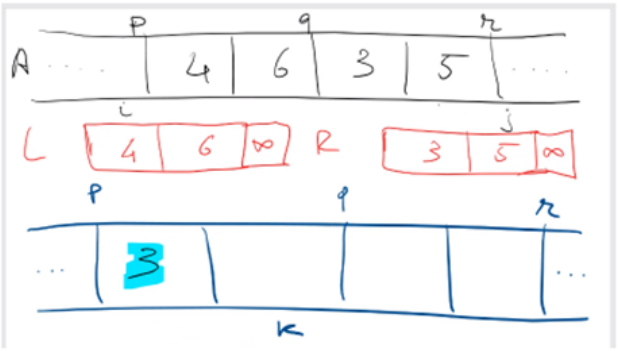
Inoltre aggiungiamo due "sentinelle" (per indicarne la fine) nei due array:

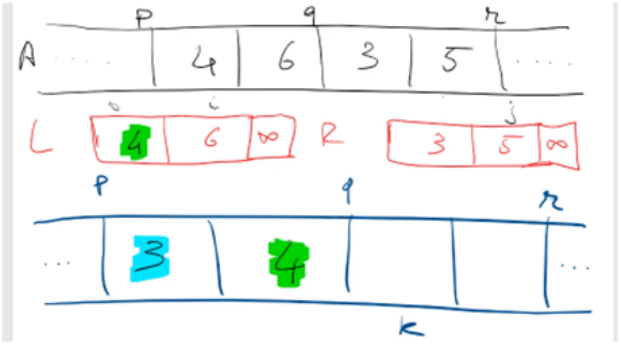
- L[n1+1]= ∞
- R[n2+1] = ∞

Inizializzo dopodiché tre indici (`i`, `j`, `k`) che *scorreranno A, L, R*. Dopodiché **scorro** di volta in volta l'array *di sinistra* e quello *di destra* coi rispettivi indici **e li inserisco**

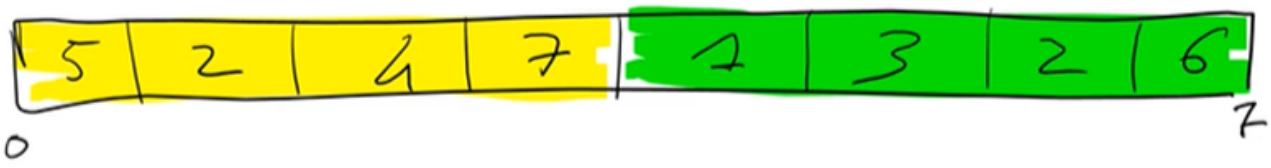
in un nuovo array. Una volta aggiunti i valori, incremento l'indice del sottoarray dal quale si è preso l'elemento.







Esempio



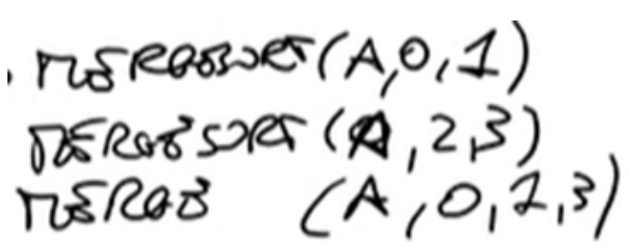
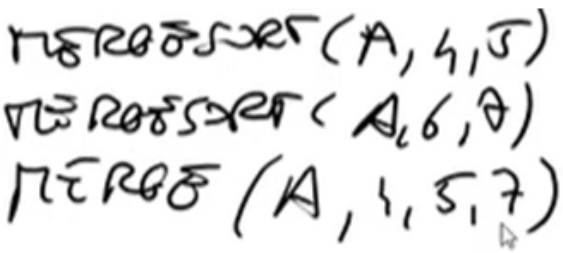
MERGESORT(A , 0 , 7) , $q = \lfloor \frac{7+0}{2} \rfloor = 3$

MERGESORT (A , 0 , 3)

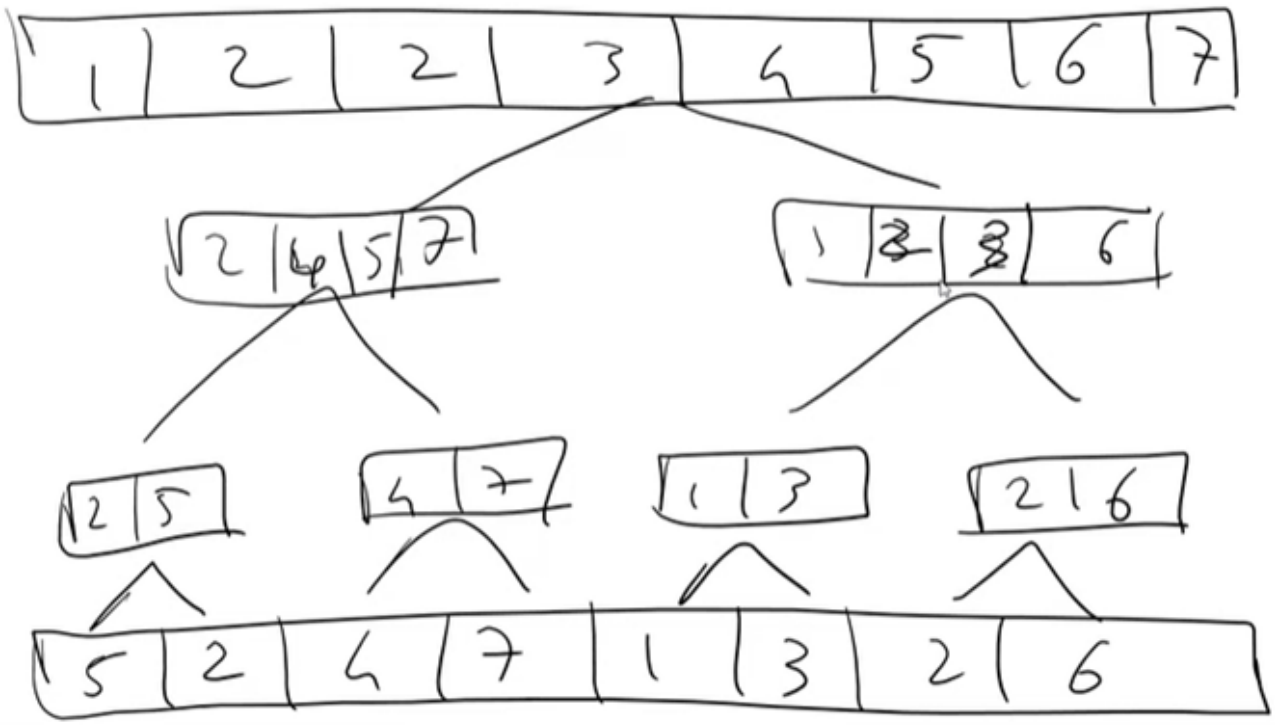
MERGESORT (A , 4 , 7)

MERGE (A , 0 , 3 , 7)

:Parte a sinistra	Parte a destra	:

:Parte a sinistra	Parte a destra :
	

Per passare alla chiamata ricorsiva è necessario prima aver terminato la chiamata corrente. Una volta giunti alla fine (*caso base*) si avrà una combinazione data dallo scambio dei valori. Quindi avrò:



Ovviamente dato che mi servo di due array (una copia ordinata e una cpia suddivisa a metà), il merge sort avrà un rilevante peso sulla complessità spaziale.

In totale ho quindi 8 numeri ed ho 3 livelli di ordine, quindi farò log2n passaggi.

Pertanto **la complessità sarà di Θ(n(log2n))**. Mentre la complessità del `merge` è di $\Theta(n)$.

Codice

Definisco la funzione `merge` alla quale *passo come parametri l'array, la dimensione dell'array, l'indice iniziale e l'indice mediano.* È importante che siano passati come parametri **tutti** quelli che sono **modificati in maniera ricorsiva**. Come primi due passaggi inizializzeremo le dimensioni dei due sotto array.

```C++
void merge(int* vett, int p, int q, int r){
    int n1 = (q-p) + 1;
    int n2 = r-q;
```

Successivamente istanzio `due array` di supporto e li *alloco in maniera dinamica*. Il **primo** array corrisponde al *sotto array di sinistra* di A pertanto dovranno essere ***inseriti gli elementi nella parte sinistra di A***. Il **secondo** corrisponde al *sotto array di destra* di A pertanto dovranno essere ***inseriti gli elementi nella parte destra di A***.

```C++
int* L = new int[n1+1];
int* R = new int[n2+1];

for (int i = 0; i<n1; i++){
    L[i] = vett[p+i]; // riempimento array L
}

for (int j = 1; j<n2; j++){
    R[j-1] = vett[q+j]; // riempimento array R
}
```

Successivamente inserisco in `posizione n1` di L e in `posizione n2` di R un valore che corrisponderà all'infinito, cioè `INT_MAX`. Esso rappresenterà la **sentinella** necessaria per segnalare il *termine dell'array*.

```C++
L[n1] = INT_MAX;
R[n2] = INT_MAX;
```

Utilizzerò quindi `un for` (il cui *indice sarà inizializzato all'indice iniziale*) per **inserire di volta in volta in un array ordinato gli elementi**. In particolare se il

valore a sinistra è minore inserirò quel valore nell'array A e incrementerò l'indice i. Se il valore a destra è minore inserirò quel valore nell'array A e incrementerò l'indice j.

```C++
for(k = p; k<=r; k++){
    if(L[i] < R[j]) {
        vett[k] = L[i];
        i++;
    }
    else {
        vett[k] = R[j];
        j++;
    }
}
```

Infine deallocherò i due array, dato che erano stati allocati in maniera dinamica.

```C++
delete[] L;
delete[] R;
```

Definisco ora una funzione `mergesort` alla quale *passo come parametri un array, il suo indice iniziale e il suo indice finale.* All'interno della funzione inseriamo una **condizione per cui se l'indice iniziale è minore di quello finale** troverò un valore q, effettuerò il mergesort per la parte a sinistra e per la parte a destra ed infine il merge.

```C++
void mergesort(int* vett, int p, int r){
    if(p<r){
        int q = floor((p+r)/2.0);

        mergesort(vett, p, q);
        mergesort(vett, q+1, r);
        merge(vett, p, q, r);
    }
}
```

Quick Sort

Il quick sort può essere considerato come un *algoritmo che lavora sul posto*. Questo perché a differenza del merge sort non utilizzerà più di un array, ciò dunque evita l'impatto sulla complessità spaziale.

Per eseguire il quick sort consideriamo tre indici:

- **p**, ovvero l'indice *iniziale*;
- **q**, ovvero un indice generico compreso tra p e r
- **r**, ovvero l'indice *finale*

Inoltre dobbiamo considerare anche altri due indici:

- **i**, ovvero l'indice corrispondente a *p-1*;
- **j**, ovvero l'indice corrispondente a *p*.

Ed infine consideriamo un elemento *a caso* (solitamente è considerato l'elemento corrispondente 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 **scambierò** l'elemento in posizione i con l'elemento in posizione j (se i due elementi dovessero essere uguali non li scambierò) ed incrementerò i. Una volta che avrò scorso con j tutto l'array scambio il pivot con l'elemento in posizione i+1.

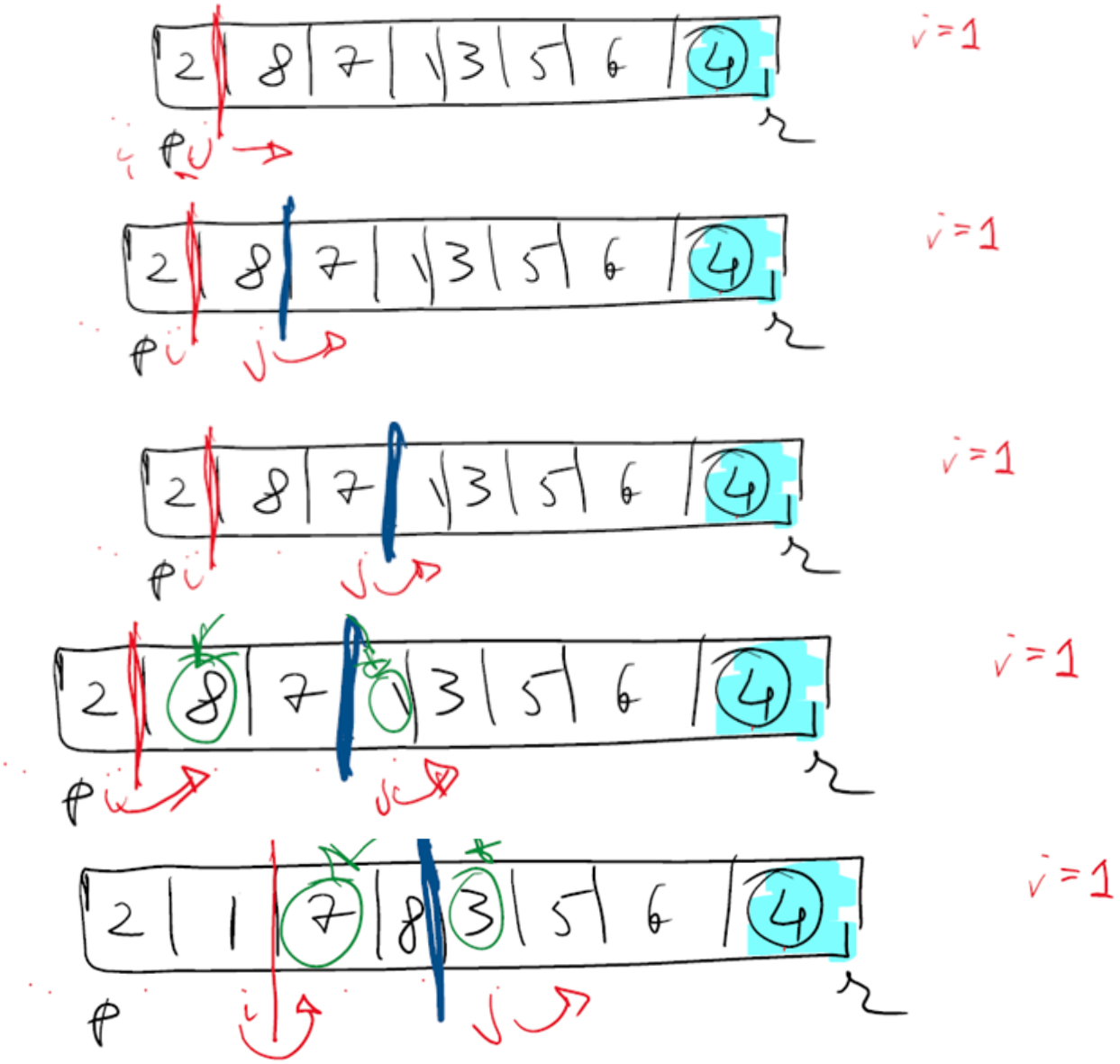
Alla fine avrò che tutti gli elementi a sinistra del pivot sono minori e tutti gli elementi a destra sono maggiori.

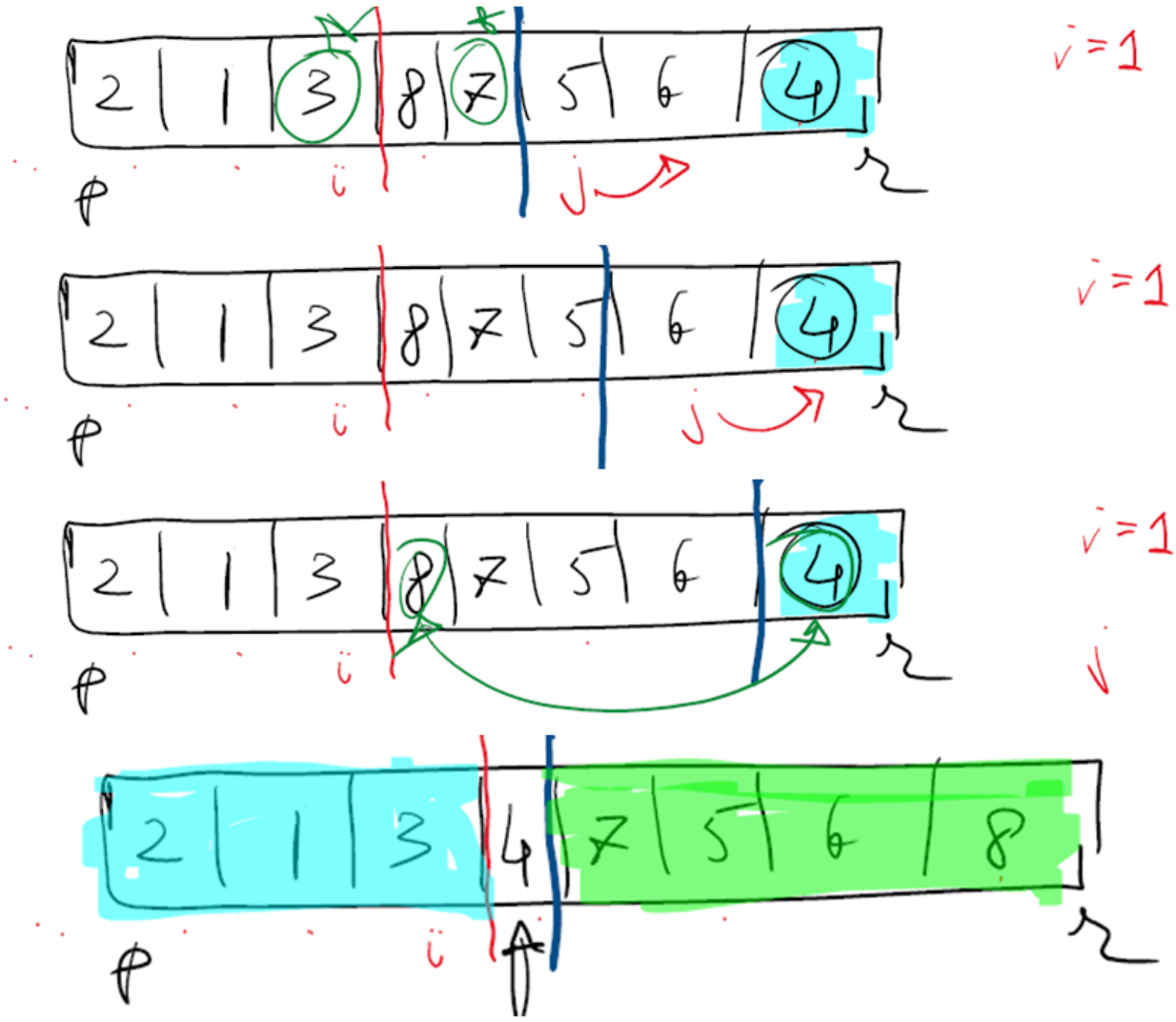
Se il **pivot è circa pari alla metà** del valore massimo degli elementi, allora la procedura lavorerà in maniera ottimale e sarà eseguito il procedimento in un tempo pari a logn, quindi **la complessità sarà di O(logn)**.
Se invece **il pivot** corrisponderà **al valore massimo o al valore minimo la complessità sarà quadratica**. Questo perché ***l'operazione che provoca un aumento della complessità di un algoritmo è quella di scambio***, quindi meno scambi saranno eseguiti e più "leggero" sarà il programma.** (Quindi minore sarà la complessità).

Esempio 1

Per comprendere bene il procedimento del quick sort possiamo fare questo esempio grafico:

Considero due linee: una linea rossa (subito successiva all'elemento i) e una sbarra blu (subito precedente all'elemento j). Gli elementi precedenti alla linea rossa sono minori del pivot, mentre gli elementi tra la linea rossa e la linea blu sono maggiori del pivot. Se la i avanza oltrepassando la linea rossa (e avanza quando j trova un elemento minore del pivot) si scambieranno i con j. Si controllano gli elementi tra la linea blu e il pivot.





Esempio 2

$$A = \{13, 19, 9, 5, 12, 8, 7, 4, 21, 2, 6, 11\}$$

I) $i = 0, j = 2$

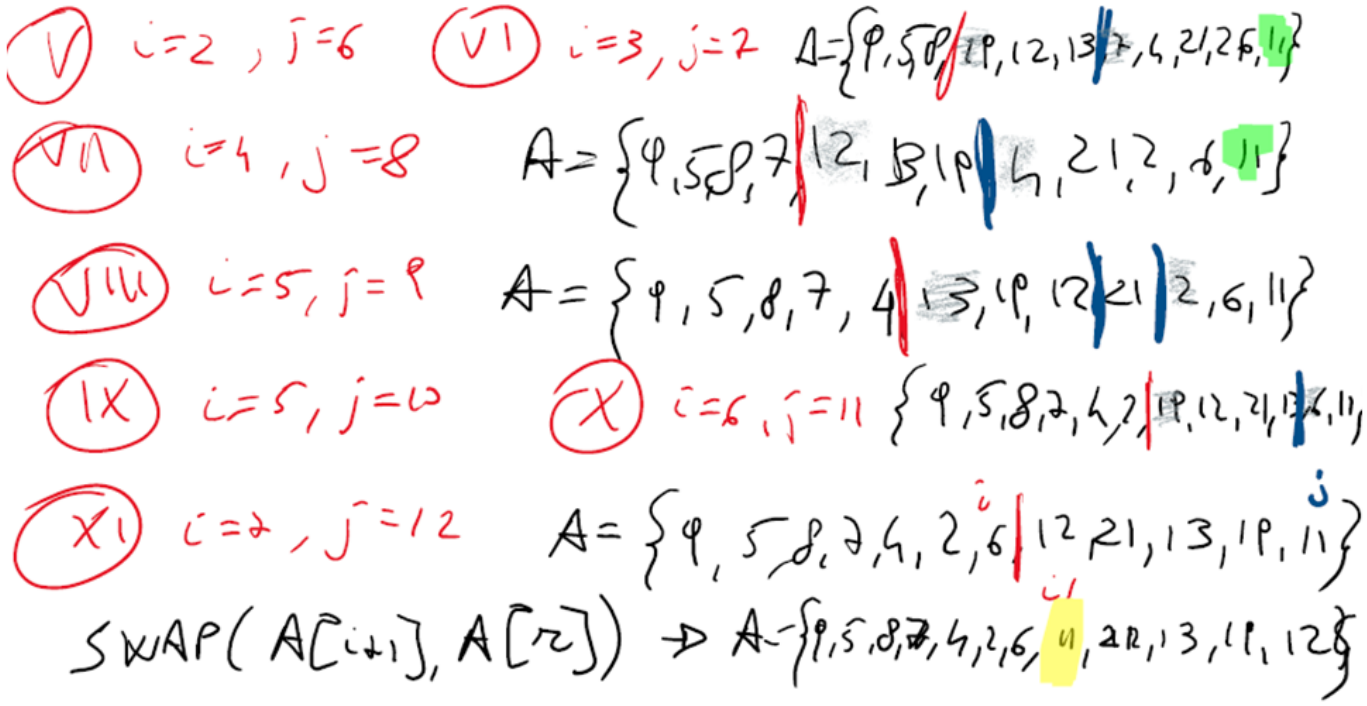
II) $i = 0, j = 3$

III) $i = 1, j = 4$ $A = \{9, 19, 13, 5, 12, 8, 7, 4, 21, 26, 11\}$

IV) $i = 2, j = 5$ $A = \{9, 5, 13, 19, 12, 8, 7, 4, 21, 26, 11\}$

V) $i = 2, j = 6$

VI) $i = 3, j = 2$ $A = \{9, 5, 8, 19, 12, 13, 7, 4, 21, 26, 11\}$



Codice

Definisco la funzione `partition` (che *tornerà l'indice del pivot*) passando come *parametri l'array, l'indice iniziale e l'indice finale*. Inizializzo il `pivot` con l'ultimo elemento dell'array e l'indice `j` con l'elemento iniziale, questo **rappresenta l'indice di separazione tra gli elementi più piccoli e più grandi del pivot.**

C++

```
int partition(int* array, int start, int end){
    int pivot = array[end];
    int j = start;
```

utilizzo un `for` nel quale **inizializzo il valore i** che servirà a scorrere tutti gli elementi compreso il pivot. All'interno del for inserisco una condizione: se *l'elemento dell'array nell'i-esima posizione è minore del pivot* eseguo lo `swap` tra questi ultimi e incremento j.

C++

```
for(int i = start; i<=end; i++){
    if (array[i]<=pivot){
        swap(array[i], array[j])
        j++;
    }
```

Definisco la funzione `quicksort` passando *come parametri un array e i suoi indici iniziali e finali*. All'interno di questa funzione inserisco una condizione: *se l'indice iniziale dovesse essere maggiore o uguale all'indice finale* esco dalla funzione.

```C++
    void quicksort(int* array, int start, int end){
        if(start>=end)
            return;
```

se la condizione invece **non è rispettata** proseguo con le istruzioni successive. In primo luogo inizializzo un intero `q` pari al *risultato della funzione partition* (prima analizzata), successivamente **reitero il quicksort sulle due parti dell'array** (da start a q-1 e da q+1 a end).

```C++
    int q = partition(array, start, end);
    quicksort(array, start, q-1);
    quicksort(array, q+1, end);
}
```