UNIVERSITA' DEGLI STUDI DI
NAPOLI FEDERICO II

Scuola Politecnica e delle Scienze di Base
Corso di Laurea in Ingegneria Informatica

Elaborato in Algoritmi e Strutture Dati

Quicksort e Randomized Quicksort

Anno Accademico 2020/2021

Candidato:

Lamboglia Anna

matr. M63001219

Indice

1. Il Quicksort

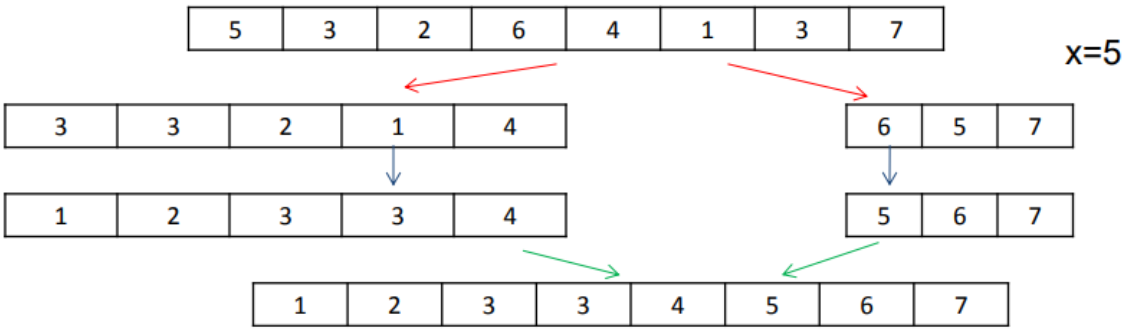
Il *Quicksort* è un algoritmo di ordinamento ricorsivo sul posto non stabile, che utilizza l'approccio *Divide et Impera*[1], utilizzato anche in altri algoritmi come ad esempio il *Merge Sort*. Tale algoritmo, ideato dallo scienziato inglese Tony Hoare nel 1959 e pubblicato nel 1961, nella pratica risulta essere tra i più efficienti e, infatti, è spesso utilizzato negli algoritmi di *sort()*.

1.1 L'algoritmo

La base del funzionamento dell'algoritmo di *Quicksort*, prevede l'utilizzo ricorsivo di una procedura strutturata in fasi, secondo cui, preso un elemento (*pivot*) da una struttura dati (es. array), si pongono gli elementi più piccoli rispetto al pivot a sinistra e gli elementi più grandi a destra. In particolare, il Quicksort si basa sui seguenti tre *step*:

- *Divide*: scelto un elemento *x*, detto *pivot*, viene partizionato l'array A[p..r] in due sotto-array tali che A[p..q-1] contiene gli elementi <= di x e A[q+1..r] contiene gli elementi >x; il pivot sarà poi inserito in A[q];
- *Conquer*: Ordina i due sotto-array mediante chiamate ricorsive;
- *Combine*: I sotto-array sono ordinati sul posto, quindi non occorre fare nulla;

[1] In gergo informatico, l'espressione "*Divide et impera*" ("Separa e conquista") indica un approccio secondo cui la risoluzione di un problema avviene dividendo lo stesso ricorsivamente in due o più sotto-problemi di egual dimensione fin quando questi ultimi diventino di semplice risoluzione; quindi si combinano le soluzioni al fine di ottenere la soluzione del problema dato (*Wikipedia*).



L'operazione più complessa e onerosa da effettuare è la suddivisione dei vettori in sotto-array (*Divide*), mentre la fase di combinazione (*Combine*) risulta semplice, a differenza del *Merge Sort* dove invece avviene il contrario.

Un'ulteriore considerazione da effettuare è che il *pivo*t iniziale può essere preso potenzialmente in qualsiasi posizione.

Nel nostro caso andremo a prendere come pivot l'ultimo elemento dell'*array*.

Come si vedrà meglio successivamente, il *Quicksort* nel caso peggiore ha un tempo d'esecuzione pari a $\theta(n^2)$ e in quello medio $\theta(nlgn)$.

1.2 Pseudocodice

Le variabili in ingresso al Quicksort sono le seguenti :

```
Quicksort (A,p,r)
if p<r
    then q ← Partition (A,p,r)
    Quicksort (A,p,q-1)
    Quicksort (A,q+1,r)
```

- *A*: vettore da ordinare ;
- *p*: indice di inizio del vettore;
- *r*: indice di fine del vettore.

Per ordinare l'intero array si invocherà la *Quicksort* con p uguale ad 1 e r sarà pari a length[A].

Il vettore viene partizionato se p è minore di r, mentre la ricorsione termina quando arriviamo alla soluzione banale che, in questo caso, è la condizione *p=r* ossia un vettore di un solo elemento.

Il *Quicksort* come visto è l'opposto al *Merge sort*, infatti a livello di tempo di esecuzione

la suddivisione dei sottoproblemi contribuisce maggiormente al tempo dell'algoritmo mentre la ricostruzione è banale.

1.3 Partition

La funzione *Partition* utilizza come pivot l'ultimo elemento del sottoarray (*x=A[r]*) ed analizza, uno ad uno ed a partire dal primo, tutti gli elementi compresi tra *p* e *r-1* suddividendo il vettore in quattro aree:

– Gli elementi <= del pivot [nelle posizioni p..i];

– Gli elementi > del pivot [nelle posizioni i+1..j-1];

– Gli elementi non ancora analizzati [j..r-1];

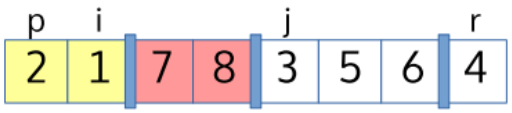
– Il pivot [r].

Il pseudo codice è il seguente:

```
Partition (A,p,r)
x ← A[r]
i ← p-1
for j ← p to r-1
      do if A[j]⩽x
            then i ← i+1
            exchange A[i]↔A[j]
exchange A[i+1] ↔ A[r]
return i+1
```

Come possiamo notare dallo pseudocodice del Quicksort, è necessario definire la funzione partizionamento del vettore iniziale chiamata *Partition*.

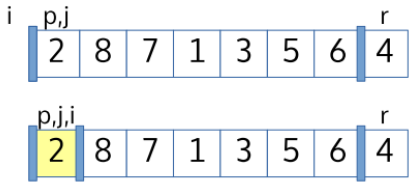
Si consideri un esempio partendo dal seguente vettore iniziale:



L'elemento di posto j (il primo non analizzato) viene confrontato con il pivot; se è

maggiore, è sufficiente incrementare la barriera tra 2ª e 3ª area, ovvero incrementare j. Se, invece, è minore o uguale, deve essere spostato nella prima area e quindi:

- Si incrementa i (si sposta barriera tra 1a e 2a area);
- Si scambia A[i] con A[j];
- Si incrementa j (si sposta barriera tra 2a e 3a area).

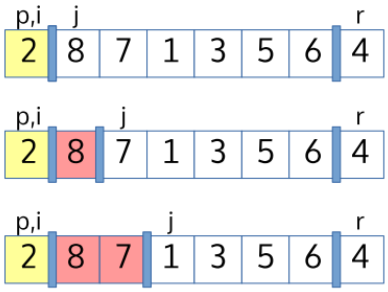


Inizialmente, due barriere (quella tra 1ª e 2ª area e tra 2ª e 3ª area) coincidono, quindi si confronta il primo elemento con il pivot e, se è minore o uguale:

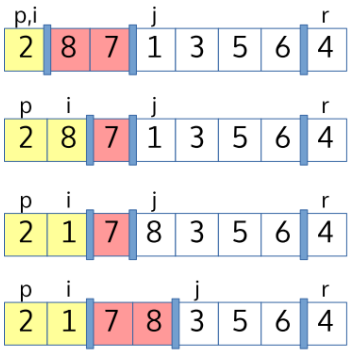
– Si incrementa i (si sposta barriera tra 1ª e 2ª area);

– Si scambia A[i] con A[j];

– Si incrementa j (si sposta barriera tra 2ª e 3ª area).



Se il secondo elemento è maggiore del pivot, si incrementa j (si sposta la barriera tra la 2ª e 3ª area) e se il terzo elemento è maggiore del pivot si incrementa j (si sposta barriera tra 2ª e 3ª area).

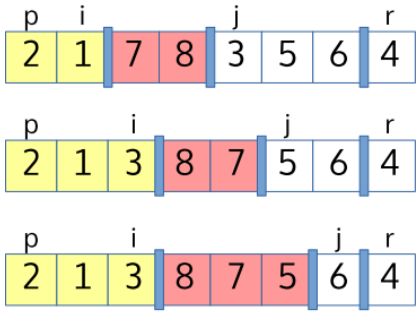


In questo caso, il quarto elemento è minore o uguale del pivot, per cui:

– Si incrementa i (si sposta barriera tra 1ª e 2ª area);

– Si scambia A[i] con A[j];

– Si incrementa j (si sposta barriera tra 2ª e 3ª area).



Il quinto elemento è minore o uguale del pivot per cui:

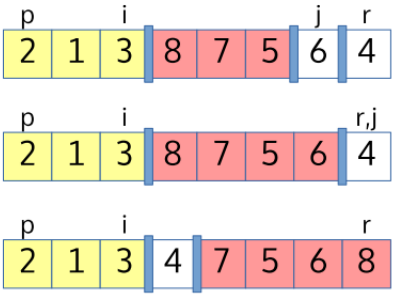
– Si incrementa i (si sposta barriera tra 1ª e 2ª area);

– Si scambia A[i] con A[j];

– Si incrementa j (si sposta barriera tra 2ª e 3ª area);

Il sesto elemento è maggiore del pivot

– Si incrementa j (si sposta barriera tra 2ª e 3ª area);



Il settimo elemento è maggiore del pivot da cui:

– Si incrementa j (si sposta barriera tra 2ª e 3ª area)

A questo punto, il ciclo è finito e si scambia A[i+1] con A[r].

2. Tempo di Esecuzione

2.1 Caso migliore

Questo caso si ha quando si hanno due sotto-problemi di dimensione $\lfloor \frac{n}{2} \rfloor$ ed $\lfloor \frac{n}{2} \rfloor$-1. Ad ogni passo il pivot scelto è la "mediana" degli elementi nell'array.

Avremo un'equazione ricorsiva:

$$T(n) \leq 2T(\frac{n}{2}) + \Theta(n)$$

Risolvendola mediante il secondo caso del teorema dell'esperto otterremo:

$$T(n) = O(n \lg n)$$

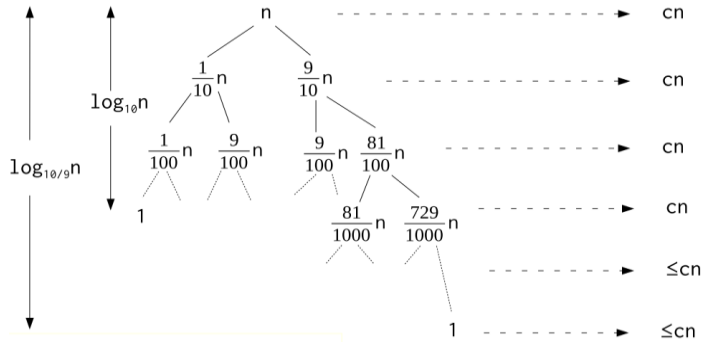
2.2 Caso medio

Si presenta quando i sotto-problemi sono abbastanza sbilanciati.

Supponiamo di avere i sotto-array con un rapporto 9:1, avremo quindi la seguente equazione ricorsiva:

$$T(n) \leq T(\frac{9}{10}n) + T(\frac{n}{10}) + cn$$

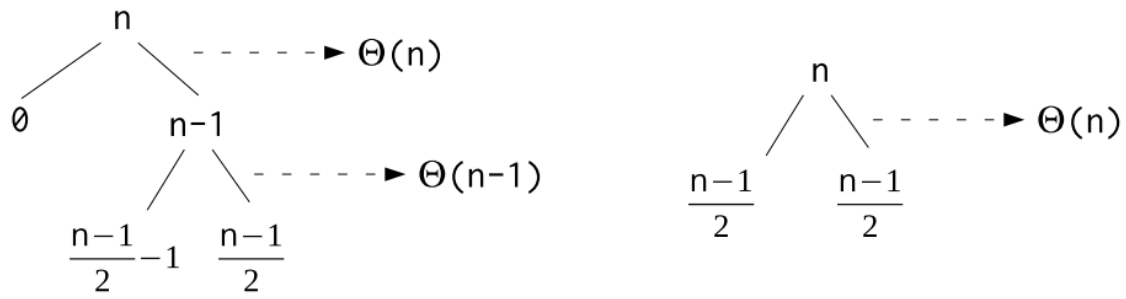
Usando il metodo dell'albero di ricorrenza possiamo poi andare a calcolare il tempo di esecuzione:



La ricorsione termina alla profondità $\log_{10/9} n = \Theta(\lg n)$.

Il costo è dunque $O(n \lg n)$, abbiamo $\Theta(\lg n)$ livelli, ciascuno di costo al più cn e qualunque partizionamento con rapporto costante (anche 99:1) produce un costo $O(n \lg n)$ perché la profondità è sempre $\Theta(\lg n)$.

In particolare, tipicamente nel caso medio partizioni "buone" e "cattive" si alternano, per cui, intuitivamente, se si alternano best case e worst case, il tempo di esecuzione complessivo è quello del best case.



Il costo Θ(n-1) della partizione cattiva viene assorbito nel costo Θ(n) della partizione buona e la partizione risultante è buona.

Inoltre, per aumentare la probabilità che in media il partizionamento effettuato da Quicksort sia ben bilanciato, si può scegliere in maniera aleatoria il pivot, utilizzando la *versione randomizzata* del *Quicksort (Randomized Quicksort)*, ritenuta la scelta migliore per ordinare array di grosse dimensioni.

2.3 Caso peggiore

il caso peggiore si ha nel momento in cui l'algoritmo effettua un partizionamento sbilanciato in cui un problema avrà dimensione 0 mentre l'altro n−1. In questo caso avremo:

$$T(n) = T(n\text{-}1) + T(0) + \Theta(n)$$

T(0) può essere racchiuso nel $\Theta(n)$

$$T(n) = T(n\text{-}1) + \Theta(n)$$

La cui soluzione è $T(n)= \Theta(n^2)$.

In generale in tal modo siamo andati a definire un limite inferiore.

A questo punto dimostriamo che: $T(n) = O(n^2)$.

Mediante il metodo di sostituzione risolveremo la seguente equazione ricorsiva:

$T(n) = \max_{0 \le q \le n-1} (T(q)+T(n-q-1))+\Theta(n)$

$\le \max_{0 \le q \le n-1} (cq^2+c(n-q-1)^2)+ \Theta(n)$

$=c*\max_{0 \le q \le n-1} (q^2+(n-q-1)^2)+ \Theta(n)$

Poiché la funzione $(q^2+(n-q-1)^2)$ è una parabola che ha la concavità verso l'alto (derivata seconda rispetto a q positiva), il massimo è assunto agli estremi, per cui consideriamo per i valori q=0 e q=(n-1):

$c*\max_{0 \le q \le n-1} (q^2+(n-q-1)^2) + \Theta(n)$

$\le c(n-1)^2 + \Theta(n)$

$=cn^2 - c(2n-1) + \Theta(n)$

Scegliendo la quantità $c(2n-1) \ge \Theta(n)$, si ottiene:

$cn^2 - c(2n-1) + \Theta(n) \le \mathbf{cn^2} = O(\boldsymbol{n^2})$

In precedenza abbiamo visto un caso (il caso sbilanciato) in cui il tempo di esecuzione è $\Theta(n^2)$. Quindi il tempo di esecuzione nel caso peggiore è $\Omega(n^2)$. Combinando questo risultato con quello precedente, si ha che il tempo di esecuzione nel caso peggiore è $\Theta(n^2)$

3. Il Randomized Quicksort

Nel Randomized Quicksort, tramite la funzione Randomized Partition, si sceglie il pivot in maniera casuale e poi lo si sostituisce all'ultimo elemento della sequenza.

Mostriamo di seguito lo pseudo codice:

```
Randomized-Partition (A,p,r)
i ← Random (p,r)
exchange A[r] ↔ A[i]
return Partition (A,p,r)
```

```
Randomized-Quicksort (A,p,r)
if p<r
    then q ← Randomized-Partition (A,p,r)
    Randomized-Quicksort (A,p,q-1)
    Randomized-Quicksort (A,q+1,r)
```

4. Implementazione Quicksort e Partition

Per le successive analisi, è stato scelto di implementare il programma nel linguaggio di programmazione *Python*. Di seguito, si riporta l'implementazione in *Python* della funzione Quicksort:

```python
#QUICKSORT
def quickSort(arr, low, high):
    if len(arr) == 1:
        return arr
    if low < high:

        pi = partition(arr, low, high)

        quickSort(arr, low, pi-1)
        quickSort(arr, pi+1, high)
```

Come si può vedere, l'implementazione della funzione *quickSort* risulta molto simile allo pseudocodice analizzato in precedenza. In particolare, in ingresso alla funzione *quickSort* sono definiti i seguenti parametri:

- *arr*: vettore in ingresso;
- *low*: valore di inizio del vettore sul quale applicare la funzione;
- *high*: valore che indica la fine del vettore.

La condizione che valuta il caso base non fa altro che verificare se nel vettore sia presente

soltanto un elemento, cosa che nello pseudocodice è descritta con if di p<r.

Il risultato dell'operazione *partition* viene assegnato alla variabile *pi* (che corrisponde alla q del pseudocodice) e successivamente è applicata la ricorsione.

Anche l'implementazione in *Python* della funzione *partition* è molto simile alla rappresentazione nel pseudocodice. Inoltre, come già indicato in precedenza, il pivot è pari all'ultimo elemento dell'array.

```
#PARTITION
def partition(arr, low, high):
    i = (low-1)
    pivot = arr[high]

    for j in range(low, high):

        if arr[j] <= pivot:
            i = i+1
            arr[i], arr[j] = arr[j], arr[i]

    arr[i+1], arr[high] = arr[high], arr[i+1]
    return (i+1)
```

Sono state inoltre implementate due funzioni che permettono di creare gli array su cui lavorare, in particolare nella prima denominata *create_array_ordinato* l'obiettivo è la creazione di un array ordinato, mentre nella seconda denominata *create_array* l'obiettivo è creare un array randomico.

Entrambe le funzioni prendono in ingresso il parametro n che indica la lunghezza del vettore da creare.

```python
#CREAZIONE DI UN ARRAY ORDINATO (CASO PEGGIORE)
def create_array_ordinato(n):
    lista=[]
    for i in range(0,n):
        lista.append(i)
    #print(lista)
    return lista


#CREAZIONE DI UN ARRAY RANDOM
import random
def create_array(n):
    lista=[]
    for i in range(n):
        casuale=random.randint(0,100)
        lista.append(casuale)
    #print(lista)
    return lista
```

Sono state inoltre implementate anche le funzioni *randomized_quickSort* e *randomized_partition* per poterne valutare la differenza con il *Quicksort*.

Le due funzioni sono molto simili, con l'unica differenza che nella *randomized_partition* il pivot è scelto tramite la funzione *randint* della libreria *random*.

```python
#RANDOMIZED-QUICKSORT
def randomized_quickSort(arr, low, high):
    if len(arr) == 1:
        return arr
    if low < high:
        pi = randomized_partition(arr, low, high)
        randomized_quickSort(arr, low, pi-1)
        randomized_quickSort(arr, pi+1, high)
```

```python
#RANDOMAZED-PARTITION
def randomized_partition(arr, low, high):
    i = (low-1)
    pivot = arr[random.randint(low,high)]

    for j in range(low, high):

        if arr[j] <= pivot:

            i = i+1
            arr[i], arr[j] = arr[j], arr[i]

    arr[i+1], arr[high] = arr[high], arr[i+1]
    return (i+1)
```

5. Main per il testing

L'obiettivo in questo capitolo è di effettuare un'analisi di sensitività tra il *Quicksort* ed il *Randomized Quicksort*, al fine di valutarne le prestazioni. In particolare, si andranno a variare alternativamente due parametri, ossia la dimensione *n* dell'array in ingresso e la disposizione degli elementi del vettore in input, ottenendo per ciascun algoritmo i seguenti quattro sotto-casi:

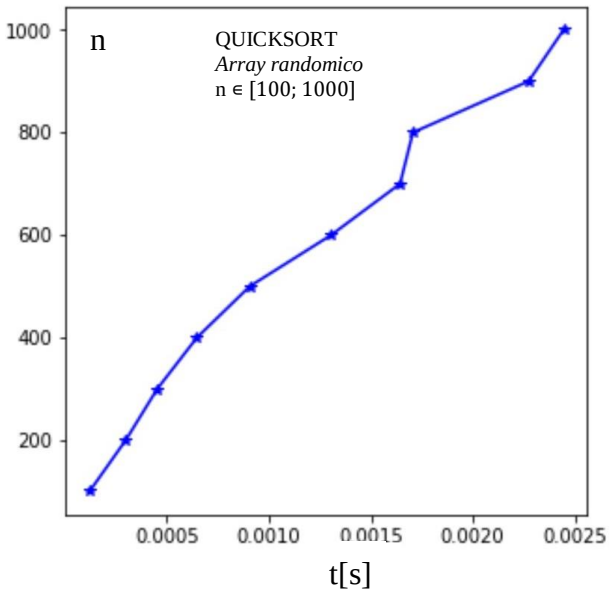
- Dimensione *n* dell'array in ingresso:
    1) *Caso 1*: n variabile nell'intervallo [100; 1000] con passo 100;
    2) *Caso 2*: n variabile nell'intervallo [1000; 10000] con passo 1000.
- Ordinamento dell'array in ingresso:
    1) *Caso medio*: vettore randomico;
    2) *Caso peggiore*: vettore ordinato.

Per comodità sono stati definiti *main* separati per il testing di *Quicksort* e *Randomized Quicksort* e, per le analisi con vettori di ordini di grandezza differenti, le prove sono state effettuate utilizzando *Google Colab*.

5.1 Quicksort - Caso medio: vettore randomico da 100 a 1000

Valutiamo i risultati del *Quicksort* considerando un vettore di grandezza *n*, variabile nell'intervallo [100; 1000] con passo 100, con elementi disposti in modo *randomico*. Si riportano di seguito la tabella con i risultati ottenuti in termini di tempi di elaborazione e la relativa trasposizione grafica:

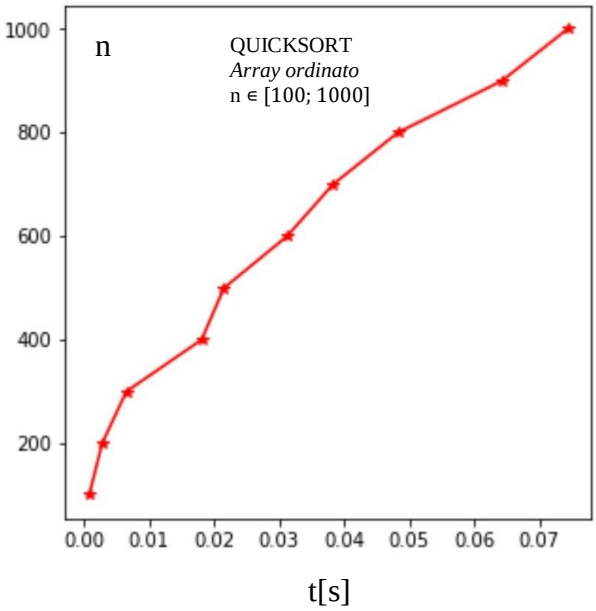
QUICKSORT	VETTORE RANDOMICO
N	TEMPO SECONDI
100	0,0001
200	0,0003
300	0,0005
400	0,0006
500	0,0009
600	0,0013
700	0,0016
800	0,0017
900	0,0023
1000	0,0024



5.2 Quicksort - Caso peggiore: vettore ordinato da 100 a 1000

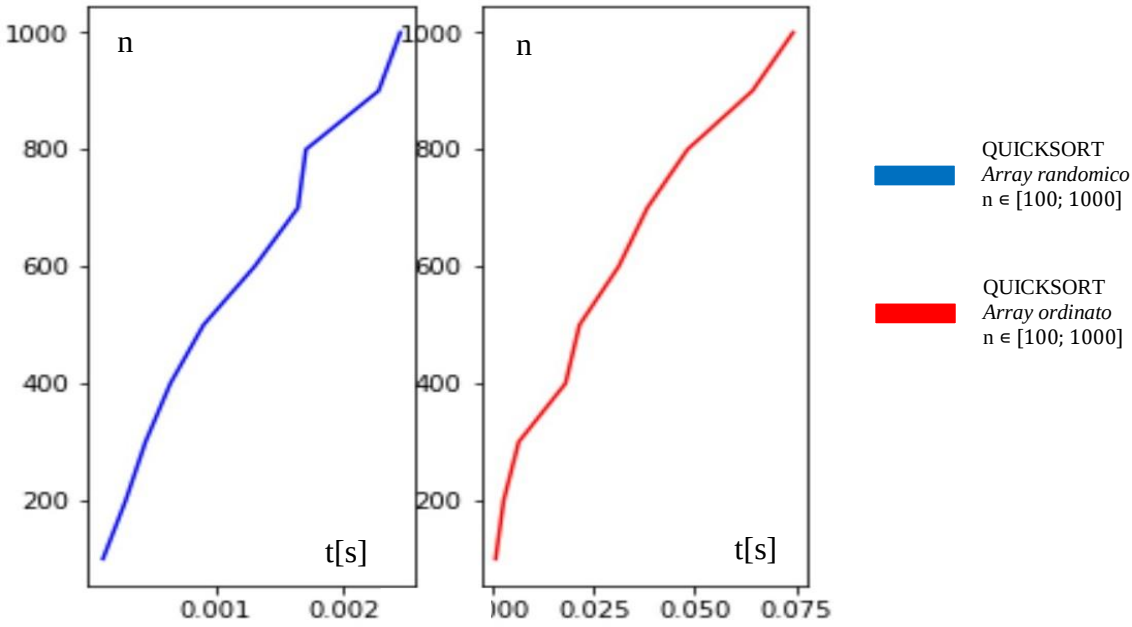
Valutiamo i risultati del *Quicksort* considerando un vettore di grandezza *n*, variabile nell'intervallo [100; 1000] con passo 100, con elementi disposti in modo *ordinato*. Si riportano di seguito la tabella con i risultati ottenuti in termini di tempi di elaborazione e la relativa trasposizione grafica:

QUICKSORT	VETTORE ORDINATO
N	TEMPO SECONDI
100	0,0008
200	0,0027
300	0,0065
400	0,018
500	0,0215
600	0,0312
700	0,0382
800	0,0482
900	0,0642
1000	0,0743



5.3 Quicksort – n da 100 a 1000 - Confronto dei risultati ottenuti

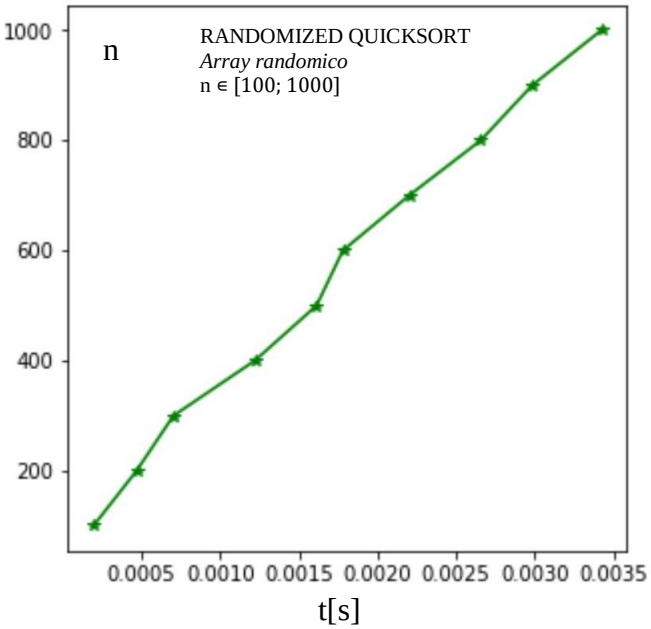
Come emerge dai risultati sopra indicati, il Quicksort che riceve in ingresso un array ordinato restituisce dei tempi di esecuzione di gran lunga superiori rispetto al caso di array generato randomicamente:



5.4 Randomized Quicksort – Caso medio: vettore randomico da 100 a 1000

In modo simile a quanto effettuato per il *Quicksort*, sono riportati di seguito i risultati ottenuti utilizzando il *randomized Quicksort*, sempre considerando un vettore di grandezza *n*, variabile nell'intervallo [100; 1000] con passo 100, con elementi disposti in modo *randomico*. Si riportano di seguito la tabella con i risultati ottenuti in termini di tempi di elaborazione e la relativa trasposizione grafica:

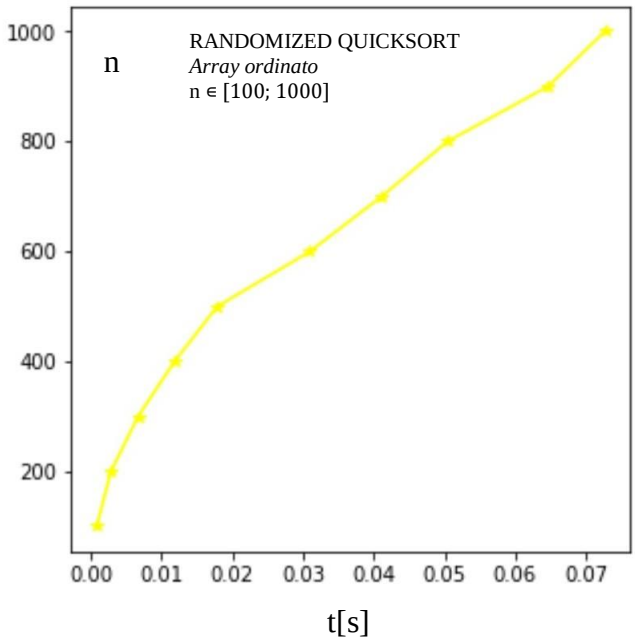
RANDOMIZED C	VETTORE RANDOMICO
N	TEMPO SECONDI
100	0,0002
200	0,0005
300	0,0007
400	0,0012
500	0,0016
600	0,0018
700	0,0022
800	0,0027
900	0,0029
1000	0,0034



5.5 Randomized Quicksort - Caso “peggiore”: vettore ordinato da 100 a 1000

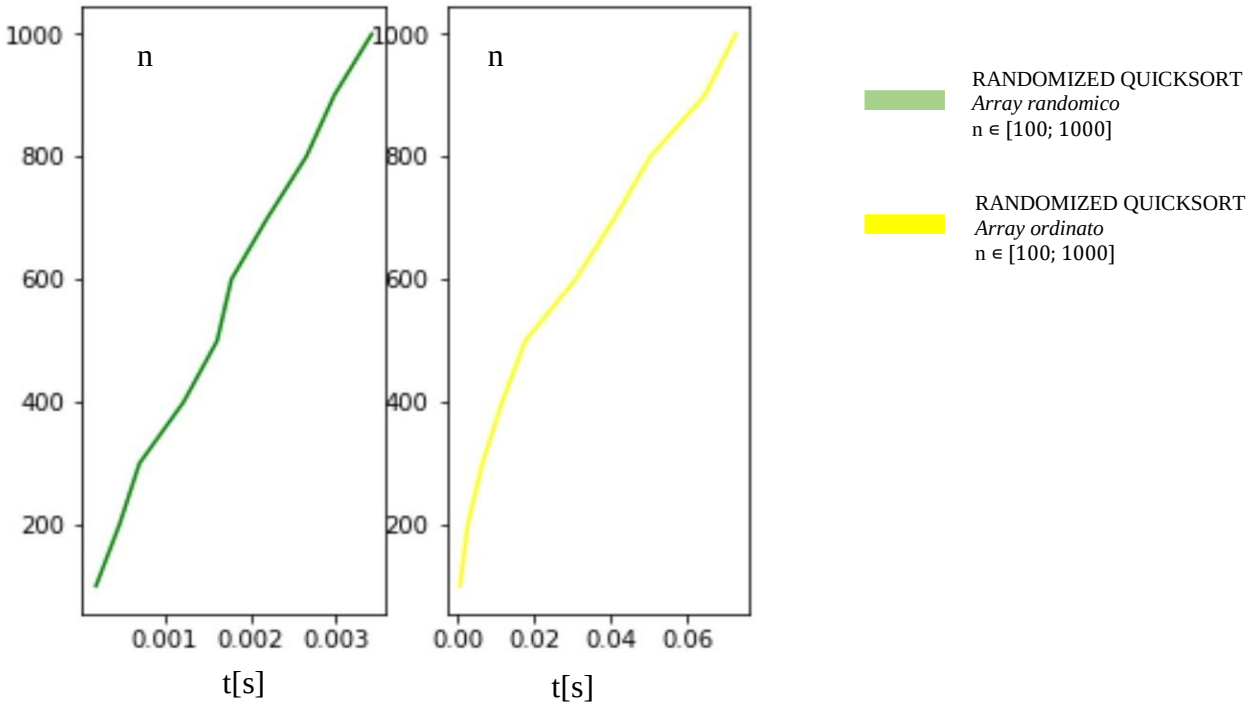
In modo simile a quanto effettuato per il caso medio del *randomized Quicksort*, sono riportati di seguito i risultati ottenuti utilizzando un vettore di grandezza *n*, sempre variabile nell’intervallo [100; 1000] con passo 100, ma con elementi disposti in modo *ordinato*. Si riportano di seguito la tabella con i risultati ottenuti in termini di tempi di elaborazione e la relativa trasposizione grafica:

RANDOMIZED C	VETTORE ORDINATO
N	TEMPO SECONDI
100	0,0008
200	0,0028
300	0,0066
400	0,0118
500	0,0178
600	0,0309
700	0,0411
800	0,0504
900	0,0646
1000	0,0727



5.6 Randomized Quicksort – n da 100 a 1000 - Confronto dei risultati ottenuti

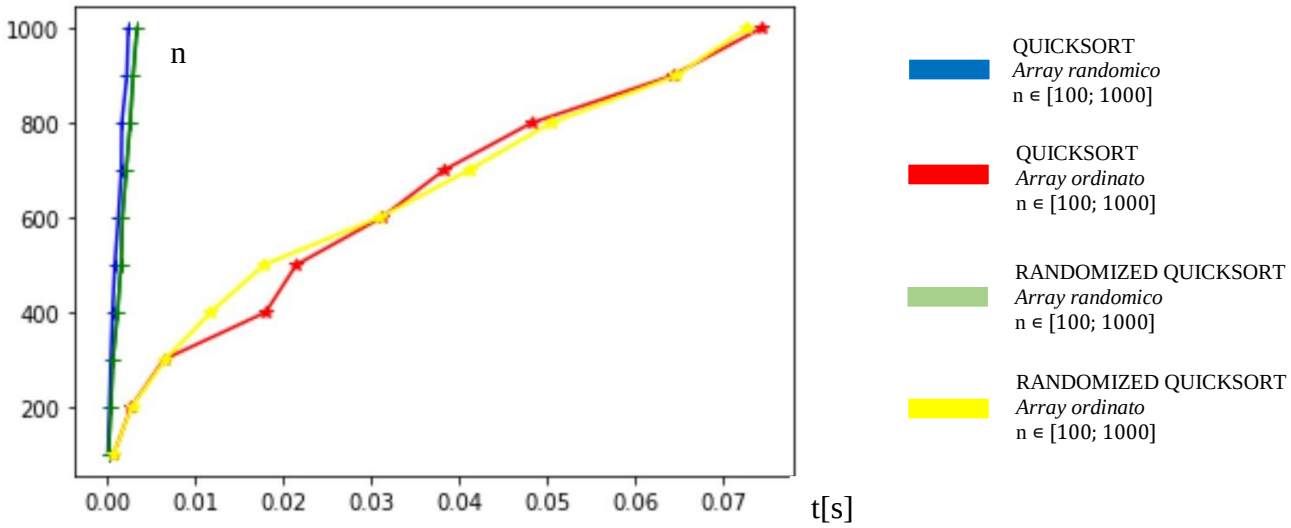
I risultati ottenuti tra caso medio e caso "peggiore" nel caso di *randomized Quicksort* sono simili a quanto ottenuto per il *Quicksort*, nonostante il pivot in quest'ultimo caso venga scelto randomicamente, infatti il tempo necessario a raggiungere il risultato finale è maggiore in caso di vettore preliminarmente *ordinato*:



5.7 Confronto tra Quicksort e Randomized Quicksort – n da 100 a 1000

Si riporta di seguito un confronto tra i risultati dei quattro casi finora esaminati, in cui l'unica variabile costante è la dimensione *n* dell'array, sempre variabile nell'intervallo [100; 1000] con passo 100.

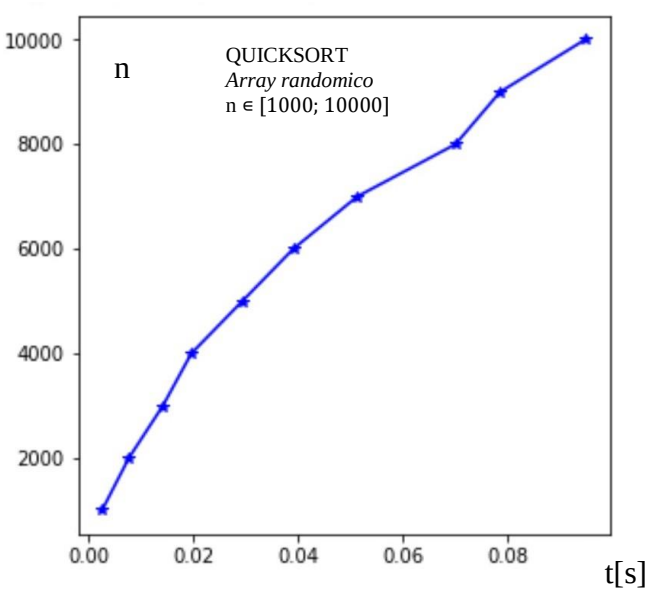
Ciò che emerge è che le due metodologie *Quicksort* e *Randomized Quicksort* forniscono risultati abbastanza simili al variare di *n* e dell'array in input, per cui è necessario verificare se tale risultato dipenda in realtà dagli ordini di grandezza di *n*, che sono abbastanza piccoli oppure se in realtà le prestazioni tra i due algoritmi siano simili. Le variazioni significative riguardano l'array in input, in quanto array randomici portano in ogni caso a risultati migliori.



5.8 Quicksort – Caso medio: vettore randomico da 1000 a 10000

Valutiamo ora i risultati del *Quicksort* considerando un vettore di grandezza *n*, variabile nell'intervallo [1000; 10000] con passo 1000, con elementi disposti in modo *randomico*. Si riportano di seguito la tabella con i risultati ottenuti in termini di tempi di elaborazione e la relativa trasposizione grafica:

QUICKSORT	VETTORE RANDOMICO
N	TEMPO SECONDI
1000	0,0025
2000	0,0076
3000	0,0141
4000	0,0195
5000	0,0293
6000	0,0391
7000	0,0513
8000	0,0701
9000	0,0786
10000	0,0948

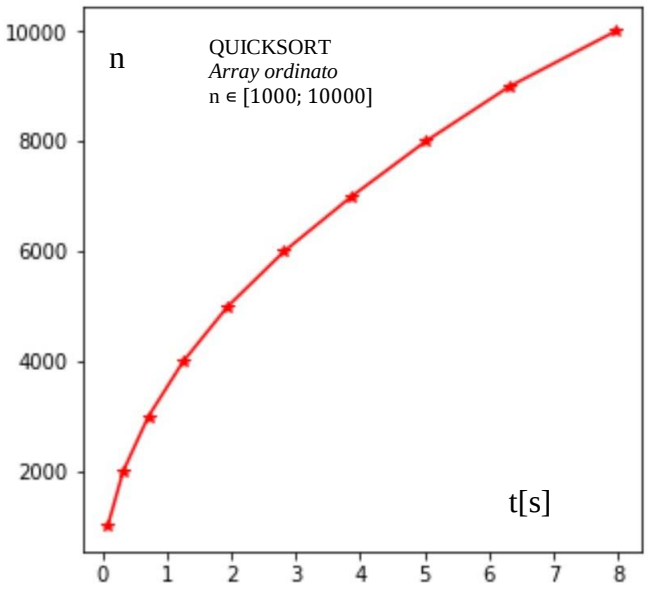


5.9 Quicksort – Caso peggiore: vettore ordinato da 1000 a 10000

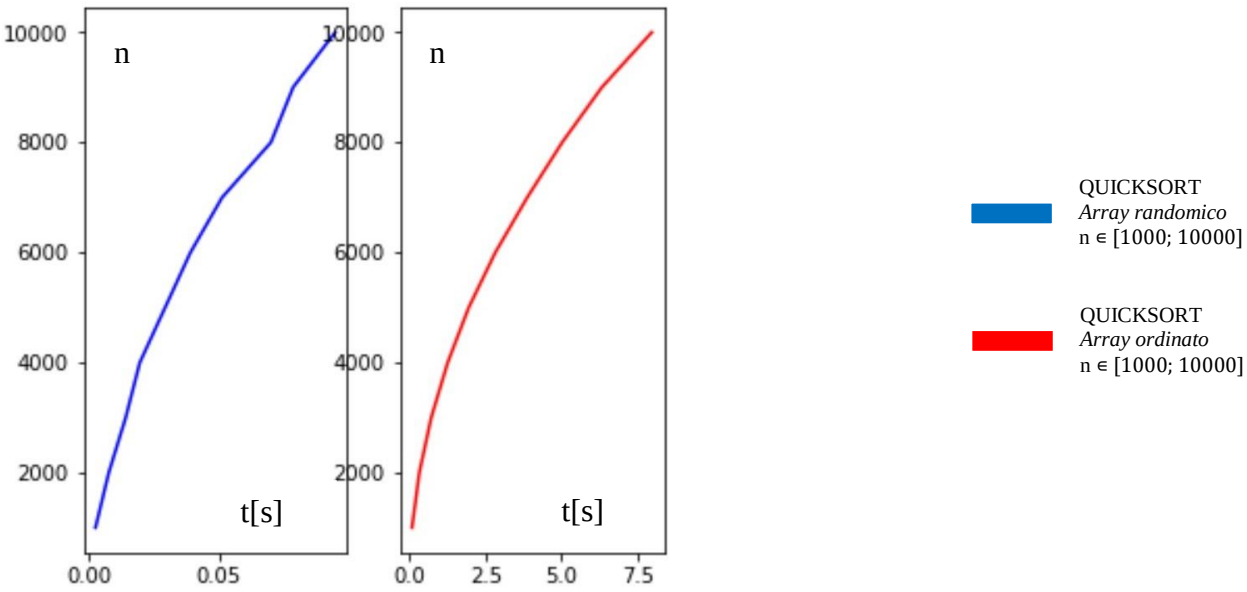
Valutiamo i risultati del *Quicksort* considerando un vettore di grandezza *n*, variabile nell'intervallo [1000; 10000] con passo 1000, con elementi disposti in modo *ordinato*. Si

riportano di seguito la tabella con i risultati ottenuti in termini di tempi di elaborazione e la relativa trasposizione grafica:

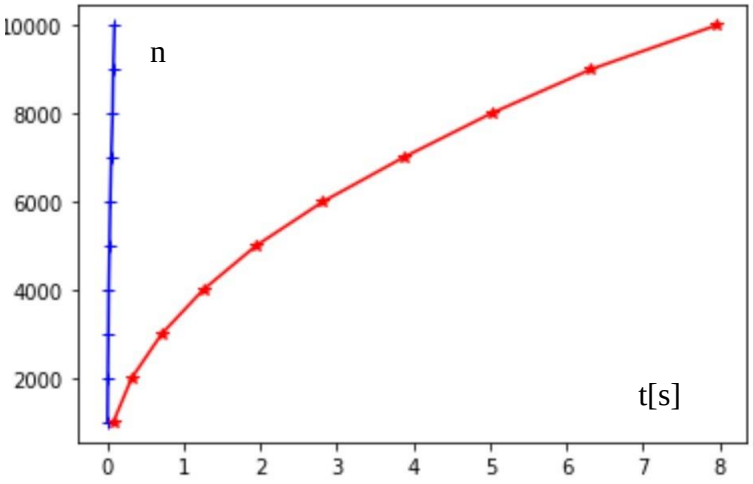
QUICKSORT	VETTORE ORDINATO
N	TEMPO SECONDI
1000	0,079
2000	0,3193
3000	0,7114
4000	1,2528
5000	1,9436
6000	2,8186
7000	3,868
8000	5,0164
9000	6,3201
10000	7,9624



5.10 Quicksort – n da 1000 a 10000 – Confronto dei risultati ottenuti



In questo caso possiamo notare la netta differenza tra i due grafici:



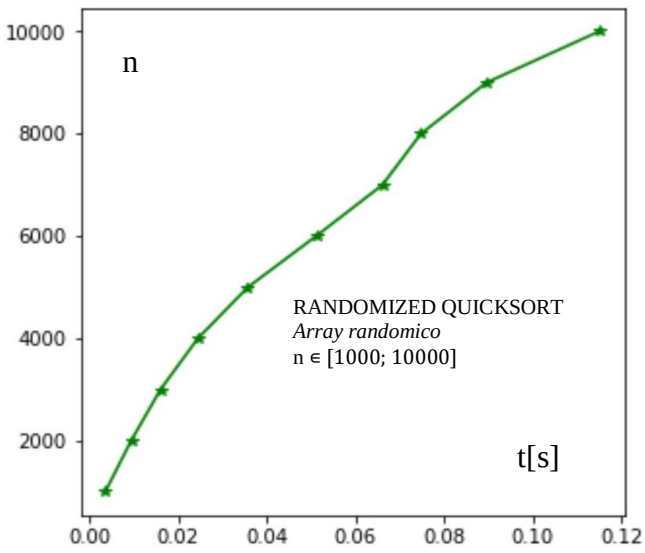
QUICKSORT
Array randomico
n ∈ [1000; 10000]

QUICKSORT
Array ordinato
n ∈ [1000; 10000]

5.11 Randomized Quicksort – Caso medio: vettore randomico da 1000 a 10000

In modo simile a quanto effettuato per il *Quicksort*, sono riportati di seguito i risultati ottenuti utilizzando il *randomized Quicksort*, sempre considerando un vettore di grandezza *n*, variabile nell'intervallo [1000; 10000] con passo 1000, con elementi disposti in modo *randomico*. Si riportano di seguito la tabella con i risultati ottenuti in termini di tempi di elaborazione e la relativa trasposizione grafica:

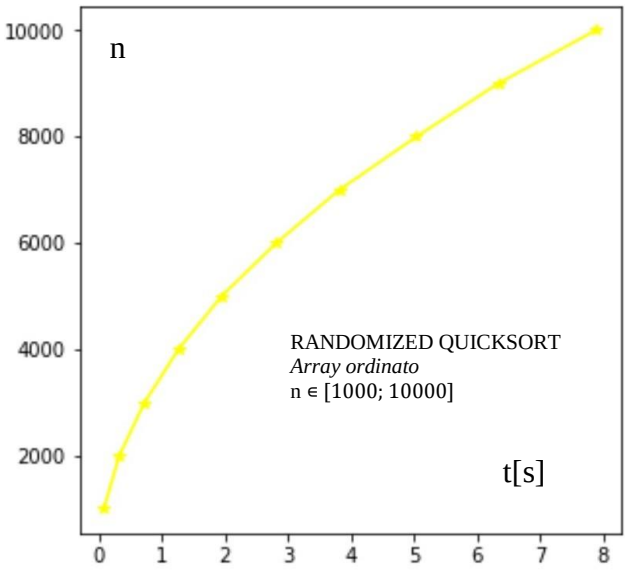
RANDOMIZED C	VETTORE RANDOMICO
N	TEMPO SECONDI
1000	0,0035
2000	0,0093
3000	0,0159
4000	0,0243
5000	0,0357
6000	0,051
7000	0,066
8000	0,0747
9000	0,0895
10000	0,1149



5.12 Randomized Quicksort – Caso “peggiore”: vettore ordinato da 1000 a 10000

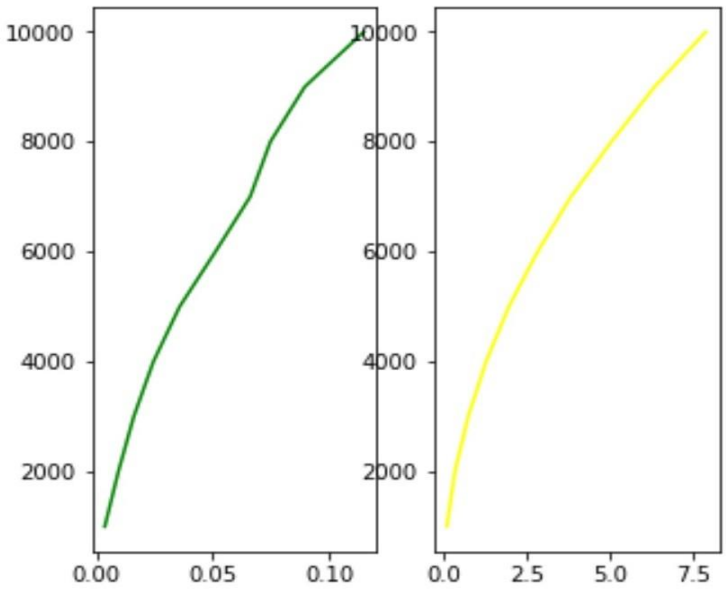
In modo simile a quanto effettuato per il caso medio del *randomized Quicksort*, sono riportati di seguito i risultati ottenuti utilizzando un vettore di grandezza *n*, sempre variabile nell’intervallo [1000; 10000] con passo 1000, ma con elementi disposti in modo *ordinato*. Si riportano di seguito la tabella con i risultati ottenuti in termini di tempi di elaborazione e la relativa trasposizione grafica:

QUICKSORT	VETTORE ORDINATO
N	TEMPO SECONDI
1000	0,0731
2000	0,3177
3000	0,7178
4000	1,2551
5000	1,9407
6000	2,808
7000	3,0325
8000	5,0325
9000	6,3332
10000	7,8776



5.13 Randomized Quicksort – n da 1000 a 10000 – Confronto dei risultati ottenuti

I risultati ottenuti tra caso medio e caso “peggiore” nel caso di *randomized Quicksort* sono simili a quanto ottenuto per il *Quicksort*, nonostante il pivot in quest’ultimo caso venga scelto randomicamente, infatti il tempo necessario a raggiungere il risultato finale è maggiore in caso di vettore preliminarmente *ordinato*:

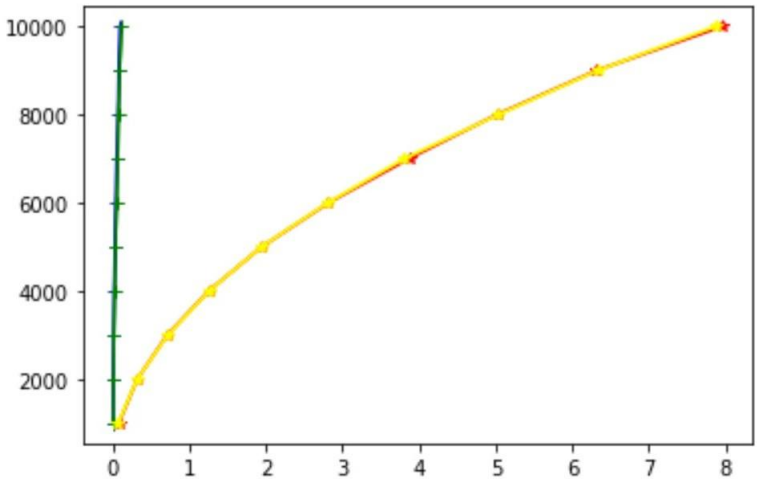


RANDOMIZED QUICKSORT
Array randomico
n ∈ [1000; 10000]

RANDOMIZED QUICKSORT
Array ordinato
n ∈ [1000; 10000]

5.14 Confronto tra Quicksort e Randomized Quicksort – n da 1000 a 10000

In questo caso possiamo notare come aumentando la grandezza dei vettori di ingresso la differenza risulti minima



QUICKSORT
Array randomico
n ∈ [1000; 10000]

QUICKSORT
Array ordinato
n ∈ [1000; 10000]

RANDOMIZED QUICKSORT
Array randomico
n ∈ [1000; 10000]

RANDOMIZED QUICKSORT
Array ordinato
n ∈ [1000; 10000]

6. Approfondimento statistico - Caso medio Quicksort

Come già esposto dalla teoria è possibile notare che, anche nel caso di vettori molto sbilanciati i risultati ottenuti risultano più vicini al caso migliore che a quello peggiore.

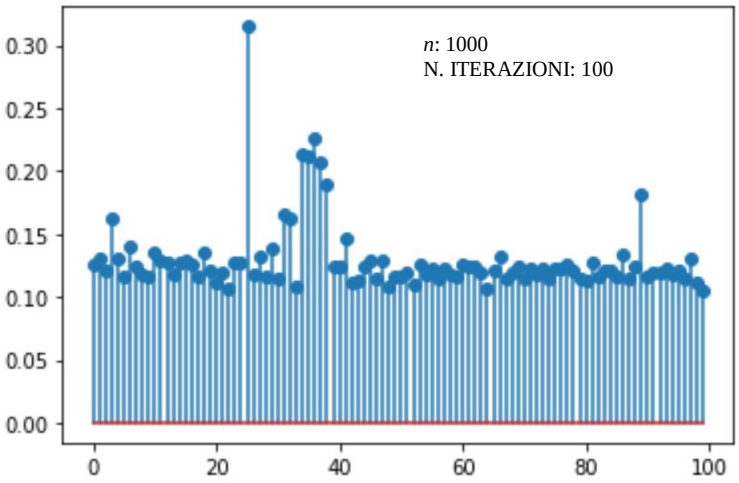
Vediamo il *main* di implementazione:

```
def main3():
    g=1000
    iterazioni=100
    tempi=[]

    for i in range(iterazioni):
        arr=create_array(g)
        ti=time.time()
        quickSort(arr, 0, g-1)
        tf=time.time()
        tempo=round((tf-ti)*1000,5)
        print("Tempo finale in millisecondi", tempo)
        tempi.append(tempo)

    print("MEDIA DEI TEMPI IN MILLISECONDI:",np.mean(tempi))
    print("MAX DEI TEMPI MILLISECONDI:",max(tempi))
    print("MIN DEI TEMPI MILLISECONDI:",min(tempi))
    print("MAX-MEDIA DEI TEMPI MILLISECONDI:",max(tempi)-np.mean(tempi))
    print("MEDIA-MIN DEI TEMPI MILLISECONDI:",np.mean(tempi)-min(tempi))
    plt.stem(tempi,linefmt=None, markerfmt=None, basefmt=None)
```

Sono state effettuate 100 iterazioni, supponendo in ingresso alla *Quicksort* un vettore generato randomicamente di 1000 elementi.

Vediamo il risultato:

```
MEDIA DEI TEMPI IN MILLISECONDI: 0.129337
MAX DEI TEMPI MILLISECONDI: 0.31495
MIN DEI TEMPI MILLISECONDI: 0.1049
MAX-MEDIA DEI TEMPI MILLISECONDI: 0.185613
MEDIA-MIN DEI TEMPI MILLISECONDI: 0.024437000000000014
/usr/local/lib/python3.6/dist-packages/ipykernel_launche
```



Come si può osservare, la media risulta molto più vicina al valore minimo che al valore massimo del tempo di esecuzione raggiunto per le varie operazioni.

È stata effettuata la stessa prova di 1000 iterazioni con un vettore di 1000 elementi e si può valutare come abbia lo stesso comportamento:

```
MEDIA DEI TEMPI IN MILLISECONDI: 2.4302313
MAX DEI TEMPI MILLISECONDI: 4.68349
MIN DEI TEMPI MILLISECONDI: 2.20227
MAX-MEDIA DEI TEMPI MILLISECONDI: 2.2532587
MEDIA-MIN DEI TEMPI MILLISECONDI: 0.22796130000000003
/usr/local/lib/python3.6/dist-packages/ipykernel_launc
```