

# Analisi Comparativa di QuickSelect, HeapSelect e MedianOfMediansSelect

Un progetto di:

Nome Cognome	Matricola	Mail
Francesco Verrengia	157847	<a href="mailto:157847@spes.uniud.it">157847@spes.uniud.it</a>
Riccardo Gottardi	162077	<a href="mailto:162077@spes.uniud.it">162077@spes.uniud.it</a>
Martina Ammirati	161831	<a href="mailto:161831@spes.uniud.it">161831@spes.uniud.it</a>

Università degli studi di Udine  
Dipartimento di Scienze Matematiche Informatiche e Fisiche. 26 Maggio 2024

## Scopo del progetto

Il progetto è incentrato sull'implementazione di tre diversi algoritmi di selezione del  $k - esimo$  elemento più piccolo contenuto in un array, e sull'analisi comparativa del comportamento di tali algoritmi nel caso medio. Si concentra, inoltre, sull'implementazione di un *benchmark* che sia in grado di misurare in maniera accurata l'andamento dei tre algoritmi e di generare un grafico su scala logaritmica che rappresenti efficacemente le loro prestazioni al variare della dimensione degli array utilizzati e alla scelta del parametro  $k$ .

## Il mio ruolo in questo progetto

Mi sono occupato principalmente della parte organizzativa del lavoro, della scrittura del benchmark e della funzione che disegna il grafico, della creazione dei commenti di ogni procedura, della generazione dei grafici, e della stesura di questa relazione.

Ogni fase del progetto è stata attivamente revisionata da ogni membro del gruppo.

## Introduzione: gli algoritmi di selezione in generale

Gli algoritmi di selezione, come *QuickSelect*, *HeapSelect* e *MedianOfMediansSelect*, sono utilizzati per trovare il  $k - esimo$  elemento più piccolo (o più grande) in un insieme di dati senza dover ordinare l'intero insieme. Questo tipo di operazione è cruciale in vari ambiti dell'informatica e della matematica applicata.

In statistica, ad esempio, trovare il mediano o altri percentili di un insieme di dati è fondamentale. Il mediano, infatti, è spesso utilizzato come misura centrale resistente agli outlier, mentre i percentili descrivono la distribuzione dei dati. Gli algoritmi di selezione permettono di calcolare tali misure in modo efficiente.

Nel machine learning, durante la fase di preprocessing dei dati, la selezione del mediano o di altri elementi ordinati può aiutare a normalizzare i dati, rendendone più semplice l'analisi. Un metodo comune di normalizzazione è la normalizzazione con mediana, in cui gli algoritmi di selezione giocano un ruolo fondamentale.

Questi algoritmi sono ampiamente utilizzati anche nell'ambito della computer grafica e nell'ottimizzazione di algoritmi di ordinamento, come *QuickSort*.

## Linguaggi e librerie utilizzate

Abbiamo scelto di strutturare l'intero progetto in *Python* perché, nonostante sia più lento rispetto a linguaggi come *C* e *C++*, offre una sintassi molto semplice.

Questa scelta privilegia la leggibilità del codice, che è la nostra prerogativa principale per questo progetto.

Sempre per motivi di leggibilità, tutte le procedure chiamate dai 3 algoritmi principali e dal *benchmark*, sono incluse nelle sezioni "Appendice" dedicate e non verranno inserite nelle sezioni relative ai singoli algoritmi.

Per gli algoritmi di selezione, non sono state utilizzate librerie esterne.

Per il *benchmark*, invece, sono state utilizzate le seguenti librerie: *random*, *time*, *os*, *matplotlib*.

Tutti i grafici sono realizzati prima su scala logaritmica e poi su scala lineare su entrambi gli assi, fatta eccezione per il grafico 'Prestazioni algoritmi di selezione al variare di k' che è stato realizzato solo su scala lineare.

## Strumenti utilizzati

Il *benchmark* è stato eseguito in un ambiente controllato, su un *Macbook Air M1* standard del 2020.

Durante l'esecuzione del *benchmark* non c'erano altri programmi in esecuzione.

*CPU* : Apple M1

*Ram* : 8GB

*SSD* : 250GB

*Sistema operativo* : macOS Sonoma 14.5

Da notare che il *benchmark* è stato testato anche su un laptop con sistema operativo *Ubuntu 22.04.4 LTS* e non è stato pensato per essere eseguito su *Windows*. In particolare i tempi di esecuzione, su *Windows* risultano estremamente elevati, probabilmente per l'utilizzo della funzione *time.monotonic()*, inoltre, nel caso dell'esecuzione su *Windows*, il grafico non verrà generato.

## QuickSelect

Nella versione (iterativa) di *QuickSelect* implementata, durante la fase di ricerca del  $k - esimo$  elemento viene iterata la chiamata a *partition* su sottoinsiemi dell'array di dimensione sempre minore fino a quando il perno sul quale viene effettuata la partizione non coincide con il  $k - esimo$  elemento oppure il sottoarray considerato non è di dimensione unitaria.

Di fatto, la complessità dell'algoritmo è dovuta quasi interamente a *partition*, il cui compito è di partizionare l'array rispetto ad un perno: spostando tutti gli elementi minori di esso alla sua sinistra, e tutti i maggiori alla sua destra, terminando con il perno nella posizione in cui si troverebbe se l'array fosse ordinato.

Successivamente alla chiamata a *partition*:

Se  $k$  è minore del pivot, viene reiterata la chiamata sulla partizione di array contenente gli elementi minori del pivot.

Se  $k$  è maggiore del pivot, viene reiterata sulla partizione contenente gli elementi maggiori.

Altrimenti, il pivot coincide con il  $k - esimo$  elemento, quindi l'algoritmo termina restituendo il pivot.

Nel caso migliore, *partition* viene chiamato  $\Theta(1)$  volte. In tal caso l'algoritmo ha quindi una complessità di *QuickSelect*  $\Theta(n)$ .

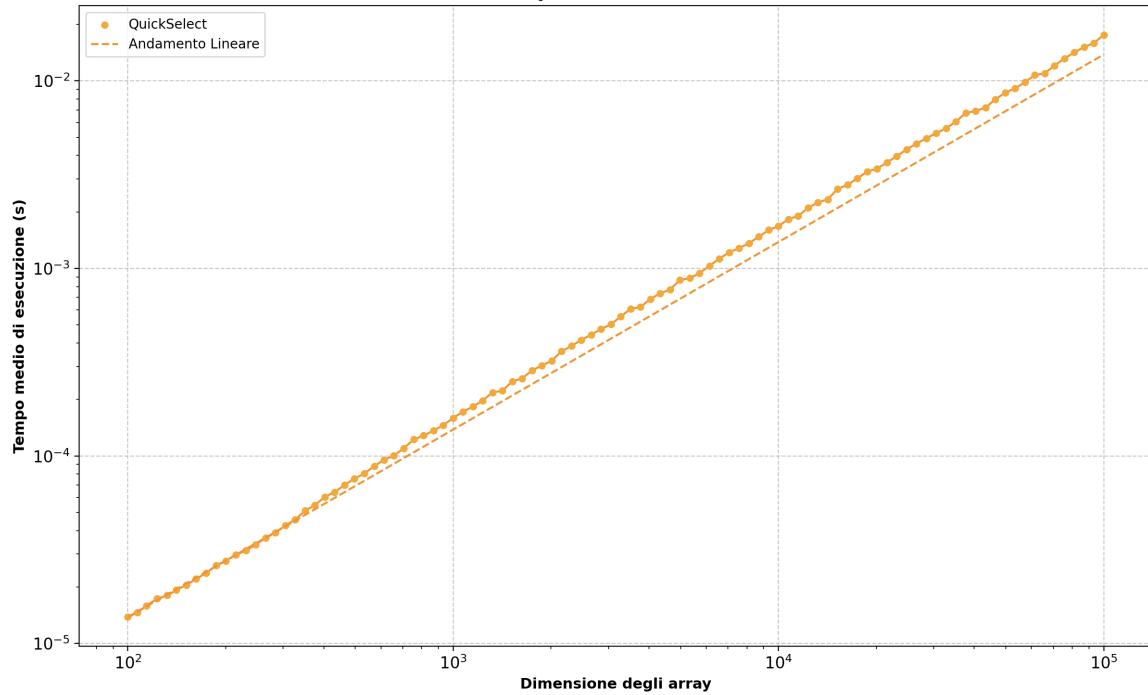
Il caso peggiore per *QuickSelect* si verifica quando l'array è ordinato in senso crescente e viene richiesto di trovare un elemento la cui posizione  $k$  è prossima all'inizio dell'array. In tal caso *partition* verrà eseguito su una partizione di array che diminuisce ogni volta di esattamente un elemento.

Eseguirà pertanto circa  $n^2/2$  operazioni, quindi *QuickSelect* avrà quindi una complessità di  $\Theta(n^2)$ .

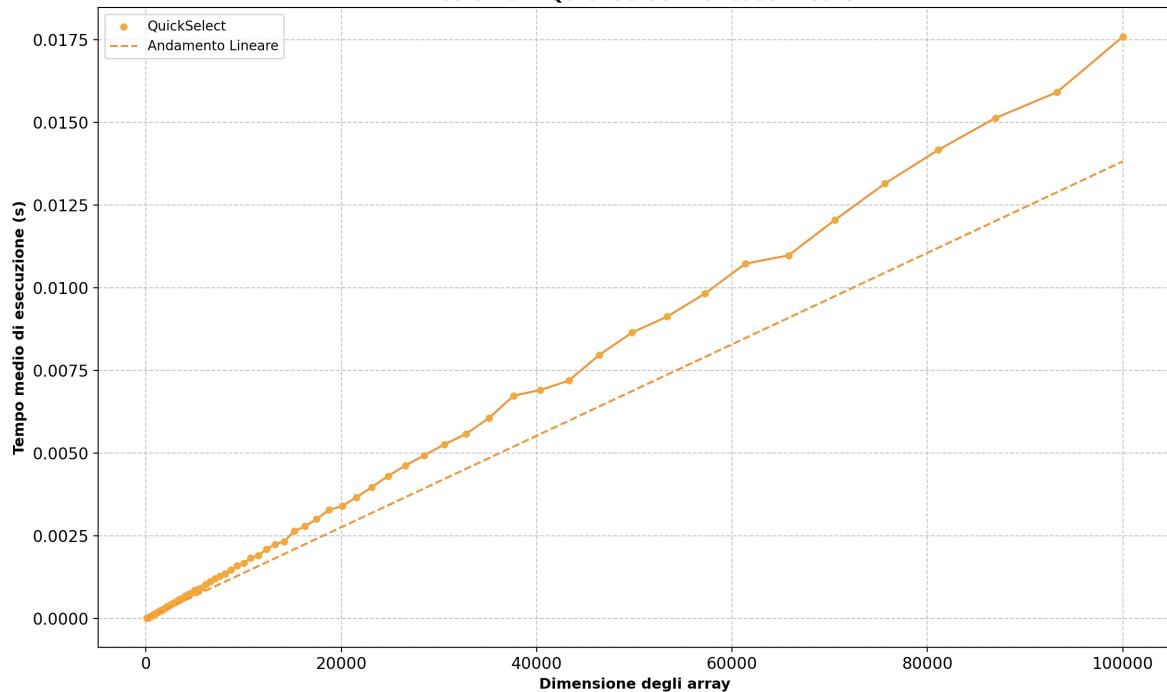
In generale possiamo affermare che *QuickSelect* abbia complessità temporale  $O(n^2)$ .

Tuttavia, nel caso medio, *QuickSelect* sembra avvicinarsi maggiormente ad una complessità lineare.

Prestazioni QuickSelect nel caso medio



Prestazioni QuickSelect nel caso medio



## MedianOfMediansSelect

Per migliorare *QuickSelect*, *Partition*, ad ogni iterazione, dovrebbe prendere come pivot il mediano dell'array (elemento che si troverebbe in posizione  $k = \frac{\text{len}(A)}{2}$ , se l'array fosse ordinato).

L'idea di *MedianOfMediansSelect* di base proprio su questo concetto.

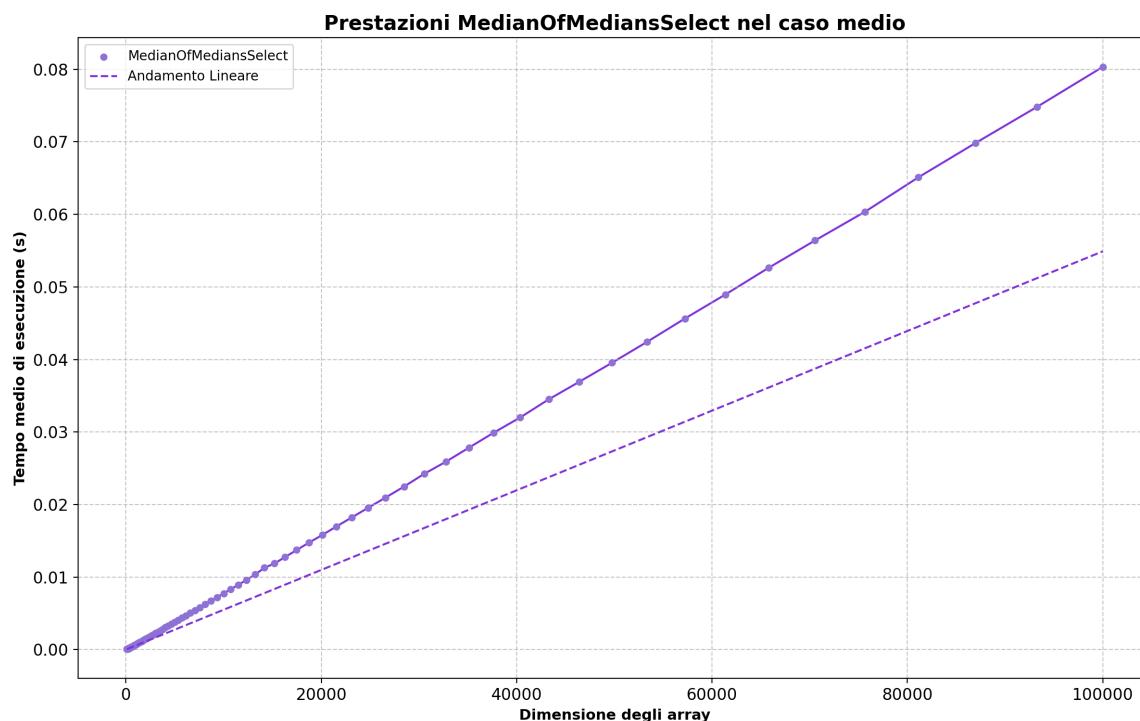
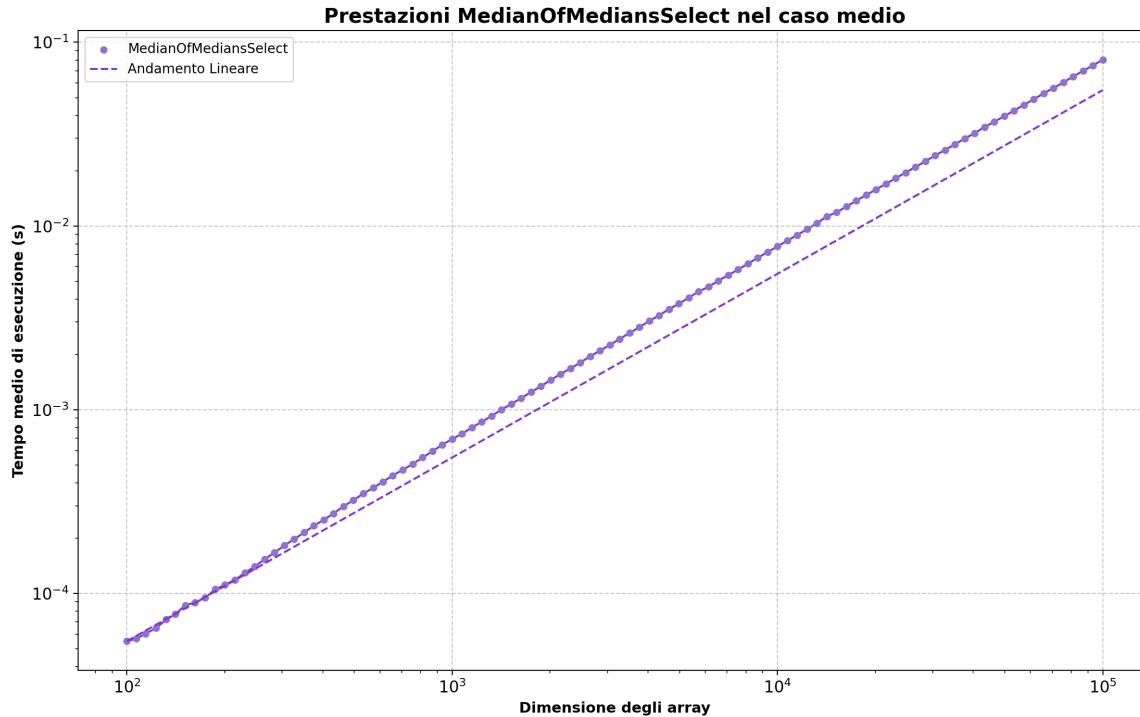
Questo algoritmo è implementato in una versione "quasi in place" in quanto non utilizza strutture ausiliarie, ma fa uso di chiamate ricorsive.

Il mediano viene trovato dalla procedura *RecMedianOfMediansSelect*, la quale esegue le seguenti operazioni:

1. Divide l'array in blocchi da 5 elementi e li ordina con *InsertionSort*.
2. Sposta il mediano di ogni blocco in testa all'array.
3. Richiama ricorsivamente sé stessa nella porzione di testa dell'array, contenente i mediani.
4. Sposta il mediano dei mediani in fondo all'array quando arriva a considerare un sub-array di testa di 5 elementi (caso base).
5. Chiama *Partition* e richiama ricorsivamente *RecMedianOfMedians* sulla metà di array che contiene l'elemento cercato fino a quando il mediano dei mediani non è il  $k - \text{esimo}$  elemento cercato.

L'algoritmo *MedianOfMediansSelect* ha una complessità nel caso peggiore, di  $\Theta(n)$ .

In generale possiamo dire che sia il meno variabile dei tre e che mantenga la stessa complessità anche nel caso medio.



## HeapSelect

Per trovare l'elemento più piccolo in una minHeap è sufficiente restituire la radice.

Allo stesso modo, per trovare l'elemento più grande in una maxHeap è sufficiente restituire la radice.

L'algoritmo sfrutta questa proprietà per trovare il  $k - \text{esimo}$  elemento più piccolo/grande, estraendo  $k - 1$  volte la radice.

Nello specifico, dopo aver validato l'indice  $k$ , se esso è minore della metà della lunghezza dell'heap, la funzione costruisce una min-heap da  $H1$  e inizializza  $H2$  come una min-heap contenente una tupla  $[elemento, posizione]$  in cui  $elemento$  corrisponde alla radice di  $H1$  e  $posizione$  corrisponde alla posizione di quell'elemento.

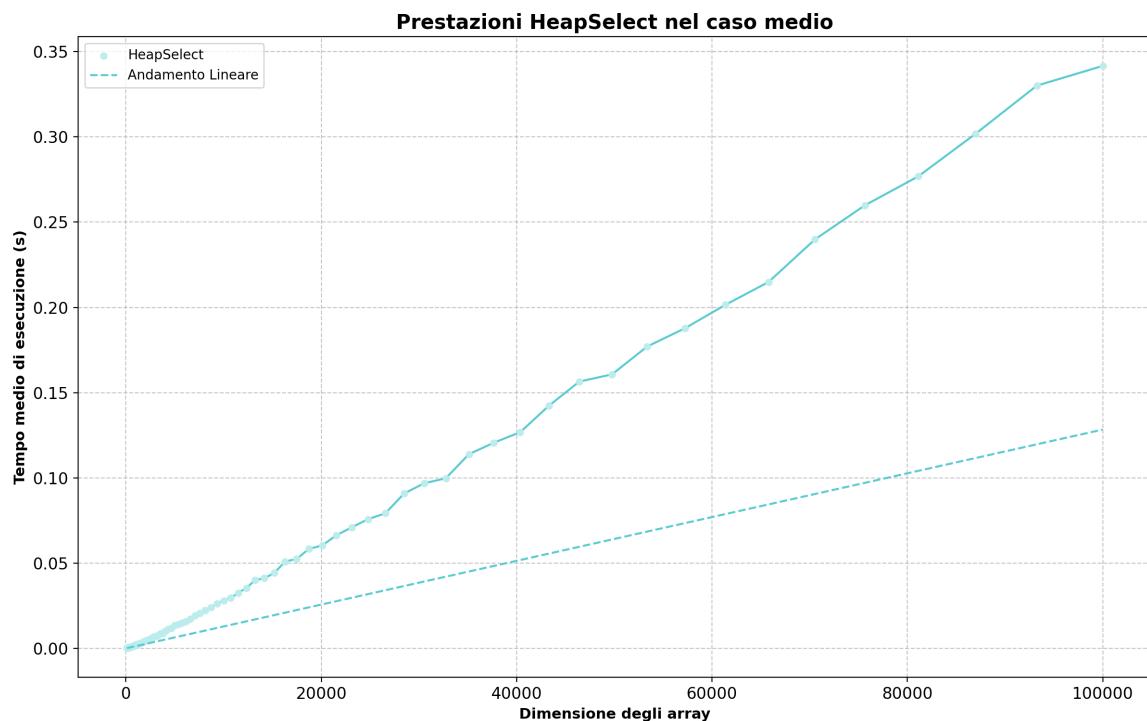
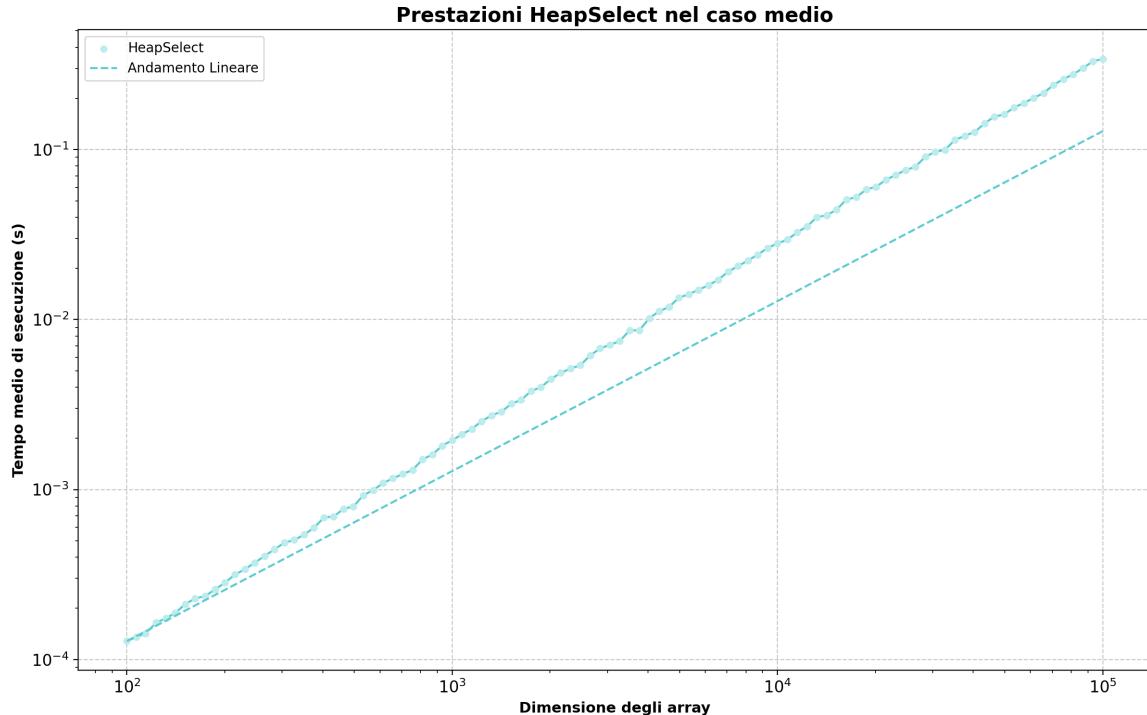
Altrimenti, costruisce una max-heap da  $H1$  e inizializza  $H2$  come una max-heap di tuple. In quest'ultimo caso, il problema viene riformulato per trovare il  $k - \text{esimo}$  elemento più grande, quindi  $k$  viene ricalcolato come  $\text{len}(H1) - k + 1$ .

Dopodiché la funzione itera il seguente processo:

1. Estrae la radice di  $H2$ .
2. Trova i figli sinistro e destro della posizione estratta nell'heap  $H1$ .
3. Inserisce i figli trovati in  $H2$  (se ce ne sono).

Dopo aver iterato  $k - 1$  volte, la funzione restituisce la radice di  $H2$  la quale corrisponde al  $k - \text{esimo}$  elemento più piccolo.

*HeapSelect* ha complessità  $O(n + k * \log(k))$  sia nel caso peggiore che nel caso medio. Dovrebbe pertanto essere preferibile a *QuickSelect* per  $k$  molto piccoli.



## Benchmark

Il *benchmark* misura il tempo medio di esecuzione di ciascun algoritmo nella lista *algoritmiDiSelezione*. Genera 100 (*passiSuccessione*) array di dimensione crescente seguendo una serie geometrica. La dimensione degli array va da 100 a 100000. Ad ogni passo, per ognuno degli algoritmi:

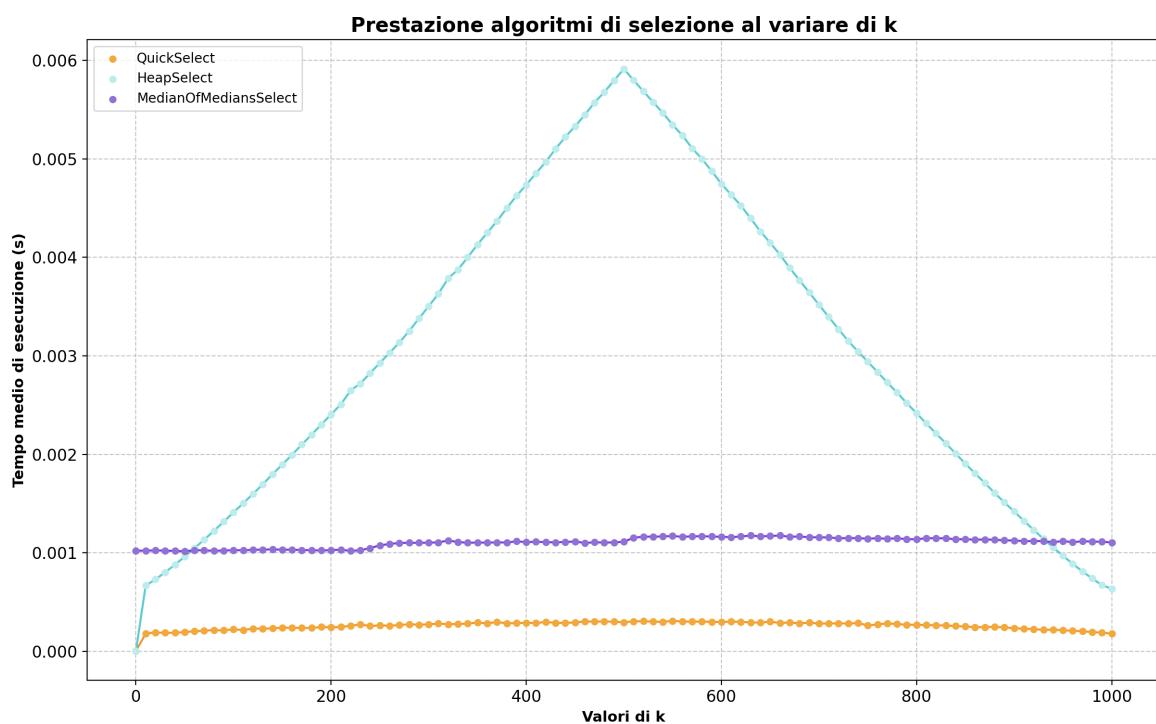
1. vengono eseguiti 500 (*testPerOgniN*) test con  $k$  random, per garantire che i tempi di esecuzione degli algoritmi siano medi.
  2. in una lista di liste (*tempiMedi*), vengono salvati:
    - 1) dimensione dell'array considerato
    - 2) tempi medi di esecuzione
- Ad ogni test:
3. l'array viene riempito con valori casuali in un range  $[-1000, 1000]$ .
  4. viene misurato il tempo di esecuzione (per ognuno degli algoritmi) assicurando un' errore relativo inferiore all' 1%.
  5. vengono salvati i tempi medi di esecuzione in una lista di liste (*tempiMedi*)
- Alla fine dei 100 passi, vengono salvati i dati dei tempi medi in dei file separati: uno per ogni algoritmo. Infine, viene chiamata la funzione che disegna il grafico prendendo i dati dai file generati.

## Eperimenti e risultati

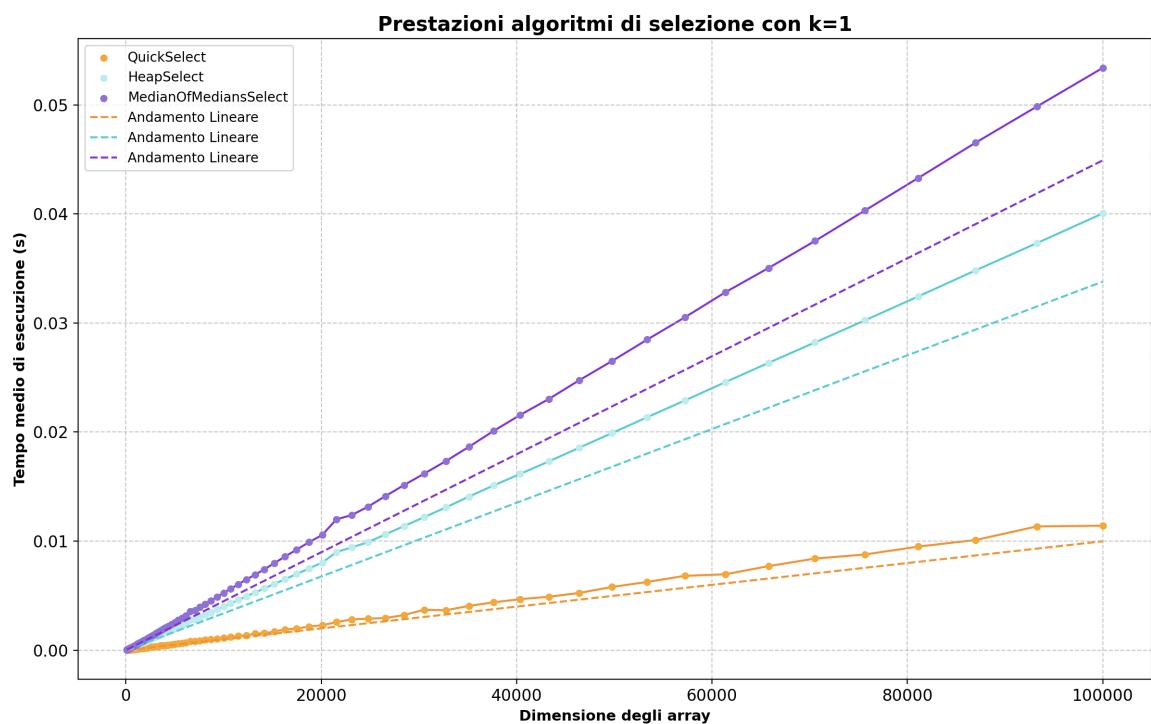
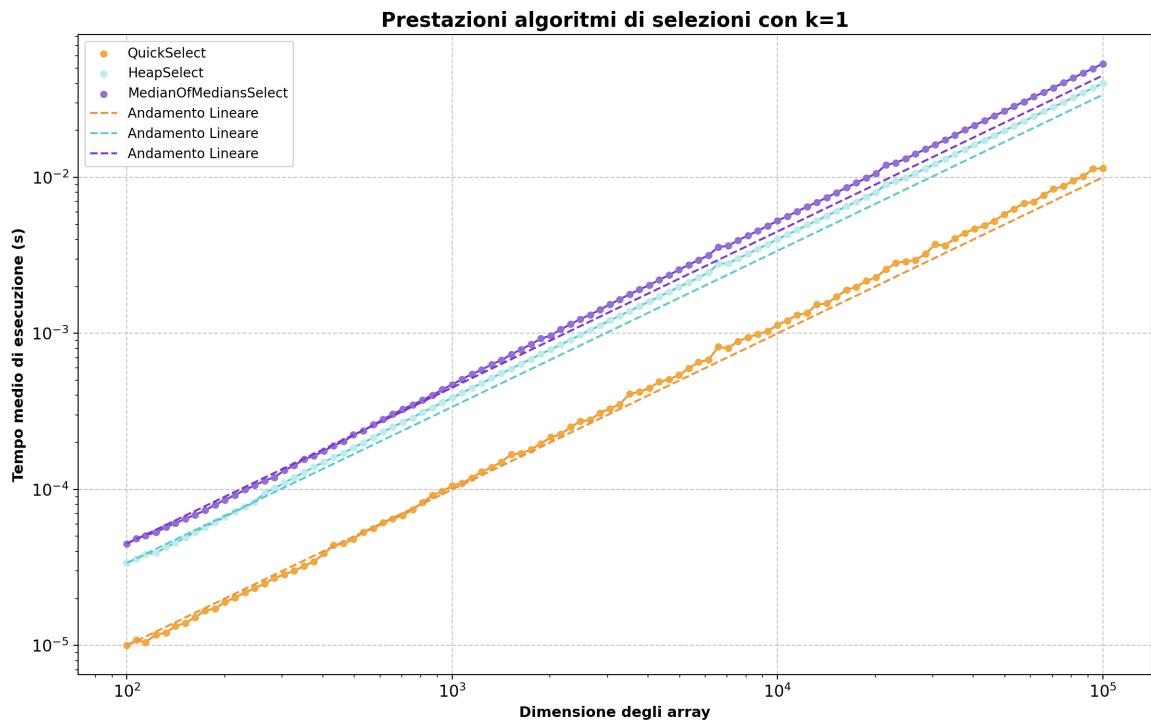
Per valutare la dipendenza dal parametro  $k$  scelto, abbiamo eseguito 4 ulteriori test.

1. Con la dimensione degli array fissata a 1000 e  $k = 0, 10, 20, \dots, 1000$
2. Fissando  $k = 1$
3. Fissando  $k = \text{lenArray}/2$
4. Fissando  $k = \text{lenArray}$

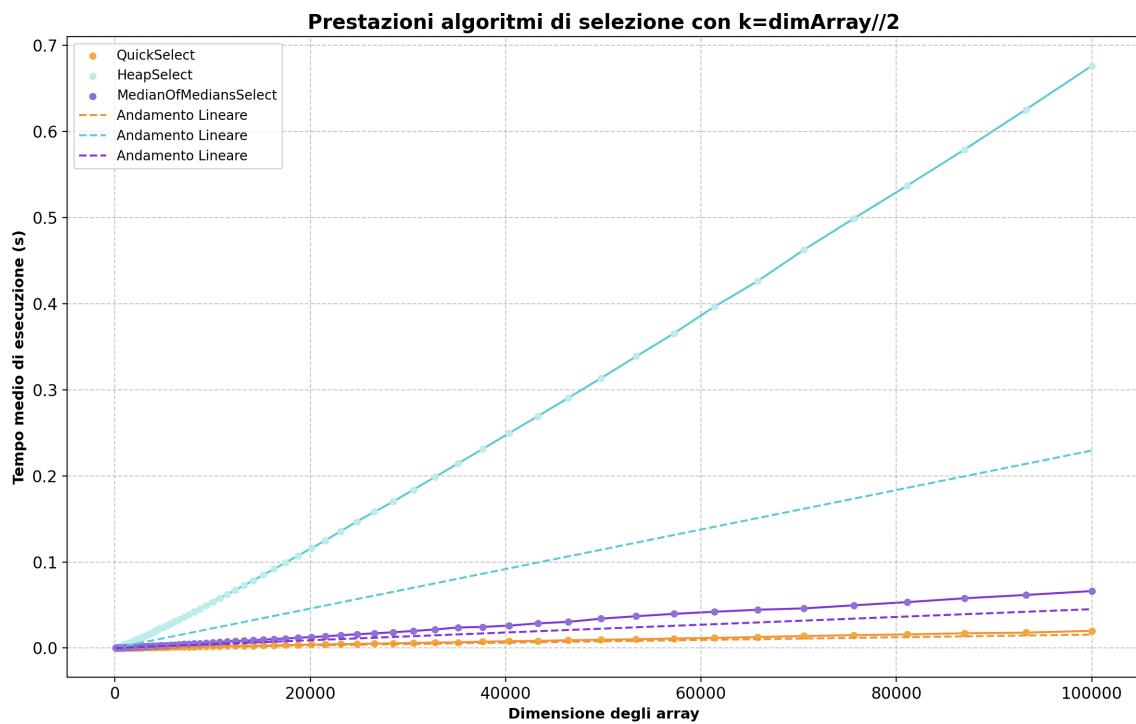
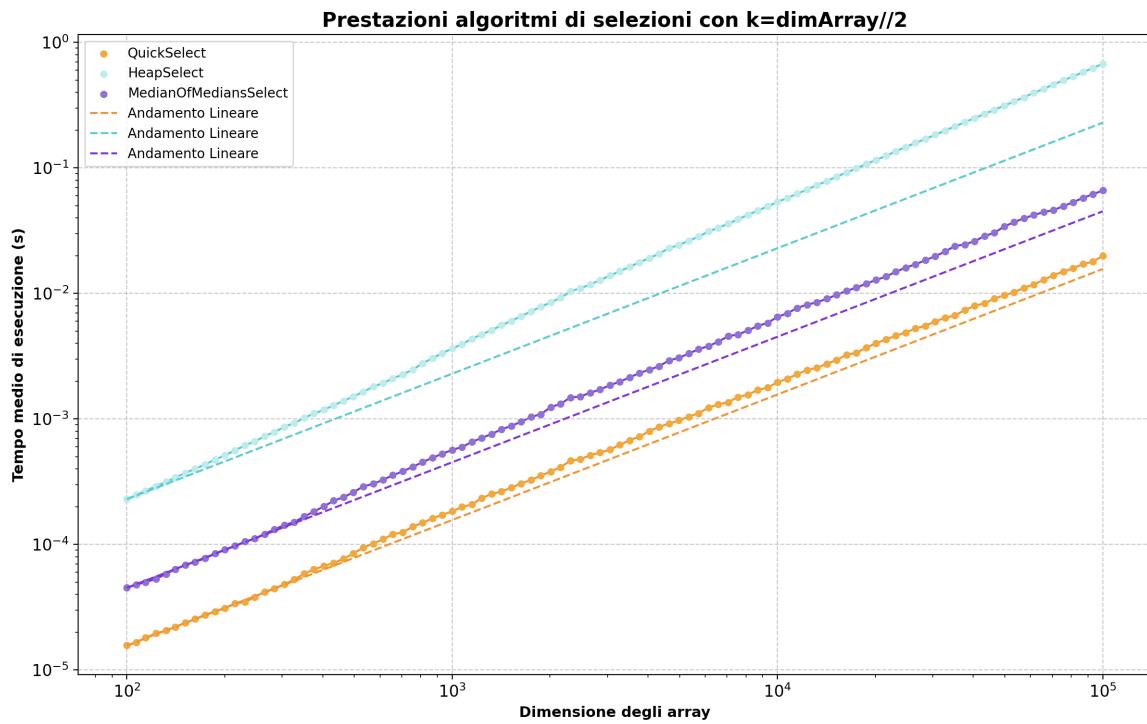
Nel primo test abbiamo ottenuto il seguente grafico da cui risulta particolarmente evidente la dipendenza da  $k$  di *HeapSelect*:



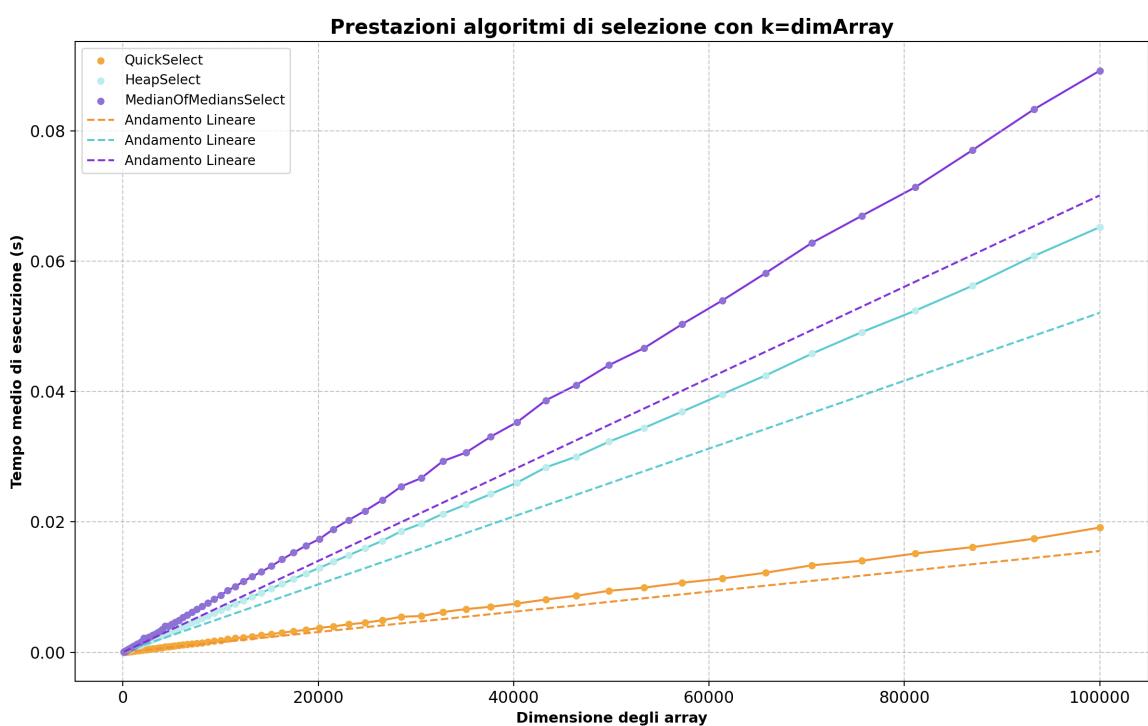
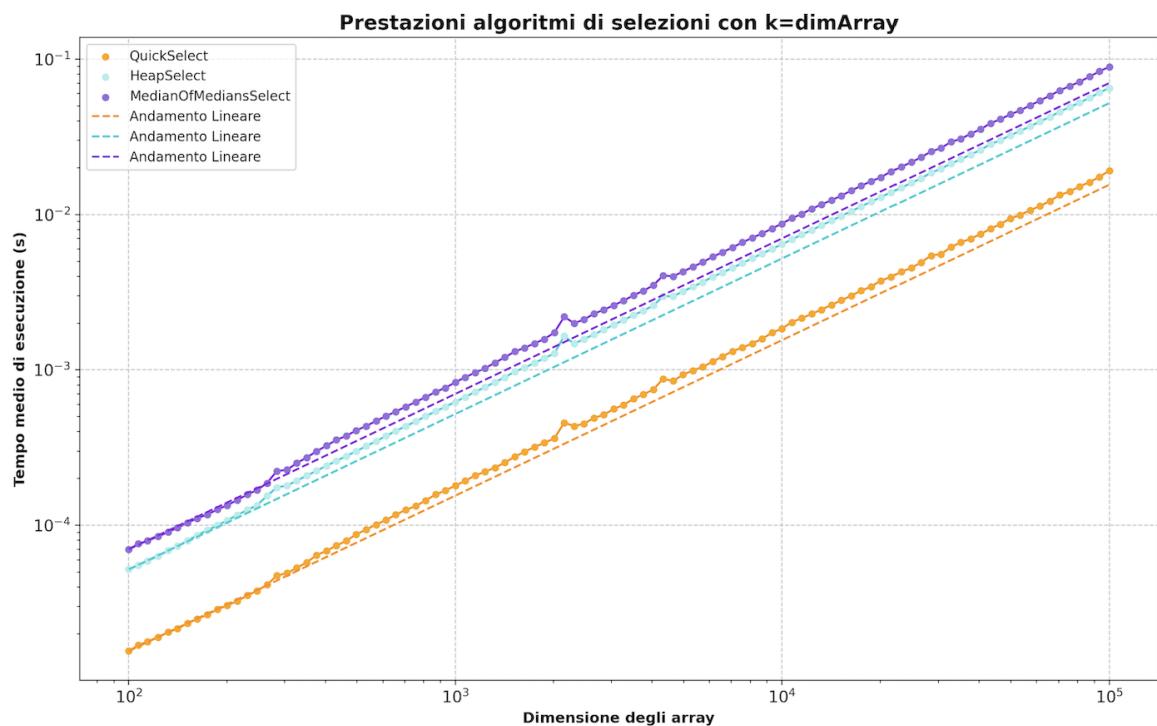
Per  $k = 1$  si ricade nel caso migliore possibile per *HeapSelect*, che risulta essere super lineare e meno variabile rispetto al caso medio.



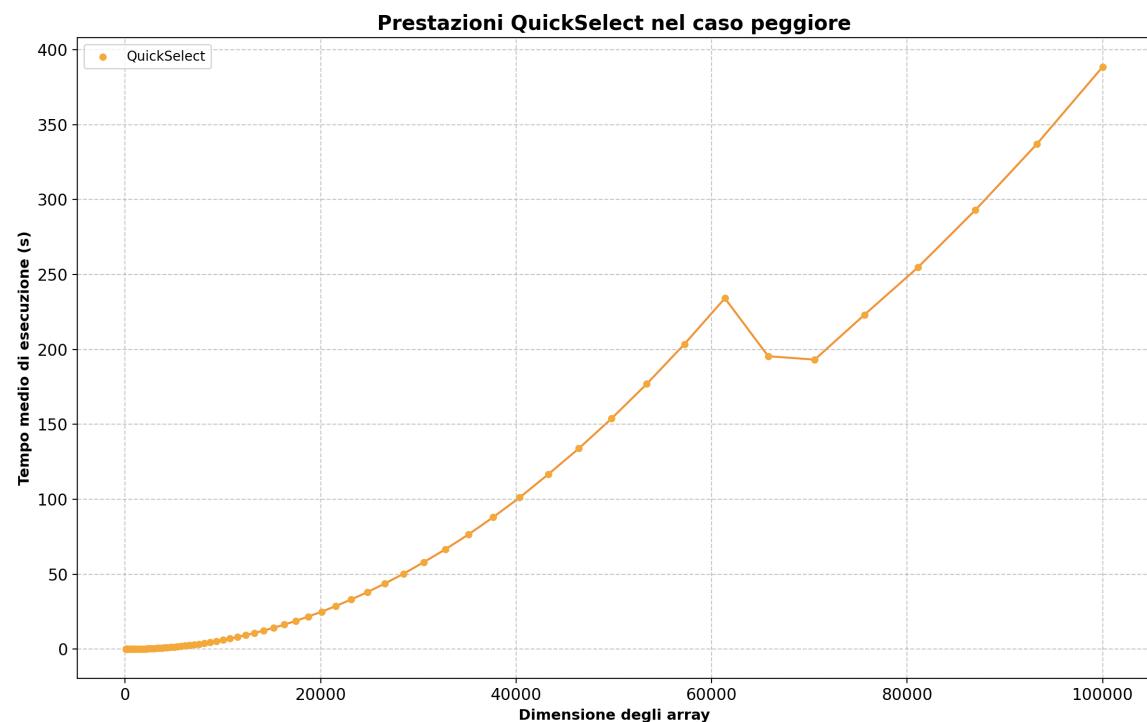
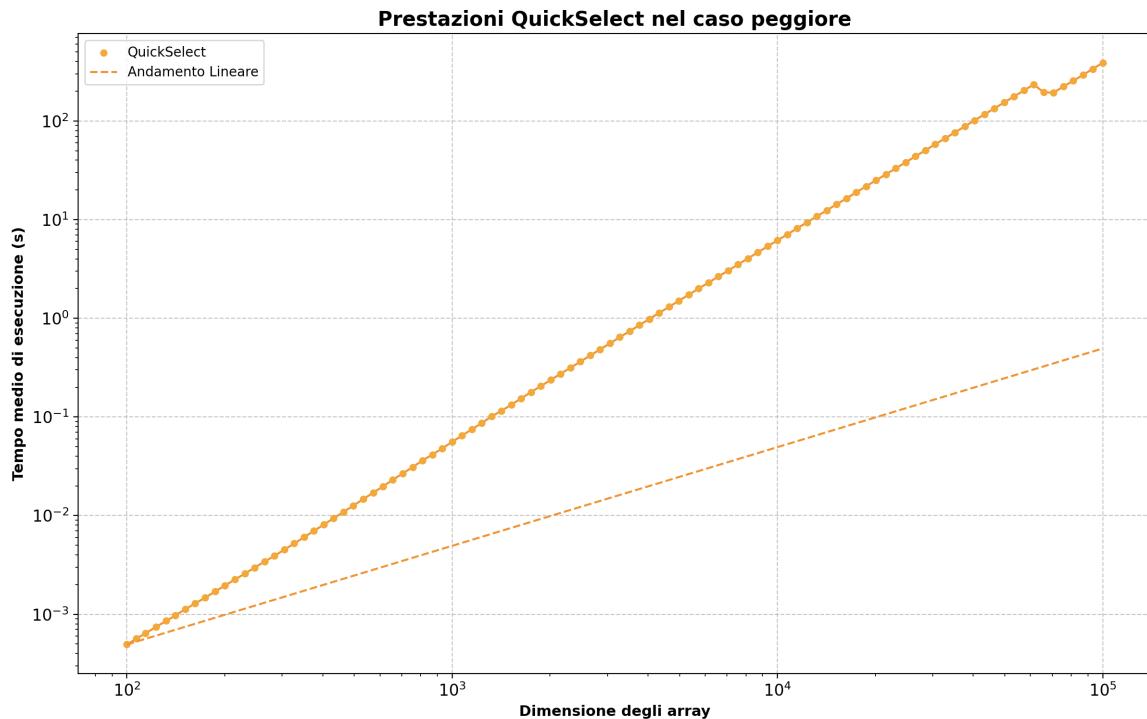
Per  $k = \text{len}(\text{array})//2$  si ricade nel caso peggiore per *HeapSelect*. Infatti i tempi di esecuzione risultano essere più elevati, l'andamento rimane molto simile a quello del caso medio. *QuickSelect* e *MedianOfMediansSelect* invece risultano essere invariati.



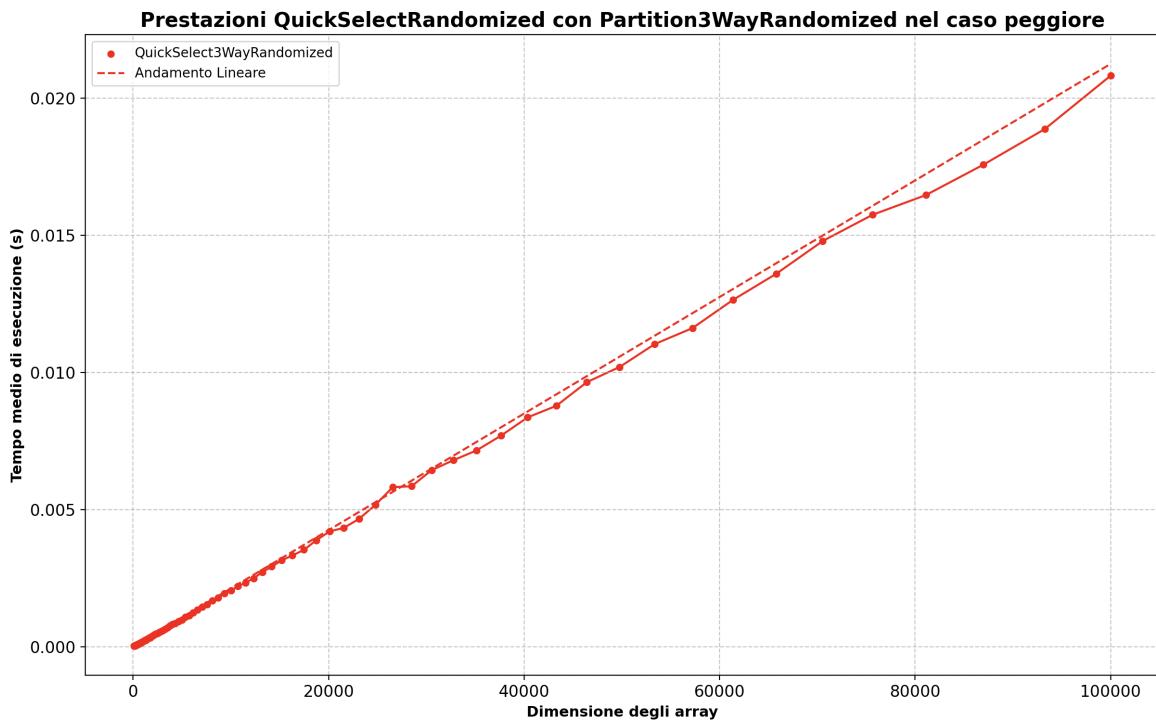
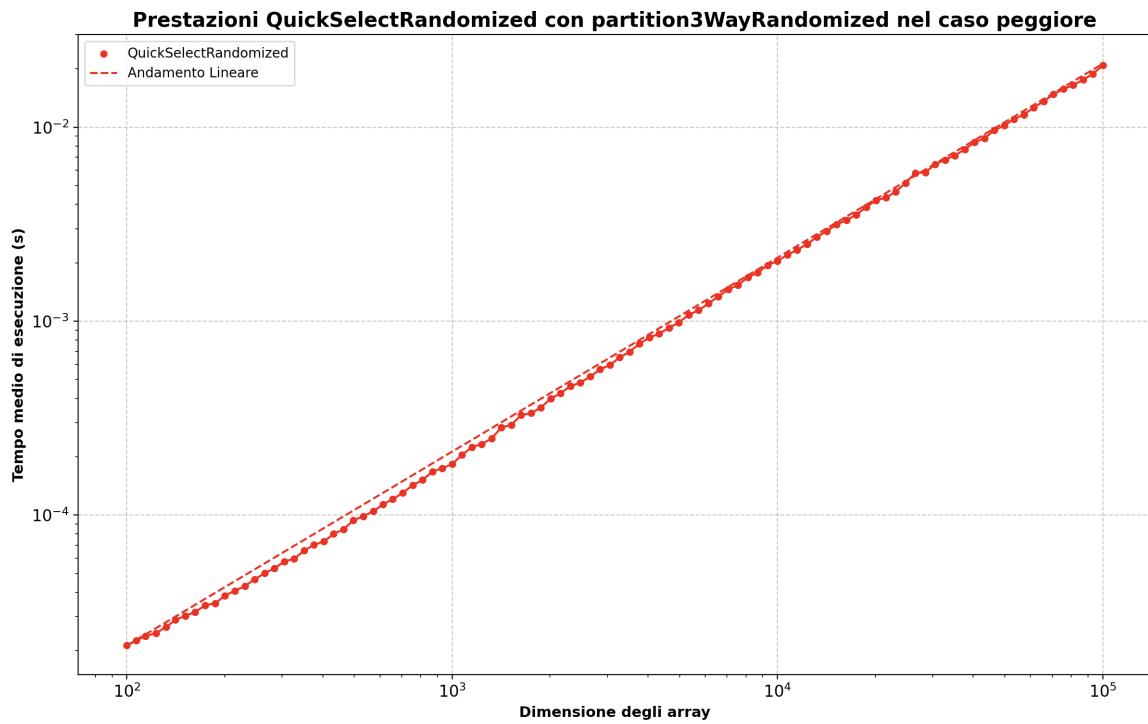
Per  $k = \text{lenArray}$  si ottiene una caso analogo a  $k = 1$ , caso migliore per *HeapSelect*.



Abbiamo visto che *QuickSelect* non è particolarmente legato a  $k$ . Abbiamo proceduto quindi con l'analisi dei tempi di esecuzione nel suo caso peggiore: con array ordinati in senso crescente,  $k = 1$  e  $testPerOgniN = 20$  (al posto di 500 in quanto il tempo di esecuzione del benchmark in quel caso, sarebbe stato ridicolmente più elevato). Inoltre avendo fissato  $k$ , e utilizzando array sempre ordinati, la variabilità delle prestazioni è stata ridotta, pertanto non c'era la necessità di eseguire così tanti  $testPerOgniN$ .  
Curiosità: per l'array di 100000 elementi, il tempo di esecuzione medio è stato di 388.4449569665987 secondi.



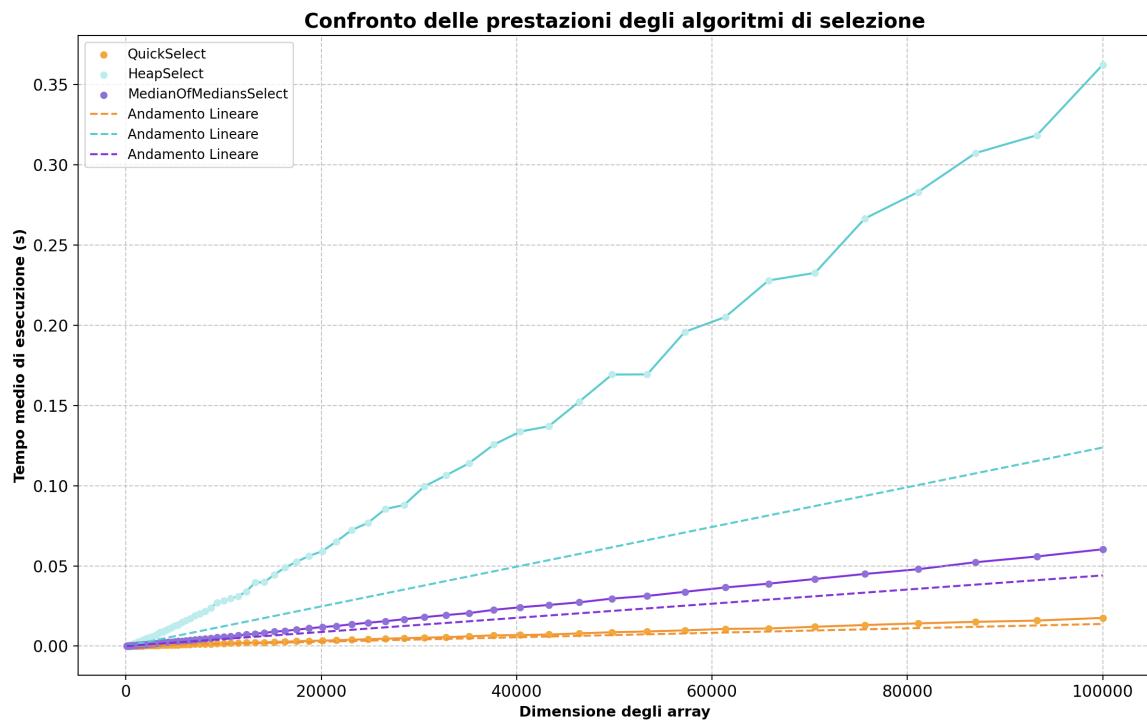
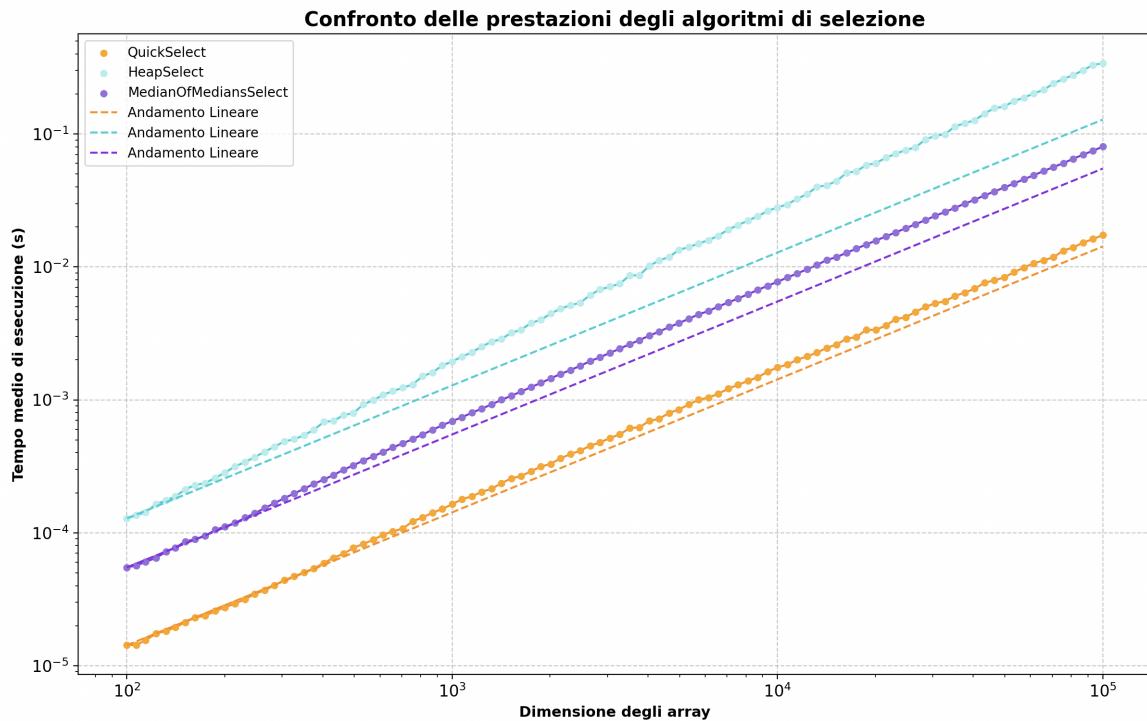
Abbiamo poi voluto provare a migliorare la complessità di *QuickSelect* utilizzando *partition3WayRandomized* e simulando le stesse condizioni del caso peggiore per *QuickSelect*(con *partition* normale), ovvero utilizzando array ordinati in senso crescente e  $k = 1$ , per mostrare che in quel caso, questa versione è molto più efficiente.  
Questo test, come tutti gli altri, fatta eccezione per quello del caso peggiore di *QuickSelect*, è stato eseguito con 500 *testPerOgniN*.  
È importante sottolineare che le condizioni proposte non rappresentano il caso peggiore per *QuickSelectRandomized*. Il suo caso peggiore, infatti, è impossibile da determinare in quanto i pivot di *partition3WayRandomized* vengono scelti casualmente.



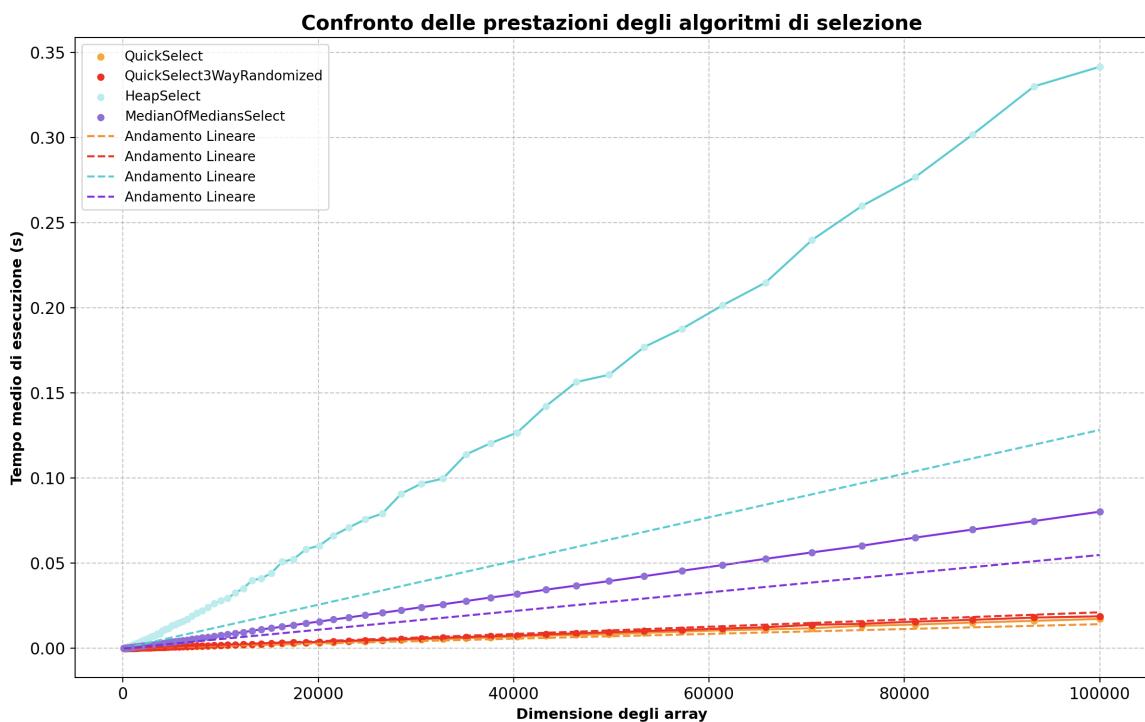
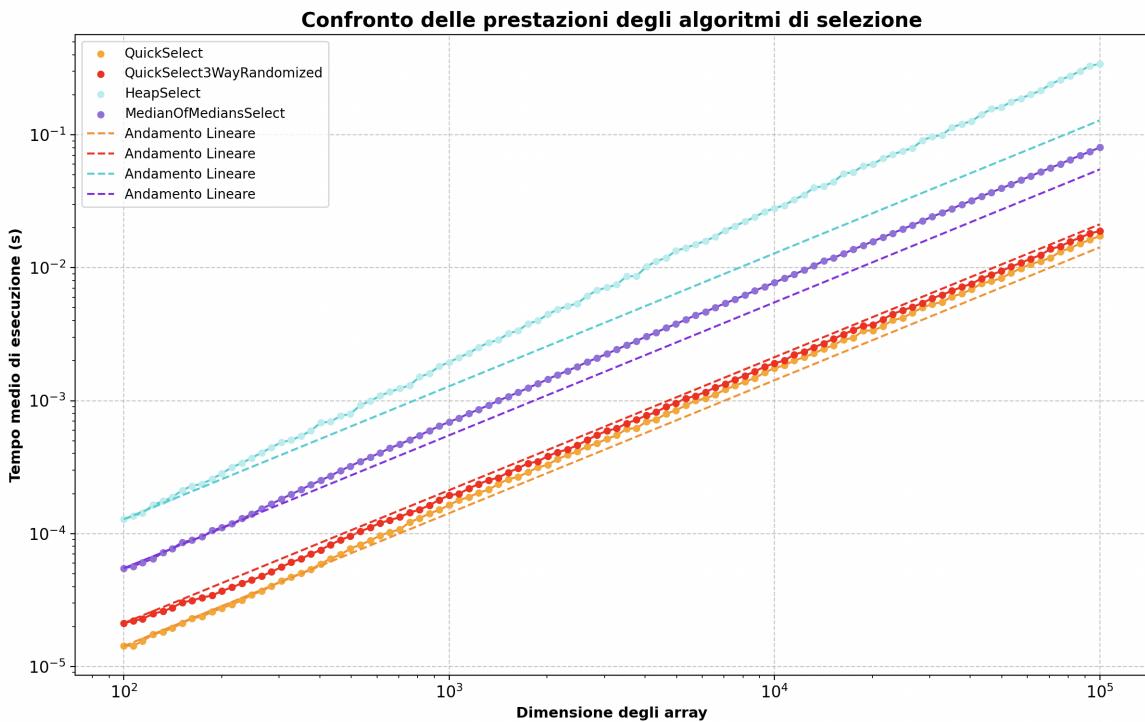
## Conclusioni e grafico finale

Limitandosi a studiare le complessità dei tre algoritmi proposti, si potrebbe pensare che *QuickSelect* sia il peggiore dei tre, infatti nel caso peggiore ha una complessità di  $\Theta(n^2)$ . Tuttavia, è curioso notare dal grafico, che per array casuali, nel caso medio, l'algoritmo più efficiente è proprio *QuickSelect*, seguito da *MedianOfMediansSelect*, *HeapSelect*.

Il *benchmark* per generare questo grafico ha eseguito 500 test per ogni dimensione dell'array, assicurando un buon caso medio per ciascun algoritmo.



Utilizzando *QuickSelectRandomized* con *partition3WayRandomized* nel caso medio abbiamo ottenuto i seguenti grafici:



## Problemi riscontrati e soluzioni trovate

- Durante il test di *QuickSelect* nel suo caso peggiore, con array ordinati superiori a 997 elementi e  $k = 1$ , l'esecuzione terminava con un errore "recursion depth" poiché python non riusciva a computare tutte quelle ricorsioni. Per questo motivo abbiamo trasformato il programma ricorsivo in uno iterativo. Tutti i grafici riportati in questa relazione sono stati generati dalla versione iterativa.
- Inizialmente per *HeapSelect* veniva utilizzata una chiave intera al posto di una tupla  $[elemento, posizione]$  per rappresentare un nodo. Questo però causava 2 problemi principali:
  - Nel caso di elementi ripetuti, per ottenere la posizione di tale chiave veniva utilizzata una scansione lineare dell'array, la quale ritorna la posizione della prima occorrenza dell'elemento specificato, ottenendo così sempre e solo i figli del nodo che occorreva per primo. È stato risolto tenendo traccia del numero di volte in cui apparivano le chiavi estratte, con una mappa delle occorrenze.
  - Avendo a disposizione solo le chiavi degli elementi, eravamo costretti a scorrere l'array ogni qual volta fosse necessario trovare la posizione di una di esse (per accedere ai suoi figli). Utilizzando una tupla  $[elemento, posizione]$ , all'occorrenza è possibile calcolare subito la posizione dei figli.
- Inizialmente per il *benchmark* l'idea era di salvare i dati generati in una matrice da passare alla funzione che disegna il grafico. Tuttavia abbiamo deciso di salvare i dati generati dal *benchmark* per ciascun algoritmo in file separati, nel caso in cui avessimo voluto:
  - Generare grafici esteticamente diversi.
  - Creare grafici comparativi per i diversi algoritmi.

Inoltre, all'inizio per ogni array venivano generati pochissimi test. Questo causava un pessimo caso medio, specialmente per *QuickSelect*, la cui variabilità è piuttosto elevata visto che è molto dipendente sia dalla scelta del parametro  $k$  che dalla disposizione degli elementi nell'array. Per questo motivo abbiamo scelto di eseguire il *benchmark* con 500 test per ogni dimensione dell'array generato.
- La CPU del macbook Air su cui abbiamo eseguito il *benchmark* raggiungeva temperature piuttosto elevate durante i test, di conseguenza anche le frequenze del processore venivano ridotte e i tempi di esecuzione degli algoritmi aumentavano più del previsto con l'aumentare della dimensione degli array fino a stabilizzarsi da un certo punto in poi. Questo però rendeva i grafici poco accurati. Si trattava di una distorsione minima ma facilmente evitabile. Per quanto possa sembrare ridicolo, abbiamo risolto il problema eseguendo il *benchmark* tenendo il laptop su un condizionatore acceso in un aula dell'università. Combinando la dissipazione attiva fornita dal condizionatore e quella passiva dovuta al case in alluminio, abbiamo estremamente ridotto la temperatura garantendo dei risultati più precisi.

## Appendice QuickSelect

```

...
quickSelect:
    Dato un array e un indice k:
    1) Restituisce il k-esimo elemento più piccolo presente nell'array.
    2) Restituisce None se k non è un valore accettabile
...
def quickSelect(A,k):
    #Validazione dell'indice k
    if k<1 or k>len(A):
        return None #Se k non e' accettabile stampa None
    else:
        #Inizializzazione delle variabili
        start = 0
        end = len(A) - 1
        k -= 1

        #Ricerca del k-esimo elemento
        while start < end:
            pivotPos = partition(A, start, end)
            if k >= start and k <= pivotPos-1: #se devo cercare prima del pivot
                end = pivotPos - 1
            elif k >= pivotPos+1 and k <= end: #se devo cercare dopo
                start = pivotPos + 1
            else: # se p = k-1
                break

        #Restituzione del risultato
        return A[k]
...

partition:
    Dato un array e un intervallo (p, q):
    1) Sposta a sinistra del pivot tutti gli elementi minori del pivot.
    2) Sposta a destra del pivot tutti gli elementi maggiori del pivot.
    3) Restituisce la posizione del pivot.
...
def partition(A,p,q):
    valorePivot=A[q]
    i=p-1
    for j in range(p,q+1):
        if A[j]<=valorePivot:
            i+=1
            swap(A,i,j)
    return i
...

swap:
    Dato un array e due indici posizionali:
    1) Scambia l'elemento in posizione i con quello in posizione j
...
def swap(A,i,j):
    temp=A[i]
    A[i]=A[j]
    A[j]=temp

def partition3WayRandomized(A, start, end):
    pos_a = partition(A, start, end, random.randint(start, end))
    if pos_a < (start + end)//2:
        if pos_a != end:
            pos_b = partition(A, pos_a+1, end, random.randint(pos_a+1, end))
        else:
            pos_b = end
    else:
        if pos_a != start:
            pos_b = partition(A, start, pos_a-1, random.randint(start, pos_a-1))
        else:
            pos_b = start

    if pos_a < pos_b:
        return (pos_a, pos_b)
    else:
        return (pos_b, pos_a)

def quickSelectRandomized(A,k):
    #Validazione dell'indice k
    if k<1 or k>len(A):
        return None #Se k non e' accettabile stampa None
    else:
        #Inizializzazione delle variabili
        start = 0
        end = len(A) - 1
        k -= 1

```

```

#Ricerca del k-esimo elemento
while start < end:
    pos_a, pos_b = partition3WayRandomized(A, start, end)
    if k == pos_a or k == pos_b:
        break
    elif k < pos_a:
        end = pos_a
    elif k < pos_b:
        start = pos_a
        end = pos_b
    else:
        start = pos_b

#Restituzione del risultato
return A[k]

```

## Appendice MedianOfMediansSelect

```

...
partition:
    Dato un array e un intervallo (p, q):
    1) Posiziona l'elemento in posizione q in posizione centrale dell'array.
    2) Mette a sinistra del pivot tutti gli elementi minori del pivot.
    3) Mette a destra del pivot tutti gli elementi maggiori del pivot.
...
def partition(A, p, q):
    pivot = A[q]
    i = p-1
    for j in range(p, q+1):
        if A[j] < pivot:
            i = i+1
            swap(A, i, j)
    i+=1
    swap(A, i, q)
    return i

...
swap:
    Dato un array e due indici posizionali:
    1) Scambia l'elemento in posizione i con quello in posizione j
...
def swap(A, i, j):
    t = A[i]
    A[i] = A[j]
    A[j] = t

...
insertionSort:
    Dato un array e un intervallo (start, end):
    1) Ordina la sezione di array delimitata dall'intervallo.
    2) Non ritorna l'array. Opera sull'array passato come parametro.
...
def insertionSort(A, start, end):
    for i in range(start, end+1):
        j = i
        while j > start and A[j] < A[j-1]:
            swap(A, j, j-1)
            j-=1

...
recMedianOfMediansSelect:
    Dato un array, un intervallo e un indice posizionale k:
    1) Divide l'array in blocchi da 5 elementi (fatta eccezione per l'ultimo che può averne meno) e li ordina con InsertionSort.
    2) Sposta il mediano di ogni blocco in testa all'array.
    3) Richiama ricorsivamente sé stessa nella porzione di testa dell'array, contenete i mediani.
    5) Sposta il mediano dei mediani in fondo all'array quando arriva a considerare un sub-array di testa di $5$ elementi (caso base).
    6) Chiama Partition e richiama ricorsivamente RecMedianOfMedians sulla metà di array che contiene l'elemento cercato fino a quando il mediano dei mediani non è il $k$-esimo elemento cercato.
...
def recMedianOfMediansSelect(A, start, end, k):
    if k<0 or k>len(A):
        return -1
    if end-start+1<=5:
        insertionSort(A, start, end)
        return(A[k-1])
    else:
        posMediano=start
        for x in range(start+4, end, 5):
            insertionSort(A, x-4, x)
            swap(A, posMediano, x-2)
            posMediano+=1
            if end-x<=5:
                insertionSort(A, x+1, end)
                swap(A, posMediano, (end+x)//2)
        recMedianOfMediansSelect(A, start, posMediano, (posMediano+start+1)//2)
        swap(A, end, (posMediano+start+1)//2)

        pivotPos = partition(A, start, end)
        if pivotPos == k-1:
            return A[k-1]
        elif pivotPos < k-1:
            start = pivotPos + 1
        else:
            end = pivotPos - 1
        return recMedianOfMediansSelect(A, start, end, k)

...
medianOfMediansSelect:
    funzione di appoggio utilizzata solo per l'inizializzazione dei parametri per iniziare l'albero delle ricorsioni
...
def medianOfMediansSelect(A, k):
    start = 0
    end = len(A)-1
    return recMedianOfMediansSelect(A, start, end, k)

```

## Appendice HeapSelect

```
from Heap import maxHeapInt, maxHeapTuple, minHeapInt, minHeapTuple
```

```

def heapSelect(H1, k):
    if len(H1) > 0 and k > 0 and k <= len(H1):
        if k < len(H1) // 2:
            #print("\nMinHeap\n")
            minHeapInt.buildHeap(H1)
            H2 = minHeapTuple([ ])
        else:
            #print("\nMaxHeap\n")
            maxHeapInt.buildHeap(H1)
            H2 = maxHeapTuple([ ])
            k = len(H1) - k + 1
        # Node of H2 are tuple that have sa 1o element the a key of H1
        # and as 2o the position of that key in H1
        H2.push((H1[0], 0))

    #print("H1 : {} \nH2 : {}".format(H1, H2))

    for i in range(1, k):
        node = H2.pop()
        l = maxHeapInt.getLeft(H1, node[1])
        r = maxHeapInt.getRight(H1, node[1])
        if l[0] != None : H2.push(l)
        if r[0] != None : H2.push(r)

    #print("H1 : {} \nH2 : {}".format(H1, H2))

    return H2.pop()[0]
else:
    return -1

```

\*\*\*\*\*  
Set of function that modifies the given array  
- Organize the array satisfying MaxHeap invariant  
- Perform Heap operation  
\*\*\*\*\*

```

class maxHeapInt:
    def getLeft(H, i):
        lPos = 2 * i + 1
        if i >= 0 and lPos < len(H):
            return (H[lPos], lPos)
        else:
            return (None, -1)

    def getRight(H, i):
        rPos = 2 * i + 2
        if i >= 0 and rPos < len(H):
            return (H[rPos], rPos)
        else:
            return (None, -1)

    def getParent(H, i):
        pPos = (i - 1) // 2
        if i > 0 and i < len(H):
            return (H[pPos], pPos)
        else:
            return (None, -1)

    def buildHeap():
        for i in range((len(H) // 2) - 1, -1, -1):
            maxHeapInt._heapify(H, i)

    def __heapify(H, i):
        l = maxHeapInt.getLeft(H, i)
        r = maxHeapInt.getRight(H, i)
        if l[0] != None and l[0] > H[i]:
            m = l[1]
        else:
            m = i
        if r[0] != None and r[0] > H[m]:
            m = r[1]
        if m != i:
            maxHeapInt.__swap(H, i, m)
            maxHeapInt.__heapify(H, m)

    def __swap(H, i, j):
        tmp = H[i]
        H[i] = H[j]
        H[j] = tmp

class minHeapInt:
    def getLeft(H, i):
        lPos = 2 * i + 1
        if i >= 0 and lPos < len(H):
            return (H[lPos], lPos)
        else:
            return (None, -1)

    def getRight(H, i):
        rPos = 2 * i + 2
        if i >= 0 and rPos < len(H):
            return (H[rPos], rPos)
        else:
            return (None, -1)

    def getParent(H, i):
        pPos = (i - 1) // 2
        if i > 0 and i < len(H):
            return (H[pPos], pPos)
        else:
            return (None, -1)

    def buildHeap():
        for i in range(len(H) // 2, -1, -1):
            minHeapInt.__heapify(H, i)

    def __heapify(H, i):
        l = minHeapInt.getLeft(H, i)

```

```

r = minHeapInt.getRight(H, i)
if l[0] != None and l[0] < H[i]:
    m = l[1]
else:
    m = i
if r[0] != None and r[0] < H[m]:
    m = r[1]
if m != i:
    minHeapInt.__swap(H, i, m)
    minHeapInt.__heapify(H, m)

def __swap(H, i, j):
    tmp = H[i]
    H[i] = H[j]
    H[j] = tmp

"""
Classes that realize Heap with node that can store indexed element
the 1° element is considered for the Heap organization (to preserve the invariant)
"""
class maxHeapTuple:
    def __init__(self, A):
        self.H = A
        self.__buildHeap()

    def getHeapSize(self):
        return len(self.H)

    def getLeftPosition(self, pos):
        l = 2 * pos + 1
        if pos >= 0 and l < self.getHeapSize():
            return l
        else:
            return -1

    def getRightPosition(self, pos):
        r = 2 * pos + 2
        if pos >= 0 and r < self.getHeapSize():
            return r
        else:
            return -1

    def getParentPosition(self, pos):
        if pos > 0 and pos < self.getHeapSize():
            return (pos - 1) // 2
        else:
            return -1

    def getNode(self, pos):
        if pos >= 0 and pos < self.getHeapSize():
            return self.H[pos]
        else:
            return None

    def getLeft(self, pos):
        l = self.getLeftPosition(pos)
        if l != -1:
            return self.getNode(l)
        else:
            return None

    def getRight(self, pos):
        r = self.getRightPosition(pos)
        if r != -1:
            return self.getNode(r)
        else:
            return None

    def getParent(self, pos):
        p = self.getParentPosition(pos)
        if p != -1:
            return self.getNode(p)
        else:
            return None

    def __buildHeap(self):
        for pos in range(self.getHeapSize()//2, -1, -1):
            self.__heapify(pos)

    def __heapify(self, pos):
        l = self.getLeftPosition(pos)
        r = self.getRightPosition(pos)
        if l != -1 and self.getNode(l)[0] > self.getNode(pos)[0]:
            m = l
        else:
            m = pos
        if r != -1 and self.getNode(r)[0] > self.getNode(m)[0]:
            m = r
        if m != pos:
            self.__swap(m, pos)
            self.__heapify(m)

    def __swap(self, i, j):
        tmp = self.getNode(i)
        self.H[i] = self.getNode(j)
        self.H[j] = tmp

    def push(self, node):
        self.H.append(node)
        pos = self.getHeapSize() - 1
        while pos > 0 and self.getNode(pos)[0] > self.getParent(pos)[0]:
            self.__swap(pos, self.getParentPosition(pos))
            pos = self.getParentPosition(pos)

    def pop(self):
        if self.getHeapSize() > 0:
            node = self.getNode(0)
            self.__swap(0, self.getHeapSize() - 1)
            self.H.pop(self.getHeapSize() - 1)
            self.__heapify(0)

```

```

        return node
    else:
        return None

def __str__(self):
    str = "[ "
    for x in self.H:
        str += "({}, {}) ".format(x[0], x[1])
    str += "]"
    return str

class minHeapTuple:
    def __init__(self, A):
        self.H = A
        self.__buildHeap()

    def getHeapSize(self):
        return len(self.H)

    def getLeftPosition(self, pos):
        l = 2 * pos + 1
        if pos >= 0 and l < self.getHeapSize():
            return l
        else:
            return -1

    def getRightPosition(self, pos):
        r = 2 * pos + 2
        if pos >= 0 and r < self.getHeapSize():
            return r
        else:
            return -1

    def getParentPosition(self, pos):
        if pos > 0 and pos < self.getHeapSize():
            return (pos - 1) // 2
        else:
            return -1

    def getNode(self, pos):
        if pos >= 0 and pos < self.getHeapSize():
            return self.H[pos]
        else:
            return None

    def getLeft(self, pos):
        l = self.getLeftPosition(pos)
        if l != -1:
            return self.getNode(l)
        else:
            return None

    def getRight(self, pos):
        r = self.getRightPosition(pos)
        if r != -1:
            return self.getNode(r)
        else:
            return None

    def getParent(self, pos):
        p = self.getParentPosition(pos)
        if p != -1:
            return self.getNode(p)
        else:
            return None

    def __buildHeap(self):
        for pos in range(self.getHeapSize()//2, -1, -1):
            self.__heapify(pos)

    def __heapify(self, pos):
        l = self.getLeftPosition(pos)
        r = self.getRightPosition(pos)
        if l != -1 and self.getNode(l)[0] < self.getNode(pos)[0]:
            m = l
        else:
            m = pos
        if r != -1 and self.getNode(r)[0] < self.getNode(m)[0]:
            m = r
        if m != pos:
            self.__swap(m, pos)
            self.__heapify(m)

    def __swap(self, i, j):
        tmp = self.getNode(i)
        self.H[i] = self.getNode(j)
        self.H[j] = tmp

    def push(self, node):
        pos = self.getHeapSize() - 1
        while pos > 0 and self.getNode(pos)[0] < self.getParent(pos)[0]:
            self.__swap(pos, self.getParentPosition(pos))
            pos = self.getParentPosition(pos)

    def pop(self):
        if self.getHeapSize() > 0:
            node = self.getNode(0)
            self.__swap(0, self.getHeapSize() - 1)
            self.H.pop(self.getHeapSize() - 1)
            self.__heapify(0)
            return node
        else:
            return None

    def __str__(self):
        str = "[ "
        for x in self.H:
            str += "({}, {}) ".format(x[0], x[1])
        str += "]"
        return str

```

```
str += "]"
return str
```

## Appendice Benchmark

```
from HeapSelect import heapSelect
from QuickSelect import quickSelect
from partition3WayRandomized import quickSelectRandomized
from MedianOfMediansSelect import medianOfMediansSelect
from DisegnaGrafico import disegnaGrafico

import random
import time
import os
#####
##### ! DISCLAIMER ! #####
##### ! IL SEGUENTE BENCHMARK NON È STATO PENSATO PER ESSERE ESEGUITO SU WINDOWS.
##### ! È STATO TESTATO SU MACOSX E UBUNTU, PERTANTO SCONSIGLIAMO L'ESECUZIONE SU SISTEMI OPERATIVI DIFFERENTI.
##### ! IN PARTICOLARE I TEMPI DI ESECUZIONE DEL BENCHMARK, SU WINDOWS, POSSONO RISULTARE ESTREMEAMENTE PIÙ ELEVATI.
##### ! INOLTRE IL GRAFICO NON VERRÀ GENERATO.

...
tempoMinimoMisurabile:
    Restituisce il tempo minimo misurabile dal contatore time.monotonic().
...
def tempoMinimoMisurabile():
    inizioMisurazione = time.monotonic()
    while time.monotonic() == inizioMisurazione:
        pass
    fineMisurazione = time.monotonic()
    return fineMisurazione - inizioMisurazione

...
misuraTempi:
    1) Misura in modo accurato il tempo di esecuzione di ciascun algoritmo nella lista algoritmiDiSelezione.
    2) Garantisce un errore relativo inferiore all' 1%.
    3) Aggiunge il tempo medio misurato al contatore del tempo medio di ogni algoritmo eseguito.
...
tMinMisurabile = tempoMinimoMisurabile() * (1 + (1 / 0.01)) # max_rel_error = 1% = 0.01

def misuraTempi(array, k, algoritmiDiSelezione, tempiMedi):
    for i, func in enumerate(algoritmiDiSelezione):
        count = 0
        tempoIniziale = time.monotonic()
        while time.monotonic() - tempoIniziale < tMinMisurabile:
            func(array.copy(), k)
            count += 1
        durata = time.monotonic() - tempoIniziale
        tempiMedi[i] += (durata / count)

...
benchmark:
    1) Misura il tempo medio di esecuzione di ciascun algoritmo nella lista algoritmiDiSelezione.
    2) Genera passiSuccessione(100) array di dimensione crescente seguendo una serie geometrica.
       La dimensione degli array va da 100 a 100000.
    3) Ad ogni passo vengono eseguiti testPerOgniN(500) test con k random per garantire che i tempi di esecuzione degli algoritmi siano medi.
    4) In ogni test viene riempito l'array di dimensione prefissata, con valori casuali in un range [-1000,1000].
    5) In ogni test viene misurato il tempo di esecuzione.
    6) Alla fine dei test viene salvata la media dei tempi per quel passo della successione in una lista di liste tempiMedi.
    7) Ad ogni passo, dopo aver eseguito i testPerOgniN(500), viene salvata la dimensione dell'array su cui è stato eseguito quel passo.
    8) Alla fine dei 100 passi, vengono salvati i dati dei tempi medi in dei file separati. Uno per ogni algoritmo.
    9) Infine, viene richiamata la funzione che disegna il grafico prendendo i dati dai file generati.
...
# Lista degli algoritmi di selezione
algoritmiDiSelezione = [quickSelect, quickSelectRandomized, heapSelect, medianOfMediansSelect]
nomiAlgoritmiDiSelezione = ["QuickSelect", "QuickSelectRandomizedPTW", "HeapSelect", "MedianOfMediansSelect"]

# Inizializza le liste per i tempi medi
tempiMedi = [] for _ in algoritmiDiSelezione] #lista di tante liste quanti sono gli algoritmi di selezione
dimensioneArrayGenerati = []

def benchmark():
    A = 100 # Inizio della serie geometrica
    B = (100000 / 100) ** (1 / 99) # Calcolo di B per ottenere n finale di 100000 (radice 99-esima)
    passiSuccessione = 100
    testPerOgniN = 1

    for i in range(passiSuccessione):
        dimArray = int(A * (B ** i))
        print("Eseguo il passo {} / {} della successione \t len(A) : {}".format(i+1, passiSuccessione, dimArray))

        # Inizializza i tempi medi per questa dimensione dell'array
        tempiMediPasso = [0] * len(algoritmiDiSelezione)

        for _ in range(testPerOgniN):
            array = [random.randint(-1000, 1000) for _ in range(dimArray)]
            k = random.randint(1, dimArray)
            misuraTempi(array, k, algoritmiDiSelezione, tempiMediPasso)

        for j in range(len(algoritmiDiSelezione)):
            tempiMedi[j].append(tempiMediPasso[j] / testPerOgniN)

        dimensioneArrayGenerati.append(dimArray)

    # Imposta la directory di lavoro relativa alla posizione di questo file
    directoryPath = os.path.join(os.path.dirname(__file__), 'RisultatiBenchmark')

    # Verifica se la cartella esiste, altrimenti crea la
    if not os.path.exists(directoryPath):
        os.makedirs(directoryPath)

    # Costruisce i percorsi dei file relativi a questa directory
    dimensioniArray_path = os.path.join(directoryPath, 'dimensioniArray.txt')
```

```

# Scrive la dimensione di ogni array generato in un file.
with open(dimensioniArray_path, 'w') as f:
    for n in dimensioneArrayGenerati:
        f.write(f"{n}\n")

# Scrive i risultati dei tempi di esecuzione in file di testo separati. Uno per ogni algoritmo.
for i, nome in enumerate(nomiAlgoritmiDiSelezione):
    file_path = os.path.join(directoryPath, f'tempi{nome}.txt')
    with open(file_path, 'w') as f:
        for tempo in tempiMedi[i]:
            f.write(f'{tempo}\n')

print("Benchmark partito")
benchmark()
print("Benchmark terminato. Creo il grafico.")
disegnaGrafico()

```

## Appendice DisegnaGrafico

```

import matplotlib.pyplot as plt
import os

...
leggiDaFile:
    Legge da un file e salva in un array i valori letti.
...
def leggiDaFile(percorsoFile):
    valori = []
    with open(percorsoFile, 'r') as file:
        for linea in file:
            valore = float(linea.strip())
            valori.append(valore)
    return valori

...
disegnaGrafico:
    1) Crea il grafico su scala logaritmica delle prestazioni di quickSelect, heapSelect, medianOfMediansSelect.
    2) Sull'asse x sono presenti le dimensioni degli array su cui sono state effettuate le misurazioni dei tempi di esecuzione.
    3) Sull'asse y sono presenti i tempi di esecuzione degli algoritmi di selezione indicati.
    4) I dati utilizzati per tracciare il grafico devono essere stati salvati nella stessa percorso in cui si trova questo programma.
    5) Il grafico generato è composto da linee e punti. Vengono rispettivamente utilizzate le funzioni plot e scatter.
    6) Il grafico generato NON viene salvato automaticamente.
...

def disegnaGrafico():
    # Imposta la directory di lavoro relativa alla posizione di questo file
    directoryPath = os.path.join(os.path.dirname(__file__), 'RisultatiBenchmark')

    percorsoFileDimensione = os.path.join(directoryPath, 'dimensioniArray.txt')
    dimensione = leggiDaFile(percorsoFileDimensione)

    #Quick Random
    percorsoFileTempoQuickRandom = os.path.join(directoryPath, 'tempiQuickSelectRandomizedPTW.txt')
    tempoDiEsecuzioneQuickSelectRandom = leggiDaFile(percorsoFileTempoQuickRandom)

    percorsoFileTempoQuick = os.path.join(directoryPath, 'tempiQuickSelect.txt')
    tempoDiEsecuzioneQuickSelect = leggiDaFile(percorsoFileTempoQuick)

    #Heap
    percorsoFileTempoHeapSelect = os.path.join(directoryPath, 'tempiHeapSelect.txt')
    tempoDiEsecuzioneHeapSelect = leggiDaFile(percorsoFileTempoHeapSelect)

    #Median
    percorsoFileTempoMedianOfMedians = os.path.join(directoryPath, 'tempiMedianOfMediansSelect.txt')
    tempoDiEsecuzioneMedianOfMedians = leggiDaFile(percorsoFileTempoMedianOfMedians)

    #Viene usato il subplot per unire i due grafici creati(punti e linee)
    fig, ax = plt.subplots(figsize=(10, 6)) #Il grafico avrà dimensione 10x6 pollici

    #Titolo del grafico
    plt.title('Prestazioni QuickSelect nel caso medio', fontsize = 15, fontweight = 'bold')
    ax.grid(True, linestyle='--', alpha=0.7)
    ax.tick_params(labelsize=12)

    #Disegna i punti. La priorità è 1 così che i punti vengano disegnati sopra le linee. s è lo spessore del punto
    plt.scatter(dimensione, tempoDiEsecuzioneQuickSelect, color='orange', label = 'QuickSelect', zorder=2, s=20) #Quick
    plt.scatter(dimensione, tempoDiEsecuzioneQuickSelectRandom, color='red', label = 'QuickSelect3WayRandomized', zorder=2, s=20) #QuickRandom
    plt.scatter(dimensione, tempoDiEsecuzioneHeapSelect, color='paleturquoise', label = 'HeapSelect', zorder=2, s=20) #Heap
    plt.scatter(dimensione, tempoDiEsecuzioneMedianOfMedians, color='mediumpurple', label = 'MedianOfMediansSelect', zorder=2, s=20) #Median

    #Disegna le linee. La priorità è 2 così che le linee vengano disegnate sotto i punti.
    ax.plot(dimensione, tempoDiEsecuzioneQuickSelect, '-', color='darkorange', zorder=1) #Quick
    ax.plot(dimensione, tempoDiEsecuzioneQuickSelectRandom, '-', color='red', zorder=1) #QuickRandom
    ax.plot(dimensione, tempoDiEsecuzioneHeapSelect, '-', color='darkturquoise', zorder=1) #Heap
    ax.plot(dimensione, tempoDiEsecuzioneMedianOfMedians, '-', color='blueviolet', zorder=1) #Median

    #Disegna la funzione lineare di riferimento per QuickSelect
    k = tempoDiEsecuzioneQuickSelect[0] / dimensione[0]
    tempoLineare = [(k * x) for x in dimensione]
    ax.plot(dimensione, tempoLineare, '--', color='darkorange', label = 'Andamento Lineare', zorder=3)

    #Disegna la funzione lineare di riferimento per QuickSelectRandomized
    k = tempoDiEsecuzioneQuickSelectRandom[0] / dimensione[0]
    tempoLineare = [(k * x) for x in dimensione]
    ax.plot(dimensione, tempoLineare, '--', color='red', label = 'Andamento Lineare', zorder=3)

    #Disegna la funzione lineare di riferimento per heapSelect
    k = tempoDiEsecuzioneHeapSelect[0] / dimensione[0]
    tempoLineare = [(k * x) for x in dimensione]
    ax.plot(dimensione, tempoLineare, '--', color='darkturquoise', label = 'Andamento Lineare', zorder=3)

    #Disegna la funzione lineare di riferimento per medians
    k = tempoDiEsecuzioneMedianOfMedians[0] / dimensione[0]
    tempoLineare = [(k * x) for x in dimensione]
    ax.plot(dimensione, tempoLineare, '--', color='blueviolet', label = 'Andamento Lineare', zorder=3)

    #Imposta la scala logaritmica su entrambi gli assi

```

```
plt.xscale('log')
plt.yscale('log')

#Imposta la legenda
ax.legend(loc='upper left', fontsize=10)

#Imposta le etichette sugli assi x,y
ax.set_xlabel('Dimensione degli array', fontsize=11, fontweight = 'bold')
ax.set_ylabel('Tempo medio di esecuzione (s)', fontsize=11, fontweight = 'bold')

#Mostra il grafico
plt.show()
```