
Università di Udine

Dipartimento di Scienze Matematiche, Informatiche e Fisiche
Corso di Laurea in Informatica

Anno Accademico 2023/2024

# PROGETTO DI ALGORITMI E STRUTTURE DATI


- Alessandro Roncadin 161904@spes.uniud.it
- Alex Schiavoni 162475@spes.uniud.it
- Francesco De Matteis 163439@spes.uniud.it
- Matteo Pavan 163853@spes.uniud.it

# Sommario

# Introduzione

Il progetto di laboratorio di Algoritmi e Strutture Dati richiede di implementare un programma per il calcolo dei tempi medi di esecuzione di tre algoritmi di ordinamento in funzione della dimensione dell'input.

I tre algoritmi analizzati sono:

- **Quick Select**,
- **Heap Select**
- **Median of Medians Select**.

Per raggiungere questo scopo, gli algoritmi sono stati implementati e testati su **array di diverse dimensioni**. I risultati dei test sono stati rappresentati graficamente sia su scala lineare che log-log per evidenziare le differenze di complessità temporale e di efficienza tra gli algoritmi.

Per ciascun algoritmo, sono riportati:

- funzionamento
- complessità temporale
- pseudocodice
- analisi dei test

Il linguaggio di programmazione usato per implementare gli algoritmi e il programma per il testing è **Python 3**.

# Quick Select

QuickSelect è un algoritmo utilizzato per trovare il k-esimo elemento più piccolo in un array. Si basa sull'algoritmo di ordinamento QuickSort, ma invece di ordinare completamente l'array, si concentra solo sulla posizione del k-esimo elemento.

## Passaggi dell'algoritmo:

1. **Partizionamento**: L'algoritmo sceglie un pivot e partiziona l'array in due sotto-array, uno contenente elementi minori o uguali al pivot e l'altro contenente elementi maggiori.
2. **Ricorsione**: QuickSelect ricorsivamente cerca il k-esimo elemento solo nella partizione che contiene il k-esimo elemento.
3. **Terminazione**: L'algoritmo termina quando il pivot è il k-esimo elemento dell'array.

Il caso peggiore si verifica quando l'array è già ordinato (in ordine crescente o decrescente). In questo caso, il pivot sarà sempre l'elemento massimo, e QuickSelect dovrà esaminare tutti gli elementi dell'array ad ogni ricorsione.

## Complessità:

- Caso medio: $O(n)$
- Caso peggiore: $\theta(n^2)$

## Pseudocodice

**Algorithm 1** QuickSelect

```
1: function QuickSelect(a, h, i, j)
2:     if h < 0 or h ≥ len(a) then
3:         return NIL
4:     end if
5:     if j == NIL then
6:         j ← len(a)
7:     end if
8:     if i ≥ j then
9:         return NIL
10:    end if
11:    k ← partition(a, i, j)
12:    nLeft ← k − i + 1
13:    if h == (nLeft − 1) then
14:        return a[k]
15:    else if h < nLeft then
16:        return QuickSelect(a, h, i, k)
17:    else
18:        return QuickSelect(a, h - nLeft, k + 1, j)
19:    end if
20: end function
```

## Parametri:

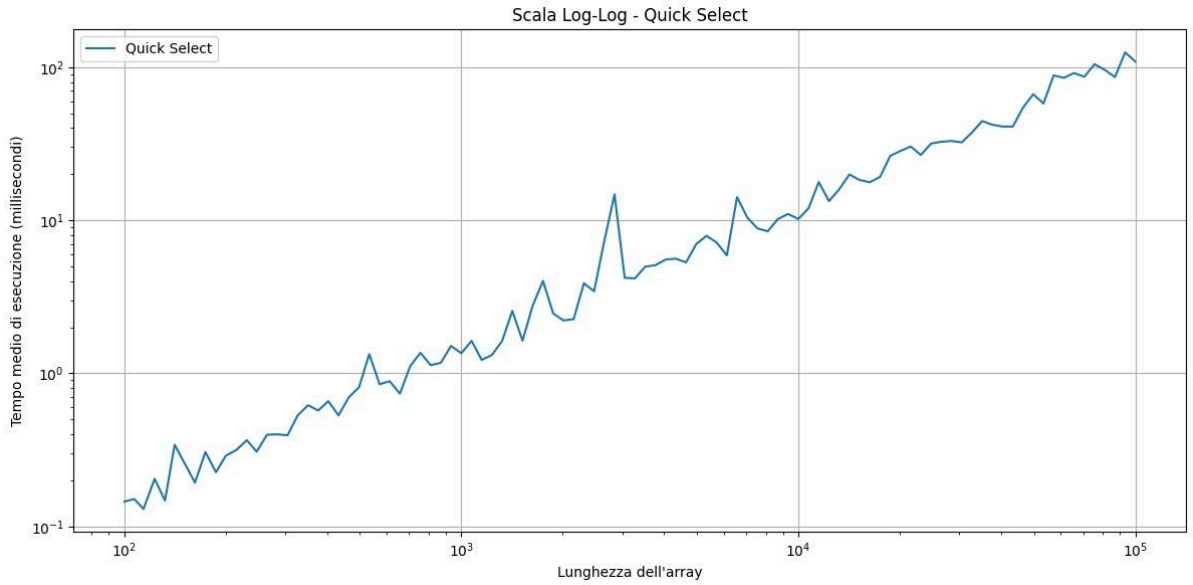
- **a:** array di interi
- **h:** indice dell'elemento k-esimo più piccolo
- **i:** indice di inizio dell'intervalo dell'array su cui lavorare
- **j:** indice di fine dell'intervalo dell'array su cui lavorare

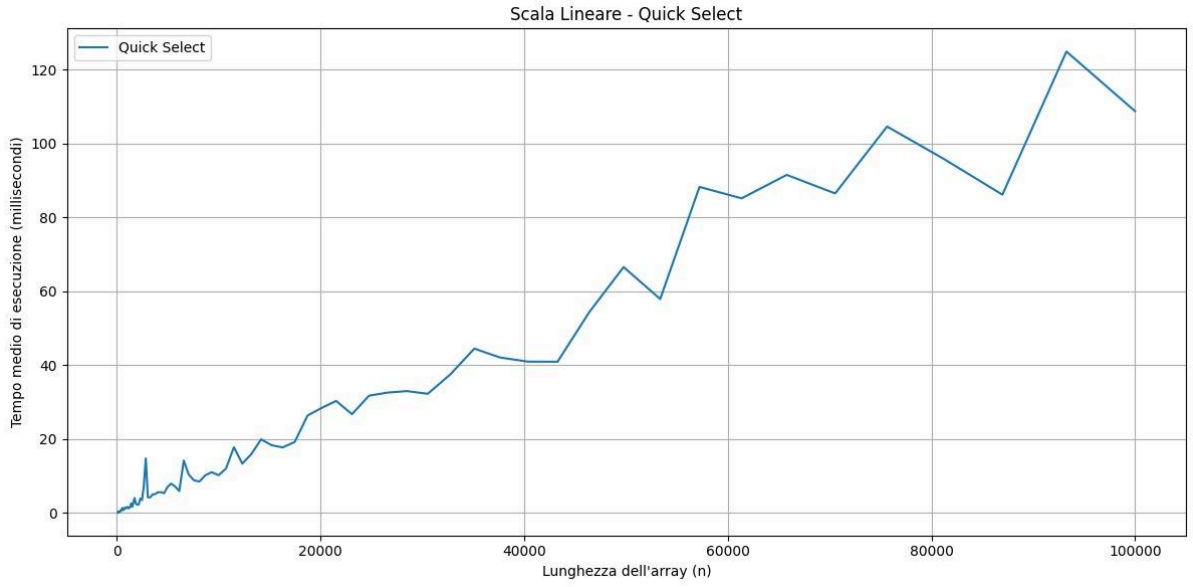
## Analisi

Il grafico mostra un aumento lineare dei tempi di esecuzione medi con l'aumento della lunghezza dell'array. Questo incremento è atteso poiché l'algoritmo deve processare un numero maggiore di elementi con l'aumento della dimensione dell'array.

Il grafico su scala log-log mostra una linea retta, confermando che l'aumento dei tempi di esecuzione è proporzionale a una potenza fissa della lunghezza dell'array indicando una crescita polinomiale della complessità temporale dell'algoritmo.

Si osservano fluttuazioni significative nei tempi di esecuzione per diverse dimensioni dell'array. Queste fluttuazioni possono essere attribuite alla natura dell'algoritmo QuickSelect, che è influenzato dalla scelta del pivot, il che può causare variazioni significative nelle prestazioni.



# Median of Medians Select

L'algoritmo Median of Medians Select è un algoritmo di selezione per trovare il k-esimo elemento più piccolo in un array. Questo algoritmo si basa sulla suddivisione dell'array in blocchi di dimensione limitata e sul calcolo della mediana delle mediane. La versione implementata è "quasi in place", il che significa che l'algoritmo riutilizza lo spazio allocato per l'array originariamente fornito in input, con l'unico spazio aggiuntivo utilizzato dalla pila delle chiamate ricorsive.

## Passaggi dell'algoritmo:

1. **Divisione in blocchi**: L'array viene diviso in blocchi di 5 elementi ciascuno, escluso eventualmente l'ultimo blocco che può contenere meno di 5 elementi.
2. **Calcolo delle mediane dei blocchi**: Ogni blocco viene ordinato e la mediana del blocco viene calcolata. Le mediane di tutti i blocchi vengono raccolte in un nuovo array.
3. **Mediana delle mediane**: La mediana delle mediane dei blocchi viene calcolata ricorsivamente utilizzando lo stesso algoritmo. Questa mediana funge da pivot per il successivo partizionamento.
4. **Partizionamento**: L'intero array viene partizionato attorno alla mediana delle mediane utilizzando una variante della procedura "partition" dell'algoritmo QuickSort.
5. **Chiamata ricorsiva**: In base al valore di k fornito in input, l'algoritmo procede ricorsivamente nella parte di array che sta a sinistra o a destra della mediana delle mediane.

## Complessità:

- Caso medio: $O(n)$
- Caso peggiore: $\theta(n)$

## Pseudocodice

**Algorithm 2** Median of Medians Select

```
1: function MEDIANOFMEDIANS(arr, left, right, k)
2:     if left == right then
3:         return arr[left]
4:     end if
5:     num_elements ← right − left + 1
6:     medians ← [ ]
7:     for i ← 0 to ⌊num_elements/5⌋ − 1 do
8:         sub_list ← sorted(arr[left + i × 5 to left + (i + 1) × 5])
9:         medians.append(sub_list[2])
10:    end for
11:    if num_elements mod 5 ≠ 0 then
12:        sub_list ← sorted(arr[left + ⌊num_elements/5⌋ × 5 to right + 1])
13:        medians.append(sub_list[⌊len(sub_list)/2⌋])
14:    end if
15:    if len(medians) ≤ 5 then
16:        sorted_medians ← sorted(medians)
17:        pivot ← sorted_medians[⌊len(medians)/2⌋]
18:    else
19:        pivot ← MedianOfMedians(medians, 0, len(medians) −
       1, ⌊len(medians)/2⌋)
20:    end if
21:    partition_index ← partition(arr, left, right, pivot)
22:    if k == partition_index then
23:        return arr[k]
24:    else if k < partition_index then
25:        return MEDIANOFMEDIANS(arr, left, partition_index - 1, k)
26:    else
27:        return MEDIANOFMEDIANS(arr, partition_index + 1, right, k)
28:    end if
29: end function
```

## Parametri:

- **arr:** array di interi
- **left:** indice di inizio dell'intervallo dell'array su cui lavorare
- **right:** indice di fine (inclusivo) dell'intervallo dell'array su cui lavorare
- **k:** posizione del k-esimo elemento più piccolo da trovare

**Nota aggiuntiva:** sorted è un algoritmo di ordinamento a scelta

**Algorithm 4** Partition

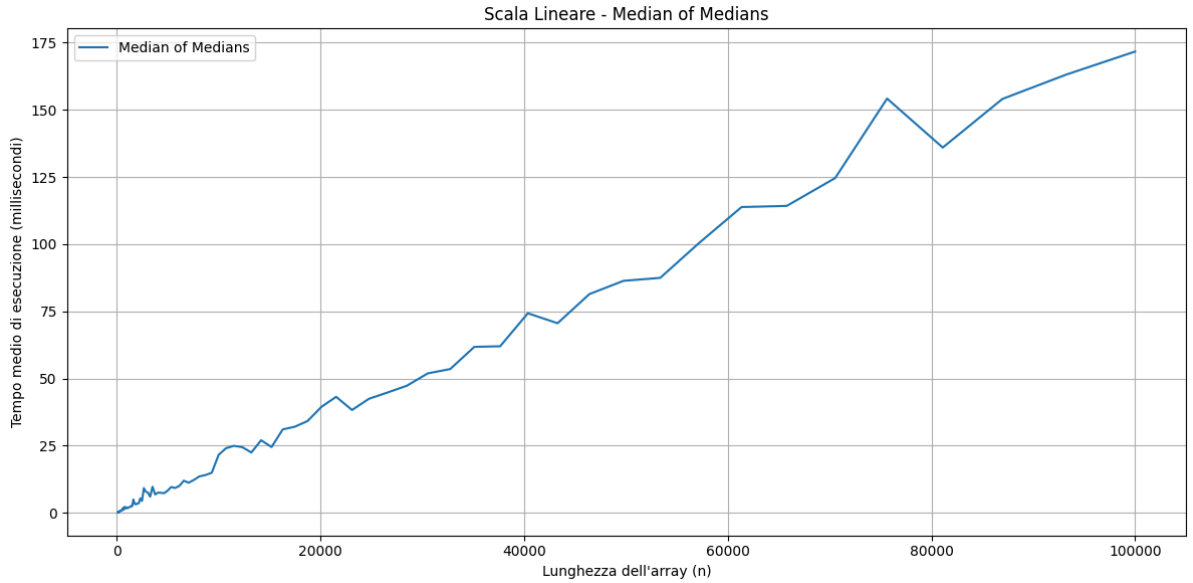
```
1: function PARTITION(array, left, right, pivot)
2:     pivot_index ← array.indexOf(pivot)
3:     swap(array[pivot_index], array[right])
4:     store_index ← left
5:     for i ← left to right − 1 do
6:         if array[i] < pivot then
7:             swap(array[i], array[store_index])
8:             store_index ← store_index + 1
9:         end if
10:    end for
11:    swap(array[store_index], array[right])
12:    return store_index
13: end function
```

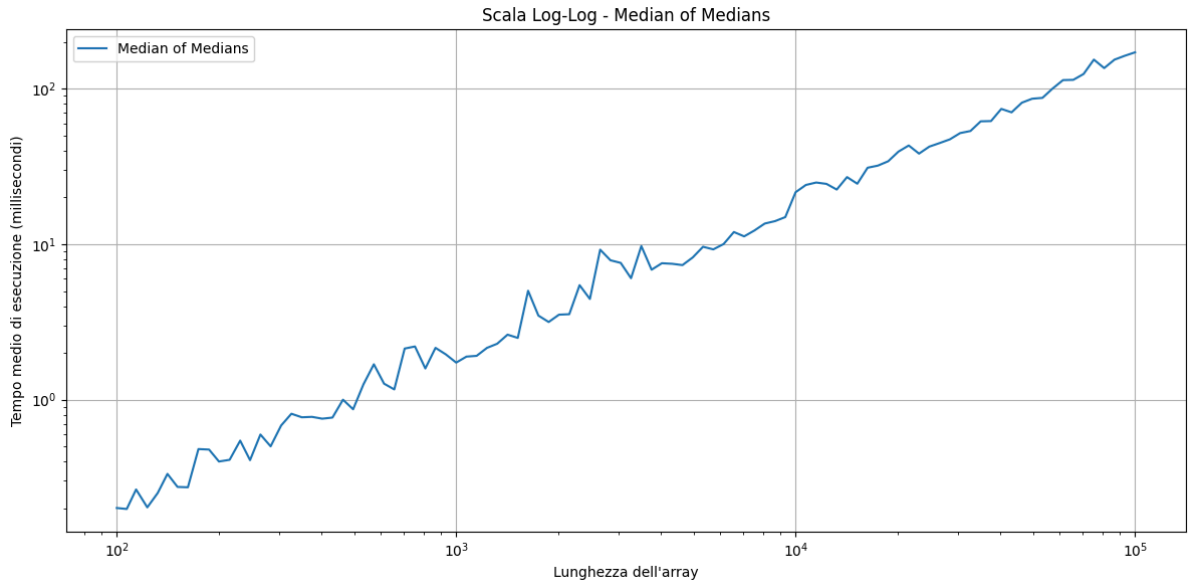
## Analisi

Il grafico mostra un aumento lineare dei tempi di esecuzione medi con l'aumento della lunghezza dell'array. Questo incremento è atteso poiché l'algoritmo deve processare un numero maggiore di elementi con l'aumento della dimensione dell'array.

Il grafico su scala log-log mostra una linea retta, confermando che l'aumento dei tempi di esecuzione è proporzionale a una potenza fissa della lunghezza dell'array indicando una crescita polinomiale della complessità temporale dell'algoritmo.

Le fluttuazioni sono poco marcate e questo potrebbe indicare una maggiore stabilità in termini di prestazioni rispetto alle variazioni nei dati di input.



# Heap Select

L'algoritmo Heap Select è progettato per trovare il k-esimo elemento più piccolo in un array utilizzando due min-heap, denominate H1 e H2. La prima heap, H1, viene costruita a partire dall'array fornito in input e non viene modificata durante l'esecuzione dell'algoritmo. La seconda heap, H2, viene utilizzata per gestire le iterazioni necessarie per selezionare il k-esimo elemento più piccolo.

## Passaggi dell'algoritmo:

1. **Costruzione di H1**:
    - L'heap H1 viene costruita a partire dall'array di input in tempo lineare $O(n)$. Questa heap rappresenta una struttura che permette di accedere rapidamente ai successori di un nodo estratto.
2. **Inizializzazione di H2**:
    - La heap H2 viene inizialmente creata con un solo nodo, che è la radice di H1. Questo nodo rappresenta il minimo elemento di H1.
3. **Iterazioni per trovare il k-esimo elemento**:
    - L'algoritmo esegue $k$ - 1 iterazioni, in cui:
        - Estrae la radice di H2, che è il minimo attuale.
        - Reinserisce in H2 i nodi successori (figli sinistro e destro) del nodo estratto dalla heap H1.
    - Ogni estrazione e reinserimento in H2 avviene in $O(logk)$ tempo.
4. **Restituzione del risultato**:
    - Dopo $k$ - 1 iterazioni, la radice di H2 sarà il k-esimo elemento più piccolo dell'array originale.

## Complessità:

- Caso medio: $O(n + klogk)$
- Caso peggiore: $O(n + klogk)$

## Pseudocodice

**Algorithm 3** HeapSelect

```
1: function HeapSelect(array, k)
2:     H1 ← MinHeap()
3:     H1.buildHeap(array)
4:     H2 ← MinHeap()
5:     H2.insert((H1.getMin(), 0))
6:     i ← 0
7:     while i < k do
8:         element, index ← H2.extract()
9:         left_child_index ← 2 × index + 1
10:        right_child_index ← 2 × index + 2
11:        if left_child_index < H1.length() then
12:            H2.insert((H1.heap[left_child_index], left_child_index))
13:        end if
14:        if right_child_index < H1.length() then
15:            H2.insert((H1.heap[right_child_index], right_child_index))
16:        end if
17:        i ← i + 1
18:    end while
19:    return element
20: end function
```

## Parametri:

- **array**: array contenente numeri interi
- **k**: posizione del k-esimo elemento più piccolo nell'array

**Nota aggiuntiva:**

Nel programma del benchmark è stata implementata una versione ottimizzata che fa uso delle funzioni di gestione delle heap fornite dalla libreria standard di Python, **heapq**, per garantire una maggiore efficienza nei tempi di esecuzione.

Nella consegna di Heap Select sono stati caricati due file contenenti entrambi i codici di Heap Select:

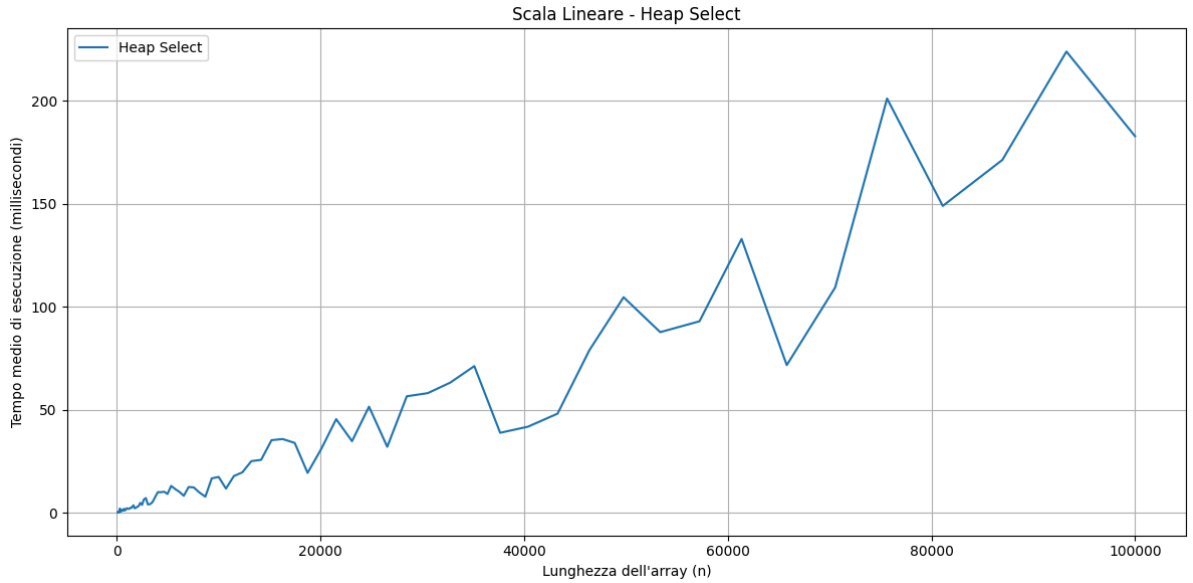
- HeapSelect.py usa la classe MinHeap() vista a lezione
- HeapSelectOptimized.py usa la libreria heapq

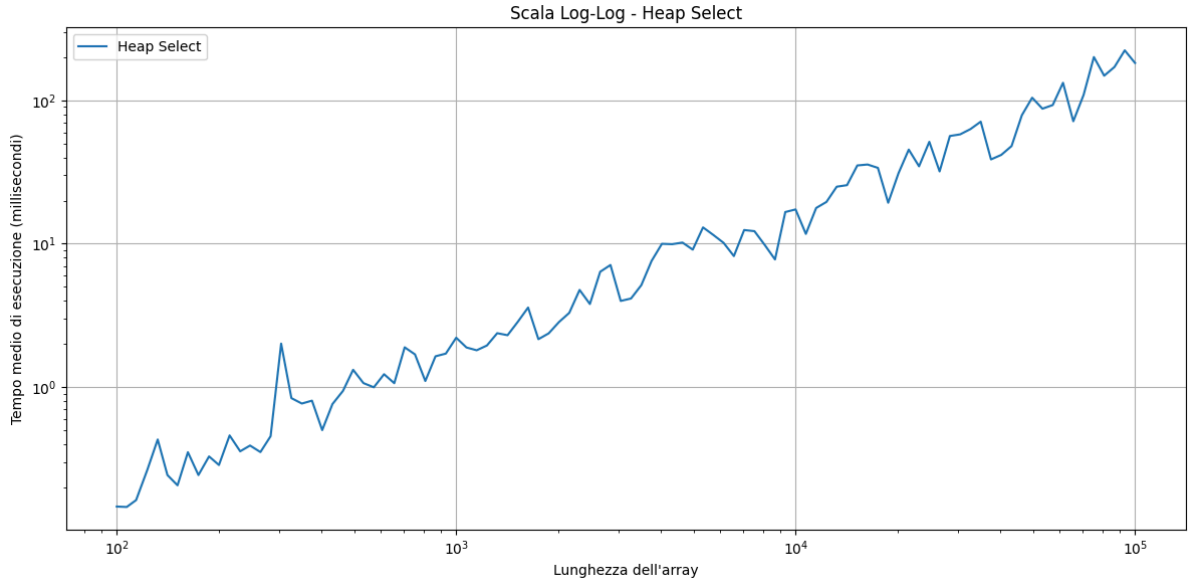
## Analisi

Il grafico mostra un aumento generale dei tempi di esecuzione medi man mano che la lunghezza dell'array (n) aumenta. Questo è previsto poiché, con l'aumento della dimensione dell'array, l'algoritmo deve processare più elementi, richiedendo più tempo.

Il grafico su scala log-log mostra una linea retta, confermando che l'aumento dei tempi di esecuzione è proporzionale a una potenza fissa della lunghezza dell'array indicando una crescita polinomiale della complessità temporale dell'algoritmo.

Si possono osservare delle fluttuazioni nei tempi di esecuzione per diverse dimensioni dell'array e possono essere dovute alla variabilità nei dati casuali generati.



# Conclusioni

Tutti e tre gli algoritmi mostrano un aumento dei tempi di esecuzione medi con l'aumento della lunghezza dell'array poiché devono processare più elementi.

Sia su scala lineare che su scala log-log, Quick Select tende a essere il più efficiente dei tre, mostrando tempi di esecuzione medi generalmente inferiori rispetto a Median of Medians Select e Heap Select.

Median of Medians Select mostra una crescita costante e lineare dei tempi di esecuzione, indicando una maggiore stabilità e prevedibilità nelle prestazioni.

Heap Select mostra fluttuazioni più evidenti nei tempi di esecuzione, ma generalmente segue un trend simile a Median of Medians Select.