MINISTRY OF EDUCATION
IMAM ABDULRAHMAN BIN
FAISAL UNIVERSITY
COLLEGE OF COMPUTER SCIENCE
& INFORMATION TECHNOLOGY

وزارة التعليم
جامعة الإمام
عبدالرحمن بن فيصل
كلية علوم الحاسب
وتقنية المعلومات

جامعة الإمام عبدالرحمن بن فيصل
IMAM ABDULRAHMAN BIN FAISAL UNIVERSITY

# Finding the Largest $K_{th}$ Element

## CS412 | Algorithms Analysis and Design
## Academic Year 2023/2024 – First Semester
## Group 1 – Team 6

---

## Instructor: Dr. Atta-ur-Rahman

# Abstract

This project explores the efficiency and practicality of three distinct sorting algorithms – heapsort, selection sort, and quicksort – in the context of finding the kth largest element in an array (KTH Largest Element of Array, 2022b). Our analysis is rooted in a comparative study that encompasses both theoretical and empirical dimensions, providing insights into the performance characteristics of each algorithm. We delve into the intricacies of each algorithm, examining their time and space complexities, and highlighting their unique advantages and disadvantages in different scenarios. Implementations in a chosen programming language demonstrate the practical application of these algorithms in solving the stated problem. A series of experiments with arrays of varying sizes and compositions are conducted to empirically evaluate the performance, particularly focusing on the running time of each algorithm. The results are visually represented through running time diagrams, offering an intuitive comparison of efficiency across different cases. Our findings aim to provide a comprehensive guide for selecting the most appropriate algorithm based on specific problem constraints and requirements, thereby contributing to a nuanced understanding of algorithm efficiency in practical applications (Ekwerike, 2021).

# Table of Contents

# Table of Tables

# Table of Figures

# Introduction

In the realm of computer science, the efficient processing and retrieval of data are paramount. Among the myriad of challenges, finding the kth largest element in an array stands out as a task that is both fundamentally important and computationally interesting. This project zeroes in on this problem, utilizing three well-known sorting algorithms as our tools of investigation: heapsort, selection sort, and quicksort.

Each of these algorithms brings its unique approach to sorting, thereby influencing its effectiveness in locating the kth largest element. Heapsort leverages the properties of a binary heap to achieve efficient sorting, while selection sort simplifies the process through repeated selection of the largest (or smallest) element. Quicksort, on the other hand, adopts a divide-and-conquer strategy, partitioning the array into segments for more efficient sorting.

Our objective is multifaceted: firstly, to implement these algorithms in a consistent computational environment and apply them to the specific task of finding the kth largest element in an array. Secondly, to delve into a comparative analysis of these algorithms, assessing them in terms of time complexity, space complexity, and practical performance under varying data conditions. Through this study, we aim to elucidate the nuances of each algorithm's performance, particularly highlighting the scenarios in which one algorithm may outshine the others.

The practical implications of this study are significant. By understanding the strengths and limitations of these sorting algorithms in a specific context, we can provide valuable insights for application developers and computer scientists in selecting the most appropriate algorithm for their specific needs, ensuring both efficiency and effectiveness in data processing tasks (Ekwerike, 2021).

# Quick Sort for Finding the kth Largest Element

Quicksort is a fast-sorting algorithm that separates an expansive data array into smaller segments. The large array is divided into two smaller in-size arrays, one holding values less than the specified value, say pivot, and the other containing values greater than the provided pivot value (Huang et al., 2014). Quicksort, as a sorting algorithm, doesn't inherently aim to find the kth largest element directly. However, you can modify the standard Quicksort algorithm slightly to achieve this goal efficiently, this modified algorithm is called Quick Select. Its more efficient and specifically designed to find the kth largest element in an unsorted array. It works similar to the quick sort by partitioning the array.

## How Quick Select Works in This Context:

Quick select is a method for finding the kth smallest or biggest element in an unordered list. It is a quicksort variant that operates by picking a pivot element and splitting the array around it, with elements smaller than the pivot to the left and elements bigger to the right.
The Quickselect algorithm's essential phases for finding the kth greatest element are as follows:

1. **Choose one of the array's pivot elements.** This pivot might be chosen at random or using a strategy (for example, selecting the first or final piece).
2. **Divide the array around the pivot point.** Rearrange the pieces such that everything smaller than the pivot is on the left and everything bigger than the pivot is on the right. If the array was sorted, the pivot will be at its sorted position after this step.
3. **Examine the pivot's location.** Return the pivot element if the pivot is at index k (i.e., the kth biggest element).
4. If the pivot is at a location less than k, the kth biggest element in the right subarray must be present. **Apply the method to the appropriate subarray recursively.**

5. If the pivot is at a position greater than k, the kth biggest element in the left subarray must be present. **Apply the procedure recursively to the left subarray, modifying the k value as necessary.**
6. **Steps 1–5 must be repeated** until the kth biggest element is discovered.

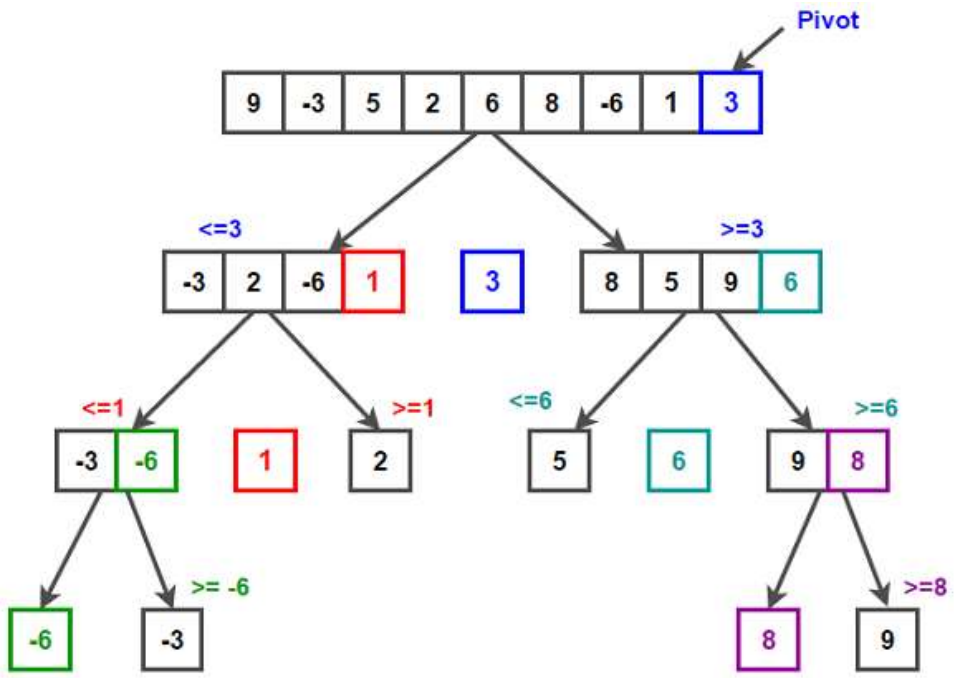



*Figure 1: Quick Sort Process*

## Cases of Quick Select

- **Best Case Scenario of Quick Select:**

Quickselect performs best when the pivot used in each recursive step consistently splits the array into almost equal halves. As a result of this circumstance, the method swiftly converges on the kth greatest element.

In the best-case scenario:

- Each step's pivot reliably splits the array into two almost equal subarrays.
- This balanced partitioning takes place at every phase of the process.

- As a result, the algorithm's performance approaches linear time complexity, precisely O(n), where 'n' is the number of array entries.

- **Average Case Scenario of Quick select:**

Quick select's average-case scenario for identifying the kth biggest element makes assumptions about the distribution of the input items and the unpredictability of pivot selection.

In the average case:

- The method chooses pivots at random or with an approach that approximates randomness (such as the median-of-three).
- On average, the chosen pivot divides the array into two unequal-sized divisions.
- The average size of the partitions is proportional to the array size.
- Each recursive step decreases the issue size by a factor near to (but not precisely) 1/2, resulting in an average time complexity of O(n), where 'n' is the number of array members.

- **Worst Case Scenario of Quick select:**

The worst-case situation for Quick select in identifying the kth biggest element is when the chosen pivot in each step regularly results in imbalanced partitions.

In the worst-case scenario:

- The pivot used in each step does not split the array correctly.
- The selected pivot results in significantly uneven divisions rather than splitting the array into two approximately equal partitions.
- As a result, the partition sizes become significantly skewed at each stage of the process, resulting in a minimum reduction of the issue size.
- The approach eventually makes only minimal progress toward locating the kth greatest element, thereby lowering the issue size by one element at a time.
- Quick select's worst-case time complexity approaches O($n^2$), where 'n' indicates the number of array entries.

## Examples of Finding the kth element using Quick Select

**Finding the 500th largest element in a random array with 3000 elements took 219.3 ms**

```
The 500th largest element is 6322
The 500th element has been found in 219.28048133850098 milliseconds
```

*Figure 2: Quick Select Random Example*

**Finding the 500th largest element in a sorted array with 3000 elements took 173.2 ms**

```
The 500th largest element is 6903
The 500th element has been found in 173.1560230255127 milliseconds
```

*Figure 3: Quick Select Sorted Example*

**Finding the 500th largest element in a reverse sorted array with 3000 elements took 204.6 ms**

```
The 500th largest element is 6691
The 500th element has been found in 204.60081100463867 milliseconds
```

*Figure 4: Quick Select Reversed Example*

# Heapsort for Finding the kth Largest Element

Heapsort is a comparison-based sorting algorithm that leverages the properties of a binary heap, a complete binary tree where each parent node is greater (in a max heap) or smaller (in a min heap) than its children (Sharma et al., 2009). This characteristic makes heapsort particularly effective for finding the kth largest element in an array (Alake, 2023).

### How Heapsort Works in This Context:

1. **Building the Heap**: The first step in heapsort involves building a max heap from the input array. In a max heap, the largest element is always at the root. This process rearranges the elements so that each parent node is greater than its children.
2. **Extracting Elements**: Once the max heap is constructed, the algorithm repeatedly removes the root (the largest element) and reheapifies the heap. This removal happens exactly $k$ times to get to the kth largest element. In each iteration, the root is swapped with the last element of the heap, reducing the heap's size, and ensuring the next largest element moves to the root.
3. **Finding the kth Largest Element**: After repeating this process k-1 times, the root of the heap is the kth largest element. This element is then extracted, providing the solution to our problem.

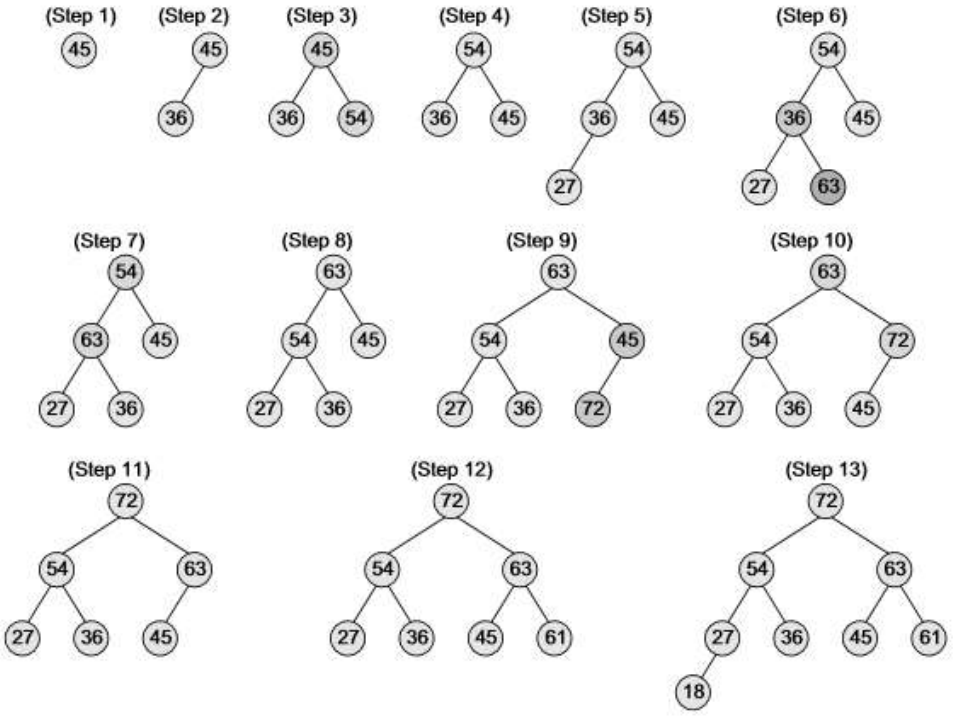



*Figure 5: Heap Sort Process*

## Cases of Heap Sort

- **Best Case Scenario of Heapsort:**

The best-case scenario for heapsort in finding the kth largest element in an array is largely consistent regardless of the initial data arrangement, as the algorithm always constructs a heap first. Efficiency improves notably when k is very small, especially for k = 1, which minimizes the number of heap restructuring needed. In such cases, the largest element is immediately available at the root after building the heap, reducing further heapify operations. The time complexity in the best case remains O(n log n) for sorting the entire array, but actual operations are reduced for smaller k values.

- **Average Case Scenario of Heapsort:**

In the average case, heapsort deals with an array of randomly ordered elements. Here, the performance is fairly stable, with the complexity averaging at O(n log n). The value of k significantly influences the practical efficiency; a moderate k value implies a balanced heapify operation count to extract the kth largest element. This represents the typical behavior of heapsort where the initial array order doesn't substantially affect the overall time complexity.

- **Worst Case Scenario of Heapsort:**

The worst-case scenario for heapsort in the context of finding the kth largest element is characterized by a large k value, particularly when k is close to the array size. Here, the algorithm nearly sorts the entire array to find the kth largest element. Nonetheless, the worst-case time complexity stays at O(n log n), consistent with the best and average cases. The "worst-case" aspect relates more to the extent of operations needed to extract and heapify elements until reaching the desired kth position, rather than a decrease in theoretical time complexity.

## Examples of Finding the kth element using Heapsort

**Finding the 500th largest element in a random array with 3000 elements took 29 ms**

```
The 500th largest element is 6798
The 500th element have been fount in 29.056072235107422
```

*Figure 6: Heapsort Random Example*

**Finding the 500th largest element in a sorted array with 3000 elements took 17.9 ms**

```
The 500th largest element is 6577
The 500th element have been fount in 17.943382263183594
```

*Figure 7: Heapsort Sorted Example*

**Finding the 500th largest element in a reversed array with 3000 elements took 16.2 ms**

```
The 500th largest element is 6764
The 500th element have been fount in 16.225576400756836
```

*Figure 8: Heapsort Reversed Example*

# Selection sort for Finding the kth Largest Element

Selection sort is a simple comparison-based sorting algorithm that works by dividing the array into two parts: a sorted part and an unsorted part. The algorithm chooses the least element from the unsorted part and replaces it with the first element in the unsorted part for each iteration. This algorithm has an $O(n^2)$ time complexity, where n is the number of elements in the array (Levitin, 2012).

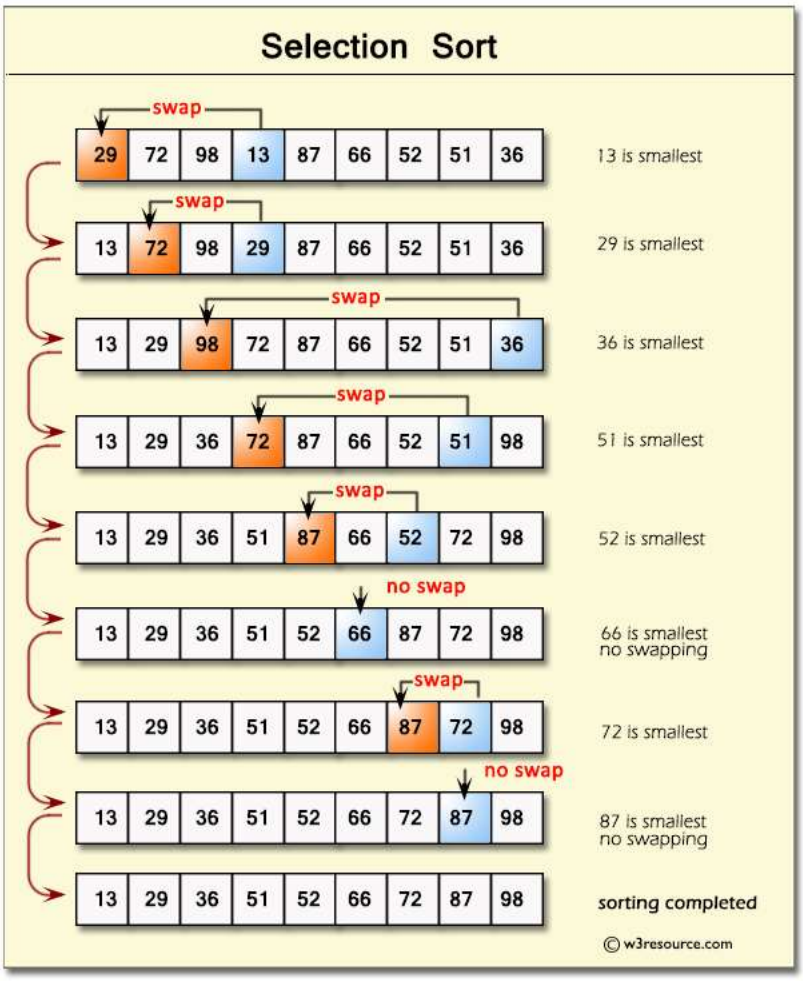



*Figure 9: Selection Sort Process*

Finding the kth largest element in an array using selection sort is not an efficient algorithm by itself, but you can modify the algorithm to terminate after k iterations. By doing so, you can locate the kth largest element without having to completely sort the entire array.

**How Selection Sort Works in This Context**:

1. **Initialize**:
    - Starting with the entire array and setting a variable k to the desired position of the kth largest element.
    - Determine the size of the array.
2. **Outer loop**:
    - Iterate k times, where each iteration focuses on the last element in the unsorted part of the array.
    - Start the iteration from the last element in the array and go backwards.
3. **Inner loop**:
    - Within each iteration of the outer loop, the inner loop iterates through the elements in the unsorted part from the beginning up to the current outer loop index.
    - Find the largest element in the unsorted part.
4. **Swapping**: Swap the maximum element found in the inner loop with the current outer loop index; this will ensure that the maximum element is in its correct position in the sorted order.
5. **Result**: The kth largest element will be at its proper location in the array and retrievable after k iterations.

It's crucial to recall that the selection sort algorithm was modified, resulting in an O(k*n) time complexity for the best and average cases, which differ from the original selection sort algorithm. This is an improvement over the initial time complexity, but it's still not the most effective way to determine the kth largest element.

## Cases of Selection Sort

- **Best case scenario of selection sort:**

The best-case scenario of this algorithm (after the modification) for this problem occurs when k is small or equal to 1 and the inner loop runs a minimal number of times. In this case, the inner loop still iterates up to the current index of the outer loop, but because k is small, the time complexity is relatively lower. As a result, the best-case time complexity is still O(k*n), but with a smaller constant factor compared to the other cases.

- **Average case scenario of selection sort:**

The average-case scenario of this algorithm (after the modification) for this problem depends on the distribution of the elements in the array and the value of k. In this case, the algorithm iterates in the inner loop by the order of elements in the array. If the array is partially sorted, the number of swaps required reduced, leading to a lower average-case time complexity. As a result, the time complexity in this case is still O(k*n), where k is the number of iterations and n is the size of the array.

- **Worst case scenario of selection sort:**

The worst-case scenario of this algorithm (after the modification) for this problem occurs when the algorithm (outer loop and inner loop) performs maximum number of k iterations for an array of size n. As a result, the time complexity in this case is $O(n^2)$.

## Examples of Finding the kth element using Selection Sort

**Finding the 500th largest element in a random array with 3000 elements took 38.5 ms**

```
The 500th largest element is 5285
The 500th element have been fount in 38.51318359375
```

*Figure 10: Selection Sort Random Example*

**Finding the 500th largest element in a sorted array with 3000 elements took 36.1 ms**

```
The 500th largest element is 6834
The 500th element have been fount in 36.15903854370117
```

*Figure 11: Selection Sort Sorted Example*

**Finding the 500th largest element in a reversed array with 3000 elements took 42 ms**

```
The 500th largest element is -6836
The 500th element have been fount in 42.01173782348633
```

*Figure 12: Selection Sort Reversed Example*

# Running Time Diagram

*Table 1: Time & Space Compliexity*

| Time Complexity | | | | Space Complexity |
|---|---|---|---|---|
| **Algorithm** | **Best** | **Average** | **Worst** | |
| **Selection** | O(k*n) | O(k*n) | $O(n^2)$ | O(1) |
| **Heap** | O(n log n) | O(n log n) | O(n log n) | O(1) |
| **Quick** | O(n) | O(n) | $O(n^2)$ | O(1) |

*Table 2: Running Time*

| Time (ms) | | | |
|---|---|---|---|
| **Size** | **Heap Sort** | **Selection Sort** | **Quick Select** |
| **N=10** | 0.0420 | 0.0134 | 0.0043 |
| **N=20** | 0.0365 | 0.0329 | 0.012 |
| **N=30** | 0.0609 | 0.1012 | 0.016 |
| **N=40** | 0.0912 | 0.1026 | 0.029 |
| **N=50** | 0.1143 | 0.1507 | 0.035 |
| **N=60** | 0.1354 | 0.1416 | 0.036 |
| **N=70** | 0.1677 | 0.1627 | 0.026 |
| **N=80** | 0.2100 | 0.2009 | 0.028 |
| **N=90** | 0.2359 | 0.2272 | 0.032 |
| **N=100** | 0.3052 | 0.2517 | 0.034 |
| **N=110** | 0.3082 | 0.2923 | 0.036 |
| **N=120** | 0.4100 | 0.3086 | 0.15 |
| **N=130** | 0.6423 | 0.3496 | 0.066 |
| **N=140** | 0.4236 | 0.3778 | 0.077 |
| **N=150** | 0.4486 | 0.4152 | 0.068 |
| **N=160** | 0.5153 | 0.4465 | 0.054 |
| **N=170** | 0.5381 | 0.4859 | 0.053 |
| **N=180** | 0.5799 | 0.519 | 0.068 |
| **N=190** | 0.6314 | 0.6712 | 0.060 |
| **N=200** | 0.6718 | 0.5943 | 0.061 |
| **N=210** | 0.6884 | 0.8759 | 0.058 |
| **N=220** | 0.7489 | 1.2025 | 0.059 |
| **N=230** | 0.7617 | 1.2965 | 0.067 |
| **N=240** | 0.7946 | 1.2569 | 0.067 |
| **N=250** | 0.8448 | 1.5982 | 0.070 |

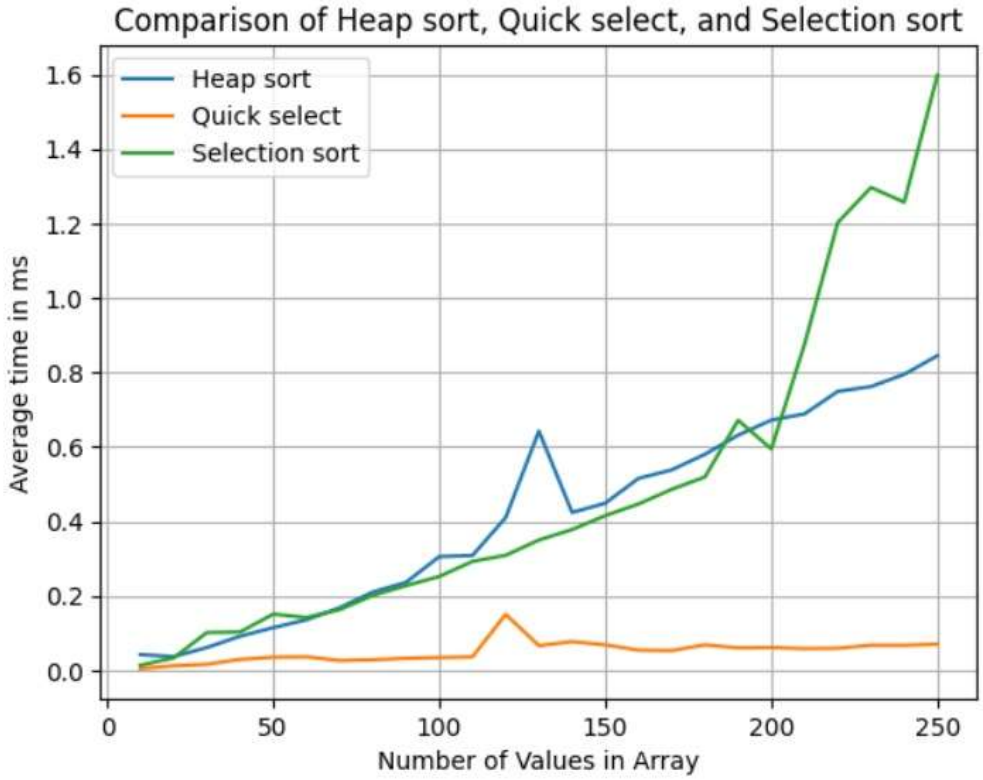
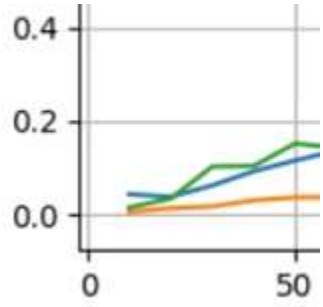


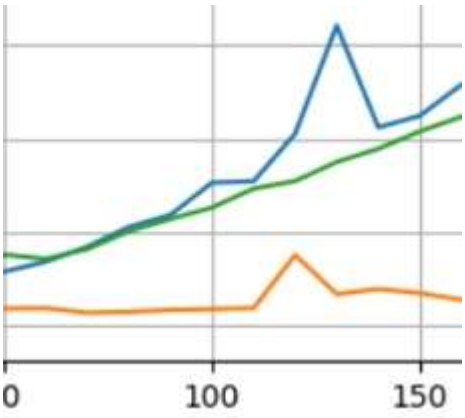

*Figure 13: Heapsort, Selection Sort, and Quicksort Behavior*

## Describe the behavior of the three algorithms:




**Initial Segment (0-50 Values):** In the initial part of the graph, all algorithms start with an almost negligible running time. Quick select initially shows a bit of fluctuation, which may be due to the constant factor overheads that are more pronounced when dealing with small datasets. Heap sort and Selection sort increase gradually, with Selection sort showing a slightly steeper slope as it begins to feel the impact of its quadratic time complexity even at this early stage. The threshold in this segment is not clearly visible, as all algorithms are managing the smaller input sizes with relative ease, but we can start to see the beginnings of a divergence, particularly with Selection sort.

**Middle Segment (50-150 Values):** As we move into the middle segment of the graph, the differences between the algorithms become more pronounced. Quick select maintains a low and steady increase, reflecting its average-case time complexity which is linear relative to the number of elements. Heap sort begins to show its logarithmic growth, remaining efficient but with a noticeable increase in running time as the size of the dataset grows. Selection sort's running time starts to escalate more noticeably compared to the other two algorithms, revealing the less efficient nature of its algorithm for larger datasets.




**Final Segment (150-250 Values):** In the final portion of the plot, Selection sort's running time increases sharply, indicating that it has hit its efficiency threshold and is now significantly outpaced by the other two algorithms. Heap sort also shows an upward trend. Quick select continues to exhibit a gentle slope, confirming its efficiency and stability for this task across all tested sizes of the array. Here, the threshold for selection sort is less distinct but could be inferred around the 200-value mark, where the spike in running time suggests a performance degradation.

# Conclusion

In conclusion, the graph analysis reveals distinct performance profiles for each algorithm when tasked with finding the kth largest element in an array. Heap sort, with its O(n log n) complexity for sorting, shows a steady increase in running time, indicative of logarithmic growth, with a notable spike towards the end that suggests a less predictable performance with larger data sets. Quick select, an algorithm designed for selection with an average-case complexity of O(n), demonstrates the most consistent and efficient performance across all array sizes, with a minimal increase in running time, aligning with its expected efficiency for the task at hand. Selection sort, in contrast, with its $O(n^2)$ complexity, shows a significant increase in running time as the number of elements grows, confirming its impracticality for larger arrays due to its quadratic time growth.

The graph logically underscores the importance of algorithm selection based on the nature of the task. Quick select stands out as the superior choice for selection problems, maintaining low average times even as the array size increases. Heap sort, while generally robust, exhibits signs that it may not be as suited for selection tasks as Quick select. Selection sort's performance trajectory clearly indicates its limitations, becoming increasingly impractical as the dataset grows.

Finally, from the array sizes analyzed, it can be concluded that Quick select should be the preferred method for finding the kth largest element in an array. The data suggests that for all tested sizes, Quick select not only outperforms Selection sort but also Heap sort, providing the most time-efficient solution for the given problem. The analysis serves as a compelling argument for using task-specific algorithms like Quick select over more general sorting algorithms when performance is a critical factor.

# References

Alake, R. (2023, April 4). Heap sort explained. https://builtin.com/data-science/heap-sort

Ekwerike, P. (2021, December 14). Technical Interviews Walkthrough: Find the Kth largest element in a given array. Medium. https://medium.com/@pekwerike/technical- interviews-walkthrough-find-the-kth-largest-element-in-a-given-array-386fa12e7e48

Huang, Q., Liu, X., Sun, X., & Zhang, J. (2014). How to select the top K elements from evolving data? *Algorithms and Computation*, 60–70. https://doi.org/10.1007/978-3-662- 6_48971-0

Implementation of selection sort algorithm in various programming languages. (2021). *International Journal of Advanced Trends in Computer Science and Engineering*, *10*(4), 2249-2255. https://doi.org/10.30534/ijatcse/2021/1071032021

KTH largest element of array. (2022b, January 11). InterviewBit. https://www.interviewbit.com/blog/kth-largest-element-of-array/

Levitin, A. (2012). Introduction to the design & analysis of algorithms. *In Pearson Addison Wesley eBooks*. https://ci.nii.ac.jp/ncid/BB14217613

Sharma, V., Singh, S., & Kahlon, K. S. (2009). Comparative performance study of improved heap sort algorithm on different hardware. *Journal of Computer Science*, *5*(7), 476–478. https://doi.org/10.3844/jcssp.2009.476.478

# Appendix

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

    for j in range(low, high):
        if arr[j] <= pivot:
            i += 1
            arr[i], arr[j] = arr[j], arr[i]

    arr[i + 1], arr[high] = arr[high], arr[i + 1]
    return i + 1


def quick_select(arr, low, high, k):
    if low <= high:
        pivot_idx = partition(arr, low, high)

        if pivot_idx == k:
            return arr[pivot_idx]
        elif pivot_idx < k:
            return quick_select(arr, pivot_idx + 1, high, k)
        else:
            return quick_select(arr, low, pivot_idx - 1, k)


def find_kth_largest(arr, k):
    n = len(arr)
    if k > 0 and k <= n:
        return quick_select(arr, 0, n - 1, n - k)
    else:
        return None
```

*Figure 14: Quicksort Python Code*

```python
def heapsort(arr):
    def heapify(arr, n, i):
        largest = i
        l = 2 * i + 1
        r = 2 * i + 2

        if l < n and arr[i] < arr[l]:
            largest = l

        if r < n and arr[largest] < arr[r]:
            largest = r

        if largest != i:
            arr[i], arr[largest] = arr[largest], arr[i]
            heapify(arr, n, largest)

    n = len(arr)

    for i in range(n // 2 - 1, -1, -1):
        heapify(arr, n, i)

    for i in range(n - 1, 0, -1):
        arr[i], arr[0] = arr[0], arr[i]
        heapify(arr, i, 0)


def findKthLargest(nums, k):
    heapsort(nums)
    return nums[-k]
```

*Figure 15: Heapsort Python Code*

```python
def selection_sort_find_kth_largest(arr, k):
    size = len(arr)

    for i in range(size - 1, size - k, -1):

        max_index = i

        for j in range(i):
            if arr[j] > arr[max_index]:
                max_index = j

        arr[i], arr[max_index] = arr[max_index], arr[i]

    return arr[size - k]
```

*Figure 16: Selection Sort Python Code*