

**MARMARA UNIVERSITY**

**FACULTY OF ENGINEERING DEPARTMENT OF**

**COMPUTER SCIENCE ENGINEERING**

**CSE 2046 – ANALYSIS OF ALGORITHMS**

**COMPARING SORTING ALGORITHMS**

**STUDENTS NUMBERS**

Serkan Koç 150118073

İbrahim Hakkı Candan 150118061

Atila İlhan Yatağan 150118033

**Submitted To:** **Due Date:**

Asst.Prof. Ömer Korçak 11.05.2022

**Purpose of the Project**

The main goal of this project is to design an experiment to compare different algorithms for the selection problem (i.e., finding *k*’ th smallest element in an unsorted list of n numbers.) and to compare our findings with theoretical complexity values.

Algorithms used in the comparisons:

1. Insertion-sort
2. Merge-sort
3. Quick-sort
4. Partial selection-sort
5. Partial heap-sort
6. Quick select algorithm (based on array partitioning, first element as pivot)
7. Quick select algorithm (median-of-three pivot selection)

**Step 1: Designing the Experiment:**

**Deciding on reasonable inputs:**

This part of the report only explains the input selections for each algorithm. Extensive explanations for each algorithm is provided individually.

Different input sizes for algorithms are selected as 100,500,1000,10000. The purpose of selecting such input sizes is to make the comparisons (the empirical results of the experiments) easier.

**1) Insertion-sort inputs**

Insertion sort is a decrease and conquer algorithm and a simple sorting algorithm that is based on splitting an array into two parts as sorted and unsorted.

* The best-case input is an array that is already sorted. In this case insertion sort has a linear running time (i.e., O(*n*))
* The simplest worst-case input is an array sorted in reverse order. The set of all worst-case inputs consists of all arrays where each element is the smallest or second smallest of the elements before it. This gives insertion sort a quadratic running time (i.e., O(*n*2))
* The average case is also quadratic, which makes insertion sort impractical for sorting large arrays. Average case input is an array that is not sorted. (random elements)

1. **Merge-sort inputs**

Merge sort is a divide and conquer algorithm. It divides the input array into two halves, calls itself for the two halves, and then merges the two sorted halves. The time complexity of all cases is O(nlogn).

* Best case input is a sorted array in ascending order algorithm to reduce the comparisons between the elements.
* Worst case input is a permutation that parses the array into n parts and reorganizes according to the algorithm. We took inputs that have 2n elements.
* Average case, random inputs that generated with varied sizes.

1. **Quick-sort inputs**

Quick Sort algorithm depends on choosing a pivot among items and finding its correct place in the array. After that array is divided into two parts from the point that pivot exists at, and the process keeps go on recursively until correct places of all items are found. Thereby array becomes sorted.

* Best case input choosing first element pivot emerges when correct position of the pivot divides array into two equal parts. And this can be possible when array is unsorted and items randomly distributed.
* Worst case input of choosing the first element as pivot emerges when the array is already sorted ascending or descending order. Since every item is already located at their correct position, when partitioning process ended array is divided into very unbalanced parts.
* Average case, random inputs that generated with varied sizes.

1. **Partial selection-sort inputs**

The concept used in Selection Sort helps us to partially sort the array up to kth smallest (or largest) element for finding the kth smallest (or largest) element in an array.

* The best case is a sorted array in ascending order and k equals 1. The time complexity of the best case is O(n).
* The worst case is a reversed sorted array in descending order and k equals n for partial selection sort. The time complexity of the worst case is O(n^2).
* Average case, random inputs that generated with varied sizes. Time complexity of the average case is O(nk)

1. **Partial heap-sort inputs**

Heap sort is a comparison-based sorting technique based on Binary Heap data structure. It is like selection sort where we first find the minimum element and place the minimum element at the beginning. We repeat the same process for the remaining elements.

* Best case input for heap sort is when all the elements are equal. In this case, no max heapify needs to be performed.
* Worst case input for heap sort might happen when all elements in the list are distinct or reversed listed (ascending order for min heap, descending order for max heap. Therefore, we would need to call max heapify every time we remove an element.
* Average case, random inputs that generated with varied sizes.

1. **Quick select algorithm inputs (based on array partitioning, first element as pivot)**

The algorithm is like Quicksort. The difference is, instead of recurring for both sides (after finding pivot), it recurs only for the part that contains the k-th smallest element.

* Best case occurs when we partition the list into two halves and continue with only the half, we are interested in.
* Worst case occurs when we pick the largest/smallest element as pivot.
* The time complexity for the average case for quick select is O(n) (reduced from O(nlogn) — quick sort). The worst-case time complexity is still O(n²) but by using a random pivot, the worst case can be avoided in most cases. So, on an average quick select provides a O(n) solution to find the kth largest/smallest element in an unsorted list.

Quick select algorithm (median-of-three pivot selection) inputu yapılacak

**Inputs burada bitecek !!**

**Insertion Sort**

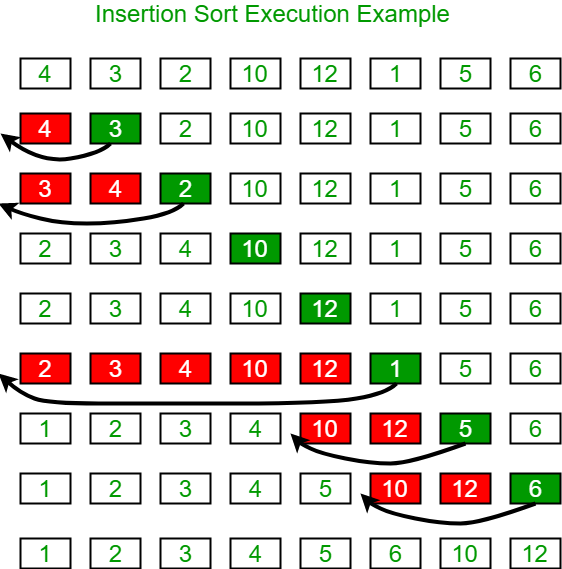
Insertion sort is a simple [sorting algorithm](https://en.wikipedia.org/wiki/Sorting_algorithm) that builds the final [sorted array](https://en.wikipedia.org/wiki/Sorted_array) (or list) one item at a time. It is much less efficient on large lists than more advanced algorithms such as [quicksort](https://en.wikipedia.org/wiki/Quicksort), [heapsort](https://en.wikipedia.org/wiki/Heapsort), or [merge sort](https://en.wikipedia.org/wiki/Merge_sort).

**Insertion Sort Algorithm**

1. Set a marker for sorted section after the first element
2. Repeat the following until unsorted section is empty

* Select the first unsorted element
* Swap other elements to the right to create the correct position and shift the unsorted element
* Advance the marker to the next element

**Insertion Sort Visualization**

****

1) **for** *j <-* 2 **to** *length*[*A*]

2) **do** *key <- A*[*j*]

3) Insert *A*[*j*] into the sorted sequence *A*[1 . . *j* - 1].

4) *i* <- *j* - 1

5) **while** *i >* 0 and *A*[*i*]> *key*

*6*)***do*** *A*[*i* + 1] <- *A*[*i*]

7) *i <- i* - 1

8) *A*[*i* + 1] <- *key*

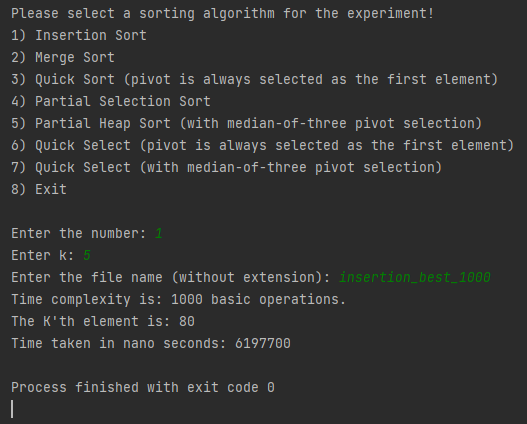
**Insertion Sort Time Complexity Analysis**

* **Best Case Complexity:** The insertion sort algorithm has a best-case time complexity of **O(n)** for the already sorted array because here, only the outer loop is running n times, and the inner loop is kept still.
* **Average Case Complexity:** The average-case time complexity for the insertion sort algorithm is **O(n2)**, which is incurred when the existing elements are in jumbled order, i.e., neither in the ascending order nor in the descending order.
* **Worst Case Complexity:** The worst-case time complexity is also **O(n2)**, which occurs when we sort the ascending order of an array into the descending order.  
  In this algorithm, every individual element is compared with the rest of the elements, due to which n-1 comparisons are made for every nthelement.

**Insertion Sort Experiment Results**

**Best case examples:**

**metin içeren bir resim

Açıklama otomatik olarak oluşturuldu **

**Worst case examples:**

**metin içeren bir resim

Açıklama otomatik olarak oluşturuldu metin içeren bir resim

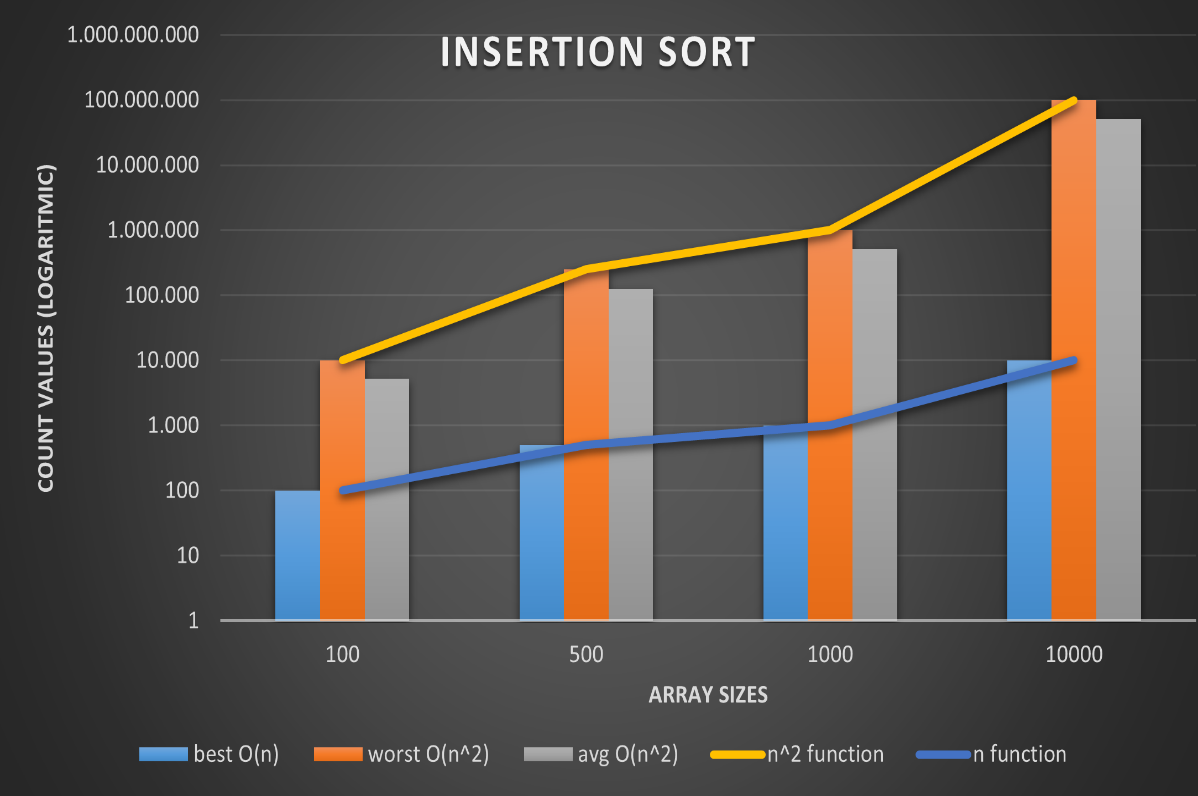
Açıklama otomatik olarak oluşturuldu**

**Average case examples:**

**metin içeren bir resim

Açıklama otomatik olarak oluşturuldu metin içeren bir resim

Açıklama otomatik olarak oluşturuldu**



**(This chart represents the results logarithmically)**

**Theoretical Expectations and Experiment Results Comparison**

**Approved Facts:**

Insertion sort theoretical time complexity expectation for best case is big O of *n* when the list is already sorted, and our experiment met with this fact.

In insertion experiment we saw that experimental results met with theoretical expectations. We confirmed that by looking at the chart above, for variant input sizes, the experimental results do not exceed the real *n^2* (shown in yellow)function for worst and average cases.

**For each individual input size scenario, the experimental results for basic operation counts turned out as expected in the theoretical analysis.**

* Big o of **n** for best case
* Big o of **n^2** for average and worst cases
* Average case (random list) is always smaller than the worst case (reverse ordered list) in basic operations count.

**Experimental Disjunctions:**

**No experimental disjunction has seen in the experiments for insertion sort.**

**Merge-sort**

Merge sort is a sorting technique based on divide and conquer technique. With worst-case time complexity being Ο(n log n), it is one of the most respected algorithms.

Merge sort first divides the array into equal halves and then combines them in a sorted manner.

In Merge Sort, the given unsorted array with n elements, is divided into n subarrays, each having one element, because a single element is always sorted in itself. Then, it repeatedly merges these subarrays, to produce new sorted subarrays, and in the end, one complete sorted array is produced.

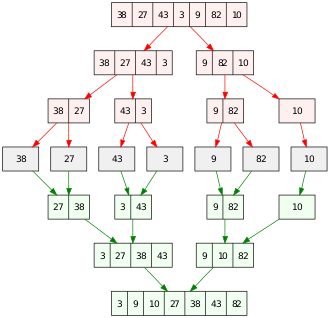
**Merge sort algorithm**

**Step 1** − if it is only one element in the list it is already sorted, return.

**Step 2** − divide the list recursively into two halves until it can no more be divided.

**Step 3** − merge the smaller lists into new list in sorted order.

**Merge sort visualization**



**Merge Sort Time Complexity Analysis**

* Time complexity of Merge Sort is O(n\*Log n) in all the 3 cases (worst, average and best) as merge sort always **divides** the array in two halves and takes linear time to **merge** two halves.

**Merge Sort Experiment Results**

**Best cases :**

**metin içeren bir resim

Açıklama otomatik olarak oluşturuldu metin içeren bir resim

Açıklama otomatik olarak oluşturuldu**

**Average cases :**

**metin içeren bir resim

Açıklama otomatik olarak oluşturuldu metin içeren bir resim

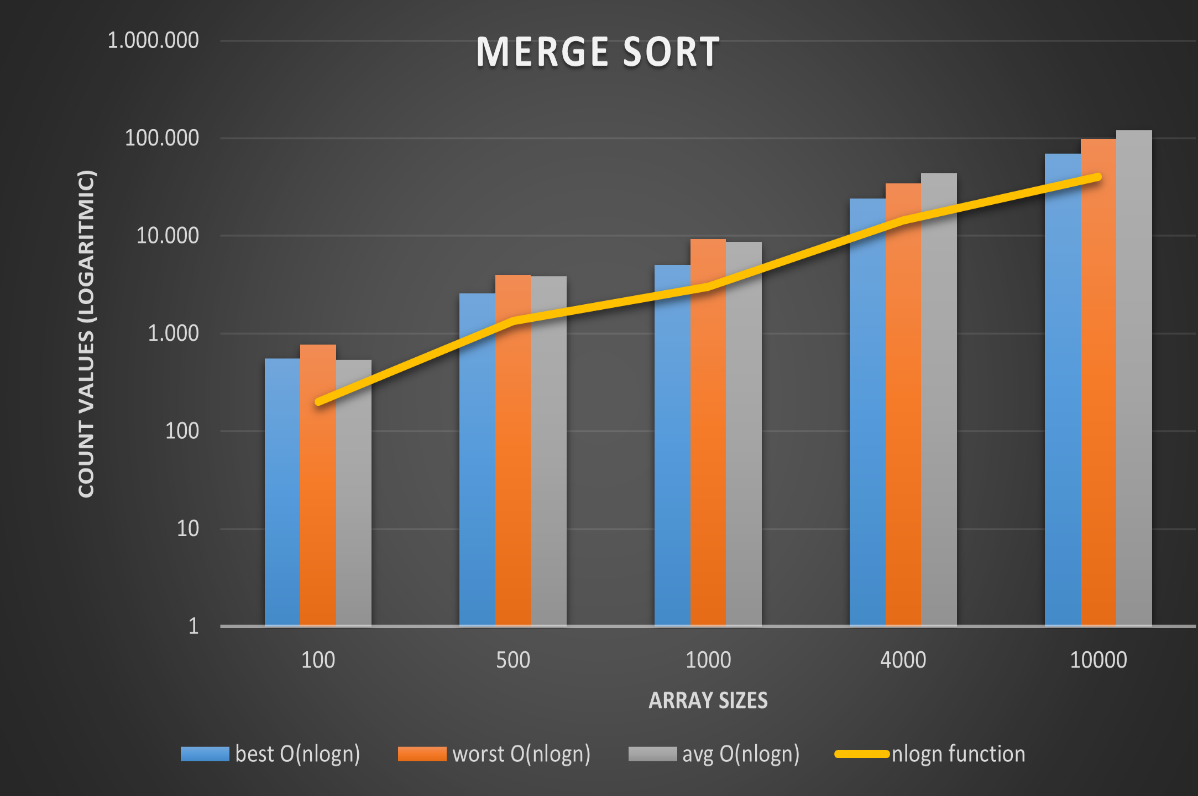
Açıklama otomatik olarak oluşturuldu**

**Worst cases :**

**metin içeren bir resim

Açıklama otomatik olarak oluşturuldu metin içeren bir resim

Açıklama otomatik olarak oluşturuldu**



**(This chart represents the results logarithmically)**

**Theoretical Expectations and Experiment Results Comparison**

**Approved Facts:**

In merge-sort experiment we saw that experimental results met with theoretical expectations. For each input size individually best case (already sorted list) operation count was the smallest, average case (randomly generated list) was the middle and the worst case was the biggest in operations count.

**For each individual input size scenario, the experimental results for basic operation counts turned out as expected in the theoretical analysis.**

**Experimental Disjunctions:**

In our merge sort experiment, experimental results of our algorithm’s basic operations count always exceeded the nlogn function for individual input sizes. We think that the reason behind this result is the basic operations counter places in the algorithms. Since the best, average and worst-case counts met with each input size this is the only explanation.

**Quicksort**

Like [Merge Sort](https://www.geeksforgeeks.org/merge-sort/), Quicksort is a Divide and Conquer algorithm. It picks an element as pivot and partitions the given array around the picked pivot. There are many different versions of quicksort that pick pivot in different ways.

1. Always pick first element as pivot.
2. Always pick last element as pivot.
3. Pick a random element as pivot.
4. Pick median as pivot.

Last element pivot selection is used in our implementations.

The key process in quicksort is partition (). Target of partitions is, given an array and an element x of array as pivot, put x at its correct position in sorted array and put all smaller elements (smaller than x) before x, and put all greater elements (greater than x) after x. All this should be done in linear time.

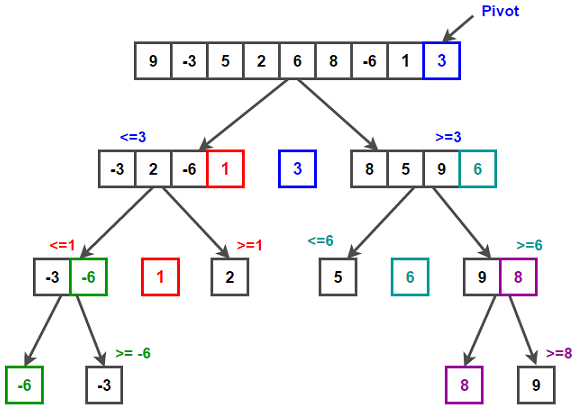
**Quicksort Algorithm**

* **Step 1 -**Consider the last element of the list as **pivot** (i.e., Element at first position in the list).
* **Step 2 -**Define two variables i and j. Set i and j to first and last elements of the list respectively.
* **Step 3 -**Increment i until list[i] > pivot then stop.
* **Step 4 -**Decrement j until list[j] < pivot then stop.
* **Step 5 -**If i < j then exchange list[i] and list[j].
* **Step 6 -**Repeat steps 3,4 & 5 until i > j.
* **Step 7 -**Exchange the pivot element with list[j] element.

metin içeren bir resim

Açıklama otomatik olarak oluşturuldu

**Quicksort algorithm visualization**



**Quicksort Time Complexity Analysis**

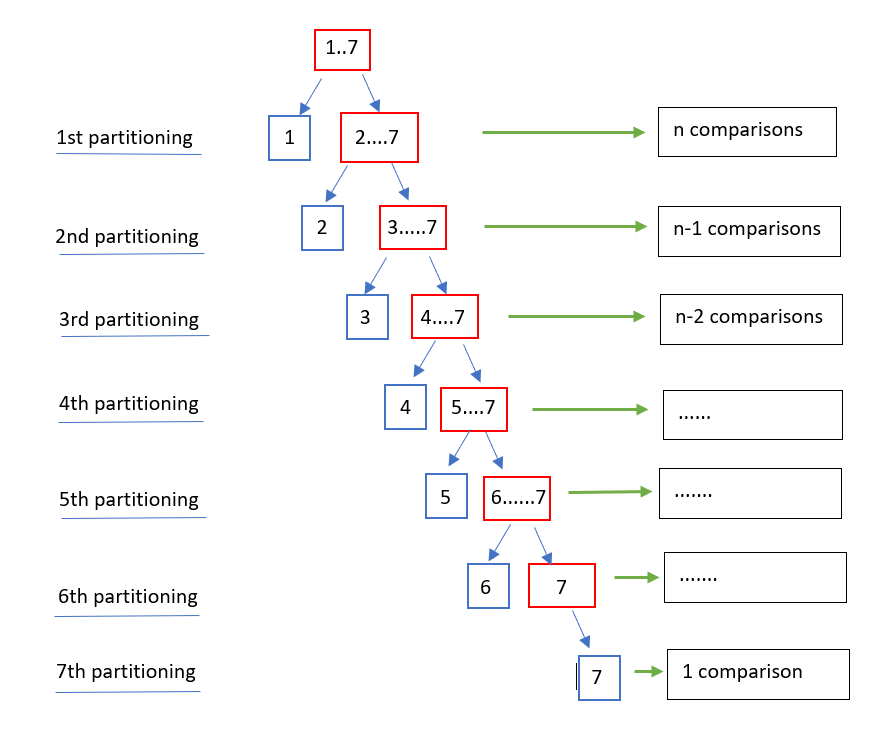
**Worst case**

Worst case of choosing the first element as pivot emerges when the array is already sorted ascending or descending order. Since every item is already located at their correct position, when partitioning process ended array is divided into very unbalanced parts.

For example lets thing about an array which is already sorted as:

1 2 3 4 5 6 7

Behavior of the algorithm is shown below for every call of recursive function:



So, time complexity will be which equals which is also .

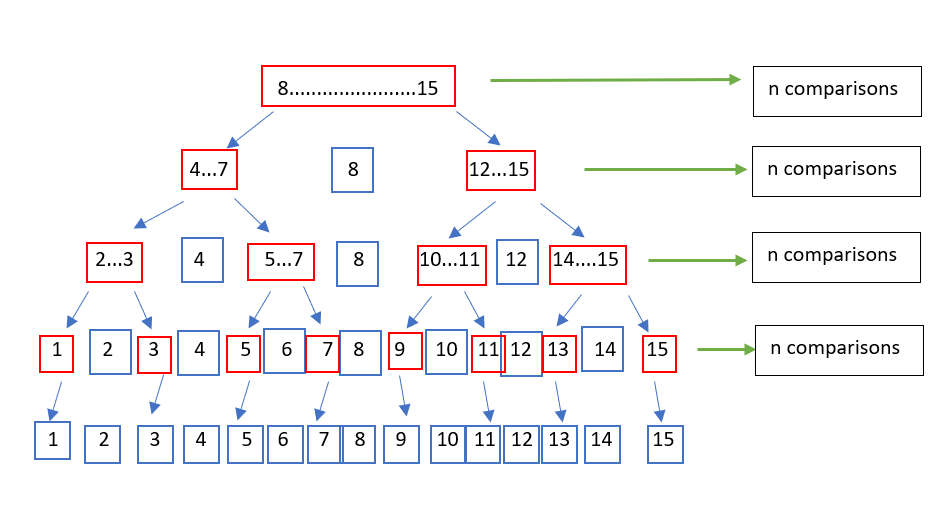
**Best & Average Case**

Best case of choosing first element pivot emerges when correct position of the pivot divides array into two equal parts. And this can be possible when array is unsorted and items randomly distributed.

For instance, we may give the following array as the best case:

8 4 2 1 3 6 5 7 12 10 9 11 14 13 15

Behavior of the algorithm is shown below for every call of recursive function:



Since pivot divides into two equal parts we can say that there are times n comparisons which gives

**Quicksort Experiment Results**

Best cases:

metin içeren bir resim

Açıklama otomatik olarak oluşturuldu metin içeren bir resim

Açıklama otomatik olarak oluşturuldu

Average cases:

metin içeren bir resim

Açıklama otomatik olarak oluşturuldu metin içeren bir resim

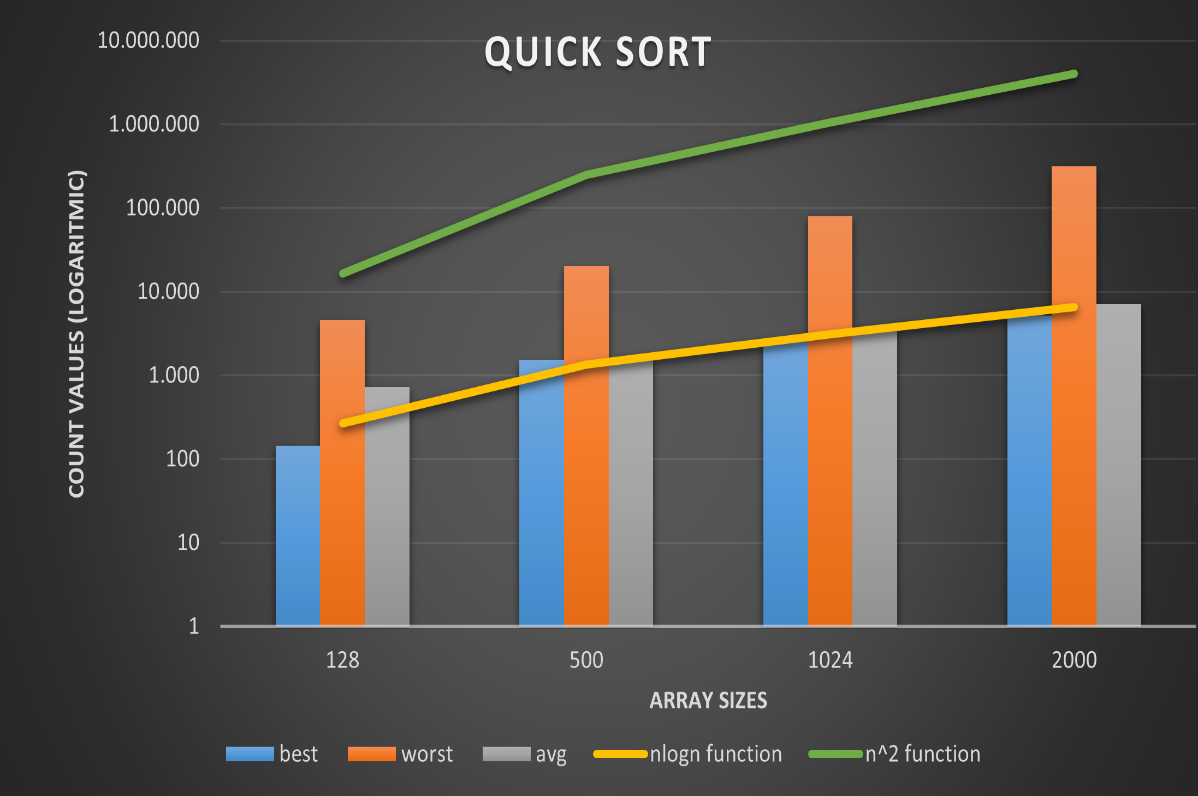
Açıklama otomatik olarak oluşturuldu

Worst Cases:

metin içeren bir resim

Açıklama otomatik olarak oluşturuldu metin içeren bir resim

Açıklama otomatik olarak oluşturuldu



**(This chart represents the results logarithmically)**

**Theoretical Expectations and Experiment Results Comparison**

**Approved Facts:**

Quick sort theoretical time complexity expectation was big O of *nlogn* for best and average cases.

In quick sort experiment we saw that experimental results met with theoretical expectations. We confirmed that by looking at the chart above, for variant input sizes, the experimental results do not exceed the real *nlogn* function in basic operation counts for best and average cases. In addition, worst case did not exceed the n^2 function which was expected.

**For each individual input size scenario, the experimental result for basic operation counts for best, average and worst case turned out as expected in the theoretical analysis.**

**Experimental Disjunctions:**

**No experimental disjunction has seen in the experiments for quick sort.**

**Partial Heapsort**

**Partial sorting** is a [relaxed](https://en.wikipedia.org/wiki/Relaxation_(approximation)) variant of the [sorting](https://en.wikipedia.org/wiki/Sorting_algorithm) problem. Total sorting is the problem of returning a list of items such that its elements all appear in order, while partial sorting is returning a list of the *k* smallest (or *k* largest) elements in order. The other elements (above the *k* smallest ones) may also be sorted, as in an in-place partial sort, or may be discarded, which is common in streaming partial sorts.

**Heapsort** is a [comparison-based](https://en.wikipedia.org/wiki/Comparison_sort) [sorting algorithm](https://en.wikipedia.org/wiki/Sorting_algorithm). Heapsort can be thought of as an improved [selection sort](https://en.wikipedia.org/wiki/Selection_sort): like selection sort, heapsort divides its input into a sorted and an unsorted region, and it iteratively shrinks the unsorted region by extracting the largest element from it and inserting it into the sorted region. Unlike selection sort, heapsort does not waste time with a linear-time scan of the unsorted region; rather, heap sort maintains the unsorted region in a [heap](https://en.wikipedia.org/wiki/Heap_(data_structure)) data structure to more quickly find the largest element in each step.

**Binary Heap:**

A **complete binary tree** is a binary tree in which all the levels are filled except the lowest one, which is filled from the left.

Complete binary tree is just like a full binary tree, but with two major differences.

1) All the leaf elements must lean towards the left. 2)The last leaf element might not have a right sibling i.e. a complete binary tree doesn't have to be a full binary tree.

A [**Binary**](https://www.geeksforgeeks.org/binary-heap/) **Heap** is a Complete Binary Tree where items are stored in a special order such that value in a parent node is greater (or smaller) than the values in its two child’s nodes. The former is called as max heap and the latter is called min-heap. The heap can be represented by a binary tree or array.

Since a Binary Heap is a Complete Binary Tree, it can be easily represented as an array and the array-based representation is space-efficient. If the parent node is stored at index I, the left child can be calculated by 2 \* I + 1 and right child by 2 \* I + 2 (assuming the indexing starts at 0).

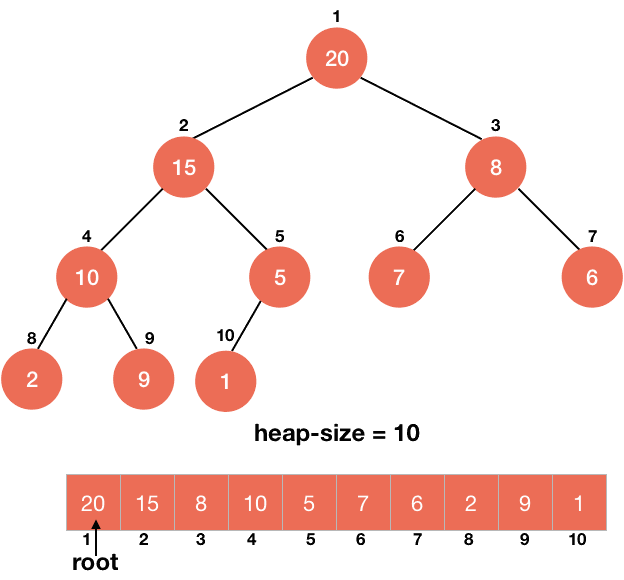
Heaps with **arrays with indexing** such as

* + Parent of a node i is ⌊i/2⌋
  + Left child of the node i is 2∗i
  + Right child of the node i is (2∗i) + 1
  + Max-heap property → The value of a node is greater than or equal to the value of its children i.e., A[Parent[i]] ≥ A[i] for all nodes i > 1.
  + Min-heap property → The value of a node is either smaller or equal to the value
  + of its children i.e., A[Parent[i]] ≤ A[i] for all nodes i >1.

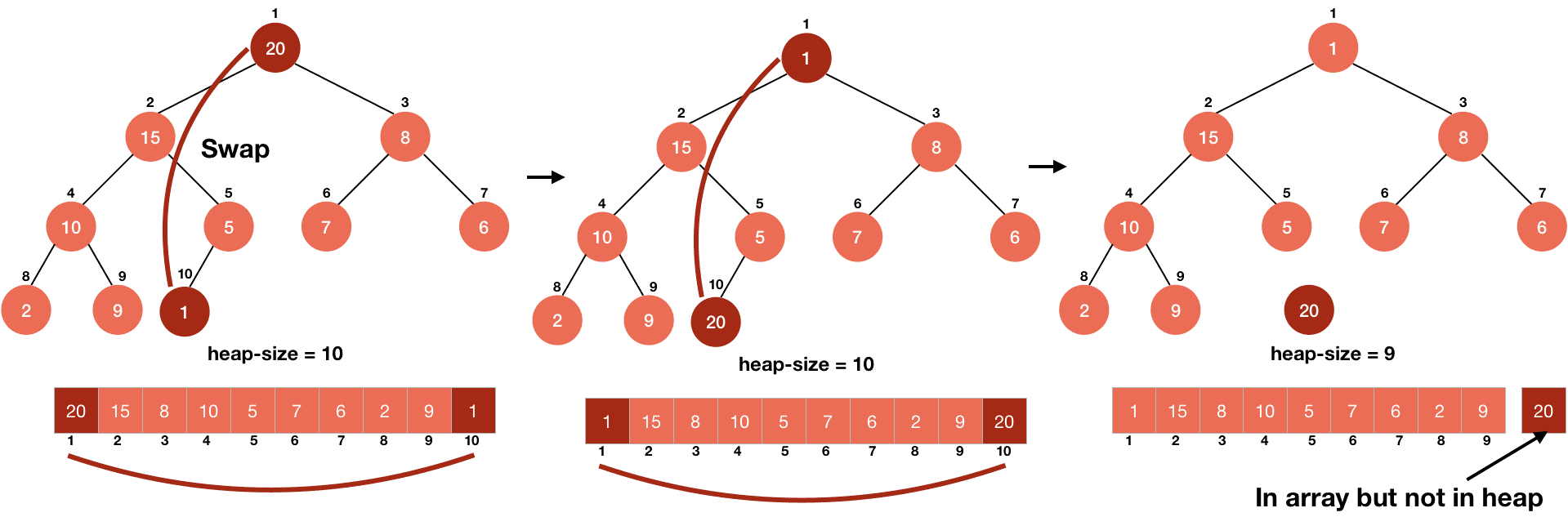
**Heap Sort Algorithm (for sorting in increasing order):**   
**1.** Build a max heap from the input data using heapify.  
**2.** At this point, the largest item is stored at the root of the heap. Replace it with the last item of the heap followed by reducing the size of heap by one. Finally, heapify the root of the tree.   
**3.** Repeat step 2 while size of the heap is greater than one.

**HEAPIFY** is a function used to maintain the heap property of any heap. It is applied to a node when the children of the node are following the property of a heap but the node itself may be violating it. It runs in O(logn) time.

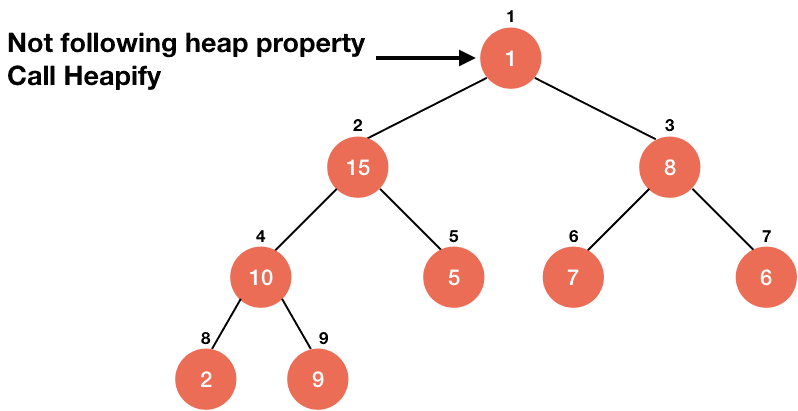
**Heapsort algorithm visualization**



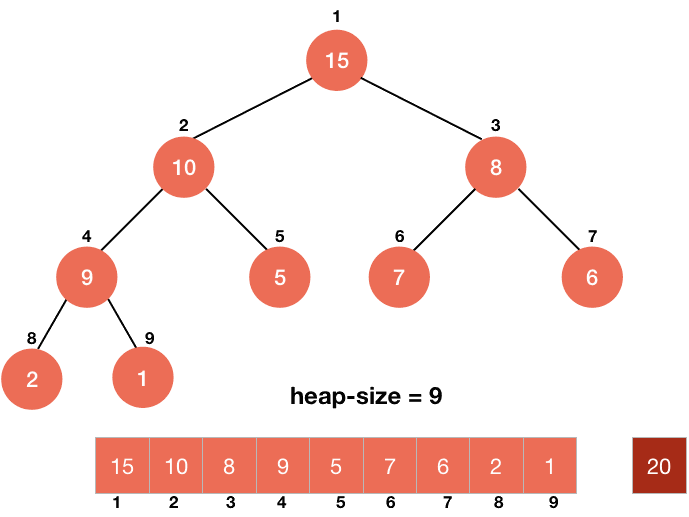
Now, we this maximum element must be at the last element of our sorted array. So, we swap it with the last element and then discard this element from the heap by reducing the size of the heap.



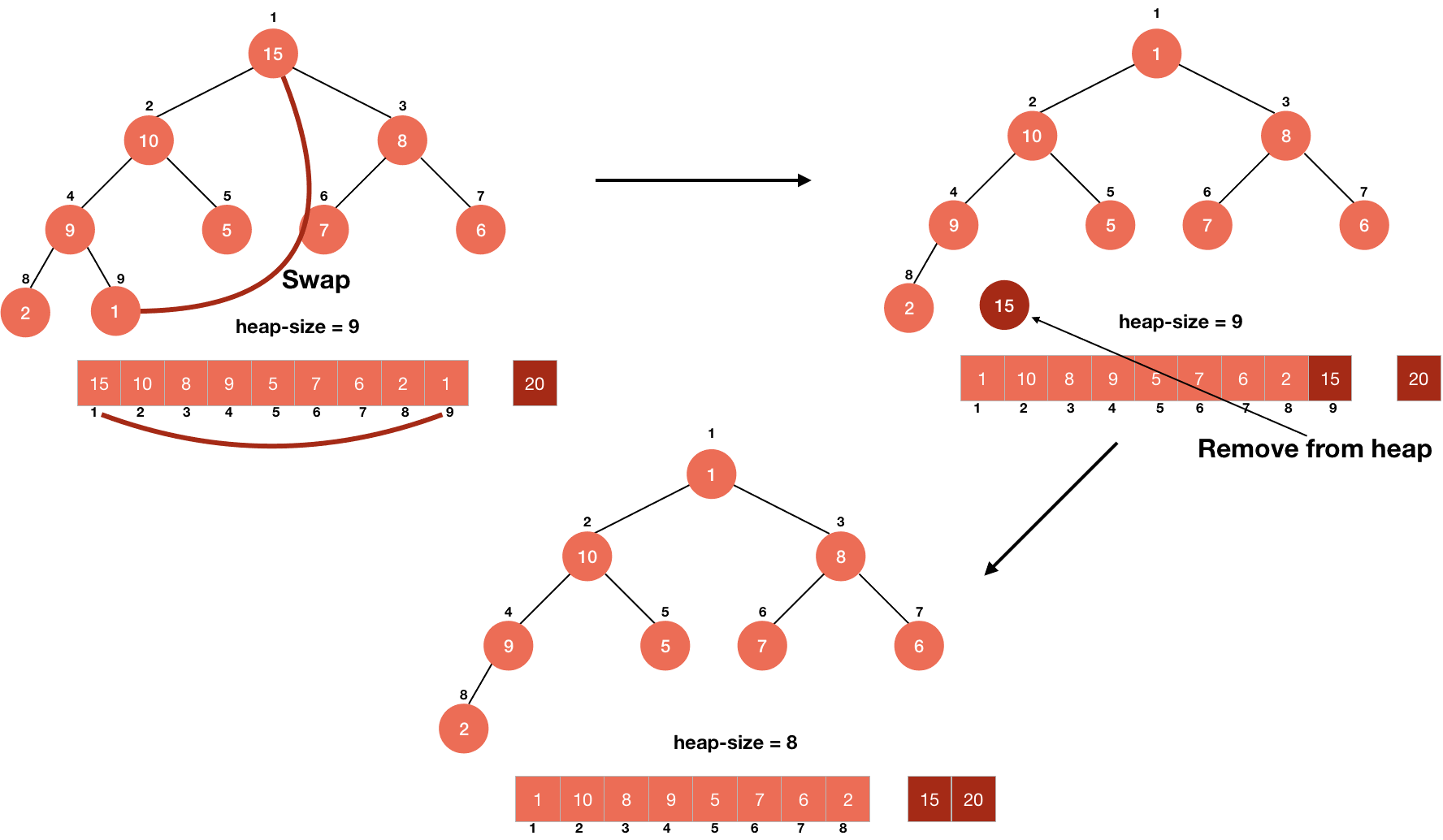
So, we have correctly put the largest element at the correct position, but in doing so, we have disturbed our heap or more precisely, the root of the heap because we haven't touched any child of the root yet. Thus, they are still following the max-heap property.

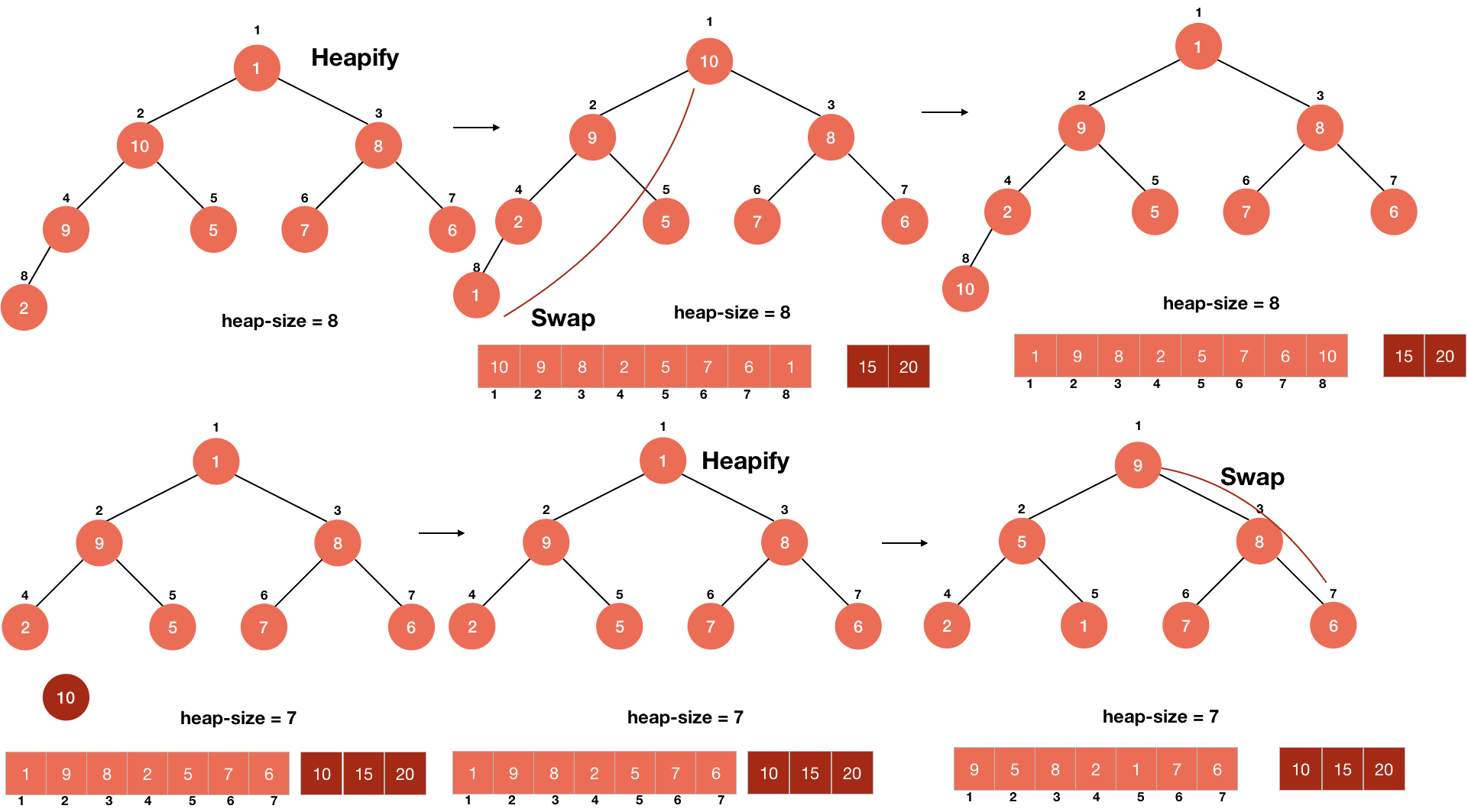


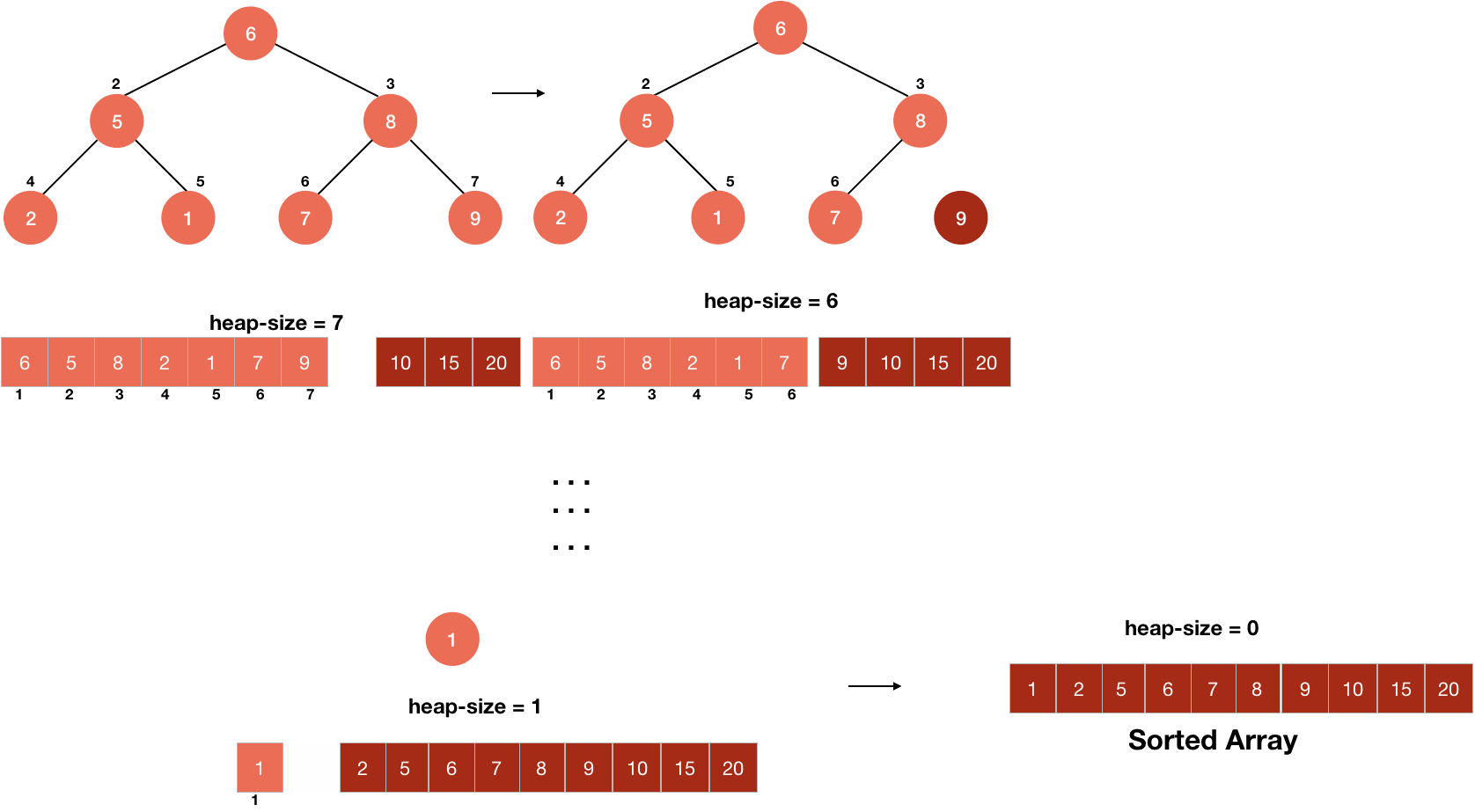
We can easily deal with this problem by calling HEAPIFY on the root of the tree and that will make our tree a max heap again, but this time the largest element won't be the part of the heap.



Now, we will just repeat this process until every element of the heap is correctly placed in sorted order.







**Difference Between Heap Sort and Partial Heap Sort**

Difference Between Heap Sort and Partial Heap Sort is the number of root deleting and heap rebuilding operations. In heap sort main goal is to build a heap and delete all the items one by one (only root items) and store them into the end of the array. This process continues until the last heap element has reached and deleted from the heap. On the other hand, in partial heap sort we only delete the necessary items after building the initial heap to reach the desired index or the subarray (which is sorted). This process reduces the time and computing power that is used for calculations. Reducing the number of operations makes a bigger impact when the desired index is smaller since we do not delete the root element and rebuild the heap repeatedly.

**Partial Heap Sort Experiment Results**

**metin içeren bir resim

Açıklama otomatik olarak oluşturuldu metin içeren bir resim

Açıklama otomatik olarak oluşturuldu**

**metin içeren bir resim

Açıklama otomatik olarak oluşturuldu metin içeren bir resim

Açıklama otomatik olarak oluşturuldu**

**metin içeren bir resim

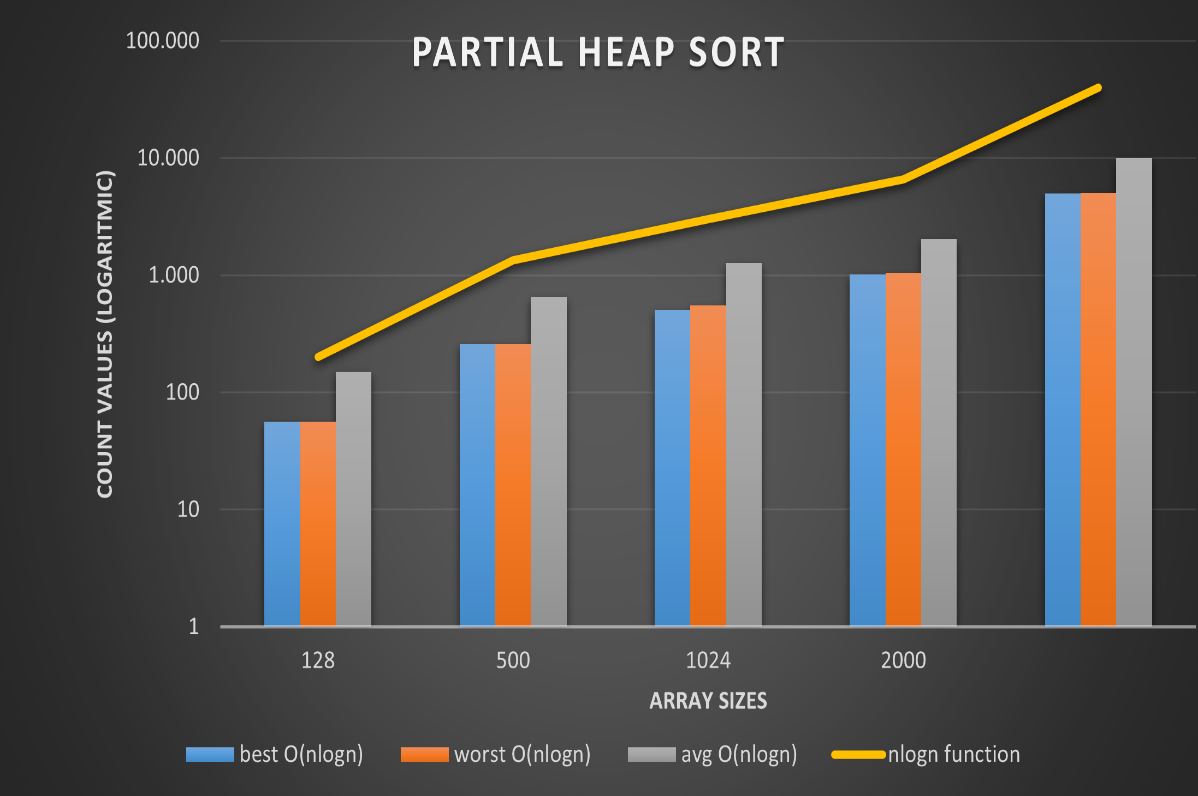
Açıklama otomatik olarak oluşturuldu metin içeren bir resim

Açıklama otomatik olarak oluşturuldu**

**metin içeren bir resim

Açıklama otomatik olarak oluşturuldu metin içeren bir resim

Açıklama otomatik olarak oluşturuldu**



**(This chart represents the results logarithmically)**

**Theoretical Expectations and Experiment Results Comparison**

**Approved Facts:**

Heap sort theoretical time complexity expectation was big O of *nlogn* for all best worst and average cases.

In heap sort experiment we saw that experimental results met with theoretical expectations. We confirmed that by looking at the chart above, for variant input sizes, the experimental results do not exceed the real *nlogn* function (shown in yellow).

**Experimental Disjunctions:**

**In heap sort experiment we expected that for each individual input size scenario the experimental results for basic operation counts would be same (or much closer to each other) but we could not approve it. As seen in the figure above, average cases (which are random lists) always exceed the best and worst-case scenarios in basic operation counts.**

**Quick Select Algorithm**

**(based on array partitioning, first element as pivot)**

The algorithm is like Quick Sort. The difference is, instead of recurring for both sides (after finding pivot), it recurs only for the part that contains the k-th smallest element. The logic is simple, if index of partitioned element is more than k, then we recur for left part. If index is same as k, we have found the k-th smallest element and we return. If index is less than k, then we recur for right part. This reduces the expected complexity from O (n log n) to O(n), with a worst case of O(n^2).

* Worst case occurs when we pick the largest/smallest element as pivot.
* Best case occurs when we partition the list into two halves and continue with only the half, we are interested in.
* The time complexity for the average case for quick select is O(n) (reduced from O(nlogn) — quick sort). The worst-case time complexity is still O(n²) but by using a random pivot, the worst case can be avoided in most cases. So, on an average quick select provides a O(n) solution to find the kth largest/smallest element in an unsorted list.

**Quick Select Algorithm**

Quick Select is a variation of the quicksort algorithm. It is an optimized way to find the kth smallest/largest element in an unsorted array.

Algorithm:

* The partition part of the algorithm is same as that of quick sort.
* After the partition function arranges the elements in list according to the pivot and returns the pivot\_index, instead of recursing both sides of the pivot index, we recurse only for the part that contains our desired element

metin içeren bir resim

Açıklama otomatik olarak oluşturuldu

**Quick Select Time Complexity Analysis:**

Worst case: Worst case occurs when we pick the largest/smallest element as pivot.

metin içeren bir resim

Açıklama otomatik olarak oluşturuldu

Best case: Best case occurs when we partition the list into two halves and continue with only the half we are interested in.

metin içeren bir resim

Açıklama otomatik olarak oluşturuldu

The time complexity for the average case for quick select is O(n) (reduced from O(nlogn) — quick sort). The worst-case time complexity is still O(n²) but by using a random pivot, the worst case can be avoided in most cases. So, on an average quick select provides a O(n) solution to find the kth largest/smallest element in an unsorted list.

**Quick select algorithm Experiment Results**

metin içeren bir resim

Açıklama otomatik olarak oluşturuldumetin içeren bir resim

Açıklama otomatik olarak oluşturulduBest Case Examples:

metin içeren bir resim

Açıklama otomatik olarak oluşturuldumetin içeren bir resim

Açıklama otomatik olarak oluşturulduAverage Case Examples:

metin içeren bir resim

Açıklama otomatik olarak oluşturuldumetin içeren bir resim

Açıklama otomatik olarak oluşturulduWorst Case Examples:



**(This chart represents the results logarithmically)**

**Theoretical Expectations and Experiment Results Comparison**

**Approved Facts:**

In this experiment we saw that experimental results met with theoretical expectations for average cases. For each input size individually average-case operation count was the smallest, worst case was the middle and the worst case was the biggest in operations count.

**For each individual input size scenario, the experimental results for basic operation counts for worst case turned out as expected in the theoretical analysis.**

**Experimental Disjunctions:**

**In this experiment we expected that for each individual input size scenario the experimental result for basic operation counts for average sizes would be in the middle of the best and worst cases. As seen in the figure best cases always exceed the average case counts.**

**In addition, in the time complexity analysis we expected the best-case scenario would follow the big O (n) notation, but it exceeded it.**

**Quick Select Algorithm**

**(based on array partitioning, pivot is the median of the last and the first element)**

The algorithm is the almost same Quick select algorithm described above except deciding which element will be the pivot. For example, if we have 7 element in an array the pivot will be the middle of the last and first position which is (7+1)/2 = 4. In this case (odd number array size) the pivot is middle of the first and last element. If the array size even we will do slightly different calculation. Let the array size 8 element. Then, the calculation is (8+1)/2 = 4,5. But the result not integer to do that we add the number comes before the result which is 4th element and comes after the result which is 5th element. After adding two number we will divide into 2 then we get the pivot. After finding pivot the procedure is the same Quick select first partitioning.

* Worst case occurs when we pick the largest/smallest element as pivot.
* Best case occurs when we partition the list into two halves and continue with only the half, we are interested in.
* The time complexity for the average case for quick select is O(nlogn) — quick sort). The worst-case time complexity is still O(n²) but by using a random pivot, the worst case can be avoided in most cases. So, on an average quick select provides a O(n) solution to find the kth largest/smallest element in an unsorted list.

Quick Select is a variation of the quicksort algorithm. It is an optimized way to find the kth smallest/largest element in an unsorted array.

The median of three has you look at the first, middle and last elements of the array, and choose the median of those three elements as the pivot.

To get the "full effect" of the median of three, it's also important to *sort* those three items, not just use the median as the pivot -- this doesn't affect what's chosen as the pivot in the current iteration, but can/will affect what's used as the pivot in the next recursive call, which helps to limit the bad behavior for a few initial orderings (one that turns out to be particularly bad in many cases is an array that's sorted, except for having the smallest element at the high end of the array (or largest element at the low end). For example:

Compared to picking the pivot randomly:

1. It ensures that one common case (fully sorted data) remains optimal.
2. It's more difficult to manipulate into giving the worst case.

metin içeren bir resim

Açıklama otomatik olarak oluşturuldu

**Quick Select median of three inputs**

* The best-case input is an array that the middlest element is in the middle. In this case insertion sort has a logarithmic running time (i.e., O(*nlogn*))
* The simplest worst-case input is an array that the biggest or lowest element is in the middle of the whole array.. This gives Quick select a quadratic running time (i.e., O(*n*2))
* The average case is also logarithmic, which makes Quick select practical for searching specific number in a random large arrays. Average case input is an array that is not sorted. (random elements)

**Quick Select Time Complexity Analysis:**

Worst case: Worst case occurs when we pick the largest/smallest element as pivot.

metin içeren bir resim

Açıklama otomatik olarak oluşturuldu

Best case: Best case occurs when we partition the list into two halves and continue with only the half we are interested in.

metin içeren bir resim

Açıklama otomatik olarak oluşturuldu

The time complexity for the average case for quick select O(nlogn) — quick sort). The worst-case time complexity is still O(n²) but by using a random pivot, the worst case can be avoided in most cases. So, on an average quick select provides a O(nlogn) solution to find the kth largest/smallest element in an unsorted list.

**Quick select algorithm Experiment Results**

Best Case Examples:

metin içeren bir resim

Açıklama otomatik olarak oluşturuldu metin içeren bir resim

Açıklama otomatik olarak oluşturuldu

metin içeren bir resim

Açıklama otomatik olarak oluşturuldu

Average Case examples :

metin içeren bir resim

Açıklama otomatik olarak oluşturuldu metin içeren bir resim

Açıklama otomatik olarak oluşturuldu

metin içeren bir resim

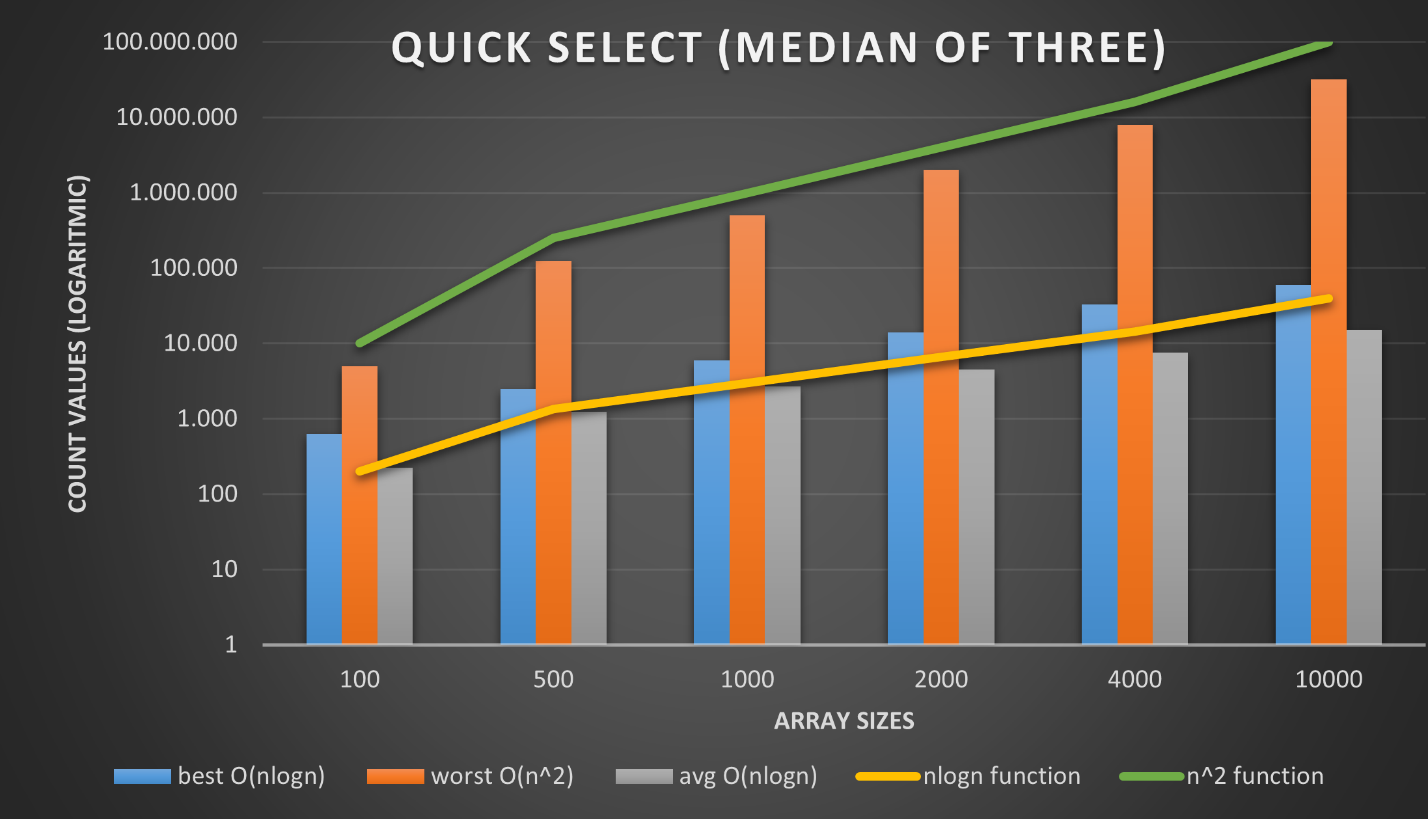
Açıklama otomatik olarak oluşturuldu

Worst Case examples:

metin içeren bir resim

Açıklama otomatik olarak oluşturuldu metin içeren bir resim

Açıklama otomatik olarak oluşturuldu



**Theoretical Expectations and Experiment Results Comparison**

**Approved Facts:**

In this experiment we saw that experimental results met with theoretical expectations for every cases. For each input size individually average-case operation count was the smallest, best case was the middle and the worst case was the biggest in operations count.

**For each individual input size, the experimental results for basic operation counts for every case turned out as expected in the theoretical analysis except for best and average cases’ slight difference but almost same which is again met with the theoretical analysis.**

**Experimental Disjunctions:**

**In this experiment we expected that for each individual input size scenario the experimental result for basic operation counts for average sizes would be in the middle of the best and worst cases. As seen in the figure best cases always lower than the average case counts.**

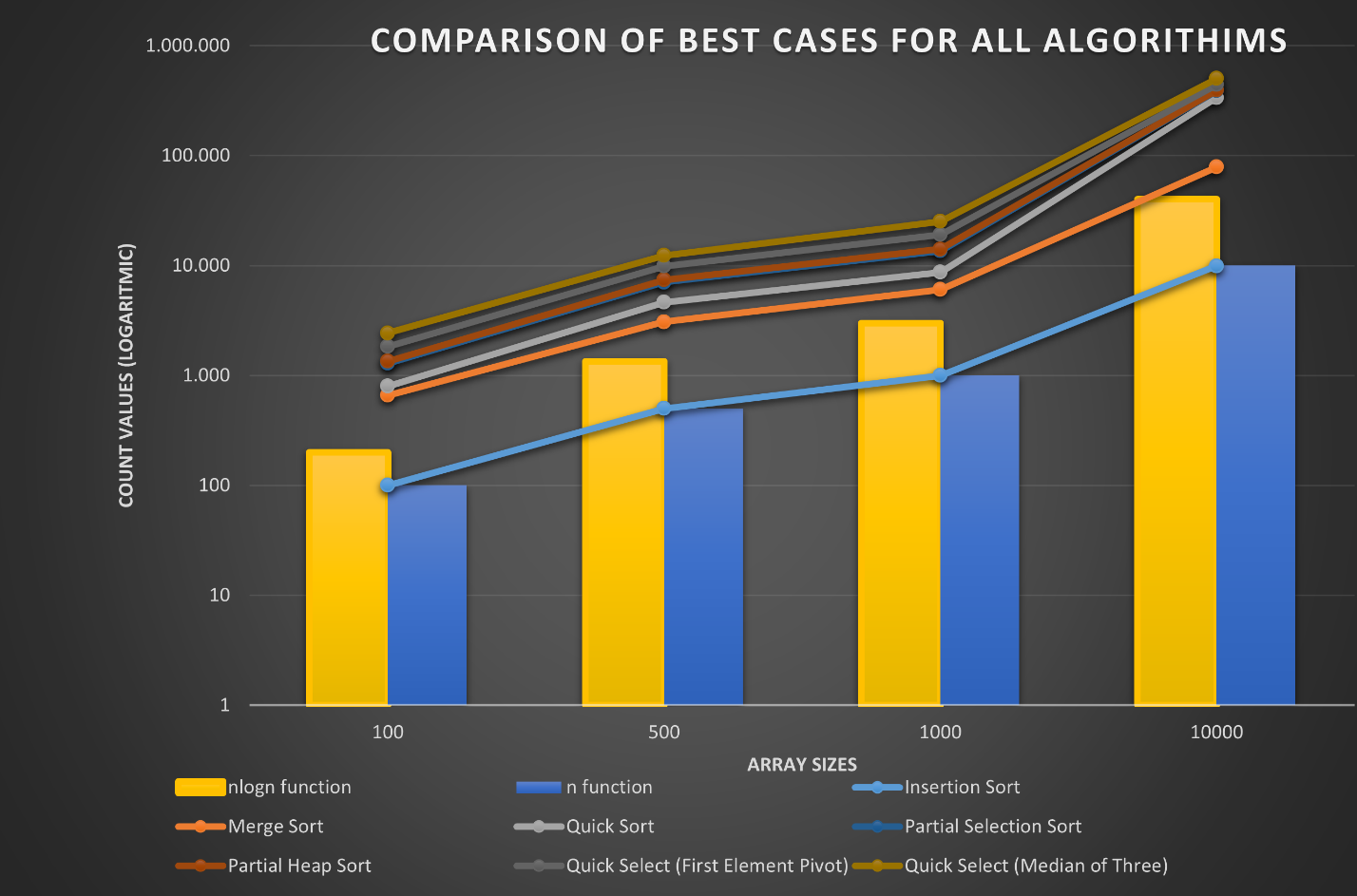
PART B

In this algorithm when we are calculating the time complexity, we use a count variable to count how many basic operations are done in a single algorithm. Then, we used the count variable in each algorithm in the list. At first, we decided to use real time to calculate time complexities but when we do that the other programs interrupt the CPU and make our measurements wrong. To achieve that, we use count variable described above for every case with different array size to measure every individual case time complexities by dividing previous measurement.

COMPARING ALL ALGORITHMS

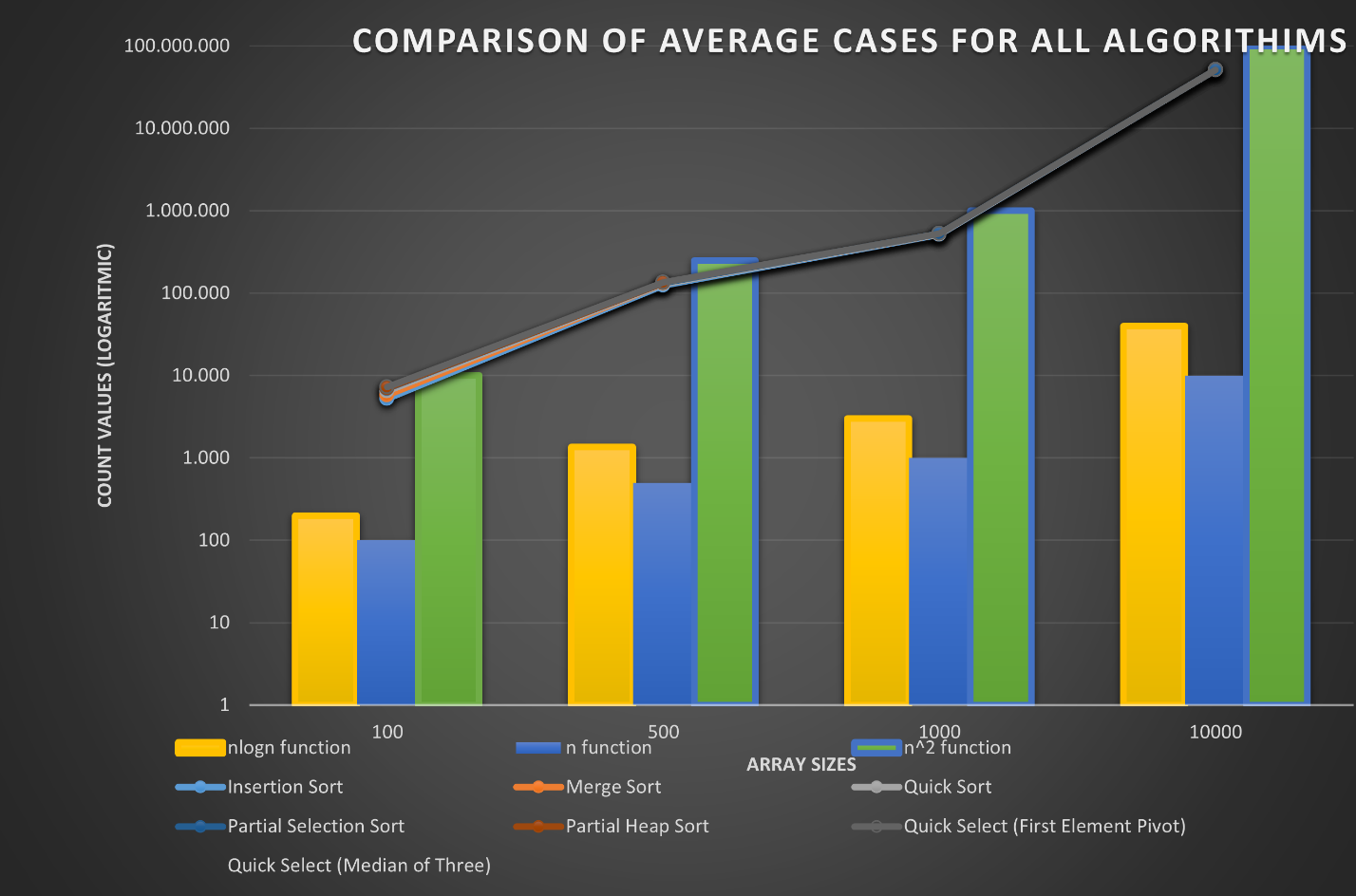
All the algorithms have special inputs and different workspace. In order to that, we can’t do healthy comparisons. To achieve over the some of problems we decided to go over the cases for each algorithm. But since we used count variable not the real time, some of the result reaches millions the others were very small besides. This causes the chart are intelligible.

**Best Cases:**



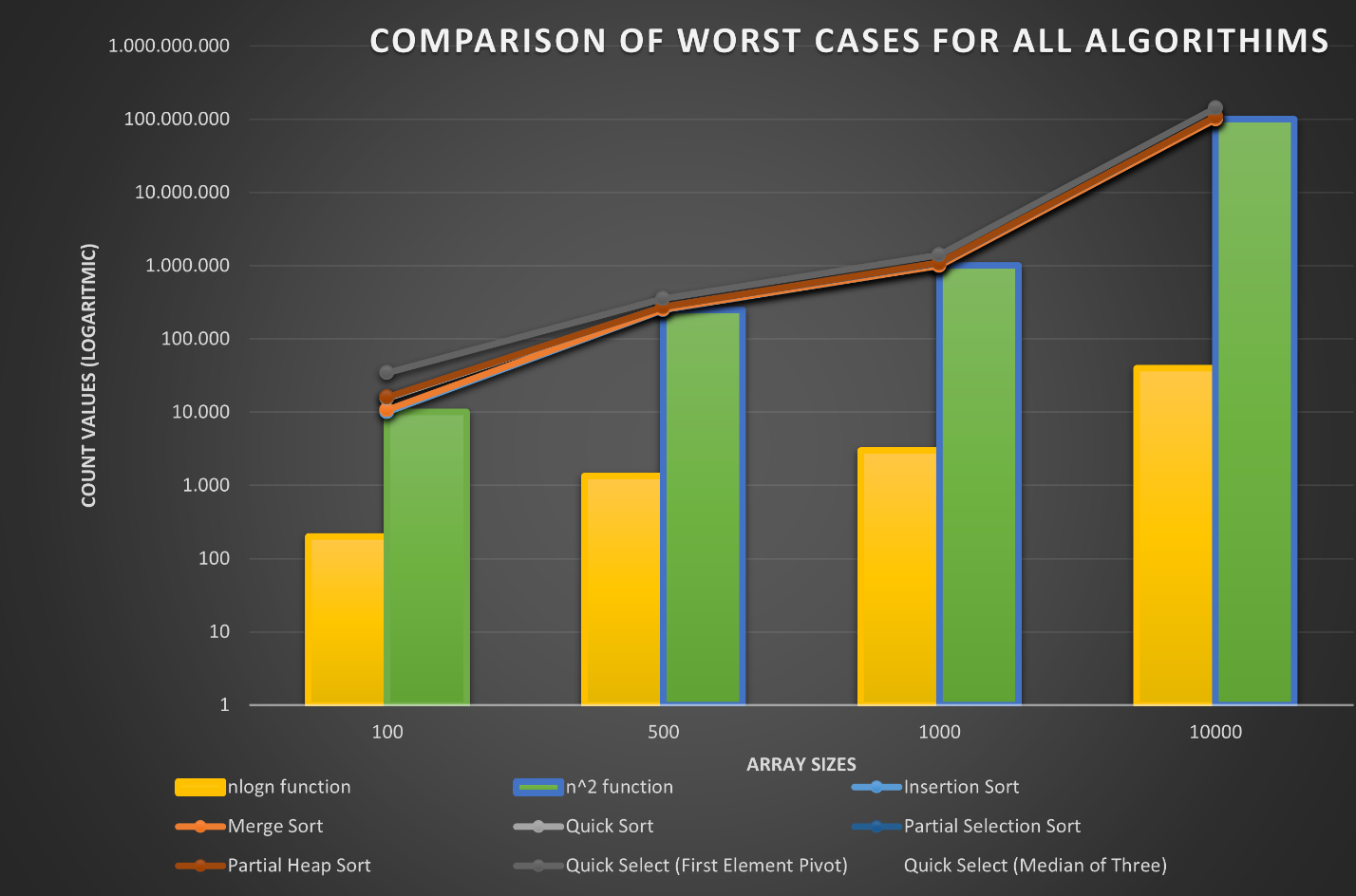
We visualize the count axis as logarithmic because of the problems described above. In this chart,The columns shows that the real nlogn, n and n^2 functions. And the lines describes the best cases for algorithms. In order to lines, its clear that the Merge sort has the minimum best case counts so we can say that the merge sort is best among the others.

**Average Cases:**



In average cases we know that some of the algorithms has O(nlogn) but in this chart we cant see it. Because of while representing the real functions we used the real times but while representing the algorithms we used count variable in this case count sizes exceed the real size.

**Worst Cases:**

****

In this chart again columns represent the real time functions and the lines counts values claimed by the experiment. By looking at the lines its clear that the merge sort has the minimum count values for different input sizes among the other. So we can say that in the worst case the best algorithm again merge sort.