ANALYSIS of SELECTION SORT

THEORETİCAL ANALYSİS OF SELECTİON SORT

- PSEUDO-CODE:

```
1. k ← length [A]
2.  for j ←1 to n-1
3. smallest ←  j
4. for I ← j + 1 to k
5. if A [i] < A [ smallest]
6. then smallest ←  i
7. exchange (A [j], A [smallest])
```

EXAMPLE:

A [] = (7, 4, 3, 6, 5).
A [] =

7	4	3	6	5

1st Iteration:

Set minimum = 7

Compare a_0 and a_1

7	4	3	6	5

As, $a_0 > a_1$, set minimum = 4.

Compare a_1 and a_2

7	4	3	6	5

As, $a_2 < a_3$, set minimum= 3.

Compare a_2 and a_3

7	4	3	6	5

As, $a_2 < a_3$, set minimum= 3.

Compare a_2 and a_4

7	4	3	6	5

As, $a_2 < a_4$, set minimum =3.

Since 3 is the smallest element, so we will swap a_0 and a_2.

3	4	7	6	5

If we continue to apply these steps, it will end at the n-th iteration (For this example it is 4).

- COMPLEXITY ANALYSIS OF SELECTION SORT:

Input: Given **n** input elements.

Output: Number of steps incurred to sort a list.

Logic: If we are given n elements, then in the first pass, it will do **n-1** comparisons; in the second pass, it will do **n-2**; in the third pass, it will do **n-3** and so on. Thus, the total number of comparisons can be found by;

Output:

$$(n-1)+(n-2)+(n-3)+(n-4)+\dots+1$$

$$\text{Sum} = \frac{n(n-1)}{2} \Rightarrow O(n^2)$$

Therefore, the selection sort algorithm encompasses a time complexity of **O(n^2)** and a space complexity of **O(1)** because it necessitates some extra memory space for temp variable for swapping.

Time Complexities:

- **Best Case Complexity:** The selection sort algorithm has a best-case time complexity of **O(n^2)** for the already sorted array.
- **Average Case Complexity:** The average-case time complexity for the selection sort algorithm is **O(n^2)**, in which the existing elements are in jumbled ordered, i.e., neither in the ascending order nor in the descending order.
- **Worst Case Complexity:** The worst-case time complexity is also **O(n^2)**, which occurs when we sort the descending order of an array into the ascending order.

In the selection sort algorithm, the time complexity is **O(n^2)** in all three cases. This is because, in each step, we are required to find **minimum** elements so that it can be placed in the correct position. Once we trace the complete array, we will get our minimum element.

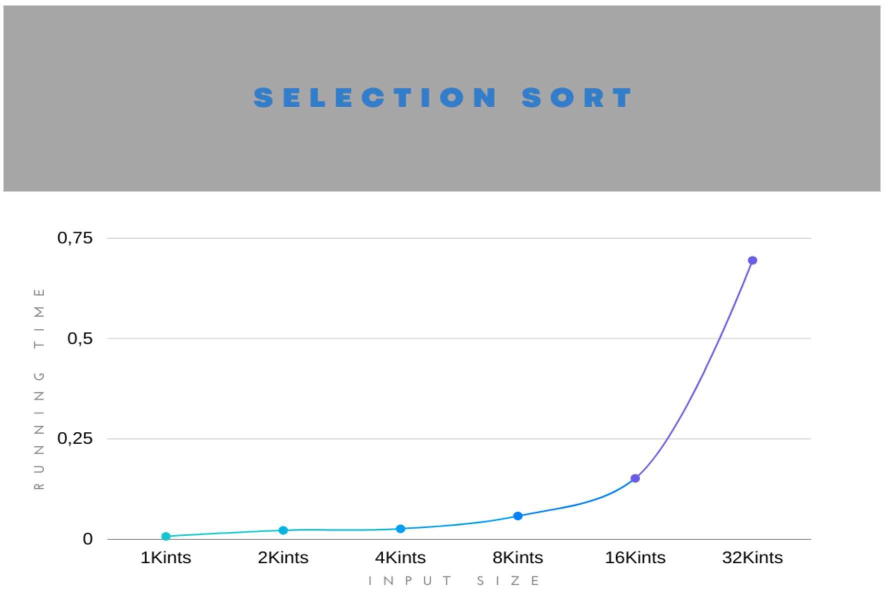
EMPİRİCAL ANALYSİS OF SELECTİON SORT:

Input Size	Running Time
1Kints	0.006
2K	0.021
4K	0.025
8K	0.057
16K	0.151
32K	0.635

First, I implemented the Selection Sort algorithm for different input types and sizes. Then, I noted their running times in the table.

According to table, as the number of inputs increased exponentially, the running times got longer.

Here is the graph of Input-Running Time of Selection Sort:



ANALYSIS of INSERTION SORT

- PSEUDO-CODE:

1.	for j = 2 to A.length
2.	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.	A[i + 1] = A[i]
7.	ii = i -1
8.	A[i + 1] = key

EXAMPLE:

A = (41, 22, 63, 14, 55, 36)

1st Iteration:

41	22	63	14	55

Set key = 22

Compare a1 with a0

41	22	63	14	55

Since a0 > a1, swap both of them.

22	41	63	14	55

2nd Iteration:

Set key = 63

Compare a2 with a1 and a0

22	41	63	14	55

Since a2 > a1 > a0, keep the array as it is.

3rd Iteration:

Set key = 14

Compare a3 with a2, a1 and a0

22	41	63	14	55

4th Iteration:

Set key = 55

Compare a4 with a3, a2, a1 and a0.

14	22	41	63	55

As a4 < a3, swap both of them.

14	22	41	55	63

Hence the array is arranged in ascending order, so no more swapping is required.

- COMPLEXİTY ANALYSİS OF INSERTİON SORT:

Input: Given n input elements.

Output: Number of steps incurred to sort a list.

Logic: If we are given **n** elements, then in the first pass, it will make **n-1** comparisons; in the second pass, it will do **n-2**; in the third pass, it will do **n-3** and so on. Thus, the total number of comparisons can be found by;

Output:

$$(n-1)+(n-2)+(n-3)+(n-4)+\ldots+1$$

$$\text{Sum} = \frac{n(n-1)}{2} \Rightarrow O(n^2)$$

Therefore, the insertion sort algorithm encompasses a time complexity of **O(n^2)** and a space complexity of **O(1)** because it necessitates some extra memory space for a **key** variable to perform swaps.

Time Complexities:

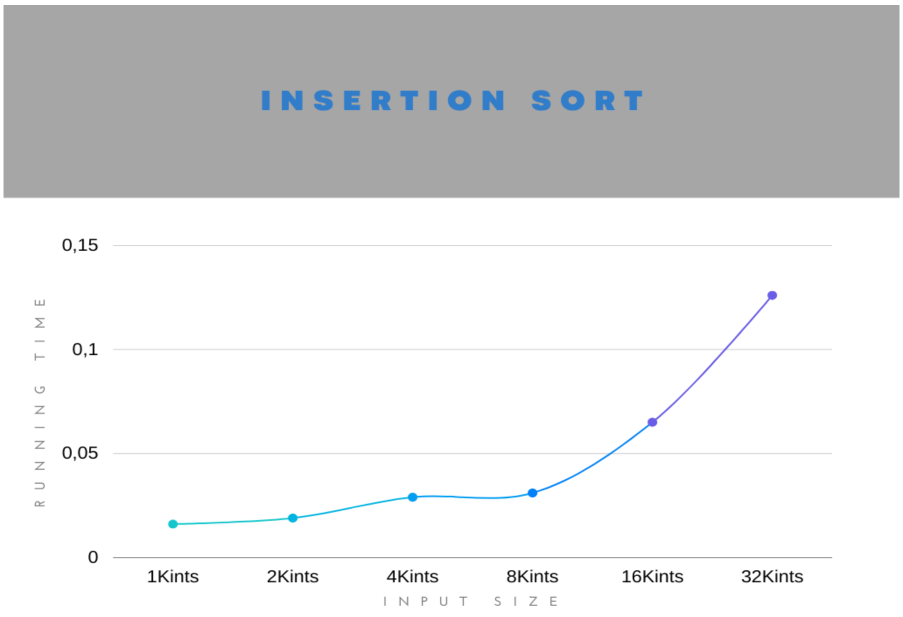
- **Best Case Complexity:** The insertion sort algorithm has a best-case time complexity of **O(n)** for the already sorted array because here, only the outer loop is running n times, and the inner loop is kept still.
- **Average Case Complexity:** The average-case time complexity for the insertion sort algorithm is **O(n^2)**, 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(n^2)**, 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 nth element.

EMPİRİCAL ANALYSİS OF INSERTION SORT:

Input Size	Running Time
1Kints	0.016
2K	0.019
4K	0.029
8K	0.031
16K	0.065
32K	0.126

First, I implemented the Insertion Sort algorithm for different input types and sizes. Then, I noted their running times in the table.

Here is the graph of Input-Running Time of Insertion Sort:



Comments:

We can say that the relationship between data size and running time is growing exponentially, but not orderly. For instance, growth curve between 4K and 8K formed unexpectedly. According to my experimental result, the reason for this situation may be the speed of our computer.

If we compare Selection Sort and Insertion Sort, we can say that Selection Sort is faster than Insertion Sort considering running time – input size graphs.

ANALYSIS of MERGE SORT

- PSEUDO-CODE

FUNCTIONS: MERGE (A, p, q, r)

```
1. n 1 = q-p+1
2. n 2= r-q
3. create arrays [1.....n 1 + 1] and R [ 1.....n 2 +1 ]
4. for i ← 1 to n 1
5. do [i] ← A [ p+ i-1]
6. for j ← 1 to n2
7. do R[j] ← A[ q + j]
8. L [n 1+ 1] ← ∞
9. R[n 2+ 1] ← ∞
10. I ← 1
11. J ← 1
12. For k ← p to r
13. Do if L [i] ≤ R[j]
14. then A[k] ← L[ i]
15. i ← i +1
16. else A[k] ← R[j]
17. j ← j+1
```

EXAMPLE:

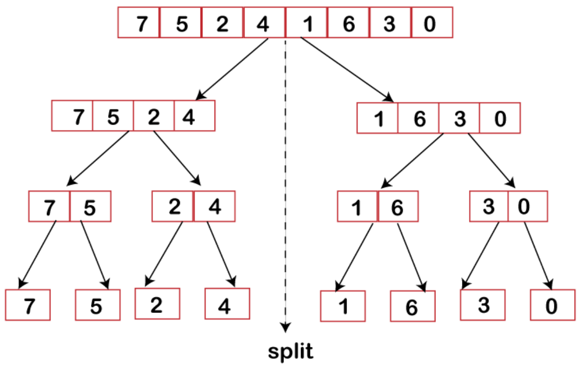


Figure 1: Merge Sort Divide Phase

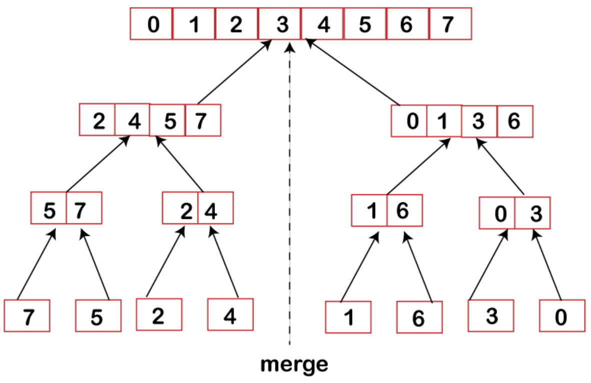


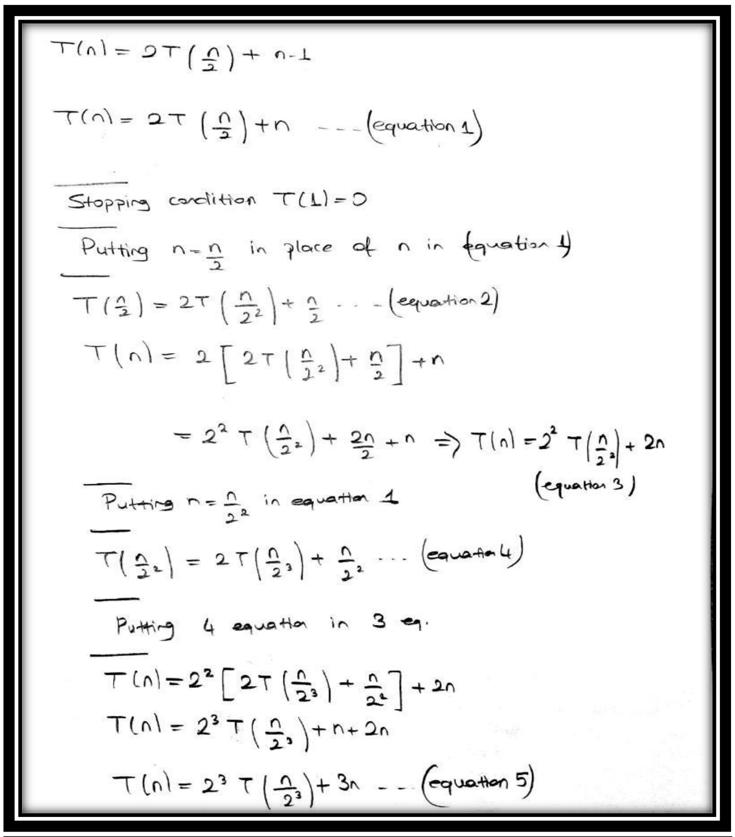
Figure 2: Merge Sort Combine Phase

- COMPLEXİTY ANALYSİS OF MERGE SORT

Let T (n) be the total time taken by the Merge Sort algorithm.

Sorting two halves will take at the most $2T\frac{n}{2}$ time.

When we merge the sorted lists, we come up with a total n-1 comparison because the last element which is left will need to be copied down in the combined list, and there will be no comparison. Thus, the relational formula will be



$T(n) = 2T\left(\frac{n}{2}\right) + n-1$

$T(n) = 2T\left(\frac{n}{2}\right) + n$ --- (equation 1)

Stopping condition $T(1) = 0$

Putting $n = \frac{n}{2}$ in place of n in equation 1

$T\left(\frac{n}{2}\right) = 2T\left(\frac{n}{2^2}\right) + \frac{n}{2}$ --- (equation 2)

$T(n) = 2\left[2T\left(\frac{n}{2^2}\right) + \frac{n}{2}\right] + n$

$= 2^2 T\left(\frac{n}{2^2}\right) + \frac{2n}{2} + n \Rightarrow T(n) = 2^2 T\left(\frac{n}{2^2}\right) + 2n$

(equation 3)

Putting $n = \frac{n}{2^2}$ in equation 1

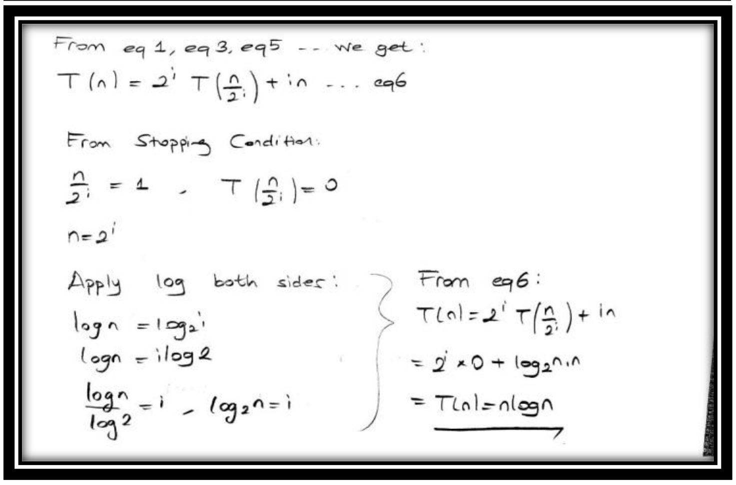
$T\left(\frac{n}{2^2}\right) = 2T\left(\frac{n}{2^3}\right) + \frac{n}{2^2}$ ... (equation 4)

Putting 4 equation in 3 eq.

$T(n) = 2^2\left[2T\left(\frac{n}{2^3}\right) + \frac{n}{2^2}\right] + 2n$

$T(n) = 2^3 T\left(\frac{n}{2^3}\right) + n + 2n$

$T(n) = 2^3 T\left(\frac{n}{2^3}\right) + 3n$ -- (equation 5)



From eq 1, eq 3, eq5 -- we get:

$T(n) = 2^i T\left(\frac{n}{2^i}\right) + in$ ... eq6

From Stopping Condition:

$\frac{n}{2^i} = 1$, $T\left(\frac{n}{2^i}\right) = 0$

$n = 2^i$

Apply log both sides:

$\log n = \log 2^i$

$\log n = i \log 2$

$\frac{\log n}{\log 2} = i$, $\log_2 n = i$

From eq6:

$T(n) = 2^i T\left(\frac{n}{2^i}\right) + in$

$= 2^i \times 0 + \log_2 n \cdot n$

$= \underline{T(n) = n\log n}$

Best Case Complexity: The merge sort algorithm has a best-case time complexity of **O(n*log n)** for the already sorted array.

Average Case Complexity: The average-case time complexity for the merge sort algorithm is **O(n*log n)**, which happens when 2 or more elements are jumbled, neither in the ascending order nor in the descending order.

Worst Case Complexity: The worst-case time complexity is also **O(n*log n)**, which occurs when we sort the descending order of an array into the ascending order.

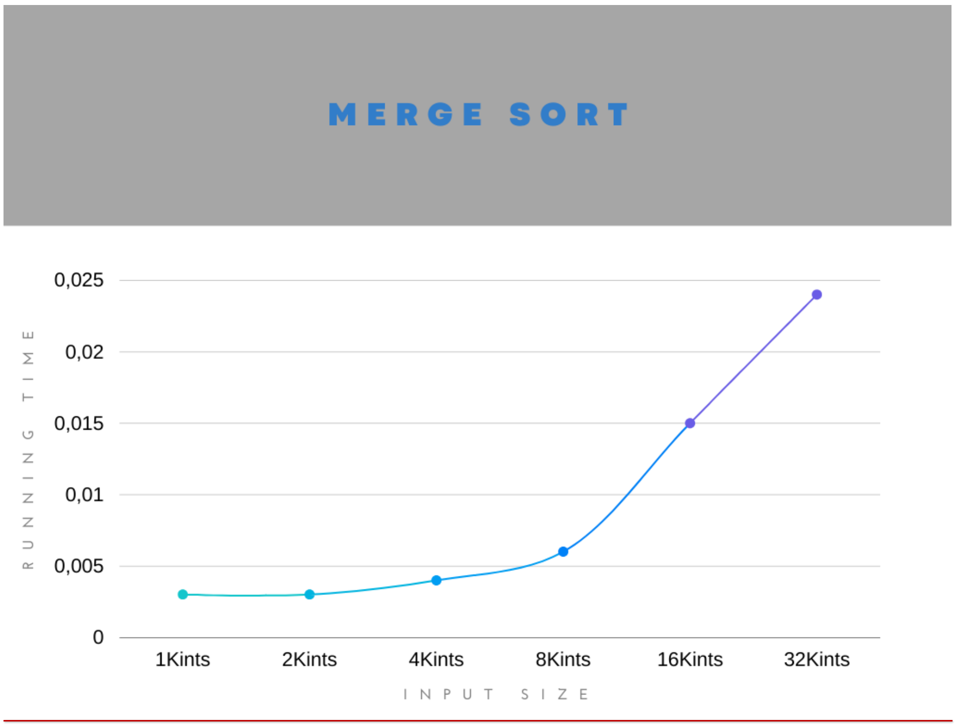
Space Complexity: The space complexity of merge sort is **O(n)**. We have used extra memory.

EMPİRİCAL ANALYSİS OF MERGE SORT:

Input Size	Running Time
1Kints	0.003
2K	0.003
4K	0.004
8K	0.006
16K	0.015
32K	0.024

First, I implemented the Merge Sort algorithm for different input types and sizes. Then, I noted their running times in the table.

Here is the graph of Input-Running Time of Merge Sort:



Comments:
Merge sort is one of the best sorting techniques that successfully build a recursive algorithm.
If we look at the graphs, comparing to other sorting algorithms (Selection and Insertion Sort), running time scale is very small.
Besides, graph is almost as a n*logn graph. That's mean, our experimental design is close to theoretical graph of worst case efficiency of Merge Sort.

ANALYSIS of QUICK SORT

LOMUTO PARTITIONING ALGORITHM:

This works by iterating over the input array, swapping elements that are strictly less than a pre-selected pivot element. They appear earlier in the array, but on a sliding target index.

- PSEUDO-CODE

```
procedure LomutoPartition($A[1..n]$)
1: p $\leftarrow A[n]$
2: i $\leftarrow 1$
3: for j $\leftarrow 1$; j $< n$; j $\leftarrow$ j $+ 1$ do
4:     if $A[j] <$ p then
5:         Swap $A[i]$ and $A[j]$
6:         i $\leftarrow$ i $+ 1$
7: Swap $A[i]$ and $A[n]$
8: return i
```

Example:

A = [4, 2, 7, 3, 1, 9, 6, 0, 8]

Choosing our pivot element as pivot = 8 and placing our initial pointers i and j (the arrow above the array is the for the i pointer whereas the arrow below the array is for the j pointer), we get

4	2	7	3	1	9	6	0	8

Now we have to simply follow the algorithm for the partition. The elements at the jth pointer will be compared to the pivot element continuously throughout the various iterations. Since pivot >= 4 (first element, where the pointers are pointing), we'll move the pointers ahead for the next comparison.

4	2	7	3	1	9	6	0	8

Once again, since pivot >= 2, we move ahead for the next comparison.

4	2	7	3	1	9	6	0	8

Here pivot >= 7 as well, so we keep on moving until we reach 9 element (5th index) of the array. Here, 9 >= pivot, and so we place one of our pointers at this index (5) until we reach another index that contains a value that is lesser than the pivot element or the pivot element itself. Moving ahead, we find that the element at the 6th index (6) is smaller than 9 (where we placed one of our pointers). A swapping operation is performed.

4	2	7	3	1	6	9	0	8

4	2	7	3	1	6	9	0	8

This process is repeated until we reach the pivot element, and so we will further go on and swap the values of 9 and 0, and then finally swap the values of 9 and the pivot element (8) itself. After the first pass, we get the following array:

4	2	7	3	1	6	0	8	9

if we notice carefully, all the elements present to the left of our current pivot element are smaller than the pivot element itself, whereas the elements present to the right of it are larger than the pivot element.
We will now pick 0 as our pivot element from the array, pivot = 0.

4	2	7	3	1	6	0	8	9

4	2	7	3	1	6	0	8	9

0	2	7	3	1	6	4	8	9

0	2	3	1	4	6	7	8	9

0	1	3	2	4	6	7	8	9

0	1	2	3	4	6	7	8	9

- COMPLEXİTY ANALYSİS OF QUICK SORT – LOMUTO PARTITIONING:

Worst-case analysis

The most unbalanced partition occurs when one of the sublists returned by the partitioning routine is of size n – 1. This may occur if the pivot happens to be the smallest or largest element in the list, or in some implementations when all the elements are equal.

If this happens repeatedly in every partition, then each recursive call processes a list of size one less than the previous list. Consequently, we can make n − 1 nested calls before we reach a list of size 1. This means that the call tree is a linear chain of n − 1 nested calls. The ith call does O(n − i) work to do the partition, and $\sum_{i=0}^{n}(n-i) = O(n^2)$so in that case quicksort takes O(n^2) time.

Best-case analysis

In the most balanced case, each time we perform a partition we divide the list into two nearly equal pieces. This means each recursive call processes a list of half the size. Consequently, we can make only logn nested calls before we reach a list of size 1. This means that the depth of the call tree is logn. But no two calls at the same level of the call tree process the same part of the original list; thus, each level of calls needs only O(n) time all together. The result is that the algorithm uses only O(n log n) time.

Average-case analysis

To sort an array of n distinct elements, quicksort takes O(n log n) time in expectation, averaged over all n! permutations of n elements with equal probability. Alternatively, if the algorithm selects the pivot uniformly at random from the input array, the same analysis can be used to bound the expected running time for any input sequence; the expectation is then take over the random choices made by the algorithm.

Using recurrences

An alternative approach is to set up a recurrence relation for the $T(n)$ factor, the time needed to sort a list of size n. In the most unbalanced case, a single quicksort call involves $O(n)$ work plus two recursive calls on lists of size 0 and $n-1$, so the recurrence relation is

$$T(n) = O(n) + T(0) + T(n-1) = O(n) + T(n-1).$$

$$T(n) = O(n) + 2T\left(\frac{n}{2}\right)$$

The master theorem for divide-and-conquer recurrences tells us that T(n) = O(n log n).

Solving the recurrence relation:

$$
\begin{aligned}
&C(n) = n - 1 + \frac{1}{n}\sum_{i=0}^{n-1}(C(i) + C(n-i-1)) = n - 1 + \frac{2}{n}\sum_{i=0}^{n-1} C(i) \\
&nC(n) = n(n-1) + 2\sum_{i=0}^{n-1} C(i) \\
&nC(n) - (n-1)C(n-1) = n(n-1) - (n-1)(n-2) + 2C(n-1) \\
&nC(n) = (n+1)C(n-1) + 2n - 2 \\
&\frac{C(n)}{n+1} = \frac{C(n-1)}{n} + \frac{2}{n+1} - \frac{2}{n(n+1)} \le \frac{C(n-1)}{n} + \frac{2}{n+1} \\
&\qquad\;\; = \frac{C(n-2)}{n-1} + \frac{2}{n} - \frac{2}{(n-1)n} + \frac{2}{n+1} \le \frac{C(n-2)}{n-1} + \frac{2}{n} + \frac{2}{n+1} \\
&\qquad\;\;\vdots \\
&\qquad\;\; = \frac{C(1)}{2} + \sum_{i=2}^{n} \frac{2}{i+1} \le 2\sum_{i=1}^{n-1}\frac{1}{i} \approx 2\int_1^n \frac{1}{x}\mathrm{d}x = 2\ln n
\end{aligned}
$$

Solving the recurrence gives $C(n) = 2n \ln n \approx 1.39n \log_2 n$.

Space complexity

In-place partitioning is used. This unstable partition requires O(1) space.

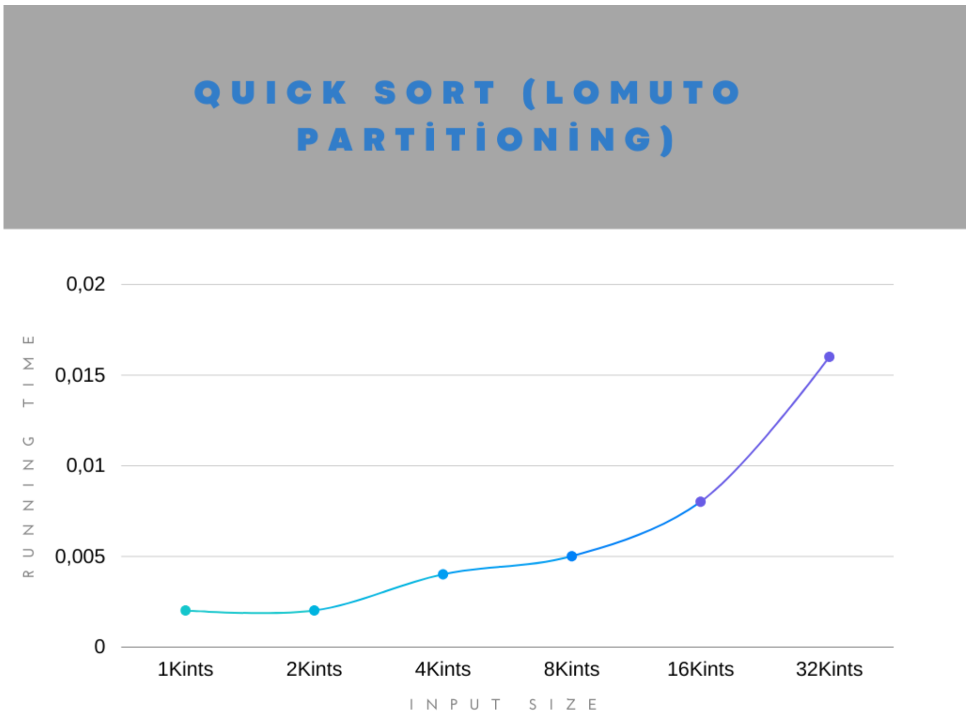
EMPİRİCAL ANALYSİS OF LOMUTO PARTITIONING ALGORTIHM:

data	running time
1Kints	0.002
2K	0.002
4K	0.003
8K	0.005
16K	0.008
32K	0.016
1M	0.127

First, I implemented the Lomuto Partitioning algorithm for different input types and sizes. Then, I noted their running times in the table.

I also try 1Mints and write down to table, but I did not include it into table to keep graph echo trend.

Here is the graph of Input-Running Time of Lomuto Partitioning:



Comments:

According to garph, we can say that Lomuto Part. Algorithm is growing exponentially. We already know that its time complexity about O(n(logn)), so that means our experimental design is close to theoretical results.

Comparing to other sorting algorithms, Lomuto partitioning is more efficient than Selection Sort and Insertion Sort. Merge sort’s worst case complexity is also O(nlog(n)), but Merge Sort uses extra memory At this point, Lomuto Partitioning is the best.

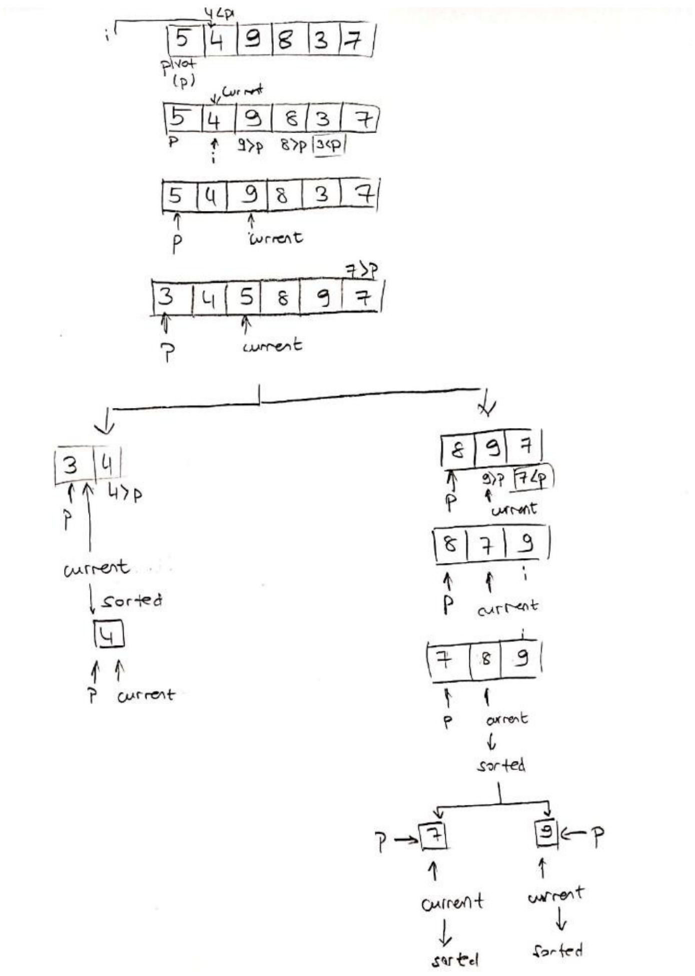
HOARE’S PARTITIONING ALGORITHM:

- PSEUDO-CODE

```
partition(arr[], lo, hi)
   pivot = arr[lo]
   i = lo - 1  // Initialize left index
   j = hi + 1  // Initialize right index // Find a value in left side greater than pivot
   do
      i = i + 1
   while arr[i] < pivot   // Find a value in right side smaller than pivot
   do
      j--;
   while (arr[j] > pivot);
   if i >= j then
      return j
swap arr[i] with arr[j]
```

EXAMPLE:

A=[5, 4, 9, 8, 3, 7]   Pivot=5



- COMPLEXİTY ANALYSİS OF QUICK SORT – HOARE'S PARTITIONING:

We traverse the array exactly once, the time complexity is O(N). This algorithm is said to be almost 3 times faster than Lomuto paritioning.

Lomuto partitioning is also a O(N) time complexity, O(1) space complexity algorithm and does the work in just one array traversal like hoare's partition but Lomuto partition requires more swaps and hence is relatively inefficient in this respect. However, Lomuto partition puts the pivot at the correct position in the array as well as returns the index whereas Hoare's partition only returns the correct index of the pivot.

Space Complexity

The algorithm does not use any auxiliary space, hence space complexity is O(1).

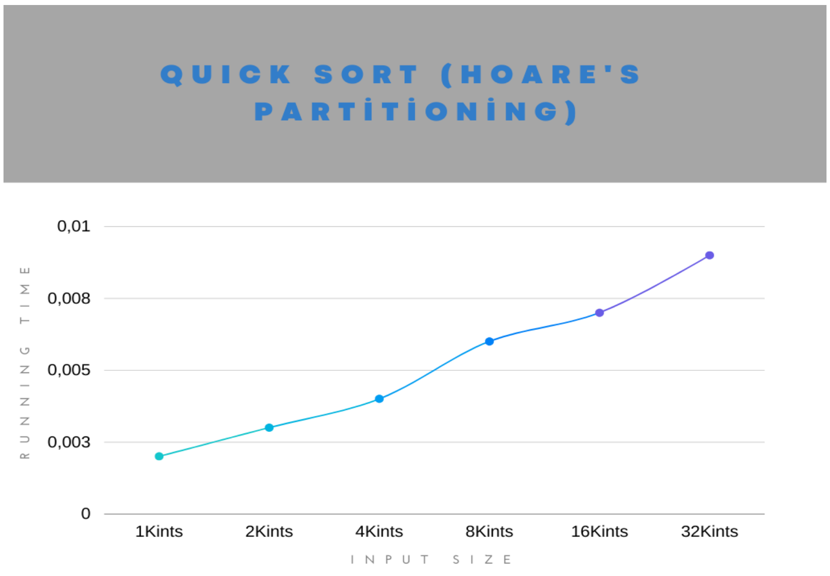
EMPİRİCAL ANALYSİS OF HOARE'S PARTITIONING ALGORTIHM:

Data input	running time
1Kints	0.002
2K	0.003
4K	0.004
8K	0.006
16K	0.007
32K	0.009
1M	0.142

First, I implemented the Hoare's Partitioning algorithm for different input types and sizes. Then, I noted their running times in the table.

I also try 1Mints and write down to table, but I did not include it into table to keep graph echo trend.

Here is the graph of Input-Running Time of Hoare's Partitioning:



Comments:

According to graph Running-time of Hoare's has small numbers. Results show us, Hoare's Part. Algorithm faster than Lomuto Part. Algorithm. In theoretically and emprirically, this is true.

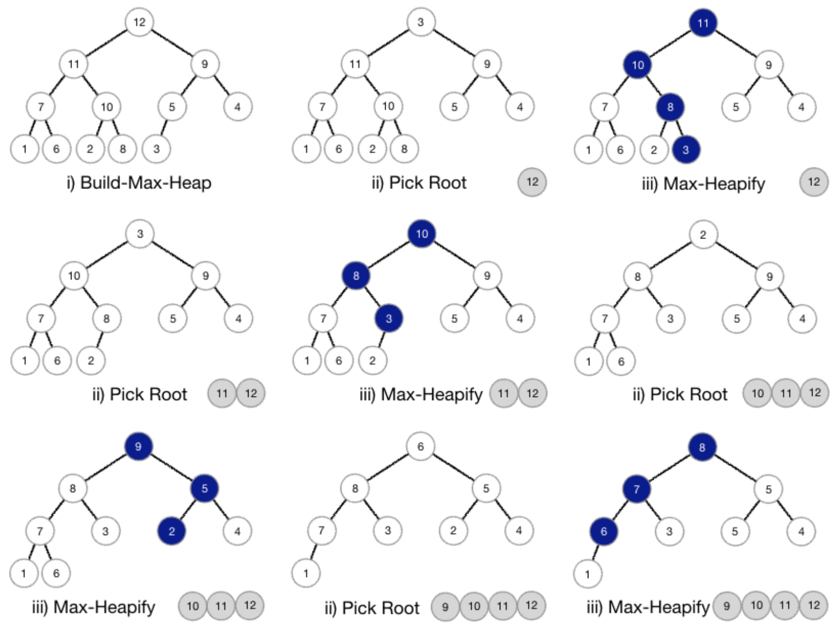
Basically, Insertion, Selection and Merge Sort Algorithms less efficient than Quick Sort Algorithms.

ANALYSIS of HEAP SORT

- PSEUDO-CODE

```
Heapsort(A) {
  BuildHeap(A)
  for i <- length(A) downto 2 {
    exchange A[1] <-> A[i]
    heapsize <- heapsize -1
    Heapify(A, 1)
}
BuildHeap(A) {
  heapsize <- length(A)
  for i <- floor( length/2 ) downto 1
    Heapify(A, i)
}
Heapify(A, i) {
  le <- left(i)
  ri <- right(i)
  if (le<=heapsize) and (A[le]>A[i])
    largest <- le
  else
    largest <- i
  if (ri<=heapsize) and (A[ri]>A[largest])
    largest <- ri
  if (largest != i) {
    exchange A[i] <-> A[largest]
    Heapify(A, largest)
  }
}
```

EXAMPLE:



- COMPLEXİTY ANALYSİS OF HEAP SORT:

We can analyze the cost of Heapsort by examining sub-functions of Max-Heapify and Build-Max-Heap.

The cost of Max-Heapify is *O(lgn)*. Comparing a node and its two children nodes costs *Θ(1)*, and in the worst case, we recurse $\lfloor log_2 n \rfloor$ times to the bottom. Alternatively, the cost of Max-Heapify can be expressed with the height *h* of the heap *O(h)*.

$$\begin{aligned}\sum_{h=0}^{\lfloor lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) &= O\left(n \sum_{h=0}^{\lfloor lg n \rfloor} \frac{h}{2^h} \right) \\ &= O\left(n \sum_{h=0}^{\infty} \frac{h}{2^h} \right) \\ &= O(n)\end{aligned}$$

Thus, the first step of heap sort, which is building a heap out of a randomly arranged array, can be done in O(N).

- Swapping: Swapping the max element with the bottom level right-most element and reducing the heap size can be done in constant time, O(1).
- Reheapification: In the worst case, the new value at the root position will have to be swapped log(N) times to be sent to the bottom of the heap to achieve a MaxHeap once again. So each reheapification after the extraction costs O(logN).

1. Time Complexity

Case	Time Complexity
Best Case	O(n logn)
Average Case	O(n log n)
Worst Case	O(n log n)

Best Case Complexity - It occurs when there is no sorting required, i.e. the array is already sorted. The best-case time complexity of heap sort is O(n logn).

Average Case Complexity - It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending. The average case time complexity of heap sort is O(n log n).

Worst Case Complexity - It occurs when the array elements are required to be sorted in reverse order. That means suppose you have to sort the array elements in ascending order, but its elements are in descending order. The worst-case time complexity of heap sort is O(n log n).

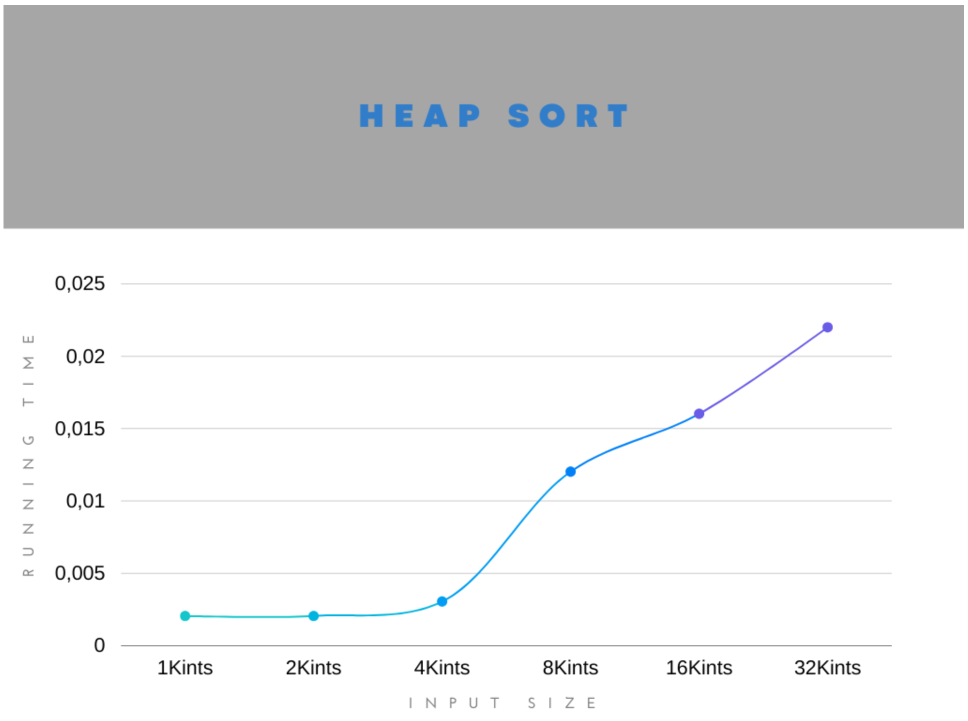
EMPİRİCAL ANALYSİS OF HEAP SORT:

Data input	running time
1Kints	0.003
2K	0.003
4K	0.004
8K	0.007
16K	0.008
32K	0.012
1M	0.291

First, I implemented the Heap Sort algorithm for different input types and sizes. Then, I noted their running times in the table.

I also try 1Mints and write down to table, but I did not include it into table to keep graph echo trend.

Here is the graph of Input-Running Time of Heap Sort:



Comments:

As expected, Heapsort follows other O(nlogn)-cost algorithms, while it is not as inexpensive as Quicksort with a higher constant hidden in O notation.

Although Heapsort does not beat Quicksort as a sorting algorithm, Heap as the data structure offers many different usages, and one of the most notable would be in priority queues.

If we compare Quick Sort and Heap Sort, the most direct competitor of quicksort is heapsort. Heapsort is typically somewhat slower than quicksort, but the worst-case running time is always Θ(nlogn). Quicksort is usually faster, though there remains the chance of worst case performance except in the introsort variant, which switches to heapsort when a bad case is detected. If it is known in advance that heapsort is going to be necessary, using it directly will be faster than waiting for introsort to switch to it.

IMPLEMENTATION of ALL ALGORITHMS:

SELECTION SORT

```
public static int sort(int arr[])
  {
      int n = arr.length;

      // One by one move boundary of unsorted subarray
      for (int i = 0; i < n-1; i++)
      {
          // Find the minimum element in unsorted array
          int min_idx = i;
          for (int j = i+1; j < n; j++)
              if (arr[j] < arr[min_idx])
                  min_idx = j;
          // Swap the found minimum element with the first
          // element
          int temp = arr[min_idx];
          arr[min_idx] = arr[i];
          arr[i] = temp;
      }
      return 0;
}
```

INSERTION SORT

```
/*Function to sort array using insertion sort*/
        public static int sort(int arr[])
        {
                int n = arr.length;
                for (int i = 1; i < n; ++i) {
                        int key = arr[i];
                        int j = i - 1;
                        /* Move elements of arr[0..i-1], that are
                        greater than key, to one position ahead
                        of their current position */
                        while (j >= 0 && arr[j] > key) {
                                arr[j + 1] = arr[j];
                                j = j - 1;
                        }
                        arr[j + 1] = key;
                }
        return 0;
        }
```

MERGE SORT

```
public static int merge(int arr[], int l, int m, int r)
    {
        // Find sizes of two subarrays to be merged
        int n1 = m - l + 1;
        int n2 = r - m;
        /* Create temp arrays */
        int L[] = new int[n1];
        int R[] = new int[n2];

        /*Copy data to temp arrays*/
        for (int i = 0; i < n1; ++i)
            L[i] = arr[l + i];
        for (int j = 0; j < n2; ++j)
            R[j] = arr[m + 1 + j];

        /* Merge the temp arrays */

        // Initial indexes of first and second subarrays
        int i = 0, j = 0;

        // Initial index of merged subarray array
        int k = l;
        while (i < n1 && j < n2) {
            if (L[i] <= R[j]) {
                arr[k] = L[i];
                i++;
            }
            else {
                arr[k] = R[j];
                j++;
            }
            k++;
        }

        /* Copy remaining elements of L[] if any */
        while (i < n1) {
            arr[k] = L[i];
            i++;
            k++;
        }
        /* Copy remaining elements of R[] if any */
        while (j < n2) {
            arr[k] = R[j];
            j++;
            k++;
        }
            return 0;
    }
    // Main function that sorts arr[l..r] using
    // merge()
    public static int sort(int arr[], int l, int r)
    {
        if (l < r) {
            // Find the middle point
            int m = l + (r - l) / 2;
            // Sort first and second halves
            sort(arr, l, m);
            sort(arr, m + 1, r);
            // Merge the sorted halves
            merge(arr, l, m, r);
        }
            return 0;
    }
```

QUICK SORT – LOMUTO PARTITIONING

```
public static int partition(int []arr, int low,
                                int high)
{
    int pivot = arr[high];

    // Index of smaller element
    int i = (low - 1);

    for (int j = low; j <= high- 1; j++)
    {
        // If current element is smaller
        // than or equal to pivot
        if (arr[j] <= pivot)
        {
            i++; // increment index of
                 // smaller element
            Swap(arr, i, j);
        }
    }
    Swap(arr, i + 1, high);
    return (i + 1);
}

/* The main function that
   implements QuickSort
arr[] --> Array to be sorted,
low --> Starting index,
high --> Ending index */
public static int quickSort(int []arr, int low,
                                int high)
{
    if (low < high)
    {
        /* pi is partitioning index,
        arr[p] is now at right place */
        int pi = partition(arr, low, high);

        // Separately sort elements before
        // partition and after partition
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
        return 0;
}
```

QUICK SORT – HOARE'S PARTITIONING

```
public static int partition(int[] arr, int low, int high)
    {
        int pivot = arr[low];
        int i = low - 1, j = high + 1;

        while (true) {
            // Find leftmost element greater
            // than or equal to pivot
            do {
                i++;
            } while (arr[i] < pivot);

            // Find rightmost element smaller
            // than or equal to pivot
            do {
                j--;
            } while (arr[j] > pivot);
            // If two pointers met.
            if (i >= j)
                return j;
            int temp = arr[i];
            arr[i] = arr[j];
            arr[j] = temp;
            // swap(arr[i], arr[j]);
        }
    }

    /* The main function that
       implements QuickSort
    arr[] --> Array to be sorted,
    low --> Starting index,
    high --> Ending index */
    public static int quickSort(int[] arr, int low, int high)
    {
        if (low < high) {
            /* pi is partitioning index,
            arr[p] is now at right place */
            int pi = partition(arr, low, high);

            // Separately sort elements before
            // partition and after partition
            quickSort(arr, low, pi);
            quickSort(arr, pi + 1, high);
        }
        return 0;
    }
```

HEAP SORT

```
public static int sort(int arr[])
{
    int N = arr.length;

    // Build heap (rearrange array)
    for (int i = N / 2 - 1; i >= 0; i--)
        heapify(arr, N, i);

    // One by one extract an element from heap
    for (int i = N - 1; i > 0; i--) {
        // Move current root to end
        int temp = arr[0];
        arr[0] = arr[i];
        arr[i] = temp;

        // call max heapify on the reduced heap
        heapify(arr, i, 0);
    }
    return 0;
}

// To heapify a subtree rooted with node i which is
// an index in arr[]. n is size of heap
public static int heapify(int arr[], int N, int i)
{
    int largest = i; // Initialize largest as root
    int l = 2 * i + 1; // left = 2*i + 1
    int r = 2 * i + 2; // right = 2*i + 2

    // If left child is larger than root
    if (l < N && arr[l] > arr[largest])
```

```
        largest = l;

    // If right child is larger than largest so far
    if (r < N && arr[r] > arr[largest])
        largest = r;

    // If largest is not root
    if (largest != i) {
        int swap = arr[i];
        arr[i] = arr[largest];
        arr[largest] = swap;

        // Recursively heapify the affected sub-tree
        heapify(arr, N, largest);
    }
    return 0;
}
```