BLM19307E Algorithm Analysis & Design- Project 1

In this project, you are expected to design an experimental study for the comparison of the following sorting algorithms: Selection Sort, Insertion Sort, Merge Sort, Quick Sort with Lomuto & Hoare partitioning, and Heapsort. The tests were performed on 1024, 2048, 4096, 8192, 16384, 32768, 65536, 131072, 262144, 524288 dimensional ascending, descending and randomly generated arrays.
The results of the tests were transferred to excels tables and compared with each other.

Implementation:

Implementation of all algorithms is written in the source code. These algorithms are:

1. Selection Sort
2. Insertion Sort
3. Merge Sort
4. Quick Sort with Lomuto
5. Quick Sort with Hoare partitioning
6. Heapsort.

Pseudo-code of the Algorithms:

- Selection Sort

```
selectionSort(arr)
  n = length of arr
  for i = 0 to n - 1
    minIndex = i
    for j = i + 1 to n
      if arr[j] < arr[minIndex]
        minIndex = j
    swap arr[i] and arr[minIndex]
  return arr
```

- Insertion Sort

```
insertionSort(arr)
  n = length of arr
  for i = 1 to n
    value = arr[i]
    j = i - 1
    while j >= 0 and arr[j] > value
      arr[j+1] = arr[j]
      j = j - 1
    arr[j+1] = value
  return arr
```

- Merge Sort

```
mergeSort(arr)
  if length of arr <= 1
    return arr
  else
    middle = length of arr / 2
    left = arr[0...middle]
    right = arr[middle...end]
    left = mergeSort(left)
    right = mergeSort(right)
    return merge(left, right)

merge(left, right)
  result = []
  while length of left > 0 and length of right > 0
    if first element of left <= first element of right
      append first element of left to result
      left = left without its first element
    else
      append first element of right to result
      right = right without its first element
  if length of left > 0
    append remaining elements of left to result
  else
    append remaining elements of right to result
  return result
```

- Quick Sort with Lomuto

```
function quicksort(array, low, high)
    if low < high
        pivot_index = lomuto_partition(array, low, high)
        quicksort(array, low, pivot_index - 1)
        quicksort(array, pivot_index + 1, high)

function lomuto_partition(array, low, high)
    pivot = array[high]
    i = low

    for j = low to high - 1
        if array[j] <= pivot
            swap(array[i], array[j])
            i = i + 1

    swap(array[i], array[high])
    return i
```

- Quick Sort with Hoare partitioning

```
function quicksort(array, low, high)
    if low < high
        pivot_index = hoare_partition(array, low, high)
        quicksort(array, low, pivot_index)
        quicksort(array, pivot_index + 1, high)

function hoare_partition(array, low, high)
    pivot = array[low]
    i = low - 1
    j = high + 1
```

```
while True
   do
      i = i + 1
   while array[i] < pivot

   do
      j = j - 1
   while array[j] > pivot

   if i >= j
      return j

   swap(array[i], array[j])
```

- Heapsort.

```
heapSort(arr)
  buildHeap(arr)
  for i = length of arr to 2
    swap arr[1] and arr[i]
    heapSize = heapSize - 1
    heapify(arr, 1, heapSize)
  return arr

buildHeap(arr)
  heapSize = length of arr
  for i = floor(heapSize / 2) to 1
    heapify(arr, i, heapSize)

heapify(arr, i, heapSize)
  left = 2 * i
  right = 2 * i + 1
  largest = i
  if left <= heapSize and arr[left] > arr[largest]
    largest = left
  if right <= heapSize and arr[right] > arr[largest]
    largest = right
  if largest != i
    swap arr[i] and arr[largest]
    heapify(arr, largest, heapSize)
```

Experimental Design:

- **Selection Sort**

The time complexity of the selection sort depends upon the iteration and then finding the minimum element in each iteration. In each iteration, we find the minimum element from the sub-array and bring it to the right position using swapping.

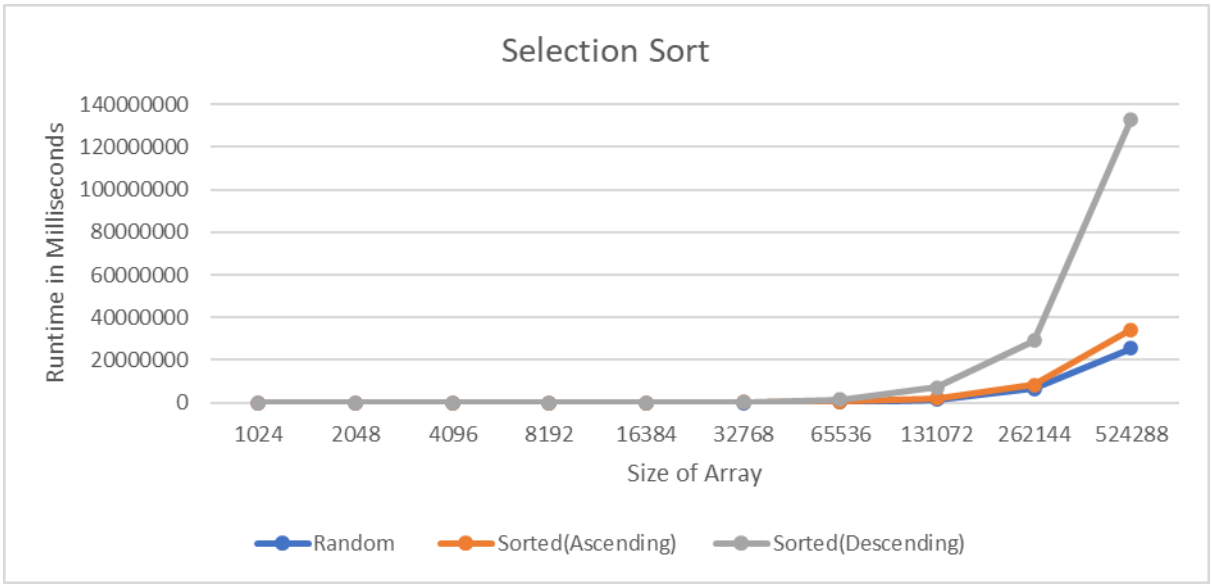
The best case is when the array is already sorted. So even if the array is already sorted, we will traverse the entire array for checking, and in each iteration or traversal, we will perform the searching operation. Only the swapping will not be performed as the elements will be at the correct position. Since the swapping only takes a constant amount of time O (1). The best time complexity of selection sort comes out to be O(N^2)

The worst case is when the array is completely unsorted or sorted in descending order. So, we will traverse the entire array for checking, and in each iteration, we will perform the searching operation. After searching, we will swap the element at its correct position. As we know that the swapping only takes a constant amount of time O (1). So, the worst time complexity of selection sort also comes out to be O(N^2).

The average case is when some part of the array is sorted. So, as we have seen earlier in the best and worst cases, we will need to perform the searching, and swapping in each iteration. So, the average time complexity of the selection sort is O(N^2) as well.

Time complexity:

$$\sum_{i=0}^{n-2}\sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2}(n-1) - (i+1) + 1 = \sum_{i=0}^{n-2}(n-1) - \sum_{i=0}^{n-2} i = \frac{(n-1)\,n}{2} \in O(n^2)$$



- **Insertion Sort**

One of the simplest sorting methods is insertion sort, which involves building up a sorted list one element at a time. By inserting each unexamined element into the sorted list between elements that are less than it and greater than it.

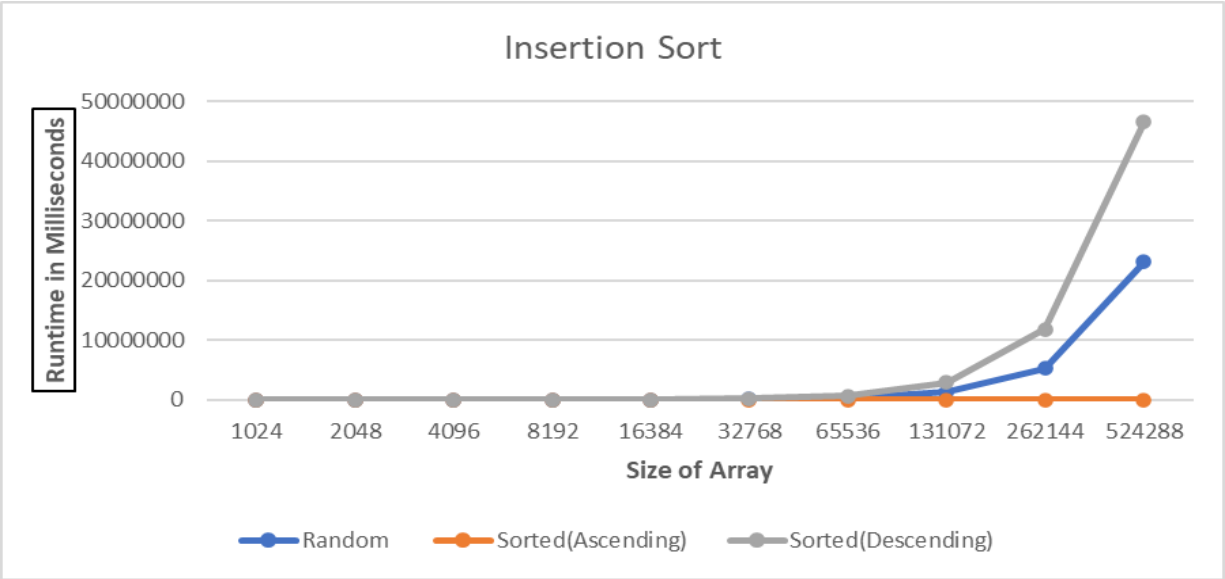
The worst-case (and average-case) complexity of the insertion sort algorithm is $O(n^2)$. Meaning that, in the worst case, the time taken to sort a list is proportional to the square of the number of elements in the list.

The best-case time complexity of the insertion sort algorithm is O(n) time complexity. Meaning that the time taken to sort a list is proportional to the number of elements in the list; this is the case when the list is already in the correct order. There's only one iteration in this case since the inner loop operation is trivial when the list is already in order.

Insertion sort is frequently used to arrange small lists. On the other hand, Insertion sort isn't the most efficient method for handling large lists with numerous elements. Notably, the insertion sort algorithm is preferred when working with a linked list. And although the algorithm can be applied to data structured in an array, other sorting algorithms such as quicksort.

Time complexity:

$$\sum_{j=1}^{n-1} j = \frac{(n-1)\,((n-1)+1)}{2} = \frac{(n-1)\,n}{2} \in O(n^2)$$



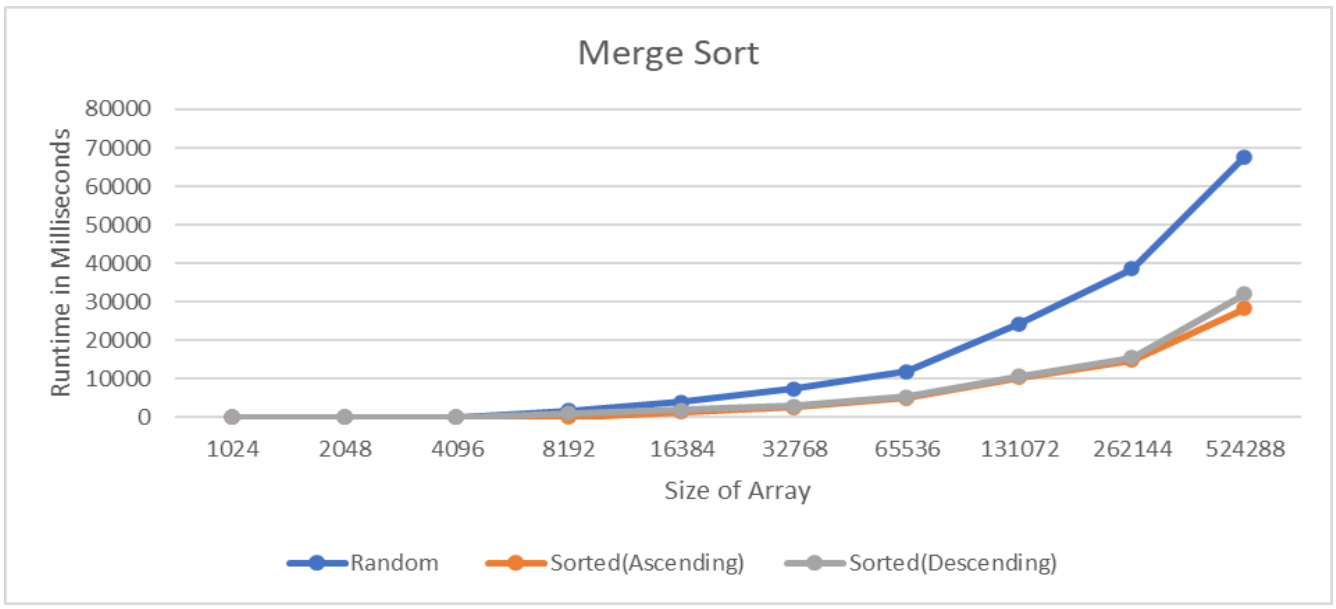
- **Merge Sort**

The Merge Sort algorithm is a sorting algorithm that is based on the Divide and Conquer paradigm. In this algorithm, the array is initially divided into two equal halves and then they are combined in a sorted manner.

Sorting arrays on different machines. Merge Sort is a recursive algorithm and time complexity can be expressed as following recurrence relation.

$T(n) = 2T(n/2) + \theta(n)$

The above recurrence can be solved using the Master method. It falls in case II of the Master Method and the solution of the recurrence is $\theta(N\log(N))$. The time complexity of Merge Sort is $\theta(N\log(N))$ in all 3 cases (worst, average, and best) as merge sort always divides the array into two halves and takes linear time to merge two halves.



- **Quick Sort with Lomuto partitioning**

This algorithm works by assuming the pivot element as the last element. If any other element is given as a pivot element, then swap it first with the last element. Then, initialize two variables i as low and j also low, iterate over the array and increment i when arr[j] <= pivot and swap arr[i] with arr[j] otherwise increment only j. After coming out from the loop swap arr[i] with arr[hi]. This i stores the pivot element.

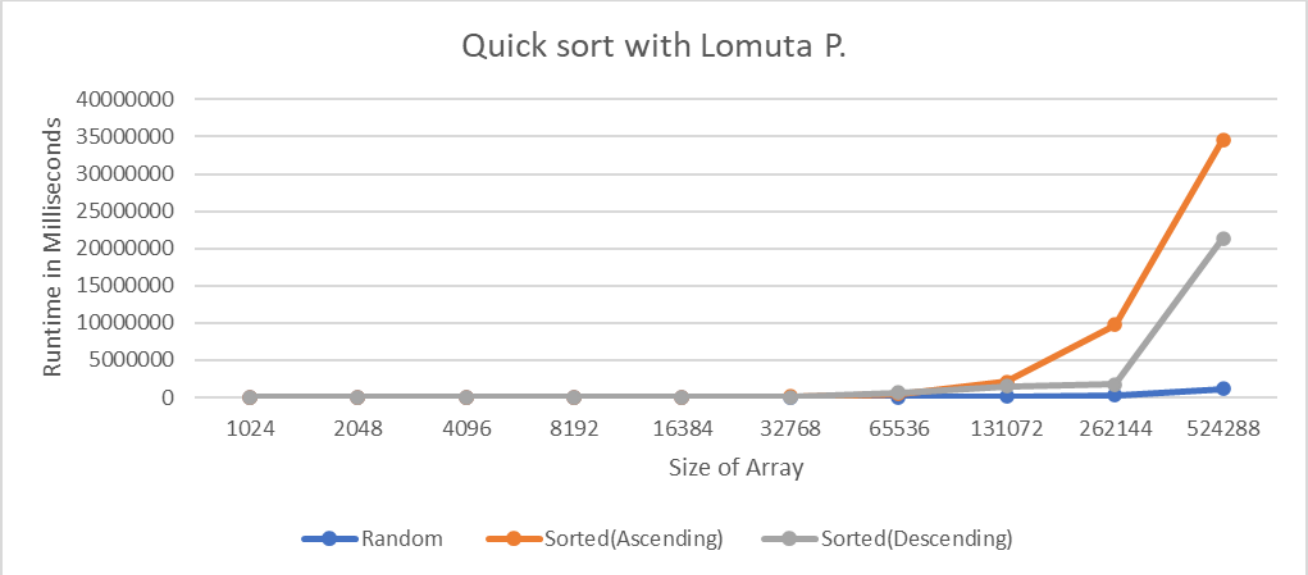
Generally, a random element of the array is located and picked and then exchanged with the first or the last element to give initial pivot values. In the aforementioned algorithm, the last element of the list is considered as the initial pivot element.It is a linear algorithm.
It is slower than the Hoare partitioning. It fixes the pivot element in the correct position.

Time complexity:

Time complexity of quicksort with using Lomuto's partitioning can be calculated using the Master Theorem as follows:

$$T(n) = T(k) + T(n-k-1) + O(n)$$

Where T(n) represents the time complexity of the algorithm when sorting a list of size n, n is the size of the list, k is the number of elements that are less than the pivot element, and O(n) represents the time it takes to partition the list around the pivot element. We have a = 2, b = 2, and d = 1, so a = b d by using the Master Theorem.
Therefore, the time complexity of quick sort using Lomuto's partitioning is: T(n) = O(n log n)



- **Quick Sort with Hoare partitioning**

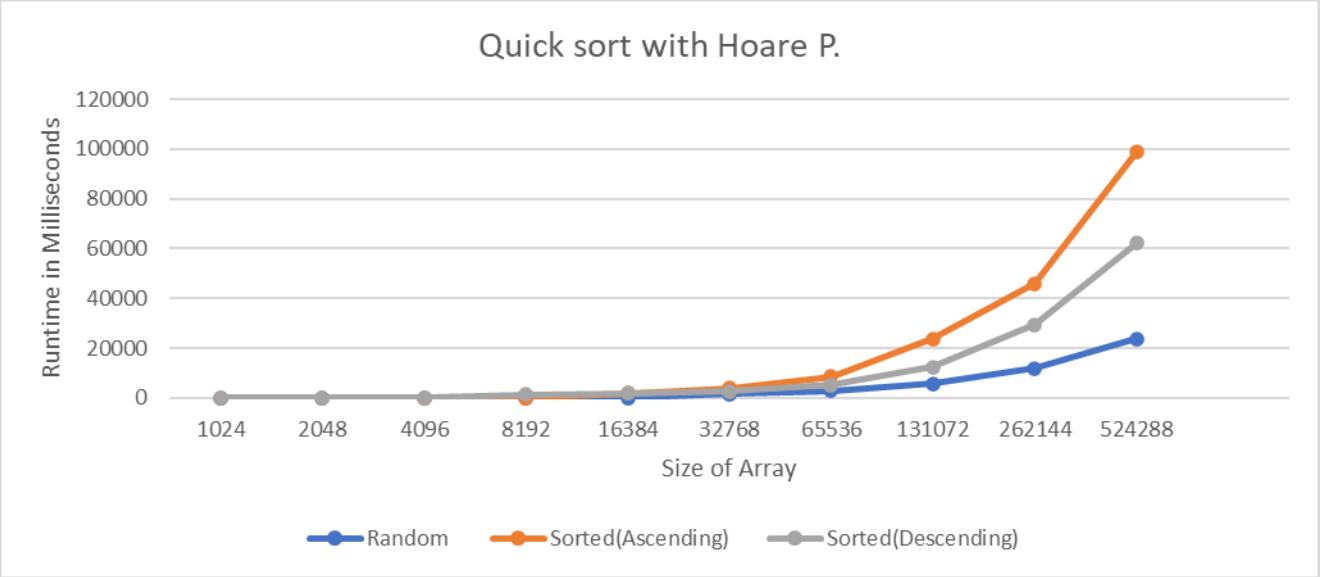
Hoare partitioning was proposed by Tony Hoare when the Quicksort algorithm was originally published. Instead of working across the array from low to high, it iterates from both ends at once towards the center. This means that we have more iterations, and more comparisons, but fewer swaps.

Generally, the first item or the element is assumed to be the initial pivot element. Some choose the middle element and even the last element. It is also a linear algorithm.
It is relatively faster than Lomuto partitioning. It doesn't fix the pivot element in the correct position.

Time complexity:
Time complexity of quick sort with Hoare's partitioning can be calculated using the Master Theorem as follows:

$$T(n) = T(k) + T(n-k) + O(n)$$

Where T(n) represents the time complexity of the algorithm when sorting a list of size n, n is the size of the list, k is the number of elements that are less than the pivot element, and O(n) represents the time it takes to partition the list around the pivot element. We have a = 2, b = 2, and d = 1, so a = b d by using the Master Theorem.
Therefore, the time complexity of quick sort using Hoare's partitioning is: T(n) = O(n log n)



Comparison:

Hoare's scheme is more efficient than Lomuto's partition scheme because it does three times fewer swaps on average, and it creates efficient partitions even when all values are equal.
Like Lomuto's partition scheme, Hoare partitioning also causes Quick sort to degrade to O(n^2) when the input array is already sorted, it also doesn't produce a stable sort.

- **Heapsort.**

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

Heap sort is an in-place algorithm. Its typical implementation is not stable but can be made stable.
Typically, 2-3 times slower than well-implemented QuickSort. The reason for slowness is a lack of locality of reference.

Time-complexity for buildMaxHeap() is O(n).Time complexity for each instance of heapify() is O(log n).
Heap Sort has O(nlog n) time complexities for all the cases (best case, average case, and worst case).

Some Advantages of heapsort:

- Efficiency – The time required to perform Heap sort increases logarithmically while other algorithms may grow exponentially slower as the number of items to sort increases. This sorting algorithm is very efficient.
- Memory Usage – Memory usage is minimal because apart from what is necessary to hold the initial list of items to be sorted, it needs no additional memory space to work
- Simplicity – It is simpler to understand than other equally efficient sorting algorithms because it does not use advanced computer science concepts such as recursion

Time complexity:
The time complexity of heap sort can be calculated using the Master Theorem as follows:

$$T(n) = T(n-1) + O(\log n)$$

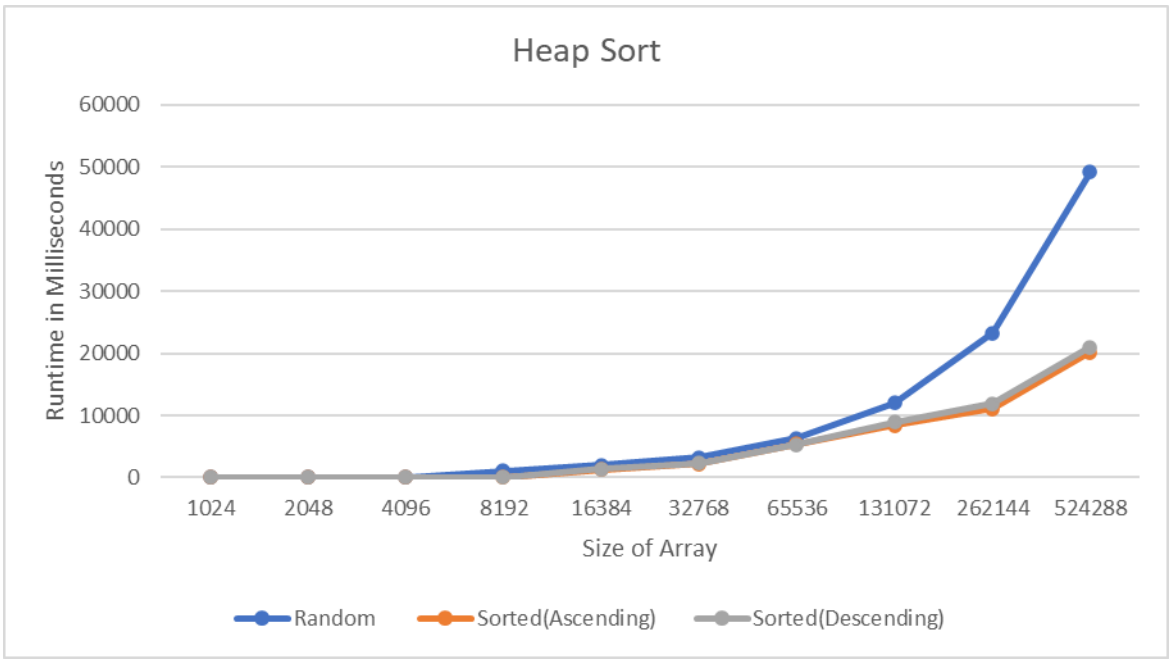
Where T(n) represents the time complexity of the algorithm when sorting a list of size n, n is the size of the list, and O(log n) represents the time it takes to remove an element from the heap. Using the Master Theorem, we have a = 1, b = 2, and d = 0, so a < b d .

Therefore, the time complexity of heap sort using the Master Theorem is:

$$T(n) = O(n)$$

It means that the time it takes for the algorithm to sort a list of size n increases linearly with the size of the input list. However, this calculation does not take into account the time it takes to build the heap, which has a time complexity of O(n log n).

Therefore, the actual time complexity of heap sort is O(n log n), which is the same as the time complexity calculated using the previous method. This makes heap sort a relatively efficient sorting algorithm, especially for large lists.



We can see clearly:

- When doubling the input quantity, sorting takes a little more than twice as long; this corresponds to the expected quasilinear runtime O(n log n).
- For presorted input data, Heapsort is about three times faster than for unsorted data.
- Input data sorted in ascending order will be sorted about as fast as input data sorted in descending order

Results and Discussion:

1. Selection Sort is an easy-to-implement, and in its typical implementation unstable, sorting algorithm with an average, best-case, and worst-case time complexity of $O(n^2)$. Selection Sort is slower than Insertion Sort, which is why it is rarely used in practice.

2. Insertion Sort is an easy-to-implement, stable sorting algorithm with time complexity of $O(n^2)$ in the average and worst case, and O(n) in the best case. For very small n, Insertion Sort is faster than more efficient algorithms such as Quicksort or Merge Sort.

3. Merge Sort is an efficient, stable sorting algorithm with an average, best-case, and worst-case time complexity of O(n log n).Merge Sort has an additional space complexity of O(n) in its standard implementation. This can be circumvented by in-place merging, which is either very complicated or severely degrades the algorithm's time complexity.

4. For Merge sort:

- If the input data is sorted in descending order, there are only about half as many comparisons in phase 1 as there are for unsorted or ascending data; there are also no swap operations. This is because a descending sorted array already corresponds to a max heap.
- Input data sorted in ascending order correspond to a min heap. The tree must be completely reversed in the buildHeap() phase, so in this case, we have about a third more swap operations than with randomly arranged data, in which the heap condition is already fulfilled on some subtrees.

Yunus SÜMER
1921221010

References:

https://www.geeksforgeeks.org/sorting-algorithms/

https://www.javatpoint.com/sorting-algorithms

https://stackoverflow.com/