Counting number of swap operation in quick-sort, heap-sort and bubble-sort.

Priyal Gupta(IHM2017004), Aditya Gupta(IHM2017005),
Pratham Prakash Gupta(IHM2017003) and Rohit Ranjan(IHM2017001)

Abstract: This paper counts the number of swapping operation while sorting an array of 1000 positive integers using quick sort, heap sort, and bubble sort. There can be many algorithm to solve the above problem but we have to choose the one which has least time complexity. For each of the sorting algorithm a variable is declared which is incremented at every swapping operation.

Keywords: Bubble sort, Heap Sort, Quick Sort.

I. Introduction

We have to find the number of swapping operation while sorting an array of 1000 positive integers using quick sort, heap sort, and bubble sort. There can be many algorithm to solve the above problem but we have to choose the one which has least time complexity. For this we need to know about the sorting algorithms.

- **Bubble Sort:** Bubble sort is a sorting algorithm that works by repeatedly stepping through lists that need to be sorted, comparing each pair of adjacent items and swapping them if they are in the wrong order. This passing procedure is repeated until no swaps are required, indicating that the list is sorted. Bubble sort gets its name because smaller elements bubble toward the top of the list.
- **Heap Sort:** Heap sort is a comparison-based sorting algorithm. It divides its input into a sorted and an unsorted region, and it iteratively shrinks the unsorted region by extracting the largest element and moving that to the sorted region. It uses a heap data structure rather than a linear-time search to find the maximum.
- **Quick Sort:** Quicksort is a popular sorting algorithm that is often faster in practice compared to other sorting algorithms. It utilizes a divide-and-conquer strategy to quickly sort data items by dividing a large array into two smaller arrays.

II. Idea

We have to find the number of swapping operation while sorting an array of 1000 positive integers using quick sort, heap sort, and bubble sort. For this we first need to write all the sorting algorithms. Then we have to initialize a variable named count to zero. Then this variable count is incremented by 1, each time a swapping operation is performed.

III. Algorithm Design

A. Algorithm 1:Bubble Sort

```
BubbleSort(array[],count)
swaps← $0$
for i← $0$ $to$ $count-1$ do
    for j← $0$ $to$ $count-1-i$ do
        if(array[j]≥ $array[j+1]$)then
            temp← $array[j+1]$
            array[j+1]← $array[j]$
            array[j]← $temp$
            swaps← $swaps+1$
return swaps
```

B. Heap Sort

```
swap(int* a, int* b)
t← $*a$
*a← $*b$
*b← $t$
```

```
Heapify(arr[],n,i)
  largest← $i$
  l← $2*i+1$
  r← $2*i+2$
```

```
if(l$\leq n$ and $arr[l] \geq arr[largest]$)then
    largest$\leftarrow l$
if(r$\leq n$ and $arr[r] \geq arr[largest]$)then
    largest$\leftarrow r$
if(largest != i) then
    swap(arr[i], arr[largest])
    count$\leftarrow count + 1$
    heapify(arr, n, largest)
```

```
HeapSort(arr[],n)
for    i$\leftarrow n/2 - 1$ to 0 do
    heapify(arr,n,i)
for    i$\leftarrow n - 1$ to 0 do
    swap(arr[0], arr[i])
    count$\leftarrow count + 1$
    heapify(arr, i, 0)
```

C. Quick Sort

```
swap(int* a, int* b)
t$\leftarrow *a$
*a$\leftarrow *b$
*b$\leftarrow t$
```

```
partition (arr[], low, high)
pivot$\leftarrow arr[high]$
i$\leftarrow (low - 1)$
for    j$\leftarrow low$
to   high-1   do
    if(arr[j]$\leq pivot$)then
        i$\leftarrow i + 1$
        swap(arr[i], arr[j])
        count$\leftarrow count + 1$
swap(arr[i + 1], arr[high])
count$\leftarrow count + 1$
return i+1
```

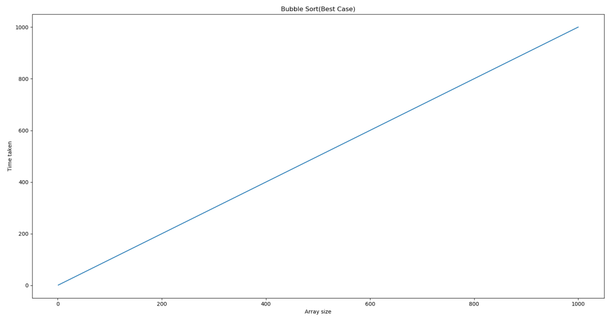
```
quickSort(arr[], low, high)
if(low$\leq high$)then
    pi$\leftarrow partition(arr, low, high)$
    quickSort(arr, low, pi - 1)
    quickSort(arr, pi + 1, high)
```

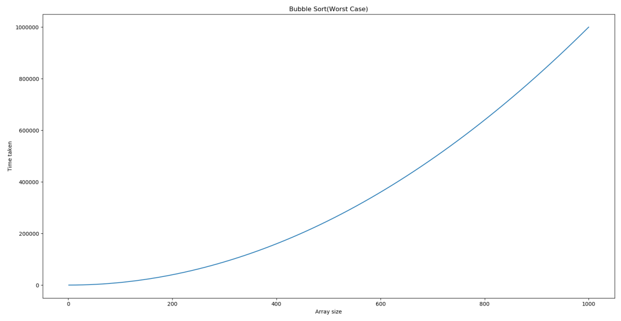
IV. Time Complexity

A. Bubble Sort

In Bubble Sort, n-1 comparisons will be done in the 1st pass, n-2 in 2nd pass, n-3 in 3rd pass and so on. So the total number of comparisons will be:
(n-1) + (n-2) + (n-3) + ... + 3 + 2 + 1
Sum = n(n-1)/2 i.e O(n*n).

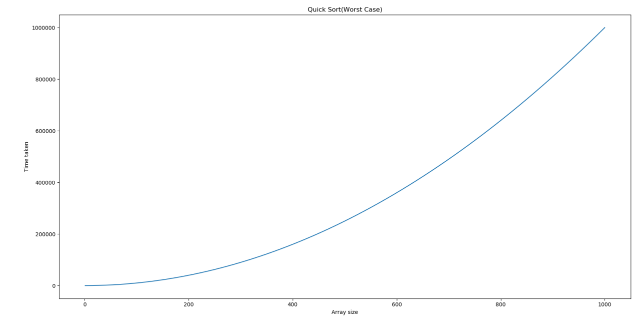
- **Best Case:** Best case time complexity will be O(n), it is when the list is already sorted.
- **Worst Case:** For both worst case and average case time complexity is O(n^2).

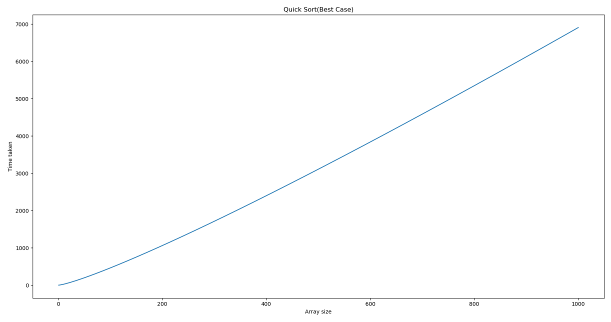




B. Heap Sort

Time complexity of heapify is O(Logn). Time complexity of heapsort() is O(n). And overall time complexity of Heap Sort algorithm is O(nLogn). Here, both the worst case and the best case time complexity is O(nLogn).



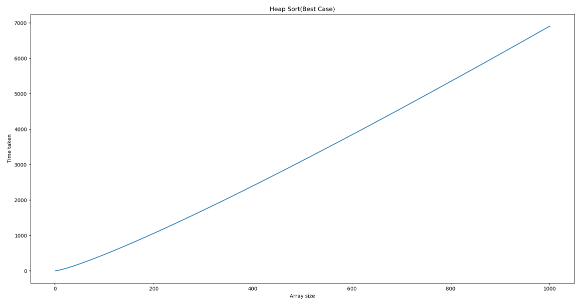


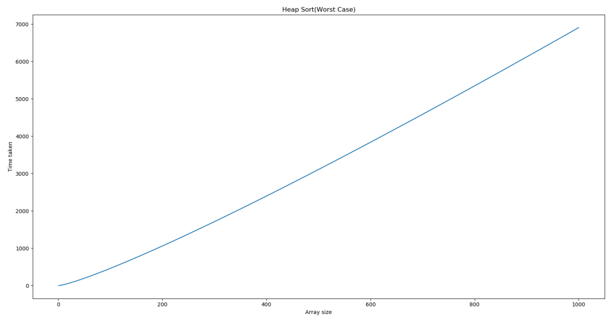
C. *Quick Sort*

Time complexity is the computational complexity that describes the amount of time it takesto run an algorithm. In this case time complexity is O(nLogn).

- **Best Case:** Best case occurs when the partition process always picks the middle element as pivot.
- **Worst Case:** It occurs when the partition process always picks greatest or smallest element as pivot. If we consider above partition strategy where last element is always picked as pivot, the worst case would occur when the array is already sorted in increasing or decreasing order. Its complexity is O(n^2).

.





V. Space Complexity

A. *Bubble Sort*

The space complexity for Bubble Sort is O(1), because only a single additional memory space is required i.e. for temp variable.

B. *Heap Sort*

Since no data actually needs to be stored anywhere else, except maybe during the swap step. Only O(1) additional space is required because the heap is built inside the array to be sorted.

C. *Quick Sort*

Since Quicksort by definition involves recursion, you must consider the space that will be taken up by the call stack. In the worst case, quicksort will be called one time for every element of the list since a solution could recurse all the way down to single element lists. So, there must be a call stack entry for every one of these function calls. This means that with a list of n integers, at worst there will be n stack entries created, hence O(n) space complexity

VI. Experimental Setup

Language Used :- C++
Plotting tool :- matplotlib
Report Making Tool :- TextPortable(Latex)

VII. Conclusion

For finding the number of swapping operation while sorting an array of 1000 positive integers using quick sort, heap sort, and bubble sort, we first need to write all the sorting algorithms. Then we have to initialize a variable named count to zero. Then this variable count is incremented by 1, each time a swapping operation is performed. In this way we can count the total number of swapping operations in a sorting algorithm.

VIII. References

Latex :- https://www.sharelatex.com/learn
Gnu plot :- https://matplotlib.org
Concept :-

- https://www.geeksforgeeks.org/bubble-sort/
- https://www.geeksforgeeks.org/heap-sort/
- https://www.geeksforgeeks.org/quick-sort/