**DOKUZ EYLUL UNIVERSITY**

**ENGINEERING FACULTY**

**DEPARTMENT OF COMPUTER ENGINEERING**

# CME 2204 Assignment-1

## Comparison of Heapsort, Quicksort and Introsort

Oğuzhan YARDIMCI

2015510066

oguzhan.yardimci@ceng.deu.edu.tr

Lecturers

Asts. Prof. Zerrin Işık

Res.Asst. Altuğ Yiğit

7.04.2019

**Comparison of Heapsort, Quicksort and Introsort** 1

**Summary** 3

**HEAPSORT** 3

Heap Sort Algorithm for sorting in increasing order: 4

Max-heaps: 4

Max-Heapify(): 4

Analysis of HeapSort Algorithm: 4

Max-Heapify Analysis: 4

**QUICKSORT** 5

Partition(): 5

QuickSort(): 6

Analysis of QuickSort Algorithm: 6

**INTROSORT** 7

Introsort steps: 7

Pseudo code: 7

Analysis of IntroSort Algorithm: 7

**COMPARE RUNNING TIME** 8

**Results** 9

**Discussion** 9

**Referanslar** 10

# Summary

The aim of the report is to explain the sorting algorithms of Heap Sort, Quick Sort (three different methods), dual pivot QuickSort, and IntroSort. Theoretically, the best case, the average case, the worst case is being evaluated. Then, this algorithm was tested and evaluated in a series of 1000, 10000, 100000 in java programming language. Finally, by comparing and evaluating all the data collected, it is determined which algorithm is more efficient in a situation encountered in real life.

# HEAPSORT

Heap sort is a comparison based sorting technique based on Binary Heap data structure. It is similar to selection sort where we first find the maximum element and place the maximum element at the end. We repeat the same process for remaining element.

### Heap Sort Algorithm for sorting in increasing order:

1. Build a max heap from the input data.
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 1. Finally, heapify the root of tree.
3. Repeat above steps while size of heap is greater than 1.

**Max-heaps:**

1. For max-heaps (largest element at root), max-heap property: for all nodes i, excluding the root, A[PARENT(i)] ≥A[i]. Max

**Max-Heapify():**

1. Node 2 violates max-heap property.
2. Compare node 2 with its children, and then swap it with the larger of the two children
3. Continue swapping until the value is properly placed at the root of a subtree that is a max-heap

### Analysis of HeapSort Algorithm:

HeapSort takes O(n log n) time to extract each of the maximum elements, since we need to extract roughly n elements and each extraction involves a constant amount of work and one Heapify. Therefore the total running time of **HeapSort is Θ(n log n).**

**Max-Heapify Analysis:**

Children subtrees have size at most 2n/3

$T(n) \leq T(2n/3) + 1$ ⇒⇒ Apply recurrence: $T(n) = \Theta(\lg n)$

# QUICKSORT

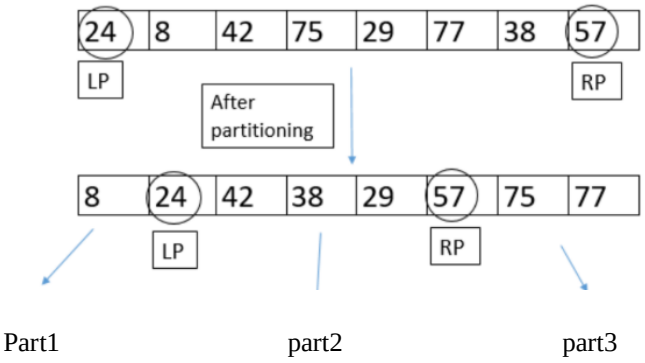
QuickSort is a Divide and Conquer algorithm. It picks an element as pivot and partitions the given array around the picked pivot. There are many different versions of quickSort that pick pivot in different ways.

1. Always pick first element as pivot: The pivot is always the first element of the array to be sorted.
2. Pick a random element as pivot: The pivot is chosen at random from any element in the array to be sorted.
3. Pick a mid of first mid last element as pivot: The pivot is an element whose value is in the middle among the {first, middle, last} elements in the array to be sorted.
4. Pick dual elements as pivots: This version has two pivots: one in the left end and one in the right end of the array. The left pivot (LP) must be less than or equal to the right pivot (RP).

### Partition():

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

1. For Type 1, the pivot is always selected as the first element.
2. For Type 2, the pivot is randomly selected and the first element swap with the selected pivot. After this step, we follow the steps of Type 1.
3. The pivot is selected from median of 3 elements. Those are first element of array, middle element of array and last element of array. After selecting the pivot, the first elements and the pivot are swapped after that follow type 1 steps.
4. Type 4 için 2 tane pivot seçilir (PR) ile (LP). Input array is divided into three parts. In the first part, all elements will be less than LP, in the second part all elements will be greater or equal to LP and also will be less than or equal to RP, and in the third part all elements will be greater than RP. Until the existing pivots have stopped their moving, sorting process continues recursively.




Type 4 Dual Pivot QuickSort Algorithm schema

**QuickSort():**

Divide: Partition A[p...r] into two subarrays A[p...q-1] and A[q+1...r], such that each element in the first subarray A[p...q-1] is ≤ A[q] and A[q] is ≤ each element in the second subarray A[q+1...r].

Conquer: Sort the two subarrays by recursive calls to QUICKSORT.

```
QUICKSORT(A,p, r)
    if p<r
        q = PARTITION(A,p, r)  //divide
        QUICKSORT(A, p, q - 1) //conquer
        QUICKSORT(A, q - 1,r) //conquer
```

**Analysis of QuickSort Algorithm:**

Runtime of partition is Θ(n)

The running time of Quicksort depends on the partitioning of the subarrays: If the subarrays are balanced, then quicksort can run as fast as mergesort. If they are unbalanced, then quicksort can run as slowly as insertion sort.

Worst-case Partitioning: Occurs when the sub-arrays are completely unbalanced. It happens when input array is already ordered. Same running time as insertion sort.

$$T(n) = T(n-1) + T(0) + \Theta(n) = T(n-1)+\Theta(n) = \Theta(n2)$$

Balanced Partitioning: Assume that PARTITION always produces a 9-to-1 split.

$$T(n) = T(9n/10) + T(n/10) + n = \Theta(n \lg n)$$

Best-case Partitioning: Assume that PARTITION always produces n/2 splits.

$$T(n) = 2T(n/2) + \Theta(n) = \Theta(n \lg n)$$

The sorting algorithm is a Dual-Pivot Quicksort by Vladimir Yaroslavskiy, Jon Bentley, and Joshua Bloch. This algorithm offers Θ(n lgn) performance on many data sets that cause other quicksorts to degrade to quadratic performance, and is typically faster than traditional(one-pivot) Quicksort implementations.

# INTROSORT

In the introSort method, you must implement a hybrid sorting algorithm. Heapsort and quicksort algorithms should be used while writing this sorting method. It is called as introsort or introspective sort, which begins quicksort and switches to heapsort when the recursion depth exceeds a level based on (the logarithm of) the number of elements being sorted.The maximum recursion depth is defined as follows.

$$maxdepth = [\ log(\ length(Input)\ )\ ] \times 2$$

**Introsort steps:**

1. Calculate the maxdepth
2. Select the pivot (Choose the last item as a pivot)
3. Partitioning of the array
4. Repeat step b and c recursively until maxdepth = 0
5. If maxdepth = 0 invoke heapsort

**Pseudo code:**

```
procedure sort(A : array):
    maxdepth = [log(length(A))] × 2
    introsort(A, maxdepth)
procedure introsort(A, maxdepth):
    n ← length(A)
    p ← partition(A) // pivot selection, p final pivot position
    if : n ≤ 1:
        return // base case
    else if : maxdepth = 0:
        heapsort(A)
    Else:
        introsort(A[0:p], maxdepth - 1)
        introsort(A[p+1:n], maxdepth - 1)
```

**Analysis of IntroSort Algorithm:**

Since Quicksort can have a worse case O(N2) time complexity and it also increases the recursion stack space (O(log N) if tail recursion applied), so to avoid all these, we need to switch the algorithm from Quicksort to another if there is a chance of worse case. So Introsort solves this problem by switching to Heapsort.

Best Case: $\Theta(N.\log N)$ Average Case:

→Θ(N.log N) Worse Case: → Θ(N.log N)

# COMPARE RUNNING TIME

HeapSort, QuickSort and IntroSort algorithms comparisons were made using Java programming language. Different four array mode were used. These are integer array, random integers array, increasing integers array and decreasing integer array. These algorithms are executed in 1000, 10000 and 100000 dimensions. Then outputs are calculated in milliseconds (double nanotube/1000000). Moreover, table-1 was created.

HeapSort algorithm has Θ(n.log n) running time.This algorithm works fastest when sorting ascending or descending sequences and running times are close to each other.

QuickSort algorithm has 4 different type. These running time depends on the inputs. Firstly, when pivot was selected as first element of array, it has a worst case for equal, increasing and decreasing integers input. However, when pivot was selected as a random element,it has a best case for increasing and decreasing integers input. When quicksort type is Mid Of First Mid Last Element, it has a best case running time but it has worst case for increasing integers. Then it has a worst case for equals and decreasing integers. Although, dual pivot is worst case for increasing and decreasing integer input, it is a best case for equal integers in dual pivot.

IntroSort algorithm has Θ(n.log n) running time. It has a good running time for each data set.

| Values: nanoTime/1000000 | EQUAL INTEGERS | | | RANDOM INTEGERS | | | INCREASING INTEGERS | | | DECREASING INTEGERS | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 1.000 | 10.000 | 100.000 | 1.000 | 10.000 | 100.000 | 1.000 | 10.000 | 100.000 | 1.000 | 10.000 | 100.000 |
| HeapSort | 0,42 | 0,94 | 3,48 | 1,44 | 4,10 | 14,55 | 0,14 | 0,77 | 8,15 | 0,08 | 0,68 | 7,94 |
| QuickSort First Element | 8,18 | 23,08 | 1.489,82 | 0,72 | 1,65 | 9,78 | 0,56 | 60,86 | 5.388,67 | 0,77 | 63,61 | 6.319,92 |
| QuickSort Random Element | 12,74 | 17,66 | 1.782,47 | 0,97 | 2,10 | 14,27 | 0,11 | 0,88 | 6,51 | 0,07 | 0,61 | 6,54 |
| QuickSort MidOf FirstMidLast Element | 9,41 | 18,52 | 1.418,23 | 0,75 | 2,13 | 10,70 | 0,05 | 0,34 | 3,95 | 0,22 | 11,59 | 1.283,24 |
| dual Pivot QuickSort | 0,90 | 1,36 | 13,86 | 0,27 | 3,05 | 33,83 | 3,34 | 94,36 | 4.318,54 | 0,83 | 38,70 | 3.815,10 |
| IntroSort | 0,81 | 7,72 | 11,63 | 1,26 | 4,85 | 20,99 | 0,08 | 0,93 | 9,99 | 0,08 | 0,79 | 10,71 |

tablo-1

# Results

Consequently, When input data set proximity the state of equals integers, we use heapSort because for this data set, it has better run time than other algorithms. If we have a random data set, we have a lot of alternative algorithm. HeapSort, QuickSort(not dual pivot) has average running times. They are very close. For the incremental and decreasing integers data set, use the heapSort, IntroSort and random elements quickSort algorithms because runtime results show the best status.

# Discussion

Assume that we have a Java hashmap that contains tweet numbers of twitter hashtags. Let's say, we have a hashmap like [“pazar”:23800, “Istanbul”:19400, “BirTavsiyeVerin”:5112, “BesiktasinMaciVar”:11000, .... ]. Imagine that there are hundreds of thousands of hashtags, if we rank the popularity of these twitter hashtags from the most popular to the unpopular one, which sorting algorithm do you use to do this operation faster?

In this example, the inputs are random integers in our data set. According to the data from Table 1, en iyi running time quicksort First element algorithm ve quicksort MidOf First-Mid-Last Element algorithms have the best running time value but if these algorithms get too bad performance in cases of increase and decrease integers, then running time will worsen if our data set approaches these situations. Therefore it is more convenient to use heapSort which is suitable for each data set. As an alternative to this algorithm, introsort comes immediately after.

# References

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

CME 2204 Algorithm Analysis / W6-Lecture.pdf / HeapSort

CME 2204 Algorithm Analysis / W6-Lecture.pdf / QuickSort

https://www.geeksforgeeks.org/dual-pivot-quicksort/

https://www.researchgate.net/publication/258454268_Java_7's_Dual_Pivot_Quicksort#pf34 //sayfa:53

https://www.geeksforgeeks.org/know-your-sorting-algorithm-set-2-introsort-cs-sorting-weapon/