# CS301 HW2

Adnan Kaan Ekiz

March 2019

## 1 Problem 1

For the first problem we have a set of n numbers S = $(n_0, n_1, ...., n_{n-1}, n_n)$ and we have to obtain the $k^{th}$ largest/smallest number in this set. For this purpose, some comparison-based and order statistics algorithms are given below:

### 1.1 A

In this section of the problem, there might be a lot of algorithms for the solution but I have dedided to use two of the most popular algorithms. Also talk about their complexity in general which is O(nlogn). These 2 algorithms are:

- Merge Sort
- Heap Sort

#### 1.1.1 Merge Sort

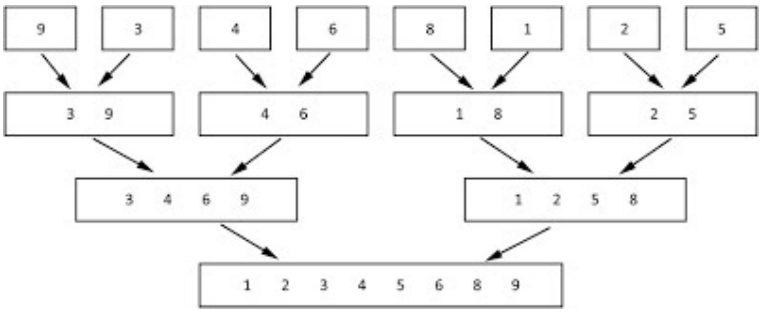
Pseudo-Code and complexity of each line:

```
Merge-Sort(a,p,r):                  # T(n) linear time
    if (p < r):                      # O(1) constant time
        q = (p + r) / 2
        Merge-Sort [A,p,q]          # 2T(n/2)
        Merge-Sort [A,q,r]
        Merge [A,p,q,r]             # O(n) linear time
```

Merge sort is one of the most famous comparison-based sorting algorithms and has a complexity of O(nlogn). Conceptually, a merge sort works as follows:

1. Divide the unsorted list into n sublists, each containing one element (a list of one element is considered sorted). This action takes $\Theta$(logn) time since we are always dividing our lists to 2. Therefore, it is considered as $\Theta$(logn) asymptotic worst case running time.

2. Repeatedly merge sublists to produce new sorted sublists until there is only one sublist remaining. This will be the sorted list. Even though dividing takes $\Theta$(logn), since we are doing it for n times it is considered as $\Theta$(nlogn)




Aside from the main part shown in the algorithm, merge action takes linear time since after sorting the lists merging sorted lists takes linear time. This algorithm has a smaller running time than insertion and bubble sort for n $<$ 30 due to smaller asymptotic worst-case running time.

Once the sorting is established, it takes $\Theta$(k) time to find the smallest/largest $k^{th}$ element from our sorted list. Since we only need to iterate through the list k times out of n elements. In general, sorting and also obtaining the $k^{th}$ largest/smallest element takes O(nlogn + k) time.

### 1.1.2 Heap Sort

Pseudo-Code and complexity of each line:

```
Heap-Sort(A,k):
    Build_Max_Heap[A]                          #O(n)
    for i <- length[A]:                        #O(n)
        do exchange A[1] <-> A[i]              #O(1)
           heap-size[A] <- heap-size[A-1]      #O(1)
           Max-Heapify[A,k]                    #O(logn)
```

Although somewhat slower in practice on most machines than a well-implemented quicksort, it has the advantage of a more favorable worst-case O(nlogn) runtime. It consists of the use of a heap data structure rather than a linear-time search to find the maximum. In general, algorith goes as follows:

**Tighter Bound:** Each call to MAX-HEAPIFY requires time $O(h)$ where $h$ is the height of node $i$. Therefore running time is

$$\sum_{h=0}^{\log n} \underbrace{\frac{n}{2^h+1}}_{\text{Number of nodes at height } h} \times \underbrace{O(h)}_{\text{Running time for each node}} = O\left(n\sum_{h=0}^{\log n}\frac{h}{2^h}\right)$$
$$= O\left(n\sum_{h=0}^{\infty}\frac{h}{2^h}\right)$$
$$= O(n) \quad (1)$$

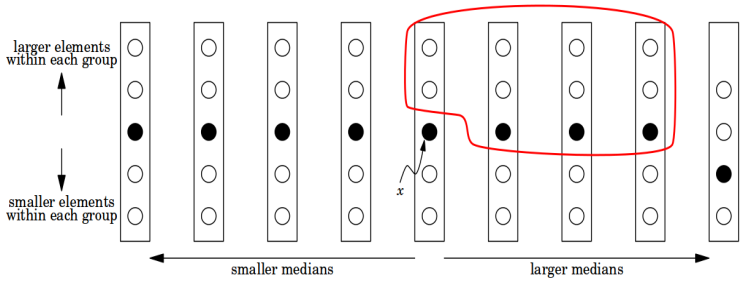
Note $\sum_{h=0}^{\infty} h/2^h = 2$.

1. Creating max-heap, assuming that number of nodes in height h is $n/(2^h + 1)$ and running time for each node is $\Theta$(h)
2. Finding the maximum in the heap which is Max-Heapify takes $\Theta$(logn) time but it is executed for n times which will be the number of the elements in the list.

In general, it is considered as O(nlogn) worst case asymptotic running time. Once the sorting established, it takes O(klogn) time to the find the the $k^{th}$ largest element from our heap. Overall time heap sort takes to finish is O(nlogn + klogn).

## 1.2 B

For the order statistics part, if chosen randomly complexity becomes $O(n^2)$. That's why its important to come up with a worst case order statistics algorithm of linear time. For this purpose, worst-case linear-time order statistics algorithm which ensures the linear time complexity can be used to handle this problem. Steps of the algorithm is given below:




- Divides n elements into groups of 5
- Find the median of each 5 element group by rote
- Recursively select the median x of the $\lfloor n/5 \rfloor$ group medians to the pivot

- We know that at least half the group medians are $\leq$ x, which is at least $\lfloor \lfloor n/5 \rfloor /2 \rfloor$ group medians
- Therefore we can say that at least $3\lfloor n/10 \rfloor$ elements are $\leq$ x. Similarly also at least $3\lfloor n/10 \rfloor$ elements are $\geq$ x

In general, this means for n$\geq$50, we have $3\lfloor n/10 \rfloor \geq n/4$. Which means recursive call to is executed recursively on $\leq 3n/4$ elements. In the end we get the following recurrence relation:
T(n) = T(n/5)+T(3n/4)+$\Theta$(n)
If we solve the recurrence relation it will give:
T(n) = cn - (cn/20 - $\Theta$(n)) which is $\leq$cn
Therefore we can prove that this algorithm which consists of choosing the pivot takes O(n) time.

After finding the pivot we know that we need to partition around the pivot that we have found(k'th largest), so for every in array, it is important to check whether that number/element is smaller or greater than our pivot. This means iterating all elements of the array for once, therefore, it will cost $\Theta$(n).

Last of all during the sorting phase, basically we need to sort k largest numbers within each other. Essentially, this problem is really close to our subproblem with one small exception. In subproblem (A), we had n numbers instead of $k^{th}$ largest numbers. Hence, our time complexity will be O(klogk) for k largest numbers.

Based on the analyze we came up with, total run time of the algorithm is O(n+klogk). I would use the second method(partition) since second algorithm has smaller average case complexity although both of the algorithms have the same worst case complexity. To explain this decision:
**WORST CASE**
A) O(nlogn+n)
B) O(nlogn+n)
**AVERAGE CASE**
A) O(nlogn+k)
B) O(klogk+n)

Since the worst case only applies when k = n, for all other cases it is much more beneficial to choose algorithm **(B)** due to smaller asymptotic worst case running time.

# 2 Problem 2

In computer science, radix-sort is a non-comparative integer sorting algorithm that sorts data with integer keys by grouping keys by the individual digits which

share the same significant position and value. Because integers can represent strings of characters (e.g., names or dates), radix-sort is not limited to just integers.

## 2.1 A

Originally radix-sort is designed to compare the digits of the number and sort these integers according to that comparison. However, for the string we know that each letter represented by an ASCII code corresponding to it. Due to this fact, my algorithm will consist several modifications such as:

- Names will be represented as seperate lists of integers such that each element of the list corresponds to ASCII code of a letter in that name.
- Instead of starting to comparison from the least significant bit, algorithm will start comparison from the last elements of the lists that belongs to different names.
- Instead of looking to a bigger value while comparing bits, algorithm needs to prioritize if value is being smaller or not since first letters of the alphabet will have smaller numbers than the last letters of the alphabet in terms of ASCII code.
- If different names has different length then add 0 to the end of the list until all lists have the same number of element.

## 2.2 B

| Dec | Hex | | Dec | Hex | | Dec | Hex | | Dec | Hex | | Dec | Hex | | Dec | Hex | | Dec | Hex | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 00 | NUL | 16 | 10 | DLE | 32 | 20 | | 48 | 30 | 0 | 64 | 40 | @ | 80 | 50 | P | 96 | 60 | ` | 112 | 70 | p |
| 1 | 01 | SOH | 17 | 11 | DC1 | 33 | 21 | ! | 49 | 31 | 1 | 65 | 41 | A | 81 | 51 | Q | 97 | 61 | a | 113 | 71 | q |
| 2 | 02 | STX | 18 | 12 | DC2 | 34 | 22 | " | 50 | 32 | 2 | 66 | 42 | B | 82 | 52 | R | 98 | 62 | b | 114 | 72 | r |
| 3 | 03 | ETX | 19 | 13 | DC3 | 35 | 23 | # | 51 | 33 | 3 | 67 | 43 | C | 83 | 53 | S | 99 | 63 | c | 115 | 73 | s |
| 4 | 04 | EOT | 20 | 14 | DC4 | 36 | 24 | $ | 52 | 34 | 4 | 68 | 44 | D | 84 | 54 | T | 100 | 64 | d | 116 | 74 | t |
| 5 | 05 | ENQ | 21 | 15 | NAK | 37 | 25 | % | 53 | 35 | 5 | 69 | 45 | E | 85 | 55 | U | 101 | 65 | e | 117 | 75 | u |
| 6 | 06 | ACK | 22 | 16 | SYN | 38 | 26 | & | 54 | 36 | 6 | 70 | 46 | F | 86 | 56 | V | 102 | 66 | f | 118 | 76 | v |
| 7 | 07 | BEL | 23 | 17 | ETB | 39 | 27 | ' | 55 | 37 | 7 | 71 | 47 | G | 87 | 57 | W | 103 | 67 | g | 119 | 77 | w |
| 8 | 08 | BS | 24 | 18 | CAN | 40 | 28 | ( | 56 | 38 | 8 | 72 | 48 | H | 88 | 58 | X | 104 | 68 | h | 120 | 78 | x |
| 9 | 09 | HT | 25 | 19 | EM | 41 | 29 | ) | 57 | 39 | 9 | 73 | 49 | I | 89 | 59 | Y | 105 | 69 | i | 121 | 79 | y |
| 10 | 0A | LF | 26 | 1A | SUB | 42 | 2A | * | 58 | 3A | : | 74 | 4A | J | 90 | 5A | Z | 106 | 6A | j | 122 | 7A | z |
| 11 | 0B | VT | 27 | 1B | ESC | 43 | 2B | + | 59 | 3B | ; | 75 | 4B | K | 91 | 5B | [ | 107 | 6B | k | 123 | 7B | { |
| 12 | 0C | FF | 28 | 1C | FS | 44 | 2C | , | 60 | 3C | < | 76 | 4C | L | 92 | 5C | \ | 108 | 6C | l | 124 | 7C | \| |
| 13 | 0D | CR | 29 | 1D | GS | 45 | 2D | - | 61 | 3D | = | 77 | 4D | M | 93 | 5D | ] | 109 | 6D | m | 125 | 7D | } |
| 14 | 0E | SO | 30 | 1E | RS | 46 | 2E | . | 62 | 3E | > | 78 | 4E | N | 94 | 5E | ^ | 110 | 6E | n | 126 | 7E | ~ |
| 15 | 0F | SI | 31 | 1F | US | 47 | 2F | / | 63 | 3F | ? | 79 | 4F | O | 95 | 5F | _ | 111 | 6F | o | 127 | 7F | DEL |

Depending on the ASCII table [”VEYSEL”,“EGE”,”SELIN”,”YASIN”] will be represented like this at the beginning of the algorithm.

VEYSEL = [86,69,89,83,69,76]

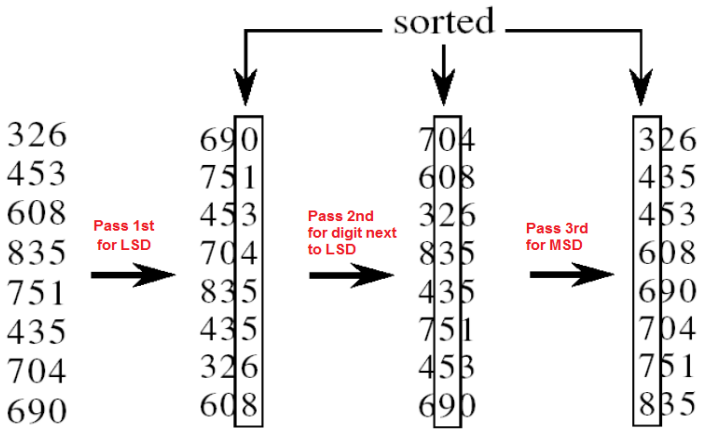
EGE = [69,71,69,0,0,0]
SELIN = [83,69,76,73,78,0]
YASIN = [89,65,83,73,78,0]
Steps of the algorithm as follows:

- First, looking at the last element of the lists. Values are 76-0-0-0 respectively. Now my name order became: **EGE-SELIN-YASIN-VEYSEL**. Since other elements all have value 0, just sorting the names depending on the order of the input.
- Then comparing the $(n-1)^{th}$ elements. Values are 69-0-78-78 respectively. Now name order became: **EGE-VEYSEL-SELIN-VEYSEL**.
- Comparing the $(n-2)^{th}$ elements. Values are 83-0-73-73 respectively. So, new name order will be: **EGE-SELIN-YASIN-VEYSEL**.
- Comparing the $(n-3)^{th}$ elements. Values are 89-69-76-83 respectively. So, new name order will be: **EGE-SELIN-YASIN-VEYSEL**.
- Comparing the $(n-4)^{th}$ elements. Values are 69-71-69-65 respectively. So, new name order will be: **YASIN-SELIN-VEYSEL-EGE**.
- Comparing the $(n-5)^{th}$ elements. Values are 86-69-83-89 respectively. So, new name order will be: **EGE-SELIN-VEYSEL-YASIN**.

Final result of the sorted list depending on the radix algorithm is:
**EGE-SELIN-VEYSEL-YASIN**

## 2.3 C

Original radix algorithm had a complexity of O(n*d) where d is the number of digits and n is the size of the array(number of elements which is going to be compared).



Since modified algorithm does compare the numbers rather than bits, for the complexity analysis of the modified algorithm; n will be represented as number of names which will be compared and e represents the max number of letters in the words since we add 0 to the end of the list if name length is smaller than the length of longest name. Finally, pseudocode of the algorithm is as follows:

```
Radix-Sort-Name(L):
    initalize a sublist                     #O(1)
    for each name in the list L:            #O(n)
        add ASCII numbers to a sublist      #O(n*e)
    for length max name to 0:               #O(e)
        compare[sublist]                    #O(n*e)
        change-order[]
```

Therefore, I can say that modified algorithm has a running time of O(n*e) in general.