Student Name: Mert Ekici
Student ID: 26772

# Algorithms (CS301)
Assignment 2

Sabancı Üniversitesi

## Problem 1 (Order Statistics)

Suppose that you are given a set of n numbers. The goal is to find the k smallest numbers in this set, in sorted order. For each method below, identify relevant algorithms with the best asymptotic worst-case running time (e.g., which sorting algorithm? which order-statistics algorithm?), and analyze the running time of the overall algorithm in terms of n and k.

(a) First sort the numbers using a comparison-based sorting algorithm, and then return the k smallest numbers.

Naive approach for this problem is based on sorting the numbers using a comparison-based algorithm and returning the k smallest elements.

**Comparison sorting** takes $\Omega(n \log n)$ time. To achieve best asymptotic worst-case running time with a comparison-based sorting algorithm (for k smallest sorted elements), we can use;

- Merge Sort
- Heap Sort

which have the worst-case running time of $O(n \log n)$. QuickSort is eliminated here since it has $O(n^2)$ worst-case running time although its average running time is $O(n \log n)$.

After sorting the elements in ascending order using Merge or Heap Sort with $O(n \log n)$, the first k numbers of the array are returned which is $O(k)$.

$$\text{Since } O(f) + O(g) = O(f + g) = O(max(f, g))$$

$$O(k) + O(n \log n) = O(k + n \log n) = O(n \log n)$$

(b) First use an order-statistics algorithm to find the k'th smallest number, then partition around that number to get the k smallest numbers, and then sort these k smallest numbers using a comparison-based sorting algorithm.

The second approach finds the k smallest numbers in three steps:

1. An order-statistics algorithm is used to find the k'th smallest number.
2. The k-smallest numbers are collected in one side by partitioning around the k'th smallest number.
3. The k-1 smallest numbers in the lower part are sorted using a comparison-based sorting algorithm and the k smallest numbers are returned.

Within these three steps, we should use relevant algorithms with the best asymptotic worst-case running time and analyze the running time of the overall algorithm in terms of n and k.

**Step 1**

*Worst-case linear-time order statistics algorithm is used as an order-statistics algorithm $O(n)$.*

**Randomized divide and conquer algorithm** works fast (linear expected time) and it is an excellent algorithm in practice. However, it performs poorly with the worst case scenario $\Theta(n^2)$.

**Worst-case linear-time order statistics algorithm** runs in linear time $O(n)$ even for the worst-case scenario.

$$\boxed{\text{Hence, finding the k'th smallest number is } O(n).}$$

**Step 2**

*QuickSort Partition algorithm is used to partition around the kth smallest element $O(n)$.*

$$\boxed{\text{Partition around the k'th smallest element is } O(n).}$$

**Step 3**

*Merge Sort or Heap Sort can be used as a comparison-based sorting algorithm since they have the best asymptotic worst-case running time $O(n \log n)$.*

$$\boxed{\text{Sorting the k-1 smallest numbers using Merge or Heap sort is } O(k \log k).}$$

$$\begin{aligned}\textbf{Running Time of the Overall Algorithm} &= Step1 + Step2 + Step3 \\ &= O(n) + O(n) + O(k \log k) \\ &= O(n + k \log k)\end{aligned}$$

***Which method would you use? Please explain why.***

If the same comparison-based algorithm is used for both methods, the second method outperforms the naive approach since **k** can be equal to **n** at most.

$$k \leq n$$
$$O(n + k \log k) \leq O(n \log n)$$

Even if we consider the space complexities of these two methods, no differences would be observed since they use space only for comparison-based sorting. Thus, I would use the second method for the worst-case scenario.

## Problem 2 (Linear-Time Sorting)

(a) How can you modify the radix sort algorithm for integers, to sort strings? Please explain the modifications.

**Radix Sort** is mostly used with numerical values but we can modify it to sort strings as well. If we approach each character of the string as numerical digits (but 256 possible values this time), numerical Radix sort algorithm can be converted to a string-based version using a pretty similar procedure.

For the numerical case, we used Radix sort by starting from the least significant digit (with auxiliary stable sort) in the lecture. However, given string array has different length strings in part (b). Thus, we will modify the Most Significant Digit Radix Sort (MSD with counting sort) to use it with strings. Unlike LSD, we can use MSD to sort variable length strings. Least Significant Digit method works for fixed-size strings. LSD is a stable sorting algorithm but MSD can either be stable or unstable. Since we modify the radix sort from the lecture into a variable length string version, the obtained results will be stable MSD sorting algorithm with counting sort.

In standard Radix sort, array is repetitively sorted for each digit. Depending on the type of Radix sort, digit sequence can start form the least significant digit to the most significant digit or vice versa. To preserve linear time sorting, counting sort is used as intermediate sorting algorithm (it collects the number of counts for each possible digit value 0-9 and sorts accordingly for each digit).

### Modifications

For the implementation of string based version, 256 size array (number of possible 1-byte characters which is $2^8$) is used instead of 10 size (0-9) to count character occurrences and sort given string array lexicographically. Where strings are not necessarily the same length, most-to-least (significance) order is preferred. Strings that start with ***c*** should appear before strings that start with ***d*** and goes that way. Character indexed counting sort is used to sort strings based on their first character and then, subarrays (smaller arrays with the same first character) are recursively sorted for each corresponding character.

The recursive calls lead to a large number of small sets where counting sort still needs $\Omega(\sigma)$ time independently from subset size. To avoid this potential cost, algorithm can be implemented in a way that it switches to quicksort for small sets (if set size is smaller than $\sigma$) but it is not implemented in this version. ($\sigma$ =length of the radix or $\sigma$-base).

### Steps of the Algorithm

Step 1: Check the recursion break condition. Terminate the execution if the base case is reached.

Step 2: Apply counting sort (radix size 256) based on the current digit. Starts with the most significant digit and goes through the least significant one.

Step 3: Recursively call MSD Radix String Sort on each partially sorted subset and sort them by their next digit (recursively sort each subarray of strings with the same digit D based on their next digit D+1).

**Algorithm 1** MSD Radix Sort

```
Input:   String array R, starting index S, ending index E, digit index D
Output:  R in ascending lexicographic order

function RadixStringSort(R,S,E,D)
    if $E \leq S$ then
        break
    end if
    Initialize count array with the size of 256
    countsort(R,D,count)
    for $i = 0$ to 256 do
        RadixStringSort(R , S+count[i] , S+count[i + 1]-1 , D+1)
    end for
    return the array R of sorted strings
```

```python
#ASCII value of the character at digit d in a given string
def chrASCII(s,d):
    if len(s)<=d:
        return -1
    else:
        return ord(s[d])

def RadixStringSort(R,S,E,D):

    if E<=S:                        #base condition check for the recursion
        return

    count = [0]*(256+2)
    temp = [""]*(len(R))            #temp is created for string swaps

    for i in range (S,E+1):         #count occurrences of each character
        c = chrASCII(R[i],D)
        #print(c)
        count[c]+=1;

    for r in range(256+1):          #now count contains actual positions
        count[r+1] += count[r]

    for j in range(E,S-1,-1):       #fill the temp
        c = chrASCII(R[j],D)
        #print(count)
        temp[count[c]-1] = R[j]
        count[c] -= 1

    for m in range(S,E+1):          #update the original array
        R[m] = temp[m-S]

    #recursively call RadixStringSort on each partially sorted subset
    #sort them by the next digit
    for k in range(256):
        RadixStringSort(R,S+count[k],S+count[k+1]-1,D+1)
```

(b) Illustrate how your algorithm sorts the following list of strings.

["BATURAY" , "GORKEM" , "GIRAY" , "TAHIR" ,"BARIS"]

Please show every step of your algorithm.

1. Check if base condition is reached: $0 < 4$ not reached
2. Order the strings by their most significant digit with counting sort (stable).

| | | |
|---|---|---|
| BATURAY | $\Longrightarrow$ | BATURAY |
| GORKEM | | BARIS |
| GIRAY | | GORKEM |
| TAHIR | | GIRAY |
| BARIS | | TAHIR |

3. Call Radix String Sort recursively on each partially sorted subsets for all possible characters. (Only the occurring characters are shown in the following steps.)

   3.1 Character 'B'

   3.1.1 Check if base condition is reached: $0 < 1$ not reached

   3.1.2 Order the strings starting with "B" by their second digit (using counting sort).

| | | |
|---|---|---|
| BATURAY | $\Longrightarrow$ | BATURAY |
| BARIS | | BARIS |
| GORKEM | | GORKEM |
| GIRAY | | GIRAY |
| TAHIR | | TAHIR |

Nothing is changed since both characters are 'A'.

   3.1.3 Call Radix String Sort recursively.

   a) Character 'A'

   a1) Check if base condition is reached: $0 < 1$ not reached

   a2) Order the strings starting with "BA" by their third digit (using counting sort).

| | | |
|---|---|---|
| BATURAY | $\Longrightarrow$ | BARIS |
| BARIS | | BATURAY |
| GORKEM | | GORKEM |
| GIRAY | | GIRAY |
| TAHIR | | TAHIR |

   a3) Call Radix String Sort recursively.

   3.2 Character 'G'

   3.2.1 Check if base condition is reached: $2 < 3$ not reached

   3.2.2 Order the strings starting with "G" by their second digit (using counting sort).

| | | |
|---|---|---|
| BARIS | $\Longrightarrow$ | BARIS |
| BATURAY | | BATURAY |
| GORKEM | | GIRAY |
| GIRAY | | GORKEM |
| TAHIR | | TAHIR |

   3.2.3 Call Radix String Sort recursively.

   3.3 Character 'T', base condition is reached: $4 = 4$. Move to the next character.

**RadixStringSort Function with the given list of strings**

```
R = ["BATURAY", "GORKEM", "GIRAY", "TAHIR", "BARIS"]
print("non-sorted: ",R)
RadixStringSort(R,0,len(R)-1,0)
print("S and E values of Recursive Cases (Non-Base Conditions): ",nonbase)
print("sorted: ",R)
```

**Output:**

```
non-sorted:  ["BATURAY", "GORKEM", "GIRAY", "TAHIR", "BARIS"]
S and E values of Recursive Cases: [[0, 4], [0, 1], [0, 1], [2, 3]]
sorted:   ["BARIS", "BATURAY", "GIRAY", "GORKEM", "TAHIR"]
```

When implemented RadixStringSort Function is applied to the given list of strings, result matches with the previous step-by-step solution (all the non-base/recursive cases are shown in the previous part: [0,4], [0,1], [0,1], [2,3]).

(c) Analyze the running time of the modified algorithm.

number of elements: $n$
average length of strings: $m$
total number of characters in the data: $m * n = T$
size of radix: $r = 256$
length of longest prefix match: $d$

If we investigate the algorithm under two steps;

- Counting Sort $= O(n + r) = O(n + 256) = O(n)$
- Number of digits checked (Recursive calls): $R$

**Time Complexity**

In the best-case scenario, all strings are different and no common prefix exists. MSD Radix String Sort uses just one pass of counting sort for the most significant digit of the strings.

**Best Case** (no common prefix - one counting sort for most significant digit is enough) = O(n)

When random inputs are given, MSD Radix String Sort examines a small fraction of the string characters, and the running time is sub-linear in the total number of characters in the data (almost linear in number of elements). For non-random inputs, MSD Radix String Sort still can perform sub-linear but might need to examine more characters than the random case. Particularly, it has to examine all the characters in common prefixes, so the running time is nearly linear in the number of characters in the data when the length of longest common prefix is non-negligible (significant numbers of equal keys).

We expect the subarrays to be all about the same size, so the recurrence:

$$T(n) = rT(n/r) + n$$
$$T(n) = 256T(n/256) + n$$

by the Master Theorem;
**Average Case** $= O(n \log_r n) = O(n \log_{256} n)$

In the worst-case scenario, MSD Radix String Sort examines all the characters in the keys, so the running time is linear in the number of characters in the data (like LSD string sort). A worst-case input is one with all strings equal.

**Worst Case** (all strings equal - all characters are examined) = O(m*n) = O(T)
$m \times n$ = average string length $\times$ number of elements = total number of characters in the data

## Space Complexity (Auxiliary Space)

**Extra Space** MSD Radix String Sort: O(n + dr) = O(n + 256d)
d = length of longest prefix match and r = size of radix
(r=10 possible numbers or r=256 characters or r=2 for Binary).

Maximum number of recursion for each subset is equal to the length of longest prefix match which indicates that 256 size arrays are created $d$ times results in $O(n + 256d) = O(n + d)$.

## Disadvantages of MSD Radix String Sort

- It requires extra auxiliary space and additional space for count[ ].
- Inner loops has lots of complex executions.
- Memory accesses are randomly which is inefficient use of cache.