CS301 Homework 2

Yağız Kılıçarslan

November 2022

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:

Comparison-based sorting algorithms, as discussed in the lecture, has best-case running time of $\Omega(nlgn)$. According to this, MergeSort and HeapSort being optimum comparison-based sorting algorithms, can be candidates. While QuickSort yields $\Theta(nlgn)$ running time for most cases, it yields $O(n^2)$ for worst-case, thus, should not be considered as a candidate. Note that although randomizedQuickSort has $\Theta(nlgn)$ expected time complexity, it is not same as the worst-case running time, therefore, not a solution.

Comparing the candidates, HeapSort and MergeSort, although their sorting worst-case time complexities are the same, which is $O(nlgn)$, selecting k smallest elements may cause different time complexities, thus, we should check how they perform for our problem.

Particularly, in HeapSort solution, to satisfy the *Heap Property*, in each iteration of selecting the smallest element will require $O(lgn)$ additional computation. Therefore, although the sorting will take $O(nlgn)$, with the selection overall algorithm will take $O(nlgn + klgn)$ complexity.

Comparing that to MergeSort, which will only require $O(k)$ additional computation for selecting as it does not have a constraint such as heap property. Thus, will take $O(nlgn + k)$ time complexity at worst-case. So, we will use MergeSort for this part. Analysis of MergeSort is as follows:

Guess: $O(nlgn)$
Assume: $T(k) \leq c * klgk$ for every $k < n$ holds.

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

$$T(n) \leq 2c(n/2 * lg(n/2)) + \Theta(n)$$

$$T(n) \leq cn(lgn - lg2) + \Theta(n)$$

$$T(n) \leq cnlgn - [cn - \Theta(n)]$$

Right-hand side is non-negative for big enough c dominating $\Theta(n)$.
Therefore, MergeSort is $O(nlgn)$. When selection is added it becomes:

$$T(n, k) = O(nlgn) + k$$

Where it will iterate over smallest k elements. Thus it is equal to:

$$T(n, k) = O(nlgn + k)$$

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.

We will use Order-Statistics SELECT(N, K) algorithms, which will give us the k'th smallest number in linear-time complexity.

SELECT(N, K) $= T(n)$

(1.) n elements are divided into sets of 5. $[\Theta(n)]$
(2.) Find the **median** of each set. $[O(\lfloor n/5 \rfloor)]$
(3.) Find the median-of-medians by recursively calling SELECT. $[T(n/5)]$
(4.) Set **pivot** = median-of-medians

At this point, rank(**pivot**) can be smaller than, equal to or larger than k, where rank() is the place in sorted order.

If rank(**pivot**) $< k$, then it means that elements bigger than pivot we should apply the above-mentioned steps recursively to the subset of elements that are bigger than the pivot.

Else If rank(**pivot**) $> k$, then we should apply the above-mentioned steps recursively to the subset of elements that are smaller than the pivot (k = k - rank(pivot)).

In the worst-case scenario, k would be equal to the size of the set (biggest number). Therefore, in that case SELECT will always recursively call itself for the bigger subset.

Since we know that when the median-of-medians (pivot) is found, the $3 * \frac{\lfloor \frac{n}{5} \rfloor}{2}$ elements are smaller than the pivot (explained in the lecture). In the worst-case scenario this will mean that $\frac{7n}{10}$ elements will be bigger than the pivot. Therefore, in this scenario SELECT will call itself recursively for the bigger subset, which has $\frac{7n}{10}$ elements, thus, will have the time complexity of $T(\frac{7n}{10})$.

After the k'th smallest element is found, **partitioning** is done, which will go over each element in the set, thus is $O(n)$.

Combining the above-mentioned steps:

1. Find the pivot (median-of-medians) $= T(\frac{n}{5})$
2. Look for k'th smallest element in bigger subset (worst-case) $= T(\frac{7n}{10})$
3. Partition around that number $= O(n)$

$$T(n) = T(\frac{n}{5}) + T(\frac{7n}{10}) + O(n)$$

Analysis of OrderStatistics Partitioning

We will show that $T(n) = \Theta(n)$ by showing that $T(n) = O(n)$ and $T(n) = \Omega(n)$

Now, we will prove $T(n) = O(n)$ by substitution method:

Guess: $T(n) = O(n)$ Assume $T(k) \leq ck$ for k < n holds.

$$T(n) = T(\frac{n}{5}) + T(\frac{7n}{10}) + O(n)$$

$$T(n) \leq c\frac{n}{5} + c\frac{7n}{10} + O(n)$$

$$T(n) \leq c\frac{9n}{10} + O(n)$$

$$T(n) \leq cn - [c\frac{n}{10} - O(n)]$$

Since we have [*something we want*] - [*something non-negative*] for large enough c that dominates $O(n)$, we can say that $T(n) = O(n)$

Now, we will prove $T(n) = \Omega(n)$ by substitution method:

Guess: $T(n) = \Omega(n)$ Assume $T(k) \geq ck$ for k < n holds.

$$T(n) = T(\frac{n}{5}) + T(\frac{7n}{10}) + O(n)$$

$$T(n) \geq c\frac{n}{5} + c\frac{7n}{10} + O(n)$$

$$T(n) \geq c\frac{9n}{10} + O(n)$$

$$T(n) \geq cn + [O(n) - c\frac{n}{10})]$$

Since we have [*something we want*] - [*something non-negative*] for small enough c that is dominated by $O(n)$, we can say that $T(n) = \Omega(n)$

Since we showed that $T(n) = O(n)$ and $T(n) = \Omega(n)$ holds, by definition $T(n) = \Theta(n)$ holds as well.

Overall Complexity

When the partitioning is applied, we have a subset of numbers that are smaller than k'th number, and a subset that is greater than it.

After the partitioning, we will apply MergeSort for the subset with smaller numbers. As we have seen above, MergeSort for k numbers will take $O(klgk)$.

Finally, the overall algorithm will take:

$$T(n,k) = \Theta(n) + O(klgk)$$

Which method would you use? Please explain why.

Comparing the complexities found in part a and b we can see that:

$$(a) : T(n,k) = O(nlgn + k)$$

$$(b) : T(n,k) = O(n + klgk)$$

Obviously, since $k < n$, algorithm in part b is much better in terms of time complexity.

Problem : Linear-time sorting

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

Firstly, assuming that strings have equal size, we can convert each character in the string to their ASCII value. Then the problem basically turns into the integer sorting.

Secondly, to relax the assumption of equal sized strings, what we can do is before converting characters to integers, we can add character 'A' to the end of a string that is shorter than the longest string until the shorter string gets as long as the longest string. For example: BARIS, BATURAY, BARISCAN $->$ BARISAAA, BATURAYA, BARISCAN

This is because initially shorter strings should prioritized while sorting strings. That is the reason we added 'A' to end of shorter strings, so that when getting sorted starting from the last character, shorter strings will be prioritized as 'A' is the smallest value that can be given. For example: BARISAA, BARISAY. BARIS will be prioritized because of 'A's added.

After that, one optimization is to change each uppercase letter to lower case by deducting ASCII value of 'A', thus the array will be used in counting sort will have length of 26.

With these modifications we can use radix sort to sort these integers. Then, by simply converting them back to characters we're able to effectively sort strings using radixsort.

(b.) Illustrate how your algorithm sorts the following list of strings ["BATURAY", "GORKEM", "GIRAY", "TAHIR", "BARIS"]. Please show every step of your algorithm.

I've written a simple Python script to demonstrate the steps, which I believe help ease of read.

```
● → hw2 python3 radixsort.py

  ------ Sorting Strings with RadixSort: -------

  Initial list of strings: ['BATURAY', 'GORKEM', 'GIRAY', 'TAHIR', 'BARIS']
  Modifications:
  1. Find longest name: BATURAY (7)
  2. Fill shorter names with 'A': ['BATURAY', 'GORKEMA', 'GIRAYAA', 'TAHIRAA', 'BARISAA']
  3. Convert names to ASCII numbers of their chars:
BATURAY: [1, 0, 19, 20, 17, 0, 24]
GORKEM: [6, 14, 17, 10, 4, 12, 0]
GIRAY: [6, 8, 17, 0, 24, 0, 0]
TAHIR: [19, 0, 7, 8, 17, 0, 0]
BARIS: [1, 0, 17, 8, 18, 0, 0]
```

Initial modifications as described above

```
--------- RadixSort Steps ---------

start of index 6: ['BATURAY', 'GORKEMA', 'GIRAYAA', 'TAHIRAA', 'BARISAA']
Reading from A to C: [4, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0]
Compound C: [4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 5, 5]
Reading from C to B: [1, 0, 17, 8, 18, 0, 0] placed at 3
Reading from C to B: [19, 0, 7, 8, 17, 0, 0] placed at 2
Reading from C to B: [6, 8, 17, 0, 24, 0, 0] placed at 1
Reading from C to B: [6, 14, 17, 10, 4, 12, 0] placed at 0
Reading from C to B: [1, 0, 19, 20, 17, 0, 24] placed at 4
B: [[6, 14, 17, 10, 4, 12, 0], [6, 8, 17, 0, 24, 0, 0], [19, 0, 7, 8, 17, 0, 0], [1, 0, 17, 8, 18, 0, 0], [1, 0, 19, 20, 17, 0, 24]]
end of index 6: ['GORKEMA', 'GIRAYAA', 'TAHIRAA', 'BARISAA', 'BATURAY']
```

Starting from index 6, the last character.
At each step, we start by initializing 3 arrays, A, B and C.
A is our original list of names
B is initially empty at each iteration, and at the end filled with sorted list with respect to the character that is compared
C is an auxiliary array for keeping count of ASCII numbers of that index

Firstly, we initialize C by checking the index i'th integer (x) and incrementing x'th index of C by one. In the first iteration where we check the last index of each name, we have 4 0's and 1 25. This can be seen in 'Reading from A to C'.

Secondly, starting from the most left cell in C, we sum each consequtive cell and add it to the latter (C[i] = C[i-1] + C[i]). This can be seen in 'Compound C'

Thirdly, starting from the last element of A (x), we check x'th element in C (x'). This x' is basically the sorted index that will be in B. (B[x'] = x).

Moreover, since we add the elements from last index to the first we keep the final array stable. To observe that you can check the placing order. Although 'GORKEMA', 'GIRAYAA', 'TAHIRAA' and 'BARISAA' all have the same last character, 'BARISAA' is placed to the 3rd position where it belongs in the stable sort.

The same steps execute in the following iterations.

```
start of index 5: ['GORKEMA', 'GIRAYAA', 'TAHIRAA', 'BARISAA', 'BATURAY']
Reading from A to C: [4, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
Compound C: [4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5]
Reading from C to B: [1, 0, 19, 20, 17, 0, 24] placed at 3
Reading from C to B: [1, 0, 17, 8, 18, 0, 0] placed at 2
Reading from C to B: [19, 0, 7, 8, 17, 0, 0] placed at 1
Reading from C to B: [6, 8, 17, 0, 24, 0, 0] placed at 0
Reading from C to B: [6, 14, 17, 10, 4, 12, 0] placed at 4
B: [[6, 8, 17, 0, 24, 0, 0], [19, 0, 7, 8, 17, 0, 0], [1, 0, 17, 8, 18, 0, 0], [1, 0, 19, 20, 17, 0, 24], [6, 14, 17, 10, 4, 12, 0]]
end of index 5: ['GIRAYAA', 'TAHIRAA', 'BARISAA', 'BATURAY', 'GORKEMA']
```

```
start of index 4: ['GIRAYAA', 'TAHIRAA', 'BARISAA', 'BATURAY', 'GORKEMA']
Reading from A to C: [0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2, 1, 0, 0, 0, 0, 0, 1, 0]
Compound C: [0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 3, 4, 4, 4, 4, 4, 5, 5]
Reading from C to B: [6, 14, 17, 10, 4, 12, 0] placed at 0
Reading from C to B: [1, 0, 19, 20, 17, 0, 24] placed at 2
Reading from C to B: [1, 0, 17, 8, 18, 0, 0] placed at 3
Reading from C to B: [19, 0, 7, 8, 17, 0, 0] placed at 1
Reading from C to B: [6, 8, 17, 0, 24, 0, 0] placed at 4
B: [[6, 14, 17, 10, 4, 12, 0], [19, 0, 7, 8, 17, 0, 0], [1, 0, 19, 20, 17, 0, 24], [1, 0, 17, 8, 18, 0, 0], [6, 8, 17, 0, 24, 0, 0]]
end of index 4: ['GORKEMA', 'TAHIRAA', 'BATURAY', 'BARISAA', 'GIRAYAA']

start of index 3: ['GORKEMA', 'TAHIRAA', 'BATURAY', 'BARISAA', 'GIRAYAA']
Reading from A to C: [1, 0, 0, 0, 0, 0, 0, 0, 2, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0]
Compound C: [1, 1, 1, 1, 1, 1, 1, 1, 3, 3, 4, 4, 4, 4, 4, 4, 4, 4, 4, 5, 5, 5, 5, 5, 5]
Reading from C to B: [6, 8, 17, 0, 24, 0, 0] placed at 0
Reading from C to B: [1, 0, 17, 8, 18, 0, 0] placed at 2
Reading from C to B: [1, 0, 19, 20, 17, 0, 24] placed at 4
Reading from C to B: [19, 0, 7, 8, 17, 0, 0] placed at 1
Reading from C to B: [6, 14, 17, 10, 4, 12, 0] placed at 3
B: [[6, 8, 17, 0, 24, 0, 0], [19, 0, 7, 8, 17, 0, 0], [1, 0, 17, 8, 18, 0, 0], [6, 14, 17, 10, 4, 12, 0], [1, 0, 19, 20, 17, 0, 24]]
end of index 3: ['GIRAYAA', 'TAHIRAA', 'BARISAA', 'GORKEMA', 'BATURAY']

start of index 2: ['GIRAYAA', 'TAHIRAA', 'BARISAA', 'GORKEMA', 'BATURAY']
Reading from A to C: [0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 3, 0, 1, 0, 0, 0, 0, 0, 0]
Compound C: [0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 4, 4, 5, 5, 5, 5, 5, 5, 5, 5]
Reading from C to B: [1, 0, 19, 20, 17, 0, 24] placed at 4
Reading from C to B: [6, 14, 17, 10, 4, 12, 0] placed at 3
Reading from C to B: [1, 0, 17, 8, 18, 0, 0] placed at 2
Reading from C to B: [19, 0, 7, 8, 17, 0, 0] placed at 0
Reading from C to B: [6, 8, 17, 0, 24, 0, 0] placed at 1
B: [[19, 0, 7, 8, 17, 0, 0], [6, 8, 17, 0, 24, 0, 0], [1, 0, 17, 8, 18, 0, 0], [6, 14, 17, 10, 4, 12, 0], [1, 0, 19, 20, 17, 0, 24]]
end of index 2: ['TAHIRAA', 'GIRAYAA', 'BARISAA', 'GORKEMA', 'BATURAY']
```

```
start of index 1: ['TAHIRAA', 'GIRAYAA', 'BARISAA', 'GORKEMA', 'BATURAY']
Reading from A to C: [3, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
Compound C: [3, 3, 3, 3, 3, 3, 3, 3, 4, 4, 4, 4, 4, 4, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5]
Reading from C to B: [1, 0, 19, 20, 17, 0, 24] placed at 2
Reading from C to B: [6, 14, 17, 10, 4, 12, 0] placed at 4
Reading from C to B: [1, 0, 17, 8, 18, 0, 0] placed at 1
Reading from C to B: [6, 8, 17, 0, 24, 0, 0] placed at 3
Reading from C to B: [19, 0, 7, 8, 17, 0, 0] placed at 0
B: [[19, 0, 7, 8, 17, 0, 0], [1, 0, 17, 8, 18, 0, 0], [1, 0, 19, 20, 17, 0, 24], [6, 8, 17, 0, 24, 0, 0], [6, 14, 17, 10, 4, 12, 0]]
end of index 1: ['TAHIRAA', 'BARISAA', 'BATURAY', 'GIRAYAA', 'GORKEMA']

start of index 0: ['TAHIRAA', 'BARISAA', 'BATURAY', 'GIRAYAA', 'GORKEMA']
Reading from A to C: [0, 2, 0, 0, 0, 0, 0, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0]
Compound C: [0, 2, 2, 2, 2, 2, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 5, 5, 5, 5, 5, 5, 5]
Reading from C to B: [6, 14, 17, 10, 4, 12, 0] placed at 3
Reading from C to B: [6, 8, 17, 0, 24, 0, 0] placed at 2
Reading from C to B: [1, 0, 19, 20, 17, 0, 24] placed at 1
Reading from C to B: [1, 0, 17, 8, 18, 0, 0] placed at 0
Reading from C to B: [19, 0, 7, 8, 17, 0, 0] placed at 4
B: [[1, 0, 17, 8, 18, 0, 0], [1, 0, 19, 20, 17, 0, 24], [6, 8, 17, 0, 24, 0, 0], [6, 14, 17, 10, 4, 12, 0], [19, 0, 7, 8, 17, 0, 0]]
end of index 0: ['BARISAA', 'BATURAY', 'GIRAYAA', 'GORKEMA', 'TAHIRAA']

A after radixsort: [[1, 0, 17, 8, 18, 0, 0], [1, 0, 19, 20, 17, 0, 24], [6, 8, 17, 0, 24, 0, 0], [6, 14, 17, 10, 4, 12, 0], [19, 0, 7, 8, 17, 0, 0]
]
A after_convert_str:  ['BARISAA', 'BATURAY', 'GIRAYAA', 'GORKEMA', 'TAHIRAA']
```

At the end, you can see that the correct sorting order is assessed.

If desired, extra 'A's can be removed by a simple 'newName − > originalName' mapping

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

1. Find the longest string: $O(n)$ since only traversing the array once.
2. Fill shorter names with 'A': $O(n * l)$ where l is the length of the longest string
3. Convert to ASCII: $O(n * l)$ since there will two nested loops for each character

4. RadixSort: At each iteration: (n times)
4.1 Create C of size 26 (fixed k = 26). ($\Theta(k) = \Theta(1)$)
5 Compound C ($O(k)$)

In total one iteration is $= O(n + k)$

This is repeated for l times where l is the length of the longest string.

Thus complexity is:

$$T(n, l, k) = \Theta(l * (n + k)) + O(n * l)$$

With large enough n, k will be dominated as it is fixed length, thus time complexity will be:

$$\Theta(n * l)$$