CS 301 - Assignment 2

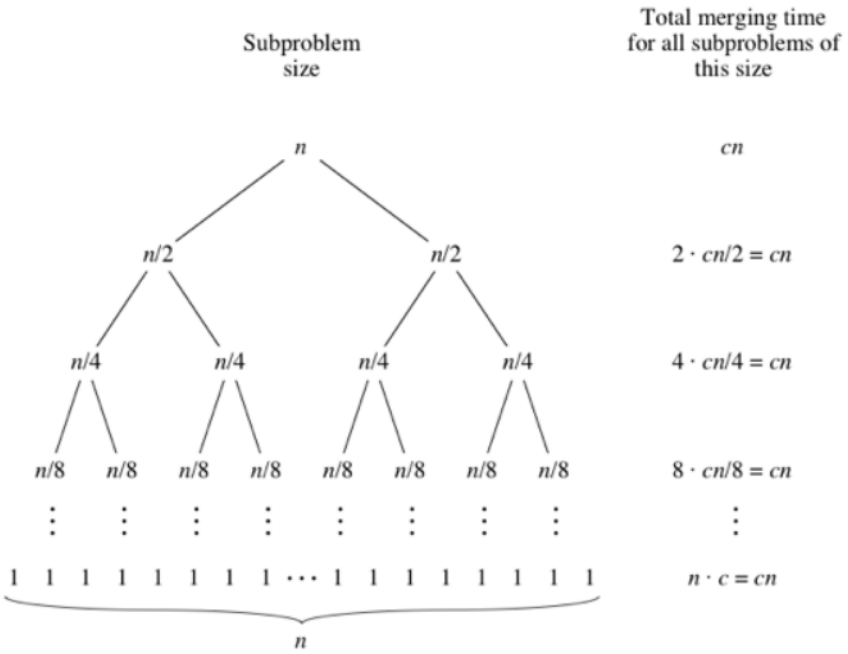
Rebah Özkoç
29207

April 1, 2022

Problem 1 (Order statistics)

a)

For the sorting n numbers, we can use merge sort algorithm. Merge sort algorithm is a comparison-based algorithm. It divides the problem into two equal sized sub-problems in every recursive step. In every step we get c*n as the total merging time. The height of the tree is $\log_2 n + 1$. This can be seen below in the picture.



So we have asymptotic running time complexity $\Theta(n * \log n)$ in the best-case, worst-case, and average-case for the merge sort algorithm. This is the optimal complexity for the sorting algorithms. After sorting the array, it takes $O(k)$ times for getting the first k elements from the array. In total, we have $O(n * \log n + k)$ for the time complexity. However, k is always smaller than n, so we can drop the k term and simply we get $O(n * \log n)$

b)

We can use the linear-time order-statistics algorithm of Blum, Floyd, Pratt, Rivest, and Tarjan to find the kth smallest element. This algorithm works like this:

1. Divide the n elements into groups of 5. Find the median of each 5-element group by rote.
2. Recursively SELECT the median x of the floor(n/5) group medians to be the pivot.
3. Partition around the pivot x. Let k = rank(x)
4. **if** i = k then return x
   elseif i < k
   then recursively SELECT the ith smallest element in the lower part
   else recursively SELECT the (i–k)th smallest element in the upper part

We can analyze the asymptotic running time algorithm by partitioning the tasks. We can divide the n element into groups of 5 and find the median of each 5-element group in $\Theta(n)$ time. The recursively selection of the median x of the floor(n/5) group medians to be the pivot takes T(n/5) time. Partitioning around the pivot x takes $\Theta(n)$ time. Lastly, the recursive part of after the if statement takes $T(3n/4)$ time.

If we sum up those terms, for the finding kth smallest element, we get:

$$T(n) = T(n/5) + T(3n/4) + \Theta(n)$$

We can claim that the algorithm runs in linear time. We can prove this by the substitution method. We want to show that there exists $c > 0 and n_0 > 0$ such that for every $n > n_0$ $T(n) < c*n$

We don't need to prove the base case because it is trivial.

We assume that $T(k) <= c * k for all k < n$

$$\begin{aligned} T(n) &= T(n/5) + T(3n/4) + \Theta(n) \\ T(n) &<= c * n/5 + 3 * c * n/4 + \Theta(n) \\ &= 19 * c * n/20 + c_2 * n \\ &= c * n - (c * n/20 - c_2 * n) \end{aligned}$$

$n_0 > 1 and c > 1 and c_2 < c/20$ satisfies the inequality.

Therefore the selection algorithm runs in linear time in the worst case. Similarly, we can choose appropriate c and c_2 and prove that the lower bound for the algorithm is also linear time. So we can find the k^{th} smallest element in $\Theta(n)$ time. By the nature of this algorithm we can find the k smallest element in linear time by just looking the preceding elements of the element k^{th} of the array that we got in the last recursive step. After finding the k smallest element in linear time, we can sort this k elements using the merge-sort. This takes $Theta(k * logk)$ time.

The total running time takes $\Theta(n) + \Theta(k) + \Theta(k * logk) = \Theta(n + k * logk)$ time.

As a conclusion, I would choose the second method if I want to have a faster algorithm. For the part (a) the best worst case asymptotic running time complexity is $\Theta(k + n * logn)$ wheras for the part (b) the best worst case asymptotic running time complexity is $\Theta(n + k * logk)$. Since $k <= n$ always, the second algorithm runs faster asymptotically and I would choose the second algorithm.

Problem 2 (Linear-time sorting)

a)

For sorting the strings with the radix sort first we need to modify the counting sort which is the algorithm the radix sort uses for sorting the digits. We can modify the counting sort to compare the ASCII values of the characters of the given strings. For handling the unequal length strings we need a different approach than the number sorting. We need to find the longest string in the array and fill the other strings end with a characters to make the strings equal length. For this purpose we need to choose a character that is unlikely to be in the strings and have a low ASCII value. For example, we can choose "!" for this character. We need to temporarily modify the given strings and make their length equal to the longest string in the array by putting "!"'s at the end of the strings.

Also we need to have a bigger auxiliary space C. Since we have 26 characters in the alphabet and plus our joker character "!", the array C needs to have 27 slots. We can disregard the uppercase letters and special characters without loss of generality for the sake of simplicity. The other modification that we need to do is holding the joker character "!" at the index 0 in the array C. By this way, shorter words with the same prefix will have high precedence. The alphabet starts at the index 1.

After these modifications we can run our algorithm like we did in the integer sorting. After the sorting we need to delete the joker characters from the strings.

b)

My algorithm first checks the length of the strings and determines the longest string in the given array. In this case it is "DILARA" with 6 characters. Then the algorithm adds joker characters (!) at the end of the strings to make their length equal to 6. The resulting array is like this:

F = ["MERT!!", "AYSU!!", "SELIN!", "ERDEM!", "DILARA"]

6^{th} Letters:

We get the last characters of the strings.

A = [”!”, ”!”, ”!”, ”!”, ”A”]

Then we initialize the array C with 27 zeros. After that we loop through B and increment the values at C at corresponding index of the characters alphabet value.

C = [4, 1, 0, 0, ..., 0]

Then we loop through the C again and do the same thing that we do in integer sorting.

for i ← 2 to 27
do C[i] ← C[i] + C[i–1]
After that:

C = [4, 5, 5, 5, ..., 5]

Then we traverse the array A starting from the end go backwards until the start and update the result array B using the auxiliary array C.

A = [”!”, ”!”, ”!”, ”!”, ”A”]	C = [4, 5, 5, 5, ..., 5]
B = [””, ””, ””, ””, ”A”]	C = [4, 4, 5, 5, ..., 5]
B = [””, ””, ””, ”!”, ”A”]	C = [3, 4, 5, 5, ..., 5]
B = [””, ””, ”!”, ”!”, ”A”]	C = [2, 4, 5, 5, ..., 5]
B = [””, ”!”, ”!”, ”!”, ”A”]	C = [1, 4, 5, 5, ..., 5]
B = [”!”, ”!”, ”!”, ”!”, ”A”]	C = [0, 4, 5, 5, ..., 5]

Now the array F is sorted by their last characters. Since the counting sort is a stable sorting algorithm, the relative sorting of the same elements did not change.

Current F = [”MERT!!”, ”AYSU!!”, ”SELIN!”, ”ERDEM!”, ”DILARA”]

5^{th} Letters:

We get the 5^{th} characters of the strings.

A = [”!”, ”!”, ”N”, ”M”, ”R”]

Then we initialize the array C with 27 zeros. After that we loop through B and increment the values at C at corresponding index of the characters alphabet value.

C = [2, 0, ..., 1, 1, ..., 1, ..., 0]

Then we loop through the C again and do the same thing that we do in integer sorting.

for i ← 2 to 27
do C[i] ← C[i] + C[i–1]
After that:

C = [2, 2, ... 3, 4, ..., 5, ..., 5]

Then we traverse the array A starting from the end go backwards until the start and update the result array B using the auxiliary array C.

A = [”!”, ”!”, ”N”, ”M”, ”R”]	C = [2, 2, ... 3, 4, ..., 5, ..., 5]
B = [””, ””, ””, ””, ”R”]	C = [2, 2, ... 3, 4, ..., 4, ..., 5]
B = [””, ””, ”M”, ””, ”R”]	C = [2, 2, ... 2, 4, ..., 4, ..., 5]
B = [””, ””, ”M”, ”N”, ”R”]	C = [2, 2, ... 2, 3, ..., 4, ..., 5]
B = [””, ”!”, ”M”, ”N”, ”R”]	C = [1, 2, ... 2, 3, ..., 4, ..., 5]
B = [”!”, ”!”, ”M”, ”N”, ”R”]	C = [0, 2, ... 2, 3, ..., 4, ..., 5]

Now the array F is sorted by their 5^{th} characters.

Current F = [”MERT!!”, ”AYSU!!”,”ERDEM!”, ”SELIN!”, ”DILARA”]

4^{th} Letters:

We get the 4^{th} characters of the strings.

A = [”T”, ”U”, ”E”, ”I”, ”A”]

Then we initialize the array C with 27 zeros. After that we loop through B and increment the values at C at corresponding index of the characters alphabet value.

C = [0, 1, 0, 0, 0, 1, ..., 1, ..., 1, 1, ..., 0]

Then we loop through the C again and do the same thing that we do in integer sorting.

for i ← 2 to 27
do C[i] ← C[i] + C[i–1]
After that:

$$C = [0, 1, 1, 1, 1, 2, ..., 3, ..., 4, 5, ..., 5]$$

Then we traverse the array A starting from the end go backwards until the start and update the result array B using the auxiliary array C.

A = [”T”, ”U”, ”E”, ”I”, ”A”]	C = [0, 1, 1, 1, 1, 2, ..., 3, ..., 4, 5, ..., 5]
B = [”A”, ””, ””, ””, ””]	C = [0, 0, 1, 1, 1, 2, ..., 3, ..., 4, 5, ..., 5]
B = [”A”, ””, ”I”, ””, ””]	C = [0, 0, 1, 1, 1, 2, ..., 2, ..., 4, 5, ..., 5]
B = [”A”, ”E”, ”I”, ””, ””]	C = [0, 0, 1, 1, 1, 1, ..., 2, ..., 4, 5, ..., 5]
B = [”A”, ”E”, ”I”, ””, ”U”]	C = [0, 0, 1, 1, 1, 1, ..., 2, ..., 4, 4, ..., 5]
B = [”A”, ”E”, ”I”, ”T”, ”U”]	C = [0, 0, 1, 1, 1, 1, ..., 2, ..., 3, 4, ..., 5]

Now the array F is sorted by their 4^{th} characters.

Current F = [”DILARA”, ”ERDEM!”, ”SELIN!”, ”MERT!!”, ”AYSU!!”]

3^{th} Letters:

We get the 3^{th} characters of the strings.

$$A = [”L”, ”D”, ”L”, ”R”, ”S”]$$

Then we initialize the array C with 27 zeros. After that we loop through B and increment the values at C at corresponding index of the characters alphabet value.

$$C = [0, ..., 1, ..., 2, ..., 1, 1, ..., 0]$$

Then we loop through the C again and do the same thing that we do in integer sorting.

for i ← 2 to 27
do C[i] ← C[i] + C[i–1]
After that:

$$C = [0, ..., 1, ..., 3, ..., 4, 5, ..., 5]$$

Then we traverse the array A starting from the end go backwards until the start and update the result array B using the auxiliary array C.

A = [”L”, ”D”, ”L”, ”R”, ”S”]	C = [0, ..., 1, ..., 3, ..., 4, 5, ..., 5]
B = [””, ””, ””, ””, ”S”]	C = [0, ..., 1, ..., 3, ..., 4, 4, ..., 5]
B = [””, ””, ””, ”R”, ”S”]	C = [0, ..., 1, ..., 3, ..., 3, 4, ..., 5]
B = [””, ””, ”L”, ”R”, ”S”]	C = [0, ..., 1, ..., 2, ..., 3, 4, ..., 5]
B = [”D”, ””, ”L”, ”R”, ”S”]	C = [0, ..., 0, ..., 2, ..., 3, 4, ..., 5]
B = [”D”, ”L”, ”L”, ”R”, ”S”]	C = [0, ..., 0, ..., 1, ..., 3, 4, ..., 5]

Now the array F is sorted by their 3^{th} characters.

Current F = [”ERDEM!”, ”DILARA”, ”SELIN!”, ”MERT!!”, ”AYSU!!”]

2^{nd} Letters:

We get the 2^{nd} characters of the strings.

$$A = [”R”, ”I”, ”E”, ”E”, ”Y”]$$

Then we initialize the array C with 27 zeros. After that we loop through B and increment the values at C at corresponding index of the characters alphabet value.

$$C = [0, ..., 2, ..., 1, ..., 1, ..., 1, 0]$$

Then we loop through the C again and do the same thing that we do in integer sorting.

for i ← 2 to 27
do C[i] ← C[i] + C[i–1]
After that:

$$C = [0, ..., 2, ..., 3, ..., 4, ..., 5, 5]$$

Then we traverse the array A starting from the end go backwards until the start and update the result array B using the auxiliary array C.

A = [”R”, ”I”, ”E”, ”E”, ”Y”]	C = [0, ..., 2, ..., 3, ..., 4, ..., 5, 5]
B = [”, ”, ”, ”, ”Y”]	C = [0, ..., 2, ..., 3, ..., 4, ..., 4, 5]
B = [”, ”E”, ”, ”, ”Y”]	C = [0, ..., 1, ..., 3, ..., 4, ..., 4, 5]
B = [”E”, ”E”, ”, ”, ”Y”]	C = [0, ..., 0, ..., 3, ..., 4, ..., 4, 5]
B = [”E”, ”E”, ”I”, ”, ”Y”]	C = [0, ..., 0, ..., 2, ..., 4, ..., 4, 5]
B = [”E”, ”E”, ”I”, ”R”, ”Y”]	C = [0, ..., 0, ..., 2, ..., 3, ..., 4, 5]

Now the array F is sorted by their 2^{nd} characters.

Current F = [”SELIN!”, ”MERT!!”, ”DILARA”, ”ERDEM!”, ”AYSU!!”]

1^{st} Letters:

We get the 1^{st} characters of the strings.

A = [”S”, ”M”, ”D”, ”E”, ”A”]

Then we initialize the array C with 27 zeros. After that we loop through B and increment the values at C at corresponding index of the characters alphabet value.

C = [0, 1, ..., 1, 1, ..., 1, ..., 1, ..., 0]

Then we loop through the C again and do the same thing that we do in integer sorting.

for i ← 2 to 27
  do C[i] ← C[i] + C[i–1]

After that:

C = [0, 1, ..., 2, 3, ..., 4, ..., 5, ..., 5]

Then we traverse the array A starting from the end go backwards until the start and update the result array B using the auxiliary array C.

A = [”S”, ”M”, ”D”, ”E”, ”A”]	C = [0, 1, ..., 2, 3, ..., 4, ..., 5, ..., 5]
B = [”A”, ”, ”, ”, ”]	C = [0, 0, ..., 2, 3, ..., 4, ..., 5, ..., 5]
B = [”A”, ”, ”E”, ”, ”]	C = [0, 1, ..., 2, 2, ..., 4, ..., 5, ..., 5]
B = [”A”, ”D”, ”E”, ”, ”]	C = [0, 1, ..., 1, 2, ..., 4, ..., 5, ..., 5]
B = [”A”, ”D”, ”E”, ”M”, ”]	C = [0, 1, ..., 1, 2, ..., 3, ..., 5, ..., 5]
B = [”A”, ”D”, ”E”, ”M”, ”S”]	C = [0, 1, ..., 1, 2, ..., 3, ..., 4, ..., 5]

Now the array F is sorted by their 1^{st} characters.

Current F = [”AYSU!!”, ”DILARA”, ”ERDEM!”, ”MERT!!”, ”SELIN!”]

Now as a last step we remove the joker characters from end of the strings.

Result = [”AYSU”, ”DILARA”, ”ERDEM”, ”MERT”, ”SELIN”]

c) Analysis

We can analyze the modified radix sort, part by part.

Finding the longest string in the array F takes $\Theta(n)$ time where n is equal to the number of elements in the array F. Adding joker characters at the end of the strings takes $\Theta(n*d)$ time where d is equal to the longest string in the array F. In total the first step takes $\Theta(n + n*d) = \Theta(n*d)$

Also we need to analyze the counting sort that we used in the radix sort. Initializing the character array A takes $\Theta(n)$ time. Initializing the array C takes $\Theta(k)$ time where k is the number of distinct characters in alphabet which is equal to 27 in our example. Counting the characters in the array A and modifying the array C takes $\Theta(n)$ time. Summing the elements in the array C takes $\Theta(k)$ time. Creating the output array B needs to traverse the array A with only one loop and it takes $\Theta(n)$ time. In total counting sort takes $\Theta(n + k + n + n) = \Theta(n + k)$

We need to call the counting sort for every characther in the longest string in the array F. This makes the running complexity equal to $\Theta(d * (n + k))$

After that we need to remove the joker characthers from the strings. This takes same amount of time with adding them and it is $\Theta(n * d)$.

If we sum up all of these times. We get in total:

$$T(n) = \Theta(n*d + d*(n+k) + n*d) = \Theta(d*(n+k))$$

For our example we use 27 letters for sorting and k is equal to 27. We can drop this constant from our result and we can get:

$$T(n) = \Theta(d*n)$$

References

Khanacademy, reference for the recursion tree in Problem 1, (a)