CS/ECE 374 Spring 2025

Heng An Cheng (hacheng2@illinois.edu)

Homework 5 Problem 1

Solution:

(a) Given TMs M_1, M_2 that decide languages L_1 and L_2:

- A TM that decides $L_1 \cup L_2$: On input x, run M_1 and M_2 on x, and accept if either accepts.
- A TM that decides $L_1 \cap L_2$: On input x, run M_1 and M_2 on x, and accept if both accept.

(b) Given TMs M_1, M_2 that decide languages L_1 and L_2:

- A TM that decides $L_1 L_2$: On input x, for each $|x| + 1$ ways to divide $x = yz$, run M_1 on y and M_2 on z, and accept if both accept. Else reject.

(c) Given TM Mthat decide languages L :

- A TM that decides L^*: On input x, if $x = \epsilon$, then accept. Else, for each $2^{|x|+1}$ ways to divide $x = w_1 w_2 \dots w_i$, run M on w_i, and accept if all accept. Else reject.

(d) Using the same idea from (c), we can divide the string x into $x = w_1 w_2 \dots w_i$ and run it on the machine M^*.

The process of M^* is as follows:

For $i = 0, 1, 2, \dots$

- Run input $w_1, w_2, \dots, w_i$ respectively on M^* for i steps.
- If one of them is accepted, then halt and accept.
- Else, increase i by 1 and repeat the loop.

Each iteration only has finite simulations. However, there is no upper bound for i, so the loop will consider all possible w_i. So if there is some string x which is accepted by M^*, our Function will eventually simulate M^* on x for enough steps to see it halt.

■

CS/ECE 374 Spring 2025 Heng An Cheng (hacheng2@illinois.edu)

Homework 5 Problem 2

Solution:

Algorithm 1 Merge Sort using Divide and Conquer

(a)

```
function MERGE(a, b)
    i ← 0, j ← 0
    sorted_arr ← []
    while i < len(a) and j < len(b) do
        if a[i] < b[j] then
            append a[i] to sorted_arr
            i ← i + 1
        else
            append b[j] to sorted_arr
            j ← j + 1
        end if
    end while
    while i < len(a) do
        append a[i] to sorted_arr
        i ← i + 1
    end while
    while j < len(b) do
        append b[j] to sorted_arr
        j ← j + 1
    end while
    return sorted_arr
end function
```

Algorithm 2 Divide and Conquer Merge for k Sorted Arrays

```
function MERGE_K_ARRAYS(arr, l, r)
    if l = r then
        return arr[l]
    end if
    mid ← (l + r)/2
    leftSorted ← MERGE_K_ARRAYS(arr, l, mid)
    rightSorted ← MERGE_K_ARRAYS(arr, mid + 1, r)
    return MERGE(leftSorted, rightSorted)
end function
```

Assume $arr = A_1, A_2, \ldots, A_k$

Then $Sorted\_Arr = Merge\_k\_Arrays(arr, 1, k)$

Time Complexity Analysis

The recursion depth is $O(\log k)$ since we repeatedly split k into halves.

Each level of recursion merges all $N = nk$ elements in $O(N)$.

The total time complexity is therefore:

$$O(N \log k)$$

(b) Just run the $Check$ function with the given array arr

Algorithm 3 Select the k-th largest element and check array conditions

```
function SELECT(arr, k)                                    ▷ Pick the k-th largest number
    chunks ← [arr[i : i+5] for i in range(0, len(arr), 5)]
    sorted_chunks ← [sorted(chunk) for chunk in chunks]
    medians ← [chunk[len(chunk)//2] for chunk in sorted_chunks]
    if len(medians) ≤ 5 then
        pivot ← sorted(medians)[len(medians)//2]
    else
        pivot ← Select(medians, len(medians)//2)
    end if
    left ← {x ∈ arr | x < pivot}
    right ← {x ∈ arr | x ≥ pivot}
    r ← len(right)
    if k == r then
        return pivot
    else if k < r then
        return Select(right, k)
    else
        return Select(left, k - r)
    end if
end function
function CHECK(arr)
    l ← len(arr)
    top_threshold ← Select(arr, ⌊l × 0.02⌋)
    bottom_threshold ← Select(arr, ⌊l × 0.2⌋)
    top_sum ← ∑_{i∈arr, i≥top_threshold} i
    bottom_sum ← ∑_{i∈arr, i≤bottom_threshold} i
    return (top_sum > bottom_sum × 10)
end function
```

Time Complexity Analysis

The $Select$ function implements the Median of Medians algorithm, which has a worst-case time complexity of $O(n)$. The steps of $Select$ are as follows:

- Dividing the array into groups of 5 takes $O(n)$.
- Sorting each group of 5 takes $O(n)$.
- Finding the medians of the groups as pivot takes $O(n)$.
- Partitioning the array around the pivot takes $O(n)$.
- Recursing on the left or right partition results in a recurrence $T(n) = T(0.7n) + O(n)$, which also solves to $O(n)$.

Therefore, the time complexity of the $Select$ function is $O(n)$.

The `Check` function calls $Select$ twice and computes sums over subsets of the array, each of which takes $O(n)$. Hence, the time complexity of $Check$ is:

$$O(n)$$

Thus, the overall time complexity of the algorithm is $O(n)$.

(c) Use similiar idea from (b), we create new functions $MultiSelect$, $Cal\_total\_earnings$

Algorithm 4 Compute the total earnings of the top $\alpha_i\%$ of earners for $1 \leq i \leq k$

```
function MultiSelect(arr, S, sums)
    chunks ← [arr[i : i+5] for i in range(0, len(arr), 5)]
    sorted_chunks ← [sorted(chunk) for chunk in chunks]
    medians ← [chunk[len(chunk)//2] for chunk in sorted_chunks]
    if len(medians) ≤ 5 then
        pivot ← sorted(medians)[len(medians)//2]
    else
        pivot ← Select(medians, len(medians)//2)
    end if
    left ← {x ∈ arr | x < pivot}
    right ← {x ∈ arr | x ≥ pivot}
    r ← len(right)
    for each s_i in S do
        if s_i == r then
            sums[i-1] ← sum(x in arr where x ≥ pivot)
            Remove s_i from S
        else if k < r then
            Add m_i to S_right
        else
            Add s_i − r to S_left
        end if
    end for
    MultiSelect(right, S_right, sums)
    MultiSelect(left, S_left, sums)
end function
function Cal_t otal_e arnings(arr, α[1..k])
    n ← len(arr)
    S ← [α[i] × n | i ∈ α]
    sums ← [0 | i ∈ α]
    MultiSelect(arr, S, sums)
    return sums
end function
```

Time Complexity Analysis

The $MultiSelect$ function implements the Median of Medians algorithm, which has a worst-case time complexity of $O(n)$. The steps of $MultiSelect$ are as follows:

- Dividing the array into groups of 5 takes $O(n)$.
- Sorting each group of 5 takes $O(n)$.
- Finding the medians of the groups as pivot takes $O(n)$.
- Partitioning the array around the pivot takes $O(n)$.
- The recursion depth is $O(\log k)$, as each recursive call assigns some of the k queries to subarrays (e.g., elements less than or greater than the pivot), reducing the number of remaining queries in a manner similar to a binary search tree.

Therefore, the time complexity of the $MultiSelect$ function is $O(nlogk)$.

The `Check` function calls $MultiSelect$ once , which takes $O(nlogk)$. Hence, the time complexity of $Cal\_total\_earnings$ is:

$$O(nlogk)$$

Thus, the overall time complexity of the algorithm is $O(nlogk)$.

■