Faculty of Computers and Information
Dept. of Computers Science
Third year

Analysis and Design of Algorithms
Fall 2023/2024
Homework 3
Due: Saturday, Dec. 21, 2024 (12:00 pm)

A.[6.0 Points] Use substitution (Iterative) method to give tight asymptotic bound for the following recurrence relation:

$T(n)$=7 $T(n2)$+ c $(n2)$, for n > 1 and T(1) =1

Answer
T(n) = 7T(n/2) + cn^2

= 7(7T(n/4) + c(n/2)^2) + cn^2

= 7^2T(n/4) + 7cn^2/4 + cn^2

= 7^2(7T(n/8) + c(n/4)^2) + 7cn^2/4 + cn^2

= 7^3T(n/8) + 7^2cn^2/16 + 7cn^2/4 + cn^2

Notice a pattern? Equation
T(n) = 7^kT(n/2^k) + ∑[i=0 to k-1] (7^i * c * n^2 / 2^(2i))

Now, let's find the value of k that makes n/2^k = 1
Since n = 2^k, we have:

$n/2^k = 1 \Rightarrow k = \log_2(n)$
Substituting $k = \log_2(n)$ into the above equation, we get:

$T(n) = 7^{\log_2(n)}T(1) + \sum[i=0 \text{ to } \log_2(n)-1] (7^i * c * n^2 / 2^{(2i)})$

Simplifying the above equation, we get:

$T(n) = n^{\log_2 7} + \sum[i=0 \text{ to } \log_2(n)-1] (7^i * c * n^2 / 2^{(2i)})$

Using the formula for geometric series, we can simplify the summation:

$\sum[i=0 \text{ to } \log_2(n)-1] (7^i * c * n^2 / 2^{(2i)}) = c * n^2 * (1 - (7/4)^{\log_2(n)}) / (1 - 7/4)$

Simplifying the above equation, we get:

$T(n) = n^{\log_2 7} + c * n^2 * (1 - (7/4)^{\log_2(n)}) / (1 - 7/4)$

Since $(7/4)^{\log_2(n)}$ grows faster than n^2, we can ignore the lower-order term n^2. Thus, we get:

$T(n) = \Theta(n^{\log_2 7})$

B.[11 Points] Describe O(n lg m)-time complexity Alg, for merging m sorted lists of objects into one sorted list, such that n represents the total number of objects in all the input lists.

Steps of the Algorithm

- Create a max-heap of size m
- Each element in the heap will be a tuple containing:
  - The value of the first object in a list.
  - The index of the list that the object came from.
  - The index of the object in the list.

This ensures we retrieve the maximum object across all lists.

Heap the Initial Elements:

- Add the first element from each of the m lists into the heap. The heap will now contain at most m elements.

- Build the heap, which takes $O(m)$ time

Extract and Add to the Result List:

- Initialize an empty list result to store the merged objects.
- Repeat the following n times (once for each object across all lists):
    1. Extract the maximum object from the heap. This takes $O(\log m)$ time.
    2. Append the extracted object to the result list.

Output the Merged List:

- Once all n objects are extracted and added to the result, the merged list is complete.

Time Complexity Analysis:

- **Heap Initialization:** Building the heap with the first mmm elements takes $O(m)$
- **Main Loop:** The loop runs n iterations (one for each object):
    - Extracting from the heap: $O(\log m)$.
    - Inserting into the heap: $O(\log m)$O.
    - Total cost per iteration: $O(\log m)$.
    - Total cost for n iterations: $O(n\log m)$.

Thus, the overall time complexity is:

$$O(m+n\log m)\approx O(n\log m)(\text{as } n\gg m \text{ in most cases})$$

example

Input:

We have m=3m = 3m=3 sorted lists:

L1=[1,4,7]

L2=[2,5,8]

L3=[3,6,9]

The total number of objects is n=9n

Step 1: Initialize the Heap

Insert the first element of each list into the heap along with the list index and the position of the element in the list:
Heap:

[(-7, 0, 2), (-8, 1, 2), (-9, 2, 2)]

Step 2: Extract Max and Add to Result

- Extract the maximum (smallest negative) element:$(-9,2,2)$.
  - Append 9 to the result list.
  - Push the previous element from L3(6) into the heap.
  - Heap:
  - [(-8, 1, 2), (-7, 0, 2), (-6, 2, 1)]
  - Result:
  - [9]
- Extract the maximum:$(-8,1,2)$.
  - Append 8 to the result list.
  - Push the previous element from L2(5) into the heap.
  - Heap:
  - [(-7, 0, 2), (-6, 2, 1), (-5, 1, 1)]
  - Result:
  - [9,8]
- Extract the maximum:$(-7,0,2)$.
  - Append 7 to the result list.
  - Push the previous element from
  - $L1(4)$ into the heap.
  - Heap:
  - [(-6, 2, 1), (-5, 1, 1), (-4, 0, 1)]
  - Result:
  - [9,8,7]

Step 3: Continue Extracting

- Result:

  [9,8,7,6,5,4,3,2,1]

[3.0 Marks] Analyze Huffman algorithm to find its time complexity.

step-by-step analysis:

1. Build Frequency Table: O(n)
2. Build Priority Queue: O(n)
3. Build Huffman Tree: O(n log n)
   - We perform n-1 extract-min operations, each taking O(log n) time.
4. Generate Huffman Codes: O(n)

Overall time complexity: O(n) + O(n) + O(n log n) + O(n) = O(n log n)

Therefore, the time complexity of the Huffman algorithm is O(n log n).

D.[8.0 Marks] Let G = (V, E) be a simple graph with n vertices and the weight of every edge of G is equal to one. Compute in detail the weight of MST of G?

To compute the weight of the Minimum Spanning Tree (MST) of graph G, we can use the following steps:

Since the weight of every edge in G is equal to 1, the MST will be a spanning tree with the minimum number of edges.

A spanning tree of a graph with n vertices has exactly n-1 edges.

Therefore, the weight of the MST of G will be:

Weight of MST = Number of edges in MST= n - 1

Since the weight of every edge is 1, the total weight of the MST is simply the number of edges in the MST.

Therefore, the weight of the MST of G is n - 1.

E.[11.0 Marks] If S is an unsorted array of k integers (any element of M could be either positive or negative integer), design O(k lg k) worst-case time algorithm that searches two numbers x, y M, x y, such that | x + y | is the minimum among all pairs in M.

step-by-step algorithm to find two numbers x, y in an unsorted array S of k integers such that |x + y| is minimized:

Algorithm: MinSumPair

Step 1: Sort the Array

- Sort the array S in non-decreasing order using a sorting algorithm like Merge Sort or Quick Sort.
- Time complexity: O(k lg k)

Step 2: Initialize Pointers

- Initialize two pointers, `left` and `right`, to the start and end of the sorted array, respectively.

Step 3: Find MinSumPair

- Initialize `min_sum` to infinity and `min_pair` to null.
- While `left` < `right`:
  1. Calculate `sum` = `S[left]` + `S[right]`.
  2. If `|sum|` < `|min_sum|`, update `min_sum` and `min_pair`.
  3. If `sum` < 0, increment `left` to increase the sum.
  4. Else, decrement `right` to decrease the sum.

Step 4: Return MinSumPair

- Return `min_pair` containing the two numbers x, y that minimize |x + y|.

Time Complexity Analysis

- Sorting the array: O(k lg k)
- Finding MinSumPair: O(k)
- Total time complexity: O(k lg k) + O(k) = O(k lg k)

This algorithm ensures that we find the pair of numbers that minimizes |x + y| in **O(k lg k) time complexity.**

F.[11 Points] Discuss how you can compute in_degree and out_degree of the nodes of a graph given when it is represented by an adjacency list.

Computing Out-Degree

The out-degree of a node is the number of edges that originate from it. In an adjacency list representation, each node has a list of its neighboring nodes.

To compute the out-degree of a node:

1. Iterate through the adjacency list of the node.
2. Count the number of neighboring nodes.
3. The count is the out-degree of the node.

Computing In-Degree

The in-degree of a node is the number of edges that point to it. To compute the in-degree, we need to iterate through the adjacency lists of all nodes.

To compute the in-degree of a node:

1. Initialize the in-degree count to 0.
2. Iterate through the adjacency lists of all nodes.
3. For each neighboring node, increment the in-degree count of the corresponding node.
4. The final count is the in-degree of the node.

Example

Suppose we have a graph with three nodes (A, B, C) represented by the following adjacency list:

A -> [B, C]

B -> [A, C]

C -> [A, B]

To compute the in-degree and out-degree of each node:

- Node A: Out-degree = 2 (B, C), In-degree = 2 (B, A)
- Node B: Out-degree = 2 (A, C), In-degree = 2 (A, C)
- Node C: Out-degree = 2 (A, B), In-degree = 2 (A, B)