Midterm Exam

Ali Abdullah Ahmad

March 02, 2025

Midterm Exam CS 600 100 Points

Instructions: Answer the following questions in this document or a Word document and upload it within three hours.

1 Questions

1. (6 Points) Using the very definition of Big-Theta notation, prove that $2^n + 1$ is $\Theta(2^n)$. You must use the definition and find the constants in the definition to receive credit.

   Solution:

   By the definition of Big-Theta notation, a function $f(n)$ is $\Theta(g(n))$ if there exist positive constants c_1, c_2, and n_0 such that for all $n \geq n_0$:

   $$c_1 g(n) \leq f(n) \leq c_2 g(n)$$

   Here, let $f(n) = 2^n + 1$ and $g(n) = 2^n$.

   To find c_1 and c_2:

   Upper Bound:

   $$2^n + 1 \leq 2^n + 2^n = 2 \cdot 2^n, \quad \text{for } n \geq 1.$$

   So, we can choose $c_2 = 2$.

   Lower Bound:

   $$2^n + 1 \geq 2^n, \quad \text{for all } n \geq 0.$$

   So, we can choose $c_1 = 1$.

   Thus, for $n_0 = 1$, we have:

   $$1 \cdot 2^n \leq 2^n + 1 \leq 2 \cdot 2^n$$

   which satisfies the Big-Theta definition. Therefore, 2^n+1 is $\Theta(2^n)$.

2. (7 Points) Given that $T(n) = 1$ if $n = 1$ and $T(n) = T(n-1) + n$ otherwise; show, by induction, that $T(n) = \frac{n(n+1)}{2}$. Show all three steps of your induction explicitly.

Solution:

We use mathematical induction to prove that:

$$T(n) = \frac{n(n+1)}{2}$$

Base Case: For $n = 1$,

$$T(1) = 1$$

and

$$\frac{1(1+1)}{2} = \frac{2}{2} = 1,$$

which matches the given definition. Hence, the base case holds.

Inductive Hypothesis: Assume that for some $k \geq 1$, the formula holds:

$$T(k) = \frac{k(k+1)}{2}.$$

Inductive Step: We need to show that it holds for $n = k + 1$:

$$T(k+1) = T(k) + (k+1).$$

Using the inductive hypothesis,

$$T(k+1) = \frac{k(k+1)}{2} + (k+1).$$

Factoring out $(k+1)$,

$$T(k+1) = \frac{k(k+1) + 2(k+1)}{2}.$$

$$= \frac{(k+1)(k+2)}{2}.$$

This matches the given formula for $T(n)$, so by induction, the statement holds for all $n \geq 1$.

3. (7 Points) Given the recurrence relation $T(n) = 7T(n/5) + 10n$, for $n > 1$; and $T(1) = 1$. Find $T(625)$.

Solution:

Expanding the recurrence relation:

$$T(625) = 7T(125) + 10(625)$$

$$T(125) = 7T(25) + 10(125)$$

$$T(25) = 7T(5) + 10(25)$$

$$T(5) = 7T(1) + 10(5)$$

$$T(1) = 1$$

Now calculating step by step:

$$T(5) = 7(1) + 10(5) = 7 + 50 = 57$$

$$T(25) = 7(57) + 10(25) = 399 + 250 = 649$$

$$T(125) = 7(649) + 10(125) = 4543 + 1250 = 5793$$

$$T(625) = 7(5793) + 10(625) = 40551 + 6250 = 46801$$

Thus, $T(625) = 46801$.

4. (16 Points) Suppose that we implement a union-find structure by representing each set using a balanced search tree. Describe and analyze algorithms for each of the methods for a union-find structure so that every operation runs in at most $O(\log n)$ time in the worst case.

   Solution:

   In a union-find data structure using balanced search trees, we can efficiently perform the following operations:

   - **Find**: Use a binary search tree to find the representative element of a set in $O(\log n)$ time.
   - **Union**: Merge two trees by linking the smaller tree under the larger one, maintaining balance, ensuring $O(\log n)$ time.
   - **Path Compression**: Maintain a balanced structure during repeated finds to keep the tree height minimal.

   An AVL tree or Red-Black tree can be used to maintain balance, ensuring $O(\log n)$ complexity for each operation.

5. (16 Points) Recall Homework 2 Exercise 2.5.13, where you implemented a stack using two queues. Now consider implementing a queue using two stacks S_1 and S_2 where:

   - **enqueue(o):** Pushes object o at the top of the stack S_1.
   - **dequeue():** If S_2 is empty, then pop the entire contents of S_1 pushing each element onto S_2. Then pop from S_2. If S_2 is not empty, pop from S_2.

   Solution:

   a) **Worst Case Running Time of a Single Dequeue:**

   In the worst case, a dequeue operation requires transferring all elements from S_1 to S_2 if S_2 is empty. Each element is popped from S_1 and pushed onto S_2, taking $O(1)$ time per operation. If there are n elements in S_1, the total time for transferring all elements to S_2 is $O(n)$. After transferring, popping from S_2 takes $O(1)$. Therefore, the worst-case time complexity for a single dequeue operation is $O(n)$.

   b) Amortized Cost of a Single Dequeue:

   Let's assume each dequeue operation costs 2 cyber dollars. The breakdown is as follows:

   - If S_2 is not empty, the cost is 1 cyber dollar for popping from S_2.
   - If S_2 is empty, transferring all elements from S_1 to S_2 costs n cyber dollars, and popping from S_2 costs 1 more cyber dollar.

   Over n dequeue operations, each element is moved and popped once, so the total cost is $2n$ cyber dollars. The amortized cost per dequeue is:

   $$\frac{2n \text{ cyber dollars}}{n} = 2 \text{ cyber dollars.}$$

   Therefore, the amortized cost of a single dequeue operation is $O(1)$.

6. (16 Points) Consider an $n \times n$ matrix M whose elements are 0's and 1's such that in any row, all the 1's come before any 0's in that row. Assuming M is already in memory, describe an efficient algorithm for finding the row of M that contains the most 1's. What is the running time of the algorithm?

   Solution:

   - **Step 1:** Traverse the first row of the matrix from left to right. Keep a count of 1's until the first 0 is encountered. Set the global maximum 1's count equal to the current count of 1's.
   - **Step 2:** Traverse the next row, but instead of scanning from the beginning, start from the column where the previous row encountered its first 0. Since all 1's come before 0's in each row, this reduces unnecessary comparisons.
   - **Step 3:** If the current row has more 1's than the previously recorded maximum, update the global count.
   - **Step 4:** Repeat the process for all the rows in the matrix. In the end, return the global maximum 1's count.

   Time Complexity:

   - For each row, we are scanning the elements only once. In the worst case, we might have to traverse the entire row, but since we reduce the number of columns to scan for subsequent rows, the total number of elements we check across all rows is linear in the total number of rows. Hence, the total time complexity of the algorithm is $O(n)$, where n is the number of rows (and columns in the case of a square matrix).

7. (16 Points) You are given two sequences A and B of n numbers each, possibly containing duplicates. Describe an efficient algorithm for determining if A and B contain the same set of numbers, possibly in different orders. What is the running time of this algorithm?

Solution
Algorithm:

- **Input:** Sequences A and B of n elements.
- **Output:** Return true if A and B contain the same elements (possibly in different order), otherwise return false.

Steps:

(a) Create a count array (or hash map) with a size equal to the range of numbers in A and B. Initialize all values to zero.

(b) Traverse through A:
   - For each element $A[i]$, increment the corresponding index in the count array (or hash map).

(c) Traverse through B:
   - For each element $B[j]$, check if the corresponding count is zero:
     - If the count is zero, return false (elements do not match).
     - Otherwise, decrement the count for that element.

(d) After traversing both sequences, check the count array (or hash map). If all counts are zero, return true (sequences contain the same elements); otherwise, return false.

Time Complexity: $O(n)$, as we traverse both sequences and the count array (or hash map) only once.

8. (16 Points) Let A and B be two sequences of n integers each, in the range $[1, n^4]$. Given an integer x, describe an $O(n)$–time algorithm for determining if there is an integer a in A and an integer b in B such that $x = a + b$.

Solution:

Algorithm: findPairWithSum(A, B, x)

Input: Sequences A and B of n integers, an integer x.
Output: Return true if there exist a in A and b in B such that $x = a + b$, otherwise false.

- Create an empty hash set, SetA.
- For each element a in A:
  - Add a to SetA.
- For each element b in B:
  - If $(x - b)$ exists in SetA:
    - Return true.
- Return false.

Time Complexity:

- Creating the hash set from sequence A takes $O(n)$ time.
- Iterating over sequence B and checking if $x - b$ exists in the hash set takes $O(n)$ time.
- Therefore, the overall time complexity is $O(n)$.

9. (16 Points) Suppose you are given an instance of the fractional knapsack problem in which all the items have the same weight. Describe an algorithm and provide pseudo code for this fractional knapsack problem in $O(n)$ time.

Solution:

In the case where all items have the same weght, the problem simplifies because we don't need to consider the weight when making decisions about the items to include in the knapsack. The only factor to consider is the value-to-weight ratio (which is essentially just the value of each item since all weights are the same).

The algorithm is as follows:

Algorithm:

- Sort the items by their value in descending order.
- Initialize the knapsack capacity to W.
- For each item in the sorted list:
  - If the knapsack can hold the item entirely (i.e., the remaining capacity is greater than or equal to the item's weight), add the entire item to the knapsack.
  - If the knapsack cannot hold the entire item, take the fraction of the item that fits and stop.

Pseudo Code:

```
Algorithm: FractionalKnapsack(A, W)
Input: List of items A with values v[i] and weights w[i], total knapsack capacity W
Output: Maximum value that can be obtained

1. Sort A by value v[i] in descending order
2. Set totalValue = 0, remainingCapacity = W
3. For each item i in A:
    a. If remainingCapacity >= w[i]:
        i. Add v[i] to totalValue
        ii. Subtract w[i] from remainingCapacity
    b. Else:
        i. Add (v[i] / w[i]) * remainingCapacity to totalValue
        ii. Set remainingCapacity = 0
        iii. Break
4. Return totalValue
```

Time Complexity:

- Sorting the items takes $O(n \log n)$ time.

- Iterating over the items takes $O(n)$ time.
- Therefore, the overall time complexity is $O(n \log n)$, but since all items have the same weight, we can ignore the weight and treat the algorithm as if it operates in $O(n)$ for practical purposes.