COSC 3320 Practice Problems Set 1: Solutions.

October 9, 2023

Problem 1

Solve the following recurrences by finding a Big-Theta form for $T(n)$. Justify your work. You may assume the existence of base cases $T(0) = T(1) = 1$. You may use the master theorem if applicable, or mathematical induction.

$$T(n) = T\left(\frac{n}{2}\right) + 5$$
$$T(n) = 7T\left(\frac{n}{2}\right) + 10n^2$$
$$T(n) = 2T(n-1) + 1$$
$$T(n) = 2T\left(\frac{n}{8}\right) + \sqrt{n}$$

Note on the Master Theorem

Recall that the Master Theorem can be applied to a recurrence:

$$T(n) = rT\left(\frac{n}{c}\right) + f(n)$$

We consider the value $\log_c(r)$, and compare the asymptotic behavior of $f(n)$ to $n^{\log_c(r)}$. It yields three cases:

- Case 1: When $f(n)$ dominates $n^{\log_c(r)}$ asymptotically (equivalent to saying that $n^{\log_c(r)} = \mathcal{O}(f(n))$ but that $n^{\log_c(r)} \neq \Theta(f(n))$, we have $T(n) = \Theta(f(n))$
- Case 2: When they are asymptotically equivalent ($f(n) = \Theta(n^{\log_c(r)})$), we have $T(n) = \Theta(f(n)\log(n))$
- Case 3: When $n^{\log_c(r)}$ dominates $f(n)$ (equivalent to saying ($f(n) = \mathcal{O}\big(n^{\log_c(r)}\big)$ but that $f(n) \neq \Theta(n^{\log_c(r)})$), we have that $T(n) = \Theta(n^{\log_c(r)})$

Solution. Now, to solve the problems. The first, second, and fourth recurrences can be solved using the Master Theorem as described above.

For the first reccurence, $r = 1, c = 2$, so $\log_c(r) = 0$. This means that $f(n)$, which is $\mathcal{O}(1)$, is asymtotically equivalent to $n^{\log_c(r)} = O(1)$ leading to Case 2, and $T(n) = \Theta(\log(n))$.

For the second recurrence, $r = 7, c = 2$, so $\log_c(r)$ is an irrational number between 2 and 3. Since $f(n) = \mathcal{O}\big(n^2\big)$, this means that we are in Case 3 as $2 < \log_2(7)$, and therefore $T(n) = \Theta(n^{\log_2(7)})$

For the third recurrence, we are unable to apply the Master Theorem and must examine the recurrence directly. Notice:

$$T(n) = 2T(n-1) + 1$$

can have the $T(n-1)$ replaced with $2T(n-2)+1$ as it is equivalent, yielding:

$$T(n) = 2(2T(n-2)+1) + 1$$

If you continue this process until we reach a base case, you will get:

$$T(n) = 2^{n-1}T(1) + 1 + 2 + 4 + \ldots + 2^{n-2}$$

It is a mathematical fact that $1 + 2 + \ldots + 2^{n-2} = 2^{n-1} - 1$. So we have:

$$T(n) = 2^{n-1}T(1) + 2^{n-1} - 1$$

but we can choose some constants $d, c > 0, d < T(1) < c$ such that

$$d2^n < T(n) < c2^n$$

for all suitably suitably large n. imply that $T(n) = \Theta(2^n)$.

For the fourth recurrence, $r = 2, c = 8$, so $\log_c(r) = \frac{1}{3}$. Since $f(n) = \mathcal{O}(\sqrt{n})$, this is dominating $n^{\log_c(r)} = n^{1/3}$, so we are in Case 1. We therefore have $T(n) = \Theta(\sqrt{n})$

□

Problem 2

You are given n stones (assume that n is a power of 2) each having a distinct weight. You are also given a two-pan balance scale (no weights are given). For example, given two stones, you can use the scale to compare which one is lighter by placing the two stones on the two different pans. The goal is the find the heaviest and the lightest stone by using as few weighings as possible. Give a divide and conquer strategy that uses $\frac{3n}{2} - 2$ weighings. Justify algorithm correctness and the number of steps taken.

Solution. To design a divide-and-conquer algorithm, we must first find a way to divide the problem into subproblems in such a way, such that, given a recursive application of the algorithm to the subproblems that yields valid solutions to the subproblems, we can efficiently calculate a solution to the whole problem from the solutions to subproblems. This, plus some direct application to base cases, are sufficient to define the algorithm.

The algorithm **WEIGH** is defined recursively:

- Divide our stones into two sets of size $n/2$: because n is a power of 2, this is always possible for n and every $n/2^k > 1$. The only exception is $n = 1$: we immediately return the single stone as both the heaviest and lightest.
- Perform **WEIGH** on each of the two sets recursively. This yields h_1, l_1, h_2, l_2 the heaviest and lightest stones from each set.
- Weigh h_1 against h_2, and pick the heavier. Weigh l_1 against l_2, and pick the lighter.
- Return the stones picked in the previous step

We shall show that this strategy requires $\frac{3n}{2} - 2$ weighings when $n > 1$ a power of 2.

To show the correctness of **WEIGH**, we may use induction. It is easily seen for $n = 1$ and $n = 2$, Our induction hypothesis is that, given we have it true for $n < k$, it must be true for $n = k$. Since we have the subproblems of size $n/2 < k$, by our induction hypothesis **WEIGH** is correct on each half, thus giving us the heaviest and lightest stones in each half. It must be the case that one of the two heavier stones is heaviest overall, and one of the lightest stones is lightest overall, as the stone that is heaviest/lightest overall is the heaviest/lightest in the half it inhabits. Therefore, the algorithm finds the heaviest and lightest overall, proving the correctness for $n = k$, and by induction, for all n a power of 2.

To show the number of weighings, we can also use induction. When $n = 2$, only one is weighing is necessary, and $3n/2 - 2 = 1$. Supposing it is true for $n < k$, by the algorithm. Then it takes $3n/4 - 2$ weighings for each of the $n/2 < k$ halves of the input, plus an additional 2 weighings. Plugging in, we have $2\left(\frac{3n}{4} - 2\right) + 2$, which is equal to $\frac{3n}{2} - 2$. □

Problem 3

You are given an $n \times n$ matrix represented as a 2-dimensional array $A[1 \ldots n][1 \ldots n]$ (there are totally n^2 elements). Each row of A is sorted in increasing order and each column is sorted in increasing order. Your goal is to find whether some given element x is in A or not. Note that you can do only comparisons between elements (no hash table, or any other set data structures). Give an algorithm that takes $\mathcal{O}(n)$ comparisons.

[Hint: try to start on one some corner, and eliminate one row/column per move].

Solution. A correct solution with $\mathcal{O}(n)$ runtime is to start at the top-right corner, and compare the value x to the value of A at this corner. These point have the virtue that we can eliminate a row or column in the first step. In the case of the top-right corner $y = A[n][1]$, if $x < y$, we have eliminated the right-most column as all values of that column are greater than y. If we have $x > y$, we eliminate the top-most row as all remaining values of that column are less that y. We can then progress to the left, or down, from the current location, and we've reduced the problem to searching a smaller rectangle with our current location as the top-right corner. This repeats until we either locate the number, or until we can no longer move in the direction we need to move, in which case we know with certainty that the number is not in the matrix.

Description as an iterative process:

- Let $y \leftarrow A[i][j]$
- If $x = y$, return true.
- If $x < y$, replace j with $j - 1$, unless $j = 1$, in which case return false.
- If instead $x > y$, replace i with $i + 1$ unless $i = n$, in which case return false.
- Repeat the previous steps as needed. It is guaranteed to terminate in no more than $2n - 2$ steps as either i increases or j decreases every iteration.

This process works because each iteration eliminates all columns/rows to the right/above the current (i, j). We only reach $A[1][n]$ when we have eliminated all remaining rows/columns and thus no solution can exist outside of that point.

An example:

1	3	4	8
5	7	15	25
6	10	16	26
9	11	19	28

Suppose we are trying to find 6. We start at 8. $6 < 8$, so we eliminate the column at 8 (25, 26, and 28 removed from consideration), and move to the left. We compare at 4. $6 > 4$, so we eliminate the row at 4 (1 and 3 are removed) and move down. We consider 15. $6 < 15$, so we eliminate the column (16, 19 removed) and move to the left. $6 < 7$, so we eliminate the column at 7 (10, 11 removed) and move to the left. $6 > 5$, so we move down (no values to eliminate) We then arrive at 6 and terminate.

If instead of 6, we try to find 14, which is not in the table, we would in turn visit: 8, 25, 15, 7, 10, 11, and then return false.

There is an equivalent alternative by starting in the bottom-left corner and moving up or to the right as well. □

Problem 4

You are given an array A of n integers, both positive and negative. The maximum possible sum of contiguous elements of the array is the maximum value of $Sum(i, j) = A[i] + A[i + 1] + \ldots A[j - 1] + A[j]$ over any i, j satisfying $0 \leq i \leq j < n$. For example, the array $[1, -5, 7, 10, -3, 15]$ has maximum sum 29, which is represented by the subarray $[7, 10, -3, 15]$.

Define a dynamic programming algorithm which can find the maximum value of $Sum(i, j)$ in $\mathcal{O}(n)$ time (note we don't need to know the i, j values that make it maximum for the final solution). Provide the base cases, recursive subproblems and the recursive formula that defines the solutions of larger subproblems in terms of smaller ones.

[Hint: The first and most difficult step is finding out the recursive relationship of larger problems in terms of smaller ones. If we have the solution on the range $[0, i]$, it matters whether the subarray that yields the maximum sum includes i or does not, when determining whether it is relevant to a solution on $[0, i + 1]$. These distinctions can be solved by adding additional subproblems.]

Solution. The typical structure for solving a dynamic programming problem is to find a set a overlapping subproblems such that the initial problem we attempt to solve such that the original problem is one of the subproblems, and some way to combine solutions of smaller problems to create solutions to larger ones.

For this problem, we can consider the sub-problems **MAX-SUM**(j), which describe the maximum value of sums of contiguous elements of the subarray $[A[1], \ldots, A[j]]$. It is clear that **MAX-SUM**(n) is the

solution of the original problem. However, it is not straightforward to apply a recursive relationship: the solution to **MAX-SUM**(j) is not just **MAX-SUM**$(j-1)+A[j]$ as the subarray yielding **MAX-SUM**$(j-1)$ might not include $A[j-1]$.

To solve this, we can create auxiliary subproblems: we define **MAX-SUM-INCLUDING-J**(j) to be the maximum value of sums $A[i] + \cdots + A[j]$ over all feasible values of i. The subarray satisfying **MAX-SUM-INCLUDING-J**(j) differs in that it must end at $A[j]$, solving the issue we had with **MAX-SUM**(j), and we can define these two sets of subproblems recursively using the other.

The recursive relationships on these subproblems:

$$\textbf{MAX-SUM-INCLUDING-J}(j) = \max(\textbf{MAX-SUM-INCLUDING-J}(j-1) + A[j], A[j])$$

This is clear as **MAX-SUM-INCLUDING-J**(j) must either be $A[j]$ by itself, or consist of some optimal subarray ending at $A[j-1]$, merged with $A[j]$. The maximal sum ending at $A[j-1]$ is in fact **MAX-SUM-INCLUDING-J**$(j-1)$.

$$\textbf{MAX-SUM}(j) = \max(\textbf{MAX-SUM}(j-1), \textbf{MAX-SUM-INCLUDING-J}(j))$$

This comes from noticing that **MAX-SUM**(j) can either include $A[j]$, or not. If it does not include $A[j]$, then the subarray yielding **MAX-SUM**(j) must be fully contained in the first $j-1$ elements, and thus must be **MAX-SUM**$(j-1)$ as well. If it does include $A[j]$, then it is equivalent to **MAX-SUM-INCLUDING-J**(j) by definition.

The base cases are simple: **MAX-SUM-INCLUDING-J**(1) = **MAX-SUM**$(j) = A[1]$.

□