CS3230: Design and Analysis of Algorithms
Semester 2, 2022-23, School of Computing, NUS

Solutions of Practice Problem Set 1

January 30, 2023

Question 1: For each of these, try figuring out why you can't use master theorem to solve these recurrences, you do not need to be formal. I also invite you to get as tight bound as possible, for both upper and lower.

1. $T(n) = 2T(n-1) + \Theta(1)$

   Ans: Using tree recursion, $T(n) = c + 2c + 4c + ... + 2^{n-1}c = (2^n - 1)c = \Theta(2^n)$

2. $T(n) = T(\sqrt{n}) + \Theta(1)$

   Ans: Using tree recursion, $T(n) = height * c = lg(lg(n)) * c = \Theta(lg(lg(n)))$

3. $T(n) = 2T(\sqrt{n}) + \Theta(1)$

   Ans: Using tree recursion, $T(n) = c+2c+4c+...+2^{lg(lg(n))}c = c(2^0+2^1+...+2^{lg(lg(n))}) = \Theta(lg(n))$ (Geometric series)

4. $T(n, m) = T(\frac{n}{2}, m) + \Theta(m)$

   Ans: Using tree recursion, $T(n) = height * m = \Theta(mlg(n))$

5. $T(n) = (\sqrt{n} + 1)T(\sqrt{n}) + \sqrt{n}$

   Ans: Use substitution for this question, the following shows us how to get the upper bound O, note that we can similarly show the lowerbound.

   - Step 1: Guess $T(n) = O(n)$.
   - Step 2: We aim to show that $T(n) \leq cn - c - 1$
     Base case: Let $n_0 = 2$, $T(2) = q$, set c such that $q \leq 2c - c - 1 = c - 1$
     Inductive case: By strong induction, we assume that $T(i) \leq ci - c - 1$ $T(n)$ where $2 \leq i < n$. Let us consider $i = n$ case:

$$\begin{aligned} T(n) &= (\sqrt{n} + 1)T(\sqrt{n}) + \sqrt{n} \\ &\leq (\sqrt{n} + 1)(c\sqrt{n} - c - 1) + \sqrt{n} \\ &\leq cn - c\sqrt{n} - \sqrt{n} + c\sqrt{n} - c - 1 + \sqrt{n} \\ &\leq cn - c - 1 \end{aligned}$$

   We have shown by induction that, T(n) = O(n)

Question 2: Notice that in case 1 of master theorem for example, the condition states: $f(n) \in O(n^{\log_b(a)-\epsilon})$ for some $\epsilon > 0$. The point of this question is to show that this way of framing the condition is crucial.

Prove that $f(n) \in O(n^{\log_b(a)-\epsilon}) \implies f(n) \in o(n^{\log_b(a)})$.

Prove that there exists functions $f(n)$ such that $f(n) \in o(n^{\log_b(a)})$ but $f(n) \notin O(n^{\log_b(a)-\epsilon})$.

Solution: I. Prove that $f(n) \in O(n^{\log_b(a)-\epsilon})$ (1) $\implies$ $f(n) \in o(n^{\log_b(a)})$. Proof sketch:

1. From (1), $\exists n_0, c > 0$, such that $\forall n > n_0$

$$f(n) \leq cn^{\log_b(a)-\epsilon}$$
$$f(n) \leq cn^{\log_b(a)}/n^{\epsilon}$$

2. Note that from here, we can set $d = c/n^{\epsilon}$ and n_0 large enough such that, $\forall d > 0$

$$f(n) < dn^{\log_b(a)}(2)$$

3. from (2) we have $f(n) \in o(n^{\log_b(a)})$ by definition.

II. Prove that there exists functions $f(n)$ such that $f(n) \in o(n^{\log_b(a)})$ but $f(n) \notin O(n^{\log_b(a)-\epsilon})$.

$$f(n) = \frac{n}{\lg n} \in o(n)$$

Then, We can simply show that $\forall \epsilon > 0$

$$f(n) = \frac{n}{\lg n} \notin O(n^{1-\epsilon})$$

Question 3: Consider the function Fun(x, y). What is its output? What is the invariant for the while loop? Can you show that this algorithm correctly compute the output?

`Fun`(x, y):

1. $ans = 0, p = x, q = y$
2. While $q > 0$ do
   (a) $r \leftarrow q \mod 2$
   (b) $q \leftarrow q/2$
   (c) $ans \leftarrow ans + r \times p$
   (d) $p \leftarrow p \times 2$
3. **return** ans

Solution: The output of the function is $x \times y$.

Let $b_k b_{k-1} \dots b_2 b_1 b_0$ be the binary representation of y. Then, the binary representation of $y/2^i$ is $b_k \dots b_i$ and the binary representation of $y \mod 2^i$ is $b_{i-1} \dots b_1 b_0$.

The invariant is: For the i-th iteration, q is $y/2^i$. p is $x \times 2^i$, and ans is $x \times (y \mod 2^i)$.

Initiation: In the 0-th iteration, $i = 0$. We have q is y, p is x and ans is $x \times (y \mod 2^0) = 0$. This is exactly the assignment in step 1.

Maintenance: After the i-th iteration, q is $y/2^i$, p is $x \times 2^i$, and ans is $x \times (y \mod 2^i)$. Then, the $(i+1)$-th iteration goes through steps 3-5. After step 3, r is $q \mod 2 = (y/2^i) \mod 2 = b_i$. q is set to be $q/2 = (y/2^i)/2$, which is $y/2^{i+1}$.

After step 4, $ans = x \times (y \mod 2^i) + b_i \times x \times 2^i = x \times (y \mod 2^i + b_i 2^i) = x \times (y \mod 2^{i+1})$

After step 5, p is set to be $p \times 2 = x \times 2^{i+1}$.

In summary, after the $(i+1)$-th iteration, q is $y/2^{i+1}$, p is $x \times 2^{i+1}$, and ans is $x \times (y \mod 2^{i+1})$.

Termination: The while loop terminates when $q = 0$, which is the $(k+1)$-th iteration. According to the invariant, ans is $x \times (y \mod 2^{k+1}) = x \times y$. The algorithm computes the correct output.

Question 4: Given an array A of integers, a pair (i, j) is said to be an *inversion* if $i < j$ and $A[i] > A[j]$. Design an $O(n \log n)$ time algorithm that counts the number of inversions in a given array of size n.

Solution: We can use a similar approach to Merge Sort.

1. Divide - Each array can be divided into a left sub array and right sub array from the center.
2. Conquer - Now we can calculate the total number of inversions by adding up the inversions in the right sub array, the inversions of the left sub array and any new inversions during the merge stage.
3. Combine - During the merging we iterate through the left sub array and right sub array. Lets use array index i for left sub array and j for right sub array. When ever we encounter arr[i] > arr[j], we need to invert all the elements in the left sub array that comes after element i. Therefore, inverse counter will be incremented by (mid - i)

Algorithm:

```
MergeSort(arr, left, right)
    inversions = 0
    inversions += MergeSort(arr, left, mid)
    inversions += MergeSort(arr, mid+1, right)
    inversions += Merge(arr, left, mid, right)
    return inversions

Merge(arr, left, mid, right)
    i = left
    j = mid + 1
    k = left
    invrsion_counter = 0
    while i<=mid, j<=right
        if arr[i] < arr[j] # no inversions
            temp_arr[k] = arr[i]
            i, k ++
        if arr[i] > arr[j] # inversions
            temp_arr[k] = arr[j]
            j, k ++
            inversion_counter += (mid - i + 1)
    return inversion_counter
```

Question 5: Given an array A of n integers suppose we know that there exists an integer that appears more than $n/2$ times in A. Design a divide-and-conquer algorithm to find that element in $O(n \log n)$ time. You are no allowed to sort the array A.

Solution: We can use divide and conquer to calculate the most occurring element in the array and check if that occurs more than $n/2$ instances.

1. Divide - Each array can be divided into a left sub array and right sub array from the center. When there is only one element in the array, its trivially the majority element of the array.
2. Conquer - Now we can calculate the majority element in the left and right sub array.
3. Combine - During the combine stage, if both left and right sub arrays have the same majority element that will be the overall majority element. If they are not equal, we loop through the combined array and count the occurrences of the two candidates from left and right sub arrays and decide based on the count.

Algorithm:

```
Majority(arr, left, right)
    if left == right, return arr[left]
```

```
    leftMaj = Majority(arr, left, mid)
    rightMaj = Majority(arr, mid + 1, right)

    if leftMaj == rightMaj, return leftMaj
    else
        count_leftMaj = count ocurrances of leftMaj in arr[left - right]
        count_rightMaj = count ocurrances of rightMaj in arr[left - right]
        if count_leftMaj > count_rightMaj, return leftMaj
        else, return rightMaj
```

Question 6: Can you design an $O(n)$ time algorithm for the problem stated in Question 5? (Note, this is not a divide-and-conquer algorithm.)

Solution: **Given $O(n)$ space** We can implement a simple hash map mapping integer encountered to the the number of times we encountered it.

Given $O(1)$ space Boyer-Moore majority voting algorithm (this algorithm only works if we know that there exist a majority element as defined in the question). **Algorithm**

```
MajorityVoting(arr)
    majorityElement = arr[0]
    counter = 1
    for num in arr[1..<n]:
        if counter == 0:
            majorityElement = num
        if num == majorityElement:
            counter++
        else:
            counter--
    return majorityElement
```

Question 7: Given an array A of n integers (possibly 0 or negative as well), find the largest possible value c that can be obtained by summing up the values in some contiguous subarray of A, i.e., $c = A[i] + A[i+1] + A[i+2] + \cdots + A[i+t]$ for some i, t. Think of a divide and conquer solution that does it in $O(n \log n)$ time. As a bonus, you can then think about how to improve it to $O(n)$ by some minor modifications.

Solution: **$O(n \log n)$ solution** We cab divide the array into a left sub array and right sub array from the center. If we keep dividing the array when the length is one the solution is trivial, the maximum sum is equal the the element in the array. Now when we combine, the maximum possible contiguous subarray could be from the right sub array or the left sub array. It can also be a contiguous sub array spanning across the center point. Therefore in the combining stage we need to consider all three possibilities. To find the largest possible contiguous sub array across the center we can scan to the left and right of the center and calculate the maximum sum in linear time.

Algorithm

```
MaxSubArray(arr, left, right)
    left_sum = MaxSubArray(arr, left, mid)
    right_sum = MaxSubArray(arr, mid + 1, right)
    mid_sum = findMidSum(arr, mid)

    return max(left_sum, right_sum, mid_sum)

findMidSum(arr, mid)
    max_r = -inf
```

```
i = mid
sum = 0
while i>=0
    sum += arr[i]
    if sum > max_r, max_r = sum
max_l = -inf
i = mid
sum = 0
while i < arr.len
    sum += arr[i]
    if sum > max_l, max_l = sum
return max_r + max_l
```

Question 8: Consider an array of distinct integers sorted in increasing order. The array has then been rotated (anti-clockwise) $k(< sizeofarray(n))$ number of times, i.e., all the numbers in the sorted array have been (cyclically) shifted k places on the left side. Now given such an array, find the value of k.

Solution: At the start the array is sorted. Therefore the largest element will be at the end of the array. When the array is rotated anti-clockwise once, the largest element will move one step to the left. If the array is rotated again, the largest element will take another step to the left and so on. Therefore, if we find the position of the largest element (i_{max}) we can calculate k as $k = n - i_{max}$. We can use a linear search for an $O(n)$ solution, or use a binary search for an $O(lg(n))$ solution.

Question 9: Consider the problem of finding a *peak* in a 2D-array of size $m \times n$, as described in Tutorial 4. In the tutorial we have seen an algorithm with running time $O(m \log n)$. Can you modify that algorithm to achieve running time $O(m + n)$? (**Hint:** In the tutorial we reduced the problem of size $m \times n$ to that of size $m \times n/2$. Now try to come up with some argument so that you can reduce the problem of size $m \times n$ to that of size $m/2 \times n/2$.)

Solution: Define crosshair of the 2D array to be subset of the array that contains the middle row, middle column, and the border of the array.

```
Find2DPeak(A):
    If A has less than or equal to 3 rows or 3 columns
        return the maximal element
    Otherwise:
        Look at the crosshair of array A
        Find the maximal element of the crosshair
        If the maximal element is the peak, return that element
        Else:
            // Note that the crosshair has split our 2D array into 4
            // quarters and reaching else case means that there is at
            // least 1 element, X, next to our maximal element that is
            // larger than it. Also note that X must be inside one
            // of the 4 quarters
            Recurse on the quarter that contains X
```

Building on our solution in tutorial, finding the maximal element in the crosshair would allow us to cut the search columns and rows by half. The proof of correctness for this is almost identical to the proof in our Week 4 Tutorial. In short, when we recurse on the quarter, the problematic cases will be the border of the quarter. Since X is in the border of the quarter, we know that any element that is chosen to be the maximal element must be larger or equals to X, and thus larger than any element in the box bounding it.