COMP3011 Homework 1

ZHANG Caiqi 18085481d

September 28, 2020

1

1.1 a)

In this question, we can use the master method with $a = 9, b = 3$, and $f(n) = n + 3011$. Therefore, the $n^{\log_b a} = n^{\log_3 9} = n^2$, and $f(n) = \Theta(n)$. By the case one of the mater method, $T(n) = \Theta(n^{\log_b a}) = \Theta(n^2)$.

1.2 b)

Firstly, we try to make a guess of $T(n)$. Let $R(n) = 3R(n/2) + n$, by the master method, $R(n) = \Theta(n^{\log_2 3})$. So the exponent of $T(n)$ should be smaller than $\log_2 3$, we may guess it is 1, which means $T(n) = \Theta(n)$. Then, we use the substitution method.

- Upper bound: $T(n) = O(n)$

  Assume $T(m) \leq cm$ for some constant $c > 0$ holds for all positive $m < n$. In particular, $T(n/2) \leq cn/2$ and $T(\lg n) \leq c \lg n$. Then, $T(n) \leq cn/2 + c \lg n + n = (c/2 + 1)n + c \lg n \leq cn$, by choosing c larger than 2, when n is big enough. For example, $c = 4$ and $n > 10$.

  Therefore, $T(n) = O(n)$.

- Lower bound: $T(n) = \Omega(n)$

  Assume $T(m) \geq cm$ for some constant $c > 0$ holds for all positive $m < n$. In particular, $T(n/2) \geq cn/2$ and $T(\lg n) \geq c \lg n$. Then, $T(n) \geq cn/2 + c \lg n + n = (c/2 + 1)n + c \lg n \geq cn$, by choosing c less than 2, when n is big enough. For example, $c = 2$ and $n > 1$.

  Therefore, $T(n) = \Omega(n)$.

Since $T(n) = O(n)$ and $T(n) = \Omega(n)$, we conclude that $T(n) = \Theta(n)$.

1.3 c)

First do change of variables, let $m = \lg n$. Then, we divide two sides by n to get:

$$\frac{T(2^m)}{2^m} = \frac{T(\sqrt{2^m})}{\sqrt{2^m}} + 1 \tag{1}$$

Set $G(m) = \frac{T(2^m)}{2^m}$, then we get $G(m) = G(m/2) + 1$. Use the master method with $a = 1, b = 2$, $f(n) = 1$, we can find $n^{\log_b a} = 1$. According to case 2, $G(m) = \Theta(\lg m)$.
Therefore, $T(n) = T(2^m) = 2^m G(m) = 2^m \Theta(\lg m) = \Theta(n \lg \lg n)$.

2

The basic idea in this question is that we firstly sort the list A. Then we check the elements in list B one by one to see whether it is in list A by doing binary search. If there is an element in A, which is the same as the one in B, we can find it and return "YES". If there is not, it will return "NO". The algorithm is shown below.

Algorithm 1: Find same elements

Input: Two unsorted lists A and B of numbers.
Output: "YES" if some number belongs to both A and B; "NO" otherwise.

```
1 MergeSort(A);
2 for i = 1 to n do
3 |   isIn = BinarySearch(A, B[i]);
4 |   if isIn = true then
5 |   |   return "YES"
6 |   end
7 end
8 return "NO"
```

Let $T(n)$ denotes the worst-case running time. Then $T(n) = M(m) + n \cdot B(m)$, where $M(m)$ is the time complexity of $MergeSort(A)$ and $B(m)$ is the time complexity of $BinarySearch(A)$. Because that $m = O(\lg n)$, $M(m) = O(m \lg m) = O(\lg n \lg \lg n)$ and $B(m) = O(\lg m) = O(\lg \lg n)$. Therefore, $T(n) = O(\lg n \cdot \lg \lg n) + O(n \cdot \lg \lg n) = O(\lg n \cdot \lg \lg n + n \cdot \lg \lg n)$. It satisfies that $T(n) = o(n \lg n)$.

3

As the list is sorted, we can use the variation of binary search to solve this problem. The algorithm is as followed.

Algorithm 2: Binary index search

Input: A sorted array of distinct integers.
Output: Any index i such that $A[i] = i$, if such an index exists; -1 otherwise.

```
initialize left = 1, right = n, mid
while left ≤ right do
    mid = (left+right)/2
    if A[mid] <mid then
        left=mid+1
    end
    if A[mid] >mid then
        right=mid-1
    end
    if A[mid] = mid then
        return mid
    end
end
return -1
```

Now let's discuss why this algorithm works. From the question, we know that the array is sorted and consists of distinct integers. When we do binary search, we compare $A[mid]$ and mid. If $A[mid] < mid$, which means we need find the element in the right part. The reason why it cannot be shown in the left part is that the decreasing rate of index cannot be greater that the decreasing rate of integers, so in this situation $A[mid] = mid$ cannot happen in the left. After finishing the binary search of the list, if there is such an element, we can finally find it, and if not, the algorithm will return -1.
The time complexity $T(n)$ of this algorithm is mainly on binary search, so worst case $T(n) = O(\lg n)$, which satisfies the requirement.

4

The answer is "YES".

In the beginning, we can prove that the minimum graph coloring of a odd length circle is 3. First, we note that 3 colors is sufficient. Any vertex in the graph only has two neighbors, so 3 colors can definitely color all of the nodes. Also, it cannot be less than 3. For example, a graph with three vertices cannot be colored by 2. So, 3 is the global optimal solution. Then we use the induction to prove the correctness of the greedy algorithm.

- *Claim* : In any iteration, the total number of colors will not exceed 3.
- *Base case:* There is only one node with one color. The claim is true.

- *General case:* Assume that there is already n nodes be colored and the total number of colors is no more that 3. Now we are going to color the $n+1$ node. There are the following cases:

  Case 1: The two neighbors of the node are not colored. Then, we can use any existing color to color the $n+1$ node.

  Case 2: One of the two neighbors of the node is not colored. Then, we can either choose one of the existing color or use a new color to color the $n+1$ node, and the total number of colors will not exceed 3 also.

  Case 3: Both of the two neighbors of the node are not colored. If they are in the same color, we can use another color for the $n+1$ node. If they are in the different color, we can always use the third color for the $n+1$ node.

  All in all, in the general case, the claim is also true.

Therefore, the greedy coloring algorithm of the odd length circle will always compute a minimum graph coloring of C_n.

5

Firstly, we define A is *greater with respect* than B if $x_A \geq x_B$ and $y_A \geq y_B$, for the convenience of description. The basic idea of the greedy algorithm is that we maintain a variable max with initial value of $P[1]$ and then compare it with other elements in P one by one. If the element is *greater with respect* then max, then we replace the max by the new element and updating the P by deleting the original max from P. In the end of each iteration, we add max to P(M). Repeat the above steps.

Algorithm 3: Find maximal with respect

```
Input: P
Output: M(P)
initialize max = P[1]
while P is not empty do
    for item in P do
        Compare item with max
        if item greater with respect than max then
            delete max from P
            max = item
        end
    end
    add max to M(P) and delete it from P
    if P is not empty then
        max = P[1] //initialize max to the next loop
    end
end
return M(P)
```

We need to prove that the algorithm: 1. is a greedy algorithm and gives the correct answer. 2. runs in $O(nm)$ time.

1. Now let's prove this algorithm satisfies the requirements. Firstly, in every loop it makes the local optimal decision so it is a greedy algorithm. Secondly, in every loop, the algorithm can find a *max* and also delete all the elements, which definitely cannot be a *maximal with respect to P*. In the end of the algorithm, all the satisfied elements in P will be selected to M(P) and all the other elements will be deleted from P. So, at last, the P is empty. Therefore, we show that using this greedy algorithm can find the set of all points in P that are maximal with respect to P and reach the global optimal solution. The algorithm is correct.

2. Then let's see the worst time complexity $T(n)$. We know that the length of P is n, and the length of M(P) is m. For steps 3 to 9, each iteration takes $O(n)$ time because every time we need to check the elements in P one by one. About step 2, the *while* loop, the times of execution depends on the elements found in M(P), because every time we can get exactly one result. So, the *while* loop will execute m times. Other steps in the algorithm are not the main time consuming steps. Therefore, the time complexity of the algorithm is $O(nm)$.