Sample Solution for Problem Set 3

Data Structures and Algorithms, Fall 2021

October 3, 2021

Contents

1 Problem 1 **2**

2 Problem 2 **4**

3 Problem 3 **5**

4 Problem 4 **6**

5 Problem 5 **7**

6 Problem 6 **8**

7 Problem 7 **9**

1 Problem 1

(a)

Proof. We prove it by induction.

Let $T(n) \geq cn \lg n$.

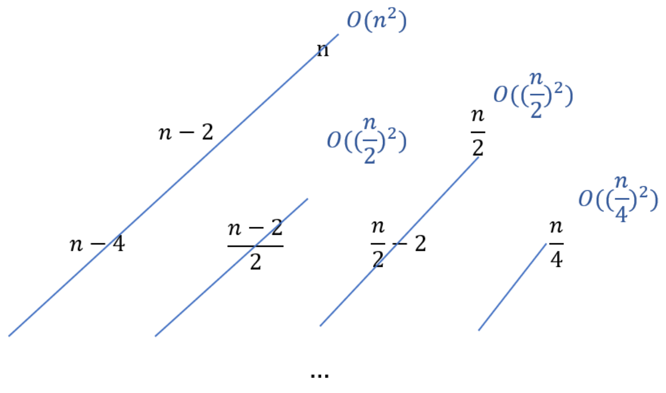
Base case: When $n = 1$, $T(n) \geq c \lg 1 = 0$.

Induction: $T(n) = 2T(n/2) + n \geq cn \lg\left(\frac{n}{2}\right) + n \geq cn \lg n$.

Thus, the induction hypothesis holds. □

(b)

Substitution method is a way to prove the recursion solution strictly. there are several ways to guess a good upper bound. Use recursion tree and sum all the layers is not a good idea for this problem since it will give you an exponential function. You can see that the recursion tree is quite unbalanced, which inspired use to sum the tree diagonally as the following figure shows.



For any i, $\left(\frac{n}{2^i}\right)^2$ will appear at most n^i times. (Each $\left(\frac{n}{2^i}\right)^2$ will generate at most n times of $\left(\frac{n}{2^{i+1}}\right)^2$). Now we get a guessed upper bound

$$\sum_{i=1}^{\log n} \left(\frac{n}{2^i}\right)^2 \cdot n^i = n^{O(\log n)}$$

The recursion tree method is equivalent to the following expansion

$$T(n) = T(n-2) + T(\frac{n}{2}) + n \leq nT(\frac{n}{2}) + n^2 = n^{O(\log n)}$$

We prove it by substitution method.

Substitution method: Suppose $T(k) < k^{c \log k}$ for $k < n$, then

$$T(n) = T(n-2) + T(\frac{n}{2}) + n \leq (n-2)^{c\log(n-2)} + \left(\frac{n}{2}\right)^{c\log n - 2c} + n$$

Consider $f(x) = x^{c\log(n-2)}$ which is a convex function and $f'(n-2) \geq (n-2)^{c\log(n-2)-1}$, which lead to

$$(n-2)^{c\log(n-2)} \leq n^{c\log(n-2)} - (n-2)^{c\log(n-2)-1}$$

For sufficiently large n and c, we have

$$(n-2)^{c\log(n-2)-1} \geq \left(\frac{n}{2}\right)^{c\log n - 2c} + n$$

Thus

$$T(n) \leq n^{c\log(n-2)} - (n-2)^{c\log(n-2)-1} + \left(\frac{n}{2}\right)^{c\log n - 2c} + n \leq n^{c\log n}$$

remark

$n^{O(\log n)}$ is not equivalent to $O(n^{\log n})$.

2 Problem 2

(a)

$$T(n) = \Theta(n^2)$$

(b)

$$T(n) = \Theta(n \lg n)$$

3 Problem 3

Overview

Note that $(2^k a + b)^2 = 2^{2k}a^2 + b^2 - 2^{k+1}((a-b)^2 - a^2 - b^2)$. Therefore, we can divide a n-digit binary numbers into halves and apply divide and conquer technique to obtain the following algorithm in $O(n^{\log 3})$ time.

Algorithm

Algorithm 1 Squaring Algorithm(SA)

```
Require A $n$-digit binary number $x$;
Ensure A $2n$-digit binary number $y = x^2$.
1: if $n = 1$ then
2:     return $x$;
3: $k = \lceil \frac{n}{2} \rceil, a = x/2^k, b = x \mod 2^k$;
4: $c = SA(|a-b|), a = SA(a), b = SA(b)$;
5: return $2^{2k}a + b - 2^{k+1}(c - a - b)$;
```

Complexity Analysis

It can be seen that $T(n) = 3T(\frac{n}{2}) + O(n)$. Hence, our algorithm runs in $O(n^{\log_2 3})$ time.

4 Problem 4

If n is given, we can apply binary search on array A to find a position containing i in $O(\log n)$ time. Therefore, it suffices to determine n in $O(\log n)$ time. Note that, if we can find an upper bound U for n satisfying that $U \leq 2n$ in $O(\log n)$ time, we completed our task as we can again apply binary search technique to search the exact value of n in $O(\log n)$ time. We now introduce the following simple but powerful "binary lifting" technique.

Algorithm

Algorithm 2 Lifting

```
Require  An array $A$. (In fact, we are not given the whole array $A$ as input. Instead, we are given an
    oracle where we can query the value of $A_i$ for $i \in \mathbb{N}$.)
Ensure  An upper bound $U$ for $n$ satisfying $U \leq 2n$.
1: $x = 1$;
2: while $A[x] \neq \infty$ do
3:     $x = 2x$;
4: return $x$;
```

Obviously, this algorithm is correct and runs in $O(\log n)$ time.

5 Problem 5

In this problem, we will provide two algorithms which works in $O(n)$ time. Obviously, it suffices to find one delegate in the majority party.

Randomized Algorithm

The randomized one is quite simple. We repeat the following procedure until the chosen delegate is in the majority party.

- Uniformly sample a delegate, and organize a meeting between this delegate and others.

Note that the probability of choosing a delegate in the majority party is at least $\frac{1}{2}$. Therefore, the expected running time of our algorithm is $O(n)$.

Voting Algorithm

Boyer Moore Voting Algorithm is a well-known algorithm for finding the majority element of a sequence in linear time. We can easily revise the algorithm to solve this given task.

Algorithm 3 Voting

```
Require Integer $n \geq 1$, and an oracle $F(x, y)$ which returns 1 if $x$ and $y$ are in the same party, and 0
    otherwise.
Ensure A index $i \in [n]$, representing that the $i$-th delegate is in the majority party.
 1: $cur = 1, cnt = 1$;
 2: for $i = 2 \to n$ do
 3:     if $cnt == 0$ then
 4:         $cur = i, cnt = 1$;
 5:     else
 6:         if $F(cur, i) == 1$ then
 7:             $cnt = cnt + 1$;
 8:         else
 9:             $cnt = cnt - 1$;
10: return $cur$
```

6 Problem 6

We only need to maintain four values for an interval $A[l, \cdots, r]$

- sum : sum of elements of $A[l, \cdots, r]$
- maxl : maximum subarray of $A[l, \cdots, r]$ which starts from l
- maxr : maximum subarray of $A[l, \cdots, r]$ which ends at r
- ans : maximum subarray of $A[l, \cdots, r]$

Algorithm 4 FIND-MAXIMUM-SUBARRAY

```
function FIND-MAXIMUM-SUBARRAY(A, low, high)
    if high == low then
        return (a[low], a[low] , a[low], a[low])
    else
        mid = ⌊ (low + high) / 2 ⌋;
        (lAns, lMaxl, lMaxr, lSum) = FIND-MAXIMUM-SUBARRAY(A, low, mid)
        (rAns, rMaxl, rMaxr, rSum) = FIND-MAXIMUM-SUBARRAY(A, mid + 1, high)
        ans = max(lMaxr + rMaxl, lAns, rAns)
        maxl = max(lMaxl, lSum + rMaxl)
        maxr = max(rMaxr, rSum + lMaxr)
        sum = lSum + rSum
        return (ans, maxl, maxr, sum)
```

7 Problem 7

To simplify our analysis, we will assume that elements are distinct. Even though, our algorithm still works when there exists duplicated elements.

Algorithm

Suppose we paint the i-th largest nodes for all $1 \le i \le \ell$ in the max-heap, the most important observation is that painted nodes form a connected component containing the root in the max-heap. Suppose we have found ℓ largest elements and painted them in the max-heap, the $\ell + 1$-th one can only be those unpainted nodes that are connected to painted one. Therefore, we can maintain another heap consisting of those candidates nodes, and get the k-th largest element with it.

In the following implementation, we assume that elements x in the max-heap S consists of its key value key, pointer to its left child $left$ and right child $right$. Furthermore, The max-heap $S.root$ points to the top elements of the heap.

Algorithm 5 k-th element

```
Require  A max-heap $S$, and an integer $k \le S.size()$.
Ensure  the $k$-th largest element in $S$.
 1: A max-heap $T$ initialized with no elements;
 2: $cur = S.root, T.push(cur \to key, cur)$;
 3: for $i = 1 \to k - 1$ do
 4:     $ret = T.top(), T.pop()$;
 5:     $left = ret.first \to left, right = ret.first \to right$;
 6:     if $left \neq NULL$ then
 7:         $T.push(left \to key, left)$;
 8:     if $right \neq NULL$ then
 9:         $T.push(right \to key, right)$;
10: return $T.top()$;
```

Time Complexity

Note that, the size of T does not exceed $2k$ during the process. Therefore, each iteration will cost $O(logk)$ time, resulting the total cost of $O(k \log k)$ time.