Sample Solution for Problem Set 2

Data Structures and Algorithms, Fall 2020

November 10, 2020

Contents

1 Problem 1

This upper bound can be verified using the substitution method.
Assuming that $T(n) \leq c(n-2)\lg(n-2) - d(n-2)$ for some $c, d > 0$,

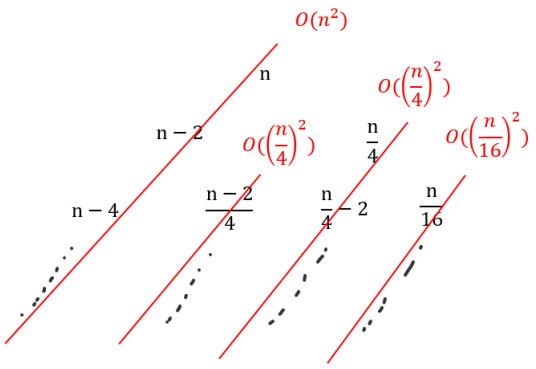
$$\begin{aligned} T(n) &= 2T(\lfloor n/2 \rfloor + 1) + n \\ &\leq 2c\left(\lfloor n/2 \rfloor - 1\right)\lg\left(\lfloor n/2 \rfloor - 1\right) - 2d(\lfloor n/2 \rfloor - 1) + n \\ &\leq c\left(n-2\right)\lg\left(n-2\right) - (d + c - 1)n - 2(d+c) \\ &\leq c(n-2)\lg\left(n-2\right) \end{aligned}$$

when $d > 1$. Therefore, $T(n) \in O(n \lg n)$.

2 Problem 2

(a)

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, $\left(\frac{n}{4^i}\right)^2$ will apear at most n^i times. (Each $\left(\frac{n}{4^i}\right)^2$ will generate at most n times of $\left(\frac{n}{4^{i+1}}\right)^2$). Now we get a guessed upper bound

$$\sum_{i=1}^{\log n} \left(\frac{n}{4^i}\right)^2 \cdot n^i = n^{O(\log n)}$$

The recursion tree method is equivalent to the following expansion

$$T(n) = T(n-2) + T(\frac{n}{4}) + n \leq nT(\frac{n}{4}) + n^2 = n^{O(\log n)}$$

We prove it by substition method.

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

$$T(n) = T(n-2) + T(\frac{n}{4}) + n \leq (n-2)^{c \log(n-2)} + \left(\frac{n}{4}\right)^{c \log n - 2c} + n$$

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

$$(n-2)^{c \log(n-2)-1} \geq \left(\frac{n}{4}\right)^{c \log n - 2c} + n$$

Thus

$$T(n) \leq n^{c \log(n-2)} - (n-2)^{c \log(n-2)-1} + \left(\frac{n}{4}\right)^{c \log n - 2c} + n \leq n^{c \log n}$$

Remark 2.1. *$n^{O(\log n)}$ is not equivalent to $O(n^{\log n})$.*

(b)

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

3 Problem 3

(a) $(1, 5)$, $(2, 5)$, $(3, 4)$, $(3, 5)$, $(4, 5)$.

(b) The running time of insertion sort is a constant times the number of inversions. Let $\text{Inv}(i)$ denote the number of $j < i$ such that $A[j] > A[i]$. Then $\sum_{i=1}^{n} \text{Inv}(i)$ equals the number of inversions in A. Consider the while loop of the insertion sort algorithm. This loop will execute once for each element of A which has index less than j is larger than $A[j]$. Thus, it will execute $\text{Inv}(j)$ times. We reach this while loop once for each iteration of the for loop, so the number of constant time steps of insertion sort is $\sum_{j=1}^{n} \text{Inv}(j)$ which is exactly the inversion number of A.

(c) We modify the MERGE procedure of the merge-sort algorithm on page 31 of CLRS 3rd Edition.
Insert **inv** $= 0$ before the for loop at line 12.
Insert **inv** $=$ **inv** $+ n_1 - i + 1$ before line 17.
Insert **return inv** at the end of the MERGE procedure.
Then we modify the main procedure of the merge-sort algorithm on page 34. Replace line 3,4,5 with
$\text{inv}_l =$ MERGE-SORT(A, p, q)
$\text{inv}_r =$ MERGE-SORT$(A, q + 1, r)$
return MERGE$(A, p, q, r) + \text{inv}_l + \text{inv}_r$.

4 Problem 4

1. $T(n) = 2T(n/2) + cn = \Theta(n \lg n)$.

2. $\Theta(n^2)$.

$$\begin{aligned}
T(n) &= 2T(n/2) + cn + 2N = 4N + cn + 2c(n/2) + 4T(n/4) \\
&= 8N + 2cn + 4c(n/4) + 8T(n/8) \\
&= \sum_{i=0}^{\lg n - 1} (cn + 2^i N) \\
&= \sum_{i=0}^{\lg n - 1} cn + N \sum_{i=0}^{\lg n - 1} 2^i \\
&= cn \lg n + N \frac{2^{\lg n} - 1}{2 - 1} \\
&= cn \lg n + nN - N = \Theta(nN) \\
&= \Theta(n^2).
\end{aligned}$$

3. $\Theta(n \lg n)$.

$$\begin{aligned}
T(n) &= 2T(n/2) + cn + 2n/2 \\
&= 2T(n/2) + (c+1)n \\
&= \Theta(n \lg n).
\end{aligned}$$

5 Problem 5

Suppose a and b are two n-digit number. Now we have an algorithm to square a n-digit number in $T_1(n)$ time.

We can calculate $a \cdot b$ by

$$a \cdot b = \frac{a^2 - b^2 - (a-b)^2}{2}$$

$a - b$ is an $n - digit$ number. Plus and minusbetween two n-digit number cost $O(n)$ time, dividing the number by 2 cost $O(n)$ time, thus we can calculate $a \cdot b$ in

$$3 \cdot T_1(n) + O(n)$$

We claim that $T_1(n) = \Omega(n)$, since any algorithm calculating the square of a number need to use $O(n)$ to read the number. Thus,

$$3 \cdot T_1(n) + O(n) = O\left(T_1(n)\right)$$

Since $T_2(n)$ is the running time of the fastest algorithm to multiply two $n - digit$ number, we get

$$T_2(n) = O\left(T_1(n)\right)$$

Which contradict the claim $T_1(n) = o\left(T_2(n)\right)$.

Remark 5.1. *If you use $a \cdot b = (a+b)^2 - a^2 - b^2$ to calculate multiplication, the running time is acctually $T_1(n+1) + 2T_1(n) + O(n)$, since $a + b$ might be a $n + 1$ digit number. However, it is hard to show that $T_1(n+1) = O\left(T_1(n)\right)$. Acctually, the conclusion is not right, we have a counter example*

$$T_1(n) = \begin{cases} n^2 & n \text{ is even} \\ n & n \text{ is odd} \end{cases}$$

One can varify that $T_1(n) = O\left(T_1(n)\right)$ is not right. However, you might come up with some techniques to make it right given some prerequisite. That is not recommended since use $(a-b)^2$ is already an perfect way to prove it.

6 Problem 6

Key idea for this task is to maintain $\max_{i=l}^{r} \sum_{j=l}^{i} a_j$ and $\max_{i=l}^{r} \sum_{j=i}^{r} a_j$ for each segment. To simplify our solution, the algorithm we provide will NOT output index (u, v) such that $\sum_{i=u}^{v} a_i$ reaches the maximum. However, it can be done by adding some extra variables to track the corresponding segments recursively.

6.1 Algorithm

Algorithm 1 FIND-MAXIMUM-SUBARRAY(FMS)

Require: array a, left boundary l, right boundary r.
Ensure: following values in order:

- $\sum_{i=l}^{r} a_i$
- $\max_{i=l}^{r} \sum_{j=l}^{i} a_j$;
- $\max_{i=l}^{r} \sum_{j=i}^{r} a_j$;
- $\max(l, r) = \max_{l \le i <= j \le r} \sum_{k=i}^{j} a_k$;

1: **if** $l == r$ **then**
2: return $(a[l], a[l], a[l], a[l])$
3: $mid \leftarrow (l + r) >> 1$;
4: $(leftsum, leftlmx, leftrmx, leftmx) \leftarrow FMS(a, l, mid)$
5: $(rightsum, rightlmx, rightrmx, rightmx) \leftarrow FMS(a, mid + 1, r)$
6: $crossmx \leftarrow leftrmx + rightlmx$
7: $lmx \leftarrow \max(leftlmx, leftsum + rightlmx)$
8: $rmx \leftarrow \max(rightrmx, rightsum + leftrmx)$
9: return $(leftsum + rightsum, lmx, rmx, \max(leftmx, rightmx, crossmx))$

6.2 Correctness

We will prove the output values are correct during the process. Note that $crossmx$ is exactly $\max_{\substack{l \le u \le mid \\ mid < v \le r}} \sum_{i=u}^{v} a_i$ as $leftrmx$ is maximum value over all segments ending at mid and $rightrmx$ starting from $mid + 1$. And correctness of other values can be verified similarly.

6.3 Time Complexity

Note that We only use $\Theta(1)$ time during the combine step, we may write down following recurrence relation $T(n) = 2T(n/2) + \Theta(1)$, which is said to be $T(n) = \Theta(n)$ according to master theorem.

6.4 Remark

If we divide segment into two subsegments with length $n - 1$ and 1 rather than $n/2$ and $n/2$ in the divide step, above algorithm is exactly the same with Excercise 4.1-5 in CLRS. (but that can NOT be your reason to argue if your algorithm is almost the same with CLRS').

7 Problem 7

Some definitions: Suppose n friends are numbered from 1 to n, and we have an operation $query(a, b)$ for $1 \leq a \neq b \leq n$, which returns a pair (I_1, I_2) where $I_1, I_2 \in \{C, W\}$ (C means $citizen$ and W means $werewolf$), I_1 is the identity of a told by b and I_2 is the identity of b told by a.

Define function $Judge(a, A)$ where A is an array of citizens as follows:

- $cnt \leftarrow 0$
- For $i = 1$ to $A.length$ where $A[i] \neq a$ do
  - $(I_1, I_2) \leftarrow query(a, A[i])$, if $I_1 = C$ then $cnt \leftarrow cnt + 1$.
- If $cnt \geq \frac{A.length-1}{2}$ then return C. Otherwise return W.

Lemma 1. *If the number of citizens in $A[1]$ to $A[A.length]$ is larger than the number of werewolves, then $Judge(a, A)$ will correctly return the identity of a.*

Proof. If a is a citizen, then $A[1]$ to $A[A.length]$ minus a at least half of citizens, since the number of citizens is larger than the number of werewolf. These citizens will say a is citizen, which will make our algorithm output the right answer.

If a is a werewolf, then the number of citizens in $A[1]$ to $A[A.length]$ minus a is larger than the number of werewolf. Thus, less than $\frac{A.length}{2}$ friends will say a is a citizen, which will make our algorithm output the right answer. □

(a) Suppose a is the given input. Call $Judge(a, A)$. According to Lemma 1, the correctness is proved. The complexity is obviously $O(n)$.

(b) Define function $FindCitizen(l, r)$ as following:

1 If $l == r$, return $A[l]$.

2 Set $m = \lfloor \frac{l+r}{2} \rfloor$. Let $a = FindCitizen(l, m)$ and $b = FindCitizen(m + 1, r)$

3 If $Judge(a, A[l...r]) = C$, retunr a. If $Judge(b, A[l..r]) = C$, return b.

Lemma 2. *If $l \leq r$ and there are more $citizen$ than $werewolf$ in $A[l...r]$, then $FindCitizen(l, r)$ will return a real citizen.*

Proof. We prove it by induction on the length $r - l + 1$.

Base case: When $l == r$, $A[l]$ must be a citizen given that there are more $citizen$ than $werewolf$.

Induction: Suppose $l < r$ and there are more $citizen$ than $werewolf$ in $A[l...r]$. One of $A[l...m]$ and $A[m+1, r]$ must have more $citizen$ than $werewolf$ (Otherwise the sum of them will contradict the prerequisite). Since $m = \lfloor \frac{l+r}{2} \rfloor$, obviously $l \leq m < r$. According to induction hypothesis, one of a and b is $citizen$. According to Lemma 1, we can return a real citizen in step 3.

Thus, the induction hypothesis holds. □

Now by calling $FindCitizen(1, n)$, we can get a citizen.

Complexity: Suppose $FindCitizen(l, r)$ use time $T(n)$ where $n = r - l + 1$, then

$$T(n) \leq 2T(\lceil \frac{n}{2} \rceil) + O(n)$$

According to mater method, $T(n) = O(n \log n)$.

(c) I will give two method based on different ideas.

(1) The algorithm use recursion. Define function $FastFind(A)$ where A is a friends array as follows.

1 Create an empty array B index begin with 1.

2 **For**($i = 1; i \leq A.length; i+ = 2$) **do**

3 If $query(A[i], A[i+1]) = (C, C)$, add $A[i]$ to B.

4 If $i + 1 \neq A.length$, $I = Judge(A[A.length], A)$. If $I = C$, return $A[A.length]$.

5 Return $FastFind(B)$.

Lemma 3. *If there are more citizens than werewolves in A, then $FastFind(A)$ will return a real citizen.*

Proof. We prove it by induction on the length of A.

Write $n = A.length$. When $n = 1$, there are only on citizen, which will be return.

When $n > 1$, consider the case when n is odd, we varify the last friends by $Judge$. According to Lemma 1, this friends will return iff. it is citizen. Otherwise it is not considered, and the case come to when n is even. Consider the third line. It is easy to see that $query(A[i], A[i+1]) = (C, C)$ iff. $A[i]$ and $A[i+1]$ are both citizens or werewolves. Thus, if citizens are not more than werewolves in B, that must hold in A (since the number is two times the number in B), which is impossible. So B also has more citizens than werewolves. According to induction hypothesis, B will return a real citizen. □

complexity The size of B is at most a half of the size of A, thus, the complexity is

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

which is $T(n) = O(n)$ according to master mathod.

Remark 7.1. *In the third line of the algorithm, we throw away any pair with $(C, W), (W, C), (W, W)$. However, one can pick the one that are identified as $citizen$ to put into B, which can also be proved to be correct.*

(b) The second solution is inspired by one of the student's solution (though he did it wrong). Consider the following question:

- Give you n number $A[1...n]$, the only operation you can use is $equal(i, j)$ which will tell you whether $A[i]$ and $A[j]$ are equal. Now suppose there is a number in $A[1...n]$ that appear more than $\frac{n}{2}$ times. Can you find the number in $O(n)$ time?

This is a well-known question called finding the dominant number. In this question $citizen$ is the dominant number, and you have an operation to judge whether two element is the same (just use $query$ and if (C, C) is returned, they are the same.) The only difference is that if the two numbers are both werewolves, they might not be considered the same since they might lie. However, it does not influence our solution. Now the following is the algorithm to solve finding the dominant number using stack.

1 Create a stack S. Initially, $S.push(A[1])$.

2 For $i = 2$ to n, do

– If $equal\ (A[i], S.top())$ is true, $S.push(A[i])$. Otherwise $S.pop()$.

3 Return $S.top()$.

The following loop invariance is hold

- At any time, all elements in S is equal.

We omit the proof here since it is quite simple.

Now define an element as $killed$ iff. it is popped from S or not equal to the top of S in step 2. In each case the element is $killed$ with another elements that is not equal to it. All elements that is not killed will be left in S at last. Consequently, there must be an element left in S, since there are more than half elements that is the same.

This is not a "carafully" proof! But it shows you why the algorithm runs correctly, both for the finding dominant element problem and the citizens and werewolves problem.