Solution for Problem Set 2

Mianzhi Pan, 181240045

September 23, 2020

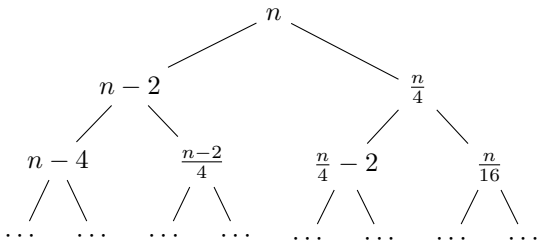
1 Problem 1

To prove $T(n) \in O(nlgn)$, let's guess that $T(n) \leq c(n-a)lg(n-a) \in O(nlgn)$, then we have to prove the existence of c,a and n_0. The proof is as the following:

$$\begin{aligned}
T(n) &= 2T(\lfloor n/2 \rfloor + 1) + n \\
&\leq 2c(\lfloor n/2 \rfloor + 1 - a)lg(\lfloor n/2 \rfloor + 1 - a) + n \\
&\leq 2c(n/2 + 1 - a)lg(n/2 + 1 - a) + n \\
&= c(n + 2 - 2a)lg(n + 2 - 2a) - c(n + 2 - 2a) + n \\
&\leq c(n + 2 - 2a)lg(n + 2 - 2a) \quad (when\ c > 1, n_0 = \lceil \frac{c(2-a)}{1-c} \rceil) \\
&\leq c(n-a)lg(n-a) \quad (when\ a \geq 2)
\end{aligned}$$

2 Problem 2

(a) The recursion tree is like the following:



The length of branch from left to right decreases exponentially and so this tree is not a complete binary tree. Only considering the most left branch shown above, we guess that $T(n) \in O(n^2)$, then we want to obtain $T(n) < cn^2$. However,

$$\begin{aligned}
T(n) &= T(n-2) + T(n/4) + n \\
&\leq c(n-2)^2 + c(n/4)^2 + n \\
&= \frac{17c}{16}n^2 - (4c-1)n + 4c
\end{aligned}$$

The coefficient of n^2 become bigger and further more we can find $T(n) \in \Omega(n^2)$.

Again, we guess $T(n) \in O(2^n)$ i.e.$T(n) \leq c \cdot 2^n$, we have

$$\begin{aligned} T(n) &= T(n-2) + T(n/4) + n \\ &\leq c \cdot 2^{n-2} + c \cdot 2^{n/4} + n \\ &= c \cdot 2^{n-2} + c\sqrt{2}^n + n \\ &\leq c \cdot 2^n \end{aligned}$$

Hence, $T(n) \in O(2^n)$.

(b) $T(n) \in O(nlog_{\frac{1}{\alpha}} n)$

3 Problem 3

(a) Suppose array indexes begin from 0, then the inversions are $(0, 4)$ $(1, 4)$ $(2, 3)$ $(2, 4)$ $(3, 4)$.

(b) The running time of insertion sort is proportional to number of inversions in the input array.

In each iteration of *while* from line 5 to line 7, suppose there are n_j elements larger than $A[j]$ in the sorted array $A[1, 2, \cdots, j-1]$, the algorithm will swap n_j times(each swap takes constant time). Therefore, the total number of swaps is $\sum_{j=2}^{A.length} n_j$, which happens to be the number of inversions.

(c) We only need to do a few modification to $MERGE - SORT$. First we divide the array in half recursively. When doing $MERGE$, we will calculate the number of inversions of the merged array according to its two subarrays.

Algorithm 1 MERGE(A, p, q, r)

```
1: n_1 = q − p + 1
2: n_2 = r − q
3: for i = 1 → n_1 do
4:     L[i] = A[p + i − 1]
5: end for
6: for j = 1 → n_2 do
7:     R[j] = A[q + i]
8: end for
9: L[n_1 + 1] = ∞
10: R[n_2 + 1] = ∞
11: i = 1
12: j = 1
13: num = 0
14: for k = p → r do
15:     if L[i] ≤ R[j] then
16:         A[k] = L[i]
17:         i = i + 1
18:     else
19:         num = num + n_1 − i + 1//because L is sorted
20:         A[k] = R[j]
21:         j = j + 1
22:     end if
23: end for
24: return num
```

Algorithm 2 MERGE-NUM(A, p, r)

```
1: if p == r then
2:     return 0
3: else
4:     mid = ⌊(p + r)/2⌋
5:     left = MERGE − NUM(A, p, mid)
6:     right = MERGE − NUM(A, mid + 1, r)
7:     return left + right + MERGE(A, p, mid, r)
8: end if
```

4 Problem 4

The time spent for merging is always $\Theta(N)$, while that for passing array is different for different passing methods, which will happen when we divide an array into two subarrays L and R.

For method 1: The time spent for passing array is $\Theta(1)$ every time. So $T(N) = 2T(N/2) + \Theta(1) + \Theta(N)$. So $T(N) = O(NlgN)$.

For method 2: Every time the origin array A is copyed twice, so $T(N) = 2T(N/2) + 2\Theta(N) + \Theta(N)$. Therefore, $T(N) = NlgN + N\sum_{i=0}^{lgN} 2^i = O(N^2)$.

For method 3: Every time the total copied length is N, so $T(N) = 2T(N/2) + \Theta(N) + \Theta(N)$. Then $T(N) = O(NlgN)$.

5 Problem 5

It is false. Suppose we have two n-bit integers a and b, we have $a \cdot b = [(a + b)^2 - (a-b)^2]/4$. The complexity of mutiplication is almost twice that of square. Notice 'twice' is a constant, which means $T_1(n) = \Theta(T_2(n))$.

6 Problem 6

In the algorithm introduced in CLRS, $FIND-MAXIMUM-SUBARRAY$ return a tuple $(low, high, max\_sum)$. In order to speed up $FIND-MAXIMUM-CROSSING-SUBARRAY$, we will return $(low, high, max\_sum, max\_prefix, max\_suffix)$, where max_prefix is the sum of the maximum prefix of an array and max_suffix is the sum of the maximum suffix of an array. Therefore, we can implement $FIND-MAXIMUM-CROSSING-SUBARRAY$ by adding together max_suffix of the left subarray and max_prefix of the right subarray, whose time complexity is $O(1)$.

To be more explicit, suppose the left subarray and the right subarray of A are L and R. Then max_sum of A is $MAX(L.max\_sum, R.max\_sum, L.max\_suffix + R.max\_prefix)$. Notice the prefix or suffix of A may pass its midpoint,
$A.max\_prefix = MAX(L.max\_prefix, L.sum() + R.max\_prefix)$ and
$A.max\_suffix = MAX(R.max\_suffix, L.max\_suffix + R.sum())$. Finally, $A.low$ and $A.high$ are the first index and the last index of the maximum subarray of A.

7 Problem 7

(a): "Query" the rest $n-1$ people, if the people whose answer is citizen is no fewer than those whose answer is werewolf, then we can determine that the input player is a citizen. Otherwise the input player is a werewolf.

Proof: Supoose the input player is a citizen, the citizens in the rest people must be no fewer than the werewolves since there are always more citizens than there are werewolves. Therefore, people whose answer is citizen will be no fewer than those whose answer is werewolf.

If the input player is a werewolf, the citizens in the rest people must be more than the werewolves, then number of answer "citizen" will always be fewer than that of answer "werewolves".

(b): To implement $citizen(A)$, where A is the set containing all people, we split the people into two groups L and R, call $citizen$ recursively, set $cl =$

$citizen(L)$ and $cr = citizen(R)$. For each of these two candidates, input it to A and use the method mentioned in question **(a)**. Then we will find at least one citizen from cl and cr. Notice that if $size(A) == 1$, return $A[0]$ directly.

Proof: **Induction basis:** When size of the set becomes 1, the origin set A has been divided into n(suppose there are totally n people) groups. For each of them, return the unique element, and there must be a citizen during the process.

Induction hypothesis: Assume $citizen$ is correct at k^{th} step.

Inductive step: Suppose there are n_k subarrays at step k, then at $(k+1)^{th}$ step there are $n_k/2$ subarrays. Since there are more citizens in the origin array, among the $n_k/2$ subarrays, there is at least one array A_{k+1} who has more citizens than werewolves. So at least one of its two subarrays L_{k+1} and R_{k+1} has more citizens than werewolves. The algorithm will return one correctly citizen from the subarrays. And the citizen can be find in A_{k+1} as well at step $k+1$.

(c): Suppose there are totally n people. We do "query" operations over and over. If both of the two picked people answer "citizen", then they are citizens. Otherwise, we "throw away" both of them and repeat the operations. Notice when n is odd, if we do $\lfloor n/2 \rfloor$ times "query" but fail to find citizen, then the left person must be a citizen.

Proof: It is not difficult to find that if either of the two picked people answer "werewolf", there must be at least one werewolf among them. They are citizens if and only if both of them answer "citizen". So when n is even, at least one pair of people we picked are both citizens since there are always citizens than werewolves. When n is odd, there will exist either at least one pair of citizens of a left citizen at last if $num(citizen) - num(werewolf) == 1$, Otherwise, the case becomes the same as above where n is even.