DSA hw1

B08502041 李芸芳

March 25, 2022

Problem 0

- Problem 1: all by myself
- Problem 2: all by myself
- Problem 3: Fewjetive(on discord, don't know student id and name)
- Problem 4: all by myself
- Problem 5: 蘇簡宏(40840903s)、xor linked list tutorial

Problem 1

1. i keeps doubling until $sum \geq n$, so the time complexity is $\Theta(lg(n))$

2. Suppose the time complexity is $T(n)$, and from the pseudo code, $T(n)$ equals $2T(n-1) + \Theta(1)$. Since the function will be recursive called until $n \leq 0$, then

$$\begin{aligned} T(n) &= 2T(n-1) + \Theta(1) \\ &= 2 \times (2T(n-2) + \Theta(1)) + \Theta(1) \\ &= ... \\ &= 2^{n-1}T(1) + \Theta(1) \end{aligned}$$

   $\therefore$ the time complexity is $\Theta(2^n)$

3. The outer loop iterates lgn times, while the inner loop iterates n times. So the time complexity is $\Theta(nlg(n))$

4. From definition,

$$\begin{aligned} f(n) &\leq c \cdot g(n) \text{ for all } n \leq n_0 \\ [f(n)]^2 &\leq c \cdot f(n) \cdot g(n) \text{ for all } n \leq n_0 \\ f(n) \cdot g(n) &\geq \frac{1}{c}[f(n)]^2 \text{ for all } n \leq n_0 \end{aligned}$$

   $\therefore f(n) \cdot g(n) = \Omega((f(n))^2)$

5. choose $c = n_0 = 1$

$$\begin{aligned} lg(n) &\leq n - 1 \\ &< n \\ &< n^k \end{aligned}$$

$\therefore\ lg(n) \leq c \cdot n^k$ for all $n \geq n_0$
$\therefore\ lg(n) = O(n^k)$

6.

From problem description,

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = c, \text{ where } c \neq \infty$$

$$\begin{aligned} \lim_{n\to\infty} \frac{lg(f(n))}{lg(g(n))} &= \lim_{n\to\infty} \frac{\frac{1}{f(n)}}{\frac{1}{g(n)}}, \text{ from L'hospital rule} \\ &= \lim_{n\to\infty} \frac{g(n)}{f(n)} \\ &= \frac{1}{c} \neq \infty \end{aligned}$$

$\therefore\ lg(f(n)) = O(lg(g(n)))$

Problem 2

1. 12+345*+*93/75*++

   First, enclose every operators and the two operands on both sides with brackets, for example the infix expression in this problem will become $(((1 + 2) \times (3 + (4 \times 5))) + ((9/3) + (7 \times 5)))$

   Second, every time choose the outermost bracket, and replace the right bracket with the operator in the middle until there's no bracket.

2. (1+2)*(5-3)*6/5, outcome = $\frac{36}{5}$

   Scan from left to right, and every time getting two operands and one operator, enclose them with a pair of bracket and transfer it to infix expression.

3.

	$a[0]$	$a[1]$	$a[2]$	$a[3]$
enqueue 1 (example)	1	NIL	NIL	NIL
enqueue 5	1	5	NIL	NIL
enqueue 3	1	5	3	NIL
dequeue	NIL	5	3	NIL
enqueue 4	NIL	5	3	4
enqueue 6	6	5	3	4
dequeue	6	NIL	3	4

4. 10 rounds
   The three 1p should be at the left most place, so every tiles on the left hand side of first 1p (counted from left) should be discard. Therefore the smallest rounded needed is the number of tiles on the left hand side of first 1p counted from left.

5. Randomly pick a person as start, push the number of this person into stack, and record that this number as being shown up and traverse the number clockwise.
   If the number is recorded not having shown up, push the number into stack, else check whether the number on top of stack is the same as the number, if it is, than pop top of the stack, if it's not, then the red cords must intersect.
   correctness: If the red cords intersect, choose any person as start will result in the same result, since there must be some number appear between the same number only once.

6. The number of every person around the table will be at most push into the stack once, and at most pop out of it once. Since the operation push and pop are $O(1)$, and the number of people is n, so the time complexity of checking is $O(n)$.

Problem 3

1. **algorithm**: First assume the structure of node in the list be

```
typedef struct _node {
    int val;
    struct _node* next;
} Node;
```

then my algorithm is as follow

```
Node* FindMiddle(Node* head)
{
    Node *slow = head, *fast = head;

    while (fast->next && fast->next->next)
    {
        slow = slow->next;
        fast = fast->next->next;
    }

    return slow;
}
```

fast is two times more quickly than *slow*.

When the length of the linked list is odd, the WHILE loop will end because $fast->next$ is NULL pointer and *slow* will stop at the middle node; when the length of the linked list is even, the WHILE loop will end because $fast->next->next$ is NULL pointer and the number of nodes that *fast* pass through are twice as the number of node that *show* pass through, so the *slow* will stop at the $\frac{n}{2}st$ node.

time complexity: Assume the length of the linked list is n, then the while loop will iterate $\lfloor \frac{n}{2} \rfloor$ times, so the time complexity is $O(n)$

extra-space complexity: Need two node pointers *slow* and *fast*, so the extra space complexity is $O(1)$.

2. **algorithm**:

```
unsigned int FindFirstMissingNumber(
unsigned int arr[], int n)
{
    FOR i = 0 to n - 1:
        IF arr[i] <= n THEN
            IF arr[abs(arr[i]) - 1] > 0 THEN
                arr[abs(arr[i]) - 1] *= (-1)
        END IF

    missing = n + 1

    FOR i = 0 to n - 1:
        IF arr[i] > 0:
            missing = i + 1

    return missing
}
```

Since all the integer in the array is positive, the first missing number must be in range $[1, n]$ (assuming the length of the array is n). So try marking every number num in range $[1, n]$ as exist by marking $arr[num]$ to be negative. Since the problem ask for the first missing, scan the array from left to right, and once the number in array is positive, it means the "index plus one" number doesn't exist in the original array, then it's the first missing number.

time complexity: In the first for loop, iterating through the array cost $O(n)$ time, and in if-else the swap operation cost $O(1)$ time, so the first for loop cost $O(n)$ time.
In the second for loop, iterating through the array cost $O(n)$ time, and in if condition the assigning operation cost $O(1)$ time, so the second for loop also cost $O(n)$ time.
Therefore the total time complexity is $O(n)$.

extra-space complexity: Need one variable *missing* to record which is the first missing number. Cost $O(1)$ space.

3. **algorithm**:

```
int FindPivot(int arr[], int n)
{
    sum = 0, weight_sum = 0
    FOR i = n - 1 to 0
        sum += arr[i]
        weight_sum += (arr[i] * i)

    num = 0, prev_sum = 0
    left = 0, right = weight_sum
    FOR i = 0 to n - 1
        num = arr[i];
        sum -= num;

        IF left == right THEN
            return i
        END IF

        right -= sum;
        prev_sum += num;
        left += prev_sum;

    return -1
}
```

Calculate the overall torque on the right side, and in the estimate loop (the second loop), every iteration after determination, update the torque on the left side with left-side sum, which in next iteration will be the torque on the left side of pivot, and the torque on the right side should minus ride-side sum, which in next iteration will be the torque on the right side of pivot.

time complexity: The first FOR loop will iterates n times, and the second FOR loop also iterates n times, so the overall time complexity is $O(n)$.

extra-space complexity: Use extra 6 integer variables to record needed number, so the extra space complexity is $O(1)$.