张语晴 zhangyq2022@shanghaitech.edu.cn 2022533170

Ddl 3.28

Problem 1

1.

Algorithm 1

res ← *res* + *ij* + *jk* takes $O(1)$ time.

The inner loop goes $\frac{n-j}{2}$ times.

The inner loop and the middle one altogether:

$$\begin{aligned}
&\frac{1}{2}\left[n\log_2 n-(n+\frac{n}{2}+\frac{n}{4}+\dots)\right]\\
=&\frac{n}{2}\cdot\log_2 n-n\\
=&\Theta(n\log n)
\end{aligned}$$

The outer loop goes $\log_2 n$ times.

Therefore, the overall time complexity is: $n\log^2 n$

Algorithm 2

res ← *res* + *ij* + *jk* takes $O(1)$ time.

The inner loop goes j times.

The inner loop and the middle one altogether:

$$1+2+4+8+\dots+n=2n-1$$

The outer loop goes n times.

Therefore, the overall time complexity is: $n(2n-1)=\Theta(n^2)$

2.

(a) $T(n)=T(n-k)+2k$

$$\begin{aligned}
T(n)&=T(n-k)+2k\\
&=T(n-k-k)+2k+2k=T(n-2k)+4k\\
&=\dots\\
&=\frac{n}{k}\cdot 2k\\
&=O(n)
\end{aligned}$$

(b) $T(n) = 2^k T(n/2^k) + kn$

Tree method. Suppose $T(1) = 1$

Suppose the depth of the tree is x.

On each layer (with the same depth), the time completixty is:

$$\frac{n}{(2^k)^x} \cdot (2^k)^x \cdot k = nk$$

Suppose the maximum depth is y.

$$n = (2^k)^y$$
$$y = \frac{\log_2 n}{k}$$

Therefore, $T(n) = nk \cdot \frac{\log_2 n}{k} = O(n \log_2 n)$

(c) $T(n) = 2^k T(n/2^k) + 2^k - 1$

Guess $T(n) = O(n)$ and prove by induction.

$T(1) = 1$.

$T(2^k) = 2^k T(1) + 2^k - 1 = O(2^k)$.

Suppose $T(2^{km}) = O(2^{km})$ for $m \geq 1$.

Then $T(2^{k(m+1)}) = 2^k T(2^{km}) + 2^k - 1 = 2^k O(2^{km}) + 2^k - 1 = O(2^{k(m+1)})$.

Therefore, $T(2^{km}) = O(2^{km})$ for all $m \geq 1$. $T(n) = O(n)$

Problem 2

7, 5, 1, 4, 6, 9, 3, 2, 8, 10

Problem 3

Name the two arrays A and B.

Suppose $A[i]$ means a query for i to array A.

Let $x \leftarrow n, y \leftarrow n$.

Compare $A[x/2]$ with $B[y/2]$

- if $A[x/2] < B[y/2]$, then $x \leftarrow 3x/2$, $y \leftarrow y/2$. Go back to compare.
- else if $A[x/2] > B[y/2]$, then $x \leftarrow x/2$, $y \leftarrow 3y/2$. Go back to compare.
- else if $A[x/2] = B[y/2]$, then return $A[x/2]$.

Each iteration reduce the size of the array from which we query by half. Since the initial size of array is $\Theta(n)$, we need $O(\log n)$ queries.

Problem 4

(a)

1. Divide the array into $\frac{n}{2}$ subarrays where each subarray consists of two adjacent elements from the original array. – $\Theta(n)$
2. Sort each subarray by filping the 2 elements. – $\Theta(n)$
3. Merge each 2 sorted subarrays by filping:
    1. Filp the second array by the first and the last element.
    2. Connect the first array with the second array.
    3. Filp the connected array by the first “2” and the last “1”

    > For example, to merge two subarrays A: [1, 1, 2, 2, 2, 2] and B: [1, 1, 1, 1, 2, 2]
    > 1. Flip B into [2, 2, 1, 1, 1, 1]
    > 2. Connect: [1, 1, 2, 2, 2, 2, 2, 2, 1, 1, 1, 1]
    > 3. Flip: [1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2]

    Time complexity for this step is determined by the length of each 2 subarray, so the overall time complexity is $\Theta(n)$
4. Go back to step 2, merge until there is only one array.

Each iteration reduce the number of the array by half. Since we have $\frac{n}{2}$ arrays in step 1, there are totally $\log_2 n$ iterations. Each iteration has time complexity $\Theta(n)$.

In recurrence relation:

$$T(n) = 2\,T(n/2) + \Theta(n)$$
$$T(n) = O(n \log n)$$

(b)

1. Randomly pick a pivot number P.
2. Put P to the end of the array by flipping P with the last element in the array. This taks $O(n)$ time.
3. See all the numbers smaller than P as 1, and all numbers larger than P as 2. Perform the sorting algorithm to these ‘1’s and ‘2’s, which takes $O(n \log n)$ time.
4. Put P in between the sorted two parts by filping P with the first element in the second part.
5. Go back to step 1. Perform steps 1, 2, 3, 4 until the array is sorted.

Each loop beginning with picking a pivot takes $O(n \log n)$ time.

On average, quick sort needs $O(\log n)$ such loop. (since the depth of the recursion tree for the average case is $O(\log n)$)

Therefore, the total time complexity for this algorithm is $O(n \log n \cdot \log n) = O(n \log^2 n)$.

Problem 5

Consider the problem as binary coding.

A binary string represents a contestant's team assignment for all the contests. For example, if the binary string for a contestant is 0101, it means that this contestant is in team P for the first contest, the other team Q for the second contest, P for the third, and Q for the fourth.

In this way, the orginal problem can be tranferred into the following problem:

> Given n numbers, how many bits do we need at least to ensure that for any two numbers, say A and B, A XOR $B \neq 0$?

The answer is $\log_2 n$ for $n = 2^k, k \in N$. Because $\log_2 n$ bits are able to display n different numbers. Since any two numbers A B are different from each other, A XOR $B \neq 0$ always holds true. In this way, all contestants are divided equally into the two teams.

Any solution less than $\log_2 n$ can't do this, since $m < \log_2 n$ bits can only express up to $2^m < n$ different numbers. So to fill up the left $n - 2^m$ numbers, there must be repeated numbers, so A XOR $B \neq 0$ is not satisfied.

And if n isn't any power of 2, then we need at least $\lceil \log_2 n \rceil$ bits.

Therefore, the contests for n contestants in G is designed as follow:

1. Let $|G| = s = \lceil \log_2 n \rceil$.
2. Generate n different s-bit binary numbers.
3. Assign the n binary numbers to the n contestants. Each i-th bit in the assigned string stands for the team the contestant should belong in the i-th contest ($1 \leq i \leq s$).

Problem 6

directed graph