Homework 1

1. Solve the following recurrence relations and give a Θ bound for each of them.

(a) $T(n) = 5T(\frac{n}{4}) + n$

By the master theorem:

$$T(n) = aT\left(\frac{n}{b}\right) + \Theta(n^d)$$

we can easily get that $a = 5, b = 4, d = 1$ and that $a > b^d$. Thus we can get that:

$$T(n) = \Theta\left(n^{\log_4 5}\right)$$

(b) $T(n) = 7T(\frac{n}{7}) + n$

By the master theorem:

$$T(n) = aT\left(\frac{n}{b}\right) + \Theta(n^d)$$

we can easily get that $a = 7, b = 7, d = 1$ and that $a = b^d$. Thus we can get that:

$$T(n) = \Theta(n^1 \log n) = \Theta(n \log n)$$

(c) $T(n) = 49T(\frac{n}{25}) + n^{\frac{3}{2}} \log n$

$$T(n) = 49T\left(\frac{n}{25}\right) + n^{\frac{3}{2}} \log n \to 49^i T\left(\frac{n}{25^i}\right) = 49^{i+1} T\left(\frac{n}{25^{i+1}}\right) + 49^i * \left(\frac{n}{25^i}\right)^{\frac{3}{2}} (\log n - \log 25^i)$$

$$T(n) = n^{\frac{3}{2}} \log n + 49 * \left(\frac{n}{25}\right)^{\frac{3}{2}} (\log n - \log 25) + \cdots + 49^i * \left(\frac{n}{25^i}\right)^{\frac{3}{2}} (\log n - \log 25^i)$$

$$T(n) = n^{\frac{3}{2}} \log n \left(1 + \frac{49}{125} + \ldots + \left(\frac{49}{125}\right)^i + \cdots\right) - 2n^{\frac{3}{2}} \log 5 \left(\frac{49}{125} + \cdots + i * \left(\frac{49}{125}\right)^i + \cdots\right)$$

$$T(n) = \frac{125}{76} * A * n^{\frac{3}{2}} \log n = \Theta(n^{\frac{3}{2}} \log n)$$

Where A is the simplification of the extra factors.

(d) $T(n) = 3T(n-1) + 2$

$$T(n) = 3T(n-1) + 2 \Leftrightarrow T(n) + 1 = 3(T(n-1) + 1) \Leftrightarrow Q(n) = 3Q(n-1)$$

$$Q(n) = 3Q(n-1) \Leftrightarrow Q(n) = \Theta(3^n) \Leftrightarrow T(n) + 1 = \Theta(3^n) \Leftrightarrow T(n) = \Theta(3^n)$$

2. A k-way merge operation. Suppose you have k sorted arrays, each with n elements, and you want to combine them into a single sorted array of kn elements. Design an efficient algorithm using divide-and-conquer (and give the time complexity).

Analysis: These k-sorted arrays can be regarded as the bottom level of the merge sort.

Plan: Divide and Conquer

Denote these arrays as $a_1 = [a_{11}, a_{12}, \dots, a_{1n}], \dots, a_k = [a_{k1}, a_{k2}, \dots, a_{kn}]$.

- Divide: Divide the input K sorted arrays into two subsets S_1 and S_2.

$$S_1 = \left[a_1, \dots, a_{\frac{k}{2}}\right], S_2 = [a_{\frac{k}{2}+1}, \dots, a_k]$$

- Recurse: Sort two subsets (small size problems).
  - Let $S_1' = \left[b_1, \dots, b_{\frac{k}{2}}\right], S_2' = [b_{\frac{k}{2}+1}, \dots, b_k]$ be the output sorted lists.
- Combine: Merge two sorted lists to one long sorted list.

  Denote two sorted arrays as $A = [a_1, a_2, \dots, a_n], B = [b_1, b_2, \dots, b_m]$ and result array C
  - Maintain 2 pointers i=1, j=1
  - Repeat
    - Append $\min\{a_i, b_j\}$ to C
    - If a_i is smaller, then move i to i+1. If b_j is smaller, then move j to j+1.
    - Break if i>n or j>m. (n may not be equal to m because maybe k is odd)
  - Append the reminder of the non-empty list to C
- Basic solver: If the array is only one, then it is already sorted.

Time complexity

It is divided by $\log k$ times, and the total number of operations in each subset is $\Sigma_{i=1}^{\infty} \frac{k}{2^i} 2^i n = kn$

Thus, the time complexity is $O(kn \log k)$.

By one different analysis we can also get the same time complexity:

	Number of elements	Number of divisions	Time complexity
Merge sort	n	log n	n log n
K-way merge sort	kn	log k	kn log k

3. You are interested in analyzing some hard-to-obtain data from two separate databases. Each database contains n numerical values-so there are 2n values total-and you may assume that no two values are the same. You'd like to determine the median of this set of 2n values, which we will define here to be the n-th smallest value. However, the only way you can access these values is through queries to the databases. In a single query, you can specify a value k to one of the two databases, and the chosen database will return the k-th smallest value that it contains. Since queries are expensive, you would like to compute the median using as few queries as possible. Give an algorithm that find the median value using O(log n) queries.

Analysis:

Denote these 2 databases as (ideal sorted): $b_1 = [a_1, a_2, \dots, a_n], \dots, b_2 = [c_1, c_2, \dots, c_n]$

Obviously, if the median = a_n, then when these two databases lose the one half, the median stays no change. It means: a_n is still the median for $b_1{}' = \left[a_{\frac{n}{2}}, a_{\frac{n}{2}+1}, \dots, a_n\right], \dots, b_2{}' = [c_1, c_2, \dots, c_{\frac{n}{2}-1}]$

Because each division is cutting half of the array, the total queries must be log n and leads to O(log n) algorithm.

Plan:

- Repeat
  - Maintain two median values for each database: $m_1 = mid(b_1), m_2 = mid(b_2)$
  - If m_1 is smaller, then save the right half of b_1 and the left half of b_2.

    If m_2 is smaller, then save the right half of b_2 and the left half of b_1.
  - Break if $m_1 = m_2$ or the size of each database = 1.
- If $m_1 = m_2$, then this is the answer. Else, the smaller one is the answer.

4. Show that the QuickSort algorithm runs in $O(n^c)$ time on average for some constant c < 2 if the pivot is chosen randomly.

Remark: The algorithm actually runs in $O(n \log n)$ time on average, but for this question it suffices to give an analysis better than $O(n^2)$.

Proof:

Suppose the sorting array is: $[a_1, a_2, \dots, a_n]$

For each step in the Quick sort, we randomly choose one pivot and one partition. For each partition, the possibility to be chosen is $\frac{1}{n}$. Thus, for each partition$(a_1, \dots, a_{i+1})(a_{i+2}, \dots, a_n)$, we will have the situation:

$$T(n) = T(i) + T(n-i-1) + O(n), \forall i \in [0, n-1]$$

$$E\big(T(n)\big) = \frac{1}{n} E\left(\Sigma_{i=0}^{n-1}\big(T(i) + T(n-i-1) + O(n)\big)\right)$$

$$E\big(T(n)\big) = \frac{2}{n}\Sigma_{i=0}^{n-1} E\big(T(i)\big) + O(n)$$

In Quick sort, obviously we don't need to do sort when n = 1, and we should do at most 1 swap when n = 2. Thus we can get that:

$$E\big(T(1)\big) = 0 \le O(1), E\big(T(2)\big) = 0.5 \le O(2) \rightarrow \text{Induction Base}$$

Goal: $\exists c < 2, \text{such that } E\big(T(n)\big) \le O(n^c)$

Induction base: When n = 1 or n = 2, $E\big(T(n)\big) \le O(n^c)$ is obviously true.

Induction hypothesis: $\forall k \ge 1,\ E\big(T(n)\big) \le O(n^c)$

Induction Proof: Set $n = k + 1$.

By induction we have: $E\big(T(k)\big) = \frac{2}{k}\Sigma_{i=0}^{k-1} E\big(T(i)\big) + O(k) \le O(k^c)$

Then we can have that: $E\big(T(n)\big) = E\big(T(k+1)\big) = \frac{2}{k+1}\Sigma_{i=0}^{k} E\big(T(i)\big) + O(k+1)$

$$E\big(T(n)\big) = \frac{2}{k+1}\left(\frac{k}{2} E\big(T(k)\big) - O(k) + E\big(T(k)\big)\right) + O(k+1)$$

$$E\big(T(n)\big) = \frac{k+2}{k+1} E\big(T(k)\big) + O(k+1) - \frac{k}{k+1} O(k) = \frac{k+2}{k+1} E\big(T(k)\big) + \frac{1}{k+1} O(k+1)$$

$$E\big(T(n)\big) = E\big(T(k)\big) + \frac{T(k) + O(k+1)}{k+1} \le O(k^c) + \frac{O(k^c) + O(k+1)}{k+1} \le O((k+1)^c) = O(n^c)$$

Thus, by induction we can prove that the Quick sort algorithm runs in $O(n^c)$ time on average for some constant c<2.

5. Given an $n \times m$ 2-dimensional integer array A[0; ... ; n-1; 0; ... ;m-1] , where A[i; j] denotes the cell at the i-th row and the j-th column, a local minimum is a cell A[i; j] such that A[i; j] is smaller than each of its four adjacent cells A[i-1; j];A[i+1; j];A[i; j-1];A[i; j+1]. Notice that A[i; j] only has three adjacent cells if it is on the boundary, and it only has two adjacent cells if it is at the corner. Assume all the cells have distinct values. Your objective is to find one local minimum (i.e., you do not have to find all of them).

(a) Suppose m = 1 so A is a 1-dimensional array. Design a divide-and-conquer-based algorithm for the problem above. Write a recurrence relation of the algorithm, and analyze its running time.

Analysis:

If $A[i] < A[i+1]$, then we can find at least one local minimum in the period $[0; i]$.

Proof: There are just 2 situations.

1: The array is sorted in the period $[0; i]$. Obviously the local minimum is A[0].

2: $\exists j \in [0, i-1], A[j] > A[j+1]$. This implies that A[j+1] is a local minimum.

Plan:

- Maintain 2 pointers l=0, r=n-1
- Repeat
  - Maintain the mid-value m = (l+r)/2. Compare A[m] with A[m-1], A[m+1].
  - Break if A[m-1] > A[m] and A[m] < A[m+1]. (i.e., local minimum)
  - If A[m]<A[m+1], then move l to mid+1. Else, move r to mid-1.
- One local minimum is A[m].

Recurrence relation:

$$T(n) = T\left(\frac{n}{2}\right) + O(1)$$

Running time:

$$T(n) = O(\log n)$$

Remark: This problem is in leetcode-162. I have finished it at 2021/09/15 20:58.

(b) Suppose m = n. Design a divide-and-conquer-based algorithm for the problem above. Write a recurrence relation of the algorithm, and analyze its running time.

Analysis:

This problem is similar with problem (a).

Plan:

- Maintain 2 diagonal pointers $l = [0; 0], r = [n-1, n-1]$.
- Repeat
  - Maintain the mid-row $m_r = \frac{l[0]+r[0]}{2}$, the mid column $m_c = \frac{l[1]+r[1]}{2}$.
  - Find the minimum element $A[i][j] = \min\left\{\min_{l[0]\le k\le r[0]} A[k][m_l] \ , \min_{l[1]\le k\le r[1]} A[m_r][k]\right\}$
  - Break if $A[i][j]$ is the local minimum. This means that A[i][j] satisfies:

    $A[i][j] < \min\{A[i][j \pm 1], A[i \pm 1][j]\}$ (If overflow, this can be regarded as true.)
  - Compare $A[i][j]$ with $A[i][j \pm 1]$ and $A[i \pm 1][j]$. Go down the smaller direction.

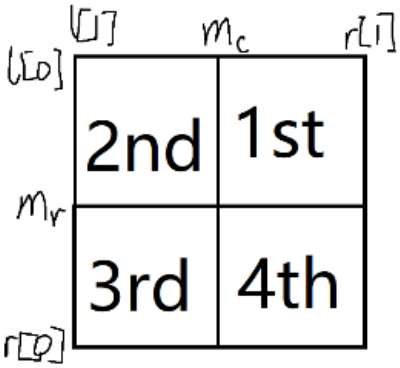
    Choose one quadrant to "roll downhill".

    The first quadrant: $l = [l[0], m_c]$ $r = [m_r - 1, r[1] - 1]$

    The second quadrant: $l = [l[0], l[1]]$ $r = [m_r - 1, m_c - 1]$

    The third quadrant: $l = [m_r + 1, l[1] + 1]$ $r = [r[0], m_c]$

    The fourth quadrant: $l = [m_r + 1, m_c + 1]$ $r = [r[0], r[1]]$



- One local minimum is $A[i][j] = A[l][r]$.

Recurrence relation:

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

Running time:

$$T(n) = O(n)$$

(c) Generalize your algorithm such that it works for general m and n. The running time of your algorithm should smoothly interpolate between the running times for the first two parts.

Analysis:

This problem can be regarded as the combination of problem (a) and (b).

Plan:

- Maintain 2 diagonal pointers $l = [0; 0], r = [n-1, m-1]$.
- Repeat
  - Maintain the mid-row $m_r = \frac{l[0]+r[0]}{2}$, the mid column $m_c = \frac{l[1]+r[1]}{2}$.
  - **Break if $l[0] = l[1]$ or $r[0] = r[1]$.**

    This step is different from problem (b). We should quit from the loop body when we cut the 2-dimentional array into 1-dimentional. Then we can apply the method used in problem (a).
  - Find the minimum element $A[i][j] = \min\left\{\min_{l[0]\le k\le r[0]} A[k][m_l] \ , \min_{l[1]\le k\le r[1]} A[m_r][k]\right\}$
  - Break if $A[i][j]$ is the local minimum. This means that A[i][j] satisfies:

    $A[i][j] < \min\{A[i][j \pm 1], A[i \pm 1][j]\}$ (If overflow, this can be regarded as true.)
  - Compare $A[i][j]$ with $A[i][j \pm 1]$ and $A[i \pm 1][j]$. Go down the smaller direction.

    Choose one quadrant to "roll downhill".

    The first quadrant: $l = [l[0], m_c]$ $r = [m_r - 1, r[1] - 1]$

    The second quadrant: $l = [l[0], l[1]]$ $r = [m_r - 1, m_c - 1]$

    The third quadrant: $l = [m_r + 1, l[1] + 1]$ $r = [r[0], m_c]$

    The fourth quadrant: $l = [m_r + 1, m_c + 1]$ $r = [r[0], r[1]]$
- If $A[i][j]$ is the local minimum, then this is the answer. (Quit without more operations)
- Else: Repeat in one-row (or one-column) to find the local minimum.

  Now we apply the method used in problem(a). Suppose now the remained part is one row. (One column is nearly the same). Denote the row array as: $A[l[0]; l[1]] \to A[l[0]; r[1]]$
  - Maintain the mid-value $m = \frac{l[1]+r[1]}{2}$. Compare A [m] with A[m-1], A[m+1].
  - Break if A[m-1] > A[m] and A[m] < A[m+1]. (i.e., local minimum)
  - If A[m]<A[m+1], then move l to mid+1. Else, move r to mid-1.
- One local minimum is $A[l[0]; m]$.

Running time analysis:

From this algorithm, we can easily get that when the scale of the 2-dimentional array is $n \times m$, then it can firstly be cut into one-row or one-column conditions (for the worst situation) and then it can use the local minimum algorithm in problem(a) to deal with. Thus, this running time cost should satisfy:

$$O(\log \min\{n, m\}) < T(n, m) < O(\max\{n, m\})$$

It meets the requirement of the time cost: *The running time of your algorithm should smoothly interpolate between the running times for the first two parts.*

6. How long does it take you to finish the assignment (include thinking and discussing)? Give a score (1,2,3,4,5) to the difficulty. Do you have any collaborators? Write down their names here.

   Almost 4 hours. Difficulty: 4.

   Collaborators: Zhiteng Li(Problem 4). Xiang'ge Huang(Problem 4).