Lab02-Divide and Conquer

CS214-Algorithm and Complexity, Xiaofeng Gao, Spring 2019.

∗ If there is any problem, please contact TA Jiahao Fan.
∗ Name:KylinChen Student ID:517030910155 Email: k1017856853@icloud.com

1. Consider the following recurrence:

$$T(n) = \begin{cases} 0 & \text{if } n = 1 \\ 2T(n/2) + O(n \log n) & \text{if } n = 2^k \text{ and } k \geq 1 \end{cases}$$

(a) Solve $T(n)$ (in the form of O-notation) by recurrence tree. Detailed derivation is required.

Solution. In this part, we will first solve T(n) by recurrence tree, an then try to test and verify our conclusion by recursive equation.

- **Analysis:** For this question, we can devide the original problem several times until each subproblem just costs T(1).
  Therefore, we can draw this recurrence tree Fig. 1. In this picture, we can clearly see that every node(except root node) have the half scale of its father node, then we sum up all the O-notation values to get the T(n)'s O-notation.

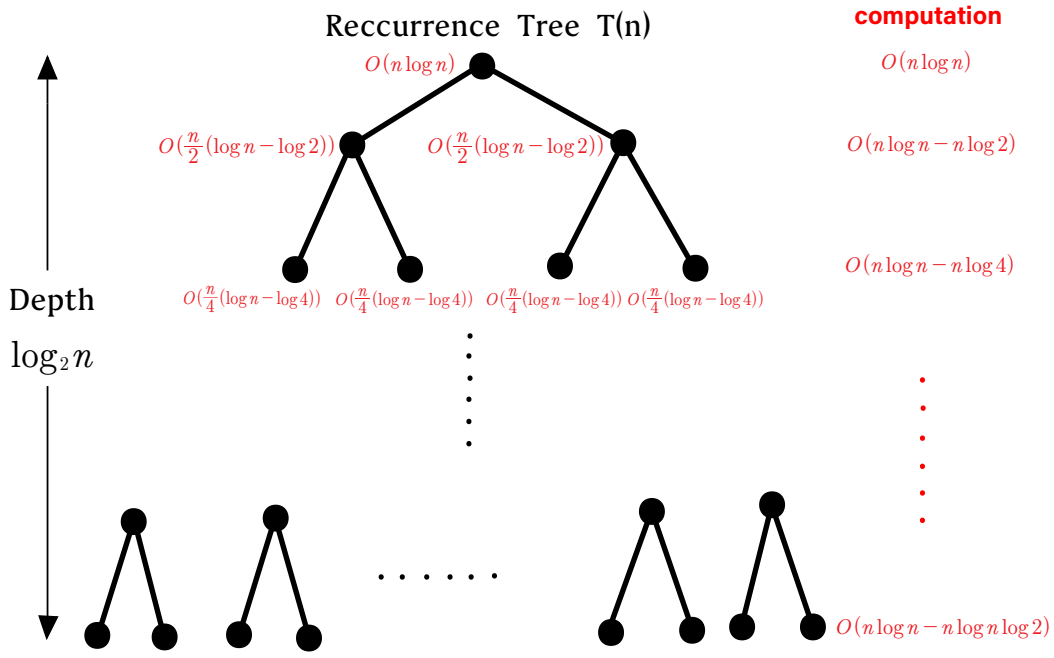

Figure 1: Reccurrence Tree

- **Calculation:** (In order to simplify the calculation process, we ignore the O-notation except final result)

$$T(n) = nlog(n)+n(log(n)-log(2))+n(log(n)-log(4))+\cdots+n(log(n)-log(n)log(2))$$

$$\Rightarrow T(n) = nlog(n)log(n) - n(log(2) + 2log(2) + 3log(2) + ... + log(n)lg(2))$$

$$\Rightarrow T(n) = nlog(n)log(n) - nlog(2) \times \frac{((1 + log(n))log(n))}{2} = n \cdot log(n) \times log(n)$$

It means, in O-notation, $T(n) = O(n \cdot log^2(n))$.

- **Recursive Proof:** Now we know, if $n = 2^k$ *and* $k \geq 1$, $T(n) = 2T(\frac{n}{2}) + O(nlogn)$, in order to simplify this recurrence, we substitute O(nlogn) to $\Theta(nlog_2 n)$, it means T(n)=2T($\frac{n}{2}$)+$nlog_2 n$, it's obvious if we get the Θ-notation, in that way, it must be $O - notation$, too.
  We assume $n = 2^m$, in which $m \geq 1$ Then we can make a substitution

$$T(2^m) = 2T(2^{m-1}) + 2^m log_2(2^m) = 2T(2^{m-1}) + m \cdot 2^m$$

  Then, we define $f(m) = T(2^m)$, we get that

$$\begin{aligned} f(m) &= 2f(m-1) + m \cdot 2^m \\ &= 2(2 \cdot f(m-2) + (m-1) \cdot 2^{m-1}) + m \cdot 2^m \\ &= 4(2 \cdot f(m-3) + (m-2) \cdot 2^{m-2}) + (m-1) \cdot 2^m + m \cdot 2^m \\ &= 8f(m-3) + (m-2) \cdot 2^m + (m-1) \cdot 2^m + m \cdot 2^m \\ &= \cdots\cdots \end{aligned}$$

  In proceeding iteration, we can get that

$$\begin{aligned} f(m) &= 2^m \cdot f(0) + 2^m \cdot (1 + 2 + 3 + 4 + 5 + \cdots + (m-1) + m) \\ &= 2^m \cdot f(0) + m \cdot (m+1) \cdot 2^{m-1} \\ &= 2^m \cdot f(0) + m(m-1)2^{m-1} \end{aligned}$$

  Therefore, $T(n) = nT(1) + n(\frac{log_2(n)(1+log_2(n))}{2}) = \Theta(n \cdot log^2 n)$ It means in O-notation, T(n)=$O(n \cdot log^2 n)$, the recurrence tree get the correct result.

(b) Can we use the Master Theorem to solve this recurrence? Please explain your answer.

Solution. From my perspective, this recurrence can't use Master Theorem to solve, reasons can be summed up as follows:

- For the recurrence $T(n) = 2T(\frac{n}{2})+nlogn$, $a = 2, b = 2, f(n) = nlogn, and\ n^{log_b(a)} = n$ In this case, f(n) is asymptotically larger than $n^{log_b(a)}$, but not polynomially larger. Thus, Master Theorem doesn't apply here.
- In wikipedia Master theorem (analysis of algorithms), we can find a ε-definition, for this question, the ratio $logn$ is asymptotically less than n^{ε} for any positive ε.
- In the recursive proof in part(a), we can find this this solving process is totally like Master Theorem's derivation. And if we use Master Theorem in this question, it can get correct answer. So, it may use **General Master Theorem**, which can refers to MIT edition.

2. Given any array num, find the number of pairs (i, j) satisfying $i < j$ and $num[i] > 2 \times num[j]$. For example, if $num = [1, 3, 2, 3, 1]$, the answer should be 2.

(a) Design an algorithm to solve this problem using divide-and-conquer strategy and complete the implementation in the provided C/C++ source code. (The source code *CodePairs.cpp* is attached on the course webpage.)
(.cpp file is attached to the compress file)

(b) Write a recurrence for the running time of your algorithm and solve it using the Master Theorem directly.

Solution.

(a) This algorithm relies on **MergeSort Algorithm**, which can be achieved by private function(refers to .cpp file) PairsCount. Its idea can be summed up as follows:

- According to **DevideAndConquer**, if the input array's size is more than one, we can devide it into sub-array A and sub-array B, which refers to Fig. 2
- Then we recursively calling the function PairCount to accumulate the number of aimed-pair. In this process, we use mergesort to sort this array, ao that we can get $O(n)$ complexity every count step.
- For accumulate the aimed-pair, we just do it between sub-array A and B. And if a1≥b2, we can directly claim a2≥b2 because array A is a nondecreasing array, therefore we just get linear time-complexity every recurrence.

Repeat these steps until recursive termination, we can get the returning value count for correct answer.

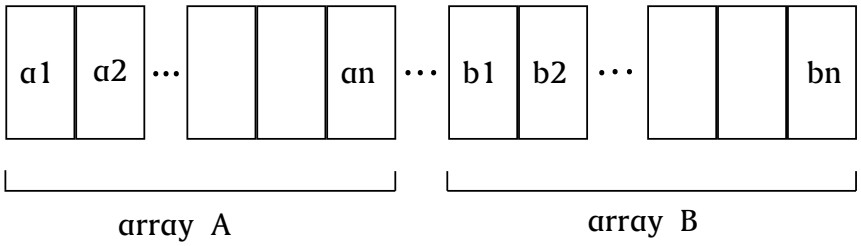

Figure 2: Devide one array into two subarrays

(b) According to algorithm subscription in (a), we can make an analysis:

- In each step we divide the array into 2 sub-arrays, and hence, every sub-array takes $\frac{n}{2}$ problem size.
- Additional $O(n)$ work needs to be done to count the inversions and to merge the 2 sub-arrays after sorting.

Therefore, we can get recurrence:

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

By **Master Theorem**, $log_b a = 1 = d$, so the running time T(n)=O(nlog(n)).

3. **Transposition Sorting Network**: A comparison network is a **transposition network** if each comparator connects adjacent lines, as in the network in Fig. 5.

(a) Prove that a transposition network with n inputs is a sorting network if and only if it sorts the sequence $\langle n, n-1, \cdots, 1\rangle$. (Hint: Use an induction argument analogous to the *Domain Conversion Lemma.*)

Proof. 1

- First, we will prove that if a comparator(two line sorting network) can correct sort number pair (for example $< 1, 2 >$, $< 2, 3 >$, $< 3, 4 > \cdots < n-1, n >$) it can sort any other number pair.

For the comparator in Fig. 3. When inputs is {x,y}, the upper output must be min{x,y} while lower is max{x,y}.
Then we can find a linear function f(x), which is monotonically increasing. Consequently we have the identity:

$$min\{f(x), f(y)\} = f(min\{x, y\})$$
$$max\{f(x), f(y)\} = f(max\{x, y\})$$

Thus, we can make a linear conversion from {x,y} to aimed {f(x),f(y)}, it is also correctly sorted.

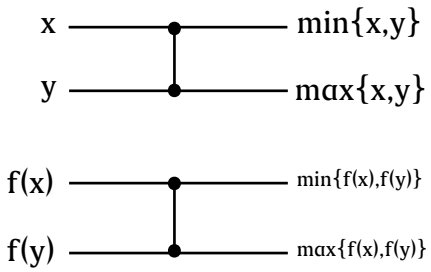

Figure 3: Comparator(Two-line Sorting Network)

- If we assume k-1 line transposition network is sorting network (k-1 values are correctly sorted using previous comparators), then we want to prove k line transposition network is sorting network.
  Once we attach k-th line to k-1 network, it must be added by the sides, otherwise it can attribute to previous already-proved condition.
  From Fig. 4 We can obviously get that once k-th line is added, we will add n-1 comparators.
  Since,

$$n_{(comparators)} = n - 1 = n_{(beforeArray)} \geq Move - Step_{max}$$

  Therefore, we can claim k values are well sorted and k line transposition network is sorting network.

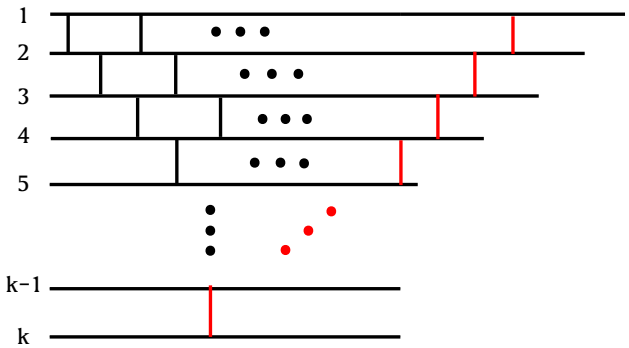

Figure 4: Induction from k-1 to k

According to mathmatical induction, we can claim if and only if a transposition network sorts the sequence $(n, n-1, \cdots, 1)$, it is a sorting network.

Proof. 2

We can change original proposition's expression as follows:

For a data sequence $X = x_1, x_2, x_3, \cdots, x_n$, we use symbol $i_{X,k}$ to express the data on i-th line after going through k-th comparator.

Equivalent proposition: For data sequence $D = (n, n-1, n-2, \cdots, 1)$ and arbitrary sequence $X = (x_1, x_2, x_3, \cdots, x_n)$, with the final L-th comparator and $i < j$, $i_{D,L} < j_{D,L}$ if and only if $i_{X,L} < j_{X,L}$.

We use induction to prove the equivalent proposition, so we assume: for r lines in this transposition network, after going through k-th comparator, for any $1 \leq i \leq j \leq r$, $i_{D,k} \leq j_{D,k}$. (Our aim is to derive $i_{X,k} \leq j_{X,k}$)

- If $|i - j| = 1$, no matter whether i,j is the last reverse pair or not, comparator will sort it in increasing order, so it's obvious to get $i_{X,k} \leq j_{X,k}$.
- If $|i - j| \neq 1$, in this time, i and j are not connected by comparator, which means the data on these two line is kept, that is to say, $i_{X,k} = j_{X,k-1}$ and $i_{X,k} = j_{X,k-1}$. By induction assumption, we can get $i_{X,k-1} \leq j_{X,k-1}$, so $i_{X,k} \leq j_{X,k}$.

In summary, a transposition network with n inputs is a sorting network if and only if it sorts the sequence $D = (n, n-1, n-2, \cdots, 1)$.

(b) (Bonus) Given any $n \in \mathbb{N}$, write a program using Tkinter in Python to draw a figure similar to Fig. 5 with n input wires.

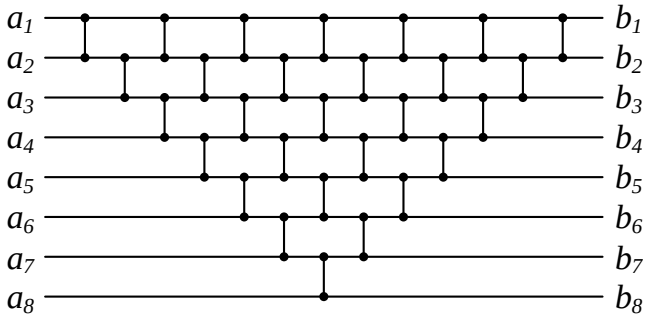

Figure 5: A Transposition Network Example

Solution. The attached file **transposition.py** is submission python code.

Require python package Tkinter.

Test pass with PyCharm, Anaconda 3.7 version, macOS.

Remark: include your .cpp, .py, .pdf and .tex files in your uploaded .rar or .zip file.

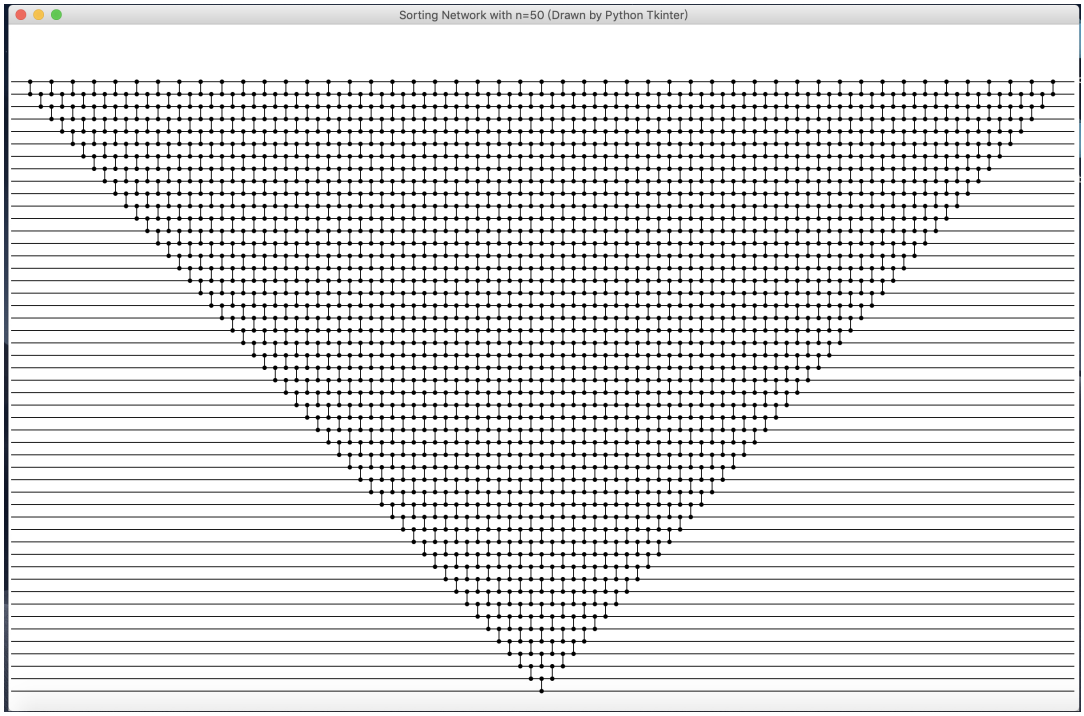

Figure 6: **transposition.py** Test with n=50