Lab02-Divide and Conquer

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

∗ If there is any problem, please contact TA Jiahao Fan.

∗ Name:Tianyao Shi   Student ID:517021910623   Email: sthowling@sjtu.edu.cn

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. It can be inferred that the scale of the problem n is power of two, i.e. $n = 2^k,\ k \in \mathbb{N}$. In other words, we have $k = \log n$. We denote computation cost at the i-th level of the recurrence tree to be the cost of merging the results in the $(i+1)$-th level. Let the original problem be the first level of the recurrence tree, so there are $(k+1)$ levels in total.

For the first level, its computation cost is $O(n \log n)$. For the second level, its computation cost is $O(\frac{n}{2} \log \frac{n}{2}) = O(n(\log n - 1))$. And for the i-th level, its computation cost is

$$O(\frac{n}{2^{i-1}} \log \frac{n}{2^{i-1}}) = O(n(\log n - i + 1)).$$

Notice that at $(k+1)$-th level are all leaf nodes whose problem scales are all $n = 1$ and $T(n) = 0$ according to the definition, therefore the computation cost of the $(k+1)$-th level is 0. The recurrence tree is shown in Figure 1.

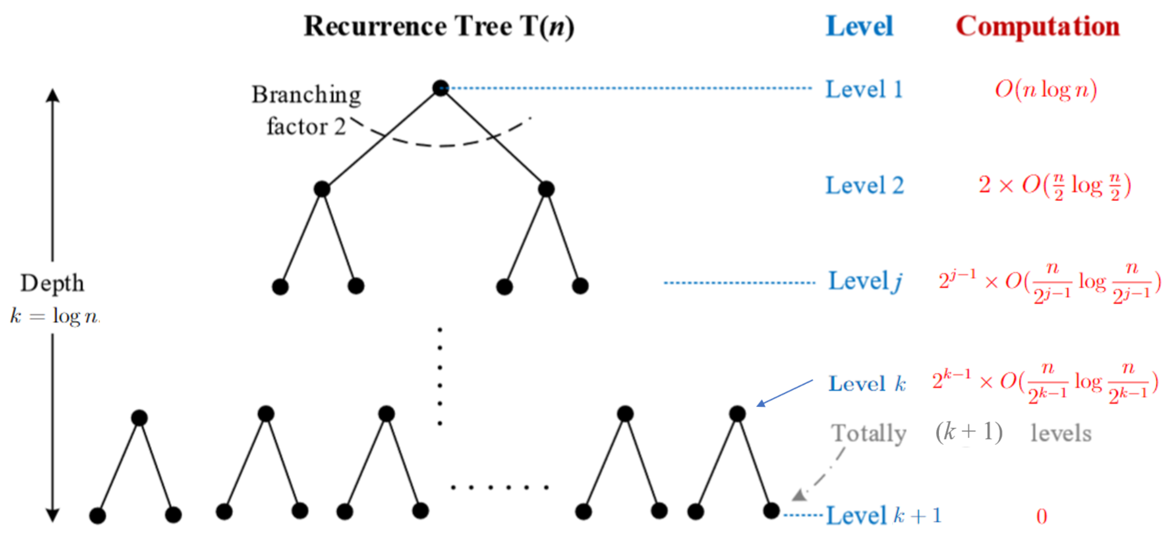
Recurrence Tree T(n)
Level
Computation
Branching factor 2
Level 1
$O(n \log n)$
Level 2
$2 \times O(\frac{n}{2} \log \frac{n}{2})$
Depth
$k = \log n$
Level j
$2^{j-1} \times O(\frac{n}{2^{j-1}} \log \frac{n}{2^{j-1}})$
Level k
$2^{k-1} \times O(\frac{n}{2^{k-1}} \log \frac{n}{2^{k-1}})$
Totally $(k+1)$ levels
Level $k+1$
0

Figure 1: Recurrence Tree for Problem 1

We sum up the total computation cost, and derive that

$$\begin{aligned} T(n) &= \sum_{i=1}^{k+1} O(n(\log n - i + 1)) = \sum_{i=0}^{k} O(n(\log n - i)) \\ &= \sum_{i=0}^{\log n} O(n(\log n - i)) = O\left((n\frac{(\log n)^2}{2}\right) = O\left(n(\log n)^2\right) \end{aligned}$$

□

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

Solution. The answer is that no, we cannot. But the reason is not a simple claim that in the recurrence relation

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

the merging cost $O(n \log n)$ does not match the form $O(n^d)$ in the Master Theorem. In fact, the Master Theorem has a stricter form [1]:

Theorem 1. *Assume that constant $a \geq 1$, $b > 1$, $f(n)$ is a function of n and $T(n)$ is a nonnegative integer. The recurrence relation gives that $T(n) = aT(b/n) + f(n)$, then*

i. if $\exists \epsilon > 0$, s.t. $f(n) = O(n^{\log_b a - \epsilon})$, then $T(n) = \Theta(n^{\log_b a})$.

ii. if $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \log n)$.

iii. if $\exists \epsilon > 0$, s.t. $f(n) = \Omega(n^{\log_b a + \epsilon})$, and $\exists N \in \mathbb{N}$ and constant $c < 1$, s.t. $\forall n > N$, $af(n/b) \leq cf(n)$, then $T(n) = \Theta(f(n))$.

For example, say there is a recurrence relation

$$T(n) = 3T(\frac{n}{4}) + n \log n.$$

In this case we have $n^{\log_b a} = n^{0.793}$, set $\epsilon = 0.2$, then $f(n) = n \log n = \Theta(n^{0.993})$. When $c = 3/4$ and n is big enough, we have

$$af(\frac{n}{b}) = 3 \cdot \frac{n}{4} \log \frac{n}{4} \leq \frac{3}{4} \cdot n \log n = cf(n),$$

which suggests that the relation satisfies the third case in Theorem 1. Thus we have $T(n) = \Theta(n \log n)$.

Back to this case, $n^{\log_b a} = n^1$ while $f(n) = O(n \log n)$. It is clear that it does not satisfy the second case in Theorem 1. Besides for the first and third case, we cannot find such $\epsilon > 0$ so that $n \log n = O(n^{1-\epsilon})$ or $n \log n = \Theta(n^{1+\epsilon})$. Therefore we cannot use the Master Theorem in this problem. $\square$

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 *Code-Pairs.cpp* is attached on the course webpage.)

Solution. The thinking behind this solution to be given is actually interesting, so I would like to be somewhat verbose to write down the following 3 algorithms. Let us denote such pairs as inversion pairs.

The problem itself is not complicated, and most of us would probably come up with the method of exhaustive search, i.e. traversing all C_n^2 pairs to check if each satisfies the requirement.

A possible algorithm for exhaustive search is shown in Alg. 1.

Algorithm 1: ExhaustiveSearch

```
Input: An array $A[1, \cdots, n]$
Output: Number of inversing pairs $Np$ in $A$

1 $i \leftarrow 1$;
2 $Np \leftarrow 0$;
3 for $i \leftarrow 1$ to $n-1$ do
4     for $j \leftarrow i+1$ to $n$ do
5         if $A[i] > 2 \times A[j]$ then
6             $Np \leftarrow Np + 1$;
7 return $Np$;
```

Clearly this algorithm would have an $O(n^2)$ complexity as line 6 would be executed C_n^2 times whatever the condition is. And this does not use the technique of divide-and-conquer either.

So we come up with a new divide-and-conquer design: this time we divide the array into two halves. We first consider the inversion pairs within each half, and then consider the inversion pairs between two halves, as is shown in Alg. 2.

Algorithm 2: Divide-And-Conquer1

```
Input: An array $A[1, \cdots, n]$, starting pointer $L$, ending pointer$R$
Output: Number of inversing pairs $Np$ in $A$

1 if $L$=$R$ then
2     return 0
3 $M \leftarrow (L+R)/2$;
4 $Np \leftarrow$ Divide-And-Conquer1(array,$L, M$)+Divide-And-Conquer1(array,$M+1, R$);
5 for $i \leftarrow 1$ to $M$ do
6     for $j \leftarrow M+1$ to $R$ do
7         if $A[i] > 2 \times A[j]$ then
8             $Np \leftarrow Np + 1$;
9 return $Np$;
```

Say that the original problem costs $T(n)$, then each half costs $T(n/2)$, and merging costs $O(n/2 \cdot n/2) = O(n^2)$. According to the Master Theorem, the total cost is still $O(n^2)$, as the merging part is still exhaustive search, which traverse all pairs between the two halves.

The problem lies in that division does not necessarily cut down the number of comparison. A possible solution to this would be sorting the two halves before merging them, so that there lies a chance that we do not have to check all pairs. Therefore we combine the Merge Sort with Alg. 2, to derive a new algorithm as is shown in Alg. 3.

Algorithm 3: Divide-And-Conquer2

Input: An array $A[1, \cdots, n]$, starting pointer L, ending pointer$R$
Output: Number of inversing pairs Np in A

```
1  if $L$=$R$ then
2      return $0$
3  $M \leftarrow (L+R)/2$;
4  $Np \leftarrow$ Divide-And-Conquer2(array,$L$, $M$)+Divide-And-Conquer2(array,$M+1$, $R$);
5  for $j \leftarrow M+1$ to $R$ do
6      while $A[i] \leq 2 \times A[j]$ and $i \leq M$ do
7          $i \leftarrow i+1$;
8      $Np \leftarrow Np + (M - i + 1)$;
9  Merge(array, $L$, $M$, $R$);      // Invoking merge sort to get an ordered array
10 return $Np$;
```

By invoking merge sort, we ensure that every time we consider the inversions between two halves, the two halves themselves are nondecreasing. Therefore, if $A[i] > 2 \times A[j]$, everything between $A[i]$ and $A[M]$ is larger than two times $A[j]$, which results an $(M - i + 1)$ increase of Np. This way there would at most be $\lceil (R-L)/2 \rceil$ comparisons between two elements in a pair in line 5 to 8 in Alg. 3. For merge sort, the time complexity is $O(\lceil (R-L)/2 \rceil) = O(n)$. So the total merging cost has a complexity of $O(n)$, which results in reduction on total complexity according to the Master Theorem. $\square$

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

Solution. The following recurrence relation can be derived from the pseudo code above:

$$T(n) = \begin{cases} 0 & \text{if } n = 1, \\ 2T(\lceil n/2 \rceil) + O(n) & \text{otherwise,} \end{cases}$$

where $\log_b a = 1 = d$. According to the Master Theorem, we have $T(n) = O(n \log n)$. $\square$

3. **Transposition Sorting Network**: A comparison network is a **transposition network** if each comparator connects adjacent lines, as in the network in Fig. 2.

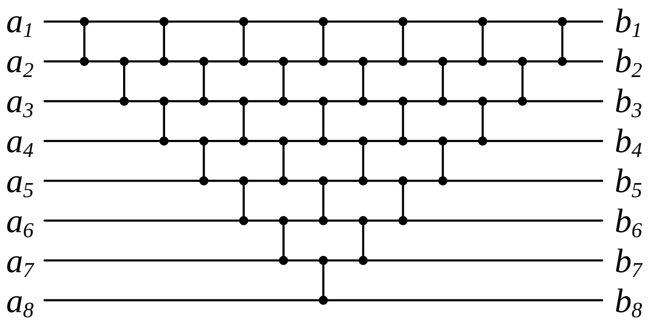

Figure 2: A Transposition Network Example

(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. The descending sequence $\langle n, n-1, \cdots, 1\rangle$ has the largest number of inversion pairs, $Num_i = C_n^2$, among all sequences whose length is n. In transposition networks, comparators only exist between two adjacent lines, so each comparator can at most cut down Num_i by one.

Transposition network in Figure 2 uses just C_n^2 comparators to sort $\langle 8, 7, \cdots, 1\rangle$ successfully, with no comparator wasted. Intuitively, if we set an arbitrary sequence as the input to this network, then after any comparator the output sequence should be no worse than when inputing the descending sequence, or in other words, is closer to the fully sorted ascending sequence. This intuition leads to Lemma 2, though the proof of it is not so elegant.

Lemma 2. *For an input sequence $S = \langle a_1, a_2, \cdots, a_n\rangle$, let us denote the element on the wire i after the k-th comparator as $i_{S,k}$. Say there are two wires i, j with $1 \le i < j \le n$(wires are numbered 1 to n starting from the very top). If when the input sequence is $S_d = \langle n, n-1, \cdots, 1\rangle$, $i_{S_d,k} < j_{S_d,k}$, then for an arbitrary input sequence S, $i_{S,k} \le j_{S,k}$.*

Before we start the proof, facts about "k-th comparator" have to be clarified. As is shown in Figure 2, there can be many parallel comparators with the same depth. To avoid misunderstanding, we define that there is only one k-th comparator in a network once k is finite, and $\forall\ l < k$, the depth of l-th comparator is not larger than that of k-th comparator. The detail how we number the comparators is not important.

Proof of Lemma We conduct induction on k to accomplish this proof.

Basis Step: Consider the network before any comparators, i.e. $k = 0$. For any $i < j$, we have $i_{S_d,0} > j_{S_d,0}$ since the input sequence is descending. Therefore, we make no claim about any $i_{S,0}$.

Induction Hypothesis: Assume that for $(k-1)$-th comparator, we have $i_{S_d,k-1} < j_{S_d,k-1}$ and $i_{S,k-1} \le j_{S,k-1}$.

Induction Proof: For k-th comparator, we need to show that if $i_{S_d,k} < j_{S_d,k}$, $i_{S,k} \le j_{S,k}$. This can be divided into several cases.

i. Wire i and j are adjacent and the k-th comparator is a comparator between i, j. Obviously, for the working principle of a comparator, we have $i_{S,k} \le j_{S,k}$.

ii. Neither wire i or j is connected to the k-th comparator. In this case, the outputs of i, j remain their previous state, i.e. $i_{S,k} = i_{S,k-1}$, $j_{S,k} = j_{S,k-1}$.
According to the hypothesis, we have $i_{S,k-1} \le j_{S,k-1}$. Thus, $i_{S,k} \le j_{S,k}$.

iii. One of wire i, j is connected to the k-th comparator. For convenience, let us denote the upper wire connected to the k-th comparator as a and the lower as b. This case can be further divided into four subcases,which are:

   A. $i < a$, $j = a$. Wire i is not connected, so $i_{S,k} = i_{S,k-1}$, $i_{S_d,k} = i_{S_d,k-1}$. According to the hypothesis, since $i < a < b$ and $i_{S_d,k} < j_{S_d,k} = \min\{a_{S_d,k-1}, b_{S_d,k-1}\} \le \max\{a_{S_d,k-1}, b_{S_d,k-1}\}$, we have $i_{S,k-1} \le a_{S,k-1}$, $i_{S,k-1} \le b_{S,k-1}$. Therefore, we get

$$i_{S,k} = i_{S,k-1} \le \min\{a_{S,k-1}, b_{S,k-1}\} = j_{S,k}.$$

   B. $i < a$, $j = b$. This time we have $j_{S,k} = \max\{a_{S,k-1}, b_{S,k-1}\}$.

   - If $a_{S_d,k-1} < b_{S_d,k-1}$, we have that

$$i_{S_d,k-1} = i_{S_d,k} < j_{S_d,k} = \max\{a_{S_d,k-1}, b_{S_d,k-1}\} = b_{S_d,k-1}.$$

   According to the hypothesis, $i_{S,k-1} \le b_{S,k-1}$. That is to say,

$$i_{S,k} = i_{S,k-1} \le b_{S,k-1} = \max\{a_{S,k-1}, b_{S,k-1}\} = j_{S,k}.$$

- If $a_{S_d,k-1} > b_{S_d,k-1}$, we have that

$$i_{S_d,k-1} = i_{S_d,k} < j_{S_d,k} = \max\{a_{S_d,k-1}, b_{S_d,k-1}\} = a_{S_d,k-1}.$$

  According to the hypothesis, $i_{S,k-1} \leq a_{S,k-1}$. That is to say,

$$i_{S,k} = i_{S,k-1} \leq a_{S,k-1} = \max\{a_{S,k-1}, b_{S,k-1}\} = j_{S,k}.$$

C. $i = a$, $j > b$. The proof is similar to Subcase B.

D. $i = b$, $j > b$. The proof is similar to Subcase A.

Therefore we have proved Lemma 2. If a tranposition network can sort $S_d = \langle n, n-1, \cdots, 1\rangle$, i.e. for any two wires $i < j$, we have $i_{S_d,m} < j_{S_d,m}$, where m is the total number of comparators. According to the lemma, $i_{S,m} \leq j_{S,m}$ holds true for arbitrary sequence S, which means that this network can sort any sequence, and therefore is a sorting network. The other way, a sorting network can absolutely sort the sequence S_d. The proof is now complete. □

(b) (Bonus) Given any $n \in \mathbb{N}$, write a program using Tkinter in Python to draw a figure similar to Fig. 2 with n input wires.

Note: Actually there are at least 3 different types of transposition sorting network. The first one(bubbling down) is like Figure 2, having a shape of upside-down pyramid. The second one(bubbling up) is like a normal pyramid, whose structure is identical to the previous one. The last one is what we know as the Odd-Even Transposition Sorting Network[2]. For each of their simplest form, I write a program to draw it, and $n = 8$ output figure is shown in Figure 3.

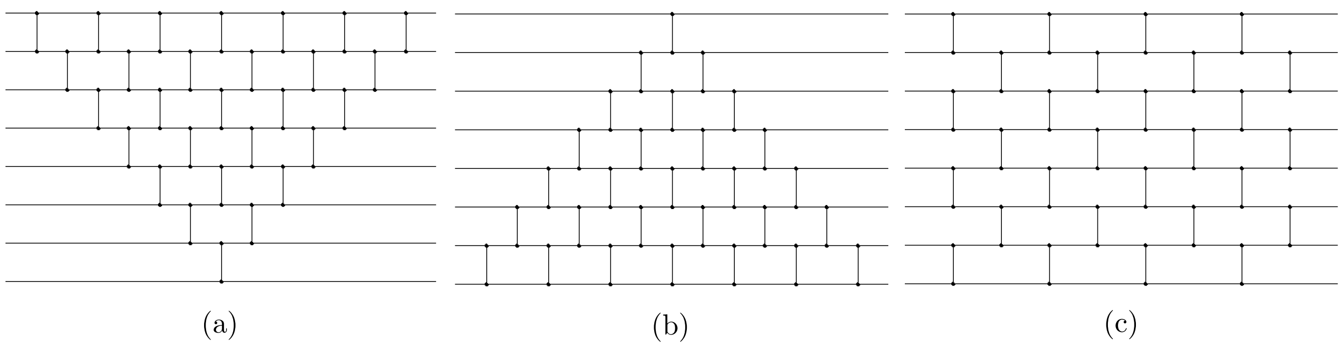

Figure 3: Three kinds of tranposition sorting networks drawn by Tkinter($n = 8$).

References

[1] “Algorithm complexity and the master theorem.” [Online]. Available: https://www.gocalf.com/blog/algorithm-complexity-and-master-theorem.html

[2] “Odd-even transposition sort.” [Online]. Available: http://www.inf.fh-flensburg.de/lang/algorithmen/sortieren/networks/oetsen.htm