Lab02-Divide and Conquer

CS214-Algorithm and Complexity, Xiaofeng Gao, Spring 2021.

∗ If there is any problem, please contact TA Haolin Zhou.
∗ Name: Wendi Chen Student ID: 519021910071 Email: chenwendi-andy@sjtu.edu.cn

1. *Recurrence examples.* Give asymptotic upper and lower bounds for $T(n)$ in each of the following recurrences. Assume that $T(n)$ is constant for sufficiently small n. Make your bounds as tight as possible.

   (a) $T(n) = 4T(n/3) + n \log n$
   (b) $T(n) = 4T(n/2) + n^2\sqrt{n}$
   (c) $T(n) = T(n-1) + n$
   (d) $T(n) = 2T(\lfloor\sqrt{n}\rfloor) + \log n$

   Solution.

   (a) We try to find the upper and lower bounds by using the **master theorem** according to *Introduction to Algorithm*.
   In this case, we have $a = 4$, $b = 3$, $f(n) = n \log n$, and $n^{\log_b a} = n^{\log_3 4} = \Omega(n^{1.2})$. Since $f(n) = O(n^{\log_b a - \epsilon})$, where $\epsilon = 0.1$, we can apply case 1 of the master theorem and conclude that the solution is $T(n) = \Theta(n^{\log_3 4})$. Thus, we find the asymptotic upper and lower bounds for $T(n)$ that $T(n) = \Omega(n^{\log_3 4}) = O(n^{\log_3 4})$.

   (b) In this case, we have $a = 2$, $b = 4$, $f(n) = n^2\sqrt{n}$, and $n^{\log_b a} = n^{\log_4 2} = n^{0.5}$. Since $f(n) = \Omega(n^{0.5+\epsilon})$, where $\epsilon = 2$, case 3 applies if we can show that the regularity condition holds for $f(n)$. For sufficiently large n, we have $af(\frac{n}{b}) = 2(\frac{n}{4})^{2.5} = \frac{1}{16}n^{2.5} = cf(n)$ for $c = \frac{1}{16}$. Consequently, by case 3, the solution to the recurrence if $T(n) = \Theta(n^2\sqrt{n})$, which means $T(n) = \Omega(n^2\sqrt{n}) = O(n^2\sqrt{n})$.

   (c) Without loss of generality, we assume that $T(n) = 1$. Then we have $T(n) = T(n-1)+n = T(n-2) + n + (n-1) = n + (n-1) + \cdots + 1 = \frac{n(n+1)}{2} = \Theta(n^2)$. Thus, we have $T(n) = \Omega(n^2) = O(n^2)$.

   (d) We can do some algebraic manipulation. Without loss of generality, we can assume $\sqrt{n}$ to be integers. Let $m = \log n$, then we have

   $$T(2^m) = 2T(2^{m/2}) + m$$

   Then set $S(m) = T(2^m)$, we get

   $$S(m) = 2S(m/2) + m$$

   Here, we have $a = 2$, $b = 2$, $f(n) = n$, and $f(m) = \Theta(m^{\log_b a}) = \Theta(m^{\log_2 2}) = \Theta(m)$. By case 2, we get S(m) $= \Theta(m \log m)$. Then $T(n) = \Theta(\log n \log\log n)$. Thus $T(n) = \Omega(\log n \log\log n) = O(\log n \log\log n)$.

   □

2. *Divide-and-conquer.* Given an integer array $A[1..n]$ and two integers $lower \leq upper$, design an algorithm using **divide-and-conquer** method to count the number of ranges (i, j) $(1 \leq i \leq j \leq n)$ satisfying

   $$lower \leq \sum_{k=i}^{j} A[k] \leq upper.$$

Example:

Given $A = [1, -1, 2]$, $lower = 1$, $upper = 2$, return 4.

The resulting four ranges are $(1, 1)$, $(3, 3)$, $(2, 3)$ and $(1, 3)$.

(a) Complete the implementation in the provided C/C++ source code (The source code *Code-Range.cpp* is attached on the course webpage).

(b) Write a recurrence for the running time of the algorithm and solve it by recurrence tree (You can modify the figure sources *Fig-RecurrenceTree.vsdx* or *Fig-RecurrenceTree.pptx* to illustrate your derivation).

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

Solution.

(a) Please refer to *Code-Range.cpp*.

(b) According to the code, we implement binary search to find m and n, whose time complexity is $O(\log n)$. In each recursion, we execute *merge_count* for $\lfloor \frac{n}{2} \rfloor$ elements and $\lceil \frac{n}{2} \rceil$ elements. And then execute *binary_search* for $\lceil \frac{n}{2} \rceil$ elements $\lfloor \frac{n}{2} \rfloor$ times. At last, we sort n elements, whose time complexity is $O(n \log n)$. Assuming that when there are n elements, the time complexity is $T(n)$, then the recurrence is

$$T(n) = T(\lfloor \frac{n}{2} \rfloor) + T(\lceil \frac{n}{2} \rceil) + 2\lfloor \frac{n}{2} \rfloor O(\log\lceil \frac{n}{2} \rceil) + O(n \log n) \tag{1}$$

For convenience, we assume n is power of 2 and $k = logn + 1$. Then by the definition of O notation, we have

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

Thus, we get the recurrence tree.

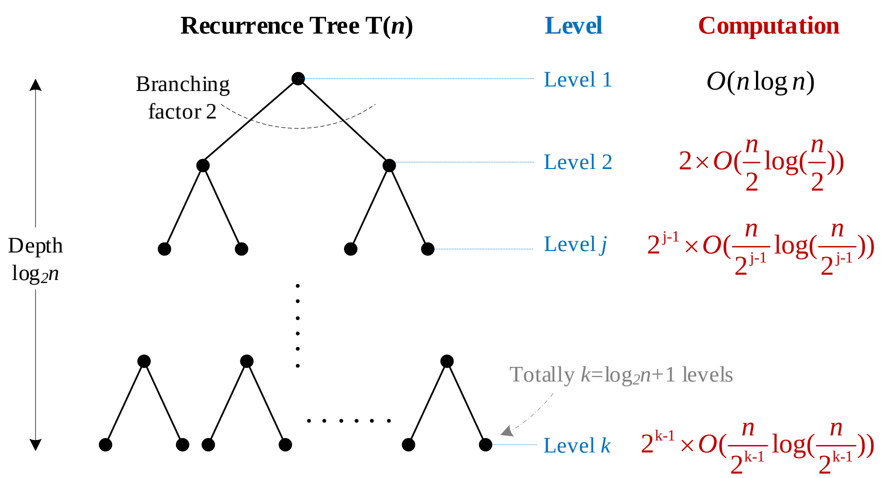


Figure 1: The Recurrence Tree of the Algorithm in *Code-Range.cpp*.

According to the recurrence tree above, the total work done can be calculated by

$$
\begin{aligned}
\sum_{j=1}^{\log n+1}(2^{j-1}\times O(\frac{n}{2^{j-1}}\log(\frac{n}{2^{j-1}}))) &= \sum_{j=1}^{\log n+1} O(n(\log n-(j-1))) \\
&= O(n\log n(\log n+1)-n\frac{(\log n+1)(\log n+2)}{2}) \\
&= O(n(\log n)^2)
\end{aligned}
\tag{3}
$$

(c) Unfortunately, we can't solve the recurrence by the typical Master Theorem. However, we can generalize the theorem to solve this problem.
According to the Master Theorem, we have $a=2$, $b=2$, $f(n)=n\log n$, and $n^{log_b a}=n^{log_2 2}=n$. Obviously, $n\log n=\Omega(n)$, but for any $\epsilon>0$, $f(n)=O(n^{1+\epsilon})$. So, for this recurrence, it falls into the gap between case 2 and case 3 of the Master Theorem.
In fact, we can generalize the Master Theorem and have the conclusion that if $f(n)=\Theta(n^{\log_b a}(\log n)^k)$, then $T(n)=\Theta(n^{\log_b a}(\log n)^{k+1})$. This can be proved by recurrence tree. By this, we can solve the recurrence and get

$$
T(n)=\Theta(n(\log n)^2) \tag{4}
$$

□

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

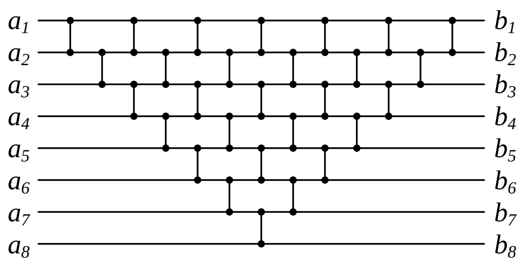


Figure 2: A Transposition Network Example

(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.*)

(b) (Optional Sub-question with 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.

Solution.

(a) **'Only if'** is easy to prove, because when a transposition network is a sorting network, it can definitely sort the sequence $\langle n, n-1, \cdots, 1\rangle$.

Then we'll prove **'if'**, which means if a transposition network sorts the sequence $\langle n, n-1, \cdots, 1\rangle$, it is a sorting network. At the very beginning, we're wondering what information are provided by a totally revered sequence. Since a totally revered sequence has the greatest number of **reversed pair**, a natural idea is to consider the relation between the number of reversed pairs of it and that of a ordinary sequence. It's kind of complex, so we need to introduce some symbols and explanations.

i. Although a sorting network is a parallel sorting algorithm. We can also view it serially. In terms of the output depth of the comparators, we can denote them by $C_1, \ldots, C_m$.
ii. We define $f(A, i, k)$ as the output of the i-th wire after the k-th comparator when the input sequence is A. $k = 0$ implies the input elements.
iii. We define sequence $A = \langle n, n-1, \cdots, 1\rangle$.

Next, we'll prove if sometimes two elements on two wires are relatively orderly when the input is A, then the two elements on the positions are also relatively orderly when the input is any sequence of $1, 2, \ldots, n$. That is, when $i < j$

$$P(k) : f(A, i, k) < f(A, j, k) \Rightarrow f(B, i, k) < f(B, j, k) \tag{5}$$

Basis. When k = 0, there is no element pair satisfying $i < j$ and $f(A, i, k) < f(A, j, k)$. Thus, $P(0)$ is true.
Induction. If $P(k)$ is true, we are trying to prove $P(k+1)$ is true. Assume the input wires of C_k are the i-th wire and $i+1$-th wire. We can prove this by cases.

i. $p = i$, $q = i+1$ and $f(A, p, k+1) < f(A, q, k+1)$. By the definition of a comparator, we have $f(B, p, k+1) < f(B, q, k+1)$.
ii. $p < i$, $q = i$ and $f(A, p, k+1) < f(A, q, k+1)$. In this case, we have $f(B, p, k+1) = f(B, p, k)$. Beside, we have

$$\begin{aligned} f(A, p, k) &= f(A, p, k+1) \\ &< f(A, q, k+1) \\ &\le min(f(A, i, k), f(A, i+1, k)) \end{aligned} \tag{6}$$

Thus, we get

$$\begin{aligned} f(B, p, k+1) &= f(B, p, k) \\ &< min(f(B, i, k), f(B, i+1, k)) \\ &= f(B, i, k+1) \\ &= f(B, q, k+1) \end{aligned} \tag{7}$$

iii. $p < i$, $q = i+1$ and $f(A, p, k+1) < f(A, q, k+1)$. In this case, we also have $f(B, p, k+1) = f(B, p, k)$. Beside, we have

$$\begin{aligned} f(A, p, k) &= f(A, p, k+1) \\ &< f(A, q, k+1) \\ &\le min(f(A, i, k), f(A, i+1, k)) \\ &\le max(f(A, i, k), f(A, i+1, k)) \end{aligned} \tag{8}$$

If $max(f(A, i, k), f(A, i+1, k)) = f(A, i+1, k)$, we can derive $f(B, p, k) < f(B, i+1, k)$ and $f(B, i, k) < f(B, i+1, k)$, then we get

$$\begin{aligned} f(B, p, k+1) &= f(B, p, k) \\ &< f(B, i+1, k) \\ &= max(f(B, i, k), f(B, i+1, k)) \\ &= f(B, i+1, k+1) \\ &= f(B, q, k+1) \end{aligned} \tag{9}$$

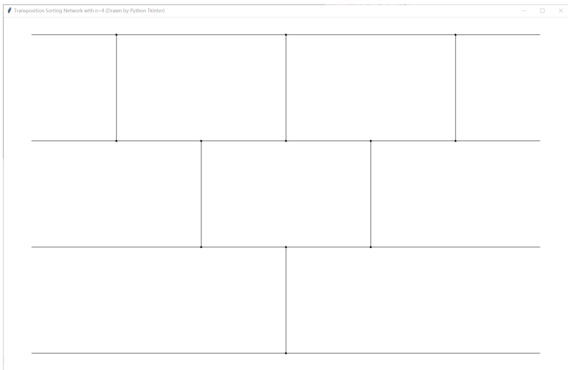
iv. $p \neq i$, $p \neq i+1$, $q \neq i$, $q \neq i+1$ and $f(A,p,k+1) < f(A,q,k+1)$. Then we get $f(A,p,k) = f(A,p,k{+}1) < f(A,q,k{+}1) = f(A,q,k)$, which implies $f(B,p,k{+}1) = f(B,p,k) < f(B,q,k) = f(B,q,k+1)$.

v. $p = i+1$, $q > i+1$ and $f(A,p,k+1) < f(A,q,k+1)$. We can prove this like ii.

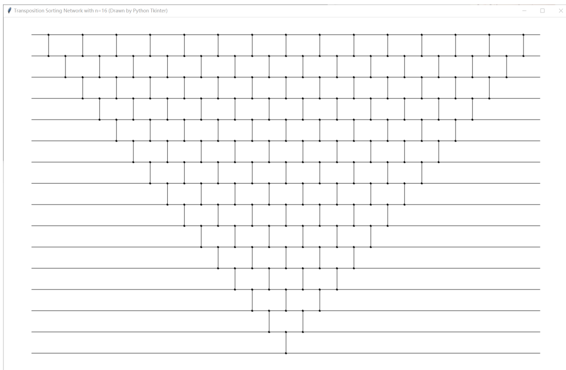
vi. $p = i$, $q > i+1$ and $f(A,p,k+1) < f(A,q,k+1)$. We can prove this like iii.

The cases above implies $P(K{+}1)$ is true. By mathematical induction, $P(m)$ is true. Since the network sorts A, $f(A,i,m) < f(A,j,m)$ are true for all $i < j$. Thus, $f(B,i,m) < f(B,j,m)$ are true for all $i < j$, which implies the network sorts B. Then this network is a sorting network.

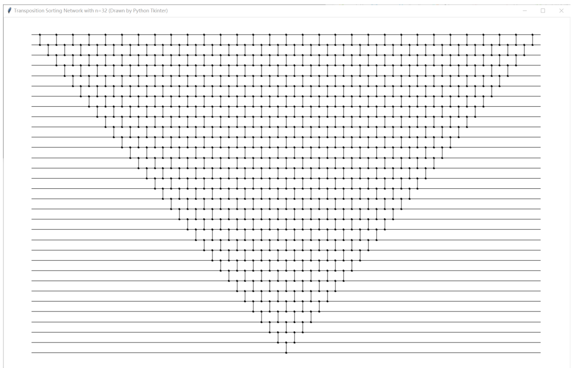
(b) Please refer to *Code-TranspositionSortingNetwork.py*.



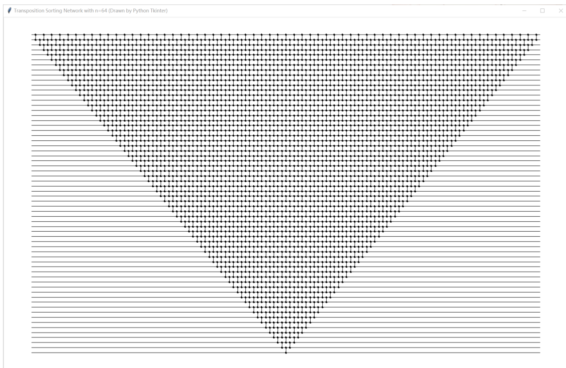
(a) $n = 4$



(b) $n = 16$



(c) $n = 32$



(d) $n = 64$

Figure 3: Transposition Networks Generated with Tkinter

$\square$