# Lab02-Divide and Conquer

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

$*$ If there is any problem, please contact TA Haolin Zhou.

$*$ Name:Qi Liu  Student ID:519021910529  Email: purewhite@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) $\forall \epsilon > 0$, $n = O(n \log n), n \log n = O(n^{1+\epsilon})$. We have

$$T_l(n) = 4T_l(n/3) + n^{1+\epsilon}$$
$$T_u(n) = 4T_u(n/3) + n$$

And for sufficiently small $n$, $T_l(n) = T(n) = T_u(n)$. Therefore

$$T_l(n) \le T(n) \le T_u(n)$$

Since according to the Master Theorem, for sufficiently small $\epsilon$ ($\epsilon < \log_3 4 - 1$),

$$T_l(n) = \Theta(n^{\log_3 4})$$
$$T_u(n) = \Theta(n^{\log_3 4})$$

We have $T(n) = \Theta(n^{\log_3 4})$

(b) According to the Master Theorem, we have

$$T(n) = \Theta(n^{\frac{5}{2}})$$

(c)

$$\begin{aligned}
T(n) &= T(n-1) + n \\
&= T(n-2) + n - 1 + n \\
&= \cdots \\
&= T(1) + 2 + 3 + .. + n \\
&= T(1) + \frac{(n+2)(n-1)}{2} \\
&= \Theta(n^2)
\end{aligned}$$

(d) Let $T_l(n) = 3T_l(n^{\frac{1}{3}}) + \log n$, $T_u(n) = 2T_u(\sqrt{n}) + \log n$, and for sufficiently small $n$, $T_l(n) = T(n) = T_u(n)$. Thus we have $T_l(n) = O(T(n))$, $T(n) = O(T_u(n))$

$$\begin{aligned} T_u(n) &= 2T_u(\sqrt{n}) + \log n \\ &= 2^2 T_u(n^{\frac{1}{2^2}}) + 2\log n^{\frac{1}{2}} + \log n \\ &= \cdots \\ &= 2^k T_u(n^{\frac{1}{2^k}}) + \sum_{i=0}^{k} 2^i \log n^{\frac{1}{2^i}} \\ &= 2^k T_u(n^{\frac{1}{2^k}}) + \sum_{i=0}^{k} \log n \\ &= 2^k T_u(n^{\frac{1}{2^k}}) + k\log n \end{aligned}$$

When $n^{\frac{1}{2^k}}$ is sufficiently small, $k = \Theta(\log(\log n))$, thus

$$\begin{aligned} T_u(n) &= 2^{\Theta(\log(\log n)} \cdot \Theta(1) + \Theta(\log(\log n)) \cdot \log n \\ &= \Theta(\log(\log n) \cdot \log n) \end{aligned}$$

As for $T_l(n)$,

$$\begin{aligned} T_l(n) &= 3T_u(n^{\frac{1}{3}}) + \log n \\ &= 3^3 T_l(n^{\frac{1}{3^3}}) + 3\log n^{\frac{1}{3}} + \log n \\ &= \cdots \\ &= 3^k T_l(n^{\frac{1}{3^k}}) + \sum_{i=0}^{k} 3^i \log n^{\frac{1}{3^i}} \\ &= 3^k T_l(n^{\frac{1}{3^k}}) + \sum_{i=0}^{k} \log n \\ &= 3^k T_l(n^{\frac{1}{3^k}}) + k\log n \end{aligned}$$

When $a_k$ is sufficiently small, $k = \Omega(\log(\log n))$, thus

$$\begin{aligned} T_u(n) &= 3^{\Theta(\log(\log n)} \cdot \Theta(1) + \Theta(\log(\log n)) \cdot \log n \\ &= \Theta(\log(\log n) \cdot \log n) \end{aligned}$$

Since $T_l(n) = O(T(n))$, $T(n) = O(T_u(n))$, we have $T(n) = \Theta(\log(\log n) \cdot \log n)$.

□

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.

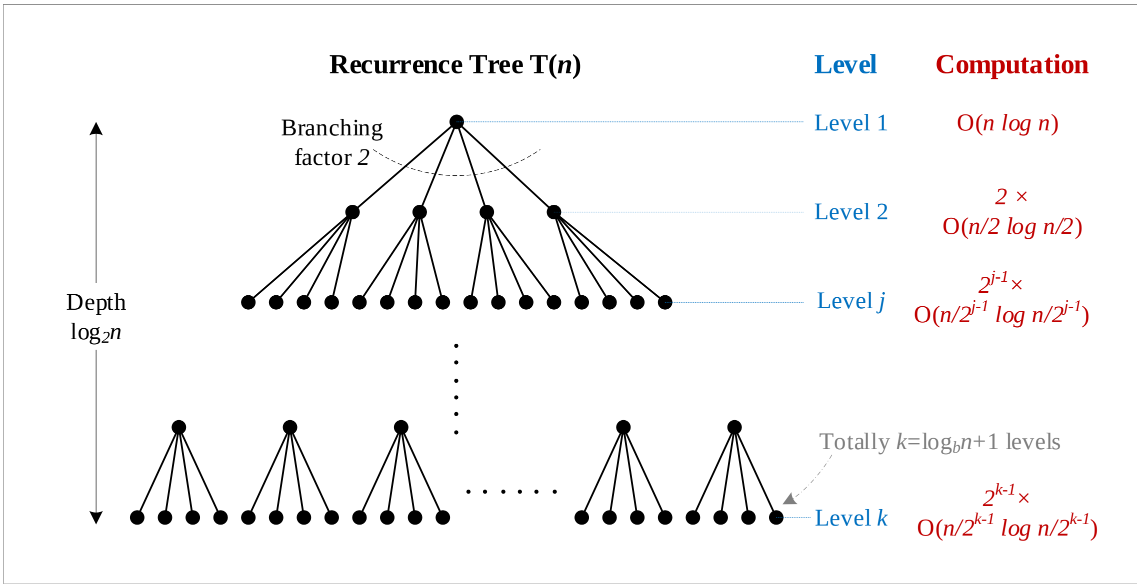


Figure 1: Design

**Solution.** (a) The source file $Code - Range.cpp$ is in the archive file $Lab02 - QiLiu.zip$.
(b) On the $k$-th layer, there will be $2^{k-1}$ $merge\_count(\frac{n}{2^{k-1}})$ instances, each $merge\_count()$ will call $sort()$ and 2 recurrent instances $binary\_search\_for\_m()$ and $binary\_search\_for\_n()$, consuming a period of time $O(\frac{n}{2^{k-1}}\log\frac{n}{2^{k-1}}) + 2O(\log\frac{n}{2^k})$.
Let's denote the time complexity of $merge\_count(n)$ as $T(n)$, therefore

$$
\begin{aligned}
T(n) &= 2T(n/2) + O(n\log n) + 2O(\log n/2)\\
&= 2T(n/2) + O(n\log n)\\
&= 2^kT(\frac{n}{2^k}) + \sum_{i=0}^{k-1} 2^iO(\frac{n}{2^i}\log\frac{n}{2^i})\\
&= 2^kT(\frac{n}{2^k}) + \sum_{i=0}^{k-1} O(n\log n - nti\log 2)\\
&= 2^{\log n}O(1) + \sum_{i=0}^{\log n-1} O(n\log n - ni\log 2)\\
&= O(n\log^2 n)
\end{aligned}
$$

(c)

$$\begin{aligned}\forall \epsilon > 0, \lim_{n\to+\infty} \frac{n\log n}{n^{(1+\epsilon)}} \\ = \lim_{n\to+\infty} \frac{\log n}{n^{\epsilon}} \\ = 0,\end{aligned}$$

thus,

$$n\log n = O(n^{(1+\epsilon)})$$

and we have

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

therefore,

$$\nexists d \in \mathbb{R}^+, n\log n = \Theta(n^d)$$

So this recurrence can't be **precisely** solved by the Master Theorem.

□

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

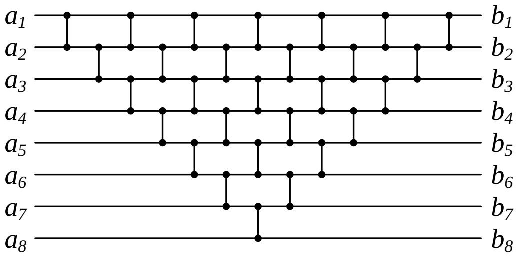


Figure 2: A Transposition Network Example

(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.** For a sequence $\mathbf{a} = \langle a_1, a_2, \cdots, a_n\rangle$, we use $\tau(\mathbf{a})$ to denote its number of inversions. And in the transposition network, after $k$ comparisons, we use $Inversion_k$ to denote the array who is originally $\langle n, n-1, \cdots, 1\rangle$ and $Input_k$ to denote the array with an arbitrary input. For a certain array $A$, we use $A[i]$ to denote its $i$-th element.

**Lemma 1.** *For a certain transposition network with $k_{total}$ comparators which sorts the sequence $\langle n, n-1, \cdots, 1\rangle$, after any $k$ comparisons and any given input sequence, we have $\tau(Input_k) \le \tau(Inversion_k)$.*

To prove Lemma 1, we have to use another lemma:

**Lemma 2.** *For a certain transposition network with $k_{total}$ comparators which sorts the sequence $\langle n, n-1, \cdots, 1\rangle, \forall k \in \{1, 2, \cdots, k_{total}\}, \forall i < j, \{i, j\} \subseteq \{1, 2, \cdots, n\}$: $Inversion_k[i] < Inversion_k[j] \Rightarrow Input_k[i] < Input_k[j]$.*

Now let's prove Lemma 2.

i. For $k = 0$, the lemma is obviously true because all element pairs in $Inversion_0$ are inversions.

ii. Suppose the lemma is true for all $k(k = 0, 1, \cdots, k_0)$, then as for the $(k_0 + 1)$-th comparison, for any given $i, j(i < j)$,

A. If the $(k_0 + 1)$-th comparison is irrelevant to $i$ and $j$, the comparison will not affect the $i$-th element and the $j$-th element in $Inversion$ and $Input$. Since the lemma is true for $k_0$, it will still be true for this $i, j$ when $k = k_0 + 1$.

B. If the $(k_0+1)$-th comparison is to compare the $i$-th element and the $j$-th element, after the comparison, the $i$-th element and the $j$-th element are well-ordered for array $Inversion$ and array $Input$. Therefore the lemma is still true for $k = k_0+1$.

C. Without loss of generality, let's consider if the $(k_0+1)$-th comparison is between the $i$-th element and the $(i+1)$-th element (other cases can be similarly proved). In this case, $j > i + 1$ ($j = i + 1$ is discussed in case B).

- If $Inversion_{k_0}[i] < Inversion_{k_0}[i+1]$, the $i$-th element and the $i+1$-th element of the array $Inversion$ will not be swapped after comparison. And after comparison, $Input_{k_0+1}[i] \leq Input_{k_0}[i]$, thus the lemma is still true for this $i, j$ when $k = k_0 + 1$, no matter $Inversion_{k_0}[i] < Inversion_{k_0}[j]$ or not.
- If $Inversion_{k_0}[i] > Inversion_{k_0}[i + 1]$,
  - If $Inversion_{k_0}[j] > Inversion_{k_0}[i] > Inversion_{k_0}[i + 1]$, according to the lemma, we have $Input_{k_0}[j] > Input_{k_0}[i]$ and $Input_{k_0}[j] > Input_{k_0}[i + 1]$. Thus after comparison, this relation will remain. Hence in this case the lemma is true for $i, j$ when $k = k_0 + 1$.
  - If $Inversion_{k_0}[i] > Inversion_{k_0}[i + 1] > Inversion_{k_0}[j]$, after swap, the $Inversion_{k_0+1}[i] = Inversion_{k_0}[i+1] > Inversion_{k_0}[j] = Inversion_{k_0+1}[j]$. The condition of the lemma on $i, j$ remains unsatisfied. Thus the lemma is still true for this case.
  - If $Inversion_{k_0}[i] > Inversion_{k_0}[j] > Inversion_{k_0}[i + 1]$, we have

    $$\begin{aligned} Inversion_{k_0+1}[j] = Inversion_{k_0}[j] > \\ Inversion_{k_0}[i + 1] = Inversion_{k_0+1}[i] \end{aligned}$$

    Since $Inversion_{k_0}[j] > Inversion_{k_0}[i+1]$, according to the lemma, we have $Input_{k_0}[i + 1] < Input_{k_0}[j]$. After this comparison, we have

    $$\begin{aligned} Input_{k_0+1}[i] \leq Input_{k_0}[i + 1] < \\ Input_{k_0}[j] = Input_{k_0+1}[j] \end{aligned}$$

    and

    $$Inversion_{k_0+1}[i] < Inversion_{k_0+1}[j]$$

    Thus this lemma is true for $i, j$ when $k = k_0 + 1$.

Thus the lemma is true for this given $i, j$ when $k = k_0 + 1$.Due to the arbitrariness of $i, j$, the lemma is true for $k = k_0 + 1$.

Thus Lemma 2 is proven. According to Lemma 2's inverse proposition, after the comparison $k$ and for any $i, j$ pair that $i < j$, if $Input_k[i] > Input_k[j]$, we have $Inversion_k[i] > Inversion_k[j]$ (obviously there is no "="), thus there is a injective function from inversions in $Input_k$ to inversions in $Inversion_k$. Therefore we have $\tau(Input_k) \leq \tau(Inversion_k)$, thus lemma 1 is proven.
After the $k_{total}$-th comparison, $\tau(Inversion_k) = 0$, thus $0 \leq \tau(Input_k) \leq \tau(Inversion_k) = 0$, $\tau(Input_k) = 0$, the sequence $Input_{k_{total}}$ is well sorted. $\square$

(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.** The source file $TranspositionNetwork.py$ is in the archive file $Lab02 - QiLiu.zip$. $\square$

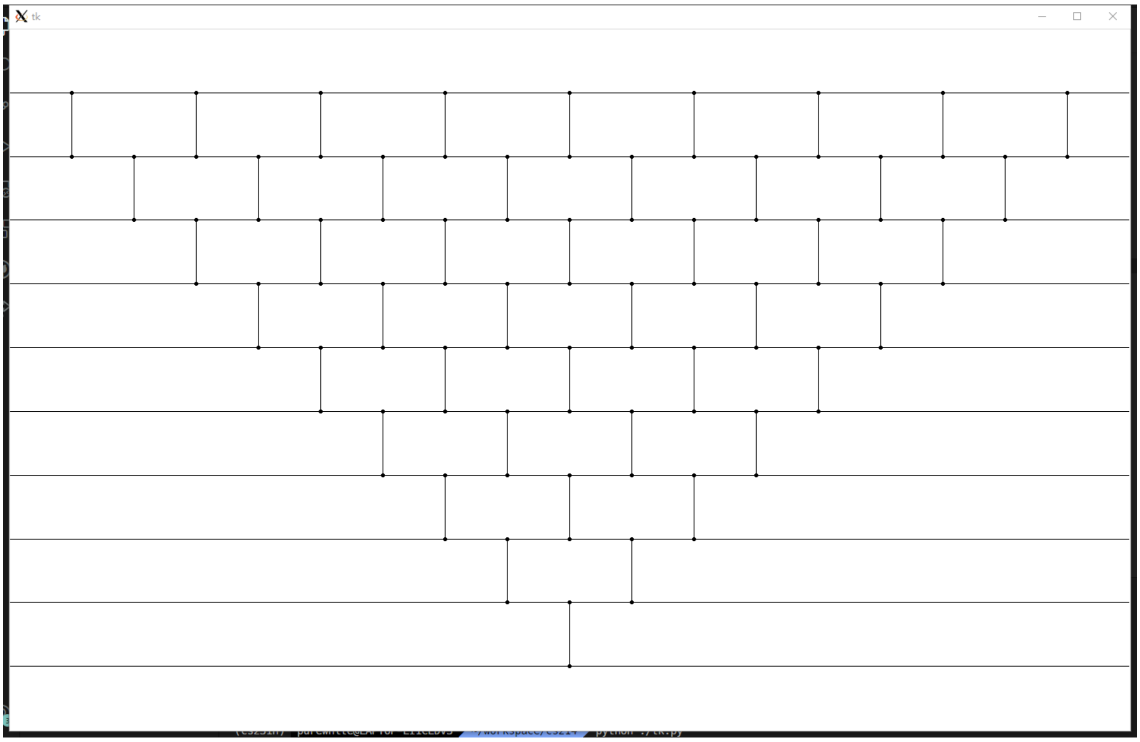

Figure 3: Transposition Network with 10 inputs