Lab02-Divide and Conquer

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

∗ If there is any problem, please contact TA Haolin Zhou.
∗ Name: Zilong Li Student ID: 518070910095 Email: logcreative-lzl@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)

$$\begin{aligned}
T(n) = 4T(n/3) + n\log n &= \sum_{i=0}^{\log_3 n} 4^i \frac{n}{3^i} \log \frac{n}{3^i} \\
&= \sum_{j=0}^{\log_3 n} \left(\frac{4}{3}\right)^{\log_3 n} \left(\frac{3}{4}\right)^j n \log \frac{n}{3^{\log_3 n - j}} \\
&= \left(\frac{4}{3}\right)^{\log_3 n} n \log 3 \sum_{j=0}^{\log_3 n} j \left(\frac{3}{4}\right)^j \\
&= \left(\frac{4}{3}\right)^{\log_3 n} n \log 3 \cdot 4 \left[\sum_{j=1}^{\log_3 n} \left(\frac{3}{4}\right)^j - \log_3 n \cdot \left(\frac{3}{4}\right)^{(\log_3 n)+1}\right] \\
&= 12 \left(\frac{4}{3}\right)^{\log_3 n} n \log 3 \cdot \left[1 - \left(\frac{3}{4}\right)^{\log_3 n} \left(1 + \frac{1}{4}\log_3 n\right)\right] \\
&= 12 \log 3 \cdot 4^{\log_3 n} - 12n \log 3 - 3n \log n \text{ (lower bound)} \\
&\le 12 \log 3 \cdot n^{\log_3 4} = O(n^{\log_3 4}) \text{ (upper bound)}
\end{aligned}$$

   (b)

$$T(n) = 4T(n/2) + n^2\sqrt{n} = 4T\left(\frac{n}{2}\right) + n^{5/2}$$

   According to the Master Theorem, $d = \frac{5}{2} > \log_b a = 2$, Then,

$$T(n) = O(n^{5/2}) \text{ (upper bound)}$$

   In fact, by the same deduction in the Master Theorem,

$$T(n) = n^{5/2} \frac{1}{2^{5/2} - 4} \approx 0.61 n^{5/2} = \Theta(n^{5/2}) \text{ (upper and lower bounds)}$$

   (c)

$$\begin{aligned}
T(n) = \sum_{i=1}^{n} i = \frac{n(n+1)}{2} &\qquad \text{(lower bound)} \\
= O(n^2) &\qquad \text{(upper bound)}
\end{aligned}$$

(d)

$$T(n) = 2T(\lfloor\sqrt{n}\rfloor) + \log n = 2T(1) + \sum_{i=0}^{k} 2^k \log n^{\frac{1}{2^k}} = 2T(1) + \sum_{i=0}^{k} \log n$$

To get the layer number k, the recurrence should stop when $\lfloor\sqrt{n}\rfloor = 1$ for no further drop on the input. As a result,

$$\begin{aligned} \lfloor n^{1/2^k} \rfloor &< 4 \\ n &< 2^{2^{k+1}} \\ k &> \log_2(\log_2 n - 1) \end{aligned}$$

$$\begin{aligned} T(n) &= 2T(1) + \log_2(\log_2 n - 1)\log n \\ &= \log_2(\log_2 n - 1)\log n && \text{(lower bound)} \\ &= O(\log\log\log n) && \text{(upper bound)} \end{aligned}$$

□

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

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

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) The implementation is shown as follows: `Code-Range.cpp`

```cpp
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

/*
You need to complete the implementation of
binary_search_for_m and binary_search_for_n.
*/

int binary_search_for_m(vector<long>& sum, long sum_i, int LOWER, int low, int high) {
        /*
        Find the smallest m in [low, high - 1]
        such that sum[m] - sum_i >= LOWER
        using binary search. If not found,
        return high.
```

```
        */
        if (low == high) return high;
        int mid = (low + high)/2;
        if (sum[mid] - sum_i >= LOWER) return binary_search_for_m(sum,sum_i,LOWER,low,mid);
        else return binary_search_for_m(sum,sum_i,LOWER,mid+1,high);
}

int binary_search_for_n(vector<long>& sum, long sum_i, int UPPER, int low, int high) {
        /*
        Find the smallest n in [low, high - 1]
        such that sum[n] - sum_i > UPPER
        using binary search. If not found,
        return high.
        */
        if (low == high) return high;
        int mid = (low + high)/2;
        if (sum[mid] - sum_i > UPPER) return binary_search_for_n(sum,sum_i,UPPER,low,mid);
        else return binary_search_for_n(sum,sum_i,UPPER,mid+1,high);
}

int merge_count(vector<long>& sum, int low, int high, int LOWER, int UPPER) {
        int mid = (low + high) / 2;
        if (mid == low)
                return 0;
        int count = merge_count(sum, low, mid, LOWER, UPPER)
                + merge_count(sum, mid, high, LOWER, UPPER);
        int m_low = mid, m_high = high, n_low = mid, n_high = high;
        for (int i = low; i < mid; i++) {
                int m = binary_search_for_m(sum, sum[i], LOWER, m_low, m_high);
                int n = binary_search_for_n(sum, sum[i], UPPER, n_low, n_high);
                count += n - m;
        }
        sort(sum.begin() + low, sum.begin() + high); // You may assume the time complexity here is
        ↪ O(nlog(n)).
        return count;
}

int main() {
        int N, LOWER, UPPER;
        vector<int> A;
        vector<long> sum(1, 0);         // sum[i]: sum of the first i elements in A

        cin >> N >> LOWER >> UPPER;
        for (int tmp, i = 0; i < N; i++) {
                cin >> tmp;
                A.push_back(tmp);
                sum.push_back(sum.back() + A.back());
        }

        cout << merge_count(sum, 0, N + 1, LOWER, UPPER) << endl;

        return 0;
}
```

(b) Because the modified binary search is worth the time complexity of $\Theta(\log n)$ because there is no $O(1)$ trival scenario for finding the lower bound in the first shot. The recurrence for this program is:

$$\begin{aligned} T(n) &= 2T\left(\frac{n}{2}\right) + \frac{n}{2} \cdot 2\log\frac{n}{2} + O(n\log n) \\ &= 2T\left(\frac{n}{2}\right) + O(n\log n) \end{aligned}$$

And the recurrence tree is shown below:

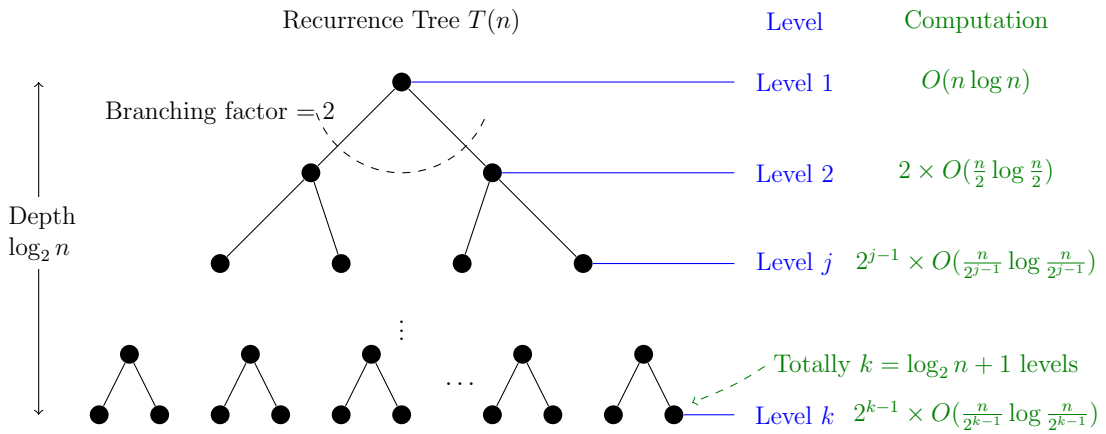


Figure 1: The recurrence tree

So, the time complexity can be calculated by the summation:

$$\begin{aligned} T(n) &= \sum_{j=0}^{\log_2 n} 2^j \times O\left(\frac{n}{2^j} \log \frac{n}{2^j}\right) \\ &= \sum_{j=0}^{\log_2 n} O\left(n \log \frac{n}{2^j}\right) \qquad (1) \\ &= (\log_2 n + 1) O(n \log n) - O(n \log 2) \frac{\log_2 n (\log_2 n + 1)}{2} \qquad (2) \\ &= O(n \log^2 n) \end{aligned}$$

(c) **Yes, at least under this condition.** The recurrence here merely satisfies the condition of $d = \log_b a = 1$ in the Master Theorem, just plug in the formula with the logarithm followed we could get:

$$T(n) = O((n \log n) \log n) = O(n \log^2 n)$$

Though there is a logarithm in the residual, as is shown in Eq. (2), the logarithm will take apart the division into subtraction, where the subtraction part will not change the complexity of the overall algorithm (the subtrahend is always smaller than the minuend, otherwise the algorithm could have a negative amount of time complexity). As we ignore the the logarithm part, the remaining thing is similar to the Master Theorem, for example, in this condition, the exponential part outside the logarithm is crossed out in Eq. (1).

□

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

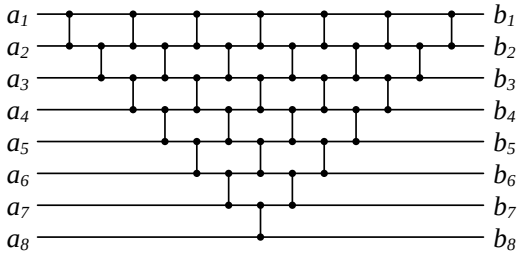


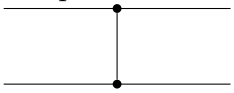
Figure 2: A Transposition Network Example

(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) It is trival that a transposition network with n inputs is a sorting network, which could sort the sequence $\langle n, n-1, \cdots, 1\rangle$. Now, it is to be proved that if a transposition network with n inputs sorts the sequence $\langle n, n-1, \cdots, 1\rangle$ into $\langle 1, 2, \cdots, n\rangle$, then it is a sorting network, i.e., a comparison network for which the output sequence $\langle b_1, b_2, \cdots, b_n\rangle$ is monotonically increasing ($b_1 \leq b_2 \leq \cdots \leq b_n$) for every input sequence $\langle a_1, a_2, \cdots, a_n\rangle$. **Show this by mathematical induction.**

Basic Step. When $n = 2$, the comparison network to convert $\langle 2, 1\rangle$ into $\langle 1, 2\rangle$ is a single comparator acrossing the two wires.



According to the definition of a comparator, if the input wires are a_i and a_j, the upper output wire always transmits $\min\{a_i, a_j\}$, and the lower output wire always transmits $\max\{a_i, a_j\}$. So it could sort any input sequence which is of size 2.

Hypothesis. For size k, if the comparsion network is capable of sorting $\langle k, k-1, \cdots, 1\rangle$ into $\langle 1, 2, \cdots, k\rangle$, then it could sort any input sequence $\langle a_1, a_2, \cdots, a_k\rangle$ into a monotonically increasing sequence $\langle b_1, b_2, \cdots, b_k\rangle$.

Induction. For size $k+1$, if the comparator network is capable of sorting $\langle k+1, k, \cdots, 1\rangle$ into $\langle 1, 2, \cdots, k+1\rangle$, then there must be a structure sorting $\langle k+1, k, \cdots, 2\rangle$ or $\langle k, k-1, \cdots, 1\rangle$ because of the final result, and another sending structure to send the remaining element into the target place.

Because the property of the transposition network, to send the remaining element (1 or $k+1$ repectively) into the other end target place (the uppermost wire or the lowest wire respectively), the sending structure will come across **all the wires** in the path. The four possiblities are shown in the Figure 3 (the sending structure here is *the best case structure*), which enumerates all the scenarios to be the proof for having a sub-sorting network and a sending structure.

As a matter of fact, because such comparison network presereve the output order if the input order is the same according to the Domain Conversion Lemma where a monotonically increase function could be constructed, the sorting comparison networks for $\langle k, k-1, \cdots, 1\rangle$ and $\langle k+1, k, \cdots, 2\rangle$ will have **the same property** for sorting any abitrary input sequence $\langle a_1, a_2, \cdots, a_k\rangle$, so they are all denoted as "k sorting network" in the figure.

For the network in Figure 3(a), if an abitrary input sequence $\langle a_1, \cdots, a_k, x\rangle$ is put into the network, by the property of the "k sorting network", the output for this sub-sorting structure is $\langle b_1, b_2, \cdots, b_k\rangle$ where $b_1 \leq b_2 \leq b_k$. And then x could be placed into **the appropriate position** and the sequence is perfectly ordered. The procedure is shown in Figure 4. And it is similar to the network in Figure 3(b), 3(c), 3(d).

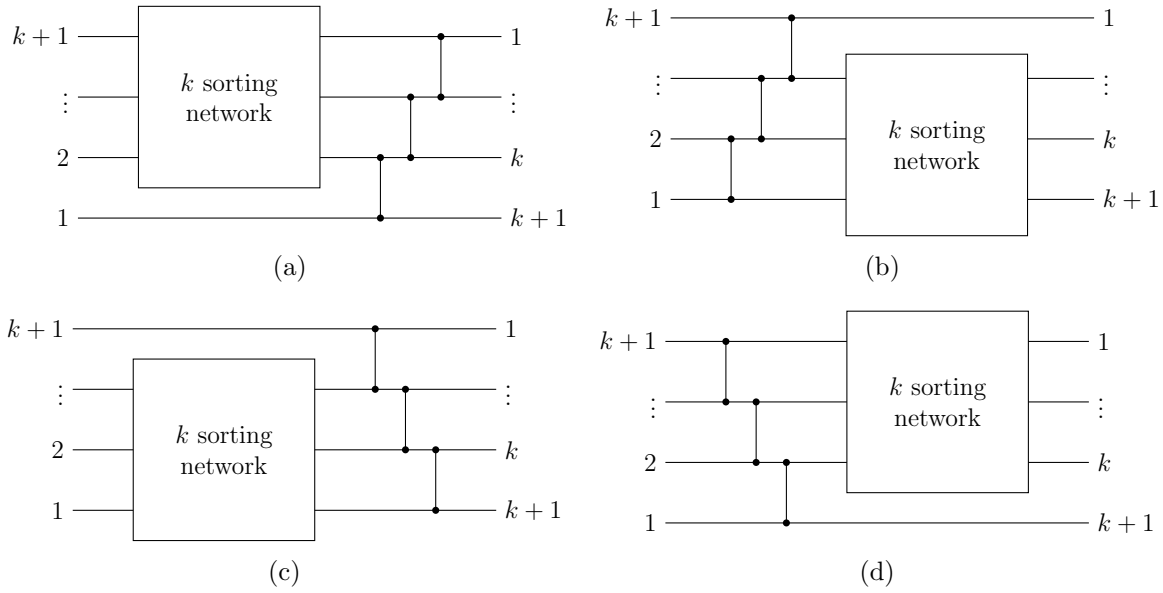


Figure 3: Four scenarios for size $k+1$

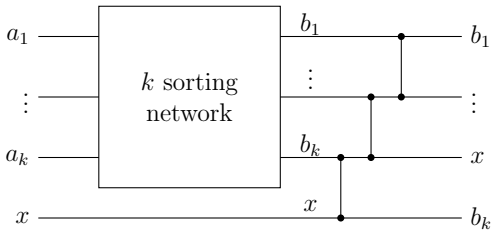


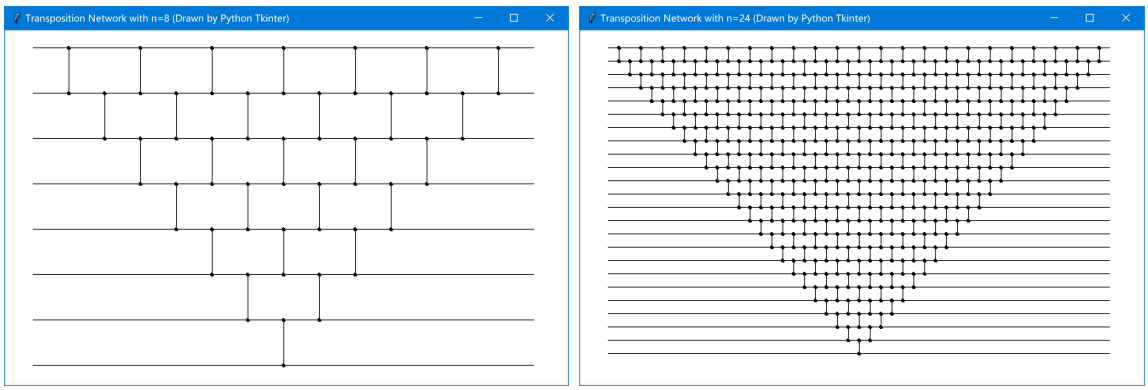
Figure 4: Input an abitrary sequence into the network

So for size $k+1$, the proposition also holds.

Conclusion. By the principle of mathematical induction, if a transposition network with n inputs sorts the sequence $\langle n, n-1, \cdots, 1\rangle$, then it is a sorting network.

As a result, a transposition network with n inputs is a sorting network $\Leftrightarrow$ it sorts the sequence $\langle n, n-1, \cdots, 1\rangle$.

(b) `Code-Transposition.py` uses the construction method of Figure 3(d) to create the $n = 8$ and $n = 24$ transposition networks, which is shown in Figure 5.



(a) $n = 8$ (b) $n = 24$

Figure 5: Tkinter graphics of the transposition networks

Code-Transposition.py

```python
import sys
from tkinter import *

class MyCanvas(Canvas):
    def __init__(self, master, hLineWidth=1, vLineWidth=1, radius=2, **kwargs):
        Canvas.__init__(self, master, kwargs)
        self.hLineWidth = hLineWidth
        self.vLineWidth = vLineWidth
        self.radius = radius

    def create_segment_h(self, x, y, l):
        self.create_line(x, y, x + l, y, width=self.hLineWidth)
        self.create_oval(x - self.radius, y - self.radius, x + self.radius, y + self.radius, fill='black')
        self.create_oval(x + l - self.radius, y - self.radius, x + l - self.radius, y + self.radius,
        ↪  fill='black')

    def create_segment_v(self, x, y, l):
        self.create_line(x, y, x, y + l, width=self.vLineWidth)
        self.create_oval(x - self.radius, y - self.radius, x + self.radius, y + self.radius, fill='black')
        self.create_oval(x - self.radius, y + l - self.radius, x + self.radius, y + l + self.radius,
        ↪  fill='black')

    def create_line_h(self, x, y, l):
        self.create_line(x, y, x + l, y, width=self.hLineWidth)

    def create_line_v(self, x, y, l):
        self.create_line(x, y, x, y + l, width=self.vLineWidth)

class Sorter:
    def __init__(self, size):
        self.size = size
        if size > 1:
            self.subSorter = Sorter(size - 1)

    def hNum(self):
        return 2 * (self.size - 1)

    def draw_all_hlines(self, cvs, x, y, hScale, vScale):
        '''Create all the horizontal lines in the first place.'''
        leftend = x
        length = hScale * (self.hNum())
        for i in range(1, self.size+1):
            cvs.create_line_h(leftend, y, length)
            y += vScale

    def draw(self, cvs, x, y, hScale, vScale):
        '''Draw the comparators.'''
        if(self.size > 1):
            self.subSorter.draw(cvs,x + 2 * hScale,y,hScale,vScale)
        for i in range(1,self.size):
            cvs.create_segment_v(x,y,vScale)
            x += hScale
            y += vScale

if __name__ == '__main__':
    n = int(input('please input the number n: '))
    sortingNetwork = Sorter(n)

    winW, winH = 2400 * 0.3, 1500 * 0.3
    hMargin, vMargin = winW // 20, winH // 20
    hScale, vScale = (winW - 2 * hMargin) // sortingNetwork.hNum(), (winH - 2 * vMargin) // (n - 1)

    root = Tk()
    root.title('Transposition Network with n=%d (Drawn by Python Tkinter)' %n)
    cvs = MyCanvas(root, bg='white', width = winW, height = winH)
    sortingNetwork.draw_all_hlines(cvs, hMargin, vMargin, hScale, vScale)
    sortingNetwork.draw(cvs, hMargin + hScale, vMargin, hScale, vScale)
    cvs.pack()
    root.mainloop()
```

□