Lab02-Divide and Conquer

CS214-Algorithm and Complexity, Xiaofeng Gao, Spring 2020.

$*$ If there is any problem, please contact TA Yiming Liu.

$*$ Name:Zehao Wang Student ID:518021910976 Email: davidwang200099@sjtu.edu.cn

1. **Quicksort** is based on the Divide-and-Conquer method. Here is the two-step divide-and-conquer process for sorting a typical subarray $A[p \dots r]$:

   (a) **Divide:** Partition the array $A[p \dots r]$ into two subarrays $A[p \dots q-1]$ and $A[q+1 \dots r]$ such that each element of $A[p \dots q-1]$ is less than or equal to $A[q]$, which is, in turn, less than or equal to each element of $A[q+1 \dots r]$. Compute the index q as part of this partitioning procedure.

   (b) **Conquer:** Sort $A[p \dots q-1]$ and $A[q+1 \dots r]$ respectively by recursive calls to Quicksort.

   Write down the recurrence function $T(n)$ of QuickSort and compute its time complexity.

   Hint: At this time $T(n)$ is split into two subarrays with different sizes (usually), and you need to describe its recurrence relation by the sum of two subfunctions plus additional operations.

   Solution. **QuickSort** first requires randomly selecting a pivot and rearrange the array into two parts:one smaller than pivot and the other smaller than the pivot.This step requires $(n-1)$ comparisons and swaps.

   Then recursively sort the two parts.

   Assumes that the pivot is randomly selected.Then every element in the array is equally possible to be selected as the pivot in the first call of **QuickSort** and the probability is $\frac{1}{n}$.

   Then the time complexity of **Conquer** is $\frac{1}{n}\sum_{i=1}^{n}(T(i-1)+T(n-i)) = \frac{1}{n}\sum_{i=0}^{n-1}(T(i)+T(n-i-1)) = \frac{2}{n}\sum_{i=0}^{n-1}T(i)$.

   Let $S(n)=\sum_{i=0}^{n-1}(T(i))$.Then the recursion function is:

$$T(n) = (n-1) + \frac{2}{n}S(n-1) \tag{1}$$

   left-hand side and right-hand side both multiplied by n, we have :

$$nT(n) = n(n-1) + 2S(n-1) \tag{2}$$

   And we have

$$(n-1)T(n-1) = (n-1)(n-2) + 2S(n-2) \tag{3}$$

   Equation (2) minus Equation (3):

$$nT(n) - (n-1)T(n-1) = 2(n-1) + 2T(n-1) \tag{4}$$

   left and right both divided by $n(n+1)$:

$$\frac{T(n)}{n+1} - \frac{T(n-1)}{n} = \frac{2(n-1)}{n(n+1)} \approx \frac{2}{n} \tag{5}$$

   Therefore $\frac{T(n)}{n+1} \approx 2\sum_{i=1}^{n}\frac{1}{n}$

   Therefore $T(n) \approx (n+1)\log(n) = O(nlogn)$. □

2. **MergeCount**. Given an integer array $A[1 \dots n]$ and two integer thresholds $t_l \le t_u$, Lucien designed an algorithm using divide-and-conquer method (As shown in Alg. 1) to count the number of ranges (i, j) $(1 \le i \le j \le n)$ satisfying

$$t_l \le \sum_{k=i}^{j} A[k] \le t_u. \tag{6}$$

Before computation, he firstly constructed $S[0 \dots n+1]$, where $S[i]$ denotes the sum of the first i elements of $A[1 \dots n]$. Initially, set $S[0] = S[n+1] = 0$, $low = 0$, $high = n+1$.

Algorithm 1: MergeCount(S, t_l, t_u, low, $high$)

```
Input: $S[0, \cdots, n+1]$, $t_l$, $t_u$, $low$, $high$.
Output: $count$ = number of ranges satisfying Eqn. (6).
1 $count \leftarrow 0$; $mid \leftarrow \lfloor \frac{low+high}{2} \rfloor$;
2 if $mid = low$ then return 0 ;
3 $count \leftarrow MergeCount(S, t_l, t_u, low, mid) + MergeCount(S, t_l, t_u, mid, high)$;
4 for $i = low$ to $mid - 1$ do
5     $m \leftarrow \begin{cases} \min\{m \mid S[m] - S[i] \ge t_l, m \in [mid, high-1]\}, & \text{if exists} \\ high, & \text{if not exist} \end{cases}$;
6     $n \leftarrow \begin{cases} \min\{n \mid S[n] - S[i] > t_u, n \in [mid, high-1]\}, & \text{if exists} \\ high, & \text{if not exist} \end{cases}$;
      // BinarySearch is used to find $m$, $n$
7     $count \leftarrow count + n - m$;
8 $Merge(S, low, mid-1, high-1)$ ; // Merge is used for two sorted arrays
9 return $count$;
```

Example: Given $A = [1, -1, 2]$, $lower = 1$, $upper = 2$, return 4. The resulting four ranges should be $(1, 1)$, $(1, 3)$, $(2, 3)$, and $(3, 3)$.

Is Lucien's algorithm correct? Explain his idea and make correction if needed. Besides, compute the running time of Alg. 1 (or the corrected version) by recurrence relation. (Note: we can't implement Master's Theorem in this case. Refer Reference06 for more details.)

Solution. Lucien's idea is to solve the problem by Divide-and-conquer.

First find such kind of (i, j) such that All the elements are in the first half of the array.Then find such kind of (i, j) such that All the elements are in the second half of the array.Finally find (i, j) such that some elements are in the first half, and the other are in the second half.

All the value assignments have $O(1)$ complexity, which can be ignored in the top level.

The two **MergeCount**'s with inputs of $\frac{n}{2}$ in size have complexity of $2T(\frac{n}{2})$.

Line 5 and 6 have complexity of $O(nlogn)$ because **BinarySearch** has a complexity of $O(logn)$ and it is executed n times on average.

The **Merge** at last has a complexity of $O(n)$.

Therefore the recursion function is $T(n) = 2T(\frac{n}{2}) + O(nlogn)$.

Let $n = 2^k$.Then $k = logn$.From the recursion tree we can get

$T(n) = n \log n + 2 \times (\frac{n}{2} \log(\frac{n}{2})) + 4 \times (\frac{n}{4} \log(\frac{n}{4})) + \cdots$

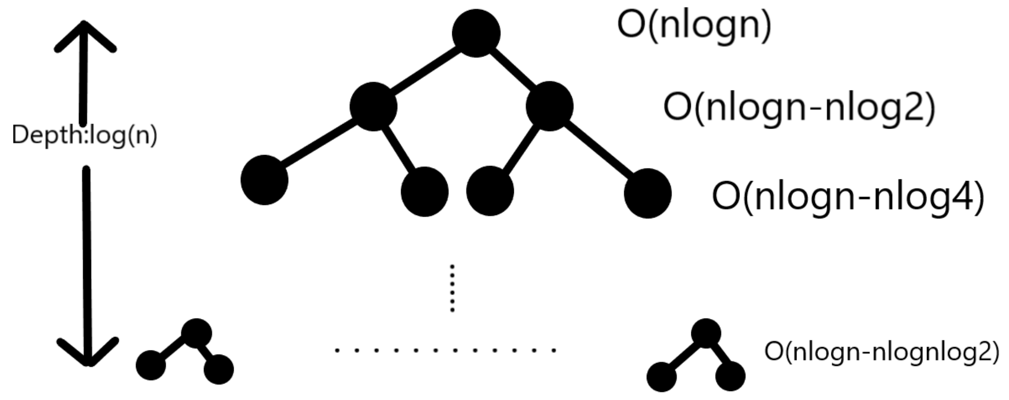

Figure 1: Reccurrence Tree

$$
\begin{aligned}
&= k2^k + 2(k-1) \times 2^{k-1} + \cdots \\
&= k2^k + (k-1)2^k + \cdots + 1 \times 2^k \\
&= 2^k \times \tfrac{k(k+1)}{2} \\
&= n\tfrac{\log n(\log n+1)}{2} \\
&= O(n \log^2 n)
\end{aligned}
$$

□

3. **Batcher's odd-even merging network.** In this problem, we shall construct an ***odd-even merging network***. We assume that n is an exact power of 2, and we wish to merge the sorted sequence of elements on lines $\langle a_1, a_2, \ldots, a_n \rangle$ with those on lines $\langle a_{n+1}, a_{n+2}, \ldots, a_{2n} \rangle$. If $n = 1$, we put a comparator between lines a_1 and a_2. Otherwise, we recursively construct two odd-even merging networks that operate in parallel. The first merges the sequence on lines $\langle a_1, a_3, \ldots, a_{n-1} \rangle$ with the sequence on lines $\langle a_{n+1}, a_{n+3}, \ldots, a_{2n-1} \rangle$ (the odd elements). The second merges $\langle a_2, a_4, \ldots, a_n \rangle$ with $\langle a_{n+2}, a_{n+4}, \ldots\ a_{2n} \rangle$ (the even elements). To combine the two sorted subsequences, we put a comparator between a_{2i} and a_{2i+1} for $i = 1, 2, \ldots, n-1$.

   (a) Replace the original Merger (taught in class) with Batcher's new Merger, and draw $2n$-input sorting networks for $n = 8, 16, 32, 64$. (Note: you are not forced to use Python Tkinter. Any visualization tool is welcome for this question.)

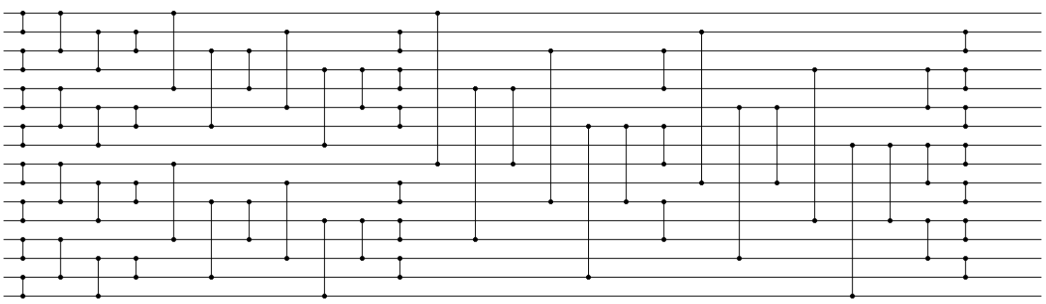

Figure 2: n=8

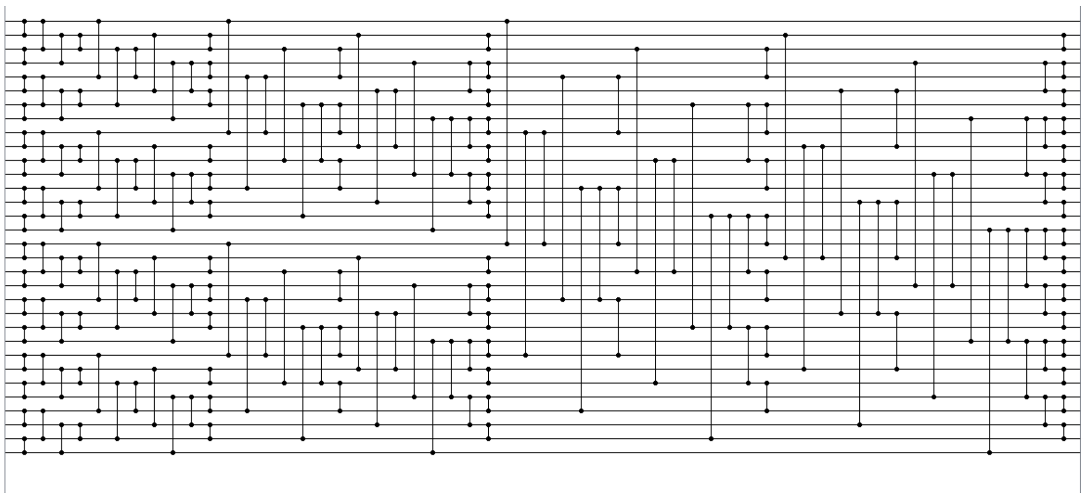
Figure 3: n=16

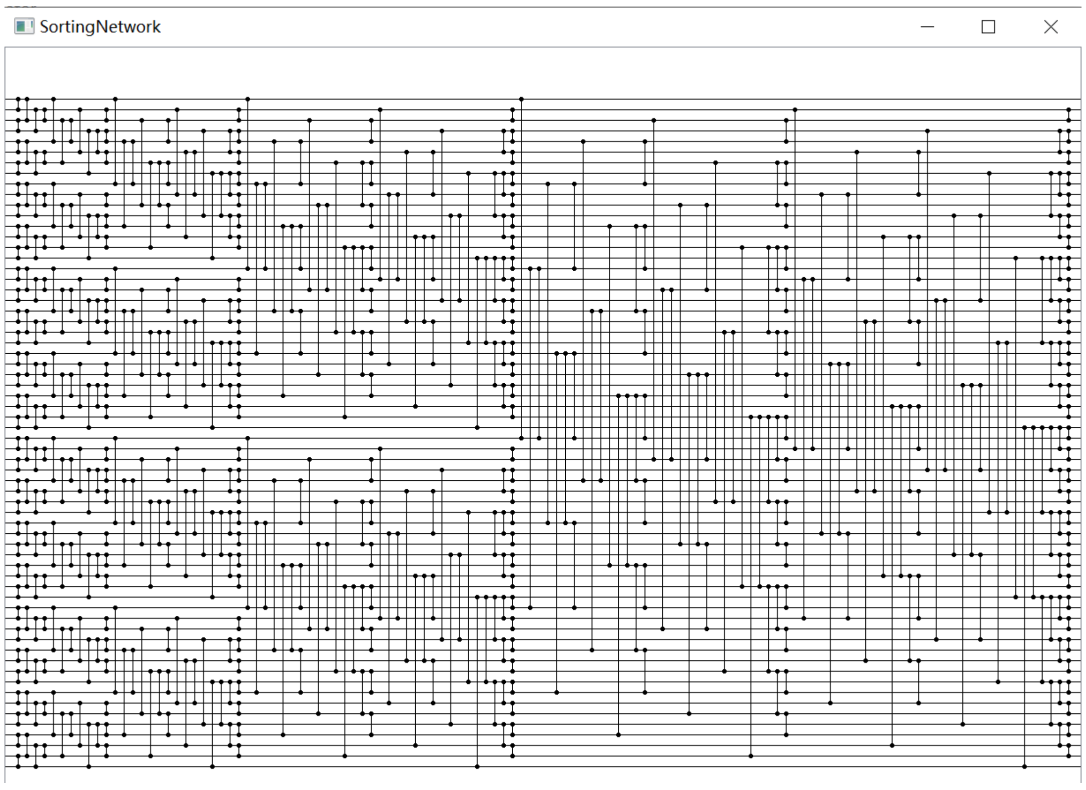

Figure 4: n=32

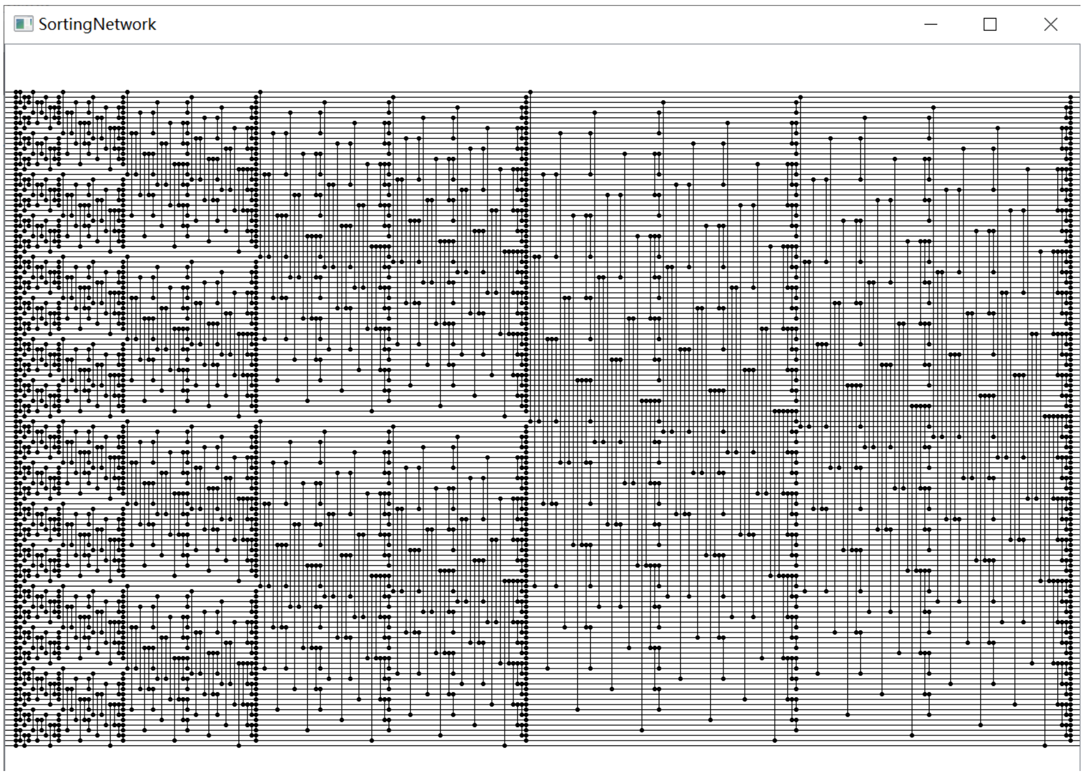

Figure 5: n=64

Solution. The picture above is drawn with Qt. "mainwindow.h","mainwindow.cpp""main.cpp""SortingNetwork.pro" are the source files. □

(b) What is the depth of a $2n$-input odd-even sorting network?

Solution. Assume that the depth of a $2n$-input odd-even sorting network is $D(n)$.
An $2n$-input odd-even sorting network is made up of 2 parallel n-input odd-even sortint network and a $2n$-input odd-even merging network.
As for an $2n$-odd-even merging network, it is made up of comparators with the length of $n = 2^k$,$\frac{n}{2} = 2^{k-1}$,$\cdots$,$1 = 2^0$.
Therefore it has a length of $\log n + 1$.
Therefore the recursion function of odd-even sorting network is:

$$\begin{cases} D(1) = 1 \\ D(n) = D(\frac{n}{2}) + \log n + 1 \end{cases} \tag{7}$$

Therefore the depth of a $2n$-input odd-even sorting network is:
$\frac{\log^2 n}{2} + \frac{3\log n}{2} + 1 = O(\log^2 n)$. □

(c) (Optional Sub-question with Bonus) Use the zero-one principle to prove that any $2n$-input odd-even merging network is indeed a merging network.

Proof. Prove that $2n$-input odd-even merging network can work well on merging 0-1 arrays by induction.

- When $n = 1$, only one comparator is needed. Obviously a comparator can sort a 0-1 array with 2 elements in nondecreasing order.

- When $n = \frac{k}{2}$, assume that k-input merging network can sort the k-sized 0-1 arrays in nondecreaseing order.
- Then when $n = k$, to build a $2k$-input merging network, we need to merge two k-sized arrays and then use a combiner to combine the two subarrays.
- The two k-sized arrays can be sorted in nondecreasing order separately, according to the former assumption.
- In $\langle a_1, a_2, \ldots, a_k \rangle$, the number of 0's is either odd or even.And the ranks of 0's are all smaller than 1's.So it is with $\langle a_{k+1}, a_{k+2}, \ldots, a_{2k} \rangle$.
  Therefore there are at most 2 more 0's in $\langle a_1, a_3, \ldots, a_{2k-1} \rangle$ than in $\langle a_2, a_4, \ldots, a_{2k} \rangle$.This happens when the number of 0's in $\langle a_1, a_2, \ldots, a_k \rangle$ and $\langle a_{k+1}, a_{k+2}, \ldots, a_{2k} \rangle$ are both odd numbers.(For example,consider $\langle 0, 0, 0, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 1, 1, 1 \rangle$.)
- We have assumed that the two k-sized subarrays can be sorted in nondecreasing order separately,but in the case above,there is a 010 subarray in $\langle a_1, a_2, \ldots, a_{2k} \rangle$,and the rank of the 1 is an even number.(The example array above can be sorted into $\langle 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 1, 1, 1, 1, 1, 1 \rangle$.)
  Then the 1 can be swapped with the 0 behind it with the help of combiner, which can make the array a completely sorted one.
- In other cases, the combiner is not neccessary because there is no such subarray like 010.The array is sorted even without the combiner.
- Therefore the odd-even merger and the combiner can make sure $\langle a_1, a_2, \ldots, a_{2k} \rangle$ are sorted in nondecresing order.
- By 0-1 principle and induction, Batcher's odd-even merging network is really a merging network.

$\square$

Remark: You need to include your .pdf, .tex and .py files (or other possible sources) in your uploaded .rar or .zip file.