Lab02-Divide and Conquer

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

∗ If there is any problem, please contact TA Yiming Liu.
∗ Name: Yulong Hui Student ID: 518030910059 Email: qinchuanhuiyulong@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. Every time we want to divide the array of n elements, there should be comparisons of $\Theta(n)$.

   Best. When it comes to the best case, the two subarrays should have the same number of elements of $n/2$. Then, we can get:

   $$T(n) = 2T(n/2) + \Theta(n)$$

   Because $2/2^1 = 1$, the $T(n)$ should be $\Theta(n \log n)$.

   Due to the best case, it can be expressed as $\Omega((r-p+1) \times log(r-p+1))$

   Average. We use the equation below:

   $$T(n) = n - 1 + T(i-1) + T(n-i) \qquad (n \geq 2)$$

   Obviously,

   $$T(1) = 0 \quad T(2) = 1 \tag{1}$$

   .

   Considering the average condition, we should think the possiblity of the value of i. Assume that the value of i between 1 and n is equally possible. So we can get an average $T(n)$:

   $$T(n) = n - 1 + \frac{1}{n} \times \sum_{i=1}^{n} [T(i-1) + T(n-i)]. \tag{2}$$

   $$T(n) = n - 1 + \frac{2}{n} \times \sum_{i=1}^{n-1} T(i). \tag{3}$$

   Then we will use the equation(1) and (3) to solve the $T(n)$:

   $$n \times T(n) = n(n-1) + 2 \times \sum_{i=1}^{n-1} T(i). \tag{4}$$

$$(n+1) \times T(n+1) = (n+1)n + 2 \times \sum_{i=1}^{n} T(i). \tag{5}$$

Let (5)-(4), we can get :

$$T(n+1) = \frac{n+2}{n+1}\, T(n) + \frac{2n}{n+1} \tag{6}$$

$$T(n) \leq \frac{n+1}{n}\, T(n-1) + 2 \tag{7}$$

Then calculate it in detail:

$$\begin{aligned} T(n) &\leq \frac{n+1}{n}\, T(n-1) + 2 \\ &= 2 + \frac{n+1}{n}\, [2 + \frac{n}{n-1}(\ldots(2 + \frac{3+1}{3} \times T(2)\ ))] \\ &= 2(n+1) \times [\frac{1}{n+1} + \frac{1}{n} + \cdots + \frac{1}{4} + \frac{1}{3} \times \frac{T(2)}{2}] \end{aligned}$$

Then we can get :

$$T(n) \leq 2(n+1) \times [\frac{1}{n+1} + \frac{1}{n-1} + \cdots + \frac{1}{4} + \frac{1}{3}]$$

Since

$$1 + \frac{1}{2} + \cdots + \frac{1}{n} = \ln n + \gamma + O(1/n)$$

We can finally get the average $T(n)$:

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

To sum up, the average time complexity is $O((r-p+1) \times log(r-p+1))$

Worst. In the worst case of quicksort, the two subarrays will have 1 and n-1 elements respectively. Therefore, we can get :

$$T(n) = T(n-1) + \Theta(n)$$

Then: $T(n) = \sum_{i=1}^{n} \Theta(i) = \Theta(n^2)$, let $n = r - p + 1$.

So, the worst complexity is $\Theta((r-p+1)^2)$.

Because it corresponds the worst case, it can also be expressed as $O((r-p+1)^2)$.

□

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

$$t_l \leq \sum_{k=i}^{j} A[k] \leq t_u. \tag{1}$$

Before computation, he firstly constructed $S[0 \ldots n+1]$, where $S[i]$ denotes the sum of the first i elements of $A[1 \ldots n]$. 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. (1).

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 $\quad m \leftarrow \begin{cases} \min\{m \mid S[m] - S[i] \geq t_l, m \in [mid, high-1]\}, & \text{if exists} \\ high, & \text{if not exist} \end{cases}$;
6 $\quad n \leftarrow \begin{cases} \min\{n \mid S[n] - S[i] > t_u, n \in [mid, high-1]\}, & \text{if exists} \\ high, & \text{if not exist} \end{cases}$;
$\quad$ `// BinarySearch is used to find` m, n
7 $\quad count \leftarrow count + n - m$;
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. First at all, Lucien's algorithm is correct.

The main idea to solve this problem is two pick out the $< i, j >$ and let the "$S[j] - S[i]$" meet the requirement. Lucien divides the S-array into two subarray everytime. The two have been sorted and the numbers of correct $< i, j >$ have been calculated. Then, the combination we need to do is: (1) pick out the correct $< i, j >$ which spans the two subarrays. (2)mergesort the two into one array.

From line 4 to line 7, there is a loop for $\frac{n}{2}$ times. And there are $O(\log n)$ comparions for the binarysearch inside the loop. So we can get that:

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

Then we can use a recurrence tree, and get: the work done at the j-th level is

$$2^{j-1} \times O\left(\frac{n}{2^{j-1}} \log \frac{n}{2^{j-1}}\right)$$

The total work done is

$$\sum_{j=1}^{\log_2 n+1} 2^{j-1} \times \left(\frac{n}{2^{j-1}} \log \frac{n}{2^{j-1}}\right)$$

$$= \sum_{j=0}^{\log_2 n} 2^{j} \times \left(\frac{n}{2^{j}} \log \frac{n}{2^{j}}\right)$$

$$= \sum_{j=0}^{\log_2 n} n(\log n - j)$$

$$= (\log n + 1) \times n \log n - n \times \frac{(\log n + 1) \log n}{2}$$

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

So, $T(n) = O(n(\log n)^2)$.

□

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.)

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

   (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.

Solution. (a)The pictures are as follows

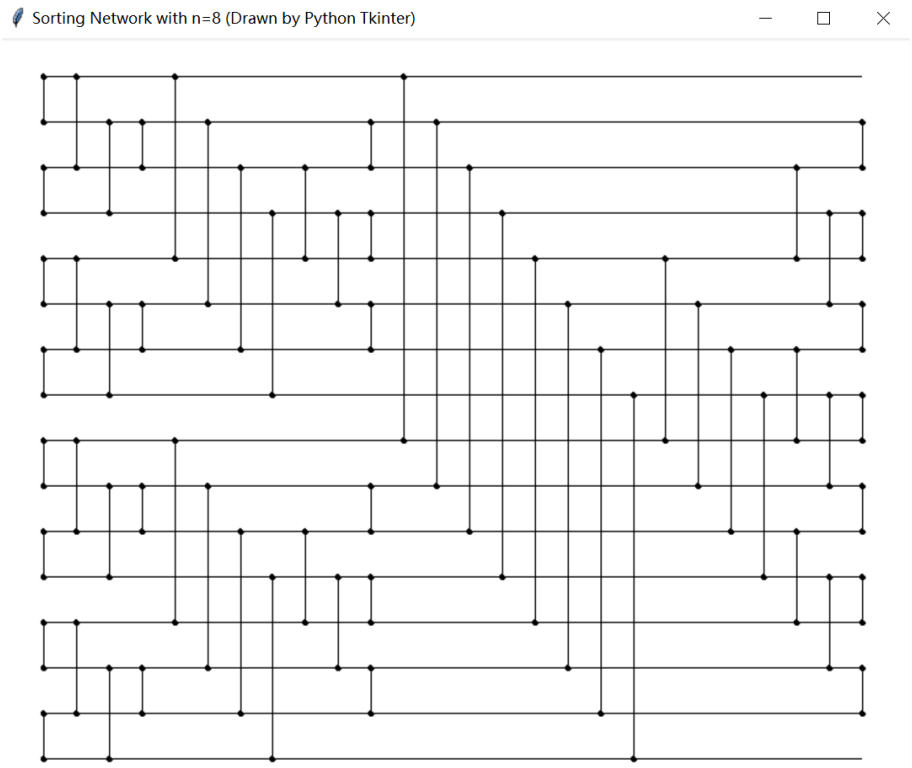

Figure 1: Picture for n=8

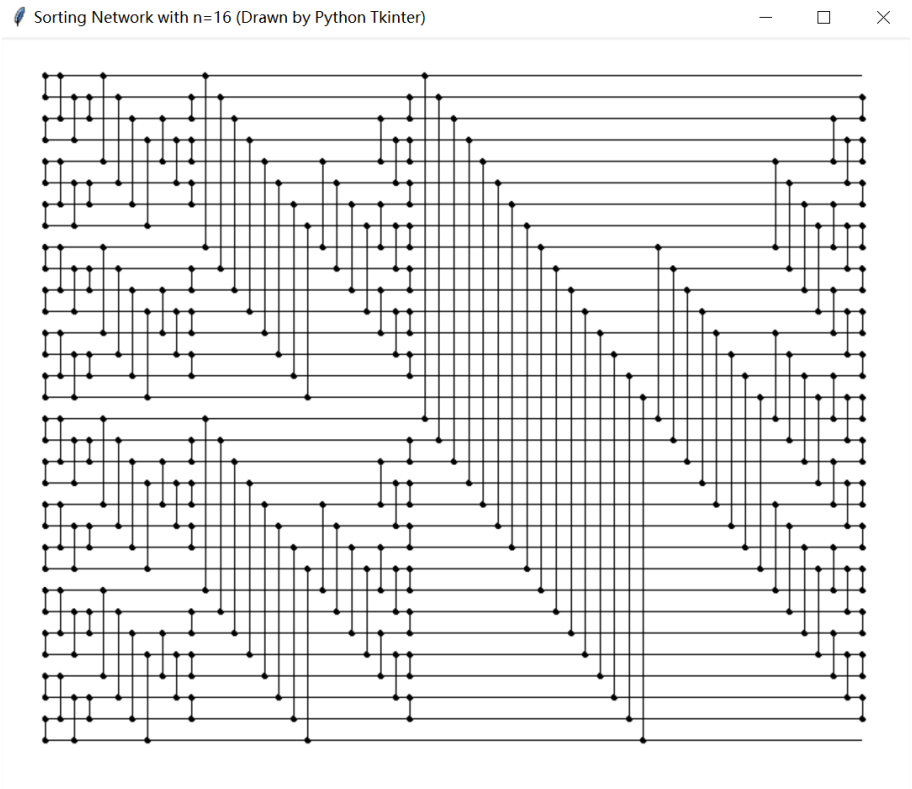

Figure 2: Picture for n=16

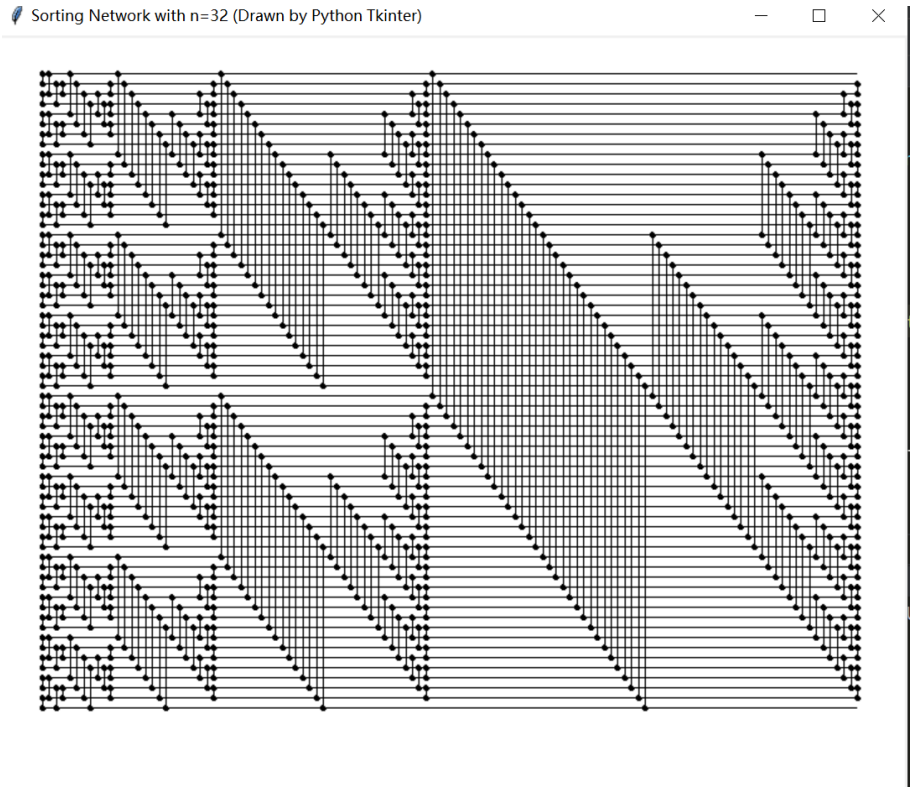

Figure 3: Picture for n=32

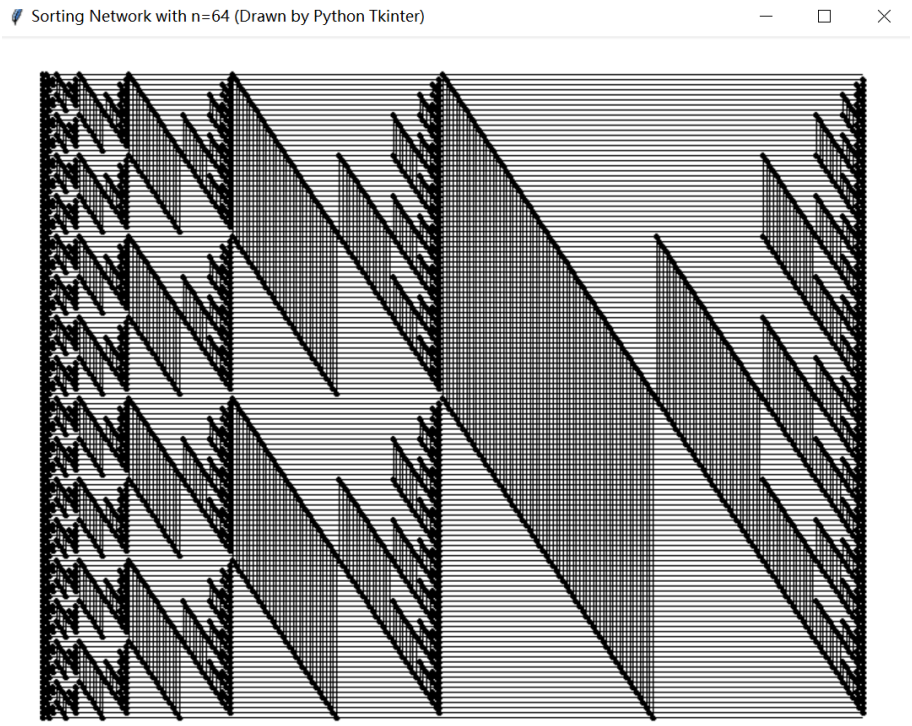

Figure 4: Picture for n=64

(b) When we deal with a sorting problem with 2n number, we should get two n-sorted array and then do the 2n-array merge. So we can just work on the depth of the 2n-merging, and then do recursive calculation to get the total depth.

When merging 2n elements, we should merge two n-array and then combine them. Keep doing this until there are 1-arrays which don't need division.Considering that the combinations can be parallel, we can get the depth of n-merging: $\log 2n$.

Then, about the total depth:

$$depth = \log 2n + \log n + \log \frac{n}{2} + \log \frac{n}{4} + \cdots + \log 2 + \log 1$$

$$depth = \frac{(1 + \log 2n) * \log 2n}{2}$$

So, the depth of a 2n-input odd-even sorting network is $O(\log^2 n)$.

(c) First we need to introduce the domain conversion lemma, which says if a merging net work transform the input $\mathbf{a} =< a_1, a_2, \cdots, a_n >$ and $\mathbf{b} =< b_1, b_2, \cdots, b_n >$into the output sequence $\mathbf{c} =< c_1, c_2, \cdots, c_n >$,then for any monotonically increasing function f, the network transforms the input sequence $f(\mathbf{a}), f(\mathbf{b})$ to $f(\mathbf{c})$.

Now, prove the introduced lemma by induction:

Basis. When there is only one element in both input sequences, it is obvious that the lemma is correct.

Assumption If a wire assumes the value i (from $\mathbf{a}$ or $\mathbf{b}$), then it assumnes the value $f(i)$ (from f($\mathbf{a}$) or f($\mathbf{b}$)).

Inducion A comparator at depth $d \geq 1$ disposes the input from the depth $d_i < d$. By assumtion the output at depth d_i should be transformed from i, j to $f(i)$ and $f(j)$, then the

comparator at depth d will dispose the $f(i)$ and $f(j)$. If i is smaller, then the $f(i)$ will be smaller and take the place of i. Then, the lemma is proved.

Then we can see that the zero-one principle applies to this case.(The continuing proof is similar to that in class).

Then, we will use the zero-one principle and prove the correctness for zero-one sequences.

Conclude the operations for a 2n-input ($< a_1, a_2, a_3, \cdots, a_n >$and $< a_{n+1}, \cdots, a_{2n} >$) as 4 steps :

(1) Divide the 2n-inputs into two $\frac{n}{2}$-odd groups and two $\frac{n}{2}$-even groups.

(2) Merge the two $\frac{n}{2}$-odd groups($< a_1, a_3, \cdots >$and$< a_{n+1}, a_{n+3}, \cdots >$) into a n-odd group

(3) Parallelly merge the two $\frac{n}{2}$-even groups($< a_2, a_4, \cdots >$and$< a_{n+2}, a_{n+4}, \cdots >$) into a n-even group

(4) Combine the n-odd group and the n-even group by comparing and adjusting between the a_{2i} and a_{2i+1}.

Then we will prove the correctness of the network by induction.

Basis. It is easy to see, when k=1, the input is 2, the network for 2-input is correct.

Assumption For any k=n/2, the input is k, the network is correct.

Inducion When k=n, we need to prove the correctness:

Because of the assumption, the step(2)(3) can be finished correctly, then we just need to prove the step(4):

We can see that the first element in n-odd (n-even) group must be the a_1 or a_{n+1} (a_2 or a_{n+2}). Then because $a_1 \leq a_2$ and $a_{n+1} \leq a_{n+2}$, the first element in final 2n-output (called **c**) must be from the n-odd group. So, to get the final sequence, there is no need to compare the c_1 and c_2. With the same reason, there is no need to compare the c_3 and c_4, c_5 and c_6 and so on.

Therefore, we just need to compare and adjust the c_2 and c_3, c_4 and c_5 and so on, which is exactly what the step (4) does. So, we can see the correctness of step(4).

Then the correctness is finally proved.

PS: The last proof needn't use 0-1 sequence, but because the process is kind of abstract, 0-1 sequence can be used to help simplify the question, which will not affect the result because of the 0-1 principle.

□

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