Lab02-Divide and Conquer

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

∗ If there is any problem, please contact TA Yiming Liu.
∗ Name:Hongjie Fang Student ID:518030910150 Email: galaxies@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. Assume the recurrence function $T(n)$ stands for total operation times of Quicksort, where n denotes the current length of the array (i.e., $n = r - p + 1$). Suppose we choose the pivot $A[q]$ in a time complexity of $O(1)$, and finish the partition in a time complexity of $O(n)$. We consider the best case, the worst case and the average case of the Quicksort.

   - **The best case:** The best case happens when every time we take a partition, two parts ($A[p \dots q-1]$ and $A[q+1 \dots r]$) have almost the same size, which indicates that q equals to $\lfloor \frac{p+r}{2} \rfloor$ in every partitioning procedure. Then suppose the original length of the array is n before partition, and after partition we will get two sub-arrays of length $\lfloor \frac{n}{2} \rfloor$ each. Thus, we can derive the following equation (Equation (1)).

   $$T(n) = \begin{cases} 0 & \text{if } n = 1 \\ 2T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + O(n) & \text{if } n > 1 \end{cases} \tag{1}$$

   We can derive that $T(n) = O(n \log n)$ using Master Theorem. So the best case of Quicksort has a time complexity of $O(n \log n)$.

   - **The worst case:** The worst case happens when every time we take a partition, one of the two parts ($A[p \dots q-1]$ and $A[q+1 \dots r]$) is empty, which indicates that q equals to either p or r. Then suppose the original length of the array is n before partition, and after partition we will get an empty sub-array and a sub-array of length $(n-1)$. Thus, we can derive the following equation (Equation (2)).

   $$T(n) = \begin{cases} 0 & \text{if } n = 1 \\ T(n-1) + O(n) & \text{if } n > 1 \end{cases} \tag{2}$$

   We can derive that $T(n) = O(n^2)$ using simple mathematical knowledge. So the worst case of Quicksort has a time complexity of $O(n^2)$.

   - **The average case:** When considering the average case of the Quicksort, we assume that every possible partition happens equally likely. Suppose one of the two parts ($A[p \dots q-1]$ and $A[q+1 \dots r]$) has a length i ($0 \le i < n$), then the other part has a length $(n-i-1)$.

There are total n possibilities and each has a probability of $\frac{1}{n}$. Thus, we can derive the following equation (Equation (3)).

$$T(n) = \begin{cases} 0 & \text{if } n = 1 \\ O(n) + \sum_{i=0}^{n-1} \frac{1}{n} \cdot (T(i) + T(n-i-1)) & \text{if } n > 1 \end{cases} \tag{3}$$

We can simplify the Equation (3) and make some mathematical derivation as follows.

$$\begin{aligned} \frac{n}{n-1} \cdot T(n) - T(n-1) &= \frac{2}{n-1} \left(\sum_{i=0}^{n-1} T(i) - \sum_{i=0}^{n-2} T(i) \right) + O(1) \\ &= \frac{2}{n-1} \cdot T(n-1) + O(1) \qquad (n > 1) \\ \Longrightarrow \qquad T(n) &= \frac{n+1}{n} \cdot T(n-1) + O(1) \qquad (n > 1) \end{aligned}$$

Solve the recurrence equation above, we can get that $T(n) = H(n) \cdot O(n)$, where $H(n)$ is the sum of the first n terms of the harmonic series. Therefore, we can derive that $T(n) = O(n \log n)$ using the Euler's conclusion that $H(n) \sim \log n$. So the average case of Quicksort has a time complexity of $O(n \log n)$.

□

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

$$t_l \le \sum_{k=i}^{j} A[k] \le t_u. \tag{4}$$

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. (4).

$count \leftarrow 0$; $mid \leftarrow \lfloor \frac{low+high}{2} \rfloor$;
if $mid = low$ **then return** 0
$count \leftarrow MergeCount(S, t_l, t_u, low, mid) + MergeCount(S, t_l, t_u, mid, high)$;
for $i = low$ ***to*** $mid - 1$ **do**
  $m \leftarrow \begin{cases} \min\{m \mid S[m] - S[i] \ge t_l, m \in [mid, high-1]\}, & \text{if exists} \\ high, & \text{if not exist} \end{cases}$;
  $n \leftarrow \begin{cases} \min\{n \mid S[n] - S[i] > t_u, n \in [mid, high-1]\}, & \text{if exists} \\ high, & \text{if not exist} \end{cases}$;
  `// BinarySearch is used to find` m, n
  $count \leftarrow count + n - m$;
$Merge(S, low, mid-1, high-1)$; `// Merge is used for two sorted arrays`
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 algorithm is **correct**. Here is Lucien's idea.

- Notice that every pair $(S[i], S[j])$ $(0 \le i < j \le n)$ represents a range $(i, j]$, and the sum of elements in it is $(S[j] - S[i])$ (i.e., $\sum_{k=i+1}^{j} A[k] = S[j] - S[i]$). And also every range can be represent as a pair $(S[i], S[j])$ $(0 \le i < j \le n)$. Hence, we are going to count the number of pairs $(S[i], S[j])$ $(0 \le i < j \le n)$ satisfying $t_l \le S[j] - S[i] \le t_u$.
- Assume that current process is $MergeCount(S, t_l, t_u, low, high)$, in which we want to count the number of pairs with indexes in $[low, high)$ that satisfy the conditions above, then let mid be $\lfloor \frac{low+high}{2} \rfloor$. Divide the current interval into two small sub-intervals, which are $[low, mid)$ and $[mid, high)$.
- Assume that pairs we concern about is $(S[i_1], S[j_1]), (S[i_2], S[j_2]), ..., (S[i_m], S[j_m])$, and we want to check whether these pairs satisfy the condition. Obviously, the indexes of these ranges must be in the current interval, that is, $i_k, j_k \in [low, high)$ $(1 \le k \le m)$. Divide these pairs into three parts:
  - **Situation I:** Pairs with both indexes in the left sub-interval, that is, all the pairs $(S[i_k], S[j_k])$ $(1 \le k \le m)$ satisfying $low \le i_k < j_k < mid$;

  - **Situation II:** Pairs with both indexes in the right sub-interval, that is, all the pairs $(S[i_k], S[j_k])$ $(1 \le k \le m)$ satisfying $mid \le i_k < j_k < high$;
  - **Situation III:** Pairs with indexes crossing two sub-intervals, that is, all the pairs $(S[i_k], S[j_k])$ $(1 \le k \le m)$ satisfying $low \le i_k < mid \le j_k < high$.

  These three situations contain all the pairs we concern about, and every two situations have a empty intersection set of pairs. Thus, this divide method is correct.

- Situation I and Situation II are easy to solve, since we can count the number of pairs by calling $MergeCount(S, t_l, t_u, low, mid)$ and $MergeCount(S, t_l, t_u, mid, high)$ recursively.
- Situation III is a little bit difficult, so we use the following method to count the number of pairs.
  - We enumerate i from low to $(mid - 1)$ to count the number of pairs with the first element being $S[i]$ (i.e., pairs in this pattern $(S[i], \cdot)$).
  - Given the first index i, we want to count the number of indexes j $(mid \le j < high)$ satisfying $t_l + S[i] \le S[j] \le t_u + S[i]$.
  - Now assume $S[mid \dots high]$ is already sorted, then the problem can be solved by binary search (search the beginning index and the ending index that satisfy the condition, then all the indexes between them are valid) with a time complexity of $O(\log(high - mid))$ each time.
- Now we come back to the assumptions we make above. We need the array S be sorted. Notice that the framework of the algorithm is similar to MergeSort, so we call the function merge in MergeSort to merge two already-sorted part $S[low \dots mid]$ and $S[mid \dots high]$ together. Therefore, the assumption in situation III is satisfied.
- Combining the number of pairs in Situation I, II, III together, we can get the number of pairs with indexes in $[low, high)$ that satisfy the conditions.
- **Explanation of correctness:** The main point of the correctness proof is that whether the sorting process will affect the answer, and the answer is no. Every time we count the pairs in Situation III, we have fix their left index in left sub-interval and right index in right sub-interval, so left index is bound to be less than the right index. When counting pairs, we do not care about the original places of indexes in the sub-intervals. Thus, the sorting processes of the left sub-interval and the right sub-interval have no effect on the answer. Thus the correctness of sorting process in the algorithm is proved.

The Time Complexity: Now we compute the time complexity of the algorithm. Assume the recurrence function is $T(n)$ with $n = high - low$, then we can derive the equation as follows (Equation (5)).

$$T(n) = \begin{cases} 0 & \text{if } n = 1 \\ 2T\left(\left\lceil \frac{n}{2} \right\rceil\right) + O(n \log n) & \text{if } n > 1 \end{cases} \tag{5}$$

Then we can unfold the recurrence equation as follows (Equation (6)).

$$\begin{aligned} T(n) &= 2T\left(\left\lceil \frac{n}{2} \right\rceil\right) + O(n \log n) \\ &= 4T\left(\left\lceil \frac{n}{4} \right\rceil\right) + 2O\left(\left\lceil \frac{n}{2} \right\rceil \log \left\lceil \frac{n}{2} \right\rceil\right) + O(n \log n) \\ &= \dots \\ &= \sum_{k=0}^{\lceil \log n \rceil - 1} 2^k \cdot O\left(\left\lceil \frac{n}{2^k} \right\rceil \log \left\lceil \frac{n}{2^k} \right\rceil\right) \qquad (n > 1) \end{aligned} \tag{6}$$

Then we can transform the equation using simple mathematical knowledge (Equation (7)).

$$\begin{aligned} T(n) &= \sum_{k=0}^{\lceil \log n \rceil - 1} 2^k \cdot O\left(\left\lceil \frac{n}{2^k} \right\rceil \log \left\lceil \frac{n}{2^k} \right\rceil \right) \\ &= O(n) \cdot \sum_{k=0}^{\lceil \log n \rceil - 1} O\left(\log \left\lceil \frac{n}{2^k} \right\rceil \right) \\ &= O(n) \cdot O\left(\log^2 n - \sum_{k=0}^{\lceil \log n \rceil - 1} k \right) \\ &= O(n \log^2 n) \qquad (n > 1) \end{aligned} \tag{7}$$

Thus, the time complexity of MergeCount (Alg. 1) is $O(n \log^2 n)$. □

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 $i = 1, 2, \ldots, n-1$.

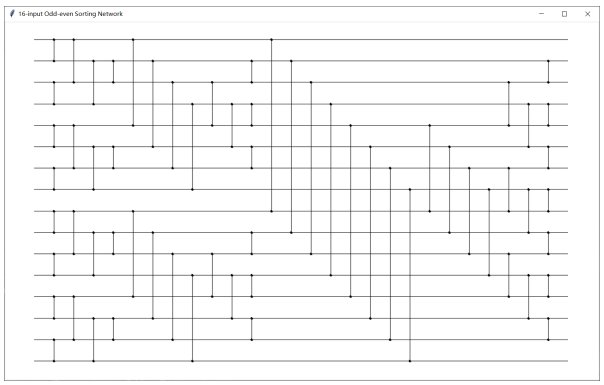
   (a) Replace the original Merger (taught in class) with Batcher's new Merger, and draw $2n$-input sorting networks for $n = 8, 16, 32, 64$. (Note: you are not forced to use Python Tkinter. Any visualization tool is welcome for this question.)

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

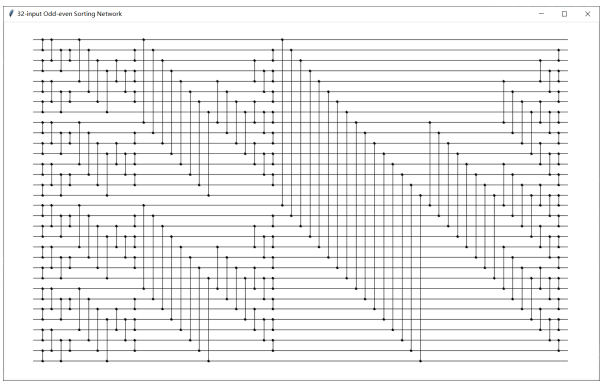
   (c) (Optional Sub-question with Bonus) Use the zero-one principle to prove that any $2n$-input odd-even merging network is indeed a merging network.

Solution. The answers to the questions are listed as follows.

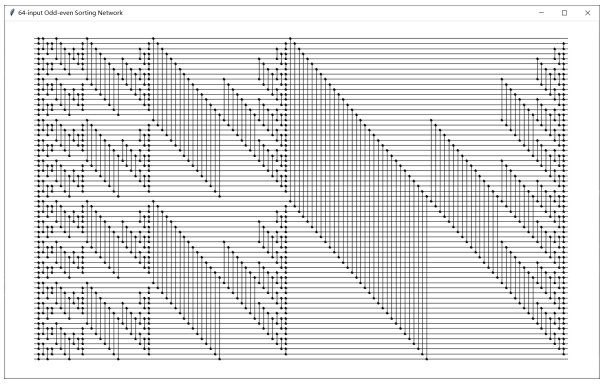
(a) I use Python Tkinter to draw the pictures of the new sorting network (Fig. 1). The visualization code is in vis.py. Some special tricks are used in the code to show the parallel process of the sorting network, and make the figures in alignment.



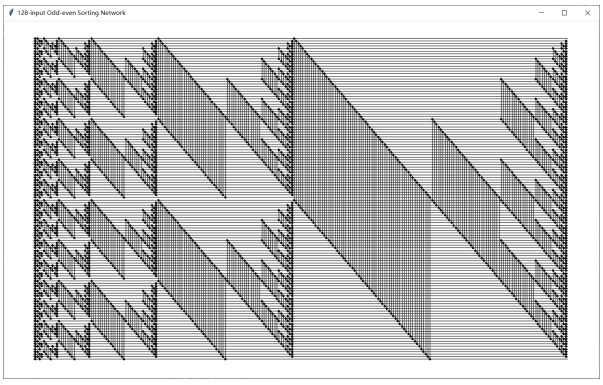
(a) $n = 8$



(b) $n = 16$



(c) $n = 32$



(d) $n = 64$

Figure 1: $2n$-input odd-even sorting networks

(b) Let $D_m(n)$ be the depth of $2n$-input odd-even merging network. Then we have the following equation (Equation (8)) according to the definition of odd-even merging network.

$$D_m(n) = \begin{cases} 1 & \text{if } n = 1 \\ D_m\left(\frac{n}{2}\right) + 1 & \text{if } n = 2^k \text{ and } k \geq 1 \end{cases} \tag{8}$$

It is obvious that $D_m(n) = O(\log n)$.

Let $D(n)$ be the depth of $2n$-input odd-even sorting network. Then we have the following equation (Equation (9)).

$$D(n) = \begin{cases} 1 & \text{if } n = 1 \\ D\left(\frac{n}{2}\right) + D_m(n) & \text{if } n = 2^k \text{ and } k \geq 1 \end{cases} \tag{9}$$

Suppose $n = 2^k$ $(k \geq 1)$, then we can solve the recurrence equation (Equation (9)) as follows (Equation (10)).

$$\begin{aligned} D(n) &= D\left(\frac{n}{2}\right) + O(\log n) \\ &= D\left(\frac{n}{4}\right) + O\left(\log \frac{n}{2}\right) + O\left(\log n\right) \\ &= ... \\ &= 1 + \sum_{i=0}^{k-1} O\left(\log \frac{2^k}{2^i}\right) \\ &= O(k^2) \\ &= O(\log^2 n) \qquad (n = 2^k, k \geq 1) \end{aligned} \tag{10}$$

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

(c) Suppose all the sequences we will discuss later are indexed from 1.

Lemma 1. *Suppose there is a sorted 01-sequence of even length, and the even-indexed 1's and odd-indexed 1's of the sequence are n_{even} and n_{odd} respectively. Then we have $n_{odd} \leq n_{even} \leq n_{odd} + 1$.*

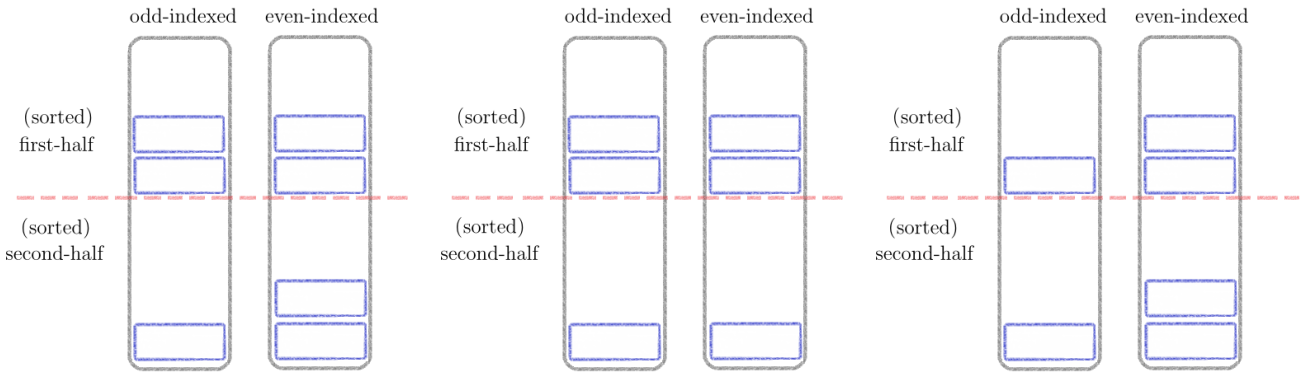
Proof. The 1's must be placed in **the last few continuous places** of the sorted 01-sequence. On one hand, the last place of the sequence is even-indexed, because the sequence has an even length and it is indexed from 1. Therefore, the even-indexed 1's can not be less than the odd-indexed 1's, that is, $n_{even} \geq n_{odd}$. On the other hand, n_{even} cannot exceed $n_{odd} + 1$, since the 1's in the sequence are continuous. Therefore, we can get $n_{odd} \leq n_{even} \leq n_{odd} + 1$. □

Theorem 2. *Let $Q(k)$ be the statement that the 2^k-input odd-even merging network can merge two sorted 01-sequences with the same length into one correctly. Then $Q(k)$ is true for every $k \geq 1$.*

Proof. Let us prove the theorem by induction.

- **Basis step.** $Q(1)$ is obviously true, since the 2-input merging network only contains a comparator and it can merge two sorted 01-sequences of length 1 into one correctly.
- **Induction Hypothesis.** Assume $Q(i)$ is true for some $i \geq 1$, that is, a 2^i-input odd-even merging network can merge two 01-sequences of length 2^{i-1} into one correctly.

- **Proof of Induction Step.** Now let's prove $Q(i+1)$ is true. A 2^{i+1}-input odd-even merging network includes two 2^i-input odd-even merging networks, which can sort the even-indexed part and the odd-index part of the sequence. Suppose the number of even-indexed 1's and odd-indexed 1's are n_{even} and n_{odd} respectively after sorting the even-indexed sub-sequence and the odd-indexed sub-sequence. According to Lemma 1, there are only three situations of the two sorted halves (the first half and the second half) in the beginning, which are shown in Fig. 2 (the blue block stands for 1 in the sequence while the white block stands for 0 in the sequence).
  - **Situation a:** (Fig. 2(a)) In one half, the number of even-indexed 1's is greater than the number of odd-indexed 1's; in the other half, the number of even-indexed 1's equals to the number of odd-indexed 1's. According to the previous lemma, after sorting the odd-indexed sub-sequence and even-indexed sub-sequence respectively, we have $n_{even} = n_{odd} + 1$. After combining two sub-sequence together, the 1's are all in the last continuous places of the sequence (i.e., the sequence looks like 00...011...1). Thus the sequence is sorted.
  - **Situation b:** (Fig. 2(b)) The number of even-indexed 1's equals to the number of the odd-indexed 1's in both two parts. According to the previous lemma, after sorting the odd-indexed sub-sequence and even-indexed sub-sequence respectively, we have $n_{even} = n_{odd}$. After combining two sub-sequence together, the 1's are all in the last continuous places of the sequence (i.e., the sequence looks like 00...011...1). Thus, the sequence is sorted.
  - **Situation c:** (Fig. 2(c)) The number of even-indexed 1's is greater than the number of the odd-indexed 1's in both two parts. According to the previous lemma, after sorting the odd-indexed sub-sequence and even-indexed sub-sequence respectively, we have $n_{even} = n_{odd} + 2$. After combining two sub-sequence together, most of the 1's are all in the last continuous places of the sequence and only an 1 is located in a places before the last continuous 1's (i.e., the sequence looks like 00...01011...1). Then we perform the last step of the merging network: use comparators between the odd-indexed and the even-indexed. The sequence will be sorted after the process, since the special 1 is even-indexed and it will be swapped with the 0 between 1's.



(a) Situation a (b) Situation b (c) Situation c

Figure 2: Sequence before merging

In summary, two sorted 01-sequences with the same length will be merged into a sorted sequence after putting it into the 2^{i+1}-input odd-even merging network. Thus $Q(i+1)$ is true.

□

Suppose $n = 2^k$ ($k \geq 0$). According to Theorem 2, we know that $Q(k+1)$ is true, that is, two sorted 01-sequences with the same length will be merged into a sorted sequence after putting it into the 2^{k+1}-input odd-even merging network. Hence, we prove that the $2n$-input odd-even merging network is indeed a merging network, since $2n = 2^{k+1}$.

According to **the zero-one principle**, we know that any $2n$-input odd-even merging network is indeed a merging network.

□

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