Lab02-Divide and Conquer

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

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

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. Let $T(N)$ be the time needed to sort N elements. $T(0) = c$, where c is a constant. The recursive relation is: $T(N) = T(LeftSz) + T(RightSz) + O(N)$, where $LeftSz + RightSz = N - 1$.

   (a) **Best case.** Best case happens when each time the pivot divides the array into two equal-sized ones. $T(N) = T(\frac{N-1}{2}) + T(\frac{N-1}{2}) + O(N)$ · The recursive relation is similar to that of merge sort. By the Master's Theorem, $T(N) = O(N \log N)$.

   (b) **Worst case.** Worst case happens when each time the pivot is the smallest item or the largest item.

$$\begin{aligned} T(N) &= T(N-1) + T(0) + O(N) \\ &\leq T(N-1) + T(0) + dN \\ &\leq T(N-2) + 2T(0) + d(N-1) + dN \\ &\leq T(0) + NT(0) + d + 2d + \cdots + d(N-1) + dN \\ &= O\left(N^2\right) \end{aligned}$$

   (c) **Average case.** For a termination situation, we have $T(0) = T(1) = 1$.
   For some length $N(N \geq 2)$, since the final position of pivot is equally possible for all positions, we have:

$$\begin{aligned} T(N) &= \frac{1}{n} \sum_{i=1}^{N} \left(T(i-1) + T(N-i) + CN\right) \\ &= \frac{2}{N} \sum_{i=0}^{N-1} T(i) + CN \end{aligned}$$

   where C is a constant. Solving the recursion we get $T(N) = \mathcal{O}(N \log N)$

□

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

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

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. (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 $\quad m \leftarrow \begin{cases} \min\{m \mid S[m] - S[i] \ge t_l, m \in [mid, high-1]\}, & \text{if exists} \\ high, & \text{if not exist} \end{cases}$;
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. Lucien's algorithm is correct. I will briefly explain the key points in the algirithm.

- For *MergrCount*, note that we count the number of pairs with indexes in $[\text{low}, high-1]$, not $[low, high]$. Note that $S[high]$ has no practical meaning, and we use its index $high$ to calculate $count$. The index $high$ is only used in line 5 and 6 when m and n don't exist. Therefore, similarly, for the two sub-problems, we count the number of pairs with indexes in $[low, mid-1]$ and $[mid, high-1]$.
- The *Merge* function in line 8 is necessary, or we can't use binary search in line 5 and 6. Note that before merging, the pairs satisfying Eqn. (1) in the subarrays have been considered completely. Therefore, the order of the two subarrays could be shuffled.
- We divide the problem into 3 subproblems:
  (a) Pairs with index satisfying $low \le i < j \le mid - 1$;
  (b) Pairs with index satisfying $mid \le i < j \le high - 1$;
  (c) Pairs with index satisfying $mid \le i < mid \le j \le high - 1$. (Note that even if $S[low, mid-1]$ and $S[mid, high-1]$ are sorted. The relative position of the elements $S[i]$ and $S[j]$ with index i in $[low, mid-1]$ and j in $[mid, high-1]$ is not changed.)

Since in each divide step, we reduce the problem size by half, and in each merge step, we use a binary search algorithm and a sorting algorithm with time complexity of $O(n \log n)$, it is easy to write the recurrence for the overall running time as follows.

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

The corresponding recurrence tree can be drawn as follows.

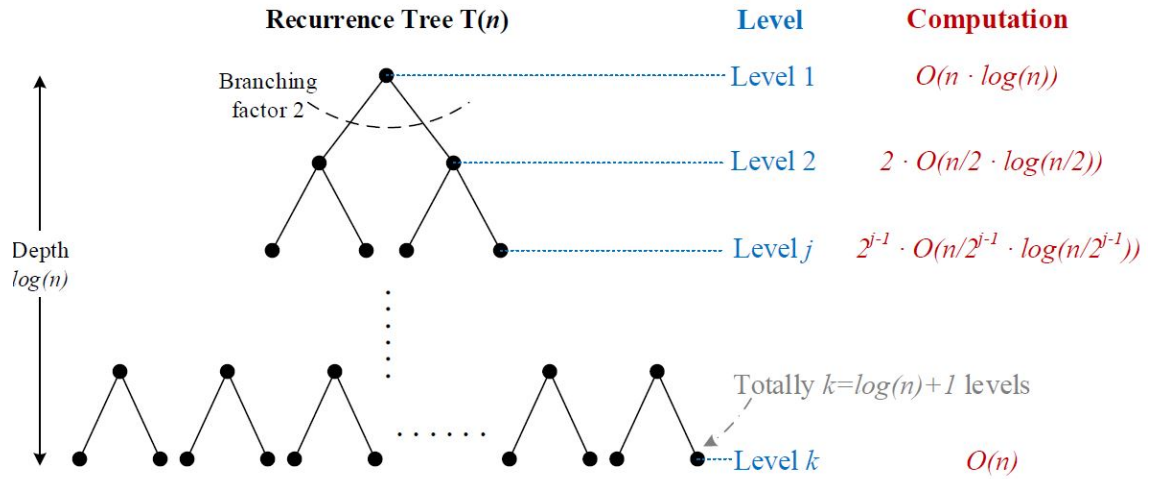

Figure 1: The Recurrence Tree for $T(n) = 2T(n/2) + O(n \log n)$

According to the recurrence tree, we have

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

Therefore, the solution of the recurrence is $T(n) = O\left(n \log^2 n\right)$.

□

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?

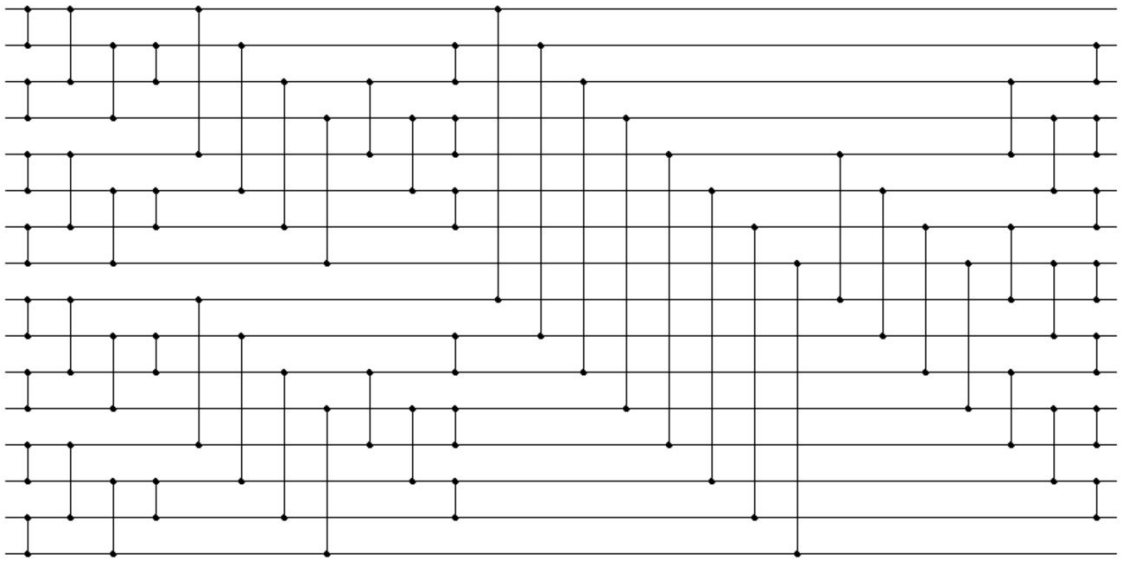

Figure 2: $n = 8$

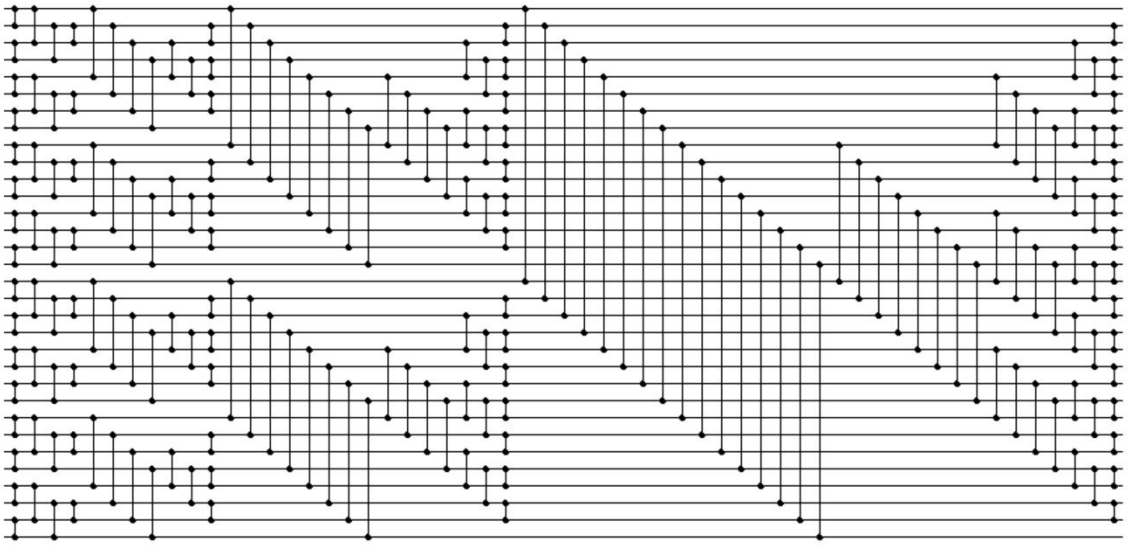

Figure 3: $n = 16$

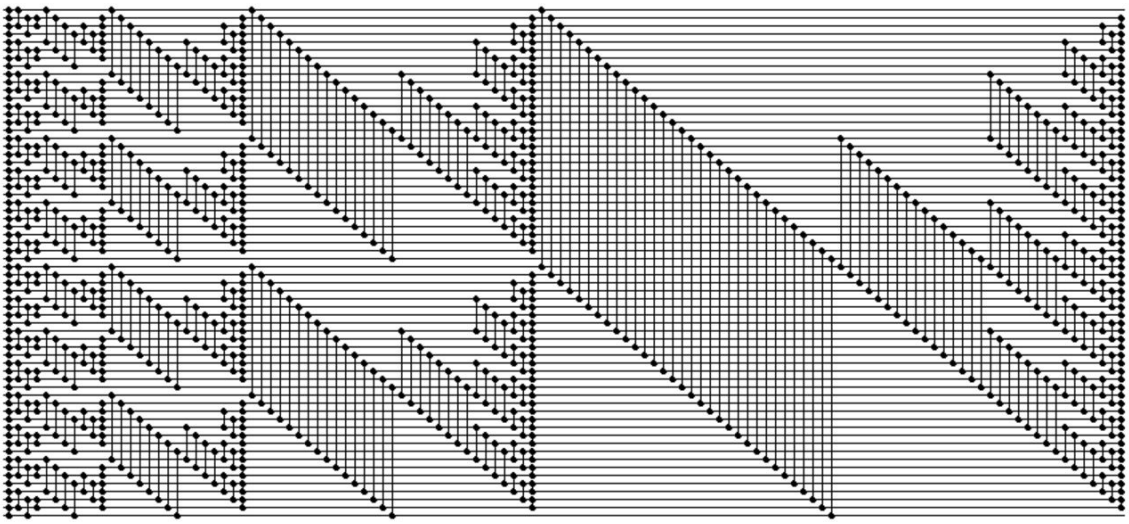

Figure 4: $n = 32$

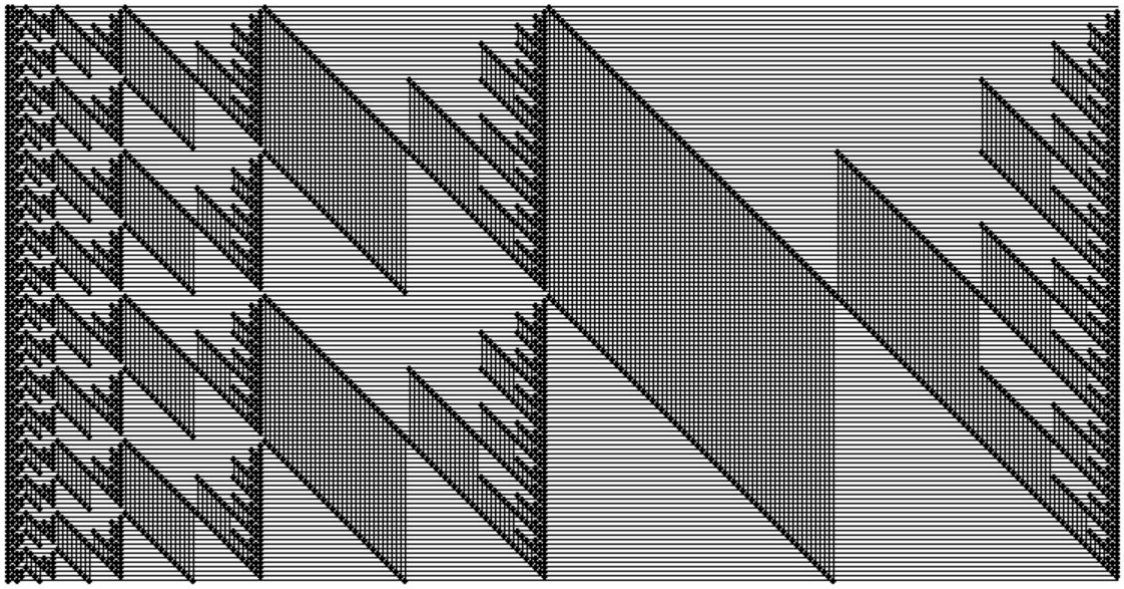

Figure 5: $n = 64$

Solution. Denote the depth of **Merger(n)** as $Dm(n)$.

$$Dm(n) = log_2 n + 1$$

Denote the depth of **Sorter(n)** as $Ds(n)$.

$$Ds(n) = \sum_{i=0}^{log_2 n} Dm(\frac{n}{2^i}) = O(log^2 n)$$

□

(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. We use Mathematical Induction here. Let $P(n)$ be the statement that the 2^n -input odd-even merging network can merge two sorted 01 -sequences with the same length into one correctly.

- **Basic Step.** Obviously, $P(1)$ is true.
- **Induction Hypothesis.** Assume $P(k)$ is true for some $k \geq 1$.
- **Proof of Induction Step.** For $P(k)$ is true, there're two sorted $0-1$ sequences A and B. For sequence A, denote the number of 0 with odd index as na_{odd} while the number of 0 with even index as na_{even}. We can find that $na_{odd} = na_{even}$ or $na_{odd} = na_{even} + 1$. Similarly, we can get $nb_{odd} = nb_{even}$ or $nb_{odd} = nb_{even} + 1$.
  After merging the sequence of odd elements and the sequence of even elements), we get a sequence A'. Denote the number of 0 with odd index as n_{odd} while the number of 0 with even index as n_{even}.There're 3 situations.:
  i. $n_{odd} = n_{even}$. A' looks like "$0 \cdots 0011 \cdots 1$".
  ii. $n_{odd} = n_{even} + 1$. A' looks like "$0 \cdots 0011 \cdots 1$".
  iii. $n_{odd} = n_{even} + 2$. A' looks like "$0 \cdots 0101 \cdots 1$". Furthermore, for the "10" subsequence, the element 0 has an even index while 1 has an odd index. After putting a comparator between a_{2i} and a_{2i+1} for $i = 1, 2, \ldots, 2^k - 1$, the "10" subsequence is changed into "01". And A' is sorted now.

□

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