Algorithm Design and Analysis (Fall 2022) Assignment 1

Xiangyuan Xue (521030910387)

1. (a) First, we consider the case that $a \le b^d$. Since $T(n) = aT(\frac{n}{b}) + n^d \log^w n$, we expand and simplify the expression as follow

$$\begin{aligned} T(n) &= O(n^d \log^w n) + aO\left((\frac{n}{b})^d \log^w(\frac{n}{b})\right) + \cdots + a^{\log_b n} O\left((\frac{n}{b^{log_b n}})^d \log^w(\frac{n}{b^{log_b n}})\right) \\ &= O(n^d \log^w n)\left[1 + (\frac{a}{b^d}) + \cdots + (\frac{a}{b^d})^{\log_b n}\right] \\ &= O(n^d \log^w n) \sum_{k=0}^{\log_b n} (\frac{a}{b^d})^k \end{aligned}$$

If $a < b^d$, then $\frac{a}{b^d} < 1$, therefore

$$T(n) = O(n^d \log^w n)\frac{1-(\frac{a}{b^d})^n}{1-\frac{a}{b^d}} = O(\lambda n^d \log^w n) = O(n^d \log^w n)$$

If $a = b^d$, then $\frac{a}{b^d} = 1$, therefore

$$T(n) = (\log_b n + 1)O(n^d \log^w n) = O(n^d \log^w n \log n) = O(n^d \log^{w+1} n)$$

(b) Then we consider the case that $a > b^d$, in other words we say $\log_b a > d$, therefore

$$\exists\, \delta > 0 : \log_b a = d + \delta$$

At the same time, it can be proved that

$$\forall\, \varepsilon > 0 : f(n) = n^d \log^w n = O(n^{d+\varepsilon})$$

We might as well let $\varepsilon_0 = \frac{\delta}{2}$, then

$$T(n) = aT(\frac{n}{b}) + O(n^{d+\varepsilon_0})$$

Since $d + \varepsilon_0 < \log_b a$, namely $a > b^{d+\varepsilon_0}$, according to original Master Theorem we have

$$T(n) = O(n^{\log_b a})$$

(c) By above discussion, following expression can be summarized

$$T(n) = \begin{cases} O(n^d \log^w n) & \text{if } a < b^d \\ O(n^{\log_b a}) & \text{if } a > b^d \\ O(n^d \log^{w+1} n) & \text{if } a = b^d \end{cases}$$

2. It is necessary to emphasize that the k here is large instead of being a small constant. Otherwise, by element-wise comparison, we can easily find an $O(n)$ algorithm.

   A divide-and-conquer algorithm solving this problem can be easily described. First, we divide k arrays into 2 groups. For each group, just keep dividing until there is only 1 array left. Then we merge the 2 arrays at one time and return the merged array to last recursion. Finally an array of kn elements will be fetched.

Algorithm 1 A k-way merge operation

```
Input: k sorted arrays $\{a_1^{(1)}, \dots, a_n^{(1)}\}, \dots, \{a_1^{(k)}, \dots, a_n^{(k)}\}$
Output: 1 sorted array of $kn$ elements
1: if $k = 1$ then
2:     return $a$
3: end if
4: Divide $k$ arrays into 2 groups $a^{(1)}, a^{(2)}, \dots, a^{(\frac{k}{2})}$ and $a^{(\frac{k}{2}+1)}, a^{(\frac{k}{2}+2)}, \dots, a^{(k)}$
5: Recursively merge $\frac{k}{2}$ arrays $a^{(1)}, a^{(2)}, \dots, a^{(\frac{k}{2})}$ into $u$
6: Recursively merge $\frac{k}{2}$ arrays $a^{(\frac{k}{2}+1)}, a^{(\frac{k}{2}+2)}, \dots, a^{(k)}$ into $v$
7: Merge $u$ and $v$ into $a'$
8: return $a'$
```

   Now let we talk about its time complexity. In every recursion, the problem is divided into 2 sub-problems, and each has half size of the orignal problem. In addition, an $O(kn)$ cost is needed for merging two arrays. Then we have a recursive relationship $T(k) = 2T(\frac{k}{2}) + O(kn)$. Just expand the expression as follow

$$\begin{aligned} T(k) &= O(kn) + 2O(\frac{kn}{2}) + \cdots + 2^{\log_2 k} O(\frac{kn}{2^{\log_2 k}}) \\ &= (1 + \log_2 k) O(kn) \\ &= O(nk \log k) \end{aligned}$$

   Therefore, the time complexity of this algorithm is $O(nk \log k)$. It is much more efficient than simply merging arrays one-by-one, whose time complexity is $O(k^2 n)$. The improvement comes from the equal division, which makes sub-problems at the same level have the same scale.

3. Consider a divide-and-conquer strategy which keeps reducing the scale of the problem. Our target is to find the median of $\{a_n\} \cup \{b_m\}$, namely the rank-k ($k = \frac{n+m}{2}$) element in $\{a_n\} \cup \{b_m\}$. Let we first find the rank-$\frac{k}{2}$ elements a_p and b_q respectively in $\{a_n\}$ and $\{b_m\}$. Make a comparison. If $a_p = b_q$, then it is the very element we are looking for. If $u < v$, then we drop the elements before a_p and after b_q. That is to say, we just need to find the rank-$\frac{k}{2}$ element in $\{a_{p+1}, a_{p+2}, \dots, a_n\} \cup \{b_1, b_2, \dots, b_{q-1}\}$ recursively, and vice versa. Note that if $\frac{k}{2}$ exceeds the size of $\{a_n\}$, the position will be changed to $p = n$ and $q = k - n$, and vice versa.

Algorithm 2 Median of two sorted lists

```
Input: Two sorted lists $\{a_n\}$ and $\{b_m\}$
Output: Median of $\{a_n\} \cup \{b_m\}$
 1: $k \leftarrow \frac{n+m}{2}$
 2: $(p, q) \leftarrow (\frac{k}{2}, \frac{k}{2})$
 3: if $\frac{k}{2} > n$ then
 4:     $(p, q) \leftarrow (n, k - n)$
 5: else if $\frac{k}{2} > m$ then
 6:     $(p, q) \leftarrow (k - m, m)$
 7: end if
 8: if $a_p = b_q$ then
 9:     return $a_p$
10: end if
11: if $a_p < b_q$ then
12:     $\{a'_{n'}\}, \{b'_{m'}\} \leftarrow \{a_{p+1}, a_{p+2}, \dots, a_n\}, \{b_1, b_2, \dots, b_{q-1}\}$
13: else if $a_p > b_q$ then
14:     $\{a'_{n'}\}, \{b'_{m'}\} \leftarrow \{a_1, a_2, \dots, a_{p-1}\}, \{b_{q+1}, b_{q+2}, \dots, b_m\}$
15: end if
16: return Recursively find the new median of $\{a'_{n'}\} \cup \{b'_{m'}\}$
```

Note that the list assignment in pseudo codes doesn't have to happen. As an alternative, the left and right boundary of the lists are passed as parameters, which guarantees the efficiency of the algorithm. The parity of n and m should be taken into consideration in practice, which is ignored here in pseudo codes.

Every recursion reduces the scale of the problem to nearly half. In the worst case $\frac{3}{4}$ of the elements will be dropped. Additional operation can be finished within $O(1)$ time, namely $T(k) = T(\frac{3}{4}k) + O(1)$ where $k = n + m$. Therefore, the time complexity of this algorithm is $O(\log k)$, which is much better than $O(n + m)$.

4. (a) When an integer triple (x, y, z) forms a good order, we have $y - x = z - y$, namely $x + z = 2y$. Since $2y$ is even, $x + z$ is also even, which means x and z are both even or both odd.

   (b) Consider the permutation $\{1, 3, 4, 2\}$. None of $(1, 3, 4)$, $(1, 3, 2)$, $(1, 4, 2)$, $(3, 4, 2)$ forms a good order. By definition, it is out-of-order.

   (c) Assume that $x = 2a - 1$, $y = 2b - 1$, $z = 2c - 1$ where $a, b, c \in \mathbb{N}$. Since (x, y, z) forms a good order, we have $(2b - 1) - (2a - 1) = (2c - 1) - (2b - 1)$, namely $b - a = c - b$, which is equivalent to $\frac{y+1}{2} - \frac{x+1}{2} = \frac{z+1}{2} - \frac{y+1}{2}$, namely $(\frac{x+1}{2}, \frac{y+1}{2}, \frac{z+1}{2})$ forms a good order.

   (d) Divide $\{1, 2, \dots, n\}$ by parity. Odd integers are placed on the left half and even ones are placed on the right half. According to conclusion in 4a, no good order will be formed across two halves. For the left half, make a transformation $f(n) = \frac{n+1}{2}$, and for the right half it is $g(n) = \frac{n}{2}$. Then both halves become new permutations of

$\{1, 2, \dots, \frac{n}{2}\}$. Repeat the operations above recursively for both halves, then make an inverse transformation. After recursion, an out-of-order permutation of $\{1, 2, \dots, n\}$ will be constructed.

Algorithm 3 An out-of-order permutation

```
Input: An original permutation $\{1, 2, \dots, n\}$
Output: An out-of-order permutation $\{a_1, a_2, \dots, a_n\}$
1: if $n = 1$ then
2:     return
3: end if
4: $\{a_1, a_2, \dots, a_{\frac{n}{2}}\} \leftarrow \{1, 3, 5, \dots\}$
5: $\{a_{\frac{n}{2}+1}, a_{\frac{n}{2}+2}, \dots, a_n\} \leftarrow \{2, 4, 6, \dots\}$
6: Transformation $f(n) = \frac{n+1}{2}$ on $\{a_1, a_2, \dots, a_{\frac{n}{2}}\}$
7: Recursively construct on $\{a_1, a_2, \dots, a_{\frac{n}{2}}\}$
8: Inverse transformation $f^{-1}(n) = 2n - 1$ on $\{a_1, a_2, \dots, a_{\frac{n}{2}}\}$
9: Transformation $g(n) = \frac{n}{2}$ on $\{a_{\frac{n}{2}+1}, a_{\frac{n}{2}+2}, \dots, a_n\}$
10: Recursively construct on $\{a_{\frac{n}{2}+1}, a_{\frac{n}{2}+2}, \dots, a_n\}$
11: Inverse transformation $g^{-1}(n) = 2n$ on $\{a_{\frac{n}{2}+1}, a_{\frac{n}{2}+2}, \dots, a_n\}$
12: return
```

Consider the complexity of this algorithm, the recursive relationship can be written as $T(n) = 2T(\frac{n}{2}) + O(n)$. According to Master Theorem, we can immediately find out its time complexity is $O(n \log n)$.

Actually, this permutation is essentially the same as *bit-reversal permutation.*[1] Using bit-wise operation and recursion, the permutation can be constructed within $O(n)$ time. Such method is commonly used in algorithms like *FFT.*[2] However, this is beyond the scope of this homework.

5. (a) According to the description, pseudo codes of Even-Paz algorithm are written as follow.[3]

Algorithm 4 Even-Paz algorithm

```
Input: Value density functions $f_1, f_2, \dots, f_n$ and current interval $[l, r]$
Output: A proportional allocation $\{I_1, I_2, \dots, I_n\}$
1: if $n = 1$ then
2:     return $\{[l, r]\}$
3: end if
4: for $i = 1 \to n$ do
5:     Calculate $i$'s half-half point $x_i$
6: end for
7: Sort $x_1, x_2, \dots, x_n$ in increasing order together with indices
8: $x^* \leftarrow x_{\lfloor \frac{n}{2} \rfloor + 1}$
9: $L \leftarrow$ Recursively find allocation for $1, 2, \dots, \lfloor \frac{n}{2} \rfloor$ on $[l, x^*]$
10: $R \leftarrow$ Recursively find allocation for $\lfloor \frac{n}{2} \rfloor + 1, \lfloor \frac{n}{2} \rfloor + 2, \dots, n$ on $[x^*, r]$
11: return $L \cup R$
```

(b) In every recursion, the problem is divided into 2 sub-problems. Each of the sub-problems has half size of the original problem. Before recursion, it takes $O(n)$ to compute n half-half points and $O(n \log n)$ to sort them. Then the recursive relationship of time complexity can be written as $T(n) = 2T(\frac{n}{2}) + O(n \log n)$. With generalized Master Theorem in 1, its time complexity is actually $O(n \log^2 n)$.

Obviously, we can optimize the algorithm by eliminating a $\log n$. Note that sorting n half-half points is not necessary. To achieve the same effect, we can use selection algorithm to find the rank-$\lfloor \frac{n}{2} \rfloor + 1$ half-half point, and division will be naturally completed at the same time. This takes only $O(n)$ time in total. Then we have $T(n) = 2T(\frac{n}{2}) + O(n)$, indicating that the time complexity is $O(n \log n)$.[4]

(c) In each division $[l, r] = [l, x^*] \cup [x^*, r]$, x^* divides interval $[l, r]$ into two sub-intervals (assume x^* itself falls into the right part). The left one includes exactly $\lfloor \frac{n}{2} \rfloor$ half-half points, and the right one includes exactly $\lceil \frac{n}{2} \rceil$ half-half points.

For each child i, we might as well assume x_i falls into the left interval $[l, x^*]$. According to the definition of half-half points and since value density function f_i is non-negative, we have the following inequality

$$\int_l^{x^*} f_i(x)\,\mathrm{d}x \geq \int_l^{x_i} f_i(x)\,\mathrm{d}x = \frac{1}{n}\lfloor \frac{n}{2} \rfloor \int_l^r f_i(x)\,\mathrm{d}x$$

If x_i falls into the right interval $[x^*, r]$, there is a symmetric inequality

$$\int_{x^*}^{r} f_i(x)\,\mathrm{d}x \geq \int_{x_i}^{r} f_i(x)\,\mathrm{d}x = \frac{1}{n}\lceil \frac{n}{2} \rceil \int_l^r f_i(x)\,\mathrm{d}x$$

For the whole recursive procedure, repeatedly use the inequality above and we have

$$\begin{aligned}\int_{l_i}^{r_i} f_i(x)\,\mathrm{d}x &\geq \prod_{k=0}^{K-1} \left[\frac{1}{\frac{n}{2^k}} \left(\frac{\frac{n}{2^k}}{2} \right) \right] \int_0^1 f_i(x)\,\mathrm{d}x \\ &= (\frac{1}{2})^K \int_0^1 f_i(x)\,\mathrm{d}x\end{aligned}$$

Note that K is the maximal depth of recursion, constrained by $K = \lfloor \log_2 n \rfloor$, therefore

$$\int_{l_i}^{r_i} f_i(x)\,\mathrm{d}x \geq \frac{1}{2^{\log_2 n}} \int_0^1 f_i(x)\,\mathrm{d}x = \frac{1}{n} \int_0^1 f_i(x)\,\mathrm{d}x$$

Since i is arbitrary, we have $\int_{I_i} f_i(x)\,\mathrm{d}x \geq \frac{1}{n}\int_0^1 f_i(x)\,\mathrm{d}x$ for each $i = 1, 2, \ldots, n$, namely the allocation $(I_1, I_2, \ldots, I_n)$ is proportional.

6. It takes me over 8 hours to finish this assignment. The work includes thinking, designing, proving and typing. I don't think it difficult in thinking, so I prefer a 3/5 score for its difficulty. However, a large amount of time is used to consider how to write down my proof formally and type LaTeX. I hope the situation will be improved in future assignments. I had no collaborators, but referred to several websites and articles. All of them have been listed below.

References

[1] Wikipedia. Bit-reversal permutation.
https://en.wikipedia.org/wiki/Bit-reversal_permutation.

[2] OI-Wiki. Fast Fourier Transform. http://oi-wiki.com/math/poly/fft.

[3] Wikipedia. Even–Paz protocol. https://en.wikipedia.org/wiki/Even-Paz_protocol

[4] Even S, Paz A. A note on cake cutting. Discrete Applied Mathematics. 7(3): 285.