SUN Yilin
520030910361

# Homework 1

Algorithms
March 11, 2022

## 1

Without loss of generality, assume $n = b^{2k}$ for some $k \in \mathbb{N}$, then

$$T(n) = aT(\frac{n}{b}) + n^d \log^w n \tag{1}$$
$$= a(aT(\frac{n}{b^2}) + (\frac{n}{b})^d \log^w(\frac{n}{b})) + n^d \log^w n \tag{2}$$
$$= a^2 T(\frac{n}{b^2}) + n^d \frac{a}{b^d} \log^w(\frac{n}{b}) + n^d \log^w n \tag{3}$$
$$= a^3 T(\frac{n}{b^3}) + n^d((\frac{a}{b^d})^2 \log^w \frac{n}{b^2} + \frac{a}{b^d} \log^w(\frac{n}{b}) + \log^w n) \tag{4}$$
$$= \dots \tag{5}$$
$$= a^{\log_b n} T(1) + n^d \sum_{i=0}^{(\log_b n)-1} (\frac{a}{b^d})^i \log^w \frac{n}{b^i} \tag{6}$$

Since $T(n) = 1$ for $n < b$, $T(1) = 1$. So

$$T(n) = n^{\log_b a} + n^d \sum_{i=0}^{(\log_b n)-1} (\frac{a}{b^d})^i \log^w \frac{n}{b^i} \tag{7}$$

And since $w \geq 0$, $b > 1$, it is obvious that $\log^w \frac{n}{b^i} \leq \log^w n$. So

$$T(n) \leq n^{\log_b a} + n^d \log^w n \sum_{i=0}^{(\log_b n)-1} (\frac{a}{b^d})^i \tag{8}$$

If $a < b^d$, then $\sum_{i=0}^{\log_b n-1}(\frac{a}{b^d})^i = O(1)$, so

$$T(n) \leq n^{\log_b a} + n^d \log^w n O(1) \tag{9}$$
$$T(n) \leq n^{\log_b a} + O(n^d \log^w n) \tag{10}$$
$$T(n) = O(n^d \log^w n) \tag{11}$$

If $a = b^d$, then $\sum_{i=0}^{\log_b n-1}(\frac{a}{b^d})^i = \log_b n$, so

$$T(n) \leq n^d + n^d \log^w n \log_b n \tag{12}$$
$$T(n) \leq n^d + n^d \log^w n \frac{\log n}{\log b} \tag{13}$$
$$T(n) \leq n^d + n^d \log^{w+1} n / \log b \tag{14}$$
$$T(n) = O(n^d \log^{w+1} n) \tag{15}$$

If $a > b^d$, note that by our assumption that $n = b^{2k}$ for some $k \in \mathbb{N}$, $\log_b \sqrt{n}$ is an integer. Then

$$\sum_{i=0}^{(\log_b n)-1} (\frac{a}{b^d})^i \log^w \frac{n}{b^i} = \sum_{i=0}^{\log_b \sqrt{n}-1} (\frac{a}{b^d})^i \log^w \frac{n}{b^i} + \sum_{i=\log_b \sqrt{n}}^{\log_b n-1} (\frac{a}{b^d})^i \log^w \frac{n}{b^i} \tag{16}$$

$$\leq \sum_{i=0}^{\log_b \sqrt{n}-1} (\frac{a}{b^d})^i \log^w n + \sum_{i=\log_b \sqrt{n}}^{\log_b n-1} (\frac{a}{b^d})^i \log^w \sqrt{n} \tag{17}$$

$$\leq O((\frac{a}{b^d})^{\log_b \sqrt{n}}) \log^w n + O((\frac{a}{b^d})^{\log_b \sqrt{n}} \log_b \sqrt{n}) \log^w \sqrt{n} \tag{18}$$

$$\leq O(\frac{\sqrt{n}^{\log_b a}}{\sqrt{n}^d}) \log^w n + O(\frac{\sqrt{n}^{\log_b a}}{\sqrt{n}^d} \log_b \sqrt{n}) \log^w \sqrt{n} \tag{19}$$

So

$$T(n) \leq n^{\log_b a} + n^d \left[ O(\frac{\sqrt{n}^{\log_b a}}{\sqrt{n}^d}) \log^w n + O(\frac{\sqrt{n}^{\log_b a}}{\sqrt{n}^d} \log_b \sqrt{n}) \log^w \sqrt{n} \right] \tag{20}$$

$$\leq n^{\log_b a} + O(n^{\frac{\log_b a + d}{2}}) \left( \log^w n + \log_b \sqrt{n} \log^w \sqrt{n} \right) \tag{21}$$

Since $a > b^d$

$$T(n) \leq n^{\log_b a} \tag{22}$$

$$T(n) = O(n^{\log_b a}) \tag{23}$$

# 2

## 2.1 Algorithm Description

**Algorithm 1** The One Third Merge Sort

```
procedure OneThirdMergeSort(a[1, ..., n]):
    if n=1 then return a
    b ← OneThirdMergeSort(a[1, ..., ⌈n/3⌉])
    c ← OneThirdMergeSort(a[⌈n/3⌉ + 1, ..., n])
    return Merge(b, c)        ▷ The Merge function is what we have defined in class
```

Here the Merge function is the same as what we have learnt in class, so I do not write it again.

## 2.2 Time Complexity Analysis

As we have learnt in class, the Merge function is $O(n)$. We have

$$\begin{aligned} T(n) &= T(\frac{2}{3}n) + T(\frac{1}{3}n) + O(n) && (24)\\ &\leq T(\frac{2}{3}n) + T(\frac{1}{3}n) + cn \quad \text{for some c} && (25)\\ &\leq T(\frac{4n}{3^2}) + 2T(\frac{2n}{3^2}) + T(\frac{n}{3^2}) + 2cn && (26)\\ &\leq \dots && (27)\\ &\leq \sum_{k=0}^{m} C_m^k T(\frac{2^k}{3^m}n) + cmn && (28) \end{aligned}$$

When $m = \lceil \log_{\frac{3}{2}} n \rceil$, we have

$$T(n) \leq \sum_{k=0}^{m} C_m^k T(\frac{2^k}{3^m}n) + cn \log_{\frac{3}{2}} n \tag{29}$$

Note that there are at most $n$ elements in our array, hence the term $\sum_{k=0}^{m} C_m^k T(\frac{2^k}{3^m}n)$ consists of $n$ terms at most. So when $m = \lceil \log_{\frac{3}{2}} n \rceil$,

$$\sum_{k=0}^{m} C_m^k T(\frac{2^k}{3^m}n) \leq nT(1) \tag{30}$$

So

$$T(n) \leq nT(1) + cn \log_{\frac{3}{2}} n \tag{31}$$

$$T(n) = O(n) + O(n \log_{\frac{3}{2}} n) \tag{32}$$

$$T(n) = O(n \log n) \tag{33}$$

# 3

In this problem our assumption is that we already have a **sort** function which can sort any given set on which an ordering relation is well defined in $O(n \log n)$ time. Also withour loss of generality assume that this function sorts the elements in a descending order.

## 3.1

Basically the idea is that we sort the two sets together with $O(n \log n)$ time and scan the result once with $O(n)$ time to get the answer. See the algorithm below for more detail.

**Algorithm 2** Count Pairs(One Dimension)

```
1:  procedure COUNT PAIRS(A, B): ▷ A,B are sets where ordering relation is well defined
2:      c ← sort(A ∪ B)          ▷ Sort two sets together and save the elements in an array
3:      b̂ ← |B|                  ▷ Can get |B| from the sorting subroutine
4:      ans ← 0                  ▷ Initialization of the result
5:      while c is not empty do
6:          a ← c.pop_front
7:          if a ∈ A then
8:              ans ← ans + b̂
9:          if a ∈ B then
10:             b̂ ← b̂ − 1
11:     return ans
```

The correctness of this algorithm lies in that after sorting we can count the number of pairs for each element in $A$ by scanning the entire array only once. Since the sort is $O(n \log n)$ and the scanning is $O(n)$, the total time complexity is $O(n \log n)$

## 3.2

**Algorithm 3** Count Pairs(Two Dimension)

1: **procedure** COUNT PAIRS($S$): ▷ In this case $S = A \cup B$ contain elements with dim=2
2: Sort $S$ by descending order of one of its coordinates denoted by $x$. Denote the other coordinate by $y$.
3: **if** $S \cap A = \emptyset \vee S \cap B = \emptyset$ **then return** 0 ▷ No pairs inside $S$
4: find $t$ such that half of the points in $S$ is with coordinates $x < t$
5: compute $S_{small} = \{(x_i, y_j) | x_i \leq t\}$ and $S_{large} = \{(x_i, y_j) | x_i > t\}$
6: $ans \leftarrow$Count Pairs($S_{large}$)+Count Pairs($S_{small}$) ▷ Recursion process
7: compute $A' = \{y | (x, y) \in A \wedge (x, y) \in S_{large}\}$ and $B' = \{y | (x, y) \in B \wedge (x, y) \in S_{small}\}$
8: $ans \leftarrow ans$+Count Pairs(One Dimension)$(A', B')$ ▷ reduce to dim=1
9: **return** $ans$

At line 6 we use a Divide and Couquer approach and at lint 8 we use the algorithm defined in the previous problem. Then the time complexity can be expressed as

$$T(n) = 2T(\frac{n}{2}) + O(n \log n) \tag{34}$$

This is exactly the case where $a = b = 2, d = w = 1$ in the generalized Master Theorem proved in problem 1 in this assignment. So we know

$$T(n) = O(n \log^2 n) \tag{35}$$

Which satisfies that

$$T(n) = o(n^{1.1}) \tag{36}$$

## 3.3

The idea is that when treating cases with dimension $d+1$, we apply the Divide and Conqure approach and reduces it to the case with dimension $d$.

**Algorithm 4** CountPairs

1: **procedure** CountPairs($S, d$): ▷ $S = A \cup B$ and $d$ is the dimension
2: Sort $S$ by descending order of one of its coordinates denoted by $x$.
3: **if** $S \cap A = \emptyset \vee S \cap B = \emptyset$ **then return** 0 ▷ No pairs inside $S$
4: find $t$ such that half of the points in $S$ is with coordinates $x < t$
5: compute $S_{small} = \{\boldsymbol{a} = (x_i, a_2, a_3, \ldots, a_d) \in S | x_i \leq t\}$
6: compure $S_{large} = \{\boldsymbol{a} = (x_i, a_2, a_3, \ldots, a_d) \in S | x_i > t\}$
7: $ans \leftarrow$CountPairs($S_{large}, d$)+Count Pairs($S_{small}, d$) ▷ Recursion process
8: compute $A' = \{(a_2, a_3, \ldots, a_n) | \boldsymbol{a} \in A \wedge \boldsymbol{a} \in S_{large}\}$
9: compute $B' = \{(a_2, a_3, \ldots, a_n) | \boldsymbol{a} \in B \wedge \boldsymbol{a} \in S_{small}\}$ ▷ generate sets with dim=$d-1$
10: $ans \leftarrow ans$+CountPairs($A' \cup B', d-1$) ▷ reduce to dim=$d-1$
11: **return** $ans$

We assume the running time is $O(n \log^d n)$ for the generalized case and prove by induction. Clearly with $d = 1, 2$ that equation holds by our proof above. Assume that equation holds when $d = k$, i.e. the running time is $O(n \log^k n)$ when $d = k$. Now consider the case where $d = k + 1$.

At line 7 we divide the original problem to two subproblem with same $d$ but a smaller size, at line 10 the subproblem with dimension $d - 1 = k$ is $O(n \log^k n)$ by our assumption step. So at dimension $d$ we have

$$T(n) = 2T(\frac{n}{2}) + O(n \log^k n) \tag{37}$$

$$\tag{38}$$

And here we need to use the generalized Master Theorem again. Likewise, by generalized Master Theorem we can directly get

$$T(n) = O(n \log^{k+1} n) \tag{39}$$

So that equation still holds with $d = k + 1$. So our assumption is true and the running time is indeed $O(n \log^d n)$

# 4

## 4.1

Since $x$ is the median of $\frac{n}{3}$ numbers and the numbers are distinctive to each other, there are exactly $\frac{n}{6}$ numbers smaller than $x$ and exactly $\frac{n}{6}$ numbers larger than $x$ for the same reason. Because those numbers are also medians of each group, exactly another number is smaller(larger) than each of them in their groups respectively. So totally there are $\frac{n}{6} * 2 = \frac{n}{3}$ numbers in $a$ smaller than $x$ and $\frac{n}{3}$ numbers in $a$ larger than $x$ as well.

## 4.2

### 4.2.1 Algorithm Design

Here we want to line up the elements in groups of $m$ and find the $k$-th largest element. And assume that we have already got a function which can return the median of a group of numbers with size $m$. Denote that function by $Median(a)$ where $a$ is an array.
Firstly we define an algorithm which can generate all the medians we want.

**Algorithm 5** Generate Medians

```
1: procedure GENERATE MEDIANS(a[1, ..., n], m):
2:     i ← 0
3:     while i + m ≤ n do
4:         x ← Median(a[i + 1, i + 2, ..., i + m])
5:         b.push_back(x)
6:         i ← i + m
7:     return b                ▷ b is the array containing all the Medians we want
```

Then we can define the actual algorithm building on that.

**Algorithm 6** Median of Medians

**procedure** MEDIAN OF MEDIANS($\boldsymbol{a}[1,\dots,n], m, k$):
  **if** n=1 **then**
    **return** a[n]
  $\boldsymbol{b} \leftarrow GenerateMedians(\boldsymbol{a}, m)$
  $x \leftarrow MedianofMedians(\boldsymbol{b}, m, \frac{m}{2n})$ ▷ x is the median of array $\boldsymbol{b}$
  $l \leftarrow \boldsymbol{a}$.length, $i \leftarrow 1$
  **while** $i \leq l$ **do**
    **if** $\boldsymbol{a}[i] \leq x$ **then** $\boldsymbol{c}$.push_back($\boldsymbol{a}[i]$)
    **if** $\boldsymbol{a}[i] > x$ **then** $\boldsymbol{d}$.push_back($\boldsymbol{a}[i]$)
    $i \leftarrow i + 1$
  **if** $\boldsymbol{d}$.length $\leq k-1$ **then return** $MedianofMedians(\boldsymbol{c}, m, k - \boldsymbol{d}.length)$
  **if** $\boldsymbol{d}$.length $> k-1$ **then return** $MedianofMedians(\boldsymbol{d}, m, k)$

### 4.2.2 Time Complexity Analysis

Suppose we line up elements in groups of $m$, $m \in \mathbb{N}, m \geq 2$. Then by our analysis in part 4.1 we can see there are $\frac{n}{m}$ groups and hence $\frac{n}{2m}$ groups contain $\left[\frac{m+1}{2}\right]$ elements larger(smaller) than the median we calculated. Then after each iteration we can discard at least $n - \frac{n}{2m}\left[\frac{m+1}{2}\right]$ elements and reduce our problem to a same problem with scale $\frac{3m-1}{4m}n$ if $m$ is odd and $\frac{3n}{4}$ if $m$ is even. Hence the Time Complexity can be expressed as

$$T(n) = \begin{cases} T(\frac{1}{m}n) + T(\frac{3m-1}{4m}n) + \frac{n}{m}O(1), & m \quad \text{is odd} \\ T(\frac{1}{m}n) + T(\frac{3n}{4}) + \frac{n}{m}O(1), & m \quad \text{is even} \end{cases}$$

If $m$ is not so large that the time of finding the medians of $m$ elements can not be treated as a constant, we can safely say for some $c > 0$ large enough

$$T(n) \leq \begin{cases} T(\frac{1}{m}n) + T(\frac{3m-1}{4m}n) + c\frac{n}{m}, & m \quad \text{is odd} \\ T(\frac{1}{m}n) + T(\frac{3n}{4}) + c\frac{n}{m}, & m \quad \text{is even} \end{cases}$$

Now we discuss the Time Complexity with different values of $m$.
If $m \geq 5$ and $m$ is odd then $\frac{1}{m} + \frac{3m-1}{4m} = \frac{3m+3}{4m} < 1$. Assume $T(n) \leq \alpha n$ for some $\alpha > 0$, it is easy to see that is true when $n = 1$. Suppose that is true for $T(\frac{1}{m}n)$ and $T(\frac{3m-1}{4m}n)$, then

$$T(n) \leq \alpha\left(\frac{1}{m} + \frac{3m-1}{4m}\right)n + c\frac{n}{m} \tag{40}$$

With an $\alpha$ large enough

$$T(n) \leq \alpha n \tag{41}$$

So $T(n) = O(n)$ with $m \geq 5$, $m$ is odd and $m$ not too large.
A similar argument shows $T(n) = O(n)$ with $m \geq 5$ and $m$ is even. Actually when $m$ is even the inequality is simpler.
Now we consider the cases where $m \leq 4$.
Specifically, with $m = 3$

$$T(n) = T(\frac{1}{3}n) + T(\frac{2}{3}n) + O(\frac{n}{3}) \tag{42}$$

With analysis same as problem 2, we can say $T(n) = O(n \log n)$ with $m = 3$. Which means that when $m$ is too small we cannot get an algorithm with speed of linearity.
With $m = 4$

$$T(n) = T(\frac{1}{4}n) + T(\frac{3}{4}n) + O(\frac{n}{4}) \tag{43}$$

With $m = 2$

$$T(n) = T(\frac{1}{2}n) + T(\frac{5}{8}n) + O(\frac{n}{2}) \tag{44}$$

Similarly we can get $T(n) = O(n \log n)$ where it is also not linearity.
To sum up,

$$T(n) = \begin{cases} O(n), m \geq 5 \\ O(n \log n), m \leq 4 \end{cases}$$

# 5

## Comments from a student's perspective

It takes 18 hours perhaps(Including discussion and thinking). But I do not remember the exact time I spent on discussion. Actually I discussed this assignment long after I have finished this pdf file.
As for the difficuities of each questions, I would say that Q1 and Q2 get a 3 because they only involve the ideas that have been thoroughly discussed in class plus some tricks from Mathematical Analysis.
Q3 definitily gets a 5 because it is the most diffcult one in this assignment. Actually most of the time it took me to finish this assignment is spent on this one.
Q4 could have got a 5 but my overhearing the idea from the other class reduced its fun. So my recommendation is a 4 because this algorithm is intrinsic brilliant. My suggestion is that our professor also teach it in class.

## Acknowledgements

When proving the case where $a > b^d$ of problem 1, I got the idea of dividing the whole sequence into two parts and analyze them separatly from Liyuan MAO. He illustrate the basic ideas for me and I then wrote a formal proof myself.
As for the algorithm MedianofMedians, some of my classmates from another class have learnt this algorithm. When I discuss this problem with them they taught me the idea of lining up the elements in group of five and I then thought this algorithm might be different with different group size. I then formalized the algorithm(their lecture slides did not illustrate the formal version) and analyzed its complexity with different group size.