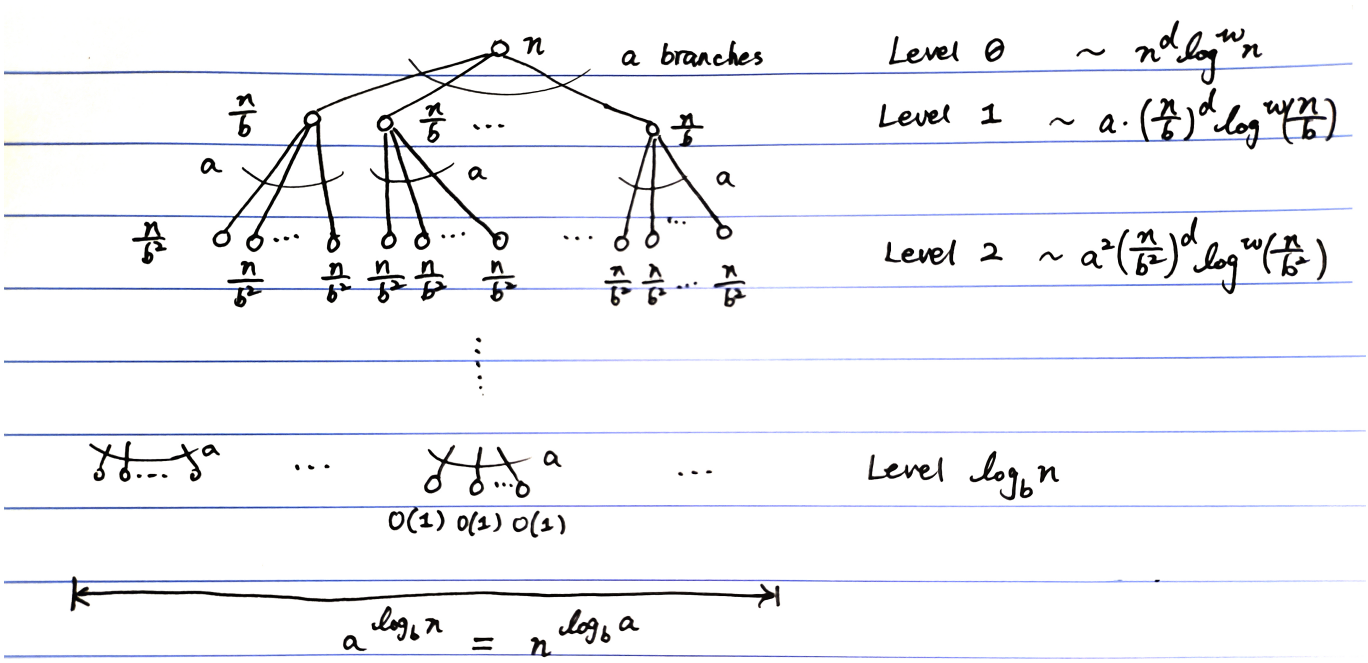
Algorithm Homework 01

Qiu Yihang

March 2022

1 Problem 01

Proof. We can plot the process of dividing and conquering as follows.



From the recursion tree above, we know for all a^k subproblems in level k, the time consumption is at most $a^k \left(\frac{n}{b^k}\right)^d \log^w \left(\frac{n}{b^k}\right) = a^k \left(\frac{n}{b^k}\right)^d (\log^w n - k \log^w b)$.

Let $q = \frac{a}{b^d}$. Thus, the total time consumption is

$$\begin{aligned} T(n) &= \sum_{k=0}^{\log_b n} a^k \left(\frac{n}{b^k}\right)^d (\log^w n - k \log^w b) \\ &= n^d \log^w n \sum_{k=0}^{\log_b n} \left(\frac{a}{b^d}\right)^k - n^d \log^w b \sum_{k=0}^{\log_b n} k \left(\frac{a}{b^d}\right)^k \\ &= n^d \left(\log^w n \sum_{k=0}^{\log_b n} q^k - \log^w b \sum_{k=0}^{\log_b n} k q^k \right) \end{aligned}$$

CASE 1. When $q = \frac{a}{b^d} < 1$, i.e. $a < b^d$. Let $K = \log_b n$.

$$\begin{aligned} T(n) &= n^d \left(\log^w n \ \frac{1 - q^{K+1}}{1 - q} - \log^w b \ \frac{Kq^{K+2} - (K+1)q^{K+1} + q}{(q-1)^2} \right) \\ &= n^d \left(O\left(\log^w n\right) - O(1) \right) \\ &= O\left(n^d \log^w n\right) \end{aligned}$$

CASE 2. When $q = \frac{a}{b^d} = 1$, i.e. $a = b^d$,

$$\begin{aligned} T(n) &= n^d \left(\log^w n \log_b n - O\left((\log_b n)^2 \right) \right) = n^d \left(O\left(\log^{w+1} n\right) - O\left(\log_b n\right)^2 \right) \\ &= n^d \left(O\left(\log^{w+1} n\right) \right) \\ &= O\left(n^d \log^{w+1} n\right) \end{aligned}$$

CASE 3. When $q = \frac{a}{b^d} > 1$, i.e. $a > b^d$. Let $K = \log_b n$.

$$\begin{aligned} T(n) &= n^d \left(\log^w n \ \frac{1 - q^{K+1}}{1 - q} - \log^w b \ \frac{Kq^{K+2} - (K+1)q^{K+1} + q}{(q-1)^2} \right) \\ &= n^d \left(O\left(q^K \log^w n\right) - O(Kq^K) \right) \\ &= O\left(n^d \left(\frac{a}{b^d} \right)^{\log_b n} \left(\log^w n - \log_b n \right) \right) \\ &= O\left(n^d \frac{a^{\log_b n}}{n^d} \right) \\ &= O\left(n^{\log_b a}\right) \end{aligned}$$

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

2 Problem 02

Solution. Similar to the process of **Merge Sort**, the process of **One Third Merge Sort Algorithm** is as follows.

Algorithm 1: One-Third Merge Sort Algorithm

```
Function One-Third Merge Sort (a[1 : n])
    mid ← ⌈n/3⌉;
    left ← OneThirdMergeSort (a[1 : mid]);
    right ← OneThirdMergeSort (a[mid + 1 : n]);
    Return: Merge(left, right)
end

Function Merge(left, right)
    i ← 1, j ← 1, k ← 1, n ← Length(left), m ← Length(right);
    while (i ≤ n and j ≤ m) do
        if left[i] ≤ right[j] then
            a[k] ← left[i];
            i ← i + 1;
        else
            a[k] ← right[j];
            j ← j + 1;
        end
        k ← k + 1;
    end
    while i ≤ n do
        a[k] ← left[i];
        i ← i + 1, k ← k + 1;
    end
    while j ≤ m do
        a[k] ← right[j];
        j ← j + 1, k ← k + 1;
    end
    Return: a[1 : n + m]
end
```

Now we analyze the time complexity of **One-Third Merge Sort Algorithm**. Let the time consumption be $T(n)$.

Obvious the time of **Merge** is $O(n)$. Then we have

$$\begin{aligned} T(n) &= T\left(\frac{n}{3}\right) + T\left(\frac{2n}{3}\right) + O(n) \\ &\leq T\left(\frac{n}{3}\right) + T\left(\frac{2n}{3}\right) + cn \\ &= T\left(\frac{4n}{3^2}\right) + 2T\left(\frac{2n}{3^2}\right) + T\left(\frac{n}{3^2}\right) + c\frac{n}{3} + c\frac{2n}{3} + cn \\ &= T\left(\frac{4n}{3^2}\right) + 2T\left(\frac{2n}{3^2}\right) + T\left(\frac{n}{3^2}\right) + 2cn \end{aligned}$$

$$= T\left(\frac{8n}{3^3}\right) + 3T\left(\frac{4n}{3^3}\right) + 3T\left(\frac{2n}{3^3}\right) + T\left(\frac{n}{3^3}\right) + 3cn$$
$$= ...$$
$$= \sum_{i=0}^{k} \binom{i}{k} T\left(\frac{2^i n}{3^k}\right) + kcn.$$

Suppose $N = \lceil \log_{2/3} n \rceil$. Then we have $k \leq N$, $\frac{2^k n}{3^N} \sim O(1)$. Obvious $T(1) = O(1)$.

$$T(n) \leq \sum_{i=0}^{N} \binom{i}{N} T(1) + Ncn = O(2^N) + O(Nn)$$
$$= O(n) + O(n \log_{2/3} n) = O(n) + O(n \log n)$$
$$= O(n \log n).$$

Thus, the time complexity of **One-Third Merge Sort Algorithm** is $O(n \log n)$. ■

3 Problem 03

3.1 $d = 1$

Solution. Based on sort algorithms, we design a $O(n \log n)$ algorithm as follows.

Algorithm 2: Unidimensional Pair Counting Algorithm

Function *Uni-Dim Pair Count* (A, B)
  $S \leftarrow sort\,(A \cup B)$;
    // $sort(\cdot)$ returns the array sorted from the largest to the smallest.
  $count \leftarrow 0, b \leftarrow |B|$;
  for $i = 1 \rightarrow n$ **do**
    if $S[i] \in A$ **then** $count \leftarrow count + b$ **else** $b \leftarrow b - 1$;
  end
  Return: *count*
end

The time complexity of *sort* is $O(n \log n)$ while the remaining part of the algorithm takes $O(n)$ time. Thus, the time complexity of the algorithm above is $O(n \log n)$. ■

3.2 $d = 2$

Solution. Inspired by the algorithm counting inverted-ordered pairs we discussed in class, a feasible algorithm to count two-dimensional $(\boldsymbol{a}, \boldsymbol{b}) \in (A, B)$ s.t. a is greater than b is as follows.

Let $\boldsymbol{a} = (a_1, a_2)$ for $\boldsymbol{a} \in A$, $\boldsymbol{b} = (b_1, b_2)$ for $\boldsymbol{b} \in B$. Use $\boldsymbol{a} > \boldsymbol{b}$ to denote $\boldsymbol{a}$ is greater than $\boldsymbol{b}$.

Each time, we can find a suitable mid to divide arrays A, B into four parts, $A_{left}, A_{right}, B_{left}$, B_{right}, where $A_{left} = \left\{\boldsymbol{a} \in A \middle| a_1 < mid\right\}, A_{right} = \left\{\boldsymbol{a} \in A \middle| a_1 \geq mid\right\}, B_{left} = \left\{\boldsymbol{b} \in B \middle| b_1 < mid\right\}$, $B_{right} = \left\{\boldsymbol{b} \in B \middle| b_1 \geq mid\right\}$.

Let $N_{l,l}, N_{r,r}, N_{r,l}, N_{l,r}$ be the number of pairs between A_{left} and B_{left}, between A_{right} and B_{right}, between A_{right} and B_{left}, and between A_{left} and B_{right} respectively. Obvious we have $N_{r,l} = 0$ since $\forall a \in A_{right}, \forall b \in B_{left}, a_1 < mid \leq b_1$. Meanwhile, since $\forall a \in A_{left}, \forall b \in B_{right}, a_1 \geq mid > b_1$, we know $N_{l,r}$ is also the number of pairs between $A' = \left\{a_2 \middle| \boldsymbol{a} = (a_1, a_2), \boldsymbol{a} \in A_{left}\right\}$ and $B' = \left\{b_2 \middle| \boldsymbol{b} = (b_1, b_2), \boldsymbol{b} \in B_{right}\right\}$, i.e. $N_{l,r} = Uni\ Dim\ Pair\ Count(A', B')$.

Thus, we can divide the problem into three subproblems since the total number of pairs between A and B is $N = N_{l,l} + N_{l,r} + N_{r,l} + N_{r,r} = N_{l,l} + N_{r,r} + Uni\ Dim\ Pair\ Count(A', B')$.

The algorithm is as follows.

Algorithm 3: Two-Dimensional Pair Counting Algorithm

Function *Two-Dim Pair Count* (A, B)
  if *A is empty or B is empty* **then** **Return:** 0;

  $S \leftarrow sort\,(A \cup B)$;
  // $sort(\cdot)$ `returns the array sorted by the first element from the largest to the smallest.`
  $mid \leftarrow$ the first element of S[Length$(S)/2$];
  $A_{left} \leftarrow \left\{\boldsymbol{a} = (a_1, a_2) \in A \mid a_1 < mid\right\}, A_{right} \leftarrow \left\{\boldsymbol{a} = (a_1, a_2) \in A \mid a_1 \geq mid\right\}$;
  $B_{left} \leftarrow \left\{\boldsymbol{b} = (b_1, b_2) \in B \mid b_1 < mid\right\}, B_{right} \leftarrow \left\{\boldsymbol{b} = (b_1, b_2) \in B \mid b_1 \geq mid\right\}$;

  $count \leftarrow$*Two-Dim Pair Count*(A_{left}, B_{left})+*Two-Dim Pair Count*(A_{right}, B_{right});
  $A' \leftarrow \left\{a_2 \middle| \boldsymbol{a} = (a_1, a_2), \boldsymbol{a} \in A_{left}\right\}$;
  $B' \leftarrow \left\{b_2 \middle| \boldsymbol{b} = (b_1, b_2), \boldsymbol{b} \in B_{right}\right\}$;
  $count \leftarrow count$+*Uni-Dim Pair Count*(A', B');
  Return: *count*
end

Now we analyze the time complexity of the algorithm above. Let the time complexity be $T(n)$, where $n = |A| + |B|$. Obvious $sort(A \cup B)$ takes $O(n \log n)$ time. From **3.1** we know *Uni-Dim Pair Count*(A', B') takes $O(n \log n)$ time.

We have

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n \log n).$$

According to **Problem 01**, set $a = b = 2, d = w = 1$ and we know $T(n) = O\left(n \log^2(n)\right)$.

(Since $a = 2 = b^d$.)

We know $O(n \log^2 n) = o(n^{1.1})$. Therefore, the algorithm satisfies all requirements given by the problem. ■

3.3 $d \in \mathbb{N}^+$

Solution. From **3.2** we know we can convert the d-dimensional problem into a $(d-1)$-dimensional one. It is quite natural to design the following algorithm. The proof of the correctness of the algorithm is similar to the one in **3.2**.

Algorithm 4: Multi-Dimensional Pair Counting Algorithm

Function *Multi-Dim Pair Count* (A, B, d)

```
// A, B: two arrays. d: the number of dimensions of elements in A and B.
```

if *A is empty or B is empty* **then** **Return:** 0;
if *d=1* **then** **Return:** *Uni-Dim Pair Count*(A, B);

$S \leftarrow sort\,(A \cup B)$;
// *sort*$(\cdot)$ `returns the array sorted by the first element from the largest to the smallest.`
$mid \leftarrow$ the first element of $S[\text{Length}(S)/2]$;
$A_{left} \leftarrow \left\{\boldsymbol{a} = (a_1, ..., a_d) \in A \;\middle|\; a_1 < mid\right\}, A_{right} \leftarrow \left\{\boldsymbol{a} = (a_1, ..., a_d) \in A \;\middle|\; a_1 \geq mid\right\}$;
$B_{left} \leftarrow \left\{\boldsymbol{b} = (b_1, ..., b_d) \in B \;\middle|\; b_1 < mid\right\}, B_{right} \leftarrow \left\{\boldsymbol{b} = (b_1, ..., b_d) \in B \;\middle|\; b_1 \geq mid\right\}$;

$count \leftarrow$*Multi-Dim Pair Count*(A_{left}, B_{left}, d)+*Multi-Dim Pair Count*$(A_{right}, B_{right}, d)$;
$A' \leftarrow \left\{(a_2, ..., a_d) \middle| \boldsymbol{a} = (a_1, a_2, ..., a_d), \boldsymbol{a} \in A_{left}\right\}$;
$B' \leftarrow \left\{(b_2, ..., b_d) \middle| \boldsymbol{b} = (b_1, b_2, ..., b_d), \boldsymbol{b} \in B_{right}\right\}$;
$count \leftarrow count$+*Multi-Dim Pair Count*$(A', B', d-1)$;
Return: *count*

end

Now we analyze the time complexity of the algorithm above. We prove that $T(n, d) = O(n \log^d n)$ by induction.

BASE STEP. When $d = 1$, from **3.1** we know $T(n, d) = n \log n$.

INDUCTIVE HYPOTHESIS. When $d = D$, $T(n, d) = n \log^d n$.

INDUCTIVE STEP. When $d = D + 1$, we have

$$T(n, D+1) = 2T\left(\frac{n}{2}, D+1\right) + T\,(n, D) = 2T\left(\frac{n}{2}, D+1\right) + O(n \log^D n)$$

From **Problem 01**, set $a = b = 2, w = D, d = 1$ and we know $T(n, D+1) = O(n \log^{D+1} n)$ (since $a = 2 = b^d$), i.e. $T(n, d) = O(n \log^d n)$ still holds for $d = D + 1$.

Thus, $\forall n, d \in \mathbb{N}^+, T(n, d) = O(n \log^d n)$.

Therefore, the time complexity of the algorithm is $O(n \log^d n)$. ■

4 Problem 04

4.1 Median of Medians Is Close to the Actual Median

Proof. Since x is the median of $\frac{n}{3}$ medians, we know at least exist $\frac{n}{6}$ numbers smaller than x.

Meanwhile, by the definition of medians, we know each median is larger than a number in the triple group. Therefore, there exist at least $\frac{n}{6}$ numbers which are not medians but are smaller than x.

In conclusion, there are $\frac{n}{6} + \frac{n}{6} = \frac{n}{3}$ numbers smaller than x.

Similarly, we know exist $\frac{n}{3}$ numbers larger than x. ■

4.2 Algorithm Design

Solution. The following algorithm requires n to be the power of 3. This is easy to reach since we can add several numbers smaller than the minimal element in the original a array into the array and extend the length to a power of 3, which takes only $O(n)$ time.

After extend the array a, the algorithm is as follows.

Algorithm 5: Median of Medians Algorithm

Function *Find k-th Largest*$(a[1:n], k)$

```
// a: the array containing distinct numbers.
//    We require n = 3^m, m ∈ ℕ⁺.
// k: the expected output should be the k-largest number in a.

// Generate medians.
Medians ← ∅;
for i = 0 → n/3 − 1 do
    b ← sort(a[3i + 1 : 3i + 3]);
    Add element b[2] into Medians;
end
mid ← Find k-th Largest(Medians, ⌈k/6⌉);

Left ← ∅, Right ← ∅;
for i = 0 → n do
    if a[i] ≥ mid then  Add element a[i] into Right;
    else  Add element a[i] into Left;
end
if |Right| = k then  Return: mid;
if |Right| > k then  Return: Find k-th Largest(Right, k);
Return: Find k-th Largest(Left, k − |Right|)
```

end

Now we analyze the time complexity of *Median of Medians* Algorithm.

Let the time complexity be $T(n)$.

Obvious median generation takes $O\left(\frac{n}{3}3\log 3\right) = O(n)$ time and dividing a takes $O(n)$ time. From **4.1** we know $|Left| \geq \frac{n}{3}, |Right| \geq \frac{n}{3}$, i.e. $|Left| \leq \frac{2n}{3}, |Right| \leq \frac{2n}{3}$. Then we have

$$\begin{aligned} T(n) &\leq T\left(\frac{n}{3}\right) + T\left(\max|Left|, |Right|\right) + O(n) \\ &\leq T\left(\frac{n}{3}\right) + T\left(\frac{2n}{3}\right) + O(n) \end{aligned}$$

We find the equation above is exactly the form of time complexity in **Problem 02**. Thus, by **Problem 02**, we know $T(n) = O(n\log n)$. ■

4.3 Alter the Size of Group (The Problem In the Margin)

Solution. It's easy to improve the algorithm in **4.2** to a version with alterable group size.

For even size, we select the $(size/2+1)$-largest number as the median; For odd size, we select the $\lceil size/2\rceil$-largest number as the median. Obvious there exist at least $\frac{n}{size}\left(\lceil size/2\rceil - 1\right)$ numbers smaller than the median of medians, which is similar to **4.1**.

The altered algorithm is as follows.

Algorithm 6: Median of Medians Algorithm

Function *Find k-th Largest*$(a[1:n], k, size)$

```
// a: the array containing distinct numbers.
//    We require n = size^m, m ∈ ℕ⁺.
// k: the expected output should be the k-largest number in a.
// size: the size of median group. We require size ≥ 3.

// Generate medians.
Medians ← ∅;
for i = 0 → n/size − 1 do
    b ← sort(a[size·i + 1 : size·(i + 1) − 1]);
    Add element b[⌈size/2⌉] into Medians;
end
mid ← Find k-th Largest(Medians, ⌈k/(2·size)⌉, size);

Left ← ∅, Right ← ∅;
for i = 0 → n do
    if a[i] ≥ mid then  Add element a[i] into Right;
    else  Add element a[i] into Left;
end
if |Right| = k then  Return: mid;
if |Right| > k then  Return: Find k-th Largest(Right, k, size);
Return: Find k-th Largest(Left, k − |Right|, size)
```

end

Obvious median generation takes $\left(\frac{n}{size} \cdot size \log size\right)$ operations and dividing a takes n operations. Also we know $|Left| \geq \frac{n}{size}\left(\lceil\frac{size}{2}\rceil - 1\right), |Right| \geq \frac{n}{size}\left(\lceil\frac{size}{2}\rceil - 1\right), |Left| \leq n - \frac{n}{size}\left(\lceil\frac{size}{2}\rceil - 1\right)$, $|Right| \leq n - \frac{n}{size}\left(\lceil\frac{size}{2}\rceil - 1\right)$. Thus, we have

$$T(n) \leq T\left(\frac{n}{size}\right) + T\left(n - \frac{n}{size}\left(\lceil\frac{size}{2}\rceil - 1\right)\right) + O(n)$$

Considering

$$\begin{aligned}\frac{n}{size} + n - \frac{n}{size}\left(\left\lceil\frac{size}{2}\right\rceil - 1\right) + \frac{n}{size} \leq n &\iff \frac{2n}{size} \leq \frac{n}{size}\left(\left\lceil\frac{size}{2}\right\rceil - 1\right)\\ &\iff \left\lceil\frac{size}{2}\right\rceil \geq 3\\ &\iff size \geq 5,\end{aligned}$$

we analyze time complexity by different $size$.

CASE 01. When $size \geq 5$. we prove that $T(n) = O(n)$ by induction.

BASE STEP. When $n = 1$, Obvious $T(n) = O(n)$ (the time median generation need.)

INDUCTIVE HYPOTHESIS. When $n \leq N, T(n) = O(n)$.

INDUCTIVE STEP. When $n = N + 1$,

$$\begin{aligned}T(N+1) &\leq T\left(\frac{N+1}{size}\right) + T\left(N + 1 - \frac{N+1}{size}\left(\left\lceil\frac{size}{2}\right\rceil - 1\right)\right)\\ &\quad + N + 1 + \frac{N+1}{size} \cdot size \log size\\ &\leq c \cdot \frac{N+1}{size} + c \cdot \left(N + 1 - \frac{N+1}{size}\left(\left\lceil\frac{size}{2}\right\rceil - 1\right)\right)\\ &\quad + N + 1 + \frac{N+1}{size} \cdot size \log size\\ &\leq c \cdot (N+1)\end{aligned}$$

when c is large enough.

Thus, when $size \geq 5$, $T(n) = O(n)$.

CASE 02. When $size = 4$,

$$T(n) = T\left(\frac{n}{4}\right) + T\left(\frac{3n}{4}\right) + O(n)$$

By analyses similar to **Problem 02**, we get $T(n) = O(n \log n)$.

CASE 03. When $size = 3$, from **4.2**, we know $T(n) = O(n \log n)$.

Thus, $T(n) = \begin{cases} O(n), & size \geq 5 \\ O(n \log n), & size = 3, 4 \end{cases}$

Obvious the best choice of $size$ is 5. ■

5 Rating and Feedback

The completion of this homework takes me seven days, about 40 hours in total. Writing formal proof and solution by *latex* is the most time-consuming part.

The ratings of each problem is as follows.

Problem	Rating
1	3
2	2
3.1	2
3.2	3
3.3	3
4.1	1
4.2	3
4 problem in margin	4

Table 1: Ratings.

Most problems are completed on my own, except ***the one in the margin of the* Problem 04**. The solution of this additional problem is completed with discussions with **Yilin Sun**.