ShanghaiTech University

CS240 Algorithm Design and Analysis

Spring 2024

Problem Set 1

Due: 23:59, March 28, 2024

1. Submit your solutions to the course Gradescope.
2. If you want to submit a handwritten version, scan it clearly.
3. Late homeworks submitted within 24 hours of the due date will be marked down 25%. Homeworks submitted more than 24 hours after the due date will not be accepted unless there is a valid reason, such as a medical or family emergency.
4. You are required to follow ShanghaiTech's academic honesty policies. You are allowed to discuss problems with other students, but you must write up your solutions by yourselves. You are not allowed to copy materials from other students or from online or published resources. Violating academic honesty can result in serious penalties.

Problem 1:

1. Analyze the time complexities of the following algorithms and explain your reasoning.

Algorithm 1

```
for i ← 1 to n by i ← 2i do
    for j ← n downto 0 by j ← j/2 do
        for k ← j to n by k ← k + 2 do
            res ← res + ij + jk
        end for
    end for
end for
```

Algorithm 2

```
for i ← n downto 0 do
    for j ← 1 to n by j ← 2j do
        for k ← 0 to j do
            res ← res + ij + jk
        end for
    end for
end for
```

2. Solve the following recurrence relations. Do not use the Master Theorem. Show all of your work.

$$T(n) = T(n-k) + 2k$$

$$T(n) = 2^k T(n/2^k) + kn$$

$$T(n) = 2^k T(n/2^k) + 2^k - 1$$

Solution:

1. (1)Algorithm 1:

The i-th loop runs $\log_2 n$ times.

Suppose $n = 2^x$, i.e. $x = \log_2 n$, then for the j-th loop and the k-th loop, the total computation is:

$$\frac{1}{2}(2^x - 0) + \sum_{j=0}^{x} \frac{1}{2}(2^x - 2^j) = 2^{x-1} + \frac{1}{2}(x+1)2^x - \frac{1}{2}(2^{x+1} - 1) = \Theta(x2^x) = \Theta(n \log n)$$

Combine the i-th loop, the computation is

$$\log n \cdot \Theta(n \log n) = \Theta(n \log^2 n)$$

So above all, the total time complexity is $\Theta(n \log_2^2 n)$.

(2) Algorithm 2:
The i-th loop runs $n + 1$ times.
Suppose $n = 2^x$, i.e. $x = \log_2 n$, then for the j-th loop and the k-th loop, the total computation is:

$$\sum_{j=0}^{x} (2^j + 1) = (2^{x+1} - 1) + (x + 1) = \Theta(2^x) = \Theta(n)$$

Combine the i-th loop, the computation is

$$(n + 1) \cdot \Theta(n) = \Theta(n^2)$$

So above all, the total time complexity is $\Theta(n^2)$.

2. (1) Suppose $n = c \cdot k$, i.e. $c = \dfrac{n}{k}$, and suppose $T(0)$ is a constant, then we have

$$\begin{aligned} T(ck) &= T[(c-1)k] + 2k \\ T[(c-1)k] &= T[(c-2)k] + 2k \\ &\cdots \\ T(2k) &= T(k) + 2k \\ T(k) &= T(0) + 2k \end{aligned}$$

And all these equations, we have

$$T(ck) = T(0) + c \cdot (2k) = O\left(\frac{n}{k} \cdot 2k\right) = O(n)$$

So above all, $T(n) = O(n)$.

(2) From the recursion tree Figure 1, we can get that the total computation

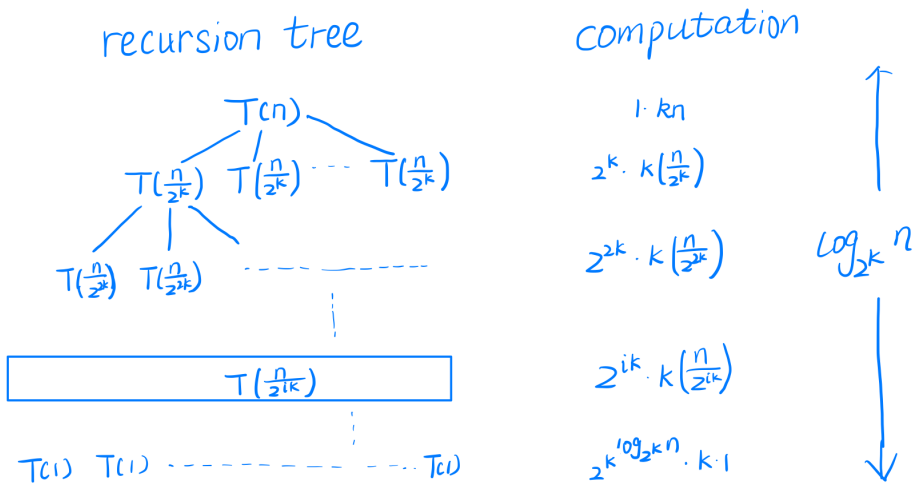

Figure 1: The recursion tree of $T(n) = 2^k T(n/2^k) + kn$

is:

$$\begin{aligned} T(n) &= \sum_{i=0}^{\log_{(2^k)} n} (2^k)^i \cdot k \cdot \frac{n}{(2^k)^i} \\ &= \sum_{i=0}^{\frac{1}{k}\log_2 n} kn \\ &= O((\frac{1}{k}\log_2 n) \cdot kn) \\ &= O(n \log n) \end{aligned}$$

So above all, $T(n) = O(n \log n)$.

(3) Similarly with (2), from the recursion tree Figure 2, we can get that the total computation is:

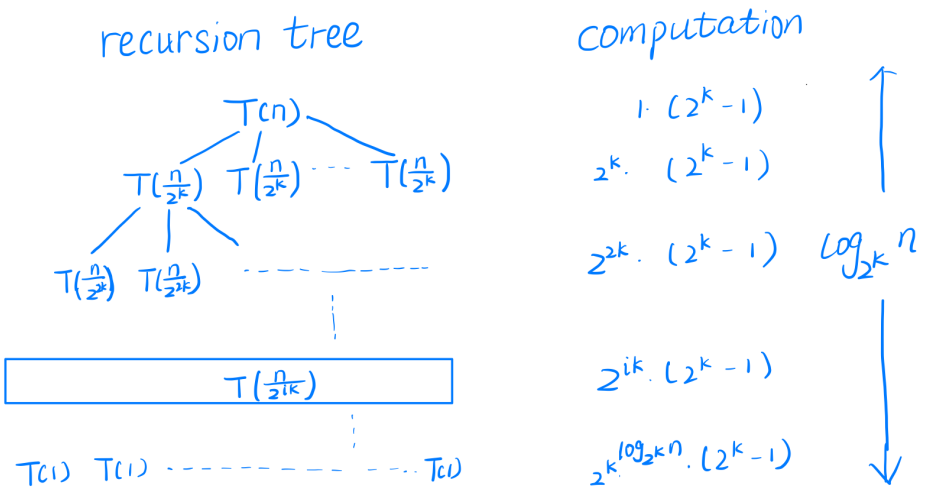

Figure 2: The recursion tree of $T(n) = 2^k T(n/2^k) + 2^k - 1$

$$
\begin{aligned}
T(n) &= \sum_{i=0}^{\log_{(2^k)} n} (2^k)^i \cdot (2^k - 1) \\
&= O((2^k - 1) \cdot \frac{(2^k)^{(\frac{1}{k}\log_2 n + 1)} - 1}{2^k - 1}) \\
&= O(2^{\log_2 n + k} - 1) \\
&= O(n)
\end{aligned}
$$

So above all, $T(n) = O(n)$.

Problem 2:

Sort the following functions in ascending order of growth.

$$f_1(n) = \log_2 n \tag{1}$$
$$f_2(n) = n^{\sqrt{n}} \tag{2}$$
$$f_3(n) = (\log_2 n)^{\log_2 n} \tag{3}$$
$$f_4(n) = n^{\frac{3}{4}} \tag{4}$$
$$f_5(n) = \log_2 \log_2 n \tag{5}$$
$$f_6(n) = 2^{\log_2 n} \tag{6}$$
$$f_7(n) = 10^{2024} \tag{7}$$
$$f_8(n) = 5^n \tag{8}$$
$$f_9(n) = \log_2 n \cdot \log_2 3^n \tag{9}$$
$$f_{10}(n) = (\log_2 n)^n \tag{10}$$

Solution:

$f_6(n) = 2^{log_2 n} = n$

$f_7(n) = 10^{2024}$, which is a constant.

$f_9(n) = \log_2 n \cdot \log_2 3^n = O(n \log n)$.

So above all, the ascending order of grouth of each function is:

$$f_7 < f_5 < f_1 < f_4 < f_6 < f_9 < f_3 < f_2 < f_8 < f_{10}$$

Problem 3:

Suppose you have two arrays each with n values, and all the values are unique. You want to find the median of the $2n$ values, i.e. the n'th smallest value. To do this you can make queries to either array, where a query for k to an array returns the k'th smallest value in that array. Give an algorithm which finds the median of the two arrays using $O(\log n)$ queries.

Solution:

Suppose $n = 2^k$.

For the two arrays, A, B, we first sort them to be the ascending order.

Then to find the median, we can do following steps.

- 1. Query the $\frac{|A|}{2}$-th smallest element of A, set it to be a, and the $\frac{|B|}{2}$-th smallest element of B, set it to be b.
- 2. Split array A into the left half and the right half, and so does array B.
  - If $a < b$, let the A be the right half of origin A, B be the left half.
  - Else $a > b$, let A be the left half, B be the right half.

We can recursively following the step 1 and step 2 repeatively in $(k-1)$ times, where $k = \log_2 n$.

And the for the k's comparison, it must has $|A| = |B| = 2$, we take the second small number among the 4 numbers, and that number is the n-th smallest number of the two arrays.

Proof:

For the i-th step, W.L.O.G, let $a < b$. Then the n-th smallest number must be bigger than a, but less than b, so the n-th smallest number must be on the right part of A, or the left part of B.

And the number we take is the $\left[\left(\frac{1}{2}\sum_{i=2}^{k} 2^i\right) + 2 = (n-2) + 2 = n\right]$-th smallest number.

And we totally has $O(2(k-1)+1) = O(\log n)$ queries.

So above all, with $O(\log n)$ queries, we can find the median of the two arrays.

Problem 4:

Suppose there are n values in an array, and we want to sort the array using "flipping" operations. A flip takes two inputs i and j, with $1 \leq i \leq j \leq n$, and reverses the order of the values between indices i and j in the array. For example, if the array is $[1, 1, 5, 3, 4]$, then a flip with indices 2 and 5 changes the array to $[1, 4, 3, 5, 1]$. Assume a flip with inputs i and j has cost $j - i$.

(a) Assume first that all the values in the array are either 1 or 2. Design an algorithm which sorts the array using $O(n \log n)$ cost, and analyze its cost.
(Hint: mergesort)

(b) Now suppose the array contains arbitrary values. Design an algorithm which has $O(n \log^2 n)$ expected cost, and analyze its cost. Note that your algorithm is allowed to make randomized choices.
(Hint: quicksort, and the previous algorithm)

Solution:

(a) Similarly to the process of merge sort, we firstly suppose that the array split from the middle, and get two ordered arraies, then we make sure the merged array is ordered.
After split from the middle, we suppose the two small arraies are ordered. Since the array is combined with only 1 and 2, so there must has a line of demarcation to seperate 1s and 2s. And we flip the origin array from the left small array's demarcation line(right side, i.e. the first 2) to the right small array's demarcation line(left size, i.e. the last 1).
This would take the cost of $O(n)$, where n is the length of the origin array.
With this flip, the array must has a continously 1s followed with continously 2s, which means that the flipped array is ordered. And we can just recursively repeat this process like merge sort.

Then we can get the cost relationship: suppose the cost of sorting an array with length n is $T(n)$, then we have

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

According to the Master Theorem, we can get that $T(n) = O(n \log n)$.
So above all, we can sort this kind of arraies with cost $O(n \log n)$.

(b) We do this similarly to the quicksort. For each range $[a, b]$, we can select a random number, such as the first element in the range as the pivot.

Then we consider the rest of the element, if the element $<$ pivot, set it into 1, otherwise set it into 2.
Then the range $[a+1, b]$ is conbined by only 1 and 2. And we can sort them with the method in (a), and its cost is $O(n \log n)$, where n is the length of the range. Suppose the boarder line of 1s and 2s are at index $a+i$(i.e. the number of numbers that are less than the pivot is i). Then we flip the range $[a, i]$ to make sure all element less than the pivot are at the left of the pivot, and all element greater than the pivot are at the right of the pivot. This will cost $O(n)$, so the total cost during the step is $O(n \log n)$.
Then we recursively do this to the left part and the right part.
So the cost is that $C(n) = C(i)+C(n-i-1)+O(n \log n)$, where i is the number of numbers that are less than the pivot.
Suppose that the expected cost is $T(n)$, then we have

$$T(n) = \sum_{i=0}^{n-1} \frac{1}{n}[T(i) + T(n-i-1) + O(n \log n)]$$

$$nT(n) = 2\sum_{i=0}^{n-1} T(i) + O(n^2 \log n) \qquad \text{(equation 1)}$$

$$(n-1)T(n-1) = 2\sum_{i=0}^{n-2} T(i) + O((n-1)^2 \log(n-1)) \quad \text{(equation 2)}$$

Let equation 1 minus equation 2, we have

$$\begin{aligned}
nT(n) &= (n+1)T(n-1) + O(n \log n) \\
\frac{T(n)}{n+1} &= \frac{T(n-1)}{n} + O(\frac{\log n}{n+1}) \\
\frac{T(n-1)}{n} &= \frac{T(n-2)}{n-1} + O(\frac{\log(n-1)}{n}) = \frac{T(n-2)}{n-1} + O(\frac{\log n}{n}) \\
&\cdots \\
\frac{T(2)}{3} &= \frac{T(1)}{2} + O(\frac{\log 2}{3}) = \frac{T(1)}{2} + O(\frac{\log n}{3})
\end{aligned}$$

Sum up these equations, we can get that

$$\frac{T(n)}{n+1} = O\left[log n \cdot \sum_{i=3}^{n+1} \frac{1}{i}\right]$$

From the knowledge of mathematics, we know that

$$\sum_{i=1}^{n} \frac{1}{i} = \ln n + \gamma = O(\log n)$$

So we have

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

i.e.

$$T(n) = O(n \log^2 n)$$

So above all, we can get that the expected cost is $O(n \log^2 n)$.

Problem 5:

Suppose n contestants will participate in a set of contests G. In each contest the contestants are split into two teams, possibly of different sizes, and such that each team contains at least one contestant. We require that for any two contestants, there is some contest in G in which these contestants are on different teams. Design the contests in G in order to minimize the total number of contests $|G|$.

Solution:

Given each contestant a number $0, 1, 2, 3, \cdots, n$. We can do binary decomposition to each number. Suppose $n = 2^k - 1$, i.e. $k = \log_2 n - 1$. For the i-th contestant, the binary decomposition of i is $b_{k-1}b_{k-2}\cdots b_0$.
We totally set k contests. For the j-th contest($j = 0, 1, \cdots, k-1$), if the i-th contestant has $b_j = 0$, then we devide him into team 1, otherwise team 2.
So total $k = O(\log n)$ contests are hold, and each contestant must have at least one contest has different group with other contestants.

Proof:

- correctness:
  $\forall x \neq y$, where x and y are the contestants' number. Then the binary decomposition of x, y must have at least one digit difference. And according to the policy above, the two contestants must be in different teams at the contest that they has different digit.
  So all contestants must have at least one contest in different teams.

- minimize:
  Suppose that we totally has $t < k$ contests. Suppose the number of the contests are $0, 1, \cdots, t-1$, then for each contest i, if the contestant j is in team 1, then we mark j's i-th digit to be 0, otherwise, mark it to be 1.
  So with this marking policy, since each contestants has at least one contest in different teams, so their marked sequence must be different. We can regard each sequence as a binary number, so each contestant is given a unique number.
  But there are at most $2^t - 1 < 2^k - 1 = n$ numbers, so it is impossible to has at least $t < k$ contests.
  So $k = \lceil \log n \rceil$ is the minimized number of contests.

So above all, the minimized total number of contests $|G|$ is $\lceil \log n \rceil$.

Problem 6:

Suppose you have a graph with n nodes. Initially the graph contains no edges, and your task is to add a set of directed edges to the graph so that for any pair of nodes i and j where $i < j$, there is a path from i to j using at most 2 of the edges you added. You can only add edges of the form (i, j) for $i < j$. Give an algorithm to find a set of $O(n \log n)$ edges which satisfy this requirement.

Solution:

For a set of nodes, we number them as $1, 2, \cdots, n$, and let the number sequence as an array A. we can split them from the middle, and generate two new set of points L, R.

Then we set edge recursively:
For the current array A', we devide it from the middle, and take out the middle node of the array, named it the 'mid_point'.
The nodes before the mid_point are array L, after it are array R.

Then we set edges: $\forall i \in L$, we set a direct edge $i \rightarrow$ mid_point, and $\forall j \in R$, we set a direct edge mid_point $\rightarrow j$.
Then we recursively set edges for array L and array R, and during each step, we totally set up $O(n)$ edges.

So the total number of connected edges $T(n)$ has the relation that

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

So according to the Master Theorem, we can get that $T(n) = O(n \log n)$.

Correctneses:
For each pair of node $i < j$, during the recursion, it must one time that $i, j \in A'$, but $i \in L, j \in R$, or i or j is the mid_point. According to our policy, there has an edge $i \rightarrow j$, otherwise, there exist edges that $i \rightarrow$mid_point and mid_point $\rightarrow j$.
So each $i < j$, i can using at most 2 edges to j.

So above all, we can find a set of $O(n \log n)$ edges to satisfy the requirement.