Lab02-Divide & Conquer and Greedy Approach

Algorithm and Complexity, Xiaofeng Gao, Spring 2022.

∗ If there is any problem, please contact TA Wanghua Shi.

∗ Name: Zhenran Xiao Student ID: 520030910281 Email: xiaozhenran@sjtu.edu.cn

1. Can Master Theorem apply to the recursive formula $T(n) = 2T(\frac{n}{5}) + O(\log n)$? What is the time complexity of $T(n)$ thereby?

   Solution. No. Master Theorem can apply to the formula like $T(n) = aT(\frac{n}{b}) + O(n^d)$ .But we can't determine d in $T(n) = 2T(\frac{n}{5}) + O(\log n)$.
   Calculate the time complexity by recurrence computation:

$$\begin{aligned}
T(n) &= 2T(\frac{n}{5}) + O(\log n) \\
&= 2(2T(\frac{n}{5^2}) + O(\log \frac{n}{5})) + O(\log n) \\
&= 2^2T(\frac{n}{5^2}) + 2O(\log \frac{n}{5}) + O(\log n) \\
&= 2^3T(\frac{n}{5^3}) + 2^2O(\log \frac{n}{5^2}) + 2O(\log \frac{n}{5}) + O(\log n) \\
&= 2^kT(\frac{n}{5^k}) + 2^{k-1}O(\log \frac{n}{5^{k-1}}) + ... + 2O(\log \frac{n}{5}) + O(\log n)
\end{aligned}$$

   There is $n/5^k = 1$. So $k = \log_5 n$.

$$\begin{aligned}
T(n) &= 2^kT(1) + 2^{k-1}O(\log n - (k-1)\log 5) + ... + 2O(\log n - \log 5) + O(\log n) \\
&= O((\sum_{i=0}^{k-1} 2^i)\log n - (\sum_{i=0}^{k-1} i \cdot 2^i)\log 5) \\
&= O((2^k - 1)\log n - [(k-2)2^k + 2]\log 5) \\
&= O(2^{\log_5 n} \cdot \log n - \log n - \log_5 n \cdot 2^{\log_5 n} \cdot \log 5 + 2 \cdot 2^{\log_5 n} \cdot \log 5 - 2\log 5) \\
&= O(n^{\log_5 2} \cdot \log n - \log n - \log n \cdot n^{\log_5 2} + 2\log 5 \cdot n^{\log_5 2} - 2\log 5) \\
&= O(2\log 5 \cdot n^{\log_5 2} - \log n - 2\log 5) \\
&\approx O(n^{0.43})
\end{aligned}$$

   Therefore, the time complexity of $T(n)$ is $O(n^{0.43})$. □

2. Given an array of positive integers, we will implement *floating point division* between adjacent integers. For instance, given an array [66, 22, 15, 78], we will execute $66/22/15/78 \approx 0.003$. However, you can add some parentheses at any position to change the priority of arithmetic and get a maximum quotient. Given an input array, design an algorithm to output an arithmetic with the maximum quotient, be sure to avoid redundant parentheses. For example, given the above input "[66, 22, 15, 78]", your algorithm should output "66/(22/15/78)", because it is the maximum quotient (illustrated as follows):

   - $66/22/15/78 \approx 0.003$;
   - $66/(22/15/78) = 3510$;
   - $66/(22/15)/78 \approx 0.58$;
   - $66/22/(15/78) = 15.6$;

- $66/(22/(15/78)) \approx 0.58$.

Solution.

Algorithm 1: *maxquot*(*array*[1, ..., *n*])

```
Input: array[1,...,n]
Output: a formula
$formula \leftarrow$ "$array[1]/array[2]/array[3]/.../array[n]$" ;
for $array[1]$ to $array[n]$ in the formula do
    if $array[2]$ then
        insert a "(" in front of it;
    if $array[n]$ then
        insert a ")" behind it;
return $formula$;
```

If we want to make the result biggest, we need to make the molecular bigger and the denominator smaller. By add parentheses, we can make all ingeters except the second integer become multipliers of the molecular. Therefore, the smallest denominator is the second integer and the biggest molecular is the product of all the other integers. □

3. Given an array $A = [a_1, \cdots, a_n]$, we define "k-reverse" operation ($1 \leq k \leq n$) as reversing the sub-array $[a_1, a_2, \cdots, a_k]$, *i.e.*, changing $A = [a_1, a_2, \cdots, a_k, a_{k+1}, \cdots, a_n]$ to $A = [a_k, a_{k-1}, \cdots, a_1, a_{k+1}, \cdots, a_n]$. For instance, if we perform a "3-reverse" operation on array $A = [72, -16, -38, 9]$, we can get the result $A = [-38, -16, 72, 9]$. Please design an algorithm to sort A in ascending order only by reverse operations. Output the list of k values per step and analyze its time complexity. For instance, given an array $A = [3, 2, 4, 1]$, your output should be as follows:

   Round 1: $k = 4$, $A = [1, 4, 2, 3]$;

   Round 2: $k = 2$, $A = [4, 1, 2, 3]$;

   Round 3: $k = 4$, $A = [3, 2, 1, 4]$;

   Round 4: $k = 3$, $A = [1, 2, 3, 4]$.

Solution.

Algorithm 2: *ReverseSort*$(array[1,..,n])$

```
Input: array[1,...,n]
Output: array[1,...,n] sorted in nondecreasing order.
round ← 1 ;
k ← n ;
do k − reverse operation;
Output round, k and array[1,..,n] ;
for i ← n to 2 do
    max ← array[1] ;
    for j ← 1 to i do
        if array[j] > max then
            k ← j ;
            max ← array[j] ;
    if k = i then
        continue ;
    else if k = 1 then
        k ← i ;
        do k − reverse operation;
        round ← round + 1 ;
        Output round, k and array[1,..,n] ;
    else
        do k − reverse operation;
        round ← round + 1 ;
        Output round, k and array[1,..,n] ;
        k ← i ;
        do k − reverse operation;
        round ← round + 1 ;
        Output round, k and array[1,..,n] ;
return;
```

$$T(n) = c_1 + \sum_{i=2}^{n}(c_2 + \sum_{j=1}^{i} c_3) \sim O(n^2)$$

The time complexity is O(n^2).

□

4. A *perfect array* A with n numbers satisfies: (1) it is a permutation of integers in the range of $[1, n]$; and (2) there is no index k with $1 \le i < k < j \le n$ where $2 \cdot A[k] = A[i] + A[j]$. For any positive integer n, design an algorithm to generate a perfect array A of length n (any perfect array is acceptable).

Solution.

Algorithm 3: $PerfectArray(n)$

```
Input: an interger $n$.
Output: a perfect array $A$ of length $n$.
if $n = 1$ then
    return [1] ;
else
    $LeftArray \leftarrow PerfertArray(\frac{n+1}{2})$ ;
    for $i \leftarrow 0$ to size of $LeftArray$ do
        $tmp \leftarrow LeftArray[i] * 2 - 1$ ;
        append $tmp$ into $A$ ;
    $RightArray \leftarrow PerfertArray(\frac{n}{2})$ ;
    for $j \leftarrow 0$ to size of $RightArray$ do
        $tmp \leftarrow RightArray[j] * 2$ ;
        append $tmp$ into $A$ ;
return $A$;
```

□

Remark: You need to include your .pdf and .tex files in your uploaded .rar or .zip file.