CS222 Homework 5

Exercises for Algorithm Design and Analysis by Li Jiang, 2016 Autumn Semester

5140309507 林禹臣 yuchenlin@sjtu.edu.cn

Codes are outside this .pdf file, and thus I mainly explain my thoughts and algorithms and then give the proof of the time complexity here as well.

1. Given a positive integer n and you can do operations as follow:

   1. If n is even, replace n with n/2.

   2. If n is odd, you can replace n with either n + 1 or n - 1.

   What is the minimum number of replacements needed for n to become 1?

   Example 1:

   Input: 8

   Output: 3

   Explanation: 8 => 4 => 2 => 1

   Example 2:

   Input:7

   Output:4

   Explanation: 7 => 8 => 4 => 2 => 1 or 7 => 6 => 3 => 2 => 1

   Input: int n;

   Output: int minNum;

Solution.

1. Obviously, this problem can be solved with divide-and-conquer strategy. For any given n, we first check if it is one already. If it is then we can just return the number of steps we have already done. Otherwise, we then check if it is even. If so, our only choice is to replace it with $n/2$. Otherwise, we need to check both $n+1$ and $n-1$ and then choose the one with smaller times of replacement to be the next n.
It is a little tricky to analyze the time complexity of this algorithm since we can not write down the exact $T(n)$ function with uncertainty of $T(n+1)$ and $T(n-1)$. But the number of times we reduce $T(n)$ to $T(n-1)$ or $T(n+1)$ can be seen as to $T(\frac{n+1}{2})$ or $T(\frac{n-1}{2})$ in the next call. Thus, we can approximate the T(n) as $T'(n) = T'(n/2) + O(1)$.
According to the Master Theorem, we can conclude that the $T'(n) = \Theta(lgn)$. Thus, $T(n) = \Omega(T'(n)) = \Omega(lgn)$.

2. However, actually we can devise a greedy strategy to decide whether we should replace n with $n-1$ or $n+1$ under a certain condition which can be check in constant time. That magical condition is just when $(n+1)\%4 == 0$. Following is a simple mathematical proof:

Proposition: When n is odd and $(n+1)\%4 == 0$, the replacement times of $n+1$ is always less than or equal to that of $n-1$. Except for 3 for that: 3->2->1 < 3->4->2->1

Simple Explanation:
If n+1 = 4k, then the route are like:
n –> n+1 –> 2k –> k. (3 Steps)
n–>n-1=4k-2–>2k-1–>2k–>k (4 Steps)
or n–>n-1=4k-2–>2k-1–>2k+2–>k+1 (4+ Steps)
PS: When n = 3, k=1, 2k-1=1 so it needs only 2 steps. It is a special case.

This algorithm is strictly fast than the previous one for it needs only once check at each step. And its time complexity can be proved to be $O(lgn)$.

□

2. Implement a basic calculator to evaluate a simple expression string.

   The expression string may contain open (and closing parentheses), the plus + or minus sign -, non-negative integers and empty spaces.

   You may assume that the given expression is always valid.

   Some examples:

   "1 + 1" = 2

   " 2-1 + 2 " = 3

   "(1+(4+5+2)-3)+(6+8)" = 23

   Note: Do not use the eval built-in library function.

   Input: string str;

   Output: int result;

   Solution. This problem is classic problem using stack and I cannot figure out the relation between this problem to greedy or divide-and-conquer or dp or network flow.. Thus, I just solve it traditionally.

   We have to initialize two stacks, one for numbers and the other for the operations and parentheses. Thus, we just process the char one by one as follows:

   If the current char c is a (then we just need to push it into the operation stack.
   If the c is + or −, we have to pop two top numbers in the number stack and calculate the result and put it into the number stack.
   If the c is a digit, if the top of the operation stack is a left parentheses and then we just push it into the number stack; Otherwise, we can calculate this number and the top of the number stack with the top operation in operation stack.
   If the c is a), we should immediately calculate all the numbers with the operations in the two stacks accordingly until we meet a (in operation stack. Then we can pop the '(' and continue processing next char.

   As to analyze the time complexity of this traditional algorithm, we can consider it in the view of the push and pop times of each char. Every element have exactly 1 time being pushed into a stack and being popped out of a stack. And the calculation times in just $O(n/2)$ at most. Thus the time complexity is $O(2n) + O(n/2) = O(n)$.

   □

3. In the computer world, use restricted resource you have to generate maximum benefit is what we always want to pursue.

   For now, suppose you are a dominator of m 0s and n 1s respectively. On the other hand, there is an array with strings consisting of only 0s and 1s.

   Now your task is to find the maximum number of strings that you can form with given m 0s and n 1s. Each 0 and 1 can be used at most once.

   Note:

   1. The given numbers of 0 and 1 will both not exceed 100.
   2. The size of given string array won't exceed 600.

   Solution. This problem is a typical 2-dimensional packing problem. In such problems, we usually find two arguments that construct our search space with the size m*n in this problem. And then given a series conditions for us to meet. Usually the problem asks us to find the maximum or minimum valid situations. In this problem, we are given a set of sequences and we need to find the maximum number of the sequences we can satisfy.

   Dynamic programming is a strong approach to attack packing problems. In this problem we define our state(x,y) as the answer to this problem such that the state(m,n) is the final answer we get.

   The core section of the dynamic programming is to make sure the order in which we calculate the states and the transforming equation of the states. We calculate the states in the order of each string and x from m to the zero numbers and y from n to the one numbers. In another word, we calculate the states in a reversing order.

   For each s in string set. We firstly find out the number of zeros in s, say z, and number of ones, say o. Then we can conclude that:

   For each x in [m,z]:
   For each y in [n,o]:
   state[x][y] = max(state[x-z][y-o]+1,state[x][y])

   In which, $state[x-z][y-o]+1$ means we want the current string s, $state[x][y]$ means we do not want s. And the transformation equation help us to select the greater one.
   Them time complexity is $T(u,m,n) = u*(m-z)(n-o) = u(mn-om-zn+oz) = O(umn)$, in which u = size of the string set, m and n are the same in the description of the problem.

   □