Lab01-Preliminary

VE281 - Data Structures and Algorithms, Xiaofeng Gao, TA: Qingmin Liu, Autumn 2019

∗ Name: Jintian Ge   Student ID: 517021911142   Email: gejintian@sjtu.edu.cn

1. What is the time complexity of the following code?

```
// REQUIRES: an integer k
// EFFECTS: return the number of times that Line 12 is executed
int count(int k)
{
    int count = 0;
    int n = pow(2,k); // n=2^k
    while (n>=1)
    {
        int j;
        for (j=0;j<n;j++)
        {
            count += 1;
        }
        n /= 2;
    }
    return count;
}
```

Solution. For each for loop, there is n times of addition to be executed. therefore, totally we have:

$$\sum_{n=1}^{2^k} n = 2^{k+1}.$$

Also, we know that

$$2^{k+1} = 2 \times 2^k$$

such that

$$2^{k+1} = O(2^k)$$

Hence, the time complexity of the following code is 2^k. □

2. Given an array **nums** of n integers, are there elements a, b, c in nums such that $a+b+c=0$? Write a program to find all unique triplets in the array which gives the sum of zero. Give your code as the answer. **Claim that the time complexity of your program should be less than or equal to $O(n^2)$.**

Examples: Input array [-1, 0, 1, 2, -1, -4], the solution is [[-1, 0, 1], [-1, -1, 2]]

Solution. Please explain your design and fill in the following block:

```
// REQUIRES: an integer array nums of size n
// EFFECTS: return a list of triplets, the sum of each triplet
   equals to 0.
#include <iostream>
#include <vector>
```

```
#include <algorithm>
using namespace std;
vector<int*> findTriplet(vector<int>&nums, int n){
    vector<int*> vec;
    sort(nums.begin(),nums.end());
    int i, j, k;
    bool is_Empty = true;
    for(i = 0;i<n-1;i++){
        k = n - 1, j = i + 1;
        while(k != j){
            if(nums[j] + nums[k] < -nums[i]){
                j++;
            }
            else if(nums[j] + nums[k] > -nums[i]){
                k --;
            }
            else{//All following is to make sure no duplication.
                if(!is_Empty){
                    if(vec.back()[0] != nums[i]   vec.back()[1] !=
                       nums[j]   vec.back()[2] != nums[k]){
                        int a[3] = {nums[i], nums[j], nums[k]};
                        vec.push_back(new int[3](a));
                        is_Empty = false;
                    }
                }
                else{
                    int a[3] = {nums[i], nums[j], nums[k]};
                    vec.push_back(new int[3](a));
                    is_Empty = false;
                }
                j++;
                k--;
            }
        }
    }
    return vec;
}
```

Explain the time complexity of your solution here.

In my algorithm, I use a vector containing arrays to hold all the possible combinations. The sort function is used to rearrange the number in the input vector. Its time complexity is $O(n^2)$.

For the "find" part, I will go through all the numbers for the "for" loop firstly. Next, I introduce a "while" loop with two parameters j and k. The basic thought is: If I decide a number x, like 2, then I need to find two numbers whose sum is $-x$, which is -2 here. These two number should be place symmetrically on both sides of $-\frac{1}{2}x$. In this case, I just need to move j and k towards the place of $-\frac{1}{2}x$. So, j starts from the beginning of the vector and k starts from the end of the vector. Since this vector is sorted, $nums[j]$ will always be the smallest one and $nums[k]$ will always be the greatest one among which are to be tested. In this case, if $nums[j] + nums[k] < -nums[i]$, we need to find a larger $nums[j]$, so I increase j to make $nums[j]$ larger. For $nums[j]+nums[k] > -nums[i]$, I decrease k to find a smaller $nums[k]$. Until $nums[j] + nums[k] = -nums[i]$, we find a pair of numbers which meet our requirement successfully.

Then I will explain how I avoid duplication. First, instead of starting from the beginning, j starts from the number next to i, which is $i + 1$. In this case, each pair will only be the same with the former one. So, I introduce a judgement in the last part of my algorithm. This will make sure that, if same numbers appear in the vector, my algorithm will jump these duplicate numbers. This avoid duplication successfully.

As for the time complexity, the "find" part is $n-1+n-2+......+1 = O(n^2)$. Adding the sort part, the whole time complexity of my algorithm is $O(n^2)$, which meets the requirement. □

3. Equivalence Class

Definition 1 (o-Notation)**.** *Let $f(n)$ and $g(n)$ be functions from the set of natural numbers to the set of nonnegative real numbers. $f(n)$ is said to be $o(g(n))$, written as $f(n) = o(g(n))$, if*

$$\forall c > 0.\exists n_0.\forall n \geq n_0.f(n) < cg(n).$$

An equivalence relation $\mathcal{R}$ on the set of complexity functions is defined as follows:

$$f\mathcal{R}g \text{ if and only if } f(n) = \Theta(g(n)).$$

A complexity class is an equivalence class of $\mathcal{R}$.

The equivalence classes can be ordered by $\prec$ defined as: $f \prec g$ iff $f(n) = o(g(n))$.

Example: $1 \prec \log\log n \prec \log n \prec \sqrt{n} \prec n^{\frac{3}{4}} \prec n \prec n\log n \prec n^2 \prec 2^n \prec n! \prec 2^{n^2}$.

Please order the following functions by $\prec$ and give your explanation:

$$(\sqrt{2})^{\log n}, (n+1)!, ne^n, (\log n)!, n^3, n^{1/\log n}.$$

Solution. The equivalence classes should be ordered as:

$$n^{1/\log n} \prec (\sqrt{2})^{\log n} \prec n^3 \prec (\log n)! \prec ne^n \prec (n+1)!$$

The proof is following:

For $n^{1/\log n}$:

Substitute $\log n$ by a, then we have

$$n^{1/\log n} = (10^a)^{1/a} = 10$$

Therefore, it should be arranged in the lowest place.

For $(\sqrt{2})^{\log n}$ and n^3:

Substitute $\log n$ by a, then we have

$$(\sqrt{2})^{\log n} = (\sqrt{2})^a$$

$$n^3 = (10^a)^3 = 10^{3a}$$

It is clear that $(\sqrt{2})^a = o(10^{3a})$, since $\lim_{a\to\infty} \frac{(\sqrt{2})^a}{10^{3a}}$. Hence, we conclude that $(\sqrt{2})^{\log n} \prec n^3$

For n^3 and $(\log n)!$:

Substitute $\log n$ by a, then we have

$$n^3 = (10^a)^3 = 10^{3a}$$

$$(\log n)! = a!$$

We know that $(10^3)^a \prec a!$. Hence, $n^3 \prec (\log n)!$.

For $(\log n)!$ and ne^n:

We know that

$$(\log n)! \prec 2^{(\log)^2} \prec e^{(\log)^2}$$

Substitute $\log n$ by a, we have

$$e^{(\log)^2} = e^{a^2}$$

$$ne^n = 10^a e^{10^a}$$

Since $e^{a^2} \prec e^{10^a}$, we have $2^{a^2} \prec e^{a^2} \prec 10^a e^{10^a}$. Hence, $(\log n)! \prec ne^n$.

For ne^n and $(n+1)!$:

We know that $e^n \prec n!$, so $n \times e^n \prec (n+1) \times n!$.

Hence, the order should be:

$$n^{1/\log n} \prec (\sqrt{2})^{\log n} \prec n^3 \prec (\log n)! \prec ne^n \prec (n+1)!$$

$\square$