Lab01-Preliminary

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

* Please upload your assignment to website. Contact webmaster for any questions.
* Name: Jin Zhejian Student ID: 517370910167 Email: jinzhejian@outlook.com

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

```
1  // REQUIRES: an integer k
2  // EFFECTS: return the number of times that Line 12 is executed
3  int count(int k)
4  {
5      int count = 0;
6      int n = pow(2,k); // n=2^k
7      while (n>=1)
8      {
9          int j;
10         for (j=0;j<n;j++)
11         {
12             count += 1;
13         }
14         n /= 2;
15     }
16     return count;
17 }
```

Solution. In the **while** loop, the inner **for** loop always executes n times, and n exponentially declines in each **while** loop, because at the end of each **while** loop, **n** is divided by 2. Therefore, the statement **count += 1** is executed precisely $\sum_0^k 2^i = 2^k - 1$ times. Thus, the time complexity of this algorithm is $\Theta(2^k)$. □

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:

```
1  // REQUIRES: an integer array nums of size n
2  // EFFECTS: return a list of triplets, the sum of each triplet
       equals to 0.
3  // MY DESIGN: First, I sort the array ascendingly. Then, I create
       a loop from the 1st number to the 2nd last number, and I set
       two index numbers, one denotes to the 2nd number in each loop
       and the other denotes to the tail. If the sum of the numbers
       the three index refers to equals to 0, I push it to the 2D
       vector; If the sum is greater than 0, I move the tail index
       forward; If the sum is smaller than 0, I move the front index
       backward. If the front index is greater or equal to 0 and no
       sum is fitted, I enter into next loop.
```

```cpp
#include <vector>
#include <iostream>
#include <algorithm>
using namespace std;
vector<vector<int> > findTriplet(vector<int>& nums, int n)
{
    sort(nums.begin(), nums.end());
    vector<vector<int> > res;
    int i=0, j=0, k=n-1;
    for (i=0;i<n-2;i++)
    {
        j = i+1;
        k = n-1;
        while (j < k){
            int sum = nums[i] + nums[j] + nums[k];
            if (sum == 0)
            {
                res.push_back({nums[i], nums[j], nums[k]});
                while (j < k && nums[j] == nums[j+1]){ j++;}
                j++;
            }
            else if (sum > 0){ k--;}
            else { j++;}
        }
        while (nums[i] == nums[i+1]){i++;}
    }
    return res;
}
```

Time complexity:

The time complexity of my program is equal to $O(n^2)$..

First, the sort function in <vector> library applies advanced quick sort, and it's time complexity is $O(\log n)$.

For the **for** loop, it iterates $n-2$ times from $i=0$ to $t<n-2$ in total. In the **for** loop, the worst case happens in the first **if** branch, in which in every **for** loop, we cannot find suitable head and tail index. In this case, given i, $n-j=n-i-1$ times of comparisons between j and k are done, and $\sum_{i=0}^{n-2} n-i-1 = \frac{n(n-1)}{2}$ times of comparisons are needed in the **for** loop, and the time complexity in the **for** loop is $O(n^2)$.

Therefore, the total time complexity of my program is $O(n^2)$.

□

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.

Order:

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

Explanation:

(i)

First, we do simplifications on the first two terms $n^{1/\log n}$ and $(\sqrt{2})^{\log n}$

$$n^{1/\log n} = n^{\log_n^2} = 2$$

$$(\sqrt{2})^{\log n} = 2^{1/2^{\log n}} = 2^{\log n^{1/2}} = \sqrt{n}$$

Obviously, we have:

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

(ii)

Now, we prove that $ne^n \prec (n+1)!$

We have:

$$\frac{ne^n}{(n+1)!} < \frac{ne^n}{n! \cdot n} = \frac{e^n}{n!} = \frac{e \cdot e..... \cdot e}{1 \cdot 2..... \cdot n}$$

When n is greater than 2, $\frac{e}{n} < 1$, so $\frac{e \cdot ...e}{3 \cdot ...dn} \to 0$ when $n \to \infty$, while $\frac{e \cdot e}{1 \cdot 2}$ is bounded.

Therefore, we have:

$$ne^n = O((n+1)!)$$

(iii)

To do the following proves, another Lemma is needed:

Lemma: $n\log n = \Theta(\log n!)$

Proof:

1)First, we prove $\log n! = O(n\log n)$

$$\log(n!) = \sum_{i=1}^{n} \log(i) \leq \sum_{i=1}^{n} \log(n) = \text{nlog}(n) = O(n\log n)$$

2) Second, we prove $n\log n = O(\log n!)$

$$\log(n!) \geq \log\left(\frac{n}{2}\right)^{\frac{n}{2}} = \frac{n}{2}\log\frac{n}{2} = \frac{n}{2}\log n - \frac{n}{2}\log 2$$

When $n \geq 4$, $\frac{n}{2}\log 2 < \frac{1}{4}n\log n$, we have:

$$\log(n!) \geq \frac{n}{2}\log n - \frac{1}{4}n\log n = \frac{1}{4}n\log n$$

So, $n\log n = O(\log n!)$

Therefore, according to 1) and 2), $n\log n = \Theta(\log n!)$.

(iv)

Now we prove another equivalence relation: $(\log n)! = \Theta((\log n)^{\log n})$

Replacing n with $\log n$ in the equation $n\log n = \Theta(\log n!)$ we just proved, we get:

$$\log n \cdot \log\log n = \Theta(\log(\log n)!)$$

Do exponential power of two with base in this equation, we get:

$$(\log n)^{\log n} = \Theta((\log n)!)$$

(v) Now we prove $n^3 \prec (\log n)!$

$$\begin{aligned}\log \frac{n^3}{(\log n)^{\log n}} &= \log n^3 - \log n \cdot \log\log n \\ &= \log n(3 - \log\log n)\end{aligned}$$

When $n \to \infty$, $\log n \to \infty$, and $3 - \log\log n \to -\infty$, so $\log \frac{n^3}{(\log n)^{\log n}} \to -\infty$

So, we have

$$\frac{n^3}{(\log n)^{\log n}} \to 0 \text{ when } n \to \infty$$

Therefore, $n^3 \prec (\log n)^{\log n}$. Also, since $(\log n)^{\log n} = \Theta((\log n)!)$

We get:

$$n^3 \prec (\log n)!$$

(vi)

We prove $(\log n)! \prec ne^n$

$$\begin{aligned}\log \frac{(\log n)^{\log n}}{ne^n} &= \log n \cdot \log\log n - \log n - n \\ &= \log n(\log\log n - 1) - n \\ &\leq \log n \cdot \log n - \log n - n \\ &< \sqrt{n} \cdot \sqrt{n} - \log n - n \\ &= -\log n\end{aligned}$$

When $n \to \infty$, $\log n \to \infty$, so $\log \frac{(\log n)^{\log n}}{ne^n} \to -\infty$

So, we have

$$\frac{(\log n)^{\log n}}{ne^n} \to 0 \text{ when } n \to \infty$$

Therefore, $(\log n)^{\log n} \prec ne^n$. Also, since $(\log n)^{\log n} = \Theta((\log n)!)$

We get:

$$(\log n)! \prec ne^n$$

□