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: Sun Yiwen   Student ID: 517370910213   Email: sunyw99@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 ?? 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. The statements **j<n; j++; count+=1;** all occur $2^k + 2^{k+1} + ... + 2 + 1$ times. Suppose $T(k) = 2^k + 2^{k+1} + ... + 2 + 1$, then if we pick constants c and k_0 so that for any $k > k_0$, $T(k) \leq c \cdot 2^k$, then we can prove $T(k) = O(2^k)$. We choose $c = 2$ and $k_0 = 1$, then for any $k > 1$, $T(k) = 2^k + 2^{k+1} + ... + 2 + 1 < 2^k + 2^{k+1} + ... + 2 + 1 + 1 = 2^{k+1} = 2 \cdot 2^k$. Therefore, the time complexity of the above code is $O(2^k)$. □

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: vector¡vector¡int¿¿ res; int i=0, j=0, k=n-1; for (i=0;i¡n-2;i++) TODO return res;

```
// REQUIRES: an integer array nums of size n
// EFFECTS: return a list of triplets, the sum of each triplet equals to 0.
#include <vector>
vector<vector<int>> findTriplet(vector<int>& nums, int n)
{
    vector<vector<int>> res;
    vector<int> ans;
    vector<int> sorted;
```

```
9      int i, j, k;
10     for (i = 0; i < n; i++) {
11         if (sorted.empty()) sorted.push_back(nums[0]);
12         else {
13             vector<int>::iterator it;
14             for (it = sorted.begin(); it != sorted.end(); ++it) {
15                 if (nums[i] <= *it) {
16                     sorted.insert(it, nums[i]);
17                     break;
18                 }
19                 else if (it == sorted.end() - 1) {
20                     sorted.push_back(nums[i]);
21                     break;
22                 }
23             }
24         }
25     }
26     for (i = 0; i < n - 2; i++) {
27         while (i > 0 && sorted[i] == sorted[i - 1]) i++;
28         j = i + 1;
29         k = n - 1;
30         while (j < k) {
31             if (sorted[i] + sorted[j] + sorted[k] < 0) j++;
32             else if (sorted[i] + sorted[j] + sorted[k] > 0) k--;
33             else {
34                 ans.push_back(sorted[i]);
35                 ans.push_back(sorted[j]);
36                 ans.push_back(sorted[k]);
37                 if (res.empty()) res.push_back(ans);
38                 else if (ans != res.back()) res.push_back(ans);
39                 ans.clear();
40                 j++;
41             }
42         }
43     }
44     return res;
45 }
```

Explain the time complexity of your solution here.
My code can be divided into two parts. The first part(ll.10-25) is where the program sorts the original array in ascending order, which is necessary because we want all the triplets in the result to be unique. The second part(ll.26-43) is where the program finds all unique triplets that meet the requirement.
For the first part, note that the statements **i<n; i++; if (sorted.empty())...else...** all occur n times. The time complexity can be calculated as $O(n)$. For the second part, note that the statements **j<k;** occur $(n-1)+(n-2)+...+2 = \frac{(n-1+2)(n-2)}{2}$ times in the worst case. So the time complexity of the second part can be calculated as $O(n^2)$. Because of the rule that "If $f_1(n) = O(g_1(n))$ and $f_2(n) = O(g_2(n))$, then $f_1(n)+f_2(n) = O(max\{g_1(n), g_2(n)\})$", the time complexity of my entire solution 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*

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

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. (a) Using the formula $a^{log_b c} = c^{log_b a}$, we can get $(\sqrt{2})^{\log n} = n^{log\sqrt{2}} = n^{1/2} = \sqrt{n}$.

(b) Since $(n+1)! - n! = n+1$, we get $n! < (n+1)!$ for $n \geq 1$. With n approaching infinity, $(n+1)! \to n!$. Therefore, $n! \prec (n+1)!$

(c) Since $e > 2$, for $n \geq 1$, $e^n > 2^n$. Since $ne^n - e^n = (n-1)e^n$, we get $ne^n > e^n$ for $n > 1$. Therefore, $2^n \prec ne^n$.
We want to show $\lim_{n\to\infty} \dfrac{ne^n}{n!} = 0$. Let $n > 2e$.
$\dfrac{ne^n}{n!} = \dfrac{e^{n-2e}a^{2e}}{(n-1)(n-2)...(2e)(2e-1)!} < \dfrac{e^{n-2e}e^{2e}}{n^{n-2e}(2e-1)!} < \left(\dfrac{1}{2}\right)^{n-2e} \dfrac{e^{2e}}{(2e-1)!}$
$\dfrac{e^{2e}}{(2e-1)!}$ is constant, $\left(\dfrac{1}{2}\right)^{n-2e} \to 0$ as $n \to \infty$.
Therefore, $ne^n \prec n!$.

(d) According to the Stirling's approximation, the most important term of the Stirling's approximation of the factorial is n^n. So $(\log n)!$ can be approximated as $\log n^{\log n}$, with some extra factors which make it a bit smaller.
$\log n^{\log n} = e^{\log\log n \cdot \log n} = e^{\log n \cdot \log\log n} = n^{\log\log n}$.
Since $\log\log n \prec \log n \prec \sqrt{n}$, we get $\log\log n \cdot \log n \prec \sqrt{n} \cdot \sqrt{n} = n$. Thus, as n approaches infinity, $e^{\log\log n \cdot \log n} < e^n$.
Therefore, $(\log n)! \prec e^n$.
$f(n) = \log\log n$ is monotonically increasing and $f(500) = \log\log 500 \approx 3.164 > 3$. Thus, as $n \to \infty$, $\log\log n > 3$. And we get $\lim_{n\to\infty} \dfrac{n^3}{n^{\log\log n}} = 0$.
Therefore, $n^3 \prec (\log n)!$.

(e) $f(n) = \log n$ is monotonically increasing and $f(4) = \log 4 = 2$. Thus, for $n > 4$, $\log n > 2 \Rightarrow \dfrac{1}{\log n} < \dfrac{1}{2} \Rightarrow n^{1/\log n} < n^{1/2}$. Therefore, $n^{1/\log n} \prec \sqrt{n}$.

Combining all the above with the given Example, we get the final result:

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

Now, I would like to plot graphs using Wolfram Mathematica to further prove my result. The Mathematica code are as follows:

```
Plot[{(Sqrt[2])^(Log2[n]), (n + 1)!, n * E^n, (Log2[n])!, n^3, n^{1/Log2[n]}}, {n, n_min, n_max}, PlotLegends → "Expressions"]
|绘图   |平方根        |以2为底的对数              |以2为底的对数                                   |绘图的图例
```

Note that n_{min} and n_{max} will be replaced with actual numbers to change the scale when plotting. The following graphs are the plot results when the scale is $n \in [0, 500]$, $n \in [0, 1000]$, $n \in [0, 1000000]$, $n \in [2, 2.7]$ correspondingly.

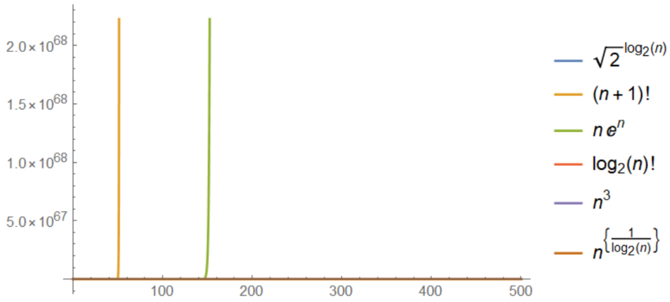

Figure 1: Function plots when $n \in [0, 500]$.

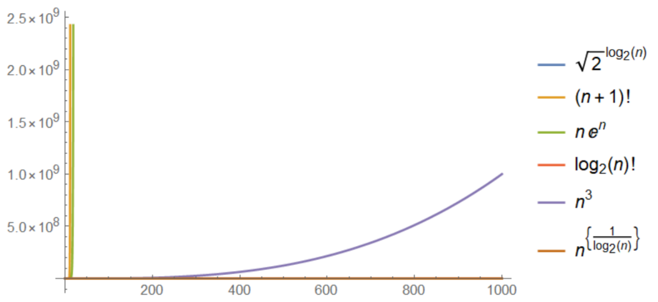

Figure 2: Function plots when $n \in [0, 1000]$.

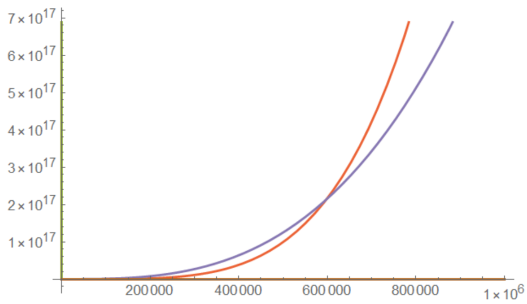

Figure 3: Function plots when $n \in [0, 1000000]$.

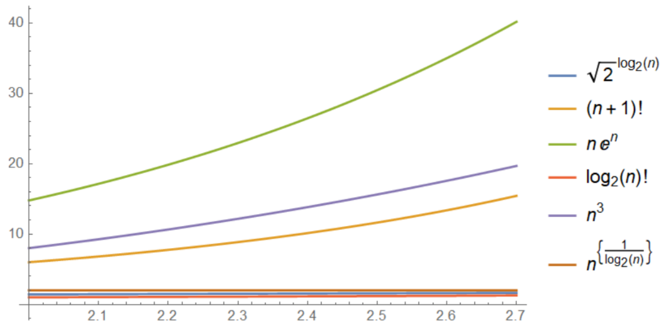

Figure 4: Function plots when $n \in [2, 2.7]$.

As shown in Figure 1 and Figure 2, the growing rate of $(n+1)!$ is greater than ne^n and the growing rate of $(n+1)!$ and ne^n are greater than the other four functions. Although $(\log n)!$ seems to grow slower than n^3 in Figure 2, according to Figure 3, it will catch up with and eventually surpass n^3 around $n = 600000$. As shown in Figure 3, $n^{1/\log n}$ and $(\sqrt{2})^{\log n}$ grow much slower than the other four functions. And according to Figure 4, the growing rate of $(\sqrt{2})^{\log n}$ is a little bit faster than $n^{1/\log n}$.
In conclusion, we get the same result:

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

Again, we cannot assume these four plots give us the correct result for they only show us parts of the functions. But they can be back-up evidence for my mathematical proof in the previous part. $\square$