

算法设计与分析实验报告



实验题目： 最大子段和三种实现算法的时间复杂度分析

姓名： 马天成

学号： 2020211376

日期： 2022-09-27

一、实验环境

1.1 设备规格

设备规格

Legion R7000P2020H

设备名称	PC
处理器	AMD Ryzen 7 4800H with Radeon Graphics 2.90 GHz
机带 RAM	16.0 GB (15.9 GB 可用)
设备 ID	370678B7-1FA2-4C35-8BAB-F92D3429406B
产品 ID	00342-35932-44511-AAOEM
系统类型	64 位操作系统, 基于 x64 的处理器
笔和触控	为 10 触摸点提供笔和触控支持

1.2 操作系统

Windows 规格

版本	Windows 10 家庭中文版
版本号	21H2
安装日期	2022/4/5
操作系统内部版本	19044.2006
序列号	PF2660CA
体验	Windows Feature Experience Pack 120.2212.4180.0

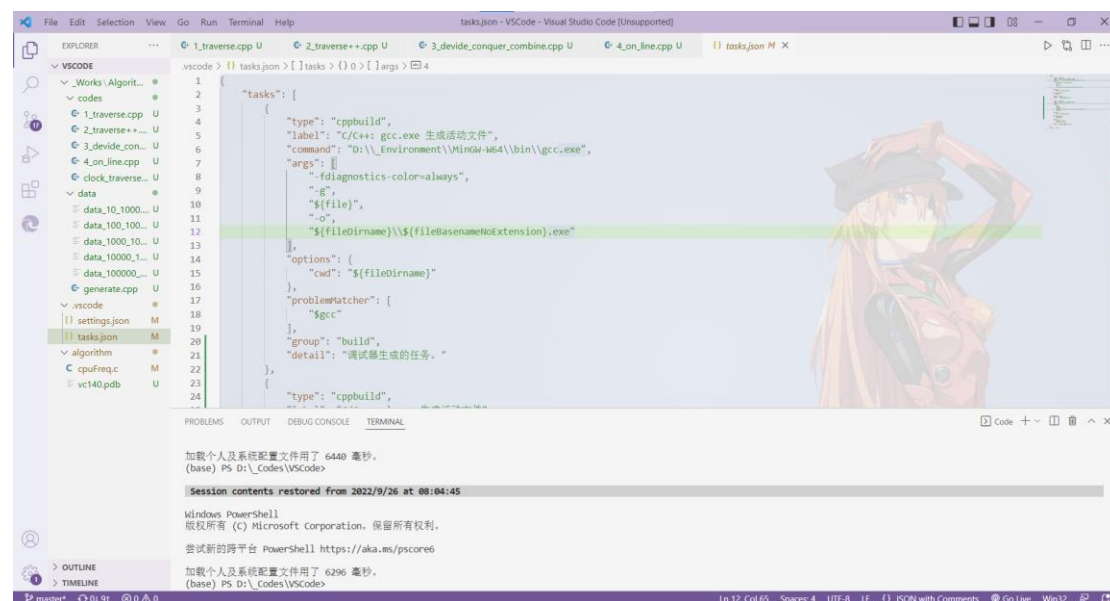
1.3 编程语言&编译器

C++11

```
version : MinGW-W64-builds-4.3.5
user    : nixman
date    : 05.12.2018-10:29:36 AM
```

1.4 开发工具

VSCode



二、实验内容

2.1 实验要求

分别实现课件中给出的基于暴力枚举、分治法和动态规划方法实现的最大字段和算法程序，测试在不同数据规模下，三种算法的时间复杂度。

2.2 实验目的

1. 理解算法时间复杂度的评价方法，初步了解不同的算法策略对算法性能的影响程度；
2. 掌握算法时间复杂度测试的基本流程。

2.3 实验内容

2.3.1 数据准备

进行数据的生成，采用随机数的思路，生成“dataSize \n data_1 data_2 …”的文件形式：

```
_Works > Algorithm > Lab1 > generate.cpp > main()
6  int main()
7
8  // get path
9  char buffer[100];
10 getcwd(buffer, 100);
11 std::string path;
12 for(int i=0; buffer[i] != '\0'; i++)
13     path.push_back(buffer[i]);
14 std::cout << path << std::endl;
15
16
17 // get parameters
18 int upper, size;
19 std::string name;
20 std::cout << "Please input the upper of data:" << std::endl;
21 std::cin >> upper;
22 std::cout << "Please input the size of data:" << std::endl;
23 std::cin >> size;
24 std::cout << "Please input the name of file:" << std::endl;
25 std::cin >> name;
26
27
28 // generate
29 path.append("\\"+name+".txt");
30 std::cout << path << std::endl;
31 std::ofstream fout(path);
32 fout << size << "\n";
33
34 srand((unsigned int)time(NULL));
35 for(int i=0; i < size; i++) {
36     int randomNum=rand()%(2*upper)-upper;//产生[-upper, upper]的随机数
37     fout << randomNum << " ";
38 }
39 fout.close();
40
41 return 0;
42
```

- 进行当前 path 的读取，以便生成文件
- 进行数据规模，数据 random，文件名称的读取（从 io）
- 生成数据，写入文件：

1. 生成文件路径并打开文件：

```
path.append("\\"+name+".txt");
std::cout << path << std::endl;
std::ofstream fout(path);
```

2. 写入 dataSize 和 data 并关闭文件：

```
fout << size << "\n";

srand((unsigned int)time(NULL));
for(int i=0; i < size; i++) {
    int randomNum=rand()%(2*upper)-upper;//产生[-upper, upper]的随机数
    fout << randomNum << " ";
}
fout.close();
```

3. 根据实验要求，选择实现 5 个文件；数据范围都在 $[-1000, 1000]$

data_10_1000.txt	U
data_100_1000.txt	U
data_1000_1000.txt	U
data_10000_1000.txt	U
data_100000_1000.txt	U

命名格式：dtat_dataSize_dataRandom.txt

形如：data_10_1000.txt

_Works > Algorithm > Lab1 > data_10_1000.txt										
1	10									
2	348	-28	930	-762	979	751	767	436	-14	150

2.2 代码编写

根据精简原则，我们这里只展示运算函数部分，不考虑主函数。

2.2.1 暴力枚举

```
int MaxSubsequenceSum(const int data[], int size)
{
    int ans = 0;
    // generate start-i and end-j
    for(int i=0; i < size; i++) {
        for(int j=i; j < size; j++) {
            // calculate sum
            int tmpAns = 0;
            for(int k=i; k <= j; k++)
                tmpAns += data[k];
            // upgrade ans
            if(tmpAns > ans) {
                ans = tmpAns;
                // std::cout << i << " " << j << " " << ans << "\n";
            }
        }
    }
    return ans;
}
```

非常简单的三重循环。按照课上所说，枚举每一个可能出现的 start 和 end 区间，计算之间的字段和，获得最大值。

只不过这里需要多用一个 int 存储空间存储当前循环计算得出的字段和，并且在循环中一直维护最大值 ans，最后在枚举完成后返回 ans 即可。

2.2.2 暴力枚举优化

```
int MaxSubsequenceSum(const int data[], int size)
{
    int ans = 0;
    // generate start-i and end-j
    for(int i=0; i < size; i++) {
        // calculate sum
        int tmpAns = 0;
        for(int j=i; j < size; j++) {
            // add end
            tmpAns += data[j];
            // upgrade ans
            if(tmpAns > ans) {
                ans = tmpAns;
                // std::cout << i << " " << j << " " << ans << "\n";
            }
        }
    }
    return ans;
}
```

减小一层循环，用之前短一个的子段和来算出现在新的子段和，充分利用了之前子段和的计算结果，把复杂度从 $O(n^3)$ 到 $O(n^2)$ 。

2.2.3 分治法

```
51 int MaxSubsequenceSum(const int data[], int size)
52 {
53     int ans = 0;
54
55     // initialize each queue unit for devide
56     std::queue<int*> myQueue;
57     for(int i=0; i < size; i++) {
58         /*
59          * each unit has four respo:
60          * { ans, l_max, r_max, sum }
61          */
62         int* cur = new int[4];
63         cur[0] = (data[i]>0) ? data[i] : 0;
64         cur[1] = data[i]; // has one number at least
65         cur[2] = data[i]; // has one number at least
66         cur[3] = data[i];
67         myQueue.push(cur);
68     }
69 }
```

```

70 // combine
71 int capacity = myQueue.size();
72 while(capacity > 1) {
73     // size is odd, select first one to pop and push
74     if(capacity&1) {
75         myQueue.push(myQueue.front());
76         myQueue.pop();
77     }
78
79     // select pair to combine
80     for(int i=0; i < capacity/2; i++) {
81         int* tmp_1 = myQueue.front();
82         myQueue.pop();
83         int* tmp_2 = myQueue.front();
84         myQueue.pop();
85
86         // generate { ans, l_max, r_max, sum } into tmp_1
87         tmp_1[0] = max(tmp_1[0], tmp_2[0]);
88         tmp_1[0] = max(tmp_1[0], tmp_1[2]+tmp_2[1]);
89         tmp_1[1] = max(tmp_1[1], tmp_1[3]+tmp_2[1]);
90         tmp_1[2] = max(tmp_2[2], tmp_2[3]+tmp_1[2]);
91         tmp_1[3] = tmp_1[3] + tmp_2[3];
92
93         // delete tmp_2 and push tmp_1
94         delete tmp_2;
95         myQueue.push(tmp_1);
96     }
97
98     capacity = myQueue.size();
99 }
100
101 return myQueue.front()[0];
102 }

```

分治法是最巧妙的一个方法。

感谢顾佳澜同学的分享。在她的维护四个变量{ *ans*, *l_max*, *r_max*, *sum* }的基础上,写出了这一份代码。两个单元四个属性的 Combine 思路如上图所示。

思路就是通过左右结合计算相应的四个属性,来从最小单元维护到最后整体。最小问题就是如何结合两个单元四个属性。其解法如上图代码标注所示。

此外我加入了队列思路:

- 在 combine 的时候,从 queue 中 pop 两个单元(此处为指针),然后合并两个单元的属性,再把一个覆盖新答案后的单元 push 进的队列,并释放另一个没用的单元空间。
- 通过当前子段 Size 奇偶数的区别(奇数意味着有一个不用 combine,那就先把第一个 pop 出来再 push 进去),来进行相邻子段循环叠加到最后只剩一个子段(原段)。

此外,我抛弃了 vector 来作为数据结构,因为 vector 即使 erase,空间只增不减。

combine 核心代码:

```
// combine
int capacity = myQueue.size();
while(capacity > 1) {
    // size is odd, select first one to pop and push
    if(capacity&1) {
        myQueue.push(myQueue.front());
        myQueue.pop();
    }

    // select pair to combine
    for(int i=0; i < capacity/2; i++) {
        int* tmp_1 = myQueue.front();
        myQueue.pop();
        int* tmp_2 = myQueue.front();
        myQueue.pop();

        // generate {ans, l_max, r_max, sum} into tmp_1
        tmp_1[0] = max(tmp_1[0], tmp_2[0]);
        tmp_1[0] = max(tmp_1[0], tmp_1[2]+tmp_2[1]);
        tmp_1[1] = max(tmp_1[1], tmp_1[3]+tmp_2[1]);
        tmp_1[2] = max(tmp_2[2], tmp_2[3]+tmp_1[2]);
        tmp_1[3] = tmp_1[3] + tmp_2[3];

        // delete tmp_2 and push tmp_1
        delete tmp_2;
        myQueue.push(tmp_1);
    }

    capacity = myQueue.size();
}
```

2.2.4 动态规划

```
int MaxSubsequenceSum(const int data[], int size)
{
    int ans = 0;
    int sum = 0;
    // scan data
    for(int i=0; i < size; i++) {
        // calculate sum
        sum += data[i];
        // upgrade ans
        if(sum > ans)
            ans = sum;
        // throw away minus sequence
        if(sum < 0)
            sum = 0;
    }
    return ans;
}
```

这个思路直接以最简的方式来进行计算。它建立在得出了最大子段和不需要负数字段和这一强结论，使得整个过程变得极为简单。

dp 是最好的方法，只需要扫描一次，用的空间也只有一个 int。

2.3 实践测试方式

对于随机生成的 random 数据可控的变量为 random 范围和数据量大小。所以我们先考虑 random 范围这个变量对于测试的影响：分别用算法 4 测试了以下五个文件：

```
Get data in 0.0016508s.  
Answer is 188  
Size: 10000, calculate answer in 0.0004247s.
```

```
Get data in 0.0017167s.  
Answer is 6994  
Size: 10000, calculate answer in 0.0006174s.
```

```
Get data in 0.0023528s.  
Answer is 29971  
Size: 10000, calculate answer in 0.0002393s.
```

```
Get data in 0.0026595s.  
Answer is 77599  
Size: 10000, calculate answer in 0.000672s.
```

```
Get data in 0.0023057s.  
Answer is 0  
Size: 10000, calculate answer in 0.0005268s.
```

```
data_10000_10.txt U  
data_10000_100.txt U  
data_10000_1000.txt U  
data_10000_10000.txt U  
data_10000_100000000.txt U
```

本地测试采用了不同的 random 范围，发现影响不大，运行速度比较随机，且没有什么数量级上的差距。所以测试用例都取 random=1000，可认为合理。

2.3.1 clock 测试

要获得运行时间，就设置时间戳，在进入函数和返回函数两个地方获得时间：

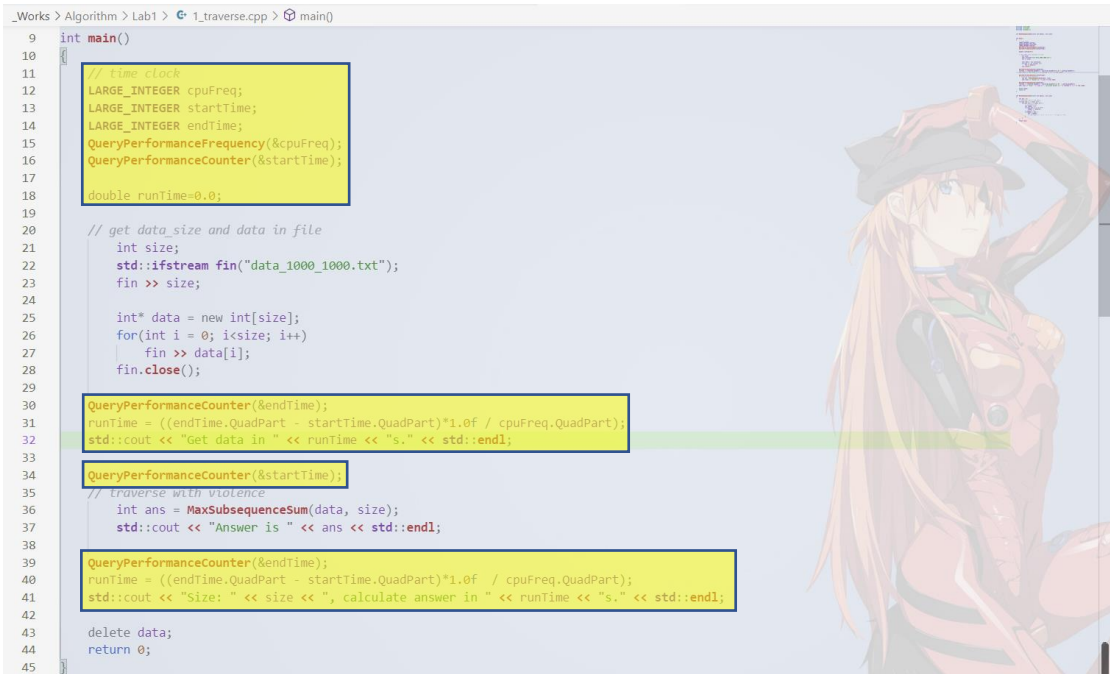
```
1  _Works > Algorithm > Lab1 > 2_traverse+.cpp > main()  
9  int main()  
10  
11  // time clock  
12  clock_t start=clock(), finish;  
13  double duration;  
14  
15  
16  // get data_size and data in file  
17  int size;  
18  std::ifstream fin("data_10000_1000.txt");  
19  fin >> size;  
20  
21  int* data = new int[size];  
22  for(int i = 0; i < size; i++)  
23  fin >> data[i];  
24  fin.close();  
25  
26  finish = clock();  
27  duration = (double)(finish - start) / CLOCKS_PER_SEC;  
28  std::cout << "Get data in " << duration << "s." << std::endl;  
29  
30  
31  start=clock();  
32  // traverse with violence  
33  int ans = MaxSubsequenceSum(data, size);  
34  std::cout << "Answer is " << ans << std::endl;  
35  
36  finish = clock();  
37  duration = (double)(finish - start) / CLOCKS_PER_SEC;  
38  std::cout << "Size: " << size << ", calculate answer in " << duration << "s." << std::endl;  
39  
40  delete data;  
41  return 0;  
42
```

一共有两次计算：

1. 进入 main 函数之后到打开文件并读数据到内存完毕的时间；
2. 在最大字段和函数运行完之后计算整个时间。

2.3.2 CPU 计数器测试

看到数量级之后，我打算尝试用更精准的方法进行时间测量。鉴于之前有过做帧率的经历，我掏出了尘封已久的 cpuFreq 代码。通过 CPU 计数单元来获得最精准的测量方式。



```
9 int main()
10
11 // time clock
12 LARGE_INTEGER cpuFreq;
13 LARGE_INTEGER startTime;
14 LARGE_INTEGER endTime;
15 QueryPerformanceFrequency(&cpuFreq);
16 QueryPerformanceCounter(&startTime);
17
18 double runTime=0.0;
19
20 // get data size and data in file
21 int size;
22 std::ifstream fin("data_1000_1000.txt");
23 fin >> size;
24
25 int* data = new int[size];
26 for(int i = 0; i<size; i++)
27     fin >> data[i];
28 fin.close();
29
30 QueryPerformanceCounter(&endTime);
31 runTime = ((endTime.QuadPart - startTime.QuadPart)*1.0f / cpuFreq.QuadPart);
32 std::cout << "Get data in " << runTime << "s." << std::endl;
33
34 QueryPerformanceCounter(&startTime);
35 // traverse with violence
36 int ans = MaxSubsequenceSum(data, size);
37 std::cout << "Answer is " << ans << std::endl;
38
39 QueryPerformanceCounter(&endTime);
40 runTime = ((endTime.QuadPart - startTime.QuadPart)*1.0f / cpuFreq.QuadPart);
41 std::cout << "Size: " << size << ", calculate answer in " << runTime << "s." << std::endl;
42
43 delete data;
44 return 0;
45
```

按照相同的思路，我们在相同的位置采用时间戳获得。但是通过 CPU 每秒计数来获得相应的运行时间。

```
runTime =
((endTime.QuadPart - startTime.QuadPart)*1.0f / cpuFreq.QuadPart);
```

这样时间测试的精准度就会高很多。

2.3.3 实验记录

	Algorithm_1	Algorithm_2	Algorithm_3	Algorithm_4
10	0.0005007s	0.0004677s	0.0006279s	0.0004522s
100	0.0008349s	0.0005496s	0.0007359s	0.0004102s
1000	0.318669s	0.0018608s	0.0004758s	0.0003931s
10000	310.168s	0.160161	0.0016142s	0.000649s
100000	/	16.2342s	0.0149988s	0.0007848s

10:

```
Get data in 0.0002761s.
Answer is 3557
Size: 10, calculate answer in 0.0005007s.
```

```
Get data in 0.0002179s.
Answer is 3557
Size: 10, calculate answer in 0.0004677s.
```

```
Get data in 0.000217s.  
Answer is 3557  
Size: 10, calculate answer in 0.0006279s.
```

```
Get data in 0.0002413s.  
Answer is 3557  
Size: 10, calculate answer in 0.0004522s.
```

100:

```
(base) PS D:\_Codes\VSCode> cd "d:\_Codes\  
Get data in 0.0002998s.  
Answer is 4088  
Size: 100, calculate answer in 0.0008349s.  
(base) PS D:\_Codes\VSCode\_Works\Algorith  
Get data in 0.0002387s.  
Answer is 4088  
Size: 100, calculate answer in 0.0005496s.  
(base) PS D:\_Codes\VSCode\_Works\Algorith  
) { .\3_devide_conquer_combine }  
Get data in 0.0002366s.  
Answer is 4088  
Size: 100, calculate answer in 0.0007359s.  
(base) PS D:\_Codes\VSCode\_Works\Algorith  
Get data in 0.0002431s.  
Answer is 4088  
Size: 100, calculate answer in 0.0004102s.
```

1000:

```
(base) PS D:\_Codes\VSCode> cd "d:\_Codes\  
Get data in 0.0004667s.  
Answer is 23583  
Size: 1000, calculate answer in 0.318669s.  
(base) PS D:\_Codes\VSCode\_Works\Algorith  
Get data in 0.0004702s.  
Answer is 23583  
Size: 1000, calculate answer in 0.0018608s.  
(base) PS D:\_Codes\VSCode\_Works\Algorith  
) { .\3_devide_conquer_combine }  
Get data in 0.0005111s.  
Answer is 23583  
Size: 1000, calculate answer in 0.0004758s.  
(base) PS D:\_Codes\VSCode\_Works\Algorith  
Get data in 0.0004597s.  
Answer is 23583  
Size: 1000, calculate answer in 0.0003931s.
```

10000:

```
Get data in 0.0018786s.  
Answer is 29971  
Size: 10000, calculate answer in 310.168s.
```

```
Get data in 0.0018447s.  
Answer is 29971  
Size: 10000, calculate answer in 0.160161s.
```

```
Get data in 0.0044767s.  
Answer is 29971  
Size: 10000, calculate answer in 0.0016142s.
```

```
Get data in 0.0018429s.  
Answer is 29971  
Size: 10000, calculate answer in 0.000649s.
```

100000:

```
Get data in 0.0148538s.  
Answer is 65946  
Size: 100000, calculate answer in 16.2342s.
```

```
Get data in 0.0140524s.  
Answer is 65946  
Size: 100000, calculate answer in 0.0149988s.
```

```
Get data in 0.0142509s.  
Answer is 65946  
Size: 100000, calculate answer in 0.0007848s.
```

三、出现问题及解决

Q1. 生成随机测试数据文件的问题

首先是生成数据用什么形式。最终采用如下文件形式：

```
_Works > Algorithm > Lab1 > data_10_1000.txt  
1 10  
2 348 -28 930 -762 979 751 767 436 -14 150
```

其次是随机数生成数据溢出的问题：

```
int randomNum=rand()%(2*upper)-upper;//产生[-upper, upper]的随机数
```

这个式子可以说是非常想当然的一个式子。完全没有考虑到数据类型。当输入的 upper=1000000000 时，文件根本就是全负数。

Q2. 分治算法如何确定维护的属性，以及 combine

非常感谢董佳澜同学的分享，要不然我真想不出来。虽然想法比较简单，但 combine 这种思维方式对于我来说确实是比较新颖。

Q3. 如何测试

对于数据集的变量把控以及对于测试的情况分析（上文均有），在此前并不是很熟悉。但在经过本次实验后，我也知道了如何去测试，如何去设计测试，以及如何用限定变量的方法来分析变量的影响度。

Q4. 没有考虑到多样例测试

导致测试文件名是在程序中写死，每次调用程序测试不同数据集都需要重新编译运行。这类情况以后一定要避免。

四、总结

4.1 时间和空间复杂度分析

	Algorithm_1	Algorithm_2	Algorithm_3	Algorithm_4
时间复杂度	$O(n^3)$	$O(n^2)$	$O(n \cdot \log n)$	$O(n)$
空间复杂度	$O(1)$	$O(1)$	$O(n)$	$O(1)$

对于这 4 个算法来说，空间复杂度应该是确定的，但时间复杂度反映出来的依旧有值得探讨的地方：

算法三采用了比较复杂的数据结构。所以再数据规模小的时候，还不一定能跑过算法一算法二。但是数据规模变大之后，这一点开销就会比数据规模带来的影响更小了。

4.2 实验经验总结

1. 了解了优化算法的重要性：在数据量庞大的情况下，不同的时间复杂度会导致极大的效率差别。
2. 算法需要循序渐进。在这四种算法中，我认为 1-2-4 是循序渐进的：
 - 建立在顺序读取枚举计算的算法一上，通过小子段推大子段这一技巧，使得时间复杂度降了一个量级；

- 在顺序读取的思路中，我们限定了一个已经推出的强结论，使得算法再次降了一个复杂度。
- 分治算法反而像是插进来的，它不是顺序读取的思路。它属于分治思想，但由于时间复杂度所以将它放在算法 3。

3. 算法复杂度测试的基本流程：

- 生成不同规模不同性质的数据集，对算法进行全方面的测试。
- 比较不同算法的复杂度，尤其在大规模数据集下的时间，分析优劣。(在本次实验中，分治法在很大程度上有一部分数据结构的开销，所以小规模数据中算法 3 甚至不如算法 1 算法 2)