目录

一、问题描述

完成钻石矿工的算法

要求：

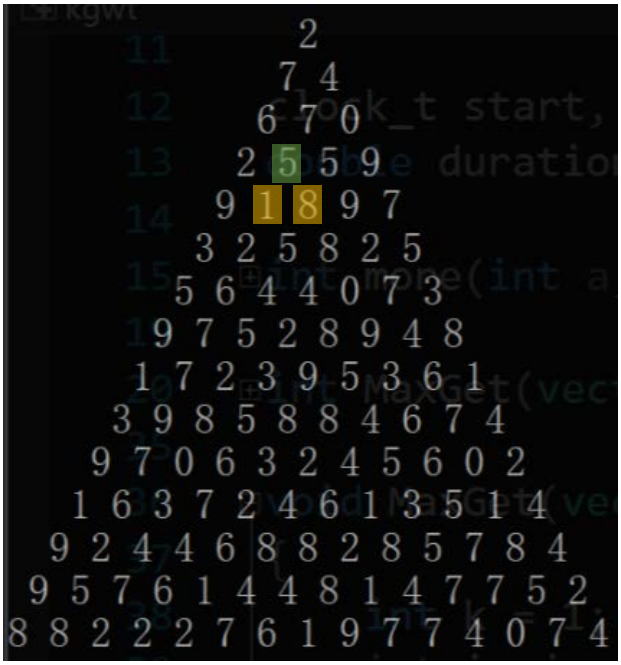
- 设计动态规划算法，描述最优子结构，在报告中写出设计思路；
- 编程序实现上述算法，并完成测试
- 分析算法的时间复杂度，并与分治法进行对比

二、解决思想

2.1 分治法

分治法采用的是自顶而下的递归思想。每一层的最大权值和都是下一层紧挨自己的两个数的较大的权值和加上本身。这种方法思考简便，易于实现。但是大量的递归会不断建立栈、消除栈，耗费很多性能。同时，在计算过程中，有非常多的重复计算。

例如：



计算黄色框中到达 1，8 两点时最大收益时，都要计算绿色框中到达 5 这个点时的最大收益。然而，采用递归的分治法，绿色 5 这一点的值并没有被记录。也就是说，同一点的值最少也要计算两次。当计算最下面一层时，上面所有的情况都要重新计算。

这种情况十分类似于之前讲过的 Fibonacci 数列的计算方法。采用递归的方法计算 Fibonacci 数列时，要重复计算先前的结果很多次。在这里，两者出现的问题是一模一样的。

2.2 动态规划 $O(n^2)$

学习递归解决 Fibonacci 数列时，老师曾留过一道思考题，线性 Fibonacci 数列求解算法。当时的想法是将已经算过的数存储起来 ，这样的话就可以避免重复计算同一个结果的尴尬情况。动态规划的方法思想我认为于此不谋而合。使用循环，而不是递归，可以存储到达每个节点时的最大收益。

动态规划采用了与分治递归完全相反的思想，采用了自底向上的方法。

2.2.1 状态转移方程

D(x , y) = max { D(x , y) , D(x , y) } + a(x , y)

D(N , k) = a(n , k)

2.2.2 最优子结构

每向上一层，都将下一层紧挨自己的两个数比较，将较大的那个数与本身相加。如此，每一层的最优解均包含下一层的最优解，即满足动态规划问题的最优子结构性质。

三、具体实现

3.1 计时函数

```
#include <time.h>
clock_t start, stop; /* clock_t is a built-in type for processor time (ticks) */
double duration; /* records the run time (seconds) of a function */
int main ( )
{/* clock() returns the amount of processor time (ticks) that has elapsed since the program began running */
   start = clock(); /* records the ticks at the beginning of the function call */
   function(); /* run your function here */
   stop = clock(); /* records the ticks at the end of the function call */
   duration = ((double)(stop - start))/CLK_TCK;
   /* CLK_TCK is a built-in constant = ticks per second */
   return 1;
}
```

3.2 随机数集生成

```
srand((unsigned)time(NULL));
ofstream data_in("data.txt",ios::trunc);/*每次删除并重建数据集*/
cin>>len;
for(i=1;i<=len;i++)
{
    x=rand()%len;
    data_in<<x<<" ";
}
data_in.close();/*创建数据集完成，保存并退出*/
```

3.3 分治法

```
int MaxGet(vector<vector<int> > a,
           vector<vector<int> > &max,
            int max_x, int max_y, int height)
{
    if (height == 1)
        return a[0][0];
    else
    {
        if (max_y == 0)
```

```
            return MaxGet(a, max, max_x - 1, max_y, height - 1)
                        + a[max_x][max_y];
        else if (max_y == height - 1)
            return MaxGet(a, max, max_x - 1, max_y - 1, height - 1)
                        + a[max_x][max_y];
        else
            return more(MaxGet(a, max, max_x - 1, max_y - 1, height - 1)
                            + a[max_x][max_y],
                        MaxGet(a, max, max_x - 1, max_y, height - 1)
                            + a[max_x][max_y]);
    }
}
```

3.4 动态规划

```
void MaxGet(vector<vector<int> > a, vector<vector<int> > &max, int height)
{
    int k = 1;
    int i, j;
    max[0][0] = a[0][0];
    for (i = 1; i <= height - 1; i++)
    {
        for (j = 0; j <= k; j++)
        {
            if (j == 0)
                max[i][j] = max[i - 1][j] + a[i][j];
            else if (j == k)
                max[i][j] = max[i - 1][j - 1] + a[i][j];
            else
                max[i][j] = more(max[i - 1][j - 1] + a[i][j],
                                 max[i - 1][j] + a[i][j]);
        }
        k++;
    }
}
```

3.5 寻找路径

```
vector<string> GetPath(vector<vector<int> > max)
{
    vector<string> path;
    int height = max.size();
    int i;
    int best = max[height - 1][0], best_i = 0;
    for (i = 1; i <= height - 1; i++)
    {
        if(max[height - 1][i] > best)
        {
```

```
            best = max[height - 1][i];
            best_i = i;
        }
    }

    for (i = height - 1; i >= 1; i--)
    {
        if (best_i == 0)
        {
            path.push_back("left");
            best_i = 0;
        }

        else if (best_i == i)
        {
            path.push_back("right");
            best_i = i - 1;
        }
        else
        {
            if (max[i - 1][best_i] > max[i - 1][best_i - 1])
                path.push_back("left");
            else
            {
                path.push_back("right");
                best_i--;
            }
        }
    }
    return path;
}
```

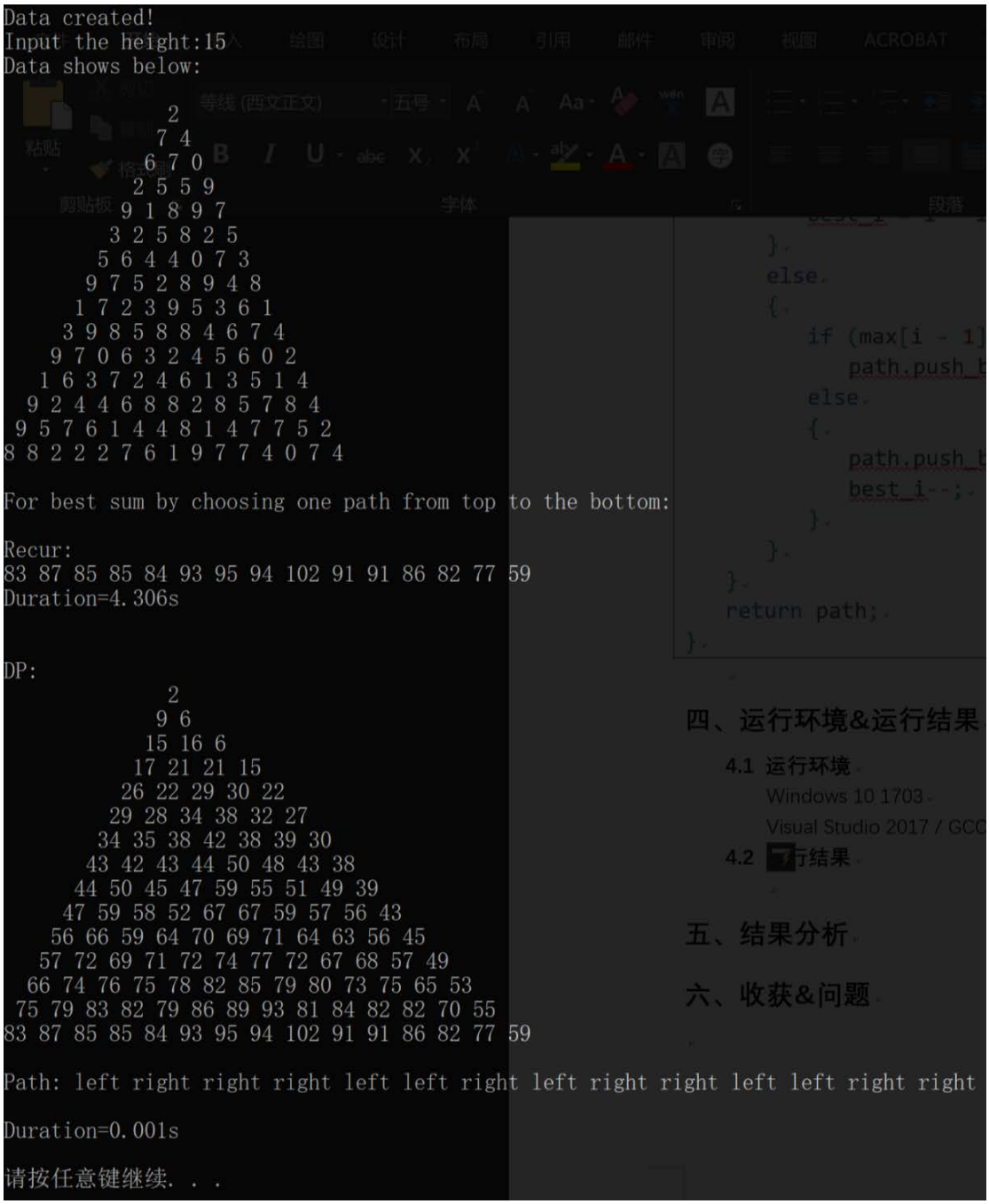
四、运行环境&运行结果

4.1 运行环境

Windows 10 1703

Visual Studio 2017 / GCC

4.2 运行结果



五、结果分析

可以看出，仅仅 15 层金字塔时，动态规划算法就已经比分治算法好出很多，分别耗时 0.001s 和 4.306s。这说明，在分治算法中，不断计算之前计算过的相同结果，消耗很大，尤其对于大数据量的情况。这一说明，详见 2.1 分治法中的具体说明。

从查找路径的方法中我们也可以看出，查找路径是由底层向高层，，也就是说，面对一个已知结果的金字塔，自底向上才不会很麻烦，而分治策略恰恰面对的是一个未知最大收益的金字塔，所以效率很低。

六、收获&问题

6.1 收获

掌握了使用 vector 定义二维数组的方法，同时对最优子结构、动态规划的状态转移有了更深刻的理解。同时也认识到分治算法的缺陷，虽然很好想，但是避免不了重复计算的问题。而且，也要避免使用递归。编译器建立、删除栈的消耗很大。应该尝试手动维护一个栈来进行操作。

6.2 问题

生成随机数的时候，输出的文件有的时候是乱码，有的时候不是。这个问题让我苦恼不堪。使用 Notepad++软件打开乱码的 txt 文件却能正常打开。但是这并不影响随机数据集的生成、销毁、重新生成和程序读取。鉴于其他软件可以顺利打开，我认为应该是系统的问题。或者是程序编码的问题。我分别试过 ANSI、Unicode 和 GB2312 编码，结果都不理想。