算法设计与分析实验报告

实验题目：蒙图版钻石矿工算法的设计与分析

姓名：马天成

学号：2020211376

日期：2022-11-14

目录

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

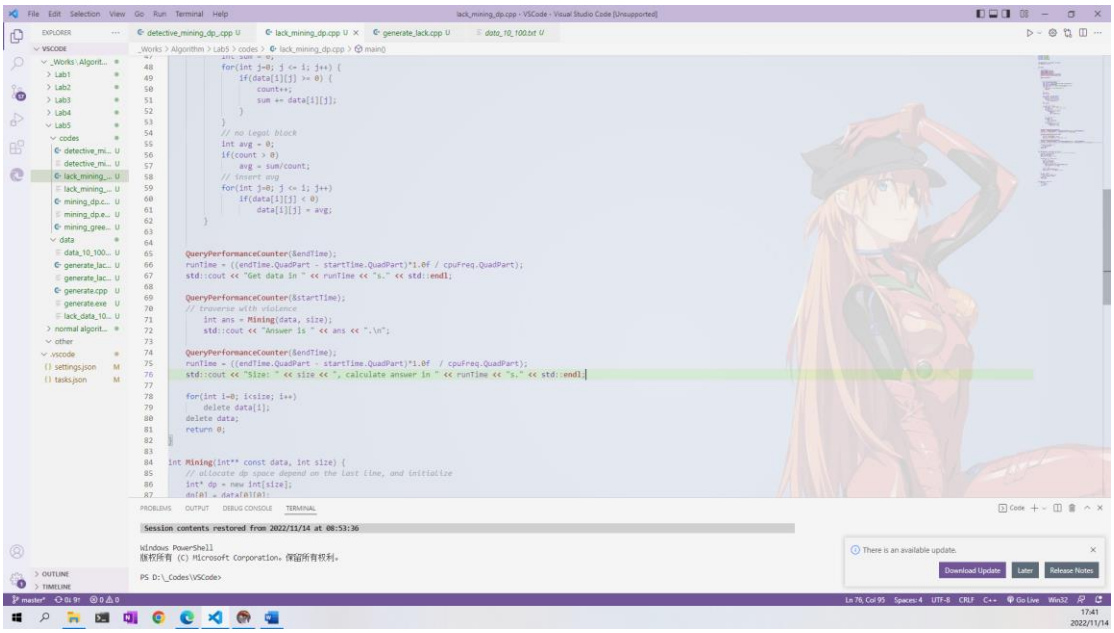
版本	Windows 10 家庭中文版
版本号	21H2
安装日期	2022/4/5
操作系统内部版本	19044.2006
序列号	PF2660CA
体验	Windows Feature Experience Pack 120.2212.4180.0

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



二、实验内容

2.1 实验目的

- 理解动态规划算法的策略，掌握 DP 算法避免重复计算的方法；
- 掌握基于最优子结构递推分解原问题和子问题的基本方法
- 掌握自底向上的 DP 算法的实现方法；
- 理解基于全局动态规划的 DP 算法在实际应用中的局限性，掌握 基于局部动态规划和贪心策略相结合的 DP 算法的设计方法；

2.2 实验内容及要求

1. 贪心算法
2. 动态规划算法（附带贪心策略）
3. 蒙图版（探测）
4. 蒙图版（缺省）

设计测试数据集，编写测试程序，用于测试：

a) 正确性：所实现算法的正确性；

b) 算法复杂性：分析评价各个算法在算法复杂性上的表现；（最差情 况、平均情况）

2.3 输入设计

规定输入为：第一行数据规模；后面跟着规模行的三角矩阵代表金字塔；数据类型都是 int。

2.4 主函数功能设计

主要功能为读取数据、调用函数处理、和输出处理结果&时间。

```cpp
{
    // time clock
    LARGE_INTEGER cpuFreq;
    LARGE_INTEGER startTime;
    LARGE_INTEGER endTime;
    QueryPerformanceFrequency(&cpuFreq);
    QueryPerformanceCounter(&startTime);

    double runTime=0;

    // get data_size and data in file
        std::string dataFileName;
        std::cout << "Please input the name of dataFile:\n";
        std::cin >> dataFileName;
        std::ifstream fin("../data/"+dataFileName+".txt");
        if(!fin) {
            std::cout << "File not exist.\n";
            exit(0);
        }

        int size;
        fin >> size;

        int** data = new int*[size];
        for(int i=0; i<size; i++) {
            data[i] = new int[i+1];
            for(int j=0; j <= i; j++)
                fin >> data[i][j];
        }
        fin.close();

    QueryPerformanceCounter(&endTime);
    runTime = ((endTime.QuadPart - startTime.QuadPart)*1.0f / cpuFreq.QuadPart);
    std::cout << "Get data in " << runTime << "s." << std::endl;

    QueryPerformanceCounter(&startTime);
    // traverse with violence
        int ans = Mining(data, size);
        std::cout << "Answer is " << ans << ".\n";

    QueryPerformanceCounter(&endTime);
    runTime = ((endTime.QuadPart - startTime.QuadPart)*1.0f  / cpuFreq.QuadPart);
    std::cout << "Size: " << size << ", calculate answer in " << runTime << "s." << std::endl;

    for(int i=0; i<size; i++)
        delete data[i];
    delete data;
    return 0;
}
```

主要结构如上图。就不过多解释了。都是套用之前实验的模板。

2.5 子函数-贪心算法

mining_greedy.cpp U

这是最简单的一个从上到下贪心算法。就是从开头位置一直往下找最好的。这种方法无法保证选出最大的。

```cpp
void Mining(int** const data, int size, int i, int j, int sum, int& ans) {
    if(i == size)
        return;

    // left
    if(data[i][j] > data[i][j+1]) {
        if(sum+data[i][j] > ans)
            ans = sum+data[i][j];
        Mining(data, size, i+1, j, sum+data[i][j], ans);
    }
    // right
    else if(data[i][j] < data[i][j+1]) {
        if(sum+data[i][j+1] > ans)
            ans = sum+data[i][j+1];
        Mining(data, size, i+1, j+1, sum+data[i][j], ans);
    }
    // left and right
    else {
        if(sum+data[i][j] > ans)
            ans = sum+data[i][j];
        Mining(data, size, i+1, j, sum+data[i][j], ans);
        Mining(data, size, i+1, j+1, sum+data[i][j], ans);
    }
}
```

因为可能左右相等，所以采用了递归更新答案的方法。这样保证找的比较靠谱一些。

随便运行了一下结果：

```
Please input the name of dataFile:
data_10_100
Get data in 5.90262s.
Answer is 580.
Size: 10, calculate answer in 0.0010268s.
```

2.6 子函数-动态规划贪心思想

这里动态规划作为重点讲解，其也是后面两个代码的基础。

- 辅助空间的选择

动态规划我的思路是 dp 辅助空间最小为最后一行的元素规模，dp 到·最后代表着从第一行开始到最后一行该 block 走出来累积的最大价值。

- dp 的思路

我一开始的思路是从下往上。因为之前开始学习 dp 的时候思路应该是这样的。但现在回想其实就是为了方便开辅助空间。但我这次有了辅助空间的选择，那就没必要自底向上，那就自顶向下，然后逐个覆盖就可以了。不过要注意行覆盖的顺序：必须是从右往左。因为输入代表着你得从最右边开始覆盖，否则覆盖的数据会影响到后面的东西。

- 算法编写（主要部分为状态转移）

```
62  int Mining(int** const data, int size) {
63      // allocate dp space depend on the last line, and initialize
64      int* dp = new int[size];
65      dp[0] = data[0][0];
66      std::cout  << "\t" << dp[0] << "\n";
67
68      for(int i=1; i < size; i++) {
69          // the right
70          dp[i] = dp[i-1]+data[i][i];
71          // the remain
72          for(int j=i-1; j >= 1; j--) {
73              dp[j] = Max(dp[j-1], dp[j])+data[i][j];
74          }
75          // the left
76          dp[0] += data[i][0];
77
78          // print
79          for(int j=0; j < i; j++)
80              std::cout << "\t" << dp[j];
81          std::cout  << "\t" << dp[i] << "\n";
82      }
83
84      int ans = dp[0];
85      for(int i=1; i < size; i++)
86          ans = Max(ans, dp[i]);
87      delete dp;
88      return ans;
89  }
```

嗯我觉得就比较好理解的一个 dp（也是入门难度）。

随便运行一下啊结果：

```
PS D:\_Codes\VSCode\_Works\Algorithm\Lab5\codes> cd "d:\_Codes\VSCode\_Works\Algorithm\Lab5\cod
Please input the name of dataFile:
data_10_100
Get data in 9.18007s.
        93
        101     102
        177     121     113
        271     186     155     196
        330     362     265     273     206
        365     393     362     369     344     227
        458     401     409     458     425     440     323
        514     500     505     516     465     514     477     394
        530     558     529     555     606     589     590     502     486
        612     655     657     648     639     699     600     653     531     582
Answer is 699.
Size: 10, calculate answer in 0.0107034s.
```

这是 dp 后的每一行的数据，都打印出来了。

2.7 子函数-蒙图版动态规划（探测）

这其实就是探测的一个子问题：我们如何用子问题题寻找到一个比较好的挖矿策略。这种方法是无法保证获得最好策略的。这其实很像数学建模题目，加一个探测成本就完美了。

所以这个思路也和明确，就是根据层数算 dp。但是这种子问题的分割和处理还是比较麻烦的，尤其是在衔接和边界问题上。

- 代码编写如下（实质上还是自顶向下的 dp）

```
int Mining(int** const data, int size, int x) {
    // solve x
    if(x > size)
        x = size;

    // allocate dp space depend on the last line, and initialize
    int** dp = new int*[size];
    for(int i=0; i<size; i++) {
        dp[i] = new int[i+1];
        for(int j=0; j <= i; j++)
            dp[i][j] = 0;
    }
    dp[0][0] = data[0][0];  // has solved the first layer

    // start dp
    int i=0, j=0;
    while(i+1 < size) {
        // generate tmp detection layers = x
        if(i+x >= size-1)
            x = size-1-i;
        std::cout << "\nx=" << x << "\n";

        // generate
        for(int s=1; s <= x; s++) {
            for(int t=0; t <= s; t++) {
                // the left
                if(j+t == 0)
                    dp[i+s][0] = dp[i+s-1][0]+data[i+s][0];
                // the right
                else if(j+t == i+s)
                    dp[i+s][i+s] = dp[i+s-1][i+s-1]+data[i+s][i+s];
                // the reamin
                else {
                    dp[i+s][j+t] += Max(dp[i+s-1][j+t-1], dp[i+s-1][j+t]);
                }
            }
        }

        // generate i and j (select the biggest one in last layer)
        i += x;
        int target_j = j;
        for(int t=1; t <= x; t++)
            if(dp[i][j+t] > dp[i][target_j])
                target_j = j;
        j = target_j;

        // print i, j and dp
        std::cout << "current position: i=" << i << ", j=" << j << "\n";
        for(int s=0; s < size; s++) {
            for(int t=0; t < s; t++)
                std::cout << "\t" << dp[s][t];
            std::cout << "\t" << dp[s][s] << "\n";
        }
    }

    // last layer for answer
    int ans = dp[i][j];
    delete dp;
    return ans;
}
```

- 代码逻辑

1. 开辟 dp 空间。进行一个代码的编写。我们在这里申请一个数据大小的 dp 空间，方便输出我们进行探测的方位。

2. 循环设置探测（默认第一层已经探测，不然处理起来很有麻烦）。根据我们的探测距离来进行一个规整化：假如能谈测完，那就设置该次探测到最后一行。

3. 在循环中进行状态转移。

4. 在最后探测的行进行选择，选择最大的一个 block 往下挖。

- 测试数据输出

```
PS D:\_Codes\VSCode\_Works\Algorithm\Lab5\codes> cd "d:\_Codes\VSCode\_Works\Algorithm\Lab5\codes\" ;
Please input the name of dataFile:
data_10_100
Get data in 3.38062s.
Please input the value of x:
4

x=4
current position: i=4, j=0
        93
        101     102
        177     102     113
        271     177     113     196
        330     271     177     196     206
        0       0       0       0       0       0
        0       0       0       0       0       0       0
        0       0       0       0       0       0       0       0
        0       0       0       0       0       0       0       0       0
        0       0       0       0       0       0       0       0       0       0

x=4
current position: i=8, j=0
        93
        101     102
        177     102     113
        271     177     113     196
        330     271     177     196     206
        365     330     0       0       0       0
        458     365     330     0       0       0       0
        514     458     365     330     0       0       0       0
        530     514     458     365     330     0       0       0       0
        0       0       0       0       0       0       0       0       0       0

x=1
current position: i=9, j=0
        93
        101     102
        177     102     113
        271     177     113     196
        330     271     177     196     206
        365     330     0       0       0       0
        458     365     330     0       0       0       0
        514     458     365     330     0       0       0       0
        530     514     458     365     330     0       0       0       0
        612     530     0       0       0       0       0       0       0       0
Answer is 612.
Size: 10, calculate answer in 0.0273456s.
```

数据一共 10 行，所以是探测行数为：4+4+1。边界处理正确。然后推的结果也对。

我们这个结果并不一定是最佳路径。我们只能说我们找到了认为的较好路径。

2.8 子函数-蒙图版动态规划（缺省）

感觉这个就完全是数据理解的问题了。在行中数据缺省的情况下，我们用平均数代替缺省数字。（但假如一行都是缺省，那么直接设置为 0，没有价值）

- 代码逻辑

先处理数据，填补所有的缺省数据。

然后直接调用相同的全局 dp 进行计算。

- 代码编写（处理缺省空间）

```
    // solve minus blocks
    for(int i=0; i < size; i++) {
        int count = 0;  // legal block count
        int sum = 0;
        for(int j=0; j <= i; j++) {
            if(data[i][j] >= 0) {
                count++;
                sum += data[i][j];
            }
        }
        // no legal block
        int avg = 0;
        if(count > 0)
            avg = sum/count;
        // insert avg
        for(int j=0; j <= i; j++)
            if(data[i][j] < 0)
                data[i][j] = avg;
    }
```

- 运行结果

_Works > Algorithm > Lab5 > data > lack_data_10_100.txt

```
10
-8
2 71
87 30 70
70 5 62 55
29 62 2 10 32
77 66 13 33 68 35
20 6 -9 64 87 25 49
6 -3 78 1 2 23 88 71
11 28 69 7 20 80 35 18 27
34 29 64 18 -2 83 9 54 78 1

```

```
Please input the name of dataFile:
lack_data_10_100
Get data in 7.55345s.
        0
        2       71
        89      101     141
        159     106     203     196
        188     221     205     213     228
        265     287     234     246     296     263
        285     293     328     310     383     321     312
        291     331     406     329     385     406     409     383
        302     359     475     413     405     486     444     427     410
        336     388     539     493     454     569     495     498     505     411
Answer is 569.
Size: 10, calculate answer in 0.0091386s.
```

2.9 白盒测试

主要逻辑分支就是三个:

1. dp 的逻辑
2. 探测的边界处理和衔接处理
3. 缺省的数据处理

- dp 的逻辑

这个是任何规范的金字塔都可以测试。在之前的测试数据中已经探测过，不需要再探测。

- 探测的边界处理和衔接处理

边界处理：和 dp 同出一辙

衔接处理：

x：小于 size-1：处理过了

x：大于 size：

```
PS D:\_Codes\VSCode\_Works\Algorithm\Lab5\codes> cd "d:\_Codes\VSCode\_Works\Algorithm\Lab5\codes
Please input the name of dataFile:
data_10_100
Get data in 4.5313s.
Please input the value of x:
10

x=9
current position: i=9, j=5
        93
        101     102
        177     121     113
        271     186     155     196
        330     362     265     273     206
        365     393     362     369     344     227
        458     401     409     458     425     440     323
        514     500     505     516     465     514     477     394
        530     558     529     555     606     589     590     502     486
        612     655     657     648     639     699     600     653     531     582
Answer is 699.
Size: 10, calculate answer in 0.0117375s.
```

可以和上述的直接同时比较，和全局的 dp 相同。逻辑无问题。

2.10 黑盒测试

测试数据分为很多类型，但无非就是数据量的差距；

但蒙图版缺省需要处理“无法探测 block”出现的几率。

（数据量大，屏蔽矩阵输出）

100->100000

```
PS D:\_Codes\VSCode> cd "d:\_Codes\VSCode\_Works\Alg
Please input the name of dataFile:
 data_100_10000
Get data in 13.8734s.
Answer is 720959.
Size: 100, calculate answer in 0.0005322s.
PS D:\_Codes\VSCode\_Works\Algorithm\Lab5\codes> cd
Please input the name of dataFile:
data_1000_10000
Get data in 5.98649s.
Answer is 7290895.
Size: 1000, calculate answer in 0.0047898s.
PS D:\_Codes\VSCode\_Works\Algorithm\Lab5\codes> cd
Please input the name of dataFile:
data_10000_10000
Get data in 12.9719s.
Answer is 72621091.
Size: 10000, calculate answer in 0.345228s.
PS D:\_Codes\VSCode\_Works\Algorithm\Lab5\codes> cd
Please input the name of dataFile:
data_100000_10000
Get data in 99.5725s.
Answer is 2147483647.
Size: 100000, calculate answer in 28.4148s.
PS D:\_Codes\VSCode\_Works\Algorithm\Lab5\codes>
```

```
Please input the name of dataFile:
data_10_100
Get data in 9.44506s.
Answer is 699.
Size: 10, calculate answer in 0.0003158s.
PS D:\_Codes\VSCode\_Works\Algorithm\Lab5\co
Please input the name of dataFile:
data_100_10000
Get data in 14.8459s.
Answer is 720959.
Size: 100, calculate answer in 0.0004868s.
Please input the name of dataFile:
data_1000_10000
Get data in 4.96168s.
Answer is 7290895.
Size: 1000, calculate answer in 0.0043488s.
PS D:\_Codes\VSCode\_Works\Algorithm\Lab5\co
Please input the name of dataFile:
data_10000_10000
Get data in 11.6101s.
Answer is 72621091.
Size: 10000, calculate answer in 0.345055s.
PS D:\_Codes\VSCode\_Works\Algorithm\Lab5\co
Please input the name of dataFile:
data_100000_10000
Get data in 82.9078s.
Answer is 2147483618.
Size: 100000, calculate answer in 30.5389s.
```

至于蒙图版的测试，其实和这个大差不差，算法都是复制过去的代码，其实没必要大量数据测试。至于 1.8G 的 100000 容量数据测试，第一次都达到溢出边界了，应该是溢出问题。

以上两个已经保证了程序正确性。

三、出现问题及解决

3.1 dp 的从右往左覆盖问题

一开始写的从左往右覆盖，直接右边最大，肯定出问题了。才知道这样覆盖会使得右边加了左边的量，所以会显得非常大。

3.2 dp 探测的衔接问题

主要还是我们的 i，j，以及探测层数 x 的处理。经过修改到没有太大问题。主要是处理一下循环的问题。当循环剩余层数小于探测层数时，探测层数改为剩余层数。

3.3 dp 缺省的数据处理方案

我们这里采用数学期望进行层数的缺省填充。这样的话是最符合直观的数学期望的。如果说有什么高级公式可以更贴近的填充这些数据，那么他的结果将会更趋近于现实。但我们没有必要说采用这种，因为这是算法题不是数学建模题，侧重点不一样。

四、总结

4.1 时间复杂度

显然是 O（n^2）

4.2 空间复杂度

经典 dp 是 O(n^2)的存储空间和 O（n）的辅助空间。

贪心是 O(n^2)的存储空间和 O（n）的辅助空间函数栈空间。

但是蒙图版为了显示整个图的 dp 策略路径，我们采用了 O(n^2)的存储空间来存储 dp 过程，使得路径非常直观。

4.3 算法效率

这很简单，因为算法效率他就是固定的。他没有最好最坏，输入什么就是什么。

所以不存在讨论因为数据集不同导致的算法效率问题。

O（n^2）

4.4 理解与思考

dp 思路在刷 leetcode 的时候就已经接触了很多。但是动态规划绝对不止于此。01 背包，n 皇后啥的，都是很经典的 dp。我们在这个路上还有很多经验需要积累。而这种子问题到问题，避免重复计算的思想，也会影响我们的方方面面，至少在我们编写程序的时候科技大大简化我们的程序。