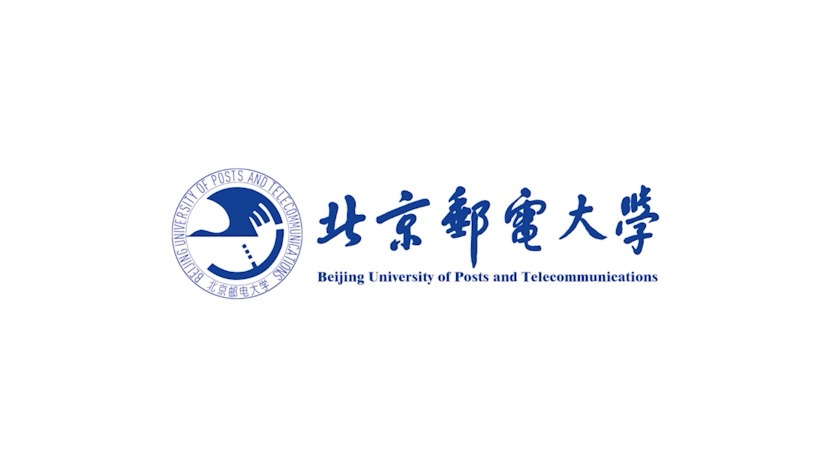
**算法设计与分析实验报告**



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 3](#_Toc119359926)

[1.1 设备规格 3](#_Toc119359927)

[1.2 操作系统 3](#_Toc119359928)

[1.3 编程语言&编译器 3](#_Toc119359929)

[1.4 开发工具 4](#_Toc119359930)

[二、实验内容 4](#_Toc119359931)

[2.1 实验目的 4](#_Toc119359932)

[2.2 实验内容及要求 4](#_Toc119359933)

[2.4 输入设计 5](#_Toc119359934)

[2.4 主函数功能设计 5](#_Toc119359935)

[2.5 子函数-贪心算法 5](#_Toc119359936)

[2.6 子函数-动态规划贪心思想 6](#_Toc119359937)

[2.7 子函数-蒙图版动态规划（探测） 7](#_Toc119359938)

[2.8 子函数-蒙图版动态规划（缺省） 8](#_Toc119359939)

[2.9 白盒测试 9](#_Toc119359940)

[2.10 黑盒测试 10](#_Toc119359941)

[三、出现问题及解决 11](#_Toc119359942)

[3.1 dp的从右往左覆盖问题 11](#_Toc119359943)

[3.2 dp探测的衔接问题 11](#_Toc119359944)

[3.3 dp缺省的数据处理方案 11](#_Toc119359945)

[四、总结 11](#_Toc119359946)

[4.1 时间复杂度 11](#_Toc119359947)

[4.2 空间复杂度 11](#_Toc119359948)

[4.3 算法效率 11](#_Toc119359949)

[4.4 理解与思考 12](#_Toc119359950)

# 一、实验环境

## 1.1 设备规格

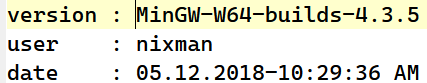


## 1.2 操作系统

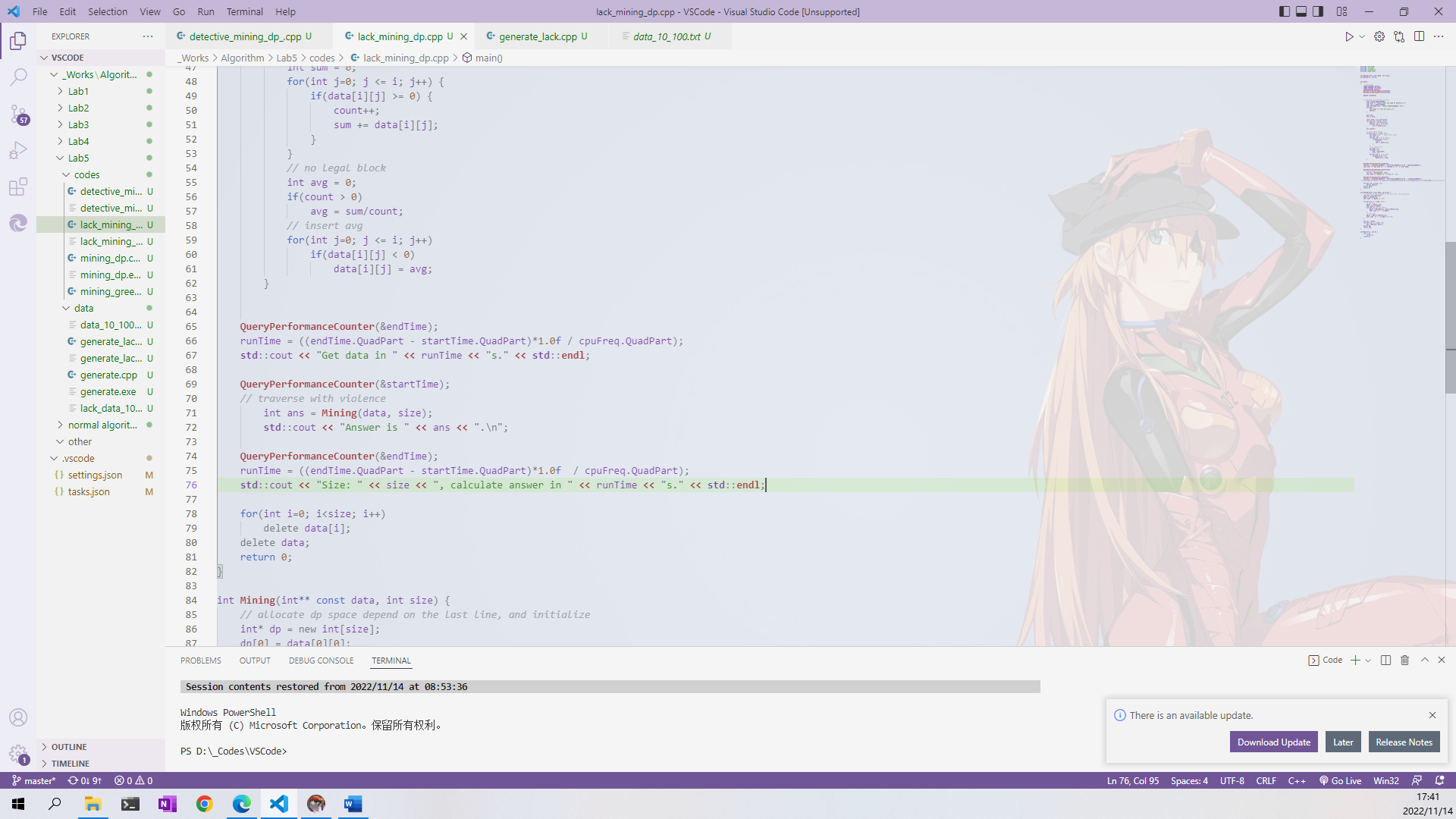


## 1.3 编程语言&编译器

C++11



## 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。

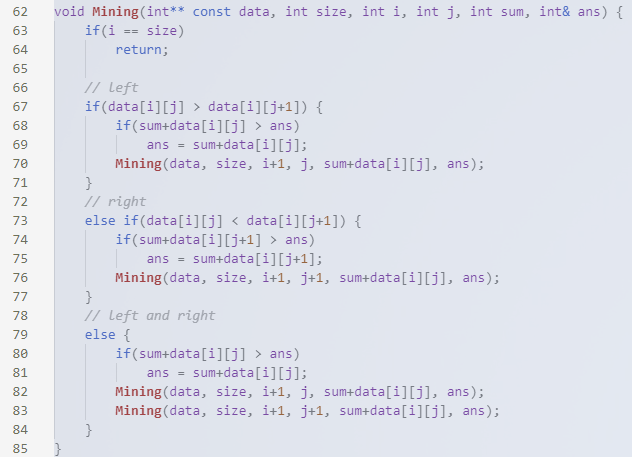


主要结构如上图。就不过多解释了。都是套用之前实验的模板。

## 2.5 子函数-贪心算法

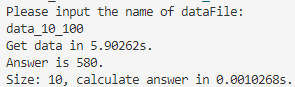


这是最简单的一个从上到下贪心算法。就是从开头位置一直往下找最好的。这种方法无法保证选出最大的。



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：



## 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。

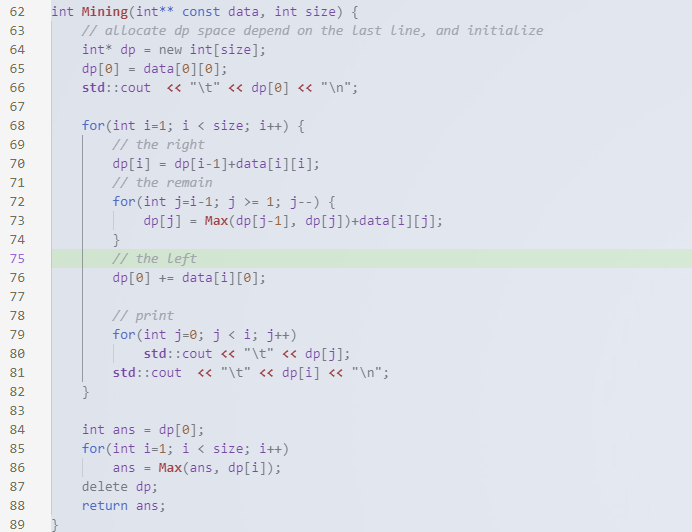
* 辅助空间的选择

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

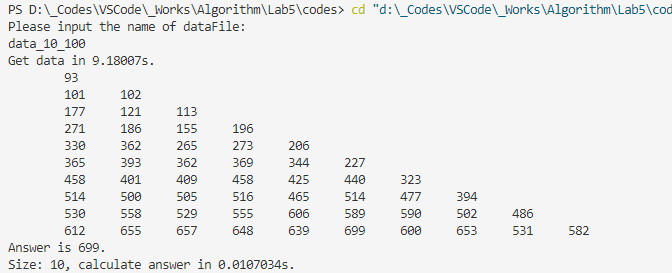
我一开始的思路是从下往上。因为之前开始学习dp的时候思路应该是这样的。但现在回想其实就是为了方便开辅助空间。但我这次有了辅助空间的选择，那就没必要自底向上，那就自顶向下，然后逐个覆盖就可以了。不过要注意行覆盖的顺序：必须是从右往左。因为输入代表着你得从最右边开始覆盖，否则覆盖的数据会影响到后面的东西。

* 算法编写（主要部分为状态转移）



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：



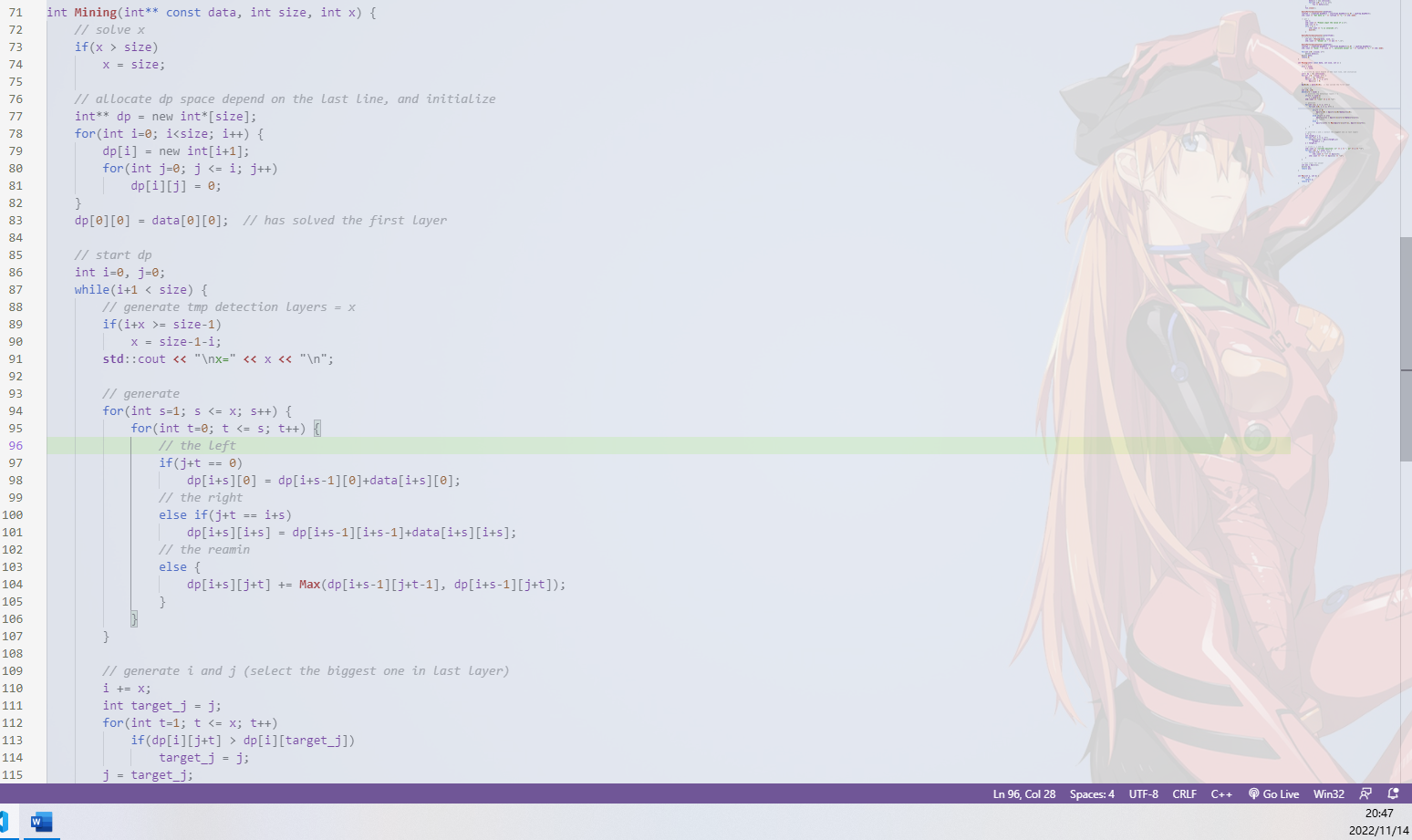
这是dp后的每一行的数据，都打印出来了。

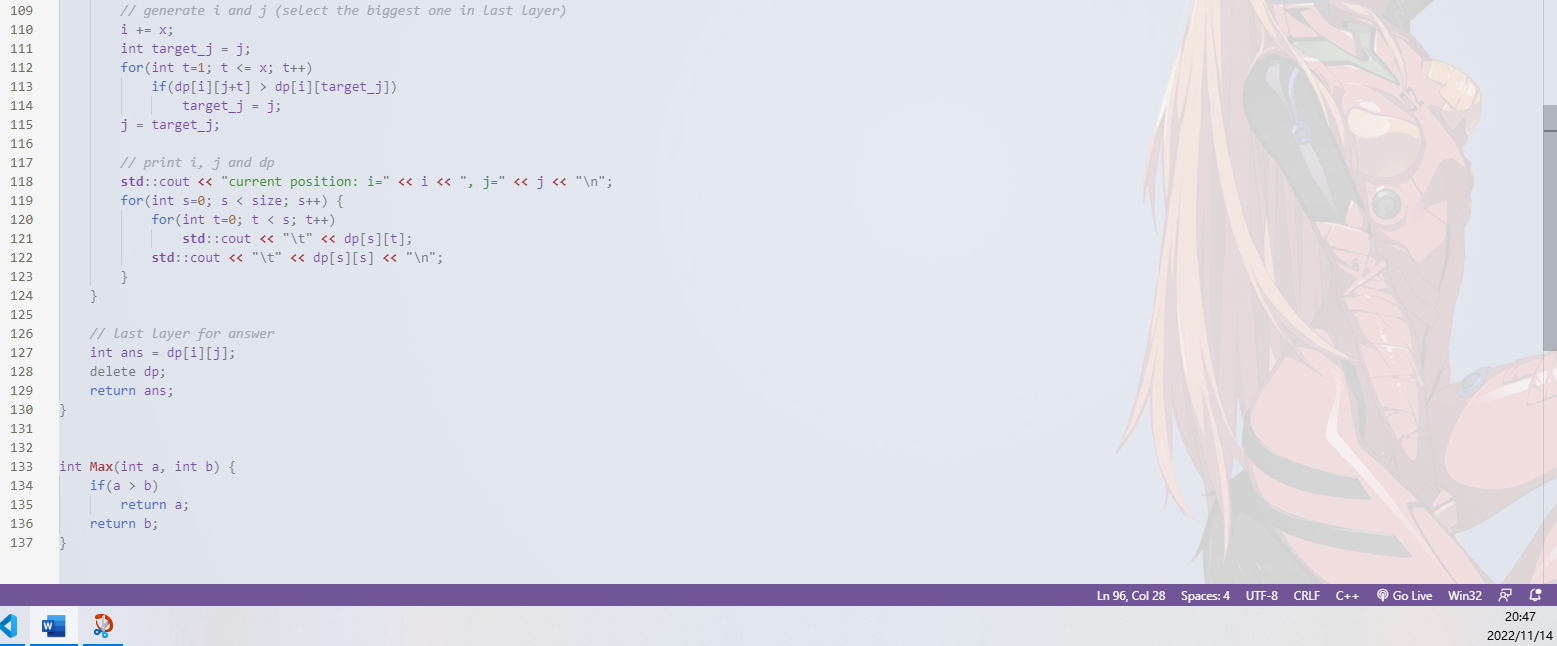
## 2.7 子函数-蒙图版动态规划（探测）

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）





* 代码逻辑

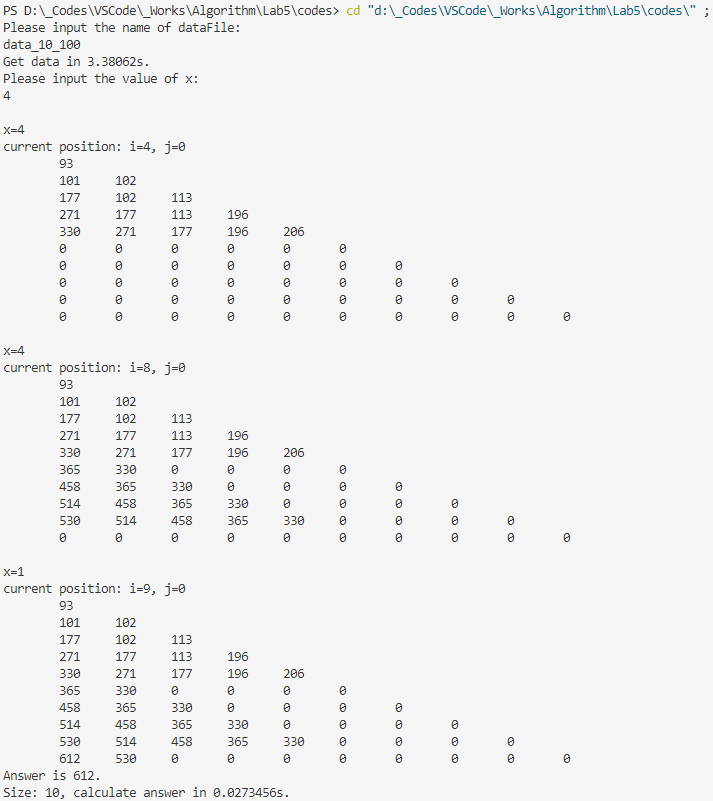
1. 开辟dp空间。进行一个代码的编写。我们在这里申请一个数据大小的dp空间，方便输出我们进行探测的方位。

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



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）

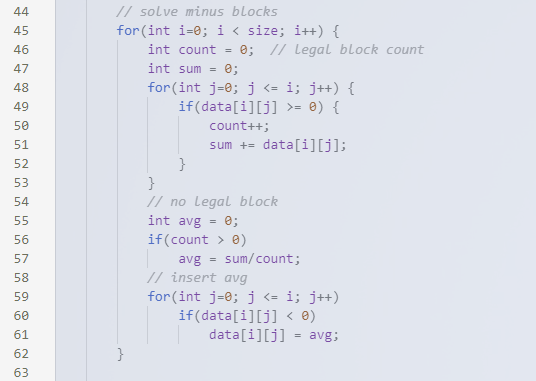
感觉这个就完全是数据理解的问题了。在行中数据缺省的情况下，我们用平均数代替缺省数字。（但假如一行都是缺省，那么直接设置为0，没有价值）

* 代码逻辑

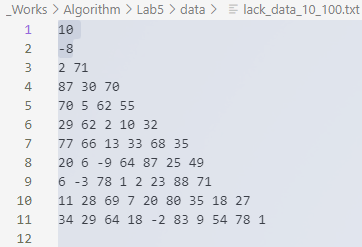
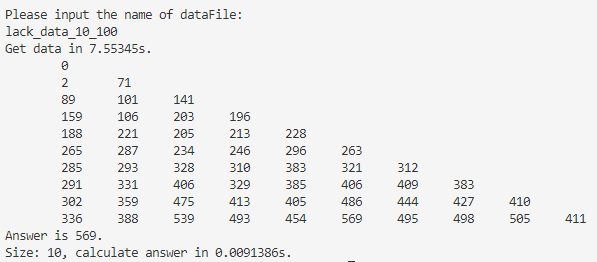
先处理数据，填补所有的缺省数据。

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）



* 运行结果

## 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。

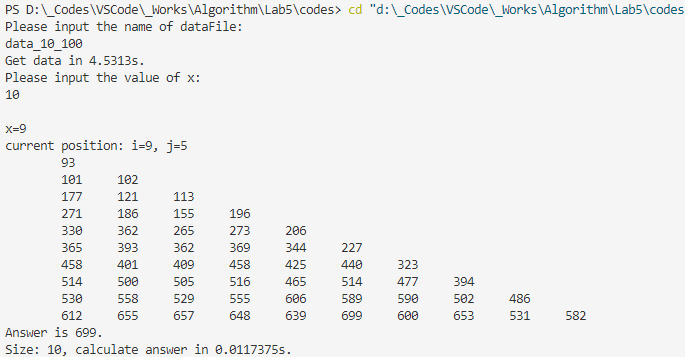
* 探测的边界处理和衔接处理

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：



可以和上述的直接同时比较，和全局的dp相同。逻辑无问题。

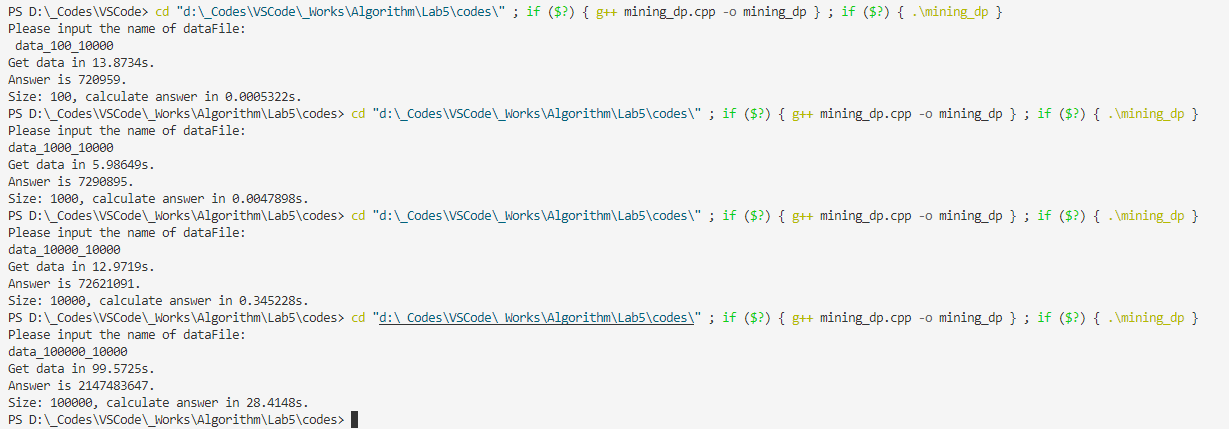
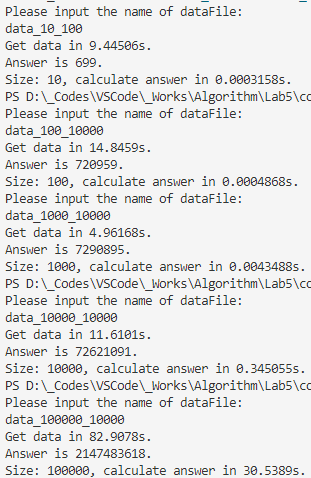
## 2.10 黑盒测试

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**

**** 

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。