本次作业提交截止时间：2023年11月19日23:59

❗注意事项

- 每次作业请上传至对应的文件夹，比如第一次作业上传至文件夹homework1。
  - 该文件夹中包含两个文件夹：code和pdf。
  - 非代码请上传pdf文件至文件夹pdf中。非pdf（比如doc文件、md文件、tex文件）请转换成pdf。
  - C++代码请上传至文件夹code中。本课程只允许C++代码。
  - pdf文件和cpp文件都用自己的学号命名。比如1001.pdf和1001.cpp。
- 涉及公式的作业，推荐使用markdown编辑器。
- 涉及算法伪代码的作业，推荐使用在线$LaTeX$编辑器Overleaf完成，提交编译生成的pdf文档即可。如果需要在overleaf中输入中文并能正确编译，首先在文档中加入package:`\usepackage[UTF8]{ctex}`，然后修改设置：设置->修改Latex引擎->选择“XeLatex”。
- 请基于提供的C++代码框架编写代码，注意：
  - 代码中不要包含任何中文
  - 不要使用<bits/stdc++.h>头文件
  - 不要在main函数后放置任何代码
  - 不要改变给定的函数原型

---

0. 动态规划算法的建议答题格式
    1. 建立递归关系
        1. 使用自然语言描述要求解的子问题
        2. 如何基于子问题求原问题
        3. 基本情况
        4. 递归情况
    2. 自底向上求解
        1. 备忘录形式
        2. 子问题求解顺序
        3. 时空复杂度分析

1. 最长递增子序列（LIS）（子问题1和3要求提供cpp代码，子问题2在pdf中回答）
    1. 编程实现求解最长递增子序列的三种动态规划算法（一些细节请参考课件）
        1. 算法1：令$L(k)$表示$s[1..n]$中以$s[k]$结尾的LIS的长度，原问题即求解$\max_{1\le k\le n} L(k)$

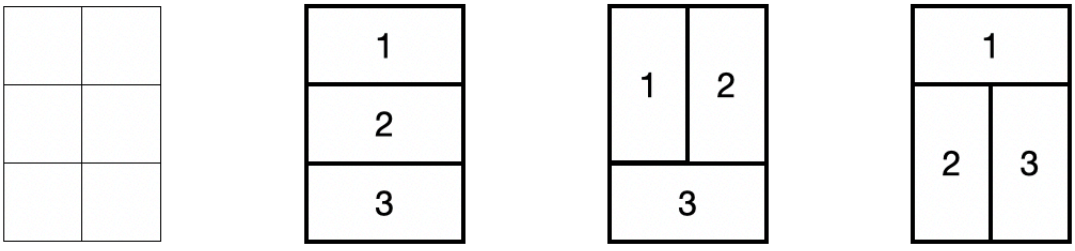
2. 算法2：令$L(i,j)$表示$s[1..n]$中每个元素都大于$s[i]$的LIS的长度，再令$s[0]=-\infty$，原问题即求解$L(0,1)$
3. 算法3：构建数组L，令$L(k)$表示$s[1..n]$中长度为k且末尾元素最小的递增子序列的末尾元素，原问题即求解数组L的长度

2. 上述算法仅仅求出LIS的长度，请对其改进以求出具体的序列。首先给出每种改进算法的基本思路，然后给出伪代码，并分析算法的时空复杂度。
3. 编程实现上述改进的三种算法。

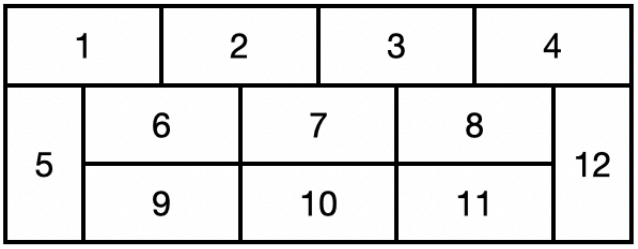
```cpp
// Note:
// You are free to utilize any C++ standard library functions.
// Please ensure to include the necessary headers below.
// Avoid using <bits/stdc++.h> to prevent potential compilation errors that could result in a score of zero.
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

ostream& operator<<(ostream& os, const vector<int>& v) {
    for (auto e : v)
        os << e << ' ';
    return os;
}

ostream& operator<<(ostream& os, const vector<vector<int>>& v) {
    for (auto e : v)
        os << e << '\n';
    return os;
}

// insert code here...

// algorithm 1 to compute the length of LIS
int lis_dp1(const vector<int>& s, int n) {
  // insert code here...
}

// algorithm 2 to compute the length of LIS
int lis_dp2(const vector<int>& s, int n) {
  // insert code here...
}

// algorithm 3 to compute the length of LIS
int lis_dp3(const vector<int>& s, int n) {
  // insert code here...
}

// revised algorithm 1 to compute a LIS
```

```cpp
vector<int> find_lis_dp1(const vector<int>& s, int n) {
  // insert code here...
}

// revised algorithm 2 to compute a LIS
vector<int> find_lis_dp2(const vector<int>& s, int n) {
  // insert code here...
}

// revised algorithm 3 to compute a LIS
vector<int> find_lis_dp3(const vector<int>& s, int n) {
  // insert code here...
}

int main(int argc, const char * argv[]) {
    vector<int> s {-1, 8, 3, 6, 1, 3, 5, 4, 7};
    cout << lis_dp1(s, s.size() - 1) << endl;
    cout << lis_dp2(s, s.size() - 1) << endl;
    cout << lis_dp3(s, s.size() - 1) << endl;

    cout << find_lis_dp1(s, s.size() - 1) << endl;
    cout << find_lis_dp2(s, s.size() - 1) << endl;
    cout << find_lis_dp3(s, s.size() - 1) << endl;
    return 0;
}

// Please refrain from including any code beyond the main function,
// as any additional code will be removed during the code evaluation
process.

```

2. 给定一个$3 \times n$的棋盘， 设计一个动态规划算法计算有多少种使用1×2的骨牌进行完美覆盖的方案。完美覆盖是指没有未覆盖的方格，也没有堆叠或者悬挂在棋盘外的骨牌。当$n = 2$时，如下图所示有3种覆盖方案。



当$n = 8$时，下图展示了其中一种覆盖方案。



3. We are given a checkerboard which has 4 rows and n columns, and has an integer written in each square. We are also given a set of $2n$ pebbles, and we want to place some or all of these on the checkerboard (each pebble can be placed on exactly one square) so as to maximize the sum of the integers in the squares that are covered by pebbles. There is one constraint: for a placement of pebbles to be legal, no two of them can be on horizontally or vertically adjacent squares (diagonal adjacency is fine).

    1. Determine the number of legal *patterns* that can occur in any column (in isolation, ignoring the pebbles in adjacent columns) and describe these patterns.

    Call two patterns *compatible* if they can be placed on adjacent columns to form a legal placement. Let us consider subproblems consisting of the first k columns $1 \leq k \leq n$. Each subproblem can be assigned a *type*, which is the pattern occurring in the last column.

    2. Using the notions of compatibility and type, give an $O(n)$-time dynamic programming algorithm for computing an optimal placement.