天津大学

程序设计综合实践课程报告

动态规划实验

学生姓名 陈昊昆

学院名称 智算学部

专　　业 软件工程

学　　号 **3021001196**

1. 冬冬爬楼梯

1.1 题目分析

由于登上第 n 级楼梯的方案取决于登上第 n-1、n-2、n-3 级的方案

故得到递归式，然后由 dp 实现

需要注意的是，由于数字增长非常快，需要用高精度加法来存取

1.2 题目代码（带注释）

```
#include<bits/stdc++.h>
using namespace std;


// 高精度加法函数
vector<int> add(const vector<int>& a, const vector<int>& b) {
    vector<int> c;
    int t = 0;
    for (int i = 0; i < a.size() || i < b.size(); i++) {
        if (i < a.size()) {
            t += a[i];
        }
        if (i < b.size()) {
            t += b[i];
        }
        c.push_back(t % 10);
        t /= 10;
    }
    if (t) {
        c.push_back(1);
    }
    return c;
}

vector<int> dp[3005];

int main() {
    // 边界条件
```

```
    dp[0] = {1};
    dp[1] = {1};
    dp[2] = {2};
    for (int i = 3; i <= 3000; i++) {
        // 由递推式 使用动态规划算法
        dp[i] = add(dp[i-1], add(dp[i-2], dp[i-3]));
    }
    int n;
    while (cin >> n) {
        for (int i = dp[n].size()-1; i >= 0; i--) {
            cout << dp[n][i];
        }
        cout << endl;
    }
    return 0;
}
```

2. 最大子段和

2.1 题目分析

利用动态规划的思想，计算数组 dp

考虑两种可能

要么把当前的数加入之前的子序列中

要么从当前数开始一个子序列

计算中不断更新最大值

2.2 题目代码（带注释）

```
#include<bits/stdc++.h>
using namespace std;

int main(){
    int n;
    cin >> n;
    for(int i = 0; i < n; i++){
        int m;
        cin >> m;
        int *arr = new int [m];
        int *dp = new int [m];
        for(int j = 0; j < m; j++){
            cin >> arr[j];
        }
        int max_sum = 0;
        dp[0] = arr[0];
        for(int u = 1; u < m; u++){
            //  考虑两种可能 要么把当前的数加入之前的子序列中 要么从当前数开始一个子序列
            dp[u] = max(dp[u-1] + arr[u], arr[u]);
            // 更新最大子序列和
            max_sum = max(max_sum, dp[u]);
        }
        max_sum =  max_sum > 0 ? max_sum : 0;
        cout << max_sum << endl;
```

```
    }
}
```

3. 最大子阵和

3.1 题目分析

利用动态规划算法

将二维数组 A 中每一行的和保存到一维数组 B 中

对于每一列，将一维数组 B 中对应的元素相加，得到一个一维数组 dp

使用一维最大连续子序列和算法求解，得到一个最大值 max

将 max 与 ans 比较，如果 max 大于 ans，则将 ans 更新为 max

3.2 题目代码（带注释）

```
#include<bits/stdc++.h>
using namespace std;
const int maxn = 105;
int A[maxn][maxn], B[maxn], dp[maxn], N; // 二维数组 A，一维数组 B、dp，变量 N

int main(){
    while(cin >> N){
        memset(A, 0, sizeof A); // 将二维数组 A 清零
        for(int i = 0; i < N; i++)
            for(int j = 0; j < N; j++)
                cin >> A[i][j];
        int ans = -32767; // ans 为答案，初值为一个极小的数
        int i = 0;
        while(i < N){
            for(int i = 0; i < sizeof(B) / sizeof(B[0]); i++) {
                B[i] = 0;
            }
        memset(dp, 0, sizeof dp);
        for(int j = i; j < N; j++){
            for(int k = 0; k < N; k++){ // 循环将二维数组 A 的每一列加到一维数组 B 中
                B[k] += A[j][k];
            }
            dp[0] = B[0];
```

```
            int i = 1;
            while(i < N){
                dp[i] = max(B[i], dp[i-1] + B[i]); // 状态转移方程
                i++; // i 加 1
            }
            int k = 0;
            for(int i = 1; i < N; i++){ // 循环找到 dp 数组中最大值的下标
                if(dp[i] > dp[k]) k = i;
            }
            if(ans < dp[k]) ans = dp[k]; // 如果当前答案小于 dp 数组中最大值，就将答案更新为最大值
        }
        i++;
        }
        cout << ans << endl;
    }
return 0;
}
```

4. 最长上升子序列

4.1 题目分析

用数组 dp 存储当前位置为止的最长上升子序列长度，初始化为 1

从位置 2 开始遍历原序列，对于位置 i，从位置 1 到 i-1 遍历原序列

如果当前位置 j 的值小于位置 i 的值，并且 dp[j]+1 大于 dp[i] 的值

则更新 dp[i] 的值为 dp[j]+1

最后遍历 dp 数组，取出最大的值

4.2 题目代码（带注释）

```
#include<bits/stdc++.h>
using namespace std;

const int MAXN = 1005;

int n, a[MAXN], dp[MAXN];

int main() {
    cin >> n;
    for (int i = 0; i < n; i++) {
        cin >> a[i];
        dp[i] = 1;  //初始化为 1
    }
    for (int i = 1; i < n; i++) {
        for (int j = 0; j < i; j++) {
            // 遍历由 0 到 i-1 加入能满足的最长序列
            if (a[j] < a[i] && dp[j] + 1 > dp[i]) {
                dp[i] = dp[j] + 1;
            }
        }
    }
    // 输出 dp 中最大的数
    cout << *max_element(dp, dp+n) << endl;
    return 0;
}
```

5. 最小乘车费用

5.1 题目分析

利用动态规划解决，用 dp[i]表示从第 1 个车站到第 i 个车站的最小花费

则 dp[i]则可由本站的前十个车站的 dp 加上从前面车站到本车站的费用的最小值表示

需要注意的是，前十个车站是可以不换乘直达的

5.2 题目代码（带注释）

```
#include<bits/stdc++.h>
using namespace std;

int main(){
    int fee[10];
    for(int i = 0; i < 10; i++){
        cin >> fee[i];
    }
    int des;
    cin >> des;
    int *dp = new int [des];
    dp[0] = fee[0];
    for(int i = 1; i < des; i++){
        int minfee = 10000;
        // 前十个车站是可以不换乘直达的
        if(i < 10) minfee = fee[i];
        //  dp[i]则可由本站的前十个车站的 dp 加上从前面车站到本车站的费用的最小值表示
        for(int j = i - 1; j >= i - 10 && j >= 0; j--){
            minfee = min(minfee, dp[j] + fee[i - j -1]);
        }
        dp[i] = minfee;

    }
    cout << dp[des - 1];
}
```

6. 方格取数

6.1 题目分析

利用动态规划解决，用 dp[i][j]表示当经过该位置时的最大值

则 dp[i] [j]则可由其上面的最大值和左边的最大值表示

最后输出右下角的 dp

6.2 题目代码（带注释）

```
#include<bits/stdc++.h>
using namespace std;

int main(){
    int n;
    cin >> n;
    int **a = new int *[n];
    int **dp = new int *[n];
    for(int i = 0; i < n; i++){
        a[i] = new int [n];
        dp[i] = new int [n];
    }
    for(int i = 0; i < n; i++){
        for(int j = 0; j < n; j++){
            cin >> a[i][j];
            dp[i][j] = 0;
        }
    }
    dp[0][0] = a[0][0];
    // 初始化第零行 和 第零列
    for(int i = 1; i < n; i++) dp[0][i] = dp[0][i - 1] + a[0][i];
    for(int i = 1; i < n; i++) dp[i][0] = dp[i - 1][0] + a[i][0];
    // 当经过这个数， 最大的值取决于其上面的最大值和左边的最大值
    for(int i = 1; i < n; i++){
        for(int j = 1; j < n; j++){
            dp[i][j] = max(dp[i-1][j], dp[i][j-1]) + a[i][j];
        }
    }
```

```
    cout << dp[n-1][n-1];
}
```

7. 01 背包

7.1 题目分析

定义两个数组 w 和 p，分别用于存放物品的重量和价值

定义一个二维数组 f，其中 f[i][j] 表示前 i 种物品放进容量为 j 的背包中所能得到的最大价值

初始化 f[n][j]，因为只有一件物品，所以最多只能放进容量为 w[n] 的背包中，所以初始化 j < w[n] 时 f[n][j] 为 0，j ≥ w[n] 时 f[n][j] 为 p[n]

从 i = n - 1 开始，从后往前循环每个物品，假设当前考虑到第 i 个物品：

若 w[i] > j，则第 i 个物品放不进容量为 j 的背包中，所以 f[i][j] = f[i+1][j]

若 w[i] ≤ j，则第 i 个物品可以选择放进背包中或不放，因此分别计算将第 i 个物品放进背包和不放第 i 个物品时的最大价值：

不放第 i 个物品，则 f[i][j] = f[i+1][j]

放第 i 个物品，则 f[i][j] = f[i+1][j-w[i]] + p[i]

循环结束后，f[1][M] 即为所求

7.2 题目代码（带注释）

```
#include<bits/stdc++.h>
using namespace std;

int main(){
    int m, n;
    cin >> m >> n;
    int *w = new int [n+1];
    int *p = new int [n+1];
    for(int i = 1; i <= n; i++){
        cin >> w[i] >> p[i];
    }
```

```
    // 动态分配二维数组，f[i][j] 表示前 i 件物品放入一个容量为 j 的背包可以获得的最大价值
    int **f = new int* [n+1];
    for(int i = 1; i <= n; i++){
        f[i] = new int [m+1];
    }

    // 初始化 f[n][j]，即只剩下最后一件物品，这时的最大价值只有两种情况，不选或选一次
    int ymax = w[n] > m ? m : w[n]; // 防止数组越界
    for(int y = 0; y < ymax; y++){ // 不选最后一件物品，背包容量为 y 时最大价值为 0
        f[n][y] = 0;
    }
    for(int y = w[n]; y <= m; y++){ // 选最后一件物品，背包容量为 y 时最大价值为 p[n]
        f[n][y] = p[n];
    }

    // 从倒数第二个物品开始考虑，一直到第一个物品
    for(int i = n - 1; i > 1; i--){
        if(w[i] > m){ // 当前物品重量大于背包容量时，无法放入背包
            for(int y = 0; y <= m; y++){ // 背包容量为 y 时，最大价值等于放入下一个物品时背包容量为 y 时的最大价值
                f[i][y] = f[i+1][y];
            }
            continue;
        }
        // 当前物品重量小于背包容量时，无法放入，最大价值等于放入下一个物品时背包容量为 y 时的最大价值
        for(int y = 0; y < w[i]; y++){
            f[i][y] = f[i+1][y];
        }
        // 当前物品重量小于等于背包容量时，最大价值等于放入下一个物品时背包容量为 y 时的最大价值和放入当前物品的最大价值的最大值
        for(int y = w[i]; y <= m; y++){
            f[i][y] = max(f[i+1][y], f[i+1][y-w[i]] + p[i]);
        }
    }

    // 最后考虑第一个物品
    f[1][m] = f[2][m];
    if(m >= w[1]){
        f[1][m] = max(f[2][m], f[2][m-w[1]] + p[1]);
```

```
    }

    cout << f[1][m];
}
```

8. 完全背包

8.1 题目分析

通过动态规划的思想，使用一个一维数组 **dynamicvalue** 来记录当前背包容量下的最大价值

每次选择一个物品后，从小到大依次更新每个背包容量下的最大价值

更新的过程中，通过判断当前物品的重量是不是当前背包容量的倍数，来确定该背包容量能否装下当前物品

对于能装下的背包容量，如果是第一次涉及到该格子，或者当前物品的装入使得该格子的价值更大，就更新该格子的价值

同时，通过将当前物品加入上一个物品所能达到的所有背包容量中，更新所有目标背包容量下的最大价值

最后，如果最大背包容量的最大价值不为负数，则输出该价值，否则输出 **NO**

8.2 题目代码（带注释）

```
#include<bits/stdc++.h>
using namespace std;
int N,items,maxc,weight,value;
long long dynamicvalue[50010];



#define nega_infinity -200000//比最大重量还大


int main()
{
    cin>>N;
    for(int i = 0;i<N;i++){
        cin>>items>>maxc;
        memset(dynamicvalue,nega_infinity,sizeof(long long)*50010);
        int choose = 1;
        while(choose <= items){//选择哪个 item
```

```
            choose++;
            cin>>weight>>value;
            if(weight > maxc)continue;
            int curcapacity = 1;
            while(curcapacity <= maxc){//当前背包容量
                curcapacity++;
                long long uppervalue = dynamicvalue[curcapacity];//上一
个物品在当前容量下的价值
                //如果自身的重量是格子倍数
                if(curcapacity % weight == 0){
                    int index = curcapacity / weight;//求倍数
                    long long  sumvalue = index * value;//综合价值
                    if(sumvalue         >         uppervalue         ||
dynamicvalue[curcapacity] <= 0){
                        //自己比以前的格子还大||这个格子从未被涉及
                        dynamicvalue[curcapacity] = sumvalue;
                        uppervalue = dynamicvalue[curcapacity];
                    }
                }
                int targetcapacity = curcapacity + weight;//在上一个物品
的基础上，装上当前物品，将会达到什么体积
                long          long          uppertargetvalue          =
dynamicvalue[targetcapacity];
                long long iftakecuritem = uppervalue + value;
                if(targetcapacity <= maxc && uppervalue > 0){
                    //如果目前的背包还装的下这个体积 && 当前格子是被装过的，
那么这个格子可以当跳板
                    if((iftakecuritem     >     uppertargetvalue     &&
uppertargetvalue > 0)
                    ||uppertargetvalue < 0){
                        dynamicvalue[targetcapacity] = iftakecuritem;
                    }
                }
            }
        }
        //如果最终格不是 0，就意味着可以装满，输出最终格，否则发 no
        if(dynamicvalue[maxc] > 0)cout<<dynamicvalue[maxc]<<endl;
        else cout<<"NO"<<endl;
    }
}
```