算法分析题

3-1

设计算法如下：

```
#include<iostream>
#include<algorithm>
using namespace std;

const int N = 1000;
int dp[N];//以 i 结尾的递增子序列的长度
int Left, Right;//最长序列的首尾

int DP(int a[], int n){
    dp[0] = 1;
    for(int i = 1; i < n; i++){
        dp[i] = 1;
        if(a[i] > a[i-1]){
            dp[i] = max(dp[i], dp[i-1]+1);
        }
    }
    int len = 0, mark = 0;
    for(int i = 0; i < n; i++){
        if(len < dp[i]){
            len = dp[i];
            mark = i;
        }
    }
    Left = mark - len + 1;
    Right = mark;
    return len;
}
```

```
PS C:\Code> .\DP题库\最长单调上升子序列.exe
10
1 2 3 4 5 4 6 7 8 9
5
PS C:\Code> .\DP题库\最长单调上升子序列.exe
15
1 2 3 4 5 1 5 99 2 4 6 8 10 12 14
7
```

时间复杂度分析：两个循环内操作都是线性的，故复杂度 O（kn）= O（n）

3-4

```
#include<iostream>
```

```cpp
#include<algorithm>
#include<cstring>

using namespace std;
const int MAX=2000;
int dp[105][105],v[MAX],m[MAX],w[MAX];

int _2D01package(int N,int V,int M){//个数、体积上限、重量上限
    for(int i=1;i<=N;i++){
        cin>>v[i]>>m[i]>>w[i];
    }
    //以体积为主，重量为次序
    for(int i=1;i<=N;i++){
        for(int j=V;j>=v[i];j--){
            for(int k=M;k>=m[i];k--){
                dp[j][k]=max(dp[j-v[i]][k-m[i]]+w[i],dp[j][k]);
            }
        }
    }
    return dp[V][M];
    //由于从大往小遍历，不会重复使用物品，并且 dp 的值是最终的结果，所以可以直接输出 dp[V][M]
    return 0;
}
```

算法复杂度分析：

时间：

三重循环结构：

外层循环：遍历所有物品，共 N 次（for(int i=1; i<=N; i++)）。

中间循环：遍历体积维度，从 V 到 v[i] 递减，共 V - v[i] + 1 次。

内层循环：遍历重量维度，从 M 到 m[i] 递减，共 M - m[i] + 1 次。

对于每个物品 i，体积和重量的循环次数均与背包容量 V 和 M 成线性关系。因此，每个物品的处理时间复杂度为 O(V × M)。

因此，**总的时间复杂度为 O(N × V × M)**。

空间复杂度分析：

总空间复杂度：

由二维 dp 数组主导，**空间复杂度为 O(V × M)**。

算法实现题

3-3 环形石子合并问题

思路：

动态规划定义：

dp_min[i][j]：合并从第 i 堆到第 j 堆的最小得分。

$dp\_max[i][j]$：合并从第 i 堆到第 j 堆的最大得分。

状态转移方程：

对于区间[i, j]，枚举分割点 k，计算合并两部分的得分：

$dp\_min[i][j] = \min\{ dp\_min[i][k] + dp\_min[k+1][j] + s[j] - s[i-1] \}$，其中 $i \le k < j$

$dp\_max[i][j] = \max\{ dp\_max[i][k] + dp\_max[k+1][j] + s[j] - s[i-1] \}$，其中 $i \le k < j$

s 是前缀和数组，$s[j] - s[i-1]$为区间[i, j]的石子总数。

环形处理：

将数组复制一倍，形成长度为 2n 的线性数组，处理所有长度为 n 的区间[i, i+n-1]，取最小值和最大值。

算法实现：

```cpp
const int MAXN = 205; // 最大处理 2n 堆石子

int a[MAXN * 2];    // 存储石子数的数组（复制后）
int s[MAXN * 2];    // 前缀和数组
int dp_min[MAXN * 2][MAXN * 2]; // 最小得分动态规划表
int dp_max[MAXN * 2][MAXN * 2]; // 最大得分动态规划表
int main() {
    ifstream fin("input.txt");
    ofstream fout("output.txt");
    int n;
    fin >> n;
    for (int i = 1; i <= n; ++i) {
        fin >> a[i];
    }
    // 复制数组形成环形
    for (int i = n + 1; i <= 2 * n; ++i) {
        a[i] = a[i - n];
    }
    // 计算前缀和
    s[0] = 0;
    for (int i = 1; i <= 2 * n; ++i) {
        s[i] = s[i - 1] + a[i];
    }
    // 初始化动态规划表
    for (int i = 1; i <= 2 * n; ++i) {
        dp_min[i][i] = 0;
        dp_max[i][i] = 0;
    }
    // 填充动态规划表
    for (int L = 2; L <= 2 * n; ++L) { // L 是区间长度
        for (int i = 1; i <= 2 * n - L + 1; ++i) {
            int j = i + L - 1;
            dp_min[i][j] = INT_MAX;
            dp_max[i][j] = INT_MIN;
```

```
            for (int k = i; k < j; ++k) {
                // 计算最小得分
                int current_min = dp_min[i][k] + dp_min[k + 1][j] + (s[j]
- s[i - 1]);
                if (current_min < dp_min[i][j]) {
                    dp_min[i][j] = current_min;
                }
                // 计算最大得分
                int current_max = dp_max[i][k] + dp_max[k + 1][j] + (s[j]
- s[i - 1]);
                if (current_max > dp_max[i][j]) {
                    dp_max[i][j] = current_max;
                }
            }
        }
    }
    // 寻找所有长度为 n 的区间的最小和最大值
    int min_total = INT_MAX;
    int max_total = INT_MIN;
    for (int i = 1; i <= n; ++i) {
        int j = i + n - 1;
        if (dp_min[i][j] < min_total) {
            min_total = dp_min[i][j];
        }
        if (dp_max[i][j] > max_total) {
            max_total = dp_max[i][j];
        }
    }
    fout << min_total << endl;
    fout << max_total << endl;
    return 0;
}
```

复杂度分析：

时间复杂度：

预处理：复制数组和计算前缀和均为 O(n)。

动态规划填充：三重循环的复杂度为 O((2n)^3)，即 O(n^3)。

寻找最优解：遍历 n 个区间，复杂度为 O(n)。

故总时间复杂度：O(n^3)。

空间复杂度：

动态规划表：dp_min 和 dp_max 的大小为(2n)^2，即 O(n^2)。

其他数组：前缀和和石子数组的空间为 O(n)。

故总空间复杂度：O(n^2)。

3-13 最大 k 乘积问题

算法设计思路：

动态规划定义：dp[j][i] 表示将前 i 个字符分割成 j 段的最大乘积。

对于每个子问题，尝试所有可能的分割方式，并选择最大的乘积。

算法实现：

```
#include <iostream>
#include <fstream>
#include <vector>
#include <string>
#include <algorithm>

using namespace std;

long long maxProduct(string s, int k) {
    int n = s.size();
    vector<vector<long long>> dp(k + 1, vector<long long>(n + 1, 0));

    for (int i = 1; i <= n; ++i) {
        dp[1][i] = stoll(s.substr(0, i));
    }

    for (int j = 2; j <= k; ++j) {
        for (int i = j; i <= n; ++i) {
            long long maxVal = 0;
            for (int p = j - 1; p < i; ++p) {
                maxVal = max(maxVal, dp[j - 1][p] * stoll(s.substr(p, i -
p)));
            }
            dp[j][i] = maxVal;
        }
    }

    return dp[k][n];
}

int main() {
    ifstream fin("input.txt");
    ofstream fout("output.txt");

    int n, k;
    fin >> n >> k;
    string s;
```

```
    fin >> s;

    long long result = maxProduct(s, k);
    fout << result << endl;

    fin.close();
    fout.close();

    return 0;
}
```

算法分析：

时间:

对于每个子问题，我们需要计算所有可能的分割方式，这需要 O(n^2)的时间。因此，总的时间复杂度是 O(k· n^2)，其中 k 是分割的数量，n 是字符串的长度。

空间:

使用了一个二维数组 dp 来存储中间结果，其大小为 O(k· n)。因此，空间复杂度是 O(k· n)。

3-14 最少费用购物问题

算法思路：

1. 定义动态规划状态

用一个多维数组来表示当前购买商品的状态：

假设有 B 种商品，每种商品的需求量为 K[1], K[2], ..., K[B]。

定义状态 dp[k1][k2]...[kB] 表示已经购买了第 1 种商品 k1 件、第 2 种商品 k2 件、...、第 B 种商品 kB 件时的最小费用。

2. 状态转移方程

对于每个状态 (k1, k2, ..., kB)，可以选择以下两种方式之一来更新：

直接购买单件商品：

如果当前状态缺少某种商品，可以按原价购买一件。

转移方程：dp[k1][k2]...[kB] = min(dp[k1][k2]...[kB], dp[k1'][k2']...[kB'] + P[i])，其中 k1' = k1 - 1 表示少买一件第 1 种商品。

使用优惠组合：

如果当前状态可以使用某个优惠组合，则尝试应用该组合。

转移方程：dp[k1][k2]...[kB] = min(dp[k1][k2]...[kB], dp[k1''][k2'']...[kB''] + P_offer)，其中 k1'', k2'', ... 是应用优惠组合后剩余的商品需求。

3. 初始化

初始状态 dp[0][0]...[0] = 0，表示没有购买任何商品时的费用为 0。

其他状态初始值设为正无穷大（INF），表示未计算的状态。

4. 得到结果

目标状态是 dp[K[1]][K[2]]...[K[B]]，即购买所有商品所需的最小费用。

算法实现：

```
#include <iostream>
#include <fstream>
```

```cpp
#include <vector>
#include <climits>
#include <unordered_map>
using namespace std;

const int MAX_B = 5; // 最大商品种类数
const int MAX_K = 5; // 每种商品的最大数量
int dp[(MAX_K + 1) * (MAX_K + 1) * (MAX_K + 1) * (MAX_K + 1) * (MAX_K + 1)];

// 将多维索引映射为一维索引
int getIndex(const vector<int>& counts, int B) {
    int index = 0;
    for (int i = 0; i < B; ++i) {
        index = index * (MAX_K + 1) + counts[i];
    }
    return index;
}

int main() {
    ifstream fin_input("input.txt");
    ifstream fin_offer("offer.txt");
   ofstream fout("output.txt");

    int B; // 商品种类数
   fin_input >> B;

    // 商品信息
    struct Item {
        int code; // 编码
        int quantity; // 需求数量
        int price; // 单价
    };
    vector<Item> items(B);
    unordered_map<int, int> code_to_index; // 商品编码到索引的映射
    for (int i = 0; i < B; ++i) {
        fin_input >> items[i].code >> items[i].quantity >> items[i].price;
        code_to_index[items[i].code] = i;
   }

    // 初始化 DP 数组
    fill(dp, dp + (MAX_K + 1) * (MAX_K + 1) * (MAX_K + 1) * (MAX_K + 1) *
(MAX_K + 1), INT_MAX);
   dp[0] = 0;
```

```cpp
    // 读取优惠组合
    int S; // 优惠组合数
    fin_offer >> S;
    vector<vector<pair<int, int>>> offers(S); // 每个优惠组合的商品列表
    vector<int> offer_prices(S); // 每个优惠组合的价格
    for (int i = 0; i < S; ++i) {
        int num_items; // 当前优惠组合中的商品种类数
        fin_offer >> num_items;
        offers[i].resize(num_items);
        for (int j = 0; j < num_items; ++j) {
            fin_offer >> offers[i][j].first >> offers[i][j].second;
        }
        fin_offer >> offer_prices[i];
   }

    // 动态规划求解
    vector<int> current_counts(B, 0); // 当前购买的商品数量
    while (true) {
        // 更新状态
        int currentIndex = getIndex(current_counts, B);
        if (dp[currentIndex] == INT_MAX) break; // 所有状态已处理完毕

        // 尝试直接购买单件商品
        for (int i = 0; i < B; ++i) {
            if (current_counts[i] < items[i].quantity) {
                vector<int> new_counts = current_counts;
                new_counts[i]++;
                int newIndex = getIndex(new_counts, B);
                dp[newIndex] = min(dp[newIndex], dp[currentIndex] +
items[i].price);
            }
        }

        // 尝试使用优惠组合
        for (int i = 0; i < S; ++i) {
            bool can_apply = true;
            vector<int> new_counts = current_counts;
            for (auto& item : offers[i]) {
                int idx = code_to_index[item.first];
                if (new_counts[idx] + item.second > items[idx].quantity) {
                    can_apply = false;
                    break;
                }
                new_counts[idx] += item.second;
```

```
            }
            if (can_apply) {
                int newIndex = getIndex(new_counts, B);
                dp[newIndex] = min(dp[newIndex], dp[currentIndex] +
offer_prices[i]);
            }
        }

        // 更新当前状态
        bool updated = false;
        for (int i = 0; i < B; ++i) {
            if (current_counts[i] < items[i].quantity) {
                current_counts[i]++;
                updated = true;
                break;
            } else {
                current_counts[i] = 0;
            }
        }
        if (!updated) break;
   }

    // 输出结果
    vector<int> final_counts(B, 0);
    for (int i = 0; i < B; ++i) final_counts[i] = items[i].quantity;
    int finalIndex = getIndex(final_counts, B);
   fout << dp[finalIndex] << endl;

    fin_input.close();
    fin_offer.close();
    fout.close();

    return 0;
}
```

复杂度分析：

时间复杂度:

状态数：每种商品最多购买 5 件，因此总状态数为 (MAXK+1^)B，即 6^5 = 7776。

状态转移：对于每个状态，尝试直接购买单件商品需要 O(B)，尝试使用优惠组合需要 O(S*i)，其中 i 是优惠组合中商品的种类数。

总时间复杂度约为 O(7776*(B+S*i))O(7776*(B+S*i))。

空间复杂度

DP 数组占用的空间为 O((MAXK+1)B)=O(7776)。

3-17 字符串合并问题

算法思路：

1.DP 状态定义：dp[i][j] 表示处理字符串 A 的前 i 个字符和字符串 B 的前 j 个字符时的最小扩展距离。

2.状态转移：对于每个状态 dp[i][j]，考虑三种可能的操作：

（1）对齐当前字符：将 A 的第 i 个字符和 B 的第 j 个字符对齐，计算它们的距离并加上 dp[i-1][j-1]。

（2）在 A 中插入空格：A 的第 i 个字符未被使用，B 的第 j 个字符被使用，距离为 k，加上 dp[i][j-1]。

（3）在 B 中插入空格：B 的第 j 个字符未被使用，A 的第 i 个字符被使用，距离为 k，加上 dp[i-1][j]。

算法实现：

```
#include <iostream>
#include <fstream>
#include <vector>
#include <algorithm>
using namespace std;

int main() {
    ifstream fin("input.txt");
   ofstream fout("output.txt");

    string A, B;
   int k;

    // 读取输入
    getline(fin, A);
    getline(fin, B);
   fin >> k;

    int m = A.length();
   int n = B.length();

    // 初始化 DP 表
   vector<vector<int>> dp(m + 1, vector<int>(n + 1, 0));

    // 初始化边界条件
    for (int i = 0; i <= m; i++) {
        dp[i][0] = i * k;
    }
    for (int j = 0; j <= n; j++) {
        dp[0][j] = j * k;
   }
```

```
    // 填充 DP 表
    for (int i = 1; i <= m; i++) {
        for (int j = 1; j <= n; j++) {
            // 计算当前字符的距离
            int cost;
            if (A[i-1] == ' ' && B[j-1] == ' ') {
                cost = 0;
            } else if (A[i-1] == ' ' || B[j-1] == ' ') {
                cost = k;
            } else {
                cost = abs(A[i-1] - B[j-1]);
            }

            // 状态转移方程
            dp[i][j] = min({
                dp[i-1][j-1] + cost,
                dp[i-1][j] + k,
                dp[i][j-1] + k
            });
        }
   }

    // 输出结果
   fout << dp[m][n] << endl;

    fin.close();
    fout.close();
    return 0;
}
```

复杂度分析：

时间复杂度：

动态规划表的大小为 O(m * n)，其中 m 和 n 分别是字符串 A 和 B 的长度。填充表的每个元素需要常数时间，因此**总时间复杂度为 O(m * n)。**

空间复杂度：

动态规划表的大小为 O(m * n)，**因此空间复杂度为 O(m * n)**。