HOMEWORK 4

1. 矩阵链乘

a. 问题描述

给一个矩阵乘法式 $A_1 * A_2 * \ldots * A_n$，根据乘法顺序不同所需的总乘法次数也不同，找出使总乘法次数最少的乘法顺序。

b. 状态转移方程

动态规划经典问题：假设从i的开始长度为l的矩阵链至少要乘$opt_{i,l}$次，则长度为 $l+1$ 的矩阵链需要的最少乘法数满足：

$$opt_{i,l} = \min_{j=1\to l}(opt_{i,j} + opt_{j+1,l-j} + p_i * p_{j+i} * p_{j+l})$$

c. 测试

测试文件使用 Github 李真同学开源的例子：

```
文件 p1-in.txt  输出
2
10880
1150
文件 p1-in-big.txt  输出
371228993909382
2909365722432806
```

d. 复杂度分析

子问题有$O(n^2)$个，每个子问题需要$O(n)$时间复杂度，总时间复杂度为$O(n^3)$

f. 代码

```
#include <iostream>
#include <cstring>
#include <cstddef>
#include <algorithm>
int main(){
    int n, r, c;
    int chain[401];
    uint64_t dp[401][401];
    while(std::cin >> n){
        memset(dp, 0xff, 401 * 401 * sizeof(uint64_t));
        for(int i = 0; i < n; ++i){
```

```
            std::cin >> r >> c;
            chain[i] = r;
        }
        chain[n] = c;

        dp[0][0] = 0;
        for(int i = 0; i < n; ++i){
            dp[i][0] = 0;
            dp[i][1] = 0;
        }
        for(int L = 2; L <= n; ++L){
            for(int i = 0; i <= n - L; ++i){
                for(int j = 1; j < L; ++j){
                    dp[i][L] = std::min(dp[i][L],
                        dp[i][j] + dp[i + j][L - j] + (uint64_t)chain[i] * chain[i + j] * chain[i + L]);
                }
            }
        }
        std::cout << dp[0][n] << std::endl;
    }
    return 0;
}
```

Sequence Alignment DP

a. 问题描述

给两个字符串 s1,s2,求将 s1 转换成 s2 所需要最小的 cost，cost 根据转换和跳过两个操作的不同取值不同，将 i 字符转化为 j 字符所需要的 cost 是 alpha[i][j]，而跳过字符所需要的 cost 是 sigema[i]。

b. 状态转移方程

子问题划分为 s1 的前 i 个字符转化为 s2 的前 j 个字符所需要的最小 cost。

$$\mathrm{opt}[\mathrm{i}][\mathrm{j}] = \min\begin{cases} opt[i-1][j-1] + alpha\big[s1[i]\big][s2[j]] \\ opt[i-1][j] + sigema[s1[i]] \\ opt[i][j-1] + sigema[s2[j]] \end{cases}$$

c. 测试

测试文件使用 Github 李真同学开源的例子：

```
文件 p2-in.txt 输出
2
4
2
文件 p2-in-big.txt 输出
948
5357887
6977309
4223775
5736652
2664187
文件 p2-in-huge.txt 输出
9470
50479302
72362621
41965222
49000710
27636412
```

d. 复杂度分析

子问题有$O(nm)$个，nm 为字符串长度每个子问题需要$O(1)$时间复杂度，总时间复杂度为$O(nm)$代码

e. 代码

```
#include <iostream>
#include <cstring>
#include <cstddef>
#include <algorithm>
```

```
#include <vector>
using namespace std;
int main(){
    uint32_t m, n1, n2;
    uint32_t sigema[65];
    uint32_t alpha[65][65];
    uint32_t s1[10001], s2[10001];

    while(std::cin >> m){

        for(uint32_t i = 0; i < m; ++i){
            std::cin >> sigema[i];
        }

        for(uint32_t i = 0; i < m; ++i){
            for(uint32_t j = 0; j < m; ++j){
                std::cin >> alpha[i][j];
            }
        }

        std::cin >> n1;
        for(uint32_t i = 1; i <= n1; ++i){
            std::cin >> s1[i];
        }

        std::cin >> n2;
        for(uint32_t i = 1; i <= n2; ++i){
            std::cin >> s2[i];
        }
        vector<vector<uint32_t>> dp(max(n1,n2) + 1, vector<uint32_t>(max(n1,n2) + 1, 0));

        dp[0][0] = 0;
        for(uint32_t i = 1; i <= n1; ++i){
            dp[i][0] = dp[i - 1][0] + sigema[s1[i]];
        }
        for(uint32_t i = 1; i <= n2; ++i){
            dp[0][i] = dp[0][i - 1] + sigema[s2[i]];
        }

        for(uint32_t i = 1; i <= n1; ++i){
            for(uint32_t j = 1; j <= n2; ++j){
                dp[i][j] = std::min(dp[i - 1][j - 1] + alpha[s1[i]][s2[j]],
                    std::min(dp[i - 1][j] + sigema[s1[i]], dp[i][j - 1] + sigema[s2[j]]));
            }
```

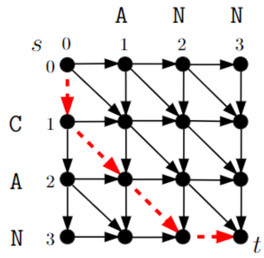
```
    }

    std::cout << dp[n1][n2] << std::endl;
  }
  return 0;
}
```

Sequence Alignment SP

a. 问题描述

在 DP 算法中，我们构造了$O(n^2)$个子问题，每次对下一个最优解的选择都是从 3 个可选项中选择出最优的，如果将所有子问题看作节点，有尝试选择就看作具有一条边，每条边的权值为作这次选择增加的 cost，则 DP 算法从源到目标选择了一条最短路。

如图所示：



红色的边就是 DP 算法选择的边。

我们可以按照这种思路将这个问题转化成求最短路的问题

b. 测试

测试文件使用 Github 李真同学开源的例子：

文件 p2-in.txt 输出
2
4
2
文件 p2-in-big.txt 输出
948
5357887
6977309
4223775
5736652
2664187

c. 复杂度分析

图中的点集数量为 n * m(两字符串长度相乘)，则按照 dijkstra 复杂度

$$\Theta((|E| + |V|)\log|V|)$$

复杂度为$O(nmlog(mn))$

d. 代码

```
#include <iostream>
#include <cstring>
#include <cstddef>
#include <algorithm>
#include <vector>
#include <queue>

using namespace std;
uint32_t sigema[65];
uint32_t alpha[65][65];
uint32_t s1[10001], s2[10001];
uint32_t m, n1, n2;

class ArrayGraph{
    struct edges{
        edges(): has(vector<bool>(3, false)) {}
        uint32_t to[3];
        uint32_t weight[3];
        vector<bool> has;
    };

    struct compare{
        bool operator()(const pair<uint32_t, uint32_t>& l, const pair<uint32_t, uint32_t>& r) {
            return l.second > r.second;
        }
    };

public:
    vector<edges> E;
    int n;
    ArrayGraph(uint32_t N) : n(N),
        E(vector<edges>(N)){}
    ~ArrayGraph(){}

    void addEdges(uint32_t i, uint32_t j){
        uint32_t u = i * (n2 + 1) + j;

        uint32_t left = u + 1;
        uint32_t down = (i + 1) * (n2 + 1) + j;
        uint32_t oblique = (i + 1) * (n2 + 1) + j + 1;

        if(j < n2)
            insertEdge(u, left, 0, sigema[s2[j + 1]]);
        if(i < n1)
```

```
        insertEdge(u, down, 1, sigema[s1[i + 1]]);
    if(i < n1 && j < n2)
        insertEdge(u, oblique, 2, alpha[s1[i + 1]][s2[j + 1]]);
}

void insertEdge(uint32_t u, uint32_t v, uint32_t id, uint32_t weight){
    E[u].to[id] = v;
    E[u].weight[id] = weight;
    E[u].has[id] = true;
}

uint32_t getMinW(){
    return dijkstra(0, n1 * (n2 + 1) + n2);
}

uint32_t dijkstra(uint32_t source, uint32_t target){
    priority_queue<pair<uint32_t, uint32_t>, vector<pair<uint32_t, uint32_t>>, compare> Q;
    vector<uint32_t> dist(n, UINT32_MAX);
    vector<bool> visisted(n, false);
    dist[source] = 0;
    Q.push(make_pair(source, 0));

    while(!Q.empty()){
        auto u = Q.top();
        Q.pop();
        if(visisted[u.first] == true){
            continue;
        }
        visisted[u.first] = true;
        for(int i = 0; i < 3; ++i){
            if(E[u.first].has[i] == false){
                continue;
            }
            uint32_t v =    E[u.first].to[i];
            uint32_t w =    E[u.first].weight[i];
            uint32_t alt = dist[u.first] + w;
            if(alt < dist[v]){
                dist[v] = alt;
                Q.push(make_pair(v, dist[v]));
            }
        }
    }
    return dist[target];
}
```

```
};
int main(){
    while(std::cin >> m){
        for(uint32_t i = 0; i < m; ++i){
            std::cin >> sigema[i];
        }

        for(uint32_t i = 0; i < m; ++i){
            for(uint32_t j = 0; j < m; ++j){
                std::cin >> alpha[i][j];
            }
        }

        std::cin >> n1;
        for(uint32_t i = 1; i <= n1; ++i){
            std::cin >> s1[i];
        }

        std::cin >> n2;
        for(uint32_t i = 1; i <= n2; ++i){
            std::cin >> s2[i];
        }

        ArrayGraph G((n1 + 1)* (n2 + 1));
        for(int i = 0; i <= n1; ++i){
            for(int j = 0; j <= n2; ++j){
                G.addEdges(i, j);
            }
        }
        cout << G.getMinW() << endl;
    }
    return 0;
}
```

DP vs SP in Sequence Alignment

SP 方法在大算例上直接超时，而在较大算例比 DP 算法慢将近 100 倍，从复杂度分析我们很容易看到原因，我们也可以了解，在这个问题使用最短路算法有点“杀鸡焉用牛刀”。