动态规划

1 什么是动态规划

动态规划(Dynamic Programming，DP)是一种用来解决一类**最优化问题**的算法思想。简单来说，动态规划将一个复杂的问题分解成若干个子问题，通过综合子问题的最优解来得到原问题的最优解。

一个问题必须拥有重叠子问题和最优子结构，才能使用动态规划去解决。

- **重叠子问题**
- 如果一个问题可以被分解成若干个子问题，且这些子问题会重复出现，那么就称这个问题拥有重叠子问题。
- 动态规划通过记录重叠子问题的解，来使下次碰到相同的子问题时，直接使用之前记录的结果，以此避免大量重复计算。
- **最优子结构**
- 如果一个问题的最优解可以由其子问题的最优解有效的构造出来，那么称这个问题拥有最优子结构。

状态的**无后效性**是指，当前状态记录了历史信息，一旦状态确定，就不会再改变，且未来的决策只能在已有的一个或者若干个状态的基础上进行，历史信息只能通过已有的状态去影响未来的决策。

如何设计状态和状态转移方程，才是动态规划的核心，也是最难的地方。

2 动态规划的递归写法

以下是斐波那契数列的递归算法；事实上，这个递归算法会涉及到很多重复的计算，导致算法的实际复杂度会高达O(2^n^)。

```
int F(int n){
  if (n == 0 || n == 1) return 1;
  else return F(n-1) + F (n-2);
}
```

为了避免重复计算，可以开一个一维数组，用以保存已经计算过的结果，其中dp[n]记录F(n)的结果，并用dp[n]=-1表示F[n]当前还没有被计算过。此时的算法复杂度为O(n)。

动态归划的递归写法在此处又称作记忆化搜索。

```
int dp[MAXN];
fill(dp, dp + MAXN - 1, -1);
int F(int n){
  if (n == 0 || n == 1) return 1;     //递归边界
  if (dp[n] != -1) return dp[n];     //已经计算过，直接返回结果，不再重复计算
  else {
    dp[n] = F(n-1) + F(n-2);     //计算F(n),并保存至dp[n]
    return dp[n];
  }
}
```

3 动态规划的递推写法

数塔问题

```
#include <cstdio>
#include <iostream>
#include <algorithm>
using namespace std;
const int MAXN = 1000;
int f[MAXN][MAXN], dp[MAXN][MAXN];

int main(){
    int n;
    cin >> n;
    //输入数塔中的树
    for (int i = 1; i <= n; i++){
        for (int j = 1; j <= i; j++){
            cin >> f[i][j];
```

```
        }
    }
    //边界
    for (int j = 1; j <= n; j++){
        dp[n][j] = f[n][j];
    }
    //状态转移方程(从下往上)
    for (int i = n - 1; i >= 1; i--){
        for (int j = 1; j <= i; j++){
            dp[i][j] = max(dp[i + 1][j], dp[i + 1][j + 1]) + f[i][j];
        }
    }
    printf("%d", dp[1][1]);
    return 0;
}
```

4 最大连续子序列和

暴力解法时间复杂度为O(n^2^)(枚举左右端点i, j)，动态规划解法时间复杂度为O(n)

边界：dp[0] = a[0]

状态转移方程：dp[i] = max{a[i], dp[i-1] + a[i]}

```
#include <iostream>
#include <algorithm>
using namespace std;
const int MAXN = 10010;
int a[MAXN], dp[MAXN];

int main(){
    int n;
    cin >> n;
    for (int i = 0; i < n; i++){
        cin >> a[i];
    }
    dp[0] = a[0];
    for (int i = 1; i < n; i++){
        dp[i] = max(dp[i - 1] + a[i], a[i]);
    }
    int k = 0;
    for (int i = 1; i < n; i++){
        if (dp[i] > dp[k]){
            k = i;
        }
    }
    cout << dp[k];
    return 0;
}
```

练习题：

- PAT A1007 Maximum Subsequence Sum (25)

5 最长不下降子序列

暴力解法枚举时间复杂度为O(2^n^)，动态规划解法时间复杂度为O(n^2^)

dp[i]表示以a[i]结尾的最长不下降子序列长度

边界：dp[i] = 1 (1<= i <= n)

状态转移方程：dp[i] = max{1, dp[j] + 1} (j = 1, 2, …, i - 1&& a[j] < a[i])

```
#include <iostream>
#include <algorithm>
```

```
using namespace std;
const int MAXN = 1000;
int a[MAXN], dp[MAXN];

int main() {
    int n;
    cin >> n;
    for (int i = 1; i <= n; i++){
        cin >> a[i];
    }
    int ans = -1;
    for (int i = 1; i <= n; i++){
        dp[i] = 1;
        for (int j = 1; j < i; j++){
            if (a[i] >= a[j] && dp[j] + 1 > dp[i]){
                dp[i] = dp[j] + 1;
            }
        }
        ans = max(ans, dp[i]);
    }
    cout << ans;
    return 0;
}
```

练习题：

- PAT A1045 Favorite Color Stripe (30)

6 最长公共子序列

暴力解法时间复杂度O(2^m+n^ x max(m, n)，无法承受数据大的情况。动态规划的时间复杂度O(nm)

用dp[i][j]表示字符串A的i号位和字符串B的j号位之前的LCS长度(下标从1开始)

边界：dp[i][0] = dp[0][j] = 0 (0<= i <= n, 0 <= j <= m)

状态转移方程：

dp[i][j] = dp[i-1][j-1]+1 (if A[i] == B[j])

dp[i][j] = max{dp[i][j-1], dp[i-1][j]} (if A[i] != B[j])

```
#include <iostream>
#include <string>
#include <algorithm>
using namespace std;
const int MAXN = 110;
int dp[MAXN][MAXN];
int main(){
    string A, B;
    cin >> A >> B;
    //边界
    for (int i = 0; i <= A.size(); i++){
        dp[i][0] = 0;
    }
    for (int j = 0; j <= B.size(); j++){
        dp[0][j] = 0;
    }
    //状态转移方程
    for (int i = 1; i <= A.size(); i++){
        for (int j = 1; j <= B.size(); j++){
            if (A[i - 1] == B[j - 1]){
                dp[i][j] = dp[i-1][j-1] + 1;
            } else {
                dp[i][j] = max(dp[i-1][j], dp[i][j-1]);
            }
        }
    }
    cout << dp[A.size()][B.size()];
```

```
    return 0;
}
```

7 最长公共子串长度

暴力解法时间复杂度O(n^3^)，动态规划解法时间复杂度O(n^2^)

dp[i][j]表示以str1[i]前一个字符和str2[j]之前一个字符结尾的连续公共子串长度。

边界：

dp[i][j] = 0; (if i = 0或j = 0)

状态转移方程：

dp[i][j] = 0; (if str1[i] != str2[j])

dp[i][j] = dp[i-1][j-1] + 1; (if str1[i] == str2[j])

```
#include <iostream>
#include <string>
#include <cstdio>
using namespace std;
const int MAXN = 10010;
int dp[MAXN][MAXN];

int main() {
    string str1, str2;
    cin >> str1 >> str2;
    //边界
    for (int i = 0; i < str1.size(); i++){
        dp[i][0] = 0;
    }
    for (int j = 0; j < str2.size(); j++){
        dp[0][j] = 0;
    }
    //状态转移方程
    int max = -1;
    for (int i = 1; i <= str1.size(); i++){
        for (int j = 1; j <= str2.size(); j++){
            if (str1[i - 1] != str2[j - 1]){
                dp[i][j] = 0;
            } else {
                dp[i][j] = dp[i-1][j-1] + 1;
            }
            if (dp[i][j] > max){
                max = dp[i][j];
            }
        }
    }
    cout << max;
    return 0;
}
```

8 最长回文串

暴力解法时间复杂度O(n^3^)， 动态规划解法时间复杂度O(n^2^)

dp[i][j]表示s[i]到s[j]所表示的子串是不是回文串，如果是，则为1，不是则为0

边界：dp[i][i] = 1, dp[i][i+1] = (S[i] == S[i+1]) ? 1:0

状态转移方程：

dp[i][j] = dp[i+1][j-1] (if s[i] == s[j])

dp[i][j] = 0 (if s[i] != s[j])

```
#include <iostream>
#include <string>
using namespace std;
const int MAXN = 1000;
int dp[MAXN][MAXN];

int main(){
    string s;
    cin >> s;
    int ans = 1;
  //边界
    for (int i = 0; i < s.size(); i++){
        dp[i][i] = 1;
        if (i < s.size() - 1){
            if (s[i] == s[i+1]){
                dp[i][i+1] = 1;
                ans = 2;
            }
        }
    }
  //状态转移方程
    for (int L = 3; L < s.size(); L++){
        for (int i = 0; i + L - 1 < s.size(); i++){
            int j = i + L - 1;
            if (s[i] == s[j] && dp[i+1][j-1] == 1){
                dp[i][j] = 1;
                ans = L;
            }
        }
    }
    cout << ans;
    return 0;
}
```

9 DAG最长路

```
int DP(int i){
  if (dp[i] > 0) return dp[i];
  for (int i = 0; j < n; j++){
    if(G[i][j] != INF){
      int temp = DP(j) + G[i][j];
      if (temp > dp[i]) {
        dp[i] = temp;
        choice[i] = j;
      }
    }
  }
  return dp[i];
}

void printPath(int i) {
  printf("%d", i);
  while (choice[i]!=-1){
    i = choice[i];
    printf("->%d", i);
  }
}
```

10 背包问题

10.1 01背包问题

问题描述：有n件物品，每件物品的重量为w[i]，价值为c[i]。现有一个容量为V的背包，问如何选取物品放入背包，使得背包内

物品的总价值最大。其中每种物品都只有一件。

用二维数组存储(时间和空间复杂度都是O(nV))：

dp[i][v]表示前i件物品放入容量为v的背包中所能获得的最大价值。

边界：dp[0][v] = 0 (0<=v<=V)

状态转移方程：dp[i][v] = max{dp[i-1][v], dp[i-1][v - w[i]] + w[i]} (1<=i<=n, w[i]<=v<=V)

```
#include <iostream>
using namespace std;
const int MAXN = 100;
int dp[MAXN][MAXN], wei[MAXN], val[MAXN];

int main() {
    int n, V;
    cin >> n >> V;
    for (int i = 1; i <= n; i++) cin >> wei[i];
    for (int i = 1; i <= n; i++) cin >> val[i];
  //边界
  for (int i = 0; i <= n; i++){
    dp[i][0] = 0;
  }
  for (int v = 0; v <= V; v++){
    dp[0][v] = 0;
  }
  //状态转移函数
    for (int i = 1; i <= n; i++){
        for (int v = 1; v <= V; v++){
            if (wei[i] > v) dp[i][v] = dp[i-1][v];
            else dp[i][v] = max(dp[i-1][v], dp[i-1][v-wei[i]] + val[i]);
        }
    }
    for (int i = 0; i <= n; i++){
        for (int j = 0; j <= V; j++){
            cout << dp[i][j] << " ";
        }
        cout << endl;
    }
    cout << dp[n][V];
    return 0;
}
```

用一维数组存储，时间复杂度是O(nV)，空间复杂度是O(V)

边界：dp[v] = 0 (0<=v<=V)

状态转移方程：dp[v]=max(dp[v], dp[v-w[j]]+c[i]) (v逆序，从V到0)

```
#include <iostream>
using namespace std;
const int MAXN = 100;
int wei[MAXN], val[MAXN];

int main() {
    int n, V;
    cin >> n >> V;
    for (int i = 1; i <= n; i++) cin >> wei[i];
    for (int i = 1; i <= n; i++) cin >> val[i];
    //边界
    int dp[MAXN];
    for (int v = 0; v <= V; v++){
        dp[v] = 0;
    }
    //状态转移函数
    for (int i = 1; i <= n; i++){
    //注意：一定要从后往前遍历，不然的话会出错
        for (int v = V; v >= wei[i]; v--){
```

```
            dp[v] = max(dp[v], dp[v - wei[i]] + val[i]);
        }
    }

    for (int i = 1; i <= V; i++){
            cout << dp[i] << " ";
    }
    cout << dp[V];
    return 0;
}
```

10.2 完全背包问题

问题描述：有n种物品，每种物品的单件重量为w[i]，价值为c[i]。现有一个容量为V的背包，问如何选取物品放入背包，使得背包内物品的总价值最大。其中每种物品都有无穷件。

二维：

边界：dp[0][v] = 0

状态转移方程：dp[i][v] = max{dp[i-1][v], dp[i][v - w[i]] + w[i]} (1<=i<=n, w[i]<=v<=V)

一维：

边界：dp[v] = 0 (0<=v<=V)

状态转移方程：dp[v]=max(dp[v], dp[v-w[j]]+c[i]) (v顺序，从wei[i]到V)

```
//边界
for (int v = 0; v <= V; v++){
  dp[v] = 0;
}
//状态转移函数
for (int i = 1; i <= n; i++){
  //从前往后
  for (int v = wei[i]; v < V; v--){
    dp[v] = max(dp[v], dp[v - wei[i]] + val[i]);
  }
}
```