动态规划（Dynamic Programming，DP）技术通常与记忆化（memoization）技术结合起来使用。

# 思想

动态规划（DP）的基本思想和策略：将待求解的问题分解为若干个子问题，按顺序求解子阶段，前一子问题的解，为后一子问题的求解提供了有用的信息。

复杂问题不能分解成几个子问题，而分解成一系列子问题；动态规划（DP）通常基于一个递推公式及一个(或多个)初始状态，当前子问题解由上一次子问题解推出。

适合于用动态规划求解的问题，经分解后得到的子问题往往**不是相互独立的**。

状态 状态转移方程 递推关系

动态规划算法的关键在于解决冗余，以空间换时间的技术，需要存储过程中的各种状态。可以看着是分治算法+解决冗余使用动态规划算法的问题的特征是子问题的重叠性，否则动态规划算法不具备优势。

## 动态规划VS分治

动态规划与分治的区别在于，分治法所要处理的那些子问题之间并没有依赖关系，而动态规划所要处理的子问题却是有所重叠的，因此，可以把已经解决的子问题保存到表格里，这就是记忆化技术。运用这种技术，算法可以把很多问题的复杂度由指数级别降至O(n2)、O(n3)这样的多项式级别。

## 动态规划VS递归计数

动态规划技术与（分治算法中的）递归计数相比，其区别在于，它会把已经解决的子问题放在表格中，以免去重复的计算；而分治算法所要递归解决的那些子问题，彼此之间不重复。由此可见，并非所有的问题都适合用动态规划技术来解决。

# 条件

动态规划不能解决所有问题，需要满足下面的条件：

1. 具备最优的子结构：整个问题的最佳解法可以由各个子问题的最佳解法所构成；
2. 具备相互重叠的子问题：在运用递归来解决问题的过程中，有几个问题会反复出现。

另外一种描述，能采用动态规划求解的问题一般具有三个性质：

1. 最优化原理：问题最优解包含的子问题的解也是最优解，称为最优子结构
2. 无后效性：某个阶段的状态一旦确定，就不再受该状态以后决策的影响，只与当前状态相关
3. 有重叠子问题：子问题之间不相互独立，一个子问题可能在后续的决策中多次被使用

# 基本步骤

1、划分问题/划分阶段

2、确定状态和状态变量

3、确定决策并写出状态转移方程（关键）

4、写出规划方程/寻找边界条件（找到递推式）

# 使用场合

## 爬楼梯

## 找零钱

## 最大字段和

## 三角形

## 最长上升子序列

## 字符串解码

题目要求：一个包含字母的消息被加密之后变成了一个只包含数字的字符串，但是我们现在知道加密的规则：

‘A’🡪1

‘B’🡪2

……

‘Z’🡪26

现在给定一个已经被加密的只包含数字的字符串，求出该字符串有多少种被解密的方法。例如“12”🡪AB或者12🡪L。

代码：

#include <iostream>

#include <string>

#include <vector>

using namespace std;

/\*

思路：

这是一个典型的DP问题，假设定义一个数组，dp[i]为到第i个字符所能够组成的所有编码方式的个数。那么对于dp[i+1]来说，肯定至少和dp[i]一样多，如果第i个字符和i+1个字符可合成一个字符，那么dp[i+1] += dp[i-1].

\*/

int Decode\_num(string& str)

{

vector<int> vec(str.size(),1);

if(str.size() <2)

return 1;

if(str[0]=='1'||(str[0]=='2'&& str[1]<='6'))

vec[1] =2;

int i;

int tmp;

for(i=2;i<str.size();i++)

{

if(str[i] >= '0' && str[i] <= '9')//判断是合法的字符

vec[i] = vec[i-1];

else

return 0;

tmp = str[i-1] -'0';

tmp = tmp\*10 + str[i]-'0';

if(str[i-1]!='0' && tmp <=26)

vec[i] += vec[i-2];

else

vec[i] = vec[i-1];

}

return vec[str.size()-1];

}

int main()

{

string str("1231725");

cout<<Decode\_num(str)<<endl;

return 0;

}

## 寻找最长的共同子序

## 路径数目&最小路径和

题目：求矩阵中从左上角到右下角的路径数目

求矩阵中左上角到右下角最小路径和

代码：

#include <iostream>

#include <vector>

#include <limits>

using namespace std;

/\*

思路：对于某一点dp[i][j]的路径数目，是该点正上方和正左方路径数目之和

dp[i][j] = dp[i][j-1] + dp[i-1][j]; 但是对于特殊地方需要特殊考虑

\*/

int Unique\_path(int m,int n,int first,int second)

{

vector<vector<int> > dp(m);

int i,j;

for(i=0;i<dp.size();i++)

dp[i].assign(n,0);

dp[0][0] =1;

for(i=0;i<dp.size();i++)

{

for(j=0;j<dp[0].size();j++)

{

if(i!=0 || j!=0)

{

if(i == first && j == second)

dp[i][j] =0;

else

{

if(i == 0)

dp[i][j] = dp[i][j-1];

else if(j== 0)

dp[i][j] = dp[i-1][j];

else

dp[i][j] = dp[i][j-1]+dp[i-1][j];

}

}

}

}

return dp[m-1][n-1];

}

/\*

第二个问题，从左上角到右下角，寻找代价最小的路径

典型的动态规划问题，和上个问题类似

\*/

int MinPathSum(vector<vector<int> >& vec)

{

vector<vector<int> > dp(vec.size());

int i,j;

for(i=0;i<vec.size();i++)

dp[i].assign(vec[i].size(),numeric\_limits<int>::max());

dp[0][0] = vec[0][0];

for(i=1;i<vec.size();i++)

dp[i][0] = vec[i][0]+dp[i-1][0];

for(j=1;j<vec[0].size();j++)

dp[0][j] = vec[0][j] + dp[0][j-1];

int tmp;

for(i=1;i<vec.size();i++)

{

for(j=1;j<vec[0].size();j++)

{

tmp = min(vec[i][j]+dp[i][j-1],vec[i][j]+dp[i-1][j]);

dp[i][j] = min(dp[i][j],tmp);

}

}

return dp[vec.size()-1][vec[0].size()-1];

}

int main()

{

// cout << Unique\_path(3,7,2,3)<<endl;

vector<vector<int> > vec(3);

int i,j;

int array[]={2,4,3,7};

int array1[]={5,3,2,1};

int array2[]={4,8,6,2};

vec[0].assign(array,array+4);

vec[1].assign(array1,array1+4);

vec[2].assign(array2,array2+4);

cout<<MinPathSum(vec)<<endl;

return 0;

}

## 最大子数组乘积

题目：给定一个整数数组，求乘积最大的子数组的值。

代码：

#include <iostream>

#include <vector>

#include <string>

using namespace std;

/\*

最大子串乘积，由于可能出现负数。也是DP问题，也是局部最优和全局最优问题。

这里需要记录最小值，假设有两个数组，分别记录包括当前元素在内的子串所能构成的最大和最小值，然后根据这个再更新全局最大，至于当前最大，可能是之前最大乘以当前元素，也可能是前一个元素最小乘以当前元素，也可能是当前元素

\*/

int maxProduct(vector<int>& vec)

{

if(vec.size()==0)

return 0;

vector<int> maxcur(vec.size(),0);

vector<int> mincur(vec.size(),0);

maxcur[0]=vec[0];

mincur[0]=vec[0];

int maxproduct = vec[0];

int i,temp;

for(i=1;i<vec.size();i++)

{

maxcur[i] = max(vec[i],max(maxcur[i-1]\*vec[i],mincur[i-1]\*vec[i]));

mincur[i] = min(vec[i],min(mincur[i-1]\*vec[i],maxcur[i-1]\*vec[i]));

maxproduct = max(maxcur[i],maxproduct);

}

return maxproduct;

}

int main()

{

int array[] ={2,3,-2,4};

vector<int> vec(array,array+sizeof(array)/sizeof(int));

cout<<maxProduct(vec)<<endl;

return 0;

}

## 链式矩阵乘法

## 子集和问题

## 0/1背包问题

## 旅行推销员问题

## 最长递增子序列

最长递增子序列（LIS Longest Increasing Subsequence）

## 编辑距离

题目要求：给定两个字符串word1和word2，求出最少需要多少个步骤可以将word1转化为word2，其中每一个操作都被记为一步，每个操作可以是：

1. 插入一个字符
2. 删除一个字符
3. 替换一个字符

代码：

#include <iostream>

#include <vector>

#include <string>

#include <limits>

using namespace std;

/\*

典型的DP问题，这里需要使用S来匹配T， 利用动态规划思想

使用dp[i][j]表示S与T的前i个字符与前j个字符的匹配子串个数。

1）初始条件：T为空串时，S为任意字符串都能匹配一次，所以dp[i][0]=1

S为空字符串，T不为空时，不能匹配，dp[0][j] =0

2)S[i] == T[j]，dp[i][j] = dp[i-1][j-1]+dp[i-1][j].表示当前字符可以保留也可以抛弃

3）S[i] != T[j]，dp[i][j] = dp[i-1][j-1]

\*/

int Distinct\_sub(string& src,string& dst)

{

vector<vector<int> > dp(src.size());

int i,j;

for(i=0;i<src.size();i++)

dp[i].assign(dst.size(),0);

if(src[0] == dst[0])

dp[0][0] =1;

for(j=1;j<dp.size();j++)

dp[j][0] = src[j]==dst[0]? dp[j-1][0]+1:dp[j-1][0];

for(i=1;i<dp.size();i++)

{

for(j=1;j<dp[0].size();j++)

{

if(src[i] == dst[j])

dp[i][j] = dp[i-1][j-1]+dp[i-1][j];

else

dp[i][j] = dp[i-1][j];

}

}

return dp[src.size()-1][dst.size()-1];

}

/\*

编辑距离问题

使用便利dp[i][j]记录包括word1[i]在内的字符串和word2[j]在内的字符串的编辑距离，如果word1[i+1] == word2[j+1] dp[i+1][j+1] = dp[i][j]，否则dp[i+1][j+1] = dp[i][j]+1,不过也有可能dp[i+1][j+1] = dp[i+1][j]+1或者dp[i][j+1]+1 取三者最小

\*/

int Edit\_distance(string& s1,string& s2)

{

vector<vector<int> > dp(s1.size());

int i,j;

for(i=0;i<dp.size();i++)

dp[i].assign(s2.size(),numeric\_limits<int>::max());

if(s1[0]==s2[0])

dp[0][0] =0;

else

dp[0][0] =1;

for(i=1;i<dp[0].size();i++)

if(s1[0] == s2[i])

dp[0][i] = i;

else

dp[0][i] = dp[0][i-1]+1;

for(i=1;i<dp.size();i++)

if(s1[i] == s2[0])

dp[i][0] = i;

else

dp[i][0] = dp[i-1][0]+1;

for(i=1;i<dp.size();i++)

{

for(j=1;j<dp[0].size();j++)

{

if(s1[i] == s2[j])

dp[i][j] = dp[i-1][j-1];

else

dp[i][j] = min(dp[i-1][j-1]+1,min(dp[i][j-1]+1,dp[i-1][j]+1));

}

}

return dp[s1.size()-1][s2.size()-1];

}

int main()

{

string src("rabbbit");

string dst("rabbit");

cout<<Distinct\_sub(src,dst)<<endl;

string s1("abcd");

string s2("abc");

cout<<Edit\_distance(s1,s2)<<endl;

return 0;

}

## 盛水最大化

题目要求：给出一系列非负整数a1，a2，…，an，每一个数都代表数轴上的一个点(i,ai)，那么这n个垂直线中的任意两个都可以组成一个区间，然后和x轴可以构成一个容器，求出可以盛水最多的容器的两条边。

代码：

#include <iostream>

#include <vector>

#include <string>

#include <stack>

using namespace std;

/\*

夹逼方法

从数组的两段走起，每次迭代时判断左边点和右边点指向的数字哪个大如果左边点下，就意味着左移动右边点不可能使得结果变得更好因为瓶颈在左边点，移动右边点只会变小，所以这个时候我们选择左边点右移反之，选择右边点左移，在这个过程中一直维护最大的容积

\*/

int area(vector<int> &height, int i, int j)

{

int h = height[i]<height[j]?height[i]:height[j];

return h\*(j-i);

}

int maxArea(vector<int> &height)

{

int max=0;

for(int i=0;i<height.size();i++)

{

for(int j=i+1;j<height.size();j++)

{

int a = area(height,i,j);

if(a>max)

max=a;

}

}

return max;

}

/\*

第二种方法

\*/

int maxarea(vector<int>& vec)

{

int maxarea=0;

int first,second;

int i=0,j=vec.size()-1;

while( i<j)

{

if(min(vec[i],vec[j])\*(j-i) > maxarea)

{

maxarea = min(vec[i],vec[j])\*(j-i);

}

if(vec[i] < vec[j])

i++;

else

j--;

}

return maxarea;

}

/\*

|

| \_\_

| | |

| \_\_ | |\_\_ \_\_

| | | | | | | |

| \_\_ | |\_\_ \_\_ | | |\_\_| |\_\_

| | | | | | | | | | | | | |

|\_\_|\_\_|\_\_|\_\_|\_\_|\_\_|\_\_|\_\_|\_\_|\_\_|\_\_|\_\_|\_\_|\_\_\_\_\_

1 2 3 4 5 6 7 8 8 9 10 11 12

上述给定的一个序列为[0,1,0,2,1,0,1,3,2,1,2,1]，每个元素代表柱子的高度

最后函数的返回值为6

思路：可以在这个序列中找到最高的柱子位置，那么从两头开始找可以

盛水的多少，假如从头开始遍历，需要遍历到柱子最高的位置，遍历到当前位置如果发现当前的柱子比之前记录的柱子高，那么更新如果没有之前记录的柱子高，那么就可以计算当前柱子相对之前的高柱子的盛水量

\*/

int TrapRainWater(vector<int>& vec)

{

int i,maxhigh;

maxhigh = 0;

int left=0,right = 0;

int sum =0;

for(i=0;i<vec.size();i++)

if(vec[i] > vec[maxhigh])

maxhigh = i;

for(i=0;i<maxhigh;i++)

{

if(vec[i] < left)

sum +=(left-vec[i]);

else

left = vec[i];

}

for(i=vec.size()-1;i>maxhigh;i--)

{

if(vec[i]<right)

sum += (right-vec[i]);

else

right = vec[i];

}

return sum;

}

int main()

{

// int array[]={4,3,4,5,7,9,7,6,8,5,3,2};

// vector<int> vec(array,array+sizeof(array)/sizeof(int));

// cout<<maxArea(vec)<<endl;

int array[]={0,1,0,2,1,0,1,3,2,1,2,1};

vector<int> vec(array,array+sizeof(array)/sizeof(int));

cout<<TrapRainWater(vec)<<endl;

return 0;

}

## 股票利润最大化

题目要求：给定一个整数数组，数组中的每个元素都是某支股票的当天的价钱，设计一个算法来找出这支股票的最大利润。你至少可以进行K次交易。

代码：

/\*

股票买卖最大利润

这里维护两个变量，一个是当前到达第i天可以最多进行j此交易，最好的利润是多少（global[i][j]）另一个是当前到达第i天，最多可以进行j此交易，并且最后一次交易在dangt卖出那么最好的利润是多少（local[i][j]）

递推公式

global[i][j] = ma(local[i][j],glboal[i-1][j]),

也就是取当前局部最好和过往全局最好的其中之一对于局部最好

local[i][j] = max(global[i-1][j-1]+maxdiff(diff,0),local[i-1][j]+diff);

\*/

/\*

在进行两次交易的利润最大化

\*/

int maxProfit(vector<int>& prices)

{

if(prices.size() <= 0)

return 0;

int global[3];

int local[3];

for(int i=0;i<prices.size()-1;i++)

{

int diff = prices[i+1]-prices[i];

for(int j=2;j>=1;j--)

{

local[j] = max(global[j-1]+(diff>0?diff:0),local[j]+diff);

global[j] = max(local[j],global[j]);

}

}

global[2];

}

/\*

多次交易之后

\*/

int helper(vector<int>& prices,int k)

{

int len = prices.size();

if(len == 0)

return 0;

int local[10][10];

int global[10][10];//临时申请的空间

for(int i=1;i<len;i++)

{

int diff = prices[i]-prices[i-1];

for(j=1;j<=k;j++)

{

local[i][j] = max(global[i-1][j-1]+max(diff,0),local[i-1][j]+diff);

glocal[i][j] = max(local[i][j],global[i-1][j]);

}

}

}

int maxProfit(vector<int>& prices)

{

return helper(prices,2);

}