**动态规划**

<https://blog.csdn.net/mmc2015/article/details/73558346>

<http://www.hawstein.com/posts/dp-novice-to-advanced.html>

<https://blog.csdn.net/together_cz/article/details/77519654>

<https://www.cnblogs.com/wuyuegb2312/p/3281264.html>

# 概念：

动态规划算法通常基于一个递推公式及一个或多个初始状态。 当前子问题的解将由上一次子问题的解推出。

使用动态规划来解题只需要多项式时间复杂度， 因此它比回溯法、暴力法等要快许多。

通过把原问题分解为相对简单的子问题的方式求解复杂问题的方法。动态规划常常适用于有重叠子问题和最优子结构性质的问题。

**基本思想**

若要解一个给定问题，我们需要解其不同部分（即子问题），再合并子问题的解以得出原问题的解。 通常许多子问题非常相似，为此动态规划法试图仅仅解决每个子问题一次，从而减少计算量： **一旦某个给定子问题的解已经算出，则将其记忆化存储**，以便下次需要同一个子问题解之时直接查表。 这种做法在重复子问题的数目关于输入的规模呈指数增长时特别有用。

**分治与动态规划**

**共同点：**

二者都要求原问题具有最优子结构性质,都是将原问题分而治之,分解成若干个规模较小(小到很容易解决的程序)的子问题.然后将子问题的解合并,形成原问题的解.

**不同点：**

分治法将分解后的子问题看成相互独立的，通过用递归来做。

动态规划将分解后的子问题理解为相互间有联系，有重叠部分，需要记忆，通常用迭代来做。

使用动态规划**特征**：

1. 求一个问题的最优解

2. 大问题可以分解为子问题，子问题还有重叠的更小的子问题

3. 整体问题最优解取决于子问题的最优解（状态转移方程）

4. 从上往下分析问题，从下往上解决问题

5. 讨论底层的边界问题

## 1、硬币问题：

如果我们有面值为1元、3元和5元的硬币若干枚，如何用最少的硬币凑够11元？

状态转移方程：d(i)=min{ d(i-vj)+1 }，其中i-vj >=0，vj表示第j个硬币的面值;j=1,…,n表示硬币的种类。i表示需要凑齐i元的硬币。

int Coin(int money)

{

vector<int> kind{ 1, 3, 5 };//硬币种类

unsigned int\* min = new unsigned int[money];//存放使用硬币的最小个数

memset(min, 0xff, money \* 4);//初始化所有变量为最大值

min[0] = 0;//初始化0元时使用0个硬币

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

{

for (int j = 0; j < kind.size(); j++)

if (i >= kind[j] && min[i] > min[i - kind[j]]+1)

min[i] = min[i - kind[j]]+1;

}

delete[] min;

return min[money];

}

## 2、最长非降子序列：

一个序列有N个数：A[1],A[2],…,A[N]，求出最长非降子序列的长度。 (LIS：longest increasing subsequence)

解：d(i) = max{d(j)+1}/1,其中j<i,A[j]<=A[i]

想要求d(i)，就把i前面的各个子序列中， 最后一个数不大于A[i]的序列长度加1，然后取出最大的长度即为d(i)。 当然了，有可能i前面的各个子序列中最后一个数都大于A[i]，那么d(i)=1， 即它自身成为一个长度为1的子序列。

int LIS()//longest increasing subsequence{

vector<int> V = { 5, 3, 4, 8, 6, 7 };

vector<int> D;

int len = 1;

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

D.push\_back(1);

for (int j = 0; j < i; j++)

if (V[j] <= V[i] && D[i] < D[j] + 1)

D[i] = D[j] + 1;

if (D[i] > len) len = D[i];

}

return len;}

**2-1**、该算法的时间复杂度是O(n^2 )，并不是最优的解法。其他解法[LIS的O(nlogn)解法](http://www.felix021.com/blog/read.php?1587)。

解：我们定义一个序列B，然后令 i = 1 to 9 逐个考察这个序列。

如：d[1..9] = 2 1 5 3 6 4 8 9 7

在B中插入数据是有序的，而且是进行替换而不需要挪动——也就是说，我们可以使用二分查找，将每一个数字的插入时间优化到O(logN)