基础动态规划

动态规划的本质思想：对所进行的规划进行暴力，取最优值。

动态规划的三个基本要素：**阶段、状态、决策。**

阶段：将一个问题分解，分解成若干个子问题的层层推进。在求解每一个子问题的时候都构成了一个阶段。

状态：就是设的变量的含义。

决策：就是我们讲的状态转移方程。

动态规划三个基本条件：**子问题重叠性、无后效性和最优子结构。**

判断一个问题是不是能不能用动态规划来解决可以通过三个基本条件来判断。

解决动态规划的一般方法总结：

我是谁（定义状态），我从哪里来，我要到那里去？（状态转移的决策）。

线性dp

即具有线性阶段的动态规划算法称为线性dp。

1.最长上升子序列问题(LIS)

给你一个序列 $a_1, a_2, \ldots, a_n$，问你最长的上升子序列的长度是多少。

考虑三个基本条件，考虑三要素，进行解题。

阶段：对于第 i 个数组的元素，前 i -1 个数组的LIS都已近求出。

状态：问什么设什么，看一维够不够，不够在往后增加维度表示。f[i]:前i个的LIS长度。

决策：$f[i] = max_{0 \le j < i, a[j] < a[i]} f[j] + 1$;

```
for(int i = 1; i <= n; ++i) {
    f[i] = 1;
    for(int j = 1; j < i; ++j) {        //想想可以怎么优化
        if(a[j] >= a[i])    continue;
        f[i] = max(f[i], f[j] + 1);
    }
}
ans = max_element(f + 1, f + n + 1);
```

2.最长公共子序列问题（LCS）

给你两个序列 $a_1, a_2, \ldots, a_n$ 和 $b_1, b_2, \ldots, b_n$，问他们得最长公共序列长度为多少。

阶段：在找到 a_i和 b_j 的时候说明$a_1 \dots a_{i-1}$与$b_1 \dots b_j - 1$的LCS已经求出来，具有子问题。

状态：一维肯定没有办法表示两个数组的LCS，故令：$f[i][j]$ 表示$a_1 \dots a_i$与$b_1 \dots b_j - 1$ 的LCS长度。

决策：$f[i][j] = \begin{cases} f[i-1][j-1]+1 & a[i]=b[j] \\ max(f[i-1][j], f[i][j-1]) & a[i] \neq b[j] \end{cases}$

```
for (int i = 1; i <= n; i++) {      //n,m分别为两个数组的长度
    for (int j = 1; j <= m; j++) {
        if (a[i] == b[j])   dp[i][j] = dp[i-1][j-1] + 1;
        else    dp[i][j] = max(dp[i-1][j], dp[i][j-1]);
    }
}
ans = f[n][m];
```

3.数字三角形

给你一个n行的三角形矩阵，第 i 行有第 i 列，从(1, 1)坐标开始，每次可以向下走或者想右下走。每走到一格，就会获取那个单元的权值。问走到最后一行的最大权值和是多少。

阶段：求解第 i 行时，前 i - 1 行已经求解出来了，具有子问题。

状态：一维肯定不行，因为有行有列，故令 $f[i][j]$ 表示第 i 行第 j 列的最大权值和。

决策：$f[i][j] = max \begin{cases} f[i-1][j] + arr[i][j] \\ f[i-1][j-1] + arr[i][j] \quad if(j \geq 1) \end{cases}$

```
for(int i = 1; i <= n; ++i) {
    for(int j = 1; j <= i; ++j) {
        f[i][j] = max(f[i][j], f[i-1][j] + arr[i][j]);
        if(j >= 1)  f[i][j] = max(f[i][j], f[i-1][j-1] + arr[i][j]);
    }
}
ans = max_element(f[n] + 1, f[n] + n + 1);
```

区间dp

区间dp以区间长度为dp的阶段，它的子问题是在你求长度为 i 的最优解时，所有的长度为 $j(j <= i)$ 的最优解已近求出来了，它由比他长度更小的且包含它的区间转移而来。

石子合并问题

给你一堆石子，重量用一个数组 $a_1, a_2, \dots, a_n$表示，每次可以选择相邻的石子合并成新的一堆，新的石子堆的重量是两个原来石子的重量的和。问将它们合并成一堆所需要的最少体力。

阶段：明显求到长度为 i 的时候前面长度小于i 的最优解已经求出来了。

状态：区间dp一般至少都是二维的，$f[i][j]$ 表示合并起点为 i 终点为 j ，长度为 j-i+1 的长度的石子的最少体力。

决策：

$f[i][j] = min(f[i][j], f[i][k] + f[k+1][j] + w[i][j])$ 其中$i \leq k \&\& k < j, w[i][j]$表示合并第$i$堆到第$j$堆的体力和

```
for(int len = 2;len <= n; ++len) {          //先枚举长度len
    for(int i = 1; i + len - 1 <= n; ++i) {
        int j = i + len - 1;
        for(int k = i; k + 1 <= j; ++k) {
            dp[i][j] = min(dp[i][j], dp[i][k] + dp[k+1][j] + sum[j]-sum[i-1]);
        }
    }
}
ans = dp[1][n];
```

树形dp

树形dp即在树上进行动态规划，一般方法是先dfs一遍，从根节点多子节点，然后在回溯的时候，通过子节点的子结构对父节点的状态进行更新。即**以节点从深到浅，子树从小到大的顺序作为dp的阶段。**

没有上司的舞会

【题意】：有n个人用1~n表示，关系用一个树表示出来。每个人有一个开心值h[i](非负数)，现在要从这个树种选一些节点，选了一个节点就不能选他的父节点。问在这个树中能选出的最大的快乐和总数是多少。

阶段：对于第 i 个节点，其所有子树的

状态：一维肯定不行，因为一个点你有两种状态：选或不选，故加一维$f[i][j]$，其中 i 表示对于第 i 个点, j = 0表示第 i 个点不选，其子树加该点的最大快乐和, 而 j = 1则表示选了第 i 个点。

决策：$\begin{cases} f[i][0] = \sum_{j \in son(i)} max(f[j][0], f[j][1]) \\ f[i][1] = \sum_{j \in son(i)} max f[i][0] + h[i] \end{cases}$

```
void dfs(int u)
{
    f[u][0] = 0;
    f[u][1] = h[u];
    for(int i = head[u]; i != -1; i = Edge[i].nex) {
        int v = Edge[i].to;
        dfs(v);          //这里建的是单向边，如果是双向边的再加个v != fa这个条件
        f[u][0] += max(f[v][0], f[v][1]);
        f[u][1] = max(f[u][1], f[u][1] + f[v][0]);
    }
}
ans = max(f[rt][0], f[rt][1]);
```

二次扫描与换根

一般做树形dp的时候，是从树的根节点进行dfs，从而对问题进行求解。但是如果给你一个无根树，然后有很多组询问，如果对每一个询问都dfs求得话复杂度会很高，这时候我们就可以通过换根来求。 **第一次随便以一个点作为根自下而上跑一次树形dp，第二次则还以刚刚那个点作为根节点， 自上而下的找到以另外一点为根的权值和原来的树形dp跑出的权值之间的关系**，这种操作称为换根，对于每一个点换根的复杂度为O(1)。

【题意】：n个点有n-1条河，每条河连接两个点x, y，那条河道的容量是c(x, y)。有一个节点是源点，可以源源不断的流出水出来，然后有一个节点是汇点，可以接收无穷多的水。现在问以哪个点作为源点的话，可以使得这个河流系存储的流量最大并求出最大值。

树形dp中阶段：求以 i 为根的子树的最大流量，其子节点设为 j ，则以 j 为根的子树的最大流量肯定已经求出。

状态：可以设$dis[i]$表示以 i 根的子树的最大流量。

决策：$dis[i] = min_{j \in son(i)}\{val(i,j), dis[v]\}$

树形dp中的整个过程是自下而上，发生在dfs回溯时。

换根中的阶段：以 i 为根的最大流量，设其父节点为 j ，则在求以 i 为个根的最大流量时，以 j 为根的最大流量已经求出。

状态：$f[i]$表示以 i 为根的最大流量。

决策：$f[i] = \begin{cases} val(i,j) & size(j) = 1 \\ dis[j] + min(val(i,j), f[i] - min(dis[j], val(i,j))) & size(j) > 1 \end{cases}$

```
memset(dis, 0, sizeof(dis));
void dp(int u, int fa)          //树形dp,  自下而上
{
    for(int i = head[u]; i != -1; i = Edge[i].nex){
        int v = Edge[i].to;
        if(fa == v) continue;
        dp(v, u);
        if(de[v] == 1)  dis[u] += Edge[i].val;          //一定要注意边界条件!
        else dis[u] += min(Edge[i].val, dis[v]);
    }
}
void dfs(int u, int fa)          //换根,  每个点O(1)换根，n个点O(n)
{
    for(int i = head[u]; i != -1; i = Edge[i].nex){
        int v = Edge[i].to;
        if(fa == v) continue;                           //找到新根与原来的dis之间的关系
        if(de[u] == 1)  f[v] = Edge[i].val + dis[v];
        else f[v] = dis[v] + min(Edge[i].val, f[u] - min(Edge[i].val, dis[v]));
        dfs(v, u);
    }
}
ans = max_element(f + 1, f + n + 1);
```

数据结构优化dp

当我们在进行数据结构的决策时，当决策的候选集只扩大不缩小时，我们可以用一个变量来维护最值，每次与新加入的候选集进行比较，可以O(1)的得到最优决策，进而进行转移。

但是如果候选集不固定，比如每次的候选集都是该数前固定长度的最值等等，用一个变量无法维护时，我们可以考虑用特定的数据结构（如线段树/树状数组等）来维护候选集合，进而进行状态的转移。

【题意】：有n个贴纸，每个贴纸可以覆盖 $[L_i \dots R_i]$ 的区间并花费 C_i 元。问现在给你一个区间 L 和 R，将这些区间全部用贴纸覆盖完至少花费多少。

【思路】：首先看动态规划的三要素是否满足？分析设计动态规划的三要素；

状态：$f[i]$ 表示覆盖 [L, i] 的最少花费。

决策：$f[i] = min_{L[i]-1<=j<R[i]}\{f[j]\} + c[i]$

目标：$ans = min_{b_i \geq r}\{f[b[i]\}$

```
for(int i = 1; i <= n; ++i) {
    ans = inf;
    query(arr[i].l - 1, min(arr[i].r - 1, e), 1);   //线段树区间查询
    f[arr[i].r] = min(f[arr[i].r], ans + arr[i].val);
    modify(arr[i].r, f[arr[i].r], 1);
    if(arr[i].r >= e)   res = min(res, f[arr[i].r]);
}
cout << res;
```