讲稿

一. 搜索初步

引入

考虑这个例子，我们把正整数 n 分解成 3 个不同的正整数，且需要满足排在后面的数必须大于等于前面的数，要求输出所有的输出方案。
对于这个例子我们会怎么做呢？当然是循环。

```
for(int i = 1;i <= n;i++){
        for(int j = i;j <= n;j++){
                for(int k = 1;k <= n;k++){
                        if(i + j + k == n) printf("%d %d %d\n",i,j,k);
                }
        }
}
```

但是如果把三个数改成四个呢，改成更多呢，这时候我们就需要用到递归搜索了。该类搜索算法的特点在于，将要搜索的目标分成若干层，每层基于前几层的状态进行决策，直到达到目标状态。

解释

考虑这样一个问题，求出将一个正整数 n 分解成若干正整数的所有方案。
我们将问题分层，第 i 层决定 a_i ，则为了进行第 i 层的决策，我们需要记录下我们“剩下”的和以及我们当前进行的层数，也即 $n-\sum_{j=1}^{i-1}a_j$ 和 i 。

```
int ans[110];//ans[i]用来存放方案
void dfs(int n,int i){
        if(n == 0){
                for(int j = 1;j <= i - 1;j++)
                printf("%d ",ans[j]);
                printf("\n");
        }
        for(int j = 1;j <= n;j++){//枚举当前可以选的值
                ans[i] = j;
                dfs(n - j,i + 1);//递归到下一层
        }
}
```

调用时只需要 `dfs(n,1)` 就可以了。
我们考虑给这个问题多加一点限制，比如要求最后分解的数的个数不能超过 m 个，且要求每个数递增，那么我们就需要多记录下来一个状态变量： a_{i-1} ，也即上一个选择的数（为了保证递增的性质），此时我们的代码就变成了：

```
int m,ans[103];  // ans 用于记录方案
void dfs(int n,int i,int a) {
  if (n == 0) {
    for (int j = 1;j <= i - 1;j++) printf("%d ", ans[j]);
    printf("\n");
  }
  if (i <= m) {//保证个数不超过m个
    for (int j = a;j <= n;j++) {
      ans[i] = j;
      dfs(n - j, i + 1, j);  // 请仔细思考该行含义。
    }
  }
}
```

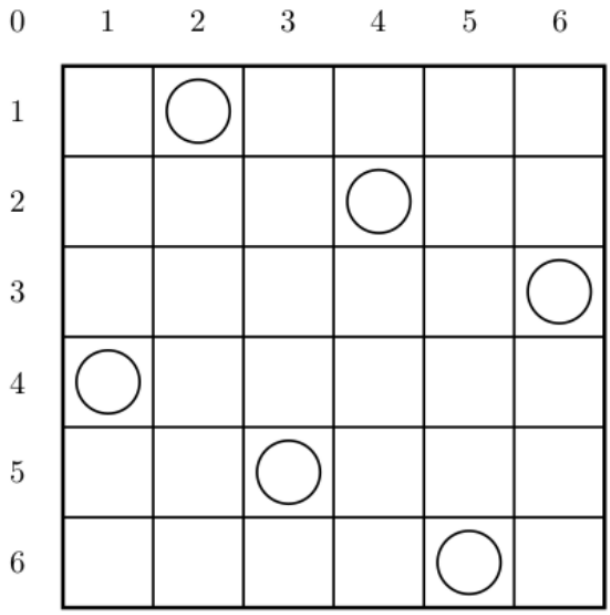
例题

例题一：按照字典序输出自然数 1 到 n 所有不重复的排列，即 n 的全排列，要求所产生的任一数字序列中不允许出现重复的数字。

```
int a[20];
void dfs(int step) {
  if (step == n + 1) {  // 边界
    for (int i = 1; i <= n; i++) {
      printf("%d",a[i]);
    }
    printf("\n");
    return;
  }
  for (int i = 1; i <= n; i++) {
    if (vis[i] == 0) {  // 判断数字i是否在正在进行的全排列中
      vis[i] = 1;
      a[step] = i;
      dfs(step + 1);
      vis[i] = 0;  // 这一步不使用该数 置0后允许下一步使用
    }
  }
  return;
}
```

例题二：

一个如下的 6×6 的跳棋棋盘，有六个棋子被放置在棋盘上，使得每行、每列有且只有一个，每条对角线（包括两条主对角线的所有平行线）上至多有一个棋子。



洛谷

上面的布局可以用序列 $2\ 4\ 6\ 1\ 3\ 5$ 来描述，第 i 个数字表示在第 i 行的相应位置有一个棋子，如下：

行号 $1\ 2\ 3\ 4\ 5\ 6$

列号 $2\ 4\ 6\ 1\ 3\ 5$

这只是棋子放置的一个解。请编一个程序找出所有棋子放置的解。
并把它们以上面的序列方法输出，解按字典顺序排列。
请输出前 3 个解。最后一行是解的总个数。

```
void dfs(int x)
{
        if(x > n)//边界
        {
                ans++;
                if(ans <= 3)//只输出前三组答案
                {
                        for(int i = 1;i <= n;i++)
                        printf("%d ",a[i]);
                        printf("\n");
                }
                return;
        }
```

```
        for(int i = 1;i <= n;i++)
        {
                if(x1[i] == 0 && x2[i + x] == 0 && x3[x - i + 15] == 0)//为了避免下
标出现负数所以+15
                {
                        a[x] = i;
                        x1[i] = 1,x2[i + x] = 1,x3[x - i + 15] = 1;//标记这一列和两个
对角线
                        dfs(x + 1);
                        x1[i] = 0,x2[i + x] = 0,x3[x - i + 15] = 0;//回溯
                }
        }
}
```

二. 动态规划基础

引入

动态规划是一种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。
动态规划通常用于最优化问题，如最短路径问题、调度问题、背包问题等。
和分治算法的区别：

- 分治算法：子问题之间**相互独立**，没有重叠的部分。
  举例：归并排序 快速幂
- 动态规划算法：子问题之间存在**重叠**的部分，即不同的子问题可能会多次使用相同的中间结果。
  和贪心算法的区别：
- 贪心算法：每一步的最优解一定包含上一步的最优解，上一步之前的最优解则不作保留。
  举例：均分纸牌删数问题
- 动态规划算法：全局最优解中一定包含某个局部最优解，但不一定包含前一个局部最优解，因此需要记录之前的所有的局部最优解。

思路

1.将原问题划分为若干阶段，每个阶段对应若干个子问题，提取这些子问题的特征（称之为状态）；
2.寻找每一个状态的可能决策，或者说是各状态间的相互转移方式（用数学的语言描述就是状态转移方程）。
3.按顺序求解每一个阶段的问题。
动态规划的求解的难点往往在于状态的设计和状态转移方程的推导上，在不同的问题中变化十分多样，需要我们在做题的时候积累，下面我们介绍两种常用的dp，大家可以去尝试理解为什么要这样设计状态。

例题

例题一：01背包：有 n个物品和一个容量为 W 的背包，每个物品有重量 w_i 和价值 v_i 两种属性，要求选若干物品放入背包使背包中物品的总价值最大且背包中物品的总重量不超过背包的容量。

设 DP 状态 $f_{i.j}$ 为在只能放前 i个物品的情况下，容量为 j 的背包所能达到的最大总价值。

考虑转移。假设当前已经处理好了前 $i-1$ 个物品的所有状态，那么对于第 i 个物品，当其不放入背包时，背包的剩余容量不变，背包中物品的总价值也不变，故这种情况的最大价值为$f_{i-1,j}$；当其放入背包时，背包的剩余容量会减小 w_i，背包中物品的总价值会增大 v_i，故这种情况的最大价值为 $f_{i-1,j-w_i}+v_i$。
由此可以得出状态转移方程：

$$f_{i,j}=max(f_{i-1,j},f_{i-1,j-w_i}+v_i)$$

这里如果直接采用二维数组对状态进行记录，会出现 MLE。可以考虑改用滚动数组的形式来优化。
由于对 f_i 有影响的只有 f_{i-1}，可以去掉第一维，直接用 f_i 来表示处理到当前物品时背包容量为 i 的最大价值，得出以下方程：

$$f_j=max(f_j,f_{j-w_i}+v_i)$$

错误代码：

```
for (int i = 1; i <= n; i++)
  for (int l = 0; l <= W - w[i]; l++)
    f[l + w[i]] = max(f[l] + v[i], f[l + w[i]]);
// 由 f[i][l + w[i]] = max(max(f[i - 1][l + w[i]], f[i - 1][l] + w[i]),
// f[i][l + w[i]]); 简化而来
```

这段代码哪里错了呢？枚举顺序错了。

仔细观察代码可以发现：对于当前处理的物品 i 和当前状态 $f_{i,j}$，在 $w_i\leq j$ 时，$f_{i,j}$ 是会被 $f_{i,j-w_i}$ 所影响的。这就相当于物品 i 可以多次被放入背包，与题意不符。（事实上，这正是完全背包问题的解法）
为了避免这种情况发生，我们可以改变枚举的顺序，从 W 枚举到 w_i，这样就不会出现上述的错误。
正确代码：

```
for (int i = 1; i <= n; i++)
  for (int l = W; l >= w[i]; l--) f[l] = max(f[l], f[l - w[i]] + v[i]);
```

例题二：完全背包
上面错误的代码就是完全背包的做法
例题三：石子合并
在一个环上有 n 个数 $a_1, a_2, \cdots, a_n$ ，进行 $n-1$ 次合并操作，每次操作将相邻的两堆合并成一堆，能获得新的一堆中的石子数量的和的得分,你需要最大化你的得分。
先考虑在一条链上的情况，令 $f(i,j)$ 表示将区间 $[i.j]$ 内石子合并到一起的最大得分，那么状态转移方程为：

$$f(i,j) = max\{f(i,k) + f(k+1,j) + \sum_{t=i}^{j} a_t\}(i \le k < j)$$

怎样进行状态转移?
由于计算 $f(i,j)$ 的值时需要直到所有 $f(i,k)$ 和 $f(k+1,j)$ 的值，这两个区间元素的数量都小于原区间，所以我们用 $len = j - i + 1$ 作为dp的阶段，从小到大枚举 len ，然后枚举左端点的值，从而可以算出右端点，最后枚举 k 。
怎样处理环?
方法一：由于石子围成一个环，我们可以枚举分开的位置，将这个环转化成一个链。
方法二：我们将这条链延长两倍，变成 $2n$ 堆，其中第 i 堆与第 $i+n$ 堆相同。
还能怎样优化?
处理出来前缀和数组，这样每次的转移可以 $O(1)$ 进行。

```
for (len = 2; len <= n; len++)
  for (i = 1; i <= 2 * n - 1 - len; i++) {
    int j = len + i - 1;
    for (k = i; k < j; k++)
      f[i][j] = max(f[i][j], f[i][k] + f[k + 1][j] + sum[j] - sum[i - 1]);//其中sum
是前缀和数组
  }
```

三 . 其它技巧

离散化

通俗地讲就是当有些数据因为本身很大或者类型不支持，自身无法作为数组的下标来方便地处理，而影响最终结果的只有元素之间的相对大小关系时，我们可以将原来的数据按照排名来处理问题，即离散化。
用来离散化的可以是大整数、浮点数、字符串等等。

双指针

双指针维护区间信息的最简单模式就是维护具有一定单调性，新增和删去一个元素都很方便处理的信息，就比如正数的和、正整数的积等等。

例：给定一个已按照 **升序排列** 的长度为 n 的整数数组 a，请你从数组中找出两个数满足相加之和等于目标数 x 。
如果此时我们两个指针分别指在 l,r 上，且 $l<r$, 如果 $a_l+a_r>x$ ，就将 r 减一，如果 $a_l+a_r<x$ ，就将 l 加一。这样 l 不断右移，r 不断左移，最后两者各逼近到一个答案。

```
while(l < r){
        if(a[l] + a[r] > x)
        r--;
        else if(a[l] + a[r] == x){
                printf("%d %d",a[l],a[r]);
                return 0;
        }
        else
        l++;
}
```