搜索算法

剪枝优化

河南省实验中学信息技术组

2025 年 07 月 29 日

知识回顾

- 搜索算法
- 树

优化概述

对于给定的问题，当我们不易建立数学模型或者有数学模型但是解决起来很困难时，可以使用搜索实现。甚至可以说在信息学奥赛中，有一部分用其他算法的题目，也可以使用搜索来解决。但是搜索时间一般时间复杂度高，用来处理这些题目只能得到部分分，为了能得到更高的分数甚至满分，就要求我们学会更多算法或对搜索进行优化。常见的优化方法有以下两种：

- 剪枝优化
- 位运算优化

【引例】圆桌会议

FJ 打算召集 $n(1 \leq n \leq 10)$ 头奶牛召开一个相当重要的圆桌会议，奶牛们感觉到很紧张，他们想把最好的记忆留在脑海中，所以为了美观，他们想在开会时让所有相邻的奶牛的身高差距都不超过 $k(1 \leq k \leq 10^6)$，奶牛的身高用 $h_i(1 \leq h_i \leq 10^6)$ 表示。

请你帮助他们计算，在满足上述条件的情况下，有多少种座位安排方案。在两个不同的安排方案中，至少有一只奶牛的左手边的奶牛都是不同的。

【引例】圆桌会议

【输入格式】

第一行有两个用空格隔开的整数 n, k。

接下来一行 n 个整数，第 i 个整数为第 i 头奶牛的身高 h_i。

【输出格式】

一个整数，表示有多少种满足条件的座位安排方案。

【输入样例】

```
4 10
2 16 6 10
```

【输出样例】

```
2
```

【样例解释】

有 4 头奶牛，身高分别是 $2, 16, 6, 10$，可行的安排方案中任意两头奶牛的身高差距不超过 10。有两种安排方案: $2, 6, 16, 10$ 和 $2, 10, 16, 6$。

【引例】圆桌会议

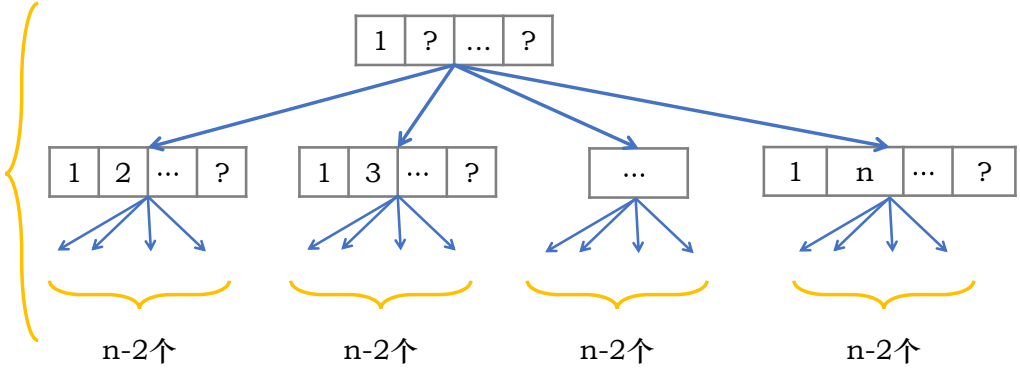
- 将所有奶牛的排列方式一一列举，然后判断奶牛的排列方式是否满足身高差距是否不超过 k(全排列)。
- 要求至少有一只奶牛的左手边的奶牛都是不同的。因为奶牛是环坐的，对于有三只奶牛的情况 1 2 3 和 2 3 1 其实是一样的。所以为了满足要去，我们将第 1 只奶牛固定，只对后面 $n-1$ 只奶牛进行全排列即可。

【引例】圆桌会议

```
int h[15]; //奶牛身高
int pos[15] = {0, 1}; // 各位置上奶牛编号
bool vis[15] = {0, 1};
int ans = 0;

void dfs(int i) // 准备安排第 i 个位置的奶牛
{
    if (i == n + 1) // 所有位置都安排完毕
    {
        if (judge()) ++ans;    // 符合身高差条件
        return;
    }
    for(int j = 2; j <= n; ++j)   // 依次尝试放置每一头奶牛
    {
        if(vis[j]) continue;
        pos[i] = j, vis[j] = true; // 安排奶牛 j 在坐在位置 i
        dfs(i + 1);
        pos[i] = 0, vis[j] = false; // 释放标记
    }
}
// 主函数
dfs(2);    // 从第 2 个位置开始安排
cout << ans;
```

【引例】圆桌会议



从搜索树明显可以发现，结点数目很大，搜索时要遍历每一个结点。仔细观察可以发现：有的中间结点就不满足“身高差距都不超过 k”这一条件，那么从此结点衍生的子结点自然就不合法。如果在搜索的过程中发现这些结点，应立即进行回溯，可以避免遍历部分结点，这就是剪枝。

【引例】圆桌会议

```
1  void dfs(int i) // 准备安排第 i 个位置的奶牛
2  {
3      if (i == n + 1) // 所有位置都安排完毕
4      {
5          if(abs(h[pos[1]] - h[pos[n]]) <= k) ++ans;
6          return;
7      }
8      for(int j = 2; j <= n; ++j)   // 依次尝试放置每一头奶牛
9      {
10         if(vis[j]) continue;
11         if(abs(h[pos[i - 1]] - h[j]) > k) continue;   // 不符合身高差
12         pos[i] = j, vis[j] = true; // 安排奶牛 j 在坐在位置 i
13         dfs(i + 1);
14         pos[i] = 0, vis[j] = false; // 释放标记
15     }
16 }
```

剪枝优化

搜索的进程可以看作是从树根出发，遍历一棵搜索树的过程。所谓剪枝，就是通过某种判断条件，避免一些不必要的遍历过程，形象的说，就是剪去了搜索树中的某些“枝条”。搜索算法中，有以下几类常见的剪枝方法：

- 最优化剪枝
- 可行性剪枝
- 优化搜索顺序
- 排除等效冗余
- 记忆化
- 迭代加深[1]

[1]自行了解。

最优化剪枝

在最优化问题的搜索过程中，如果当前代价已经超过了当前已搜索到的最优解，那么继续向下搜索都不可能得出更优的解。此时，我可以停止当前搜索，进行回溯，这种剪枝方法就叫作最优化剪枝。

例如，当前花费最优解是 30，从花费为 10 的结点可以扩展三个结点。第 1、2 个结点的花费分别为 15 和 20 均小于 30，可以继续向下扩展；第 3 个结点的花费是 35，比当前最优解大，说明继续向下搜索已经没有价值，此时直接回溯。

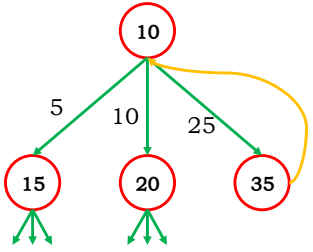


图: 最优化剪枝

可行性剪枝

在搜索过程中，及时对当前状态进行检查，如果发现当前分支绝无可能到达解，则立即执行回溯。这就好比我们在道路上行走时，远远看到前方一个死胡同，就应当立即折返绕路，而不是走到胡同尽头再折返。

优化搜索顺序

在一些搜索问题中，搜索树的各个层次、各个分支之间的顺序不是固定的。

- 不同的搜索顺序会产生不同的搜索树，其规模大小也相差甚远。
- 对于找到答案就可以结束的搜索，搜索顺序有时很重要。

排除等效冗余

在搜索过程中，如果从当前结点沿着某几条不同分支到达的子树是等效的，那么只需要对其中的一条分支进行搜索即可。

迭代加深[2]

- 搜索树随着深度的增加会变得越来越大，那么如果问题的答案在某个深度较小的结点上，而在搜索开始时选择了错误分支，就可能在不包含答案的深层子树上浪费时间。
- 因此，我们可以从小到大逐步限制搜索的深度，如果在当前深度限制下搜不到答案，就把深度限制增加，重新进行一次搜索，这就是迭代加深。

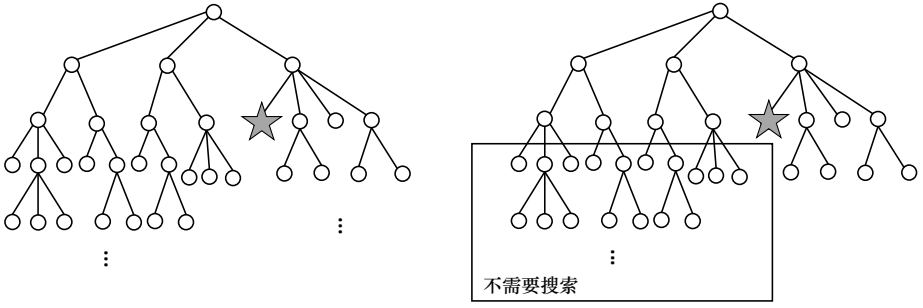


图: 迭代加深

[2]自行了解。

剪枝原则

正确性 进行剪枝时，必须保证不剪掉包含解或最优解的枝条，这是剪枝优化的前提。

高效性 在设计剪枝方法时，要考虑剪枝策略的效率。

- 设计好剪枝方法后，我们对搜索树的每个枝条都要执行一次判断操作。然而，设计剪枝方法利用的是解的“必要条件”，所以，必然有很多不含正解的枝条没有被剪枝，此时剪枝方法本身反而成为消耗时间的因素。为了尽量减少剪枝方法的副作用，除了要改善方法的准确性外，还需要提高方法操作本身的时间效率。
- 然而这就带来了一个矛盾：为了加强优化的效果，必须提高剪枝判断的准确性，因此，常常不得不提高判断操作的复杂度；但是，如果剪枝判断的时间消耗过多，就有可能会减弱或抵消提高判断准确性所能带来的优化效果。很多情况下，能否较好的解决这个矛盾，往往成为搜索算法优化的关键。

【例】小猫爬山

Freda 和 Rainbow 饲养了 n 只小猫，这天，小猫们要去爬山。经历了千辛万苦，小猫们终于爬上了山顶，但是疲倦的它们再也不想徒步走下山了。

Freda 和 Rainbow 只好花钱让它们坐索道下山。索道上的缆车最大承重量为 w，而 n 只小猫的重量分别是 $c_1, c_2, \cdots, c_n$。当然，每辆缆车上的小猫的重量之和不能超过 w。每租用一辆缆车，Freda 和 Rainbow 就要付 1 美元，所以他们想知道，最少需要付多少美元才能把这 n 只小猫都运送下山?

【例】小猫爬山

【输入格式】
第一行两个整数 $n, w(n \leq 18, w \leq 10^8)$，分别表示小猫的数量和缆车的承重量。
接下来一行 n 个整数，表示每只小猫的重量 $c_i(c_i \leq w)$。
【输出格式】
一行一个整数，表示最少需要支付的钱。
【输入样例】

```
5 30
1 2 13 12 29
```

【输出样例】

```
2
```

【样例解释】
第 1 辆缆车装 $1, 29$，第 2 辆缆车装 $2, 13, 12$。

【例】小猫爬山

1. 按顺序为每一只小猫安排缆车。
2. 在轮到第 x 只小猫时，假定当前已经安排的缆车数为 k，每辆缆车当前的负载为 $d[1], d[2], \cdots, d[k]$。
3. 那么第 x 只小猫的安置规则共有 $k+1$ 种：
   - 从第 1 至第 k 辆缆车中顺序尝试，如果可以安排到某辆缆车，就将这只小猫安置在这辆缆车中，然后继续处理下一只小猫；
   - 或者启用第 $k+1$ 辆缆车来安置它。
4. 当第 n 只小猫安置完毕，且当前的 k 更优，则更新最优解。

【例】小猫爬山


```
1  // 准备装载第 x 只猫，当前已经使用 k 辆缆车
2  void dfs(int x, int k)
3  {
4      if(x == n + 1) { ans = min(ans, k); return;}
5      for(int i = 1; i <= k; ++i)
6      {
7          if(c[x] + d[i] > w) continue; // 装不下
8          d[i] += c[x];
9          dfs(x + 1, k);
10         d[i] -= c[x];
11     }
12     // 准备新开一个缆车来装猫
13     d[k + 1] = c[x];
14     dfs(x + 1, k + 1);
15     d[k + 1] = 0;
16 }
```

```
测试点编号：20
使用时间：36.031秒
```

图: 运行结果

剪枝方案 1：优化搜索顺序

- 将小猫按体重从大到小排序后，再进行搜索。
- 如果从大到小排序，那么在搜索过程中，因为已经启用的缆车负载相对较大，则缆车中剩余的空间就较小，后续小猫的选择余地就小，那么当前状态能够扩展出来的结点数就会较少。所以整棵搜索树呈现的形态是：离根近的地方分支较少，这种树如果配合其他剪枝策略时，剪枝力度会相对更大。
- 如果将小猫按照体重从小到大排序后再搜索，效果如何?
- 例如，有 5 只小猫的重量分别为 $1, 2, 3, 4, 5$，缆车承重量为 6，可以画出升序和降序的搜索树进行比较。

```
测试点编号：20
使用时间：34.953秒
```

图: 从大到小排序

```
测试点编号：20
使用时间：170.391秒
```

图: 从小到大排序

剪枝方案 2：最优化剪枝

- 在搜索的任何时刻发现 k 已经大于或等于已经搜索到的答案，那么即可立即回溯。

```
1 // 准备装载第 x 只猫，当前已经使用 k 辆缆车
2 void dfs(int x, int k)
3 {
4     if(k >= ans) return;   // 最优化剪枝
5     if(x == n + 1) { ans = min(ans, k); return;}
6     ...
7 }
```

```
测试点编号：20
使用时间：0.016秒
```

图: 仅使用剪枝方案 2

```
测试点编号：20
使用时间：0.000秒
```

图: 使用剪枝方案 1 和方案 2

优化方案 3：迭代加深[3]

- 题目要求乘坐的缆车数最小，因此可以利用迭代加深，逐次增加缆车数，然后在限定的缆车数上进行搜索。

```
1  // 准备装载第 x 只猫，当前已经使用 k 辆缆车 最多用 deep 辆缆车
2  bool dfs(int x, int k, int deep)
3  {
4      if(k > deep) return false;   // 超出了缆车的使用量
5      if(x == n + 1) return true;  // 搜索成功
6      for(int i = 1; i <= k; ++i) {
7          if(c[x] + d[i] > w) continue; // 装不下
8          d[i] += c[x];
9          if(dfs(x + 1, k, deep)) return true;
10         d[i] -= c[x];
11     }
12     // 准备新开一个缆车来装猫
13     d[k + 1] = c[x];
14     if(dfs(x + 1, k + 1, deep)) return true;
15     d[k + 1] = 0;
16     return false;
17 }
18 for(int deep = sum / w; deep <= n; ++deep) // 迭代加深
19     if(dfs(1, 0, deep)) { printf("%d\n", deep); break; }
```

[3] 自学

【例】最佳调度问题

【题目描述】
已知有 n 个任务，第 i 个任务需要 t_i 的时间完成。现在有 k 个可以并行工作的机器，请你设计一个算法找出完成这 n 个任务的最佳调度，使得完成全部任务的时间最早。

【输入格式】
第一行两个整数 $n, k(n \leq 19, k \leq 6)$，表示任务数和机器数。接下来一行 n 个整数，表示每个任务需要的时间 $t_i(t_i \leq 100)$。

【输出格式】
一行一个整数，表示最少需要支付的钱。

【输入样例】

```
7 3
2 14 4 16 6 5 3
```

【输出样例】

```
17
```

【样例解释】
第 1 台机器完成任务 $2, 4, 6, 5$，第 2 台机器完成任务 $14, 3$，第 3 台机器完成任务 16。

【例】最佳调度问题

- 每个任务可以在编号为 1 到 k 的机器上运行，每台机器运行结束时间为安排在这台机器上的任务消耗的时间之和。

```
int ans = 2000;
// 当前安排第 x 个任务
void dfs(int x)
{
    if(x == n + 1)
    {
        int t = 0; // 所有机器的最晚结束时间
        for(int i = 1; i <= n; ++i) t = max(t, b[i]);
        ans = min(ans, t);
        return ;
    }
    for(int i = 1; i <= k; ++i)
    {
        b[i] += a[x];
        dfs(x + 1);
        b[i] -= a[x];
    }
}
// 主函数
dfs(1)
```

最优化剪枝 1

- 如果当前某一台机器增加任务后运行时间 $\geq ans$，那么可以立即回溯。

```
1 void dfs(int x)
2 {
3     ...
4     for(int i = 1; i <= k; ++i)
5     {
6         if(b[i] + a[x] >= ans) continue;  // 最优化剪枝 1
7         b[i] += a[x];
8     ...
9 }
```

最优化剪枝 2

- 搜索过程中实时记录当前状态下机器运行的最长时间 t，如果 $t \geq ans$ 也要立即回溯。

```
1  // 当前安排第 x 个任务 当前所有机器的最晚结束时间
2  void dfs(int x, int t)
3  {
4      if(t >= ans) return ;// 最优化剪枝 2
5      if(x == n + 1)
6      {
7          ans = min(ans, t);
8          return ;
9      }
10     for(int i = 1; i <= k; ++i)
11     {
12         if(b[i] + a[x] >= ans) continue; // 最优化剪枝 1
13         b[i] += a[x];
14         dfs(x + 1, max(t, b[i]));
15         b[i] -= a[x];
16     }
17 }
```

优化搜索顺序

- 将任务按照时间从大到小排序，这样更容易导致某一台机器的工作时间 $\geq ans$，加快剪枝速度。

```
bool cmp(int a, int b)
{
    return a > b;
}
// 主函数
sort(a + 1, a + n + 1, cmp); // 从大到小排序
dfs(1, 0);
```

【例】漫游小镇

一个正方形的小镇被分成 n^2 个小方格 $(1 \leq n \leq 7)$，Betsy 要从左上角的方格到达左下角的方格，并且经过每个方格恰好一次。编程对于给定的 n，计算出 Betsy 能采用的所有的旅行路线的数目。

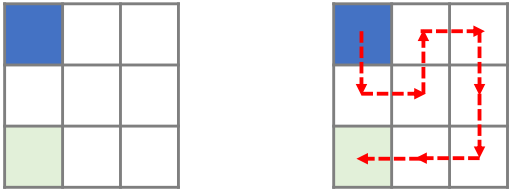


图: N=3 时可行解之一

【例】漫游小镇

从左上角出发，每一步可以从一个格子移动到相邻的没有到过的格子中，遇到“死胡同”则回溯，当移动了 n^2-1 步并达到左下角时，即得到了一条新的路径，继续回溯搜索，直至遍历完所有道路。

【例】漫游小镇

```
1  int g[N][N];    // 地图
2  // 当前在位置 (x,y) 准备要填的数字为 step
3  void dfs(int x, int y, int step)
4  {
5      if(x == n && y == 1)
6      {
7          if(step > n * n) ++ans;
8          return;
9      }
10     for(int i = 1; i <= 4; ++i)
11     {
12         int nx = x + d[i][0], ny = y + d[i][1];
13         if(g[nx][ny]) continue;    // 下一个位置已经走过
14         g[nx][ny] = step;
15         dfs(nx, ny, step + 1);
16         g[nx][ny] = 0;
17     }
18 }
```

【例】漫游小镇

正在编译... 开始运行

点	结果	得分	运行时间	内存使用	返回
1	答案正确	14	0.001 s	13990 KiB	0
2	答案正确	14	0.000 s	13990 KiB	0
3	答案正确	14	0.001 s	13990 KiB	0
4	答案正确	14	0.001 s	13990 KiB	0
5	答案正确	14	0.006 s	13990 KiB	0
6	答案正确	14	0.498 s	13990 KiB	0
7	超过时间限制	0	1.000 s	13990 KiB	-1

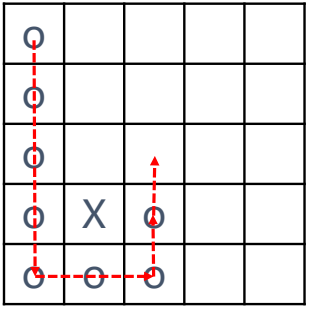
图: 运行结果

剪枝方案 1

对于一条合法的路径，除出发点和目标格子外，每一个中间格子都必然有“一进一出”的过程。所以在搜索过程中，必须保证每个尚未经过的格子都与至少两个尚未经过的格子相邻（除非当时 Betsy 就在它旁边）。

每尝试新走一步时，扫描与当前位置相邻的四个格子，除了选择要走的位置其周围未走过的格子数可以少于 2 个外，其它三个方向上相邻的未走过格子中如果存在其周围未走过的格子数小于 2 者，即可剪枝，提前回溯 (可行性剪枝)。

设一个整型标志数组，分别保存与每个格子相邻的未走过的格子的数目。



剪枝方案 1

```
int l[N][N]; //l[x][y] 表示与格子 (x, y) 相邻的未走过的格子数
// 当前在 (x,y) 即将向方向 k 走
bool cut1(int x, int y, int k)
{
    for(int i = 1; i <= 4; ++i)
    {
        if (i == k) continue;
        int tx = x + d[i][0], ty = y + d[i][1];
        if(tx == n && ty == 1) continue;
        if (g[tx][ty] == 0 && l[tx][ty] < 2)
            return true;
    }
    return false;
}
// 修改 (x,y) 周围四个格子的相邻的未走过的格子数
void change(int x, int y, int val)
{
    for(int i = 1; i <= 4; ++i)
    {
        int nx = x + d[i][0], ny = y + d[i][1];
        l[nx][ny] += val;
    }
}
```

剪枝方案 1

```
// 当前在位置 (x,y) 准备要填的数字为 step
void dfs(int x, int y, int step)
{
    if(x == n && y == 1)
    {
        if(step > n * n) ++ans;
        return;
    }
    for(int i = 1; i <= 4; ++i)
    {
        int nx = x + d[i][0], ny = y + d[i][1];
        if(g[nx][ny]) continue;    // 下一个位置不能走
        if(cut1(x, y, i)) continue; // 剪枝 1
        change(nx, ny, -1); // (nx,ny) 周围四个格子的相邻的未走过的格子数减 1
        g[nx][ny] = step;
        dfs(nx, ny, step + 1);
        g[nx][ny] = 0;
        change(nx, ny, 1); // (nx,ny) 周围四个格子的相邻的未走过的格子数加 1
    }
}
```

剪枝方案 1

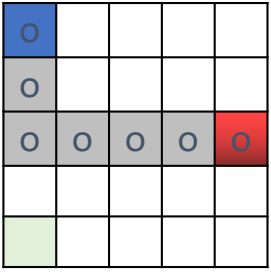
正在编译... 开始运行

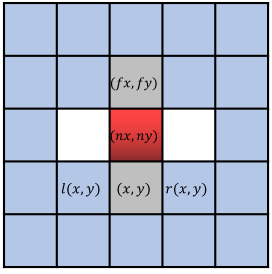
点	结果	得分	运行时间	内存使用	返回
1	答案正确	14	0.000 s	13990 KB	0
2	答案正确	14	0.000 s	13990 KB	0
3	答案正确	14	0.000 s	13990 KB	0
4	答案正确	14	0.001 s	13990 KB	0
5	答案正确	14	0.001 s	13990 KB	0
6	答案正确	14	0.019 s	13990 KB	0
7	答案正确	14	0.916 s	13990 KB	0

图: 运行结果——剪枝方案 1

剪枝方案 2

每走一步，都不能造成孤立的区域存在，虽然孤立区域中的每一个格子也可能都有至少两个相邻的未走过的格子，但它们作为一个整体，Betsy 已经不能达到。如果出现这种情况，应当及时剪枝 (可行性剪枝)。

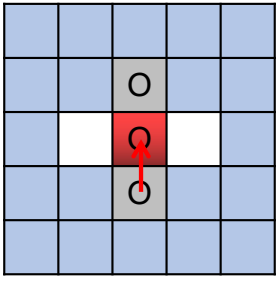




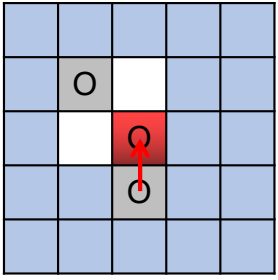
设当前位置为 (x, y), 下一步选择走的位置为 (nx, ny), 到达新位置后沿着相同方向再走一步的位置为 (fx, fy),$l(x, y)$ 表示 (x, y) 左边的格子,$r(x, y)$ 表示 (x, y) 右边的格子。

剪枝方案 2

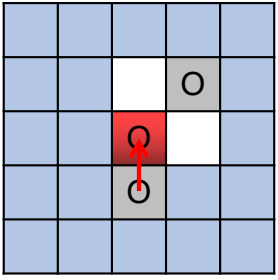
以下三种情况均会造成孤立区域，横向亦然：



图：(fx, fy) 已走过并且 $l(nx, ny)$ 和 $r(nx, ny)$ 均未走过



图：(fx, fy) 未走过并且 $l(nx, ny)$ 未走过 $l(fx, fy)$ 已走过



图：(fx, fy) 未走过并且 $r(nx, ny)$ 未走过 $r(fx, fy)$ 已走过

剪枝方案 2

每尝新走一步之前，以当前方向为基准，对前面提到的三种情况分别加以判断，如果其中任意一种出现，即说明这样走便会造成孤立区域产生，即可剪枝，提前回溯。

```
1  // 当前在 (x,y) 即将向方向 k 走
2  bool cut2(int x, int y, int k)
3  {
4      int nx = x + d[k][0], ny = y + d[k][1];
5      int fx = nx + d[k][0], fy = ny + d[k][1];
6      int dl, dr; // 当前方向的两侧方向
7      if(k == 1 || k == 4) dl = 2, dr = 3;
8      else if(k == 2 || k == 3) dl = 1, dr = 4;
9      if(g[fx][fy] != 0 && g[nx + d[dl][0]][ny + d[dl][1]] == 0   // 情况 1
10         && g[nx + d[dr][0]][ny + d[dr][1]] == 0) return true;
11     if(g[fx][fy] == 0 && g[fx + d[dl][0]][fy + d[dl][1]] > 0    // 情况 2
12         && g[nx + d[dl][0]][ny + d[dl][1]] == 0) return true;
13     if(g[fx][fy] == 0 && g[fx + d[dr][0]][fy + d[dr][1]] > 0    // 情况 3
14         && g[nx + d[dr][0]][ny + d[dr][1]] == 0) return true;
15     return false;
16 }
```

剪枝方案 2

```
// 当前在位置 (x,y) 准备要填的数字为 step
void dfs(int x, int y, int step)
{
    if(x == n && y == 1)
    {
        if(step > n * n) ++ans;
        return;
    }
    for(int i = 1; i <= 4; ++i)
    {
        int nx = x + d[i][0], ny = y + d[i][1];
        if(g[nx][ny]) continue;    // 下一个位置不能走
        if(cut2(x, y, i)) continue; // 剪枝 2
        g[nx][ny] = step;
        dfs(nx, ny, step + 1);
        g[nx][ny] = 0;
    }
}
```

剪枝方案 2

正在编译... 开始运行

点	结果	得分	运行时间	内存使用	返回
1	答案正确	14	0.000 s	13990 KiB	0
2	答案正确	14	0.001 s	13990 KiB	0
3	答案正确	14	0.000 s	13990 KiB	0
4	答案正确	14	0.001 s	13990 KiB	0
5	答案正确	14	0.001 s	13990 KiB	0
6	答案正确	14	0.012 s	13990 KiB	0
7	答案正确	14	0.452 s	13990 KiB	0

图: 运行结果——剪枝方案 2

剪枝方案 1 和 2 合并

```
// 当前在位置 (x,y) 准备要填的数字为 step
void dfs(int x, int y, int step)
{
    if (x == n && y == 1)
    {
        if(step > n * n) ++ans;
        return;
    }
    for (int i = 1; i <= 4; ++i)
    {
        int nx = x + d[i][0], ny = y + d[i][1];
        if(g[nx][ny]) continue;    // 下一个位置不能走
        if (cut1(x, y, i)) continue; // 剪枝 1
        if (cut2(x, y, i)) continue; // 剪枝 2
        change(nx, ny, -1); // (nx,ny) 周围四个格子的相邻的未走过的格子数减 1
        g[nx][ny] = step;
        dfs(nx, ny, step + 1);
        g[nx][ny] = 0;
        change(nx, ny, 1); // (nx,ny) 周围四个格子的相邻的未走过的格子数加 1
    }
}
```

剪枝方案 1 和 2

正在编译... 开始运行

点	结果	得分	运行时间	内存使用	返回
1	答案正确	14	0.000 s	13990 KiB	0
2	答案正确	14	0.000 s	13990 KiB	0
3	答案正确	14	0.000 s	13990 KiB	0
4	答案正确	14	0.001 s	13990 KiB	0
5	答案正确	14	0.001 s	13990 KiB	0
6	答案正确	14	0.009 s	13990 KiB	0
7	答案正确	14	0.298 s	13990 KiB	0

图: 运行结果——剪枝方案 1 和 2

【例】木棍拼接

【题目描述】
乔治拿来一组等长的木棒，将它们随机地砍断，使得每一节木棍的长度都不超过 50 个长度单位。然后他又想把这些木棍恢复到为裁截前的状态，但忘记了初始时有多少木棒以及木棒的初始长度。请你设计一个程序，帮助乔治计算木棒的可能最小长度。

【输入格式】
第一行是一个不超过 64 的整数，表示砍断之后共有多少节木棍。
第二行是截断以后，所得到的各节木棍的长度。

【输出格式】
输出原始木棒的可能最小长度。

【输入样例】

```
9
5 2 1 5 2 1 5 2 1
```

【输出样例】

```
6
```

【例】木棍拼接

- 从小到大枚举木棒的长度 len，len 是所有木棍长度总和 sum 的约数，原始木棒的根数 $tot = sum/len$。
- 对于枚举的每个长度 len，我们可以使用搜索算法得出每根原始木棒由哪些木棍拼成。
- 如果可以用木棍拼成 tot 个长度为 len 的木棍，则 len 就是答案。

【例】木棍拼接

```
int len, tot;   // 木棒长度 木棒总根数
int a[N];       // 木棍的长度
int vis[N];     // 木棍是否已经被使用
// 正在拼第 cnt 根木棒
// 第 cnt 根木棒已经拼了长度 clen
bool dfs(int cnt, int clen)
{
    if(cnt > tot) return true;      // 所有木棒都拼完
    if(clen == len) return dfs(cnt + 1, 0); // 继续拼下一个木棒
    for (int i = 1; i <= n; ++i)
    {
        if(vis[i] || clen + a[i] > len) continue;
        vis[i] = true;
        if (dfs(cnt, clen + a[i])) return true;
        vis[i] = false;
    }
    return false;
}
```

【例】木棍拼接

```
1  int sum = 0;      // 所有木棍的长度总和
2  int min_len = 0; // 木棒的最小可能长度
3  for(int i = 1; i <= n; ++i)
4  {
5      sum += a[i];
6      max_len = max(max_len, a[i]);
7  }
8
9  for(len = max_len; len <= sum; ++len) // 枚举木棒的长度
10 {
11     if(sum % len) continue;
12     tot = sum / len;  // 木棍总根数
13     if(dfs(1, 0)) { cout << len; break; }
14 }
```

剪枝方案 1

- 考虑拼接木棒时，相比于短木棍，长木棍更容易导致拼接失败。
- 把木棍长度从大到小进行排序，优先尝试较长的木棍 (优化搜索顺序)。

```
1 bool cmp(const int &a, const int &b)
2 {
3     return a > b;
4 }
5
6 sort(a + 1, a + n + 1, cmp);          // 剪枝 1
```

剪枝方案 2

- 考虑拼接木棒只要求成功与否，故而限制先后加入一根原始木棒的木棍长度是递减的 (排除等效冗余)。
- 因为先拼上一个长度为 x 的木棍，再拼上一根长度为 $y(x > y)$ 的木棍，与先拼上 y 再拼上 x 是等效的，只需要搜索其中的一种。

```
1  // 正在拼第 cnt 根木棒
2  // 第 cnt 根木棒已经拼了长度 clen
3  // last 为拼当前木棒上一次用的木棍下标
4  bool dfs(int cnt, int clen, int last)
5  {
6      if(cnt > tot) return true;      // 所有木棒都拼完
7      if(clen == len) return dfs(cnt + 1, 0, 1); // 继续拼下一个木棒
8      for (int i = last; i <= n; ++i)   // 剪枝 2: 同一根木棒中的木棍长度递减
9      {
10         if(vis[i] || clen + a[i] > len) continue;
11         vis[i] = true;
12         if(dfs(cnt, clen + a[i], i)) return true;
13         vis[i] = false;
14     }
15     return false;
16 }
```

剪枝方案 3

- 对于当前木棒，记录最近一次尝试拼接的木棍长度。如果搜索失败，则不再尝试向该木棒中拼接其他相同长度的木棍 (因为必然会失败，排除等效冗余)。

```
// 其他代码
    int fail = 0;   // 剪枝 3: 相同木棒长度失败 就不再搜索
    for(int i = last; i <= n; ++i) // 剪枝 2: 同一根木棒中的木棍长度递减
    {
        if(vis[i] || clen + a[i] > len) continue;
        if(fail == a[i]) continue; // 剪枝 3: 相同木棒长度失败 就不再搜索
        vis[i] = true;
        if (dfs(cnt, clen + a[i], i)) return true;
        fail = a[i];            // 剪枝 3: 相同木棒长度失败 就不再搜索
        vis[i] = false;
// 其他代码
```

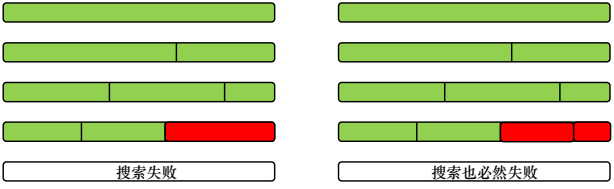
剪枝方案 4

- 如果在当前原始木棒中尝试拼入的第一根木棍搜索失败，那么直接回溯 (排除等效冗余)。
- 因为在拼入这根木棍前，面对的原始木棒都是“空”的，这些木棒是等效的。木棍在拼当前木棒失败，则在其他木棒中一样会失败。


```
// 其他代码
            vis[i] = false;
            // 剪枝 4: 空木棒放入木棍失败 也不再搜索
            if(clen == 0) return false;
        }
// 其他代码
```

剪枝方案 5

- 如果在当前原始木棒拼入最后一个木棍后，木棒被拼接完整，而后续搜索失败，则直接回溯 (排除等效冗余)。
- 我们可以借助贪心来解释，“再用 1 根木棍拼完当前原始木棒”必然比“再用若干根木棍拼接完当前原始木棒”更好。




```
// 其他代码
            vis[i] = false;
            // 剪枝 4: 当前木棒的第一根木棍拼接失败 直接回溯
            // 剪枝 5: 当前木棒拼接完成后失败 直接回溯
            if(clen == 0 || clen + a[i] == len) return false;
        }
// 其他代码
```

【例】生日蛋糕

【题目描述】

7 月 17 日是 Mr.W 的生日，ACM-THU 为此要制作一个体积为 $N\pi$ 的 M 层生日蛋糕，每层都是一个圆柱体。

设从下往上数第 i（$1 \le i \le M$）层蛋糕是半径为 R_i，高度为 H_i 的圆柱。当 $i \le M$ 时，要求 $R_i > R_{i+1}$ 且 $H_i > H_{i+1}$。

由于要在蛋糕上抹奶油，为尽可能节约经费，我们希望蛋糕外表面（最下一层的下底面除外）的面积 Q 最小。

请编程对给出的 N 和 M，找出蛋糕的制作方案（适当的 R_i 和 H_i 的值），使 $S = \frac{Q}{\pi}$ 最小。

（除 Q 外，以上所有数据皆为正整数）

【例】生日蛋糕

【输入格式】
第一行为一个整数 N（$N \leq 2 \times 10^4$），表示待制作的蛋糕的体积为 $N\pi$。
第二行为 M（$M \leq 25$），表示蛋糕的层数为 M。
【输出格式】
输出一个整数 S，若无解，输出 0。

【输入样例】

```
100
2
```

【输出样例】

```
68
```

【样例解释】蛋糕第一层高度和半径分别为 $6, 4$，第二层高度和半径分别为 $1, 2$，表面积为 $4 \times 4 \times \pi + 2 \times 4 \times 6 \times \pi + 2 \times 2 \times 1 \times \pi = 68\pi$。

【例】生日蛋糕

- 从下往上枚举每层的半径和高度，验证体积是否为 $N\pi$，在所有的方案中取最小值即可。
- 搜索时需要记录的状态有：当前正在枚举的蛋糕的层数 d，当前外表面积 s，当前体积 v，已经枚举完成的若干层蛋糕的高度 h 和半径 r。
- 表面积计算：整个蛋糕的“上表面”面积之和为第 M 层蛋糕的圆面积，当第 M 层蛋糕确定后，后续只需要计算侧面积即可。
- 在所有可行的蛋糕方案中选择表面积最小的即可。

【例】生日蛋糕

```
// 准备搜索第 d 层蛋糕 当前表面积为 s 体积为 v
void dfs(int d, int s, int v)
{
    if(d == 0)
    {
        if(v == n) ans = min(ans, s);
        return;
    }
    for(int R = 1; R < r[d + 1]; ++R) // 枚举半径
        for(int H = 1; H < h[d + 1]; ++H)    // 枚举高度
        {
            if(v + R * R * H > n) continue; // 体积不能超过 n
            h[d] = H, r[d] = R;
            if(d == m) dfs(d - 1, s + 2 * R * H + R * R, v + R * R * H);
            else dfs(d - 1, s + 2 * R * H, v + R * R * H);
            h[d] = 0, r[d] = 0;
        }
}
// 主函数
h[m + 1] = r[m + 1] = n;
dfs(m, 0, 0);
```

【例】生日蛋糕

- 最优化剪枝 1：当前表面积已经大于历史上的最有答案，直接回溯。

```
1 // 准备搜索第 d 层蛋糕 当前表面积为 s 体积为 v
2 void dfs(int d, int s, int v)
3 {
4     if(s > ans) return;     // 最优化剪枝 1
```

【例】生日蛋糕

- 进一步考虑，每一层的高度均为整数，那么当还剩上面 d 层时，蛋糕每层最低要求应当为：第 $1 \sim d$ 层的半径取 $1, 2, \cdots, d$，高度也取 $1, 2, \cdots, d$；在上述情况下蛋糕有最小体积和表面积：
  - 可行性剪枝 1：当前体积 v 加上 $1 \sim d$ 层的最小体积大于 n，直接回溯。
  - 最优化剪枝 1 改进：当前表面积 s 加上 $1 \sim d$ 层的最小表面积大于当前最优解，直接回溯。

```
1  // 准备搜索第 d 层蛋糕 当前表面积为 s 体积为 v
2  void dfs(int d, int s, int v)
3  {
4      if(v + minv[d] > n) return ; // 可行性剪枝 1
5      if(s + mins[d] > ans) return;    // 最优化剪枝 1 优化
6
7  // 主函数
8  for(int i = 1; i <= m; ++i)
9  {
10     minv[i] = minv[i - 1] + i * i * i;
11     mins[i] = mins[i - 1] + 2 * i* i;
12 }
13 mins[m] += m * m;
```

【例】生日蛋糕

- 上下界剪枝：在第 d 层时，枚举的高度不应当为 $1 \sim n$，半径不应当从 1 开始。
  - 首先，枚举 R 的范围为 $[d, min(r[d+1]-1, \lfloor\sqrt{n-v}\rfloor)]$；
  - 其次，枚举 H 的范围为 $[d, min(h[d+1]-1, \lfloor\frac{n-v}{R^2}\rfloor$。
- 优化搜索顺序：不应当从小到大枚举半径和高度，修改为从大到小枚举。

```
1    for(int R = 1; R < r[d + 1]; ++R) // 枚举半径
2        for(int H = 1; H < h[d + 1]; ++H)    // 枚举高度
3
4    for(int R = min(r[d + 1] - 1, (int)sqrt(n - v)); R >= d; --R) // 枚举半径
5        for(int H = min(h[d + 1] - 1, (int)((double)(n - v) / R / R)); H >= d; --H)
```

【例】生日蛋糕

- 第 $1 \sim d$ 层的体积可以表示为 $n - v = \sum_{i=1}^{d} h[i]r[i]^2$，第 $1 \sim d$ 层的表面积可以表示为 $2\sum_{i=1}^{d} h[k]r[k]$。
- 当枚举半径 R 时

$$2\sum_{i=1}^{d} h[i]r[i] = \frac{2}{R}\sum_{i=1}^{d} h[i]r[i]R \geq \frac{2}{R}\sum_{i=1}^{d} h[i]r[i]^2 \geq \frac{2(n-v)}{R}$$

所以当 $\frac{2(n-v)}{R} + s$ 大于当前最优解，直接回溯。

```
1    for(int R = min(r[d + 1] - 1, (int)sqrt(n - v)); R >= d; --R) // 枚举半径
2        for(int H = min(h[d + 1] - 1, (int)((double)(n - v) / R / R)); H >= d; --H)
3        {
4            if(v + R * R * H > n) continue; // 体积不能超过 n
5            if(2 * (n - v) / R + s > ans) continue;    // 最优化剪枝 2
6            ...
7        }
```

小结

剪枝方法无论多么巧妙，都不能从本质上降低搜索算法的时间复杂度，这是不争的事实。因此，我们在动手设计一个搜索算法之前，不妨先考虑一下是否存在着更为有效的方法。

对于剪枝的实现，还有一个编程复杂度的问题。在对一个搜索算法使用了很多优化技巧后，虽然可能使程序的时间效率得到一定的提高，但却往往要消耗大量的编程时间，在实战中也是一个重要因素。

剪枝优化练习

- 漫游小镇 (COGS 913)
- 小猫爬山 (COGS 1197)
- 最优调度问题【SYOI】(COGS 2846)
- 木棍拼接 (COGS 1196)
- 生日蛋糕【NOI 1999】(COGS 67)
- 素数方阵 (COGS 890)
- 虫食算【NOIP 2004】(COGS 69)
- 玛雅游戏【NOIP 2011】(COGS 622)
- 回文【CSP 2021】(COGS 3621)
- 荆轲刺秦王【NOI Online 2020 2nd PJ】(COGS 3423)
- 加成序列 (COGS 3471)
- 天气预报 (COGS 2515)

记忆化搜索

在搜索过程中记录每个状态的搜索结果，那么在重复遍历某一状态时直接返回存储的结果。

【引例】斐波那契数列

定义斐波那契数列第 n 项为 $fib(n)$，则有如下公式：

$$fib(n) = \begin{cases} 1, & n = 1 \text{ or } n = 2 \\ fib(n-1) + fib(n-2), & n \geq 3 \end{cases}$$

容易，我们可以利用递归写出如下程序：

```
1 long long fib(int n)
2 {
3     if(n == 1 || n == 2) return 1;
4     else return fib(n - 1) + fib(n - 2);
5 }
```

然而，实际运行过程中，例如运行`fib(90)`会发现运行时间过长。是什么原因导致运行时间长？如何缩短运行时间呢？

【引例】斐波那契数列

- 原因：重复计算。
- 解决方案：将计算的结果存储起来，如果某个结果已经计算过，直接返回而不是递归计算。
- 改进后的程序如下：

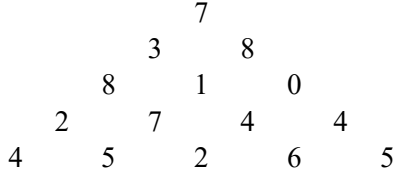
```
long long f[110] = {0, 1, 1};
long long fib(int n)
{
    if(f[n]) return f[n];
    else return f[n] = fib(n - 1) + fib(n - 2);
}
```

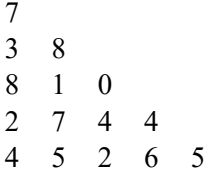
- 那么，对于搜索算法，我们能否借鉴这种思路呢？答案是肯定的。
- 在搜索算法中，如果出现重复状态搜索，那么就可以考虑将状态运行的结果存储起来。在后续的搜索中，如果遇到同一状态，则可以直接返回。

【例】数字三角形

如下所示为一个数字三角形。请编一个程序计算从顶层到底层某处的一条路径，使该路径所经过的数字总和最大。只要求输出总和。

- 每一步可沿左斜线向下或右斜线向下走；
- 三角形行数小于等于 100；
- 三角形中的数字为非负整数。





数字三角形 (DFS)

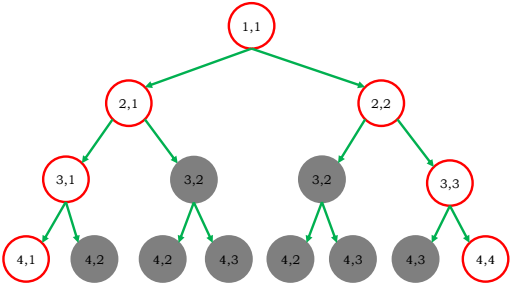
定义搜索函数 `int dfs(int i, int j)`，其中 i, j 表示从 (i, j) 走到最后一行的最大权值和，则

- 当 $i = n$ 时，已经搜索至最底层，直接返回 $a[i][j]$ 即可;
- 否则，向下一行两个位置行走，递归执行 `dfs(i + 1, j)` 和 `dfs(i + 1, j + 1)`，然后返回二者的最大值与 $a[i][j]$ 的和。

```
1 int dfs(int i, int j)
2 {
3     if(i == n) return a[i][j];
4     int ta = dfs(i + 1, j);
5     int tb = dfs(i + 1, j + 1);
6     return a[i][j] + max(ta, tb);
7 }
8 // 主函数
9 cout << dfs(1, 1);
```

数字三角形 (DFS)

- 上述算法的时间复杂度为 $O(2^N)$，造成如此高时间复杂度的原因是什么？重复状态搜索。
- 如下图，`dfs(3,2)` 被执行了两次，而 `dfs(3,2)` 两次执行的结果完全一致。



数字三角形 (记忆化搜索)

- 因为存在重复的计算，于是用二维数组 `f[i][j]=dfs(i,j)` 记录从 (i, j) 出发到最底层的最大权值和，初始时全为 -1 表示未被计算过。
- 在执行行 `dfs(i, j)` 时，先查询 $f[i][j]$ 是否已经计算过，如果计算过，直接返回之，否则执行 `dfs(i, j)` 得出值并存储在 $f[i][j]$ 中。

```
1 int dfs(int i, int j)
2 {
3     if(f[i][j] != -1) return f[i][j];   // f[i][j] 已经计算过
4     if(i == n) return f[i][j] = a[i][j];
5     int ta = dfs(i + 1, j);
6     int tb = dfs(i + 1, j + 1);
7     return f[i][j] = a[i][j] + max(ta, tb);
8 }
```

- 每个状态至多被计算一次，所以时间复杂度：$\mathrm{O}(N^2)$

【例】01 背包

给定 n 个物品，其中第 i 个物品的重量为 w_i，价值为 v_i。有一个承重量为 m 的背包，要求选择一些物品放入背包，使得物品总重量不超过 m 的情况下，物品的价值总和最大是多少?

例如，有一个容量为 10 的背包，有 4 个物品，他们的重量如下表:

编号	1	2	3	4	5
$w[i]$	2	2	6	5	4
$v[i]$	6	3	5	4	6

那么，我们选择装入物品 1、2、5，装入的总价值为 15，共装入重量为 8。

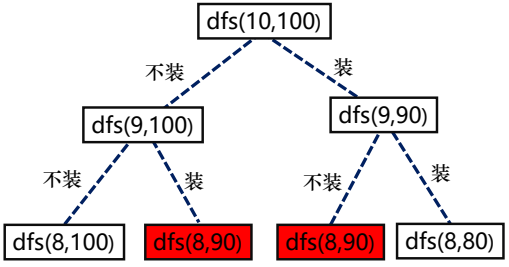
01 背包

- 我们可以按照每一种商品选或不选来进行搜索。
- 物品编号从小到大搜索和从大到小搜索是相同的，为了方便后续表述我们从大到小搜索。

```
// 在容量为 c 的背包中装入前 i 件物品 能产生的最大价值
int dfs(int i, int c)
{
    if(i == 0) return 0;     // 装入前 0 件物品 返回 0
    if(w[i] > c) return dfs(i - 1, c);// 如果背包容量不足 只能选择不装第 i 件物品
    else              // 背包容量充足
    {
        int ta = dfs(i - 1, c);  // 不装第 i 件物品
        int tb = dfs(i - 1, c - w[i]) + v[i];   // 装第 i 件物品
        return max(ta, tb);
    }
}
// 主函数中执行
// 在容量为 m 的背包中装入前 n 件物品
cout << dfs(n, m);
```

01 背包

- 上述搜索算法的时间复杂度为 $O(2^N)$，导致这种问题的原因同样是因为重复状态搜索。
- 例如，有 10 件物品，它们的重量分别为 $\{w_1, \cdots, w_8, 10, 10\}$，背包的容量为 100，那么搜索树如下：



- 定义二维数组 $f[][]$，其中 `f[i][c]=dfs(i,c)` 表示在容量为 c 的背包中装入前 i 件物品能产生的最大价值。

01 背包

```
1  // 在容量为 c 的背包中装入前 i 件物品 能产生的最大价值
2  int dfs(int i, int c)
3  {
4      if(i == 0) return f[i][c] = 0;
5      if(f[i][c] != -1) return f[i][c];   // 如果以前搜索过 直接返回
6      if(w[i] > c) return f[i][c] = dfs(i - 1,c);// 如果背包容量不足 只能选择不装第 i 件物品
7      else             // 背包容量充足
8      {
9          int ta = dfs(i - 1,c);  // 不装第 i 件物品
10         int tb = dfs(i - 1, c - w[i]) + v[i];   // 装第 i 件物品
11         return f[i][c] = max(ta, tb);
12     }
13 }
14 // 主函数中执行
15 // 在容量为 m 的背包中装入前 n 件物品
16 cout << dfs(n, m);
```

- 每个状态至多被计算一次，所以时间复杂度：O(NM)。

小结

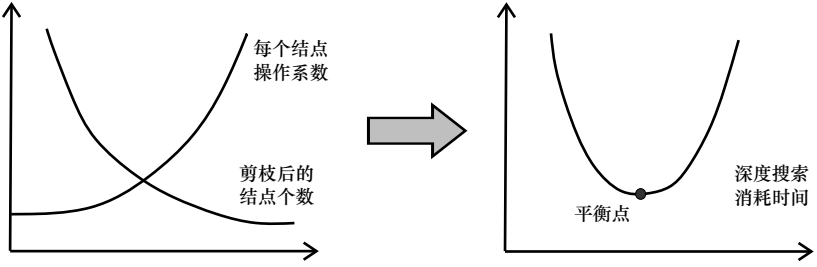
- 使用记忆化搜索要求搜索的状态会出现重复，而且同一状态的值是一样的。
- 记忆化搜索的搜索函数的参数个数和记录数组的维数相同，搜索函数返回值存储在数组中。
- 与普通剪枝优化方法不同的是，记忆化搜索可能可以显著的降低算法的时间复杂度。

练习

- 数塔问题【IOI1994】(COGS 77)
- 采药【NOIP2005】(COGS 68)
- 开心的金明【NOIP2006】(COGS 71)
- 数的划分【NOIP 2001】(COGS 93)
- 天气预报 (COGS 2515)

总结

搜索消耗时间 ≈ 每个结点操作系数 × 结点个数



故，在优化时要综合考虑结点个数和结点操作系数，在可能的情况下：

- 减少结点个数——剪枝优化
- 减少每个结点的操作系数——位运算优化