搜索

搜索是已知初始状态和目标状态，对问题的中间状态进行枚举和遍历的一种解题算法，所以说搜索是**一种变化的和复杂的枚举**

枚举过程一般较简单，且变化较少，如：

· 需枚举的**变量较少**

· 循环的**层数固定**

· 每一个变量的**枚举范围相对固**定

· 可循的**数学规律较少**，较简单，且不能回撤

搜索从本质上来说是枚举，但搜索更灵活，更加能够体现智能逻辑的魅力。

基于枚举思想的搜索较其"溯源"有许多不可替代的优势。如：

· 循环的层数是可变化的、动态的，可以把抽象的、复杂的数学逻辑规律统一起来

· 简便地对枚举遍历过程进行管理
· 同时也可以利用这些规律减少枚举量

从枚举开始

基本思想：

枚举**所有可能的解**，用**条件**进行检验

算法要点：

枚举对象的**分量** $(a_1, a_2, \cdots, a_n)$

求解对象的**取值范**围 $L_i \leq a_i \leq R_i$

枚举对象的选择，决定**枚举的策**略，必须保证能**覆盖最优方**案。

约束条件

代码实现框架：

```
for (int a1=l1;a1<=r1;a1++)
    for (int a2=l2;a2<=r2;a2++)
        ……
            for(int an=ln;an<=rn;an++)
            {
                if (a1，a2,…,an满足约束条件)
                {
                    输出合法的方案;
                }
            }
```

例题——火柴棒等式

题目描述

给你 n 根火柴棍，你可以拼出多少个形如 $A+B=C$ 的等式？等式中的 A、B、C 是用火柴棍拼出的整数（若该数非零，则最高位不能是 0）。用火柴棍拼数字 $0\sim 9$ 的拼法如图所示：

注意：

1. 加号与等号各自需要两根火柴棍；
2. 如果 $A\neq B$，则 $A+B=C$ 与 $B+A=C$ 视为不同的等式（$A,B,C\geq 0$）；
3. n 根火柴棍必须全部用上。

输入格式

一个整数 $n(1\leq n\leq 24)$。

输出格式

一个整数，能拼成的不同等式的数目。

样例 #1

样例输入 #1

```
14
```

样例输出 #1

```
2
```

样例 #2

样例输入 #2

```
18
```

样例输出 #2

```
9
```

提示

【输入输出样例 1 解释】

2 个等式为 $0+1=1$ 和 $1+0=1$。

【输入输出样例 2 解释】

9 个等式为

$0+4=4$、$0+11=11$、$1+10=11$、$2+2=4$、$2+7=9$、$4+0=4$、$7+2=9$、$10+1=11$、$11+0=11$。

题目分析

首先，预处理每个数字 $0\sim9$ 需要用几根火柴棒，存储在数组 f 中。然后，穷举 a 和 b，算出它们的和 c，再判断下列约束条件是否成立：

$$f(a)+f(b)+f(c)==n-4$$

现在的问题是：a 和 b 的范围有多大？

可以发现尽量用数字 1 拼成的数比较大，分析可知最多不会超过 1111。程序实现时，分别用三个循环语句预处理好所有两位数、三位数、四位数构成所需要的火柴棒数量。

参考代码1：

```
#include<bits/stdc++.h>
using namespace std;
int f[10000];
int main()
{
    f[0]=6;f[1]=2;f[2]=5;f[3]=5;f[4]=4;
    f[5]=5;f[6]=6;f[7]=3;f[8]=7;f[9]=6;
    for(int i=10;i<=99;i++)
        f[i]=f[i/10]+f[i%10];
    for(int i=100;i<=999;i++)
        f[i]=f[i/100]+f[i-(i/100)*100/10]+f[i%10];
    for(int i=1000;i<=9999;i++)
        f[i]=f[i/1000]+f[(i-(i/1000)*1000)/100]+f[(i-
(i/100)*100)/10]+f[i%10];
    int n,total=0;
    scanf("%d",&n);
    for(int a=0;a<1111;a++)
        for(int b=0;b<=1111;b++)
            {
                int c=a+b;
                if(f[a]+f[b]+f[c]==n-4)total++;
            }
    printf("%d\n",total);
    return 0;
 }
```

参考代码2：

```
#include<cstdio>
#include<cstring>
#include<iostream>
#include<algorithm>
using namespace std;
int n,ans=0;
int s[10]={6,2,5,5,4,5,6,3,7,6};

int shuliang(int x)
{
    if(x==0) return 6;
    int sum=0;
    while(x>0)
    {
        sum+=s[x%10];
        x/=10;
    }
    return sum;
 }

```

```
int main()
{
   scanf("%d",&n);
   for(int i=0;i<=1000;i++)
    for(int j=0;j<=1000;j++)
     {
       int a=shuliang(i)+shuliang(j);
       int b=shuliang(i+j);
       if(a+b+4==n)
       ans++;
      }
   printf("%d",ans);
   return 0;
}

```

例题——奶牛碑文

题目描述

小伟暑假期间到大草原旅游，在一块石头上发现了一些有趣的碑文。碑文似乎是一个神秘古老的语言，只包括三个大写字母 C, O, W 。尽管小伟看不懂，但是令他高兴的是，C, O, W 的顺序形式构成了一句他最喜欢的奶牛单词 COW。现在，他想知道有多少次 COW 出现在文本中。

注意：即使 COW 内穿插了其他字符，但只要 COW 字符出现在正确的顺序，小伟也不介意。甚至，他也不介意出现不同的 COW 共享一些字母。例如，$CWOW$ 出现了 1 次 COW，$CCOW$ 算出现了2 次 COW，$CCOOWW$算出现了 8 次 COW。

输入格式

第 1 行为1个整数 N。

第2 行为 N个字符的字符串，每个字符是一个C、O 或 W。

输出格式

输出 COW作为输入字符串的字串出现的次数（不一定是连续的）。

提示：答案数据可能比较大，建议用 64 位整数（$long\ long$）。

输入样例

6

$COOWWW$

输出样例

6

数据规模

对于 50% 的数据满足：$N \leq 60$。

对于 100% 的数据满足：$N \leq 10^5$。

题目分析

因为只有 3 个字母，所以可以穷举字符串中的每一个 O，假设位置 i，然后分别计算其左边 C 的个数 $l[i]$ 和右边 W 的个数 $r[i]$，再利用乘法原理进行计数 $l[i] \times r[i]$，每次把答案累加到 ans 中。

参考代码

```cpp
#include<bits/stdc++.h>
using namespace std;
int t,sum,a[100010];
long long ans;
string st;
int main()
{
    cin>>t;
    getchar();
    t--;
    getline(cin,st);
    sum=0;
    for(int i=t;i>=0;i--)
    {
        if(st[i]=='w')++sum;
        if(st[i]=='o')a[i]=sum;
    }
    ans=0;sum=0;
    for(int i=t;i>=0;i--)
    {
        if(a[i])sum+=a[i];
        if(st[i]=='c')ans+=sum;
    }
    cout<<ans<<endl;
    return 0;
}
```

枚举的优化

枚举算法的特点是算法设计、实现都相对简单，但时间复杂度和空间复杂度往往较大。因此，用枚举算法解决问题时，往往需要尽量**优化算法**，从而**减少穷举的次数**，提高穷举的效率。

枚举算法优化的思路主要有结合约束条件，通过**数学推导**，减少**穷举的范围和数量**；通过**预处理**（如部分和、是否素数等），以**空间换时间**，避免在穷举过程中重复计算或判断等。

空间换时间

一般来说，计算机提供了相当大的内存，可以利用这些内存空间，使用一些技巧提升处理数据的效率。

[CSP-J2020] 直播获奖

题目描述

NOI2130 即将举行。为了增加观赏性，CCF 决定逐一评出每个选手的成绩，并直播**即时**的获奖分数线。本次竞赛的获奖率为 $w\%$，即当前排名前 $w\%$ 的选手的最低成绩就是即时的分数线。

更具体地，若当前已评出了 p 个选手的成绩，则当前计划获奖人数为 $\max(1, \lfloor p * w\% \rfloor)$，其中 w 是获奖百分比，$\lfloor x \rfloor$ 表示对 x 向下取整，$\max(x, y)$ 表示 x 和 y 中较大的数。**如有选手成绩相同，则所有成绩并列的选手都能获奖**，因此实际获奖人数可能比计划中多。

作为评测组的技术人员，请你帮 CCF 写一个直播程序。

输入格式

第一行有两个整数 n, w。分别代表选手总数与获奖率。
第二行有 n 个整数，依次代表逐一评出的选手成绩。

输出格式

只有一行，包含 n 个非负整数，依次代表选手成绩逐一评出后，即时的获奖分数线。相邻两个整数间用一个空格分隔。

样例 #1

样例输入 #1

```
10 60
200 300 400 500 600 600 0 300 200 100
```

样例输出 #1

```
200 300 400 400 400 500 400 400 300 300
```

样例 #2

样例输入 #2

```
10 30
100 100 600 100 100 100 100 100 100 100
```

样例输出 #2

```
100 100 600 600 600 600 100 100 100 100
```

提示

样例 1 解释

已评测选手人数	1	2	3	4	5	6	7	8	9	10
计划获奖人数	1	1	1	2	3	3	4	4	5	6
已评测选手的分数从高到低排列（其中分数线红色表示）	200	300 200	400 300 200	500 400 300 200	600 500 400 300 200	600 600 500 400 300 200	600 600 500 400 300 200 0	600 600 500 400 300 300 200 0	600 600 500 400 300 300 200 200 0	600 600 500 400 300 300 200 200 100 0

数据规模与约定

各测试点的 n 如下表：

测试点编号	$n=$
$1\sim3$	10
$4\sim6$	500
$7\sim10$	2000
$11\sim17$	10^4
$18\sim20$	10^5

对于所有测试点，每个选手的成绩均为不超过 600 的非负整数，获奖百分比 w 是一个正整数且 $1\le w\le 99$。

提示

在计算计划获奖人数时，如用浮点类型的变量（如 C/C++ 中的 `float`、`double`，Pascal 中的 `real`、`double`、`extended` 等）存储获奖比例 $w\%$，则计算 $5 \times 60\%$ 时的结果可能为 3.000001，也可能为 2.999999，向下取整后的结果不确定。因此，建议仅使用整型变量，以计算出准确值。

分析

最直观的做法是，每评测出一个人的成绩就将所有人排序一次，然后输出对应排名。一共来 n 人，共排序 n 次。如果使用快速排序，或者是 $sort()$ 函数，总的时间复杂度是 $O(n^2logn)$ 的，只能获得 50 分；如果使用插入排序，将新得到的分数插入到有序的分数数组中，总的时间复杂度是 $O(n^2)$ 的，可以获得 85 分。

发现需要排序的数值不超过 600 ,因此考虑其他的排序方法。使用计数排序，每次评出一个人的成绩，就加入到对应的 "桶" 里，然后再从高往低统计累加人数，达到获奖人数时输出分数线。这种做法的时间复杂度是 $O(An)$,这里的 A 是指成绩的值域上限，可以通过全部测试点。

```cpp
#include<iostream>
using namespace std;
int t[610],n,w;
int main()
{
    int tmp;
    cin>>n>>w;
    for(int i=1;i<=n;i++)
    {
        cin>>tmp;
        t[tmp]++;//加入对应的桶
        int sum=0;
        for(int j=600;j>=0;j--)
        {
            sum+=t[j];
            if(sum>=max(1,i*w/100))
            {
                cout<<j<<' ';
                break;
            }
         }
    }
    return 0;
 }
```

解决同一种问题可能有不同的算法，其算法复杂度和某些变量的大小有关，这时应该选择合适的算法使得效率最高。例如，对一个数列进行排序，如果这个数列的值域（假设是 A ）比较小而数量 n 较多，可以使用时间复杂度是 $O(n+A)$ 的计数排序；如果值域很大而数量没有那么多，则可以使用时间复杂度为 $O(nlogn)$ 的快速排序。

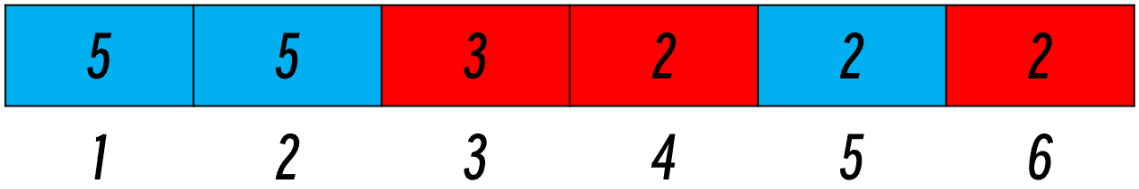
[NOIP2015 普及组] 求和

题目背景

洛谷 P2671，NOIP2015 普及组 T3

题目描述

一条狭长的纸带被均匀划分出了n个格子，格子编号从1到n。每个格子上都染了一种颜色$color_i$用$[1,m]$当中的一个整数表示），并且写了一个数字$number_i$。



定义一种特殊的三元组：(x,y,z)，其中x,y,z都代表纸带上格子的编号，这里的三元组要求满足以下两个条件：

1. xyz是整数,$x<y<z,y-x=z-y$
2. $colorx=colorz$

满足上述条件的三元组的分数规定为$(x+z)\times(number_x+number_z)$。整个纸带的分数规定为所有满足条件的三元组的分数的和。这个分数可能会很大，你只要输出整个纸带的分数除以$10,007$所得的余数即可。

输入格式

第一行是用一个空格隔开的两个正整数n和m, n表纸带上格子的个数，m表纸带上颜色的种类数。

第二行有n用空格隔开的正整数，第i数字$number$表纸带上编号为i格子上面写的数字。

第三行有n用空格隔开的正整数，第i数字$color$表纸带上编号为i格子染的颜色。

输出格式

一个整数，表示所求的纸带分数除以10007所得的余数。

样例 #1

样例输入 #1

```
6 2
5 5 3 2 2 2
2 2 1 1 2 1
```

样例输出 #1

```
82
```

样例 #2

样例输入 #2

```
15 4
5 10 8 2 2 2 9 9 7 7 5 6 4 2 4
2 2 3 3 4 3 3 2 4 4 4 4 1 1 1
```

样例输出 #2

```
1388
```

提示

【输入输出样例 1 说明】

纸带如题目描述中的图所示。

所有满足条件的三元组为：$(1,3,5),(4,5,6)$。

所以纸带的分数为$(1+5)\times(5+2)+(4+6)\times(2+2)=42+40=82$。

对于第 1 组至第 2 组数据，$1\le n\le 100,1\le m\le 5$；

对于第3 组至第 4 组数据，$1\le n\le 3000,1\le m\le 100$；

对于第 5 组至第6组数据，$1\le n\le 100000,1\le m\le 100000$，且不存在出现次数超过20的颜色；

对于全部 10 组数据，
$1\le n\le 100000,1\le m\le 100000,1\le color_i\le m,1\le number_i\le 100000$

分析

题目要求有几个三元组 (x,y,z) 符合要求。最简单的思路就是使用三重循环依次枚举 x,y,z 然后验证是否符合要求。这样的时间复杂度是 $O(n^3)$，效率很低，无法在限定时间内完成计算。

根据第一个条件可以得到 $y-x=z-y$ 得到 $(x+z)=2y$ 与 $\frac{x+z}{2}=y$，这个式子中 y 对答案没有影响，只是限制 x,z 满足条件 $z-x$ 为大于0的偶数（数学上称 x,z 奇偶性相同）。在满足前面的条件下，如果 x,z 颜色相同，那么有且仅有一组符合条件的三元组 (x,y,z)。将格子编号为奇数和格子编号为偶数的格子分开处理，这样可以得到一个时间复杂度为 $O(n^2)$ 的做法：枚举 x,z 的位置，再判断 x,z 的颜色和奇偶性是否相同，若同时满足，则可以统计进入最后的答案。虽然有所优化，但这种做法依然会超时。

将题目中给出计算分数的式子拆开，得到 $(x+z)\times(a_x+a_z)=xa_x+xa_x+za_z$ ，这是每对 (x,z) 对答案的贡献。如何快速计算所有 (x,z) 对的贡献总和呢?

不妨简化问题，对于一个数列 a_1,a_2,a_3,a_4,a_5 ，对于每一个满足 $1\le x<z\le 5$ 的一队数 (a_x,a_z) ，对答案都有 $xa_x+xa_z+za_x+za_z$ 的贡献，那么贡献的总和是多少呢？可以列表统计：

$i\setminus j$	1	2	3	4	5
1		$1a_1+1a_2+2a_1+2a_2$	$1a_1+1a_3+3a_1+3a_3$	$1a_1+1a_4+4a_1+4a_4$	$1a_1+1a_5+5a_1+5a_5$
2			$2a_2+2a_3+3a_2+3a_3$	$2a_2+2a_4+4a_2+4a_4$	$2a_2+2a_5+5a_2+35a_5$
3				$3a_3+3a_4+4a_3+4a_4$	$3a_3+3a_5+5a_3+5a_5$
4					$4a_4+4a_5+5a_4+5a_5$
5					

考察结果中所有系数是 1 的项，发现有4个 $1a_1$,此外还有 $1(a_2+a_3+a_4+a_5)$ 。

考察结果中所有系数是 2 的项，发现有4个 $2a_2$,此外还有 $2(a_1+a_3+a_4+a_5)$ .....以此类推，可以归纳为：对于所有系数为 i 的项，有 $5-1$ 个 ia_i 和 $i\sum_{j=1}^{5}a_j-a_i$ ，注意需要扣掉自己本身的一次。因此，对于同一颜色同一奇偶性的一类格子，都可以用这种方式计算。

使用 $s1$ 数组记录某种颜色且编号为奇数或偶数格子数目，用 $s2$ 数组记录某种颜色且编号为奇数或偶数格子的数字之和。那么对于每一个 i ，对答案的贡献就是

```
i*(s2[y] [i%2]-a[i]+a[i] * (s1[y] [i%2] -1))
```

其中，y 为第 i 个格子的颜色， i%2 是指 i 的奇偶性。

```cpp
#include<iostream>
using namespace std;
const int mod=10007;
const int maxn=1e5+10;
int n,m;
int a[maxn],b[maxn];
int s1[maxn][2],s2[maxn][2];
int ans;
int main()
{
    cin>>n>>m;
    for(int i=1;i<=n;i++)
        cin>>a[i];
    for(int i=1;i<=n;i++)
    {
        cin>>b[i];
        s1[b[i]][i%2]++;// 记录颜色为 b[i]，且编号与编号i奇偶性相同的格子的数目。
        s2[b[i]][i%2]=(s2[b[i]][i%2]+a[i])% mod;//记录颜色为 b[i] ，且编号与编号i奇偶性相同的格子上面的数字的和。
    }
    for(int i=1;i<=n;i++)
    {
        int y=b[i];
        ans+=i*(s2[y][i%2]+a[i]*(s1[y][i%2]-2)%mod)%mod;
        //多次取模的目的是在计算的过程中取余数而不会运算溢出。减法取模要小心，避免产生负数。
        ans%=mod;
    }
    cout<<ans;
    return 0;
 }
```

例题——Dart

题目描述

小明喜欢玩飞镖游戏，他会把每次的得分都记录在数组si中。

有个飞镖大奖，得奖规则是：如果你4次飞镖的得分**先后**是(a,b,c,d)，满足$a \times b \times c = d$

小明准备把记录里的其他项删除，只留下满足获奖条件的4个数，有多少种不同方案?

数据规模: $4 \leq N \leq 2000, 0 < si \leq 10^6$

输入样例

6 10 2 2 7 40 160

输出样例

2

题目分析

1. 枚举对象 a, b, c, d 的组合，判断是否满足条件
2. 该问题的是否存在数字上的特性可供优化?

   · 将表达式转化为 $a * b = d/c$
   · 则问题特性为
   (1) d/c 满足 c 是 d 的因子，$1 \leq d/c \leq 10^6$
   (2) $1 \leq a \times b \leq Max(si)$
3. 能否改变枚举的对象?

枚举 c，对于当前的 c，$cnt[k]$ 表示 $k = a * b$ 的方案数。

对于当前的 c，枚举 满足条件的 d，方案数增量为 $cnt[d/c]$

由当前 c 来更新 $cnt[k]$，算法效率 $O(n^2)$

```
#include<cstdio>
int N,maxn,s[2005],cnt[1000005];
long long t,Ans;
int main()
{
    scanf("%d",&N);
    for (int i=1;i<=N;i++)
    {
```

```
        scanf("%d",&s[i]);
        if (s[i]>maxn) maxn=s[i];
    }
    for (int c=1;c<N;c++)
    {
        for (int d=c+1;d<=N;d++)
            if (s[d]%s[c]==0) Ans+=cnt[s[d]/s[c]];
        for (int a=1;a<c;a++)
            if (((long long)s[a]*s[c])<=maxn)
            {
                t=s[a]*s[c];
                cnt[t]++;
            }
    }
    printf("%lld",Ans);
    return 0;
}
```

深度优先搜索

设想我们现在身处一个巨大的迷宫中，我们只能自己想办法走出去，下面是一种看上去很盲目但实际上会很有效的方法。

以当前所在位置为起点，沿着一条路向前走，当碰到岔道口时，选择其中一个岔路前进。如果选择的这个岔路前方是一条死路，就退回到这个岔道口，选择另一个岔路前进。如果岔路口存在新的岔道口，那么仍然按上面的方法枚举新岔道口的每一条岔道。这样，只要迷宫存在出口，那么这个方法一定能够找到它。

也就是说，当碰到岔道口时，总是以"**深度**"作为前进的关键词，不碰到死胡同就不回头，因此这种搜索的方式称为深度优先搜索$(Depth-First-Search)$

回溯算法

从问题的某一种可能情况出发，搜索所有能达到的可能情况，然后再以其中的一种可能情况为新的出发点，继续向下探求，这样就走出了一条“路”。当这一条路走到“尽头”但仍没寻找到目标的时候，再倒回到上个出发点，从另一个可能情况出发，继续搜索。这种不断“回溯”寻找目标的方法，称作“回溯法”。

回溯算法是所有搜索算法中最基本的一种算法。

算法思想

回溯法的基本思想是**穷举搜索**，一般适用于**寻找解集**或找出**满足某些约束条件的最优解**的问题。这些问题所具有的共性是**顺序性**，即必须先探求第一步，确定第一步采取的可能值，再探求第二步采取的可能值，然后是第三步……直至达到目标状态。

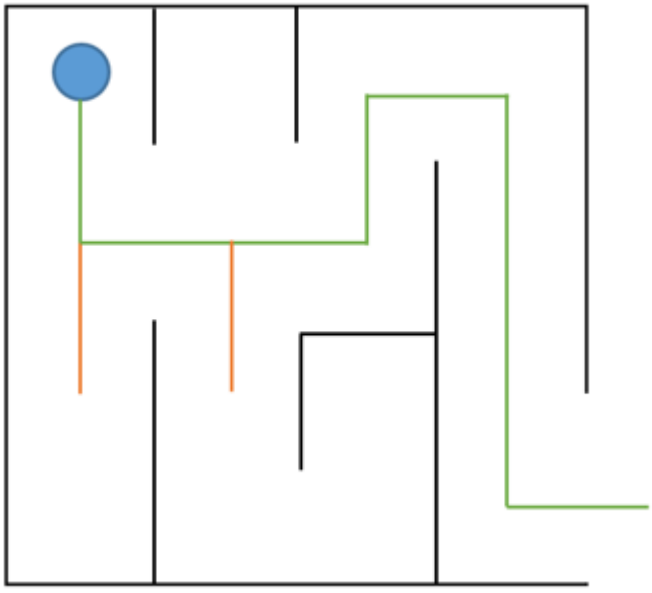
深度优先搜索（DFS）——回溯法

回溯算法实现的步骤：

我们一般采用**递归**程序来实现回溯的算法。

递归，即**函数调用自身**，以逐步减小问题的规模。但在一些问题中，并不是所有的递归路径都是有效的。

如图所示迷宫，很可能会进入橙色所标识的“死胡同”,只能回到之前的路径,直到找到绿色的解为止。



这种方法被称为**回溯法。**

回溯法往往会尝试一条**尽可能深而完整的搜索路线**，直至完全无法继续递归时才回溯，因而需要用**深度优先搜索（DFS）**实现。

在实现过程中，我们把问题分成 n 步完成，每一步都要解决相同的问题。因此，我们首先要从题目中分析出每一步具体是指什么问题，接着把每一步的所有可能列出来。例如:

在全排列问题中，每一步是指确定每一个位置上的数字，而每个位置上都有可能是$1 \sim n$中的任一数字。

在 $0/1$ 背包问题中，每一步是指每一个物品，每一个物品都有两种可能，选和不选。

深度优先搜索的基本框架

```
void dfs(当前状态)
   {
      if (到达搜索终点)
      {
          check() ;  // 判断解的合法性或更新较优解。
          return ;
      }
      else
      {
          枚举当前状态的所有可能取值：
              记录当前状态的值；
              dfs(下一状态)；
              将当前状态的值复原；
      }
   }
```

例题——四阶数独(Suduku)

这里讨论一种简化的数独—四阶数独。

给出一个 4×4 的格子，每个格子只能填写 1 到 4 之间的整数，要求每行、每列和四等分更小的正方形部分都刚好由 1 到 4 组成。

给出空白的方格，请问一共有多少种合法的填写方法。

2	4	1	3
1	3	2	4
4	2	3	1
3	1	4	2

问题分析

使用暴力枚举法。复杂度 $O(n^{n^2})$。

一共有$4 \times 4 = 16$ 个空格。使用16层循环。

每个空格可填入$1 \sim 4$共4个选项。总情况数

$4^{16} = 4294967296$。

$$4^{16} = (2^2)^{16} = 2^{32}$$

即比32位无符号整数的最大值$max\ unsigned\ int$恰好多1。

目前计算机一秒约可处理 10^8（一亿）次有效计算。

肯定是不行的。

继续

使用排列枚举法。复杂度$O(n^2(n!)^n)$

由于已经知道每一行内的数字不可能出现**重复**，每一行均可视为$1 \sim 4$ 的1个排列。因此可以使用排列枚举

1	2	3	4

1	2	4	3

1	3	2	4

使用类似计数排序的思路，使用数组维护每一列、每一个子正方形中 $1/2/3/4$ 是否出现，可以 $O(1)$ 完成**合法性判断**。

总情况数$(4!)^4 = 24^4 = 331776$。

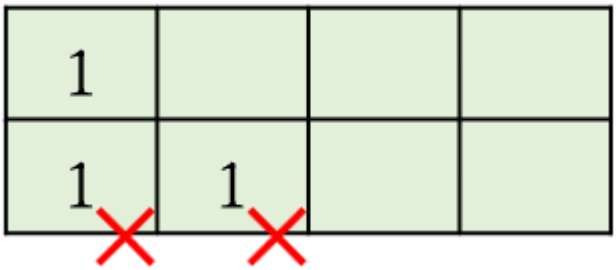
情况数不太大，乘上解的合法性判断$O(1)$，可以压入1秒。

回溯

第1行选择第一列为1，则：

第2行将1放入第1列，违反列规则和子正方形规则。

第2行将1放入第2列，违反子正方形规则。



但是，由于使用循环进行排列枚举，不得不生成后方的所有排列之后，才能进行判断是否合法，产生了大量的浪费，此时应考虑剪枝。

本题中，最基本的行为单元为“填入单个数字”。

因此使用回溯法时，每填入一个数字，都立刻进行判断并剪枝。

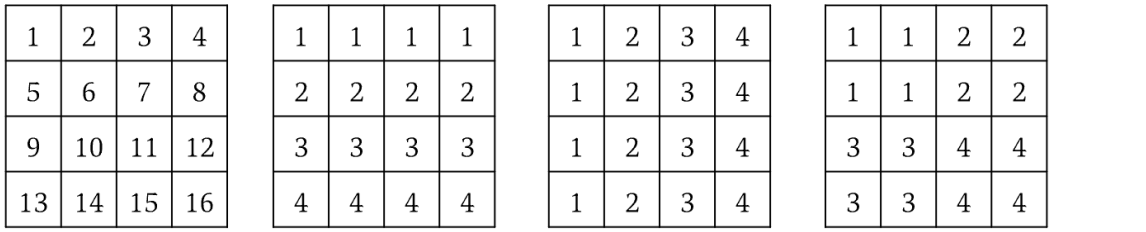
参考代码

```
void dfs(int x)
{
    // 第 x 个空填什么
    if (x>n) //  如果所有空都已填满
    {
        ans++; // 增加结果数量
        return ;
     }
 }

 // 根据 x 计算出所在行、列、小块编号

 int row=(x-1)/4+1; // 横行编号
 int col=(x-1)%4+1; // 竖排编号
 int block=(row-1)/2*2+(col-1)/2+1; // 小块编号

 // 枚举所填内容为 i

 for (int i=1;i<=4;i++)
    if (b1[row][i]==0 && b2[col][i]==0 && b3[block][i]==0)
     {
        // 合法性判断
         a[x]=i; // 记录放置位置
         b1[row][i]=1;b2[col][i]=1;b3[block][i]=1; // 占位
         dfs(x+1); // 下一层的递归
```

```
            b1[row][i]=0;b2[col][i]=0;b3[block][i]=0; // 恢复
        }
```

合法性判断

如何保证放置方法合法?

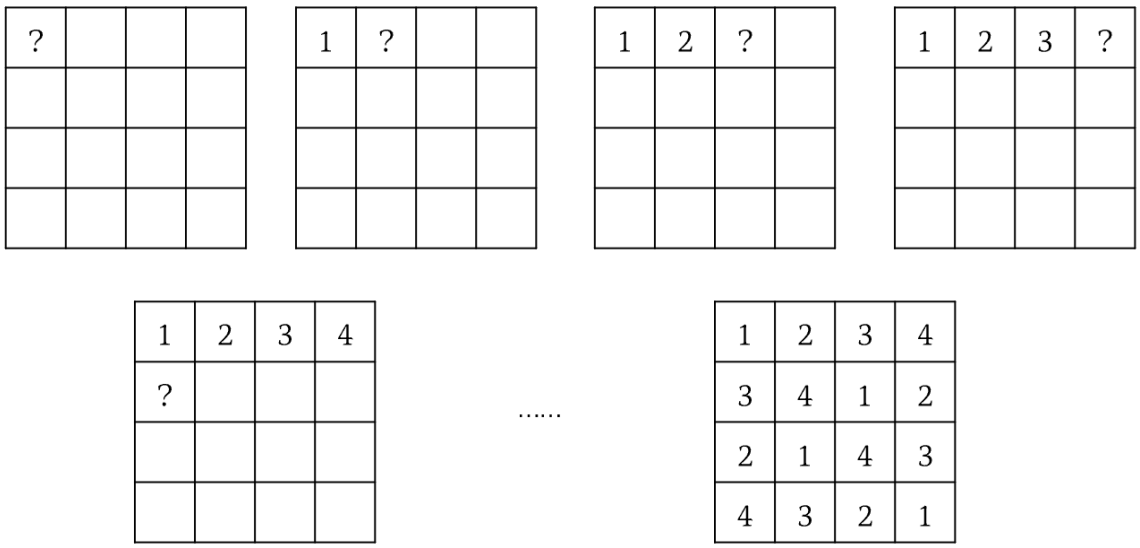


如图，可以构造方格编号与行列号之间的关系。

使用三个数组$b1$、$b2$、$b3$。$bi[i][j]$ 表示第i行是否已经存在j。类似地，$b2$、$b3$分别表示第i列、第i块是否已经存在j。

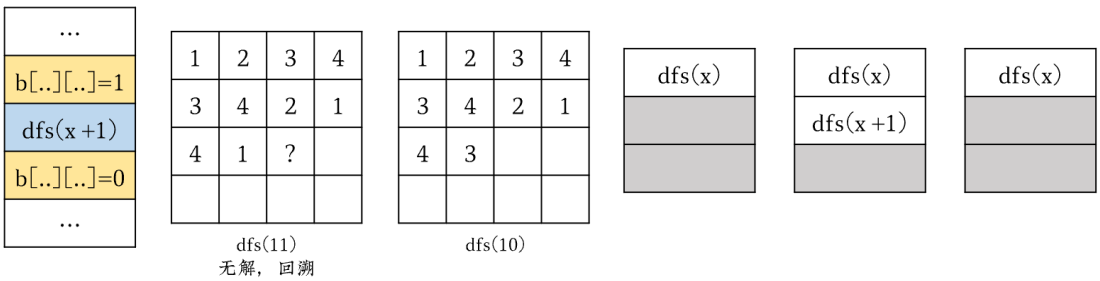
方案合法，当且仅当新数与现有的$b1$、$b2$、$b3$不冲突。

那么，递归是如何进行的?



为什么函数返回后就能回到上一状态?

如果所有方法枚举完毕，则这条路已经死胡同，需要**回溯**。



计算机运行函数时，为每一个子函数都分配了一片栈空间。

当回溯到上层时，下层的内容会离开栈，从而恢复上层的状态。

注意b是全局数组，并不能随着出栈恢复，需要手动清理。

引入递归法后，需要枚举状态数量级别，相对于枚举排列有了指数级的提升。排列枚举所对应的状态数量是阶乘的指数级，而可以证明，回溯法所枚举的状态数量只有指数级，由于删除了大量无需计算的状态，运行时间会有明显的提升。

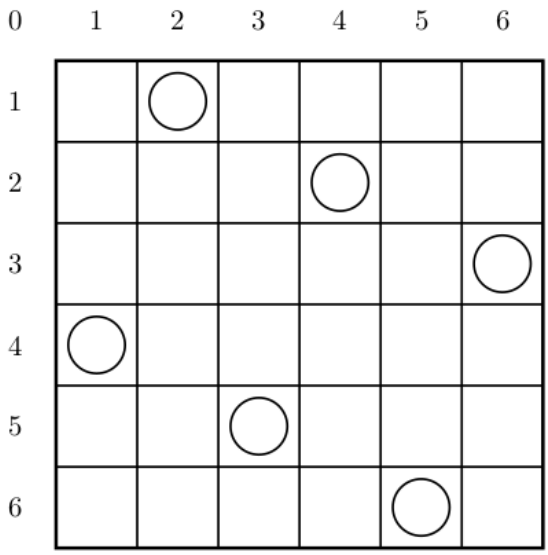
回溯法的基本模板：

```
void dfs(int k)
{
    if (所有空已经填完了)
    {
        判断最优解 或 记录答案;
        return ;
    }

    for (枚举这个空能填的选项)
        if (这个选项是合法的)
        {
            记录下这个空(当前状态)
            dfs(k+1);
            取消这个空(恢复)
        }
}
```

例题 [USACO1.5] 八皇后 Checker Challenge

题目描述

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。

输入格式

一行一个正整数 n，表示棋盘是 $n \times n$ 大小的。

输出格式

前三行为前三个解，每个解的两个数字之间用一个空格隔开。第四行只有一个数字，表示解的总数。

样例 #1

样例输入 #1

```
6
```

样例输出 #1

```
2 4 6 1 3 5
3 6 2 5 1 4
4 1 5 2 6 3
4
```

提示

【数据范围】
对于 100% 的数据，$6 \leq n \leq 13$。

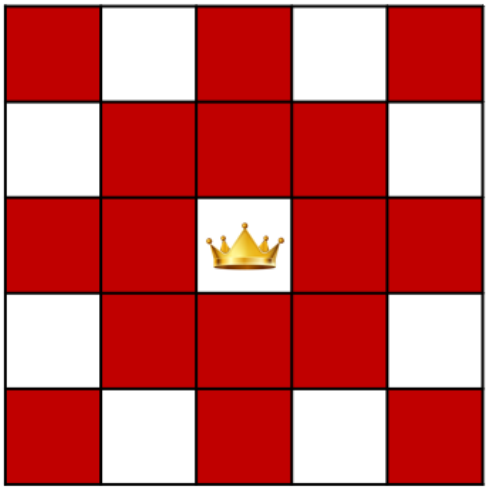
题目翻译来自NOCOW。

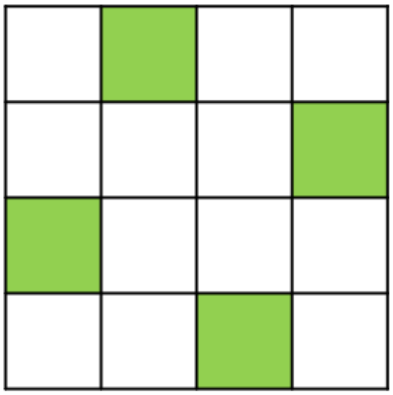
USACO Training Section 1.5

题目分析

在$n \times n$ 的国际象棋棋盘上放置n个皇后使他们互不攻击。皇后的攻击范围是同一行、同一列、同一斜线上的其他棋子。

n不超过13，求方案数，并输出前3种方法。





本题是经典的搜索回溯例题。可以套用回溯法基本模板。

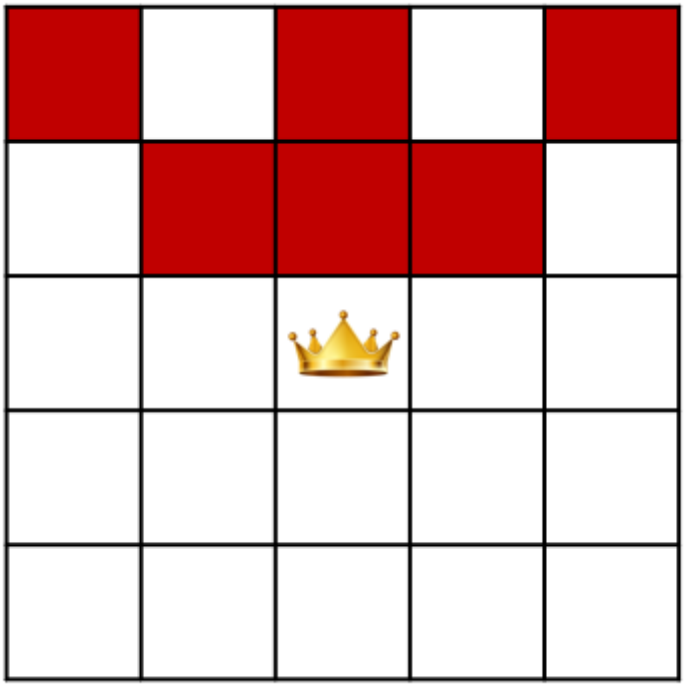
如何表达状态，便于判定皇后的合法性,也就是不会互相攻击。

只需要考虑上方的是否重复。为什么?

1. 行：因为按行枚举,所以每一行显然只能有一个皇后。

2. 列/斜角：循环枚举判断是否有重复列或者斜对角?可行，但要增加$O(n)$的复杂度，用于遍历上方所有皇后。

能否像前面例子一样,使用一个占位标记实现这一判定?



继续

需要判定：**正上方、左对角、右对角**上没有皇后。

正上方容易判断,直接为每一列上一个标记是否存在即可。

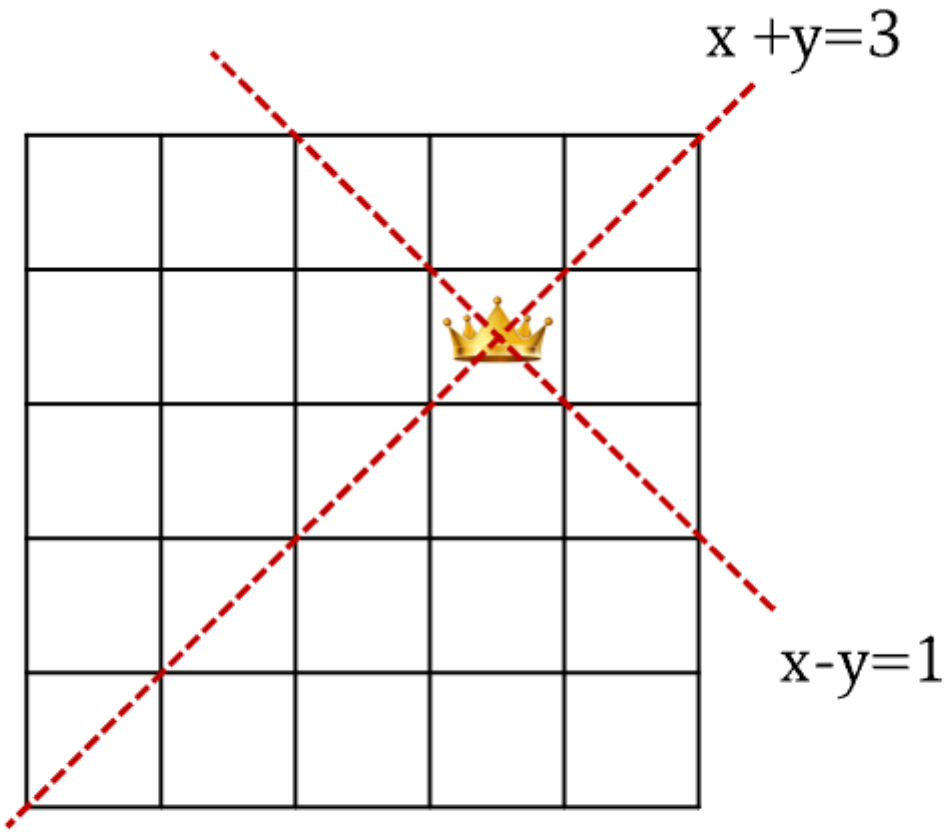
对角线上，可以发现一些性质:

1. 左下右上斜线：这些格子，(横坐标+纵坐标) 的值相同

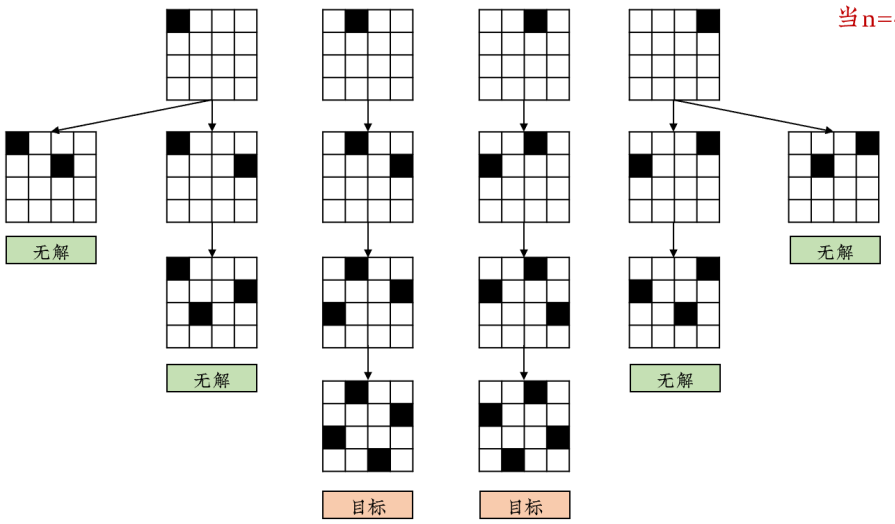
2. 左上右下斜线：这些格子，(横坐标-纵坐标) 的值相同

所以就建立三个数组表示列、两个斜线的状态。

(横坐标-纵坐标)可能是负数，加上n即可非负。



皆为数学上的一次函数



当n=4时的搜索过程

建立好标记数组，枚举各种状态，计划恢复状态

参考代码

```
int a[maxn],n,ans=0;
int b1[maxn],b2[maxn],b3[maxn]
// 分别记录 y ， x+y ,x-y+15 是否被占用
void dfs(int x)
{
    // 第 x 行的皇后放哪儿
    if (x>n) // 如果所有的皇后 放好
    {
        ans++; // 增加结果数量
        if (ans<=3)
        {
            for (int i=1;i<=n;i++)
                printf("%d",a[i]);
            puts(" ");
         }
         return ;
     }
    for (int i=1;i<=n;i++)
        if (b1[i]==0 && b2[x+i]==0 && b3[x-i+15]==0)
        {
            a[x]=i; // 记录放置位置
            b1[i]=1;b2[x+i]=1;b3[x-i+15]=1; // 占位
            dfs(x+1); // 递归
            b1[i]=0;b2[x+i]=0;b3[x-i+15]=0; // 恢复
        }
 }
```

例题——桐桐的全排列

问题描述

今天，桐桐的老师布置了一道数学作业，要求列出所有从数字1到数字n的连续自然数的排列，要求所产生的任一数字序列中不允许出现重复的数字。因为排列数很多，桐桐害怕写漏了，所以她决定用计算机编程来解决。

输入样例

3

输出样例

1 2 3

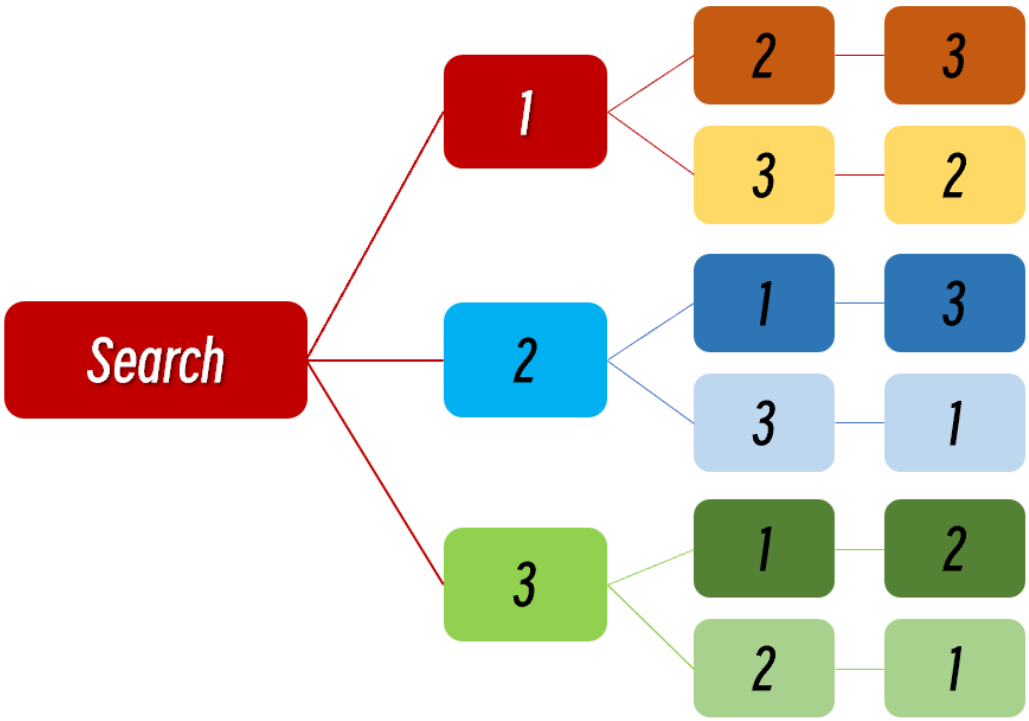
1 3 2

2 1 3

2 3 1

3 1 2

3 2 1



参考代码

```
#include<iostream>
#include<algorithm>
#include<cstring>

using namespace std;
typedef long long LL;
const int N=10;
int n;
int a[N];
bool flag[N];//标记某个数是否被使用过
void dfs(int k)
{
```

```
    if (k>n)
    {
        for (int i=1;i<=n;i++)
            cout<<a[i]<<" ";
            cout<<endl;
            return; //停止搜索
    }
    for (int i=1;i<=n;i++)
    {
        if (flag[i]==0)
        {
            a[k]=i;
            flag[i]=1;
            dfs(k+1);
            flag[i]=0;
        }
    }
 }
int main()
{
    cin>>n;
    dfs(1); //从第一个开始填
    return 0;
  }
```

深度优先搜索——剪枝

如果将搜索的状态看作结点，状态之间的转移看作边，搜索的过程就构建出了一棵“搜索树”。所谓“剪枝”，就是在搜索树上去掉一些枝权不进行搜索，从而减小时间消耗。

系统地说，剪枝的策略分为可行性剪枝和最优性剪枝两种。需要指出的是，搜索题目的性质极其灵活，在应用时一定要具体问题具体分析，设计合适的剪枝策略。

例题——桐桐的运输方案

题目描述

桐桐有N件货物需要运送到目的地，它们的质量和价值分别记为：

质量：$w1, w2, \cdots，wn$

价值：$v1, v2, \cdots, vn$

已知某辆货车的最大载货量为 x，并且当天只能运送一趟货物。这辆货车应该运送哪些货物，才能在不超载的前提下使运送的价值最大?

输入格式

第 1 行是一个实数，表示货车的最大载货量$x(1 < x \leq 100)$。

第2行是一个正整数，表示待运送的货物数$n(1 < n \leq 20)$。
后面 n 行每行两个用空格隔开的实数，分别表示第 1 至第 n 件货物的质量w和价值v。

输出格式

第 1 行为被运送货物的总价值（只输出整数部分);
第 2 行为按编号大小顺序输出所有被运送货物的编号（当一件都不能运送时，不输出)。

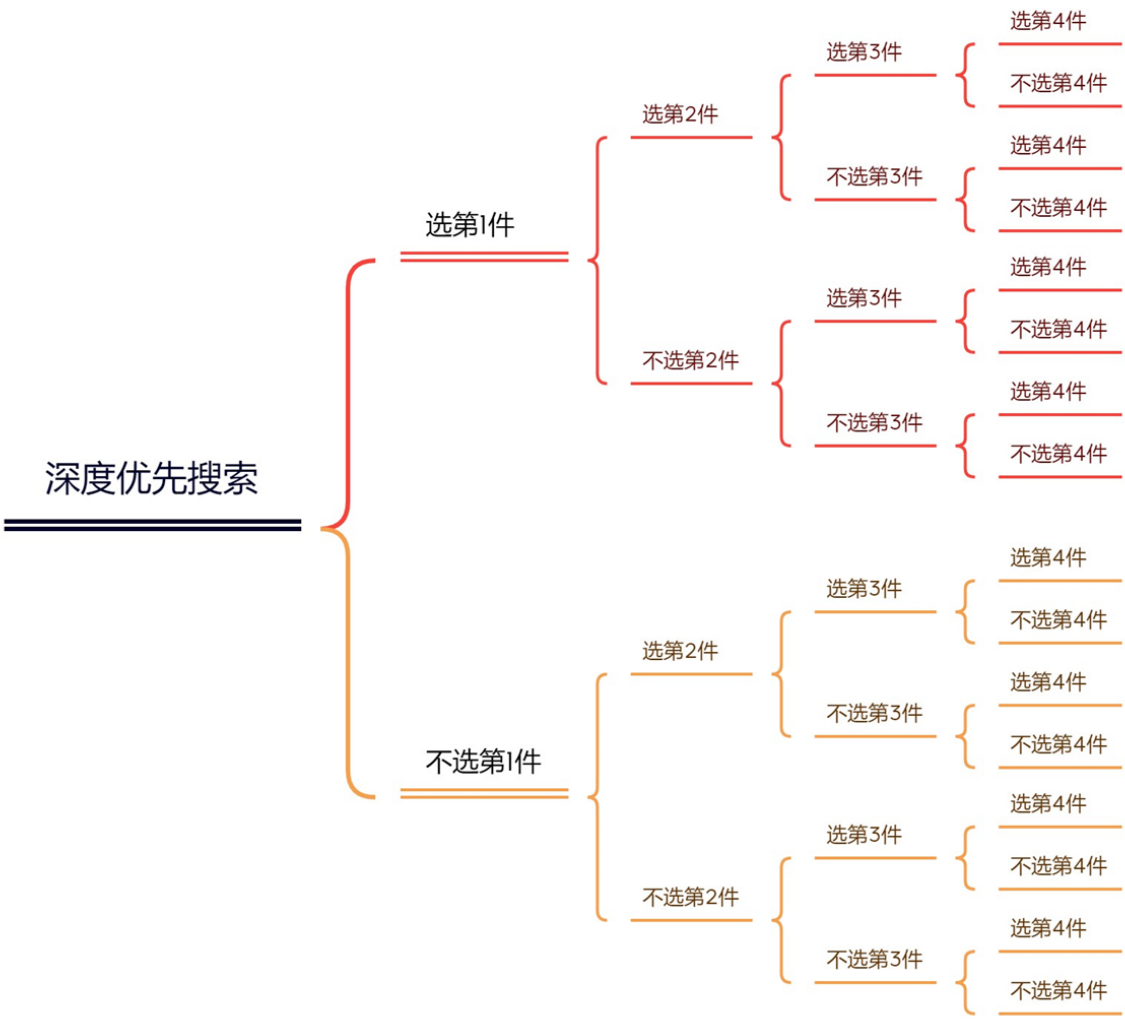
输入样例

```
20
4
3.5 4
4 5
5 6.8
6.9 7
```

输出样例

```
22
1 2 3 4
```

题目分析



根据题意，我们可知对于每一种货物只有选和不选两种情况，所以最坏情况下它的时间复杂度是 $O(2^n)$

参考代码1

```
#include<iostream>
#include<algorithm>
#include<cstring>

using namespace std;
typedef long long LL;
const int N=30;
int m,n;
double w[N],v[N];
double best;
bool flag[N];
int b[N]; // 存储物品编号
void dfs(int k,double weight,double price)
{
    // 搜索的边界问题
    if (price>best)
    {// 如果搜索到最大值则更新最大值与路径
        best=price;
```

```
        for(int i=1;i<=n;i++)
            if(flag[i])
                b[i]=i;
    }
    if (k>n) return;
    if (weight>m) return;
    flag[k]=1;
    dfs(k+1,weight+w[k],price+v[k]);
    flag[k]=0;
    dfs(k+1,weight,price);
}
int main()
{
    cin>>m>>n;
    for (int i=1;i<=n;i++)
        cin>>w[i]>>v[i];
    dfs(1,0,0);
    cout<<int(best)<<endl;
    for(int i=1;i<=n;i++)
    {
        if(b[i])cout<<i<<endl;
    }
    return 0;
 }
```

参考代码2——剪枝

增加一个数组 $f[x]$ 用于存放后缀和，增加条件判定若分支后续（剩余）价值和比当前最大值还小，就剪掉

```
#include<iostream>
#include<algorithm>
#include<cstring>

using namespace std;
typedef long long LL;
const int N=30;
int m,n;
double w[N],v[N];
double best;
bool flag[N];
int b[N]; // 存储物品编号
int f[N]; // 增加一个数组 用于后缀和
void dfs(int k,double weight,double price)
{
    // 搜索的边界问题
    if (price>best)
    {// 如果搜索到最大值则更新最大值与路径
        best=price;
        for(int i=1;i<=n;i++)
            if(flag[i])
                b[i]=i;
    }
```

```
    if (k>n) return;
    if (weight>m) return;

    //剪枝：价值和若不大于最大值，则舍弃
    if (price+f[k]<best) return ;

    flag[k]=1;
    dfs(k+1,weight+w[k],price+v[k]);
    flag[k]=0;
    dfs(k+1,weight,price);
}
int main()
{
    cin>>m>>n;
    for (int i=1;i<=n;i++)
        cin>>w[i]>>v[i];

    // 后缀和：本身+后面的值
    for (int i=n;i>0;i--)
        f[i]=f[i+1]+v[i];

        dfs(1,0,0);
    cout<<int(best)<<endl;
    for(int i=1;i<=n;i++)
    {
        if(b[i])cout<<i<<endl;
    }
    return 0;
 }
```

例题——[NOIP2002 普及组] 选数

题目描述

已知 n 个整数 $x_1, x_2, \cdots, x_n$，以及 1 个整数 k（$k < n$）。从 n 个整数中任选 k 个整数相加，可分别得到一系列的和。例如当 $n = 4$，$k = 3$，4 个整数分别为 $3, 7, 12, 19$ 时，可得全部的组合与它们的和为：

$3 + 7 + 12 = 22$

$3 + 7 + 19 = 29$

$7 + 12 + 19 = 38$

$3 + 12 + 19 = 34$

现在，要求你计算出和为素数共有多少种。

例如上例，只有一种的和为素数：$3 + 7 + 19 = 29$。

输入格式

第一行两个空格隔开的整数 n, k（$1 \le n \le 20$，$k < n$）。

第二行 n 个整数，分别为 $x_1, x_2, \cdots, x_n$（$1 \le x_i \le 5 \times 10^6$）。

输出格式

输出一个整数，表示种类数。

样例 #1

样例输入 #1

```
4 3
3 7 12 19
```

样例输出 #1

```
1
```

提示

【题目来源】

NOIP 2002 普及组第二题

题目分析

设计算法不难，设 $dfs(int\ x, int\ y, int\ z)$ 表示已经考虑前 $x-1$ 个数，正在考虑要不要选择第 x 个数，目前已经选择了 y 个，且选择的数字的和为 z 。那么，函数 $dfs(x,y,z)$ 这个状态可以转移到 $dfs(x+1,y,z)$ ，即不选择第 x 个数，或者是 $dfs(x+1,y+1,z+a[x])$,即选择了第 x 个数。

所谓可行性剪枝，就是“保证当前搜索的状态是合法的”，并且“从当前状态继续搜索时，至少能找到一个最终解”。

注意到只能选择 k 个数，因此当 $y>k$ 时，当前状态已经不合法了，因为选择了多于 k 个数，此时可以直接结束当前层的递归返回上一状态，因为继续搜索下去绝不可能达到任何一个“选择了 k 个数”的合法状态。

同时，因为必须选择 k 个数，如果将剩下没被考虑的数 $(x\sim n)$ 全部选上，再加上前 $x-1$ 个数中被选择的，还是凑不够 k 个数时，当前状态无论向后怎么选都无法选够 k 个数，那么这种情况也可以直接返回上一层递归，即 $n-x+1+y<k$ 时，直接返回。

上述两个剪枝就是可行性剪枝，对弈代码如下：

```
void dfs(int x,int y,int z)
{
    if (y>k) return; // 可行性剪枝 1
    if (n-x+1+y<k) return;  // 可行性剪枝 2
    if (x>n)
    {
        if(is_prime(z)) // 如果总和是质数，则增加答案
            ans++;
        return;
     }
    dfs(x+1,y,z);
    dfs(x+1,y+1,z+a[x]);
 }
```

不难发现，在上述代码中，搜索树最下面一层只有 C_n^k 个结点，而去掉被剪枝的状态，搜索树上每一个结点都对应了至少一个最下一层的结点，即每一层的合法结点数都不超过 C_n^k ,因为搜索树只有 n 层，因此总的合法结点数不超过 $n \times C_n^k$ 。每一个合法结点最多拓展出两个需要被剪枝的不合法结点，不合法结点不会有后代，因此搜索树的总结点数为 $O(nC_n^k)$ 。

经过剪枝，将一个原本为 $O(n2^n)$ 的做法优化为一个与合法结果相关的复杂度。同时，不是每一层都与最下层有相同的结点数，因此实际的结点个数远低于之前的分析。

例题——吃奶酪

题目描述

房间里放着 n 块奶酪。一只小老鼠要把它们都吃掉，问至少要跑多少距离？老鼠一开始在 $(0,0)$ 点处。

输入格式

第一行有一个整数，表示奶酪的数量 n。

第 2 到第 $(n+1)$ 行，每行两个实数，第 $(i+1)$ 行的实数分别表示第 i 块奶酪的横纵坐标 x_i, y_i。

输出格式

输出一行一个实数，表示要跑的最少距离，保留 2 位小数。

样例 #1

样例输入 #1

```
4
1 1
1 -1
-1 1
-1 -1
```

样例输出 #1

```
7.41
```

提示

数据规模与约定

对于全部的测试点，保证 $1 \leq n \leq 15$，$|x_i|, |y_i| \leq 200$，小数点后最多有 3 位数字。

提示

对于两个点 (x_1, y_1)，(x_2, y_2)，两点之间的距离公式为 $\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$。

2022.7.13：新增加一组 Hack 数据。

题目分析

考虑一个最优化问题，在搜索过程中已经得出了上述的一个解 x，但不确定它是否是最优的。在某个状态中，发现从当前状态在最优情况下到达终点所得到的解也不优于已得到的解 x，则可以直接剪枝返回。这种优化思路被称为最优性剪枝。

考虑用深度优先搜索解决这个问题，设 $dfs(deep, now, sum)$ 表示目前正在决定走第 $deep$ 步，当前正在第 now 个点，已经走过的距离是 sum。假设已经得到一个吃掉所有奶酪的距离是 ans，如果继续搜索过程，当下一个的距离超过了 ans，显然后续无论如何走，都不可能搜索到一个更优的解，此时直接剪枝，不去搜索这个点即可。

```cpp
#include<cstdio>
#include<cmath>
#include<cstdlib>
using namespace std;
int n,cnt=0;
double x[16],y[16],dist[16][16];
double ans;
bool vis[16];
void dfs(int deep,int now,double sum)
{
    cnt++;
    if(cnt>=100000000) // 卡个时间位，快要超时的情况下输出答案退出
    {
        printf("%.2f",ans);
        exit(0); // 退出整个程序，需要用到 cstdlib 头文件
    }
    if (deep>n)
    {
        if (sum<ans)
            ans=sum;
        return;
     }
    for (int i=1;i<=n;i++)
        if (!vis[i])
        {
```

```
            double t=sum+dist[now][i];
            if (t>=ans) continue; // 最优性剪枝
            vis[i]=1;
            dfs(deep+1,i,t);
            vis[i]=0;
        }
 }
 int main()
 {
    scanf("%d",&n);
    ans=1e6;
    x[0]=0;y[0]=0;
    for (int i=1;i<=n;i++)
        scanf("%lf%lf",&x[i],&y[i]);
    for (int i=0;i<=n;i++) // 预先处理出各点之间的距离
        for (int j=0;j<=n;j++)
            dist[i][j]=sqrt((x[i]-x[j])*(x[i]-x[j])+(y[i]-y[j])*(y[i]-
y[j]));
    dfs(1,0,0);
    printf("%.2f",ans);
    return 0;
 }
```

本代码还使用上被称为“卡时”的技巧：记录已经搜索的结点数量 cnt ，当 cnt 的值超过一定数量时停止搜索，直接将目前掌握的最佳答案输出。使用卡时技巧带有一点“赌博”的成分，首先需要赌已经搜索到的最优答案就是最优答案，其次需要赌搜索次数上限——上限太低可能会漏过最佳答案，而上限过高就会导致程序超时。可以在本地运行程序时构造可能造成超时的极限数据，借助每次 IDE 运行时间调整确定合适的上限。当然需要考虑到评测机的配置，做一个保守估计。

由于使用“卡时”技巧并不一定搜索完全部的状态，运气不好的话可能得到错误的答案，但效率高且实现起来较为容易。类似的技巧还有爬山法，模拟退火算法和遗传算法等。本题可以使用状态压缩动态规划控制时间复杂度。

最优剪枝还有几种策略：

1. 使用贪心的方式，将可能性大的策略优先枚举。例如走迷宫时，优先往终点方向前进。就本题而言，对于每次决策，可以先去枚举距离这个点最近的点（需要预处理每个点到其他点的距离排行）。这样可能可以略微加快到最优解的速度。
2. 目前状态中，如果可以比较方便的判断出后面任何决策都无法使当前状态相比于已经得到的最优解更优，则可以立刻放弃继续搜索下去。例如在走迷宫时，当前已经走的路径距离加上当前点到终点的直线距离已经超过了最优解，则可以不用继续搜索下去。

例题—— 小木棍

题目描述

乔治有一些同样长的小木棍，他把这些木棍随意砍成几段，直到每段的长都不超过 50。

现在，他想把小木棍拼接成原来的样子，但是却忘记了自己开始时有多少根木棍和它们的长度。

给出每段小木棍的长度，编程帮他找出原始木棍的最小可能长度。

输入格式

第一行是一个整数 n，表示小木棍的个数。
第二行有 n 个整数，表示各个木棍的长度 a_i。

输出格式

输出一行一个整数表示答案。

样例 #1

样例输入 #1

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

样例输出 #1

```
6
```

提示

对于全部测试点，$1 \leq n \leq 65$，$1 \leq a_i \leq 50$。

题目分析

简单的想法是：枚举原始木棍的长度 d，通过深度优先搜索尝试将所有的木棍拼起来，看能不能拼出若干根长度为 d 的长木棍。具体来说，设 $dfs(u,k)$ 表示还剩 k 根长木棍要拼起来（包括当前的），当前长木棍已经拼了 u 这么长。边界条件是 $k=0$ 时能拼出长木棍，初始时状态为 $dfs(0,\frac{\sum_{i=1}^{n}a_i}{k})$。转移时，枚举下一根木棍选择哪一根拼在当前长木棍上。

```
void dfs(int u,int k)
{
    if (u==0)
    {
        dfs(d,k-1);
        return;
    }
    if (k==0)
    {
        printf("%d\n",ans);
        exit(0);
    }
    for (int i=1;i<=n;++i)
        if ((used[i]==false) && (a[i]<=u))
            dfs(u-a[i],k);

}
```

但是这段代码无法通过本题，考虑进行剪枝：

1. 设 $s=\sum_{i=1}^{n} a_i$，显然 d 应该是 s 的因数，因此对于 d 不能整除 s 的情况，可以直接不搜索。
2. d 最小也需要所有木棍中最长的长度，因此对 d 的枚举可以从 a_i 的最大值开始。
3. 考虑对一个被枚举的 a_i，若它不能满足要求，则对于所有的 j 使 $a_j=ai$ 都不能满足题意，因此一个长度不符合要去时，无需搜索同长度的其他木棍。
4. 直观上，考虑短木棍比长木棍更灵活，也就是更能胜任长度较小的情况。因此我们在枚举新拼接的木棍时，优先考虑较长的木棍，把较短的木棍留给剩余长度更小的情况。
5. 如果当前木棍剩余长度等于当前考虑的木棍长度（即 $ai=u$），但搜索后发现当前木棍拼接并不合适，则说明以前的选择不对，无需考虑其他木棍直接回溯即可。（理由：用一些短棍拼接出当前长度和一根较长的棍子拼接效果是一样的，而短棍更灵活，留到后面显然更好）
6. 如果当前木棍长度就是长木棍的长度，但拼接后不符合要去，也可以直接返回。

根据以上分析，重置组织代码，用一个 len 数组存当前长度为 i 的木棍还剩多少根，预处理一个 pre 数组表示长度小于 i 的最长木棍的长度，在 dfs 函数中加入参数 p，表示当前长棍用的最短棍长是 p，则枚举当前小木棍时，只需要从 p 到 $mina_i$ 逐次枚举长度即可。

```
#include<cstdio>
#include<cstdlib>
#include<algorithm>
const int maxn=66;
int n,sum,mn=maxn,mx,d;
int len[maxn],a[maxn],pre[maxn];

void dfs(int u,int k,int p)
{ //当前长棍还有u没拼，还有k根长棍没拼，当前长棍的最短长度是p
    if (u==0)
    {
        dfs(d,k-1,a[n]);
        return ;
     }
    if (k==0)
    {
        printf("%d\n",d);
        exit(0); // 找到解
    }
    p=(p<u)?p:u;
    while(p && len[p]==0)
        --p;
    while(p)
        if (len[p])  // 枚举
        {
            --len[p];
            dfs(u-p,k,p);
            ++len[p];  // 回溯
            if ((u==p)||(u==d))  // 最后一条剪枝
                return;
            p=pre[p] ;
         }
        else p=pre[p];

 }
 int main()
```

```
{
    scanf("%d",&n);
    for (int i=1,x=1;i<=n;++i)
    {
        scanf("%d",&x);
        sum+=x;
        ++len[a[i]=x];
     }
    std::sort(a+1,a+1+n);
    for (int i=1;i<=n;++i) //预处理长度为i的上一个长度
        if (a[i]!=a[i-1])
            pre[a[i]]=a[i-1];
    for (d=a[n];(d<<1)<=sum;++d)
        if ((sum%d)==0)
            dfs(d,sum/d,a[n]);
    printf("%d\n",sum);
    return 0;
}
```

深度优先搜索——面积与路径

例题——面积计算

题目描述

桐桐拿到了一幅图，它全是由 '0' 和 '*' 组成，她想计算由 '*' 号所围成的图形的面积。面积的计算方法是统计 '*' 号所围成的闭合曲线中 '0' 的数目。

输入格式

由 '0' 和 '*' 组成的图（最多10行，每行不超过200个字符）

输出格式

面积数

输入样例

* * 0 0 *

0 * * * *

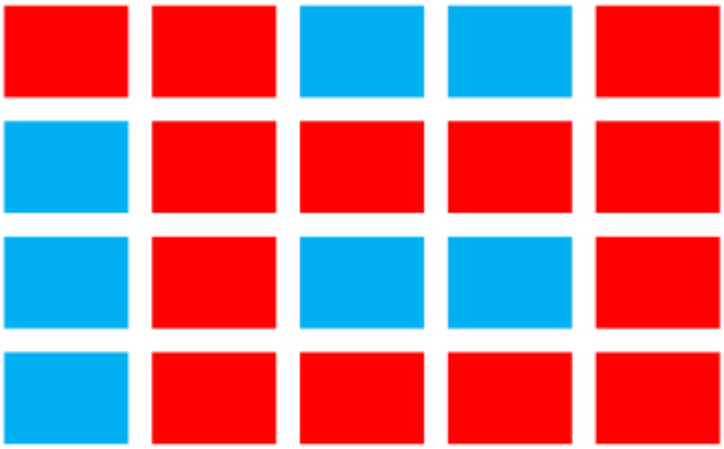
0 * 0 0 *

0 * * * *

输出样例

2

题目分析



首先要解决一个问题，就是搜索从那个字符的位置开始。如果从左边的 '0' 的位置开始，那么一开始就会遇上边界问题，导致可继续搜索的几率骤降。

但是有种特殊情况，如果左边两列都是 '*' 的话，那如何保证深度优先搜索覆盖到右边的正确数据呢？

深度优先搜索算法要点

1. **结点及结点之间的关系**

结点是搜索中到达的状态，在面积计算中，问题的状态是用矩阵的坐标表示。结点与结点之间的关系表示一个结点与其他结点相关联的方法，本题中指可到达**上下左右**四个方格。

2. **边界条件**

即在什么情况下程序**不再递归**下去。在面积计算中边界条件为矩阵的边界。

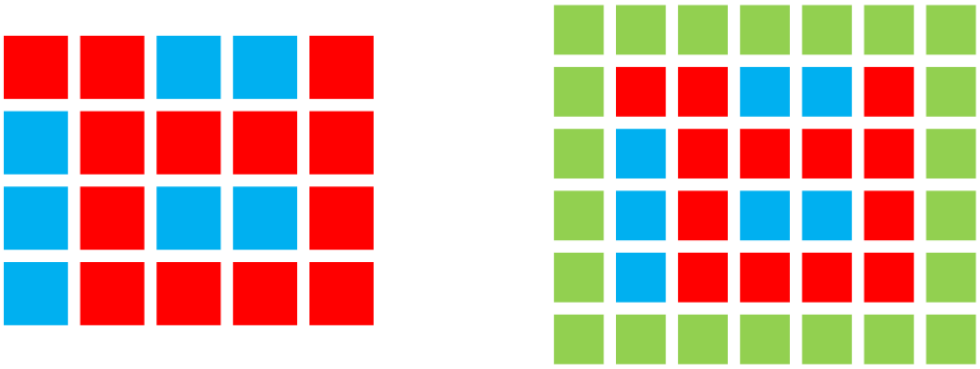
3. **约束条件**

当前**扩展**出一个子结点后应满足什么**条件**方可继续递归下去。在面积计算中，只有当前结点是未访问过的并且不是'*'号才能继续递归下去。

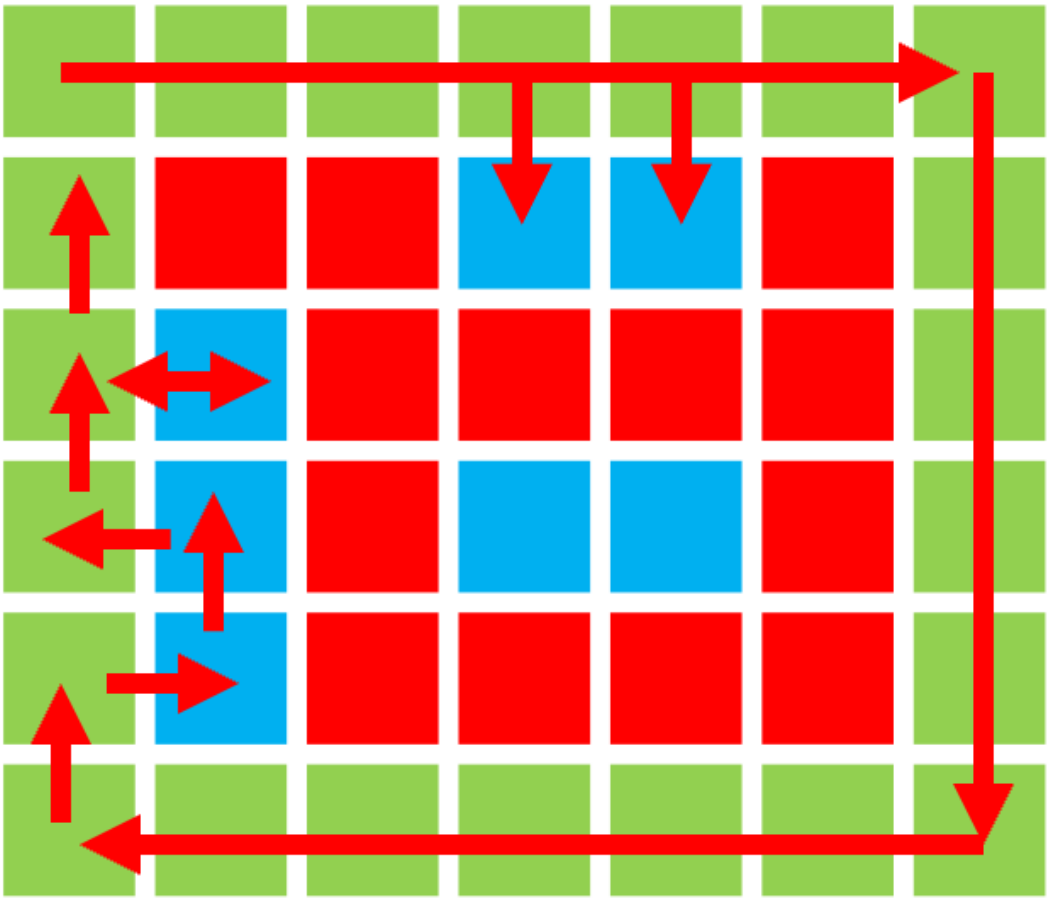
4. **恢复递归前的状态**

如果扩展子结点的过程有**全局变量**参与，则回溯前必须**恢复**其递归前的值。

回到面积这道题，一般人都以为是从左上角开始，但如果左上角是$'*'$，那就无法搜下去了。因此我们应该从四条边的每一个位置开始搜一遍，为了使程序实现更简单，我们可以**在矩阵外面加四条边**，全是$'0'$，然后从**新的左上角**开始搜索，就可以解决这个问题。另外，当矩阵中的某个位置已经访问过，就不需要再访问，否则会出现死循环。



如图，添加四条边后，搜索就从新的左上角开始，从第一行由左向右搜索，到达边界后（此时$y > n$）往下继续搜，到达右下边界后（此时$x > m$）往左继续，到达左下边界（此时$y < 0$）改为往上，然后继续



```cpp
#include<iostream>
#include<algorithm>
#include<cstring>

using namespace std;
typedef long long LL;
const int N=210;
int map[20][210]; // 图,1表示走过了，2表示墙，0表示没走过
int ans; // 结果
string st;
int m,n; // 行与列

void dfs(int x,int y)
{  // 边界问题，约束条件
    if (x<0 || y<0 || x>m+1 || y>n+1 ||map[x][y]!=0) return ;
    map[x][y]=1;
    // 结点与结点间的关系
    dfs(x+1,y); // 向下
    dfs(x-1,y); // 向上
    dfs(x,y+1); // 向右
    dfs(x,y-1); // 向左
}


int main()
```

```
{
    freopen("area.in","r",stdin);
    freopen("area.out","w",stdout);
    while(cin>>st)
    {
        ++m;n=st.length();
        for(int i=1;i<=n;i++)
        {
            if(st[i-1]=='*') map[m][i]=2;
        }
    }
    dfs(0,0); // 外围起点开始跑图
    for (int i=1;i<=m;i++)
        for (int j=1;j<=n;j++)
            if (map[i][j]=0) ++ans;
    cout<<ans<<endl;
    return 0;

}
```

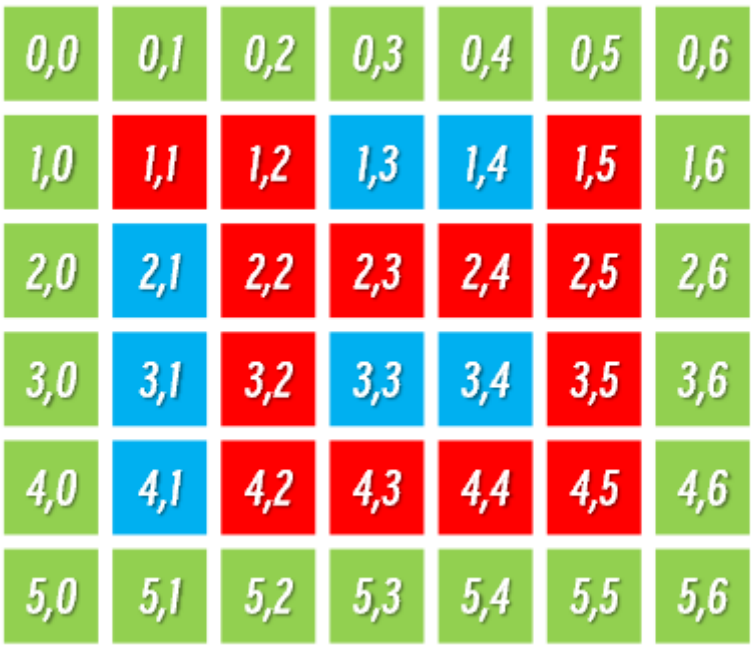
栈的问题

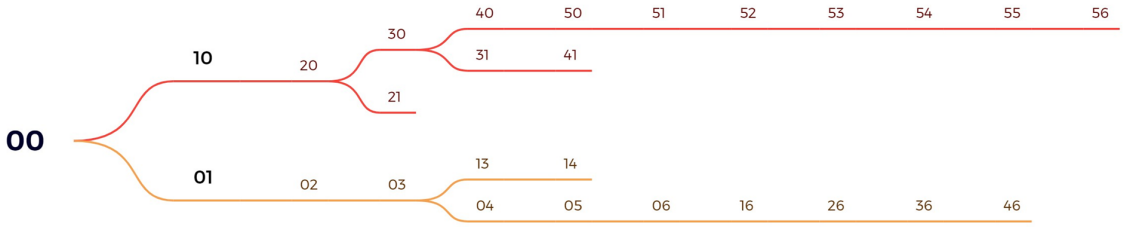
深度优先搜索 DFS 是先搜索到最底层再返回,如果图形里没有$'*'$ 而全是$'0'$的话，那么最多是有 10×200个$'0'$，总共是2000层，栈空间有些紧张。

宽度优先搜索解法

与深搜往一个方向搜到底，不同宽搜聚焦当前点周围的情况，使用队列扩展到位置周围，总体按照**由内向外**的顺序进行。

与深搜相比，宽搜的时间复杂度没有太大变化，但空间复杂度在一般情况下有所优化。





参考代码

```
#include<iostream>
#include<algorithm>
#include<cstring>

using namespace std;
typedef long long LL;
const int N=1010;
const int dx[4]={1,-1,0,0};
const int dy[4]={0,0,1,-1};
int map[20][210]; // 图,1表示走过了，2表示墙，0表示没走过
int ans; // 结果
string st;
int m,n; // 行与列
int que[N*2][2]; // 每个节点最多扩展2个 [0]表示 x，[1]表示 y

void bfs(int x,int y)
{
    int head=0,tail=1;
    int nx,ny; //扩展的子节点的坐标信息
    map[x][y]=1; // 走过的
    que[tail][0]=x,que[tail][1]=y;
    while(head!=tail)
    {
        ++head; // 头指针右移扩展
        x=que[head][0],y=que[head][1];
        for (int i=0;i<4;i++)
        {
            nx=x+dx[i],ny=y+dy[i];
            if(nx<0 || ny<0 || nx>m+1 || ny>n+1|| map[nx][ny]!=0) continue;
            // 边界或者走过 注意不能return 这不是递归
            map[nx][ny]=1; // 标记当前点
            tail++; //尾指针移动
            que[tail][0]=nx,que[tail][1]=ny;
         }
     }

}


int main()
{
    freopen("area.in","r",stdin);
```

```
    freopen("area.out","w",stdout);
    while(cin>>st)
    {
        ++m;n=st.length();
        for(int i=1;i<=n;i++)
        {
            if(st[i-1]=='*') map[m][i]=2;
        }
    }
    bfs(0,0); // 外围起点开始跑图
    for (int i=1;i<=m;i++)
        for (int j=1;j<=n;j++)
            if (map[i][j]=0) ++ans;
    cout<<ans<<endl;
    return 0;

}
```

宽度优先搜索算法

扩展顺序

沿着深度逐层进行

路径最短

搜索方法保证找到最短（步数最少）的解答序列

基本策略

用**队列**保存带扩展的节点，从**队首**取出节点，扩展出的新节点**放入队尾**，直接找到目标节点（问题的解）

```
void bfs()
{
    将初始节点带入队列;
    while (队列不为空且未找到目标节点)
    {
        取队头结点扩展，扩展出其子结点;
        判断子节点特性必要时记住每个节点的父结点;
        判断子节点是否是目标结点;
        判断子节点是否与易扩展的结点重复，否就入队。
    }
}
```