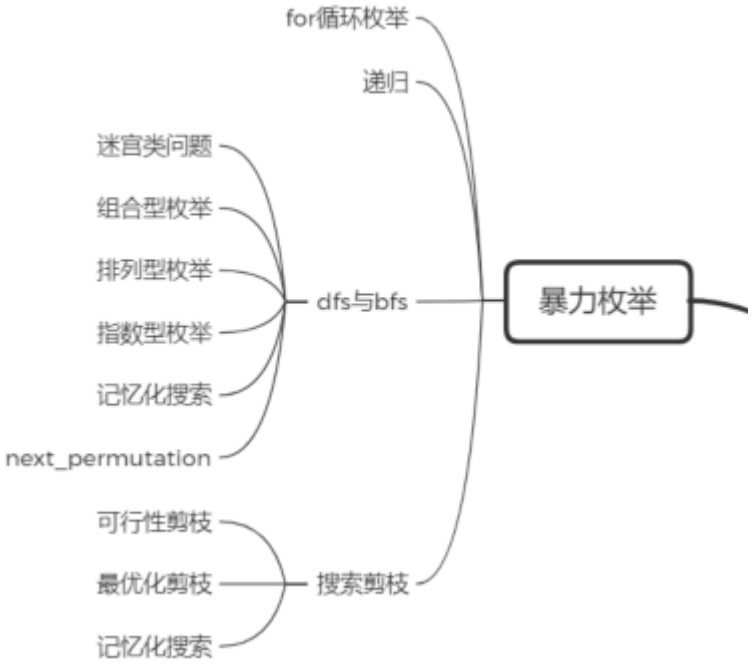
蓝桥杯十天冲刺省一


for循环枚举
递归
迷宫类问题
组合型枚举
排列型枚举
dfs与bfs
指数型枚举
记忆化搜索
next_permutation
暴力枚举
可行性剪枝
最优化剪枝
搜索剪枝
记忆化搜索

Day-2 暴力枚举

for及while循环枚举

- 主要就是利用循环嵌套和 `dfs` 去求解，但由于循环层数或递归层数过多过多存在超出时间复杂度的情况，所以通常用来骗取部分分。关于循环枚举的题目很多，大家可以在题库中的历年真题找到对应章节进行练习。

填数类问题

蛇形填数（蓝桥杯C/C++2020B组省赛第二场）

```
#include <bits/stdc++.h>

using namespace std;

int main() {
    int m = 20, n = 20, num = 0;
    int cnt = m + n - 1; // 确定是哪一斜排
    for (int i = 1; i < cnt; i++) { // 把前i-1排的数都加到结果中
        num += i;
    }
    // 确定那一排是从上往斜下填还是从下往斜上填
    if (cnt % 2 == 0) {
        // 偶数是从右上往下填
        int row = 1;
        while (row <= m) {
            num++;
            row++;
        }
    } else {
        int row = cnt;
        while (row >= m) {
```

```
            num++;
            row--;
        }
    }
    cout << num << endl;
    return 0;
}
```

蛇形填数II

```
#include<iostream>
#include<cstring>
#include<cstdio>
using namespace std;

int a[110][110];
int main()
{
    int i,j,tot,x,y,m,n;
    cin>>m>>n;
    memset(a,0,sizeof(a));
    x=0;y=n-1;
    a[x][y]=1;
    tot=1;
    while(tot<m*n) // 顺时针填写数字
    {
        while(x+1<m&&!a[x+1][y]) a[++x][y]=++tot; // 右方一列从上到下
        while(y-1>=0&&!a[x][y-1]) a[x][--y]=++tot; // 下方一行从右到左
        while(x-1>=0&&!a[x-1][y]) a[--x][y]=++tot; // 左方一列从下到上
        while(y+1<=n&&!a[x][y+1]) a[x][++y]=++tot; // 上方一行从左到右
    }
    for(i=0;i<m;i++)
    {
      for(j=0;j<n;j++)
        printf("%5d",a[i][j]);
```

```
        cout<<endl;
    }
    cout<<endl;
    return 0;
}
```

循环枚举

特别数的和（蓝桥杯C/C++2019B组省赛）

```
#include <bits/stdc++.h>

using namespace std;

bool check(int x) {
    while (x > 0) {
        if (x % 10 <= 2 || x % 10 == 9)//数位分离判断每一位上是否含有2，0，1，9中的数字
            return true;
        x /= 10;//判断完当前这一位就去除掉这一位，比如2108 % 10 == 8，8这一位已经判断过了，2108/10=210
    }
    return false;
}

int main() {
    int n, sum = 0;
    cin >> n;

    for (int i = 1; i <= n; ++i) {//枚举所有数字
        if (check(i)) {//判断是否符合条件
            sum += i;
        }
    }

    cout << sum;
    return 0;
}
```

数的分解（蓝桥杯C/C++2019B组省赛）

```cpp
#include <bits/stdc++.h>

using namespace std;

// 函数用于判断一个数是否包含数字2或数字4
bool judge(int num) {
    while (num > 0) {
        int k = num % 10;
        if (k == 2 || k == 4) {
            return false;
        }
        num /= 10;
    }
    return true;
}

int main() {
    long result = 0;

    // 使用三重循环计算符合条件的组合数
    for (int i = 1; i < 2019; i++) {
        for (int j = 1; j < 2019; j++) {
            for (int k = 1; k < 2019; k++) {
                if (i + j + k == 2019) {
                    // 排除重复的组合
                    if (i != j && i != k && j != k) {
                        // 判断三个数是否满足要求
                        if (judge(i) && judge(j) && judge(k)) {
                            result++;
                        }
                    }
                }
            }
        }
    }
```

```
    // 结果除以6是因为每个组合会被计算6次（因为排列组合的原因）
    cout << result / 6 << endl;
    return 0;
}
```

利用规律减少枚举次数

乘积数量

解法1超时

- 直接三重循环枚举，看看有多少乘积符合条件的情况

```
#include<bits/stdc++.h>
using namespace std;

int main() {
    long long n, s = 0; // 声明变量n和累加器s，用于存储输入的数字和计算结果
    cin >> n; // 从标准输入读取一个数字，存入变量n中
    for (long long a = 1; a <= n; a++) { // 外层循环，遍历a从1到n的所有可能取值
        for (long long b = a; b <= n; b++) { // 第二层循环，遍历b从a到n的所有可能取值
            for (long long c = b; c <= n; c++) { // 第三层循环，遍历c从b到n的所有可能取值
                if (a * b * c <= n) { // 判断条件，如果a*b*c小于等于n
                    s++; // 累加器加一
                }
            }
        }
    }
    cout << s; // 将结果输出到标准输出
}
```

解法2优化循环次数

- 题目思路

1.首先第一层循环没有必要从 $1\sim n$，因为 $a\le b\le c$，所以如果 $a*a*a>n$，第一层循环就没有必要再继续进行了

2.对于第二层循环同样因为 $a\le b\le c$，如果 $a*b*b>n$，第二层循环也就没有必要再继续进行了

3.最后我们可以优化掉最内层的循环，最后的 c 的范围 $b\le c\le n/(a*b)$，c 的数量等于 $n/(a*b)-b+1$

- 代码

```
#include<bits/stdc++.h>
using namespace std;

int main() {
    long long n,s = 0;
    cin >> n;
    for(long long a = 1;a * a * a <= n;a++) {
        for(long long b = a;a * b * b <= n;b++) {
            if(b <= n / (a * b)) {//左边界必须小于右边界
                s += (n / (a * b) - b + 1);
            }
        }
    }
    cout << s;
}
```

递归

什么是递归

递归，在计算机科学中是指一种通过重复将问题分解为同类的子问题而解决问题的方法。简单来说，递归表现为函数调用函数本身。在知乎看到一个比喻递归的例子，个人觉得非常形象，大家看一下：

我们使用的词典，本身就是递归，为了解释一个词，需要使用更多的词。当你查一个词，发现这个词的解释中某个词仍然不懂，于是你开始查这第二个词，可惜，第二个词里仍然有不懂的词，于是查第三个词，这样查下去，直到有一个词的解释是你完全能看懂的，那么递归走到了尽头，然后你开始后退，逐个明白之前查过的每一个词，最终，你明白了最开始那个词的意思。

递归代码最重要的两个特征:结束条件和自我调用。自我调用是在解决子问题，而结束条件定义了最简子问题的答案。

```
int func(传入数值) {
  if (终止条件) return 最小子问题解;
  return func(缩小规模);
}

public int sum(int n) {
    if (n <= 1) {
        return 1;
    }
    return sum(n - 1) + n;
}
```

递归的特点

实际上，递归有两个显著的特征,终止条件和自身调用:

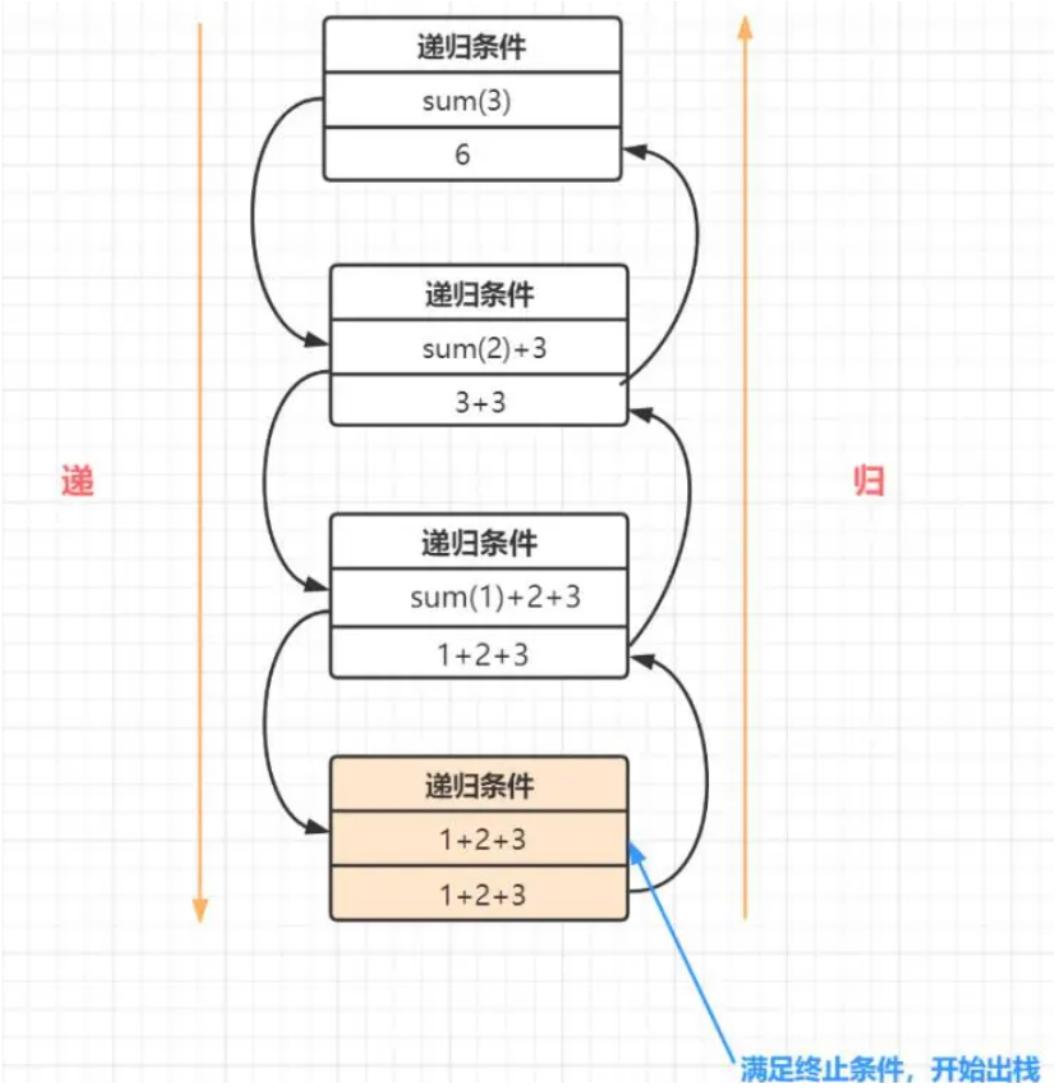
- 自身调用:原问题可以分解为子问题，子问题和原问题的求解方法是一致的，即都是调用自身的同一个函数。
- 终止条件:递归必须有一个终止的条件，即不能无限循环地调用本身。

结合以上demo代码例子，看下递归的特点:

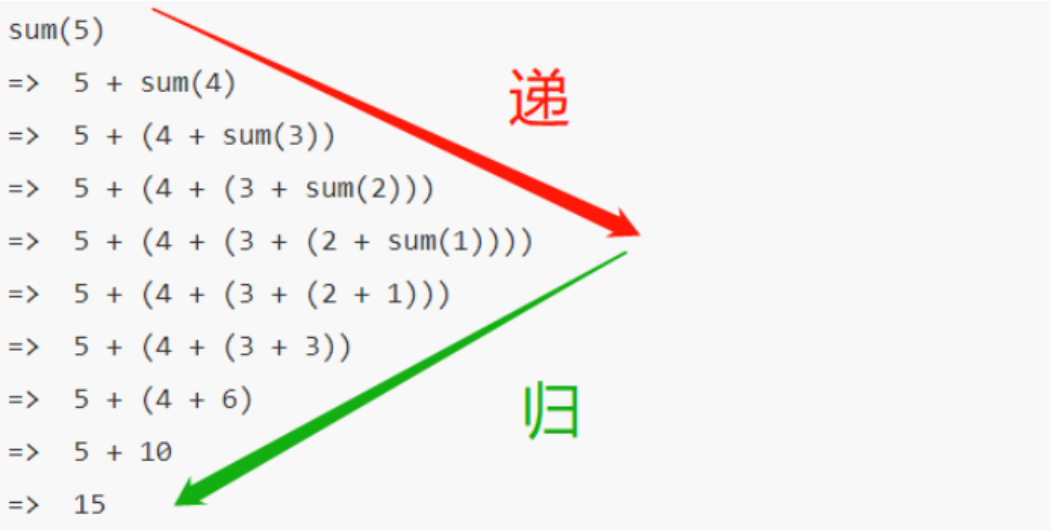
```
 int sum(int n) {
    if (n <= 1) {   // 终止条件
        return 1;
    }
    return sum(n - 1) + n;    // 调用自身
}
```

递归与栈的关系

其实，递归的过程，可以理解为出入栈的过程的，这个比喻呢，只是为了方便读者朋友更好理解递归哈。以上代码例子计算sum(n=3)的出入栈图如下：



为了更容易理解一些，我们来看一下 函数sum(n=5)的递归执行过程，如下：



递归解题思路

- 定义函数功能，就是说，你这个函数是干嘛的，做什么事情，换句话说，你要知道递归原问题是什么呀?比如你需要解决阶乘问题，定义的函数功能就是 n 的阶乘，如下:

```
//n的阶乘（n为大于0的自然数）
int factorial (int n)
{


}
```

- 递归的一个典型特征就是必须有一个终止的条件，即不能无限循环地调用本身。所以，用递归思路去解决问题的时候就需要寻找递归终止条件是什么。比如阶乘问题，当n=1的时候，不用再往下递归了，可以跳出循环啦，$n = 1$ 就可以作为递归的终止条件，如下:

```
//n的阶乘（n为大于0的自然数）
int factorial (int n)
{
    if(n==1)
    {
        return 1;
    }
}
```

- 递归的「本义」，就是原问题可以拆为同类且更容易解决的子问题，即「原问题和子问题都可以用同一个函数关系表示。递推函数的等价关系式，这个步骤就等价于寻找原问题与子问题的关系，如何用一个公式把这个函数表达清楚」阶乘的公式就可以表示为$f(n)=n*f(n-1)$,因此，阶乘的递归程序代码就可以写成这样，如下:

```
//n的阶乘（n为大于0的自然数）
int factorial (int n)
{
    if(n==1)
    {
        return 1;
    }

    return n * factorial(n - 1);
}
```

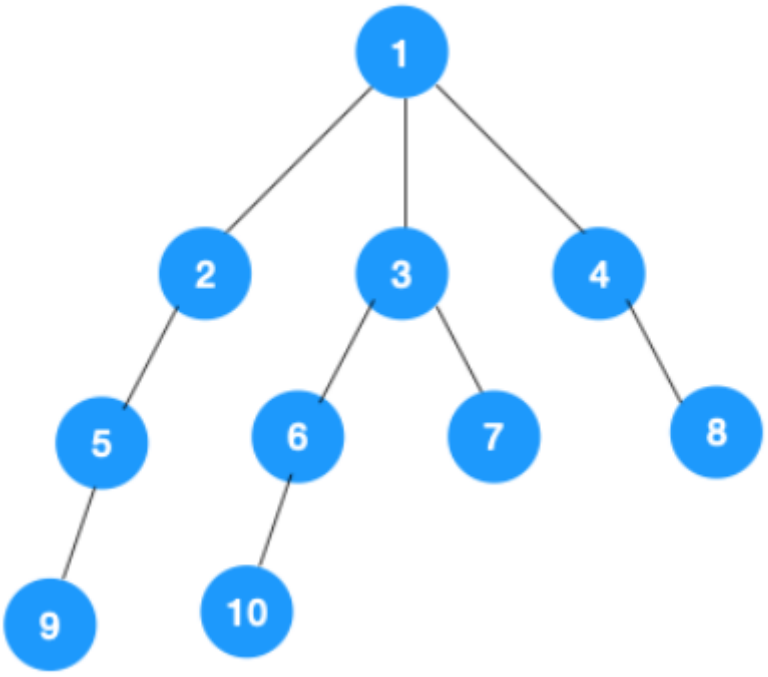
练习题

外星密码

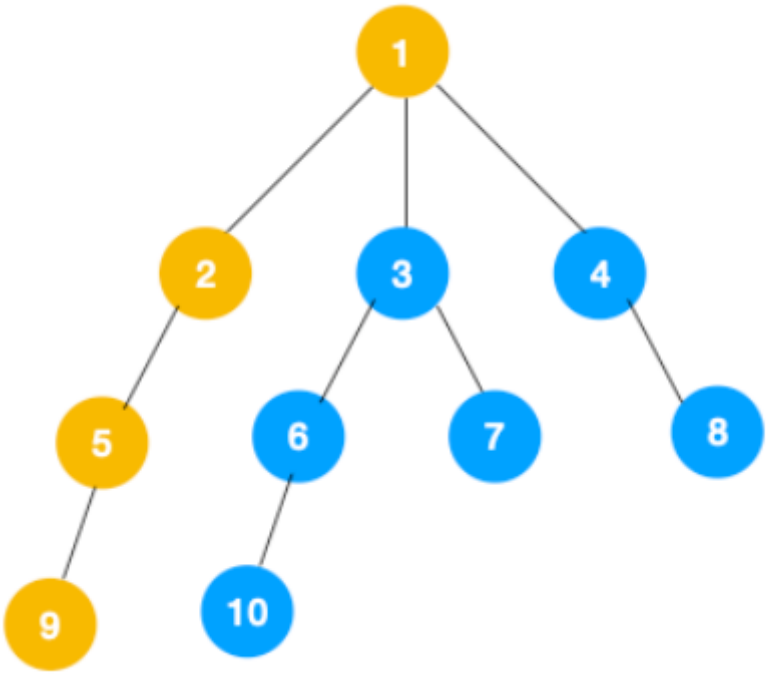
DFS

DFS讲解

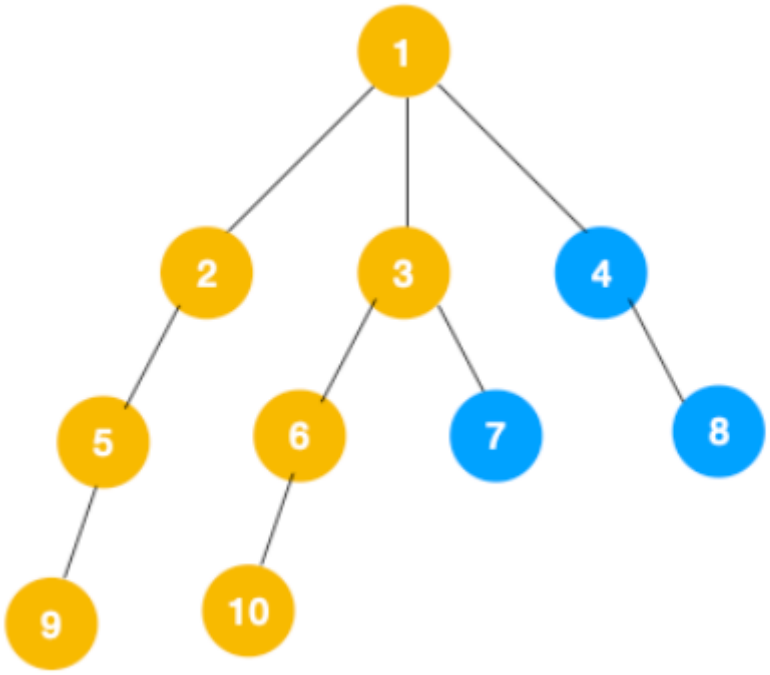
主要思路是从图中一个未访问的顶点 V 开始，沿着一条路一直走到底，然后从这条路尽头的节点回退到上一个节点，再从另一条路开始走到底....，不断递归重复此过程，直到所有的顶点都遍历完成，它的特点是不撞南墙不回头，先走完条路到底，再换一条路继续走。
树是图的一种特例(连通无环的图就是树)，接下来我们来看看树用深度优先遍历该怎么遍历。



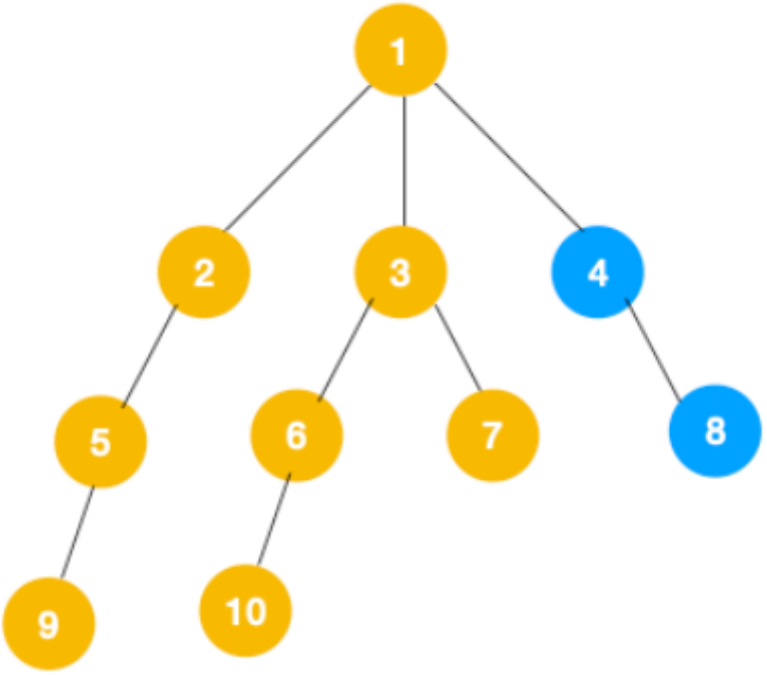
- 我们从根节点 1 开始遍历，它相邻的节点有 2，3，4，先遍历节点 2 ，再遍历 2 的子节点 5，然后再遍历 5 的子节点 9。



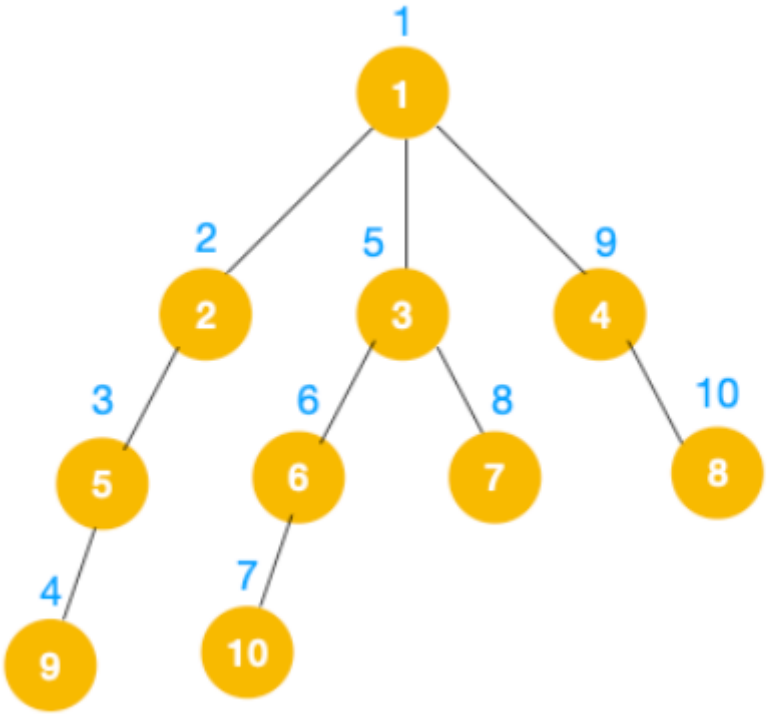
- 上图中一条路已经走到底了(9 是叶子节点，再无可遍历的节点)，此时就从 9 回退到上一个节点 5 ，看下节点 5 是否还有除 9 以外的节点，没有继续回退到 2，2 也没有除 5 以外的节点，回退到 1 ,1 有除 2 以外的节点 3， 所以从节点 3 开始进行深度优先遍历，如下:



- 同理从 10 开始往上回溯到 6, 6 没有除 10 以外的子节点，再往上回溯，发现 3 有除 6 以外的子点 7，所以此时会遍历 7。



- 从 7 往上回溯到 3，1，发现 1 还有节点 4 未遍历，所以此时沿着 4，8 进行遍历,这样就遍历完成了完整的节点的遍历顺序如下(节点上的的蓝色数字代表):



> **Important**
>
> 由于DFS不到返回条件不返回的特性，所以有的时候DFS并不是遍历到一个结果就是最优的。

全局状态变量

```
void dfs(当前状态)
{
    if(当前状态是目标状态)     // 判断
        进行相应处理（输出当前解、更新最优解、退出返回等）
    // 扩展
    for(所有可行的新状态)
    {
        if(新状态没有访问过 && 需要访问)  // 可行性剪枝、最优性剪枝、重复性剪枝
        {
            标记
            dfs(新状态);
            取消标记
        }
    }
}
```

```
int main()
{
    ...
    dfs(初始状态);
    ...
}
```

凑算式（蓝桥杯C/C++2016B组省赛）

解法1 DFS

- 首先考虑清楚递归边界，我们要求的是9个数字全排列，也就是目前已经要选到第十个数字的时候递归返回
- 然后 `for` 循环遍历

```
全局状态变量

void dfs(当前状态)
{
    if(当前状态是目标状态)     // 判断
        进行相应处理（输出当前解、更新最优解、退出返回等）
    // 扩展
    for(所有可行的新状态)
    {
        if(新状态没有访问过 && 需要访问)  // 可行性剪枝、最优性剪枝、重复性剪枝
        {
            标记
            dfs(新状态);
            取消标记
        }
    }
}

int main()
{
    ...
```

```
    dfs(初始状态);
    ...
}
```

- 代码

```
#include<bits/stdc++.h>
using namespace std;

bool st[15]; // 标记数组，标记数字是否已经被使用过
int num[15], ans; // 数组num存储排列的数字，ans记录满足条件的排列
数量

// 深度优先搜索函数，用于生成排列并判断是否满足条件
void dfs(int n){
    // 如果已经生成了一个排列
    if(n==10){
        // 计算出数字x和y
        int x = num[4] * 100 + num[5] * 10 + num[6];
        int y = num[7] * 100 + num[8] * 10 + num[9];
        // 判断是否满足条件
        if (num[1] * num[3] * y + num[2] * y + x * num[3] == 10
* num[3] * y){
            ans++; // 满足条件的排列数量加一
            return;
        }
    }
    // 遍历可能的数字
    for(int i = 1; i <= 9; ++i){
        // 如果数字已经被使用过，则跳过
        if(st[i]) continue;
        st[i] = 1; // 标记当前数字已经被使用
        num[n] = i; // 将当前数字加入排列中
        dfs(n+1); // 递归生成下一个位置的数字
        st[i] = 0; // 回溯，撤销当前数字的使用标记
    }
}
```

```
int main(){
    dfs(1); // 从第一个位置开始生成排列
    cout << ans; // 输出满足条件的排列数量
    return 0;
}
```

解法2 next_permutation()全排列

- 除了手写全排列外，C++还有 `next_permutation()` 求全排列

```
#include<bits/stdc++.h>
using namespace std;

int a[9] = {1,2,3,4,5,6,7,8,9}; // 存储排列的数组
int b, c, d, e; // 用于存储计算结果的变量
int ans; // 记录满足条件的排列数量

int main(){
    // 使用 while 循环代替 do-while 循环
    while(next_permutation(a, a+9)){ // 生成下一个排列，直到所有排列都被枚举过
        b = a[0]*a[2]*(a[6]*100+a[7]*10+a[8]); // 计算b的值
        c = a[1]*(a[6]*100+a[7]*10+a[8]); // 计算c的值
        d = (a[3]*100+a[4]*10+a[5])*a[2]; // 计算d的值
        e = 10*a[2]*(a[6]*100+a[7]*10+a[8]); // 计算e的值
        if(b+c+d==e) ans++; // 判断是否满足条件，如果满足条件则计数器加一
    }
    cout<<ans; // 输出满足条件的排列数量
    return 0;
}
```

迷宫类、Flood Fill问题（也可以用BFS再写一下）

全球变暖（蓝桥杯C/C++2018B组省赛）

```
#include<bits/stdc++.h>
using namespace std;
```

```
int n;
char a[1010][1010]; //地图
int vis[1010][1010]={0};  //标记是否搜过
int d[4][2] = {{0,1}, {0,-1}, {1,0}, {-1,0}}; //四个方向
int flag;  //用于标记这个岛中是否被完全淹没
void dfs(int x, int y){
    vis[x][y] = 1;      //标记这个'#'被搜过。注意为什么可以放在这
里
    if(a[x][y+1]=='#' && a[x][y-1]=='#' && a[x+1][y]=='#' &&
a[x-1][y]=='#')
        flag = 1;       //上下左右都是陆地，不会淹没
    for(int i = 0; i < 4; i++){ //继续DFS周围的陆地
        int nx = x + d[i][0], ny = y + d[i][1];
      //if(nx>=1 && nx<=n && ny>=1 && ny<=n && vis[nx][ny]==0
&& a[nx][ny]=='#') //题目说边上都是水，所以不用这么写了
        if(vis[nx][ny]==0 && a[nx][ny]=='#') //继续DFS未搜过的陆
地，目的是标记它们
            dfs(nx,ny);
    }
}

int main(){
    cin >> n;
    for(int i = 1; i <= n; i++)
        for(int j = 1; j <= n; j++)
            cin >> a[i][j];
    int ans = 0 ;
    for(int i = 1; i <= n; i++)  //DFS所有像素点
        for(int j = 1; j <= n; j++)
            if(a[i][j]=='#' && vis[i][j]==0){
                flag = 0;
                dfs(i,j);
                if(flag == 0)  //这个岛全部被淹
                    ans++;     //统计岛的数量
            }
    cout<<ans<<endl;
    return 0;
}
```

走迷宫输出路径

```
#include <bits/stdc++.h>
using namespace std;

int a[20][20];//存图
int n,m,qx,qy,zx,zy;
int dx[]={0,-1,0,1};
int dy[]={-1,0,1,0};

int rec[410][2];//存放路径
int cnt;

//向rec数组中第k层存放点(x,y)
void dfs(int x, int y, int k)
{
    //记录路径
    rec[k][0] = x;
    rec[k][1] = y;

    //判断是否到达终点
    if(x==zx&&y==zy)
    {
        cnt++;
        for(int i=1; i<k; i++)
        {
            printf("(%d,%d)->",rec[i][0], rec[i][1]);
        }
        printf("(%d,%d)\n",rec[k][0], rec[k][1]);
        return ;
    }

    //扩展新的结点
    for(int i=0; i<4; i++)
    {
        int nx = x+dx[i];
        int ny = y+dy[i];
        if(nx>=1&&nx<=n&&ny>=1&&ny<=m  &&  a[nx][ny]==1)
        {
```

```
            a[nx][ny] = 0;//已经走过，标记为不可通行
            dfs(nx, ny, k+1);
            a[nx][ny] = 1;//全部走完，取消标记
        }
    }
}

int main()
{
    cin>>n>>m;
    for(int i=1;i<=n;i++)
        for(int j=1;j<=m;j++)
            cin>>a[i][j];
    cin>>qx>>qy>>zx>>zy;

    a[qx][qy] = 0;//注意：起始点标记为已走过
    dfs(qx, qy, 1);

    if(cnt == 0) cout << -1;
}
```

排列枚举

全排列问题

```
#include<bits/stdc++.h>
using namespace std;
int n, a[10];// 存放全排列的结果
bool f[10];// 标记哪些数使用过
void print() {
    for (int i = 1; i < n; i++)
        cout << a[i] << " ";
    cout << a[n] << endl;
}
// 递归函数：为a数组每个元素赋值
void dfs(int k) {
    for (int i = 1; i <= n; i++) {
        if (f[i] == false) {    // 如果这个数i没有被用过
```

```
            a[k] = i;            // 则把i填到a[k]位置
            f[i] = true;         // 标记i已选择
            if (k == n) print();//如果已经存了n个数，输出结果
            else dfs(k + 1);     //否则，继续递归
            f[i] = false;        //取消标记i
        }
    }
}
int main() {
    cin >> n;
    dfs(1);// 递归从a[1]开始赋值
}
```

组合枚举

部分元素排列

```
#include<bits/stdc++.h>
using namespace std;

int a[20], b[20], n, r; // 数组a用于存储输入的元素，数组b用于存储当前排列，n表示总元素数量，r表示选取元素数量

// 打印函数，用于输出当前排列
void print(int m)
{
    for(int i=1; i<=m; i++)
    {
        cout<<b[i]<<" "; // 输出当前排列中的元素
    }
    cout<<endl; // 换行
    return ;
}

// 深度优先搜索函数，用于生成排列
void dfs(int step, int pos)
{
    if(step == r + 1) // 如果已经选取了r个元素
```

```
    {
        print(r); // 输出当前排列
        return ;
    }
    else
    {
        for(int i = pos + 1; i <= n; i++) // 从pos+1开始选取下一个元素，保证排列中的元素不重复
        {
            b[step] = a[i]; // 将当前选取的元素加入排列中
            dfs(step + 1, i); // 递归选取下一个元素，注意第step+1个元素的选取位置为i
        }
    }
}

int main()
{
    cin >> n >> r; // 输入总元素数量n和选取元素数量r
    for(int i = 1; i <= n; i++) // 输入元素值
    {
        cin >> a[i];
    }
    sort(a + 1, a + 1 + n); // 对输入的元素值进行排序
    dfs(1, 0); // 从第一个位置开始生成排列，初始选取位置为0
    return 0;
}
```

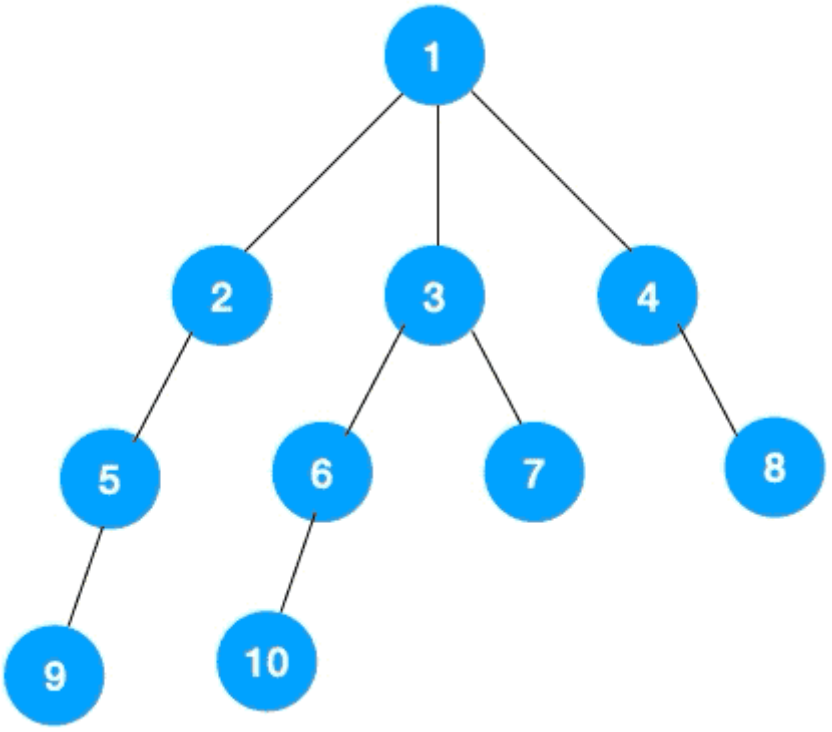
练习题

飞机降落（蓝桥杯C/C++2023B组省赛）

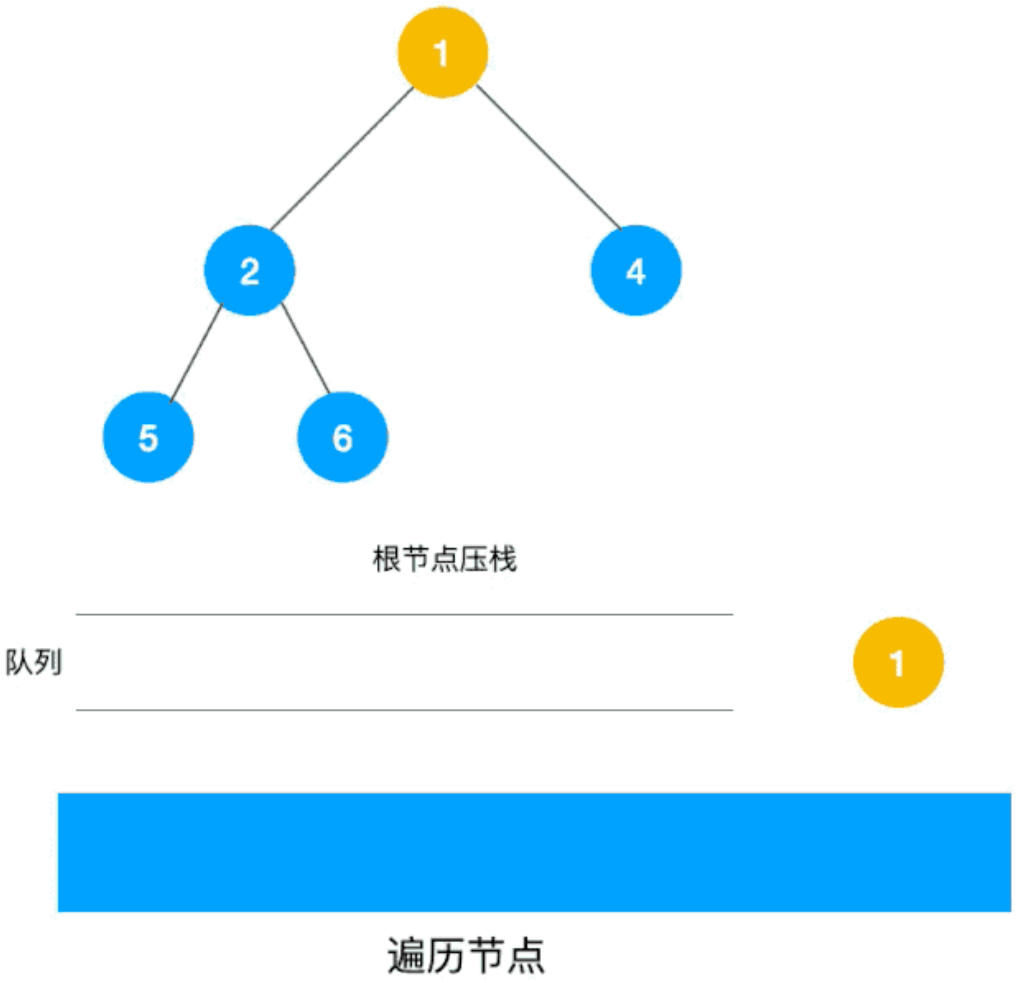
BFS

BFS讲解

广度优先遍历，指的是从图的一个未遍历的节点出发，先遍历这个节点的相邻节点，再依次遍历每个相邻节点的相邻节点。
上文所述树的广度优先遍历动图如下，每个节点的值即为它们的遍历顺序。所以广度优先遍历也叫层序遍历，先遍历第层(节点 1)，再遍历第二层(节点 2，3，4)，第三层(5，6，7，8)，第四层(9，10)。



深度优先遍历用的是栈，而广度优先遍历要用队列来实现，我们以下图二叉树为例来看看如何用队列来实现广度优先遍历。



> **Important**
>
> 由于 BFS 按层级逐步探索图，因此当目标节点在起始节点的相邻节点之一时，BFS 将最快找到该目标节点。这是因为 BFS 会首先探索所有距离起始节点为 1 的节点，然后是距离为 2 的节点，以此类推。因此，当目标节点位于距离起始节点最近的层级时，BFS 将会首先到达该目标节点。也就是可以认为BFS最先到达的节点是当前BFS条件下的最优解。

全局状态变量

```
void BFS()
{
    定义状态队列
    初始状态入队
    while(队列不为空)
    {
        取出队首状态作为当前状态
        if(当前状态是目标状态)
```

```
            进行相应处理（输出当前解、更新最优解、退出返回等）
        else
            for(所有可行的新状态)
            {
                if(新状态没有访问过 && 需要访问)
                {
                    新状态入队
                }
            }
    }
}
```

岛屿个数（蓝桥杯C/C++2023B组省赛）

```
#include <bits/stdc++.h>
using namespace std;

const int N = 55;
#define pii pair<int, int>
#define ft first
#define sd second

int dx[8] = {-1, 0, 1, 0, -1, -1, 1, 1}; // 方向数组，用于表示8个方向的横坐标变化
int dy[8] = {0, 1, 0, -1, -1, 1, 1, -1}; // 方向数组，用于表示8个方向的纵坐标变化

char g[N][N]; // 存储地图信息的二维数组
int t[N][N], st[N][N]; // 标记数组，用于标记已访问过的点
int n, m, res = 0; // n和m分别表示地图的行数和列数，res用于记录满足条件的连通块数目

// BFS函数，用于搜索一个连通块
void bfs(int i, int j) {
    queue<pii> q; // 定义一个队列，用于存储待访问的点
    q.push({i, j}); // 将起始点加入队列
    while (!q.empty()) { // 当队列不为空时循环
```

```
        int x = q.front().ft, y = q.front().sd; // 取出队首元素的坐标
        q.pop(); // 弹出队首元素
        if (t[x][y]) continue; // 如果该点已经访问过，则跳过
        t[x][y] = true; // 标记该点为已访问
        for (int i = 0; i < 4; i++) { // 遍历该点的上下左右四个方向
            int xx = x + dx[i], yy = y + dy[i]; // 计算下一个点的坐标
            if (t[xx][yy] || xx < 1 || xx > n || yy < 1 || yy > m || g[xx][yy] == '0') continue; // 如果下一个点已经访问过、越界或者是海洋，则跳过
            q.push({xx, yy}); // 将下一个点加入队列
        }
    }
}

// 检查函数，用于检查一个连通块是否与边界相连
int check(int i, int j) {
    queue<pii> q; // 定义一个队列，用于存储待访问的点
    q.push({i, j}); // 将起始点加入队列
    while (!q.empty()) { // 当队列不为空时循环
        int x = q.front().ft, y = q.front().sd; // 取出队首元素的坐标
        q.pop(); // 弹出队首元素
        if (st[x][y]) continue; // 如果该点已经访问过，则跳过
        st[x][y] = true; // 标记该点为已访问
        if (x == 1 || x == n || y == 1 || y == m) return true; // 如果该点在边界上，则返回true
        for (int i = 0; i < 8; i++) { // 遍历该点的八个方向
            int xx = x + dx[i], yy = y + dy[i]; // 计算下一个点的坐标
            if (st[xx][yy] || g[xx][yy] == '1') continue; // 如果下一个点已经访问过或者是陆地，则跳过
            q.push({xx, yy}); // 将下一个点加入队列
        }
    }
    return false; // 如果没有与边界相连的点，则返回false
}
```

```
// 主函数，用于读入数据并调用处理函数
void solve() {
    memset(t, 0, sizeof t); // 初始化标记数组
    res = 0; // 初始化结果变量
    cin >> n >> m; // 读入地图的行数和列数
    for (int i = 1; i <= n; i++) // 循环读入地图信息
        for (int j = 1; j <= m; j++)
            cin >> g[i][j];

    for (int i = 1; i <= n; i++) // 遍历地图
        for (int j = 1; j <= m; j++)
            if (!t[i][j] && g[i][j] == '1') { // 如果当前点未访问过且是陆地
                bfs(i, j); // 对当前连通块进行BFS搜索
                memset(st, 0, sizeof st); // 初始化边界标记数组
                if (check(i, j)) res++; // 检查当前连通块是否与边界相连，并更新结果
            }
    cout << res << endl; // 输出结果
}

int main() {
    int T;
    cin >> T; // 读入测试用例数量
    while (T--) solve(); // 循环处理每个测试用例
    return 0;
}
```

奇怪的电梯

```
#include <bits/stdc++.h>
using namespace std;

int n, a, b; // 输入的数值范围n，起点a，终点b
int x[205], step[205];   // 数组x用于存储每个位置可以跳跃的步数，step数组用于标记每个位置的步数
queue<int> q; // 队列用于广度优先搜索
```

```
// 广度优先搜索函数
void bfs() {
    q.push(a); // 将起点a加入队列
    step[a] = 0; // 将起点的步数标记为0
    while (!q.empty()) { // 当队列不为空时循环
        int t = q.front(); // 取出队首元素
        q.pop(); // 弹出队首元素
        if (t == b) { // 如果当前位置等于终点b
            cout << step[b]; // 输出步数
            return ; // 结束搜索
        }
        if (t - x[t] >= 1 && !step[t - x[t]]) { // 如果向下跳不越界且下一个位置未被访问过
            q.push(t - x[t]); // 将下一个位置加入队列
            step[t - x[t]] = step[t] + 1; // 更新下一个位置的步数
        }
        if (t + x[t] <= n && !step[t + x[t]]) { // 如果向上跳不越界且下一个位置未被访问过
            q.push(t + x[t]); // 将下一个位置加入队列
            step[t + x[t]] = step[t] + 1; // 更新下一个位置的步数
        }
    }
    cout << -1; // 如果无法到达终点b，输出-1
}

int main()
{
    cin >> n >> a >> b; // 输入数值范围n，起点a，终点b
    for (int i = 1; i <= n; i++) // 输入每个位置的可跳跃步数
        cin >> x[i];
    bfs(); // 进行广度优先搜索
}
```

练习题

迷宫（蓝桥杯C/C++2019B组省赛）

搜索剪枝

剪枝，顾名思义，就是通过一些判断，砍掉搜索树上不必要的子树。有时候，我们会发现某个结点对应的子树的状态都不是我们要的结果，那么我们其实没必要对这个分支进行搜索，砍掉这个子树，就是剪枝。最常用的剪枝有三种，可行性剪枝、重复性剪枝、最优性剪枝。

- 可行性剪枝:在搜索过程中，一旦发现如果某些状态无论如何都不能找到最终的解，就可以将其"剪枝"了，比如越界操作、非法操作。一般通过条件判断来实现，如果新的状态节点是非法的，则不扩展该节点。
- 重复性剪枝:对于某一些特定的搜索方式，一个方案可能会被搜索很多次，这样是没必要的。在实现上，一般通过一个记忆数组来记录搜索到目前为止哪些状态已经被搜过了，然后在搜索过程中，如果新的状态已经被搜过了，则不再扩展该状态节点。
- 最优性剪枝:对于求最优解的一类问题，通常可以用最优性剪枝，比如在求解迷宫最短路的时候，如果发现当前的步数已经超过了当前最优解，那从当前状态开始的搜索都是多余的，因为这样搜索下去永远都搜不到更优的解。通过这样的剪枝，可以省去大量兄余的计算，避免超时。在实现上，一般通过一个记忆数组来记录搜索到目前为止的最优解，然后在搜索过程中，如果新的状态已经不可能是最优解了，那再往下搜索肯定搜不到最优解，于是不再扩展该状态节点。

海贼王之伟大航路

```
#include<cstdio>
#include<string.h>
#include<stdlib.h>
#include<math.h>

const int inf = 0x3f3f3f3f; // 无穷大值

int step; // 当前步数
```

```
int minstep; // 最小步数
int a[16][16]; // 存储距离矩阵
int book[15]; // 标记数组，标记节点是否被访问过
int postion[20]; // 用于计算状态压缩的数组
int n; // 节点数量
int npostion[15][1<<15]; // 用于存储不同状态下的最小步数
int po; // 当前状态

// 深度优先搜索函数
void dfs(int cur,int qc)
{
    // 如果当前节点是倒数第二个节点
    if(cur+2==n)
    {
        step+=a[qc][n-1]; // 将当前节点到终点的距离加入步数
        if(step<minstep) // 如果当前步数小于最小步数
            minstep=step; // 更新最小步数
        step-=a[qc][n-1]; // 回溯，将当前节点到终点的距离从步数中减去
        return;
    }
    // 遍历除起点和终点外的所有节点
    for(int i=1; i<n-1; i++)
    {
        // 如果节点未被访问过
        if(!book[i])
        {
            // 如果当前步数已经大于等于最小步数，直接跳过该节点
            if(step>=minstep)
                continue;
            // 如果通过当前节点到达终点的最小步数已经大于等于当前步数加上从起点到当前节点的距离
            if((npostion[i][po+postion[i-1]]<=step+a[qc][i]))
                continue; // 直接跳过该节点
            // 更新状态
            po+=postion[i-1];
            step+=a[qc][i];
            npostion[i][po]=step;
            book[i]=1; // 标记当前节点为已访问
```

```
            dfs(cur+1,i); // 递归搜索下一个节点
            // 回溯
            po-=postion[i-1];
            book[i]=0;
            step-=a[qc][i];
        }
    }
}

int main()
{
    // 初始化状态压缩数组
    for(int i=0; i<14; i++)
        postion[i]=1<<i;
    // 读入节点数量
    while (~scanf("%d",&n))
    {
        // 初始化
        po=0;
        step=0;
        minstep=1e17;
        memset(npostion,inf,sizeof(npostion)); // 初始化状态压缩
数组
        memset(a,0,sizeof(a)); // 初始化距离矩阵
        memset(book,0,sizeof(book)); // 初始化标记数组
        // 读入距离矩阵
        for(int i=0; i<n; i++)
            for(int j=0; j<n; j++)
                scanf("%d",&a[i][j]);
        // 深度优先搜索
        dfs(0,0);
        printf("%d\n",minstep); // 输出最小步数
    }
    return 0;
}
```

小红与字符串矩阵

```cpp
#include <bits/stdc++.h>
using namespace std;

int n, m, ans; // n和m分别表示矩阵的行数和列数，ans用于记录满足条件的路径数量
const int N = 1005;
char a[N][N]; // 存储地图信息的二维数组
int dx[] = {-1, 1, 0, 0}; // 方向数组，用于表示上下左右四个方向的横坐标变化
int dy[] = {0, 0, -1, 1}; // 方向数组，用于表示上下左右四个方向的纵坐标变化
string tencent = "tencent"; // 目标字符串

struct point // 定义结构体表示点的坐标和当前字符串
{
    int x, y; // 横纵坐标
    string s; // 当前字符串
};

// 宽度优先搜索函数，用于搜索从指定点出发的路径
void bfs(int x, int y)
{
    queue<point> q; // 定义一个队列，用于存储待访问的点
    string tmp = ""; // 初始化当前字符串为空字符串
    tmp.push_back(a[x][y]); // 将起点的字符加入当前字符串
    q.push(point{x, y, tmp}); // 将起点加入队列

    while (q.size()) // 当队列不为空时循环
    {
        point p = q.front(); // 取出队首元素
        q.pop(); // 弹出队首元素
        string str = p.s; // 取出当前字符串
        if (str == "tencent") // 如果当前字符串与目标字符串相同
            ans++; // 满足条件的路径数量加一
        if (str.size() == 7) // 如果当前字符串长度为7，继续下一个循环
```

```
            continue;
        int flag = 0; // 标志变量，用于判断当前字符串是否与目标字符串不同
        for (int i = 0; i < str.size(); i++) // 遍历当前字符串
            if (str[i] != tencent[i]) // 如果当前字符与目标字符串的对应字符不同
            {
                flag = 1; // 设置标志为1
                break; // 结束循环
            }
        if (flag) // 如果当前字符串与目标字符串不同，继续下一个循环
            continue;
        int x = p.x; // 取出当前点的横坐标
        int y = p.y; // 取出当前点的纵坐标
        for (int i = 0; i < 4; i++) // 遍历四个方向
        {
            int xx = x + dx[i]; // 计算下一个点的横坐标
            int yy = y + dy[i]; // 计算下一个点的纵坐标
            if (xx >= 1 && xx <= n && yy >= 1 && yy <= m) // 如果下一个点在地图范围内
                q.push(point{xx, yy, str + a[xx][yy]}); // 将下一个点加入队列，并更新当前字符串
        }
    }
}

int main()
{
    cin >> n >> m; // 输入矩阵的行数和列数
    for (int i = 1; i <= n; i++) // 遍历输入矩阵的每一行
        for (int j = 1; j <= m; j++)
            cin >> a[i][j]; // 输入矩阵的每个元素

    for (int i = 1; i <= n; i++) // 遍历矩阵的每一个元素
        for (int j = 1; j <= m; j++)
            bfs(i, j); // 从当前点开始进行宽度优先搜索

    cout << ans; // 输出满足条件的路径数量
}
```

记忆化搜索

每次搜索前先判断是否搜索过:
1.如果搜过，就直接返回之前存储的结果
2.否则搜索，并在返回之前将当前状态保存下来

```
int f[][][];

int dfs(状态列表)
{
    if(f[][][]) return f[][][];    // 避免重复搜索
    if(边界状态) {
        return f[][][] = ......;
    }
    int ans = ...;
    for(新状态) {
        ans = max/min(dfs(...), ans);
    }
    return f[][][] = ans;     // 返回前先保存
}
```

数的计算

- 超时暴搜

```
//超时只能通过30%数据
#include<bits/stdc++.h>
using namespace std;

int sum = 1; // 初始化sum为1，用于记录结果

// 递归函数f，用于计算结果
void f(int x) {
    if(x / 2 == 0) { // 如果x除以2等于0，即x为偶数且不大于1，递归
终止条件
        return;
    }
```

```
    for(int i = 1; i <= x / 2; i++) { // 对x的一半进行遍历
        sum++; // 每次遍历sum加1，记录结果
        f(i); // 对当前的i进行递归调用
    }
}

int main() {
     int n;
     cin >> n; // 输入一个整数n
     f(n); // 调用递归函数f，计算结果
     cout << sum; // 输出结果
}
```

- 记忆化搜索

```
#include<bits/stdc++.h>
using namespace std;

int dp[1010]; // 记忆化数组，用于存储中间结果

// 递归函数f，计算结果并使用记忆化搜索
int f(int x) {
    if(dp[x]) return dp[x]; // 如果已经计算过dp[x]，直接返回结果
    int s = 1; // 初始化s为1，用于累加计算结果
    for(int i = 1; i <= x / 2; i++) { // 对x的一半进行遍历
        s = s + f(i); // 递归调用f函数，累加计算结果
    }
    dp[x] = s; // 将计算结果存储到记忆化数组中，避免重复计算
    return s; // 返回计算结果
}

int main() {
     int n;
     cin >> n; // 输入一个整数n
     cout << f(n); // 调用递归函数f，计算结果并输出
}
```

走方格（蓝桥杯C/C++2020B组省赛第一场）

- 超时暴搜

```
#include<bits/stdc++.h>

using namespace std;

int n, m, cnt; // 定义全局变量，分别表示行数n、列数m和路径数量cnt
const int N = 1e3 + 1; // 定义常量N，表示数组大小上限
int dx[2] = {1, 0}, dy[2] = {0, 1}; // 定义数组，表示向下和向右的移动方向

// 深度优先搜索函数，用于计算从左上角到右下角的路径数量
void dfs(int x, int y) {
    // 如果当前位置为右下角，路径数量加1并返回
    if (x == n && y == m) {
        cnt++;
        return;
    }
    // 遍历向下和向右的两个方向
    for (int i = 0; i < 2; i++) {
        int a = x + dx[i], b = y + dy[i]; // 计算下一个位置的坐标
        // 如果下一个位置越界或者是偶数行偶数列，则跳过当前方向
        if (a > n || b > m || (a % 2 == 0 && b % 2 == 0))
            continue;
        dfs(a, b); // 递归搜索下一个位置
    }
}

int main() {
    cin >> n >> m; // 输入行数和列数
    if (n % 2 == 0 && m % 2 == 0) {
        cnt = 0; // 如果行数和列数都是偶数，则路径数量为0
    } else {
        dfs(1, 1); // 否则从左上角开始搜索路径
    }
    cout << cnt << endl; // 输出路径数量
    return 0;
```

```
}
```

- 记忆化搜索

```
#include<bits/stdc++.h>

using namespace std;

const int N = 40;
int f[N][N]; // f[i][j]: 表示该点是否已经走过了!

int dfs(int x, int y) {
    // 如果x或y为奇数，表示当前点为需要考虑的点
    if (x & 1 || y & 1) {
        // 如果当前点已经访问过，直接返回该点的值
        if (f[x][y])
            return f[x][y];
        // 如果当前点未访问过，递归求解下一步可到达的点，并将结果累加到当前点上
        if (x < n)
            f[x][y] += dfs(x + 1, y);
        if (y < m)
            f[x][y] += dfs(x, y + 1);
    }
    return f[x][y];
}

int main() {
    cin >> n >> m; // 输入矩阵的行数n和列数m
    // 如果n和m中有奇数，则可以从右下角到左上角的路径总数为奇数，否则为偶数
    f[n][m] = n & 1 || m & 1;
    cout << dfs(1, 1); // 计算从左上角到右下角的路径总数并输出
    return 0;
}
```