中国人民大学 信息学院
RENMIN UNIVERSITY OF CHINA SCHOOL OF INFORMATION

程序设计荣誉课程

14. 算法4——搜索

授课教师：游伟 副教授、孙亚辉 副教授

授课时间：周二14:00 – 15:30，周四10:00 – 11:30（教学三楼3304）

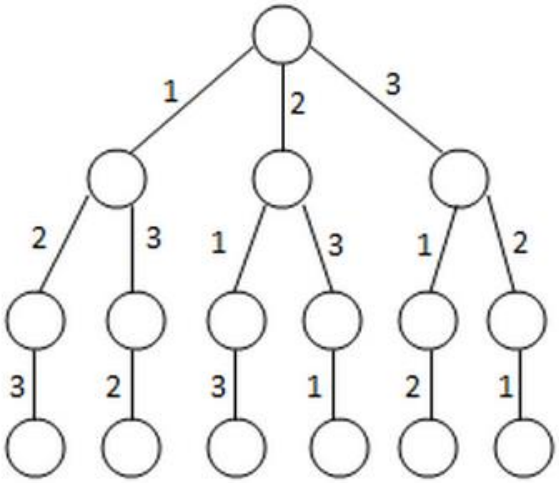
上机时间：周二18:00 – 21:00（理工配楼二层机房）

课程主页：https://www.youwei.site/course/programming

引子：全排列

- 从n个不同元素中任取m（m≤n）个元素，按照一定的顺序排列起来，叫做从n个不同元素中取出m个元素的一个排列。
- 当 m = n 时所有的排列情况叫全排列。
- 例如{1,2,3}的全排列：

1 2 3　　1 3 2　　2 1 3　　2 3 1　　3 1 2　　3 2 1



引子：全排列

- 利用穷举法求3个数的全排列

```
1.  #include <stdio.h>

2.  void main() {
3.    int i, j, k;
4.    for (i = 1; i <= 3; i++)
5.      for (j = 1; j <= 3; j++)
6.        for (k = 1; k <= 3; k++)
7.          if (i != j != k)          i != j && i != k && j != k
8.            printf("%d %d %d\n", i, j, k);
9.  }
```

- 如何求n个数的全排列?

  本质：穷举n个变量所有的取值可能（条件：互不相等）

目录

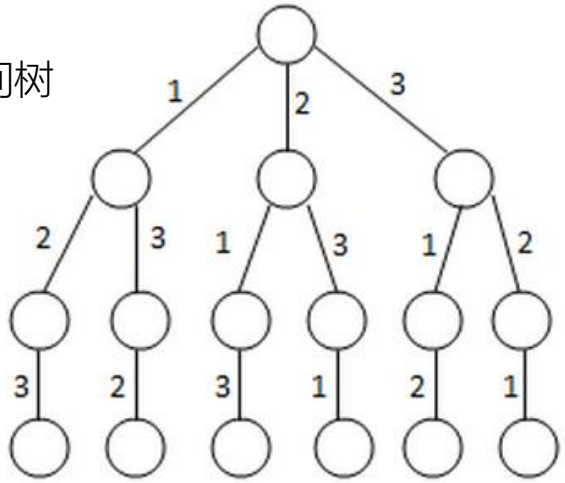
14.1 搜索问题概述

- 解空间
  - 定义：所有可能的解组成的空间
  - 规范化：将问题的解规范为一个n元组$\{x_1, x_2, \ldots x_n\}$
  - 解空间越小，搜索效率越高
- 问题的解
  - 可行解：满足约束条件的n元组
  - 最优解：在所有可行解中，找到一个使得评估函数值最优的解
- 约束条件
  - 显约束：对解分量的取值范围的限定
  - 隐约束：对能否得到问题的可行解或最优解做出的约束
    (分别称为可行性隐约束和最优化隐约束)

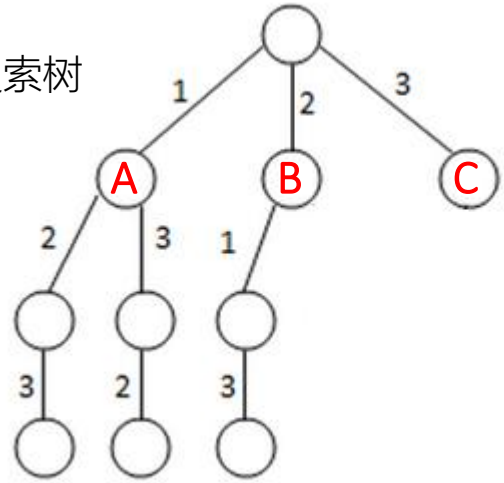
14.1 搜索问题概述

- 解空间树与搜索树
  - 解空间树：依据待解决问题的特性，用树结构表示问题的解空间结构
  - 搜索树：遍历解空间树过程中产生的子树
- 结点
  - 扩展结点：正在生成孩子的结点（如B结点）
  - 活结点：自身已生成，但孩子还没有全部生成的结点（如C结点）
  - 死结点：所有孩子都已经生成的结点（如A结点）
  - 子孙结点：结点B的子树上所有结点都是B的子孙
  - 祖先结点：从结点B到树根路径上的所有结点都是B的祖先

解空间树



搜索树

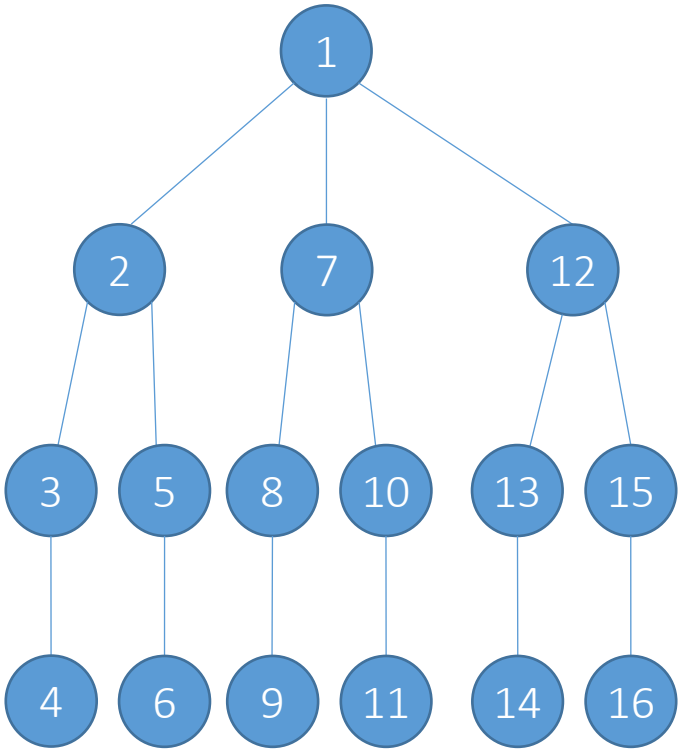


14.1 搜索问题概述

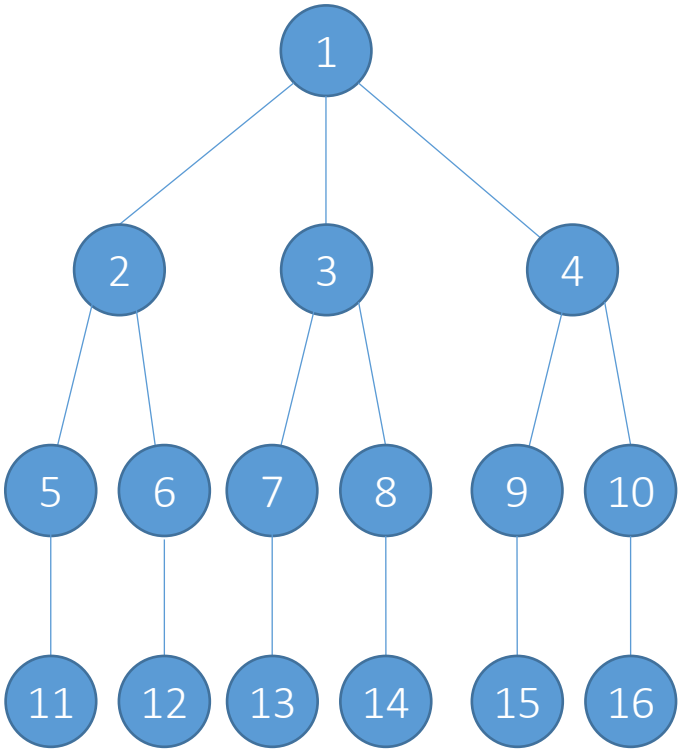
- 状态：用于描述问题求解过程中关键变量的取值
  - 初始状态：初始时的状态
  - 目标状态：求得问题解的状态
  - 搜索：将问题求解过程表现为从初始状态到目标状态的寻找过程
- 状态转移：在显约束范围内尝试产生新的状态
  - 确保可行性：确保新产生的状态符合可行性隐约束
  - 确保最优化：确保新产生的状态符合最优化隐约束
  - 避免重复搜索：去重/记忆化搜索
- 状态共享：在不同结点间共享状态信息
  - 通常是父子结点之间共享部分状态信息
  - 父结点->子结点：设置增量状态信息
  - 子结点->父结点：撤销增量状态信息

14.1 搜索问题概述

- 搜索策略：产生新状态的系统化方式



深度优先搜索



广度优先搜索

14.2 深度优先搜索（递归回溯法）

- 基本思想
  - 根据显约束遍历解空间，每次探索一个孩子结点
  - 根据隐约束确定是否深入递归
  - 当发现当前状态不满足求解条件时，就回溯尝试其他的路径
  - 回溯时需要恢复原先的状态
- 例题
  - 八皇后：可行性隐约束
  - 机器零件加工：可行性隐约束 + 最优化隐约束

思考：从父结点生成子结点时，增量更改了ary数组；
为何从子结点返回时，没有显式撤销更改的增量？

14.2.1 全排列的深度优先搜索算法

- 先不考虑条件（互不相等）
  - 定义状态：
    - k：当前正在穷举的变量
    - ary：存放每个变量的取值（在父子结点间共享）
  - 定义递归函数dfs (int k)：
    - 递归边界：k > n
    - 给第 k 个变量尝试各种可能的取值，取值范围1~n

```
1.  #include <stdio.h>
2.  #define N 101

3.  int ary[N], int n;

4.  void output() {
5.    int i;
6.    for (i = 1; i <= n; i++) printf("%d ", ary[i]);
7.    printf("\n");
8.  }

9.  void main() {
10.   scanf("%d", &n);
11.   dfs(1);
12. }
```

```
13. void dfs(int k) {
14.   int i;

15.   if (k > n) {          //边界条件：n个变量都穷举完了
16.     output();           //输出一组解
17.     return;
18.   }

19.   for (i=1; i<=n; i++) {     //n个可能的取值
20.     ary[k] = i;              //第k变量取值i
21.     dfs(k+1);                //穷举第k+1个变量
22. }
```

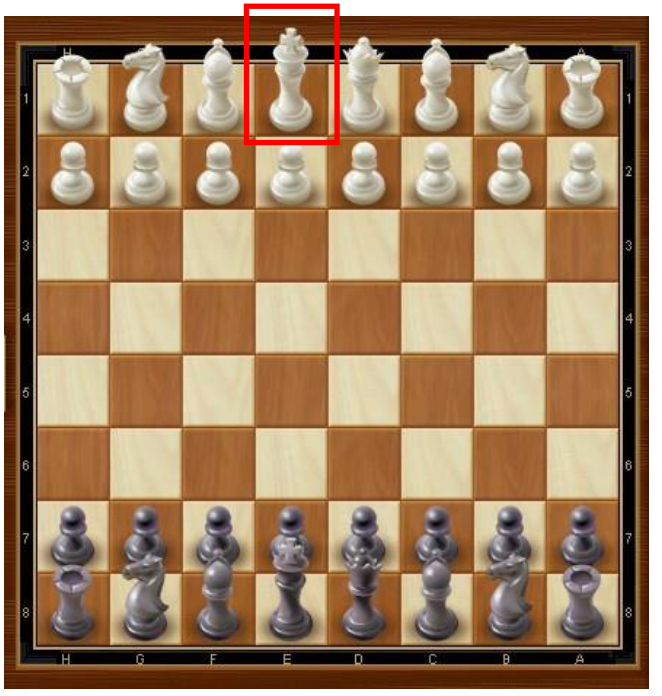
14.2.1 全排列的深度优先搜索算法

- 考虑条件（互不相等）
  - 定义状态：
    - k：当前正在穷举的变量
    - ary：存放每个变量的取值（在父子结点间共享）
    - used：记录每个数值是否用过（在父子结点间共享）
  - 定义函数： dfs (int k)
    - 递归边界：k > n
    - 给第 k 个变量尝试未被用过的值
    - 进入新状态前设置used
    - 从新状态返回后恢复used

```
1.   int ary[N], used[N], n;

2.   void dfs(int k) {
3.     int i;

4.     if (k > n) {          //边界条件：n个变量都穷举完了
5.       output();  //输出一组解
6.       return;
7.     }

8.     for (i=1; i<=n; i++) {        //n个可能的取值
9.       if (used[i]) continue;      //判断i是否用过
10.      ary[k] = i;                 //第k变量取值i
11.      used[i] = 1;                //标志i用过了
12.      dfs(k+1);    //穷举第k+1个变量
13.      used[i] = 0;                //恢复i未用过
14.  }
```

14.2.2 八皇后

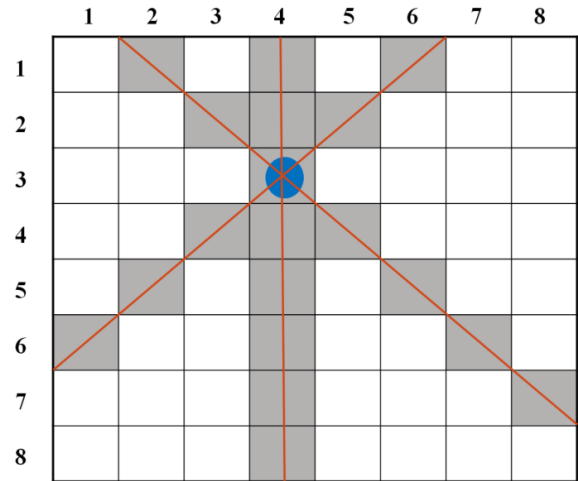
- 在8×8的棋盘上，放置8个皇后（棋子），使两两之间互不攻击。
- 所谓互不攻击是说任何两个皇后都要满足：
  - 不在棋盘的同一行
  - 不在棋盘的同一列
  - 不在棋盘的同一对角线上
- 因此可以推论出，棋盘共有8行，
  每行有且仅有一个皇后，故至多有8个皇后
- 这8个皇后每个应该放在哪一列上？



思考：为何不像全排列那样，用一个true/false的二值作为safe标志？

14.2.2 八皇后

- 解题思路：用试探的方法“向前走，碰壁回头” （回溯法）
  - 状态的描述：
    - i：当前正在放置第 i 行的皇后
    - 一维数组queen用于记录皇后摆放的位置
      queen[i]的值代表第 i 行皇后放在哪一列
    - 二维数组safe记录棋盘上放置皇后的冲突情况
      safe[i][j]的值代表第 i 行第 j 列和前面已放置的皇后产生多少个冲突
      safe[i][j] == 0代表第 i 行第 j 列可以放置皇后（和前面的皇后不冲突）
  - 定义递归函数dfs (int i)
    - 递归边界：i > 8
    - 尝试在第 i 行第 j 列放置皇后
      - 如果safe[i][j] == 0，则可以放置
      - 放置皇后之后，把所在列/左对角线/右对角线的冲突数加1
      - 回溯时，相应列/ /左对角线/右对角线的冲突数减1



14.2.2 八皇后

思考：

1. 数组queen和safe是否可以设置成局部变量？
2. 整型变量cnt是否可以设置成局部变量？
3. 设置成局部变量，需要做什么额外操作？

```
1. #include <stdio.h>
2. #define N 9

3. int queen[N];
4. int safe[N][N];
5. int nTotal, n = 8;

6. void main() {
7.   initialize();
8.   dfs(1);
9.   printf("nTotal: %d\n", nTotal);
10.}

11.void dfs(int i) {
12.  int j;
13.  if (i > n) {                //递归边界
14.    output();
15.    nTotal++;
16.    return;
17.  }
18.  for (j = 1; j <= n; j++)
19.    if (check_safe(i, j)) {   //检查是否可以安全放置
20.      queen[i] = j;           //记录第i行皇后的放置在第j列
21.      set_conflict(i, j);     //设置冲突值
22.      dfs(i + 1);             //尝试第i+1行皇后的放置
23.      queen[i] = 0;           //取消第i行皇后的放置
24.      unset_conflict(i, j); //取消冲突值
25.    }
26.}
```

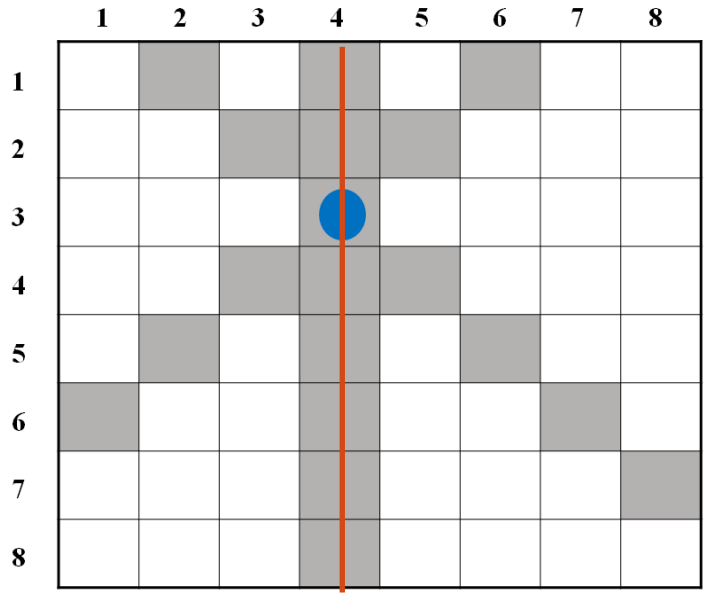
```
27.void initialize(){} //全局变量自动初始化为0，无需额初始化

28.int check_safe(int i, int j) {
29.  return (safe[i][j] == 0);
30.}

31.void set_conflict(int i, int j) {
32.  for (int k = 1; k <= n; k++) {
33.    safe[k][j]++; //同列的冲突值增加
34.    if (j+(k-i) >= 1 && j+(k-i) <= n && k != i)
35.      safe[k][j+(k-i)]++; //左对角线的冲突值增加
36.    if (j+(i-k) >= 1 && j+(i-k) <= n && k != i)
37.      safe[k][j+(i-k)]++; //右对角线的冲突值增加
38.  }
39.}

40.void unset_conflict(int i, int j) {
41.  for (int k = 1; k <= n; k++) {
42.    safe[k][j]--; //同列的冲突值减少
43.    if (j+(k-i) >= 1 && j+(k-i) <= n && k != i)
44.      safe[k][j+(k-i)]--; //左对角线的冲突值减少
45.    if (j+(i-k) >= 1 && j+(i-k) <= n && k != i)
46.      safe[k][j+(i-k)]--; //右对角线的冲突值减少
47.  }
48.}

49.void output() {
50.  for (int i=1; i<=n; i++) printf("%d ", queen[i]);
51.}
```

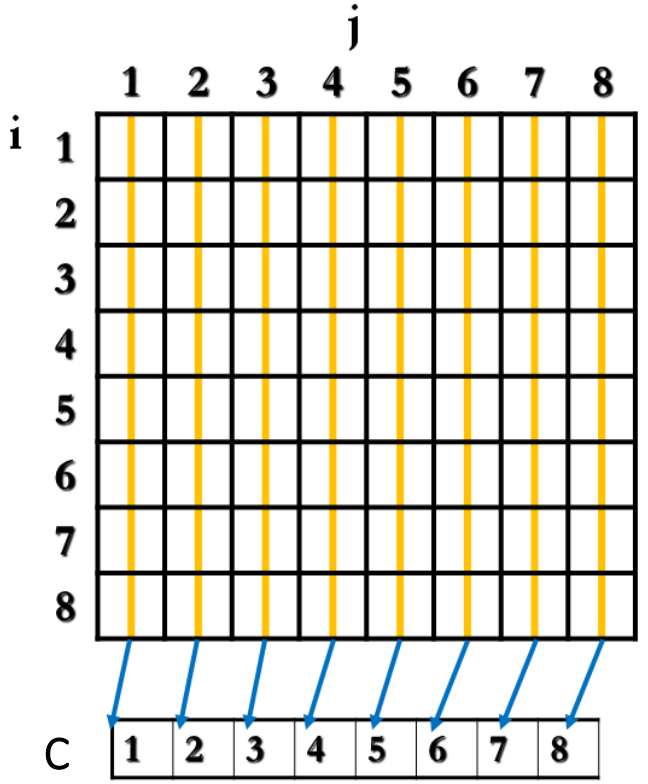
14.2.2 八皇后

- 优化：空间压缩（将二维标志数组safe坍塌成一维数组）
  - 当第 i 行的皇后放置在第 j 列时，棋盘上每一行的第 j 列都变得不安全
  - 定义一维数组C，记录列是否安全

    C[j] == 1：安全，C[j] == 0：不安全

    j = 1, 2, …, 8



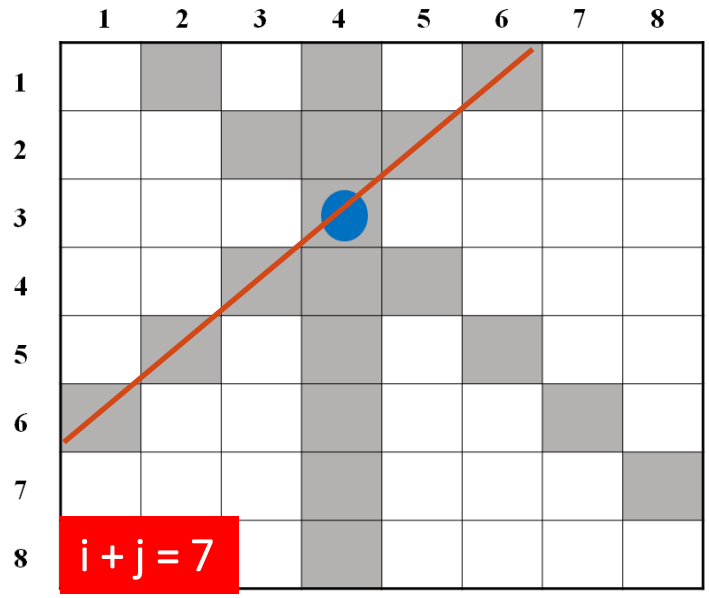


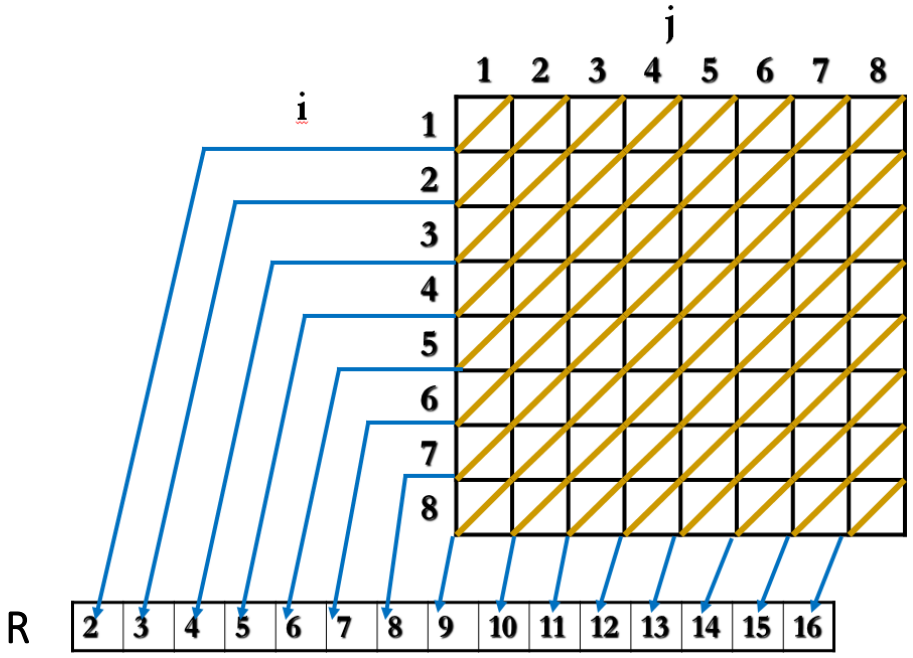
14.2.2 八皇后

- 优化：空间压缩（将二维标志数组safe坍塌成一维数组）
  - 观察到右对角线每个位置都有 i+j = 常数
  - 定义一维数组R，记录右对角线是否安全

    R[x] == 1：安全，R[x] == 0：不安全

    x = i + j，i = 1, 2, …, 8, j = 1, 2, …, 8, x = 2, 3, …, 16





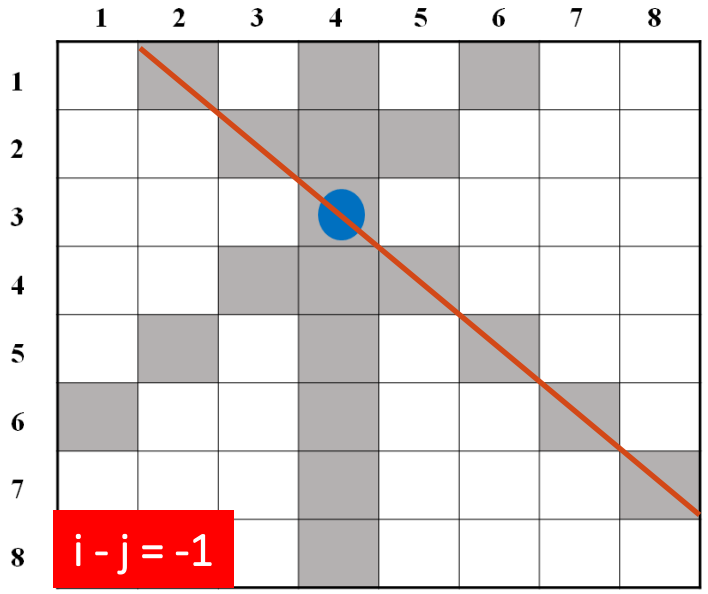
14.2.2 八皇后

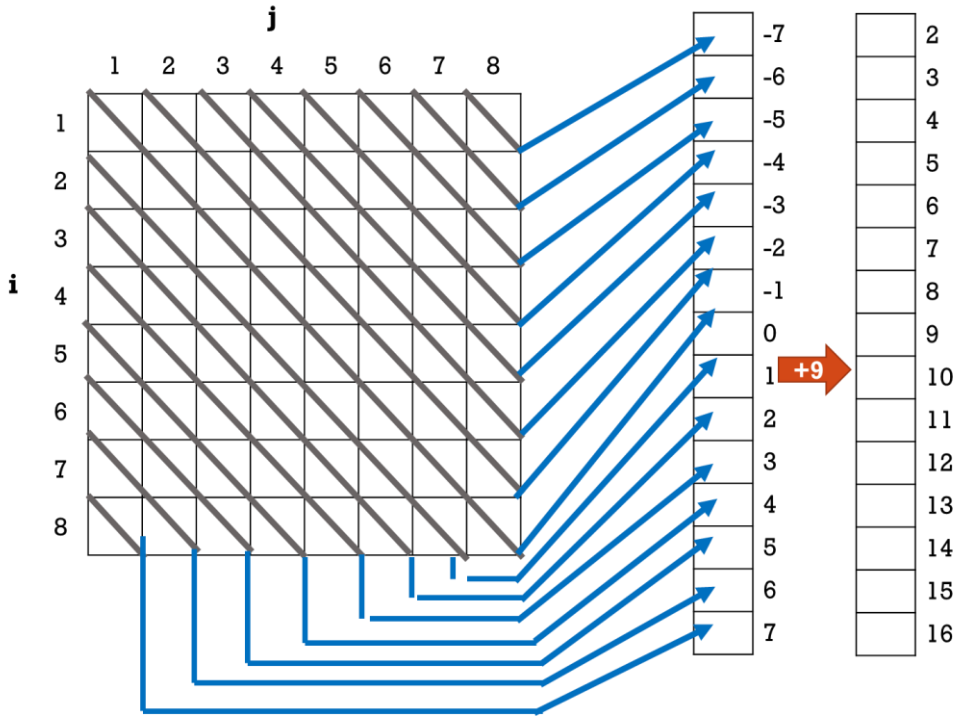
- 优化：空间压缩（将二维标志数组safe坍塌成一维数组）
  - 观察到左对角线每个位置都有 i-j = 常数
  - 定义一维数组L，记录左对角线是否安全

    L[y] == 1：安全，L[y] == 0：不安全

    y = i - j + 9，i = 1, 2, …, 8, j = 1, 2, …, 8, y = 2, 3, …, 16

注：数组的下标不能为负值，
因此要让y = i - j + 9，而不是y = i - j





14.2.2 八皇后

- 优化：空间压缩（将二维标志数组safe坍塌成一维数组）
  - 初始时，设置C, R, L数组元素值为1，代表全部都是安全的
  - 将第 i 行皇后放置在第 j 列之前，检测：
    C[j] && R[i+j] && L[i-j+9] == 1
  - 将第 i 行皇后放置在第 j 列之后，设置：
    C[j] = 0, R[i+j] = 0, L[i-j+9] = 0
  - 将第 i 行皇后放置在第 j 列回溯时，设置：
    C[j] = 1, R[i+j] = 1, L[i-j+9] = 1

```
1. int C[N], R[2*N], L[2*N];

2. void set_conflict(int i, int j) {
3.   int x = i+j, y = i-j+9;
4.   C[j] = 0, R[x] = 0, L[y] = 0;
5. }

6. void unset_conflict(int i, int j) {
7.   int x = i+j, y = i-j+9;
8.   C[j] = 1, R[x] = 1, L[y] = 1;
9. }
```

```
10.int check_safe(int i, int j) {
11.   int x = i+j, y = i-j+9;
12.   return (C[j] && R[x] && L[y]);
13.}

14.void initialize(){
15.   int i;
16.   for (i = 1; i <= n; i++) C[i] = 1;
17.   for (i = 1; i <= 2*n; i++) R[i] = 1;
18.   for (i = 1; i <= 2*n; i++) L[i] = 1;
19.}
```

14.2.2 八皇后

- 优化前后的对比
  - 空间上
    - 优化前：需要$O(n^2)$的空间用于存储安全标志
    - 优化后：需要$O(n)$的空间用于存储安全标志（C数组长度为n，R和L数组长度为$2n$）
  - 时间上
    - 优化前：需要$O(1)$的检查标志，需要$O(n)$的时间设置/取消标志
    - 优化后：需要$O(1)$的检查标志，需要$O(1)$的时间设置/取消标志

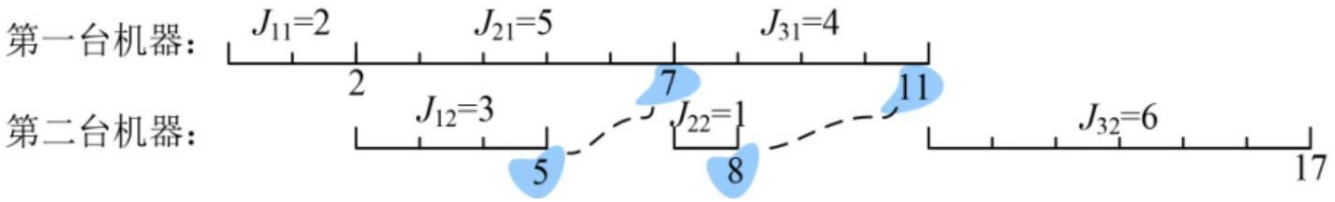
14.2.3 机器零件加工

- 有n个机器零件{J_1，J_2，…，J_n}
- 每个零件必须先由机器1处理，然后由机器2处理
- 零件Ji需要机器1、机器2的处理时间为t_{1i}、t_{2i}
- 如何安排零件加工顺序，使第一个零件从机器1上加工开始到最后一个零件在机器2上加工完成，所需的总加工时间最短?

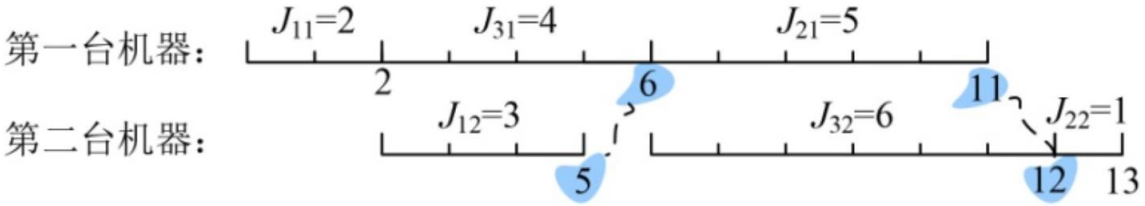
14.2.3 机器零件加工

- 问题分析
  - 根据问题的描述，不同的加工顺序，加工完所有零件所需的时间不同
  - 例如：现在有3个机器零件{J_1，J_2，J_3}，在第一台机器上的加工时间分别为2、5、4，在第二台机器上的加工时间分别为3、1、6

1. 如果按照{J_1，J_2，J_3}的顺序加工



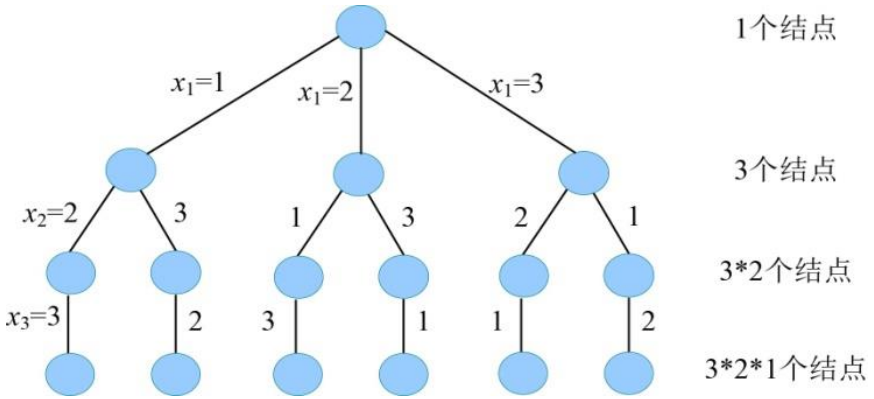
2. 如果按照{J_1，J_3，J_2}的顺序加工



  - 观察：第一台机器可以连续加工，而第二台机器开始加工的时间是**当前第一台机器的下线时间**和**第二台机器下线时间**的**最大值**。

14.2.3 机器零件加工

- 解空间
  - 3个机器零件有多少种加工顺序呢？即3个机器零件的全排列

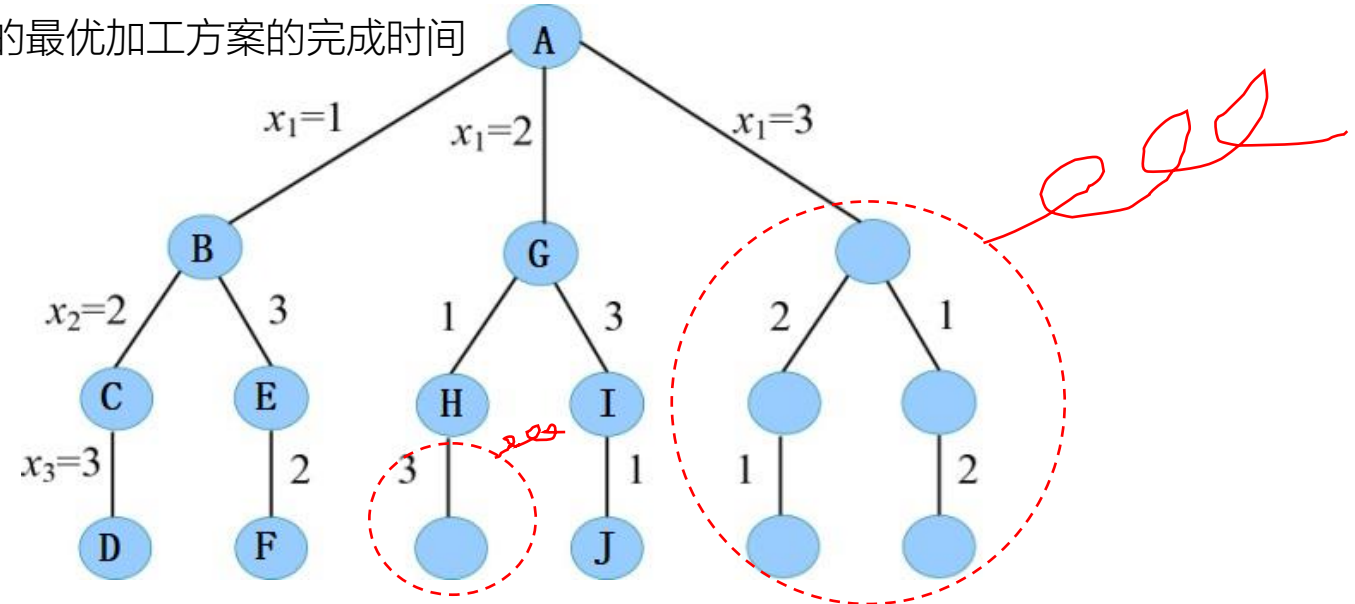


从根到叶子的路径就是机器零件的一个加工顺序，例如最右侧路径（3，1，2），表示先加工3号零件，再加工1号零件，最后加工2号零件

- 隐约束
  - 可行性约束：解的各个分量符合排列的要求（即互不相等）
  - 最优化约束：
    - 用f_2表示当前已完成的零件在第二台机器加工结束所用的时间
    - 用*bestf* 表示当前找到的最优加工方案的完成时间
    - 显然，继续向深处搜索时， f_2单调递增
    - 因此，当$f_2 \geq bestf$时，没有继续向深处搜索的必要

14.2.3 机器零件加工

- 状态
  - t：代表当前扩展结点在第几层（即第t个加工序），$t>n$时到达递归边界
  - $x[t]$：第t个加工序加工的零件编号
  - f_1：当前第一台机器上加工的完成时间
  - f_2：当前第二台机器上加工的完成时间
- 最优方案
  - *bestx*：当前找到的最优加工方案
  - *bestf*：当前找到的最优加工方案的完成时间



14.2.3 机器零件加工

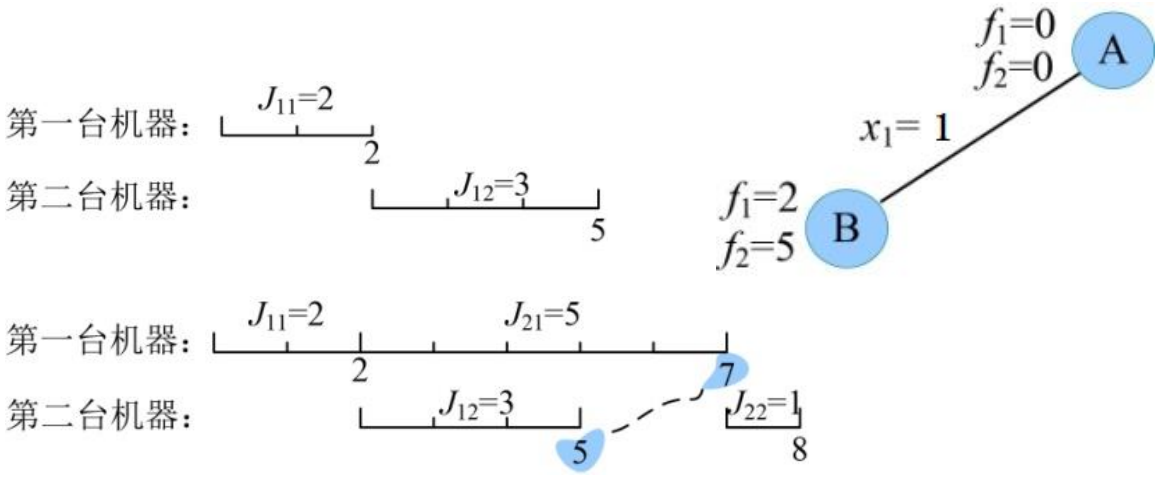
- 搜索过程

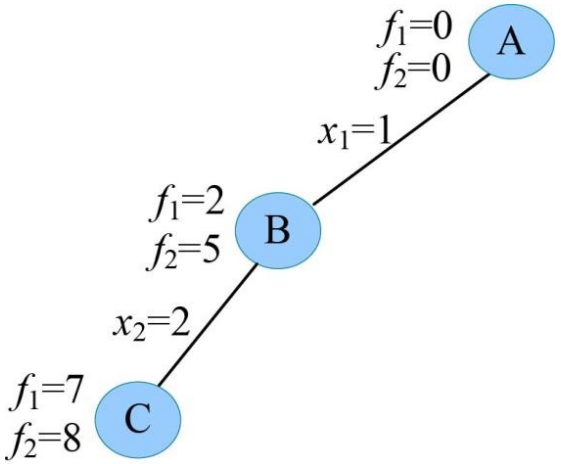
（1）开始搜索第1层（*t*=1）

扩展A结点的分支*x*1=1，*f*2=5，*bestf*的初值为无穷大，*f*2<*bestf*，满足限界条件，令x[1]=1，生成B结点，如图所示。

（2）扩展B结点（*t*=2）

扩展B结点的分支*x*2=2，*f*2=8，*bestf*的初值为无穷大，*f*2<*bestf*，满足限界条件，令x[2]=2，生成C结点，如图所示。





14.2.3 机器零件加工

- 搜索过程

（3）扩展C结点（t=3）

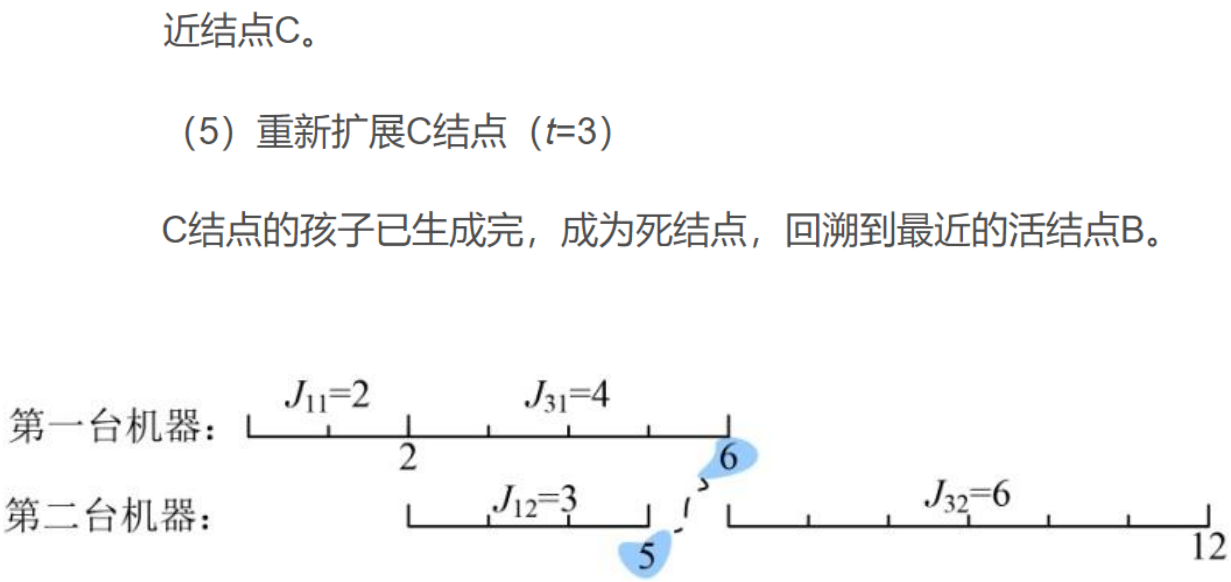
扩展C结点的分支$x3$=3，$f2$=17，$bestf$的初值为无穷大，$f2$<$bestf$，满足限界条件，令$x[3]$=3，生成D结点，如图所示。

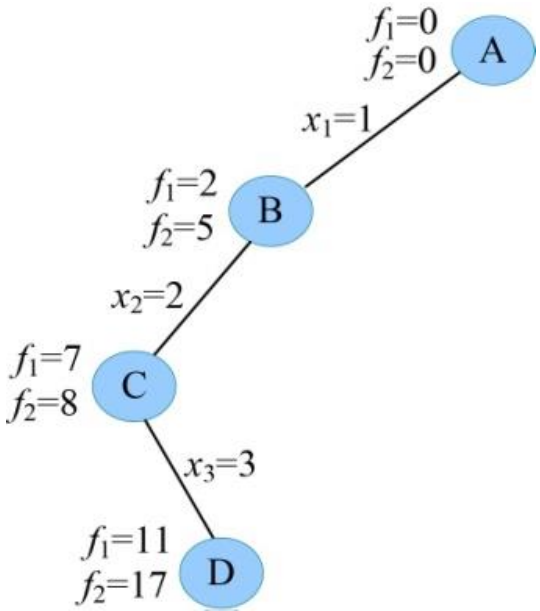
（4）扩展D结点（t=4）

t>n，找到一个当前最优解，记录最优值$bestf$=$f2$=17，用$bestx$[]保存当前最优解{1，2，3}。回溯到最近结点C。

（5）重新扩展C结点（t=3）

C结点的孩子已生成完，成为死结点，回溯到最近的活结点B。





14.2.3 机器零件加工

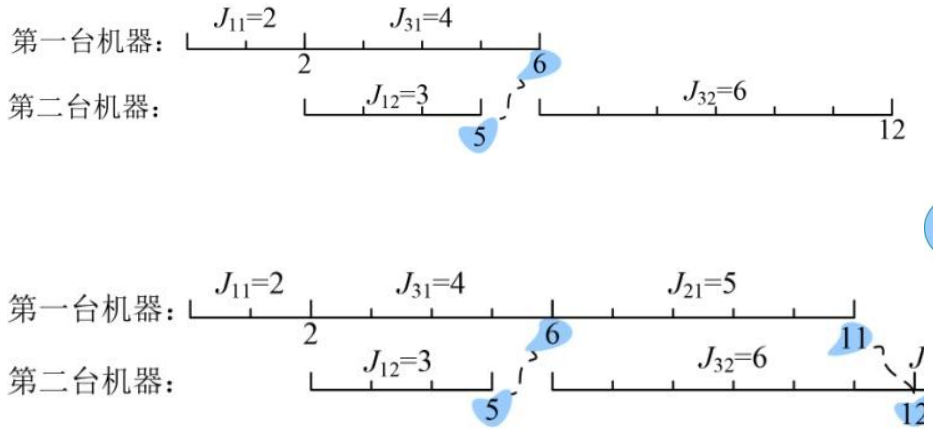
- 搜索过程

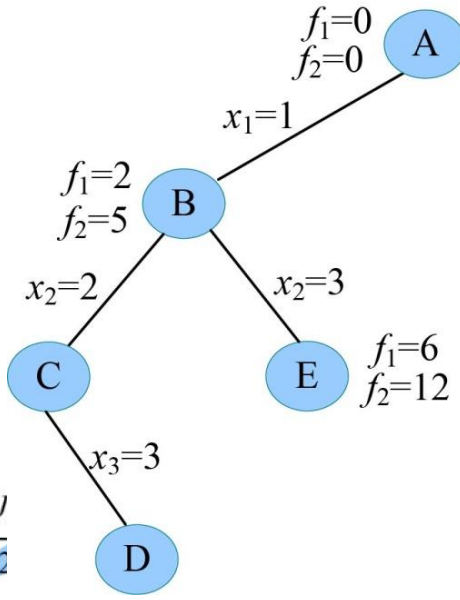
（6）重新扩展B结点（*t*=2）

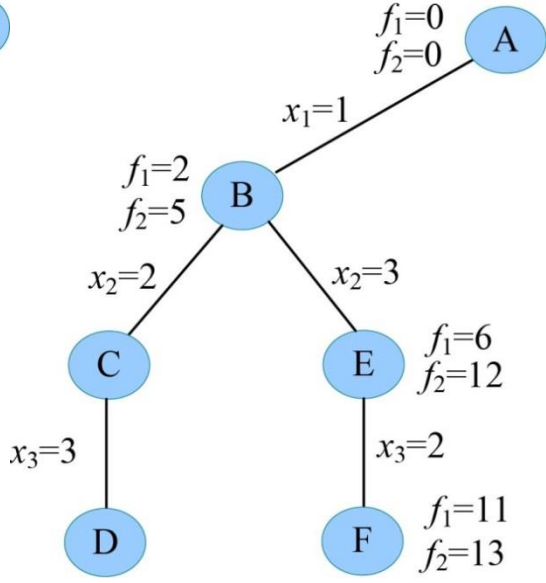
扩展B结点的分支*x*2=3，*f*2=12，*bestf*=17，*f*2<*bestf*，满足限界条件，令*x*[2]=3，生成E结点，如图所示。

（7）扩展E结点（*t*=3）

扩展E结点的分支*x*3=2，*f*2=13，*bestf*=17，*f*2<*bestf*，满足限界条件，令*x*[3]=2，生成F结点，如图所示。







14.2.3 机器零件加工

- 搜索过程

（8）扩展F结点（*t*=4）

t>*n*，找到一个当前最优解，记录最优值*bestf*=*f*2=13，用*bestx*[]保存当前最优解{1，3，2}。回溯到最近结点E。

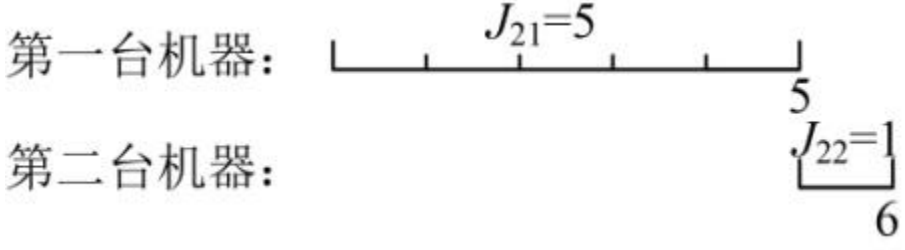
（9）扩展E结点（*t*=3）

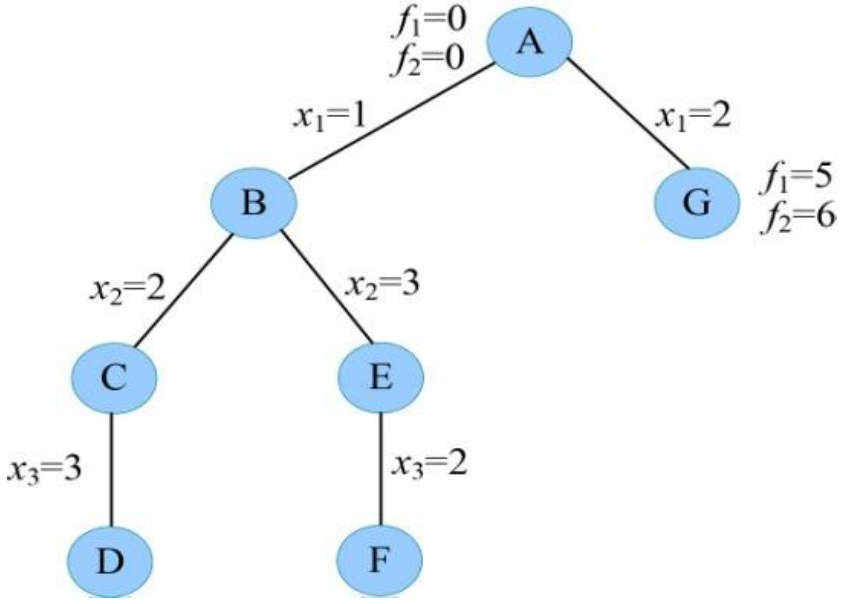
E结点的孩子已生成，成为死结点，回溯到最近的结点B。

E结点的孩子已生成完，成为死结点，回溯到最近的活结点A。

（10）重新扩展A结点（*t*=1）

扩展A结点的分支*x*1=2，*f*2=6，*bestf*=13，*f*2<*bestf*，满足限界条件，令*x*[1]=2，生成G结点，如图所示。





14.2.3 机器零件加工

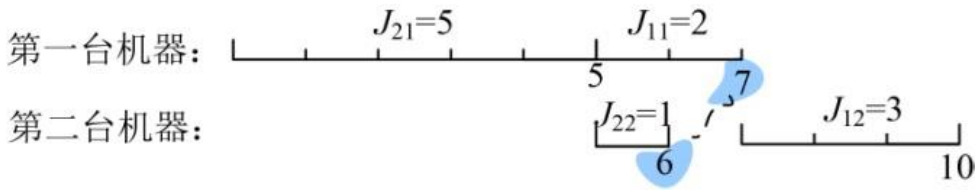
- 搜索过程

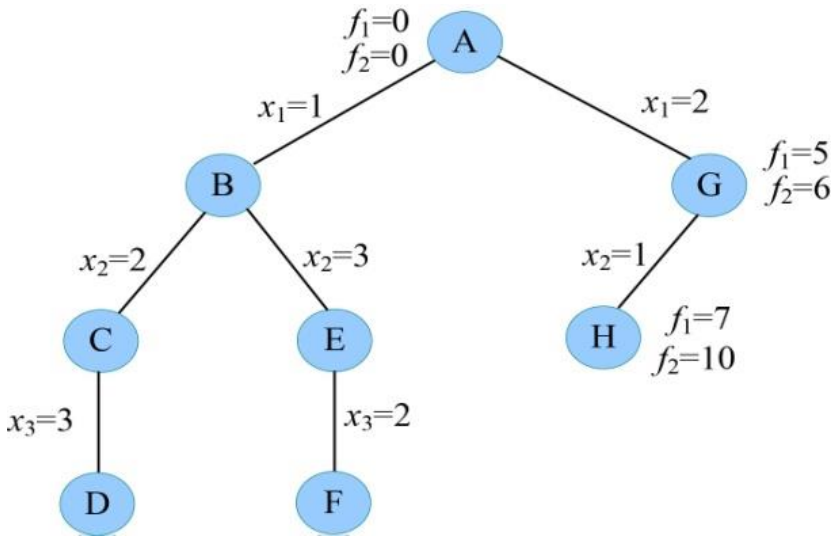
（11）扩展G结点（*t*=2）

扩展G结点的分支*x*2=1，*f*2=10，*bestf*=13，*f*2<*bestf*，满足限界条件，令*x*[2]=1，生成H结点，如图所示。

（12）扩展H结点（*t*=3）

扩展H结点的分支*x*3=3，*f*2=17，*bestf*=13，*f*2>*bestf*，不满足限界条件，剪掉该分支。H结点没有其他可扩展分支，成为死结点。回溯到G结点。





14.2.3 机器零件加工

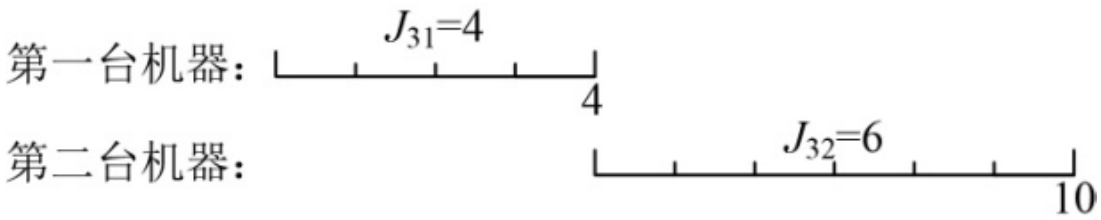
- 搜索过程

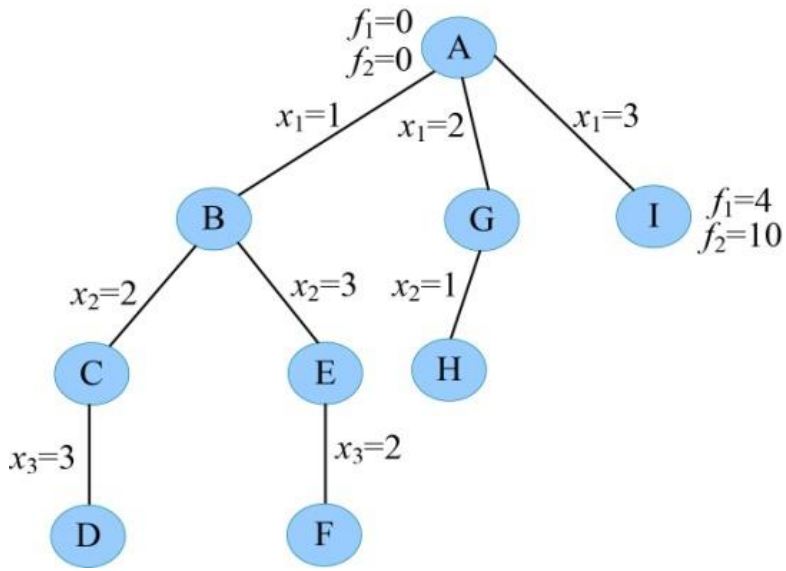
（13）重新扩展G结点（*t*=2）

扩展G结点的分支*x*2=3，*f*2=15，*bestf*=13，*f*2>*bestf*，不满足限界条件，剪掉该分支。G结点没有其他可扩展分支，成为死结点。回溯到A结点。

（14）重新扩展A结点（*t*=1）

扩展A结点的分支*x*1=3，*f*2=10，*bestf*=13，*f*2<*bestf*，满足限界条件，令*x*[1]=3，生成I结点，如图所示。





14.2.3 机器零件加工

- 搜索过程

（15）扩展I结点（*t*=2）

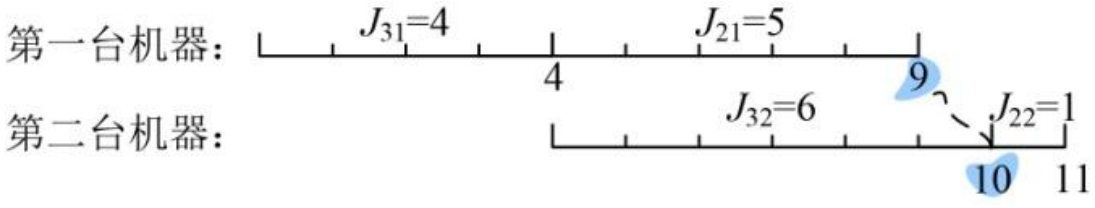
扩展I结点的分支*x*2=2，*f*2=11，*bestf*=13，*f*2<*bestf*，满足限界条件，令*x*[2]=2，生成J结点，如图所示。

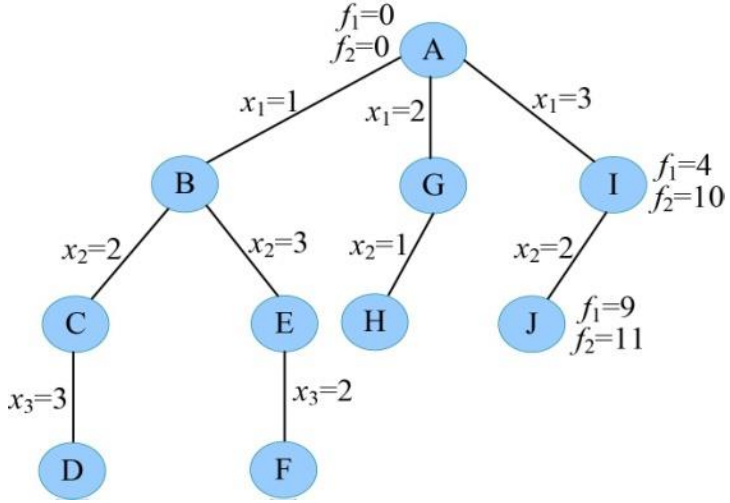
（16）扩展J结点（*t*=3）

扩展J结点的分支*x*3=1，*f*2=14，*bestf*=13，*f*2>*bestf*，不满足限界条件，剪掉该分支。J结点没有其他可扩展分支，成为死结点。回溯到I结点。

（17）重新扩展I结点（*t*=2）

扩展I结点的分支*x*2=1，*f*2=13，*bestf*=13，*f*2=*bestf*，不满足限界条件，剪掉该分支。I结点没有其他可扩展分支，成为死结点。回溯到A结点。A结点没有其他可扩展分支，成为死结点，算法结束。





14.2.3 机器零件加工

```
1. #include <stdio.h>
2. #define N 101

3. typedef struct  //记录每个零件在每台机器的加工时间
4. {
5.     int m1;     //零件第一台机器的加工时间
6.     int m2;     //零件第二台机器的加工时间
7. } Component;

8. Component comps[N];
9. int x[N], bestx[N]; //x:当前方案，bestx:当前最优加工方案
10.int used[N];
11.int n;
12.int f1, f2;  //当前第一台机器、第二台机器加工的完成时间
13.int bestf = 0x7FFFFFFF;//当前找到的最优加工方案的完成时间

14.void dfs(int);
15.int max(int, int);

16.void main() {
17.  scanf("%d", &n);
18.  for (int i = 1; i <= n; i++)
19.    scanf("%d %d", &comps[i].m1, &comps[i].m2);

20.  dfs(1);

21.  printf("best: %d\n", bestf);
22.  for (i = 1; i <= n; i++) printf("%d ", bestx[i]);
23.}
```

```
24.int max(int x, int y) { return (x >=y ? x : y); }

25.void dfs(int t) {
26.  int i, tmp;
27.  if (t > n) {                          //递归边界
28.    if (f2 < bestf) {                   //更新最优解
29.      for (i = 1; i <= n; i++) bestx[i] = x[i];
30.      bestf = f2;
31.    }
32.    return;
33.  }

34.  for (i = 1; i <= n; i++) {
35.    if (used[i]) continue;              //可行性隐约束

36.    x[t] = i;
37.    used[i] = 1;
38.    f1 += components[x[t]].m1;
39.    tmp = f2;
40.    f2 = max(f1, f2) + components[x[t]].m2;

41.    if (f2 < bestf)                     //最优化隐约束
42.      dfs(t + 1);

43.    used[i] = 0;
44.    f1 -= components[x[t]].m1;
45.    f2 = tmp;
46.  }
47.}
```

全排列相关的代码

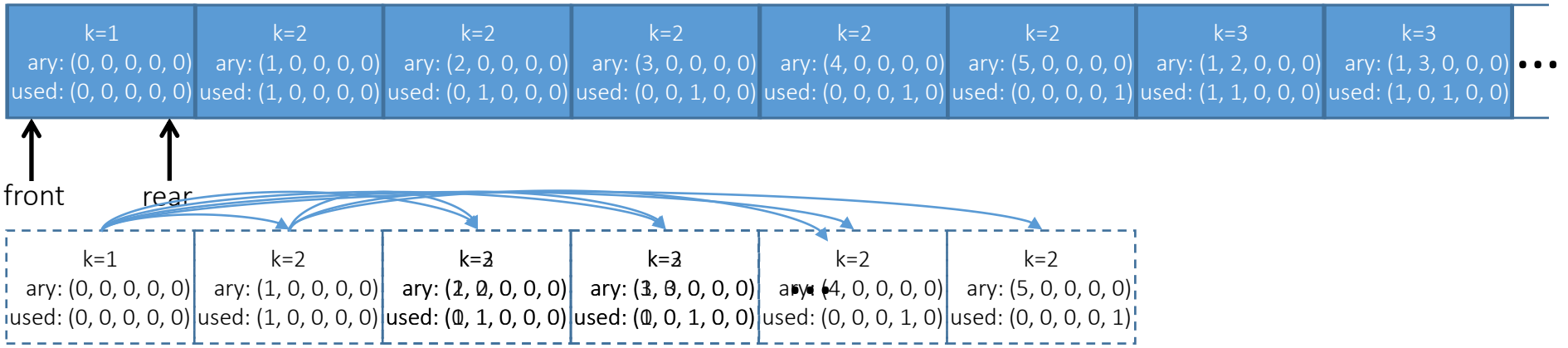
最优值相关的代码

14.3 广度优先搜索（分支限界法）

- 基本思想
  - 根据显约束遍历解空间，一次性生成所有的孩子结点
  - 根据隐约束判定孩子结点时保留还是舍弃
  - 维护一个活结点队列（先进先出队列或优先队列）
    - 先进先出队列：元素在队列尾追加，而从队列头删除
    - 优先队列：元素被赋予优先级，具有最高优先级的元素最先删除
  - 每次从活结点队列中取出一个结点进行扩展，直至活结点队列为空
  - 每个活结点最多只有一次机会成为扩展结点
- 例题
  - 八数码：先进先出队列
  - 拯救公主：优先队列

14.3.1 全排列的广度优先搜索算法

- 定义一个状态结构体State
  - k：当前正在穷举的变量
  - ary：存放当前已排列好的变量值
  - used：标志数组，used[i]的值代表i是否被使用过
- 定义一个先进先出队列queue
  - front：队头，指向当前队列中最早入队的元素
  - rear：队尾，--rear指向当前队列中最晚入队的元素
  - 初始时：设置front = rear = 0；入队：rear++；出队：front++



14.3.1 全排列的广度优先搜索算法

```
1. #include <stdio.h>
2. #include <stdlib.h>
3. #define N 101                  //最多N个数的全排列
4. #define QueueSize 10000        //队列的最大长度

5. typedef struct {
6.   int k; int ary[N]; int used[N];
7. } State;
8. typedef State DataType;
9. typedef struct {
10.  DataType data[QueueSize];
11.  int front, rear;
12.} SeqQueue;

13.SeqQueue queue;                 //状态队列
14.int n, nTotal;

15.void InitQueue(SeqQueue *queue);
16.void ClearQueue(SeqQueue *queue);
17.int IsEmpty(SeqQueue *queue);
18.int IsFull(SeqQueue *queue);
19.void Enqueue(SeqQueue *queue, DataType data);
20.DataType Front(SeqQueue *queue);
21.DataType Dequeue(SeqQueue *queue);

22.void output(State *state) {
23.  for(int i=1;i<=n;i++) printf("%d ", state->ary[i]);
24.  printf("\n");
25.}
```

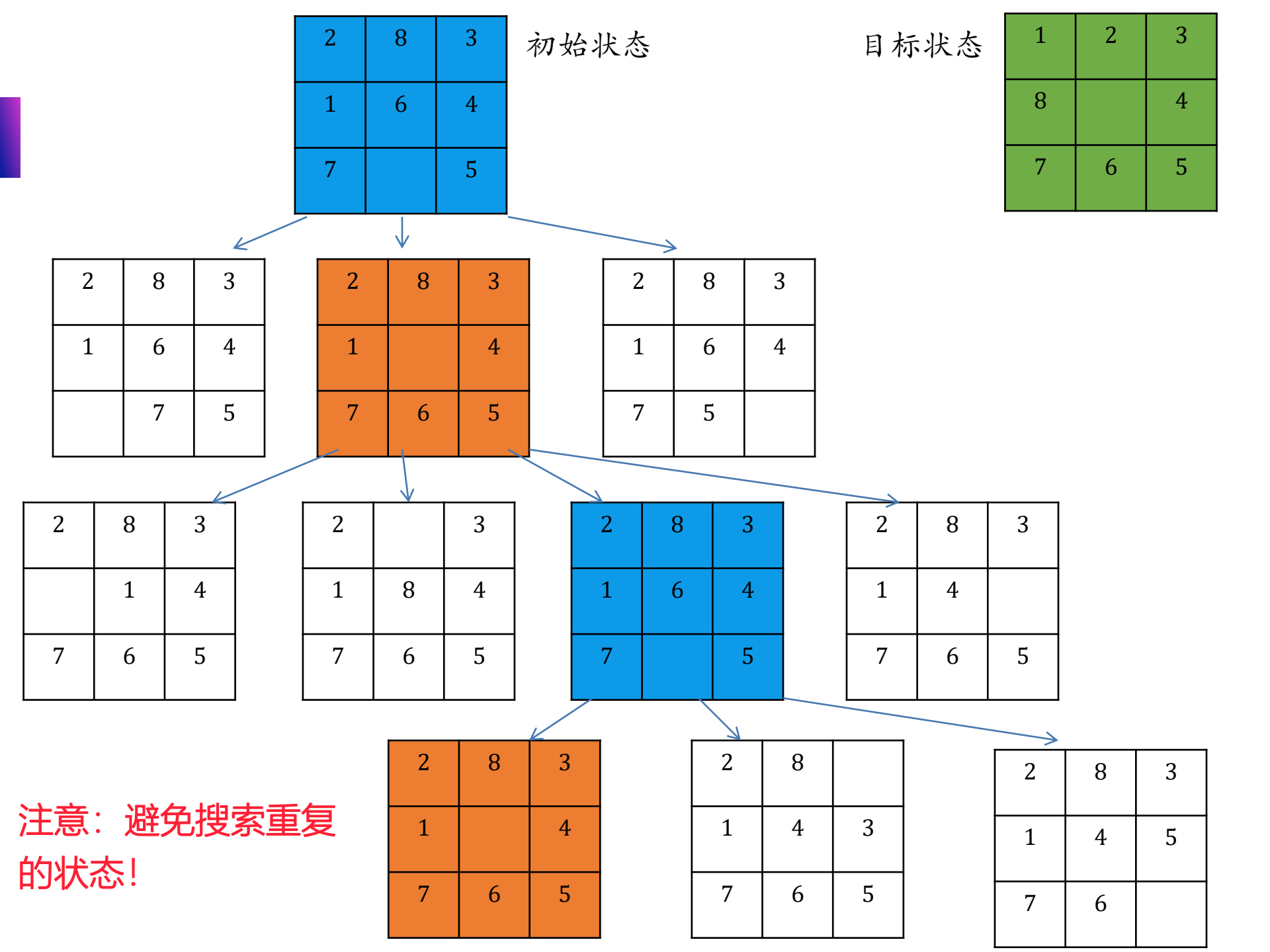
```
26.void bfs(SeqQueue *queue) {
27.  State state, new_state;
28.  while (!IsEmpty(queue)) {            //直到队列为空
29.    state = Dequeue(queue);            //出队
30.    for (int i = 1; i <= n; i++) {
31.      if (state.used[i]) continue;     //可行性隐约束
32.      new_state = state;
33.      new_state.ary[new_state.k++] = i;
34.      new_state.used[i] = 1;
35.      if (new_state.k > n) {           //找到可行解
36.        nTotal++;
37.        output(&new_state);
38.        continue;
39.      }
40.      Enqueue(queue, new_state);       //入队
41.    }
42.  }
43.}

44.void main() {
45.  State init = {1, {0}, {0}};
46.  InitQueue(&queue);                  //初始化队列
47.  Enqueue(&queue, init);              //初始化状态入队
48.  scanf("%d", &n);
49.  bfs(&queue);
50.  printf("nTotal: %d\n", nTotal);
51.  ClearQueue(&queue);                 //清空队列
52.}
```

14.3.2 八数码

- 3×3九宫棋盘，放置数码为1 -8的8个棋牌，剩下一个空格，只能通过棋牌向空格的移动来改变棋盘的布局。
- 根据给定初始布局（即初始状态）和目标布局（即目标状态），如何移动棋牌才能从初始布局到达目标布局。
- 找到移动步数最少的合法走步序列。

八数码难题（8-puzzle problem）

（初始状态）　　（目标状态）


初始状态
目标状态
2 8 3
1 6 4
7 5
1 2 3
8 4
7 6 5
2 8 3
1 6 4
7 5
2 8 3
1 4
7 6 5
2 8 3
1 6 4
7 5
2 8 3
1 6 4
7 5
2 8 3
1 8 4
7 6 5
2 8 3
1 6 4
7 5
2 8 3
1 4
7 6 5
2 8 3
1 4 3
7 6 5
2 8 3
1 4 5
7 6
注意：避免搜索重复的状态！

14.3.2 八数码

- 问题分析
  - 棋盘状态：如果把空格记成0，棋盘上的所有状态共有 9! = 362880种
  - 棋盘状态编码：
    - 用3*3矩阵记录棋盘状态，至少需要9 * sizeof(char)=9个字节
    - 将棋盘状态编码成整数，范围[012345678，876543210]，需要sizeof(int)=4个字节
  - 去除重复状态：
    - 遍历整个队列，看待判定的状态是否出现过，费时间
    - 用一个标志数组记录状态的编码是否出现过，至少需要10000000000，费空间

14.3.2 八数码

```
1. #include <stdio.h>
2. #include <stdlib.h>

3. #define QueueSize 10000          //队列的最大长度
4. #define RepeatSize 10000000000 //重复标志数组的最大长度
5. #define TraceSize 1000          //操作轨迹数组的最大长度

6. typedef struct
7. {
8.    int code;                  //棋盘状态编码
9.    int from;                  //前一个状态在队列中的位置
10.} State;

11.typedef State DataType;

12.typedef struct
13.{
14.    DataType data[QueueSize];
15.    int front, rear, cnt;
16.} SeqQueue;

17.SeqQueue queue;              //状态队列
18.char repeat[RepeatSize];   //重复标志数组
19.State trace[TraceSize];    //操作轨迹数组
20.int source, target;         //初始棋盘布局，目标棋盘布局

21.int direction[4][2] = {   //上下左右移动坐标增量
22.  {-1, 0}, {1, 0}, {0, -1}, {0, 1}
23.};
```

```
24.void InitQueue(SeqQueue *queue);
25.void ClearQueue(SeqQueue *queue);
26.int IsEmpty(SeqQueue *queue);
27.int IsFull(SeqQueue *queue);
28.void Enqueue(SeqQueue *queue, DataType data);
29.DataType Front(SeqQueue *queue);
30.DataType Dequeue(SeqQueue *queue);

31.int check_repeat(State state)
32.{
33.    return (repeat[state.code] == 1);
34.}

35.void set_repeat(State state)
36.{
37.    repeat[state.code] = 1;
38.}

39.void swap(int *x, int *y)
40.{
41.    int tmp;
42.    tmp = *x;
43.    *x = *y;
44.    *y = tmp;
45.}
```

14.3.2 八数码

```
1. int encode(int plate[3][3]){
2.   int i, j, weight = 1, code = 0;
3.   for (i = 2; i >= 0; i--)
4.     for (j = 2; j >= 0; j--) {
5.       code += plate[i][j] * weight;
6.       weight *= 10;
7.     }
8.   return code;
9. }   //编码: code->plate (数位合并)

10.void decode(int code, int plate[3][3]) {
11.  int i, j, weight = 1;
12.  for (i = 2; i >= 0; i--)
13.    for (j = 2; j >= 0; j--) {
14.      plate[i][j] = code % 10;
15.      code /= 10;
16.    }
17.}   //解码: code->plate (数位分离)

18.int find_zero(int plate[3][3]) {
19.  int i, j, find = 0;
20.  for (i = 0; i <= 2; i++) {
21.    for (j = 0; j <= 2; j++)
22.      if (plate[i][j] == 0) { find = 1; break; }
23.    if (find) break;
24.  }
25.  return (find ? i * 3 + j : -1);
26.}   //查找0所在的位置 (二维坐标转一维坐标)
```

```
27.void output_plate(int plate[3][3]) {
28.  for (int i = 0; i <= 2; i++) {
29.    for (int j = 0; j <= 2; j++)
30.        printf("%d ", plate[i][j]);
31.    printf("\n");
32.  }
33.}

34.void output(SeqQueue *queue, State state) {
35.  int plate[3][3];
36.  int top = 0, step = 0;
37.  State prev_state, curr_state;

38.  prev_state = state;
39.  do {
40.    curr_state = prev_state;
41.    trace[top++] = curr_state;
42.    prev_state = queue->data[curr_state.from];
43.  } while (curr_state.code != prev_state.code);

44.  printf("minimum steps: %d\n", top-1);

45.  while (top > 0) {
46.    curr_state = trace[--top];
47.    decode(curr_state.code, plate);
48.    printf("Step-%d:\n", step++);
49.        output_plate(plate);
50.  }
51.}
```

```
1. void bfs(SeqQueue *queue) {
2.   State state, new_state;
3.   int i, j, k, new_i, new_j, from, zero, plate[3][3];

4.   while (!IsEmpty(queue)) {            //直到队列为空
5.     from = queue->front;
6.     state = Dequeue(queue);            //出队

7.     decode(state.code, plate);
8.     zero = find_zero(plate);           //寻找0所在位置
9.     i = zero / 3, j = zero % 3;

10.    for (k = 0; k < 4; k++) {
11.      new_i = i + direction[k][0]; //移动改变0所在行
12.      new_j = j + direction[k][1]; //移动改变0所在列

13.      if (new_i<0 || new_i>2 || new_j<0 || new_j>2)
14.        continue;  //可行性约束：移动后0的位置必须在棋盘内
15.      //新方案改变棋盘布局
16.      swap(&plate[i][j], &plate[new_i][new_j]);

17.      new_state.code = encode(plate);
18.      new_state.from = from;

19.      if (new_state.code == target) { //找到解
20.        output(queue, new_state); exit(0);
21.      }

22.      if (!check_repeat(new_state)) {//判断重复状态
23.        Enqueue(queue, new_state);    //入队
24.        set_repeat(new_state);        //设置重复状态
25.      }
26.      //尝试下一个新方案前，要先恢复棋盘布局
27.      swap(&plate[i][j], &plate[new_i][new_j]);
28.    }
29.  }
30.}
```

①

②

```
31.void main() {
32.  State init;

33.  scanf("%d %d", &source, &target);

34.  InitQueue(&queue);
35.  init.code = source;
36.  init.from = 0;
37.  Enqueue(&queue, init);

38.  set_repeat(init);
39.  bfs(&queue);

40.  ClearQueue(&queue);
41.}
```

思考：①处的代码放到②处，有何不同？

```
283164705 123804765
minimum steps: 5
```

```
Step-0: Step-1: Step-2:
2 8 3   2 8 3   2 0 3
1 6 4   1 0 4   1 8 4
7 0 5   7 6 5   7 6 5
```

```
Step-3: Step-4: Step-5:
0 2 3   1 2 3   1 2 3
1 8 4   0 8 4   8 0 4
7 6 5   7 6 5   7 6 5
```

14.3.3 拯救公主

- 背景：法庭授权绝地武士逮捕西斯，维护和平。然而Dastan却听到纳波星球公主被杰吧怪兽俘虏的消息决定前去救援。Dastan经过重重难关，攻入地牢，拯救公主。
- 设定：目前Dastan只剩下M(0<=M<=100)点HP。怪兽和守卫散布在地牢里。当Dastan到达敌人所在位置时，发生战斗会受到一定伤害，并最终战胜敌人。根据主角不死定律，王子HP为0时先手发动必杀技，直接秒杀敌人。迷宫中，Dastan每次只能上下左右移动1格。
- 任务：对与给定的地图、Dastan和公主坐标。你的任务是计算Dastan救出公主能够剩下的最多HP。

14.3.3 拯救公主

- 输入：地牢由一个N*N（2<=N<=10）的矩阵描述，矩阵上'D'代表Dastan所在位置，'P'代表公主所在位置，'#'代表石柱或墙，'0'代表空地，数字'1' - '9'代表王子在这里发生战斗要消耗的HP。输入的第一行是N和M，之后是N*N的矩阵
- 输出：若Dastan能够救出公主，输出所剩的最大HP，否则输出"Impossible"

样例输入：

3 10
D12
9#6
87P

样例输出：

1

样例说明：

(0, 0) → (0,1) →(0, 2) →(1, 2) →(2, 2)

	0	1	2
0	D	1	2
1	9	#	6
2	8	7	P

14.3.3 拯救公主

- 问题分析
  - 状态：
    - 当前在地图中所处的位置
    - 当前体力值
    - 记录访问历史的标记数组（某个位置是否曾经被访问过）
  - 状态转移的约束条件：
    - 可行性隐约束1：新的位置要在地图范围内
    - 可行性隐约束2：新的位置没有路障
    - 可行性隐约束3：新的位置没被访问过
  - 状态转移时的变化：
    - 在地图中所处的位置
    - 体力值减少（按题意只要当前体力值降到0，就一直保持0不再减少）

14.3.3 拯救公主

```
1. #include <stdio.h>
2. #include <stdlib.h>

3. #define N 10

4. #define QueueSize 10000 //队列的最大长度

5. typedef struct
6. {
7.     int x, y; //当前所处的坐标
8.     int hp;   //体力值
9. } State;

10.typedef State DataType;

11.typedef struct
12.{
13.    DataType data[QueueSize];
14.    int front, rear;
15.} SeqQueue;

16.SeqQueue queue;  //状态队列

17.char map[N][N];  //地图信息
18.int visit[N][N]; //标记坐标是否访问过
19.int found;       //是否找到解
20.int n, m;
21.int source, target;

22.int direction[4][2] =
23.{  //上下左右移动坐标增量
24.  {-1, 0}, {1, 0}, {0, -1}, {0, 1}
25.};

26.int compar(const void *a, const void *b)
27.{
28.    DataType *aa = (DataType *)a;
29.    DataType *bb = (DataType *)b;
30.    return bb->hp - aa->hp;  //hp值大的排前面
31.}

32.void InitQueue(SeqQueue *queue);
33.void ClearQueue(SeqQueue *queue);
34.int IsEmpty(SeqQueue *queue);
35.int IsFull(SeqQueue *queue);
36.void Enqueue(SeqQueue *queue, DataType data);
37.DataType Front(SeqQueue *queue);
38.DataType Dequeue(SeqQueue *queue);
```

```
1. void bfs(SeqQueue *queue) {
2.   State state, new_state;
3.   int k, new_x, new_y;

4.   while (!IsEmpty(queue)) {
5.     qsort(queue->data + queue->front,
6.           queue->rear - queue->front,
7.           sizeof(DataType),
8.           compar); //队列元素按hp值从高到低排序（优先队列）
9.     state = Dequeue(queue);          //出队
10.    for (k = 0; k < 4; k++) {
11.      new_x = state.x + direction[k][0];
12.      new_y = state.y + direction[k][1];
13.      //新位置必须在地图范围内
14.      if (new_x<0 || new_x>=n || new_y<0 || new_y>=n)
15.        continue;
16.      //新位置必须没有路障，并且没被访问过
17.      if (map[new_x][new_y] == '#' ||
18.          visit[new_x][new_y]) continue;
19.      //到达公主所在位置，优先队列保证找到的第一个解是最优解
20.      if (map[new_x][new_y] == 'P') {
21.        printf("%d\n", state.hp);
22.        found = 1;
23.        return;
24.      }
25.      //从父结点状态产生新的子结点状态，位置和体力值发生变化
26.      new_state = state;
27.      new_state.x = new_x;
28.      new_state.y = new_y;
29.      new_state.hp -= (map[new_x][new_y] - '0');
30.      if (new_state.hp < 0) new_state.hp = 0;

31.      visit[new_x][new_y] = 1;
32.      Enqueue(queue, new_state);    //入队
33.    }
34.  }
35.}
```

```
36.int main() {
37.  int i, j;
38.  State init;

39.  scanf("%d %d", &n, &m);
40.  for (i = 0; i < n; i++) {
41.    scanf("\n");
42.    for (j = 0; j < n; j++) {
43.      scanf("%c", &map[i][j]);
44.      if (map[i][j] == 'D') source = i * n + j;
45.      if (map[i][j] == 'P') target = i * n + j;
46.    }
47.  }
48.  //初始状态：位于'D'所在位置，体力值为m
49.  init.x = source/n;
50.  init.y = source%n;
51.  init.hp = m;
52.  InitQueue(&queue);
53.  Enqueue(&queue, init);
54.  bfs(&queue);
55.  if (!found) printf("Impossible\n");
56.  ClearQueue(&queue);
57.}
```

思考1：为什么visit数组可以设置为全局，而不是每个结点一个？

思考2：这题使用优先队列为何可以保证找到的第一个解就是最优解？

14.4 两种搜索算法的比较

- 空间耗费
  - 深度优先搜索法不全部保留结点，扩展完的结点从数据存储结构栈中弹出删去，在栈中存储的结点数就是解空间树的深度，因此占用空间较少
  - 广度优先搜索算法一般需存储产生所有结点，占用的存储空间要比深度优先搜索大得多，必须考虑溢出和节省内存空间的问题
- 时间耗费
  - 深度优先搜索一般使用递归回溯的方式，需要频繁进行函数调用和返回，运行速度较慢
  - 广度优先搜索法一般无回溯操作（即入栈和出栈的操作），故运行速度比深度优先搜索要快些

14.4 两种搜索算法的比较

- 求解能力
  - 深度优先搜索，若选择的分支沿着正确的解方向，则可以保证较快求得一个可行解。但如果误入无穷分枝（深度无限），则可能找不到解
  - 广度优先搜索，能保证最终在问题有解的情况下，找到由起始节点到目标节点的最短路径的解。但它在每一步搜索时，都遍历了上一步往后的所有可行结点，因此搜索过程很长
- 适用场景：取决于搜索树结构，解个数及其在搜索树中的位置

- If you know a solution is not far from the root of the tree, a breadth first search (BFS) might be better.
- If the tree is very deep and solutions are rare, depth first search (DFS) might take an extremely long time, but BFS could be faster.
- If the tree is very wide, a BFS might need too much memory, so it might be completely impractical.
- If solutions are frequent but located deep in the tree, BFS could be impractical.
- If the search tree is very deep, you will need to restrict the search depth for depth first search (DFS).
- But these are just rules of thumb; you'll probably need to experiment.

14.4.1 全排列问题的DFS与BFS对比

DFS实现

```
1. int ary[N], used[N], n;

2. void dfs(int k) {
3.    ……
4.    for (i=1; i<=n; i++) {          //n个可能的取值
5.      if (used[i]) continue;        //判断i是否用过
6.      ary[k] = i;                   //第k变量取值i
7.      used[i] = 1;                  //标志i用过了
8.      dfs(k+1);                     //穷举第k+1个变量
9.      used[i] = 0;                  //恢复i未用过
10.   }
11.}
```

- 在DFS中，父子结点共享ary, used状态信息，递归时设置增量状态，回溯时撤销增量状态
- 在BFS中，子结点从父结点处复制一份独立的状态信息，设置的增量状态无需撤销
- 对于需要遍历整个搜索空间才能求解的问题，DFS，BFS访问的结点个数相同

BFS实现

```
1. typedef struct {
2.    int k; int ary[N]; int used[N];
3. } State;

4. typedef State DataType;

5. typedef struct {
6.      DataType data[QueueSize];
7.      int front, rear;
8. } SeqQueue;

9. SeqQueue queue;

10.  void bfs(SeqQueue *queue) {
11.     ……
12.     for (i = 1; i <= n; i++) {
13.       if (state.used[i]) continue; //可行性隐约束

14.       new_state = state;
15.       new_state.ary[new_state.k++] = i;
16.       new_state.used[i] = 1;
17.       Enqueue(&queue, new_state); //入队
18.     }
19.  }
```

14.4.2 八皇后问题的DFS与BFS对比

- 八皇后问题的BFS实现

```
1. #include <stdio.h>
2. #include <stdlib.h>
3. #define N 9
4. #define QueueSize 10000 //队列的最大长度

5. typedef struct {
6.    int i;
7.    int queen[N], C[N], R[2 * N], L[2 * N];
8. } State;
9. typedef State DataType;
10.typedef struct {
11.   DataType data[QueueSize];
12.   int front, rear;
13.} SeqQueue;

14.SeqQueue queue; //状态队列

15.int nTotal, n = 8;

16.void InitQueue(SeqQueue *queue);
17.void ClearQueue(SeqQueue *queue);
18.int IsEmpty(SeqQueue *queue);
19.int IsFull(SeqQueue *queue);
20.void Enqueue(SeqQueue *queue, DataType data);
21.DataType Front(SeqQueue *queue);
22.DataType Dequeue(SeqQueue *queue);
```

```
23.void initialize(State *state) {
24.     state->i = 1;
25.     for (int i = 1; i <= n; i++) state->C[i] = 1;
26.     for (int i = 1; i <= 2 * n; i++) state->R[i] = 1;
27.     for (int i = 1; i <= 2 * n; i++) state->L[i] = 1;
28.}

29.void set_conflict(State *state, int i, int j) {
30.   int x = i + j, y = i - j + 9;
31.   state->C[j] = 0, state->R[x] = 0, state->L[y] = 0;
32.}

33.void unset_conflict(State *state, int i, int j) {
34.   int x = i + j, y = i - j + 9;
35.   state->C[j] = 1, state->R[x] = 1, state->L[y] = 1;
36.}

37.int check_safe(State *state, int i, int j) {
38.   int x = i + j, y = i - j + 9;
39.   return (state->C[j] && state->R[x] && state->L[y]);
40.}

41.void output(State *s){
42.   for (int i=1; i<=n; i++) printf("%d ",s->queen[i]);
43.   printf("\n");
44.}
```

14.4.2 八皇后问题的DFS与BFS对比

- 八皇后问题的BFS实现

```
1. void bfs(SeqQueue *queue) {
2.   int j;
3.   State state, new_state;
4.   while (!IsEmpty(queue)) {
5.     state = Dequeue(queue);                                          //出队
6.     for (j = 1; j <= n; j++)
7.       if (check_safe(&state, state.i, j)) {                          //检查是否可以安全放置
8.         new_state = state;
9.         new_state.queen[new_state.i] = j;                            //记录第i行皇后的放置在第j列
10.        set_conflict(&new_state, new_state.i, j);                    //设置冲突值
11.        new_state.i++;
12.        if (new_state.i > n) { output(&new_state); nTotal++; continue; }  //找到可行解
13.        Enqueue(queue, new_state);                                   //入队
14.      }
15.  }
16.}

17.void main() {
18.    State init;
19.    InitQueue(&queue);
20.    initialize(&init);
21.    Enqueue(&queue, init);
22.    bfs(&queue);
23.    printf("nTotal: %d\n", nTotal);
24.    ClearQueue(&queue);
25.}
```

14.4.2 八皇后问题的DFS与BFS对比

生成第x个解所遍历的结点数	DFS	BFS
x = 1	113	1665
x = 2	146	1669
x = 3	171	1671
x = 4	188	1672
x = 5	251	1678
x = 10	408	1702
x = 20	614	1739
x = 50	1033	1819
x = 92	1860	1954
total	1965	1965

- 解在搜索树中分布较为均匀的情况下，DFS相比于BFS能更快产生可行的解
- 遍历整个搜索空间，DFS，BFS访问的结点个数相同

14.4.3 八数码问题的DFS与BFS对比

- 八数码问题的DFS实现

```
1. #include <stdio.h>
2. #include <stdlib.h>

3. #define RepeatSize 10000000000 //重复标志数组的最大长度
4. #define TraceSize 1000         //操作轨迹数组的最大长度
5. #define DepthLimit 40          //最大的深度限制

6. typedef struct
7. {
8.     int code; //棋盘状态编码
9. } State;

10.char repeat[RepeatSize]; //重复标志数组
11.State trace[TraceSize];  //操作轨迹数组
12.int top;                 //当前记录的操作轨迹个数
13.int source, target;      //初始棋盘布局，目标棋盘布局
14.int depth = 0;           //当前搜索深度

15.void main() {
16.  State init;

17.  scanf("%d %d", &source, &target);
18.  init.code = source;
19.  set_repeat(init);
20.  dfs(init);
21.}
```

```
22.void dfs(State state) {
23.  State new_state;
24.  int i, j, k, new_i, new_j, from, zero, plate[3][3];
25.  trace[depth] = state;
26.  if (state.code==target) { output(state); exit(0); }
27.  if (depth > DepthLimit) return;  //最大深度限制
28.  decode(state.code, plate);
29.  zero = find_zero(plate);         //寻找0的位置
30.  i = zero / 3, j = zero % 3;
31.  for (k = 0; k < 4; k++) {
32.    new_i = i + direction[k][0]; //移动改变0所在行
33.    new_j = j + direction[k][1]; //移动改变0所在列
34.    if (new_i<0 || new_i>2 || new_j<0 || new_j>2)
35.      continue; //可行性约束：移动后0的位置必须在棋盘内
36.    swap(&plate[i][j], &plate[new_i][new_j]);
37.    new_state.code = encode(plate);
38.    new_state.from = from;
39.    if (!check_repeat(new_state)) { //判断重复状态
40.      set_repeat(new_state); //设置重复状态
41.      depth++;
42.      dfs(new_state);
43.      depth--;
44.    }
45.    swap(&plate[i][j], &plate[new_i][new_j]);
46.  }
47.}
```

14.4.3 八数码问题的DFS与BFS对比

	不限深度		限制深度<=10		限制深度<=20		限制深度<=40	
	访问结点数	路径长度	访问结点数	路径长度	访问结点数	路径长度	访问结点数	路径长度
DFS	StackOverflow		52	5	1241	15	48127	35
BFS	20	5	20	5	20	5	20	5

- 解在搜索树中分布不均匀的情况下，DFS如果误入歧途则无法求得可行解
- 可以通过限制遍历深度的方式，在一定程度上减少DFS误入歧途的风险
- 但是限制遍历深度会使得所搜索变得不完备，即有可能遗漏可行解
- DFS不能保证找到的第一个解一定是深度最小的解，BFS则可以保证

14.5 例题讲解

- YOJ-122：迷宫
- YOJ-124：滑雪
- YOJ-126：Travel
- YOJ-117：摘桃子
- YOJ-513：加法表达式（进阶版）

YOJ-122：迷宫

题目描述

给定一个由0（表示墙壁）和1（表示道路）的迷宫，请你判断进入迷宫后，仅通过横向和纵向的行走是否能从迷宫中走出来，即能否从坐标（1，1）走到（n，m）。

输入格式

共n+1行。
第一行为两个数n，m（1≤n，m≤9），表示迷宫的长和宽。
第2行到第n+1行，每行m个数，为一个由0和1组成的n*m的矩阵，表示迷宫，其中0表示墙壁（不通），1表示道路（坐标（1，1）和坐标（n，m）处都为1）。

输出格式

仅一行。
若该迷宫存在从坐标（1，1）到坐标（n，m）的由数字1组成的通路，则输出YES，否则输出NO。

输入样例

```
5 5
1 1 1 0 1
0 0 1 0 1
1 1 1 1 1
1 0 1 0 0
1 0 1 1 1
```

输出样例

```
YES
```

YOJ-122：迷宫

- 解题思路
  - 状态：当前所在的位置坐标
  - 状态转移：尝试上下左右四个方向到达新位置
    - 新位置必须在矩阵内
    - 新位置的值必须为1（道路）而不能是0（墙壁）
    - 新位置没有被访问过
- 进阶挑战：输出最短的路径及其长度
  - 变量len：记录当前的路径长度
  - 数组trace：记录当前的路径细节
  - 变量min_len：记录目前找到的最短路径长度，初始值设置为INT_MAX
  - 数组min_trace：记录目前找到的最短路径细节

YOJ-122：迷宫

```
1. #include <stdio.h>
2. #include <stdlib.h>
3. #include <limits.h>
4. #define N 10
5. #define TRACE_LEN 1000

6. typedef struct { int i, j; } Position;  //路径坐标结构
7. int map[N][N];                  //记录地图上的障碍物
8. int used[N][N] = {0};           //记录单元格是否走过
9. int n, m, found;
10.int len, min_len = INT_MAX;
11.Position trace[TRACE_LEN], min_trace[TRACE_LEN];
12.int direction[4][2] = {         //方向向量
13.  {-1, 0}, {1, 0}, {0, -1}, {0, 1}
14.};

15.void main() {
16.  int i, j, k;
17.  scanf("%d %d", &n, &m);
18.  for (i = 1; i <= n; i++)
19.    for (j = 1; j <= m; j++)
20.      scanf("%d", &map[i][j]);
21.  used[1][1] = 1;               //单元格(1,1)设置成走过
22.  dfs(1, 1);
23.  if (!found) { printf("NO\n"); exit(0); }
24.  printf("min_len: %d\n", min_len);
25.  for (k = 0; k <= min_len; k++)  //输出最短路径的细节
26.    printf("(%d,%d) ",min_trace[k].i,min_trace[k].j);
27.}
```

```
28.void dfs(int i, int j) {
29.  int k, new_i, new_j;
30.  trace[len].i=i, trace[len].j=j; //记录最短路径细节

31.  if (i == n && j == m)
32.  {
33.    found = 1;
34.    if (len < min_len) {//若当前路径长度比已求的最短路径短
35.      min_len = len;    //更新最短路径长度
36.      for (k = 0; k <= len; k++) //更新最短路径细节
37.        min_trace[k] = trace[k];
38.    }
39.  }
40.  for (k = 0; k < 4; k++) //尝试4个方向
41.  {
42.    new_i = i + direction[k][0];
43.    new_j = j + direction[k][1];
44.    //判断目标单元格在界内，并且没有路障，并且未被走过
45.    if (new_i>=1 && new_i<=n && new_j>=1 && new_j<=m
46.        && map[new_i][new_j] && !used[new_i][new_j])
47.    {
48.      used[new_i][new_j] = 1; //设置单元格走过
49.      len++;
50.      dfs(new_i, new_j);
51.      used[new_i][new_j] = 0; //设置单元格未走过
52.      len--;
53.    }
54.  }
55.}
```

YOJ-124：滑雪

输入样例	输出样例
5 5 1 2 3 4 5 16 17 18 19 6 15 24 25 20 7 14 23 22 21 8 13 12 11 10 9	25

题目描述

小袁非常喜欢滑雪， 因为滑雪很刺激。为了获得速度，滑的区域必须向下倾斜，而且当你滑到坡底，你不得不再次走上坡或者等待升降机来载你。小袁想知道在某个区域中最长的一个滑坡。区域由一个二维数组给出。数组的每个数字代表点的高度。如下：

1	2	3	4	5
16	17	18	19	6
15	24	25	20	7
14	23	22	21	8
13	12	11	10	9

一个人可以从某个点滑向上下左右相邻四个点之一，当且仅当高度减小。在上面的例子中，一条可滑行的滑坡为24-17-16-1。当然25-24-23-...-3-2-1更长。事实上，这是最长的一条。

你的任务就是找到最长的一条滑坡，并且将滑坡的长度输出。

滑坡的长度定义为经过点的个数，例如滑坡24-17-16-1的长度是4。

输入格式

输入的第一行表示区域的行数R和列数C(1<=R, C<=50)。下面是R行，每行有C个整数，依次是每个点的高度h（0<= h <=10000）。

输出格式

只有一行，为一个整数，即最长区域的长度。

思考：len作为状态的一部分，为什么在YOJ-124的实现代码里不需要在回溯时恢复其值？而在YOJ-122的实现代码里需要？

YOJ-124：滑雪

- 解题思路
  - 递归边界：当前格子无任何合法走向即返回
  - 可行性隐约束：下一个单元格在合法矩形区域内，且数值比当前单元格小
  - 最优化隐约束：必须要走到递归边界才能判断是否是最优
  - 初始状态：遍历矩形内的每一个位置作为初始位置

```
1. #include <stdio.h>
2. #define N 50

3. int R, C, area[N][N];

4. int main() {
5.   int i, j;

6.   scanf("%d %d", &R, &C);
7.   for (i = 0; i < R; i++)
8.     for (j = 0; j < C; j++)
9.       scanf("%d", &area[i][j]);
10.  for (i = 0; i < R; i++)
11.    for (j = 0; j < C; j++)
12.      dfs(i, j, 1);

13.  printf("%d", max_len);
14.}
```

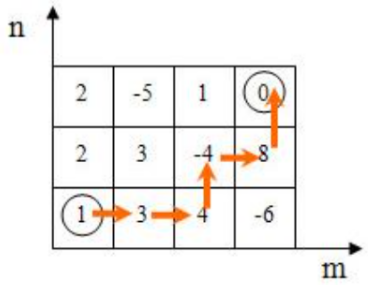
```
15.void dfs(int i, int j, int len) {
16.  int k, new_i, new_j;
17.  int stop = 1; //是否无任何合法走向，初始设置为1

18.  for (k = 0; k < 4; k++) {  //遍历4个方向
19.    new_i = i + direction[k][0];
20.    new_j = j + direction[k][1];
21.    if (new_i>=0 && new_i<R && new_j>=0 && new_j<C
22.        && area[new_i][new_j] < area[i][j]) {
23.      stop = 0; //有合法走向，则设置stop为0
24.      dfs(new_i, new_j, len+1); //递归进入下一个状态
25.    }
26.  }

27.  if (stop) { //stop代表到达递归边界，判断此时的解是否更优
28.    if (len > max_len) max_len = len;
29.  }
30.}
```

YOJ-126：Travel

输入样例	输出样例
3 4 1 3 4 -6 2 3 -4 8 2 -5 1 0	12

题目描述

有一个n*m的棋盘，如图所示，骑士X最开始站在方格(1,1)中，目的地是方格(n,m)。他的每次都只能移动到上、左、右相邻的任意一个方格。每个方格中都有一定数量的宝物k（可能为负），对于任意方格，骑士X能且只能经过最多1次（因此从(1,1)点出发后就不能再回到该点了）。



你的任务是，帮助骑士X从(1,1)点移动到(n,m)点，且使得他获得的宝物数最多。

输入格式

输入共有n+1行。
第一行为两个整数n和m（1 <= n, m <= 8 ）。
接下来n行，每行有m个整数，每2个整数之间由一个空格分隔。第i+1行第j个整数表示方格(i，j)中的宝物数目k（-100 <= k <= 100）。

输出格式

输出数据仅一个整数，即为骑士获得的宝物数。

YOJ-126：Travel

- 基本思想：深度优先搜索（递归回溯）
  - 状态：当前位置，宝藏值，单元格访问历史
  - 状态转移：
    - 在(i,j）单元格里有3个可能的走向：(i+1,j)，(i,j-1)，(i,j+1)
    - 依次选择一个合法的走向：新位置必须在矩形内且未曾访问过
  - 初始状态：当前位置为(1,1)，宝藏值为0
  - 递归边界：(i==n && j==m)
- 注意事项：
  - 回溯时，需要设置去往的单元格未曾访问过
  - 回溯时，宝藏的数值要进行恢复

YOJ-126：Travel

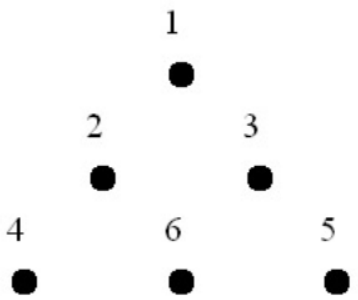
```
1. #include <stdio.h>
2. #include <limits.h>

3. int treasure[9][9];
4. int visited[9][9];
5. int n, m;
6. int max = INT_MIN;
7. int direction[3][2] = {    //方向向量
8.   {1, 0}, {0, -1}, {0, 1}
9. };

10.int main()
11.{
12.    int i, j;
13.    scanf("%d %d", &n, &m);

14.    for (i = 1; i <= n; i++)
15.        for (j = 1; j <= m; j++)
16.            scanf("%d", &treasure[i][j]);

17.    dfs(1, 1, 0);
18.    printf("%d\n", max);
19.}
```

```
20.void dfs(int i, int j, int value)
21.{
22.  int k, new_i, new_j;

23.  value = value + treasure[i][j];
24.  visited[i][j] = 1;
25.  if (i == n && j == m)
26.  {
27.    if (value > max) max = value;
28.    visited[i][j] = 0;
29.    return;
30.  }

31.  for (k = 0; k < 3; k++)
32.  {
33.    new_i = i + direction[k][0];
34.    new_j = j + direction[k][1];
35.    if (new_i<=n && new_j>=1 && new_j<=m
36.        && !visited[new_i][new_j])
37.      dfs(new_i, new_j, value);
38.  }
39.  visited[i][j] = 0;
40.}
```

YOJ-117：摘桃子

题目描述

一只小猴子正在到处觅食，突然他发现一颗长满桃子的树，小猴子特别高兴就决定爬上去摘桃子吃。 这颗长满桃子的树很大，共有n层（最高层为第1层），第i层有i条树枝，树的形状呈一个三角形（如图）：



图中的点表示树枝，每个点上方的数字当前这条树枝最多能摘到的桃子数（小于100），在摘得某枝条的桃子之后，小猴子只能选择往左上方爬或者是往右上方爬（也就是说在摘了有6个桃子的树枝之后只能摘有2个桃子的树枝或是有3个桃子的树枝），然后继续摘桃子。小猴子现在想要从最低层开始一直爬到树顶（也就是最上面的那个枝条），摘尽可能多的桃子。请你编程帮他解决这个问题。

输入格式

包含n+1行，第一行是一个整数n（1<=n<=100），表示这颗树一共有n层。
第2~n+1行中，第i行有i个用空格隔开的整数，表示树上第i层中每条树枝上的桃子数。

输出格式

共有一行，包括一个整数，表示小猴子能摘到的最多的桃子数。

输入样例

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

输出样例

```
10
```

另一组测试数据

输入:

```
5
45
76 32
45 99 22
33 44 55 66
24 75 32 64 75
```

输出:

```
339
```

YOJ-117：摘桃子

- 解题思路
  - 状态：当前所在位置坐标、沿当前路径到当前位置所收集到的桃子数
  - 状态转移：只能往左上或右上移动，移动时更新位置和收集到的桃子数
  - 记忆化剪枝：如果当前搜索路径到达某个位置时所收集到的桃子数，比历史上到达这个位置时所收集到的最多桃子数要少，就没必要沿这个路径继续搜索了

```
1. #include <stdio.h>

2. int peach[100][100]; //记录每个位置的桃子数
3. int direction[2] = {0, 1}; //方向向量
4. int max_log[100][100];//记录到达各个位置收集到的最多桃子数
5. int max_sum = 0; //记录最优解
6. int n;

7. void main() {
8.     scanf("%d", &n);
9.     for (int i = 1; i <= n; i++)
10.        for (int j = 1; j <= i; j++)
11.            scanf("%d", &peach[i][j]);
12.    dfs(1, 1, 0);
13.    printf("%d\n", max_sum);
14.}
```

```
15.void dfs(int i, int j, int sum) {
16.  int new_j;
17.  if (i > n) { //递归边界
18.    if (sum > max_sum) max_sum = sum;  //更新最优解
19.  }
20.  else {
21.    //最优化隐约束：到达当前位置所收集到的桃子数必须史上最优
22.    if (sum + peach[i][j] <= max_log[i][j]) return;
23.    sum += peach[i][j];
24.    max_log[i][j] = sum; //更新到达(i,j)位置的最多桃子数
25.    for (int k = 0; k < 2; k++) {//往左上或右上移动
26.      new_j = j + direction[k];
27.      dfs(i + 1, j_t, sum);
28.    }
29.  }
30.}
```

YOJ-513：加法表达式（进阶版）

【问题描述】

给定一个数字字符串S，在其中添加m个加号使其形成一个加法表达式。遍历所有可以构成合法加法表达式的加号添加方案，计算这些合法加法表达式的值，输出最大的表达式值。

例如：对于数字字符串12345，添加3个加号可以构成的加法表达式包括：

a. 1+2+3+45=51

b. 1+2+34+5=42

c. 1+23+4+5=33

d. 12+3+4+5=24

输出最大的表达式结果是：51

【输入格式】

一行，包含一个整数m，和一个长度不超过15且只含有数字的字符串S，m与S之间用空格分隔。

【输出格式】

一行，一个整数，代表最大的表达式结果。

【输入样例】

3 12345

【输出样例】

51

【数据规模】

1<=m<strlen(S)<=15

YOJ-513：加法表达式（进阶版）

- 基本思路：
  - 长度为n的字符串，有n-1个可以插入加号的位置
  - 递归枚举第m个加号的位置，第k个加号的位置须在第k-1个加号的位置之后，且须保证之后的m-k个加号还有可插入的位置
  - 递归边界k==m，到达递归边界后，使用“数位求和”的基本功能代码，把m个加号拆分出来的m+1个数字识别出来，相加得到的值与当前最大值进行比较
  - 状态表示：k表示当前正在枚举第几个加号；s表示
- 注意事项：
  - m个加号会拆分出m+1个数，记得要加上最后一个数
  - 给第k个加号设置插入位置时，要保证之后的m-k个加号还有可插入位置

YOJ-513：加法表达式（进阶版）

```
1. #include <stdio.h>
2. #include <string.h>
3. #define MAX_LEN 16

4. char s[MAX_LEN];
5. int m, n, ary[MAX_LEN];   //ary记录加号放到第几个字符后面
6. long long max = 0;

7. long long get_value()
8. {
9.     int i, j, start = 0;
10.    long long item, base, sum;

11.    sum = 0;
12.    ary[m + 1] = n - 1; //最后一个数字一直截取到n-1位置
13.    for (i = 1; i <= m + 1; i++)   //数位合并
14.    {
15.        item = 0, base = 1;
16.        for (j = ary[i]; j >= start; j--)
17.        {
18.            item += (s[j] - '0') * base;
19.            base *= 10;
20.        }
21.        sum += item;
22.        start = ary[i] + 1;
23.    }
24.    return sum;
25.}
```

```
26.void dfs(int k, int s, int e)   //从位置s到e插入第k个加号
27.{
28.    int i;
29.    long long value;

30.    if (k > m)   //递归边界：已插入m个加号
31.    {
32.        value = get_value();
33.        if (value > max) max = value;   //更新最大值
34.        return;
35.    }

36.    for (i = s; i <= e; i++)//从位置s到e穷举第k个加号位置
37.    {
38.        ary[k] = i;
39.        dfs(k + 1, i + 1, n - m + k - 1);
40.    }
41.}

42.void main()
43.{
44.    scanf("%d %s", &m, s);

45.    n = strlen(s);

46.    dfs(1, 0, n - m - 1);
47.    printf("%lld ", max);
48.}
```

放第k+1个加号，要保证最后还剩m-(k+1)+1个字符，才够放余下加号。故第k+1个加号最多只能放到n-m+k-1个字符后面。

2021、2022期中期末考试例题讲解

- YOJ-892：单人棋（22程设I荣誉课程期末题）
- YOJ-766：degree（21程设I荣誉课程期末题）
- YOJ-851：移动火柴2.0（22程设I荣誉课程期中题）

YOJ-851：移动火柴2.0（22程设I荣誉课程期中题）

【问题描述】

玩了一会游戏，小 C 突然很有负罪感，马上就要期中考试了，还是得多刷一刷题。于是小 C 打开了某 OJ，一眼就看到了“移动火柴”这题，小 C 很高兴，这题我做过啊，于是 Ctrl+C，Ctrl+V 喜提 WA，小 C 定睛一看，才发现这道题暗藏玄机……

题面如下：

移动一根火柴使得给定的形如“A+B=C”或者“A-B=C”的算术运算式成立，但是存在 **8 进制，10 进制以及 16 进制**的火柴，并且 A/B/C 代表的是数字字符串，而非单个数字字符。

火柴示意图如下：

注意：

1. 可以移动数字上的火柴，但需保证移动后所有的数字仍是有效的数字；
2. +运算符可以移走一根火柴，变成-运算符；还可以改动一根火柴，变成=运算符；
3. -运算符可以添加一根火柴，变成+运算符和=运算符；
4. =运算符的火柴可以移走一根火柴变成-运算符，**但你应该保证等式中有且仅有一个等号；**
5. 不能在数字前面添加火柴，使其变为负数；
6. **移动后三个数在对应进制下仍然合法；**
7. 若移动后火柴组成的数字含前导 0 属于**合法**情况，且**前导零必须输出**；
8. 输入输出中涉及到的字母均为**大写**，保证输入不含前导零。

数字火柴的所有可行移动方案：

数字	移走一根	添加一根	自身变换
0	N/A	8	6, 9, A
1	N/A	7	N/A
2	N/A	A	3, D, E
3	N/A	9, A	2, 5, D
4	N/A	N/A	N/A
5	N/A	6, 9	3, B, E
6	5, B, E	8	0, 9, A
7	1	N/A	N/A
8	0, 6, 9, A	N/A	N/A
9	3, 5	8	0, 6, A
A	2, 3, D	8	0, 6, 9
B	N/A	6	5, D, E
C	N/A	E	F
D	N/A	A	2, 3, B
E	C, F	6	2, 5, B
F	N/A	E	C

【输入格式】

第一行三个数，依次表示A, B, C**三个数**的进制。

第二行一个字符串，形如“A+B=C”或者“A-B=C”的算术运算式。

【输出格式】

输出移动一根火柴后成立的等式，请保证你的输出有且仅有一个等号，所有测试用例均保证有解。

【数据范围】

对于 50%的数据，保证均为 10 进制数，且$0 \le A, B, C \le 9$。

对于 100%的数据，保证$0 \le A, B, C \le 100$，输入仅包含数字和 A,B,C,D,E,F。

注意：

1. 可以移动数字上的火柴，但需保证移动后所有的数字仍是有效的数字；
2. +运算符可以移走一根火柴，变成-运算符；还可以改动一根火柴，变成=运算符；
3. -运算符可以添加一根火柴，变成+运算符和=运算符；
4. =运算符的火柴可以移走一根火柴变成-运算符，**但你应该保证等式中有且仅有一个等号；**
5. 不能在数字前面添加火柴，使其变为负数；
6. **移动后三个数在对应进制下仍然合法；**
7. 若移动后火柴组成的数字含前导 0 属于**合法**情况，且**前导零必须输出**；
8. 输入输出中涉及到的字母均为**大写，**保证输入不含前导零**。**

数字火柴的所有可行移动方案：

数字	移走一根	添加一根	自身变换
0	N/A	8	6, 9, A
1	N/A	7	N/A
2	N/A	A	3, D, E
3	N/A	9, A	2, 5, D
4	N/A	N/A	N/A
5	N/A	6, 9	3, B, E
6	5, B, E	8	0, 9, A
7	1	N/A	N/A
8	0, 6, 9, A	N/A	N/A
9	3, 5	8	0, 6, A
A	2, 3, D	8	0, 6, 9
B	N/A	6	5, D, E
C	N/A	E	F
D	N/A	A	2, 3, B
E	C, F	6	2, 5, B
F	N/A	E	C

与之前不同的地方，

1.多位数字

2.包含进制要求

3.可以移动等号

Solution

- 我们枚举移动的火柴是哪一根来得到最后的结果，我的方法是枚举三种情况
  1. 移动的是等号上的火柴，由于等式中一定要存在一根火柴，那么仅当运算符是减号时可以移动。
  2. 火柴变化涉及一个位置
  3. 火柴变化涉及两个位置

一些预处理

```
81  bool check(int * l, int * r, char * str, int * base)
82  {
83      int num[3] = {0};
84      char * endptr = NULL;
85      for (int i = 0; i < 3; i++) {
86          num[i] = strtol(&str[l[i]], &endptr, base[i]);
87          if (endptr - str != r[i]) return false;
88      }
89      int eqnum = (str[r[0]] == '=') + (str[r[1]] == '=');
90      if (eqnum != 1) return false;
91
92      if (str[r[0]] == '-' && num[0] - num[1] == num[2]) {
93          return true;
94      }
95
96      if (str[r[0]] == '+' && num[0] + num[1] == num[2]) {
97          return true;
98      }
99
100     if (str[r[0]] == '=' && num[0] == num[1] - num[2]) {
101         return true;
102     }
103
104     return false;
105 }
```

检查str串是否是一个合法的等式，以及是否符合进制的要求(base)
最后的等式一共三种情况

1. A'-B'=C'
2. A'+B'=C'
3. A'=B'-C'

一些预处理

```
10 char remove1[][16] = {
11     {0},
12     {0},
13     {0},
14     {0},
15     {0},
16     {0},
17     {'5','B','E', 0},
18     {'1', 0},
19     {'0', '6', '9', 'A', 0},
20     {'3', '5', 0},
21     {'2', '3', 'D', 0},
22     {0},
23     {0},
24     {0},
25     {'C', 'F', 0},
26     {0},
27     {'-', 0},    // '+'
28     {0}          // '-'
29 };
```

```
31 char insert1[][16] = {
32     {'8', 0},
33     {'7', 0},
34     {'A', 0},
35     {'9', 'A', 0},
36     {0},
37     {'6', '9', 0},
38     {'8', 0},
39     {0},
40     {0},
41     {'8', 0},
42     {'8', 0},
43     {'6', 0},
44     {'E', 0},
45     {'A', 0},
46     {'6', 0},
47     {'E', 0},
48     {0},              // '+'
49     {'+', 0}          // '-'
50 };
```

```
52 char trans[][16] = {
53     {'6', '9', 'A', 0},
54     {0},
55     {'3', 'D', 'E', 0},
56     {'2', '5', 'D', 0},
57     {0},
58     {'3', 'B', 'E', 0},
59     {'0', '9', 'A', 0},
60     {0},
61     {0},
62     {'0', '6', 'A', 0},
63     {'0', '6', '9', 0},
64     {'5', 'D', 'E', 0},
65     {'F', 0},
66     {'2', '3', 'B', 0},
67     {'2', '5', 'B', 0},
68     {'C', 0}
69 };
```

情况一：移动等号上的火柴

- 那么最后的等式一定形如A'=B'-C'
- 我们直接模拟火柴的移动，然后调用之前的check函数即可

```
107 void SOLVE0(int * l, int * r, char * str, int * base)
108 {
109     if (str[r[0]] == '+') return ; // 只有减法式子可以移动等号
110 //     char * endptr;
111 //     assert(str[r[0] == '-']);
112     str[r[0]] = '=';
113     str[r[1]] = '-';
114     if (check(l, r, str, base)) {
115         puts(str);
116     }
117     str[r[1]] = '=';
118     str[r[0]] = '-';
119 }
```

情况二：火柴移动只涉及一个位置

- 那么就是一个位置上的数位自身的变化，只涉及到trans数组

```
121 void SOLVE1(int * l, int * r, char * str, int * base)
122 {
123     int len = strlen(str);
124     for (int i = 0; i < len; i++) { //枚举移动的位置
125         if (str[i] == '+' || str[i] == '-' || str[i] == '=') continue;
126         char tmp = str[i];
127         int idx = get_idx(tmp);
128         int ways = strlen(trans[idx]);
129         for (int j = 0; j < ways; j++) { //枚举当前位置上的数字变换成哪一种数字
130             str[i] = trans[idx][j];       // 根据我们的枚举，修改字符串
131             if (check(l, r, str, base))  // 调用check函数来判断修改后的字符串是否为合法的答案
132                 puts(str);
133             str[i] = tmp;                 // 还原字符串用于下一次的枚举
134         }
135     }
136 }
```

情况三：火柴移动涉及两个位置

- 我们枚举哪个位置上的火柴被移走，以及移走后变成了什么
- 之后再枚举移动后的火柴移到了哪个位置，以及移来后变成了什么

```
137 void SOLVE2(int * l, int * r, char * str, int * base)
138 {
139     int len = strlen(str);
140     for (int i = 0; i < len; i++) {//枚举哪个位置上的火柴发生移动
141         int idx = get_idx(str[i]);
142         char tmpi = str[i];
143         for (int ii = 0; ii < strlen(remove1[idx]); ii++) { //火柴移走后当前数位变成了什么
144             str[i] = remove1[idx][ii];          // 修改字符串
145             for (int j = 0; j < len; j++) if (i != j) {//枚举这个火柴最后移动到了哪个位置
146                 int jdx = get_idx(str[j]);
147                 char tmpj = str[j];
148                 for (int jj = 0; jj < strlen(insert1[jdx]); jj++) { //火柴移过来之后数位变成了什么
149                     str[j] = insert1[jdx][jj];  // 修改字符串
150                     if (check(l, r, str, base)) // 检查该字符串是否合法
151                         puts(str);
152                     str[j] = tmpj;
153                 }
154             }
155             str[i] = tmpi;
156         }
157     }
158 }
```

YOJ-892：单人棋（22程设I荣誉课程期末题）

【问题描述】

在炼丹之余为了打发时间，小C玩起了一个名叫“单人棋”的小游戏，游戏规则是这样的：有一个$n\times m$的棋盘，棋盘上分布着若干个棋子。

分别给出棋盘的初始状态与目标状态，小C需要通过移动棋子将棋盘由初始状态变为目标状态，在每一轮中，小C选择棋盘上任意一个棋子进行移动，规则如下：

- 棋子可以跳过与之相邻（只包含左右上下四个方向）的棋子走到后面的空格上，被它跳过的棋子从棋盘上移走。若相邻棋子的另一侧有棋子，则不能跳。棋子只能在棋盘内移动。
- 在每一轮中，你可以重复1中的操作任意次。

所有棋子都是本质相同的。小C想要知道，最少需要几轮才能将棋盘由初始状态变为目标状态，保证存在解。

【输入格式】

第一行两个有空格分隔的整数n, m，表示棋盘的大小。

接下来n行描述起始棋盘，第i行包含一个长度为m、只包含01的字符串map_i；若$map[i][j]=1$则表示第i行第j列有一个棋子，反之则该处无棋子。

接下来n行描述目标棋盘，格式同上。

【输出格式】

第一行一个整数k，表示将起始棋盘变为目标棋盘的最少轮数。

接下来k行，每行至少三个用空格分隔的整数，表示该轮的移动方案，每一行的格式形如$x\ y\ num\ d_1d_2...d_{num}$

- 前两个整数x, y表示该轮选择移动的棋子位置
- num表示该轮移动了选定棋子多少次
- 序列d依次表示选定棋子的移动方向，$d_i\in\{0,1,2,3\}$。$d_i=0,1,2,3$时分别对应第i次向左,右,上,下移动

整体思路：记忆化搜索

介绍：记忆化搜索不是什么高级的算法，只是用一个数组储存某个状态的值，这样一来以后每次搜索到这个状态时就直接返回已经储存在数组中的值

适用情景：状态数不是很多，但是重复的状态经常被搜索到（观察一下发现这道题就十分符合这一种情况）

时间复杂度：O(n*m) n代表数组的大小，m表示分支的个数

By陈礼邦

辅助工具1：状态压缩

状态压缩就是把bool数组压缩成一个int或者long long。
比如bool a[5]; int b;
其中a[0]=0,a[1]=1,a[2]=1,a[3]=0,a[4]=1;
压缩完就是b=(10110)_2 = 22

操作方式：
1.b=b<<1为b在二进制下整体左移一位b=(101100)_2
2.b=b>>1为b在二进制下整体左移一位b=(1011)_2
3.b>>i&1判断b的第i位时0还是1
4.b|=1<<i 把b的第i位变成1

辅助工具2：哈希表

因为状态压缩之后太大，不能直接用数组存，因此我们给每个很大的数都赋值一个很小的数，使其能够用数组存下，为了快速查找和插入，就可以用哈希表。

```
int tot,hd[mod+3],nxt[M];ll num[M];
int id(ll now){
    int x=now%mod;
    for(int i=hd[x];i;i=nxt[i]){
        if(num[i]==now)return i;
    }
    num[++tot]=now;
    nxt[tot]=hd[x];
    hd[x]=tot;
    return tot;
}
```

statu(long long)表示棋盘的状态
pos表示当前使第几个棋子（因为棋子不超过20个）
f[statu][pos]表示还要走几步

YOJ-766：degree（21程设I荣誉课程期末题）

四、十步万度

【问题描述】

“十步万度”是一款非常流行的益智类游戏。游戏规则简述如下：在一个n*n矩阵的每个单元格里放置一个表盘，表盘的指针可以指向上、右、下、左四个方向。允许玩家进行m步旋转，每一步从矩阵中任选一个单元格，对放置其中的表盘进行如下操作：将表盘指针按顺时针方向旋转90度，后续被指向的单元格中的表盘依次进行旋转，如此反复直至指针指向的位置不在矩阵范围内，便完成了一步旋转操作。在旋转过程中，累计表盘转动度数之和。

请编程求解如下问题：给定一个n*n矩阵上每个表盘指针的初始方向，求m步旋转操作完成后，所有表盘转动度数之和的最大值。

【输入格式】

第一行两个整数n，m，分别代表矩阵的阶和转动次数。

接下来的n行，每行n个数字，代表表盘指针的初始方向。其中，0代表向上、1代表向右、2代表向下、3代表向左。

【输出格式】

输出一行整数，表示所有表盘转动度数之和的最大值。

【输入样例】

```
3 2
1 0 0
0 1 0
0 0 0
```

【输出样例】

```
810
```

YOJ-766:degree

- 思路:
  - 通过递归暴力求解
  - 优化?

By卢虹宇

YOJ-766:degree

```c
#include <stdio.h>
#include <math.h>
#include <string.h>
int n, m, map[10][10], max;
// n为矩阵的阶，m为转动次数，map为表盘矩阵，max用于记录最大步数;
int turn(int x, int y)
// turn返回值为转动（x，y）的表盘所引起的度数变化;
{
    int sum = 90; // 最初传入值时已经转动了一次;
    map[x][y] = (map[x][y] + 1) % 4;
    switch (map[x][y])
    {
    case 0:
        if (x - 1 >= 0)
        {
            sum = turn(x - 1, y) + 90;
        }
        break;
    case 1:
        if (y + 1 < n)
        {
            sum = turn(x, y + 1) + 90;
        }
        break;
    case 2:
        if (x + 1 < n)
        {
            sum = turn(x + 1, y) + 90;
        }
        break;
    case 3:
        if (y - 1 >= 0)
        {
            sum = turn(x, y - 1, map) + 90;
        }
        break;
    }

    return sum;
}
void mapcpy(int map1[10][10], int map2[10][10])
// 用于复制表盘矩阵;
{
    for (int k = 0; k < n; k++)
    {
        for (int v = 0; v < n; v++)
        {
            map1[k][v] = map2[k][v];
        }
    }
}
```

YOJ-766:degree

```
void find(int k, int sum) // 用于找到转动k次后表盘转动度数之和的最大值;
{
    if (k == 0)
    {

        max = (sum > max) ? sum : max; // 找到最大值

        return;
    }

    for (int w = 0; w < n; w++)
    {
        for (int v = 0; v < n; v++)
        {
            int temmap[10][10]; // 局部变量
            mapcpy(temmap, map); // 储存当前表盘状态;
            int h = turn(w,v);
            find(k - 1, sum + h);
            mapcpy(map, temmap); // 恢复表盘状态
        }
    }
    return;
}
int main()
{
    scanf("%d %d", &n, &m);
    for (int k = 0; k < n; k++)
    {
        for (int v = 0; v < n; v++)
        {
            scanf("%d", &map[k][v]);
        }
    }
    find(m, 0);
    printf("%d", max);
}
```