14. 搜索

14.1 DFS

14.1.1 全排列

题意

给定正整数n $(1 \leq n \leq 7)$, 按字典序输出数字$1 \sim n$的全排列.

思路I

n个空位从左往右填, 每次填的数不能与前面相同, 用一个bool数组$vis[]$记录哪些数被用过.

DFS记录当前填到哪一位, 回溯时注意恢复所用的位置.

代码I

```cpp
int n;
vector<int> ans;
vector<bool> vis;

void dfs(int pos) {
    if (pos == n) {
        for (auto ai : ans)
            cout << ai << " \n"[ai == ans.back()];
        return;
    }

    for (int i = 1; i <= n; i++) {
        if (!vis[i]) {
            ans[pos] = i;
            vis[i] = true;
            dfs(pos + 1);
            vis[i] = false;
        }
    }
}

void solve() {
    cin >> n;

    ans.resize(n), vis.resize(n + 1);
    dfs(0);
}

int main() {
    solve();
}
```

思路II

注意到可用一个n位二进制数的每个数位记录每个数是否用过,1表示用过,0表示没用过,则无需额外开一个bool数组记录.

代码II

```cpp
const int MAXN = 10;
int n;  // 数字个数
int path[MAXN];  // 存搜索树的路径

void dfs(int pos, int state) {  // 当前已填完前pos位且状态为state
  if (pos == n) {  // 填完一个排列就输出
    for (int i = 0; i < n; i++) cout << path[i] << ' ';
    cout << endl;
    return;
  }

  for (int i = 0; i < n; i++) {  // 枚举哪个数还没用
    int u = state >> i & 1;  // 取出state的二进制表示的第i位
    if (!u) {
      path[pos] = i + 1;  // 填上该数,注意从0开始故要+1
      dfs(pos + 1, state + (1 << i));  // 将状态的第i位置为1,搜下一个位置
    }
  }
}

int main() {
  cin >> n;

  dfs(0, 0);
}
```

14.1.2 n皇后

题意

将n个皇后放在$n \times n$的棋盘上, 使得任一两皇后不在同一行、同一列、同一斜线上. 给定n $(1 \leq n \leq 9)$, 输出满足条件的摆法, 用'.'表示当前位置为空, 用'Q'表示当前位置有皇后, 每个方案输出完成后用一个空行分隔.

思路I

逐个枚举格子, 有放和不放两种选择, 枚举到第n^2个格子时即得到一个答案.

开两个数组$row[]$和$col[]$记录每个位置有没有放皇后.

注意斜线指对角线和反对角线, 开两个数组$dg[]$和$udg[]$分别记录对角线和反对角线上每个位置有没有皇后. $n \times n$的棋盘有$(2n-1)$条对角线, 故这两个数组要开数据范围的两倍. 点(row, i)对应$dg[row+i]$和$udg[n-row+i]$.

总时间复杂度$O(2^{n^2})$.

代码I

```cpp
const int MAXN = 10;
int n;  // 数字个数
bool row[MAXN], col[MAXN];  // 记录每个位置是否有皇后
bool dg[MAXN << 1], udg[MAXN << 1];  // 记录对角线、反对角线有无皇后,注意大小开2倍
char ans[MAXN][MAXN];  // 合法的方案

void dfs(int x, int y, int cnt) {  // 当前填到(x,y),已经放了cnt个皇后
  if (y == n) y = 0, x++;  // 下一行

  if (x == n) {
    if (cnt == n) {  // 放完,输出解
      for (int i = 0; i < n; i++) puts(ans[i]);
      puts("");
    }
    return;
  }

  dfs(x, y + 1, cnt);  // 不放皇后

  // 放皇后
  if (!row[x] && !col[y] && !dg[x + y] && !udg[x - y + n]) {
    row[x] = col[y] = dg[x + y] = udg[x - y + n] = true;
    ans[x][y] = 'Q';
    dfs(x, y + 1, cnt + 1);  // 搜本行下一个位置
    row[x] = col[y] = dg[x + y] = udg[x - y + n] = false;
    ans[x][y] = '.';  // 复位
  }
}

int main() {
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++)
      ans[i][j] = '.';
  }

  dfs(0, 0, 0);
}
```

思路II

注意到每行只能有一个皇后, 类似于全排列, 从上往下枚举每一行, 对每一行枚举皇后能放的位置, 边放边检查是否冲突, 有冲突则直接回溯, 即剪枝. 故只需开一个数组$col[]$记录每列有没有放皇后, 对角线和反对角线同理**思路I**.

总时间复杂度$O(n \cdot n!)$.

代码II

```cpp
int n;
vector<bool> col;
vector<bool> diag, idiag;
vector<vector<char>> ans;

void dfs(int row) {
    if (row == n) {a
```

```
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++)
                cout << ans[i][j];
            cout << endl;
        }
        cout << endl;
        return;
    }

    for (int j = 0; j < n; j++) {  // 枚举列
        if (!col[j] && !diag[row + j] && !idiag[n - row + j]) {
            ans[row][j] = 'Q';
            col[j] = diag[row + j] = idiag[n - row + j] = true;
            dfs(row + 1);
            col[j] = diag[row + j] = idiag[n - row + j] = false;
            ans[row][j] = '.';
        }
    }
}

void solve() {
    cin >> n;

    col.resize(n + 1);
    diag.resize(n + 1 << 2), idiag.resize(n + 1 << 2);
    ans = vector<vector<char>>(n + 1, vector<char>(n + 1, '.'));

    dfs(0);
}

int main() {
    solve();
}
```

14.1.3 Bitwise Exclusive-OR Sequence

题意

现有n $(1 \leq n \leq 1e5)$个非负数$a_1, \cdots, a_n$,用m $(0 \leq m \leq 2e5)$个限制$a_u \ xor \ a_v = w$描述.若满足限制的序列存在,输出a_i $(i = 1, \cdots, n)$之和的最小值;否则输出-1.

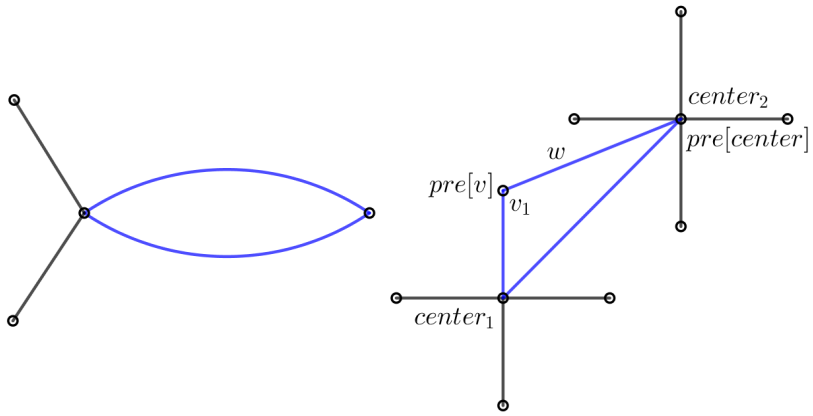
思路I

若$a_u \ xor \ a_v = w$,则在节点u和v间连一条权值为w的无向边.问题转化为求各节点点权之和的最小值.

注意到m条限制不能保证所连成的图是树,则可能出现重边和环:①出现重边时,若两重边权值不同,则无解;②出现环时,以三元环为例,设三个节点分别为A、B、C,且AB、BC、CA的边权分别为w_1, w_2, w_3.若满足限制的序列存在,应满足$w_1 \ ^ w_2 \ ^ w_3 = A \ ^ B \ ^ B \ ^ C \ ^ C \ ^ A = 0$,即环上的每个节点都会被异或两次.

遇到重边和环即判断是否无解,剩下的都是链.考察链$A_1 A_2 \cdots A_n$,设$A_1 \ ^ A_2 = w_1, A_2 \ ^ A_3 = w_2, \cdots, A_{n-1} \ ^ A_n = w_{n-1}$.注意到$A_2 = w_1 \ ^ A_1, A_3 = w_2 \ ^ A_2 = (w_1 \ ^ w_2) \ ^ A_1, \cdots, A_n = (w_1 \ ^ w_2 \ ^ \cdots \ ^ w_{n-1}) \ ^ A_1$,即$A_2, \cdots, A_n$由$A_1$唯一确定.而每一位的异或值仅与两数的该位数码是否相同有关,则可枚举A_1的二进制表示的数位.遍历链时维护链上的节点的前缀异或和,确定A_1后用其快速求出$A_2, \cdots, A_n$.

DFS.注意到确定节点u的点权后,与其连通的所有节点的点权也唯一确定,则DFS维护$a_1,\cdots,a_n$分别所在的连通块,每次遍历以节点$center$为中心的菊花图.若$center$的某条出边的终点v已被遍历过,则出现重边或自环,如下图:



对$a_1,\cdots,a_n$分别所在的连通块,考察它对答案的贡献.设第pos个连通块共tot个节点.枚举A_1的数位,统计tot个节点中对应数位为1的个数cnt.若tot个节点中该数位1的数量比0的数量多,为使$a_i\ \ (i=1,\cdots,n)$的总和最小,A_1的该数位应取1,否则取0.确定A_1的值后,用前缀异或和求出$A_2,\cdots,A_n$,它们之和即该连通块对答案的贡献.

菊花图的遍历还可用BFS.

代码I

```cpp
const int MAXN = 1e5 + 5;
int n, m;  // 节点数、限制数
vii graph[MAXN];  // graph[u]={v,w}
bool vis[MAXN];  // 记录每个节点是否已被遍历过
ll pre[MAXN];  // 每条链上的前缀异或和
vi to[MAXN];  // 与每个节点连通的点

void dfs(int pos, int center) {  // 遍历到第pos个节点所在的连通块,当前菊花图的中心为center
  vis[center] = true;
  for (auto item : graph[center]) {  // 枚举中心节点的所有出边
    int v = item.first, w = item.second;
    if (vis[v]) {  // 重边或成环
      if ((pre[center] ^ w) != pre[v]) {  // 检查边权异或值是否矛盾,注意位运算符优先级低
        cout << -1;
        exit(0);
      }
    }
    else {
      pre[v] = pre[center] ^ w;  // 维护前缀异或和
      to[pos].push_back(v);  // 该点可与第pos个节点连通
      dfs(pos, v);  // 搜以节点v为中心的菊花图
    }
  }
}

ll cal(int pos) {  // 计算第pos个节点所在的连通块对答案的贡献
  int tot = to[pos].size();  // 第pos个节点所在的连通块的节点数
  int a1 = 0;
  for (int i = 0; i < 30; i++) {  // 枚举a1的数位
    int cnt = 0;  // 统计与第pos个节点连通的节点的点权中第i位为1的节点数
    for (auto v : to[pos]) // 枚举与第pos个节点连通的节点
      if ((pre[v] >> i) & 1) cnt++;
    if (cnt > tot - cnt) a1 += (1 << i);  // 若1比0多,则a1的第i位取1可使总和最小
```

```
  }
  ll res = a1;
  for (auto v : to[pos]) res += pre[v] ^ a1;
  return res;
}

int main() {
  cin >> n >> m;
  while (m--) {
    int u, v, w; cin >> u >> v >> w;
    graph[u].push_back(make_pair(v, w)), graph[v].push_back(make_pair(u, w));
  }

  ll ans = 0;
  for (int i = 1; i <= n; i++) {
    if (!vis[i]) {
      dfs(i, i);  // 遍历到第i个节点所在的连通块,当前菊花图的中心为第i个节点
      ans += cal(i);  // 计算第i个节点所在的连通块对答案的贡献
    }
  }
  cout << ans;
}
```

代码II

```
const int MAXN = 1e5 + 5;
int n, m;  // 节点数、限制数
vii graph[MAXN];  // graph[u]={v,w}
bool vis[MAXN];  // 记录每个节点是否已被遍历过
ll pre[MAXN];  // 每条链上的前缀异或和
vi to[MAXN];  // 与每个节点连通的点

void bfs(int pos) {  // 遍历到第pos个连通块
  qi que;
  que.push(pos);
  vis[pos] = true;
  while (que.size()) {
    int u = que.front(); que.pop();
    for (auto item : graph[u]) {  // 枚举u的所有出边
      int v = item.first, w = item.second;
      if (vis[v]) {  // 出现重边或自环
        if ((pre[u] ^ w) != pre[v]) {  // 检查边权异或值是否矛盾
          cout << -1;
          exit(0);
        }
      }
      else {
        vis[v] = true;
        pre[v] = pre[u] ^ w;
        to[pos].push_back(v);
        que.push(v);
      }
    }
  }
}

```

```
ll cal(int pos) {  // 计算第pos个节点所在的连通块对答案的贡献
  int tot = to[pos].size();  // 第pos个节点所在的连通块的节点数
  int a1 = 0;
  for (int i = 0; i < 30; i++) {  // 枚举a1的数位
    int cnt = 0;  // 统计与第pos个节点连通的节点的点权中第i位为1的节点数
    for (auto v : to[pos]) // 枚举与第pos个节点连通的节点
      if ((pre[v] >> i) & 1) cnt++;
    if (cnt > tot - cnt) a1 += (1 << i);  // 若1比0多,则a1的第i位取1可使总和最小
  }
  ll res = a1;
  for (auto v : to[pos]) res += pre[v] ^ a1;
  return res;
}

int main() {
  cin >> n >> m;
  while (m--) {
    int u, v, w; cin >> u >> v >> w;
    graph[u].push_back(make_pair(v, w)), graph[v].push_back(make_pair(u, w));
  }

  ll ans = 0;
  for (int i = 1; i <= n; i++) {
    if (!vis[i]) {
      bfs(i);  // 遍历第i个节点所在的连通块
      ans += cal(i);  // 计算第i个节点所在的联通块对答案的贡献
    }
  }
  cout << ans;
}
```

思路II

对边权的每个数位维护一个连通块,其中边权的二进制表示中对应数位为1时连一条边权为1的边,否则连一条边权为0的边.因$w < 2^{30}$,则图中有30个连通块,分别对它们染色,每个节点染1或0,则每个连通块中1的个数的最小值乘对应的权值即为该数位对答案的贡献.

1的个数不确定的原因:将一张已染色的图颜色反转后仍满足条件.

代码III

```
const int MAXN = 1e5 + 5, MAXM = 2e5 + 5;
int n, m;  // 节点数、限制数
int head[30][MAXN], nxt[30][MAXM << 1], val[30][MAXM << 1], weight[30][MAXM << 1],
idx[MAXM << 1];  // 至多有30个连通块
int color[30][MAXN];
bool vis[30][MAXN];
int cnt = 0;  // 统计连通块中的节点数

void add(int pos, int u, int v, int w) {  // 第pos个连通块中建u<->v的权值为w的双向边
  val[pos][idx[pos]] = v;
  weight[pos][idx[pos]] = w;
  nxt[pos][idx[pos]] = head[pos][u];
  head[pos][u] = idx[pos]++;
}

```

```cpp
int dfs(int pos, int center, int c) {  // 将第pos个连通块中以center为中心的菊花图染成c色,返回连通块中1的个数
  color[pos][center] = c;
  int res = c;  // 统计第pos个连通块中1的个数
  for (int u = head[pos][center]; ~u; u = nxt[pos][u]) {
    int v = val[pos][u];
    if (vis[pos][v]) {  // 出现重边或自环
      if ((c ^ weight[pos][u]) != color[pos][v]) {  // 检查是否矛盾
        cout << -1;
        exit(0);
      }
    }
    else {
      cnt++;  // 更新该连通块中的节点数
      vis[pos][v] = true;
      res += dfs(pos, v, c ^ weight[pos][u]);  // 染以v为中心的菊花图
    }
  }
  return res;
}

ll cal(int pos) {  // 求第pos个连通块对答案的贡献
  ll res = 0;
  for (int i = 1; i <= n; i++) {
    if (vis[pos][i]) continue;

    vis[pos][i] = true;
    cnt = 1;  // 用于DFS时统计连通块中的节点数
    int tmp = dfs(pos, i, 1);  // 将连通块中的第一个节点染成1得到的整个连通块中1的个数
    res += min(tmp, cnt - tmp);  // cnt-tmp是将连通块的第一个节点染成0得到的连通块中1的个数
  }
  return res;
}

int main() {
  memset(head, -1, so(head));

  cin >> n >> m;
  while (m--) {
    int u, v, w; cin >> u >> v >> w;
    for (int i = 0; i < 30; i++) {  // 每个数位维护一个连通块
      if (w >> i & 1) add(i, u, v, 1), add(i, v, u, 1);  // 建边权为1的边
      else add(i, u, v, 0), add(i, v, u, 0);  // 建边权为0的边
    }
  }

  ll ans = 0;
  for (int i = 0; i < 30; i++) ans += cal(i) * (1 << i);  // 统计每个连通块对答案的贡献
  cout << ans;
}
```

思路III

对$a_u \ ^\wedge a_v = w$,考察w的每个数位:①若该数位为1,则a_u, a_v的该数位不同;②若该数位为0,则a_u, a_v的该数位相同.

按a_u, a_v的对应数位的异同分为两类,用带扩展域的并查集维护:①若a_u和a_v的对应数位相同,则合并$fa[u]$和$fa[v], fa[u+n]$和$fa[v+n]$;②若a_u和a_v的对应数位不同,则合并$fa[u]$和$fa[v+n], fa[u+n]$和$fa[v]$.

代码IV

```cpp
const int MAXN = 2e5 + 10;
int n, m;  // 节点数、限制数
struct Edge {
  int u, v, w;
}edges[MAXN];
int fa[MAXN], siz[MAXN];  // 并查集的fa数组、集合的大小

int find(int x) { return x == fa[x] ? x : fa[x] = find(fa[x]); }

void init() {  // 初始化并查集
  for (int i = 1; i <= n; i++) {
    fa[i] = i, fa[i + n] = i + n;
    siz[i] = 1, siz[i + n] = 0;
  }
}

int main() {
  cin >> n >> m;
  for (int i = 1; i <= m; i++) cin >> edges[i].u >> edges[i].v >> edges[i].w;

  ll ans = 0;
  for (int i = 0; i < 30; i++) {  // 枚举每个数位
    init();

    for (int j = 1; j <= m; j++) {
      int tmpu = find(edges[j].u), tmpv = find(edges[j].v);
      int tmpun = find(edges[j].u + n), tmpvn = find(edges[j].v + n);

      if ((edges[j].w >> i) & 1) {  // 该位为1,说明对应位数码不同
        if (tmpu == tmpv) {  // 矛盾
          cout << -1;
          exit(0);
        }

        if (tmpu == tmpvn) continue;

        // 合并tmpu和tmpvn,tmpv和tmpun
        fa[tmpvn] = tmpu, siz[tmpu] += siz[tmpvn];
        fa[tmpun] = tmpv, siz[tmpv] += siz[tmpun];
      }
      else {  // 该位为0,说明对应位数码相同
        if (tmpu == tmpvn) {  // 矛盾
          cout << -1;
          exit(0);
        }

        if (tmpu == tmpv) continue;

        // 合并tmpu和tmpv,tmpvn和tmpun
```

```
        fa[tmpu] = tmpv, siz[tmpv] += siz[tmpu];
        fa[tmpun] = tmpvn, siz[tmpvn] += siz[tmpun];
      }
    }

    for (int j = 1; j <= n; j++) {  // 统计每个节点对答案该数位的贡献
      ans += (ll)min(siz[find(j)], siz[find(j + n)]) * (1 << i);
      siz[find(j)] = siz[find(j + n)] = 0;
    }
  }
  cout << ans;
}
```

14.1.4 Kanbun

题意

给定一个长度为n $(3 \leq n \leq 1e5)$的只包含'('、'-'、')'且符合括号匹配规则的字符串,下标从1开始.按如下规则阅读该字符串,输出每次阅读的字符的下标:①从第一个字符开始阅读;②对第i个字符,若它是'-'或')',则直接阅读,并跳到第$(i+1)$个字符;③对第i个字符,若它是'(',先找到与其匹配的')',假设其下标为j,先阅读第$(i+1)$到第j个字符,再阅读第i个字符,最后跳到第$(j+1)$个字符;④阅读到第$(n+1)$个字符时终止.

思路I

实现函数dfs(l,r)表示阅读$s[l \cdots r]$.从左往右阅读,遇到'-'或')'时直接阅读;遇到'('时先找到与其匹配的右括号,再递归到它们之间的区间即可.注意边界有$l > r$时的"()"和$l = r$"(-)"两种情况.

最坏的情况是字符串为5e4个左括号和5e4个右括号,若对每个左括号都暴力找到其对应的右括号,时间复杂度$1+2+\cdots+(n-1)=\dfrac{n(n-1)}{2}$,会TLE.考虑优化,注意可在DFS前先预处理出每个左括号匹配的右括号,该过程可通过栈实现.总时间复杂度$O(n)$.

代码I

```
const int MAXN = 1e5 + 5;
int n;  // 长度
char s[MAXN];
int match[MAXN];  // match[l]=r表示下标为l的左括号匹配下标为r的右括号

void dfs(int l, int r) {
  if (l > r) return;  // ()的情况
  if (l == r) {  // (-)的情况
    cout << l << ' ';
    return;
  }

  for (int i = l; i <= r; i++) {
    if (s[i] == '-' || s[i] == ')') cout << i << ' ';
    else {
      // 暴力求每个左括号匹配的右括号,TLE
      //int sum = 1;  // 左括号+1,右括号-1
      //int j = i + 1;
      //while (sum) {
      //  if (s[j] == '(') sum++;
      //  else if (s[j] == ')') {
```

```
      //    sum--;
      //    if (!sum) break;
      //  }
      //  j++;
      //}

      int j = match[i];
      // cout << '(' << i + 1 << ',' << j - 1 << ") ";  // 每次递归到的小区间
      dfs(i + 1, j - 1);
      cout << j << ' ';
      cout << i << ' ';
      i = j;  // 因为还有i++
    }
  }
}

void solve() {
  cin >> n >> s + 1;

  // 预处理每个左括号对应的右括号
  stack<int> stk;
  for (int i = 1; i <= n; i++) {
    if (s[i] == '(') stk.push(i);  // 左括号入栈
    else if (s[i] == ')') {  // 找到一组匹配
      match[stk.top()] = i;
      stk.pop();  // 左括号出栈
    }
  }

  dfs(1, n);  // 从整个串开始搜
}

int main() {
  solve();
}
```

思路II

观察样例知:该阅读顺序等价于遇到'-'直接输出下标;遇到'('时待输出与其匹配的')'的下标后再输出其下标.该过程可用栈实现,时间复杂度$O(n)$.

代码II

```
const int MAXN = 1e5 + 5;
int n;  // 长度
char s[MAXN];

void solve() {
  cin >> n >> s + 1;

  // 预处理每个左括号对应的右括号
  stack<int> stk;
  for (int i = 1; i <= n; i++) {
    if (s[i] == '-') cout << i << ' ';
    else if (s[i] == '(') stk.push(i);  // 左括号入栈
```

```
    else {  // 找到一组匹配
      cout << i << ' ' << stk.top() << ' ';
      stk.pop();  // 左括号出栈
    }
  }
}

int main() {
  solve();
}
```

14.1.5 Make a Square

原题指路: https://codeforces.com/problemset/problem/962/C

题意 (2 s)

给定一个不包含前导零的整数n $(1 \le n \le 2e9)$. 现有操作: 删除n的一个数码. 问至少经过多少次操作可使得n变为一个不包含前导零的完全平方数. 若无法将n变为完全平方数, 输出-1.

思路I

因$n \le 2e9$, 则n至多有10个数码. 预处理2e9内的完全平方数后, 二进制枚举n的数码, 检查保留的数码组成的数字的合法性并更新答案即可.

时间复杂度$O\left(\sqrt{A} + 2^{\lg n} \cdot \lg n \cdot \log \sqrt{A}\right)$, 其中值域$A = 2e9$.

代码I

```
set<int> squares;

void init() {
    for (int i = 1; ; i++) {
        if ((ll)i * i <= 2e9) squares.insert(i * i);
        else return;
    }
}

bool check(string s) {  // 检查s是否包含前导零
    int len = s.length(), i;
    for (i = 0; i < len; i++)
        if (s[i] - '0') break;
    return !i;
}

void solve() {
    string s; cin >> s;

    int ans = INF;
    int len = s.length();
    for (int mask = 1; mask < (1 << len); mask++) {  // mask = 0不符合题意
        string tmp;
        for (int i = 0; i < len; i++)
            if (mask >> i & 1) tmp.push_back(s[i]);

```

```
        if (check(tmp)) {
            int num = stoi(tmp);
            if (squares.count(num)) {
                int res = 0;
                for (int i = 0; i < len; i++)
                    res += !(mask >> i & 1);
                ans = min(ans, res);
            }
        }
    }
    cout << (ans == INF ? -1 : ans) << endl;
}

int main() {
    init();
    solve();
}
```

思路II

因每次操作只能删除数码而不能改变数码, 则有解时, 最优解是$\leq n$的最大的完全平方数. 从大到小枚举$\leq n$的完全平方数, 暴力逐位匹配并更新答案即可.

时间复杂度$O\left(\sqrt{n}\cdot\lg n\right)$.

代码II

```
vector<int> get(int n) {
    vector<int> res;
    for (; n; res.push_back(n % 10), n /= 10);
    return res;
}

void solve() {
    int n; cin >> n;

    auto a = get(n);
    int len1 = a.size();

    for (int i = sqrt(n); i; i--) {
        auto b = get(i * i);
        int len2 = b.size();

        int same = 0;
        for (int j = 0; j < len1; j++) {
            if (a[j] == b[same]) {
                if (++same == len2) {
                    cout << len1 - same << endl;
                    return;
                }
            }
        }
    }
    cout << -1 << endl;
}

```

```
int main() {
    solve();
}
```

思路III

DFS是否保留每个数位, 其中前导零的部分可剪枝.

时间复杂度$O\left(2^{\lg n} \cdot \lg n \cdot t_{\text{sqrt}}\right)$.

代码III

```
const int MAXN = 15;
char s[MAXN], cur[MAXN];  // 输入数、当前DFS到的数
int length;
int ans = INF;

bool check(int len) {
    int res = 0;
    for (int i = 0; i < len; i++)
        res = res * 10 + cur[i] - '0';
    int tmp = sqrt(res + 0.1);
    return res == tmp * tmp;
}

void dfs(int pos, int len) {  // 当前位、当前长度
    if (pos == length) {
        if (len && check(len))
            ans = min(ans, length - len);
        return;
    }

    dfs(pos + 1, len);  // 不保留该位

    if (s[pos] == '0' && !len) return;  // 包含前导零

    // 保留该位
    cur[len++] = s[pos];
    dfs(pos + 1, len);
}

void solve() {
    cin >> s;
    length = strlen(s);

    dfs(0, 0);  // 从第0位开始搜, 当前长度为0
    cout << (ans == INF ? -1 : ans) << endl;
}

int main() {
    solve();
}
```

14.1.6 Mouse Hunt

原题指路: https://codeforces.com/problemset/problem/1027/D

题意 (2 s)

有编号$1 \sim n$的n $(1 \le n \le 2e5)$个房间, 在i $(1 \le i \le n)$号房间设置捕鼠夹的花费为c_i $(1 \le c_i \le 1e4)$. 现有一只老鼠, 初始时老鼠在任一房间中. 每个房间有一个参数, 其中i $(1 \le i \le n)$号房间的参数为a_i $(1 \le a_i \le n)$, 表示若老鼠该时刻在i号房间, 则下一步将移动到a_i号房间. 若老鼠到达设置有捕鼠夹的房间, 则将被抓住. 为保证无论初始时老鼠在哪个房间都能被抓住, 求在房间设置捕鼠夹所需的最小花费.

思路I

若i号房间的参数为a_i, 则节点i与节点a_i间连双向边. 显然这样建立的无向图由若干条(可能为零)链和若干个(可能为零)环构成.

注意到无论初始时老鼠在哪个节点, 充分长的时间后它都会移动到某个环中, 且到某个环中后会遍历该环中的所有节点, 不会再离开该环, 则对每个环, 只需在环中的任一节点处设置捕鼠夹即可. 问题转化为: 求图中的所有环的代价之和, 每个环的代价定义为环上节点中设置捕鼠夹的最小花费, 这可用DFS实现.

DFS求环的方法: 搜索过程中记录路径$path[]$. 用$state[]$记录节点的遍历情况, 其中$state[u] = 0$表示节点u未被遍历, $state[u] = 1$表示u正在被遍历, $state[u] = 2$表示u已被遍历过. 设当前节点的u的下一个节点为a_u. 若$state[a_u] = 1$, 则出现环, 该环包含当前路径中以a_u为起点的后缀.

代码I

```cpp
void solve() {
    int n; cin >> n;
    vector<int> c(n + 1), a(n + 1);  // c[u]表示在节点u设置捕鼠夹的花费, a[u]表示u的下一个节点
    for (int i = 1; i <= n; i++) cin >> c[i];
    for (int i = 1; i <= n; i++) cin >> a[i];

    vector<vector<int>> loops;
    // state[u] = 0, 1, 2分别表示节点u未被遍历过、正在被遍历、已被遍历过
    vector<int> state(n + 1);
    vector<int> path;  // 搜索路径

    function<void(int)> dfs = [&](int u) {
        path.push_back(u);
        state[u] = 1;  // 节点u正在被遍历

        int nxt = a[u];
        if (!state[nxt]) dfs(nxt);
        else if (state[nxt] == 1) {
            vector<int> loop;

            int idx = (int)path.size() - 1;
            while (path[idx] != nxt)
                loop.push_back(path[idx--]);
            loop.push_back(nxt);
            loops.push_back(loop);
        }

        path.pop_back();
        state[u] = 2;
    };

```

```
    for (int u = 1; u <= n; u++)
        if (!state[u]) dfs(u);

    int ans = 0;
    for (auto& loop : loops) {
        int res = INF;
        for (auto u : loop)
            res = min(res, c[u]);
        ans += res;
    }
    cout << ans << endl;
}

int main() {
    solve();
}
```

思路II

若i号房间的参数为a_i, 则节点i向节点a_i连边. 显然这样建立的有向图由若干条(可能为零)链和若干个(可能为零)环构成.

以该有向图的每个入度为0且未被遍历过的节点为起点作拓扑排序, 拓扑排序完成后, 未被遍历到的节点有剩余的入度, 则都在某个环中, 对该环统计答案即可.

代码II

```
void solve() {
    int n; cin >> n;
    vector<int> c(n + 1), a(n + 1);  // c[u]表示在节点u设置捕鼠夹的花费, a[u]表示u的下一个节点
    vector<int> d(n + 1);  // 节点的入度
    for (int i = 1; i <= n; i++) cin >> c[i];
    for (int i = 1; i <= n; i++) {
        cin >> a[i];
        d[a[i]]++;
    }

    vector<bool> vis(n + 1);

    function<void(int)> topo = [&](int u) {  // 以节点u为起点作拓扑排序
        vis[u] = true;

        int nxt = a[u];
        if (!--d[nxt]) topo(nxt);
    };

    // 预处理出环中的节点, 即vis[v] = false的节点v
    for (int u = 1; u <= n; u++)
        if (!d[u] && !vis[u]) topo(u);

    function<int(int)> dfs = [&](int u) {  // 求节点u所在的环的代价
        vis[u] = true;

        int nxt = a[u];
        return vis[nxt] ? c[u] : min(dfs(nxt), c[u]);
    };
```

```
    int ans = 0;
    for (int u = 1; u <= n; u++)
        if (!vis[u]) ans += dfs(u);
    cout << ans << endl;
}

int main() {
    solve();
}
```

思路III

同**思路II**, 建立有向图. 将图按SCC缩点后, 统计每个SCC的出度. 显然只需在出度为0的SCC中设置捕鼠夹. 问题转化为: 求图中的所有出度为0的SCC的代价之和, 每个SCC的代价定义为在其中的节点处设置捕鼠夹的最小花费.

代码III

```
struct TarjanSCC {
    int n;
    vector<vector<int>> edges;
    vector<int> dfn;  // 节点的DFS序
    vector<int> low;  // 节点所能追溯到的最小时间戳
    int tim;  // 时间戳
    vector<int> stk;
    vector<bool> state;  // 记录节点是否在栈中
    vector<int> id;  // 节点所在的SCC的编号
    vector<int> siz;  // 节点所在的SCC的大小
    int sccCnt;  // SCC的个数
    vector<int> out;  // 节点的出度

    TarjanSCC(int _n, const vector<vector<int>>& _edges)
        :n(_n), edges(_edges), tim(0), sccCnt(0) {
        dfn.resize(n + 1), low.resize(n + 1), state.resize(n + 1);
        id.resize(n + 1), siz.resize(n + 1), out.resize(n + 1);

        for (int u = 1; u <= n; u++)
            if (!dfn[u]) tarjan(u);

        for (int u = 1; u <= n; u++) {
            for (auto v : edges[u]) {
                int a = id[u], b = id[v];
                if (a != b) out[a]++;
            }
        }
    }

    void tarjan(int u) {
        dfn[u] = low[u] = ++tim;
        stk.push_back(u);
        state[u] = true;

        for (auto v : edges[u]) {
            if (!dfn[v]) {
                tarjan(v);
                low[u] = min(low[u], low[v]);
```

```
            }
            else if (state[v]) low[u] = min(low[u], dfn[v]);
        }

        if (dfn[u] == low[u]) {  // u是所在的SCC中深度最小的节点
            sccCnt++;
            int cur;
            do {
                cur = stk.back(); stk.pop_back();
                state[cur] = false;
                id[cur] = sccCnt;
                siz[sccCnt]++;
            } while (cur != u);
        }
    }
};

void solve() {
    int n; cin >> n;
    vector<int> c(n + 1);  // c[u]表示在节点u设置捕鼠夹的花费
    for (int i = 1; i <= n; i++) cin >> c[i];
    vector<vector<int>> edges(n + 1);
    for (int u = 1; u <= n; u++) {
        int v; cin >> v;
        edges[u].push_back(v);
    }

    TarjanSCC solver(n, edges);
    vector<int> minCost(solver.sccCnt + 1, INF);  // 每个SCC的代价
    for (int u = 1; u <= n; u++)
        minCost[solver.id[u]] = min(minCost[solver.id[u]], c[u]);

    int ans = 0;
    for (int i = 1; i <= solver.sccCnt; i++)
        if (!solver.out[i]) ans += minCost[i];
    cout << ans << endl;
}

int main() {
    solve();
}
```

14.2 BFS

BFS可在边权相等的图中找到最短路,因为它每次都先遍历完距离相等的点,再依次遍历距离更大的点.

BFS的模板:

```
queue<pii> que;  // 用队列存状态

while(que.size()){  // 队列非空
  auto tmp = que.front(); que.pop();
  // 扩展tmp
}
```

14.2.1 走迷宫

题意

给定一个$n \times m$ $(1 \leq n \leq m \leq 100)$的只包含0和1的二维整数数组表示一个迷宫, 其中0表示可以走的路, 1表示不可穿过的墙壁. 最初某人位于左上角$(1, 1)$处, 他每次可向上、下、左、右任一方向移动一个格子. 输出他从左上角移动到右下角(n, m)处的最短移动距离. 数据保证$(1, 1)$处和(n, m)处的数字为0, 且至少存在一条通路.

思路

开一个$dis[]$数组存每个点到起点的距离, 每行进一步更新距离.

代码

```cpp
typedef pair<int, int> pii;
#define x first
#define y second

const int dx[] = { -1, 0, 1, 0 }, dy[] = { 0, 1, 0, -1 };
int n, m;
vector<vector<int>> a;

int bfs() {
    vector<vector<int>> dis(n + 1, vector<int>(m + 1, INF));
    queue<pii> que;
    que.push({ 1, 1 });
    dis[1][1] = 0;

    while (que.size()) {
        auto [x, y] = que.front(); que.pop();
        for (int i = 0; i < 4; i++) {
            int curx = x + dx[i], cury = y + dy[i];
            if (curx < 1 || curx > n || cury < 1 || cury > m || a[curx][cury]) continue;

            if (dis[curx][cury] > dis[x][y] + 1) {
                dis[curx][cury] = dis[x][y] + 1;
                que.push({ curx, cury });
            }
        }
    }
    return dis[n][m];
}

void solve() {
    cin >> n >> m;
    a = vector<vector<int>>(n + 1, vector<int>(m + 1));
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= m; j++)
            cin >> a[i][j];
    }

    cout << bfs() << endl;
}

int main() {
    solve();
```

```
}
```

14.2.2 八数码

题意

将$1\sim 8$和一个x不重不漏地填在3×3的网格中. 游戏过程中, 可将x与起上下左右方向的数字之一交换. 给定初始的网格, 输出至少经多少次交换可将其变为如下的排列: $\begin{bmatrix}1&2&3\\4&5&6\\7&8&x\end{bmatrix}$, 若不存在可行方案, 输出$-1$.

输入一行, 描述初始的网格. 如网格$\begin{bmatrix}1&2&3\\x&4&6\\7&5&8\end{bmatrix}$描述为$1\ 2\ 3\ x\ 4\ 6\ 7\ 5\ 8$.

思路

将网格的每一个状态作为一个节点, 若可从当前状态经依次交换转移到下一个状态, 则连一条当前节点指向下一个状态对应的节点的边权为1的有向边, 则问题化为求初始状态对应的节点到目标状态对应的节点的最短路.

用一个字符串来表示一个状态, 如目标状态表示为"$12345678x$", 则存状态的队列定义为queue<stirng> que, 再用一个哈希表unordered_map<stirng,int> dis存每个节点与起点的距离.

状态转移时, 先将字符串恢复为网格, 在其中找到x的位置, 后将其分别与上下左右交换得到新的网格, 再将该网格变为字符串存进队列. 若字符串下标从0开始, 且x的下标为k, 则其在网格中的坐标为$(k/3, k\%3)$(其中$/$是整除). 网格中的(x, y)对应字符串中的下标$(3x+y)$.

代码

```cpp
const int dx[] = { -1, 0, 1, 0 }, dy[] = { 0, 1, 0, -1 };
string st, ed = "12345678x";
unordered_map<string, int> dis;

int bfs() {
    queue<string> que;
    que.push(st);
    dis[st] = 0;

    while (que.size()) {
        auto tmp = que.front(); que.pop();
        int distance = dis[tmp];
        if (tmp == ed) return distance;

        int pos = tmp.find('x');  // 'x'在串中的下标
        int x = pos / 3, y = pos % 3;  // 'x'在网格中的坐标

        for (int i = 0; i < 4; i++) {
            int curx = x + dx[i], cury = y + dy[i];
            if (curx < 0 || curx >= 3 || cury < 0 || cury >= 3) continue;

            swap(tmp[curx * 3 + cury], tmp[pos]);  // 交换
            if (!dis.count(tmp)) {
                dis[tmp] = distance + 1;
                que.push(tmp);
            }
```

```
            swap(tmp[curx * 3 + cury], tmp[pos]);  // 恢复
        }
    }
    return -1;
}

void solve() {
    for (int i = 0; i < 9; i++) {
        char ch; cin >> ch;
        st.push_back(ch);
    }

    cout << bfs() << endl;
}

int main() {
    solve();
}
```

14.2.3 Luggage Lock

题意

有一个四位数密码锁,每次操作可将一段连续的数字同时向上或向下转动一格.有T $(1 \leq T \leq 1e5)$组询问,每组询问输入一行,包含两个密码状态$a_0a_1a_2a_3$和$b_0b_1b_2b_3$.求将前者转化为后者的最少操作次数.

思路

显然BFS,从$a_0a_1a_2a_3$开始搜,时间复杂度$O(10^4)$,而T最大1e5,会T.

注意到从一个状态转移到另一个状态只与两状态的数码的差值有关,则可将所有状态化为从0000开始的状态.设$a_0a_1a_2a_3$转化为0000至少需x次操作,$b_0b_1b_2b_3$转化为0000至少需y次操作,则最终答案$ans = |x - y|$.

预处理出从0000开始能转移到的状态的最小操作次数,共$1e4$种状态.每组测试样例$O(1)$查询,总时间复杂度$O(1e4)$.

每个状态用字符串表示.BFS的过程:枚举字符串的后缀的起点,得到这段后缀同时的$+1$、-1的状态,若状态未搜过,则入队并记录步数.

代码

```
map<string, int> state;  // 记录状态及其步数

void bfs() {
  queue<string> que;
  que.push("0000"), state["0000"] = 0;  // 起点
  while (que.size()) {
    auto u = que.front(); que.pop();
    for (int i = 0; i < 4; i++) {  // 枚举字符串后缀的起点
      for (int j = i; j < 4; j++) {
        string up = u, down = u;  // 记录+1、-1的结果
        for (int k = i; k <= j; k++) {
          up[k] = '0' + (u[k] - '0' + 1) % 10;  // +1
          down[k] = '0' + (u[k] - '0' + 9) % 10;  // -1,其中+9即-1+10
        }
```

```
        if (!state.count(up)) {
          que.push(up);
          state[up] = state[u] + 1;  // 更新步数
        }
        if (!state.count(down)) {
          que.push(down);
          state[down] = state[u] + 1;  // 更新步数
        }
      }
    }
  }
}

int main() {
  bfs();

  CaseT{
    string a,b; cin >> a >> b;
    for (int i = 0; i < 4; i++) a[i] = '0' + (a[i] - b[i] + 10) % 10;
    cout << state[a] << endl;
  }
}
```

14.2.4 Matrix Repair

题意 $(0.5\ \mathrm{s})$

传输$n \times n$的$0-1$矩阵A的过程中有些元素可能丢失,丢失的元素用-1表示.原矩阵有一个行校验码、列校验码,分别是行元素的异或值、列元素的异或值,它们在传输过程中不会丢失.给定A和各行的行校验码、各列的列校验码,判断是否能确定原矩阵,若能则输出原矩阵;否则输出-1.

第一行输入一个整数n $(1 \le n \le 1000)$.接下来输入一个$n \times n$的$0-1$矩阵A,元素范围$\{-1, 0, 1\}$.接下来输入n个整数$r_1, \cdots, r_n$,表示各行的行校验码.接下来输入n个整数$c_1, \cdots, c_n$,表示各列的列校验码.数据保证至少存在一种-1的取法使得矩阵满足各行的行校验码和各列的列校验码.

思路

维护每行、列-1的个数$rowcnt[]$和$colcnt[]$.显然能确定填0还是1的只有-1的个数为1的行或列,填入后更新$rowcnt[]$和$colcnt[]$.用BFS完成填数过程,初始时先将只有一个-1的行和列入队.①对-1在行的情况,-1所在的列号pos当且仅当$colcnt[pos]$减到1时入队;②对-1在列的情况,-1所在的行号pos当且仅当$rowcnt[pos]$减到1时入队.

代码

```
const int MAXN = 1005;
int n;
int a[MAXN][MAXN];
int rowcnt[MAXN], colcnt[MAXN];  // 行、列-1的个数
int row[MAXN], col[MAXN];  // 行校验码、列校验码

void bfs() {
  qii que;  // first为-1所在的行或列,second行为1,列为2

  // 只有一个-1的行和列入队
  for (int i = 1; i <= n; i++)
    if (rowcnt[i] == 1) que.push({ i,1 });
```

```
  for (int j = 1; j <= n; j++)
    if (colcnt[j] == 1) que.push({ j,2 });

  while (que.size()) {
    auto& [idx, op] = que.front(); que.pop();

    int cnt = 0;  // 行或列中1的个数
    int pos = -1;  // 行或列中-1的位置
    if (op == 1) {  // 行
      for (int j = 1; j <= n; j++) {
        if (~a[idx][j]) cnt += a[idx][j];
        else pos = j;
      }

      if (!rowcnt[idx] || !colcnt[pos]) continue;  // 已无-1

      a[idx][pos] = row[idx] ^ (cnt & 1);  // 填数
      rowcnt[idx]--, colcnt[pos]--;  // 更新行、列-1的个数
      if (colcnt[pos] == 1) que.push({ pos,2 });  // 列-1的个数减到1时入队
    }
    else {  // 列
      for (int i = 1; i <= n; i++) {
        if (~a[i][idx]) cnt += a[i][idx];
        else pos = i;
      }

      if (!rowcnt[pos] || !colcnt[idx]) continue;  // 已无-1

      a[pos][idx] = col[idx] ^ (cnt & 1);  // 填数
      rowcnt[pos]--, colcnt[idx]--;  // 更新行、列-1的个数
      if (rowcnt[pos] == 1) que.push({ pos,1 });  // 行-1的个数减到1时入队
    }
  }
}

void solve() {
  cin >> n;
  for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= n; j++) {
      cin >> a[i][j];
      if (a[i][j] == -1) rowcnt[i]++, colcnt[j]++;
    }
  }
  for (int i = 1; i <= n; i++) cin >> row[i];
  for (int i = 1; i <= n; i++) cin >> col[i];

  bfs();

  for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= n; j++) {
      if (a[i][j] == -1) {
        cout << -1;
        return;
      }
    }
  }

  for (int i = 1; i <= n; i++)
```

```
    for (int j = 1; j <= n; j++) cout << a[i][j] << " \n"[j == n];
}

int main() {
  solve();
}
```

14.2.5 Nearest Excluded Points

题意 $(4\ \mathrm{s})$

给定平面上n $(1 \le n \le 2\mathrm{e}5)$个相异的整点, 对每个整点$(x_i, y_i)$ $(1 \le x_i, y_i \le 2\mathrm{e}5; 1 \le i \le n)$, 求任一最近的、不在给定的整点中的、与$(x_i, y_i)$的Manhattan距离最小的整点.

思路

称给定的整点为黑点, 未给定的整点为白点. 将黑点分为两类: ①四周(上、下、左、右, 下同)至少存在一个白点; ②四周不存在白点, 即构成一个黑色的十字. 显然对①类黑点, 其四周的白点即为最优解; 对②类黑点$black$, 设其四周的黑点分别为$neighbor_i$ $(i = 1, 2, 3, 4)$, 则与$black$的Manhattan距离最小的白点$white$到$black$的路径可分为两段: 从$black$到某一$neighbor_i$, 再从$neighbor_i$到$white$. 这是一个递归的过程, 递归终止条件为走到某个已确定最优解的①类黑点.

考察黑点构成的各个连通块, 显然①类黑点只能出现在连通块的边界. 显然可用BFS求最短路, 考虑调整搜索的顺序使得上述的递归的过程变为递推的过程, 显然只需保证每个连通块的搜索顺序都是从边界往中心搜. 预处理出①类黑点的最优解, 将其加入队列, 以它们为起点BFS, 递推出②类黑点的最优解即可. 用map<pii, pii>记录每个黑点的最优解, 每个黑点只会记录一次最优解, 总时间复杂度$O(n \log n)$.

代码

```
typedef pair<int, int> pii;
#define x first
#define y second
const int dx[] = { 0, 0, -1, 1 }, dy[4] = { -1, 1, 0, 0 };

void solve() {
  int n; cin >> n;
  vector<pii> points(n);
  set<pii> s;
  for (auto& p : points) {
    cin >> p.x >> p.y;
    s.insert(p);
  }

  map<pii, pii> ans;
  queue<pii> que;
  for (auto [x, y] : points) {
    for (int i = 0; i < 4; i++) {
      int curx = x + dx[i], cury = y + dy[i];
      if (s.count({ curx, cury })) continue;

      ans[{ x, y }] = { curx, cury };
      que.push({ x, y });
      break;
    }
  }

```

```
  while (que.size()) {
    auto [x, y] = que.front(); que.pop();
    for (int i = 0; i < 4; i++) {
      int curx = x + dx[i], cury = y + dy[i];
      if (!s.count({ curx, cury }) || ans.count({ curx, cury })) continue;

      ans[{ curx, cury }] = ans[{ x, y }];
      que.push({ curx, cury });
    }
  }

  for (auto p : points) {
    auto [x, y] = ans[p];
    cout << x << ' ' << y << endl;
  }
}

int main() {
  solve();
}
```

14.3 Flood Fill算法的BFS实现

Flood Fill算法用于在$O(n)$的时间复杂度内找到某个点所在的连通块.

14.3.1 池塘计数

题意

有一片$n \times m$的矩形土地,用字符矩阵表示,有水的单元格用'W'表示,不含水的单元格用'.'表示.每组相连的有水单元格视为一个池塘,每个单元格视为与其周围的8个单元格相连.求池塘个数.

第一行输入整数n, m $(1 \leq n, m \leq 1000)$.第二行起输入一个$n \times m$的字符矩阵,表示土地的积水情况.

思路

一行一行扫描,若发现未被标记的水单元格,则以该单元格为起点做一次Flood Fill标记所有与其相连的单元格,更新答案.

代码

```
const int MAXN = 1005, MAXM = MAXN * MAXN;
int n, m;
char graph[MAXN][MAXN];
bool vis[MAXN][MAXN];

void bfs(int x, int y) {
  qii que;
  que.push({ x,y });
  vis[x][y] = true;

  while (que.size()) {
    pii tmp = que.front(); que.pop();
    for (int i = tmp.first - 1; i <= tmp.first + 1; i++) {
```

```
      for (int j = tmp.second - 1; j <= tmp.second + 1; j++) {
        if (i == tmp.first && j == tmp.second) continue;  // 中间的格子
        if (i < 0 || i >= n || j < 0 || j >= m) continue;
        if (graph[i][j] == '.' || vis[i][j]) continue;
        que.push({ i,j });
        vis[i][j] = true;
      }
    }
  }
}

int main() {
  cin >> n >> m;
  for (int i = 0; i < n; i++) cin >> graph[i];

  int ans = 0;
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < m; j++) {
      if (graph[i][j] == 'W' && !vis[i][j]) {
        bfs(i, j);
        ans++;
      }
    }
  }
  cout << ans;
}
```

14.3.2 城堡问题

题意

```
   1   2   3   4   5   6   7
   #############################
 1 #   |   #   |   #   |   |   #
   #####---#####---#---#####---#
 2 #   #   |   #   #   #   #   #
   #---#####---#####---#####---#
 3 #   |   |   #   #   #   #   #
   #---#########---#####---#---#
 4 #   #   |   |   |   |   #   #
   #############################

   #  = Wall
   |  = No wall
   -  = No wall
   方向:上北下南左西右东
```

上图是一个城堡的地形图,现要计算该城堡的房间数和最大房间的大小.

第一行输入整数m, n $(1 \leq m, n \leq 50)$,分别表示城堡南北、东西方向的长度.接下来m行每行输入n格整数p $(0 \leq p \leq 15)$,描述平面图对应位置的墙的特征,用1、2、4、8分别表示西墙、北墙、东墙、南墙,p为该方块包含的墙的数字之和.城堡的内墙被计算了两次,方块$(1, 1)$的南墙同时也是方块$(2, 1)$的北墙.数据保证城堡至少有两个房间.

输出共两行,第一行输出房间总数,第二行输出最大房间的面积(所含方格数).

思路

一个房间即一个连通块.

西、北、东、南分别视为一个方向0、1、2、3,判断每个方向有无墙只需判断该位置的p值的对应二进制数位是否为1.

代码

```cpp
const int MAXN = 55, MAXM = MAXN * MAXN;
int n, m;
int graph[MAXN][MAXN];
bool vis[MAXN][MAXN];

int bfs(int x, int y) {
  int area = 0;
  qii que;
  que.push({ x,y });
  vis[x][y] = true;

  while (que.size()) {
    pii tmp = que.front(); que.pop();
    area++;
    for (int i = 0; i < 4; i++) {  // 枚举四个方向
      int curx = tmp.first + dx[i], cury = tmp.second + dy[i];
      if (curx < 0 || curx >= n || cury < 0 || cury >= m) continue;
      if (vis[curx][cury]) continue;
      if (graph[tmp.first][tmp.second] >> i & 1) continue;

      que.push({ curx,cury });
      vis[curx][cury] = true;
    }
  }
  return area;
}

int main() {
  cin >> n >> m;
  for (int i = 0; i < n; i++)
    for (int j = 0; j < m; j++) cin >> graph[i][j];

  int cnt = 0, maxarea = 0;
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < m; j++) {
      if (!vis[i][j]) {
        cnt++;
        maxarea = max(maxarea, bfs(i, j));
      }
    }
  }
  cout << cnt << endl << maxarea;
}
```

14.3.3 山峰和山谷

题意

给定一个$n \times n$的矩阵描述一个地图,其中格子(i, j)的高度$h(i, j)$给定.每个单元格视为与其周围的8个单元格相邻.称一个格子的集合S为山峰(山谷),如果:①S的所有格子都有相同的高度;②S的所有格子都连通;③对$s \in S$,与$\forall s$相邻的$s' \notin S$,有$h_s > h_{s'}$(山峰)或$h_s < h_{s'}$(山谷);④若周围不存在相邻区域,则将其同时视为山峰和山谷.现要对给定的地图,求山峰和山谷的数量.若所有格子高度相同,则整个地图既是山峰也是山谷.

第一行输入整数n $(1 \le n \le 1000)$,表示地图大小.第二行起输入一个$n \times n$的矩阵,表示每个格子的高度h $(0 \le h \le 1e9)$.

输出山峰和山谷的数量.

思路

BFS时记录周围有无比当前格子高、低的格子,进而判断该连通块的类型.

代码

```cpp
const int MAXN = 1005, MAXM = MAXN * MAXN;
int n, m;
int h[MAXN][MAXN];
bool vis[MAXN][MAXN];

void bfs(int x, int y, bool& has_higher, bool& has_lower) {
  qii que;
  que.push({ x,y });
  vis[x][y] = true;

  while (que.size()) {
    pii tmp = que.front(); que.pop();
    for (int i = tmp.first - 1; i <= tmp.first + 1; i++) {
      for (int j = tmp.second - 1; j <= tmp.second + 1; j++) {
        if (i == tmp.first && j == tmp.second) continue;
        if (i < 0 || i >= n || j < 0 || j >= n) continue;

        if (h[i][j] != h[tmp.first][tmp.second]) {  // 山脉的边界
          if (h[i][j] > h[tmp.first][tmp.second]) has_higher = true;
          else has_lower = true;
        }
        else if (!vis[i][j]) {
          que.push({ i,j });
          vis[i][j] = true;
        }
      }
    }

  }
}

int main() {
  cin >> n;
  for (int i = 0; i < n; i++)
    for (int j = 0; j < n; j++) cin >> h[i][j];

  int peak = 0, valley = 0;
  for (int i = 0; i < n; i++) {
```

```
    for (int j = 0; j < n; j++) {
      if (!vis[i][j]) {
        bool has_higher = false, has_lower = false;
        bfs(i, j, has_higher, has_lower);
        if (!has_higher) peak++;
        if (!has_lower) valley++;  // 注意不是else
      }
    }
  }
  cout << peak << ' ' << valley;
}
```

14.4 BFS最短路模型

14.4.1 迷宫

题意

给定一个$n \times n$ $(0 \leq n \leq 1000)$的整型矩阵表示一个迷宫(下标从0开始),其中1表示墙壁,0表示可走的路,玩家只能横竖走,不能斜着走.求从左上角到右下角的最短路径,数据保证有解.若有多组解,任输出一组解.

思路

开一个pii类型的$pre[][]$数组记录每个格子的前驱.可从终点开始搜,这样从起点往回找的路径即正向路径.

代码

```
const int MAXN = 1005, MAXM = MAXN * MAXN;
int n, m;
int graph[MAXN][MAXN];
pii pre[MAXN][MAXN];

void bfs(int x, int y) {
  memset(pre, -1, so(pre));
  pre[x][y] = { 0,0 };

  qii que;
  que.push({ x,y });

  while (que.size()) {
    pii tmp = que.front(); que.pop();
    for (int i = 0; i < 4; i++) {
      int curx = tmp.first + dx[i], cury = tmp.second + dy[i];
      if (curx < 0 || curx >= n || cury < 0 || cury >= n) continue;
      if (graph[curx][cury]) continue;
      if (~pre[curx][cury].first) continue;

      que.push({ curx,cury });
      pre[curx][cury] = tmp;
    }
  }
}

int main() {
  cin >> n;
```

```
  for (int i = 0; i < n; i++)
    for (int j = 0; j < n; j++) cin >> graph[i][j];

  bfs(n - 1, n - 1);  // 从终点开始搜

  pii cur = { 0,0 };
  while (true) {
    cout << cur.first << ' ' << cur.second << endl;
    if (cur.first == n - 1 && cur.second == n - 1) break;
    cur = pre[cur.first][cur.second];
  }
}
```

14.4.2 武士风度的牛

题意

用一个$n \times m$ $(1 \leq n,m \leq 150)$的字符矩阵描述一个牧场,障碍用'*'表示,草用'H'表示,其余位置用'.'表示.有一头牛,用字符'K'表示,可在牧场上跳"日"字,但它不能跳到树上或石头上.问该牛向吃到草至少要跳多少次.数据保证有解.

代码

```
const int MAXN = 155, MAXM = MAXN * MAXN;
int n, m;
char graph[MAXN][MAXN];
int dis[MAXN][MAXN];

int bfs() {
  const int dx[8] = { -2,-1,1,2,2,1,-1,-2 }, dy[8] = { 1,2,2,1,-1,-2,-2,-1 };

  pii cow;
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < m; j++) {
      if (graph[i][j] == 'K') {
        cow = { i,j };
        break;
      }
    }
  }
  memset(dis, -1, so(dis));
  dis[cow.first][cow.second] = 0;
  qii que;
  que.push(cow);

  while (que.size()) {
    pii tmp = que.front(); que.pop();
    for (int i = 0; i < 8; i++) {
      int curx = tmp.first + dx[i], cury = tmp.second + dy[i];
      if (curx < 0 || curx >= n || cury < 0 || cury >= m) continue;
      if (graph[curx][cury] == '*') continue;
      if (~dis[curx][cury]) continue;
      if (graph[curx][cury] == 'H') return dis[tmp.first][tmp.second] + 1;

      que.push({ curx,cury });
      dis[curx][cury] = dis[tmp.first][tmp.second] + 1;
```

```
    }
  }
}

int main() {
  cin >> m >> n;
  for (int i = 0; i < n; i++) cin >> graph[i];

  cout << bfs();
}
```

14.4.3 抓住那头牛

题意

农夫和牛分别在数轴上的点n、k $(0 \le n, k \le 1e5)$处,农夫知道牛的位置,他想抓牛,假设牛不动.农夫每一步有两种移动方式:①从x移动到$x-1$或$x+1$;②从x移动到$2x$.问农夫至少需要多少步才能抓到牛.

思路

考虑最优解中至多用到数轴上坐标多少的点.显然最优解中不会用到负数坐标的点,因为走到负数坐标后还需走回正数.若$k < n$,则不会使用$x \to x+1$或$x \to 2x$的移动方式.直观上坐标范围为$[0, 2e5]$,但事实上$[0, 1e5]$足够.

代码

```
const int MAXN = 1e5 + 5;
int n, k;
int dis[MAXN];

int bfs() {
  memset(dis, -1, so(dis));
  dis[n] = 0;

  qi que;
  que.push(n);
  while (que.size()) {
    int tmp = que.front(); que.pop();
    if (tmp == k) return dis[k];

    if (tmp + 1 < MAXN && dis[tmp + 1] == -1) {  // x走到x+1
      dis[tmp + 1] = dis[tmp] + 1;
      que.push(tmp + 1);
    }
    if (tmp - 1 >= 0 && dis[tmp - 1] == -1) {  // x走到x-1
      dis[tmp - 1] = dis[tmp] + 1;
      que.push(tmp - 1);
    }
    if (tmp * 2 < MAXN && dis[tmp * 2] == -1) {  // x走到2x
      dis[tmp * 2] = dis[tmp] + 1;
      que.push(tmp * 2);
    }
  }
}

int main() {
```

```
  cin >> n >> k;

  cout << bfs();
}
```

14.4.4 Fight Against Traffic

原题指路: https://codeforces.com/problemset/problem/954/D

题意

给定一张包含编号$1 \sim n$的n $(2 \leq n \leq 1000)$个节点和m $(1 \leq m \leq 1000)$条边的无向无权图. 给定起点s $(1 \leq s \leq n)$和终点t $(1 \leq t \leq n)$. 现需将一对未连边的节点相连, 使得s到t的最短路不减. 求方案数.

思路

分别求节点s、t到其他节点的最短路$dis1[], dis2[]$. 枚举未连边的节点u和v, 则连边后s到t的最短路为$\min\{dis1[u] + 1 + dis2[v], dis1[v] + 1 + dis2[u]\}$.

代码

```
void solve() {
    int n, m, s, t; cin >> n >> m >> s >> t;
    vector<set<int>> edges(n + 1);
    while (m--) {
        int u, v; cin >> u >> v;
        edges[u].insert(v), edges[v].insert(u);
    }

    auto bfs = [&](int s) {
        vector<int> dis(n + 1, INF);
        queue<int> que;
        que.push(s);
        dis[s] = 0;

        while (que.size()) {
            auto u = que.front(); que.pop();
            for (auto v : edges[u]) {
                if (dis[v] > dis[u] + 1) {
                    dis[v] = dis[u] + 1;
                    que.push(v);
                }
            }
        }
        return dis;
    };

    int ans = 0;
    auto dis1 = bfs(s), dis2 = bfs(t);
    for (int u = 1; u <= n; u++) {
        for (int v = u + 1; v <= n; v++) {
            if (!edges[u].count(v)) {
                int d = min(dis1[u] + 1 + dis2[v], dis1[v] + 1 + dis2[u]);
                ans += d >= dis1[t];
            }
```

```
        }
    }
    cout << ans << endl;
}

int main() {
    solve();
}
```

14.5 多源BFS

14.5.1 矩阵距离

题意

给定一个$n \times m$ $(1 \leq n, m \leq 1000)$的01矩阵A,定义$A[i][j]$与$A[k][l]$间的Manhattan距离$dis(A[i][j], A[k][l]) = |i - k| + |j - l|$.输出一个$n \times m$的整数矩阵$B$,其中$B[i][j] = \min\limits_{1 \leq x \leq n, 1 \leq y \leq m, A[x][y]=1} dis(A[i][j], A[x][y])$.

思路

即求每个元素到最近的1的距离.

类比图论中有多个起点,求点到其最近的起点的最短路时,可建立与多个起点相连的虚拟源点,再求目标点到虚拟源点的最短路.在本题中,只需先将B中对应的A中所有1的位置初始化为0并入队即可.

代码

```
const int MAXN = 1005, MAXM = MAXN * MAXN;
int n, m;
char graph[MAXN][MAXN];
int dis[MAXN][MAXN];

void bfs() {
  memset(dis, -1, so(dis));

  qii que;
  for (int i = 1; i <= n; i++) {  // 将A中所有1的位置入队
    for (int j = 1; j <= m; j++) {
      if (graph[i][j] == '1') {
        dis[i][j] = 0;
        que.push({ i,j });
      }
    }
  }

  while (que.size()) {
    pii tmp = que.front(); que.pop();
    for (int i = 0; i < 4; i++) {
      int curx = tmp.first + dx[i], cury = tmp.second + dy[i];
      if (curx < 1 || curx > n || cury < 1 || cury > m) continue;
      if (~dis[curx][cury]) continue;

```

```
      dis[curx][cury] = dis[tmp.first][tmp.second] + 1;
      que.push({ curx,cury });
    }

  }
}

int main() {
  cin >> n >> m;
  for (int i = 1; i <= n; i++) cin >> graph[i] + 1;

  bfs();

  for (int i = 1; i <= n; i++)
    for (int j = 1; j <= m; j++) cout << dis[i][j] << " \n"[j == m];
}
```

14.6 BFS的最小步数模型

14.6 魔板

题意

```
1 2 3 4
8 7 6 5
```

上图是一张有8个大小相同的格子的魔板,魔板的每个格子有一种颜色,这8种颜色用$1 \sim 8$的整数表示.用颜色序列表示魔板的一种状态,规定从魔板左上角开始,沿顺时针方向取出整数,构成一个颜色序列,如上图的颜色序列为$\{1,2,3,4,5,6,7,8\}$,这是基本状态.

现有三种操作,可通过这些操作改变魔板状态:

A:交换上下两行.如初始时的魔板经一次A操作变为:

```
8 7 6 5
1 2 3 4
```

B:将最右边的一列插入到最左边.

```
4 1 2 3
5 8 7 6
```

C:对中间四个数顺时针旋转.

```
1 7 2 4
8 6 3 5
```

现给定魔板的特殊状态,求一个操作序列将其转化为基本状态,数据保证有解.

第一行输入8个整数(数的范围$[1,8]$),表示目标状态.

第一行输出一个整数,表示最短操作序列的长度len.若$len > 0$,第二行输入字典序最小的操作序列.

思路

可用哈希表存已经搜到过的状态.

BFS入队时,按做操作A、B、C依次入队即可保证答案字典序最小.

代码

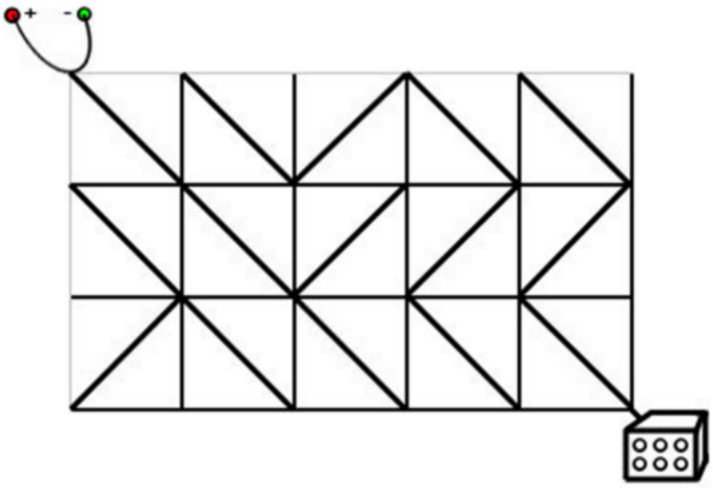
```
char board[2][4];  // 魔板
umap<string, pair<char, string>> pre;  // 记录每个状态的前驱和经过的操作
umap<string, int> dis;

void set_board(string state) {  // 将魔板更新为state对应的状态
  for (int i = 0; i < 4; i++) board[0][i] = state[i];
  for (int i = 7, j = 0; j < 4; i--, j++) board[1][j] = state[i];
}

string get_board() {
  string res;
  for (int i = 0; i < 4; i++) res += board[0][i];
  for (int i = 3; i >= 0; i--) res += board[1][i];
  return res;
}

string opA(string state) {
  set_board(state);
  for (int i = 0; i < 4; i++) swap(board[0][i], board[1][i]);
  return get_board();
}

string opB(string state) {
  set_board(state);
  int tmp0 = board[0][3], tmp1 = board[1][3];
  for (int i = 3; i >= 0; i--) board[0][i] = board[0][i - 1], board[1][i] = board[1][i -
1];
  board[0][0] = tmp0, board[1][0] = tmp1;
  return get_board();
}

string opC(string state) {
  set_board(state);
  int tmp = board[0][1];
  board[0][1] = board[1][1], board[1][1] = board[1][2], board[1][2] = board[0][2],
board[0][2] = tmp;
  return get_board();
}

int bfs(string start, string end) {
  if (start == end) return 0;

  queue<string> que;
  que.push(start);
  dis[start] = 0;

```

```
  while (que.size()) {
    auto tmp = que.front(); que.pop();
    string cur[3] = { opA(tmp),opB(tmp),opC(tmp) };
    for (int i = 0; i < 3; i++) {
      if (!dis.count(cur[i])) {
        dis[cur[i]] = dis[tmp] + 1;
        pre[cur[i]] = { 'A' + i,tmp };
        if (cur[i] == end) return dis[end];
        que.push(cur[i]);
      }
    }
  }
}

int main() {
  string start = "12345678";
  string end;
  for (int i = 0; i < 8; i++) {
    int x; cin >> x;
    end.push_back(x + '0');
  }

  int step = bfs(start, end);
  cout << step << endl;

  if (step) {
    string ans;
    while (start != end) {
      ans += pre[end].first;
      end = pre[end].second;
    }
    reverse(all(ans));
    cout << ans;
  }
}
```

14.7 双端队列BFS

14.7.1 电路维修

题意



如上图,电路板的整体结构是一个$r \times c$的网格,每个格点都是电线的接点,每个格子包含一个电子元件.电子元件的主要部分是一个可旋转的、连接一条对角线上的两端点的短电缆,旋转后它可连接另一对角线的两接点.电路板左上角的接点接入直流电源,右下角的接点接入飞行车的发动装置.初始时,电路可能处于断路状态.现需旋转最少的元件使得电源与发动装置接通.注意电流只能走斜向的线段.

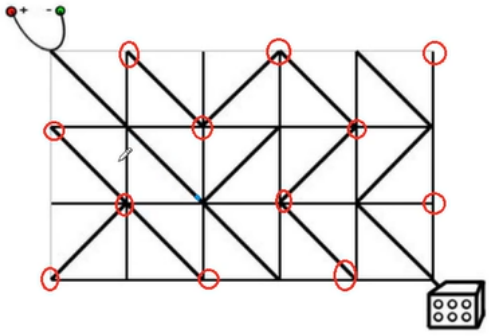
有$t\ (1 \leq t \leq 5)$组测试数据.每组测试数据第一行输入整数$r, c\ (1 \leq r, c \leq 500)$,表示电路板的行数和列数.之后输入一个$r \times c$的字符为'/'或'\'的字符矩阵,表示初始时元件的方向.

对每组测试数据,若能通过旋转使得电路接通,则输出最小的旋转次数;否则输出"NO SOLUTION".

思路

将接点视为图中的节点,无需旋转元件即可相连的节点间边权为0,否则边权为1.可建图后用Dijkstra跑最短路.

因只能沿斜线走,则每走一步横纵坐标同时变化1,则起点的横纵坐标之和与终点的横纵坐标之和的奇偶性不同时无解.



注意到上图中红色圈出的节点不可到达,每个格子内只有对角线上的点能到达,同一条边上的点不能到达(因方案中每个元件的角度固定,不能同时用到两组对角线上的点).

对边权为0或1的图,可用双端队列BFS,即考察当前节点扩展出来的边的边权,若边权为0,则插入队首;否则插入队尾.注意每个点可能不止入队一次,则每个点出队时才能确定最小值.

代码

```
const int MAXN = 505, MAXM = MAXN * MAXN;
int n, m;
char graph[MAXN][MAXN];
int dis[MAXN][MAXN];
bool vis[MAXN][MAXN];

```

```cpp
int bfs() {
  memset(vis, 0, so(vis));
  memset(dis, INF, so(dis));

  dis[0][0] = 0;
  deque<pii> que;
  que.push_back({ 0,0 });

  const char str[] = "\\/\\/";  // 注意\是转义字符,故要2个
  const int dx[4] = { -1,-1,1,1 }, dy[4] = { -1,1,1,-1 };  // 左上、左下、右下、右上的偏移量
  const int ix[4] = { -1,-1,0,0 }, iy[4] = { -1,0,0,-1 };  // 格点到graph[][]的偏移量

  while (que.size()) {
    auto tmp = que.front(); que.pop_front();
    int x = tmp.first, y = tmp.second;
    if (x == n && y == m) return dis[x][y];  // 每个点出队时才能确定最小值

    if (vis[x][y]) continue;
    vis[x][y] = true;

    for (int i = 0; i < 4; i++) {
      int curx = x + dx[i], cury = y + dy[i];
      if (curx < 0 || curx > n || cury < 0 || cury > m) continue;

      int gx = x + ix[i], gy = y + iy[i];  // 对应到graph[][]中的坐标
      int w = graph[gx][gy] != str[i];  // 无需旋转边权为0,否则边权为1
      int d = dis[x][y] + w;
      if (d <= dis[curx][cury]) {
        dis[curx][cury] = d;
        if (!w) que.push_front({ curx,cury });
        else que.push_back({ curx,cury });
      }
    }
  }
}

int main() {
  CaseT{
    cin >> n >> m;
    for (int i = 0; i < n; i++) cin >> graph[i];

    if (n + m & 1) cout << "NO SOLUTION" << endl;
    else cout << bfs() << endl;
  }
}
```

14.7.2 Journey

原题指路:https://ac.nowcoder.com/acm/contest/33188/J

题意

某城市有n个十字路口.某人每次有两种行进方式:①直走、左转或原地转身:要等一个红灯;②右转:无需等红灯.求他从起始的马路到达他想到的马路最少需等多少红灯.

第一行输入整数n $(2 \leq n \leq 5e5)$,表示该城市的十字路口数.接下来n行每行输入四个相异的整数$c_{i,1}, c_{i,2}, c_{i,3}, c_{i,4}$ $(0 \leq c_{i,j} \leq n)$,分别表示第i个十字路口的四条马路的起点,其中$c_{i,j} = 0$的马路某人不会经过.马路以逆时针排列,换句话说,设某人当前在马路$< c_{i,j}, i >$,若他想前往马路$< i, c_{i,j\%4+1} >$,因为这是右转,所以他无需等红灯;否则,若他想前往其他马路,则他需等红灯.数据保证地图中不存在重边和自环,且每条马路都是双向边.最后一行输入四个整数s_1, s_2, t_1, t_2 $(1 \leq s_1, s_2, t_1, t_2 \leq n)$,表示初始时某人在马路$< s_1, s_2 >$,他想到达马路$< t_1, t_2 >$,数据保证起点和终点的马路在上述地图中.

输出一个整数,表示他从起点到终点最少需等多少红灯.

Input

```
4
3 4 0 0
0 0 4 3
2 1 0 0
2 0 0 1
4 2 4 1
```

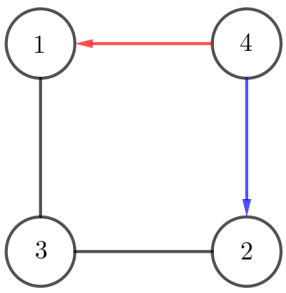
Output

```
1
```

样例解释

题面表述不清,大意为某人有四个朝向,在他所在的十字路口,右转则无需等红灯,其余情况都需等一个红灯.输入的最后一行描述他的起点马路和朝向、终点马路和朝向.

对输入描述的地图的解释:



第一行4,表示有4个十字路口,可理解为有4个城市.

第二行3 4 0 0,表示城市1与城市3, 4间有马路相连,且城市1, 3, 4逆时针排列.

第三行0 0 4 3,表示城市2与城市4, 3间有马路相连,且城市2, 4, 3逆时针排列.

第四行2 1 0 0,表示城市3与城市2, 1间有马路相连,且城市3, 2, 1逆时针排列.

第五行2 0 0 1,表示城市4与城市2, 1间有马路相连,且城市4, 2, 1逆时针排列.

对朝向的解释:

某人初始时在马路$< 4, 2 >$,表示他在城市4与2间的边上,且朝向城市2,如上图蓝色箭头所示.

某人想到马路$< 4, 1 >$,表示他想到城市4与1间的边上,且朝向城市1,如上图红色箭头所示.

对答案的解释:

某人当前在马路$< 4, 2 >$,他直走到城市2,因为马路不是十字路口所以无需等红灯.

现他在城市2处,朝向依旧是$4 \to 2$.而城市2的输入行0 0 4 3中城市4在下标为3的位置,则朝向$4 \to 2$即$pos = 3$.

因$pos\%4 + 1 = 4$,则城市2的输入行中下标为4的城市3即在城市2处右转能到达的城市,无需等红灯;

到达其余城市(本行中无)需等红灯.

现他在城市2的十字路口处右转,直走到达城市3,过程中无需等红灯.

现他在城市3处,朝向$2 \to 3$.而城市3的输入行2 1 0 0中城市2在下标为1的位置,则朝向$2 \to 3$即$pos = 1$.

因$pos\% + 1 = 2$,则城市3的输入行中下标为2的城市1即在城市3处右转能到达的城市,无需等红灯;

到达其余城市(本行中无)需等红灯.

现他在城市3的十字路口处右转,直走到达城市1,过程中无需等红灯.

现他在城市1处,朝向$3 \to 1$,而城市1的输入行3 4 0 0中城市3在下标1的位置,则朝向$3 \to 1$即$pos = 1$.

因$pos\% + 1 = 2$,则城市1的输入行中下标为2的城市4即在城市1处右转能到达的城市,无需等红灯;

到达其余城市(本行中无)需等红灯.

现他在城市1的十字路口处右转,直走到达城市4,过程中无需等红灯.

现他在马路$< 1, 4 >$上,但朝向为$1 \to 4$,而他的目标朝向是$4 \to 1$,故他需等一次红灯完成转向.

可以证明,等一次红灯是最优解.

思路

将城市视为点,城市间的马路视为边,则能通过直走、左转、右转到达的城市间有边相连.右转无需等红灯,则当前城市到右转到达的城市间有边权为0的边;其他行进方式需等一个红灯,则当前城市到其他行进方式到达的城市间有边权为1的边.边权只有0和1,考虑$0/1$BFS.注意这样转化后无需将整个图建出来,只需根据输入的矩阵$c[][]$即可得到每个点能以什么行进方式到达哪些点.总体思路:通过边权为0的边能到达的点(右转能到达的城市)插入到双端队列的队首,花费不变;其他城市插入到队尾,花费$+1$.BFS扩展时优先扩展通过边权为0的边能到达的点,显然这是最优的.

用一个三元组$(cur, pre, cost)$表示从城市pre到城市cur的花费为$cost$,用一个全局的ans记录等红灯数的最小值(后面会看见它可能会被多次更新),ans初始化为llf的最大值$INFF$.注意每取出一个新的队首都需更新当前的朝向pos,即找到一个 $s.t.\ c[cur][pos] = pre$的pos.

①每次取出队首元素tmp,若tmp的$cost \geq ans$,因策略是优先扩展通过边权为0的边能到达的点,则出现了第一个不小于之前更新过的ans的$cost$时,后面的$cost$只能不变或增加,显然ans是等红灯数的最小值,直接返回.

②若$(cur, pre) = (t_2, t_1)$,即走到了目标马路且朝向正确,因策略是优先扩展通过边权为0的边能到达的点,则第一次到达时一定是最优解,直接返回当前的$cost$.

③若$(cur, pre) = (t_1, t_2)$,即走到了目标马路但朝向相反,注意此时不能确定$(cost + 1)$是否是最优解,因为可能存在另一条一直右转的路径使得最后到达了目标马路且朝向正确,故先用ans记录当前的$(cost + 1)$,并继续扩展下一个队首元素.

④判重,注意不能只开一个一维的$vis[]$表示每个点是否被遍历过,因为从不同的方向到达相同的点花费可能不同,故应开一个二维的$vis[][]$,其中$vis[u][pos]$表示表示从pos方向到点u是否被遍历过.

⑤若队列空后都未返回,则检查ans是否是$INFF$,若是则无解,返回-1;否则返回ans.

代码

```cpp
typedef tuple<int, int, ll> tiil;
const int MAXN = 5e5 + 5;
int n;
int c[MAXN][5];
int s1, s2, t1, t2;
bool vis[MAXN][5];  // vis[u][pos]表示从pos方向到点u是否被遍历过

ll bfs(int s1, int s2) {
  ll ans = INFF;
  deque<tiil> que;
  que.push_back({ s2, s1, 0 });  /// cur,pre,cost

  while (que.size()) {
    auto tmp = que.front(); que.pop_front();
    int cur, pre;
    ll cost;
    tie(cur, pre, cost) = tmp;
    if (!cur) continue;  // 走到了不存在的节点
    if (cost >= ans) return ans;  // 当前的花费不是最优解,返回之前找到的最优解
    if (cur == t2 && pre == t1) return cost;  // 能直接走到的一定是最优解
    if (cur == t1 && pre == t2) {  // 走到边上但还要转向的未必是最优解,注意不能直接return cost + 1;
      ans = cost + 1;
      continue;
    }

    int pos;  // 当前朝向
    for (int i = 1; i <= 4; i++) {
      if (c[cur][i] == pre) {
        pos = i;
        break;
      }
    }

    if (vis[cur][pos]) continue;  // 走过该点的该方向
    vis[cur][pos] = true;

    ll res = cost;
    que.push_front({ c[cur][pos % 4 + 1], cur, res });  // 右转无需花费,插到队首
    for (int i = 1; i <= 4; i++)  // 其他方向需要花费,插到队尾
      if (i != pos) que.push_back({ c[cur][i % 4 + 1], cur, res + 1 });
  }

  return ans == INFF ? -1 : ans;
}

int main() {
  cin >> n;
  for (int i = 1; i <= n; i++) cin >> c[i][1] >> c[i][2] >> c[i][3] >> c[i][4];
  cin >> s1 >> s2 >> t1 >> t2;

  cout << bfs(s1, s2);
}
```

14.8 双向BFS

双向BFS常用于优化BFS的最小步数模型,一般不用于BFS的最短路模型.

14.8.1 字串变换

题意

已知有两字串A、B和一组(至多6个)字串变换规则:$A_1 \to B_1, A_2 \to B_2, \cdots$,表示$A$中子串$A_1$可变换为$B_1$,子串$A_2$可变换为$B_2, \cdots$.

第一行输入两给定字符串A和B.接下来若干行每行输入两个字符串A_i和B_i描述一组字串变换规则.数据保证所有字符串长度不超过20.

若在10步内(含10步)能将A变换为B,则删除最小变换步数;否则输出"NO ANSWER!".

思路

最坏情况下每个字符串长度都为20且都能应用6个规则,每个规则都是单个字符替换,则每一步有$20 \times 6 = 120$种情况,10步以内共120^{10}种情况.故朴素BFS从起点搜到终点可能会TLE或MLE.

双向BFS同时从起点和终点往中间搜,若在中间相遇,则有解;否则无解.这样从起点和终点各搜5步,共2×6^5种情况.

进一步优化,每次选择元素较少的队列的方向扩展.

代码

```cpp
const int MAXN = 6;
int n;  // 规则数
string A, B;
string str1[MAXN], str2[MAXN];  // 变换规则

// 队列、到起点的距离、到终点的距离、变换规则a→b
int extend(queue<string>& que, umap<string, int>& disA, umap<string, int>& disB, string a[MAXN], string b[MAXN]) {
  int d = disA[que.front()];
  while (que.size() && disA[que.front()] == d) {
    auto tmp = que.front(); que.pop();
    for (int i = 0; i < n; i++) {  // 枚举变换规则
      for (int j = 0; j < tmp.size(); j++) {  // 枚举应用变化规则的起点
        if (tmp.substr(j, a[i].size()) == a[i]) {
          string state = tmp.substr(0, j) + b[i] + tmp.substr(j + a[i].size());
          if (disB.count(state)) return disA[tmp] + disB[state] + 1;  // 相遇
          if (disA.count(state)) continue;

          disA[state] = disA[tmp] + 1;
          que.push(state);
        }
      }
    }
  }
  return 11;  // 返回一个比10大的数
}

int bfs() {
  if (A == B) return 0;
```

```
29
30    queue<string> queA, queB;  // 从起点、终点开始搜索的队列
31    umap<string, int> disA, disB;  // 与起点、终点的距离
32    queA.push(A), disA[A] = 0, queB.push(B), disB[B] = 0;
33
34    while (queA.size() && queB.size()) {
35      int step = queA.size() <= queB.size() ?  // 选择元素较少的队列的方向扩展
36        extend(queA, disA, disB, str1, str2) : extend(queB, disB, disA, str2, str1);  // 注意
  后者规则反着用
37      if (step <= 10) return step;
38      if (++step == 10) return 11;  // 返回一个比10大的数
39    }
40    return 11;  // 返回一个比10大的数
41  }
42
43  int main() {
44    cin >> A >> B;
45    while (cin >> str1[n] >> str2[n])n++;
46
47    int ans = bfs();
48    cout << (ans <= 10 ? to_string(ans) : "NO ANSWER!");
49  }
```

14.9 A*

A*算法类似于Dijkstra算法,是对BFS的优化.朴素BFS中直接从起点搜到终点可能会经过很多状态,而A*算法中加入启发函数(估价函数),使得只需搜较少的状态即可找到从起点到终点的一条最短路.A*算法只在搜索空间很大时才有明显的优化效果.A*算法和Dijkstra算法都能解决边权非负的图中的最短路.Dijkstra算法可看作从每个点到终点的估计距离都是0的最短路.

A*算法将BFS中的队列换为小根堆,队列中不仅存起点到当前点的真实距离,还存该点到终点的估计距离.每次选择与终点的估计距离最小的点扩展.

A*算法中每个点可能会被扩展多少次.

①BFS入队时判重;②Dijkstra算法出队时判重;③A*算法不判重.

A*算法的成立条件:估计距离$\leq$真实距离.

A*算法的应用场景:确定有解.若无解时,A*算法会将整个搜索空间都搜索一遍,效率低于朴素BFS.

但实际应用中未必知道是否有解,也可用A*算法,大部分时候比朴素BFS快.

当终点第一次出队时已确定最短距离.

[**证**] 设终点u当前出队,此时u与起点的距离为$d(u)$.

设起点到u的真实距离为$dis(u)$,u到终点的估计距离和真实距离分别为$f(u)$和$g(u)$.

设真实的最短路径为d,则$d = dis(u) + g(u) \geq dis(u) + f(u)$.

若u出队时不是最短距离,则$d(u) > d \geq dis(u) + f(u)$,且是最短距离的节点在队列中.

这表明:当前出队的不是队列中的最小值,矛盾.

[**注**] A*算法只能保证终点出队时是最短距离,不能保证其他节点.

14.9.1 八数码

题意

在3×3的网格中不重不漏地填入$1 \sim 8$这8个数字,空位用'X'表示.游戏过程中,可将X与其上、下、左、右四个方向之一的数字交换(若存在),目标是通过交换将初始网格变为如下形式(称为正确排列):

```
1 2 3
4 5 6
7 8 x
```

将X与其上、下、左、右四个方向的数字交换分别记作u、d、l、r.现给定一个初始网格,求通过最少的移动次数将其变为正确排列的交换序列,若有多种方案,输出任一种;若无方案,输出"unsolvable".

用一个字符串描述3×3的初始网格.若初始网格为:

```
1 2 3
x 4 6
7 5 8
```

则输入$1\ 2\ 3\ x\ 4\ 6\ 7\ 5\ 8$.

思路

八数码问题有解的充要条件:初始序列的逆序对数量是偶数.

[**证**] (充) 较难,略.

(必) 显然左右交换不改变逆序对数量,上下交换只会改变两对逆序对,故初始序列的逆序对数与正确排列的逆序对数同奇偶.

注意到最优解中每次交换可使得交换的数与其目标位置的Manhattan距离减1,估价函数取每个数码与其目标位置的Manhattan距离之和.

代码

```cpp
int f(string state) {  // 估价函数
  int res = 0;
  for (int i = 0; i < state.size(); i++) {
    if (state[i] != 'x') {
      int tmp = state[i] - '1';
      res += abs(i / 3 - tmp / 3) + abs(i % 3 - tmp % 3);
    }
  }
  return res;
}

string bfs(string start) {
```

```
  const int dx[4] = { -1,0,1,0 }, dy[4] = { 0,1,0,-1 };
  const char op[] = "urdl";  // 与上面的偏移量对应
  string end = "12345678x";
  umap<string, int> dis;
  umap<string, pair<string, char>> pre;  // 记录前驱和操作
  priority_queue<pair<int, string>, vector<pair<int, string>>, greater<pair<int, string>>>
heap;
  heap.push({ f(start),start });
  dis[start] = 0;

  while (heap.size()) {
    auto tmp = heap.top(); heap.pop();
    string state = tmp.second;
    if (state == end) break;

    int x, y;  // x的位置
    for (int i = 0; i < state.size(); i++) {
      if (state[i] == 'x') {
        x = i / 3, y = i % 3;
        break;
      }
    }

    int step = dis[state];
    string backup = state;
    for (int i = 0; i < 4; i++) {
      int curx = x + dx[i], cury = y + dy[i];
      if (curx < 0 || curx >= 3 || cury < 0 || cury >= 3) continue;

      swap(state[x * 3 + y], state[curx * 3 + cury]);
      if (!dis.count(state) || dis[state] > step + 1) {
        dis[state] = step + 1;
        pre[state] = { backup,op[i] };
        heap.push({ dis[state] + f(state),state });
      }
      swap(state[x * 3 + y], state[curx * 3 + cury]);  // 恢复现场
    }
  }

  string res;
  while (end != start) {
    res += pre[end].second;
    end = pre[end].first;
  }
  reverse(all(res));
  return res;
}

int main() {
  string s1, s2;  // s1为原序列,s2为去掉x的序列
  char c;
  while (cin >> c) {
    s1.push_back(c);
    if (c != 'x') s2.push_back(c);
  }

  int cnt = 0;  // 逆序对个数
  for (int i = 0; i < s2.size(); i++)
```

```
    for (int j = i + 1; j < s2.size(); j++)
      if (s2[i] > s2[j]) cnt++;

  if (cnt & 1) cout << "unsolvable";
  else cout << bfs(s1);
}
```

14.9.2 第k短路

题意

给定一个包含n个点(编号$1 \sim n$)、m条边的有向图,求起点S到终点T的第k短路的长度,路径允许重复经过点或边.要求每条最短路中至少包含一条边,即$S = T$时,$k++$.

第一行输入整数n, m $(1 \leq n \leq 1000, 0 \leq m \leq 1e4)$.接下来$m$行每行输入三个整数$u, v, w$ $(1 \leq u, v \leq n, 1 \leq l \leq 100)$.最后一行输入三个整数$S, T, k$ $(1 \leq S, T \leq n, 1 \leq k \leq 100)$,表示求起点$S$到终点$T$的第$k$短路.

若最短路存在,输出最短路长度;否则输出-1.

思路

与最短路问题不同,扩展时应将当前点能扩展到的所有点都入队.显然路径中存在环时,起点到终点的路径可能有无数条,故搜索空间很大,考虑A*算法.

估价函数可取每个点到终点的最短距离,该距离可通过以终点为起点跑一遍Dijksta算法得到.

A*算法中当终点出队时确定最短路,一直出队至第k次即第k短路.

[**证**] 仿照终点出队时确定最短路的方法即证.

代码

```
const int MAXN = 1005, MAXM = 2e5 + 5;  // 两倍边
int n, m;  // 点数、边数
int S, T, k;  // 起点S到终点T的第k短路
int head[MAXN], rhead[MAXN], edges[MAXM], w[MAXM], nxt[MAXM], idx = 0;  // head[]为正向边的头节点,rhead[]为反向边的头节点
int dis[MAXN];
bool vis[MAXN];
int cnt[MAXN];  // cnt[u]表示节点u出队的次数

void add(int h[], int a, int b, int c) {
  edges[idx] = b, w[idx] = c, nxt[idx] = h[a], h[a] = idx++;
}

void dijkstra() {
  priority_queue<pii, vii, greater<pii>> heap;
  heap.push({ 0,T });  // 终点
  memset(dis, INF, so(dis));
  dis[T] = 0;

  while (heap.size()) {
    auto tmp = heap.top(); heap.pop();
```

```
21
22      int u = tmp.second;
23      if (vis[u]) continue;
24
25      vis[u] = true;
26      for (int i = rhead[u]; ~i; i = nxt[i]) {
27        int v = edges[i];
28        if (dis[v] > dis[u] + w[i]) {
29          dis[v] = dis[u] + w[i];
30          heap.push({ dis[v],v });
31        }
32      }
33    }
34  }
35
36  int Astar() {
37    priority_queue<tiii, vector<tiii>, greater<tiii>> heap;  // 估价距离、真实距离、节点编号
38    heap.push({ dis[S],0,S });
39
40    while (heap.size()) {
41      auto tmp = heap.top(); heap.pop();
42
43      int d, u;  // 真实距离、节点编号
44      tie(ignore, d, u) = tmp;
45      cnt[u]++;
46      if (cnt[T] == k) return d;  // 终点出队k次即找到第k短路
47
48      for (int i = head[u]; ~i; i = nxt[i]) {
49        int v = edges[i];
50        if (cnt[v] < k) heap.push({ d + w[i] + dis[v],d + w[i],v });  // 出队不足k次才扩展
51      }
52    }
53    return -1;  // 无解
54  }
55
56  int main() {
57    memset(head, -1, so(head)), memset(rhead, -1, so(rhead));
58
59    cin >> n >> m;
60    for (int i = 0; i < m; i++) {
61      int a, b, c; cin >> a >> b >> c;
62      add(head, a, b, c), add(rhead, b, a, c);  // 分别建正向边和反向边
63    }
64
65    cin >> S >> T >> k;
66    if (S == T) k++;  // 最短路至少包含一条边
67
68    dijkstra();  // 求终点到每个节点的最短距离
69
70    cout << Astar();
71  }
```

14.10 Flood Fill算法的DFS实现

连通性模型既可用DFS,又可用BFS,如树和图的遍历.BFS第一次搜到目标点时不仅能确定连通性,还能确定最短距离,而DFS只能确定连通性.

14.10.1 迷宫

题意

有一$n \times n$的迷宫,每个格子有两种状态'.'和'#',前者表示可以通行,后者表示不能同行.玩家在每个格子时只能移动到上下左右4个相邻的格子.现你要从迷宫中的A点走到B点,问在不走出迷宫的前提下能否办到.若起点或终点有一个不能通行,视为不能办到.

有t组测试数据.每组测试数据第一行输入整数n $(1 \leq n \leq 100)$.第二行起输入一个$n \times n$的字符矩阵,描述迷宫.接下来一行输入四个整数x_a, y_a, x_b, y_b,表示A点坐标为(x_a, y_a),B点坐标为(x_b, y_b),下标都从0开始.

对每组测试数据,若能办到,则输出"YES";否则输出"NO".

代码

```cpp
const int MAXN = 105;
int n;
char graph[MAXN][MAXN];
int xa, ya, xb, yb;
bool vis[MAXN][MAXN];

bool dfs(int x, int y) {
  if (graph[x][y] == '#') return false;
  if (x == xb && y == yb) return true;

  vis[x][y] = true;
  for (int i = 0; i < 4; i++) {
    int curx = x + dx[i], cury = y + dy[i];
    if (curx < 0 || curx >= n || cury < 0 || cury >= n) continue;
    if (vis[curx][cury]) continue;

    if (dfs(curx, cury)) return true;
  }
  return false;
}

int main() {
  CaseT{
    memset(vis,0,so(vis));

    cin >> n;
    for (int i = 0; i < n; i++) cin >> graph[i];
    cin >> xa >> ya >> xb >> yb;

    cout << (dfs(xa, ya) ? "YES" : "NO") << endl;
  }
}
```

14.10.2 红与黑

题意

有一间长方形的房子,地上铺了红、黑两种颜色的正方形瓷砖.你站在其中一块黑色瓷砖上,只能移动到上、下、左、右四个相邻的黑色瓷砖.问最多能到达多少块黑色瓷砖.

有多组测试数据.每组测试数据第一行输入整数w, h $(1 \leq w, h \leq 20)$,分别表示x、y方向的瓷砖数.第二行起输入一个$h \times w$的字符矩阵,每个字符表示一块瓷砖的颜色,其中'.'表示黑色瓷砖,'#'表示红色瓷砖,'@'表示起点的黑色瓷砖.数据保证'@'在每个矩阵中只出现一次.当一行输入0 0时表示输入结束.

对每组测试数据,输出从起点出发能到达的黑色瓷砖数,包含起点的瓷砖.

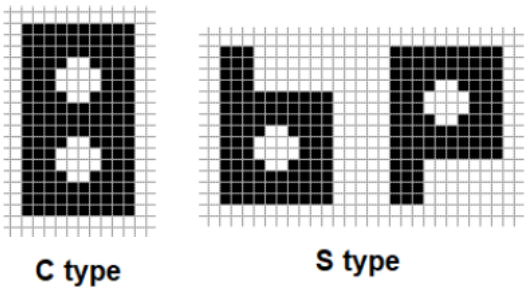
代码

```cpp
const int MAXN = 25;
int n, m;
char graph[MAXN][MAXN];
bool vis[MAXN][MAXN];

int dfs(int x, int y) {
  int res = 1;

  vis[x][y] = true;
  for (int i = 0; i < 4; i++) {
    int curx = x + dx[i], cury = y + dy[i];
    if (curx < 0 || curx >= n || cury < 0 || cury >= m) continue;
    if (graph[curx][cury] != '.') continue;
    if (vis[curx][cury]) continue;

    res += dfs(curx, cury);
  }
  return res;
}

int main() {
  while (cin >> m >> n, n || m) {  // 注意先输入的是列数
    memset(vis, 0, so(vis));

    for (int i = 0; i < n; i++) cin >> graph[i];

    int x, y;  // 起点坐标
    for (int i = 0; i < n; i++) {
      for (int j = 0; j < m; j++) {
        if (graph[i][j] == '@') {
          x = i, y = j;
          break;
        }
      }
    }

    cout << dfs(x, y) << endl;
  }
}
```

14.10.3 BpbBppbpBB

原题指路:https://codeforces.com/gym/103687/problem/M

题意



有如上图所示的两种印章,盖在纸上时可旋转90°,两个印章可相邻,但不能重叠.给定一个$n \times m$ $(1 \leq n, m \leq 1000)$的字符矩阵描述盖印后的纸,其中'#'表示黑格子,'.'表示白格子.分别求C型、S型印章的使用次数.

思路

C型印章的黑格子数为146,S型印章的黑格子数为100.

设C型、S型印章分别用了x、y个.因印章不重叠,则黑格子数为$146x + 100y$,洞数为$2x + y$,联立解出x和y即可.

此处不使用白格子数列方程是因为纸上未盖印的部分也是白格子.

考虑如何统计洞数.从上往下、从左往右扫一遍字符矩阵,遇到白格子时做一遍DFS,统计每个连通块中白格子的个数,若为12,检查该白格子的附近是否满足洞的特征.

代码

```
const int MAXN = 1005;
int n, m;
string graph[MAXN];
bool vis[MAXN][MAXN];  // 记录每个格子是否已遍历过

void dfs(int x, int y, int& white) {  // 统计每个连通块中白格子的个数
  white++;
  vis[x][y] = true;

  for (int i = 0; i < 4; i++) {
    int curx = x + dx[i], cury = y + dy[i];
    if (curx < 0 || curx >= n || cury < 0 || cury >= m || graph[curx][cury] == '#')
continue;

    if (!vis[curx][cury]) dfs(curx, cury, white);
  }
}

bool check(int x, int y) {  // 检查白格子所在的连通块是否是洞:(x,y)为洞的第一行第一个白格子
  if (y < 4) return false;

  if (graph[x][y + 1] == '.'  // 第一行
    && graph[x + 1][y - 1] == '.' && graph[x + 1][y] == '.' && graph[x + 1][y + 1] == '.'
&& graph[x + 1][y + 2] == '.'  // 第二行
    && graph[x + 2][y - 1] == '.' && graph[x + 2][y] == '.' && graph[x + 2][y + 1] == '.'
&& graph[x + 2][y + 2] == '.'  // 第三行
```

```
    && graph[x + 3][y] == '.' && graph[x + 3][y + 1] == '.') return true;  // 第四行
  else return false;
}

void solve() {
  cin >> n >> m;
  for (int i = 0; i < n; i++) cin >> graph[i];

  int black = 0, hole = 0;
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < m; j++) {
      if (graph[i][j] == '#') black++;
      else {  // 白格子
        if (!vis[i][j]) {
          int white = 0;
          dfs(i, j, white);
          if (white == 12) hole += check(i, j);
        }
      }
    }
  }

  cout << (100 * hole - black) / 54 << ' ' << hole - (100 * hole - black) / 27;
}

int main() {
  solve();
}
```

14.11 DFS的搜索顺序

14.11.1 马走日

题意

中国象棋中的马以"日"字规则移动.

给定一个$n \times m$的棋盘和马的初始位置(x, y),要求不重复经过棋盘上的同一点,问马有多少种途径遍历完棋盘上的所有点.

有t $(1 \le t \le 9)$组测试数据.每组测试数据输入四个整数
n, m, x, y $(1 \le m, n \le 9, 0 \le x \le n-1, 0 \le y \le m-1)$.

对每组测试数据,输出马有多少种途径遍历完棋盘上的所有点.若无途径,输出0.

代码

```
const int MAXN = 10;
int n, m;
int x, y;
bool vis[MAXN][MAXN];
int ans;

void dfs(int x, int y, int cnt) {  // 当前在点(x,y),当前在搜第cnt个点
  static const int dx[8] = { -2,-1,1,2,2,1,-1,-2 }, dy[8] = { 1,2,2,1,-1,-2,-2,-1 };
```

```
9
10    if (cnt == n * m) {
11      ans++;
12      return;
13    }
14
15    vis[x][y] = true;
16    for (int i = 0; i < 8; i++) {
17      int curx = x + dx[i], cury = y + dy[i];
18      if (curx < 0 || curx >= n || cury < 0 || cury >= m) continue;
19      if (vis[curx][cury]) continue;
20
21      dfs(curx, cury, cnt + 1);
22    }
23    vis[x][y] = false;  // 恢复现场
24  }
25
26  int main() {
27    CaseT{  // 注意先输入的是列数
28      ans = 0;
29      memset(vis, 0, so(vis));
30
31      cin >> n >> m >> x >> y;
32
33      dfs(x, y, 1);  // 从起点出发,当前在搜第1个点
34
35      cout << ans << endl;
36    }
37  }
```

14.11.2 单词接龙

题意

已知一组单词,给定一个开头字母,求以该字母开头的最长的"龙",每个单词最多用两次.两单词相连时,其重合部分合为一部分,如beast和astonish相连时,变为beastonish.重合部分长度任意,但必须$\geq$ 1且严格小于两串的长度,如at与atide不能相连.

第一行输入一个整数n $(1 \leq n \leq 20)$表示单词数.接下来n行每行输入一个单词(只含大小写字母,长度不超过20),最后一行输入单个字符,表示"龙"的开头字母,数据保证以该字母开头的"龙"存在.

输出以"龙"的开头字母开头的"龙"的最大长度.

思路

最多20个单词,每个单词最多用2次,故搜索深度最大为40.

为使得龙尽可能长,单词的重合部分应尽可能短.

代码

```cpp
const int MAXN = 21;
int n;
string words[MAXN];
int len[MAXN][MAXN];  // len[i][j]表示单词i的后缀与单词j的前缀重合的最短长度
int used[MAXN];  // 记录每个单词用的次数
int ans;

void dfs(string dragon, int last) {  // 当前的龙、最后一个接的单词
  ans = max(ans, (int)dragon.length());

  used[last]++;
  for (int i = 0; i < n; i++)  // 枚举接在后面的单词
    if (len[last][i] && used[i] < 2) dfs(dragon + words[i].substr(len[last][i]), i);
  used[last]--;  // 恢复现场
}

int main() {
  cin >> n;
  for (int i = 0; i < n; i++) cin >> words[i];
  char start; cin >> start;

  // 预处理len[][]
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
      string a = words[i], b = words[j];
      for (int k = 1; k < min(a.size(), b.size()); k++) {  // 枚举重合长度
        if (a.substr(a.size() - k, k) == b.substr(0, k)) {
          len[i][j] = k;
          break;
        }
      }
    }
  }

  for (int i = 0; i < n; i++)
    if (words[i][0] == start) dfs(words[i], i);

  cout << ans;
}
```

14.11.3 分成互素组

题意

给定n $(1 \leq n \leq 10)$个$\leq$ 1e4的正整数,将它们分为若干组,使得每组中的数两两互素.问至少要分为几组.

思路

每个数视为一个点,若两数互素则连边,转化为图论问题.

依次枚举每个组放哪些数,有两种情况:①将当前数放在最后一组中;②新开一个组放当前数.显然②随时可做,但①未必可做.若能做①,则做①比②更优.

按数的下标顺序依次枚举每个数,可保证组中的元素不重复,且搜索空间小.

代码

```cpp
const int MAXN = 15;
int n;
int p[MAXN];
int group[MAXN][MAXN];
bool vis[MAXN];
int ans = MAXN;

bool check(int g[], int cnt, int idx) {  // 检查p[idx]能否放入含有cnt个元素的组g[]中
  for (int i = 0; i < cnt; i++)
    if (gcd(p[g[i]], p[idx]) > 1) return false;
  return true;
}

// 当前组号、当前组内有多少个数、已将多少个数分组、可从哪个下标开始枚举数
void dfs(int g, int cnt, int tot, int start) {
  if (g >= ans) return;  // 剪枝
  if (tot == n) ans = g;  // 放完了

  bool flag = false;  // 记录当前元素能否放入已有的组中
  for (int i = start; i < n; i++) {
    if (!vis[i] && check(group[g], cnt, i)) {
      vis[i] = true;
      group[g][cnt] = i;
      dfs(g, cnt + 1, tot + 1, i + 1);
      vis[i] = false;  // 恢复现场

      flag = true;
    }
  }

  if (!flag) dfs(g + 1, 0, tot, 0);  // 新开一个组
}

int main() {
  cin >> n;
  for (int i = 0; i < n; i++) cin >> p[i];

  dfs(1, 0, 0, 0);  // 从第1组开始枚举,当前组内有9个数,已将0个数分组,可从下标0开始枚举数

  cout << ans;
}
```

14.12 DFS的剪枝与优化

DFS剪枝:

①优化搜索顺序: 大部分情况下, 应优先搜索分支较少的节点.

②排除等效冗余: 若不考虑顺序, 尽量用组合的方式搜索.

③可行性剪枝: 搜到不合法的方案返回.

④最优性剪枝: 如求最小花费时, 若当前方案的花费已>当前的最小花费, 则返回.

⑤记搜: 即DP.

14.12.1 小猫爬山

题意

有n $(1 \leq n \leq 18)$只猫,重量分别为$c_1, \cdots, c_n$.一辆缆车的最大承重为w $(1 \leq c_i \leq w \leq 1e8)$,问至少需要多少辆缆车才能将载上所有猫.

思路

将猫按重量降序排序,依次枚举每只猫放在当前已有的缆车上还是新开一辆车.

代码

```cpp
const int MAXN = 20;
int n, w;  // 猫数、每辆缆车的最大承重
int c[MAXN];
int sum[MAXN];  // 每辆缆车上的重量
int ans = MAXN;

void dfs(int pos, int car) {  // 当前要放下标为pos的猫,当前缆车数为car
  if (car >= ans) return;  // 最优性剪枝

  if (pos == n) {
    ans = car;
    return;
  }

  for (int i = 0; i < car; i++) {
    if (sum[i] + c[pos] <= w) {  // 可行性剪枝
      sum[i] += c[pos];
      dfs(pos + 1, car);
      sum[i] -= c[pos];  // 恢复现场
    }
  }

  // 新开一辆车
  sum[car] = c[pos];
  dfs(pos + 1, car + 1);
  sum[car] = 0;  // 恢复现场
}

int main() {
  cin >> n >> w;
  for (int i = 0; i < n; i++) cin >> c[i];

  sort(c, c + n, greater<int>());  // 优化搜索顺序
  dfs(0, 0);  // 当前要放下标为0的猫,当前缆车数为0

```

```
  cout << ans;
}
```

14.12.2 数独

题意

有多组测试数据,每组测试数据输入一个包含81个字符的字符串,描述当前的数独.其中每个字符都是$1 \sim 9$的数字或'.'(表示未填充).最后一行输入"end"表示输入结束.

对每组测试数据,输出填好的数独,数据保证有唯一解.

思路

每次选择一个空格子,枚举它可填入的数.

分别用一个9位二进制数记录每一行、每一列、每个九宫格中用到的数.

代码

```
const int MAXN = 9, MAXM = 1 << MAXN;
int row[MAXN], col[MAXN], cell[3][3];  // 行、列、九宫格
char str[100];
int ones[MAXM];  // ones[u]表示u的二进制表示中所含1的个数
int lg2[MAXM];  // log[u]表示log_2(u)

void init() {  // 初始化row[],col[],cell[][]为每个数都可填
  for (int i = 0; i < MAXN; i++) row[i] = col[i] = (1 << MAXN) - 1;  // 初始化为111111111,即每个数都能用
  for (int i = 0; i < 3; i++)
    for (int j = 0; j < 3; j++) cell[i][j] = (1 << MAXN) - 1;
}

void draw(int x, int y, int t, bool op) {  // 在(x,y)填t,op为true表示填入,为false表示删去(恢复现场)
  if (op) str[x * MAXN + y] = t + '1';  // 注意从1开始
  else str[x * MAXN + y] = '.';  // 恢复现场

  int v = 1 << t;
  if (!op) v = -v;
  row[x] -= v, col[y] -= v, cell[x / 3][y / 3] -= v;  // 填入或删去对应数
}

int get(int x, int y) { return row[x] & col[y] & cell[x / 3][y / 3]; }  // (x,y)能填哪些数

bool dfs(int space) {
  if (!space) return true;  // 填完

  int mincnt = 10;  // 记录最少能填多少数
  int x, y;  // 记录能填的数最少的空格位置
  for (int i = 0; i < MAXN; i++) {
    for (int j = 0; j < MAXN; j++) {
      if (str[i * MAXN + j] == '.') {
        int state = get(i, j);  // 该位置能填哪些数
```

```
        if (ones[state] < mincnt) {
          mincnt = ones[state];
          x = i, y = j;
        }
      }
    }
  }

  int state = get(x, y);  // 该位置能填哪些数
  for (int i = state; i; i -= lowbit(i)) {
    int t = lg2[lowbit(i)];
    draw(x, y, t, true);  // 填入
    if (dfs(space - 1)) return true;
    draw(x, y, t, false);  // 删除
  }

  return false;
}

int main() {
  // 预处理ones[],log[]
  for (int i = 0; i < 1 << MAXN; i++)
    for (int j = 0; j < MAXN; j++) ones[i] += i >> j & 1;
  for (int i = 0; i < MAXN; i++) lg2[1 << i] = i;

  while (cin >> str, str[0] != 'e') {
    init();

    int space = 0;  // 空位数
    for (int i = 0, k = 0; i < MAXN; i++) {
      for (int j = 0; j < MAXN; j++, k++) {
        if (str[k] != '.') {
          int t = str[k] - '1';
          draw(i, j, t, true);  // 填入对应数
        }
        else space++;
      }
    }

    dfs(space);

    cout << str << endl;
  }
}
```

14.12.3 木棒

题意

有一组等长的木棒,某人将它们随机锯断,使得每节木棍的长度(大于0的整数)都不超过50.现他忘了初始时有多少木棒和木棒的初始长度,求初始时木棒可能的最小长度.

有多组测试数据,每组测试数据第一行和输入一个整数n $(1 \leq n \leq 64)$,表示当前有多少节木棍.第二行输入n个不超过50的整数,表示各节木棍的长度.最后一行输入0表示输入结束.

对每组测试数据,输出原始木棒可能的最小长度.

思路

枚举木棒长度,再枚举它是由哪些木棍拼起来的.

剪枝:

①设给定木棍的长度之和为sum,则原始木棒的长度是sum的约数.

②先枚举较长的木棍.

③按组合方式枚举,因为同一组木棍无论拼接顺序,拼出来的长度都一样.

④在拼某根木棒时,若发现长度为l的木棍不能构成合法解,则后面长度为l的木棍也不能组成合法解,跳过.

⑤在拼某根木棒时,若发现第一根木棍不能构成合法解,则该原始长度无解.

[**证**] 设当前在拼木棒2时,发现木棍x不能构成合法解.若该原始长度有解,则木棍x在另一根木棒y中.

将x换到y的开头,再将木棒y作为木棒2,则木棒2中x放在开头能构成合法解,矛盾.

⑥在拼某根木棒时,若发现最后一根木棍不能构成合法解,则该原始长度无解.

代码

```cpp
const int MAXN = 70;
int n;
int a[MAXN];  // 木棍长度
int length;  // 原始木棒长度
int sum;  // 木棍长度之和
bool vis[MAXN];

bool dfs(int cnt, int cur, int start) {  // 当前枚举到第pos根木棒,其长度为cur,从下标start开始枚举木棍
  if (cnt * length == sum) return true;  // 找到合法解
  if (cur == length) return dfs(cnt + 1, 0, 0);  // 已拼出长度为length的木棒,继续搜下一根

  for (int i = start; i < n; i++) {
    if (vis[i] || cur + a[i] > length) continue;  // 用过或加上后长度超过length

    vis[i] = true;
    if (dfs(cnt, cur + a[i], start + 1)) return true;
    vis[i] = false;

    if (!cur) return false;  // 是当前木棒的第一根木棍
    if (cur + a[i] == length) return false;  // 是当前木棒的最后一根木棍

    int j = i;
    while (j < n && a[j] == a[i]) j++;  // 跳到下一根长度不相等的木棍
    i = j - 1;
  }

  return false;
}

int main() {
  while (cin >> n, n) {
    memset(vis, 0, so(vis));
    sum = 0;
```

```
34
35      int maxlen = 0;
36      for (int i = 0; i < n; i++) {
37        cin >> a[i];
38        sum += a[i];
39        maxlen = max(maxlen, a[i]);
40      }
41
42      sort(a, a + n, greater<int>());  // 先枚举长的木棍
43
44      length = maxlen;  // 原始木棒长度至少为木棍的最长长度
45      while (true) {
46        if (sum % length == 0 && dfs(0, 0, 0)) {
47          cout << length << endl;
48          break;
49        }
50        else length++;
51      }
52    }
53  }
```

14.12.4 生日蛋糕

题意

现要制作一个体积为$n\pi$的m层蛋糕,每层都是一个圆柱体,从下往上数第i层蛋糕是半径为r_i、高度为h_i的圆柱.$i < m$时,要求$r_i > r_{i+1}, h_i > h_{i+1}$.现要给蛋糕外表面涂奶油,希望蛋糕外表面(不含底面)的面积Q最小.设$Q = S\pi$,求一组适当的$r_i, h_i\ s.t.\ S$最小.除Q外,上述数据皆为正整数.

第一行输入整数$n\ (1 \le n \le 1e4)$,表示要制作的蛋糕的体积为$n\pi$.第二行输入整数$m\ (1 \le m \le 20)$,表示要制作的蛋糕层数为m.

输出一个正整数,表示最小的S值.若无解,输出0.

思路

从上往下依次为第$1 \sim m$层.注意到各层蛋糕的顶面要涂奶油的面积之和为最底层蛋糕的底面积,则$Q = \pi r_m^2 + 2\pi(r_m h_m + r_{m-1} h_{m-1} + \cdots + r_1 h_1)$.

自底向上搜,因表达式中r是平方级别,h是一次方级别,故应先枚举r再枚举h,且从大到小枚举.

剪枝:

①设当前枚举到第u层.因r_u是整数,故$r_u \ge u$.设第$(u+1) \sim m$层的总体积为V,则$n\pi - V \ge \pi R^2 h$.

若将V中的π去掉得到V',因h最小取1,则$R \le \sqrt{n - V}$.

结合$r_u < r_{u+1}$知:$u \le r_u \le \min\left\{r_{u+1} - 1, \sqrt{n - V}\right\}$,同理$u \le h_u \le \min\left\{h_{u+1} - 1, \dfrac{n - V}{R^2}\right\}$.

②预处理前u层的体积和表面积的最小值$minV(u)$和$minS(u)$,即$r, h = 1, 2, \cdots$时的体积和表面积.

则$V + minV(u) \le n, S + minS(u) < ans$.

③第$1 \sim u$的表面积$S_{1\sim u} = \sum_{k=1}^{u} 2r_k h_k = \dfrac{2}{r_{u+1}} \sum_{k=1}^{u} r_k h r_{u+1}$. (忽略$\pi$)

因$r_k > r_{u+1}$,则$S_{1\sim u} > \dfrac{2}{r_{u+1}}\sum_{k=1}^{u} r_k^2 h_k = \dfrac{2}{r_{u+1}}(n-V).$

若当前表面积为S,则$S + \dfrac{2(n-V)}{r_{u+1}} > ans$时可退出.

代码

```cpp
const int MAXN = 25;
int n, m;  // 蛋糕的体积、层数
int R[MAXN], H[MAXN];  // 每一层的半径、高
int minv[MAXN], mins[MAXN];  // 每一层最小的体积和表面积
int ans = INF;

void dfs(int pos, int v, int s) {  // 当前在搜第pos层,当前体积为v,表面积为s
  if (v + minv[pos] > n || s + mins[pos] >= ans) return;  // 加上最小的体积或表面积后已不是最优解
  if (s + 2 * (n - v) / R[pos + 1] >= ans) return;  // 当前s加上估算的表面积后已不是最优解


  if (!pos) {  // 搜到顶层
    if (v == n) ans = s;  // 找到解
    return;
  }

  for (int r = min(R[pos + 1] - 1, (int)sqrt(n - v)); r >= pos; r--) {
    for (int h = min(H[pos + 1] - 1, (n - v) / r / r); h >= pos; h--) {
      int tmp = 0;
      if (pos == m) tmp = r * r;  // 顶层的体积单独算
      R[pos] = r, H[pos] = h;
      dfs(pos - 1, v + r * r * h, s + 2 * r * h + tmp);
    }
  }
}

int main() {
  cin >> n >> m;

  for (int i = 1; i <= m; i++) {  // 预处理出minv[]和mins[]
    minv[i] = minv[i - 1] + i * i * i;
    mins[i] = mins[i - 1] + 2 * i * i;
  }

  R[m + 1] = H[m + 1] = INF;  // 哨兵

  dfs(m, 0, 0);  // 从最底层第m层开始,当前体积和面积为0

  cout << (ans == INF ? 0 : ans);
}
```

14.13 迭代加深搜索

有时搜索树中不合法的方案的节点可能很深,但合法的方案的节点可能很浅.

迭代加深搜索按层搜索,每次定义一个层数上限,搜索层数超过层数上限的部分直接return.若当前层数上限的范围内能搜到解,则找到解;否则提高层数上限继续搜索.

迭代加深搜索与BFS的区别:BFS有队列,时间复杂度是指数级别;迭代加深搜索本质是DFS,只记录路径上的信息,时间复杂度与高度h成正比.

迭代加深搜索不会特别浪费时间的原因:如合法方案在第10层时,设前9层的节点都有两个分支,则前9层的节点数之和$(2^1+2^2+\cdots+2^9)=2^{10}-1$,小于第$10$层的节点数$2^{10}$.若前$9$层的节点分治更多,则前$9$层的节点数之和小于第$10$层的节点数.

14.13.1 加成序列

题意

给定一个元素下标从1开始的"加成序列"x,它满足如下性质:①$x[1]=1$;②$x[m]=n$;③$x[1]<x[2]<\cdots<x[m]$;④对$\forall k\in[2,m],\exists i,j\in\mathbb{Z},1\le i,j\le k-1\ s.t.\ x[k]=x[i]+x[j]$.

有多组测试数据.每组测试数据输入整数$n\ (1\le n\le 100)$,最后一行输入0表示输入结束.

对每组测试样例,求一个符合上述条件的长度m最小的"加成序列".若有多组解,输出任一解.

思路

首先肯定有解,$1,2,3\cdots$是个加成序列.

n最大100,朴素DFS的搜索树深度可能很大.考察加成序列$1,2,4,8,16,64,128$,可发现数字增长较快,则合法解应在搜素树中较浅的位置,考虑迭代加深搜索.

从前往后依次枚举加成序列中每个位置上的数可以填什么,显然第一个位置只能填1,第二个位置只能填2.

剪枝:

①对每个位置,优先枚举较大的数.

②排除等效冗余,若不同组的两个数之和相等,则只需枚举其中一组.

代码

```
const int MAXN = 105;
int n;
int ans[MAXN];

bool dfs(int pos, int depth) {  // 当前在填第pos位,层数上限为depth
  if (pos == depth) return ans[pos - 1] == n;

  bool vis[MAXN] = { false };  // 注意vis[]开在函数中,只保证同一层中不搜索重复的节点
  for (int i = pos - 1; i >= 0; i--) {
    for (int j = i; j >= 0; j--) {  // 规定i≥j,防止重复
      int tmp = ans[i] + ans[j];
      if (tmp > n || tmp <= ans[pos - 1] || vis[tmp]) continue;

      vis[tmp] = true;
      ans[pos] = tmp;
      if (dfs(pos + 1, depth)) return true;
```

```
    }
  }

  return false;
}

int main() {
  ans[0] = 1, ans[1] = 2;

  while (cin >> n, n) {
    int depth = 1;
    while (!dfs(1, depth)) depth++;


    for (int i = 0; i < depth; i++) cout << ans[i] << " \n"[i == depth - 1];
  }
}
```

14.14 双向DFS / 折半搜索

14.14.1 送礼物

题意 $(4\ \mathrm{s})$

某人准备了$n\ \ (1 \le n \le 46)$个礼物,其中第i个礼物的重量为g_i.他一次能搬重量之和不超过$w\ \ (1 \le w, g_i \le 2^{31}-1)$的任意多个物品.他希望一次搬掉尽量重的物品,求他在力气范围内一次性能搬动的最大重量.

思路

背包DP的时间复杂度是$O(nV)$,会TLE.

暴搜最坏2^{46},会T.

考虑优化,可先预处理前$\left\lfloor \frac{n}{2} \right\rfloor$个物品能组成的重量的集合$S$,则最坏有$2^{23} \approx 8\mathrm{e}6$个元素.对后$\left\lfloor \frac{n}{2} \right\rfloor$个物品的重量$g_i$,看$S$中是否存在最大的$x\ s.t.\ x+g_i \le w$,即求$S$中$\le w-g_i$的最大值,可用二分求得.

总时间复杂度$O\left(2^{\frac{n}{2}} + 2^{\frac{n}{2}} \cdot \log_2 2^{\frac{n}{2}}\right)$,最坏$2^{23} \times 24$,还是可能TLE.考虑均衡时间复杂度的两项,让它们更相近.设时间复杂度为$2^k + k \cdot 2^{n-k}$,考虑让第一项多预处理一些,则第二项会少暴搜一些,尝试得可取$2^{25} + 2^{21} \times 25 \approx 5\mathrm{e}7$.故可取$k = \left\lfloor \frac{n}{2} \right\rfloor + 2$.但$n = 1$时这样取会死循环,故还是取$k = \left\lfloor \frac{n}{2} \right\rfloor$.

可先给物品重量降序排列,先搜重量更大的物品.

代码

```
const int MAXN = 1 << 24;
int n, w;  // 礼品数、最大承重
int k;  // k=n/2
int gifts[50];
ll weights[MAXN];  // 前k个物品物品能组成的重量
int cnt;  // 前k个物品物品能组成的重量的个数
ll ans;
```

```cpp

void dfs1(int pos, ll sum) {  // 预处理前k个物品物品能组成的重量
  if (pos == k) {
    weights[cnt++] = sum;
    return;
  }

  if (sum + gifts[pos] <= w) dfs1(pos + 1, sum + gifts[pos]);  // 选该礼品
  dfs1(pos + 1, sum);  // 不选该礼品
}

void dfs2(int pos, ll sum) {  // 后k个物品与weights[]配对
  if (pos == n) {
    int l = 0, r = cnt - 1;
    while (l < r) {
      int mid = l + r + 1 >> 1;
      if (sum + weights[mid] <= w) l = mid;
      else r = mid - 1;
    }

    if (sum + weights[l] <= w) ans = max(ans, sum + weights[l]);

    return;
  }

  if (sum + gifts[pos] <= w) dfs2(pos + 1, sum + gifts[pos]);  // 选该礼品
  dfs2(pos + 1, sum);  // 不选该礼品
}

int main() {
  cin >> w >> n;
  for (int i = 0; i < n; i++) cin >> gifts[i];

  sort(gifts, gifts + n, greater<int>());

  k = n / 2;
  dfs1(0, 0);  // 预处理出weights[]

  sort(weights, weights + cnt);

  // 对weights[]去重
  int idx = 1;
  for (int i = 1; i < cnt; i++)
    if (weights[i] != weights[i - 1]) weights[idx++] = weights[i];
  cnt = idx;

  dfs2(k, 0);  // 后k个物品与weights[]配对

  cout << ans;
}
```

14.14.2 Maximum Subsequence

原题指路: https://codeforces.com/problemset/problem/888/E

题意

给定一个长度为 n $(1 \le n \le 35)$ 的序列 $a = [a_1, \cdots, a_n]$ $(1 \le a_i \le 1e9)$ 和一个整数 m $(1 \le m \le 1e9)$. 现需从 $a[]$ 中选出若干个元素 $a_{b_1}, \cdots, a_{b_k}$ $(1 \le b_i \le n)$ $s.t.$ $\sum_{i=1}^{k}(a_{b_i} \bmod m)$ 最大, 求其最大值.

思路

朴素的子集枚举的时间复杂度为 $O(2^n \cdot n)$ 或 $O(2^n)$, 都会 TLE.

考虑折半搜索. 先对 $a[]$ 的前 $\left\lfloor \frac{n}{2} \right\rfloor$ 个元素暴力枚举子集, 将子集的元素和模 m 的值加入序列 $sums[]$ 中, 将序列升序排列.

枚举 $a[]$ 的后 $\left\lceil \frac{n}{2} \right\rceil$ 个元素, 设当前方案的集合的元素和模 m 的余数为 sum, 则最大值可能在如下两种情况取得:

① sum 与 $sums[]$ 中的最大元素之和.

② sum 与 $sums[]$ 中最大的 $s.t.$ $sum + tmp < m$ 的最大的元素 tmp 之和, tmp 可在 $sums[]$ 中二分求得.

对上述两种情况取 $\max$ 即可. 总时间复杂度 $O\left(2^{\lfloor \frac{n}{2} \rfloor} \cdot \log 2^{\lceil \frac{n}{2} \rceil}\right)$.

因用到了 $a[]$ 的前 $\left\lfloor \frac{n}{2} \right\rfloor$ 个元素, 故需特判 $n = 1$ 的情况, 此时用上述算法会RE.

代码

```cpp
void solve() {
    int n, m; cin >> n >> m;
    vector<int> a(n + 1);
    for (int i = 1; i <= n; i++) cin >> a[i];

    if (n == 1) {
        cout << a[1] % m << endl;
        return;
    }

    vector<int> sums;

    // 预处理a[]的前(n / 2)的子集的元素和模m的结果
    function<void(int, int)> dfs1 = [&](int pos, int sum) {
        if (pos == n / 2) {
            sums.push_back(sum);
            return;
        }

        dfs1(pos + 1, sum), dfs1(pos + 1, (sum + a[pos]) % m);
    };

    dfs1(1, 0);
    sort(all(sums));

    int ans = 0;

    function<void(int, int)> dfs2 = [&](int pos, int sum) {
```

```
29          if (pos > n) {
30              ans = max(ans, (sums.back() + sum) % m);
31
32              auto it = lower_bound(all(sums), m - sum);  // 最靠前的 >= m - sum的位置
33              if (it != sums.begin()) {
34                  it = prev(it);  // 最靠后的 < m - sum的位置
35                  ans = max(ans, *it + sum);
36              }
37              return;
38          }
39
40          dfs2(pos + 1, sum), dfs2(pos + 1, (sum + a[pos]) % m);
41      };
42
43      dfs2(n / 2, 0);
44      cout << ans << endl;
45  }
46
47  int main() {
48      solve();
49  }
```

14.15 IDA*

IDA*算法在迭代加深搜索的基础上加入了启发式的剪枝.在搜索每个节点时,会估算当前节点还需搜多少步才能得到答案.若发现当前节点可能所需的步数超过了当前迭代加深搜索的层数上限,则剪枝.估价函数需满足估计步数≤真实步数.

14.15.1 排书

题意

给定编号$1 \sim n$的n本书,初始时它们任意排列.每次操作中可取连续的一段,将该段插入到某个其他位置.求将书按编号升序排列最少需几次操作.

有t组测试数据.每组测试数据第一行输入整数n $(1 \leq n \leq 15)$,表示书的数量.第二行输入n个整数,表示这n本书的初始排列.

对每组测试数据,若最小操作次数不小于5步,则输出"5 or more";否则输出最小操作次数.

思路

求和公式$1 \times 2 + 2 \times 3 + \cdots + n(n+1) = \dfrac{n(n+1)(n+2)}{3}$.

考察搜索时每种情况有多少种选择.设选取长度为len的段,不妨设选取$1 \sim (n - len + 1)$,则剩下的$(n - len)$个数有$(n - len + 1)$个空,其中第一个空是所选取的段的原位置,则有$(n - len)$个空可以插.故每个len有$(n - len + 1) \times (n - len)$种选择.$n$最大为$15$,则最坏$\dfrac{15 \times 14 + 14 \times 13 + \cdots + 2 \times 1}{2} = 560$种选择,其中除以$2$是因为将段A换到段B后面等价于将段B换到段A前面,故每个操作都被重复计算了1次.因需要输出具体步数的至多为4步,则最坏560^4种选择,会TLE.此时用双向BFS可降低至560^2,可过.

下面叙述IDA*的做法.考察对每个给定的序列,至少需多少步才能将其排成一个升序序列.注意到升序排列后,每个元素的后继都是当前元素+1,考察一次操作会改变多少后继关系.显然将段A换到段B(A的右端点与B的左端点重合)的后面时,会改变A的左、右端点和B的右端点的后继关系,则最优时每次操作能修正3个错误的后继关系.设当前序列中有cnt个错误的后继关系,则估价函数可取$\left\lceil \frac{cnt}{3} \right\rceil = \left\lfloor \frac{cnt+2}{3} \right\rfloor$.

代码

```
const int MAXN = 15;
int n;
int a[MAXN];
int backup[5][MAXN];  // backup[depth][]表示搜索深度为depth时的a数组的备份

int f() {  // 估价函数
  int cnt = 0;
  for (int i = 0; i + 1 < n; i++)
    if (a[i + 1] != a[i] + 1) cnt++;
  return (cnt + 2) / 3;
}

bool check() {  // 检查当前a[]是否有序
  for (int i = 0; i + 1 < n; i++)
    if (a[i + 1] != a[i] + 1) return false;
  return true;
}

bool dfs(int depth, int max_depth) {
  if (depth + f() > max_depth) return false;  // 当前深度加上预计深度已超过当前的层数上限
  if (check()) return true;

  for (int len = 1; len <= n; len++) {  // 枚举区间长度
    for (int l = 0; l + len - 1 < n; l++) {  // 枚举区间左端点
      int r = l + len - 1;  // 区间右端点
      for (int k = r + 1; k < n; k++) {  // 将[l,r]换到k的后面,即交换[l,r]和[r+1,k]
        memcpy(backup[depth], a, so(a));
        int y = l;
        for (int x = r + 1; x <= k; x++, y++) a[y] = backup[depth][x];
        for (int x = l; x <= r; x++, y++) a[y] = backup[depth][x];
        if (dfs(depth + 1, max_depth)) return true;
        memcpy(a, backup[depth], so(a));
      }
    }
  }

  return false;
}

int main() {
  CaseT{
    cin >> n;
    for (int i = 0; i < n; i++) cin >> a[i];

    int depth = 0;
    while (depth < 5 && !dfs(0, depth)) depth++;

    cout << (depth >= 5 ? "5 or more" : to_string(depth)) << endl;
  }
```

```
50 }
```

14.15.2 回转游戏 (4 s)

题意

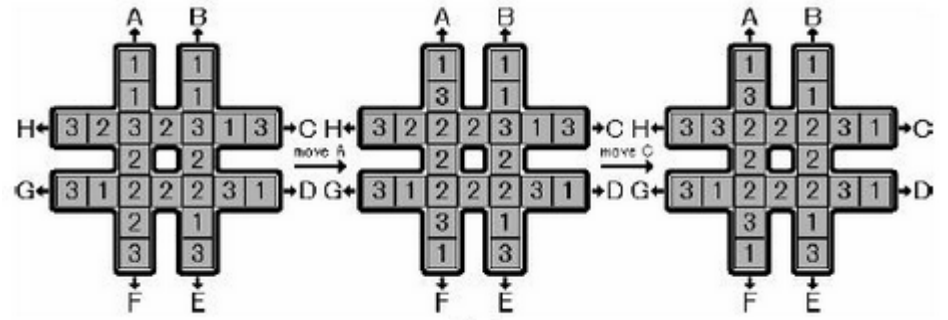


Fig.1

如上图是一个#形棋盘,其上有$1, 2, 3$三个数字各8个.给定8种操作,分别是上图中的A~H,这些操作会按图中字母和箭头所指的方向将长度为7的序列循环移动1个单位.如上图是先执行A操作再执行C操作得到的棋盘.给定一个初始状态,用最少的操作次数使得#的中间8个格子内的数字相同.

有多组测试数据.每组测试数据输入24个数字,表示初始棋盘中每一位上的数字,按整体从上到下、同行从左到右的顺序给出.输入一个只包含0的行表示输入结束.

对每组测试样例,第一行输入移动步骤,每步用大写字母A~H之一表示;若无需移动,输出"No moves needed".第二行输出一个整数,表示移动完成后中间8个格子内的数字.若有多种方案,输出字典序最小的方案.

思路

搜索时依次枚举操作A~H即可得到字典序最小的方案.

显然搜索树的深度很大,但解的深度不大,考虑IDA*.注意到每次操作只能改变中间八个格子中的一个数的出现次数,设当前中间的八个格子中出现次数最多的数出现了cnt次,则最少还需$(8-cnt)$步操作才能使得中间八个格子的数字相同,故估价函数可取$(8-cnt)$.

优化:若当前操作为A,则下一步操作不枚举F,否则两操作抵消.

因棋盘不规则,可先给每个格子和操作编号,再考虑每个操作改变哪些编号的格子:

```
          (0)    (1)
           A      B
           00    01
           02    03
(7)H 04 05 06 07 08 09 10 C(2)
           11    12
(6)G 13 13 15 16 17 18 19 D(3)
           20    21
           22    23
           F      E
          (5)    (4)
```

代码

```
const int MAXN = 24;
int op[8][7] = {  // 每个操作会移动的格子的编号
  {0, 2, 6, 11, 15, 20, 22},
  {1, 3, 8, 12, 17, 21, 23},
  {10, 9, 8, 7, 6, 5, 4},
  {19, 18, 17, 16, 15, 14, 13},
  {23, 21, 17, 12, 8, 3, 1},
  {22, 20, 15, 11, 6, 2, 0},
  {13, 14, 15, 16, 17, 18, 19},
  {4, 5, 6, 7, 8, 9, 10}
};
int oppsite[8] = { 5, 4, 7, 6, 1, 0, 3, 2 };  // 每个操作的逆操作的编号
int center[8] = { 6, 7, 8, 11, 12, 15, 16, 17 };  // 中间8个格子的编号

int a[MAXN];
int ans[100];  // 答案长度一开始不能确定,可开大一些

int f() {  // 估价函数
  // 统计中间8个格子中的数的出现次数
  int sum[4] = { 0 };
  for (int i = 0; i < 8; i++) sum[a[center[i]]]++;

  int cnt = 0;  // 出现次数最多的数的出现次数
  for (int i = 1; i <= 3; i++) cnt = max(cnt, sum[i]);
  return 8 - cnt;  // 至少还需(8-cnt)次操作可使得中间8个格子中的数相同
}

void operate(int x) {  // 执行操作x
  int tmp = a[op[x][0]];
  for (int i = 0; i < 6; i++) a[op[x][i]] = a[op[x][i + 1]];
  a[op[x][6]] = tmp;
}

bool dfs(int depth, int max_depth, int last) {  // last为上一个操作
  if (depth + f() > max_depth) return false;
  if (!f()) return true;  // 找到解

  for (int i = 0; i < 8; i++) {
    if (last != oppsite[i]) {  // 相邻两操作互为逆操作无意义
      operate(i);
      ans[depth] = i;
      if (dfs(depth + 1, max_depth, i)) return true;
      operate(oppsite[i]);  // 恢复现场
    }
  }

  return false;
}

int main() {
  while (cin >> a[0], a[0]) {
    for (int i = 1; i < 24; i++) cin >> a[i];

    int depth = 0;
    while (!dfs(0, depth, -1)) depth++;  // 一开始last=-1表示无上一个操作
```

```
56
57      if (!depth) cout << "No moves needed";
58      else
59        for (int i = 0; i < depth; i++) cout << (char)('A' + ans[i]);
60      cout << endl << a[6] << endl;
61    }
62  }
```

14.? 模拟退火

温度:①初始温度T_0;②终止温度T_E;③衰减系数,一般取$(0,1)$间的数,越接近1温度衰减越慢,找到最优解的概率越大,如$T_E = T_0 \times 0.999$.

以求函数$f(x)$的全局最小值为例.每次在当前随机区域内随机选一个点,计算$\Delta E = f(\text{新点}) - f(\text{当前点})$.①若$\Delta E < 0$,则跳到新点;②若$\Delta E > 0$,以一定概率跳到新点(否则可能收敛到局部最小值),使得ΔE越大,跳的概率越小,概率的经验值取$\mathrm{e}^{-\frac{\Delta E}{T}}$,条件判断可用$exp(-delta/T) > rand(0,1)$,则$\Delta E < 0$时,$\mathrm{e}^{-\frac{\Delta E}{T}} > 1$,一定跳转;$\Delta E > 0$时,$0 < \mathrm{e}^{-\frac{\Delta E}{T}} < 1$,以一定概率跳转,且$\Delta E$越小跳转的概率越大.

随着温度不断降低,随机区域不断缩小,跳出随机区域的概率也不断减小,温度降低到终止温度时稳定于的点可能是全局最小值.迭代若干次以降低收敛到局部最优解的概率.

14.3.1 星星还是树

例题

给定二维平面上的n $(1 \le n \le 100)$个点(x_i, y_i) $(0 \le x_i, y_i \le 1\mathrm{e}4)$,求一点使得它到这$n$个点的距离和最小,输出最小距离,四舍五入取整.

代码

```cpp
#define x first
#define y second

const int MAXN = 105;
int n;  // 点数
pdd points[MAXN];
double ans = INF;

double rand(double l, double r) { return (double)rand() / RAND_MAX * (r - l) + l; }  // 返回[l,r)内的随机数

double get_dis(const pdd& A, const pdd& B) { return hypot(A.x - B.x, A.y - B.y); }

double cal(const pdd& p) {  // 计算当前点的距离和
  double res = 0;
  for(int i = 0; i < n; i++) res += get_dis(p, points[i]);
  ans = min(ans, res);
  return res;
}

void simulated_annealing() {  // 模拟退火
  pdd cur(rand(0, 1e4), rand(0, 1e4));  // 随机一个点
  for (double T = 1e4; T > 1e-4; T *= 0.9) {  // 初始温度、终止温度、衰减系数
    pdd newp(rand(cur.x - T, cur.x + T), rand(cur.y - T, cur.y + T));  // 以温度为半径随机一个点
    double delta = cal(newp) - cal(cur);  // ΔE
    if (exp(-delta / T) > rand(0, 1)) cur = newp;
  }
}

int main() {
  cin >> n;
  for (int i = 0; i < n; i++) cin >> points[i].x >> points[i].y;

  for (int i = 0; i < 100; i++) simulated_annealing();  // 做100次模拟退火
  cout << (int)(ans + 0.5);
}
```