Part VI 搜索相关(DFS & BFS)

一、八皇后

> 会下国际象棋的人都很清楚：皇后可以在横、竖、斜线上不限步数地吃掉其他棋子。如何将8个皇后放在棋盘上（有8 * 8个方格），使它们谁也不能被吃掉！这就是著名的八皇后问题。 对于某个满足要求的8皇后的摆放方法，定义一个皇后串a与之对应，即a=b1b2...b8，其中bi为相应摆法中第i行皇后所处的列数。已经知道8皇后问题一共有92组解（即92个不同的皇后串）。 给出一个数b，要求输出第b个串。串的比较是这样的：皇后串x置于皇后串y之前，当且仅当将x视为整数时比y小。

输入描述:

> 每组测试数据占1行，包括一个正整数$b(1 \leq b \leq 92)$

输出描述:

> 输出有n行，每行输出对应一个输入。输出应是一个正整数，是对应于b的皇后串。

示例1 输入

> 1

输出

> 15863724

完整代码实现：

```
#include <stdio.h>
#include <stdlib.h>
#define SOLUTION 92
#define COL_ROW_NUM 8
int count;                                      // 记录结果数
int sol[SOLUTION][COL_ROW_NUM];                 // 存放结果
int maze[COL_ROW_NUM][COL_ROW_NUM];             // 标记位置
int position[COL_ROW_NUM];                      // 标记行

//判断当前摆放的皇后与前r-1行是否有冲突
int whetherConflict(int r, int c) {
    int i, j;
    for (i = 0;i < r;i++) {
        for (j = 0;j < COL_ROW_NUM;j++) {
            if (maze[i][j]) {
```

```
                if (c == j || r - i == abs(c - j)) {
                    return 0;                   //同一列或者位于同一对角线,不满足
                }
            }
        }
    }
    return 1;
}

void dfs(int step) {
    int i, c;
    if (step == COL_ROW_NUM) {
        for(i = 0; i < 8; i++) {
            sol[count][i] = position[i];
        }
        count++;
        return;
    }
    for (c = 0;c < COL_ROW_NUM;c++) {
        if (!maze[step][c]) {
            if (whetherConflict(step, c)) {
                position[step] = c + 1;
                maze[step][c] = 1;
                dfs(step + 1);
                maze[step][c] = 0;
            }
        }
    }
}

int main() {
    int i, n;
    dfs(0);
    while (scanf("%d", &n) != EOF) {
        for(i = 0; i < 8; i++){
            printf("%d", sol[n - 1][i]);
        }
        printf("\n");
    }
    return 0;
}
```

二、放苹果

题目描述 把M个同样的苹果放在N个同样的盘子里，允许有的盘子空着不放，问共有多少种不同的分法？（用K表示）5，1，1和1，5，1 是同一种分法。 **输入描述：** 每行均包含两个整数M和N，以空格分开。$1 \leq M , N \leq 10$。 **输出描述：** 对输入的每组数据M和N，用一行输出相应的K。 **示例1 输入** 7 3 **输出** 8

题解： **方法一：** 该问题可以抽象成整数划分的一种：求将一个整数m至多划分成n个数有多少种情况(**这里默认n小于等于m，当n大于m时，该类情况与将一个整数m至多划分成m个数一致**)，因此原问题是求正整数m的不同划分个数，划分个数范围为[1,n]，因此可以记f(m, n)为正整数m划分成n个正整数的划分个数，sum为总的划分个数，则总的划分个数为：

$$sum = f(m,1) + f(m,2) + \cdots + f(m,n)$$

故原问题转化为：**求解正整数m划分成n个正整数的划分个数。** 对于问题正整数m划分成n个正整数的划分个数，同样的，又可以将问题转化为将正整数m - n划分成1，...，m - n个正整数的划分个数。而当划分数为1时，可以特殊判断。因此，就又回到了原来的问题，但是问题规模却变小了，因此，综上所述，该整数划分问题可以利用递归的方法求解。递归的终止条件也就是当划分数为1时，返回1(为了减少递归次数，当划分数与整数相等时，也返回1)，但是需要注意的是：**当将原问题转化成规模更小的子问题时，例如上述描述的情况，得到的子问题将正整数m - n划分成1，...，m - n个正整数的划分个数，但是划分数m – n必须≤原问题的划分数n**

完整代码实现：

```
#include <stdio.h>
#define MIN(a, b) (a < b ? a : b)

int divide(int m, int n, int limit) {
    int i, sum = 0;
    //递归终止条件
    if (n == 1 || m == n) {
        return 1;
    }

    for (i = 1; i <= MIN(m - n, limit); i++) {
        sum += divide(m - n, i, i);
    }
    return sum;
}

int main() {
    int i, m, n, ans;
    while(scanf("%d %d", &m, &n) != EOF) {
        ans = 0;
        for (i = 1; i <= MIN(m, n); i++) {
            ans += divide(m, i, i);
        }
        printf("%d\n", ans);
    }
    return 0;
}
```

方法二： 稍加思考，对上述思路进行改进，有以下分析： 记g(m, n)为将m个苹果放入n个盘子的分法总数，因此可以做以下讨论：

- 当m < n时，此时盘子比较多，肯定有空盘子，因此去掉必空的盘子，即g(m, n) = g(m, m)
- 当m ≥ n时，此时苹果比较多，根据是否含有空盘子，可以分两种情况讨论：
    - 若至少有一个空盘子，则拿掉这个空盘子，即g(m, n) = g(m, n - 1)

    - 若没有空盘子，则每个盘子拿掉一个苹果，即g(m, n) = g(m - n, n)
- 当m = 0时，表示盘子中苹果全部拿掉了，此时说明只有这一种方法(也就相当于m个苹果刚好放入m个盘子中)，当n = 1时，只有一个空盘子，此时也只有一种方法。

因此，有了以上的分析，可以得到如下完整代码：

```
#include <stdio.h>

int divide(int m, int n) {
    if (m == 0 || n == 1) {
        return 1;
    } else if (n > m) {
        return divide(m, m);
    } else {
        return divide(m, n - 1) + divide(m - n, n);
    }
}
int main() {
    int m, n;
    while (scanf("%d %d", &m, &n) != EOF) {
        printf("%d\n", divide(m, n));
    }
    return 0;
}
```

Part VII 贪心算法

一、找零钱

前段时间，小x买了个钱包，结果买完就没钱放了，一气之下将钱包搁置箱底，常常忘记带出来。但是没有钱包的话，纸币放在口袋里很不方便，容易乱，也容易掉，所以每次有买什么东西的时候，他都会让收银员找给他最少张数的纸币。收银员忙于找零，经常没法顾及这个问题，所以求助会编程的你。正如我们所知，纸币面值一般有1元，5元，10元，20元，50元，100元。

输入描述：

> 输入包含多组数据。 输入第一行包含一个整数N，表示要找的零钱总额。

输出描述：

> 每次输出一个整数表示答案

示例1 输入

> 75 13

输出

```
3 4
```

题解： 找零钱在实际生活中是非常常见的问题，一般情况下，为了使张数最少，只需要尽量用面额大的钱即可。 具体如下：需要找x的零钱，每次取出小于x且面额最大的零钱，然后x赋值为剩下需要找的零钱。然后重复操作，直至全部找完。

完整代码实现如下：

```
#include <stdio.h>
int m[6] = {100, 50, 20, 10, 5, 1};
int main() {
    int i, n;
    int ans;
    while(scanf("%d",&n) != EOF) {
        ans = 0;
        for(i = 0;i < 6;i++) {
            if(n && m[i] <= n) {
                ans += n / m[i];
                n = n % m[i];
            }
        }
        printf("%d\n",ans);
    }
    return 0;
}
```

二、区间调度问题

作为新时代的好青年，你一定还会看一些其它的节目，比如新闻联播（永远不要忘记关心国家大事）、非常6+7、超级女生，以及王小丫的《开心辞典》等等，假设你已经知道了所有你喜欢看的电视节目的转播时间表，你会合理安排吗？（目标是能看尽量多的完整节目）

输入描述：

输入数据包含多个测试实例，每个测试实例的第一行只有一个整数$n(n \leq 100)$，表示你喜欢看的节目的总数，然后是n行数据，每行包括两个数据$Ti_s, Ti_e (1 \leq i \leq n)$，分别表示第i个节目的开始和结束时间，为了简化问题，每个时间都用一个正整数表示。n=0表示输入结束，不做处理。

输出描述：

对于每个测试实例，输出能完整看到的电视节目的个数，每个测试实例的输出占一行。

示例1 输入

```
12 1 3 3 4 0 7 3 8 15 19 15 20 10 15 8 18 6 12 5 10 4 14 2 9 0
```

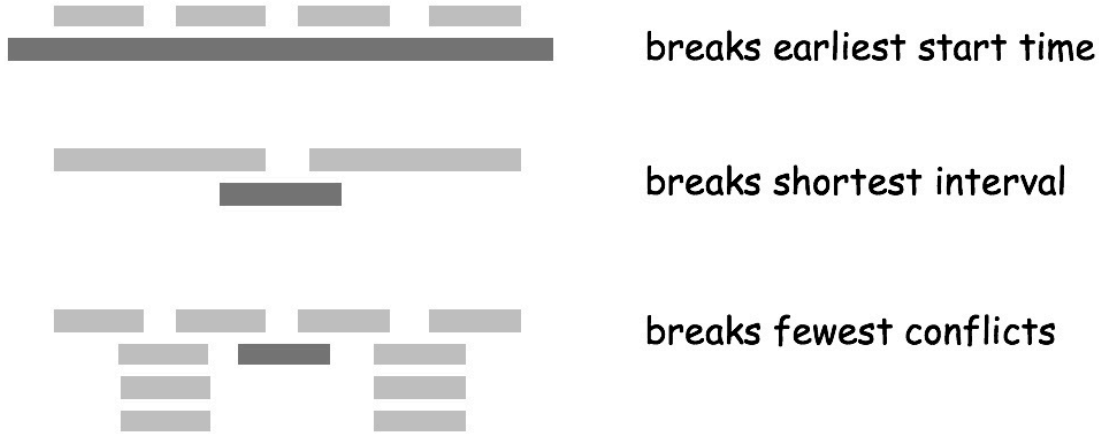
输出

```
5
```

题解： 一些可能的解决策略(策略编号为1，2，3，4)：

- [Earliest start time] Consider jobs in ascending order of start time s_j.
- [Earliest finish time] Consider jobs in ascending order of finish time f_j.
- [Shortest interval] Consider jobs in ascending order of interval length $f_j - s_j$.
- [Fewest conflicts] For each job, count the number of conflicting jobs c_j. Schedule in ascending order of conflicts c_j.

对应于策略1，3，4的反例：



策略2(将区间按照结束时间升序排序)的实现：

Greedy algorithm. Consider jobs in increasing order of finish time. Take each job provided it's compatible with the ones already taken.

```
Sort jobs by finish times so that f_1 ≤ f_2 ≤ ... ≤ f_n.

                jobs selected
A ← φ
for j = 1 to n {
   if (job j compatible with A)
      A ← A ∪ {j}
}
return A
```

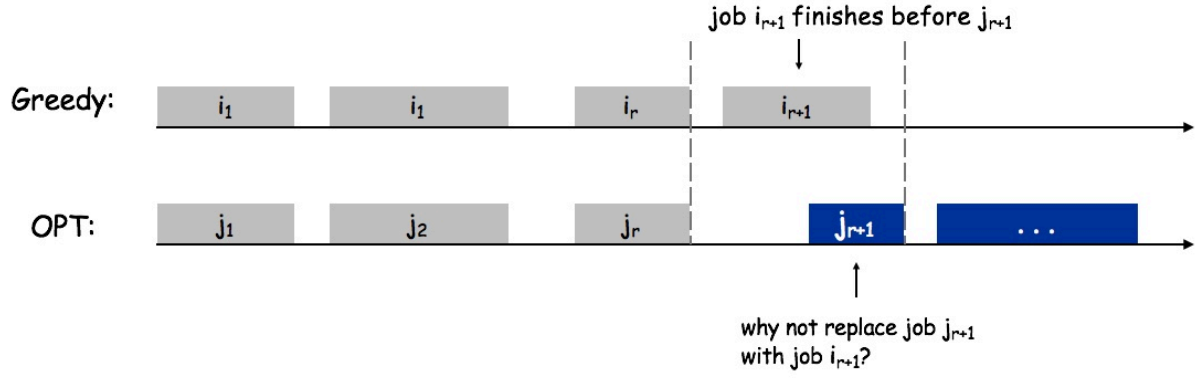
Implementation. O(n log n).

- Remember job j* that was added last to A.
- Job j is compatible with A if $s_j \geq f_{j^*}$.

策略2(将区间按照结束时间升序排序)的正确性证明：

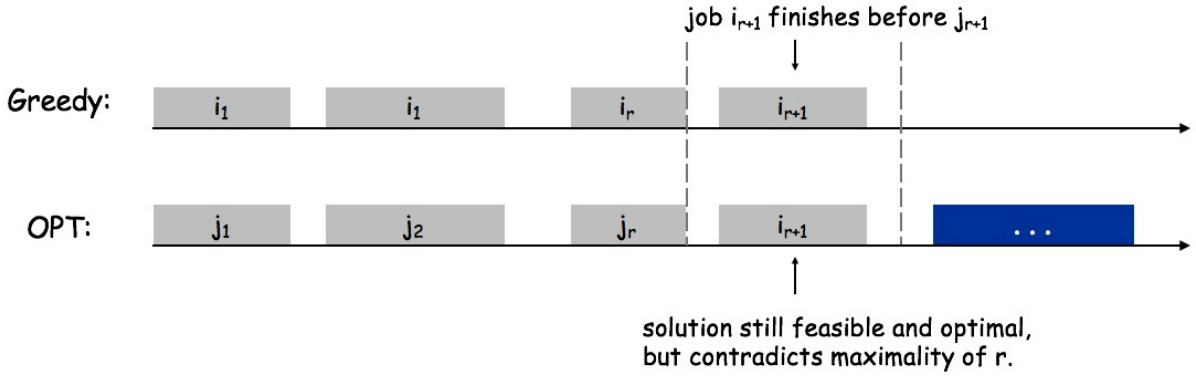
Pf. (by contradiction)

- Assume greedy is not optimal, and let's see what happens.
- Let $i_1, i_2, \ldots i_k$ denote set of jobs selected by greedy.
- Let $j_1, j_2, \ldots j_m$ denote set of jobs in the optimal solution with $i_1 = j_1, i_2 = j_2, \ldots, i_r = j_r$ for the largest possible value of r.



Pf. (by contradiction)

- Assume greedy is not optimal, and let's see what happens.
- Let $i_1, i_2, \ldots i_k$ denote set of jobs selected by greedy.
- Let $j_1, j_2, \ldots j_m$ denote set of jobs in the optimal solution with $i_1 = j_1, i_2 = j_2, \ldots, i_r = j_r$ for the largest possible value of r.



上图中，Greedy是运用贪心算法得出来的解。假设Greedy不是最优解，并且OPT是与Greedy前r个选择一致的最优解，即上述中的$i_1 = j_1, i_2 = j_2, \cdots, i_r = j_r$，考虑贪心算法和最优解选择的第r+1个区间$i_{r+1}$和$j_{r+1}$，由于贪心策略的选择，故区间$i_{r+1}$的结束时间要早于区间$j_{r+1}$，而显然此时选择区间$i_{r+1}$是更优的结果(越早结束，留给后续区间的“空间”就越大)，故此时最优解应该选择区间i_{r+1}，这样的话OPT与Greedy前r+1个区间是一致的。而这与假设相悖(假设中是r个区间一致)，故假设矛盾。因此得出结论：该贪心算法得出的是最优解。

完整代码实现：

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

struct iterm {
    int s;
    int e;
} a[110];
bool cmp(iterm x, iterm y) {
    return x.e < y.e;
}
int main() {
    int n;
    while(scanf("%d", &n) != EOF && n) {
        memset(a, 0, sizeof(a));
        for(int i = 1; i <= n; i++) {
            scanf("%d %d", &a[i].s, &a[i].e);
        }
        sort(a + 1, a + n + 1, cmp);            // 按照结束时间升序排序
        int cnt = 0, t = 0;
        for(int i = 1; i <= n; i ++) {
            if(t <= a[i].s) {
                cnt++;
```

```
                t = a[i].e;
            }
        }
        printf("%d\n", cnt);
    }
    return 0;
}
```

Part VIII 字符串

一、字符串统计

编写一个函数，该函数可以统计一个长度为2的字符串在另一个字符串中出现的次数。 完整代码实现：

```
#include <stdio.h>
#include <string.h>
int main() {
    char a[] = "as", b[] = "asaassdfdasdfdasdfd";
    puts(a);
    puts(b);
    printf("%d\n", fun(a, b));
    return 0;
}
int fun(char *a,char *b) {
   int i, j = 0;
   int len = strlen(b);
   for(i = 0;i < len - 1;i++) {
       if(a[0] == b[i] && a[1] == b[i+1]) {
           j++;
       }
   }
   return j;
}
```

二、子串计算

> 给出一个01字符串（长度不超过100），求其每一个子串出现的次数。

输入描述:

> 输入包含多行，每行一个字符串。

输出描述:

> 对每个字符串，输出它所有出现次数在1次以上的子串和这个子串出现的次数，输出按字典序排序。

示例1 输入

```
10101
```

输出

```
0 2 01 2 1 3 10 2 101 2
```

题解： 利用C++ STL模板库中的map即可，完整代码实现如下：

```
#include<iostream>
#include<string>
#include<map>
using namespace std;

int main() {
    string s;
    while (cin >> s) {
        map<string, int> m;
        for (int i = 1;i <= s.size();i++)
            for (int j = 0;j < i;j++)
                m[s.substr(j, i - j)]++;

        for (auto it = m.begin(); it != m.end(); it++) {
            if (it -> second > 1)
                cout << it -> first << ' ' << it -> second << endl;
        }
    }
    return 0;
}
```

三、玛雅人的密码

玛雅人有一种密码，如果字符串中出现连续的2012四个数字就能解开密码。给一个长度为N的字符串，（$2 \leq N \leq 13$）该字符串中只含有0,1,2三种数字，问这个字符串要移位几次才能解开密码，每次只能移动相邻的两个数字。例如02120经过一次移位，可以得到20120,01220,02210,02102，其中20120符合要求，因此输出为1.如果无论移位多少次都解不开密码，输出-1。

输入描述：

输入包含多组测试数据，每组测试数据由两行组成。 第一行为一个整数N，代表字符串的长度（$2 \leq N \leq 13$）。 第二行为一个仅由0、1、2组成的，长度为N的字符串。

输出描述：

对于每组测试数据，若可以解出密码，输出最少的移位次数；否则输出-1。

示例1 输入

```
5 02120
```

输出

```
1
```

题解： 典型的BFS，节点就是一个字符串，与它相邻的节点就是由它交换一次所得的字符串。 从初始节点出发，看看它交换一次形成的字符串里哪些没出现过，这些字符串如果有符合要求的就结束BFS返回结果，如果不符合要求那就把它们加入队列。遍历完之后还是没碰到符合要求的字符串就返回-1。

完整代码实现如下：

```
#include <stdio.h>
#include <iostream>
#include <map>
#include <string>
#include <queue>
using namespace std;
map<string, int> M;
queue<string> Q;

//将字符串i位与i+1位交换
string swapChar(string str, int i) {
    string newStr = str;
    char tmp = newStr[i];
    newStr[i] = newStr[i + 1];
    newStr[i + 1] = tmp;
    return newStr;
}

//判断字符串中是否含有"2012"
bool judge(string str) {
    if(str.find("2012", 0) == string::npos) {
        return false;
    }
    else {
        return true;
    }
}

int BFS(string str) {
    string newStr;
    M.clear();
    //清空队列
    while(!Q.empty()) {
        Q.pop();
    }
    Q.push(str);            //初始字符串作为起点放入队列
    M[str] = 0;             //初始字符串经历的交换次数是0
    while(!Q.empty()) {
        str = Q.front();
```

```
        Q.pop();            //取出队首，存入str
        for(int i = 0; i < str.size() - 1; i++){
            newStr = swapChar(str, i);
            //如果这个字符串没出现过
            if(M.find(newStr) == M.end()) {
                M[newStr] = M[str] + 1;           //现在出现过了，且交换次数比其父节点多1
                if(judge(newStr)) {
                    return M[newStr];             //符合要求返回结果
                } else {                          //不合要求，将其加入队列
                    Q.push(newStr);
                }
            } else {                //出现过的字符串，不用进行处理
                continue;
            }
        }
    }
    return -1;              //遍历完成，没发现符合要求的字符串，返回-1
}

int main() {
    int n;
    string str;
    while(cin >> n) {
        cin >> str;
        if(judge(str)) {
            cout << 0 << endl;
        } else {
            cout << BFS(str) << endl;
        }
    }
    return 0;
}
```