Table of Contents

[〇、说明 2](#_Toc155759663)

[一、算法概述 2](#_Toc155759664)

[1、常见算法 2](#_Toc155759665)

[欧几里得算法=辗转相除法：求GCD 2](#_Toc155759666)

[货郎问题=旅行商问题=Traveling Salesman Problem=TSP 3](#_Toc155759667)

[2、P、NP=Nondeterministic Polynomial、NP-hard=NP困难 3](#_Toc155759668)

[3、时间复杂度 3](#_Toc155759669)

[4、算法的伪码表示 4](#_Toc155759670)

[5、函数的“渐近的界” 5](#_Toc155759671)

[六、贪心 5](#_Toc155759672)

[1、简单的贪心问题：采购礼物、硬币找零、部分背包问题（用结构体）、数列分段（跟区间覆盖很像）；股票交易、排队接水（Short Job First时的等待时间之和最少）、仓库选址、均分图书、合并果子（Huffman树、priority\_queue） 6](#_Toc155759673)

[2、较简单的贪心问题：区间选点（用结构体）、最优活动安排（仅仅需要把区间选点问题的代码中的<=be改为<be即可） 6](#_Toc155759674)

[代码： 6](#_Toc155759675)

[3、区间覆盖: 按左端点从小到大排序 8](#_Toc155759676)

[代码1： 8](#_Toc155759677)

[代码2： 10](#_Toc155759678)

[七、回溯 11](#_Toc155759679)

[1、枚举子集2 12](#_Toc155759680)

[代码1： 13](#_Toc155759681)

[代码2： 14](#_Toc155759682)

[2、全排列 15](#_Toc155759683)

[代码： 16](#_Toc155759684)

[3、n-皇后问题（可行解问题） 17](#_Toc155759685)

[代码1： 17](#_Toc155759686)

[代码2： 19](#_Toc155759687)

[代码3： 20](#_Toc155759688)

[4、0-1背包（最优解问题） 21](#_Toc155759689)

[代码1： 21](#_Toc155759690)

[代码2： 23](#_Toc155759691)

[5、力扣17：电话号码的字母组合 24](#_Toc155759692)

[八、分支限界 25](#_Toc155759693)

[1、BFS=Breadth-First Searc=宽度优先搜索=广度优先搜索 25](#_Toc155759694)

[2、分支限界 26](#_Toc155759695)

[3、例题1：走迷宫 26](#_Toc155759696)

[代码1： 27](#_Toc155759697)

[代码2： 29](#_Toc155759698)

[4、例题2：0-1背包：难 30](#_Toc155759699)

[5、例题3：任务分配：难 31](#_Toc155759700)

[五、分治Divide and Conquer 31](#_Toc155759701)

[1、简单的： 32](#_Toc155759702)

[归并排序Merge Sort（递归地排序两个子数组、合并两个有序数组）、逆序对的数量（仅仅需要在归并排序中加一个变量cnt用于计数即可，O(nlogn)时间）、第k小的数（仅仅需要在快速排序中加一个对递归的剪枝，使得只进行左半部分的递归或右半部分的递归即可）；模幂运算（快速幂）（利用a^(2b)=(a^2)^b和模运算的简单性质，O(log b)时间）、最大连续子数组和（2个区间合并后的最大子序和=max（左子区间的最大子序和，右子区间的最大子序和，横跨两个子区间的最大子序和），O(nlogn)时间）；有序数组中查找数1、有序数组中查找数2、数的三次方根（ 32](#_Toc155759703)

[2、快速排序Quick Sort 32](#_Toc155759704)

[代码1：自己错误的改法： 33](#_Toc155759705)

[代码2：老师版的，但不能让pivot在中间： 34](#_Toc155759706)

[代码3：可以让pivot到自己最终的位置上： 35](#_Toc155759707)

# 〇、说明

注：

“简单”：指自己可以独立做出来。自己写的代码全部传到GitHub上了

“较简单”：指虽然自己可以独立做出来，但有小bug，bug用红色全部标出

Github地址：

<https://github.com/WangSoooCute/ruanwei_courses>

# 一、算法概述

## 1、常见算法

### 欧几里得算法=辗转相除法：求GCD

//时间复杂度：最坏情况下为 **O(log(min(a,b)))**。因为在每次迭代中，数字都会大致减少一半。

//空间复杂度：O(1)

long long gcd(long long a, long long b){

if (b == 0)

return a;

else

return gcd(b, a%b);

}

### 货郎问题=旅行商问题=Traveling Salesman Problem=TSP

是NP-hard问题，至今没有有效的算法，为了得到这个解，可以采用多种策略，包括：

暴力法：考虑所有可能的路线并选择最短的，但需要 O(n!)时间

启发式方法：如最近邻法、最小生成树法、Christofides算法等。

元启发式方法：如模拟退火、遗传算法、蚁群优化等。

精确方法：如分支和定界、整数线性规划等。

## 2、P、NP=Nondeterministic Polynomial、NP-hard=NP困难

P类问题：可在多项式时间内由确定性图灵机解决的问题，是被普遍认为“容易”的问题

NP类问题：可在多项式时间内由非确定性图灵机猜测一个解，并由确定性图灵机在多项式时间内验证这个解是否正确的问题。这意味着我们可以在合理的时间内验证一个给定的解，但是找到这个解可能会非常困难。

NP-hard的定义： 一个决策问题A是NP-hard的，如果所有在NP中的问题都可以在多项式时间内归约到问题A。换句话说，如果你能为某个NP-hard问题找到一个多项式时间的解决方案，那么你也可以为NP中的所有问题找到一个多项式时间的解决方案。

关于NP-hard的重要点：

1、至今未找到有效算法：至今未找到能在多项式时间内解决NP-hard问题的算法。现有的算法的运行时间往往是输入规模的指数或更高阶函数，这意味着随着输入规模的增加，所需的计算时间将急剧增加。

2、未知的边界：虽然NP-hard问题是非常难的，但到目前为止，还没有人能够证明对于这类问题不存在多项式时间的算法。这是计算理论中最大的未解决问题之一，也就是P是否等于NP的问题。

3、问题之间的等价性：从是否存在多项式时间算法的角度看，NP-hard问题彼此是等价的。这意味着如果你能为一个NP-hard问题找到一个多项式时间的算法，那么其他所有的NP-hard问题也可以在多项式时间内解决。

## 3、时间复杂度

从小到大列举：

O(1)：常量时间复杂度。

O(logn)：对数时间复杂度。

O(n)：线性时间复杂度。

O(nlogn)：线性对数时间复杂度。

O(n^2)：多项式时间复杂度。

O(2^n)：指数时间复杂度。

O(n!)：阶乘时间复杂度

易错题：

以下代码的时间复杂度为O(logn)：

A screenshot of a test

Description automatically generated

以下代码的时间复杂度为O(2^n)：

Screens screenshot of a test

Description automatically generated

## 4、算法的伪码表示

1、赋值语句：variable ← value

2、分支语句：

if condition then

// 执行的操作

else

// 执行其他操作

3、循环语句：

while condition do

// 循环体

for variable in range do

// 循环体

repeat

// 循环体

until condition

4、转向语句：虽然goto可用于无条件地跳到指定位置，但它通常不被推荐使用

5、输出语句：用return

6、调用：直接使用函数或过程的名称进行调用。

7、注释：用//

## 5、函数的“渐近的界”

1、大O符号 (O)：

如果存在某个常数 c>0和某个值x0使得对于所有x>x0，|f(x)|≤c⋅|g(x)|，那么我们说 "f(x) 是 g(x) 的一个大O界"，或简单地说 "f(x) 是 O(g(x))"。

这实际上意味着，对于大的x值，f(x)的增长速度不会比g(x)快。

2、小o符号 (o)：

如果对于任何常数 c>0都存在某个值x0使得对于所有x>x0，|f(x)|<c⋅|g(x)|，那么说 "f(x) 是 g(x) 的一个小o界"，或简单地说 "f(x) 是 o(g(x))"。

这意味着f(x)的增长速度总是慢于g(x)，无论你选择了多大的常数c。

3、大Ω符号 (Ω)：

如果存在某个常数 c>0和某个值x0使得对于所有x>x0，|f(x)|≥c⋅|g(x)|，那么我们说 "f(x) 是 g(x) 的一个大Ω界"，或简单地说 "f(x) 是 Ω(g(x))"。

这意味着f(x)的增长速度至少与g(x)一样快。

4、大Θ符号 (Θ)：

如果f(x)是O(g(x))并且f(x)也是 Ω(g(x))，那么我们说 "f(x) 是 Θ(g(x))"。

这意味着f(x)的增长速度与g(x)相当。

# 六、贪心

基本思想：

贪心的本质是通过**选择局部最优实现全局最优**：每一步都采取当前最好的选择，从而希望导致结果是全局最优

每次只着眼于当前的最优决策，取**当前看上去最好的**

另一个常用思路是**为后续留下更好的局面**

往往是将元素按照一定顺序**先排序**后一个一个取：用**sort**函数！

贪心算法有时候就是**常识性**的推导，有时候自己都不知道在用贪心策略

**没有任何套路**可以直接看出来题目是否可以使用贪心算法

最好的办法是**反例法，来推翻贪心策略不可行**

严格的数学证明方法：**数学归纳法和反证法**

常见贪心问题：

活动选择问题

分数背包问题

项目投资问题

最小生成树问题

单源最短路径问题

压缩编码问题

任务调度问题

## 1、简单的贪心问题：

### 采购礼物

### 硬币找零

### 部分背包问题：用结构体

### 区间选点：用结构体，按区间的左端点排序

### 最优活动安排：仅仅需要把区间选点问题的代码中的<=be改为<be即可

### 数列分段：跟区间覆盖很像；

### 股票交易

### 排队接水：Short Job First时的等待时间之和最少

### 仓库选址

### 均分图书

### 合并果子：Huffman树、priority\_queue

## 2、区间覆盖: 按**左**端点从小到大排序

Description：

给定 N 个闭区间 [a\_i, b\_i] 以及一个指定的线段区间 [s, t] ，需要选择尽可能少的区间来完全覆盖指定的线段区间。

Input：

第一行包含两个整数 ( s ) 和 ( t )，表示给定线段区间的两个端点。

第二行包含整数 ( N )，表示给定的区间数量。

接下来的 ( N ) 行，每行包含两个整数 ( a\_i, b\_i )，表示一个区间的两个端点。

数据范围：

( 1 <= N <= 10^5 )

( -10^9 <= a\_i <= b\_i <= 10^9 )

( -10^9 <= s <= t <= 10^9 )

Output：

输出一个整数，表示所需的最少区间数。

如果无法完全覆盖给定的线段区间，则输出 (-1)。

Sample Input：

1 5

3

-1 3

2 4

3 5

Sample Output：

2

### 代码1：

#include <iostream>

#include <algorithm>

using namespace std;

struct node{

int begin, end;

};

bool cmp(node a, node b){

return a.begin < b.begin;

}

int main() {

int start, t;

cin >> start >> t;

int n;

cin >> n;

node a[100010];

for(int i = 0; i < n; i++)

cin >> a[i].begin >> a[i].end;

sort(a, a+n,cmp);

int cnt = 0;

for(int i = 0; i < n; i++){

int rmax = -2e9;

//寻找能够包含start点的区间中，最大的右端点的值

for(int j = i;j < n && a[j].begin <= start;j++){

rmax = max(rmax, a[j].end);

}

if(rmax < start){

cout << -1 << endl;

return 0;

}

else{

start = rmax; //更新起始端点，准备下一轮寻找

cnt++;

if(start >= t){ //如果已经满足题目要求

cout << cnt << endl;

return 0;

}

}

}

cout << -1 << endl;

return 0;

}

### 代码2：

//自己写的，应该是对的

#include <iostream>

#include <algorithm>

using namespace std;

struct node{

int begin, end;

};

bool cmp(node a, node b){

return a.begin < b.begin;//按左端点从小到大排序

}

int main(){

int start, t;

cin >> start >> t;

int n;

cin >> n;

node a[100010];

for(int i = 0; i < n; i++)

cin >> a[i].begin >> a[i].end;

sort(a, a+n,cmp);

int cnt = 0;

int i = 0;

while(1){

int rmax = -2e9;

//寻找能够包含start点的区间中，最大的右端点的值

while(i < n && a[i].begin <= start){

rmax = max(rmax, a[i].end);

i++;

}

if(rmax < start){//包含情况：while一次都没运行

cout << -1 << endl;

return 0;

}

else{

start = rmax; //更新起始端点，准备下一轮寻找

cnt++;

//因为可能程序一开始就start==t，且没有能覆盖这个点的区间

//所以下面这段不能写在前面

if(start >= t){ //如果已经满足题目要求

cout << cnt << endl;

return 0;

}

}

}

return 0;

}

# 七、回溯

基本步骤：

从选择列表中选择：从现有的选项中选择一个，并尝试解决问题。

**递归**：基于这个选择，继续向前走一步，进入下一层递归（会形成一棵**决策树**来表示求解过程）。

回溯：如果发现当前选择并不是一个有效的解决方案，或者已经达到问题的边界条件，则撤销这个选择（回溯），返回上一步，尝试别的选择。

void dfs(int step){

if (step > n){

处理

return;

}

//两种

//第1种：多种选择用循环（如排列数字、n-皇后）：

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

if (isUsed[i])

continue; // 判断是否使用过

isUsed[i] = true; // 标记使用

b[step] = i; // 记录下该值的下标

dfs(step + 1); //继续搜索

isUsed[i] = false; // 回溯

}

//第2种：只有两种选择就用左右子树（如枚举子集2、01背包）：

左子树；

右子树；

}

典型问题：

组合问题、子集问题、排列问题；

解数独问题、最优加工顺序问题、走迷宫问题、八数码问题；

路径和图搜索问题：

N皇后问题：在NxN的国际象棋上摆放N个皇后，使其不能相互攻击。

单词搜索：在给定的字符网格中搜索特定的单词。

划分问题：

分割回文串：给定一个字符串，将其划分为若干子串，使每个子串都是回文的，并返回所有这些可能的划分。

0-1背包问题：给定物品的重量和价值以及一个固定容量的背包，选择物品的子集，使得背包中物品的总价值最大化且不超过背包的容量。

优化问题：

货郎问题 (TSP)：找到访问所有给定的城市并返回到起始城市的最短路径。

魔术索引：在数组A[0...n-1]中找到满足条件A[i] = i的魔术索引。

## 1、枚举子集2

Description：

写一个递归函数，枚举n个数{x\_1 , ... , x\_n}中取若干个数的每种方法（子集枚举）. 对每种方法把取出的数从小到大排序，若两种方法中前k-1个数的选取情况相同，则取了第k个数的在前. 例如对n=4,X={100,20,3,4},应该按如下顺序给出结果：

100,20,3,4

100,20,3

100,20,4

100,20

100,3,4

100,3

100,4

100

20,3,4

20,3

20,4

20

3,4

3

4

Input：

第一行一个正整数n

第二行n个正整数

Output：

输出2^n行, 每行一个子集, 按题目要求顺序.

Sample Input：

4

100 20 3 4

Sample Output：

100 20 3 4

100 20 3

100 20 4

100 20

100 3 4

100 3

100 4

100

20 3 4

20 3

20 4

20

3 4

3

4

### 代码1：

#include <iostream>

using namespace std;

int n;

int a[30];

int vis[30];

void print(){

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

if(vis[i] == 1)

cout << a[i] << " ";

}

cout << endl;

}

void dfs(int step){

if(step > n) {

print();

return;

}

//遍历（选与不选），左子树，右子树

vis[step] = 1;

dfs(step + 1);

vis[step] = 0;

dfs(step + 1);

}

int main(){

cin >> n;

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

cin >> a[i];

dfs(1);

return 0;

}

### 代码2：

#include <iostream>

#include <vector>

#include <algorithm>

using namespace std;

void backtrack(vector<int>& nums, int pos, vector<int>& subset) {

// 输出当前子集

if(pos==nums.size()){

for (int i = 0; i < subset.size(); i++) {

cout << subset[i] << " ";

}

cout << endl;

return ;

}

// 递归回溯

subset.push\_back(nums[pos]);

backtrack(nums, pos + 1, subset);

subset.pop\_back();

backtrack(nums, pos + 1, subset);

}

void enumerateSubsets(vector<int>& nums) {

//sort(nums.begin(), nums.end()); // 对输入的数进行排序

vector<int> subset;

backtrack(nums, 0, subset);

}

int main() {

int n;

cin >> n;

**vector<int> nums(n);**

for (int i = 0; i < n; i++) {

cin >> nums[i];

}

enumerateSubsets(nums);

return 0;

}

## 2、全排列

Description：

编写一个程序，用于枚举 n 个数 x1, x2, ..., xn 的所有排列方式（即全排列枚举）。如果两种排列方式的前 k-1 个数相同，则将第 k 个数序号更小的排列放在前面。例如，当 n=3 且 x={100, 20, 5} 时，应该按以下顺序输出结果：{100, 20, 5}, {100, 5, 20}, {20, 100, 5}, {20, 5, 100}, {5, 100, 20}, {5, 20, 100}。

Input：

第一行是一个正整数 n。第二行是 n 个正整数，x1, x2, ..., xn。

Output：

输出 n! 行，每行显示一个排列，排列应按题目要求的顺序排列。

Sample Input：

3

100 20 5

Sample Output：

100 20 5

100 5 20

20 100 5

20 5 100

5 100 20

5 20 100

### 代码：

#include<iostream>

#include<cstdio>

#include<algorithm>

using namespace std;

int n;

bool isUsed[30]={0};

int b[30], x[30];

void print(){

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

cout << x[b[i]] << ' ';

cout << endl;

}

void dfs(int step){

if (step > n){

print();

return;

}

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

if (isUsed[i])

continue; // 判断是否使用过

isUsed[i] = true; // 标记使用

b[step] = i; // 记录下该值的下标

dfs(step + 1); //继续搜索

isUsed[i] = false; // 回溯

}

}

int main() {

cin >> n;

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

cin >> x[i];

dfs(1);

return 0;

}

## 3、n-皇后问题（可行解问题）

Description：

在一个 n×n 的国际象棋棋盘上，需要放置 n 个皇后，使得每一行、每一列以及每条对角线上最多只有一个皇后。这就构成了 n 皇后问题。例如，当 n=8 时，八皇后问题就是在 8×8 的棋盘上放置八个皇后，遵循上述规则。

你的任务是根据给定的 n，输出 n 皇后问题的前 k 个解。由于每一行必定只有一个皇后，如果两个解的前 k-1 行的皇后位置相同，那么在第 k 行中皇后位置更靠左的解应排在前面（即按字典序排序）。

Input：

输入只有一行，包含两个正整数 n 和 k，表示需要求解的是 n 皇后问题的前 k 个解。

Output：

输出前 k 个解：每行包含 n 个用空格隔开的正整数，依次表示每组解中第 1, 2, ..., n 行的皇后所在的列位置。

Sample Input：

8 3

Sample Output：

1 5 8 6 3 7 2 4

1 6 8 3 7 4 2 5

1 7 4 6 8 2 5 3

### 代码1：

#include<cstdio>

#include<iostream>

#include<algorithm>

using namespace std;

int n, k;

int cnt; //记录已经求到第几个答案

int a[20]; //记录每行皇后放的位置

bool flag1[20];

bool flag2[40];

bool flag3[40];

**// flag1[j]表示第j列有没有皇后**

**// flag2[i + j]表示行列和为i+j有没有皇后**

**// flag3[i - j + n]表示行列差为i-j+n有没有皇后**

// 排列树

void dfs(int step){ // step代表第几行

if(step > n) { // 找到一个可行解后，打印

if(cnt++ == k)

**exit(0);** //超过了k

// 打印一组可行解

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

cout << a[i] << " ";

cout << endl;

}

// 每行挨个位置尝试1~n

for(int y = 1 ; y <= n; y++){

//如果step行第y个位置冲突or......，则剪枝

if(flag1[y] || flag2[step + y] || flag3[step-y+n])

continue;

a[step] = y;

flag1[y] = true;

flag2[step + y] = true;

flag3[step-y+n] = true;

dfs(step + 1); // 继续下一行

flag1[y] = false;

flag2[step + y] = false;

flag3[step-y+n] = false;

}

}

int main(){

cin >> n >> k;

dfs(1);

return 0;

}

### 代码2：

#include <iostream>

#include <vector>

using namespace std;

int cnt=0;

bool isSafe(vector<int>& queenPos, int row, int col) {

// 检查当前位置是否与之前的皇后冲突

for (int i = 0; i < row; i++) {

// 在同一列或者在对角线上

if (queenPos[i] == col || queenPos[i] - col == i - row || queenPos[i] - col == row - i)

return false;

}

return true;

}

void solveNQueens(int n, int k, int row, vector<int>& queenPos) {

if (row == n) {

// 找到一个解

for (int i = 0; i < n; i++) {

cout << queenPos[i] + 1 << " ";

}

cout << endl;

if (++cnt == k){

exit(0);

}

return ;//注释掉这一句也能正常运行

}

for (int col = 0; col < n; col++) {

if (isSafe(queenPos, row, col)) {

// 在当前位置放置皇后

queenPos[row] = col;

solveNQueens(n, k, row + 1, queenPos);

}

}

}

int main() {

int n, k;

cin >> n >> k;

vector<int> queenPos(n, 0);

solveNQueens(n, k, 0, queenPos);

return 0;

}

### 代码3：

#include <iostream>

#include <vector>

using namespace std;

int cnt=0;

bool isSafe(vector<int>& queenPos, int row, int col) {

// 检查当前位置是否与之前的皇后冲突

for (int i = 0; i < row; i++) {

// 在同一列或者在对角线上

if (queenPos[i] == col || queenPos[i] - col == i - row || queenPos[i] - col == row - i)

return false;

}

return true;

}

void solveNQueens(int n, int k, int row, vector<int>& queenPos) {

if (row == n) {

// 找到一个解

for (int i = 0; i < n; i++) {

cout << queenPos[i] + 1 << " ";

}

cout << endl;

++cnt;

return ;

}

for (int col = 0; col < n; col++) {

if (isSafe(queenPos, row, col)) {

// 在当前位置放置皇后

queenPos[row] = col;

solveNQueens(n, k, row + 1, queenPos);

if (cnt == k){

//cout<<"row:"<<row<<endl;//观察这一行的输出

return ;

}

}

}

}

int main() {

int n, k;

cin >> n >> k;

vector<int> queenPos(n, 0);

solveNQueens(n, k, 0, queenPos);

return 0;

}

## 4、0-1背包（最优解问题）

Description：

给定 N 件物品和一个容量为 V 的背包。每件物品只能使用一次。

第i件物品的体积是vi，价值是wi。

你的任务是决定将哪些物品装入背包，以使得这些物品的总体积不超过背包容量，同时总价值最大。请输出最大价值。

Input：

第一行包含两个整数，分别为N和V，表示物品数量和背包容量。

接下来的 N 行，每行包含两个整数vi和wi，表示第i件物品的体积和价值。

Output：

输出一个整数，表示最大价值。

Sample Input：

4 5

1 2

2 4

3 4

4 5

Sample Output：

8

### 代码1：

#include <iostream>

using namespace std;

int n, v;

int v\_i[20]; // 第i个物品的体积

int w\_i[20]; // 第i个物品的价值

int bestw; // 最优价值，最优解

int x[20]; // 物品放入状态数组

int curv; // 当前体积

int curw; //当前价值

int bound(int x){ //计算价值上界

int rw = 0; //rw:第x个商品~第n个商品全部装入的总价值，先初始化为0

for(int i=x;i<=n;i++)

rw+=w\_i[x];

return rw+curw; //返回当第t个商品不装时，返回前t个商品(不包括第t个)的总价值+剩余的全部商品价值

}

void dfs(int step){

if(step > n){ //某个分支搜索到叶子节点，找到一个可行解

if(curw > bestw){ //如果当前价值大于最优价值，更新最优解

bestw = curw;

}

return ;

}

if(curv + v\_i[step] <= v){ //判断放入第step个物品不超重，尝试放入

x[step] = 1; //标记放入

curv += v\_i[step]; //累加体积

curw += w\_i[step]; //累加价值

dfs(step+1);

curv -= v\_i[step]; //回溯，恢复当前体积

curw -= w\_i[step]; //回溯，恢复当前价值

}

//剪枝

if(bound(step + 1) > bestw){ //判断不放入第step个物品，是否还有必要继续在该分支下搜索

x[step] = 0; // 尝试不放入第i个物品。这一行写在上一个if里也是一样的

dfs(step+1);

}

}

int main(){

cin >> n >> v;

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

cin >> v\_i[i] >> w\_i[i];

}

dfs(1);

cout << bestw << endl;

return 0;

}

### 代码2：

#include <iostream>

#include <vector>

using namespace std;

int max\_value = 0;

void backtrack(vector<pair<int, int>>& items, int capacity, int index, int current\_value) {

if (index >= items.size() || capacity <= 0) {

max\_value = max(max\_value, current\_value);

return;

}

// 不选择当前物品

backtrack(items, capacity, index + 1, current\_value);

// 选择当前物品（前提是容量足够，容量不够的话就相当于被剪枝了）

if (capacity >= items[index].first) {

backtrack(items, capacity - items[index].first, index + 1, current\_value + items[index].second);

}

}

int main() {

int N, V;

cin >> N >> V;

**vector<pair<int, int>> items(N);**

for (int i = 0; i < N; i++) {

cin >> items[i].**first** >> items[i].**second**;

}

backtrack(items, V, 0, 0);

cout << max\_value << endl;

return 0;

}

## 5、力扣17：电话号码的字母组合

class Solution {

public:

vector<string> letterCombinations(string digits) {

vector<string> combinations;

if (digits**.empty()**) {

return combinations;

}

**unordered\_map**<char, string> phoneMap{

{'2', "abc"},

{'3', "def"},

{'4', "ghi"},

{'5', "jkl"},

{'6', "mno"},

{'7', "pqrs"},

{'8', "tuv"},

{'9', "wxyz"}

};

string combination;

backtrack(combinations, phoneMap, digits, 0, combination);

return combinations;

}

void backtrack(vector<string>**&** combinations, **const** unordered\_map<char, string>**&** phoneMap, const string& digits, int index, string& combination) {

if (index == digits**.length()**) {

combinations.push\_back(combination);

} else {

char digit = digits[index];

**const** string**&** letters = phoneMap**.at(**digit);

for (const char& letter**:** letters) {

combination**.push\_back**(letter);

backtrack(combinations, phoneMap, digits, index + 1, combination);

combination**.pop\_back**();

}

}

}

};

class Solution {

public **List<String>** letterCombinations(String digits) {

**List<String>** combinations = **new ArrayList<String>();**

if (digits.length() == 0) {

return combinations;

}

**Map**<Character, String> phoneMap = **new HashMap**<Character, String>() **{{**

put('2', "abc");

put('3', "def");

put('4', "ghi");

put('5', "jkl");

put('6', "mno");

put('7', "pqrs");

put('8', "tuv");

put('9', "wxyz");

}};

backtrack(combinations, phoneMap, digits, 0, **new StringBuffer()**);

return combinations;

}

public void backtrack(List<String> combinations, Map<Character, String> phoneMap, String digits, int index, StringBuffer combination) {

if (index == digits.length()) {

combinations.**add**(combination.toString());

} else {

char digit = digits.**charAt**(index);

String letters = phoneMap.**get**(digit);

int lettersCount = letters.**length**();

for (int i = 0; i < lettersCount; i++) {

combination.**append**(letters.charAt(i));

backtrack(combinations, phoneMap, digits, index + 1, combination);

combination.**deleteCharAt**(index);

}

}

}

}

# 八、分支限界

## 1、BFS=Breadth-First Searc=宽度优先搜索=广度优先搜索

应用场景：图的最短路径、树的按层级遍历

void bfs(){

// 1. 初始化**队列**queue：将起始顶点加入队列

while ("队列queue不空") {// 2. 遍历整个队列

//3. 取出队列首部的顶点，访问该顶点，并**标记**为已访问

//4. 将所有未访问过且与当前顶点相邻的顶点加入队列

}

}

## 2、分支限界

工作原理：

分支：这一步涉及将问题分解成一系列的子问题。分支类似于回溯算法中的“递归”，用于探索解空间的不同区域。

限界：这一步涉及对子问题进行评估以确定是否继续探索这个分支。如果根据某些标准或启发式方法判断，当前分支不能产生比已知更好的解，则该分支被“剪枝”（不再探索）。

应用：

旅行商问题（TSP）、0-1背包问题、组合优化问题

路径和图搜索问题、任务调度和分配问题

|  |  |  |
| --- | --- | --- |
| **不同点** | **分支限界** | **回溯** |
| 目的 | 寻找最优解 | 找到所有可能的解 |
| 搜索方式 | BFS或最佳优先搜索 | DFS |
| 数据结构（子问题的存储） | 队列或优先队列（这使得算法可以按照不同的策略（如最小成本、最大边界等）来优先探索最有希望的子问题） | 栈 |

## 3、例题1：走迷宫

Description：

一个n×m的二维整数数组表示迷宫。数组中包含的元素只有0和1：

0表示这个位置是可以走的路径；1表示这个位置是不可通过的墙壁。

数组的起始位置(1,1)（左上角）和结束位置(n,m)（右下角）的元素保证为0。

迷宫至少存在一条从起点到终点的通路。

Input：

第一行两个整数n，m，其中1 ≤ n, m ≤ 100

接下来的n行，每行m个整数,只有0和1，中间用空格隔开

Output

输出一个整数，表示从左上角(1,1)到右下角(n,m)的最少移动次数

Sample Input：

5 5

0 1 0 0 0

0 0 0 1 0

0 0 0 0 0

0 1 1 1 0

0 0 0 1 0

Sample Output：

8

### 代码1：

#include <iostream>

using namespace std;

// 定义队列，用数组模拟

struct Node{

int x,y;

int step; //代表第几层遍历到该节点，（1，1）= 0

}q[10010];

int z[2][4] = { // 用x，y向量值变化表示方向，以下为下、上、右、左方向

1, -1, 0, 0, // 代表 x

0, 0, 1, -1 // 代表 y

};

int a[110][110];//存迷宫地图

bool flag[110][110];//标记x，y点是否被遍历过

int bfs(int n,int m){

//1、初始化队列queue，将第一个节点放入队列

int h = 0; // 指向队列头

int t = -1; // 指向队列尾

t++;

**q[t].x** = 1;

q[t].y = 1;

q[t].step = 0;

flag[1][1] = true;

//2、循环遍历队列

while(h <= t){ //队列不空

//3、取出队头,存入cur

**Node** cur = q[h];

h++;

//4、利用产生式规则，拓展cur的关联节点入队

for(int i = 0; i < 4; i++) {//四个方向拓展

int nx = cur.x + z[0][i];

int ny = cur.y + z[1][i];

if(nx < 1 || nx > n || ny < 1 || ny > m)//越界

continue;

if(a[nx][ny] == 1)//有墙

continue;

if(flag[nx][ny])//走过

continue;

//如果找到答案，结束搜索：**因为是层次遍历，所以第一个找到的答案就是最短的**

if(nx == n && ny == m){

return cur.step + 1;

}

flag[nx][ny] = true;

t++;

q[t].x = nx;

q[t].y = ny;

q[t].step = cur.step + 1;

}

}

//return -1;//表示没有通路

}

int main(){

int n,m;

cin >> n >> m;

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

for(int j = 1; j <= m; j++)

cin >> a[i][j];

cout << bfs(n,m) << endl;

return 0;

}

### 代码2：

#include <iostream>

#include <queue>

using namespace std;

struct Node {

int x;

int y;

int steps;

**Node(int x, int y, int steps) : x(x), y(y), steps(steps) {}**

};

int bfs(**vector<vector<int>>&** maze) {

int n = maze.size();

int m = maze**[0].size()**;

**vector<vector<bool>> visited(n, vector<bool>(m, false));**

**queue<Node>** q;

**q.push**(Node(0, 0, 0));

visited[0][0] = true;

while (!**q.empty()**) {

Node curr = **q.front()**;

**q.pop()**;

if (curr.x == n - 1 && curr.y == m - 1) {

return curr**.**steps; // 已到达终点，返回最少移动次数

}

// 向上移动

if (curr.x > 0 && maze[curr.x - 1][curr.y] == 0 && !visited[curr.x - 1][curr.y]) {

q.push(Node(curr.x - 1, curr.y, curr.steps + 1));

visited[curr.x - 1][curr.y] = true;

}

// 向下移动

if (curr.x < n - 1 && maze[curr.x + 1][curr.y] == 0 && !visited[curr.x + 1][curr.y]) {

q.push(Node(curr.x + 1, curr.y, curr.steps + 1));

visited[curr.x + 1][curr.y] = true;

}

// 向左移动

if (curr.y > 0 && maze[curr.x][curr.y - 1] == 0 && !visited[curr.x][curr.y - 1]) {

q.push(Node(curr.x, curr.y - 1, curr.steps + 1));

visited[curr.x][curr.y - 1] = true;

}

// 向右移动

if (curr.y < m - 1 && maze[curr.x][curr.y + 1] == 0 && !visited[curr.x][curr.y + 1]) {

q.push(Node(curr.x, curr.y + 1, curr.steps + 1));

visited[curr.x][curr.y + 1] = true;

}

}

return -1; // 无法到达终点

}

int main() {

int n, m;

cin >> n >> m;

**vector<vector<int>> maze(n, vector<int>(m));**

for (int i = 0; i < n; i++) {

for (int j = 0; j < m; j++) {

cin >> maze[i][j];

}

}

int minSteps = bfs(maze);

cout << minSteps << endl;

return 0;

}

## 4、例题2：0-1背包：难

Description：

给定 N 件物品和一个容量为 V 的背包。每件物品只能使用一次。

第i件物品的体积是vi，价值是wi。

你的任务是决定将哪些物品装入背包，以使得这些物品的总体积不超过背包容量，同时总价值最大。请输出最大价值。

Input：

第一行包含两个整数，分别为N和V，表示物品数量和背包容量。

接下来的 N 行，每行包含两个整数vi和wi，表示第i件物品的体积和价值。

Output：

输出一个整数，表示最大价值。

Sample Input：

4 5

1 2

2 4

3 4

4 5

Sample Output：

8

## 5、例题3：任务分配：难

Description：

假设有 ( n ) 个任务和 ( n ) 个人。每个人完成每个任务所需的时间都是不同的。你的任务是分配每个人恰好一个任务，以使完成所有任务的总时间最小。

Input：

第一行包含一个整数n,（1 <= n <= 20），表示任务和人的数量。

接下来的n行，每行包含n个整数，表示完成任务的时间矩阵。第i行第j个整数表示第i个人完成第j个任务所需的时间。

Output：

输出一个整数，表示最小的完成所有任务所需的总时间。

Sample Input：

3

8 5 6

4 6 7

6 8 9

Sample Output：

18

# 五、分治Divide and Conquer

基本步骤：

分解Divide：将问题分解成若干个更小的、相似的子问题。

解决Conquer：**递归**（recursive）地解决这些子问题。

合并Combine：将这些子问题的解合并为原问题的解。

常见的分治算法问题：

1、排序和查找问题：

Merge Sort、Quick Sort；

二分查找Binary Search

查找第k大（小）的元素：类似于快速排序，但只处理包含目标元素的那一侧

2、数学和计算问题：

Karatsuba算法：高效地将两个大数相乘

汉诺塔问题Tower of Hanoi：时间复杂度为O(2^n)，空间复杂度为O(n)（由于递归的深度是 n）

3、几何问题：

最近点对问题：在二维平面上，给定多个点，找出距离最近的一对点。

4、矩阵计算问题：

Strassen算法：用于快速矩阵乘法

5、测试与验证问题：

芯片测试：通过一系列的测试组合，快速确定哪些芯片是好的，哪些是坏的，尤其是在一批芯片中只有少量是坏的情况下。

6、字符串和匹配问题

## 1、简单的：

### 归并排序Merge Sort：递归地排序两个子数组、合并两个有序数组

### 逆序对的数量：仅仅需要在归并排序中加一变量cnt用于计数即可，O(nlogn)时间

### 第k小的数：仅仅需要在快速排序中加一个对递归的剪枝，使得只进行左半部分的递归或右半部分的递归即可；

### 模幂运算（快速幂）：利用a^(2b)=(a^2)^b和模运算的简单性质，O(log b)时间

### 最大连续子数组和：2个区间合并后的最大子序和=max（左子区间的最大子序和，右子区间的最大子序和，横跨两个子区间的最大子序和），O(nlogn)时间；

### 有序数组中查找数1

### 有序数组中查找数2

### 数的三次方根：令初始时l=-10000, r=10000

## 2、快速排序Quick Sort

基本步骤：

1、确定分界点（选择基准值 Pivot）：

选择一个元素作为基准值，如第一个元素、最后一个元素、中间元素，甚至随机元素等。

2、调整区间（分区操作 Partitioning）：

重新排列数列，所有比基准值小的元素摆放在基准前面，所有比基准值大的元素摆在基准的后面（相同的数可以到任一边）。

3、递归处理子序列：

递归地将小于基准值元素的子序列和大于基准值元素的子序列进行排序。

### 代码1：自己错误的改法：

#include <iostream>

using namespace std;

void quick\_sort(int a[], int l, int r){

if(l >= r)//递归出口

return;

const int x=a[l];//选择基准值，x=a[r], x=a[l+r>>1]等均可

int i = l;

int j = r;

**while(i < j){//开始分区操作**

**while(a[i] < x)//从左向右找到第一个>=x的数**

**i++;**

**while(a[j] > x)//从右向左找到第一个<=x的数**

**j--;**

**if(i < j){**

**swap(a[i],a[j]);**

**i++;**

**j--;**

**}**

**}**

**//此时有2种情况：**

**//i==j：只能保证i-1及之前都<=x，j+1及之后都>=x，如当输入为3 3 5 1时即可发现错误之处**

**//i==j+1：pivot既可能是在i上，也可能是在j上**

//输出调试部分：

cout<<"!"<<endl;

cout<<"x:"<<x<<endl;

cout<<"left:"<<endl;

for(int i=l;i<=j;i++){

cout<<"No."<<i<<":"<<a[i]<<endl;

}

cout<<"right:"<<endl;

for(int i=j+1;i<=r;i++){

cout<<"No."<<i<<":"<<a[i]<<endl;

}

quick\_sort(a,l,j);//递归地对左半部分进行快速排序

quick\_sort(a,j+1,r);

}

int main(){

int n;

cin >> n;

int a[n];

for(int i = 0; i < n; i++)

cin >> a[i];

quick\_sort(a, 0, n-1);

for(int i = 0; i < n; i++)

cout << a[i] << " ";

return 0;

}

### 代码2：老师版的，但不能让pivot在中间：

//当输入为7 10 7 8 9 1 5 2时，可以看出这个问题（即不能让pivot在中间）

#include <iostream>

using namespace std;

const int N = 1000010;

int n;

int a[N];

void quick\_sort(int a[], int l, int r){

if(l >= r) return; // 递归出口

int x=a[l]; //选择基准值， x=a[r], x=a[ l + r >> 1] 均可

int i = l - 1;

int j = r + 1;

while( i < j){ // 开始分区操作

// 从左向右找到第一个大于等于x的数

do i++; while(a[i] < x);

// 从右向左找到第一个小于等于x的数

do j--; while(a[j] > x);

if(i < j) swap(a[i],a[j]);

}

//输出调试部分：

cout<<"!"<<endl;

cout<<"x:"<<x<<endl;

cout<<"left:"<<endl;

for(int i=l;i<=j;i++){

cout<<"No."<<i<<":"<<a[i]<<endl;

}

cout<<"right:"<<endl;

for(int i=j+1;i<=r;i++){

cout<<"No."<<i<<":"<<a[i]<<endl;

}

quick\_sort(a,l,j);//**[l,j]都<=x**

quick\_sort(a,j+1,r);//**[j+1,r]都>=x**

}

int main(){

cin >> n;

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

quick\_sort(a, 0, n-1);

for(int i = 0; i < n; i++) cout << a[i] << " ";

return 0;

}

### 代码3：可以让pivot到自己最终的位置上：

#include<iostream>

using namespace std;

int partition(int arr[], int low, int high) {

int pivot = arr[high];

int i = low;

for (int j = low; j <= high-1; j++) {

if (arr[j] < pivot) {

swap(arr[i], arr[j]);

i++;

}

}

swap(arr[i], arr[high]);

return i;

}

void quickSort(int arr[], int low, int high) {

if (low < high) {

int pi = partition(arr, low, high);

quickSort(arr, low, pi - 1);

quickSort(arr, pi + 1, high);

}

}

int main() {

int arr[] = {10, 7, 8, 9, 1, 5};

int n = sizeof(arr)/sizeof(arr[0]);

quickSort(arr, 0, n-1);

for (int i=0; i < n; i++)

cout << arr[i] << " ";

return 0;

}