规范1：if()判断语句永远固定值在左

Eg.if(0==n){

}

规范2：永远不要遗漏任何花括号

**关于不确定数量的输入处理**

While (scanf(“%d”,&h)!=EOF)

Scanf的返回值为整数，表示正确读取内容的个数；

如果使用ctrl+D，表文件终止符，返回-1.

EOF明确表达文件终止的意义。

**图形打印的通法**

定义二维数组绘制图形，便于从任意位置开始对图形进行绘制；

步骤：

一般定义为全局变量且为固定大小

其次要初始化，把所有位置设置为’\0’

按照题目条件，从任意位置开始设置数组

使用printf(“%s”)配合循环打印每一行/列输出

**作用域问题**

若for(int i=0;;){}...

或者for(){

Int i=0;

}

作用域都只在循环体内部

**Flag的巧用**

用以表示是否是第一个结果，从而增加换行、空格等

Eg.

Bool flag=True;

If(flag==true){

Flag=false;

}else{

Printf(“ ”);

}

**日期问题**

①数组实现空间换时间：定义每个月天数的数组；

②闰年判定：if(n%400==0||n%100!=0&&n%4==0)

通法:

//nextday

For(int i=0;i<n-1;i++){

Bool isleap=year%400==0||year%100!=0&&year%4==0;

If(isleap){

Mday[2]=29;

}

Day++;

If(day>mday[month]){

Day=1;

Month++;

If(month>12){

Year++;

Month=1;

}

}

}

**输出处理:**

%d

%4d

%04d

Double-%.1lf 带1位小数点

**字符串的处理**

输入输出用C风格

Char str[100];

Scanf(“%s”,str);

Printf(“%s”,str);

复杂处理用C++风格

#include <string>

Using namespace std;

初始化 string str1=str;

连接 str1=str1+”world”;

字符 str1[0];

长度 str1.length( );

判断相等 str1==”hello”;

比较字典序 str1>”abandon”

转化为C风格 str1.C\_str();

如何删除fgets录入的换行符？Str.erase(str.size()-1);

清空 str.clear( );

删除 str.erase( );//注意参数可以是指针，可以用str.begin( )；也可以是下标，如参数只有下标则表示删除包括该元素及其后的的所有字符串，或者参数添加下标及长度

切割 str.substr(int index,int length);//前者为起点，注意：后者为长度，如果终点为末尾则无需长度， 且该语句可直接作为子串使用

查找 str.find(ElemType object);//返回值为目标元素的下标，若没找到则返回值为string::npos，如有多个目标相同，则返回第一个目标的下标。

注意与unordered\_set.find返回值区分，前者为npos，后者为end()

注意一开始str长度为0，填入字符时不宜用str[i]=...，应直接用str.push\_back( ... )

数组的指针可以直接当做string类型作为函数传参，如：

Char arr1[10];

Char arr2[10];

Build\_tree(arr1,arr2);//该函数的两个参数均为string类型

Map(C++风格)

#include <map>

Using namespace std;

Map<string,string> myMap={

{“……”,”……”}，

{ }

}

**排序**

#include<algorithm>

Sort(begin,end，comp);

Eg:sort(vec.begin(),vec.end(),compare);//最好不要写vec[0],vec[n-1]这种形式

begin为开头，end为尾后，即结尾元素后一位，相当于左闭右开区间

comp是比较交换的条件，要另定义一个函数描述最原子的比较交换条件；

即考虑什么情况下不发生交换。

如何维持稳定性？

定义一个元素含seq成员的数组，在判定时增加

***Sort(arr,arr+n,comp);//万分注意***

注意不可以写成：

Sort(arr[0],arr[n],comp);//参数必须是指针

if(arr[lhs].grade==arr[rhs].grade&&arr[lhs].seq<arr[rhs].seq){

Return true;

}

**用map代替二分查找**

1. 初始化

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

Scanf(“%d”,&arr[i]);

Map[arr[i]]=i;

1. 判定查找结果

If(map.find(arr[i])==map.end()){

Printf(“NO\n”);

}else{

Printf(“YES\n”);

}

**区分字符还是数字的方法：**

If(arr[i]>=’A’&&arr[i]<=’Z’||arr[i]>=’a’&&arr[i]<=’z’){

Printf(“%c ”,arr[i]);

}

Else if(arr[i]>=’0’&&arr[i]<=’9’){

Printf(“%d ”,arr[i]);

}

**字符串**

读取：scanf只能读一整个没有空格的字符串

fgets可以读取一整行

Fgets(buf,100,stdin);//但是会读入换行

**注意下列情况：**

Scanf(“%d”,&n);

Fgets(arr,100,stdin);

String str=arr;

Str.pop\_back( )

此时，str中只有”\n”，因为在输入n值时会换行，换行符被fgets读入，进入str

正确做法为：

Scanf(“%d\n”,&n);//提前输入换行

查找：string str=”howareyou”;

Int pos=str.find(“are”);

If(pos!=string::npos){

Printf(“Found,pos=%d\n”,pos);

}else{

Printf(“Not Found\n”);

}

字符串的迭代器，就是指针

String::iterator it=str.begin();//定义一个迭代器it为起始值

Printf(“%c\n”,\*it);//通过地址访问元素

**数组的限制**

数组定义时必须用给定常量指定大小

错误做法：

Int n=0;

Int arr[n];

函数内定义数组不能过大，大数组应该作为全局变量。因为给函数分配的栈空间有限。

**使用vector(向量、或动态数组)代替数组**

#include <vector>

Using namespace std;

//定义

Vector<int> vec;//长度为0

//初始化

Vector<int> vec{1,2,3};

//尾部扩容

Vec.push\_back(1);

//设置长度

Vector<int> vec(1000);//值全为0

Vec.Pop\_back( );//弹出末尾元素

Vec[i];//i从0开始直到n-1

Vec.size();//当前容量

//迭代器

Vec.begin( ) vec.end( )//尾后

For(vector<int>::iterator it=vec.begin();it!=vec.end();it++ ){

Printf(“vec[ ]=%d\n”,\*it);

}

//指定元素之前插入

Vec2.insert(it,4);//It即指定元素，4即插入值

Vec2.erase(it);//删除指定位置元素

原理：

Vector的空间在堆空间上，栈上只有堆空间的首地址（ptr）

当size和capacity相等时，先在堆空间申请2\*capacity的内存，再拷贝原数据，最后释放原内存。

而插入的时间复杂度为O(N)

注意：

Vector<edge> graph;//定义一个向量

Graph[x].push\_back( edge);//错误

Vector<edge> graph[300];//实质为定义一个元素为向量类型的数组

Graph[x].push\_back(edge);//正确

**队列的实现**

#include <queue>

Using namespace std;

Queue <int> q;

For(int i =0;i<5;i++){

Q.push(i);

}

While(true){

If(q.empty){

Printf(“empty\n”);

Break;

}else{

Printf(“%d ”,q.front( ));

Q.pop( );//注意：int cur=Q.pop( )；是错误的

}

}

模拟循环队列的方法

先pop再push

**栈的实现**

Stack <int> S;

.size( );//栈的元素多少

.push( );//压栈

.top( );//获取栈顶元素

.pop( );//弹栈

.empty( );//判定栈是否为空

**整型的数据类型**

当告知整型数的范围时，应考虑用哪种数据类型表示

Int 32bit -2^31~2^31-1,-2\*10^9~2\*10^9

Unsigned int 32bit 0-4^10^9

Long long 64bit -2^63~0~2^63-1,-8\*10^18~8\*10^18，占位符%lld

Unsigned long long 64bit,0~2^64-1,0~16\*10^18

double类型输出用%.2lf(占两位小数)

**读取一整行字符串**

Char buffer[1000];

Fgets(buffer,1000,stdin);//ctrl+D输出NULL(不是EOF)，故写为while(fgets(arr,200,stdin)!=EOF)

会额外读一个换行

string str=buf;

Str.pop\_back( );

**stod函数**:即把string类型to double类型

**Switch的用法**

Switch(oper){

Case ’+’:

Numstack.push(lhs+rhs);

Break;//必须记住

Case ’-’:

Numstack.push(lhs-rhs);

Break;

}

**如何把字符数组的多个数字变成一个数**

String型数据收集

String num=””;

If(str[i]>=’0’&&str[i]<=’9’){

Num.push\_back(str[i]);

}else if(str[i]==’ ’){

If(num!=’’){

Numstack.push(stod(num));

Num=””;

}

}else{

If(‘$’==str[i]){

If(num!=’’){

Numstack.push(stod(num));

Num=””;

}

}

}

**递归的使用**

首先要找到从大问题转换为小问题的思路，并且提出解决最小问题的方法和递归出口。

注意：递归出口必然出现在被反复调用的递归函数中。

**在编写代码时务必假设小一级（F(n-1)）的问题已经解决的情形下，如何解决大问题(F(n))。往往需要把小一级的问题视作整体。**

大问题就是函数定义，小一级的问题就是函数调用，最小的问题就是递归出口，问题规模不断缩小的过程就是函数调用栈不断入栈的过程，由小问题不断解决大一级的问题就是函数调用栈不断弹栈的过程。

**分治法**

分：把大规模问题分解为若干个相似的小问题

治：将这些规模更小的问题逐个击破，找到边界，解决最小问题

合：合并小问题的解决方案，直到大问题解决

模板：斐波拉契数列

代码模板

If(n为边界规模){

直接解决；

}else{

①Function(n-1)/Function(n/2);//逐个解决小问题

②把问题合并为大问题

}

解决复杂的递归、模拟等问题，一定要画流程图

**指针与引用**

Int\* i;//\*是声明变量为指针，而int是指针的基类型，指针只存储了变量的第一个字节的地址，基类型则指明变量的范围。

当需要通过调用函数改变原函数的变量值时，需要引用。

如何通过指针申请自由空间：

Int \*addNode(int i){

Int \* newNode=new int;//new是在堆空间上申请一片空间，空间大小根据其后变量类型而定，注意声明必须为指针类型，但new后为指针指向数据的类型

\*newNode=i;

Return newNode;

}

Int main( ){

Int \*arr[10];//存储所申请空间的指针

For(int i=0;i<10;i++){

Arr[i]=addNode(i);

}

For(int i=0;i<10;i++){

Delete arr[i];//delete函数用来释放所申请空间，避免内存泄漏，注意delete后必须是指针

}

}

**利用含‘#’层序遍历序列的层次建树的逻辑（重难点）：**

初始时，声明 TreeNode \* root并置为NULL;

创建一个队列存储结点指针；

依次访问字符串，

如果非#，

则创建新的TreeNode;

把TreeNode 指针入队，并置isLeftIn=false；

如果root为空，则令root为TreeNode指针；

否则，如果isLeftIn为false使出队结点左孩子指针赋值为新TreeNode地址，否则令右孩子指针赋值为TreeNode地址，并出队；

如果为#，

则直接令队列出队，使出队指针置为NULL;

注意：①处理复杂问题，首先举例，罗列详细的步骤，然后抽象为文字逻辑，再写代码，能大大提高效率。

②辅助队列元素类型应该是二叉树结点的指针，如果是二叉树结点类型，那么修改队列元素的成员的值时，无法作用到原树结点

③TreeNode和QueueNode都应该建立在堆空间，否则函数调用结束结点即销毁

④出队、入队、插入等操作均与root无关

⑤new/delete函数后均只能是指针

指定字符型数组

Char charlist[]=”abcdefg”;

利用显式空结点的先序遍历可以构建

**利用中序、先序序列建树的逻辑**

中序序列的第一个元素为根节点；

在先序序列中找到根节点，得到左右子树的结点数；

回到中序序列找到对应数量的子序列，对子序列分别调用自身函数；

如果序列长度为0则返回NULL;

**建立二叉搜索树**

①如果树为空，则新插入的结点即根；

②如果树非空，则与根的data域比较大小，

若比根更大，则到右子树；

若比根更小，则到左子树；

③如果当前结点为空，则插入；

注意：非递归方式更简单

string返回值类型的前中后遍历的简便方法

Return root->data+PreOrder(root->lchild)+PreOrder(root->rchild);

**注意：以上建树过程root初始时必须先置为空指针**

**优先队列**

适合反复求某一集合的最值，求一次就剔除该数或加入新的数。

类似大根堆，采取二叉树的数据结构。

#include <queue>;

Using namespace std;

Priority\_queue<int> myqueue;//构建大根堆

Push,pop,top,empty用法与栈或队列类似。

如何自定义priority\_queue的优先级规则：

Struct Element{

Int value;

}//只有自定义的类才能重载运算符

Bool operator <(Element e1,Element e2){//返回值类型为bool,且函数名不能自定义

Return e1.value > e2.value;//只有把小于符重载为>，才能建立小根堆:返回true时触发交换条件

}

Int main( ){

priority\_queue<element> myQueue;

int arr[ ]={5,3,1,2,4,6};

for(int i=0;i<5;i++){

Element e;

e.value=arr[i];

myQueue.push(e);

}

}

注意：不建议按照上面的方式自定义优先级，无法定义元素类型为指针的优先级队列。

如：priority\_queue<ListNode\*> pq;

使用下面的模板：

Struct compare{

bool operator()(ListNode \*e1,ListNode \*e2){ // operator后面应该有()空括号，表示重载()；

return e1->val > e2->val; // 定义小根堆

}

}

Priority\_queue<ListNode\*, vector<ListNode\*>,compare> pq;

①只能定义小根堆，无法定义大根堆

②适合优先队列元素类型为指针的情况

③优先队列传入的三参数为<元素类型，容器类型，比较参考结构体>

容器类型一般默认为vector

④自定义的比较号与sort刚好相反，**定义小根堆符号为大于号。**

⑤不能重载>号

**复数的定义方式**

Struct complex{

Int re;

Int im;

}//模为re\*re+im\*im;

**建立小根堆的简易方法**

myQueue.push(-leaf);//改造成负数，输出时把负数改为正数即集合中的最小值

**输入含空格的数字串的特殊方法**

法一：

Eg.2 1 3 5 2 2

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

Scanf(“%d”,&Arr[i]);

Scanf(“%c”,&useless[i]);

}

最后数字都会依次存储在arr数组，空格及换行符会存储在useless数组

法二：

Int arr[100];

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

Scanf(“%d”,&arr[i]);

}

**map与unordered\_map**

map底层为红黑树（BST），时间复杂度为O(logN)，数据有序

Unordered\_map底层为散列表，时间复杂度为O(1)，但数据无序且所占空间更大，用法与map相同。

注意：MAP无法存储相同键不同值

Map的基本操作：

Map<string,int> myMap;//<键，值>

If(myMap.empty( )){

Printf(“myMap is empty.\n”);

}

//插入法一：[ ]

MyMap[“CaiXukun”]=1;

//插入法二：insert

MyMap.insert(pair<string,int>(“WuYifan”,2));

Printf(“%d\n”,myMap.size( ));

//myMap的遍历方法——迭代器

Map<string,int>::iterator it;

For(it=myMap.begin();it!=myMap.end();it++){

Printf(“%s,%d\n”,it->first.c\_str,it->second);//it指向的元素含两个成员<first,second>

}

//注意，插入后已经按照键的大小顺序排序

//查找方法

It=myMap.find(“Bob”);//find内参数必须是键，返回迭代器

//删除

myMap.erase(“Banete”);//erase内参数可以是键

myMap.erase(it);//也可以是迭代器

If(it!=myMap.end( )){

Printf(“Found!%d,%s\n”,it->first.c\_str,it->second);//注意it->first即键为string类型，因此要转换为C风格才能打印

}else{

Printf(“Not found!\n”);

}

注意：map.end( )是尾后的指针，map.rbegin( )是末尾的指针（r表示逆序）

**利用map如何双向查找？**

存两次。如：

Map<string,string> myMap;

myMap[“ab”]=”AB”;

myMap[“AB”]=”ab”;

map函数常用查找和初始化方法

If(myMap.find(arr[i])!=myMap.end( )){

Int count=myMap[arr[i]]+1;

myMap[arr[i]]=count;

}else{

myMap[arr[i]]=1;

}

**切割子串的常见方法**：string.substr(i,length);

**编写复杂问题代码的重要原则**：增量编写法，便于后期出现问题打准断点。

**搜索问题**

搜索是一种特殊的枚举，枚举是自由转移，按照任意方式列出所有状态。搜索是有限定方式的转移，按照特定方式列出状态（如树等）。

BFS 广度优先遍历

①创建辅助队列，创建isVisit数组，把初始结点入队；

②循环，直到队列为空：

1. 取出front，修改isVisit;
2. 入队front的所有邻居结点（排除isVisit）；
3. 访问front;

适合求解最优路径、数字串的问题。

**超时的解决方法**

如：输入值n<=200，即有200种情况，因此可以提前在编程环境中输出这两百种情况（形如0,1,2,3,4……），然后另外写一个程序，将这两百种情况存储在数组，根据数组直接打印即可。

**DFS的基本思路**

适合解决有无解/路径的问题

Bool DFS(node cur,node end){

将cur压栈;//存储路径

isVisit修改为true;

If(end==cur){

Return true;

}

While(遍历cur的邻居&&isVisit==false){

If(DFS(cur邻居,end)==true){//存在路径

return true;

}

弹栈;

}

Return false;

}

注：

①isVisit的修改位置

②路径用stack储存

Void dfs(int cur,int pos){

If(cur==aim){//达成目标，递归出口

Sum++;//统计达到该目标的方法数目

Return;

}

For(int i=pos;i<n;i++){

If(cur+arr[i]>aim||visit[i]){

Continue;//不合法

}

Visit[i]=true;//合法即访问

Dfs(cur+arr[i],i+1);//搜索从i+1是否开始存在达成目标的方法，而不是position+1

Visit[i]=false;//解除锁定，以便后续使用

}

Return;

}

//**复杂情况解除锁定的方法：复制老数组**

循环复制visit[10][10]到old\_visit[10][10]；

DFS递归；

复制old\_visit到visit；

**上述伪代码用于解决计算能达成目标的方法总数问题，请记忆该代码模板。**

快速初始化数组的方法

#include <cstring>

Using namespace std;

Memset(arr,false,sizeof(arr));//<数组名，初始化内容，大小>

**重要提醒：没思路超过一个小时停笔，无法运行或者通过率低超过两个小时停笔！**

注意：①如果设置全局变量，递归结束后记得释放锁定。

②如果递归前修改某变量进入递归，递归后应复原。

③递归时用迭代指针，而非静态position+1.

**动态规划**

即在递归的记忆化方案思路基础上，只采取问题规模有小到大的过程。

三要素包含：

①状态：定义一个状态来陈述子问题。子问题一般是原问题的弱化版本，如规模更小情形下如何解决问题。是动态规划思路最重要的部分。对应于Fibonacci函数的dps[ ]数组。

②阶段：子问题的解决方案。基于已经解决的子问题处理未解决的问题。对应于Fibonacci函数的F(i-1)+F(i-2).

③决策：从一个状态转移到另一个状态，就是递归的回归问题。注意：这种转移不能有回头路，即每种状态只能出现一次。

**基本思路：**

1. 采用递推方法解决问题
2. 考虑重复的子问题
3. 设计状态进行记忆，设计阶段模拟递归的回归。

或，

设计状态（包含边界），分析如何用规模更小的dp[ ]解决dp[i].

如果要求序列连续，一般只用考虑dp[i-1]，否则考虑dp[j](j:0到i-1)

解决序列段的问题，常用动态规划，状态的设计中，**务必记住目标序列要包含端点**。

如求序列串的最长子序列，设计状态dps[i]为前i个数中子序列的最大值，且包含a[i]点，以使加入新元素时不会中断。

**经典问题：最大子矩阵**

思路：

以最大连续子序列为基础，采取压缩矩阵的思路，把二维数组转换为一维。

设置辅助数组sum[][]，计算每列不同行到第一行的总和，注意第一行设置为全0，因此后续计算行数要加减处理。

对矩阵遍历，分别固定上下边界，对列遍历，分别计算各列元素的和，存储到一维数组total[ ],即完成对二维数组的压缩。

对该数组求取最大连续子序列，并求最大值。

**图论**

**创建图的方法**（邻接表法）：用向量模拟链表。

Struct Edge{

Int y;//边的对端顶点

Int weight;

}

Vector<Edge> graph[N];

Void addEdge(int x,int y,int weight){//x为边起点，y为边终点

Struct Edge edge;

Edge.y=y;

Edge.weight=weight;

Graph[x].push\_back(edge);

}

**检查连通性的方法**：并查集算法（查、并操作）。

设置连通子图个数n，不断合并点，在合并函数中若根不同，则使n减1。若n==1则图具有连通性。

find路径压缩：使find路径上每个顶点的父亲都是根节点。

Father[x]=find(father[x]);

union路径压缩：小树合并到大树。

构造height数组，存储根所在树的高度。

注意下列find函数写法不正确。

Int find(int x){

If(x==father[x]){

Return father[x];

}

Father[x]=find(father[x]);

}

记住int返回值类型返回语句必须放在最后。

**动态规划专题**

**特点**

常用于求解最值问题，具备重叠子问题和最优子结构，能够构造状态转移方程对所有状态和选择进行穷举的问题。

**注释：**

重叠子问题常出现在暴力递归求解时，一个子问题被重复计算，如Fibonacci；

最优子结构指能够由子问题推出原问题，如由每个班级的最高最低分差无法推出全校的最高最低分差，因此不具有最优子结构。

**解题思路**

一：状态。状态就是发送变化的量，一般是函数传递的变参，或者循环使用的索引。状态可以有多个，多个状态导致dp数组有多维。

For(状态1){

For(状态2){

Dp[状态1][状态2]=求最值（选择1，选择2，……）；

}

}

二：选择。导致状态变化的量，如零钱问题中导致金额变化的量就是硬币数额，各种硬币数额即选择。

三：定义。dp数组清晰的定义或含义，因此可以写出状态转移方程。

四：base case.即初始化最基本的情况。

**初始化**

一般大小为N+1，取值也需要符合后续判断。

**重叠子问题的判定：**

如果存在多种路径可以使一种状态转移到另一种状态，那么必然存在重叠子问题。

**遍历顺序或状态转移方向的确定：**

由dp数组的定义，可以确定base case以及返回目标答案，从而得出欲求dp[i][j]所需dp[□][□]，可以画出求解矩阵和图示辅助分析，最后得到遍历顺序，遍历顺序直接影响到状态转移的方向，该方向决定了思考起点。

如编辑次数问题，由遍历次序均为从小到大可以确定要求dp[i][j]必须直到dp[i-1][j-1]，因此对dp数组的遍历从前往后，但思考上对字符串的扫描是从后往前。

出现段错误警告常见的原因是数组设置太小以致越界。

**索引偏移**

由于base case置dp[..][0]和dp[0][..]为0，因此原数据整体后移，在dp数组和一般数组的换算间一般要i-1.

注意：base case情况不同赋值也可能不同。如背包问题中dp[0][..]=MAX;//前0个物品达到任何重量所需数量均为MAX；但前0个物品达到任何重量所达到价值均为0.

背包问题实质上是条件最值问题

**二叉树专题**

**两种思维模式：**

①遍历：即采用一个traverse函数配合外部变量解决问题。

②分解问题：采用递归，一般要充分利用返回值，常出现在处理子树问题，通常做法是写明定义，对左右子树作相同处理，再对根特殊处理，往往采取后序遍历。

复杂的算法往往建立在简单的数据结构上，如快排就是二叉树的前序遍历，归并排序就是后序遍历。

**对前序、中序、后序位置的再理解**：

即进入结点的不同时机。前序就是进入结点，后序就是离开结点，中序就是离开左子树进入右子树的时机。

关键在于不同位置插入代码逻辑，考虑一个节点需要做的动作。

前序访问自顶向下，后序访问自底向上，因此在获取数据上不同，前序只能接受参数和外部变量的值，而后序可以充分利用递归函数传递的返回值。

**回溯（DFS）专题**

实质：即穷举，遍历二叉树到叶子结点得到可行解。

**遍历二叉树的框架**

Void traverse(TreeNode\* root){

If(root==NULL){

Return;

}

for(TreeNode \*child: root->children){

traverse(child);

}

}

**回溯框架**

Vector<value> res;//可以以vector为元素类型

void backtrack(路径，选择列表){

If(满足结束条件){

Res.add(路径)；

return;

}

for(选择：选择列表){

做选择；

Backtrack(路径，选择列表)；

撤销选择；

}

}

Find使用方法

If(find(vec.begin(),vec.end(),key)!=vec.end())//必须引用algorithm;表示找到元素

**注意：**

**vector作为形参时一般需要引用，避免出错！**

for(int i=0, j=n; i < n && j >= 0; i++, j-- )//双变量在写约束条件时注意用运算符&&

**回溯框架解决排列、组合、子集问题**

**本质**就是对树的遍历。

一般有三种情况：

①元素没有重复，不可重复选择；

②元素重复不可复选

③元素不重复可复选

三个问题3种形式，共9种情况。

**经验：**

①组合、子集本质为同一类问题，一般用start控制其相对位置。

②有重复元素一般采取排序，重复元素必然相邻，且组合问题中重复元素必须大于start。

即num[i]==num[i-1]&&i>start

③相同元素排列时相对位置变化无意义。

综合②③可知num[i]==num[i-1]无法作为独立条件判断。

④可重复选元素时，注意避免树无限生长的条件。

传入二维数组作为函数参数的方法

Void find(arr[3][10]){}

Void find(arr[][10]){}

说明列数是必须的。

Square问题经验总结：

①路径是否存在一般backtrack用布尔型

②组合木棍组成边时类似于组合问题，要用start控制避免重复；

③一旦backtrack用布尔型，在以下情境务必注意：

If(backtrack()){

Visit[i]=false;

Return true;

}else{

Visit[i]=false;

}

即backtrack函数无论返回值为t/f均需要解除锁定，因此也可简化为：

If(backtrack()){

Return true;

}

Visit[i]=false;

记住解除锁定不一定需要放在return前面。

④函数最后的返回值为return false;

⑤**务必记得backtrack函数里传递的实参一般是索引（i）而非pos（形参传递的值）**

**BFS框架**

**本质**是图的层序遍历的推广，一般用于求路径最优解问题。

**步骤：**

先确定起点、终点，确定选取规则，一层层向下遍历，有三层嵌套循环。

**与DFS的区别：**

DFS需要遍历完所有结点后才能确定结果，BFS可以中途暂停。

但BFS需要存储每一层的结点，DFS仅仅是堆栈的深度，因此空间复杂度比BFS更小。

框架：

Int bfs(node start,node target){

Queue<node> q;

q.push\_back(start);

Int step=0;

Unordered\_set<node> visited;

Visited.insert(start);//visited是unordered\_set类型

While(!q.empty()){

Int sz=q.size();

For(int i=0;i<sz;i++){

Node cur=q.front();

q.pop();

If(cur==target){

Return step;

}

For(node j: cur.adj){

If(visited.find(j)==visited.end()){

q.push(j);

Visited.insert(j);

}

}

}

Step++;

}

Return -1;

}

技巧：如果q队列元素为二维数组，那么一般要抽象为一维。

Unordered\_set 适合查找、插入，查找时间复杂度为O(1)

需要include<bits/stdc++.h>

注意vector类型无法被哈希

注意：在使用BFS时，应对入队元素进行限定和判断，避免入队元素太多以致超过限制内存。

技巧：如何把其他类型数据转换为字符串--to\_string(key);

注意如果输入数据为整数型，但要求转换为字符串，若输入的数据首数字为0则会被省略，因此应着重考虑。

注意：输入时以下语句不合法

String str;

Scanf(“%s”,str.c\_str);//无法输入；只能先输到数组再令str=arr

**二分搜索框架**

思路清晰但是代码细节多。

应注意的**细节**：

①搜索区间的确定。

依赖于left/right的初始值确定，使搜索区间表现为左闭右闭还是左闭右开区间。如：

Right = num.length-1，搜索区间为左闭右开。

②终止条件的确定。

依赖于搜索区间的右边界开闭，若右开则终止条件为left<right

③参数的更新。

依赖于搜索区间的开闭，如右开且num[mid]<target时，right = mid;

**拓展题型**：寻找目标值的左边界或右边界

假设搜索区间为左闭右闭，则

当num[mid] == target时，不立刻返回

如果寻找左边界，

Right = mid - 1;

如果寻找右边界，

Left = mid + 1;

循环结束时，left = right+1，此时存在三种情况，

如果寻找左边界，

当num[right+1] != target 或者 left >= num.length,

返回-1；

当num[right + 1]==target.

返回right +1;

如果寻找右边界，

当num[left-1] != target 或者 right < 0,

返回 -1；

当num[left - 1]==target.

返回left - 1;

注意：始终记得循环结束被移动的指针是非法的，需要回移一次检验最后一个mid指针。

寻找左边界移动right，寻找右边界移动left。

left可以上溢，right可以下溢

**二分查找的应用拓展**

**思路**：只要是运用到连续的线性搜索算法，都可以尝试使用二分查找。

关键在于确定搜索区间、目标及函数关系。

**应用场景；**

如果简单思路符合以下框架，均可尝试二分查找。

For x in range(left, right):

If(func(x)满足某条件):

Return x;

**优化：**

①确定函数关系。

明确自变量和因变量，题目一般限制因变量的最值，求解自变量的最小值；

运输问题中题目给定运输天数不得超过days，求解最小承重能力capacity.

②确定左右范围。

即找到自变量的变化区间，最小值就是left，最大值就是right.

③确定查找条件。

一般是函数关系，且因变量应满足给定条件。

运输问题中查找条件即编写函数less\_than\_days,传递参数承载能力（自变量），返回运输天数（因变量）,判断返回值与题目给定限制关系，调整查找区间。

注意：①如果为递减函数，则返回值更大应使自变量更小，反之亦然。

如果为递增函数，则返回值比限制更大应使自变量更大，恰好相反。

自变量搜索区间[left, right]一定是递增

②尽量不要写else，全部写else if把条件展示清楚，避免出错。

**由此可知：**

必须严格按照框架返回值，因为结果不一定恰好等于题目给定限制。

如运输问题要求运输天数为6，但按照所设定的运输能力无法达到，最小运输能力刚好使运输天数也为5，运输能力再小则运输天数为7。

参考阶跃函数的特性。

代码框架：

Int left = min;

Int right = max;

While (left <= right){

Mid = left + (right - left) / 2;

If ( f(mid) < target ){

使f (x) 变大；

}

Else if ( f(mid) == target){

使 x 减小； // 寻找左边界

}

Else if ( f(mid) > target){

使f(x) 减小；

}

}

If (right < 0 || f(right + 1) > target){ // 函数值不满足条件

Return -1;

}

Return right + 1;

《丑数》总结

数学方法：

求满足多个条件的数，往往要用集合论知识。如满足A或B或C的集合大小，为setA + setB + setC - setAB - setAC - setBC + setABC

求[0, .., num]中能被a,b,c整除的数的个数 = num / LCM(a,b,c)。LCM是a,b,c的最小公倍数。

求最小公倍数 = a\*b\*c / gcd（a,b,c）。gcd就是a,b,c的最大公因数。

求最大公因数用辗转相除法。gcd(a,b)，a>b，a作为被除数，b作为除数，求得余数a%b，再求gcd(b,a%b)，直到余数为0，b即最大公因数。

二分查找：

自变量是数的范围num，函数是该范围内丑数的个数。

一般都要求左边界，f(num)==n时，当num减小一段距离可能仍然得到n

**链表题型技巧**

**类型1**：合并、分解若干链表。

常用方法：“另起炉灶”。

建立若干新的头结点创建新链表，再分别遍历原链表按题目条件加入新链表。

如果要合并K个链表，常用优先队列比较N个数的大小，注意需要定义新数据类型和运算符。

注意：如果修改原链表结构，如新建链表的指针直接指向原链表结点或者等于其指针，那么最好把原链表指针p->next置为空，与后续链表断链，即遍历一个结点丢弃一个，否则当新指针指向原链表中间一个结点会出现循环现象。

注意无论是合并两个链表还是返回头结点，都为dummy->next，dummy仅为占位符（空头结点）

**类型二**：双指针法，运用不同距离/速度的指针。

删除第K个结点，即找到第K+1个结点，当链表长度为1时，会出现非法情况，因此必须提前设置占位符DUMMY。

计算链表环的入口，注意起点到入口的距离等于相遇点到入口的距离，因此两个指针相遇后让slow返回起点，与fast指针齐头并进，即找到入口。

《合并K个链表》经验：

①充分考虑非法情况：传进来的向量为空；容器内的元素为空；将取出元素的后继加入堆时检查是否为空。

②小心使用优先队列：自定义小根堆，用大于号，模板是建立结构体定义Bool函数，重载operator()，而不能重载operator <，因其不能传入指针。

**数组双指针技巧**

**类型一**：快慢指针（slow,fast）。

常运用于原地修改数组，如删除元素等。

注意：在使用fast向前探路时，尽量不要再套用循环。

**类型二**：左右指针（left, right）。

常用于有序数组，如找特定和、翻转、回文串等，常为相向而行。

特殊情况会出现从数组一点背向而行，如求最大回文串。

技巧：当同时有两种情况且处理不同时，往往统一为复杂的情况处理。比如求最大回文段，当回文段为偶数与为奇数时，起点不同，偶数起点有两个，奇数有一个，统一为偶数情况。

注意：遍历时一般按照0~n-1范围遍历，尽量不要提前把可能的非法情况排除，否则容易遗漏特殊用例，如输入数据为空或仅一个等。

Two sum经验：

Map常见两种方法，其一把数组元素和索引按键值对全部存储，其二把要寻找的目标元素（不一定存在）放入map，遍历数组时检查map中是否存在，若存在则说明原数组确实存在寻找的目标元素。

当需要使用map而存在重复元素时，一般使用第二种方法。

vector<int> twoSum(vector<int>& nums, int target) {  
 unordered\_map<int, int> mp;  
 vector<int> index;  
 for (int i = 0; i < nums.size(); ++i) {  
 if (mp.find(nums[i]) != mp.end()){  
 index.push\_back(i);  
 index.push\_back(mp[nums[i]]);  
 }  
 mp.insert(pair<int,int>(target - nums[i],i));  
 }  
 return index;  
}

分解回文串经验（**重做该题**）：

问是否存在某路径，考虑回溯算法；

手画出用例解决过程；

严格遵照模板格式；

尽量**按字符串为单元**进行处理，采用分割的想法，不要以字符为单元。

bool isPalindrome(string str,int l, int r) {  
 while (l < r){  
 if (str[l] != str[r]){  
 return false;  
 }  
 l++;  
 r--;  
 }  
 return true;  
}  
  
void backtrack(string s, int start, vector<string> &track){  
 if ( start == s.size()){  
 res.push\_back(track);  
 return;  
 }  
 for (int i = start; i < s.size(); ++i) {  
 if (isPalindrome(s, start, i)){  
 track.push\_back(s.substr(start,i - start + 1));  
 backtrack(s, i + 1,track);  
 track.pop\_back();  
 } else{  
 continue;  
 }  
 }  
 return;  
}  
  
vector<vector<string>> partition(string s) {  
 vector<string> track;  
 backtrack(s, 0, track);  
 return res;  
}

注意：sort用法

数组：sort(arr, arr+n);

向量：sort(vec.begin(), vec.end());

substr用法：substr(index,length);

找到K个最接近的元素 经验：

首先在搜索最接近目标的元素时，应考虑到用二分搜索替代线性搜索。

注意：

查找最接近目标的元素应该用查找左侧边界。

删除重复的数组元素 成功经验：

1. 读懂题目需求，抄下题目用例，思考与已学题型的联系，写下模板，尝试以穷举和程序化的思维手算解决，在解决用例的过程中应该列举不同的情况和不同处理方式，方便后续参考；
2. 把手算过程整理成中文草稿，应包括设置变量的初值及含义，循环结束条件，条件判断，不同情况下不同的处理等。
3. 根据自然语言写下代码，结合用例思考每一句是否正确，并且写完一段就要注意手算或者调试验证；
4. 最后用之前的用例在IDE中调试。

下雨问题 经验：

注意：函数体内不要轻易直接修改传递给该函数的变量，容易出错。

应重点做思维训练，这题蕴藏的思想即边扫描边更新关键变量，把各种情况囊括到一个更宽泛而恰当的判断。

组织行程 经验：

模块化测试代码时，注意所写函数必须返回某个值，否则直接测试会出现调用错误。

注意回溯算法中尽量不要掺杂二分查找.

注意严格遵守回溯算法模板，尤其不能去掉循环或者替换为二分查找。

回溯算法中vis最好设置为unordered\_set<int> vis;

最大的经验是：不要再盲目挑战HARD，当超过一个小时还是没有明显进展，建议放弃，性价比过低了。