1.题目：编程实现书P32 ADT Stack 基本操作9个，用顺序存储结构实现；

源代码：

#include<iostream>

#define TYPE int

#define INITSIZE 100

using namespace std;

class SqStack

{

public:

TYPE \* top;

TYPE \* base;

int stacksize;

};

//初始化函数

void InitStack(SqStack &S)

{

S.base=new TYPE[INITSIZE];

if(!S.base)

{

cout<<"初始化失败"<<endl;

return;

}

else

{

S.top=S.base;

S.stacksize=INITSIZE;

cout<<"初始化成功"<<endl;

}

}

//判空函数

bool StackEmpty(SqStack &S)

{

if(S.top==S.base)

{

return true;

}

else

{

return false;

}

}

//求元素个数函数

int GetLength(SqStack &S)

{

return S.top-S.base;

}

//栈清空函数

void ClearStack(SqStack &S)

{

if(S.base)//如果栈存在

{

S.top=S.base;//逻辑清空

}

else

{

cout<<"该栈不存在"<<endl;

}

}

//栈销毁函数

void DestroyStack(SqStack &S)

{

if(S.base)

{

delete S.base;

S.stacksize=0;

S.top=NULL;

S.base=NULL;//销毁空间后记得将指针也置空

}

else

{

cout<<"该栈不存在"<<endl;

}

}

//元素入栈函数

void Push(SqStack &S,TYPE e)

{

if(S.top-S.base==S.stacksize)

{

TYPE \*newbase=new TYPE[S.stacksize+10];

for(int i=0;i<S.stacksize;i++)

{

newbase[i]=S.base[i];

}

delete S.base;

S.base=newbase;

S.top=&(S.base[S.stacksize-1]);

S.top++;

}//申请 拷贝 删除

else

{

\*S.top=e;

S.top++;

}

}

//元素出栈函数 e接收出栈值

void Pop(SqStack &S,int &e)

{

if(S.top==S.base)//判空

{

cout<<"栈已为空 无法出栈"<<endl;

return;

}

else

{

S.top--;//逻辑删除

e=\*S.top;

}

}

//从底到顶遍历栈元素函数

void StackTraverse(SqStack &S)

{

TYPE \*iter=S.base;

while(iter!=S.top)

{

cout<<\*iter<<" ";

iter++;

}

cout<<endl;

}

//取栈顶元素函数

void GetTop(SqStack &S,TYPE &e)

{

if(S.top==S.base)

{

cout<<"栈为空 无法返回栈顶元素"<<endl;

return;

}

S.top--;

e=\*S.top;

S.top++;

}

int main()

{

SqStack S;

return 0;

}

2.题目：编程实现书P48 ADT Queue 基本操作9个，用链式存储结构实现；

源代码：

#include<iostream>

#define TYPE int

using namespace std;

class QueneNode//队列节点类定义

{

public:

TYPE data;

QueneNode \*next;

};

class LinkQuene//链式队列声明

{

public:

QueneNode \*front;

QueneNode \*rear;

};

//初始化函数

void InitQuene(LinkQuene &Q)

{

Q.front=Q.rear=new QueneNode;

if(!Q.front)

{

cout<<"初始化失败"<<endl;

}

else

{

Q.front->next=NULL;

cout<<"初始化成功"<<endl;

}

}

//销毁函数 算法：依次释放所有节点

void DestroyQuene(LinkQuene &Q)

{

while(Q.front)

{

QueneNode \*p=Q.front->next;

delete Q.front;

Q.front=p;

}

cout<<"队列销毁成功"<<endl;

}

//清空函数

void ClearQuene(LinkQuene &Q)

{

if(Q.front)

{

Q.rear=Q.front;

}

else

{

cout<<"队列不存在"<<endl;

return;

}

}

//判空函数

bool QueneEmpty(LinkQuene &Q)

{

if(Q.front==Q.rear)

{

return true;

}

else

{

return false;

}

}

//入队函数

void EnQuene(LinkQuene &Q,TYPE e)

{

QueneNode \*p=new QueneNode;

p->data=e;

p->next=NULL;

Q.rear->next=p;

Q.rear=p;

}

//出队函数 注意只能是对头元素出队 e接收删除值

void DeQuene(LinkQuene &Q,TYPE &e)

{

if(Q.front==Q.rear)

{

cout<<"队列为空 无法出队"<<endl;

return;

}

QueneNode \*p=Q.front->next;

e=p->data;

Q.front->next=p->next;

if(Q.rear==p)

{

Q.rear=Q.front;//若出队节点已经是队列的尾节点 还需要将尾指针指向头指针以示空队列

}

delete p;

}

//取头元素函数

void GetHead(LinkQuene &Q,TYPE &e)

{

if(Q.front==Q.rear)

{

cout<<"队列为空 无法取头元素"<<endl;

return;

}

e=Q.front->next->data;

}

int QueneLength(LinkQuene &Q)

{

return Q.rear-Q.front;

}

void QueneTraverse(LinkQuene &Q)

{

QueneNode \*p=Q.front->next;

while(p)

{

cout<<p->data<<" ";

p=p->next;

}

cout<<endl;

}

int main()

{

LinkQuene Q;

return 0;

}

3.题目：八皇后问题求解

算法：逐行放置皇后 对每一行中的所有列一次检索 检查改列是否满足放置规则。若满足，则将皇后入栈，进行下一行的操作；若不满足，则将检查的列数加一。当改行所有列均不满足时，将栈中皇后出栈，检查列数加一继续检索。

源代码：

* 非栈递归实现

//T(O)=N!

//S(O)=N

#include<iostream>

#include<vector>

#include<cmath>

using namespace std;

//判断是否为不攻击位函数 返回值为布尔类型

bool IsNonAttackingPlacements(int row,int col,vector<int> &columnplacements)

{

for(int previousrow=0;previousrow<row;previousrow++)//遍历前行 进行比对

{

int columnToCheck=columnplacements[previousrow];

bool samecolumn=columnToCheck==col;

bool onDiagonal=abs(columnToCheck-col)==row-previousrow;//由于已知大小关系所以不需要加绝对值

//判断是否在对角线上的算法是判断行数差的绝对值是否等于列数差的绝对值

if(samecolumn||onDiagonal)

{

return false;

}

}

return true;//若前面都没有返回 则不为攻击位 可以返回true

}

//求解个数函数 返回值为解的个数

int GetNumberOfNonAttackingQueens(int row,vector<int> &columnplacements,int &boardsize)//递归求解函数

{

int validplacements=0;//计数器 最后作为返回值返回找到的方法个数 记得要初始化为0！

//递归结束条件-> row到达boardsize 这意味着走到了最后 找到了一个解

if(row==boardsize)

{

return 1;

}

for(int col=0;col<boardsize;col++)

{

if(IsNonAttackingPlacements(row,col,columnplacements))

{

columnplacements[row]=col;//如果满足不攻击 则将该位置计入 其逻辑类似于入栈

validplacements+=GetNumberOfNonAttackingQueens(row+1, columnplacements, boardsize);

}

}

return validplacements;

}

//输出放置位函数

void GetPlacements(vector<int> &columnplacements,int n,int count)

{

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

{

cout<<"第"<<i+1<<"种解为："<<endl;

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

{

cout<<row<<" "<<columnplacements[row]<<endl;

}

cout<<endl;

}

}

//求解函数 n表示皇后的个数

int NonAttackingQueens(int n)

{

vector<int> columnplacements;//逐行判断位置可否放置皇后 从而避免了对行参数的存储 只需要存储列参数

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

{

columnplacements.push\_back(0);

}

int count=GetNumberOfNonAttackingQueens(0,columnplacements,n);//返回方法个数 从第0行开始操作

GetPlacements(columnplacements, n, count);

return count;

}

int main()

{

cout<<NonAttackingQueens(4)<<endl;

return 0;

}

* 栈非递归实现

#include<iostream>

#include<cmath>

using namespace std;

//皇后类定义

class Queen

{

public:

//构造函数 便于初始化

Queen(int row,int col)

{

this->row=row;

this->col=col;

}

//重载空实现构造函数

Queen()

{

}

int row;

int col;

};

//栈定义

class Stack

{

public:

Queen \*base;

Queen \*top;

void push(Queen &q)

{

\*(this->top)=q;

this->top++;

}

//出栈函数 返回被出栈节点的列

int pop()

{

if(!isEmpty())

{

(this->top)--;

return this->top->col;

}

else

{

return -1;

}

}

bool isEmpty()

{

if(this->base==this->top)

return true;

else

return false;

}

void stackTraverse()

{

Queen \*p=this->base;

while(p!=this->top)

{

cout<<(\*p).row<<" "<<(\*p).col<<endl;

p++;

}

cout<<endl;

}

};

//栈初始化函数

void initStack(Stack &S,int n)

{

S.base=new Queen[n];

S.top=S.base;

}

//判是否为攻击位函数

bool isNotAttacking(Stack &S,int rowToCheck,int colToCheck)

{

Queen \*p=S.base;

while(p!=S.top)

{

//比较是否为同列或在对角线上即可

bool samecolumn=colToCheck==p->col;

bool ondiagonal=abs(colToCheck-p->col)==rowToCheck-p->row;

if(samecolumn||ondiagonal)

{

return false;

}

p++;

}

return true;

}

//求解函数

int nonAttackingQueens(int n)

{

Stack S;

initStack(S, n);

Queen firstqueen(0,0);//初始化 第一个皇后定为第一行第一列

//皇后入栈

S.push(firstqueen);

//设定检索量 从第二行第一列开始

int rowToCheck=1;

int colToCheck=0;

//计数器

int count=0;

//结束条件为第一个皇后的col已经到达n

while(1)

{

if(isNotAttacking(S,rowToCheck,colToCheck))//如果不是攻击位

{

//该点入栈

Queen newqueen(rowToCheck,colToCheck);

S.push(newqueen);

//从下一行的第一列开始遍历

rowToCheck=rowToCheck+1;

colToCheck=0;

}

else//如果是攻击位

{

//列数+1

colToCheck++;

if(colToCheck==n)//如果已达到n 应回溯至上一行

{

//首先出栈 并用某量存储被出节点的列

int problemCol=S.pop();

//检索量定位至被出节点的行与下一列

rowToCheck--;

colToCheck=problemCol+1;

//若回溯后上一行的col也已达到n 则继续回溯 再回溯则不可能为col 因为已经是放置好了的皇后 不可能出现col相同的情况

if(colToCheck==n)

{

problemCol=S.pop();

//如果本次出栈时返回了空栈标志 则说明第一个皇后也已经到达了最后一列 从而说明遍历结束 循环结束

if(problemCol==-1)

{

delete S.base;

return count;

break;

}

rowToCheck--;

colToCheck=problemCol+1;

}

}

}

//如果row已经到达n 说明找到了一组解

if(rowToCheck==n)

{

count++;

cout<<"找到第"<<count<<"个答案"<<endl;

//首先输出找到的解

S.stackTraverse();

//然后出栈 将出栈节点col+1 然后检索行-1 继续判断

int okCol=S.pop();

colToCheck=okCol+1;

rowToCheck--;

//如果+1后已经到达n 则需要回溯 此次回溯后不可能出现上一皇后col也达到n的情况 因为前面的皇后是已经放置好了的 不可能出现相同的col

if(colToCheck==n)

{

okCol=S.pop();//此为倒数第二个queen的col

rowToCheck--;

colToCheck=okCol+1;

}

}

}

}

int main()

{

cout<<nonAttackingQueens(8)<<endl;

return 0;

}

4.题目：顿顿评估了m位同学上学期的安全指数，其中第i（1≤i≤m）位同学的安全指数为yi，是一个[0,108]范围内的整数；同时，该同学上学期的挂科情况记作resulti∈0,1，其中0表示挂科、1表示未挂科。相应地，顿顿用predictθ(y)表示根据阈值θ将安全指数y转化为的具体预测结果。如果predictθ(yj)与resultj相同，则说明阈值为θ时顿顿对第j位同学是否挂科预测正确；不同则说明预测错误。



最后，顿顿设计了如下公式来计算最佳阈值θ＊：



该公式亦可等价地表述为如下规则：

1.  最佳阈值仅在yi中选取，即与某位同学的安全指数相同；

2.  按照该阈值对这m位同学上学期的挂科情况进行预测，预测正确的次数最多（即准确率最高）；

3.  多个阈值均可以达到最高准确率时，选取其中最大的。

算法：利用前缀和降低时间复杂度。首先对数据按照安全指数升序排序，然后对安全指数去重处理。对索引为i的第一次出现的安全指数，sum[m]-sum[i-1]即为正确匹配的1的个数，i-1-sum[i-1]即为正确匹配的0的个数。典型的牺牲空间换时间的算法。

源代码：

//暴力算法的时间复杂度为m\*m 运用前缀和可以将算法复杂度减至m

//前缀和很适合用来给需要多次遍历计数的算法降低时间复杂度

//v的存储下标要大改! 变成下标为1开始存储数据 前面浪费一个空间 更方便

#include<iostream>

#include<vector>

#include<set>

#include<algorithm>

using namespace std;

bool cmp(pair<int, int> &p1,pair<int, int> &p2)

{

return p1.first<p2.first;

}

int main()

{

int m;

cin>>m;

vector<pair<int,int>> v;

pair<int,int> p0=make\_pair(-1,-1);

v.push\_back(p0);//用（-1，-1）占位 从而使数据下标从1开始

int sum[m+1];//前缀和数组

for(int i=0;i<m+1;i++)

{

sum[i]=0;

}//前缀和数组初始化

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

{

int y;//安全指数

int result;//挂科结果

cin>>y>>result;

pair<int, int>p=make\_pair(y, result);

v.push\_back(p);

}

//首先按照安全指数进行升序排序

sort(v.begin(),v.end(),cmp);

//求前缀和数组 注意一定要在排序之后进行此操作

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

{

sum[i]=sum[i-1]+v[i].second;

}

set<int> s;//利用集合进行去重 注意可以利用集合中是默认升序排序的

int ans=0;//记录答案阈值

int maxright=0;//记录过程中产生的预测正确结果数

//遍历集合 逐一求解

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

{

int a=v[i].first;//获取安全指数

if(s.count(a))

{

continue;

}//注意此处由于遇到重复直接跳过 所以安全指数所有重复值中 只会取挂科为0的第一个 从而能够保证之后的算法不会漏或多出正确匹配

s.insert(a);//若集合中没有a 说明没有对a进行过统计 将a加入集合中 避免a以后重复统计

int right1=sum[m]-sum[i-1];//正确匹配的1的个数

int right0=i-1-sum[i-1];//正确匹配的0的个数

int right=right1+right0;

if(right>=maxright)

{

maxright=right;

ans=a;

}

}

cout<<ans<<endl;

return 0;

}

5.题目：某次测验后，顿顿老师在黑板上留下了一串数字23333便飘然而去。凝望着这个神秘数字，小P同学不禁陷入了沉思……

已知某次测验包含n道单项选择题，其中第i题（1≤i≤n）有ai个选项，正确选项为bi，满足ai≥2且0≤bi＜ai。比如说，ai=4表示第i题有4个选项，此时正确选项bi的取值一定是0、1、2、3其中之一。

顿顿老师设计了如下方式对正确答案进行编码，使得仅用一个整数m便可表示b1,b2,…,bn。

首先定义一个辅助数组ci，表示数组ai的前缀乘积。当1≤i≤n时，满足：



特别地，定义c0=1。

于是m便可按照如下公式算出：



易知，0≤m＜cn，最小值和最大值分别当bi全部为0和bi=ai-1时取得。

试帮助小P同学，把测验的正确答案b1,b2,…,bn从顿顿老师留下的神秘整数m中恢复出来。

算法：按照题目给出提示进行操作即可 无特殊算法

源代码：

#include<iostream>

using namespace std;

int main()

{

int n;//题目个数

int m;//神秘数字

cin>>n>>m;

int a[n+1];

a[0]=0;//由于题目要求下标i从1开始 所以将a[0]无效化

//一次输入每题选项个数

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

{

cin>>a[i];

}

int c[n+1];

c[0]=1;//特别地 定义c[0]为1

//二重循环求ci

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

{

c[i]=c[i-1]\*a[i];

}

int b[n+1];

b[0]=-1;//无效化

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

{

b[i]=(m%c[i]-m%c[i-1])/c[i-1];

cout<<b[i]<<" ";

}

return 0;

}