## **试推导求解n阶梵塔问题至少要执行的移动操作move次数。**

#include <iostream>

using namespace std;

int hanoiMove(int n)

{

return (n == 1) ? 1 : 2 \* hanoiMove(n - 1) + 1;

}

int main()

{

int n;

cout << "请输入求解的最大盘子个数：";

cin >> n;

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

{

int move = hanoiMove(i);

cout << i << "个盘子最少需要移动" << move << "次" << endl;

}

return 0;

}

**运行该程序，输出结果如下：**

**请输入求解的最大盘子个数：5**

**1个盘子最少需要移动1次**

**2个盘子最少需要移动3次**

**3个盘子最少需要移动7次**

**4个盘子最少需要移动15次**

**5个盘子最少需要移动31次**

**hanoiMove(n)={**

**1 (n=1)**

**2 \* hanoiMove(n - 1) + 1 (n>1)**

**}**

**可推出，n阶hanoi问题至少要执行的移动次数为2^n-1次。 // 推导过程**

## 一个简化的背包问题：一个背包能装总重量为 T，现有 n 个物件，其重量分别为（W1、W2、…、Wn）。问能否从这 n 个物件中挑选若干个物件放入背包中，使其总重量正好为 T ？若有解则给出全部解，否则输出无解。

#include <iostream>

#define MAX\_ITEMS\_NUM 110

using namespace std;

/\*\*

\* 背包问题求解函数

\* @param targetWeight 目标重量

\* @param itemsNums 物品数量

\* @param itemsWeight 物品重量数组

\*/

void chooseItems(int totalWeight, int itemsNums, int \*itemsWeight)

{

/\*

dp[i][j]表示前i个物品，能否凑出j的重量:

无法凑出，0

可以凑出但没有第i个物品，1

可以凑出且有第i个物品，2

\*/

int dp[itemsNums + 1][totalWeight + 1];

// 填写dp表格：第一行、列

for (int j = 0; j < totalWeight + 1; j++) // 第一行为0：前0个物品无法凑出任何重量

dp[0][j] = 0;

for (int i = 0; i < itemsNums + 1; i++) // 第一列为1：任选物品都能凑出重量

dp[i][0] = 1;

// 填写dp表格：剩余行、列

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

{

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

{

if (itemsWeight[i] > j) // 物品无法放入背包，因此dp[i][j]取决于dp[i-1][j]

dp[i][j] = (dp[i - 1][j] == 0) ? 0 : 1;

else // 物品可能可以放进背包

{

if (dp[i - 1][j - itemsWeight[i]] != 0) // 物品可以放入背包

dp[i][j] = 2;

else // 物品仍无法放入背包

dp[i][j] = (dp[i - 1][j] == 0) ? 0 : 1;

}

}

}

// 分析dp表格，得到所有解

if (dp[itemsNums][totalWeight] == 0)

cout << "无解" << endl;

else

{

cout << "全部解：" << endl;

for (int i = itemsNums; dp[i][totalWeight] != 0; i--) // 不同的解

{

if (dp[i][totalWeight] == 2)

{

for (int row = i, col = totalWeight; row && col; row--) // 同一个解里的数据

{

if (dp[row][col] == 2)

{

cout << itemsWeight[row] << " ";

col -= itemsWeight[row];

}

}

cout << endl;

}

}

}

return;

}

int main()

{

int totalWeight, itemsNums, itemsWeight[MAX\_ITEMS\_NUM] = {0};

cout << "请输入背包总重量：";

cin >> totalWeight;

cout << "请输入物品总数，不超过100：";

cin >> itemsNums;

cout << "请输入各个物品的重量：";

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

cin >> itemsWeight[i];

chooseItems(totalWeight, itemsNums, itemsWeight);

return 0;

}

**运行该程序，输出结果如下：**

**请输入背包总重量：10**

**请输入物品总数，不超过100：5**

**请输入各个物品的重量：2 3 5 7 9**

**全部解：**

**7 3**

**5 3 2**

## **已知以字符型顺序表表示的表达式含有三种扩号“（”、“）”、“[”、“]”、“{”和“}”，它们可嵌套使用，试写出算法判断给定表达式中所含扩号是否正确配对出现。**

#include <iostream>

using namespace std;

#define MAX\_SIZE 220

typedef struct

{

char data[MAX\_SIZE];

int size;

} Stack;

// 判断是否是空栈

bool isEmptyStack(Stack \*stack)

{

return (stack->size == -1) ? true : false;

}

// 元素入栈

void push(Stack \*stack, char value)

{

stack->data[++stack->size] = value;

return;

}

// 元素出栈

void pop(Stack \*stack)

{

stack->size--;

return;

}

// 获取栈顶元素

char getTop(Stack \*stack)

{

return stack->data[stack->size];

}

// 判断两个括号是否匹配

bool isMatch(char front, char back)

{

if ((front == '(' && back == ')') ||

(front == '[' && back == ']') ||

(front == '{' && back == '}'))

return true;

else

return false;

}

// 判断表达式中的括号是否正确配对

bool isCorrectPairing(char \*exp)

{

Stack bracket;

bracket.size = -1;

for (int i = 0; exp[i]; i++) // 扫描表达式

{

// 扫描到左括号，入栈

if (exp[i] == '(' || exp[i] == '[' || exp[i] == '{')

push(&bracket, exp[i]);

// 扫描到右括号，配对

else if (exp[i] == ')' || exp[i] == ']' || exp[i] == '}')

{

if (isEmptyStack(&bracket) || !isMatch(getTop(&bracket), exp[i]))

return 0; // 括号不配对（包括栈为空）

else

pop(&bracket); // 括号配对，出栈

}

}

return isEmptyStack(&bracket); // 栈为空，则表达式括号正确配对

}

int main()

{

char exp[MAX\_SIZE];

cout << "请输入表达式，最大长度为200: ";

fgets(exp, MAX\_SIZE, stdin);

if (isCorrectPairing(exp))

cout << "正确配对\n";

else

cout << "非正确配对\n";

return 0;

}

**运行该程序，输出结果如下：**

**请输入表达式，最大长度为200: {2+8\*2\*[10/(5+5)]}**

**正确配对**

## **若栈的输入序列为 1234，给出所有可能的输出序列。**

**如果4最先出栈——输出序列只能是4321；**

**如果3最先出栈，那么2一定在1之前输出，4一定在3之后输出——输出序列可能为3214、3241、3421；**

**如果2最先出栈，那么若1紧接着出栈——输出序列可能为2134、2143；若1暂时不出栈——输出序列可能为2314、2341、2431；**

**如果1最先出栈，那么输出序列只有423不可能——输出序列可能为1234、1243、1324、1342、1432**

**综上，所有可能的序列共14种，分别为：4321、3214、3241、3421、2134、2143、2314、2341、2431、1234、1243、1324、1342、1432**

## **已知有两个栈 S1 和 S2 及其基本操作 Push(S , x)、Pop(S)、Full(S) 和 Empty(S)，给出用此二栈实现队列操作Enqueue、Dequeue、Fullq 和Emptyq的算法。**

#include <iostream>

using namespace std;

#define MAX\_SIZE 100

typedef struct

{

int data[MAX\_SIZE];

int size;

} Stack;

// 判断栈是否为空

bool Empty(Stack \*stack)

{

return (stack->size <= -1) ? true : false;

}

// 判断栈是否为满

bool Full(Stack \*stack)

{

return (stack->size >= MAX\_SIZE - 1) ? true : false;

}

// 元素入栈

void Push(Stack \*stack, int x)

{

if (stack->size == MAX\_SIZE - 1) // 如果栈上溢

{

printf("Error: Stack Overflow\n");

exit(1);

}

stack->data[++stack->size] = x;

return;

}

// 元素出栈并返回

int Pop(Stack \*stack)

{

if (Empty(stack)) // 如果栈下溢

{

printf("Stack Underflow\n");

exit(1);

}

return stack->data[stack->size--];

}

// 入队，若成功返回true

bool Enqueue(Stack \*queue, int x)

{

if (Full(queue))

{

cout << "入队失败，队伍已满！" << endl;

return false;

}

else

Push(queue, x);

return true;

}

// 离队，若成功返回true

bool Dequeue(Stack \*queue, Stack \*s)

{

if (Empty(queue))

{

cout << "离队失败，队伍已空！" << endl;

return false;

}

else

{

while (!Empty(queue))

{

int value = Pop(queue);

Push(s, value);

}

Pop(s);

while (!Empty(s))

{

int value = Pop(s);

Push(queue, value);

}

}

return true;

}

// 判断队列是否为满

bool Fullq(Stack \*queue)

{

return (queue->size >= MAX\_SIZE - 1) ? true : false;

}

// 判断队列是否为空

bool Emptyq(Stack \*queue)

{

return (queue->size <= -1) ? true : false;

}

int main()

{

Stack queue, s1, s2;

queue.size = -1, s1.size = -1, s2.size = -1;

cout << "初始入队元素：1 2" << endl;

Enqueue(&queue, 1), Enqueue(&queue, 2);

cout << "当前队列元素为：";

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

cout << queue.data[i] << " ";

cout << endl;

// 测试 Enqueue

int value;

cout << "测试Enqueue。请输入新入队元素：" << endl;

cin >> value;

Enqueue(&queue, value);

cout << "当前队列元素为：";

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

cout << queue.data[i] << " ";

cout << endl;

// 测试 Dequeue

cout << "测试Dequeue。按下任意键，开始测试" << endl;

getchar(),getchar();

Dequeue(&queue, &s1); // 离队一个元素

cout << "当前队列元素为：";

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

cout << queue.data[i] << " ";

cout << endl;

// 测试 Fullq

cout << "测试Fullq。按下任意键，开始测试" << endl;

getchar();

cout << "当前队列是否满：";

cout << (Fullq(&queue) ? "是" : "否") << endl;

// 测试 Emptyq

cout << "测试Emptyq。按下任意键，开始测试" << endl;

getchar();

cout << "当前队列是否空：";

cout << (Emptyq(&queue) ? "是" : "否") << endl;

return 0;

}

**运行该程序，输出结果如下：**

**初始入队元素：1 2**

**当前队列元素为：1 2**

**测试Enqueue。请输入新入队元素：**

**3**

**当前队列元素为：1 2 3**

**测试Dequeue。按下任意键，开始测试**

**当前队列元素为：2 3**

**测试Fullq。按下任意键，开始测试**

**当前队列是否满：否**

**测试Emptyq。按下任意键，开始测试**

**当前队列是否空：否**

## **已知串s和t ,试求由所有包含在s中而不包含在t中的字符构成的新串r（字符不重复），以及r中每个字符在s中第一次出现的位置。利用Concat、Len、Sub和Equal四种基本运算。**

#include <iostream>

using namespace std;

#define MAX\_LEN 100

// 获取字符串R

void getStrR(char \*s, char \*t, char \*r)

{

for (int i = 0; t[i]; i++)

{

for (int j = 0; s[j]; j++)

{

if (t[i] == s[j])

s[j] = ' ';

}

}

for (int i = 0, j = 0; s[i]; i++)

{

if (s[i] != ' ')

r[j++] = s[i];

}

return;

}

// 输出r的每个字符在s第一次出现的位置

void printPosition(char \*s, char \*r)

{

for (int i = 0, j = 0; r[i]; j++)

{

if (r[i] == s[j])

{

cout << r[i] << ": " << j << endl;

i++;

}

}

return;

}

int main()

{

char s[MAX\_LEN] = {0}, t[MAX\_LEN] = {0}, r[MAX\_LEN] = {0};

cout << "请输入字符串 s：";

cin >> s;

cout << "请输入字符串 t：";

cin >> t;

getStrR(s, t, r);

cout << "新串r为：" << r << endl;

cout << "r中每个字符在s中第一次出现的位置：" << endl;

printPosition(s, r);

return 0;

}

**运行该程序，输出结果如下：**

**请输入字符串 s：excuseme**

**请输入字符串 t：me**

**新串r为：xcus**

**r中每个字符在s中第一次出现的位置：**

**x: 1**

**c: 2**

**u: 3**

**s: 4**