程序设计课后练习

1. 欧几里得算法
2. 算法描述

用于求两不全为0的非负整数之间的最大公约数，首先判断除数是否为0，如果为0直接返回被除数作为两数之间最大公约数。如果不为0，则进行除余计算，同时将除数赋予被除数，余数赋予除数继续进行除余计算直到余数为零，并将此时的被除数作为两数之间的最大公约数输出。

二．伪代码：

Euclid(m,n)

//使用欧几里得算法计算gcd(m,n)

//输入：两个不全为0的非负整数m,n

//输出：m,n的最大公约数

While n!=0 do

r<-m mod n

m<- n

n<-r

return m

代码：

#include<stdio.h>

int main()

{

int m, n;

printf("输入两整数：");

scanf\_s("%d%d", &m, &n);//键盘输入进行判断两个不全为0的非负整数

int r;//定义余数

while (n != 0)//判断除数是否为0

{

r = m % n;//除余运算，余数赋予r

m = n;//除数赋予被除数

r = n;//余数赋予除数

if (n == 0)//如果除数为零跳出循环

{

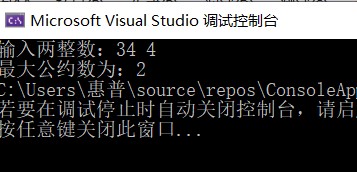
break;

}

}

printf("最大公约数为：%d",m);//输出最大公约数

结果：



算法分析：

此算法为计算两不全为0的非负整数之间的最大公约数。辗转相除法， 又名欧几里德算法，是求最大公约数的一种方法。它的具体做法是：用较大数除以较小数，再用出现的余数（第一余数）去除除数，再用出现的余数（第二余数）去除第一余数，如此反复，直到最后余数是0为止。如果是求两个数的最大公约数，那么最后的除数就是这两个数的最大公约数。

1. 埃拉托色尼筛选法
2. 算法描述：

埃拉托色尼筛选法简称埃氏筛法，是古希腊数学家埃拉托色尼(提出的一种筛选法。 是针对自然数列中的自然数而实施的，用于求一定范围内的质数.

（1）先把1删除（现今数学界1既不是质数也不是合数）

（2）读取队列中当前最小的数2，然后把2的倍数删去

（3）读取队列中当前最小的数3，然后把3的倍数删去

（4）读取队列中当前最小的数5，然后把5的倍数删去

（5）如上所述直到需求的范围内所有的数均删除或读取

二．伪代码：

Sieve(n)

//实现“埃拉托色尼筛选法”

//输入：一个正整数n>1

//输出：包括所有小于等于n的质数的数组l

For p<-2 to n do A[p]<-p

For p<- 2 to [n] do

If A[p]!=0

J<- p\*p

While j<=n do

A[j]<-0

J<-j+p

//将A中剩余的元素复制到质数数组L中

I<-0

For p<-2 to n do

If A[p]!=0

L[i]<-A[p]

I<-i+1

Return L

代码

#include <cstdio>

#define N 1000

int main()

{

int n;

scanf\_s("%d", &n);

//定义两个变量i,j,一个数组a[N]

int i, j, a[N];

//将数组初始化为1,a[i] = 1表示i为素数

//初始时默认从2到N-1均为素数

for (i = 2; i < n; i++)

{

a[i] = 1;

}

//遍历数组找出下标为素数的,并将所有下标为该素数的倍数的值改为0

for (i = 2; i < n; i++)

{

if (a[i])

{

for (j = i; i \* j < n; j++)

{

a[i \* j] = 0;

}

}

}

//输出2~N范围内的素数

for (i = 2; i < n; i++)

{

if (a[i])

{

printf("%4d", i);

}

}

return 0;

}

结果



1. 递归计算n!
2. 算法描述：

通过递归调用函数(n-1) \* n完成任意非负整数的阶乘。

二．伪代码：

F(n)

//递归计算n!

//输入：非负整数n

//输出：n!的值

If n=0 return 1

Else return F(n-1)\*n

三．代码：

#include<stdio.h>

int recursion(int n) {

if (n == 0)

return 1;

else

return recursion(n-1) \* n;

}

void main() {

int n;

scanf\_s("%d", &n);

int a = recursion(n);

printf("n!=%d", a);

}

四．结果



1. 计算第n个斐波那契数
2. 算法描述：

斐波那契数列是一个著名的数列：

0，1，1，3，5，8，13，21，34，…

这个数列可以用一个简单的递推式和两个初始条件来定义。

在此，我们可以使用递归的方法来进行求解:

二．伪代码：

F(n)

//根据定义，递归计算第n个斐波那契数

//输入：一个非负整数

//输出：第n个斐波那契数

If n<=1 return n

Else return F(n-1)+F(n-2)

三．代码：

#include<stdio.h>

int Fibonacci(int n) {

if (n <= 1) {

return n;

}

else

return Fibonacci(n - 1) + Fibonacci(n - 2);

}

void main() {

int n;

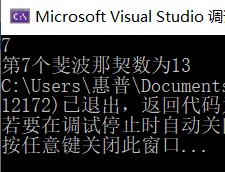
scanf\_s("%d", &n);

int a = Fibonacci(n);

printf("第%d个斐波那契数为%d", n,a);

}

1. 结果



1. 选择排序
2. 算法描述：

二．伪代码：

SelectionSort(A[0…n-1])

//该算法用选择排序对给定的数组排序

//输入：一个可排序的数组A[0…n-1]

//输出：升序排列的数组A[0…n-1]

For i<-0 to n-2 do

Min <- i

For j<-i+1 to n-1 do

If A[j]<A[min] min<-j

Swap A[i] and A[min]

三．代码：

#include<stdio.h>

void choice(int\* a, int n)

{

int i, j, temp;

for (i = 0; i < n - 1; i++)

{

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

{

if (a[i] > a[j])

{

temp = a[i];

a[i] = a[j];

a[j] = temp;

}

}

}

}

int main(int argc, char\* argv[])

{

int i;

int a[10] = { 2,4,7,1,6,9,8,3,0,5 };

printf("输出原数列：");

for (i = 0; i < 10; i++)

{

printf("%2d", a[i]);

}

printf("\n");

choice(a, 10);

printf("\n");

printf("输出排序序列：");

for (i = 0; i < 10; i++)

{

printf("%2d", a[i]);

}

printf("\n");

return 0;

}

四．结果：



1. 冒泡排序
2. 算法描述：

二．伪代码：

BubbleSort(A[0…-1])

//该算法用冒泡排序对数组A[0…n-1]进行排序

//输入：一个可排序数组A[0…n-1]

//输出：非降序排列的数组A[0…n-1]

For i<- 0 to n-2 do

For j<- 0 to n-2-i do

If A[j+1]<A[j] swap A[j]andA[j+1]

三．代码：

#include<stdio.h>

void bubble(int\* a,int n)

{

int i, j, temp;

for (i = 0; i < n - 1; i++)

{

for (j = 0; j < n - 1; j++)

{

if (a[j] > a[j + 1])

{

temp = a[j];

a[j] = a[j + 1];

a[j + 1] = temp;

}

}

}

}

int main(int argc, char\* argv)

{

int a[10] = { 3,4,5,2,1,0,6,7,8,9};

int i;

printf("输出原数列为：");

for (i = 0; i < 10; i++)

{

printf("%2d", a[i]);

}

printf("\n");

bubble(a, 10);

printf("排序数列为：");

for (i = 0; i < 10; i++)

{

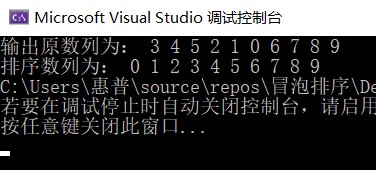
printf("%2d", a[i]);

}

return 0;

}

四．结果：



7.选择排序

1. 算法描述：

二．伪代码：

sequentialSearch2(A[0…n],k)

//顺序查找的算法实现，它用了查找键来作限位器

//输入：一个n个元素的数组A和一个查找键k

//输出：第一个值等于k的元素位置，如果找不到这样的元素，返回 -1

A[n]<-K

i<-0

while A[i]!=K do

i<-i+1

if i<n return i

else return -1

三．代码：

#include<stdio.h>

#define L 1000

void main() {

int a[L];

int n;

int c;

int b;

printf("c=");

scanf\_s("%d", &c);

printf("请输入数列长度：");

scanf\_s("%d",&n);

for ( b = 0; b < n; b++) {

scanf\_s("%d",&a[b]);

}

for ( b = 0; b < n; b++) {

printf("%d ", a[b]);

}

for (b = 0; b < n; b++) {

if (a[b] == c) {

printf("\n第一个和C相同的数是第%d个数",b+1);

}

}

}

1. 结果：

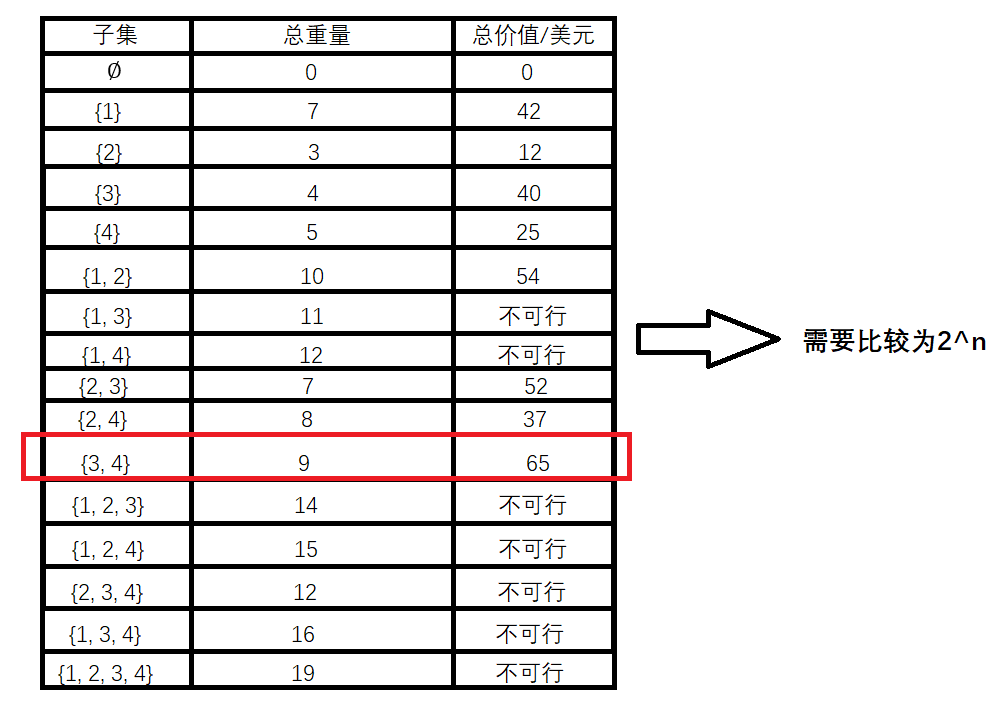


8.背包问题

一．算法描述：

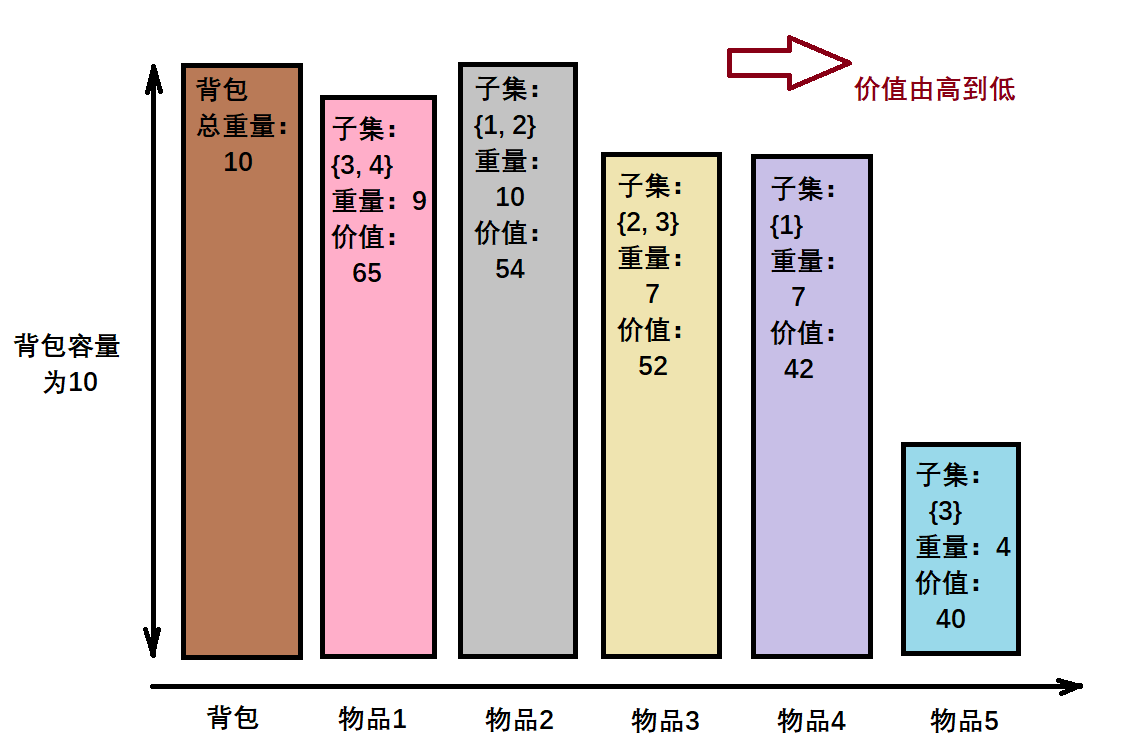
给定n个重量为，，…，，价值为，，…，的物品和一个承重为W的背包，求这些物品中一个最有价值的自己并且能够装到包中。

在这个问题中，我们可以利用穷举查找的方法需要考虑n个物品集合的所有子集，为了找出可行的子集（也就是说，总重量不能超过背包承重能力的子集），要计算每个子集的总重量。然后在他们中间找到价值最大的子集。



**（图一）利用穷举查找求得总价值最高的情况**

已知，有n个物品，若使用穷举的方法需要查找次（一个n元素的集合的子集数量）,所以无论生成独立子集的效率有多高，穷举算法都会导致一个Ω

二．伪代码

bt(List<List<Integer>> ans, List<Integer> tempList, int[] nums)

//求得一个集合的所有子集（一个集合中有n个元素那么它有2^n个子集）

//输入：List总集合，List子集，int数组（记录该物品的编号）

//无输出

list (new ArrayList<>)(new HashSet<>(tempList)))

if !ans.contains(list)

三．代码：

#include<iostream>

using namespace std;

#include <algorithm>

int w[5] = { 0 , 2 , 3 , 4 , 5 }; //商品的体积2、3、4、5

int v[5] = { 0 , 3 , 4 , 5 , 6 }; //商品的价值3、4、5、6

int bagV = 8; //背包大小

int dp[5][9] = { { 0 } }; //动态规划表

int item[5]; //最优解情况

void findMax() { //动态规划

for (int i = 1; i <= 4; i++) {

for (int j = 1; j <= bagV; j++) {

if (j < w[i])

dp[i][j] = dp[i - 1][j];

else

dp[i][j] = max(dp[i - 1][j], dp[i - 1][j - w[i]] + v[i]);

}

}

}

void findWhat(int i, int j) { //最优解情况

if (i >= 0) {

if (dp[i][j] == dp[i - 1][j]) {

item[i] = 0;//数组对应的位置置0作为标识符代表未选中

findWhat(i - 1, j);

}

else if (j - w[i] >= 0 && dp[i][j] == dp[i - 1][j - w[i]] + v[i]) {

item[i] = 1;//数组对应的位置置1作为标识符代表选中

findWhat(i - 1, j - w[i]);

}

}

}

void print() {

for (int i = 0; i < 5; i++) { //动态规划表输出

for (int j = 0; j < 9; j++) {

cout << dp[i][j] << ' ';

}

cout << endl;

}

cout << endl;

for (int i = 0; i < 5; i++) //最优解输出

cout << item[i] << ' ';

cout << endl;

}

int main()

{

findMax();

findWhat(4, 8);

print();

return 0;

}

四．结果：



9.插入排序

一．算法描述

插入排序的基本操作是“有序插入”，也就是将元素逐一插到有序序列中，保持序列有序，从而使有序序列的长度不断增加。对数组a[n]排序时，起初a[0]被认为是长度为1的有序子序列。然后，按照有序插入法,i从1到n-1循环地将a[i]插入到有序序列中。在插入a[i]前，数组a的前半段(a[0]至a[i-1])是有序段，后半段(a[i]~a[n-1])是停留于输入次序的无序段。插入a[i]使a[0]~a[i]有序。也就要为a[i[找到有序位置j(0<=j<=i)，将a[i]插在a[j]的位置上

二．伪代码

insertionSort(A[0…n-1])

//用插入排序对给定数组排序

//输入：n个可排序元素构成的一个数组A[0…n-1]

//输出：非降序排列的数组A[0…n-1]

For i<-1 to n-1 do

v<-A[i]

j<-i-1

while j>=0 and A[j]>v do

A[j+1]<-A[j]

j<-j-1

A[j+1]<-v

三．代码

#include<stdio.h>

int main() {

int a[7] = {89,45,68,90,29,34,17 };

int j,v,i;

for (i = 0; i < 7; i++)

{

printf("%4d", a[i]);

}

printf("\n");

for (i = 1; i <7 ; i++)

{

v = a[i];

j = i - 1;

while (j >= 0 && a[j]>v)

{

a[j + 1] = a[j];

j = j - 1;

}

a[j + 1] = v;

}

for (i = 0; i < 7; i++)

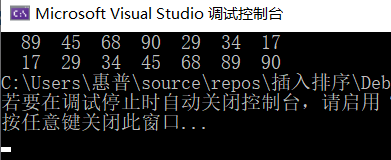
{

printf("%4d", a[i]);

}

}

四．结果



10.johnson-trotter算法

一．算法描述

二．伪代码

Johnson Ttotter(n)

//实现用来生成排列的Johnson Ttotter算法

//输入；一个正整数n

//输出：{1,….n}的所有排列的列表

将第一个排列初始化为12…n

While 存在一个移动元素do

求最大的移动元素k

把 k和他箭头指向的相邻元素互换

调转所有大于k的元素方向

将新排列添加到列表中

三．代码

#include<iostream>

#include<vector>

using namespace std;

struct jiedian

{

int zhi;

int dirct;

//dirct==0代表左边，dirct==1代表右边，

};

bool can\_move(jiedian a[], int len) {//判断是否存在可以移动的元素

vector<jiedian> move\_list;

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

{

if (a[i].dirct == 0)

{

if (i - 1 >= 0 && a[i - 1].zhi < a[i].zhi)

{

move\_list.push\_back(a[i]);

}

}

else

{

if (i + 1 <= len - 1 && a[i + 1].zhi < a[i].zhi)

{

move\_list.push\_back(a[i]);

}

}

}

if (move\_list.empty())

{

return false;

}

else return true;

}

vector<jiedian> search\_max\_index(jiedian a[], int len) {//求可以移动的最大元素

vector<jiedian> move\_list;

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

{

if (a[i].dirct == 0)

{

if (i - 1 >= 0 && a[i - 1].zhi < a[i].zhi)

{

move\_list.push\_back(a[i]);

}

}

else

{

if (i + 1 <= len - 1 && a[i + 1].zhi < a[i].zhi)

{

move\_list.push\_back(a[i]);

}

}

}

return move\_list;

}

void Johnson\_Trotter(jiedian a[], int n) {

//打印初始序列

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

{

cout << a[dayin].zhi;

}

cout << endl;

while (can\_move(a, n))//存在可移动元素即执行

{

vector<jiedian> s\_max = search\_max\_index(a, n);

int temp = s\_max[0].zhi;

for (int i = 0; i < s\_max.size(); i++)

{

if (s\_max[i].zhi > temp)

{

temp = s\_max[i].zhi;

}

}

for (int k = 0; k < n; k++)//找最大元素并且移动

{

if (a[k].zhi == temp)

{

if (a[k].dirct == 0)

{

jiedian temp1 = a[k];

a[k] = a[k - 1];

a[k - 1] = temp1;

break;

}

else {

jiedian temp1 = a[k];

a[k] = a[k + 1];

a[k + 1] = temp1;

break;

}

}

}

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

if (a[j].zhi > temp)

{

if (a[j].dirct == 1)

{

a[j].dirct = 0;

}

else a[j].dirct = 1;

}

}

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

{

cout << a[dayin].zhi;

}

cout << endl;

}

}

int main()

{

jiedian a[3];

for (int i = 0; i < 3; i++)

{

a[i].zhi = i + 1;

a[i].dirct = 0;

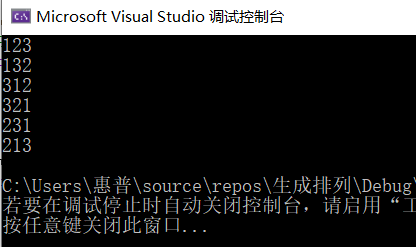
}

Johnson\_Trotter(a, 3);

return 0;

}

四．结果



11.LexicographicPermute算法

一．算法描述

二．伪代码

算法 LexicographicPermute(n)

//以字典序生成排列

//输入：一个正整数n

//输出：{1,2,...,n}的所有排列的列表

初始化第一个排列为1,2,...,n

while 最后一个排列有两个连续升序的元素 do

找出使得a(i)<a(i+1)的最大的i //a(i+1)>a(i+2)>...>a(n)

找到使得a(i)<a(j)的最大索引j //j>=i+1,因为a(i)<a(i+1)

交换a(i)和a(j) //a(i+1)到a(n)仍保持降序

将a(i+1)到a(n)的元素反序

将这个新排列添加到列表中

三．代码

#include<cstdio>

#include<iostream>

#include<cstring>

using namespace std;

void LexicographicPermute(int\* a, int n)

{

//开始时数组a[0,1,2,3，……,n-1]的序列时为1,2,3,4,5……n

int k = n - 2, j;

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

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

cout << endl;

while (1)

{

for (j = n - 1; j > k; j--)

{

if (a[j] > a[k])

{

swap(a[j], a[k]);

break;

}

}

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

swap(a[i], a[n - i + k]);

for (k = n - 1; k > 0; k--)

if (a[k] > a[k - 1])

break;

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

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

cout << endl;

if (k-- == 0) break;

}

}

int main()

{

int a[1000], n;

cin >> n;

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

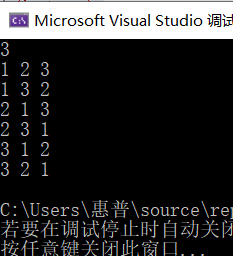
a[i] = i + 1;

LexicographicPermute(a, n);

return 0;

}

四．结果



12.折半查找

一．算法分析

给定已 按升序排好序的n个元素a[0:n-1]，现要在这n个元素中找出一特定元素x

比较x和 a的 中间元素 a[mid]，

若x=a[mid]，则x在L中的位置就是mid；

如果x<a[mid]，则x在a[mid]的前面；

如果x>a[mid]，则x在a[mid]的后面。

无论在哪部分查找x，其方法都和在a中查找x一样，只不过是查找的规模缩小了。

二．伪代码

BinarySearch(A[0….n-1],k)

//实现非递归的折半查找

//输入：一个升序数组A[0…n-1]和一个查找键k

//输出：一个数组元素的下标，该元素等于k；如果没有这样一个元素，则返回-1

L <- 0;r <- n-1

While l<=r do

M <- (l+r)/2

If k=A[m] return m

Else if k<A[m] r <- m-1

Else l<- m+1

Return -1

三．代码

#include<stdio.h>

int main()

{

int a[13] = { 3,14,27,31,39,42,55,70,74,81,85,93,98 };

int l = 0,r=12,m,k,i;

printf("输出数列：");

for (i = 0; i < 13; i++)

{

printf("%4d", a[i]);

}

printf("\n");

printf("请输入你要查询的数：");

scanf\_s("%d",&k);

while (l <= r)

{

m = (l + r) / 2;

if (k == a[m])

{

printf("查询数的位置为；");

printf("%d",m);

break;

}

else if (k < a[m])

{

r = m - 1;

}

else

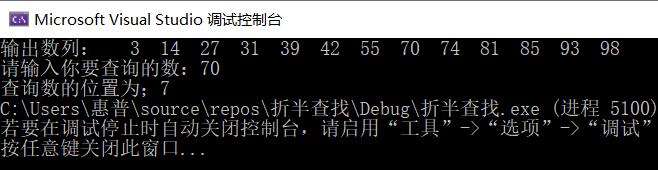
{

l = m + 1;

}

}

}四．结果



13计算中值和选择问题

一．算法描述

二．伪代码

Quickselect(A[l..r],k)

//用基于划分的递归算法解决选择问题

//输入：可排序数组A[0…n-10]的子数组A[l…r]和整数k（1<=k<=r-l+1）

//输出：A[l…r]中第k小元素的值

S <- lomutopartition(A[l..r])

If s= l+k-1 return A[s]

Else if s> l+k-1 Quickselect(A[l..s-1],k)

Else Quickselect(A[s+1…r],l+k-1-s)

三．代码

#include<iostream>

using namespace std;

#define M 99

int a[M];

void select(int left, int right, int n)

{

int temp, t;

int i = left, j = right;

temp = a[left];

while (i != j)

{

while (a[j] > temp&& i < j)

j--;

if (i < j)

{

a[i] = a[j];

i++;

}

while (a[i] < temp && i < j)

i++;

if (i < j)

{

a[j] = a[i];

j--;

}

}

a[i] = temp;

if (i == n)

return; //如果跟需要差找的位数相等就是需要查询的

else if (i < n) //如果下于需要查询的位数，则从下一个开始查询

select(i + 1, right, n);

else //如果大于查询的位数，则从left开始到right的上一个开始

select(left, i - 1, n);

}

int main()

{

int n, m;

cout << "输入数组大小：";

cin >> n;

cout << "输入数组：" << endl;

int temp;

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

cin >> a[i];

cout << "输入选择的数（第几小的数）：";

cin >> m;

select(1, n, m); //传入开始的数和结束的数，需要寻找的数 跟快速排序类似

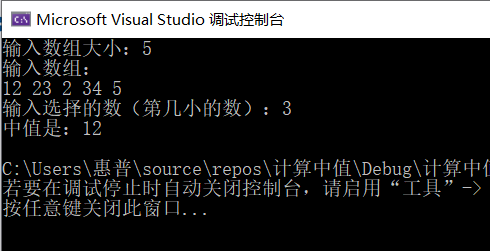
temp = a[m];

cout << "中值是：" << temp << endl;

return 0;

}

四．结果



14.合并排序

1.算法描述

2.伪代码

算法 Mergesort(A[0…n-1])

//递归调用mergesort来对数组A[0…n-1]排序

//输入：一个可排序数组A[0…n-1]

//输出：非降序排列的数组A[0…n-1]

If n>1

Copy A[0…(n/2)-1] to B[0…(n/2)-1]

Copy A[(n/2)…n-1] to B[(n/2)…n-1]

Mergesort(B[0…(n/2)-1])

Mergesort(C[(n/2)…n-1])

Merge(B,C,A)

算法 Merge(B[0…p-1],C[0…q-1],A[0…p+q-1])

//将两个有序数组合并为一个有序数组

//输入：两个有序数组B[0..p-1]和C[0…q-1]

//输出：A[0…p+q-1]中已经有序存放了B和C中的元素

i <-0;j<-0;k<-0

while i<p and j<q do

if B[i]<=C[j]

A[k]<-B[i];i<i+1

Else A[k]<-c[j];j<-j+1

K<-k+1

If i=p

Copy C[j…q-1] to A[k…p+q-1]

Else Copy B[i…p-1] to A[k…p+q-1]

3.代码

#include<stdio.h>

#include <corecrt\_malloc.h>

void Merge(int a[], int start, int mid, int end)

{

int\* tmp = (int\*)malloc((end - start + 1) \* sizeof(int));

int i = start;

int j = mid + 1;

int k = 0;

while (i <= mid && j <= end)

{

if (a[i] <= a[j])

{

tmp[k++] = a[i++];

}

else

tmp[k++] = a[j++];

}

while (i <= mid)

{

tmp[k++] = a[i++];

}

while (j <= end)

{

tmp[k++] = a[j++];

}

for (i = 0; i < k; i++)

a[start + i] = tmp[i];

free(tmp);

}

void Mergesort(int a[], int start, int end)

{

if (a == NULL || start >= end)

{

return;

}

int mid = (end + start) / 2;

Mergesort(a, start, mid);

Mergesort(a, mid + 1, end);

Merge(a, start, mid, end);

}

void shuchu(int a[])

{

for (int i = 0; i < 8; i++)

{

printf("%2d", a[i]);

}

}

int main()

{

int a[] = { 8,3,2,9,7,1,5,4 };

printf("原排列为：");

shuchu(a);

printf("\n");

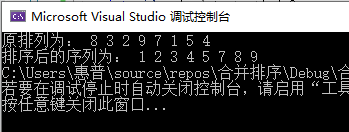
Mergesort(a, 0, 7);

printf("排序后的序列为：");

shuchu(a);

}

4.结果



15快速排序

1. 算法描述
2. 伪代码

算法 Quicksort(A[l…r])

//用Quicksort对子数组排序

//输入：数组A[0…n-1]中的子数组A[l..r]，由左右下标l和r定义

//输出：非降序排列的子数组A[l…r]

If l<r

S<-Partition(A[l…r])//s是分裂的位置

Quicksort(A[l…s-1])

Quick(A[s+1…r])

1. 代码

#include<stdio.h>

void Quicksort(int a[], int low, int high)

{

if (low < high) //判断是否满足排序条件，递归的终止条件

{

int i = low, j = high; //把待排序数组元素的第一个和最后一个下标分别赋值给i,j，使用i,j进行排序；

int x = a[low]; //将待排序数组的第一个元素作为哨兵，将数组划分为大于哨兵以及小于哨兵的两部分

while (i < j)

{

while (i < j && a[j] >= x) j--; //从最右侧元素开始，如果比哨兵大，那么它的位置就正确，然后判断前一个元素，直到不满足条件

if (i < j) a[i++] = a[j]; //把不满足位次条件的那个元素值赋值给第一个元素，（也即是哨兵元素，此时哨兵已经保存在x中，不会丢失）并把i的加1

while (i < j && a[i] <= x) i++; //换成左侧下标为i的元素开始与哨兵比较大小，比其小，那么它所处的位置就正确，然后判断后一个，直到不满足条件

if (i < j) a[j--] = a[i]; //把不满足位次条件的那个元素值赋值给下标为j的元素，（下标为j的元素已经保存到前面，不会丢失）并把j的加1

}

a[i] = x; //完成一次排序，把哨兵赋值到下标为i的位置，即前面的都比它小，后面的都比它大

Quicksort(a, low, i - 1); //递归进行哨兵前后两部分元素排序 ， low,high的值不发生变化，i处于中间

Quicksort(a, i + 1, high);

}

}

void shuchu(int a[])

{

for (int i=0; i < 9; i++)

{

printf("%2d", a[i]);

}

}

int main()

{

int a[] = { 8,1,9,7,2,4,5,6,3 };

printf("原数列为：");

shuchu(a);

printf("\n");

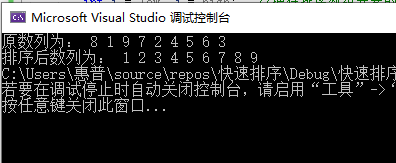
Quicksort(a, 0, 8);

printf("排序后数列为：");

shuchu(a);

}

1. 结果



16.预排序（检验数组中的元素唯一性）

1.算法描述

2.伪代码

算法 PersortElementUniqueness(A[0…n-1])

//先对数组排序来解元素唯一性问题

//输入：n个可排序元素构成的一个数组A[0…n-1]

//输出：如果A没有相等的元素，返回true，否则返回false

For i<-0 to n-2 do

If A[i]=A[i+1] return false

Return true

3.代码

#include<stdio.h>

void paixu(int a[])

{

int i, v, j;

for (i = 0; i < 10; i++)

{

v = a[i];

j = i - 1;

while (j >= 0 && a[j] > v)

{

a[j + 1] = a[j];

j = j - 1;

}

a[j + 1] = v;

}

for (int k = 0; k < 10; k++)

{

printf("%2d", a[k]);

}

}

int main()

{

int a[10] = {};

int i;

printf("请输入10个数字：");

for (i = 0; i < 10; i++)

{

scanf\_s("%d", &a[i]);

}

printf("输出数列为：");

paixu(a);

printf("\n");

for (i = 0; i < 9;i++)

{

if (a[i] == a[i + 1])

{

printf("1");

}

else

{

printf("0");

}

}

}

1. 结果

17.预排序（模式计算）

1.算法描述

2.伪代码

算法：PresortMode(A[0…n-1])

//先对数组排序来计算它的模式

//输入：可排序元素构成的数组A[0…n-1]

//输出；该数组的模式

对数组A排序

I<=0

Modefrequency<-0

While i<=n-1 do

Runlength=1;runvalue=A[i]

While i+runlength<=n-1 and A[i+runlength]=runvalue

Runlength=runlength+1

If runlength>modefrequency

Modefrequency=runlength;modevalue=runlength

I=i+runlength

Return modevalue

3.代码

#include<stdio.h>

void paixu(int a[])

{

int i, v, j;

for (i = 1; i < 10; i++)

{

v = a[i];

j = i - 1;

while (j >= 0 && a[j] > v)

{

a[j + 1] = a[j];

j = j - 1;

}

a[j + 1] = v;

}

for (i = 0; i < 10; i++)

{

printf("%4d", a[i]);

}

}

void presortmode(int a[])

{

int i = 0, m = 0,modevalue;

while (i <= 9)

{

int runlength = 1, runvalue = a[i];

int k;

k = i + runlength;

while (k<=9 && (a[i + runlength] == runvalue))

{

runlength = runlength + 1;

}

if (runlength>m)

{

m = runlength;

modevalue = runvalue;

}

i = i + runlength;

}

printf("%d",modevalue);

}

int main()

{

int a[10] = {};

int i;

printf("请输入10个数");

for (i = 0; i < 10; i++)

{

scanf\_s("%d", &a[i]);

}

paixu(a);

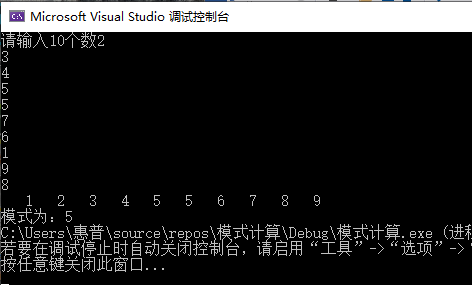
printf("\n");

printf("模式为：");

presortmode(a);

}

4.结果



18.Dijistra算法

1.算法描述

算法思想：设G=(V,E)是一个带权有向图，把图中顶点集合V分成两组，第一组为已求出最短路径的顶点集合（用S表示，初始时S中只有一个源点，以后每求得一条最短路径 , 就将加入到集合S中，直到全部顶点都加入到S中，算法就结束了），第二组为其余未确定最短路径的顶点集合（用U表示），按最短路径长度的递增次序依次把第二组的顶点加入S中。在加入的过程中，总保持从源点v到S中各顶点的最短路径长度不大于从源点v到U中任何顶点的最短路径长度。此外，每个顶点对应一个距离，S中的顶点的距离就是从v到此顶点的最短路径长度，U中的顶点的距离，是从v到此顶点只包括S中的顶点为中间顶点的当前最短路径长度。

2.伪代码

// 输入：具非负权重加权连通图G=<V,E>以及它的顶点s

// 输出：对于V中的每个顶点v来说，从s到v的最短路径长度d以及路径上的倒数第二个顶点p

//  初始化,设从0开始

for i=[0,n)

    dist[i] = map[0][i]

visit[0] = true;

for i=[1,n)

    //  寻找最短路径(s,t)，同时把t加入S集合

    min = MAX\_VALUE

    for j=[0,n)

        if !visit[j] && dist[j]<min

            min = dist[j]//记录最小值和最小值的下标

min\_j = j

    visit[j] = true

    //  松弛边(t,v)，其中v为顶点

    for k=[0,n)

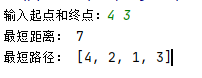
        if !visit[k] && dist[k]>dist[j]+tab[j][k]

            dist[k] = dist[j]+tab[j][k]

3.代码

*# 定义不可达距离*\_ = float(**'inf'**)  
  
  
*# points点个数，edges边个数,graph路径连通图,start七点,end终点*def Dijkstra(points, edges, graph, start, end):  
 map = [[\_ for i in range(points + 1)] for j in range(points + 1)]  
 pre = [0] \* (points + 1) *# 记录前驱* vis = [0] \* (points + 1) *# 记录节点遍历状态* dis = [\_ for i in range(points + 1)] *# 保存最短距离* road = [0] \* (points + 1) *# 保存最短路径* roads = []  
 map = graph  
  
 for i in range(points + 1): *# 初始化起点到其他点的距离* if i == start:  
 dis[i] = 0  
 else:  
 dis[i] = map[start][i]  
 if map[start][i] != \_:  
 pre[i] = start  
 else:  
 pre[i] = -1  
 vis[start] = 1  
 for i in range(points + 1): *# 每循环一次确定一条最短路* min = \_  
 for j in range(points + 1): *# 寻找当前最短路* if vis[j] == 0 and dis[j] < min:  
 t = j  
 min = dis[j]  
 vis[t] = 1 *# 找到最短的一条路径 ,标记* for j in range(points + 1):  
 if vis[j] == 0 and dis[j] > dis[t] + map[t][j]:  
 dis[j] = dis[t] + map[t][j]  
 pre[j] = t  
 p = end  
 len = 0  
 while p >= 1 and len < points:  
 road[len] = p  
 p = pre[p]  
 len += 1  
 mark = 0  
 len -= 1  
 while len >= 0:  
 roads.append(road[len])  
 len -= 1  
 return dis[end], roads  
  
  
*# 固定map图*def map():  
 map = [[\_, \_, \_, \_, \_, \_],  
 [\_, \_, 2, 3, \_, 7],  
 [\_, 2, \_, \_, 2, \_],  
 [\_, 3, \_, \_, \_, 5],  
 [\_, \_, 2, \_, \_, 3],  
 [\_, 7, \_, 5, 3, \_]  
 ]  
 s, e = input(**"输入起点和终点："**).split()  
 dis, road = Dijkstra(5, 7, map, int(s), int(e))  
 print(**"最短距离："**, dis)  
 print(**"最短路径："**, road)  
  
  
*# 输入边关系构造map图*def createmap():  
 a, b = input(**"输入节点数和边数："**).split()  
 n = int(a)  
 m = int(b)  
 map = [[\_ for i in range(n + 1)] for j in range(n + 1)]  
 for i in range(m + 1):  
 x, y, z = input(**"输入两边和长度："**).split()  
 point = int(x)  
 edge = int(y)  
 map[point][edge] = float(z)  
 map[edge][point] = float(z)  
 s, e = input(**"输入起点和终点："**).split()  
 start = int(s)  
 end = int(e)  
 dis, road = Dijkstra(n, m, map, start, end)  
 print(**"最短距离："**, dis)  
 print(**"最短路径："**, road)  
  
  
if \_\_name\_\_ == **'\_\_main\_\_'**:  
 map()

4.结果



19.Kruskal算法

1.算法描述

输入： 图G

输出： 图G的最小生成树

具体流程：

(1)将图G看做一个森林，每个顶点为一棵独立的树

(2)将所有的边加入集合S，即一开始S = E

(3)从S中拿出一条最短的边(u,v)，如果(u,v)不在同一棵树内，则连接u,v合并这两棵树，同时将(u,v)加入生成树的边集E'

(4)重复(3)直到所有点属于同一棵树，边集E'就是一棵最小生成树

2.伪代码

// 把所有边排序，记第i小的边为e[i] (1<=i<=m)m为边的个数

// 初始化MST为空

// 初始化连通分量，使每个点各自成为一个独立的连通分量

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

{

if (e[i].u和e[i].v不在同一连通分量)

{

// 把边e[i]加入ＭＳＴ

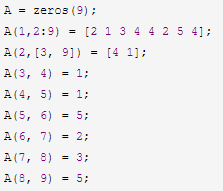
// 合并e[i].u和e[i].v所在的连通分量

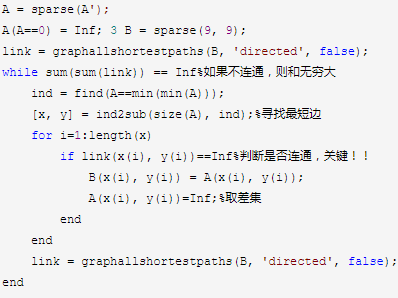
}

}

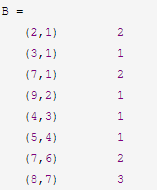
3.代码

预输入





4.结果



20.Prim算法

1.算法描述

1).输入：一个加权连通图，其中顶点集合为V，边集合为E；

2).初始化：Vnew = {x}，其中x为集合V中的任一节点（起始点），Enew = {},为空；

3).重复下列操作，直到Vnew = V：

a.在集合E中选取权值最小的边<u, v>，其中u为集合Vnew中的元素，而v不在Vnew集合当中，并且v∈V（如果存在有多条满足前述条件即具有相同权值的边，则可任意选取其中之一）；

b.将v加入集合Vnew中，将<u, v>边加入集合Enew中；

4).输出：使用集合Vnew和Enew来描述所得到的最小生成树。

2.伪代码

void Prim()

{

MST = {s};

while (1) {

V = 未收录顶点中dist最小者;

if ( 这样的V不存在 )

break;

将V收录进MST: dist[V] = 0;

for ( V 的每个邻接点 W )

if ( dist[W]!=W未被收录 0 )

if ( E (V,W) < dist[W] ){

dist[W] = E (V,W) ;

parent[W] = V;

}

}

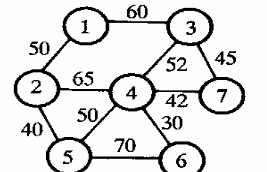
if ( MST中收的顶点不到|V|个 )

Error ( “生成树不存在” );

}

3.代码

给一个加权连通图



用matlab算法进行求解

a=zeros(7);

a(1,2)=50;a(1,3)=60;

a(2,4)=65;a(2,5)=40;

a(3,4)=52;a(3,7)=45;

a(4,5)=50;a(4,6)=30;a(4,7)=42;

a(5,6)=70;

a=a+a';a(find(a==0))=inf;

result=[];

p=1; %起点为1

tb=2:length(a);

while length(result)~=length(a)-1

temp=a(p,tb);temp=temp(:);

d=min(temp);

[jb,kb]=find(a(p,tb)==d);

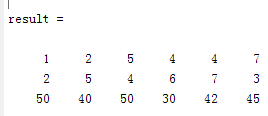
j=p(jb(1));k=tb(kb(1));

result=[result,[j;k;d]];p=[p,k];tb(find(tb==k))=[];

end

result

4.结果



21.Dijistra算法

1.算法描述

2.伪代码

3.代码

4.结果

22.哈夫曼树算法

1.算法描述

假设有n个权值，则构造出的哈夫曼树有n个叶子结点。 n个权值分别设为 w1、w2、…、wn，则哈夫曼树的构造规则为：

(1) 将w1、w2、…，wn看成是有n 棵树的森林(每棵树仅有一个结点)；

(2) 在森林中选出两个根结点的权值最小的树合并，作为一棵新树的左、右子树，且新树的根结点权值为其左、右子树根结点权值之和；

(3)从森林中删除选取的两棵树，并将新树加入森林；

(4)重复(2)、(3)步，直到森林中只剩一棵树为止，该树即为所求得的哈夫曼树。

2.伪代码

procedure Huffman( f)

Input: An array f[1...n] of frequencies

0utput: 1 An encoding tree with n leaves临8

let H be a priority queue of integers，ordered by f中V

for i = 1 to n: insert(H,i)

fork=n+1to2n-1:

i = deletemin(H)， j = deletemin(H)

create a node numbered : k with children i, j

f[k]= f[i]+ f[j]

insert(H, k)

3.代码

#节点类

class Node(object):

def \_\_init\_\_(self,name=None,value=None):

self.\_name=name

self.\_value=value

self.\_left=None

self.\_right=None

#哈夫曼树类

class HuffmanTree(object):

#根据Huffman树的思想：以叶子节点为基础，反向建立Huffman树

def \_\_init\_\_(self,char\_weights):

self.a=[Node(part[0],part[1]) for part in char\_weights] #根据输入的字符及其频数生成叶子节点

while len(self.a)!=1:

self.a.sort(key=lambda node:node.\_value,reverse=True)

c=Node(value=(self.a[-1].\_value+self.a[-2].\_value))

c.\_left=self.a.pop(-1)

c.\_right=self.a.pop(-1)

self.a.append(c)

self.root=self.a[0]

self.b=range(10) #self.b用于保存每个叶子节点的Haffuman编码,range的值只需要不小于树的深度就行

#用递归的思想生成编码

def pre(self,tree,length):

node=tree

if (not node):

return

elif node.\_name:

print node.\_name + '的编码为:',

for i in range(length):

print self.b[i],

print '\n'

return

self.b[length]=0

self.pre(node.\_left,length+1)

self.b[length]=1

self.pre(node.\_right,length+1)

#生成哈夫曼编码

def get\_code(self):

self.pre(self.root,0)

if \_\_name\_\_=='\_\_main\_\_':

#输入的是字符及其频数

char\_weights=[('a',5),('b',4),('c',10),('d',8),('f',15),('g',2)]

tree=HuffmanTree(char\_weights)

tree.get\_code()

4.结果

