## 一、Top-K问题

#### 1.理解问题

问题：已知一个序列，返回序列中top-k （如前k个最小的）元素

该问题属于排序问题。

输入：依次输入k、序列个数n、序列每个元素，中间用空格隔开,k<=n。

例如：输入5 9 66 79 5 34 84 666 128 468 53

其中k=5，n=9，序列元素依次为66,79,5,34,84,666,128,468,53

经过排序得到序列 5,34,53,66,79,84,128,468,666

输出为：5 34 53 66 79

#### **2.**算法设计

排序问题有多种算法实现，如：蛮力法（选择排序、冒泡排序），减治法（插入排序），分治法（合并排序、快速排序），变治法（预排序、堆排序）、时空权衡（计数排序），以及希尔排序等。

优化：

1.部分算法可以只进行k趟就可以输出top-K（前K个最小的），如选择排序、冒泡排序（从 后向前冒泡K次）、插入排序。

2.计数排序可以由绝对映射变为相对映射，避免数组中某些元素为负数该方法不可用。

#### **3.**算法代码与分析

###### 蛮力法-选择排序

void SelectionSort(int length,int k,int \*arr)

{

    for (int i = 1; i <=k; i++)

    {

        int k=i;

        for (int j = i+1; j <= length; j++)

        {

            if(arr[j]<arr[k])k=j;

        }

        if(i!=k)

        {

            int tmp=arr[i];

            arr[i]=arr[k];

            arr[k]=tmp;

        }

    }

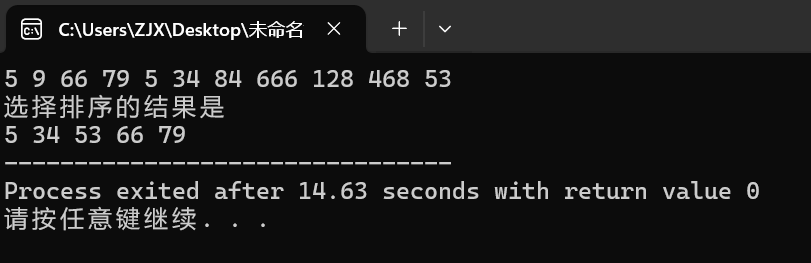
    for (int i = 1; i <= k; i++)

    {

        printf("%d ",arr[i]);

    }

}



###### 蛮力法-冒泡排序

//从后向前起泡 将最小的K个放在前面即可

void BubbleSort(int length,int k,int \*arr)

{

    for (int i = 1,change=1; i <=k&&change; i++)

    {

        change =0;

        for (int j = length; j >i; j--)

        {

            if (arr[j]<arr[j-1])

            {

                int tmp=arr[j];

                arr[j]=arr[j-1];

                arr[j-1]=tmp;

                change=1;

            }

        }

    }

}



###### 减治法-插入排序

//arr[0]为监视哨

void InsertionSort(int length,int k,int \*arr)

{

    for (int i = 2; i <=k ; i++) //只需要做到k

    {

        if (arr[i]<arr[i-1])

        {

            arr[0]=arr[i];//复制为监视哨

            int j;

            for (j = i-1; arr[0] < arr[j]; j--)

            {

                arr[j+1]=arr[j];//记录后移

            }

            arr[j+1]=arr[0];//插入到正确位置

        }

    }

}



###### 分治法-合并排序

void Merge(int \*SR,int \*TR,int i,int m,int n)

{

    //将有序的SR[i....m]和SR[m+1....n]归并为有序的TR[i....n]

    int k,j;

    for (int j = m+1,k=i; i<=m && j<=n; k++)

    {

        //将SR记录中按关键字从小到大地复制到TR中

        //K指示新表TR的当前元素的下标

        if (SR[i]<=SR[j])TR[k]=SR[i++];

        else TR[k]=SR[j++];

    }

    while(i<=m)TR[k++]=SR[i++];// 将剩余的 SR[i..m] 复制到TR

    while(j<=n)TR[k++]=SR[j++];// 将剩余的 SR[j..n] 复制到TR

}

void MSort(int \*SR, int \*TR1,int s,int t)

{

    //对SR[s....t]进行归并排序，排序后的记录存入TR1[s...t]

    if(s==t)TR1[s]=SR[s];

    else

    {

        int m=(s+t)/2;// 将 SR[s..t] 平分为 SR[s..m] 和 SR[m+1..t]

        Msort(SR,TR1,s,m,t);

    }

}



###### 分治法-快速排序

//分治法-快速排序

int Partition(int \*R,int low,int high)

{

    // 对记录子序列 R[low..high] 进行一趟快速排序，并返回枢轴记录

    // 所在位置，使得在它之前的记录的关键字均不大于它的关键字

    // 而在它之后的记录的关键字均不小于它的关键字

    R[0]=R[low];// 将枢轴记录移至数组的闲置分量

    int pivotkey=R[low];// 枢轴记录关键字

    while (low<high)

    {

        while (low<high&&R[high]>=pivotkey)--high;

        R[low++]=R[high];// 将比枢轴记录小的记录移到低端

        while (low<high&&R[low]<=pivotkey)++low;

        R[high--]=R[low];

    }

    R[low]=R[0];// 枢轴记录移到正确位置

    return low; //返回枢轴位置

}

void QSort(int \*R,int s,int t)

{

    // 对记录序列 R[s..t] 进行快速排序

    if(s<t)

    {

        int pivotloc=Partition(R,s,t);//进行一趟快排，返回pivot位置

        QSort(R,s,pivotloc-1);//对低子序列递归排序

        QSort(R,s,pivotloc,t);//对高子序列递归排序

    }

}



###### 变治法-堆排序

void HeapAdjust(int \*H,int s,int m)

{

    // 已知H[s..m]中除H[s]之外均满足堆的定义

    // 本函数依据关键字的大小对H[s]进行调整，使H[s..m]成为一

    // 个大顶堆（对其中记录的关键字而言）

    H[0]=H[s];

    for (int j = 2\*s; j <= m; j\*=2)// 沿关键字较大的孩子结点向下筛选

    {

        if(j<m&&H[j]<H[j+1])++j;// j 为关键字较大的孩子记录的下标

        if(H[0]>=H[j])break; //不需调整，跳出循环

        H[s]=H[j];// 将大关键字记录往上调

        s=j;

    }

    H[s]=H[0];// 回移筛选下来的记录

}

void HeapSort(int \*H,int length)

{

    for(int i=length/2;i>0;i--)HeapAdjust(H,i,length);//自下而上的筛选,建大顶堆

    for(int i=length;i>1;i--)

    {

        int tmp=H[1];// 将堆顶记录和当前未经排序子序列H.r[1..i]中最后一个记录相互交

        H[1]=H[i];

        H[i]=tmp;

        HeapAdjust(H,1,i-1);//自上而下的筛选

    }

}



###### 时空权衡-计数排序

//对于每一个元素，算出该集合中小于该元素的个数(从0下标开始存储)

void CountSort(int \*arr,int n)

{

    //找序列中的最大最小值

    int max=arr[0];

    int min=arr[0];

    for (int i = 1; i < n; i++)

    {

        if(arr[i]>max)max=arr[i];

        if(arr[i]<min)min=arr[i];

    }

    int range=max-min+1;    //开辟空间的数量

    int \*countArr=(int \*)malloc(sizeof(int)\*range);//开辟空间

    //初始化数组全部为0

    memset(countArr,0,sizeof(int)\*range);

    //开始计数

    for (int i = 0; i < n; i++)

    {

        countArr[arr[i]-min]++;

    }

    //排序

    int j=0;

    for (int i = 0; i <range; i++)

    {

        while (countArr[i]--)

        {

            arr[j++]=i+min;

        }

    }

    free(countArr);

}



###### 希尔排序

//缩小增量法 shell sort

void ShellInsert(int \*arr,int dk,int length)

{

    //对顺序表L作一趟增量为dk的希尔排序

    for (int i = dk+1; i <= length; i++)

    {

        if (arr[i]<arr[i-dk])

        {

            arr[0]=arr[i];

            int j;

            for (int j = i-dk; j>0&& arr[0]< arr[j]; j-=dk)

            {

                arr[j+dk]=arr[j];

            }

            arr[j+dk]=arr[0];

        }

    }

}

void SheelSort(int \*arr,int \*dlta,int t,int length)

{

    //按照增量序列dlta[0....t-1]对arr做希尔排序

    for (int k = 0; k < t; k++)

    {

        ShellInsert(arr,dlta[k],length);

    }

}



#### **4.**对比分析

排序问题有多种算法实现，如：蛮力法（选择排序、冒泡排序），减治法（插入排序），分治法（合并排序、快速排序），变治法（预排序、堆排序）、时空权衡（计数排序），以及希尔排序等。

1. 平均时间性能：

时间复杂度为O（nlogn）:合并排序、快速排序、堆排序

时间复杂度为O（n2）: 选择排序、冒泡排序、插入排序

1. 平均空间性能：

插入排序、冒泡排序、简单选择、堆排序的空间复杂度为O(1)；

快速排序为O(logn)，为递归程序执行过程中，栈所需的辅助空间

归并排序所需辅助空间最多，其空间复杂度为 O(n);

1.  计数排序:

缺点:不能对小数进行排序；空间复杂度O（range），空间浪费不能彻底的解决。

优点：时间复杂度为：O（N+range），对集中的数据排序效率极高。

## 二、O（1）找栈的最小值

#### 1.理解问题

问题：给栈增加一个求最小值的操作，请给出栈的实现策略，使得入栈、出栈与求最小值都能在O（1）内完成，并进行相应算法分析。

输入：依次输入栈的大小size、栈的每个元素，中间用空格隔开。

例如：输入5 9 66 79 6 34

其中size=5，序列元素依次为9,66,79,6,34

输出为：5

#### **2.**算法设计

栈是一个只允许在一端进行插入和删除的线性表。

选择顺序栈的数据结构，入栈、出栈操作时间为O（1），现设置一个辅助栈S2，初始化tmp为第一个元素值，并压入S2栈，在入栈S1时用tmp与当前入栈元素比较，若入栈元素值小于或等于tmp,则更新tmp值并将该元素值压入S2中，最后S2栈顶的元素即为最小值。

#### **3.**算法代码与分析

#include<stdio.h>

#include<stdlib.h>

typedef struct{

    int \*top;//栈顶指针

    int \*base;//栈底指针

    int stacksize;//栈可用最大容量

}SqStack;

int InitStack(SqStack \*S,int size)

{

    S->base=(int \*)malloc(sizeof(int)\*size);

    if(!S->base)return 0;

    S->top=S->base;

    S->stacksize=size;

    return 1;

}

int Push(SqStack \*S,int elem)

{

    if (S->top-S->base==S->stacksize)

    {

        return 0;

    }

    \*S->top++=elem;//先把elem赋给栈顶

    return 1;

}

int Pop(SqStack \*S,int \*elem)

{

    if(S->top==S->base)return 0;//栈空

    \*elem=\*--S->top;//栈顶指针下移，元素赋给elem

    return 1;

}

int main()

{

    SqStack S1;

    SqStack S2;//辅助栈

    int size;

    scanf("%d",&size);

    InitStack(&S1,size);

    InitStack(&S2,size);

    int data;

    int tmp;//tmp即最小值

    for (int i = 0; i < size; i++)

    {

        scanf("%d",&data);

        Push(&S1,data);

        if(i==0)

        {

            tmp=data;

            Push(&S2,tmp);

        }

        else

        {

            if(data<tmp)Push(&S2,data);

            tmp=data;

        }

    }

}

