## 一、三分查找

#### 1.理解问题

二分查找的数组必须是有序的，所以二分法解决的是单调函数的求解问题。

而针对于二次函数，如果想搜索二次函数的最小值或最大值，可以用三分查找，将区间分成三份。前提是数组要在左区间单调递减（递增），在右区间单调递增（递减）。

输入：依次输入数组的长度、数组内的所有元素

例如：7 1 3 5 9 7 6 2

输出：9

#### **2.**算法设计

求解分为两种情况，分为求波峰和波谷。

1. 求波谷
   1. midl=L+(R-L)/3，midr=R-(R-L)/3
   2. 如果arr[midl]==arr[midr]，返回（midl+midr）/2
   3. 如果arr[midl] > arr[midr]，说明midr处的值更接近波谷，令 L=midl+1
   4. 如果arr[midl] < arr[midr]，说明midl处的值更接近波谷，令 R=midr-1
   5. 重复上述步骤直到返回或者L.> R没找到
2. 求波峰
   1. midl=L+(R-L)/3，midr=R-(R-L)/3
   2. 如果arr[midl]==arr[midr]，返回（midl+midr）/2
   3. 如果arr[midl] < arr[midr]，说明midr处的值更接近波峰，令 L=midl+1
   4. 如果arr[midl] > arr[midr]，说明midl处的值更接近波峰，令 R=midr-1
   5. 重复上述步骤直到返回或者L.> R没找到

要注意左右对称的时候要返回（midl+midr）/2

#### 算法代码与分析

#include <stdio.h>

#include <stdlib.h>

int TriSearchMin(int L,int R,int arr[])

{

    int midl,midr;

    while (L<=R)

    {

        midl=L+(R-L)/3;

        midr=R-(R-L)/3;

        //可能会有midl和midr左右对称的情况

        if (arr[midl]==arr[midr])return (midl+midr)/2;

        if (arr[midl]>arr[midr])L=midl+1;

        if (arr[midl]<arr[midr])R=midr-1;

    }

    return -1;//没找到

}

int TriSearchMax(int L,int R,int arr[])

{

    int midl,midr;

    while (L<=R)

    {

        midl=L+(R-L)/3;

        midr=R-(R-L)/3;

        if(arr[midl]==arr[midr])return (midl+midr)/2;

        if(arr[midl]>arr[midr])R=midr-1;

        if(arr[midl]<arr[midr])L=midl+1;

    }

    return -1;

}

int main(){

    int length;

    printf("Enter the length of the array: \n");

    scanf("%d", &length);

    while (length>0)

    {

        int arr[100];

        printf("Enter the elements of the array: \n");

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

        {

            scanf("%d", &arr[i]);

        }

        printf("\n1.Get min\n");

        printf("2.Get max\n");

        printf("Please input the command order:");

        int command;

        again:

        scanf("%d", &command);

        int rs;

        switch (command)

        {

        case 1:

            rs = TriSearchMin(0, length-1, arr);

            break;

        case 2:

            rs = TriSearchMax(0, length-1, arr);

            break;

        default:

            printf("Invalid command!Please input command again!!\n");

            goto again;

        }

        if (rs==-1)

        {

            printf("Not Found!\n");

        }

        else

        {

            printf("待求元素是第%d个元素,元素为%d\n",rs+1,arr[rs]);

        }

        printf("\n\nEnter the length of the array: \n");

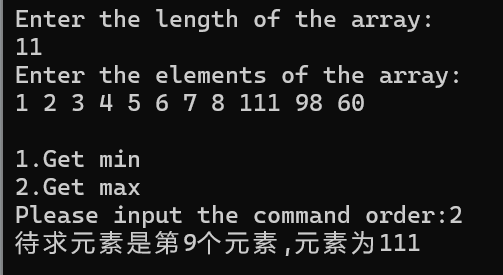
        scanf("%d", &length);

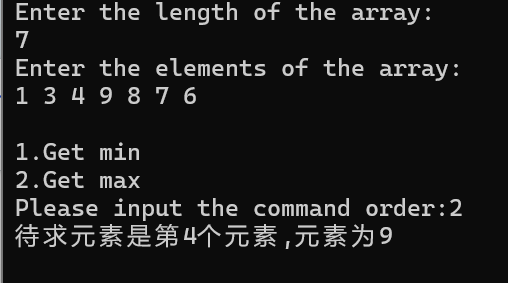
    }

    return 0;

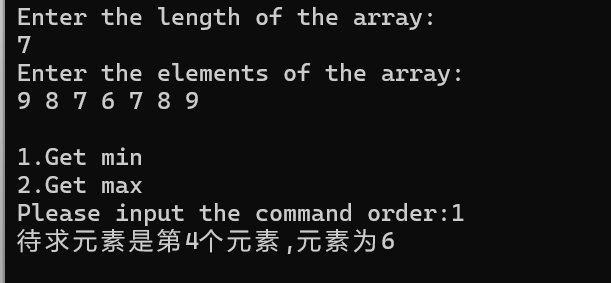
}

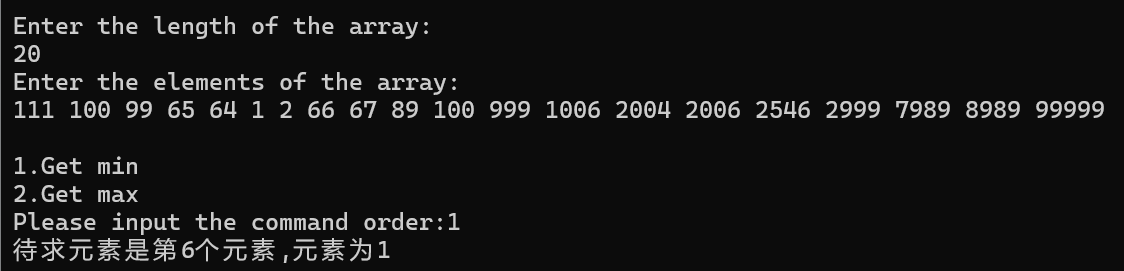
求最大值测试：



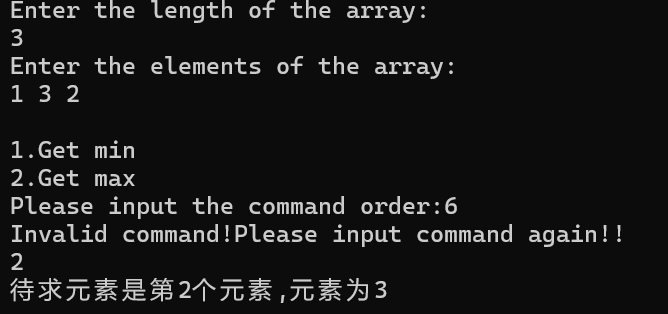


求最小值测试：





输入命令错误：



#### 性能分析

三分查找的时间复杂度为O(logN)级别，类似于构造了一颗递归树，一共有log3（n）层；需要的额外空间复杂度为O（1）级别，因为只需要midl、midr、L、R这几个临时变量。相对于顺序查找是一个性能较好的算法。

## 二、Fibonacci查找

#### 1.理解问题

假设有一个有序数组 arr，长度为 n，而 n 正好是斐波那契数列中的某个值。

斐波那契查找算法是一种基于黄金分割的有序查找算法，通过斐波那契数列的特性，在有序序列中快速定位目标元素的位置。

通常，斐波那契数列的前几项是：0, 1, 1, 2, 3, 5, 8, 13, 21, ...

假设这个有序数组arr的长度是5，arr = [1, 3, 5, 7, 9]

我们需要查找元素5在数组中的位置，以下为查找的步骤：

**1. 选择斐波那契数列的值**： 在斐波那契数列中找到一个最小的值，使得 F(k) >= n，其中 k 是最小可能满足的索引。在这个例子中，我们选择 F(5) = 5。

**2. 分割数组**： 将数组分成两个部分，长度比例为斐波那契数列中的两个相邻值的比例。在这个例子中，我们有两部分，长度比例是 3:2。

arr\_left = [1, 3, 5] arr\_right = [7, 9]

**3. 比较与查找**： 比较要查找的元素（5）与分割点元素（arr[2] = 5）。如果相等，找到了目标元素。如果待查找元素小于分割点元素，继续在前半部分进行查找。如果待查找元素大于分割点元素，继续在后半部分进行查找。

在这个例子中，5 等于分割点元素，所以我们找到了目标元素的位置。

**4. 迭代**： 重复上述步骤，直到找到目标元素或确定元素不在序列中。

#### **2.**算法设计

斐波那契数列：

斐波那契数列是一组按以下递归关系定义的数字序列：F(0) = 0，F(1) = 1，F(n) = F(n-1) + F(n-2)（n > 1）。通常，斐波那契数列的前几项是：0, 1, 1, 2, 3, 5, 8, 13, 21, ... 在这个算法中用于对数组进行分割。

1. 查找过程：

对给定的有序序列arr，首先选择一个斐波那契数列中的下标为k的值，使fibonacci[k]>=arr.length，然后使用将arr分成前fibnacci[k-1]的部分和后fibonacci[k-2]两个部分。这两个部分的长度之比就是相邻两个斐波那契数的比例。

1. 比较：

比较要查找的元素x与序列中分割点的元素。如果相等，则找到了目标元素；如果待查找元素小于分割点元素，继续在前半部分进行查找；如果待查找元素大于分割点元素，继续在后半部分进行查找。

1. 迭代：

重复上述步骤，不断缩小查找范围，直到找到目标元素或确定元素不在序列中。

注意点：第一次写程序调试的时候没法输出“没找到该元素”，后来改变了while循环的条件，加了一个k>0的判断，就可以正常输出了。

#### 3.算法代码与分析

#include <stdio.h>

#include <stdlib.h>

int fibonacci[1000]={0,1,1};

void generateFib(){

    for(int i=3;i<1000;i++){

        fibonacci[i]=fibonacci[i-1]+fibonacci[i-2];

    }

}

int FibonacciSearch(int x,int arr[],int length){

    int k=0;

    //找到最接近且>=length的值，用来进行划分

    while (fibonacci[k]<length)

    {

        k++;

    }

    //分界点的数的下标

    int L=0,R=length-1;

    int mid;

    while(L<=R&&k>0)

    {

        mid=L+fibonacci[k-1]-1;

        if(arr[mid]==x)return mid;

        else if(arr[mid]<x){

            L=mid+1;

            k-=2;

        }else if(arr[mid]>x){

            R=mid;

            k--;

        }

    }

    return -1;//没找到

}

int main(){

    generateFib();

    int length;

    printf("Enter the length of the array: \n");

    scanf("%d", &length);

    int arr[100];

    printf("Enter the elements of the array: \n");

    for (int i = 0; i < length; i++)

    {

        scanf("%d", &arr[i]);

    }

    int x;

    printf("Enter the element you want to search: \n");

    scanf("%d", &x);

    int result=FibonacciSearch(x, arr, length);

    if(result==-1)printf("没找到该元素\n");

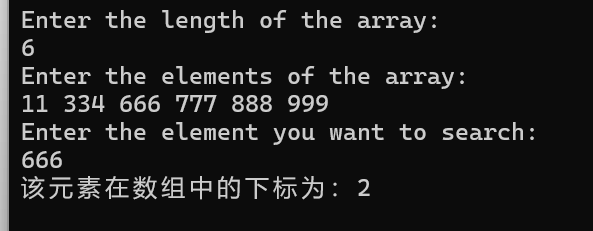
    else printf("该元素在数组中的下标为：%d\n", result);

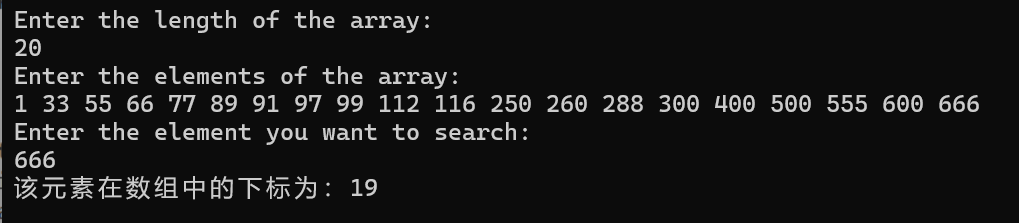
    return 0;

}

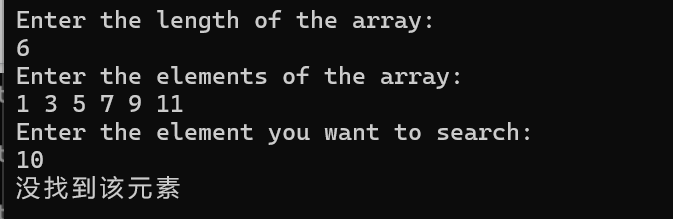
测试结果：

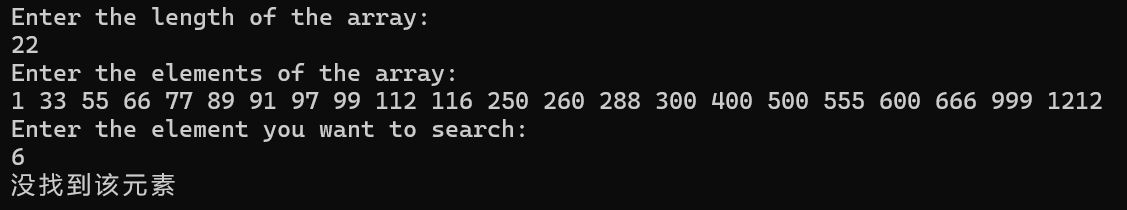
查找成功





查找失败：





#### 4.性能分析

时间复杂度：

1. 查找过程： 斐波那契查找的主要操作是不断地缩小查找范围，通过比较待查找元素与分割点元素来确定继续在前半部分还是后半部分进行查找。在每一步操作中，都可以将待查找范围缩小为原来的**黄金分割比例**，即约为 0.618。
2. 时间复杂度： 斐波那契查找的时间复杂度可以表示为 O(log n)，其中 n 是待查找序列的长度。与二分查找相比，它的复杂度相对更低。

空间复杂度：

1. 常数空间：斐波那契查找算法的空间复杂度非常低。它只需要常数级别的额外空间来存储一些中间变量，如斐波那契数列的值fibonacci[]、分割点mid等。
2. 空间复杂度是O（1）级别的。

#### 5.使用场景

1. 有序序列： 斐波那契查找要求待查找的序列是有序的，因为它是基于比较来缩小查找范围的。如果序列无序，需要先进行排序操作。

2.长度接近斐波那契数： 算法对序列的长度有一定的要求，最好是恰好是斐波那契数列中的某个值。在实际应用中，可以选择最接近并大于待查找序列长度的斐波那契数。

3.分布均匀： 如果数据在序列中的分布相对均匀，斐波那契查找可以更好地发挥其优势。这是因为它能够在分割序列时保持更好的平衡。

4.查找频繁但数据变动不频繁： 如果对同一序列进行多次查找而且序列基本保持不变，斐波那契查找的一些前期计算可以提前完成，然后多次使用相同的计算结果进行查找，从而提高效率。

5.适用于内存有限的情况： 斐波那契查找只需要常数级别的额外空间，因此在内存有限的情况下比一些其他算法更为适用。

#### 6.优缺点

优点：

1.适用性广泛： 斐波那契查找适用于有序序列，尤其在序列长度接近斐波那契数列的某个值时效果更佳。相较于二分查找，斐波那契查找能够在某些特定情况下减少查找次数。

2.更好的平衡： 由于斐波那契查找使用黄金分割比例进行分割，使得分割后的两个子序列的长度比例更加接近，有助于保持查找的平衡性。

3.相对均匀的分割： 斐波那契查找相对于其他分割方法，如二分查找，能够产生更为均匀的分割，有助于在查找过程中更快地接近目标元素。

缺点：

1.数组长度限制： 斐波那契查找要求待查找的序列长度必须是斐波那契数列中的某个值，这在实际应用中可能不太灵活，特别是当数据规模不是恰好是斐波那契数列中的某个值时，可能需要对数据进行补齐。

2.比较次数不稳定： 斐波那契查找在某些情况下可能会比二分查找效果更好，但并不是在所有情况下都表现更好。具体的性能取决于待查找数据的分布情况和序列的长度。在一些场景下，二分查找可能更为稳定。

3.计算斐波那契数列： 为了选择合适的斐波那契数列的值，需要事先计算斐波那契数列，这可能涉及到一些计算成本，特别是对于较大的数据集。

总体来说，斐波那契查找算法在某些特定条件下表现优秀，但在实际应用中需要谨慎选择，并根据具体情况考虑使用。在一些情况下，简单的二分查找可能更加实用和高效。