

# 目录

Introduction	1.1
分治法	1.2
快速排序	1.2.1
中位数	1.2.2
最大子序和	1.2.3
线性时间排序	1.3
计数排序	1.3.1

# Introduction

记录我的算法分析学习历程

## 第一篇 分治法

# 快速排序

## 问题：

实现对数组 `int arr[9]={-2,1,-3,4,-1,2,1,-5,4}` 的快速排序，并画出流程图

## 方法：分治法

## 快速排序原理：

- 任找一个元素作为基准，对待排数组进行分组
- 使基准元素左边的数据都比基准元素小，右边的数据都比基准元素大。这样基准元素就放在了正确的位置上。
- 然后对基准元素左边和右边的数据分组进行相同的操作，最后完成数组的排序。

## 代码如下：

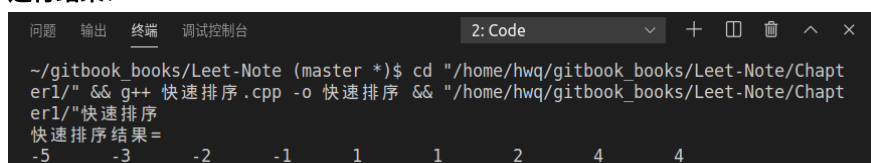
```
#include <iostream>
using std::cin;
using std::cout;
using std::endl;

int Partition(int arr[], int low, int high){
    int pivot_key=arr[low]; //临时存储基准值
    while(low<high){
        while (low<high && arr[high]>=pivot_key) --high;
        arr[low]=arr[high];
        while (low<high && arr[low]<=pivot_key) ++low;
        arr[high]=arr[low];
    }
    arr[low]=pivot_key; //把基准值放到最后准确的位置
    return low;
}

void QuickSort(int arr[], int low, int high){
    int pivot;
    if(low<high){
        pivot=Partition(arr, low, high);
        QuickSort(arr, low, pivot-1);
        QuickSort(arr, pivot+1, high);
    }
}

int main(){
    int arr[9]={-2,1,-3,4,-1,2,1,-5,4};
    QuickSort(arr, 0, 8);
    cout<<"快速排序结果= "<<endl;
    for (char i = 0; i < 9; i++){
        cout<<arr[i]<<"\t";
    }
    cout<<endl;
    return 0;
}
```

## 运行结果：



```
~/gitbook_books/Leet-Note (master *)$ cd "/home/hwq/gitbook_books/Leet-Note/Chapter1/" && g++ 快速排序.cpp -o 快速排序 && "/home/hwq/gitbook_books/Leet-Note/Chapter1/"快速排序
快速排序结果=
-5    -3    -2    -1    1    1    2    4    4
```

快排流程图

元素 第 N 趟	-2	1	-3	4	-1	2	1	-5	4
1	-5	-3	-2	4	-1	2	1	1	4
2	-5	-3	-2	1	-1	2	1	4	4
3	-5	-3	-2	-1	1	2	1	4	4
4	-5	-3	-2	-1	1	1	2	4	4

注: 褐色的数字格表示此趟快排的基准值

#### 算法复杂度分析:

时间复杂度:  $O(n\lg(n))$

空间复杂度:  $O(\lg(n))$

## Median of Two Sorted Arrays

### 问题：

There are two sorted arrays `nums1` and `nums2` of size `m` and `n` respectively. Find the median of the two sorted arrays. The overall run time complexity should be  $O(\log(m+n))$ .

### 中位数的概念

- 将一个集合划分为两个长度相等的子集，其中一个子集中的元素总是大于另一个子集中的元素。

### 方法：分治法

### 算法分析

- 将有序数组分成两部分，可以得到如下关系式：

```
len(left_part)=len(right_part)
max(left_part)≤min(right_part)
```

left_part	right_part
A[0], A[1], ..., A[i-1]	A[i], A[i+1], ..., A[m-1]
B[0], B[1], ..., B[j-1]	B[j], B[j+1], ..., B[n-1]

- 那么，中位数就是：

$$median = [\max(left\_part) + \min(right\_part)] / 2$$

### 代码如下：

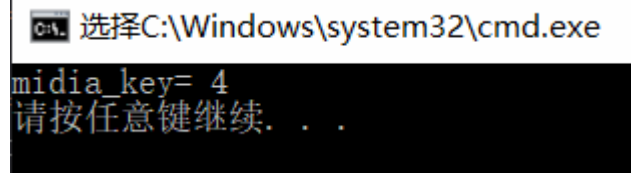
```
int findMedianSortedArrays(int A[], int A_len, int B[], int B_len) {
    int m=A_len, n=B_len;
    int iMin = 0, iMax = m, halfLen = (m + n + 1) / 2;
    while (iMin <= iMax) {
        int i = (iMin + iMax) / 2;
        int j = halfLen - i;
        if (i < iMax && B[j-1] > A[i]) {
            iMin = i + 1; // i is too small, 需要增大i, 减小j
        }
        else if (i > iMin && A[i-1] > B[j]) {
            iMax = i - 1; // i is too big, 需要减小i, 增大j
        }
        else { // i is perfect, i是临界值, 0或者m
            int maxLeft = 0;
            if (i == 0) { maxLeft = B[j-1]; }
            else if (j == 0) { maxLeft = A[i-1]; }
            else { maxLeft = max(A[i-1], B[j-1]); }
            if ((m + n) % 2 == 1) { return maxLeft; }

            int minRight = 0;
            if (i == m) { minRight = B[j]; }
            else if (j == n) { minRight = A[i]; }
            else { minRight = min(B[j], A[i]); }

            return (maxLeft + minRight) / 2;
        }
    }
}
```

**运行结果:**

数组元素为:array1[3] = {1,2,7}; array2[3] = {3,5,6};



**算法复杂度分析:**

- 时间复杂度：查找的区间是 $[0, m]$ , 每次循环之后，查找区间的长度都会降为原先的一半。所以，最多执行 $\lg(m)$ 次。由于 $m \leq n$ , 所以时间复杂度为 $O(\lg(\min(m, n)))$ 。

## 最大子序和

### 问题：

给定一个整数数组nums，找到一个具有最大和的连续子数组(子数组最少包含一个元素)，返回其最大和。

### 示例：

输入: [-2,1,-3,4,-1,2,1,-5,4]

输出: 6

解释: 连续子数组 [4,-1,2,1] 的和最大为 6

### 方法：分治法

### 算法分析：

1. 把数组分成左右两个子数组，最大子序和只可能出现在
  - 1.左子数组
  - 2.右子数组
  - 3.横跨左右子数组的部分或全部元素
2. 然后对左子数组或右子数组时，进一步拆分，依次循环，直至拆分的子数组中只有一个元素。
  - 拆分序列（直到只剩下一个数的数组）
  - 求左子数组最大值
  - 求右子数组最大值
  - 求横跨左右子数组的最大值
3. 合并，得出以上三个最大值的最大值
4. 当最大子数组有  $n$  个数字时：
  - 若  $n == 1$ ，返回此元素。
  - $left\_sum$  是左子数组的元素之和最大值
  - $right\_sum$  是右子数组的元素之和最大值
  - $cross\_sum$  是横跨左右子数组元素之和的最大值

代码如下：



```

#include <iostream>
using std::cout;
using std::cin;
using std::endl;
int CrossSum(int nums[],int left, int right, int mid) {
    if (left == right) return nums[left];
    int leftSubSum=0;
    int leftMaxSum=nums[mid]; //横跨左右子数组，则基准元素(左边第一个元素)必然包含在内
    for(int i=mid; i>=left; i--) {
        leftSubSum+=nums[i];
        leftMaxSum=leftMaxSum>=leftSubSum?leftMaxSum:leftSubSum;
    }
    int rightSubSum=0;
    int rightMaxSum=nums[mid+1]; //横跨左右子数组，则右边第一个元素必然包含在内
    for(int i=mid+1; i<=right; i++) {
        rightSubSum+=nums[i];
        rightMaxSum=rightMaxSum>=rightSubSum?rightMaxSum:rightSubSum;
    }
    return leftMaxSum+rightMaxSum;
}

int fun(int nums[],int left, int right) {
    if (left == right) return nums[left];

    int mid = (left + right) / 2;
    int leftSum = fun(nums, left, mid);
    int rightSum = fun(nums, mid + 1, right);
    int crossSum = CrossSum(nums, left, right, mid);
    int temp=leftSum>rightSum?leftSum:rightSum;

    return temp>crossSum?temp:crossSum;
}

int MaxSubArray(int nums[],int length) {
    return fun(nums, 0, length-1);
}

int main() {
    int nums[9]={-2,1,-3,4,-1,2,1,-5,4};
    int maxSum=MaxSubArray(nums,9);
    cout<<"maxSum= " <<maxSum<<endl;
}

```

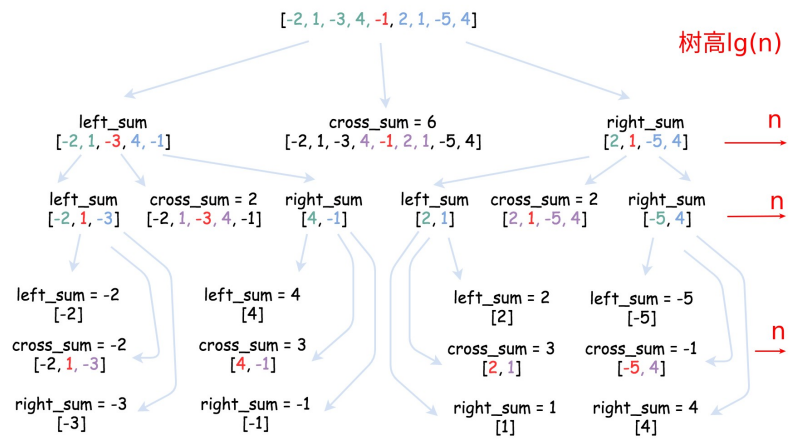
### 运行结果：

```
maxSum= 6
```

### 算法复杂度分析:

- 递归树法：

时间复杂度： $O(n\lg(n))$



## 线性时间排序方法

## 计数排序

### 问题：

实现对数组`int arr[10]={95,94,91,98,99,90,99,93,91,92}`的计数排序，并画出流程图

### 计数排序原理：

计数排序是由额外空间的辅助和元素本身的值决定的。计数排序过程中不存在元素之间的比较和交换操作，根据元素本身的值，将每个元素出现的次数记录到辅助空间后，通过对辅助空间内数据的计算，即可确定每一个元素最终的位置。

#### • 算法过程

1. 根据待排序集合中最大元素与最小元素的差值范围，申请额外辅助空间
2. 遍历待排序集合，将每一个元素出现的次数记录到元素值对应的辅助空间
3. 对辅助空间内的数据进行计算，得出每一个元素的正确位置
4. 将待排序集合的每一个元素移动到计算出的正确位置上，排序完成

### 代码如下：

```
#include <iostream>
#include <new>
using namespace std;

void CountSort(int *arr,int len,int max, int min)
{
    int *count=new int[max-min+1]; //计数数组
    int *Result=new int[len]; //存放排序后的结果
    int index;
    for (int i =0; i <=max-min; i++){ //初始化
        count[i]=0;
    }
    for(int i=0; i<len;i++){ //计算arr[i]元素出现的个数
        count[arr[i]-min]++;
    }
    for(int i=1; i<=max-min; i++){
        count[i]+=count[i-1];
    }

    for (int i = len-1; i >= 0; i--){
        index=count[arr[i]-min]-1;
        Result[index]=arr[i];
        count[arr[i]-min]--;
    }
    cout<<"计数排序后的结果= "<<endl;
    for(int i=0;i<len;i++){
        cout<<Result[i]<<'\\t';
    }

    cout<<endl;
    delete [] count;
    delete [] Result;
}

int main()
{
    int arr[10]={95,94,91,98,99,90,99,93,91,92}; //待排数组
    CountSort(arr,10,99,90); //计数排序
}
```

## 运行结果：

```

问题 输出 终端 调试控制台
2: Code
~/gitbook_books/Leet-Note (master *)$ cd "/home/hwq/gitbook_books/Leet-Note/Chapter2/" && g++ 计数排序.cpp -o 计数排序 && "/home/hwq/gitbook_books/Leet-Note/Chapter2/"计数排序
计数排序后的结果=
90    91    91    92    93    94    95    98    99    99

```

## 计数排序流程图

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y
1	arr[10]												4	Result[10]										
	序号	0	1	2	3	4	5	6	7	8	9			序号	0	1	2	3	4	5	6	7	8	9
	元素	95	94	91	98	99	90	99	93	91	92			元素				92						
2	count[10]												5	Result[10]										
	序号	0	1	2	3	4	5	6	7	8	9			序号	0	1	2	3	4	5	6	7	8	9
	个数	1	2	1	1	1	1	0	0	1	2			元素			91	92						
3	count[10]												6	Result[10]										
	序号	0	1	2	3	4	5	6	7	8	9			序号	0	1	2	3	4	5	6	7	8	9
	次序	1	3	4	5	6	7	7	7	8	10			元素			91	92	93					
														7	Result[10]									
														序号	0	1	2	3	4	5	6	7	8	9
														元素			91	92	93					99
														8	Result[10]									
														序号	0	1	2	3	4	5	6	7	8	9
														元素	90		91	92	93					99
														9	Result[10]									
														序号	0	1	2	3	4	5	6	7	8	9
														元素	90		91	92	93				99	99
														10	Result[10]									
														序号	0	1	2	3	4	5	6	7	8	9
														元素	90		91	92	93			98	99	99
														11	Result[10]									
														序号	0	1	2	3	4	5	6	7	8	9
														元素	90	91	91	92	93			98	99	99
														12	Result[10]									
														序号	0	1	2	3	4	5	6	7	8	9
														元素	90	91	91	92	93	94		98	99	99
														13	Result[10]									
														序号	0	1	2	3	4	5	6	7	8	9
														元素	90	91	91	92	93	94	95	98	99	99

注: Result下的深灰色格子代表此次排好序的元素

## 算法复杂度分析:

时间复杂度: 因为 $k = n$ , 所以为 $O(n)$

空间复杂度: 申请了额外辅助空间, 且 $k = n$ , 所以为 $O(n)$

## 计数排序与快速排序的区别:

1. 计数排序属于线性时间排序, 用非比较的操作确定排序顺序; 而快速排序是基于元素之间的比较, 属于比较排序
2. 任意一个比较排序算法, 在最坏情况下, 都需要做  $\Omega(\lg(n))$  次的比较; 而计数排序的运行时间为  $\Theta(k+n)$
3. 计数排序是一种稳定的排序算法; 而快速排序不是
4. 计数排序算法适合待排序元素在一定范围内, 数值比较集中; 而快速排序没有这种要求