十大排序算法

参考链接：https://mp.weixin.qq.com/s/vn3KiV-ez79FmbZ36SX9lg

排序算法	平均时间复杂度	最好情况	最坏情况	空间复杂度	排序方式	稳定性
冒泡排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	In-place	稳定
选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	In-place	不稳定
插入排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	In-place	稳定
希尔排序	$O(n \log n)$	$O(n \log^2 n)$	$O(n \log^2 n)$	$O(1)$	In-place	不稳定
归并排序	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$	Out-place	稳定
快速排序	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(\log n)$	In-place	不稳定
堆排序	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(1)$	In-place	不稳定
计数排序	$O(n + k)$	$O(n + k)$	$O(n + k)$	$O(k)$	Out-place	稳定
桶排序	$O(n + k)$	$O(n + k)$	$O(n^2)$	$O(n + k)$	Out-place	稳定
基数排序	$O(n \times k)$	$O(n \times k)$	$O(n \times k)$	$O(n + k)$	Out-place	稳定

稳定性定义：排序前后两个相等的数相对位置不变，则算法稳定。

稳定性得好处：从一个键上排序，然后再从另一个键上排序，第一个键排序的结果可以为第二个键排序所用。

一、冒泡排序

- 比较相邻的元素。如果第一个比第二个大，就交换他们两个。
- 对每一对相邻元素作同样的工作，从开始第一对到结尾的最后一对。这步做完后，最后的元素会是最大的数。
- 针对所有的元素重复以上的步骤，除了最后一个。
- 持续每次对越来越少的元素重复上面的步骤，直到没有任何一对数字需要比较。

```
#include <iostream>
#include <ctime>
using namespace std;
//基础版本
void bubble_sort(int*p,int num)
```

```
{
    for(int i=0;i<num;i++)
    {
        for(int j=1;j<num;j++)
        {
            if(p[j-1]>p[j])
            {
                int temp=p[j-1];
                p[j-1]=p[j];
                p[j]=temp;
            }
        }
    }
}
//加强版1：增加flag标识，没有发生交换就停止
void bubble_sort_1(int*p,int num)
{
    bool flag=true;
    while(flag)
    {
        flag=false;
        for(int j=1;j<num;j++)
        {
            if(p[j-1]>p[j])
            {
                int temp=p[j-1];
                p[j-1]=p[j];
                p[j]=temp;
                flag=true;
            }
        }
    }
}

//加强版2：增加flag标识,记录最新不需要发生交换的位置，能减少交换次数。
void bubble_sort_2(int*p,int num)
{
    int len=num;
    while(len>0)
    {
        int flag=0;
        for(int j=1;j<len;j++)
        {
            if(p[j-1]>p[j])
            {
                int temp=p[j-1];
                p[j-1]=p[j];
                p[j]=temp;
                flag=j;
            }
        }
        len=flag;
    }
}
int main() {
    int a[]={8,7,6,5,4,3,2,1,10,11,12,13,14,15,16,17,19,19,20};
```

```
    int len=sizeof(a)/4;
    clock_t startTime,endTime;
    startTime = clock();//计时开始
//    bubble_sort(a, len);
//    bubble_sort_1(a, len);
    bubble_sort_2(a, len);
    endTime = clock();//计时开始
    cout<<"算法执行持续时间: "<<(double)(endTime - startTime) / CLOCKS_PER_SEC<<"秒"<<endl;
    for(int i=0;i<len;i++)
    {
        cout<<a[i]<<' ';
    }
    return 0;
}
/*
 * 时间输出为:
 * 算法执行持续时间: 4e-06秒
 * 1 2 3 4 5 6 7 8 10 11 12 13 14 15 16 17 19 19 20
 *
 * 算法执行持续时间: 3e-06秒
 * 1 2 3 4 5 6 7 8 10 11 12 13 14 15 16 17 19 19 20
 *
 * 算法执行持续时间: 2e-06秒
 * 1 2 3 4 5 6 7 8 10 11 12 13 14 15 16 17 19 19 20
 *
 */
```

二、选择排序

- 首先在未排序序列中找到最小（大）元素，存放到排序序列的起始位置
- 再从剩余未排序元素中继续寻找最小（大）元素，然后放到已排序序列的末尾。
- 重复第二步，直到所有元素均排序完毕。

```
#include <iostream>
#include <vector>

using namespace std;

// 简单选择排序
vector<int> SelectSort(vector<int> list){
    // 需要遍历获得最小值的次数
    // 要注意一点，当要排序 N 个数，已经经过 N-1 次遍历后，已经是有序数列
    vector<int> result = list;
    for (int i = 0; i < result.size(); i++){
        // 用来保存最小值得索引
        int index = i;
        // 每轮需要比较N-i次
        for (int j = i + 1; j < result.size(); j++){
            if (result[index] > result[j]){
                index = j;//记录当前最小值的下标。
            }
        }
        if (index == i){//当前位置就是最小值
```

```
            continue;
        }
        // 将找到的第i个小的数值放在第i个位置上
        swap(result[i], result[index]);
    }
    return result;
}

int main(){
    int arr[] = { 6, 4, 8, 9, 2, 3, 1 };
    vector<int> test(arr, arr + sizeof(arr) / sizeof(arr[0]));
    cout << "排序前" << endl;
    for (int i = 0; i < test.size(); i++){
        cout << test[i] << " ";
    }
        cout << endl;
    vector<int> result;
    result = SelectSort(test);
    cout << "排序后" << endl;
    for (int i = 0; i < result.size(); i++){
        cout << result[i] << " ";
    }
    cout << endl;
    system("pause");
    return 0;
}

```

三、插入排序

- 将第一待排序序列第一个元素看做一个有序序列，把第二个元素到最后一个元素当成是未排序序列。
- 从头到尾依次扫描未排序序列，将扫描到的每个元素插入有序序列的适当位置。（如果待插入的元素与有序序列中的某个元素相等，则将待插入元素插入到相等元素的后面。）

```
#include <iostream>
using namespace std;
//基础版本
void insert_sort(int *p,int num)
{
    for(int i=1;i<num;i++)
    {
        int temp=p[i];
        int j=i-1;
        for(;j>=0&&temp<p[j];j--)//注意判断条件
        {
            p[j + 1] = p[j];
        }
        p[j+1]=temp;
    }
}

//增强版（二叉查插入排序）：在查找每个元素时使用二分查找
void insert_sort_1(int *p,int num)
{
```

```
    for(int i=1;i<num;i++)
    {
        //二分查找，减少查找过程中的比较次数
        int left=0,right=i-1,m=-1;
        while (left<=right)
        {
            m=(left+right)/2;
            if(p[m]>p[i])
            {
                right=m-1;
            }
            else
            {
                left=m+1;
            }
        }
        //统一移动
        int temp=p[i];
        for(int j=i-1;j>=right+1;j--)//注意这里的判断条件是right+1,因为不需要插入到i前面时，right=i-1,
                                     // 此时不需要移动只需要令p[i]=temp即可，也就是p[right+1]=temp;
        {
            p[j+1]=p[j];
        }
        p[right+1]=temp;
    }
}
int main() {
    int a[]={8,7,6,5,4,3,2,1,10,11,12,13,14,15,16,17,19,19,20};
    int len=sizeof(a)/4;
    clock_t startTime,endTime;
    startTime = clock();//计时开始
//    bubble_sort(a, len);
//    bubble_sort_1(a, len);
    insert_sort_1(a, len);
    endTime = clock();//计时开始
    cout<<"算法执行持续时间: "<<(double)(endTime - startTime) / CLOCKS_PER_SEC<<"秒"<<endl;
    for(int i=0;i<len;i++)
    {
        cout<<a[i]<<' ';
    }
    return 0;
}
```

四、希尔排序

- 选择一个增量序列 t1，t2，……，tk，其中 ti > tj, tk = 1；
- 按增量序列个数 k，对序列进行 **k 趟排序**；
- 每趟排序，根据对应的增量 ti，将待排序列分割成若干长度为 m 的子序列，分别对各子表进行直接插入排序。仅增量因子为 1 时，整个序列作为一个表来处理，表长度即为整个序列的长度。

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

```cpp
void shell_sort(int *a,int num)
{
    if(a==NULL||num<2)
        return;
    int d=num/2;

    while (d>0)
    {
        for(int i=0;i<num;i++)
        {
            if(i+d<num)
                if(a[i]>a[i+d])swap(a[i],a[i+d]);
        }
        d/=2;
    }
}

int main() {
    int a[]={8,7,6,5,4,3,2,1,10,11,12,13,14,15,16,17,19,19,20};
    int len=sizeof(a)/4;
    clock_t startTime,endTime;
    startTime = clock();//计时开始
    shell_sort(a, len);
    endTime = clock();//计时开始
    cout<<"算法执行持续时间: "<<(double)(endTime - startTime) / CLOCKS_PER_SEC<<"秒"<<endl;
    for(int i=0;i<len;i++)
    {
        cout<<a[i]<<' ';
    }
    return 0;
}
/*
算法执行持续时间: 2e-06秒
1 2 3 4 5 6 7 8 10 11 12 13 14 15 16 17 19 19 20
*/
```

五、归并排序

- 申请空间，使其大小为两个已经排序序列之和，该空间用来存放合并后的序列；
- 设定两个指针，最初位置分别为两个已经排序序列的起始位置；
- 比较两个指针所指向的元素，选择相对小的元素放入到合并空间，并移动指针到下一位置；
- 重复步骤 3 直到某一指针达到序列尾；
- 将另一序列剩下的所有元素直接复制到合并序列尾。

```cpp
#include <iostream>

using namespace std;
int inverse_count=0;
void merge(int *arr,int l,int r,int m)
{
```

```
7
8       int l1=m-l+1;//第一个数组的元素个数
9       int l2=r-m;//第二个数组的元素个数
10      //两个中转数组，arr作为结果数组
11      int*a_l=new int[l1];
12      int *a_r=new int[l2];
13      for(int i=0;i<l1;i++)
14      {
15          a_l[i]=arr[i+l];
16      }
17      for(int j=0;j<l2;j++)
18      {
19          a_r[j]=arr[j+m+1];//注意，第二个数组是从mid+1开始的；
20      }
21      int i=0,j=0;
22      while (i<l1&&j<l2)
23      {
24          if(a_l[i]<a_r[j])//i对应前半部分数组；j对应后半部分数组
25          {
26              arr[l]=a_l[i];
27              l++;
28              i++;
29          }
30          else
31          {
32              arr[l]=a_r[j];
33              l++;
34              j++;
35          }
36      }
37
38      while (i<l1)
39      {
40          arr[l]=a_l[i];
41          l++;
42          i++;
43      }
44      while (j<l2)
45      {
46          arr[l]=a_r[j];
47          l++;
48          j++;
49      }
50      delete []a_l;
51      delete []a_r;
52      return;
53  }
54
55  void mergeSort(int *arr,int start,int end)
56  {
57
58      if(start<end)
59      {
60          int mid=(start+end)/2;
61          mergeSort(arr,start,mid);
62          mergeSort(arr,mid+1,end);
```

```
63          merge(arr,start,end,mid);
64      }
65
66
67  }
68
69  int main() {
70      int a[]={4,6,8,5,9};
71      for(int i=0;i<5;i++)
72      {
73          cout<<a[i]<<" ";
74      }
75      cout<<endl;
76      mergeSort(a,0,4);
77      for(int i=0;i<5;i++)
78      {
79          cout<<a[i]<<" ";
80      }
81      std::cout << "Hello, World!" << std::endl;
82      return 0;
83  }
84  /*
85   * output:
86   * 4 6 8 5 9
87   * 4 5 6 8 9
88   */
```

java版本，辅助数组为结果数组，不需要创建两个中专数组

```
1  1//Java 代码实现
2      public class MergeSort implements IArraySort {
3  2
4  3    @Override
5  4    public int[] sort(int[] sourceArray) throws Exception {
6  5        // 对 arr 进行拷贝，不改变参数内容
7  6        int[] arr = Arrays.copyOf(sourceArray, sourceArray.length);
8  7
9  8        if (arr.length < 2) {
10 9            return arr;
11 10       }
12 11       int middle = (int) Math.floor(arr.length / 2);
13 12
14 13       int[] left = Arrays.copyOfRange(arr, 0, middle);
15 14       int[] right = Arrays.copyOfRange(arr, middle, arr.length);
16 15
17 16       return merge(sort(left), sort(right));
18 17   }
19 18
20 19   protected int[] merge(int[] left, int[] right) {
21 20       int[] result = new int[left.length + right.length];
22 21       int i = 0;
23 22       while (left.length > 0 && right.length > 0) {
24 23           if (left[0] <= right[0]) {
25 24               result[i++] = left[0];
26 25               left = Arrays.copyOfRange(left, 1, left.length);
27 26           } else {
28 27               result[i++] = right[0];
```

```
                right = Arrays.copyOfRange(right, 1, right.length);
            }
        }

        while (left.length > 0) {
            result[i++] = left[0];
            left = Arrays.copyOfRange(left, 1, left.length);
        }

        while (right.length > 0) {
            result[i++] = right[0];
            right = Arrays.copyOfRange(right, 1, right.length);
        }

        return result;
    }

}
```

六、快速排序

- 从数列中挑出一个元素，称为“基准”（pivot）；
- 重新排序数列，所有元素比基准值小的摆放在基准前面，所有元素比基准值大的摆在基准的后面（相同的数可以到任一边）。在这个分区退出之后，该基准就处于数列的中间位置。这个称为分区（partition）操作；
- 递归地（recursive）把小于基准值元素的子数列和大于基准值元素的子数列排序；

伪代码：

```
//伪代码:
int partition(int* a,int left,int right)
{
  if(a==NULL||left>=right)return left;
  int base_idx=left;
  int i,j;
  while(left<right)
  {
    i=find_bigger_base();
    j=find_smaller_base();
    swap(a[i],a[j]);
  }
  if(a[base_idx]>a[j])swap(a[base],a[j]);
}
qsort(int*a,int left,int right)
{
  int base_idx=partition();
  qsort(a,left,base_idx-1);
  qsort(a,base_idx+1,right);
}

```

基础版本（c++）

```
void swap(int *a,int i,int j)
{
    int temp=a[j];
    a[j]=a[i];
    a[i]=temp;
}

void quickSort(int *a,int left,int right)
{
    if(left<right)
    {
        int i=left,j=right-1,base=right;
        while (i<j)
        {
            while(a[i]<a[base])i++;//// 从左向右找第一个大于等于x的数
            while (a[j]>=a[base])j--;//从右向左找第一个小于x的数
            if(i<j) {
                swap(a, i, j);
            }
        }
        if(a[i]>a[base])swap(a,i,base);
        quickSort(a,left,i-1);
        quickSort(a,i+1,right);
    }
}
```

基础版本（java）

```
//Java 代码实现
 2public class QuickSort implements IArraySort {
 3
 4    @Override
 5    public int[] sort(int[] sourceArray) throws Exception {
 6        // 对 arr 进行拷贝，不改变参数内容
 7        int[] arr = Arrays.copyOf(sourceArray, sourceArray.length);
 8
 9        return quickSort(arr, 0, arr.length - 1);
10    }
11
12    private int[] quickSort(int[] arr, int left, int right) {
13        if (left < right) {//在这里判断也可以。
14            int partitionIndex = partition(arr, left, right);
15            quickSort(arr, left, partitionIndex - 1);
16            quickSort(arr, partitionIndex + 1, right);
17        }
18        return arr;
19    }
20
21    private int partition(int[] arr, int left, int right) {
22        // 设定基准值（pivot）
23        int pivot = left;
24        int index = pivot + 1;
25        for (int i = index; i <= right; i++) {//把小于基准的值都调到前面去，后面自然而然就是大于基
准的。
26            if (arr[i] < arr[pivot]) {
27                swap(arr, i, index);
```

```
28                index++;
29            }
30        }
31        swap(arr, pivot, index - 1);
32        return index - 1;
33    }
34
35    private void swap(int[] arr, int i, int j) {
36        int temp = arr[i];
37        arr[i] = arr[j];
38        arr[j] = temp;
39    }
40}
```

改进版本1：快排+插排

```
1  void insert_sort(int *arr,int left,int right)
2  {
3      for(int i=left+1;i<=right;i++)
4      {
5          int j=i-1;
6          int temp=arr[i];
7          while (j>=0&&temp<arr[j])
8          {
9              arr[j+1]=arr[j];
10             j--;
11         }
12         arr[j+1]=temp;
13     }
14 }
15
16 int partition(int *arr,int left,int right)
17 {
18     int base=arr[left];
19
20     int i=left+1,j=right;
21
22     while(i<j)
23     {
24         while(arr[i]<=base)
25         {
26             i++;
27         }
28
29         while(arr[j]>base)
30         {
31             j--;
32         }
33         if(i<j)
34         {
35             swap(arr[i],arr[j]);
36         }
37     }
38     if(arr[j]<arr[left])
39         swap(arr[left],arr[j]);
40     return j;
41 }
```

```

void qsort_k(int * arr,int left,int right,int k,int delta=5)
{
    if(left>=right)return;

    if(right-left+1<delta)
    {
        insert_sort(arr,left,right);
        return ;
    }
    int mid=partition(arr,left,right);
    if(mid<right)
    {
        qsort_k(arr,mid+1,right,k,delta);
    }
    if(mid>left)
    {
        qsort_k(arr,left,mid-1,k,delta);
    }
}

```

改进2：三路快排

```
void QuickSort(int arr[], int left, int right) {
    assert(arr);
    if (left >= right) {
        return;
    }
    int l = left - 1, r = right, value = arr[right];
    int p = left - 1;
    int q = right;
    while (1) {
        //向左向右扫描找到不小于value的值
        while (arr[++l] < value) { ; }
        //从右向左扫描找到不大于value的值
        while (arr[--r] > value) { if(r == left) break; }
        if (l >= r) {
            break;
        }
        swap(arr[l], arr[r]);
        //如果arr[l]的值和value是相等的，就把它放入到数组左边
        if (arr[l]== value) {
            p++;
            swap(arr[p], arr[l]);
        }
        //同理，把和value相等的值放入到数组右边
        if (arr[r]==value) {
            q--;
            swap(arr[q], arr[r]);
        }
    }
    /*上面的循环找到了需要的元素,然后交换*/
    swap(arr[l], arr[right]);
    r = l - 1; l = l + 1;
    int k = 0;
    //把相等的元素都交换到数组的中间
```

```
    for (k = left; k <= p; ++k, --r) {
        swap(arr[k], arr[r]);
    }
    for (k = right - 1; k >= q; --k, ++l) {
        swap(arr[k], arr[l]);
    }
    QuickSort(arr, left, r);
    QuickSort(arr, l, right);
}
```

扩展：第k大的数字

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

int partition(int *arr,int left,int right)
{
    int base=arr[left];

    int i=left+1,j=right;

    while(i<j)
    {
        while(arr[i]<=base)
        {
            i++;
        }

        while(arr[j]>base)
        {
            j--;
        }
        if(i<j)
        {
            swap(arr[i],arr[j]);
        }
    }
    if(arr[j]<arr[left])
        swap(arr[left],arr[j]);
    return j;
}

int qsort_k(int * arr,int left,int right,int k)
{
    if(left>=right)return arr[left];
    int mid=partition(arr,left,right);
    if(k==mid+1)return arr[mid];
    else if(k>mid)
    {
        return qsort_k(arr,mid+1,right,k);
    }
    else
    {
        return qsort_k(arr,left,mid-1,k);
    }
}
int main()
```

```
{
    int a[]={6,7,8,5,4,3,2,1};
    for(int i=0;i<8;i++)
    {
        cout<<a[i]<<" ";
    }
    cout<<endl;
    cout<<qsort_k(a,0,7,5)<<endl;

    for(int i=0;i<8;i++)
    {
        cout<<a[i]<<" ";
    }
    return 0;
}
```

七、堆排序

详解可参见博客：https://narcissuscyn.github.io/2019/01/23/堆排序的深入理解/

- 创建一个堆 H[0……n-1]；
- 把堆首（最大值）和堆尾互换；
- 把堆的尺寸缩小 1，并调用 shift_down(0)，目的是把新的数组顶端数据调整到相应位置；
- 重复步骤 2，直到堆的尺寸为 1。

模板基类

```
//
// Created by Narcissus Chen on 2019-01-22.
//

#ifndef SORT_HEAP_H
#define SORT_HEAP_H

enum{DEFAULT_SIZE=10};
template <class T>
class Heap
{
public:
    Heap()=default;
    virtual ~Heap()= default;
    virtual void show_heap()const =0;
    virtual bool insert_heap(const T&)=0;
    virtual bool  remove_heap(T&)=0;
    virtual void heap_sort()=0;
};
#endif //SORT_HEAP_H
```

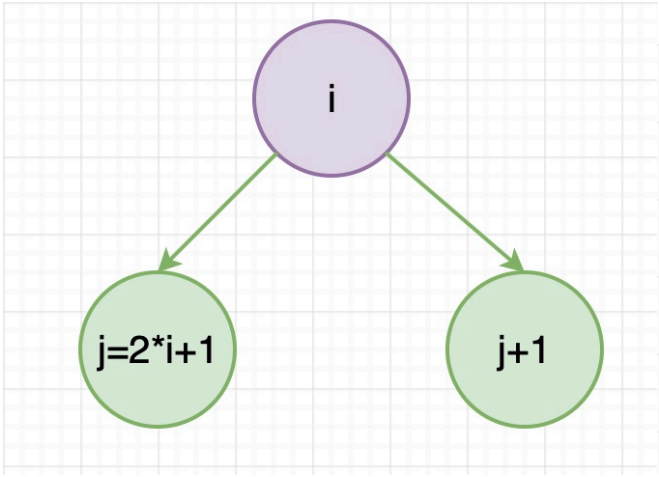
最大堆头文件

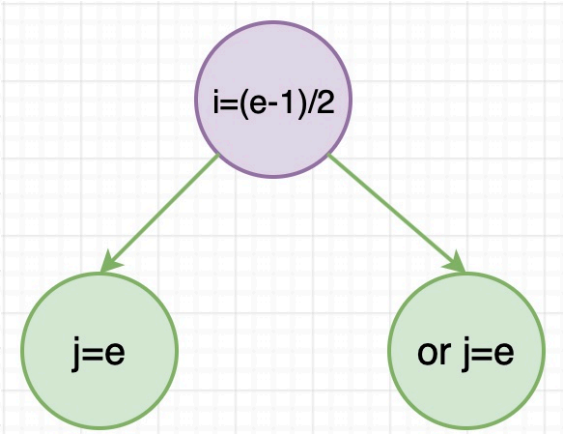
```
//
// Created by Narcissus Chen on 2019-01-22.
//
```

```
4
5  #ifndef SORT_MAXHEAP_H
6  #define SORT_MAXHEAP_H
7
8  #include "heap.h"
9  class MaxHeap: public Heap<int> {
10 public:
11     /**
12      * 创建一个空堆
13      */
14     MaxHeap(int sz=DEFAULT_SIZE);
15     /**
16      * 根据已有数组堆创建一个堆
17      */
18     MaxHeap(const int[],const int );
19     /**
20      * 析构函数
21      */
22     ~MaxHeap();
23     /**
24      * 显示堆
25      */
26     void show_heap()const;
27     /**
28      * 向堆中插入元素
29      * @return
30      */
31     bool insert_heap(const int&);
32     /**
33      * 移除堆中的元素
34      * @return
35      */
36     bool  remove_heap(int&);
37     void heap_sort();
38 protected:
39     /**
40      * 下浮
41      */
42     void shift_down(int,int);
43     /**
44      * 上浮
45      */
46     void shift_up(int);
47
48 private:
49     //指向堆的指针
50     int *heap;
51     //已有堆中元素个数
52     int size;
53     //堆的容量
54     int capacity;
55
56 };
57 #endif //SORT_MAXHEAP_H
```

最大堆实现

完全二叉树的坐标关系：（下沉、上浮时用到的坐标关系）






```
//
// Created by Narcissus Chen on 2019-01-22.
//
#include <iostream>
#include <form.h>
#include "MaxHeap.h"
using namespace std;
MaxHeap::~MaxHeap() {
    delete heap;
    heap= nullptr;
    capacity=size=0;
}

MaxHeap::MaxHeap(int sz) {
    capacity=sz>DEFAULT_SIZE?sz:DEFAULT_SIZE;
    heap=new int[capacity];
    assert(heap!= nullptr);
    size=0;
}

MaxHeap::MaxHeap(const int arr[],const int arrSize)
{
    capacity=arrSize>DEFAULT_SIZE?arrSize:DEFAULT_SIZE;
    heap=new int[capacity];
    size=arrSize;
    for(int i=0;i<arrSize;i++)
    {
        heap[i]=arr[i];
    }
    int curPos=size/2-1;
    while(curPos>=0)
    {
        shift_down(curPos,arrSize-1);//下沉操作
        curPos--;
    }
}

void MaxHeap::shift_down(int start, int end) {
    int i=start;//第i个非叶子节点
    int j=start*2+1;//第i个非叶子节点的左儿子
```

```
    int temp=heap[i];
    while (j<=end)//是否继续进行往下沉的条件
    {
        if(j+1<=end&&heap[j]<heap[j+1])//找到两个儿子节点更大的一个
        {
            j++;
        }
        if(temp>=heap[j])break;//比最大的儿子节点要大，表示此节点i已经满足堆的条件，不需要继续往下沉；
        //否则，会继续往下判断
        heap[i]=heap[j];
        i=j;//新的节点
        j=2*i+1;//新节点的左子节点
    }
    heap[i]=temp;//完成两个节点的交换，也可能是heap[i]和自身的赋值,比如在while循环的第一遍循环就break出去。
}

void MaxHeap::shift_up(int e) {

    int j=e;//j为最后新加入的元素
    int i=(e-1)/2;//j的父节点
    int temp=heap[j];
    while (j>=0)
    {
        if(temp<heap[i])//插入的时候就已经满足堆的条件
        {
            break;
        } else {//插入时不满足
            heap[j]=heap[i];
        }
        j=i;//j指向其父节点
        i=(i-1)/2;//j为新j的父节点
    }
    heap[j]=temp;//完成交换
}

bool MaxHeap::insert_heap(const int &val) {
    if(size>=capacity)return false;
    heap[size]=val;
    shift_up(size);
    size+=1;
    return true;
}

bool MaxHeap::remove_heap(int &val) {
    if(size<=0)return false;
    val=heap[0];
    heap[0]=heap[size-1];
    --size;
    shift_down(0,size-1);//这里root就是数组的第一个元素。
    // 所以从0开始；而在初始建堆的时候是从最后一个非叶子节点开始下沉的，
    // 这样是为了保证在下沉每个节点时，其左右子树都满足堆的条件。
}

void MaxHeap::show_heap() const {
    for(int i=0;i<size;i++)
```

```
    {
        std::cout<<heap[i]<<" ";
    }
    cout<<endl;
}

void MaxHeap::heap_sort() {
    for(int i=size-1;i>0;i--)
    {
        int temp=heap[0];
        heap[0]=heap[i];
        heap[i]=temp;
        shift_down(0,i-1);
    }
}

int main()
{
    int a[]={4,6,8,5,9};
    //初始建堆
    MaxHeap h(a,5);
    h.show_heap();
    //堆排序
    h.heap_sort();
    h.show_heap();
    return 0;
}
```

八、计数排序-只适用于整数排序、数据值比较集中的数据。

- 花O(n)的时间扫描一下整个序列 A，获取最小值 min 和最大值 max
- 开辟一块新的空间创建新的数组 B，长度为 (max - min + 1)
- 数组 B 中 index 的元素记录的值是 A 中某元素出现的次数
- 最后输出目标整数序列，具体的逻辑是遍历数组 B，输出相应元素以及对应的个数

```
#include<iostream>
using namespace std;
void count_sort(int*a,int num)
{
    if(a==NULL||num==1)return ;
    int min_=a[0],max_=a[0];
    for(int i=1;i<num;i++)
    {
        if(a[i]>max_)max_=a[i];
        if(a[i]<min_)min_=a[i];
    }
    int *count=new int[max_+1];
//    for (int i=0;i<=max_;i++)count[i]=0;
    memset(count,0, (max_+1)* sizeof(int));

    for(int i=0;i<num;i++)count[a[i]]++;
    int idx=0;
```

```
    for(int i=0;i<=max_;i++)
    {
        for(int j=0;j<count[i];j++)
        {
            a[idx]=i;
            idx++;
        }
    }
    delete[] count;
}

int main() {
    int a[]={4,6,8,5,9};
    for(int i=0;i<5;i++)
    {
        cout<<a[i]<<" ";
    }
    cout<<endl;
    count_sort(a,5);
    for(int i=0;i<5;i++)
    {
        cout<<a[i]<<" ";
    }
    cout<<endl;
    std::cout << "Hello, World!" << std::endl;
    return 0;
}
```

九、桶排序

- 设置固定数量的空桶。
- 把数据放到对应的桶中。
- 对每个不为空的桶中数据进行排序。
- 拼接不为空的桶中数据，得到结果

```
#include<iostream>
#include <vector>
using namespace std;
void insert_sort(vector<int>&p ,int num)
{
    for(int i=1;i<num;i++)
    {
        int temp=p[i];
        int j=i-1;
        for(;j>=0&&temp<p[j];j--)//注意判断条件
        {
            p[j + 1] = p[j];
        }
        p[j+1]=temp;
    }
}
/**
```

```
 * a:待排序数组
 * num:数组个数
 * bucket_size:每个桶中元素的个数
**/
void bucket_sort(int*a,int num,int bucket_size)
{
    if(a==NULL||num==1)return ;
    int min_=a[0],max_=a[0];
    for(int i=1;i<num;i++)
    {
        if(a[i]>max_)max_=a[i];
        if(a[i]<min_)min_=a[i];
    }
    int bucket_count=(max_-min_)/bucket_size+1;//桶的个数
    vector<int>* p=new vector<int>[bucket_count];
    for(int i=0;i<bucket_size;i++)
    {
        p[i]=vector<int>();
    }
    // 利用映射函数将数据分配到各个桶中
    for (int i = 0; i < num; i++) {
        int index = (a[i] - min_) / bucket_size;//求桶坐标
        p[index].push_back(a[i]);
    }
    int idx=0;
    for(int i=0;i<bucket_size;i++)
    {
        insert_sort(p[i],p[i].size());//桶内排序。
        for(int j=0;j<p[i].size();j++)//将桶内元素放到原数组中去。
        {
            a[idx]=p[i][j];
            idx++;
        }
    }
}

int main() {
    int a[]={4,6,8,5,9,10,19,13,11,39};
    int count= sizeof(a)/ sizeof(int);
    for(int i=0;i<count;i++)
    {
        cout<<a[i]<<" ";
    }
    cout<<endl;
    bucket_sort(a,count,5);
    for(int i=0;i<count;i++)
    {
        cout<<a[i]<<" ";
    }
    cout<<endl;
    std::cout << "Hello, World!" << std::endl;
    return 0;
}
/*
 * 4 6 8 5 9 10 19 13 11 39
 *4 5 6 8 9 10 11 13 19 39
```

```
*/
```

十、基数排序

- 将所有待比较数值（正整数）统一为同样的数位长度，数位较短的数前面补零
- 从最低位开始，依次进行一次排序
- 从最低位排序一直到最高位排序完成以后, 数列就变成一个有序序列

```cpp
#include <iostream>
#include <vector>

using namespace std;

// 求出数组中最大数的位数的函数
int MaxBit(vector<int> input){
    // 数组最大值
    int max_data = input[0];
    for (int i = 1; i < input.size(); i++){
        if (input[i] > max_data){
            max_data = input[i];
        }
    }

    // 数组最大值的位数
    int bits_num = 0;
    while (max_data){
        bits_num++;
        max_data /= 10;
    }

    return bits_num;
}

// 取数xxx上的第d位数字
int digit(int num, int d){
    int pow = 1;
    while (--d > 0){
        pow *= 10;
    }
    return num / pow % 10;
}

// 基数排序
vector<int> RadixSort(vector<int> input, int n){
    // 临时数组，用来存放排序过程中的数据
    vector<int> bucket(n);
    // 位记数器，从第0个元素到第9个元素依次用来记录当前比较位是0的有多少个...是9的有多少个数
    vector<int> count(10);
    // 从低位往高位循环
    for (int d = 1; d <= MaxBit(input); d++){
        // 计数器清0
        for (int i = 0; i < 10; i++){
            count[i] = 0;
```

```
        }

        // 统计各个桶中的个数
        for (int i = 0; i < n; i++){
            count[digit(input[i],d)]++;
        }

        /*
        * 比如某次经过上面统计后结果为：[0, 2, 3, 3, 0, 0, 0, 0, 0, 0]则经过下面计算后 结果为： [0, 2,
        * 5, 8, 8, 8, 8, 8, 8, 8]但实质上只有如下[0, 2, 5, 8, 0, 0, 0, 0, 0, 0]中
        * 非零数才用到，因为其他位不存在，它们分别表示如下：2表示比较位为1的元素可以存放在索引为1、0的
        * 位置，5表示比较位为2的元素可以存放在4、3、2三个(5-2=3)位置，8表示比较位为3的元素可以存放在
        * 7、6、5三个(8-5=3)位置
        */
        for (int i = 1; i < 10; i++){
            count[i] += count[i - 1];
        }

        /*
        * 注，这里只能从数组后往前循环，因为排序时还需保持以前的已排序好的顺序，不应该打
        * 乱原来已排好的序，如果从前往后处理，则会把原来在前面会摆到后面去，因为在处理某个
        * 元素的位置时，位记数器是从大到到小（count[digit(arr[i], d)]--）的方式来处
        * 理的，即先存放索引大的元素，再存放索引小的元素，所以需从最后一个元素开始处理。
        * 如有这样的一个序列[212,213,312]，如果按照从第一个元素开始循环的话，经过第一轮
        * 后（个位）排序后，得到这样一个序列[312,212,213]，第一次好像没什么问题，但问题会
        * 从第二轮开始出现，第二轮排序后，会得到[213,212,312]，这样个位为3的元素本应该
        * 放在最后，但经过第二轮后却排在了前面了，所以出现了问题
        */
        for (int i = n - 1; i >= 0; i--){
            int k = digit(input[i], d);
            bucket[count[k] - 1] = input[i];
            count[k]--;//保证索引减小
        }

        // 临时数组复制到 input 中
        for (int i = 0; i < n; i++){
            input[i] = bucket[i];
        }
    }

    return input;
}

int main(){
    int arr[] = { 50, 123, 543, 187, 49, 30, 0, 2, 11, 100 };
    vector<int> test(arr, arr + sizeof(arr) / sizeof(arr[0]));
    cout << "排序前:";
    for (int i = 0; i < test.size(); i++){
        cout << test[i] << " ";
    }
    cout << endl;

    vector<int> result = test;
    result = RadixSort(result, result.size());
    cout << "排序后:";
```

```
    for (int i = 0; i < result.size(); i++){
        cout << result[i] << " ";
    }
    cout << endl;
    system("pause");
    return 0;
}
```