数组10 大排序

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(n²)	O(n)	O(n²)	O(1)	In-place	稳定
选择排序	O(n²)	O(n²)	O(n²)	O(1)	In-place	不稳定
插入排序	O(n²)	O(n)	O(n²)	O(1)	In-place	稳定
希尔排序	O(n log n)	O(n log² n)	O(n log² 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(n²)	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(n²)	O(n + k)	Out-place	稳定
基数排序	O(n × k)	O(n × k)	O(n × k)	O(n + k)	Out-place	稳定

名称	数据对象	稳定性	时间复杂度		额外空间复杂度	描述
			平均	最坏		
冒泡排序	数组	✓	$O(n^2)$		$O(1)$	（无序区，有序区）。 从无序区透过交换找出最大元素放到有序区前端。
选择排序	数组	✗	$O(n^2)$		$O(1)$	（有序区，无序区）。 在无序区里找一个最小的元素跟在有序区的后面。对数组：比较得多，换得少。
	链表	✓				
插入排序	数组、链表	✓	$O(n^2)$		$O(1)$	（有序区，无序区）。 把无序区的第一个元素插入到有序区的合适的位置。对数组：比较得少，换得多。
堆排序	数组	✗	$O(n \log n)$		$O(1)$	（最大堆，有序区）。 从堆顶把根卸出来放在有序区之前，再恢复堆。
归并排序	数组	✓	$O(n \log^2 n)$		$O(1)$	把数据分为两段，从两段中逐个选最小的元素移入新数据段的末尾。 可从上到下或从下到上进行。
			$O(n \log n)$		$O(n) + O(\log n)$ 如果不是从下到上	
	链表				$O(1)$	
快速排序	数组	✗	$O(n \log n)$	$O(n^2)$	$O(\log n)$	（小数，基准元素，大数）。 在区间中随机挑选一个元素作基准，将小于基准的元素放在基准之前，大于基准的元素放在基准之后，再分别对小数区与大数区进行排序。
希尔排序	数组	✗	$O(n \log^2 n)$	$O(n^2)$	$O(1)$	每一轮按照事先决定的间隔进行插入排序，间隔会依次缩小，最后一次一定要是1。
计数排序	数组、链表	✓	$O(n + m)$		$O(n + m)$	统计小于等于该元素值的元素的个数i，于是该元素就放在目标数组的索引i位（i≥0）。
桶排序	数组、链表	✓	$O(n)$		$O(m)$	将值为i的元素放入i号桶，最后依次把桶里的元素倒出来。
基数排序	数组、链表	✓	$O(k \times n)$	$O(n^2)$		一种多关键字的排序算法，可用桶排序实现。

选择排序

时间复杂度 `O(n^2)`

空间复杂度 `O(1)`

不稳定

比较次数`1 + 2 + 3 +...+ n-1 = n(n-1)/2`

移动次数 `3*n(n-1)/2`

```
void select_sort(vector<int> &arr) {
    for (int i = 0; i < arr.size(); ++i) {
        int min = i;
        for (int j = i + 1; j < arr.size(); ++j) {
            if (arr[j] <= arr[min]) {
                min = j;
            }
        }
        swap(arr[i], arr[min]);
    }
}
```

冒泡排序

时间复杂度 `O(n^2)`

空间复杂度 `O(1)`

稳定

最好情况

- `123456`
- 比较次数 `n-1`
- 移动次数`0`
- 时间复杂度 `O(n)`

最坏情况

- `654321`
- 比较次数`1 + 2 + 3 +...+ n-1 = n(n-1)/2`
- 移动次数 `3*n(n-1)/2`
- 时间复杂度 `O(n^2)`

```
void swap_xor(vector<int>::value_type &value1, vector<int>::value_type &value2) {
    value1 = value1 ^ value2;
    value2 = value1 ^ value2;
    value1 = value1 ^ value2;
};

void bubble_sort(vector<int> &arr) {
    if (arr.empty() || arr.size() < 2) return;
    bool flag = false;
    for (int e = arr.size() - 1; e >= 0; --e) {
        for (int i = 0; i < e; ++i) {
            if (arr[i] > arr[i + 1]) {
                swap(arr[i], arr[i + 1]);
                flag = true;
            }
        }
        if (!flag) break;
    }
}
```

插入排序

时间复杂度 `O(n^2)`

空间复杂度 `O(1)`

稳定

最好情况

- `123456`
- 比较次数 `n-1`
- 移动次数 `0`
- 时间复杂度 `O(n)`

最坏情况

- `654321`
- 比较次数 `1 + 2 + 3 +...+ n-1 = n(n-1)/2`
- 移动次数 `2+3+4+...+n=(n+2)(n-1)/2`
- 时间复杂度 `O(n^2)`

```
    void insertSort(vector<int> &arr) {
        for (int i = 1; i < arr.size(); ++i) {
            for (int j = i-1; j >=0 && arr[j] > arr[j+1] ; j--) {
                swap(arr[j],arr[j+1]);
            }

        }
    }
```

希尔排序 `(while)`

```
void shell_sort1(vector<int> &arr) {
    int len = arr.size();
    int interval = len >> 1; // 获取初始长度
    while (interval >= 1) {
        for (int i = interval; i < len; ++i) {

            vector<int>::value_type tmp = arr[i];
            int j = i;
            while ((j - interval >= 0) && (arr[j - interval] > tmp)) {

                arr[j] = arr[j - interval];
                j -= interval;

            }
            arr[j] = tmp;
        }
        interval /= 2;
    }
}
```

希尔排序（for）

时间复杂度 `O(n^1.3~n^2)`

空间复杂度 `O(1)`

不稳定

```
// 4. 希尔排序 3 for
void shell_sort2(vector<int> &nums) {
    int len = nums.size();
    for (int gap = len / 2; gap > 0; gap /= 2) {
        for (int i = gap; i < len; i++) {
            for (int j = i - gap; j >= 0; j -= gap) {
                if (nums[j] > nums[j + gap]) {
                    swap(nums[j + gap], nums[j]);
                } else break;
            }
        }
    }
}
```

归并排序 (递归)

时间复杂度 `O(nlogn)`

空间复杂度 `O(n)`

稳定

```
    vector<int> tmp;

    void  merge_sort(vector<int> &arr){
        tmp.resize(arr.size());
        merge_sort(arr,0,arr.size()-1);
    }

    void merge(vector<int> & arr, int lo, int mid ,int hig) {
        for (int i = lo; i <=hig; ++i) {
            tmp[i]=arr[i];
        }
        int i=lo,j=mid+1;
        for (int k = lo; k <=hig; ++k) {
            if(i==mid+1){
                arr[k]=tmp[j++];
            }else if(j==hig+1){
                arr[k]=tmp[i++];
            }else if(tmp[i]<tmp[j]){
                arr[k]=tmp[i++];
            }else{
                arr[k]=tmp[j++];

            }
```

```
        }

    }

    void merge_sort(vector<int> &arr, int lo, int hig) {
        if (lo >= hig) return;
        int mid = lo + (hig - lo) / 2;
        merge_sort(arr, lo, mid);
        merge_sort(arr, mid + 1, hig);
        merge(arr, lo, mid,hig);
    }
```

归并排序 (迭代)

时间复杂度 `O(nlogn)`

空间复杂度 `O(n)`

稳定

```
// 5. 归并排序 （迭代）
void merge_sort2(vector<int> & arr) {
    int len = arr.size();
    vector<int> tmp(len,0);

    for (int seg = 1; seg < len; seg += seg) {
        for (int start = 0; start < len; start += seg + seg) {
            int low = start, mid = min(start + seg, len);
            int high = min(start + seg + seg, len);
            int k = low;
            int start1 = low, end1 = mid;
            int start2 = mid, end2 = high;

            while (start1 < end1 && start2 < end2) {
                tmp[k++] = arr[start1] < arr[start2] ? arr[start1++] : arr[start2++];
            }
            while (start1 < end1) {
                tmp[k++] = arr[start1++];
            }
            while (start2 < end2) {
                tmp[k++] = arr[start2++];
            }
        }
        copy(tmp.begin(), tmp.end(), arr.begin());
    }
}
```

快速排序

时间复杂度 `O(nlogn)`

空间复杂度 `O(1)`

不稳定

最好情况

- 每次选择的分界点都是最好的分界点
- 时间复杂度 `O(nlogn)`

最坏情况

- `654321` `123456`
- 时间复杂度 `O(n^2)`

```
int quick_sort_parition(vector<int> &arr, int  low, int  hig) {

    vector<int>::value_type pivot =arr[low];


    while (low < hig) {

        while (low < hig && arr[hig] >= pivot  ) {
            hig--;
        }

        arr[low]=arr[hig];
        while (low<hig && arr[low] <= pivot ) {
            low++;
        }
        arr[hig]=arr[low];
    }
    arr[low] = pivot;
    return low;

}
void quick_sort(vector<int> &arr,int low, int hig){
    if(low<hig){
        int pivot = quick_sort_parition(arr, low, hig);
        quick_sort(arr, low, pivot - 1);
        quick_sort(arr, pivot + 1, hig);
    }
}
```

记数排序

```
void count_sort(vector<int> &arr) {

    int len = arr.size();
    vector<int>::value_type maxx = *max_element(arr.begin(), arr.end());
    vector<int>::value_type minx = *min_element(arr.begin(), arr.end());
    vector<int> tmp(maxx - minx + 1, 0);
    vector<int> ans;
    for (int i = 0; i < len; ++i) {
        tmp[arr[i] - minx]++;
    }
    for (int j = 0; j < maxx - minx + 1; ++j) {
        while (tmp[j] != 0) {

            ans.push_back(j + minx);
            tmp[j]--;
```

```
        }

    }
    copy(ans.begin(), ans.end(), arr.begin());
}
```

桶排序

```
void insert_list_sort(vector<int> &arr) {
    int len = arr.size();
    for (int i = 1; i < len; ++i) {
        for (int j = i-1; j >=0 ; j--) {
            if(arr[j]>arr[j+1]){
                swap(arr[j],arr[j+1]);
            }
        }
    }
}

void bucket_sort(vector<int> & arr) {

    vector<int>::value_type maxx = *max_element(arr.begin(), arr.end());
    vector<int>::value_type minx = *min_element(arr.begin(), arr.end());
    const int bucket_size=maxx / 10 - minx / 10 + 1;;

    vector<vector<int>> bucket(bucket_size);  // 10个桶    或者使用 unordered_map 创建桶
    // 初始化空桶
    for (int i = 0; i < bucket_size; ++i) {
        vector<int> x{0};
        bucket.push_back(x);
    }
    for (int i = 0; i < arr.size(); ++i) {
        bucket[arr[i] / 10].push_back(arr[i]);
    }
    int index = 0;
    for (int i = 0; i < bucket_size; ++i) {
        // sort of bucket
        insert_list_sort(bucket[i]);

        for (auto it = bucket[i].begin(); it != bucket[i].end(); ++it) {
            arr[index++] = *it;

        }
    }
}
```

基数排序

`d`为操作的趟数

`n` 为分配的次数 也是元素的个数

`r` 为队列的个数 也是划分元素的子集个数 例如 `{0,1,2,3,4,5,6,7,8,9}`

时间复杂度 `O(d(n+r))`

空间复杂度 `O(r)`

稳定

```
void radix_sort(vector<int> &arr) {
    // get numberOfDigits  : numberOfDigits为位数
    vector<int>::value_type max = *max_element(arr.begin(), arr.end());
    int numberOfDigits = 0;
    while (max > 0) {
        max /= 10;
        numberOfDigits++;
    }
    const int BUCKETS = 10;
    vector<vector<int>> buckets(BUCKETS);
    for (int poss = 0; poss <= numberOfDigits - 1; ++poss) {
        int denominator = static_cast<int> (pow(10, poss));
        for (int &tmp: arr) {
            buckets[(tmp / denominator) % 10].push_back(tmp);

        }
        int index = 0;
        for (auto &thebuckett: buckets) {
            for (int &k: thebuckett) {
                arr[index++] = k;
                thebuckett.clear();
            }
        }
    }
}
```

堆排序

排序时间复杂度 `O(nlgn)` 调用`n`次`Heapify O(lgn)`

建堆时间复杂度 `O(nlgn)`

空间复杂度 `O(n)`

不稳定

```
class Solution {
public:




    void maxHeapify(vector<int> &arr, int i, int heapSize) {
        while (2*i+1<=heapSize){
            int leftChild=(2*i)+1,rightChild=(2*i)+2,target=leftChild;
            if(rightChild <=heapSize && arr[rightChild]> arr[leftChild]){
                target=rightChild;
            }
            if(arr[target] > arr[i]){
                swap(arr[target],arr[i]);
            }else{
```

```
                break;
            }
            i=target;
        }
    }

    void maxHeapify2(vector<int>& a, int i, int heapSize) {
        int l = i * 2 + 1, r = i * 2 + 2, largest = i;
        if (l <= heapSize && a[l] > a[largest]) {
            largest = l;
        }
        if (r <= heapSize && a[r] > a[largest]) {
            largest = r;
        }
        if (largest != i) {
            swap(a[i], a[largest]);
            maxHeapify2(a, largest, heapSize);
        }
    }




    void build_heap(vector<int> &arr, int size) {
        for (int i =  size/2; i >= 0; i--) {
            maxHeapify(arr, i, size);
        }
    }

    void heap_sort(vector<int> & arr) {
        int len=arr.size()-1;
        build_heap(arr, len);
        for (int i = len ; i >=1; i--) {
            swap(arr[i], arr[0]);
            len--;
            maxHeapify(arr, 0, len);
        }
    }


};



class Solution2 {
public:
    void minHeapify(vector<int> &arr, int i, int heapSize) {
        while (2*i+1<=heapSize){
            int leftChild=(2*i)+1,rightChild=(2*i)+2,target=leftChild;
            if(rightChild <=heapSize && arr[rightChild]> arr[leftChild]){
                target=rightChild;
            }
            if(arr[target] > arr[i]){
                swap(arr[target],arr[i]);
```

```
            }else{
                break;
            }
            i=target;
        }
    }


    void build_heap(vector<int> &arr, int size) {
        for (int i =  size/2; i >= 0; i--) {
            minHeapify(arr, i, size);
        }
    }

    void heap_sort(vector<int> & arr) {
        int len=arr.size()-1;
        build_heap(arr, len);
        for (int i = len ; i >=1; i--) {
            swap(arr[i], arr[0]);
            len--;
            minHeapify(arr, 0, len);
        }
    }


};
```

链表排序 力扣148

插入排序

```
class Solution {
public:
    ListNode *insertionSortList(ListNode *head) {

        if (head == nullptr) {
            return head;
        }
        ListNode *dummyHead = new ListNode(-1);
        dummyHead->next = head;
        ListNode *lastSorted = head;
        ListNode *cur = head->next;
        while (cur != nullptr) {

            if (lastSorted->val <= cur->val) {
                lastSorted = lastSorted->next;
            } else {
                ListNode *prev = dummyHead;
                while (prev->next->val <= cur->val) {
                    prev = prev->next;
                }
                lastSorted->next = cur->next;
```

```
                // 将cur插入到 prev 后
                cur->next = prev->next;
                prev->next = cur;
            }
            cur = lastSorted->next;
        }
        return dummyHead->next;
    }
};
```

归并排序 递归

```
class Solution {
    public ListNode sortList(ListNode head) {
        if (head == null || head.next == null)
            return head;
        ListNode fast = head.next, slow = head;
        while (fast != null && fast.next != null) {
            slow = slow.next;
            fast = fast.next.next;
        }
        ListNode tmp = slow.next;
        slow.next = null;
        ListNode left = sortList(head);
        ListNode right = sortList(tmp);
        ListNode h = new ListNode(0);
        ListNode res = h;
        while (left != null && right != null) {
            if (left.val < right.val) {
                h.next = left;
                left = left.next;
            } else {
                h.next = right;
                right = right.next;
            }
            h = h.next;
        }
        h.next = left != null ? left : right;
        return res.next;
    }
}
```

```
class Solution {
public:
    ListNode* sortList(ListNode* head) {
        if (head == nullptr || head->next == nullptr) return head;

        ListNode* head1 = head;
        ListNode* head2 = split(head);

        head1 = sortList(head1);        //一条链表分成两段分别递归排序
        head2 = sortList(head2);
```

```
        return merge(head1, head2);     //返回合并后结果
    }

    //双指针找单链表中点模板
    ListNode* split(ListNode* head)
    {
        ListNode *slow = head, *fast = head->next;

        while (fast != nullptr && fast->next != nullptr)
        {
            slow = slow->next;
            fast = fast->next->next;
        }

        ListNode* mid = slow->next;
        slow->next = nullptr;           //断尾

        return mid;
    }

    //合并两个排序链表模板
    ListNode* merge(ListNode* head1, ListNode* head2)
    {
        ListNode *dummy = new ListNode(0), *p = dummy;

        while (head1 != nullptr && head2 != nullptr)
        {
            if (head1->val < head2->val)
            {
                p = p->next = head1;
                head1 = head1->next;
            }
            else
            {
                p = p->next = head2;
                head2 = head2->next;
            }
        }

        if (head1 != nullptr) p->next = head1;
        if (head2 != nullptr) p->next = head2;

        return dummy->next;
    }
};
```

归并排序 迭代

```
class Solution {
public:
    ListNode* sortList(ListNode* head) {
        if (head == nullptr) {
            return head;
```

```
        }
        int length = 0;
        ListNode* node = head;
        while (node != nullptr) {
            length++;
            node = node->next;
        }
        ListNode* dummyHead = new ListNode(0, head);
        for (int subLength = 1; subLength < length; subLength <<= 1) {
            ListNode* prev = dummyHead, *curr = dummyHead->next;
            while (curr != nullptr) {
                ListNode* head1 = curr;
                for (int i = 1; i < subLength && curr->next != nullptr; i++) {
                    curr = curr->next;
                }
                ListNode* head2 = curr->next;
                curr->next = nullptr;
                curr = head2;
                for (int i = 1; i < subLength && curr != nullptr && curr->next !=
nullptr; i++) {
                    curr = curr->next;
                }
                ListNode* next = nullptr;
                if (curr != nullptr) {
                    next = curr->next;
                    curr->next = nullptr;
                }
                ListNode* merged = merge(head1, head2);
                prev->next = merged;
                while (prev->next != nullptr) {
                    prev = prev->next;
                }
                curr = next;
            }
        }
        return dummyHead->next;
    }

    ListNode* merge(ListNode* head1, ListNode* head2) {
        ListNode* dummyHead = new ListNode(0);
        ListNode* temp = dummyHead, *temp1 = head1, *temp2 = head2;
        while (temp1 != nullptr && temp2 != nullptr) {
            if (temp1->val <= temp2->val) {
                temp->next = temp1;
                temp1 = temp1->next;
            } else {
                temp->next = temp2;
                temp2 = temp2->next;
            }
            temp = temp->next;
        }
        if (temp1 != nullptr) {
            temp->next = temp1;
        } else if (temp2 != nullptr) {
            temp->next = temp2;
```

```
        }
        return dummyHead->next;
    }
};
```