排序算法

算法分类

- 十种常见的排序算法
  - 比较类排序
    - 交换排序
      - **冒泡排序**
      - **快速排序**
    - 插入排序
      - **简单插入排序**
      - **希尔排序**
    - 选择排序
      - **简单选择排序**
      - **堆排序**
    - **归并排序**
      - 二路归并排序
      - 多路归并排序
  - 非比较类排序
    - **计数排序**
    - **桶排序**
    - **基数排序**

排序算法稳定性

假定在待排序的记录序列中，存在多个具有相同的关键字的记录，若经过排序，这些记录的相对次序保持不变，即在原序列中，$A1 = A2$，且 $A1$ 在 $A2$ 之前，而在排序后的序列中，$A1$ 仍在 $A2$ 之前，则称这种排序算法是稳定的；否则称为不稳定的。

稳定也可以理解为一切皆在掌握中，元素的位置处在你在控制中；而不稳定算法有时就有点碰运气，随机的成分。当两元素相等时它们的位置在排序后可能仍然相同，但也可能不同，是未可知的。

算法性能

排序方法	时间复杂度（平均）	时间复杂度（最坏）	时间复杂度（最好）	空间复杂度	稳定性
插入排序	$O(n^2)$	$O(n^2)$	$O(n)$	$O(1)$	稳定
希尔排序	$O(n^{1.3})$	$O(n^2)$	$O(n)$	$O(1)$	不稳定
选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	不稳定
堆排序	$O(nlog_2n)$	$O(nlog_2n)$	$O(nlog_2n)$	$O(1)$	不稳定
冒泡排序	$O(n^2)$	$O(n^2)$	$O(n)$	$O(1)$	稳定
快速排序	$O(nlog_2n)$	$O(n^2)$	$O(nlog_2n)$	$O(nlog_2n)$	不稳定
归并排序	$O(nlog_2n)$	$O(nlog_2n)$	$O(nlog_2n)$	$O(n)$	稳定
计数排序	$O(n+k)$	$O(n+k)$	$O(n+k)$	$O(n+k)$	稳定
桶排序	$O(n+k)$	$O(n^2)$	$O(n)$	$O(n+k)$	稳定
基数排序	$O(n*k)$	$O(n*k)$	$O(n*k)$	$O(n+k)$	稳定

冒泡排序（bubble sort）

- 冒泡排序是一种简单的排序算法。它重复地走访过要排序的数列，一次比较两个元素，如果它们的顺序错误就把它们交换过来。
- 代码

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

选择排序（selection sort）

- 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，然后，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。
- 代码：

```
void selection_sort(vector<int> &vec)
{
    int size = vec.size();
    for (int i = 0; i < size - 1; ++i)
    {
        int min_index = i;
        for (int j = i + 1; j < size; ++j)
```

```
        {
            if (vec[min_index] > vec[j])
                min_index = j;
        }
        if (min_index != i)
            swap(vec[i], vec[min_index]);
    }
}
```

- 注意：选择排序在最好和最坏的情况下的时间复杂度都是O(n^2)

插入排序（insert sort）

- 通过构建有序序列，对于未排序数据，在已排序序列中从后向前扫描，找到相应位置并插入
- 代码：

```
void insert_sort(vector<int> &vec)
{
    int size = vec.size();
    for (int i = 1; i < size; ++i)
    {
        int j = i - 1;
        int current = vec[i];
        while (j >= 0 and vec[j] > current)
        {
            vec[j + 1] = vec[j];
            --j;
        }
        vec[j + 1] = current;
    }
}
```

- 插入排序在实现上，通常采用in-place排序（即只需用到O(1)的额外空间的排序），因而在从后向前扫描过程中，需要反复把已排序元素逐步向后挪位，为最新元素提供插入空间。

希尔排序（shell sort）

- 简单插入排序的改进版，时间复杂度小于O(n^2)，最坏情况为O(n^2)。它与插入排序的不同之处在于，它会优先比较距离较远的元素。希尔排序又叫**缩小增量排序**
- 代码：

```
void shell_sort(vector<int> &vec)
{
    int size = vec.size();
    for (int gap = size / 2; gap > 0; gap /= 2)
    {
        for (int i = gap; i < size; ++i)
        {
            int j = i - gap;
            int current = vec[i];
            while (j >= 0 and vec[j] > current)
```

```
            {
                vec[j + gap] = vec[j];
                j -= gap;
            }
            vec[j + gap] = current;
        }
    }
}
```

- 希尔排序的核心在于**间隔序列**的设定。既可以提前设定好间隔序列，也可以动态的定义间隔序列。

归并排序（merge sort）

- 归并排序是建立在归并操作上的一种有效的排序算法。该算法是采用分治法（Divide and Conquer）的一个非常典型的应用。将已有序的子序列合并，得到完全有序的序列；即先使每个子序列有序，再使子序列段间有序。若将两个有序表合并成一个有序表，称为2-路归并。

- 代码：

```
void merge(vector<int> &vec, vector<int> &temp, int start, int end)
{
    if (start >= end)
        return;
    int len = end - start;
    int mid = (len >> 1) + start;
    int start1 = start, end1 = mid;
    int start2 = mid + 1, end2 = end;
    merge(vec, temp, start1, end1);
    merge(vec, temp, start2, end2);
    int k = start;
    while (start1 <= end1 and start2 <= end2)
        temp[k++] = vec[start1] < vec[start2] ? vec[start1++] :
vec[start2++];
    while (start1 <= end1)
        temp[k++] = vec[start1++];
    while (start2 <= end2)
        temp[k++] = vec[start2++];
    for (k = start; k <= end; ++k)
        vec[k] = temp[k];
}

void merge_sort(vector<int> &vec)
{
    int size = vec.size();
    vector<int> temp(size);
    merge(vec, temp, 0, size - 1);
}
```

- 归并排序是稳定排序，它也是一种十分高效的排序，能利用完全二叉树特性的排序一般性能；每次合并操作的平均时间复杂度为O(n)，而完全二叉树的深度为|log2n|。总的平均时间复杂度为O(nlogn)。**归并排序的最好，最坏，平均时间复杂度均为O(nlogn)。**
- 归并排序的优化
    - **插入排序**：当待排序列长度为5~20之间，此时使用插入排序能避免一些有害的退化情形

    - 小数据量可使用插入排序加快速度

```
if (end - start <= 10)
{
    insertSort(arr, start, end);
    return;
}
```

  - **插入排序+测试待排序序列中左右半边是否已有序**

```
int len = end - start;
int mid = start + (len >> 1);
int start1 = start, end1 = mid;
int start2 = mid + 1, end2 = end;
merge(arr, temp, start1, end1);
merge(arr, temp, start2, end2);
if (arr[mid] <= arr[mid + 1])
    return;
```

- 链表的归并排序

  注意查找中间节点的判断条件`fast->next != nullptr and fast->next->next != nullptr`

```
ListNode *merge(ListNode *head)
{
    if (head == nullptr or head->next == nullptr)
        return head;
    ListNode *slow = head, *fast = head;
    while (fast->next != nullptr and fast->next->next != nullptr)
    {
        fast = fast->next->next;
        slow = slow->next;
    }
    ListNode *head2 = slow->next;
    slow->next = nullptr;
    ListNode *newhead1 = merge(head);
    ListNode *newhead2 = merge(head2);
    ListNode dummyhead;
    ListNode *curr = &dummyhead;
    while (newhead1 != nullptr and newhead2 != nullptr)
    {
        if (newhead1->val < newhead2->val)
        {
            curr->next = newhead1;
            newhead1 = newhead1->next;
        }
        else
        {
            curr->next = newhead2;
            newhead2 = newhead2->next;
        }
        curr = curr->next;
    }
    if (newhead1 != nullptr)
        curr->next = newhead1;
```

```
    if (newhead2 != nullptr)
        curr->next = newhead2;
    return dummyhead.next;
}
```

快速排序（quick sort）

- 快速排序使用分治法来把一个串（list）分为两个子串（sub-lists）。具体算法描述如下：
  - 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。
- 代码：使用最后一个元素作为pviot

```
void quick_rec(vector<int> &vec, int start, int end)
{
    if (start >= end)
        return;
    int pviot = vec[end];
    int left = start, right = end - 1;
    while (left < right)
    {
        while (vec[left] < pviot)
            ++left;
        while (vec[right] >= pviot)
            --right;
        if (left < right)
            swap(vec[left], vec[right]);
    }
    if (vec[left] > pviot)
        swap(vec[left], vec[end]);
    quick_rec(vec, start, left - 1);
    quick_rec(vec, left + 1, end);
}

void quick_sort(vector<int> &vec)
{
    int size = vec.size();
    quick_rec(vec, 0, size - 1);
}
```

- 分析：
  - 在最优的情况下，快速排序算法的时间复杂度为O(nlogn)。
  - 在最坏的情况下，最终其时间复杂度为$O(n^2)$
  - 快速排序的平均时间复杂度为O(nlogn)
- 快速排序优化
  - 随机选取pviot

```
/*随机选择枢轴的位置，区间在low和high之间*/
int SelectPivotRandom(int arr[],int low,int high)
{
    srand((unsigned)time(NULL));//产生枢轴的位置
    int pivotPos = rand()%(high - low) + low;

    swap(arr[pivotPos],arr[low]);//把枢轴位置的元素和low位置元素互换，此时可以和普通的快排一样调用划分函数
    return arr[low];
}
```

- **三数取中**

```
int getPivot(vector<int> &arr, int start, int end)
{
    int mid = start + (end - start) / 2;
    if (arr[mid] < arr[start])
        swap(arr[mid], arr[start]);
    if (arr[end] < arr[start])
        swap(arr[end], arr[start]);
    if (arr[mid] < arr[end])
        swap(arr[end], arr[mid]);
    return arr[end];
}
```

- **三数取中 + 插入排序**：当待排序列长度为5~20之间，此时使用插入排序能避免一些有害的退化情形
  - 快排需要的时间代价更多，因为递归需要大量指令以及内存空间来执行堆栈运行现场，函数调用等操作。对于极小规模数据而言，确实要比直接插入排序的代价更高

```
if (end - start < 10)
{
    insertSort(arr, start, end);
    return;
}//else时，正常执行快排
```

- **三数取中+插入排序+聚集相等元素**：在一次分割结束后，可以把与pviot相等的元素聚在一起，继续下次分割时，不用再对与pviot相等元素分割
  - 在划分过程中，把与pviot相等元素放入数组的两端
  - 划分结束后，把与pviot相等的元素移到枢轴周围

```
int getPivot(vector<int> &arr, int start, int end)
{
    int mid = start + (end - start) / 2;
    if (arr[mid] < arr[start])
        swap(arr[mid], arr[start]);
    if (arr[end] < arr[start])
        swap(arr[end], arr[start]);
    if (arr[mid] < arr[end])
        swap(arr[end], arr[mid]);
    return arr[end];
}
```

```
void insertSort(vector<int> &arr, int start, int end)
{
    for (int i = start + 1; i <= end; ++i)
    {
        int j = i - 1;
        int curr = arr[i];
        while (j >= start and arr[j] > curr)
        {
            arr[j + 1] = arr[j];
            --j;
        }
        arr[j + 1] = curr;
    }
}

void gather(vector<int> &arr, int start, int end, int boundkey, int
&left, int &right)
{
    if (start >= end)
        return;
    int count = boundkey - 1;
    for (int i = count; i >= start; --i)
    {
        if (arr[i] == arr[boundkey])
        {
            swap(arr[i], arr[count]);
            --count;
        }
    }
    left = count;
    count = boundkey + 1;
    for (int i = count; i <= end; ++i)
    {
        if (arr[i] == arr[boundkey])
        {
            swap(arr[i], arr[count]);
            ++count;
        }
    }
    right = count;
}

void quictSort(vector<int> &arr, int start, int end)
{
    if (end - start < 10)
    {
        insertSort(arr, start, end);
        return;
    }
    int pviot = getPivot(arr, start, end);
    int left = start, right = end - 1;
    while (left < right)
    {
        while (arr[left] < pviot)
            ++left;
        while (arr[right] >= pviot)
            --right;
        if (left < right)
```

```
            swap(arr[left], arr[right]);
    }
    if (arr[left] > pviot)
        swap(arr[left], arr[end]);
    gather(arr, start, end, left, left, right);
    quictSort(arr, start, left);
    quictSort(arr, left, end);
}
```

- 链表的快速排序
  - 直接交换节点值：left和right初始条件

```
void quickSort(ListNode *head, ListNode *tail)
{
    if (head == tail or head->next == tail)
        return;
    int pivot = head->val;
    ListNode *left = head, *right = head->next;
    while (right != tail)
    {
        if (right->val < pivot)
        {
            left = left->next;
            swap(left->val, right->val);
        }
        right = right->next;
    }
    swap(left->val, head->val);
    ListNode *mid = left;
    quickSort(head, mid);
    quickSort(mid->next, tail);
}
```

  - 只能交换节点：分成两个链表，分别保存小于和大于pivot值的节点，然后对两个链表分别排序，最后再组合

```
ListNode *quickSort(ListNode *head)
{
    if (head == nullptr or head->next == nullptr)
        return head;
    int pivot = head->val;
    ListNode left, right;
    ListNode *l = &left, *r = &right, *curr = head->next;
    while (curr != nullptr)
    {
        if (curr->val < pivot)
        {
            l->next = curr;
            l = l->next;
        }
        else
        {
            r->next = curr;
            r = r->next;
        }
        curr = curr->next;
```

```
        }
        l->next = r->next = nullptr;
        ListNode *left_res = quickSort(left.next);
        ListNode *right_res = quickSort(right.next);
        if (left_res == nullptr)
        {
            head->next = right_res;
            return head;
        }
        curr = left_res;
        while (curr->next != nullptr)
        {
            curr = curr->next;
        }
        curr->next = head;
        curr->next->next = right_res;
        return left_res;
    }
```

堆排序（heap sort)

- 利用堆这种数据结构所设计的一种排序算法
- 堆是一个近似完全二叉树的结构，并同时满足堆积的性质：即子结点的键值或索引总是小于（或者大于）它的父节点
- 代码：

```
void max_heapify(vector<int> &vec, int start, int end)
{
    int parent = start;
    int child = parent * 2 + 1; //左子节点
    while (child <= end)
    {
        if (child + 1 <= end and vec[child] < vec[child + 1]) //选择左右子节点中的最大值
            ++child;
        if (vec[child] < vec[parent])
            return;
        swap(vec[parent], vec[child]);
        parent = child;
        child = parent * 2 + 1;
    }
}

void heap_sort(vector<int> &vec)
{
    int size = vec.size();
    for (int i = size / 2; i >= 0; --i) //从最后一棵子树开始构造最大堆
        max_heapify(vec, i, size - 1);
    for (int i = size - 1; i > 0; --i)
    {
```

```
        swap(vec[0], vec[i]);        // 将最大元素与堆中最后一个元素交换
        max_heapify(vec, 0, i - 1); // 前i-1个元素继续构造最大堆
    }
}
```

- 堆排序整体的时间复杂度是 O(nlogn)

计数排序（counting sort）

- 将输入的数据值转化为键存储在额外开辟的数组空间中，作为一种线性时间复杂度的排序，计数排序要求输入的数据必须是有确定范围的整数
- 步骤：
  - 找出待排序的数组中最大和最小的元素；
  - 统计数组中每个值为i的元素出现的次数，存入数组C的第i项；
  - 对所有的计数累加（从C中的第一个元素开始，每一项和前一项相加）；
  - 反向填充目标数组：将每个元素i放在新数组的第C(i)项，每放一个元素就将C(i)减去1。
- 代码：

```
void counting_sort(vector<int> &vec)
{
    int size = vec.size();
    int max_value = vec[0];
    int min_value = vec[0];
    for (int i = 1; i < size; ++i)
    {
        max_value = max_value < vec[i] ? vec[i] : max_value;
        min_value = min_value > vec[i] ? vec[i] : min_value;
    }
    int len = max_value - min_value + 1;
    vector<int> bucket(len, 0);
    for (int i = 0; i < size; ++i)
        bucket[vec[i] - min_value]++;
    int sorted_index = 0;
    for (int i = 0; i < len; ++i)
    {
        while (bucket[i] > 0)
        {
            vec[sorted_index++] = i + min_value;
            --bucket[i];
        }
    }
}
```

桶排序（bucket sort）

- 在额外空间充足的情况下，尽量增大桶的数量
- 使用的映射函数能够将输入的 N 个数据均匀的分配到 K 个桶中
- 当输入的数据可以均匀的分配到每一个桶中时最快
- 当输入的数据被分配到了同一个桶中时最慢

- 代码:
  - 向量数组版:

```
public static void bucketSort(int[] arr){

    // 计算最大值与最小值
    int max = Integer.MIN_VALUE;
    int min = Integer.MAX_VALUE;
    for(int i = 0; i < arr.length; i++){
        max = Math.max(max, arr[i]);
        min = Math.min(min, arr[i]);
    }

    // 计算桶的数量
    int bucketNum = (max - min) / arr.length + 1;
    ArrayList<ArrayList<Integer>> bucketArr = new ArrayList<>(bucketNum);
    for(int i = 0; i < bucketNum; i++){
        bucketArr.add(new ArrayList<Integer>());
    }

    // 将每个元素放入桶
    for(int i = 0; i < arr.length; i++){
        int num = (arr[i] - min) / (arr.length);
        bucketArr.get(num).add(arr[i]);
    }

    // 对每个桶进行排序
    for(int i = 0; i < bucketArr.size(); i++){
        Collections.sort(bucketArr.get(i));
    }

    // 将桶中的元素赋值到原序列
    int index = 0;
    for(int i = 0; i < bucketArr.size(); i++){
        for(int j = 0; j < bucketArr.get(i).size(); j++){
            arr[index++] = bucketArr.get(i).get(j);
        }
    }
}
```

  - 链表版:

```
#include<iterator>
#include<iostream>
#include<vector>
using namespace std;
const int BUCKET_NUM = 10;
struct ListNode{
    explicit ListNode(int i=0):mData(i),mNext(NULL){}
    ListNode* mNext;
    int mData;
};
ListNode* insert(ListNode* head,int val){
    ListNode dummyNode;
    ListNode *newNode = new ListNode(val);
    ListNode *pre,*curr;
```

```
    dummyNode.mNext = head;
    pre = &dummyNode;
    curr = head;
    while(NULL!=curr && curr->mData<=val){
        pre = curr;
        curr = curr->mNext;
    }
    newNode->mNext = curr;
    pre->mNext = newNode;
    return dummyNode.mNext;
}
ListNode* Merge(ListNode *head1,ListNode *head2){
    ListNode dummyNode;
    ListNode *dummy = &dummyNode;
    while(NULL!=head1 && NULL!=head2){
        if(head1->mData <= head2->mData){
            dummy->mNext = head1;
            head1 = head1->mNext;
        }else{
            dummy->mNext = head2;
            head2 = head2->mNext;
        }
        dummy = dummy->mNext;
    }
    if(NULL!=head1) dummy->mNext = head1;
    if(NULL!=head2) dummy->mNext = head2;

    return dummyNode.mNext;
}
void BucketSort(int n,int arr[]){
    vector<ListNode*> buckets(BUCKET_NUM,(ListNode*)(0));
    for(int i=0;i<n;++i){
        int index = arr[i]/BUCKET_NUM;
        ListNode *head = buckets.at(index);
        buckets.at(index) = insert(head,arr[i]);
    }
    ListNode *head = buckets.at(0);
    for(int i=1;i<BUCKET_NUM;++i){
        head = Merge(head,buckets.at(i));
    }
    for(int i=0;i<n;++i){
        arr[i] = head->mData;
        head = head->mNext;
    }
}
```

基数排序（radix sort)

- 基数排序是按照低位先排序，然后收集；再按照高位排序，然后再收集；依次类推，直到最高位。有时候有些属性是有优先级顺序的，先按低优先级排序，再按高优先级排序。最后的次序就是高优先级高的在前，高优先级相同的低优先级高的在前。
- 代码：略