6.十大排序

算法

十大排序

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	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	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

排序基本问题

- Conception：
  a. 就地排序
  b. 内部/外部排序
  c. 稳定排序
  d. 任何排序算法都可以通过指定方式变的稳定
- Question：
  a. 为什么需要稳定排序?
    - 让第一个关键字的排序结果服务于第二个关键字排序中数值相同的那些数
    - 主要是为了第一次考试分数相同时候，可以按照第二次分数的高低进行排序

冒泡排序

- 交换相邻的元素，每次把最大的放在最后

```
vector<int> bubbleSort(vector<int> arr){
    for(int i = 0; i < arr.size(); i++){
        bool flag = false;
        for(int j = 0; j < n - i - 1; j++){
            if(arr[j] > arr[j+1]){
                swap(arr[j], arr[j + 1]);
                flag = true;
```

```
            }
        }
        if(flag == false) break;
    }
    return arr;
}
```

- 稳定排序、原地排序
- 最差O(n)、最好O(n^2)
- 拓展
  a. 优化：代码中的flag，如果数组整个已经有序，数组有序就退出
  b. 再优化：后半部分可能已经有序，每次可以更新j循环的右边界
  c. 使用递归实现冒泡排序
  d. 使用两个栈实现冒泡排序
  e. 对链表进行冒泡排序（交换值or交换节点）
- 景禹 参考

选择排序

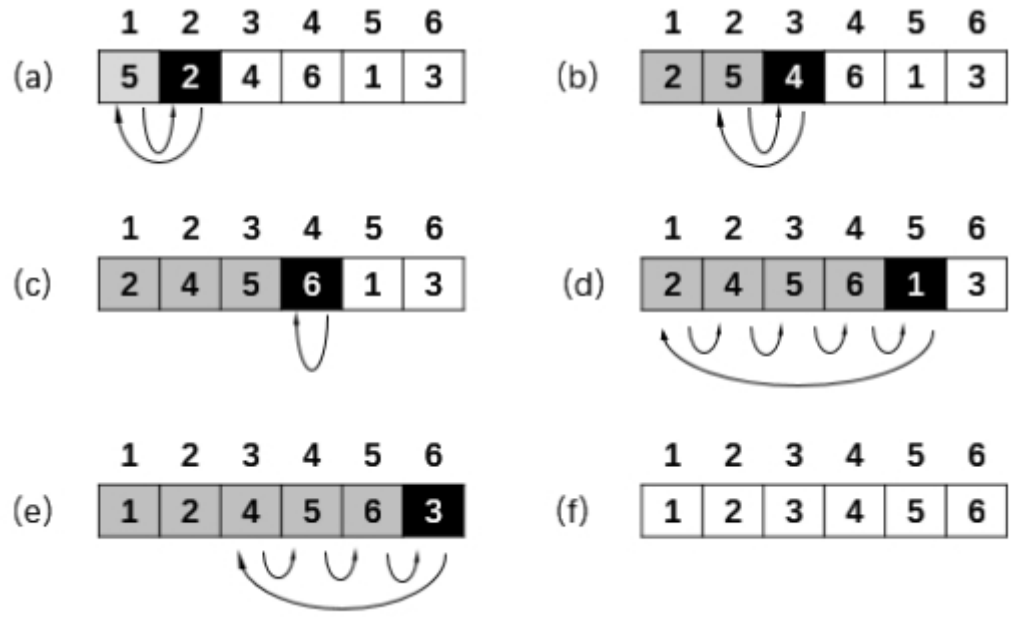
- 选一个最小的和前面的交换

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

- 不稳定排序（模拟441）、原地排序
- 拓展如何变的稳定：每次交换之后，其他元素向后移
- 景禹 参考

插入排序

- 每次新加一个元素，放到截至到自己的位置的区间中合适的位置（该元素前面的已经排好序，所以i从1开始）




```
vector<int> insertionSort(vector<int> arr){
    if(arr.size() == 0) return arr;
    for(int i = 1; i < arr.size(); i++){
        int cur = arr[i];
        int pre = i - 1;
        while(pre >= 0 && arr[pre] > cur){
            arr[pre + 1] = arr[pre];
            pre--;
        }//大于cur的元素向后移
        arr[pre+1] = cur;//把选中的元素放到该放的位置
    }
    return arr;
}
```

- 稳定排序、原地排序

希尔排序

- 分组进行插入排序

```
vector<int> sheelSort(vector<int> arr){
    if(arr.size() == 0) return arr;
    int gap = arr.size() / 2;
    while(gap > 0){
        for(int i = gap; i < arr.size(); i++){
            int cru = arr[i];
            int pre = i - gap;
            while(pre >= 0 && arr[pre] > cru){
                arr[pre + gap] = arr[pre];
                pre -= gap;
            }
            arr[pre + gap] = cru;
        }
        gap /= 2;
    }
    return arr;
}
```

- 不稳定排序（eg：44231）、原地排序

归并排序

- 分治

```
void mergeSort(vector<int> arr,  int l, int r){
    if(l >= r) return;

    int mid = l + r >> 1;
    mergeSort(arr, l, mid);
    mergeSort(arr, mid + 1, r);

    int i = l, j = mid + 1, k = 0;
    vector<int> tmp(r - l + 1);
    while(i <= mid && j <= r){
        if(arr[i] <= arr[j]) tmp[k++] = arr[i++];
        else tmp[k++] = arr[j++];
    }
    while (i <= mid) tmp[k ++ ] = arr[i ++ ];
    while (j <= r) tmp[k ++ ] = arr[j ++ ];

    //再装回原数组中
    for (i = l, k = 0; i <= r; i ++, j ++ ) q[i] = tmp[k];
}
```

- 稳定排序、非原地排序

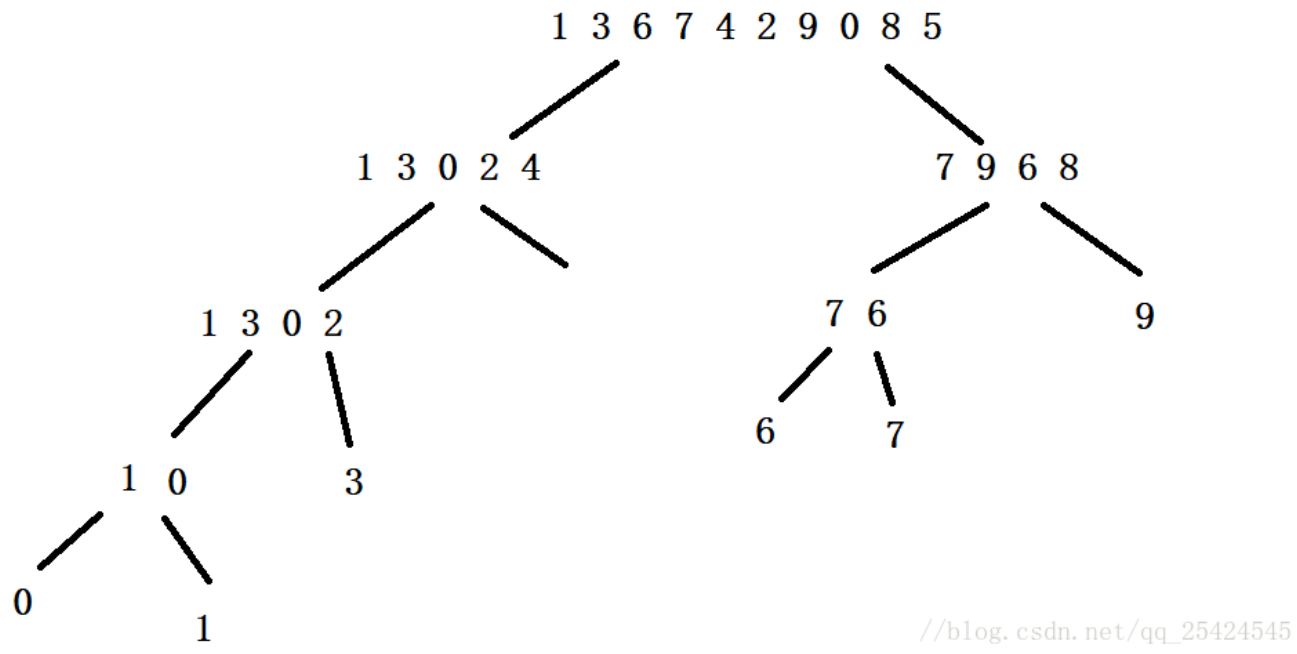
快速排序

- 选>=pivot和<=pivot的元素进行交换

```
void quickSort(vector<int> arr, int l, int r)
{
    if (l >= r) return;

    int i = l - 1, j = r + 1;
    int x = arr[(l + r ) / 2];
    int x = arr[rand() % (r - l) + l];
    int x = arr[l];
    while (i < j)
    {
        do i ++ ; while (arr[i] < x);
        do j -- ; while (arr[j] > x);
        if (i < j) swap(arr[i], arr[j]);
    }
    quick_sort(arr, l, j), quick_sort(arr, j + 1, r);
}
```

- 不稳定排序、原地排序
- 时间复杂度为什么是O(nlogn)



    - 每一层的遍历次数近似为N，树的高度近似为logn
- 如何防止时间复杂度退化?
    - 只有序列已经有序，比如3 2 1，但是利用快排把有序序列反过来，比如1 2 3，这样才会出现O(n^2)
    - 序列基本有序，你选中间元素可以避免时间复杂度的退化。但是大部分排序序列都是无序的，你选择的中间元素最终位置可能是序列最前端或者最后端，这样反而是O(n^2)的。所以，你每次选择中间元素应该不能防止时间复杂度的退化。
- 如何解决序列中重复数字较多的问题?

堆排序

- 数组->堆->顺序排列

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

int n, m;

void down(vector<int>& arr, int i) {
    int maxIndex = i;  // t记录最小值
    if (2 * i <= n && arr[2 * i] < arr[maxIndex]) maxIndex = 2 * i; // 左儿子存在
    if (2 * i + 1 <= n && arr[2 * i + 1] < arr[maxIndex]) maxIndex = 2 * i + 1;
    if (maxIndex != i) {
        swap(arr[maxIndex], arr[i]);
        down(arr, maxIndex);
    }
    return;
}

void heapSort(vector<int> arr){
    for (int i = n / 2; i > 0; i --) down(arr, i);

    while(n) {
        cout << arr[1] << " ";
        arr[1] = arr[n];
        down(arr, 1);
        n--;

```

```
    }
}

int main() {
    cin >> n ;
    vector<int> arr(n + 1);
    for (int i = 1; i <= n; i ++) cin >> arr[i];
    // 初始化堆
    heapSort(arr);
    return 0;
}
```

- 不稳定排序、原地排序

计数排序

- 统计每个数字出现的个数

```
vector<int> countingSort(vector<int> arr){
    if(arr.size() == 0) return arr;
    //bias偏移值
    int bias, min = arr[0], max = arr[0];
    for(int i = 1; i < arr.size(); i++){
        if(arr[i] > max) max = arr[i];
        if(arr[i] < min) min = arr[i];
    }
    bias = 0 - min;
    vector<int> bucket(max - min + 1, 0);
    for(int i = 0; i < arr.size(); i++){
        bucket[arr[i] + bias]++;
    }
    int k = 0;
    for(int i = 0; i < bucket.size(); i++){
        while(bucket[i] > 0){
            arr[k++] = i + min;
            bucket[i]--;
        }
    }
    return arr;
}
```

- 稳定排序 非原地排序
- O(n + k) O(n)
- 不适合跨度太大的数组

桶排序

- 分成x个桶，x个桶内排序之后再组合

```
vector<int> bucektSort(vector<int> arr){
    if(arr.size() == 0) return arr;
    int min = arr[0];
    int max = arr[0];

    for(int i = 1; i < arr.size(); i++){
```

```
        if(arr[i] > max) max = arr[i];
        if(arr[i] < min) min = arr[i];
    }
    //创建d/5+1个桶，每个桶存放5*i ~ 5*i + 5-1范围的数
    int bucketNum = (max - min) / 5 + 1;
    //初始化桶
    vector<vector<int>> bucket(bucketNum, vector<int>(0,0));
    //放入桶中
    for(int i = 0; i < arr.size(); i++){
        bucket[(arr[i] - min) / d].push_back(arr[i] - min);
    }
    //每个桶内排序
    for(int i = 0; i < bucketNum; i++){
        sort(bucket[i].begin(), bucket[i].end());
    }
    //存放结果
    int k = 0;
    for(int i = 0; i < bucketNum; i++){
        for(int j = 0; j < bucket[i].size(); j++){
            arr[k++] = bucket[i][j] + min;
        }
    }
    return arr;
}
```

- 稳定排序、非原地排序

基数排序

- 适合元素大小跨度比较大的
- 找出最大值；选出最高位；从个位开始往上排

```
vector<int> radixSort(vector<int> arr){
    if(arr.size() == 0) return arr;
    int max = arr[0];
    for(int i = 1; i < arr.size(); i++){
        if(arr[i] > max) max = arr[i];
    }

    int maxDigit = 0;
    while(max != 0){
        max /= 10;
        maxDigit++;
    }

    int mod = 10, div = 1;

    vector<vector<int>> bucket(10, vector<int>(0,0));
    for(int i = 1; i <= maxDigit; i++){
        int mod = pow(10, i);
        int div = pow(10, i - 1);
        for(int j = 0; j < arr.size(); j++){
            int num = (arr[j] % pow) / 10;
            bucket[num].push_back(arr[j]);
        }
        int k = 0;
        for(int j = 0; j < 10; j++){
```

```
26              for(int k = 0; k < bucket[j].size(); k++){
27                  arr[k++] = bucket[i][j];
28              }
29              bucket[j].clear();
30          }
31      }
32      return arr;
33  }
```

- 稳定排序、非原地排序
- O(n * k) O(n)

问题

1. 有比快速排序、堆排序时间复杂度更好的排序算法吗?
    - 快速排序最坏O(n^2)，堆排序都是O(nlogn)
    - 插入排序和冒泡排序在数据有序时时间复杂度为O(n)
    - 计数排序和桶排序最好下可以达到O(n+k)