Search Algorithm

- 各种排序算法指标对比

排序方法	平均情况	最好情况	最坏情况	辅助空间	稳定性
冒泡排序	O(n2)	O(n)	O(n2)	O(1)	稳定
简单选择排序	O(n2)	O(n2)	O(n2)	O(1)	稳定
直接插入排序	O(n2)	O(n)	O(n2)	O(1)	稳定
希尔排序	O(nlogn)~O(n2)	O(n1.3)	O(n2)	O(1)	不稳定
堆排序	**O(nlogn)**	O(nlogn)	O(nlogn)	**O(1)**	不稳定
归并排序	**O(nlogn)**	O(nlogn)	O(nlogn)	O(n)	**稳定**
快速排序	**O(nlogn)**	O(nlogn)	O(n2)	O(logn)~O(n)	不稳定

- 稳定：如果a原本在b前面，而a=b，排序之后a仍然在b的前面
- 不稳定：如果a原本在b的前面，而a=b，排序之后a可能会在b的后面
- **归并排序与快排**：
    - 1. 都是分治思想，但分解和合并的策略不一样
    - 2. 归并是从中间分两个，合并后两个数列再次排序
    - 3. 快排是比较后，小的在左大的在右，直接合并不再需要排序
    - 所以**快排比归并排序更高效**一些

---

index

- 分治法应用
    - 1. [代码框架] mergeSort
    - 2. [代码框架] quickSort

mergeSort

- 排序思想：
    1. 将待排序序列从中间一分为二，对左右两边再进行递归分割操作，得到n个相互独立的子序列；
    2. 对n个独立的子序列递归的执行合并操作，最终得到有序的序列。
- python版本

```
def merge(left, right):
    l = 0
    r = 0
    m = []
    while l < len(left) and r < len(right):
        if left[l] <= right[r]:
            m.append(left[l])
            l += 1
        else:
            m.append(right[r])
```

```
            r += 1
    remains = right[r:] if l == len(left) else left[l:]
    m.extend(remains)
    return m

def merge_sort(arr):
    if len(arr) <= 1:
        return arr
    middle = len(arr) // 2
    left = merge_sort(arr[:middle])
    right = merge_sort(arr[middle:])
    return merge(left, right)
```

- cpp版本

```
void merge(int *ms,int startIndex,int endIndex)
{
    //进入归并步骤时，数组将由两个数组合并，升序排序划分，左边的称之为左数组，同理，右边的称之为右数组。

    // 待定左数组的右边界
    int left_mid = (startIndex+endIndex)/2;
    //待定右数组的左边界
    int mid_right = ((startIndex+endIndex)/2)+1;
    //待定左数组长度
    int left_length = (left_mid-startIndex)+1;
    //待定右数组长度
    int right_length = (endIndex-mid_right)+1;
    //初始化左数组
    int left_array[left_length];
    //初始化右数组
    int right_array[right_length];

    for(int i=left_mid;i>=startIndex;i--)
    {
        //将左数组挂起
        left_array[i-startIndex] = ms[i];
    }

    for(int i=endIndex;i>=mid_right;i--)
    {
        //将右数组挂起
        right_array[i-mid_right] = ms[i];
    }
    //将两个指针指向左、右数组的头元素
    int l_index=0,r_index=0;

    //双数组循环排序，复杂度O(n)，排序后的结果直接赋回原数组ms上
    for(int i=startIndex;i<=endIndex;i++)
    {
        if(l_index!=left_length && r_index!=right_length)
        {
```

```
            if(left_array[l_index]<right_array[r_index])
            {
                ms[i] = left_array[l_index++];
            }
            else
            {
                ms[i] = right_array[r_index++];
            }
        }
        else if(l_index==left_length)
        {
            ms[i] = right_array[r_index++];
        }
        else
        {
            ms[i] = left_array[l_index++];
        }
    }
}
/*
* 归并排序方法有三个参数，第一个是初始的数组，
* 第二个是该数组的起始索引，第三是该数组的尾巴索引
* note ms[] = {2,3,5,1,0,8,6,9,7}
*/
void mergeSort(int *ms,int startIndex,int endIndex)
{
    //如果数列划分至最小单位(一个数)则停止分割
    if(endIndex-startIndex>0)
    {
        //将数列分为左右部分进行分治
        //左分治
        mergeSort(ms,startIndex,(startIndex+endIndex)/2);
        //右分治
        mergeSort(ms,((startIndex+endIndex)/2)+1,endIndex);
        //归并
        merge(ms,startIndex,endIndex);
    }
}
```

---

quickSort

- 快速排序是一种基于**分治技术**的重要排序算法
- 快排是一种不稳定排序，比如基准值的前后都存在与基准值相同的元素，那么相同值就会被放在一边，这样就打乱了之前的相对顺序
- 时间复杂度：快排的时间复杂度为O(nlogn)
- 空间复杂度：排序时需要另外申请空间，并且随着数列规模增大而增大，其复杂度为：O(nlogn)
- 快速排序有一个缺点就是对于小规模的数据集性能不是很好
- 快排由于是原地交换所以没有合并过程 传入的索引是存在的索引（如：0、length-1 等），越界可能导致崩溃
- python版本

```python
def quick_sort(b):
    """快速排序"""
    if len(b) < 2:
        return arr
    # 选取基准，随便选哪个都可以，选中间的便于理解
    pivot = arr[len(b) // 2]
    # 定义基准值左右两个数列
    left, right = [], []
    # 从原始数组中移除基准值
    b.remove(pivot)
    for item in b:
        # 大于基准值放右边
        if item >= pivot:
            right.append(item)
        else:
            # 小于基准值放左边
            left.append(item)
    # 使用迭代进行比较
    return quick_sort(left) + [pivot] + quick_sort(right)
```

- cpp版本

```cpp
void Swap(int &p,int &q)
{
    int temp;
    temp = p;
    p = q;
    q = temp;
}

int Partition(int InputArray[],int nLow,int nHigh)
{
    int i = nLow,j = nHigh+1;
    int x=InputArray[i];
    while (true)
    {
        //将 < x的元素交换到中轴左边区域
        while (InputArray[++i]<x);
        //将 >x的元素交换到中轴右边区域
        while (InputArray[--j]>x);
        if (i>=j)break;
        Swap(InputArray[i],InputArray[j]);
    }
    //将x交换到它在排序序列中应在的位置上
    InputArray[nLow]=InputArray[j];
    InputArray[j]=x;
    return j;
}

void QuickSort(int InputArray[],int nLow,int nHigh)
```

```
{
    if (nLow < nHigh)
    {
        int index = Partition(InputArray,nLow,nHigh);
        QuickSort(InputArray,nLow,index-1);
        QuickSort(InputArray,index+1,nHigh);
    }
}
```
