排序

快速排序

```
list = [1, 6, 9, 2, 5, 0, 7, 3]
def quickSort(list, low, high):
  if (low < high):
    pivot = huafen(list, low, high) # 取基准
    quickSort(list, low, pivot - 1) #左子区
    quickSort(list, pivot + 1, high) #右子区
def huafen(list, low, high):
  pivot = list[low] #定基准
  # 判断是否有小于pivot的值，没有则往左以此检查
  while(low < high and list[high] > pivot):
    high -= 1
  list[low] = list[high]
  # 判断是否有大于pivot的值，没有则往右以此检查
  while(low < high and list[low] <= pivot):
    low += 1
  list[high] = list[low]
  # 放回pivot (因为最后low=high，所以这里写low或者是high都
相同)
  list[low] = pivot
  # 返回基准索引
  return low
if __name__ == "__main__":
  print('排序前：', list)
  quickSort(list, 0, len(list) - 1)
  print('排序后：', list)

排序前：  [1, 6, 9, 2, 5, 0, 7, 3]
排序后：  [0, 1, 2, 3, 5, 6, 7, 9]
```

```c
#include <stdio.h>
// 划分函数（类似于 Python 的 huafen）
int partition(int arr[], int low, int high) {
    int pivot = arr[low];  // 选择第一个元素作为基准
    while (low < high) {
        // 从右往左找比 pivot 小的元素
        while (low < high && arr[high] > pivot) {
            high--;
        }
        arr[low] = arr[high];  // 把小的值移到左侧
        // 从左往右找比 pivot 大的元素
        while (low < high && arr[low] <= pivot) {
            low++;
        }
        arr[high] = arr[low];  // 把大的值移到右侧
    }

    // 放回 pivot
    arr[low] = pivot;
    return low;  // 返回基准索引
}

// 快速排序递归实现
void quickSort(int arr[], int low, int high) {
    if (low < high) {
        int pivot = partition(arr, low, high);  // 划
分
        quickSort(arr, low, pivot - 1);  // 递归排序左
侧
        quickSort(arr, pivot + 1, high); // 递归排序右
侧
    }
}
// 打印数组
void printArray(int arr[], int size) {
    for (int i = 0; i < size; i++) {
```

```
        printf("%d ", arr[i]);
    }
    printf("\n");
}
// 主函数
int main() {
    int arr[] = {1, 6, 9, 2, 5, 0, 7, 3};
    int size = sizeof(arr) / sizeof(arr[0]);
    printf("排序前: ");
    printArray(arr, size);

    quickSort(arr, 0, size - 1);

    printf("排序后: ");
    printArray(arr, size);
    return 0;
}
```

堆排序

```
// 建立大根堆
void BuildMaxHeap(int A[], int len){
    for(int i = len / 2;i > 0;i--)  // 从后往前调整所有非终端结点
        HeadAdjust(A, i, len);
}

// 将以K为根的字树调整为大根堆
void HeadAdjust(int A[],int k,int len){
    A[0] = A[k];                          // A[0]暂存子树的根节点
    for(int i = 2 * k;i <= len;i*=2){   // 沿key较大的子结点向下筛选
```

```
        if(i < len && A[i] < A[i+1])
            i++;                                // 取key较大的
子结点下标
        if(A[0] >= A[i])    break;              // 筛选结果
        else{
            A[k] = A[i];                        // 将A[i]调整
到双亲结点上
            k = i;                              // 修改k值，以
便继续向下筛选
        }
    }
    A[k] = A[0];                                // 被筛选结点
的值放入最终位置
}

// 堆排序的完整逻辑
void HeapSort(int A[], int len){
    BuildMaxHeap(int A[], int len);
    for (int i=len;i>1;i--){
        swap(A[i], A[1]);
        HeadAdjust(A,1,i-1);
    }
}
```

归并排序

```
int *B = (int *)malloc(n*sizeof(int));  //辅助数组B
// A[low~mid]和A[mid+1~high]各自有序，将两部分进行合并
void Merge(int A[], int low, int mid, int high){
    int i, j, k;
    // 将A中所有元素复制到B中
    for(k=low;k<=high;k++){
        B[k] = A[k];
    }
    for(i=low,j=mid+1,k=i;i<=mid&&j<=high;k++){
```

```
        if(B[i]<=B[j])
            A[k] = B[i++];
        else
            A[k] = B[j++];
    }
    while(i<=mid)   A[k] = B[i++];
    while(j<=high)  A[k] = B[j++];
}

void MergeSort(int A[], int low, int high){
    if(low<high){
        int mid = (low+high) / 2;    // 从中间划分
        MergeSort(A, low, mid);      // 对左半部分归并排
序
        MergeSort(A, mid+1,high);    // 对右半部分归并排
序
        Merge(A,low,mid,high);       // 归并
    }
}

// 思路：调用 MergeSort 函数将数组划分为两部分，依此递归调用
MergeSort 函数，将左半部分排序好后，对右半部分进行排序，最后
将排序好的两部分进行归并排序
```