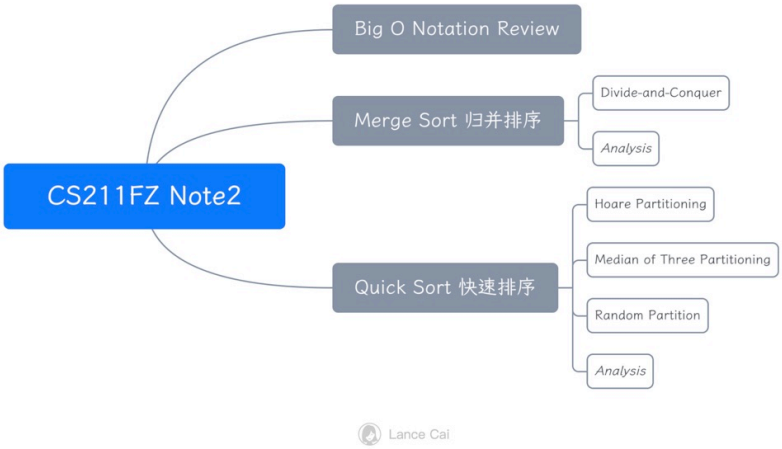
CS211FZ Note 2 | Algorithms & Data Structures | DSA2

Key Points 2 (Sort)

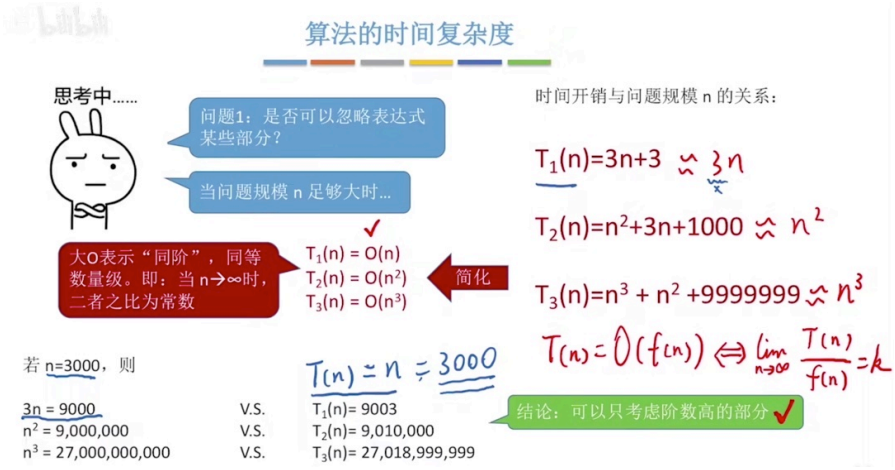
- **Merge Sort Order**
- **Median of Three Partitioning**

CS211FZ Note2

- Big O Notation Review
- Merge Sort 归并排序
  - Divide-and-Conquer
  - *Analysis*
- Quick Sort 快速排序
  - Hoare Partitioning
  - Median of Three Partitioning
  - Random Partition
  - *Analysis*

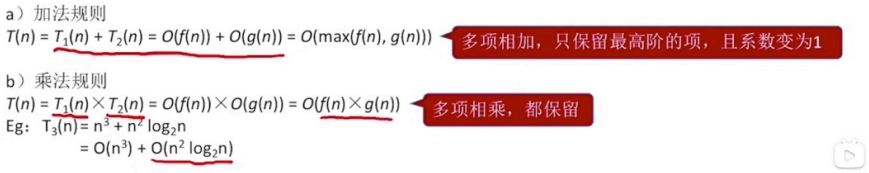


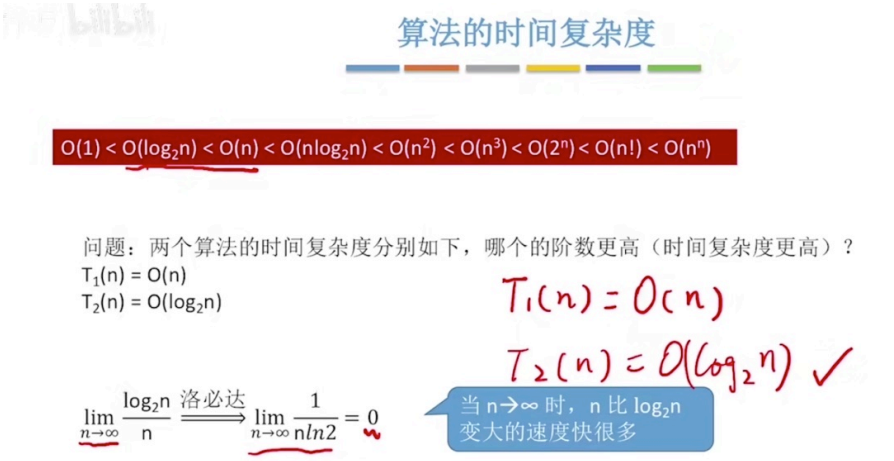
L3.0 Big O Notation Review 时间复杂度分析



时间复杂度规则

- 加法规则
  - 多项相加，只保留最高项
- 乘法规则
  - 多项相乘，都需要保留





常对幂指阶（常数 < 对数 < 幂函数 < 指数 < 阶乘）

```c
//算法4— 搜索数字型爱你
void loveYou(int flag[], int n) {   //n 为问题规模
    printf("I Am Iron Man\n");
    for(int i=0; i<n; i++){ //从第一个元素开始查找
        if(flag[i]==n){ //找到元素n
            printf("I Love You %d\n", n);
            break;  //找到后立即跳出循环
        }
    }
}
```

```c
//flag 数组中乱序存放了 1~n 这些数
int flag[n]={1...n};
loveYou(flag, n);
```

很多算法执行时间与输入的数据有关

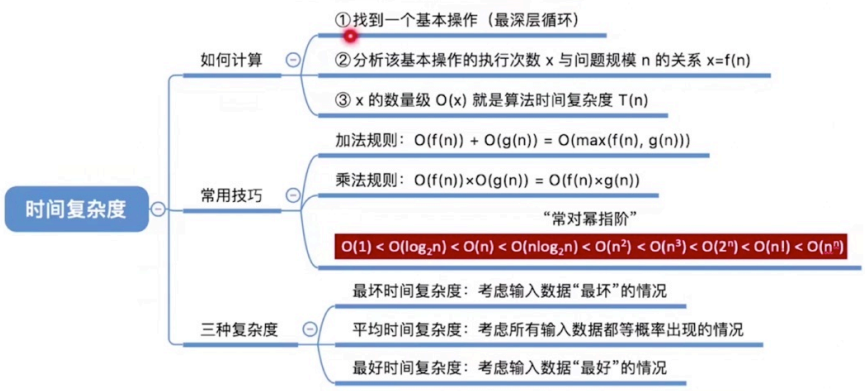
计算上述算法的时间复杂度 T(n)

最好情况：元素n在第一个位置 ——最好时间复杂度 T(n)=O(1)
最坏情况：元素n在最后一个位置 ——最坏时间复杂度 T(n)=O(n)
平均情况：假设元素n在任意一个位置的概率相同为 $\frac{1}{n}$ ——平均时间复杂度 T(n)=O(n)

循环次数 x = (1+2+3+...+n) $\frac{1}{n}$ = $(\frac{n(1+n)}{2})\frac{1}{n}$ = $\frac{1+n}{2}$ T(n)=O(x)=O(n)

1. 最坏时间复杂度
2. 平均时间复杂度
3. 最好时间复杂度

3.0 Overall



Summary of the Running Times

Algorithm	Worst-case	Average-case
Insertion Sort	$\Theta(n^2)$	$\Theta(n^2)$
Merge Sort	$\Theta(n \lg n)$	$\Theta(n \lg n)$
Heapsort	$O(n \lg n)$	--
Quicksort	$\Theta(n^2)$	$\Theta(n \lg n)$
Counting sort	$\Theta(k+n)$	$\Theta(k+n)$
Radix sort	$\Theta(d(k+n))$	$\Theta(d(k+n))$
Bucket sort	$\Theta(n^2)$	$\Theta(n)$

3.1 Merge Sort

Merge Sort

- Merge sort is an efficient comparison-based, divide-and-conquer recursive algorithm.
- Merge sort runs in $\boldsymbol{O(N\lg N)}$ both worst-case and average-case running time and the number of comparisons used by the algorithm is nearly optimal.
- Many implementations of merge sort are stable, i.e., the order of equal elements is the same in the input and output.

3.1.1 分而治之

divide and conquer：

1. **分解**：把一个大问题拆成小问题；
2. **解决**：逐个解决小问题；
3. **合并**：再把这些小问题的结果组合起来组成大问题的解。

1、**分解**待排序的n个元素的序列成n/2个元素的两个子序列；

2、使用归并排序递归得排序这两个子序列；

3、**合并**两个已排序的子序列来产生最终答案。

Divide-and-Conquer

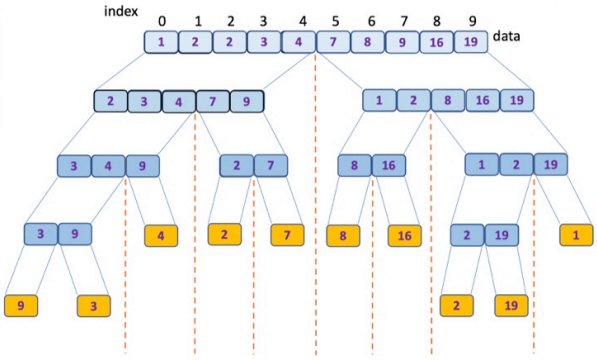
The merge sort algorithm follows the divide-and-conquer paradigm

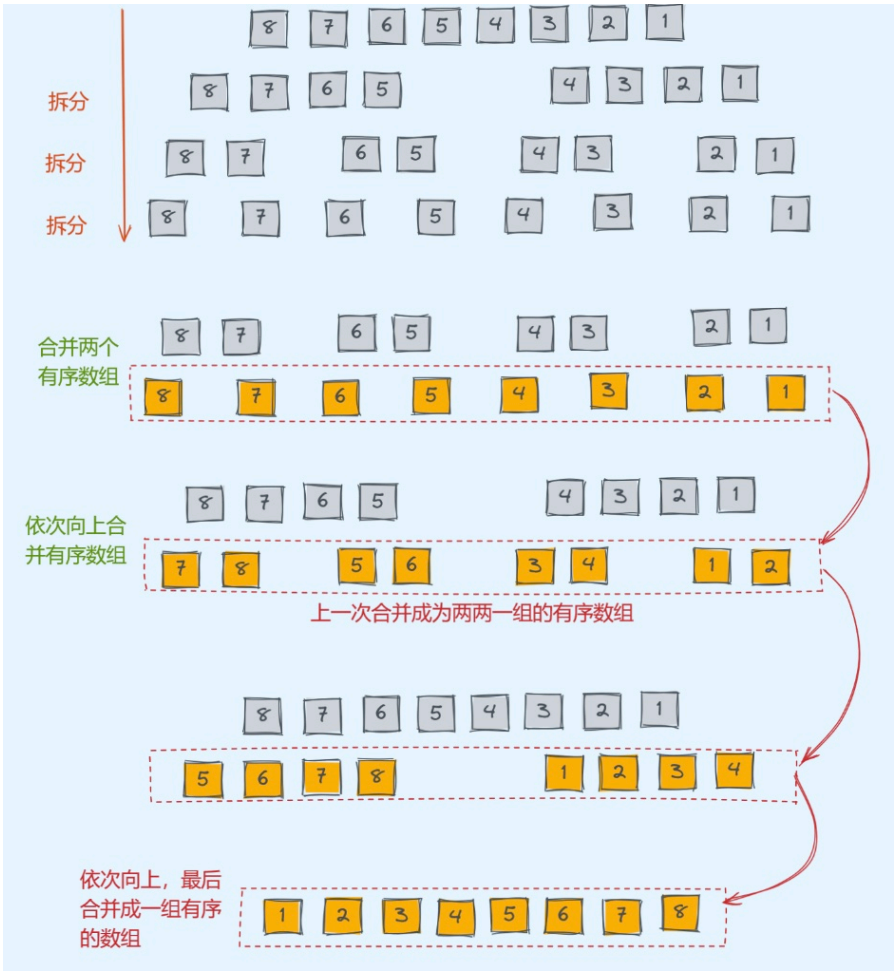
- **Divide** the problem into a number of subproblems that are smaller instances of the same problem
  - ***Merge sort***: divide the $\boldsymbol{N}$-element sequence to be sorted into two sub-sequences of $\frac{N}{2}$ elements each
- **Conquer** the subproblems by solving them recursively.
  - ***Merge sort***: sort the two sub-sequences recursively
- **Combine** the solutions to the subproblems into the solution for the original problem
  - ***Merge sort***: merge the two sorted sub-sequences to produce the sorted answer

Merge Sort Example

```
mergeSort(data, left, right)
  if left < right
    centre = (left + right) / 2
    mergeSort(data, left, centre)
    mergeSort(data, centre+1, right)
    merge(data, left, centre, right)
```

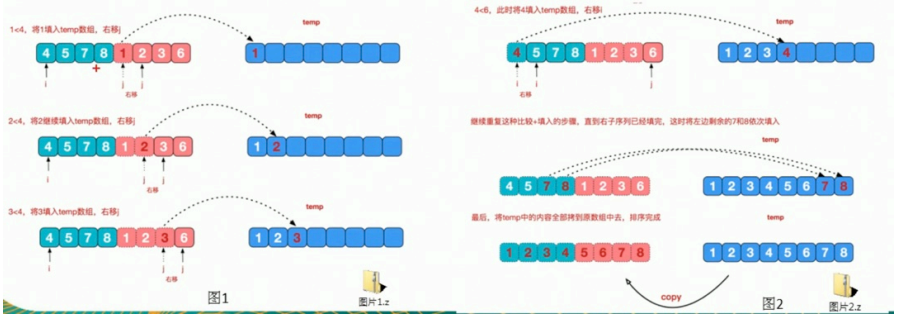
```
merge(data, left, centre, right)
  n1 = centre – left + 1
  n2 = right - centre
  let L[ 0..n1 ] and R[ 0.. n2 ] be new arrays
  for i = 0 to n1
    L[ i ] = data[ left + i ]
  for j = 0 to n2
    R[ j ] = data[ centre + j ]
  L[ n1 ] = R[ n2 ] = ∞
  i = j = 0
  for k = left to right
    if L[ i ] ≤ R[ j ]
      data[ k ] = L[ i++ ]
    else data[ k ] = R[ j++ ]
```





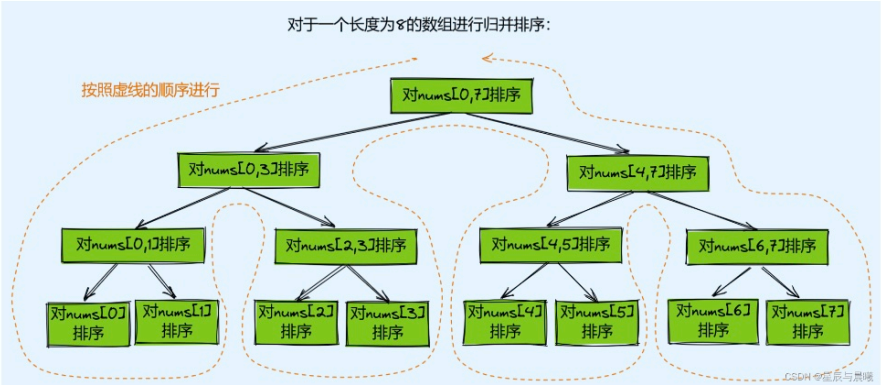
归并排序思想示意图2-合并相邻有序子序列：

再来看看治阶段，我们需要将两个已经有序的子序列合并成一个有序序列，比如上图中的最后一次合并，要将[4,5,7,8]和[1,2,3,6]两个已经有序的子序列，合并为最终序列[1,2,3,4,5,6,7,8]，来看下实现步骤



这也是个典型的空间换时间的一种做法。

在开始时那为了演示，使用的是同时拆分同时合并的，但在归并排序当中，实际使用的是按照深度优先的顺序完成归并排序的。



3.1.2 代码实现：

```java
    /**
     * 递归函数对nums[left...right]进行归并排序
     * @param nums 原数组
     * @param left 左边的索引
     * @param right 右边记录索引位置
     * @param temp
     */
    private static void mergeSort(int[] nums, int left, int right, int[] temp) {
        if (left == right){//当拆分到数组当中只要一个值的时候，结束递归
            return;
        }
        int mid = (left+right)/2;   //找到下次要拆分的中间值
        mergeSort(nums,left,mid,temp);//记录树左边的
        mergeSort(nums,mid+1,right,temp);//记录树右边的

        //合并两个区间
        for (int i = left; i <= right; i++) {
            temp[i] = nums[i];
//temp就是辅助列表，新列表的需要排序的值就是从辅助列表中拿到的
        }
        int i = left;       //给辅助数组里面的值标点
        int j = mid +1;
        for (int k = left; k <= right ; k++) {//k 就为当前要插入的位置
            if (i == mid + 1){
                nums[k] = temp[j];
                j++;
            }else if (j == right+1){
                nums[k] = temp[i];
                i++;
            }
            else if (temp[i] <= temp[j]){
                nums[k] = temp[i];
                i++;
            }else {
                nums[k] = temp[j];
                j++;
            }
        }
    }
}
```

优化后：

1、可以在小区间内使用插入排序。就是在判断当left == right这里使用一个区间的插入排序，**当数组长度较小的时候，使用插入排序。**

2、就是在合并两个有序的数组之前，我们先看看这两个数组拼接起来，是不是**已经成为了有序的数组。**

```java
    private static void mergeSort(int[] nums, int left, int right, int[] temp) {
        if (right - left < 16) {//当拆分到数组长度小于16的时候，直接插入排序来提高代码运行速度
            insertionSort(nums, right, left);
            return;
        }
        int mid = (left + right) / 2;   //找到下次要拆分的中间值
        mergeSort(nums, left, mid, temp);//记录树左边的
        mergeSort(nums, mid + 1, right, temp);//记录树右边的
        if (nums[mid] <= nums[mid + 1]) {
            //因为整数除法是向下取整，所以这里不会代写mid + 1越界
            return;
        }


        //合并两个区间
        for (int i = left; i <= right; i++) {
            temp[i] = nums[i]; //temp就是辅助列表，新列表的需要排序的值就是从辅助列表中拿到的
        }
        int i = left;       //给辅助数组里面的值标点
        int j = mid + 1;
        for (int k = left; k <= right; k++) {//k 就为当前要插入的位置
            if (i == mid + 1) {
                nums[k] = temp[j];
                j++;
            } else if (j == right + 1) {
                nums[k] = temp[i];
                i++;
            } else if (temp[i] <= temp[j]) {
                nums[k] = temp[i];
                i++;
            } else {
                nums[k] = temp[j];
                j++;
            }
        }
    }
```

Analysis -- Using Recursion Tree

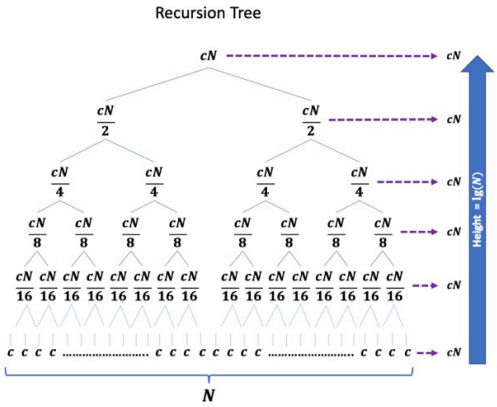
$$T(N) = \begin{cases} c & \text{if } N = 1, \\ 2T\left(\frac{N}{2}\right) + cN & \text{if } N > 1, \end{cases}$$

1. From the recursion tree, given the height of the tree $\lg N$, there are in total $\lg N + 1$ levels

   Thus, $T(N) = cN\lg N + cN$

2. Ignoring the low-order term and the constant c, thus $N\lg N$ is an asymptotically tight bound for $T(N)$, i.e.,

   $T(N) = \theta(N\lg N)$



Merge Sort Implementation using an Auxiliary Array

Merge Sort Implementation using an Auxiliary Array

```
private static <E extends Comparable<E>> void mergeSort(E[] data, E[] auxArray, int left, int right) {
  if (left < right) {
    int centre = (left + right) / 2;
    mergeSort(data, auxArray, left, centre);
    mergeSort(data, auxArray, centre + 1, right);
    merge(data, auxArray, left, centre + 1, right);
  }
}
```

```
private static <E extends Comparable<E>> void merge(E[] data, E[] auxArray, int leftPos, int rightPos, int rightEnd) {
  int leftEnd = rightPos - 1;
  int tempPos = leftPos;
  int numElements = rightEnd - leftPos + 1;

  while (leftPos <= leftEnd && rightPos <= rightEnd) {
    if (data[leftPos].compareTo(data[rightPos]) <= 0)
      auxArray[tempPos++] = data[leftPos++];
    else
      auxArray[tempPos++] = data[rightPos++];
  }
  while (leftPos <= leftEnd) // Copy rest of first half
    auxArray[tempPos++] = data[leftPos++];
  while (rightPos <= rightEnd) // Copy rest of right half
    auxArray[tempPos++] = data[rightPos++];
  for (int i = 0; i < numElements; i++, rightEnd--) { // Copy auxArray back
    data[rightEnd] = auxArray[rightEnd];
}
```

并轨 && 快速

快速排序与归并排序都采用的是分而治之的算法思想来完成排序的。

只不过归并排序我们是不管怎么样，都是将数组一分为二，然后不停地合并两个有序的数据。而快速排序是在划分数组这件事上下足了文章，但是它没有合并的过程。

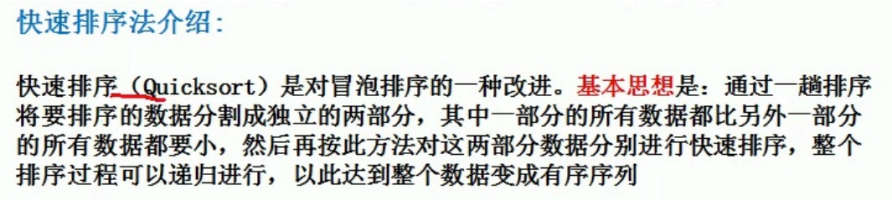
快速排序这种特殊的分而治之的思想也可以称为**减而治之**的算法思想。就是将问题的规模进行减少。减而治之是分而治之的特例。

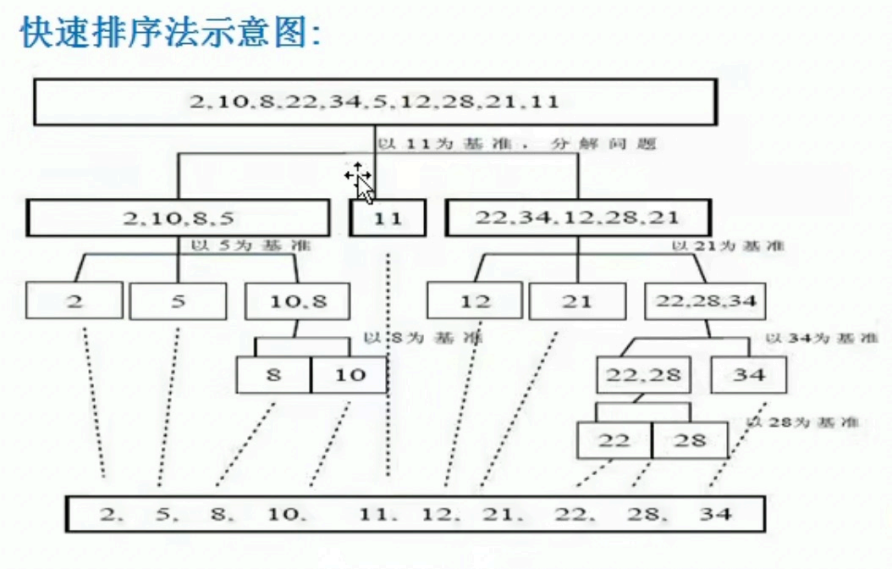
Summary of the Running Times

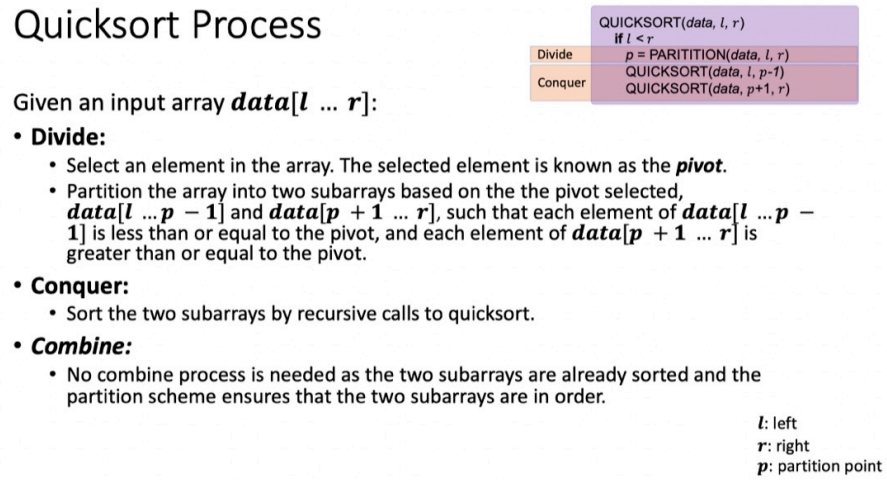
Algorithm	Worst-case	Average-case
Insertion Sort	$\Theta(n^2)$	$\Theta(n^2)$
Merge Sort	$\Theta(n\,lg\,n)$	$\Theta(n\,lg\,n)$
Heapsort	$O(n\,lg\,n)$	--
Quicksort	$\Theta(n^2)$	$\Theta(n\,lg\,n)$
Counting sort	$\Theta(k+n)$	$\Theta(k+n)$
Radix sort	$\Theta(d(k+n))$	$\Theta(d(k+n))$
Bucket sort	$\Theta(n^2)$	$\Theta(n)$

3.2 Quick Sort

基本思想：







3.2.1 快速排序类别：

- Hoare partition
- Median of Three Partitioning 三数取中
- Random Partition 随机分区

1. *Hoare partition*

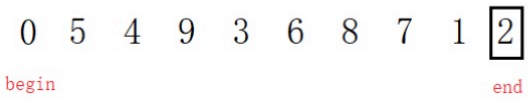
1、取最右边right为基准值pivot

Hoare版本

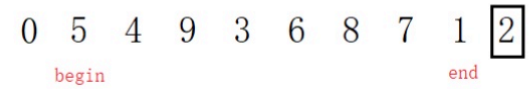
基准值的选择方法：一般情况下我们会选择待排序序列的最后一个元素作为基准值、但是，若待排序序列的最后一个元素是最值或者接近于最值的情况，就会使排序后的左右序列元素个数相差巨大，效率降低，所以我们一般采用三数取中法，找出序列开始结点、序列之间结点、序列最后结点的中间值作为基准值

1:设置两个指针：begin、end分别指向待排序的序列的两端

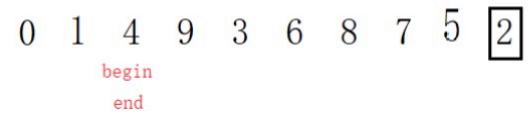
三数取中法获得基准值为2



2：begin从待排序序列的左边开始，找到比基准值大的元素停止。
end从待排序序列的右边开始，找到比基准值小的元素停止。



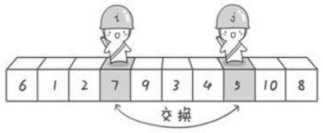
3. 如果begin!=end、则两个下标对应的元素交换，继续向后比较
直至begin==end

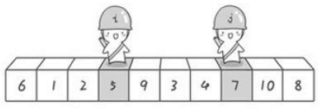


4. begin==end之后将begin指向的值与基准值交换，返回begin
若begin==待排序序列的最后一个元素即end指针还没有移动，说明元素有序，不用交换

2、取最左边left为基准值pivot

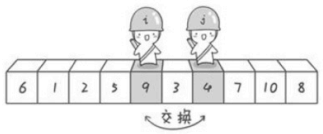
首先哨兵j开始出动。因为此处设置的基准数是最左边的数，所以需要让哨兵j先出动，这一点非常重要（请自己想一想为什么）。哨兵j一步一步地向左挪动（即j--），直到找到一个小于6的数停下来。接下来哨兵i再一步一步向右挪动（即i++），直到找到一个数大于6的数停下来。最后哨兵j停在了数字5面前，哨兵i停在了数字7面前。

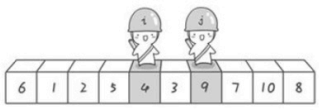




现在交换哨兵i和哨兵j所指向的元素的值。交换之后的序列如下：

6 1 2 5 9 3 4 7 10 8





代码实现：

```java
public class QuickSort {
    public static void quickSort(int[] arr,int low,int high){
        int i,j,temp,t;
        if(low>high){
            return;
        }
        i=low;
        j=high;
        //temp就是基准位
        temp = arr[low];

        while (i<j) {
            //先看右边，依次往左递减
            while (temp<=arr[j]&&i<j) {
                j--;
            }
            //再看左边，依次往右递增
            while (temp>=arr[i]&&i<j) {
                i++;
            }
            //如果满足条件则交换
            if (i<j) {
                t = arr[j];
                arr[j] = arr[i];
                arr[i] = t;
            }

        }
        //最后将基准为与i和j相等位置的数字交换
         arr[low] = arr[i];
         arr[i] = temp;
        //递归调用左半数组
        quickSort(arr, low, j-1);
        //递归调用右半数组
        quickSort(arr, j+1, high);
    }

```

2. *Median of Three Partitioning 三数取中*

三个参数：low，high，和mid分别表示0号下标，最大号下标，最中间的下标，三者的值进行比较，把中间大的数字放在low，最大

三平均划分法快排：以最左元素、最右元素、最中间元素的中位数为枢轴

```java
public static void midThree(int[] array, int low, int high){
    //array[mid] <= array[low] <= array[high]  把中间大的数字放在第一个，最大的放在最后，最小的放在中间
    int mid = (high + low)/2;   //中间位置的下标
    if(array[low] > array[high]){
      swap(array,low,high);
    } //array[low] <= array[high]
    if(array[mid] > array[high]){
        swap(array,mid,high);
    }//array[mid] <= array[high]
    if(array[mid] > array[low]){
        swap(array,low,mid);
    }//array[mid] <= array[low]
}
```

3. *Random Partition 随机分区*

随机选取一个数字，和0号位置交换，再把这个数放入tmp，在后面找比它小的数，放到0号位置，h--，在前面找比它大的数，l++

随机化快排：使用随机元素作为枢轴

```java
public static void swap(int[] array,int low,int high){  //交换low号下标和随机数下标的值
      int tmp = array[low];
      array[low] = array[high];
      array[high] = tmp;
    }
 public static void quick1(int[] array,int low,int high) {
     Random random = new Random();
     int ranNum = random.nextInt(high-low+1)+low;
     swap(array,low,ranNum);
     int par = partion(array,low,high);
     if(par > low+1){  //左边有两个元素：起始low为0，par>1,在2号下标的位置,
         quick(array,low,par-1);
     }
     if(par < high-1){ //右边有两个元素：起始位par+1,
         quick(array,par+1,high);
     }
 }
public static void quickSort1(int[] array){
    quick(array,0,array.length-1);
}
```

3.2.2 Analysis

Quicksort

- The quicksort algorithm is also a divide-and-conquer recursive algorithm, which has a worst-case running time of $\boldsymbol{\theta(N^2)}$ on an input array of $\boldsymbol{N}$ values.
- It has an average-case running time of $\boldsymbol{\theta(NlgN)}$, but it is remarkably efficient on the average because the constant factors in the $\theta(NlgN)$ notation are quite small (highly optimised inner loop).
- It has the advantage of sorting in place, i.e., transforms input without using auxiliary data structures.

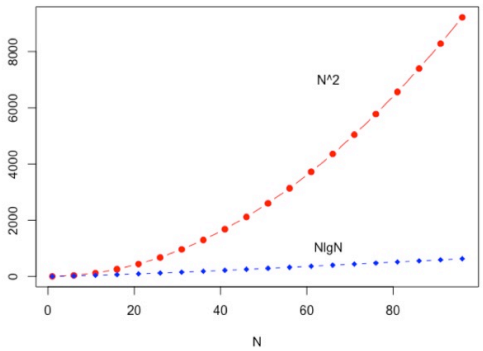
Picking a Right *Pivot* is Important

Worst-case running time:

$\boldsymbol{T(N) = \theta(N^2)}$

Best-case running time:

$\boldsymbol{T(N) = \theta(NlgN)}$



Quicksort Implementation (Optimized with Median of Three Partitioning Strategy)

Quicksort Implementation (Optimized with Median of Three Partitioning Strategy)

```java
private static final int CUTOFF = 3;

private static <E extends Comparable<E>>
void quicksort(E[] data, int left, int right) {
  if (left + CUTOFF <= right) {
    E pivot = medianOfThree(data, left, right);

    // Begin partitioning
    int i = left, j = right - 1;
    for (;;) {
      while (data[++i].compareTo(pivot) < 0) { }
      while (data[--j].compareTo(pivot) > 0) { }
      if (i < j)
        swap(data, i, j);
      else
        break;
    }

    swap(data, i, right - 1); // Restore pivot
    quicksort(data, left, i - 1); // Sort small elements
    quicksort(data, i + 1, right); // Sort large elements
  } else // Do an insertion sort on the subarray
    insertionSort(data, left, right);
}
```

```java
private static <E extends Comparable<E>>
void insertionSort(E[] data, int left, int right) {
  for (int p = left + 1; p <= right; p++) {
    E temp = data[p];
    int j;
    for (j = p; j > left && temp.compareTo(data[j - 1]) < 0; j--)
      data[j] = data[j - 1];

    data[j] = temp;
  }
}
```

```java
private static <E> void swap(E[] data, int idx1, int idx2) {
  E temp = data[idx1];
  data[idx1] = data[idx2];
  data[idx2] = temp;
}
```

```java
private static <E extends Comparable<E>>
E medianOfThree(E[] data, int left, int right) {
  int centre = (left + right) / 2;
  if (data[centre].compareTo(data[left]) < 0)
    swap(data, left, centre);
  if (data[right].compareTo(data[left]) < 0)
    swap(data, left, right);
  if (data[right].compareTo(data[centre]) < 0)
    swap(data, centre, right);

  // Place pivot at position 'right-1'
  swap(data, centre, right - 1);
  return data[right - 1];
}
```

历年卷习题

2019-Autumn-Q2

2 (a) Show how the following numbers would be sorted by median of three quicksort, identifying all of the pivots and swaps involved.

63 27 77 28 11 60 79 24 85

Median of Three Partitioning:

```
Current:  [11, 27, 60, 28, 24, 63, 79, 77, 85]
Current:  [11, 24, 28, 27, 60, 63, 79, 77, 85]
Current:  [11, 24, 27, 28, 60, 63, 79, 77, 85]
Current:  [11, 24, 27, 28, 60, 63, 77, 79, 85]
最终排序结果：[11, 24, 27, 28, 60, 63, 77, 79, 85]
```

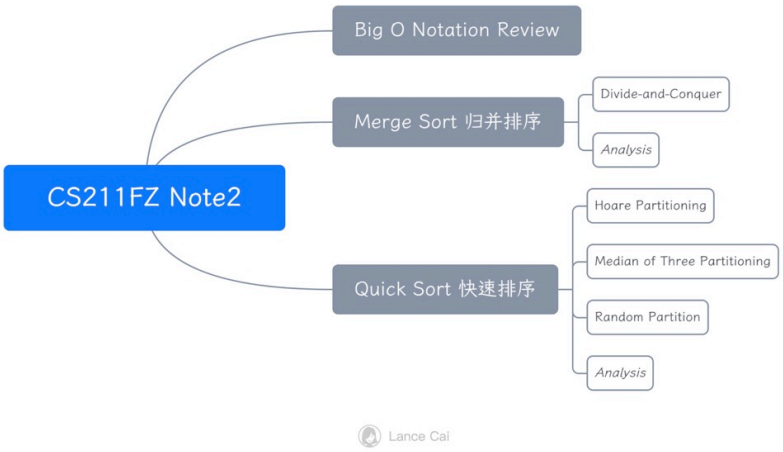
```java
public static void midThree(int[] array, int low, int high){
    //array[mid] <= array[low] <= array[high]  把中间大的数字放在第一个，最大的放在最后，最小的放在中间
    int mid = (high + low)/2;   //中间位置的下标
    if(array[low] > array[high]){
      swap(array,low,high);
    } //array[low] <= array[high]
    if(array[mid] > array[high]){
        swap(array,mid,high);
    }//array[mid] <= array[high]
    if(array[mid] > array[low]){
        swap(array,low,mid);
    }//array[mid] <= array[low]
}
```

可参考：http://t.csdn.cn/C6YvT

CS211FZ Note2 Review

Summary of the Running Times

Algorithm	Worst-case	Average-case
Insertion Sort	$\Theta(n^2)$	$\Theta(n^2)$
Merge Sort	$\Theta(n \lg n)$	$\Theta(n \lg n)$
Heapsort	$O(n \lg n)$	--
Quicksort	$\Theta(n^2)$	$\Theta(n \lg n)$
Counting sort	$\Theta(k+n)$	$\Theta(k+n)$
Radix sort	$\Theta(d(k+n))$	$\Theta(d(k+n))$
Bucket sort	$\Theta(n^2)$	$\Theta(n)$



CS211 Note-2

by Lance Cai

2022/07/05