week8

位运算

含义	运算符	示例
左移	<<	0011 => 0110
右移	>>	0110 => 0011

含义	运算符	示例
按位或	|	0011 ------- => 1011 1011
按位与	&	0011 ------- => 0011 1011
按位取反	~	0011 => 1100
按位异或（相同为零不同为一）	^	0011 ------- => 1000 1011

异或操作的一些特点：

x ^ 0 = x

x ^ 1s = ~x　　// 注意 1s = ~0

x ^ (~x) = 1s

x ^ x = 0

c = a ^ b　=>　a ^ c = b, b ^ c = a　　// 交换两个数

a ^ b ^ c = a ^ (b ^ c) = (a ^ b) ^ c　　// associative

1. 将x 最右边的n 位清零：x& (~0 << n)

//0取反全部都是1，向左移n位，1111000000，右边n位全是0

2. 获取x 的第n 位值（0 或者1）：(x >> n) & 1

//x右移n位，第n位就变成最后一位了

3. 获取x 的第n 位的幂值：x& (1 <<n)

4. 仅将第n 位置为1：x | (1 << n)

5. 仅将第n 位置为0：x & (~ (1 << n))

6. 将x 最高位至第n 位（含）清零：x& ((1 << n) –1)

实战位运算要点

（1）判断奇偶：（判断二进制最后一位是0还是1，比模操作快很多）

x % 2 == 1 —> (x & 1) == 1

x % 2 == 0 —> (x & 1) == 0

（2）x / 2

x >> 1 —> x / 2

即： x = x / 2; —> x = x >> 1;

mid = (left + right) / 2; —> mid = (left + right) >> 1; （二分查找中(左边界+右边界)/2)

（3）清零最低位的 1，将最末尾的1变为0

X = X & (X–1)

（4）得到最低位的 1

X & –X

（5）

X & ~X => 0

191.位1的个数

以下方法都需要算32次

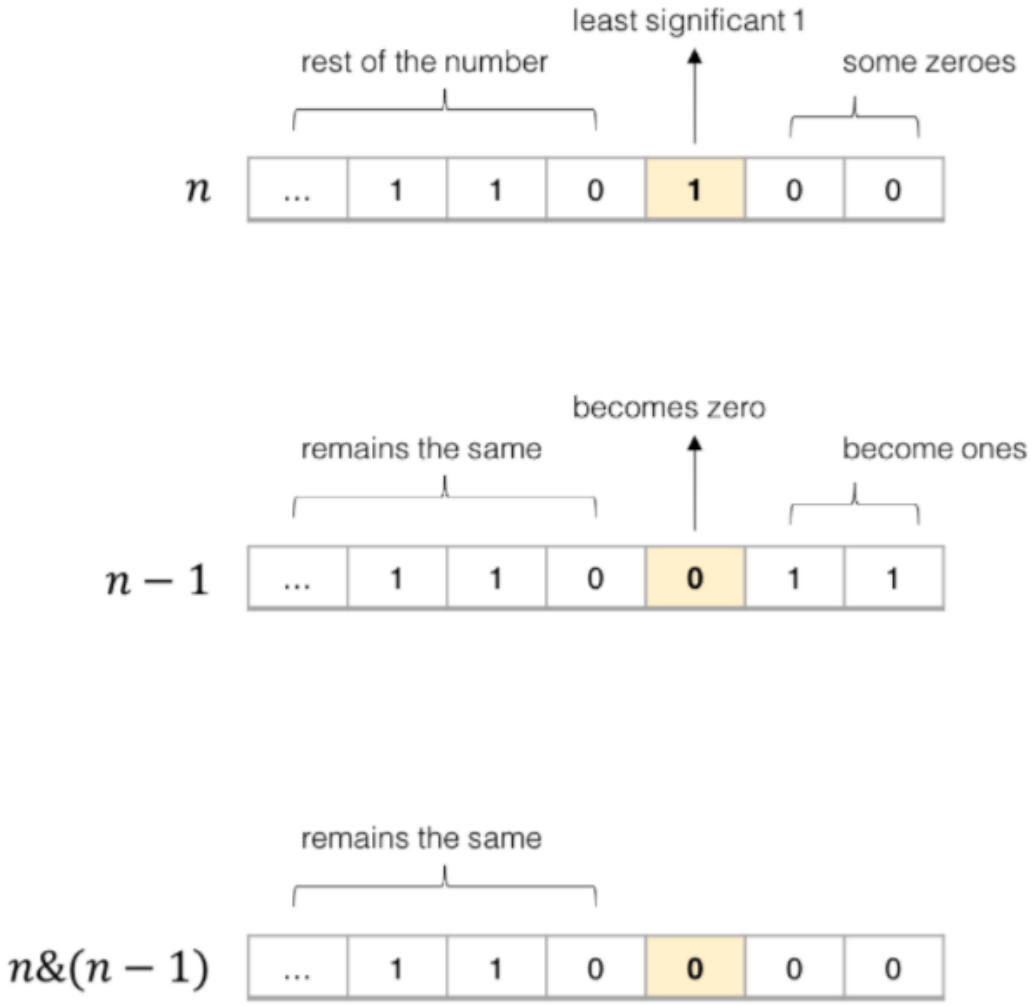
1.for loop：32次

2.%2，/2，相当于把最后一位打掉

3.&1，x = x >> 1

有多少个1就循环多少次

4.while(x > 0) {count++; x = x & (x – 1);}



图片 1. 将 n 和 $n-1$ 做与运算会将最低位的 1 变成 0

```
public int hammingWeight(int n) {
    int sum = 0;
    while (n != 0) {
        sum++;
        n &= (n - 1);
    }
    return sum;
}
```

布隆过滤

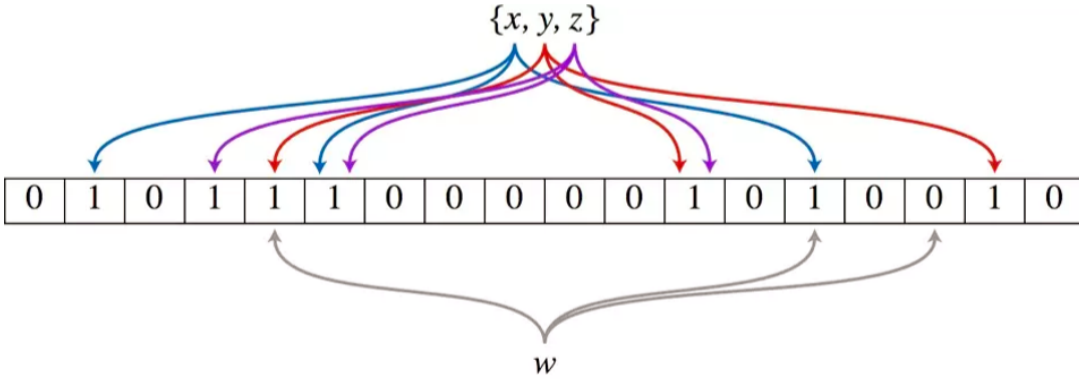
哈希表用哈希函数得到index值，且会把整个要存的元素全都存到哈希表中，有多少元素每个元素多大，所有元素占的内存空间，在哈希表中都要找相应的内存大小存起来，会存很多冗余的信息，是没有误差的数据结构。工业应用的时候，有时候只需要知道有没有，布隆过滤。

布隆过滤器是一个很长的二进制向量和一系列随机映射函数。布隆过滤器可以用于检索 一个元素是否在一个集合中。

优点是空间效率和查询时间都远远超过一般的算法，

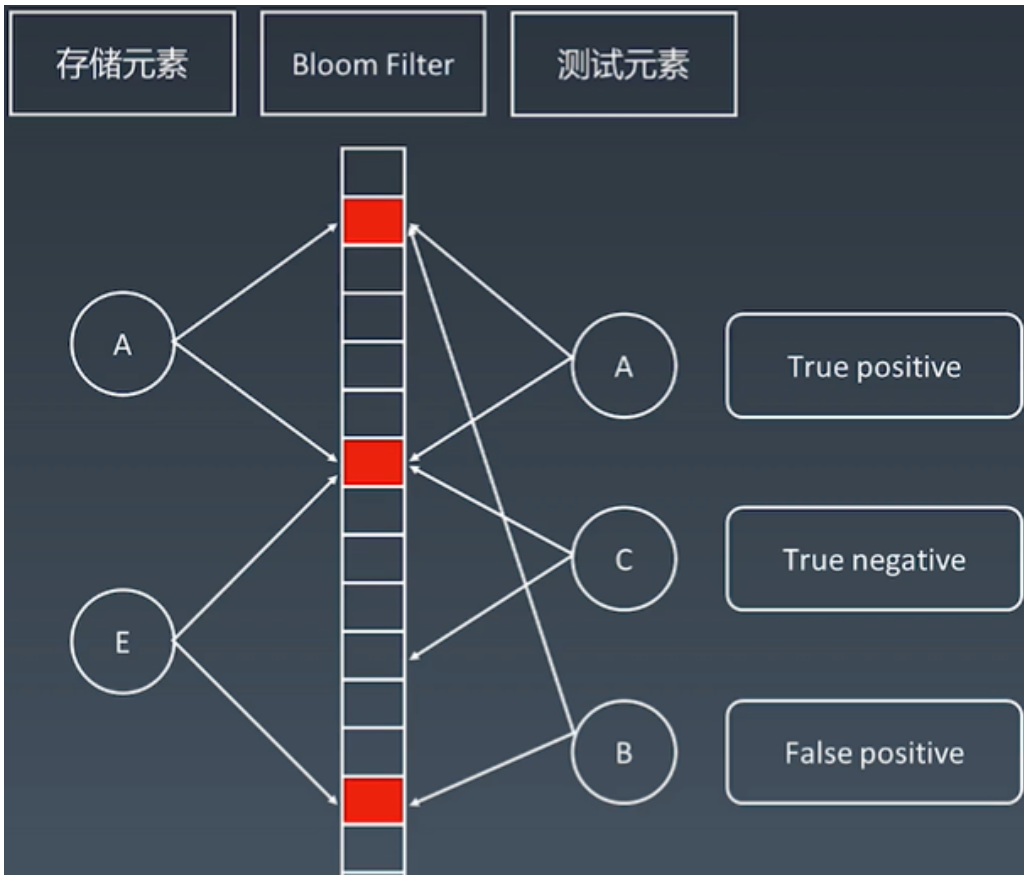
缺点是有一定的误识别率和删除困难。

给x, y, z分配二进制位，对于新来的元素w，看对应的二进制位是不是都为1。



C的二进制位不全为1，一定不在布隆过滤器里面。

B的二进制位都为1，但是B不在布隆过滤器里面。



元素去布隆过滤器里面查：

如果查到有二进制位为0，那就一定不在

如果查到二进制位都为1，可能存在

布隆过滤器只是放在机器最外面快速查询的一个缓存。如果B在布隆过滤器里面可能存在，B就会继续在这台机器的数据库里面查是否存在。C不在布隆过滤器，肯定不在这台机器的数据库里，节省了访问数据库的时间。

案例

1. 比特币网络

比特币是分布式系统，一个地址是否在这个节点里，以及transaction是否在node里面，都需要用到布隆过滤器。
2.分布式系统（Map-Reduce） — Hadoop、search engine
搜索引擎常做的事情就是把大量的网页信息和图片信息存在服务器里，一般不同的网页存在不同的集群里，search时根据索引得到在哪个集群，现在这个集群的布隆过滤器里面查一下，如果存在再去集群的DB里面查，如果不存在就直接略过。
3.Redis 缓存
4.垃圾邮件、评论等的过滤
科普：https://www.cnblogs.com/cpselvis/p/6265825.html https://blog.csdn.net/tianyaleixiaowu/article/details/74721877

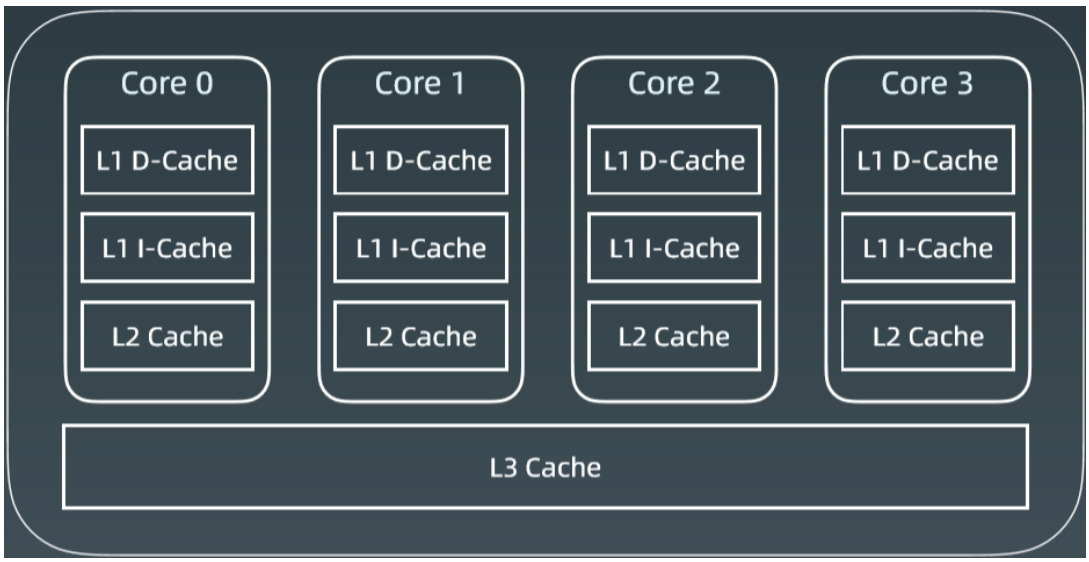
```
from bitarray import bitarray #存二进制位的数组
import mmh3
class BloomFilter:
    def __init__(self, size, hash_num):
        self.size = size
        self.hash_num = hash_num #一个元素分成多少个二进制位
        self.bit_array = bitarray(size)
        self.bit_array.setall(0) #开始二进制数组索引全为0
    def add(self, s):
        for seed in range(self.hash_num): #循环hash_num次
            result = mmh3.hash(s, seed) % self.size
            #seed和s哈希，再模bitarray的size，不能让下标超出
            self.bit_array[result] = 1 #result所在的二进制位置为1
    def lookup(self, s):
        for seed in range(self.hash_num): #循环hash_num次
            result = mmh3.hash(s, seed) % self.size
            #seed和s哈希，再模bitarray的size，找到对应的下标
            if self.bit_array[result] == 0:
                return "Nope"
        return "Probably"
bf = BloomFilter(500000, 7)
bf.add("dantezhao")
print (bf.lookup("dantezhao"))
print (bf.lookup("yyj"))
```

其他实现：
https://www.geeksforgeeks.org/bloom-filters-introduction-and-python-implementation/
https://github.com/jhgg/pybloof

LRU Cache

Intel四核处理器

最常用的数据，马上要给cpu计算模块进行计算处理的，放在L1 D-Cache里面，次之不太常用的，放在L1 I-Cache里面，再次之就放在L2 Cache里面，最后放在L3 Cache，再外面就是内存了。速度L1 D-Cache最快，能存数据的体积L3 Cache最大。

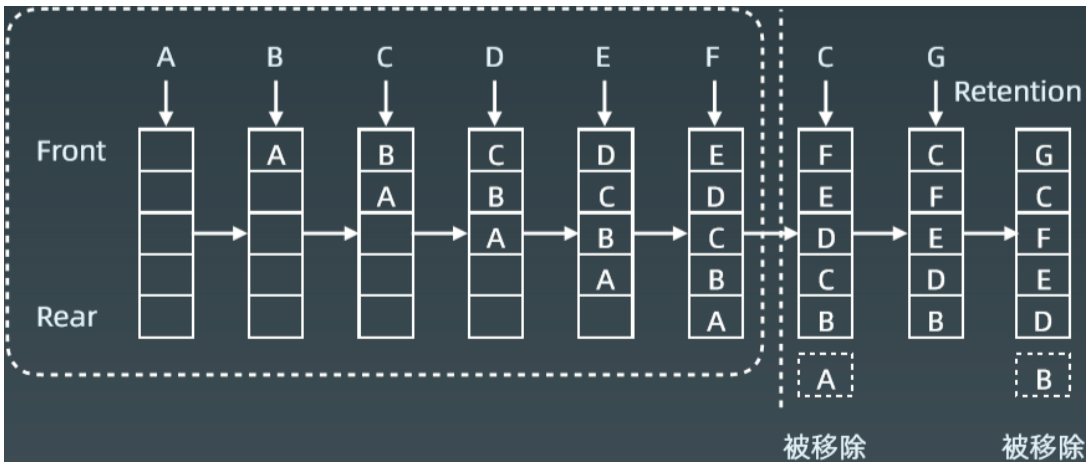


两个要素： 大小 、替换策略(鉴别哪些是不常用的)

least recently used 最近最少使用的放在最后淘汰，一般是用哈希表+双向链表实现

- Hash Table + Double LinkedList
- O(1) 查询
  O(1) 修改、更新

LRU Cache工作示例



替换策略

- LFU – least frequently used 使用频次最少的放在最下面，最先被淘汰
- LRU – least recently used

替换算法总览： https://en.wikipedia.org/wiki/Cache_replacement_policies

146. LRU缓存机制

排序算法

各种排序的动画：

https://www.cnblogs.com/onepixel/p/7674659.html

重点看3个O(nlogn)的排序，把原理弄得很清楚，代码实现记清楚。(n^2的排序不要花太多时间)

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

快速排序

数组取标杆 pivot，将小元素放 pivot左边，大元素放右侧，然后依次对左边和右边的子数组继续快排；以达到整个序列有序。

不申请新的内存空间，在原数组中互相调换位置。

partition函数返回pivot的下标，并把小于pivot的元素放左边，大于放右边。

如果a[i] < a[pivot]，就要交换a[counter]和a[i]，让所有小于a[pivot]的元素放在counter之前（counter表示最后大于pivot的元素的位置），最后counter的位置就是pivot的位置。

```
public static void quickSort(int[] array, int begin, int end) {
    if (end <= begin) return;
    int pivot = partition(array, begin, end);
    quickSort(array, begin, pivot - 1);
    quickSort(array, pivot + 1, end);
}
static int partition(int[] a, int begin, int end) {
    //partition返回pivot的下标，并把小于pivot的元素放左边，大于放右边
    // pivot: 标杆位置，counter: 小于pivot的元素的个数
    int pivot = end, counter = begin;
    for (int i = begin; i < end; i++) {
        if (a[i] < a[pivot]) {
            int temp = a[counter]; a[counter] = a[i]; a[i] = temp;
            //交换a[counter]和a[i]
```

```
            counter++;//a[i] < a[pivot],counter++,统计有多少个小于pivot
            //counter的值加上begin，就是最后pivot要挪过去的位置
        }
    }
    int temp = a[pivot]; a[pivot] = a[counter]; a[counter] = temp;
    //交换a[counter]和a[pivot]
    return counter;
}
```

```
def quick_sort(begin, end, nums):
    if begin >= end:
        return
    pivot_index = partition(begin, end, nums)
    quick_sort(begin, pivot_index-1, nums)
    quick_sort(pivot_index+1, end, nums)

def partition(begin, end, nums):
    pivot = nums[begin]
    mark = begin
    for i in range(begin+1, end+1):
        if nums[i] < pivot:
            mark +=1
            nums[mark], nums[i] = nums[i], nums[mark]
    nums[begin], nums[mark] = nums[mark], nums[begin]
    return mark
```

归并排序

1. 把长度为n的输入序列分成两个长度为n/2的子序列；
2. 对这两个子序列分别采用归并排序；
3. 将两个排序好的子序列合并成一个最终的排序序列

代码解释：

当成代码模板，记住，并灵活运用。

重要的是merge函数，从left到mid有序，从mid到right有序，如何合并为一个有序数组。

i-第一个数组的起始位置，j-第二个数组的起始位置，k-temp数组中已经填入的元素的个数，temp-归并排序必须额外申请一个内存空间，最后返回的数组，

两个数组有序合并在一起，永远是三段式while：

1st while：i 和 j 都没循环完，比较arr[i]和arr[j]，将较小者放入temp[k]，放了arr[i]就i++，放了arr[i]就j++，之后k++。这个循环完成后，可以保证 i 全部走完子数组或 j 全部走完

子数组。(要习惯这种简洁的代码书写方式)

2nd & 3rd while：如果 i 没有走完，把 i 的剩余部分放到temp[k]中，如果 j 没有走完，把 j 的剩余部分放到temp[k]中。

三个while做完之后，temp被装满且有序，此时temp为合并之后的总数组，再把temp的所有元素拷贝回array里面。

```
public static void mergeSort(int[] array, int left, int right){
    if (right <= left) return;
    int mid = (left + right) >> 1; // (left + right) / 2
    mergeSort(array, left, mid);
    mergeSort(array, mid + 1, right);
    merge(array, left, mid, right);
}
public static void merge(int[] arr, int left, int mid, int right) {
        int[] temp = new int[right - left + 1]; // 中间数组
        int i = left, j = mid + 1, k = 0;
        while (i <= mid && j <= right) {
            temp[k++] = arr[i] <= arr[j] ? arr[i++] : arr[j++];
        }
        while (i <= mid)   temp[k++] = arr[i++];
        while (j <= right) temp[k++] = arr[j++];
        for (int p = 0; p < temp.length; p++) {
            arr[left + p] = temp[p];
        }
        // 也可以用 System.arraycopy(a, start1, b, start2, length)
    }
```

```
def mergesort(nums, left, right):
    if right <= left:
        return
    mid = (left+right) >> 1
    mergesort(nums, left, mid)
    mergesort(nums, mid + 1, right)
    merge(nums, left, mid, right)
def merge(nums, left, mid, right):
    temp = []
    i = left
    j = mid + 1
    while i <= mid and j <= right:
        if nums[i] <= nums[j]:
            temp.append(nums[i])
```

```
            i += 1
        else:
            temp.append(nums[j])
            j += 1
    while i<=mid:
        temp.append(nums[i])
        i += 1
    while j<=right:
        temp.append(nums[j])
        j += 1
    nums[left:right+1] = temp
```

493.翻转对

```
class Solution {
    public int reversePairs(int[] nums) {
        return mergeSort(nums, 0, nums.length - 1);
    }
    private int mergeSort(int[] nums, int s, int e) { //返回值int
        if(s >= e) return 0;
        int mid = s + (e - s) / 2;
        int cnt = mergeSort(nums, s, mid) + mergeSort(nums, mid + 1, e);
        //左右分别调mergeSort统计重要翻转对，并加起来，且左右两部分排好序
        //再统计左右两个序列之间的重要翻转对
        for (int i = s, j = mid + 1; i <= mid; i++) {
            while(j <= e && nums[i] / 2.0 > nums[j]) j++; //防止
nums[j] * 2溢出，或转换为long，long(nums[i]) > 2*long(nums[j])
            cnt += j - (mid + 1);
        }
        Arrays.sort(nums, s, e + 1);
        //最后归并，为了简单，取巧调了系统的函数进行归并
        //复杂度(NlogN)会比merge(N)高一点。程序NlogN·logN
        return cnt;
    }
}
```

此题最好的写法，在统计数量的同时进行归并。

这两行是在归并

while (t <= mid && nums[t] < nums[j]) cache[c++] = nums[t++];

cache[c] = nums[j];

此时 j 是固定的，把小于nums[j]的左侧序列元素放到cache中，再把nums[j]放到cache中

```java
public class Solution {
    public int reversePairs(int[] nums) {
        if (nums == null || nums.length == 0) return 0;
        return mergeSort(nums, 0, nums.length - 1);
    }
    private int mergeSort(int[] nums, int l, int r) {
        if (l >= r) return 0;
        int mid = l + (r - l)/2;
        int count = mergeSort(nums, l, mid) + mergeSort(nums, mid + 1, r);
        int[] cache = new int[r - l + 1]; //生成一个中间数组
        int i = l, t = l, c = 0;
        for (int j = mid + 1; j <= r; j++, c++) {
            while (i <= mid && nums[i] <= 2 * (long)nums[j]) i++;//非翻转对数量
            while (t <= mid && nums[t] < nums[j]) cache[c++] = nums[t++];
            cache[c] = nums[j];
            count += mid - i + 1;
            //此次循环中，有多少个逆序对被找出来
        }
        while (t <= mid) cache[c++] = nums[t++];
        System.arraycopy(cache, 0, nums, l, r - l + 1);
        return count;
    }
}
```

堆排序

堆插入 O(logN)，取最大/小值 O(1)

1. 数组元素依次建立小顶堆
2. 依次取堆顶元素，并删除

```java
static void heapify(int[] array, int length, int i) {
    int left = 2 * i + 1, right = 2 * i + 2;
    int largest = i;
    if (left < length && array[left] > array[largest]) {
        largest = left;
    }
    if (right < length && array[right] > array[largest]) {
        largest = right;
```

```
    }
    if (largest != i) {
        int temp = array[i]; array[i] = array[largest]; array[largest] = temp;
        heapify(array, length, largest);
    }
}
public static void heapSort(int[] array) {
    if (array.length == 0) return;
    int length = array.length;
    for (int i = length / 2-1; i >= 0; i-)
        heapify(array, length, i);
    for (int i = length - 1; i >= 0; i--) {
        int temp = array[0]; array[0] = array[i]; array[i] = temp;
        heapify(array, i, 0);
    }
}
```

```
def heapify(parent_index, length, nums):
    temp = nums[parent_index]
    child_index = 2*parent_index+1
    while child_index < length:
        if child_index+1 < length and nums[child_index+1] > nums[child_index]:
            child_index = child_index+1
        if temp > nums[child_index]:
            break
        nums[parent_index] = nums[child_index]
        parent_index = child_index
        child_index = 2*parent_index + 1
    nums[parent_index] = temp
def heapsort(nums):
    for i in range((len(nums)-2)//2, -1, -1):
        heapify(i, len(nums), nums)
    for j in range(len(nums)-1, 0, -1):
        nums[j], nums[0] = nums[0], nums[j]
        heapify(0, j, nums)
```