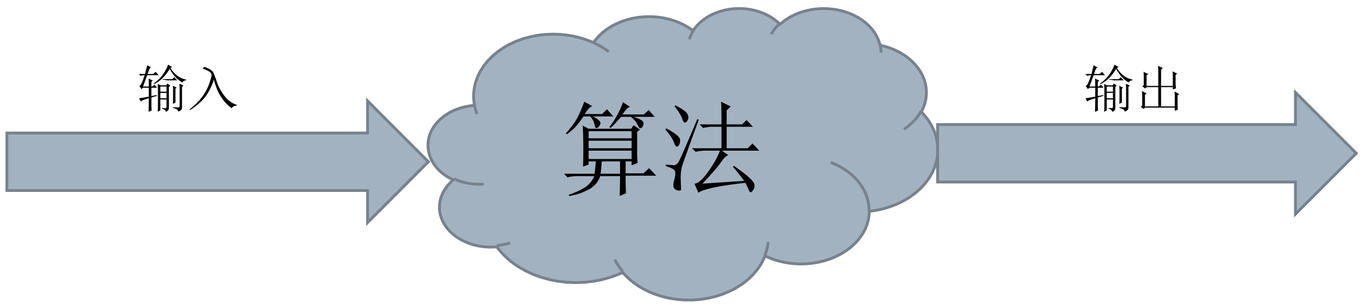
算法基础

本节内容

- 算法概念
- 时间复杂度
- 空间复杂度
- 复习：递归

算法

- 算法（Algorithm）：一个计算过程，解决问题的方法
- Niklaus Wirth: “程序=数据结构+算法”



时间复杂度

```
print('Hello World')
```

```
for i in range(n):
    print('Hello World')
```

```
for i in range(n):
    for j in range(n):
        print('Hello World')
```

```
for i in range(n):
    for j in range(n):
        for k in range(n):
            print('Hello World')
```

- 左面四组代码，哪组运行时间最短?
- 用什么方式来体现算法运行的快慢?

时间复杂度

- 类比生活中的一些事件，估计时间：
  - 眨一下眼 —— 一瞬间/几毫秒
  - 口算“29+68” —— 几秒
  - 烧一壶水 —— 几分钟
  - 睡一觉 —— 几小时
  - 完成一个项目 —— 几天/几星期/几个月
  - 飞船从地球飞出太阳系 —— 几年

时间复杂度

- 时间复杂度：用来评估算法运行效率的一个式子

```
print('Hello World')
```

$O(1)$

```
for i in range(n):
    print('Hello World')
```

$O(n)$

```
for i in range(n):
    for j in range(n):
        print('Hello World')
```

$O(n^2)$

```
for i in range(n):
    for j in range(n):
        for k in range(n):
            print('Hello World')
```

$O(n^3)$

时间复杂度

```
print('Hello World')
print('Hello Python')
print(‘Hello Algorithm’)
```

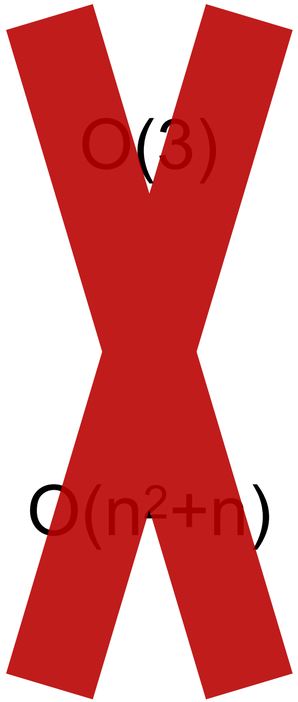
O(3)

O(1)

```
for i in range(n):
    print('Hello World’)
    for j in range(n):
        print('Hello World')
```

$O(n^2+n)$

$O(n^2)$



时间复杂度

```
while n > 1:
    print(n)
    n = n // 2
```

n=64输出：

64
32
16
8
4
2

$2^6=64$

$\log_2 64=6$

- 时间复杂度记为$O(\log_2 n)$或$O(\log n)$
- 当算法过程出现循环折半的时候，复杂度式子中会出现$\log n$.

时间复杂度-小结

- 时间复杂度是用来估计算法运行时间的一个式子（单位）。
- 一般来说，时间复杂度高的算法比复杂度低的算法慢。
- 常见的时间复杂度（按效率排序）
  - $O(1)<O(\log n)<O(n)<O(n\log n)<O(n^2)<O(n^2\log n)<O(n^3)$
- 复杂问题的时间复杂度
  - $O(n!)$ $O(2^n)$ $O(n^n)$ …

如何简单快速地判断算法复杂度

- 快速判断算法复杂度（适用于绝大多数简单情况）：
  - 确定问题规模n
  - 循环减半过程→logn
  - k层关于n的循环→n^k
- 复杂情况：根据算法执行过程判断

空间复杂度

- 空间复杂度：用来评估算法内存占用大小的式子
- 空间复杂度的表示方式与时间复杂度完全一样
  - 算法使用了几个变量：O(1)
  - 算法使用了长度为n的一维列表：O(n)
  - 算法使用了m行n列的二维列表：O(mn)
- “空间换时间”

复习：递归

- 递归的两个特点：
  - 调用自身
  - 结束条件

```
def func1(x):
    print(x)
    func1(x-1)
```

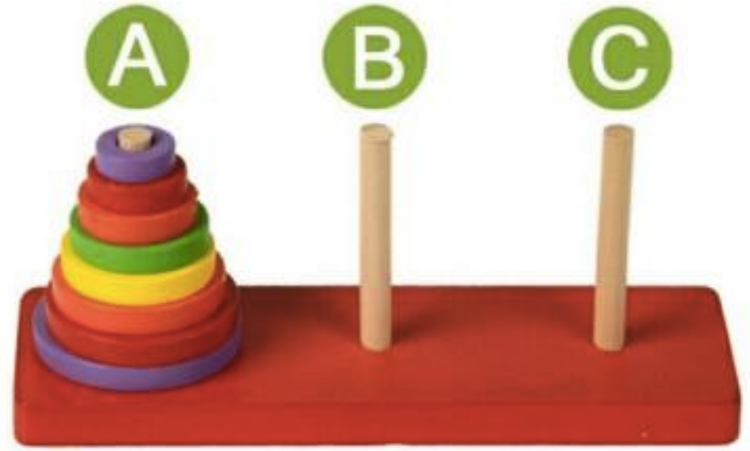
```
def func2(x):
    if x>0:
        print(x)
        func2(x+1)
```

```
def func3(x):
    if x>0:
        print(x)
        func3(x-1)
```

```
def func4(x):
    if x>0:
        func4(x-1)
        print(x)
```

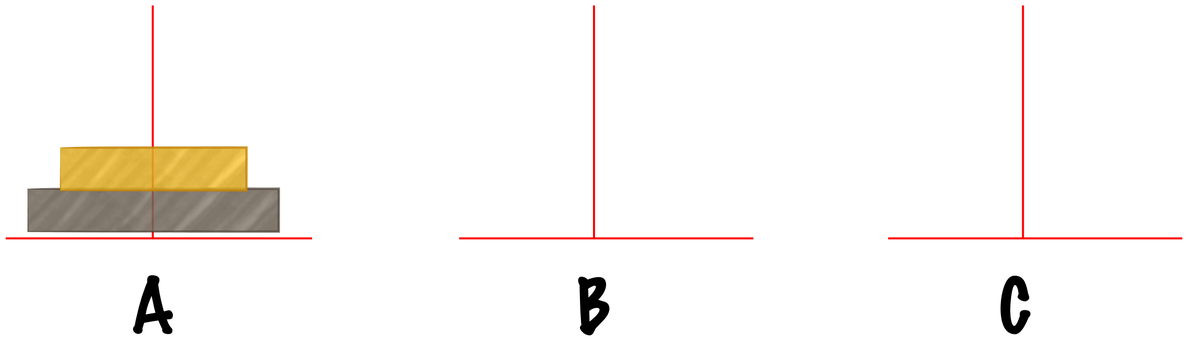
递归实例：汉诺塔问题

- 大梵天创造世界的时候做了三根金刚石柱子，在一根柱子上从下往上按照大小顺序摞着64片黄金圆盘。
- 大梵天命令婆罗门把圆盘从下面开始按大小顺序重新摆放在另一根柱子上。
- 在小圆盘上不能放大圆盘，在三根柱子之间一次只能移动一个圆盘。
- 64根柱子移动完毕之日，就是**世界毁灭之时**。



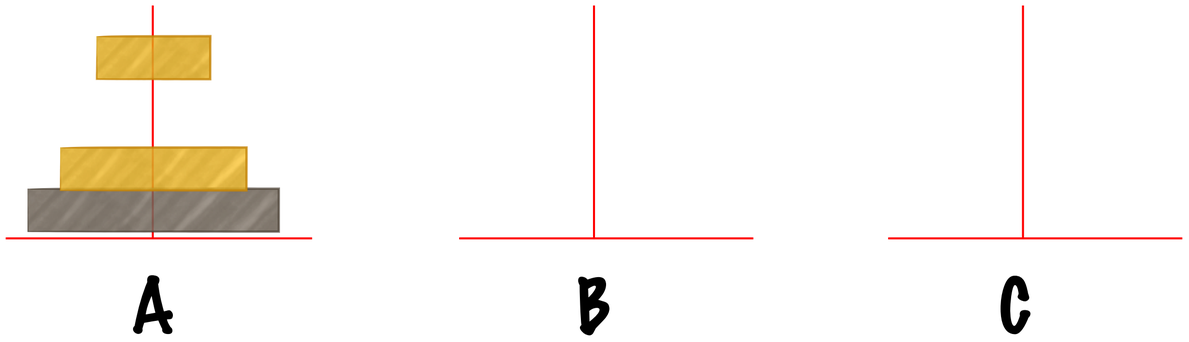
递归实例：汉诺塔问题

- n=2时：
  - 1.把小圆盘从A移动到B
  - 2.把大圆盘从A移动到C
  - 3.把小圆盘从B移动到C



递归实例：汉诺塔问题

- n个盘子时：
  - 1.把n–1个圆盘从A经过C移动到B
  - 2.把第n个圆盘从A移动到C
  - 3.把n–1个小圆盘从B经过A移动到C



递归实例：汉诺塔问题

```
def hanoi(n, a, b, c):
    if n > 0:
        hanoi(n-1, a, c, b)
        print("#%d: moving from %s to %s." % (num, a, c))
        hanoi(n-1, b, a, c)
```

递归实例：汉诺塔

- 汉诺塔移动次数的递推式：$h(x)=2h(x-1)+1$
- $h(64)=18446744073709551615$
- 假设婆罗门每秒钟搬一个盘子，则总共需要**5800亿年**！

列表查找

本节内容

- 什么是列表查找
- 顺序查找
- 二分查找

查找

- 查找：在一些数据元素中，通过一定的方法找出与给定关键字相同的数据元素的过程。
- 列表查找（线性表查找）：从列表中查找指定元素
  - 输入：列表、待查找元素
  - 输出：元素下标（未找到元素时一般返回None或-1）
- 内置列表查找函数：index()

顺序查找 (Linear Search)

- 顺序查找：也叫线性查找，从列表第一个元素开始，顺序进行搜索，直到找到元素或搜索到列表最后一个元素为止。
- 时间复杂度：O(n)

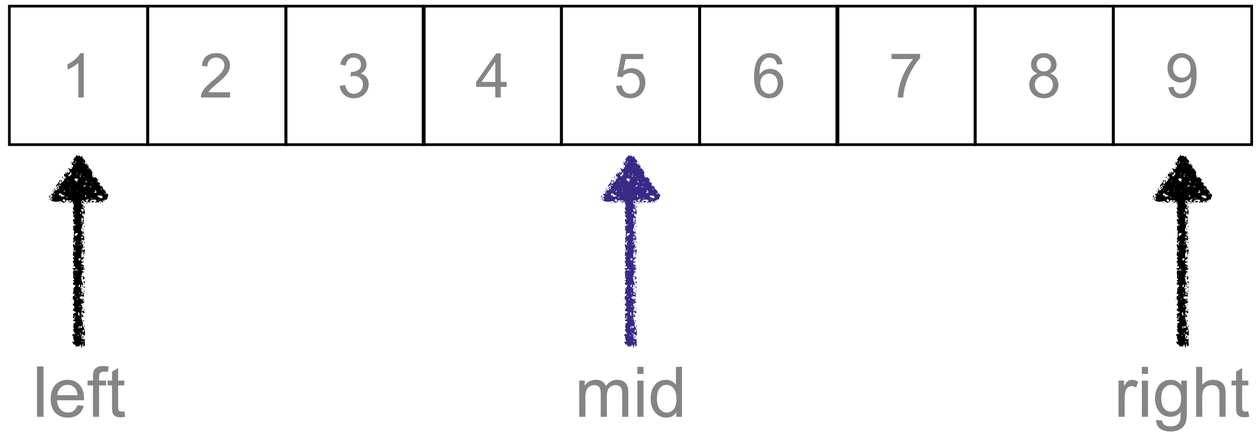
```
def linear_search(data_set, value):
    for i in range(range(data_set)):
        if data_set[i] == value:
            return i
    return
```

二分查找 (Binary Searh)

- 二分查找：又叫折半查找，从有序列表的初始候选区li[0:n]开始，通过对待查找的值与候选区中间值的比较，可以使候选区减少一半。

二分查找-实例

- 从列表中查找元素3：



二分查找

```
def bin_search(data_set, value):
    low = 0
    high = len(data_set) - 1
    while low <= high:
        mid = (low+high)//2
        if data_set[mid] == value:
            return mid
        elif data_set[mid] > value:
            high = mid - 1
        else:
            low = mid + 1
```

- 时间复杂度：O(logn)

列表排序

本节内容

- 什么是列表排序
- 常见排序算法介绍
- 排序算法分析

- 排序：将一组“无序”的记录序列调整为“有序”的记录序列。
- 列表排序：将无序列表变为有序列表
  - 输入：列表
  - 输出：有序列表
- 升序与降序
- 内置排序函数：sort()

常见排序算法

- 排序Low B三人组
  - 冒泡排序
  - 选择排序
  - 插入排序
- 排序NB三人组
  - 快速排序
  - 堆排序
  - 归并排序
- 其他排序
  - 希尔排序
  - 计数排序
  - 基数排序

冒泡排序 (Bubble Sort)

- 列表每两个相邻的数，如果前面比后面大，则交换这两个数。
- 一趟排序完成后，则无序区减少一个数，有序区增加一个数。
- 代码关键点：趟、无序区范围

冒泡排序

```
def bubble_sort(li):
    for i in range(len(li)-1):
        for j in range(len(li)-i-1):
            if li[j] > li[j+1]:
                li[j], li[j+1] = li[j+1], li[j]
```

- 时间复杂度：$O(n^2)$

冒泡排序-优化

- 如果冒泡排序中的一趟排序没有发生交换，则说明列表已经有序，可以直接结束算法。

```
def bubble_sort_1(li):
    for i in range(len(li)-1):
        exchange = False
        for j in range(len(li)-i-1):
            if li[j] > li[j+1]:
                li[j], li[j+1] = li[j+1], li[j]
                exchange = True
        if not exchange:
            return
```

选择排序 (Select Sort)

- 一趟排序记录最小的数，放到第一个位置
- 再一趟排序记录记录列表无序区最小的数，放到第二个位置
- ……
- 算法关键点：有序区和无序区、无序区最小数的位置

选择排序

- 时间复杂度：$O(n^2)$

```
def select_sort(li):
    for i in range(len(li) - 1):
        min_loc = i
        for j in range(i+1, len(li)):
            if li[j] < li[min_loc]:
                min_loc = j
        if min_loc != i:
            li[i], li[min_loc] = li[min_loc], li[i]
```

插入排序

- 初始时手里(有序区)只有一张牌
- 每次(从无序区)摸一张牌，插入到手里已有牌的正确位置

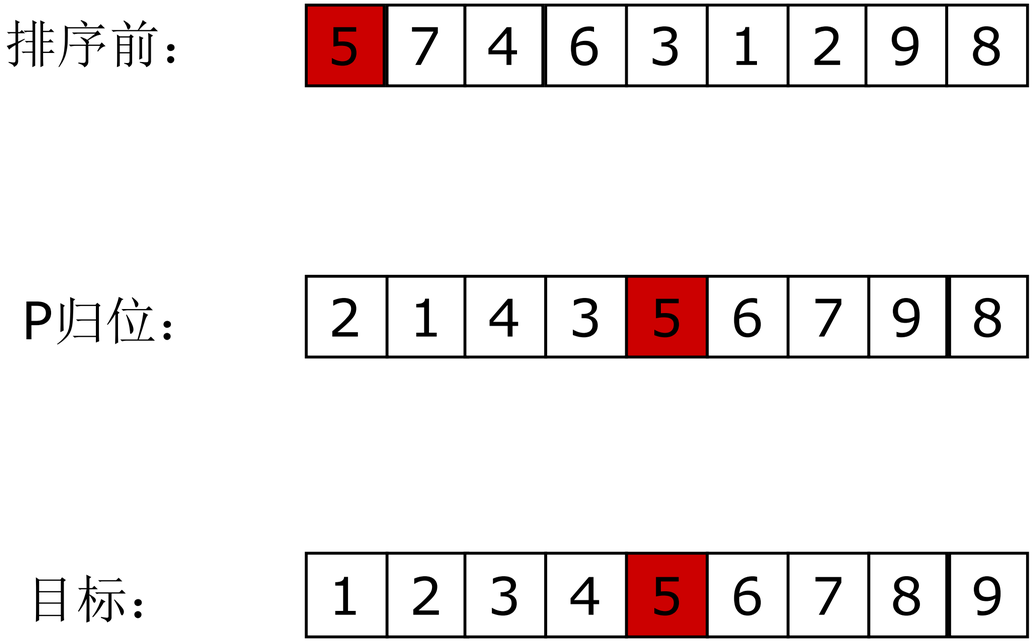
插入排序

- 时间复杂度：$O(n^2)$

```
def insert_sort(li):
    for i in range(1, len(li)):
        tmp = li[i]
        j = i - 1
        while j >= 0 and tmp < li[j]:
            li[j + 1] = li[j]
            j = j - 1
        li[j + 1] = tmp
```

快速排序

- 快速排序：快
- 快速排序思路：
  - 取一个元素p（第一个元素），使元素p归位；
  - 列表被p分成两部分，左边都比p小，右边都比p大；
  - 递归完成排序。



快速排序-框架

```
def quick_sort(data, left, right):
    if left < right:
        mid = partition(data, left, right)
        quick_sort(data, left, mid - 1)
        quick_sort(data, mid + 1, right)
```

快速排序-partition函数

```
def partition(data, left, right):
    tmp = data[left]
    while left < right:
        while left < right and data[right] >= tmp:
            right -= 1
        data[left] = data[right]
        while left < right and data[left] <= tmp:
            left += 1
        data[right] = data[left]
    data[left] = tmp
    return left
```

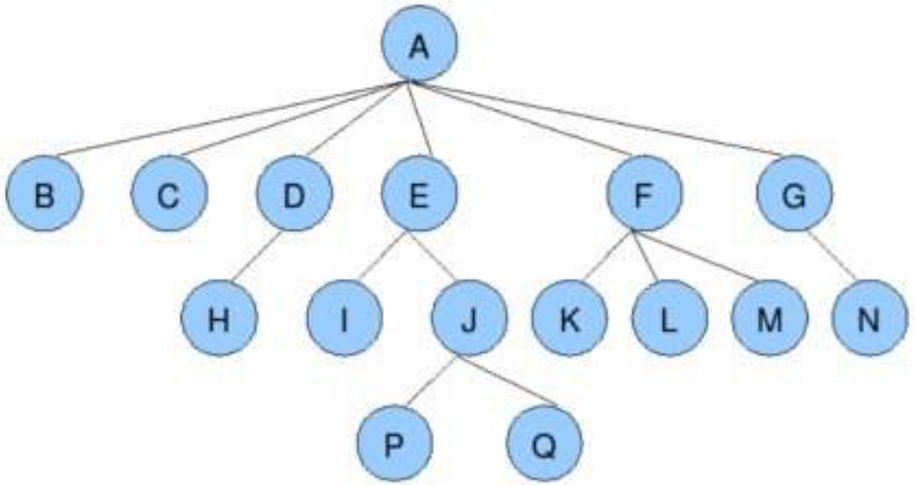
快速排序

- 快速排序的效率：
  - 快速排序的时间复杂度　O(nlogn)
- 快速排序的问题：
  - 最坏情况
  - 递归

堆排序

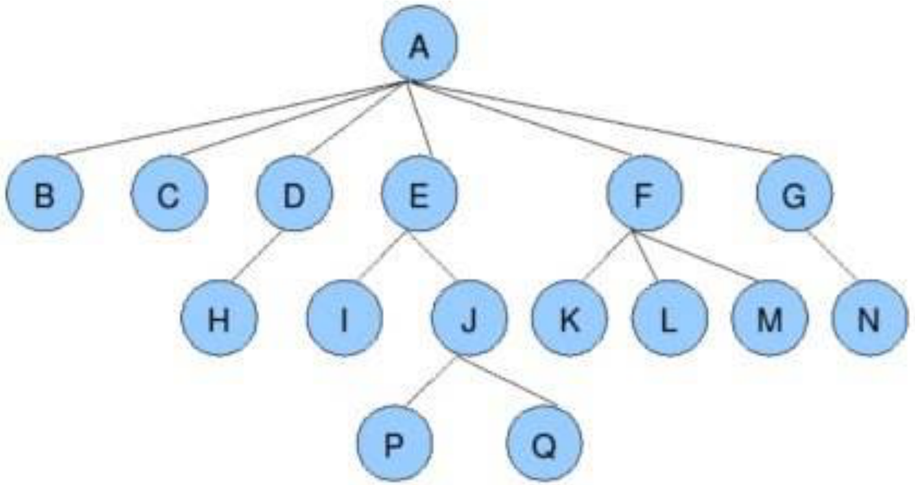
堆排序前传-树与二叉树

- 树是一种数据结构　　比如：目录结构
- 树是一种可以递归定义的数据结构
- 树是由n个节点组成的集合：
  - 如果n=0，那这是一棵空树；
  - 如果n>0，那存在1个节点作为树的根节点，其他节点可以分为m个集合，每个集合本身又是一棵树。



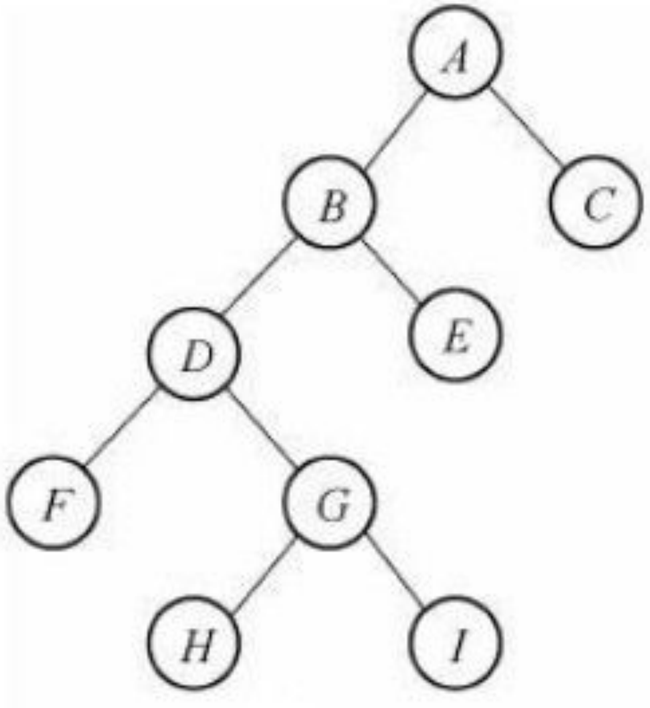
堆排序前传-树与二叉树

- 一些概念
  - 根节点、叶子节点
  - 树的深度（高度）
  - 树的度
  - 孩子节点/父节点
  - 子树



堆排序前传——二叉树

- 二叉树：度不超过2的树
- 每个节点最多有两个孩子节点
- 两个孩子节点被区分为左孩子节点和右孩子节点



堆排序前传-完全二叉树

- 满二叉树：一个二叉树，如果每一个层的结点数都达到最大值，则这个二叉树就是满二叉树。
- 完全二叉树：叶节点只能出现在最下层和次下层，并且最下面一层的结点都集中在该层最左边的若干位置的二叉树。

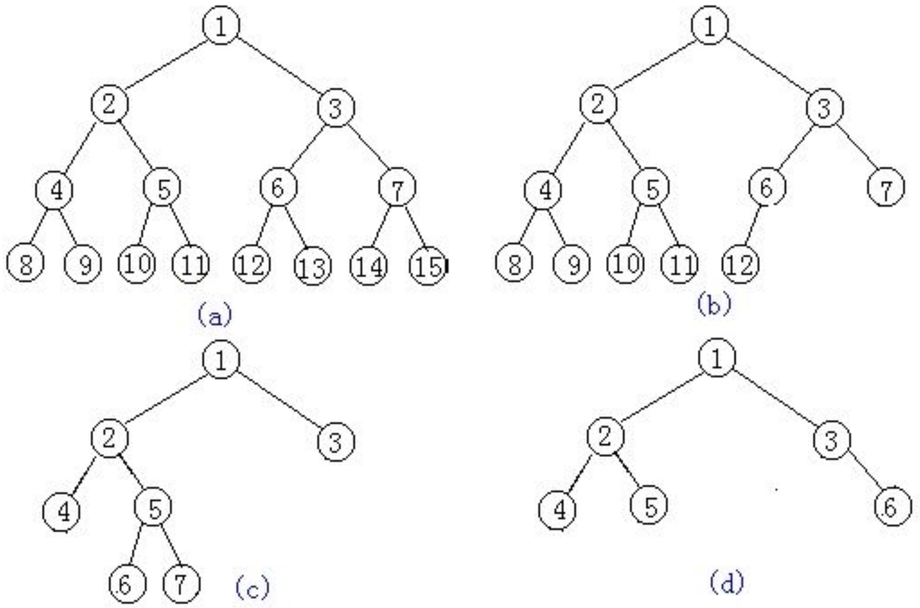


图6.5　特殊形态的二叉树

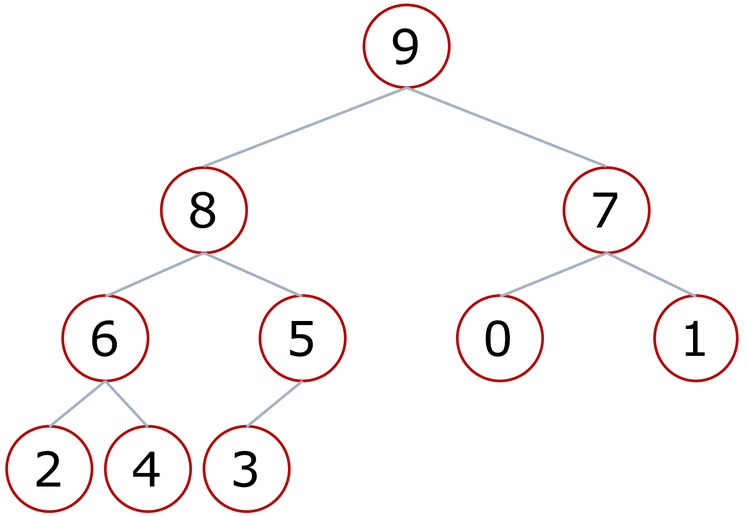
（a）满二叉树；（b）完全二叉树；（c）和（d）非完全二叉树。

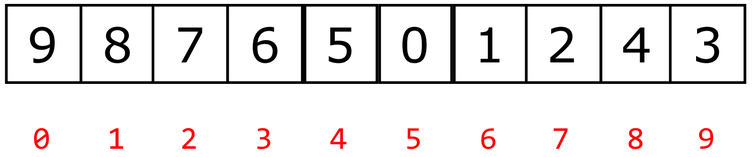
堆排序前传——二叉树的存储方式

- 二叉树的存储方式（表示方式）
  - 链式存储方式
  - 顺序存储方式

堆排序前传——二叉树的顺序存储方式

- 父节点和左孩子节点的编号下标有什么关系?
  - 0–1 1–3 2–5 3–7 4–9
  - i → 2i+1
- 父节点和右孩子节点的编号下标有什么关系?
  - 0–2 1–4 2–6 3–8 4–10
  - i → 2i+2





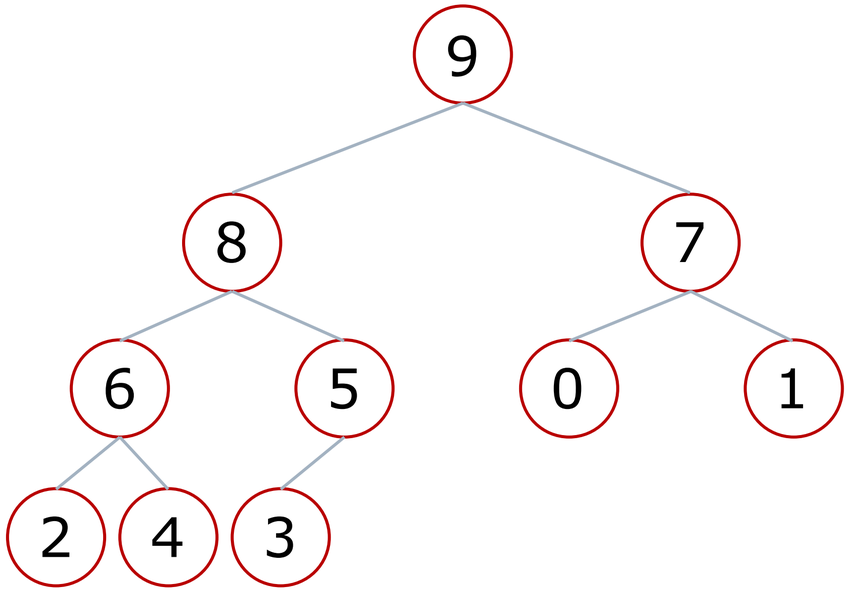
堆排序前传——小结

- 树
- 二叉树
- 完全二叉树
- 完全二叉树的顺序存储方式

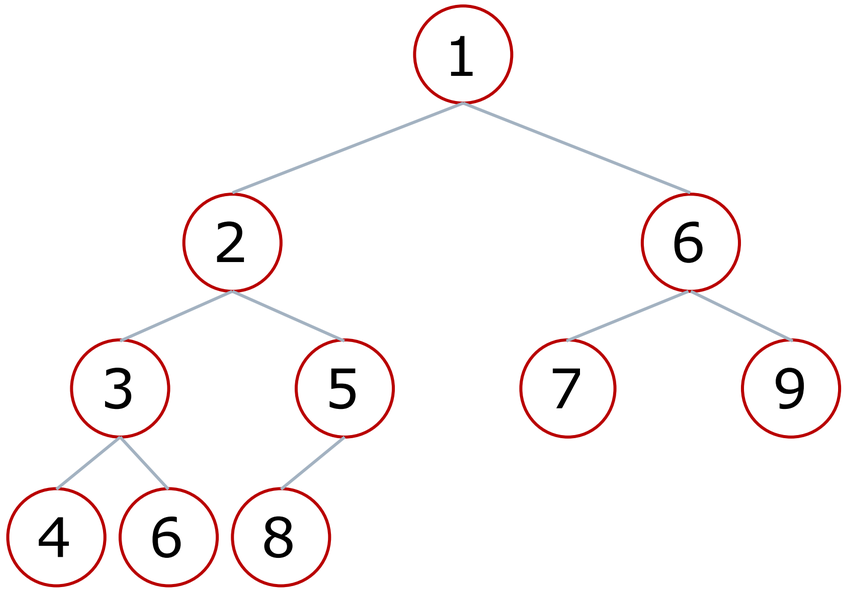
堆排序——什么是堆

- 堆：一种特殊的完全二叉树结构
  - 大根堆：一棵完全二叉树，满足任一节点都比其孩子节点大
  - 小根堆：一棵完全二叉树，满足任一节点都比其孩子节点小

大根堆：



小根堆：



- 大根堆：一棵完全二叉树，满足任一节点都比其孩子节点大
- 小根堆：一棵完全二叉树，满足任一节点都比其孩子节点小

堆排序——堆的向下调整性质

- 假设根节点的左右子树都是堆，但根节点不满足堆的性质
- 可以通过一次向下的调整来将其变成一个堆。

堆排序过程

- 1.建立堆。
- 2.得到堆顶元素，为最大元素
- 3.去掉堆顶，将堆最后一个元素放到堆顶，此时可通过一次调整重新使堆有序。
- 4.堆顶元素为第二大元素。
- 5.重复步骤3，直到堆变空。

堆排序

```
def sift(data, low, high):
    i = low
    j = 2 * i + 1
    tmp = data[i]
    while j <= high:
        if j < high and data[j] < data[j + 1]:
            j += 1
        if tmp < data[j]:
            data[i] = data[j]
            i = j
            j = 2 * i + 1
        else:
            break
    data[i] = tmp
```

```
def heap_sort(data):
    n = len(data)
    for i in range(n // 2 - 1, -1, -1):
        sift(data, i, n - 1)
    for i in range(n - 1, -1, -1):
        data[0], data[i] = data[i], data[0]
        sift(data, 0, i - 1)
```

堆排序——内置模块

- Python内置模块——heapq
- 常用函数
  - heapify(x)
  - heappush(heap,item)
  - heappop(heap)

堆排序——topk问题

- 现在有n个数，设计算法得到前k大的数。（k<n）
- 解决思路：
  - 排序后切片　　O(nlogn)
  - 排序LowB三人组　　O(mn)

堆排序——topk问题

- 解决思路：
  - 取列表前k个元素建立一个小根堆。堆顶就是目前第k大的数。
  - 依次向后遍历原列表，对于列表中的元素，如果小于堆顶，则忽略该元素；如果大于堆顶，则将堆顶更换为该元素，并且对堆进行一次调整；
  - 遍历列表所有元素后，倒序弹出堆顶。

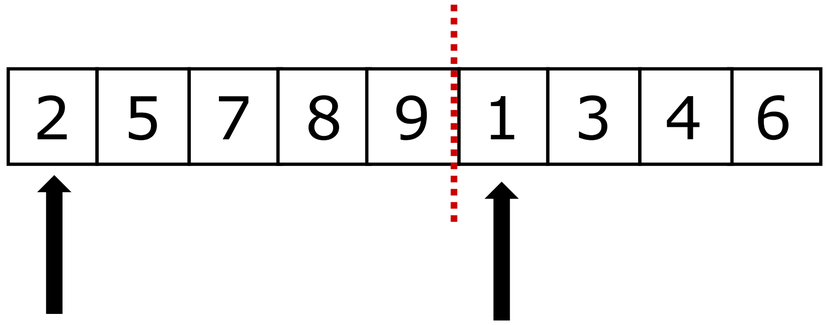
堆排序——topk问题

```
def topk(li, k):
    heap = li[0:k]
    for i in range(k // 2 - 1, -1, -1):
        sift(heap, i, k-1)
    for i in range(k, len(li)):
        if li[i] > heap[0]:
            heap[0] = li[i]
            sift(heap, 0, k - 1)
    for i in range(k - 1, -1, -1):
        heap[0], heap[i] = heap[i], heap[0]
        sift(heap, 0, i - 1)
```

归并排序

归并排序——归并

- 假设现在的列表分两段有序，如何将其合成为一个有序列表



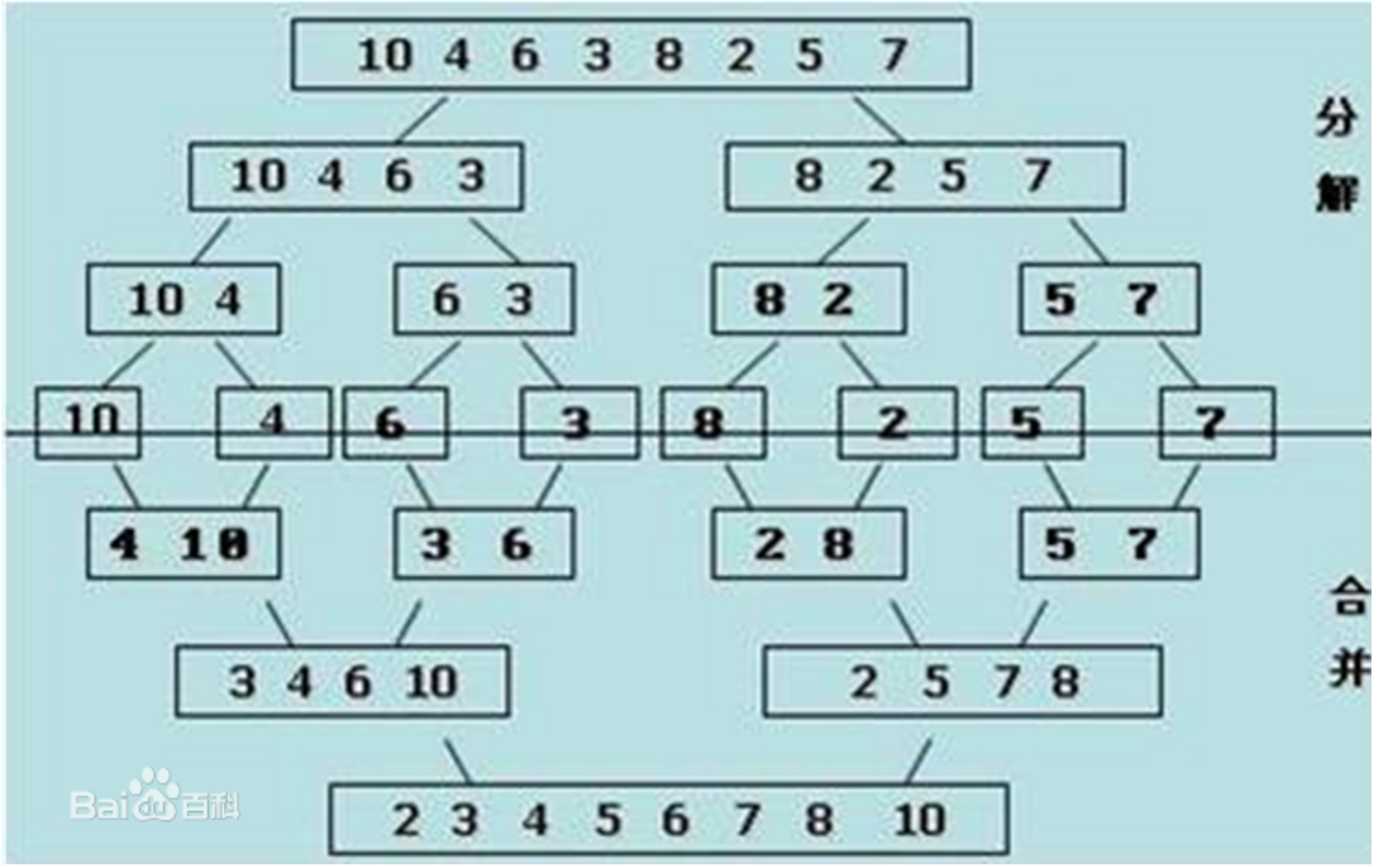
- 这种操作称为一次归并。

一次归并——代码

```
def merge(li, low, mid, high):
    i = low
    j = mid + 1
    ltmp = []
    while i <= mid and j <= high:
        if li[i] <= li[j]:
            ltmp.append(li[i])
            i += 1
        else:
            ltmp.append(li[j])
            j += 1
    while i <= mid:
        ltmp.append(li[i])
        i += 1
    while j <= high:
        ltmp.append(li[j])
        j += 1
    li[low:high + 1] = ltmp
```

归并排序——使用归并

- 分解：将列表越分越小，直至分成一个元素。
- 终止条件：一个元素是有序的。
- 合并：将两个有序列表归并，列表越来越大。


10 4 6 3 8 2 5 7
10 4 6 3
8 2 5 7
10 4
6 3
8 2
5 7
10
4
6
3
8
2
5
7
4 10
3 6
2 8
5 7
3 4 6 10
2 5 7 8
2 3 4 5 6 7 8 10
分解
合并
Bai百科

归并排序——代码

```
def mergesort(li, low, high):
    if low < high:
        mid = (low + high) // 2
        mergesort(li, low, mid)
        mergesort(li, mid + 1, high)
        merge(li, low, mid, high)
```

时间复杂度：O(nlogn)
空间复杂度：O(n)

NB三人组小结

- 三种排序算法的时间复杂度都是O(nlogn)
- 一般情况下，就运行时间而言：
  - 快速排序 < 归并排序 < 堆排序
- 三种排序算法的缺点：
  - 快速排序：极端情况下排序效率低
  - 归并排序：需要额外的内存开销
  - 堆排序：在快的排序算法中相对较慢

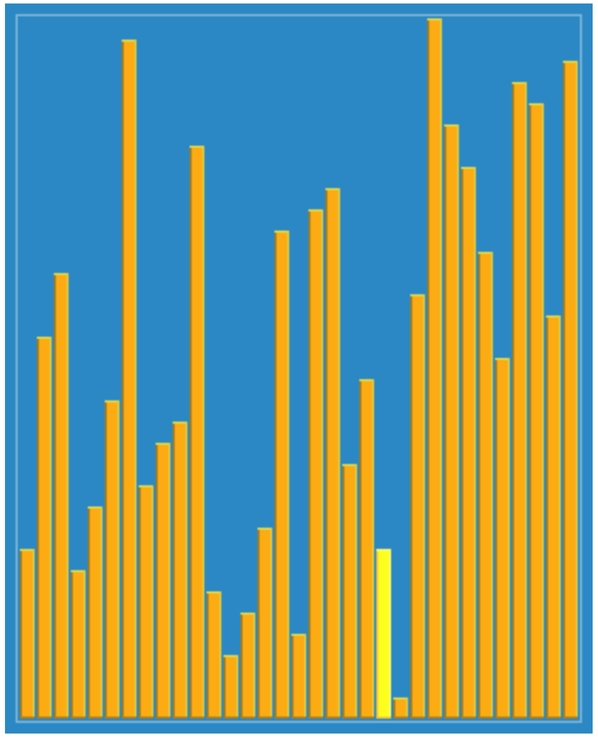
NB三人组小结

排序方法	时间复杂度			空间复杂度	稳定性	代码复杂度
	最坏情况	平均情况	最好情况			
冒泡排序	$O(n^2)$	$O(n^2)$	O(n)	O(1)	稳定	简单
直接选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	O(1)	不稳定	简单
直接插入排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	O(1)	稳定	简单
快速排序	$O(n^2)$	O(nlogn)	O(nlogn)	平均情况O(logn)； 最坏情况O(n)	不稳定	较复杂
堆排序	O(nlogn)	O(nlogn)	O(nlogn)	O(1)	不稳定	复杂
归并排序	O(nlogn)	O(nlogn)	O(nlogn)	O(n)	稳定	较复杂

希尔排序

希尔排序

- 希尔排序(Shell Sort)是一种分组插入排序算法。
- 首先取一个整数$d_1=n/2$，将元素分为d_1个组，每组相邻量元素之间距离为d_1，在各组内进行直接插入排序；
- 取第二个整数$d_2=d_1/2$，重复上述分组排序过程，直到$d_i=1$，即所有元素在同一组内进行直接插入排序。
- 希尔排序每趟并不使某些元素有序，而是使整体数据越来越接近有序；最后一趟排序使得所有数据有序。



Shellsort with gaps 23, 10, 4, 1 in action.

希尔排序——代码

```
def shell_sort(li):
    gap = len(li) // 2
    while gap > 0:
        for i in range(gap, len(li)):
            tmp = li[i]
            j = i - gap
            while j >= 0 and tmp < li[j]:
                li[j + gap] = li[j]
                j -= gap
            li[j+gap] = tmp
        gap /= 2
```

希尔排序——讨论

- 希尔排序的时间复杂度讨论比较复杂，并且和选取的gap序列有关。

计数排序

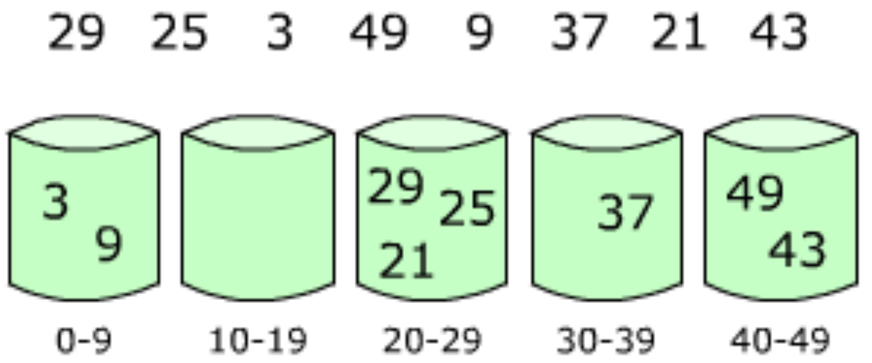
- 对列表进行排序，已知列表中的数范围都在0到100之间。设计时间复杂度为O(n)的算法。

计数排序——代码

```
def count_sort(li, max_num):
    count = [0 for i in range(max_num + 1)]
    for num in li:
        count[num] += 1
    i = 0
    for num,m in enumerate(count):
        for j in range(m):
            li[i] = num
            i += 1
```

桶排序

- 在计数排序中，如果元素的范围比较大（比如在1到1亿之间），如何改造算法?
- 桶排序(Bucket Sort)：首先将元素分在不同的桶中，在对每个桶中的元素排序。

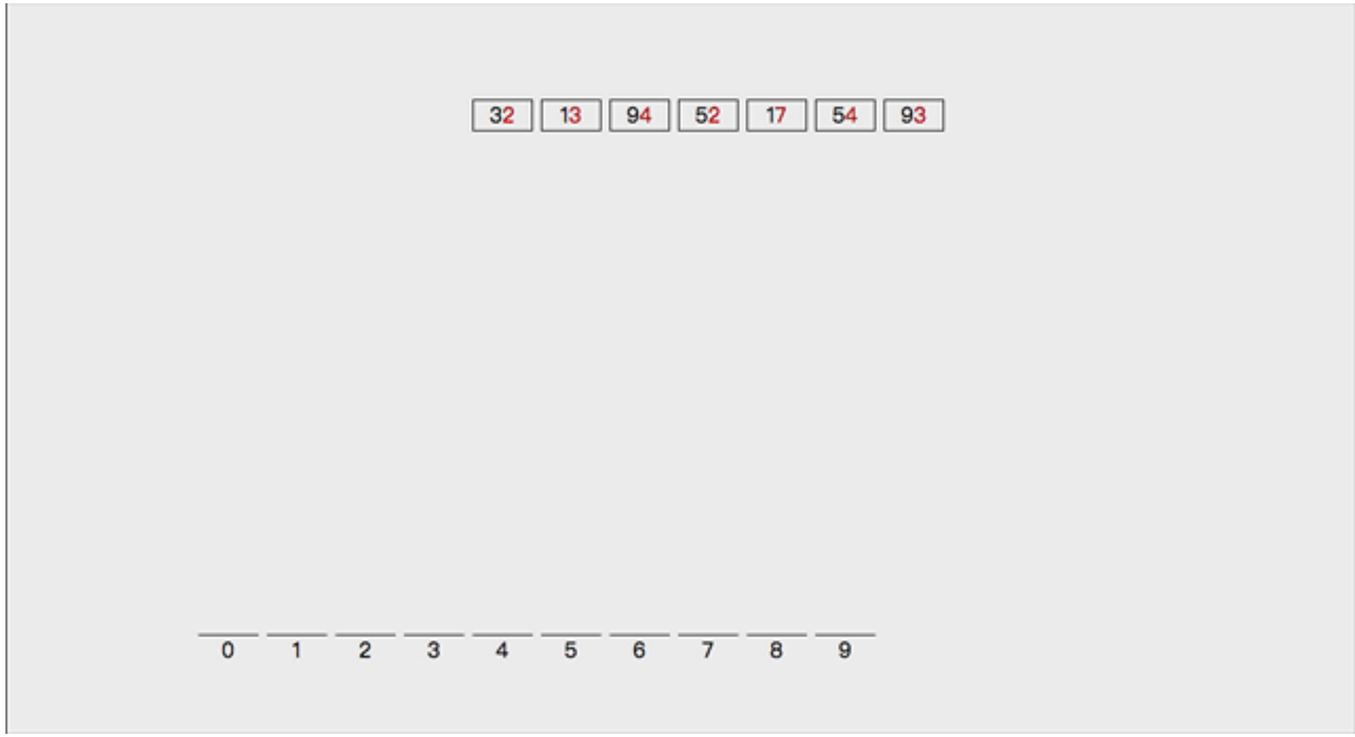


桶排序——讨论

- 桶排序的表现取决于数据的分布。也就是需要对不同数据排序时采取不同的分桶策略。
- 平均情况时间复杂度：$O(n+k)$
- 最坏情况时间复杂度：$O(n^2k)$
- 空间复杂度：$O(nk)$

基数排序

- 多关键字排序：加入现在有一个员工表，要求按照薪资排序，年龄相同的员工按照年龄排序。
  - 先按照年龄进行排序，再按照薪资进行稳定的排序。
- 对32,13,94,52,17,54,93排序，是否可以看做多关键字排序?



基数排序——代码

```
def list_to_buckets(li, base, iteration):
    buckets = [[] for _ in range(base)]
    for number in li:
        digit = (number // (base ** iteration)) % base
        buckets[digit].append(number)
    return buckets

def buckets_to_list(buckets):
    return [x for bucket in buckets for x in bucket]

def radix_sort(li, base=10):
    maxval = max(li)
    it = 0
    while base ** it <= maxval:
        li = buckets_to_list(list_to_buckets(li,  base, it))
        it += 1
    return li
```

基数排序——讨论

- 时间复杂度：O(kn)
- 空间复杂度：O(k+n)
- k表示数字位数

查找排序相关面试题

- 1. 给两个字符串s和t，判断t是否为s的重新排列后组成的单词
  - s = "anagram", t = "nagaram", return true.
  - s = "rat", t = "car", return false.

查找排序相关面试题

- 2. 给定一个m*n的二维列表，查找一个数是否存在。列表有下列特性：
  - 每一行的列表从左到右已经排序好。
  - 每一行第一个数比上一行最后一个数大。

```
[
  [1,   3,  5,  7],
  [10, 11, 16, 20],
  [23, 30, 34, 50]
]
```

查找排序相关面试题

- 3. 给定一个列表和一个整数，设计算法找到两个数的下标，使得两个数之和为给定的整数。保证肯定仅有一个结果。
  - 例如，列表[1,2,5,4]与目标整数3，1+2=3，结果为(0, 1).