第5章

排序算法

APP 开发是对计算机语言的应用，当然也需要涉及对算法的使用。虽然作为 iOS 开发，似乎接触更多的是对界面和数据的处理，但对于开发第三方库以及底层框架，会有很多场景需要采用适当的算法，例如，APP 的缓存策略，需要有一些算法来保证缓存的重用或销毁，诸如此类等。不管是什么开发语言，对算法的掌握是非常有必要的，不仅是在面试中经常会考算法，在实际应用中，算法的使用能更高效地处理数据。同时算法的思想也能更好地帮助我们理解计算机语言。

本章内容：

- 冒泡排序
- 选择排序
- 插入排序
- 快速排序
- 希尔排序
- 归并排序
- 堆排序
- 基数排序

排序算法是算法中一个比较常见的知识点，主要功能是对一个乱序数组进行排序，为了达到更少的计算量，对一些排序方法进行了命名，其中比较出名的 8 大排序算法是：冒泡排序，选择排序，希尔排序，快速排序，归并排序，堆排序，基数排序。

如果不涉及架构方面，以及多重计算优化的话，算法其实并不常用，当然在很多比较大的公司的面试题中，算法是一个比较重要的考察方面，并且考察的算法主要就是排序算法和二叉树。而在面试排序算法中，主要考察的排序并不是全部的 8 大算法，主要分为三个层次：①手写冒泡排序；②了解冒泡、选择、快速排序，并以伪代码形式写出；③除了前面两个层次，需要对剩下的排序算法有了解，并可以简单说出原理。这是面试题中关于考算法的三个等级，一般的公司不强求算法的话都只要求掌握第一个就行了，而大一些的公司，例如

BAT，需要掌握到第二个层次，而第三个层次基本上没有公司会考到，当然不排除有些主攻算法的面试官会问到，所以这里对所有的排序算法进行分析讲解，希望读者们在以后的面试算法中能够轻松应付。

由于 Objective-C 的数组和字典不能存储值类型、结构体，也就是包括 int、NSInteger、BOOL、CGRect 类型都不能直接存储，需要转换成 NSNumber 或者 NSValue 来存储。所以为了展示排序算法，我们采用 Swift 语言来实现，首先 Swift 相比于 Objective-C 来说不区分可变和不可变数组，另外，Swift 的数组和字典属于值类型，并且可以存储基本数据类型，甚至是 nil 也可以。

5.1 冒泡排序

这是 8 大排序算法中最简单的排序，这里也不做详细介绍，代码如下。

```
func BubbleSort(arr: inout [Int]) -> [Int] {
    for i in 0..<arr.count {
        for j in i+1..<arr.count {
            if arr[i] > arr[j] {
                let tmp = arr[i]
                arr[i] = arr[j]
                arr[j] = tmp
            }
        }
            print(arr)
    }
    return arr
}

var arr = [2, 1, 5, 9, 4, 0, 6, 3, 8, 7]
print(BubbleSort(arr: &arr))
```

这里需要直接对传递进来的数组进行修改，所以函数在参数上要设置添加 inout 标识符表示这个数组可以与在函数内保持同一份，因为数组是值类型。虽然是简单的冒泡排序，我们也来分析一下，首先第一层循环从头到尾，每次将数组的每一位，通过第二次循环依次与之后的每一位进行比较，如果比后面某一个数大的话就与其交换。我们根据代码来进一步分析，首先外传循环第一次，进入内循环，i 是 0，j 是 1 到 arr.count－1，第一次内循环将 arr[0]与 arr[1]比较，也就是 2 与 1 比较，因为 2 是比 1 大的，所以交换，但是注意，内循环第二次的时候，是 arr[0]与 arr[2]比较，这时的 arr[0]已经是 1 了，同理，从 arr[2]开始到最后，遇到数字 0 时会与 1 再交换一次，所以第一次内循环一遍结束后数组的情况是：

```
[0, 2, 5, 9, 4, 1, 6, 3, 8, 7]
```

然后外循环第二次，将 2 与后面的数字进行比较，最后发现 1 比 2 小，进行交换，并且之后没有比数字 1 更小的了，所以第二次循环结束后数组情况是：

```
[0, 1, 5, 9, 4, 2, 6, 3, 8, 7]
```

同理接下来的每次如下：

```
[0, 1, 2, 9, 5, 4, 6, 3, 8, 7]
[0, 1, 2, 3, 9, 5, 6, 4, 8, 7]
[0, 1, 2, 3, 4, 9, 6, 5, 8, 7]
[0, 1, 2, 3, 4, 5, 9, 6, 8, 7]
[0, 1, 2, 3, 4, 5, 6, 9, 8, 7]
[0, 1, 2, 3, 4, 5, 6, 7, 9, 8]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

综合起来可以发现，每次都是将数组中剩下数字的最小值找出来，类似于冒泡泡一样，最终得到排好序的有序数组。当然也可以改变代码的实现逻辑，每次循环将数组剩下数字的最大值找出来放在数组的后面，这也是可以的。

冒泡排序的时间复杂度是 $O(n^2)$。

5.2　选择排序

选择排序与冒泡排序有些类似，但比冒泡要更好一些，代码如下。

```
func selectSort(arr: inout [Int]) -> [Int] {
    for i in 0..<arr.count {
        var minIndex = i
        for j in i+1..<arr.count {
            if arr[minIndex] > arr[j] {
                minIndex = j
            }
        }
        if i != minIndex {
            let tmp = arr[i]
            arr[i] = arr[minIndex]
            arr[minIndex] = tmp
        }

        print(arr)
    }
    return arr
}

var arr = [2, 1, 5, 9, 4, 0, 6, 3, 8, 7]
print(selectSort(arr: &arr))
```

打印结果：

```
[0, 1, 5, 9, 4, 2, 6, 3, 8, 7]
```

```
[0, 1, 5, 9, 4, 2, 6, 3, 8, 7]
[0, 1, 2, 9, 4, 5, 6, 3, 8, 7]
[0, 1, 2, 3, 4, 5, 6, 9, 8, 7]
[0, 1, 2, 3, 4, 5, 6, 9, 8, 7]
[0, 1, 2, 3, 4, 5, 6, 9, 8, 7]
[0, 1, 2, 3, 4, 5, 6, 9, 8, 7]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

通过打印结果可以看出，选择排序也是在外层循环每次结束将数组剩下数字的最小值找出来，放在已排好序的末尾，结果是一样的，但是在实现逻辑上，选择排序会更好一些，因为在选择排序中，都是通过记录最小值的 index 来获取最小值的位置，最后才进行交换，少做了无用功。

但是时间复杂度仍然是 $O(n^2)$。

5.3 插入排序

插入排序是比较直观的排序算法，将数组从头至尾依次通过交换向前排至正确的位置。

```
func insertSort(arr: inout [Int]) -> [Int] {
    for i in 1..<arr.count {
        let temp = arr[i]
        var j = i
        while j > 0, temp < arr[j - 1] {
            arr[j] = arr[j-1]
            j -= 1
        }
        arr[j] = temp
        print(arr)
    }
    return arr
}

var arr = [2, 1, 5, 9, 4, 0, 6, 3, 8, 7]
print(insertSort(arr: &arr))
```

打印结果：

```
[1, 2, 5, 9, 4, 0, 6, 3, 8, 7]
[1, 2, 5, 9, 4, 0, 6, 3, 8, 7]
[1, 2, 5, 9, 4, 0, 6, 3, 8, 7]
[1, 2, 4, 5, 9, 0, 6, 3, 8, 7]
[0, 1, 2, 4, 5, 9, 6, 3, 8, 7]
[0, 1, 2, 4, 5, 6, 9, 3, 8, 7]
[0, 1, 2, 3, 4, 5, 6, 9, 8, 7]
```

```
[0, 1, 2, 3, 4, 5, 6, 8, 9, 7]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

之所以插入排序是一种比较直观的排序方法，是因为插入排序用语言来描述的话是非常简单的，在无序数组中，从第 1 位开始(前面有第 0 位)，与前面的已排好序的数中从后往前比较，直到插入在这个数所在顺序的位置中，以此类推。

在如上代码中，for 循环从 1 开始，利用 temp 存储 arr[1]的值，然后与 arr[1]之前的数进行比较，也就是 arr[0]，由于 arr[0]比 arr[1]大，则将 arr[0]的值赋给 arr[1]，由于 arr[1]就一位，所以 while 结束，此时将 temp 存储的数赋给 arr[0]，达到移动值的目的，此时数组已成为：

```
[1, 2, 5, 9, 4, 0, 6, 3, 8, 7]
```

第二轮循环开始，从 arr[2]开始，将 temp 赋值为 arr[2]，也就是 5，拿 5 与前面的依次比较，直到找到比 5 大的数，由于 5 比 1 和 2 都大，找到最左边都没有，所以将存储的 temp 仍然返回给 arr[j]，即 arr[2]。

直到数字 6 的时候，temp 为 6，此时数组已经为：

```
[0, 1, 2, 4, 5, 9, 6, 3, 8, 7]
```

将 0 与前一位比较，没有 9 大，则将 6 的位置赋值为 9，此时数组为：

```
[0, 1, 2, 4, 5, 9, 9, 3, 8, 7]
```

再与前面比较，比 5 大，则 break while 循环，同时将记录的 j 的位置替换为原来的 temp，所以此次循环结束的数组为：

```
[0, 1, 2, 4, 5, 6, 9, 3, 8, 7]
```

之后以此类推，完成整个排序算法。虽然上面用了很浅显的文字描述了插入排序的过程，但是插入排序确实是逻辑上很清晰的排序算法。举个例子，10 个人按身高排队，先找第一个人出来站好，再找第二个人来与第一个比较身高，高则站后面，矮则站前面，同理，后面再有人来排队的话，只要与站在后面最高的人依次向前比较，就能找到正确的位置。

插入排序相当于冒泡排序和选择排序来说，是一种值移动的方法，而冒泡和选择排序是产生中间变量用于交换，所以在数组个数不大的情况下插入排序是要优于冒泡和选择排序的。由于仍然是需要两轮循环，所以插入排序的时间复杂度仍然是 $O(n^2)$。

5.4 快速排序

快速排序又叫二分排序、二分插入排序、折半排序，相比于前几种排序算法，是一种真正体现出算法优越性的排序。快速排序有些是在插入排序的基础上，使用二分查找的方式，将一个 list 划分为两个 list 来执行，所以在时间复杂度上有很明显的优势。代码如下。

```
func partition(arr: inout [Int], left: Int, right: Int) -> Int {
    var left = left
    var right = right
    let pivot = arr[left]

    while left < right {
        while left < right, arr[right] >= pivot {
            right -= 1
        }
        arr[left] = arr[right]
        while left < right, arr[left] <= pivot {
            left += 1
        }
        arr[right] = arr[left]
    }
    arr[left] = pivot
    return left
}

func quickSort(arr: inout [Int], left: Int, right: Int) {
    guard left <= right else {
        return
    }

    let pivotIndex = partition(arr: &arr, left: left, right: right)
    quickSort(arr: &arr, left: left, right: pivotIndex - 1)
    quickSort(arr: &arr, left: pivotIndex + 1, right: right)
}


var arr = [2, 1, 5, 9, 4, 0, 6, 3, 8, 7]
quickSort(arr: &arr, left: 0, right: arr.count - 1)
print(arr)
```

我们将快速排序的算法分成两个函数，这是采用了二分查找与分治的思想，partition 函数采用二分查找，获取划分的界限，从而将问题分解为更小的问题来解决。事实上，我们可以将 partition 放在快速排序的函数体内，成为其私有的函数，从而体现完整性，不过这里暂且只讨论算法，所以不做考虑。

可以从代码中看出，partition 函数是将数组的某一区域进行一个简单的排序划分，获取到划分的位置，根据其位置再将数组划分为两个区域，然后分别递归调用快速排序函数。

接下来根据数组以及代码详细分析一下排序过程。

我们传递的数组是[2，1，5，9，4，0，6，3，8，7]，并且传递了数组的一个区域，表示在该区域里执行。当然在最外层调用快速排序的就是整个数组的区域，即 0～arr. count－1。进入 quickSort 函数，由于数组个数至少是大于 1 的(个数为 0 或者为 1 没有必要排序)，所以执行 partition 函数，partition 函数是快速排序的要点所在，接下来，我们调用 partition 函数传递的参数，仍然是该数组以及其整个范围。鉴于 Swift 的参数默认为 let 不可修改，所

以可以使用同 arr 参数类似的 inout 修饰，但这将使整个函数显得过于冗杂，因此在 partition 内创建同名的局部变量 left、right，并且记录区域的第一个值 pivot 为 arr[left]，我们称其为比较数，然后便开始 while 循环了。

外层循环是 left 一定要小于 right 的，因为内部查找是从数组两边往中间合拢的，当 left 大于或等于 right 则表示此次遍历交汇了，即查找完毕。详细看一下内部代码，内部第一个 while 循环表示，从数组右侧开始，在仍然满足 left 小于 right 的情况下，一直找到比比较数 pivot 小的数，数组一开始如下：

```
[2, 1, 5, 9, 4, 0, 6, 3, 8, 7]
```

一开始 left 为 0，right 为 9，pivot 为 arr[0]，即 2。接着刚才的分析，从数组 right 位开始向前(右)找，一直找到比 2 小的数，即 arr[5]=0。然后跳出内部第一个循环，此时 left 为 0，right 为 5，执行语句：

```
arr[left] = arr[right]
```

此时数组为：

```
[0, 1, 5, 9, 4, 0, 6, 3, 8, 7]
```

可以看到第 0 位已经被 arr[5]覆盖了，接着执行内部下一个循环。相反地，仍然在 left 小于 right 的情况下，从 left 开始向后(左)一直找出比 pivot 大的值，即比 2 大的值，结果在第 2 位找到了，因为第 2 位是 5，因此跳出循环。此时，left 为 2，right 为 5，接着执行语句：

```
arr[right] = arr[left]
```

此时数组为：

```
[0, 1, 5, 9, 4, 5, 6, 3, 8, 7]
```

将第 5 位的 0 赋值为第 2 位的数 5，此时外部 while 循环执行一遍结束，仍然满足 left 小于 right 条件，所以继续 while，此时 left 为 2，right 为 5，pivot 值为 2。同理，从 right 开始向后查找比 pivot 小的数，right 从 5 一直到 2，也没有找到比 pivot 小的数，而此时 left 已经等于 right 了，所以最外层的 while 循环结束了。结束后，left 为 2，right 为 2，执行语句：

```
arr[left] = pivot
[0, 1, 2, 9, 4, 5, 6, 3, 8, 7]
```

最后，将 left 的值 2 返回出去，至此 partition 函数执行结束。这时可以看到底 partition 函数做了什么？在函数一开始的地方，我们设置数组第一位 2 为我们的 pivot 比较值，最后得到的结果是将 2 排到了正确的位置，并且比 2 小的数都在其左边，比 2 大的数都在其右边，但是左右也并非是有序的。

我们将 2 返回给了 quickSort 函数内部的局部变量 pivotIndex，这表示已经将该位置确

定好，接下来只要同理分别执行两边的数组就能达到目的。

quickSort(arr：&arr，left：left，right：pivotIndex－1)调用表示将 arr[2]左边的区域再次进行划分，quickSort(arr：&arr，left：pivotIndex＋1，right：right)调用表示将 arr[2]右侧的区域再次进行划分，这样在每个划分的区域内确定其比较值的位置，当划分到最小单位时，整个数组就排好序了。其时间复杂度为 O(nlogn)。

5.5 希尔排序

前面介绍了 4 种排序：冒泡排序，选择排序，插入排序和快速排序。这是 4 种最常见的排序方法，而后面介绍的 4 种排序或许读者只是听过，却很少接触过，对于一般的排序事务，前面 4 种排序已然足够应付，况且快速排序是 8 大排序中效率最高的，自然而然后面这 4 种排序就很少有人问津了，但作为了解以及算法思路还是值得学习的。

希尔排序又称为缩小增量排序，是插入排序的进化版，但与插入排序不同的是，希尔排序将子序列按照某个增量值进行排序，随着增量值的变小，整个数组形成了一个大致排好序的序列，最后按照增量 1 一个数一个数地直接进行插入排序，整个过程避免了很多重复的交换值操作。但希尔排序是一种非稳定的排序算法，所谓非稳定性，指的是对于数组中两个相同的元素，可能在排序之后顺序发生变化，即使两个数大小相同。希尔排序中两个相同的数可能会因为增量划分到不同的组中，导致最后排序好之后两个数位置发生了变化，因此不稳定。

代码如下。

```
func shellSort(arr: inout [Int]) {
    var gap = arr.count / 2
    while gap >= 1 {
        var i = gap
        while i < arr.count {
            let temp = arr[i]
            var j = i - gap
            while j >= 0, temp < arr[j] {
                arr[j + gap] = arr[j]
                j -= gap
            }
            if j != (i - gap) {
                arr[j + gap] = temp
            }
            i += 1
        }
        gap = gap / 2
    }
}
var arr = [2, 1, 5, 9, 4, 0, 6, 3, 8, 7]
shellSort(arr: &arr)
print(arr)
```

希尔排序的算法并不复杂，可以看出有插入排序的影子。首先定义最初的增量值，为数组个数的一半，在例子中，gap 值即为 5。在 gap 始终大于等于 1 的情况下，先将 gap 值赋给 i，即 i 为 5，j 为 0，记录 arr[5]的值给 temp，即 temp = 0，然后便开始增量排序了，首先将 temp 与 arr[0]比较，即 arr[5]与 arr[0]比较，如果 arr[0]比 temp 大，则应将 arr[5]赋值为 temp，此时数组由：

```
[2, 1, 5, 9, 4, 0, 6, 3, 8, 7]
```

变成了

```
[2, 1, 5, 9, 4, 2, 6, 3, 8, 7]
```

j -= gap，表示将 temp 与该组中下一位进行比较，此时 j 已经为-5，跳出循环了，同理，最后判断 j 是否不等于(i - gap)，如果不等于，则表示 j 已经自减过，也就是表示更换过数，所以将 temp 的值赋给最后 j+gap 的那一位，在此处，j+gap 是 0，即将 temp 存储的值赋给 arr[0]，从而达到交换的目的。同理，arr[6]与 arr[1]比较，arr[7]与 arr[2]比较，一直到arr[9]与 arr[4]比较，这时增量为 5 的循环结束，自除以 2，增量为 2，再次按照如上逻辑进行插入排序，这次会形成较为有序的数组。直到增量为 1，真正的插入排序，此时只要做比较少的操作就可以了。

为了方便读者理解，再举个例子，例如一群身高不一的学生站成一排，然后 0～4 报数，报数完毕后，先让报 0 的同学站出来，按照插入排序的方法进行排序，这时，报 0 的同学排好身高位置后归队，同理报 1 的同学站出来排列，一直到一轮结束后，再按照 0、1 报数，同理，再次细分、排序，一直到最后，所有同学再进行逐个插入排序，由于上一次已经排好一部分，所以这一次并不会做太多操作就能实现整个希尔排序。

从宏观角度上看，希尔排序每次先一轮排序都为下一轮排序提供了有利条件，从而避免了重复的无用交换，从而达到对插入排序的优化。希尔排序是按照不同步长对元素进行插入排序，当刚开始元素很无序的时候，步长最大，所以插入排序的元素个数很少，速度很快；当元素基本有序了，步长很小，插入排序对于有序的序列效率很高。所以，希尔排序的时间复杂度会比 $O(n^2)$好一些。

5.6　归并排序

归并排序又称为二路归并，是采用分治法最典型的例子，其基本思想是将整个数组的排序问题递归划分为子序列，最后层层往上合并为一个有序序列，直至最终实现全部排序完成。

代码如下。

```
func mergeSort(arr: inout [Int], left: Int, right: Int) {
    guard left < right else {
        return
```

```
        }
        let mid = (left + right) / 2

        mergeSort(arr: &arr, left: left, right: mid)
        mergeSort(arr: &arr, left: mid + 1, right: right)

        var temp: [Int] = Array(repeatElement(0, count: right - left + 1))
        var i = left
        var j = mid + 1
        var k = 0

        while i <= mid, j <= right {
            if arr[i] <= arr[j] {
                temp[k] = arr[i]
                k += 1
                i += 1
            } else {
                temp[k] = arr[j]
                k += 1
                j += 1
            }
        }
        while i <= mid {
            temp[k] = arr[i]
            k += 1
            i += 1
        }
        while j <= right {
            temp[k] = arr[j]
            k += 1
            j += 1
        }
        for p in 0..<temp.count {
            arr[left + p] = temp[p]
        }
    }
```

通过代码的前 7 行可以看到归并排序的大概分治逻辑，即将数组按照折半来逐级划分，直至划分到 left＝right，也就是划分的子序列仅有一个元素，也就是划分到了最小的单位，随后，每一层级执行下面的代码，将当前区域 left 至 right 的数组变成有序的。首先创建一个含有 right－left＋1 个元素的数组，数组每个元素都暂且为 0，作为存储用。

由于该区域是下一级返回过来的，所以在 left 至 right 区域中，left 到 mid 是有序的，mid＋1 至 right 也是有序的，现在就是要将 left 至 right 再排序，所以做法是将两个区域从最小的值开始比较，然后将较小值存入 temp 中，再与较小值所在区域的下一位进行比较，直至某一区域的元素检索完，将另一元素的剩下元素全部赋值给 temp 剩下的位置，这样 temp 就得到了一个在 left 至 right 的有序数组。这是实现了该区域的排序，当然这不是终点，而只是众多分治中的某一步，接下来就返回上一级的分治之中，将该区域的有序数组与

邻近区域的有序数组再次合并为另一个更大的有序数组，如此到最后实现排序的全部完成。

由于采用的是较为容易理解的分治递归思想，使归并排序看起来并不复杂，并且时间复杂度为 $O(n\log n)$，是一种效率较高且稳定的算法。

5.7　堆排序

堆排序是这几个排序中逻辑稍复杂的算法，之所以显得有些复杂，是因为堆排序涉及一个新概念，就是堆，堆的数据结构可以看作一个完全二叉树。二叉树是每个节点最多只有两个子树的树结构，类似于图 5-1 的树状结构就称为二叉树。

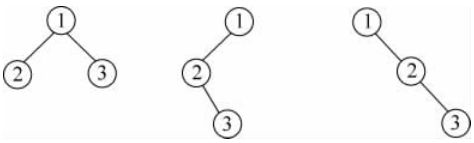

图 5-1　二叉树

而完全二叉树是深度为 h，有 n 个节点的二叉树，当且仅当其每个节点的编号都与深度为 h 的满二叉树中从 1 至 n 的节点编号一一对应。换句话说，我们生成一个完全二叉树，是从左至右依次添加到树结构上，如图 5-2 所示的二叉树即为完全二叉树。

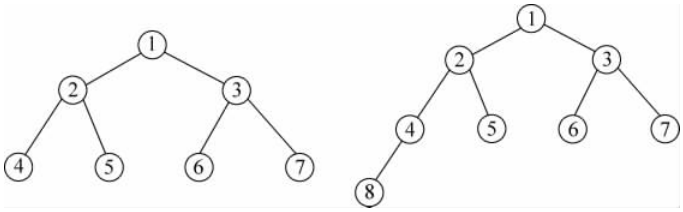

图 5-2　完全二叉树

本节中堆的结构就大致是完全二叉树的结构，而在下面对堆排序的实现中，也是通过这种结构来实现的。而二叉树的结构与数组的对应关系也是比较简单的，即数组中每个数所在的位置对应于完全二叉树的节点顺序。例如，我们排序中用到的数组：

```
var arr = [2, 1, 5, 9, 4, 0, 6, 3, 8, 7]
```

用完全二叉树表现如图 5-3 所示。

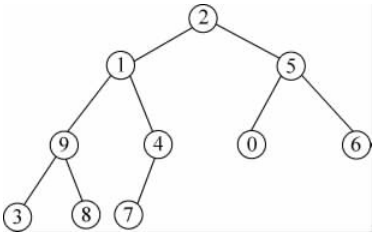

图 5-3　数组的完全二叉树表示

而这样的完全二叉树结构还不能称为堆，因为堆还有两个条件：

（1）堆的最大元素或者最小元素出现在堆顶；

（2）堆的父节点的值都是大于或者小于其子节点的值。

其实，如果满足第二个条件，第一个条件也自然成立，根据以上对堆的定义，以及堆的元素顺序，可以分为

最大堆和最小堆，即父节点大于子节点的称为最大堆，反之称为最小堆。

在之前的排序中，都是将数组排成从小到大的顺序，此例中，采用最大堆或者最小堆都可以实现，下面看一下代码。

```
func maxHeapify(arr: inout [Int], index: Int, size: Int) {
    var i = index
    while true {
        var iMax = i
        let iLeft = 2 * i + 1
        let iRight = 2 * i + 2

        if iLeft < size, arr[i] < arr[iLeft] {
            iMax = iLeft
        }
        if iRight < size, arr[iMax] < arr[iRight] {
            iMax = iRight
        }
        if iMax != i {
            let temp = arr[iMax]
            arr[iMax] = arr[i]
            arr[i] = temp
            i = iMax
        } else {
            break
        }
    }
}

func buildMaxHeap(arr: inout [Int]) {
    let lastParent = arr.count / 2 - 1
    for i in (0...lastParent).reversed() {
        maxHeapify(arr: &arr, index: i, size: arr.count)
    }
}

func heapSort(arr: inout [Int]) {
    buildMaxHeap(arr: &arr)
    for i in (1...arr.count - 1).reversed() {
        let temp = arr[0]
        arr[0] = arr[i]
        arr[i] = temp
        maxHeapify(arr: &arr, index: 0, size: i)
    }
}

var arr = [2, 1, 5, 9, 4, 0, 6, 3, 8, 7]
heapSort(arr: &arr)
```

我们将堆排序分成了三部分代码，heapSort 是主方法，buildMaxHeap 是创建最大堆（本例中采用最大堆来实现从低到高的排序，这并不冲突），maxHeapify 是对节点 index 进行调整。

在解释这段代码之前，先图解一番其过程。数组[2，1，5，9，4，0，6，3，8，7]用二叉树来表现如图 5-4 所示。

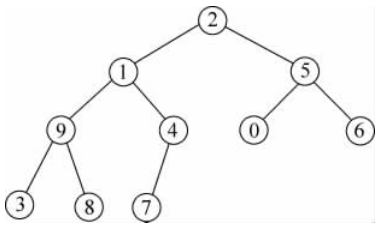

图 5-4　数组建堆前

接下来，将对这个二叉树进行构建成堆，首先从最后一个父节点开始。首先确定一下对于父节点和子节点的公式。

对于节点 i 来说：

i 的父节点为：$(i-1)/2$

i 的左子节点为：$2\times i+1$

i 的右子节点为：$2\times i+2$

并且对于节点 i 来说，是必有父节点的，但其左子节点和右子节点不一定有，因为节点 i 可能为叶子节点。

接下来的任务是构建成堆，堆的定义在之前讲过，是在完全二叉树的基础上对于任一节点的值都大于其左右子节点，所以我们从这个二叉树的最后一个父节点开始查找并替换。而最后一个父节点即是数组的最后一个元素的父节点，为(arr.count－1) / 2，即第 4 位，正好也为 4。

判断第 4 位与其左右子节点的大小，由于只有左子节点，所以将其和左子节点比较，又因为 arr[4] ＜ arr[9]，子节点大于父节点，所以替换，如图 5-5 所示。

由于 arr[4]的子节点是叶子节点，所以不能继续向下查找判断，此轮循环结束，寻找下一个父节点 arr[3]，arr[3]的值为 9，在图 5-5 所示二叉树中有两个子节点，但值 9 是比值 3 和值 8 大的，也就是 arr[3]比其两个子节点的值都大，所以不用交换，并且也不用向下执行，此循环结束。下一个循环，arr[2]值为 5，其右节点的值 6 大于值 5，所以交换，如图 5-6 所示。

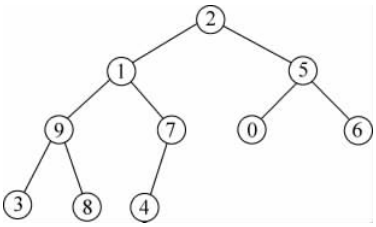

图 5-5　建堆过程 1

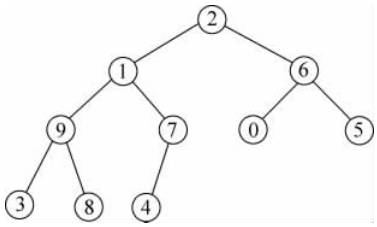

图 5-6　建堆过程 2

接下来到父节点 arr[1]，其左节点比 arr[1]大，交换，如图 5-7 所示。

然而到这里，由于其左节点 arr[3]为父节点，同时 arr[3]的右子节点比当前的 arr[3]的值大，所以继续交换，如图 5-8 所示。

此时可能有读者会有疑问，如果 arr[3]的某个子节点会有比值 9 还大的存在，该如何与 arr[1]进行交换呢？实际上，我们之所以从最后一个父节点开始往上遍历，为的就是如果在

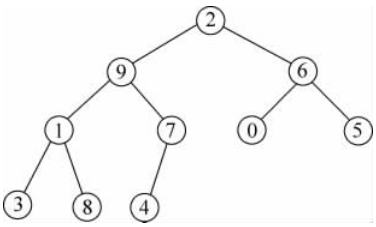

图 5-7　建堆过程 3

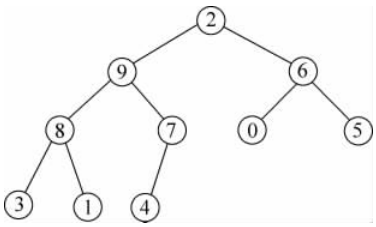

图 5-8　建堆过程 4

某一节点，例如节点 a 与其子节点 b 比较大小的时候，子节点 b 也为父节点，能够保证 a 的这一子节点 b 已经是比 b 的子节点都大。简单总结一下就是，如果某一节点小于其某一子节点，那么交换后，该节点肯定大于其任意子节点，以及孙子节点。结合上面过程可以理解为，值 9 一开始是值 3 和值 8 的父节点，即使值 9 被交换上去了，仍然是大于值 3 和值 8 的。

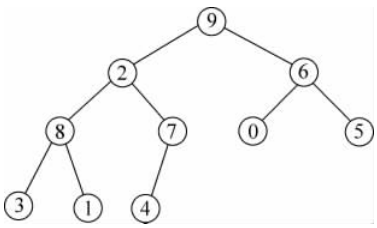

图 5-9　建堆过程 5

继续到节点 arr[0]，已是堆顶，此时 arr[0]值 2，比其左子节点值 9 小，交换，得到图 5-9。

但是此时 arr[1]小于左子节点 arr[3]，交换后仍然小于 arr[7]，再交换。这一过程如图 5-10 所示。

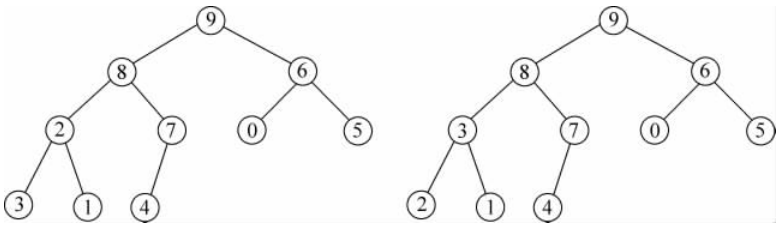

图 5-10　建堆过程 6

至此，该完全二叉树已经满足堆的条件，所以可以称此时的完全二叉树为堆。

同时，以上图解的这一过程，我们称之为最大堆调整，目的就是将一个不满足堆的完全二叉树，通过交换，最后成为一个堆结构。再对比之前的堆排序代码，可以看到如上过程就是函数 buildMaxHeap 的过程，其中对于某一节点进行查找子节点并交换是由函数 maxHeapify 完成的。至此已经完成了堆排序最复杂的一步了，接下来就很好理解了。

通过如上的操作，现在将最大值 9 置换到了堆顶，堆顶也是数组中的首元素 arr[0]，接下来需要将值 9 再交换至函数最后，如图 5-11 所示。

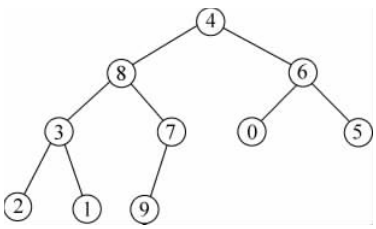

图 5-11　建堆过程 7

有读者可能会有疑问，为何要将最大值交换至末尾？这是因为，我们刚刚通过堆排序，找到了数组中的最大值，由于是从小到大的顺序，所以每次将堆排序结构中的最大值找出来依次放在末尾，就可以实现完整的排序了。

此时值 9 已经被交换至末尾，我们在函数 heapSort 中缩小 size，表示接下来的构建最大堆以及寻找最大值，

都是除了 arr[9]，在其之前操作。同时由于值 4 被交换至堆顶，此时以及破坏了堆的结构，所以要重新对堆顶进行检查，注意此时，因为已经不管值 9 了，并且在现在的二叉树的操作范围内，我们只改动了堆顶节点，对于其他父节点，仍然保持大于子节点的特性，所以此时只要对堆顶节点进行操作即可，操作方法即为函数 maxHeapify。以此类推，我们每次构建一次最大堆，都能找出剩下元素中的最大值，并将其排列在数组第 n 大的位置上，最终实现完整的堆排序。

5.8　基数排序

基数排序，又称为“桶子法”排序，属于一种分配式排序，排序思想也不同于之前介绍的几种排序，而是一种借助于多关键字排序思想进行排序的方法，其多关键字排序就是根据其不同的优先级来划分关键字。其中，在数字排序中，一般将排序数的个十百千位划分为其优先级的关键字，逐级根据其关键字依次从个位向高位划分，每次划分结束，将划分好的数据收集起来，再进行下一次的高位划分，直至最后整个排序完成。

为了达到更好的演示效果，我们使用复杂一点儿的数据来进行图示。首先用于图示的数组如下：

```
var arr = [132, 785, 064, 527, 308, 457, 002, 921, 233, 477]
```

我们将整个待排序数组分为 10 个桶，因为每一位数字为 0～9，刚好 10 个，所以可以将数组按照某一逻辑划分到相应的桶中，这个逻辑就是从个位开始，将 0～9 作为关键字，按照个位数值来划分依次入列。

如图 5-12 所示，首先按照个位进行了一个简单的划分，数组的值都进入了相应的桶中，然后将这些数再收集起来，按照十位来再次划分，如图 5-13 所示。

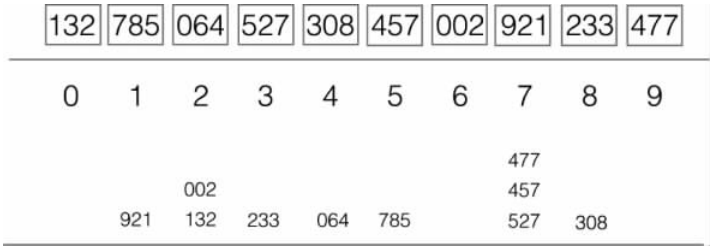

图 5-12　数组按个位划分

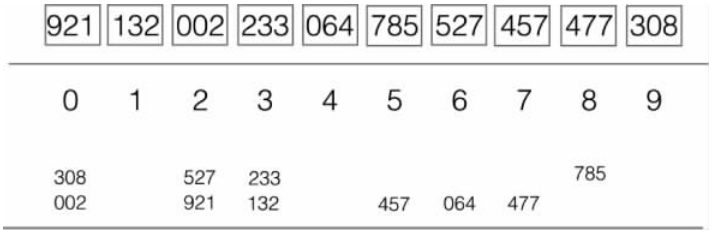

图 5-13　数组再按十位划分

同理，将按照十位划分过后的数据收集起来，再次按照百位来进行划分，因为最大数的位数就是百位。

如图 5-14 所示，因为已经循环到了最高位——百位，所以循环结束，此时再次将桶中的数据收集起来，你或许会惊奇地发现数组已经排好序了，也就是说基数排序已经完成了。

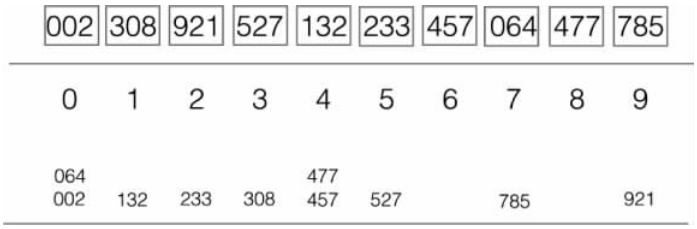

图 5-14　数组再按百位划分

下面先来看一下代码。

```
func radixSort(list: inout Array<Int>) {
    var bucket = createBucket()
    let maxNumber = listMaxItem(list: list)
    let maxLength = numberLength(number: maxNumber)

    for digit in 1...maxLength {
        for item in list {
            let baseNumber = fetchBaseNumber(number: item, digit: digit)
            bucket[baseNumber].append(item)
        }
        print("第\(digit)轮入桶结果")
        print("\(bucket)")

        //出桶
        var index = 0
        for i in 0..<bucket.count {
            while !bucket[i].isEmpty {
                list[index] = bucket[i].remove(at: 0)
                index += 1
            }
        }
        print("第\(digit)轮出桶结果")
        print("\(list)\n")
    }
}

func createBucket() -> Array<Array<Int>> {
    var bucket: Array<Array<Int>> = []
    for _ in 0..<10 {
        bucket.append([])
    }
    return bucket
}
```

```
func listMaxItem(list: Array<Int>) -> Int {
    var maxNumber = list[0]
    for item in list {
        if maxNumber < item {
            maxNumber = item
        }
    }
    return maxNumber
}

func numberLength(number: Int) -> Int {
    return "\(number)".characters.count
}

func fetchBaseNumber(number: Int, digit: Int) -> Int{
    if digit > 0 && digit <= numberLength(number: number) {
        var numbersArray: Array<Int> = []
        for char in "\(number)".characters {
            numbersArray.append(Int("\(char)")!)
        }
        return numbersArray[numbersArray.count - digit]
    }
    return 0
}
```

可以看到，首先创建了 10 个空数组，表示 10 个桶结构，这个是后面所要用到的，接着找出数组中的最大数以及该最大数的位数来确定后面的循环次数。

然后开始循环，digit 从 1 到最大位数表示从个位开始到最高位。在每次的位循环中，按照我们根据当前是循环哪一位，来获取数组中每个数的该位的数字，并将这个数放到创建好的桶中，例如第一个数 132，第一次个位循环时，判断个位是 2，所以将 132 放入第二个桶中的数组中，以此类推。分配完毕后，桶中 10 个数组中的每个数组，有的可能含有值，有的可能为空数组。所以遍历循环桶结构，只要桶中某个数组还有值，就将其复写到原数组中，并将其 remove。这样个位的一遍循环结束后，数组进行了一次分配收集，桶也实现了填充数据到提出数据，最后又变为含有 10 个空数组。同理，再进行十位以及百位的循环。

基数排序的时间复杂度为 $O(d(n+\text{radix}))$，其中，一趟分配时间复杂度为 $O(n)$，一趟收集时间复杂度为 $O(\text{radix})$，共进行 d 趟分配和收集，是属于一种稳定的排序算法。

小结

本章介绍了著名的 8 大排序。其中，冒泡排序、选择排序、插入排序是简单的排序，快速排序、希尔排序、堆排序是比较高效率的排序，还有两种是体现分治思想的归并排序和基数排序。

每种排序都有各自的特点，有的简单有效，有的利用空间换时间来达到高效排序，其实最终的思想是大致相同的。读者可以根据实际需求来选择使用，所以说，没有最好的排序，

只有最适合的排序。

在实际的 iOS 开发中，或许很少会用到排序算法，所以大多数开发者对此也并没有深入了解，其实在涉及底层以及数据结构和算法时，就会使用得到，例如一些图片压缩算法。并且，以上介绍的几种算法也有比较优秀的设计思路，在实际开发中也用得到，例如分治递归。所以笔者将排序算法单独拿出来讲，可以作为了解，同时也希望读者可以在这些算法的精华中获取到更有用的编程思想以及理论知识，从而为以后打下良好的理论基础。