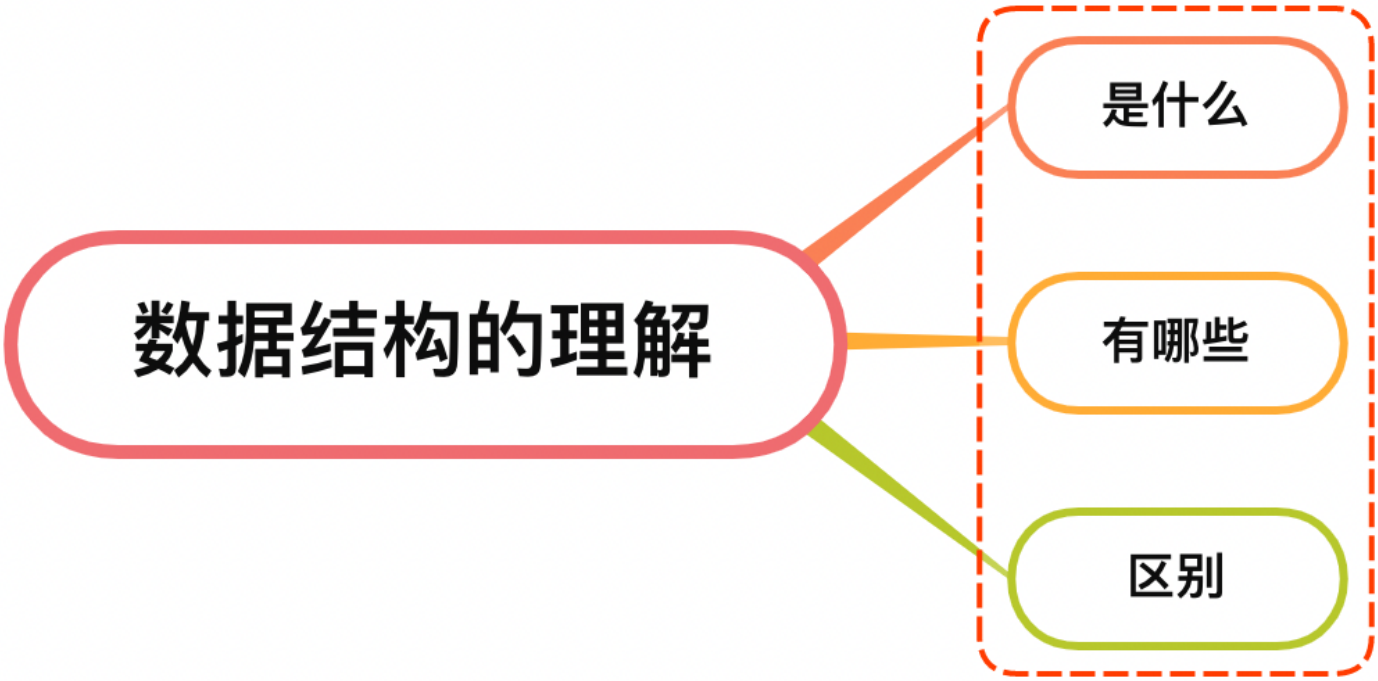
算法面试真题（18题）

1. 说说你对数据结构的理解？有哪些？区别？



1.1. 是什么

数据结构是计算机存储、组织数据的方式，是指相互之间存在一种或多种特定关系的数据元素的集合

前面讲到，一个程序 = 算法 + 数据结构，数据结构是实现算法的基础，选择合适的数据结构可以带来更高的运行或者存储效率

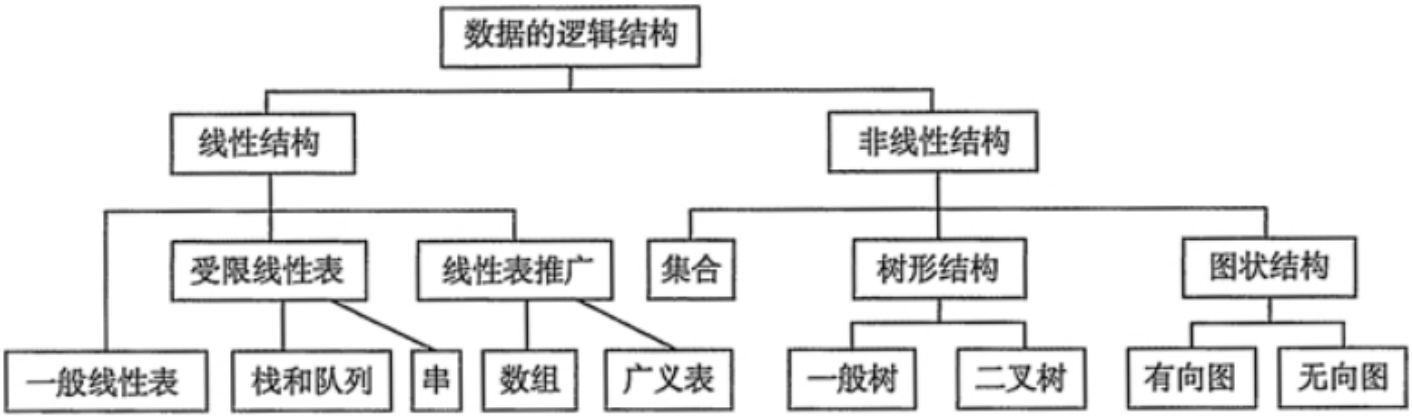
数据元素相互之间的关系称为结构，根据数据元素之间关系的不同特性，通常有如下四类基本的结构：

- 集合结构：该结构的数据元素间的关系是“属于同一个集合”
- 线性结构：该结构的数据元素之间存在着一对一的关系
- 树型结构：该结构的数据元素之间存在着一对多的关系
- 图形结构：该结构的数据元素之间存在着多对多的关系，也称网状结构

由于数据结构种类太多，逻辑结构可以再分成为：

- 线性结构：有序数据元素的集合，其中数据元素之间的关系是一对一的关系，除了第一个和最后一个数据元素之外，其它数据元素都是首尾相接的

- 非线性结构：各个数据元素不再保持在一个线性序列中，每个数据元素可能与零个或者多个其他数据元素发生关联



1.2. 有哪些

常见的数据结构有如下：

- 数组
- 栈
- 队列
- 链表
- 树
- 图
- 堆
- 散列表

1.2.1. 数组

在程序设计中，为了处理方便， 一般情况把具有相同类型的若干变量按有序的形式组织起来，这些按序排列的同类数据元素的集合称为数组

1.2.2. 栈

一种特殊的线性表，只能在某一端插入和删除的特殊线性表，按照先进后出的特性存储数据

先进入的数据被压入栈底，最后的数据在栈顶，需要读数据的时候从栈顶开始弹出数据

1.2.3. 队列

跟栈基本一致，也是一种特殊的线性表，其特性是先进先出，只允许在表的前端进行删除操作，而在表的后端进行插入操作

1.2.4. 链表

是一种物理存储单元上非连续、非顺序的存储结构，数据元素的逻辑顺序是通过链表中的指针链接次序实现的

链表由一系列结点（链表中每一个元素称为结点）组成，结点可以在运行时动态生成

一般情况，每个结点包括两个部分：一个是存储数据元素的数据域，另一个是存储下一个结点地址的指针域

1.2.5. 树

树是典型的非线性结构，在树的结构中，有且仅有一个根结点，该结点没有前驱结点。在树结构中的其他结点都有且仅有一个前驱结点，而且可以有两个以上的后继结点

1.2.6. 图

一种非线性结构。在图结结构中，数据结点一般称为顶点，而边是顶点的有序偶对。如果两个顶点之间存在一条边，那么就表示这两个顶点具有相邻关系

1.2.7. 堆

堆是一种特殊的树形数据结构，每个结点都有一个值，特点是根结点的值最小（或最大），且根结点的两个子树也是一个堆

1.2.8. 散列表

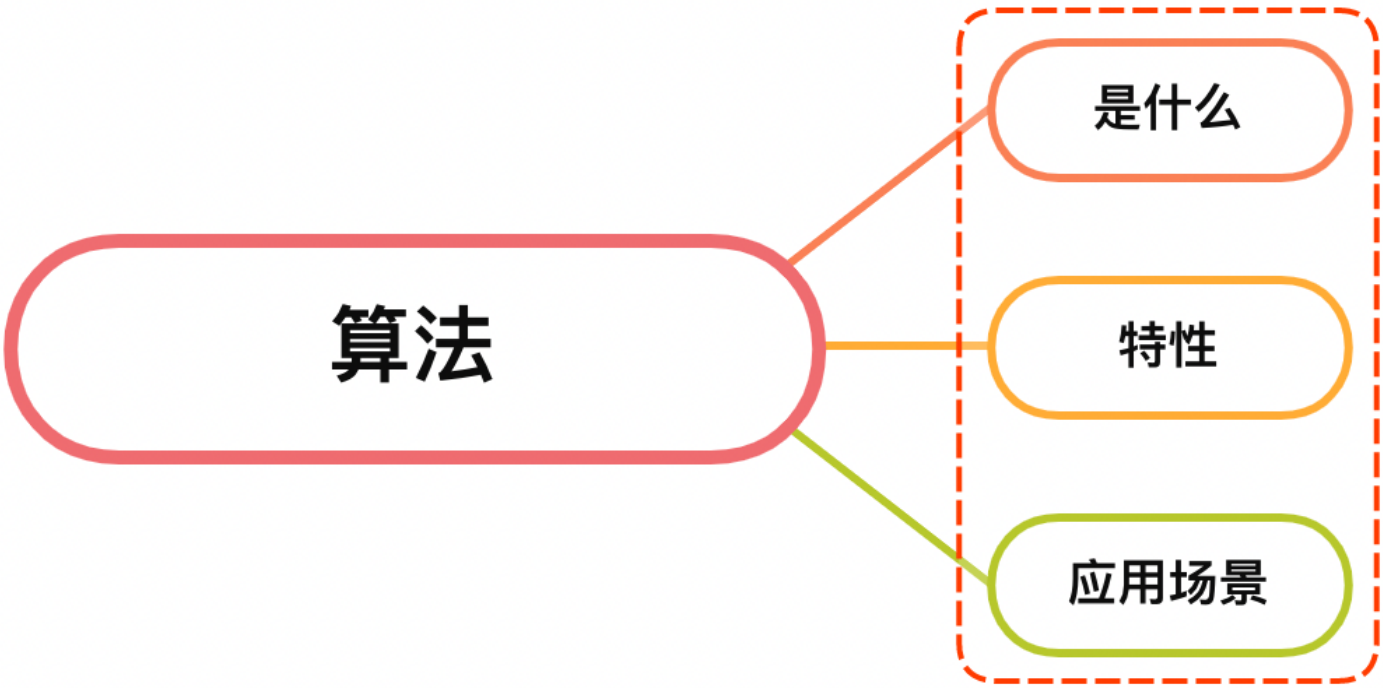
若结构中存在关键字和 `K` 相等的记录，则必定在 `f(K)` 的存储位置上，不需比较便可直接取得所查记录

1.3. 区别

上述的数据结构，之前的区别可以分成线性结构和非线性结构：

- 线性结构有：数组、栈、队列、链表等
- 非线性结构有：树、图、堆等

2. 说说你对算法的理解？应用场景？



2.1. 是什么

算法（Algorithm）是指解题方案的准确而完整的描述，是一系列解决问题的清晰指令，算法代表着用系统的方法描述解决问题的策略机制

也就是说，能够对一定规范的输入，在有限时间内获得所要求的输出

如果一个算法有缺陷，或不适合于某个问题，执行这个算法将不会解决这个问题

一个程序=算法+数据结构，数据结构是算法实现的基础，算法总是要依赖于某种数据结构来实现的，两者不可分割

因此，算法的设计和选择要同时结合数据结构，简单地说数据结构的设计就是选择存储方式，如确定问题中的信息是用数组存储还是用普通的变量存储或其他更加复杂的数据结构

针对上述，可以得出一个总结：不同的算法可能用不同的时间、空间或效率来完成同样的任务

2.2. 特性

关于算法的五大特性，有如下：

- 有限性（Finiteness）：一个算法必须保证执行有限步之后结束
- 确切性（Definiteness）：一个算法的每一步骤必须有确切的定义
- 输入（Input）：一个算法有零个或多个输入，以刻画运算对象的初始情况，所谓零个输入是指算法本身给定了初始条件
- 输出（Output）：一个算法有一个或多个输出。没有输出的算法毫无意义
- 可行性（Effectiveness）：算法中执行的任何计算步骤都是可以被分解为基本的可执行的操作步骤，即每个计算步骤都可以在有限时间内完成（也称之为有效性）

2.3. 应用场景

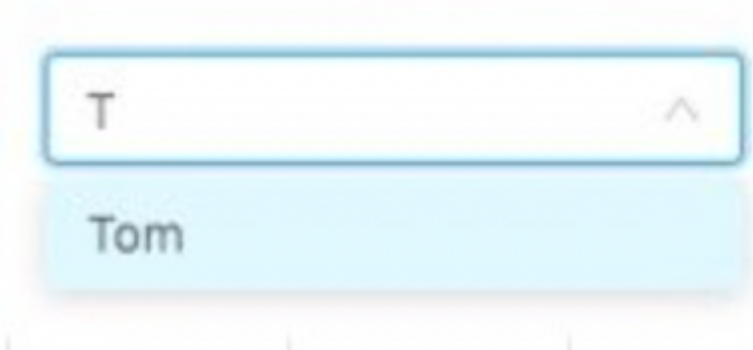
在前端领域中，数据结构与算法无法不在，例如现在的 `vue` 或者 `react` 项目，实现虚拟 `DOM` 或者 `Fiber` 结构，本质就是一种数据结构，如下一个简单的虚拟 `DOM`：

```
{
  type: 'div',
    props: {
      name: 'lucifer'
    },
      children: [{
        type: 'span',
        props: {},
        children: []
      }]
}
```

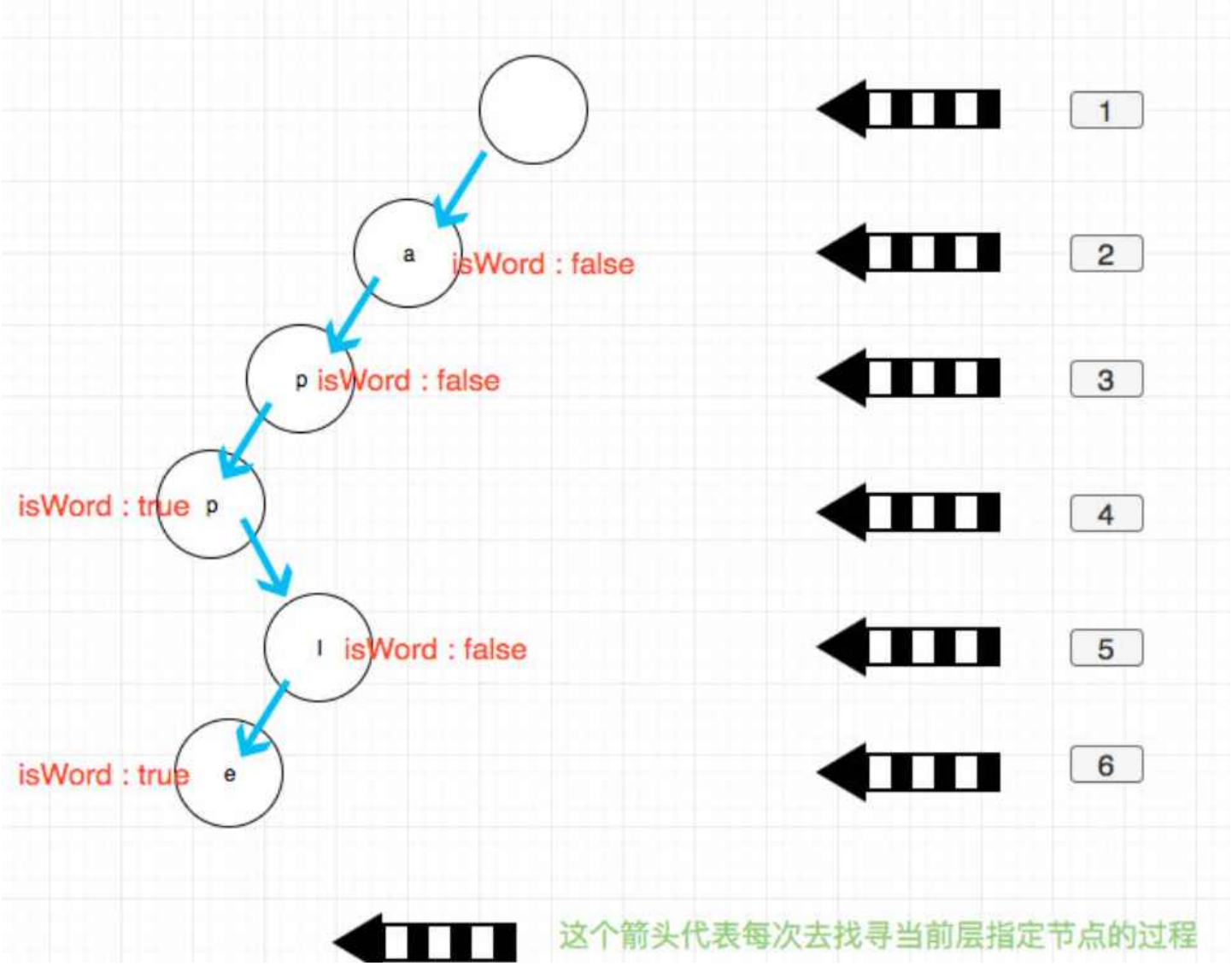
`vue` 与 `react` 都能基于基于对应的数据结构实现 `diff` 算法，提高了整个框架的性能以及拓展性

包括在前端 `javascript` 编译的时候，都会生成对应的抽象语法树 `AST` ，其本身不涉及到任何语法，因此你只要编写相应的转义规则，就可以将任何语法转义到任何语法，也是 `babel` ，`PostCSS` , `prettier` ， `typescript`

除了这些框架或者工具底层用到算法与数据结构之外，日常业务也无处不在，例如实现一个输入框携带联想功能，如下：



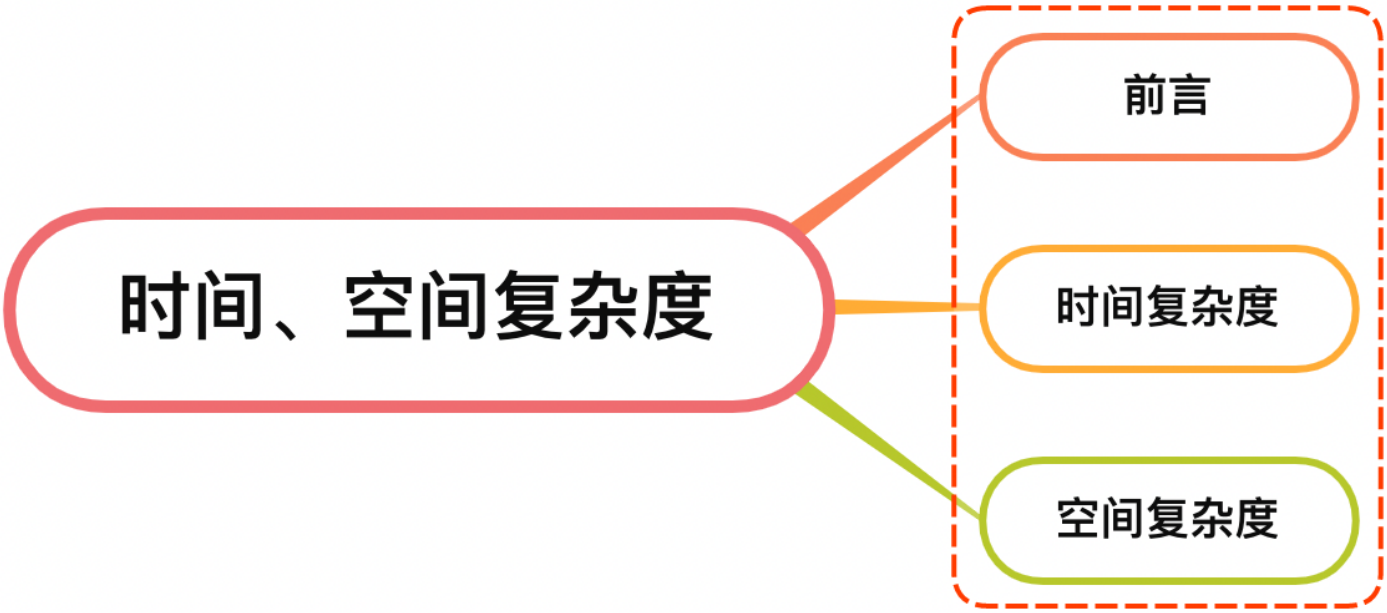
如果我们要实现这个功能， 则可以使用前缀树，如下：



包括前端可能会做一些对字符串进行相似度检测，例如"每日一题"和"js每日一题"两个字符串进行相似度对比，这种情况可以通过“最小编辑距离”算法，如果 `a` 和 `b` 的编辑距离越小，我们认为越相似

日常在编写任何代码的都需要一个良好的算法思维，选择好的算法或者数据结构，能让整个程序效率更高

3. 说说你对算法中时间复杂度，空间复杂度的理解？如何计算？



3.1. 前言

算法（Algorithm）是指用来操作数据、解决程序问题的一组方法。对于同一个问题，使用不同的算法，也许最终得到的结果是一样的，但在过程中消耗的资源和时间却会有很大的区别

衡量不同算法之间的优劣主要是通过**时间**和**空间**两个维度去考量：

- 时间维度：是指执行当前算法所消耗的时间，我们通常用「时间复杂度」来描述。
- 空间维度：是指执行当前算法需要占用多少内存空间，我们通常用「空间复杂度」来描述

通常会遇到一种情况，时间和空间维度不能够兼顾，需要在两者之间取得一个平衡点是我们需要考虑的

一个算法通常存在最好、平均、最坏三种情况，我们一般关注的是最坏情况

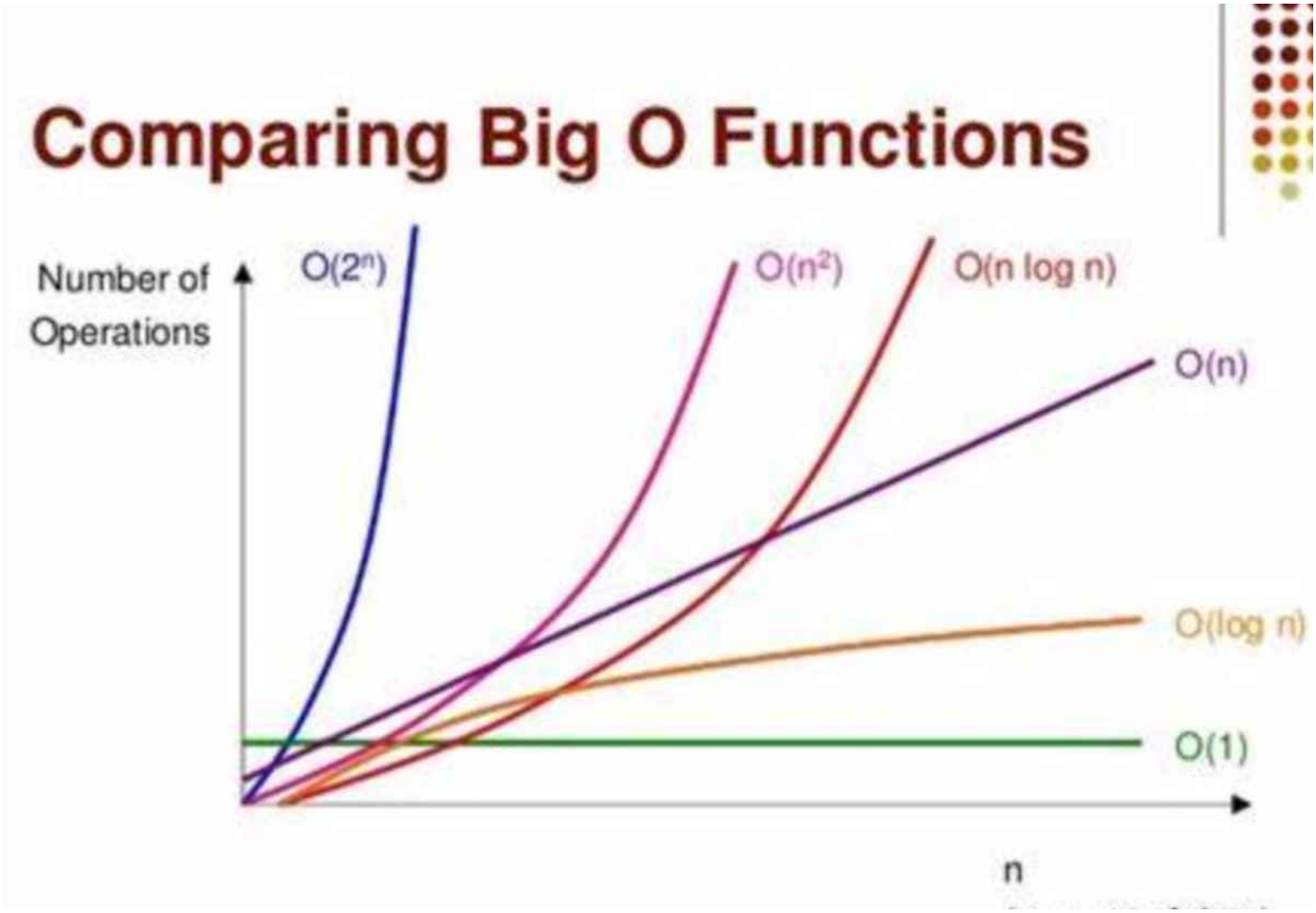
最坏情况是算法运行时间的上界，对于某些算法来说，最坏情况出现的比较频繁，也意味着平均情况和最坏情况一样差

3.2. 时间复杂度

时间复杂度是指执行这个算法所需要的计算工作量，其复杂度反映了程序执行时间「随输入规模增长而增长的量级」，在很大程度上能很好地反映出算法的优劣与否

一个算法花费的时间与算法中语句的「执行次数成正比」，执行次数越多，花费的时间就越多

算法的复杂度通常用大O符号表述，定义为 `T(n) = O(f(n))` ，常见的时间复杂度有：O(1)常数型、O(log n)对数型、O(n)线性型、O(nlogn)线性对数型、O(n(2)平方型、O(n)3)立方型、O(n(k)k次方型、O(2)n)指数型，如下图所示：



从上述可以看到，随着问题规模 `n` 的不断增大，上述时间复杂度不断增大，算法的执行效率越低，由小到大排序如下：

```
O(1)<O(log n)<O(n)<O(nlog n)<O(n2)<O(n3)<…<O(2^n)<O(n!)
```

注意的是，算法复杂度只是描述算法的增长趋势，并不能说一个算法一定比另外一个算法高效，如果常数项过大的时候也会导致算法的执行时间变长

关于如何计算时间复杂度，可以看看如下简单例子：

```
function process(n) {
  let a = 1
  let b = 2
  let sum = a + b
  for(let i = 0; i < n; i++) {
    sum += i
  }
  return sum
}
```

该函数算法需要执行的运算次数用输入大小 `n` 的函数表示，即 `T(n) = 2 + n + 1`，那么时间复杂度为 `O(n + 3)`，又因为时间复杂度只关注最高数量级，且与之系数也没有关系，因此上述的时间复杂度为 `O(n)`

又比如下面的例子：

```
function process(n) {
 let count = 0
  for(let i = 0; i < n; i++){
    for(let i = 0; i < n; i++){
      count += 1
    }
  }
}
```

循环里面嵌套循环，外面的循环执行一次，里面的循环执行 `n` 次，因此时间复杂度为 `O(n*n*1 + 2) = O(n^2)`

对于顺序执行的语句，总的时间复杂度等于其中最大的时间复杂度，如下：

```
function process(n) {
  let sum = 0
  for(let i = 0; i < n; i++) {
    sum += i
  }
  for(let i = 0; i < n; i++){
    for(let i = 0; i < n; i++){
      sum += 1
    }
  }
  return sum
}
```

上述第一部分复杂度为 `O(n)`，第二部分复杂度为 `O(n^2)`，总复杂度为 `max(O(n^2), O(n)) = O(n^2)`

又如下一个例子：

```
function process(n) {
  let i = 1; // ①
  while (i <= n) {
     i = i * 2; // ②
  }
```

```
6 }
```

循环语句中以2的倍数来逼近 `n` ，每次都乘以2。如果用公式表示就是1 *2* 2 *2* ⋯ 2 <=n，也就是说2的 `x` 次方小于等于 `n` 时会执行循环体，记作 `2^x <= n` ，于是得出 `x<=logn`

因此循环在执行 `logn` 次之后，便结束，因此时间复杂度为 `O(logn)`

同理，如果一个 `O(n)` 循环里面嵌套 `O(logn)` 的循环，则时间复杂度为 `O(nlogn)` ，像 `O(n^3)` 无非也就是嵌套了三层 `O(n)` 循环

3.3. 空间复杂度

空间复杂度主要指执行算法所需内存的大小，用于对程序运行过程中所需要的临时存储空间的度量

除了需要存储空间、指令、常数、变量和输入数据外，还包括对数据进行操作的工作单元和存储计算所需信息的辅助空间

下面给出空间复杂度为 `O(1)` 的示例，如下

```
let a = 1
let b = 2
let c = 3
```

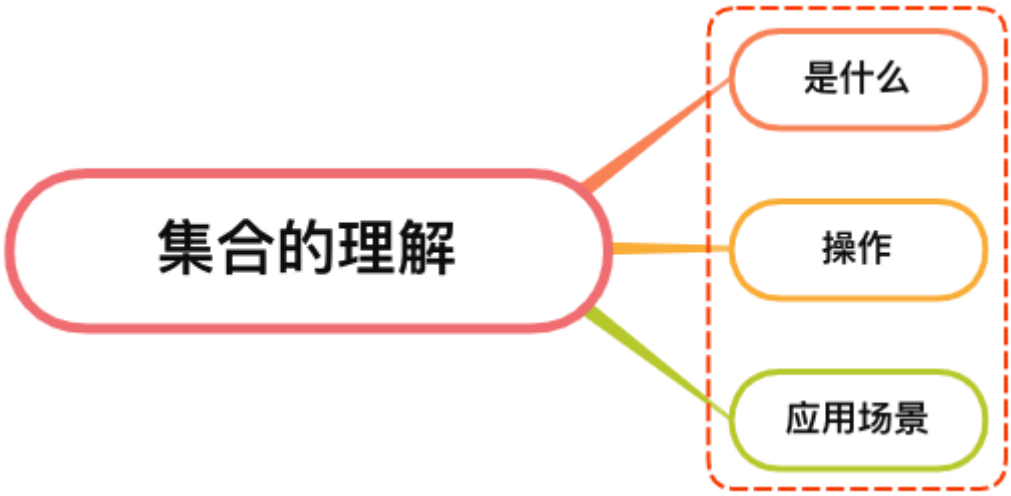
上述代码的临时空间不会随着 `n` 的变化而变化，因此空间复杂度为 `O(1)`

```
let arr []
for(i=1; i<=n; ++i){
  arr.push(i)
}
```

上述可以看到，随着 `n` 的增加，数组的占用的内存空间越大

通常来说，只要算法不涉及到动态分配的空间，以及递归、栈所需的空间，空间复杂度通常为 `O(1)` ，一个一维数组 `a[n]` ，空间复杂度 `O(n)` ，二维数组为 `O(n^2)`

4. 说说你对集合的理解？常见的操作有哪些？



4.1. 是什么

集合（Set），指具有某种特定性质的事物的总体，里面的每一项内容称作元素

在数学中，我们经常会遇到集合的概念：

- 有限集合：例如一个班集所有的同学构成的集合
- 无限集合：例如全体自然数集合

在计算机中集合道理也基本一致，具有三大特性：

- 确定性：于一个给定的集合，集合中的元素是确定的。即一个元素，或者属于该集合，或者不属于该集合，两者必居其一
- 无序性：在一个集合中，不考虑元素之间的顺序，只要元素完全相同，就认为是同一个集合
- 互异性：集合中任意两个元素都是不同的

4.2. 操作

在 `ES6` 中，集合本身是一个构建函数 `Set` ，用来生成 `Set` 数据结构，如下：

```
1 const s = new Set();
```

关于集合常见的方法有：

- add()：增
- delete()：删
- has()：改
- clear()：查

4.2.1. add()

添加某个值，返回 `Set` 结构本身

当添加实例中已经存在的元素， `set` 不会进行处理添加

```
1 s.add(1).add(2).add(2); // 2只被添加了一次
```

体现了集合的互异性特性

4.2.2. delete()

删除某个值，返回一个布尔值，表示删除是否成功

```
1 s.delete(1)
```

4.2.3. has()

返回一个布尔值，判断该值是否为 `Set` 的成员

```
1 s.has(2)
```

4.2.4. clear()

清除所有成员，没有返回值

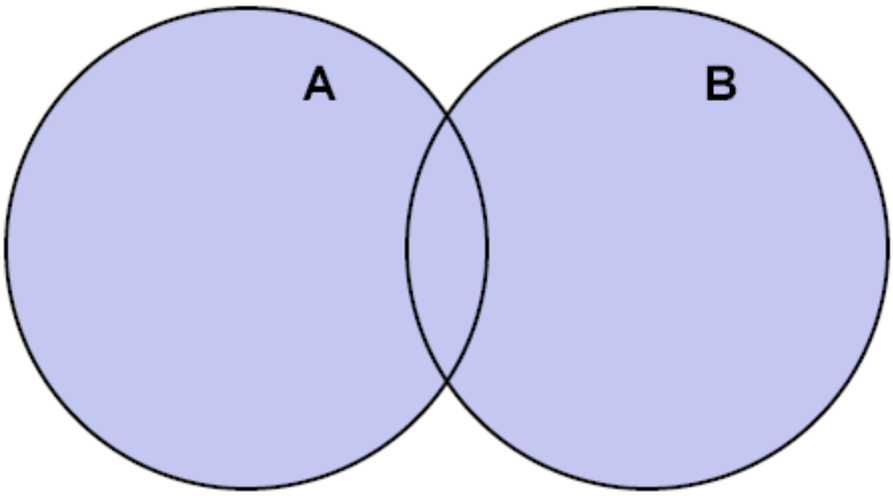
```
1 s.clear()
```

关于多个集合常见的操作有：

- 并集
- 交集
- 差集

4.2.5. 并集

两个集合的共同元素，如下图所示：

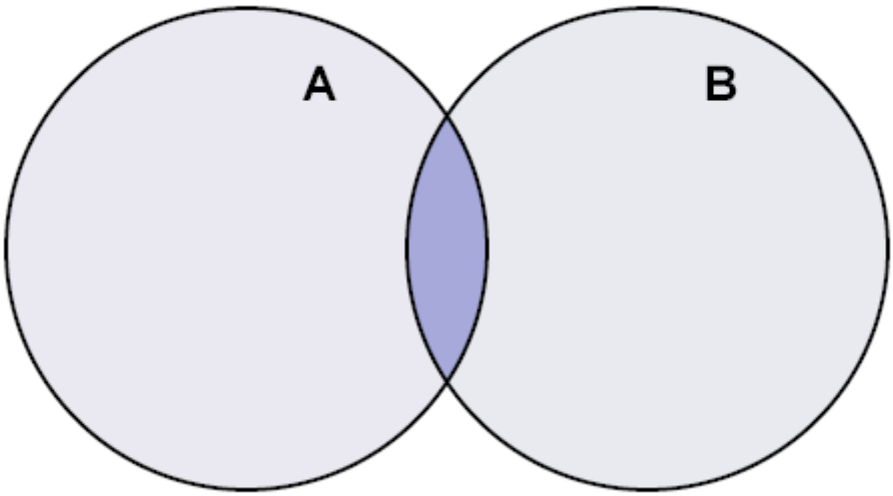


代码实现方式如下：

```
let a = new Set([1, 2, 3]);
let b = new Set([4, 3, 2]);
// 并集
let union = new Set([...a, ...b]);
// Set {1, 2, 3, 4}
```

4.2.6. 交集

两个集合 `A` 和 `B` ，即属于 `A` 又属于 `B` 的元素，如下图所示：



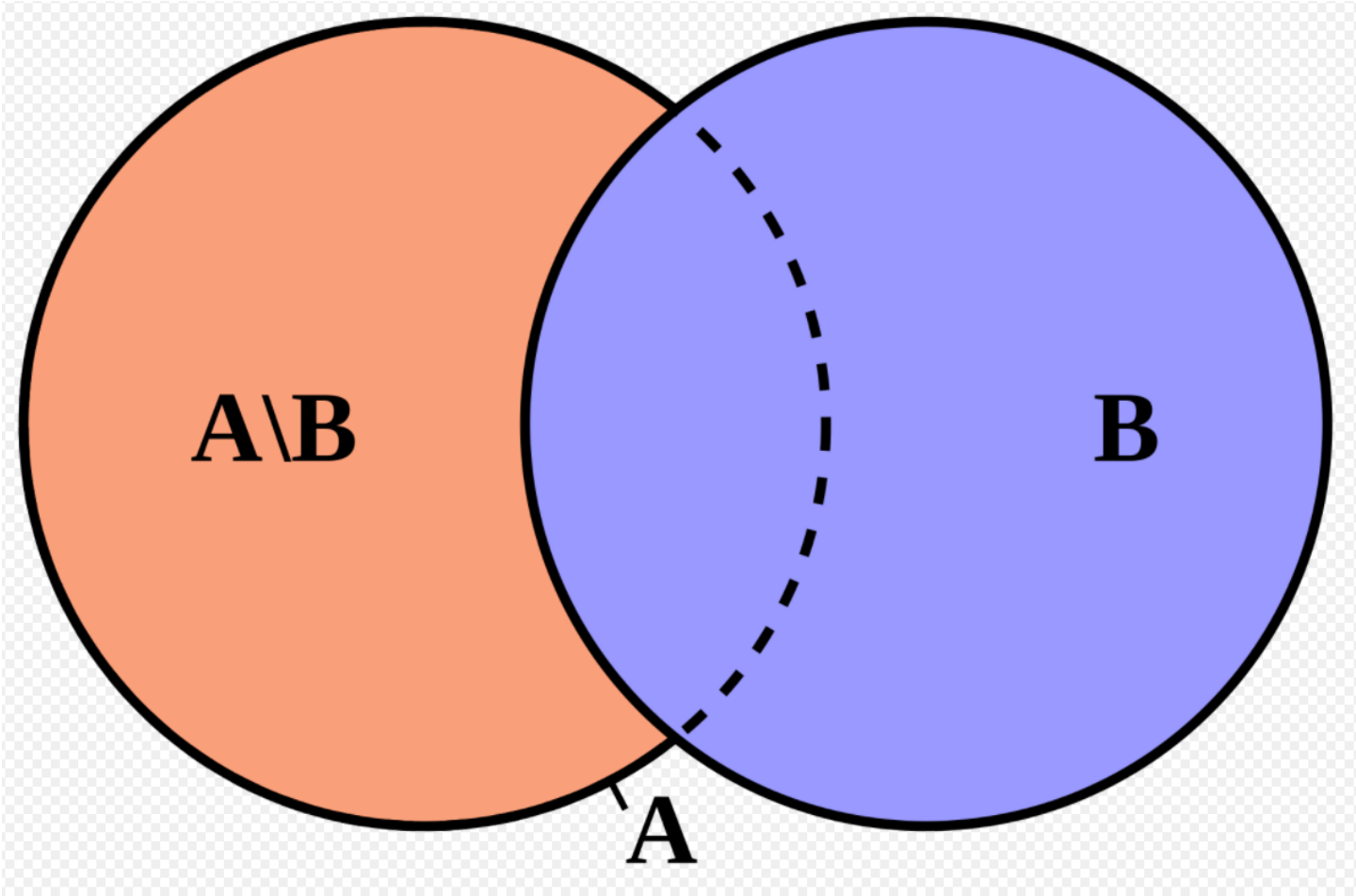
用代码标识则如下：

```
let a = new Set([1, 2, 3]);
let b = new Set([4, 3, 2]);
// 交集
let intersect = new Set([...a].filter(x => b.has(x)));
```

```
5 // set {2, 3}
```

4.2.7. 差集

两个集合 `A` 和 `B` ，属于 `A` 的元素但不属于 `B` 的元素称为 `A` 相对于 `B` 的差集，如下图所示：



代码标识则如下：

```
1 let a = new Set([1, 2, 3]);
2 let b = new Set([4, 3, 2]);
3 // （a 相对于 b 的）差集
4 let difference = new Set([...a].filter(x => !b.has(x)));
5 // Set {1}
```

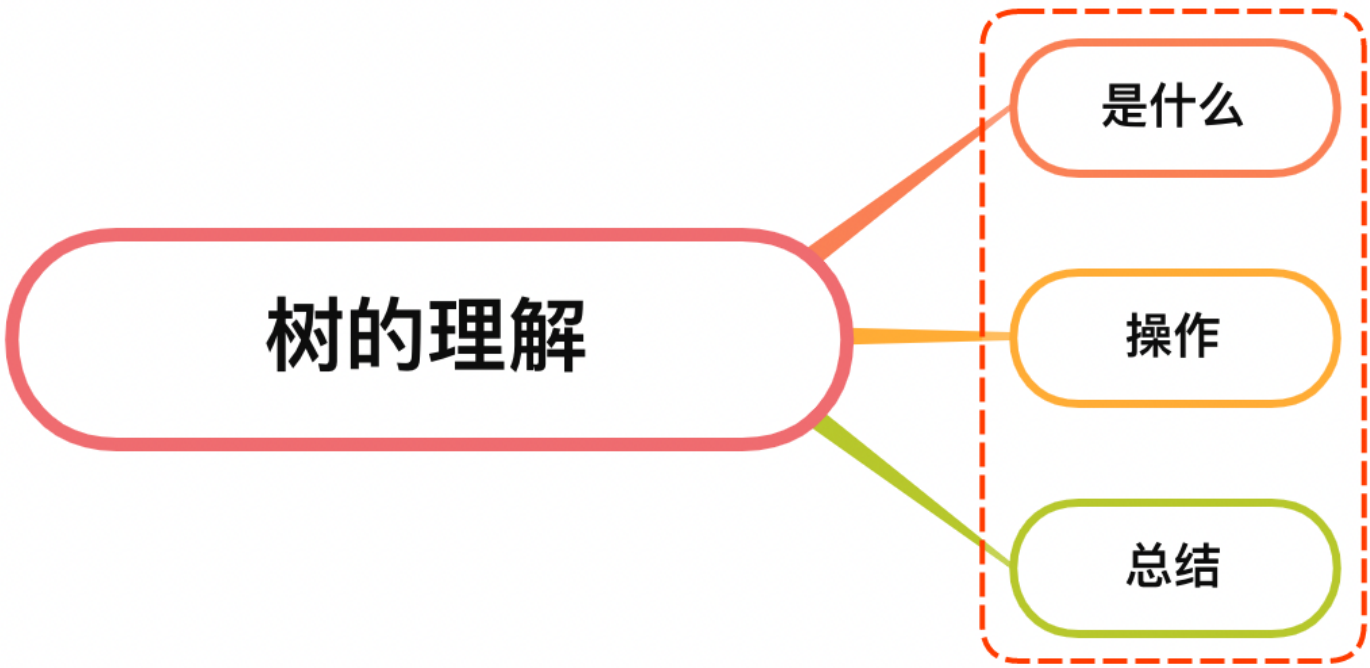
4.3. 应用场景

一般情况下，使用数组的概率会比集合概率高很多

使用 `set` 集合的场景一般是借助其确定性，其本身只包含不同的元素

所以，可以利用 `Set` 的一些原生方法轻松的完成数组去重，查找数组公共元素及不同元素等操作

5. 说说你对树的理解？相关的操作有哪些？



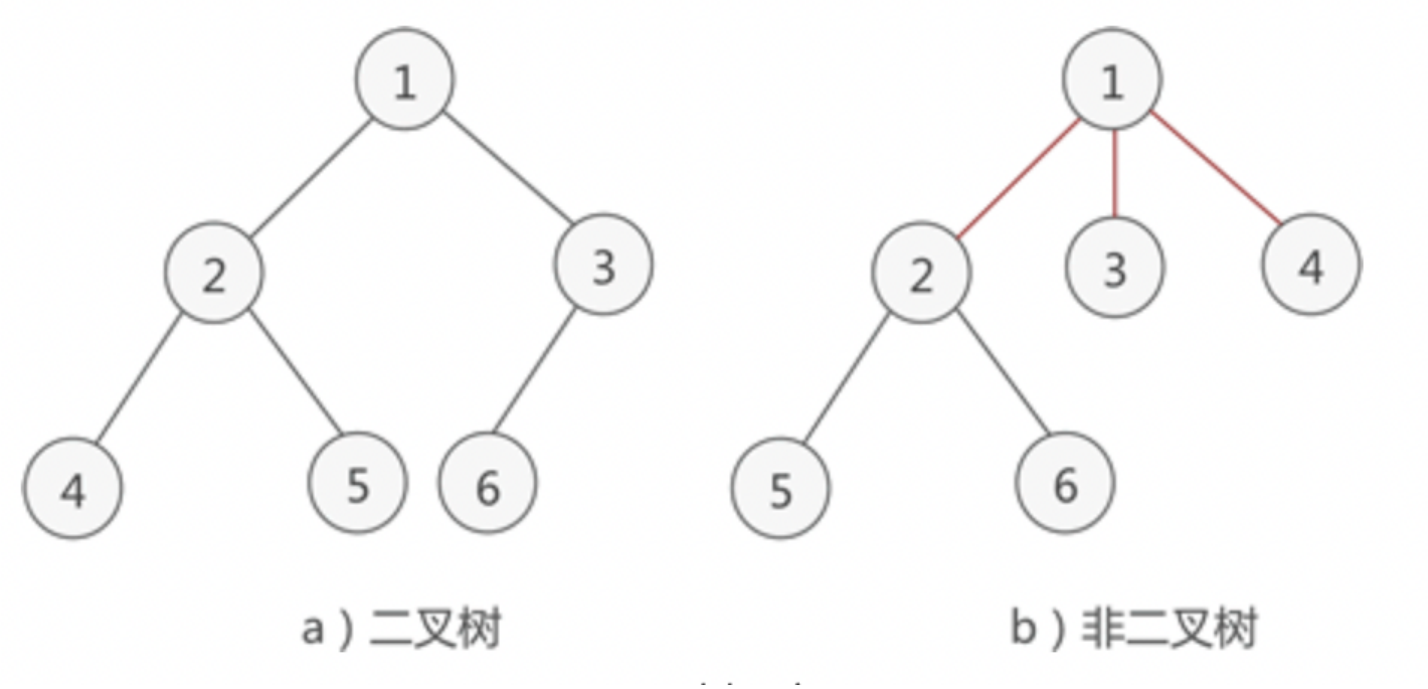
5.1. 是什么

在计算机领域，树形数据结构是一类重要的非线性数据结构，可以表示数据之间一对多的关系。以树与二叉树最为常用，直观看来，树是以分支关系定义的层次结构

二叉树满足以下两个条件：

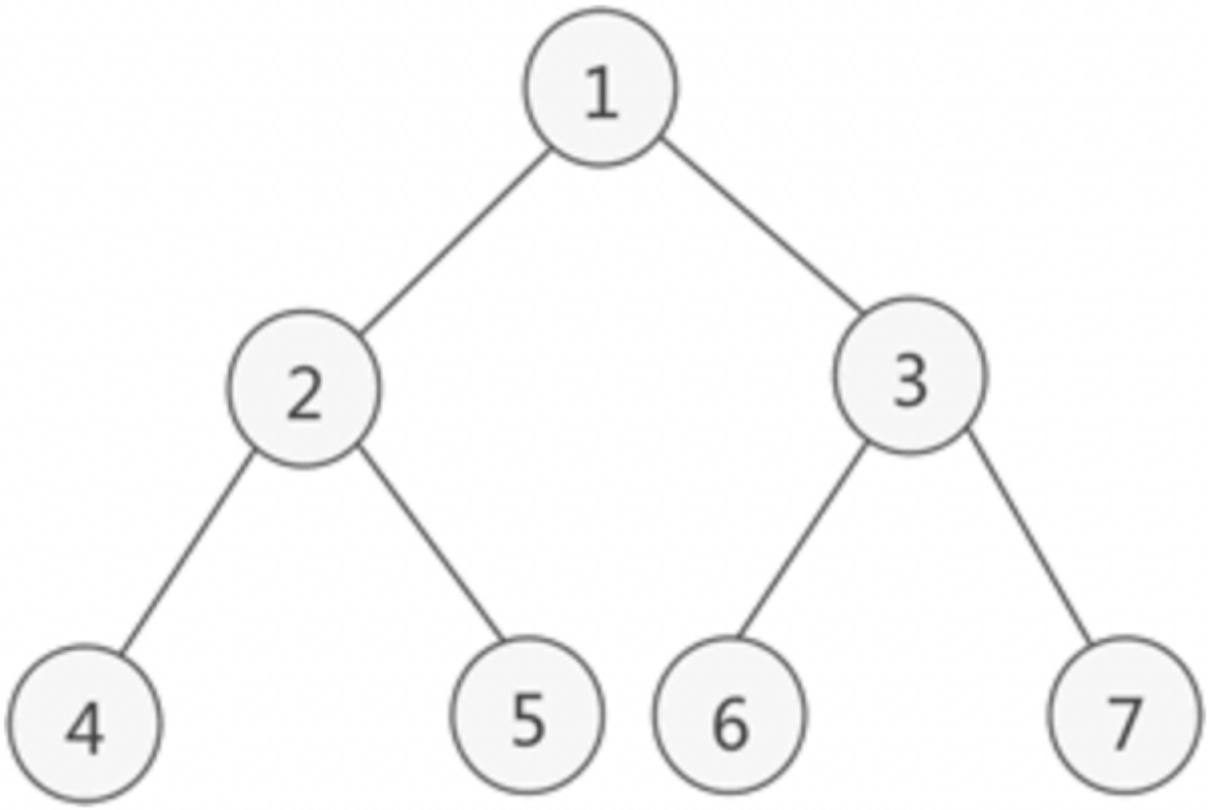
- 本身是有序树
- 树中包含的各个结点的不能超过 2，即只能是 0、1 或者 2

如下图，左侧的为二叉树，而右侧的因为头结点的子结点超过2，因此不属于二叉树：

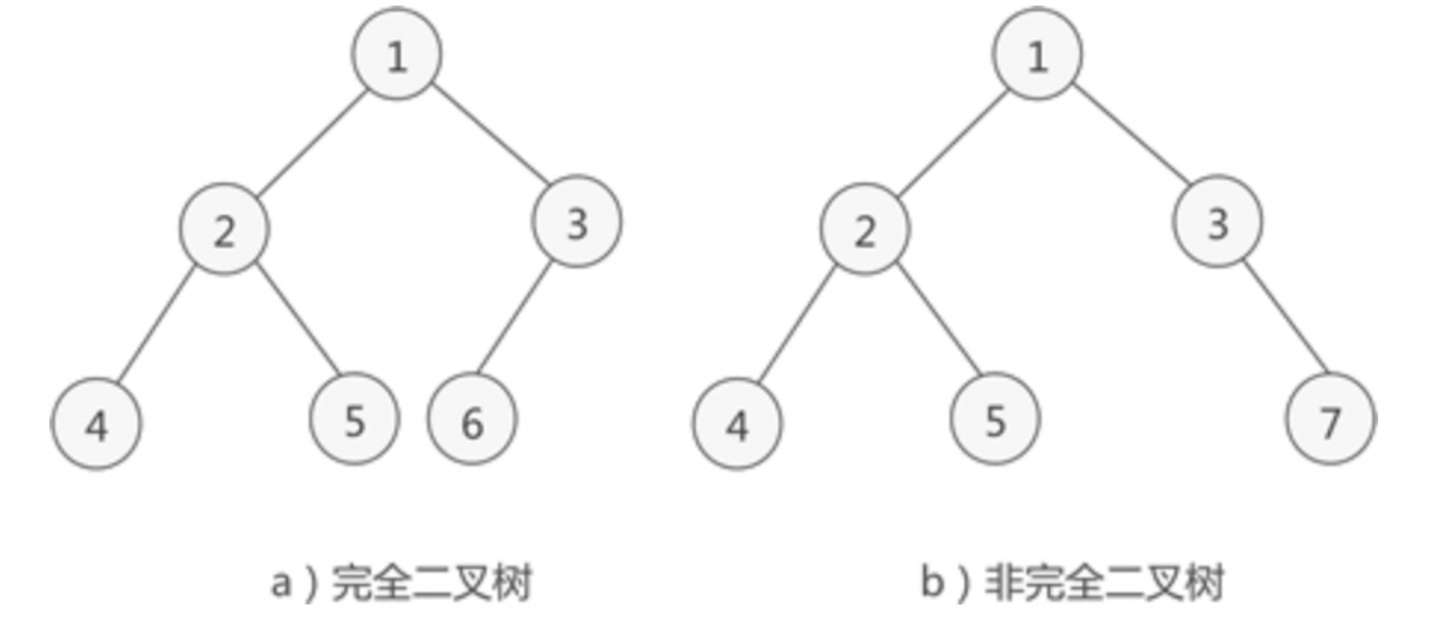


同时，二叉树可以继续进行分类，分成了满二叉树和完成二叉树：

- 满二叉树：如果二叉树中除了叶子结点，每个结点的度都为 2



- 完成二叉树：如果二叉树中除去最后一层节点为满二叉树，且最后一层的结点依次从左到右分布



5.2. 操作

关于二叉树的遍历，常见的有：

- 前序遍历
- 中序遍历
- 后序遍历
- 层序遍历

5.2.1. 前序遍历

前序遍历的实现思想是：

- 访问根节点
- 访问当前节点的左子树
- 若当前节点无左子树，则访问当前节点的右子

根据遍历特性，递归版本用代码表示则如下：

```
1 const preOrder = (root) => {
2   if(!root){ return }
3   console.log(root)
4   preOrder(root.left)
5   preOrder(root.right)
6 }
```

如果不使用递归版本，可以借助栈先进后出的特性实现，先将根节点压入栈，再分别压入右节点和左节点，直到栈中没有元素，如下：

```
const preOrder = (root) => {
  if(!root){ return }
  const stack = [root]
  while (stack.length) {
    const n = stack.pop()
    console.log(n.val)
    if (n.right) {
      stack.push(n.right)
    }
    if (n.left) {
      stack.push(n.left)
    }
  }
}
```

5.2.2. 中序遍历

前序遍历的实现思想是：

- 访问当前节点的左子树
- 访问根节点
- 访问当前节点的右子

递归版本很好理解，用代码表示则如下：

```
const inOrder = (root) => {
  if (!root) { return }
  inOrder(root.left)
  console.log(root.val)
  inOrder(root.right)
}
```

非递归版本也是借助栈先进后出的特性，可以一直首先一直压入节点的左元素，当左节点没有后，才开始进行出栈操作，压入右节点，然后有依次压入左节点，如下：

```
const inOrder = (root) => {
  if (!root) { return }
  const stack = [root]
  let p = root
  while(stack.length || p){
    while (p) {
      stack.push(p)
      p = p.left
```

```
    }
    const n = stack.pop()
    console.log(n.val)
    p = n.right
  }
}
```

5.2.3. 后序遍历

前序遍历的实现思想是：

- 访问当前节点的左子树
- 访问当前节点的右子
- 访问根节点

递归版本，用代码表示则如下：

```
const postOrder = (root) => {
  if (!root) { return }
  postOrder(root.left)
  postOrder(root.right)
  console.log(n.val)
 }
```

后序遍历非递归版本实际根全序遍历是逆序关系，可以再多创建一个栈用来进行输出，如下：

```
const preOrder = (root) => {
  if(!root){ return }
  const stack = [root]
  const outPut = []
  while (stack.length) {
    const n = stack.pop()
    outPut.push(n.val)
    if (n.right) {
      stack.push(n.right)
    }
    if (n.left) {
      stack.push(n.left)
    }
  }
  while (outPut.length) {
    const n = outPut.pop()
    console.log(n.val)
```

```
  }
}
```

5.2.4. 层序遍历

按照二叉树中的层次从左到右依次遍历每层中的结点

借助队列先进先出的特性，从树的根结点开始，依次将其左孩子和右孩子入队。而后每次队列中一个结点出队，都将其左孩子和右孩子入队，直到树中所有结点都出队，出队结点的先后顺序就是层次遍历的最终结果

用代码表示则如下：

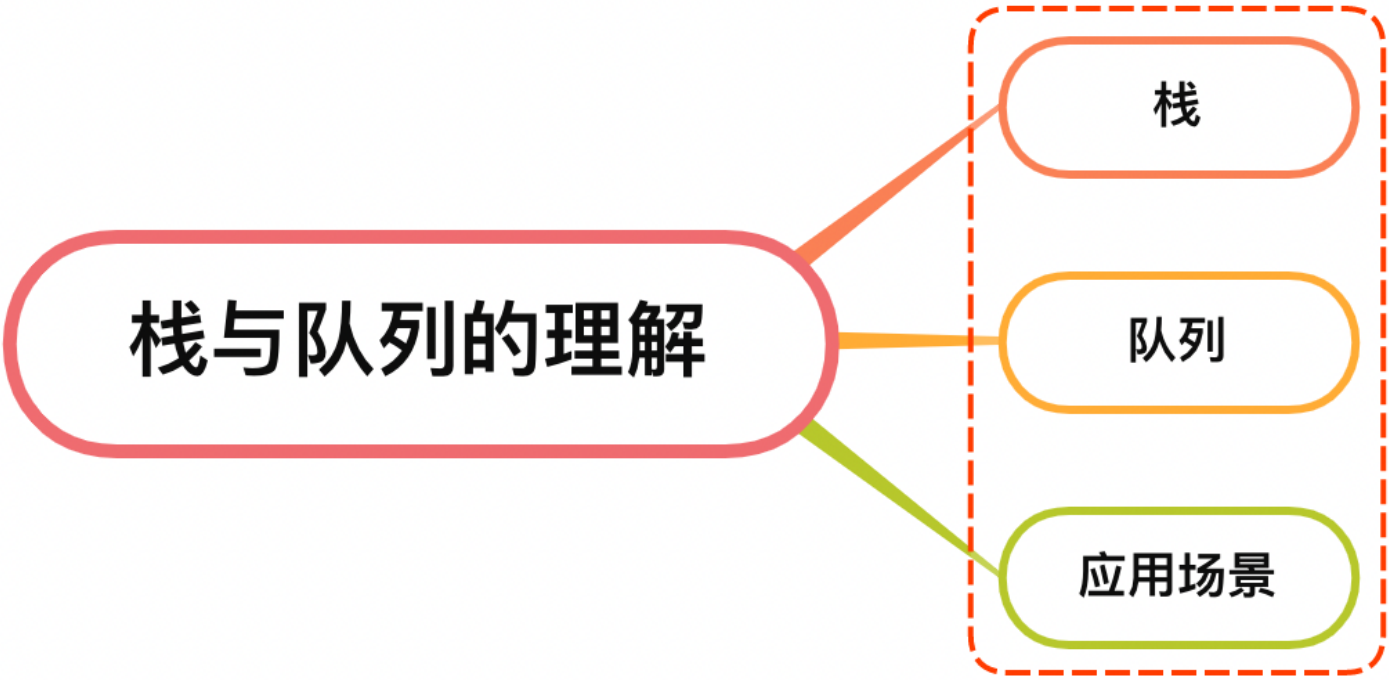
```
const levelOrder = (root) => {
    if (!root) { return [] }
    const queue = [[root, 0]]
    const res = []
    while (queue.length) {
        const n = queue.shift()
        const [node, leval] = n
        if (!res[leval]) {
            res[leval] = [node.val]
        } else {
            res[leval].push(node.val)
        }
        if (node.left) { queue.push([node.left, leval + 1]) }
        if (node.right) { queue.push([node.right, leval + 1]) }
    }
    return res
};
```

5.3. 总结

树是一个非常重要的非线性结构，其中二叉树以二叉树最常见，二叉树的遍历方式可以分成前序遍历、中序遍历、后序遍历

同时，二叉树又分成了完成二叉树和满二叉树

6. 说说你对栈、队列的理解？应用场景？



6.1. 栈

栈（stack）又名堆栈，它是一种运算受限的线性表，限定仅在表尾进行插入和删除操作的线性表

表尾这一端被称为栈顶，相反地另一端被称为栈底，向栈顶插入元素被称为进栈、入栈、压栈，从栈顶删除元素又称作出栈

所以其按照先进后出的原则存储数据，先进入的数据被压入栈底，最后的数据在栈顶，需要读数据的时候从栈顶开始弹出数据，具有记忆作用

关于栈的简单实现，如下：

- class Stack {
  constructor() {
  this.items = [];
  }
  /**
  添加一个（或几个）新元素到栈顶
  @param {*} element 新元素
  */
  push(element) {
  this.items.push(element)
  }
  /**
  移除栈顶的元素，同时返回被移除的元素
  */
  pop() {
  return this.items.pop()

```
}
 /**
返回栈顶的元素，不对栈做任何修改（这个方法不会移除栈顶的元素，仅仅返回它）
*/
peek() {
return this.items[this.items.length - 1]
}
 /**
如果栈里没有任何元素就返回true,否则返回false
*/
isEmpty() {
return this.items.length === 0
}
 /**
移除栈里的所有元素
*/
clear() {
this.items = []
}
 /**
返回栈里的元素个数。这个方法和数组的length属性很类似
*/
size() {
return this.items.length
}
}
```

关于栈的操作主要的方法如下：

- push：入栈操作
- pop：出栈操作

6.2. 队列

跟栈十分相似，队列是一种特殊的线性表，特殊之处在于它只允许在表的前端（front）进行删除操作，而在表的后端（rear）进行插入操作

进行插入操作的端称为队尾，进行删除操作的端称为队头，当队列中没有元素时，称为空队列

在队列中插入一个队列元素称为入队，从队列中删除一个队列元素称为出队。因为队列只允许在一端插入，在另一端删除，所以只有最早进入队列的元素才能最先从队列中删除，故队列又称为先进先出

简单实现一个队列的方式，如下：

```
class Queue {
    constructor() {
        this.list = []
        this.frontIndex = 0
        this.tailIndex = 0
    }
    enqueue(item) {
        this.list[this.tailIndex++] = item
    }
    unqueue() {
        const item  = this.list[this.frontIndex]
        this.frontIndex++

        return item
    }
}
```

上述这种入队和出队操作中，头尾指针只增加不减小，致使被删元素的空间永远无法重新利用

当队列中实际的元素个数远远小于向量空间的规模时，也可能由于尾指针已超越向量空间的上界而不能做入队操作，出该现象称为"假溢"

在实际使用队列时，为了使队列空间能重复使用，往往对队列的使用方法稍加改进：

无论插入或删除，一旦 `rear` 指针增1或 `front` 指针增1 时超出了所分配的队列空间，就让它指向这片连续空间的起始位置，这种队列也就是循环队列

下面实现一个循环队列，如下：

```
class Queue {
    constructor(size) {
        this.size = size; // 长度需要限制，来达到空间的利用，代表空间的长度
        this.list = [];
        this.font = 0; // 指向首元素
        this.rear = 0;  // 指向准备插入元素的位置
    }
    enQueue() {
        if (this.isFull() == true) {
            return false
        }
        this.rear = this.rear % this.k;
        this._data[this.rear++] = value;
        return true
    }
    deQueue() {
        if(this.isEmpty()){
            return false;
```

```
        }
        this.font++;
        this.font = this.font % this.k;
        return true;
    }
    isEmpty() {
        return this.font == this.rear - 1;
    }
    isFull() {
        return this.rear % this.k == this.font;
    }
}
```

上述通过求余的形式代表首尾指针增1 时超出了所分配的队列空间

6.3. 应用场景

6.3.1. 栈

借助栈的先进后出的特性，可以简单实现一个逆序数处的功能，首先把所有元素依次入栈，然后把所有元素出栈并输出

包括编译器的在对输入的语法进行分析的时候，例如`"()"`、`"{}"`、`"[]"`这些成对出现的符号，借助栈的特性，凡是遇到括号的前半部分，即把这个元素入栈，凡是遇到括号的后半部分就比对栈顶元素是否该元素相匹配，如果匹配，则前半部分出栈，否则就是匹配出错

包括函数调用和递归的时候，每调用一个函数，底层都会进行入栈操作，出栈则返回函数的返回值

生活中的例子，可以把乒乓球盒比喻成一个堆栈，球一个一个放进去（入栈），最先放进去的要等其后面的全部拿出来后才能出来（出栈），这种就是典型的先进后出模型

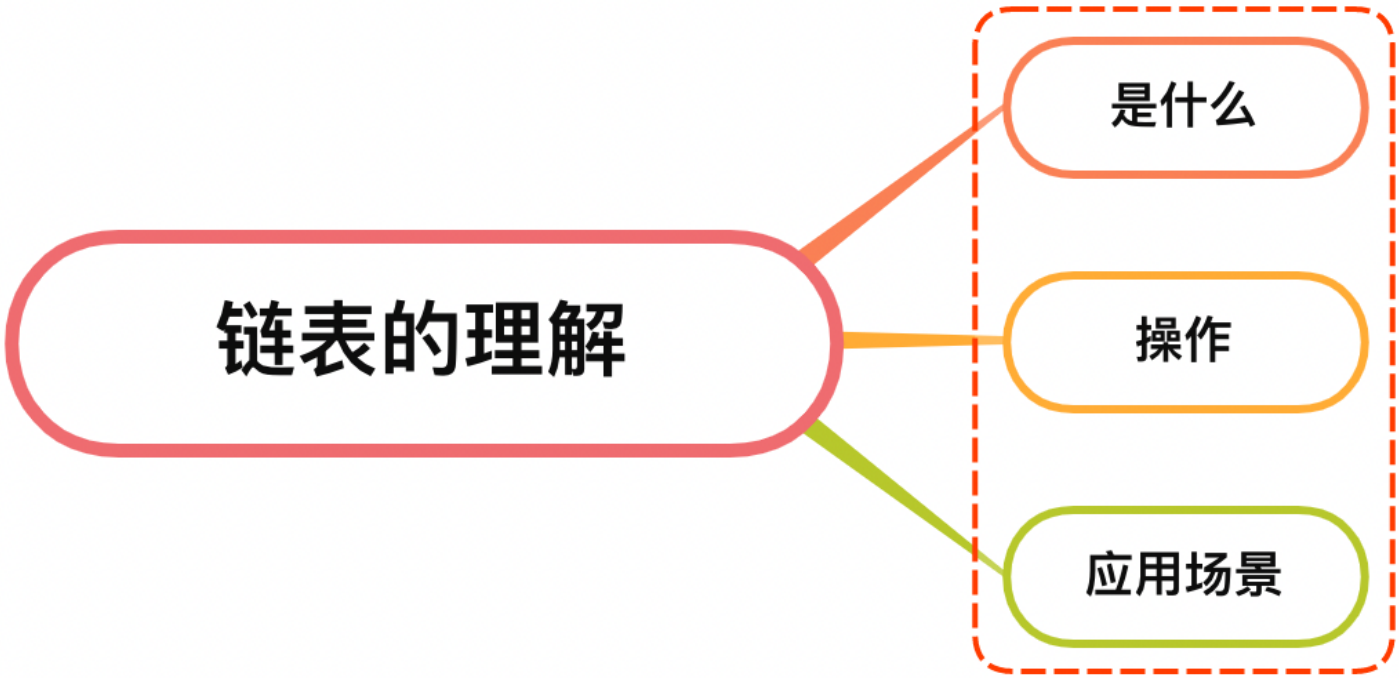
6.3.2. 队列

当我们需要按照一定的顺序来处理数据，而该数据的数据量在不断地变化的时候，则需要队列来帮助解题

队列的使用广泛应用在广度优先搜索种，例如层次遍历一个二叉树的节点值（后续将到）

生活中的例子，排队买票，排在队头的永远先处理，后面的必须等到前面的全部处理完毕再进行处理，这也是典型的先进先出模型

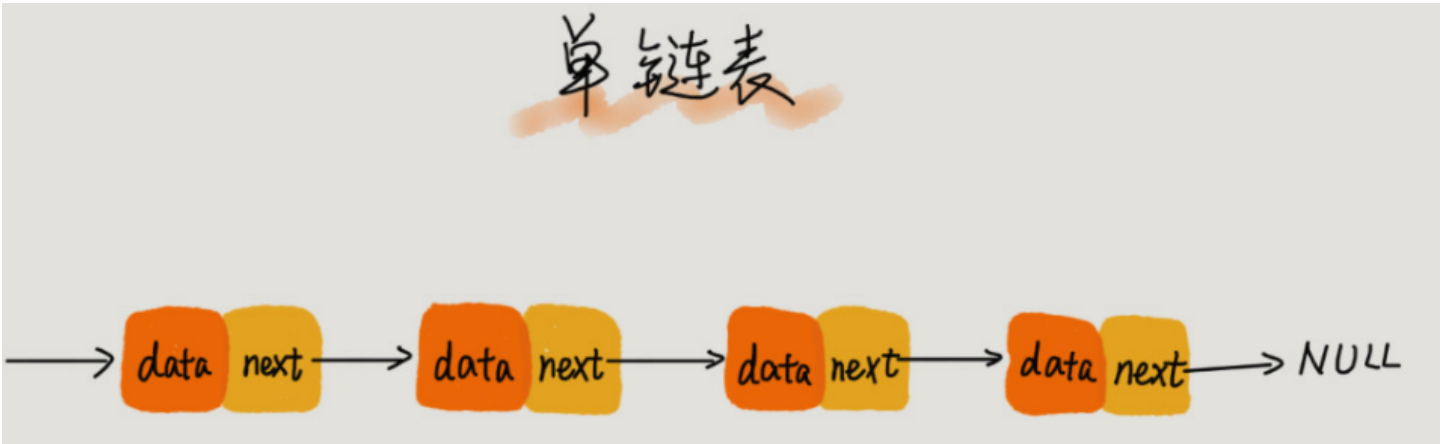
7. 说说你对链表的理解？常见的操作有哪些？



7.1. 是什么

链表（Linked List）是一种物理存储单元上非连续、非顺序的存储结构，数据元素的逻辑顺序是通过链表中的指针链接次序实现的，由一系列结点（链表中每一个元素称为结点）组成

每个结点包括两个部分：一个是存储数据元素的数据域，另一个是存储下一个结点地址的指针域



节点用代码表示，则如下：

```
class Node {
  constructor(val) {
    this.val = val;
    this.next = null;
  }
}
```

- data 表示节点存放的数据

- next 表示下一个节点指向的内存空间

相比于线性表顺序结构，操作复杂。由于不必须按顺序存储，链表在插入的时候可以达到 `O(1)` 的复杂度，比另一种线性表顺序表快得多，但是查找一个节点或者访问特定编号的节点则需要O(n)的时间，而线性表和顺序表相应的时间复杂度分别是 `O(logn)` 和 `O(1)`

链表的结构也十分多，常见的有四种形式：

- 单链表：除了头节点和尾节点，其他节点只包含一个后继指针
- 循环链表：跟单链表唯一的区别就在于它的尾结点又指回了链表的头结点，首尾相连，形成了一个环
- 双向链表：每个结点具有两个方向指针，后继指针(next)指向后面的结点，前驱指针(prev)指向前面的结点，其中节点的前驱指针和尾结点的后继指针均指向空地址NULL
- 双向循环链表：跟双向链表基本一致，不过头节点前驱指针指向尾迹诶单和尾节点的后继指针指向头节点

7.2. 操作

关于链表的操作可以主要分成如下：

- 遍历
- 插入
- 删除

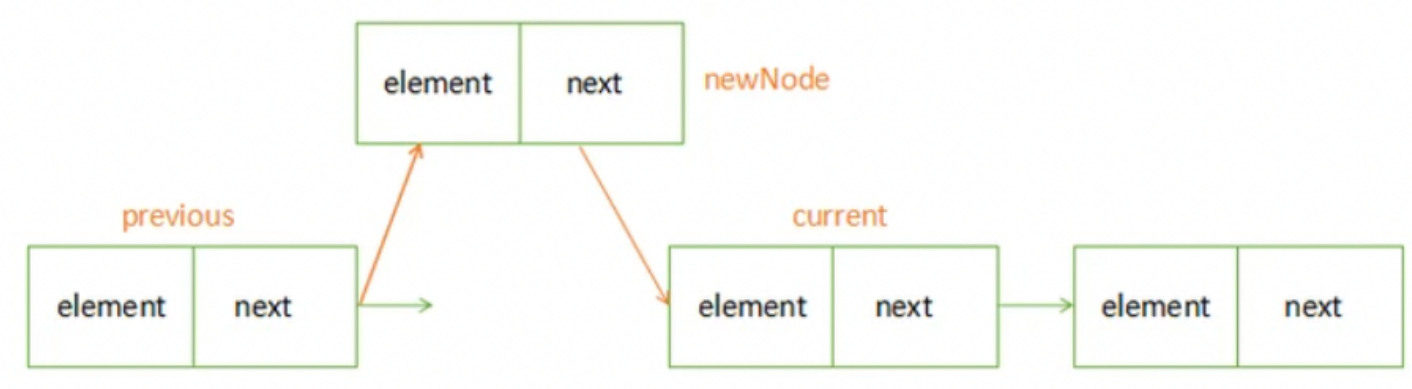
7.2.1. 遍历

遍历很好理解，就是根据 `next` 指针遍历下去，直到为 `null` ，如下：

```
let current = head
while(current){
 console.log(current.val)
  current = current.next
}
```

7.2.2. 插入

向链表中间插入一个元素，可以如下图所示：



可以看到，插入节点可以分成如下步骤：

- 存储插入位置的前一个节点
- 存储插入位置的后一个节点
- 将插入位置的前一个节点的 next 指向插入节点
- 将插入节点的 next 指向前面存储的 next 节点

相关代码如下所示：

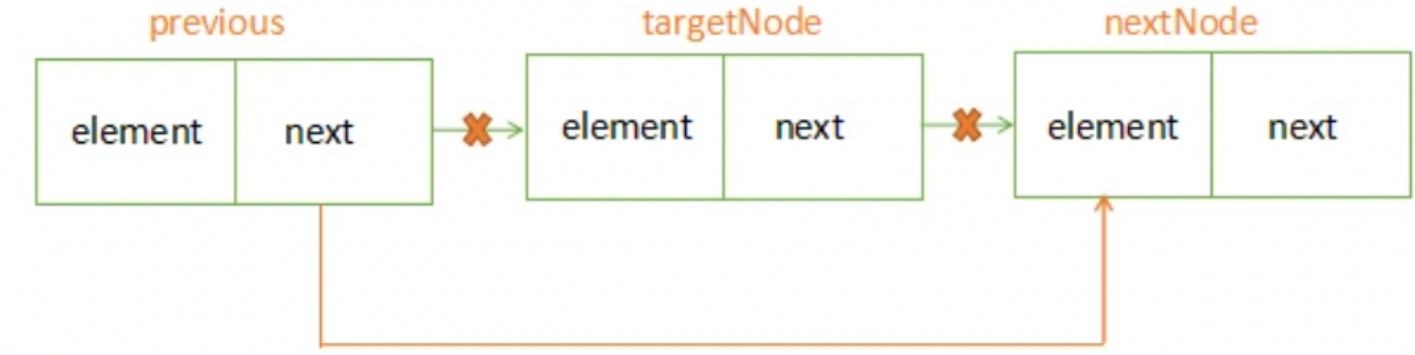
```
let current = head
while (current < position){
  pervious = current;
  current = current.next;
}
pervious.next = node;
node.next = current;
```

如果在头节点进行插入操作的时候，会实现 `previousNode` 节点为 `undefined` ，不适合上述方式

解放方式可以是在头节点前面添加一个虚拟头节点，保证插入行为一致

7.2.3. 删除

向链表任意位置删除节点，如下图操作：



从上图可以看到删除节点的步骤为如下：

- 获取删除节点的前一个节点
- 获取删除节点的后一个节点
- 将前一个节点的 next 指向后一个节点
- 向删除节点的 next 指向为null

如果想要删除制定的节点，示意代码如下：

```
while (current != node){
  pervious = current;
  current = current.next;
  nextNode = current.next;
}
pervious.next = nextNode
```

同样如何希望删除节点处理行为一致，可以在头节点前面添加一个虚拟头节点

7.3. 应用场景

缓存是一种提高数据读取性能的技术，在硬件设计、软件开发中都有着非常广泛的应用，比如常见的 `CPU` 缓存、数据库缓存、浏览器缓存等等

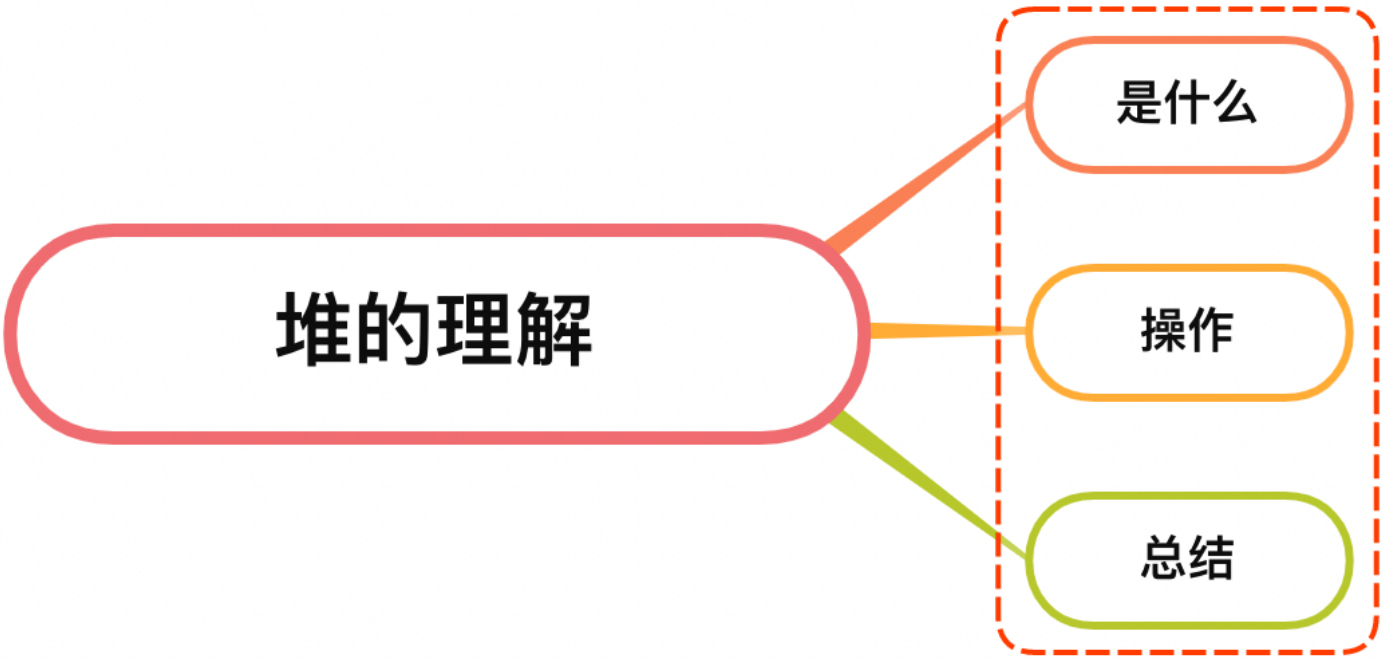
当缓存空间被用满时，我们可能会使用 `LRU` 最近最好使用策略去清楚，而实现 `LRU` 算法的数据结构是链表，思路如下：

维护一个有序单链表，越靠近链表尾部的结点是越早之前访问的。当有一个新的数据被访问时，我们从链表头部开始顺序遍历链表

- 如果此数据之前已经被缓存在链表中了，我们遍历得到这个数据的对应结点，并将其从原来的位置删除，并插入到链表头部
- 如果此数据没在缓存链表中
  - 如果此时缓存未满，可直接在链表头部插入新节点存储此数据
  - 如果此时缓存已满，则删除链表尾部节点，再在链表头部插入新节点

由于链表插入删除效率极高，达到O(1)。对于不需要搜索但变动频繁且无法预知数量上限的数据的情况的时候，都可以使用链表

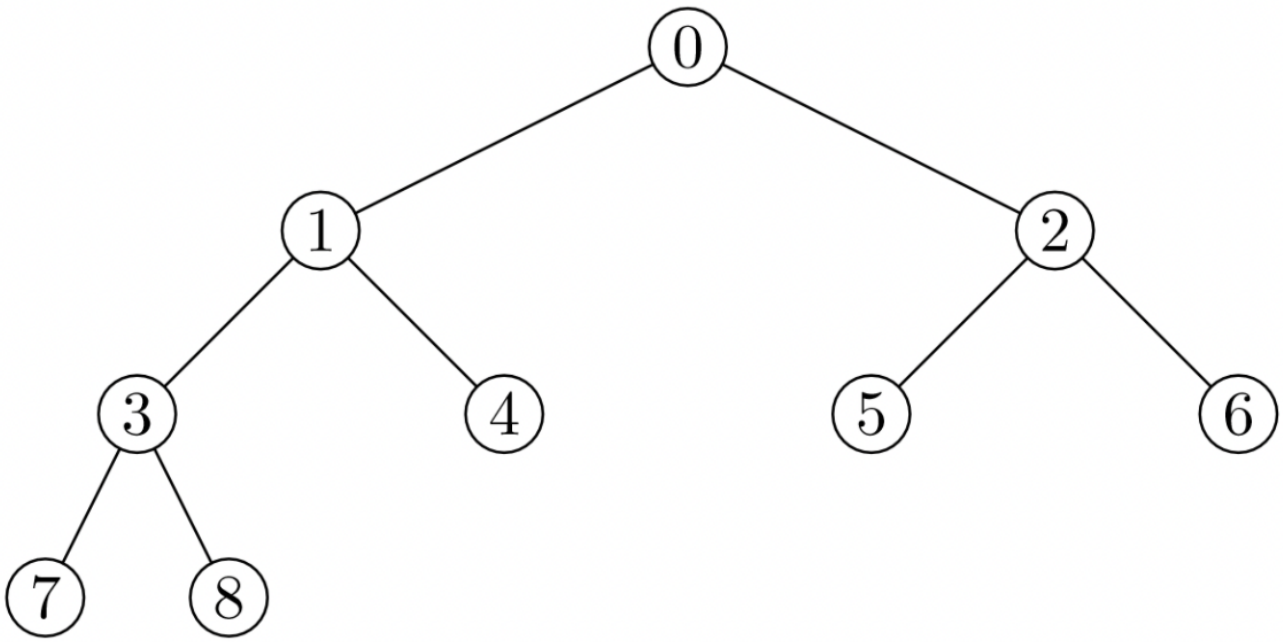
8. 说说你对堆的理解？如何实现？应用场景？



8.1. 是什么

堆(Heap)是计算机科学中一类特殊的数据结构的统称

堆通常是一个可以被看做一棵完全二叉树的数组对象，如下图：



总是满足下列性质：

- 堆中某个结点的值总是不大于或不小于其父结点的值
- 堆总是一棵完全二叉树

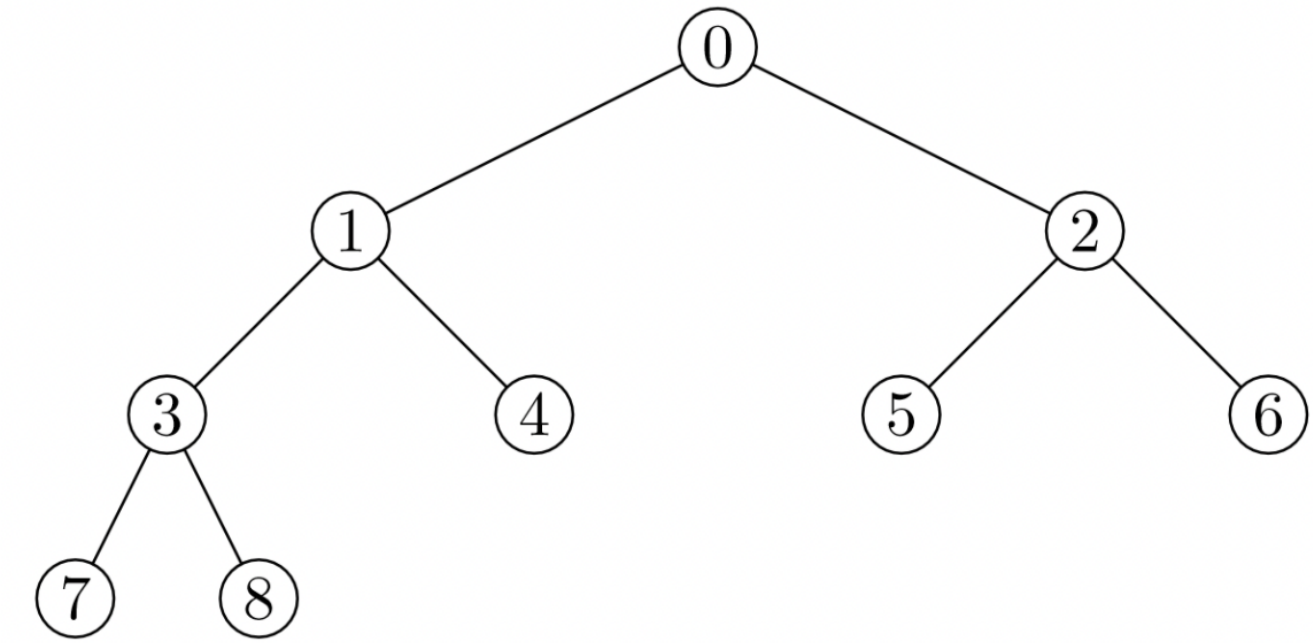
堆又可以分成最大堆和最小堆：

- 最大堆：每个根结点，都有根结点的值大于两个孩子结点的值

- 最小堆：每个根结点，都有根结点的值小于孩子结点的值

8.2. 操作

堆的元素存储方式，按照完全二叉树的顺序存储方式存储在一个一维数组中，如下图：



用一维数组存储则如下：

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

根据完全二叉树的特性，可以得到如下特性：

- 数组零坐标代码的是堆顶元素
- 一个节点的父亲节点的坐标等于其坐标除以2整数部分
- 一个节点的左节点等于其本身节点坐标 * 2 + 1
- 一个节点的右节点等于其本身节点坐标 * 2 + 2

根据上述堆的特性，下面构建最小堆的构造函数和对应的属性方法：

```
class MinHeap {
  constructor() {
    // 存储堆元素
    this.heap = []
  }
  // 获取父元素坐标
  getParentIndex(i) {
    return (i - 1) >> 1
```

```
  }
  // 获取左节点元素坐标
  getLeftIndex(i) {
    return i * 2 + 1
  }
 // 获取右节点元素坐标
  getRightIndex(i) {
    return i * 2 + 2
  }
  // 交换元素
  swap(i1, i2) {
    const temp = this.heap[i1]
    this.heap[i1] = this.heap[i2]
    this.heap[i2] = temp
  }
  // 查看堆顶元素
  peek() {
    return this.heap[0]
  }
  // 获取堆元素的大小
  size() {
    return this.heap.length
  }
}
```

涉及到堆的操作有：

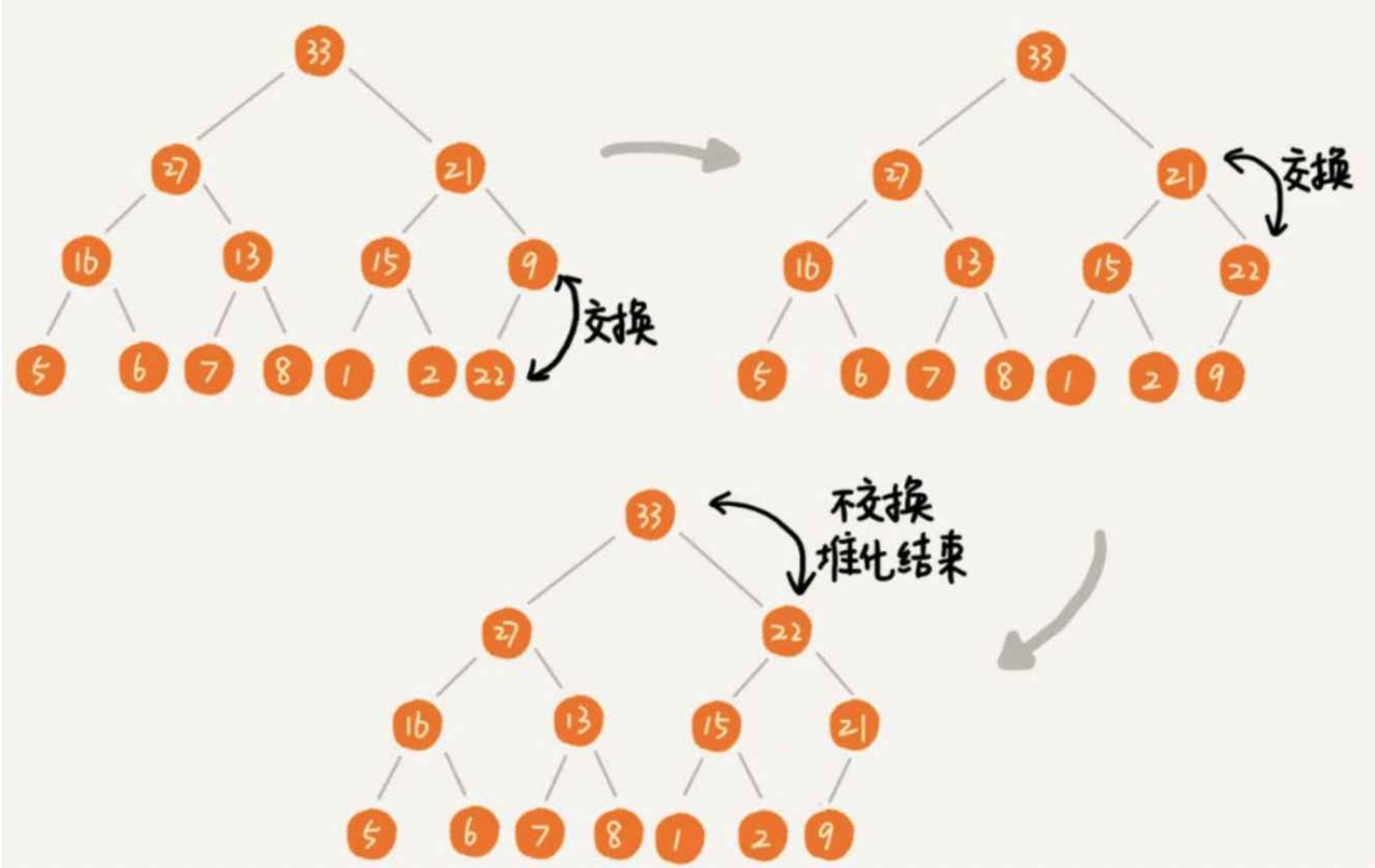
- 插入
- 删除

8.2.1. 插入

将值插入堆的底部，即数组的尾部，当插入一个新的元素之后，堆的结构就会被破坏，因此需要堆中一个元素做上移操作

将这个值和它父节点进行交换，直到父节点小于等于这个插入的值，大小为 `k` 的堆中插入元素的时间复杂度为 `O(logk)`

如下图所示，22节点是新插入的元素，然后进行上移操作：



相关代码如下：

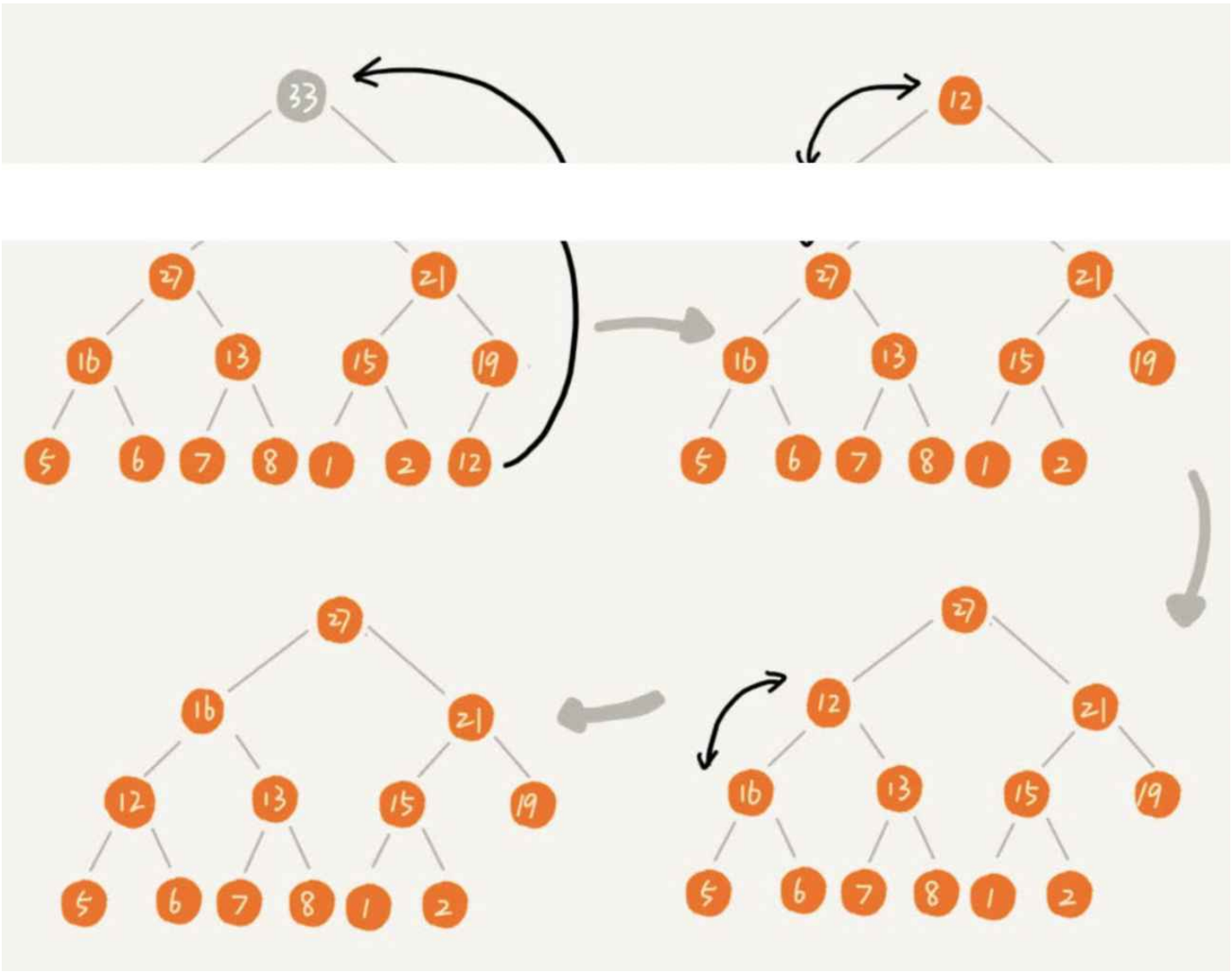
```
// 插入元素
insert(value) {
  this.heap.push(value)
  this.shifUp(this.heap.length - 1)
}
// 上移操作
shiftUp(index) {
  if (index === 0) { return }
  const parentIndex = this.getParentIndex(index)
  if(this.heap[parentIndex] > this.heap[index]){
    this.swap(parentIndex, index)
    this.shiftUp(parentIndex)
  }
}
```

8.2.2. 删除

常见操作是用数组尾部元素替换堆顶，这里不直接删除堆顶，因为所有的元素会向前移动一位，会破坏了堆的结构

然后进行下移操作，将新的堆顶和它的子节点进行交换，直到子节点大于等于这个新的堆顶，删除堆顶的时间复杂度为 `O(logk)`

整体如下图操作：



相关代码如下：

```
// 删除元素
pop() {
  this.heap[0] = this.heap.pop()
  this.shiftDown(0)
}
// 下移操作
shiftDown(index) {
  const leftIndex = this.getLeftIndex(index)
  const rightIndex = this.getRightIndex(index)
  if (this.heap[leftIndex] < this.heap[index]){
    this.swap(leftIndex, index)
    this.shiftDown(leftIndex)
  }
  if (this.heap[rightIndex] < this.heap[index]){
    this.swap(rightIndex, index)
    this.shiftDown(rightIndex)
```

```
17    }
18 }
```

8.2.3. 时间复杂度

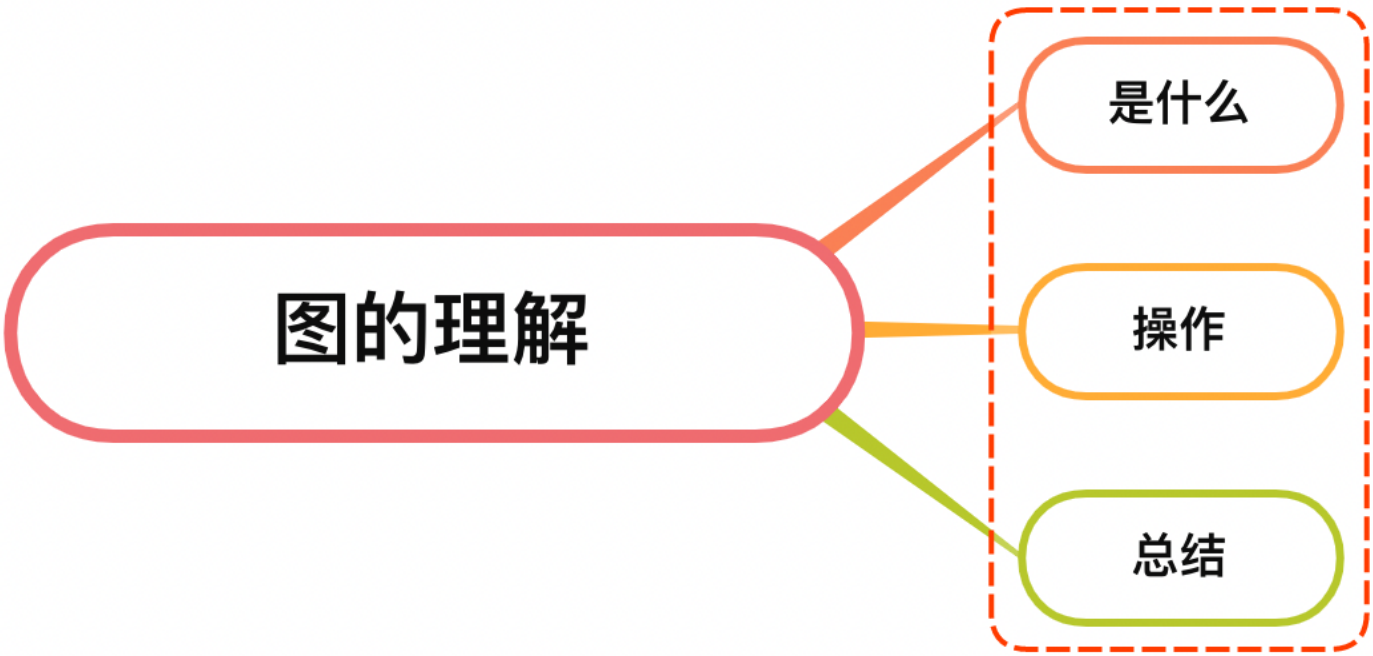
关于堆的插入和删除时间复杂度都是 `Olog(n)` ，原因在于包含n个节点的完全二叉树，树的高度不会超过 `log2n`

堆化的过程是顺着节点所在路径比较交换的，所以堆化的时间复杂度跟树的高度成正比，也就是 `Olog(n)` ，插入数据和删除堆顶元素的主要逻辑就是堆化

8.3. 总结

- 堆是一个完全二叉树
- 堆中每一个节点的值都必须大于等于(或小于等于)其子树中每个节点的值
- 对于每个节点的值都大于等于子树中每个节点值的堆，叫作“大顶堆”
- 对于每个节点的值都小于等于子树中每个节点值的堆，叫作“小顶堆”
- 根据堆的特性，我们可以使用堆来进行排序操作，也可以使用其来求第几大或者第几小的值

9. 说说你对图的理解？相关操作有哪些？



9.1. 是什么

在计算机科学中，图是一种抽象的数据类型，在图中的数据元素通常称为结点， `V` 是所有顶点的集合， `E` 是所有边的集合

如果两个顶点 `v`,`w` ，只能由 `v` 向 `w` ，而不能由 `w` 向 `v` ，那么我们就把这种情况叫做一个从 `v` 到 `w` 的有向边。 `v` 也被称做初始点， `w` 也被称为终点。这种图就被称做有向图

如果 `v` 和 `w` 是没有顺序的，从 `v` 到达 `w` 和从 `w` 到达 `v` 是完全相同的，这种图就被称为无向图

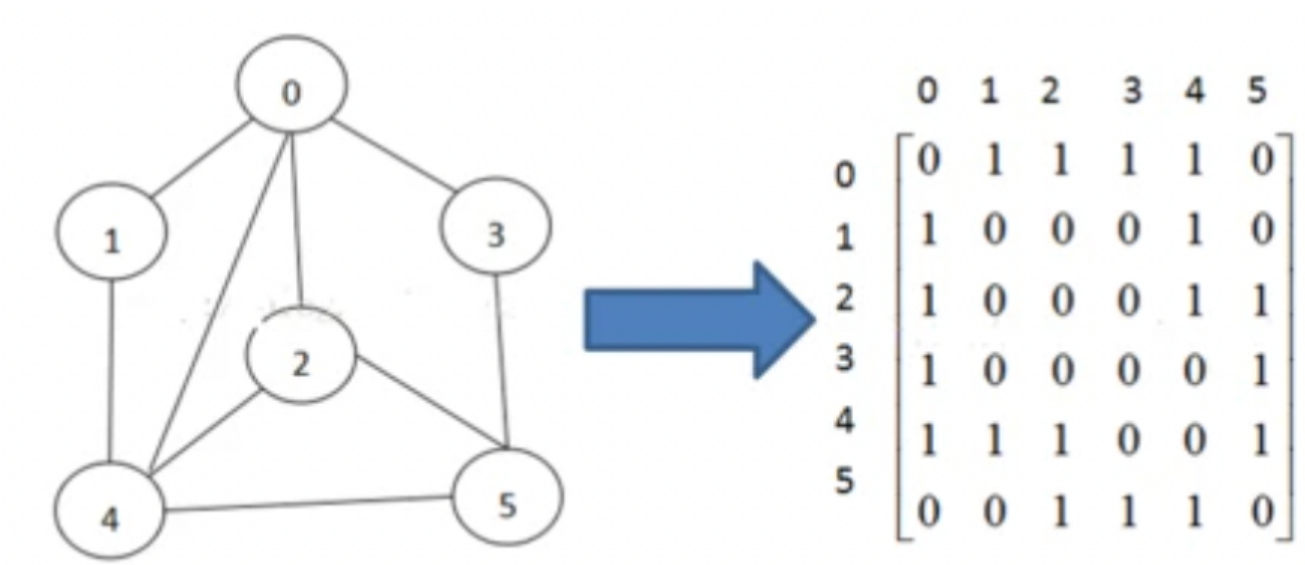
图的结构比较复杂，任意两个顶点之间都可能存在联系，因此无法以数据元素在存储区中的物理位置来表示元素之间的关系

常见表达图的方式有如下：

- 邻接矩阵
- 邻接表

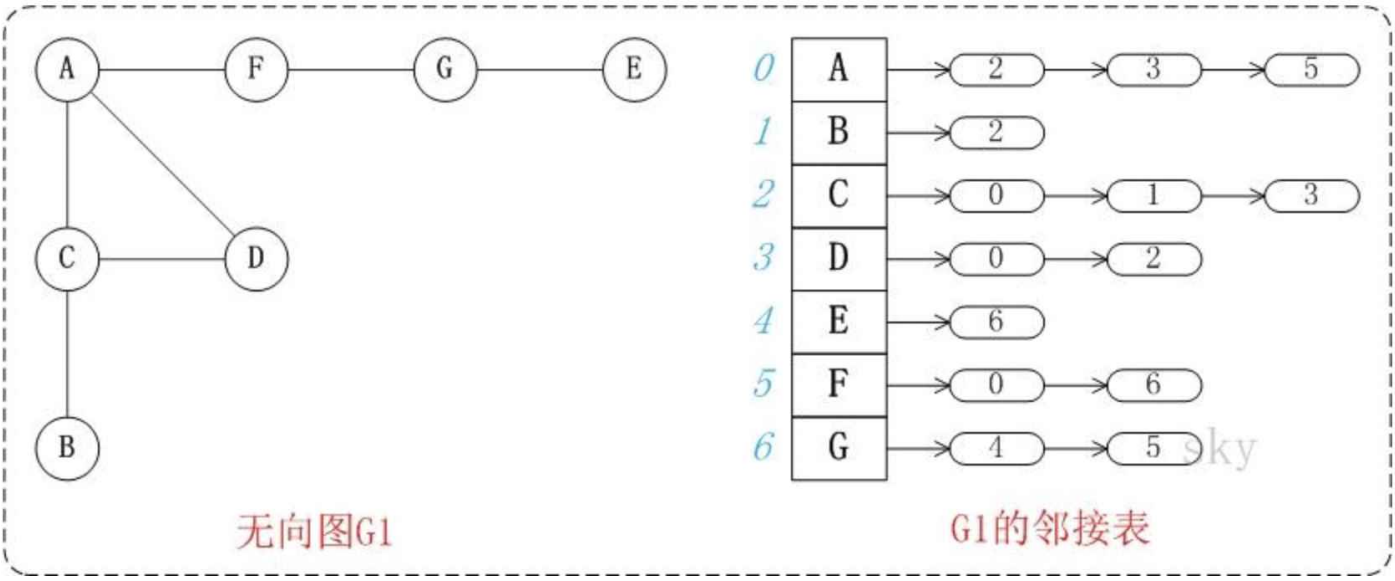
9.1.1. 邻接矩阵

通过使用一个二维数组 `G[N][N]` 进行表示 `N` 个点到 `N-1` 编号，通过邻接矩阵可以立刻看出两顶点之间是否存在一条边，只需要检查邻接矩阵行 `i` 和列 `j` 是否是非零值，对于无向图，邻接矩阵是对称的



9.1.2. 邻接表

存储方式如下图所示：



在 `javascript` 中，可以使用 `Object` 进行表示，如下：

```
const graph = {
  A: [2, 3, 5],
  B: [2],
  C: [0, 1, 3],
  D: [0, 2],
  E: [6],
  F: [0, 6],
  G: [4, 5]
}
```

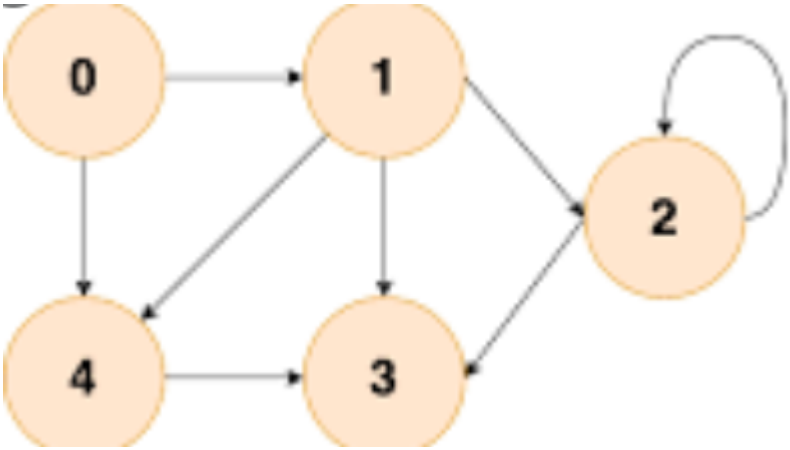
图的数据结构还可能包含和每条边相关联的数值（edge value），例如一个标号或一个数值（即权重，weight；表示花费、容量、长度等）

9.2. 操作

关于的图的操作常见的有：

- 深度优先遍历
- 广度优先遍历

首先构建一个图的邻接矩阵表示，如下面的图：



用代码表示则如下：

```
const graph = {
  0: [1, 4],
  1: [2, 4],
  2: [2, 3],
  3: [],
  4: [3],
}
```

9.2.1. 深度优先遍历

也就是尽可能的往深处的搜索图的分支

实现思路是，首先应该确定一个根节点，然后对根节点的没访问过的相邻节点进行深度优先遍历

确定以 0 为根节点，然后进行深度遍历，然后遍历1，接着遍历 2，然后3，此时完成一条分支 `0 - 1- 2- 3` 的遍历，换一条分支，也就是4，4后面因为3已经遍历过了，所以就不访问了

用代码表示则如下：

```
const visited = new Set()
const dfs = (n) => {
  console.log(n)
  visited.add(n) // 访问过添加记录
  graph[n].forEach(c => {
    if(!visited.has(c)){ // 判断是否访问呢过
      dfs(c)
    }
  })
}
```

9.2.2. 广度优先遍历

先访问离根节点最近的节点，然后进行入队操作，解决思路如下：

- 新建一个队列，把根节点入队
- 把队头出队并访问
- 把队头的没访问过的相邻节点入队
- 重复二、三步骤，知道队列为空

用代码标识则如下：

```
const visited = new Set()
const dfs = (n) => {
  visited.add(n)
  const q = [n]
  while(q.length){
    const n = q.shift()
    console.log(n)
    graph[n].forEach(c => {
      if(!visited.has(c)){
        q.push(c)

        visited.add(c)
      }
    })
  }
}
```

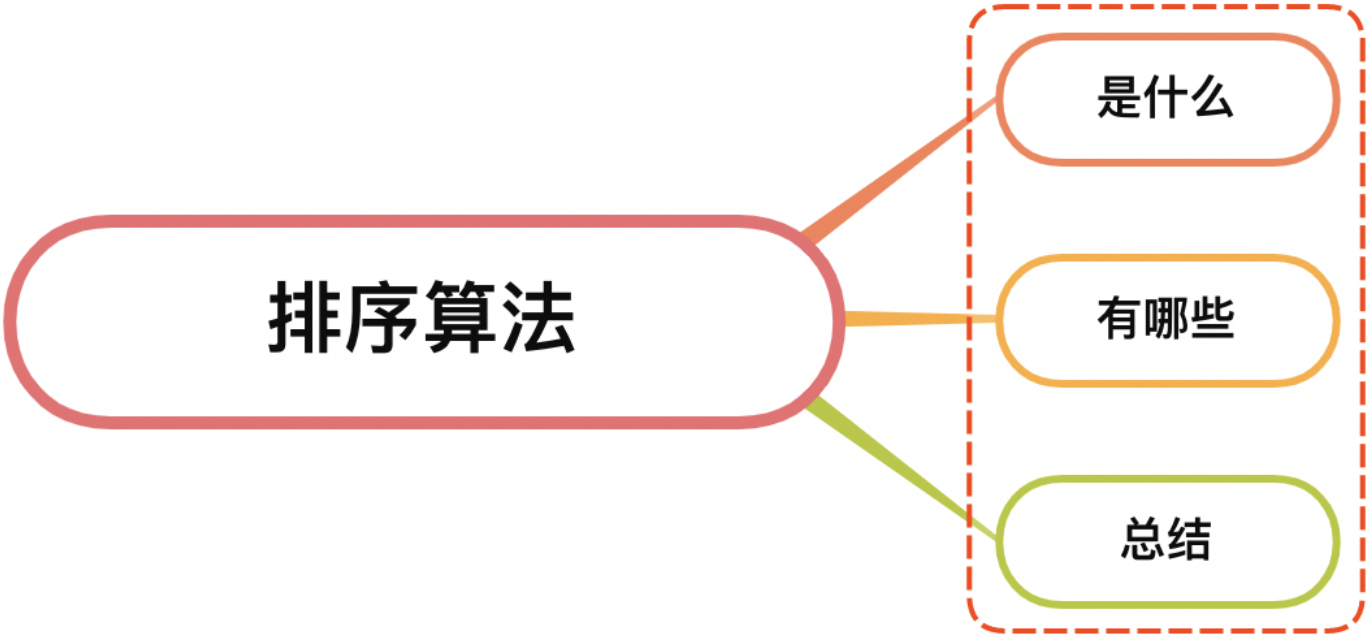
9.3. 总结

通过上面的初步了解，可以看到图就是由顶点的有穷非空集合和顶点之间的边组成的集合，分成了无向图与有向图

图的表达形式可以分成邻接矩阵和邻接表两种形式，在 `javascript` 中，则可以通过二维数组和对象的形式进行表达

图实际是很复杂的，后续还可以延伸出无向图和带权图，对应如下图所示：

10. 说说常见的排序算法有哪些？区别？



10.1. 是什么

排序是程序开发中非常常见的操作，对一组任意的数据元素经过排序操作后，就可以把他们变成一组一定规则排序的有序序列

排序算法属于算法中的一种，而且是覆盖范围极小的一种，彻底掌握排序算法对程序开发是有很大的帮助的

对与排序算法的好坏衡量，主要是从时间复杂度、空间复杂度、稳定性

时间复杂度、空间复杂度前面已经讲过，这里主要看看稳定性的定义

稳定性指的是假定在待排序的记录序列中，存在多个具有相同的关键字的记录，若经过排序，这些记录的相对次序保持不变

即在原序列中，r[i] = r[j]，且 r[i] 在 r[j] 之前，而在排序后的序列中，r[i] 仍在 r[j] 之前，则称这种排序算法是稳定的；否则称为不稳定的

10.2. 有哪些

常见的算法排序算法有：

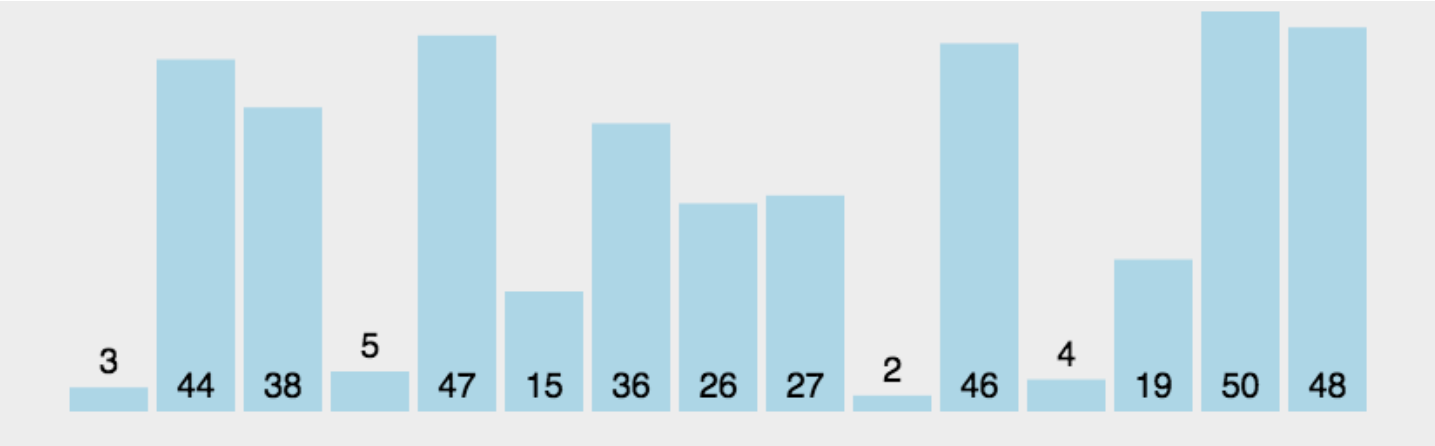
- 冒泡排序
- 选择排序
- 插入排序
- 归并排序
- 快速排序

10.2.1. 冒泡排序

一种简单直观的排序算法。它重复地走访过要排序的数列，一次比较两个元素，如果他们的顺序错误就把他们交换过来

思路如下：

- 比较相邻的元素，如果第一个比第二个大，就交换它们两个
- 对每一对相邻元素作同样的工作，从开始第一对到结尾的最后一对，这样在最后的元素应该会是最大的数
- 针对所有的元素重复以上的步骤，除了最后一个
- 重复上述步骤，直到没有任何一堆数字需要比较



10.2.2. 选择排序

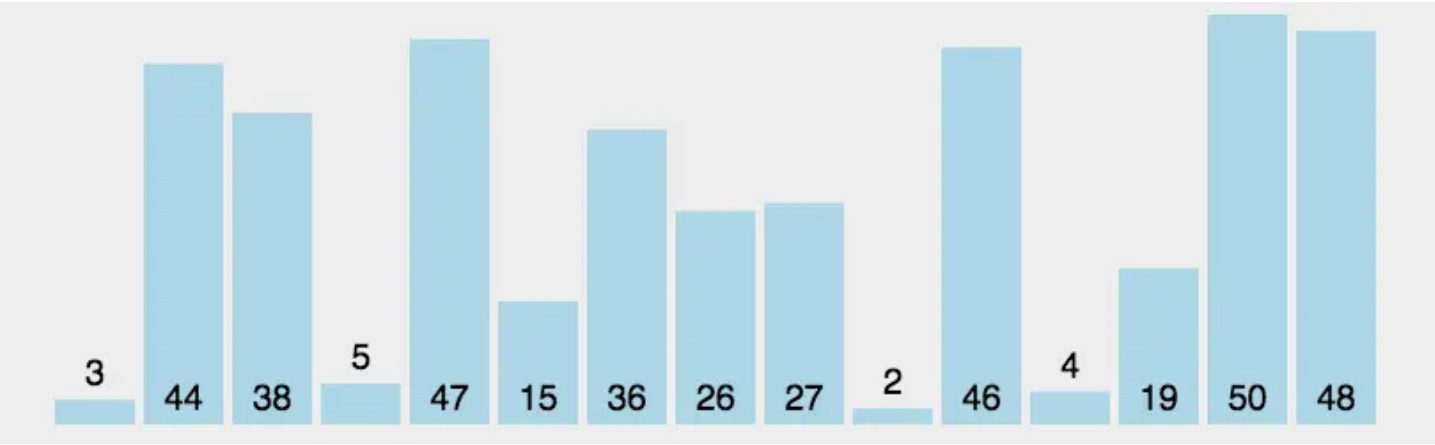
选择排序是一种简单直观的排序算法，它也是一种交换排序算法

无论什么数据进去都是 `O(n²)` 的时间复杂度。所以用到它的时候，数据规模越小越好

唯一的好处是不占用额外的内存存储空间

思路如下：

- 在未排序序列中找到最小（大）元素，存放到排序序列的起始位置
- 从剩余未排序元素中继续寻找最小（大）元素，然后放到已排序序列的末尾。
- 重复第二步，直到所有元素均排序完毕



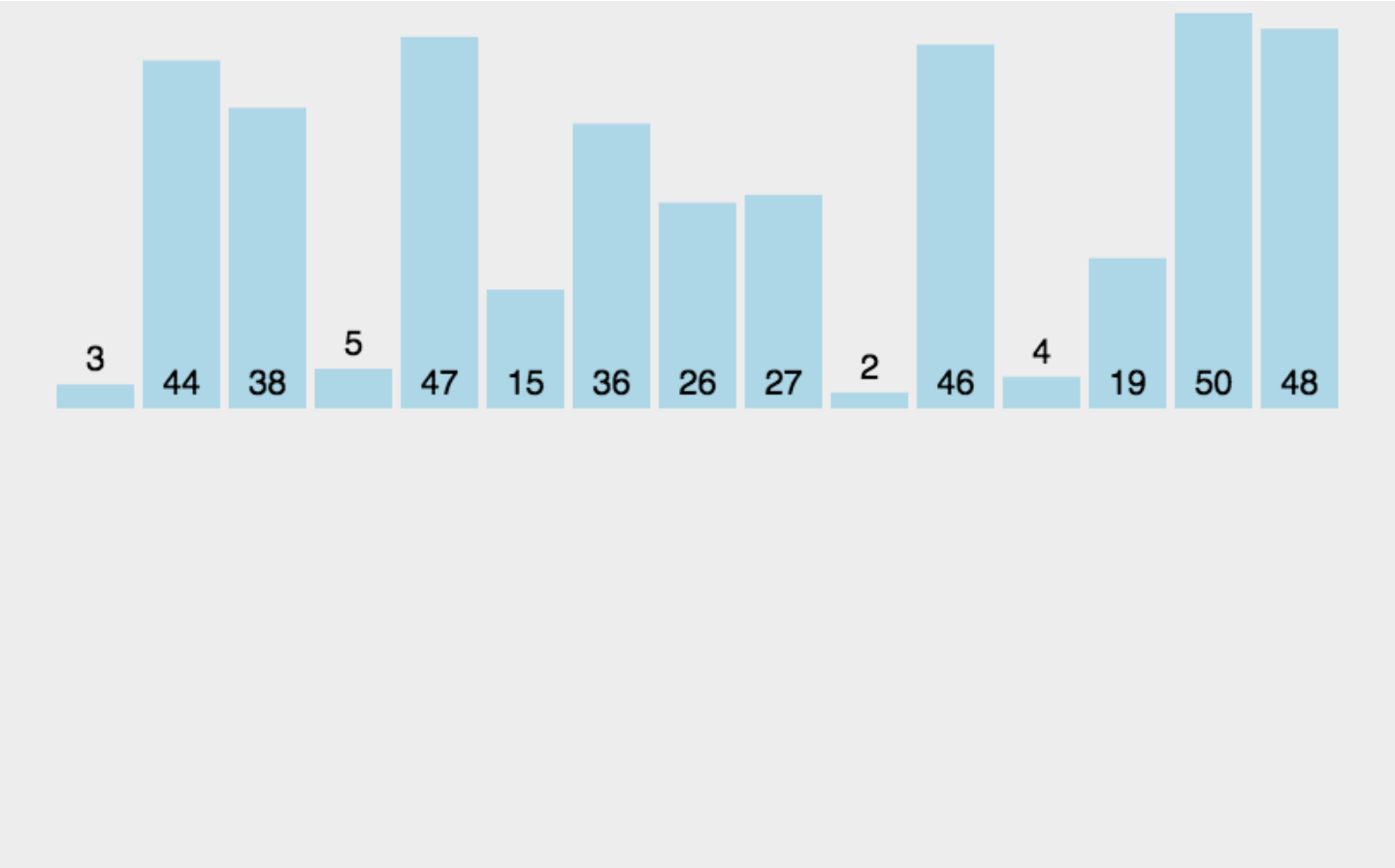
10.2.3. 插入排序

插入排序是一种简单直观的排序算法

它的工作原理是通过构建有序序列，对于未排序数据，在已排序序列中从后向前扫描，找到相应位置并插入

解决思路如下：

- 把待排序的数组分成已排序和未排序两部分，初始的时候把第一个元素认为是已排好序的
- 从第二个元素开始，在已排好序的子数组中寻找到该元素合适的位置并插入该位置（如果待插入的元素与有序序列中的某个元素相等，则将待插入元素插入到相等元素的后面。）
- 重复上述过程直到最后一个元素被插入有序子数组中



10.2.4. 归并排序

归并排序是建立在归并操作上的一种有效的排序算法

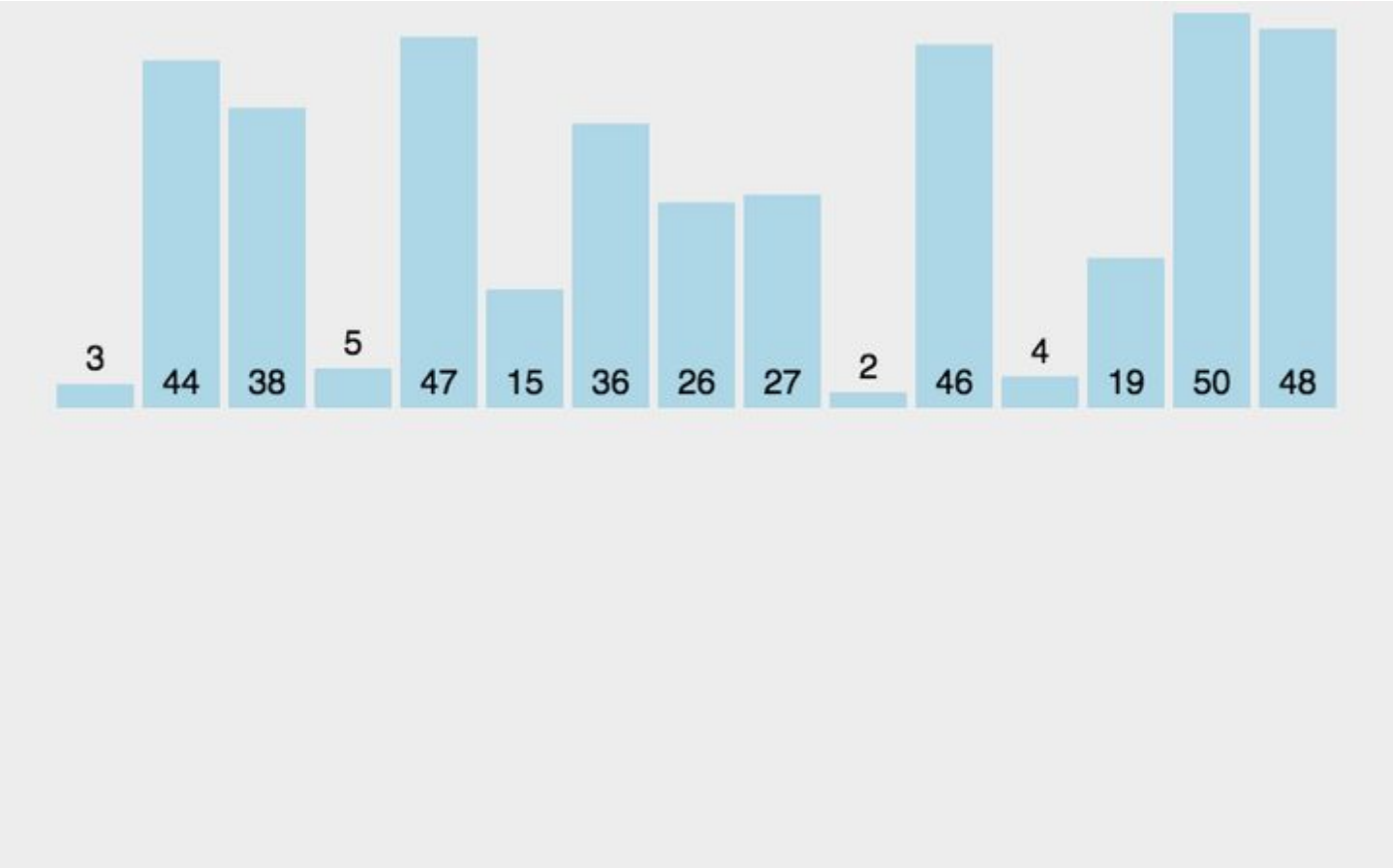
该算法是采用分治法的一个非常典型的应用

将已有序的子序列合并，得到完全有序的序列，即先使每个子序列有序，再使子序列段间有序

解决思路如下：

- 申请空间，使其大小为两个已经排序序列之和，该空间用来存放合并后的序列
- 设定两个指针，最初位置分别为两个已经排序序列的起始位置

- 比较两个指针所指向的元素，选择相对小的元素放入到合并空间，并移动指针到下一位置
- 重复步骤3直到某一指针到达序列尾
- 将另一序列剩下的所有元素直接复制到合并序列尾



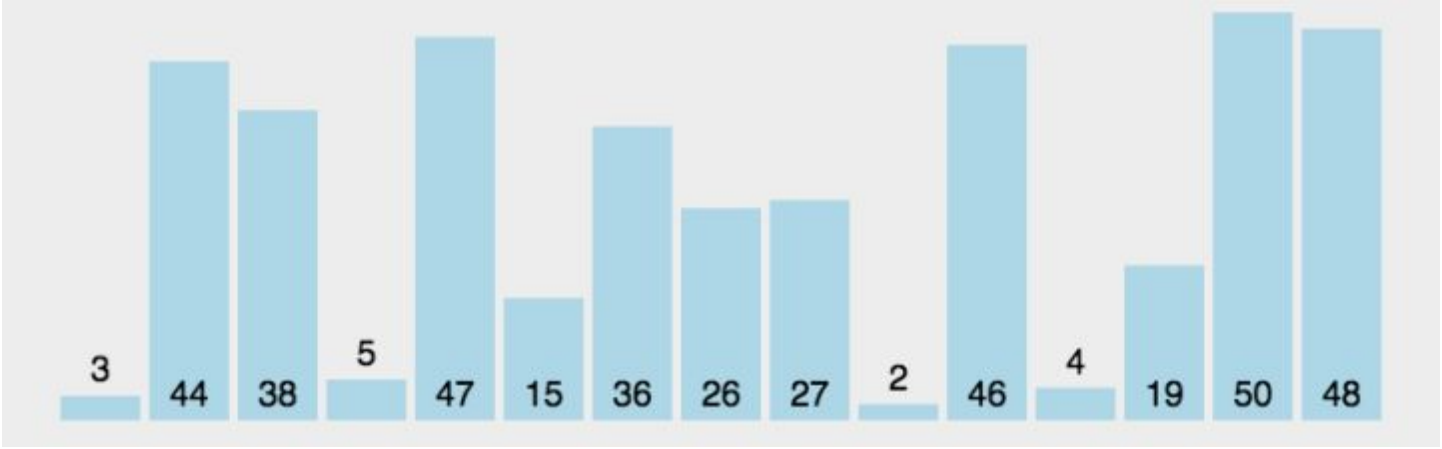
10.2.5. 快速排序

快速排序是对冒泡排序算法的一种改进，基本思想是通过一趟排序将要排序的数据分割成独立的两部分，其中一部分的所有数据比另一部分的所有数据要小

再按这种方法对这两部分数据分别进行快速排序，整个排序过程可以递归进行，使整个数据变成有序序列

解决思路如下：

- 从数列中挑出一个元素，称为"基准"（pivot）
- 重新排序数列，所有比基准值小的元素摆放在基准前面，所有比基准值大的元素摆在基准后面（相同的数可以到任何一边）。在这个分区结束之后，该基准就处于数列的中间位置。这个称为分区（partition）操作
- 递归地（recursively）把小于基准值元素的子数列和大于基准值元素的子数列排序



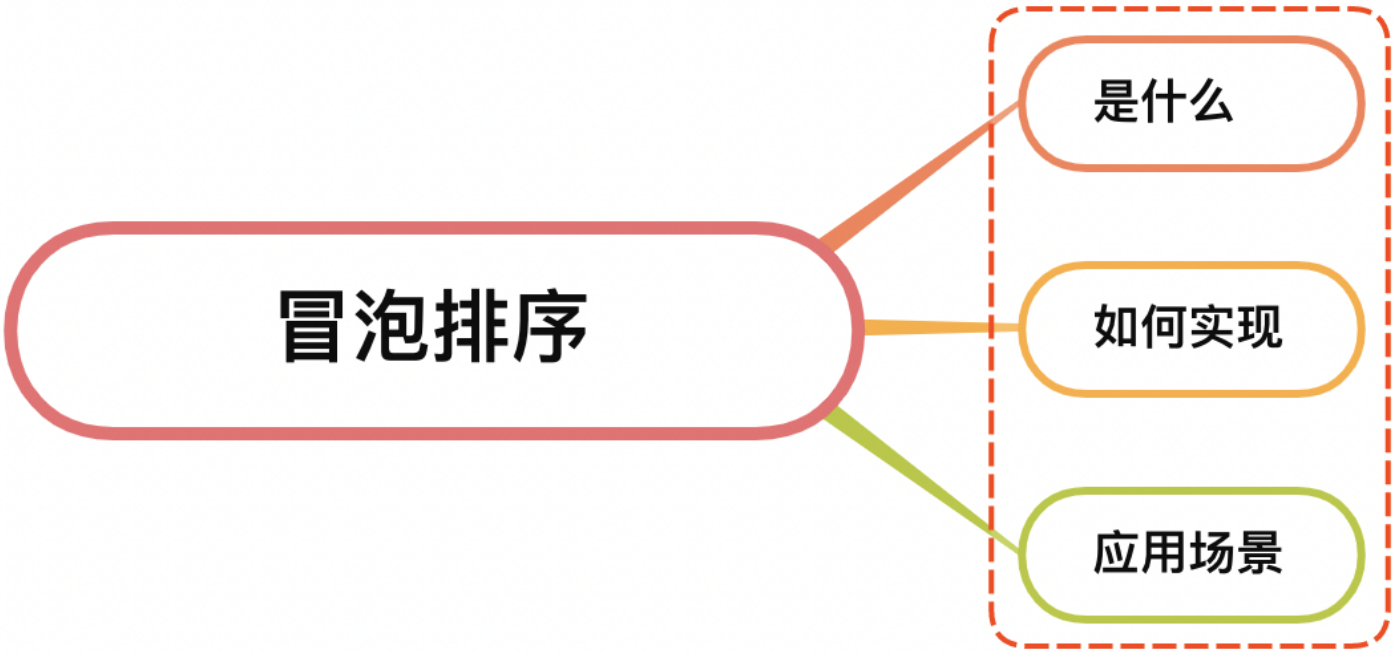
10.3. 区别

除了上述的排序算法之外，还存在其他的排序算法，例如希尔排序、堆排序等等.....

区别如下图所示：

排序算法	平均时间复杂度	最好情况	最坏情况	空间复杂度	排序方式	稳定性
冒泡排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	In-place	稳定
选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	In-place	不稳定
插入排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	In-place	稳定
希尔排序	$O(n \log n)$	$O(n \log^2 n)$	$O(n \log^2 n)$	$O(1)$	In-place	不稳定
归并排序	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$	Out-place	稳定
快速排序	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(\log n)$	In-place	不稳定
堆排序	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(1)$	In-place	不稳定
计数排序	$O(n + k)$	$O(n + k)$	$O(n + k)$	$O(k)$	Out-place	稳定
桶排序	$O(n + k)$	$O(n + k)$	$O(n^2)$	$O(n + k)$	Out-place	稳定
基数排序	$O(n \times k)$	$O(n \times k)$	$O(n \times k)$	$O(n + k)$	Out-place	稳定

11. 说说你对冒泡排序的理解？如何实现？应用场景？



11.1. 是什么

冒泡排序（Bubble Sort），是一种计算机科学领域的较简单的排序算法

冒泡排序的思想就是在每次遍历一遍未排序的数列之后，将一个数据元素浮上去（也就是排好了一个数据）

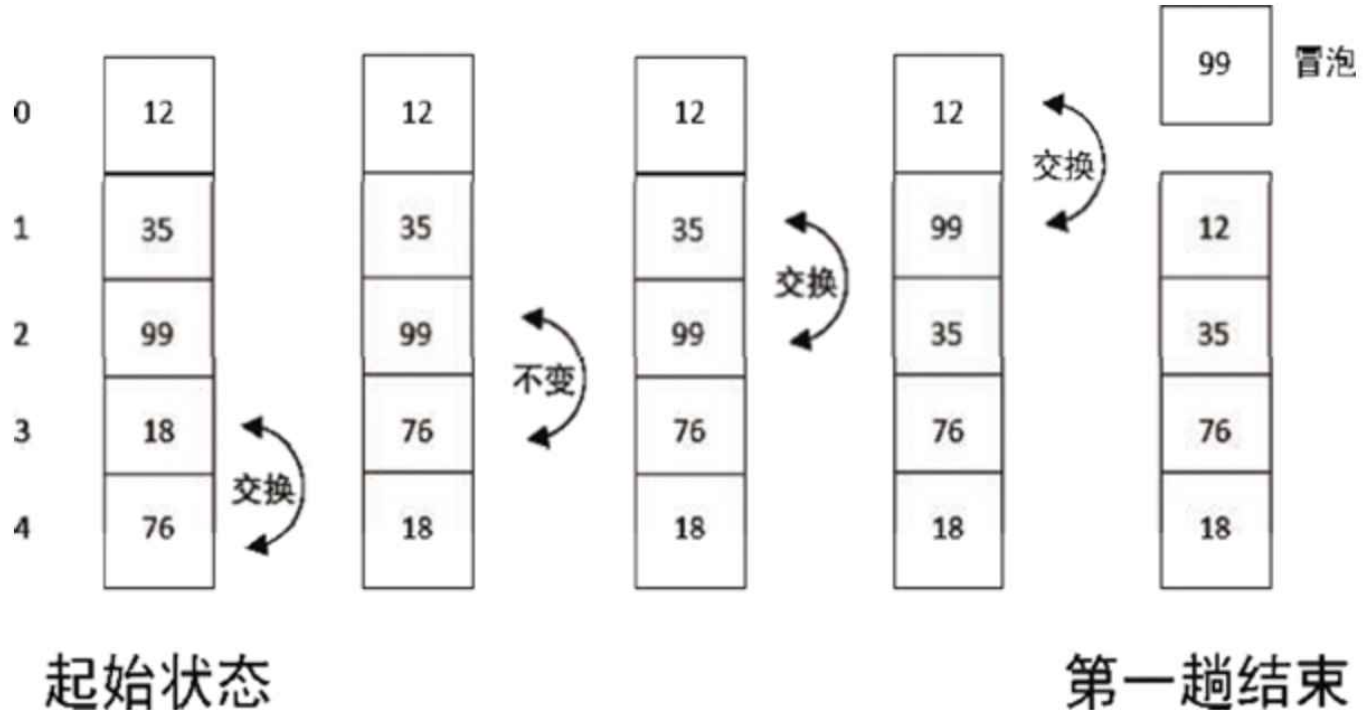
如同碳酸饮料中二氧化碳的气泡最终会上浮到顶端一样，故名“冒泡排序”

假如我们要把 12、35、99、18、76 这 5 个数从大到小进行排序，那么数越大，越需要把它放在前面

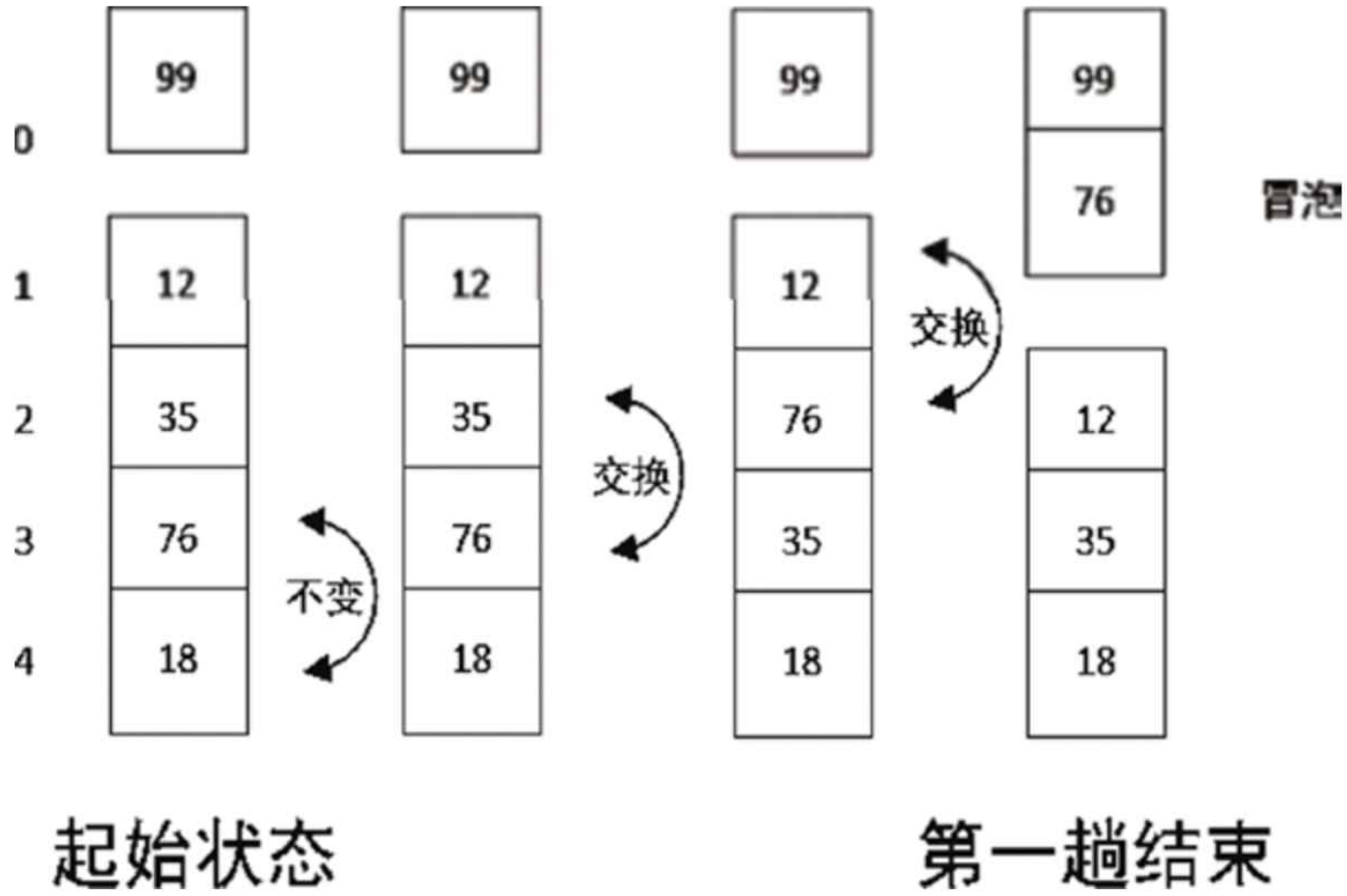
思路如下：

- 从后开始遍历，首先比较 18 和 76，发现 76 比 18 大，就把两个数交换顺序，得到 12、35、99、76、18
- 接着比较 76 和 99，发现 76 比 99 小，所以不用交换顺序
- 接着比较 99 和 35，发现 99 比 35 大，交换顺序
- 接着比较 99 和 12，发现 99 比 12 大，交换顺序

最终第 1 趟排序的结果变成了 99、12、35、76、18，如下图所示：



上述可以看到，经过第一趟的排序，可以得到最大的元素，接下来第二趟排序则对剩下的的4个元素进行排序，如下图所示：



经过第 2 趟排序，结果为 99、76、12、35、18

然后开始第3趟的排序，结果为99、76、35、12、18

然后第四趟排序结果为99、76、35、18、12

经过 4 趟排序之后，只剩一个 12 需要排序了，这时已经没有可比较的元素了，这时排序完成

11.2. 如何实现

如果要实现一个从小到大的排序，算法原理如下：

- 首先比较相邻的元素，如果第一个元素比第二个元素大，则交换它们
- 针对每一对相邻元素做同样的工作，从开始第一对到结尾的最后一对，这样，最后的元素回事最大的数
- 针对所有的元素重复以上的步骤，除了最后一个
- 持续每次对越来越少的元素重复上面的步骤，直到没有任何一对数字需要比较

用代码表示则如下：

```
function bubbleSort(arr) {
    const len = arr.length;
    for (let i = 0; i < len - 1; i++) {
        for (let j = 0; j < len - 1 - i; j++) {
            if (arr[j] > arr[j+1]) {        // 相邻元素两两对比
                var temp = arr[j+1];        // 元素交换
                arr[j+1] = arr[j];
                arr[j] = temp;
            }
        }
    }
    return arr;
}
```

可以看到：冒泡排序在每一轮排序中都会使一个元素排到一趟， 也就是最终需要 n-1 轮这样的排序

而在每轮排序中都需要对相邻的两个元素进行比较，在最坏的情况下，每次比较之后都需要交换位置，此时时间复杂度为 `O(n^2)`

11.2.1. 优化

对冒泡排序常见的改进方法是加入一标志性变量 `exchange` ，用于标志某一趟排序过程中是否有数据交换

如果进行某一趟排序时并没有进行数据交换，则说明数据已经按要求排列好，可立即结束排序，避免不必要的比较过程

可以设置一标志性变量 `pos` ，用于记录每趟排序中最后一次进行交换的位置，由于 `pos` 位置之后的记录均已交换到位，故在进行下一趟排序时只要扫描到 `pos` 位置即可，如下：

```
function bubbleSort1(arr){
 const i=arr.length-1;//初始时,最后位置保持不变

 while(i>0){
  let pos = 0;//每趟开始时,无记录交换
  for(let j = 0; j < i; j++){
   if(arr[j] > arr[j+1]){
        let tmp = arr[j];
        arr[j] = arr[j+1];
        arr[j+1] = tmp;
     pos = j;//记录最后交换的位置

   }

  }
  i = pos;//为下一趟排序作准备
 }
 return arr;
}
```

在待排序的数列有序的情况下，只需要一轮排序并且不用交换，此时情况最好，时间复杂度为 `O(n)`

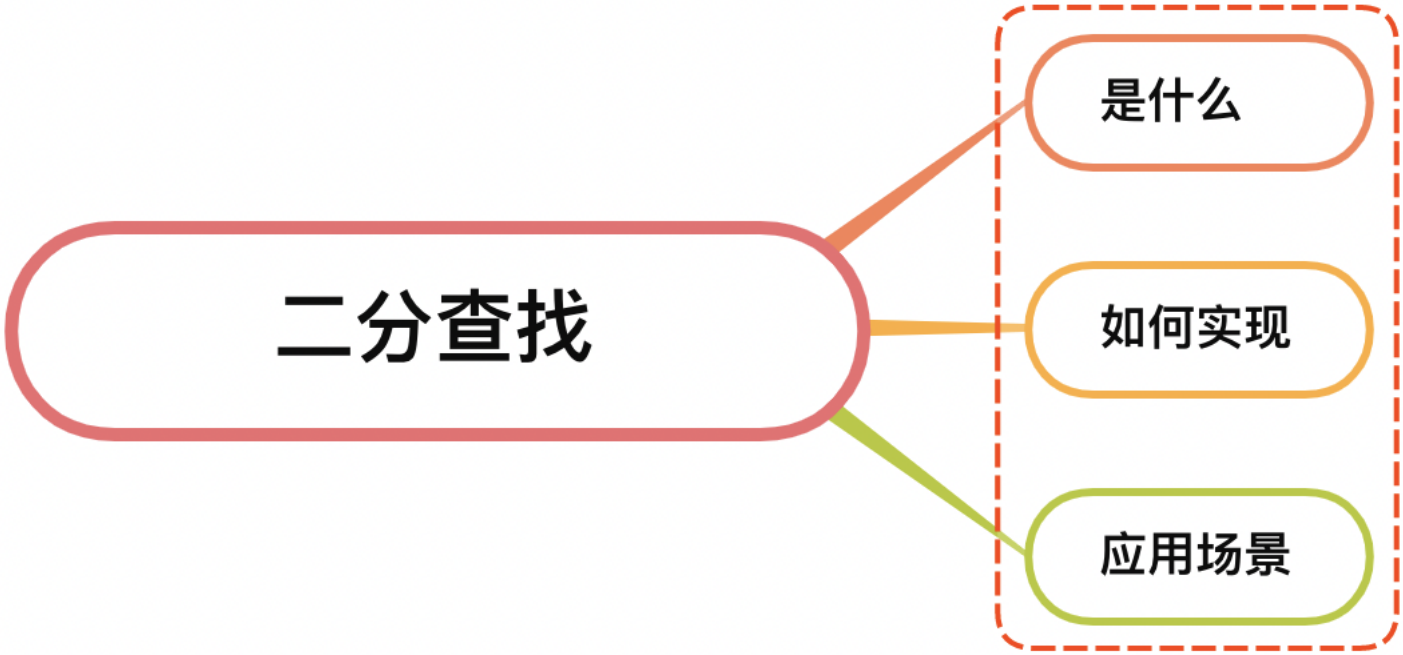
并且从上述比较中看到，只有后一个元素比前面的元素大（小）时才会对它们交换位置并向上冒出，对于同样大小的元素，是不需要交换位置的，所以对于同样大小的元素来说，相对位置是不会改变的，因此， 冒泡排序是稳定的

11.3. 应用场景

冒泡排的核心部分是双重嵌套循环，

时间复杂度是 O(N 2)，相比其它排序算法，这是一个相对较高的时间复杂度，一般情况不推荐使用，由于冒泡排序的简洁性，通常被用来对于程序设计入门的学生介绍算法的概念

12. 说说你对二分查找的理解？如何实现？应用场景？



12.1. 是什么

在计算机科学中，二分查找算法，也称折半搜索算法，是一种在有序数组中查找某一特定元素的搜索算法

想要应用二分查找法，则这一堆数应有如下特性：

- 存储在数组中
- 有序排序

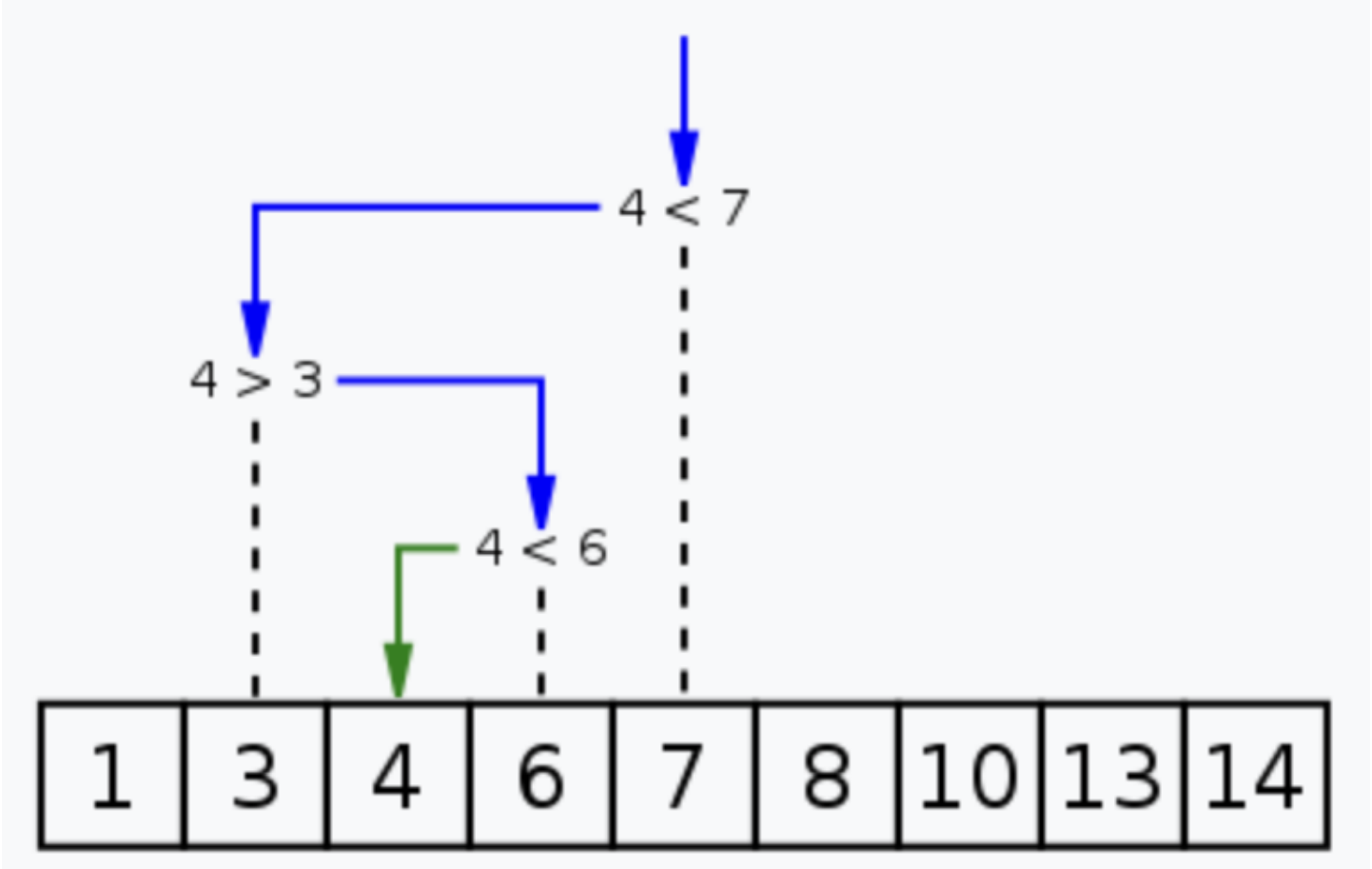
搜索过程从数组的中间元素开始，如果中间元素正好是要查找的元素，则搜索过程结束

如果某一特定元素大于或者小于中间元素，则在数组大于或小于中间元素的那一半中查找，而且跟开始一样从中间元素开始比较

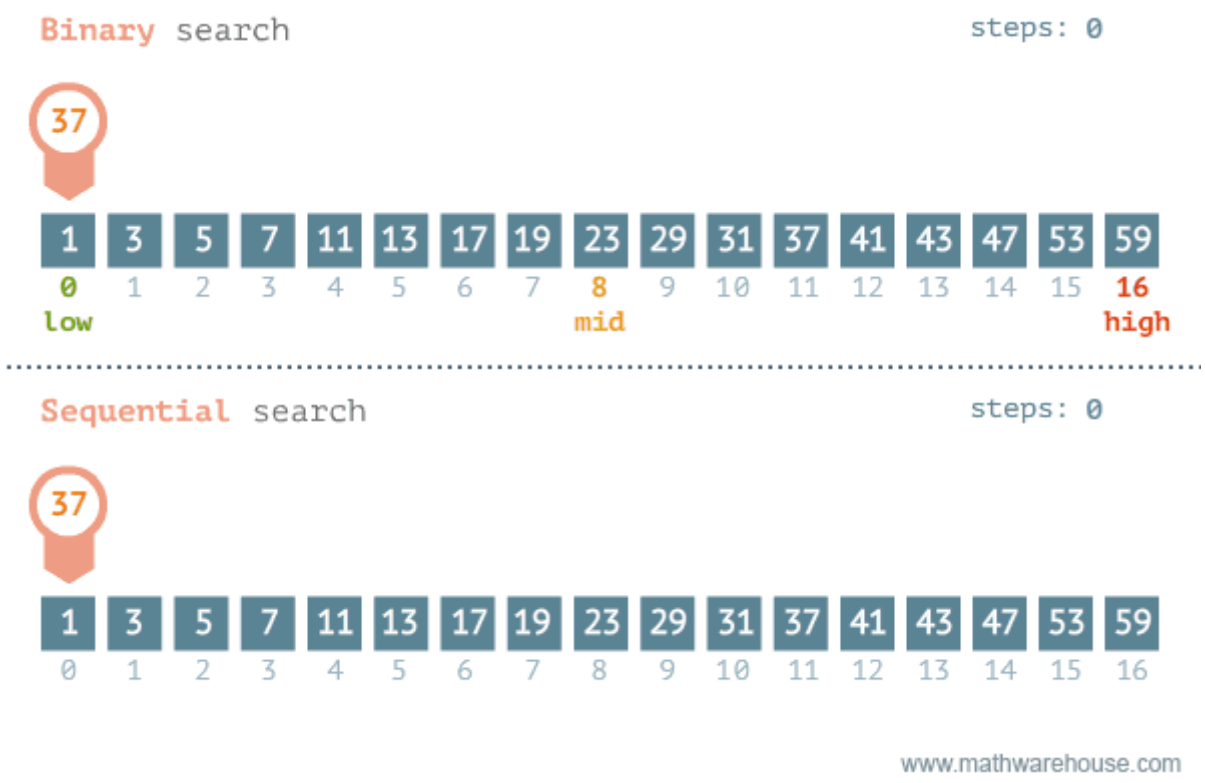
如果在某一步骤数组为空，则代表找不到

这种搜索算法每一次比较都使搜索范围缩小一半

如下图所示：



相比普通的顺序查找，除了数据量很少的情况下，二分查找会比顺序查找更快，区别如下所示：



12.2. 如何实现

基于二分查找的实现，如果数据是有序的，并且不存在重复项，实现代码如下：

```
function BinarySearch(arr, target) {
    if (arr.length <= 1) return -1
    // 低位下标
    let lowIndex = 0
    // 高位下标
    let highIndex = arr.length - 1
    while (lowIndex <= highIndex) {
        // 中间下标
        const midIndex = Math.floor((lowIndex + highIndex) / 2)
        if (target < arr[midIndex]) {
            highIndex = midIndex - 1
        } else if (target > arr[midIndex]) {
            lowIndex = midIndex + 1
        } else {
            // target === arr[midIndex]
            return midIndex
        }
    }
    return -1
}
```

如果数组中存在重复项，而我们需要找出第一个制定的值，实现则如下：

```
function BinarySearchFirst(arr, target) {
    if (arr.length <= 1) return -1
    // 低位下标
    let lowIndex = 0
    // 高位下标
    let highIndex = arr.length - 1
    while (lowIndex <= highIndex) {
        // 中间下标
        const midIndex = Math.floor((lowIndex + highIndex) / 2)
        if (target < arr[midIndex]) {
            highIndex = midIndex - 1
        } else if (target > arr[midIndex]) {
            lowIndex = midIndex + 1
        } else {
            // 当 target 与 arr[midIndex] 相等的时候，如果 midIndex 为0或者前一个数比 target 小那么就找到了第一个等于给定值的元素，直接返回
            if (midIndex === 0 || arr[midIndex - 1] < target) return midIndex
            // 否则高位下标为中间下标减1，继续查找
            highIndex = midIndex - 1
        }
    }
    return -1
```

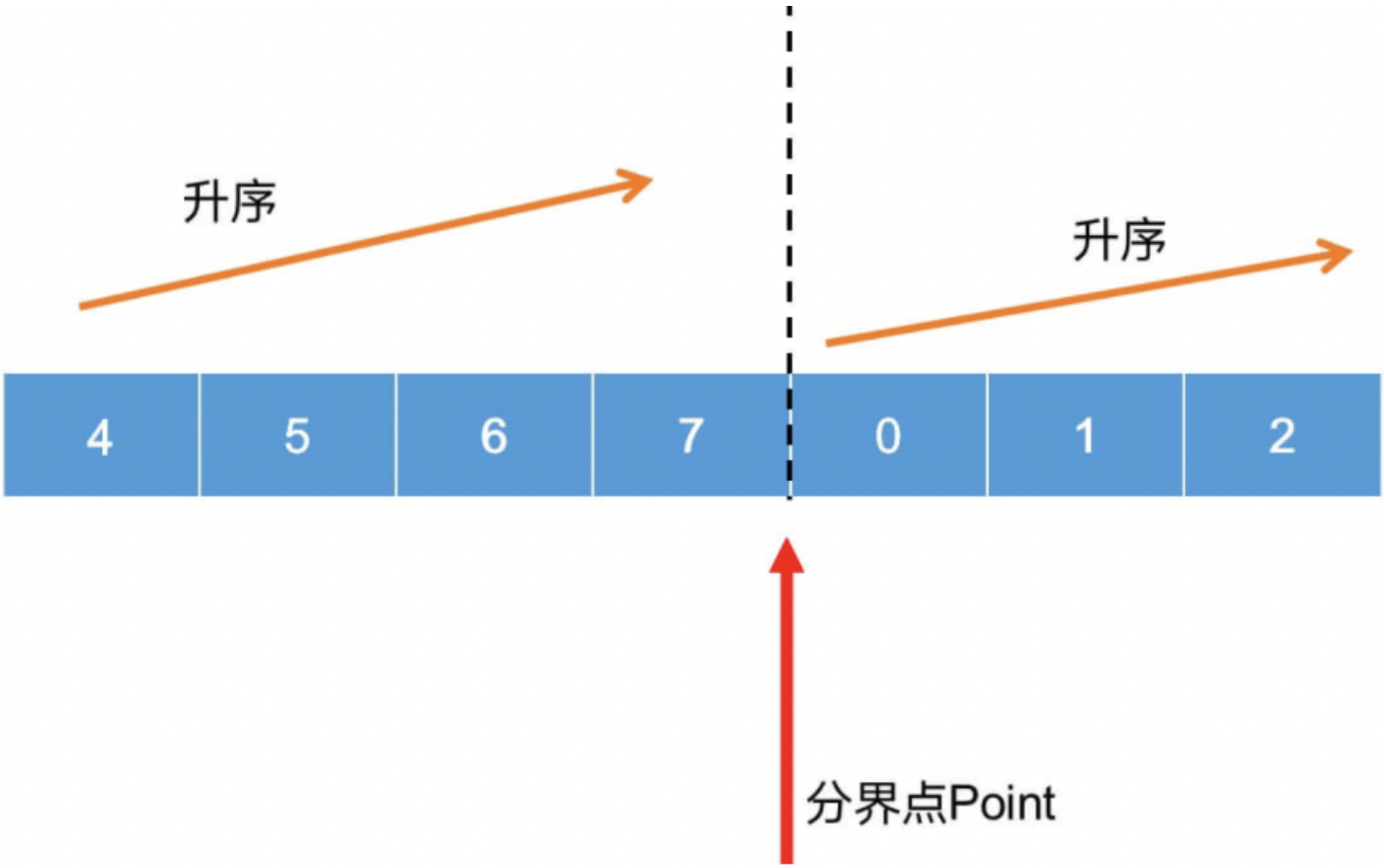
```
22 }
```

实际上，除了有序的数组可以使用，还有一种特殊的数组可以应用，那就是轮转后的有序数组

有序数组即一个有序数字以某一个数为轴，将其之前的所有数都轮转到数组的末尾所得

例如，[4, 5, 6, 7, 0, 1, 2]就是一个轮转后的有序数组

该数组的特性是存在一个分界点用来分界两个有序数组，如下：



分界点有如下特性：

- 分界点元素 >= 第一个元素
- 分界点元素 < 第一个元素

代码实现如下：

```
1  function search (nums, target) {
2    // 如果为空或者是空数组的情况
3    if (nums == null || !nums.length) {
4      return -1;
5    }
6    // 搜索区间是前闭后闭
7    let begin = 0,
8      end = nums.length - 1;
9    while (begin <= end) {
```

```
    // 下面这样写是考虑大数情况下避免溢出
    let mid = begin + ((end - begin) >> 1);
    if (nums[mid] == target) {
      return mid;
    }
    // 如果左边是有序的
    if (nums[begin] <= nums[mid]) {
      //同时target在[ nums[begin],nums[mid] ]中，那么就在这段有序区间查找
      if (nums[begin] <= target && target <= nums[mid]) {
        end = mid - 1;
      } else {
        //否则去反方向查找
        begin = mid + 1;
      }
      //如果右侧是有序的
    } else {
      //同时target在[ nums[mid],nums[end] ]中，那么就在这段有序区间查找
      if (nums[mid] <= target && target <= nums[end]) {
        begin = mid + 1;
      } else {
        end = mid - 1;
      }
    }
  }
  return -1;
};
```

对比普通的二分查找法，为了确定目标数会落在二分后的哪个部分，我们需要更多的判定条件

12.3. 应用场景

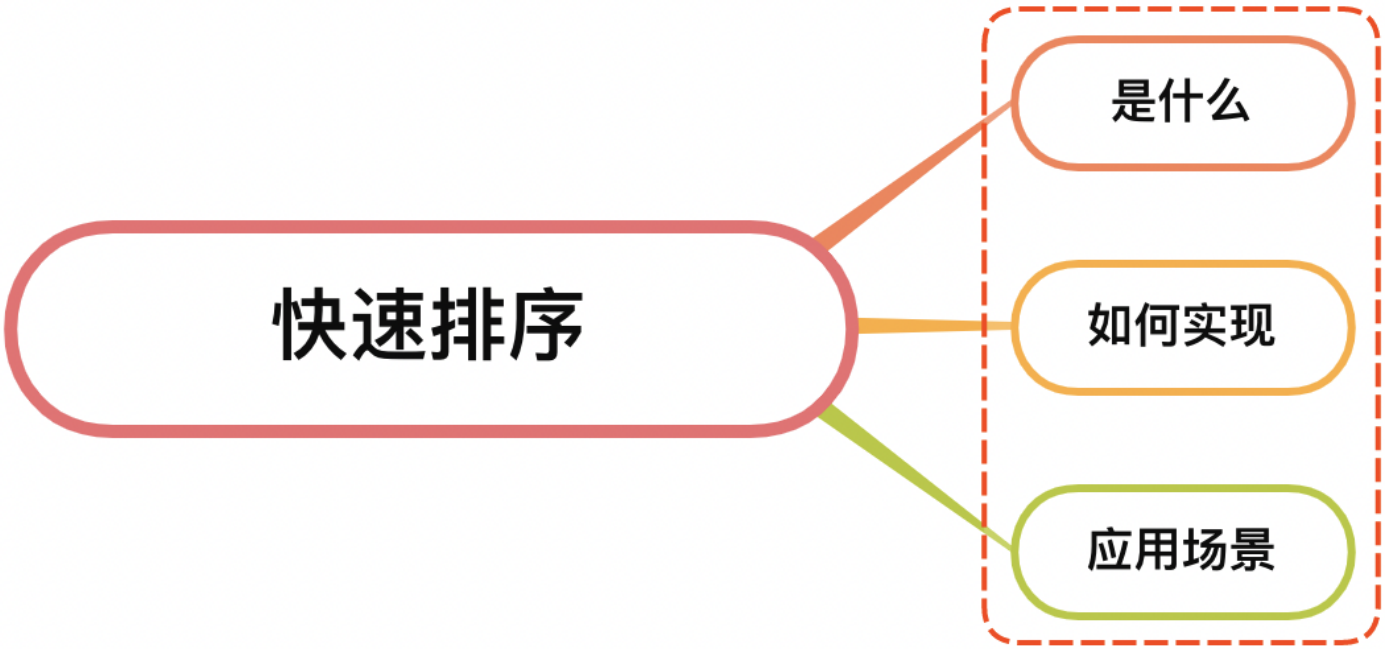
二分查找法的`O(logn)`让它成为十分高效的算法。不过它的缺陷却也是比较明显，就在它的限定之上：

- 有序：我们很难保证我们的数组都是有序的
- 数组：数组读取效率是O(1)，可是它的插入和删除某个元素的效率却是O(n)，并且数组的存储是需要连续的内存空间，不适合大数据的情况

关于二分查找的应用场景，主要如下：

- 不适合数据量太小的数列；数列太小，直接顺序遍历说不定更快，也更简单
- 每次元素与元素的比较是比较耗时的，这个比较操作耗时占整个遍历算法时间的大部分，那么使用二分查找就能有效减少元素比较的次数
- 不适合数据量太大的数列，二分查找作用的数据结构是顺序表，也就是数组，数组是需要连续的内存空间的，系统并不一定有这么大的连续内存空间可以使用

13. 说说你对快速排序的理解？如何实现？应用场景？



13.1. 是什么

快速排序（Quick Sort）算法是在冒泡排序的基础上进行改进的一种算法，从名字上看就知道该排序算法的特点是快、效率高，是处理大数据最快的排序算法之一

实现的基本思想是：通过一次排序将整个无序表分成相互独立的两部分，其中一部分中的数据都比另一部分中包含的数据的值小

然后继续沿用此方法分别对两部分进行同样的操作，直到每一个小部分不可再分，所得到的整个序列就变成有序序列

例如，对无序表49，38，65，97，76，13，27，49进行快速排序，大致过程为：

- 首先从表中选取一个记录的关键字作为分割点（称为“枢轴”或者支点，一般选择第一个关键字），例如选取 49
- 将表格中大于 49 个放置于 49 的右侧，小于 49 的放置于 49 的左侧，假设完成后的无序表为：{27，38，13，49，65，97，76，49}
- 以 49 为支点，将整个无序表分割成了两个部分，分别为{27，38，13}和{65，97，76，49}，继续采用此种方法分别对两个子表进行排序
- 前部分子表以 27 为支点，排序后的子表为{13，27，38}，此部分已经有序；后部分子表以 65 为支点，排序后的子表为{49，65，97，76}
- 此时前半部分子表中的数据已完成排序；后部分子表继续以 65 为支点，将其分割为{49}和{97，76}，前者不需排序，后者排序后的结果为{76， 97}
- 通过以上几步的排序，最后由子表{13，27，38}、{49}、{49}、{65}、{76，97}构成有序表：{13，27，38，49，49，65，76，97}

13.2. 如何实现

可以分成以下步骤：

- 分区：从数组中选择任意一个基准，所有比基准小的元素放在基准的左边，比基准大的元素放到基准的右边
- 递归：递归地对基准前后的子数组进行分区

用代码表示则如下：

```
function quickSort (arr) {
  const rec = (arr) => {
    if (arr.length <= 1) { return arr; }
    const left = [];
    const right = [];
    const mid = arr[0]; // 基准元素
    for (let i = 1; i < arr.length; i++){
      if (arr[i] < mid) {
        left.push(arr[i]);
      } else {
        right.push(arr[i]);
      }
    }
    return [...rec(left), mid, ...rec(right)]
  }
  return res(arr)
};
```

快速排序是冒泡排序的升级版，最坏情况下每一次基准元素都是数组中最小或者最大的元素，则快速排序就是冒泡排序

这种情况时间复杂度就是冒泡排序的时间复杂度：`T[n] = n * (n-1) = n^2 + n`，也就是`O(n^2)`

最好情况下是`O(nlogn)`，其中递归算法的时间复杂度公式：`T[n] = aT[n/b] + f(n)`，推导如下所示：

$$T(n)=2T(\frac{n}{2})+n$$

$$T(n)=2(2T(\frac{n}{4})+\frac{n}{2})+n=4T(\frac{n}{4})+2n$$

$$T(n)=4(2T(\frac{n}{8})+\frac{n}{4})+2n=8T(\frac{n}{8})+3n$$

.....

且 $T(1)=0$

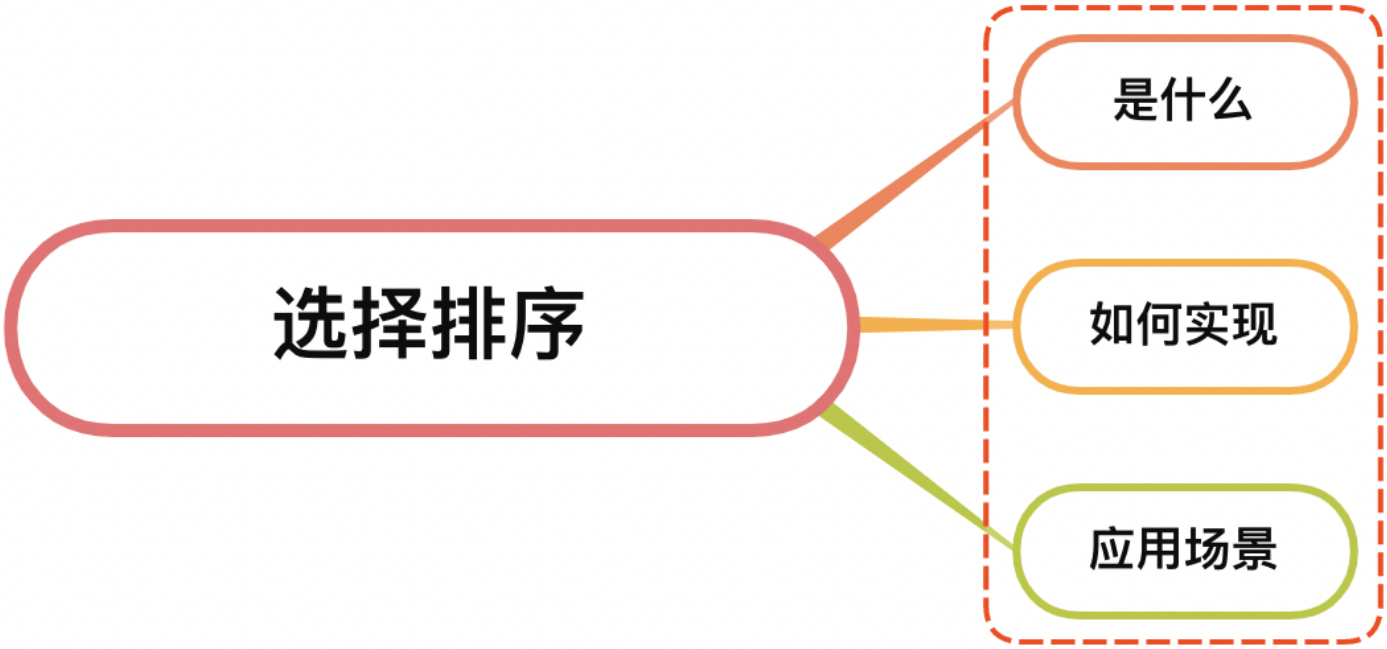
$$T(n)=nT(1)+(\log(n))\times n=O(n\times\log(n))$$

关于上述代码实现的快速排序，可以看到是稳定的

13.3. 应用场景

快速排序时间复杂度为 `O(nlogn)` ，是目前基于比较的内部排序中被认为最好的方法，当数据过大且数据杂乱无章时，则适合采用快速排序

14. 说说你对选择排序的理解？如何实现？应用场景？



14.1. 是什么

选择排序（Selection sort）是一种简单直观的排序算法，无论什么数据进去都是 $O(n^2)$ 的时间复杂度，所以用到它的时候，数据规模越小越好

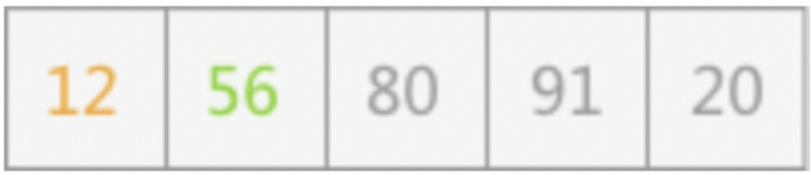
其基本思想是：首先在未排序的数列中找到最小(or最大)元素，然后将其存放到数列的起始位置

然后再从剩余未排序的元素中继续寻找最小(or最大)元素，然后放到已排序序列的末尾

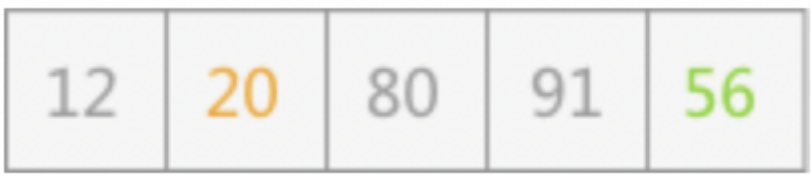
以此类推，直到所有元素均排序完毕

举个例子，一个数组为 56、12、80、91、29，其排序过程如下：

- 第一次遍历时，从下标为 1 的位置即 56 开始，找出关键字值最小的记录 12，同下标为 0 的关键字 56 交换位置。此时数组为 12、56、80、91、20



- 第二次遍历时，从下标为 2 的位置即 56 开始，找出最小值 20，同下标为 2 的关键字 56 互换位置，此时数组为12、20、80、91、56



- 第三次遍历时，从下标为 3 的位置即 80 开始，找出最小值 56，同下标为 3 的关键字 80 互换位置，此时数组为 12、20、56、91、80

- 第四次遍历时，从下标为 4 的位置即 91 开始，找出最小是 80，同下标为 4 的关键字 91 互换位置，此时排序完成，变成有序数组

14.2. 如何实现

从上面可以看到，对于具有 `n` 个记录的无序表遍历 `n-1` 次，第 `i` 次从无序表中第 `i` 个记录开始，找出后序关键字中最小的记录，然后放置在第 `i` 的位置上

直至到从第 `n` 个和第 `n-1` 个元素中选出最小的放在第 `n-1` 个位置

如下动画所示：

用代码表示则如下：

```
function selectionSort(arr) {
    var len = arr.length;
    var minIndex, temp;
    for (var i = 0; i < len - 1; i++) {
        minIndex = i;
        for (var j = i + 1; j < len; j++) {
            if (arr[j] < arr[minIndex]) {     // 寻找最小的数
                minIndex = j;                 // 将最小数的索引保存
            }
        }
        temp = arr[i];
        arr[i] = arr[minIndex];
        arr[minIndex] = temp;
    }
    return arr;
}
```

第一次内循环比较 `N - 1` 次，然后是 `N-2` 次， `N-3` 次，……，最后一次内循环比较1次
共比较的次数是 `(N - 1) + (N - 2) + ... + 1`，求等差数列和，得 `(N - 1 + 1)* N / 2 = N^2 / 2`，舍去最高项系数，其时间复杂度为 `O(N^2)`

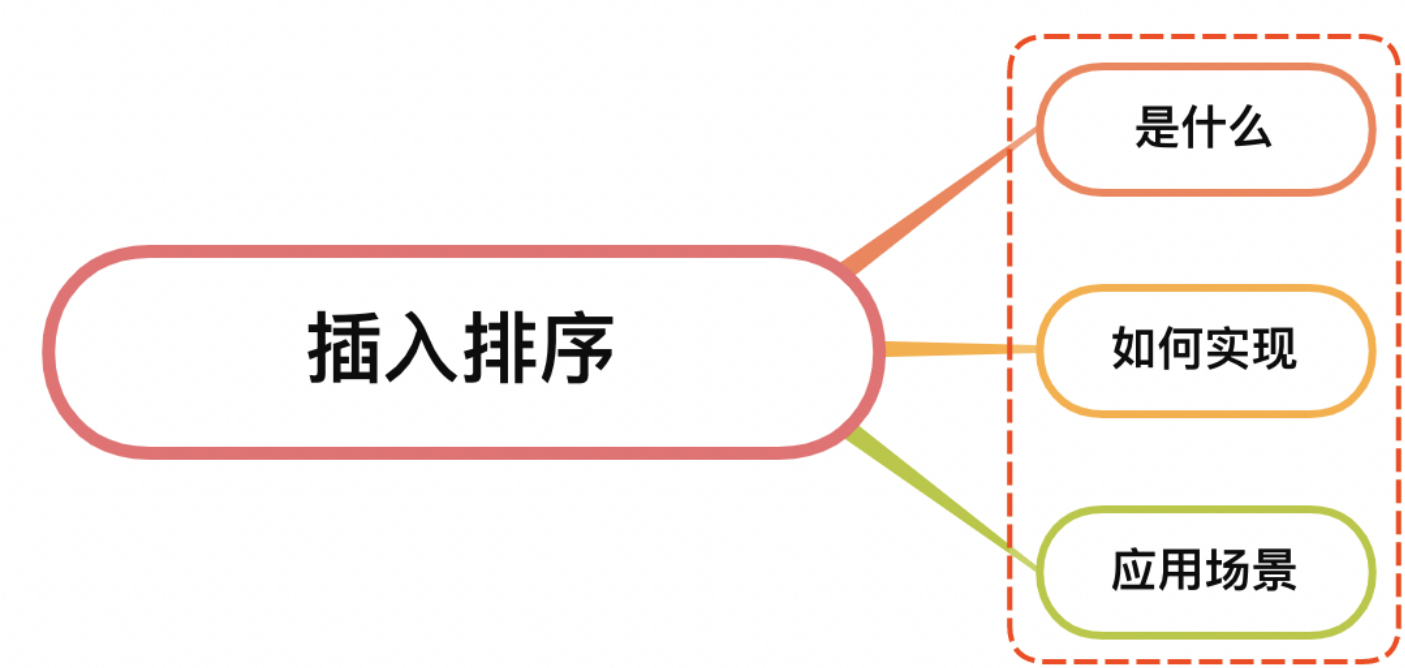
从上述也可以看到，选择排序是一种稳定的排序

14.3. 应用场景

和冒泡排序一致，相比其它排序算法，这也是一个相对较高的时间复杂度，一般情况不推荐使用

但是我们还是要掌握冒泡排序的思想及实现，这对于我们的算法思维是有很大帮助的

15. 说说你对插入排序的理解？如何实现？应用场景？



15.1. 是什么

插入排序（Insertion Sort），一般也被称为直接插入排序。对于少量元素的排序，它是一个有效、简单的算法

其主要的实现思想是将数据按照一定的顺序一个一个的插入到有序的表中，最终得到的序列就是已经排序好的数据

插入排序的工作方式像许多人排序一手扑克牌，开始时，我们的左手为空并且桌子上的牌面向下

然后，我们每次从桌子上拿走一张牌并将它插入左手中正确的位置，该正确位置需要从右到左将它与已在手中的每张牌进行比较

例如一个无序数组 3、1、7、5、2、4、9、6，将其升序的结果则如下：

一开始有序表中无数据，直接插入3

从第二个数开始，插入一个元素1，然后和有序表中记录3比较，1<3，所以插入到记录 3 的左侧

向有序表插入记录 7 时，同有序表中记录 3 进行比较，3<7，所以插入到记录 3 的右侧

向有序表中插入记录 5 时，同有序表中记录 7 进行比较，5<7，同时 5>3，所以插入到 3 和 7 中间

照此规律，依次将无序表中的记录 4，9 和 6插入到有序表中

15.2. 如何实现

将第一待排序序列第一个元素看做一个有序序列，把第二个元素到最后一个元素当成是未排序序列。

从头到尾依次扫描未排序序列，将扫描到的每个元素插入有序序列的适当位置

如果待插入的元素与有序序列中的某个元素相等，则将待插入元素插入到相等元素的后面

用代码表示则如下：

```
function insertionSort(arr) {
    const len = arr.length;
    let preIndex, current;
    for (let i = 1; i < len; i++) {
```

```
        preIndex = i - 1;
        current = arr[i];
        while(preIndex >= 0 && arr[preIndex] > current) {
            arr[preIndex+1] = arr[preIndex];
            preIndex--;
        }
        arr[preIndex+1] = current;
    }
    return arr;
}
```

在插入排序中，当待排序数组是有序时，是最优的情况，只需当前数跟前一个数比较一下就可以了，这时一共需要比较 `N- 1` 次，时间复杂度为 `O(n)`

最坏的情况是待排序数组是逆序的，此时需要比较次数最多，总次数记为：1+2+3+...+N-1，所以，插入排序最坏情况下的时间复杂度为 `O(n^2)`

通过上面了解，可以看到插入排序是一种稳定的排序方式

15.3. 应用场景

插入排序时间复杂度是 O(n2)，适用于数据量不大，算法稳定性要求高，且数据局部或整体有序的数列排序

16. 说说你对分而治之、动态规划的理解？区别？

16.1. 分而治之

分而治之是算法设计中的一种方法，就是把一个复杂的问题分成两个或更多的相同或相似的子问题，直到最后子问题可以简单的直接求解，原问题的解即子问题的解的合并

关于分而治之的实现，都会经历三个步骤：

- 分解：将原问题分解为若干个规模较小，相对独立，与原问题形式相同的子问题
- 解决：若子问题规模较小且易于解决时，则直接解。否则，递归地解决各子问题
- 合并：将各子问题的解合并为原问题的解

实际上，关于分而治之的思想，我们在前面已经使用，例如归并排序的实现，同样经历了实现分而治之的三个步骤：

- 分解：把数组从中间一分为二
- 解决：递归地对两个子数组进行归并排序
- 合并：将两个字数组合并称有序数组

同样关于快速排序的实现，亦如此：

- 分：选基准，按基准把数组分成两个字数组
- 解：递归地对两个字数组进行快速排序
- 合：对两个字数组进行合并

同样二分搜索也能使用分而治之的思想去实现，代码如下：

```
function binarySearch(arr,l,r,target){
    if(l> r){
        return -1;
    }
    let mid = l + Math.floor((r-l)/2)
    if(arr[mid] === target){
        return mid;
    }else if(arr[mid] < target ){
        return binarySearch(arr,mid + 1,r,target)
    }else{
        return binarySearch(arr,l,mid - 1,target)
    }
}
```

16.2. 动态规划

动态规划，同样是算法设计中的一种方法，是一种在数学、管理科学、计算机科学、经济学和生物信息学中使用的，通过把原问题分解为相对简单的子问题的方式求解复杂问题的方法

常常适用于有重叠子问题和最优子结构性质的问题

简单来说，动态规划其实就是，给定一个问题，我们把它拆成一个个子问题，直到子问题可以直接解决

然后呢，把子问题答案保存起来，以减少重复计算。再根据子问题答案反推，得出原问题解的一种方法。

一般这些子问题很相似，可以通过函数关系式递推出来，例如斐波那契数列，我们可以得到公式：当 n 大于 2的时候，F(n) = F(n-1) + F(n-2) ，

f(10)= f(9)+f(8),f(9) = f(8) + f(7)...是重叠子问题，当n = 1、2的时候，对应的值为2，这时候就通过可以使用一个数组记录每一步计算的结果，以此类推，减少不必要的重复计算

16.2.1. 适用场景

如果一个问题，可以把所有可能的答案穷举出来，并且穷举出来后，发现存在重叠子问题，就可以考虑使用动态规划

比如一些求最值的场景，如最长递增子序列、最小编辑距离、背包问题、凑零钱问题等等，都是动态规划的经典应用场景

关于动态规划题目解决的步骤，一般如下：

- 描述最优解的结构
- 递归定义最优解的值
- 按自底向上的方式计算最优解的值
- 由计算出的结果构造一个最优解

16.3. 区别

动态规划算法与分治法类似，其基本思想也是将待求解问题分解成若干个子问题，先求解子问题，然后从这些子问题的解得到原问题的解

与分治法不同的是，适合于用动态规划求解的问题，经分解得到子问题往往**不是互相独立**的，而分而治之的子问题是相互独立的

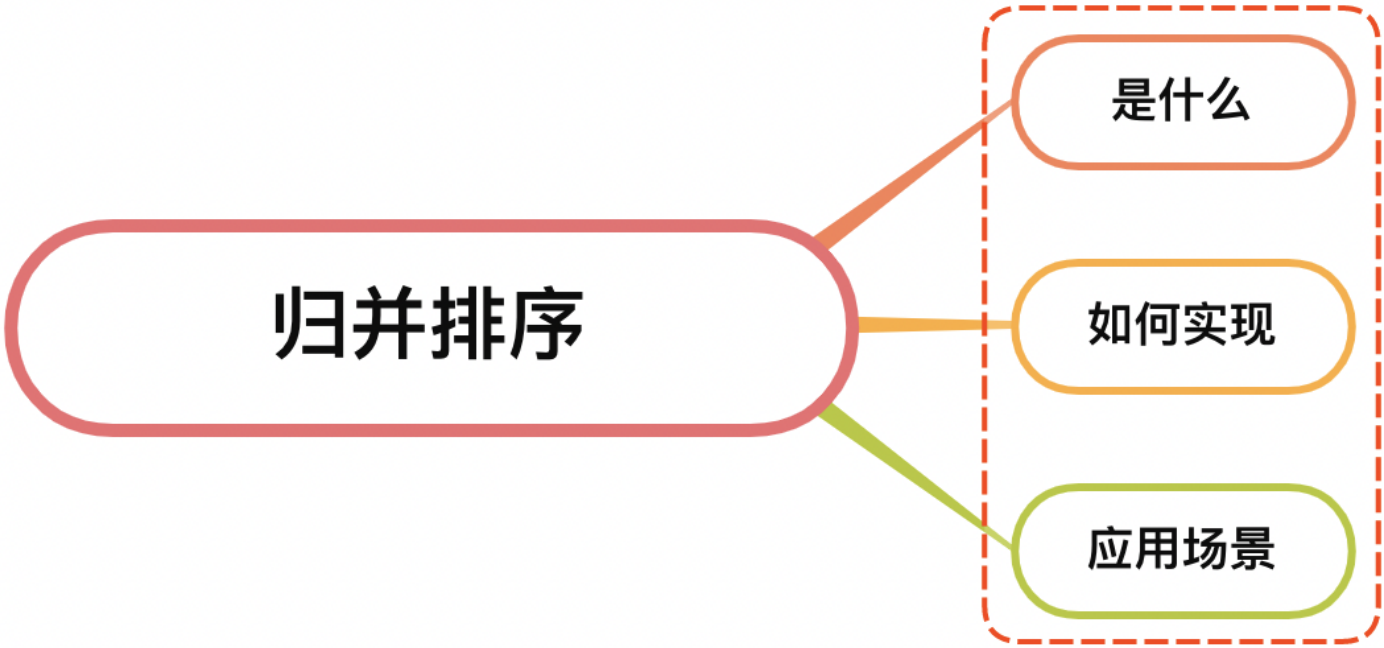
若用分治法来解这类问题，则分解得到的子问题数目太多，有些子问题被重复计算了很多次

如果我们能够保存已解决的子问题的答案，而在需要时再找出已求得的答案，这样就可以避免大量的重复计算，节省时间

综上，可得：

- 动态规划：有最优子结构和重叠子问题
- 分而治之：各子问题独立

17. 说说你对归并排序的理解？如何实现？应用场景？



17.1. 是什么

归并排序（Merge Sort）是建立归并操作上的一种有效，稳定的排序算法，该算法是采用分治法的一个非常典型的应用

将已有序的子序列合并，得到完全有序的序列，即先使每个子序列有序，再使子序列段间有序

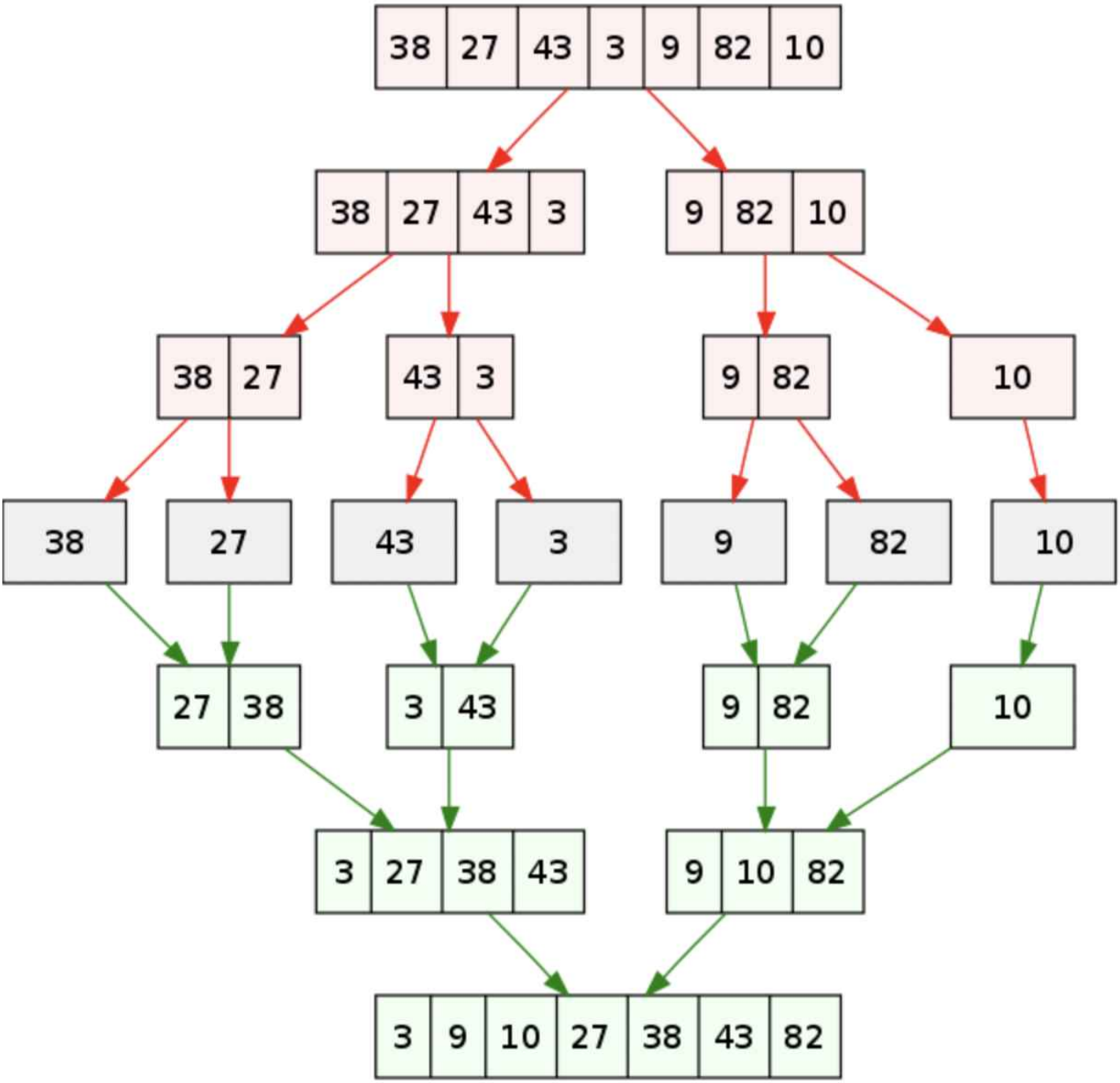
例如对于含有 `n` 个记录的无序表，首先默认表中每个记录各为一个有序表（只不过表的长度都为 1）

然后进行两两合并，使 `n` 个有序表变为 `n/2` 个长度为 2 或者 1 的有序表（例如 4 个小有序表合并为 2 个大的有序表）

通过不断地进行两两合并，直到得到一个长度为 `n` 的有序表为止

例如对无序表{49，38，65，97，76，13，27}进行归并排序分成了分、合两部分：

如下图所示：



归并合过程中，每次得到的新的子表本身有序，所以最终得到有序表

上述分成两部分，则称为二路归并，如果分成三个部分则称为三路归并，以此类推

17.2. 如何实现

关于归并排序的算法思路如下：

- 分：把数组分成两半，再递归对子数组进行分操作，直至到一个个单独数字
- 合：把两个数合成有序数组，再对有序数组进行合并操作，直到全部子数组合成一个完整的数组
  - 合并操作可以新建一个数组，用于存放排序后的数组
  - 比较两个有序数组的头部，较小者出队并且推入到上述新建的数组中
  - 如果两个数组还有值，则重复上述第二步

- 如果只有一个数组有值，则将该数组的值出队并推入到上述新建的数组中

用代码表示则如下图所示：

```
function mergeSort(arr) {  // 采用自上而下的递归方法
    const len = arr.length;
    if(len < 2) {
        return arr;
    }
    let middle = Math.floor(len / 2),
        left = arr.slice(0, middle),
        right = arr.slice(middle);
    return merge(mergeSort(left), mergeSort(right));
}
function merge(left, right)
{
    const result = [];
    while (left.length && right.length) {
        if (left[0] <= right[0]) {
            result.push(left.shift());
        } else {
            result.push(right.shift());
        }
    }
    while (left.length)
        result.push(left.shift());
    while (right.length)
        result.push(right.shift());
    return result;
}
```

上述归并分成了分、合两部分，在处理分过程中递归调用两个分的操作，所花费的时间为2乘 `T(n/2)` ，合的操作时间复杂度则为 `O(n)` ，因此可以得到以下公式：

总的执行时间 = 2 × 输入长度为 `n/2` 的 `sort` 函数的执行时间 + `merge` 函数的执行时间 `O(n)`

当只有一个元素时， `T(1) = O(1)`

如果对 `T(n) = 2 * T(n/2) + O(n)` 进行左右 / n的操作，得到 `T(n) / n = (n / 2) * T(n/2) + O(1)`

现在令 `S(n) = T(n)/n` ，则 `S(1) = O(1)` ，然后利用表达式带入得到 `S(n) = S(n/2) + O(1)`

所以可以得到： `S(n) = S(n/2) + O(1) = S(n/4) + O(2) = S(n/8) + O(3) = S(n/2^k) + O(k) = S(1) + O(logn) = O(logn)`

综上可得，`T(n) = n * log(n) = nlogn`

关于归并排序的稳定性，在进行合并过程，在1个或2个元素时，1个元素不会交换，2个元素如果大小相等也不会交换，由此可见归并排序是稳定的排序算法

17.3. 应用场景

在外排序中通常使用排序-归并的策略，外排序是指处理超过内存限度的数据的排序算法，通常将中间结果放在读写较慢的外存储器，如下分成两个阶段：

- 排序阶段：读入能够放进内存中的数据量，将其排序输出到临时文件，一次进行，将带排序数据组织为多个有序的临时文件
- 归并阶段：将这些临时文件组合为大的有序文件

例如，使用100m内存对900m的数据进行排序，过程如下：

- 读入100m数据内存，用常规方式排序
- 将排序后的数据写入磁盘
- 重复前两个步骤，得到9个100m的临时文件
- 将100m的内存划分为10份，将9份为输入缓冲区，第10份为输出缓冲区
- 进行九路归并排序，将结果输出到缓冲区
    - 若输出缓冲区满，将数据写到目标文件，清空缓冲区
    - 若缓冲区空，读入相应文件的下一份数据

18. 说说你对贪心算法、回溯算法的理解？应用场景？

18.1. 贪心算法

贪心算法，又称贪婪算法，是算法设计中的一种思想

其期待每一个阶段都是局部最优的选择，从而达到全局最优，但是结果并不一定是最优的

举个零钱兑换的例子，如果你有1元、2元、5元的钱币数张，用于兑换一定的金额，但是要求兑换的钱币张数最少

如果现在你要兑换11元，按照贪心算法的思想，先选择面额最大的5元钱币进行兑换，那么就得到11 = 5 + 5 + 1 的选择，这种情况是最优的

但是如果你手上钱币的面额为1、3、4，想要兑换6元，按照贪心算法的思路，我们会 6 = 4 + 1 + 1这样选择，这种情况结果就不是最优的选择

从上面例子可以看到，贪心算法存在一些弊端，使用到贪心算法的场景，都会存在一个特性：

一旦一个问题可以通过贪心法来解决，那么贪心法一般是解决这个问题的最好办法

至于是否选择贪心算法，主要看是否有如下两大特性：

- 贪心选择：当某一个问题的整体最优解可通过一系列局部的最优解的选择达到，并且每次做出的选择可以依赖以前做出的选择，但不需要依赖后面需要做出的选择
- 最优子结构：如果一个问题的最优解包含其子问题的最优解，则此问题具备最优子结构的性质。问题的最优子结构性质是该问题是否可以用贪心算法求解的关键所在

18.2. 回溯算法

回溯算法，也是算法设计中的一种思想，是一种渐进式寻找并构建问题解决方式的策略

回溯算法会先从一个可能的工作开始解决问题，如果不行，就回溯并选择另一个动作，知道将问题解决

使用回溯算法的问题，有如下特性：

- 有很多路，例如一个矩阵的方向或者树的路径
- 在这些的路里面，有死路也有生路，思路即不符合题目要求的路，生路则符合
- 通常使用递归来模拟所有的路

常见的伪代码如下：

```
result = []
function backtrack(路径, 选择列表):
  if 满足结束条件:
    result.add(路径)
  return
  for 选择 of 选择列表:
    做选择
    backtrack(路径, 选择列表)
    撤销选择
```

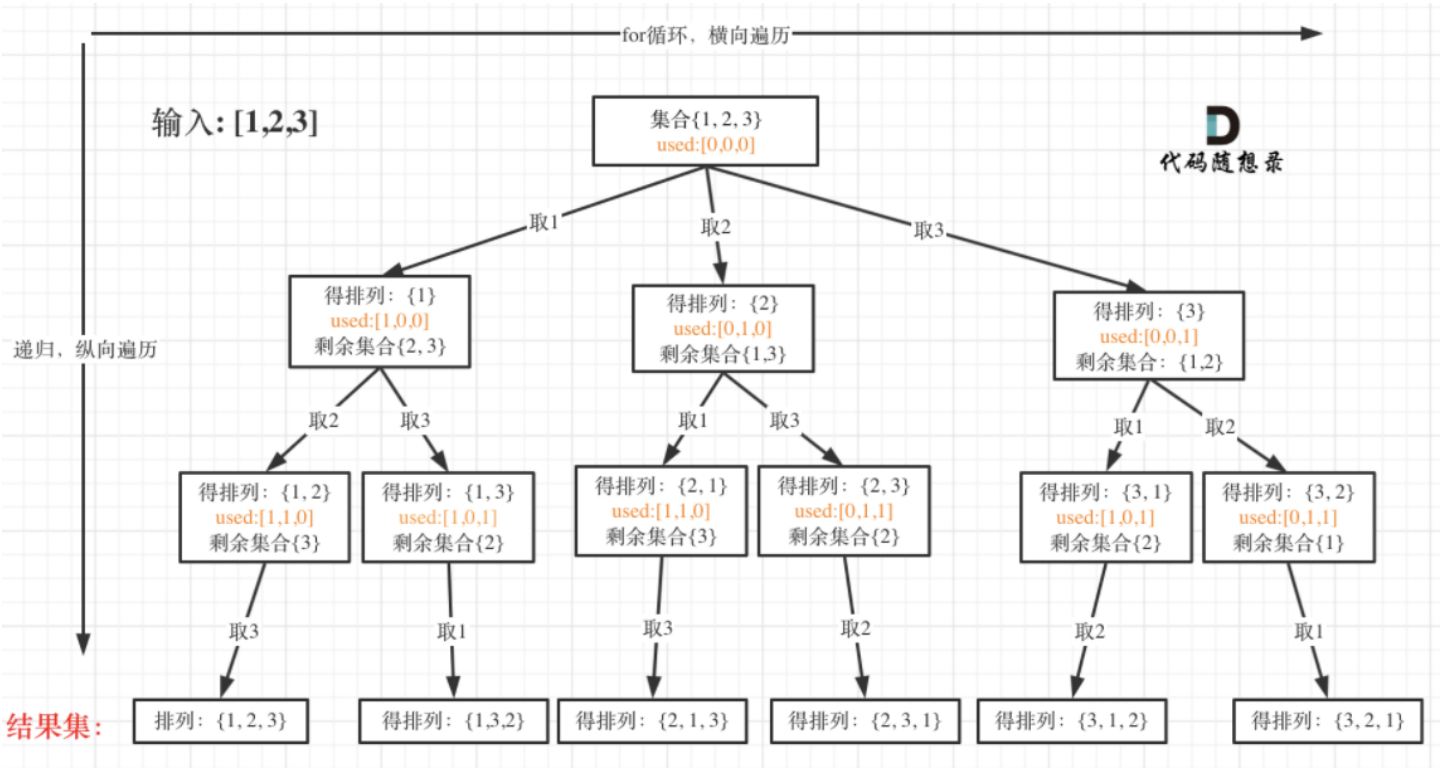
重点解决三个问题：

- 路径：也就是已经做出的选择
- 选择列表
- 结束条件

例如经典使用回溯算法为解决全排列的问题，如下：

一个不含重复数字的数组 `nums` ，我们要返回其所有可能的全排列，解决这个问题的思路是：

- 用递归模拟所有的情况
- 遇到包含重复元素的情况则回溯
- 收集到所有到达递归终点的情况，并返回、



用代码表示则如下：

```
var permute = function(nums) {
    const res = [], path = [];
    backtracking(nums, nums.length, []);
    return res;

    function backtracking(n, k, used) {
        if(path.length === k) {
            res.push(Array.from(path));
            return;
        }
        for (let i = 0; i < k; i++ ) {
            if(used[i]) continue;
```

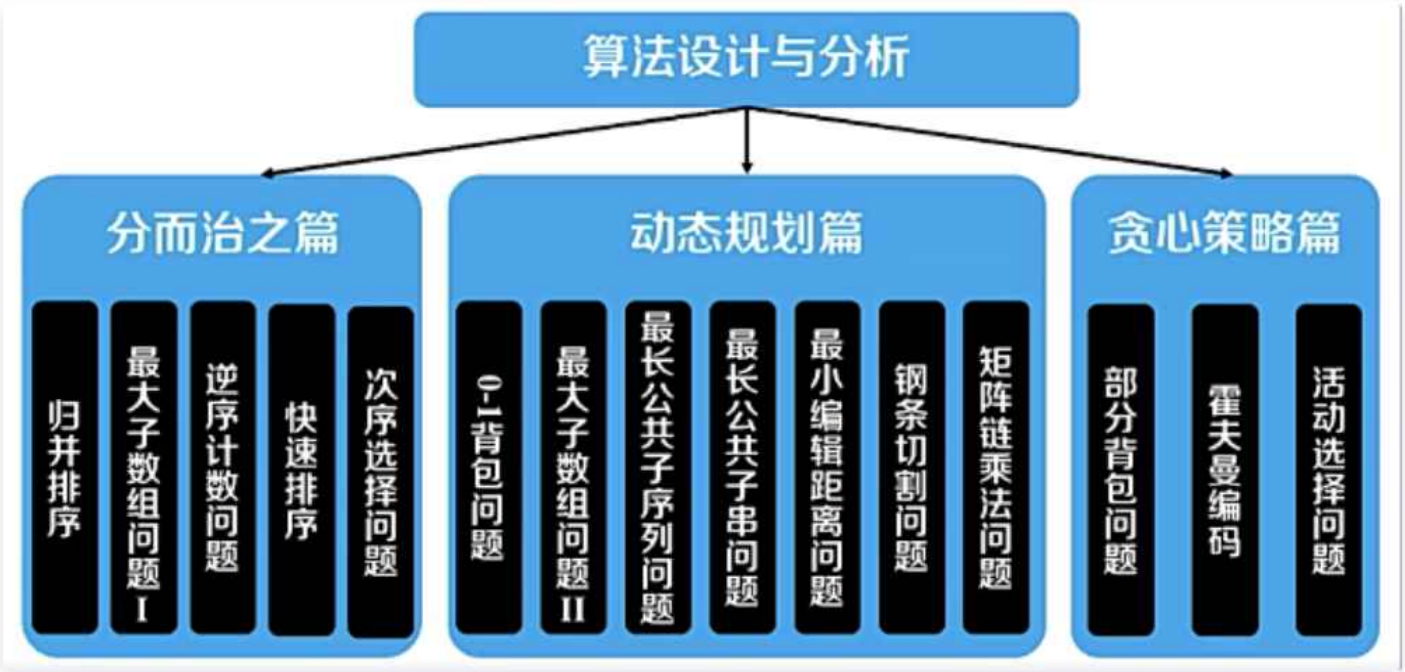
```
            path.push(n[i]);
            used[i] = true; // 同支
            backtracking(n, k, used);
            path.pop();
            used[i] = false;
        }
    }
};
```

18.3. 总结

前面也初步了解到分而治之、动态规划，现在再了解到贪心算法、回溯算法

其中关于分而治之、动态规划、贪心策略三者的求解思路如下：

其中三者对应的经典问题如下图：


算法设计与分析
分而治之篇
归并排序
最大子数组问题 I
逆序计数问题
快速排序
次序选择问题
动态规划篇
0-1背包问题
最大子数组问题 II
最长公共子序列问题
最长公共子串问题
最小编辑距离问题
钢条切割问题
矩阵链乘法问题
贪心策略篇
部分背包问题
霍夫曼编码
活动选择问题