递归

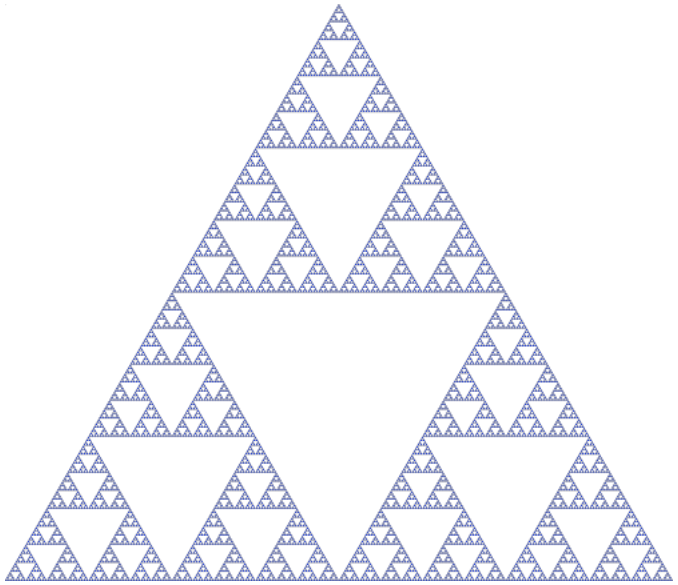
算法I课程

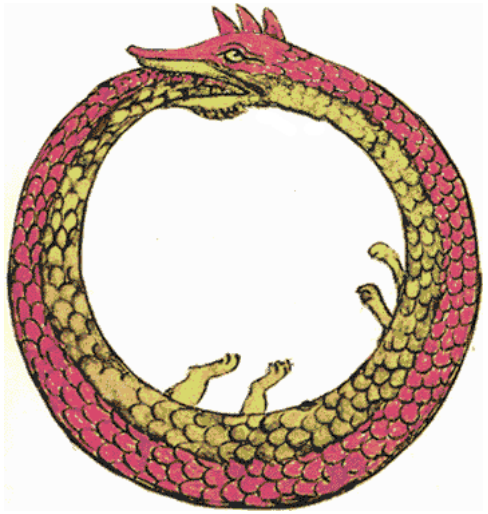
课程目录

- 递归和递归表达式
- 递归性能的思考
- 前端相关递归算法精选
  - DOM的绝对位置
  - 深度拷贝
  - 深度比较
- 树和递归
  - 树的定义
  - 前端的应用场景
  - 树的递归定义
  - 树的遍历
  - 树的查找
  - 树的路径
  - CSS选择器
  - 最长相同节点路径问题
- 其他经典问题选讲
  - 全排列
    - 一般解法
    - 基于交换的解法
    - Heap的方法
  - 数组相邻项最大和问题
    - 一种暴力的解法

递归

当事物用它本身定义自己，就会发生递归





递归表达式

- $n! = n \times (n-1)!$ $0! = 1$

- $$f(n) = \begin{cases} 1, & n = 1 \\ 1, & n = 2 \\ f(n-1) + f(n-2), & n > 2 \end{cases}$$

递归通常需要初始条件和递归表达式。

例27-阶乘

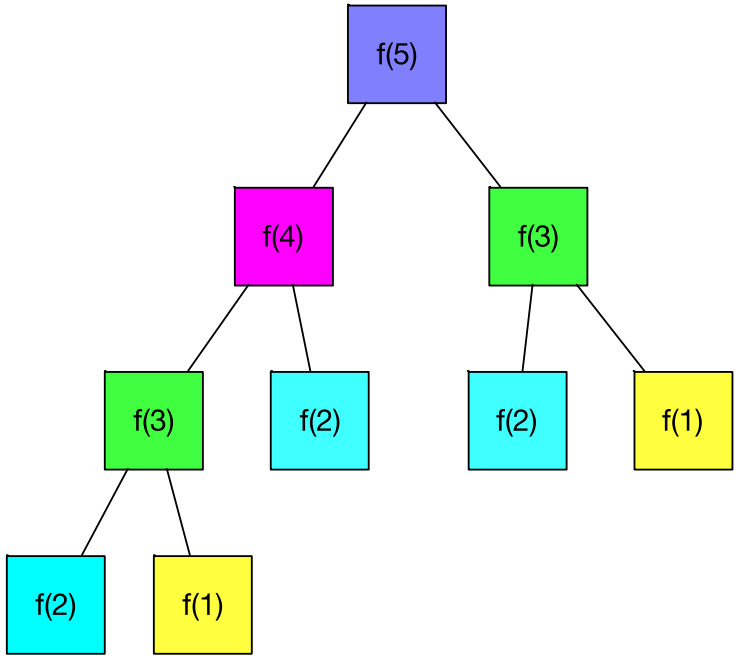
```
function factorial(n){
  return n === 0 ? 1 : factorial(n-1) * n
}
```

例28-斐波那契数列

$$1,\ 1,\ 2,\ 3,\ 5,\ 8,\ 13,\ 21,\ 34,\ 55,\ 89,\ 144,\ \ldots$$

```
function fibonacci(n){
  return n == 1 || n== 2 ? 1 :
      fibonacci(n - 1) + fibonacci(n-2)
```

对性能的思考-斐波那契数列的递归



- f(5)f(4)执行了1次
- f(3)执行了2次
- f(2)执行了3次
- f(1)执行了2次

递归次数$N \geq 1 + 2 + 4 + 8 + \ldots + 2^{n-2}$

$= \frac{(1 - 2^{n-1})}{1 - 2}$

$= 2^{n-1} - 1$

$2^{10} = 1024$

$2^{20} = 1{,}048{,}576$

$2^{30} = 1{,}073{,}741{,}824$

例28-从底端构造递归（斐波那契数列）

Reduce和For循环的两种写法，本质一样

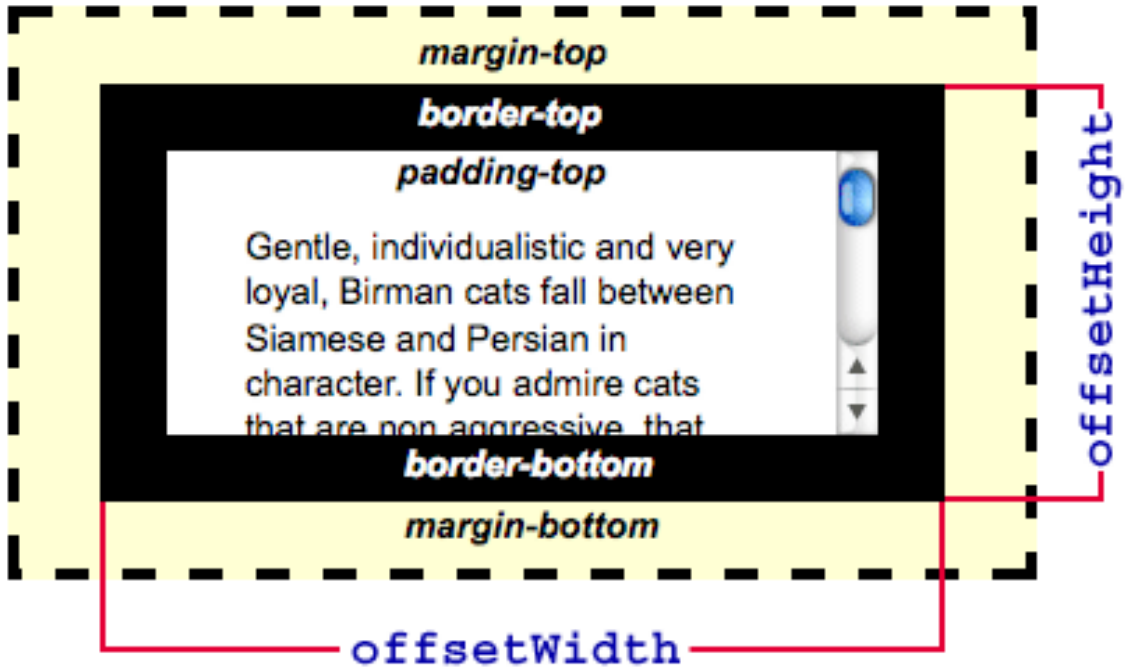
```
function fibonacci(n){
  let [a,b] = [0, 1]
  for(let i = 0; i < n; i++){
    [a, b] = [b, a+b]
  }
  return b
}
```

```
function fibonacci(n){
  return Array(n).fill()
    .reduce( ([a,b], _) => {
      return [b, a+b]
    }, [0, 1])[1]
}
```

前端问题举例

- DOM节点绝对位置
- 深度拷贝
- 深度比较

例29-DOM节点的绝对位置



offsetLeft, offsetRight是相对于offsetParent的位置;
Element.getBoundingClientRect()是相对于视窗的位置，回受滚动的影响

例29-DOM节点的绝对位置

```
1   function get_layout(ele){
2     const layout = {
3       width : ele.offsetWidth,
4       height : ele.offsetHeight,
5       left : ele.offsetLeft,
6       top : ele.offsetTop
7     }
8     if(ele.offsetParent){
9       const parentLayout = get_layout(ele.offsetParent)
1       layout.left += parentLayout.left
0       layout.top += parentLayout.top
1     }
1     return layout
1   }
```

```
function get_layout(ele){
  let left = ele.offsetLeft, top = ele.offsetTop
  let p = ele.offsetParent
  while(p){
    left += p.offsetLeft
    top += p.offsetTop
    p = p.offsetParent
  }
  return {
    width : ele.offsetWidth,
    height : ele.offsetHeight,
    left : left,
    top : top
  }
}
```

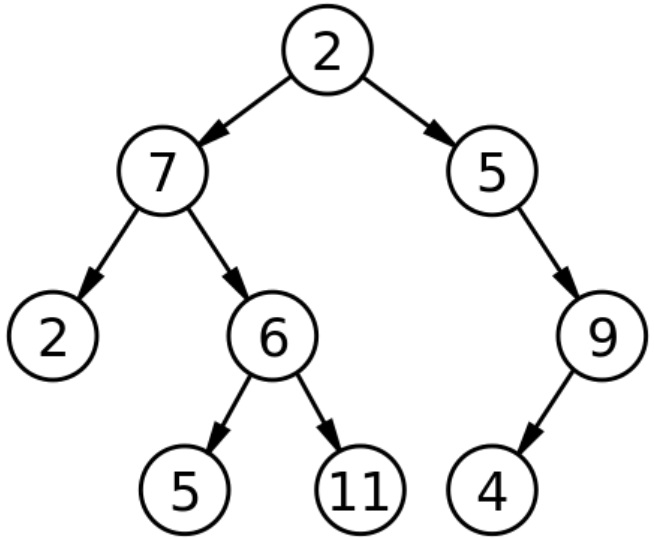
例30-深度拷贝

```
1  function clone(obj)
2  {
3    if(obj == null || typeof obj !== 'object') return obj
4    const newObj = new obj.constructor()
5    for(let key in Object.getOwnPropertyDescriptors(obj)){
6      newObj[key] = clone( obj[key] )
7    }
8    return newObj
9  }
```

例31-深度比较

```
1   function deepCompare(a, b){
2     if(a === null || typeof a !== 'object'
3       || b === null || typeof b !== 'object'){
4       return a === b
5     }
6     const propsA = Object.getOwnPropertyDescriptors(a)
7     const propsB = Object.getOwnPropertyDescriptors(b)
8     if(Object.keys(propsA).length !==
9                         Object.keys(propsB).length){
1       return false
0     }
1     return Object.keys(propsA).every(
1               key => deepCompare(a[key], b[key]))
1   }
2
```

树



- 根节点(**root**) - 树的顶端
- 子节点(**children**) – 直接连到另一个节点的节点
- 父节点(**parent**) - 和子节点相反
- 相邻（兄弟）节点(**siblings**)-拥有同一个父节点的节点
- 后代节点(**descendant**) – 从一个节点重复处理（父到子）得到的所有节点
- 祖先节点(**ancestor**) - 从一个节点重复处理从子到父得到的所有节点
- 叶子节点(**leaf**) – 没有子节点的节点
- 分支(**branch**) – 至少有1个子节点的节点
- 度(**degree**) – 一个节点的子节点数
- 边(**edge**) – 两个节点的连接
- 路径(**path**) – 连接一个节点和它的某个后代的所有节点和边
- 深度(**depth**) – 从根节点到某个节点的边的数量
- 节点高度(**height of node**) – 一个节点和它后代叶子节点的最长路径
- 森林(**forest**) – 多个不相交树组成的集合

继续向写一个React源代码DEMO靠近

- 先尝试写个选择器
- Keep Fighting!~

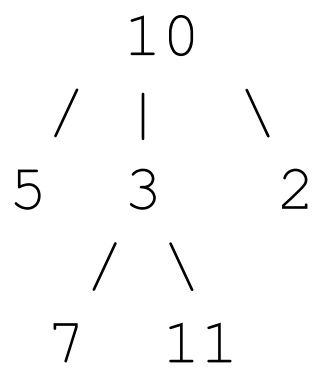
树的算法和前端

- DOM
- 选择器
- JSON
- 虚拟DOM（React,Vue,AngularJS)
- 文本查找和输入提示

例32-树的递归表示

$T:\ v,[T_1,\ldots,T_k]$ 数含有值v和一个子树的列表

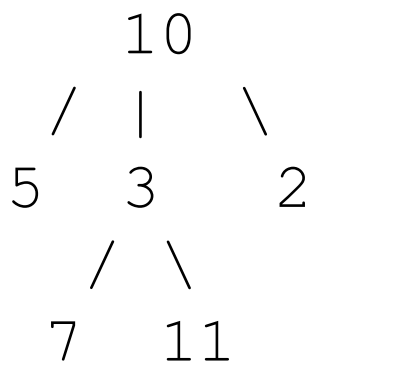
```
class Tree{
  constructor(v, children){
    this.v = v
    this.children = children || null
  }
}
```




```
const tree = new Tree(10, [
  new Tree(5),
  new Tree(3, [
    new Tree(7),
    new Tree(11)
  ]),
  new Tree(2)
])
```

例子33-树的遍历（先序）

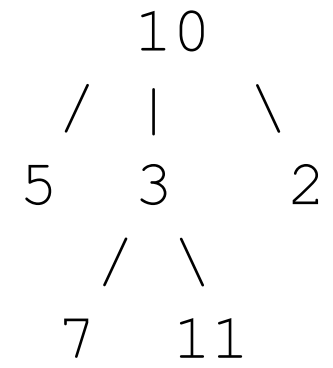
```
function tree_transverse(tree){
  console.log(tree.v)
  tree.children && tree.children.forEach(tree_transverse)
}
```



10 5 3 7 11 2

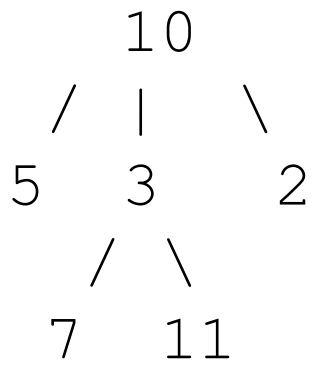
例子33-树的遍历（后序)

```
function tree_transverse_l(tree) {
  tree.children && tree.children.forEach(tree_transverse_l)
  console.log(tree.v)
}
```



5 7 11 3 2 10

例33-树的遍历（中序）




```
tree_transverse_m(tree, 0)
// 10 5 3 7 11 2
tree_transverse_m(tree, 3)
// 5 7 11 3 2 10
tree_transverse_m(tree, 1)
// 5 10 7 3 11 2
```

例33-树的遍历（中序）

```
1  function tree_transverse_m(tree, ord = 0) {
2    let transversed = false
3    if(!tree.children) {
4      console.log(tree.v)
5      return
6    }
7    tree.children.forEach( (child, i) => {
8      if( i === ord ){
9        transversed = true
1        console.log(tree.v)
0      }
1      tree_transverse_m(child, ord)
1    })
1    !transversed && console.log(tree.v)
2  }
```

例34-树的遍历（回调）

```
1  function tree_transverse(tree, ord = 0, callback) {
2    let transversed = false
3    if(!tree.children) {
4      callback(tree.v)
5      return
6    }
7    tree.children.forEach( (child, i) => {
8      if( i === ord ){
9        transversed = true
1        callback(tree.v)
0      }
1      tree_transverse(child, ord, callback)
1    })
1    !transversed && callback(tree.v)
2  }
```

例34-树的遍历（回调）

```
// 先序遍历
tree_transverse(tree, (node) => console.log(node.v) )

// 中序遍历
tree_transverse(tree, (node) => console.log(node.v) , 1)
tree_transverse(tree, (node) => console.log(node.v) , 2)
// 后序遍历
tree_transverse(tree, (node) => console.log(node.v) , 3)
tree_transverse(tree, (node) => console.log(node.v) , 4)
...
```

例35-树的遍历（基于Generator)

使用Generator函数可以将遍历操作变成一个数组结构

```
const nodes = [...tree_transverse(tree)]
```

或

```
for(let node of tree){
  // ...
}
```

```
function* tree_transverse(tree, ord=0) {
  let transversed = false
  if(!tree.children) {
    yield tree
    return
  }
  for(let i = 0; i < tree.children.length; i++){
    if( i === ord ){
      transversed = true
      yield tree
    }
    yield *tree_transverse(tree.children[i], ord)
  }
  if(!transversed){
    yield tree
  }
}
```

例36-树的查找

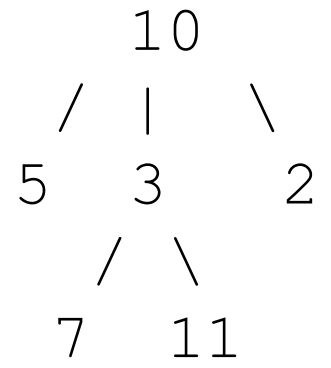
```
function find(tree, prediction){
  return [...tree_transverse(tree)].find(prediction)
}

function find(tree, prediction){
  for (let node of tree_transverse(tree)){
    if(prediction(node)){return node}
  }
}


find(tree, node => node.v === 2)
```

例子37-树的路径

通常我们会用路径来描述一个子节点，
比如：
Css的选择器：#app ul li a
XML的 xpath : /bookstore/book[price>35.00]
这样我们只需要知道根节点和路径就可以定位**一个或多个**后代

构造一个先序遍历，除了返回节点外，还返回路径

```
function* tree_transverse(tree, path = []) {
  yield { tree, path }
  if(tree.children){
    for(let i = 0; i < tree.children.length; i++){
      yield * tree_transverse(tree.children[i], [...path, i])
    }
  }
}
```



节点11的路径[1,1]

我们可以通过上述遍历函数查询一个节点的路径

```
function find_path(v){
  for(let {tree, path} of v){
    if(tree.v === v){
      return path
    }
  }
}

//find_path(tree, 11)
// [1,1]
```

当然我们可以根据路径反查节点（选择器）

```
function find_by_path(tree, path){
  return path.length === 0 ? tree
    : find_by_path(tree.children[path[0]], path.slice(1))
}

// find_by_path(tree, [1,1])
// Tree {v: 11, children: null}
```

例38-选择器

扩展例37中find_by_path的语法，比如支持:

```
// select([1, '[>5]']) => [7, 11]
```

进行标准化的化简

```
    10
  /  |  \
5    3   2
   /  |  \
  7   11  1
```

```
[
  {child : 1},
  {op : (x) => x.v > 5}
]
```

标准化的选择器函数

```
function select(node, path){
  if(path.length === 0){ return [node]}
  const p = path.shift()
  if(p.child){
    return select(node.children[p.child], [...path])
  } else if(p.op) {
    return [...tree_transverse(node)]
      .filter(_n => p.op(_n.node))
      .map( n => n.node)
  }
}
```

解析选择表达式: 1 [>5] =>

```
    {child : 1},
    {op : x => x.v > 5}
]
```

```
function parse_selection_exp(expr){
  return expr.split(' ')
    .map(p => {
      if(p.match(/^\d+$/)){
        return {child : parseInt(p)}
      } else {
        return {
          op : eval( `(x) => x.v ${p.replace(/[\[\]]/g, '')}`)
        }
      }
    })
}
```

```
function select_easy(tree, expr){
  return select(tree, parse_selection_exp(expr))
}

// select_easy(tree, '1 [>5]')
// [Tree(7) Tree(11)]
```

例子39-css选择器

实现一个基于class和Tag的简单css选择器

```
<div>
  <div
class="content">
    <table>
      <tr>
        <td>1</td>
        <td>2</td>
        <td>3</td>
      </tr>
    </table>
  </div>
</div>
```

```
function select(node, expr){}

select(tree, '.content tr td')
```

树的抽象和表示

```
class DOMTree{
  constructor(tag, className,children = []){
    this.tag = tag
    this.className = className
    this.children = children
  }
}
```

```
const tree = new DOMTree('div', '', [
  new DOMTree('div', 'content', [
    new DOMTree('table', '', [
      new DOMTree('tr', '', [
        new DOMTree('td', '', [
          new DOMTree('text', '', null, '1')
        ]),
        new DOMTree('td', '', [
          new DOMTree('text', '', null, '2')
        ]),
        new DOMTree('td', '', [
          new DOMTree('text', '', null, '2')
        ])
      ])
    ])
  ])
])
```

节点的遍历/查找方法

```
function * transverse(node){
  yield node
  if(node.children) {
    for(let i = 0; i < node.children.length; i++){
      yield *transverse(node.children[i])
    }
  }
}

function findByClassName(node, className){
  return [...transverse(node)].filter(node => node.className == className)
}

function findByTagName(node, tagName){
  return [...transverse(node)].filter(node => node.tag == tagName)
}
```

表达式解析

```
function selection_expr_parse(expr){
  return expr.split(' ')
    .map(x => {
      if(x[0] === '.'){
        return {className : x.substr(1)}
      }else {
        return {tagName : x}
      }
    })
}
```

```
// '.content tr td' =>
// [
//   { className : 'content' },
//   { tagName : 'tr' },
//   { tagName : 'td' }
// ]
```

```
function select(node, path){
  if(path.length === 0) {return [node]}
  const p = path.shift()
  let nodes = []
  if(p.className){
    nodes = findByClassName(node, p.className)
  } else { // p.tag
    nodes = findByTagName(node, p.tagName)
  }
  return [].concat(
    ...nodes.map(n => select(n, [...path]))
  )
}
```

性能和思考

- 对className和tagName进行倒排，获得索引对象

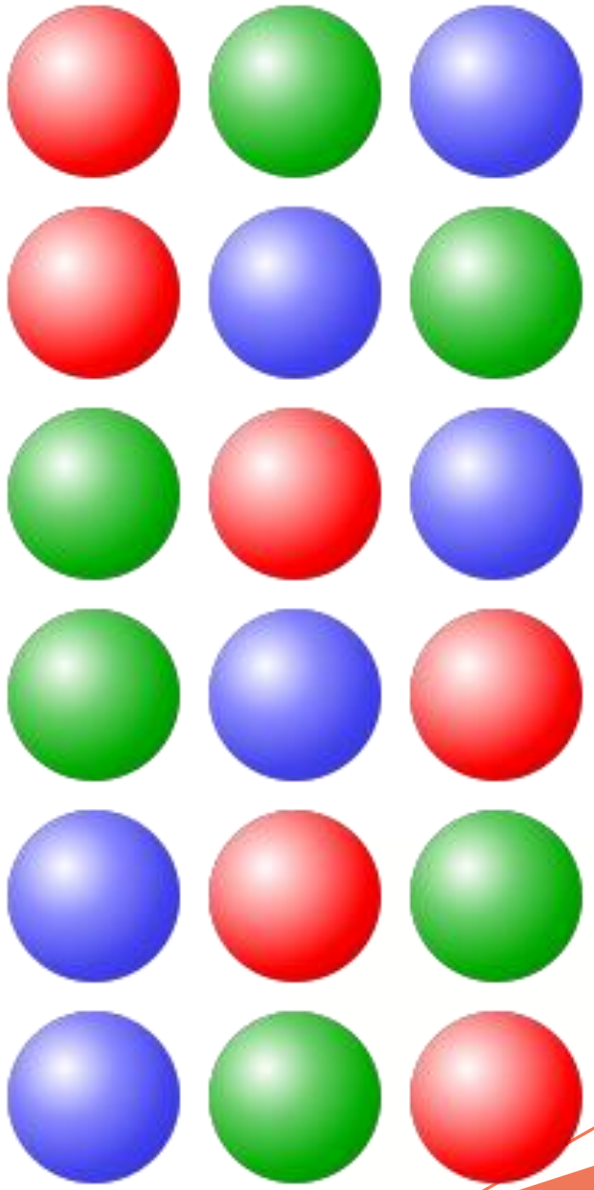
```
function index(tree){
  const classes = {}
  const nodes = [...transverse(tree)]
  nodes.forEach( node => {
    if(node.className) {
      if (!classes[node.className]) {
        classes[node.className] = []
      }
      classes[node.className].push(node)
    }
  })
  return classes
}
```

例39-1 DOM选择器

复杂问题的思考方法

- 全排列
- 树的最长重复路径

全排列问题



将相异的物件或者符号进行排序，每个顺序称作一个排列。「全排列」问题，就是求出所有可能的排列。
比如1,2,3的全排列：
1,2,3
1,3,2
2,1,3
2,3,1
3,1,2
3,2,1

例40-全排列简单解

$$P(A) = a_1\ P(A - a_1)\ \cup\ a_2\ P(A - a_2) \cup \ldots\ \cup a_n\ P(A\text{-}a_n)$$

P[1,2,3,4] = 1 P[2,3,4] U 2 P [1,3,4] U 3 P [1,2,4] U 4 P[1,2,3]

```
1  function perm(A){
2    if(A.length === 1) {return [A]}
3    return [].concat(...A.map((a, i) =>
4      perm(A.slice(0, i)
5              .concat(A.slice(i+1))).map(p => [a].concat(p))
6    ))
7  }
```

例41-全排列（交换）

依次把元素换到最后一个

```
function* perm(A, N){
  if(!N) {N = A.length}
  if(N === 1) { yield A.slice(); return }
  for(let i = 0; i < N; i++) {
    swap(A, i, N - 1)
    yield * perm(A, N - 1)
    swap(A, i, N - 1)
  }
}
```

```
// [ 1, 2, 3 ] 3
// [ 3, 2, 1 ] 2
// [ 2, 3, 1 ] 1
// [ 3, 2, 1 ] 1
// [ 1, 3, 2 ] 2
// [ 3, 1, 2 ] 1
// [ 1, 3, 2 ] 1
// [ 1, 2, 3 ] 2
// [ 2, 1, 3 ] 1
// [ 1, 2, 3 ] 1
```

```
//A
swap(A, i, N - 1)
yield * perm(A, N - 1)
swap(A, i, N - 1)
//A
```

N = 2 成立
N = k 假设成立
N = k+1 成立（因为所有yield语句都不改变前k个元素的顺序)，所以最后换回来顺序不变

开始时的A和结束时的A相同

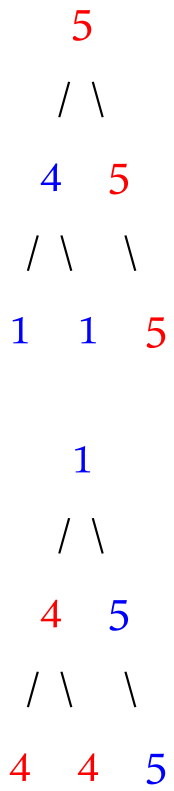
```
begin [ 1, 2, 3 ] 3
    begin [ 3, 2, 1 ] 2
    end [ 3, 2, 1 ] 2
    begin [ 3, 2, 1 ] 2
    end [ 3, 2, 1 ] 2
end [ 1, 2, 3 ] 3
begin [ 1, 2, 3 ] 3
    begin [ 1, 3, 2 ] 2
    end [ 1, 3, 2 ] 2
    begin [ 1, 3, 2 ] 2
    end [ 1, 3, 2 ] 2
end [ 1, 2, 3 ] 3
begin [ 1, 2, 3 ] 3
    begin [ 1, 2, 3 ] 2
    end [ 1, 2, 3 ] 2
    begin [ 1, 2, 3 ] 2
    end [ 1, 2, 3 ] 2
end [ 1, 2, 3 ] 3
```

全排列-Heap的方法

```
function* perm(A, N){
  if(!N) {N = A.length}
  if(N === 1) { yield A.slice(); return }
  for(let i = 0; i < N; i++) {
    yield * perm(A, N - 1)
    if(N % 2 == 1)
      swap(A, i, N - 1)
    else
      swap(A, 0, N - 1)
  }
}
```

```
// [ 1, 2, 3 ]
// [ 2, 1, 3 ]
// [ 3, 2, 1 ]
// [ 2, 3, 1 ]
// [ 3, 1, 2 ]
// [ 1, 3, 2 ]
```

例37-树的最长同值「路径」




```
class BinaryTree{

  constructor(value, left = null, right = null) {
    this.value = value
    this.left = left
    this.right = right
  }

}
```

```
const tree = new BinaryTree(
  5,
  new BinaryTree(
    6,
    new BinaryTree(
      6
    ),
    new BinaryTree(
      6
    )
  ),
  new BinaryTree(
    8,
    new BinaryTree(
      7
    ),
    new BinaryTree(
      9
    )
  )
)
```

统计节点到叶子的所有路径，找到最大的节点个数

```
//      5
//    /   \
//  6      8
// / \    / \
//6   6 7   9
```

```
function max_longest_level(node, val){
  return (node && node.value === val) ?
    Math.max( max_longest_level(node.left, val),
      max_longest_level(node.right, val) )
      + 1 : 0
}
```

```
function* transverse(node){
  yield node
  if(node.left){
    yield* transverse(node.left)
  }
  if(node.right){
    yield* transverse(node.right)
  }
}
function solve(node){
  return [...transverse(node)].reduce( (max, o) => {
    return Math.max( max_longest_level(o.left, o.value)
      + max_longest_level(o.right, o.value), max)
  }, 0)
}
```