# 算法复杂度

参考 ：https://www.cnblogs.com/xiaohuochai/p/8203717.html

## 大O表示法

大O表示法是描述算法的性能和复杂程度。 分析算法时，时常遇到以下几类函数

复制代码

```
符号              名称
O(1)             常数的
O(log(n))        对数的
O((log(n))c)     对数多项式的
O(n)             线性的
O(n2)            二次的
O(nc)            多项式的
O(cn)            指数的
```

复制代码

如何衡量算法的效率？通常是用资源，例如CPU（时间）占用、内存占用、硬盘占用和网络占用。当讨论大O表示法时，一般考虑的是CPU（时间）占用

下面用一些例子来理解大O表示法的规则

【O(1)】

```
function increment(num){
  return ++num;
}
```

假设运行increment(1)函数，执行时间等于X。如果再用不同的参数（例如2）运行一次increment函数，执行时间依然是X。和参数无关，increment函数的性能都一样。因此，我们说上述函数的复杂度是O(1)（常数）

【O(n)】

现在以顺序搜索算法为例：

复制代码

```
function sequentialSearch(array, item){
  for (var i=0; i<array.length; i++){
    if (item === array[i]){ //{1}
      return i;
    }
  }
  return -1;
}
```

复制代码

如果将含10个元素的数组（[1, ..., 10]）传递给该函数，假如搜索1这个元素，那么，第一次判断时就能找到想要搜索的元素。在这里我们假设每执行一次行{1} ，开销是1。

现在，假如要搜索元素11。行{1}会执行10次（遍历数组中所有的值，并且找不到要搜索的元素，因而结果返回 -1）。如果行{1}的开销是1，那么它执行10次的开销就是10，10倍于第一种假设

现在，假如该数组有1000个元素（[1, ..., 1000]）。搜索1001的结果是行{1}执行了1000次（然后返回-1）

sequentialSearch函数执行的总开销取决于数组元素的个数（数组大小），而且也和搜索的值有关。如果是查找数组中存在的值，行{1}会执行几次呢？如果查找的是数组中不存在的值，那么行{1}就会执行和数组大小一样多次，这就是通常所说的最坏情况

最坏情况下，如果数组大小是10，开销就是10；如果数组大小是1000，开销就是1000。可以得出sequentialSearch函数的时间复杂度是O(n)，n是（输入）数组的大小

回到之前的例子，修改一下算法的实现，使之计算开销：

复制代码

```
function sequentialSearch(array, item){
 var cost = 0;
 for (var i=0; i<array.length; i++){
  cost++;
```

```
    if (item === array[i]){ //{1}
      return i;
    }
  }
  console.log('cost for sequentialSearch with input size ' +  array.length + ' is ' + cost);
  return -1;
}
```

复制代码

用不同大小的输入数组执行以上算法，可以看到不同的输出

**【$O(n2)$】**

用冒泡排序做$O(n2)$的例子：

复制代码

```
function swap(array, index1, index2){
  var aux = array[index1];
  array[index1] = array[index2];
  array[index2] = aux;
}
function bubbleSort(array){
  var length = array.length;
  for (var i=0; i<length; i++){ //{1}
    for (var j=0; j<length-1; j++ ){ //{2}
      if (array[j] > array[j+1]){
        swap(array, j, j+1);
      }
    }
  }
}
```

复制代码

假设行{1}和行{2}的开销分别是1。修改算法的实现使之计算开销：

复制代码

```
function bubbleSort(array){
  var length = array.length;
  var cost = 0;
  for (var i=0; i<length; i++){ //{1}
    cost++;
    for (var j=0; j<length-1; j++ ){ //{2}
      cost++;
      if (array[j] > array[j+1]){
        swap(array, j, j+1);
      }
    }
  }
  console.log('cost for bubbleSort with input size ' + length + ' is ' + cost);
}
```

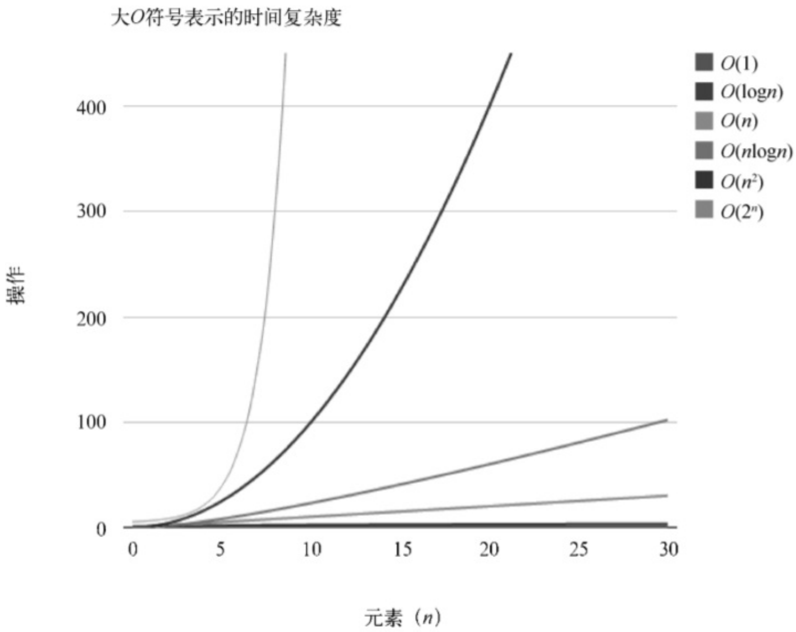
复制代码

如果用大小为10的数组执行bubbleSort，开销是100（102）。如果用大小为100的数组执 行bubbleSort，开销就是10 000（1002）。需要注意，我们每次增加输入的大小，执行都会越来越久

时间复杂度$O(n)$的代码只有一层循环，而$O(n2)$的代码有双层嵌套循环。如 果算法有三层遍历数组的嵌套循环，它的时间复杂度很可能就是$O(n3)$

## 时间复杂度

下图比较了前述各个大$O$符号表示的时间复杂度：



arithmetic21

下表是常用数据结构的时间复杂度

| 数据结构 | 一般情况 | | | 最差情况 | | |
|---|---|---|---|---|---|---|
| | 插入 | 删除 | 搜索 | 插入 | 删除 | 搜索 |
| 数组/栈/队列 | $O(1)$ | $O(1)$ | $O(n)$ | $O(1)$ | $O(1)$ | $O(n)$ |
| 链表 | $O(1)$ | $O(1)$ | $O(n)$ | $O(1)$ | $O(1)$ | $O(n)$ |
| 双向链表 | $O(1)$ | $O(1)$ | $O(n)$ | $O(1)$ | $O(1)$ | $O(n)$ |
| 散列表 | $O(1)$ | $O(1)$ | $O(1)$ | $O(n)$ | $O(n)$ | $O(n)$ |
| 二分搜索树 | $O(\log(n))$ | $O(\log(n))$ | $O(\log(n))$ | $O(n)$ | $O(n)$ | $O(n)$ |
| AVL树 | $O(\log(n))$ | $O(\log(n))$ | $O(\log(n))$ | $O(\log(n))$ | $O(\log(n))$ | $O(\log(n))$ |

arithmetic22

下表是图的时间复杂度：

| 节点/边的管理方式 | 存储空间 | 增加顶点 | 增加边 | 删除顶点 | 删除边 | 轮　询 |
|---|---|---|---|---|---|---|
| 邻接表 | $O(\|V\|+\|E\|)$ | $O(1)$ | $O(1)$ | $O(\|V\|+\|E\|)$ | $O(\|E\|)$ | $O(\|V\|)$ |
| 邻接矩阵 | $O(\|V\|^2)$ | $O(\|V\|^2)$ | $O(1)$ | $O(\|V\|^2)$ | $O(1)$ | $O(1)$ |

arithmetic23

下表是排序算法的时间复杂度：

| 算法（用于数组） | 时间复杂度 | | |
|---|---|---|---|
| | 最好情况 | 一般情况 | 最差情况 |
| 冒泡排序 | $O(n)$ | $O(n^2)$ | $O(n^2)$ |
| 选择排序 | $O(n^2)$ | $O(n^2)$ | $O(n^2)$ |
| 插入排序 | $O(n)$ | $O(n^2)$ | $O(n^2)$ |
| 归并排序 | $O(n\log(n))$ | $O(n\log(n))$ | $O(n\log(n))$ |
| 快速排序 | $O(n\log(n))$ | $O(n\log(n))$ | $O(n^2)$ |
| 堆排序 | $O(n\log(n))$ | $O(n\log(n))$ | $O(n\log(n))$ |
| 桶排序 | $O(n+k)$ | $O(n+k)$ | $O(n^2)$ |
| 基数排序 | $O(nk)$ | $O(nk)$ | $O(nk)$ |

arithmetic24

下表是搜索算法的时间复杂度：

| 算　法 | 数据结构 | 最差情况 |
|---|---|---|
| 顺序搜索 | 数组 | $O(n)$ |
| 二分搜索 | 已排序的数组 | $O(\log(n))$ |
| 深度优先搜索（DPS） | 顶点数为$\|V\|$，边数为$\|E\|$的图 | $O(\|V\|+\|E\|)$ |
| 广度优先搜索（BFS） | 顶点数为$\|V\|$，边数为$\|E\|$的图 | $O(\|V\|+\|E\|)$ |

arithmetic25

## NP

一般来说，如果一个算法的复杂度为O(nk)，其中k是常数，我们就认为这个算法是高效的，这就是多项式算法

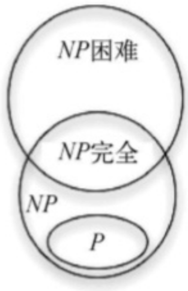
对于给定的问题，如果存在多项式算法，则计为P（polynomial，多项式）

还有一类NP（nondeterministicpolynomial，非确定性多项式）算法。如果一个问题可以在多项式时间内验证解是否正确，则计为NP。如果一个问题存在多项式算法，自然可以在多项式时间内验证其解。因此，所有的P都是NP。然而，P=NP是否成立，仍然不得而知。NP问题中最难的是NP完全问题，它满足以下两个条件：(1)是NP问题，也就是说，可以在多项式时间内验证解，但还没有找到多项式算法；(2)所有的NP问题都能在多项式

时间内归约为它。为了理解问题的归约，考虑两个决策问题L和M。假设算法A可以解决问题L，算法B可以验证输入y是否为M的解。目标是找到一个把L转化为M的方法，使得算法B可以用于构造算法A

还有一类问题，只需满足NP完全问题的第二个条件，称为NP困难问题。因此，NP完全问题也是NP困难问题的子集

下面是满足P < > NP时，P、NP、NP完全和NP困难问题的欧拉图：




arithmetic26

非NP完全的NP困难问题的例子有停机问题和布尔可满足性问题（SAT）。 NP完全问题的例子有子集和问题、旅行商问题、顶点覆盖问题等等

我们提到的有些问题是不可解的。然而，仍然有办法在符合要求的时间内找到一个近似解。启发式算法就是其中之一。启发式算法得到的未必是最优解，但足够解决问题了。启发式算法的例子有局部搜索、遗传算法、启发式导航、机器学习等