



5 用或不用大 O 来优化代码

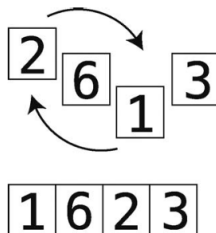
5.1 选择排序

1. 选择排序的例子

(1) 从左至右检查数组的每个格子，找出值最小的那个。在此过程中，我们会用一个变量来记住检查过的数字的最小值（事实上记住的是索引，但为了看起来方便，下图就直接写出数值）。如果一个格子中的数字比记录的最小值还要小，就把变量改成该格子的索引，如图所示。



(2) 知道哪个格子的值最小之后，将该格与本次检查的起点交换。第 1 次检查的起点是索引 0，第 2 次是索引 1，以此类推。下图展示的是第一次检查后的交换动作。



(3) 重复第(1) (2)步，直至数组排好序。

5.2 选择排序实战

以数组[4, 2, 7, 1, 3]为例

第一轮

42713 中检查出1最小，调换4和1，变成12743

第二轮

12743 检查出2最小，不需要调换

第三轮

12743 检查出3最小，调换7和3，变成12347

第四轮

12347 已经完成

5.3 选择排序的实现

```
function selectionSort(array) {  
  for(var i = 0; i < array.length; i++) {  
    var lowestNumberIndex = i;  
    for(var j = i + 1; j < array.length; j++) {  
      if(array[j] < array[lowestNumberIndex]) {  
        lowestNumberIndex = j;  
      }  
    }  
  
    if(lowestNumberIndex !== i) {  
      var temp = array[i];  
      array[i] = array[lowestNumberIndex];  
      array[lowestNumberIndex] = temp;  
    }  
  }  
  return array;  
}
```

以下是进一步的分析

```
for(var i = 0; i < array.length; i++) {
```

这个外层的循环代表每一轮检查。在一轮检查之初，我们会先记住目前的最小值的索引。

```
  var lowestNumberIndex = i;
```

记录最小值的索引，每轮开始时lowestNumberIndex都是起点索引i

```
for(var j = i + 1; j < array.length; j++) {
```

发起一个以 i + 1 开始的内层循环

```
if(array[j] < array[lowestNumberIndex]) {  
    lowestNumberIndex = j;  
}
```

逐个检查未排序的格子，如果碰到比之前记录的本轮最小值还小的值，lowestNumberIndex更新为该格子的索引



```
if(lowestNumberIndex != i) {  
    var temp = array[i];  
    array[i] = array[lowestNumberIndex];  
    array[lowestNumberIndex] = temp;  
}
```

检查最小值是否在正确的位置（该索引是否为i），如果不是，就把i的值和最小值交换

5.4 选择排序的效率

1. 选择排序的步骤主要分为**比较**和**交换**。
2. 以之前5个元素为例。

5个元素的选择排序

 第#轮	 #次比较
<u>1</u>	4
<u>2</u>	3
<u>3</u>	2
<u>4</u>	1

$$\text{sum} = 4+3+2+1 = 10$$

3. 当有N个元素，会有 $(N-1)+(N-2)+(N-3)+\dots+1$ 次**比较**。每轮的**交换**最多只有 1 次。

4. 下表为冒泡排序和选择排序的并列对比。

N 个元素	冒泡排序最多要#步	选择排序最多要#步
5	20	14 (10 次比较 + 4 次交换)
10	90	54 (45 次比较 + 9 次交换)
20	380	199 (180 次比较 + 19 次交换)
40	1560	819 (780 次比较 + 39 次交换)
80	6320	3239 (3160 次比较 + 79 次交换)

选择排序的步数大概只有冒泡排序的一半

5.5 忽略常数

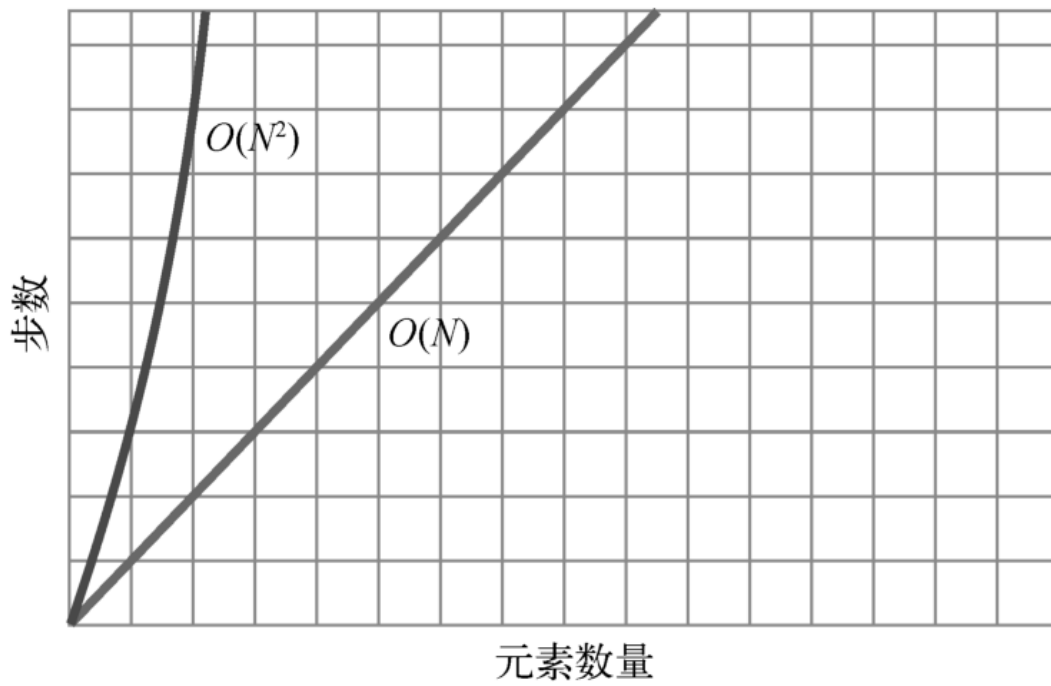
N 个元素	$N^2/2$	选择排序最多要#步
5	$5^2 / 2 = 12.5$	14
10	$10^2 / 2 = 50$	54
20	$20^2 / 2 = 200$	199
40	$40^2 / 2 = 800$	819
80	$80^2 / 2 = 3200$	3239

- 大O 记法忽略常数。不包含一般数字，除非是指数。例如 $O(2N)$ 是 $O(N)$
- 选择排序是 $O(N^2)$

5.6 大O的作用

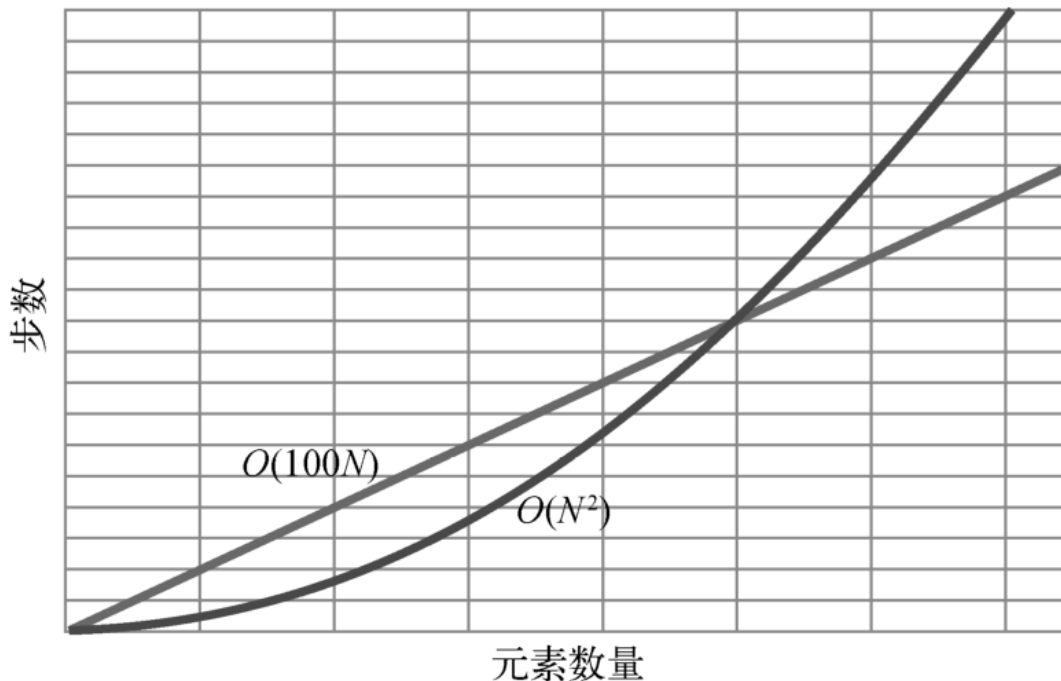
- 能够区分不同算法的**长期增长率**。
- 当数据量**达到一定程度**时， $O(N)$ 的算法就会永远快过 $O(N^2)$

- 下图为 $O(N)$ 和 $O(N^2)$ 的对比



它显示了不管数据量是多少， $O(N)$ 总是快过 $O(N^2)$

- $O(100N)$ 和 $O(N^2)$



一开始 $O(N^2)$ 占优势，数据增加到一定程度， $O(100N)$ 反超

- 大 O 记法忽略常数的原因：大 O 记法只表明，对于不同分类，**存在一临界点**，在这一点之后，一类算法会快于另一类，并永远保持下去。至于这个点在哪里，大 O 并不关心。
- 大 O 记法非常适合用于不同分类下的算法对比（例如 $O(\log N)$ 和 $O(N)$ ）。对于大 O 同类的算法，需要进一步的解析。

5.7 一个实例

- 从一个数组里取出**间隔**的元素，组成新的数组

```
def every_other(array)
  new_array = []

  array.each_with_index do |element, index|
    new_array << element if index.even?
  end
end
```

```
    return new_array
end
```

它迭代原数组的每一个元素，如果元素索引值为偶数，则将该元素插入到新数组里。

步骤分为两种：读取元素(N)和插入新元素(1/2N)

O(N)

- 只读取数组中间隔的元素

```
def every_other(array)
  new_array = []
  index = 0

  while index < array.length
    new_array << array[index]
    index += 2
  end

  return new_array
end
```

这种做法的 while 循环会跳过间隔的元素，因此避免了检查每个元素。

当有N个元素，会进行1/2N次读取和1/2N次插入。