浅谈时间复杂度

原创 王一华 LOA算法学习笔记 2021-03-07 17:30

00 前言

在我们分析算法的时候，时间复杂度 (Time complexity) 总是一个绕不开的重要指标，在学习过程中，我曾有过很多无知的困惑，最后我整理了其相关定义、计算方法、实际性能等方面的知识分享在这里。浅薄之谈，有不当之处请大家批评指正。

01 究竟何为时间复杂度

T(n)：一般情况下，算法中基本操作重复执行的次数是问题规模n的某个函数，用 T(n) 表示；

T(n)的含义简单来说就是：n为输入数据的大小或数量，T(n)即当输入数据量为n时，某段代码的总执行次数。

但当代码行数比较多的时候，一条一条的数语句就有些麻烦了，这个时候我们就用T(n)简化的估计值来替代。

O(n)也是一个函数，它表示渐进时间复杂度，又叫大O表示法。

若有某个辅助函数f(n)，使得当n趋近于无穷大时，T(n)/f(n)的极限值为不等于零的常数，则称f(n)是T(n)的同数量级函数。记作T(n) = O(f(n))，称 O(f(n)) 为算法的渐进时间复杂度，简称时间复杂度。数学语言表达就是：存在足够大的正整数M，使得T(n) ≤ M × T(n)。

T(n)=O(f(n))，这是大O表示的渐进时间复杂度，它表示随着问题规模n的增长，算法执行时间的增长率是相同的，而且f(n)比T(n)更为简洁，f(n)由T(n)"去粗取精"（低次项忽略，常系数忽略）得到。

三种复杂度指标：

$$O(BigO)\text{：最差情况}$$

$$\Omega(BigOmega)\text{：最好情况}$$

$$\Theta(BigTheta)\text{：一个算法的区间}$$

02 时间复杂度的分析计算方法总结

(1) 分治算法中的Master Theorem

主定理(Master Theorem)提供了用于分析一类有递归结构算法时间复杂度的方法。这种递归算法通常有这样的结构：

```
def solve(problem):
    solve_without_recursion()
    for subProblem in problem:
        solve(subProblem)
```

通俗语言描述即：规模为 n 的问题，把它拆分成 a 个子问题，每个子问题的size为 n/b，combine/merge的时间复杂度为 n 的 d 次方。其中 a 和 b 不一定相等。

Theorem

Let $T(n)$ *be defined by* $T(n) = aT(\frac{n}{b}) + O(n^d)$ *for* $a > 1$, $b > 1$ *and* $d > 0$, *then* $T(n)$ *can be bounded by:*

1. *If* $d < \log_b a$, *then* $T(n) = O(n^{\log_b a})$;
2. *If* $d = \log_b a$, *then* $T(n) = O(n^{\log_b a} \log n)$;
3. *If* $d > \log_b a$, *then* $T(n) = O(n^d)$.

例如Merge-Sort算法里面，我们将规模为的**n**排序问题先转换为 2 个规模为 $\frac{n}{2}$ 的子问题，然后合并这两个子问题需要花费O(n)的时间，因此其时间复杂度的递归公式为 $T(n) = 2T(\frac{n}{2}) + O(n)$ 。

(2) 动态规划问题如何分析时间复杂度

时间复杂度取决于状态转移方程，如果第i个状态的确定需要利用前i-1个状态，即dp[i]由dp[i-1],dp[i-2],...,dp[0]的取值共同决定，那么此时的时间复杂度为 $O(n^2)$ 。动态规划的时间复杂度可简单做一个估计：子问题个数 ✖ 每个子问题需要的时间，例如：

- OPT（i，j）：分析i和j的取值范围，设其为 $O(n^2)$ 的复杂度，若每个子问题中都有for循环，则总的时间复杂度约为 n 的三次方。

动态规划有两种等价的实现方法，一是自顶向下法 (Top-Down With Memoization)，按照自然的递归形式实现，但过程中开辟空间保存每个子问题的解。当需要一个子问题的解时，先检查是否已经保存过此解，如果存在解，则直接返回保存的值，从而达到了节省计算时间的目的；若不存在则需按照流程计算这个子问题。

二是自底向上法 (Bottom-Up Method)。这种方法需要恰当的定义一个子问题，使得任意子问题的求解都只依赖于"更小的"子问题的求解，故而可以将子问题按照规模排序，按由小至大的顺序求解，每个子问题只需要求解一次，当求解某个子问题时，它所依赖的那些更小的子问题都已求解完毕，已经保存的结果就可以直接拿过来用了。

比如斐波那契数列的求解：如果用分治，不加任何缓存，则时间复杂度和空间复杂度会高达 $O(2^n)$ 。由于存在重复子问题，动态规划是更合适的做法，时间复杂度和空间复杂度都可以降低到O(n)。代码如下：

```
#include<stdio.h>
int F(int n)
{
    if(n == 1||n == 2)
     {
         return 1;
      }
    return F(n-1) + F(n-2);
}
int main(void)//分治法求解斐波那契//
{
    int n;
    while(scanf("%d", &n) != EOF)
    {
        printf("%d\n", F(n));
    }
    return 0;
}
```

```c
#include<stdio.h>
int main(void)//动态规划求解斐波那契//
{
  long n, *a, i;
  while(scanf("%ld", &n) != EOF)
  {
    a = malloc((n + 1) * sizeof(int));
    a[0] = 0;
    a[1] = 1;
    for(i=2; i<=n; i++)
    {
      a[i] = a[i-1] + a[i-2];
    }
    printf("%d\n", a[n]);
  }
  return 0;
}
```

03 实际应用中的时间性能

除了从理论角度分析算法的时间复杂度，在实际应用中，电脑一开，代码一跑，运行结果往往会出乎我们的意料。分析实际影响时间性能的因素，主要有以下几点：

首先是编程语言的选择不同，效率不同：比如建立索引，用哈希表存储，动态开辟空间等问题的处理方式不同，在规模比较小的问题上，效率差别就已经很明显了。比如c++，个人觉得用vector虽然方便，但是效率并不高。

其次，不同的数据类型、存储方式的选择等也会影响算法实现的效率。比如，在串和队列的有关操作中用堆操作合适，在树的操作中用栈操作合适。还有优先队列、滑动窗口等编程技巧的应用，也会在一定程度范围内提高算法的执行效率。

还有一些看起来“不太重要”但很有趣的细微之处，以快速排序为例，partition部分有两种实现方式，如何实现递归、正确终止递归，在不开辟新数组空间的情况下，如何实现（与pivot比较后）大小数的交换，即为了节省空间而牺牲了一点点时间，用来进行pivot两边必须的交换数字的操作。

- **第一种是：**

1. 先走right，右边找到小于pivot的值时，right指针停下；

（一定要先走right，因为它遇到比pivot小的值停下，最后 left 和 right 遇到的时候，可以拿二者指向的值与pivot交换，先走right可以保证它一定小于pivot。）

2. 再走left，左边找到大于pivot的值时，left指针停下；
3. 此时如果 left < right，交换二者；
4. 循环上述三步，直到 left 和right 相遇；
5. 交换当前 left（right）指针指向的位置与初始first位置的数值。

代码如下：

```c
int partition(int *nums, int left, int right)
{
    int pivot = nums[left];
    int first = left;
        while (left < right) {
            while (left < right & nums[right] >= pivot) {
                right--;
```

```
            }
            while (left < right & nums[left] <= pivot) {
                left++;
            }
            if(left < right){
                int tem = nums[right];
                nums[right] = nums[left];
                nums[left] = tem;
            };
        }
        nums[first] = nums[left];
        nums[left] = pivot;
        return left;
}
```

- **第二种是：**

1. 选取 left 指针指向的值为pivot（首个元素）；
2. 先走 right，遇到比pivot小的值，停下指针，将这个值赋给 left 指针指向的元素；（此时，left 的值已经赋值给了pivot，所以直接覆盖没问题）
3. 再走 left，遇到比 pivot 大的值，就停下指针。将 left 中的值赋给第2步中 right 指针指向的元素；（第2步中，刚给 left 赋的值是一定小于pivot的）
4. 循环执行2、3步，直至 left 和 right 指针相遇；
5. 至此，2、3两步就完成了交换一大一小两个值，然后最开始 0 位置的值存在 pivot中，将 pivot 放到排序之后的枢纽位置，即 left 和 right 指针指向的位置。

代码如下：

```
int partition(vector<int> &nums, int left, int right) {
        int pivot = nums[left];
        while (left < right) {
            while (left < right & nums[right] >= pivot) {
                right--;
            }
            nums[left] = nums[right];
            while (left < right & nums[left] < pivot) {
                left++;
            }
            nums[right] = nums[left];
        }
        nums[left] = pivot;
        return left;
}
```

经过多次测试，结果表明，同样测试用例的情况下，第二种实现方式的用时更短。（具体为啥我也没彻底搞明白，可能也没啥价值，就是一个小小的交换策略......）

而当对代码继续优化，通过上课讲解的随机选取pivot的方法，效率又有了小幅度的（还挺明显的）提升：

```
public void quickSort(int[] nums,int start, int end, int target) {
    if (start < end) {
```

```java
        int i = start, j = end;
        int random =  new Random().nextInt(j - i + 1) + i;
        int temp = nums[i];
        nums[i] = nums[random];
        nums[random] = temp;
        int vot = nums[i];
        while (i != j) {
            while (i < j && nums[j] >= vot) j--;
            if (i < j) nums[i++] = nums[j];
            while (i < j && nums[i] <= vot) i++;
            if (i < j) nums[j--] = nums[i];


        }
        nums[i] = vot;
        if (i == nums.length - target)
            return ;
        else if (i < nums.length - target)
            quickSort(nums, i + 1, end, target);
        else
            quickSort(nums, start, i - 1,target);
    }
}
public int findKthLargest(int[] nums, int k) {
    quickSort(nums, 0, nums.length - 1, k);
    return nums[nums.length - k];
}
```