

# Data Structures and Algorithms

## Running time, Divide and Conquer

January 26, 2016

---

**Example.** Consider the following code fragment.

```
for (i = 0; i < n; i++)
    for (j = 0; j < i; j=j+10)
        print ("run time analysis")
```

Give a tight bound on the running time of this code fragment.

**Solution.** For each value of  $i$ , the body of the inner loop executes  $i/10$  times. Thus the running time of the body of the outer loop is at most  $c(i/10)$ , for some positive constant  $c$ . Hence the total running time of the code fragment is given by

$$\sum_{i=0}^{n-1} c \lceil \frac{i}{10} \rceil = \sum_{i=0}^{n-1} c \left( \frac{i}{10} + 1 \right) = \frac{c(n-1)n}{20} + cn \leq 2cn^2 = O(n^2)$$

We will now show that  $\sum_{i=0}^{n-1} c \lceil \frac{i}{10} \rceil = \Omega(n^2)$ . Note that

$$\sum_{i=0}^{n-1} c \lceil \frac{i}{10} \rceil \geq \sum_{i=0}^{n-1} \frac{ci}{10} = c(n-1)n/20$$

We want to find positive constants  $c'$  and  $n_0$ , such that for all  $n \geq n_0$ ,

$$\frac{c(n-1)n}{20} \geq c'n^2$$

This is equivalent to showing that  $n(c - 20c') \geq c$ . This is true when  $c' = c/40$  and  $n \geq 2$ . Thus, the running time of the code fragment is  $\Omega(n^2)$ .

**Example.** Consider the following code fragment.

```
i = n
while (i >= 10) do
    i = i/3
    for j = 1 to n do
        print ("Inner loop")
```

What is an upper-bound on the running time of this code fragment? Is there a matching lower-bound?

**Solution.** The running time of the body of the inner loop is  $O(1)$ . Thus the running time of the inner loop is at most  $c_1 n$ , for some positive constant  $c_1$ . The body of the outer loop takes at most  $c_2 n$  time, for some positive constant  $c_2$  (note that the statement  $i = i/3$  takes  $O(1)$  time). Suppose the algorithm goes through  $t$  iterations of the while loop. At the end of the last iteration of the while loop, the value of  $i$  is  $n/3^t$ . We know that the code fragment surely finishes when  $n/3^t \leq 1$ , solving which gives us  $t \geq \log_3 n$ . This means that the number of iterations of the while loop is at most  $O(\log n)$ . Thus the total running time is  $O(n \log n)$ .

We will now show that the running time is  $\Omega(n^2)$ . We will lower-bound the number of iterations of the outer loop. Note that when the value of  $i$  is more than 10 (say,  $3^3$ ), the outer loop has not terminated. Solving  $n/3^t \geq 3^3$ , gives us that  $\log_3 n - 3$  is a lower bound on the number of iterations of the outer loop. For each iteration of the outer loop, the inner loop runs  $n$  times. Thus the total running time is at least  $cn(\log_3 n - 3)$ , for some positive constant  $c$ . Note that  $cn(\log_3 n - 3) \geq c'n \log n$ , when  $c' = c/2$  and  $n \geq 3^6$ . Thus the running time is  $\Omega(n \log n)$ .

**Example.** Consider the following code fragment.

```
for i = 0 to n do
  for j = n to 0 do
    for k = 1 to j-i do
      print (k)
```

What is an upper-bound on the running time of this algorithm? What is the lower bound?

**Solution.** Note that for a fixed value of  $i$  and  $j$ , the innermost loop goes through  $\max\{0, j - i\} \leq n$  times. Thus the running time of the above code fragment is  $O(n^3)$ .

To find the lower bound on the running time, consider the values of  $i$ , such that  $0 \leq i \leq n/4$  and values of  $j$ , such that  $3n/4 \leq j \leq n$ . Note that for each of the  $n^2/16$  different combinations of  $i$  and  $j$ , the innermost loop executes at least  $n/2$  times. Thus the running time is at least

$$(n^2/16)(n/2) = \Omega(n^3)$$

**Discussion:** Consider a problem  $X$  with an algorithm  $A$ .

- Algorithm  $A$  runs in time  $O(n^2)$ . This means that the worst case asymptotic running time of algorithm  $A$  is upper-bounded by  $n^2$ . Is this bound tight? That is, is it possible that the run-time analysis of algorithm  $A$  is loose and that one can give a tighter upper-bound on the running time?
- Algorithm  $A$  runs in time  $\Theta(n^2)$ . This means that the bound is tight, that is, a better (tighter) bound on the worst case asymptotic running time for algorithm  $A$  is not possible.
- Problem  $X$  takes time  $O(n^2)$ . This means that there is an algorithm that solves problem  $X$  on *all* inputs in time  $O(n^2)$ .

- Problem  $X$  takes  $\Theta(n^{1.5})$ . This means that there is an algorithm to solve problem  $X$  that takes time  $O(n^{1.5})$  and no algorithm can do better.

Consider the problem of computing  $2^n$  for any non-negative integer  $n$ . Below are four similar looking algorithms to solve this problem.

```
powerof2(n)
  if n = 0
    return 1
  else
    return 2 * powerof2(n-1)
```

```
powerof2(n)
  if n = 0
    return 1
  else
    return powerof2(n-1)+ powerof2(n-1)
```

```
powerof2(n)
  if n = 0
    return 1
  else
    tmp = powerof2(n-1)
    return tmp + tmp
```

```
powerof2(n)
  if n = 0
    return 1
  else
    tmp = powerof2(floor(n/2))
    if (n is even) then
      return tmp * tmp
    else
      return 2 * tmp * tmp
```

The recurrence for the first and the third method is  $T(n) = T(n-1) + O(1)$ . The recurrence for the second method is  $T(n) = 2T(n-1) + O(1)$ , and the recurrence for the last method is  $T(n) = T(n/2) + c$  (assuming that  $n$  is a power of 2). In all cases the base case is  $T(0) = 1$ . We solve both these recurrences below. The recurrence for the first and the third method can be solved as follows.

$$\begin{aligned}
T(n) &= T(n-1) + c \\
&= T(n-2) + 2c \\
&= T(n-3) + 3c \\
&\dots \\
&\dots \\
&= T(n-k) + kc
\end{aligned}$$

The recursion bottoms out when  $n - k = 0$ , i.e.,  $k = n$ . Thus, we get

$$\begin{aligned}
T(n) &= T(0) + nc \\
&= 1 + nc \\
&= \Theta(n)
\end{aligned}$$

The recurrence for the second method can be solved as follows.

$$\begin{aligned}
T(n) &= 2T(n-1) + c \\
&= 2^2T(n-2) + (2^0 + 2^1)c \\
&= 2^3T(n-3) + (2^0 + 2^1 + 2^2)c \\
&\dots \\
&\dots \\
&= 2^kT(n-k) + c \sum_{i=0}^{k-1} 2^i
\end{aligned}$$

The recursion bottoms out when  $n - k = 0$ , i.e.,  $k = n$ . Thus, we get

$$\begin{aligned}
T(n) &= 2^nT(0) + c \sum_{i=0}^{n-1} 2^i \\
&= 2^n + c(2^n - 1) \\
&= \Theta(2^n)
\end{aligned}$$

The recurrence for the fourth method can be solved as follows.

$$\begin{aligned}
T(n) &= T(n/2) + c \\
&= T(n/2^2) + 2c \\
&= T(n/2^3) + 3c \\
&\dots \\
&\dots \\
&= T(n/2^k) + kc
\end{aligned}$$

The recursion bottoms out when  $n/2^k < 1$ , i.e., when  $k > \lg n$ . Thus, we get

$$\begin{aligned} T(n) &= T(0) + c(\lg n + 1) \\ &= 1 + \Theta(\lg n) \\ &= \Theta(\lg n) \end{aligned}$$