

# CSC165

## RUNNING TIME OF PROGRAMS

GARY BAUMGARTNER

Consider the following algorithm:

```
// Return the first index of x in the array A
// or -1 if x isn't in A.
LS(A, x)
  i := 0
  while i < A.length
    if A[i] = x
      return i
    i := i + 1
  return -1
```

The time taken for the above code depends not just on  $A$  and  $x$ , but on how long assignment, addition, array access, etc, take for the particular computer, compiler, etc. We start our analysis by counting the operations, since they each take some amount of time:

```
// Return the first index of x in the array A
// or -1 if x isn't in A.
LS(A, x)
  i := 0           // 3 steps
  while i < A.length // 6 steps
    if A[i] = x    // 6 steps
      return i     // 2 steps
    i := i + 1    // 5 steps
  return -1       // 2 steps
```

Each variable access (e.g.  $i$ ), constant use (e.g.  $-1$ ), assignment ( $:=$ ), **while** or **if** branching, comparison (e.g.  $<, =$ ), array access ( $[\ ]$ ), arithmetic operation (e.g.  $+$ ), object access ( $.$ ) or **return** counts as a step.

The number of steps depends only on the input, so let  $t_{LS}(A, x)$  = the number of steps to execute LS with input  $A$  and  $x$ . For example,  $t_{LS}([3, 1, 4], 3) = 3 + 6 + 6 + 2 = 17$ , since it executes the first three lines, and returns on the fourth.

More generally, suppose  $x$  is in  $A$  and let  $j$  be the first index of  $x$  in  $A$ . The first line is executed (taking 3 steps). The loop iterates fully  $j$  times (taking  $17j$  steps), and then it iterates partially one time to return  $j$  (taking 14 steps). So  $t_{LS}(A, x) = 3 + 17j + 14 = 17(j + 1)$ .

Suppose  $x$  is not in  $A$ . Then the first line is executed (3 steps), the loop iterates fully  $A.length$  times ( $17A.length$  steps), checks the `while` condition and fails (6 steps) and returns -1 (2 steps). In this case:  $t_{LS}(A, x) = 11 + 17A.length$ . Thus:

$$t_{LS}(A, x) = \begin{cases} 17(j+1), & j \text{ 1st index of } x \text{ in } A \\ 11 + 17A.length, & x \text{ not in } A. \end{cases}$$

How does this depend on the length of  $A$ ? One worthwhile measure is the worst-case: for  $n \in N$ , let  $T_{LS}(n) = \max\{t_{LS}(A, x) : A.length = n\}$ . In our formula for  $t_{LS}$ ,  $j < n$ , so the maximum is when  $x$  is not in  $A$ :  $T_{LS}(n) = 11 + 17n$ .

What's the actual *time* taken in the worst-case? Not every type of operation takes the same amount of time (and may even vary within the same program). But consider the minimum and maximum times  $m$  and  $M$  for all the operations: then the actual time is between  $(11 + 17n)m$  and  $(11 + 17n)M$ , so it's  $\Theta(n)$ . Counting steps lets us determine the actual time in terms of  $\Theta$ .

### WORST-CASE

In general, for a program  $P$  and input  $z$  we let  $t_P(z)$  = the number of steps to execute  $P$  on  $z$ . It can be complicated to calculate  $t_P$  exactly, and often we're just interested in the worst-case. The worst-case requires choosing some measure of size for the input, and then the worst-case is defined for  $n \in N$  by  $T_P(n) = \max\{t_P(z) : z \text{ is an input of size } n\}$ .  $T_P$  can still be too complicated to calculate exactly, but we may still be able to reason about its  $O$  and  $\Omega$  behaviour.

Reasoning about  $T_P$  in terms of  $O$  or  $\Omega$  requires comparing a maximum with an upper or lower bound. Notice that  $\max\{x \in D : q(x)\} \leq u$  iff  $\forall x \in D, q(x) \rightarrow x \leq u$ , and  $\max\{x \in D : q(x)\} \geq l$  iff  $\exists x \in D, q(x) \wedge x \geq l$ . Letting  $I$  be the set of input for  $P$ , we can rewrite  $T_P \in O(U)$  as:

$$\begin{aligned} \exists c \in R^+, \exists b \in N, \forall n \in N, n \geq b \rightarrow \max\{t_P(z) : z \text{ is an input of size } n\} &\leq U(n) \\ \exists c \in R^+, \exists b \in N, \forall n \in N, n \geq b \rightarrow \forall z \in I, \text{size}(z) = n \rightarrow t_P(z) &\leq U(n) \\ \exists c \in R^+, \exists b \in N, \forall z \in I, \text{size}(z) \geq b \rightarrow t_P(z) &\leq U(\text{size}(z)). \end{aligned}$$

And we can rewrite  $T_P \in \Omega(L)$  as:

$$\begin{aligned} \exists c \in R^+, \exists b \in N, \forall n \in N, n \geq b \rightarrow \max\{t_P(z) : z \text{ is an input of size } n\} &\geq L(n) \\ \exists c \in R^+, \exists b \in N, \forall n \in N, n \geq b \rightarrow \exists z \in I, \text{size}(z) = n \wedge t_P(z) &\geq L(n) \end{aligned}$$

Let's apply this for the following program that implements insertion sort:

```
// A is an array.
IS(A)
  i := 1
  while i < A.length
    t := A[i]
    j := i
    while j > 0 and A[j-1] > t
      A[j] := A[j-1]
      j := j-1
    A[j] := t
```

Measuring size by  $A.length$ , we claim that  $T_{IS}(n) \in \Theta(n^2)$ . We show this by showing  $T_{IS}(n) \in O(n^2)$  and  $T_{IS}(n) \in \Omega(n^2)$ , using our formulations above.

For  $T_{IS}(n) \in O(n^2)$  we only need an upper bound, so we'll put some easy upper bounds on the lines of the algorithm:

```

i := 1                                // <= 10 steps
while i < A.length                    // <= 10 steps
  t := A[i]                            // <= 10 steps
  j := i                               // <= 10 steps
  while j > 0 and A[j-1] > t          // <= 20 steps
    A[j] := A[j-1]                   // <= 20 steps
    j := j-1                          // <= 10 steps
  A[j] := t                            // <= 10 steps

```

Here's the proof:

Let  $c = 140$ ,  $b = 1$ . Then  $c \in R^+$  and  $b \in N$ .

Let  $n \in N$ .

Let  $A$  be an array. Suppose  $A.length \geq b$ .

Let  $n = A.length$ . Note that  $n \geq b \geq 1$ .

The outer loop iterates  $n$  times.

The inner loop iterates  $\leq i$  times, so  $\leq n$  times.

Each iteration of the inner loop takes  $\leq 50$  steps, so the loop, counting the final test to end the loop, takes  $\leq 50n + 20 \leq 70n$  steps (since  $n \geq b = 1$ ).

Each iteration of the outer loop takes  $\leq 40 + 70n \leq 110n$  steps, so the outer loop, counting the final test, takes  $\leq (110n)n + 10 \leq 120n^2$  steps.

So  $t_{IS}(n) \leq 20 + 120n^2 \leq 140n^2 = cn^2$ .

[The conclusions are left as an exercise.]

For  $T_{IS}(n) \in \Omega(n^2)$  we need, for each sufficiently large length, an array that takes quadratic time to process. Any array sorted in decreasing will do:

Let  $c = 1/2$  and  $b = 2$ . Then  $c \in R^+$  and  $b \in N$ .

Let  $n \in N$ . Suppose  $n \geq b$ .

Let  $A = [n, n-1, n-2, \dots, 2, 1]$ . Then  $A$  is an array.

Then  $A.length = n$ .

The outer loop iterates  $n-1$  times.

For the inner loop, the elements  $A[0], A[1], \dots, A[j-1]$  are a rearrangement of the original values  $n, n-1, \dots, n-(j-1)$ , and in particular are larger than  $t = n-j$ . So the inner loop iterates  $i$  times, down to  $j = 0$ .

The “ $j := j-1$ ” takes  $\geq 2$  steps, so the inner loop takes at least  $2i$  steps.

So the outer loop takes  $\geq 2 + \dots + 2(n-1) = 2(1 + \dots + (n-1)) = n^2 - n$  steps.

Thus  $t_{IS}(A) \geq n^2 - n = \frac{n^2}{2} + \frac{n^2}{2} - n = cn^2 + n(\frac{n}{2} - 1) \geq cn^2 + n(\frac{2}{2} - 1) = cn^2$ .

[The conclusions are left as an exercise.]