Chapter 6

Correctness, running time of programs

So far we have been proving statements about databases, mathematics and arithmetic, or sequences of numbers. Though these types of statements are common in computer science, you'll probably encounter algorithms most of the time. Often we want to reason about algorithms and even prove things about them. Wouldn't it be nice to be able to *prove* that your program is correct? Especially if you're programming a heart monitor or a NASA spacecraft?

In this chapter we'll introduce a number of tools for dealing with computer algorithms, formalizing their expression, and techniques for analyzing properties of algorithms, so that we can prove correctness or prove bounds on the resources that are required.

6.1 Binary (base 2) notation

Let's first think about numbers. In our everyday life, we write numbers in decimal (base 10) notation (although I heard of one kid who learned to use the fingers of her left hand to count from 0 to 31 in base 2). In decimal, the sequence of digits 20395 represents (parsing from the right):

$$5 + 9(10) + 3(100) + 0(1000) + 2(10000) \quad =$$

$$5(10^0) + 9(10^1) + 3(10^2) + 0(10^3) + 2(10^4)$$

Each position represents a power of 10, and 10 is called the BASE. Each position has a digit from $[0, 9]$ representing how many of that power to add. Why do we use 10? Perhaps due to having 10 fingers (however, humans at various times have used base 60, base 20, and mixed base 20,18 (Mayans)). In the last case there were $(105)_{20,18}$ days in the year. Any integer with absolute value greater than 1 will work (so experiment with base -2).

Consider using 2 as the base for our notation. What digits should we use?[1] We don't need digits 2 or higher, since they are expressed by choosing a different position for our digits (just as in base 10, where there is no single digit for numbers 10 and greater).

Here are some examples of binary numbers:

$$(10011)_2$$

represents

$$1(2^0) + 1(2^1) + 0(2^2) + 0(2^3) + 1(2^4) = (19)_{10}$$

We can extend the idea, and imitate the decimal point (with a "binary point"?) from base 10:

$$(1011.101)_2 = 19\frac{5}{8}$$

How did we do that?[2] Here are some questions:

- How do you multiply two base 10 numbers?[3] Work out 37×43.
- How do you multiply two binary numbers?[4]
- What does "right shifting" (eliminating the right-most digit) do in base 10?[5]
- What does "right shifting" do in binary?[6]
- What does the rightmost digit tell us in base 10? In binary?

Convert some numbers from decimal to binary notation. Try 57. We'd like to represent 57 by adding either 0 or 1 of each power of 2 that is no greater than 57. So $57 = 32 + 16 + 8 + 1 = (111001)_2$. We can also fill in the binary digits, systematically, from the bottom up, using the % operator (the remainder after division operator, at least for positive arguments):

$$\begin{aligned}
57\%2 = 1 &\quad \textit{so} \quad (?????1)_2 \\
(57-1)/2 = 28\%2 = 0 &\quad \textit{so} \quad (????01)_2 \\
28/2 = 14\%2 = 0 &\quad \textit{so} \quad (???001)_2 \\
14/2 = 7\%2 = 1 &\quad \textit{so} \quad (??1001)_2 \\
(7-1)/2 = 3\%2 = 1 &\quad \textit{so} \quad (?11001)_2 \\
(3-1)/2 = 1\%2 = 1 &\quad \textit{so} \quad (111001)_2
\end{aligned}$$

Addition in binary is the same as (only different from...) addition in decimal. Just remember that $(1)_2 + (1)_2 = (10)_2$. If we add two binary numbers, this tells us when to "carry" 1:

$$\begin{array}{rr}
 & 1011 \\
+ & 1011 \\
\hline
 & 10110
\end{array}$$

LOG$_2$

How many 5-digit binary numbers are there (including those with leading 0s)? These numbers run from $(00000)_2$ through $(11111)_2$, or 0 through 31 in decimal — 32 numbers. Another way to count them is to consider that there are two choices for each digit, hence 2^5 strings of digits. If we add one more digit we get twice as many numbers. Every digit doubles the range of numbers, so there are two 1-digit binary numbers (0 and 1), four 2-digit binary numbers (0 through 3), 8 3-digit binary numbers (0 through 7), and so on.

Reverse the question: how many digits are required to represent a given number. In other words, what is the smallest integer power of 2 needed to exceed a given number? $\log_2 x$ is the power of 2 that gives $2^{\log_2 x} = x$. You can think of it as how many times you must multiply 1 by 2 to get x, or roughly the number of digits in the binary representation of x. (The precise number of digits needed is $\lfloor (\log_2 x) + 1 \rfloor$, which is equal to (why?) $\lfloor \log_2 x \rfloor + 1$).

6.2 Proving the correctness of programs

We often want to write computer programs that are correct: programs that actually do what we want them to do. (Unfortunately a lot of software and hardware out there don't work correctly, and you have probably experienced the frustrations of dealing with incorrect programs.) We want to be sure that our program works correctly. Ideally, we would like to be able to prove to someone that our program is correct, especially if someone's life might depend on it. We will now demonstrate some of the tools we can use to prove facts about a program by examining a basic example.

Loop invariant for base 2 multiplication

Integers are naturally represented on a computer in binary, since a gate can be in either an on or off (1 or 0) position. It is very easy to multiply or divide by 2, since all we need to do is perform a left or right shift (an easy hardware operation). Similarly, it is also very easy to determine whether an integer is even or odd.

Putting these together, we can write a multiplication algorithm that uses these fast operations:

```
public class MultiplicationExample {

  /**
   * mult multiplies m times n.
   * @arg m a natural number
   * @arg n an integer
   * @return mn
   * precondition: m >= 0
   */
  public static int mult(int m, int n) {
    int x = m;
    int y = n;
    int z = 0;
    // loop invariant: z = mn - xy
    while (x != 0) {
      if (x % 2 == 1) { // x odd
        z = z + y;
      }
      x = x >> 1; // x = x div 2 (right shift)
      y = y << 1; // y = 2y (left shift)
    }
    // post condition: z = mn
    return z;
  }

}
```

After reading this algorithm, there is no reason you should believe it actually multiplies two integers: we'll need to prove it to you. Let's consider the precondition first. We can always ensure that $m \geq 0$ (how?[7]). The postcondition states that z, the value that is returned, is equal to the product of m and n (that would be nice, but we're not convinced).

Let's look at the stated loop invariant. A loop invariant is a relationship between the variables that is always true at the start and at the end of a loop iteration (we'll need to prove this). It's sufficient to verify that the invariant is true at the start of first iteration, and verify that if the invariant is true at the start of any iteration, it must be true at the end of the iteration.[8] Before we start the loop, we set $x = m$, $y = n$ and $z = 0$, so it is clear that $z = mn - xy = mn - mn = 0$. Now we need to show that if $z = mn - xy$ before executing the body of the loop, and $x \neq 0$, then after executing the loop body, $z = mn - xy$ is still true (can you write this statement formally?). Here's a sketch of a proof:

> Let $x', y', z', x'', y'', z'', m, n \in \mathbb{Z}$, and assume the $'$ elements related to the $''$ elements by the action of the loop. Assume $m \geq 0$. Observe that the values of m and n are never changed in the loop.
>
> Assume $z' = mn - x'y'$.
>
> Case 1: x' odd.
>
> Then $z'' = z' + y', \quad x'' = (x' - 1)/2, \quad \text{and } y'' = 2y'$.
>
> So
>
> $$\begin{aligned} mn - x''y'' &= mn - (x'-1)/2 \cdot 2y' \quad \text{(since } x' \text{ is odd)} \\ &= mn - x'y' + y' \\ &= z' + y' \\ &= z'' \end{aligned}$$
>
> Case 2: x' even.
>
> Then $z'' = z', \quad x'' = x'/2, \quad \text{and } y'' = 2y'$.
>
> So
>
> $$\begin{aligned} mn - x''y'' &= mn - x'/2 \cdot 2y' \\ &= mn - x'y' \\ &= z' \\ &= z'' \end{aligned}$$
>
> Since x' is either even or odd, in all cases $mn - x''y'' = z''$
>
> Thus $mn - x'y' = z' \Rightarrow mn - x''y'' = z''$.
>
> Since $x', x'', y', y'', z', z'', m, n$ are arbitrary elements of $\mathbb{Z}$,
>
> $\forall x', x'', y', y'', z', z'', m, n \in \mathbb{Z}, mn - x'y' = z' \Rightarrow mn - x''y'' = z''$.

We should probably verify the postcondition to fully convince ourselves of the correctness of this algorithm. We've shown the loop invariant holds, so let's see what we can conclude when the loop terminates (i.e., when $x = 0$). By the loop invariant, $z = mn - xy = mn - 0 = mn$, so we know we must get the right answer (assuming the loop eventually terminates).

We should now be fairly convinced that this algorithm is in fact correct.[9] One might now wonder, how many iterations of the loop are completed before the answer is returned?

6.3 Expressing the running time of programs

For any program P and any input x, let $t_P(x)$ denote the number of "steps" P takes on input x. We need to specify what we mean by a "step." A "step" typically corresponds to machine instructions being executed, or some indication of time or resources expended.

Consider the following (somewhat arbitrary) accounting for common program steps:

METHOD CALL: 1 step + steps to evaluate each argument, + steps to execute the method.

RETURN STATEMENT: 1 step + steps to evaluate return value.

IF STATEMENT: 1 step + steps to evaluate condition.

ASSIGNMENT STATEMENT: 1 step + steps to evaluate each side.

ARITHMETIC, COMPARISON, BOOLEAN OPERATORS: 1 step + steps to evaluate each operand.

ARRAY ACCESS: 1 step + steps to evaluate index.

MEMBER ACCESS: 2 steps.

CONSTANT, VARIABLE EVALUATION: 1 step.

Let's use this accounting scheme to study the running time of a familiar algorithm.

6.3.1 Linear search

Let's consider the following implementation of linear search as an example.

```
// A is an array, x is an element to search for.
// Return an index i such that A[i] = x;
// if there is no such index, return −1.
// Convention: array indices start at 0

   LS (A, x) {
1.    i = 0;                      // 3 steps (evaluate variable, constant, assignment)
2.    while (i < A.length) {      // 5 steps (while, A.length (2 steps), i, <)
3.       if (A[i] == x) {         // 5 steps (A[i], ==, x, if)
4.          return i;             // 2 steps (return, i)
5.       }
6.       i = i + 1;               // 5 steps (i, assignment, i, +, 1)
7.    }
8.    return -1;                  // 2 steps (return, −1)
9. }
```

Now let's trace a function call, `LS([2,4,6,8],4)`:

Line 1: 3 steps ($i = 0$)

Line 2: 5 steps ($0 < 4$)

Line 3: 5 steps ($A[0] == 4$)

Line 6: 5 steps ($i = 1$)

Line 2: 5 steps ($1 < 4$)

Line 3: 5 steps ($A[1] == 4$)

Line 4: 2 (return 1)

So $t_{LS}([2,4,6,8],4) = 30$. Notice that if the first index where x is found is j, then $t_{LS}(A,x)$ will count lines 2, 3, and 6 once for each index from 0 to $j-1$ (j indices), and then count lines 2, 3, 4 for index j, and so $t_{LS}(A,x)$ will be $3 + 15j + 12 = 15(j+1)$.

If x does not appear in A, then $t_{LS}(A,x) = 3 + 15A.length + 7 = 15A.length + 10$, because line 1 executes once, lines 2,3, and 6 execute for each index from 0 to $A.length - 1$ ($A.length$ indices), and then lines 2 and 8 execute.

We want a measure that depends on the size of the input, not the particular input. There are three standard ways. Let P be a program, and let I be the set of all inputs for P. Then:

BEST-CASE COMPLEXITY: $\min(t_P(x))$, where x is an input of size n.
In other words, $B_P(n) = \min\{t_P(x) \mid x \in I \wedge size(x) = n\}$.

WORST-CASE COMPLEXITY: $\max(t_P(x))$, where x is an input of size n.
In other words, $W_P(n) = \max\{t_P(x) \mid x \in I \wedge size(x) = n\}$.

AVERAGE-CASE COMPLEXITY: the weighted average over all possible inputs of size n.
Assuming all the inputs are equally likely,

$$A_P(n) = \frac{\sum_{x \text{ of size } n} t_P(x)}{\text{number of inputs of size } n}$$

Best-case: mostly useless. Average-case: difficult to compute. Worst-case: easier to compute, and gives a performance guarantee.

What is meant by "input size"? This depends on the algorithm. For linear search, the number of elements in the array is a reasonable parameter. Technically (in CSC363, for example), the size is the number of bits required to represent the input in binary. In practice we use the number of elements of input (length of array, number of nodes in a tree, etc.) since this differs from the number of bits only by a constant factor. Remember that if we're using asymptotic notation to bound run times, multiplicative constants usually don't matter too much (they'll get absorbed into the c).

What is the best-case for linear search?[10] What about the worst-case for linear search?[11] The average-case for linear search?[12] Once we've answered these questions, we can use the machinery of Big-O. Suppose U is an upper bound on the worst-case running time of some program P, denoted $T_P(n)$:

$$T_P \in O(U)$$
$$\Leftrightarrow$$
$$\exists c \in \mathbb{R}^+, \exists B \in \mathbb{N}, \forall n \in \mathbb{N}, n \geq B \Rightarrow T_P(n) \leq cU(n)$$
$$\Leftrightarrow$$
$$\exists c \in \mathbb{R}^+, \exists B \in \mathbb{N}, \forall n \in \mathbb{N}, n \geq B \Rightarrow \max\{t_P(x) \mid x \in I \wedge size(x) = n\} \leq cU(n)$$
$$\Leftrightarrow$$
$$\exists c \in \mathbb{R}^+, \exists B \in \mathbb{N}, \forall n \in \mathbb{N}, n \geq B \Rightarrow \forall x \in I, size(x) = n \Rightarrow t_P(x) \leq cU(n)$$
$$\Leftrightarrow$$
$$\exists c \in \mathbb{R}^+, \exists B \in \mathbb{N}, \forall x \in I, size(x) \geq B \Rightarrow t_P(x) \leq cU(size(x))$$

So to show that $T_P \in O(U(n))$, you need to find constants c and B and show that for an arbitrary input x of size n, P takes at most $c \cdot U(n)$ steps.

In the other direction, suppose L is a lower bound on the worst-case running time of algorithm P:

$$T_P \in \Omega(L)$$
$$\Leftrightarrow$$
$$\exists c \in \mathbb{R}^+, \exists B \in \mathbb{N}, \forall n \in \mathbb{N}, n \geq B \Rightarrow \max\{t_P(x) \mid x \in I \wedge size(x) = n\} \geq cL(n)$$
$$\Leftrightarrow$$
$$\exists c \in \mathbb{R}^+, \exists B \in \mathbb{N}, \forall n \in \mathbb{N}, n \geq B \Rightarrow \exists x \in I, size(x) = n \wedge t_P(x) \geq cL(n)$$

So, to prove that $T_p \in \Omega(L)$, we have to find constants c, B and for arbitrary n, find an input x of size n, for which we can show that P takes at least $cL(n)$ steps on input x.

6.3.2 Insertion sort

Here is an intuitive[13] sorting algorithm:

```
// A is an array of comparable elements
// that will be rearranged (sorted) in non-decreasing order
   IS (A) {
1.    i = 1;
2.    while (i < A.length) {
3.       t = A[i];
4.       j = i;
5.       while (j > 0 && A[j-1] > t) {
6.          A[j] = A[j-1];
7.          j = j-1;
8.       }
9.       A[j] = t;
10.      i = i+1;
11.   }
12. }
```

Since we last computed running time, we have become lazier. We could use the list from last time for the number of steps of each expression, and we'd find that there are between 3 and 11 steps for the lines in the program above. Since we are interested in big-O comparisons, that four-fold difference in steps will be absorbed into our multiplicative constants, so a better use of our time would be to count each line as one step.[14]

Let's find an upper bound for $T_{IS}(n)$, the maximum number of steps to Insertion Sort an array of size n. We'll use the proof format to prove and find the bound simultaneously — during the course of the proof we can fill in the necessary values for c and B.

We show that $T_{IS}(n) \in O(n^2)$ (where $n = A.length$):

Let $c =$ ____. Let $B =$ ____.
  Then $c \in \mathbb{R}^+$ and $B \in \mathbb{N}$.
  Let $n \in \mathbb{N}$, and let A be an array of length n, and assume $n \geq B$.
    So lines 5–7 execute at most n times, for $3n$ steps, plus 1 step for the last loop test.
    So lines 2–11 take less than $n(6 + 3n) + 1 = 6n + 3n^2 + 1$ steps.
    So $3n^2 + 6n + 1 \leq cn^2$ (fill in the values of c and B that makes this so — setting $c = 10, B = 1$ should do).
  So $n \geq B \Rightarrow T_{IS}(n) \leq cn^2$.
  Since n is the length of an arbitrary array A and a natural number, $\forall n \in \mathbb{N}, n \geq B \Rightarrow T_{IS}(n) \leq cn^2$ (so long as $B \geq 1$).
Since c is a positive real number and B is a natural number, $\exists c \in \mathbb{R}^+, \exists B \in \mathbb{N}, \forall n \in \mathbb{N}, n \geq B \Rightarrow T_{IS}(n) \leq cn^2$.
So $T_{IS} \in O(n^2)$ (by definition of $O(n^2)$).

Similarly, we prove a lower bound. Specifically, $T_{IS} \in \Omega(n^2)$:

Let $c =$ ____. Let $B =$ ____.

Then $c \in \mathbb{R}^+$ and $B \in \mathbb{N}$.

Let $n \in \mathbb{N}$, and let $A = [n-1, \ldots, 1, 0]$ (notice that this means $n \geq 1$). Assume $n \geq B$.

Note that at any point during the outside loop, $A[0..(i-1)]$ contains the same elements as before but sorted (i.e., no element from $A[(i+1)..(n-1)]$ has been examined yet). Since the value $A[i]$ is less than all the values $A[0..(i-1)]$, by construction of the array, the inner while loop makes i iterations, at a cost of 3 steps per iteration, plus 1 for the final loop check. This is strictly greater than $2i+1$, or greater than or equal to $2i+2$., so (since the outer loop varies from $i = 1..i = n-1$ and we have $n-1$ iterations of lines 3 and 4, plus one iteration of line 1), we have that $t_{IS}(n) \geq 1 + 3 + 5 + \cdots + (2n-1) + (2n+1) = n^2$ (the sum of the first n odd numbers), so long as n is at least 4.

So $n \geq B \Rightarrow T_{IS}(n) \geq cn^2$ (setting $B = 4, c = 1$ will do).

So there is some array A of size n such that $t_{IS}(A) \geq cn^2$.

Since n was an arbitrary natural number, $\forall n \in \mathbb{N}, n \geq B \Rightarrow T_{IS}(n) \geq cn^2$.

Since $c \in \mathbb{R}^+$ and B is a natural number, $\exists c \in \mathbb{R}^+, \exists B \in \mathbb{N}, \forall n \in \mathbb{N}, n \geq B \Rightarrow T_{IS}(n) \geq cn^2$.

So $T_{IS} \in \Omega(n^2)$ (definition of $\Omega(n^2)$).

From these proofs, we conclude that $T_{IS} \in \Theta(n^2)$.

Exercises

1. We determined that the running time for linear search $LS(A, x)$ looking for x in an array A of length n, denoted $t_{LS}(A, x)$, was:

   - $t_{LS}(A, x) = 15$ if $A[0] = x$ (best case).
   - $t_{LS}(A, x) = 15n + 10$, if x is not in the array (worst case).

   Let $T_{LS}(n) = \max\{t_{LS}(A, x) \mid A \text{ has length } n\}$.

   (a) Is $T_{LS}(n) \in O(n)$?[15]

   (b) Is $T_{LS}(n) \in O(\frac{1}{n})$?[16]

   (c) Is $T_{LS}(n) \in O(n^2)$?

   (d) Is $T_{LS}(n) \in O(2^n)$?

   (e) Is $T_{LS}(n) \in \Omega(\frac{1}{n+1})$?

2. Consider the following method that was recently posted to a newsgroup:

```
void f(int k) {
    if (k < 100) {
         FunctionOne(k);   //O(n)
    } else {
         int i;
         for(i=0; i<10; i++)
             FunctionTwo(k);   //O(log n)
    }
}
```

Let $n = \lfloor \log_2 k \rfloor + 1$ be the size of the input k (the number of bits needed to represent the value k). Assume that `FunctionOne()` always takes $\Theta(n)$ time to execute, and that `FunctionTwo()` always takes $\Theta(\log n)$ time to execute.

Let $T_{\mathtt{f}}(n)$ be the maximum time to execute `f` on input of size n. Determine a function g such that $T_{\mathtt{f}} \in \Theta(g)$, and prove your answer.[17]

Chapter 6 Notes

[1]From 0 to $(2-1)$, if we work in analogy with base 10.

[2]To parse the 0.101 part, calculate $0.101 = 1(2^{-1}) + 0(2^{-2}) + 1(2^{-3})$.

[3]You should be able to look up this algorithm in an elementary school textbook.

[4]Same as the previous exercise, but only write numbers that have 0's and 1's, and do binary addition.

[5]Integer divides by 10.

[6]Integer divide by 2.

[7]If only one of m, n are negative, ensure n is the negative one, perhaps by swapping. If both of m, n are negative, negate both of them and then call `mult`.

[8]This is the principle of mathematical induction, which you'll study in detail in CSC 236.

[9]As long as the loop eventually terminates! How would we prove that this happens?

[10]15 steps, when $A[0] == x$.

[11]$15n + 10$, where $n = A.length$.

[12]Inputs at index 0 through $n-1$, plus missing value equally likely $(n+1)$ input categories, so

$$\begin{aligned}
\frac{\left(\sum_{i=0}^{n-1} 15(i+1)\right) + 15n + 10}{n+1} &= \frac{15n(n+1)/2 + 15n + 10}{n+1} \\
&= \frac{7.5n(n+1) + 15n + 10}{n+1} \\
&= \frac{7.5n(n+1) + 15n + 10}{n+1} = 7.5n + \frac{15n+10}{n+1}.
\end{aligned}$$

[13]but not particularly efficient...

[14]Technically, we'll bound the number of steps performed on each line by some constant k, so each line costs at most k. When we bound the growth rate using asymptotic notation, k will be absorbed into c, so it's like each line costs only one step.

Note that this only works when *each* line does only a constant amount of work: we'll need to deal with any lines that do more more than a constant amount of work separately (such as method calls).

[15]Yes. If $n \geq 10$ and $c = 16$, then $T_{LS}(n) = 15n + 10 \leq 16n$.

[16]No. There is no constant that will make $15n + 10 \leq c/n$, once n is somewhat greater than $16c$.

[17]You picked $g(n) = \log n$, I hope. What value do you pick for B in your proof?