5 Modeling Computation and Runtime

What does it mean for an algorithm to be efficient? This depends on the context. In general an efficient algorithm is one which doesn't use "too much" of any resource. There are lots of resources that we might care about, but the two that we will care about in this course are *time* (how long the algorithm takes to run) and *space* (how much computer memory the program needs access to).

Comparing the runtime of two algorithms is more difficult than it may sound. In Sections 5.1 and 5.2 we will discuss several choices we need to make in order to meaningfully define what it even means for one algorithm to be faster than another. In Section 5.3, we will discuss a simplified model of computation called the (word) RAM model that allows us to describe algorithms more rigorously than we have done so far and to formally define their runtime. In Section 5.4, we'll show that despite its simplicity, we can actually implement the kinds of features we expect from a modern programming language in the RAM model. In Section 5.5 we'll see an example of a full program in the RAM model and rigorously analyze its runtime. Finally in Section 5.6, we'll show how to use our RAM model-based definition of runtime to analyze algorithms written in pseudocode, like those we've seen in Chapter 3.

5.1 Measuring the Runtime of Algorithms

Given an algorithm, how do we determine how fast it runs? For example, suppose we want to analyze the runtime of the algorithm below (which we previously saw in Section 3.3). How should we do it?

```
InsertionSort(A[1, ..., n]):
    For i from 1 to n:
        For j from i − 1 down to 1:
            If A[j] > A[j + 1]:
                Swap A[j] with A[j + 1]
            Else:
                Break
```

One approach would be to implement the algorithm in a real programming language, then run it and time how long the program ran for. This can be a good solution if we know exactly how the algorithm is going to be used. As a general way of comparing algorithms, however, it has serious problems:

- How do we choose what language/implementation to use? (This may affect the runtime significantly)
- What hardware do we run the program on? (The same code can run much faster on some computers than others)
- What input do we run the program on? (Insertion sort will be much faster on an array that is already mostly sorted than on an array that is in reverse sorted order. We need to know what kind of arrays we expect to get as inputs)
- How do we control for other factors that can impact the runtime? (Even on the same input and the same computer, the same program will not always take the same amount of time because of factors like what other processes are using RAM in the background).

Because of these reasons and others, the standard way of comparing the runtime of algorithms is to use a theoretical model. Instead of measuring the real time that a program takes, we measure the number of "steps" that the algorithm performs, where a step is a single basic instruction. On a real computer, these steps will not all take the exact same amount of time, but as long as we make reasonable choices about what to count as a step, the time each one takes on a real computer should be similar. We'll discuss how we determine what exactly counts as a single step of computation in Sections 5.2 and 5.3.

Counting computation steps in an abstract model solves most of the issues mentioned above, but it still leaves us with the problem of choosing what input to run the program on. The truth is that we cannot accurately capture the runtime of an algorithm by running it on any single input. Instead it makes more sense to view the runtime as a function of the input. Assuming that the input to our algorithm is encoded as a single string, we define, for each $s \in \Sigma^*$, Runtime(s) as the number of steps that our algorithm takes on input s.

Typically what we are most interested in is how the *size* (number of characters) of the input affects the runtime. To compute this we need some way to summarize the runtime of the algorithm across all inputs of a given size. Here are three natural ways we could define R(n), the runtime of our algorithm on inputs of size n.

- The *best-case* runtime is defined as $R_{best}(n) := \min_{s \in \Sigma^n}$ Runtime(s)
- The *worst-case* runtime is defined as $R_{worst}(n) = \max_{s \in \Sigma^n}$ Runtime(s)
- The *average-case* runtime is defined as $R_{average}(n) = (1/|\Sigma|^n) \sum_{s \in \Sigma^n}$ Runtime(s)

The best-case is mostly useful for impressing naive investors, but it can occasionally give insights on how to speed up an algorithm (perhaps we can find a way to achieve the best-case runtime on a larger set of strings). The average-case runtime sounds practical, but it is usually complicated to calculate, and assuming that the input is a uniformly random string is often not realistic. The worst-case runtime is what we will focus on in this course. When we talk about the "runtime" of an algorithm, we mean "the worst-case runtime function R_{worst}". An advantage of the worst-case runtime is that it gives us a guaranteed upper bound on how long our algorithm will take (on inputs of size n), even when we know nothing about the specific inputs our algorithm will be given.

We'll calculate the worst-case runtime of InsertionSort later in this chapter, after clarifying a few more details about how we analyze runtimes.

5.2 Defining Steps of Computation

As we discussed in the previous section, we would like to define the runtime of an algorithm on an input as the number of steps/instructions/lines of code that the algorithm executes when run on that input. We have to be careful how we define a "step", however, because different choices can give very different answers for the runtime of the same algorithm.

For example, consider the following high level pseudocode for an algorithm which determines whether or not a given string is a palindrome (reads the same backwards and forwards).

```
IsPalindrome(s):
    Return s = reverse(s)
```

What is the runtime of IsPalindrome? To answer this, let's try to count the number of "steps" that this algorithm takes when we implement it in an actual programming language.

Here is a simple Python implementation of IsPalindrome.

```
def is_palindrome(s):
    return s == s[::-1]
```

It's fine if you aren't familiar with this syntax, all the program is doing is comparing the string s with its reverse and returning whether or not they are equal. In Python, we can do this in a single line of code, regardless of how long s is. So if we define a "step" as "a single line of Python", then this algorithm runs in $\Theta(1)$ time.

If we wanted to write the same algorithm in C (or C++), then we don't have built in syntax for reversing an array, so we would need to write something like

```
bool is_palindrome(char s[], int size) {
    char* reverse = malloc(size);
    for(int i = 0; i < size; i++) {
```

```
        reverse[i] = s[size-i-1];
    }
    for(int i = 0; i < size; i++) {
        if (s[i] != reverse[i]) {
            return false;
        }
    }
    free(reverse);
    return true;
}
```

This version of the function has two for loops, each of which runs for up to `size` iterations, so if we define a "step" as a line of C code, then this algorithm runs in $\Theta(n)$ time.

The first formal models of computation were lambda calculus and Turing machines, developed in the 1930s by Alonzo Church and his student, Alan Turing. Defining these models is outside the scope of this course, but it can be proven that they have the same power as modern programming languages in the sense that any function which can be computed by a Python program (for example) can also be computed by some Turing machine. This does not mean, however, that a Turing machine can compute any function with the same asymptotic efficiency as a Python program.

A key distinction between Turing machine and actual programming languages like C is that Turing machines do not have random access, i.e. a Turing machine has no equivalent to the expression `s[i]` in our C program. Instead, a Turing machine has a single "head", which moves through memory one symbol at a time. Thus, if the Turing machine takes an array s as input and wants to access $s[i]$, it has to "move right" i times (i.e. the Turing machine treats an array more like a doubly linked list). If we were to implement the `is_palindrome` on a Turing machine it would look something like this (once again, it is not important to understand the details of this program):

```
is_palindrome(s):
    while head is not blank:
        char c = head
        write blank
        move right
        if head is blank:
            accept
        while head is not blank:
            move right
        move left
        if head != c:
            reject
        write blank
        move left
        while head is not blank:
            move left
        move right
    accept
```

This program features nested while loops. The outer loop iterates through $\lfloor n/2 \rfloor$ pairs of characters, and for each pair the inner while loops have to travel all the way across the input array and then all the way back to check if the first character of the input matches the last one. Thus, if we define a "step" as a single Turing machine instruction, then IsPalindrome runs in $\Theta(n^2)$ time.

So which of these is the right answer? Does IsPalindrome run in $\Theta(1)$, $\Theta(n)$, or $\Theta(n^2)$ time? A good way to build intuition here is to try running the Python and C programs that we wrote on an actual computer. While we said before that "the time the algorithm takes on an actual computer" is too imprecise to be a definition of runtime, we still want the asymptotic runtimes of programs in our theoretical model to match their asymptotic runtimes on actual hardware (i.e. if the number of

seconds that a program takes to run grows quadratically with the input size, ideally it should be an $O(n^2)$ time algorithm in our model).

If we run both the Python and C programs on strings of various sizes we find that the runtime for both programs seems to be linear in the length of the input string, suggesting that $O(n)$ is the right answer for this algorithm. In fact, if we look at how the Python interpreter translates the line `return s == s[::-1]` into machine code, we'll find that it uses a loop structure similar to our C program.

This doesn't mean that "a line of C code" is the right definition of a step of computation, however. We can still write single lines of C code that take a long time to run (e.g. if `a` and `b` are structs, then `a=b` copies all the attributes of `b` into `a`, which may take a long time if the struct is large). Furthermore, even if we did find a programming language where each instruction takes about the same amount of time (maybe some version of assembly), real programming languages are far too complicated, with too many instructions and edge cases to make them a reasonable definition of computation (the C++ standard, for example, is over 1000 pages long!). We want a simple minimalist model (like a Turing machine, which can be described in 1 page), but one which is powerful enough to allow manipulating arrays/strings with the same asymptotic efficiency as C or Python. It turns out that the right answer is essentially a Turing machine augmented with the power of random access, i.e. a "Random Access Machine".

5.3 The RAM Model

The property of being able to jump to any desired location in an array (or more generally, in memory) just using the index is called "random access". Essentially all modern computers rely heavily on RAM (random-access memory) so to model the time that operations take on real computers, it makes sense to use a model which includes random access. You can think of the RAM model as a very simple programming language.

Every RAM model program takes a string as input and returns a string as output (i.e. computes a function from $\Sigma^* \to \Sigma^*$ for some alphabet Σ). This is not actually a major limitation, because any type of input or output that a regular program uses is ultimately represented using a string of bytes. We'll talk more about encoding objects as strings in Section 6.2.

Memory in the RAM model consists of one long array of cells, which we call M. Each cell of M stores a single character from Σ. Initially the size of M is equal to n, the input length, and M stores the input to the algorithm in cells $M[1]$ through $M[n]$. While running, the machine can increase the size of M by using the "Malloc" instruction, which adds one cell to the end of M containing a specified character.

In addition to the memory array (which can grow arbitrarily large) a RAM model program also has a finite number of variables. RAM model programs have only two types of variables, character variables, which store a single symbol from Σ and integer variables. Integer variables in the RAM model function like a cross between int and pointer types in a language like C, because we can perform arithmetic with them, but we can also use them to index into memory. Like integer and pointer variables in C or many other languages, integer variables in the RAM model cannot store arbitrarily large integers. Instead, integer variables can store values in the range $[-2^w, 2^w]$, where w is a value called the "word size".

On a real computer, the word size is a fixed constant. These days most computers have a word size of 64 (i.e. a 64-bit operating system), but in the 90s, a word size of 32 was most common, and in the 70s many computers used a word size of 16. Because the word size bounds the values of variables which are used as indices into memory (i.e. addresses), the word size for a particular computer needs to be large enough to store all of the possible indices of memory locations. So if our computer has word size w, the maximum amount of RAM that it can use is 2^w. This means that with a word size of 16, we can use up to 64 kilobytes of RAM, with a word size of 32, we can use up to 4 gigabytes, and with a word size of 64, we can use up to 16 exabytes (about 17 billion gigabytes). This is why the word size used in actual computers has increased over time, because the amount of memory that we can cram into a computer chip has increased exponentially over the past several decades.

In our model, using a fixed word size doesn't work. We want to be able to describe algorithms that work for arbitrarily large inputs (because we are interested in the asymptotic behavior of their runtime), and so our machine's memory has unbounded size. Therefore, we define the word size w as $\max(64, \lceil \log_2(\text{memory\_size}+1) \rceil)$, so that we always have $\text{memory\_size} < 2^w$, meaning that we can store the index of any memory cell in an integer variable, but we also have $w \geq 64$ so that we store the kind of values that fit in an integer variable on a modern computer even if our program's memory is very small (e.g. if our program takes an empty string as input, then memory is initially empty). Note that this means that the word size can change over the course of the program as we allocate more memory.

Formally, a program in the (word)-RAM model consists of:

- An alphabet Σ
- A finite collection of variables (sometimes called "registers"), each of which is either a character variable (storing a single character) or an integer variable (storing a single integer between -2^w and 2^w, inclusive) including the following special variables which cannot be assigned to directly, only modified by the Malloc instruction):
  - memory_size: an integer variable initialized with the number of characters in the input
  - word_size: an integer variable initialized with $\max(64, \lceil \log_2(\text{memory\_size}+1) \rceil)$
- A finite list of instructions (or "lines of code") where each instruction is one of the following:
  - $a \leftarrow b$
    - a must be a variable and b must be a variable or constant of the same type (integer or character).
    - This instruction copies the value of b into the variable a.
  - $M[i] \leftarrow c$
    - i must be an integer variable or constant and c must be a character variable or constant.
    - This instruction copies the value of c into position i in memory. If i is out of bounds, no memory is changed.
  - $c \leftarrow M[i]$
    - i must be an integer variable or constant and c must be a character variable.
    - This instruction copies the character at position i of memory into c. If i is out of bounds, c is left unchanged.
  - $i \leftarrow j$ op k
    - i must be an integer variable, j and k must be integer variables or constants, and op must be one of the following: $+, -, *, /$.
    - This instruction stores the result of j op k in i. / performs integer division (i.e. the fractional part is discarded). If j op k would not fit in an integer variable (because it is $< -2^w$ or $> 2^w$) or is undefined (because of division by 0), i is left unchanged.
  - If a op b: goto ℓ
    - a must be a variable, b must be a variable or constant of the same type (integer or character) as a and ℓ must be the number of an instruction (i.e. an integer between 1 and the total number of instructions), and op must be one of the following: $<, \leq, =, \geq, >$.
    - If the comparison a op b is true, then this instruction causes the machine to jump to instruction number ℓ and continue executing instructions from that position. If the comparison is false, the machine proceeds to the instruction immediately after this one, as usual.

- $i \leftarrow \text{decode}(start, size)$
  - i must be an integer variable, while $start$ and $size$ must be integer variables or constants.
  - This instruction interprets the string of length $size$ symbols beginning at position $start$ in memory (i.e. $M[start : start + size - 1]$) as the representation of a single integer, and stores that integer in i. If these characters do not represent a valid integer (either they don't encode an integer or the encoded integer is $< -2^w$ or $> 2^w$) then i is left unchanged.
- $M[start, size] \leftarrow \text{encode}(i)$
  - $i, start$, and $size$ must be integer variables or constants.
  - This instruction encodes i as a string over Σ and stores the resulting string beginning at index $start$. If the encoding of i has fewer than $size$ characters, it is padded with leading 0s until it has length $size$ (this is analogous to how an integer in C++ takes up 4 bytes, whether the number being stored is 1 or 100000). If the encoding of i has more than $size$ characters, then memory is left unchanged.
- $\text{Malloc}(c)$
  - c must be a character variable or constant.
  - This instruction adds a single new cell storing c to the end of M. It also increases memory_size by 1. If this causes the length of M to become 2^w, then this instruction also increases w by 1 (meaning that integer variables can now store larger values, so that it is possible to index into this new memory).
- $\text{Output}(start, size)$
  - $start$ and $size$ must be integer variables or constants.
  - Ends the program, outputting the string of length $size$ beginning at $M[start]$ (i.e. $M[start], \ldots, M[start + size - 1]$).

Since the RAM model is not a real programming language, different authors have their own slightly different versions of it. We could've allowed more instructions, or slightly fewer, without changing the asymptotic runtime of programs in the model (we'll see in Section 5.4 how some features we didn't define as part of the model, like loops and functions, are actually pretty easy to implement).

The one major difference you may encounter relates to the word size. The earliest descriptions of the RAM model did not feature a word size: variables could store arbitrarily large integers. This leads to an unrealistic model, however, because real computers cannot perform arithmetic on arbitrarily large integers in constant time. In fact, the time to perform arithmetic operations is a major bottleneck in areas like cryptography, where working with numbers with thousands of digits is common. Therefore the version of the RAM model where the values for integer variables are limited by the word size (often called the word-RAM model to distinguish it from the version without a word size) is the one most commonly used in modern algorithms research, and is the only one we will care about in this course. We'll discuss the runtime of arithmetic with large integers in our model in Chapter 9.

5.4 Implementing Quality of Life Features in the RAM Model

The instruction set we gave in Section 5.3 may seem pretty restrictive. It lacks a lot of nearly universal programming language features like loops, functions, and arrays. It turns out, however, that even with the limited instruction set of the RAM model, we can simulate all of these features and more. In this section we'll give some examples of how to do this.

First let's look at implementing if statements. An if-else statement like the one below (where "do X" is a stand-in for additional blocks of code) can be implemented in the RAM model using our conditional goto.

```
if x = y:
    do A
else:
    do B
do C
```

An equivalent RAM model program is:

1: If $x = y$: goto 4
2: do B
3: If $0 = 0$: goto 5
4: do A
5: do C

Note that we used the always true statement $0 = 0$ to convert the conditional goto into a statement that always jumps to the given line, so that we can avoid executing the "if" and "else" cases together. Also note that because our program moves on to the next line after the if when the condition is not met, we put the code from the else block before the code from the if block.

Our translation of an if-else statement relied on the condition in the if statement having a simple form like "x = y", because our conditional goto instruction requires the condition to be a basic comparison between two variables. We can simulate more complicated boolean expressions in the RAM model, however. For example, we can translate

```
if w = x and y = z:
    do A
do B
```

into

1: w_equals_x $\leftarrow 1$
2: If $w = x$: goto 4
3: w_equals_x $\leftarrow 0$
4: y_equals_z $\leftarrow 1$
5: If $y = z$: goto 7
6: y_equals_z $\leftarrow 0$
7: true_count $\leftarrow$ w_equals_x + y_equals_z
8: If true_count < 2: goto 10
9: do A
10: do B

Here we have created new "boolean" variables (integer variables that we only set to 0 or 1) to represent the truth values of the two sides of the "and". To check if both are true, we add up the truth values and check if the total is 2. We can implement other logical operators in a similar way (e.g. an "or" statement is true if the sum of the truth values of both sides is at least 1).

Now that we know how to implement a variety of conditionals, we can also implement loops. A while loop like

```
while x ≤ y:
    do A
do B
```

becomes

1: If $0 = 0$: goto 3
2: do A
3: If $x \leq y$: goto 2
4: do B

and a for loop like

```
for i from 1 to n:
    do A
do B
```

becomes

1: $i \leftarrow 1$
2: If $i > n$: goto 6
3: do A
4: $i \leftarrow i + 1$
5: If $0 = 0$: goto 2
6: do B

Conditional goto allows us to repeatedly return to the beginning of the loop as long as the loop condition is met.

Our RAM model has only two primitive data types, characters and integers, but we can create more complicated data types in memory out of these basic building blocks. For example, if we want to store an array of characters (i.e. a string), we can use two variables, one to store the index where the array starts, and one to store the length of the array. For example, we could represent a string s with the variables s_{start} and s_{length}. Then, to access $s[i]$, we simply need to access $M[s_{start} + i - 1]$ (since our indices for arrays and strings begin at 1).

We can represent an array of integers in a similar way, however, if we want to allow the integers in the array to be arbitrarily large, we should add a third variable keeping track of the number of characters we're using to represent each integer. Given variables A_{start}, A_{length}, and $A_{element\_size}$, we can access $A[i]$ via $\text{decode}(A_{start} + (i-1) * A_{element\_size}, A_{element\_size})$.

We can also implement function calls in our RAM model, although the process is a bit more complicated (akin to how function calls are actually translated into low-level machine code, which also relies on goto instructions).

- We can jump to the code of a function with an unconditional goto (if $0 = 0$ goto)
- In order to return from the function, however, we need a way to remember which line of code called the function.
- Since we can have arbitrarily long chains of functions calling each other, we need to store this line number to return to in memory.
- Similarly, the parameters/local variables of the function also need to be stored in our memory array M
  - We could try making the parameters global variables (the only kind we have in the RAM model) instead, but this approach cannot handle recursive functions, which can have multiple different copies of their local variables existing at the same time.
- In essence what we want is an array of "stack frames" where each frame stores the local variables for a function as well as the instruction to return to when the function finishes.
- The details of this process are too technical for this course, but hopefully this gives you a sense of how we could translate large complicated programs into the RAM model

There are many more features we could implement, but this should be enough to give you some idea of how we can reduce more complicated code down to the instruction set from the RAM model. It turns out that in fact any function mapping strings to strings which can be computed by a program in C or Python or another normal programming language can also be computed by a program in the RAM model. Proving this is rather tedious, however, and beyond the scope of this course.

5.5 Analyzing the Runtime of Algorithms

Now that we've fully described a computational model, we can finally define the runtime of an algorithm. The (worst-case) runtime of an algorithm ALG is a function $f : \mathbb{Z}_{\geq 0} \to \mathbb{Z}^+$ where $f(n) :=$ the maximum over all $x \in \Sigma^n$ of the number of RAM model instructions that ALG(x) executes.

Let's look at a couple of examples of full programs in the RAM model and analyze their runtimes. We'll start with the IsPalindrome algorithm that we discussed in Section 5.2.

```
IsPalindrome(s):
  Return s = reverse(s)
```

Here is an implementation of IsPalindrome in our model. Lines 1 through 9 compute the reverse of the input string, storing it in positions $n+1$ through $2n$ in memory, and then lines 10 through 32 compare the string with its reverse to check for equality. As you can see, even an algorithm that is extremely conceptually simple can have a rather long description in the RAM model!

- Alphabet: Any set containing all possible input characters as well as English letters a-z.
- Integer variables: memory_size, word_size, input_size, i, j
- Character variables: temp, forward_char, reverse_char

1: input_size ← memory_size
2: i ← 0
3: If i = input_size: goto 9
4: j ← input_size - i
5: temp ← $M[j]$
6: Malloc(temp)
7: i ← i + 1
8: If 0 = 0: goto 3
9: i ← 1
10: If i > input_size: goto 25
11: forward_char ← $M[i]$
12: j ← i + input_size
13: reverse_char ← $M[j]$
14: i ← i + 1
15: If forward_char = reverse_char: goto 10
16: If memory_size ≥ 5: goto 19
17: Malloc('a')
18: If 0 = 0: goto 16
19: $M[1]$ ← 'f'
20: $M[2]$ ← 'a'
21: $M[3]$ ← 'l'
22: $M[4]$ ← 's'
23: $M[5]$ ← 'e'
24: Output(1,5)
25: If memory_size ≥ 4: goto 28
26: Malloc('a')
27: If 0 = 0: goto 25
28: $M[1]$ ← 't'
29: $M[2]$ ← 'r'
30: $M[3]$ ← 'u'
31: $M[4]$ ← 'e'
32: Output(1,4)

Alright, so what is the runtime of this algorithm?

Theorem 27. *The runtime of* IsPalindrome *is* $12n + 11 + 3\max(0, 4 - 2n)$.

Proof. We break the program into pieces.

- Lines 1-2 run once each, so the total time (number of instructions run) for these lines is just 2.
- Lines 4-8 each run input_size many times (since i starts at 0 and they run until i is equal to input_size. Line 3 runs input_size + 1 times (since it is also executed in the final iteration when i is equal to input_size). Thus the total time for lines 3-8 is $6n + 1$.

- Line 9 runs once, for a total time of 1.
- The runtime of the rest of the program depends on the actual characters in the input—in particular it depends on whether or not the input string is a palindrome. Let's separately consider both cases
- First suppose that the input string is a palindrome
  - Since the input is a palindrome, forward_char will always be equal to reverse_char on line 15. Thus, lines 11-15 run input_size many times (once for each character of the input), and line 10 runs input_size + 1 times (since it also runs in the final iteration when i is equal to input_size + 1). Thus, the total time for lines 10-15 is $6n + 1$
  - After matching all of the characters in the forward and reverse strings, the program jumps to line 25, so lines 16-24 do not run at all.
  - When we reach line 25, memory_size is equal to double input_size (since we doubled the size of memory to create the reversed copy). Therefore lines 26-27 run $\max(0, 4 - 2n)$ times and line 25 runs $1 + \max(0, 4 - 2n)$ times (since it runs each time our memory size is less than 4, plus one extra time when our memory size is at least 4). Thus, the total time for lines 25-27 is $1 + 3\max(0, 4 - 2n)$.
  - Lines 28-32 run once each, for a total time of 5
  - Thus, if the input is a palindrome the total runtime of IsPalindrome is

$$2 + (6n + 1) + 1 + (6n + 1) + 1 + 3\max(0, 4 - 2n) + 5 = 12n + 11 + 3\max(0, 4 - 2n)$$

- Now suppose that the input x is not a palindrome. In this case, there must be some index i where $x[i] \neq \text{reverse}(x)[i]$. Furthermore, the first index where this happens must be less than or equal to $n/2$ (i.e. before the middle of the string), because when we are comparing x to its reverse, every character in the first half gets paired with one from the second half and vice-versa, so every unmatched pair has a character from the first half of the string.
- Therefore, lines 10-15 each run at most $\lfloor n/2 \rfloor$ times before we reach the non-matching pair of characters and continue on to line 16 (and they run exactly this many times if the input is an "almost palindrome" where only the middle pair of characters don't match), so the worst-case runtime lines 10-15 is $6\lfloor n/2 \rfloor$
- As above, we note that the current memory_size is $2n$, so lines 16-18 run $1 + 3\max(0, 5 - 2n)$ times in total.
- Lines 19-24 run once each, for a total time of 6
- line 24 ends the program, so lines 25-32 never run
- Thus, the worst-case total runtime when the input is not a palindrome is

$$2 + (6n + 1) + 1 + 6\lfloor n/2 \rfloor + 1 + 3\max(0, 5 - 2n) + 6 = 6n + 6\lfloor n/2 \rfloor + 11 + 3\max(0, 5 - 2n)$$

The overall worst-case runtime is the maximum of the worst-case runtime for palindromes and the worst-case runtime for non-palindromes:

$$\max(12n + 11 + 3\max(0, 4 - 2n), 6n + 6\lfloor n/2 \rfloor + 11 + 3\max(0, 5 - 2n)).$$

We note that if $n \geq 3$, then $\max(0, 4 - 2n) = \max(0, 5 - 2n) = 0$, and so the runtime on palindromes is strictly larger than the runtime on non-palindromes. We can also check that for $n = 2$, the runtime for palindromes is still larger (the extra iterations of the loop comparing characters takes longer than the extra time to output false). Finally, if $n \leq 1$, the input must be a palindrome (an empty string

or a single character string is always the same as its reverse). Thus, for all input lengths the worst case input is a palindrome, and we can simplify the runtime to:

$$12n + 11 + 3\max(0, 4 - 2n).$$

□

That was an insane amount of effort that we would like to never go through again. Fortunately, calculating the *exact* runtime of algorithms in our model is not important (it depends too much on the specific choice of instruction set for the RAM model, which, as we previously noted is somewhat arbitrary). Instead what we really care about is the asymptotic runtime of the algorithm. Using the tools we learned in Chapter 4, we can confirm that the runtime expression we obtained for IsPalindrome is $\Theta(n)$. As we'll see in the next section, however, as long as all we care about is this final asymptotic answer, we can avoid needing to write out the fully messy RAM model program and calculating the exact runtime function.

5.6 Pseudocode

The simplicity of the RAM model is useful as a definition for runtime. We can calculate the exact number of steps needed for any operation that our program uses by breaking it down into individual RAM model instructions. Writing anything but the simplest algorithms directly in the RAM model is rather tedious, however, and in this course we are not going to concern ourselves with the exact number of steps that an algorithm takes.

Therefore, for the rest of this course, instead of describing algorithms using only RAM model instructions, we will write them in higher level pseudocode, like what we used in Chapter 3 when we discussed proving the correctness of algorithms.

Pseudocode does not have a specific list of valid instructions. In a similar way to how proofs can be written in regular English, using mathematical notation only when it is the clearest way to convey an idea, you can use everyday language in pseudocode instead of trying to adhere to a strict set of syntax rules. I will attempt to use a consistent style for the pseudocode algorithms in these notes, but you do not have to follow it when describing your own algorithms. If you prefer to make your pseudocode look more like your favorite programming language, that's fine, as long as the intent would be clear to someone who doesn't know that language.

A heuristic for whether something is a valid "single instruction" in pseudocode is whether you (or a slightly less clever classmate) would be able to break it down into the basic RAM model instructions. For example, in the previous section, we saw how to write a loop in the RAM model that reverses a string. So it's fine to write pseudocode like the following (which we've seen a bunch of times at this point), where we treat "reverse" as a built in operation, as long as we remember that this instruction actually takes $O(|s|)$ time.

```
IsPalindrome(s):
    Return s = reverse(s)
```

To analyze the runtime of an algorithm written in pseudocode, we don't need to convert each line into RAM model instructions. Instead all we need to figure out is the asymptotic behavior of the RAM model equivalent (e.g. could we implement this in a constant number of RAM model instructions). Generally it will be clear which lines run in constant time, and you can state this without justification (i.e. you don't need to describe the RAM model program at all). Be careful not to make the assumption that every non-loop line of pseudocode corresponds to a constant number of RAM model instructions, however. As we've seen with "reverse" this isn't always the case. We'll see more examples in chapter 9 of pseudocode lines that may look constant time at first glance but actually require many RAM model instructions.

Let's see an example of analyzing the asymptotic runtime of a pseudocode algorithm.

```
InsertionSort(A[1, ..., n]):
    For i from 1 to n:
        For j from i − 1 down to 1:
            If A[j] > A[j + 1]:
                Swap A[j] with A[j + 1]
            Else:
                Break
```

Theorem 28. *The runtime of* InsertionSort *is* $\Theta(n^2)$.

Proof. First we show that the runtime is $O(n^2)$. We note that the outer loop runs n times, and the inner loop runs at most n times, so the body of the inner loop runs at most n^2 times. The body of the loop takes $O(1)$ time per iteration, so the overall runtime is $O(n^2)$.

To show that the runtime is $\Theta(n^2)$, however, we also need to show that the worst-case runtime is $\Omega(n^2)$, i.e. we need to show that for every sufficiently large n we can find an input on which InsertionSort actually takes $\Omega(n^2)$ time.

Suppose that A is in reverse sorted order (i.e. sorted from largest to smallest). Then at the start of every iteration of the outer for loop, $A[i]$ will be smaller than every element of $A[1 : i-1]$, so the inner for loop will run the full $i-1$ times. Each iteration of the inner for loop includes at least one instruction, so we can lower bound the total runtime with the total number of iterations of the inner for loop.

Summing across all n iterations of the outer for loop, we find that the total number of iterations of the inner for loop is

$$\sum_{i=1}^{n}(i-1) = \sum_{i=1}^{n-1} i = (n-1)n/2 = n^2/2 - n/2 \in \Omega(n^2).$$

Note that we simplified the summation using Theorem 3. □

If you are paying very close attention, you may notice that this proof cheated. Look back and see if you can find the false claim.

With what we've stated so far, we can't assume that the body of the loop runs in $O(1)$ time! While comparing two integer variables is a constant time operation in the RAM model, remember that integer variables in the RAM model can store values at most 2^w, where $w = \max(64, \lceil \log_2(\text{memory\_size}+1) \rceil)$. Thus, if our array A contains very large numbers, we may not be able to compare them in constant time. Consider, for example, the case where A contains just two numbers, each with millions of digits.

If A actually contains incredibly huge numbers, however, then n is not actually a good measure of the input size. e.g. if A has two numbers with millions of digits, then input_size is "millions", but n is only 2. It is much more convenient to work with the number of elements in an array rather than the total number of characters/bits. Therefore, unless otherwise stated, whenever we have an algorithm which takes an array of integers as input, we will implicitly assume that each of the elements of the array has at most $100w$ digits (there's nothing special about 100 here, it's just a nice large constant). That way we guarantee that we can store each one in a constant number of RAM model variables and perform comparisons (and arithmetic) on them in constant time. Given this new assumption, the proof of Theorem 28 is correct.

5.7 Comprehension Questions

1. When we talk about the "runtime" of an algorithm in this course, which runtime function do we mean?
   (a) The best-case runtime
   (b) The average-case runtime
   (c) The worst-case runtime

2. True or false: in the RAM model, the word size is a fixed constant which cannot change during the execution of a program.

3. Suppose that we run a RAM model program with the empty string as input. At the start of that program, what is the largest value that it can store in an integer variable?

4. Which of the following are valid single instructions in a RAM model program? (assume that a, b, and c are integer variables).

   (a) $M[a] \leftarrow b + c$

   (b) If $a \bmod 2 = 1$: goto 1

   (c) $a \leftarrow \text{decode}(b, c)$

   (d) $a \leftarrow b/c$

   (e) Malloc

   (f) Output(a, c)

5. Give a big-Θ expression for the runtime of IsPalindrome.

6. Give a big-Θ expression for the runtime of InsertionSort.

7. Give an exact expression (not big-O/big-Θ) for the runtime of each of the following RAM model programs in terms of the input size n:

   1: If memory_size ≥ 10: goto 4
   2: Malloc('0')
   3: If $0 = 0$: goto 1
   4: Output(1,1)

8. Below is an algorithm which takes as input a non-empty array of n distinct integers and outputs the median. Give a simple (polynomial) big-Θ expression for the runtime of the algorithm.

```
FindMedian(A[1, ..., n]):
    While |A| > 2:
        x ← max(A)
        y ← min(A)
        remove x from A
        remove y from A
    If |A| = 1:
        Return A[1]
    Else:
        Return (A[1] + A[2])/2
```

9. Below is an algorithm which takes as input a string of length n and reverses it. Give a simple (polynomial) big-Θ expression for the runtime of the algorithm.

```
Reverse(s):
    n ← |s|
    For i from 1 to n:
        temp ← s[n]
        For j from n down to i + 1:
            s[j] ← s[j − 1]
        s[i] ← temp
    Return s
```

5.8 Exercises

1. The RAM model does not have a "mod" operation. Show that nevertheless we can implement $a \leftarrow b \mod c$ in a constant number of RAM model instructions. That is, give a sequence of RAM model instructions which will set a to $b \mod c$, assuming that a, b, c are existing integer variables. You may assume that b and c both have **positive** values. You may introduce any additional variables you wish.

2. Translate this pseudocode into a RAM model program.

```
CountAs(s):
    c ← 0
    For i from 1 to |s|:
        If s[i] = 'a':
            c ← c + 1
    Return c
```

3. In Section 5.6 we claim that we can represent numbers with at most $100w$ digits in a constant number of RAM model variables and perform arithmetic on them in constant time. Justify this by explaining how to actually represent numbers with $100w$ binary digits and perform addition on them in the RAM model (Hint: start by showing how to represent and manipulate numbers with $2w$ binary digits).

4. Determine with proof the runtime of this algorithm

```
ExchangeSort(A[1, ..., n])
    Repeat forever:
        i ← −1
        For j from 1 to n − 1:
            If A[j] > A[j + 1]:
                i ← j
        If i = −1:
            //A is now sorted
            Return
        Swap A[i] with A[i + 1]
```