8.1 An introduction to algorithms

This section has been set as optional by your instructor.

Suppose you were given a list of five numbers and asked to find the smallest one. You would probably only require a quick glance at the list of numbers to find the smallest one. Suppose, however, you were given a list of temperatures recorded in a city every hour for the past ten years, and you were asked to find the lowest temperature. Finding the smallest value in such a long list of numbers would require a more systematic method. Computers are especially useful for processing large amounts of data. Using a computer to solve a problem requires a carefully-specified sequence of instructions in the form of a computer program.

An ***algorithm*** is a step-by-step method for solving a problem. A description of an algorithm specifies the input to the problem, the output to the problem, and the sequence of steps to be followed to obtain the output from the input. A recipe is an example of an algorithm in which the ingredients are the input and the final dish is the output. A good recipe gives a clear sequence of steps indicating how to produce the dish from the ingredients. A description of an algorithm usually includes:

- A name for the algorithm
- A brief description of the task performed by the algorithm
- A description of the input
- A description of the output
- A sequence of steps to follow

Algorithms are often described in ***pseudocode***, which is a language in between written English and a computer language. Steps are formatted carefully as indented lists so as to convey the structure of the approach. The steps themselves are expressed in brief English phrases or mathematical symbols.

An important type of step in an algorithm is an ***assignment***, in which a variable is given a value. An assignment operator is denoted by:

```
x := y
```

The assignment statement above would change the value of x to be the current value of the variable y. It is also possible to assign a specific value to a variable as in x := 5, after which the value of x would be 5.

The output of an algorithm is specified by a ***return*** statement. For example, the statement Return(x) would return the value of x as the output of the algorithm. A return statement in the middle of an algorithm causes the execution of the algorithm to stop at that point and return the indicated value. The following animation gives an algorithm written in pseudocode to find the sum of three numbers:

PARTICIPATION ACTIVITY 8.1.1: Algorithm to compute the sum of three numbers.

Animation content:

undefined

Animation captions:

1. A description of an algorithm starts with the name of the algorithm.
2. The name is followed by a description of what the algorithm computes. The algorithm named "Sum of three" finds the sum of three numbers.
3. The next two lines describe the input and output of the algorithm, followed by the steps of the algorithm.
4. In a trial run of the algorithm on input a = 7, b = 8, c = 9. The variable "sum" is assigned the value a + b + c, which is 7 + 8 + 9 = 24.
5. The return statement returns the value of sum as the output of the algorithm, which is 24 for inputs a = 7, b = 8, and c = 9.

PARTICIPATION ACTIVITY 8.1.2: Assignment operations.

Give your answers to the following question in numerical form (e.g., "4" and not "four").

```
x := 5
y := 6
y := x
```

1) What is the value of variable x after the lines of code are executed?

Check Show answer

2) What is the value of variable y after the lines of code are executed?

Check Show answer

The if-statement and the if-else-statement

An ***if-statement*** tests a condition, and executes one or more instructions if the condition evaluates to true. In a single line if-statement, a condition and instruction appear on the same line. In the example below, if x currently has the value of 5, then y is assigned the value 7.

```
If (x = 5), y := 7
```

An if-statement can also be followed by a sequence of indented operations with a final end-if statement. In the block of code below, all n operations would be executed if the condition evaluates to true. If the condition evaluates to false, then the algorithm proceeds with the next instruction after the end-if.


```
If ( condition )
    Step 1
    Step 2
    . . .
    Step n
End-if
```

An ***if-else-statement*** tests a condition, executes one or more instructions if the condition evaluates to true, and executes a different set of instructions if the condition evaluates to false. The steps to be executed if the condition evaluates to true are indented below the if-statement. The steps to be executed if the condition evaluates for false are indented below the else statement, followed by a final end-if statement.

```
If ( condition )
    One or more steps
Else
    One or more steps
End-if
```

PARTICIPATION ACTIVITY 8.1.3: If-else-statement.

Animation content:

undefined

Animation captions:

1. The condition of the if-else statement is x < y. Since x = 3 and y = 6, the condition evaluates to true. The lines under "If" are executed.
2. The variable "min" is set to x and the variable "max" is set to y. The lines under "Else" are skipped.
3. In the next block of code, x = 7 and y = 2. The condition of the if-else statement is x < y which evaluates to false. Lines under "If" are skipped. Lines under "Else" are executed.
4. The variable "min" is set to y and max is set to x.

PARTICIPATION ACTIVITY 8.1.4: Algorithm to compute the smallest of three numbers.

Animation content:

undefined

Animation captions:

1. A description of an algorithm starts with the name of the algorithm.
2. The name is followed by a description of what the algorithm computes. The algorithm named "Smallest of three" finds the smallest of three numbers.
3. The next two lines describe the input and output of the algorithm, followed by the steps of the algorithm.
4. In a trial run of the algorithm on input a = 7, b = 5, c = 9, the first line sets min to the value of a, which is 7.
5. The next line is an if-statement that tests whether (b < min), which evaluates to true.
6. Since (b < min) is true, min is set to the value of b, which is 5.
7. The next line is an if-statement that tests whether (c < min), which evaluates to false.
8. Since (c < min) is false, the algorithm proceeds to the next line which returns the current value of min which is 5.

PARTICIPATION ACTIVITY 8.1.5: If-else-statement.

Give your answers to the following question in numerical form (e.g., "4" and not "four").

1) What is the value of abs after the following lines of code are run?

```
x := 2
If ( x > 0 )
    abs := x
Else
    abs := -x
End-if
```

Check Show answer

2) What is the value of abs after the following lines of code are run?

```
x := -2
If ( x > 0 )
    abs := x
Else
    abs := -x
End-if
```

Check Show answer

The for-loop

To solve a problem on a set of data, it is often necessary to perform an operation involving each piece of data in turn. Looping structures provide a means of specifying a sequence of steps that are to be repeated. The discussion below introduces two common kinds of loops: for-loops and while-loops.

In a ***for-loop***, a block of instructions is executed a fixed number of times as specified in the first line of the for-loop, which defines an ***index***, a starting value for the index, and a final value for the index. Each repetition of the block of instructions inside the for-loop is called an ***iteration***. The index is initially assigned a value equal to the starting value, and is incremented just before the next iteration begins. The final iteration of the for-loop happens when the index reaches the final value. In the example below, i is the index, s is the initial value, and t is the final value of the index.

```
For i = s to t
    Step 1
    Step 2
    . . .
    Step n
End-for
```

In the first iteration, i has a value of s. In the next iteration i = s + 1, and so on. In the final iteration, i = t. If t ≥ s, then the for-loop iterates t - s + 1 times. If t < s, then the for-loop is skipped entirely and the algorithm proceeds with the line after the end-for.

The animation below illustrates how the for-loop is used in an algorithm that finds the smallest number in a sequence of n numbers. The animation also shows how the algorithm works on a sample input sequence.

PARTICIPATION ACTIVITY 8.1.6: Algorithm to compute the smallest in a sequence of numbers.

Animation content:

undefined

Animation captions:

1. The algorithm finds the smallest in a sequence of numbers, a_1 through a_n. The first line sets min to the value of a_1, which is 5.
2. The for-loop has an index i which starts at 2 and goes to n. Inside the loop, an if-statement evaluates whether (a_i < 5), which is true because a_2 = 3.
3. Since the condition is true, min gets the value of a_2, which is 3. In the next iteration of the for-loop, i = 3. Again, (a_i < min) is true because a_3 = -1.
4. Min gets the value of a_3, which is -1. In the next iteration of the for-loop, i = 4 and the condition (a_i < min) is false.
5. The for-loop ends when i = n. Since n = 4 and i = 4, the algorithm proceeds to the line after the for-loop which returns current value of min = -1.

PARTICIPATION ACTIVITY 8.1.7: For-loops.

Consider the following pseudocode fragment:

```
sum := 0
For i = 2 to 5
    sum := sum + i
End-for
```

1) What is the value of "sum" after the second iteration?

Check Show answer

2) How many iterations will the for-loop execute?

Check Show answer

3) What is the final value for "sum" after executing the for-loop?

Check Show answer

The while-loop

The while-loop is similar to the for-loop in that it provides a way to specify that a sequence of steps should be repeated. A for-loop specifies the number of iterations in advance, via the beginning and ending index values. A ***while-loop*** iterates an unknown number of times, ending when a certain condition becomes false. A while-loop is written as follows:

```
While ( condition )
    Step 1
    Step 2
    . . .
    Step n
End-while
```


The condition is an expression that evaluates to true or false. If the expression evaluates to true, then steps 1 through n are performed and the algorithm goes back to the first statement where the condition is re-evaluated. The process continues until the condition evaluates to false, at which point the algorithm proceeds with the next step after the while-loop.

The following animation illustrates the while-loop in an algorithm that searches for a particular number in a sequence of numbers:

PARTICIPATION ACTIVITY 8.1.8: Algorithm to search for an item in a sequence.

Animation content:

undefined

Animation captions:

1. The algorithm searches for a number x in a sequence of numbers a_1 through a_n. The first line sets the value of i to 1.
2. The condition of the while-loop is that $a_i \neq$ x and i < n. Since input value x = 1 and a_1 = 7, $a_i \neq$ x. Also n, the number of numbers in the list is 4 and i = 1 , so i < n.
3. The line inside the while-loop, increments i by 1. The value of i is now 2 and the input value a_i = a_2 is 6.
4. The condition of the while-loop, $a_i \neq$ x and i < n, is true because 6 $\neq$ 1 and 2 < 4.
5. The line inside the loop increments i by 1. The value of i is now 3 and the input value a_i = a_3 is 1.
6. The condition of the while-loop is now false because a_i = x.
7. The next line after the while-loop is an if-statement that returns i if (a_i = x). Since i = 3 and a_3 = x, the algorithm returns 3.
8. Now the sequence is 7, 6, 1. n = 3 and x = 2. Since x is not present in the list, the while loop goes through the whole list. The condition fails when i = n.

9. In the next step, the algorithm tests whether a_3 = x. Since a_3 = 1 and x = 2, the condition is false. The final step returns -1 because x is not in the list.

PARTICIPATION ACTIVITY 8.1.9: While-loops.

Consider the following pseudocode fragment:


```
product := 1
count := 5
While ( count > 0 )
    product := product·count
    count := count - 2
End-while
```

1) What is the value of product after the second iteration?

Check Show answer

2) How many iterations will the while-loop execute?

Check Show answer

3) What is the final value for "product"?

Check Show answer

Nested loops

A ***nested loop*** is a loop that appears within another loop. The nested loop, known as the *inner loop*, is treated as a sequence of steps inside the outer loop. A nested loop is illustrated below in an algorithm that counts the number of duplicate pairs in a sequence of numbers.

PARTICIPATION ACTIVITY 8.1.10: Algorithm to count duplicate pairs in a sequence.

Animation content:

undefined

Animation captions:

1. The algorithm sets count to 0. The first iteration of the nested loops has indices i = 1 and j = 2 and tests whether (a_i = a_j): false because a_1 = 1 and a_2 = 5.
2. In the next iteration of the inside loop, j = 3. (a_i = a_j) is false because a_1 = 1 and a_3 = 6.
3. The last iteration of the inside loop is when j = 4. (a_i = a_j) is false because a_1 = 1 and a_4 = 5.
4. In the next iteration of the outside loop, i is now 2 and j is set to i+1 = 3. (a_i = a_j) is false because a_2 = 5 and a_3 = 6.
5. In the next iteration of the inside loop, j = 4. (a_i = a_j) is true because a_2 = 5 and a_4 = 5. The condition of the if-statement is true, so count is incremented by 1.
6. In the next iteration of the outside loop, i is now 3 and j is set to i+1 = 4. (a_i = a_j) is false because a_3 = 6 and a_4 = 5.
7. Both for-loops are finished and the algorithm returns the current value of "count" which is 1. There is one pair of duplicates in the list (a_2 = a_4).

PARTICIPATION ACTIVITY 8.1.11: Nested loops - example 1.

Consider the following pseudocode fragment:

```
count := 0
For i = 1 to 3
    For j = 1 to 4
        count := count + i · j
    End-for
End-for
```


1) How many times is the variable "count" increased?

Check Show answer

2) What is the final value of "count"?

Check Show answer

PARTICIPATION ACTIVITY 8.1.12: Nested loops - example 2.

Consider the following pseudocode fragment:

```
count := 0
For i = 1 to 3
    For j = i+1 to 4
        count := count + i * j
    End-for
End-for
```

1) For the iteration of the outer loop where i = 2, how many times does the inner loop iterate?

Check Show answer

2) How many times is the variable "count" increased?

Check Show answer

3) What is the final value of "count"?

Check Show answer

Additional exercises

EXERCISE 8.1.1: Writing algorithms in pseudocode.

Write an algorithm in pseudocode for each description of the input and output.

(a) Input: $a_1, a_2, \ldots, a_n$, a sequence of numbers, where $n \geq 1$
n, the length of the sequence.
Output: "True" if the sequence is non-decreasing and "False" otherwise.
A sequence of numbers is non-decreasing if each number is at least as large as the one before.

(b) Input: $a_1, a_2, \ldots, a_n$, a sequence of numbers, where $n \geq 1$
n, the length of the sequence.
Output: "True" if there are two consecutive numbers in the sequence that are the same and "False" otherwise.

(c) Input: $a_1, a_2, \ldots, a_n$, a sequence of numbers, where $n \geq 1$
n, the length of the sequence.
Output: "True" if there are any two numbers in the sequence whose sum is 0 and "False" otherwise.

(d) Input: $a_1, a_2, \ldots, a_n$, a sequence of numbers, where $n \geq 1$
n, the length of the sequence.
Output: "True" if there are any three numbers in the sequence that form a Pythagorean triple.
The numbers x, y, and z are a Pythagorean triple if $x^2 + y^2 = z^2$.

(e) Input: $a_1, a_2, \ldots, a_n$, a sequence of distinct numbers, where $n \geq 2$
n, the length of the sequence.
Output: The second smallest number in the sequence.

8.2 Asymptotic growth of functions

This section has been set as optional by your instructor.

Let f be a function that maps positive integers to non-negative real numbers(f: $\mathbf{Z^+} \rightarrow \mathbf{R^{\geq}}$). The symbol $\mathbf{R^{\geq}}$ denotes the set of non-negative real numbers: $\mathbf{R^{\geq}} = \mathbf{R^+} \cup \{0\}$. The ***asymptotic growth*** of the function f is a measure of how fast the output f(n) grows as the input n grows. The classification of functions using Oh, Ω, and Θ notation (called ***asymptotic notation***) provides a way to concisely characterize the asymptotic growth of a function. Asymptotic notation is a useful tool for evaluating the efficiency of algorithms.

The notation f = O(g) is read "f is ***Oh of*** g". f = O(g) essentially means that the function f(n) is less than or equal to g(n), if constant factors are omitted and small values for n are ignored. In the expressions $7n^3$ and $5n^2$, the 7 and the 5 are called ***constant factors*** because the values of 7 and 5 do not depend on the variable n. If f is O(g), then there is a positive number c such that when f(n) and c·g(n) are graphed, the graph of c·g(n) will eventually cross f(n) and remain higher than f(n) as n gets large.

Consider the function $f(n) = 2n^3 + 3n^2 + 7$. There are many functions g(n) such that f = O(g). For example, f(n) is $O(4n^3 + 2n + 1)$. However, the idea is to select a function g that characterizes the asymptotic growth of f as simply as possible. Therefore, g is selected so as to eliminate all unnecessary constants and additional terms. Instead of saying that $f = O(4n^3 + 2n + 1)$, one would typically say that $f = O(n^3)$.

Sometimes the functions in this material will evaluate to a negative number, such as n/2 - 10, when n < 20. In order to ensure that every function maps to a non-negative number for every possible input from $\mathbf{Z^+}$, the output value can be replaced by 0 if the output value is less than or equal to 0. Thus, the function f(n) really means max { f(n), 0 }.

Definition 8.2.1: Oh notation.

Let f and g be functions from $\mathbf{Z^+}$ to $\mathbf{R^{\geq}}$. Then f = O(g) if there are positive real numbers c and n_0 such that for any n ∈ $\mathbf{Z^+}$ such that n ≥ n_0,

$$f(n) \leq c \cdot g(n).$$

PARTICIPATION ACTIVITY 8.2.1: Oh notation example.

Animation content:

undefined

Animation captions:

1. $f(n) = 2n^3 + 3n^2 + 7$. $g(n) = n^3$. To prove that f is O(g), pick c = 3, n_0 = 4.
2. 3g(n) is larger than f(n) for $n \geq n_0$.

The animation shown above provides intuition for what it means for a function f to be O(g). However, a mathematical proof is required to formally establish that a function f(n) is Oh of another function g(n).

Proof 8.2.1: Proof that f is O(g).

$f(n) = 3n^3 + 5n^2 - 7$
$g(n) = n^3$

Claim: $f = O(g)$.

Proof.
Select $c = 8$ and $n_0 = 1$. We will show that for any $n \geq 1$, $f(n) \leq 8 \cdot g(n)$.

$$f(n) = 3n^3 + 5n^2 - 7 \leq 3n^3 + 5n^2$$

For $n \geq 1$, $n^2 \leq n^3$, so

$$3n^3 + 5n^2 \leq 3n^3 + 5n^3$$

Finally, $3n^3 + 5n^3 = 8n^3 = 8 \cdot g(n)$. Putting the inequalities together, we get that for any $n \geq 1$,

$$f(n) = 3n^3 + 5n^2 - 7 \leq 3n^3 + 5n^2 \leq 3n^3 + 5n^3 = 8n^3 = 8 \cdot g(n)$$

and therefore, $f(n) \leq 8 \cdot g(n)$ which means that $f = O(g)$. ■

Video explaining the steps in the proof.

The constants c and n_0 in the definition of Oh-notation are said to be a ***witness*** to the fact that $f = O(g)$. In a proof that $f = O(g)$, there are many different choices for the witness c and n_0 that will suffice. The proof given above that $3n^3 + 5n^2 - 7$ is $O(n^3)$ used witness $c = 8$ and $n_0 = 1$. The combination $c = 4$ and $n_0 = 5$ would have also worked, although the algebra in the proof would have been more involved. In showing that a polynomial function f(n) of degree k is $O(n^k)$, the following combination will work as a witness:

- $n_0 = 1$
- c = the sum of the positive coefficients in f (including the constant term)

For example, if $f(n) = 3n^5 + 7n^4 - 3n^3 + 2$, a proof that f is $O(n^5)$ could use $c = 3 + 7 + 2 = 12$ and $n_0 = 1$.

PARTICIPATION ACTIVITY 8.2.2: Proving $f = O(g)$, where f and g are polynomials.

Define the functions

$f(n) = 5n^6 - 4n^4 + 2n^2 + 4$
$g(n) = 4n^6 + 3n^5 + 2n^2 + 1$

1) Indicate whether the following statement is true or false:
For any $n \geq 1$, $f(n) \leq 11 \cdot n^6$.

- ○ True

- False

2) Indicate whether the following statement is true or false:
For any n ≥ 1, g(n) ≤ 9·n^6.

- True
- False

3) Is it true that f(n) = O(n^7)?

- True
- False

Proving that f is not O(g)

By definition, if f = O(g), then there must be a witness, c and n_0, showing that f = O(g). Showing that f is not O(g) requires showing that every possible combination of values for c and n_0 fails to be a witness. A proof that f is not O(g) must establish that for every n_0 and c, there is a value of n such that n ≥ n_0 and f(n) > c · g(n).

PARTICIPATION ACTIVITY 8.2.3: A proof that f is not O(g).

Animation content:

undefined

Animation captions:

1. $f(n) = \frac{n^3}{2} - 2n^2 + 3$. $g(n) = n^2$.Prove that f is not O(g) by contradiction. Suppose that for constants c and n_0, and all $n \geq n_0$, $\frac{n^3}{2} - 2n^2 + 3 \leq c \cdot n^2$.
2. If $c \cdot n^2$ is bigger than $\frac{n^3}{2} - 2n^2 + 3$, then $c \cdot n^2$ is also bigger than $\frac{n^3}{2} - 2n^2$. Therefore, $\frac{n^3}{2} - 2n^2 \leq c \cdot n^2$.
3. Each side of the last inequality is divided by n^2.
4. The resulting inequality is n/2 - 2 ≤ c. Now 2 is added to both sides.
5. The resulting inequality is n/2 ≤ c + 2. Both sides are multiplied by 2 to get n ≤ 2c + 4.
6. When n is the maximum of n_0 and 2c + 5, the last inequality is false. All steps can be reversed, so f(n) > c · g(n) for some n > n_0, a contradiction.

PARTICIPATION ACTIVITY 8.2.4: Proofs that for functions f and g, f is not O(g).

1) A proof that f is not O(g), must show that it is **not** the case that:

   There are positive constants c and n_0 such that for every $n \geq n_0$, $f(n) \leq c \cdot g(n)$.

   Which statement is equivalent to the statement that f is not O(g)?

   - For every choice of positive values for c and n_0, there is an $n \geq n_0$ such that $f(n) \leq c \cdot g(n)$.
   - For every choice of positive values for c and n_0, there is an $n \geq n_0$ such that $f(n) > c \cdot g(n)$.
   - There are constants c and n_0 such that for every $n \geq n_0$, $f(n) > c \cdot g(n)$.

2) Is $n^2 = O(n)$?

   - Yes
   - No

Ω notation

The O notation serves as a rough upper bound for functions (disregarding constants and small input values). The Ω notation is similar, except that it provides a lower bound on the growth of a function:

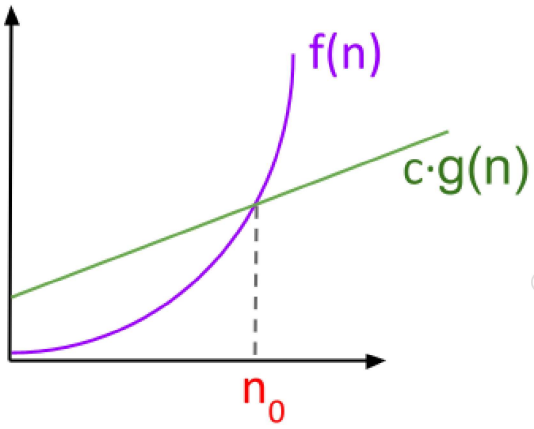
Definition 8.2.2: Ω Notation.

Let f and g be functions from $\mathbf{Z^+}$ to $\mathbf{R^{\geq}}$. Then f = Ω(g) if there are positive real numbers c and n_0 such that for any $n \in \mathbf{Z^+}$ such that for $n \geq n_0$,

$$f(n) \geq c \cdot g(n).$$

The notation f = Ω(g) is read "f is ***Omega*** of g".

Figure 8.2.1: f is Ω(g).



There is a natural relationship between Oh-notation and Ω-notation, stated in the following theorem:

Theorem 8.2.1: Relationship of Oh-notation and Ω-notation.

Let f and g be functions from $\mathbf{Z^+}$ to $\mathbf{R^{\geq}}$. Then f = Ω(g) if and only if g = O(f).

In a proof that f is Ω(g), there are many different choices for the constants c and n_0 that will suffice as a witness. If f(n) is a polynomial of degree k, then f = $\Omega(n^k)$ only if the coefficient of the n^k term in f (call it a_k) is positive. Here are combinations for c and n_0 that suffice as a witness to show that f = $\Omega(n^k)$.

- If f has no negative coefficients, then c = a_k and n_0 = 1 suffice.
- If f has negative coefficients (but $a_k > 0$), then let A be the sum of the absolute values of the negative coefficients in f(n). The choices c = $a_k/2$ and $n_0 = \max\{1, 2A/(a_k)\}$ are sufficient.

For example, if f(n) = $(1/7)\cdot n^6 + 2n^5 + 3$, then the combination c = 1/7 and n_0 = 1 will suffice as a witness to show that f is $\Omega(n^6)$. If f(n) = $7n^6 - 2n^5 - 3$, then the combination c = 7/2 and $n_0 = 2\cdot(|-2| + |-3|)/7 = 10/7$ will suffice to show that f is $\Omega(n^6)$. If f(n) = $-7n^6 + 2n^5 + 3$, then f is not $\Omega(n^6)$.

The following videos provide examples that illustrate why the values given above suffice to show that a a polynomial of degree k with a positive coefficient for x^k is $\Omega(n^k)$. The first example is a polynomial whose coefficients are all non-negative. The second example is a polynomial with some negative coefficients. Examples of formal proofs are given below.

Proof 8.2.2: Proof that f is Ω(g) for f and g with all non-negative coefficients.

f(n) = $(1/2)n^2 + 7n + 3$

$g(n) = n^2$

Claim: $f = \Omega(g)$.

Proof.
Select $c = 1/2$ and $n_0 = 1$. We will show that for any $n \geq 1$, $f(n) \geq (1/2) \cdot g(n)$.

Since $n \geq 1$, $7n \geq 0$. Adding the inequalities $7n \geq 0$ and $3 \geq 0$ gives that

$$7n + 3 \geq 0$$

Add $(1/2)n^2$ to both sides to get

$$(1/2)n^2 + 7n + 3 \geq (1/2)n^2$$

Therefore, for $n \geq 1$, $f(n) \geq (1/2) \cdot g(n)$ which means that $f = \Omega(g)$. ■

Proof 8.2.3: Proof that f is Ω(g) for f and g with some negative coefficients.

$f(n) = n^2 - 7n - 3$
$g(n) = n^2$

Claim: $f = \Omega(g)$.

Proof.
Select $c = 1/2$ and $n_0 = 20$. We will show that for any $n \geq 20$, $2 \cdot f(n) \geq g(n)$, which implies that $f(n) \geq (1/2) \cdot g(n)$. Plugging in the definitions for $f(n)$ and $g(n)$, into $2 \cdot f(n) \geq g(n)$, the goal is to show that:

$$2n^2 - 14n - 6 \geq n^2$$

Since $n \geq 20$, it is also true that $n \geq 1$. Start with the inequality

$$n \geq 1$$

Multiply both sides by -6 to get

$$-6n \leq -6.$$

Add $(2n^2 - 14n)$ to both sides to get:

$$2n^2 - 14n - 6 \geq 2n^2 - 14n - 6n$$

Using basic algebra:

$$2n^2 - 14n - 6n = n(2n - 14 - 6) = n(2n - 20).$$

We need to show that $(2n - 20) \geq n$ in order to show that $n(2n - 20) \geq n^2$. Take the inequality $n \geq 20$, add n and subtract 20 from each side to get that $(2n - 20) \geq n$. Multiply both sides by n to get:

$n(2n - 20) \geq n^2$.

Putting together all the inequalities, we get that for $n \geq 20$,

$$2 \cdot f(n) = 2n^2 - 14n - 6 \geq 2n^2 - 14n - 6n = n(2n - 20) \geq n^2 = g(n)$$

and therefore, for $n \geq 20$, $f(n) \geq (1/2) \cdot g(n)$ which means that $f = \Omega(g)$. ■

PARTICIPATION ACTIVITY 8.2.5: Showing f is Ω(g).

Indicate whether each statement is true or false.

1) The values $c = 1$ and $n_0 = 1$ are sufficient to show that $f(n) = n^3 + n$ is $\Omega(n^3)$.
   - True
   - False

2) The values $c = 1/2$ and $n_0 = 1$ are sufficient to show that $f(n) = n^3 - (3/4)n$ is $\Omega(n^3)$.
   - True
   - False

3) The values $c = 1/2$ and $n_0 = 2$ are sufficient to show that $f(n) = n^3 - n$ is $\Omega(n^3)$.
   - True
   - False

Θ notation and polynomials

The Θ notation indicates that two functions have the same rate of growth.

Definition 8.2.3: Θ Notation.

Let f and g be functions from $\mathbf{Z^+}$ to $\mathbf{R^{\geq}}$.

$f = \Theta(g)$ if $f = O(g)$ and $f = \Omega(g)$.

The notation f = Θ(g) is read "f is ***Theta of*** g(n)".

If f = Θ(g), then f is said to be ***order of*** g. For example $f(n) = 4n^3 + 7n + 16$ is order of n^3. The terms 7n and 16 are called the ***lower order*** terms of the function $f(n) = 4n^3 + 7n + 16$ because removing those terms from f does not change the order of f.

The examples given so far are similar in that they are all polynomials. The following theorem establishes a general rule for bounding the growth of polynomials. The proof of the theorem is left as an exercise.

Theorem 8.2.2: Asymptotic growth of polynomials.

Let p(n) be a degree-k polynomial of the form

$$p(n) = a_k n^k + a_{k-1} n^{k-1} + \ldots + a_1 n + a_0$$

in which $a_k > 0$. Then p(n) is $\Theta(n^k)$.

PARTICIPATION ACTIVITY 8.2.6: Growth rates of polynomials.

$f(n) = n^5 + 4n - 3$
$g(n) = 1001 \cdot n - 100$
$h(n) = 4 \cdot n^2 - n + 3$

1) Is $f = \Omega(n^4)$?

- Yes
- No

2) Is $f = O(n^4)$?

- Yes
- No

3) Is $g = \Theta(n)$?

- Yes
- No

4) Is $h = \Omega(n^2)$?

- Yes
- No

Asymptotic growth of logarithm functions with different bases

Let a and b be two real numbers greater than 1, then

$$\log_a n = \Theta(\log_b n)$$

For example, $\log_5 n = \Theta(\log_{17} n)$. The fact that $\log_a n$ and $\log_b n$ have the same asymptotic growth rate (as long as a and b are both strictly greater than 1) follows from two facts about the logarithm function:

- If a and b are constants greater than 1, then $\log_a b$ is a positive constant.
- $\log_a n = \log_a b \cdot \log_b n$

When a function is said to be Oh of or Ω of a logarithmic function, the base is often omitted because it is understood that as long as the constant in the base is greater than 1, the value of the base does not affect the asymptotic growth of the function. For example the function $17(\log_4 n) + 4 = \Theta(\log n)$.

PARTICIPATION ACTIVITY 8.2.7: Asymptotic growth of logs.

1) Which function is not $\Theta(\log n)$?

- $\log_{17} n$
- $\log_{(\log n)} n$
- $\log_{1.001} n$

The growth rate of common functions in analysis of algorithms

A function that does not depend on n at all is called a ***constant function***. f(n) = 17 is an example of a constant function. Any constant function is $\Theta(1)$.

Certain types of functions are encountered frequently in analyzing the complexity of algorithms. These functions are common enough that they have names that describe their rate of growth. For example, if a function f(n) is $\Theta(n)$, we say that f(n) is a "linear" function. The table below gives a list of common functions and their names. Except for polynomials which include linear, quadratic and cubic functions, the functions are ordered according to the rate of growth, so each function is Oh of the functions below it in the table but not Ω-of the functions lower in the table. For example, n^2 is $O(n^3)$, but n^2 is not $\Omega(n^3)$. Also n! is $\Omega(\log n)$ but $(\log n)$ is not $\Omega(n!)$.

A function f(n) is said to be ***polynomial*** if f(n) is $\Theta(n^k)$ for some positive real number k. The class of polynomial functions includes the classes of linear, quadratic, and cubic functions. A function f(n) is said to be ***exponential*** if f(n) is $\Theta(c^n)$ for some real number c > 1. $f(n) = 2^n$ is an example of an exponential function. Every polynomial function is Oh of every exponential function, which also means that every exponential function is Ω of every polynomial function. There is no polynomial function that is Ω of any exponential function, which also means that there is no exponential function which is Oh of any polynomial function.

A word of warning: while the definitions allow for dropping constant factors in determining the growth rate of functions, the constant in the base of an exponential function is important. For example, it is not the case that 3^n is $O(2^n)$. Although, since $2 \leq 3$, it is true that 2^n is $O(3^n)$.

Table 8.2.1: Common functions in algorithmic complexity.

The functions below are given in increasing order of complexity, with the m in the third from the bottom being any m > 3.

Function	Name
$\Theta(1)$	Constant
$\Theta(\log \log n)$	Log log
$\Theta(\log n)$	Logarithmic
$\Theta(n)$	Linear
$\Theta(n \log n)$	n log n
$\Theta(n^2)$	Quadratic
$\Theta(n^3)$	Cubic
$\Theta(n^m)$ for a positive integer m	Power
$\Theta(c^n)$, c > 1	Exponential
$\Theta(n!)$	Factorial

PARTICIPATION ACTIVITY 8.2.8: Asymptotic growth of common functions.

1) 2^n is $O(n^3)$

- True
- False

2) 2^n is $\Omega(\log n)$

- True
- False

3) n^{20} is $O((1.1)^n)$

- True
- False

4) n^{20} is $\Omega((1.1)^n)$

- True
- False

Rules about asymptotic growth

There are a few rules that are useful in determining the growth rate of functions that are sums of or constant multiples of the standard functions given in the table.

Figure 8.2.2: Rules for the asymptotic growth of functions.

Let f, g, and h be functions from $\mathbf{Z}^+$ to $\mathbf{R}^{\geq}$:

- If $f = O(h)$ and $g = O(h)$, then $f+g = O(h)$.
- If $f = \Omega(h)$ or $g = \Omega(h)$, then $f+g = \Omega(h)$.
- If $f = O(g)$ and c is a positive real number, then $c \cdot f = O(g)$.
- If $f = \Omega(g)$ and c is a positive real number, then $c \cdot f = \Omega(g)$.
- If $f = O(g)$ and $g = O(h)$, then $f = O(h)$.
- If $f = \Omega(g)$ and $g = \Omega(h)$, then $f = \Omega(h)$.

PARTICIPATION ACTIVITY 8.2.9: Asymptotic growth of sums of functions.

Animation content:

undefined

Animation captions:

1. f(n) = $5n^3$ + 16(n log n) + 5·2^n. $5n^3$ is O(n^3) and n^3 is O(2^n). Therefore $5n^3$ is O(2^n).
2. 16(n log n) is O(n log n) and (n log n) is O(2^n). Therefore 16(n log n) is O(2^n). Also, 5·2^n is O(2^n).
3. $5n^3$, 16(n log n), and 5·2^n are all O(2^n). Therefore, f(n) = $5n^3$ + 16(n log n) + 5·2^n is also O(2^n).
4. Since 5·2^n is $\Omega(2^n)$, f(n) = $5n^3$ + 16(n log n) + 5·2^n is also $\Omega(2^n)$. Since f(n) is O(2^n) and Ω(2^n), f(n) is $\Theta(2^n)$.

PARTICIPATION ACTIVITY 8.2.10: Growth rates of functions.

f(n) = 5·2^n + n + 3
g(n) = 10(log n) + n! + n^2
h(n) = 7(log log n) + 17(n log n)

1) Is f = $\Omega(n^{1000})$?
   - Yes
   - No

2) Is h = $\Theta(n^2)$?
   - Yes
   - No

3) Is f = O(n!)?
   - Yes
   - No

4) Is g = $\Omega(n^2)$?
   - Yes
   - No

5) Is h = O(1)?
   - Yes
   - No

CHALLENGE ACTIVITY | 8.2.1: Rate of growth.

455912.3056722.qx3zqy7

Start

Order the functions by growth rate.

$a(x) = x^2$

$b(x) = 2300^7 \cdot 35$

$c(x) = 3x$

$d(x) = 20x!$

[ ] ≥ [ ] ≥ [ ] ≥ [ ]

1 2 3 4

Check Next

Additional exercises

EXERCISE | 8.2.1: Characterizing the growth rate of functions.

Characterize the rate of growth of each function f below by giving a function g such that f = Θ(g). The function g should be one of the functions in the table of common functions.

(a) $f(n) = n^8 + 3n - 4$

(b) $f(n) = 2 \cdot 3^n$

(c) $f(n) = 2^n + 3^n$

(d) $f(n) = 7(\log \log n) + 3(\log n) + 12n$

(e) $f(n) = 9(n \log n) + 5(\log \log n) + 5$

(f) $f(n) = n \cdot \log_{37} n$

(g) $f(n) = n^{21} + (1.1)^n$

(h) $f(n) = 23n + n^3 - 2$

EXERCISE 8.2.2: Proving the growth rate for polynomials.

Give complete proofs for the growth rates of the polynomials below. You should provide specific values for c and n_0 and prove algebraically that the functions satisfy the definitions for O and Ω.

(a) $f(n) = (1/2)n^5 - 100n^3 + 3n - 1$. Prove that $f = \Theta(n^5)$.

(b) $f(n) = n^3 + 3n^2 + 4$. Prove that $f = \Theta(n^3)$.

EXERCISE 8.2.3: Proving negative results on the growth of functions.

(a) $f(n) = n/100$. Prove that it is not true that $f = O(\sqrt{n})$.

(b) $f(n) = n^{(3/2)}$. Prove that it is not true that $f = \Omega(n^2)$.

EXERCISE 8.2.4: Relationships between bounds on the growth rate of functions.

(a) Let f be a function whose domain is $\mathbf{Z}^+$ and whose target is $\mathbf{R}^{\geq}$. Show that if $f = O(n)$, then $f = O(n^2)$.

(b) Let f be a function whose domain is $\mathbf{Z}^+$ and whose target is $\mathbf{R}^{\geq}$. Show that if $f = O(n)$, then it is not true that $f = \Omega(n^2)$.

8.3 Analysis of algorithms

This section has been set as optional by your instructor.

An algorithm describes the underlying method for how a computational problem is solved. The choice of algorithm can have a dramatic effect on how efficiently the solution is obtained. The amount of resources used by an algorithm is referred to as the algorithm's ***computational complexity***. The primary resources to optimize are the time the algorithm requires to run (time complexity) and the amount of memory used (***space complexity***). This material focuses on the time complexity of algorithms.

Naturally, a program will take more time on larger inputs. For example, a program that sorts a sequence of numbers will take a longer time to sort a long sequence than a short sequence. Therefore, time and space efficiency are measured in terms of the input size. For the problem of sorting numbers, a natural measure of the input size is the number of numbers to be sorted. For a different problem, the size of the input may depend on a different feature of the input. For example, a program that processes a digital image would likely be measured as a function of the number of pixels in the image. The input size for a problem is usually denoted by the variable n.

Atomic operations (assignment, arithmetic operations, comparison, return statements etc.) form the basic building blocks for the pseudocode. Once a set of atomic operations is identified, it is possible to define the time complexity of an algorithm as a function:

Definition 8.3.1: Time complexity as a function.

The ***time complexity*** of an algorithm is defined by a function f: $\mathbf{Z^+} \rightarrow \mathbf{Z^+}$ such that f(n) is the maximum number of atomic operations performed by the algorithm on any input of size n. $\mathbf{Z^+}$ is the set of positive integers.

Although the computational complexity of an algorithm is defined by functions that map positive integers to positive integers, sometimes algorithmic complexity is described in terms of functions that are real valued or negative for small input values (e.g., $\sqrt{n}$ or n^2 - 10). A function f(n) in the context of computational complexity means $\max\{\lceil f(n) \rceil, 1\}$. Also, some functions are not well-defined for small values of n, such as log(log(1)). In those cases, the value of the function is assumed to be 1.

PARTICIPATION ACTIVITY 8.3.1: Counting atomic operations for an algorithm ComputeSum.

Animation content:

undefined

Animation captions:

1. In the "ComputeSum" algorithm, the first line is an assignment which counts as 1 operation. The next line is a for-loop which iterates n times.
2. Each iteration of the for-loop does 4 operations: 2 to test and increment the counter i, and an addition and assignment inside the loop.
3. The line after the for-loop is a return statement which counts as 1 operation.
4. f(n) is the number of operations executed on a sequence of length n. f(n) = 1 + n(2 + 2) + 1 = 4n + 2.

PARTICIPATION ACTIVITY 8.3.2: Number of atomic operations performed by ComputeSum.

1) How many atomic operations are performed by the algorithm ComputeSum on a sequence of 100 numbers?

Check Show answer

Asymptotic complexity

A programmer writing a computer program that implements an algorithm may reduce execution time by optimizing the code to favor instructions that execute in less time than others, or by reducing the number of atomic operations inside a looping structure. Such optimizations may improve the function f(n) from $3n^2$ to $2n^2$. On the other hand, finding an entirely new approach to the problem could result in an algorithm that requires only 3n operations on an input of size n, which would have a more dramatic effect on the running time. Another consideration in evaluating algorithms is that the efficiency of the algorithm is most critical for large input sizes. Small inputs are likely to result in fast running times, so efficiency is less of a concern.

In evaluating algorithms, the focus is on how the function f grows with n, ignoring small input sizes and constant factors that depend on the specifics of the implementation and have less impact on the execution time.

The ***asymptotic time complexity*** of an algorithm is the rate of asymptotic growth of the algorithm's time complexity function f(n).

The figure below illustrates how dramatically the running times of algorithms with different asymptotic time complexity can vary. The figure shows how long it takes to perform f(n) instructions for different functions f and different values of n. For large n, the difference in computation time varies greatly with the rate of growth of the function f.

Table 8.3.1: Growth rates for different input sizes.

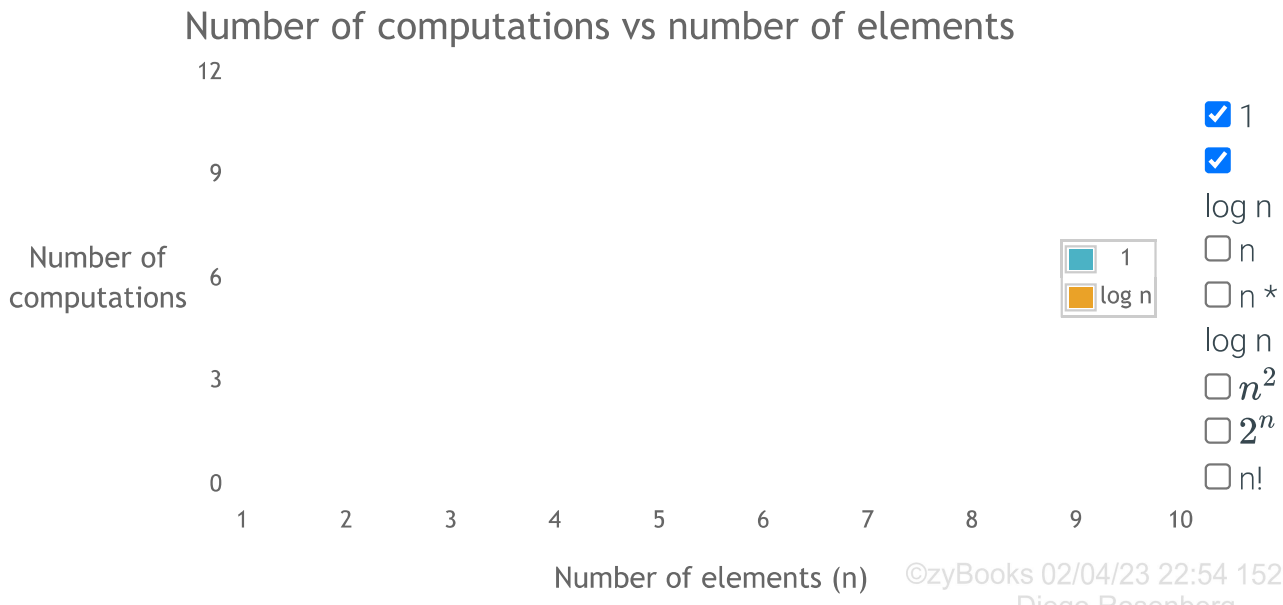
f(n)	n=10	n=50	n=100	n=1000	n=10000	n=100000
$\log_2 n$	3.3 µs	5.6 µs	6.6 µs	10.0 µs	13.3 µs	16.6 µs
n	10 µs	50 µs	100 µs	1000 µs	10 ms	.1 s

$n \log_2 n$	.03ms	.28 ms	.66 ms	10.0 ms	.133 s	1.67 s
n^2	.1 ms	2.5 ms	10 ms	1 s	100 s	2.8 hours
n^3	1 ms	.125 s	1 s	16.7 min	11.6 days	31.7 years
2^n	1.0 ms	35.7 years	4.0×10^{16} years	3.4×10^{287} years	6.3×10^{2996} years	*

This data assumes that a single instruction takes 1 µs to execute. A µs is a microsecond and is equal to 10^{-6} seconds. An ms is a millisecond and is equal to 10^{-3} seconds.

The interactive tool below illustrates graphically the growth rate of commonly encountered functions. For example, the growth rate of 2^n is much larger than n^2 and the growth rate of n! is much larger than 2^n.

PARTICIPATION ACTIVITY 8.3.3: Computational complexity graphing tool.



PARTICIPATION ACTIVITY 8.3.4: Running time for different growth rates.

Use the table above to answer the following questions. Assume that each atomic operation requires 1 µs to execute.

1) On an input of size n = 100000, how long would it take an algorithm that runs in time $f(n) = n^2$?

- 1.67 s
- 2.8 hours
- 31.7 years

2) On an input of size n = 100000, how long would it take an algorithm that runs in time $f(n) = n^3$?

- 1.67 s
- 2.8 hours
- 31.7 years

It may seem simplistic that all atomic operations are counted as +1 in determining the time complexity of an algorithm because in reality, some operations do take longer than others. In addition, some arbitrary decisions were made in counting atomic operations. For example, in the line "For i = 1 to n" we counted the cost of incrementing the counter i as 1 operation. However, computing "i := i+1" is technically an addition and an assignment operation, which should count as two operations. Fortunately, the distinction is not important if the goal is to understand the asymptotic complexity of the algorithm because the functions 5n + 2 and 4n + 2 are both Θ(n). In fact, it is sufficient just to determine that the number of operations per loop is some constant c and that the number of operations before and after the loop is another constant d. Since the loop is executed exactly n times on an input of size n, the number of atomic operations that will be performed is $f(n) = c \cdot n + d = \Theta(n)$.

The animation below shows the asymptotic analysis for the algorithm to find the smallest of n numbers. The time complexity of the algorithm is Θ(n). Inside the loop, the number of atomic operations performed will actually depend upon the values of the input sequence. If the condition of the if-statement inside the loop evaluates to true, then an additional assignment is performed. If the condition evaluates to false, then the assignment is skipped. However, since the number of atomic operations performed inside the loop is at least 3 and at most 4, for all input values, the variation does not affect the asymptotic running time of the algorithm.

PARTICIPATION ACTIVITY | 8.3.5: Analysis of the algorithm to find the smallest in a sequence of numbers.

Animation content:

undefined

Animation captions:

1. In the "FindSmallest" algorithm, the first line is an assignment which counts as 1 operation. The next line is a for-loop which iterates n-1 times.
2. Each iteration of the for-loop does 2 ops to test and increment the counter. 1 or 2 operations are performed inside the loop. A constant c number of operations are performed per iteration.
3. The line after the for-loop is a return statement, which counts as 1 operation. There are a constant d number of operations performed before and after the for-loop.
4. f(n) is the number of operations executed on a sequence of length n. f(n) = (n-1)c + d = nc - c + d. f(n) is $\Theta(n)$.

PARTICIPATION ACTIVITY 8.3.6: Asymptotic analysis of algorithms on sequences.

1) What is the asymptotic time complexity of the following algorithm:

```
SequenceProduct
This algorithm finds the product of
all the elements in a sequence of n
numbers.

Input: a_1, a_2,...,a_n
       n, the length of the sequence.
Output: the product of all numbers
in the sequence

prod:= 1

For i = 1 to n
    prod:= prod·a_i
End-for

Return( prod )
```

- $\Theta(1)$
- $\Theta(n)$
- $\Theta(n^2)$

2) What is the asymptotic time complexity of the following algorithm:

AverageOfEnds
This algorithm finds the average of the first and last elements in a sequence.

Input: $a_1, a_2,...,a_n$
n, the length of the sequence.
Output: the average of the first and last elements in the sequence

sum := $a_1 + a_n$
avg := sum / 2

Return(avg)

- Θ(1)
- Θ(n)
- Θ(n^2)

Worst-case complexity

The number of operations performed by the previous algorithm FindSmallest may depend on the actual numbers in the input sequence, not just the size of the input. However, the variation does not affect the asymptotic running time of the algorithm which is Θ(n) for any input sequence. The next example is an algorithm that searches for a particular item x in a sequence of n numbers. The number of operations performed on different inputs of the same size can vary more dramatically.

Figure 8.3.1: Searching a sequence.

SearchSequence

This algorithm returns the index of the first occurrence of a particular number in a sequence of numbers.
If the number does not occur in the sequence, the algorithm returns -1.

Input: A value x to search for.
A sequence $a_1, a_2,\ldots,a_n$.
n, the length of the sequence
Output: the index of the first occurrence of x in the sequence, or -1 if x is not in the sequence.


```
i := 1
While ( a_i ≠ x and i < n )
    i := i+1
End-while

If ( a_i = x ), Return( i )
Return( -1 )
```

PARTICIPATION ACTIVITY 8.3.7: The number of iterations in the SearchSequence algorithm.

For each input (value for x and sequence $a_1,\ldots,a_n$) given, indicate the number of times the instruction inside the while loop (i := i + 1) is executed.

1) x = 5. The sequence is: 3, 2, 5, 7, 1

Check Show answer

2) x = 5. The sequence is: 5, 2, 3, 7, 1

Check Show answer

3) x = 5. The sequence is: 6, 2, 3, 7, 1

Check Show answer

The algorithm starts at the beginning of the list and compares each item a_i to x, starting with a_1. If x occurs early in the list (for example, if a_1 = x), then the algorithm returns right away after only performing O(1) operations. However, if the first occurrence of x is last in the list or if x does not occur in the list at all, the algorithm must check every item in the list which takes Ω(n) time. Thus, the time complexity of the algorithm depends greatly on the input. What is the correct complexity of the algorithm?

The most common way to analyze the complexity of algorithms whose run time varies with different features of the input is to count the number of operations performed in the worst (most time-consuming) case. The ***worst-case analysis*** of an algorithm evaluates the time complexity of the algorithm on the input of a particular size that takes the longest time.

The ***worst-case*** time complexity function f(n) is defined to be the maximum number of atomic operations the algorithm requires, where the maximum is taken over all inputs of size n.

The input of a given size that maximizes f(n) is the worst-case input for the algorithm. For the searching algorithm given above, the worst-case input is when the item x does not occur in the list at all. Determining an upper bound and a lower bound for the time complexity of an algorithm require two different tasks:

- When proving an ***upper bound*** on the worst-case complexity of an algorithm (using Oh-notation), the upper bound must apply for every input of size n.
- When proving a ***lower bound*** for the worst-case complexity of an algorithm (using Ω notation), the lower bound need only apply for at least one possible input of size n.

Worst-case analysis can lead to overly pessimistic results in some cases if the worst-case input for an algorithm is highly unusual and the algorithm takes much less time on typical inputs. Average-case analysis is an alternative to worst-case analysis that tries to address this shortcoming. ***Average-case analysis*** evaluates the time complexity of an algorithm by determining the average running time of the algorithm on a random input. Average case analysis uses probability theory to formally define a "random" input and is not covered in this material.

PARTICIPATION ACTIVITY 8.3.8: Analysis of the algorithm SearchSequence.

Animation content:

undefined

Animation captions:

1. In the "SearchSequence" algorithm, there is 1 operation before the while-loop and 2 after. The while-loop executes 4 operations per iteration.

2. The while-loop iterates at most n-1 times. If x is not present in the list, then the while-loop executes exactly n-1 times.
3. f(n) is the number of operations on a sequence of length n.
   $f(n) \leq 1 + (n-1) \cdot 4 + 2 = 4n - 1 = O(n)$.
4. In the worst case, $f(n) \geq 1 + 4(n-1) + 2 = 4n - 1 = \Omega(n)$.

PARTICIPATION ACTIVITY 8.3.9: Determining the worst case input for SearchSequence.

1) Each of the choices is a possible input to the algorithm SearchSequence. The size of the input is the same for all three choices. Which input will result in the worst case running time for the algorithm?

- x = 9. The sequence is 4, 1, 6, 8, 3, 5, 2.
- x = 9. The sequence is 4, 9, 6, 8, 3, 5, 2.
- x = 9. The sequence is 4, 1, 6, 8, 9, 5, 2.

Analyzing an algorithm with nested loops

Figure 8.3.2: Algorithm to count number of duplicate pairs in a sequence.

CountDuplicatePairs

This algorithm counts the number of duplicate pairs in a sequence.

Input: $a_1, a_2,...,a_n$, where n is the length of the sequence.
Output: count = the number of duplicate pairs.


```
count := 0

For i = 1 to n-1
    For j = i+1 to n
        If ( a_i = a_j ), count := count +1
    End-for
End-for

Return( count )
```

The outer loop iterates n times on a sequence of length n. The number of iterations of the inner loop depends on the value of i, the index for the outer loop. In the first iteration of the outer loop, i = 1 and the inner loop iterates n - 1 times. In the next iteration of the outer loop, i = 2 and the inner loop iterates n - 2 times. In general, in the i^{th} iteration of the outer loop, the inner loop iterates n - i times. The process continues until i = n - 1 and the inner loop iterates only one time. The total number of iterations of the inner loop is:

$$(n-1) + (n-2) +... + 2 + 1$$

The material on summations gives a method to analyze sums like the one in the expression above. For now, we will just use the fact that:

$$(n-1) + (n-2) + \cdots + 2 + 1 = \frac{n(n-1)}{2} = \Theta(n^2)$$

Note that for the CountDuplicatePairs algorithm, the number of times the loops iterate are the same, regardless of the input. The number of operations inside the inner loop may be 1 or 2, because the variable count is incremented only when $a_i = a_j$. However, the difference between 1 and 2 only affects the constant factor which does not appear in the $\Theta(n^2)$. Therefore, it is sufficient to say that the number of instructions per iteration is some constant c. There are also a constant number of operations for each iteration of the outer loop (incrementing and checking the index i) and a constant number of operations before and after the nested loop (initializing and returning the variable count). These numbers end up as constant factors and can be given arbitrary names such as "c" or "d" because constants factors do not appear in the final asymptotic notation.

PARTICIPATION ACTIVITY 8.3.10: Analysis of the algorithm CountDuplicatePairs.

Animation captions:

1. In the "CountDuplicates" algorithm, the inner for-loop executes c operations per iteration.
2. When the index of the outer for-loop is i, the inner for-loop executes (n-i) times.
3. The number of operations to update the index and check the termination condition for the for-loop is a constant b.
4. The total number of operations in the iteration of the outer for-loop with index i is $c(n - i) + b$. The index i goes from 1 to n-1.
5. d operations are executed after the nested for-loops.
6. The total number of operations executed is $c[(n - 1) + (n - 2) + \cdots + 1] + b \cdot (n - 1) + d$.
7. $c[(n - 1) + (n - 2) + \cdots + 1] + b \cdot (n - 1) + d$ is $\Theta(n^2)$.

The formal proof that the time complexity of CountDuplicatePairs is $\Theta(n^2)$ is given below. The lower bound can be simplified by omitting the constant number of operations before and after the nested loop because the additive constants do not affect the asymptotic bound. In proving a lower bound on the time complexity, it is not necessary to count every instruction because the goal is to show that the number of operations is at least a certain number. In particular with nested loops, the number of times the innermost loop executes often dominates the running time of the algorithm, so it is often sufficient to count instructions executed in the innermost loop. The upper bound can also be simplified by using n (instead of n - i) as an upper bound on the number of iterations of the inner loop. Even with these simplifications, the asymptotic growth rates of the upper and lower bounds match.

Proof 8.3.1: Proof that the growth rate of the worst-case time complexity of CountDuplicatePairs is quadratic.

Proof.

Lower bound: For inputs of size n, the number of times the inner loop iterates with outer index i is (n-i). The outer index ranges from 1 through (n-1). Therefore the total number of times that the inner loop iterates in the entire nested loop is (n-1) + (n-2) + … + 1. The value of the sum is $\Omega(n^2)$, so the worst-case complexity of CountDuplicatePairs is $\Omega(n^2)$.

Upper bound: For any input of size n, the number of times the inner loop iterates with outer index i is (n-i). Since n - i ≤ n, the inner loop iterates at most n times for every iteration of the outer loop. The outer loop iterates at most n times, so the total number of iterations of the inner loop is at most n^2. The total number of operations executed in the entire nested loop is at most cn^2, for some constant c. There are at most d operations performed before and after the nested loop, so the total number of operations performed is at most $cn^2 + d$ which is $O(n^2)$. ■

PARTICIPATION ACTIVITY 8.3.11: Worst-case complexity - searching for a duplicate.

The algorithm below determines whether or not there is a pair of duplicate numbers in a sequence of numbers.

```
FindDuplicate
This algorithm determines if there is a pair of duplicate entries in the sequence.

Input: a1, a2,...,an
       n, the length of the sequence.
Output: "Yes" if there is a duplicate pair and "No" if the elements in the sequence are
distinct.

For i = 1 to n - 1
    For j = i+1 to n
        If ( ai = aj ), Return("Yes")
    End-for
End-for

Return( "No" )
```


1) Which description corresponds to the type of input that will cause FindDuplicate to execute the most number of instructions?

- A sequence of n numbers in which the first and second entries are the same.
- A sequence of n numbers that are all distinct, except the first and last numbers which are equal.
- A sequence of n numbers that are all distinct.

2) Use your answer in the previous question to identify the best possible lower bound for the worst-case asymptotic time complexity of the algorithm.
   - $\Omega(1)$
   - $\Omega(n)$
   - $\Omega(n^2)$

3) What is the best upper bound for the asymptotic time complexity of the algorithm?
   - $O(n)$
   - $O(n^2)$

Additional information: 8.3.1: Efficient algorithms.

In practice, the criteria for an algorithm to be "efficient" may depend greatly on the context. An algorithm that processes terabytes of data may be completely impractical unless its time complexity is at most linear.

An algorithm is said to run in ***polynomial time*** if its time complexity is $O(n^k)$ for some fixed constant k. As a shorthand, an algorithm is called "efficient" if the algorithm runs in polynomial time. For example, an algorithm whose time complexity is $O(n^5)$ would be considered efficient. However, an algorithm whose time complexity is $\Omega(n^{\log(n)})$ or $\Omega(2^n)$ would be considered inefficient. In reality, an algorithm that runs in time $\Omega(n^{100})$ would only be practical for very small input sizes. However, the distinction between algorithms that run in polynomial time and those that do not is a useful starting point to understanding the algorithm's efficiency in practice.

CHALLENGE ACTIVITY | 8.3.1: Code complexity.

455912.3056722.qx3zqy7

Start

The below code has a worst-case complexity of

Input: $a_1, a_2, a_3, \ldots, a_n$
n, the length of the sequence.

```
For counter = 1 to n
    Output: counter
End-for
```

1 2 3

Check Next

Additional exercises

EXERCISE | 8.3.1: Worst-case time complexity - counting numbers in a sequence less than a target value.

```
CountValuesLessThanT

Input: a1, a2,...,an
       n, the length of the sequence.
       T, a target value.
Output: The number of values in the sequence that are less than T.

count := 0

For i = 1 to n
    If ( ai < T ), count := count + 1
End-for

Return( count )
```


(a) Characterize the asymptotic growth of the worst-case time complexity of the algorithm. Justify your answer.

EXERCISE 8.3.2: Worst-case time complexity - maximum subsequence sum.

The input is a sequence of numbers a_1, a_2,..., a_n. A subsequence is a sequence of numbers found consecutively within a_1, a_2,..., a_n. For example, in the sequence: -3, -1, 17, 5, 66, 22, -5, 42, both 17, 5, 66 and -1, 17 are subsequences. However, 66, -5 is not a subsequence because 66 and -5 do not appear consecutively in the sequence. A subsequence can contain only one number such as 66. It can also be empty.

In the maximum subsequence sum problem, the input is a sequence of numbers and the output is the maximum number that can be obtained by summing the numbers in a subsequence of the input sequence. If the input was the sequence -3, -1, 17, 5, 66, 22, -5, 42, then the output would be 147 because the sum of subsequence 17, 5, 66, -5, 42 is 147. Any other subsequence will sum to an equal or smaller number. The empty subsequence sums to 0, so the maximum subsequence sum will always be at least 0.

The algorithm below computes the maximum subsequence sum of an input sequence.

```
MaximumSubsequenceSum

Input: a1, a2,...,an
       n, the length of the sequence.
Output: The value of the maximum subsequence sum.

maxSum := 0

For i = 1 to n
    thisSum := 0
    For j = i to n
        thisSum := thisSum + aj
        If ( thisSum > maxSum ), maxSum := thisSum
    End-for
End-for

Return( maxSum )
```


(a) Characterize the asymptotic growth of the worst-case time complexity of the algorithm. Justify your answer.

(b) Can you find an algorithm that solves the same problem whose worst-case time complexity is linear?

EXERCISE 8.3.3: Worst-case time complexity - finding the maximum value of a function.

The function M takes three input values that are positive integers and outputs a positive integer. We don't know much about the function M but we are given some instructions to compute M. The line M(x, y, z) in the algorithm below computes the value of M on input values x, y, and z. You can assume that it takes O(1) operations to compute the value of M on any input. The input to the algorithm is a sequence of n numbers. We would like to find the maximum value of the function M where the three input values are numbers from the sequence. Numbers can be repeated, so M(a_1, a_1, a_1) is a possibility.

```
FindMaxFunctionValue

Input: a1, a2,...,an
       n, the length of the sequence.
Output: The largest values of M on input values from the sequence.

max := M(a1, a1, a1)

For i = 1 to n
    For j = 1 to n
        For k = 1 to n
            new := M(ai, aj, ak)
            If ( new > max ), max := new
        End-for
    End-for
End-for

Return( max )
```


(a) Characterize the asymptotic growth of the worst-case time complexity of the algorithm. Justify your answer.

EXERCISE 8.3.4: Worst-case time complexity - finding a target product of two numbers.

```
FindProduct

Input: a1, a2,...,an
       n, the length of the sequence.
       prod, a target product.
Output: "Yes" if there are two numbers in the sequence whose product equals the input
"prod". Otherwise, "No".

For i = 1 to n-1
    For j = i+1 to n
        If (ai·aj = prod), Return( "Yes" )
    End-for
End-for

Return( "No" )
```

(a) Characterize the input that will cause the "If" statement in the inner loop to execute the most number of times.

(b) Give an asymptotic lower bound for the running time of the algorithm based on your answer to the previous question.

(c) Was it important to use the worst-case input for the lower bound? That is, would any input of a given length have resulted in the same asymptotic lower bound?

(d) Give an upper bound (using O-notation) for the time complexity of the algorithm that matches your asymptotic lower bound for the algorithm.

EXERCISE 8.3.5: Worst-case time complexity - mystery algorithm.

The algorithm below makes some changes to an input sequence of numbers.

```
MysteryAlgorithm

Input: a_1, a_2,...,a_n
       n, the length of the sequence.
       p, a number.
Output: ??

i := 1
j := n

While (i < j)
    While (i < j and a_i < p)
        i := i + 1
    End-while
    While (i < j and a_j ≥ p)
        j := j - 1
    End-while
    If (i < j), swap a_i and a_j
End-while

Return( a_1, a_2,...,a_n )
```


(a) Describe in English how the sequence of numbers is changed by the algorithm. (Hint: try the algorithm out on a small list of positive and negative numbers with p = 0)

(b) What is the total number of times that the lines "i := i + 1" or "j := j - 1" are executed on a sequence of length n? Does your answer depend on the actual values of the numbers in

the sequence or just the length of the sequence? If so, describe the inputs that maximize and minimize the number of times the two lines are executed.

(c) What is the total number of times that the swap operation is executed? Does your answer depend on the actual values of the numbers in the sequence or just the length of the sequence? If so, describe the inputs that maximize and minimize the number of times the swap is executed.

(d) Give an asymptotic lower bound for the time complexity of the algorithm. Is it important to consider the worst-case input in determining an asymptotic lower bound (using Ω) on the time complexity of the algorithm? (Hint: argue that the number of swaps is at most the number of times that i is incremented or j is decremented).

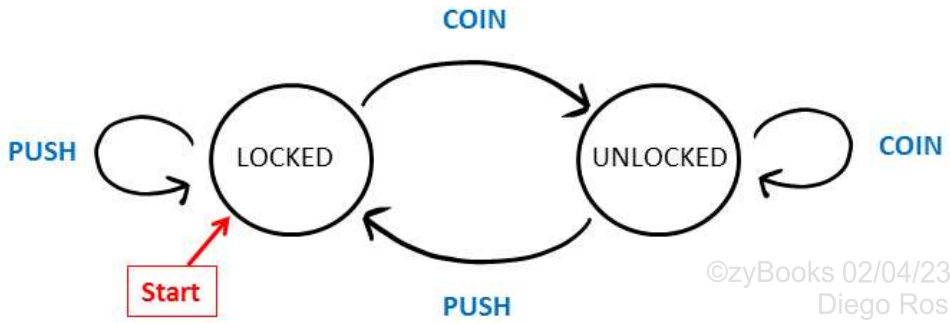
(e) Give a matching upper bound (using O-notation) for the time complexity of the algorithm.

8.4 Finite state machines

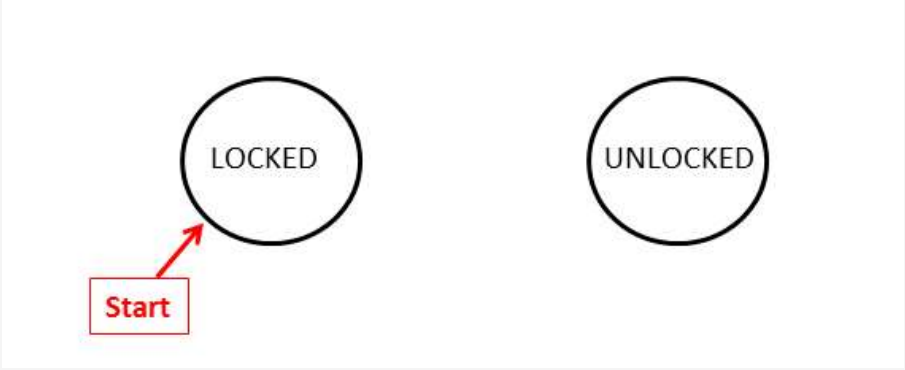
This section has been set as optional by your instructor.

Computer scientists develop abstract models for computing devices in order to reason about what can and cannot be computed by a particular kind of device. One such model is a finite state machine. A ***finite state machine*** (or ***FSM***) consists of a finite set of states with transitions between the states triggered by different input actions. Finite state machines are used in various computer science applications. Most of this section's examples show how FSMs can model logical devices. FSMs are also used in specifying network and communication protocols, compiler design, and text search. A finite state machine is sometimes called a ***finite state automaton*** (plural: ***finite state automata***).

We illustrate the components of an FSM using a small example before giving a formal definition. The diagram below shows an FSM that controls a turnstile. The turnstile starts out in a locked state and unlocks when a person inserts a coin. Additional coins inserted when the turnstile is unlocked do not change the state of the system. If the turnstile is unlocked and the person pushes it, then the turnstile transitions to the locked state, awaiting a coin from the next person. Pushing the turnstile while it is locked does not affect the system state.



Now we will introduce and define each component of the FSM. A finite state machine has, as the name suggests, a finite set of states. The FSM always resides in one of the states. Furthermore, the current state is the only information that the FSM remembers about the past. To help a human understand what the states represent, the states are given descriptive names. The first example presented here is a turnstile which is either locked or unlocked. So the set of states Q would be {LOCKED, UNLOCKED}. *Although the states can be given any two distinct labels, it is good practice to give states descriptive names.* The FSM also has a designated start state which is the initial state of the system before any input is received. In the case of the turnstile, the start state is the LOCKED state. In the diagram, there is a vertex (circle) labeled with each state. The start state is designated with an arrow:



The FSM receives input from the outside in the form of a sequence of actions. Each individual action is from a finite set of inputs. In the case of the turnstile, a person can insert a coin into the turnstile or push the turnstile to pass through it. The input set I = {COIN, PUSH}. The input to the FSM is a sequence of actions from the input set. In the case of a turnstile, the input might be: COIN, COIN, PUSH, COIN, PUSH.

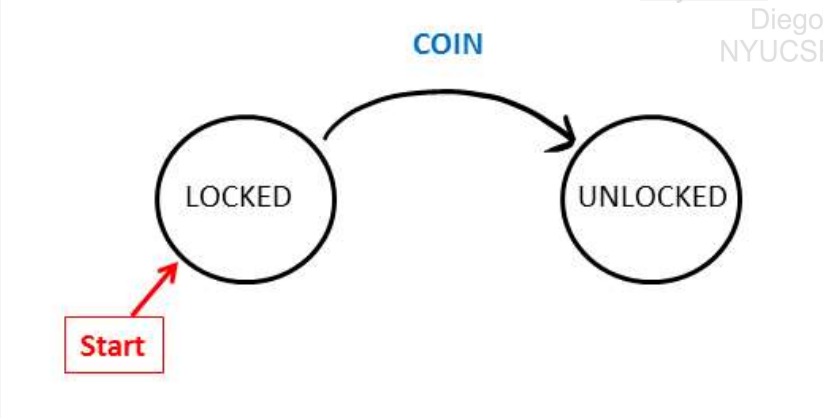
The FSM then reacts to the input. The reaction depends on the current state and the input action. We present a variety of different types of FSMs that have different mechanisms for reacting to the input. In the case of the turnstile, the reaction is encoded in the state: the turnstile is either LOCKED (so that the person is prevented from passing through) or UNLOCKED (so that the person is allowed to pass through).

The reaction of a finite state machine to the input received is denoted by a ***transition function***. The input to the transition function is the current state and the input action, so the domain is the Cartesian product set Q × I. The output of the transition function for this particular style of FSM is just the next state to which the FSM transitions. Therefore the target set of the transition function is the set Q. The

image below shows that the FSM upon receiving input COIN while in the LOCKED state transitions to the UNLOCKED state. Thus, the output of the transition function on input (LOCKED, COIN) is UNLOCKED. The transition function is often denoted by the symbol δ.

$$\delta(\text{LOCKED}, \text{COIN}) = \text{UNLOCKED}$$

The transition is depicted in the diagram by an arrow from LOCKED to UNLOCKED and is labeled with the input COIN.



The transition function can also be specified by a state transition table. The rows represent the current state, the columns represent the possible inputs. The entry for a particular row and column indicate the new state resulting from that state/input combination. Below is the state transition table for the turnstile example:

Table 8.4.1: State transition table for the turnstile finite state machine.

-	Coin	Push
Locked	Unlocked	Locked
Unlocked	Unlocked	Locked

The table below summarizes the components of a finite state machine:

Table 8.4.2: Components of a FSM.

Notation	Description
Q	Finite set of states.
$q_0 \in Q$	q_0 is the start state.

I	Finite set of input actions.
$\delta: Q \times I \rightarrow Q$	Transition function.

The animation below shows how the turnstile FSM behaves on a particular sequence of input actions:

PARTICIPATION ACTIVITY 8.4.1: The actions of the turnstile FSM.

Animation content:

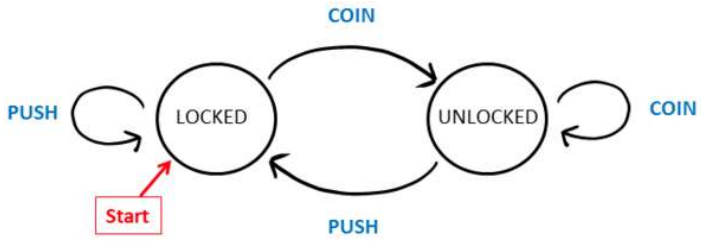
undefined

Animation captions:

1. The start state is "LOCKED". On input "COIN", the current state transitions to "UNLOCKED".
2. The next input is "COIN" and the current state remains "UNLOCKED".
3. The next input is "PUSH" and the current state transitions to "LOCKED". The last input is "PUSH" and the current state remains "LOCKED", the final state.

PARTICIPATION ACTIVITY 8.4.2: Input and output of the transition function for the turnstile FSM.

Consider the turnstile FSM below:



1) Which pair would be a valid input to the transition function δ?

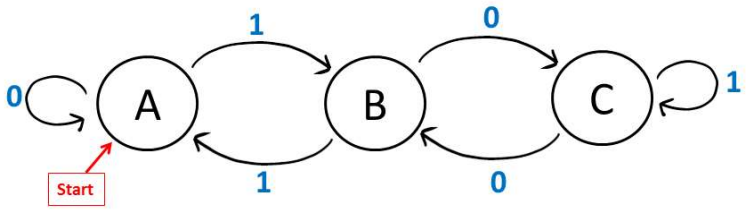
- (COIN, PUSH)
- (LOCKED, PUSH)
- (LOCKED, UNLOCKED)

2) What is the value of δ(UNLOCKED, COIN)?

- UNLOCKED
- LOCKED
- PUSH

PARTICIPATION ACTIVITY 8.4.3: Following the transitions of a finite state machine.

Consider the FSM defined in the state diagram below. The set of states is {A, B, C} and the set of input actions are labeled 0 or 1.



1) What is the current state after the FSM has processed the input sequence 00101?

Check Show answer

2) What is the current state after the FSM has processed the input sequence 111010?

Check Show answer

Finite state machines with output

In the turnstile example, the response of the FSM was entirely expressed in the state, i.e., whether the turnstile is locked or unlocked. In some applications, a more detailed response is required. A ***finite state machine with output***, or ***FSM with output***, produces a response that depends on the current state as well as the most recently received input. We illustrate an FSM with output using an example of a gumball machine. The machine sells gumballs for 20 cents and accepts only nickels and dimes. To keep the example simple, the machine will not make change. If the user has already inserted 15 cents and then inserts a dime, the machine just returns the dime and remains in the same state. Five states represent the amount of money that has been inserted so far:

$Q = \{ q_0, q_5, q_{10}, q_{15}, q_{20} \}$

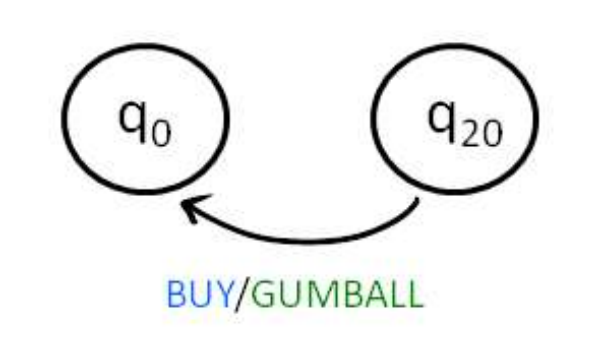
The user can insert a nickel, insert a dime, or press a button to buy a gumball:

$I = \{ \text{NICKEL, DIME, BUY} \}$

The FSM has different ways of responding to the user's actions. The gumball machine can release a gumball, can return the most recently inserted coin, or can issue a message that there is not enough money to buy a gumball. The gumball may also not have an output in some situations. The responses of the gumball machine is a finite set called the output set and is denoted by O:

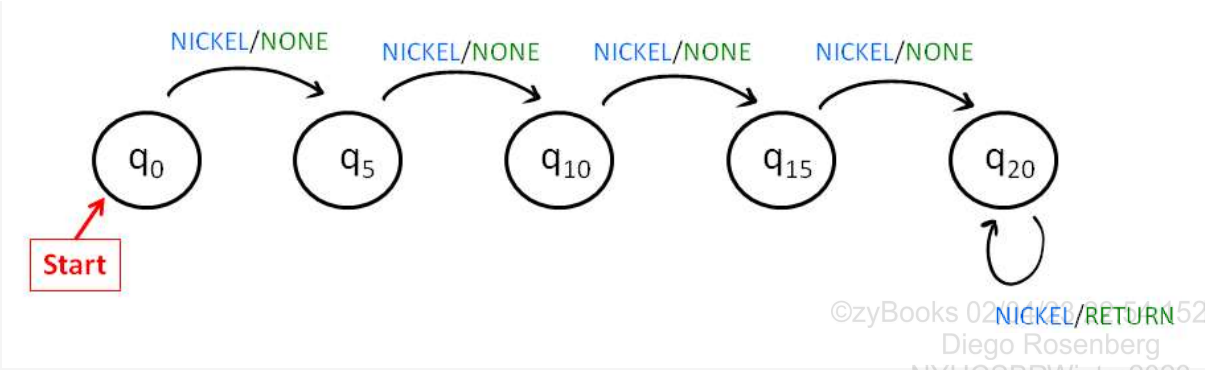
$O = \{ \text{GUMBALL, RETURN, MESSAGE, NONE} \}$

The next part is to determine the transition function. The input to the transition function is a state and an input action. The output of the transition function is the new state to which the FSM transitions and a response from the set O. Therefore δ is a function whose domain is $Q \times I$ and whose target is $Q \times O$. We specify the transition function by a state diagram which is a directed graph whose nodes represent states. As in the example of the turnstile, a directed edges represent transitions from one state to another. However, since the gumball machine example is an FSM with output, each edge is labeled with an input action from I and an output response from O. For example, a directed edge from state q_{20} to state q_0 labeled BUY/GUMBALL indicates that if the FSM is in state q_{20} and receives input BUY, then the FSM will transition to state q_0 and release a gumball:

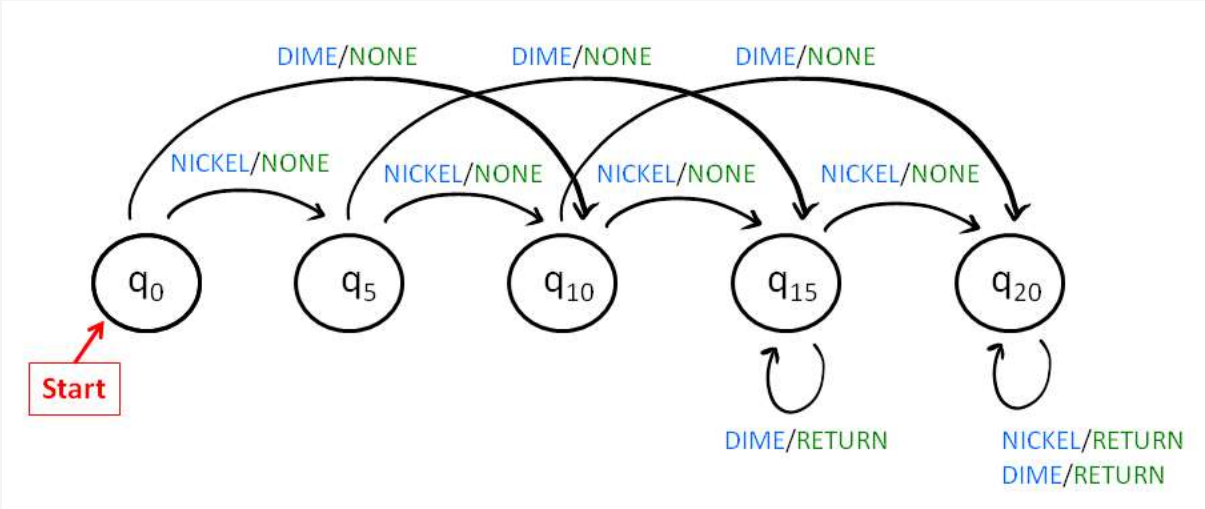


The directed edge corresponds to the fact that $\delta(q_{20}, \text{BUY}) = (q_0, \text{GUMBALL})$.

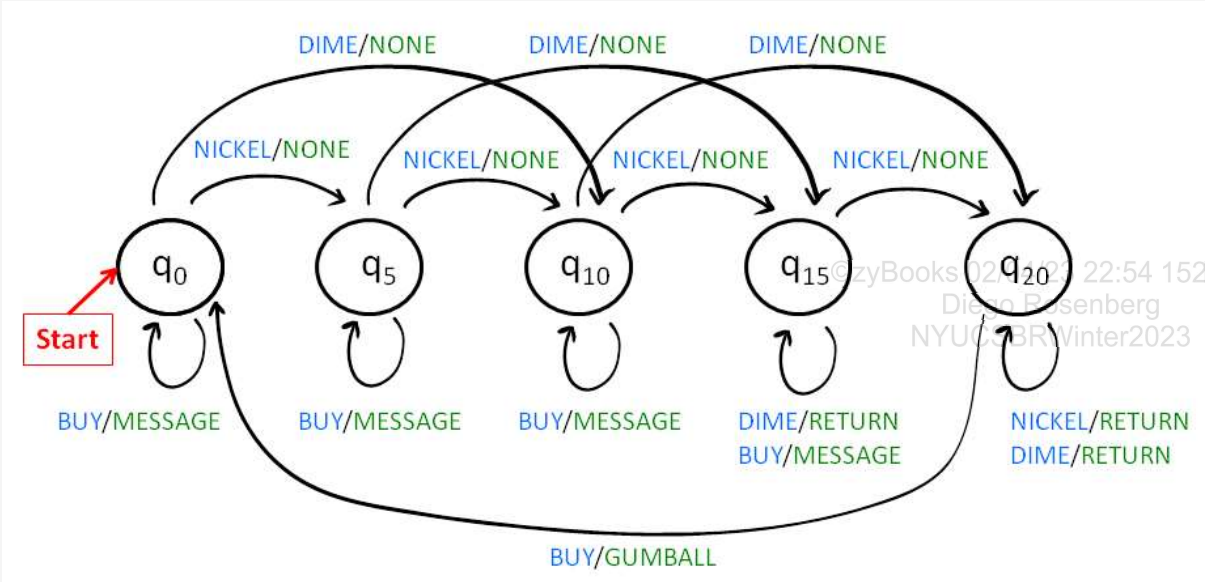
We construct the transition function by adding in edges to the state diagram in phases in order to illustrate how one might design an FSM to perform a specific function. First we show how the gumball machine responds when the user inputs a nickel. If the user has input less than 20 cents, the FSM advances to the next state (reflecting five more cents have been inserted) and produces no output. If the FSM is in state q_{20} and the user inputs a nickel, the coin is returned and the FSM stays in state q_{20}. The transitions are reflected in the diagram below:



Now we add edges to the state diagram reflecting how the gumball machine responds when the user inputs a dime. If the user has input 10 cents or less (q_0, q_5, or q_{10}), the FSM transitions to a state reflecting 10 more cents have been inserted and produces no output. If the user has inserted more than ten cents (q_{15} or q_{20}), the FSM stays in the same state and returns the coin.



Now we add the edges to the state diagram reflecting how the gumball machine responds when the user presses the "buy" button. If the user has input less than 20 cents, it stays in the same state and outputs the message that the user has not input enough money. However, if the FSM is in state q_{20}, the user has input enough money for a gumball. The gumball machine releases a gumball and returns to the q_0 state:



The state diagram is complete: every state has an outgoing edge for every possible input action. Note that the only memory the FSM has is encoded in its current state. If the FSM is in state q_{10}, it knows that 10 cents have been inserted so far, but it does not know whether the user put in one dime or two nickels. Fortunately, the only information an FSM needs to know to operate correctly is the amount of money they user has inserted. The table below summarizes the components of an FSM with output. The new components that are not included in an FSM without output are highlighted in red:

Table 8.4.3: Components of a finite state machine with output.

Notation	Description
Q	Finite set of states.
$q_0 \in Q$	q_0 is the start state.
I	Finite set of input actions.
O	Finite set of output responses.
δ: $Q \times I \rightarrow Q \times O$	Transition function.

The animation below shows how the gumball machine FSM behaves on a particular sequence of input actions:

PARTICIPATION ACTIVITY 8.4.4: The actions of the gumball machine FSM.

Animation content:

undefined

Animation captions:

1. The start state is q_0. On input "BUY", the current state remains q_0 and the output is "MESSAGE".
2. The next input is "NICKEL", the current state transitions to q_5 and the output is "NONE".
3. The next input is "DIME", the current state transitions to q_{15} and the output is "NONE".
4. The next input is "DIME", the current state remains q_{15} and the output is "RETURN".
5. The next input is "NICKEL", the current state transitions to q_{20} and the output is "NONE".
6. The final input is "BUY", the current state transitions back to q_0 and the output is "GUMBAL

PARTICIPATION ACTIVITY 8.4.5: The gumball machine FSM.

1) What is the last output response after the gumball machine processes the sequence: DIME, DIME, NICKEL, DIME?

Check Show answer

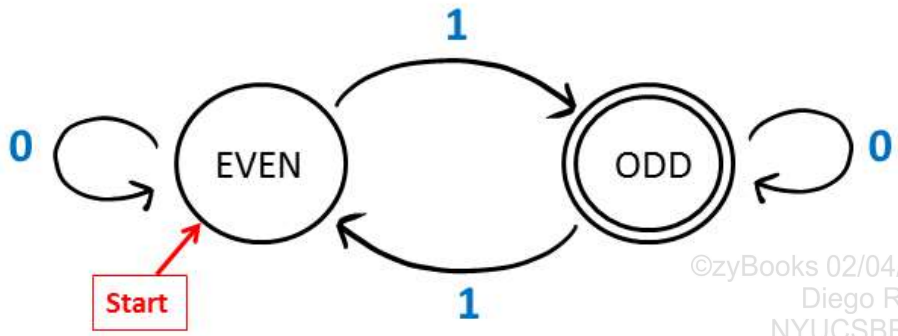
2) How many gumballs are purchased after the following input sequence: NICKEL, DIME, NICKEL, BUY, DIME, NICKEL, BUY? Write your answer in numerical form.

Check Show answer

Finite state machines that recognize properties

Finite state machines can also be used to determine whether an input string has a particular property. Instead of having input actions from a user, each input is a character from a finite alphabet. The sequence of inputs is a string over the input alphabet. An input string is accepted if the FSM ends up in an ***accepting state*** after each character in the string is processed. The set of all accepting states is a designated subset of the states.

Consider a finite state machine with input alphabet {0, 1}. There are two states: {ODD, EVEN}. The state EVEN is the start state and the state ODD is the only accepting state. The transition function for the FSM is specified by the state diagram below. As before, the start state is denoted by an arrow. The accepting state is shown with a double outline.



The ***parity of a string*** is whether the number of bits in the string is odd or even. The string 01011 has odd parity because 01011 has three 1's and three is an odd number. The string 11011 has even parity because 11011 has four 1's and four is an even number. The finite state machine shown above accepts a binary string if and only if the string has odd parity. The state encodes whether the number of 1's seen so far is odd or even. An input of 0 causes the FSM to stay in the same state. An input of 1 causes the FSM to change states.

The table below summarizes the components of an FSM that recognizes a property. The new components that are not included in an FSM without output are highlighted in red:

Table 8.4.4: Components of a finite state machine that recognizes a property.

Notation	Description
Q	Finite set of states.
$q_0 \in Q$	q_0 is the start state.
I	Finite set of input actions.
$A \subseteq Q$	Accepting states are a subset of the states.
$\delta: Q \times I \rightarrow Q$	Transition function.

Here is an animation of the action of the FSM on a particular input:

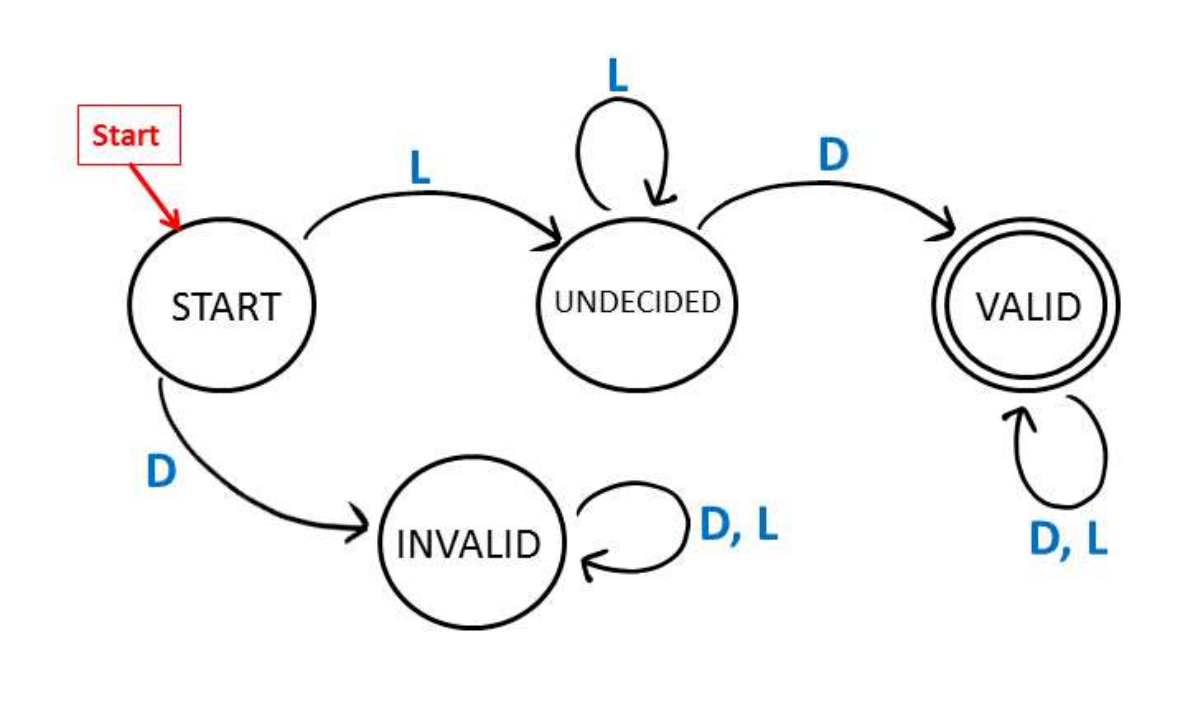
PARTICIPATION ACTIVITY 8.4.6: The actions of the FSM that computes the parity of a string.

Animation captions:

1. The start state is "EVEN". On input 1, the current state transitions to "ODD".
2. The rest of the inputs are 1, 0, 1. The state transitions to "EVEN", "EVEN", then "ODD". Since the final state "ODD" is accepting, the input 1101 is accepted.
3. On inputs 101, the sequence of states is "EVEN", "ODD", "ODD", "EVEN". Since the final state "EVEN" is not accepting, the input 101 is not accepted.

Recognizing valid passwords

In another example, consider a computer system that accepts a password composed of digits and letters of any length. A valid password must begin with a letter and have at least one digit. We will show an FSM that accepts an input string if and only if it is a valid password. The input alphabet is {D, L} which stand for "digit" and "letter". There are four states: {START, INVALID, VALID, UNDECIDED}. The only accepting state is the VALID state. The state diagram for the FSM is given below:



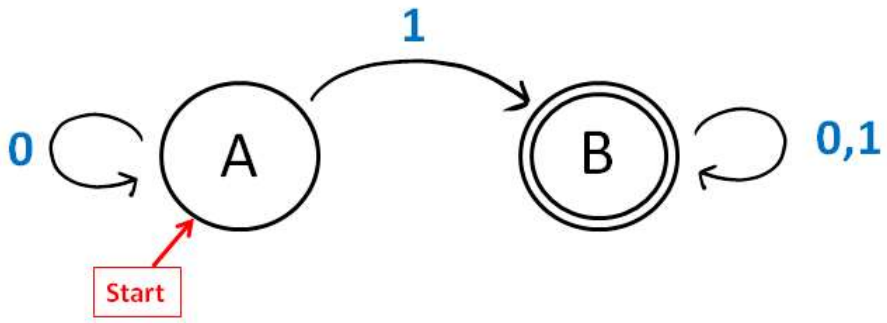
Note that if the first input character is a digit, the FSM will transition to the INVALID state because the first character is required to be a letter. If the FSM reaches the invalid state, the FSM will not accept the input regardless of what follows. On the other hand, if the first input character is a letter, the FSM will reach the UNDECIDED state. In the UNDECIDED state, the FSM is waiting to see if the input string contains a digit. If the FSM is in the UNDECIDED state and a digit appears, then all the conditions for the password have been met and the FSM will transition to the VALID state and will accept the input string regardless of what other characters follow. On the other hand, if the string ends while the FSM is in the UNDECIDED state (i.e., without having seen a digit in the input string), the FSM will not accept.

Additional Information 8.4.1: Limitations of FSMs.

While finite state machines can be used to recognize many useful properties, there are limits to the kinds of properties that can be recognized by an FSM. For example, it is impossible to design an FSM that takes as input any binary string and accepts if and only if the majority of the bits are 1. The proof that FSMs can not recognize certain properties is beyond the scope of this material, but it is not hard to see why FSMs are so limited. Essentially, the only memory that an FSM has is encoded in the current state. Since the set of states are finite, an FSM inherently has a finite memory. In order to compute whether there is a majority of 1's in a string the machine would need to keep track of how many more 1's than 0's have been observed in the input string up to a given point. Since the set of all binary strings is infinite, there is no bound to the number that a device computing majority would need to hold.

PARTICIPATION ACTIVITY 8.4.7: Finite state machines as recognizers.

The input alphabet for the FSM below is {0, 1}. The state set is {A, B}

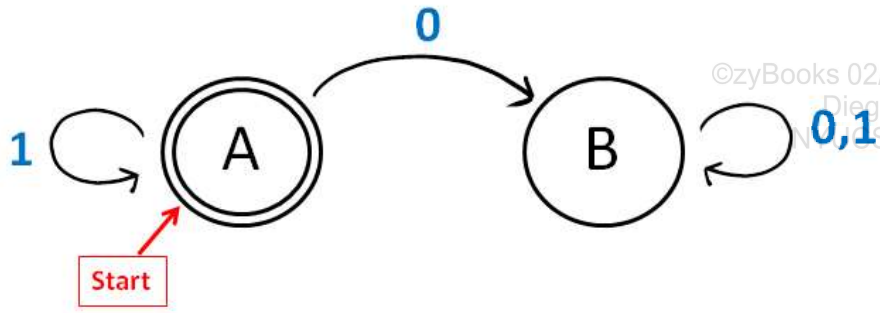


1) Which sentence describes the set of strings accepted by the FSM?

- The FSM accepts a binary string if and only if the majority of bits are 1's.
- The FSM accepts a binary string if and only if the parity of the string is odd.
- The FSM accepts a binary string if and only if all the input bits are 1.
- The FSM accepts a binary string if and only if there is at least one 1 in the input sequence.

PARTICIPATION ACTIVITY 8.4.8: Finite state machines as recognizers.

Consider the finite state machine denoted below. The input alphabet for each is {0, 1}. The state set for each is {A, B}

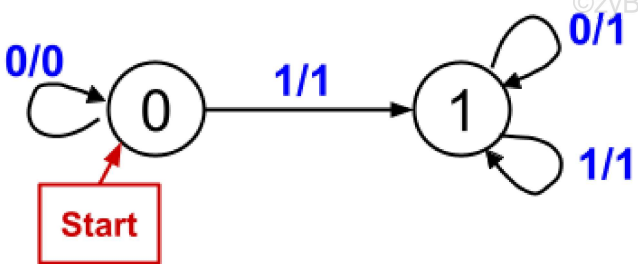


1) Which sentence describes the set of strings accepted by the FSM?

- The FSM accepts a binary string if and only if the majority of bits are 1's.
- The FSM accepts a binary string if and only if the parity of the string is odd.
- The FSM accepts a binary string if and only if all the input bits are 1.
- The FSM accepts a binary string if and only if there is at least one 1 in the input sequence.

Additional exercises

EXERCISE 8.4.1: Output for an FSM on a given input - example 1.



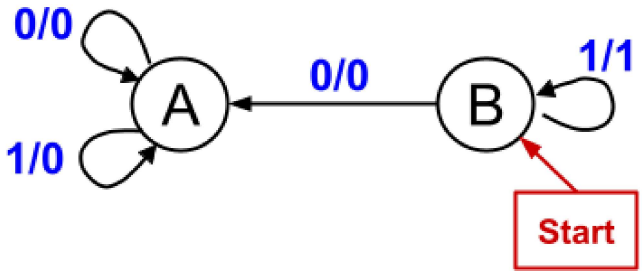
The input and output alphabet for the FSM shown above is {0, 1}. Give the output for the FSM for each input string.

(a) 000

(b) 010

(c) 111

EXERCISE 8.4.2: Output for an FSM on a given input - example 2.



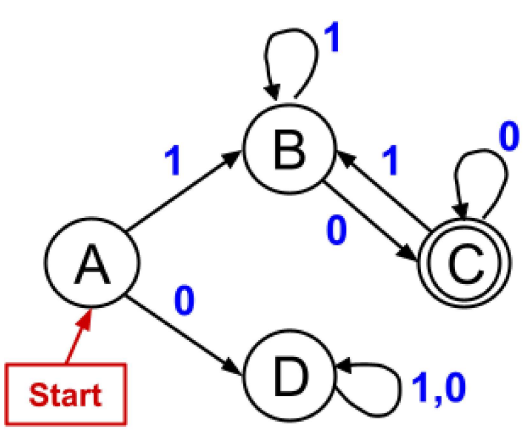
The input and output alphabet for the FSM shown above is {0, 1}. Give the output for the FSM for each input string.

(a) 000

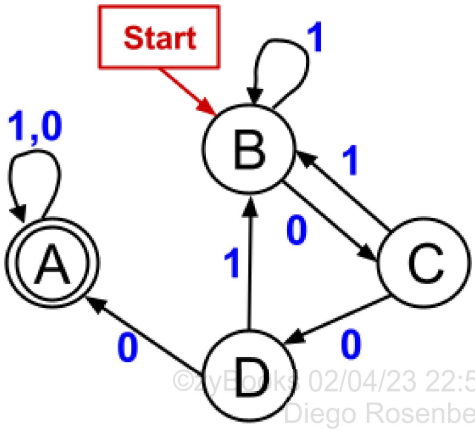
(b) 010

(c) 101

EXERCISE 8.4.3: Identifying the set of strings accepted by an FSM.



FSM 1



FSM 2

(a) Give the final state of each FSM after each input string below. That is, start in the start state, process the string, and indicate which state (A, B, C, or D) the FSM is in at the end of the string. Then indicate whether or not the final state is an accepting state.

- 100011

- 0000
- 0010
- 1100

(b) Which one of the following statements below describes the set of strings accepted by FSM 1?

1. The FSM accepts a string x if and only if x contains the pattern "01" somewhere in the string.
2. The FSM accepts a string x if and only if x starts with a 1 and ends with a 0.
3. The FSM accepts a string x if and only if x is all 0's or all 1's (i.e. x only contains one type of character).
4. The FSM accepts a string x if and only if there are at least three consecutive 0's somewhere in x.
5. The FSM accepts a string x if and only if x ends with "000".
6. The FSM accepts a string x if and only if x ends with "00".

(c) Refer to the same set of statements in the previous question and indicate which statement describes the set of strings accepted by FSM 2.

EXERCISE 8.4.4: Designing FSMs that accept a given set of strings.

For each property, design an FSM with input alphabet {0, 1} that accepts a string x if and only if the string has the property described.

(a) There are no occurrences of "00" or "11" in the string. (The empty string has no occurrences of "00" or "11".)

(b) The number of 1's is a multiple of 3. (Zero is a multiple of 3).

(c) There is at least one 0 and at least one 1.

8.5 Turing machines

This section has been set as optional by your instructor.

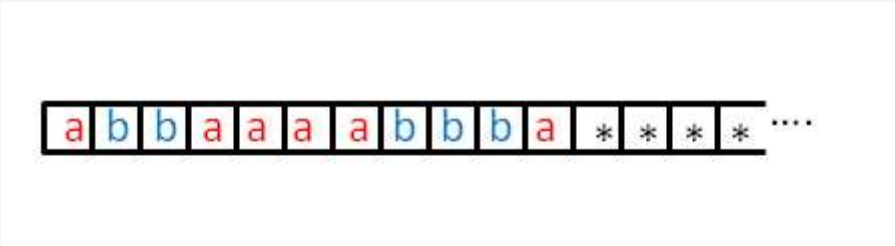
Finite state machines are effective in some settings, such as modeling devices with limited input and output capabilities. However, finite state machines are unable to solve even simple computational tasks such as determining whether a binary string has more 0's than 1's.

Mathematician Alan Turing developed a model of a computer, called a ***Turing machine***, in order to reason about general purpose computers. Turing machines are much simpler than modern processors, but Turing machines are believed to be every bit as powerful. The ***Church-Turing conjecture*** says that any problem that can be solved efficiently on any computing device can be solved efficiently by a Turing machine.

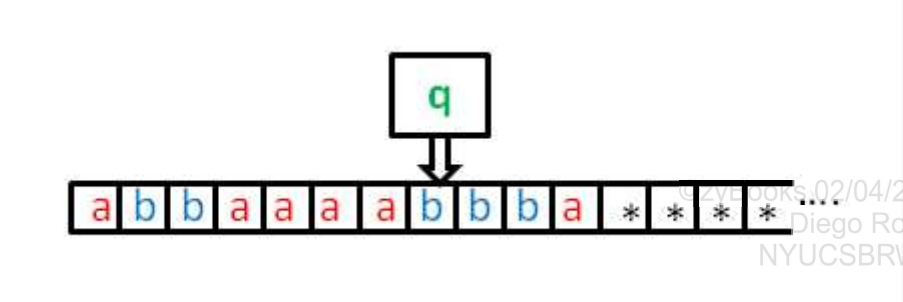
There is often a tradeoff between the complexity of a device and how easy that device is to program. A modern computer with its layers of hardware, operating system and compilers, provides an intuitive user interface so that people can easily write programs to solve complex tasks. Turing machines, on the other hand, are simple to describe in the abstract but are very cumbersome to use in solving specific computational problems. Nonetheless, the Turing machine is a useful abstraction for reasoning about the nature and limits of computation.

The definition of a Turing machine

The memory of a Turing machines is a one-dimensional tape. The tape is divided into individual cells, each of which can hold one symbol from a finite ***tape alphabet*** denoted by Γ. The tape is infinite to the right but has a leftmost cell. The tape alphabet must contain a special symbol called the blank symbol (represented by a * symbol) and at least one other symbol. Here is what the tape of a Turing machine might look like for tape alphabet {a, b, *}:



The Turing machine has a finite set of states, denoted by Q. The set Q includes three special states: q_0 is the start state, q_{acc} is the accept state, and q_{rej} is the reject state. The Turing machine also has a head that always points to a cell of the tape. The symbol in the cell to which the head is currently pointing is called the current symbol. A ***configuration*** of a Turing machine consists of the contents of the tape, the current state, and the tape cell to which the head is currently pointing. Here is a drawing of a configuration of a Turing machine that is in state q and whose current symbol is b:



The action of the Turing machine is defined by a transition function δ. The input to the transition function is the current state and the current symbol. The output of the transition function is a state, a tape symbol and a direction for the tape head to move. The direction is an element from the set {L, R}. L denotes a move to the left and R denotes a move to the right. The Turing machine proceeds in a series of steps. Each step is dictated by the transition function. Suppose, for example, that the state of

the Turing machine is as pictured above and δ(q, b) = (q', a, L). Then the Turing machine will write an 'a' in the current cell, transition to state q' and the head will move one cell to the left. If the head is supposed to move left and is already in the leftmost tape cell, then the head stays in the same location after the step. Here is an animation showing several steps of the Turing machine:

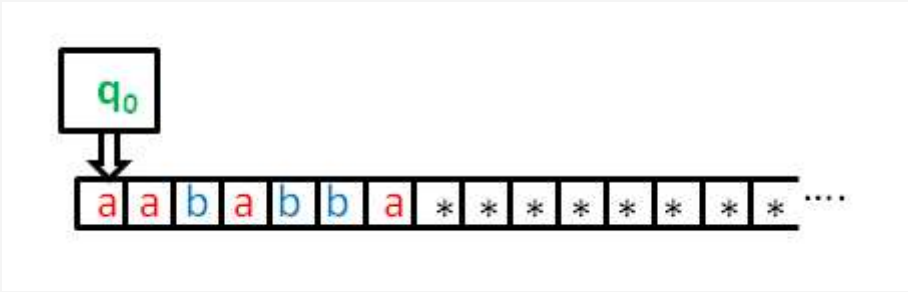
PARTICIPATION ACTIVITY 8.5.1: Steps of a Turing machine.

Animation captions:

1. The state is q_1 and the current symbol is a. δ (q_1, a) = (q_2, *, L). The state becomes q_2, the symbol a is replaced by *, and the head moves left.
2. The state is q_2 and the current symbol is b. δ (q_2, b) = (q_1, b, R). The state becomes q_1, the symbol b is unchanged, and the head moves right.
3. The state is q_1 and the current symbol is *. δ (q_1, *) = (q_3, b, R). The state becomes q_3, the symbol * is replaced by b, and the head moves right.

The transition function is defined for every (state, tape symbol) pair, except for the states q_{acc} and q_{rej}. If the Turing machine transitions to q_{acc} or q_{rej} during a step, then the Turing machine completes the step and then stops.

The input to the Turing machine is specified as a string over the ***input alphabet***. The input alphabet, denoted by Σ, must be a subset of the tape alphabet (Σ ⊂ Γ) and can not contain the blank symbol (* ∉ Σ). A Turing machine starts out an execution in state q_0 with the head in the leftmost position. The input to the problem to be solved is written on the tape as a string of input symbols starting in the leftmost cell. The rest of the tape is all blank symbols that extend infinitely to the right. Here is a drawing of the initial configuration of a Turing machine on input "aababba":



The table below lists all the components in the definition of a Turing machine:

Table 8.5.1: Components of a Turing machine.

Notation	Description
Q	Finite set of states.

Γ	Finite set of tape symbols.
$\Sigma \subset \Gamma$	A subset of the tape symbols are input symbols.
$q_0 \in Q$	q_0 is the start state.
$q_{acc} \in Q$	q_{acc} is the accept state.
$q_{rej} \in Q$	q_{rej} is the reject state.
$\delta: (Q - \{q_{acc}, q_{rej}\}) \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}$	Transition function.

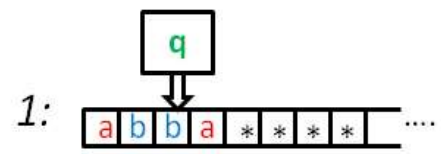
The Turing machine proceeds according to the rules designated by the transition function. If the Turing machine reaches the accept state after starting with a particular input string x, then the ***Turing machine accepts*** x. If the Turing machine reaches the reject state after starting with a particular input string x, then the ***Turing machine rejects*** x. If a Turing machine reaches either the accept or reject state on input x, we say that the ***Turing machine halts*** on input x. As with other computer programs, the Turing machine may never halt on a particular input in which case the Turing machine neither accepts or rejects the input.

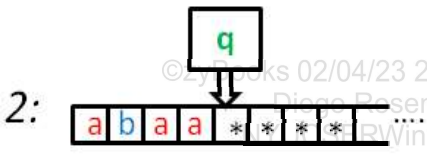
PARTICIPATION ACTIVITY 8.5.2: Identifying the next configuration of a Turing machine.

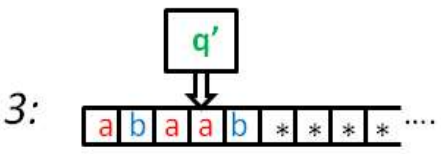
Consider a Turing machine whose transition function is partially specified below:

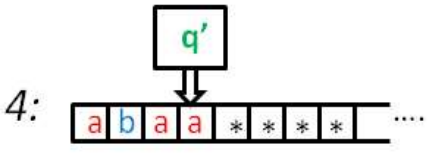
$\delta(q, b) = (q', a, R)$
$\delta(q', a) = (q, a, R)$
$\delta(q, *) = (q', b, L)$

Below are four Turing machine configurations:









1) The current configuration of a Turing machine is shown in drawing 1. Which drawing shows the configuration of the Turing machine after one step? Enter your answer as a number from 1 to 4.

Check Show answer

2) The current configuration of a Turing machine is shown in drawing 1. Which drawing shows the configuration of the Turing machine after two steps?

Check Show answer

3) The current configuration of a Turing machine is shown in drawing 1. Which drawing shows the configuration of the Turing machine after three steps?

Check Show answer

A simple Turing machine

The first example of a Turing machine takes as input any string over the alphabet {a, b} and accepts the string if and only if the string has at least two b's anywhere in the string. The two b's do not have to be consecutive, so the Turing machine should accept the string baba, for example. The input alphabet is {a, b} and the tape alphabet is {a, b, *}. The set of state is $\{q_0, q_1, q_{acc}, q_{rej}\}$. The head starts in the left-most cell and moves from left to right. In each move the Turing machine writes the same symbol that was read, so the tape never changes content. The number of b's encountered so far is encoded in the state. q_0 represents the condition that no b's have been seen. q_1 represents the condition that one b has been seen. If the Turing machine is in state q_0 and the tape symbol b is encountered, the Turing machine transitions from q_0 to q_1 and continues moving to the right. If the Turing machine is in state q_1 and the tape symbol b is encountered, the Turing machine transitions from q_1 to q_{acc}, reflecting the

fact that two b's have been seen. If the Turing machine encounters a * symbol in state q_0 or q_1, then the Turing machine has reached the end of the input string without having seen two b's and therefore transitions to q_{rej}.

The transition function is given in the table below. The animation illustrates the execution of the Turing machine on two different input strings.

Table 8.5.2: Transition function for the Turing machine that recognizes strings with two b's.

-	a	b	*
q_0	(q_0, a, R)	(q_1, b, R)	(q_{rej}, *, L)
q_1	(q_1, a, R)	(q_{acc}, b, R)	(q_{rej}, *, L)

PARTICIPATION ACTIVITY 8.5.3: A Turing machine that accepts strings with two b's.

Animation content:

undefined

Animation captions:

1. Input abba is written on the tape, followed by *'s. The start state is q_0 and the head is at the first a. In step 1, the head moves right. The state is unchanged.
2. In the next step, the head moves right, the state transitions to q_1 and the tape does not change.
3. In the next step, the state transitions to q_{acc}. The Turing machine halts and accepts.
4. Input abaa is written on the tape, followed by *'s. The start state is q_0 and the head is at the first a. In step 1, the head moves right. The state is unchanged.
5. In the next step, the head moves right, the state transitions to q_1 and the tape does not change.
6. After the next two steps, the current symbol is * and the state is q_1. In the next step, the state goes to q_{rej}. The Turing machine halts and rejects.

A more complex Turing machine

The next example of a Turing machine has an input alphabet with just one symbol: {a}. The Turing machine accepts if the number of a's is a power of two. The tape alphabet contains three symbols: {a, x, *}. Although there are no x's on the tape at the beginning of the execution of the Turing machine, x symbols are written on the tape to help keep track of what is going on in the execution.

The Turing machine uses the fact that if a positive integer n is a power of two, then it is possible to repeatedly divide the number evenly by 2 until the number gets down to 1. The head of the Turing machine shuttles back and forth between the left end and the right end of the Turing machine. As the head moves from left to right, every other a is changed to an x. The Turing machine keeps track of whether an even or an odd number of a's have been encountered. If the head reaches the right end of the string after having seen an odd number of a's, the string is rejected. If the Turing machine reaches the end of the the input string after having encountered an even number of a's, the head moves back to the left end of the string and continues. The Turing machine loops, changing every other a to an x, until there is only one a in the string.

At the first step, the Turing machine changes the leftmost 'a' to a * symbol in order to mark the leftmost end of the input string. When the head moves left and encounters the * symbol, that triggers the head to change directions. The leftmost * symbols counts as an 'a' in determining whether there are an odd or even number of a's in the string.

The transition function for the Turing machine is given below so that the definition is precise. However, the transition function is not very helpful in understanding how and why the Turing machine works. It's best to understand the action of the Turing machine by actually running the Turing machine on different inputs. The tool below allows you to experiment with the execution of the Turing machine on different inputs. Remember that the input alphabet is just the single symbol a, so the tape has to be initialized to a sequence of a's following by blank symbols.

Table 8.5.3: Transition function for the Turing machine that recognizes powers of 2.

-	a	x	*
q_0	(q_{first}, *, R)	(q_{rej}, *, R)	(q_{rej}, x, R)
q_{first}	(q_{even}, x, R)	(q_{first}, x, R)	(q_{acc}, *, R)
q_{even}	(q_{odd}, a, R)	(q_{even}, x, R)	(q_{ret}, *, L)
q_{odd}	(q_{even}, x, R)	(q_{odd}, x, R)	(q_{rej}, *, L)

q_{ret}	(q_{ret}, a, L)	(q_{ret}, x, L)	$(q_{first}, *, R)$

PARTICIPATION ACTIVITY 8.5.4: Turing machine that recognizes powers of 2.

Start | Modify tape | Next | Reset

Symbol tape	**a**	**a**	**a**	**a**	*	*	*	*	*	*

Transition table for the Turing machine

	a	**x**	*
q_0	$(q_{first}, *, R)$	(q_{rej}, x, R)	$(q_{rej}, *, R)$
q_{first}	(q_{even}, x, R)	(q_{first}, x, R)	$(q_{acc}, *, R)$
q_{even}	(q_{odd}, a, R)	(q_{even}, x, R)	$(q_{ret}, *, L)$
q_{odd}	(q_{even}, x, R)	(q_{odd}, x, R)	$(q_{rej}, *, L)$
q_{ret}	(q_{ret}, a, L)	(q_{ret}, x, L)	$(q_{first}, *, R)$

PARTICIPATION ACTIVITY 8.5.5: Power of 2 Turing machine.

The questions below refer to the Turing machine from the tool above whose input alphabet is {a}. The Turing machine accepts a string of a's if and only if the number of a's is a power of 2. Use the tool to answer the following questions.

1) If the Turing machine starts with input "aaaa", what are the contents of the tape after four steps?
   - ○ axax
   - ○ *xax
   - ○ xaxa

2) If the Turing machine starts with input "aaaaa", what is the state of the Turing machine after 6 steps?

- q_{rej}
- q_{acc}
- q_{ret}

3) How many times does the head move from left to right on input "aaaa" before it halts?

- 1
- 2
- 3

CHALLENGE ACTIVITY 8.5.1: Turing machine transitions.

455912.3056722.qx3zqy7

Start

Use the input and table to execute.

Input: aaabbb

	a	b	*
q0	(q3, a, R)	(q0, b, R)	(q2, a, R)
q1	(q0, *, R)	(q2, b, R)	(q1, b, L)
q2	(q3, *, L)	(q0, a, L)	(q2, *, R)
q3	(q0, b, R)	(q3, a, R)	(q1, b, R)

First 6 characters of the tape after step 1: Ex: *abb*b

Select the state of the Turing Machine after each step:

1 2 3 4

Check Next

Additional exercises

EXERCISE 8.5.1: Simulating a Turing machine - example 1.

Here is a description of a Turing machine. The input alphabet is {a, b}. The state set is:

$$\{ q_0, q_1, q_2, q_3, q_4, q_{acc}, q_{rej} \}$$

The transition function is given in the table below:

	q_0	q_1	q_2	q_3	q_4
a	(q_1, a, R)	(q_1, a, R)	(q_2, a, R)	(q_{acc}, a, R)	(q_{rej}, a, R)
b	(q_2, b, R)	(q_1, b, R)	(q_2, b, R)	(q_{rej}, b, R)	(q_{acc}, b, R)
*	$(q_{rej}, *, R)$	$(q_3, *, L)$	$(q_4, *, L)$	$(q_{rej}, *, R)$	$(q_{rej}, *, R)$

(a) Draw the configuration of the Turing machine after 3 steps on input "aabba". When the Turing machine is in the initial configuration, it has executed zero steps.

(b) For how many steps does the Turing machine run on input string "a" before the Turing machine halts? Does the Turing machine accept the string "a"?

(c) Simulate the Turing machine on input "aaba". Does it accept? Draw the final two configurations of the Turing machine computation. In the last configuration, the Turing machine is either in the accept or reject state.

(d) Simulate the Turing machine on input "aabb". Does it accept? Draw the final two configurations of the Turing machine computation. In the last configuration, the Turing machine is either in the accept or reject state.

(e) Determine whether the Turing Machine accepts or rejects strings "bab" and "bba".

(f) Describe in words the conditions under which the Turing machine accepts the input string.

EXERCISE 8.5.2: Simulating a Turing machine - example 2.

Here is a description of a Turing machine. The input alphabet is {a, b}. The state set is:

$$\{ q_0, q_a, q_b, q_{ca}, q_{cb}, q_{left}, q_{acc}, q_{rej} \}$$

The transition function is given in the table below:

	q_0	q_a	q_b	q_{ca}	q_{cb}	q_{left}
a	(q_a, *, R)	(q_a, a, R)	(q_b, a, R)	(q_{left}, *, L)	(q_{rej}, *, R)	(q_{left}, a, L)
b	(q_b, *, R)	(q_a, b, R)	(q_b, b, R)	(q_{rej}, *, R)	(q_{left}, *, L)	(q_{left}, b, L)
*	(q_{acc}, *, R)	(q_{ca}, *, L)	(q_{cb}, *, L)	(q_{acc}, *, R)	(q_{acc}, *, R)	(q_0, *, R)

(a) Draw the configuration of the Turing machine after 3 steps on input "aabba". When the Turing machine is in the initial configuration, it has executed zero steps.

(b) For how many steps does the Turing machine run on input string "a" before the Turing machine halts? Does the Turing machine accept the string "a"?

(c) Simulate the Turing machine on input "aba". Does it accept? Draw the configuration of the Turing machine after the Turing machine executes 4 steps on input "aba".

(d) Simulate the Turing machine on input "abba". Does it accept? Draw the configuration of the Turing machine after the Turing machine executes 5 steps on input "abba".

(e) Simulate the Turing machine on input "abbbaa". Does it accept? Draw the final two configurations of the Turing machine computation. In the last configuration, the Turing machine is either in the accept or reject state.

(f) Describe in words the conditions under which the Turing machine accepts the input string.

EXERCISE 8.5.3: Turing machine design.

Design a Turing machine with input alphabet {a, b} that accepts a string if and only if the string has the property described.

(a) The input string has an even number of b's.

(b) The input string has the same number of a's and b's.

8.6 Decision problems and languages

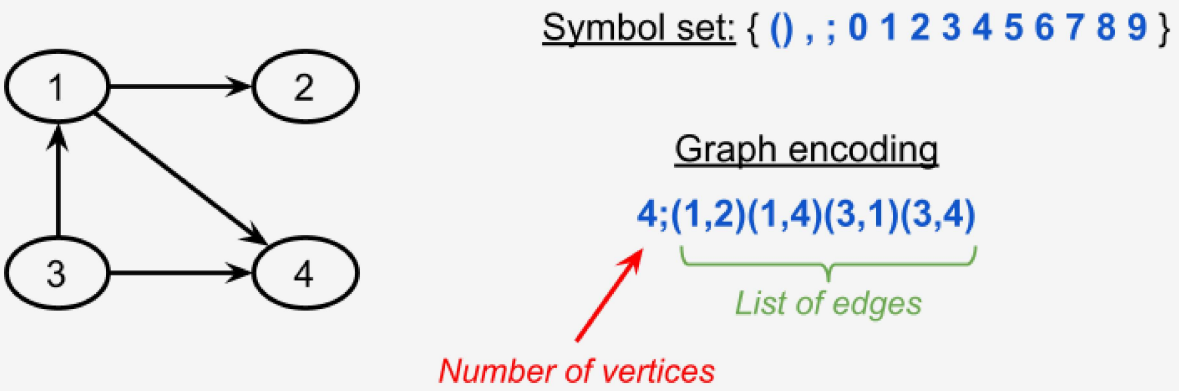
This section has been set as optional by your instructor.

At first glance Turing machines seem limited in terms of the kinds of problems they can solve. For example, Turing machines can only handle inputs in the form of strings, whereas many important computational problems operate on different kinds of mathematical objects such as graphs, boolean expressions, or numbers. Of course, any computer program receives input in the form of a string of 0's and 1's, so the fact that Turing machines require every input to be encoded as a string over a finite alphabet is not a special requirement of Turing machines alone.

Consider, for example, a problem whose input is a directed graph. A computer program that solves a graph problem might read the input graph from a text file. The program would read the characters in the file and interpret them in a particular way. One way to encode a graph would be to first indicate the number of vertices in the graph, followed by a semicolon. The labels for the vertices are the integers in the range from 1 up to and including the number of vertices in the graph. The encoding then lists each edge as an ordered pair in the format:

(*Vertex label of the tail of the edge*, *Vertex label of the head of the edge*)

There are many other perfectly acceptable ways to encode a graph as a string of symbols. The important point is that the input file is prepared using the same encoding as the program that will use the file, so that the contents of the input file are interpreted correctly. The picture below shows a small directed graph and a string encoding of the graph according to the encoding scheme described above.



Once a symbol set and method of encoding a graph are decided, there will be many strings that do not correspond to a valid graph. The Turing machine should be able to recognize if an input string is not a valid encoding of a graph and reject all such inputs immediately, just as a computer program will issue an error message and stop if an error is detected in the formatting of an input file.

PARTICIPATION ACTIVITY 8.6.1: Recognizing valid graph encodings.

1) According to the graph encoding scheme described above, which string corresponds to a valid encoding of a graph?

- 4;(1,3)(2,3)(1,4)(5,3)(5,2)
- (1,3)(2,3)(1,4)(5,3)(5,2)

- 5;(1,3)(2,3)(1,4)(5,3)(5,2)
- 5(1,3)(2,3)(1,4)(5,3)(5,2)

Another limitation of Turing machines as described is that their only output is to accept or reject an input string. The class of problems for which the answer is "Yes" or "No" are called ***decision problems***. Consider the following two problems:

- Decision problem: Given a Boolean expression, is there an assignment to the Boolean variables that causes the expression to evaluate to 1?
- Search problem: Given a Boolean expression, find an assignment to the Boolean variables that causes the expression to evaluate to 1 if one exists, or output that no such assignment exists.

Applications typically solve search problems rather than decision problems. However, in most cases, a search problem has a corresponding decision problem that captures the essential difficulty of the problem. That is, an efficient algorithm that solves the decision problem can be used to solve the corresponding search problem efficiently. Since decision problems are easier to reason about, a discussion of the computational complexity of a problem typically focuses on the decision version of the problem.

PARTICIPATION ACTIVITY 8.6.2: Decision vs. Search problems.

Indicate whether the following problems are decision or search problems.

1) Given a list of cities and distances between each pair of cities, what is the shortest route that reaches every city?
   - Decision
   - Search

2) Given a list of cities and distances between each pair of cities, how long is the shortest route that reaches every city?
   - Decision
   - Search

3) Given a list of cities and distances between each pair of cities, is there a route shorter than 1000 miles that visits every city?
   - Decision

- Search

4) Given an integer N such that N > 1, is there an integer greater than one and less than N that evenly divides N?
   - Decision
   - Search

5) Given an integer N such that N > 1, find an integer greater than one and less than N that evenly divides N or indicate that no such integer exists.
   - Decision
   - Search

Consider a finite alphabet Σ. The set of all finite strings over Σ is denoted by Σ^*. Note that the strings themselves are finite but the set of all possible strings is infinite since there is no limit to the length of a string. For example, if B = {0, 1}, the set B^+ is: {0, 1, 00, 01, 10, 11, 000, …}.

If Σ is a finite alphabet, then a subset of Σ^* is called a ***language*** over Σ. Every decision problem (along with an encoding of the input objects to strings in Σ^*) defines a language over Σ as follows: $x \in \Sigma^*$ is in the language if x is a valid encoding of an input and the answer to the decision problem on input x is "Yes". The string x is not in the language if either x does not correspond to a valid input or the answer on input x is "No".

Definition 8.6.1: Language computed by a Turing machine.

Let Σ denote a finite alphabet and let L be a language over Σ. A Turing machine M ***computes language L*** (or ***decides language L***) if for every $x \in \Sigma^*$, if $x \in L$, then M accepts x in a finite number of steps and if $x \notin L$, then M rejects x in a finite number of steps.

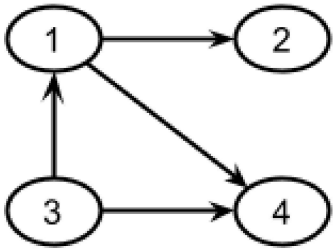
The model of Turing machines and the formulation of decision problems as languages provide a mathematically well-defined way to study what kinds of problems can be solved using any kind of computing device. Turing machines also provide a way to measure the resources used to solve a problem. ***Time complexity*** is measured by the number of steps taken by a Turing machine on a particular input. ***Space complexity*** is measured by the number of tape cells that the Turing machine uses in the course of its execution on a particular input.

PARTICIPATION ACTIVITY 8.6.3: Decision problems to languages.

Consider the following decision problem:

Given a graph G, does G have an even number of edges?

Define a language L corresponding to the decision problem that uses the graph encoding illustrated in the diagram below. The alphabet Σ for L consists of a comma, semicolon, left and right parentheses and all ten digits. Recall that the graph encoding indicates the number of vertices, followed by a semicolon, followed by a list of the edges in the graph:



Alphabet: Σ = { ( ) , ; 0 1 2 3 4 5 6 7 8 9 }

Graph encoding

4;(1,2)(1,4)(3,1)(3,4)

List of edges

Number of vertices

A string is in the language if and only if the string corresponds to a valid encoding of a graph and the graph has an even number of edges. Indicate whether the following strings are in the language L.

1) 4;(1,2)(2,3)(3,4)(3,2)(1,4)
   - ○ in L
   - ○ not in L

2) 4;(a,b)(a,c)(c,d)(a,d)
   - ○ in L
   - ○ not in L

3) 4;(1,2)(2,3)(3,4)(3,2)
   - ○ in L
   - ○ not in L

4) (1,2)(2,3)(3,4)(3,2)
   - ○ in L
   - ○ not in L

5) 09(;;)23,4,3()3,4,5()));;;
   - ○ in L
   - ○ not in L

Additional information 8.6.1: Uncomputable problems.

Turing machines may not be the preferred model for devising algorithms to solve real problems, but they are useful for reasoning about what we can and cannot hope to solve with a computer. As observed earlier, the description of a Turing machine contains 6 components: the tape alphabet, the set of states, the start state, the accept state, the reject state, and the transition function. Each component can be expressed as a string of symbols over a finite alphabet. For example, the states could be denoted as: q0, q1, q2, etc. The tape characters could be denoted as s0, s1, s2, etc. Each transition rule can be expressed as a sequence, e.g., (q3, s2) → (q1, s17, L). Therefore, we can fix in advance a finite set of letters, digits, and punctuation symbols, and encode any Turing machine as a string of the selected characters. Encoding a Turing machine as a string is similar to representing a computer program as a string of symbols in a text file. Denote the encoding of Turing machine M as <M>. Since <M> is just a string over a finite alphabet, <M> can be used as the input to another Turing machine. Consider the following problem:

The Halting Problem

- Input: <M> the description of a Turing machine and an input x.
- Output: Accept if M halts on input x. Reject if M runs forever on input x.

It can be proven that there is no Turing machine that can compute the language defined by the Halting problem. A language is ***uncomputable*** if there is no Turing machine that computes the language. The fact that the halting problem is uncomputable has profound implications for areas in computer science such as program verification. For example, it is impossible to write an automatic verifier that takes as input a computer program and determines whether the program always terminates, or whether the program computes the correct answer to a particular problem.

Additional exercises

EXERCISE 8.6.1: Decision problems from search problems.

Give a decision problem corresponding to each of the search problems given below.

(a)

- Input: A set of classes to be scheduled. A list of pairs of the classes which can not be scheduled during the same period.
- Output: The largest set of classes that can all be scheduled during the same period.

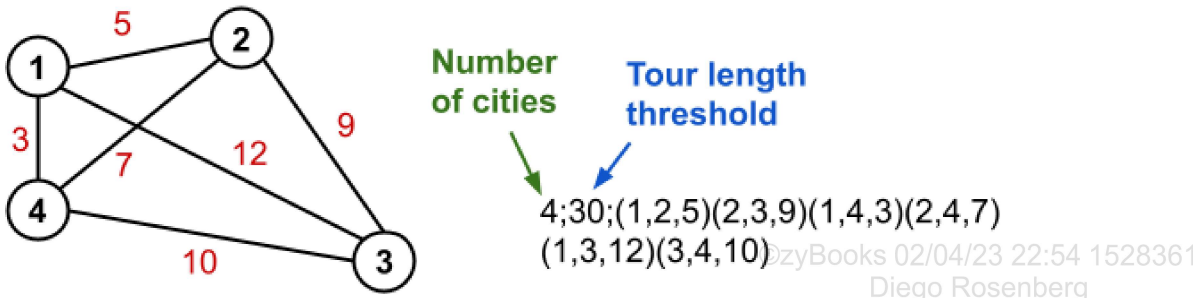
(b)
- Input: A set of classes to be scheduled. A list of pairs of the classes which can not be scheduled during the same period.
- Output: A schedule for the classes that uses the smallest number of periods.

(c)
- Input: A list of items, each with a value and a weight. The values and weights of the items are positive integers. A positive integer W.
- Output: A subset of the items whose total weight is at most W and whose total value is as large as possible.

EXERCISE 8.6.2: Valid encodings of the Traveling Salesman Problem.

The Traveling Salesman Problem (TSP) is defined as follows: the input is a set of cities and a distance between each pair of cities. The problem is to find a tour of all the cities so that the salesman ends up at the same place he started so that he visits each city exactly once and travels the shortest possible distance. In the decision version of the problem, the input also includes a threshold k and the question is whether there is a tour that visits every city exactly once and whose total distance is at most k.

Here is an encoding of the Traveling Salesman Problem. The input alphabet is {0, 1, 2, 3, 4, 5, 6, 7, 8, 9} along with a comma, semicolon, left and right parentheses. A valid encoding of a Traveling Salesman Problem will start with two numbers, each followed by a semicolon. The first number is the number of cities and the second number is the threshold for the tour. (Note that since we don't have the character "-" or ".", all numbers are non-negative integers). Each city will have a unique name which is a number in the range from 1 through n, where n is the number of cities. The encoding then lists triplets of the form (i, j, d) indicating that the distance between cities i and j is d. If the distance between two cities is given more than once or not at all, the encoding is not valid. The diagram below shows a picture of 4 cities and their distances along with a valid encoding for that particular input to the Traveling Salesman Problem:



The Traveling Salesman language is the set of all strings that correspond to valid encodings and are "yes" inputs for the Traveling Salesman decision problem. Which of the following strings are in the language and why?

(a) 4;10;(1,2,1)(1,3,5)(2,3,4)(1,4,5)(2,4,2)(3,4,3)

(b) 5;10;(1,2,1)(1,3,5)(2,3,4)(1,4,5)(2,4,2)(3,4,3)

(c) 4;11;(1,2,1)(1,3,5)(2,3,4)(1,4,5)(2,4,2)(3,4,3)

(d) 4;10;(1,2,1)(1,3,5)(1,4,5)(2,4,2)(3,4,3)

(e) 4;10;(1,2,1)(1,3,5)(2,3,4)(1,4,5)(2,4,2)(3,4,3)(2,4,2)

(f) 4;11;(((1,2,1)(1,3,5)8(2,3,4)(1,4,5)(2,4,2)(3,4,3)

©zyBooks 02/04/23 22:54 1528361
Diego Rosenberg
NYUCSBRWinter2023