An algorithm is a step-by-step procedure for performing some task in a finite amount of time.

Experimental studies on running times are useful, but they have some limitations:

- Experiments can be done only on a limited set of test inputs, and care must be taken to make sure these are representative.
- It is difficult to compare the efficiency of two algorithms unless experiments on their running times have been performed in the same hardware and software environments.
- It is necessary to implement and execute an algorithm in order to study its running time experimentally.

Thus, while experimentation has an important role to play in algorithm analysis, it alone is not sufficient. Therefore, in addition to experimentation, we desire an analytic framework that

- Takes into account all possible inputs
- Allows us to evaluate the relative efficiency of any two algorithms in a way that is independent from the hardware and software environment
- Can be performed by studying a high-level description of the algorithm without actually implementing it or running experiments on it.

"Algorithm A runs in time proportional to n" =>
If we were to perform experiments, then we would find that the actual running time of algorithm A on any input of size n never exceeds c.n, where c is a constant that depends on the hardware and software environment used.

Given two algorithms A and B, where A runs in time proportional to n and B runs in time proportional to n^2 , we will prefer A to B, since the function n grows at a smaller rate than the function n^2 .

"Algorithm A runs in time proportional to n" =>
If we were to perform experiments, then we would find that the actual running time of algorithm A on any input of size n never exceeds c.n, where c is a constant that depends on the hardware and software environment used.

- A language for describing algorithms
- A computational model that algorithms execute within
- A metric for measuring algorithm running time
- An approach for characterizing running times, including those for recursive algorithms.

Algorithm arrayMax(A, n):
Input: An array A storing $n \geq 1$ integers.
Output: The maximum element in A.

```
currentMax ← A[0]
for i ← 1 to n − 1 do
    if currentMax < A[i] then
        currentMax ← A[i]
return currentMax
```

By inspecting the pseudocode, we can argue about the correctness of algorithm arrayMax with a simple argument. Variable *currentMax* starts out being equal to the first element of A. We claim that at the beginning of the ith iteration of the loop, *currentMax* is equal to the maximum of the first i elements in A. Since we compare *currentMax* to $A[i]$ in iteration i, if this claim is true before this iteration, it will be true after it for $i + 1$ (which is the next value of counter i). Thus, after $n - 1$ iterations, *currentMax* will equal the maximum element in A. As with this example, we want our pseudocode descriptions to always be detailed enough to fully justify the correctness of the algorithm they describe, while being simple enough for human readers to understand.

- Assigning a value to a variable
- Calling a method
- Performing an arithmetic operation
- Comparing two numbers
- Indexing into an array
- Following an object reference
- Returning from a method.

Specifically, a primitive operation corresponds to a low-level instruction with an execution time that depends on the hardware and software environment but is nevertheless constant. Instead of trying to determine the specific execution time of each primitive operation, we will simply ***count*** how many primitive operations are executed, and use this number t as a high-level estimate of the running time of the algorithm. This operation count will correlate to an actual running time in a specific hardware and software environment, for each primitive operation corresponds to a constant-time instruction, and there are only a fixed number of primitive operations. The implicit assumption in this approach is that the running times of different primitive operations will be fairly similar. Thus, the number, t, of primitive operations an algorithm performs will be proportional to the actual running time of that algorithm.

RAM (Random Access Machine) Model -
A computer is viewed simply as a CPU connected to a bank of memory cells.
Each memory cell stores a word, which can be a number, a string, or an address.
The term "random access" refers to the ability of the CPU to access an arbitrary memory location using just one single primitive operation.
We assume the CPU in the RAM model can perform any primitive operation in a constant number of steps, which do not depend on the size of the input.

Algorithm arrayMax(A, n):
Input: An array A storing $n \geq 1$ integers.
Output: The maximum element in A.

```
currentMax ← A[0]
for i ← 1 to n − 1 do
    if currentMax < A[i] then
        currentMax ← A[i]
return currentMax
```

- Initializing the variable *currentMax* to $A[0]$ corresponds to two primitive operations (indexing into an array and assigning a value to a variable) and is executed only once at the beginning of the algorithm. Thus, it contributes two units to the count.
- At the beginning of the for loop, counter i is initialized to 1. This action corresponds to executing one primitive operation (assigning a value to a variable).
- Before entering the body of the for loop, condition $i < n$ is verified. This action corresponds to executing one primitive instruction (comparing two numbers). Since counter i starts at 1 and is incremented by 1 at the end of each iteration of the loop, the comparison $i < n$ is performed n times. Thus, it contributes n units to the count.
- The body of the for loop is executed $n - 1$ times (for values $1, 2, \ldots, n - 1$ of the counter). At each iteration, $A[i]$ is compared with *currentMax* (two primitive operations, indexing and comparing), $A[i]$ is possibly assigned to *currentMax* (two primitive operations, indexing and assigning), and the counter i is incremented (two primitive operations, summing and assigning). Hence, at each iteration of the loop, either four or six primitive operations are performed, depending on whether $A[i] \leq$ *currentMax* or $A[i] >$ *currentMax*. Therefore, the body of the loop contributes between $4(n - 1)$ and $6(n - 1)$ units to the count.
- Returning the value of variable *currentMax* corresponds to one primitive operation, and is executed only once.

Algorithm arrayMax(A, n):
Input: An array A storing $n \geq 1$ integers.
Output: The maximum element in A.

```
currentMax ← A[0]
for i ← 1 to n − 1 do
    if currentMax < A[i] then
        currentMax ← A[i]
return currentMax
```

To summarize, the number of primitive operations $t(n)$ executed by algorithm arrayMax is at least

$$2 + 1 + n + 4(n - 1) + 1 = 5n$$

and at most

$$2 + 1 + n + 6(n - 1) + 1 = 7n - 2.$$

The best case ($t(n) = 5n$) occurs when $A[0]$ is the maximum element, so that variable *currentMax* is never reassigned. The worst case ($t(n) = 7n - 2$) occurs when the elements are sorted in increasing order, so that variable *currentMax* is reassigned at each iteration of the for loop.

We will, for the remainder of this course, typically characterize running times in terms of the worst case. We say, for example, that algorithm arrayMax executes t(n) = 7n − 2 primitive operations in the worst case, meaning that the maximum number of primitive operations executed by the algorithm, taken over all inputs of size n, is 7n − 2.

This type of analysis is much easier than an average-case analysis, as it does not require probability theory; it just requires the ability to identify the worst-case input, which is often straightforward. In addition, taking a worst-case approach can actually lead to better algorithms. Making the standard of success that of having an algorithm perform well in the worst case necessarily requires that it perform well on every input.