# **What Do We Care About When Creating Algorithms?**

When making and analyzing algorithms, we care about:

1. **Correctness**:
   1. **Does the algorithm do what is intended?**
2. **Performance**:
   1. **Speed: time complexity**
   2. **Memory: space complexity**

# **Why Analyze Algorithms?**

* To make good design decisions
* Enable you to look at an algorithm (or code) and identify the bottlenecks, etc.

# **Complexity**

* As you have progressed in your programming career, you’ve written increasingly complex programs. You’re also realized that there are often many ways to solve the same problem.
* How do you compare different solutions to the same problem to see which is better?
* The technical term that refers to algorithms’ runtime is ***complexity***.
* Complexity is a measure of the computing resources that are used by a piece of code
  + time,
  + memory,
  + disk space,
  + etc.
* An algorithm with higher complexity uses more time or resources to solve a problem.

# **Empirical Analysis**

* One way to determine an algorithm’s approximate time complexity is to program it, run the program, and measure how long it takes to run.
* This is sometimes called an ***empirical analysis***of the algorithm.
* For example, consider two algorithms to search an array:
  + one that sequentially searches for the desired target element,

and

* + one that first sorts the array and then performs a binary search on the sorted array.
* You could empirically analyze the algorithms by writing both as programs, running them on the same input, and timing them.
* But empirically analyzing an algorithm *isn’t a very reliable measure*, because on a different computer with a different processor speed and more or less memory the program might not take the same amount of time.
* Also, in order to empirically test an algorithm, you must write it and time it, which can be a chore.

# **Algorithm Analysis**

* A more neutral way to measure a program’s performance is to examine its code or pseudocode and roughly count the number of statements that are executed.
* This is a form of ***algorithm analysis***, the practice of applying techniques to mathematically approximate the performance of various computing algorithms.
* This is where we differentiate between:

1. **Performance**: how much time/memory/disk/... is actually used when a program is run. This depends on the machine, compiler, etc. as well as the code.
2. **Complexity**: how do the resource requirements of a program or algorithm scale, i.e., what happens as the size of the problem being solved gets larger.

# **Time Complexity**

* Usually when we talk about the efficiency of a program, we are talking about how long the program takes to run, or its ***time complexity****.*
* Time complexity is the reason we perform algorithm analysis.
* The time complexity for a program to be “fast enough” depends on the task.
  + A program running on a modern computer that requires five minutes to look up a dictionary word is probably too slow.
  + An algorithm that renders a complex three-dimensional movie scene in five minutes is probably very fast.

So, how can you measure time complexity?

* The runtime of an algorithm is proportional to the number of "basic operations" that it performs.
* The total number of basic operations we are given in an algorithm is a function of a given ***input of size*** ***(n)***.

# **Measuring Time Complexity**

1. **Single** **Statement**

Any ***single statement*** takes a **constant** amount of time to run

**statement 1;**

This code takes 1 unit of time to run.

If a statement takes only 1 unit of time to run, we say it takes constant time.

1. **Sequence of Statements**

The runtime of a ***sequence* of *statements*** is the **sum** of their runtimes

**statement 1;**

**statement 2;**

**statement 3;**

**...**

The total runtime is found by adding the times of all the statements.

total time = time(statement 1) + time(statement 2) + time(statement 3)

If each statement is a basic operations, then the time for each statement is constant and the total time is also constant: O(1). We say that O(3) = O(1). O(100000) is also equal to O(1).

In the following examples, assume the statements are simple unless noted otherwise.

1. **if/else**

An ***if/else’s*** runtime is the runtime of the **if test**, **plus** the runtime of **worst-case branch**.

We take the worst-case branch. This worst-case branch acts as the worst-case runtime for the entire if/else if/else statement body.

The body of a branch is made up of a sequence of statements.

Therefore, we need to find the branch with the worst-case by adding each branch’s runtime of the **condition + body**

**if (condition) {**

**sequence of statements 1**

**}**

**else if(condition) {**

**sequence of statements 2**

**}**

**else {**

**sequence of statements 3**

**}**

In this example, there are three possible branches to be taken.

The worst-case time is the slowest of the three branches:

branch 1 = (if condition) + (sequence 1)

branch 2 = (else if condition) + (sequence 2)

branch 3 = (else condition) + (sequence 3)

**max**(time(branch 1), time(branch 2), time(branch 3))

For example:

if (time(N)) {

time(1)

}

else if(time(1)) {

time(N \* N)

}

else {

time(logN)

}

Notice how there can be a mix of varying runtimes between the conditions and the bodies.

In this example, the worst-case time for the whole if-then-else statement would be O(N \* N).

1. **Loops**

A ***loop’s*** runtime, if the loop repeats *N* times, is ***N* times** the runtime of the statements in its body

for (i = 0; i < N; i++) {

sequence of statements

}

We can quantify the runtime of a loop by using the formula: **N \* body**, where

N is the number of times we execute the loop body

body is the runtime of all statements inside the loop

1. **Method Calls**

A ***method call’s*** runtime is measure by the total number of statements inside the method’s body

# **More Ways of Measuring Complexities**

**Constant Time Operations**

* For the purposes of simplification, let’s assume that the following actions require an equal and constant amount of time to execute:
* **Variable declarations and assignments**
* **Evaluating mathematical and logical expressions**
* **Accessing or modifying an individual element of an array**
* **Simple method calls (where the method does not perform a loop)**
* **etc…**

**Consecutive Statements**

* From the preceding simple rules, we can extrapolate the runtimes of larger and more complex pieces of code.
* For example, the runtime of a group of statements in sequential order is the sum of the individual runtimes of the statements:

Text

Description automatically generated with medium confidence

* These statements themselves may take varying times to complete
  + statement1 could take N time
  + statement2 could take 1 time
  + statement2 could take logn time
* We take the **sum** of these group of statements in **sequential order** to find the runtime for the group of statements
  + N + 1 + logn

**Loops**

* One kind of variable that does not require a constant amount of time to initialize is an array.
* Arrays come in many different sizes.
* When an array is constructed, Java zeroes each array element.
* If this array had a size of 10, we would have to zero out an array element 10 times.

* Some types of objects also have lengthy code in their constructors that makes them take even longer to construct.
* The runtime of a loop is roughly equal to
  + the number of iterations of the loop

**multiplied by**

* + the runtime of its body

**runtime of loop = iterations of the loop \* runtime of body**

* For example, a loop with a body that contains *K* simple statements that repeats *N* times will have a runtime of roughly (*K* \* *N*):

Graphical user interface, text, application

Description automatically generated

* This example has three statements that each take constant time to execute.
* We multiply the number of iterations in the loop, N, by the body, 3, to get 3N.

**Consecutive Loops**

* The runtime of multiple loops placed sequentially (not nested) with other statements is the
  + sum of the loops’ runtimes

**plus**

* + the other statements’ runtimes:

**runtime = runtime of loop + runtime of loop + other statements’ runtime**

Diagram

Description automatically generated with medium confidence

**Nested Loops**

* The runtime of a loop containing a nested loop is roughly equal to the runtime of the
  + inner loop

**multiplied by**

* + the number of repetitions of the outer loop

**runtime of loop = iterations of the outer loop \* iterations of the inner loop \* runtime of body**

Diagram

Description automatically generated

* Normally, the loops in long-running algorithms are processing some kind of data.
* Many algorithms run very quickly if the input dataset is small, so we generally worry about the performance only for large datasets.
* For example, consider the following set of loops that process an array of *N* elements:

Diagram

Description automatically generated

**Possible Gotchas**

1. **Multi-Part Algorithms: Add vs. Multiply**

Suppose you have an algorithm that has two steps.

When do you multiply the runtimes and when do you add them?

This is a common source of confusion.

A white paper with black writing

Description automatically generated with low confidence

In the example on the left, we do A chunks of work then B chunks of work.

Therefore, the total amount of work is O(A + B).

In the example on the right, we do B chunks of work for each element in A.

Therefore, the total amount of work is O(A \* B).

In other words:

If your algorithm is in the form "**do this, then, when you're all done, do that**" then you add the runtimes.

If your algorithm is in the form "**do this for each time you do that**" then you multiply the runtimes.

It's very easy to mess this up in an interview, so be careful.

1. **Don’t Drop Additive Terms**

When attempting to drop all terms except the most dominant, it may be tempting to drop a term from an additive Big O like O(A + B), especially if you know that, for example, A is always supposed to be greater than B.

However, it is impossible to put this precondition on two variables whos sizes may vary between runtimes. Therefore, it is always better to abstract them, and assume A and B have no relationship, especially since they aren’t nested.

They are two completely separate entities in two separate loops (or whatever operation you’re doing).