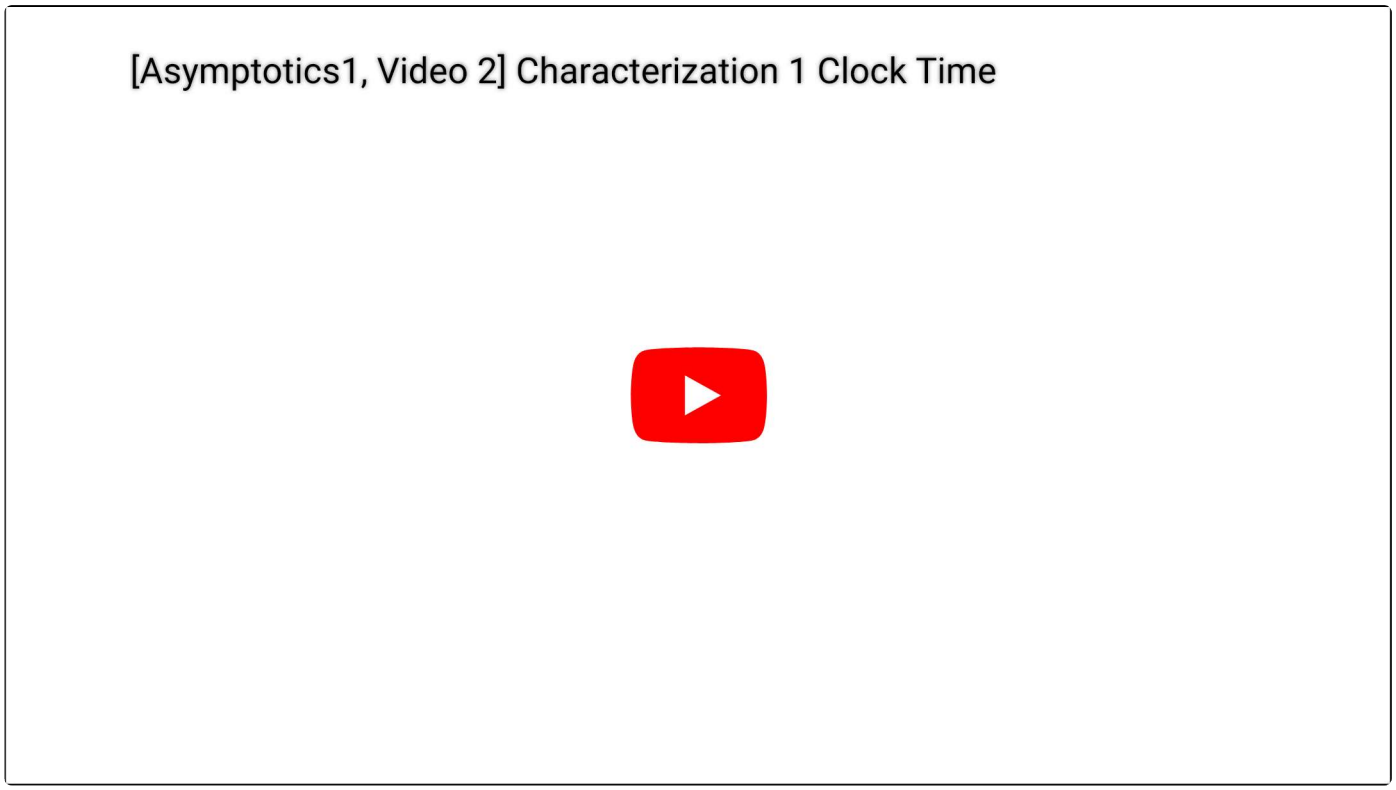
13.2 Runtime Characterization

Techniques for Measuring Computational Cost.

We want to be able to *characterize* the runtime of the two algorithms we previously saw. In other words, we want to come up with some proxy that communicates the overall performance of each algorithm.

When characterizing runtimes, we have two goals in mind:

- They should be simple but mathematically rigorous.
- They should clearly demonstrate the superiority of one algorithm over another if one algorithm is better than the other.



We have converted both the naïve algorithm and the better algorithm into Java code. The function dup1 corresponds to the naïve algorithm and dup2 corresponds to the better algorithm.

```java
// Naïve algorithm: compare everything
public static boolean dup1(int[] A) {
  for (int i = 0; i < A.length; i += 1) {
    for (int j = i + 1; j < A.length; j += 1) {
      if (A[i] == A[j]) {
         return true;
      }
    }
  }
  return false;
}
```

```java
// Better algorithm: compare only neighbors
public static boolean dup2(int[] A) {
  for (int i = 0; i < A.length - 1; i += 1) {
    if (A[i] == A[i + 1]) {
      return true;
    }
  }
  return false;
}
```

Technique 1

The first technique we will consider using for runtime characterization is directly measuring execution time in seconds using a client program.

There are a few ways we could do this:

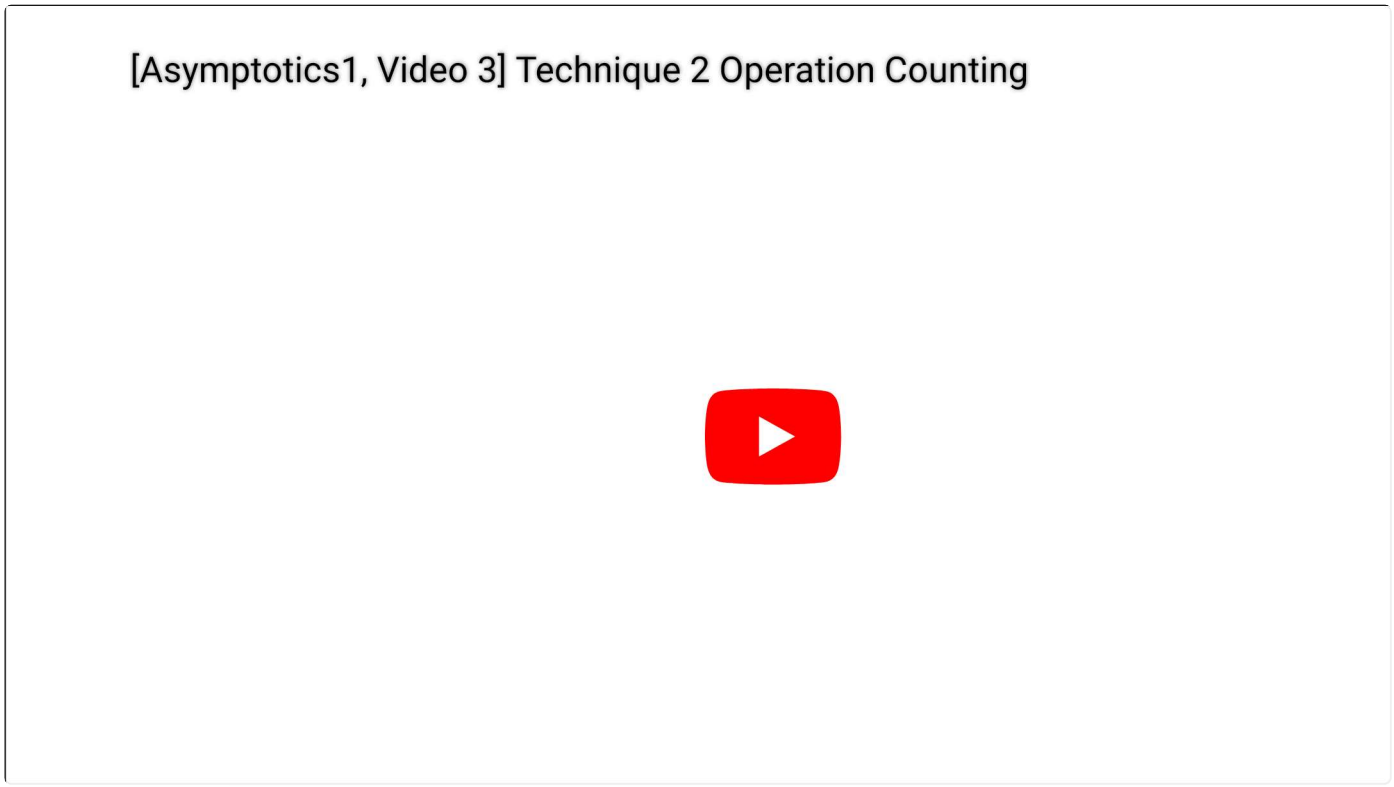
- Use a physical stopwatch (not recommended).
- Use Unix's built in `time` command.
- Use the Princeton Standard library which has a `stopwatch` class.

Using any of these methods will show (with varying levels of accuracy, depending on whether the physical stopwatch path was chosen) that as input size increases, `dup1` takes a longer time to complete, whereas `dup2` completes at relatively around the same rate.

It seems like technique 1 works perfectly fine for characterizing our runtimes! It is very easy to understand the results of the experiment, and it is easy to implement. However, there are some serious cons associated it that dissuades us from using it for everything:

- It could take a *long* time to finish running.
- Running times can vary by machine, compiler, input data, etc.

For these reasons, technique 1 does not meet our goals in characterizing runtimes. It's simple, but it's not mathematically rigorous. Moreover, the differences based on machine, compiler, input, etc. mean that the results may not clearly demonstrate the relationship between `dup1` and `dup2`.



Technique 2A

Rather than physically timing the amount of time it takes for the algorithm to run, we can instead count the total number of operations completed by each algorithm! Intuitively, the algorithm that runs the fewer number of operations would be superior to the other. We can fix the input size `N` to be the same for both algorithms and compare the number of operations run.

Let us apply this to the `dup1` algorithm with an input size of `N=10000`:

```
for (int i = 0; i < A.length; i += 1) {
  for (int j = i+1; j < A.length; j += 1) {
    if (A[i] == A[j]) {
       return true;
    }
  }
}
return false;
```

Operation	Count (for N=10000)
i = 0	1
j = i+1	1 (in the best case) to 10000 (in the worst case)
<	2 to 50,015,001
+= 1	0 to 50,005,000
==	1 to 49,995,000
array accesses	2 to 99,990,000

The operation `i = 0` is only run a single time at the very start of the function call. `j = i+1` is more complicated--in the best case when `A[0] == A[1]`, it only runs a single time (convince yourself of this fact!). In the worst case, `j` is initialized once for each value that `i` takes on in the outer loop, so it is initialized a total of 10000 times.

As we dive further into the loop, it gets progressively less feasible to calculate the exact counts. We will show later on that the exact numbers do not matter that much.

To summarize technique 2A, we have solved the issue of *machine independence*. Differences in machines do not affect the total of operations run (usually). However, it is tedious to compute all the counts for each operation. In addition, our chosen input size is arbitrary, and we do not know how much time it actually takes to run.

Technique 2B

We can resolve the issue of choosing input size by calculating the *symbolic count* instead, which means that we calculate our counts in terms of the input `N`. Using this technique,

we can update our table to include the symbolic counts:

Operation	Symbolic Count	Count (for N=10000)
i = 0	1	1
j = i+1	1 to N	1 (in the best case) to 10000 (in the worst case)
<	2 to $(N^2 + 3N + 2)/2$	2 to 50,015,001
+= 1	0 to $(N^2 + N)/2$	0 to 50,005,000
==	1 to $(N^2 - N)/2$	1 to 49,995,000
array accesses	2 to N^2-N	2 to 99,990,000

Using symbolic counts allows us to see how the algorithms *scales* with input size. However, it is now even more tedious to calculate! It also still does not tell the actual time.

Previous
13.1 An Introduction to Asymptotic Analysis

Next
13.3 Checkpoint: An Exercise

Last updated 1 year ago