CSE 12
Algorithm Time Cost Measurement

- Algorithm analysis vs. measurement
- Timing an algorithm
- Average and standard deviation
- Improving measurement accuracy

Introduction

- These three characteristics of programs are important:
  - **robustness**: a program's ability to spot exceptional conditions and deal with them or shutdown gracefully
  - **correctness**: does the program do what it is "supposed to" do?
  - **efficiency**: all programs use resources (time, space, and energy); how can we measure efficiency so that we can compare algorithms?

Analysis and Measurement

An algorithm's performance can be described by:

- ***time*** complexity or cost – how long it takes to execute. In general, less time is better!
- ***space*** complexity or cost – how much computer memory it uses. In general, less space is better!
- ***energy*** complexity or cost – how much energy uses. In general, less energy is better!

- Costs are usually given *as functions of the size of the input* to the algorithm
- A big instance of the problem will probably take more resources to solve than a small one, but how much more?

Figuring algorithm costs

- For a given algorithm, we would like to know the following as functions of *n*, the size of the problem:
  - T(*n*) , the time cost of solving the problem
  - S(*n*) , the space cost of solving the problem
  - E(*n*) , the energy cost of solving the problem
- Two approaches:
  - We can *analyze* the written algorithm
  - Or we could implement the algorithm and run it and *measure* the time, memory, and energy usage

Asymptotic algorithm analysis

- Asymptotic algorithm analysis (counting statements executed, and writing the result as a simple function using big-O, big-omega, or big-theta notation) is elegant and useful
- Consider the important cost function classes (on the next slide)
- If algorithm A has big-theta time cost of one column, and algorithm B has big-theta time cost of a column to the right of that one, then:

  For sufficiently large problem sizes *n*, algorithm B **will** take more time than algorithm A

Some Common Cost Function Classes

$\log_2 n$	n	$n \log_2 n$	n^2	2^n
1	2	2	4	4
2	4	8	16	16
3	8	24	64	256
4	16	64	256	65,536
5	32	160	1,024	4,294,967,296
6	64	384	4,096	1.84×10^{19}
7	128	896	16,384	3.40×10^{38}
8	256	2,048	65,536	1.16×10^{77}
9	512	4,608	262,144	1.34×10^{154}
10	1,024	10,240	1,048,576	1.80×10^{308}

Algorithm analysis vs. measurement

- Asymptotic algorithm analysis is elegant, and it's important to know how to do it... but it is an abstraction, so it doesn't tell the full story.
- If two algorithms' time cost functions are in the same cost function class, they may take very different times to solve a problem of the same size
- Example: linear search in an array of `int`s and an array of `Integer` objects are both $\Theta(n)$, but in reality…

- So it can be important also to consider *algorithm measurement*, also known as *benchmarking*

Algorithm Measurement

- The basic idea is simple:
  - Implement the algorithm
  - Measure the time, space, or energy it takes for the implementation to run, on inputs of different sizes
  - Plot the measurements and characterize T(*n*), S(*n*), or E(*n*)

- The data should give you a good idea of the actual cost function of the algorithm in practice

- But it can be tricky to get good measurements...

Basic Algorithm Time Cost Measurement

Pseudocode: for timing a data structure algorithm

1. *for problem size N = min,...max*
2. *initialize the data structure*
3. *get the current (starting) time*
4. *run the algorithm on problem size N*
5. *get the current (finish) time*
6. *elapsed time = finish time – start time*
7. *output elapsed time on problem size N*

Getting start and finish time

- Most programming languages have a way of accessing the operating system clock
- Java has two static methods in the System class:

```
/** Returns the current time in milliseconds. */
static long System.currentTimeMillis()

/** Returns the current value of the most precise
     available system timer, in nanoseconds */
static long System.nanoTime()
```

- If the algorithm can take less than a millisecond to run, you should use System.nanoTime() !

Accuracy of algorithm time measurement

- On a typical computer, the algorithm you are trying to benchmark isn't the only program running
- Operating system tasks, other user processes, even other threads within your Java program itself can be using processor resources
- Operating system caching strategies, and just-in-time (JIT) compilation can be affecting runtime
- Result: measured elapsed time will be 'noisy', including other factors than actual algorithm running time

Improving accuracy: averaging

- One way to reduce the effect of noise in measurements is to *average* several measurements
- This means: running the algorithm several times, totaling the elapsed times of the runs, and dividing the total time by the number of runs
- Result: average measured elapsed time over several runs will, hopefully, be less 'noisy' than the measured time of a single run

Algorithm Measurement with averaging

Pseudocode: for timing with averaging

1. *for problem size N = min,...,max*
2. *initialize the data structure; totaltime = 0*
3. *for K runs*
4. *get the current (starting) time*
5. *run the algorithm on problem size N*
6. *get the current (finish) time*
7. *totaltime = totaltime + (finish time – start time)*
8. *output average time on problem size N = total time / K*

Checking quality: standard deviation

- Averaging *can* improve the quality of measurements. But is it really an improvement?
- In statistics, *standard deviation* is sometimes used to indicate the quality of the average of a set of measurements
- Standard deviation says how 'spread out' the measurements are around the average

Standard deviation

- Standard deviation says how 'spread out' the measurements are
  - A high standard deviation means the measurements were very spread out, and so were probably very influenced by noise
  - A low standard deviation means the measurements were close to each other, and so were probably less influenced by noise
- So, standard deviation can be an indicator of how confident you should be that the average of the measurements reflects a 'true' value

Computing standard deviation

- Standard deviation of a set of measurements is defined as:
  - the square root of the average of the squared difference between the measurements and the average of the measurements

- Suppose you have a set of *K* measurements

$$\{x_i\}, \quad i = 1, \ldots, K$$

- Then the *average* or *mean* of the measurements is

$$\mu = \frac{1}{K} \sum_{i=1}^{K} x_i$$

Computing standard deviation

- The *variance* of the measurements is

$$\sigma^2 = \frac{1}{K}\sum_{i=1}^{K}(x_i - \mu)^2$$

- And the *standard deviation* of the measurements is the square root of the variance:

$$\sigma = \sqrt{\sigma^2}$$

- This suggests first iterating through the measurements to compute the mean, and then iterating through them again to compute the variance (and then the standard deviation). But there is another approach…

Computing standard deviation

- Rewrite the formula for the variance:

$$\sigma^2 = \frac{1}{K}\sum_{i=1}^{K}(x_i - \mu)^2$$

$$\sigma^2 = \frac{1}{K}\sum_{i=1}^{K}\left(x_i^2 - 2\mu x_i + \mu^2\right)$$

$$\sigma^2 = \frac{1}{K}\sum_{i=1}^{K}x_i^2 - \frac{1}{K}\sum_{i=1}^{K}2\mu x_i + \frac{1}{K}\sum_{i=1}^{K}\mu^2$$

Computing standard deviation

$$\sigma^2 = \frac{1}{K}\sum_{i=1}^{K} x_i^2 - 2\mu\left(\frac{1}{K}\sum_{i=1}^{K} x_i\right) + \frac{1}{K}\sum_{i=1}^{K} \mu^2$$

$$\sigma^2 = \frac{1}{K}\sum_{i=1}^{K} x_i^2 - 2\mu^2 + \frac{1}{K}\sum_{i=1}^{K} \mu^2$$

$$\sigma^2 = \frac{1}{K}\sum_{i=1}^{K} x_i^2 - 2\mu^2 + \mu^2$$

$$\sigma^2 = \frac{1}{K}\sum_{i=1}^{K} x_i^2 - \mu^2$$

Computing standard deviation

- And so the standard deviation can be computed as

$$\sigma = \sqrt{\left(\frac{1}{K}\sum_{i=1}^{K} x_i^2\right) - \left(\mu^2\right)}$$

- Another way to say this:
  - let A be the average of the measurements
  - let V be the average of the square of the measurements
  - Then, standard deviation can be computed as: sqrt (V - A^2)

Algorithm Measurement with avgerage and standard deviation

Pseudocode: for timing with averaging and standard deviation

1. *for problem size N = min,...,max*
2. *initialize the data structure; totaltime = 0; totalsqtime = 0*
3. *for K runs*
4. *get the current (starting) time*
5. *run the algorithm on problem size N*
6. *get the current (finish) time*
7. *et = finish time – start time*
8. *totaltime += et*
9. *totalsqtime += et * et*
10. *avg = totaltime / K*
11. *standard deviation = sqrt (totalsqtime / K – avg * avg)*

Improving accuracy: reducing other demands on the processor

- The accuracy of algorithm time cost measurement will be improved if the number of concurrent demands on the machine can be reduced
- Running on a single-user machine, not running other programs while benchmarking, etc.
- But also: reduce the number of *threads* running concurrently in your Java virtual machine...

Improving accuracy: running the Java garbage collector

- Every JVM has a *garbage collector* thread that runs if the JVM is low on memory
- The gc searches for objects that can no longer be accessed by your program, and reclaims the memory they occupy
  - This is computationally intensive!
- So, try to get the gc to run when you are *not* timing your algorithm!

- You suggest that the JVM run the garbage collector with:

```
System.gc()
```

Algorithm Measurement: A Code Fragment

```
for(int N = startN; i<=endN; N += stepN) {
 intArray = new int[N];
 for ( int j = 0, i = N - 1; i >= 0; j++, i-- )
   intArray[j] = i;

 // find the largest value in the int array
 for ( int i = 0; i < NUM_TRIALS; i++ ) {
   int largest = intArray[0];
   start = System.currentTimeMillis();
   for ( int j = 1; j < SIZE; j++ )
      if ( intArray[j] > largest )
         largest = intArray[j];
   finish = System.currentTimeMillis();
   intArrayTimeTotal += finish - start;
 }

 // force cleanup to prevent it happening while
 // looking for the largest in the Integer array
 intArray = null;  // make the array garbage
 System.gc();      // invoke the garbage collector
}
```

Get the start time

Get the finish time

Cleanup between timings

Improving accuracy: disabling the JIT compiler

- Every JVM has a *Just-in-time (JIT) compiler* that looks at patterns of Java bytecode execution and can compile them to native code at runtime to improve performance
- This means that if you are benchmarking the same operation repeatedly (for different N, or different trials with the same N), the time to do an operation can decrease the more often it is done!
- If you want to run your program with JIT disabled, do:
- `java -Xint Prog`
- This will make your program slower, but you may get better-looking measurements

Explain these results...

Timings for `findMax()` on an array of `int`s and an array of `Integer`s (times are in milliseconds)

n	array of `int`	array of `Integer`
800,000	2.314	4.329
4,000,000	11.363	21.739
8,000,000	22.727	42.958

- From these measurements, what is the likely big-O time cost of findMax() on array of int?_____ on array of Integer?_____
- Why would array of Integer take more time?

Next time

- **The Collection and Iterable interfaces**
- **The Iterator Design Pattern**
- **The Iterator interface**
- **Defining classes that implement the Iterator interface**
- **The concurrent modification problem**

Reading: Gray, Ch 4