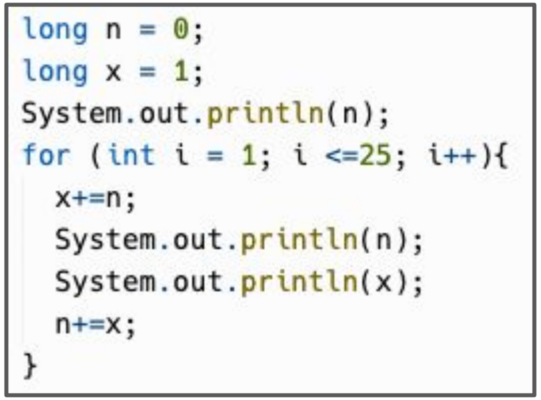
10/19/22

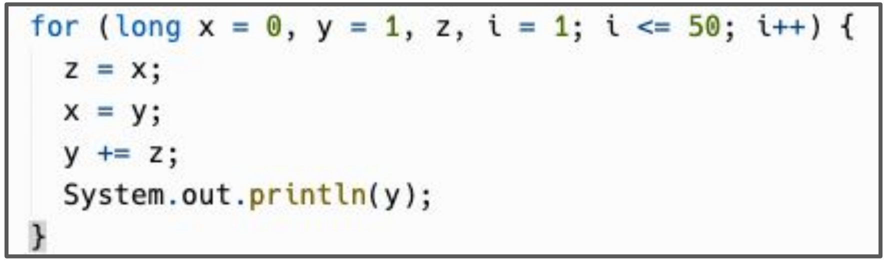
QUIZ

ON

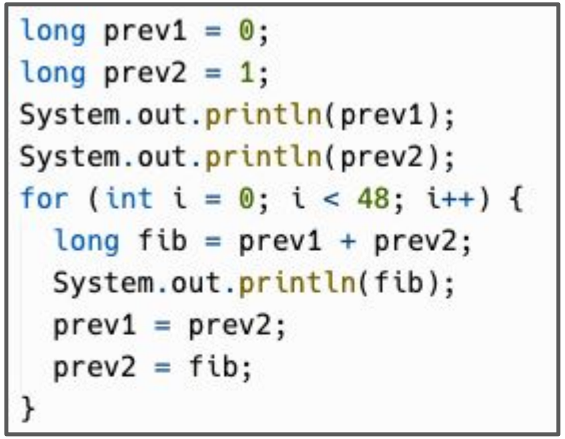
FRIDAY



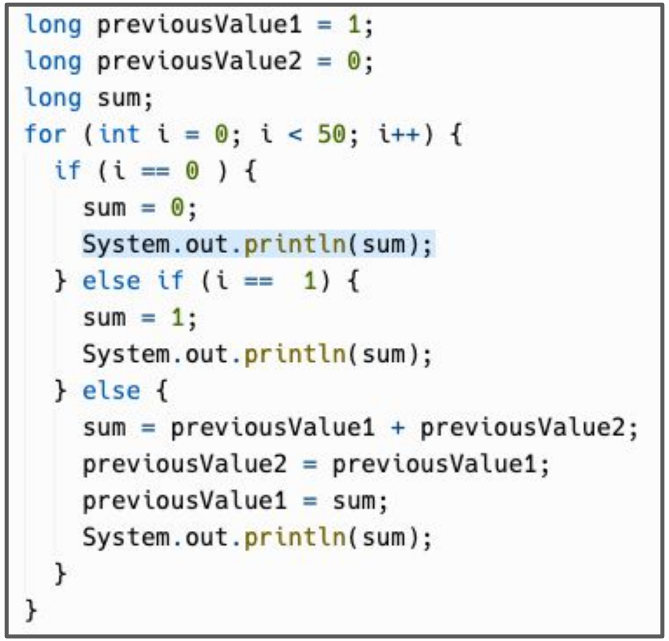
```java
long n = 0;
long x = 1;
System.out.println(n);
for (int i = 1; i <=25; i++){
  x+=n;
  System.out.println(n);
  System.out.println(x);
  n+=x;
}
```



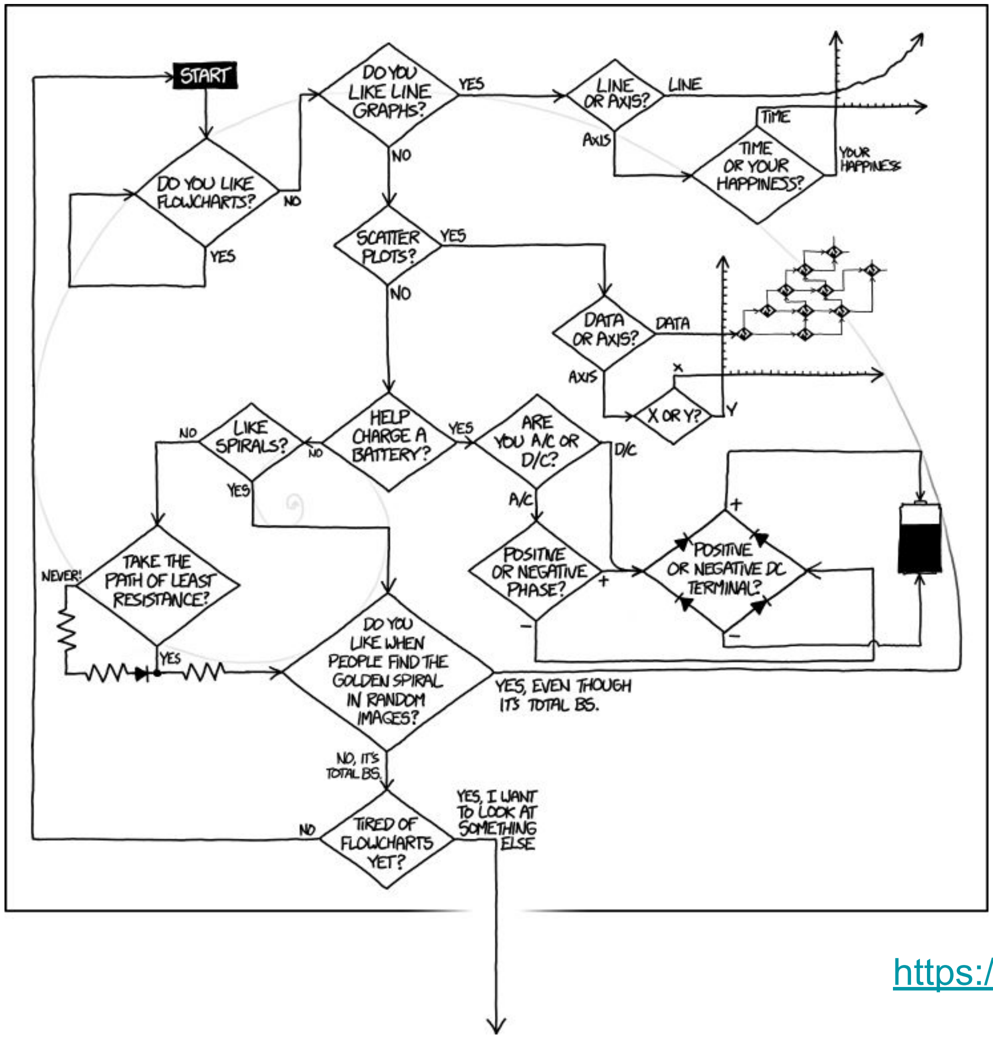
```java
for (long x = 0, y = 1, z, i = 1; i <= 50; i++) {
  z = x;
  x = y;
  y += z;
  System.out.println(y);
}
```



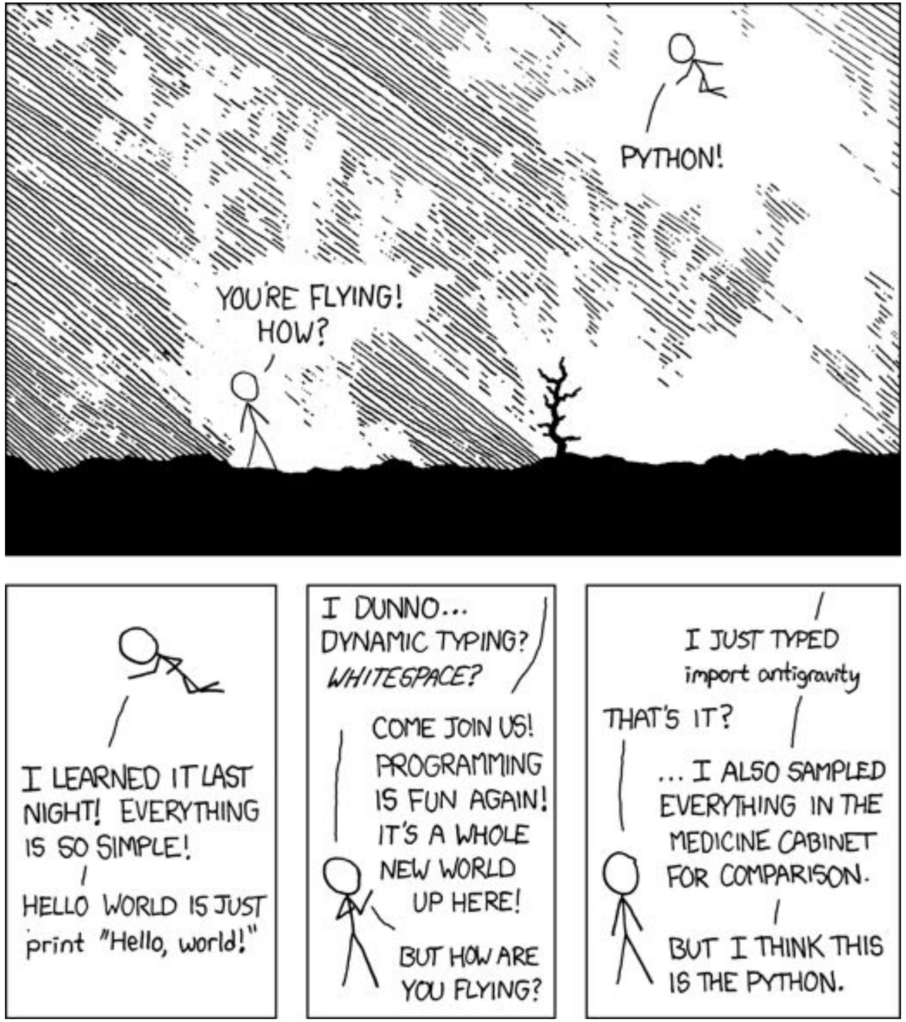
```java
long prev1 = 0;
long prev2 = 1;
System.out.println(prev1);
System.out.println(prev2);
for (int i = 0; i < 48; i++) {
  long fib = prev1 + prev2;
  System.out.println(fib);
  prev1 = prev2;
  prev2 = fib;
}
```



```java
long previousValue1 = 1;
long previousValue2 = 0;
long sum;
for (int i = 0; i < 50; i++) {
  if (i == 0 ) {
    sum = 0;
    System.out.println(sum);
  } else if (i ==  1) {
    sum = 1;
    System.out.println(sum);
  } else {
    sum = previousValue1 + previousValue2;
    previousValue2 = previousValue1;
    previousValue1 = sum;
    System.out.println(sum);
  }
}
```



https://xkcd.com/1488/


YOU'RE FLYING! HOW?
PYTHON!
I LEARNED IT LAST NIGHT! EVERYTHING IS SO SIMPLE!
HELLO WORLD IS JUST print "Hello, world!"
I DUNNO... DYNAMIC TYPING? WHITESPACE?
COME JOIN US! PROGRAMMING IS FUN AGAIN! IT'S A WHOLE NEW WORLD UP HERE!
BUT HOW ARE YOU FLYING?
I JUST TYPED import antigravity
THAT'S IT?
... I ALSO SAMPLED EVERYTHING IN THE MEDICINE CABINET FOR COMPARISON.
BUT I THINK THIS IS THE PYTHON.

https://xkcd.com/353/

$$\lim_{n\to\infty} \frac{F_{n+1}}{F_n} = \varphi.$$

$$\frac{1}{1} = 1.000000000000000$$
$$\frac{2}{1} = 2.000000000000000$$
$$\frac{3}{2} = 1.500000000000000$$
$$\frac{5}{3} = 1.666666666666667$$
$$\frac{8}{5} = 1.600000000000000$$
$$\frac{13}{8} = 1.625000000000000$$
$$\frac{21}{13} = 1.615384615384615$$
$$\frac{34}{21} = 1.619047619047619$$
$$\frac{55}{34} = 1.617647058823529$$
$$\frac{89}{55} = 1.618181818181818$$
$$\frac{144}{89} = 1.617977528089888$$
$$\frac{233}{144} = 1.618055555555556$$
$$\frac{377}{233} = 1.618025751072961$$
$$\frac{610}{377} = 1.618037135278515$$
$$\frac{987}{610} = 1.618032786885246$$
$$\frac{1597}{987} = 1.618034447821682$$
$$\frac{2584}{1597} = 1.618033813400125$$
$$\frac{4181}{2584} = 1.618034055727554$$
$$\frac{6765}{4181} = 1.618033963166706$$
$$\frac{10946}{6765} = 1.618033998521803$$
$$\frac{17711}{10946} = 1.618033985017358$$
$$\frac{28657}{17711} = 1.618033990175597$$
$$\frac{46368}{28657} = 1.618033988205325$$
$$\frac{75025}{46368} = 1.618033988957902$$
$$\frac{121393}{75025} = 1.618033988670443$$
$$\frac{196418}{121393} = 1.618033988780243$$
$$\frac{317811}{196418} = 1.618033988738303$$
$$\frac{514229}{317811} = 1.618033988754323$$
$$\frac{832040}{514229} = 1.618033988748204$$
$$\frac{1346269}{832040} = 1.618033988750541$$
$$\frac{2178309}{1346269} = 1.618033988749648$$
$$\frac{3524578}{2178309} = 1.618033988749989$$
$$\frac{5702887}{3524578} = 1.618033988749859$$
$$\frac{9227465}{5702887} = 1.618033988749909$$
$$\frac{14930352}{9227465} = 1.618033988749890$$
$$\frac{24157817}{14930352} = 1.618033988749897$$
$$\frac{39088169}{24157817} = 1.618033988749894$$
$$\frac{63245986}{39088169} = 1.618033988749895$$
$$\frac{102334155}{63245986} = 1.618033988749895$$
$$\frac{165580141}{102334155} = 1.618033988749895$$
$$\frac{267914296}{165580141} = 1.618033988749895$$

φ = 1.

```
6180339887 4989484820 4586834365 6381177203 0917980576 2862135448
6227052604 6281890244 9707207204 1893911374 8475408807 5386891752
1266338622 2353693179 3180060766 7263544333 8908659593 9582905638
3226613199 2829026788 0675208766 8925017116 9620703222 1043216269
5486262963 1361443814 9758701220 3408058879 5445474924 6185695364
8644492410 4432077134 4947049565 8467885098 7433944221 2544877066
4780915884 6074998871 2400765217 0575179788 3416625624 9407589069
7040002812 1042762177 1117778053 1531714101 1704666599 1466979873
1761356006 7087480710 1317952368 9427521948 4353056783 0022878569
9782977834 7845878228 9110976250 0302696156 1700250464 3382437764
8610283831 2683303724 2926752631 1653392473 1671112115 8818638513
3162038400 5222165791 2866752946 5490681131 7159934323 5973494985
0904094762 1322298101 7261070596 1164562990 9816290555 2085247903
5240602017 2799747175 3427775927 7862561943 2082750513 1218156285
5122248093 9471234145 1702237358 0577278616 0086883829 5230459264
7878017889 9219902707 7690389532 1968198615 1437803149 9741106926
0886742962 2675756052 3172777520 3536139362 1076738937 6455606060
5921658946 6759551900 4005559089 5022953094 2312482355 2122124154
4400647034 0565734797 6639723949 4994658457 8873039623 0903750339
9385621024 2369025138 6804145779 9569812244 5747178034 1731264532
2041639723 2134044449 4873023154 1767689375 2103068737 8803441700
9395440962 7955898678 7232095124 2689355730 9704509595 6844017555
1988192180 2064052905 5189349475 9260073485 2282101088 1946445442
2231889131 9294689622 0023014437 7026992300 7803085261 1807545192
8877050210 9684249362 7135925187 6077788466 5836150238 9134933331
2231053392 3213624319 2637289106 7050339928 2265263556 2090297986
4247275977 2565508615 4875435748 2647181414 5127000602 3890162077
7322449943 5308899909 5016803281 1219432048 1964387675 8633147985
7191139781 5397807476 1507722117 5082694586 3932045652 0989698555
```

Series approximations

Deep down, computers are built out of circuits that do basic math operations like add, subtract, multiply, divide. To do a calculation like sin(x) or cos(x), computers use series approximations such as Taylor Series. There are series approximations that can be derived for many important mathematical formulas. Even pi itself has series approximations.

$$\pi = 4\left(1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7}\dots\right)$$

$$sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \dots$$

$$cos(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \dots$$

Runtime Analysis

Chapter 4.5

What makes a good algorithm?

From CodeHS:

What makes a good algorithm?

- Correctness
- Easy to understand by someone else
- Efficiency (run time and memory requirements)

Correctness refers to whether the algorithm solves the given problem.

Easy to understand can be a function of the complexity of the algorithm design, as well as how the code is laid out, use of appropriate variable names, and the use of comments.

Efficiency can be looked at in several ways, which we will explore in this section.

Runtime analysis

How long an algorithm takes to execute (run) is called its **running time** or **runtime**.

Runtime analysis is the process of understanding how an algorithm or complete program will perform when it is run.

It includes how long the program takes to run, as well as other factors like how much memory is consumed.

Timing program execution

To understand how long a program takes to execute, can't we just… time it?

Every computer has multiple internal "clocks" that keep track of time (usually, the number of fractions of a second that have elapsed since a given base date).

In Java, people often measure the delta in `System.currentTimeMillis()` which returns the number of milliseconds since the "Unix Epoch" (January 1, 1970 00:00:00 in UTC time zone). This is **wall clock time**… if you adjust the date/time on your computer, the value will change.

`System.nanoTime()` is a newer method which returns nanoseconds since an arbitrary point in time. This timer is independent of the "wall clock" … it will continue increasing even if you futz with your computer's date and time.

Either may be acceptable, depending on your benchmark scenario.

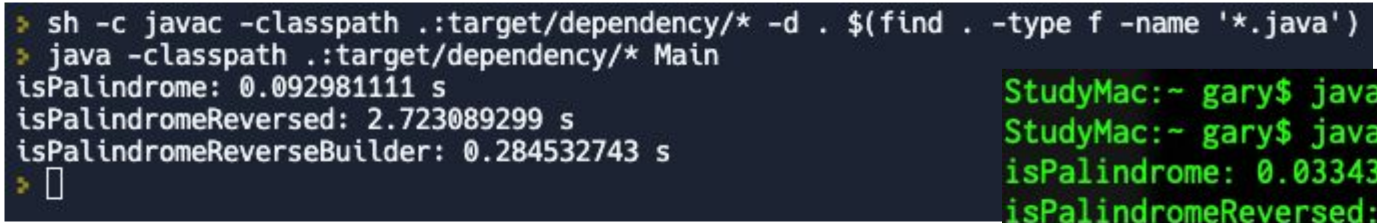
Why not just time things?

Different computers will give you different results, so timing benchmarks are only valid on the same hardware.

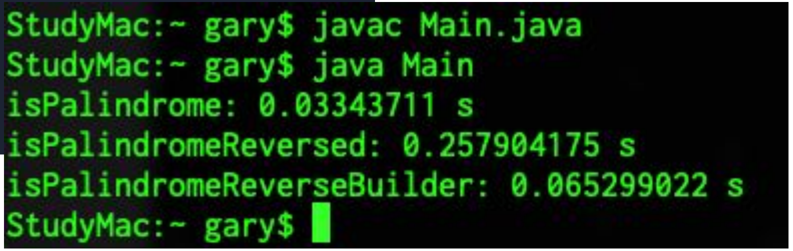
A lot of other stuff can be happening in other processes on a modern computer.

Different programming languages have different performance characteristics. (A fast algorithm in a "slow" language might be slower for some inputs than a slow algorithm in a "fast" language.)

Even the same programming language may behave differently in different environments.



```
> sh -c javac -classpath .:target/dependency/* -d . $(find . -type f -name '*.java')
> java -classpath .:target/dependency/* Main
isPalindrome: 0.092981111 s
isPalindromeReversed: 2.723089299 s
isPalindromeReverseBuilder: 0.284532743 s
>
```



```
StudyMac:~ gary$ javac Main.java
StudyMac:~ gary$ java Main
isPalindrome: 0.03343711 s
isPalindromeReversed: 0.257904175 s
isPalindromeReverseBuilder: 0.065299022 s
StudyMac:~ gary$
```

Statement Execution Count

An alternative to timing a program segment is **statement counting**.

The number of times a given statement in a program gets executed is independent of the computer on which the program is run and is probably close for programs coded in closely related languages.

Statement Execution Count

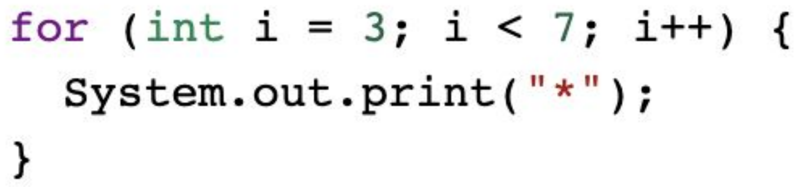
Getting an absolute number of statements executed is tricky, and some of the things we may want to count are actually just expressions, not full statements.

Here, one way we might do it is to "point" it as follows:

int i = 3; counts as 1 point

This part repeats for i=3, 4, 5, 6:
i < 7 counts as 1 point
System.out.print("*") counts as 1 point
i++ counts as 1 point

1 + 3 * 4 = 13 "statements" total



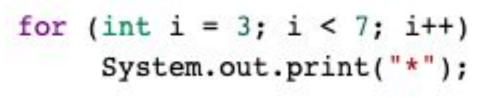
```java
for (int i = 3; i < 7; i++) {
  System.out.print("*");
}
```

There could be different counting methods here. On the AP exam, you'll probably be asked something more like: How many times is System.out.print called in this loop? That is, count how many times a **specific** statement is executed.

Loop Execution Count

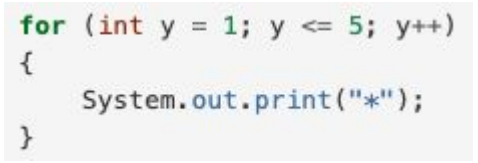
You can use a trace table to figure out how many times a loop executes. But, you may be able to shortcut that just looking at it.

With a < condition, the number of iterations is (ending value - starting value) = 7 - 3 = 4 iterations.



```
for (int i = 3; i < 7; i++)
        System.out.print("*");
```

When a <= is involved, the number of iterations is (ending value - starting value + 1) = 5 - 1 +1 = 5 iterations.

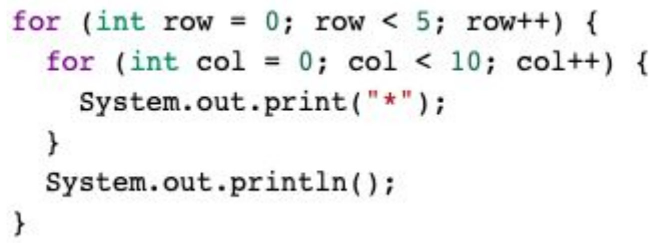


```
for (int y = 1; y <= 5; y++)
{
    System.out.print("*");
}
```

Both of these shortcuts assume the increment expression is just adding 1 to the counter.

Loop Execution Count

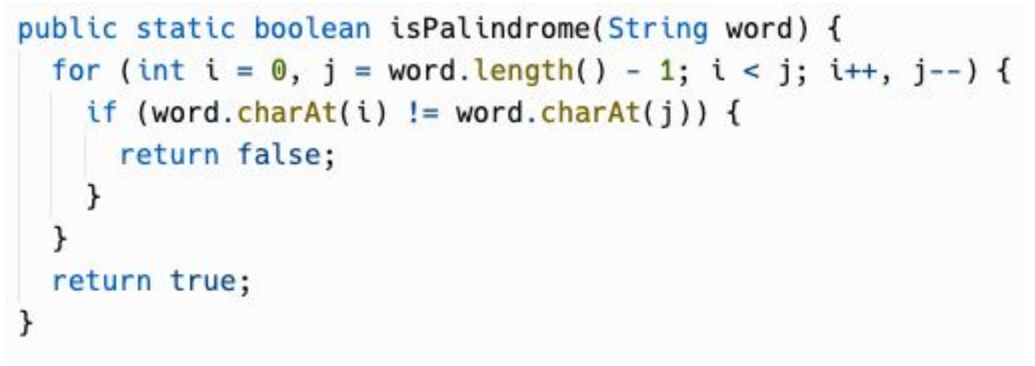
For nested loops: The number of times a nested for loop body is executed is the number of times the outer loop runs multiplied by the number of times the inner loop runs (outer loop runs * inner loop runs).



```java
for (int row = 0; row < 5; row++) {
  for (int col = 0; col < 10; col++) {
    System.out.print("*");
  }
  System.out.println();
}
```

Best and worst case

Let's count the loop executions in this implementation of isPalindrome.



```java
public static boolean isPalindrome(String word) {
  for (int i = 0, j = word.length() - 1; i < j; i++, j--) {
    if (word.charAt(i) != word.charAt(j)) {
      return false;
    }
  }
  return true;
}
```

The loop execution count depends on the input.

The **best case** is that the first and last letters do not match, which would be a loop execution count of 0 (but a statement execution count of 3, for the init expression, condition check, and return statement.)

The **worst case** is… if the string is really a palindrome, and we have to check every pair of characters! That would be word.length() / 2 loop executions.

What about the **average case**?

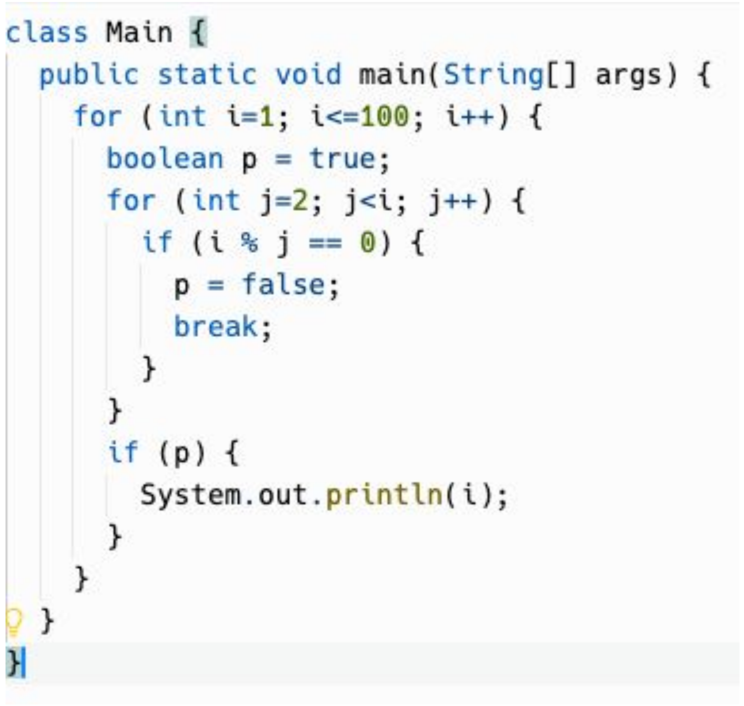
A more complicated nested loop

The outer loop executes 100 times. But the inner loop executes a different number of times for each i, and breaks out when it finds a number is not prime.

The inner loop executes at most once for i=3, twice for i=4, 3x for i=5, 4x for i=6. So the inner loop executes at most 3+4+5+6+...+98 times. We could calculate worst case time using this formula for 1+2+...+n and subtract 1+2:

$$\sum_{k=1}^{n} k = \frac{n(n+1)}{2}$$

But usually, we are not concerned with an exact count, but with getting a solid estimate. It's enough to determine what the loop execution time is **proportional to**.



```
class Main {
  public static void main(String[] args) {
    for (int i=1; i<=100; i++) {
      boolean p = true;
      for (int j=2; j<i; j++) {
        if (i % j == 0) {
          p = false;
          break;
        }
      }
      if (p) {
        System.out.println(i);
      }
    }
  }
}
```

Running time is proportional to…

Let N=100. (We use capitals for this a lot, for some reason.)

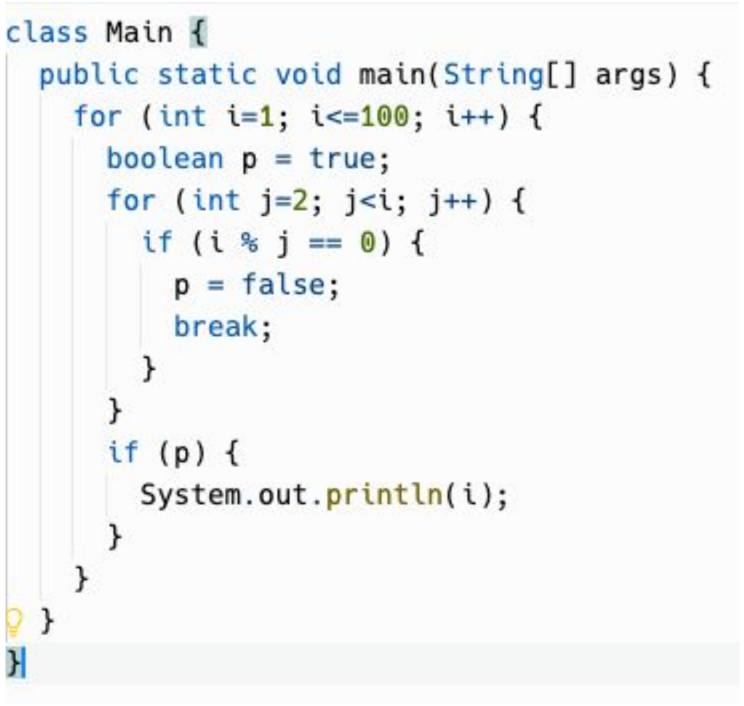
The outer loop executes N times.

The inner loop executes a variable number of times, but it is bounded by N.

The loop execution count isn't exactly N^2… it's less. But it is proportional to N^2; it will grow in concert with N^2, even if it doesn't follow an exact linear relationship with it.

The running time of this loop is **proportional to** N^2.

An algorithm that runs in N^2 time is said to run in **quadratic time**.

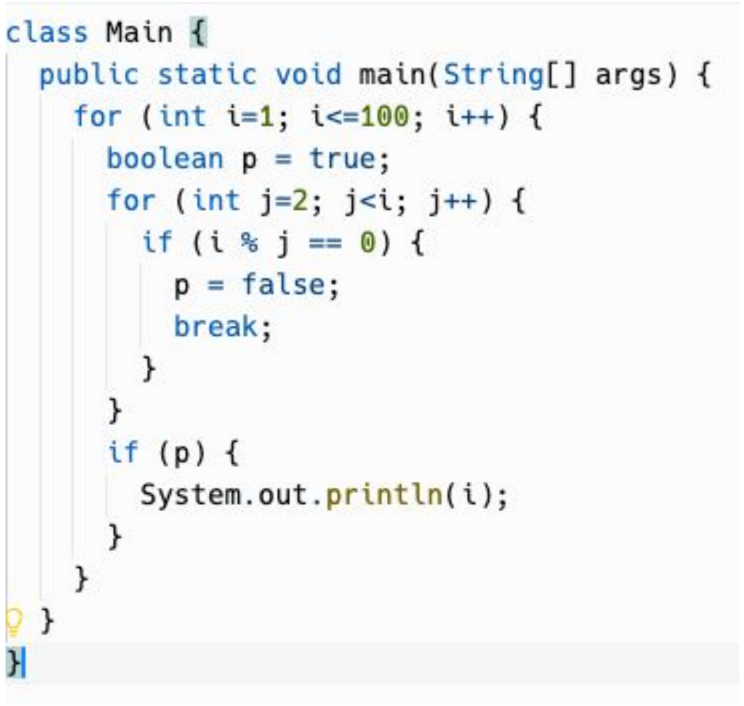


```
class Main {
  public static void main(String[] args) {
    for (int i=1; i<=100; i++) {
      boolean p = true;
      for (int j=2; j<i; j++) {
        if (i % j == 0) {
          p = false;
          break;
        }
      }
      if (p) {
        System.out.println(i);
      }
    }
  }
}
```

Running time is proportional to…

The concept of **proportional to** also removes our concerns about how fast a particular computer is, or how fast a particular programming language is.

We're analyzing the performance of the algorithm in the abstract world of mathematics, not in the concrete world of actual silicon.



```
class Main {
  public static void main(String[] args) {
    for (int i=1; i<=100; i++) {
      boolean p = true;
      for (int j=2; j<i; j++) {
        if (i % j == 0) {
          p = false;
          break;
        }
      }
      if (p) {
        System.out.println(i);
      }
    }
  }
}
```

Linear time

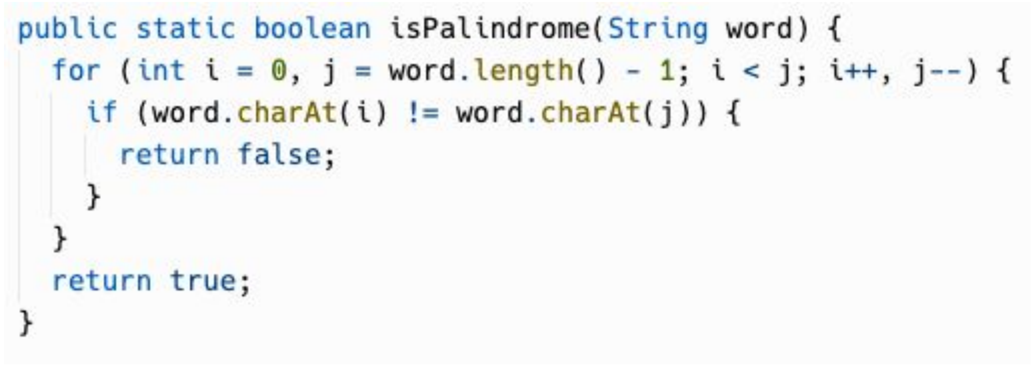
What is our isPalindrome implementation proportional to?

The loop execution count depends on the input, but it is **bounded** by the length of the input. If the input is length N, the loop will not execute more than N/2 times.

We can ignore the N/2 factor since we're talking about proportionality.

So, isPalindrome running time is proportional to N, where N is the length of the string input.

An algorithm with running time proportional to N is said to run in **linear time**.



```java
public static boolean isPalindrome(String word) {
  for (int i = 0, j = word.length() - 1; i < j; i++, j--) {
    if (word.charAt(i) != word.charAt(j)) {
      return false;
    }
  }
  return true;
}
```

What about the **average case**?

Constant time

Some algorithms have a running time that isn't a function of its input parameters at all.

Such a function can be said to run in **constant time**.

The running time may not be exactly the same every time, like some exact drumbeat. But it is not a function of the input's length in any way.

Computational Complexity

In computer science, the computational complexity or simply complexity of an algorithm is the amount of resources required to run it.

Time complexity is the amount of time that an algorithm takes to run, but in an abstract sense of how many operations need to be executed, not a quantity of seconds or milliseconds. The usual units of time (seconds, minutes etc.) are not used because they are too dependent on the choice of a specific computer and on the evolution of technology.

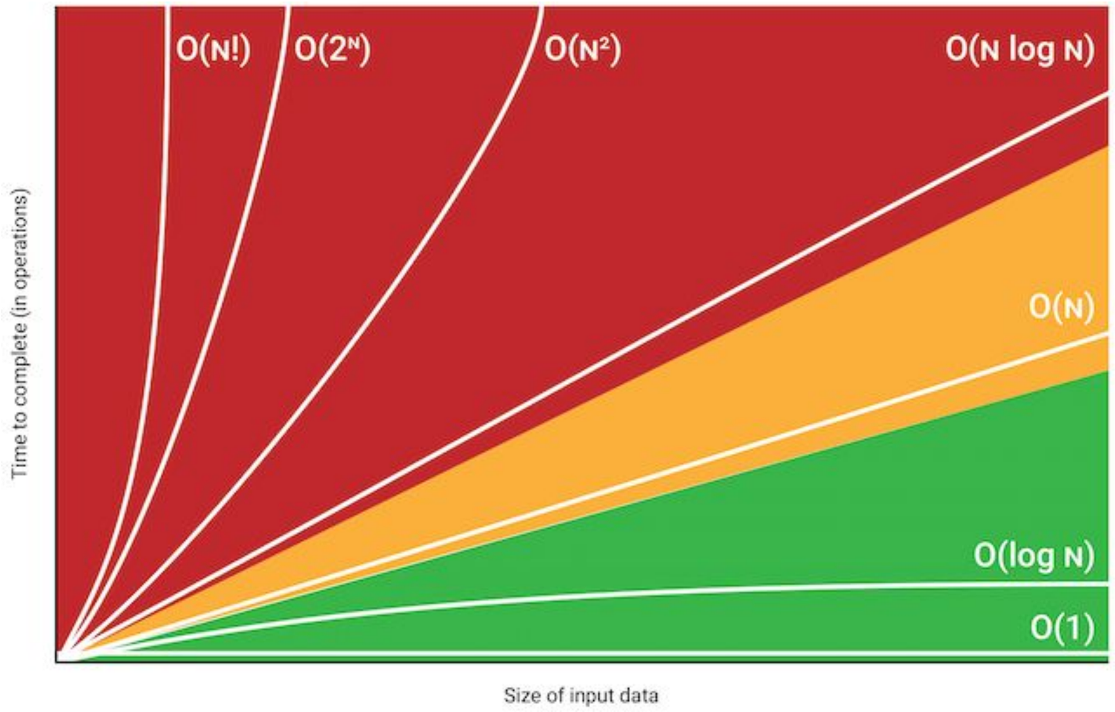
Space complexity is how much memory an algorithm consumes. A lot of our algorithms so far allocate no memory, or allocate one or two things. This is constant space complexity. Constant space complexity means that the amount of space that your algorithm uses is independent of the input parameters. Linear space complexity means you probably allocated an array of size N, or for, quadratic, N^2, etc.

Big O notation

Big O notation is a common way of stating the **upper bound** of an algorithm's time or space complexity.

You might calculate that an algorithm takes at most 5N^2+8N+4 operations.

As N gets very large, the N^2 dominates everything else, and we take out the constant factor, so we say the algorithm is O(N^2).



Big O notation

The time complexity and space complexity of algorithms are often expressed using "Big O notation"

O(1) = Constant time

O(N) = Linear time

O(N^2) = Quadratic time

You can think of it as, the algorithm's upper bound (maximum) running time or memory usage is **proportional to** 1, N, N^2, etc.

There are other common ones, such as O(N^3), O(log N), O(N log N). We'll be seeing algorithms with some of these time and space complexities later in the course.

(There is also Big Omega and Big Theta notation but our brains are full enough for one day.)

"Real" time and space

When we talk about time complexity and space complexity, we eliminate constant factors so that advances in computer hardware, speed of programming languages isn't a factor.

In practice, real time taken by an algorithm does matter, as well as how much memory it consumes, too.

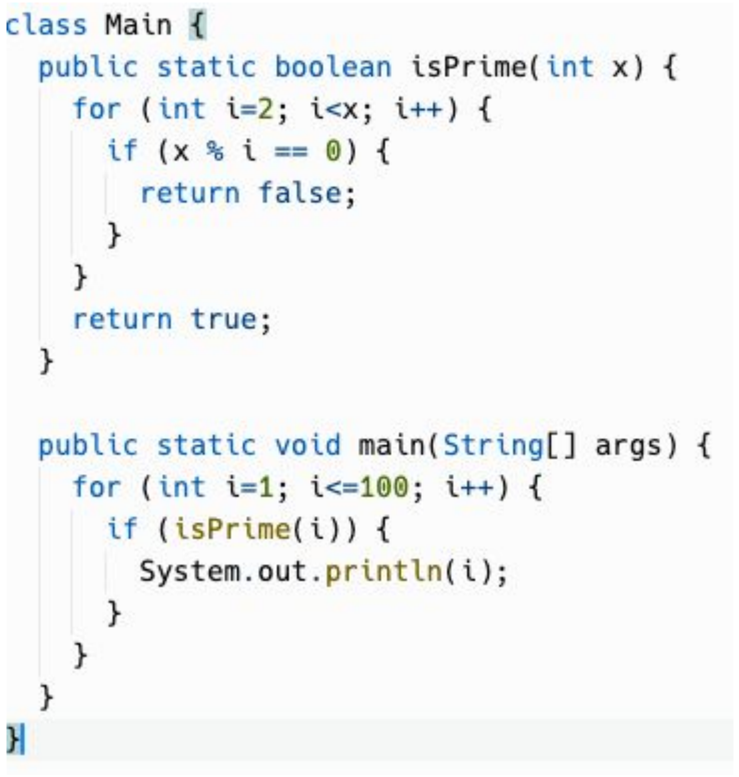
If you make an algorithm run 2X faster in wall clock time, you've saved your company money, or made your scientific computation give an answer sooner, etc.

Same if you reduce your program's memory use from 1000 bytes per record to 500 bytes per record.

Finding Prime Numbers

Remember our prime number finder from last session, and the assertion that a 2000-year old algorithm can do it more efficiently.

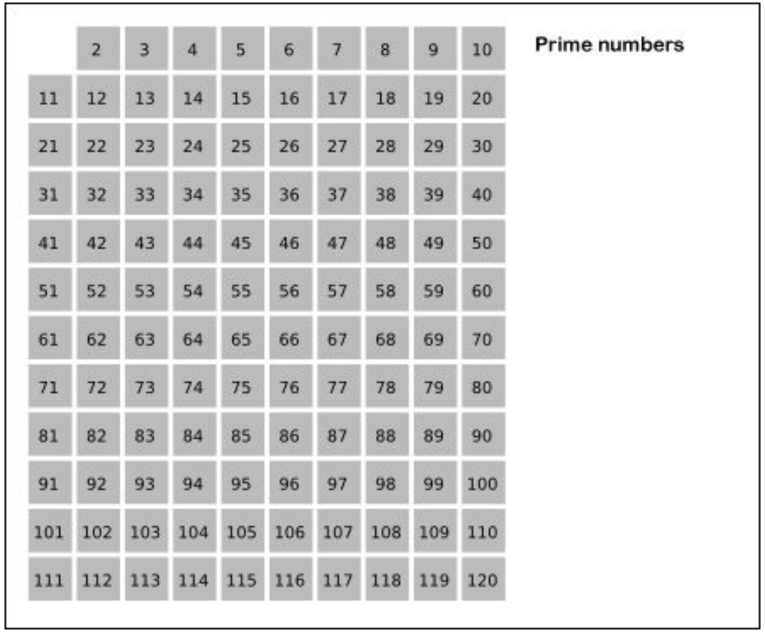
How much more efficiently?



```
class Main {
  public static boolean isPrime(int x) {
    for (int i=2; i<x; i++) {
      if (x % i == 0) {
        return false;
      }
    }
    return true;
  }

  public static void main(String[] args) {
    for (int i=1; i<=100; i++) {
      if (isPrime(i)) {
        System.out.println(i);
      }
    }
  }
}
```

Sieve of Eratosthenes vs. My Naive Prime Finder

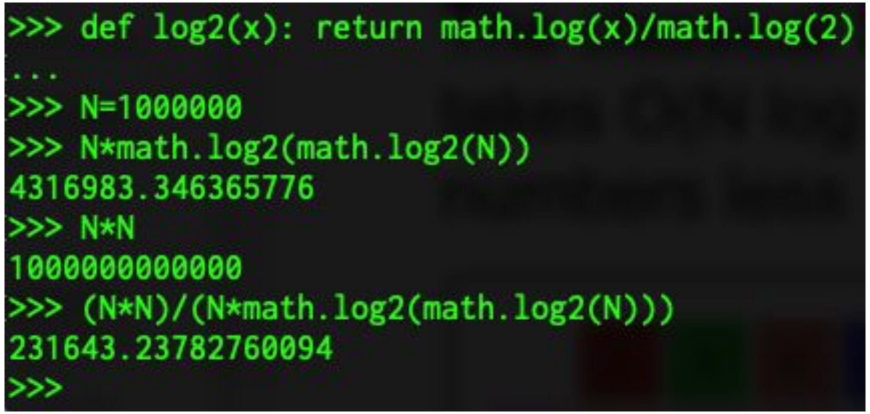
The classical Sieve of Eratosthenes algorithm takes O(N log (log N)) time to find all prime numbers less than N.



	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100
101	102	103	104	105	106	107	108	109	110
111	112	113	114	115	116	117	118	119	120

My naive approach using "trial division" takes O(N^2) running time.

Suppose N = 1,000,000. Using Python as a calculator… $\log_2 x = \ln x / \ln 2$



```
>>> def log2(x): return math.log(x)/math.log(2)
...
>>> N=1000000
>>> N*math.log2(math.log2(N))
4316983.346365776
>>> N*N
1000000000000
>>> (N*N)/(N*math.log2(math.log2(N)))
231643.23782760094
>>>
```

Assuming the same constant factor, the naive approach takes ~230,000 times longer to run!

Practice!

William Shakespeare
IN STATISTICS

BORN AND DIED ON APRIL 23RD

LIVED UNTIL 52

INTRODUCED ALMOST 3,000 WORDS TO THE ENGLISH LANGUAGE, AND USED OVER 7,000 WORDS ONLY ONCE IN OF HIS PLAYS

HE WROTE CLOSE TO 1/10 OF THE MOST QUOTED LINES EVER WRITTEN OR SPOKEN IN ENGLISH, AND IS THE 2ND MOST QUOTED WRITER IN THE ENGLISH LANGUAGE.

HAD 7 SIBLINGS & 3 CHILDREN

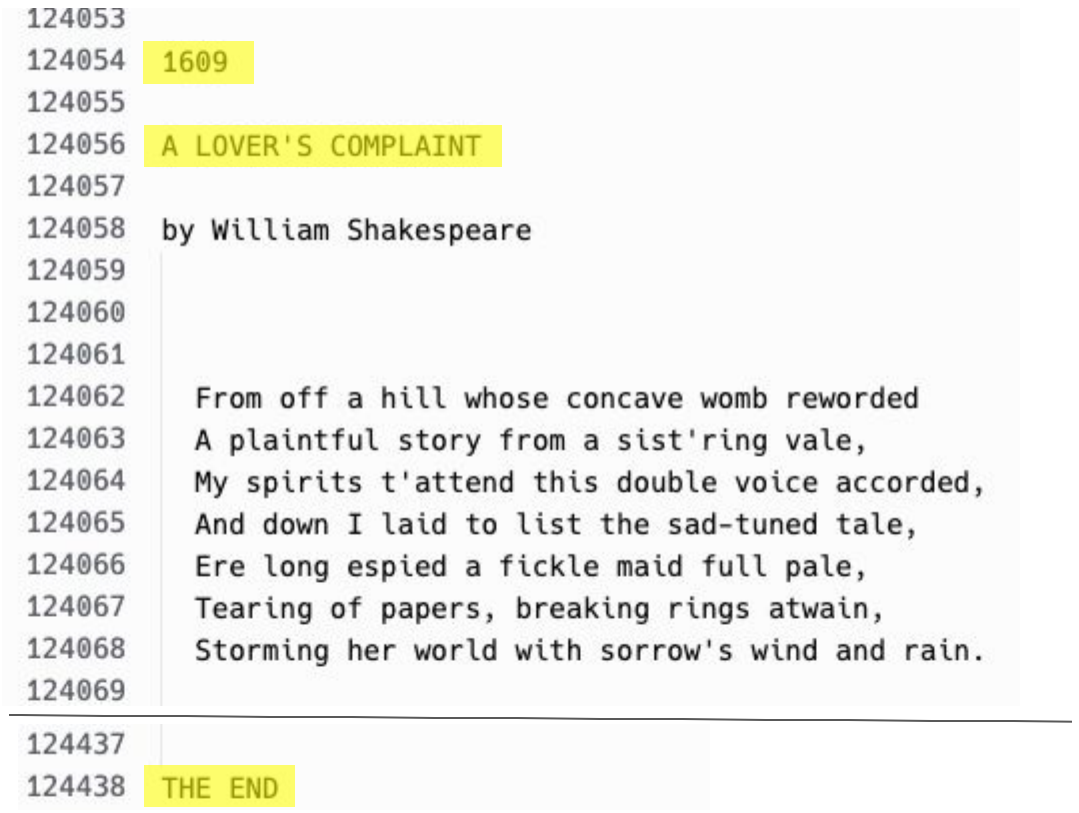
THERE ARE MORE THAN 80 VARIATIONS RECORDED FOR THE SPELLING OF HIS NAME

HIS PLAYS ARE MADE UP OF A TOTAL OF 884,429 WORDS

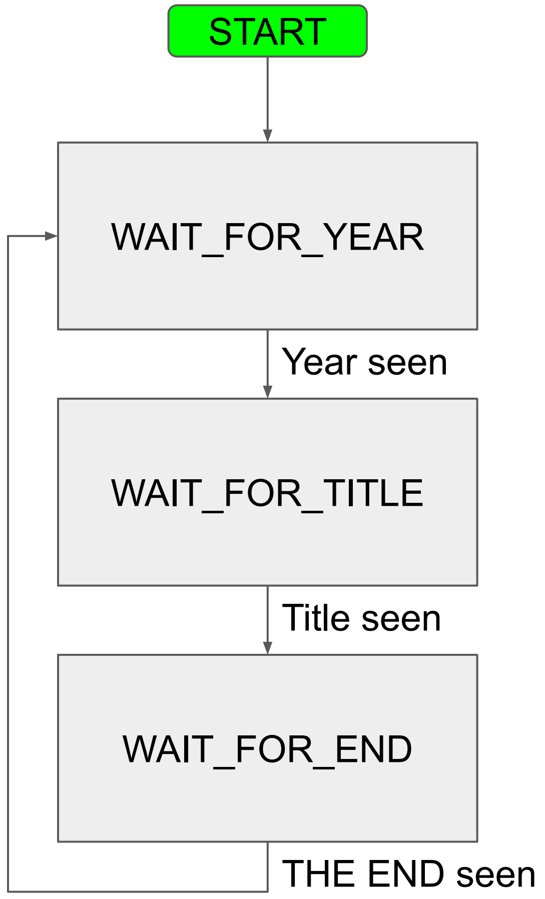
HE WROTE 37 PLAYS AND 154 WORKS THAT WE KNOW OF.

WANT YOUR OWN PERSONAL INFOGRAPHIC?
GO TO WWW.RIOKAELANI.COM
OR EMAIL KAELANI@ATRAVELBROAD.COM

Shakespeare: State Machine



```
1609

A LOVER'S COMPLAINT

 by William Shakespeare



   From off a hill whose concave womb reworded
   A plaintful story from a sist'ring vale,
   My spirits t'attend this double voice accorded,
   And down I laid to list the sad-tuned tale,
   Ere long espied a fickle maid full pale,
   Tearing of papers, breaking rings atwain,
   Storming her world with sorrow's wind and rain.

```
```

THE END
```

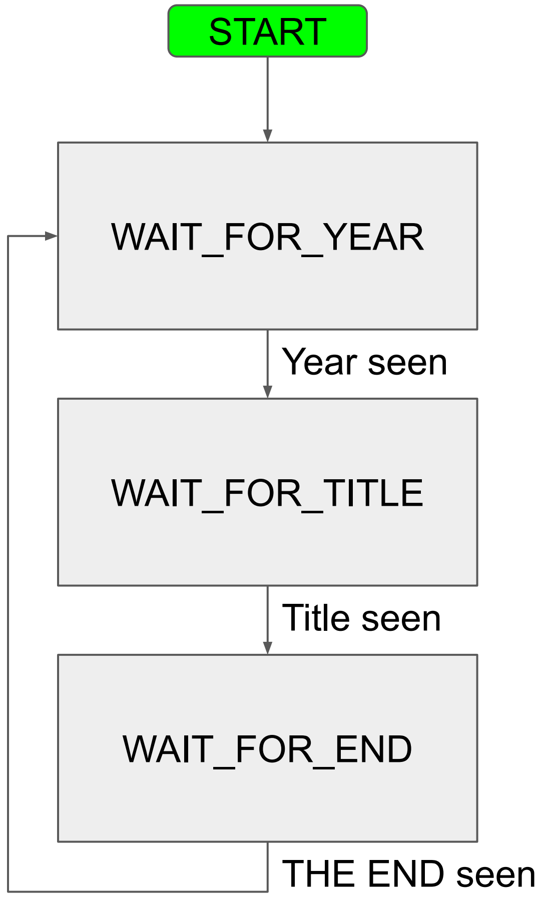


$$\begin{tikzcd}
\text{START} \arrow[d] \\
\text{WAIT\_FOR\_YEAR} \arrow[d, "\text{Year seen}"] \\
\text{WAIT\_FOR\_TITLE} \arrow[d, "\text{Title seen}"] \\
\text{WAIT\_FOR\_END} \arrow[uu, bend left=60, "\text{THE END seen}"']
\end{tikzcd}$$

Shakespeare: State Machine

```java
private static final int WAIT_FOR_YEAR_STATE  = 0;
private static final int WAIT_FOR_TITLE_STATE = 1;
private static final int WAIT_FOR_END_STATE   = 2;

private int state = WAIT_FOR_YEAR_STATE;
```

```java
while (scanner.hasNextLine()) {
  String s = scanner.nextLine();
  if (state == WAIT_FOR_YEAR_STATE) {
    handleWaitForYearState(s);
  } else if (state == WAIT_FOR_TITLE_STATE) {
    handleWaitForTitleState(s);
  } else if (state == WAIT_FOR_END_STATE) {
    handleWaitForEndState(s);
  }
}
```


START
WAIT_FOR_YEAR
Year seen
WAIT_FOR_TITLE
Title seen
WAIT_FOR_END
THE END seen