IT 179
6

Big-O Notation and Algorithm Efficiency

Start the presentation to see live content. For screen share software, share the entire screen. Get help at **pollev.com/app**

How many times will the body of this loop execute?

```
final int n = 100;
int[] T = new int[n];
for (int i = 0; i < n; i ++ ) {
        T[i] = i;
        System.out.println(T[i]);
}
```

n/2 times

n times

n + 1 times

2n times

n^2 (n square) times

Powered by Poll Everywhere

Start the presentation to see live content. For screen share software, share the entire screen. Get help at **pollev.com/app**

The of number of times the body of this loop is executed is proportional to:

```
final int n = 100;
int[] T = new int[n];
for (int i = 0; i < n; i ++ ) {
        T[i] = i;
        System.out.println(T[i]);
}
```

n

n^2

n^3

Sqrt(n)

None of the above

Powered by Poll Everywhere

Start the presentation to see live content. For screen share software, share the entire screen. Get help at **pollev.com/app**

Big-O

- The algorithm runs in O(n)

```java
final int n = 100;
int[] T = new int[n];
for (int i = 0; i < n; i ++ ) {
        T[i] = i;
        System.out.println(T[i]);
}
```

Algorithm Efficiency and Big-O

- Getting a precise measure of the performance of an algorithm is difficult

- **Big-O** notation expresses the performance of an algorithm as a function of the number of items to be processed

- This permits algorithms to be compared for efficiency

Linear Growth Rate

- If processing time increases in proportion to the number of inputs *n*, the algorithm grows at a **linear rate**

```
public static int search(int[] x, int target) {
  for(int i=0; i<x.length; i++) {
    if (x[i]==target)
      return i;
  }
  return -1; // target not found
}
```

Linear Growth Rate (cont.)

- If processing time increases in proportion to the number of inputs *n*, the algorithm grows at a linear rate

```
public static int search(int[] x, int target) {
  for(int i=0; i<x.length; i++) {
    if (x[i]==target)
      return i;
  }
  return -1; // target not found
}
```

- If the target is not present, the `for` loop will execute `x.length` times
- If the target is present the `for` loop will execute (on average) `x.length/2`
- Therefore, the total execution time is directly proportional to `x.length`
- This is described as a **growth rate** of order *n* or **O(n)**

n x m Growth Rate

- Processing time can be dependent on two different inputs.

```
public static boolean areDifferent(int[] x, int[] y) {
  for(int i=0; i<x.length; i++) {
    if (search(y, x[i]) != -1)
      return false;
  }
  return true;
}
```

n x m Growth Rate (cont.)

- Processing time can be dependent on two different inputs.

```
public static boolean areDifferent(int[] x, int[] y) {
  for(int i=0; i<x.length; i++) {
    if (search(y, x[i]) != -1)
      return false;
  }
  return true;
}
```

- The `for` loop will execute `x.length` times
- But it will call `search`, which will execute `y.length` times
- The total execution time is proportional to (`x.length * y.length`)
- The growth rate has an order of n x m or O(n x m)

Quadratic Growth Rate

- If processing time is proportional to the square of the number of inputs *n*, the algorithm grows at a quadratic rate

```
public static boolean areUnique(int[] x) {
  for(int i=0; i<x.length; i++) {
    for(int j=0; j<x.length; j++) {
      if (i != j && x[i] == x[j])
        return false;
    }
  }
  return true;
}
```

Quadratic Growth Rate (cont.)

- If processing time is proportional to the square of the number of inputs *n*, the algorithm grows at a quadratic rate

```
public static boolean areUnique(int[] x) {
  for(int i=0; i<x.length; i++) {
    for(int j=0; j<x.length; j++) {
      if (i != j && x[i] == x[j])
        return false;
    }
  }
  return true;
}
```

- The `for` loop with `i` as index will execute `x.length` times
- The `for` loop with `j` as index will execute `x.length` times
- The total number of times the inner loop will execute is `(x.length)`^{2}
- The growth rate has an order of n^2 or $O(n^2)$

Big-O Notation

- The **O()** in the previous examples can be thought of as an abbreviation of "**order of magnitude**"
- A simple way to determine the big-O notation of an algorithm is to **look at the loops** and to see whether the loops are nested
- Assuming a loop body consists only of simple statements,
  - a single loop is $O(n)$
  - a pair of nested loops is $O(n^2)$
  - a nested pair of loops inside another is $O(n^3)$
  - and so on . . .

This code runs in O(?)

```
final int n = 10;
int[][][] T = new int[n][n][n];
for (int i = 0; i < n; i++)
    for (int j = 0; j < n; j++)
        for (int k = 0; k < n; k++)
        {
            T[i][j][k] = i + j + k;
            System.out.println(T[i][j][k]);
        }
```

Powered by Poll Everywhere
Start the presentation to see live content. For screen share software, share the entire screen. Get help at **pollev.com/app**

Big-O Notation (cont.)

- You must also examine the *number of times* a loop is executed

```
for(i=1; i < x.length; i *= 2) {
   // Do something with x[i]
}
```

- The loop body will execute k-1 times, with `i` having the following values:

  $$1, 2, 4, 8, 16, \ldots, 2^k$$

  until 2^k is greater than `x.length`
- Since 2^{k-1} = `x.length` < 2^k and $\log_2 2^k$ is k, we know that k-1 = $\log_2$(`x.length`) < k
- Thus we say the loop is O(log n) (in analyzing algorithms, we use logarithms to the base 2)
- Logarithmic functions grow slowly as the number of data items n increases

This code runs in O(?)

```
int n = 1000;
int[] M = new int[n];
int i = 1;
while (i < n)
{
    M[i] = 0;
    System.out.println("M[" + i + "] = " + M[i]);
    i = i * 2;

}
```

Powered by Poll Everywhere

Start the presentation to see live content. For screen share software, share the entire screen. Get help at **pollev.com/app**

Formal Definition of Big-O

- Consider the following program structure:

```
for (int i = 0; i < n; i++) {
  for (int j = 0; j < n; j++) {
      Simple Statement
  }
}
for (int i = 0; i < n; i++) {
     Simple Statement 1
     Simple Statement 2
    Simple Statement 3
    Simple Statement 4
    Simple Statement 5
}
Simple Statement 6
Simple Statement 7
. . .
Simple Statement 30
```

Formal Definition of Big-O (cont.)

- Consider the following program structure:

```
for (int i = 0; i < n; i++) {
  for (int j = 0; j < n; j++) {
      Simple Statement
  }
}
for (int i = 0; i < n; i++) {
    Simple Statement 1
    Simple Statement 2
    Simple Statement 3
    Simple Statement 4
    Simple Statement 5
}
Simple Statement 6
Simple Statement 7
. . .
Simple Statement 30
```

This nested loop executes a *Simple Statement* n^2 times

Formal Definition of Big-O (cont.)

- Consider the following program structure:

```
for (int i = 0; i < n; i++) {
   for (int j = 0; j < n; j++) {
        Simple Statement
   }
}
for (int i = 0; i < n; i++) {
      Simple Statement 1
      Simple Statement 2
     Simple Statement 3
     Simple Statement 4
     Simple Statement 5
}
Simple Statement 6
Simple Statement 7
. . .
Simple Statement 30
```

This loop executes 5 *Simple Statements n* times ($5n$)

Formal Definition of Big-O (cont.)

- Consider the following program structure:

```
for (int i = 0; i < n; i++) {
  for (int j = 0; j < n; j++) {
      Simple Statement
  }
}
for (int i = 0; i < n; i++) {
     Simple Statement  1
     Simple Statement 2
    Simple Statement 3
    Simple Statement 4
    Simple Statement 5
}
 Simple Statement 6
 Simple Statement 7
. . .
 Simple Statement 30
```

Finally, 25 *Simple Statements* are executed

Formal Definition of Big-O (cont.)

- Consider the following program structure:

```
for (int i = 0; i < n; i++) {
   for (int j = 0; j < n; j++) {
        Simple Statement
   }
}
for (int i = 0; i < n; i++) {
     Simple Statement 1
     Simple Statement 2
     Simple Statement 3
     Simple Statement 4
     Simple Statement 5
}
Simple Statement 6
Simple Statement 7
. . .
Simple Statement 30
```

We can conclude that the relationship between processing time and *n* (the number of data items processed) is:

$$T(n) = n^2 + 5n + 25$$

Formal Definition of Big-O (cont.)

- In terms of T(n),

$$T(n) = O(f(n))$$

- There exists
  - two constants, n_0 and c, greater than zero, and
  - a function, f(n),
- such that for all $n > n_0$, $cf(n) = T(n)$
- In other words, as n gets sufficiently large (larger than n_0), there is some constant c for which the processing time will always be less than or equal to cf(n)
- **cf(n)** is an upper bound on performance

Formal Definition of Big-O (cont.)

- The growth rate of f(n) will be determined by the fastest growing term, which is the one with the largest exponent
- In the example, an algorithm of

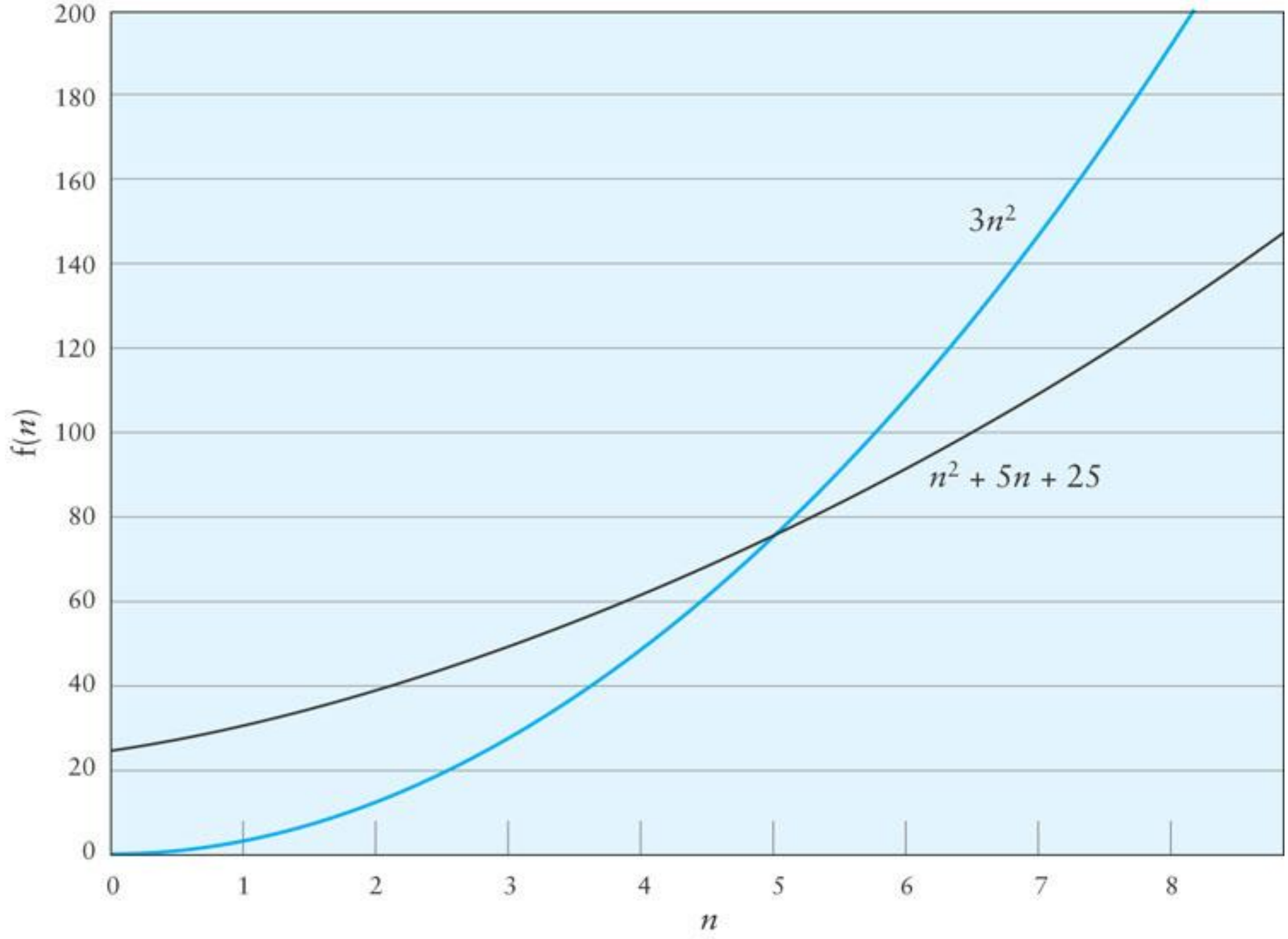
$$O(n^2 + 5n + 25)$$

  is more simply expressed as

$$O(n^2)$$

- In general, it is safe to ignore all constants and to drop the lower-order terms when determining the order of magnitude

Big-O Example 1 (cont.)



Big-O Example 2

- Consider the following loop

```
for (int i = 1; i < n; i++) {
  for (int j = 0; j < i; j++) {
      3 simple statements
  }
}
```

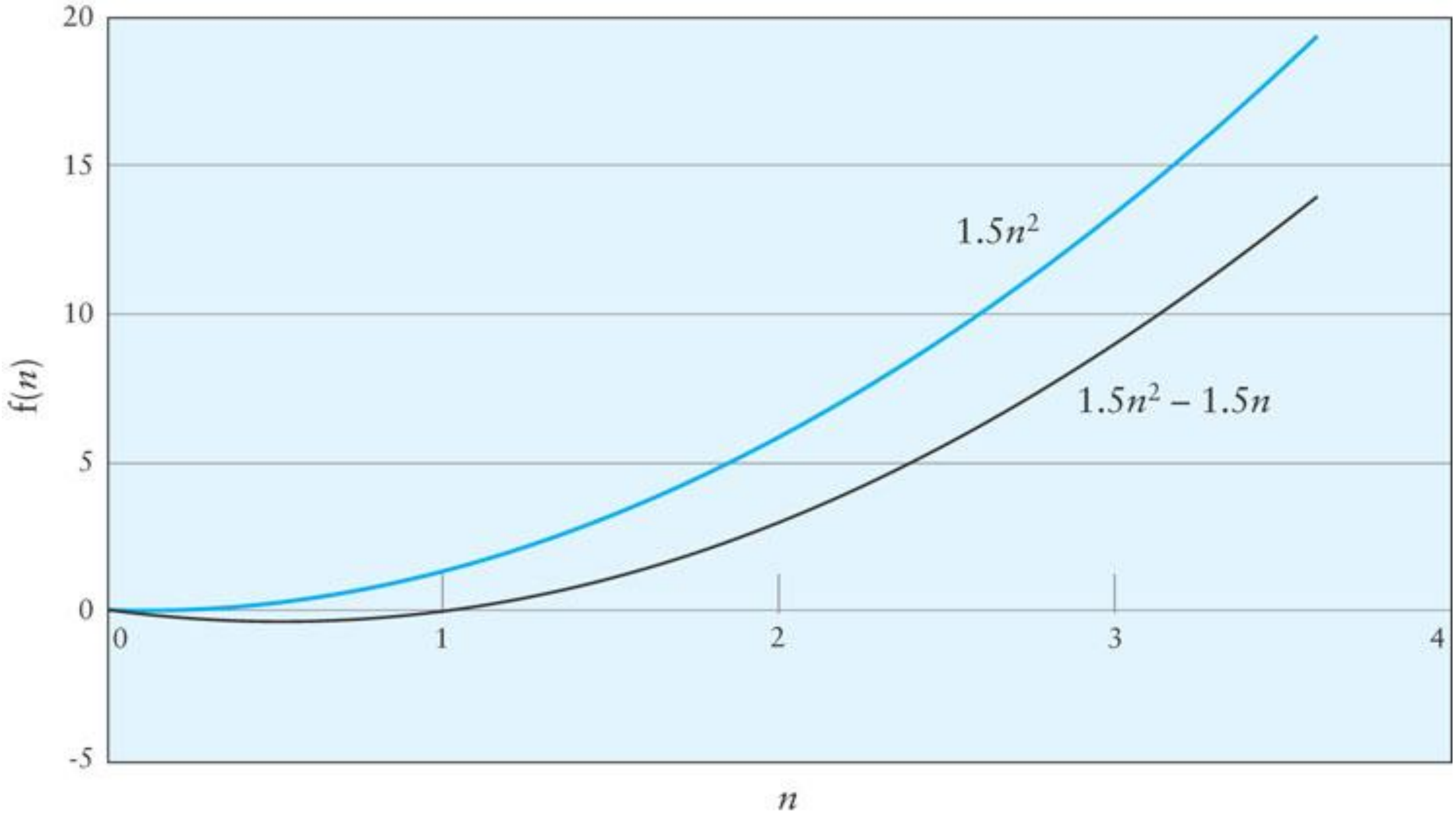
- $T(n) = 3(n - 1) + 3\ (n - 2) + \ldots + 3$
- Factoring out the 3,

  $3(n - 1 + n - 2\ + n + \ldots + 1)$
- $1 + 2 + \ldots + n - 1 = (n \times (n\text{-}1))/2$

Big-O Example 2 (cont.)

- Therefore $T(n) = 1.5n^2 - 1.5n$
- When $n = 0$, the polynomial has the value 0
- For values of $n > 1$, $1.5n^2 > 1.5n^2 - 1.5n$
- Therefore $T(n)$ is $O(n^2)$ when n_0 is 1 and c is 1.5

Big-O Example 2 (cont.)



Symbols Used in Quantifying Performance

Symbol	Meaning
$T(n)$	The time that a method or program takes as a function of the number of inputs, n. We may not be able to measure or determine this exactly.
$f(n)$	Any function of n. Generally, $f(n)$ will represent a simpler function than $T(n)$, for example, n^2 rather than $1.5n^2 - 1.5n$.
$O(f(n))$	Order of magnitude. $O(f(n))$ is the set of functions that grow no faster than $f(n)$. We say that $T(n) = O(f(n))$ to indicate that the growth of $T(n)$ is bounded by the growth of $f(n)$.

Common Growth Rates

Big-O	Name
$O(1)$	Constant
$O(\log n)$	Logarithmic
$O(n)$	Linear
$O(n \log n)$	Log-linear
$O(n^2)$	Quadratic
$O(n^3)$	Cubic
$O(2^n)$	Exponential
$O(n!)$	Factorial

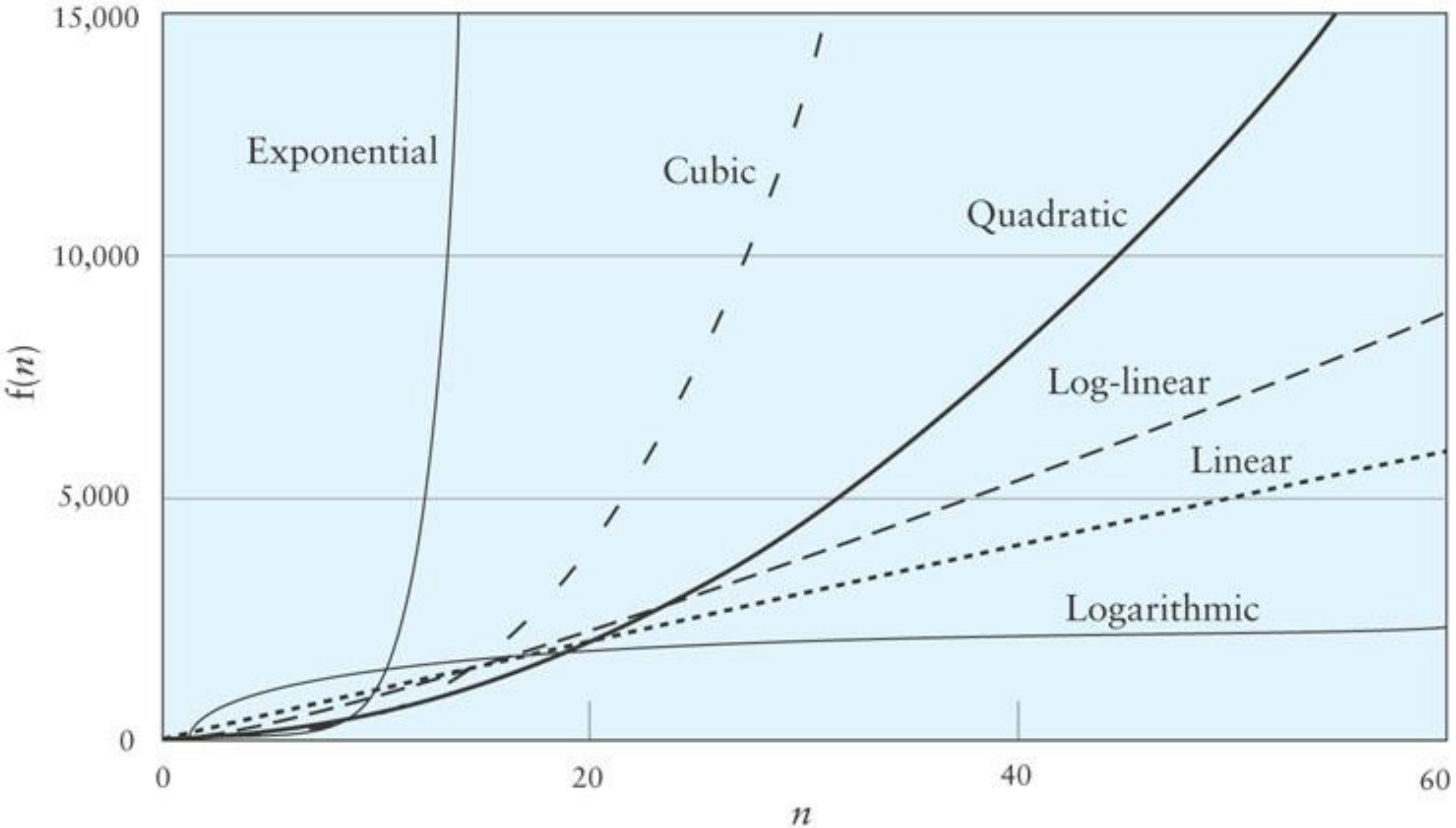
This runs in O(?)

```
i = j + 2;
```

Powered by Poll Everywhere

Start the presentation to see live content. For screen share software, share the entire screen. Get help at **pollev.com/app**

Different Growth Rates



Power of O(log *n*) Algorithm

- Searching an array with linear search (O(*n*)) requires checking each element to see if it matches the search target
- With an O(log *n*) algorithm, number of elements is cut in half with each probe into the array
- For 1024 elements in an array – need to search all 1024, then 512, 256, 128, 64, 32, 16, 8, 4, 2, 1
- Only 10 probes are required to find the target or determine it is not in the array

Algorithms with Exponential and Factorial Growth Rates

- Algorithms with exponential and factorial growth rates have an effective practical limit on the size of the problem they can be used to solve
- With an $O(2^n)$ algorithm, if 100 inputs takes an hour then,
  - 101 inputs will take 2 hours
  - 105 inputs will take 32 hours
  - 114 inputs will take 16,384 hours (almost 2 years!)

Algorithms with Exponential and Factorial Growth Rates (cont.)

- Encryption algorithms take advantage of this characteristic
- Some cryptographic algorithms can be broken in $O(2^n)$ time, where n is the number of bits in the key
- A key length of 40 is considered breakable by a modern computer,
- But a key length of 100 bits will take a billion-billion (10^{18}) times longer than a key length of 40 bits

What is the time complexity of this code?

```
public static void main(String[] args)
  {
    int a = 0, b = 0;
    int N = 4, M = 4;

    // This loop runs for N time
    for (int i = 0; i < N; i++)
    {
      a = a + 10;
    }

    // This loop runs for M time
    for (int i = 0; i < M; i++)
    {
      b = b + 40;
    }
    System.out.print(a + " " + b);
  }
}
```