Data Structures and Algorithm

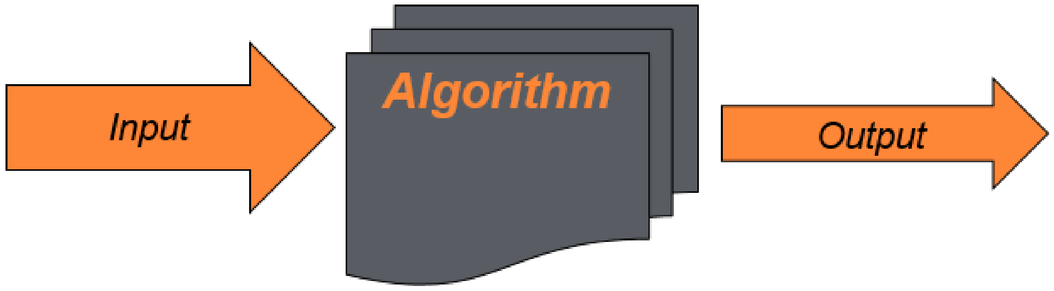
(CSE 225)

Lecture 3

(Complexity of Algorithm)

What is an algorithm?

- A computational procedure that takes
  - ❑some value, or set of values, as ***input***
  - ❑produces some value, or set of values, as ***output***
  - ❑may be specified:
    - o In English
    - o As a pseudocode
    - o As a computer program



- A sequence of computational steps that transform the input into the output.

Analysis of Algorithms

- What does it mean to analyze an algorithm?
  - To have an estimate about how much time an algorithm may take to finish, or in other words, to analyze its running time (aka. Time complexity)
  - Sometimes, instead of running time, we are interested in how much memory/space the algorithm may consume while it runs (space complexity)
- It enables us to compare between two algorithms
- What do we mean by running time analysis?
  - Also referred to as time-complexity analysis
  - To determine how running time increases as the size of the problem increases.

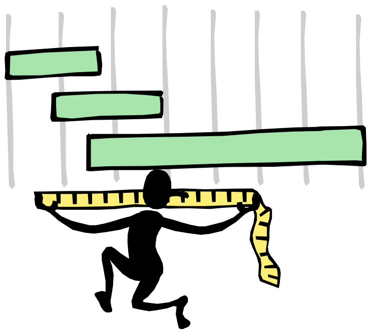
Analysis of Algorithms

- Input size (number of elements in the input)
  - size of an array
  - # of elements in a matrix
  - # of bits in the binary representation of the input
  - vertices and edges in a graph

Why study algorithms and performance?

- Algorithms help us to understand ***scalability***.
- Performance often draws the line between what is feasible and what is impossible.
- Algorithmic mathematics provides a ***language*** for talking about program behavior.
- Performance is the ***currency*** of computing.

Measure Actual Running Time?



- We can measure the actual running time of a **program**
  - Use **stopwatch** or insert timing code into program

- However, actual running time is not meaningful when **comparing** two **algorithms**
  - Coded in different languages
  - Using different data sets
  - Running on different computers

Should we give up trying to measure?

Counting Operations

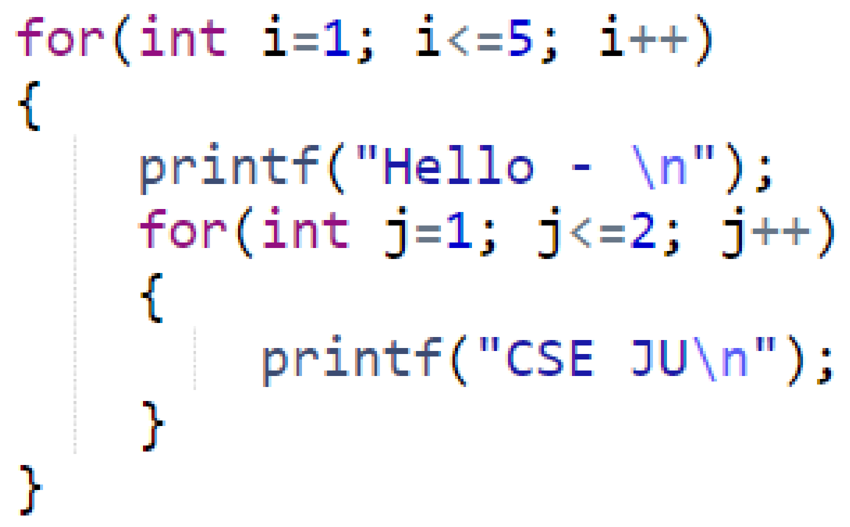
- What we can control?
  - ❑ Count operations instead of time
    - ➢ The number of operations and count them.
    - ➢ Large input, more operations
    - ➢ Small input, less operations
  - ❑ Focus on how performance scales (how the number of operations grows as our input grows)
  - ❑ Go beyond input size

(how the algorithm performs in all sort of situations.. How the internal structure of the input)

Example: Counting Operations

- How many operations are required?

```
for (int i = 1; i <= n; i++) {
     perform 100 operations;     // A
     for (int j = 1; j <= n; j++) {
          perform 2 operations; // B
     }
}
```



```
for(int i=1; i<=5; i++)
{
    printf("Hello - \n");
    for(int j=1; j<=2; j++)
    {
        printf("CSE JU\n");
    }
}
```

Example: Counting Operations

- How many operations are required?

```
for (int i = 1; i <= n; i++) {
     perform 100 operations;     // A
     for (int j = 1; j <= n; j++) {
          perform 2 operations; // B
     }
}
```

Total Ops = A + B $= \sum_{i=1}^{n} 100 + \sum_{i=1}^{n} (\sum_{j=1}^{n} 2)$

$$= 100n + \sum_{i=1}^{n} 2n \quad = 100n + 2n^2 \quad = 2n^2 + 100n$$

Example: Counting Operations

- Knowing the number of operations required by the algorithm, we can state that
  - Algorithm X takes $2n^2 + 100n$ operations to solve problem of size n
- If the time t needed for one operation is known, then we can state
  - Algorithm X takes $(2n^2 + 100n)t$ time units

Example: Counting Operations

- However, time *t* is directly dependent on the factors mentioned earlier
  - e.g. different languages, compilers and computers
- Instead of tying the analysis to actual time *t*, we can state
  - Algorithm *X* takes time that is **proportional to** $2n^2 + 100n$ for solving problem of size *n*

Approximation of Analysis Results

- Suppose the time complexity of
  - Algorithm *A* is $\mathbf{3n^2} + 2n + \log n + 1/(4n)$
  - Algorithm *B* is $\mathbf{0.39n^3} + n$
- Intuitively, we know Algorithm *A* will outperform *B*
  - When solving larger problem, i.e. larger *n*
- The **dominating term** $\mathbf{3n^2}$ and $\mathbf{0.39n^3}$ can tell us approximately how the algorithms perform
- The terms $\mathbf{n^2}$ and $\mathbf{n^3}$ are even simpler and preferred
- These terms can be obtained through **asymptotic analysis**

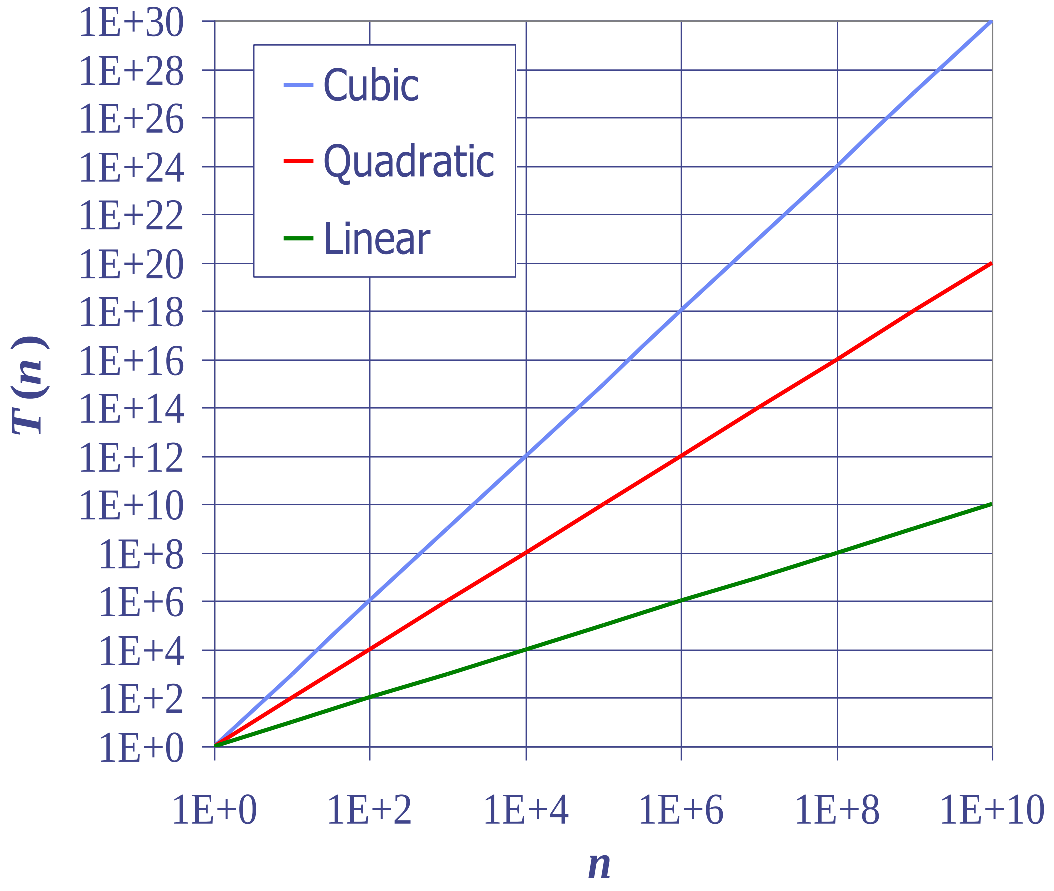
Asymptotic Analysis

- Asymptotic analysis is an analysis of algorithms that focuses on
  - Analyzing problems of large input size
  - Consider only the leading term of the formula
  - Ignore the coefficient of the leading term

Why Choose Leading Term?

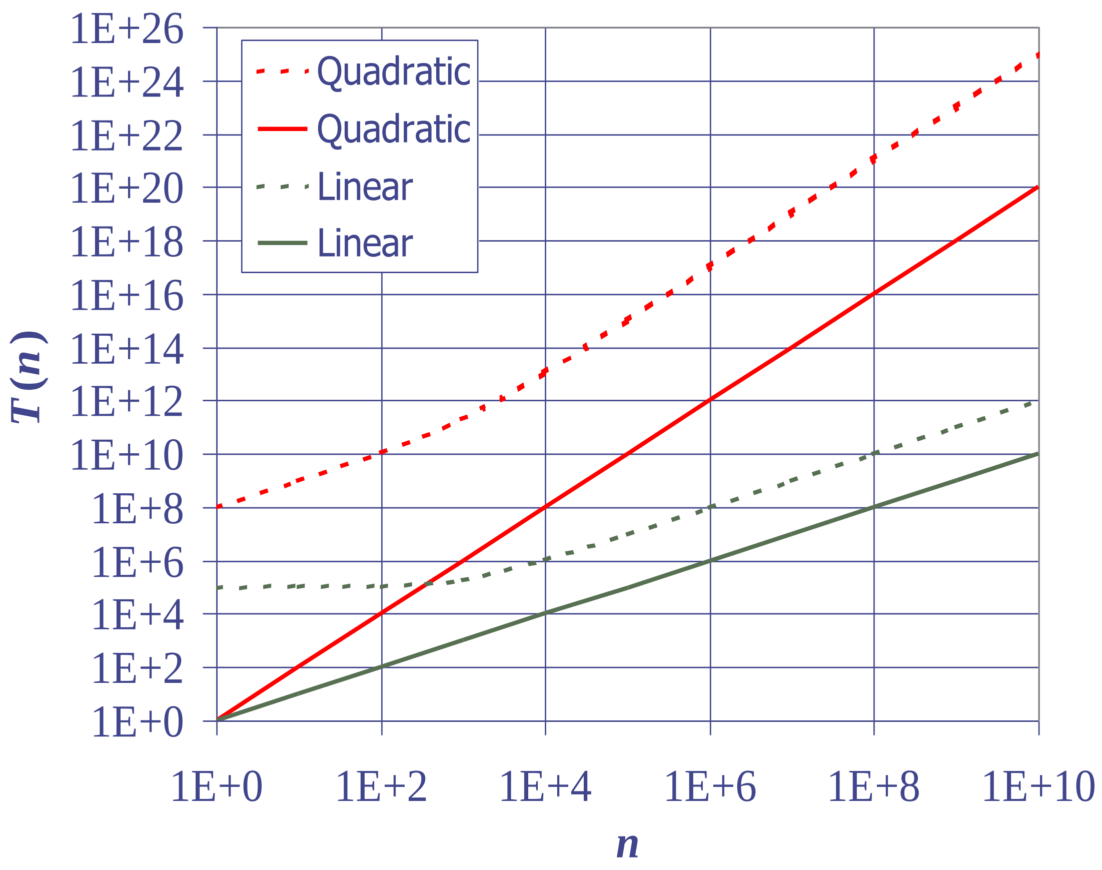
- Growth rates of functions:
  - Linear $\approx \boldsymbol{n}$
  - Quadratic $\approx \boldsymbol{n}^2$
  - Cubic $\approx \boldsymbol{n}^3$

- In a log-log chart, the slope of the line corresponds to the growth rate of the function



Why Choose Leading Term?

- The growth rate is not affected by
  - constant factors or
  - lower-order terms
- Examples
  - $10^2 n + 10^5$ is a linear function
  - $10^5 n^2 + 10^8 n$ is a quadratic function



Why Choose Leading Term?

- Lower order terms contribute lesser to the overall cost as the input grows larger

n	$f(n) = n^3$	$f(n) = n^3 + n^2$	$f(n) = n^3 + n^2 + 20n$
1	1	2	22
10	1,000	1,100	1,300
1,000	1,000,000,000	1,001,000,000	1,001,020,000
1,000,000	1.0×10^{18}	1.000001×10^{18}	1.000001×10^{18}

Why Choose Leading Term?

- Lower order terms contribute lesser to the overall cost as the input grows larger

n	$f(n) = n^3$	$f(n) = n^3 + n^2$	$f(n) = n^3 + n^2 + 20n$
1	1	2	22
10	1,000	1,100	1,300
1,000	1,000,000,000	1,001,000,000	1,001,020,000
1,000,000	1.0×10^{18}	1.000001×10^{18}	1.000001×10^{18}

Why Choose Leading Term?

- Lower order terms contribute lesser to the overall cost as the input grows larger

n	$f(n) = n^3$	$f(n) = n^3 + n^2$	$f(n) = n^3 + n^2 + 20n$
1	1	2	22
10	1,000	1,100	1,300
1,000	1,000,000,000	1,001,000,000	1,001,020,000
1,000,000	1.0×10^{18}	1.000001×10^{18}	1.000001×10^{18}

Why Choose Leading Term?

- Lower order terms contribute lesser to the overall cost as the input grows larger

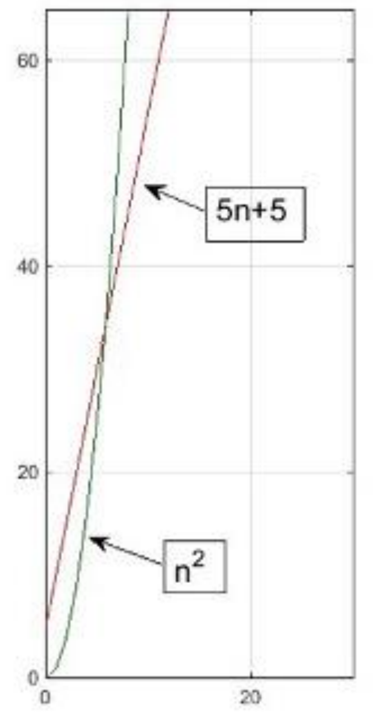
n	$f(n) = n^3$	$f(n) = n^3 + n^2$	$f(n) = n^3 + n^2 + 20n$
1	1	2	22
10	1,000	1,100	1,300
1,000	1,000,000,000	1,001,000,000	1,001,020,000
1,000,000	1.0×10^{18}	1.000001×10^{18}	1.000001×10^{18}

Why Choose Leading Term?

- Lower order terms contribute lesser to the overall cost as the input grows larger

n	$f(n) = n^3$	$f(n) = n^3 + n^2$	$f(n) = n^3 + n^2 + 20n$
1	1	2	22
10	1,000	1,100	1,300
1,000	1,000,000,000	1,001,000,000	1,001,020,000
1,000,000	1.0×10^{18}	1.000001×10^{18}	1.000001×10^{18}

Why ignore the coefficient of the leading term



How Do We Analyze Running Time?

- We need to define an <u>objective measure</u>.
  - Count the number of statements executed
    - Associate a "cost" with each statement and find the "total cost" by finding the total number of times each statement is executed.
    - ***Not good***: number of statements vary with the programming language as well as the style of the individual programmer; therefore it can't be an objective measure of running time. For e.g. following two algorithms do the same task using different programming style.

Algorithm 1

Time (micro sec)

```
arr[0] = 0;
arr[1] = 0;
arr[2] = 0;                          ----------------------------
...
arr[N-1] = 0;
 ----------------------
```

How Do We Analyze Running Time?

- We need to define an <u>objective measure</u>.
  - Count the number of statements executed
    - Associate a "cost" with each statement and find the "total cost“ by finding the total number of times each statement is executed.
    - ***Not good***: number of statements vary with the programming language as well as the style of the individual programmer; therefore it can’t be an objective measure of running time. For e.g. following two algorithms do the same task using different programming style.

Algorithm 1

```
                          Time (micro sec)
arr[0] = 0;                    c1
arr[1] = 0;                    c1
arr[2] = 0;                    c1
...
arr[N-1] = 0;                  c1
 ----------------------
```

$f(n) = c_1+c_1+\ldots+c_1 = c_1N$

Algorithm 2

```
    c1   c2   c3                          Time (micro sec)
for(i=0; i<N; i++)      c1+c2(N+1)+ c3N
        arr[i] = 0;                 c1N
----------------------------
```

$g(n) = c_1+c_2(N+1)+c_3N+c_1N = (c_1+c_2+c_3)N+(c_1+c_2)$

Observation: For very large values of N, execution time of both programs is similar. Thus we can say that both of them have roughly the same running time.

Ideal Solution to Express running time

- Express runtime as a function of the input size *n* (i.e., *f(n)*) in order to understand how *f(n)* grows with *n*
- *and* count only the most significant term of *f(n)* and ignore everything else (because those won't affect running time much for very large values of n).

- Thus the running times (also called time complexity) of the programs of the previous slide becomes:
  - $f(N)= c_1 N \approx N*(some\ constant)$
  - $g(N) = (c1+c2+c3)N+(c1+c2) \approx N*(some\ constant)$
- Thus both these functions grows linearly with N and as such both have the same running time (specifically, linear running time). We say that both of them are O(N) functions, which means that, the running time of each of these algorithms is a constant multiple of N (we ignore the value of the constant).

- We compare running times of different functions in an *asymptotic* manner (i.e., we check if $f(n) > g(n)$ for very large values of n). That's why, the task of computing **time complexity** (of an algorithm) is also called **asymptotic analysis**.

Intuition behind Asymptotic Analysis

- Consider the example of buying *elephants* and *goldfish:*

  Cost: cost_of_elephants + cost_of_goldfish

  Cost ~ cost_of_elephants (approximation)

- The low order terms, as well as constants in a function are relatively insignificant for **large** *n*

$$f(n) = 6n + 4 \approx n$$

$$g(n) = n^4 + 100n^2 + 10n + 50 \approx n^4$$

i.e., we say that $n^4 + 100n^2 + 10n + 50$ and n^4 have the same **rate of growth**. However, *f(n)* and *g(n)* have different rate of growths.

The Purpose of Asymptotic Analysis

- To **estimate** how long a program will run.
- To estimate the largest input that can reasonably be given to the program.
- To compare the efficiency of different algorithms.
- To help focus on the parts of code that are executed the largest number of times.
- To choose an algorithm for an application.

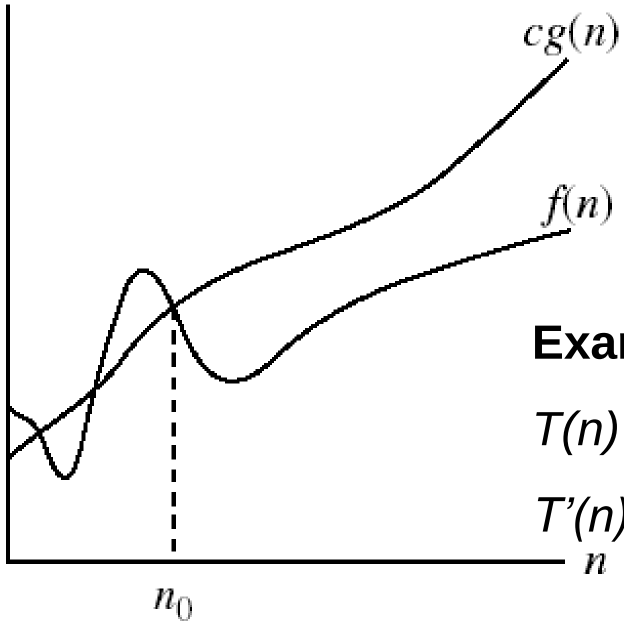
Asymptotic Notations

- O notation: asymptotic “upper bound”:
- Ω notation: asymptotic “lower bound”:
- Θ notation: asymptotic “tight bound”:

Asymptotic Notations

- *O-notation (Big Oh)*

$$O(g(n)) = \{f(n) : \text{ there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \le f(n) \le cg(n) \text{ for all } n \ge n_0\}\,.$$



$O(g(n))$ is the set of functions with smaller or same order of growth as $g(n)$

Examples:

$T(n) = 3n^2+10n\lg n+8$ is $O(n^2)$, $O(n^2\lg n)$, $O(n^3)$, $O(n^4)$, …

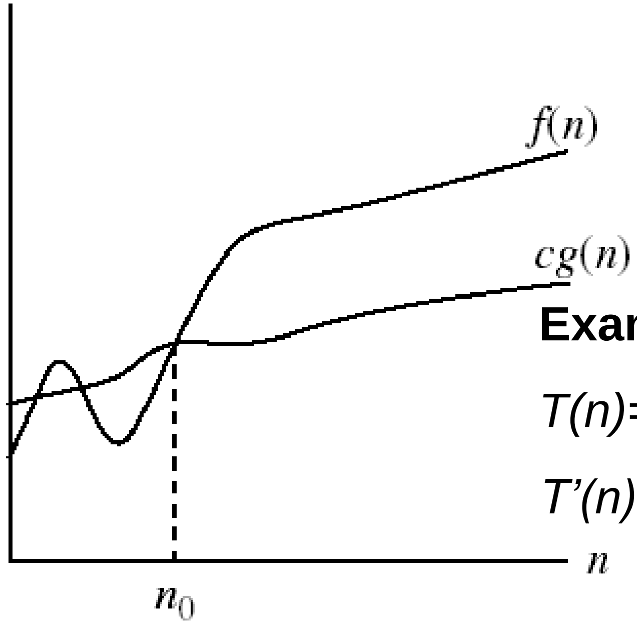
$T'(n) = 52n^2+3n^2\lg n+8$ is $\underbrace{O(n^2\lg n), O(n^3), O(n^4), \ldots}_{\text{Loose upper bounds}}$

$g(n)$ is an ***asymptotic upper bound*** for $f(n)$.

Asymptotic Notations

- Ω - *notation (Big Omega)*

$$\Omega(g(n)) = \{f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \le cg(n) \le f(n) \text{ for all } n \ge n_0\} .$$



$\Omega(g(n))$ is the set of functions with larger or same order of growth as $g(n)$

Examples:

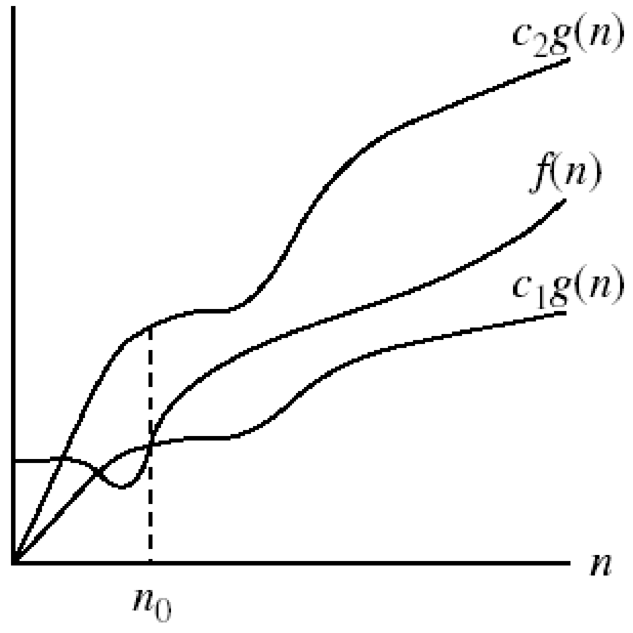
$T(n)=3n^2+10n\lg n+8$ is $\Omega(n^2)$, $\Omega(n\lg n)$, $\Omega(n)$, $\Omega(\lg n)$,$\Omega(1)$

$T'(n) = 52n^2+3n^2\lg n+8$ is $\Omega(n^2\lg n)$, $\Omega(n^2)$, $\Omega(n)$, …

$g(n)$ is an ***asymptotic lower bound*** for $f(n)$.

Asymptotic Notations

$$\Theta(g(n)) = \{f(n) : \text{there exist positive constants } c_1, c_2, \text{and } n_0 \text{ such that } 0 \le c_1 g(n) \le f(n) \le c_2 g(n) \text{ for all } n \ge n_0\} .$$



$\Theta(g(n))$ is the set of functions with the same order of growth as $g(n)$

* $f(n)$ is both $O(g(n))$ & $\Omega(g(n))$ $\leftrightarrow$ $f(n)$ is $\Theta(g(n))$

Examples:

$T(n) = 3n^2+10n\lg n+8$ is $\Theta(n^2)$

$T'(n) = 52n^2+3n^2\lg n+8$ is $\Theta(n^2\lg n)$

$g(n)$ is an ***asymptotically tight bound*** for $f(n)$.

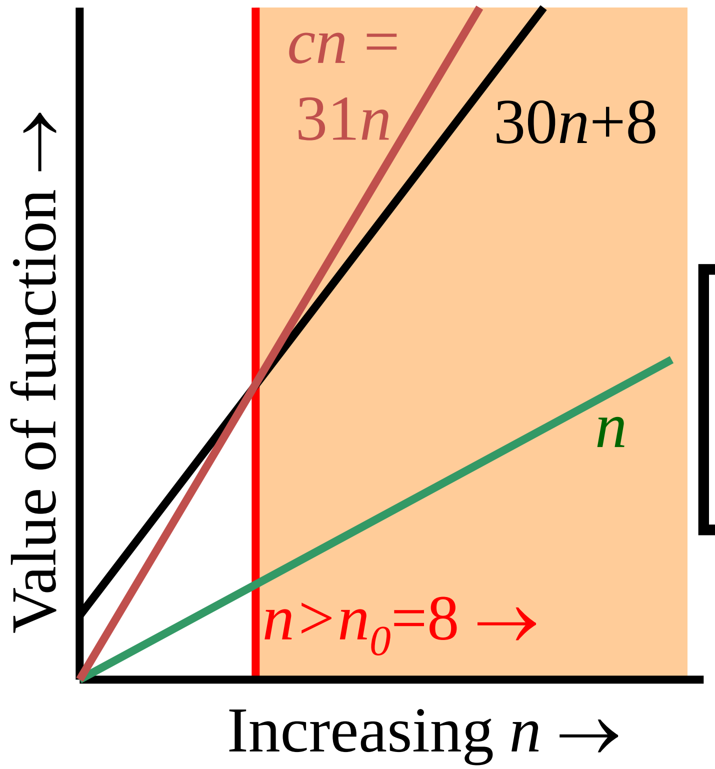
Examples

- Show that $30n+8$ is $O(n)$.
  - Show $\exists c, n_0$: $30n+8 \leq cn$, $\forall n \geq n_0$.

  - Let $c=31$, $n_0=8$
    $cn = 31n = 30n + n \geq 30n+8$,
  - so $30n+8 \leq cn$.

Big-O example, graphically

- Note 30*n*+8 isn't less than *n* *anywhere* (*n*>0).
- It isn't even less than 31*n* *everywhere*.
- But it *is* less than 31*n* <u>everywhere to the right of *n*=8</u>.



$$\boxed{30n+8 \in O(n)}$$

Example: Exponential-Time Algorithm

- Suppose we have a problem that, for an input consisting of *n* items, can be solved by going through 2^n cases
- We use a supercomputer, that analyses 200 million cases per second
  - Input with 15 items — 163 microseconds
  - Input with 30 items — 5.36 seconds
  - Input with 50 items — more than two months
  - Input with 80 items — 191 million years

Example: Quadratic-Time Algorithm

- Suppose solving the same problem with another algorithm will use $300n^2$ clock cycles on a Handheld PC, running at 33 MHz
  - Input with 15 items — 2 milliseconds
  - Input with 30 items — 8 milliseconds
  - Input with 50 items — 22 milliseconds
  - Input with 80 items — 58 milliseconds
- Therefore, to speed up program, don't simply rely on the raw power of a computer
  - Very important to use an efficient algorithm

- Once upon a time there was an Indian king who wanted to reward a wise man for his excellence. The wise man asked for nothing but some wheat that would fill up a chess board.

- But here were his rules: in the first tile he wants 1 grain of wheat, then 2 on the second tile, then 4 on the next one…each tile on the chess board needed to be filled by double the amount of grains as the previous one. The naïve king agreed without hesitation, thinking it would be a trivial demand to fulfill, until he actually went on and tried it…

- So how many grains of wheat does the king owe the wise man? We know that a chess board has 8 squares by 8 squares, which totals 64 tiles, so the final tile should have 2^{64} grains of wheat. If you do a calculation online, you will end up getting $1.8446744*10^{19}$, that is about 18 followed by 18 zeroes. Assuming that each grain of wheat weights 0.01 grams, that gives us 184,467,440,737 tons of wheat. And 184 billion tons is quite a lot, isn't it?

```
public static boolean hasLetter (String word, char letter)
{
  for (int i = 0; i < word.length(); i++)
  {
    if (word.charAt(i) == letter)
    {
       return true;
    }
  }
  return false;
}
```

Linear search

```
public static boolean hasLetter (String word, char letter)
{
  for (int i = 0; i < word.length(); i++)
  {
    if (word.charAt(i) == letter)
    {
       return true;
    }
  }
  return false;
}
```

Linear search

```
public static boolean hasLetter (String word, char letter)
{
  for (int i = 0; i < word.length(); i++)
  {
    if (word.charAt(i) == letter)
    {
       return true;
    }
  }
  return false;
}
```

Linear search

0	1	2	3	4	5	6	7	8
S	a	n		D	i	e	g	o

```
public static boolean hasLetter (String word, char letter)
{
  for (int i = 0; i < word.length(); i++)
  {
    if (word.charAt(i) == letter)
    {
      return true;
    }
  }
  return false;
}
```

0	1	2	3	4	5	6	7	8
S	a	n		D	i	e	g	o

```java
public static boolean hasLetter (String word, char letter)
{
  for (int i = 0; i < word.length(); i++)
  {
    if (word.charAt(i) == letter)
    {
       return true;
    }
  }
  return false;
}
```

0	1	2	3	4	5	6	7	8
S	a	n		D	i	e	g	o

```java
public static boolean hasLetter (String word, char letter)
{
  for (int i = 0; i < word.length(); i++)
  {
    if (word.charAt(i) == letter)
    {
       return true;
    }
  }
  return false;
}
```

0	1	2	3	4	5	6	7	8
S	a	n		D	i	e	g	o

```java
public static boolean hasLetter (String word, char letter)
{
  for (int i = 0; i < word.length(); i++)
  {
    if (word.charAt(i) == letter)
    {
       return true;
    }
  }
  return false;
}
```

0	1	2	3	4	5	6	7	8
S	a	n		D	i	e	g	o

```
public static boolean hasLetter (String word, char letter)
{
  for (int i = 0; i < word.length(); i++)
  {
    if (word.charAt(i) == letter)
    {
       return true;
    }
  }
  return false;
}
```

0	1	2	3	4	5	6	7	8
S	a	n		D	i	e	g	o

```
public static boolean hasLetter (String word, char letter)
{
  for (int i = 0; i < word.length(); i++)
  {
    if (word.charAt(i) == letter)
    {
       return true;
    }
  }
  return false;
}
```

0	1	2	3	4	5	6	7	8
S	a	n		D	i	e	g	o

```java
public static boolean hasLetter (String word, char letter)
{
  for (int i = 0; i < word.length(); i++)
  {
    if (word.charAt(i) == letter)
    {
       return true;
    }
  }
  return false;
}
```

0	1	2	3	4	5	6	7	8
S	a	n		D	i	e	g	o

```
public static boolean hasLetter (String word, char letter)
{
  for (int i = 0; i < word.length(); i++)
  {
    if (word.charAt(i) == letter)
    {
      return true;
    }
  }
  return false;
}
```

How many operations get executed?

For a single iteration:

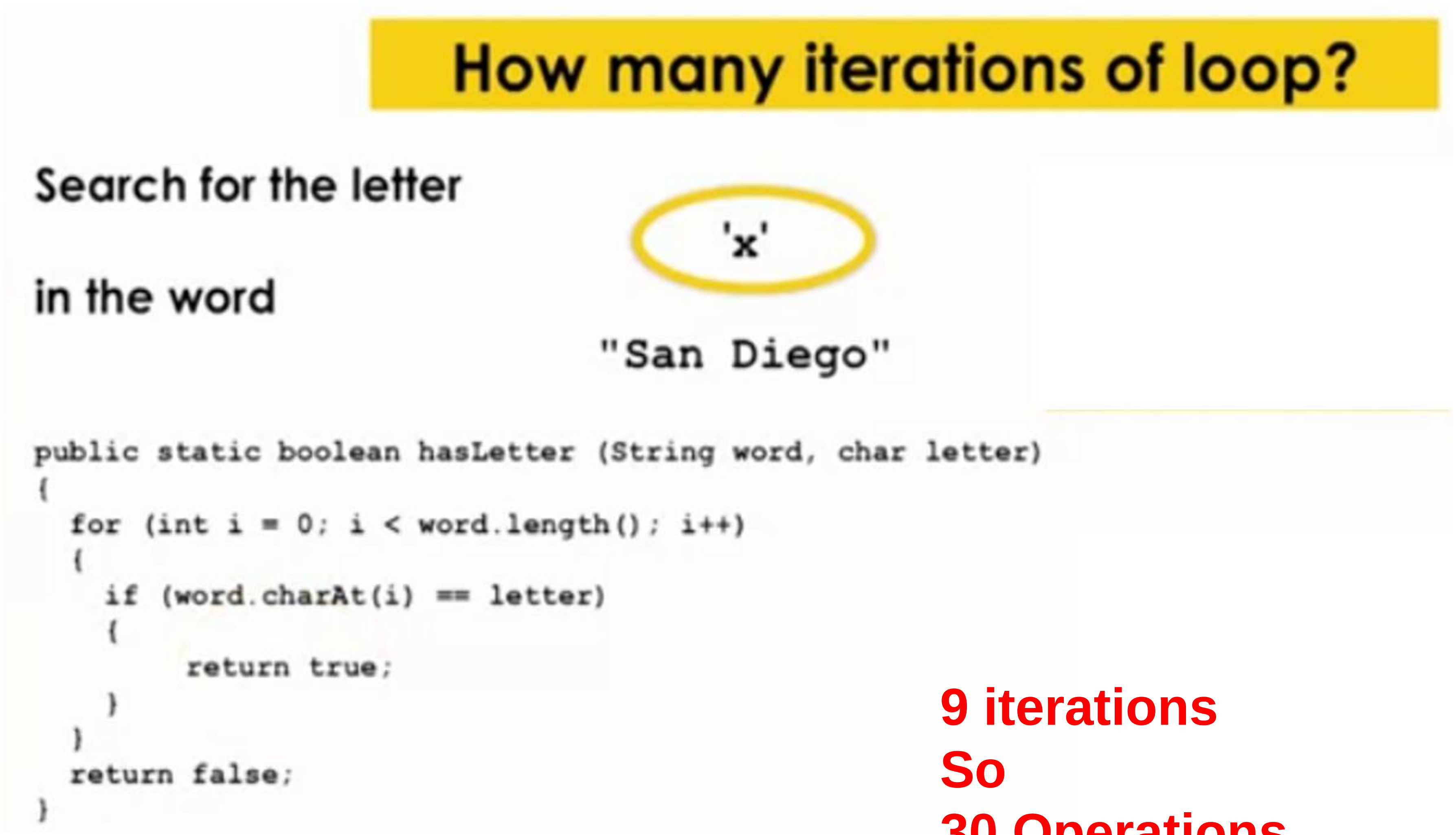
```
public static boolean hasLetter (String word, char letter)
{
  for (int i = 0; i < word.length(); i++)
  {
    if (word.charAt(i) == letter)
    {
       return true;
    }
  }
  return false;
}
```

How many operations get executed?

For a single iteration:

```
public static boolean hasLetter (String word, char letter)
{
  for (int i = 0; i < word.length(); i++)
  {
    if (word.charAt(i) == letter)
    {
       return true;
    }
  }
  return false;
}
```

3



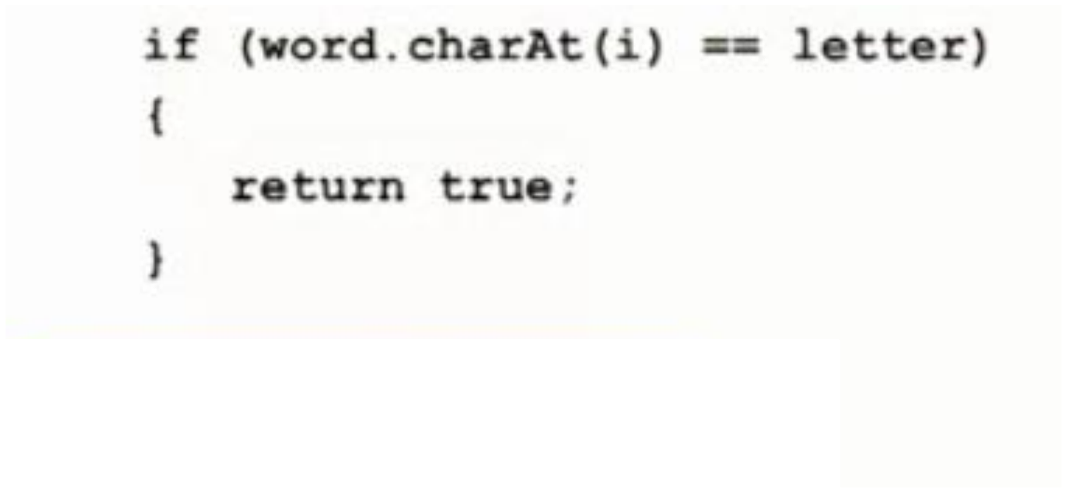
How many iterations of loop?

Search for the letter 'x'

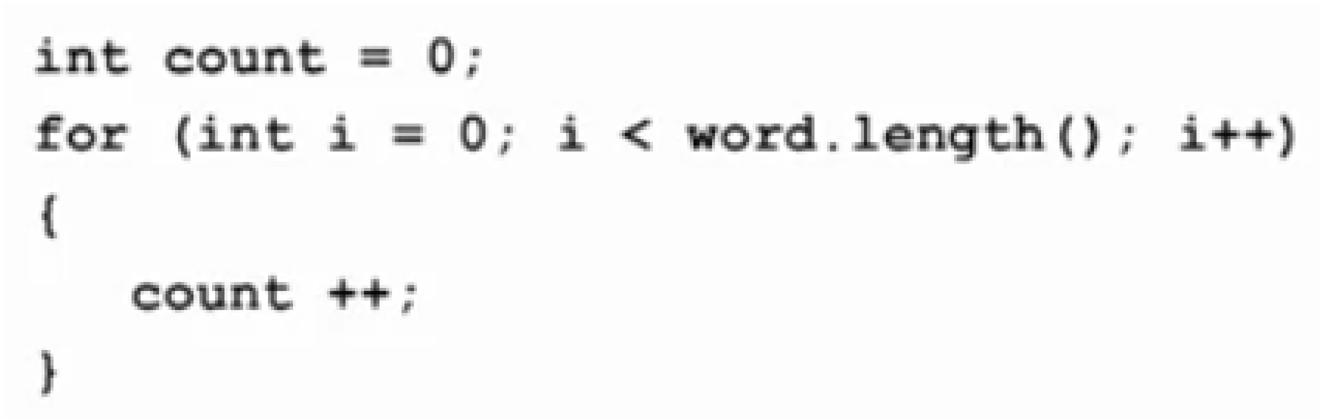
in the word "San Diego"

```
public static boolean hasLetter (String word, char letter)
{
   for (int i = 0; i < word.length(); i++)
   {
     if (word.charAt(i) == letter)
     {
          return true;
     }
   }
   return false;
}
```

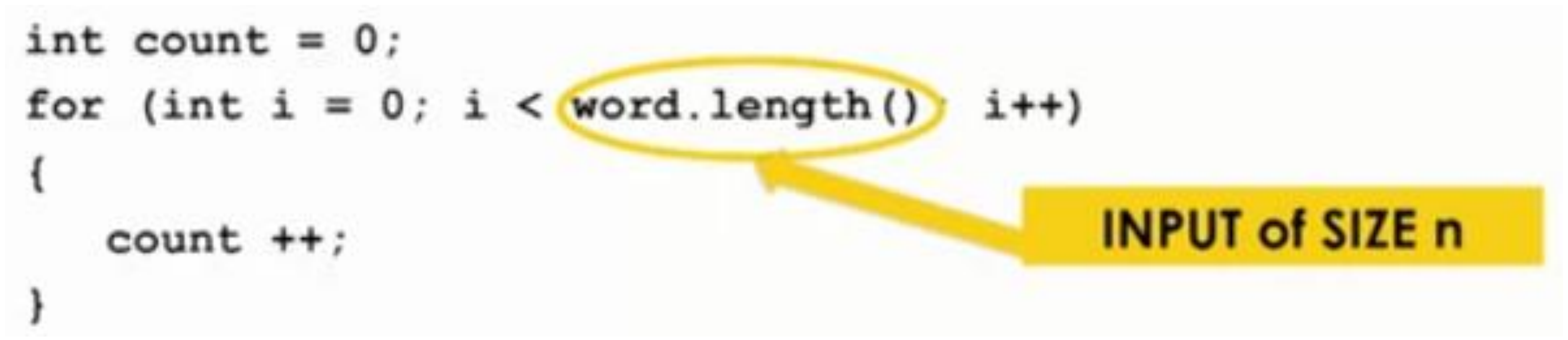
9 iterations
So
30 Operations



```
if (word.charAt(i) == letter)
{
   return true;
}
```



```
int count = 0;
for (int i = 0; i < word.length(); i++)
{
   count ++;
}
```

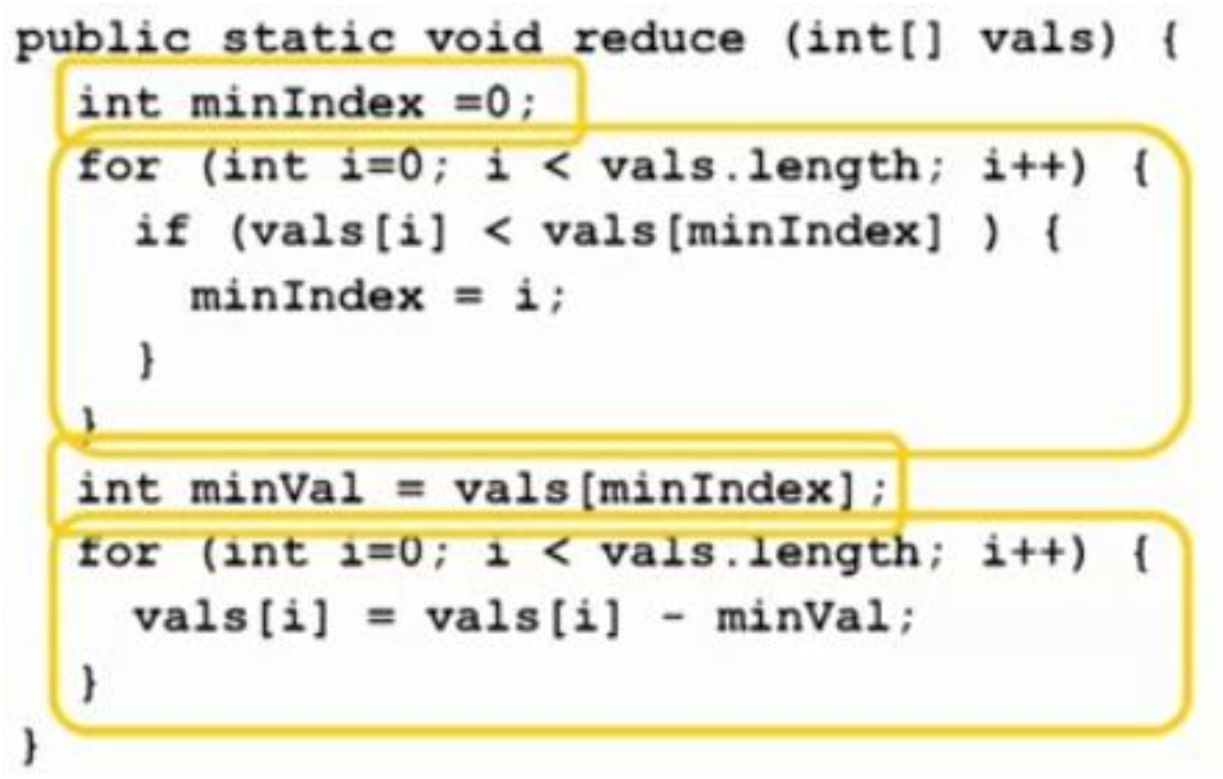


```
int count = 0;
for (int i = 0; i < word.length(); i++)
{
   count ++;
}
```

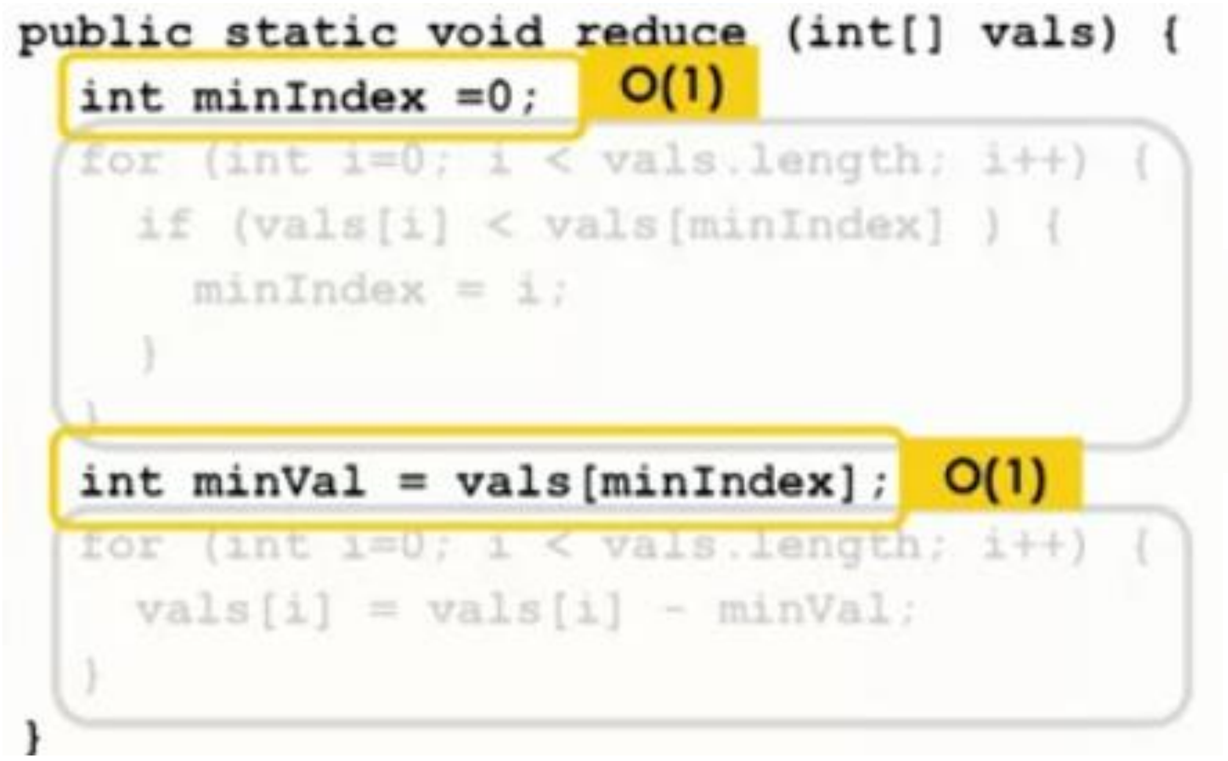
```
int count = 0;
for (int i = 0; i < word.length(); i++)
{
   count ++;
}
```

3n + 3

```
public static void reduce (int[] vals) {
  int minIndex =0;
  for (int i=0; i < vals.length; i++) {
    if (vals[i] < vals[minIndex] ) {
      minIndex = i;
    }
  }
  int minVal = vals[minIndex];
  for (int i=0; i < vals.length; i++) {
    vals[i] = vals[i] - minVal;
  }
}
```

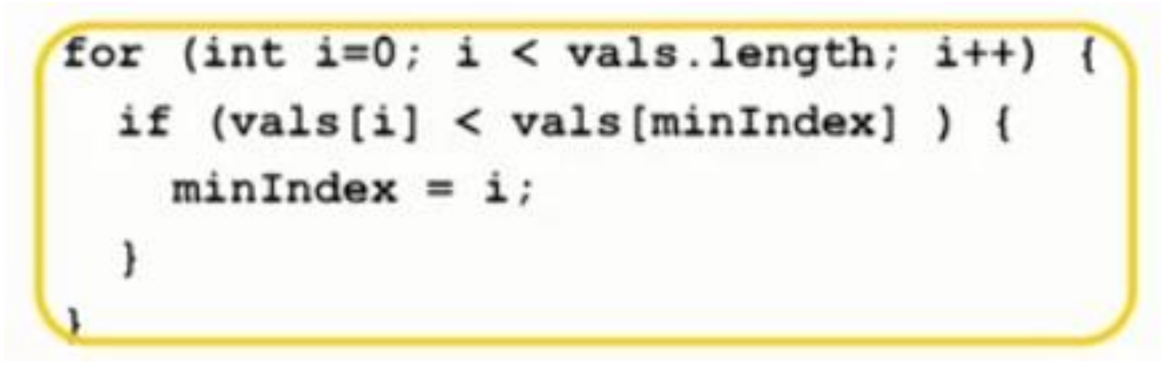


```
public static void reduce (int[] vals) {
  int minIndex =0;
  for (int i=0; i < vals.length; i++) {
    if (vals[i] < vals[minIndex] ) {
      minIndex = i;
    }
  }
  int minVal = vals[minIndex];
  for (int i=0; i < vals.length; i++) {
    vals[i] = vals[i] - minVal;
  }
}
```

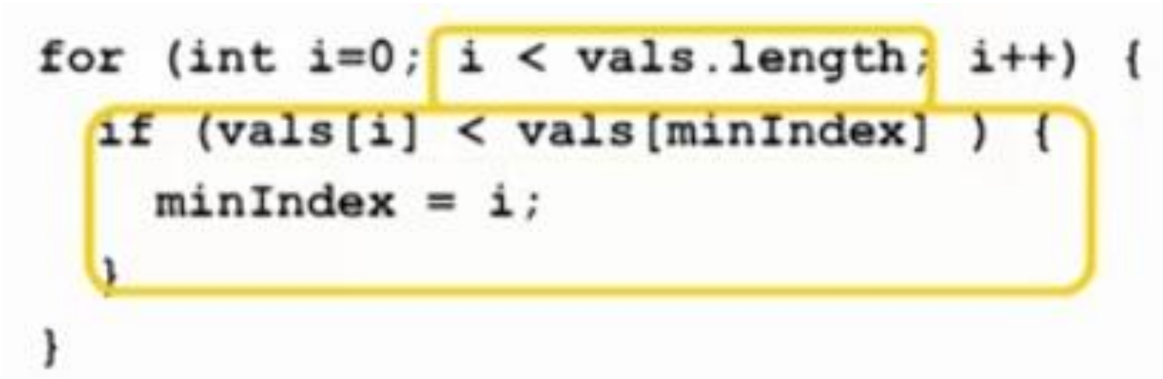


```
public static void reduce (int[] vals) {
  int minIndex =0;
  for (int i=0; i < vals.length; i++) {
    if (vals[i] < vals[minIndex] ) {
      minIndex = i;
    }
  }
  int minVal = vals[minIndex];
  for (int i=0; i < vals.length; i++) {
    vals[i] = vals[i] - minVal;
  }
}
```

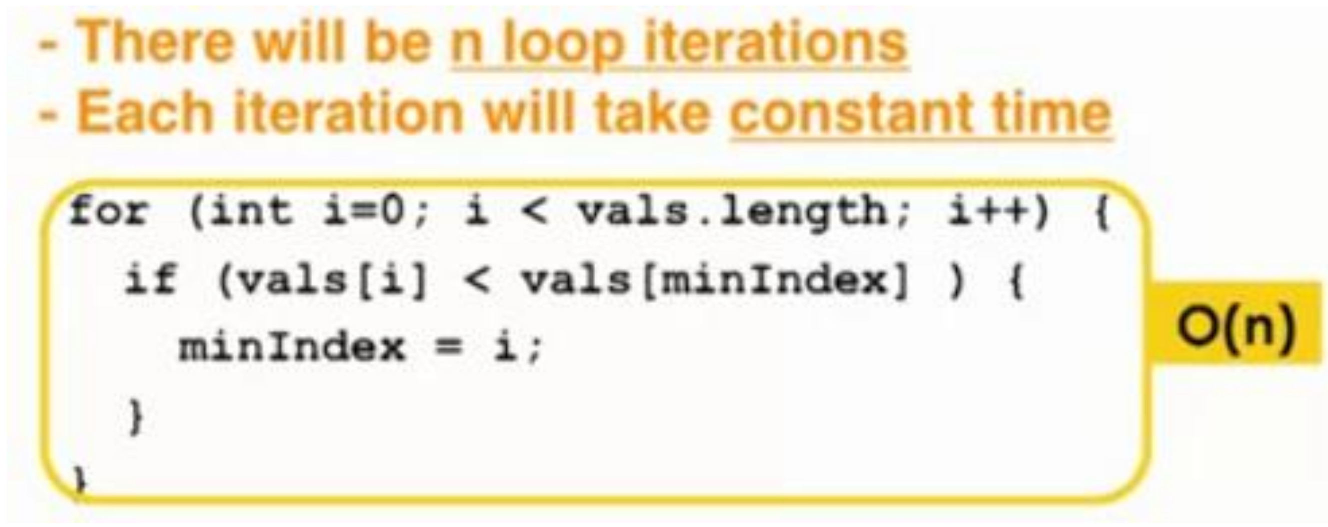
- These <u>don't</u> depend on the input size (n)



```
for (int i=0; i < vals.length; i++) {
  if (vals[i] < vals[minIndex] ) {
    minIndex = i;
  }
}
```



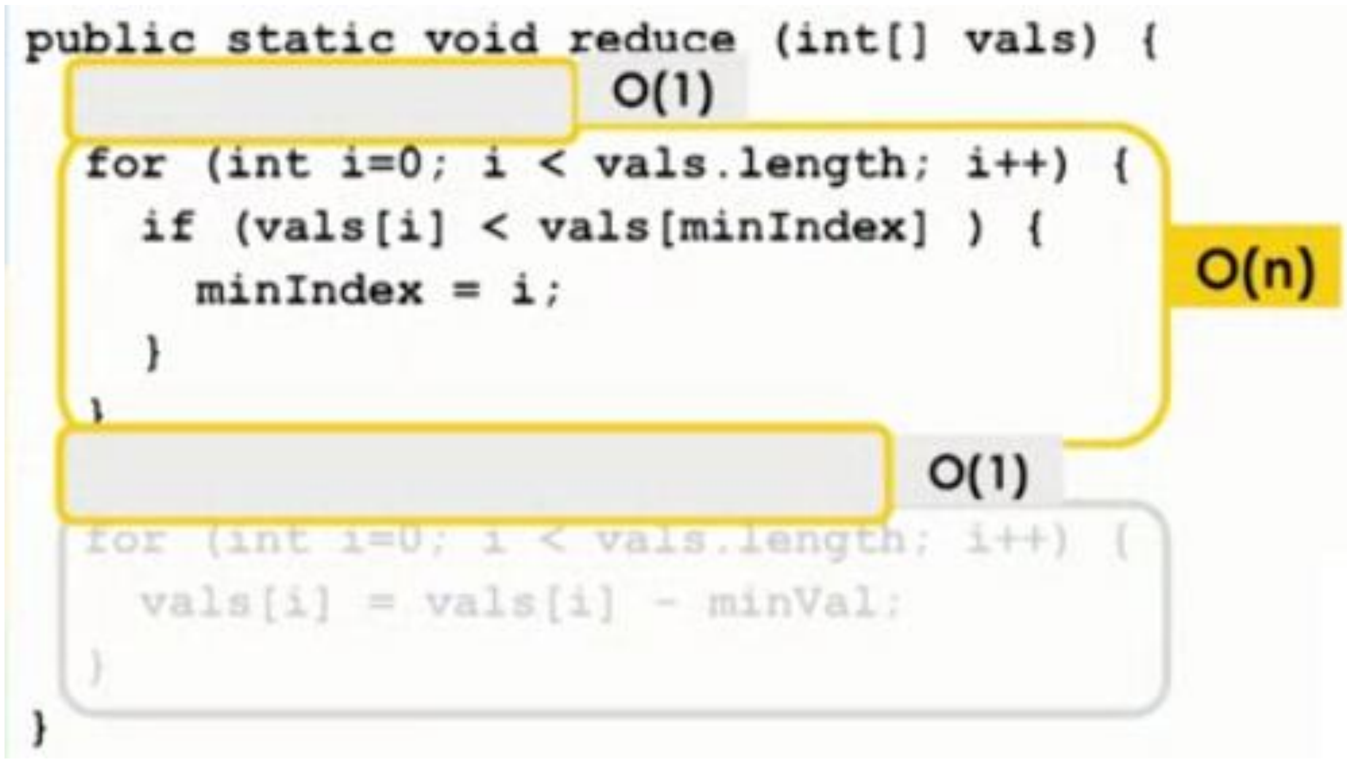
```
for (int i=0; i < vals.length; i++) {
  if (vals[i] < vals[minIndex] ) {
    minIndex = i;
  }
}
```



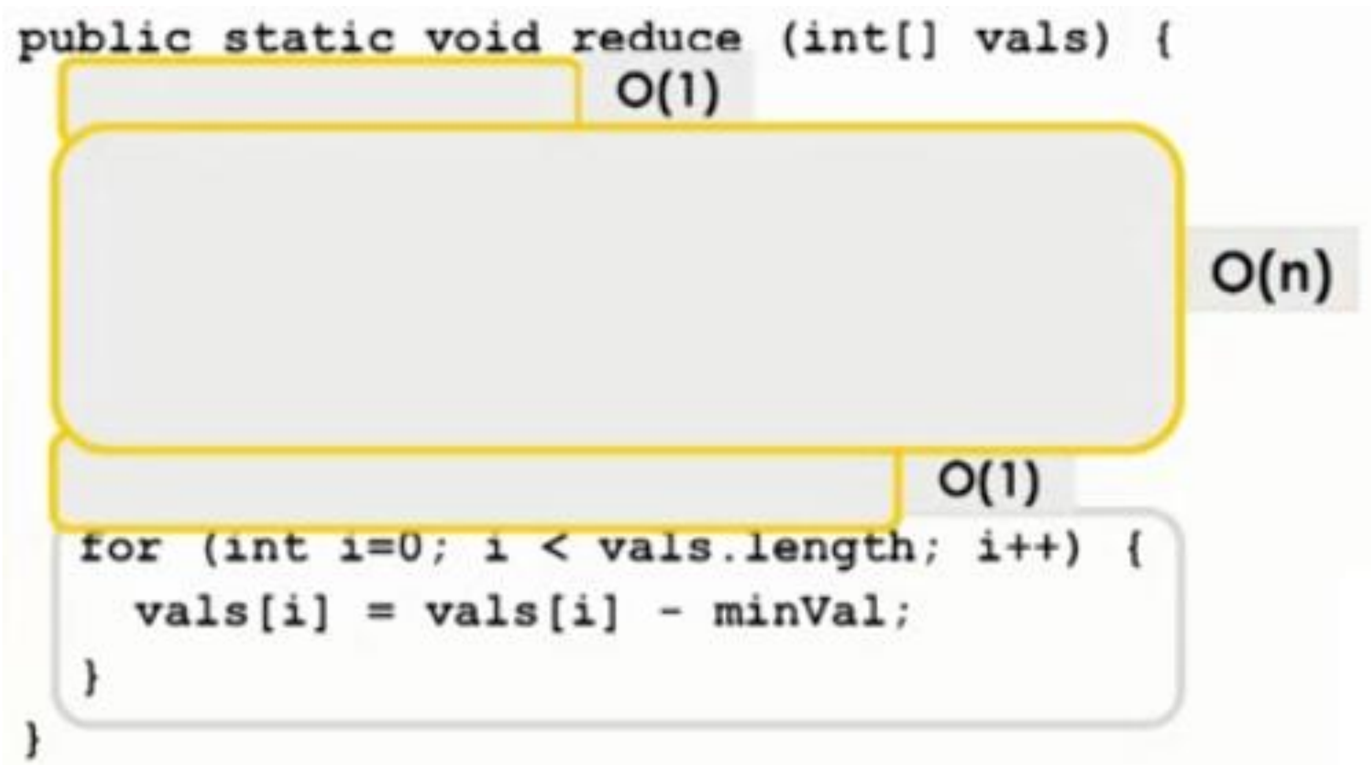
- There will be n loop iterations
- Each iteration will take constant time

```
for (int i=0; i < vals.length; i++) {
  if (vals[i] < vals[minIndex] ) {
    minIndex = i;
  }
}
```

O(n)



```java
public static void reduce (int[] vals) {

  for (int i=0; i < vals.length; i++) {
    if (vals[i] < vals[minIndex] ) {
      minIndex = i;
    }
  }

  for (int i=0; i < vals.length; i++) {
    vals[i] = vals[i] - minVal;
  }
}
```



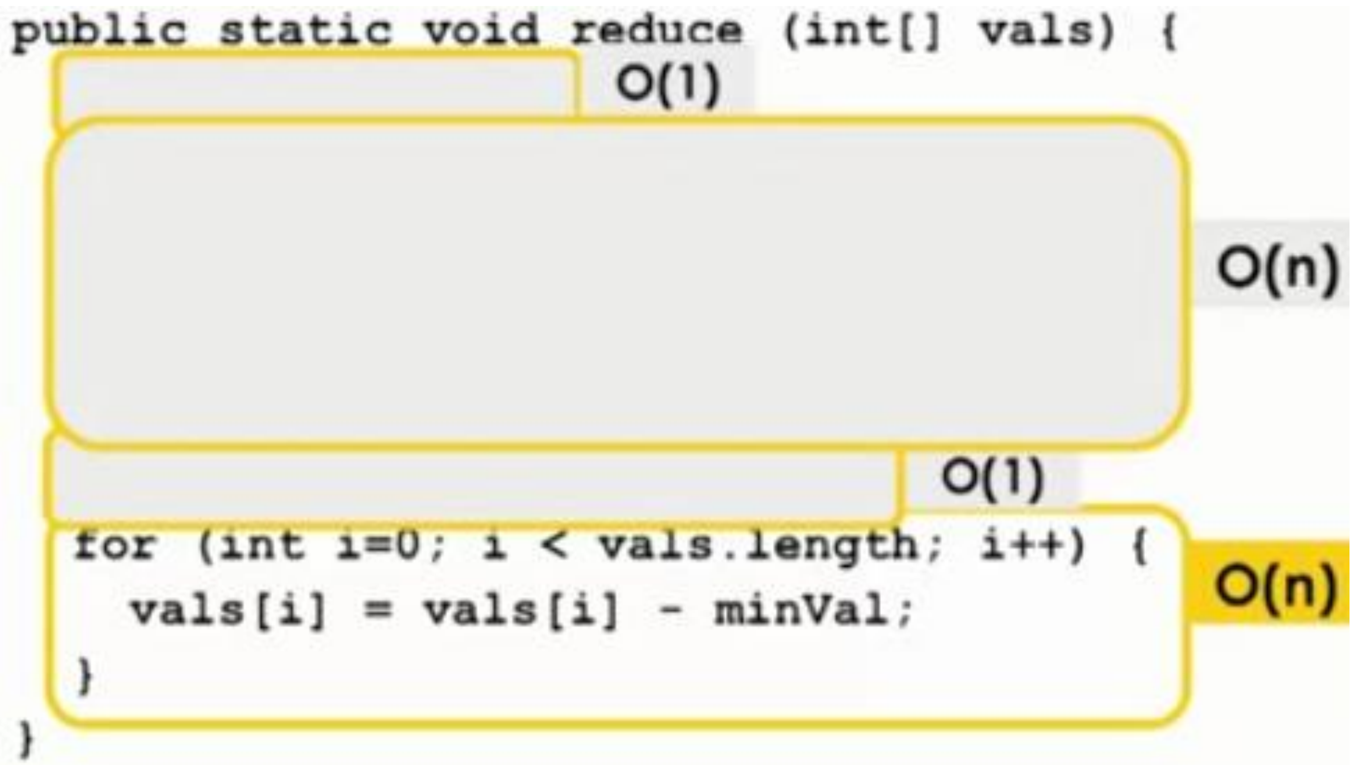
```
public static void reduce (int[] vals) {
                                O(1)

                                                          O(n)

                                             O(1)
  for (int i=0; i < vals.length; i++) {
    vals[i] = vals[i] - minVal;
  }
}
```

```
for (int i=0; i < vals.length; i++) {
  vals[i] = vals[i] - minVal;
}
```

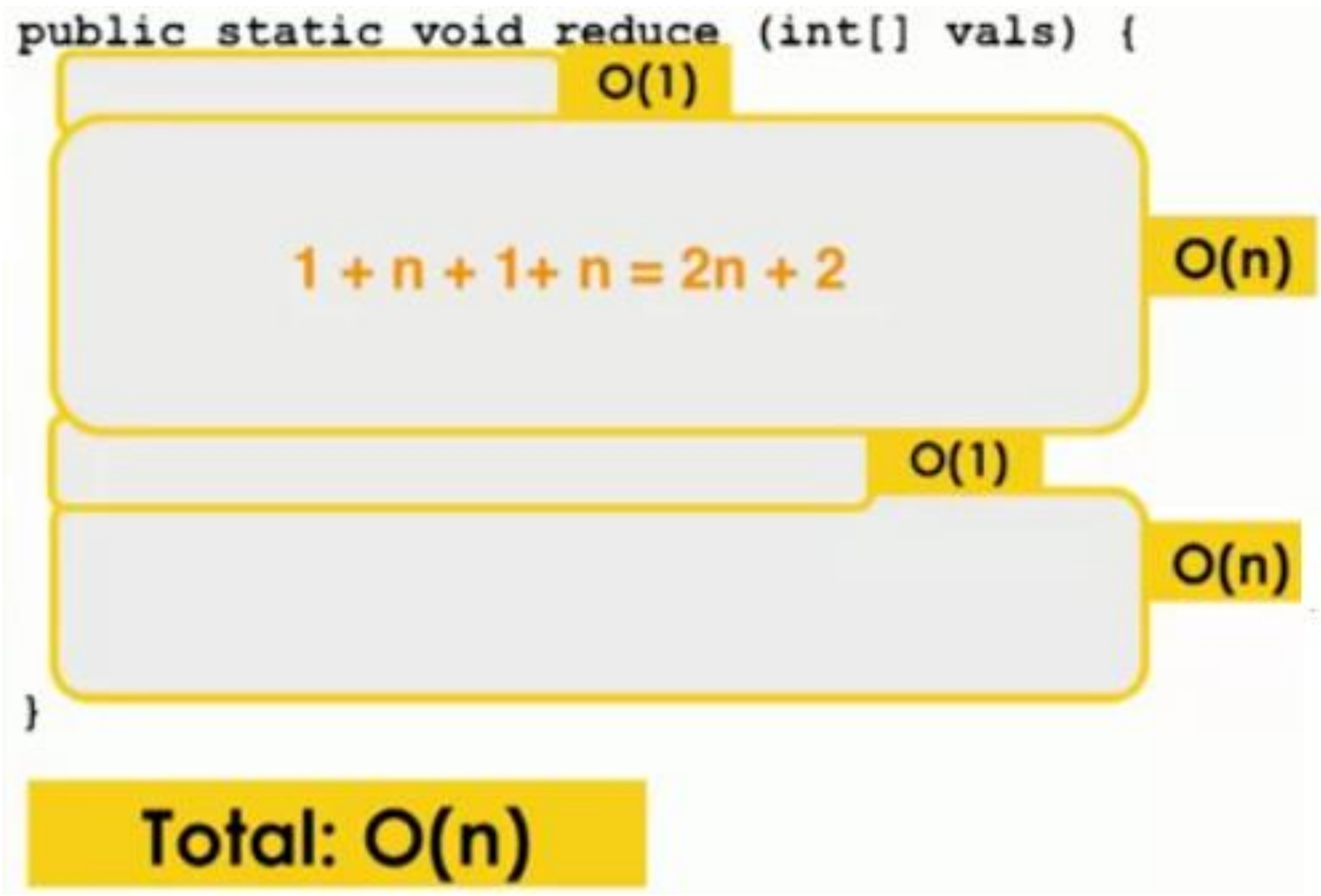
- Again, there will be <u>n loop iterations</u>
- Each iteration will take <u>constant time</u>

```
for (int i=0; i < vals.length; i++) {
  vals[i] = vals[i] - minVal;
}
```

O(n)

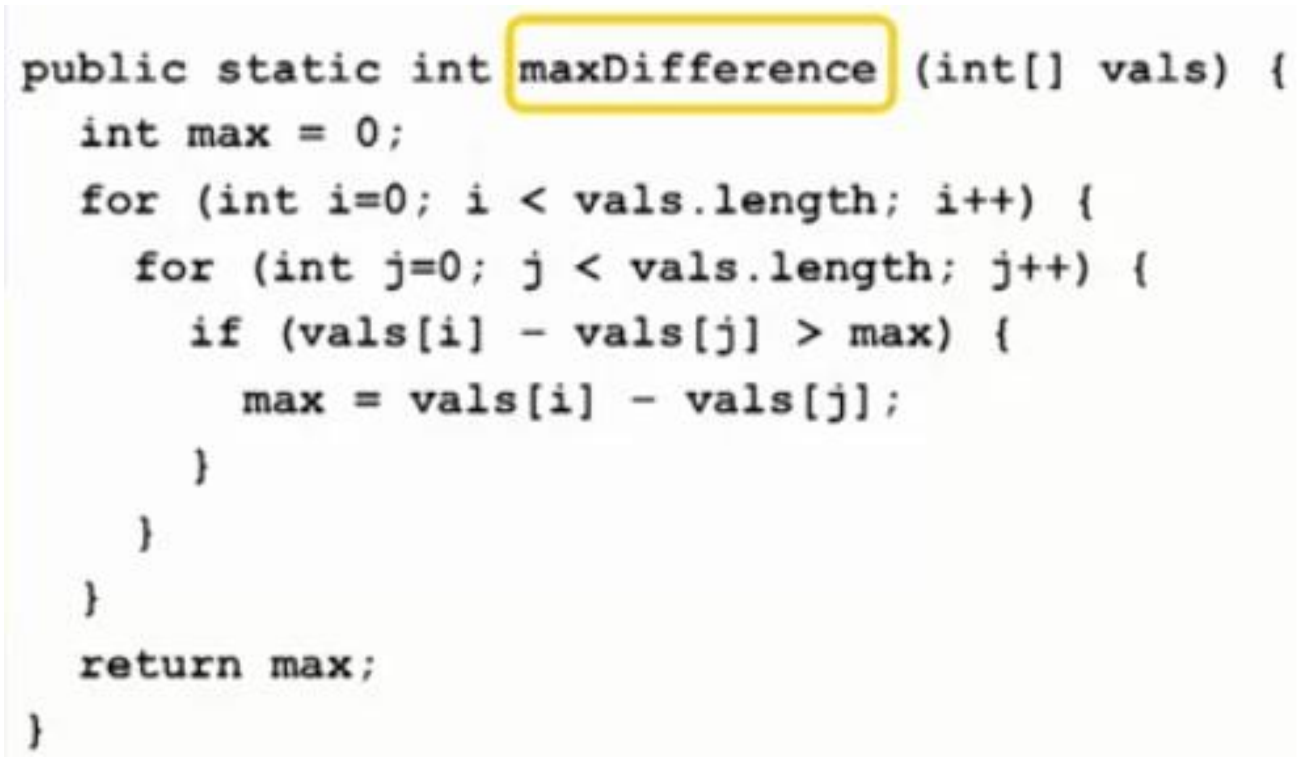


```
public static void reduce (int[] vals) {
  for (int i=0; i < vals.length; i++) {
    vals[i] = vals[i] - minVal;
  }
}
```

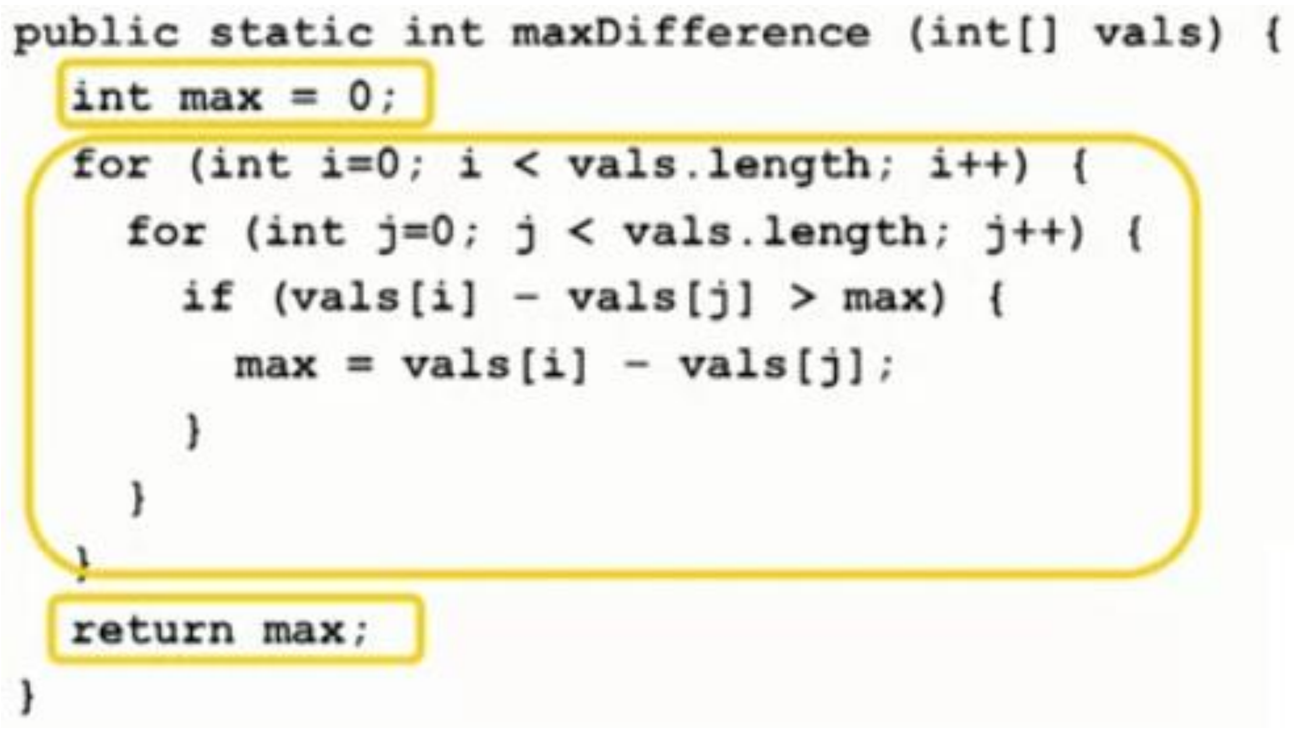


```
public static void reduce (int[] vals) {

}
```

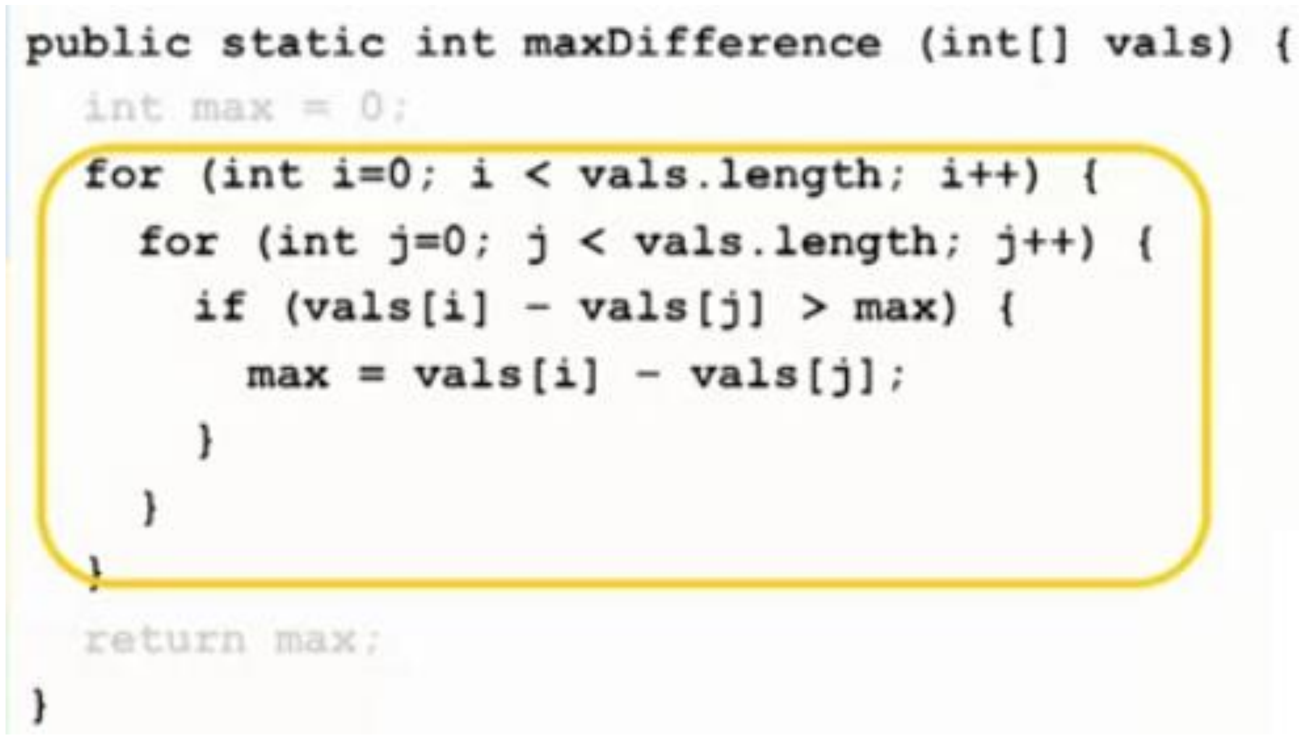
Total: O(n)



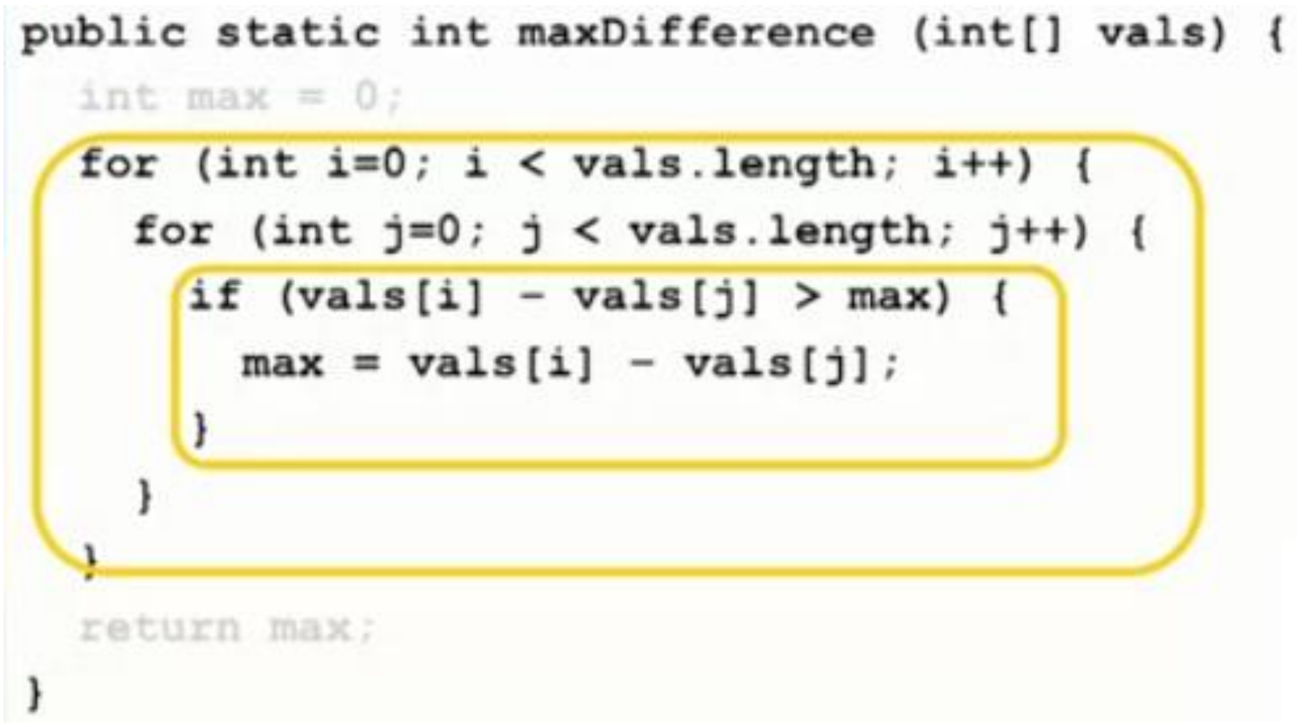
```
public static int maxDifference (int[] vals) {
  int max = 0;
  for (int i=0; i < vals.length; i++) {
    for (int j=0; j < vals.length; j++) {
      if (vals[i] - vals[j] > max) {
        max = vals[i] - vals[j];
      }
    }
  }
  return max;
}
```



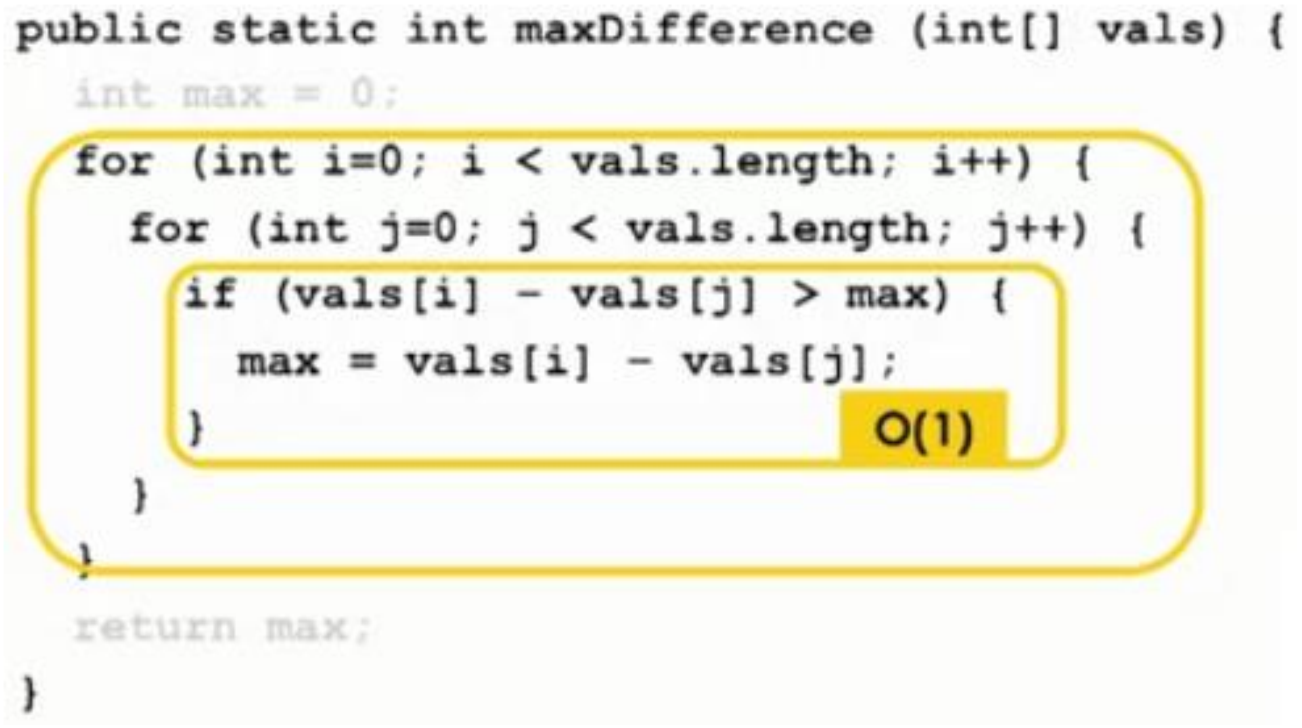
```
public static int maxDifference (int[] vals) {
  int max = 0;
  for (int i=0; i < vals.length; i++) {
    for (int j=0; j < vals.length; j++) {
      if (vals[i] - vals[j] > max) {
        max = vals[i] - vals[j];
      }
    }
  }
  return max;
}
```



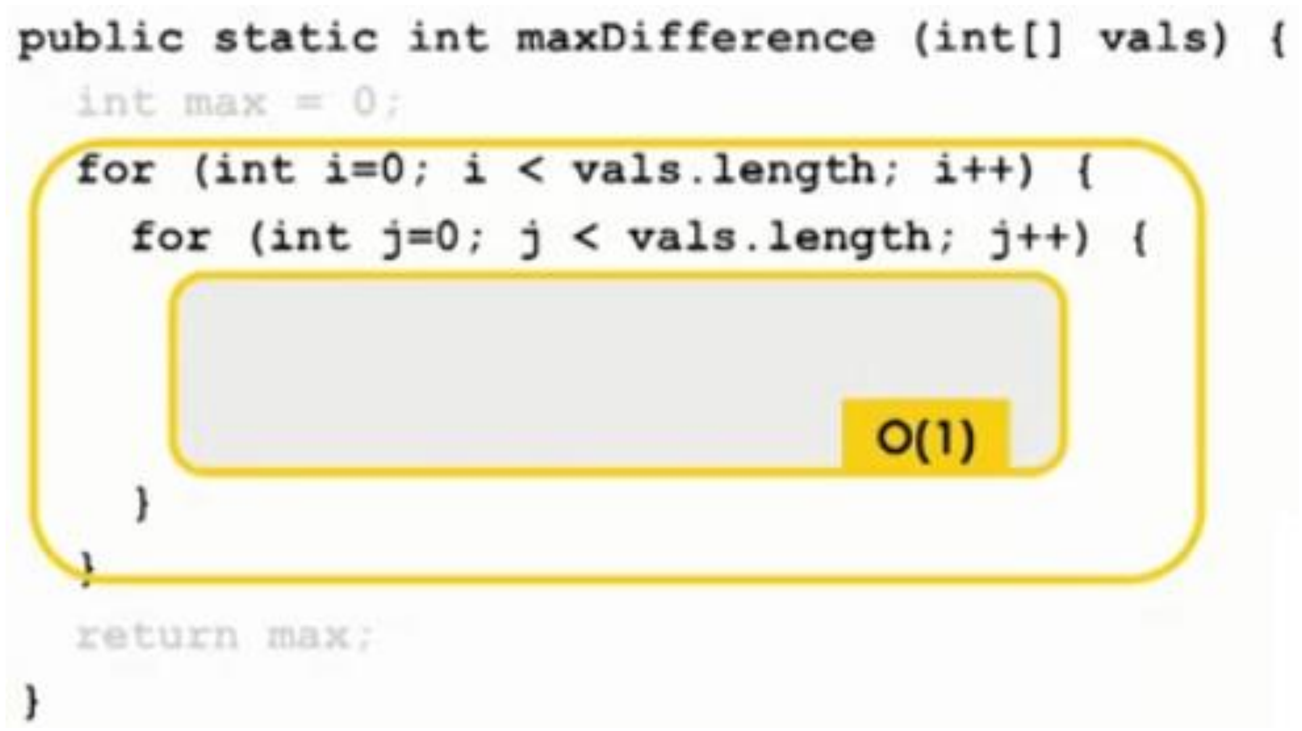
```java
public static int maxDifference (int[] vals) {
  int max = 0;
  for (int i=0; i < vals.length; i++) {
    for (int j=0; j < vals.length; j++) {
      if (vals[i] - vals[j] > max) {
        max = vals[i] - vals[j];
      }
    }
  }
  return max;
}
```



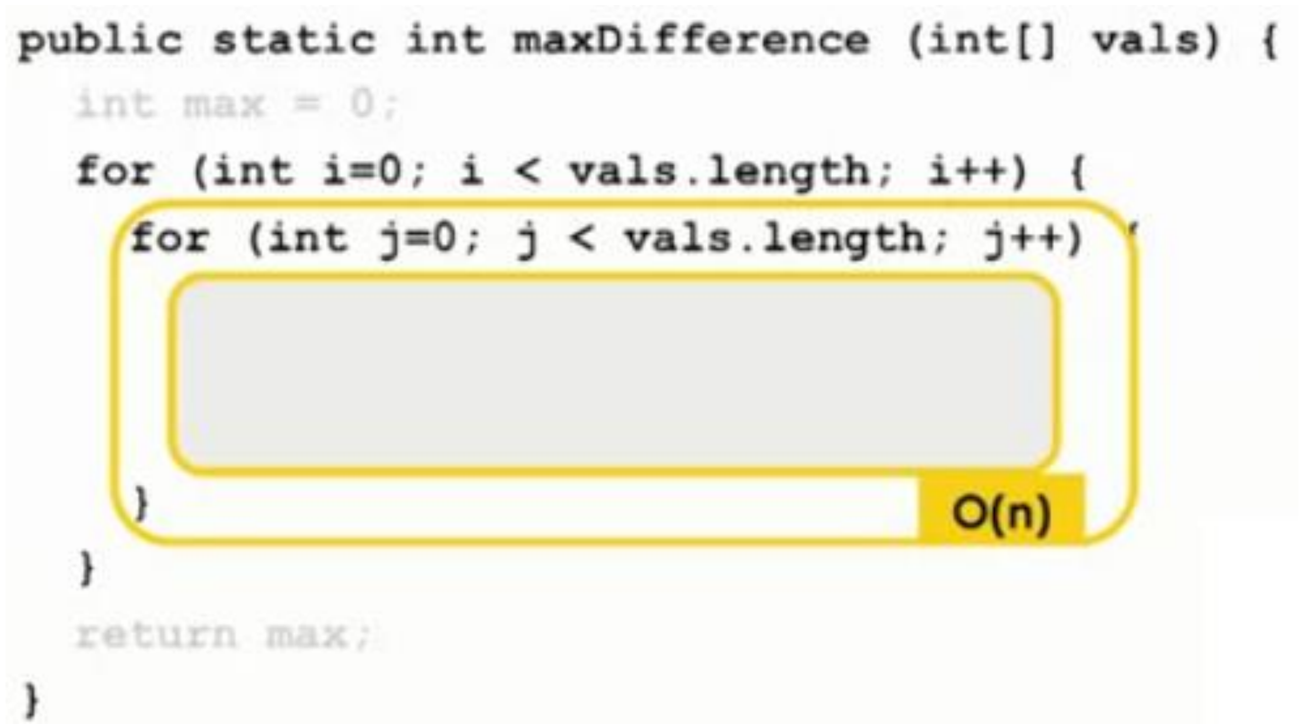
```java
public static int maxDifference (int[] vals) {
  int max = 0;
  for (int i=0; i < vals.length; i++) {
    for (int j=0; j < vals.length; j++) {
      if (vals[i] - vals[j] > max) {
        max = vals[i] - vals[j];
      }
    }
  }
  return max;
}
```



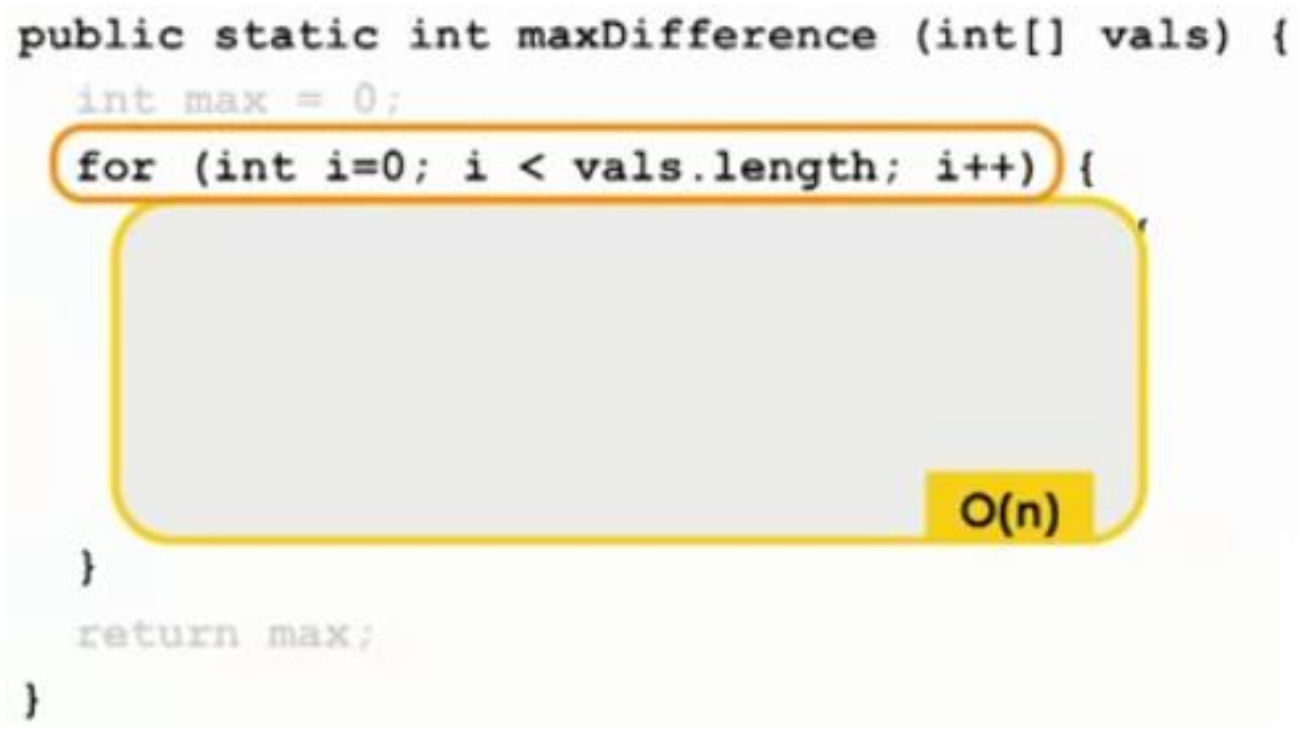
```
public static int maxDifference (int[] vals) {
  int max = 0;
  for (int i=0; i < vals.length; i++) {
    for (int j=0; j < vals.length; j++) {
      if (vals[i] - vals[j] > max) {
        max = vals[i] - vals[j];
      }
    }
  }
  return max;
}
```



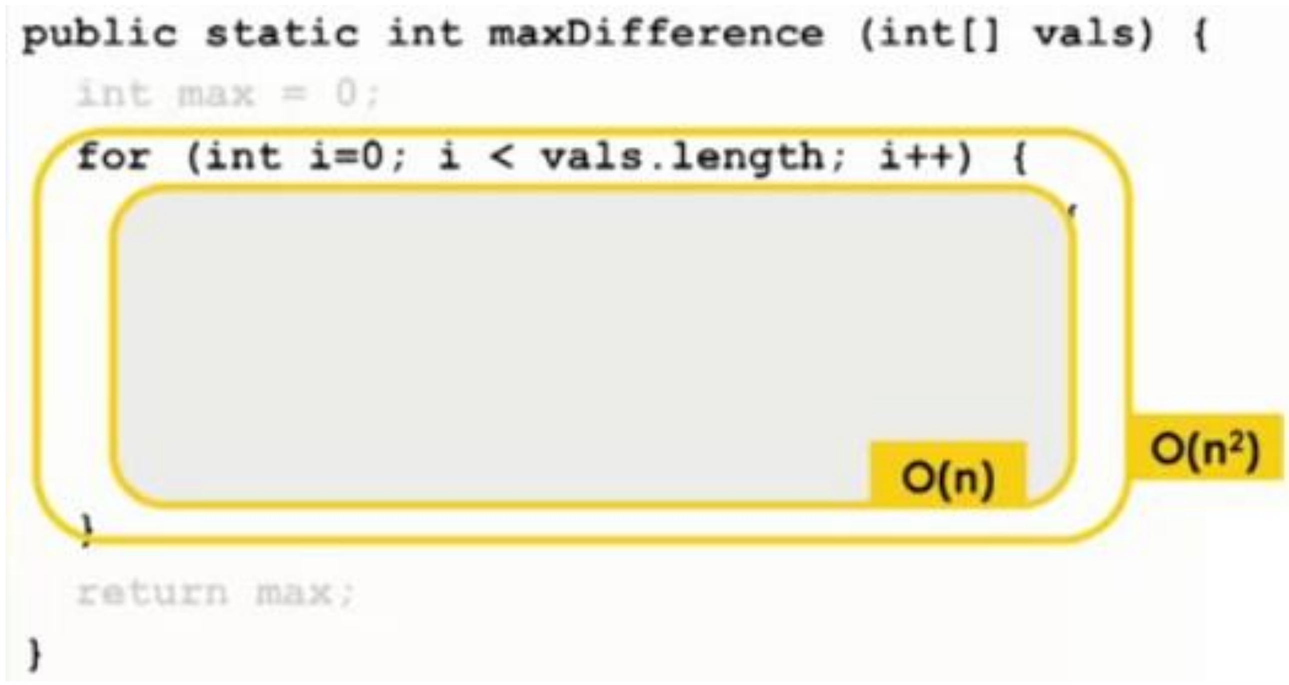
```
public static int maxDifference (int[] vals) {
  int max = 0;
  for (int i=0; i < vals.length; i++) {
    for (int j=0; j < vals.length; j++) {



    }
  }
  return max;
}
```



```
public static int maxDifference (int[] vals) {
  int max = 0;
  for (int i=0; i < vals.length; i++) {
    for (int j=0; j < vals.length; j++) {

    }
  }
  return max;
}
```



```
public static int maxDifference (int[] vals) {
  int max = 0;
  for (int i=0; i < vals.length; i++) {

  }
  return max;
}
```



```java
public static int maxDifference (int[] vals) {
  int max = 0;
  for (int i=0; i < vals.length; i++) {

  }
  return max;
}
```

```
public static int maxDifference (int[] vals) {
  int max = 0;  O(1)

                                                   O(n²)

  return max;  O(1)
}
```

Some Examples

Determine the time complexity for the following algorithm.

```
count = 0;
for(i=0; i<10000; i++)
    count++;
```

Some Examples

Determine the time complexity for the following algorithm.

```
count = 0;
for(i=0; i<10000; i++)
    count++;
```

$O(1)$

Some Examples

Determine the time complexity for the following algorithm.

```
count = 0;
for(i=0; i<n; i++)
    count++;
```

Some Examples

Determine the time complexity for the following algorithm.

```
count = 0;
for(i=0; i<n; i++)
    count++;
```

$O(n)$

Some Examples

Determine the time complexity for the following algorithm.

```
sum = 0;
for(i=0; i<n; i++)
    for(j=0; j<n; j++)
        sum += arr[i][j];
```

Some Examples

Determine the time complexity for the following algorithm.

```
sum = 0;
for(i=0; i<n; i++)
    for(j=0; j<n; j++)
        sum += arr[i][j];
```

$O(\mathbf{n}^2)$

Some Examples

Determine the time complexity for the following algorithm.

```
sum = 0;
for(i=1; i<=n; i=i*2)
    sum += i;
```

Some Examples

Determine the time complexity for the following algorithm.

```
sum = 0;
for(i=1; i<=n; i=i*2)
    sum += i;
```

O **(lg n)**

WHY? Show mathematical analysis to prove that it is indeed O **(lg n)**

Some Examples

Determine the time complexity for the following algorithm.

```
sum = 0;
for(i=1; i<=n; i=i*2)
    for(j=1; j<=n; j++)
        sum += i*j;
```

Some Examples

Determine the time complexity for the following algorithm.

```
sum = 0;
for(i=1; i<=n; i=i*2)
    for(j=1; j<=n; j++)
        sum += i*j;
```

***O* (n lg n)**

Why? Outer for loop runs *O*(lg n) times (prove it!) and for each iteration of outer loop, the inner loop runs *O*(n) times

Big-O Notation

- We say $f(n) = 30000$ is in the *order of* 1, or $\boldsymbol{O}(\mathbf{1})$
  - Growth rate of 30000 is constant, that is, it is not dependent on problem size. $\rightarrow$ Constant time
- $f(n) = 30n + 8$ is in the *order of* n, or $\boldsymbol{O}(\boldsymbol{n})$
  - Growth rate of $30n + 8$ is roughly
    n. $\rightarrow$ Linear time
- $f(n) = n^2 + 1$ is in the *order of* n
  - Growth rate of $n^2 + 1$ is roughly pr
    n^2. $\rightarrow$ Quadratic time

In general, $O(n^{c1} (\lg n)^{c2})$ time is called **polynomial time** (here c1 and c2 are constants). For E.g. $O(n^2)$, $O(n^3)$, $O(1)$, $O(n \lg n)$, $O(n^2 \lg n)$, $O(n^3 (\lg n)^2)$, etc.

- In general, any $O(n^2)$ function is faster- growing than any $O(n)$ function.
  - For large n, a $O(n^2)$ algorithm runs a lot slower than a $O(n)$ algorithm.

- *$O(2^n)$, $O(4^n)$, $O(2^{n^2})$, etc.*, are called exponential times.

Polynomial & non-polynomial time algorithms

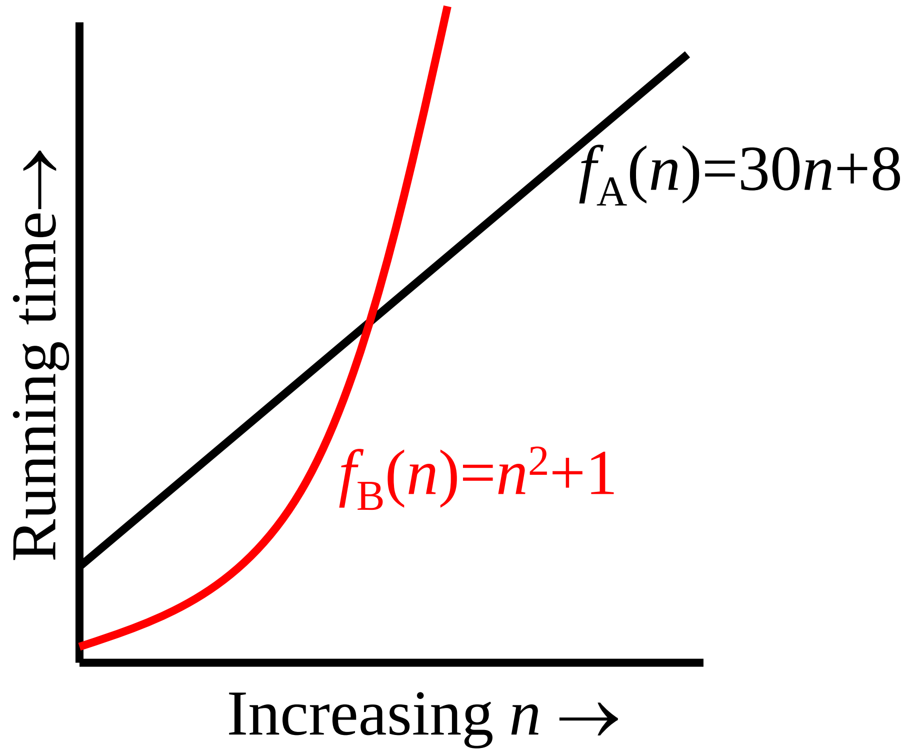
- Polynomial time algorithm: Algorithm whose worst-case running time is polynomial
  - Examples: Linear Search (in unsorted array): O(n), Binary Search (in sorted array): O(lg n), etc.
- Non-polynomial time algorithm: Algorithm whose worst-case running time is not polynomial
  - Examples: an algorithm to enumerate and print all possible orderings of n persons: O(*n*!), an algorithm to enumerate and print all possible binary strings of length n: O(2^n)
- Theoretically, polynomial algorithms are expected to be more efficient than non-polynomial or exponential algorithms but this is not always true in practice …

Are non-polynomial time algorithms always worse than polynomial time algorithms?

- Theoretically, yes; but not always true in practice. For e.g. consider algorithms A and B where
  - *A* is an $O(n^{1,000,000})$ algorithm
  - B is an $O(n^{\log \log \log n})$ algorithm
- **A**'s running time is *theoretically* polynomial, so we may expect it to be more efficient than **B** which is an exponential time algorithm. But practically **A** takes much longer time to finish than **B** which is *theoretically* exponential algorithm.
  - So there may exist a non-polynomial time algorithm which is better than a polynomial time algorithm

Visualizing Orders of Growth of Runtimes

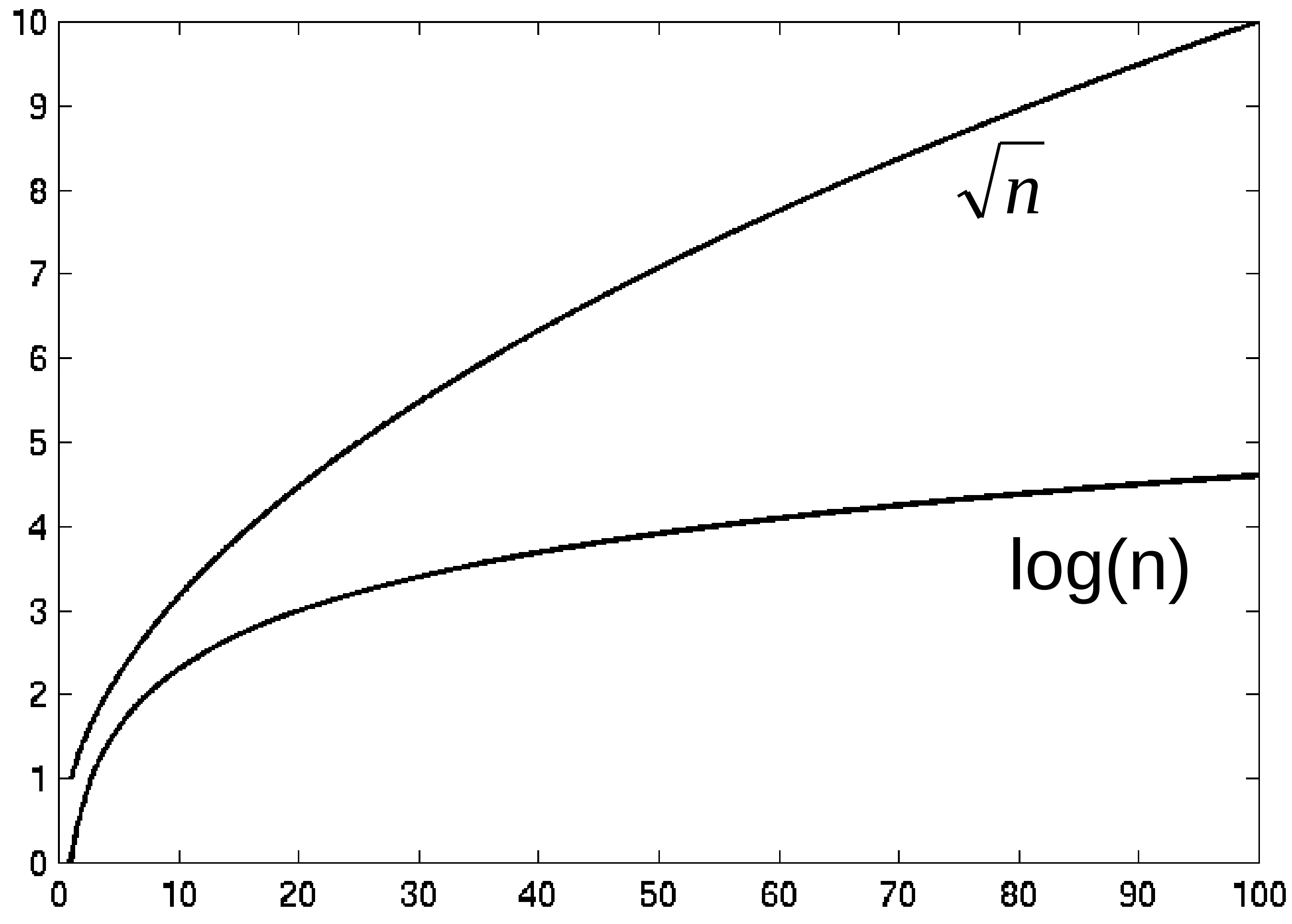
- On a graph, as you go to the right, a faster growing function eventually becomes larger.



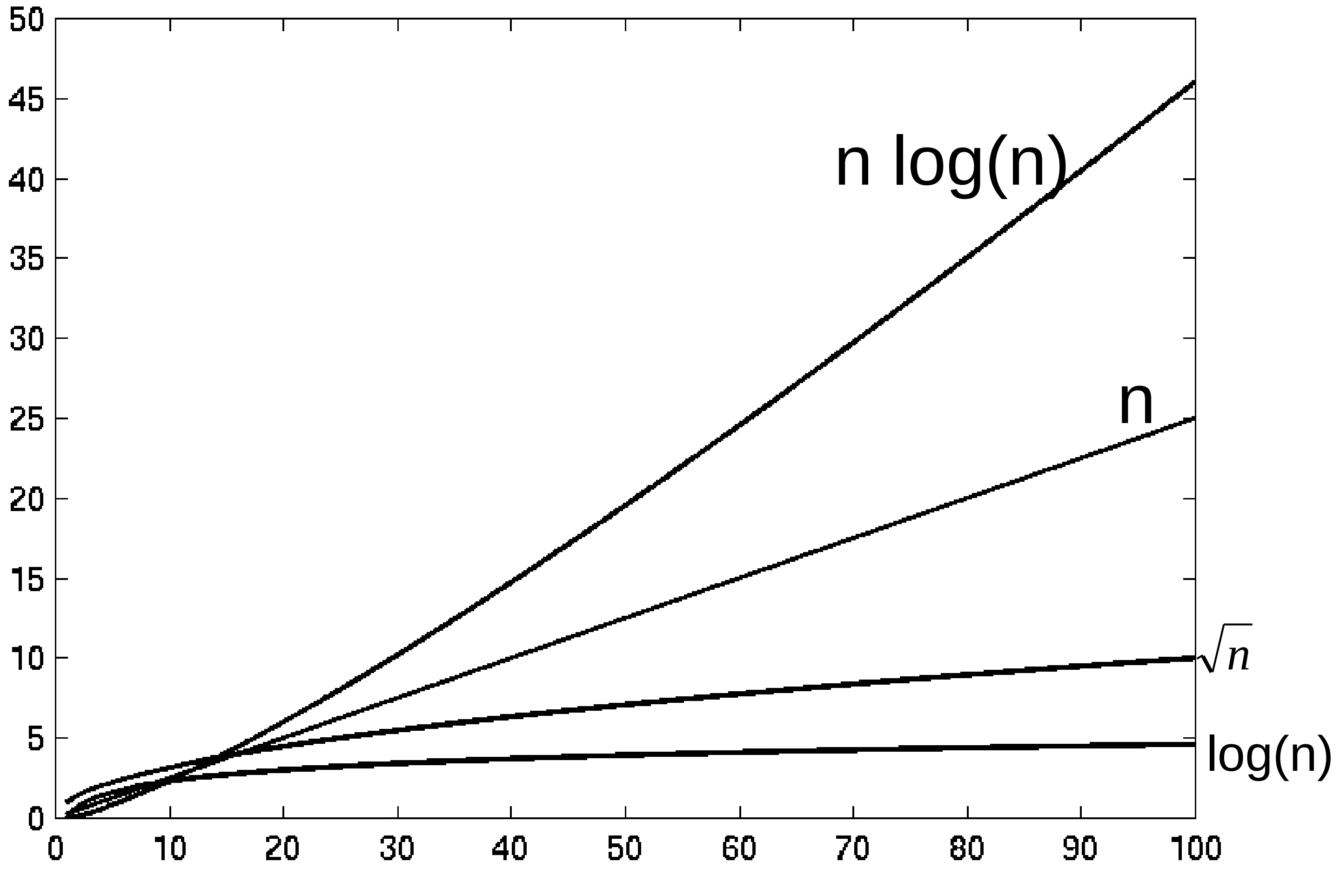
Growth of Functions

n	1	lgn	n	nlgn	n^2	n^3	2^n
1	1	0.00	1	0	1	1	2
10	1	3.32	10	33	100	1,000	1024
100	1	6.64	100	664	10,000	1,000,000	1.2×10^{30}
1000	1	9.97	1000	9970	1,000,000	10^9	1.1×10^{301}

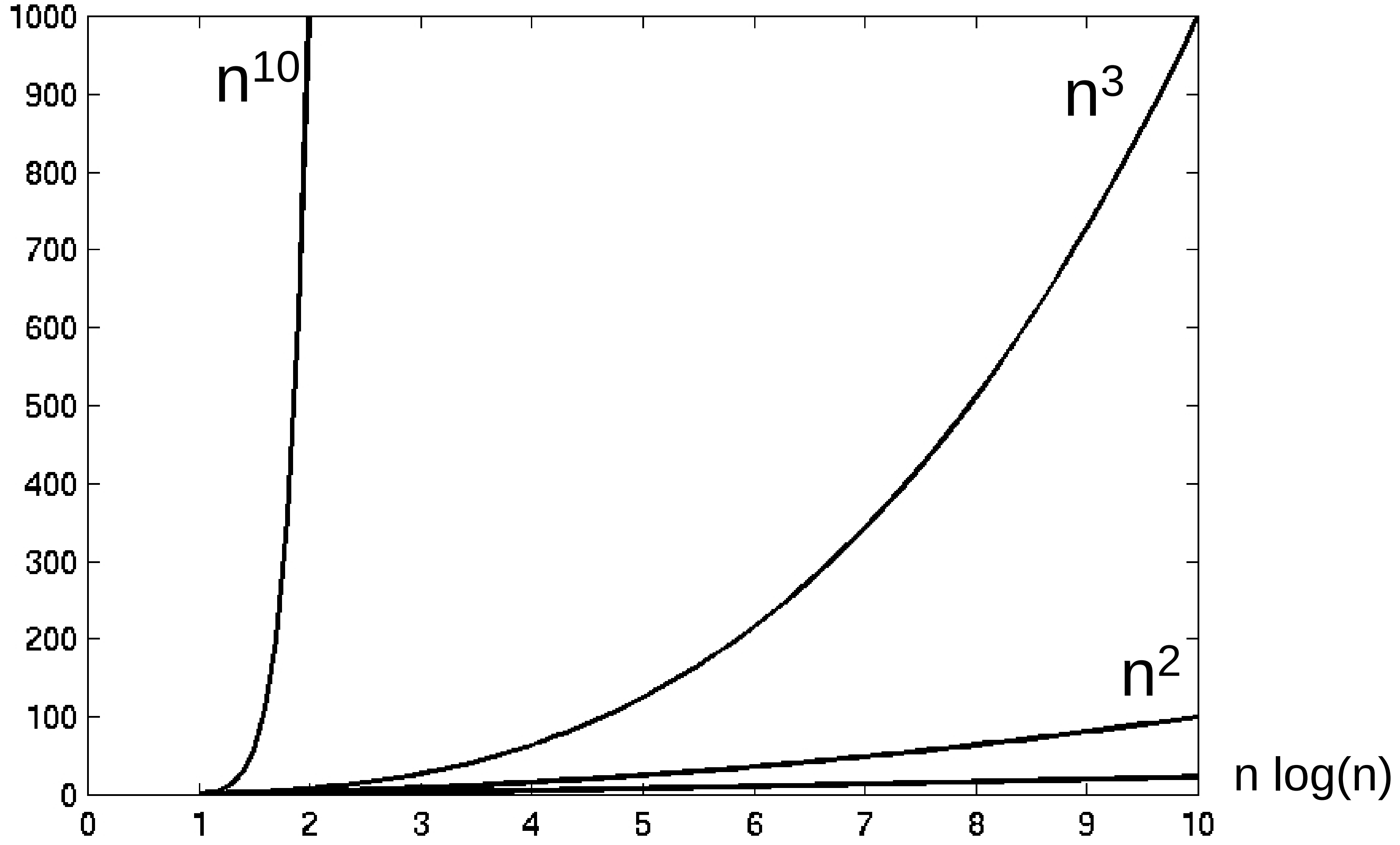
Complexity Graphs



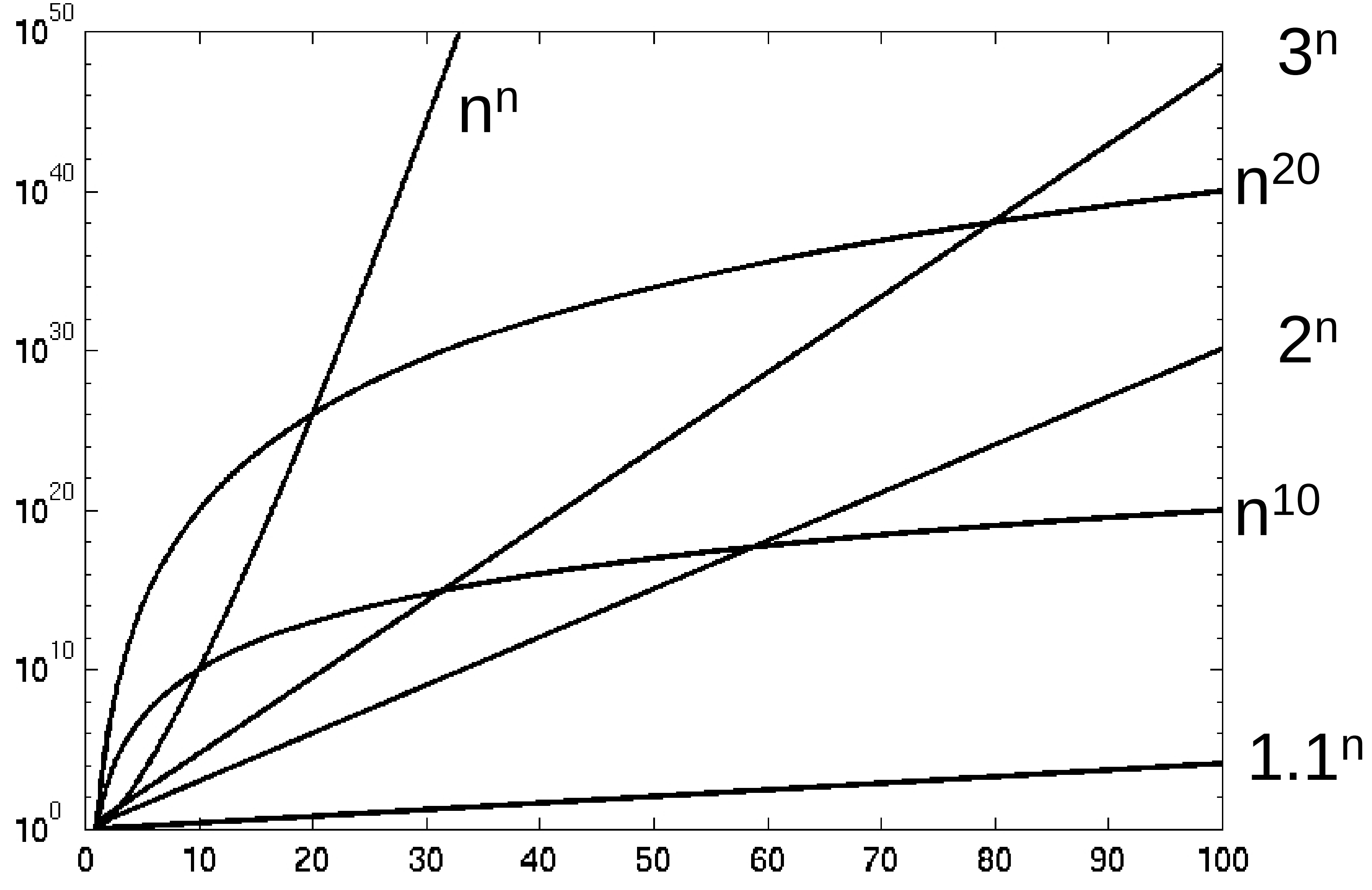
Complexity Graphs



Complexity Graphs



Complexity Graphs (log scale)


10^{50}
10^{40}
10^{30}
10^{20}
10^{10}
10^{0}
0
10
20
30
40
50
60
70
80
90
100
n^n
3^n
n^{20}
2^n
n^{10}
1.1^n