Big O Notation

Big-O Notation

Big-O notation is an assessment of an algorithm's efficiency. Big-O notation helps gauge the amount of work that is taking place.

Common Big-O Notations :

$O(1)$	**$O(\text{Log}_2 N)$**
$O(2^N)$	**$O(N^2)$**
$O(N \text{ Log}_2 N)$	**$O(N)$**
$O(\text{Log}_2 N)$	**$O(N^3)$**

Big-O

frequently used notations

Name	Notation
constant	$O(1)$
logarithmic	$O(\log_2 N)$
linear	$O(N)$
linearithmic	$O(N \log_2 N)$
quadratic	$O(N^2)$
exponential	$O(N^n)$

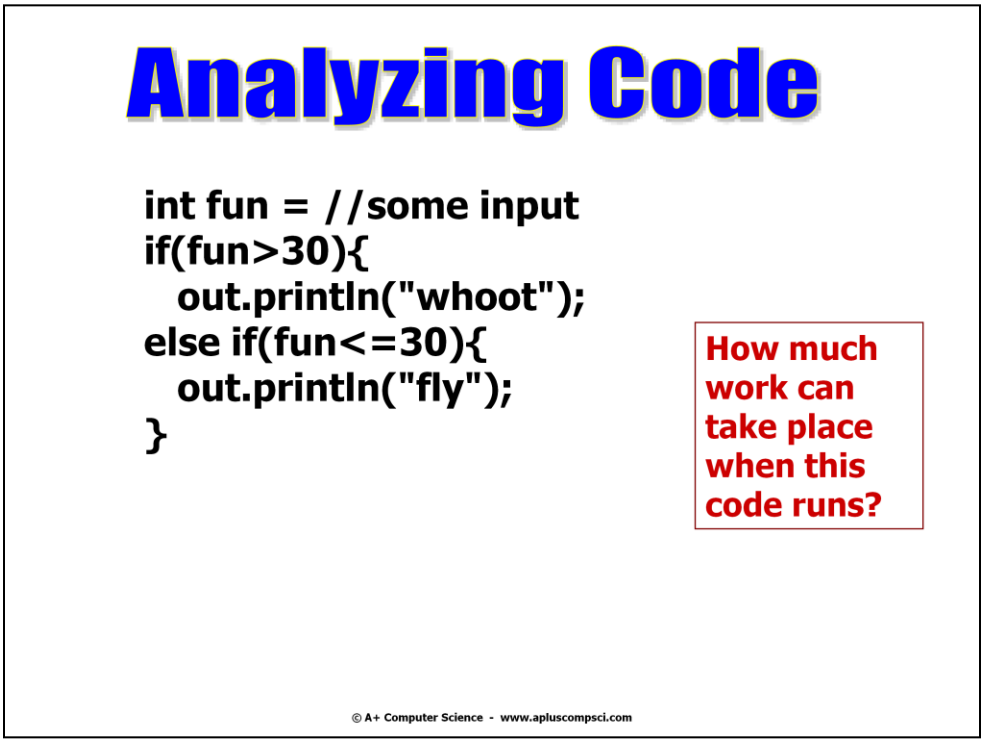
Big-O Notation

One of the main reasons for consulting Big-O is to make decisions about which algorithm to use for a particular job.

If you are designing a program to sort 2 trillion data base records, writing an N^2 sort instead of taking the time to design and write an N*LogN sort, could cost you your job.

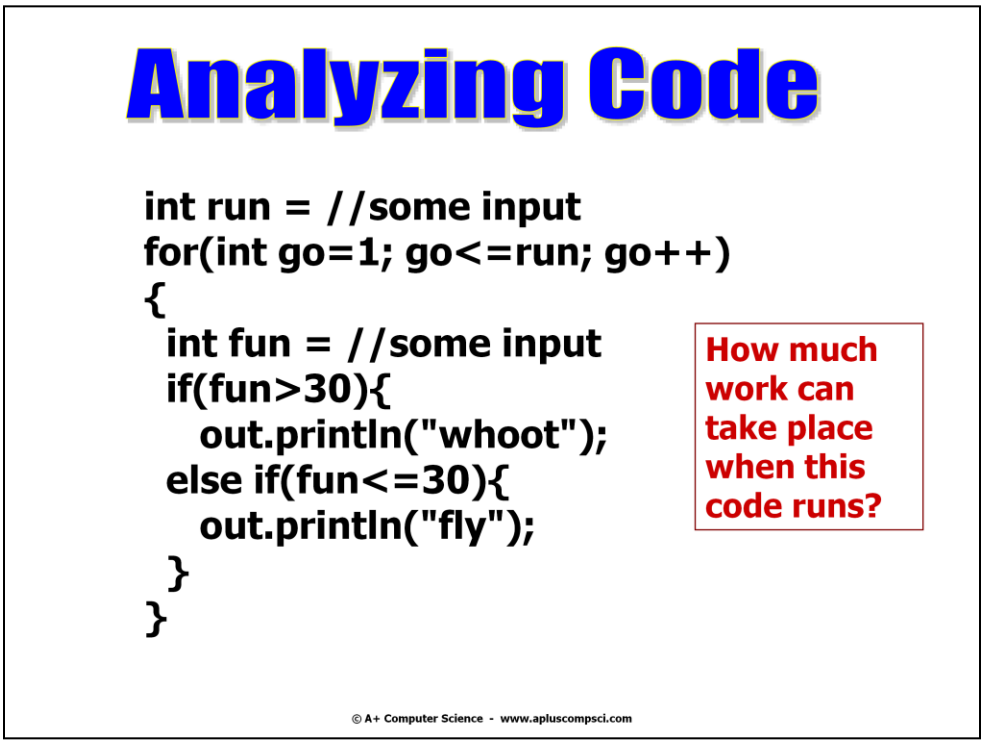
Analyzing Code

In order to properly apply a BigO notation, it is important to analyze a piece of code to see what the code is doing and how many times it is doing it.



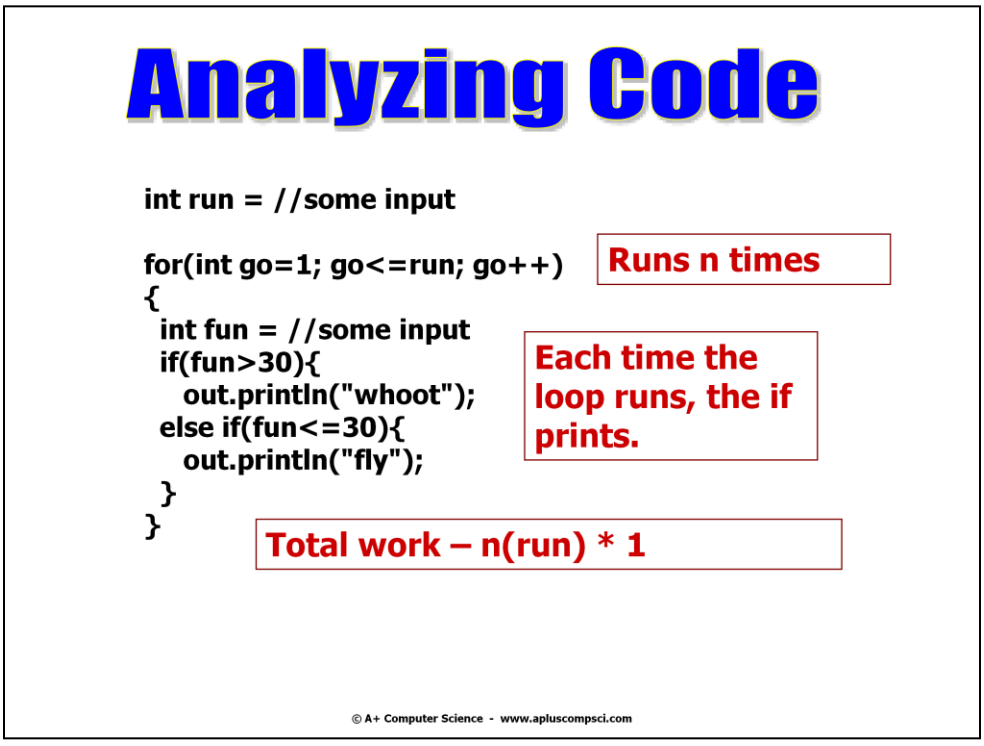
In the example above, the if else statement will only print out one time. The if will print `whoot` if fun is greater than 30. The if will print out `fly` if fun is less than or equal to 30. The code above can only do one thing each time it is executed.

Total work = 1



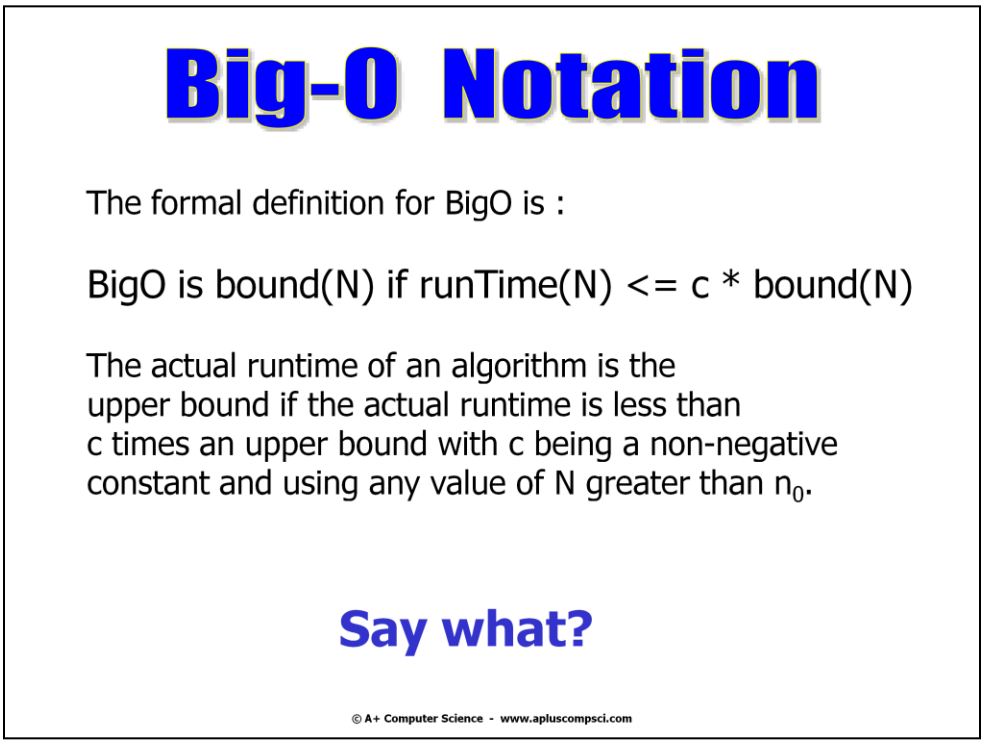
In the example above, the for loop will execute run times. The if statement will execute each time the for loop iterates. The if else statement will only print out one time. The if will print `whoot` if fun is greater than 30. The if will print out `fly` if fun is less than or equal to 30.

Total work = run * 1



In the example above, the for loop will execute run times. The if statement will execute each time the for loop iterates. The if else statement will only print out one time. The if will print `whoot` if fun is greater than 30. The if will print out `fly` if fun is less than or equal to 30.

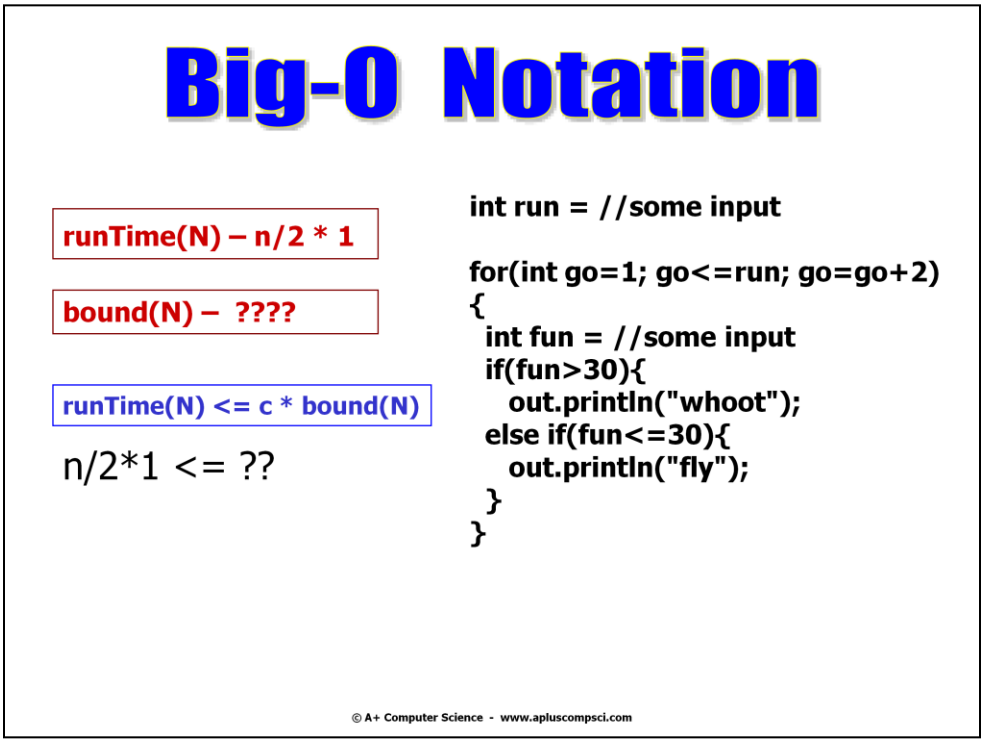
Total work = run * 1



First, we calculate the actual runtime (how much actual work is going on) for the code we are analyzing. The most common way to determine the actual runtime is by looking at the code and determining how many times the code prints, adds, iterates, or does any type of real work.

After the actual run time is known, an upper bound needs to be determined. Many times the upper bound is very obvious and other times it is less obvious.

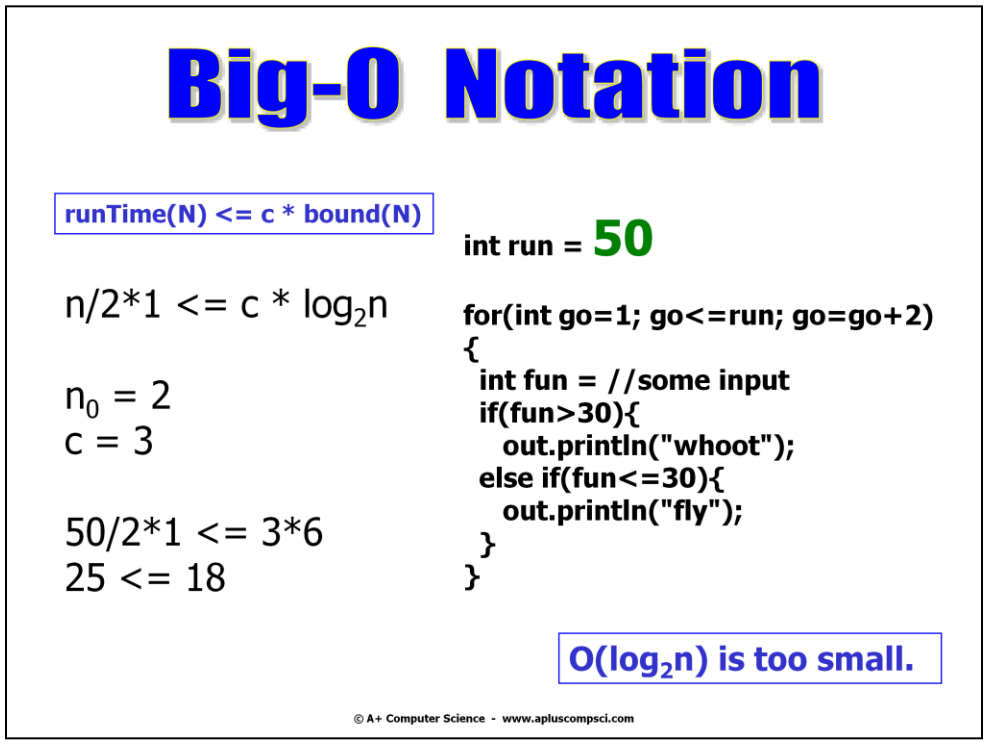
The formula above allows the actual runtime to be compared to the upper bound to determine if it is appropriate. When proofing an upper bound, c and n_0 are constant values. n_0 gives a point from which to pick N as N must be larger than n_0 c is used as a multiplier for bound(N).



First, we calculate the actual runtime for the code above. The loop runs n/2 times. Each time the loop iterates it prints either `whoot` or `fly`. The loop iterates n/2 times which equals n/2 units of work. For each of the n/2 iterations, the loop performs one print which is equal to 1 unit of work.

Actual run time = n/2*1

Now that the actual run time is known, an upper bound needs to be chosen so that the formula can be tested.



Actual run time = n/2*1

Now that the actual run time is known, lets proof the formula by picking an upper bound of $\log_2 n$.

For the formula, we need a value for c, n_0, and a value for N.

If c is 3 and n_0 is 2, we can pick 8 for N and the formula would be

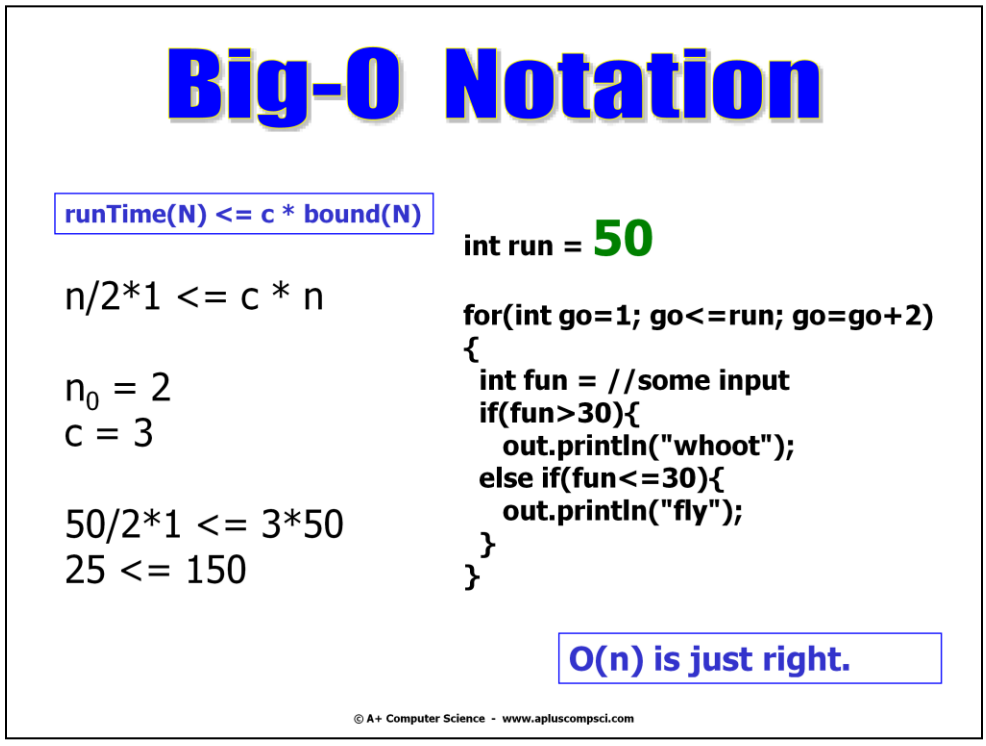
8/2*1 <= 3*3 4<=9

This looks pretty good so far.

If c is 3 and n_0 is 2, we can pick 45 for N and the formula would be

45/2*1 <= 3*6 22<=18

This does not look so good.



Actual run time = n/2*1

Now that the actual run time is known, lets proof the formula by picking and upper bound of n.

For the formula, we need a value for c, n_0, and a value for N^2.

If c is 3 and n_0 is 2, we can pick 8 for N and the formula would be

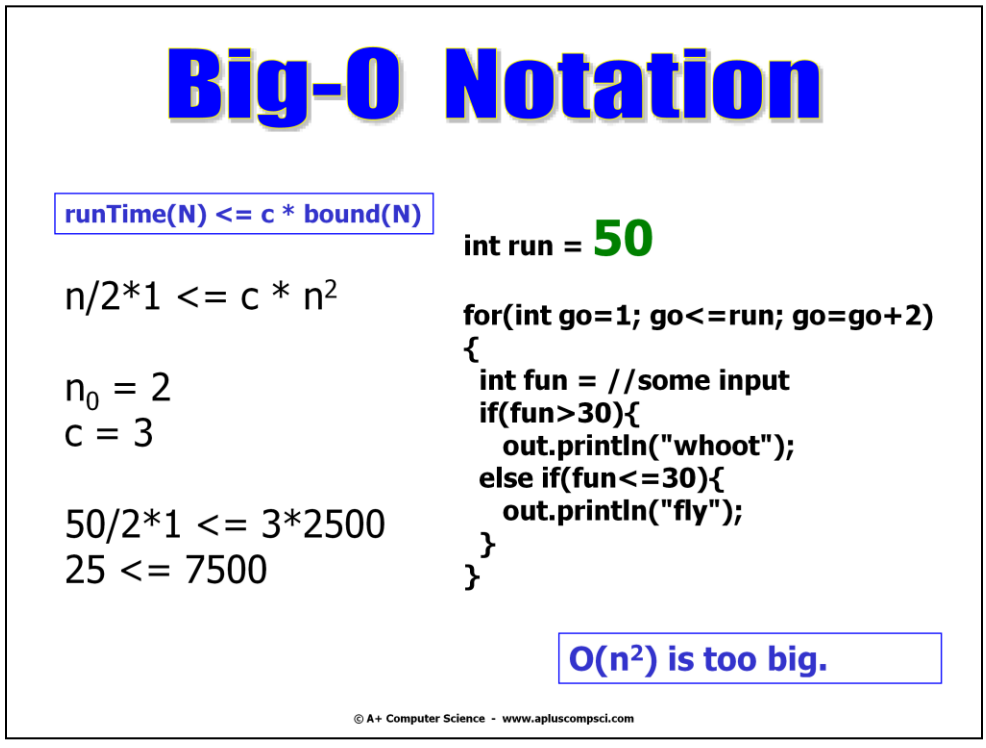
8/2*1 <= 3*8 4<=24

This looks pretty good so far.

If c is 3 and n_0 is 2, we can pick 45 for N and the formula would be

45/2*1 <= 3*45 22<=135

This still looks pretty good.



Actual run time = n/2*1

Now that the actual run time is known, lets proof the formula by picking and upper bound of n.

For the formula, we need a value for c, n_0, and a value for N.

If c is 3 and n_0 is 2, we can pick 8 for N and the formula would be

8/2*1 <= 3*8*8 4<=192

This looks okay as the 4 is less than 192, but seems a bit excessive.

If c is 3 and n_0 is 2, we can pick 45 for N and the formula would be

45/2*1 <= 3*45*45 22<=6075

This looks okay as the 22 is less than 6075, but seems way beyond what is needed.

N^2 will not work as it is not the most restrictive bound that could be used.

Big-O Notation

The BigO determined for a section of code should be the most restrictive BigO possible so that the BigO grows at a faster rate than the actual runtime of the code.

For the previous example, N is the most appropriate BigO as it meets the criteria and is the most restrictive BigO that would match the formal definition.

Now it is time for another round of ....

What is the Big-O?

What is the Big-O?

```
int n = ray.size();
for(int i=0; i<n; i++)
  out.println( ray.get(i) );
```

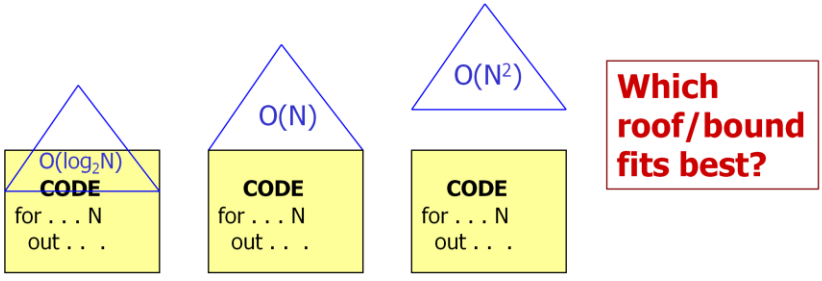
ray is an ArrayList!

This one is clearly N as we access all N items.

Big O Notation – O(N)

What is the Big-O?

```
int n = ray.size();
for(int i=0; i<n; i++)
   out.println( ray.get(i) );
```



Which roof/bound fits best?

What is the Big-O?

```
int n = ray.size();
for(int i=0; i<n; i+=2)
  out.println( ray.get(i) );
```

ray is an ArrayList!

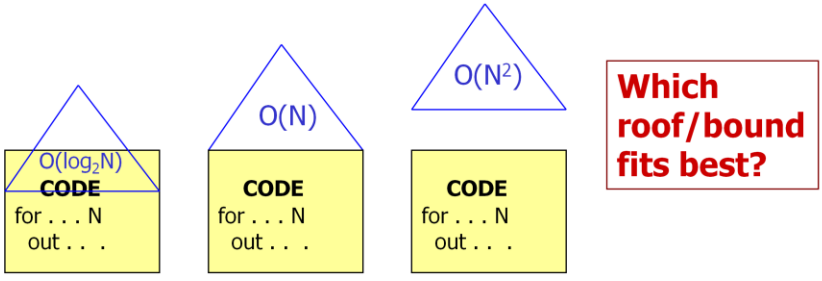
This example is not as easy as the last.

This code will print N/2 items.

Big O Notation – O(N)

What is the Big-O?

```
int n = ray.size();
for(int i=0; i<n; i+=2)
   out.println( ray.get(i) );
```



Big-O Notation

N/2*1 - N is the dominant term as N gets larger. Because N dominates the expression the constants can be dropped.

N/2*1 == N

What is the Big-O?

```
int n = ray.size();
for(int i=0; i<n; i++)
  for(int j=0; j<n;j++)
    out.println( ray.get(i) );
```

ray is an ArrayList!

Big-O Notation – N*N

N*N units of work are needed to print each N*N element.

What is the Big-O?

ray is an ArrayList!

```
int n = ray.size();
for ( int i=0; i<n; i++)
  for(int j=1; j<n;j*=2)
     out.println( ray.get(i) );
```

Big-O Notation – $N*Log_2(N)$

N * log2N units of work are needed to print each element log_2 times.

Comparing Runtimes of Arrays, Lists, Trees, and Collections

Arrays

traverse all spots	**O(N)**
search for an item	**O(N) or O(Log_2N)**
remove any item location unknown	**O(N)**
get any item location unknown	**O(1)**
add item at the end	**O(1)**
add item at the front	**O(N)**

If the array is sorted, a binary search would be the best choice and result in a log_2N runtime.

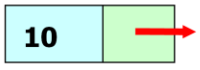
An array is a collection of like variables.

1	5	8	9	11

Single Linked Lists

traverse all nodes	O(N)
search for an item	O(N)
remove any item location unknown	O(N)
get any item location unknown	O(N)
add item at the end	O(N)
add item at the front	O(1)

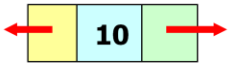
A single linked node has a reference to the next node only. A single linked node has no reference to the previous node.



Double Linked Lists

traverse all nodes	O(N)
search for an item	O(N)
remove any item location unknown	O(N)
get any item location unknown	O(N)
add item at the end	O(1)
add item at the front	O(1)

A double linked node has a reference to the next node and to the previous node.

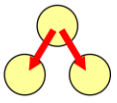


Binary Search Tree

traverse all nodes	O(N)
search for an item	$O(\log_2 N)$
remove any item location unknown	$O(\log_2 N)$
get any item location unknown	$O(\log_2 N)$
add item at the end	$O(\log_2 N)$
add item at the front	O(1)

A binary tree node has a reference to its left and right nodes. Nodes are ordered.

These notations assume the tree is balanced or near balanced.



Binary Tree Breakdown

If you insert items into a Binary Search Tree in order, the tree becomes a linked list.

1 - >2-> 3-> 4-> 5-> 6-> 7->

An unbalanced tree would have a worst case of O(N) for searching, adding at the end, removing any item, and getting any item.

Java Collections
ArrayList

traverse all spots	**O(N)**
search for an item	**O(N) or O(Log_2N)**
remove any item location unknown	**O(N)**
get any item location unknown	**O(1)**
add item at the end	**O(1)**
add item at the front	**O(N)**

If the array is sorted, a binary search would be the best choice and result in a log_2N runtime.

ArrayList is implemented with an array.

Java Collections

LinkedList

traverse all spots	O(N)
search for an item	O(N)
remove any item location unknown	O(N)
get any item location unknown	O(N)
add item at the end	O(1)
add item at the front	O(1)

LinkedList is implemented with a double linked list.

Java Collections

Set

	Tree Set	Hash Set
add	$O(Log_2N)$	O(1)
remove	$O(Log_2N)$	O(1)
contains	$O(Log_2N)$	O(1)

TreeSets are implemented with balanced binary trees (red/black trees).

HashSets are implemented with hash tables.

Java Collections
Map

	Tree Map	Hash Map
put	$O(Log_2N)$	O(1)
get	$O(Log_2N)$	O(1)
containsKey	$O(Log_2N)$	O(1)

TreeMaps are implemented with balanced binary trees (red/black trees).

HashMaps are implemented with hash tables.

Now it is time for another round of ....

What is the Big-O?

What is the Big-O?

```
int n = ray.size();
Set s = new HashSet();
for(int i=0; i<n; i++)
   s.add(ray.get(i));
```

Big O Notation – N

The work needed to add each element of ray to s would be N*1. Ray has N items and add() for HashSet has an O(1) bigO.

What is the Big-O?

```
int n = ray.size();
Set s = new TreeSet();
for(int i=0; i<n; i++)
   s.add(ray.get(i));
```

Big O Notation – $N*Log_2(N)$

The work needed to add each element of ray to s would be $N * log_2 N$. Ray has N items and add() for TreeSet has a log_2 bigO.

General Big O Chart for Searches

Name	Best Cast	Avg. Case	Worst
Linear/Sequential Search	O(1)	O(N)	O(N)
Binary Search	O(1)	$O(\log_2 N)$	$O(\log_2 N)$

All searches have a best case run time of O(1) if written properly. You have to look at the code to determine if the search has the ability to find the item and return immediately. If this case is present, the algorithm can have a best case of O(1).

General Big O Chart for N^2 Sorts

Name	Best Case	Avg. Case	Worst
Selection Sort	$O(N^2)$	$O(N^2)$	$O(N^2)$
Bubble Sort	$O(N^2)$	$O(N^2)$	$O(N^2)$
Insertion Sort	O(N) (@)	$O(N^2)$	$O(N^2)$

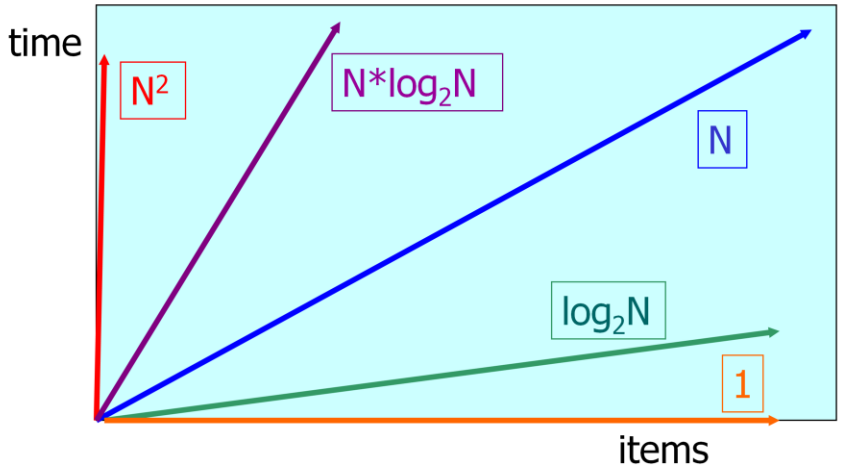
@ If the data is sorted, Insertion sort will only make one pass through the list.

General Big O Chart for NLogN Sorts

Name	Best Case	Avg. Case	Worst
Merge Sort	$O(N \log_2 N)$	$O(N \log_2 N)$	$O(N \log_2 N)$
QuickSort	$O(N \log_2 N)$	$O(N \log_2 N)$	$O(N^2)$ (@)
Heap Sort	$O(N \log_2 N)$	$O(N \log_2 N)$	$O(N \log_2 N)$

@ QuickSort can degenerate to N^2. It typically will degenerate on sorted data if using a left or right pivot. Using a median pivot will help tremendously, but QuickSort can still degenerate on certain sets of data. The split position determines how QuickSort behaves.

Graphing BigO



This is very general.