A+ Computer Science

BIG O NOTATION

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 :

$O(1)$
$O(2^N)$
$O(N \log_2 N)$
$O(\log_2 N)$
$O(\log_2 N)$
$O(N^2)$
$O(N)$
$O(N^3)$

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.

Analyzing Code

```
int fun = //some input
if(fun>30){
   out.println("whoot");
else if(fun<=30){
   out.println("fly");
}
```

How much work can take place when this code runs?

Analyzing Code

```
int run = //some input
for(int go=1; go<=run; go++)
{
  int fun = //some input
  if(fun>30){
     out.println("whoot");
  else if(fun<=30){
     out.println("fly");
  }
}
```

How much work can take place when this code runs?

Analyzing Code

```
int run = //some input

for(int go=1; go<=run; go++)
{
  int fun = //some input
  if(fun>30){
     out.println("whoot");
  else if(fun<=30){
     out.println("fly");
  }
}
```

Runs n times

Each time the loop runs, the if prints.

Total work – n(run) * 1

Big O Notation

The formal definition for BigO is :

BigO is bound(N) if runTime(N) <= c * bound(N)

The actual runtime of an algorithm is the
upper bound if the actual runtime is less than
c times an upper bound with c being a non-negative
constant and using any value of N greater than n_0.

Say what?

Big O Notation

runTime(N) – n/2 * 1

bound(N) – ????

runTime(N) <= c * bound(N)

n/2*1 <= ??

```
int run = //some input

for(int go=1; go<=run; go=go+2)
{
  int fun = //some input
  if(fun>30){
     out.println("whoot");
  else if(fun<=30){
     out.println("fly");
  }
}
```

Big O Notation

runTime(N) <= c * bound(N)

$n/2*1 <= c * \log_2 n$

$n_0 = 2$
$c = 3$

$50/2*1 <= 3*6$
$25 <= 18$

```
int run = 50

for(int go=1; go<=run; go=go+2)
{
  int fun = //some input
  if(fun>30){
    out.println("whoot");
  else if(fun<=30){
    out.println("fly");
  }
}
```

$O(\log_2 n)$ is too small.

Big O Notation

runTime(N) <= c * bound(N)

n/2*1 <= c * n

$n_0 = 2$
c = 3

50/2*1 <= 3*50
25 <= 150

```
int run = 50

for(int go=1; go<=run; go=go+2)
{
  int fun = //some input
  if(fun>30){
     out.println("whoot");
  else if(fun<=30){
     out.println("fly");
  }
}
```

O(n) is just right.

Big O Notation

runTime(N) <= c * bound(N)

$n/2*1 <= c * n^2$

$n_0 = 2$
$c = 3$

$50/2*1 <= 3*2500$
$25 <= 7500$

```
int run = 50

for(int go=1; go<=run; go=go+2)
{
  int fun = //some input
  if(fun>30){
    out.println("whoot");
  else if(fun<=30){
    out.println("fly");
  }
}
```

$O(n^2)$ is too big.

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) );
```

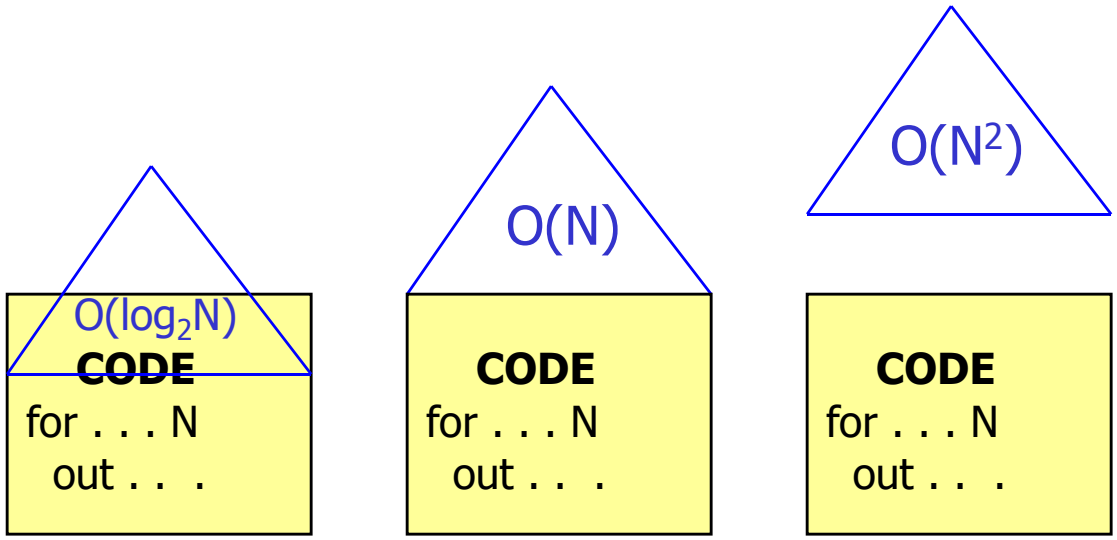
ray is an ArrayList!

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?

ray is an ArrayList!

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

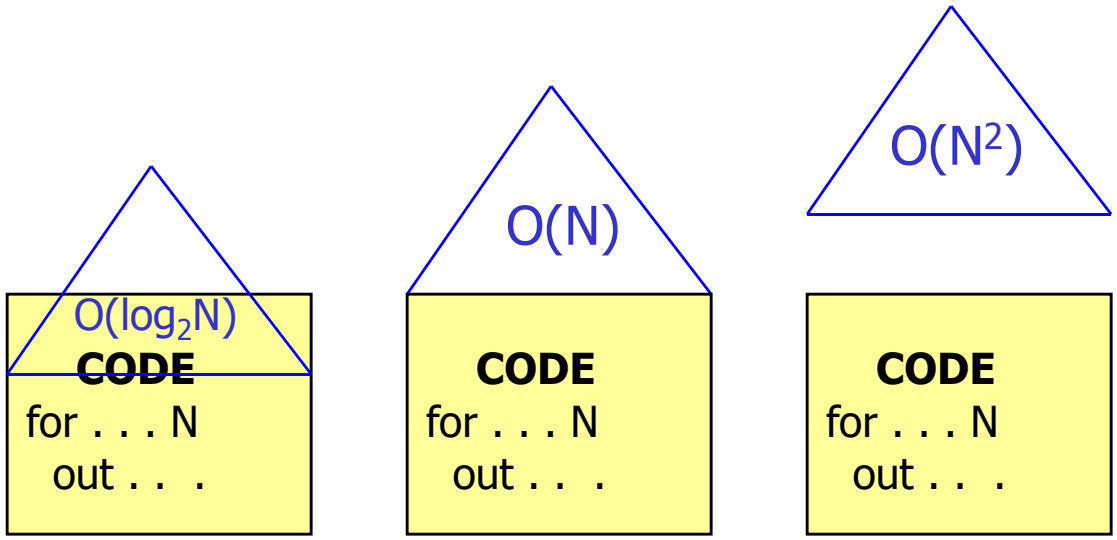
This example is not as easy as the last.

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) );
```



Which roof/bound fits best?

What is the Big O?

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

Array Runtimes

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.

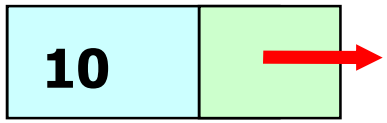
An array is a collection of like variables.

1	5	8	9	11

Linked List Runtimes

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)**

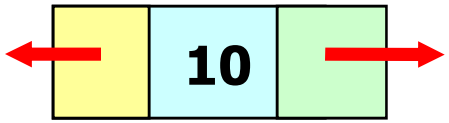
A single linked node has a reference to the next node only. A single linked node has no reference to the previous node.



Linked List Runtimes

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.

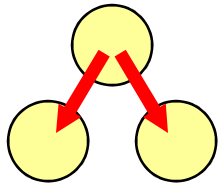


Binary Tree Runtimes

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 Runtimes

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.

ArrayList Runtimes

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.

LinkedList Runtimes

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.

Set Runtimes

	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.

Map Runtimes

	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.

What is the Big O?

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).

What is the Big O?

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) (@)	$O(N^2)$	$O(N^2)$

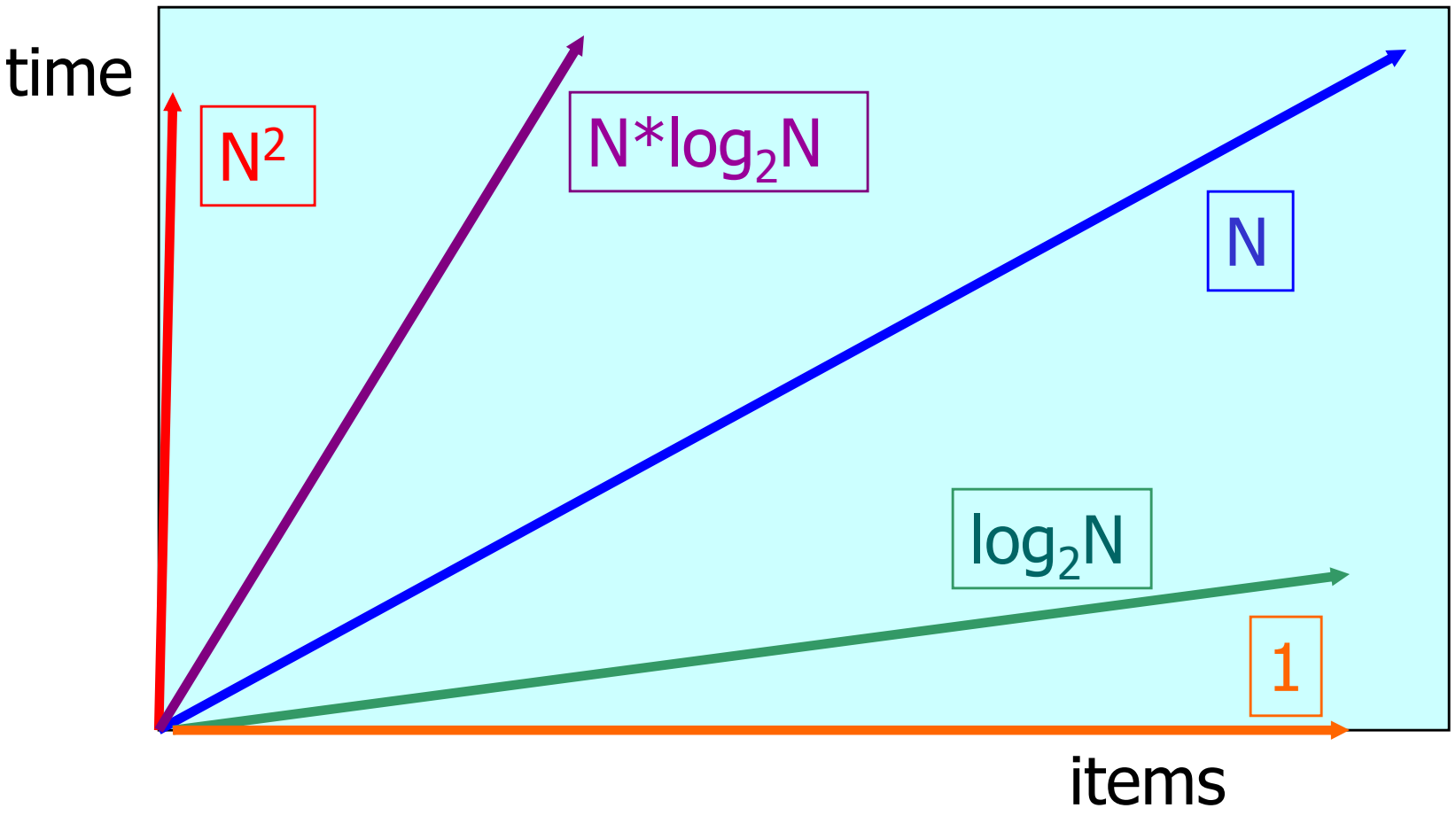
@ If the data is sorted, Insertion sort will only make one pass through the list.

What is the Big O?

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.

What is the Big O?



This is very general.

A+ Computer Science

BIG O NOTATION