
BIG O NOTATION

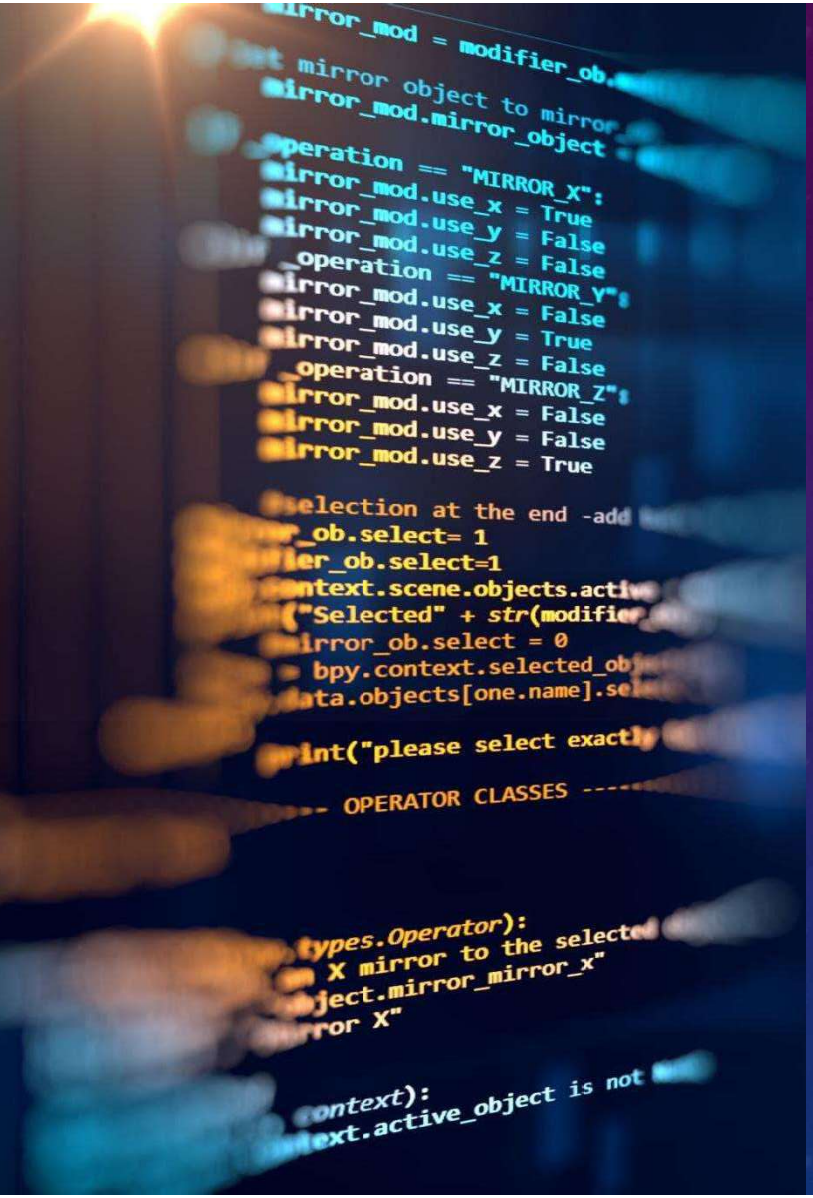
WHAT IS BIG O NOTATION?

Big O notation describes the **complexity** of our code using algebraic terms.

The “O” refers to “Order” as in “order of the function” from mathematics

The “order of the function” is the growth rate of the function

“Big” because we are interested in the upper bound – the “biggest” number of operations



ALGORITHM EFFICIENCY

- We want to think about how to design efficient algorithms
- We need a universal standard for measuring algorithm efficiency
- Consider two pieces of code we want to compare
  - One is run a very fast well-maintained computer
  - Another is run on an old, slow computer
- How useful is this comparison?

RUNNING TIMES FOR A DICE SIMULATOR

5 Dice rolled 1 million times

Algorithm 1	Supercomputer	40ms
Algorithm 2	Laptop	70ms

- Which Dice algorithm is better?

THE METRE

- Historically, (1889-1960) the metre was defined by the French Academy of Sciences
- It was the length between two marks on a platinum-iridium bar
- 1/10 millionth of the distance from the equator to the North Pole through Paris

ALGORITHM EFFICIENCY

- In the same way, we could try running all algorithms on the same computer
- But is this a good universal standard?
- We would need to have a single benchmark machine on which all of the world's programs were tested
- This machine would quickly become antiquated and need to be updated, invalidating the previous measurements

RUNNING TIMES

		1 million rolls	2 million rolls	3 million rolls
Algorithm 1	Supercomputer	40ms	160ms	360ms
Algorithm 2	Laptop	70ms	140ms	210ms

Which Dice algorithm is better?

STANDARD MEASURE

- The **relationship** between the increase in the size of a problem and the increase in the running time is platform independent
- No matter what platform you run it on, the same relationship will emerge
- Because the platform isn't important, this is useful for defining algorithm efficiency
- Knowing the relationship is very useful for **predicting** how long an algorithm will take to run on a particular problem

BIG O NOTATION

We use Big O Notation to describe this ratio

We are not concerned with the actual time it takes to run the algorithm

100 ms on a laptop	10 ms on a supercomputer

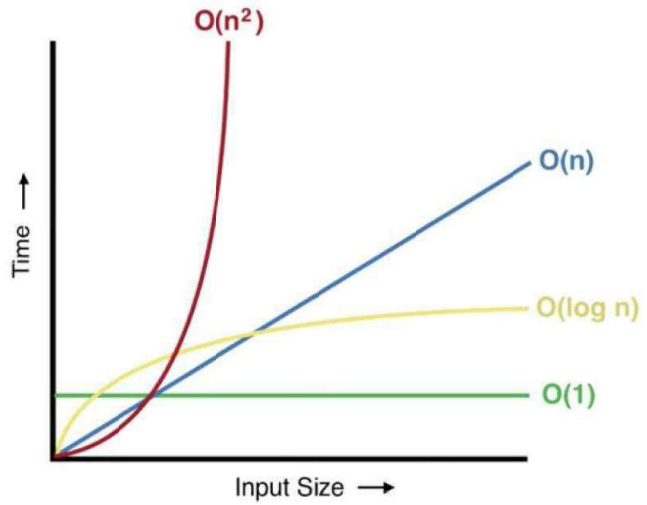
BIG O NOTATION

We want a way to describe the rate with which the running time of the algorithm increases compared to the rate at which the size of the problem (n) increases

Running time relative to problem size

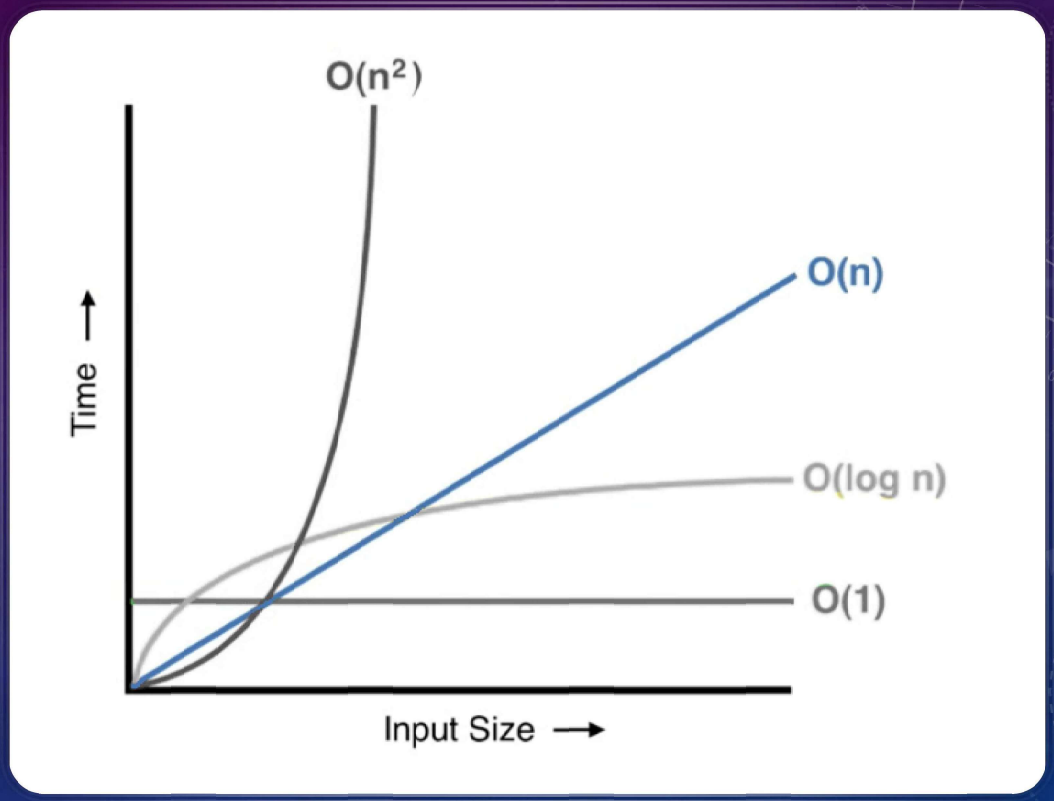
Big O is always concerned with **worst** case time requirement

LET'S LOOK AT SOME EXAMPLES...



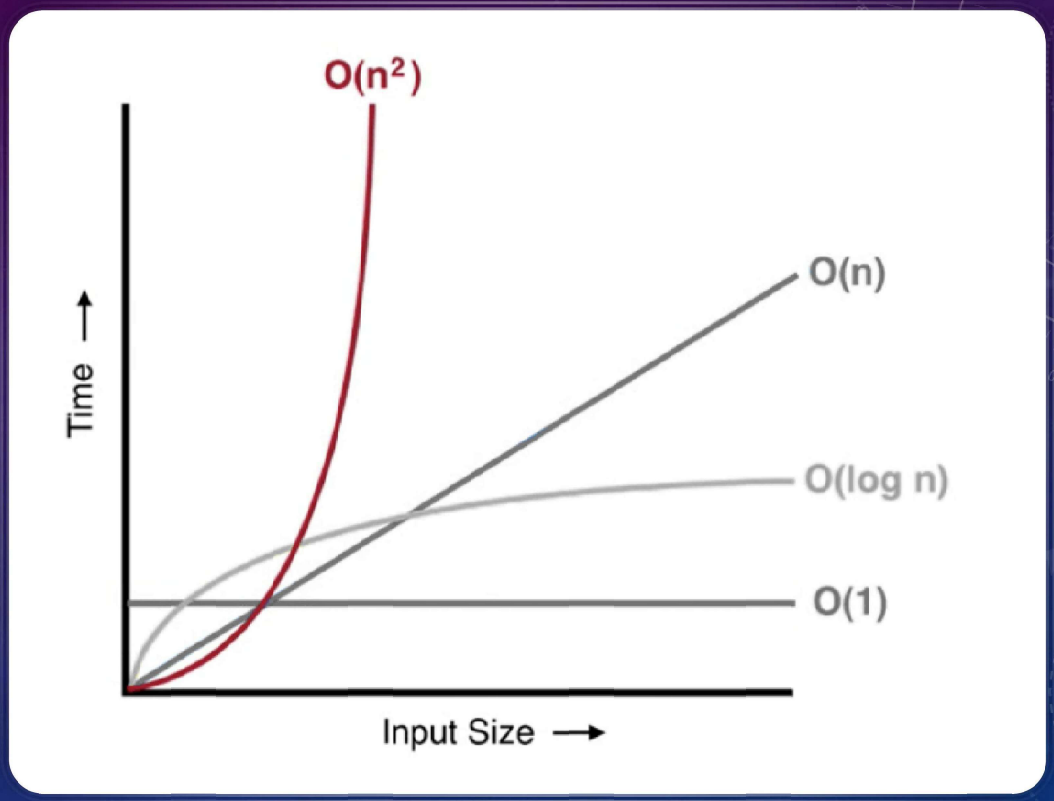
O(N) - LINEAR TIME

- **<u>Running time increases proportional to the rate at which the problem size increases</u>**
- Example: Print all the values in a list.
- Printing 10 items will be 10 operations
- Printing 30 items will be 30 operations
- Printing 9999999 will be 9999999 operations



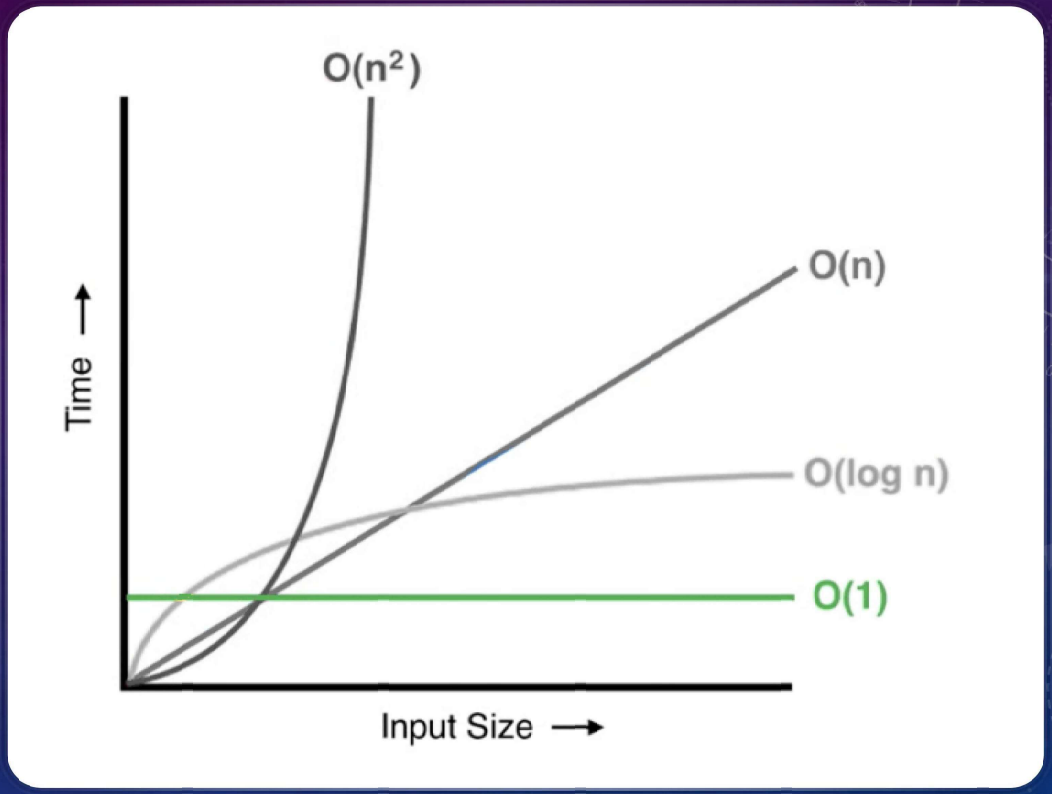
$O(N^2)$ - QUADRATIC TIME:

- **<u>Running time increases proportional to the square of the input size</u>**
- Example: Bubble sort (more on this later)
- Input of size 2 is 4 operations.
- Input of size 4 is 16 operations.
- Input of size 8 is 64 operations.



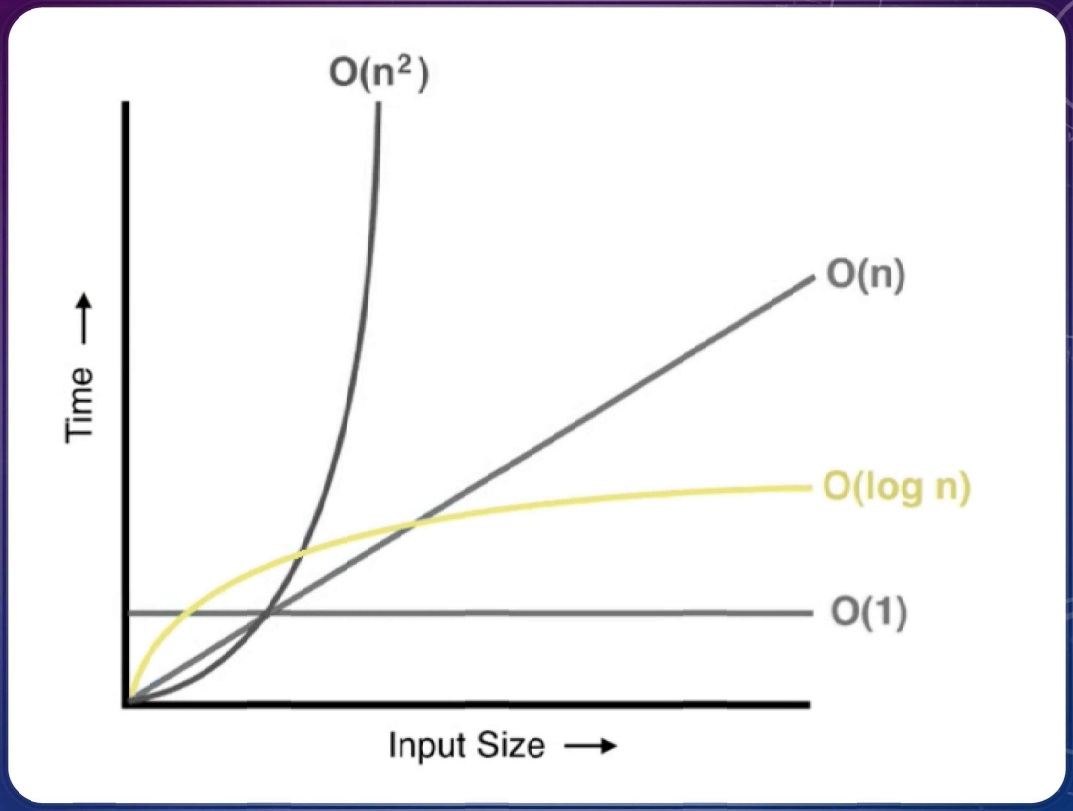
O(1) - CONSTANT TIME

- **<u>Running time does not depend on the size of the input</u>**
- Example: print the first item in an array
- How many operations in an array of size 10?
- How many operations in an array of size 8000?



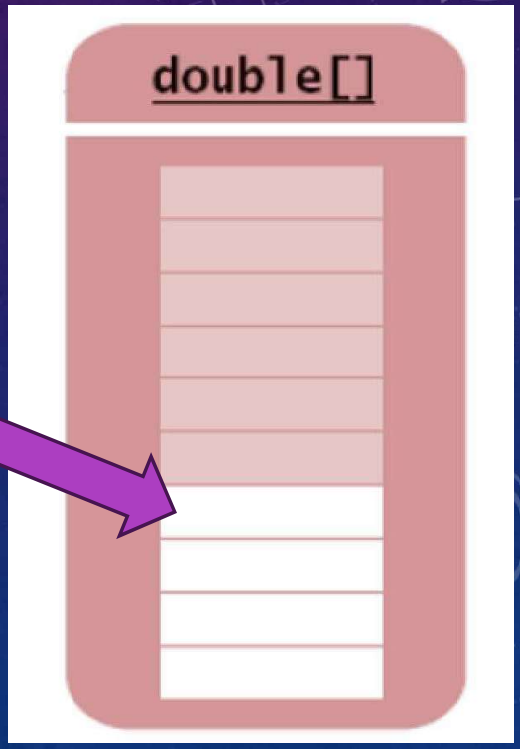
O(LOG N) - LOGARITHMIC TIME

- **Running time increases logarithmically to the rate at which the size increases**
- Example: binary search is a logarithmic time algorithm



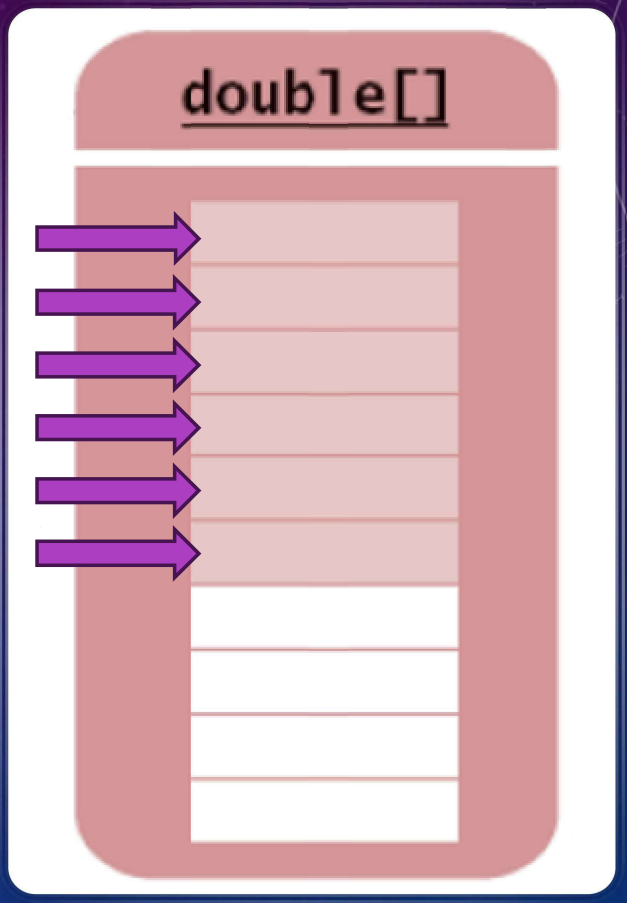
INSERTION IN AN UNORDERED ARRAY – O(1)

- The running time of insertion into an unordered array doesn't depend on the size of the array – we just stick the element on at the end
- So, we want to think of this algebraically:
- We'll say that time (T) = some constant time (K) which won't change → T = K
- K can depend on factors such as the speed of the computer, the amount of RAM etc.
- We don't care what K actually is – Big O Notation is only concerned with describing the relationship between the running time and the size of the problem
- We just say the algorithm is O(1) – running time is unaffected by *n*



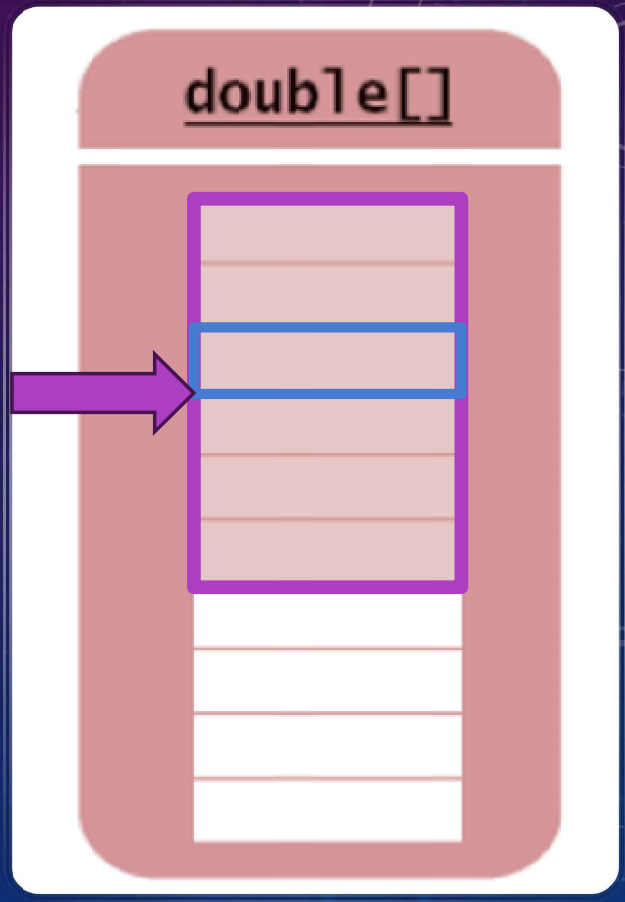
LINEAR SEARCH

- What is the running time of linear search?
- We have to search through all the elements in an array
- On average, we'll have to check half of them
- So T = K * (n / 2)
- Because K is a constant, K / 2 will still be a constant (value doesn't depend on n)
- So T = K * n
- This algorithm is O(n)



BINARY SEARCH

- We have already shown that for a binary search iterations = $\log_2(\text{size})$
- Therefore $T = K * \log_2(n)$
- Consider the equation:
  - $\log_{10}X = 3.32 * \log_2X$
- Incorporating the 3.32 into the K, we get $T = 3.32K * \log_{10}(n)$
- 3.32K is just a constant which is irrelevant to Big O Notation
- This algorithm is O(log n)



REMINDER - A LOG RELATIONSHIP

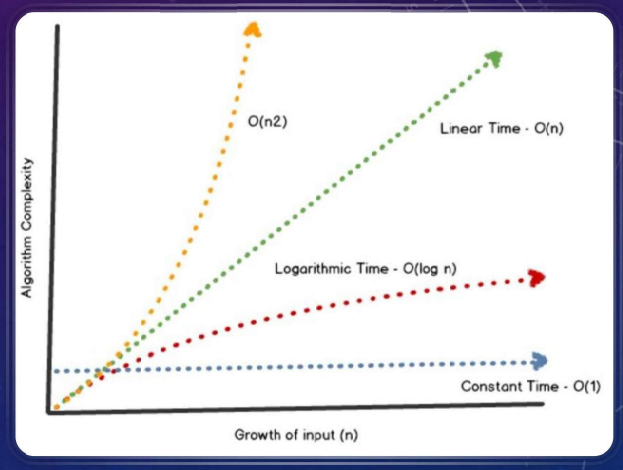
- Each step halves the size, so the number of iterations needed to search through an array using a binary search is the number of times the size of the array can be halved

  $size = 2^{iterations}$

- The opposite of raising something to a power is to take its log

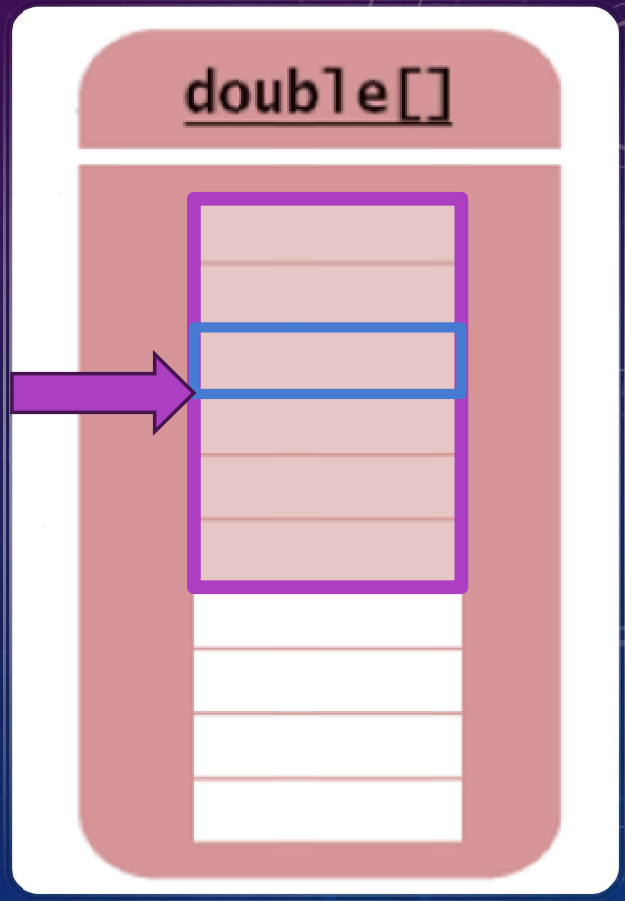
$iterations = \log_2(size)$

- Number of steps required increases very slowly compared to increases in size – logarithmically as opposed to linearly
- We express this log type relationship between array size and number of steps required by saying that the complexity of binary search is O(log n)



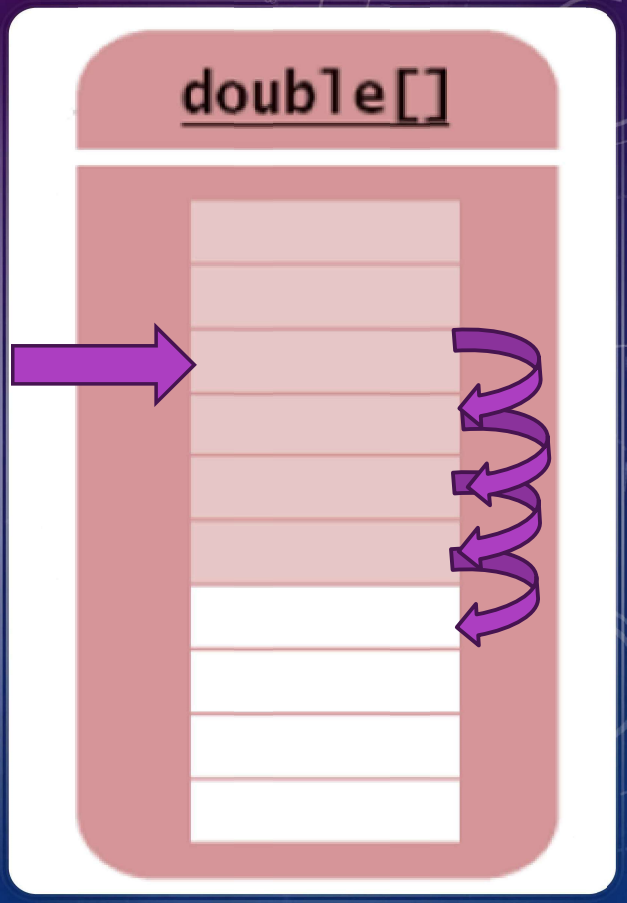
BINARY SEARCH

- We have already shown that for a binary search iterations = $\log_2(\text{size})$
- Therefore $T = K * \log_2(n)$
- Consider the equation:
  - $\log_{10}X = 3.32 * \log_2X$
- Incorporating the 3.32 into the K, we get $T = 3.32K * \log_{10}(n)$
- 3.32K is just a constant which is irrelevant to Big O Notation
- This algorithm is O(log n)



OPERATIONS IN AN ORDERED ARRAY

- Ordered arrays are handy because we can use binary search on them, and this is O(log*n*)
- However, if we want to insert or delete we have to make space / remove a space
- On average, we will have to move half of the items up or down → **K * (n/2)**
- Therefore, these operations are O(n)



RUNNING TIMES IN BIG O NOTATION

Algorithm	Running Time
Linear Search	O(n)
Binary Search	O(log n)
Insertion in unordered array	O(1)
Insertion in ordered array	O(n)
Deletion in unordered array	O(n)
Deletion in ordered array	O(n)

EXPRESSING ITERATIONS IN TERMS OF *N*

- Usually, we can look at a piece of code and derive a function f(*n*) which describes the number of loop **steps** in it
- How many loop iterations in this code?
- In other words, how many time will `counter++` be run?

```
for (int i = 10; i < n; i++){



}
```

EXPRESSING ITERATIONS IN TERMS OF *N*

- Usually, we can look at a piece of code and derive a function f(*n*) which describes the number of loop **steps** in it
- How many loop iterations in this code?
- In other words, how many time will `counter++` be run?

```
for (int i = 10; i < n; i++){
        for (int j = 10; j > 0; j--){
            counter++;
        }
}
```

- There are (n - 10) * 10 iterations = 10n - 100

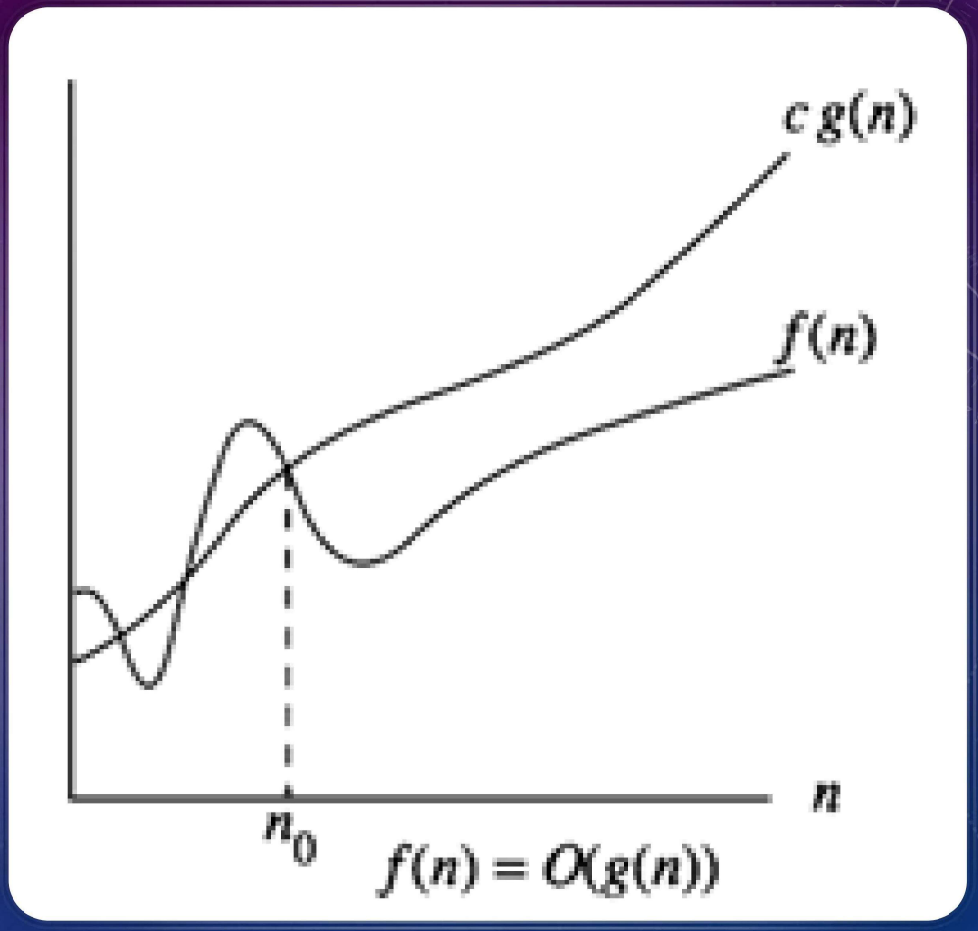
FORMALITIES

- Formal mathematical definition of Big O
- A function f(n) = O(g(n)) if
  - a positive real number *c* and positive integer n_0 exist such that

$$f(n) \leq c \bullet g(n) \text{ for all } n \geq n_0$$

GRAPH

- *c*.g(n) is the upper bound on f(n) when n is sufficiently large



INTERPRETATION

- We want to describe how the size of a function f(n) (which describes the running time of a program) increases as n gets really huge
- The biggest power of n will always dominate
- Accordingly, we pick this as the Big O complexity $g(n)$
- We don't care about constants

INTERPRETATION

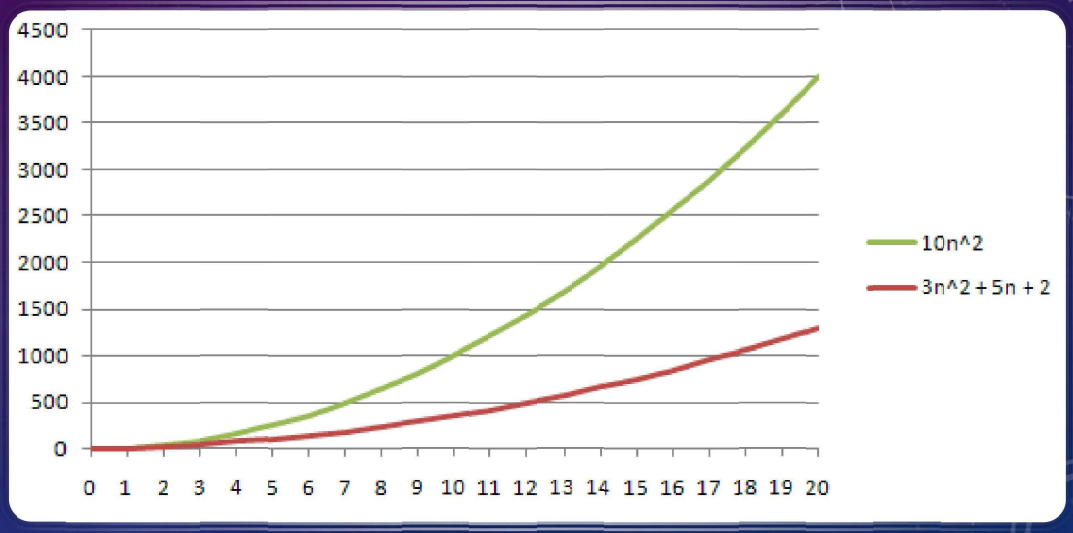
- To justify that this pick is a good description of f(n), we show that f(n) is always bounded by the Big O complexity g(n) (multiplied by some constant, which doesn't matter as we don't care about constants!) as long as n is bigger than some value n_0
- In other words, to show g(n) provides a good description of f(n) we show that $f(n) \leq c \cdot g(n)$ for all $n \geq n_0$

INTERPRETATION

- For example $O(n^2)$ is a good description of $3n^2 + 5n + 2$ since n^2 multiplied by the arbitrary constant **10** will always be bigger than $3n^2 + 5n + 2$ for every value of n greater than 1
- $O(n^2)$ manages to capture the behaviour of this function as *n* becomes bigger (with only a constant amount of inaccuracy)
- We don't care that $3n^2 + 5n + 2$ could be up to **10** times bigger than $O(n^2)$
- **10** is only a constant and in the long run as *n* gets huge, constants will become insignificant

EXAMPLE

- The function $10n^2$ will always exceed $3n^2 + 5n + 2$, so long as n is 2 or greater
- Therefore $3n^2 + 5n + 2$ is $O(n^2)$ because...
  - $10n^2 = 3n^2 + 5n^2 + 2n^2$ which is > than $3n^2 + 5n + 2$ when n >= 2...
  - $3n^2 = 3n^2$
  - $5n^2 > 5n$
  - $2n^2 > 2$



EXPLAIN?

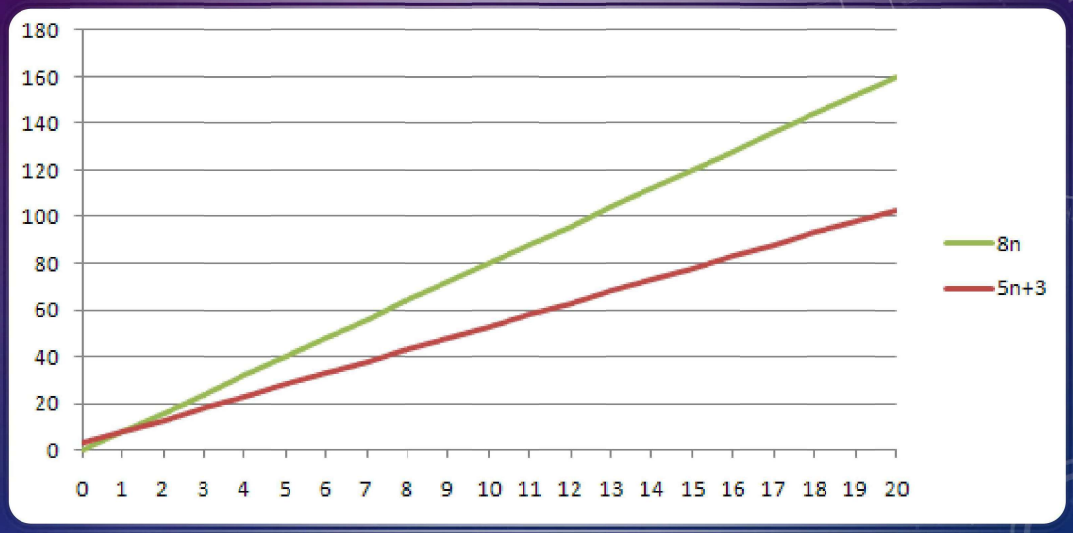
- We want the Big O function which is the closest description of the performance of our function (i.e. computer program)
- The function must be bounded by the Big O function beyond a certain problem size $\mathbf{n_0}$
- The Big O function can be multiplied by any constant in order to meet this requirement
- For example, if I describe my function as being O(n), what I mean is that my function always has a running of less than **K * n** when n is bigger than n_0
  - K can be a million, a billion, a trillion, it doesn't matter
  - n_0 can be any value too, but it is usually more sensible to keep it low
  - Remember - even though it is huge, 2^{100} is a constant because it has no **n** term

EXAMPLE

- Show that f(n) = 5n + 3 = O(n)
  - Find a g(n), c and n_0 such that f(n) ≤ c.g(n) for all n ≥ n_0
  - How about g(n) = n, c = 8, n_0 = 1?
  - f(n) <= 8n for every value of n greater than 1
  - 5n + 3 is always less than 5n + 3n when n is at least 1
  - Therefore, we can say f(n) is O(n)
- Why don't we let g(n) = n^2 ?
- Although the conclusion is correct since f(n) will always be less than O(n^2) as well, this is not the closest description of the algorithm

EXAMPLE

- 8*n* will always bound 5*n* + 3 when n is bigger than 1
- Therefore, we can say that a program with 5*n* + 3 steps is O(n)
- Of course, it would also be bounded by 6*n* but so long as we show it for any constant then that's sufficient



USAGE

- Always use the most concise formula for the O-notation.
- We write
  - $3n^2+2n+5 = O(n^2)$
- The followings are all technically correct but not what we're looking for:
  - $3n^2+2n+5 = O(3n^2+2n+5)$
  - $3n^2+2n+5 = O(n^2+n)$
  - $3n^2+2n+5 = O(3n^2)$

HOW DO WE KNOW THE ORDER OF THE FUNCTION?

1. In order to figure out what the order of a function is, look at the highest order of n
   - If the highest order is an $\mathbf{n^2}$ term, then the formula is $O(n^2)$
   - Always put g(n) equal to this power
     - $g(n) = n^2$
2. Now choose **c** so that it equals the sum of all the variables in the function
   - If $f(n) = 3n^2+2n+5$, then choose **c** to be 10 (3 + 2 + 5)
   - This makes it easy to show that $3n^2+2n+5 < 3n^2+2n^2+5n^2$
3. Finally, figure out what value n_0 needs to have in order to make the above statement $f(n) \leq c.g(n)$ true

FORMALITIES

- The following identities hold for Big O notation:

1. **$O(k * f(n)) = O(f(n))$**
   - If an algorithms complexity is multiplied, it still has the same Big O Notation
2. **$O(f(n)) + O(g(n)) = O(f(n) + g(n))$**
   - If we run one algorithm after the other, the complexity is added
   - However, if algorithm 1 is $O(n^2)$ and algorithm 2 is $O(n)$ then $O(n^2 + n)$ can be more concisely described as $O(n^2)$
3. **$O(f(n)) * O(g(n)) = O(f(n) * g(n))$**
   - If algorithm 1 is $O(n^2)$ and algorithm 2 is $O(n)$ and one algorithm is run **inside** the other as a loop, then the Big O Notation is $O(n^3)$

BIG-O EXAMPLES

- $7n - 2$

  $7n-2$ is $O(n)$

  need $c > 0$ and $n_0 \geq 1$ such that $7n-2 \leq c \bullet n$ for $n \geq n_0$

  this is true for $c = 9$ and $n_0 = 1$

- $3n^3 + 20n^2 + 5$

  $3n^3 + 20n^2 + 5$ is $O(n^3)$

  need $c > 0$ and $n_0 \geq 1$ such that $3n^3 + 20n^2 + 5 \leq c \bullet n^3$ for $n \geq n_0$

  this is true for $c = 28$ and $n_0 = 1$

- $3 \log n + 5$

  $3 \log n + 5$ is $O(\log n)$

  need $c > 0$ and $n_0 \geq 1$ such that $3 \log n + 5 \leq c \bullet \log n$ for $n \geq n_0$

  this is true for $c = 8$ and $n_0 = 10$ ($\log 10 = 1$ so n_0 has to be 10 before $\log n$ exceeds 1)

KEEPING IT SIMPLE

- $f(n) = 10\ n + 25\ n^2$
- $f(n) = 20\ n \log n + 5\ n$
- $f(n) = 12\ n \log n + 0.05\ n^2$
- $f(n) = n^{1/2} + 3\ n \log n$

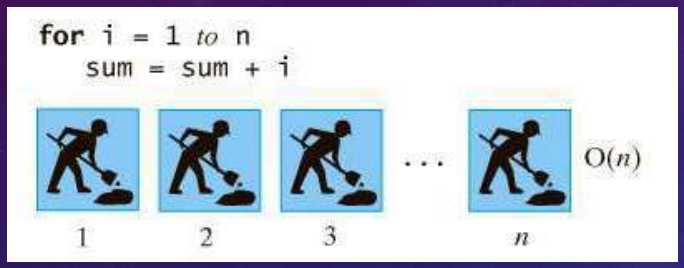
- $O(n^2)$
- $O(n \log n)$
- $O(n^2)$
- $O(n \log n)$

GETTING BIG O OF A PROGRAM

- When trying to determine the Big O Notation of a computer program, look at the loop structure
- Statements that are run the same number of times regardless of the size of the problem are just **constants**
- All you're interested in is how increasing the size of **n** increases the number of iterations of the loops
- Increasing the size of **n** will only have an effect i there a loop structure which depends on **n**
  - A single loop running n times indicates O(n)
  - A nested loop each running n times indicates $O(n^2)$

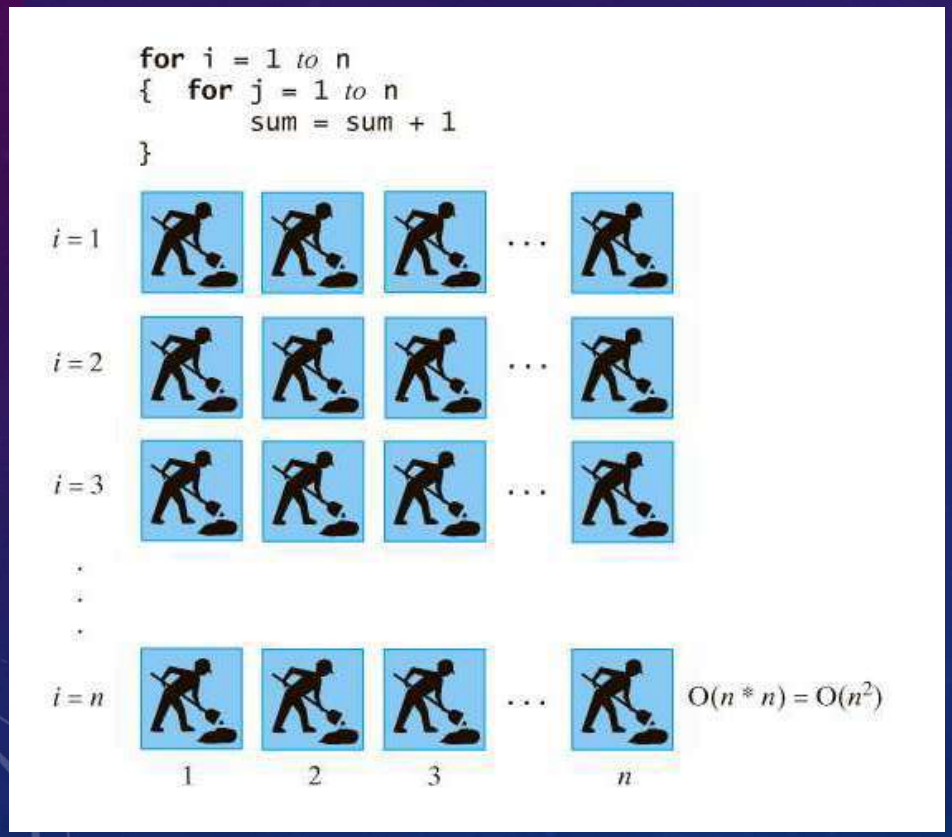
PICTURING OPERATIONS

- Consider this algorithm:



- The work done by the body of the loop (i.e. `sum = sum + i`) requires a constant amount of time O(1)
- This body is executed n times
- Therefore, the algorithm is O(n)

PICTURING OPERATIONS



- n steps of work are repeated n times
- An $O(n^2)$ algorithm

SHAKING HANDS AT A PARTY

- If there are n people at the party, we will need to shake (n-1) hands
- The next person will have to shake (n-2) hands (they don't have to shake your hand again)
- The last person has to shake 0 hands because everybody has already shaken their hand
- Total number of handshakes:
  - (n-1) + (n-2) + (n-3)……0

- There will be n terms in the above and the average term is
  - ((n-1) + 0) / 2 = (n-1) / 2

- Total number = n*(n-1) / 2 = $n^2/2 – n/2$
- What is the big O notation of this algorithm?

COMPUTE AVERAGE OF ARRAY

```
double average(int[] array) {
    double sum = 0;
    int n = array.length;
    for (int i=0; i<n; i++) {
        sum += array[i];
    }
    return sum / n;
}
```

What is the big O notation of this code?

One loop running n times so this is…

NESTED LOOPS

```
double sum = 0;
for(int i=0; i<n; i++) {
    for(int j=0; j<n; j++) {
        sum += 5;
    }
}
```

What is the big O notation of this code?

Nested for loops each running n times so this is….

LOOP RUNNING CONSTANT NUMBER OF TIMES

```
for (int pass = 1; pass <= n; pass++)
{
   for (int index = 0; index < n; index++)
   {
        for (int count = 1; count < 10; count++)
        {
. . .
        } // end for
   } // end for
} // end for
```

What is the big O notation of this code?

Two loops run to n…

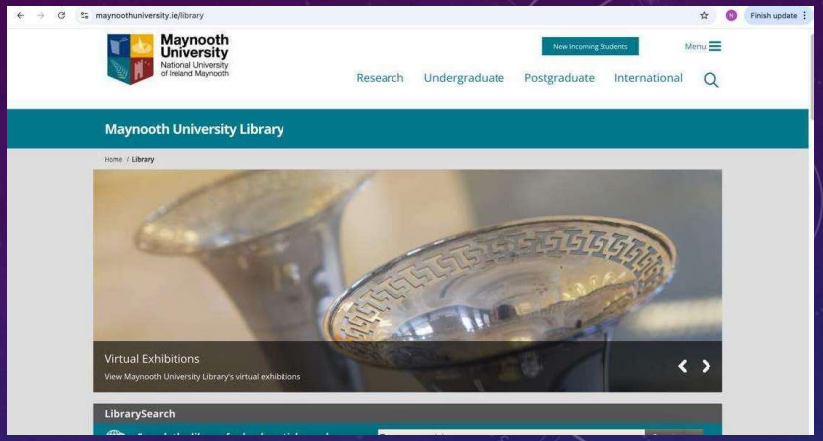
Third loop runs a consistent number of times…

HOW ABOUT THIS LOOP?

```
for(int i=0; i<10; i++) {
  for(int j=0; j<20; j++) {
   counter++;
   }
}
```

- We want to analyse the complexity of this algorithm
- Notice that the loops do not depend on the size of n
- No matter what size n is, the loops will run the same number of times
- Therefore, the running time will always be the same
- The order of the algorithm is?

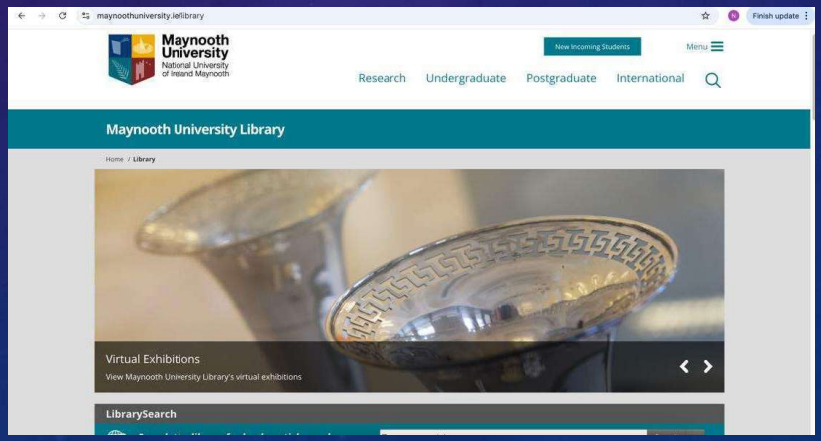
EXAM QUESTIONS



- Go to the Maynooth University library website:
  - https://www.maynoothuniversity.ie/library
- Scroll down until you see “Quicklinks” on the left of the screen
- Click on exam papers

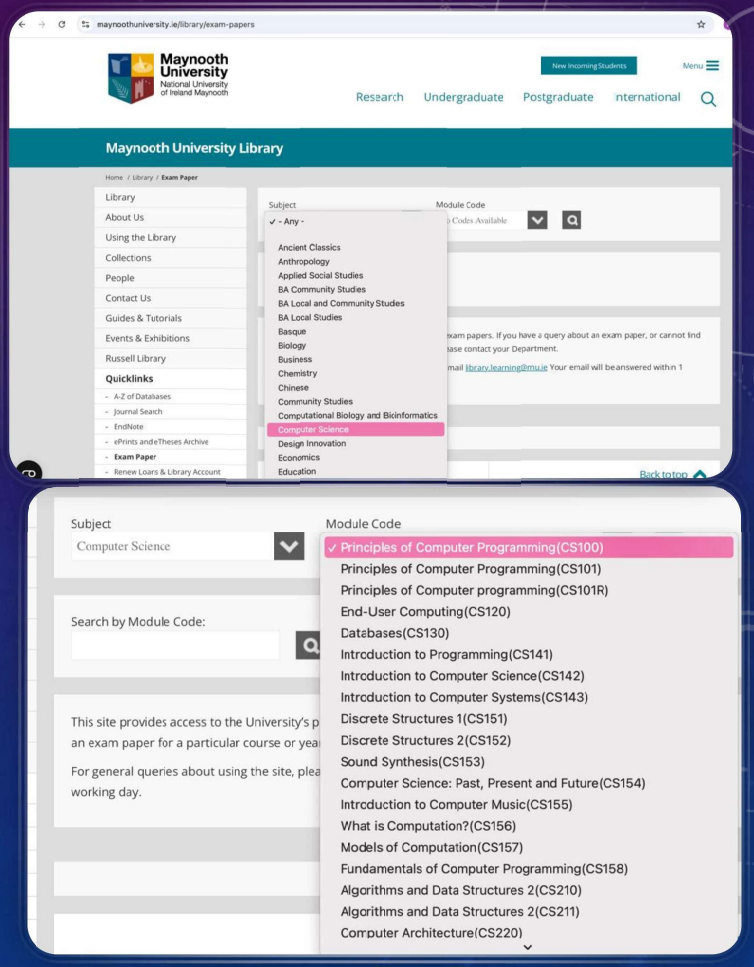
EXAM PAPERS

- Go to the Maynooth University library website:
  - https://www.maynoothuniversity.ie/library
- Scroll down until you see “Quicklinks” on the left of the screen
- Click on exam papers



EXAM PAPERS

- On the exam papers page, click the subject dropdown and choose computer science
- Then choose the module you're looking for form the Module Code drop down menu and click search



EXAM QUESTION – JANUARY 2008

A function involves the following number of steps where *n* is the size of the problem:

$$f(n) = \log(n) + n/2 + 5$$

State the Big-O complexity of the function and prove that this is the case using the mathematical definition.

EXAM QUESTION – JANUARY 2008

- g(n) = n since n is the biggest term
- Let c = 7 since there are 7 units in the function
- We must show c.g(n) >= f(n) above some threshold n_0

- 7n = n + n + 5n >= logn + n/2 + 5

  as long as n > =1
- QED

TIMING PROGRAMS

- Out of interest - we can check how long our program has been running
- There is a System method that allows us to store the current value of the system clock
- By comparing two different system clock values we can figure out how long the program has been running

```
long start = System.currentTimeMillis();
long elapsed = System.currentTimeMillis() - start;
```