Data Structures & Algorithms 1

Topic 5 – Big O Notation

Algorithm Efficiency

- Throughout the course we will seek to design efficient algorithms
- But how can we come up with a universal standard for algorithm efficiency?
- Imagine I compare my dice program with yours
- I run my program on my fast office computer
- You run your program on a slower lab machine

Running Times

5 Dice rolled 1 million times

My computer	40ms
Your computer	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
- Every measurement was based on this bar

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
  - 386 25Mhz with 2MB RAM
- 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
My computer	40ms	160ms	360ms
Your computer	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 (apart from a constant)
- No matter what platform you run it on, the same relationship will emerge
- We therefore use this relationship to define algorithm efficiency
- Knowing the relationship is very useful for predicting how long an algorithm will take to run on a particular problem

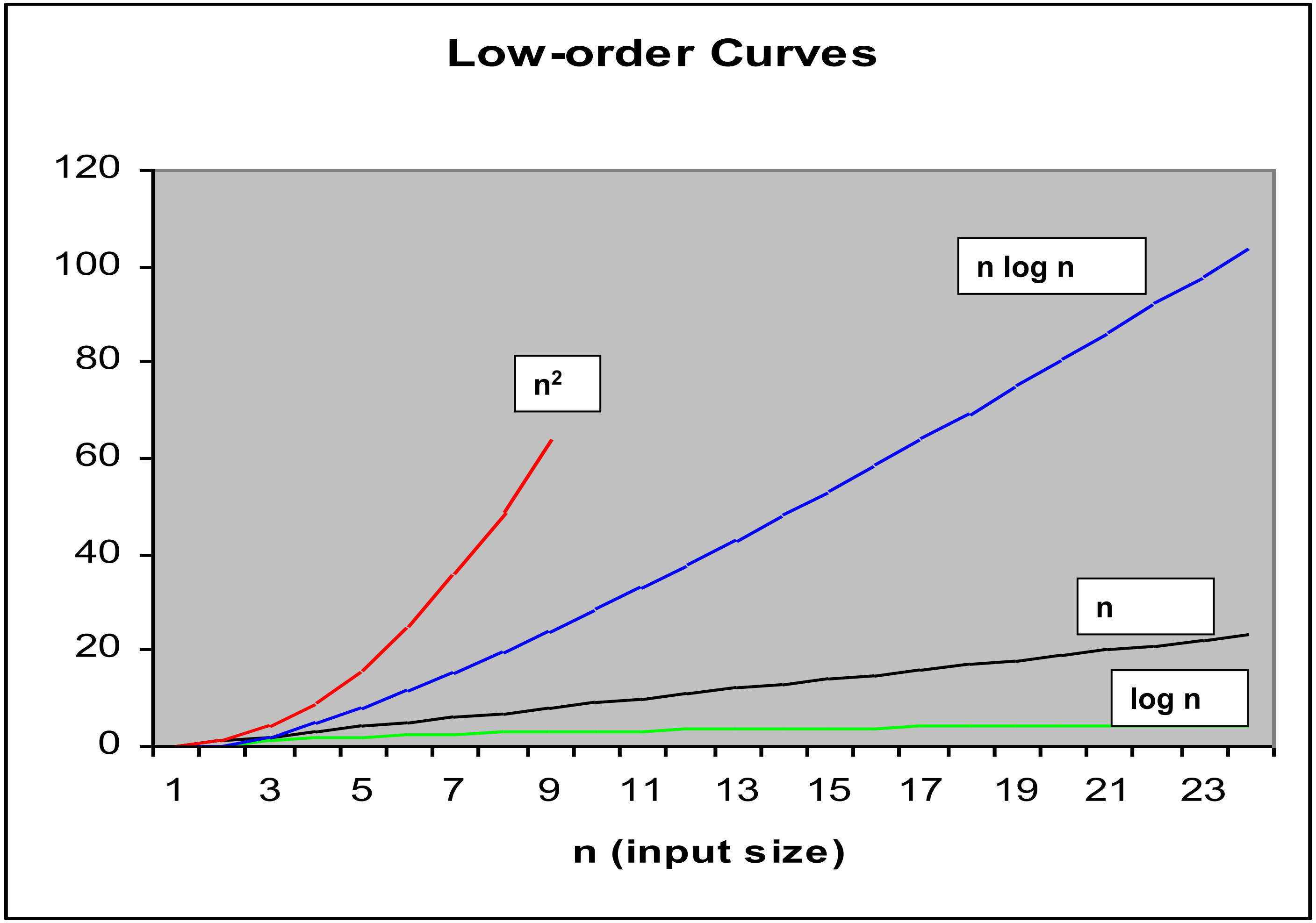
Big O Notation

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

Examples

- O(n) – The rate at which the running time increases is proportional to the rate at which the size increases
- $O(n^2)$ – Running time increases proportional to the square of the size of the problem
- O(1) – Running time is not related to the size of the problem
- O(log n) – Time increases slowly at log the rate of the size

Big O graph



Low-order Curves
120
100
80
60
40
20
0
1
3
5
7
9
11
13
15
17
19
21
23
n (input size)
n log n
n^2
n
log n

Insertion in an Unordered Array

- For insertion into an unordered array running time doesn't depend on the size of the array – we just stick the element on at the end
- We can say that running 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

- 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 iterations = $\log_2(\text{size})$
- Therefore $T = K * \log_2(n)$
- As it happens $\log_{10}X = 3.32 * \log_2 X$
- 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)

Question

- Which of the following statements about Big O Notation is false?

A)	Big O Notation provides a standard mechanism for comparing algorithms
B)	Big O Notation is useful for identifying parts of a program that are causing bottlenecks because they are inefficient
C)	The same algorithm will always have the same Big O Notation no matter what machine it is run on
D)	Algorithms with the same Big O Notation can take different amounts of time to run on the same computer for a given problem size
E)	Knowing the Big O Notation of an algorithm allows you to predict exactly how long the algorithm will take to run for a given problem size

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

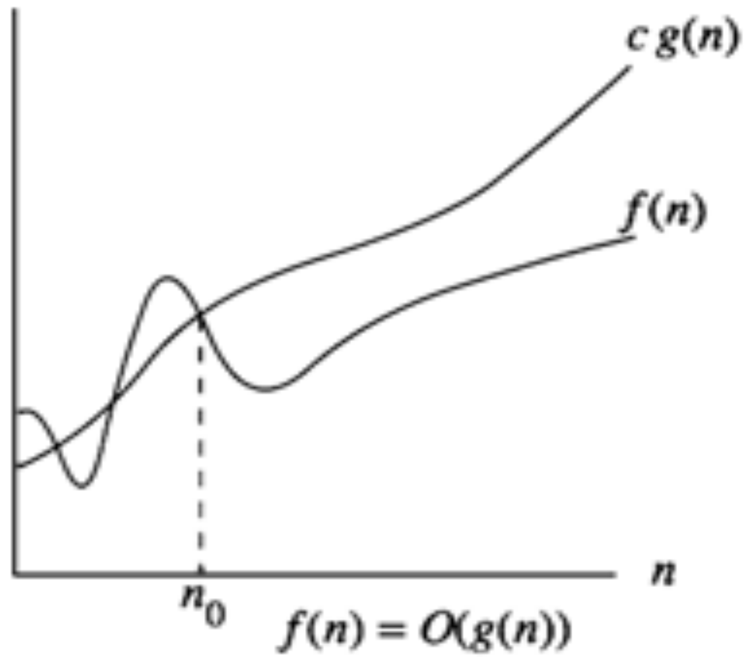
Formalities

- 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

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

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
- 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)

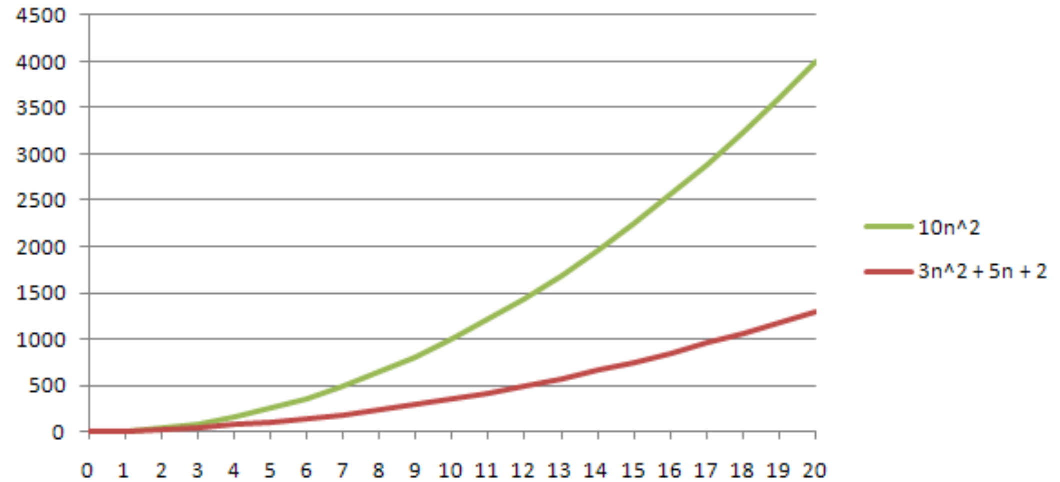
- We don't care that $3n^2 + 5n + 2$ could be up to 10 times bigger than $O(n^2)$

- 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 as 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?

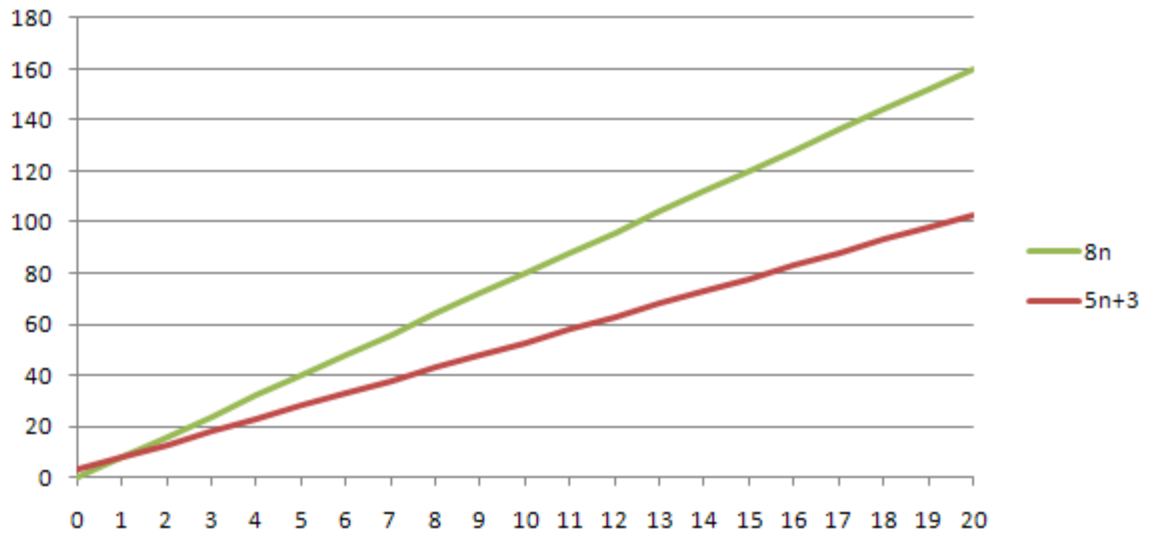
- I'm looking for the Big O function which is the closest description of the performance of my function (i.e. computer program)
- My function must be bounded by the Big O function beyond a certain problem size n_0
- The Big O function can be multiplied by any constant in order to meet this requirement
- 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
  - Even though it is huge, 2^{100} is actually 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 $f(n) \leq c.g(n)$ for all $n \geq n_0$
  - 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

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



Usage

- Always use the most parsimonious formula for the O-notation.
- We write
  - $3n^2+2n+5 = O(n^2)$
- The followings are all correct but we want the most concise
  - $3n^2+2n+5 = O(3n^2+2n+5)$
  - $3n^2+2n+5 = O(n^2+n)$
  - $3n^2+2n+5 = O(3n^2)$

Tip

- In order to figure out what the order of a function is, just look at the highest order of n
- If there's an n^2 term, then the formula is $O(n^2)$
- Always put g(n) equal to this power
  - $g(n) = n^2$
- 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 $3n^2+2n+5 < 3n^2+2n^2+5n^2$
- Finally, figure out what value n_0 needs to have in order to make the above statement $f(n) \leq c.g(n)$ true

Example of O-notation

- Show that $3n^2+2n+5 = O(n^2)$
  - $g(n) = n^2$, $c = 10$, $n_0 = 1$
  - Pick $c = 10$ because it's easy to show $10n^2 \geq 3n^2 + 2n + 5$

$10n^2 = 3n^2 + 2n^2 + 5n^2$

$3n^2 + 2n^2 + 5n^2 \geq 3n^2 + 2n + 5$ for all $n \geq 1$

Formalities

- The following identities hold for Big O notation:
- $O(k * f(n)) = O(f(n))$
  - If an algorithm is doubled in complexity, it still has the same Big O Notation
- $O(f(n)) + O(g(n)) = O(f(n) + g(n))$
  - 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 parsimoniously described as $O(n^2)$
- $O(f(n)) * O(g(n)) = O(f(n) * g(n))$
  - 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 $n_0 = 10$ (log10 = 1 so n_0 has to be 10 before logn exceeds 1)

Keeping it simple

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

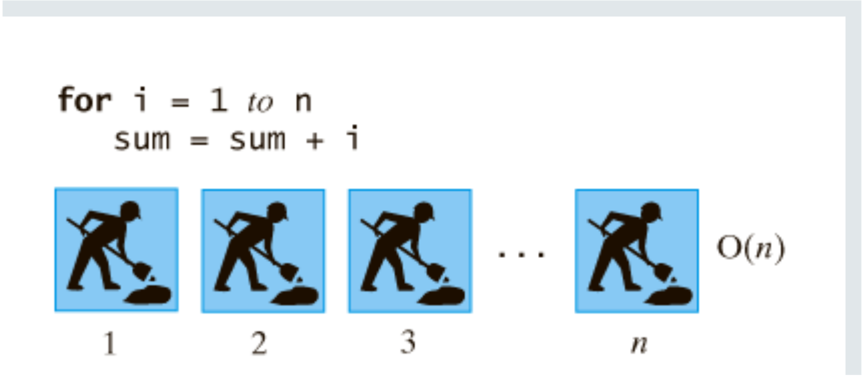
- $O(n^2)$
- $O(n \log n)$
- $O(n^2)$
- $O(n \log n)$

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 is 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 Efficiency

- 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 Efficiency

```
for i = 1 to n
{   for j = 1 to n
        sum = sum + 1
}
```

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 his 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 = $n*(n-1) / 2 = n^2/2 – n/2$
- Using our usual methodology we can show that this is $O(n^2)$

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;
}
```

One loop running n times = O(n)

Nested Loops

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

Nested for loops each running n times = $O(n^2)$ steps

Loop running constant number of times

- Suppose that your implementation of a particular algorithm appears in Java as follows:

```
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
```

- Two loops running to n, third loop runs a constant number of times → $O(n^2)$

How about this loop?

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

- Analyse the complexity of the above 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 above algorithm is O(1)

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

- You can check how long your 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;
```