Big O Analysis

What is Big O

- For starters, lets look at this in terms of **speed of execution only**.
- Think of Big O as a rough bucketing of algorithms based on how efficient they are… i.e., how fast they can perform.

- Keep in mind that these are loose estimates, and we are not accounting for every little thing. We will explain what we mean by this.

- When doing a Big O analysis for execution time, we are trying to figure out how execution time is expected to change with change in input size (denoted by N).
- So, N is the number of elements that the algorithm is to work upon.

- If sorting 10,000 numbers, the N is 10,000.
- If computing factorial of 10, N is 10.

O(1)

```
void MyFunction( int [] intArray, int N ) {
        print ("Number of items passed to MyFunction : " , N );
}
```

- Printing the number of items will take the same time if N is 1, or if N is 1000000 or 1000000000
- In other words, this does NOT depend on the size of input passed in.
- The time taken to run this code is the same, regardless of value of N.
- So, this time does NOT change when N changes.
- This is called constant time, or O(1)

- Note that O(1) does not necessarily indicate its very fast, it just means that the time does NOT depend on size of input.

- It could be a slow or a fast operation.

O(N)

```
void MyFunction( int [] intArray, int N ) {
        int sum = 0;
        for ii = 0 to  ii < N {
                sum = sum + intArray[ii];
}
```

- In this code, the for loop is executed N times.
- Lets say I call this function twice:
  - in the 1st call, N is 10
  - in 2nd call, N is 1000000000 (1 billion)
- Definitely the 2nd call will take much longer compared to first call.
- The number of iterations of the loop changes at the same rate as N.
  - If N doubles, so do the number of iterations
  - If N is halved, so do the number of iterations
- This is called O(N) complexity

- Example of O(N) operations
  - Finding a max element in an unsorted array
  - Finding all elements with a certain value in an unsorted array.
  - Finding if an element exists in an unsorted array

This is O(N):

```
void MyFunction( int [] intArray, int N )
{
        for ii = 0 to  ii < N
        {
                print ( intArray[ii] );
        }
}
```

$O(N^2)$

- $O(N^2)$ algorithms are usually more expensive than O(1) or O(N) algorithms.

- This is because their run time grows at a rate of square of the input size.

  - If input size goes from, say, 10 to 100, it is a 10 times growth in input.
  - The run time growth here will be from 100 to 10,000, i.e. 100 times growth.

  - If input size goes from 1000 to 1000,000, it's a 1000 times growth.
  - Run time growth is from 10^6 to 10^{12}, i.e., a 1000,000 times growth.

- As you can see, this does not scale well at all.
- For large values of N, the time taken would go up by a lot.

- $O(N^2)$ is also called quadratic

- Examples of $O(N^2)$ algorithm
    - Sum all elements in an N x N matrix.
    - Bubble sort algorithm
    - Insertion sort algorithm
        - http://bigocheatsheet.com/

```
void MyFunction( int [][] intArray2D, int N )
{
        int sum = 0;
        for row = 0 to  row < N {                 ← This line executes N times
                for col = 0 to col < N {          ← This line executes N times
                        sum = sum + intArray[row][col];   ← This line is executed N^2 times
                }
        }
}
```

Analogies:

- N people, all shake hands with each other.
- Directory print example. This can be changed slightly to get analogy for O(N) as well.
  - N people, so we print N directories, one for each person.
  - N^2 analogy: Last digit of every phone number is 1 more than it should be.
  - N analogy: Last digit of one phone number is 1 more than it should be.

O(log N)

- O(log N) algorithms are usually faster than O (N) algorithms.
- When we say log, we usually mean log base 2, ie, $\log_2$ (N)

- Rate of growth here is much smaller than N.

  - If input size goes from, say, 100 to 1000, it is a 10 times growth in input.
  - The run time growth here will be from about 7 to about 10 (so, about 1.5 times)

  - If input size goes from 1000 to 1000,000, it is a 1000 times growth in input.
  - The run time growth here will be from about 10 to about 20 (that is only 2 times)

  - If input size goes from 1000 to 1000,000,000, it is a 10^6 times growth in input (million times).
  - The run time growth here will be from about 10 to about 30.

- Examples of $O(\log_2 N)$ algorithm

  - Finding an element in a sorted array.

  - Finding an element in a Binary Search Tree.
    - If BST has 1000 elements (and is balanced), worst case it will take 10 searches to find the element (or know it is not there).

    - If BST has 10^9 elements (and is balanced), worst case it will take 30 searches to find the element (or know it is not there).

  - Insertion of element into a Binary Search Tree.

  - Deletion of element from a Binary Search Tree.

What is the complexity of the following functions:

```
void MyFunction_1( int N )
{
        for index = 1 to  index < N   ← How many times is this loop executed?
        {
                print index;
                index = index * 2;
        }
}

void MyFunction_2( int N )
{
        for index = 1 to  index < N   ← How many times is this loop executed?
        {
                print index;
                index = index * 3;
        }
}
```

What is the complexity of the following :

```
void MyFunction_3( int N )
{
        for index = 1 to  index < N   ← How many times is this loop executed?
        {
                print index;
                index = index  +  4;
        }
}
```

N Log (N)

- Generally means that you are doing a log N operation for each input item.
- Lot of efficient sorting algorithms have a complexity of N Log N.
- Quicksort best case is O(N log N)
- Mergesort and heapsort are both O(N log N)

Exponential

- Exponential algorithms are very expensive.
- The complexity is O (2^n)

- This means that with each increasing element, the runtime will double.
- 2^5 that is 32 operations
- 2^6 64 operations
- 2^{10} 1024 operations
- 2^{30} 1 billion (approx.) operations

- Example:
  - Lets say we have a crypto key of size 2 bits which we use for some basic encryption.
  - This would be trivial to break, because there can only be 4 possible combinations, and these can be obtained using brute force.
  - With a key size of 4 bits, brute force evaluation would give 16 possible combinations (4 times more), also trivial.

Exponential

With a key size of 64 bits, we have 2^{64} possible combinations, which is a lot.

And a key size of 65 bits yields 2 x 2^{64} possible combinations, which means an increase in the bit size by 1 doubled the possible combinations.

And a key size of 128 bits will have 2^{64} x 2^{64} possible combinations !!!

Imagine breaking that with a brute force attack.

From: https://en.wikipedia.org/wiki/Brute-force_attack

Breaking a symmetric 256-bit key by brute force requires 2^{128} times more computational power than a 128-bit key.

Fifty supercomputers that could check a billion billion (10^{18}) keys per second (if such a device could ever be made) would, in theory, require about 3×10^{51} years to exhaust the 256-bit key space !!!

Exponential

- Another example is computing all subsets of a given set of elements.

- If a set has n elements, it has 2^n subsets.
- So, set with 10 elements has 2^{10} subsets (1024)

- And a set with 11 elements has 2^{11} subsets (2048)
- Example:

  - Set with 2 elements: { a, b }
  - Enumeration of its subsets:
    - { }, {a}, {b}, {ab} ← So that's 2^2 = 4 subsets

  - Set with 3 elements: { a, b, c }
  - Enumeration of its subsets:
    - { }, {a}, {b}, {c}, {ab}, {ac}, {bc}, {abc} ← So that's 2^3 = 8 subsets

Exponential

- Another example is recursively computing Fibonacci numbers.
- This has 2^n calls to Fibonacci function.

```
int  Fibonacci( int n )
{
        if ( n < = 1 )
                return n;
        return Fibonacci( n – 1 )  + Fibonacci( n – 2 );
}
```

Exponential question ☺

If you take a really big piece of paper (regular paper thickness, like the one used in printers), and fold it a few times, lets see how thick it can get:

Num folds	**Layers of paper**	**Thickness**	**Units**
0	1	0.05	mm
1	2	0.10	mm
10	1024	2	in.
20	1048576	2064	in.

Num folds	Layers of paper	Thickness	Units	
30	1073741824	34	miles	
40	1.09951E+12	34360	miles	> circumference of earth (25K miles)
43	8.79609E+12	274878	miles	distance to moon
44	1.75922E+13	549756	miles	roundtrip to moon

Num folds	Layers of paper	Thickness	Units	
47	1.40737E+14	4398047	miles	4 million miles
51	2.2518E+15	70368744	miles	70 million miles
52	4.5036E+15	140737488	miles	140m miles, which is > distance to sun (93m miles)

So, as we see, exponential rate makes the numbers go up very very fast.

Factorial

An algorithm with factorial complexity is O (n!)

This is even more expensive than exponential.

Factorial (5) = 1 * 2 * 3 * 4 *5

Factorial (N) = 1 * 2 * 3 * … (N-1) * N

For even small values of n, this becomes very very expensive.

10! = 3,628,800

15! = 1,307,674,368,000

20! = 2,432,902,008,176,640,000

LAB (Stop class recording)

1. Write a for loop in which:
   - The control variable, say, ii, starts with an initial value of 1.
   - ii increases by a factor of 2 in each iteration, i.e., ii is multiplied by 2 in every iteration.
   - Stopping condition for the loop is ii <= 1000,000,000 , i.e., 1 billion
   - In each iteration, print the value of ii, and then a new line, i.e., value of ii is on a line by itself.

   - How many lines does your output have?
     - In other words, how many iterations did it take to get to the stopping condition?
2. What if the value of ii increases by a factor of 10 in each iteration, how many lines will your output have?

3. Write a function that:
   - Takes an integer as its parameter
   - Returns the factorial of the passed in parameter.
   - So, the function signature will look something like:
     - long Factorial (int n)

   - What validation do you need to perform on the parameter?

   - Factorial of a number is defined as the product of all numbers starting from 1 to that number.
   - E.g.:
     - Factorial (3) = 1 * 2 * 3 = 6
     - Factorial (7) = 1 * 2 * 3 * 4 * 5 * 6 * 7 = 5040
       - = Factorial (6) * 7

LAB

1. Write a for loop in which:
   - The control variable, say, ii, starts with an initial value of 1.
   - ii increases by a factor of 2 in each iteration, i.e., ii is multiplied by 2 in every iteration.
   - Stopping condition for the loop is ii <= 1000,000,000 , i.e., 1 billion
   - In each iteration, print the value of ii, and then a new line, i.e., value of ii is on a line by itself.

   - How many lines does your output have?
     - In other words, how many iterations did it take to get to the stopping condition?

2. What if the value of ii increases by a factor of 10 in each iteration, how many lines will your output have?

LAB

3. Write a function that:

- Takes an integer as its parameter
- Returns the factorial of the passed in parameter.
- So, the function signature will look something like:
  - long Factorial (int n)

- What validation do you need to perform on the parameter?

- Factorial of a number is defined as the product of all numbers starting from 1 to that number.
- E.g.:
  - Factorial (3) = 1 * 2 * 3 = 6
  - Factorial (7) = 1 * 2 * 3 * 4 * 5 * 6 * 7 = 5040
    - = Factorial (6) * 7

- So, repeating what we said earlier:
  - Big O analysis is a rough bucketing of algorithms.
  - We are simplifying a lot of things and making some assumptions in order to categorize algorithms into these buckets.
  - Does it make sense… are we doing the right thing?
  - The graphs coming up certainly makes it look like we are:

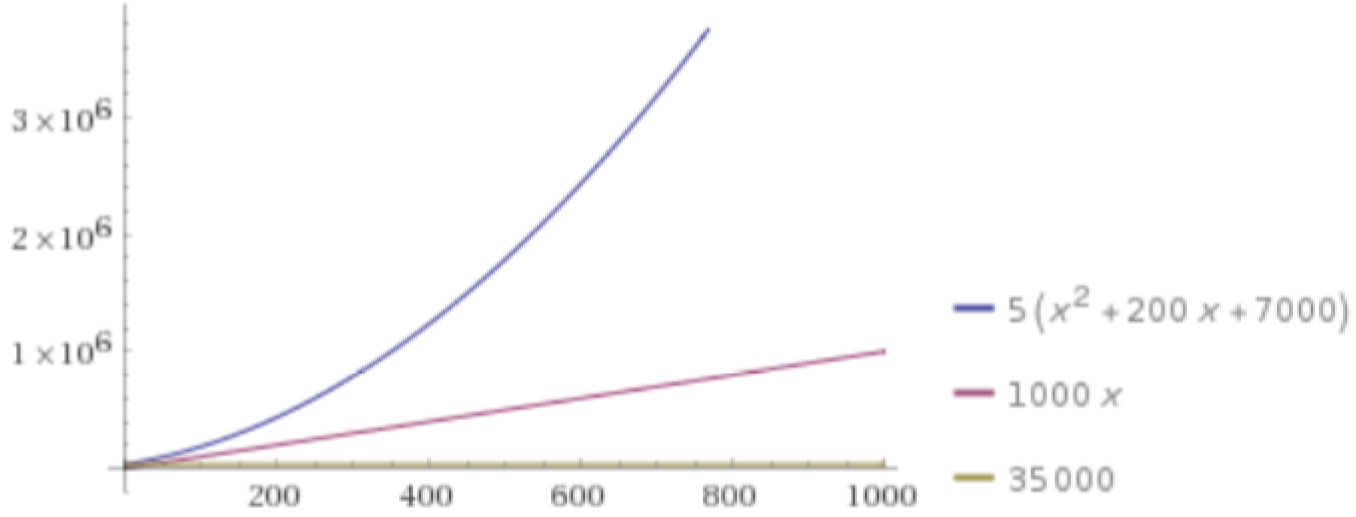
- It graphs three functions

$f1(x) = 5x^2 + 1000x + 35000$

$f2(x) = 1000x$

$f3(x) = 35000$

- As you can see, the term that has **$5x^2$** dominates, especially so when value of x increases.



- http://www.wolframalpha.com/input/?x=0&y=0&i=plot+++5x%5E2+%2B+1000x+%2B+35000,+1000x,+35000+from+x%3D0+to+1000

Higher order term prevails for large x

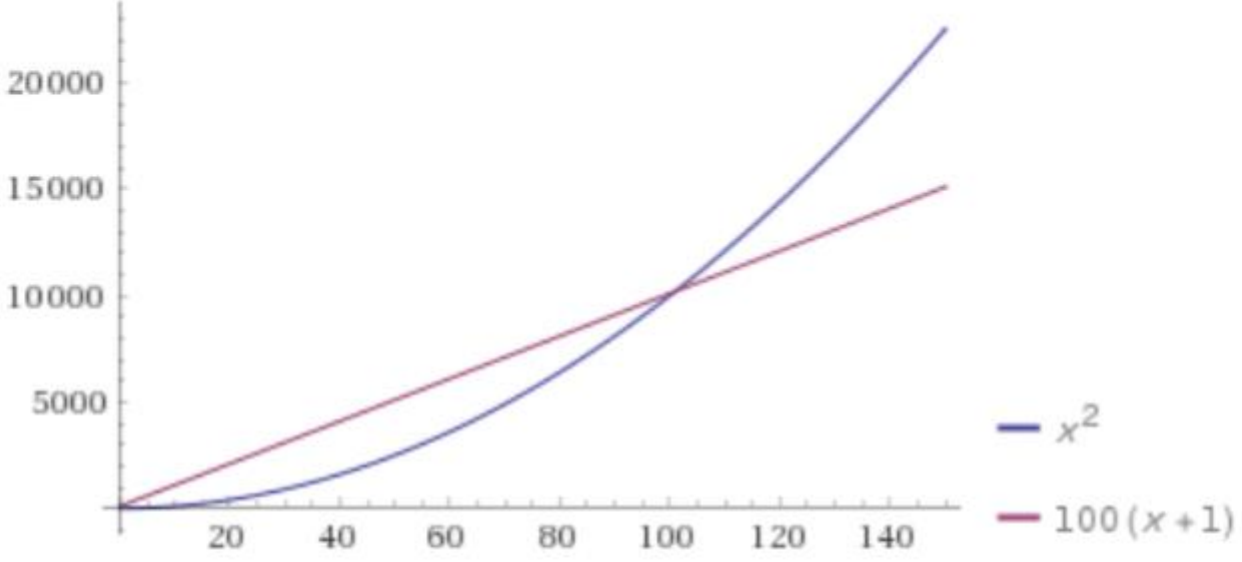
x goes from 0 to 150

$f1(x) = x^2$

$f2(x) = 100\,x + 100$

For x < about 110, **f1** (quadratic) is faster than **f2** (linear).

But f1 increases very rapidly (see next slide)



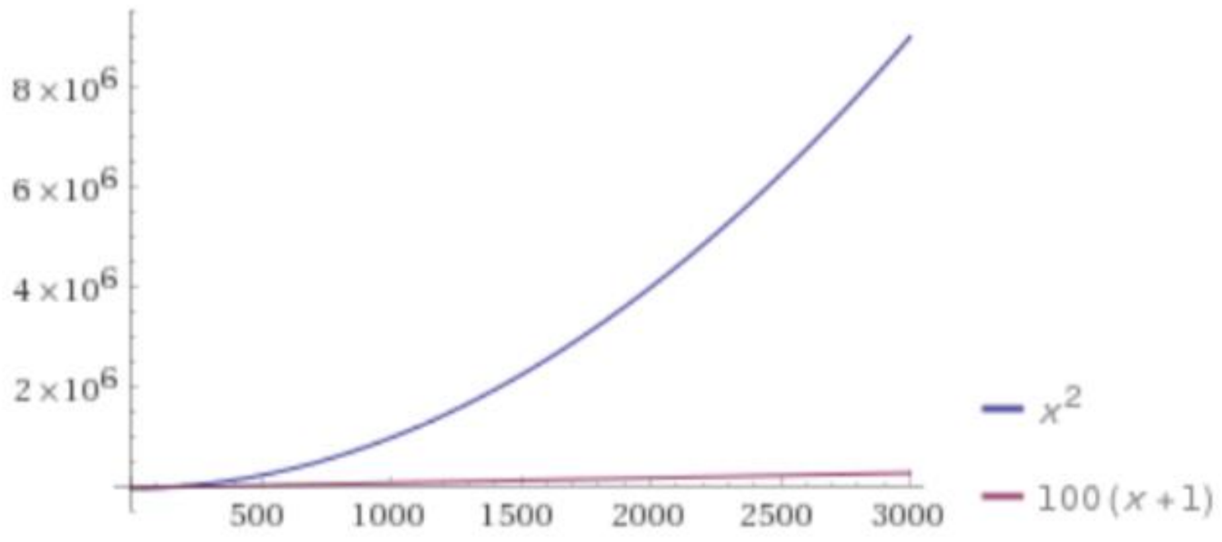
- http://www.wolframalpha.com/input/?x=0&y=0&i=plot+++x%5E2+,+100x+%2B+100+from+x%3D0+to+150

See how rapidly **f1** (quadratic) increases compared to **f2** (linear) for higher values of x:

x goes from 0 to 3000

f1(x) = x^2

f2(x) = 100 x + 100



http://www.wolframalpha.com/input/?x=0&y=0&i=plot+++x%5E2+,+100x+%2B+100+from+x%3D0+to+3000

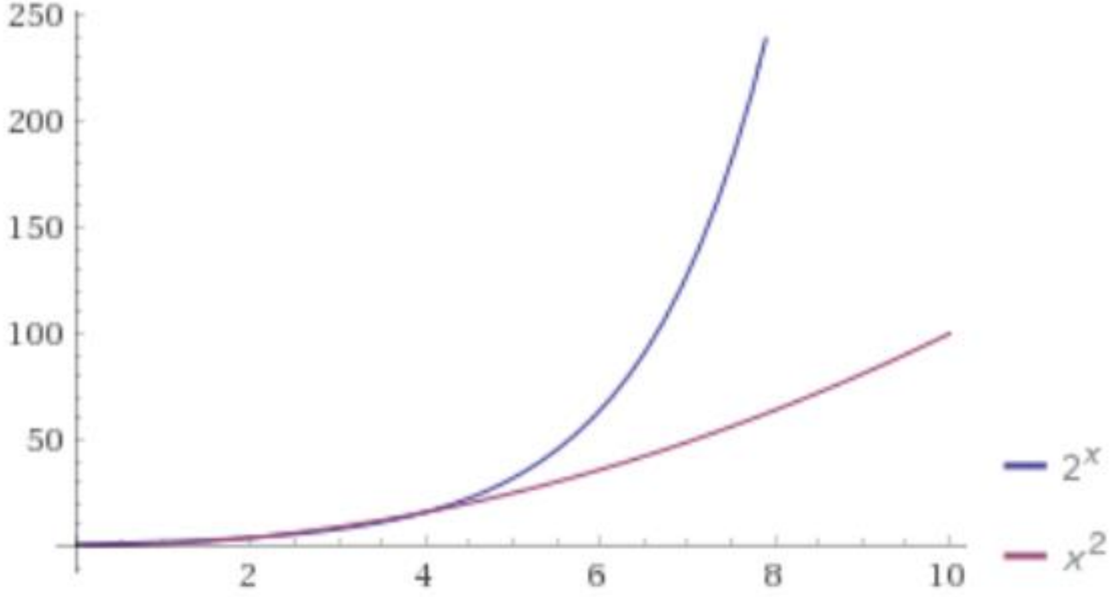
Exponential vs Quadratic

See how rapidly **f1** (exponential) increases compared to **f2** (quadratic)

The value of x here is very small, goes from 0 to 10

f1(x) = 2^x

f2(x) = x^2



- http://www.wolframalpha.com/input/?x=0&y=0&i=plot+++2%5Ex+,+x%5E2+from+x%3D+0+to+10

Exponential vs Quadratic

And here the value of x goes from 25 to 30.

$f1(x) = 2^x$

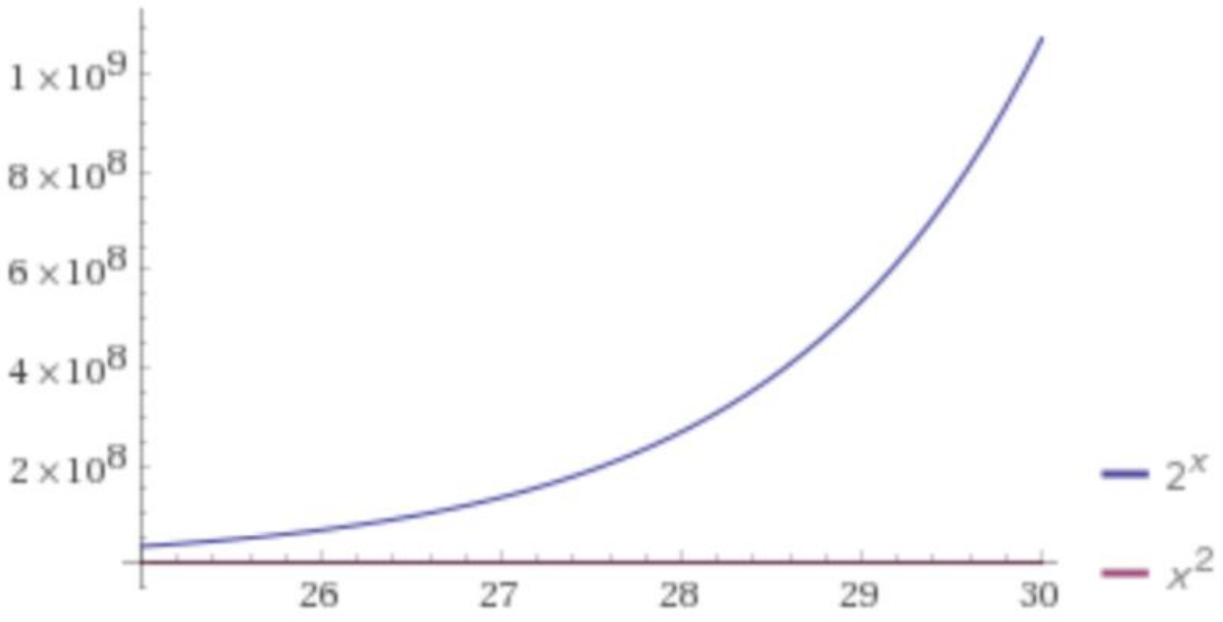
$f2(x) = x^2$

For x=30:

f1 is about a billion

and

f2 is about 900 only



- http://www.wolframalpha.com/input/?x=0&y=0&i=plot++++2%5Ex,++x%5E2++from+x%3D+25+to+30

Factorial vs Exponential

value of x goes from 1 to 10

f1(x) = x!

f2(x) = 2^x

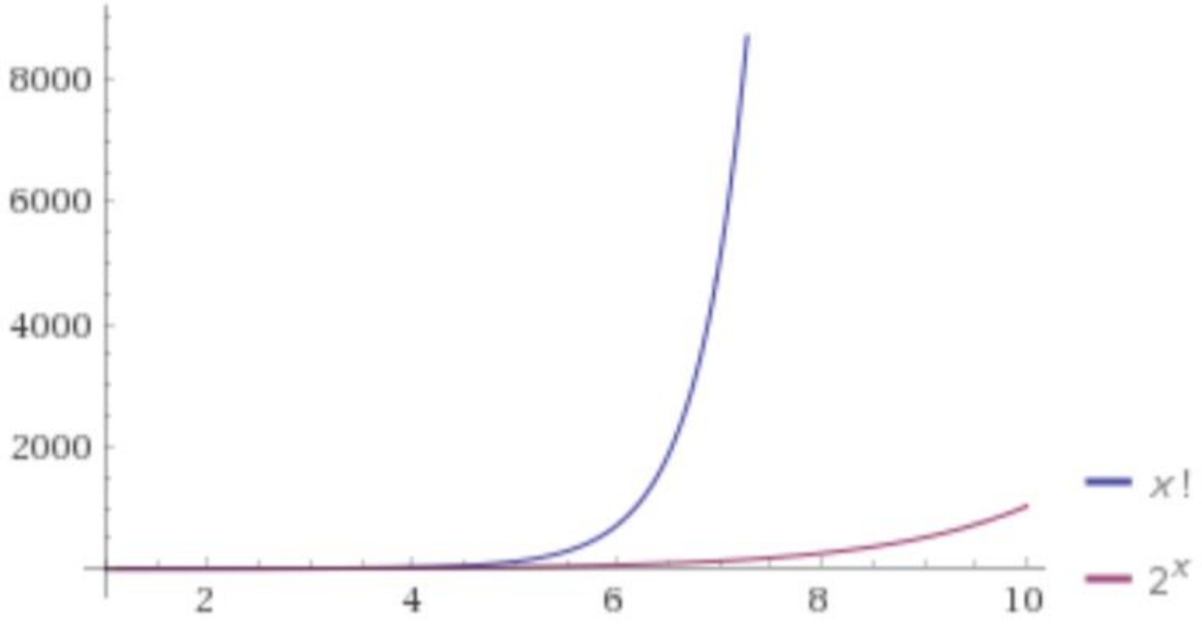
For x=10:

f1 is about

3.6 million (graph shows incorrect value)

and

f2 is about 1000 only



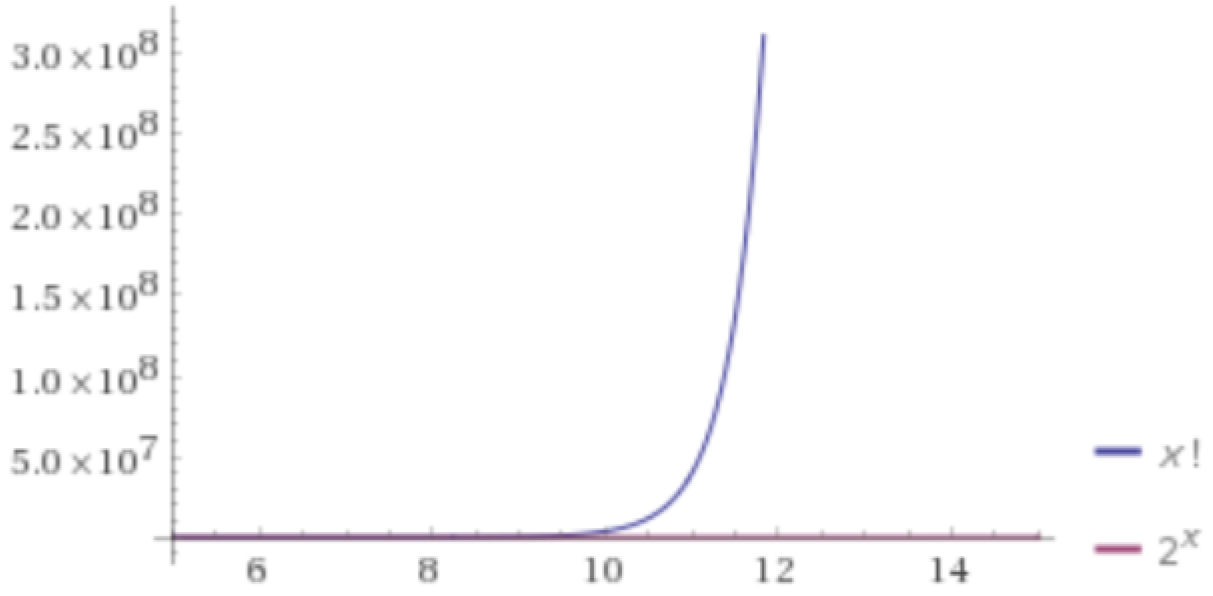
- http://www.wolframalpha.com/input/?x=0&y=0&i=plot++++x!,+2%5Ex,++from+x%3D+5+to+15

Factorial vs Exponential

value of x goes from 5 to 15.

$f1(x) = x!$

$f2(x) = 2^x$



For x=30:

f1 is about 2.x X 10^{32}

and

f2 is about 1 X 10^{9} only

- http://www.wolframalpha.com/input/?x=0&y=0&i=plot+++x!,+2%5Ex,++from+x%3D+5+to+15

- So, in order of growth with respect to input size, we have:

- O(1) constant
- O(log N) sub linear
- O(N) linear
- O(N log N)
- O(N^2) quadratic
- O(2^N) exponential
- O(N!) factorial

- Polynomial time algorithms are algorithms that are based on powers of N, so constant time, linear, quadratic, cubic, etc. are all polynomial time algorithms.

“Small” values of N

- Note that its not always the case that, say, an O(N^2) algorithm would be slower than an O(N). Yes, in general that is the case, but if your input size (N) is “small”, then it could be that the O(N^2) algorithm is faster.

- As we see in the graph below, for values of x < approx. 110, the quadratic function has lower values than the linear function.

“Directory look up stair case” analogy