COMPLEXITY ANALYSIS

Software Life Cycle

- **Requirements** – specifications for a given project that includes what is to be input and what is to be output.
- **Analysis** – the problem is broken down into manageable pieces typically using a top-down approach where the program is continually refined into more manageable pieces. During this phase there are several alternative solutions that are developed and compared. We will talk how to compare these pieces shortly.
- **Design** – this continues the work of the analysis phase and includes data objects the program needs and the operations performed on the data objects. The data types during this phase are ADTs and no implementation details exist during this phase.
- **Refinement and coding** – actual representations for each ADT are developed and algorithms for each operation are written.
- **Verification** - program correctness must be developed including extensive testing using various datasets.

Once You're Done

1. Are the original specifications met by the program?
2. Is the program implemented correctly and work correctly?
3. Is there documentation that shows how to use the program?
4. Does the program contain well defined modules and strive for reusability?
5. How readable is the code?
6. How efficiently and effectively is storage used?
7. Does the program have an acceptable running time?

Complexity and Algorithm Efficiency

- Questions 6. and 7. are best identified by the terms "Space Complexity" and "Time Complexity."
- For each of the data structures we will discuss in this course, we will want to know the associated space complexity and time complexity.
- We need some method to talk about complexity issues.

Algorithm Efficiency

- Let's look at the following algorithm for initializing the values in an array:

```
int n = 500;
int i;
int counts[n];
for ( i = 0; i < n; ++i)
{
    counts[i] = 0;
}
```

- What does the time that the algorithm takes depend on? Which variable?

Algorithm Efficiency

- In that algorithm, we have one loop that processes all of the elements in the array.
- Intuitively:
  - If **`n`** was half of its value, we would expect the algorithm to take half the time.
  - If **`n`** was twice its value, we would expect the algorithm to take twice the time.
- That is true and we say that the algorithm efficiency relative to **`n`** is linear.

Algorithm Efficiency

- Let's look at another algorithm for initializing the values in a different array:

```
int n = 500;
int i;
int counts[n][n];
for ( i = 0; i < n; i++)
{
  for ( j = 0; j < n; j++)
  {
    counts[i][j] = 0;
  }
}
```

- What does the length of time that the algorithm takes to execute depend on?

Algorithm Efficiency

- In the second algorithm, we have two nested loops to process the elements in the two dimensional array.
- Intuitively:
  - If `n` is half its value, we would expect the algorithm to take one quarter the time.
  - If `n` is twice its value, we would expect the algorithm to take quadruple the time.
- That is true and we say that the algorithm efficiency relative to `n` is quadratic.

Big-O

- Algorithms are measured according to a notation called "Big-O" notation (e.g. O(N)).

- How does the execution time change with a change in data size?

- Big-O measures the computational complexity of a particular algorithm based on the number of steps relative to some data size, N
    - Number of items

If you add more data, runtime goes up. By how much?

Big-O

- Measures the growth rate of an algorithm as the size of its input grows.
  - Huh?
- "O" is a math function that helps estimate how much longer it takes to run n inputs versus n+1 inputs (or n+2, 2n, 3n…).
- "O" doesn't care what programing language you use! Only cares about the underlying algorithm.

Big-O

- What Doesn't "O" do?
  - Doesn't tell you that algorithm A is faster than algorithm B for a particular input.
  - Why not? Only tells you if one grows faster than another in a general sense for all inputs.
  - Usually concerned with very large data inputs.
  - Called asymptotic algorithm analysis.

Big-O Notation

- We use a shorthand mathematical notation to describe the efficiency of an algorithm relative to any parameter n as its “Order” or Big-O.
  - We can say that the first algorithm is O(n).
  - We can say that the second algorithm is O(n2).

- For any algorithm that has a function g(n) of the parameter n that describes its length of time to execute, we can say the algorithm is O(g(n)).

- We only include the fastest growing term and ignore any multiplying by or adding of constants.

What is N? Why?

```
#define TRUE 1
#define FALSE 0

int isSorted (const int nums[], int howmany)
{
  int bSorted;
  int i;

  bSorted = TRUE;

  for (i = 0; i < (howmany - 1); ++i)
  {
    if (nums[i] > nums[i + 1])
    {
      bSorted = FALSE;
    }
  }

  return bSorted;
}
```

How many times is each statement executed per invocation of isSorted()?

What is the overall complexity? O(   ) ?

Big-O

- Determines the relative speeds of algorithms, but doesn’t depend on:
  - Hardware used (Mac vs. PC)
  - Clock speed of the processor
  - The compiler used
  - The programming language used

Complexity Scenerios

- When looking at computational complexity, we typically examine three scenarios:
  - Best Case Performance
  - Average Case Performance
  - Worst Case Performance

Not-So-Simple Loops

```
#define TRUE 1
#define FALSE 0

int isPrime (int num)
{
  int bIsPrime = TRUE
  int i;

  for (i = 2; i * i < n; ++i)
  {
    if (0 == n % i)
    {
      bIsPrime = FALSE;
    }
  }

  return bIsPrime;
}
```

But what happens if it exits early?

- Best case?
- Average case?
- Worst case?

Complexity Categories

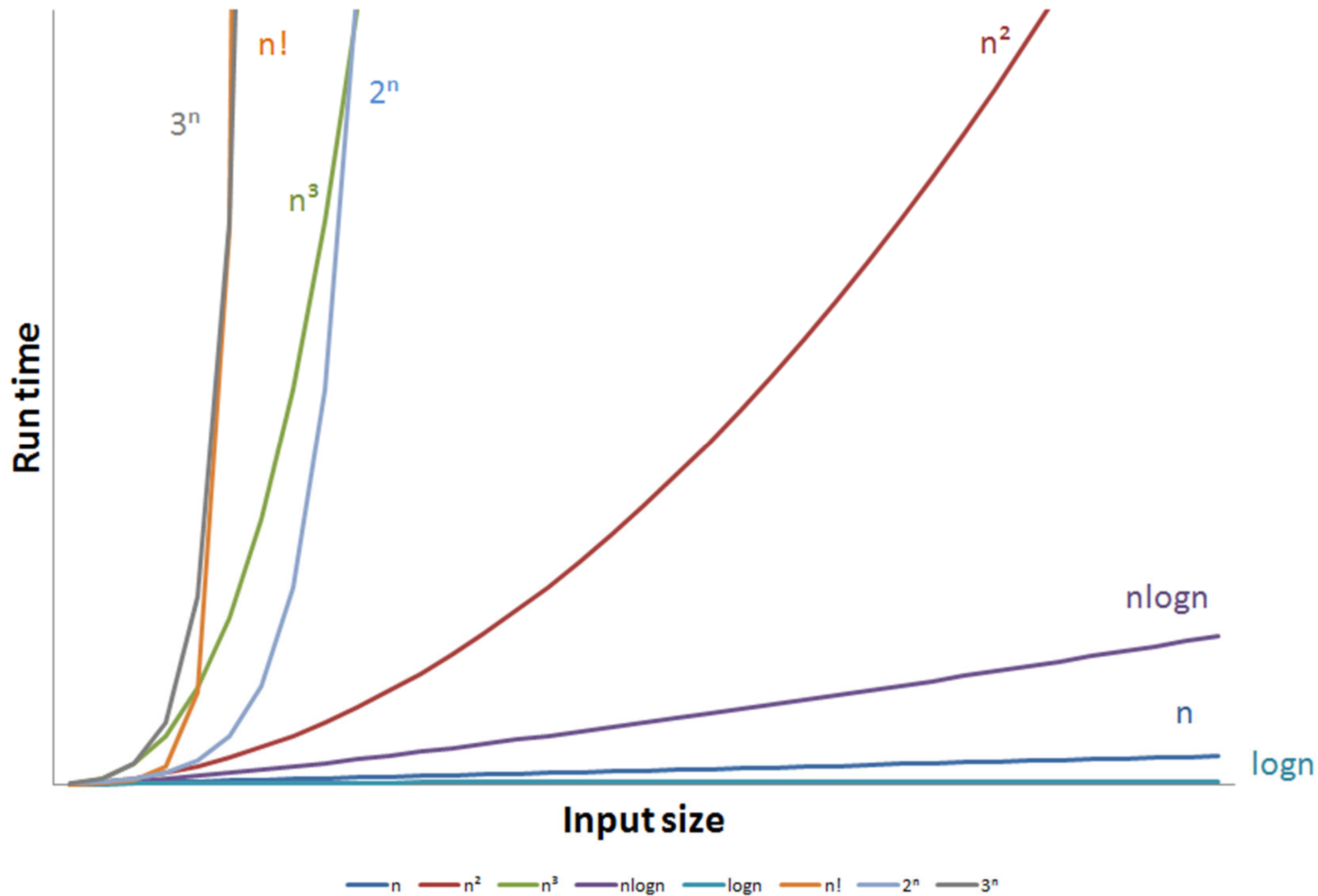
- Typically we find that computational complexities fall into polynomial, logarithmic, or exponential time and are named:
  - $O(1)$ – constant (what might fall into this category?)
  - $O(\log_2 N)$ – logarithmic
  - $O(N)$ – linear
  - $O(N\log_2 N)$ – Log linear
  - $O(N^2)$ – quadratic
  - $O(N^3)$ – cubic
  - $O(2^N)$ – exponential
  - $O(N!)$ - factorial

Growth Rates

- Let's examine how the complexity grows for various computing times.

log2N	Nlog2N	N2	N3	2n
1	2	4	8	4
2	8	16	64	16
3	24	64	512	256
4	64	256	4096	65536

Growth Rates



http://blog.philenotfound.com/2011/12/brief-intro-to-algorithmic-time.html

Big-O for a Problem

- O(g(n)) for a problem means there is some O(g(n)) algorithm that solves the problem.
- Don’t assume that the specific algorithm that you are currently using is the best solution for the problem.
- There may be other correct algorithms that grow at a smaller rate with increasing n.
- Many times, the goal is to find an algorithm with the smallest possible growth rate.

Role of Data Structures

- We can do the same thing for algorithms in our computer programs.
- Example: Finding a numeric value in a list
  - If we assume that the list is unordered, we must search from the beginning to the end .
    - On average, we will search
    - Worst case, we will search
    - Best case, we will search
    - Algorithm is O( ), where n is size of array

Role of Data Structures

- The difference in the structure of the data between an unordered list and an ordered list can be used to reduce algorithm Big-O.
- This is the role of data structures and why we study them.
- We need to be as clever in organizing our data efficiently as we are in figuring out an algorithm for processing it efficiently.

Identify Big-O

	Best	Worst	Average
strLength			
strEqual			
strConcat			
strAppend			
strReverse			
strClear			
strCopy			

```
typedef struct String
{
  int length;
  char data[1024];
} String;
```

What is n?