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

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:

```
const int N = 500;
int i;
int aCounts[N];
for ( i = 0; i < N; ++i)
{
    aCounts[i] = 0;
}
```

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

Algorithm Efficiency

- In the previous 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:

```
const int N = 500;
int i;
int aCounts[N][N];
for (i = 0; i < N; i++)
{
  for (j = 0; j < N; j++)
  {
    aCounts[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 a nested loop 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

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 $O(n^2)$.
- 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 aNums[], int howmany)

{

  int bSorted;
   int i;

  bSorted = TRUE;

  for (i = 0; i < (howmany - 1); ++i)
     {

     if (aNums[i] > aNums[i + 1])
        {
          bSorted = FALSE;
        }
     }

  return bSorted;

}
```

What is the Complexity?

If **isSorted** is called only one time:

1) How many times is the statement bSorted = true; executed?
2) How many times is the for statement executed?
3) How many times is the if statement executed?
4) How many times is the statement return bSorted; executed?
5) What am I missing?
6) What is the overall time complexity of function isSorted? 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:

1) Best Case Performance

2) Average Case Performance

3) Worst Case Performance

Complexity Categories

Typically we find that computational complexities fall into polynomial, logarithmic, or exponential time and are named:

1) $O(1)$ – constant

2) $O(\log_2 N)$ – logarithmic

3) $O(N)$ – linear

4) $O(N\log_2 N)$ – Log linear

5) $O(N^2)$ – quadratic

6) $O(N^3)$ – cubic

7) $O(2^N)$ – exponential

8) $O(N!)$ - factorial

Growth Rates

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

N	log_2N	$Nlog_2N$	N^2	N^3	2^n
2	1	2	4	8	4
4	2	8	16	64	16
8	3	24	64	512	256
16	4	64	256	4096	65536

Growth Rates Graphically

Big-O Complexity

Operations

1000
900
800
700
600
500
400
300
200
100
0

0 10 20 30 40 50 60 70 80 90 100

Elements

O(1)
O(logn)
O(n)
O(nlogn)
O(n^2)
O(2^n)
O(n!)

Identify Big-O

	Average				Worst			
		Search	Insertion	Deletion		Search	Insertion	Deletion
Unordered Array								
Ordered Array								
Singly Linked List								

Identify Big-O

Function	Best	Worst	Average
strLength			
strEqual			
strConcat			
strAppend			
strReverse			
strClear			
strCopy			

What is N?

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

Formal Complexity Analysis

- Formally, we define Big-O as follows:

Function f(n) is O(g(n)) iff there exist positive constants c and n_0 such that f(n) <= cg(n) for all n, where n >= n_0.

What is happening?

```
for (i = 0; i < howmany; ++i)
{
  for (j = i + 1; j < howmany; ++j)
  {
    if(aNums[i] < aNums[j])
    {
      temp = aNums[i];
      aNums[i] = aNums[j];
      aNums[j] = temp;
    }
  }
}
```

What is the Computing Complexity?

In this case, the N we are talking about is the variable howmany. What we need to figure out is how many times the segment below is executed.

```
if(aNums[i] < aNums[j])
{
   temp = aNums[i];
   aNums[i] = aNums[j];
   aNums[j] = temp;
}
```

Number of Iterations

For various values of i, let's take a look:

i	# of iterations
0	N - 1
1	N - 2
2	N - 3

and you get the picture

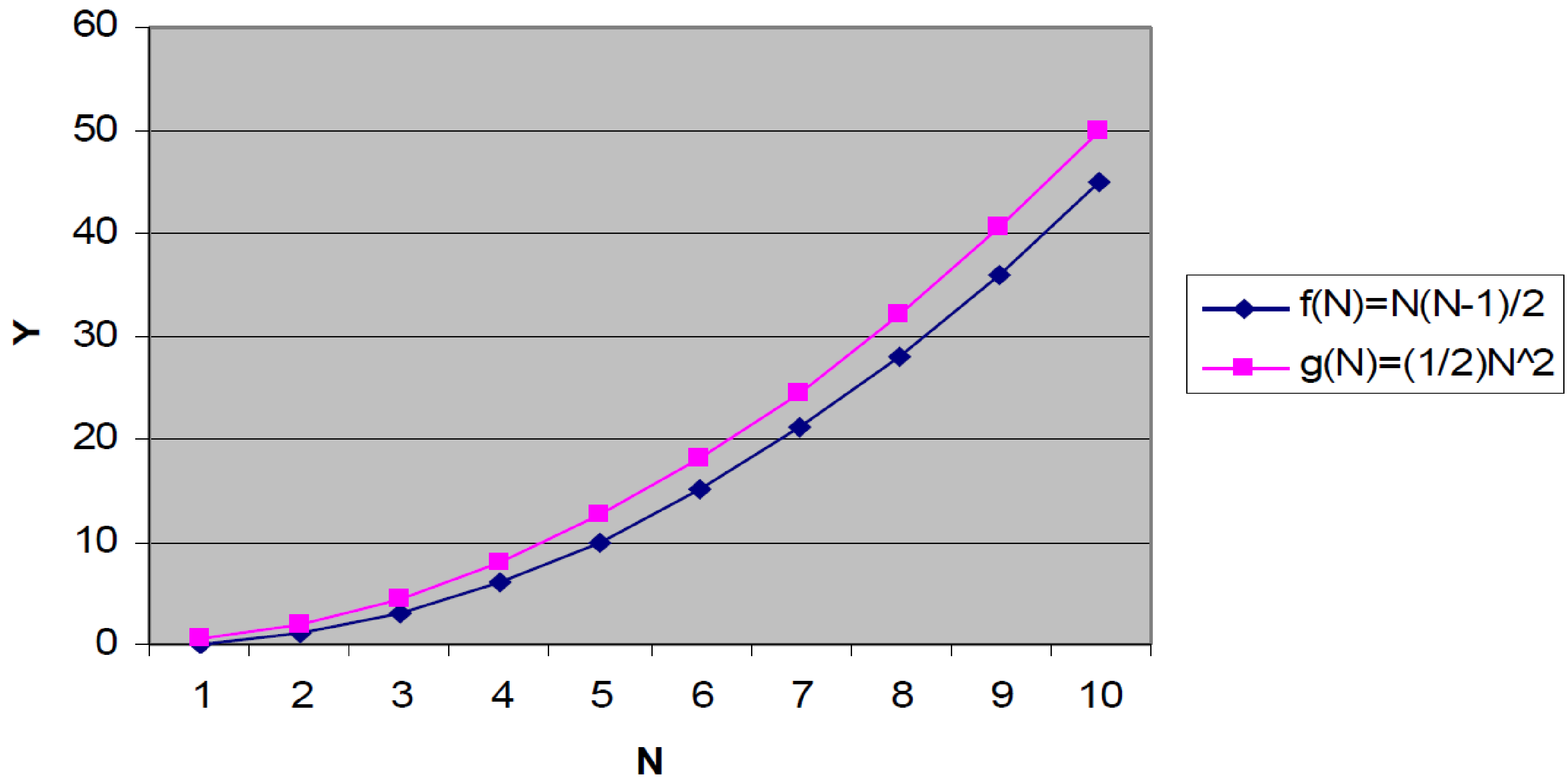
What is f(n)?

- This means that if the function f represents the number of executions of the above segment, then f(N) = (N-1) + (N-2) + (N-3) + ... + 2 + 1.
- Those who have taken a statistics class or studied summations can see that this equates to f(N) = N(N-1)/2.
- We can see that this function f can be bounded by some polynomial of N^2.

Not so obvious

- What might not be so obvious is that:
- $f(n) <= (1/2)n^2$, for $n >= 1$ and therefore, $n_0 = 1$, $g(n) = n^2$, and $c = 1/2$.
- This implies that $f(n)$ is $O(n^2)$.

Graphically



Other Computing Complexities

Problem: Give an algorithm that works in each of the following times:

1. $O(1)$
2. $O(n)$
3. $O(\log_2 n)$
4. O(n^2)
5. $O(n \log_2 n)$