Some Basic Concepts

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

Big-O

- Algorithms are measured according to a notation called "Big-O" notation (e.g. O(N)).
- Big-O measures the computational complexity of a particular algorithm based on the number of iterations, compares, assignment statements, and so on relative to some number N (the number of data items in the data structure).

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;

}
```

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(__)

Complexity Scenerios

When looking at computational complexity, we typically examine three scenerios:

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:

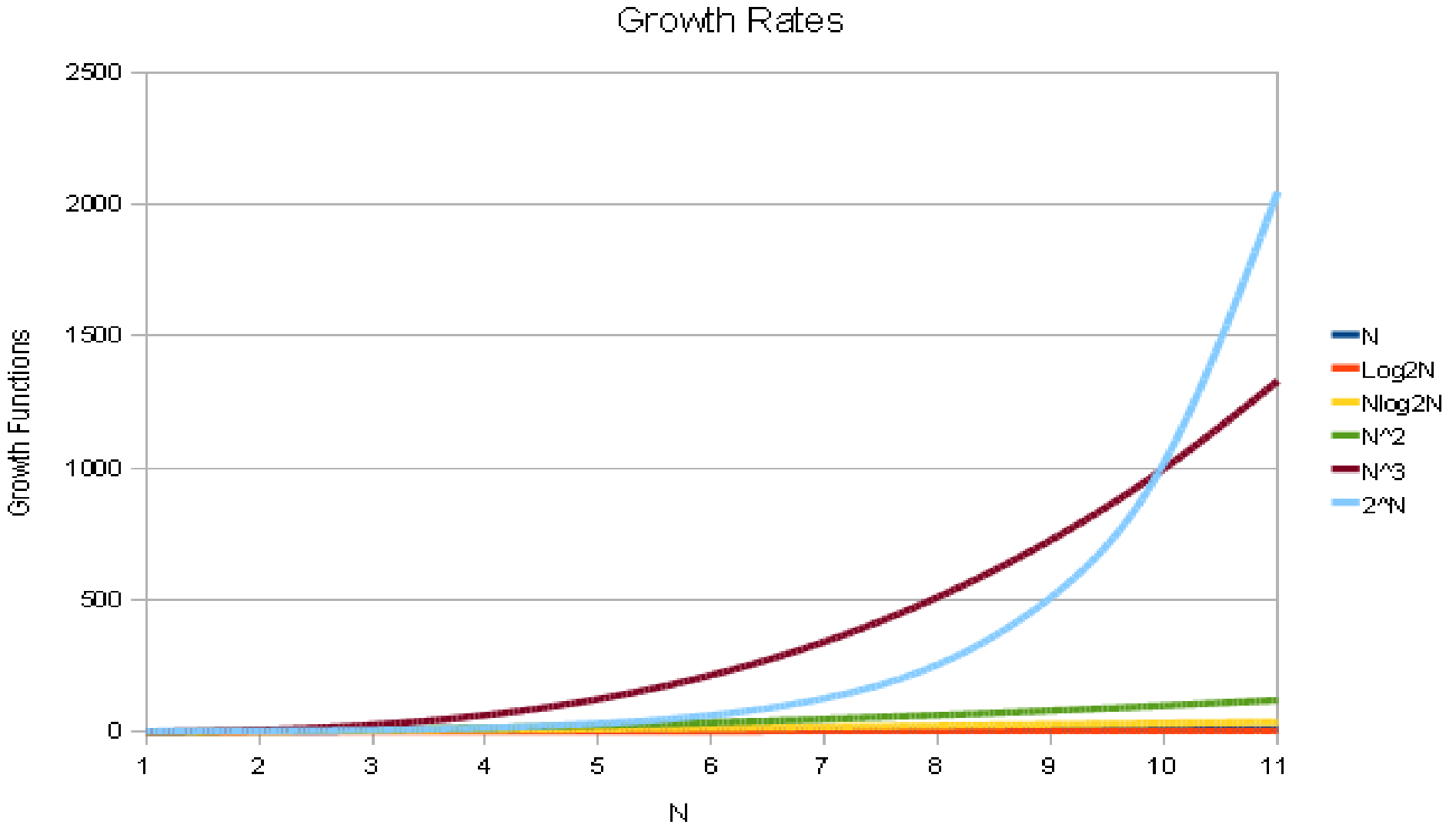
1) $O(1)$ – constant
2) $O(\log_2 N)$ – logarithmic
3) $O(N)$ – linear
4) $O(N\log_2 N)$ – Log linear
5) $O(N^2)$ – quadratic
6) $O(N^3)$ – cubic
7) $O(2^N)$ – exponential
8) $O(N!)$ - factorial

Growth Rates

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

N	$\log_2 N$	$N \log_2 N$	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



Identify Big-O

What is the **worst** case Big-O for:

Function	String Representation #1	String Representation #2
strLength		
strEqual		
strConcat		
strAppend		
strReverse		
strClear		
strCopy		

Identify Big-O

What is the **best** case Big-O for:

Function	String Representation #1	String Representation #2
strLength		
strEqual		
strConcat		
strAppend		
strReverse		
strClear		
strCopy		

Identify Big-O

What is the **average** case Big-O for:

Function	String Representation #1	String Representation #2
strLength		
strEqual		
strConcat		
strAppend		
strReverse		
strClear		
strCopy		