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

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

Two types of performance

This algorithm/datastructe takes N^3 steps to run

- We care about this one

This loop takes 10 instructions but I can rewrite it to take 9!

- We don't care about this one in this class
- Later in life, this will be important
- Many people use this as an excuse to write bad code

But I can save one instruction...

- You are not smarter than the compiler writers

```
gcc -O# -o runMe driver.o stack.o -Wall
```

# is zero to 3

  - Zero = default, no speed optimization
    - best for debugging
  - 1,2,3 increasing levels of speed optimization
    - Almost no change of the debugger working

http://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html

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)).

- 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 steps relative to some data size, N
  - Number of items

Add more data, runtime goes up. **By how much?**

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(   ) ?

Complexity Scenerios

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

- Best Case Performance
- Average Case Performance
- Worst Case Performance

Complexity Categories

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

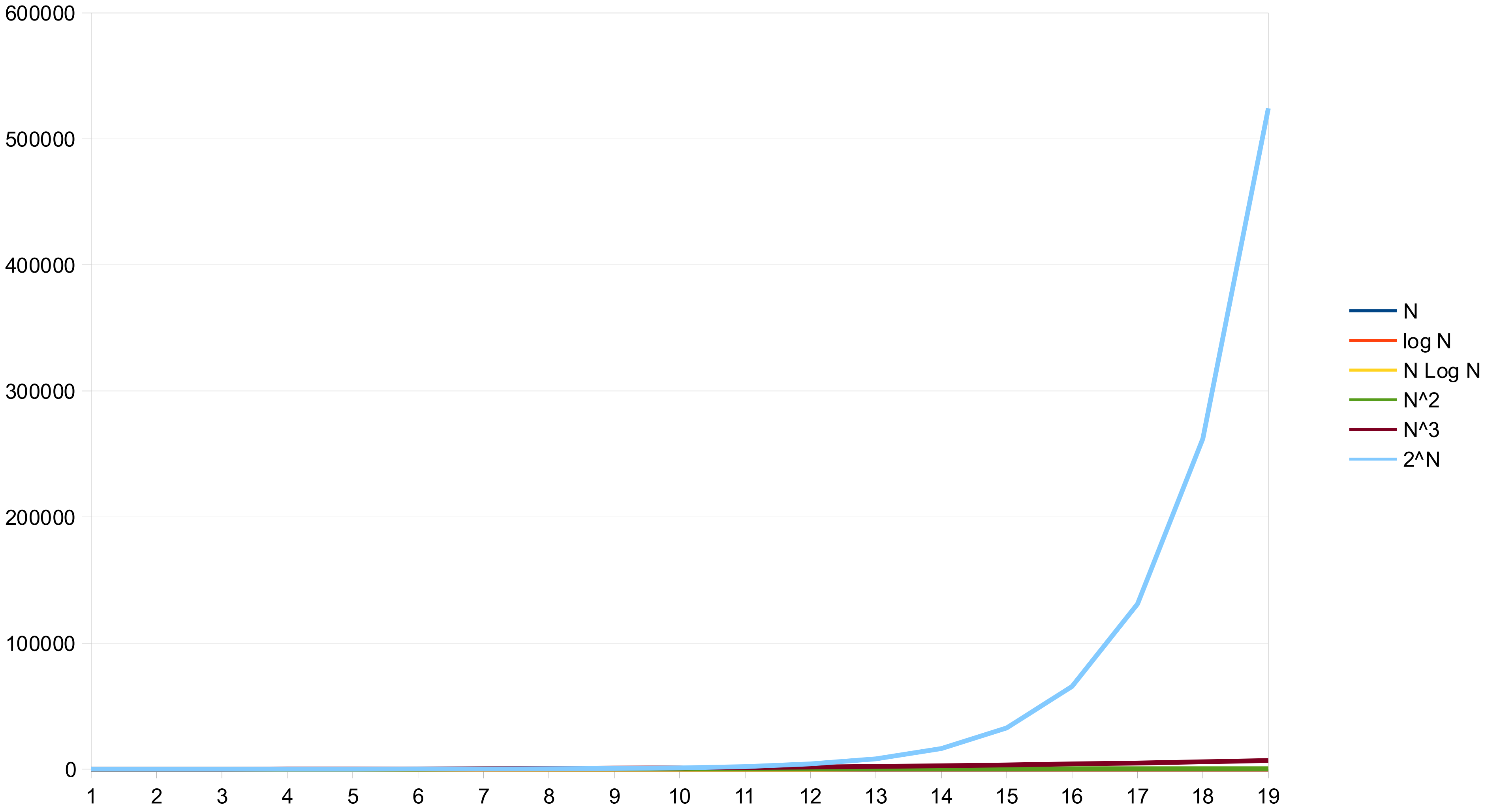
- O(1) – constant (what might fall into this category?)
- O($\log_2 N$) – logarithmic
- O(N) – linear
- O($N\log_2 N$) – Log linear
- O(N^2) – quadratic
- O(N^3) – cubic
- O(2^N) – exponential
- O(N!) - factorial

Growth Rates

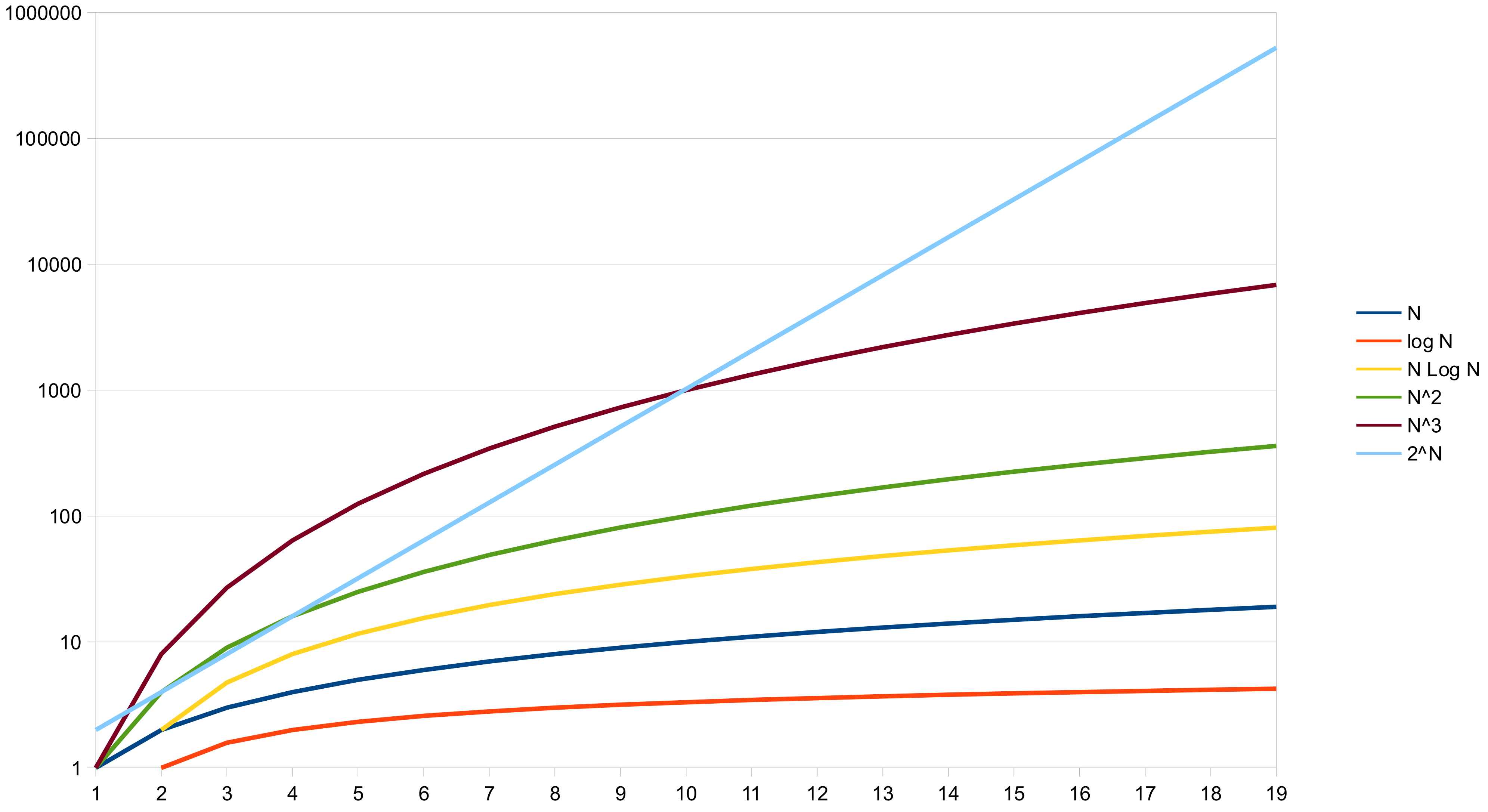
Let's examine how the complexity grows for various computing times.

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



log scale, y-axis



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 n0 such that f(n) <= cg(n) for all n, where n >= n0.

Upper Bound

What is happening?

```
for (i = 0; i < howmany; ++i)
{
  for (j = i + 1; j < howmany; ++j)
  {
    if(nums[i] < nums[j])
    {
      temp = nums[i];
      nums[i] = nums[j];
      nums[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 statement below is executed.

```
if(nums[i] < nums[j])
```

Why do we ignore the stuff inside the if() ?

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.

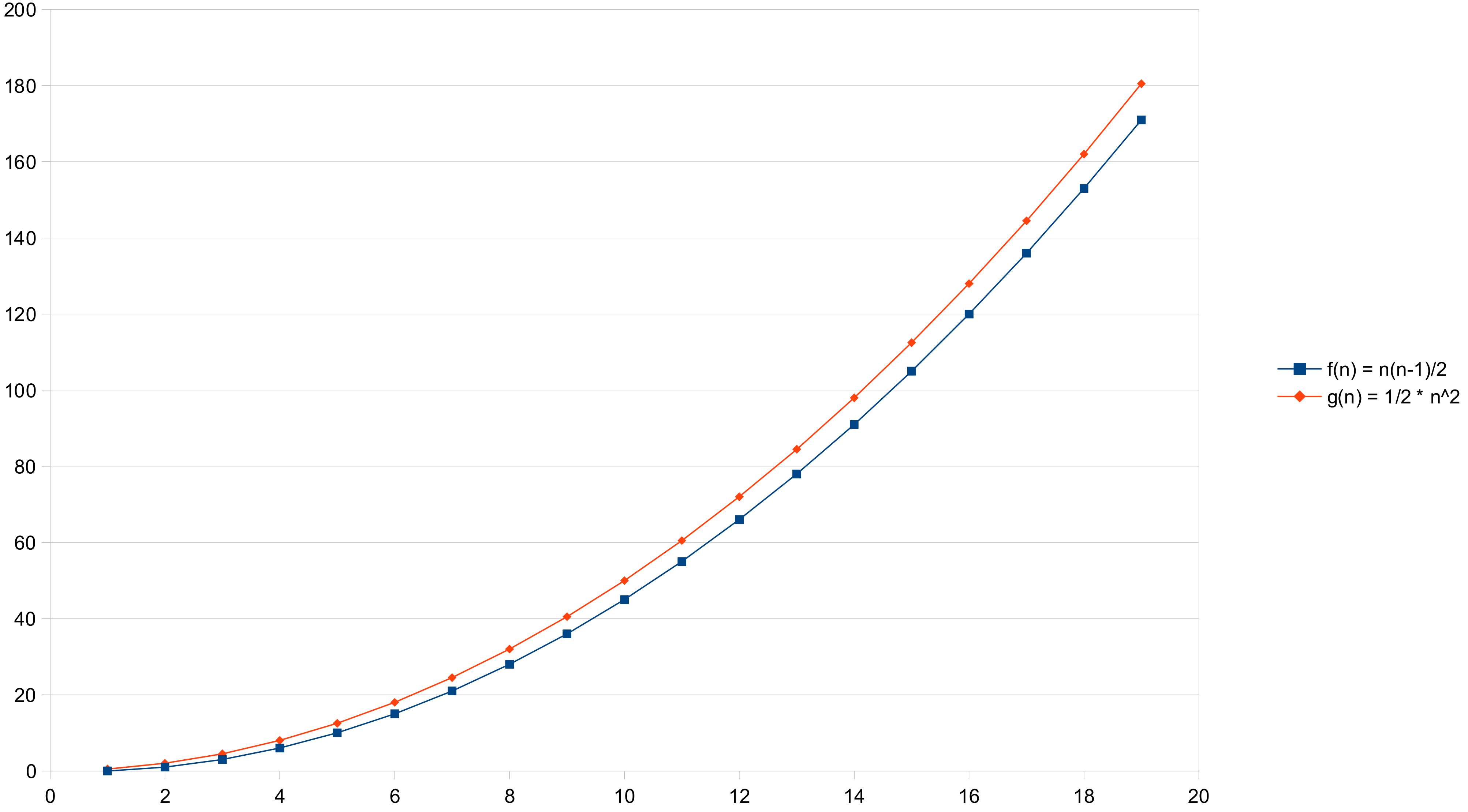
Not so obvious

- What might not be so obvious is that:

  f(n) <= (1/2)n^2, for n >= 1 and therefore, n0 = 1, g(n) = n^2, and c = 1/2.

- This implies that f(n) is O(n^2).

Graphically


200
180
160
140
120
100
80
60
40
20
0
0
2
4
6
8
10
12
14
16
18
20
f(n) = n(n-1)/2
g(n) = 1/2 * n^2