Code Time Complexity

Worrying about how long code/algos. take

- with respect to input since we cannot control
  - processors
  - read/write speed
  - 32 vs 64 vs 128 … bit machines
- these are the basics of determining complexity in code
- why do we even care about how efficient our code is?
  - can you imagine if Facebook took a minute to log in!!
    - mass riots
    - government overthrows
    - etc…
- finding clever ways to do the exact same thing but in shorter time will yield results, and happier clients
- how to time you're algorithms?
  - start clock when launched
  - stop clock when launched
  - compare the algorithm times
  - what's wrong with this??
- the only reliable way is to do it mathematically using Big-O notation
- will be given code, need to determine the run time
  - not all are created equal!

Analyzing Running Time Complexity of Algorithms

- the formation of loops in your code can make things take longer
  - might be necessary
  - but try to avoid
- loop time based on n (number of elements)
  - single loop
    - O(n)
  - double nested loop
    - $O(n^2)$
  - both of these can be adjusted if they don't through the entire loop

The eye ball test just to get the point across	
Summation Algorithm 1	**Summation Algo. 2**

```
int sum1(int N)
{
    int s = 0;
    for(int i = 1; i <= N; i++)
    {
        s = s + i;
    }

    return s;
}
```

```
int sum2(int N)
{
    int s = 0;
    s = N * (N + 1)/2;
    return s;
}
```

What are the functions doing? Which function will be quicker? Why?
How long will each take?? (Possible answers are O(1), O(n), O(n log n), $O(n^2)$ …

Time complexity of code, Estimating Big Oh

- will be looking at code to determine Big Oh
  - each line will have a value
  - loops are harder to look for running time
  - add all up together
- the rules below help you calculate the run time of code
- Final answer
  - will most likely have an "n"/"N"
  - add up each line in the code
    - after a while, you'll just start ignoring constants
  - pick highest order number
  - **will be from O(constant) , O(N), … $O(N^2)$, $O(N^3)$, …**

Code Analysis – Declarations

- don't count at all (not worth counting)
- **different than assignment!!!**

Declarations	
`int count;` `float average;`	~~`int count;`~~ `0` ~~`float average;`~~ `0`

C. A. - Assignments/Calls/Returns/Math

- are considered 1 "unit" per each occurrence
  - unit is some constant time
- **watch for the initialization portion of a loop**
  - which is frankly negligible and usually overtaken by a loop

Assignments	
`count = 1;`	~~`count = 1;`~~ `1`
`int counter = 1;` `// declaration + assignment`	~~`int counter = 1;`~~ `0 + 1` `// declaration + assignment`
`return counter;`	~~`return counter;`~~ `1`

Code Analysis – Comparison

- using any comparison operator (<, >, ==, etc…)
- 1 "unit" per each occurrence of the test PLUS the larger running times of the internal statements
- used **within** many other structures
  - if/else
  - loops
  - etc…

Code Analysis – For loops (Part 1)

- running time of a for loop is at MOST the run time of the statements INSIDE the loop (including tests) ***TIMES*** the number iterations
- variable "n" is used for calculations in loops
  - remember, your answer will have an "n" somewhere in it
- notice that uses, **declaration**, **assignment and comparison**
  - in the initialization portion of the loop

For loops Example #1

```
for(int i = 0; i < n; i++)
{
    cout << "looped" << endl;
}
```

```
for(int i = 0; i < n; i++)      1   n + 1   n
{                               = 1 + (n + 1) + n  =>  2n + 2
    cout << "looped" << endl;   1 = 1 * n
}
                                = (2n + 2) + (n)
                                = 3n + 2
                                ~ n (highest term)
```

1. Why is it n ***+ 1*** for the code portion "i < n"??
2. Why is the inner portion of the loop 1 * ***n***??
3. Try the loop below on your own.

For Loops Exercise

```
for(int i = 0; i < n; i++)
{
    count += i * i * i;
    // count = count + (i * i * i);
}
```

Answer$_b$:

Code Analysis – For loops (Part 2)

- watch ***incrementation*** portion
  - there are applications where the incrementation will be other than i++
    - j += 1 will equal n for that loop (or -= too)
      - j = j + 1
    - j *= 2 will equal $\log_2 n$ for that loop
      - j != 0
      - j = j * 2
    - j /= 2 will equal $\log_2 n$ for that loop
      - j != 0
      - j = j / 2
      - usually wants to approach **integer** 0
      - j > 0; j \= 2

Why n vs. $\log_2 n$?

j	j += 1	j *= 2
0	1	1
1	2	2
2	3	4
3	4	8
4	5	
5	6	
6	7	
7	8	
8	9	
9		

Code Analysis – Nested loops

- Analyze these inner loop to outer loop
- running time determined by the **product** of all sizes of all the loops

For loops Example #1

```
for(int i = 0; i < n; i++)
{
     for int j = 0; j < n; j++ )
     {

          cout << "looped" << endl;

     }
}
```

```
1    n + 1  n
for(int i = 0; i < n; i++)
{
     1    n + 1  n
     for(int i = 0; i < n; i++)
     {

          cout << "looped" << endl; 1 = 1 * n * n

     }
}
```

= 1 + (n + 1) + n => 2n + 2

= 1 + (n + 1) + n => 2n + 2 * n <= Level 1

<= Level 2

$= (2n + 2) + ((2n + 2) * n) + n^2$

$= 2n + 2 + 2n^2 + 2n + n^2$

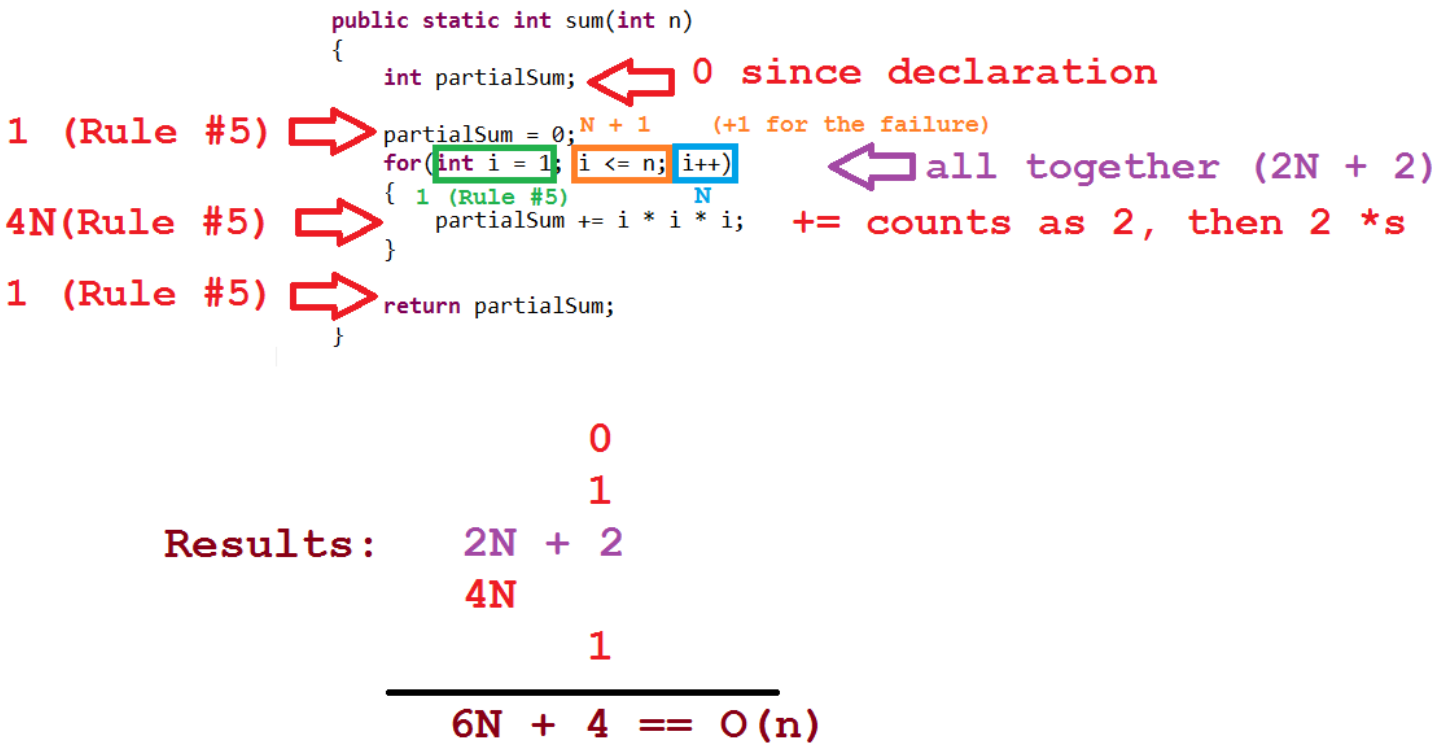
$= 3n^2 + 4n + 2$

$\sim n^2$ (highest term)

Code Analysis – Consecutive Statements

- just add all internal parts together, but the highest term counts
- usually the final step

Finalizing with Consecutive Statements



Results:

0
1
2N + 2
4N
1

6N + 4 == O(n)

O(n) = linear

Solve the exercises below. (13 in total, but skip last 2)

Code Snippet Example #1

```
int [] a = new int[10];

for(int i = 1; i < n; i++)
{
     a[i] = 0;
}


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

```
                                return k;
            1        ⇨  int [] a = new int[10];

  + N  +  2          ⇨  for(int i = 1; i < n; i++)
                        {
  + N                ⇨       a[i] = 0;
                        }


  + 2N + 2           ⇨  for(int i = 0; i < n; i++)
                        {
  + 2N + 2               ⇨  for(int j = 0; j < n; j++)
                            {
  * 4N                       ⇨  a[i] += a[j] + i + j;
                            }
                        }
```

Results: $4N^2 + 8N + 7$

$= 4N^2$

$= N^2$

Finding the Upper Bound – Code Snippets Exercises

#1

```
a = b;
++sum;
int y = Mystery( 42 );
```

#2

```
sum = 0;
for (i = 1; i <= n; i++)
     sum += n;
```

#3

```
sum1 = 0;
for (i = 1; i <= n; i++)
     for (j = 1; j <= n; j++)
          sum1++;
```

#4

```
sum1 = 0;
for (i = 1; i <= m; i++)
     for (j = 1; j <= n; j++)
          sum1++;
```

#5

```
int sum2 = 0;
for (int i = 1 ; i <= n; i++)
     for (int j = 1; j <= i; j++)
          sum2++;
```

#6

```
sum = 0;
for (j = 1; j <= n; j++)
     for (i = 1; i <= j; i++)
          sum++;
for (k = 0; k < n; k++)
     a[ k ] = k;
```

#7

```
sum1 = 0;
for (k = 1; k <= n; k *= 2)
     for (j = 1; j <= n; j++)
          sum1++;
```

#8

```
for( int i = n; i > 0; i /= 2 ) {
     for( int j = 1; j < n; j *= 2 ) {
          for( int k = 0; k < n; k += 2 ) {
               ... // constant number of operations
          }
     }
}
```

#9

```
for (int i=1; i < n; i *= 2 ) {
     for (int j = n; j > 0; j /= 2 ) {
          for (int k = j; k < n; k += 2 ) {
               sum += (i + j * k );
          }
     }
}
```

#10

```
for( int i = n; i > 0; i-- ) {
     for( int j = 1; j < n; j *= 2 ) {
          for( int k = 0; k < j; k++ ) {
               ... // constant number C of operations
          }
     }
}
```

#11

```
for( int bound = 1; bound <= n; bound *= 2 ) {
      for( int i = 0; i < bound; i++ ) {
            for( int j = 0; j < n; j += 2 ) {
                  ... // constant number of operations
            }
            for( int j = 1; j < n; j *= 2 ) {
                  ... // constant number of operations
            }
      }
}
```

Finding Upper Bound in Entire Function Ex #12

```
1       /**
2        * Cubic maximum contiguous subsequence sum algorithm.
3        */
4       public static int maxSubSum1( int [ ] a )
5       {
6           int maxSum = 0;
7
8           for( int i = 0; i < a.length; i++ )
9               for( int j = i; j < a.length; j++ )
10              {
11                  int thisSum = 0;
12
13                  for( int k = i; k <= j; k++ )
14                      thisSum += a[ k ];
15
16                  if( thisSum > maxSum )
17                      maxSum = thisSum;
18              }
19
20          return maxSum;
21      }
```

Figure 2.5 Algorithm 1

Finding Upper Bound in Entire Function Ex #13

```
1       /**
2        * Quadratic maximum contiguous subsequence sum algorithm.
3        */
4       public static int maxSubSum2( int [ ] a )
5       {
6           int maxSum = 0;
7
8           for( int i = 0; i < a.length; i++ )
9           {
10              int thisSum = 0;
11              for( int j = i; j < a.length; j++ )
12              {
13                  thisSum += a[ j ];
14
15                  if( thisSum > maxSum )
16                      maxSum = thisSum;
17              }
18          }
19
20          return maxSum;
21      }
```

Figure 2.6 Algorithm 2

Real life Application in Code Complexity

- Thanks to Will Morgan, F'14
- below are 2 pieces of code that complete the same procedure
  - using File IO
- one is efficient, other is not
- determine the runtimes for each

File IO Code Complexity Applications

Version A

```
// declare vars
ifstream infile; string line; string tempString; int endStringIndex; double lineNum = 0;

infile.open(filename.c_str(), ios_base::in); // open file

if (infile.fail())// tripple check that file opens
{
	cout << "Index Words file failed to open" << endl;
	exit(1);
}

while (getline(infile, line)) // read data in from file
{
	lineNum++;	// iterate the line number

	// handling splitting line by spaces
	istringstream buf(line);
	istream_iterator<string> beg(buf), end;
	vector<string> tokens(beg, end);

	// remove all punctuation except for ' and -
	for (unsigned int i = 0; i < tokens.size(); i++)
	{
		tempString = tokens.at(i);

		for (int j = 0; j < TOTALPUNC; ++j) // strip punctuation
		{
			tempString.erase(remove(tempString.begin(), tempString.end(), PUNC[j]),
tempString.end());
		}

		// make sure that the string has length and isnt null after stripping
		if (tempString.length() != 0)
		{
			// stupid c++98 doesnt have a back() method for strings
			endStringIndex = tempString.length() - 1;
			// strip leading/ending - and '
			while ((tempString.length() > 1) && (tempString[0] == '-' || tempString[0] == '\''
|| tempString[endStringIndex] == '-' || tempString[endStringIndex] == '\''))
			{
				endStringIndex = tempString.length() - 1;

				if (tempString[0] == '-' || tempString[0] == '\'')
				{ tempString.erase(0, 1); }
				if (tempString[endStringIndex] == '-' || tempString[endStringIndex] ==
'\'')
				{ tempString.erase(endStringIndex, - 1); }
			}

			// make each char lowercase
			transform(tempString.begin(), tempString.end(),tempString.begin(), ::tolower);

			// make tempString is not a punc that should have been stripped
			// only happens for an orig string of allowed punc
			if (tempString[0] != '-' || tempString[endStringIndex] != '\'')
			{
				Word temp = Word(tempString, lineNum); // make temp Word object
```

```
                        if (!m_filteredBST.contains(temp)) // insert into tree
                        { m_indexedBST.insert(temp); }
                }
        }
}


}
```

Version B

```
while (getline(infile, line))
{
        lineNum++;    // iterate the line number

        for (unsigned int i = 0; i < line.length(); i++)       // iterates thru each char in the line
        {
                // find punctuation or is char a letter
                if (isalpha(line[i]) || (tempString.length() > 0 &&
                        (line[i] == '-' || line[i] == '\'')))
                {
                        // no punctuation to save or special case
                        if(!hasApos || line[i] != '\'' || line[i] != '-')
                        { tempString.push_back(tolower(line[i])); }
                        if (line[i] == '\'' || line[i] == '-') // punctuation to save
                        { hasApos = true; }
                }
                else if(tempString.length() > 0)         // reached end of word, push onto BST
                {
                        // truncate back end of word to get rid of newlines, spaces, etc
                        while(tempString.length() > 0 &&
                                !isalpha(tempString[tempString.length()-1]))
                        { tempString.resize(tempString.length()-1); }

                        // do stuff with your string
                        tempString.clear();  // reset value of tempString to null
                        hasApos = false;     // new word, no punc to save
                }
        }

        if(tempString.length() > 0)       // end of if statement, end of line, end of word
        {
                // truncate back end of word to get rid of newlines, spaces, etc
                while(tempString.length() > 0 &&
                        !isalpha(tempString[tempString.length()-1]))
                { tempString.resize(tempString.length()-1); }

                // do stuff with your string

                tempString.clear();         // reset value of tempString to null
                hasApos = false;            // new word, no punc to save
        }
}
```

Answers

For Loop Coding Complexity Problem

```
for(int i = 0; i < n; i++)
{
    count += i * i * i;
    // count = count * i * i * i;
}
```

1 (int i = 0) · n + 1 (i < n) · n (i++)

= 1 + (n + 1) + n => 2n + 2

= 4 * n

= (2n + 2) + 4n

= 6n + 2

~ n

Finding the Upper Bound – Code Snippets Exercises

#1	`a = b;`
	`++sum;`
	`int y = Mystery( 42 );`
		constant = O(1)

#2	`sum = 0;`
	`for (i = 1; i <= n; i++)`
	`  sum += n;`
		linear = O(n)

#3	`sum1 = 0;`
	`for (i = 1; i <= n; i++)`
	`  for (j = 1; j <= n; j++)`
	`  sum1++;`
		$O(n^2)$

#4	`sum1 = 0;`
	`for (i = 1; i <= m; i++)`
	`  for (j = 1; j <= n; j++)`
	`  sum1++;`
		$O(n^2)$

#5	sum2 = 0;
	for (i = 1 ; i <= n; i++)
	  for (j = 1; j <= i; j++)
	  sum2++;
		$O(n^2)$

#6	sum = 0;
	for (j = 1; j <= n; j++)
	  for (i = 1; i <= j; i++)
	  sum++;
	for (k = 0; k < n; k++)
	  a[k] = k;
		$O(n^2)$

#7	sum1 = 0;
	for (k = 1; k <= n; k *= 2)
	  for (j = 1; j <= n; j++)
	  sum1++;
		O(n log n)

#8	for(int i = n; i > 0; i /= 2) {
	  for(int j = 1; j < n; j *= 2) {
	  for(int k = 0; k < n; k += 2) {
	  ... // constant number of operations
	  }
	  }
	}
		In the outer for-loop, the variable i keeps halving so it goes round $\log_2$ n times. For each i, next loop goes round also $\log_2$ n times, because of doubling the variable j. The innermost loop by k goes round n/2 times. Loops are nested, so the bounds may be multiplied to give that the algorithm is
		$O(n (\log n)^2)$
		.

#9	for (int i=1; i < n; i *= 2) {
	for (int j = n; j > 0; j /= 2) {
	for (int k = j; k < n; k += 2) {
	sum += (i + j * k);
	}
	}
	}
		Running time of the inner, middle, and outer loop is proportional to n, log n, and log n, respectively. Thus the overall Big-O complexity is
		$O(n(\log n)^2)$.

#10	`for( int i = n; i > 0; i-- ) {`
	`for( int j = 1; j < n; j *= 2 ) {`
	`for( int k = 0; k < j; k++ ) {`
	`... // constant number C of operations`
	`}`
	`}`
	`}`
	
	This is a tricky one.
		The outer for-loop goes round n times. For each i, the next loop goes round $m = \log_2 n$ times, because of doubling the variable j. For each j, the innermost loop by k goes round j times, so that the two inner loops together go round $1 + 2 + 4 + \ldots + 2^{m-1} = 2^m - 1 \approx n$ times. Loops are nested, so the bounds may be multiplied to give that the algorithm is $O(n^2)$
		
		.

#11	`for( int bound = 1; bound <= n; bound *= 2 ) {`
	`for( int i = 0; i < bound; i++ ) {`
	`for( int j = 0; j < n; j += 2 ) {`
	`... // constant number of operations`
	`}`
	`for( int j = 1; j < n; j *= 2 ) {`
	`... // constant number of operations`
	`}`
	`}`
	`}`
		The first and second successive innermost loops have O(n) and O(log n) complexity, respectively. Thus, the overall complexity of the innermost part is O(n). The outermost and middle loops have complexity O(log n) and O(n), so a straightforward (and valid) solution is that the overall complexity is $O(n^2 \log n)$.

Finding the Upper Bound #12

```
1       /**
2        * Cubic maximum contiguous subsequence sum algorithm.
3        */
4       public static int maxSubSum1( int [ ] a )
5       {
6           int maxSum = 0;
7
8           for( int i = 0; i < a.length; i++ )
9               for( int j = i; j < a.length; j++ )
10              {
11                  int thisSum = 0;
12
13                  for( int k = i; k <= j; k++ )
14                      thisSum += a[ k ];
15
16                  if( thisSum > maxSum )
17                      maxSum = thisSum;
18              }
19
20          return maxSum;
21      }
```

Figure 2.5 Algorithm 1

$O(n^3)$

Finding the Upper Bound #13

```
1       /**
2        * Quadratic maximum contiguous subsequence sum algorithm.
3        */
4       public static int maxSubSum2( int [ ] a )
5       {
6           int maxSum = 0;
7
8           for( int i = 0; i < a.length; i++ )
9           {
10              int thisSum = 0;
11              for( int j = i; j < a.length; j++ )
12              {
13                  thisSum += a[ j ];
14
15                  if( thisSum > maxSum )
16                      maxSum = thisSum;
17              }
18          }
19
20          return maxSum;
21      }
```

Figure 2.6 Algorithm 2

$O(n^3)$

File IO Code Complexity Application

Version A (inefficient) $O(n^2)$

```
// read in file by line
   // for loop, read thru line char by char
      // if statement, is it a letter or is tempstring > 0 AND is it an allowed punctuation
         // if no allowed punc seen, and not an allowed punc (de morgans law, i think?)
            // append to a temp string
         // if statement, is it an allowed punc
            // set boolean to true (tracks allowed punc)
      // else if tempstring > 0
         // truncate punc fromcof string
         // you have a parsed string, yay! do something with it
      // reset tempstring to be empty
      // reset bool for tracking punctuation to false
```

```
    // end of for loop, end of line, end of word
      // truncate punc from back of string
      // you have a parsed string, yay! do something with it
      // reset tempstring to be empty
      // reset bool for tracking punctuation to false
```

Version B (efficient) O(n)

```
// read in file by line
   // split line by spaces into a vector of tokens
     // for loop to go thru the vector
       // for loop to go thru each token
         // for loop for index of each token
            // if index is a letter
               // push onto tempstring
            // else if index is a number OR index is a punc AND is not allowed punc
              // use string erase(std.remove()) w/ parameters to take out non-allowed char
         // do something with parsed string
     // clear the vector
```

Sources

http://www.youtube.com/watch?v=8syQKTdgdzc
https://www.cs.drexel.edu/~kschmidt/Lectures/Complexity/big-o.pdf

Online Graphing Calculator
https://www.desmos.com/calculator

Reading from Files efficiently
http://www.cplusplus.com/forum/beginner/87238/
http://stackoverflow.com/questions/20326356/how-to-remove-all-the-occurrences-of-a-char-in-c-string
http://www.cplusplus.com/forum/beginner/115247/