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(n2)
  + 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(n2) …

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(N2), O(N3), …**

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

Code Analysis – Comparison

* using any comparison operator
* 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;  } |
|  |

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 Example #2 |
| for(int i = 0; i < n; i++)  {  count += i \* i \* i;  // count = count \* i \* i \* i;  } |
|  |

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 log2n for that loop
      * j != 0
      * j = j \* 2
    - j /= 2 will equal log2n for that loop
      * j != 0
      * j = j / 2
* Rule #2 – nested loops
  + Analyze these inside out
  + running time determined by the product of all sizes of all the loops
* Rule #3 – Consecutive Statements
  + just add all internal parts together, but the highest term counts

|  |
| --- |
| Rule #6 – Declarations (0 unit) |
|  |

|  |
| --- |
| Rule #5 – Assignments/Calls/Returns (1 Unit) |
|  |

|  |
| --- |
| Rule #4 – Comparison (1 unit) |
|  |

|  |
| --- |
| Rule #1 – For Loop |
| Notice n+1 since we need one more iteration just to fail and get out of the loop  Also “n” since we are incrementing each time by 1. If this is ANY different, the operational count will be different. |

|  |
| --- |
| Finalizing (Rule #3) |
| O(n) = linear |

|  |
| --- |
| Rule #2 – Nested Loops |
| **public** **static** **int** nothing1(**int** n)  {  **int** k;    k = 0;    **for**(**int** i = 1; i <= n; i++)  {  **for**(**int** j = 0; j <= n; j++)  {  k++;  }  }    **return** k;  } |
|  |

|  |
| --- |
| Code Snippet Example #3 |
| **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;  }  } |
|  |

|  |  |
| --- | --- |
| 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 | sum2 = 0;  **for** (i = 1 ; i <= n; i++)  **for** (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 #1 |
|  |

|  |
| --- |
| Finding Upper Bound in Entire Function Ex #1 |
|  |

Answers

|  |
| --- |
| Finding the Upper Bound #1 |
| O(n3) |

|  |
| --- |
| Finding the Upper Bound #2 |
| O(n3) |

|  |  |  |
| --- | --- | --- |
| 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(n2) |
| #4 | sum1 = 0;  **for** (i = 1; i <= m; i++)  **for** (j = 1; j <= n; j++)  sum1++; | O(n2) |
| #5 | sum2 = 0;  **for** (i = 1 ; i <= n; i++)  **for** (j = 1; j <= i; j++)  sum2++; | O(n2) |
| #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(n2) |
| #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 log2 n times. For each i, next loop goes round also log2 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  }  }  } | The outer for-loop goes round n times. For each i, the next loop goes round m = log2 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 + . . . + 2m−1 = 2m − 1 ≈ n times. Loops are nested, so the bounds may be multiplied to give that the algorithm is  O(n2)  . |
| #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(n2 log n). |

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>