Time Complexity Notes (USACO Bronze Prep)

12/6/2021(Monday) - 12/8/2021(Wednesday)

Time complexity

* The Big O Notation can be used to tell you the **largest possible** amount of times a function can run.

O(1)

* As seen in the example below, the time complexity is actually O(4) because there are 4 operations, however, since 4 is a constant number it can be simplified to O(1) ~~however, since it is a constant number of operations~~, the time complexity becomes O(1).

int a = 5;

int b = 7;

int c = 4;

int d = a + b + c + 153;

* Code such as 1+2+3 is considered O(1) meaning the most possible amount of times that this function can run is one.

O(N)

* The time complexity of this for loop is O(n) because the loop has N operations. This rule applies for while loops and for loops.

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

// constant time code here

}

* If there is a loop, such as “for(int i = 0; i <= n; i++)” then the most amount of times possible would be O(N)
* The time complexity of this loop is actually O(5n + 17) because the for loop is said to run that many times. However, since 5 and 17 are constant factors and lower order terms, those numbers are not counted. Thus, the time complexity is O(n).

for (int i = 1; i <= 5 \* n + 17; i++) {

// constant time code here

}

O(NM)

* If there is a loop within a loop, those variables get multiplied. If there is a loop, such as “for(int i = 0; i <= n; i++)” and inside is another for loop, “for(int x = 0; x <= m; x++)” makes the largest possible times that this function can run, O(NM)
* However, if there are two separate for loops, these for loops are not inside each other, then the one with the bigger possible amount of times is put inside the Big O Notation

O(N^2)

* If the for loop ends at the same variable so the for loop is “(int i = 0; i <= n; i++)” and inside that for loop is  “for(int x = 0; x <= n; x++)” then the largest amount of times the function will be run is O(n^2)\
* If there are for-loops that are not part of each other, the larger time complexity between them is the worst-case time complexity.

* The time complexity for these for-loops is O(n^2+m) because these variables are both not low order functions.
  + Why doesn’t the example above work then?
    - The example above doesn’t work because N is the same number. For all we know, N could be 10 while maybe M could be 1000.

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

for (int j = 1; j <= n; j++) {

// constant time code here

}

}

for (int i = i; j <= m; i++) {

// more constant time code here

}

* If N is being added such as “for(int x = 0; x <= n + 20; x++)” N will still be the largest possible number of times a function can run. O(N)

* *Mathematical formulas that just calculate an answer: O(1)*
* Binary search: O(log⁡n)
* Ordered set/map or priority queue: O(log⁡n) per operation
* Prime factorization of an integer, or checking primality or compositeness of an integer naively: O( sqrt(n) )
* *Reading in N items of input: O(n)*
* *Iterating through an array or a list of n elements: O(n)*
* Sorting: usually O(nlog⁡n) for default sorting algorithms (mergesort, Collections.sort, Arrays.sort)
* Java Quicksort Arrays.sort function on primitives: O(n2)
  + See Introduction to Data Structures for details.
* Iterating through all subsets of size K of the input elements:O(n^k). For example, iterating through all triplets is O(n^3)
* Iterating through all subsets: O(2^n)
* Iterating through all permutations:O(n1)

Constant Factor

* Constant factors take very slightly different times to run depending on how many operations there are. O(1) will be faster than O(3). Constant factors however are mainly ignored in the Big-O notation, but maybe of use to speed things up later.