Data Structures and Algorithms
CS-206

Asymptotic Notations

Instructor
Dr. Maria Anjum
Assistant Professor

Department of Computer Science
Lahore College for Women University

- Asymptotic analysis refers to computing the running time of any operation in mathematical units of computation.

- When we study algorithms, we are interested in characterizing them according to their efficiency.
- We are usually interesting in the order of growth of the running time of an algorithm, not in the exact running time. This is also referred to as the asymptotic running time.
- We need to develop a way to talk about rate of growth of functions so that we can compare algorithms.
- Asymptotic notation gives us a method for classifying functions according to their rate of growth.

Every line code or algorithm has performance or cost associated with it. We are interested in two such cost here, specifically

- Time - also called as Time Complexity : How much time does this code take to execute ?
- Memory - also called as Space Complexity : How much memory does this program require to execute ?

Three Cases

- **Best Case** – Minimum time required for program execution.
- **Average Case** – Average time required for program execution.
- **Worst Case** – Maximum time required for program execution.

Asymptotic Notations

- Following are the commonly used asymptotic notations to calculate the running time complexity of an algorithm.

- O Notation → Big Oh
- Ω Notation → Omega
- θ Notation → Theta

Big Oh Notation, O

- The notation O(n) is the formal way to express the upper bound of an algorithm's running time. It measures the worst case time complexity or the longest amount of time an algorithm can possibly take to complete.

Omega Notation, Ω

- The notation Ω(n) is the formal way to express the lower bound of an algorithm's running time. It measures the best case time complexity or the best amount of time an algorithm can possibly take to complete.

Theta Notation, θ

- The notation θ(n) is the formal way to express both the lower bound and the upper bound of an algorithm's running time.

Common Asymptotic Notations

constant	–	O(1)
logarithmic	–	O(log n)
linear	–	O(n)
n log n	–	O(n log n)
quadratic	–	$O(n^2)$
cubic	–	$O(n^3)$
polynomial	–	$n^{O(1)}$
exponential	–	$2^{O(n)}$

Constant Time

- If we want to access 3rd element in the array, we could do as
- int thirdElement = a[2]; // Note arrays are indexed from 0 not 1;

Now 5th Element

- int fifthElement = a[4];
- So what is the cost associated with accessing an element in an array ? Its a constant time - This time remains as a constant on a given computer irrespective which element we access in the array.
- Such constant time operations are represented in Big-O as
- O(1) - Constant time
- Where (1) is one unit of work required, The number of elements in the array does not matter to us in this case, irrespective of length of the array the time to access any element in an array remains a constant which is represented as O(1).

Linear Time (cont.)

- int myArray[] = { 10, 6, 5, 4, 11, 7 };
- Now lets say we need to print all the elements in the array, our code would look like
- for(int i = 0; i<myArray.length; i++) {
- System.out.println(myArray[i]);
- }

- Since we have 'n' elements in the array and it takes O(1) to access each element we have
- **O(n) - Linear time**

Linear Time

* Now coming back to the definition where we said, Big-O is expressed as a function of the length of its input.

In this example we can see that

- 'n' is the number of input elements
- The time complexity of this code is O(n) -

So Big-O is expressed as a function of number of elements of the input array. This means when 'n' increases, the time required to complete the above operation increases linearlywith respect to 'n' (input).

A single iteration (loop) over all the elements in the array gives us a complexity of O(n). If there are NO nested loops we can probably guess the complexity of the code we looking at would be in the O(n).

Quadratic Time (Cont.)

Lets consider the same above sample array:
int myArray[] = { 10, 6, 5, 4, 11, 7};

Now lets say we need to sort all the elements in the array. In this case we take each element in the array and compare it with every other element in the array

```
// Sample bubble sort code.
for(int i=0; i < myArray.length; i++){

   for(int j=1; j < (myArray.length - i); j++){

        if(myArray[j-1] > myArray[j]){

          //Swap
          int temp = myArray[j-1];
          myArray[j-1] = myArray[j];
          myArray[j] = temp;
      }
   }
}
```

Quadratic Time

- **So what is the complexity in the above case ?**

  Since we have 'n' elements in the array and compare each element with every other element in the array, it is

  $O(n^2)$

- There are nested loops in the above code/operation hence we can **probably** guess the complexity of the code we looking at could be in the $O(n^2)$.
-

  A simple rule of thumb is complexity is equal to $O(\mathbf{n}^{\mathbf{x}})$ where **x** is the number of levels of loop nesting. In the above operation x = 2.

Logarithmic Time (Cont.)

- This one would be a little tricky to understand if you are new to asymptotic notations or looking at binary search for the first time.
- Lets consider this sample SORTED array :
- int myArray[] = { 1, 4, 5, 6, 7, 10, 11, 25, 36 };
- Problem :
- Given an element 'e' , We need to write code to see if the element is present in the array.
- Eg : see if element 11 is present in the above sorted array. (e = 11 in this case)

Logarithmic Time (Cont.)

- **Binary search algorithm :**

  Binary search algorithm takes advantage of the fact that the array is sorted, rather than checking each element in the array from 0 to array length with the value 'e', binary search does the below
- Take the middle element of the array
- Compare with element 'e'
- If middle element is equal to 'e' - We found the element
- else if 'e' is less than middle element, The element we are looking for is in the lower half of the array - So search lower half
- else 'e' is greater than middle element, The element we are looking for is in the upper half of the array - So search upper half

Logarithmic Time (Cont.)

- Here is the code for binary search:

```
int binarySearch(int[] array, int value, int left, int right) {

    if (left > right) {
        return -1;
    }
```

-

-

```
    int middle = (left + right) / 2;

    if (array[middle] == value) {
      return middle;

    } else if (array[middle] > value) {
      return binarySearch(array, value, left, middle - 1);

    } else {
      return binarySearch(array, value, middle + 1, right);
    }
}
```

So what is the complexity in the above case ?

Logarithmic Time (Cont.)

- Since we have 'n' elements in the array and for each wrong guess, we discard half of the current array, so the MAXIMUM number of iterations required to find an element is

  O(lg n)

  Where lg = log to the base 2.

Summary

- Big-O notation is used to represent the cost associated with the code specifically **time** and **memory**.
- Its an abstract mathematical representation which allows us to compare cost of the code irrespective of the type of machine/speed/memory.
- We can compare performance of two different algorithms by just looking at the Big-O functions of these algorithms and choose which one is better for our problem in-hand.
- Look at the levels of nesting loops in your code it helps to guess the complexity.
- Space-time trade-off is one of the important constraints in choosing an algorithm.

Must be Consulted

- https://www.youtube.com/watch?v=6TsmsGn-N0E
- https://en.wikipedia.org/wiki/Time_complexity
- http://www.codenlearn.com/2011/07/understanding-algorithm-complexity.html
- https://betterexplained.com/articles/an-intuitive-guide-to-exponential-functions-e/
- https://www.youtube.com/watch?v=iOq5kSKqeR4
- https://www.youtube.com/watch?v=iOq5kSKqeR4