BRAC University

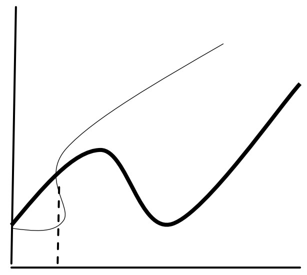
Algorithms

Lecture 1.2

Rubayat Khan
9/9/2015

Big Omega (Ω):

We have learnt the longest time taken by an algorithm to produce its output. What about the quickest time? The quickest time or the best case complexity is known as the lower bound and is denoted by Ω. The lower bound of a function f1(n), is any function f2(n) such that f2(n)<=f1(n) for all input n greater that n_o. n_o is the point where both the functions meet..



The thin curve is f1(n) and the thick one is f2(n). In the same way we have proved for the upper bound, we can prove it for the lower bound. f1(n) = Ω(f2(n)).

Let $f1(n) = 2n^2 -3$ and f2(n) = n. Can we say f1(n) = Ω(f2(n))?

$2n^2 -3 = n$
$2n^2 -3 - n = 0$
we get 2 values of n, 1.5 and -1. We discard -1. Put any value greater than 1.5. We will see that f1(n)>f2(n). Like the upper bound there could be multiple lower bounds too (if there exist such functions). If there is more than one lower bound we will pick the one which is nearest to the original function.
The reason for choosing the tightest upper bound and lower bound is to reduce error.

Usefulness of O and Ω
O gives the worst case andΩ gives the best case. We are generally interested in finding out the worst case and work on it to make it as less as possible. A good algorithm is the one that has the lowest worst time.

So far we have expressed the behavior of an algorithm as a function of input, f(n). F(n) is the curve we get when we feed the algorithm with both best and worst input. The curve g(n) >= f(n) is the curve we get in the worst case when all the input fed are the worst. Lastly the curve h(n) <= f(n) is the curve produced in the best case when all the input are the best.
Imagine you have developed an algorithm and you want to sell it to the company. Suppose that your algorithm works perfect. Which factor, the upper bound or the lower bound, should be the buyers concern?
In this lecture we will learn how to find out the function f(n) of an algorithm. What is f(n)? It is an equation in terms of n by which we can calculate the time taken for n number of input.

Algorithms can be categorized into two types. 1) iterative 2) Recursive
Iterative algorithms: Algorithms those do their jobs in one or more loops, while/for/do while.

```
methodA( ){
        for ( ){

        }
}
```

To find the time complexity (time taken) for an iterative program we need to count how many times a or more loops run to complete the program. What if there are no loops at all? Programs with no loops are independent to the size of the input, that is, that program will take **constant** time.

<u>Problems with singe loop(s):</u>

```
1. for (i =1 to n){
        print("hi");
  }
```

Consider the following program. How many times will the loop run? The loop will run n times. If n = 5, it will run times if n=100 then 100 times. Hence we can say the time complexity, the function f(n) = n. Let us now find the worst case and the best case of the algorithm. A theoretical curve can have multiple upper bounds and lower bounds but is it possible in reality? [Upper bound = worst case, lower bound = best case].

Consider the situation of you coming to university on a **maximum** traffic day and on a **minimum** traffic day. On a maximum day it takes 2 hrs and on a minimum day it takes 30 minutes. Now is it ever possible to take you more than 2hrs on the worst day and less than 30 minutes on the best day? Never! Time taken by an algorithm is real hence there we will be one worst case and one best case of each.

```
2. i =1
   while i < n
        print("hi");
        i = i+2
```

In this example i increases by 2 thus we are sure that "hi" will not be printed n times. The table below shows as the value of i increases from 1 to n, the word "hi" is printed k times.

i	1	3	5	7	....	n
Hi	1	2	3	4	....	k

Our goal is to find an equation in terms of n. Notice the pattern, i = 3, is equal to (2 + 1) "hi," i = 5 is equal to (2 +3) "hi", therefore i = n is equal to (k-1)+k.

$k-1+k = n$

$2k=n+1$

$k=n/2$

The best and the worst case is n because regardless of the type of input the loop will always run n times.