Asymptotic Notations

MD. MUKTAR HOSSAIN
LECTURER

DEPT. OF CSE
VARENDRA UNIVERSITY

Introduction

- Asymptotic Notations are mathematical tools used to analyze the performance of algorithms by understanding how their efficiency changes as the input size grows.
- There are mainly three asymptotic notations:
  - Big-O Notation (O-notation) ---- Worst Case
  - Omega Notation (Ω-notation) ---- Best Case
  - Theta Notation (Θ-notation) ---- Average Case

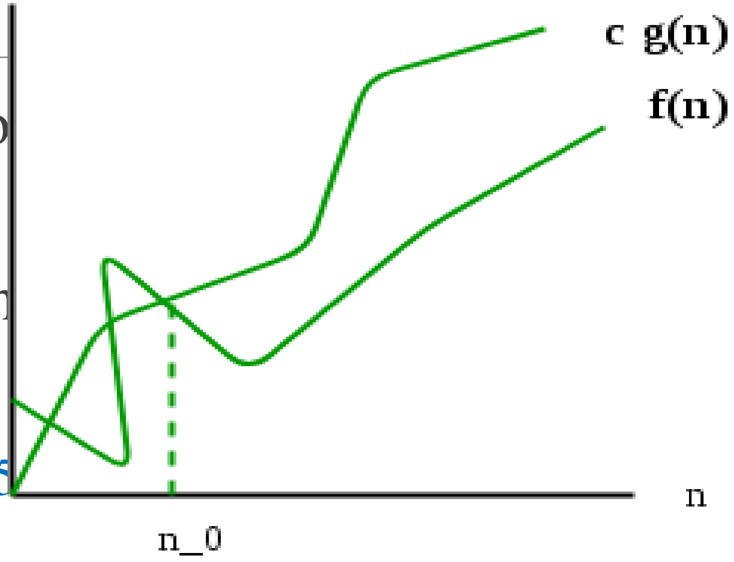
Big-O Notation (O-notation)

- The upper bound of the running time of an algo gives the worst-case complexity of an algorithm.
- $O(g(n)) = \{ f(n)$: there exist positive constants c an $f(n) \leq cg(n)$ for all $n \geq n0 \}$
- Example: Let us consider a given function, $f(n)=4n^3+10n^2+5$

  Considering $g(n)=n^3$,

  $f(n) \leqslant 5g(n)$, for all the values of $n>10$

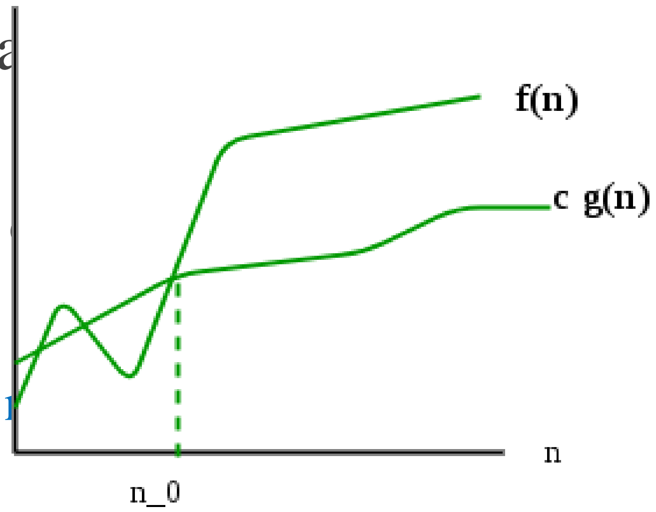
  Hence, the complexity of f(n) can be represented as $O(g(n))$, i.e. $O(n^3)$



f(n) = O(g(n))

Omega Notation (Ω-Notation)

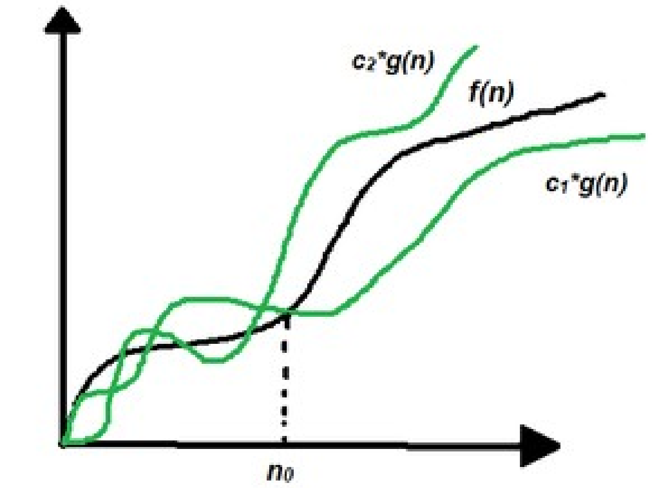
- The lower bound of the running time of an a provides the best case complexity of an algorithm.
- $\Omega(g(n))$ = { f(n): there exist positive constants c an cg(n) ≤ f(n) for all n ≥ n0 }
- Example: Let us consider a given function, $f(n)=4n^3+10n^2+5$
  Considering $g(n)=n^3$
  $f(n) \geqslant 4g(n)$, for all the values of $n \geqslant 0$
  Hence, the complexity of f(n) can be represented as $\Omega(g(n))$, i.e. $\Omega(n^3)$



f(n) = Omega(g(n))

Theta Notation (Θ-Notation)

- The upper and the lower bound of the running time of an algorithm, it is used for analyzing the average-case complexity of



- $\Theta(g(n)) = \{f(n)$: there exist positive constants c1, c2 $\leq c1 * g(n) \leq f(n) \leq c2 * g(n)$ for all $n \geq n0\}$
- Example: Let us consider a given function, $f(n)=4n^3+10n^2+5$

  Considering $g(n)=n^3$

  $4g(n)\leqslant f(n)\leqslant 5g(n)$, for all the large values of n.

  Hence, the complexity of f(n) can be represented as $\theta(g(n))$, i.e. $\theta(n^3)$

Insertion Sort Algorithm

```
Insertion-Sort(A)
for j := 2 to A.length do
   key := A[j]
   //insert A[j] into the sorted sequence A[1 ... j − 1]
   i := j − 1
   while i > 0 and A[i] > key do
      A[i + 1] := A[i]
      i := i − 1
   end
   A[i + 1] := key
end
```

Analysis of The Insertion Sort Algorithm

```
Insertion-Sort(A)
    for j := 2 to A.length do
        key := A[j]
        i := j − 1
        while i > 0 and A[i] > key do
            A[i + 1] := A[i]
            i := i − 1
        end
        A[i + 1] := key
    end
```

cost	times
c_7	$n-1$

Analysis of The Insertion Sort Algorithm (Best Case)

Now if we represent the running time of insertion sort as a function T of input size n, than the definition of T becomes,

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{j=2}^{n} t_j + c_5 \sum_{j=2}^{n} (t_j - 1) + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7(n-1).$$

If the input sequence is already sorted $t_j = 1$ for $j = 2, 3, \ldots, n$

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4(n-1) + c_7(n-1)$$

$$= (c_1 + c_2 + c_3 + c_4 + c_7)n - (c_2 + c_3 + c_4 + c_7)$$

Analysis of The Insertion Sort Algorithm (Worst Case)

If the array is reversed sorted $t_j = j$ for $j = 2, 3, \dots n$

We know $\sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1$

And $\sum_{j=2}^{n} (j-1) = \frac{n(n-1)}{2}$

Then $T(n)$ becomes,

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4\left(\frac{n(n+1)}{2} - 1\right) + c_5\left(\frac{n(n-1)}{2}\right) + c_6\left(\frac{n(n-1)}{2}\right) + c_7(n-1)$$

$$= \left(\frac{c_4}{2} + \frac{c_5}{2} + \frac{c_6}{2}\right) n^2 + \left(c_1 + c_2 + c_3 + \frac{c_4}{2} - \frac{c_5}{2} - \frac{c_6}{2} + c_7\right) n - (c_2 + c_3 + c_4 + c_7)$$

We notice that this is of the form $an^2 + bn + c$. Thus $T(n)$ is a quadratic function of the input size n.

Properties of Asymptotic Notations

- General properties
  - If f(n) is O(g(n)), then a*f(n) is O(g(n))
  - If f(n) is Ω(g(n)), then a*f(n) is Ω(g(n))
- Reflexive
  - If f(n) is given, then f(n) is O(f(n))
- Transitive
  - If f(n) is O(g(n)) and g(n) is O(h(n)), then f(n)=O(h(n))
  - $f(n)=n$, $g(n)=n^2$, and $h(n)=n^3$ then

    $n = O(n^2)$, $n^2 = O(n^3)$, $\rightarrow n = O(n^3)$
- Symmetric
  - If f(n) is Θ(g(n)), then g(n) is Θ(f(n))
  - Suppose $f(n)=n^2$; $g(n)=n^2$

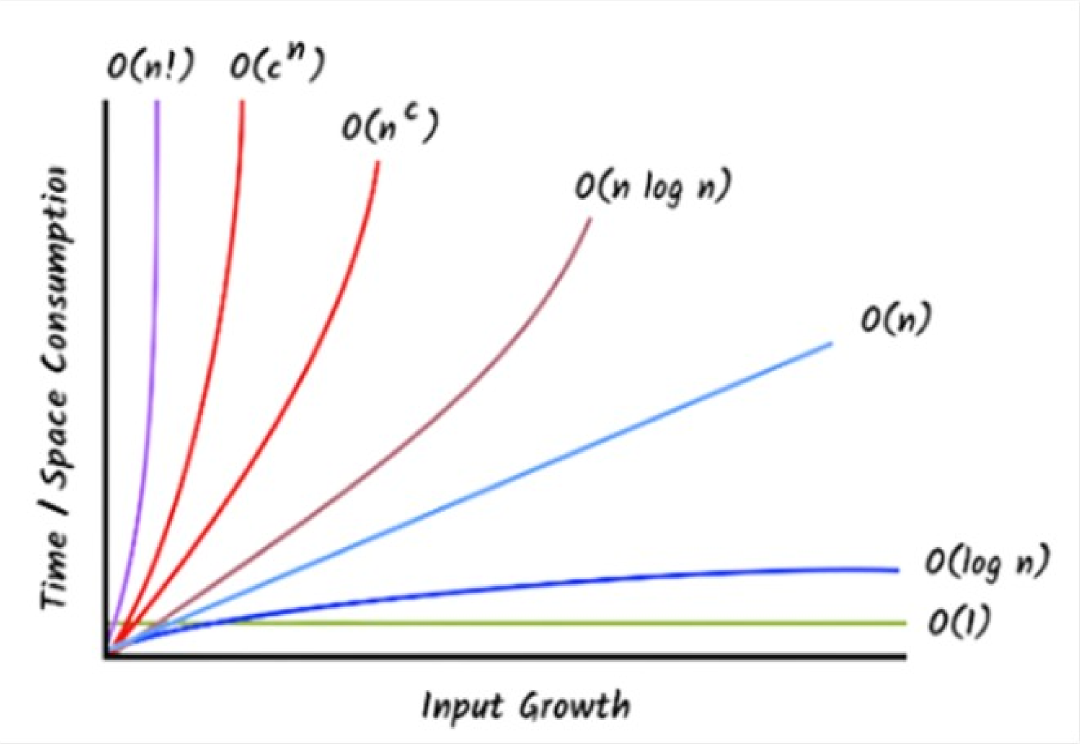
    $f(n)=\Theta(n^2)$

    $g(n)=\Theta(n^2)$
- Transpose symmetric
  - If f(n) = O(g(n)), then g(n) is Ω(f(n))
  - $f(n)=n$ and $g(n)=n^2$

    Then n is $O(n^2)$ and n^2 is $\Omega(n)$

Common Asymptotic Notations

Name	Notation
Constant	O(1)
Logarithmic	O(log n)
Linear	O(n)
Linearithmic, Log-linear	O(n log n)
Quadratic	$O(n^2)$
Polynomial	$O(n^c)$
Exponential	$O(c^n)$; where c>1
Factorial	O(n!)



Comparison of Complexities

- If $f1(n) = 3n \log n$, $f2(n) = n^2$, $f3(n) = 2^n$, then what will be the formation according to complexity in ascending order?
  - Answer: f1 < f2 < f3
- If $f1(n) = 3n \log n$, $f2(n) = n^2$, $f3(n) = 2^n$, $f4 = 2^{10}$, then what will be the formation according to complexity in descending order?
  - Answer: f3 > f2 > f1 > f4.
- If $f1 = O(n)$ and $f2(n) = O(\log n)$, respectively. Therefore, "*f2* always runs faster than *f1*" – is true or false?
  - Answer: Home Work.
- Which one is better between $f1(n) = 10^5$ and $f2(n) = 5^{10}$ in terms of complexity of algorithm.
  - Answer: Equal. O(1).

Time Complexity Analysis

<table>
<tr>
<td>

```
int a = 0, b = 0;
for (i = 0; i < N; i++) {
  a = a + rand();
}
for (j = 0; j < M; j++) {
  b = b + rand();
}
```

</td>
<td>

i. O(N * M) time, O(1) space
ii. O(N + M) time, O(N + M) space
iii. O(N + M) time, O(1) space
iv. O(N * M) time, O(N + M) space

</td>
</tr>
<tr>
<td>

```
int a = 0;
for (i = 0; i < N; i++) {
  for (j = N; j > i; j--) {
    a = a + i + j;
  }
}
```

</td>
<td>

i. O(N)
ii. O(N*log(N))
iii. O(N * Sqrt(N))
iv. O(N*N)

</td>
</tr>
</table>

Time Complexity Analysis Cont'd

`int i, j, k = 0;`
`for (i = n / 2; i <= n; i++)`
`{`
`    for (j = 2; j <= n; j = j * 2) {`
`        k = k + n / 2;`
`    }`
`}`
	i. O(n)
	ii. O(n log n)
	iii. O(n^2)
	iv. O(n^2Logn)

`int a = 0, i = N;`
`while (i > 0) {`
`    a += i;`
`    i /= 2;`
`}`
	i. O(N)
	ii. O(Sqrt(N))
	iii. O(N / 2)
	iv. O(log N)

Time Complexity Analysis Cont'd

```
for (int i = 1; i < n; i++) {
    i *= k;
}
```

i. $O(n)$

ii. $O(k)$

iii. $O(\log_k n)$

iv. $O(\log_n k)$

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

i. n

ii. $(n+1)$

iii. $n(n-1)/2$

iv. $n(n+1)$

Thank You