Unit 8 (Growth Function)

1. Asymptotic notations: θ , O, Ω , o,ω notations and their properties

Asymptotic Analysis of algorithms (Growth of function)

Resources for an algorithm are usually expressed as a function regarding input. Often this function is messy and complicated to work. To study Function growth efficiently, we reduce the function down to the important part.

Let f (n) = an^2+bn+c

In this function, the n^2 term dominates the function that is when n gets sufficiently large.

Dominate terms are what we are interested in reducing a function, in this; we ignore all constants and coefficient and look at the highest order term concerning n.

Asymptotic notation:

The word **Asymptotic** means approaching a value or curve arbitrarily closely (i.e., as some sort of limit is taken).

1<logn<√n<n<nlogn<n^2<n^3<…..2^n<3^n<…..<n^n

Asymptotic notations:

1. O(Big-Oh) – upper bound
2. Ω(Big-Omega)- lower bound
3. θ(theta) - average/tightly bound
4. o(small-oh)
5. ω(small-omega) notations

Asymptotic analysis

It is a technique of representing limiting behavior. The methodology has the applications across science. It can be used to analyze the performance of an algorithm for some large data set.

1. In computer science in the analysis of algorithms, considering the performance of algorithms when applied to very large input datasets

Compiled by: Er. Sandesh S. Poudel

The simplest example is a function $f(n) = n^2+3n$, the term 3n becomes insignificant compared to n^2 when n is very large. The function "$f(n)$ is said to be **asymptotically equivalent** to n^2 as $n \rightarrow \infty$", and here is written symbolically as $f(n) \sim n^2$.

Asymptotic notations are used to write fastest and slowest possible running time for an algorithm. These are also referred to as 'best case' and 'worst case' scenarios respectively.

"In asymptotic notations, we derive the complexity concerning the size of the input. (Example in terms of n)"

"These notations are important because without expanding the cost of running the algorithm, we can estimate the complexity of the algorithms."

Why is Asymptotic Notation Important?

1. They give simple characteristics of an algorithm's efficiency.

2. They allow the comparisons of the performances of various algorithms.

Compiled by: Er. Sandesh S. Poudel

Analysis of Algorithm

- Measures the efficiency of an algorithm by checking:
  - Correctness of an algorithm
  - Implementation "
  - Simplicity "
  - Execution time and memory requirements

Types of Analysis:

- Worst case running time
- Average "
- Best "

Worst-case:

- The behaviour of the algorithm with respect to worst possible case of the input instance.
- is an upper-bound on the running time for any input.

Average case :

- The expected behaviour when the iput is randomly drawn from a given distribution.
- is an estimate of the running time for an "average" input.

Best Case:

- When the input is already in order. eg, in sorting, if the elements are already sorted for a specific algorithm.
- rarely occurs in practice.

The choice of a particular algorithm depends on following performance analysis and measurements:

- Space complexity
- time "

Algorithm performance:

The performance evaluation of an algorithm is obtained by totalling the number of occurances of each operation when running the algorithm.

The performance of an algorithm is evaluated as a function of the input size "n" and is said to be considered modulo a multiplicative constant.

The following notations are commonly used in performance analysis and used to characterize the complexity of an algorithm:

- Θ Notation (Tightly Bound) - Theta
- O Notation (Upper Bound) - Big-Oh
- Ω - Notation (Lower Bound) - Big-omega

Chapter 8:

Growth Functions

Asymptotic Notations

Algorithm Efficiency

- How fast an algorithm is?
- How much memory does it cost?
- Computational complexity: measure of the difficulty degree (time or space) of an algorithm.

Let us discuss about various looping

Linear Loops

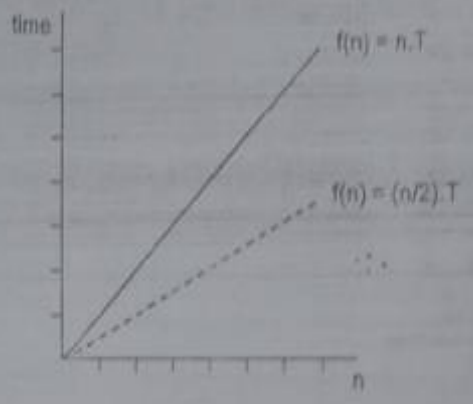
```
1 i = 1
2 loop (i <= 1000)
  1 application code
  2 i = i + 1
```

The number of times the body of the loop is replicated is 1000

```
1 i = 1
2 loop (i <= 1000)
  1 application code
  2 i = i + 2
```

The number of times the body of the loop is replicated is 500

LInear Loops



Logarithmic Loops

Multiply loops

```
1 i = 1
2 loop (i <= 1000)
  1 application code
  2 i = i * 2
```

The number of times the body of the loop is replicated is $\log_2 n$.

e.g. If n=32 then $\log_2 32=5$ (i=2,4,8,16 and 32)

Logarithmic Loops

Multiply loops

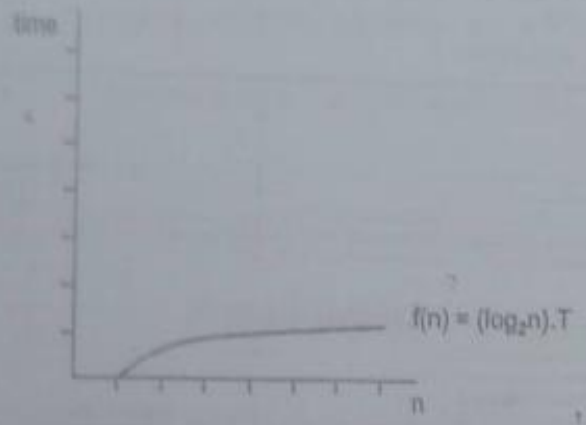
```
1 i = 1
2 loop (i <= 1000)
  1 application code
  2 i = i * 2
```

Divide loops

```
1 i = 1000
2 loop (i >= 1)
  1 application code
  2 i = i / 2
```

The number of times the body of the loop is replicated is $\log_2 n$

Logarithmic Loops



Nested Loops

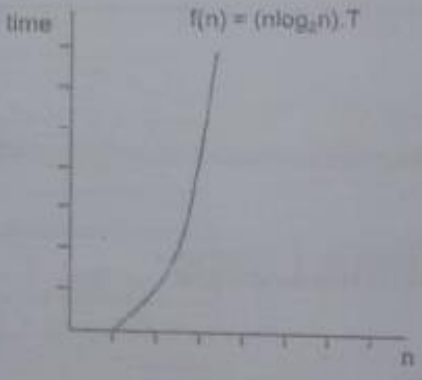
Iterations = Outer loop iterations * Inner loop iterations

Linear Logarithmic Loops

```
1 i = 1
2 loop (i <= 10)
    1 j = 1
    2 loop (j <= 10)
        1 application code
        2 j = j * 2
    3 i = i + 1
```

The number of times the body of the loop is replicated is $n\log_2 n$

Linear Logarithmic Loops



Quadratic Loops

```
1 i = 1
2 loop (i <= 10)
    1 j = 1
    2 loop (j <= 10)
        1 application code
        2 j = j + 1
    3 i = i + 1
```

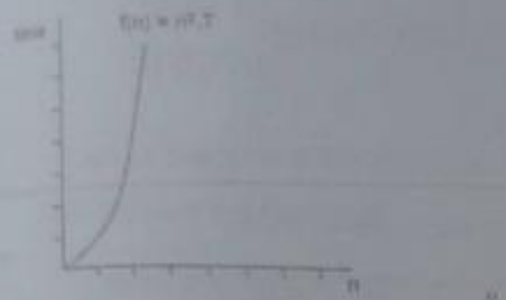
The number of times the body of the loop is replicated is n^2

Dependent Quadratic Loops

```
1 i = 1
2 loop (i <= 10)
    1 j = 1
    2 loop (j <= i)
        1 application code
        2 j = j + 1
    3 i = i + 1
```

The number of times the body of the loop is replicated is $1 + 2 + \ldots + n = n(n + 1)/2$

Quadratic Loops



Standard Measures of Efficiency

Standard Measures of Efficiency

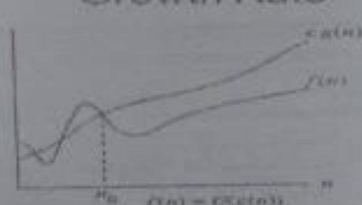
Efficiency	Big-O	Iterations	Est. Time
logarithmic	$O(\log_2 n)$	14	microseconds
linear	$O(n)$	10,000	.1 seconds
linear logarithmic	$O(n\log_2 n)$	140,000	2 seconds
quadratic	$O(n^2)$	$10{,}000^2$	15-20 min.
polynomial	$O(n^k)$	$10{,}000^k$	hours
exponential	$O(2^n)$	$2^{10{,}000}$	intractable
factorial	$O(n!)$	10,000!	intractable

Assume instruction speed of 1 microsecond and 10 instructions in loop.
n = 10,000

Asymtotic Complexity

- Algorithm efficiency is considered with only big problem sizes.
- We are not concerned with an exact measurement of an algorithm's efficiency.
- Terms that do not substabtially change the function's magnitude are eliminated.

Growth Rate



- The idea is to establish a relative order among functions for large n
- $\exists\, c, n_0 > 0$ such that $f(n) \le c\, g(n)$ when $n \ge n_0$
- f(n) grows no faster than g(n) for "large" n

Asymptotic notation: Big-Oh

- $f(n) = O(g(n))$:
  If there are positive constants c and n_0 such that

  $$f(n) \le c\, g(n) \text{ when } n \ge n_0$$

- The growth rate of f(n) is *less than or equal to* the growth rate of g(n)

Big-O Notation

- Set the coefficient of the term to one.
- Keep the largest term and discard the others. [In general]
- If $T_1(N) = O(f(N))$ and $T_2(N) = O(g(N))$, then
  - $T_1(N) + T_2(N) = \max(O(f(N)),\ O(g(N)))$,
  - $T_1(N) * T_2(N) = O(f(N) * g(N))$

Examples:

- $f(n) = c\,n \Rightarrow f(n) = O(n)$.
- $f(n) = n(n+1)/2 = n^2/2 + n/2 \Rightarrow f(n) = O(n^2)$.
- $f(n) = n^2/2 - 3n \Rightarrow O(n^2)$
- $f(n) = 1 - cn \Rightarrow O(n)$
- $f(n) = 7n^2 - 10n + 3 \Rightarrow O(n^2)$
- $f(n) = \log n + n \Rightarrow O(n)$

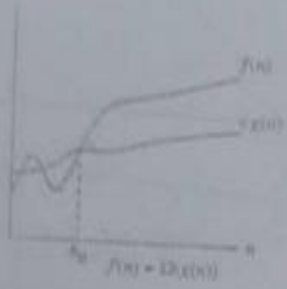
Examples

- Show that
  - $f(n) = n^2 + 2n + 1$ is $O(n^2)$

    When n>2, we have $2n \le n^2$ and $1 \le n^2$. Consequently, if n>2, we have $0 \le n^2 + 2n + 1 \le n^2 + n^2 + n^2 = 3n^2$, it follows that c=3 and $n_0 = 2$. So f(n) is $O(n^2)$.
  - $f(n) = 7n^2$ is $O(n^3)$

    When n>7, we have $7n^2 < n^3$. Consequently, we take c=1 and $n_0 = 7$ so that $7n^2$ is $O(n^3)$.
  - $n < 2^n$ is $O(n)$

Big-Omega

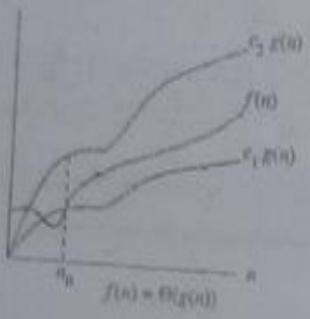


- $\exists c, n_0 > 0$ such that $f(n) \geq c\, g(n)$ when $n \geq n_0$
- $f(n)$ grows no slower than $g(n)$ for "large" n

Big-Omega

- $f(n) = \Omega(g(n))$:
  If there are positive constants c and n_0 such that
  $$f(n) \geq c\, g(n) \text{ when } n \geq n_0$$
- The growth rate of f(n) is *greater than or equal to* the growth rate of g(n).

$f(n) = \Theta(g(n))$



- the growth rate of f(n) *is the same as* the growth rate of g(n)

Big-Theta

- $f(n) = \Theta(g(n))$ if and only if
  $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$
- The growth rate of f(n) *equals* the growth rate of g(n)
- Example: Let $f(n)=n^2$, $g(n)=2n^2$
  - We write $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$, thus $f(n) = \Theta(g(n))$.

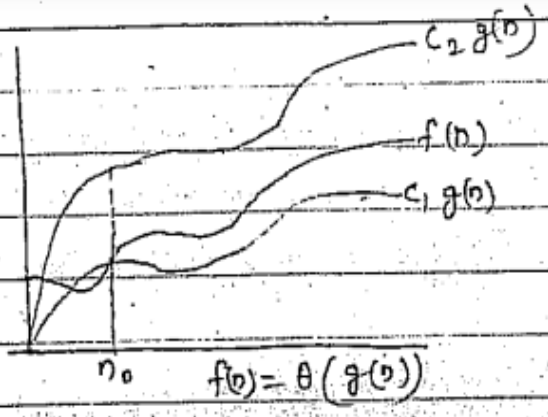
Little-oh

- $f(n) = o(g(n))$:
  If $f(n) = O(g(n))$ and $f(n) \neq \Theta(g(n))$
- The growth rate of f(n) is *less than* the growth rate of g(n)

Θ- Notation (Tightly Bound)

This notation bounds a function to within constant factors.

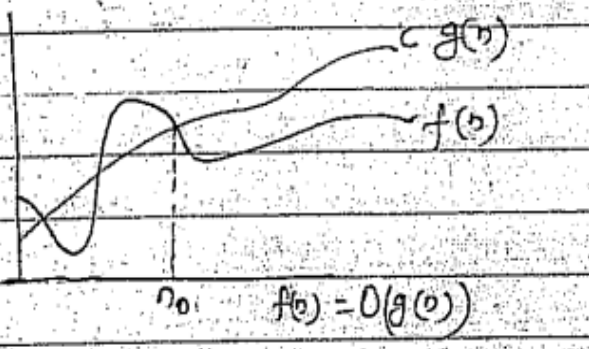
$f(n) = \Theta(g(n))$ if there exists positive constants n_0, c_1 and c_2 such that to the right of n_0, the value of $f(n)$ always lies between $c_1 g(n)$ and $c_2 g(n)$.



O- Notation (Upper Bound)

This notation gives an upper bound for a function to within a constant factor.

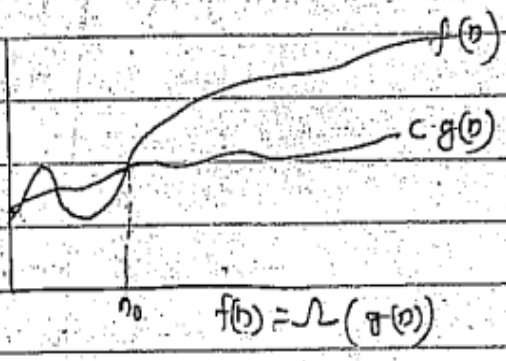
$f(n) = O(g(n))$ if there exist positive constants n_0 and c such that to the right of n_0, the value of $f(n)$ always lies on or below $cg(n)$.



Ω- Notation (Lower Bound)

This notation gives a lower bound for a function to within a constant factor.

$f(n) = \Omega g(n)$ if there are positive constants n_0 and c such that to the right of n_0, the value of $f(n)$ always lie on or above $cg(n)$.



Based on Big Oh Notation, the algorithm can be categorized as:

- Constant time ($O(1)$) algorithms.
- Logarithmic time $O(\log n)$
- Exponential time ($O(k^n)$, $k > 1$)
- Linear time ($O(n)$)
- polynomial time ($O(n^k)$, $k > 1$)