Lecture 2

Growth of Functions - Asymptotic Notations

Growth of Functions

- We can sometimes determine the exact running time of the algorithm, however, the *extra precision is not usually* worth the effort of computing it.

- For large inputs, the *multiplicative constants* and *lower order terms* of an exact running time are dominated by the effects of the *input size itself*.

Things to consider when Analyzing Algorithms

1. Ignore constants. For example,

   $f(n) = 25n^2$

   or

   $f(n) = 25n^2 + 2000$

   $f(n) = O(n^2)$

2. Ignore small terms

   $f(n) = 25n^3 + 30n^2 + 10n$

   $f(n) = O(n^3)$

3. Application Independent

   i.e. not dependent on any tool, platform or application.

Asymptotic Performance

- In this course, we care most about *asymptotic performance. i.e.*
  - ***How does the algorithm behave as the problem size gets very large?***
    - Running time
    - Memory/storage requirements
    - Bandwidth/power requirements/logic gates/etc.

- When we look at the *input sizes large enough* to make only the order of growth of the running time relevant, we are studying the *asymptotic efficiency* of the algorithm.
- That is we are concerned with how the running time of the algorithm increases with the size of the input *in the limit*, as the size of the input increases without any bound.
- We will study standard methods for simplifying the asymptotic analysis of the algorithm. We will begin with by defining several types of "asymptotic notations" like Theta (Θ) Notation, Big-Oh (O) Notation and Omega (Ω) Notation.

Theta Notation (Θ)

- For a given function g(n), we denote by Θ(g(n)) the set of functions

Θ(g(n)) = { f(n) : there exist positive constants c1,c2 and n0 such that

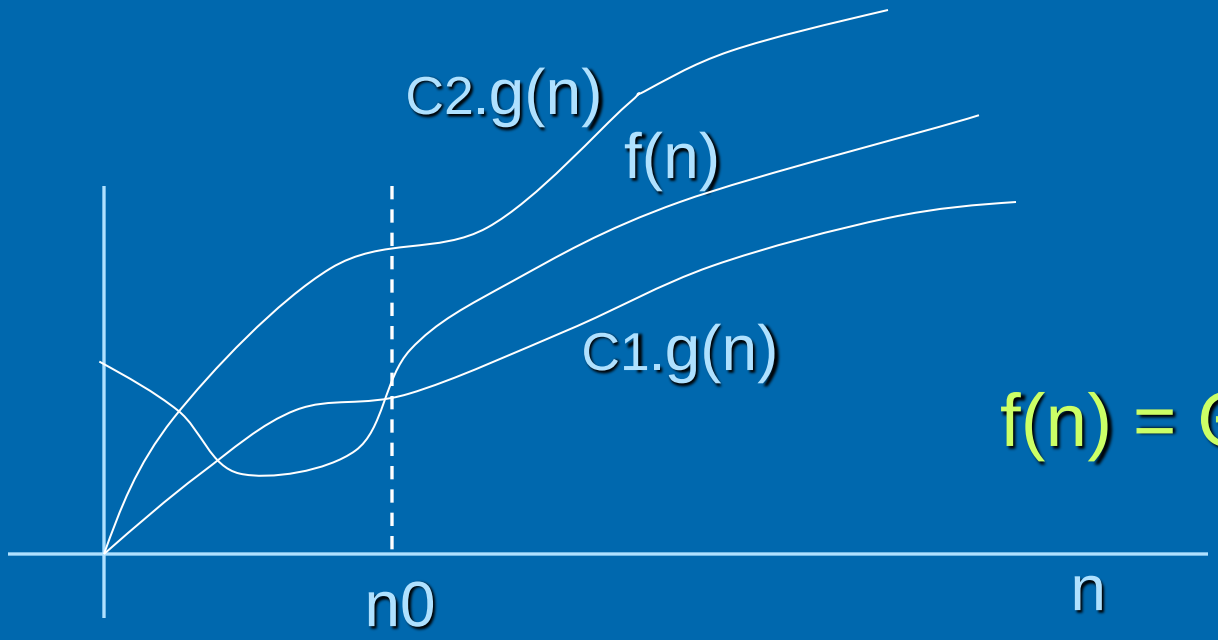
$0 \leq c1.g(n) \leq f(n) \leq c2.g(n)$ for all $n \geq n0$ }[1]

[1] colon in set means “such that”

- A function f(n) belongs to the set Θ(g(n)) if there exist positive constants c1 and c2 such that it can be *sandwiched* between c1g(n) and c2g(n) for sufficiently large n.

- Since Θ(g(n)) is a set, we could write f(n) Є Θ(g(n)) to indicate that f(n) is a member of Θ(g(n)). Instead we will usually write f(n) = Θ(g(n)).

Є means “belongs to”



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

- In the above figure, for all values of n to the right of n0, the value of f(n) lies at or above C1.g(n) and at or below C2.g(n).
- In other words, for all n ≥ n0, the function f(n) is equal to g(n) to *within constant factor*.
- We say that g(n) is ***asymptotic tight bound*** for f(n).

- Lets us justify this intuition by using the formal definition to show that $f(n) = n^2/2-3n = \Theta(n^2)$.
- To do so we must find the constants c1,c2, and n0 such that

$$c1.n^2 \leq n^2/2-3n \leq c2.n^2$$

for all $n \geq n0$. dividing by n^2

$$c1 \leq 1/2-3/n \leq c2$$

The right hand side inequality can be made to hold for any value of $n \geq 1$ by choosing $c2 \geq ½$.

Likewise , the left hand side inequality can be made to hold for any value of $n \geq 7$ by choosing $c1 \leq 1/14$. ***Thus***, by choosing $c1=1/14$, $c2=1/2$ and $n0 = 7$, we can verify that $n^2/2-3n = \Theta(n^2)$.

- In above example, certainly other choices for the constants exist, but the important thing is that some choice exists.

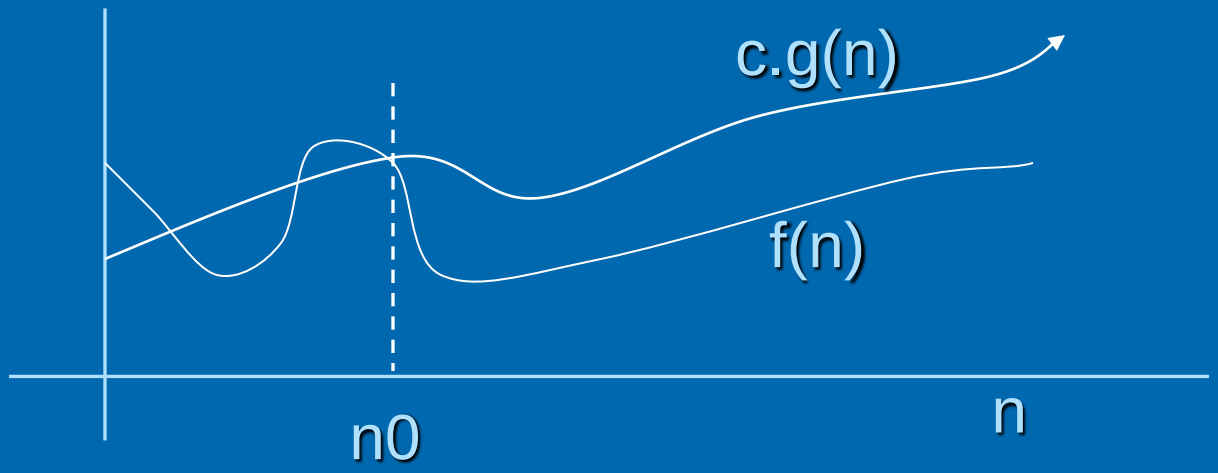
- Since any constant is a degree-0 polynomial, we can express any constant function as $\Theta(n^0)$ or $\Theta(1)$. We shall always use the notation $\Theta(1)$ to mean either constant or constant function with respective to some variable.

Big-Oh Notation (*O*)

- When we have only an ***asymptotic upper bound***, we use *O* – notation. For a given function g(n), we denote by *O*(g(n)) the set of functions

O(g(n)) = { f(n) : there exist positive constants c, and n0 such that

0 ≤ f(n) ≤ c.g(n) for all n ≥ n0 }[1]



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

- We write $f(n) = O(g(n))$ to indicate that $f(n)$ is a member of the set $O(g(n))$.

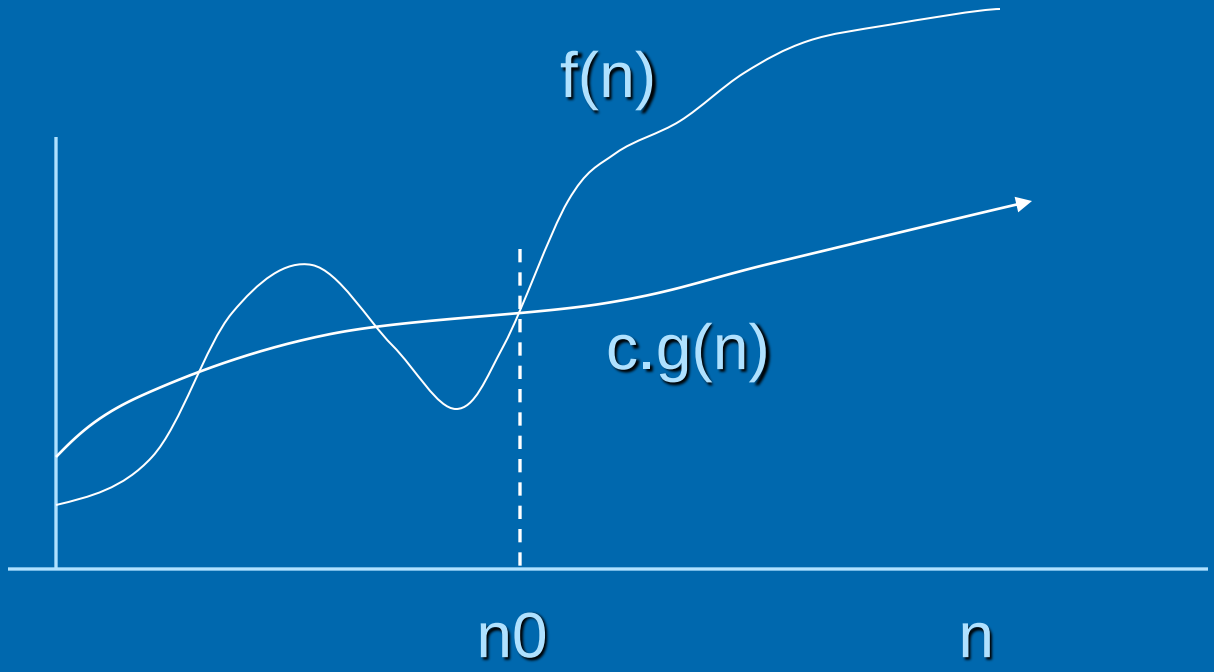
- Note that $f(n) = \Theta(g(n))$ implies $f(n) = O(g(n))$, since Θ notation is stronger notation than O notation. So for quadratic function $an^2+bn + c$, where $a>0$, is in $\Theta(n^2)$ also shows that any quadratic function is in $O(n^2)$.

Omega Notation (Ω)

- Just as *O*-notation provides an upper bound on a function, Ω notation provides an ***asymptotic lower bound***. For a given function g(n), we denote by Ω(g(n)) the set of functions

Ω(g(n)) = { f(n) : there exist positive constants c, and n0 such that

0 ≤ c.g(n) ≤ f(n) for all n ≥ n0 }[1]



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

Example:

- Objective is to find the maximum of an unordered set having N elements.
  - Input : An unordered list.
  - Output : Maximum of input set.
- Algorithm

```
1. int Max = 0 //assume S[N] is filled with +ve numbers
2. For (int I = 0;I<N;I++)
3.  {
4.      If (S[I] > Max)
5.          Max = S[I]
6.  }
7. cout<< Max
```

Simple Analysis of previous Algorithm

Instruction	No of times Executed	cost
1	1	1
2	N	N
3	-	-
4	N	N
5	N	N
6	-	-
7	1	1

So f(n) = 1 + N + N + N + 1 = 2 + 3N = 3N+2

so f(n) = *O* (N) or *O* (n)

NOTE : we always look for worst case of each instruction to be executed while finding Big-Oh (*O*). (Assume all instructions take *unit time in above example)*.

- The worst case running time of an algorithm is an upper bound on the running time for an input.
- Knowing it, gives us guarantee that algorithm will never take any longer.
- We need not make some educated guess about the running time and hope that it never gets much worse.

Types of Functions (Bounding Functions)

1. Constant Function: O (1)
   For example, addition of two numbers will take same for Worst case, Best case and Average case.
2. Logarithmic Function: O (log(n))
3. Linear Function : O (n)
4. O (n.log(n))
5. Quadratic Function : O (n^2)
6. Cubic Function : O (n^3)
7. Polynomial Function : O (n^k)
8. Exponential Function : O (2^n)

   etc…

Some facts about summation

- If c is a constant

$$\sum_{i=1}^{n} c\,a_i = c \sum_{i=1}^{n} a_i$$

And

$$\sum_{i=1}^{n} (a_i + b_i) = \sum_{i=1}^{n} a_i + \sum_{i=1}^{n} b_i$$

Mathematical Series

$$\sum_{i=1}^{n} i = 1 + 2 + \ldots + n = \frac{n(n+1)}{2} = \Theta(n^2)$$

$$\sum_{i=1}^{n} i^2 = 1 + 4 + 9 + \ldots + n^2 = \frac{2n^3 + 3n^2 + n}{6} = \Theta(n^3)$$

$$\sum_{i=0}^{n} x^i = 1 + x + x^2 + \ldots + x^n = \frac{x^{(n+1)} - 1}{x - 1}$$

- If $0 < x < 1$ then this is $\Theta(1)$, and if $x > 1$, then this is $\Theta(x^n)$.

- **Harmonic series For n ≥ 0**

$$
\begin{aligned}
H_n &= \sum_{i=1}^{n} \frac{1}{i} \\
&= 1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n} \approx \ln n \\
&= \Theta(\ln n)
\end{aligned}
$$

Analysis: A Harder Example

- Let us consider a harder example.

```
NESTED-LOOPS()
1   for i ← 1 to n
2   do
3       for j ← 1 to 2i
4       do k = j ...
5           while (k ≥ 0)
6           do k = k − 1 ...
```

- How do we analyze the running time of an algorithm that has complex nested loop? The answer is we write out the loops as summations and then solve the summations. To convert loops into summations, we work from inside-out.

- Consider the *inner most while* loop.

```
NESTED-LOOPS()
1   for i ← 1 to n
2   do for j ← 1 to 2i
3       do k = j
4           while (k ≥ 0)
5           do k = k − 1
```

- It is executed for k = j, j - 1, j - 2, . . . , 0. Time spent inside the while loop is constant. Let I() be the time spent in the while loop. Thus

$$I(j) = \sum_{k=0}^{j} 1 = j + 1$$

- Consider the *middle for* loop.

```
NESTED-LOOPS()
1  for i ← 1 to n
2  do for j ← 1 to 2i
3     do k = j
4        while (k ≥ 0)
5        do k = k − 1
```

- Its running time is determined by i. Let M() be the time spent in the for loop:

$$
\begin{aligned}
M(i) &= \sum_{j=1}^{2i} I(j) \\
&= \sum_{j=1}^{2i} (j+1) \\
&= \sum_{j=1}^{2i} j + \sum_{j=1}^{2i} 1 \\
&= \frac{2i(2i+1)}{2} + 2i \\
&= 2i^2 + 3i
\end{aligned}
$$

- Finally the *outer-most for* loop.

```
NESTED-LOOPS()
1  for i ← 1 to n
2  do for j ← 1 to 2i
3      do k = j
4          while (k ≥ 0)
5          do k = k − 1
```

- Let T() be running time of the entire algorithm

$$
\begin{aligned}
T(n) &= \sum_{i=1}^{n} M(i) \\
&= \sum_{i=1}^{n} (2i^2 + 3i) \\
&= \sum_{i=1}^{n} 2i^2 + \sum_{i=1}^{n} 3i \\
&= 2\frac{2n^3 + 3n^2 + n}{6} + 3\frac{n(n+1)}{2} \\
&= \frac{4n^3 + 15n^2 + 11n}{6} \\
&= \Theta(n^3)
\end{aligned}
$$

Theta Notation (Θ) Example

- Let $f(n) = 8n^2 + 2n – 3$. Let's show why f(n) is not in some other asymptotic class. First, let's show that $f(n) \neq \Theta(n)$.
- If this were true, we would have had to satisfy both the upper and lower bounds. The lower bound is satisfied because $f(n) = 8n^2 + 2n - 3$ does grow at least as fast asymptotically as n.
- But the upper bound is false. Upper bounds requires that there exist positive constants c_2 and n_0 such that $f(n) \leq c_2 n$ for all $n \geq n_0$.

- Informally we know that $f(n) = 8n^2 + 2n - 3$ will eventually exceed $c_2 n$ no matter how large we make c_2.
- To see this, suppose we assume that constants c_2 and n0 did exist such that $8n^2 + 2n - 3 \leq c_2 n$ for all $n \geq n_0$ since this is true for all sufficiently large n then it must be true in the limit as n tends to infinity. If we divide both sides by n, we have

$$\lim_{n \to \infty} \left(8n + 2 - \frac{3}{n} \right) \leq c_2.$$

- It is easy to see that in the limit, the left side tends to ∞. So, no matter how large c_2 is, the statement is violated. Thus $f(n) \neq \Theta(n)$.

- Let's show that $f(n) \neq \Theta(n^3)$. The idea would be to show that the lower bound $f(n) \geq c_1 n^3$ for all $n \geq n_0$ is violated. (c_1 and n_0 are positive constants). Informally we know this to be true because any cubic function will overtake a quadratic.
- If we divide both sides by n^3:

$$\lim_{n \to \infty} \left(\frac{8}{n} + \frac{2}{n^2} - \frac{3}{n^3} \right) \geq c_1$$

- The left side tends to 0. The only way to satisfy this is to set c1 = 0. But by hypothesis, c1 is positive. This means that $f(n) \neq \Theta(n^3)$.

References: Mathematical Series*

$$\sum_{k=1}^{n} k = \frac{1}{2} n (n+1)$$

$$\sum_{k=1}^{n} k^2 = \frac{1}{6} n (n+1) (2n+1)$$

$$\sum_{k=1}^{n} k^3 = \frac{1}{4} n^2 (n+1)^2$$

$$\sum_{k=1}^{n} k^4 = \frac{1}{30} n (n+1) (2n+1) \left(3n^2 + 3n - 1\right)$$

$$\sum_{k=1}^{n} k^5 = \frac{1}{12} n^2 (n+1)^2 \left(2n^2 + 2n - 1\right)$$

* http://mathworld.wolfram.com/PowerSum.html

References: Mathematical Series

$$\sum_{k=1}^{n} k^6 = \frac{1}{42} n(n+1)(2n+1)\left(3n^4 + 6n^3 - 3n + 1\right)$$

$$\sum_{k=1}^{n} k^7 = \frac{1}{24} n^2 (n+1)^2 \left(3n^4 + 6n^3 - n^2 - 4n + 2\right)$$

$$\sum_{k=1}^{n} k^8 = \frac{1}{90} n(n+1)(2n+1)\left(5n^6 + 15n^5 + 5n^4 - 15n^3 - n^2 + 9n - 3\right)$$

$$\sum_{k=1}^{n} k^9 = \frac{1}{20} n^2 (n+1)^2 \left(n^2 + n - 1\right)\left(2n^4 + 4n^3 - n^2 - 3n + 3\right)$$

$$\sum_{k=1}^{n} k^{10} = \frac{1}{66} n(n+1)(2n+1)\left(n^2 + n - 1\right)\left(3n^6 + 9n^5 + 2n^4 - 11n^3 + 3n^2 + 10n - 5\right)$$