Data Structures and Algorithms Design (DSECLZG519)

BITS Pilani
Hyderabad Campus

Febin.A.Vahab
Asst.Professor(Offcampus)
BITS Pilani,Bangalore

SESSION 2 -PLAN

Online Sessions(#)	List of Topic Title	Text/Ref Book/external resource
1 *(Covered Already)*	*Algorithms and it's Specification, Experimental Analysis, Analytical model-Random Access Machine Model, Counting Primitive Operations, Basic Operation method, Analyzing non recursive algorithms,Order of growth*	T1: 1.1, 1.2
2	Notion of best case, average case and worst case. Use of asymptotic notations- Big-Oh, Omega and Theta Notations. Correctness of Algorithms. Analyzing Recursive Algorithms: Recurrence relations, Iteration Method.	T1: 1.4, 2.1

Order of growth-Refresher!

	constant	*logarithmic*	*linear*	*N-log-N*	*quadratic*	*cubic*	*exponential*
n	$O(1)$	$O(\log n)$	$O(n)$	$O(n \log n)$	$O(n^2)$	$O(n^3)$	$O(2^n)$
1	1	1	1	1	1	1	2
2	1	1	2	2	4	8	4
4	1	2	4	8	16	64	16
8	1	3	8	24	64	512	256
16	1	4	16	64	256	4,096	65536
32	1	5	32	160	1,024	32,768	4,294,967,296
64	1	6	64	384	4,069	262,144	1.84×10^{19}

Refresher!!

- Which kind of growth best characterizes each of these functions?
- $(3/2)^n$
- $3n$
- 1
- $(3/2)n$
- $2n^3$
- 2^n
- $3n^2$
- 1000

Refresher!!

- Which kind of growth best characterizes each of these functions?

	Constant	Linear	Polynomial	Exponential
(3/2)^n				✓
3n		✓		
1	✓			
(3/2)n		✓		
2n^3			✓	
2^n				✓
3n^2			✓	
1000	✓			

Refresher!!

Consider a program with time complexity $O(n^2)$.

- For the input of size n, it takes 5 seconds.
- If the input size is doubled (2n). –
- **then it takes 20 seconds**.

Consider a program with time complexity O(n).

- For the input of size n, it takes 5 seconds.
- If the input size is doubled (2n). –
- **then it takes 10 seconds.**

Consider a program with time complexity $O(n^3)$.

- For the input of size n, it takes 5 seconds.
- If the input size is doubled (2n) . –
- **then it takes 40 seconds**.

Refresher!!

What is the order of growth of the below function?

```
int fun1(int n)
{
  int count = 0;
  for (int i = 0; i < n; i++)
    for (int j = i; j > 0; j--)
      count = count + 1;
  return count;
}
```

Ans:O(n^2)

Exercises-Analysis of algorithms

Adobe Acrobat
Document

Time Complexity- Why should we care?

for i =2 to n-1

If i divides n

n is not prime

----

1 ms for a division

In worst case (n-2) times.

n =11 ------?

n = 101 -------?

for i ← 2 to √n

if i divides n

n is not prime

----

1 ms for a division

In worst case (√n-1) times.

n=11,(3-1) = 2ms

n=101,(√101-1) times = 9ms

Notion of best case and worst case

- Best case: where algorithm takes the least time to execute.
  - In arrayMax ex, occurs when A[0] is the maximum element.
  - T(n)=5n
- Worst case :where algorithm takes maximum time.
  - Occurs when elements are sorted in increasing order so that variable *currentMax* is reassigned at each iteration of the loop.
  - T(n)=7n-2

Algorithm ***arrayMax*(*A*,*n*)**

currentMax ← *A*[0]
for (*i* =1; i<n; i++)
if *A*[*i*] > *currentMax* **then**
currentMax ← *A*[*i*]
return *currentMax*

Use of asymptotic notation

- How the running time of an algorithm increases with the input size, as the size of the input increases without bound?

- Used to compare the algorithms based on the order of growth of their basic operations.

Informal Introduction

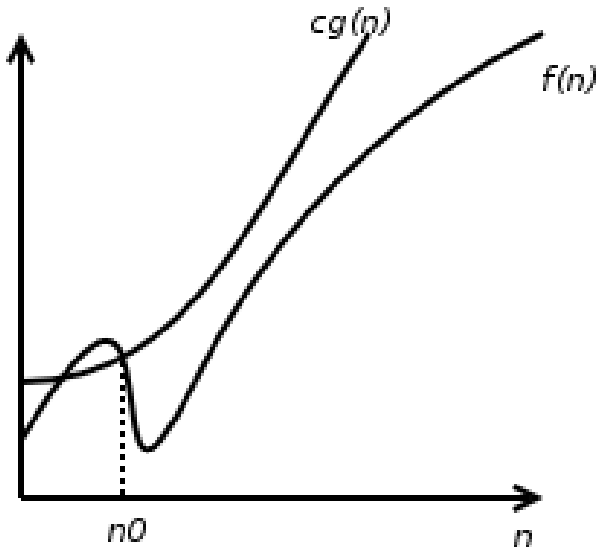
- $O(g(n))$ is the set of all functions with a lower or same order of growth as $g(n)$ (to within a constant multiple, as n goes to infinity)

- $\Omega(g(n))$, stands for the set of all functions with a higher or same order of growth as $g(n)$ (to within a constant multiple, as n goes to infinity).

- $\Theta(g(n))$ is the set of all functions that have the same order of growth as $g(n)$ (to within a constant multiple, as n goes to infinity).

Big-Oh Notation

- Let f and g be functions from nonnegative numbers to nonnegative numbers. Given functions $\boldsymbol{f(n)}$ and $\boldsymbol{g(n)}$, we say that $\boldsymbol{f(n)}$ is $\boldsymbol{O(g(n))}$ if there is a real constant c>0 and an integer constant $\boldsymbol{n_0}$ >=1 such that $\boldsymbol{f(n)} \leq \boldsymbol{cg(n)}$ for every integer $\boldsymbol{n} \geq \boldsymbol{n_0}$



Big-Oh Notation

- Big-Oh notation provides an upper bound on a function to within a constant factor.
- To prove big-Oh, find witnesses, specific values for C and n0, and prove n >= n0 implies $f(n) \leq C * g(n)$.

One Approach for Finding Witnesses

- *Generate a table for f(n) and g(n) using n = 1, n = 10 and n = 100. [Use values smaller than 10 and 100 if you wish.]*
- *Guess* $C = \lceil f(1)/g(1) \rceil$ *(or* $C = \lceil f(10)/g(10) \rceil$*).*
- *Check that* $f(10) \leq C * g(10)$ *and* $f(100) \leq C * g(100)$*.[If this is not true, f(n) might not be O(g(n)).]*
- *Choose n0 = 1 (or n0 = 10).*
- *Prove that* $\forall n(n >= n0 \rightarrow f(n) \leq C * g(n))$*.*
- *[It's ok if you end up with a larger, but still constant, value for C.]*

One Approach for Finding Witnesses

– Assume n > 1 if you chose n0 = 1 (or n > 10 if you chose n0 = 10).
– To prove $f(n) \leq C * g(n)$, you need to find expressions larger than f(n) and smaller than $C * g(n)$.
– If the lowest-order term is negative, just eliminate it to obtain a larger expression.
– Repeatedly use n > k and 2n > 2k and 3n > 3k and so on to "convert" the lowest-order term into a higher-order term.
– Check that your expressions are less than $C * g(n)$ by using n = 100.

Big-Oh Notation Example 1

Show that 3n + 7 is O(n).

- In this case, f(n) = 3n + 7 and g(n) = n.

n	f(n)	g(n)	Ceil(f(n)/g(n))
1	10	1	10
10	37	10	4
100	307	100	4

- This table suggests trying n0 = 1 and c = 10 or
- n0 = 10 and c = 4.
- Proving either one is good enough to prove big-Oh.

Big-Oh Notation Example 1

Try n0 = 1 and c = 10.

- Want to prove ***n > 1 implies 3n + 7 ≤ 10n.***
- Assume n > 1. Want to show ***3n + 7 ≤ 10n.***
- 7 is the lowest-order term, so work on that first.
- **n > 1 implies 7n > 7**, which implies
- 3n + 7 < 3n + 7n = 10n.
- This finishes the proof.

Big-Oh Notation Example 2

- Show that $n^2 + 2n + 1$ is $O(n^2)$.
  - In this case, f(n) = n^2 + 2n + 1 and g(n) = n^2.

n	f(n)	g(n)	Ceil(f(n)/g(n))
1	4	1	4
10	121	100	2
100	10201	10000	2

  - This table suggests trying n0 = 1 and C = 4
  - or n0 = 10 and C = 2.

Big-Oh Notation Example 2

- Try n0 = 1 and c = 4.
  - Want to prove ***n > 1 implies*** $\boldsymbol{n^2 + 2n + 1 \leq 4 n^2}$.
  - Assume n > 1.
  - Want to show $n^2 + 2n + 1 \leq 4 n^2$.
  - Work on the lowest-order term first.
  - n > 1 implies
  - $n^2 + 2n + 1$

  $< n^2 + 2n + n$

  $= n^2 + 3n$
  - Now 3n is the lowest-order term.
  - n > 1 implies 3n > 3 and $3 n^2 > 3n$, which implies
  - $n^2 + 3n$

  $< n^2 + (3n)n$

  $= n^2 + 3 n^2 = 4 n^2$. This finishes the proof.

Big-Oh Notation Example 3

- Example
- 2n+10 is O(n)

  ie.

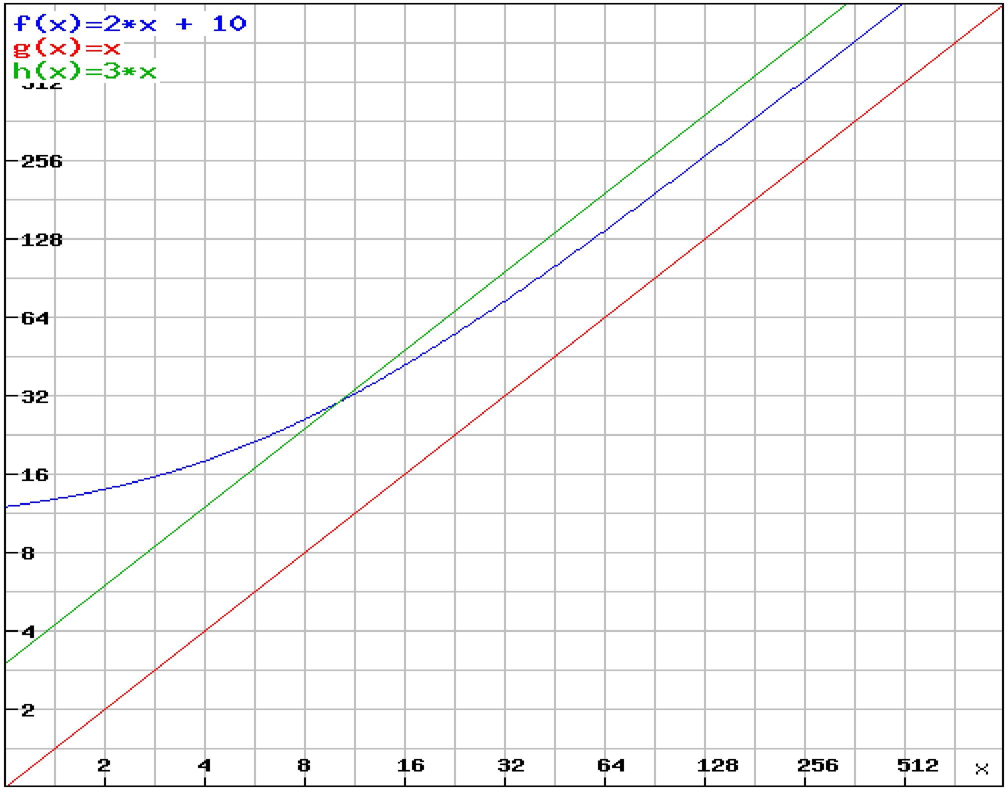
  2n+10 <=c*n

  10/n   <=(c-2)

  10/c-2 <=n

          n >=10/(c-2)

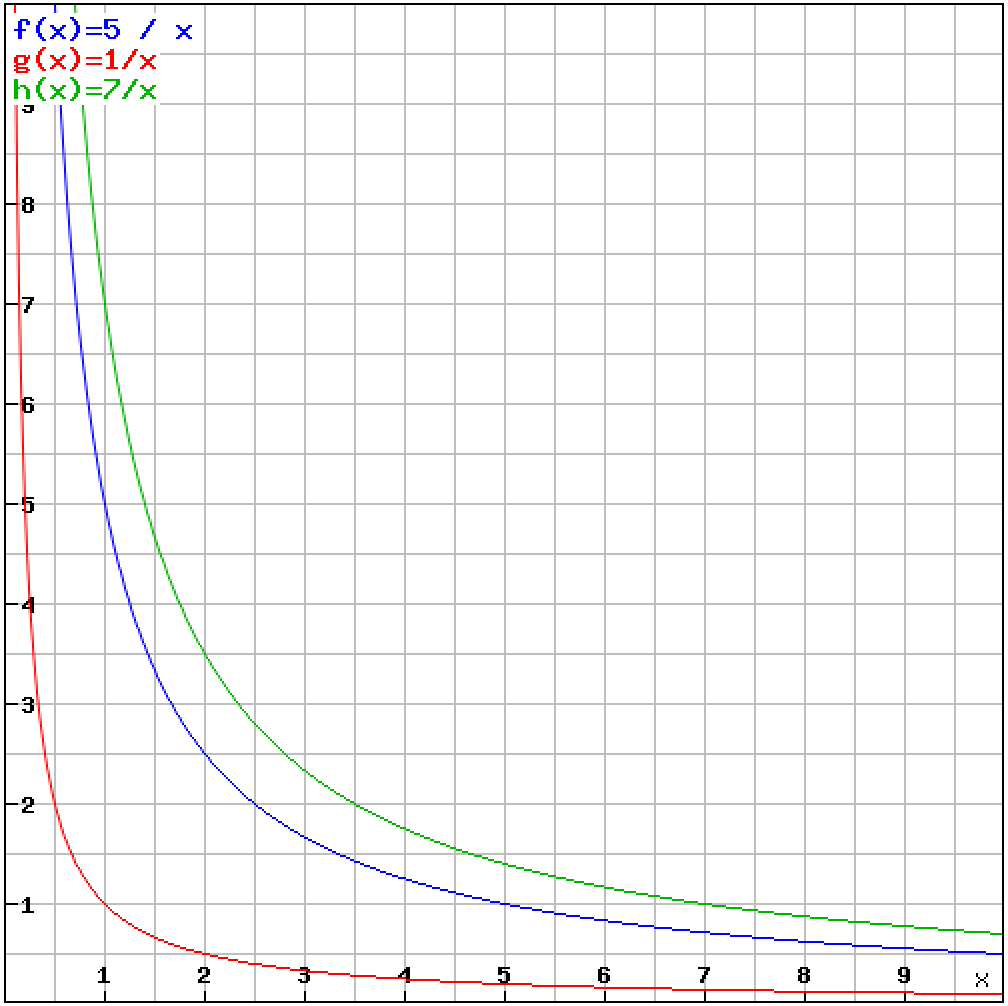
  ie .If c=3,n=10



https://rechneronline.de/function-graphs/

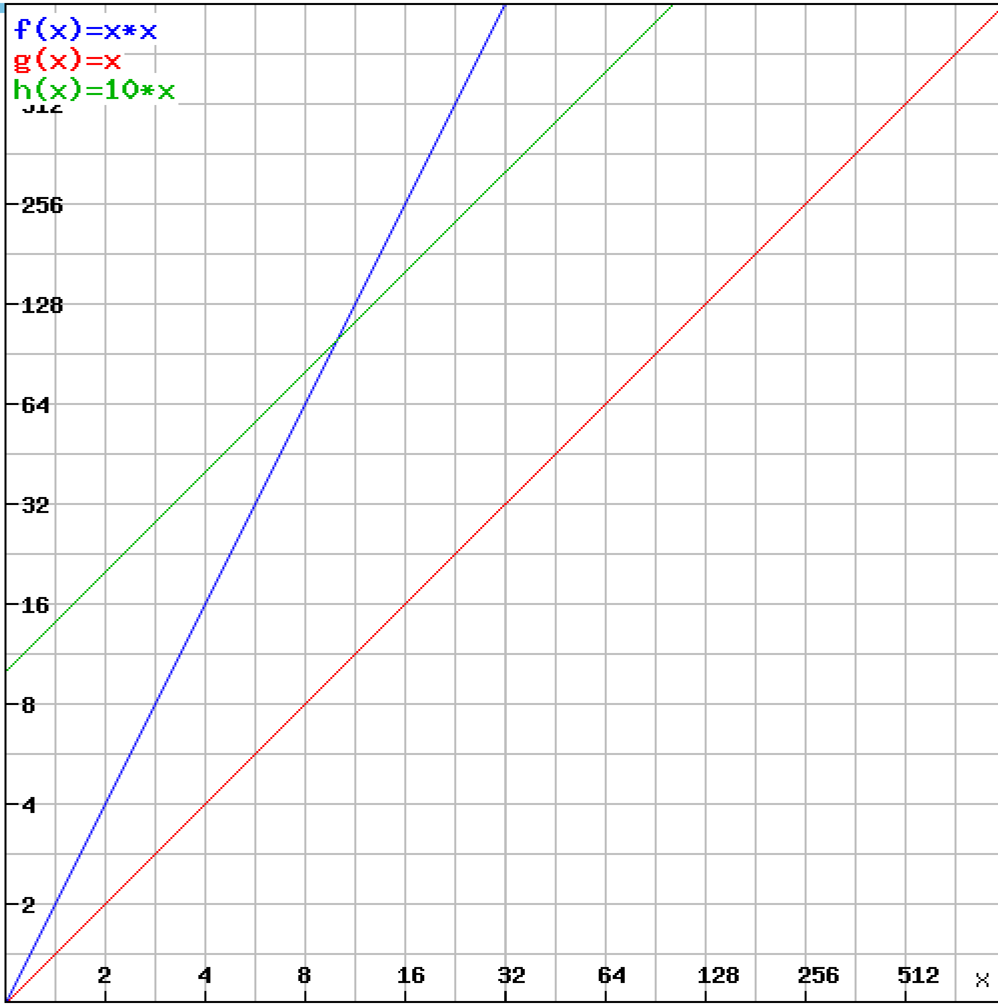
Big-Oh Notation Example 4

- Example
- 5/x is O(1/x)
- 5/x<=c*1/x
- c>=5 for x>=1



Big-Oh Notation Example 5

- Example:
- The function n^2 is not $O(n)$
  - $n^2 \leq cn$
  - $n \leq c$
  - The above inequality cannot be satisfied since c must be a positive constant



Big-Oh Notation Example 6

- Show that $8n^3 - 12n^2 + 6n - 1$ is $O(n^3)$.
  - In this case, $f(n) = 8n^3 - 12n^2 + 6n - 1$ and $g(n) = n^3$

n	f(n)	g(n)	Ceil(f(n)/g(n))
1	1	1	1
10	6859	1000	7
100	7880599	1000000	8

  - This table suggests trying n0 = 100 and C = 8.

Big-Oh Notation Example 6

- Try n0 = 100 and c = 8.
  - Want to prove n > 100 implies $8n^3 - 12n^2 + 6n - 1 \leq 8n^3$
  - Assume n > 100. Want to show $f(n) \leq 8n^3$.
  - The lowest-order term is negative, so eliminate it.
  - $8n^3 - 12n^2 + 6n - 1 < 8n^3 - 12n^2 + 6n$.
  - n > 100 implies n > 6, $n^2 > 6n$ which implies
  - $8n^3 - 12n^2 + 6n < 8n^3 - 12n^2 + n^2 = 8n^3 - 11n^2$.
  - Now lowest-order term is negative, so eliminate.
  - n > 100 implies $8n^3 - 11n^2 \leq 8n^3$.
  - This finishes the proof.

Big-Oh Notation -More examples

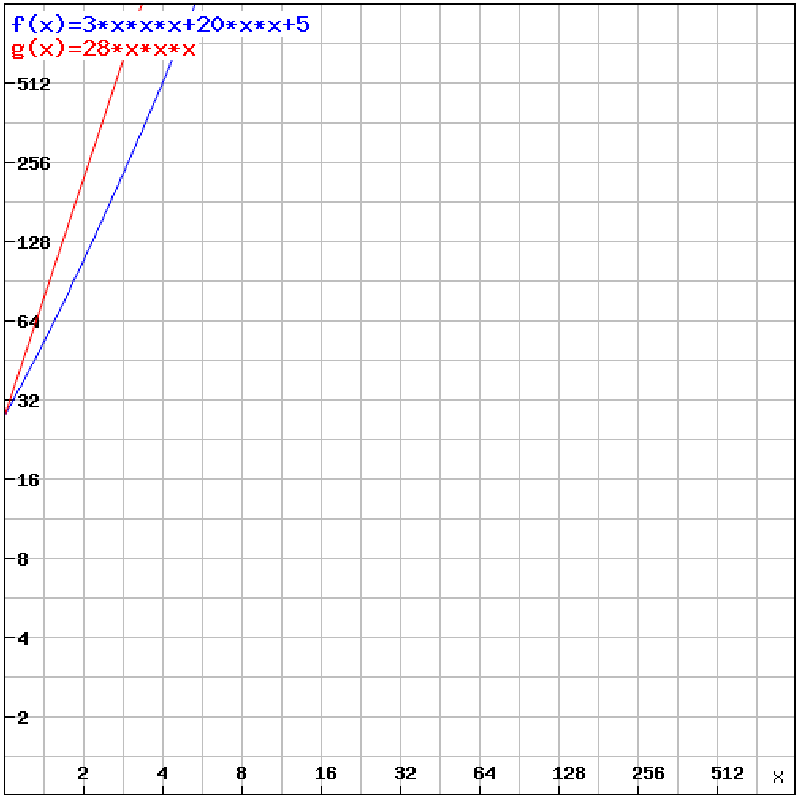
- 7n-2 is O(n)
- $3n^3 + 20n^2 + 5$ is $O(n^3)$
- 3 log n+ log logn is O(log n)

- Solution using one method is given below. Try other one.

Big-Oh Notation -More examples

- $7n-2$ is $O(n)$
  - $7n-2<=cn$
  - $7-2/n<=c$
  - $c>=7-2/n$
  - **n0=1 and c=7** is true .
- $3n^3 + 20n^2 + 5$ is $O(n^3)$
  - $3n^3+20n^2+5<=c.n^3$
  - $3+20/n+5/n^3<=c$
  - $c>=3+20/n+5/n^3$

  c>=28 and n0>=1 is true



Big-Oh Notation -More examples

- 3 log n+ log log n is O(log n)
  - 3log n+ log logn< c.log n
  - Let n=8,
  - 3*3+log 3<= 3c
  - 9+1.58 <=3c
  - c >=4

OR

- 3 log n + log log n <=4 logn , for n >=2 .
  - Note that log log n i s not even defined for n = 1 . That is why we use n >= 2.

Big-Oh Notation

- If is $f(n)$ a polynomial of degree d, then $f(n)$ is $O(n^d)$, i.e.,
  1. Drop lower-order terms
  2. Drop constant factors
- Use the smallest possible class of functions
  - Say "$2n$ is $O(n)$" instead of "$2n$ is $O(n^2)$"
- Use the simplest expression of the class
  - Say "$3n + 5$ is $O(n)$" instead of "$3n + 5$ is $O(3n)$"

Big-Oh Notation Theorem 1.7

Let d(n), e(n), f(n), and g(n) be functions mapping nonnegative integers to nonnegative reals. Then

1. ***If d(n) is O(f(n)), then ad(n) is O(f(n)), for any constant a> 0.***
2. ***If d(n) is O(f(n)) and e(n) is O(g(n)), then d(n) + e(n) is O(f(n) + g(n)).***
3. If d(n) is O(f(n)) and e(n) is O(g(n)), then d(n)e(n) is O(f(n)g(n)).
4. If d(n) is O(f(n)) and f(n) is O(g(n)), then d(n) is O(g(n)).
5. n^x is $O(a^n)$ for any fixed $x > 0$ and $a > 1$.
6. ***Log n^x is O(log n) for any fixed $x > 0$.***

Big-Oh Notation
Proof of Theorem 1.7

1.If d(n) is O(f(n)), then a*d(n) is O(f(n)) for any constant a>0.

- $d(n) \le C * f(n)$ where C is a constant
- $a * d(n) \le a * C * f(n)$
- $a * d(n) \le C1 * f(n)$ where $a* C = C1$
- Therefore $a*d(n) = O(f(n))$

Big-Oh Notation
Proof of Theorem 1.7

2.If d(n) is O(f(n)) and e(n) is O(g(n)), then d(n)+e(n) is O(f(n)+g(n)). The proof will extend to orders of growth

$d(n) \leq C1 * f(n)$ for all $n \geq n1$ where C1 is a constant

$e(n) \leq C2 * g(n)$ all $n \geq n2$ where C2 is a constant

$d(n) + e(n) \leq C1 * f(n) + C2 * g(n)$

$\leq C3 (f(n) + g(n))$ where $C3=\max\{C1,C2\}$

and $n \geq \max\{n1,n2\}$

Big-Oh Notation
Proof of Theorem 1.7

6.Log n ^{x} is O(log n) for any fixed x > 0.

$\log n^{x} <= c.\log n$

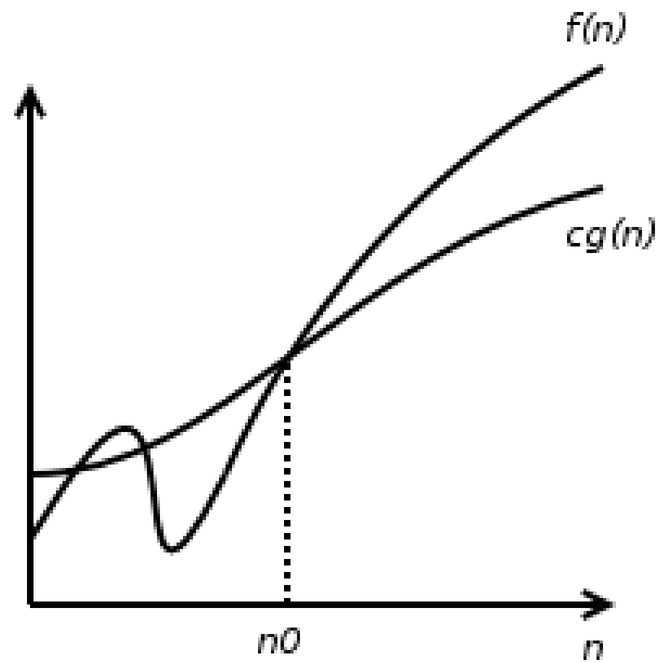
$x * \log n <= c. \log n$

$c >= x.$

Big-Omega Notation

- The function f(n) is said to be in Ω(g(n)) iff there exists a positive constant c and a positive integer n0 such that

  f(n)>= c.g(n) for all n>=n0.



- Asymptotic lower bound
- n^3 ∈ Ω(n^2)
- n^5+n+3 ∈ Ω(n^4)

Big-Omega Notation

- Big-Omega notation provides a lower bound on a function to within a constant factor.
- To prove big-Omega, find witnesses, specific values for C and n0, and prove n > n0 implies $f(n) \geq C * g(n)$.

Tricks for Proving Big-Omega

- Assume $n > 1$ if you chose $n0 = 1$ (or $n > 10$ if you chose $n0 = 10$).
- To prove $f(n) \geq C * g(n)$, you need to find expressions smaller than $f(n)$ and larger than $C * g(n)$.
- If the lowest-order term is positive, just eliminate it to obtain a larger expression.
- Repeatedly use $-n0 > -n$ and $-0.1n0 > -0.1n$ and so on to "convert" the lowest-order term into a higher-order term.
- Check that your expressions are greater than $C * g(n)$ by using $n = 100$.

Tricks for Proving Big-Omega

- Generate a table for f(n) and g(n). using n = 1, n = 10 and n = 100.[Use values smaller than 10 and 100 if you wish.]
- Guess 1/C = [g(1)/f(1)] (or more likely 1/C = [g(10)/f(10)]).
- Check that $f(10) \geq C * g(10)$ and $f(100) \geq C * g(100)$.[If this is not true, f(n) might not be (g(n)).]
- Choose n0 = 1 (or n0 = 10).
- Prove that $\forall n(n > n0 \rightarrow f(n) \geq C * g(n))$.[It's ok if you end up with a smaller, but still positive, value for C.]

Big-Omega Example 1

- Show that 3n + 7 is Ω(n).
  - In this case, f(n) = 3n + 7 and g(n) = n.

n	f(n)	g(n)	Ceil(g(n)/f(n))	C
1	10	1	1	1
10	37	10	1	1
100	307	100	1	1

  - This table suggests trying n0 = 1 and C = 1.
  - Want to prove **n > 1 implies 3n + 7 ≥ n.**
  - n > 1 implies 3n + 7 > 3n > n.

Big-Omega Example 3

- Show that n^2 − 2n + 1 is Ω(n^2).
- In this case, f(n) = n^2 − 2n + 1 and g(n) = n^2.

n	f(n)	g(n)	Ceil(g(n)/f(n))	C
1	10	1	1	1
10	81	100	2	1/2
100	9801	10000	1	1/2

- This table suggests trying n0 = 10 and C = 1/2.

Big-Omega Example 2

- Try n0 = 10 and C = 1/2.
  - Want to prove n > 10 implies $n^2 - 2n + 1 \geq n^2/2$.
  - Assume n > 10. Want to show $f(n) \geq n^2/2$.
  - The lowest-order term is positive, so eliminate.
  - $n^2 - 2n + 1 > n^2 - 2n$
  - $n > 10$ implies $-10 > -n$, implies $-2 > -0.2n$.
  - $-2 > -0.2n$ implies $n^2 - 2n > n^2 - 0.2n^2 = 0.8n^2$.
  - $n > 10$ implies $0.8n^2 > n^2/2$.
  - This finishes the proof.

Big-Omega Example 3

- Show that $n^3/8 - n^2/12 - n/6 - 1$ is $O(n^3)$.
- In this case,$f(n) = n^3/8 - n^2/12 - n/6 - 1$ and $g(n) = n^3$.

n	f(n)	g(n)	Ceil(g(n)/f(n))	C
1	-8	1	-1	-1
10	117.3	1000	9	1/9
100	124,182.3	1000000	9	1/9

- $C = -1$ is useless, so try $n0 = 10$ and $C = 1/9$

Big-Omega Example 3

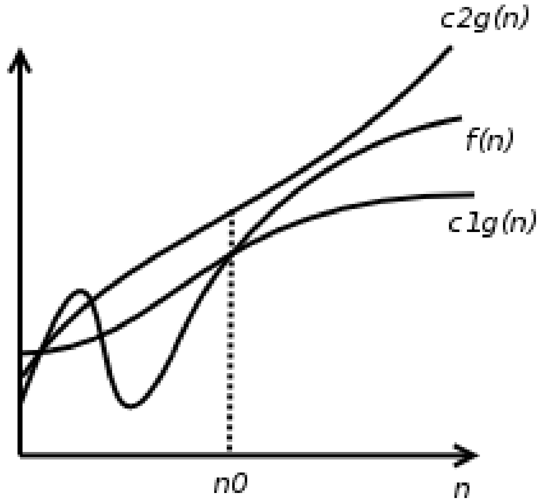
- Try n0 = 10 and C = 1/9.
  - Want to prove n > 10 implies $n^3/8 - n^2/12 - n/6 - 1 \geq n^3/9$
  - Assume n > 10, which implies the following:
  - $n^3/8 - n^2/12 - n/6 - 1$
  - $= (3n^3 - 2n^2 - 4n - 24)/24$
  - $> (3n^3 - 2n^2 - 4n - 2.4n)/24$
  - $> (3n^3 - 2n^2 - 7n)/24$
  - $> (3n^3 - 2n^2 - 0.7n^2)/24$
  - $> (3n^3 - 3n^2)/24$
  - $> (3n^3 - 0.3\, n^3)/24$
  - $> (3n^3 - n^3)/24$
  - $= (2n^3)/24 = n^3/12$
  - Ended up with n0 = 10 and C = 1/12, proving
  - n > 10 implies $n^3/8 - n^2/12 - n/6 - 1 \geq n^3/12$

Big-Theta Notation

- The function f(n) is said to be in $\Theta(g(n))$ iff there exists some positive constants c1 and c2 and a non negative integer n0 such that

  c1.g(n)<=f(n)<=c2.g(n) for all n>=n0

- **Asymptotic tight bound**
- an^2+bn+c ∈ Θ(n^2)
- n^2 ∈ Θ(n^2)



Examples – Ω and Θ

- **f(n)=5n^2 .Prove that f(n) is Ω(n)**
  - 5n^2>=c.n
  - c.n<=5n^2
  - c<=5n
  - If n=1,c<=5
  - 5*1<=4*1 hence the proof.

Examples – Ω and Θ

- **f(n)=5n^2 .Prove that f(n) is Ω(n)**
  - 5n^2>=c.n
  - c.n<=5n^2
  - c<=5n
  - If n=1,c<=5
  - 5*1<=4*1 hence the proof.
- Prove that f(n) is Θ(n^2)

Little-Oh and little omega Notation

- f(n) is o(g(n)) (or f(n) ∈ o(g(n))) if for any real constant $c > 0$, there exists an integer constant $n0 \geq 1$ such that
  - ***f(n)* < *c ∗ g(n)*** for every integer $n \geq n0$.

- f(n) is ω(g(n)) (or f(n) ∈ ω(g(n))) if for any real constant $c > 0$, there exists an integer constant $n0 \geq 1$ such that
  - ***f(n)* > *c * g(n)*** for every integer $n \geq n0$.

Little-Oh and Little omega Notation

- $12n^2 + 6n$ is $o(n^3)$
- $4n+6$ is $o(n^2)$
- $4n+6$ is $\omega(1)$
- $2n^9 + 1$ is $o(n^{10})$
- n^2 is $\omega(\log n)$

USING LIMITS

Little Oh- $= \lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$

Little Omega $= \lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$

Correctness of algorithm

- An algorithm is said to be correct if, for every input instance, it halts with the correct output.
- When it can be incorrect?
  - Might not halt on all input instances
  - Might halt with an incorrect answer

- Does it makes sense to think of incorrect algorithm?
  - Might be useful if we can control the error rate and can be implemented very fast

Analyzing Recursive Algorithms

- Recursive calls:-A procedure P calling itself-calls to P are for solving sub problems of smaller size.
- Recursive procedure call should always define a ***base case***.
- Base case-small enough that it can be solved directly without using recursion.

Analyzing Recursive Algorithms

- Algorithm recursiveMax(A,n)

  // Input : An array A storing n>=1 integers

  //Output: The maximum element in A

  if n = 1 then

  return A[0]

  return max{ recursiveMax(A,n-1),A[n-1]}

Analyzing Recursive Algorithms

- A ***recurrence*** is an equation or inequality that describes a function in terms of its value on smaller inputs.
- ***Recurrence equation***: defines mathematical statements that the running time of a recursive algorithm must satisfy
- Analysis of ***recursiveMax***
  - T(n)-Running time of algorithm on an input size n

$$T(n) = \begin{cases} 3 & \text{if } n=1 \\ T(n-1) + 7 & \text{otherwise} \end{cases}$$

Solving recurrences : Iterative Method

Analyzing Recursive Algorithms-Iterative method

- **General Plan-Iterative Method**
  - Identify the parameter to be considered based on the size of the input.
  - Identify the basic operation in the algorithm
  - Obtain the number of times the basic operation is executed.
  - Obtain an initial condition-base case
  - Obtain a recurrence relation
  - Solve the recurrence relation and obtain the order of growth and express using asymptotic notations.

Analyzing Recursive Algorithms-RecursiveMax

- Analysis of ***recursiveMax***
  - T(n)-Running time of algorithm on an input size n

$$T(n) = \begin{cases} 3 & \text{if } n=1 \\ T(n-1) + 7 & \text{otherwise} \end{cases}$$

```
Algorithm recursiveMax(A,n)
// Input : An array A storing  n>=1 in
 //Output: The maximum element in
 if  n = 1 then
 return A[0]
return max{ recursiveMax(A,n-1),A
```

Analyzing Recursive Algorithms- Example **1:-Factorial of a number**

– ***Algorithm fact(n)***

//Purpose: Computes factorial of n

//Input: A positive integer n

//Output: factorial of n

If(n=0)

return 1

return n*fact(n-1)

Analyzing Recursive Algorithms-Example **1:-Factorial of a number**

- **Analysis**
  - Parameter to be considered –n
  - Basic operation –Multiplication
  - T(n) = 0 if n=0

    1+T(n-1) Otherwise

Time taken to compute fact(n-1)

Time to multiply n*fact(n-1)

Analyzing Recursive Algorithms- Example **1:-Factorial of a number**

- **Solve the recurrence**

  $T(n) = T(n\text{-}1) + 1$

  $[T(n\text{-}2) + 1] + 1 = T(n\text{-}2) + 2$ substituted $T(n\text{-}2)$ for $T(n\text{-}1)$

  $[T(n\text{-}3) + 1] + 2 = T(n\text{-}3) + 3$ substituted $T(n\text{-}3)$ for $T(n\text{-}2)$

  .. a pattern evolves

  $T(n) = 1+T(n-1)$

  $=2+T(n-2)$

  $=3+T(n-3)$

  $=\ldots.$

  $=i+T(n-i)$

  When n=0 T(0)=0,No multiplications

  When i=n, T(n) $=n+T(n-n)$

  $=n+0$

  $=n$ **$\underline{T(n) \in \Theta(n)}$**

Analyzing Recursive Algorithms-Example **2:-Tower of hanoi**

Step 1 − Move n-1 disks from **source** to **temp**

Step 2 − Move n^{th} disk from **source** to **dest**

Step 3 − Move n-1 disks from **temp** to **dest**

Algorithm Hanoi(n, source, dest, temp)

//Input: n :number of disks

//Output :All n disks on dest

If disk = 1

move disk from source to dest

Hanoi(n - 1, source, temp, dest) // Step 1

move nth disk from source to dest // Step 2

Hanoi(n - 1, temp, dest, source) // Step 3

Tower of hanoi

Analyzing Recursive Algorithms-Example **2:-Tower of hanoi**

1.Problem size is n, the number of discs

2. The basic operation is moving a disc from rod to another

3. Base case $M(1) = 1$

4. Recursive relation for moving n discs

$$M(n) = M(n\text{-}1) + 1 + M(n\text{-}1) = 2M(n\text{-}1) + 1$$

Analyzing Recursive Algorithms-Example **2: Tower of hanoi**

Solve using backward substitution

$M(n) = 2M(n\text{-}1) + 1$

$= 2[2M(n\text{-}2) + 1] + 1 = 2^2M(n\text{-}2) + 2+1$

$=2^2[2M(n\text{-}3) + 1] + 2+1$

$= 2^3M(n\text{-}3) + 2^2 + 2 + 1$

...

$M(n) = 2^iM(n\text{-}i) + \underbrace{2\text{^}i\text{-}1+2\text{^}i\text{-}2+\ldots.+2\text{^}3+2\text{^}2+2\text{^}1+2\text{^}0}_{\text{It's a GP with a=1,r=2,n=i}}$

$M(n) = 2^iM(n\text{-}i) + (2\text{^}i\text{-}1)/(2\text{-}1)$

$= 2^iM(n\text{-}i) + 2\text{^}i\text{-}1$

Analyzing Recursive Algorithms- Example **2:- Tower of hanoi**

- When i=n-1

$M(n) = 2^{n-1} M(n-(n-1)) + 2^{n-1} - 1$

$= 2^{n-1} M(1) + 2^{n-1} - 1$

$= 2^{n-1} + 2^{n-1} - 1$

$= 2 * 2^{n-1} - 1$

$= 2 * (2^{n} / 2) - 1$

$= 2^{n} - 1$

$M(n) \in O(2^n)$

- Time complexity is exponential
- More computattions even for smaller value of n
- Doesnt necessarily mean algorithm is poor
- Nature of the problem itself is computationally expensive.

Analyzing Recursive Algorithms- Example **3:Exercise**

ALGORITHM *BinRec(n)*

//Input: A positive decimal integer n

//Output: The number of binary digits in n's binary representation

if $n = 1$ **return** 1

else return $BinRec(n/2) + 1$

Let us set up a recurrence and an initial condition for the number of additions $A(n)$ made by the algorithm. The number of additions made in computing

$BinRec(n/2)$ is $A(n/2)$, plus one more addition is made by the algorithm to increase the returned value by 1. This leads to the recurrence

$A(n) = A(n/2) + 1$ for $n > 1$.

$A(1)=0$

Analyzing Recursive Algorithms- Example **3:Exercise**

Base condition A(1)= 0

$$A(n)=2A(n/2)+1$$

The presence of $n/2$ in the function's argument makes the method of backward substitutions stumble on values of n that are not powers of 2. Therefore, the standard approach to solving such a recurrence is to solve it only for $n = 2^k$ and then take advantage of the theorem called the ***smoothness rule,*** which claims that under very broad assumptions the order of growth observed for $n = 2^k$ gives a correct answer about the order of growth for all values of n.

Analyzing Recursive Algorithms-Example **3:Exercise**

$A(2^k) = A(2^{k-1}) + 1$ for $k > 0$,

$A(2^0) = 0$.

$$
\begin{aligned}
A(2^k) &= A(2^{k-1}) + 1 && \text{substitute } A(2^{k-1}) = A(2^{k-2}) + 1 \\
&= [A(2^{k-2}) + 1] + 1 = A(2^{k-2}) + 2 && \text{substitute } A(2^{k-2}) = A(2^{k-3}) + 1 \\
&= [A(2^{k-3}) + 1] + 2 = A(2^{k-3}) + 3 && \dots \\
&\dots \\
&= A(2^{k-i}) + i \\
&\dots \\
&= A(2^{k-k}) + k.
\end{aligned}
$$

Thus, we end up with

$$A(2^k) = A(1) + k = k,$$

or, after returning to the original variable $n = 2^k$ and hence $k = \log_2 n$,

$$A(n) = \log_2 n \in \Theta(\log n).$$

THANK YOU!

BITS Pilani
Hyderabad Campus