Design and Analysis of Algorithms

Dr. Shaik Naseera

Professor & HoD of CSE Department

JNTUA College of Engineering, Kalikiri

Objectives

- Introduction
- Performance Measurement
  - Space Complexity
  - Time Complexity
- Asymptotic Notations
- Solving Recurrence Relations

Algorithm

- Definition
  - An *algorithm* is a finite set of instructions that accomplishes a particular task.
  - An **algorithm** is a step-by-step procedure for solving a problem in a finite amount of time.
- Criteria
  - input
  - output
  - definiteness: clear and unambiguous
  - finiteness: terminate after a finite number of steps
  - effectiveness: instruction is basic enough to be carried out

Study of Algorithms

- How to device an algorithm : Study of various design strategies to device new and useful algorithms
- How to validate the algorithm : Proof of Correctness
- How to analyze the algorithm : Measuring the space and time complexity
- How to test a program : Checking the correctness of the program and measuring the space and time it takes to compute the results.

Measurements

- Performance Analysis (machine independent)
  - space complexity: storage requirement
  - time complexity: computing time
- Performance Measurement (machine dependent)
- Performance analysis is also called as Priori Estimates
- Performance measurement is also called as Posteriori Testing

Space Complexity
$S(P)=C+S_P(I)$

- Fixed Space Requirements (C)
  Independent of the characteristics of the inputs and outputs
  - instruction space
  - space for simple variables, fixed-size structured variable, constants
- Variable Space Requirements ($S_P(I)$)
  depend on the instance characteristic I
  - number, size, values of inputs and outputs associated with I
  - recursive stack space, formal parameters, return address

Algorithm 1: Simple arithmetic function

```
Algorithm abc(a, b, c)
{
   return a + b + b * c + (a + b - c) / (a + b) + 4.00;
 }
```

S(abc)=3+0=3

$S_{abc}(I) = 0$

Note: 4.00 does not require memory. If any element is declared as a constant, it requires memory. Ex:- const int x;

Algorithm 2 : Iterative function for summing a list of numbers

```
 Algorithm Isum(list, n)
{
  sum = 0;
  for (i = 0; i<n; i++)
     sum += list [i];
  return sum;
}
```

$S_{sum}(I) \geq (n+3)$

Recall: pass the address of the first element of the array & pass by value

Algorithm 3 : Recursive function for summing a list of numbers

```
Algorithm rsum(list, n)
{
  if (n ≤ 0) return 0.0
   else return rsum(list, n-1) + list[n];
 }
```

$S_{sum}(I)=S_{sum}(n)>=3(n+1)$

Assumptions:

*__Figure 1.1:__ Space needed for one recursive call of **Algorithm 3:**

Type	Name	Number of Units
parameter: float parameter: integer return address:(used internally)	list [ ] n	1 1 1
TOTAL per recursive call		3

Time Complexity

•T**ime complexity** is the computational complexity that describes the amount of time it takes to run an algorithm.

•Time complexity is commonly estimated by counting the number of elementary operations performed by the algorithm

•We Assume that each elementary operation takes a fixed amount of time to perform.

•Thus, the amount of time taken is the total number of elementary operations performed by the algorithm.

Time Complexity

$$T(P)=C+T_P(I)$$

- Compile time (C)
  independent of instance characteristics
- run (execution) time T_P

Step Table Method

***Figure 1.2:** Step count table for Program with iterative function

Statement	s/e	Frequency	Total steps
Algorithm sum(list, n)	0	0	0
{	0	0	0
s := 0;	1	1	1
i:=0;	1	1	1
for i := 1 to n do	1	n+1	n+1
s:= s+ list[i];	1	n	n
return s;	1	1	1
}	0	0	0
Total			2n+4

Recursive Function to sum of a list of numbers

Statement	s/e	Frequency		Total steps	
		n=0	n>0	n=0	n>0
Algorithm RSum(a, n)	0	-	-	0	0
{	0	0	0	0	0
if (n≤0) then	1	1	1	1	1
return 0.0;	1	1	0	1	0
else return RSum(a, n-1)+a[n];	1+x	0	1	0	1+x
}	0	-	-	0	0
Total				2	2+x
				x=t_{Rsum}(n-1)	

$$tRSum(n) = \begin{cases} 2 & if\ n \leq 0 \\ 2 + x & if\ n > 0 \end{cases}$$

Matrix Addition

Statement	s/e	Frequency	Total steps

Algorithm add (a, b, c, m,n)
{
 	
for i:= 1 to m do
for j:= 1 to n do
c[i][j] := a[i][j] + b[i][j];
}
	0
	0
	
	1
	1
	1
	0
		0
		0
		
		m+1
		m • (n+1)
		m • n
		0
			0
			0
			
			m+1
			mn+m
			mn
			0

Total			2mn+2m+1

Printing of a Matrix

Statement	s/e	Frequency	Total steps
Algorithm print_matrix(a, r, c)	0	0	0
{	0	0	0
for i:= 1 to r do	1	r+1	r + 1
{ for j:= 1 to c do	1	r•(c+1)	rc + r
Print(a[i][j]);	1	r•c	rc
Print(“\n”);	1	r	r
}	0	0	0
}	0	0	0
Total			2rc+3r+1

Matrix multiplication function

Statement	s/e	Frequency	Total steps
Algorithm mult(a, b, c, M)	0	0	0
{	0	0	0
for i:= 1 to M do	1	M+1	M + 1
for j:= 1 to M do	1	M•(M+1)	M^2 +M
{ c[i][j] = 0;	1	M. M	M^2
for i:= k to M do	1	M.M.(M+1)	M^3 +M^2
c[i][j] := c[i][j] + a[i][k] * b[k][j];	0	M.M.M	M^3
}	0	0	0
}	0	0	0
Total			$2M^3+3M^2+2M+1$

Asymptotic Notations

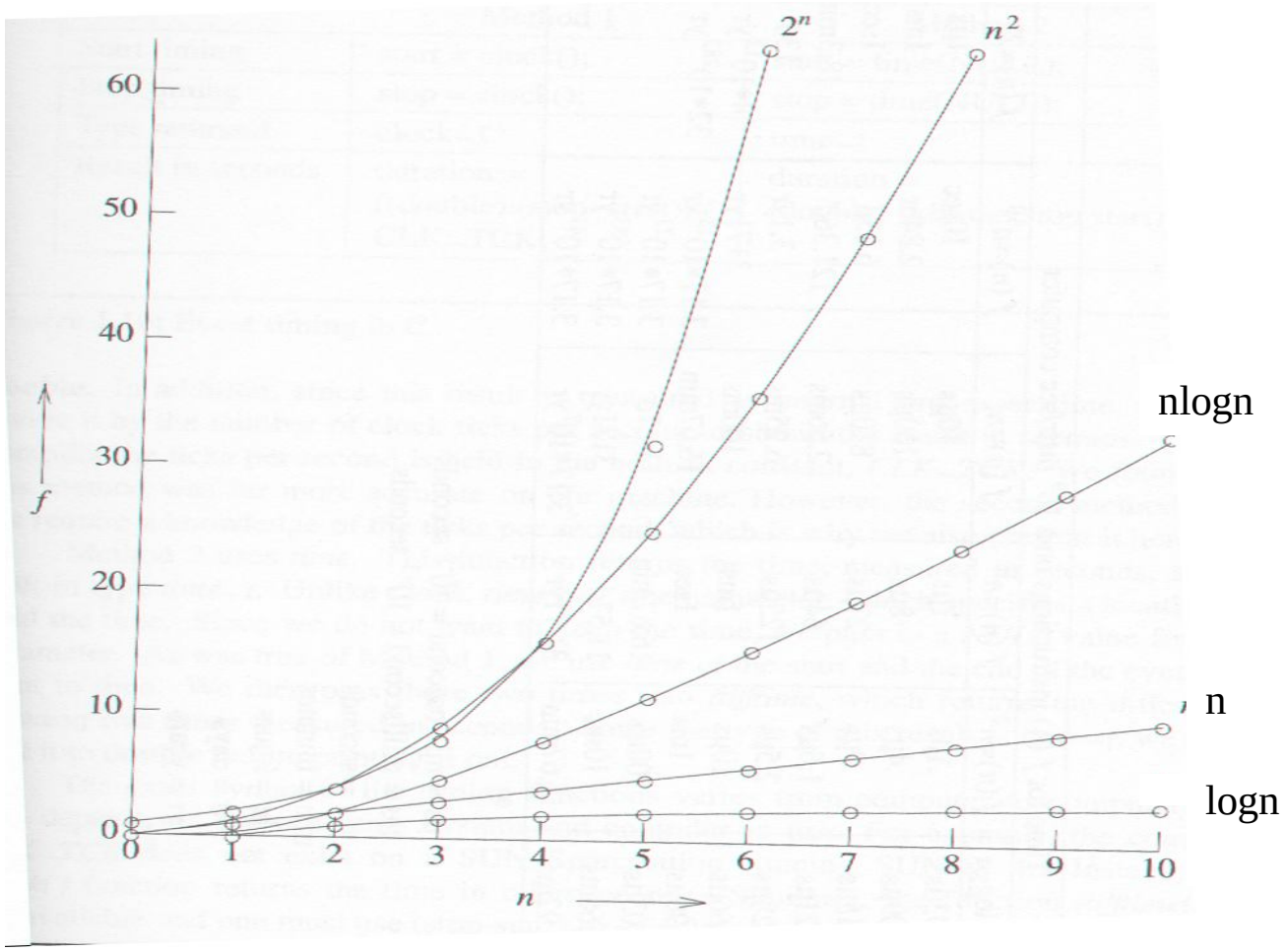
- Big Oh (O)
- Big Omega (Ω)
- Theta (Θ) Notation
- Small oh (o)
- Small Omega (ω)

Big Oh Notation (O)

- Definition
  f(n) = O(g(n)) iff there exist positive constants c and n_0 such that $f(n) \leq cg(n)$ for all n, $n \geq n_0$.
- Examples
  - $3n+2=O(n)$ /* $3n+2 \leq 4n$ for $n \geq 2$ */
  - $3n+3=O(n)$ /* $3n+3 \leq 4n$ for $n \geq 3$ */
  - $100n+6=O(n)$ /* $100n+6 \leq 101n$ for $n \geq 6$ */
  - $10n^2+4n+2=O(n^2)$ /* $10n^2+4n+2 \leq 11n^2$ for $n \geq 5$ */
  - $6*2^n+n^2=O(2^n)$ /* $6*2^n+n^2 \leq 7*2^n$ for $n \geq 4$ */

- O(1): constant
- O(n): linear
- O(n^2): quadratic
- O(n^3): cubic
- O(2^n): exponential
- O(logn)
- O(nlogn)

Figure 1: Plot of function values



Big Omega (Ω) Notation

- The function $f(n)=\Omega(g(n))$ iff there exists positive constants c and n_0 such that $f(n) \geq c\, g(n)$ for all $n \geq n_0$.

Examples

$3n+2= \Omega(n)$ // $3n+2 \geq 3n$ for $n \geq 1$

$3n+3= \Omega(n)$ // $3n+3 \geq 3n$ for $n \geq 1$

$100n+6= \Omega(n)$ // $100n+6 \geq 100n$ for $n \geq 1$

$10n^2+4n+2= \Omega(n^2)$ // $10n^2+4n+2 \leq n^2$ for $n \geq 1$

$6*2^n+n^2= \Omega(2^n)$ // $6*2^n+n^2 \geq 6*2^n$ for $n \geq 1$

Theta Notation (Θ)

- The function f(n)= Θ (g(n)) iff there exists positive constants c_1, c_2 and n_0 such that $c_1 g(n) \le f(n) \le c_2 g(n)$ for all n, $n \ge n_0$.

Examples:

- $3n+2=\Theta(n)$ //$3n+2 \ge 3n$ for all $n \ge 2$ and $3n+2 \le 4n$ for all $n \ge 2$, so $3n \le 3n+2 \le 4n$ for $n \ge 2$, $c_1=3$, $c_2=4$, $n_0=2$
- $3n+3= \Theta(n)$ /* $3n \le 3n+3 \le 4n$ for $n \ge 2$ */
- $100n+6= \Theta(n)$ /* $100n \le 100n+6 \le 101n$ for $n \ge 6$ */
- $10n^2+4n+2= \Theta(n^2)$ /*$10n^2 \le 10n^2+4n+2 \le 11n^2$ for $n \ge 5$ */
- $6*2^n+n^2= \Theta(2^n)$ /* $2^n \le 6*2^n+n^2 \le 7\ 2^n$ for $n \ge 4$*/

Asymptotic notation

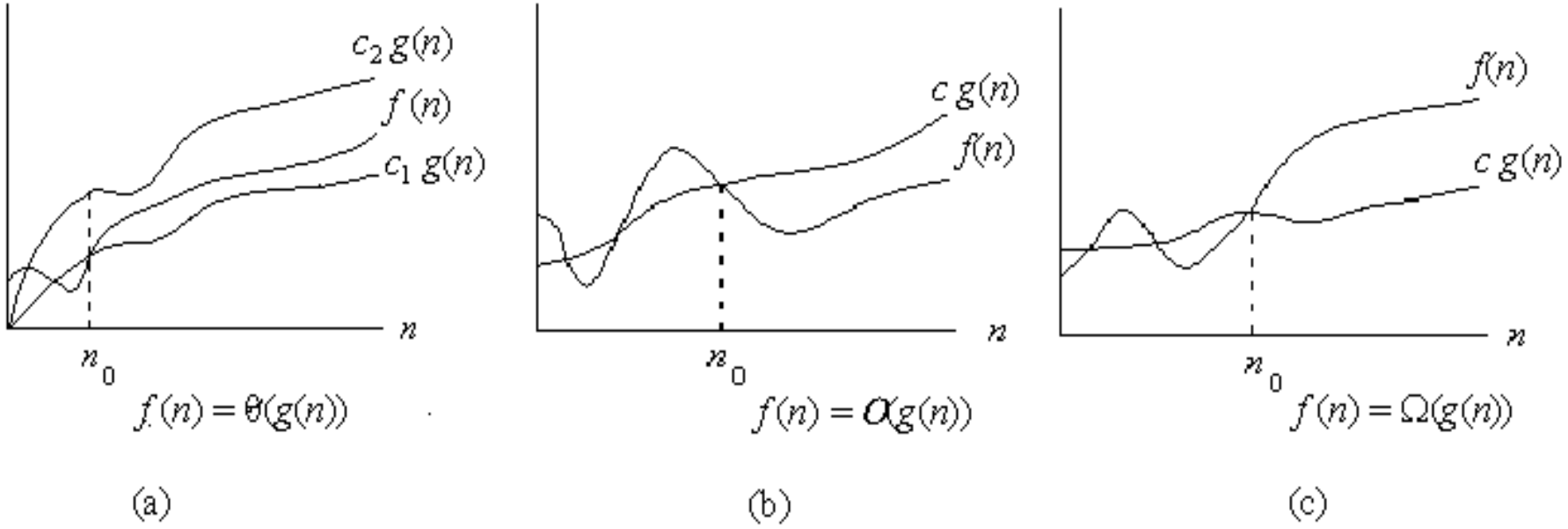


Figure 2: Graphic examples of Θ, O, and Ω.

Little oh (o) Notation

– The function f(n)= o (g(n)) iff

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

Examples:

- $3n+2=o(n^2)$ // $\lim_{n\to\infty}\frac{3n+2}{n^2}=\lim_{n\to\infty}(\frac{3n}{n^2}+\frac{2}{n^2})=0$
  - Little oh is a method of expressing the upper bound on the growth rate of the algorithms running time which may or may not be asymptotically tight therefore little oh is also called a loose upper bound.
  - We use little oh notation to denote upper bound that is asymptotically not tight.

Little Omega (ω) Notation

– The function f(n)= ω(g(n)) iff

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

Examples:

- 3n+2= ω (1) //Here f(n)=3n+2 and g(n)=1 so $\lim_{n\to\infty} \frac{1}{3n+2} = 0$
  - Little ω is a method of expressing the lower bound on the growth rate of the algorithms running time which may or may not be asymptotically tight therefore little ω is also called a loose lower bound.
  - We use little ω notation to denote lower bound that is asymptotically not tight.

Solving of Recurrence Relation

- solving recurrence relation.pdf

Previous Gate Questions

Consider the following three claims

1. $(n + k)^m = \Theta(n^m)$, where k and m are constants
2. $2^{n+1} = O(2^n)$
3. $2^{2n+1} = O(2^n)$

Which of these claims are correct ?

(A) 1 and 2

(B) 1 and 3

(C) 2 and 3

(D) 1, 2, and 3

Answer: (A)

Explanation: $(n + k)^m$ and $\Theta(n^m)$ are asymptotically same as theta notation can always be written by taking the leading order term in a polynomial expression.

Previous Gate Questions

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

=[T(n-2)+log(n-1)]+log n

=[T(n-3)+log(n-2)]+log(n-1)+log n

= .

.

.

=T(n-k)+log(n-(k-1))+.....+log n

=T(1)+log 2+log 3+….+log n

=T(1)+log(1)+log 2+....+log n //log 1 is 0

=log(1.2.3....n)

=log(n!) (note: n! upper bound is n^n)

=O(nlogn)

1) $T(n) = 2T(\frac{n}{2}) + n.$

$= 2\left[2 \cdot T(\frac{n}{4}) + \frac{n}{2}\right] + n.$

$= 2^2 T(\frac{n}{4}) + 2n$

$= 2^2 \left[2T(\frac{n}{8}) + (\frac{n}{4})\right] + 2n.$

$= 2^3 T(\frac{n}{8}) + 4(\frac{n}{4}) + 2n$

$= 2^3 T(\frac{n}{2^3}) + 3n.$

$\vdots$

$= 2^k T(\frac{n}{2^k}) + kn$

$n = 2^k \Rightarrow k = \log_2 n$

$= 2^k T(\frac{n}{n}) + k \cdot n$

$= n \cdot T(1) + \log_2 n \cdot n$

$= n \cdot c + n \cdot \log n$

$\leq c \cdot n \log n$

$= O(n \log n).$

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

$= T(n-2) + (n-1) + n.$

$= T(n-3) + (n-2) + (n-1) + n.$

$\vdots$

$= T(n-k) + (n-(k-1)) + \cdots + n.$

Here $n-k = 1$

$\Rightarrow k = n-1$

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

$= T(1) + (n-n+2) + \cdots + n.$

$= 1 + 2 + \cdots + n.$

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

$\leq c \cdot n^2$

$= O(n^2)$

$=.$

3) $T(n) = 4T\left(\frac{n}{2}\right) + n^2$

$= 4\left[4 \cdot T\left(\frac{n}{4}\right) + \left(\frac{n}{2}\right)^2\right] + n^2$

$= 4^2 T\left(\frac{n}{4}\right) + n^2 + n^2$

$= 4^2 T\left(\frac{n}{4}\right) + 2n^2$

$= 4^2\left[4T\left(\frac{n}{8}\right) + \left(\frac{n}{4}\right)^2\right] + 2n^2$

$= 4^3 T\left(\frac{n}{2^3}\right) + 3n^2$

$\vdots$

$= 4^k T\left(\frac{n}{2^k}\right) + k \cdot n^2$

Here $2^k = n \Rightarrow k = \log_2 n$

$= 4^{\log_2 n} T(1) + \log n \cdot n^2$

$= n^{\log_2 4} T(1) + n^2 \log n$

$= n^2 T(1) + n^2 \log n$

$\leq c \cdot n^2 \log n$

$= O(n^2 \log n)$

$\Rightarrow$

4) $T(n) = 8T(n/2) + n^2$

$= 8\left[8T\left(\frac{n}{4}\right) + \left(\frac{n}{2}\right)^2\right] + n^2$

$= 8^2 T\left(\frac{n}{4}\right) + 2n^2 + n^2$

$= 8^2\left[8T\left(\frac{n}{8}\right) + \left(\frac{n}{4}\right)^2\right] + 2n^2 + n^2$

$= 8^3 T\left(\frac{n}{2^3}\right) + 4n^2 + 2n^2 + n^2$

$= 8^3 T\left(\frac{n}{2^3}\right) + 7n^2$

$\vdots$

$= 8^k T\left(\frac{n}{2^k}\right) + (2^k - 1)n^2$

substitute $n = 2^k \Rightarrow k = \log_2 n$

$= 8^{\log n} T(1) + (n-1)n^2$

$= 2^{3\log_2 n} T(1) + n \cdot n^2 - n^2$

$= 2^{\log_2 n^3} T(1) + n^3 - n^2$

$= (n^3)^{\log_2 2} T(1) + n^3 - n^2$

$= n^3 \cdot 1 + n^3 - n^2$

$= 2n^3 - n^2$

$\leq c \cdot n^3$

$= O(n^3)$

5) $T(n) = T(n-2) + n^2$

$= T(n-4) + (n-2)^2 + n^2$

$= T(n-6) + (n-4)^2 + (n-2)^2 + n^2$

$= T(n-6) + (n-4)^2 + (n-2)^2 + (n-0)^2$

$2 \cdot k$

$k = 0, 1, 2, 3 \cdots$

$= T(n-2k) + (n-2k)^2 + \cdots + n^2$

$n - 2k = 1$

$\Rightarrow k = \frac{n-1}{2}$

$= T(1) + \left(n - 2 \cdot \left(\frac{n-1}{2}\right)\right)^2 + \cdots + n^2$

$= T(1) + 1^2 + 2^2 + \cdots + n^2$

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

$\leq c \cdot n^3$

$= O(n^3)$

5.

$$T(n) = 2 * T(\sqrt{n}) + \log n \text{ and } T(1) = 1$$

let $n = 2^m$

$\Rightarrow T(2^m) = 2 * T(\sqrt{2^m}) + \log(2^m)$

$\Rightarrow T(2^m) = 2 * T(2^{m/2}) + m$

let $S(m) = T(2^m)$

$\Rightarrow S(m) = 2 * S(m/2) + m$

$\Rightarrow S(m) = 2 * (2 * S(m/4) + m/2) + m$

$\Rightarrow S(m) = 2^2 * S(m/2^2) + m + m$

By substituting further,

$\Rightarrow S(m) = 2^k * S(m/2^k) + m + m + ... + m + m$

let $m = 2^k \Rightarrow S(m/2^k) = S(1) = T(2) = 2$

$\Rightarrow S(m) = 2 + m(k-1) + m$

$\Rightarrow S(m) = 2 + m.k$

$\Rightarrow S(m) = 2 + m \log m$

$\Rightarrow S(m) = O(m \log m)$

$\Rightarrow S(m) = T(2^m) = T(n) = O(\log n \log \log n)$