innovate achieve lead

BITS Pilani
Hyderabad Campus

S2 Characterizing Time Complexity, Asymptotic Notation, Recurrence Relation, Master Theorem

Content of S2

1. Characterizing Time Complexity
    1. Use of Asymptotic Notation
    2. Big-Oh, Big-Omega, Theta Notations
2. Analyzing Recursive Algorithm
    1. Recurrence Relation
    2. Runtime of Recursive Algorithm
    3. Master Theorem

Analyzing Algorithm

→Used to mean the prediction of **resource** consumption

→But, **what** is the **resource**?

Analyzing Algorithm

→Used to mean the prediction of **resource** consumption

→But, **what** is the **resource**?

Primarily i) **memory**, ii) communication **bandwidth**, iii) computer **hardware**

But, most often we are interested in **computational time**

Which computer should be taken as a base case or standard?

- **Random Access Machine (RAM)** model of a computer

Random Access Machine Model

Instructions in RAM that takes **one unit of time**

1) **Arithmetic:** Add, Sub, Mul, Div, Rem, Floor, Ceil
2) **Data movement:** Load, Store, Copy
3) **Control:** Subroutine call, Return, Conditional and Unconditional Branch

Data Types in RAM (**fixed size**, like 8 bit or 16 bit or 32 bit)

1) Integer
2) Float

RAM model: What is not an instruction?

1) "Sort" – even if, in some computer, sort can be done in one instruction
2) "exponentiation" – x^y

→ there may be many algorithms to compute x^y, but it is not a single instruction if y is a variable or a large integer

→ But, x^k is a single instruction, where **k is a constant and very small**

RAM model: memory hierarchy

We do not consider any complex memory hierarchy, like having cache or virtual memory.

RAM model: memory hierarchy

We do not consider any complex memory hierarchy, like having cache or virtual memory.

Simplicity of RAM model

→ Though simple, but an excellent predictor of performance on actual computer

→Though simple, exact prediction can still be challenging

→Often, it would require tools like combinatorics, probability theory, algebraic dexterity and the ability to identify the most significant terms in a formula

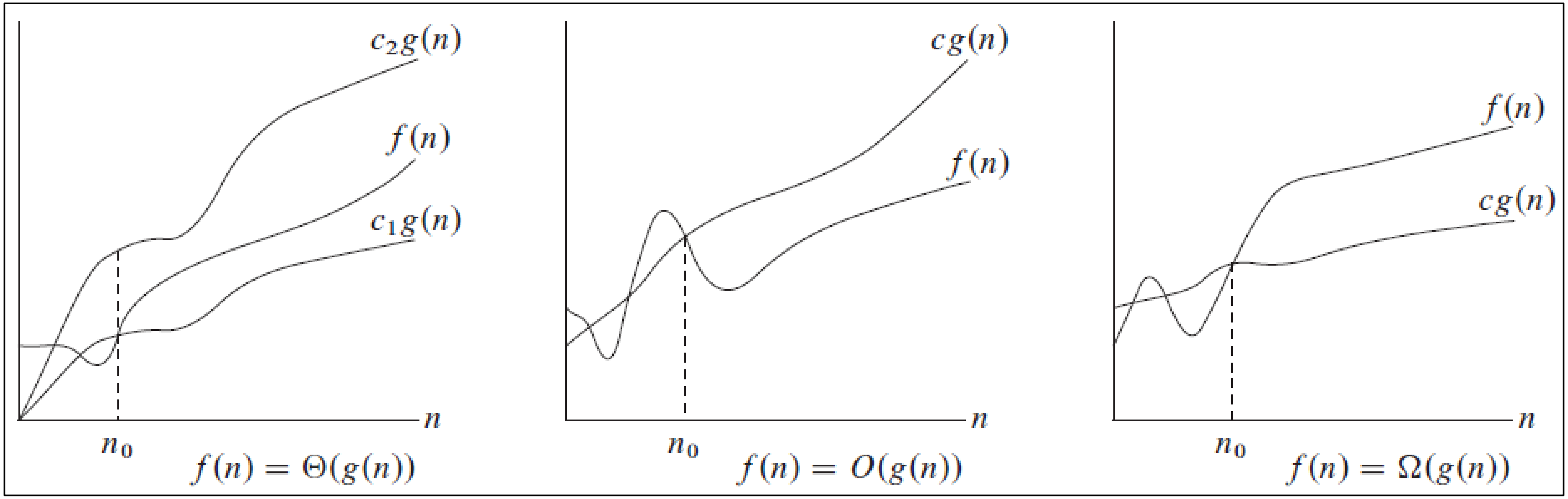
Content of S2

1. **Characterizing Time Complexity**
    1. **Use of Asymptotic Notation**
    2. Big-Oh, Big-Omega, Theta Notations
2. Analyzing Recursive Algorithm
    1. Recurrence Relation
    2. Runtime of Recursive Algorithm
    3. Master Theorem

Characterizing Time Complexity

Big-Oh Notation, Omega and Theta Notations:

- Asymptotic notation primarily describes the running times of algorithms, i.e., time complexity



Content of S2

1. **Characterizing Time Complexity**
    1. Use of Asymptotic Notation
    2. **Big-Oh, Big-Omega, Theta Notations**
2. Analyzing Recursive Algorithm
    1. Recurrence Relation
    2. Runtime of Recursive Algorithm
    3. Master Theorem

Characterizing Time Complexity

Big-Oh Notation: f(n) = O(g(n)).

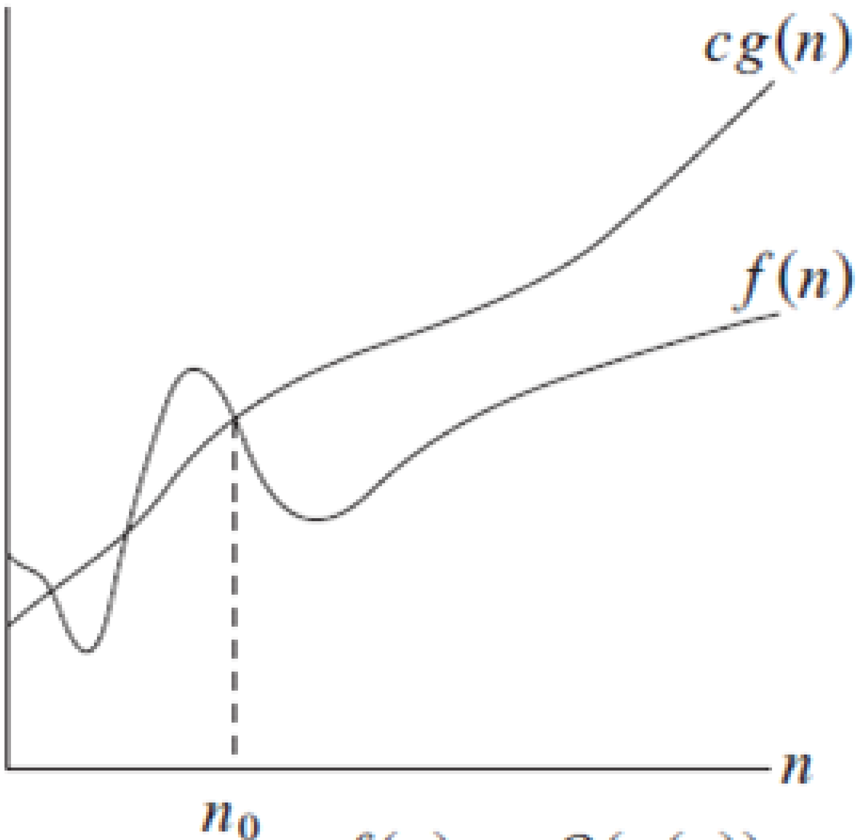
- g(n) is an asymptotically upper bound for f(n).

$$O(g(n)) = \{ f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \le f(n) \le c g(n) \text{ for all } n \ge n_0 \} .$$

- f(n) = Θ(g(n)) implies that f(n) = O(g(n)),

  i.e., Θ(g(n)) ϵ O(g(n))

Graphical representation of Big-O



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

Example: Time Complexity Big-O

Ex-1 $f(n) = 2n+2$

$2n+2 \leq \underline{10n}$, where $n \geq 1$

Here, $c = 10$, $\mathbf{g(n) = n}$

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

Ex-2 $f(n) = 2n+2$

$2n+2 \leq \underline{10n^2}$, where $n \geq 1$

Here, $c = 10$, $\mathbf{g(n) = n^2}$

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

Ex-3 $f(n) = 2n+2$

$2n+2 \leq \underline{10n^3}$, where $n \geq 1$

Here, $c = 10$, $g(n) = n^3$

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

Ex-4 $f(n) = 2n^2+5$

$2n^2+5 \leq \underline{2n^2+5n^2} = 7n^2$, where $n \geq 1$

Here, $c = 7$, $g(n) = n^2$

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

$$O(g(n)) = \{f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0\}.$$

Example: Time Complexity Big-O

Ex-5 $f(n) = 7n-2$

Here, $c = 7$, $n >= 1$

→ $7n - 2 \leq cn$, $\mathbf{g(n) = n}$

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

Ex-6 $f(n) = 20n^3 + 10n\log n + 5$

Here, $c = 35$, $\mathbf{g(n) = n^3}$

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

Ex-7 $f(n) = 3\log n + \log\log n$

Here, $c = 4$, $\mathbf{g(n) = \log n}$

$f(n) = O(g(n)) = O(\log n)$.

Ex-8 $f(n) = \mathbf{2^{100}}$

Here, $c = \mathbf{2^{100}}$, $\mathbf{g(n) = 1}$

$f(n) = O(g(n)) = O(1)$.

Ex-9 $f(n) = 5/n$

Here, $c = \mathbf{5}$, $\mathbf{g(n) =} 1/n$

$f(n) = O(g(n)) = O(1/n)$.

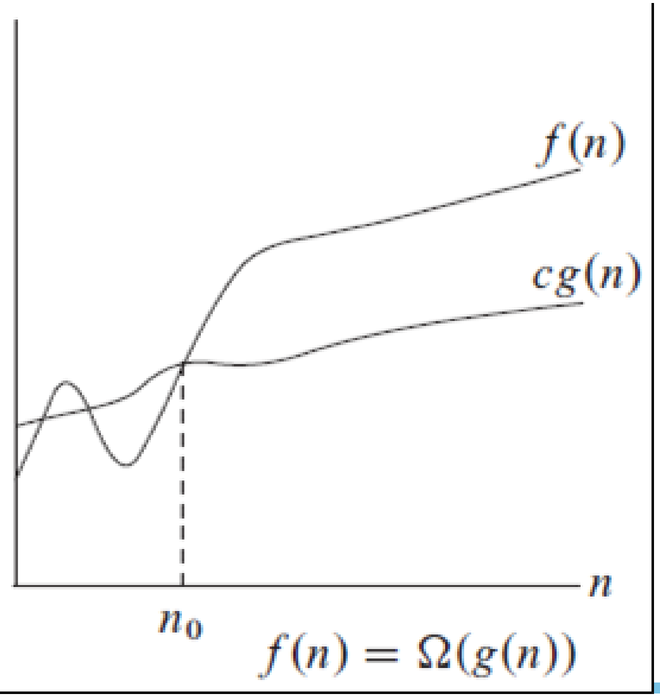
$$O(g(n)) = \{f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0\}.$$

Time Complexity: Big-Omega

Omega Notation: f(n) = Ω(g(n)).

- g(n) is an asymptotically lower bound for f(n).

$$\Omega(g(n)) = \{ f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \le c g(n) \le f(n) \text{ for all } n \ge n_0 \} .$$



Example: Omega Notation

Ex-1 $f(n) = 2n+2$

$2n+2 \geq \underline{2n}$, where $n \geq 1$

Here, $c = 2$, $\mathbf{g(n) = n}$

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

Ex-2 $f(n) = 2n+2$

$2n+2 \geq \underline{\sqrt{n}}$, where $n \geq 1$

Here, $c = 1$, $\mathbf{g(n)} = \underline{\sqrt{n}}$

$f(n) = \Omega(g(n)) = \Omega(\underline{\sqrt{n}})$

Ex-3 $f(n) = 2n+2$

$2n+2 \geq \underline{\log n}$, where $n \geq 1$

Here, $c = 1$, $\mathbf{g(n) = \log n}$

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

Ex-4 $f(n) = 2n^2+5$

$2n^2+5 \geq \underline{2n^2}$, where $n \geq 1$

Here, $c = 2$, $\mathbf{g(n) = n^2}$

$f(n) = \Omega(g(n)) = \Omega(n^2)$.

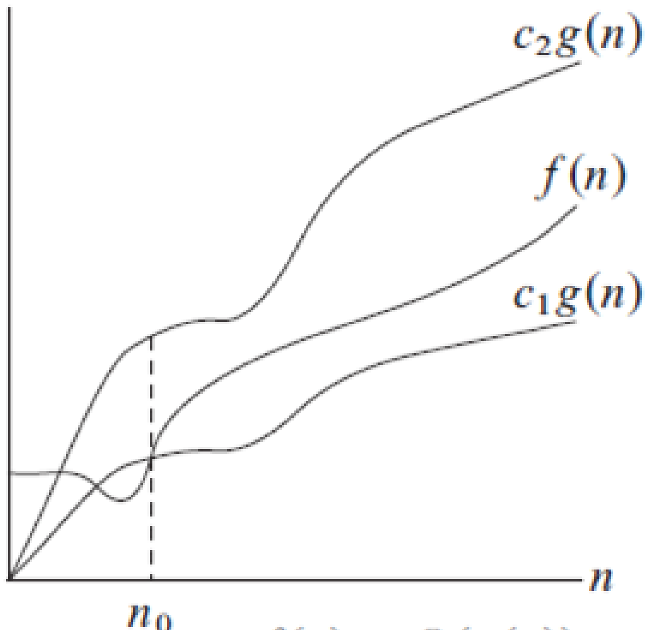
$$\Omega(g(n)) = \{f(n) : \text{ there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \leq cg(n) \leq f(n) \text{ for all } n \geq n_0\} .$$

Characterizing Run Time

Theta Notation: f(n) = Θ(g(n)).

- g(n) is an asymptotically tight bound for f(n).

$$\Theta(g(n)) = \{ f(n) : \text{there exist positive constants } c_1, c_2, \text{ and } n_0 \text{ such that } 0 \le c_1 g(n) \le f(n) \le c_2 g(n) \text{ for all } n \ge n_0 \}$$.[1]



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

Example: Theta Notation

Ex-1 f(n) = $\frac{n^2}{2} - \frac{n}{2}$

$\frac{n^2}{4} \leq \frac{n^2}{2} - \frac{n}{2} \leq \frac{n^2}{2}$, where **n≥2**

$c_1 = \frac{1}{4}$, $c_2 = \frac{1}{2}$, **g(n) = n^2**

f(n) = Θ(g(n)) = Θ(n^2).

Ex-2 f(n) =$6n^3$ **{≠ Θ(n^2), why?}**

$c_1 n^2 \leq 6n^3 \leq c_2 n^2$, where n≥1

There exists no c_2 that implies $6n^3 \leq c_2 n^2$

$$\Theta(g(n)) = \{ f(n) : \text{there exist positive constants } c_1, c_2, \text{ and } n_0 \text{ such that } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ for all } n \geq n_0 \}$$.[1]

Time Complexity: Little-Oh, Little-omega

o-notation:

$$o(g(n)) = \{f(n) : \text{for any positive constant } c > 0, \text{ there exists a constant } n_0 > 0 \text{ such that } 0 \leq f(n) < cg(n) \text{ for all } n \geq n_0\}.$$

Time Complexity: Little-Oh, Little-omega

ω-notation:

$$\omega(g(n)) = \{ f(n) : \text{ for any positive constant } c > 0 \text{, there exists a constant } n_0 > 0 \text{ such that } 0 \le cg(n) < f(n) \text{ for all } n \ge n_0 \} .$$

Notation Summary

Because these properties hold for asymptotic notations, we can draw an analogy between the asymptotic comparison of two functions f and g and the comparison of two real numbers a and b:

$$f(n) = O(g(n)) \quad \text{is like} \quad a \leq b ,$$
$$f(n) = \Omega(g(n)) \quad \text{is like} \quad a \geq b ,$$
$$f(n) = \Theta(g(n)) \quad \text{is like} \quad a = b ,$$
$$f(n) = o(g(n)) \quad \text{is like} \quad a < b ,$$
$$f(n) = \omega(g(n)) \quad \text{is like} \quad a > b .$$

Properties of Time Complexity

- Comparison

Transitivity:

$$f(n) = \Theta(g(n)) \text{ and } g(n) = \Theta(h(n)) \quad \text{imply} \quad f(n) = \Theta(h(n)),$$
$$f(n) = O(g(n)) \text{ and } g(n) = O(h(n)) \quad \text{imply} \quad f(n) = O(h(n)),$$
$$f(n) = \Omega(g(n)) \text{ and } g(n) = \Omega(h(n)) \quad \text{imply} \quad f(n) = \Omega(h(n)),$$
$$f(n) = o(g(n)) \text{ and } g(n) = o(h(n)) \quad \text{imply} \quad f(n) = o(h(n)),$$
$$f(n) = \omega(g(n)) \text{ and } g(n) = \omega(h(n)) \quad \text{imply} \quad f(n) = \omega(h(n)).$$

Reflexivity:

$$f(n) = \Theta(f(n)),$$
$$f(n) = O(f(n)),$$
$$f(n) = \Omega(f(n)).$$

Summary of Properties

- Comparison

Symmetry:

$$f(n) = \Theta(g(n)) \text{ if and only if } g(n) = \Theta(f(n)) .$$

Transpose symmetry:

$$f(n) = O(g(n)) \text{ if and only if } g(n) = \Omega(f(n)) ,$$

$$f(n) = o(g(n)) \text{ if and only if } g(n) = \omega(f(n)) .$$

Content of S2

1. Characterizing Time Complexity
    1. Use of Asymptotic Notation
    2. Big-Oh, Big-Omega, Theta Notations
2. **Analyzing Recursive Algorithm**
    1. **Runtime of Recursive Algorithm**
    2. Recurrence Relation
    3. Master Theorem

Analyzing Recursive Algorithms

innovate achieve lead

Algorithm recursiveMax(A, n):
Input: An array A storing $n \geq 1$ integers.
Output: The maximum element in A.

if $n = 1$ **then**
  return $A[0]$
return $\max\{\text{recursiveMax}(A, n-1), A[n-1]\}$

Analyzing Recursive Algorithms

Algorithm recursiveMax(A, n):

Input: An array A storing $n \geq 1$ integers.

Output: The maximum element in A.

if $n = 1$ **then**

  return $A[0]$

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

$$T(n) = \begin{cases} 2, if\ n = 1 \\ T(n-1) + 4, otherwise \end{cases}$$

Analyzing Recursive Algorithms

innovate achieve lead

Algorithm recursiveMax(A, n):

  Input: An array A storing $n \geq 1$ integers.

  Output: The maximum element in A.

1 | **if** $n = 1$ **then**

1 +1 | **return** $A[0]$

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

1 + 1 + T(n-1) + 1 1 + 1

$$T(n) = \begin{cases} 3, if\ n = 1 \\ T(n-1) + 6, otherwise \end{cases}$$

Content of S2

1. Characterizing Time Complexity
    1. Use of Asymptotic Notation
    2. Big-Oh, Big-Omega, Theta Notations
2. **Analyzing Recursive Algorithm**
    1. Runtime of Recursive Algorithm
    2. **Recurrence Relation**
    3. Master Theorem

Recurrence Relation

Def^n: A recurrence relation is an equation that defines a sequence based on a rule that gives the next term as a function of the previous term(s).

Mathematically, $x_{n+1} = f(x_n)$: a simple recurrence relation, also called as first order recurrence relation.

Recurrence Relation

Def^n: A recurrence relation is an equation that defines a sequence based on a rule that gives the next term as a function of the previous term(s).

Mathematically, $x_{n+1} = f(x_n)$: a simple recurrence relation, also called as first order recurrence relation.

Example of first order recurrence relation:

1) $x_{n+1}=2-x_{n/2}$

Recurrence Relation

Def^n: A recurrence relation is an equation that defines a sequence based on a rule that gives the next term as a function of the previous term(s).

Mathematically, $x_{n+1} = f(x_n)$: a simple recurrence relation, also called as first order recurrence relation.

Example of first order recurrence relation:

1) $x_{n+1}=2-x_{n/2}$

A second order recurrence relation depends just on x_n and x_{n-1} and is of the form $x_{n+1}=f(x_n,x_{n-1})$

Example: $x_{n+1}=x_n+x_{n-1}$

Content of S2

1. Characterizing Time Complexity
    1. Use of Asymptotic Notation
    2. Big-Oh, Big-Omega, Theta Notations
2. **Analyzing Recursive Algorithm**
    1. Runtime of Recursive Algorithm
    2. Recurrence Relation
    3. **Master Theorem**

Analyzing Recursive Algorithms

Solving recurrence equations

1. Master Theorem for Dividing Functions

$$T(n) = aT(\frac{n}{b}) + g(n)$$

where g(n) is O($n^k \, log^p$ n), where p and k are integers.

a) $a < b^k$: if $p < 0$, then $T(n) = O(n^k)$

Analyzing Recursive Algorithms

Solving recurrence equations

1. Master Theorem for Dividing Functions

$$T(n) = aT(\frac{n}{b}) + g(n)$$

where g(n) is $O(n^k \log^p n)$, where p and k are integers.

a) $a < b^k$: if $p < 0$, then $T(n) = O(n^k)$

if $p \geq 0$, then $T(n) = O(n^k \log^p n)$

Analyzing Recursive Algorithms

Solving recurrence equations

1. Master Theorem for Dividing Functions

$$T(n) = aT(\frac{n}{b}) + g(n)$$

where g(n) is $O(n^k \log^p n)$

a) $a < b^k$: if $p < 0$, then $T(n) = O(n^k)$

if $p \geq 0$, then $T(n) = O(n^k \log^p n)$

b) $a = b^k$: if $p > -1$, then $T(n) = O(n^k \log^{p+1} n)$

if $p = -1$, then $T(n) = O(n^k \log \log n)$

if $p < -1$, then $T(n) = O(n^k)$

Analyzing Recursive Algorithms

Solving recurrence equations

1. Master Theorem for Dividing Functions

$$T(n) = aT(\frac{n}{b}) + g(n)$$

where g(n) is $O(n^k \, log^p \, n)$

a) $a < b^k$: if $p < 0$, then $T(n) = O(n^k)$

if $p \geq 0$, then $T(n) = O(n^k \, log^p \, n)$

b) $a = b^k$: if $p > -1$, then $T(n) = O(n^k \, log^{p+1} \, n)$

if $p = -1$, then $T(n) = O(n^k \log \log n)$

if $p < -1$, then $T(n) = O(n^k)$

c) $a > b^k$: $T(n) = O(n^{log_b a})$

Solution using Master Theorem

g(n) is O($n^k \log^p n$)

Ex-1 T(n) = 4T($\frac{n}{2}$)+n,

a = 4, b = 2, k = 1, p = 0.

a = 4, b^k = 2 ➔ a > b^k

T(n) = O($n^{\log_2 4}$) = O(n^2)

Ex-2 T(n) = 8T($\frac{n}{2}$)+n^2,

a = 8, b = 2, k = 2, p = 0.

a = 8, b^k = 4 ➔ a > b^k

T(n) = O($n^{\log_2 8}$) = O(n^3)

Ex-3 T(n) = 8T($\frac{n}{2}$)+n log n,

a = 8, b = 2, k = 1, p = 1.

a = 8, b^k = 2 ➔ a > b^k

T(n) = O($n^{\log_2 8}$) = O(n^3)

Solution using Master Theorem

Ex-4 $T(n) = 2T(\frac{n}{2})+n$,

$a = 2, b = 2, k = 1, p = 0.$

$a = 2, b^k = 2 \rightarrow a = b^k$

$T(n) = O(n^k \ log^{p+1} n)$

$= O(n \log n)$

Ex-5 $T(n) = 4T(\frac{n}{2})+n^2$,

$a = 4, b = 2, k = 2, p = 0.$

$a = 4, b^k = 4 \rightarrow a = b^k$

$T(n) = O(n^k \ log^{p+1} n)$

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

Ex-6 $T(n) = 4T(\frac{n}{2})+n^2\log n$,

$a = 4, b = 2, k = 2, p = 1.$

$a = 4, b^k = 4 \rightarrow a = b^k$

$T(n) = O(n^k \ log^{p+1} n)$

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

Solution using Master Theorem

Ex-7 $T(n) = 2T(\frac{n}{2}) + \frac{n}{\log n}$,

$a = 2, b = 2, k = 1, p = -1.$

$a = 2, b^k = 2 \Rightarrow a = b^k$

$T(n) = O(n^k \log \log n)$

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

Ex-8 $T(n) = T(\frac{n}{2}) + n^2$,

$a = 1, b = 2, k = 2, p = 0.$

$a = 1, b^k = 4 \Rightarrow a < b^k$

$T(n) = O(n^k \log^p n)$

$= O(n^2)$

Ex-9 $T(n) = 2T(\frac{n}{2}) + n^2 \log^2 n$,

$a = 2, b = 2, k = 2, p = 2.$

$a = 2, b^k = 4 \Rightarrow a < b^k$

$T(n) = O(n^k \log^p n)$

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

Master Theorem for Decreasing Functions

$$T(n) = aT(n-b) + g(n)$$

where g(n) is $O(n^k)$

a) $a < 1 : T(n) = O(n^k)$

b) $a = 1 : T(n) = O(n^{k+1})$

c) $a > 1 : T(n) = O(n^k a^{n/b})$

Solution using Master Theorem

Ex-1 T(n) = T(n-1)+1,

a = 1, b = 1, k = 0.

T(n) = $O(n^{k+1})$ = O(n)

Ex-3 T(n) = 2T(n-1)+1,

a = 2, b = 1, k = 0.

T(n) = $O(n^k a^{n/b})$

= $O(2^n)$

Ex-2 T(n) = T(n-1)+n,

a = 1, b = 1, k = 1.

T(n) = $O(n^{k+1})$ = $O(n^2)$

Ex-4 T(n) = 2T(n-1)+n,

a = 2, b = 1, k = 1.

T(n) = $O(n^k a^{n/b})$

= $O(n2^n)$

Correctness of Algorithms

- An algorithm is said to be correct
  - if, for every input instance, it halts with the correct output.
- We say that a correct algorithm
  - solves the given computational problem.
- An incorrect algorithm
  - might not halt at all on some input instances, or
  - it might halt with an incorrect answer.

Some Mathematics

Ordering Functions by Their Growth Rates

n	$\log n$	$\sqrt{n}$	n	$n \log n$	n^2	n^3	2^n
2	1	1.4	2	2	4	8	4
4	2	2	4	8	16	64	16
8	3	2.8	8	24	64	512	256
16	4	4	16	64	256	4,096	65,536
32	5	5.7	32	160	1,024	32,768	4,294,967,296
64	6	8	64	384	4,096	262,144	1.84×10^{19}
128	7	11	128	896	16,384	2,097,152	3.40×10^{38}
256	8	16	256	2,048	65,536	16,777,216	1.15×10^{77}
512	9	23	512	4,608	262,144	134,217,728	1.34×10^{154}
1,024	10	32	1,024	10,240	1,048,576	1,073,741,824	1.79×10^{308}

$1 < \log n < \text{sqrt}(n) < n < n \log n < n^2 < n^3 < \ldots < 2^n < 3^n < n^n$

Some Mathematics

- $\sum_{i=0}^{n} a^i = 1 + a + \cdots + a^n = \frac{1-a^{n+1}}{1-a}$
- $log_b a = c \; if \; a = b^c$
- $log_b ac = log_b a + log_b c$
- $log_b (a/c) = log_b a - log_b c$
- $log_b a^c = c log_b a$
- $log_b a = log_c a / log_c b$
- $b^{log_c a} = a^{log_c b}$

Case Studies: Analyzing Algorithms

```
Ex-1
#include <stdio.h>
void main(){
    int n=10;
    int a[n];
    a[3]=5;
    printf("%d",a[3]);
}
```

T(n) = 1+(1+1) + (1+1) → T(n) = O(1)

```
Ex-2
#include <stdio.h>
void main(){
    int n; scanf("%d",&n);
    int a[n];
    for(int i=0;i<n;i++)
        scanf("%d",&a[i]);
    for(int i=0;i<n;i++)
        printf("%d",a[i]);
}
```

T(n) = 2+(1+ (n+1) + 2(n)) + 2n + (1+ (n+1) + 2(n)) + 2n = 10n + 6 → T(n) = O(n)

Case Studies: Analyzing Algorithms

```
Ex-3
#include <stdio.h>
void main(){
 int n; scanf("%d",&n);
    int a[n];
    for(int i=0;i<n;i++)
        scanf("%d",&a[i]);
    for(int i=0;i<n;i++)
        for(int j=0;j<n;j++)
            printf("%d",a[i]);
}
```

T(n) = 2+(1+ (n+1) + 2(n)) + 2n + (1+ (n+1) + 2(n)) + n (1+ (n+1) + 2(n))
= $3n^2 + 10n + 6$
→ T(n) = $O(n^2)$

```
Ex-4
#include <stdio.h>
void main(){
 int n; scanf("%d",&n);
    int a[n];
    for(int i=0;i<n;i++)
        scanf("%d",&a[i]);
    for(int i=0;i<n;i++)
        for(int j=0;j<n/2;j++)
            printf("%d",a[i]);
}
```

T(n) = 2+(1+ (n+1) + 2(n)) + 2n + (1+ (n+1) + 2(n)) + n (1+ (n+1)/2 + 2(n/2))
→ T(n) = $O(n^2)$

Case Studies: Analyzing Algorithms

```
Ex-5
int findMinimum(int array[]) {
    int min = array[0];
    for(int i = 1; i < n; i++){
        if (array[i] < min) {
            min = array[i];
        }
    }
    return min;
}
```

$T(n) = O(n)$

Case Studies: Analyzing Algorithms

```
Ex-6
void fun(int n){
        if(n<=0)
                return;
        printf("%d",n);
        fun(n-1);
}
```

$T(n) = T(n-1) + 2 \rightarrow T(n) = O(n)$
Master Theorem for Decreasing Functions

```
Ex-7
void fun(int n){
        if(n<=0)
                return;
        printf("%d",n);
        fun(n/2);
}
```

$T(n) = T(n/2) + 2 \rightarrow T(n) = O(\log n)$
Master Theorem for Dividing Functions

Case Studies: Analyzing Algorithms

```
Ex-8
void fun(int n){
    if(n<=0)____1
        return;
    for(int i=0;i<k';i++) ____(k'+1)
        fun(n-1); ____k'*T(n-1)
}
```

(T(n) = 1 + (k'+1) + (k'(T(n-1)) = k'*T(n-1) + (k' + 2) ➔ T(n) depends on value of k' (Master Theorem for Decreasing Functions))*

```
Ex-9
void fun(int n){
    if(n>1){ ____1
        for(int i=0;i<n;i++) ____(n+1)
            printf("%d",i); ____n
        fun(n/2); ____T(n/2)
        fun(n/2); ____T(n/2)
    }
}
```

T(n) = 1 + (n+1) + n + 2T(n/2) = 2T(n/2) + (2n + 2)
a = 2, b= 2, k = 1, p = 0. O(n log n) as per Master Theorem for Dividing Functions

References

1. Algorithms Design: Foundations, Analysis and Internet Examples Michael T. Goodrich, Roberto Tamassia, 2006, Wiley (Students Edition)
2. Data Structures, Algorithms and Applications in C++, Sartaj Sahni, Second Ed, 2005, Universities Press
3. Introduction to Algorithms, TH Cormen, CE Leiserson, RL Rivest, C Stein, Third Ed, 2009, PHI

innovate achieve lead

BITS Pilani
Hyderabad Campus

Any Question!!

Thank you!!

BITS Pilani
Hyderabad Campus

T(n) = 2T(n/2) + 2

This will be third case of master's theorem or the second case?

Calculation:

f(n) = 2

Number of leaves = n ^ logb(a) = n ^ log(1) = n ^ 0 = 1

f(n) > n^(logb(a)): Third case. Right?

The answer will be Theta(2)?

has context menu