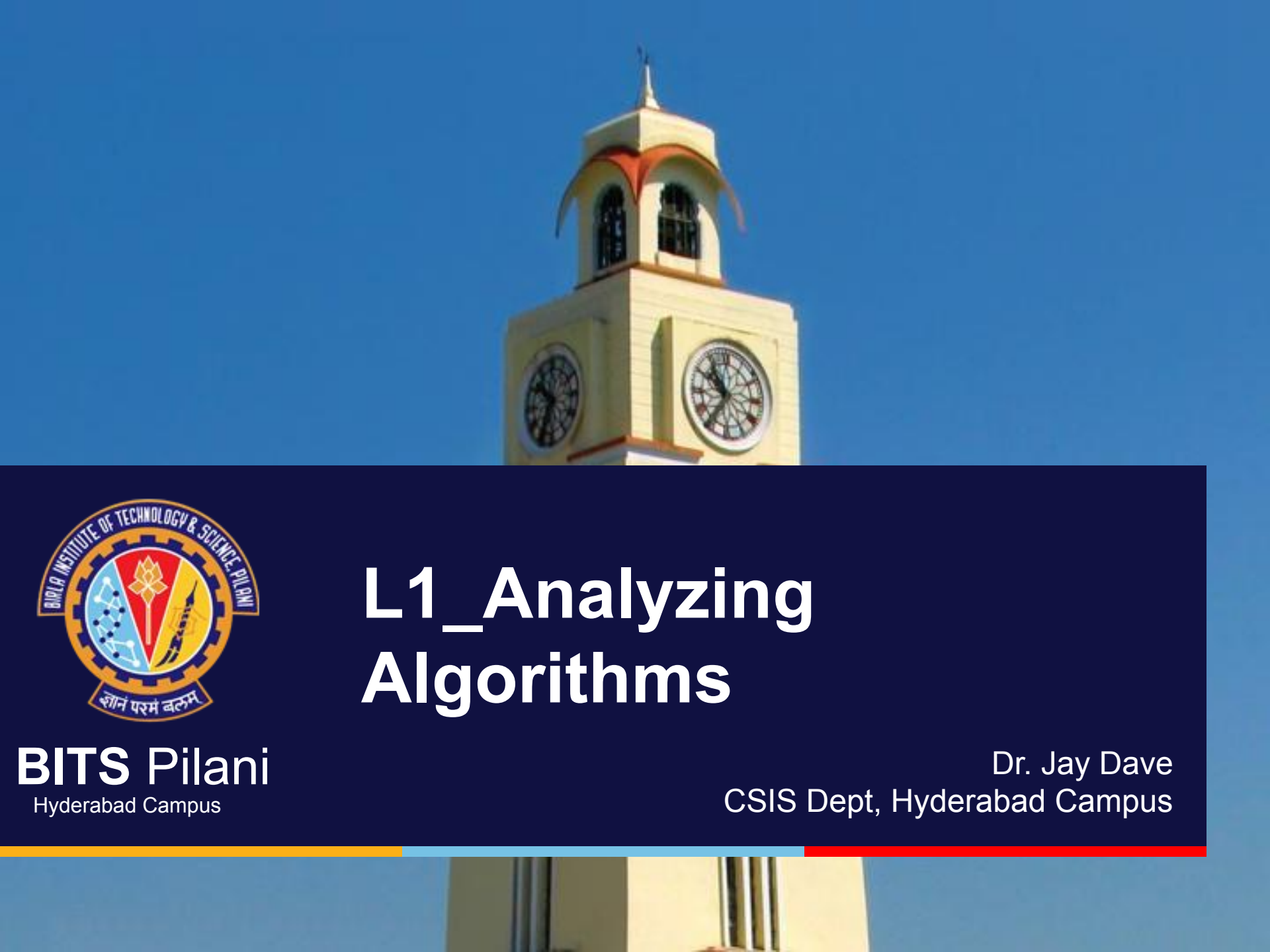


L1_Analyzing Algorithms

BITS Pilani
Hyderabad Campus

Dr. Jay Dave
CSIS Dept, Hyderabad Campus

Data Structures and Algorithms Design (Merged-SEZG519/SSZG519)

L1_Analyzing Algorithms

Content of L-1

1. Theoretical Foundation
   i. Algorithms and it's Specification
   ii. Random Access Machine Model
   iii. Counting Primitive Operations
   iv. Notion of best case, average case and worst case
2. Characterizing Run Time
   i. Use of asymptotic notation
   ii. Big-Oh Notation, Little-Oh, Omega and Theta Notations
3. Correctness of Algorithms
4. Analyzing Recursive Algorithms
   i. Recurrence relations
   ii. Specifying runtime of recursive algorithms
   iii. Solving recurrence equations
5. Case Study: Analyzing Algorithms

Theoretical Foundation

- What are algorithms?
- Why is the study of algorithms worthwhile?
- What is the role of algorithms relative to other technologies used in computers?

Theoretical Foundation

- Informally, an algorithm is any well-defined computational procedure that takes some value, or set of values, as input and produces some value, or set of values, as output.
- An algorithm is thus a sequence of computational steps that transform the input into the output.
- A tool for solving a well-specified computational problem.

Theoretical Foundation

Data structure

- Data: value or set of values. E.g. Rohit, 34, {11,33,2,51}, etc.
- Data structure: Logical or mathematical model of particular organization of data is called data structure.
- Array: A list of finite number of similar data elements. A[1], A[2], ..., A[N].
- Linked List: A list linked to one another.
- Stack: A linear list following Last In First Out system (LIFO).
- Queue: A linear list following First In First Out system (FIFO).
- Graph: A non-linear list consisting vertices and edges.
- Tree: A graph without cycles.

Theoretical Foundation

Data structure operations

- Traversing
- Searching
- Inserting
- Deleting
- Sorting
- Merging

Theoretical Foundation

- If an algorithm has been implemented, we can study its running time by executing it on various test, inputs and recording the actual time spent in each execution.
- Depends on language or operating system or execution environment.
- We are interested in determining the dependency of the running time on the size of the input.
- One way is that we can perform several experiments on many different test inputs' of various sizes.
- The running time is affected by the hardware environment (processor, clock rate, memory, disk, etc.). and software environment (operating system, programming language, compiler, interpreter, etc.) in which the algorithm is implemented, compiled, and executed.

Theoretical Foundation

Limitations of experimental analysis

- Experiments can be done only on a limited set of test inputs, and care must be taken to make sure these are representative.
- It is difficult to compare the efficiency of two algorithms unless experiments on their running times have been performed in the same hardware and software environments.
- It is necessary to implement and execute an algorithm in order to study its running time experimentally.

Theoretical Foundation

We desire an analytic framework that

- Takes into account all possible inputs
- Allows us to evaluate the relative efficiency of any two algorithms in a way that is independent from the hardware and software environment.
- Can be performed by studying a high-level description of the algorithm without actually implementing it or running experiments on it.

This methodology aims at associating with each algorithm a function f(n) that characterizes the running time of the algorithm in terms of the input size n.

Theoretical Foundation

We will learn

- A language for describing algorithms
- A computational model that algorithms execute within
- A metric for measuring algorithm running time
- An approach for characterizing running times, including those for recursive algorithms.

Theoretical Foundation

Pseudocode: A mixture of natural language and high-level programming constructs that describe the main ideas behind a generic implementation of a data structure or algorithm.

- Expressions: We use standard mathematical symbols to express numeric and Boolean expressions We use the left arrow sign (□) as the assignment operator in assignment statements (equivalent to the = operator in C, C++, and Java) and we use the equal sign (=) as the equality relation in Boolean expressions. (equivalent to the "==" relation in C, C++, and Java).
- Method declarations: Algorithm name(paraml , param2 ,...) declares a new method "name" and its parameters.
- Decision structures: if condition, then true-actions [else false-actions]. We use indentation to indicate what actions should be included in the true-actions and false-actions.

Theoretical Foundation

- While-loops: while condition do actions. We use indentation to indicate what actions should be included in the loop actions
- Repeat-loops: repeat actions until condition. We use indentation to indicate whit actions should be included in the loop actions
- For-loops for variable-increment-definition do actions We use indentation to indicate what actions should be included among the loop actions
- Array indexing: A[i] represents the ith cell in the array A The cells' of an n-celled array A are indexed from A[0] to A[n-1] (consistent with C, C++, and Java).
- Method calls: object.method(args) (object is optional if it is understood).
- Method returns: return value This operation returns the value specified to the method that calls this one.

Theoretical Foundation

Algorithm arrayMax(A, n):

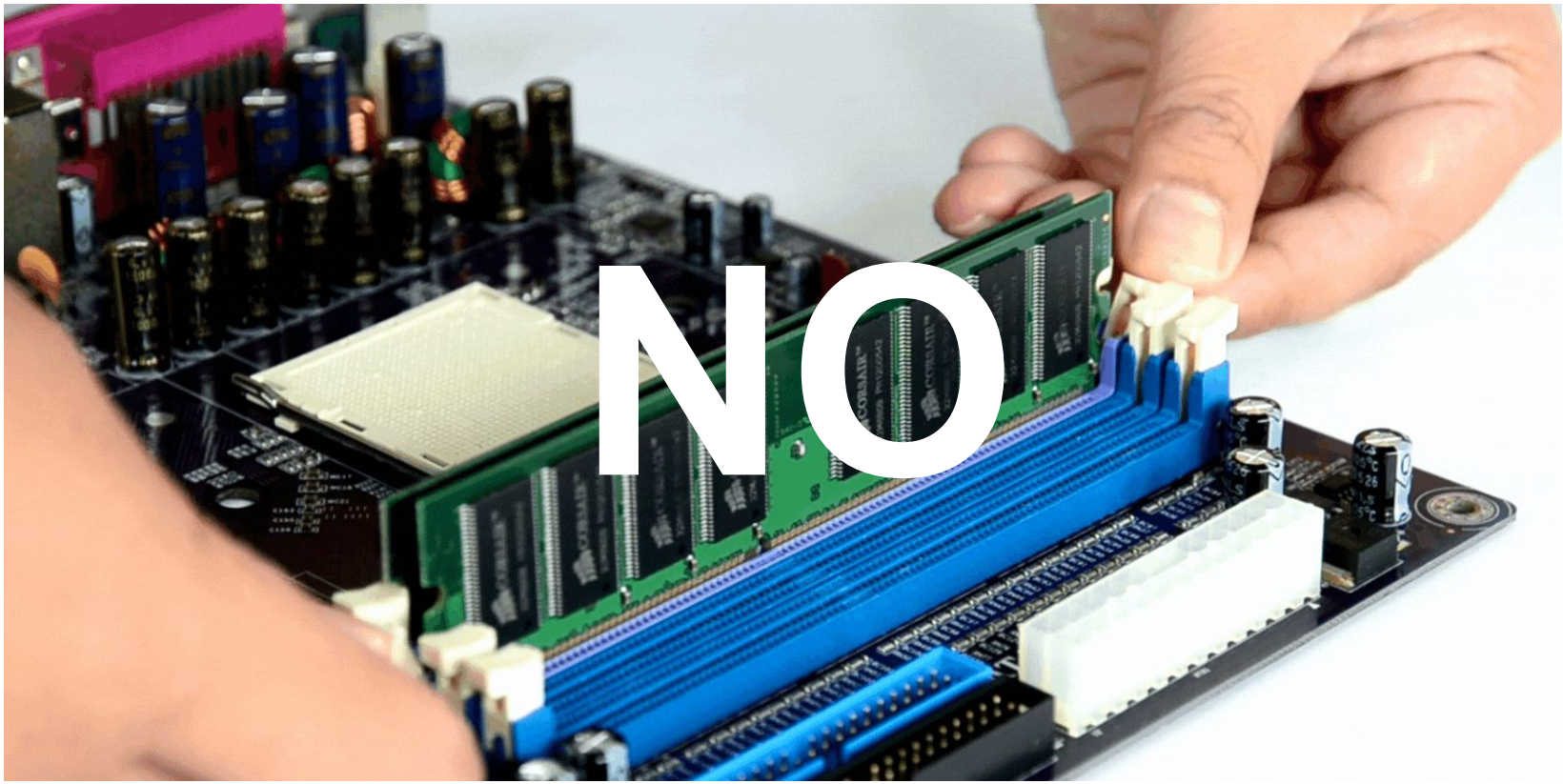
Input: An array A storing $n \geq 1$ integers.

Output: The maximum element in A.

```
currentMax ← A[0]
for i ← 1 to n − 1 do
    if currentMax < A[i] then
        currentMax ← A[i]
return currentMax
```

Theoretical Foundation

Random Access Machine (RAM) Model



Theoretical Foundation

Random Access Machine (RAM) Model

- As we noted, experimental analysis is valuable, but having limitations.
- If we wish to analyze a particular algorithm without performing experiment on running time, we can analyze high-level code or pseudo-code.
- High-level primitive operations for pseudo-code:

Assigning a value to variable	Calling a method
Performing an arithmetic operation	Comparing two numbers
Indexing into an array	Following an object reference
Returning from a method	

Theoretical Foundation

Random Access Machine (RAM) Model

- Depends on hardware/software environment but is constant.
- Instead of trying to understand execution time of primitive operations, we simply count the number of operations.
- This count correlates to actual execution time in a specific environment.
- Here, the assumption is that primitive operations takes almost similar execution time.
- We assume that CPU in RAM model can perform any primitive operation in a constant number of steps, which not depends on size of input.

Theoretical Foundation

Counting Primitive Operations

Algorithm arrayMax(A, n):

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

Output: The maximum element in A.

```
currentMax ← A[0]
for i ← 1 to n − 1 do
    if currentMax < A[i] then
        currentMax ← A[i]
return currentMax
```

1
n
n
0 to n
1

Theoretical Foundation

Notion of best case, average case and worst case

- Best case: minimum number of primitive operations executed. E.g. max is at the first place in array.
- Worst case: maximum number of primitive operations executed. E.g. max is at the last place in array.
- Average case: average number of primitive operations executed. E.g. max is in between the first and last places of the array.

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}

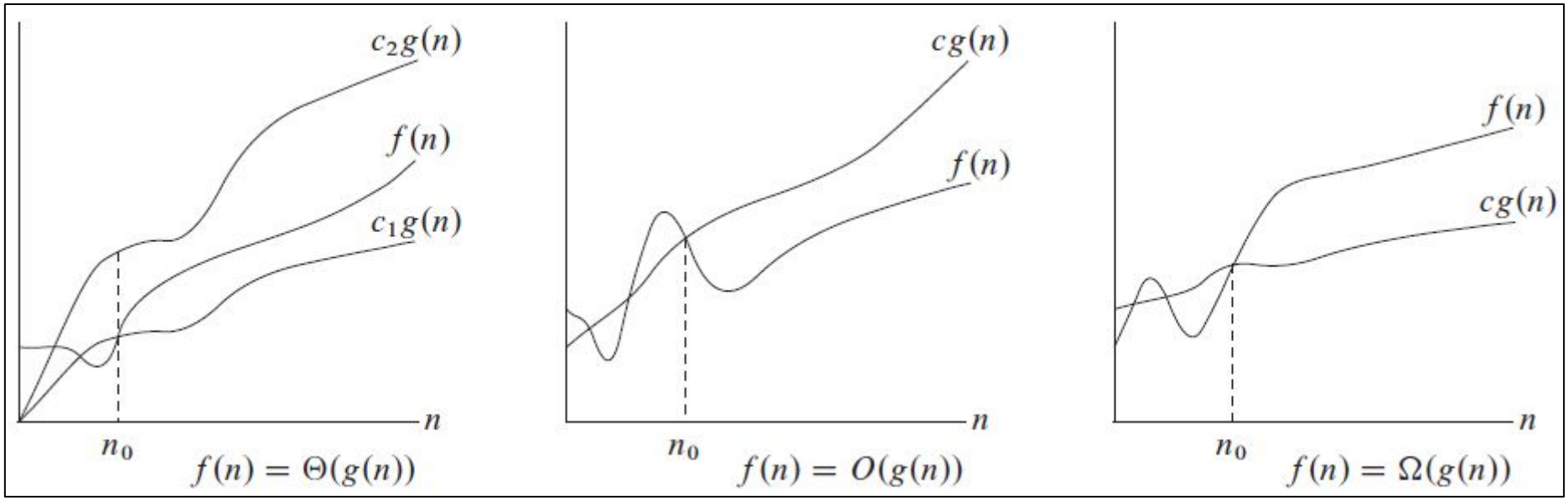
Some Mathematics

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

Characterizing Run Time

Big-Oh Notation, Little-Oh, Omega and Theta Notations:

- Asymptotic notation primarily to describe the running times of algorithms



Characterizing Run Time

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

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

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

- f(n) = Θ(g(n)) implies that f(n) = O(g(n)),
  i.e., Θ(g(n)) ϵ O(g(n))

Characterizing Run Time

Big-Oh Notation

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

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

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

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

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

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

Here, $c = 10$, $g(n) = n^2$

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

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

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

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

Characterizing Run Time

Omega Notation: f(n) = Ω(g(n)).

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

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

Characterizing Run Time

Omega Notation

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Characterizing Run Time

Theta Notation: f(n) = Θ(g(n)).

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

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

Characterizing Run Time

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 $n \geq 1$

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

$f(n) = \Theta(g(n)) = \Theta(n^2)$

Ex-2 $f(n) = 6n^3 \neq \Theta(n^2)$

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

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

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

Characterizing Run Time

Big-Oh Notation, Little-Oh, Omega and Theta Notations

- o-notation:

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

- w-notation:

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

Characterizing Run Time

Big-Oh Notation, Little-Oh, Omega and Theta Notations

- 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:

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

Characterizing Run Time

Big-Oh Notation, Little-Oh, Omega and Theta Notations

- Comparison

Transitivity:

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

Reflexivity:

$$
\begin{aligned}
f(n) &= \Theta(f(n)),\\
f(n) &= O(f(n)),\\
f(n) &= \Omega(f(n)).
\end{aligned}
$$

Characterizing Run Time

Big-Oh Notation, Little-Oh, Omega and Theta Notations

- Comparison

Symmetry:

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

Transpose symmetry:

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

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.

Analyzing Recursive Algorithms

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

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$(A, n-1), A[n-1]$}

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

Analyzing Recursive Algorithms

innovate achieve lead

1. Master Theorem for Dividing Functions

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

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

$a = 4, b^k = 2 \rightarrow a > b^k$

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

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

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

$a = 8, b^k = 4 \rightarrow a > b^k$

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

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

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

$a = 8, b^k = 2 \rightarrow a > b^k$

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

Analyzing Recursive Algorithms

1. Master Theorem for Dividing Functions

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

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

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

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

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

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

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

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

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

Analyzing Recursive Algorithms

1. Master Theorem for Dividing Functions

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

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

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

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

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

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

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

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

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

Analyzing Recursive Algorithms

Solving recurrence equations

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

Analyzing Recursive Algorithms

2. Master Theorem for Decreasing Functions

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

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

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

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

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

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

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

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

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

$= O(n2^n)$

Case Studies: Analyzing Algorithms

Ex-1

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

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]);
}
```

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]);
}
```

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]);
}
```

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;
}
```

Case Studies: Analyzing Algorithms

Ex-6

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

Ex-7

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

Ex-8

```
void fun(int n){
    if(n<=0)
        return;
    for(int i=0;i<k';i++)
        fun(n-1);
}
```

Ex-9

```
void fun(int n){
    if(n>1){
        for(int i=0;i<n;i++)
            printf("%d",i);
        fun(n/2);
        fun(n/2);
    }
}
```

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

Any Question!!

Thank you!!

BITS Pilani
Hyderabad Campus