Data Structures and Algorithms Design

DSECLZG519

BITS Pilani

Pilani|Dubai|Goa|Hyderabad

Parthasarathy

innovate achieve lead

BITS Pilani

Pilani|Dubai|Goa|Hyderabad

Contact Session #2
DSECLZG519 – Analyzing Algorithms

Agenda for CS #2

1) Recap of CS#1
2) General Rules and Problems
3) Mathematical analysis of Non-recursive algorithms
4) Mathematical analysis of Recursive algorithms
   - Iteration vs Recursion
   - Recurrence Tree
   - Iterative Substitution
5) Masters Theorem with examples
6) Exercises, Q&A!

Asymptotic Notations Analogies

- "The delivery will be there ***within*** your lifetime."
  - *(big-O, upper-bound)*
- "I can pay you ***at least*** one dollar."
  - *(big-omega, lower bound)*
- "The high today will be 25ºC *and* the low will be 19ºC."
  - *(big-theta, narrow)*

General rules to determine running time of algorithm

Loops: The running time of a loop is, at most, the running time of the statements inside the loop(including tests) multiplied by the number of iterations.

```
//executes n-times
        for(i=1;i<=n;i++)
                … // constant time
```

Total number

$$= \text{a constant } c \times n = cn = O(n)$$

$$F(n) = \sum_{i=1}^{n} 1$$

$$= n - 1 + 1$$

$$= n$$

General rules to determine running time of algorithm (Cont..,)

Nested loops: Analyze from inside out. Total running time is the *product of the sizes* of all the loops

```
//outer loop executed n-times
        for(i=1;i<=n;i++){
            //inner loop execute n-times
            for(j=1;j<=n;j++)
              //constant time
              k=k+1;
}
```

Total number = $c \times n \times n = cn^2 = O(n^2)$

General rules to determine running time of algorithm (Cont..,)

If-then-else statements: worst-case running time: the test, plus either the then part or the else part (whichever is the larger).

```
//test: constant c0
If(length()==0){
return false; // then part: constant c1
}
else {   //else part : (constant c2 +constant c3)× n
      for(int n =0; n<length();n++){
      //another if: constant + constant (no else part)
            if(! List[n].equals(otherlist.list[n]))
            //constant
            return false;
            }
   }
```

Total time = $c_0 + c_1 + (c_2 + c_3) \times n = O(n)$

General rules to determine running time of algorithm (Cont..,)

Logarithmic Complexity: an algorithm is O(log n) if it takes constant time to cut the problem size by a fraction (usually by ½)

As an example let us consider the following program

```
for(i=1; i<=n;)
        i=i*2;
```

If we observe carefully, the value of i is doubling every time initially i=1, in next step i=2, and in subsequent steps i = 4,8 and so on.

Let us assume that the loop is executing some k-times.

At k^{th} step $2^k=n$ and we come out of loop

Taking logarithm on both sides, gives

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

$k\log 2 = \log n$

$k = \log n$ // if we assume base ~ 2.

Total number = O(log n)

Problem-1

What is the complexity of the program given below

```
void function (int n){
        for (i =1; i<=n; i++)
            for (j=1;j+n/2<=n;j=j++)
                //loop execute logn times
                for (k=1;k<=n;k=k*2)
                count++;
}
```

Time complexity of the above function is $O(n^2 \log n)$

More on it in the discussion thread …

Problem-2

```
function (int n){
        //constant time
        if(n==1) return;
        //outer loop execute n times
        for (int i =1;i<=n;i++){
        // inner loop executes only time due to break statement
                for (int j =1;j<=n;j++){
                        printf("*");
                        break;
                }
        }
}
```

Time complexity of the above function is?

Time Complexity of the above function O(n). *Even though the inner loop is bounded by n, but due to the break statement, it is executing only once.*

Complexity Definition

- **Linear Complexity:** Processing time or storage space is directly proportional to the number of input elements to be processed.
- **Quadratic complexity:** An algorithm is of quadratic complexity if the processing time is proportional to the square of the number of input elements.
- **Cubic complexity:** In the case of cubic complexity, the processing time of an algorithm is proportional to the cube of the input elements.
- **Logarithmic complexity:** An algorithm is of logarithmic complexity if the processing time is proportional to the logarithm of the input elements. The logarithm base is typically 2

Analysis of Algorithms

Techniques employed by algorithms to solve problems

- **Iterative**
- **Recursion**

What is Recursion

- The process in which a function calls **itself** directly or indirectly is called recursion and the corresponding function is called as recursive function
- Define a procedure P that is allowed to make calls to itself, provided those calls to P are for solving sub-problems of smaller size.

Condition for recursive procedure

- A recursive procedure should always define a base case
- Base case
  - Small sized problem which can be solved by the algorithm directly without using recursion

Recursive Algorithm

1. if problem is "*small enough*"
2. solve it *directly*
3. else
4. break into one or more *smaller sub problems*
5. solve each sub problem *recursively*
6. *combine* results into solution to whole problem

Requirements for Recursive Solution

- At least one "*small*" case that you can solve directly
- A way of *breaking* a larger problem down into:
  - One or more *smaller* sub problems
  - Each of the *same kind* as the original
- A way of *combining* sub problem results into an overall solution to the larger problem

Recursive Algorithm - Example

Recursive algorithm for finding length of a string:

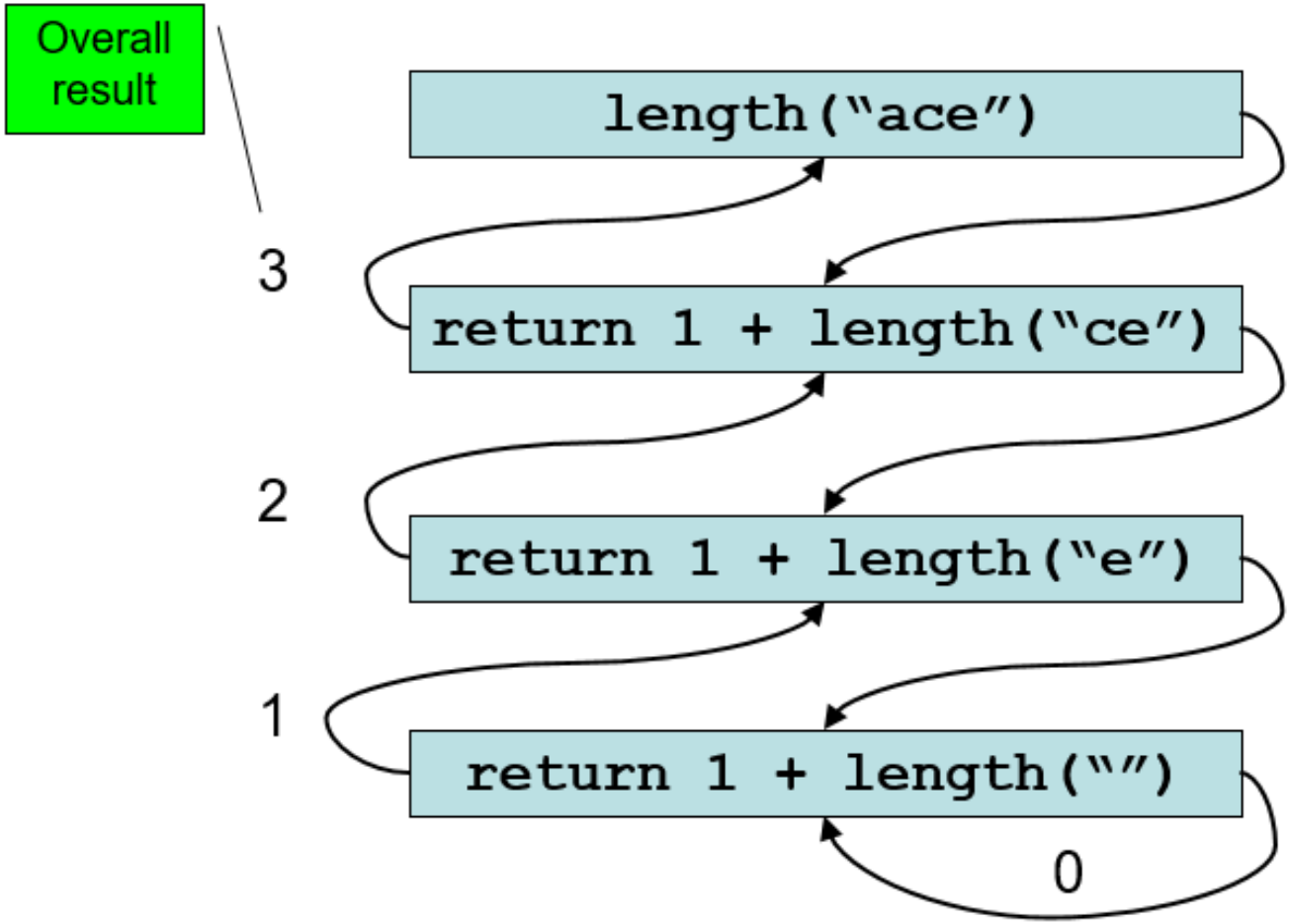
1. if string is empty (no characters)
2. return 0 ← *base case*
3. else ← *recursive case*
4. compute length of string without first character
5. return 1 + that length

Recursive Design Example: Code

Recursive algorithm for finding length of a string:

```
public static int length (String str) {
   if (str == null || str.equals(“”))
      return 0
   else
      return length(str.substring(1)) + 1
}
```

Tracing a Recursive Method



Example

- Compute N! = 1 * 2 * 3 * .... * N

```
//An iterative solution
int factorial(int N){
    result = 1;
    for (i = 1; i <= N; i++)
       result = i * result;
    return result;
}
```

- Alternative Solution: factorial calls itself recursively

```
int factorial (int N) {
  if (N == 1) {
    return 1;
  } else {
    return N * factorial (N-1);
  }
}
```

base case

recursive case

Analysis of non-recursive Algorithms

- Based on the input size, determine the number of parameters to be considered.
- Identify the basic operation in the algorithm
- Check whether the number of times the basic operation is executed depends only on the size of the input.
- Obtain the total number of times a basic operation is executed.
- Simplify using standard formulas, obtain the order of growth.

Example

```
Algorithm Linear_Search(key, a[], n)
//Purpose: To search for key in an array of n elements
// Inputs :
        n – the number of items present in the table
        a – the items to be searched are present in the table
        key – the item to be searched
//Output:
        return i whenever key is present in the list
        return -1  whenever key is not present in the list

Step 1: [Search for the key in the array/table]
        for  i ← 0 to n-1 do
                if (key = a[i]) return i;        // Key Found at position i
        end for

Step 2:[Key not found]
        return -1;
```

- In this problem, the size of input is n and this is the parameter to be considered.
- The statement executed in loop is the basic operation which is "if(key = a[i])"
- The basic operation is inside a single loop and hence the complexity is O(n)

Analysis of recursive Algorithms

- Based on the input size, determine the number of parameters to be considered.
- Identify the basic operation in the algorithm
- Obtain the number of times the basic operation is executed on different inputs of the same size.
- Obtain the **recurrence relation** with an appropriate initial condition.
- Solve the recurrence relation and obtain the order of growth and express using asymptotic notations.

Recurrence equation

- When an algorithm contains a recursive call to itself, its running time can often be described by a recurrence. A ***recurrence*** is an equation or inequality that describes a function in terms of its value on smaller inputs.

Solution to recurrence equations

$$T(n) = \begin{cases} 1 & ; n = 0 \text{ // base case} \\ 1 + T(n-1) & ; n > 0 \text{ // recursive case} \end{cases}$$

- Recursion tree method
- Iterative substitution method
- Master's theorem

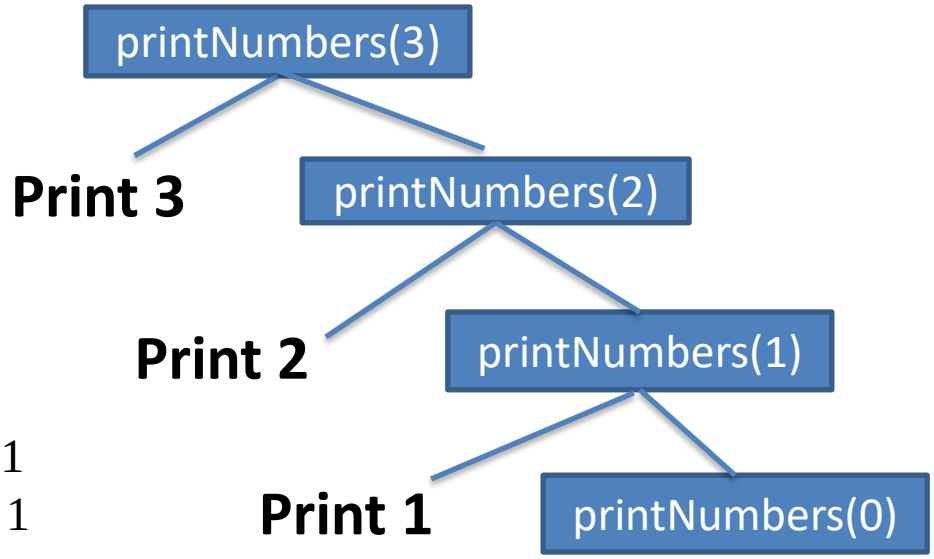
Recursive Tree method

Recursion tree method

- Visual method
- Converts the recurrence into a tree
- Sum all per-level cost to get the total cost of recursion

```
function printNumbers(n)
    if(n > 0)
        print n
        printNumbers(n-1)
end
```

Lets assume n = 3



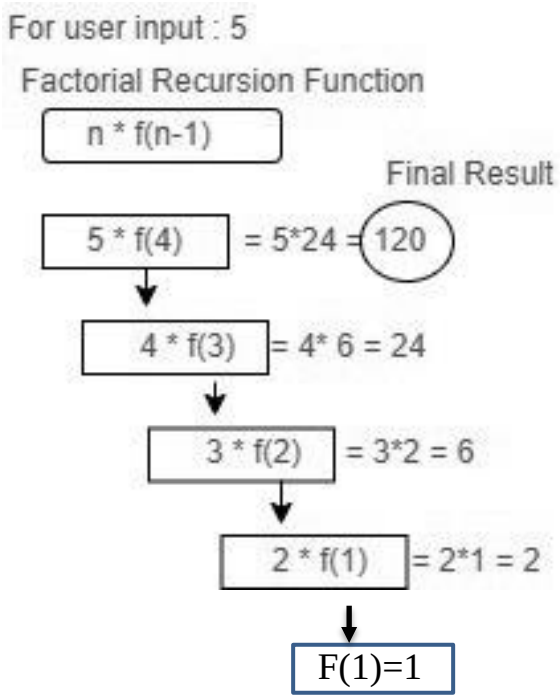
From the tree, its clear that it takes T(3)= 3 + 1

T(n)= n + 1

O(n)

Factorial using Recursion – Recursive Tree Method

Recursive tree method




```
func factorial(n)
    if(n == 1)
        return 1
    return n * factorial(n-1)
end
```

It is clear from the recursive tree that for the recursion unfolds n levels with 1 unit in each level of the tree. Hence, the complexity is n * 1 = **O(n)**

Iterative substitution method

Iterative substitution method

- Known as the "plug-and-chug" method
- Substitute the general form of the recurrence for each occurrence of the function T on the right-hand side (mathematical induction ☺)
- The hope in applying the iterative substitution method is that, at some point, we will see a pattern that can be converted into a general form equation!
- Then we use the base case to solve the equation

Example

```
function printNumbers(n)
    if(n > 0)
        print n
        printNumbers(n-1)
end
```

$$T(n) = \begin{cases} 1 & ; n = 0 \text{ // base case} \\ 1 + T(n-1) & ; n > 0 \text{ // recursive case} \end{cases}$$

Let us solve this recurrence relation using iterative substitution method!!

Substitution Method

- A way to solve a linear system algebraically is to use the substitution method. The substitution method functions by substituting the one *y* value with the other.
- Example:

$$y = 2x + 4$$

$$3x + y = 9$$

We can substitute y in the second equation with the first equation since y = y.

$$3x + y = 9$$

$$3x + (2x + 4) = 9$$

$$5x + 4 = 9$$

$$5x = 5$$

$$x = 1$$

This value of x can then be used to find y by substituting 1 with x e.g. in the first equation

$$y = 2x + 4$$

$$y = 2 \cdot 1 + 4$$

$$y = 6$$

The solution of the linear system is (1, 6).

Substitution Method

$$T(n) = \begin{cases} 1 & ; n = 0 \text{ // base case} \\ 1 + T(n-1) & ; n > 0 \text{ // recursive case} \end{cases}$$

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

Substitute value of T(n-1) in above:

$T(n) = [\ T(n-2) + 1\ ] + 1$

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

Substitute value of T(n-2) in above:

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

$T(n) = T(n-3) + 3$

.

.

.

In general, $T(n) = T(n-k) + k$ …. (2)

The recursion will end when the subproblem size reaches the base case, i.e. when n-k = 0 , which means when n= k . Substitute the value of k in (2) and solve:

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

$= T(0) + n$

$= 1 + n$

What is T(n-1) ?

We know that T(n)=T(n-1) + 1 …(1)

- *Put n=n-1 in (1) :*

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

= T(n-2) + 1

- *Put n=n-2 in (1) :*

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

= T(n-3) + 1

We have now solved the recurrence relation, drop the constant, we get **O(n)**

Example 2 – Factorial using Recursion

Iterative substitution method

```
func factorial(n)
    if(n == 1)
        return 1
    return n * factorial(n-1)
end
```

function T(n) is not actually for calculating the value of an factorial but it is telling you about the time complexity of the factorial algorithm.

$$T(n) = \begin{cases} 1 & ; n = 1 \text{ // base case} \\ 1 + T(n-1) & ; n > 1 \text{ // recursive case} \end{cases}$$

Let us solve this recurrence relation using iterative substitution method!!

Find attached CS#2 - PDF for solution. Complexity is O(n)

The Master's Method

- Each of the methods described above for solving recurrence equations is ad hoc and requires mathematical sophistication in order to be used effectively.
- The masters method is like a "cook-book" / A utility method for analysis recurrence relations.
- Useful in many cases for divide and conquer algorithms.
- They come handy when the recurrence relation is of the form :

T(n) = aT(n/b)+f(n) where a>=1, b>1, n>=d

  - n = the size of the current problem and d>=1 is a constant
  - a = the number of sub-problems in the recursion.
  - n/b = the size of each sub problem
  - *f(n)* = the cost of work that has to be done outside the recursive calls (Cost of dividing + merging).

The Master Theorem

The master method for solving such recurrence equations involves simply writing down the answer based on whether one of the three cases applies. Each case is distinguished by comparing $f(n)$ to the special function $n^{\log_b a}$

1. *If there is a small constant* $\varepsilon > 0$, *such that* $f(n)$ *is* $O(n^{\log_b a - \varepsilon})$, *then* $T(n)$ *is* $\Theta(n^{\log_b a})$.
2. *If there is a constant* $k \geq 0$, *such that* $f(n)$ *is* $\Theta(n^{\log_b a} \log^k n)$, *then* $T(n)$ *is* $\Theta(n^{\log_b a} \log^{k+1} n)$.
3. *If there are small constants* $\varepsilon > 0$ *and* $\delta < 1$, *such that* $f(n)$ *is* $\Omega(n^{\log_b a + \varepsilon})$ *and* $af(n/b) \leq \delta f(n)$, *for* $n \geq d$, *then* $T(n)$ *is* $\Theta(f(n))$.

Case 1 characterizes the situation where $f(n)$ is polynomially smaller than the special function, $n^{\log_b a}$. Case 2 characterizes the situation when $f(n)$ is asymptotically close to the special function, and Case 3 characterizes the situation when $f(n)$ is polynomially larger than the special function.

Simplified version of Master Theorem

$$T(n) = aT(n/b) + f(n)$$

- ✓ Where $f(n) = \theta(n^k \log^p n)$
- ✓ n = size of the problem
- ✓ a = number of subproblems in the recursion and a >= 1
- ✓ n/b = size of each subproblem
- ✓ b > 1, k >= 0 and p is a real number.

Case 1. if $\log_b a > k$, then $T(n) = \theta(n^{\log_b a})$

Case 2. if $\log_b a = k$, then

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

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

(c) if $p < -1$, then $T(n) = \theta(n^k)$

Case 3. if $\log_b a < k$, then

(a) if $p >= 0$, then $T(n) = \theta(n^k \log^p n)$

(b) if $p < 0$, then $T(n) = \theta(n^k)$

Simplified version Case 1 Example

innovate achieve lead

Case 1: if $\log_b a > k$, *then* $T(n) = \theta(n^{\log_b a})$

Consider **T(n) = 2T(n/2) + 1**

Solution:

$T(n) = aT(n/b) + f(n)$ where $f(n) = \theta(n^k \log^p n)$

Step 01: Here, a = 2, b = 2, f(n) = 1 which can happen when $\theta(n^0 \log^0 n)$. So, k=0 and p=0

Step 02: $\log_b a = \log_2 2 = 1$ and k = 0

Step 03: From step 02, $\log_b a > k$

Step 04: From SMT(Simplified Master theorem) Case 1:

$T(n) = \theta(n^{\log_b a})$

Hence, $T(n) = \theta(n^1)$

$= \theta(n)$

Simplified version Case 2 (a) Example

innovate achieve lead

Case 2: if $\log_b a = k$, *then*
(a) if $p > -1$, *then* $T(n) = \theta(n^k \log^{p+1} n)$

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

Solution:

T(n) = aT(n/b) + f(n) where f(n) =$\theta(n^k \log^p n)$

Step 01: Here, a = 2, b = 2 , k=1 and p=0

Step 02: $\log_b a = \log_2 2 = 1$ and k = 1

Step 03: From step 02, $\log_b a = k$ so goto case 2

Step 04: P > -1 so goto 2(a) of simplified masters theorem.

So, $T(n) = \theta(n^k \log^{p+1} n)$

Hence, $T(n) = \theta(n^1 \log^{0+1} n)$

$= \theta(n \log n)$

Simplified version Case 2 (b) Example

innovate achieve lead

Case 2 : if $\log_b a = k$, *then*
(b) if $p = -1$, *then* $T(n) = \theta(n^k \log\log n)$

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

Solution:

$T(n) = aT(n/b) + f(n)$ where $f(n) = \theta(n^k \log^p n)$

Step 01: Here, $a = 2$, $b = 2$,

$n/\log n$ is $n^1 * \log^{-1} n$ so $k = 1$ and $p = -1$

Step 02: $\log_b a = \log_2 2 = 1$ and $k = 1$

Step 03: From step 02, $\log_b a = k$ so goto Case 2

Step 04: P = -1 So, goto simplified Masters Theorem Case 2b.

So, $T(n) = \theta(n^k \log\log n)$

Hence, $T(n) = \theta(n^1 \log\log n)$

$= \theta(n \log\log n)$

Simplified version Case 2 (c) Example

Case 2 : if $\log_b a = k$, *then*
(c) if $p < -1$, *then* $T(n) = \theta(n^k)$

Consider T(n) = 2 T(n/2) + n/log^2 n

Solution:

T(n) = aT(n/b) + f(n) where f(n)=$\theta(n^k \log^p n)$

Step 01: Here, a = 2, b = 2 ,

n/log^2 n is $n^1 * \log^{-2} n$ so k = 1 and p = -2

Step 02: $\log_b a = \log_2 2 = 1$ and k = 1

Step 03: From step 02, $\log_b a = k$ so goto Case 2

Step 04: P < -1 So, goto simplified Masters Theorem Case 2c.

So, $T(n) = \theta(n^k)$

Hence, $T(n) = \theta(n^1)$

$= \theta(n)$

Simplified version Case 3 (a) Example

innovate achieve lead

Case 3 : if $\log_b a < k$, *then*
(a) if $p >= 0$, *then* $T(n) = \theta(n^k \log^p n)$

Consider T(n) = 4T(n/2) + n^3 log n

Solution:

T(n) = aT(n/b) + f(n) where $f(n)=\theta(n^k \log^p n)$

Step 01: Here, a = 4, b = 2 , k = 3 and p = 1

Step 02: $\log_b a = \log_2 4 = 2$ and k = 3

Step 03: From step 02, $\log_b a < k$ so goto Case 3

Step 04: P > 0 So, goto simplified Masters Theorem Case 3a.

So, $T(n) = \theta(n^k \log^p n)$

Hence, $T(n) = \theta(n^3 \log^1 n)$

$= \theta(n^3 \log n)$

Simplified version Case 3 (b) Example

innovate achieve lead

Case 3 : if $\log_b a < k$, *then*
(b) if $p < 0$, *then* $T(n) = \theta(n^k)$

Consider T(n) = 2 T(n/2) + $n^2/\log^2 n$

Solution:

T(n) = aT(n/b) + f(n) where $f(n)=\theta(n^k \log^p n)$

Step 01: Here, a = 2, b = 2 , k = 2 and p = -2 (because $n^2 * \log^{-2} n$)

Step 02: $\log_b a = \log_2 2 = 1$ and k = 2

Step 03: From step 02, $\log_b a < k$ so goto Case 3

Step 04: P < 0 So, goto simplified Masters Theorem Case 3b.

So, $T(n) = \theta(n^k)$

Hence, $T(n) = \theta(n^2)$

$= \theta(n^2)$

Master Method Capability

- Not all recurrence relations can be solved with the use of the master theorem i.e. if
  - T(n) is not of the form as mentioned earlier, ex: T(n) = sin n
  - f(n) is not a polynomial, ex: $T(n) = 2T(n/2) + 2^n$

Exercises

Find recursive solution for following problems:

1. Return Nth Fibonacci Number
2. Power of a given number
3. Recursive array search
4. Sum of elements of an array using recursive function

Exercises

- *Try to run the Practice sheet 1 to get a feel of order of growth,*
- *Solve the below recurrence relations using the iterative substitution method:*

1) $T(n) = \begin{cases} 1 & ; n = 1 \\ n + T(n-1) & ; n >= 1 \end{cases}$

2) $T(n) = \begin{cases} 1 & ; n = 1 \\ 2T(n-1) & ; n > 1 \end{cases}$

- Tree using recurrence tree for: $T(n) = T(n/4) + T(n/2) + n^2$
- *Solve the below recurrence relations masters theorem :*

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

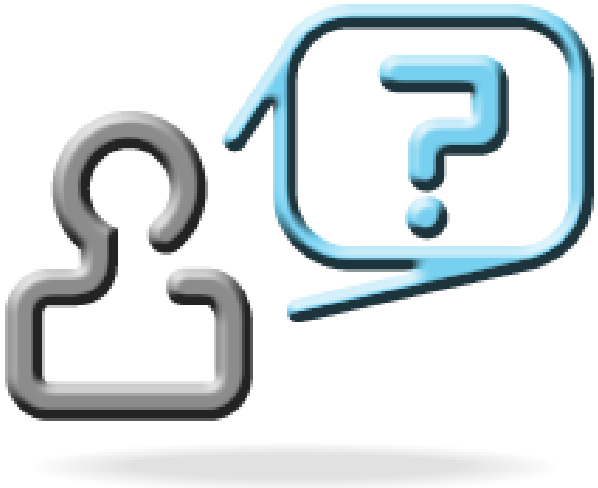
b) $T(n) = 9T(n/3) + n$

c) $T(n) = 3T(n/4) + n \log n$

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

e) $T(n) = 4T(n/2) + n$

f) $T(n) = 9T(n/3) + 1$



We will look at Linear Data structures in the next class

Thank You for your time & attention !

Contact : parthasarathypd@wilp.bits-pilani.ac.in