Complexity Analysis of Algorithms

Dr Varun Ojha

Department of Computer Science

University of Reading

Learning Objectives

On completion of three parts of this lecture, you will be able to

- Understand how to write algorithm / pseudocode
- Evaluate complexity of an algorithm from a pseudocode
- Understand recursive algorithms
- Evaluate time complexity of recursive algorithms
- Create a program to plot standard functions

Content of this lecture

- Part – I: Algorithms, Code Snippets, and Time Order
  - Writing an algorithm and pseudocode
  - Time order definition
  - Example pseudocode and complexity evaluation
- Part – II: Recursive algorithms
  - Recursive algorithm’s complexity
  - Asymptotic order evaluation
- Part –III: Exercise
  - Exercises
  - Write a recursive algorithm

Algorithm and Code Snippets

Writing an Algorithm/ Pseudocode

Write an algorithm to count distinct elements of an array of size n.

Algorithm: Counting distinct elements of an array

Input: An array A of size n

Output: number of distinct elements

CountDistinctElements(data A)

```
Count  = 1; /* Initialise a variable to 1 */
for i = 1 to n do  /* Pick all elements one by one */
    j = 0
    for j = 0 to j < i do /* scan array and compare elements*/
        if A[i] == A[j] then
            break loop
        end if
    end for
    if i == j then
        Count  = Count  + 1
    end if
end for
return Count
```

Writing a Code Snippets/Listing

```
int countDistinctElement(int A[ ])
{
  int n = sizeof(A)
  int count = 1;
  /* Pick all elements one by one */
  for (int i = 1; i < n; i++) {
    int j = 0;
    for (j = 0; j < i; j++)
      if (A[i] == A[j])
        break;
    /* increment counter if all previous elements were distinct */
    if (i == j)
       count ++;
  }
  return count;
}
```

Time Order

- $O(1)$ – Constant
- $O(\log n)$ – Logarithmic
- $O(n)$ – Linear
- $O(n \log n)$ – Logarithmic
- $O(n^k)$ – Polynomial
- $O(k^n)$ – Exponential
- $O(n!)$ – Factorial

Iterative Algorithm

Dr Varun Ojha
Department of Computer Science

University of Reading

Constant

```
// Code Snippet: SUM
int sumSeries(int[] A){
    n = size(A); // 1 unit
    ans = n*(n+1)/2; // 1 unit
return ans;
}
```

Algorithm Complexity

1 unit + 1 unit

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

Since 1 + 1 is constant, we will write **O(1)** instead of saying O(1+1). This is because the “rate of growth” will be constant no matter what the size of input A is.

Linear

```
// Code Snippet: Sum Series
int sumSeries(int[] A):
   n = size(A); // c unit
   sum = 0; // c unit
   for(i = 0; i<n; i++){
       sum += A[i]; // 1 unit n times
   }
return sum;
}
```

Algorithm Complexity

Trace variables	execution times
i = 0	1
i = 1	1
:	:
i = k	1

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

$$T(n) = k + 2c$$

For $k = n$, the algorithm will stop. Hence, $T(n) = n$

We will write

$$T(n) = O(n)$$

Logarithmic

```
// Code Snippet: Count
int count(int n):
   n = size(A); // c unit
   count = 0; // c unit
   for(i = n; i >= 1; i/2){
       count += 1; // 1 unit
   }
return sum;
}
```

Algorithm Complexity

Trace variables	execution times
i = n	1
i = $n/2$	1
i = $n/2^2$	1
⋮	⋮
i = $n/2^{k-1}$	1

For $\frac{n}{2^k} \leq 1$,

i.e., $2^k \leq n$ or $k = \log n$ iterations the algorithm will stop.

Hence, $T(n) = \log n$ and, we will write

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

Polynomial

```
// Code Snippet COUNT
int count(int[] A){
   n = size(A); // 1 unit
   count = 0;  // 1 unit
   for(i = 0; i<n; i++){
     for(j = 0; j<n; j++){
         count += 1 // 1 unit n^2 times
     }
   }
return count;
}
```

Algorithm Complexity

Trace variables	execution times
i = 0, j = 0,1,2,...n	n
i = 1, j = 0,1,2,...n	n
:	:
i = k, j = 0,1,2,...n	n

$$T(n) = n + n + \cdots + n = kn$$

For $k = n$, the algorithm will stop. Hence, $T(n) = n^2$

We will write:

$$T(n) = O(n^2)$$

Fundamental of Computer Science
CS1FC16: Lecture 02, Part – II

Recursive Algorithms

Dr Varun Ojha
Department of Computer Science

University of Reading

Recursive Algorithm, Example 1

```
// Recursive Algorithm
int funCall(int n){
   // do some other stuff
   if (n > 0){
      // do some other stuff
      print(n); # 1 unit
   funCall(n -1) # calls itself
   // do some other stuff
   }
}
```

Equation

$$T(n) = \begin{cases} 1 & n = 0 \\ T(n-1) + 1 & n > 0 \end{cases}$$

Complexity -> O(n)

How?

Substitution method

T(n) = T(n-1) + 1 -> O(n)?

We have:

$$\boldsymbol{T(n)} = \begin{cases} \mathbf{1} & \boldsymbol{n = 0} \\ \boldsymbol{T(n-1) + 1} & \boldsymbol{n > 0} \end{cases}$$

We want to solve:

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

Substitute $\boldsymbol{T(n-1)}$ in Eq. (1)

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

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

Substitute $\boldsymbol{T(n-2)}$ in Eq. (2)

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

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

Substitute $\boldsymbol{T(n-3)}$ in Eq. (3) and so on up to $\boldsymbol{k}$

:

We will have

$$T(n) = [T(n-k) + 1] + k - 1$$

$$T(n) = T(n-k) + k \qquad (k)$$

Find $\boldsymbol{T(n-1)}$ value

Since we have

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

Therefore,

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

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

Find $\boldsymbol{T(n-2)}$ value

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

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

Assume $\boldsymbol{k = n}$ in Eq. ($\boldsymbol{k}$), for this recurrence comes to a halt.

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

$$T(n) = T(0) + n$$

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

$$\boldsymbol{T(n) = O(n)}$$

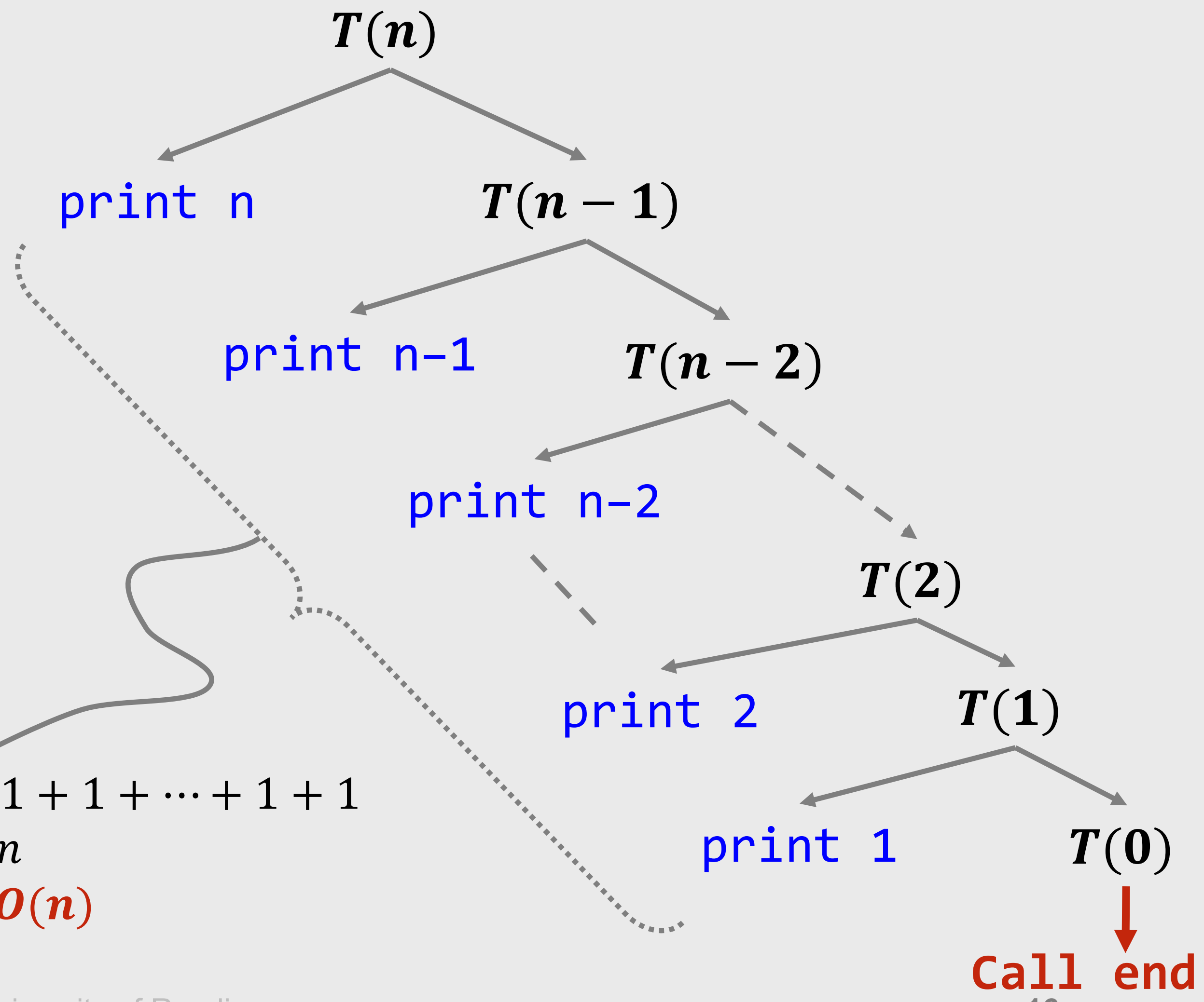
Tracing a recurrence tree

T(n) = T(n-1) + 1 -> O(n)?

```
// Recursive Algorithm
int funCall(int n){
    // do some other stuff
    if (n > 0){
        // do some other stuff
        print(n); # 1 unit
    funCall(n -1) # calls itself
    // do some other stuff
    }
}
```

Equation

$$T(n) = \begin{cases} 1 & n = 0 \\ T(n-1) + 1 & n > 0 \end{cases}$$



$$\begin{aligned} T(n) &= 1 + 1 + \cdots + 1 + 1 \\ &= n \\ &= \boldsymbol{O(n)} \end{aligned}$$

Recursive Algorithm, Example 2

```
// Recursive Algorithm
int funCall(int n):
    // do some other stuff
    if (n > 1){
        // do some other stuff
        for(i = 0; i<n; i++){
            count += 1 // 1 unit n times
        }
        funCall(n/2) // calls itself
        funCall(n/2) // calls itself
        // do some other stuff
    }
}
```

Equation

$$T(n) = \begin{cases} 1 & n = 1 \\ 2T\left(\frac{n}{2}\right) + n & n > 1 \end{cases}$$

Complexity -> O(n log n)

How?

Substitution method

T(n) = 2T(n/2) + n -> O(nlog n)?

We have:

$$\boldsymbol{T(n) = \begin{cases} 1 & n = 1 \\ 2T(n/2) + n & n > 1 \end{cases}}$$

We want to solve:

$$T(n) = 2T(n/2) + n \qquad (1)$$

Substitute $\boldsymbol{T(n/2)}$ in Eq. (1)

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

$$T(n) = 2^2T(n/2^2) + 2n \qquad (2)$$

Substitute $\boldsymbol{T(n/2^2)}$ in Eq. (2)

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

$$T(n) = 2^3T(n/2^3) + 3n \qquad (3)$$

Substitute $\boldsymbol{T(n/2^3)}$ in Eq. (3) and so on upto $\boldsymbol{k}$

:

We will have

$$T(n) = 2^k\left[2T\left(n/2^k\right) + n/2^k\right] + (k-1)n$$

$$T(n) = 2^kT\left(n/2^k\right) + kn \qquad (k)$$

Find $\boldsymbol{T(n/2)}$ value

Since we have

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

Therefore,

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

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

Find $\boldsymbol{T(n/2^2)}$ value

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

$$= 2T(n/2^3) + n/2^2$$

Assume $\boldsymbol{2^k = n}$ in Eq. ($\boldsymbol{k}$), for this recurrence comes to a halt.

$$T(n) = T(n/n) + kn$$

$$T(n) = nT(1) + kn$$

$$T(n) = n + nk$$

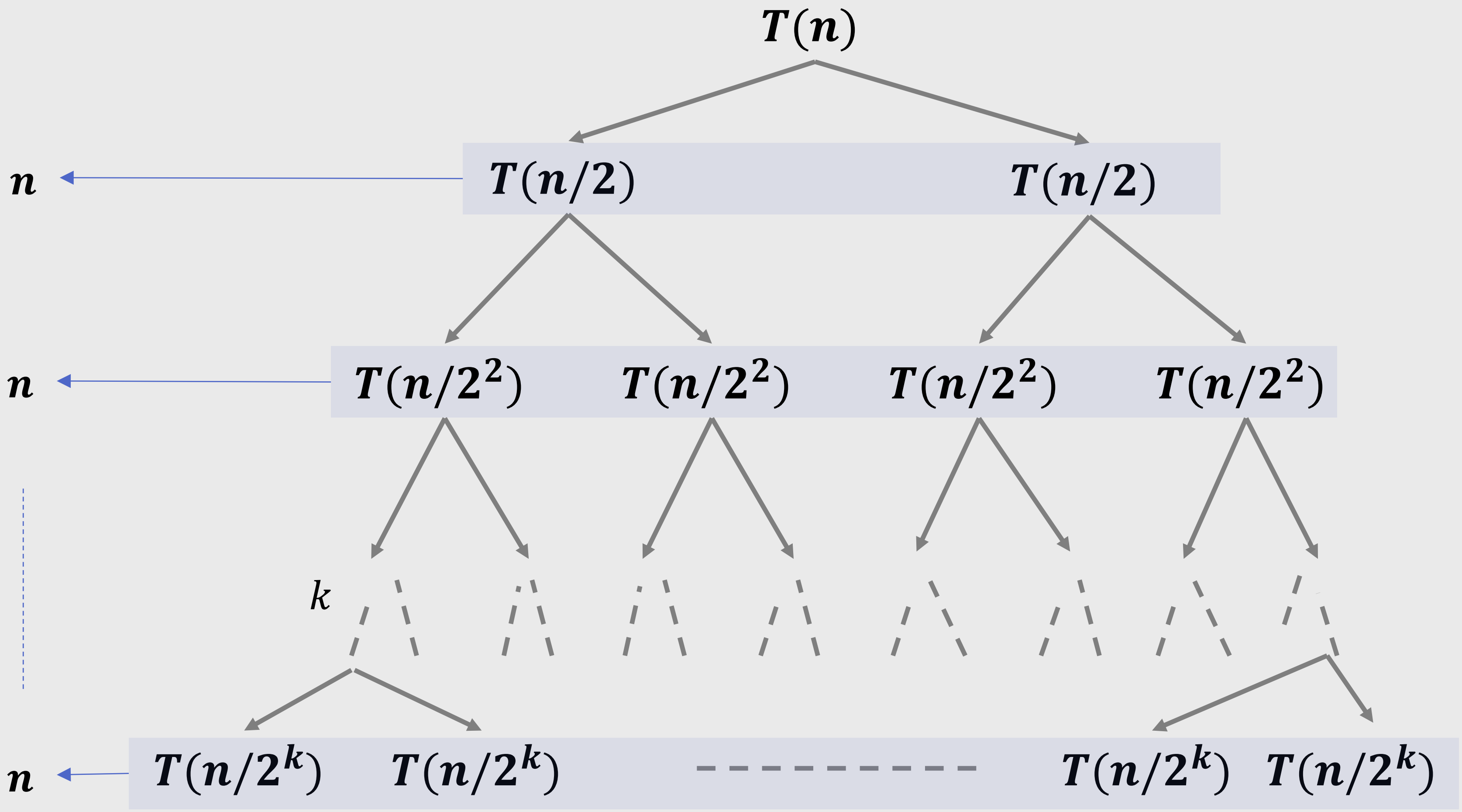
If $\boldsymbol{2^k = n}$, then $\boldsymbol{k = \log n}$

$\boldsymbol{T(n) = O(n \log n)}$ /* we ignore n because highest term is $\boldsymbol{n} \log \boldsymbol{n}$ */

Tracing a recurrence tree

T(n) = 2T(n/2) + n -> O(nlog n)?

```
// Recursive Algorithm
int funCall(int n):
    // do some other stuff
    if (n > 1){
        // do some other stuff
        for(i = 0; i<n; i++){
            count += 1
        }
        funCall(n/2) // calls
        funCall(n/2) // calls
    }
}
```

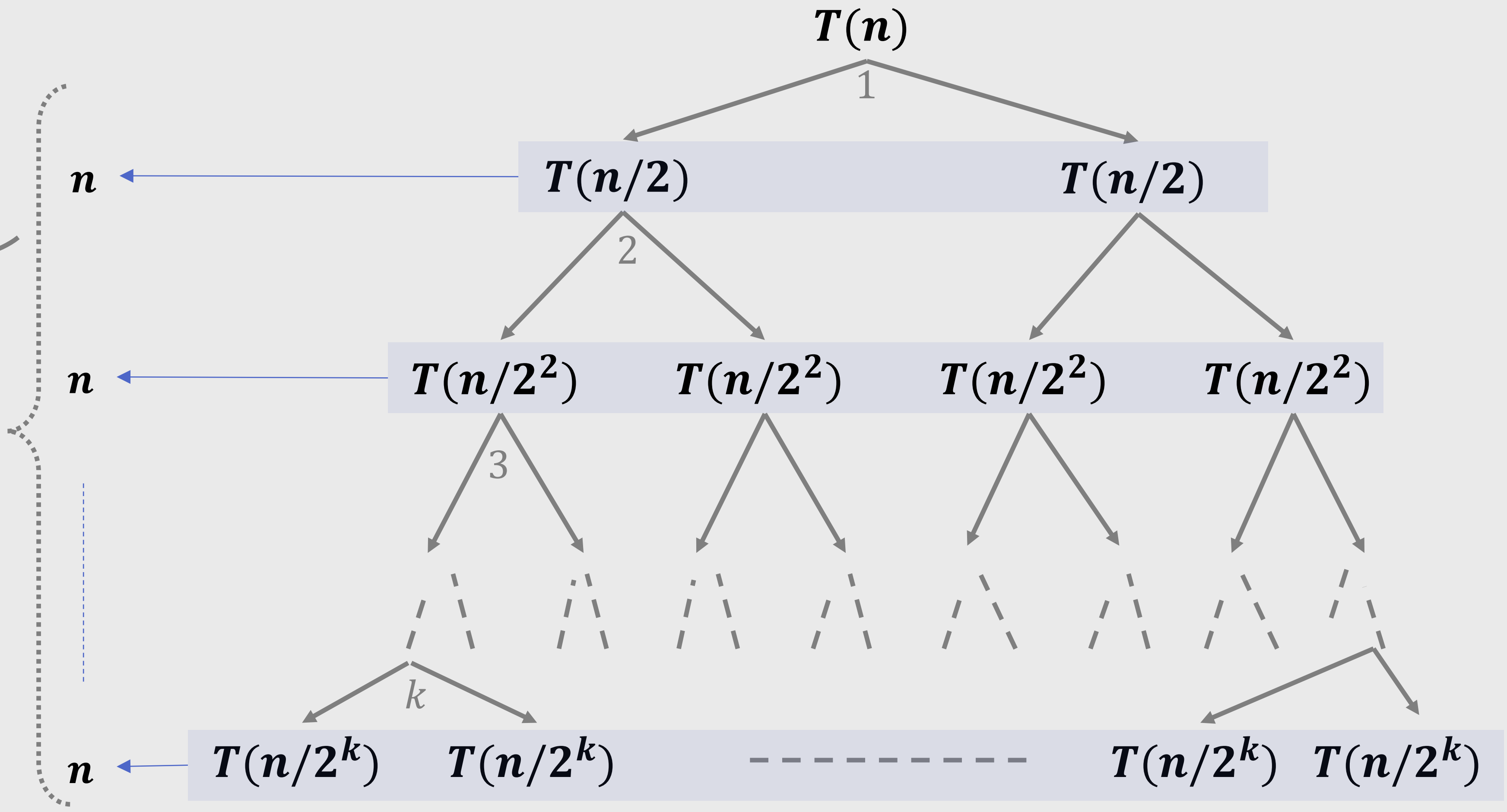


Equation

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

Tracing a recurrence tree

T(n) = 2T(n/2) + n -> O(nlog n)?



$\boldsymbol{T(n) = nk}$

Assume $2^k = n$ in tree for this recurrence comes to a halt.

$T(n) = nk$

If $\boldsymbol{2^k = n}$ **, then** $\boldsymbol{k = \log n}$

$\boldsymbol{T(n) = O(n \log n)}$

Fundamental of Computer Science
CS1FC16: Lecture 02, Part – III

Exercises and Practical

Dr Varun Ojha
Department of Computer Science
University of Reading

Exercise

1. Show that $T(n) = T(n-1) + n$ is $O(n^2)$

2. Show that $T(n) = T\left(\frac{n}{2}\right) + 1$ is $O(\log n)$

Show answers for 1 and 2 using a tree.

Exercise

- Write a program to produce sum of a series using a recursive algorithm and analysis time order of your algorithm.

- Write a program to compute factorial of a number using a recursive algorithm and analysis time order of your algorithm.

- Watch video for practicals