SCC.121: ALGORITHMS AND COMPLEXITY
Time Complexity of Recursive Algorithms

Emma Wilson e.d.wilson1@lancaster.ac.uk

Today's Lecture

Aim: To introduce some approaches for evaluating the time complexity of recursive algorithms

Learning objectives:

- To know what a recursive algorithm is and some examples of recursive algorithms
- To be able to evaluate the time complexity of simple recursive algorithms

Outline

- **What are Recursive Algorithms?**
- Examples of Recursive Algorithms
- How to evaluate the time complexity of Recursive algorithms?
  - Back Substitution
  - Recursion Tree
  - Master Theorem

Recursive Algorithms (Definition)

- A **recursive algorithm** is an algorithm which calls itself with *smaller* input values
- A **recursive algorithm** obtains the result for the current input by applying simple operations to the returned value for the smaller input.

When to use a recursive algorithm?

  - If a problem can be solved utilizing solutions to smaller versions of the same problem, and the smaller versions reduce to easily solvable cases, then one can use a recursive algorithm to solve that problem.

Outline

- What are Recursive Algorithms?
- **Examples of Recursive Algorithms**
- How to evaluate the time complexity of Recursive algorithms?
  - Back Substitution
  - Recursion Tree
  - Master Theorem

Recursive Algorithms (Example#1)

This following recursive function takes a number n as input and returns the sum 1 + 2 + ... + n

```
// Sum returns the sum 1 + 2 + 3 + … + n, where n >= 1.

func Sum(n){
    if n==1 {
        return n
    }
    return n + Sum(n-1)
}
```

Recursive Algorithms
(Example #1 - Time complexity function T(*n*))

In order to calculate the time complexity of a recursive function, we need to define its time complexity function T(*n*).

```
// Sum returns the sum 1 + 2 + 3 + … + n, where n >= 1.

func Sum(n){
    if n==1 {
        return n
    }
    return n + Sum(n-1)
}
```

Recursive Algorithms (Example #1 - Time complexity function T(*n*))

We identify two properties of T(*n*):

- Since Sum(1) is computed using a fixed number of operations C_1, $T(1) = C_1$
- If $n > 1$ the function will perform a fixed number of operations C_2, and in addition, it will make a recursive call to Sum(n-1). This recursive call will perform $T(n-1)$ operations.
  - In total, we get $T(n) = C_2 + T(n-1)$.

```
// Sum returns the sum 1 + 2 + 3 + … + n, where n >= 1.

func Sum(n) {
    if n==1 {
        return n
    }
    return n + Sum(n-1)
}
```

Recursive Algorithms (Example #1 - Time complexity function T(n))

- We are only looking for an asymptotic estimate of the time complexity
- Remember we drop the specific constants when using big-O, Ω, θ
- So, for simplicity,
  - let $C_1 = C_2 = 1$

Finding the time complexity of the *Sum* function can then be reduced to solving the recurrence relation:

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

Recursive Algorithms (Example#2 - iterative implementation)

This function is an **iterative** implementation of Binary Search algorithm

```
#include <stdio.h>
// iterative implementation of binary search
// algorithm to return the position of target
// x in the array A of size N

int binarySearch(int A[], int N, int x)
{
    int low = 0, high = N-1;
    while (low <= high)
    {
        int mid = (low + high)/2;

        if (x == A[mid])
            return mid;
        else if (x < A[mid])
            high = mid – 1;
        else
            low = mid + 1;
    }
    return -1
}
```

Recursive Algorithms (Example#2- recursive implementation)

Lancaster University

This function is a **recursive** implementation of Binary Search algorithm

Recursive function:

- Compare the mid of the search space with the key.
- Either return the index where the key is found
- Or call the recursive function for the next search space.
- For each call the search space is halved

```
#include <stdio.h>
// Recursive implementation of binary search
// algorithm to return the position of target
// x in the sub-array A[low…high]

int binarySearch(int A[], int low, int high, int x)
{
    if (low > high)
        return -1;

    int mid = (low + high)/2;

    if (x == A[mid])
        return mid;
    else if (x < A[mid])
        return binarySearch(A, low, mid - 1, x);
    else
        return binarySearch(A, mid + 1, high, x);
}
```

Recursive Algorithms (Example #2 - Time complexity function T(n))

What is the recurrence relation of the recursive **binarySearch** function?

- For each recursive call we perform a constant number of operations (required for the comparison at each iteration and deciding action accordingly) as well as dividing the array (so problem size) by 2. So, we can write T(n) = T($\frac{n}{2}$) + 1 when n > 1
- For n=1, we perform a constant number of operations, so can write T(1)=1

Finding the time complexity of the **binarySearch** function can be reduced to solving the recurrence relation:

- T(n) = T($\frac{n}{2}$) + 1 when n > 1
- T(1) = 1

Outline

- What are Recursive Algorithms?
- Examples of Recursive Algorithms
- How to evaluate the time complexity of Recursive algorithms?
  - **Back Substitution**
  - Recursion Tree
  - Master Theorem

Recursive Algorithms (Example #1 – Back Substitution)

Finding the time complexity of the *Sum* function can then be reduced to solving the recurrence relation. We want to find T(n) in terms of n (i.e. not T(n-1)).

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

From equation (2), we can write:

- T(n-1) = T(n-2) + 1 (3)
- T(n-2) = T(n-3) + 1 (4)
- T(n-3) = T(n-4) + 1 (5)

Next, using substitution:

T(n) = T(n-1) + 1 = T(n-2) + 1 + 1 = T(n-2) + 2 sub (3) into (2) to give (6)

T(n) = T(n-2) + 2 = T(n-3) + 1 + 2 = T(n-3) + 3 sub (4) into (6) to give (7)

Recursive Algorithms (Example #1 – Back Substitution)

Continuing the substitution, and remembering from equation (2), we wrote:

- $T(n-1) = T(n-2) + 1$ (3)
- $T(n-2) = T(n-3) + 1$ (4)
- $T(n-3) = T(n-4) + 1$ (5)

Next, using substitution:

- $T(n) = T(n-1) + 1 = T(n-2) + 1 + 1 = T(n-2) + 2$ sub (3) into (2) to give (6)
- $T(n) = T(n-2) + 2 = T(n-3) + 1 + 2 = T(n-3) + 3$ sub (4) into (6) to give (7)
- $T(n) = T(n-3) + 3 = T(n-4) + 1 + 3 = T(n-4) + 4$ sub (5) into(7) to give (8)

Notice if we continue the substitution, in general we can write

- **$T(n) = T(n-k) + k$** (where k is a constant) (9)

Recursive Algorithms (Example #1 – Back Substitution)

Notice if we continue the substitution, in general we can write

- **T(n) = T(n-k) + k** (9)

Next: we can use equation (1) of our recurrence relation

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

From this, we know the value of T(1)=1

  - In (9) we want (n-k)=1, so we can use T(1)=1
  - (n-k)=1 can be rearranged to give k = n-1

Substituting k=(n-1) into (9) gives:

  - T(n) = T(n-k) + k = T(1) + n-1 = 1+n-1 = n
  - <u>So, T(n) = n which means the algorithm is θ(n)</u>

Recursive Algorithms (Example #2 - Back Substitution)

The recursive **`binarySearch`** function can be reduced to solving the recurrence relation:

- $T(n) = T(\frac{n}{2}) + 1$ when $n > 1$
- $T(1) = 1$

Can you use back substitution to find the time complexity of the recursive **`binarySearch`** function?

Recursive Algorithms (Example #2 - Back Substitution)

The recursive **`binarySearch`** function can be reduced to solving the recurrence relation:

- $T(n) = T(\frac{n}{2}) + 1$ when $n > 1$
- $T(1) = 1$

From the recurrence relation, we can write:

- $T(\frac{n}{2}) = T(\frac{n}{4}) + 1$
- $T(\frac{n}{4}) = T(\frac{n}{8}) + 1$
- $T(\frac{n}{8}) = T(\frac{n}{16}) + 1$

Next, substitution…

- $T(n) = T(\frac{n}{2}) + 1 = T(\frac{n}{4}) + 1 + 1 = T(\frac{n}{4}) + 2$
- $T(n) = T(\frac{n}{4}) + 2 = T(\frac{n}{8}) + 2 + 1 = T(\frac{n}{8}) + 3$
- $T(n) = T(\frac{n}{8}) + 3 = T(\frac{n}{16}) + 3 + 1 = T(\frac{n}{16}) + 4$

Recursive Algorithms (Example #2 - Back Substitution)

From substitution, we have

- T(n) = T($\frac{n}{4}$) + 2
- T(n) = T($\frac{n}{8}$) + 3
- T(n) = T($\frac{n}{16}$) + 4

We can write in terms of a constant k

- T(n) = T($\frac{n}{4}$) + 2 = T($\frac{n}{2^2}$) + 2
- T(n) = T($\frac{n}{8}$) + 3 = T($\frac{n}{2^3}$) + 3
- T(n) = T($\frac{n}{16}$) + 4 = T($\frac{n}{2^4}$) + 4

- **T($\boldsymbol{n}$) = T($\boldsymbol{\frac{n}{2^k}}$) + $\boldsymbol{k}$**

We already know T(1) = 1

So, $\frac{n}{2^k}$=1 means $n = 2^k$ or, $k = \log_2 n$

So, we can write,

T(n) = T($\frac{n}{2^k}$) + $\log_2 n$

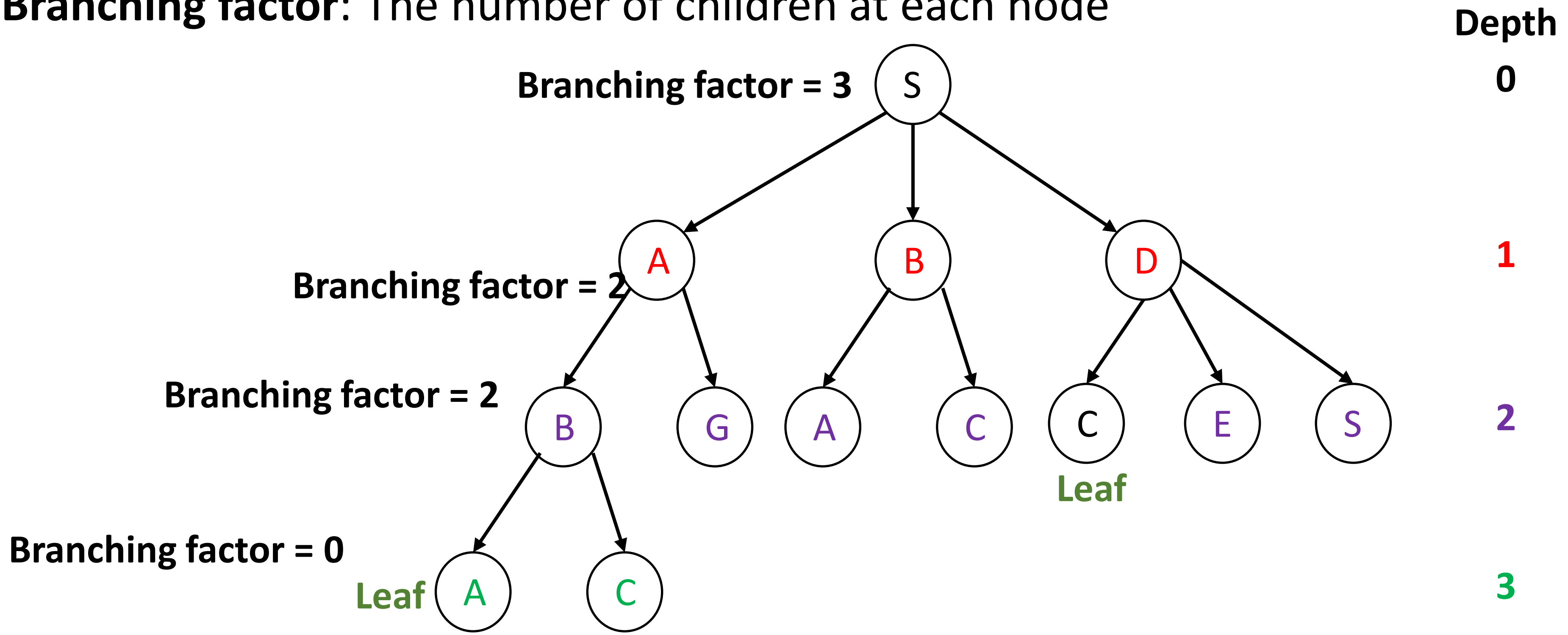
So, the algorithm is $\boldsymbol{\theta(\log(n))}$

Outline

- What are Recursive Algorithms?
- Examples of Recursive Algorithms
- How to evaluate the time complexity of Recursive algorithms?
  - Back Substitution
  - **Recursion Tree**
  - Master Theorem

Recap from last Term: Trees

- Trees are special cases of graphs having **no loops**; having **only one path** between any two vertices (vertices are also called nodes)
- **Depth of tree**: Depth of the deepest node
- **Branching factor**: The number of children at each node



Recursion Tree

- A *recursion tree* is useful for visualizing what happens when a recurrence is iterated.
- It diagrams the tree of recursive calls and the amount of work done at each call.

Recursion Tree (Example#1)

Given the following recurrence relation

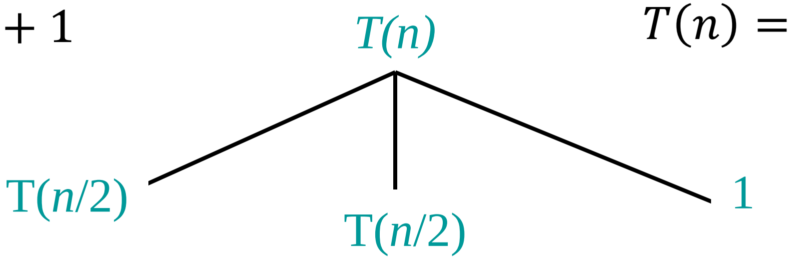
- $T(1) = 1$
- $T(n) = 2T\left(\frac{n}{2}\right) + 1$

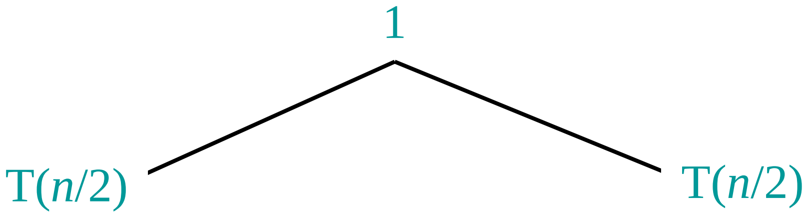
We can use a recursion tree to find the time complexity…

Recursion Tree (Example#1)

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

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





Recursion Tree (Example#1)

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

1

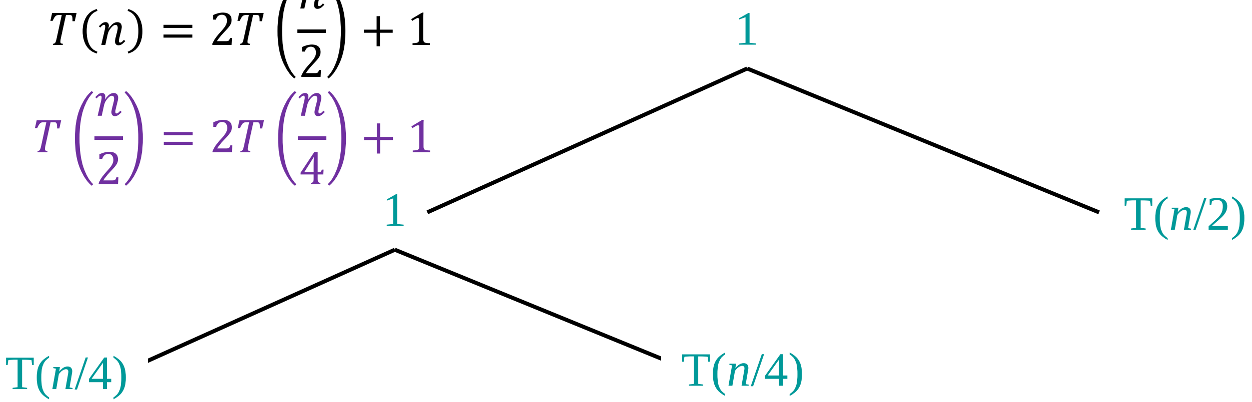
T(n/2) T(n/2)

$$T\left(\frac{n}{2}\right) = 2T\left(\frac{n}{4}\right) + 1$$

Recursion Tree (Example#1)

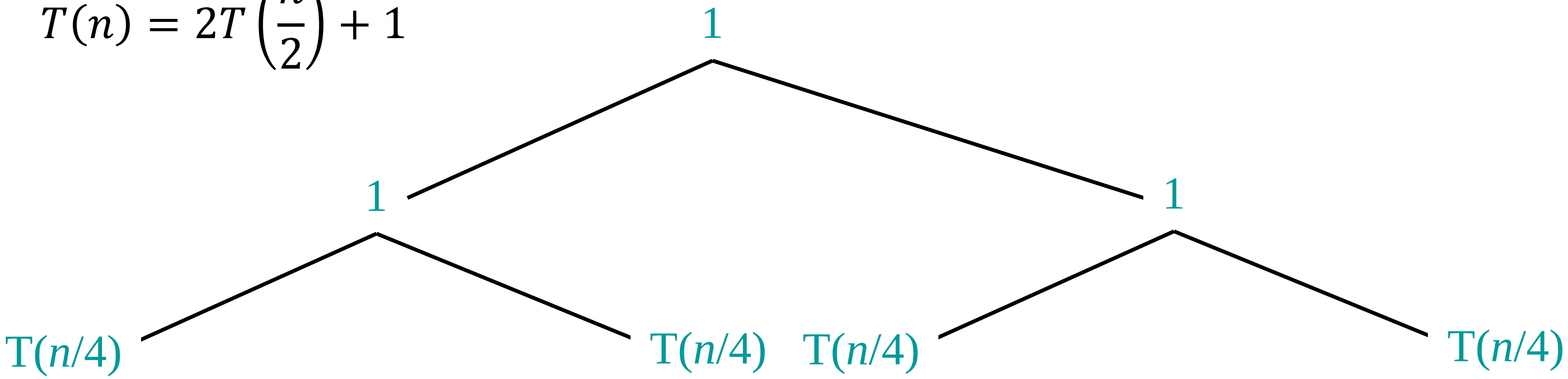
$$T(n) = 2T\left(\frac{n}{2}\right) + 1$$

$$T\left(\frac{n}{2}\right) = 2T\left(\frac{n}{4}\right) + 1$$



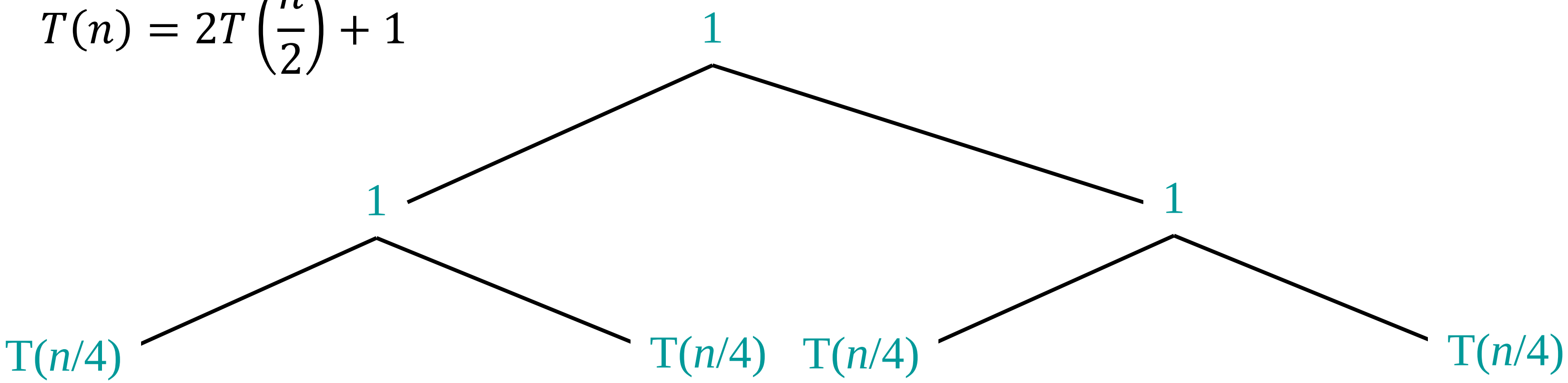
Recursion Tree (Example#1)

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



Recursion Tree (Example#1)

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

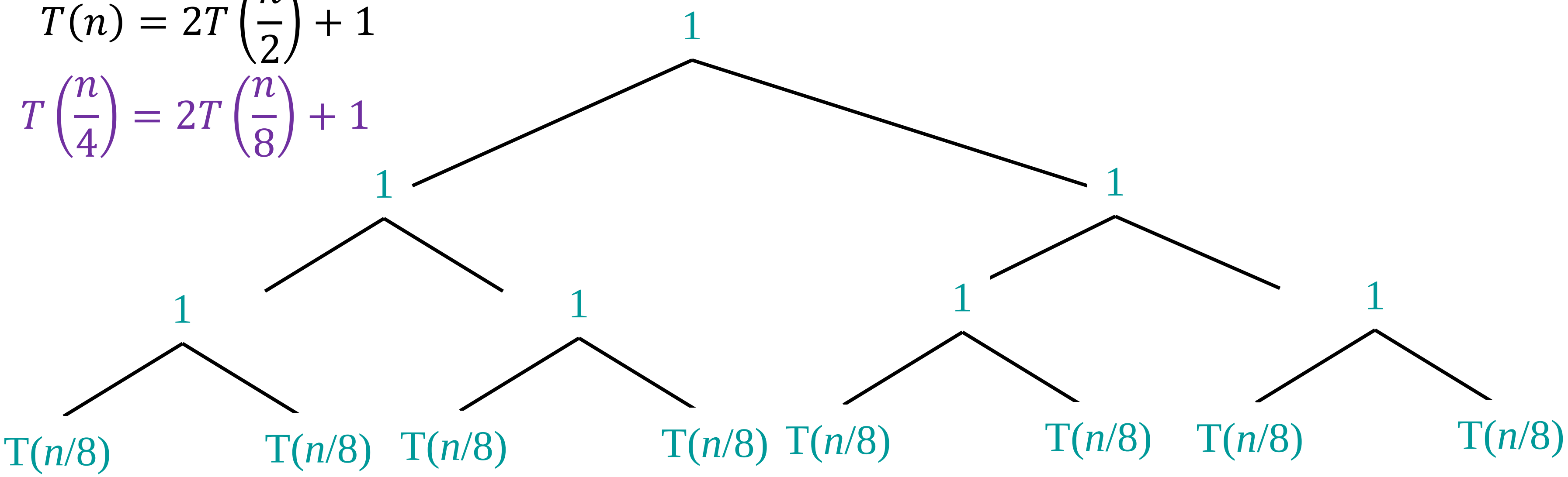


$$T\left(\frac{n}{4}\right) = 2T\left(\frac{n}{8}\right) + 1$$

Recursion Tree (Example#1)

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

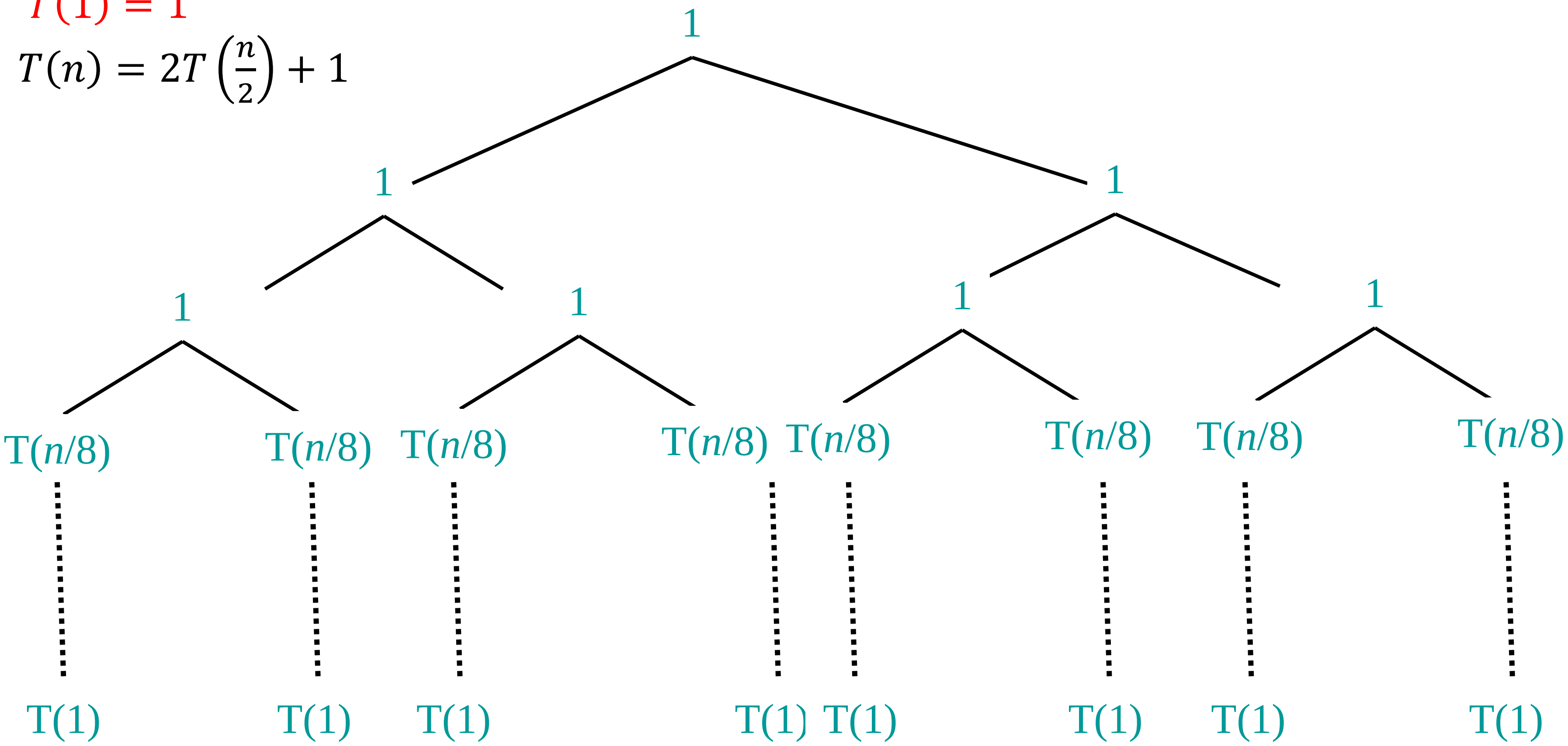
$$T\left(\frac{n}{4}\right) = 2T\left(\frac{n}{8}\right) + 1$$



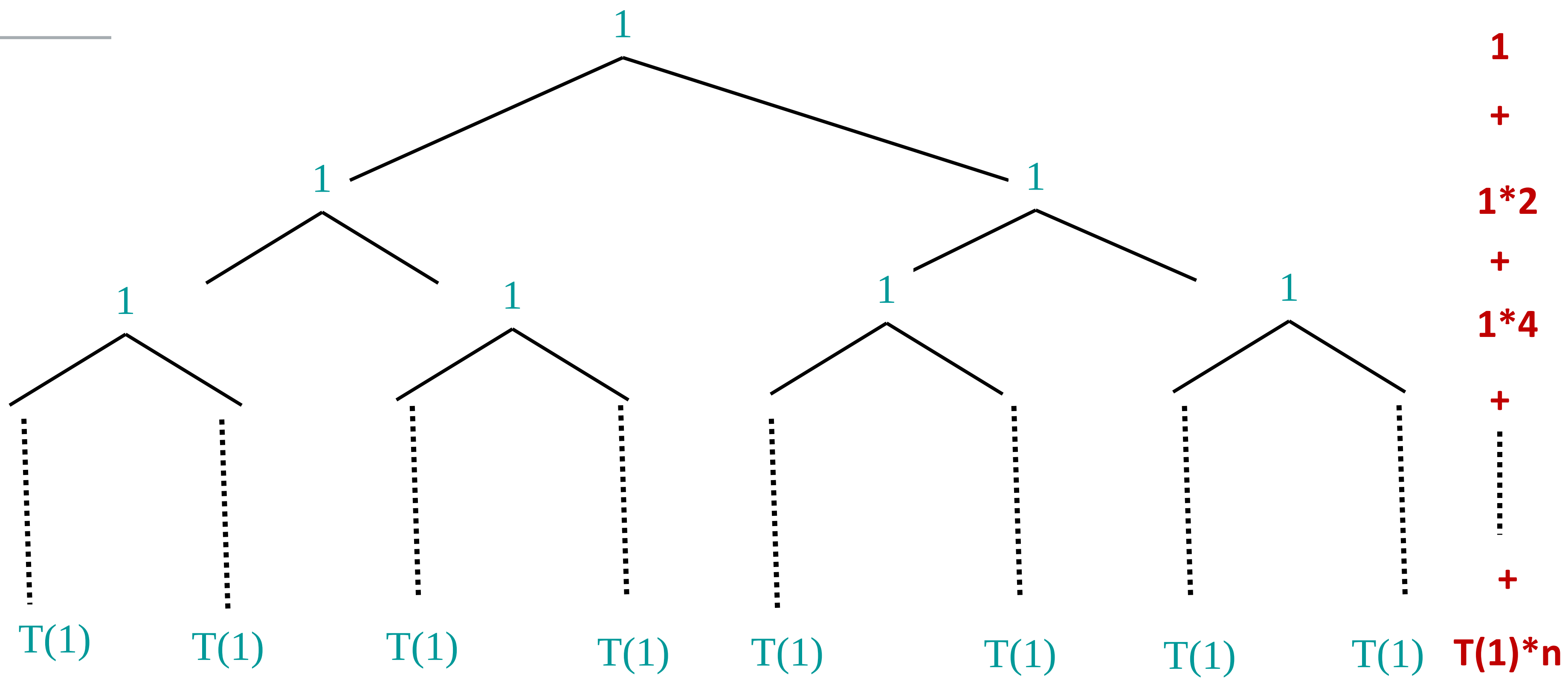
Recursion Tree (Example#1)

$T(1) = 1$

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



Recursion Tree (Example#1)



- Full binary tree (every node has two children)
- For n leaves, number of internal nodes = n-1
- Total nodes = n leaves + n-1 internal nodes = 2n-1
- Which means algorithm is $\theta(n)$

Aside: Recursion tree example using back substitution

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

So we have

- $T\left(\frac{n}{2}\right) = 2T\left(\frac{n}{4}\right)$+1
- $T\left(\frac{n}{4}\right) = 2T\left(\frac{n}{8}\right)$+1
- $T\left(\frac{n}{8}\right) = 2T\left(\frac{n}{16}\right)$+1

Now, substitution

- $T(n) = 2\left(2T\left(\frac{n}{4}\right) + 1\right) + 1 = 4T\left(\frac{n}{4}\right) + 3$
- $T(n) = 4\left(2T\left(\frac{n}{8}\right) + 1\right) + 3 = 8T\left(\frac{n}{8}\right) + 7$

In general

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

Want
$\left(\frac{n}{k}\right)$=1 as we know T(1)
So k=n
$T(n) = nT(1) + n - 1$
$T(n) = 2n - 1$
Which means algorithm is
θ(n)

Outline

- What are Recursive Algorithms?
- Examples of Recursive Algorithms
- How to evaluate the time complexity of Recursive algorithms?
  - Back Substitution
  - Recursion Tree
  - **Master Theorem**

Master Theorem

- Let T(n) be a monotonically increasing function that satisfies

$$T(n) = a\,T\left(\frac{n}{b}\right) + f(n)$$

$$T(1) = c$$

where $a \geq 1, b \geq 2, c > 0$ and $f(n)$ is $\Theta(n^d)$ where $d \geq 0$ then

$$T(n) = \begin{cases} \Theta(n^d) & if\ a < b^d \\ \Theta(n^d \log n) & If\ a = b^d \\ \Theta(n^{\log_b a}) & if\ a > b^d \end{cases}$$

Masters Theorem

- Let T(n) be <u>a monotonically increasing</u> function that satisfies

$$T(n) = a\,T\left(\frac{n}{b}\right) + f(n)$$
$$T(1) = c$$

- You cannot use the Master Theorem if
  - T(n) is not monotone, e.g. T(n) = sin(x)
  - f(n) is not a polynomial, e.g., T(n)=2T(n/2)+2^n
  - b cannot be expressed as a constant, e.g.

$$T(n) = T(\sqrt{n})$$

Master Theorem (Example#1)

- Let $T(n) = T(\frac{n}{2}) + ½\, n^2 + n$. What are the parameters?

  a = 1

  b = 2

  d = 2

$$f(n) = ½\, n^2 + n \in \Theta(n^2)$$

$$T(n) = a\,T\left(\frac{n}{b}\right) + f(n)$$

where $a \geq 1, b \geq 2, c > 0$ and $f(n)$ is $\Theta(n^d)$ where $d \geq 0$

Master Theorem (Example#1)

- Let $T(n) = T(\frac{n}{2}) + ½\, n^2 + n$. What are the parameters?

  a = 1

  b = 2

  d = 2

- Therefore, which condition applies?

$$T(n) = \begin{cases} \Theta(n^d) & if\ a < b^d \\ \Theta(n^d \log n) & If\ a = b^d \\ \Theta(n^{\log_b a}) & if\ a > b^d \end{cases}$$

$1 < 2^2$

$1 = 2^2$

$1 > 2^2$

Master Theorem (Example#1)

- Let $T(n) = T(\frac{n}{2}) + ½\, n^2 + n$. What are the parameters?

  a = 1

  b = 2

  d = 2

- Therefore, which condition applies?
  - $1 < 2^2$, case 1 applies
  - We conclude that
    - $T(n) \in \Theta(n^d) = \Theta\,(n^2)$

Master Theorem (Example#2)

- Let $T(n) = 2T(\frac{n}{4}) + \sqrt{n} + 42$. What are the parameters?

  a = 2

  b = 4

  d = $\frac{1}{2}$

$$f(n) = \sqrt{n} + 42 \in \Theta(\sqrt{n}) = \Theta(n^{\frac{1}{2}})$$

$$T(n) = a\,T\left(\frac{n}{b}\right) + f(n)$$

where $a \geq 1, b \geq 2, c > 0$ and $f(n)$ is $\Theta(n^d)$ where $d \geq 0$

Master Theorem (Example#2)

- Let $T(n) = 2T(\frac{n}{4}) + \sqrt{n} + 42.$ What are the parameters?

  a = 2

  b = 4

  d = $\frac{1}{2}$

- Therefore, which condition applies?

$$T(n) = \begin{cases} \Theta(n^d) & if\ a < b^d \\ \Theta(n^d \log n) & If\ a = b^d \\ \Theta(n^{\log_b a}) & if\ a > b^d \end{cases}$$

$2 < 4^{\frac{1}{2}} = \sqrt{4} = 2$

$2 = 4^{\frac{1}{2}} = \sqrt{4} = 2$

$2 > 4^{\frac{1}{2}} = \sqrt{4} = 2$

Master Theorem (Example#2)

- Let $T(n) = 2T(\frac{n}{4}) + \sqrt{n} + 42.$ What are the parameters?

  a = 2

  b = 4

  d = $\frac{1}{2}$

- Therefore, which condition applies?
  - $2 = 4^{\frac{1}{2}}$, case 2 applies
  - We conclude that
    - T(n) $\in \Theta(n^d \log n)$ = $\Theta\ (\sqrt{n} \log n)$

slido

Please download and install the Slido app on all computers you use

Audience Q&A

ⓘ Start presenting to display the audience questions on this slide.

slido

Please download and install the Slido app on all computers you use

Summer term lectures will focus on theoretical time complexity - P, NP, NP hard and NP complete problems. Some flexibility here (e.g. one lecture not accounted for). What would you prefer to cover here: e.g. exam practice, more on theoretical time complexity, recap of any particular topics? anything else?

ⓘ Start presenting to display the poll results on this slide.

Summary

Today's lecture: Considered how to find the time complexity of recursive algorithms

- A recursive algorithm is an algorithm which calls itself with smaller input values
- How to evaluate the time complexity of Recursive algorithms?
  - Back Substitution (simple substitution)
  - Recursion Tree (use tree to visualize the recurrence, depth of the tree helps solving recurrence relations)
  - Master Theorem (don't need to solve the recurrence relations, but can only use under some conditions)

- **Summer Term Lectures (2 weeks):** A more theoretical look at complexity: P, NP, NP-hard and NP-complete problems
- **Next lectures (week 16-21):** Abstract Data Types with Dr Fabien Dufoulon