UNIT 1:
Fundamentals of the Analysis of Algorithmic Efficiency…

Mathematical Analysis of Recursive Algorithms

General plan for analyzing efficiency of recursive algorithms

1. Decide on parameter/s ***n*** indicating input's size
2. Identify algorithm’s basic operation
3. Check/Determine worst, average, and best cases for input of size n
4. Set up a recurrence relation, with an appropriate initial condition, for the number of times the basic operation is executed.
5. Solve the recurrence or at least ascertain the order of growth of its solution.

Example 1: Factorial function

Example 1: Factorial function

```
ALGORITHM $F(n)$
    //Computes $n!$ recursively
    //Input: A nonnegative integer $n$
    //Output: The value of $n!$
    if $n = 0$
        return 1
    else
        return $F(n - 1) * n$
```

Example 1: Factorial function
Algorithm analysis

1. input's size: **n**
2. basic operation: **multiplication**

```
return $F(n-1) * n$
```

3. **No worst, average, and best cases**
4. Let M(n) = number of times the basic operation is executed.

Since the function F(n) is computed according to the formula

$$F(n) = F(n-1) * n \text{ for } n > 0,$$
$$F(0) = 1$$

We get the recurrence relation:

$$M(n) = \underbrace{M(n-1)}_{\substack{\text{to compute}\\ F(n-1)}} + \underbrace{1}_{\substack{\text{to multiply}\\ F(n-1) \text{ by } n}} \quad \text{for } n > 0.$$

i.e.,

$$M(n) = M(n-1) + 1 \quad \text{for } n > 0,$$
$$M(0) = 0.$$

Using method of backward substitutions:

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

substitute $M(n-1) = M(n-2) + 1$

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

substitute $M(n-2) = M(n-3) + 1$

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

$$= \cdots$$

$$= M(n-i) + i$$

$$= \cdots$$

$$= M(n-n) + n$$

$$= n. \in \Theta(n)$$

$M(n) = M(n-1) + 1 \quad \text{for } n > 0,$
$M(0) = 0.$

Example 2: Tower of Hanoi

Example 2: Tower of Hanoi

```
ALGORITHM towerOfHanoi(n, src, dest, aux)
   if (n == 1)
          Move disk from src to dest
          return
   else
          towerOfHanoi(n-1, src, aux, dest);
          Move nth disk from src to dest
          towerOfHanoi(n-1, aux, dest, src);
}
```

Example 2: Tower of Hanoi
Algorithm analysis

1. input's size: **n – number of disks**
2. basic operation: **moving one disk**
3. **No worst, average, and best cases**
4. Let M(n) = total number of moves
5. The recurrence equation M(n),

$$M(n) = M(n-1) + 1 + M(n-1) \quad \text{for } n > 1$$

$$M(1) = 1$$

$$M(n) = 2M(n-1) + 1 \quad \text{for } n > 1,$$
$$M(1) = 1.$$

Using method of backward substitutions:

$M(n) = 2M(n-1) + 1$ sub. $M(n-1) = 2M(n-2) + 1$

$= 2[2M(n-2) + 1] + 1 = 2^2 M(n-2) + 2 + 1$ sub. $M(n-2) = 2M(n-3) + 1$

$= 2^2[2M(n-3) + 1] + 2 + 1 = 2^3 M(n-3) + 2^2 + 2 + 1.$

$= 2^4 M(n-4) + 2^3 + 2^2 + 2 + 1$

$= 2^i M(n-i) + 2^{i-1} + 2^{i-2} + \cdots + 2 + 1$

$= 2^i M(n-i) + 2^i - 1.$

for $i = n - 1$.

$M(n) = 2^{n-1} M(n-(n-1)) + 2^{n-1} - 1$

$= 2^{n-1} M(1) + 2^{n-1} - 1 = 2^{n-1} + 2^{n-1} - 1 = 2^n - 1 \quad \in \Theta(2^n)$

Let’s check our understanding…

Algorithm computes the sum of the first *n* cubes:

$S(n) = 1^3 + 2^3 + \ldots + n^3$

```
ALGORITHM  S(n)
    //Input: A positive integer n
    //Output: The sum of the first n cubes
    if n = 1 return 1
    else return S(n − 1) + n ∗ n ∗ n
```

a. What does the algorithm computes?
b. Set up and solve a recurrence relation for the number of times the algorithm’s basic operation is executed.

$M(n) = M(n-1) + 2$, for $n > 1$

$M(1) = 0$

Divide and Conquer

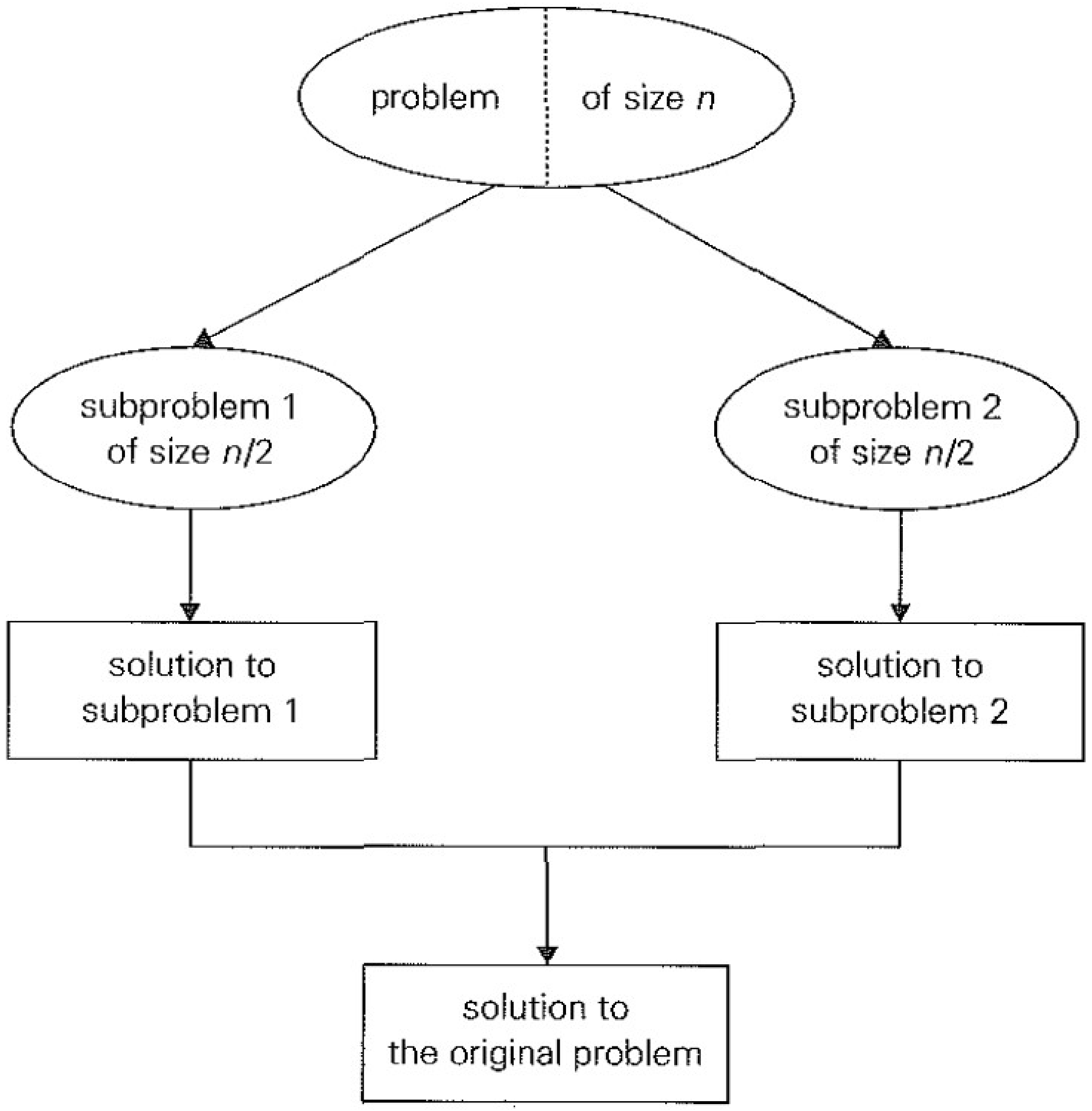
Best-known general algorithm design technique.

Ideally suited for parallel computations, in which each subproblem can be solved simultaneously by its own processor.

Divide and Conquer: General plan

1. A problem's instance is divided into several smaller instances of the same problem, ideally of about the same size.
2. The smaller instances are solved (typically recursively, though sometimes a different algorithm is employed when instances become small enough).
3. If necessary, the solutions obtained for the smaller instances are combined to get a solution to the original instance.

Divide-and-conquer technique (typical case)


problem
of size n
subproblem 1 of size $n/2$
subproblem 2 of size $n/2$
solution to subproblem 1
solution to subproblem 2
solution to the original problem

A problem's instance of size n is divided into two instances of size n/2.

General divide-and-conquer recurrence

- An instance of size **n** can be divided into **b** instances of size **n/b**, with **a** of them needing to be solved.

  (Here, a and b are constants; $a \geq 1$ and $b > 1$).

- Assuming that size **n is a power of b**, the recurrence for the running time T(n) is:

$$T(n) = a\,T(n/b) + f(n)$$

where f(n) is a function that accounts for the time spent on dividing the problem into smaller ones and on combining their solutions.

The order of growth of its solution T(n) depends on the values of the constants a and b and the order of growth of the function f(n)

Master theorem

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

If $f(n) \in \Theta(n^d)$ with $d \geq 0$ in recurrence equation then

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

(Analogous results hold for the O and Ω notations, too.)

1. Solve the recurrence using Master method

$$A(n) = 2A(n/2) + 1$$

Solution:

Here, $a=2$, $b=2$, $f(n)=1$; i.e., $\Theta(n^d) \Rightarrow d=0$

$$a > b^d$$

$$2 > 2^0$$

Case 3 holds good.

$$\boxed{T(n) \in \Theta(n^{\log_b a}) \qquad \text{if } a > b^d}$$

$$A(n) = \Theta(n^{\log_b a}) = \Theta(n^{\log_2 2}) = \Theta(n)$$

2. Solve the recurrence using Master method

$$T(n) = 4T(n/2) + n^2,\ T(1) = 1$$

Solution:

Here, $a=4$, $b=2$, $f(n)=n^2$; i.e., $\Theta(n^d) \Rightarrow d=2$

$$a = b^d$$

$$4 = 2^2$$

Case 2 holds good.

$$\boxed{T(n) \in \Theta(n^d \log n) \quad \text{if } a = b^d}$$

$T(n) = \Theta(n^d log\ n) = \Theta(n^2 \log n)$

Let’s check our understanding…

Solve the recurrence using Master method

$$T(n) = 16T(n/4) + n$$

$$\boxed{T(n) = \Theta(n^2) \quad \text{(Case 1)}}$$

When Master theorem cannot be applied?

Does not apply

$T(n) = 2^n T(n/2) + n^n$ a is not constant

$T(n) = 0.5T(n/2) + 1/n$ $a < 1$

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

non-polynomial difference between $f(n)$ and $n^{\log_b a}$

$T(n) = 64T(n/8) - n^2 \log n$

$f(n)$ is not positive

Next session…

Divide and Conquer…

Merge sort, Quicksort