CSC 411
Design and Analysis of Algorithms

Chapter 2 Fundamentals of the Analysis of Algorithm Efficiency

- Part 2

Instructor: Minhee Jun
junm@cua.edu

Common time complexities

BETTER

$O(1)$	constant time
$O(\log n)$	log time
$O(n)$	linear time
$O(n \log n)$	log linear time
$O(n^2)$	quadratic time
$O(n^3)$	cubic time
$O(2^n)$	exponential time

WORSE

Math you need to review

Logarithms and Exponents

- **properties of logarithms:**

  $\log_b(xy) = \log_b x + \log_b y$

  $\log_b (x/y) = \log_b x - \log_b y$

  $\log_b x^a = a\log_b x$

  $\log_b a = \log_x a/\log_x b$

- **properties of exponentials**:

  $a^{(b+c)} = a^b a^c$

  $a^{bc} = (a^b)^c$

  $a^b /a^c = a^{(b-c)}$

  $b = a^{\log_a b}$

  $b^c = a^{c*\log_a b}$

Non-Recursive Programming Method

- A **non-recursive algorithm** does the sorting all at once, without calling itself.
- Let sum up squares from n to m (m>=n):
  - SumS(n,m) = $n^2 + (n+1)^2 + (n+2)^2 + \ldots + m^2$
  - A none recursive version (iterative):

```
public int SumS ( int n, int m )
{
    int i, sum=1; // i: counter, sum: to hold result

    for ( i = n; i <=m ; i++) // loop: n to m
        sum + = i * i; // (n*n) + (n+1)*(n+1) + … + m*m

    return sum; // returns the result
}
```

Recursive Method

- A **recursive method** that it calls itself.
  - In other words: A method that contains a method call with the same name and signature of that method

```
public int SumS (int n, int m )
{
    if (n == m ) // Stop when n reaches m
        return m * m; // and return last squared sum

    else // multiply n by n and add the result of the
        return (n*n) + SumS (n+1,m); // next sum (n+1)
}
// important note : n increase by 1 at each call
//                              until it reaches m
```

2.3 Mathematical Analyze of Non-recursive Algorithms

- General Plan for Analysis
  1. Decide on parameter *n* indicating *input size*
  2. Identify algorithm's *basic operation*
  3. Check whether the number of times the basic operation is executed depending on the size of input.
  4. Set up a sum for the number of times the basic operation is executed
  5. Using standard formulas and rules of sum manipulation, either find a closed form formula for the count or establish its order of growth.

Useful summation formulas and rules

$\Sigma_{l \le i \le u} 1 = 1+1+ \cdots +1 = u - l + 1$

In particular, $\Sigma_{1 \le i \le u} 1 = n - 1 + 1 = n \in \Theta(n)$

$\Sigma_{1 \le i \le n}\, i = 1+2+ \cdots +n = n(n+1)/2 \approx n^2/2 \in \Theta(n^2)$

$\Sigma_{1 \le i \le n}\, i^2 = 1^2+2^2+ \cdots +n^2 = n(n+1)(2n+1)/6 \approx n^3/3 \in \Theta(n^3)$

$\Sigma_{0 \le i \le n}\, a^i = 1 + a + \cdots + a^n = (a^{n+1} - 1)/(a - 1)$ for any $a \neq 1$

In particular, $\Sigma_{0 \le i \le n}\, 2^i = 2^0 + 2^1 + \cdots + 2^n = 2^{n+1} - 1 \in \Theta(2^n)$

$\Sigma(a_i \pm b_i) = \Sigma a_i \pm \Sigma b_i \qquad \Sigma c a_i = c \Sigma a_i \qquad \Sigma_{l \le i \le u} a_i = \Sigma_{l \le i \le m} a_i + \Sigma_{m+1 \le i \le u} a_i$

Exercise 2.3

1. Compute the following sums.

a. $1 + 3 + 5 + 7 + \cdots + 999$

b. $2 + 4 + 8 + 16 + \cdots + 1024$

c. $\sum_{i=3}^{n+1} 1$ **d.** $\sum_{i=3}^{n+1} i$ **e.** $\sum_{i=0}^{n-1} i(i+1)$

f. $\sum_{j=1}^{n} 3^{j+1}$ **g.** $\sum_{i=1}^{n} \sum_{j=1}^{n} ij$ **h.** $\sum_{i=1}^{n} 1/i(i+1)$

Example 1: Maximum element

Find the value of the largest element in a list of n numbers.

ALGORITHM $MaxElement(A[0..n-1])$
  //Determines the value of the largest element in a given array
  //Input: An array $A[0..n-1]$ of real numbers
  //Output: The value of the largest element in A
  $maxval \leftarrow A[0]$
  for $i \leftarrow 1$ **to** $n-1$ **do**
    if $A[i] > maxval$
      $maxval \leftarrow A[i]$
  return $maxval$

What is Θ of the algorithm?

Example 1: Maximum element

ALGORITHM *MaxElement*$(A[0..n-1])$

```
//Determines the value of the largest element in a given array
//Input: An array A[0..n − 1] of real numbers
//Output: The value of the largest element in A
maxval ← A[0]
for i ← 1 to n − 1 do
    if A[i] > maxval
        maxval ← A[i]
return maxval
```

- One comparison executed on each loop

$$C(n) = \sum_{i=1}^{n-1} 1 = n - 1 \in \Theta(n)$$

Example 2: Element uniqueness problem

Check whether all the elements in a given array are distinct

ALGORITHM *UniqueElements*($A[0..n-1]$)

//Determines whether all the elements in a given array are distinct
//Input: An array $A[0..n-1]$
//Output: Returns "true" if all the elements in A are distinct
//   and "false" otherwise
for $i \leftarrow 0$ **to** $n-2$ **do**
    for $j \leftarrow i+1$ **to** $n-1$ **do**
        if $A[i] = A[j]$ **return false**
return true

What is Θ of the algorithm?

Example 2: Element uniqueness problem

ALGORITHM *UniqueElements*$(A[0..n-1])$
//Determines whether all the elements in a given array are distinct
//Input: An array $A[0..n-1]$
//Output: Returns "true" if all the elements in A are distinct
// and "false" otherwise
for $i \leftarrow 0$ **to** $n-2$ **do**
  for $j \leftarrow i+1$ **to** $n-1$ **do**
    if $A[i] = A[j]$ **return false**
return true

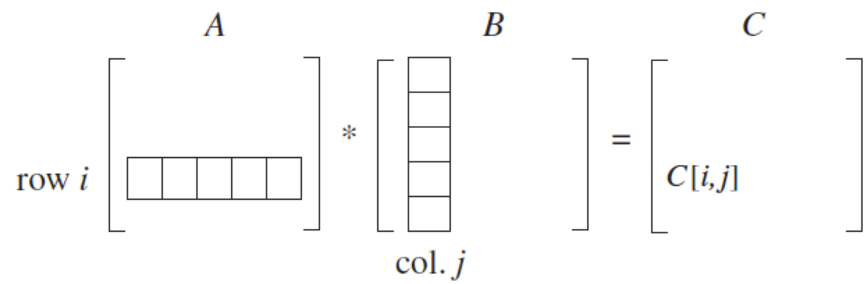
- One comparison is made for each repetition of the innermost loop
  - For each loop variable j between its limits $i+1$ and $n-1$
- This is repeated for each value of the outer loop
  - For each loop variable I between its limits 0 and $n-2$

$$C_{worst}(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \frac{(n-1)n}{2} \approx \frac{1}{2}n^2 \in \Theta(n^2)$$

Example 3: Matrix multiplication

Given two n-by-n matrices A and B, compute their product

```
ALGORITHM MatrixMultiplication(A[0..n − 1, 0..n − 1], B[0..n − 1, 0..n − 1])
    //Multiplies two n-by-n matrices by the definition-based algorithm
    //Input: Two n-by-n matrices A and B
    //Output: Matrix C = AB
    for i ← 0 to n − 1 do
        for j ← 0 to n − 1 do
            C[i, j] ← 0.0
            for k ← 0 to n − 1 do
                C[i, j] ← C[i, j] + A[i, k] ∗ B[k, j]
    return C
```



What is Θ of the algorithm?

Example 3: Matrix multiplication

ALGORITHM *MatrixMultiplication*$(A[0..n-1, 0..n-1], B[0..n-1, 0..n-1])$

```
//Multiplies two n-by-n matrices by the definition-based algorithm
//Input: Two n-by-n matrices A and B
//Output: Matrix C = AB
for i ← 0 to n − 1 do
    for j ← 0 to n − 1 do
        C[i, j] ← 0.0
        for k ← 0 to n − 1 do
            C[i, j] ← C[i, j] + A[i, k] ∗ B[k, j]
return C
```

- The total number of multiplications $M(n)$:

$$M(n) = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} \sum_{0}^{n-1} 1 = n^3$$

- If we took into account the time spent on the addition, too

$$T(n) \approx c_m M(n) + c_a A(n) = c_m n^3 + c_a n^3 = (c_m + c_a) n^3 \in \Theta(n^3)$$

Example 4: Counting binary digits

- Find the #(binary digits) in the binary representation of a positive decimal integer

ALGORITHM *Binary*(n)
  //Input: A positive decimal integer n
  //Output: The number of binary digits in n's binary representation
  $count \leftarrow 1$
  while $n > 1$ **do**
    $count \leftarrow count + 1$
    $n \leftarrow \lfloor n/2 \rfloor$
  return $count$

What is Θ of the algorithm?

Example 4: Counting binary digits

ALGORITHM *Binary*(n)
  //Input: A positive decimal integer n
  //Output: The number of binary digits in n's binary representation
  $count \leftarrow 1$
  while $n > 1$ **do**
    $count \leftarrow count + 1$
    $n \leftarrow \lfloor n/2 \rfloor$
  return *count*

- Since the value of n is about halved on each repetition of the loop, the answer should be about $\log_2 n$
- The number of comparison $C(n)$ is actually,

$$C(n) = \lfloor \log_2 n \rfloor + 1$$

Exercise 2.3

4. Consider the following algorithm.

ALGORITHM *Mystery*(n)

//Input: A nonnegative integer n
$S \leftarrow 0$
for $i \leftarrow 1$ **to** n **do**
    $S \leftarrow S + i * i$
return S

a. What does this algorithm compute?

b. What is its basic operation?

c. How many times is the basic operation executed?

d. What is the efficiency class of this algorithm?

e. Suggest an improvement, or a better algorithm altogether, and indicate its efficiency class. If you cannot do it, try to prove that, in fact, it cannot be done.

2.4 Mathematical Analysis of Recursive Algorithms

- General Plan for Analysis
  1. Decide on a parameter indicating an input's size.
  2. Identify the algorithm's basic operation.
  3. Check whether the number of times the basic operation is executed may vary on different inputs of the same size.
  4. Set up a recurrence relation with an appropriate initial condition expressing the number of times the basic op. is executed.
  5. Solve the recurrence by backward substitutions or another method (or, at the very least, establish its solution's order of growth)

Solving Recurrence Relations

- In evaluating the summation one or more of the following summation formulae may be used:
  - Arithmetic series:

$$\sum_{k=1}^{n} k = 1 + 2 + 3 + \cdots + n = \frac{n(n+1)}{2}$$

$$\sum_{k=1}^{n} k^2 = 1^2 + 2^2 + 3^2 + \cdots + n^2 = \frac{n(n+1)(2n+1)}{6}$$

  - Geometric Series:

$$\sum_{k=1}^{n} x^k = 1 + x + x^2 + \cdots + x^n = \frac{x^{n+1} - 1}{x - 1}$$

  - Harmonic Series:

$$\sum_{k=1}^{n} \frac{1}{k} = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n} \approx \ln n$$

$$\sum_{k=1}^{n} \log k \approx n \log n$$

Example 1: Factorial Function $F(n) = n!$

- Definition:

  $n! = 1 \cdot 2 \cdot 3 \cdot \cdots \cdot (n-1) \cdot n$ for $n \geq 1$ and $0! = 1$
- Recursive definition of n!:

  $F(n) = F(n-1) \cdot n$ for $n \geq 1$, and $F(0) = 1$

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

- Size: n
- Basic operation: multiplication
- Recurrence relation: $M(n) = M(n-1) + 1$

Method of Backward substitutions

- Solving the recurrence for $M(n)$:

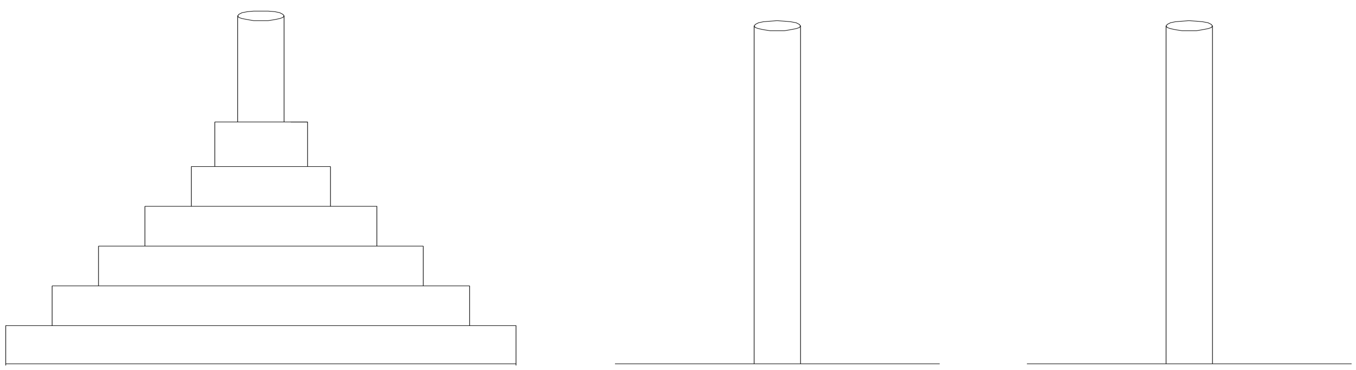
$$M(n) = M(n-1) + 1,\ M(0) = 0$$

- <u>Method of Backward substitutions</u>.

$$\begin{aligned}
M(n) &= M(n-1) + 1 \\
&= (M(n-2) + 1) + 1 = M(n-2) + 2 \\
&= (M(n-3) + 1) + 2 = M(n-3) + 3 \\
&= (M(n-4) + 1) + 3 = M(n-4) + 4 \\
&\\
&= (M(n-(n-1)) + 1) + n - 2 = M(1) + n - 1 \\
&= \cdots = n
\end{aligned}$$

Example 2: The Tower of Hanoi Puzzle

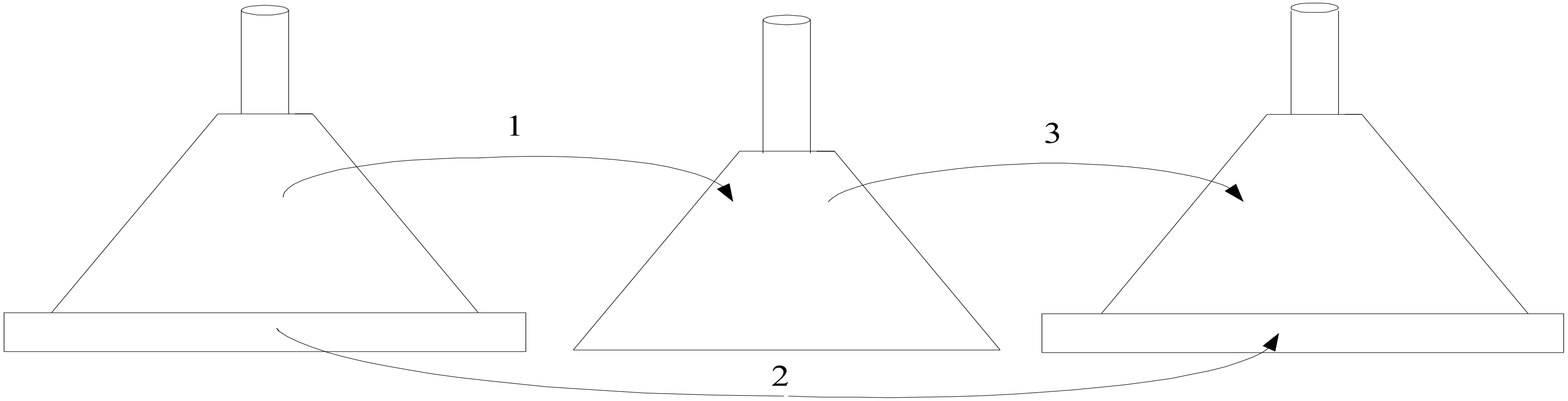
- The Towers of Hanoi is a puzzle made up of three vertical pegs and several disks that slide onto the pegs



- The disks are of varying size, initially placed on one peg with the largest disk on the bottom and increasingly smaller disks on top

Example 2: The Tower of Hanoi Puzzle

- The goal is to move all of the disks from one peg to another following these rules:
  - Only one disk can be moved at a time
  - A disk cannot be placed on top of a smaller disk
  - All disks must be on some peg (except for the one in transit)



Recurrence for number of moves

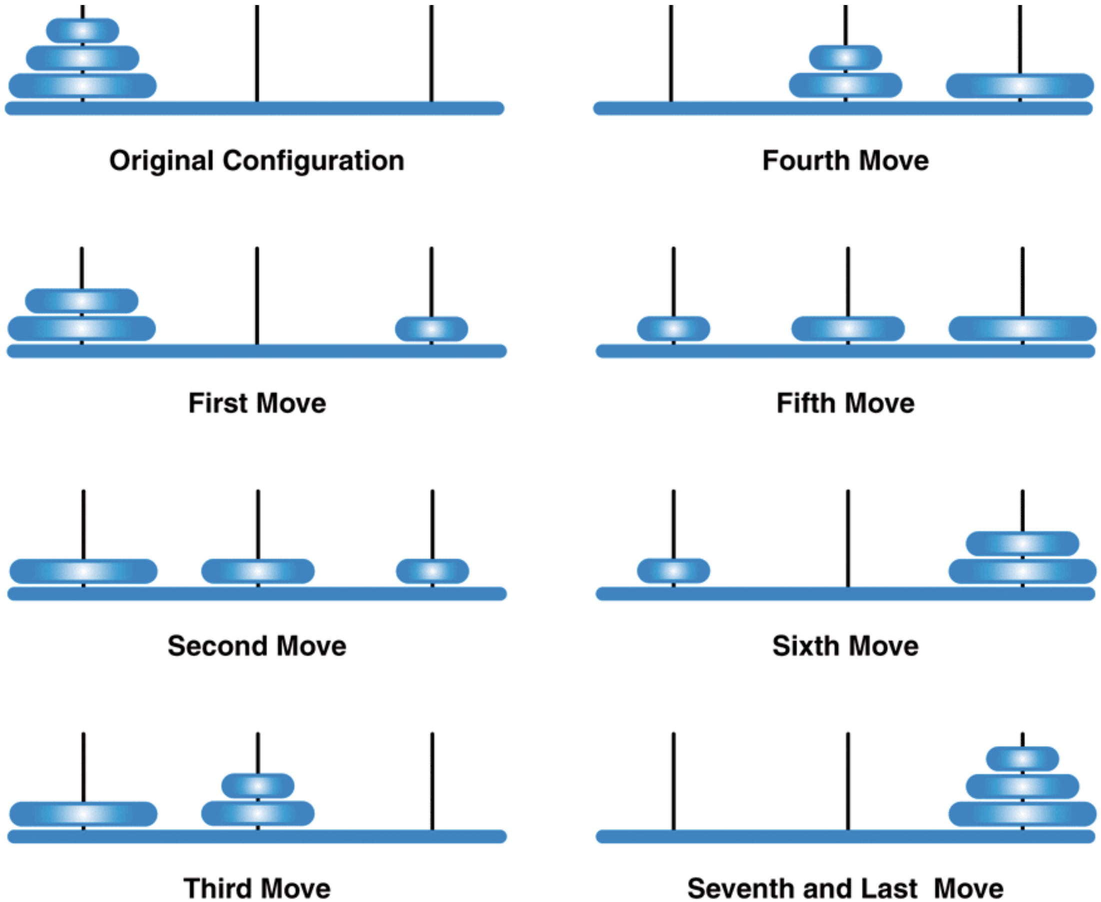


FIGURE 7.7 A solution to the three-disk Towers of Hanoi puzzle

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

Method of Backward substitutions

- Solving the recurrence for $M(n)$:

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

- Method of Backward substitutions.

$$\begin{aligned}
M(n) &= 2M(n-1) + 1 \\
&= 2(2M(n-2) + 1) + 1 = 2^2 M(n-2) + 3 \\
&= 2^2(2M(n-3) + 1) + 3 = 2^3 M(n-3) + 7 \\
&= 2^3(2M(n-4) + 1) + 7 = 2^4 M(n-4) + 15 \\
&= \cdots = 2^i(M(n-i) + 1) + 2^i - 1 \\
&= \cdots = 2^{n-1}(M(n-(n-1)) + 1) + 2^{n-1} - 1 \\
&= 2^n - 1 \in \Theta(2^n)
\end{aligned}$$

Example3: Counting binary digits

- Find the #(binary digits) in the binary representation of a positive decimal integer

ALGORITHM $BinRec(n)$

//Input: A positive decimal integer n
//Output: The number of binary digits in n's binary representation
if $n = 1$ **return** 1
else return $BinRec(\lfloor n/2 \rfloor) + 1$

- Size: n
- Basic operation: addition
- Recurrence relation: $A(n) = A(\lfloor n/2 \rfloor) + 1$

Method of Backward substitutions

- Solving the recurrence for $A(n)$:

$$A(n) = A(\lfloor n/2 \rfloor) + 1, A(1) = 0$$

- Assume $n = 2^k$
- Method of Backward substitutions.

$$\begin{aligned}
A(2^k) = A(2^{k-1}) + 1 & \\
&= (A(2^{k-2}) + 1) + 1 = A(2^{k-2}) + 2 \\
&= (A(2^{k-3}) + 1) + 2 = A(2^{k-3}) + 3 \\
&= \cdots = A(2^0) + k \\
&= k = \log_2 n \in \Theta(\log_2 n)
\end{aligned}$$

Exercise 2.4

1. Solve the following recurrence relations.

a. $x(n) = x(n-1) + 5$ for $n > 1$, $x(1) = 0$

b. $x(n) = 3x(n-1)$ for $n > 1$, $x(1) = 4$

c. $x(n) = x(n-1) + n$ for $n > 0$, $x(0) = 0$

d. $x(n) = x(n/2) + n$ for $n > 1$, $x(1) = 1$ (solve for $n = 2^k$)

e. $x(n) = x(n/3) + 1$ for $n > 1$, $x(1) = 1$ (solve for $n = 3^k$)

Exercise 2.4

2. Set up and solve a recurrence relation for the number of calls made by $F(n)$, the recursive algorithm for computing $n!$.

2.5 Advance Example in Algorithm Analysis

```
int Fib(int n)
{
        if (n <= 1)
            return 1;
        else
            return Fib(n - 1) + Fib(n - 2);
}
```

a strong candidate for the title of *Worst Algorithm in the World*

- $A(0) = 0, A(1) = 0$
- $A(n) = A(n-1) + A(n-2) + 1$

General 2nd order linear homogeneous recurrence with constant coefficients:

Advance Example in Algorithm Analysis

$$F(n) = F(n-1) + F(n-2), \quad F(0) = 0, \;\; F(1) = 1$$

- $A(0) = 0, A(1) = 0$
- $A(n) = A(n-1) + A(n-2) + 1$
- $B(n) = A(n) + 1 \;\rightarrow\; B(n) = B(n-1) + B(n-2)$

What is the order of the algorithm?

- General 2nd order linear homogeneous recurrence in a form of

$$aX(n) + bX(n-1) + cX(n-2) = 0$$

$$B(n) - B(n-1) - B(n-2) = 0$$

Solving $aX(n) + bX(n\text{-}1) + cX(n\text{-}2) = 0$

- To solve a general 2nd order linear homogeneous recurrence in the form of

$$aX(n) + bX(n-1) + cX(n-2) = 0$$

- Set up the characteristic equation (quadratic)

$$ar^2 + br + c = 0$$

- Solve to obtain roots r_1 and r_2

- General solution to the recurrence

  if r_1 and r_2 are two <u>distinct real roots</u>: $X(n) = \alpha r_1^{\,n} + \beta r_2^{\,n}$

  if $r_1 = r_2 = r$ are two <u>equal real roots</u>: $X(n) = \alpha r^n + \beta n r^n$

- Particular solution can be found by using initial conditions

Advance Example in Algorithm Analysis

$$F(n) = F(n-1) + F(n-2), \quad F(0) = 0, \;\; F(1) = 1$$

- $A(0) = 0, A(1) = 0$
- $A(n) = A(n-1) + A(n-2) + 1$
- $B(n) = A(n) + 1 \;\rightarrow\; B(n) = B(n-1) + B(n-2)$

What is the order of the algorithm?

- General 2nd order linear homogeneous recurrence in a form of

$$aX(n) + bX(n-1) + cX(n-2) = 0$$

$$B(n) = \frac{1}{\sqrt{5}}(\phi^{n+1} - \bar{\phi}^{n+1})$$

$$A(n) = \frac{1}{\sqrt{5}}(\phi^{n+1} - \bar{\phi}^{n+1} - 1) \in \Theta(\phi^{2^b}), \textit{ where } b = \lfloor \log_2 n \rfloor + 1$$