Algorithms and Lab (CSE130)

Complexity Functions

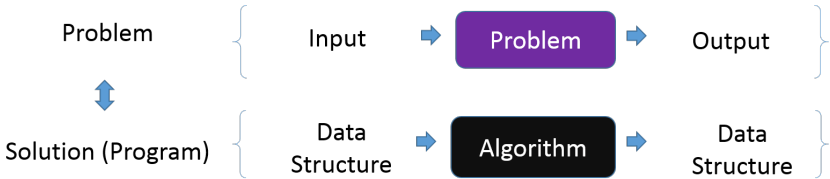
Muhammad Tariq Mahmood

tariq@koreatech.ac.kr
School of Computer Science and Engineering

Note: These notes are prepared from the following resources.

- (main text)Foundations of Algorithms, by Richard Neapolitan and Kumarss Naimipour
- Python Algorithm ( ) by Y.K. Choi (2021) (Korean)
- Introduction to the Design and Analysis of Algorithms by Anany Levitin
- Introduction to Algorithms, by By Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein
- https://www.geeksforgeeks.org

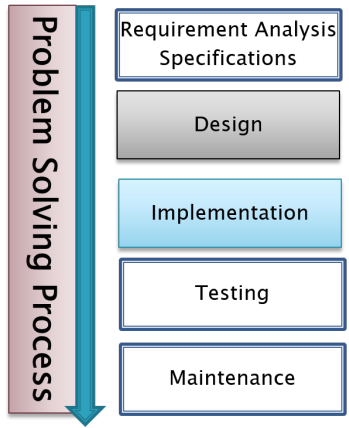
What is an Algorithm?



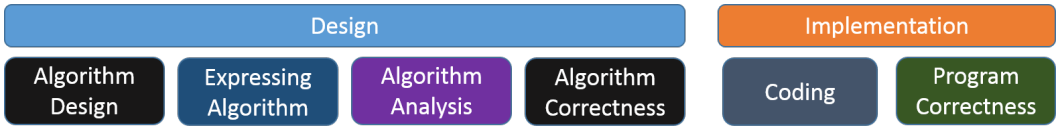
- A computer **algorithm** is a detailed **step-by-step method** for solving a **problem** using a computer.
- An algorithm is a **sequence of unambiguous instructions** for solving a **problem**, i.e., for obtaining a required output for any **legitimate input** in a finite amount of time.
- An algorithm is a **well-defined computational procedure** that takes some value or a set of values as **input** and produces some value or a set of values as **output**.
- Applying a technique to solve a **problem** results in a **step by step procedure**. This step by step procedure is called an algorithm for the problem.

Problem Solving Process

- Requirements analysis and specifications: It involves in determining 1) input and output required to solve the problem. 2) other information, such as software usage, feasibility of the solution
- Design: It involves in 1) selecting appropriate data structures to organize the input/output data and designing algorithms, needed to process the data efficiently
- Implementation: It involves in selecting appropriate programming language and translating algorithms into well-structured, well-documented, readable code
- Testing: It involves in executing the software and correcting errors and proving correctness of algorithms and modifying algorithms
- Maintenance: It involves in modifying software to improve performance and adding new features, etc



Design and Implementation Phases



Algorithm Analysis

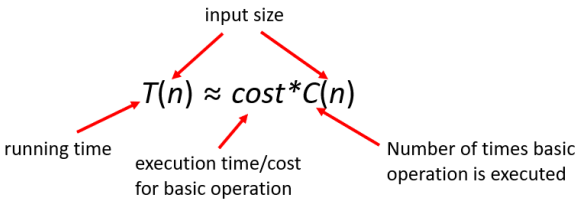
- The analysis of algorithms is to determine the computational complexity in terms of time and storage.
- Two approaches: Empirical/Experimental analysis and theoretical/formal analysis
- Theoretical/Formal analysis:
  - It involves determining a function (the time complexity function) that relates the algorithm's **input size** to the number of steps it takes or the number of storage locations it uses (in case of space complexity function).
  - It is common to estimate algorithm complexity in the asymptotic sense i.e., using order notations, Big-O notation, Big-omega(Ω) notation and Big-theta(Θ) notation
  - It dose not depend on
    - Actual number of CPU cycles (Computer)
    - Number of instructions
    - Programming language
    - Programmer

Contents

Complexity Functions

Complexity Function

- Time efficiency of an algorithm is measured by counting the number of times the algorithms basic operations are executed.
- Let *cost* be the execution time of a (basic operation), C(n) be the number of times this operation needs to be executed for this algorithm. Then we can estimate the running time $T(n)$ as



- $T(n)$ is the complexity function of n, where n is the input size
- For simplicity, it can be assumed that each operation takes 1 1 unit of time i.e. cost $= 1$ unit of time
- Basic operation: is the operation that contributes towards the running time of the algorithm

Complexity Functions (cont...)

- Basic operations in computing include read(input) , write(output), assignment (copy), arithmetic and relational operations
- The running time (complexity) of an algorithm increases with the input size
- All algorithms run longer on larger inputs. For example, it takes longer to sort larger arrays, multiply larger matrices, and so on. Therefore it seems logical to investigate an algorithm's efficiency in terms of its input size
- It is common to have algorithms that require more than one parameter (e.g. Graph algorithms need two parameters; No. of nodes and No. of edges)
- There may also be more than one choices for picking up one parameter as input size(e.g. In matrix multiplication; the dimensions of matrices, No of all elements)
- The choice of an appropriate size metric can be influenced by operations of the algorithm

Complexity Functions (cont...)

- Few examples for input size and basic operations

Problem	Input size measure	Basic operation
computing the sum of n numbers	n	addition operation
Searching for key in a list of items	Number of lists items	Key comparison
Multiplication of two matrices	Matrix dimensions	Multiplication of two numbers
n-Queens Problem	number of Queens/number of nodes	visiting a node
Graph problem	Numbers of vertices and/or edges	Visiting a vertex or traversing an edge

Complexity Functions (cont...)

Setting up a Complexity Function

- The total running time is roughly proportional to how many times some **basic operation(s)** is/are done
- Let us consider an algorithm to compute average of integers in an Array S and examine basic operation, input size and complexity function(s) from the given example.

```
1:  procedure AVERAGE(integer n, number S[])
2:      integer i, sum
3:      double ave
4:      sum = 0, ave = 0.0
5:      for (i = 1; i <= n; i++) do
6:          sum = sum + S[i]
7:      end for
8:      ave = sum/n
9:      return ave
10: end procedure
```

- No of operations
  - 0
  - 0
  - 2
  - 0
  - $\sum_{i=1}^{n} 1$
  - 0
  - 1
  - 0

- Complexity Function:

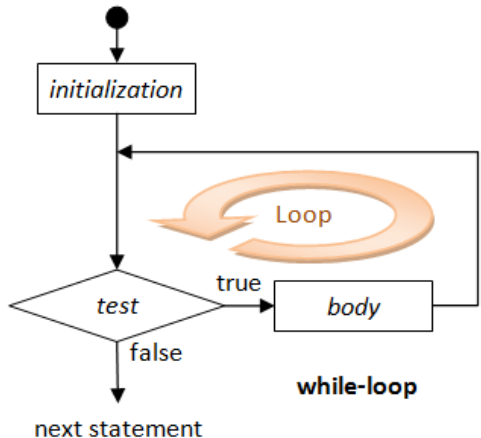
$$T(n) = 2 + \sum_{i=1}^{n} 1 + 1 = 2 + n + 1 = n + 3$$

Complexity Functions for Iterative Algorithms

Algorithms can be divided into two main categories: **Iterative Algorithms** and **Recursive Algorithms**

Iterative Algorithms

- Iteration is a process of executing the set of instructions repeatedly till the condition in iteration statement becomes false.
- The iteration statement includes the initialization, comparison, execution of the statements inside the iteration statement and finally the updating of the control variable.
- Infinite iteration due to mistake in iterator assignment or increment, or in the terminating condition, will lead to infinite loops



Complexity Functions for Iterative Algorithms (cont...)

- When an algorithm contains an iterative control construct such as a while or for loop, its running time can be expressed as the sum of the times spent on each execution of the body of the loop.
- By adding up the time spent on each iteration, we obtained the summation (or series)
- In order to simplify the summations (or series), few useful formulas are provided here

$$\sum_{j=1}^{n} 1 = \underbrace{1+1+1+\cdots+1}_{n \quad times} = n \;\Rightarrow [(n)-(1)+1]$$

$$\sum_{j=1}^{n-1} 1 = \underbrace{1+1+1+\cdots+1}_{n-1 \quad times} = n-1 \Rightarrow [(n-1)-(1)+1]$$

$$\sum_{j=5}^{n+10} 1 = \underbrace{1+1+1+\cdots+1}_{n+6 \quad times} = n+6 \Rightarrow [(n+10)-(5)+1]$$

$$\sum_{j=i+1}^{n-1} 1 = \underbrace{1+1+1+\cdots+1}_{n-i-1 \quad times} = n-i-1 \Rightarrow [(n-1)-(i+1)+1]$$

Complexity Functions for Iterative Algorithms (cont...)

- An arithmetic series :

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2} \tag{1}$$

$$\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6} \tag{2}$$

$$\sum_{i=1}^{n} i^3 = \left[\frac{n(n+1)}{2}\right]^2 \tag{3}$$

$$\sum_{k=0}^{n} x^k = \frac{x^{n+1}-1}{x-1}, \quad \text{(assuming } x \neq 1\text{)}. \tag{4}$$

- A geometric or exponential series : $\sum_{j=0}^{n} (r)^j = \frac{r^{n+1}-1}{r-1}$
- The above closed form solutions for a summations can be proved through the formal method: Mathematical Induction

Complexity Functions for Iterative Algorithms (cont...)

- Principle of Mathematical Induction (Mathematics)
  - Show true for n $= 1$
  - Assume true for n $=$ k
  - Show true for n $=$ k $+ 1$
  - Conclusion: Statement is true for all n $>= 1$

Complexity Functions for Iterative Algorithms (cont...)

Example-1: Add Array Members

- **Problem:** Add all the numbers in the array S of n numbers.
- **Inputs:** positive integer n, array of numbers S indexed from 1 to n.
- **Outputs:** sum, the sum of the numbers in S.

```
1: procedure sumArray(integer n, number S[])
2:     integer i
3:     number sum
4:     sum = 0
5:     for (i = 1; i <= n; i++) do
6:         sum = sum + S[i]
7:     end for
8:     return sum
9: end procedure
```

- Input size: n, number of elements in the array S.
- Basic operation(s): comparison $i <= n$ and \ or $sum = sum + S[i]$ inside loop
- Complexity Function

$$\begin{aligned} T(n) &= 1 + \sum_{i=1}^{n} +1 \\ &= 1 + n + 1 \\ &= n + 2 \end{aligned}$$

Example-2: Exchange Sort

- **Problem:** Sort n keys in nondecreasing order.
- **Inputs:** positive integer n, array of keys S indexed from 1 to n.
- **Outputs:** Sorted array S.

```
1:  procedure exchangeSort(number S[])
2:      integer i, j
3:      for (i = 1; i <= n − 1; i++) do
4:          for (j = i + 1; j <= n; j++) do
5:              if (S[j] < S[i]) then
6:                  exchange S[i] and S[j]
7:              end if
8:          end for
9:      end for
10: end procedure
```

- Input size: n, the size of the input array S.
- Basic operation(s): comparisons inside the inner most loop

- Complexity function and its simplification

$$
\begin{aligned}
T(n) &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} 1 \\
&= \sum_{i=1}^{n-1} (n - (i+1) + 1) \\
&= \sum_{i=1}^{n-1} n - \sum_{i=1}^{n-1} i \\
&= n(n-1) - \frac{(n-1)(n-1+1)}{2} \\
&= n^2 - n - \frac{n^2 - n}{2} \\
&= \frac{2n^2 - 2n - n^2 + n}{2} = \frac{n(n-1)}{2}
\end{aligned}
$$

Complexity Functions for Iterative Algorithms (cont...)

Matrix Multiplication :

```
1:  procedure MATRIXMULTUPLY(A[][],B[][])
2:      integer i, j, k
3:      number C[][]
4:      for (i = 1; i <= n; i++) do
5:          for (j = 1; j <= n; j++) do
6:              C[i][j] = 0
7:              for (k = 1; k <= n; k++) do
8:                  C[i][j] = C[i][j] + A[i][k] * B[k][j]
9:              end for
10:         end for
11:     end for
12:     return C
13: end procedure
```

Complexity Function:

$$
\begin{aligned}
T(n) &= \left(\sum_{i=1}^{n}\left(\sum_{j=1}^{n}\left(1+\sum_{k=1}^{n}1\right)\right)\right)+1 \\
&= \sum_{i=1}^{n}\left(\sum_{j=1}^{n}(1+n)\right)+1 \\
&= \sum_{i=1}^{n}\left(\sum_{j=1}^{n}1\right)+\sum_{i=1}^{n}\left(\sum_{j=1}^{n}n\right)+1 \\
&= \sum_{i=1}^{n}n+\sum_{i=1}^{n}n^2+1 \\
&= n^2+n^3+1 \\
&= n^3+n^2+1
\end{aligned}
$$

Complexity Functions for Iterative Algorithms (cont...)

Matrix Multiplication :

```
1:  procedure matrixMultuply(A[][],B[][])
2:      integer i, j, k
3:      number C[][]
4:      for (i = 1; i <= n; i++) do
5:          for (j = 1; j <= n; j++) do
6:              C[i][j] = 0
7:          end for
8:      end for
9:      for (i = 1; i <= n; i++) do
10:         for (j = 1; j <= n; j++) do
11:             for (k = 1; k <= n; k++) do
12:                 C[i][j] = C[i][j] + A[i][k] * B[k][j]
13:             end for
14:         end for
15:     end for
16:     return C
17: end procedure
```

Complexity Function:

$$
\begin{aligned}
T(n) &= \sum_{i=1}^{n}\left(\sum_{j=1}^{n}1\right)+\left(\sum_{i=1}^{n}\left(\sum_{j=1}^{n}\left(\sum_{k=1}^{n}1\right)\right)\right)+1\\
&= \sum_{i=1}^{n} n+\sum_{i=1}^{n}\left(\sum_{j=1}^{n} n\right)+1\\
&= n^2+\sum_{i=1}^{n}\left(\sum_{j=1}^{n} n\right)+1\\
&= n^2+\sum_{i=1}^{n} n^2+1\\
&= n^2+n^3+1\\
&= n^3+n^2+1
\end{aligned}
$$

Complexity Functions for Iterative Algorithms (cont...)

General cases

- Consider a general case for a single parameter for input size

```
1:  procedure EXAMPLE01(n)
2:      statement/operation
3:      statement/operation
4:      ........
5:      for (i = 1; i <= n; i++) do
6:          statement/operation
7:          statement/operation
8:          ........
9:      end for
10:     statement/operation
11:     statement/operation
12:     ........
13: end procedure
```

- The complexity function

$$\begin{aligned} T(n) &= c_1 + \sum_{i=1}^{n} c_2 + c_3 \\ &= c_1 + c_2 n + c_3 \\ &= c_2 n + c_4, \quad c_4 = c_1 + c_3 \end{aligned}$$

where c_1, c_2, c_3, c_4 are constant

- The complexity function can be simplified as

$$\begin{aligned} T(n) &= 1 + \sum_{i=1}^{n} 1 + 1 \\ &= 1 + n + 1 \\ &= n + 2 \end{aligned}$$

Complexity Functions for Iterative Algorithms (cont...)

- Consider a general case with if-else statement

```
1:  procedure EXAMPLE02(n)
2:      operations
3:      for (i = 1; i <= n; i++) do
4:          if a < b then
5:              operations
6:              ....
7:          else
8:              operations
9:              ....
10:         end if
11:     end for
12:     operations
13: end procedure
```

- The complexity function

$$
\begin{aligned}
T(n) &= 1 + \sum_{i=1}^{n} (1 + p + (1 - p)) + 1 \\
&= 2 + \sum_{i=1}^{n} 1 + 0.3 \sum_{i=1}^{n} 1 + 0.7 \sum_{i=1}^{n} 1 \\
&\because p = 0.3 \\
&= 2 + n + 0.3n + 0.7n \\
&= 2n + 2
\end{aligned}
$$

where p is the probability for the condition being true

Complexity Functions for Iterative Algorithms (cont...)

- Consider another general case for multiple parameters for input size

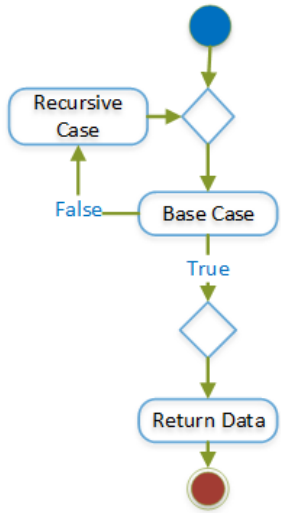
```
1:  procedure EXAMPLE03(n, m, p)
2:      statement/operation
3:      statement/operation
4:      av = 1
5:      for (i = 1; i <= n; i++) do
6:          for (j = 1; j <= m; i++) do
7:              for (k = 1; k <= p; i++) do
8:                  statement/operation
9:                  statement/operation
10:                 av = n × m × p
11:             end for
12:         end for
13:     end for
14:     statement/operation
15:     statement/operation
16:     .....
17: end procedure
```

- The complexity function

$$
\begin{aligned}
T(n,m,p) &= 1 + \sum_{i=1}^{n}\sum_{j=1}^{m}\sum_{k=1}^{p} 1 + 1 \\
&= \sum_{i=1}^{n}\sum_{j=1}^{m} p + 2 \\
&= \sum_{i=1}^{n} mp + 2 \\
&= nmp + 2
\end{aligned}
$$

Recursive Algorithms

- Recursion splits a problem into one or more simpler versions of itself
- Recursive definitions of problems naturally lead to recursive algorithms
- A recursive algorithm is an algorithm which calls itself with "smaller (or simpler)" input values
- A Recursive algorithm consists of two phases: base case and recursive case.
- base case is used to stop the recursion and the recursive call is made in recursive case..
- Without the base case, the recursive algorithm may run forever.



Example-1 A Recursive Algorithm for computing the n-th power of 2

- Problem: Computing the n-th power of 2.
- Inputs: n , a natural number .
- Outputs: n-th power of 2 .

```
procedure Power2(n)
    if (n == 0) then
        return 1;
    else
        return 2 * Power2(n − 1)
    end if
end procedure
```

- **Input size:** Input size is n
- **Basic Operation** No. of calls
- **complexity function**

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

Complexity Functions for Recursive Algorithms (cont...)

Example-2: Binary Search (Recursive)

- **Problem:** Determine whether x is in the sorted array S of size n.
- **Inputs:** positive integer n, sorted (nondecreasing order) array of keys S indexed from 1 to n, a key x.
- **Outputs:** location, the location of x in S (0 if x is not in S).
- **Input size:** Input size is n
- **Basic Operation** No. of calls
- **Complexity function**

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

n *is power of* 2

```
1:  procedure LOCATION(low,high)
2:      integer mid
3:      if (low > high) then
4:          return 0
5:      else
6:          mid = ⌊(low+high)/2⌋
7:          if (x == S[mid]) then
8:              return mid
9:          else
10:             if (x < S[mid]) then
11:                 return location(low, mid − 1)
12:             else
13:                 return location(mid + 1, high)
14:             end if
15:         end if
16:     end if
17: end procedure
```

General forms of recursive algorithms

- **Recursive Algorithm GRA1 (Decrease and conquer)**

Algorithm 1 A General Recursive Algorithm(Pseudo-code)

```
1: procedure GRA1(n)                                              ▷
2:     if (n == 0) then
3:         { Statement − 1,.....,Statement − c1                    ▷ base-case
4:     else
5:         { Statement − 1, ....,Statement − c2                    ▷ recursive-case
           { GRA1(n − 1)
6:     end if
7: end procedure
```

- Setup the Complexity function

$$\begin{cases} T(0) = c_1 \ , & n = 0 \\ T(n) = \ T(n-1) + c_2, & n > 0 \end{cases}$$

- **Recursive Algorithm GRA2 (Divide and conquer-1)**

Algorithm 2 A General Recursive Algorithm(Pseudo-code)

```
1: procedure GRA2(n)                                              ▷
2:     if (n == 0) then
3:         { Statement − 1,.....,Statement − c1                    ▷ base-case
4:     else
5:         { Statement − 1, ....,Statement − c2                    ▷ recursive-case
           { GRA2(n/2)
6:     end if
7: end procedure
```

- **Setup the Complexity function**

$$\begin{cases} T(0) = c_1 \ , & n = 0 \\ T(n) = \ T(n/2) + c_2, & n > 0 \end{cases}$$

- **Recursive Algorithm GRA3(Divide and conquer-2)**

Algorithm 3 A General Recursive Algorithm(Pseudo-code)

```
1: procedure GRA3(n)                                              ▷
2:     if (n == 0) then
3:         { Statement − 1,.....,Statement − c1                    ▷ base-case
4:     else
           { Statement − 1, ....,Statement − c2
5:         { GRA3(n/3)                                            ▷ recursive-case
           { GRA3(n/3)
6:     end if
7: end procedure
```

- **Setup the Complexity function**

$$\begin{cases} T(0) = c_1 \ , & n = 0 \\ T(n) = \ 2T(n/3) + c_2, & n > 0 \end{cases}$$

Solving Recurrence Equations

Recurrence relation

- A recurrence relation is an equation that recursively defines a sequence where the next term is a function of the previous terms.

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

- A recurrence by itself does not represent a unique function. We must also have a starting point, which is called an initial condition. In the above example , the initial condition is $T(0) = 1$

- An explicit expression that enables us to compute $T(n)$, for an arbitrary n without starting at 0 is called a solution to the recurrence equation.

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

- The analysis of recursive algorithms involves recurrence relations. Therefore,the recurrence equations must be solved to determine the time complexity of such algorithms.

- Appendix B of the Book explains different types of recurrence relations and methods for solving them

Solving Recurrence Equations (cont...)

Types of Recurrence Relations: Recurrence relations can be classified with respect to Homogeneous or non-homogeneous, Linear or non-linear, Order of the Recurrence Relation, Constant versus non-constant coefficients

- First order Recurrence relation : A recurrence relation of the form

$$T(n) = \begin{cases} c_1 & n = 1 \\ c_2 T(n-1) + f(n) & n > 1 \end{cases}$$

where c_1 and c_2 are constants and $f(n)$ is a known function is called linear recurrence relation of first order with constant coefficient. If $f(n) = 0$, the relation is homogeneous otherwise non-homogeneous.

- Second order linear Recurrence relation : A recurrence relation of the form

$$T(n) = \begin{cases} c_1 & n = 0 \\ c_2 & n = 1 \\ c_3 T(n-1) + c_4 T(n-2) + f(n) & n > 1 \end{cases}$$

where c_1 , c_2, c_3 , c_4 are constants and $f(n)$ is a known function is called linear recurrence relation of second order with constant coefficient. If $f(n) = 0$, the relation is homogeneous otherwise non-homogeneous.

Solving Recurrence Equations (cont...)

Solving Recurrence relations: There are many approaches to solving recurrence relations, few of them are listed here

1. Guess and Confirm (Inductive method)
2. Backward Substitution method
3. Forward Substitution method
4. Change of variables (Domain Transformation) method
5. Recursion tree method
6. Master Theorem

Solving Recurrence Equations (cont...)

- Few examples: recurrence relations and their closed form solutions.

Recurrence Relation	(closed form)Solution
$\begin{cases} T(1) = 1 \\ T(n) = T\left(\frac{n}{2}\right) + 1, \ n > 1, \ n \text{ is power of } 2 \end{cases}$	$T(n) = \log n + 1$
$\begin{cases} T(1) = 1 \\ T(n) = 7T\left(\frac{n}{2}\right), , n > 1, \ n \text{ is power of } 2 \end{cases}$	$T(n) = n^{\log 7} \approx n^{2.81}$
$\begin{cases} T(0) = 0 \\ T(1) = 1 \\ T(n) = T(n-2) + T(n-1), \quad n > 1 \end{cases}$	$T(n) = \frac{\left[\frac{\left(1+\sqrt{5}\right)}{2}\right]^n - \left[\frac{\left(1-\sqrt{5}\right)}{2}\right]^n}{\sqrt{5}}$
$\begin{cases} T(0) = 0 \\ T(n) = T(n-1) + n - 1, \quad n > 0 \end{cases}$	$T(n) = \frac{n(n-1)}{2}$
$\begin{cases} T(1) = 0 \\ T(n) = 7T\left(\frac{n}{2}\right) + 18\left(\frac{n}{2}\right)^2, \quad n > 1, \ n \text{ is power of } 2 \end{cases}$	$T(n) = 6n^{\log 7} - 6n^2$

Solving Recurrence Equations (cont...)

- Example-1: Recurrence relation

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

- Simplification of recurrence relation by using backward substitutions method

$$\begin{aligned} T(n) &= T(n-1) + 1 \\ &= T(n-2) + 2 \\ &= T(n-3) + 3 \\ &\vdots \\ &= T(n-k) + k \\ &\vdots \\ &= T(n-n) + n \\ T(n) &= 1 + n \end{aligned}$$

- Example-2: Recurrence relation

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

n is power of 2

- Simplification of recurrence relation by using forward substitutions method

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

$$T(4) = T(\frac{4}{2} + 1 = T(2) + 1 = 2 + 1 = 3$$

$$T(6) = T(\frac{6}{2} + 1 = T(3) + 1 = 3 + 1 = 4$$

$$T(8) = T(\frac{8}{2} + 1 = T(4) + 1 = 4 + 1 = 5$$

It appears : $T(n) = log_2(n) + 1$

Solving Recurrence Equations (cont...)

- Example-3 The Fibonacci sequence is a series of numbers where a number is the addition of the last two numbers, starting with 0, and 1, i.e. $0, 1, 1, 2, 3, 5, 8, 13, 21, ...$
- The recurrence relation for Fibonacci sequence is given as

$$T(n) = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ T(n-1) + T(n-2) & n > 1 \end{cases}$$

- Obtain the characteristic equation:

$$T(n) - T(n-1) - T(n-2) = 0$$
$$r^2 - r - 1 = 0$$

- Solve the characteristic equation: From the formula for the solution to a quadratic equation, the roots of this characteristic equation are

$$r = \frac{1+\sqrt{5}}{2}, r = \frac{1-\sqrt{5}}{2}$$

Solving Recurrence Equations (cont...)

- Get the general solution to the recurrence:

$$T(n) = c_1 \left(\frac{1+\sqrt{5}}{2}\right)^n + c_2 \left(\frac{1-\sqrt{5}}{2}\right)^n$$

- Determine the values of the constants by applying the general solution to the initial conditions:

$$T(0) = c_1 \left(\frac{1+\sqrt{5}}{2}\right)^0 + c_2 \left(\frac{1-\sqrt{5}}{2}\right)^0 = 0$$
$$T(1) = c_1 \left(\frac{1+\sqrt{5}}{2}\right)^1 + c_2 \left(\frac{1-\sqrt{5}}{2}\right)^1 = 1$$

- Solving the above system yields: $c_1 = \frac{1}{\sqrt{5}}, c_2 = -\frac{1}{\sqrt{5}}$ and then final solution:

$$\begin{aligned} T(n) &= \frac{1}{\sqrt{5}} \left(\frac{1+\sqrt{5}}{2}\right)^n - \frac{1}{\sqrt{5}} \left(\frac{1-\sqrt{5}}{2}\right)^n \\ &= \frac{\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n}{\sqrt{5}} \end{aligned}$$

Solving Recurrence Equations (cont...)

- Example04 : Consider the following non-homogeneous recurrence relation

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

- The characteristic equation with root $r = 1$

$$(r-1)(r-1)^2 = 0$$
$$(r-1)^3 = 0$$

- The general solution to the recurrence:

$$T(n) = c_1(1^n) + c_2 n(1^n) + c_3 n^2(1^n)$$
$$= c_1 + c_2 n + c_3 n^2$$

- We need three initial conditions to get three values

$$T(0) = 0$$
$$T(1) = T(0) + 1 - 1 = 0 + 0 = 0$$
$$T(2) = T(1) + 2 - 1 = 0 + 1 = 1$$

- Set of questions after applying initial conditions

$$c_1 = 0$$
$$c_1 + c_2 + c_3 = 0$$
$$c_1 + 2c_2 + 4c_3 = 1$$

- Getting values for constants
  $c_1 = 0, c_2 = -1/2, c_3 = 1/2$, the final solution

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