Continuation of unit 1 Recurrence relation for factorial

Algorithm F(n)

```
F(n)
{
  if n == 0
      return 1
  else
      return n * F(n-1)
}
```

* Recurrence Relation for above algorithm is

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

$T(n) = n * T(n-1)$ — ①

$T(n-1) = (n-1) * T(n-2)$ — ②

$T(n-2) = (n-2) * (n-1) * T(n-3)$ — ③

$T(n) = n*$

* Process continues until k^{th} step

$T(n) = n*(n-k)$

$T(n) = n * T(n-1)$ ~ (1)

$T(n-1) = (n-1) * T(n-2)$ ~ (2)

$T(n-2) = (n-2)*(n-1) * T(n-3)$ ~ (3)

* substitue eq (2) in eq (1)

$T(n) = n * (n-1) * T(n-2)$ ~ (4)

* Substitute ~~eq (3) in eq~~ equation (3) in equation (4)

$T(n) = n * (n-1) * (n-2) * (n-1) * T(n-3)$

⋮

k^{th} step [continue till k^{th} step]

$T(n) = K * (n-K) * (n-K) * (n-K) * T(n-K)$ ~ (5)

∴ whenever n==1 || n==0 programs terminate

$\therefore n-K=1$

$\boxed{K = n-1}$

* Substitute K in equation (5)

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

$= n-1 * \cancel{n}-\cancel{n}+1 * \cancel{n}-\cancel{n}+1 * \cancel{n}-\cancel{n}+1 * T(1)$

$= n-1 * T(1)$

$= (n-1) * 1$

$= n-1$

$T(n) = n \quad \therefore O(n)$

Master theorem

$$T(n) = aT(n-b) + O(n^k)$$

$a \geq 1, \ b \geq 1, \ k \geq 0$

1) if $a == 1$ $\quad T(n) = O(n^{k+1})$

2) if $a > 1$ $\quad T(n) = O(n^k \cdot a^{n/b})$

3) if $a < 1$ $\quad T(n) = O(n^k)$

Problems

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

$T(n) = 1T(n-1) + 1 \cdot n^0$

$a = 1 \quad b = 1, \quad k = 0$

case 1:

$T(n) = O(n^{0+1})$

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

2) Algorithm Test(n)

```
{
    if (n>0)
    {
        printf("%d ", n);
        Test(n-1);
        Test(n-2);
    }
}
```

Recurrance relation for is

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

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

$\therefore T(n) = 2T(n-1) + 1 \cdot n^0$

$a = 2 \quad b = 1 \quad k = 0$

case 2:

$T(n) = O(n^0 \cdot 2^{n/1}) = \underline{O(2^n)}$

Problem 3

```
void Test(n)
{
   if(n==0)          — 1
   {
      for(i=0; i<n; i++)      -> n+1
      {
         Statement;      — n
      }
      Test(n-1)      — (n-1)
   }
}
```

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

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

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

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

$a=1 \quad b=1 \quad K=1$

case 1:

$T(n) = O(n^{1+1})$

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

Scanned by CamScanner

Problem 4: ~~Do~~ Dividing Function

~~Recursive Tested~~

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(n/2) + 1

* Recurrence Relation :-

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

$$T(n) = T(n/2)+1 \quad \sim (1)$$

$$T(n/2) = T(n/2^2)+1 \quad \sim (2)$$

$$T(n/2^2) = T(n/2^3)+1 \quad \sim (3)$$

* Substitute eq (2) in eq (1)

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

$$= T(n/2^2)+1+1$$

$$= T(n/2^3)+1++1$$

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

understanding the problem

* First thing you need to do before designing an algorithm is to understand completely the problem given

* Read the problem's description carefully and ask question if you have any doubts about the problem.

* An input to an algorithm specifies an instance of the problem the algorithm solves.

* verify what should be the input and what should be output that your expecting to be

* correct algorithm is not one that works most of the time, but one that works correctly for all legitimate inputs.

Ascertaining the capabilities of a computational Design

* Decide the capabilities of the computational devices the algorithm is intended for.

1. Sequential algorithm
2. parallel algorithm

* Instructions are executed one after the another, one operation at a time, sequential algorithm.

* Instructions are executed concurrently or parallelly, called as parallel algorithm.

Scanned by CamScanner

Choosing b/w exact and approximate problem solving

* Next principal decision is to choose b/w solving the problem
    - exactly or
    - solving it approximately.

* We have two kinds of algorithm.
    1. exact algorithm
    2. approximation algorithm.

* Choose the appropriate Data Structure for a given algorithm.

Algorithm Design Technique

* An algorithm design technique is a general approach to solving problems algorithmically that is applicable to a variety of problems from different areas of computing.

* There are various Algorithm Design Techniques are available such as
    1. Brute force
    2. Divide and conquere
    3. Decrease and conquere
    4. Dynamic programming
    5. Greedy Method etc.

* Choose the appropriate Algorithm Design Technique for the given problem.

Scanned by CamScanner

Designing an algorithm and data structure ~~and data and dat~~

* Some design technique can be ~~used~~ simply ~~inap~~ inapplicable to the problem in question

* Several techniques need to be combined and there are as applications algorithm that are hard to pinpoint of the ~~des~~ known design technique.

* Pay close attention to chooing data structures appropriate for the operations performed by the algorithm.

* There are 3 ways to design an algorithm
   1. writing natural language
   2. writing Flow chart
   3. writing pseudocode

* ~~Pseudocode~~ pseudocode is usually more precise than natural language.

Proving an Algorithm's correctness

* once an algorithm has been specified, you have to prove its correctness.

* you have to prove that the algorithm produces a required results for every legitimate input in a finite amount of time.

* A common technique for proving correctness is to use mathematical induction because an algorithm's iterations provide a natural sequence of steps needed for such progs'

Analysing algorithm

* we usually want our algorithms to possess several qualities.
* After correctness, by far the most important is efficiency'
* There are two kinds of algorithm efficiency
  1. Time efficiency → indicating how fast the algorithm runs
  2. Space efficiency → indicating how much extra memory it uses.

Coding an algorithm

* Most algorithms are destined to be ultimately implemented as computer.
* programming an algorithm presents both a peril and an opportunity.
* Validate program
* Test the computer programs.

Tower of Hanoi

```
void TOH{ int n, int A, int B, int C}
{
    ~~if(n>0)~~
    {
        if (n==1)
        {
            ~~...~~
            printf("move DISK from %d to %d", A, C);
        }
        else
        {
            TOH(n-1, A, C, B);
            printf("move DISK from %d to %d", A, C)
            TOH(n-1, B, A, C);
        }
    }
```

Recursive Equation

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

* Tower of Hanoi is one of the mathematical game where it has 3 rods and n disk
* The objective of the game is to move the entire disk to another rod with following rules:

1. Only one disk can be move at a time.
2. DISK can be moved if it is in the uppermost disk on a stack
3. NO DISK may be placed on top of a smaller disk.

$T(n) = 2T(n-1) + 1 \quad \sim (1)$

$2T(n-1) = 4T(n-2) + 1 + 1 \quad - (2)$

$4T(n-2) = 8T(n-2) + 1 + 2 \quad \sim (3)$

* Substitue (2) in equation (1)

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

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

$= 8T(n-3) + 1 + 3$

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

⋮

continue till k^th step

$= 2^K (KT(n-K) + K \} + K \quad \sim (4)$

$\therefore$ program terminates when $n - K = 1$

$\therefore n - K = 1$

$\boxed{K = n-1}$

* Substitue K in equation (4)

$= \cancel{\ldots} (\cancel{n-1-n+1}) + n-1 \} + n-1$

$= 2^{n-1} \{ n-1 * T(n-(n-1)) + (n-1) \} + n-1$

$= 2^{n-1} \{ n-1 * T(1) + n-1 \} + n-1$

$= 2^{n-1} \{ n-1 * 1 + n-1 \} + n-1$

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

$$T(n) = 2^{n-1}\{3n-3\}$$

$$T(n) = 2^{n-1} * \{3n-3\}$$

$$T(n) = 2^{n} * \frac{1}{2} * \{3n-3\}$$

$$T(n) = 2^{n-1}$$

$$O(2^{n})$$

By Master Theorem

order of growth

* order of growth provides th behavior of an algorithm
* order of growth allows us to compare th relative performance of alternate algorithm.
* order of growth means, when input size of a given algorithm increases, th computational time also increases.
* ∴ computational time is directly proportional to input size
* order of growth

$2^{2^n} \geq n! \geq 4^n \geq 2^n \geq n^4 \geq n^3 \geq n^2 \geq n\log n \geq n \geq \log n \geq \log\log n \geq \frac{\log n}{\log\log n} \geq 1$