1. FUNDAMENTALS OF ALGORITHMS AND PROBLEM SOLVING

Data Structures and Algorithms

19ECSC201

SOCSE, KLE TU, HUBBALLI-31

Prakash Hegade

Syllabus:
Space and Time Complexities, Order of an Algorithm, Efficiency Analysis and Stacks and Queues, Recursive Definitions and Functions, Towers of Hanoi, Backtracking, Recursion Vs. Iteration.

Algorithms: Introduction

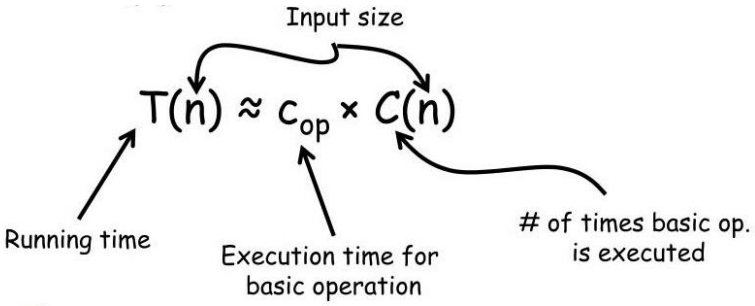
"Algorithm is a sequence of unambiguous instructions for solving a problem, i.e., for obtaining a required output for any legitimate input in a finite amount of time."

Refer to the definition of Finite Automaton. Can you map the tuples to the above definition? An algorithm is associated with two types of efficiencies: time and space.

Time Efficiency: How fast an algorithm runs
Space efficiency: Extra space the algorithm requires

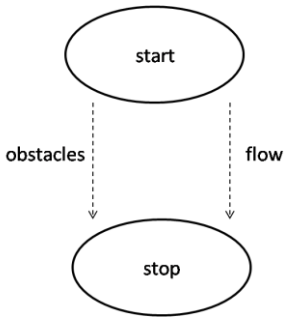
Analysis Framework

1. Time efficiency is measured by counting the number of times algorithms Basic Operation is executed. We use Basic Count as against Step Count - the total number of instructions executed. Basic operation is the critical operation in the algorithm (most executed, one that takes longer time)



2. Space efficiency is the extra memory units consumed by algorithm
3. Space and time are measured as functions of algorithms input size
4. Efficiencies of some algorithms may differ significantly for inputs of the same size. In such cases we go for best case, average case and worst case efficiencies
5. Every algorithm has associated order of growth

Design of New Algorithms



A new algorithm design can be tackled in four ways:
- Strength of start
- Strength of stop
- Strength in flow
- Winning over obstacles

One with a good start:
- Security algorithms – need to have strong assumptions
- "Integer factorization is hard" was the assumption made in the discovery of ssh protocol

One with obstacles:
- Can we have a pool of algorithm and pick one at random?
- When obstacles are too difficult to beat, we go proactive – take a small password as safe password is of infinite length and change it frequently
- Solving problems by introducing obstacles to obstacles. Example: rice price goes up → hold them in warehouses. Then to get the stock out, release the rats.

One with stop:
- We have a proper stop specification
- There are also problems with pseudo-randomness
- Approximate algorithms – change the stopping condition

One with the flow:
- Algorithm design techniques, which we shall study in detail through the course syllabus

On the whole:
- **S**tart well
- **O**bstacles guaranteed
- **F**low skillfully
- **T**erminate efficiently

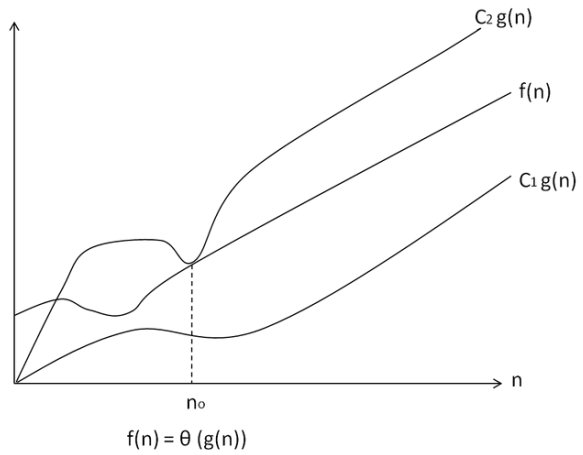
And this is possibly the SOFT part of the software.

Asymptotic Notations

Let's start with the fundamental question, 'What are Asymptotic Notations?'
Asymptotic notations are mathematical notations that describe the limiting behavior of a function when the argument tends towards a particular value (or maybe infinity). The phrase 'limiting behavior' here is important. The notations describe the restriction and boundaries towards a function.

Let's stay for a while in the math world and understand the three members that belong to this family.

Member 01: θ Notation
A function f(n) belongs to the set of θ(g(n)) if there exist positive constants C1 and C2 such that it can be packed between two functions **C1g(n)** and **C2g(n)**, this being true from the value n_0. The notation is graphically presented in the figure given below.

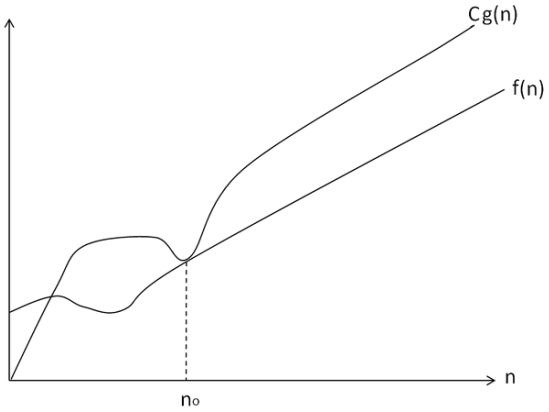


For all n >= n_0, the value of f(n) lies at or above C1g(n) and at or below C2g(n). i.e.,
C1 g(n) <= f(n) <= C2 g(n) for all n >= n_0

Example:
As an example let us express the function f(n) = 10n + 10 using θ notation.
10n <= 10n+10 <= 15n
C1 = 10
C2 = 15
g(n) = n
n_0 = 1

Member 02: O Notation

O notation is used to give an upper bound on a function, to within a constant factor. For all the values of n at and to the right of n_0, the value of the function f(n) is on or below Cg(n). The notation is graphically presented in the figure given below.



f(n) = O (g(n))

For all n >= n_0, the value of f(n) lies at or below Cg(n) i.e,
f(n) <= C g(n) for all n >= n_0

Example:

As an example let us express the function f(n) = 10n + 10 using O notation

10n+10 <= 15n

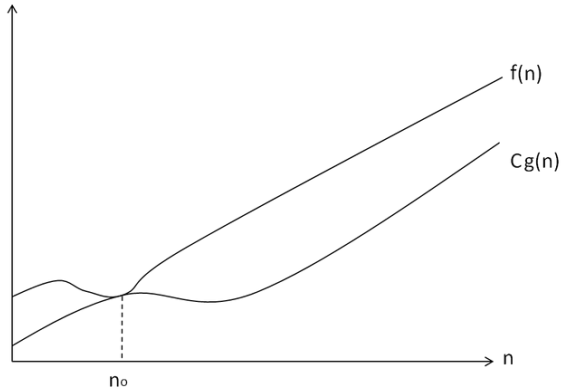
C = 15

g(n) = n

n_0 = 1

Member 03: Ω Notation

Ω notation is used to give a lower bound on a function, to within a constant factor. For all the values of n at and to the right of n_0, the value of the function f(n) is on or above Cg(n). The notation is graphically presented in the figure given below.



f(n) = Ω (g(n))

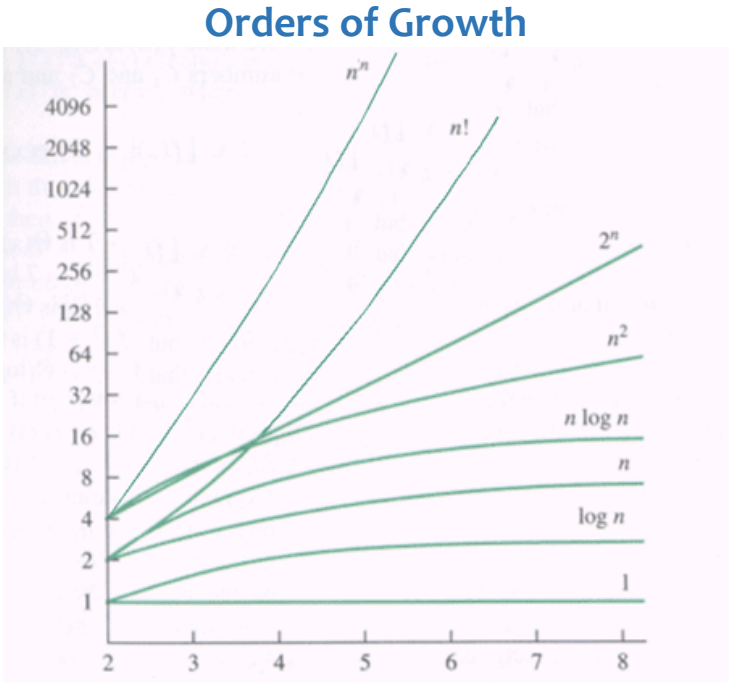
For all n >= n_0, the value of f(n) lies at or above Cg(n). i.e,
f(n) >= C g(n) for all n >= n_0

Example:
As an example let us express the function f(n) = 10n + 10 using Ω notation.
10n+10 >= 10n
C = 10
g(n) = n
n_0 = 1

These notations that we understood from the math world can be used to characterize aspects or features of any domains applicable and relevant.

Asymptotic Notions for Algorithms: In algorithms, we use asymptotic notations to characterize the running times of the algorithms. However, that said, they can be used to characterize some other or any other feature like space as well.

When we say running time, we need to be little more specific about the nature of the input – Is it the worst case, best case, or an average case. Considering the cases and the notations, we can easily map the three notations to the three cases. Hence we have the notations as:
Θ – Average Case, O – Worst Case and **Ω – Best Case**



Also refer to graph drawn in class notes. The above image is referenced from cs.odu.edu. The behavior of these classes can be seen in the table below:

	constant	*logarithmic*	*linear*	*N-log-N*	*quadratic*	*cubic*	*exponential*
n	$O(1)$	$O(\log n)$	$O(n)$	$O(n \log n)$	$O(n^2)$	$O(n^3)$	$O(2^n)$
1	1	1	1	1	1	1	2
2	1	1	2	2	4	8	4
4	1	2	4	8	16	64	16
8	1	3	8	24	64	512	256
16	1	4	16	64	256	4,096	65536
32	1	5	32	160	1,024	32,768	4,294,967,296
64	1	6	64	384	4,069	262,144	1.84×10^{19}

Can you write a short note on orders of growth?

Algorithmic Conventions

We are going to adapt to the following conventions while writing an algorithm.

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

Analysis:

1. Find out the **Basic Operation**. There is no hard and fast rule or any definition to determine the BO.
   But it is the one which consumes maximum number of clock cycles (or say takes the longest time or runs many times while code executes)
   In above case, it is the comparison operation**:** ***if A[i] > maxval***

2. Secondly, we need to check do we have separate best, worst, and average-case analysis?
   The algorithm is not going to consume different time at each run for the same input size 'n.' Hence we have only one analysis to be made.

3. Let C(n) be the function for the number of times the comparison is made. Setting up the equation:
   The BO is going to be executed once in every iteration of for() which runs for the condition variable 'i' ranging from 1 to n-1. Hence we can set up a summation formula.
   $C(n) = \sum_{i=1}^{n-1} 1$
   $= (n - 1 - 1 + 1)$ *[Upper Bound – Lower Bound + constant]*
   $= n - 1$

4. The order of growth is linear. As the input size increases, the time taken also increases in a linear fashion.

Iteration, Recursion, and Backtracking: as seen in Wikipedia

Iteration
Means the act of repeating a process with the aim of approaching a desired goal, target or result. Each repetition of the process is also called an "iteration," and the results of one iteration are used as the starting point for the next iteration. Iteration in computing is the repetition of a block of statements within a computer program.

Recursion
A class of objects or methods exhibit recursive behavior when they can be defined by two properties:

1. A simple base case (or cases)
2. A set of rules that reduce all other cases toward the base case

Backtracking
is a general algorithm for finding all (or some) solutions to some computational problem, that incrementally builds candidates to the solutions, and abandons each partial candidate *c* ("backtracks") as soon as it determines that *c* cannot possibly be completed to a valid solution.

Recursion: Introduction

Consider the **Factorial Function:** Product of all the integers between 1 to n.
n! = 1 if n= 0
n! = n* (n-1) * (n-2) * . . . * 1 if n > 0

To avoid the shorthand definition for n! We have to list a formula for n! for each value of n separately.
0! = 1
1! = 1
2! = 2 * 1
3! = 3* 2* 1
….

To avoid the shorthand and to avoid the infinite set of definitions, yet to define function precisely, we can write an algorithm that accepts an integer n and returns the value of n!

```
prod = 1;
   for(x = n; x > 0; x--)
      prod * = x;
return (prod);
```

This process is **Iterative.**
Iterative: explicit repetition of some process until a certain condition is met.

Let us take a closer look:
0! = 1
1! = 1 * 0!
2! = 2 * 1!
3! = 3 * 2!
4! = 4 * 3!
…
And so on…

Using mathematical notation, we can write it as:
n! = 1 if n =0
n! = n * (n-1)! If n > 0

This defines factorial in terms of itself.

Looks like a circular definition. A definition which defines an object in terms of a simpler case of itself is called a **recursive definition.**

Usage:

1 5! = 5 *4!
2 4! = 4 * 3!
3 3! = 3 * 2!
4 2! = 2 * 1!
5 1! = 1 * 0!
6 0! = 1

Now start back tracking

6' 0! = 1
5' 1! = 1 * 1
4' 2! = 2 * 1
3' 3! = 3 * 2
2' 4! = 4 * 6
1' 5! = 5 * 24

Which finally results in 120

Algorithm:

```
if(n == 0)
   fact =1;
else {
    x = n – 1
   find the value of x! call it y;          re-executing the algorithm with value x
   fact = n * y;
}
```

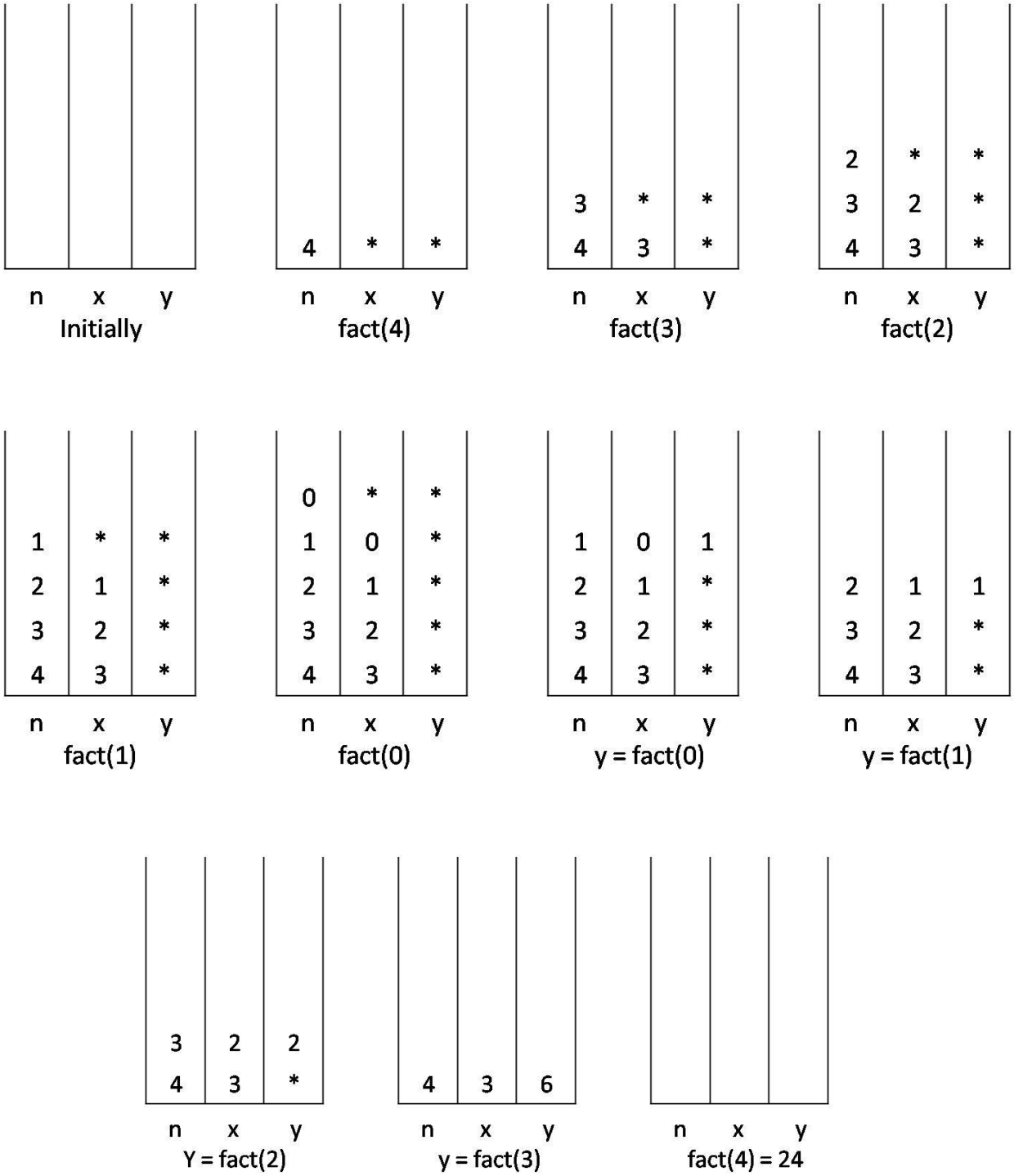
This example is to introduce recursion, not as a more effective method of solving a particular problem.

Implementing in C language:

```
int fact (int n)
{
   int x, y;
   if(n ==0)
      return 1;
   x = n – 1;
   y = fact(x);
   return (n * y);
}
```

Recursive calls maintain stack to keep the local copies of variable of each call which is invisible to the user. On each call a new allocation of variable is pushed onto the stack. When the function returns, the stack is popped.

Below diagram summarizes the process for the call: fact(4)



Recursive Binary Search Example:

```
int binsrch(int a[], int x , int low, int high)
{
    int mid;
    if(low > high)
        return -1;
    mid = (low + high )/ 2;
    return ( x == a[mid] ? mid : x < a[mid] ? binsrch(a, x, low, mid -1) :
                                              binsrch(a, x, mid+ 1, high));
}
```

Recursive Version of Pingala Series Numbers:

```
int pingala(int n) {
    int x, y;
    if( n <= 1)
        return n;
    x = pingala (n-1);
    y = pingala (n-2);
    return (x + y);
}
```

Writing recursive programs:

- Find the trivial case
- Find a method of solving a complex case in terms of a simpler case
- The transformation of complex to simpler case should eventually result in trivial case

However, it is not easy to write recursive codes. It comes with lot of practice and patience. One should also be able to analyze the difference between Iterative and recursive method.

Recursive Example and Analysis:

```
ALGORITHM Binary(n)
// Counts the number of digits in binary representation of a given
// positive decimal integer
// Input:  A positive decimal integer
// Output: Number of digits in a binary representation of a given positive
//         decimal integer

if n=1
    return 1
else
   return Binary (n/2) + 1
```

```
[ Note:
The iterative version of the above algorithm would be written as:
count ← 1
while n > 1
    count ← count + 1
    n ← n / 2
return count ]
```

Analysis:
Here Addition operation is the Basic operation. Setting up the recurrence relation:

$$B(n) = \begin{cases} 0 & \text{if } n = 1 \\ B(n/2) + 1 & \text{otherwise} \end{cases}$$

(How did we set up the equation?
when n = 0, there are no additions hence it is 0.
In all other cases, there is one addition and recursive function call by halving the input size.)

Solving,
$B(n) = B(n/2) + 1$
Assume $n = 2^k$ (smoothness rule - refer class notes)
$B(2^k) = B(2^{k-1}) + 1$

On simplifying further, we end up with the order of growth of **$\log_2 n$.**

The Towers of Hanoi Problem

Task:
There are 3 pegs and N disks. The larger disk is always at the bottom. We need to move the N disks from A to C using B as auxiliary with the constraint that the smaller disk is always at the top.

Solution:
1. If n == 1, move the single disk from A to C and stop
2. Move the top n-1 disks from A to B using C as an auxiliary
3. Move the remaining disk from A to C
4. Move the n-1 disks from B to C using A as auxiliary

(If you are not able to understand the above procedure, apply the solution on three disks problem. The procedure will follow.)

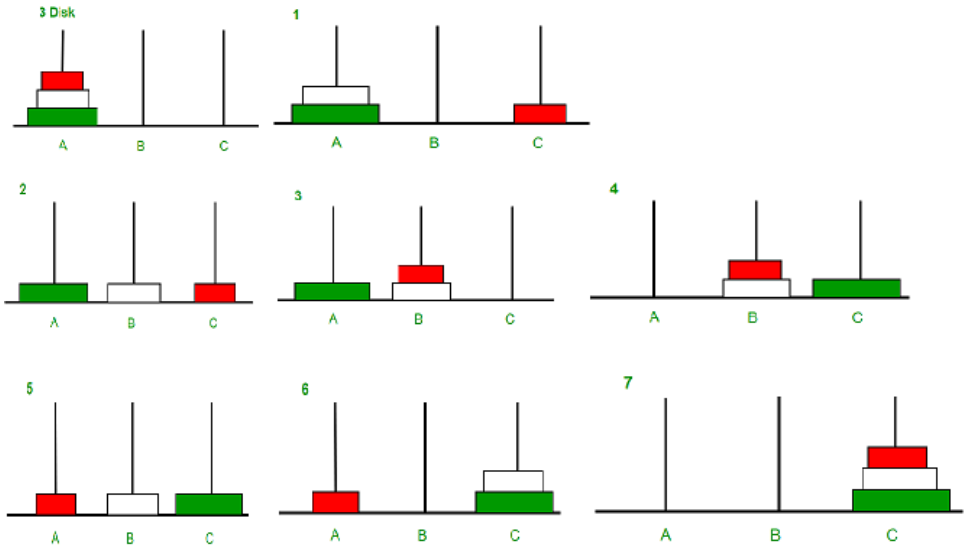


Image referenced from geeksforgeeks

Program:

```
#include <stdio.h>
#include <stdlib.h>
void towers(int, char, char, char);

int main()
{
    int n;
    printf("Enter the number of Disks to be moved\n");
    scanf("%d", &n);
    towers(n, 'A', 'C', 'B');
    return 0;
}

void towers(int n, char from, char to, char aux)
{
    if( n == 1) {
        printf("Move disk 1 from %c to %c\n", from, to);
        return;
    }
    // Move top n-1 disks from A to B using C as auxiliary
    towers(n-1, from, aux, to);

    // Move remaining disk from A to C
    printf("Move disk %d from %c to %c\n", n, from, to);

    // Move n-1 disks from B to C using A as auxiliary
    towers(n-1, aux, to, from);
}
```

Efficiency Analysis:

Here the basic operation is the moment of the disk – the printf() as referred in code. Setting up the recurrence relation:

```
T(n)        = 1                         if n = 1
            = T(n-1) + 1 + T(n-1)       otherwise
```

$T(n) = 2\,T(n-1) + 1$ → (1)

Substitute n = n-1 in (1), we have

$T(n-1) = 2T(n-2) + 1$ → (2)

Substitute (2) in (1) we have,

$T(n) = 2[\,2\,T(n-2) + 2] + 1$

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

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

$= 2^4\,T(n-4) + 2^3 + 2^2 + 2 + 1$

…

…

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

$= 2^n\,T(0) + 2^{n-1} + 2^{n-2} + \ldots + 2^3 + 2^2 + 2 + 1$

$= (2^n * 0) + 2^{n-1} + 2^{n-2} + \ldots + 2^3 + 2^2 + 2 + 1$

$= 2^{n-1} + 2^{n-2} + \ldots + 2^3 + 2^2 + 2 + 1$

This is a GP series.

$S = a\,(r^n - 1) / r - 1$

Where a = 1, r = 2 and n = number of terms.

Solving:

$S = 1\,(2^n - 1) / 2\text{-}1$

$= 2^n - 1$

The Algorithm has the order of growth which is **exponential.**

Efficiency of Recursion

- The non-recursive version of a program will execute more efficiently in terms of time and space
- The stacking activity hinders the performance of recursive procedures
- Some of the variables can be identified which do not have to be moved into a stack, and recursion can be simulated by pushing and popping only the relevant variables
- Sometimes recursion is the most natural and logical way of solving the problem – Towers of Hanoi

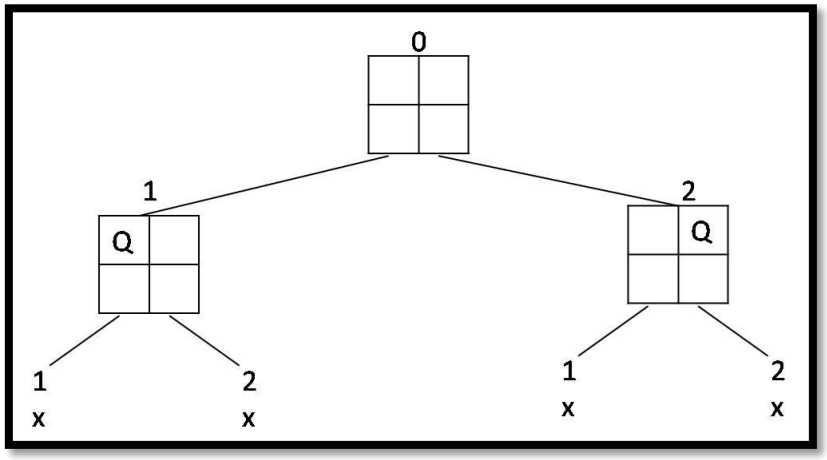
- At some cases, it is better to create a non-recursive version by simulating and transforming the recursive version than attempting to create a non-recursive solution from the problem
- Some of the function calls can be replaced with inline codes so as the reduce the usage of stack

Backtracking Example: N Queens Problem

Task: Place 'n' queens on an 'n x n' board such that no queen attacks diagonally, vertically or horizontally.

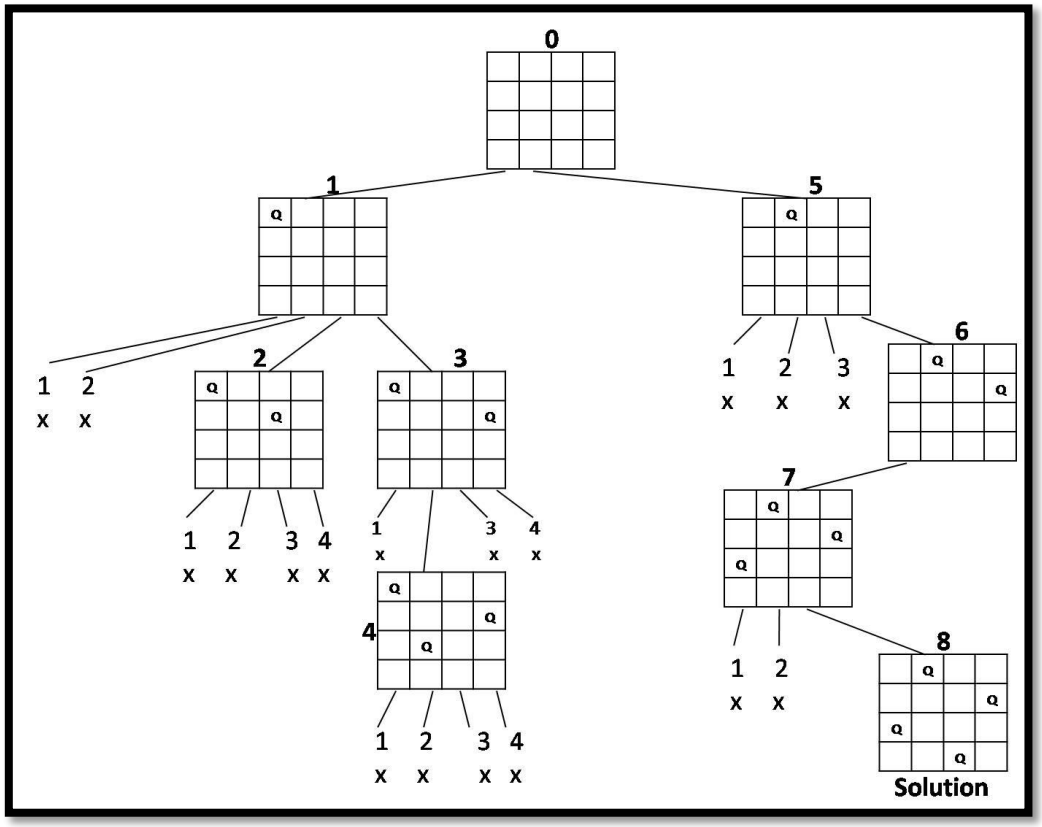
For a one queen problem, we need to place the queen on 1 x 1 board. And the solution is:

For a 2 queen and 3 queen problem, we do not have a solution. Let us look at the procedure for 2 queen problem.



Above shown is the 'state-space tree' for 2 queen problem. Nodes in the tree are solution instances in the order generated. '0' is the initial node in state space tree. Node '1' has placed successfully the first queen. 'x' indicates an unsuccessful attempt to put a queen in the indicated column. As we cannot place the 2nd queen in node '1', we backtrack and place 1st queen in next avail position. However that fails too! Similar cases arise for a 3-queen problem too. Can you draw the state space tree and verify the solution?

State space tree for 4 Queen Problem:



Important Note:

- This is only summary notes. Refer class notes for more problems and examples.
- Refer to handouts given in class
- Refer to homework problems

More Recursion Examples

1. Find sum of N natural numbers using recursion

```
#include <stdio.h>
int sum(int num) {
    if (num!=0)
        return num + sum(num-1);
    else
        return num;
}

int main() {
    int num, result;
    printf("Enter a number: \n");
    scanf("%d", &num);
    result = sum(num);
    printf("sum=%d", result);
}
```

2. Find GCD of two numbers using recursion

```
#include <stdio.h>
int gcd(int n1, int n2)
{
    if (n2 != 0)
       return gcd(n2, n1%n2);
    else
       return n1;
}

int main()
{
   int n1, n2;
   int result;
   printf("Enter two positive integers: ");
   scanf("%d %d", &n1, &n2);

   result = gcd(n1, n2);
   printf("GCD of %d and %d is %d", n1, n2, result);
   return 0;
}
```

3. Compute factorial of a number using recursion

```
#include <stdio.h>

long long fact(int n) {
    if (n >= 1)
        return n * fact(n-1);
    else
        return 1;
}

int main() {
    int n;
    printf("Enter a number\n");
    scanf("%d", &n);
    printf("Factorial of %d = %lld", n, fact(n));
    return 0;
}
```

4. Calculate the length of a string using recursion

```
#include <stdio.h>
int getlength(char *str) {
    static int length=0;
    if(*str != '\0'){
        length++;
        str++;
        getlength(str);
    }
    else
        return length;
}

int main() {
    char str[100];
    int length=0;
    printf("Enter a string: ");
    scanf("%s", str);
    length = getlength(str);
    printf("The length is %d\n",length);
    return 0;
}
```

5. Recursive program to count the number of digits in a given number

```
#include <stdio.h>
int count_digits(int num) {
    static int count=0;
    if(num>0) {
        count++;
        count_digits(num/10);
    }
    else
        return count;
}

int main() {
    int num;
    int count=0;

    printf("Enter a number: ");
    scanf("%d",&num);
    count=count_digits(num);
    printf("Total digits in number %d is: %d\n", num, count);

    return 0;
}
```

6. Compute largest number in an array using recursion

```
#include <stdio.h>

int array_largest(int a[], int lower, int upper)
{
    int max;
    if (lower == upper)
        return a[lower];
    else {
        max = array_largest(a, lower + 1, upper);
        if (a[lower] >= max)
            return a[lower];
        else
            return max;
    }
}
```

```
int main()
{
    int a[10] = { 23, 43, 35, 38, 67, 12, 76, 10, 34, 8 };
    printf("The largest in array is %d", array_largest(a, 0, 9));
    return 0;
}
```

7. Reverse a string using recursion

```
# include <stdio.h>
void reverse(char *str) {
   if (*str) {
       reverse(str+1);
       printf("%c", *str);
   }
}

int main()
{
   char a[] = "DSA";
   reverse(a);
   return 0;
}
```

8. Nth Fibonacci number using recursion

```
#include<stdio.h>
int fib(int n) {
   if (n <= 1)
      return n;
   return fib(n-1) + fib(n-2);
}

int main ()
{
  int n;
  printf("Enter n\n");
  scanf("%d", &n);
  printf("%d", fib(n));
  return 0;
}
```

9. Sum of array elements using recursion

```
#include <stdio.h>
int array_sum(int a[], int n)
{
    if (n <= 0)
        return 0;
    return (array_sum(a, n - 1) + a[n - 1]);
}

int main()
{
    int a[] = { 1, 2, 3, 4, 5 };
    int n = sizeof(a) / sizeof(a[0]);
    printf("%d\n", array_sum(a, n));
    return 0;
}
```

10. Check if the number is prime using recursion

```
#include<stdio.h>

int i;

int is_prime(int n)
{
    if(i==1)
        return 1;
    else if(n % i == 0)
         return 0;
    else {
         i = i - 1;
         is_prime(n);
    }
}

int main()
{
    int n, prime;
    printf("Enter a number\n");
    scanf("%d",&n);
```

```
    i = n / 2;
    prime = is_prime(n);
   if(prime == 1)
        printf("Number is Prime\n");
   else
      printf("Number is Not Prime\n");
   return 0;
}
```

~*~*~*~*~*~*~*~*~