CSE 1201
Data Structure

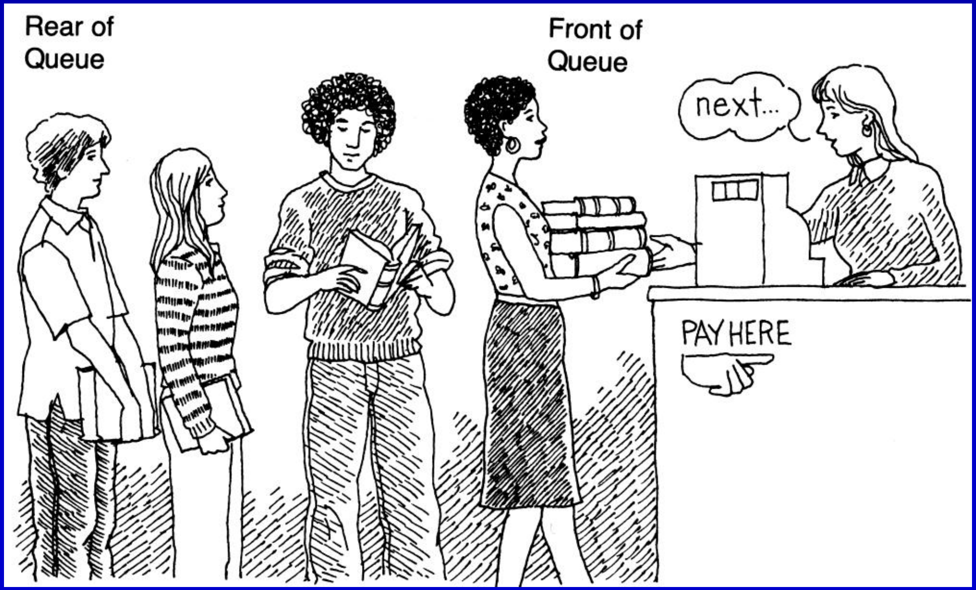
Chapter 4: Queues

Instructor: Md. Shahid Uz Zaman

Dept. of CSE, RUET

What is a queue?

- It is an ordered group of homogeneous items of elements.
- Queues have two ends:
  - Elements are added at one end called ***rear***.
  - Elements are removed from the other end called ***front***.
- The element added first is also removed first (**FIFO**: First In, First Out).



Queue Specification

- <u>Definitions</u>: (provided by the user)
  - *MAX_ITEMS*: Max number of items that might be on the queue
  - *ItemType*: Data type of the items on the queue
- <u>Operations</u>
  - Enqueue (ItemType newItem)
  - Dequeue ()

Queue Operation

Enqueue (ItemType newItem)

- *Function*: Adds newItem to the rear of the queue.
- *Preconditions*: Queue has been initialized and is not full.
- *Postconditions*: newItem is at rear of queue.

Queue Operation

Dequeue (ItemType& item)

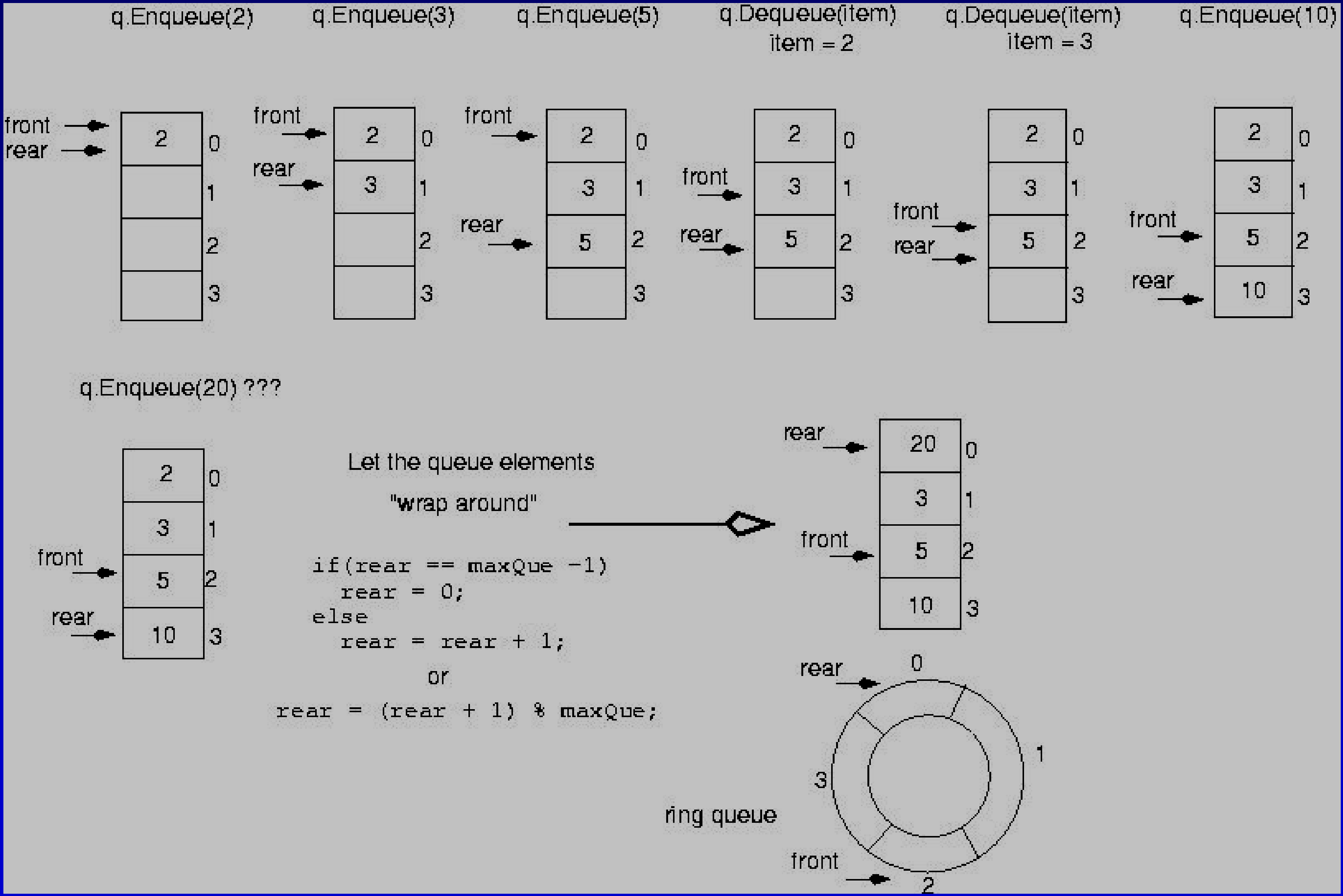
- *Function*: Removes front item from queue and returns it in item.
- *Preconditions*: Queue has been initialized and is not empty.
- *Postconditions*: Front element has been removed from queue and item is a copy of removed element.

Queue Operation

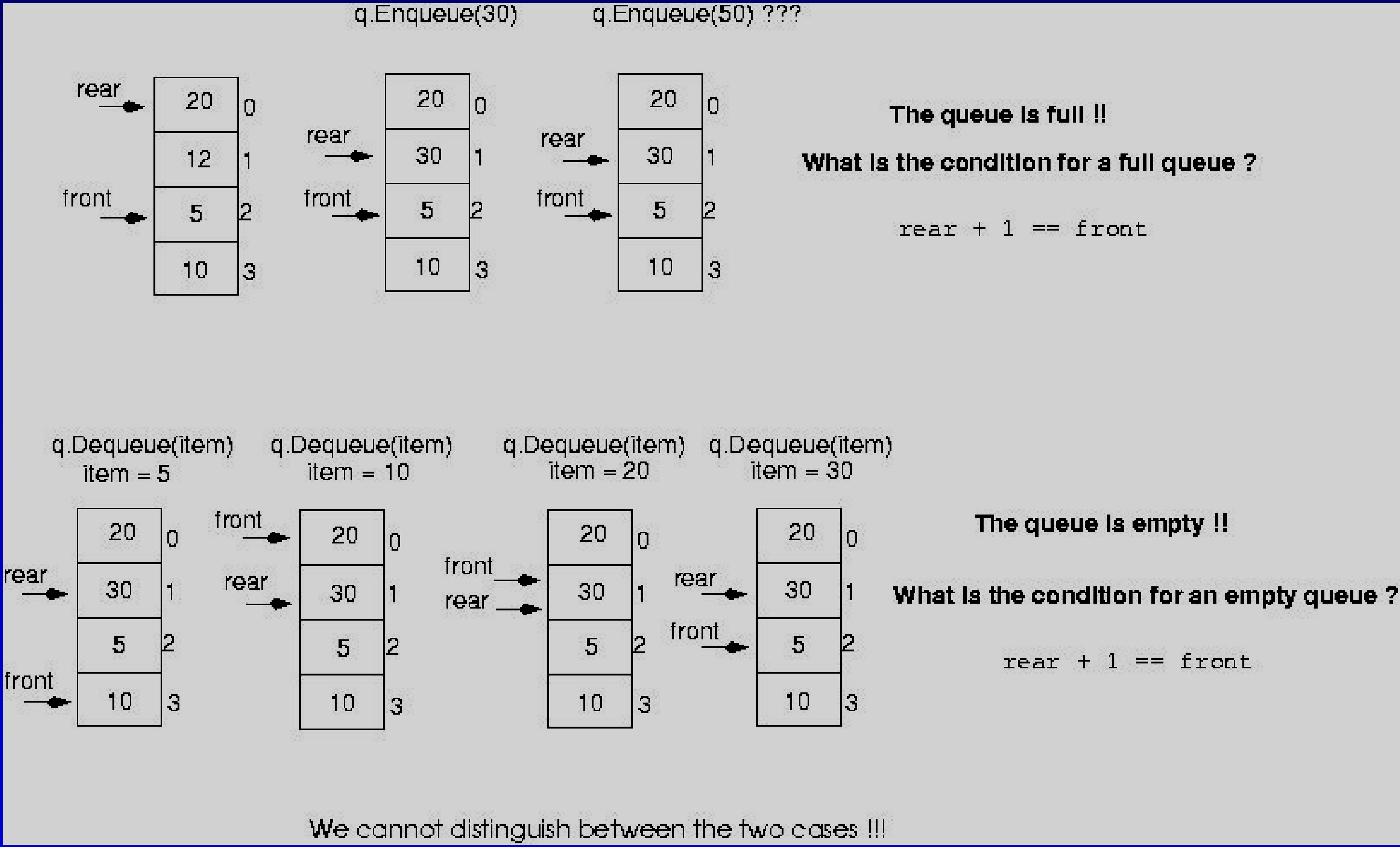
Implementation issues

- Implement the queue as a *circular structure*.
- How do we know if a queue is full or empty?
- Initialization of *front* and *rear*.
- Testing for a *full* or *empty* queue.

Queue Operation

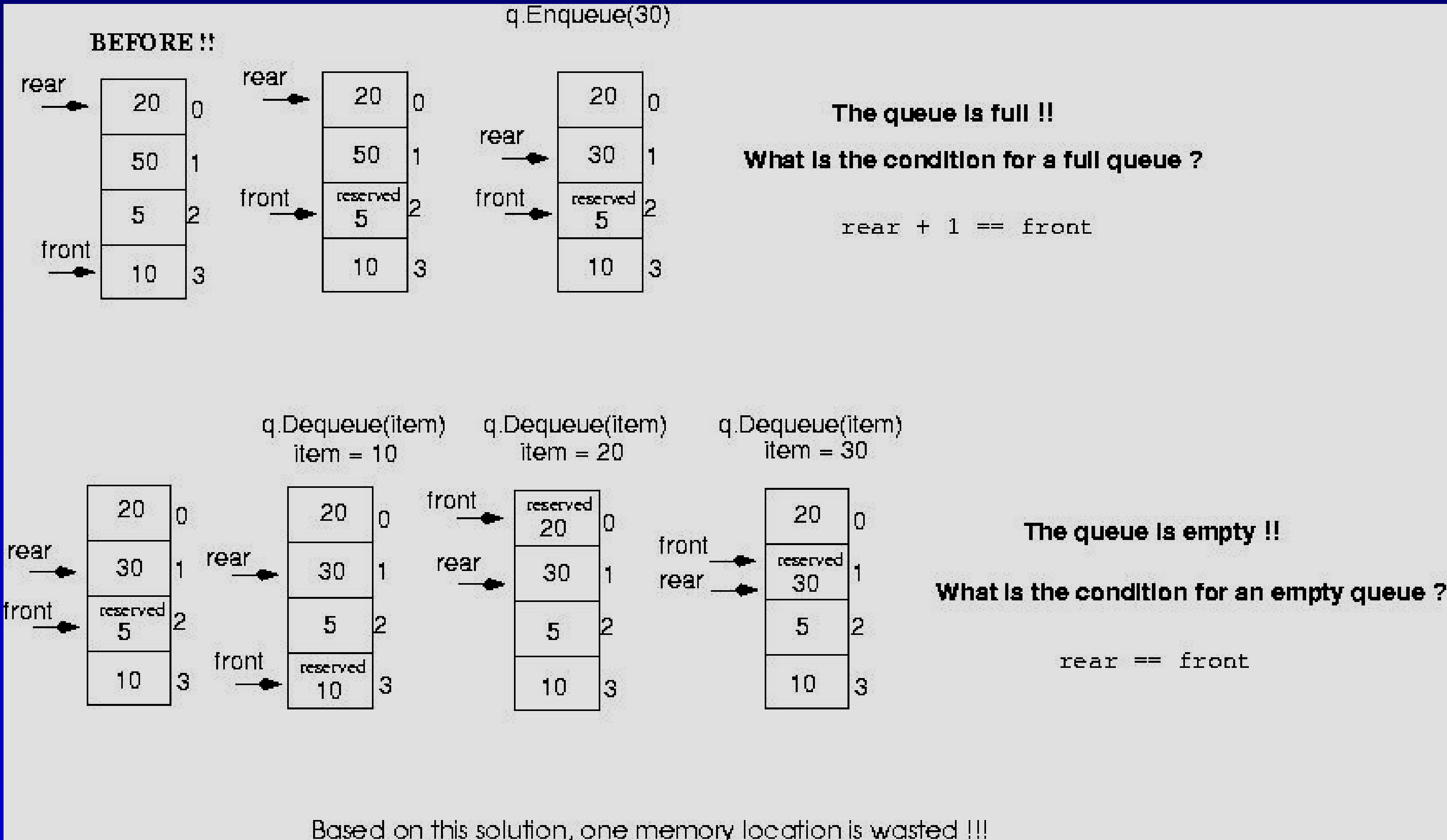


Queue Operation



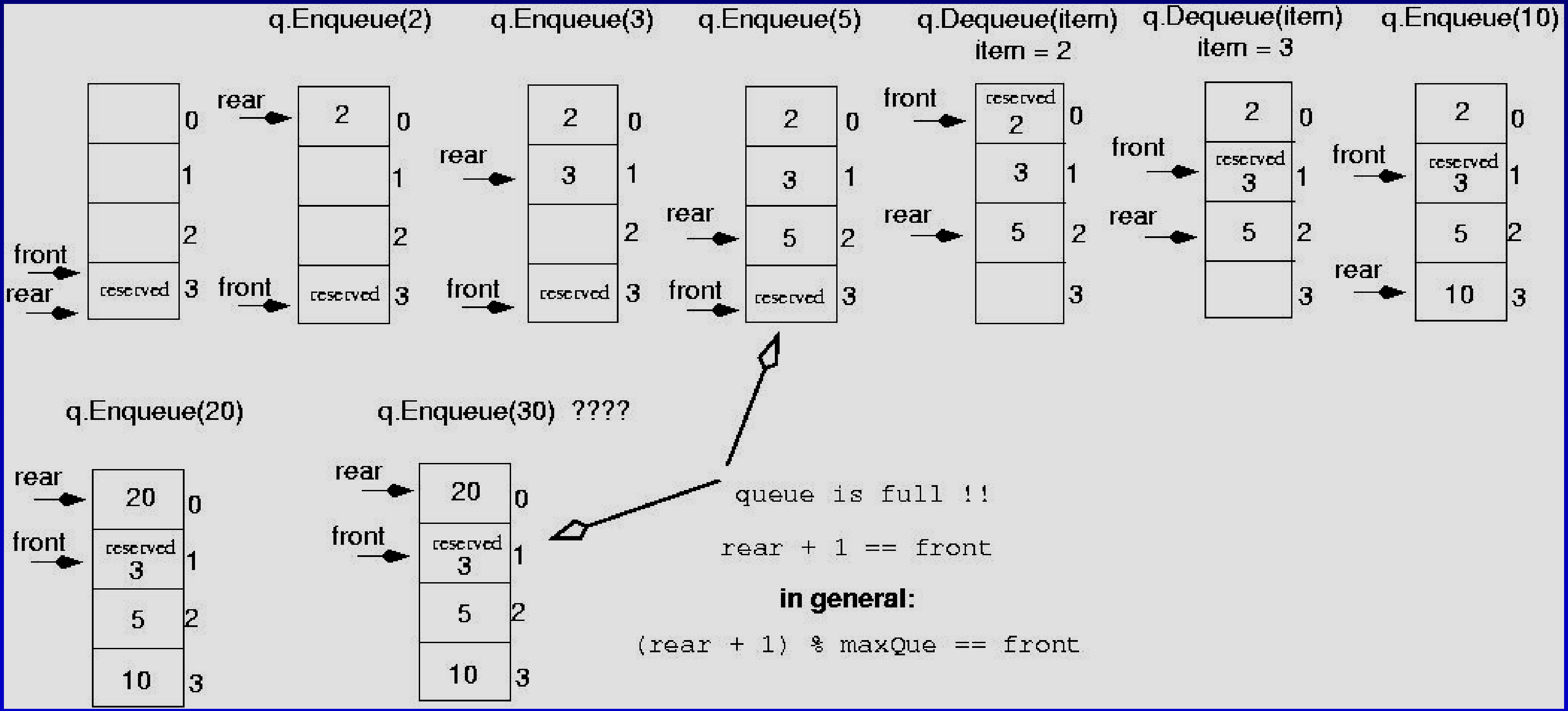
Queue Operation

Make *front* point to the element **preceding** the front element in the queue (one memory location will be wasted).

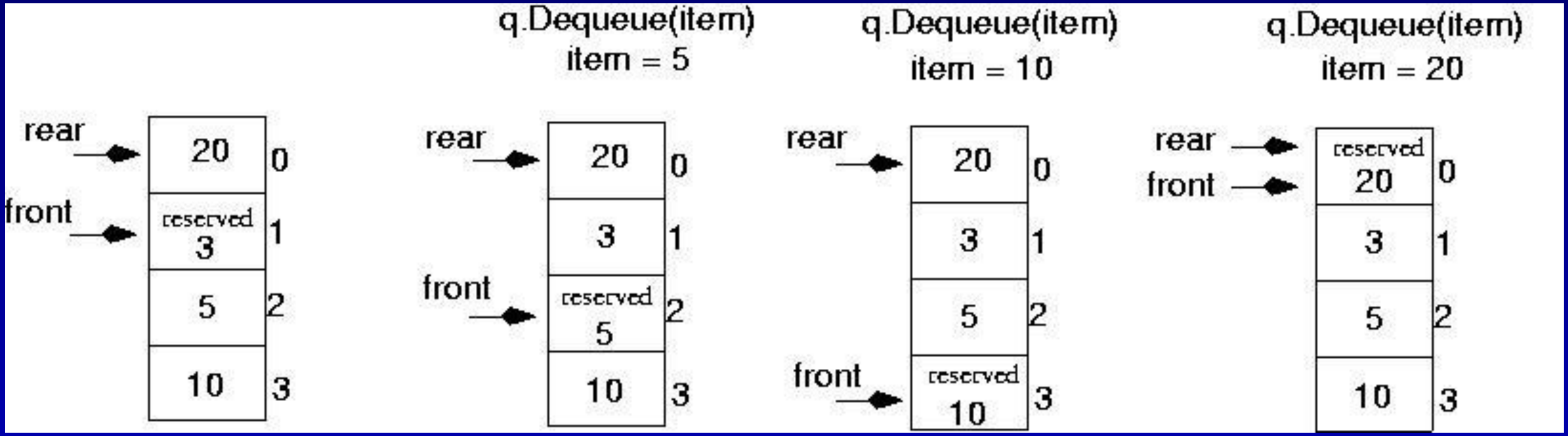


Queue Operation

Initialize *front* and *rear*



Queue Operation



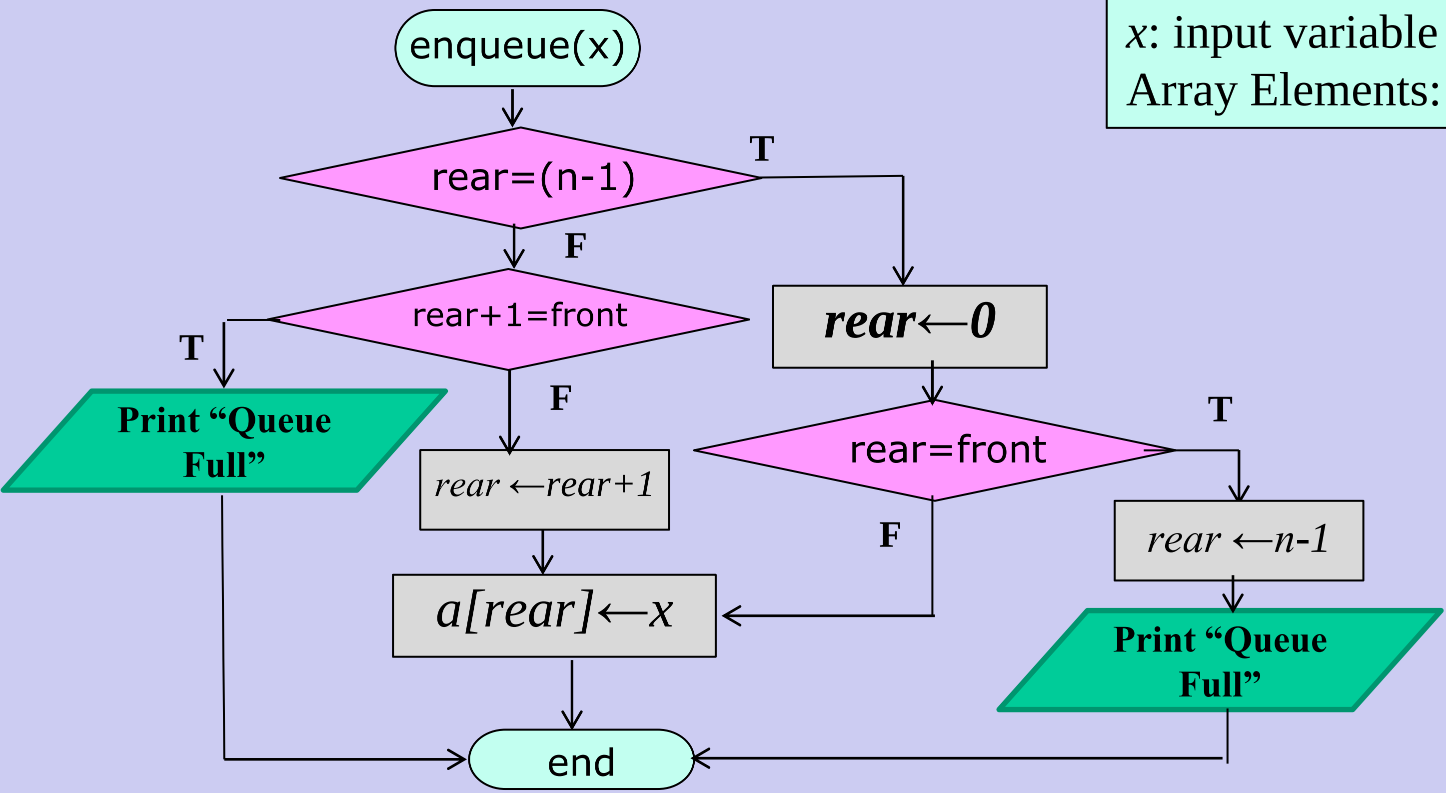
Queue is empty now!!

rear == front

Queue: Enqueue() function

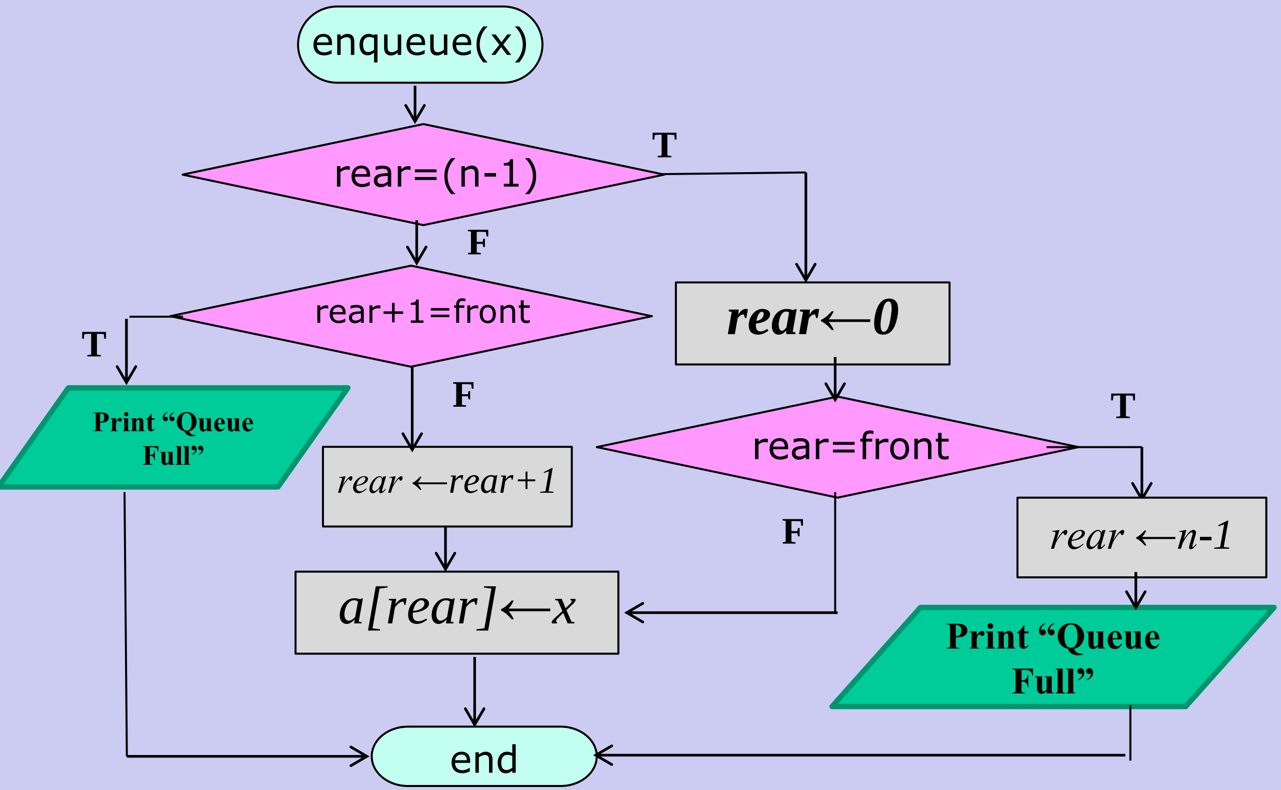
Topic 1: Write an Algorithm to insert a new element in a queue

n: size of *a[]*
x: input variable
Array Elements: *a*[0]…..*a*[*n*-1]



Queue : Enqueue() function

Topic 1: Write an Algorithm to insert a new element in a queue



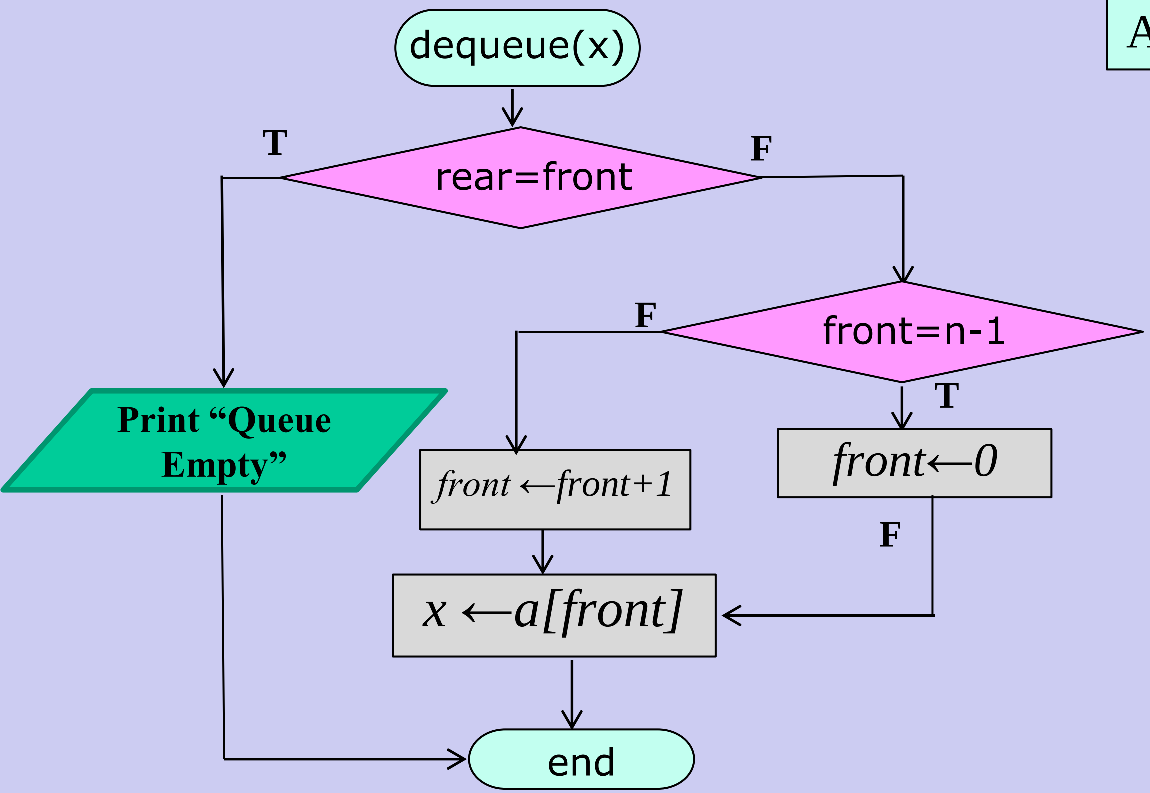

```
void enqueue(int x){
 if(rear==n-1){
   rear=0;
   if(rear==front)
      {printf("Queue is full.\n");
        rear=n-1;}
   else
      a[rear]=x;
 }
 else{
   if(rear+1==front)
      printf("Queue is full.\n");
   else
      a[++rear]=x;
 }
}
```

Initially top = -1

Queue : Dequeue() function

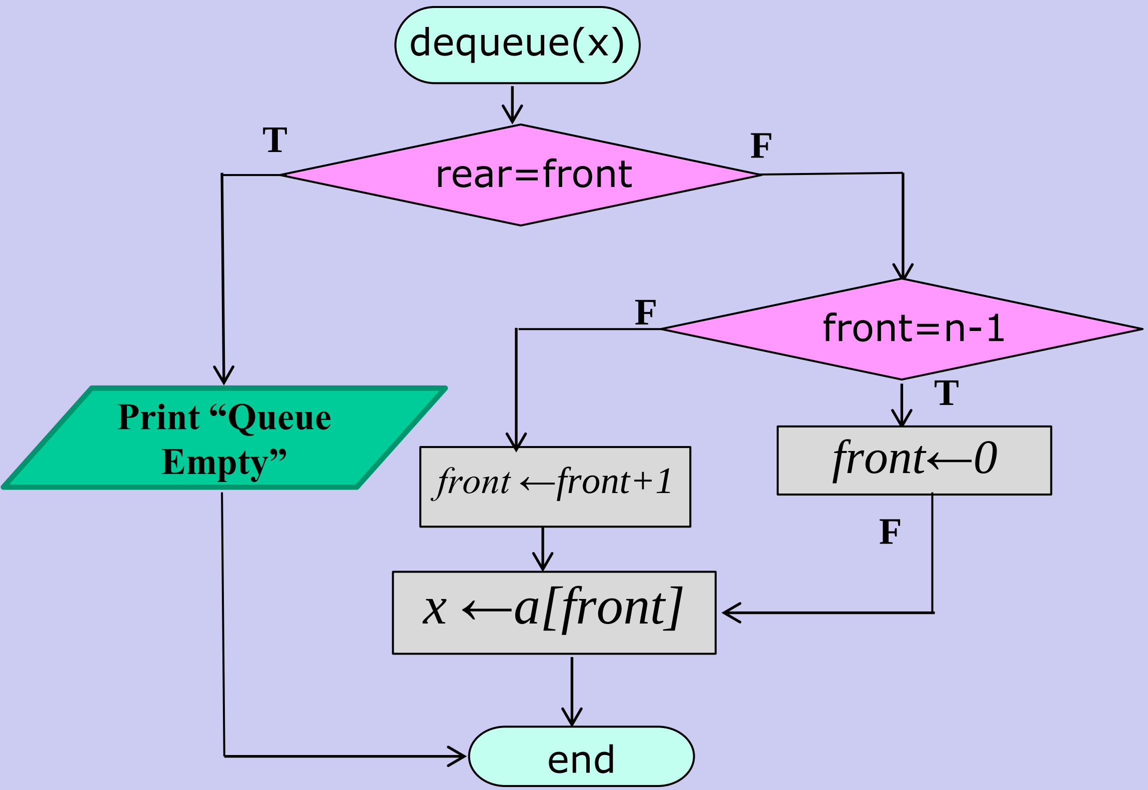
Topic 1: Write an Algorithm to delete an from a queue

n: size of *a[]*
Array Elements: *a*[0]…..*a*[*n*-1]



Queue: Dequeue() function

Topic 1: Write an Algorithm to delete an from a queue




```
void dequeue(){
   int x;
 if(rear==front)
   printf("Queue is empty..\n");
 else
 {
    if(front==n-1)
      front=0;
    else
      front++;
    x=a[front];
 }
}
```

Recursion

What is recursion?

- Sometimes, the best way to solve a problem is by solving a smaller version of the exact same problem first
- Recursion is a technique that solves a problem by solving a smaller problem of the same type

Problems defined recursively

- There are many problems whose solution can be defined recursively

Example: *n factorial*

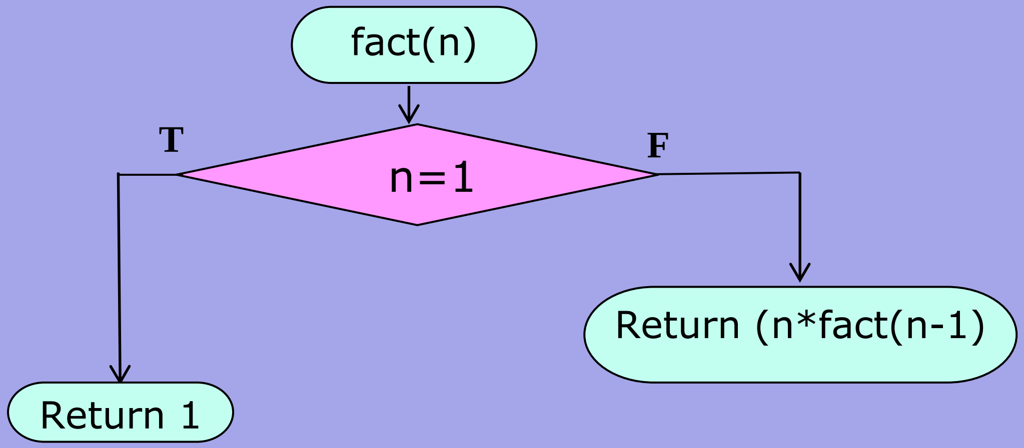
$$n! = \begin{cases} 1 & \text{if } n = 0 \\ (n-1)! * n & \text{if } n > 0 \end{cases}$$

(*recursive* solution)

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ 1*2*3*\ldots*(n-1)*n & \text{if } n > 0 \end{cases}$$

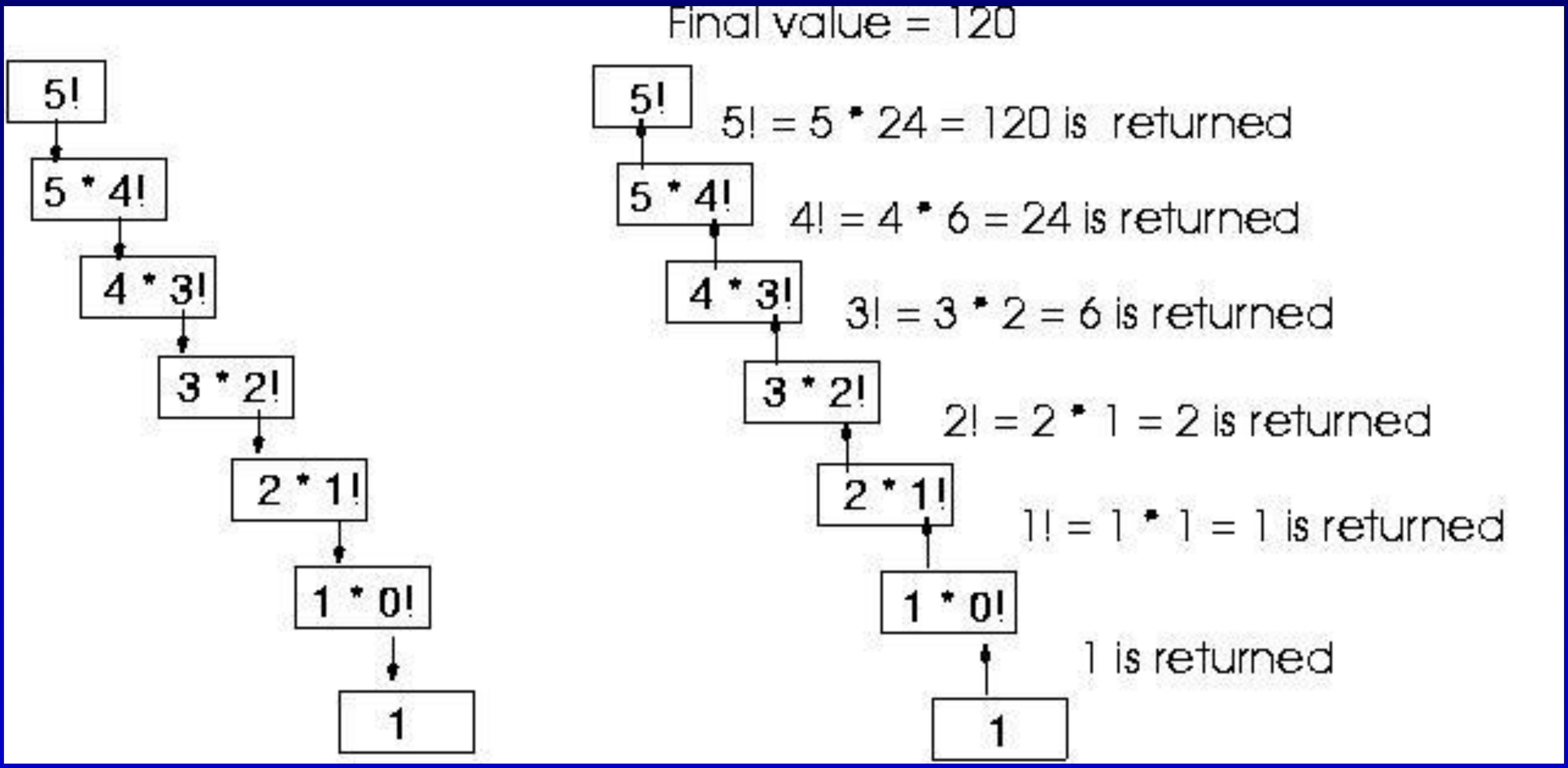
(*closed form* solution)

Coding the factorial function



- Recursive implementation

```
int fact( int n)
{
 if (n==0)  // base case
   return 1;
 else
   return n * factl(n-1);
}
```


Final value = 120
5!
5 * 4!
4 * 3!
3 * 2!
2 * 1!
1 * 0!
1
5!
5! = 5 * 24 = 120 is returned
5 * 4!
4! = 4 * 6 = 24 is returned
4 * 3!
3! = 3 * 2 = 6 is returned
3 * 2!
2! = 2 * 1 = 2 is returned
2 * 1!
1! = 1 * 1 = 1 is returned
1 * 0!
1 is returned
1

Coding the factorial function (cont.)

- Iterative implementation

```
int Factorial(int n)
{
 int fact = 1;

 for(int count = 2; count <= n; count++)
   fact = fact * count;

 return fact;
}
```

Another example:
n choose *k* (combinations)

- Given *n* things, how many different sets of size *k* can be chosen?

$$\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}, \quad 1 < k < n \qquad \textit{(recursive solution)}$$

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}, \quad 1 < k < n \qquad \textit{(closed-form solution)}$$

with base cases:

$$\binom{n}{1} = n \ \ (k = 1), \quad \binom{n}{n} = 1 \ \ (k = n)$$

Recursion vs. iteration

- Iteration can be used in place of recursion
  - An iterative algorithm uses a *looping construct*
  - A recursive algorithm uses a *branching structure*
- Recursive solutions are often less efficient, in terms of both *time* and *space*, than iterative solutions
- Recursion can simplify the solution of a problem, often resulting in shorter, more easily understood source code

The Fibonacci Sequence

The Fibonacci Sequence

He gave us our 10 digit number system!

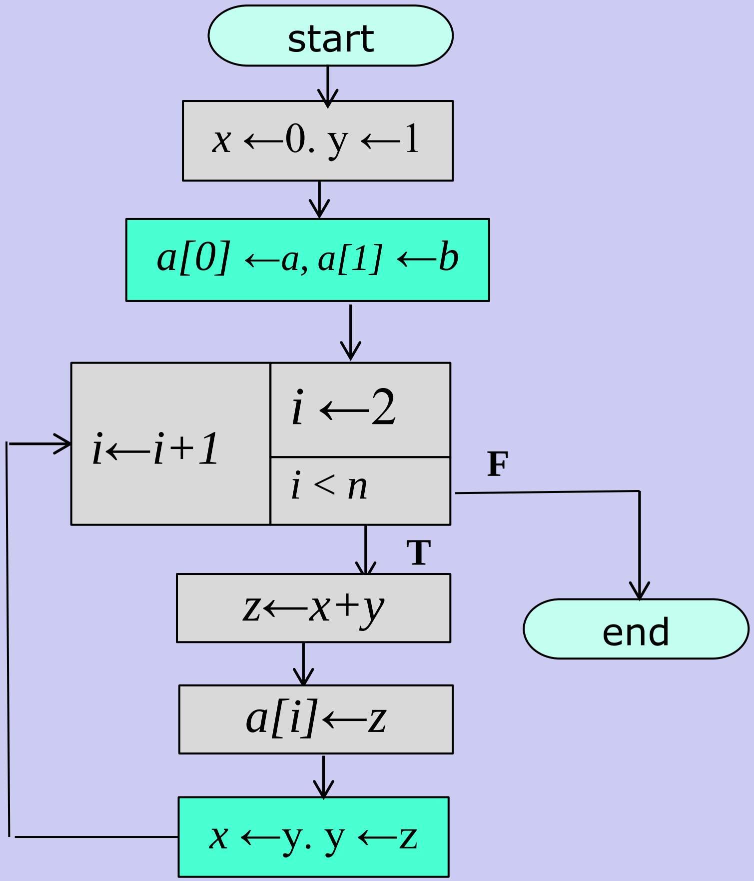
He recognized a series of numbers that often occur in nature. These are now called the Fibonacci numbers.   The series starts with 0 & 1. All following numbers are the sum of the 2 previous numbers!

Here's how it starts...
0,1,1,2,3,5.....

Leonardo of Pisa
or Fibonacci:
Born 1175 AD

Fibonacci Series

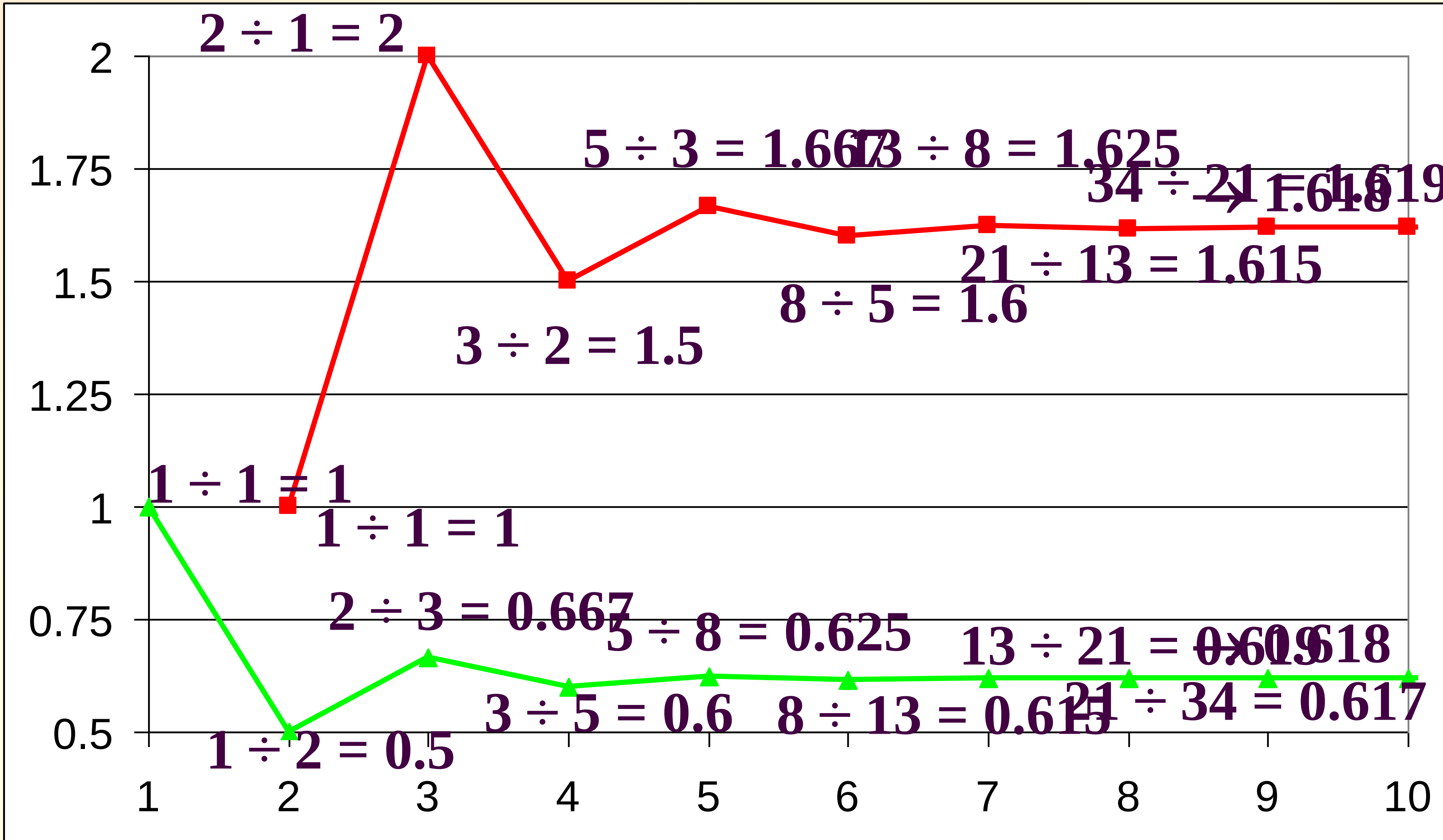
Topic 1: Write an Algorithm to create n elements Fibonacci series

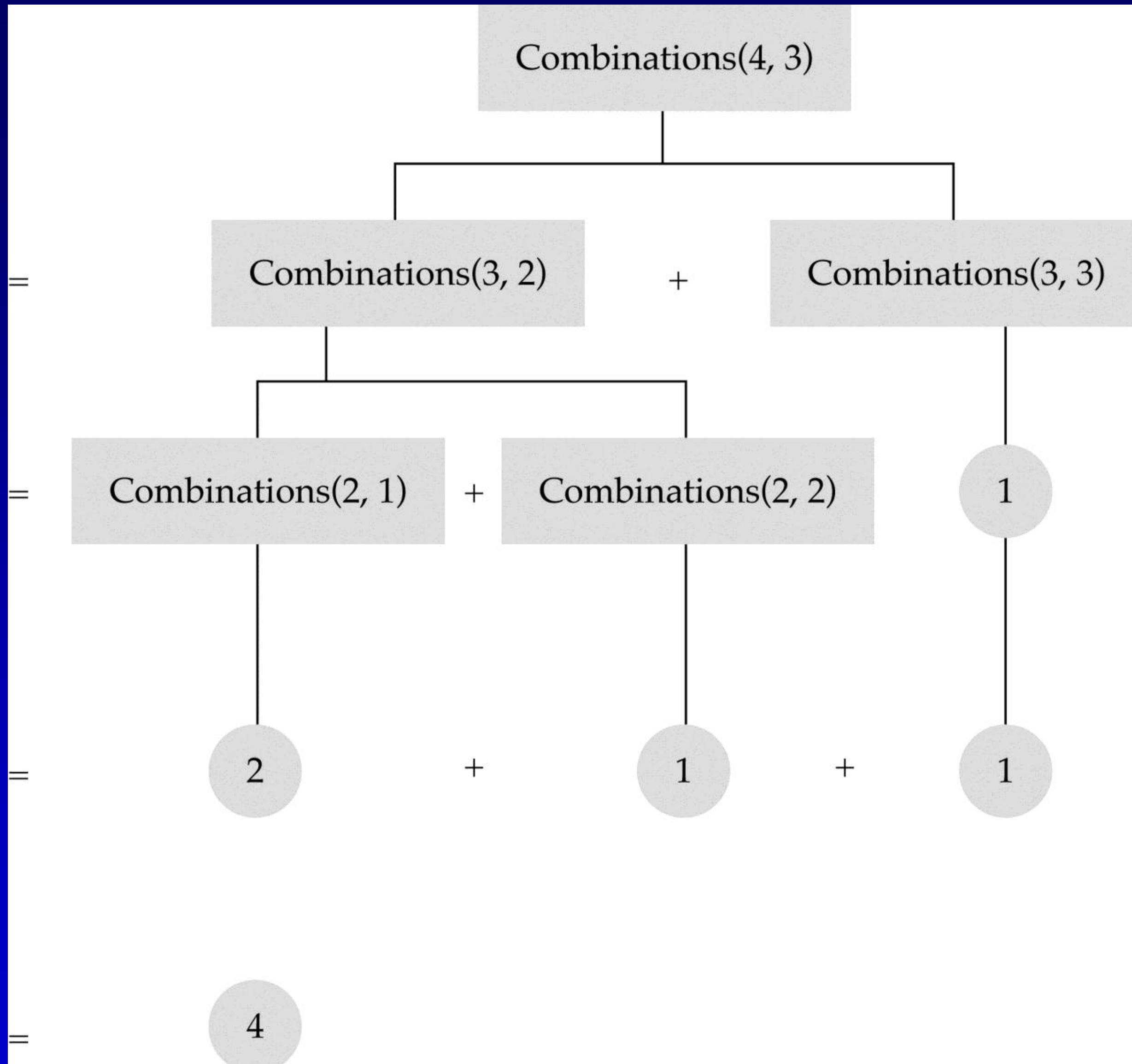


n: total elements
a[i]: stores elements

Fibonacci Series

- Find the ratio of successive Fibonacci numbers:
  - 1 : 1, 2 : 1, 3 : 2, 5 : 3, 8 : 5, …
  - 1 : 1, 1 : 2, 2 : 3, 3 : 5, 5 : 8, …
- What do you notice?


2 ÷ 1 = 2
5 ÷ 3 = 1.667
13 ÷ 8 = 1.625
34 ÷ 21 = 1.619
→ 1.618
21 ÷ 13 = 1.615
8 ÷ 5 = 1.6
3 ÷ 2 = 1.5
1 ÷ 1 = 1
1 ÷ 1 = 1
2 ÷ 3 = 0.667
5 ÷ 8 = 0.625
13 ÷ 21 = 0.619
→ 0.618
21 ÷ 34 = 0.617
3 ÷ 5 = 0.6
8 ÷ 13 = 0.615
1 ÷ 2 = 0.5
2
1.75
1.5
1.25
1
0.75
0.5
1
2
3
4
5
6
7
8
9
10


Combinations(4, 3)
= Combinations(3, 2) + Combinations(3, 3)
= Combinations(2, 1) + Combinations(2, 2) 1
= 2 + 1 + 1
= 4

Look at the following rectangles:

Now ask yourself, which of them seems to be the most naturally attractive rectangle? If you said the first one, then you are probably the type of person who likes everything to be symmetrical. Most people tend to think that **the third rectangle is the most appealing.**

If you were to measure each rectangle's length and width, and compare the ratio of length to width for each rectangle you would see the following:

Rectangle one: Ratio 1:1

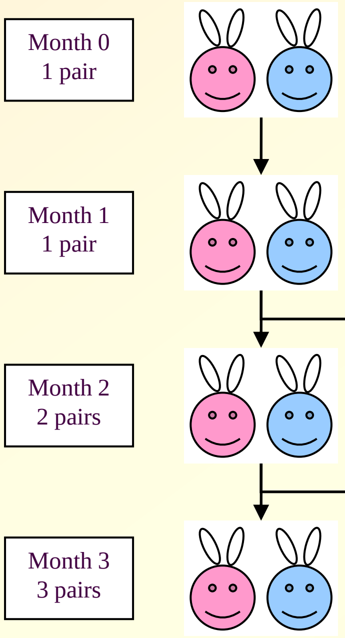
Rectangle two: Ratio 2:1

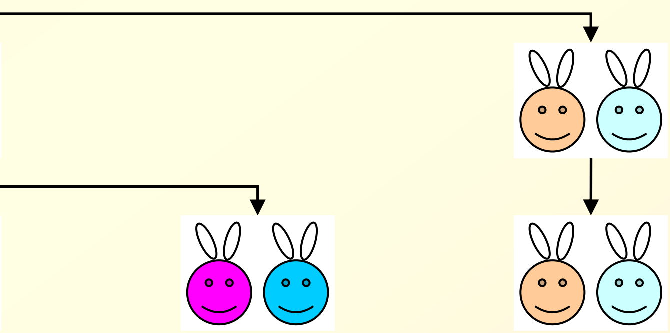
Rectangle Three: Ratio 1.618:1

Have you figured out why the third rectangle is the most appealing? That's right - because the ratio of its length to its width is the Golden Ratio! For centuries, designers of art and architecture have recognized the significance of the Golden Ratio in their work.


1 : 1.618

Fibonacci numbers



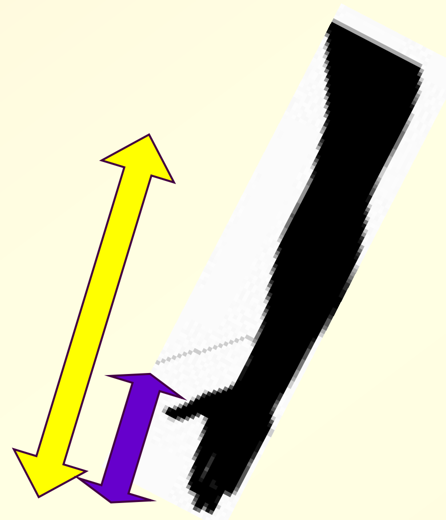


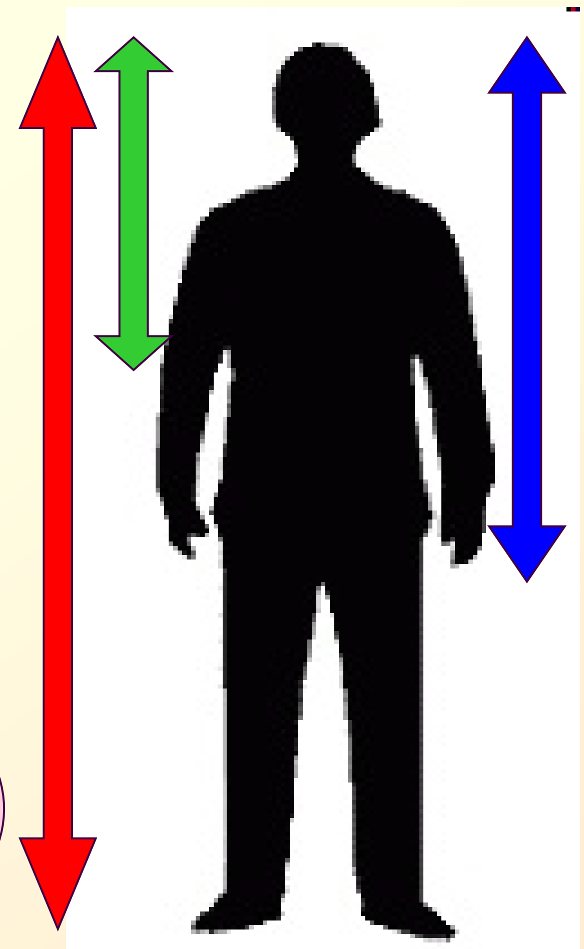
Are our bodies based on Fibonacci numbers?

What do you notice?

Find the ratio of

- Height (red) : Top of head to fingertips (blue)
- Top of head to fingertips (blue) : Top of head to elbows (green)
- Length of forearm (yellow) : length of hand (purple)





Report back at next class

Ackermann’s Function

Function Definition

Ackermann's function was defined in 1920s by German mathematician and logician Wilhelm Ackermann (1896-1962).

$A(m,n)$, $m,n \in \mathbf{N}$ such that**,**

$$
\begin{aligned}
&A(0, n) = n + 1, && n \geq 0; \\
&A(m,0) = A(m-1, 1), && m > 0; \\
&A(m,n) = A(m-1, A(m, n-1)), && m, n > 0;
\end{aligned}
$$

Ackermann's Function

Example - 1

$$
\begin{aligned}
A(1, 2) &= A(0, A(1, 1)) \\
&= A(0, A(0, A(1, 0))) \\
&= A(0, A(0, A(0, 1))) \\
&= A(0, A(0, 2)) \\
&= A(0, 3) \\
&= 4
\end{aligned}
$$

Simple addition and subtraction!!

Ackermann's Function

Example - 2

A (2, 2) = A (1, A (2, 1))
= A (1, A (1, A (2, 0)))
= A (1, A (1, A (1, 1)))
= A (1, A (1, A (0, A (1, 0))))
= A (1, A (1, A (0, A (0, 1))))
= A (1, A (1, A (0, 2)))
= A (1, A (1, 3))
= A (1, A (0, A (1, 2)))
= A (1, A (0, A (0, A (1, 1))))
= A (1, A (0, A (0, A (0, A (1, 0)))))
= A (1, A (0, A (0, A (0, A (0, 1)))))
= A (1, A (0, A (0, A (0, 2))))
= A (1, A (0, A (0, 3)))
= A (1, A (0, 4))

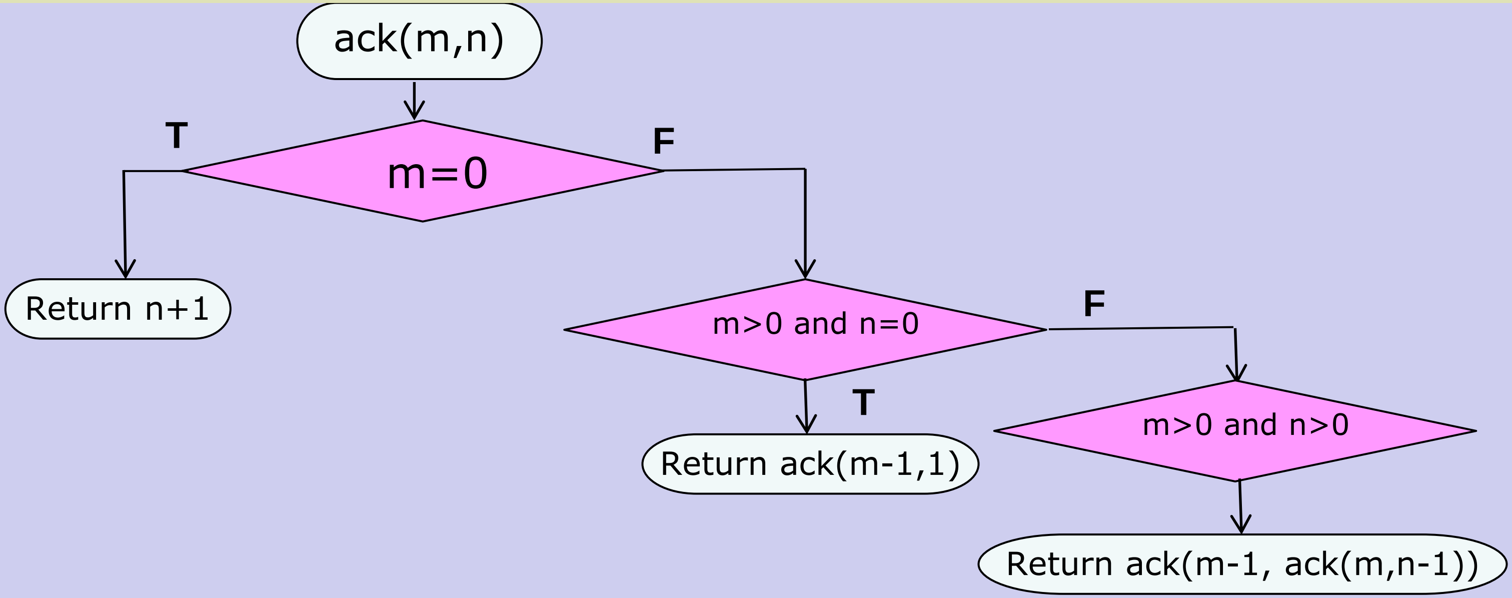
= A (1, 5)
= A (0, A (1, 4))
= A (0, A (0, A (1, 3)))
= A (0, A (0, A (0, A (1, 2))))
= A (0, A (0, A (0, A (0, A (1, 1)))))
= A (0, A(0, A(0, A(0, A(0, A(1, 0))))))
= A (0, A(0, A(0, A(0, A(0, A(0, 1))))))
= A (0, A (0, A (0, A (0, A (0, 2)))))
= A (0, A (0, A (0, A (0, 3))))
= A (0, A (0, A (0, 4)))
= A (0, A (0, 5))
= A (0, 6)
= 7

Ackermann's Function

- It is a well defined total function.
- Computable but not primitive recursive.
- Grows faster than any primitive recursive function.
- It is μ-recursive.

A(m,n)	n = 0	n = 1	n = 2	n = 3	n = 4
m = 0	1	2	3	4	5
m = 1	2	3	4	5	6
m = 2	3	5	7	9	11
m = 3	5	13	29	61	125
m = 4	13	65533	$2^{65533} - 3$	A(3, $2^{65533} – 3$)	A(3, A(4,3))
m = 5	65533	A(4, 65533)	A(4, A(5,1))	A(4, A(5,2))	A(4, A(5,3))
m = 6	A(4,65533)	A(5, A(5,1))	A(5, A(6,1)	A(5, A(6,2)	A(5, A(6,3)

Ackermann's Function



- Recursive version

```
function ack (m, n)
   if m = 0
      return n+1
   else if m > 0 and n = 0
      return ack (m-1, 1)
   else if m > 0 and n > 0
      return ack (m-1, ack (m, n-1))
```

Tower of Hanoi

Tower of Hanoi

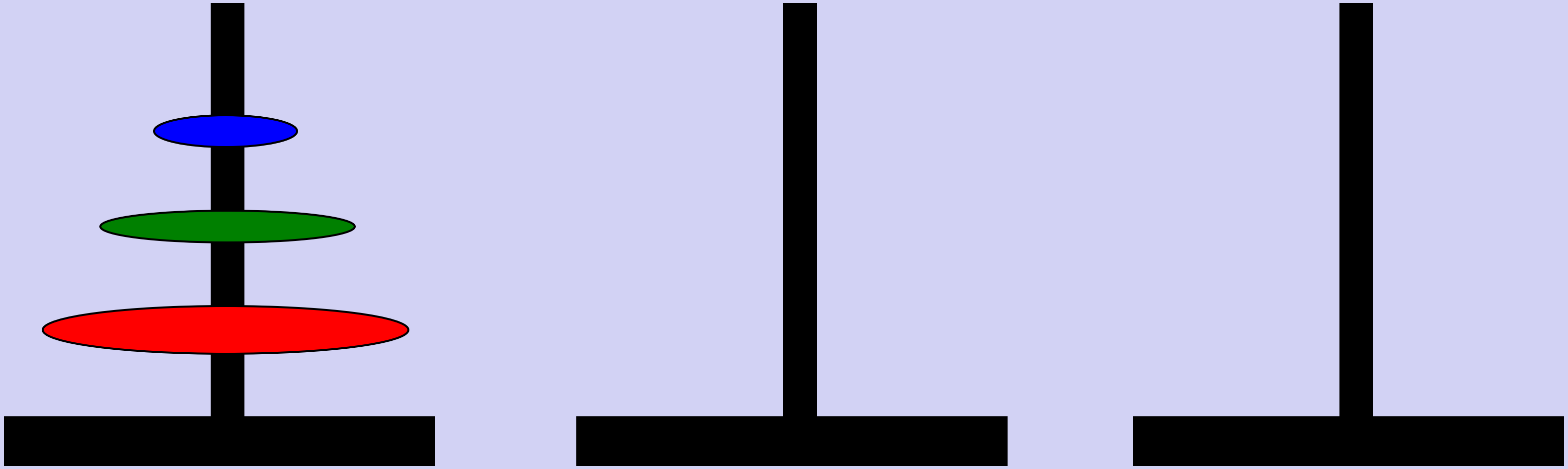
- There are three towers
- 64 gold disks, with decreasing sizes, placed on the first tower
- You need to move all of the disks from the first tower to the last tower
- Larger disks can not be placed on top of smaller disks
- The third tower can be used to temporarily hold disks

Tower of Hanoi

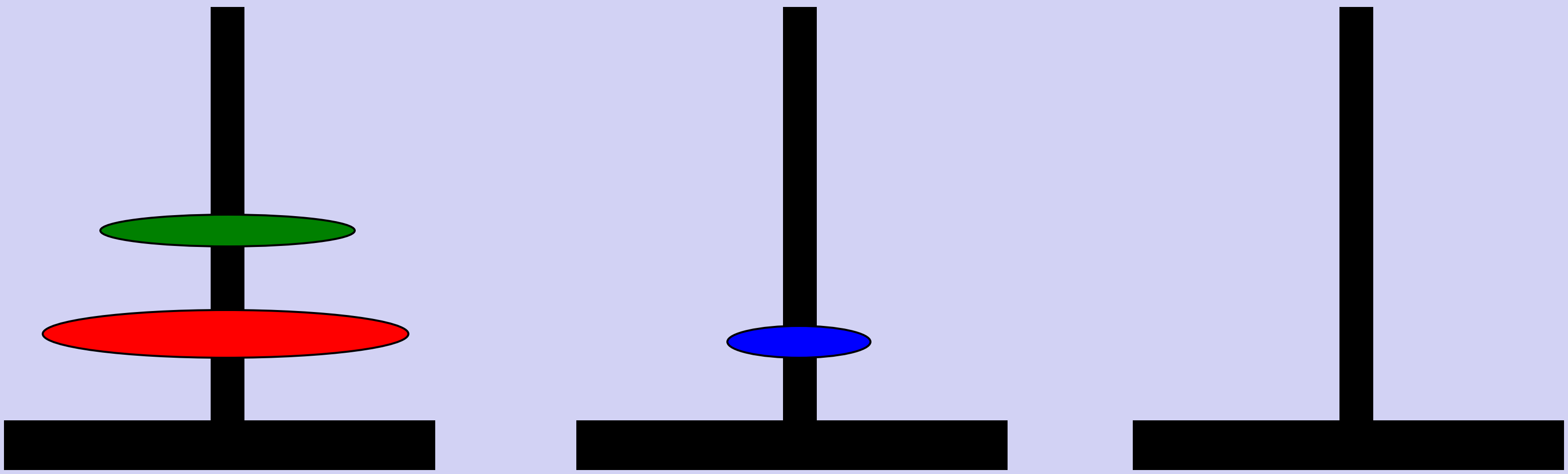
- The disks must be moved within one week. Assume one disk can be moved in 1 second. Is this possible?

- To create an algorithm to solve this problem, it is convenient to generalize the problem to the “N-disk” problem, where in our case N = 64.

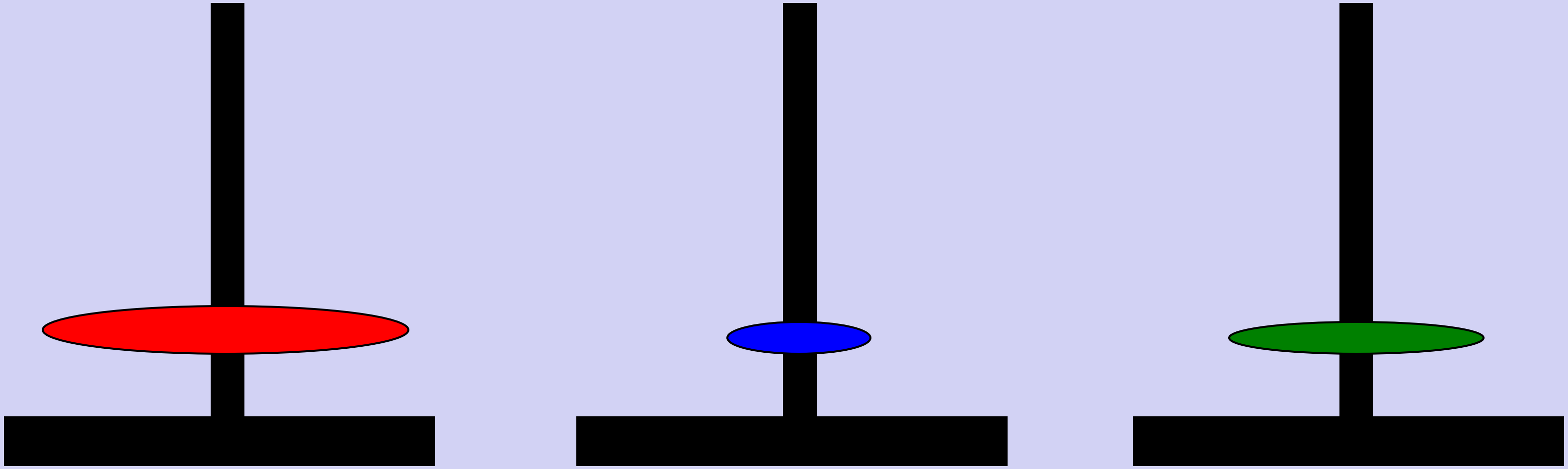
Tower of Hanoi



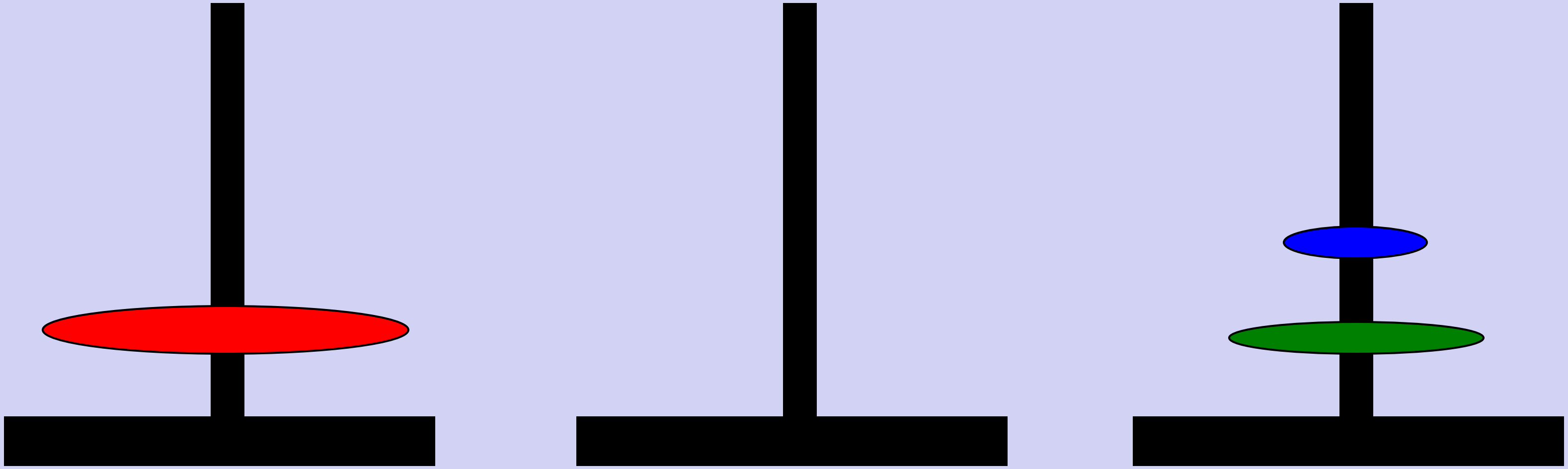
Tower of Hanoi



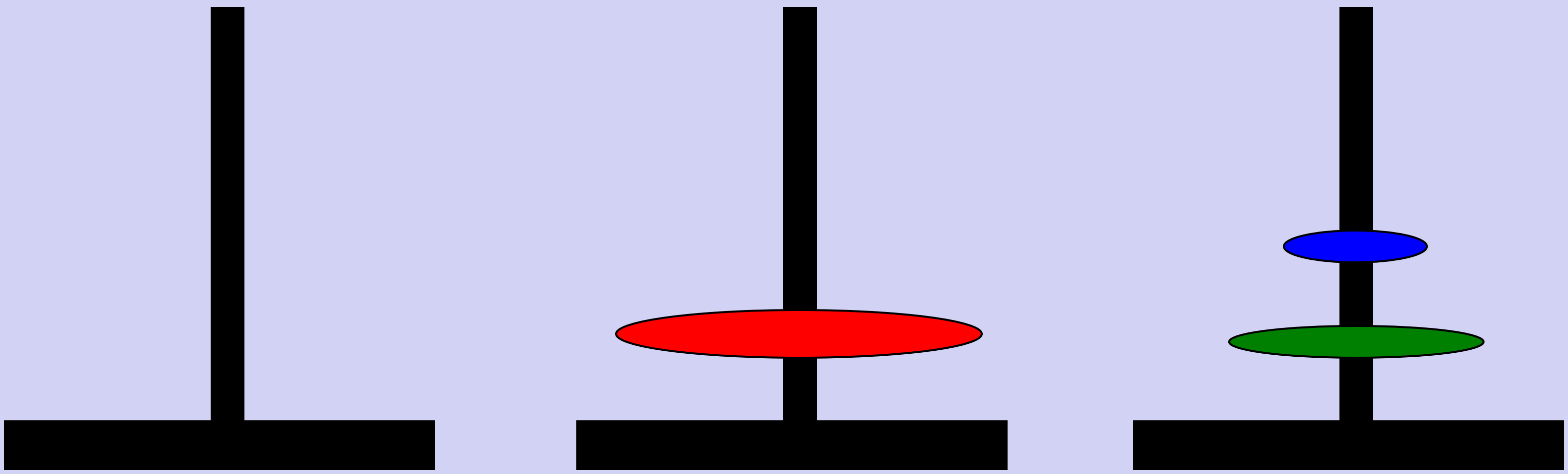
Tower of Hanoi



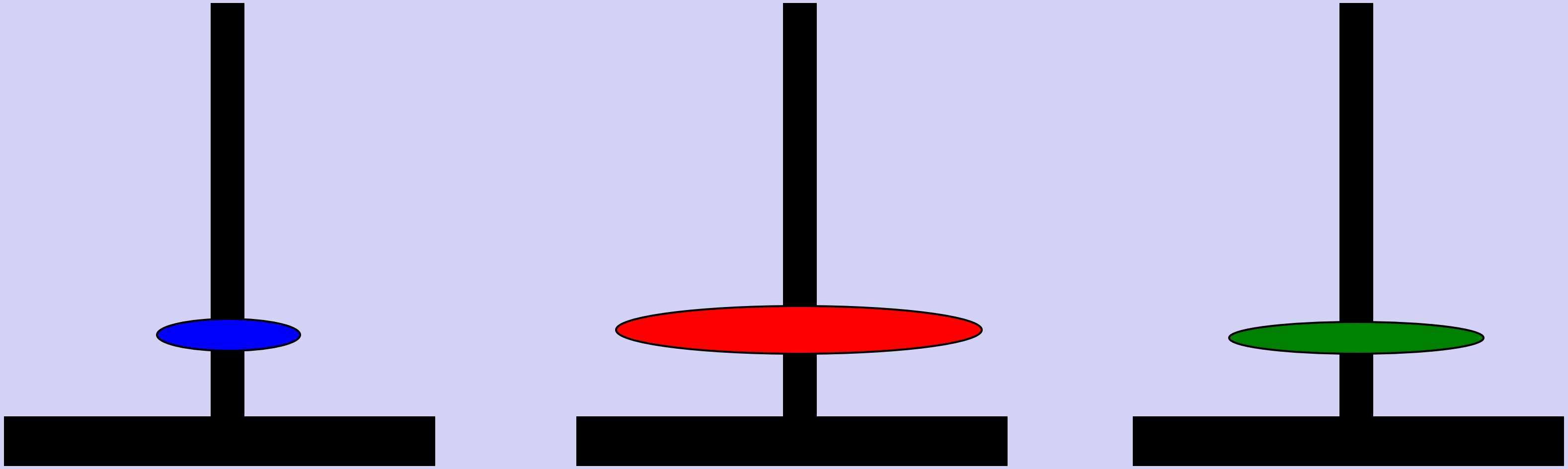
Tower of Hanoi



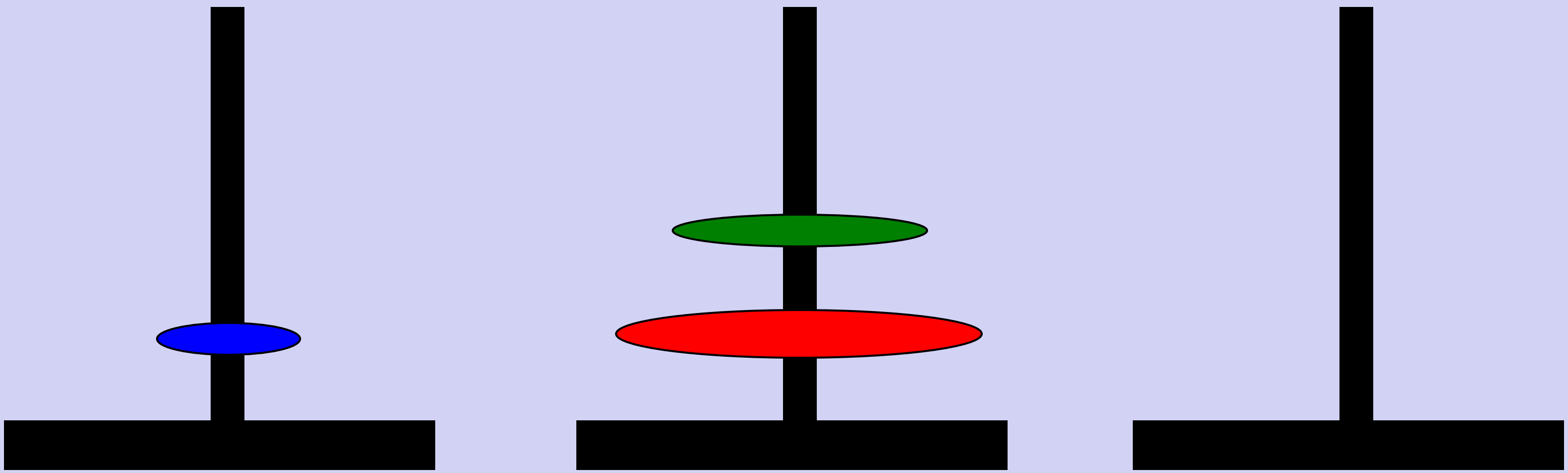
Tower of Hanoi



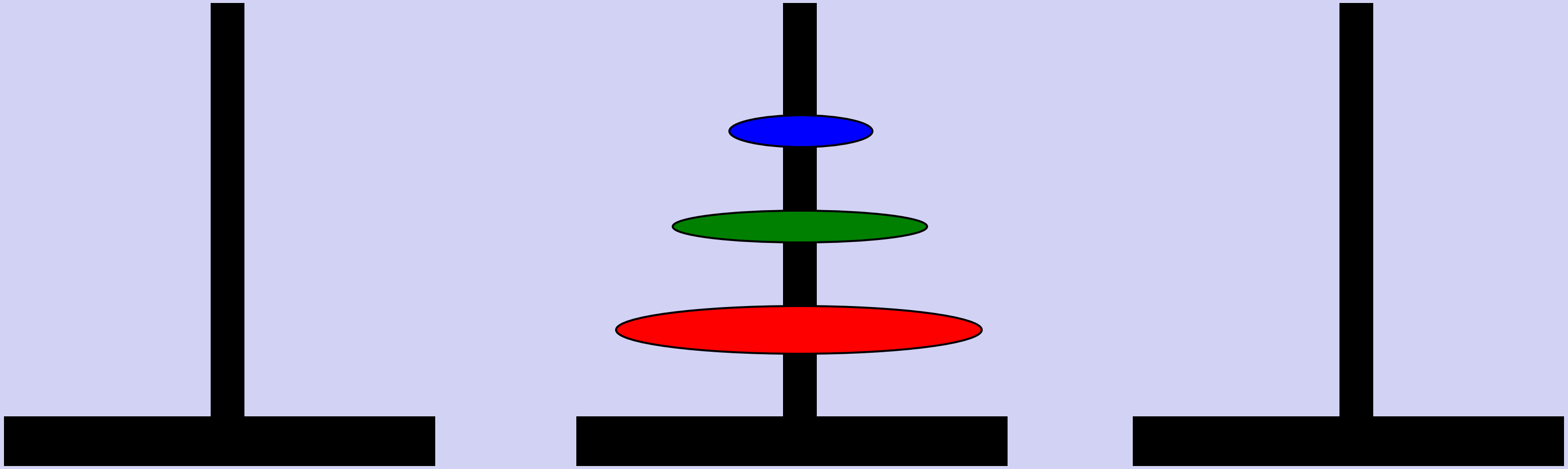
Tower of Hanoi



Tower of Hanoi



Tower of Hanoi



Algorithm

We can combine these steps into the following algorithm:

0. Receive *n, src, dest, aux*.
1. If *n* > 1:

   a. Move(*n-1, src, aux, dest*);

   b. Move(1, *src, dest, aux*);

   c. Move(*n-1, aux, dest, src*);

   Else

   Display "Move the top disk from ", *src*, " to ", *dest*.

   End if.

VIDEO

Coding

```
// ...

void Move(int n, char src, char dest, char aux)
{
  if (n > 1)
  {
    Move(n-1, src, aux, dest);
    Move(1, src, dest, aux);
    Move(n-1, aux, dest, src);
  }
  else
    cout << "Move the top disk from "
         << src << " to " << dest << endl;
}
```

Testing

```
The Hanoi Towers

Enter how many disks: 1
Move the top disk from A to B
```

Testing (*Ct'd*)

```
The Hanoi Towers

Enter how many disks: 2
Move the top disk from A to C
Move the top disk from A to B
Move the top disk from C to B
```

Testing (*Ct'd*)

```
The Hanoi Towers

Enter how many disks: 3
Move the top disk from A to B
Move the top disk from A to C
Move the top disk from B to C
Move the top disk from A to B
Move the top disk from C to A
Move the top disk from C to B
Move the top disk from A to B
```

Testing (*Ct'd*)

```
The Hanoi Towers

Enter how many disks: 4
move a disk from needle A to needle B
move a disk from needle C to needle B
move a disk from needle A to needle C
move a disk from needle B to needle A
move a disk from needle B to needle C
move a disk from needle A to needle C
move a disk from needle A to needle B
move a disk from needle C to needle B
move a disk from needle C to needle A
move a disk from needle B to needle A
move a disk from needle C to needle B
move a disk from needle A to needle C
move a disk from needle A to needle B
move a disk from needle C to needle B
```

Analysis

Let's see how many moves" it takes to solve this problem, as a function of *n*, the number of disks to be moved.

n	Number of disk-moves required
1	1
2	3
3	7
4	15
5	31
…	
i	2^i-1
64	2^{64}-1 (a big number)

Assignment

Prob 1: Write functions to check whether the queue is full

or empty.

Prob 2: Write an algorithm to find ${}^{n}c_{r}$ using recursion.

Prob 3: Write an algorithm to calculate Ackermann function.

Prob 4: Write an algorithm to find first n elements of Fibonacci series.