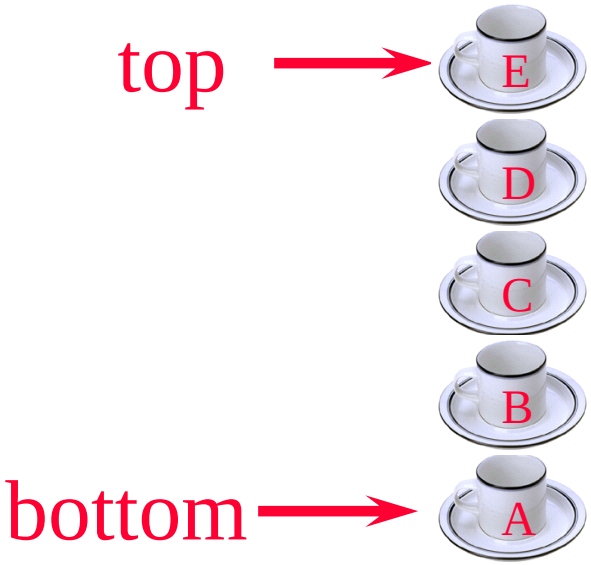
# Chapter Three

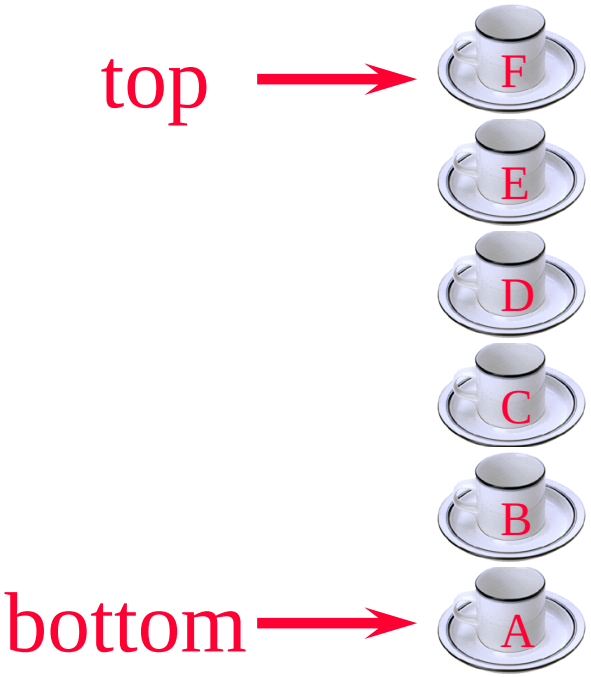
# Stacks & Queues

# Stacks

- Linear list.
- One end is called top.
- Other end is called bottom.
- Additions to and removals from the top end only.

# Stack Of Cups




- Add a cup to the stack.
- Remove a cup from new stack.
- A stack is a LIFO list.

# Parentheses Matching

- (((a+b)*c+d-e)/(f+g)-(h+j)*(k-l))/(m-n)
  - Output pairs (u,v) such that the left parenthesis at position u is matched with the right parenthesis at v.
    - (2,6) (1,13) (15,19) (21,25) (27,31) (0,32) (34,38)
- (a+b))*((c+d)
  - (0,4)
  - right parenthesis at 5 has no matching left parenthesis
  - (8,12)
  - left parenthesis at 7 has no matching right parenthesis

# Parentheses Matching

- scan expression from left to right
- when a left parenthesis is encountered, add its position to the stack
- when a right parenthesis is encountered, remove matching position from stack

# Example

- (((a+b)*c+d-e)/(f+g)-(h+j)*(k-l))/(m-n)

# Example

- (((a+b)*c+d-e)/(f+g)-(h+j)*(k-l))/(m-n)


(2,6) (1,13)

# Example

- (((a+b)*c+d-e)/(f+g)-(h+j)*(k-l))/(m-n)


(2,6) (1,13) (15,19)

# Example

- (((a+b)*c+d-e)/(f+g)-(h+j)*(k-l))/(m-n)


(2,6) (1,13) (15,19) (21,25)

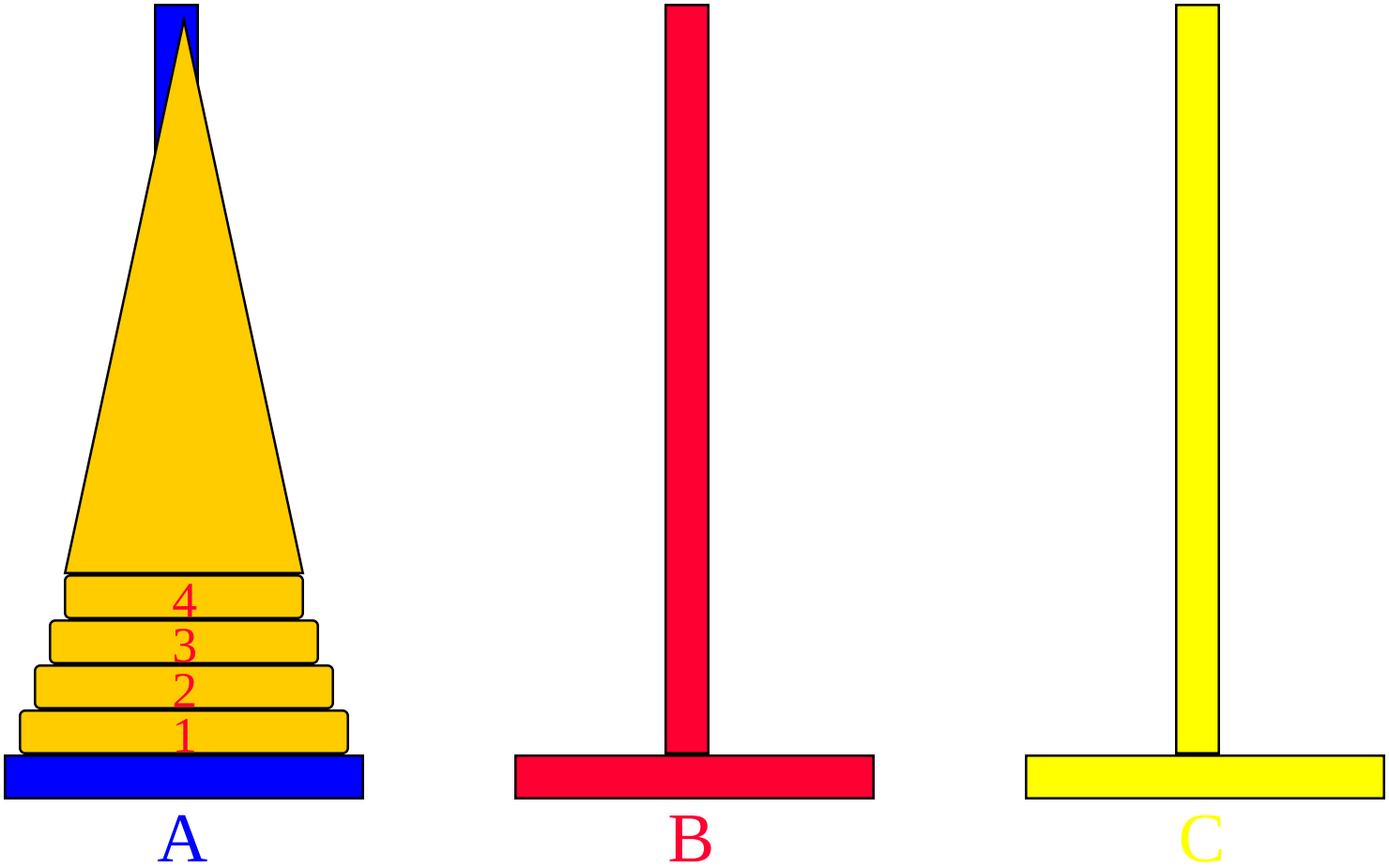
# Example

- (((a+b)*c+d-e)/(f+g)-(h+j)*(k-l))/(m-n)

(2,6) (1,13) (15,19) (21,25)(27,31) (0,32)

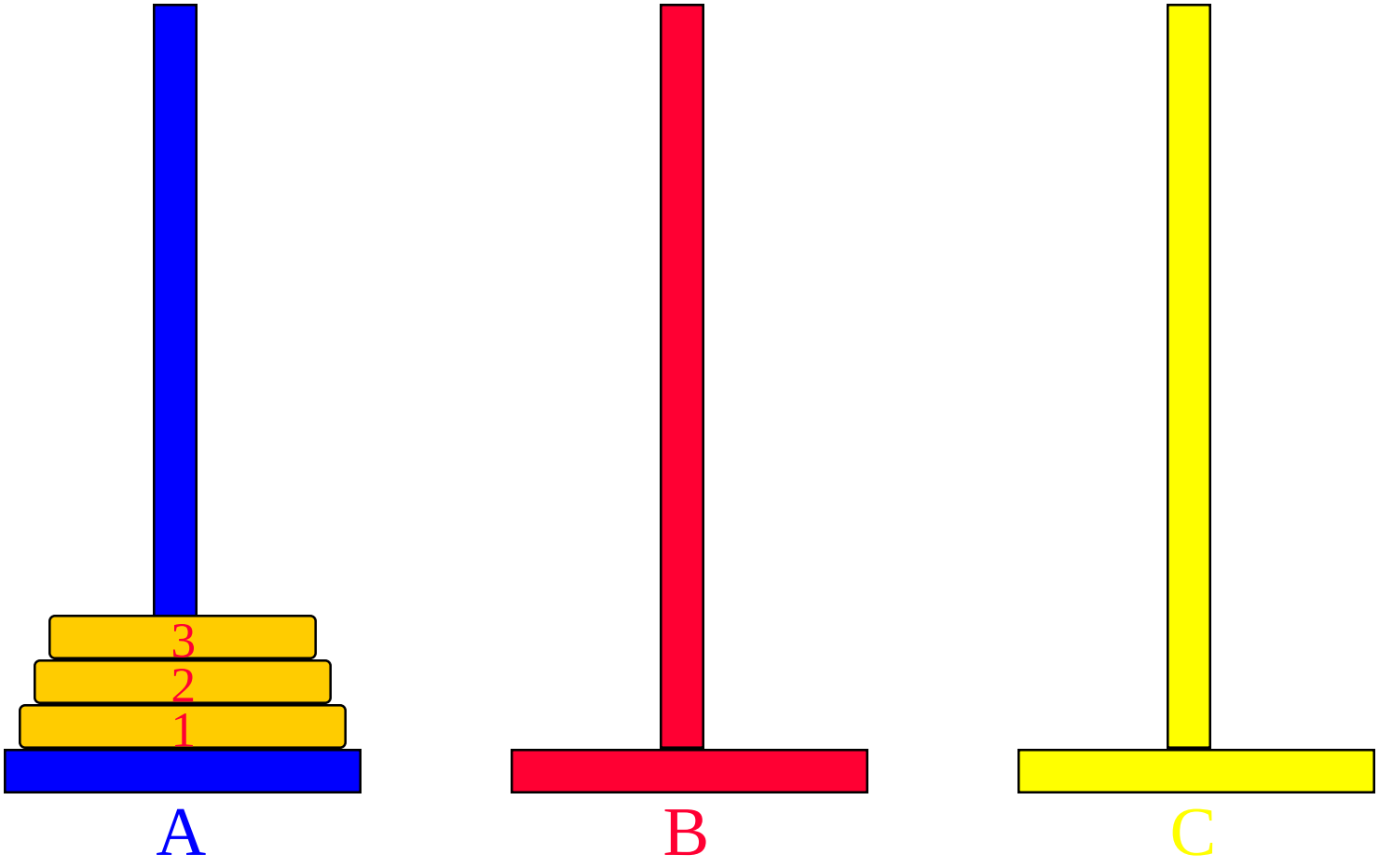
- and so on

# Towers Of Hanoi/Brahma



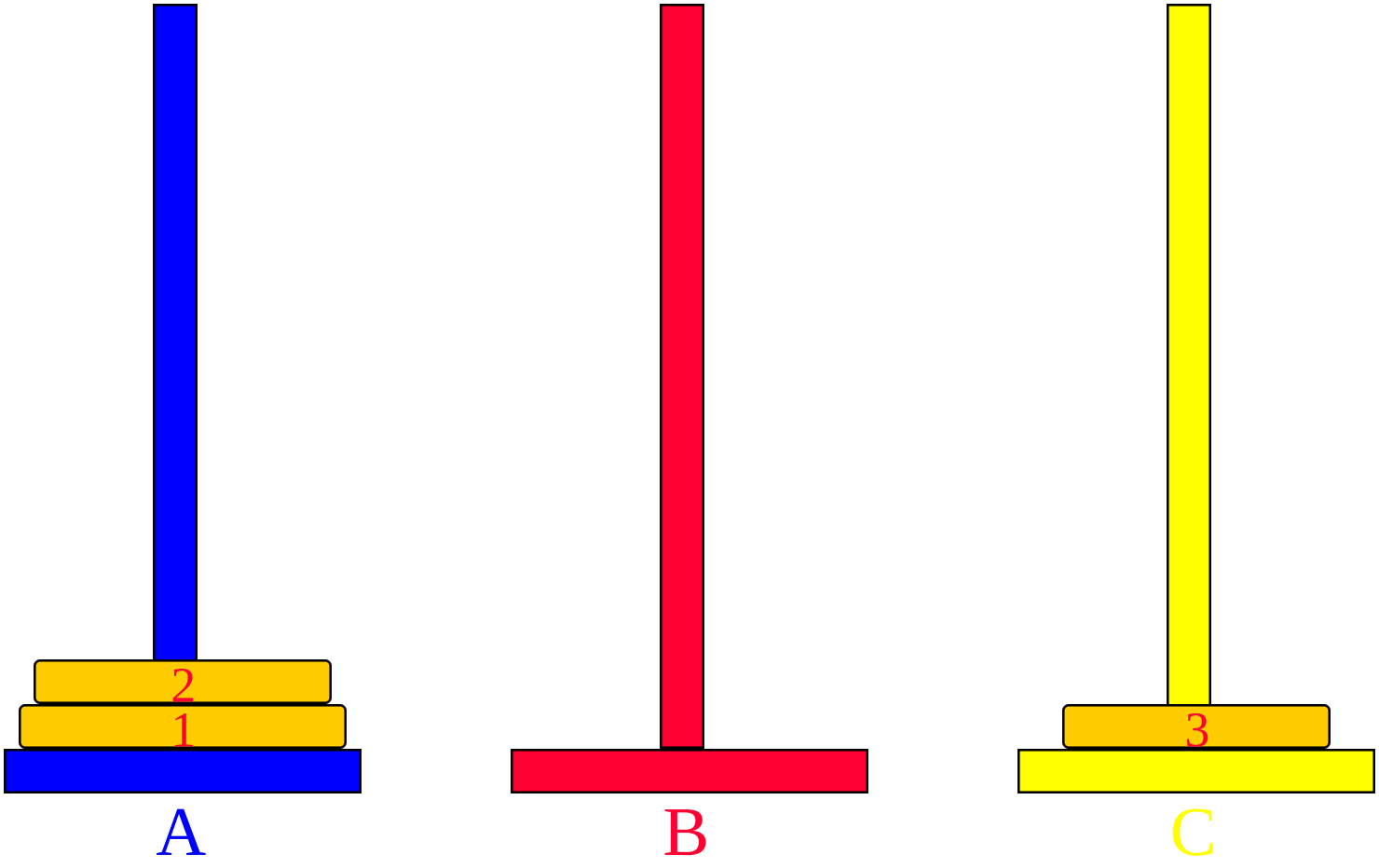

- 64 gold disks to be moved from tower A to tower C
- each tower operates as a stack
- cannot place big disk on top of a smaller one

# Towers Of Hanoi/Brahma



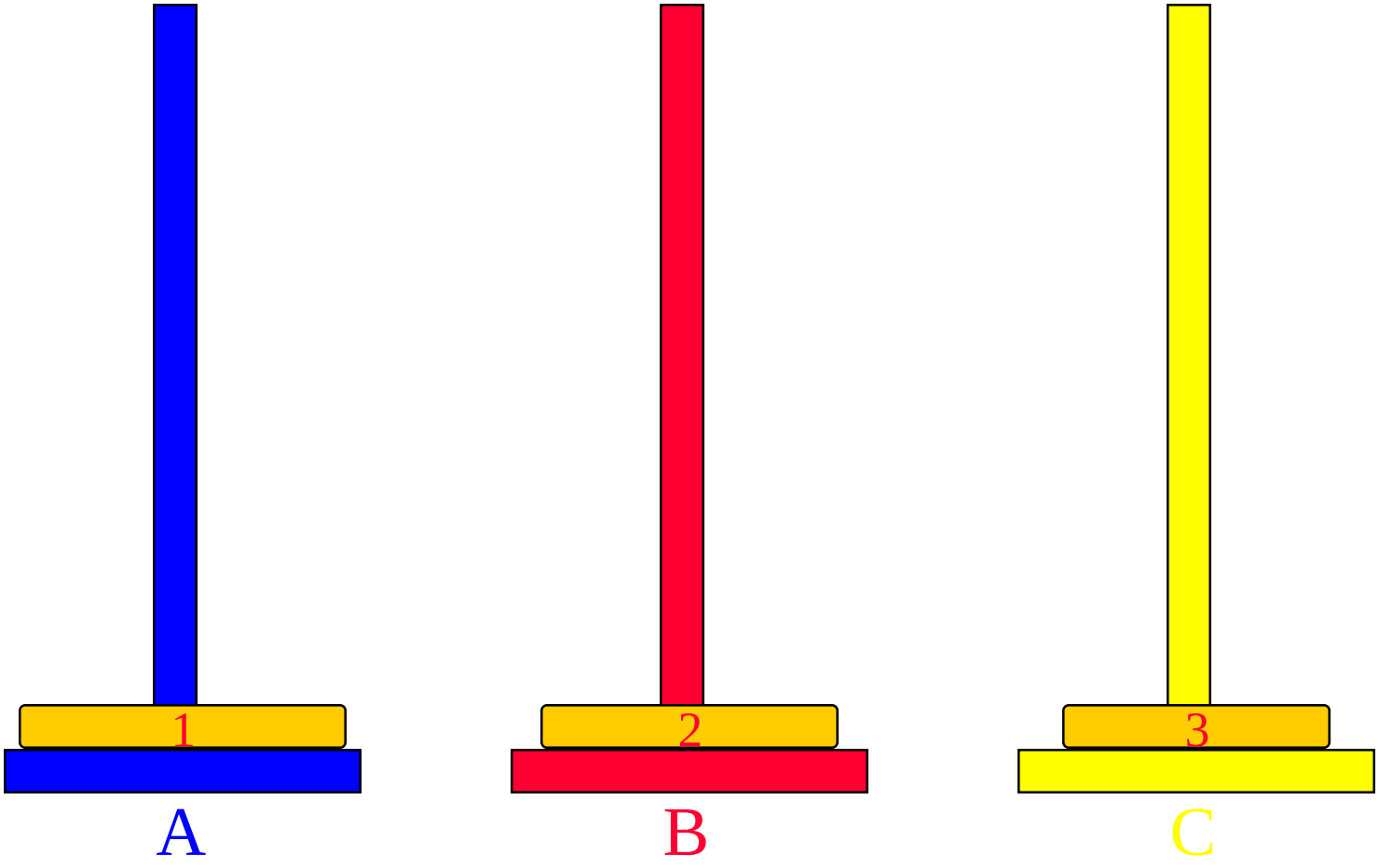

- 3-disk Towers Of Hanoi/Brahma

# Towers Of Hanoi/Brahma



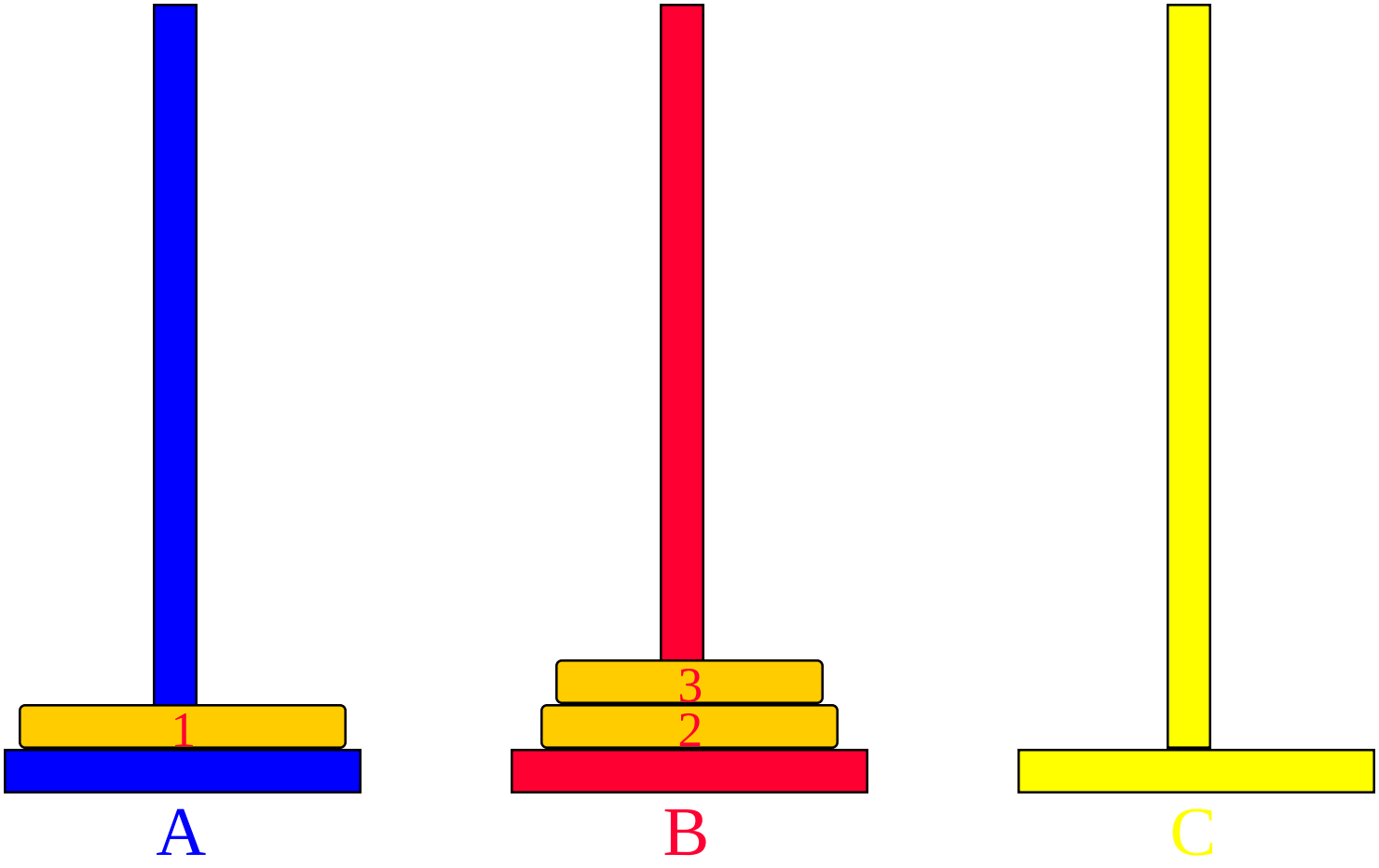

- 3-disk Towers Of Hanoi/Brahma

# Towers Of Hanoi/Brahma



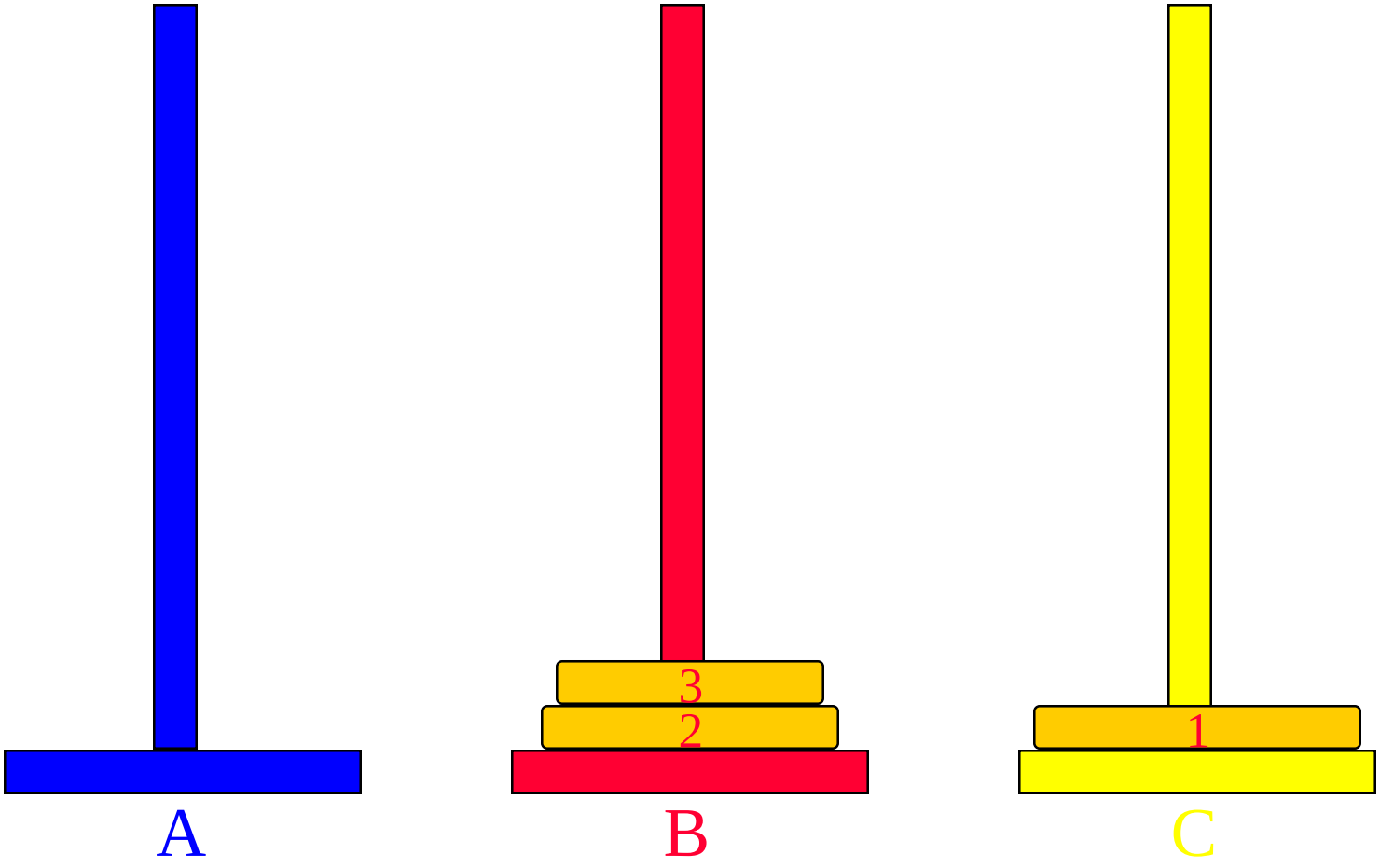

- 3-disk Towers Of Hanoi/Brahma

# Towers Of Hanoi/Brahma



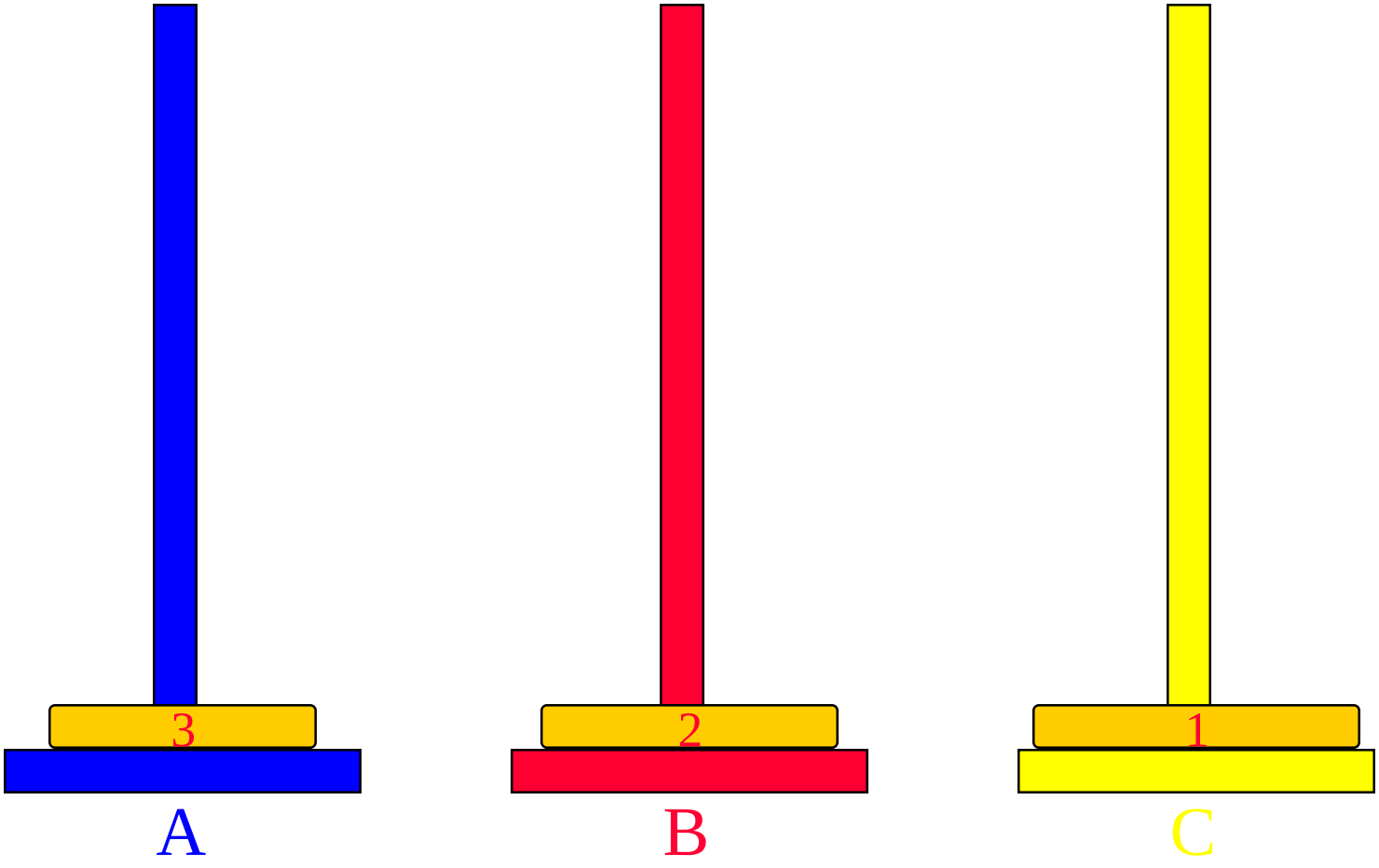

- 3-disk Towers Of Hanoi/Brahma

# Towers Of Hanoi/Brahma



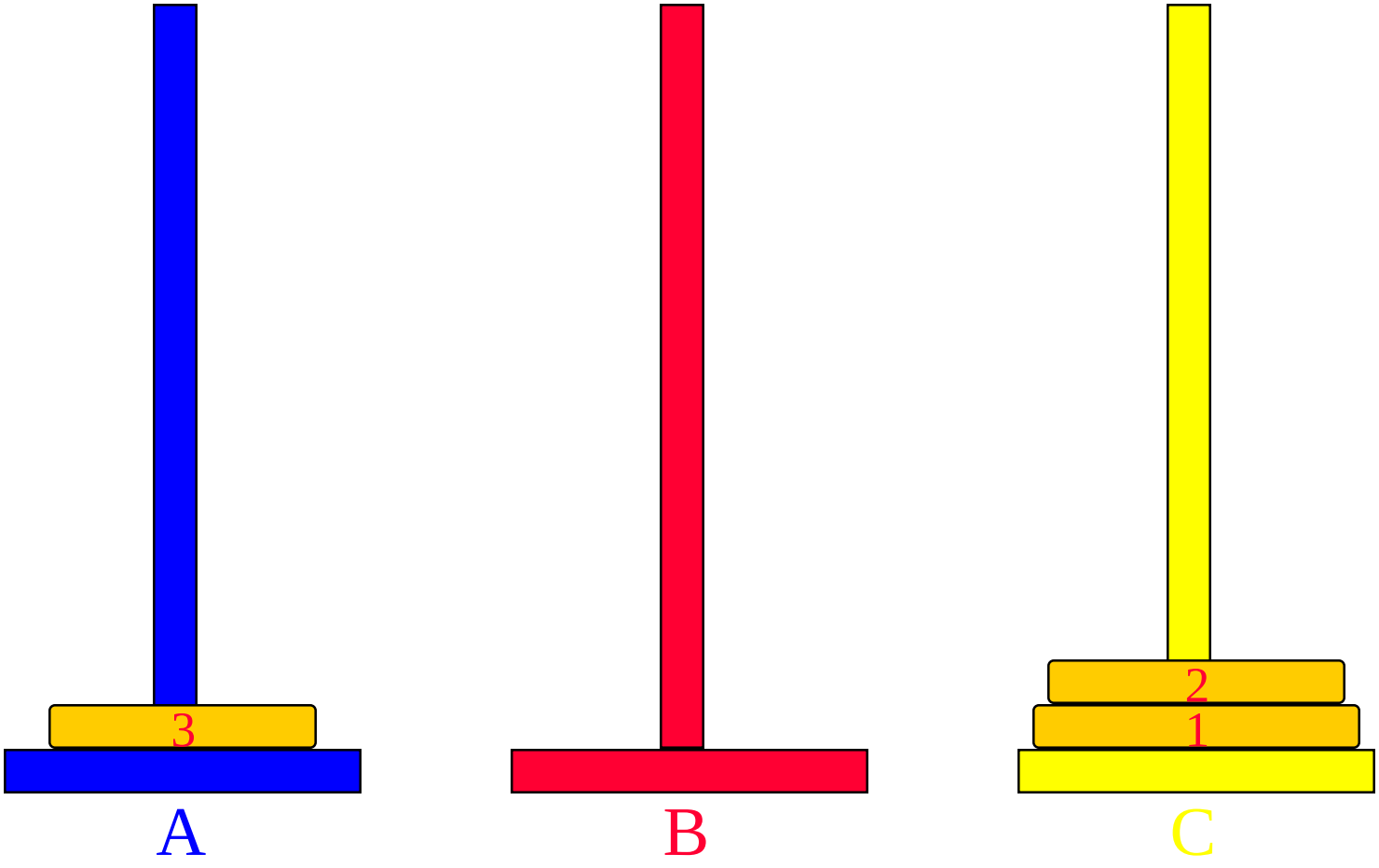

- 3-disk Towers Of Hanoi/Brahma

# Towers Of Hanoi/Brahma



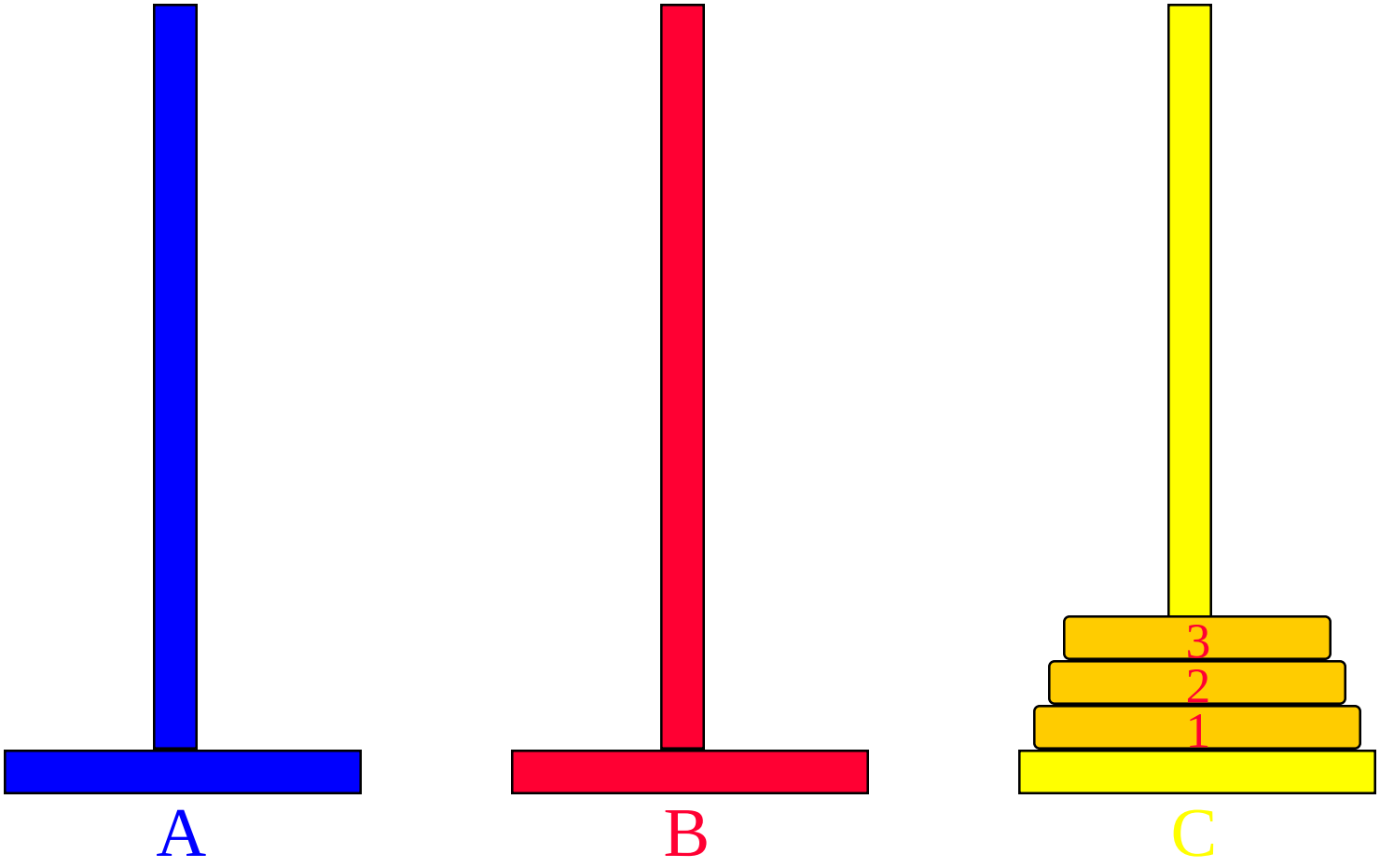

- 3-disk Towers Of Hanoi/Brahma

# Towers Of Hanoi/Brahma



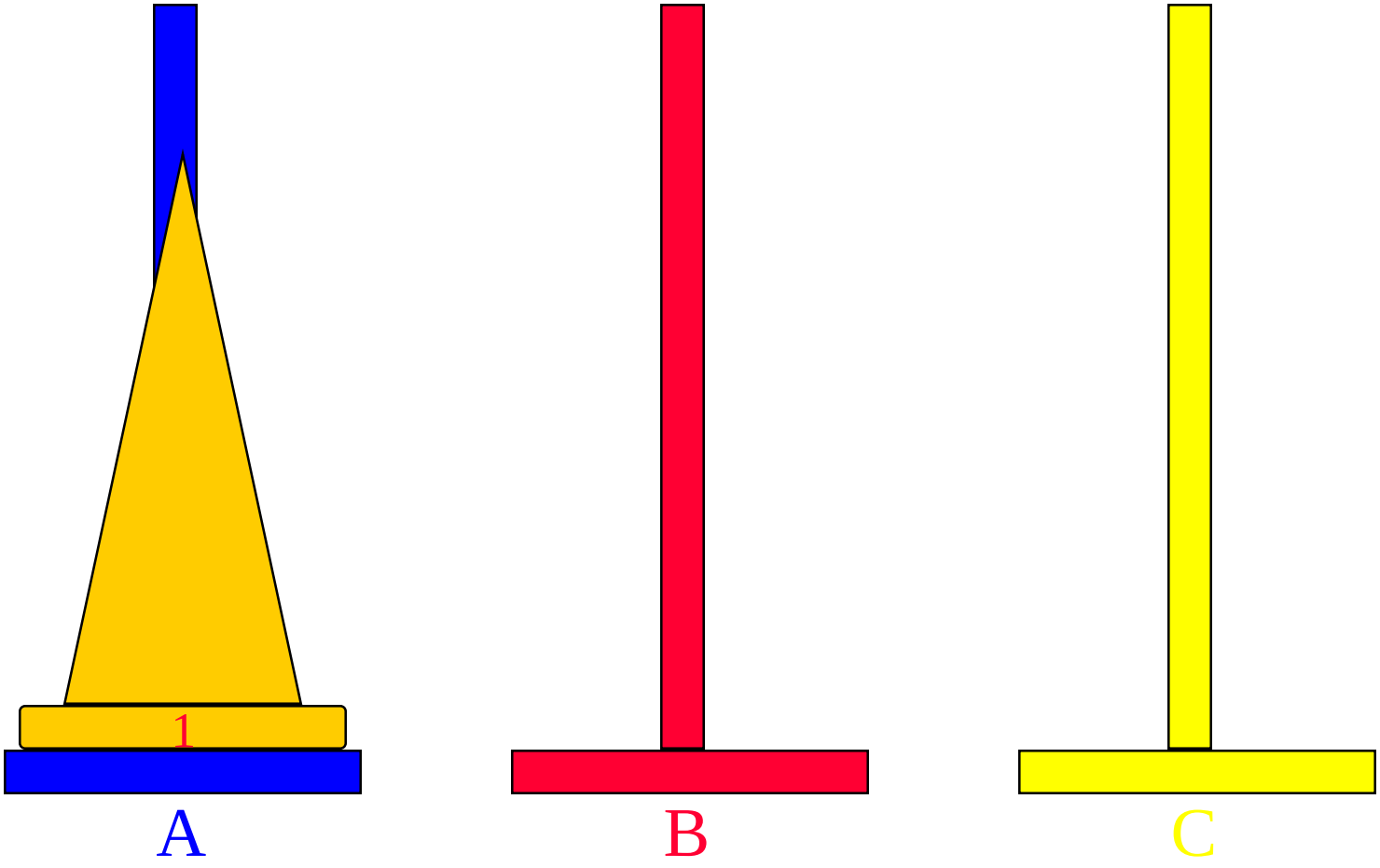

- 3-disk Towers Of Hanoi/Brahma

# Towers Of Hanoi/Brahma



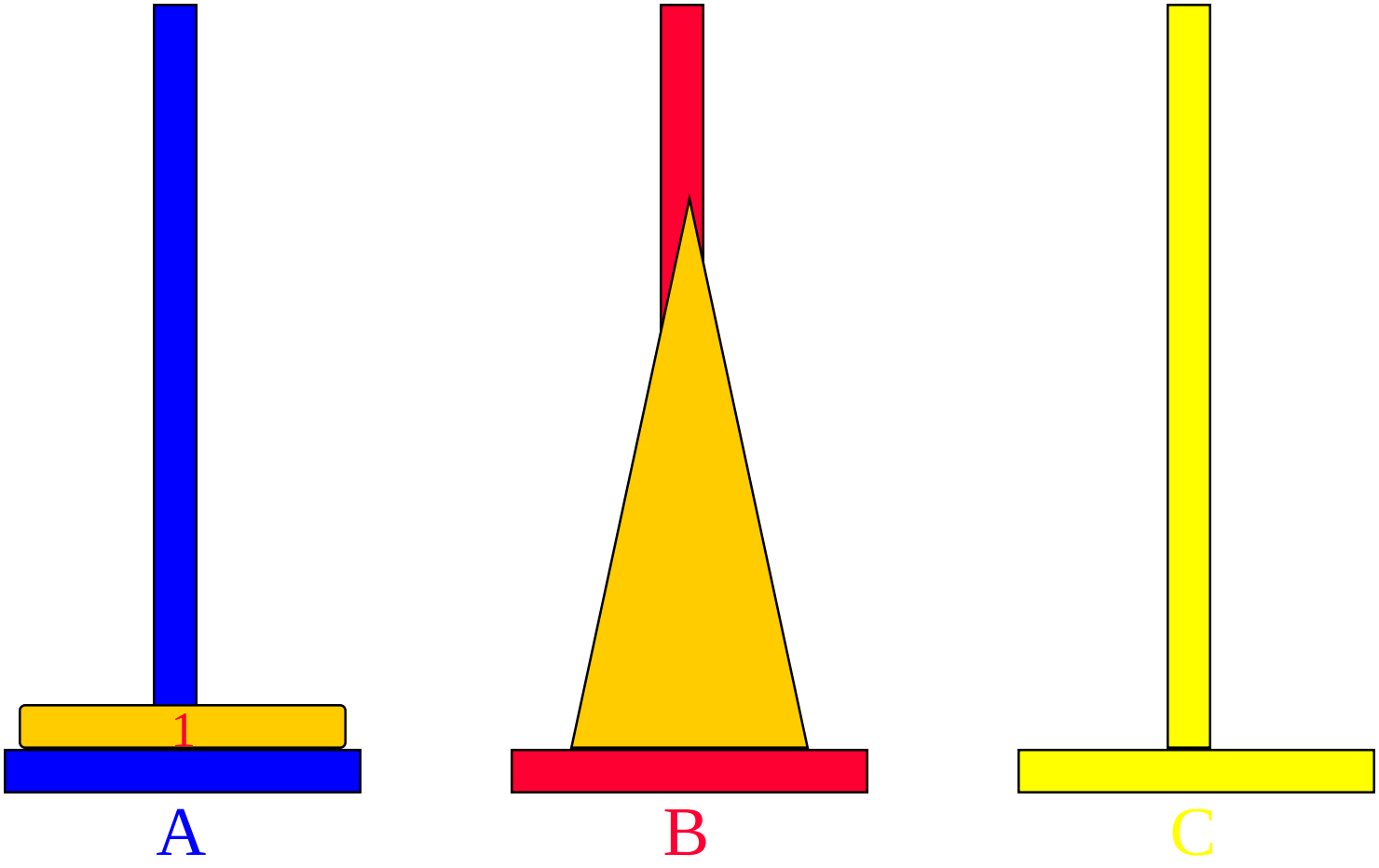

- 3-disk Towers Of Hanoi/Brahma
- 7 disk moves

# Recursive Solution



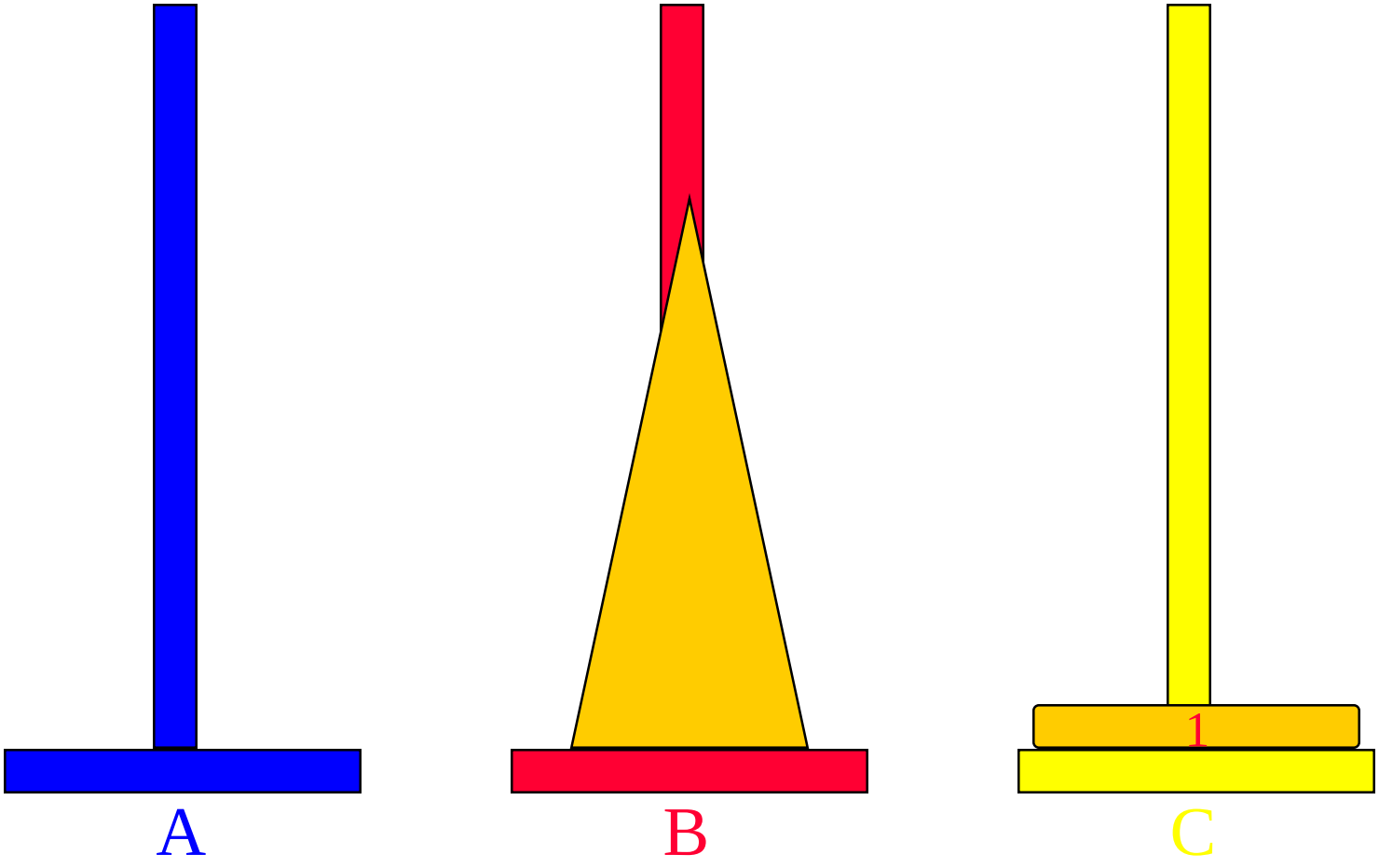

- n > 0 gold disks to be moved from A to C using B
- move top n-1 disks from A to B using C

# Recursive Solution



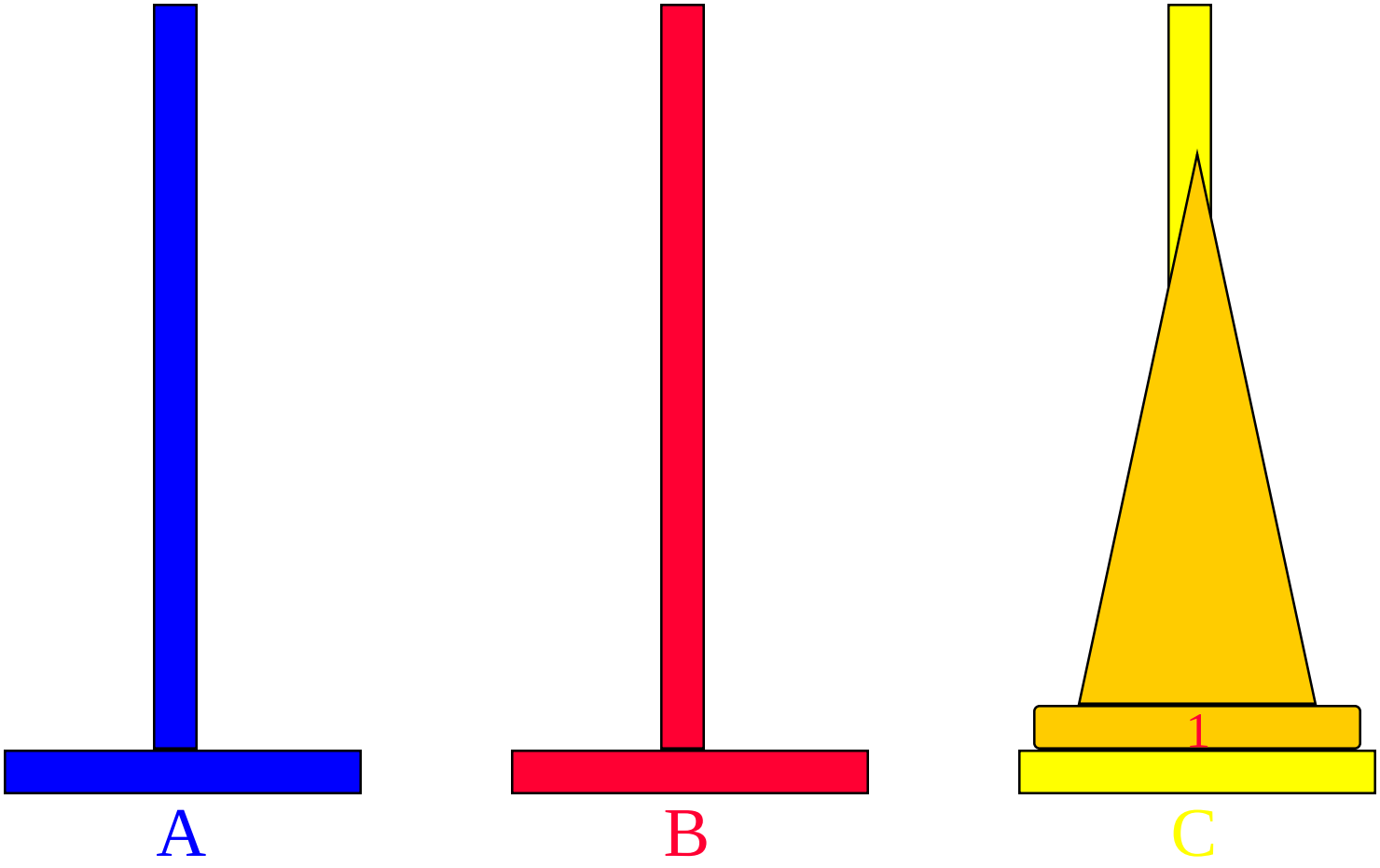

- move top disk from A to C

# Recursive Solution




- move top n-1 disks from B to C using A

# Recursive Solution




- moves(n) = 0 when n = 0
- moves(n) = 2*moves(n-1) + 1 = $2^n$-1 when n > 0

# Towers Of Hanoi/Brahma

- moves(64) = $1.8 * 10^{19}$ (approximately)
- Performing $10^9$ moves/second, a computer would take about 570 years to complete.
- At 1 disk move/min, the monks will take about $3.4 * 10^{13}$ years.
- (* TH(i,j, A,B,C); *)
- (* begin if i>j {TH(i, j+1, A,C,B); *)
- (* println(j); *)
- (* TH(i, j+1, B,A,C) ; *)
- (* else println(i); end; *)

# Method Invocation And Return

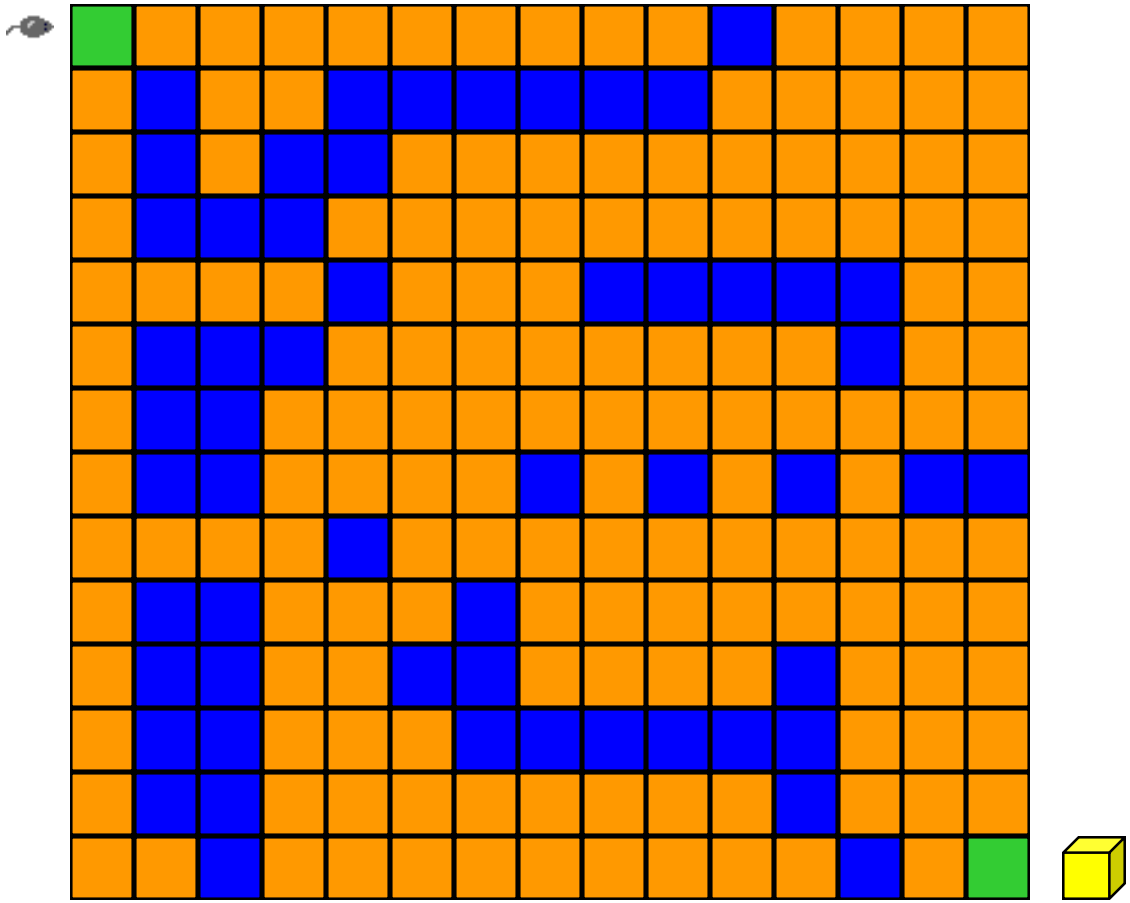
```
public void a()
{ …; b(); …}
public void b()
{ …; c(); …}
public void c()
{ …; d(); …}
public void d()
{ …; e(); …}
public void e()
{ …; c(); …}
```

return address in d()
return address in c()
return address in e()
return address in d()
return address in c()
return address in b()
return address in a()

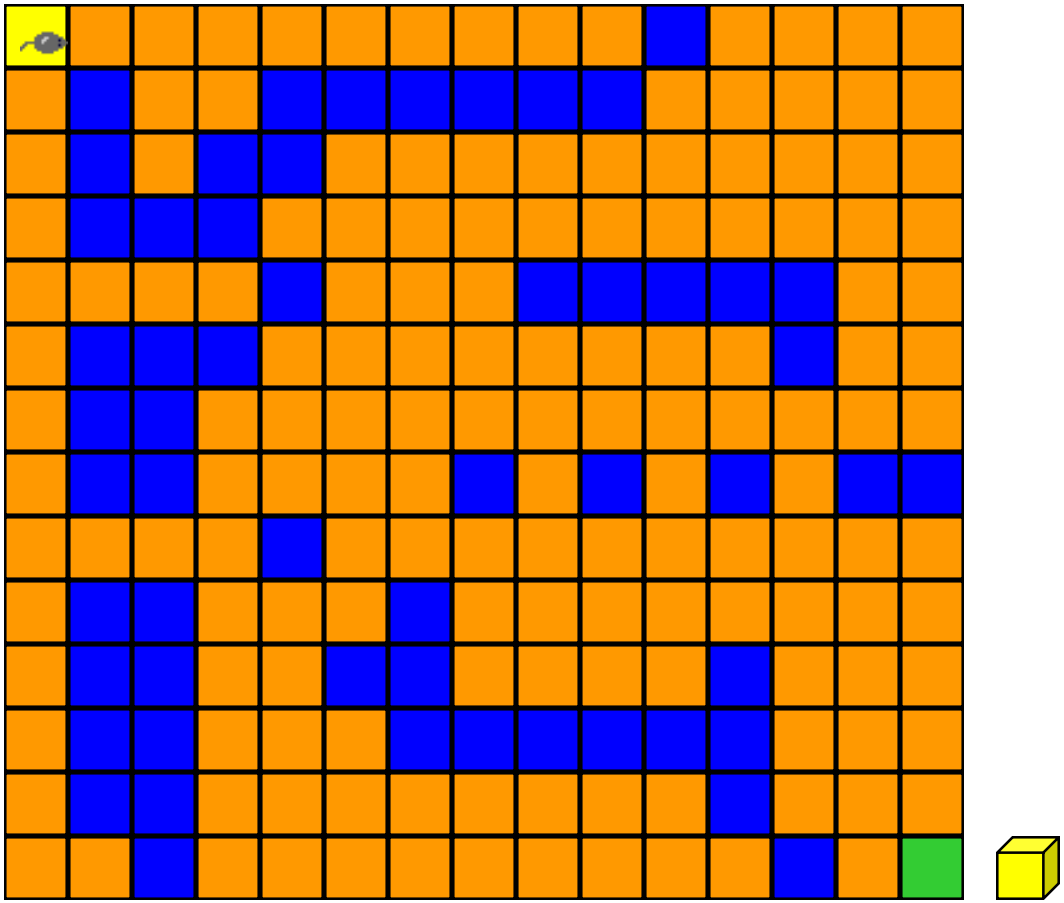
# Try-Throw-Catch

- When you enter a try block, push the address of this block on a stack.
- When an exception is thrown, pop the try block that is at the top of the stack (if the stack is empty, terminate).
- If the popped try block has no matching catch block, go back to the preceding step.
- If the popped try block has a matching catch block, execute the matching catch block.

# Rat In A Maze

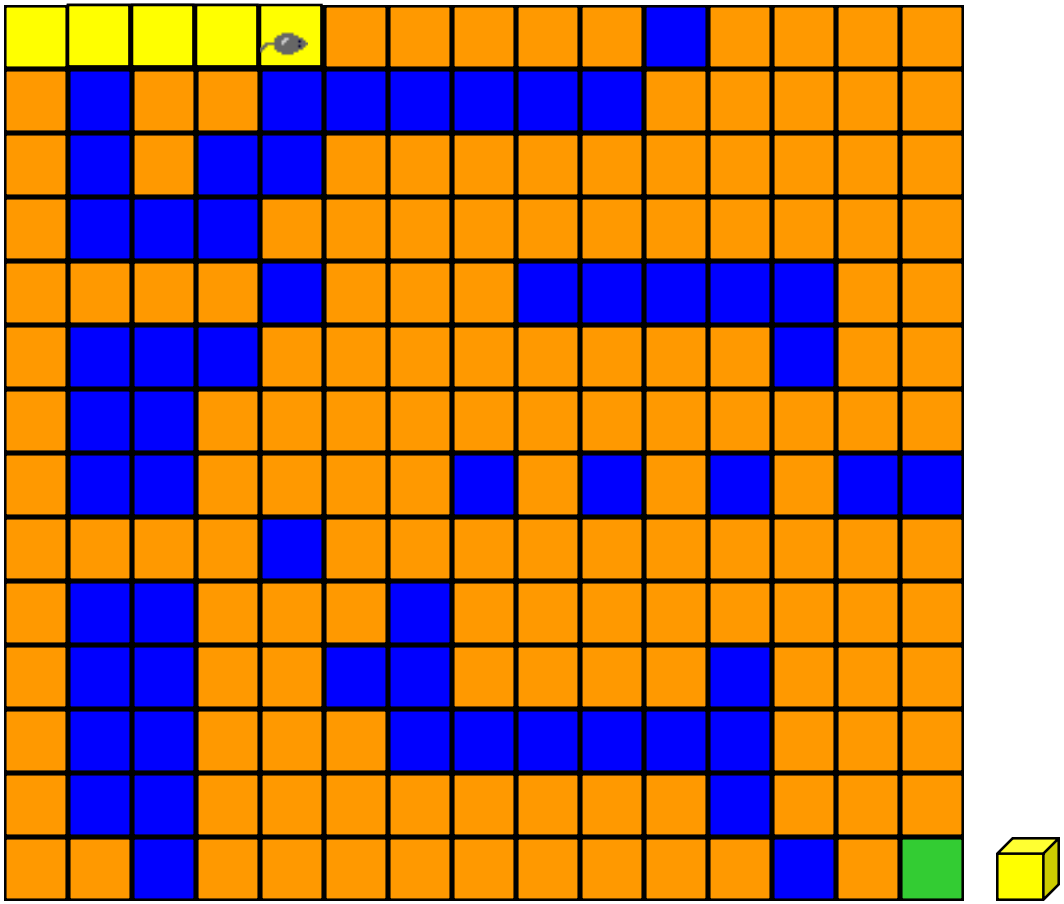


# Rat In A Maze



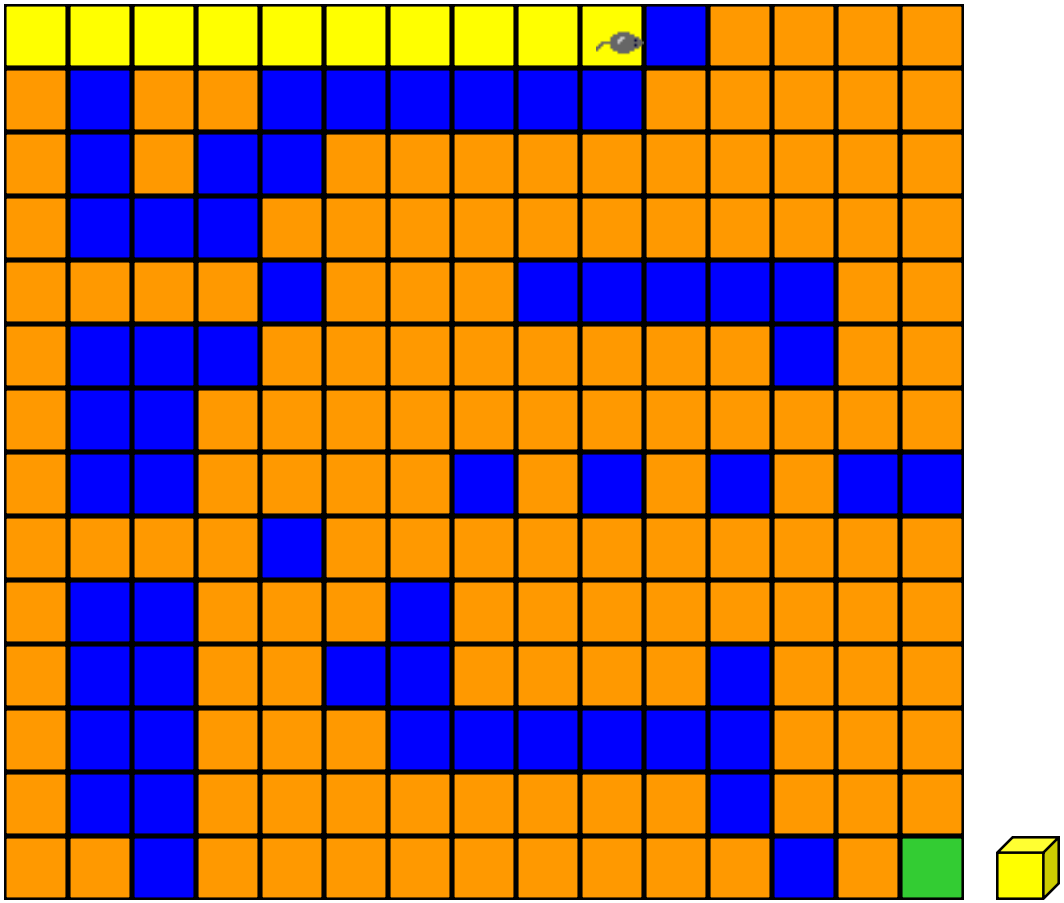
- Move order is: right, down, left, up
- Block positions to avoid revisit.

# Rat In A Maze



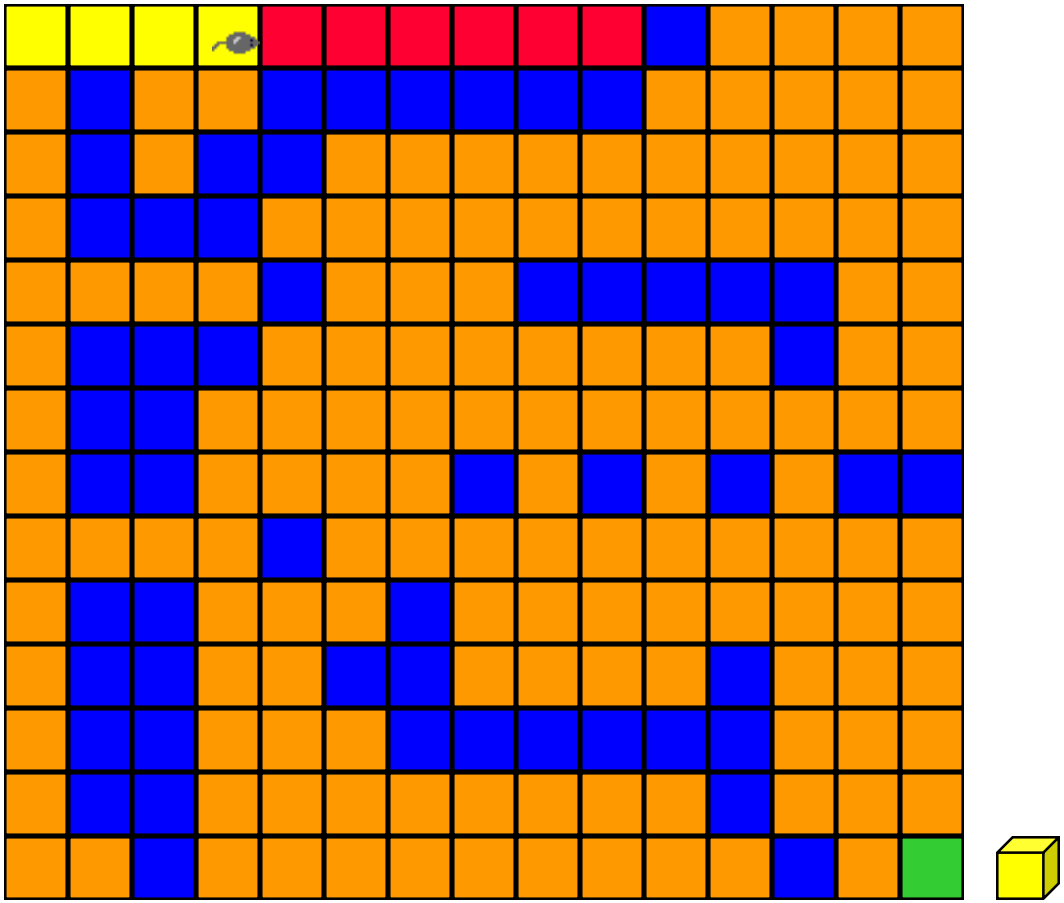
- Move order is: right, down, left, up
- Block positions to avoid revisit.

# Rat In A Maze



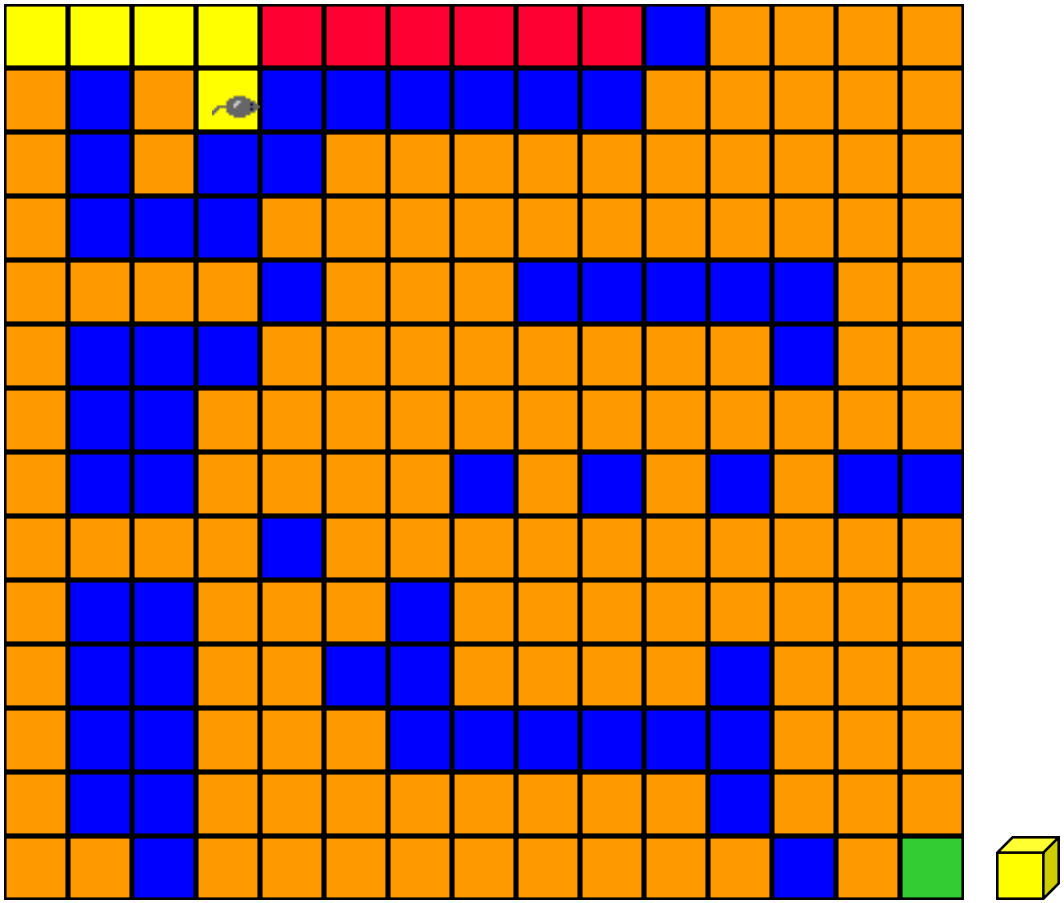
- Move backward until we reach a square from which a forward move is possible.

# Rat In A Maze



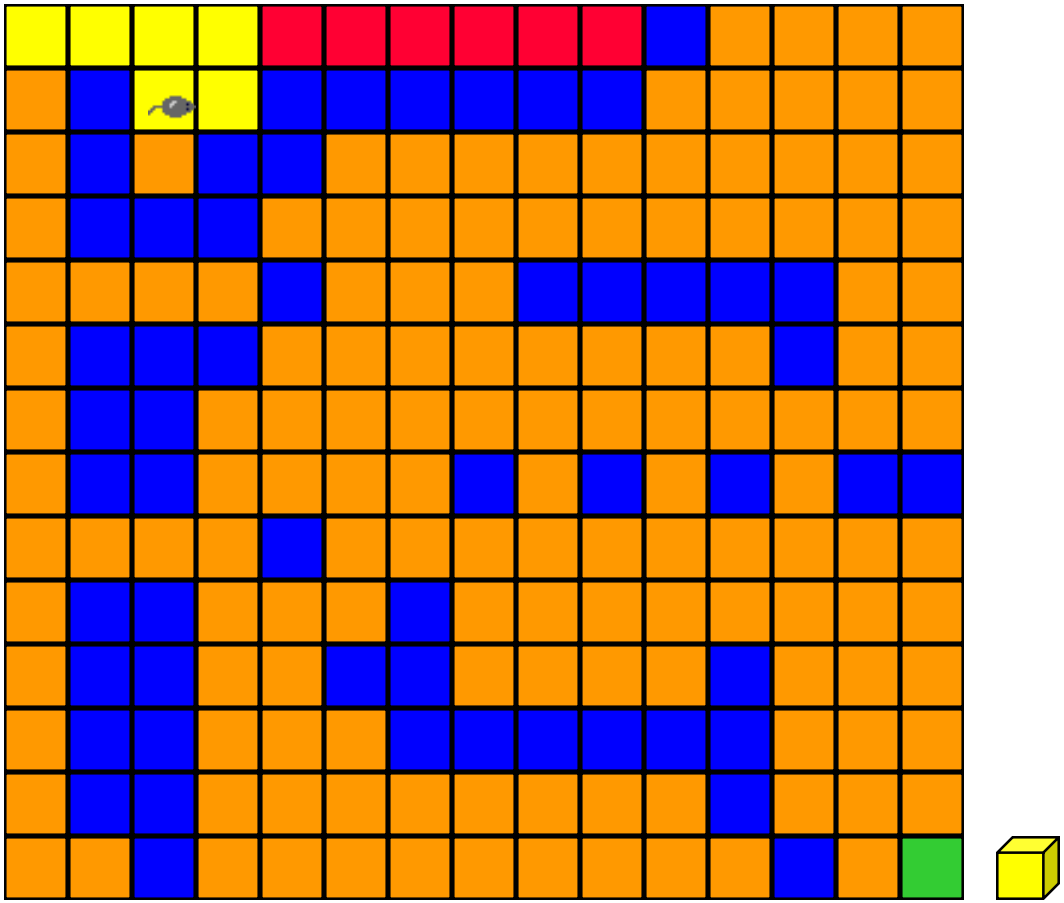
- Move down.

# Rat In A Maze



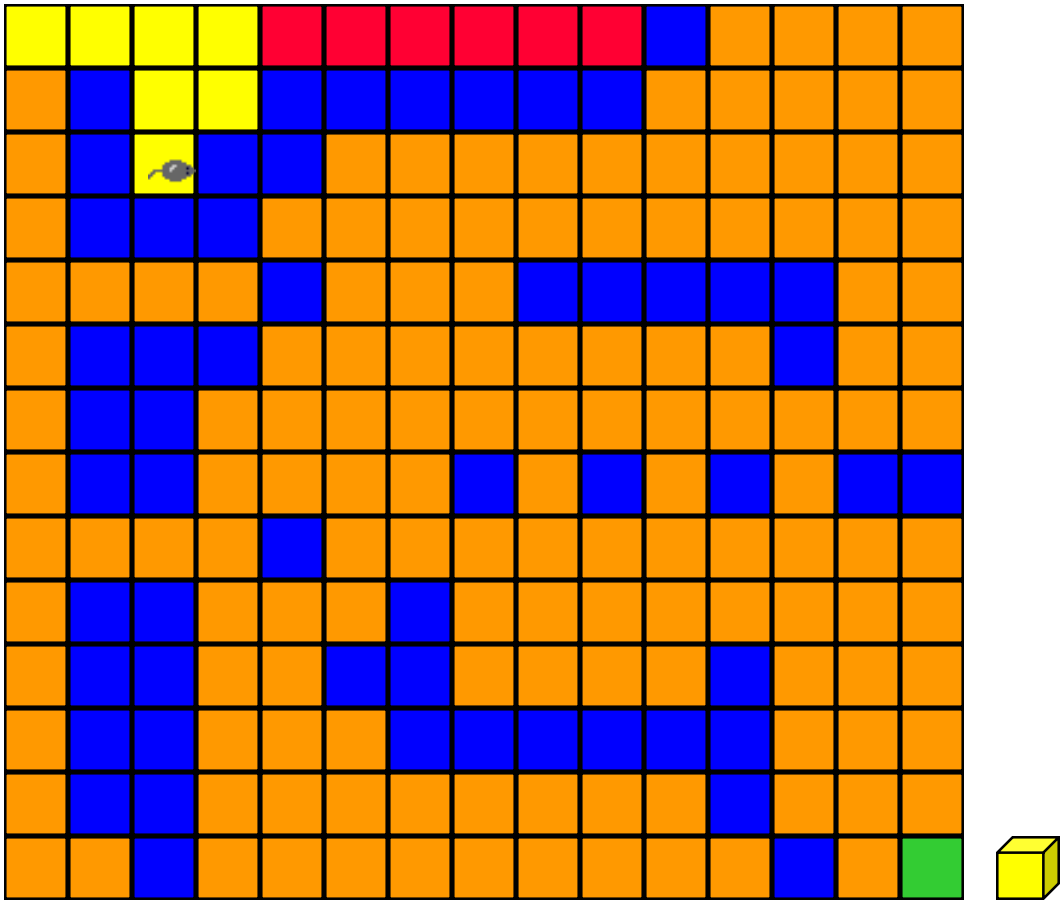
- Move left.

# Rat In A Maze



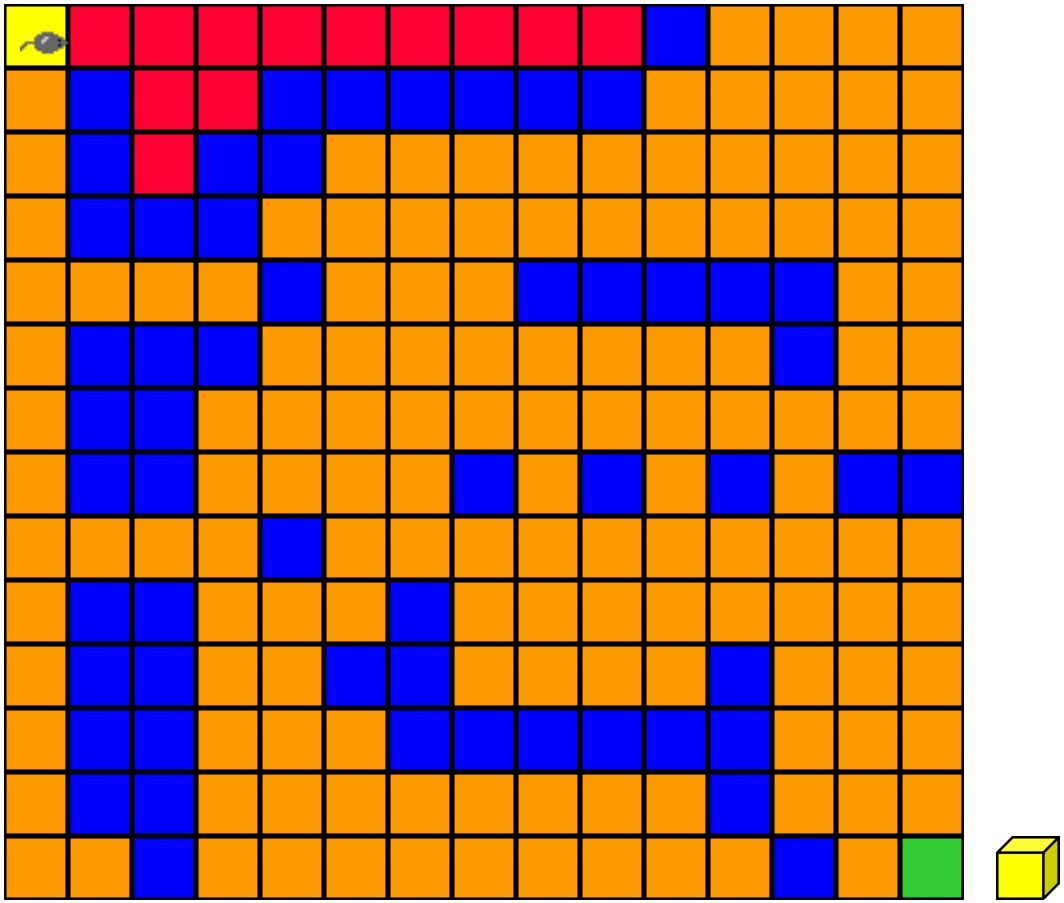
- Move down.

# Rat In A Maze



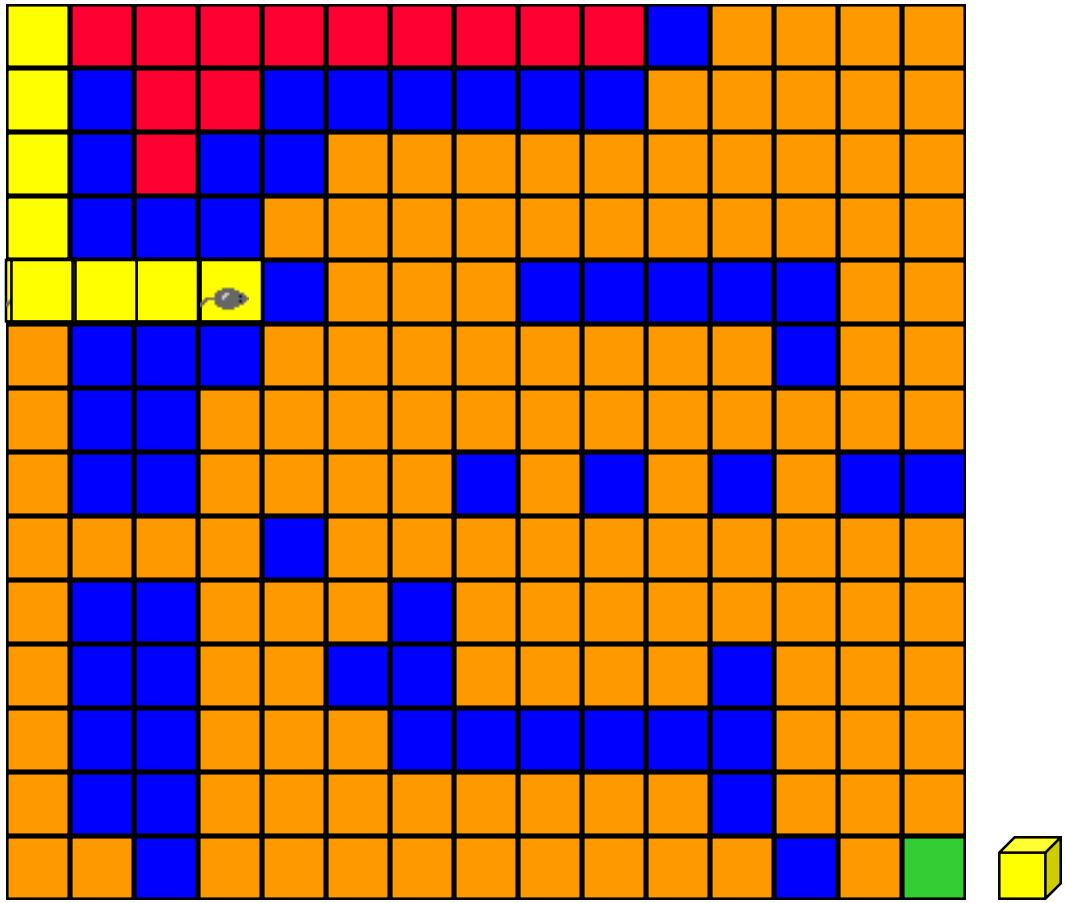
- Move backward until we reach a square from which a forward move is possible.

# Rat In A Maze



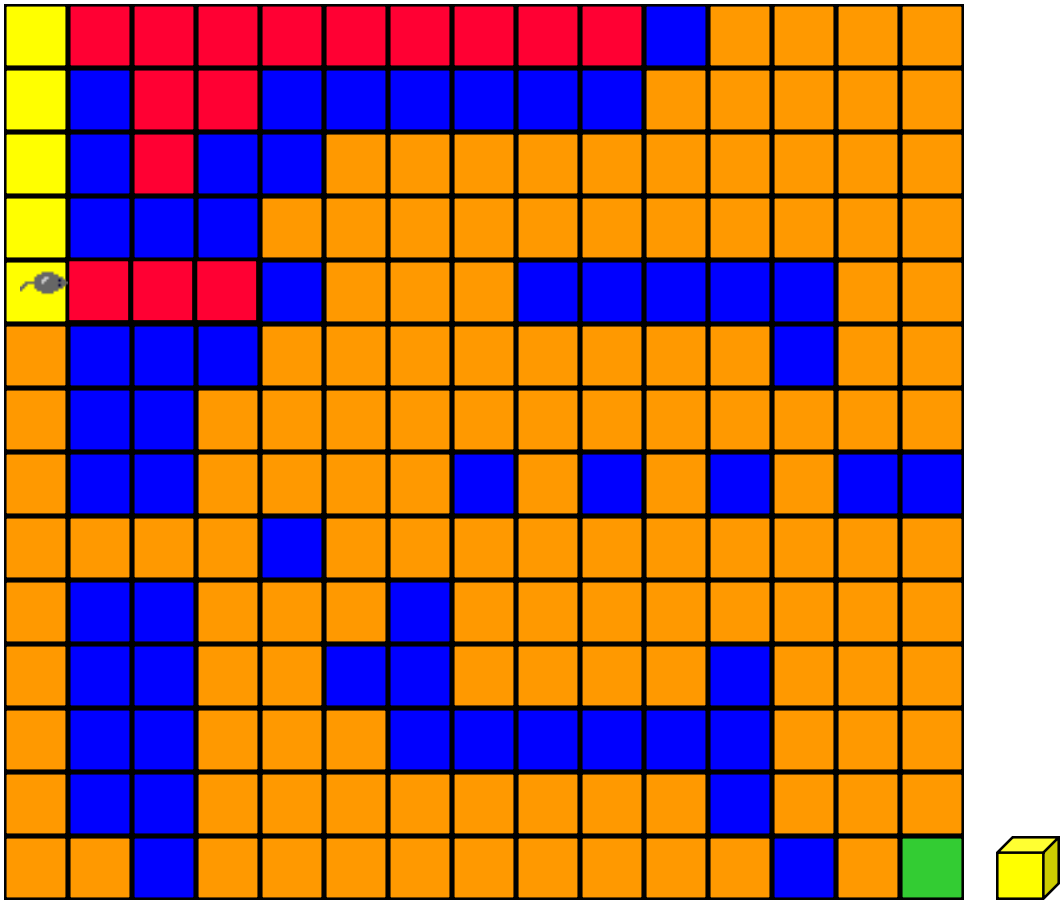
- Move backward until we reach a square from which a forward move is possible.
- Move downward.

# Rat In A Maze



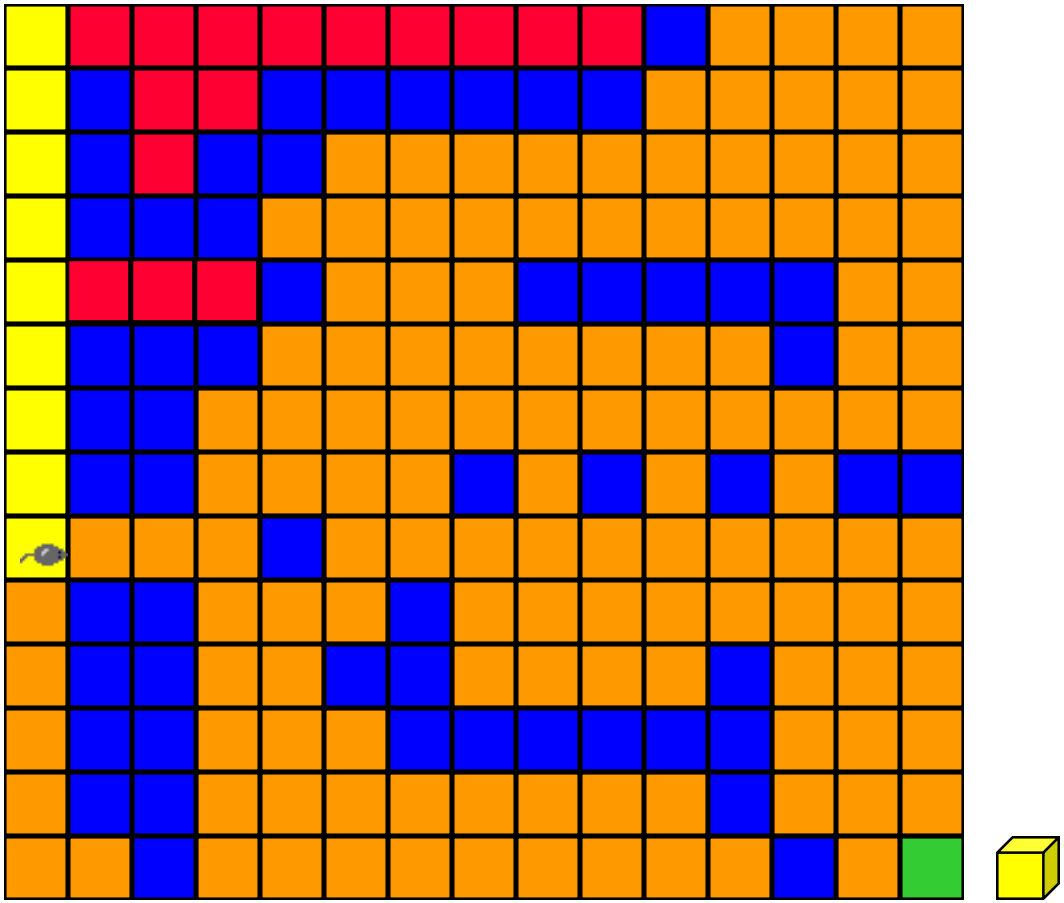
- Move right.
- Backtrack.

# Rat In A Maze



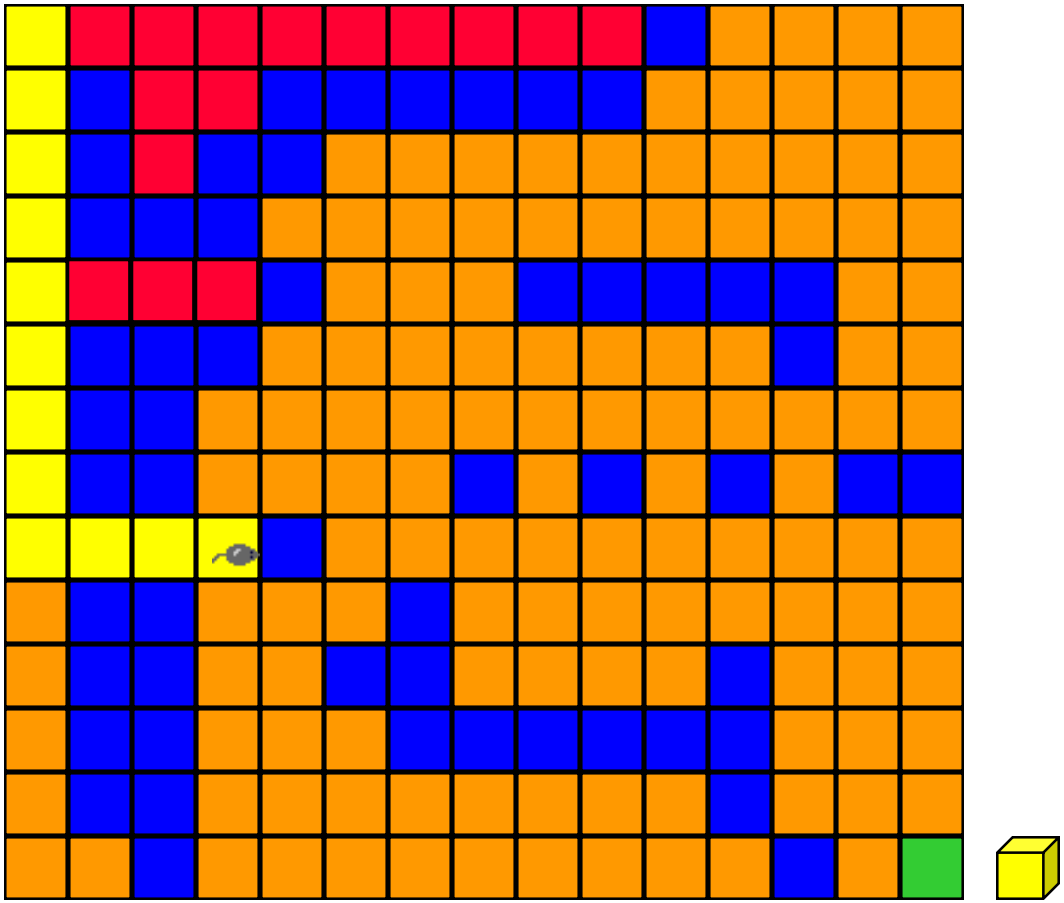
- Move downward.

# Rat In A Maze



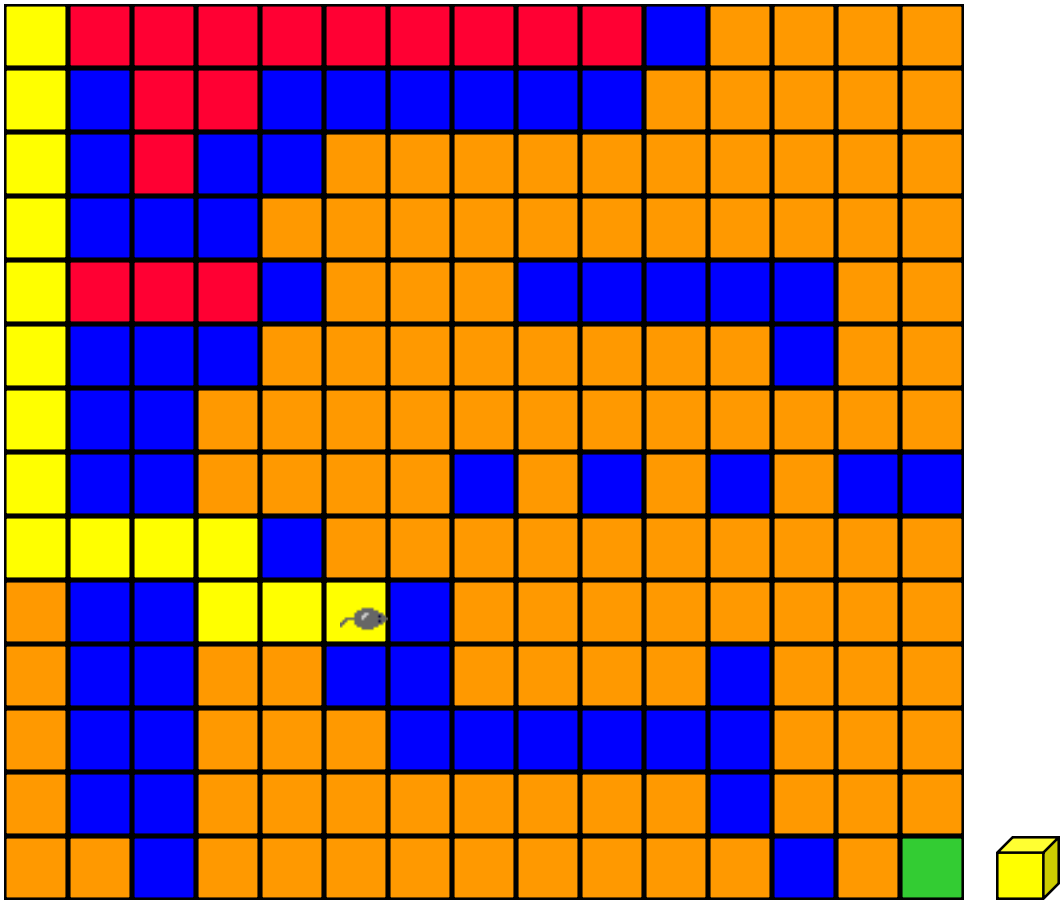
- Move right.

# Rat In A Maze



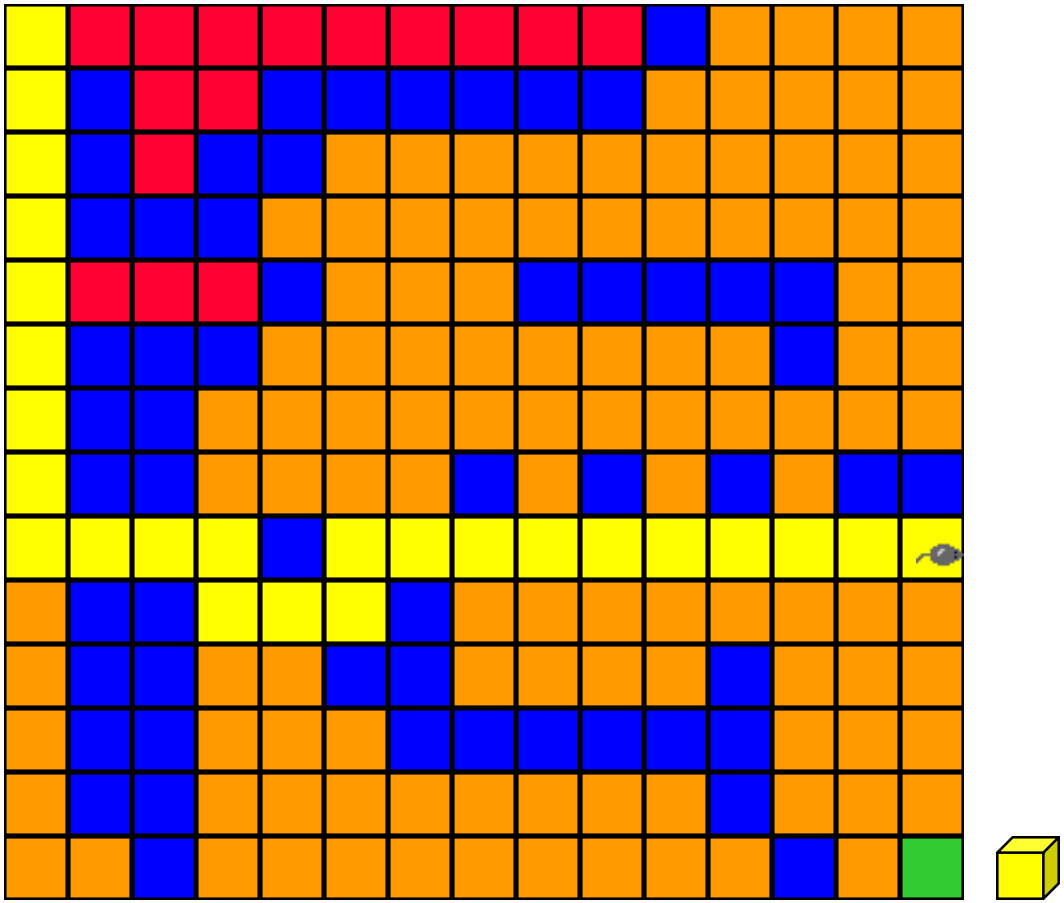
- Move one down and then right.

# Rat In A Maze



- Move one up and then right.

# Rat In A Maze



- Move down to exit and eat cheese.
- Path from maze entry to current position operates as a stack.

# Stacks

- Standard operations:
  - IsEmpty … return true iff stack is empty
  - Top … return top element of stack
  - Push … add an element to the top of the stack
  - Pop … delete the top element of the stack

# Stacks

- Use a 1D array to represent a stack.
- Stack elements are stored in stack[0] through stack[top].

# Stacks

| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| a | b | c | d | e | | | | | | | | | | |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | | | | | | | | |

- stack top is at element e
- IsEmpty() => check whether top >= 0
  - O(1) time
- Top() => If not empty return stack[top]
  - O(1) time

# Derive From arrayList

| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| a | b | c | d | e | | | | | | | | | | |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | | | | | | | | |

- Push(theElement) => if array full (top == capacity – 1) increase capacity and then add at stack[top+1]
- O(capacity) time when full; otherwise O(1)
- pop() => if not empty, delete from stack[top]
- O(1) time

# The Class Stack

```
template<class T>
class Stack
{
   public:
    Stack(int stackCapacity = 10);
    ~Stack() {delete [] stack;}
    bool IsEmpty() const;
    T& Top() const;
    void Push(const T& item);
    void Pop();
   private:
    T *stack;      // array for stack elements
    int top;       // position of top element
    int capacity; // capacity of stack array
};
```

# Constructor

```
template<class T>
Stack<T>::Stack(int stackCapacity)
                :capacity(stackCapacity)
{
   if (capacity < 1)
     throw "Stack capacity must be > 0";
   stack = new T[capacity];
   top = -1;
}
```

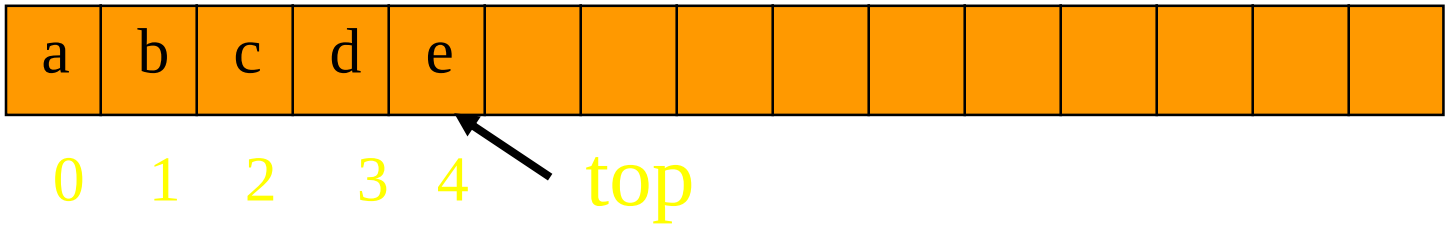
# IsEmpty

```
template<class T>
inline bool Stack<T>::IsEmpty() const
   {return top == -1}
```

# Top

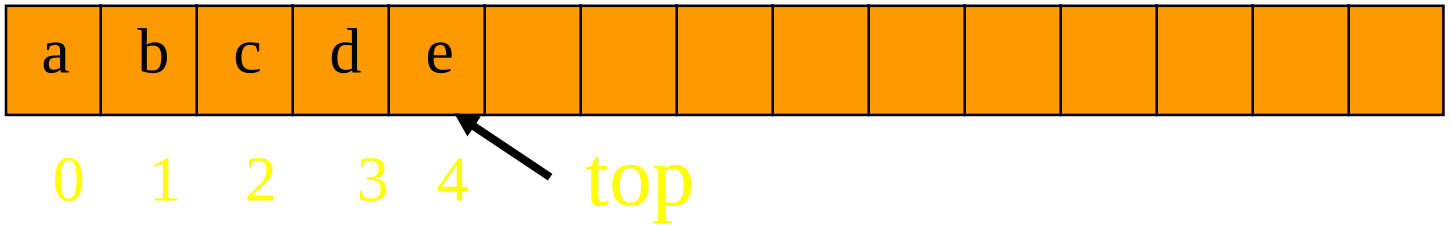
```
template<class T>
inline T& Stack<T>::Top() const
{
   if (IsEmpty())
       throw "Stack is empty";
   return stack[top];
}
```

# Push




```
template<class T>
void Stack<T>::Push(const T& x)
{// Add x to the stack.
   if (top == capacity - 1)
      {ChangeSize1D(stack, capacity,
                                 2*capacity);
       capacity *= 2;
      }
   // add at stack top
   stack[++top] = x;
}
```

# Pop




```
void Stack<T>::Pop()
{
    if (IsEmpty())
        throw "Stack is empty. Cannot delete.";
    stack[top--].~T(); // destructor for T
}
```

# Queues

- Linear list.
- One end is called front.
- Other end is called rear.
- Additions are done at the rear only.
- Removals are made from the front only.

# Bus Stop Queue

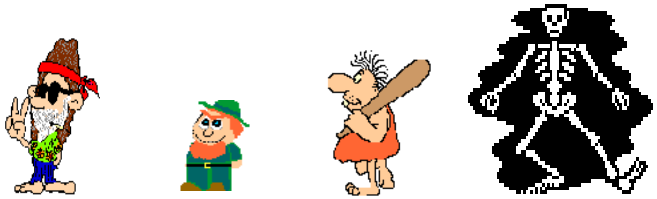

front rear

# Bus Stop Queue

# Bus Stop Queue


front rear

# Bus Stop Queue

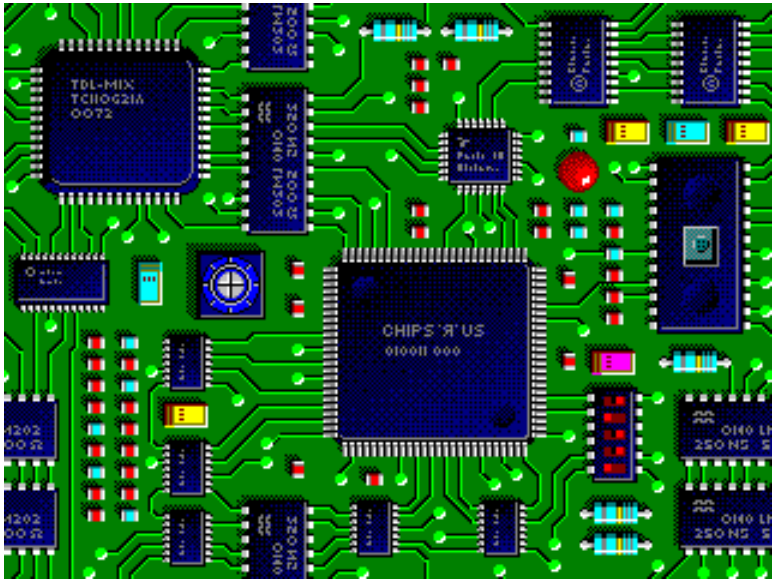



front rear

# Revisit Of Stack Applications

- Applications in which the stack cannot be replaced with a queue.
  - Parentheses matching.
  - Towers of Hanoi.
  - Switchbox routing.
  - Method invocation and return.
  - Try-catch-throw implementation.
- Application in which the stack may be replaced with a queue.
  - Rat in a maze.
    - Results in finding shortest path to exit.

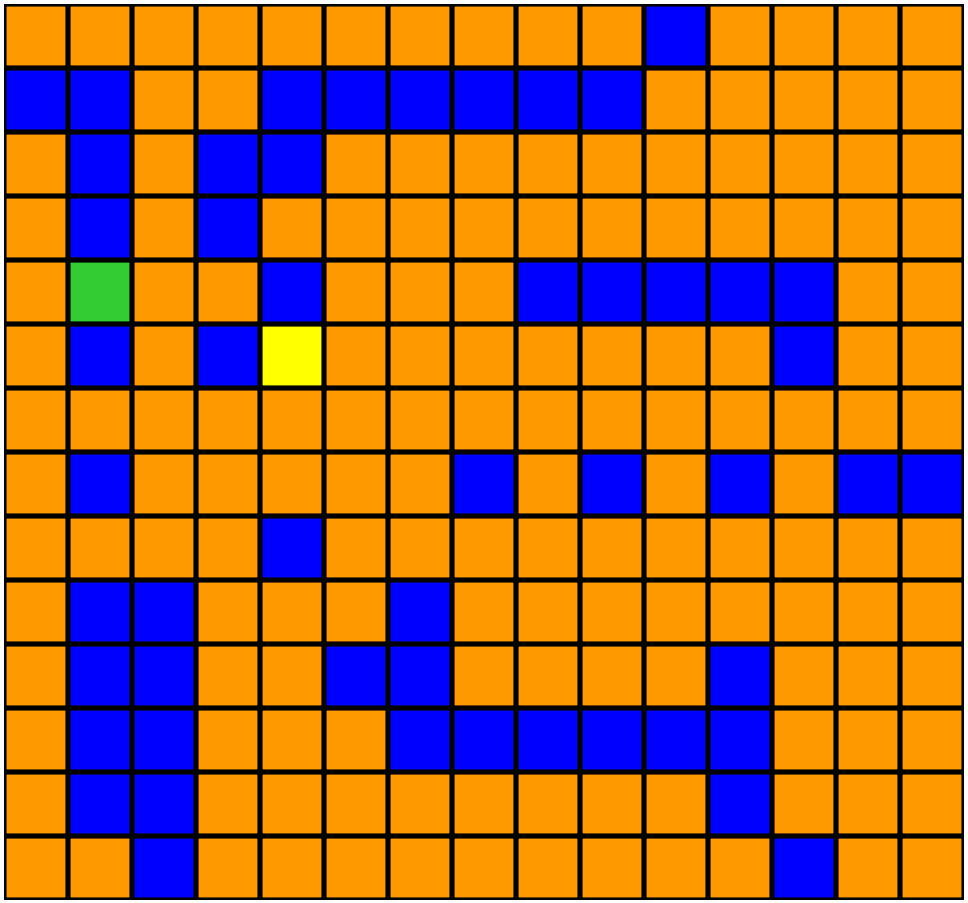
# Wire Routing



# Lee's Wire Router

start pin

end pin

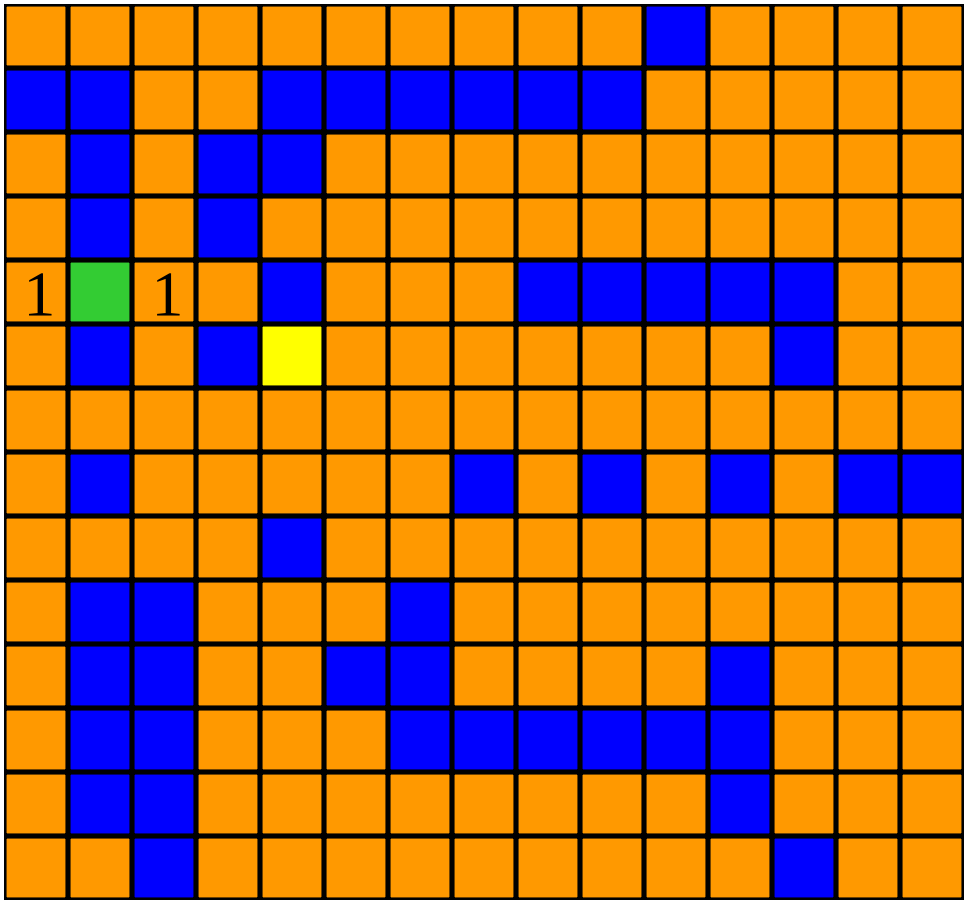


Label all reachable squares 1 unit from start.

# Lee's Wire Router

start pin

end pin

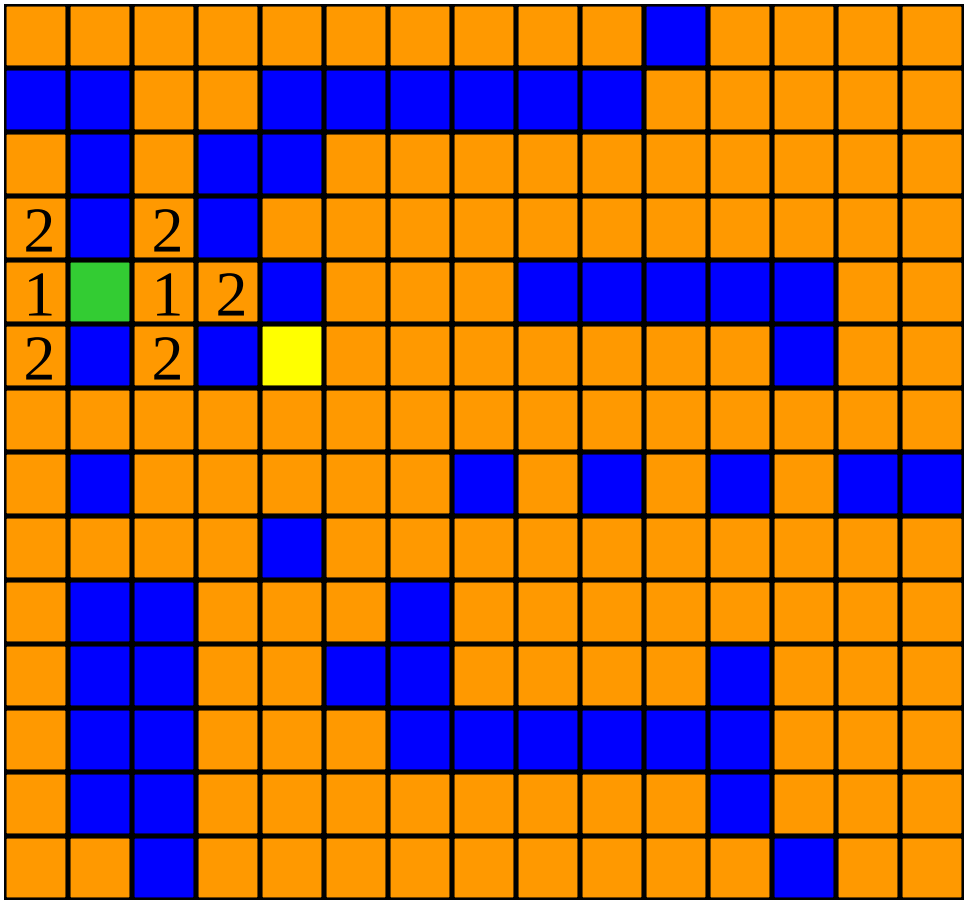



Label all reachable unlabeled squares 2 units from start.

# Lee's Wire Router

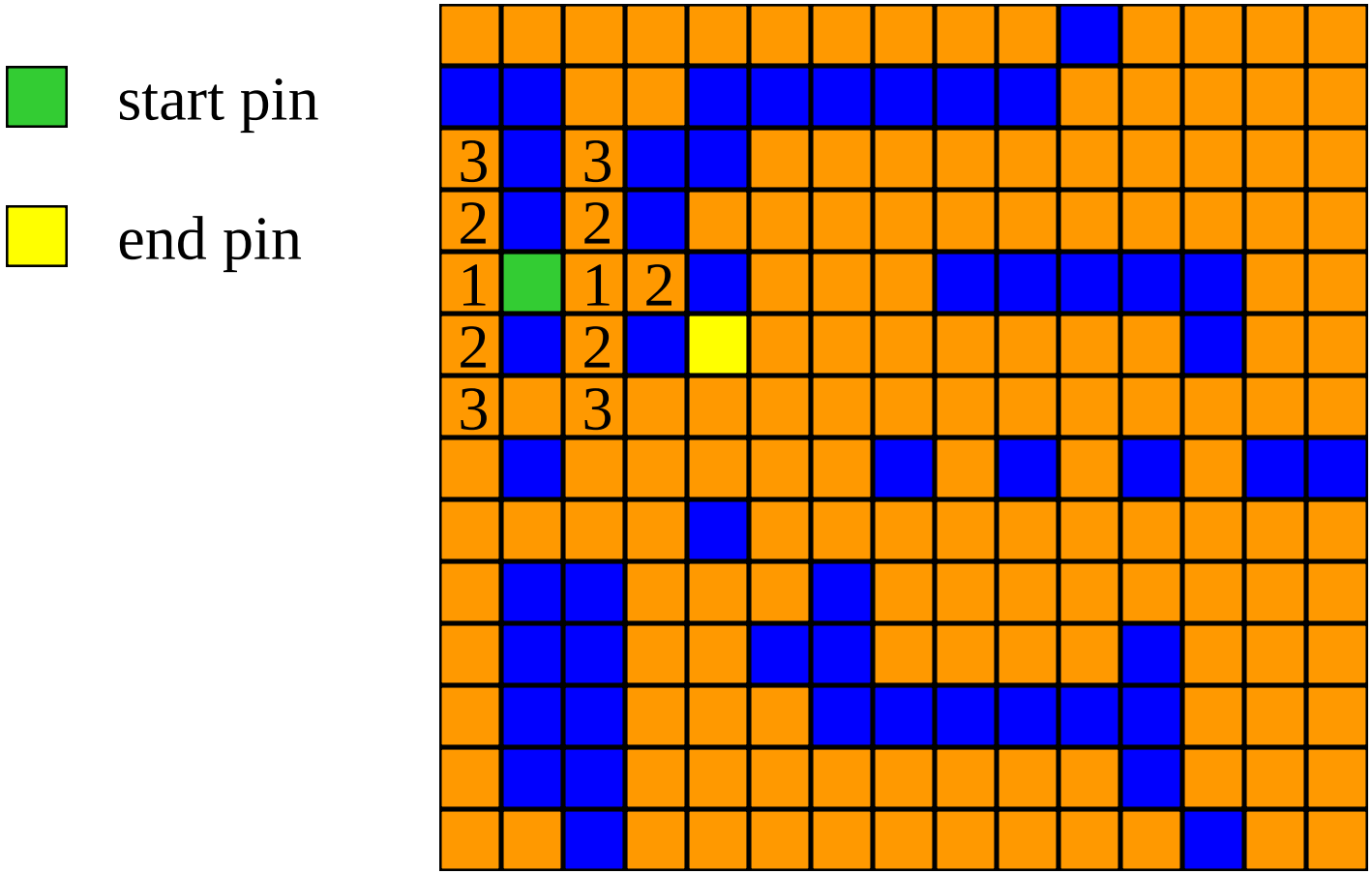
start pin

end pin




Label all reachable unlabeled squares 3 units from start.

# Lee's Wire Router

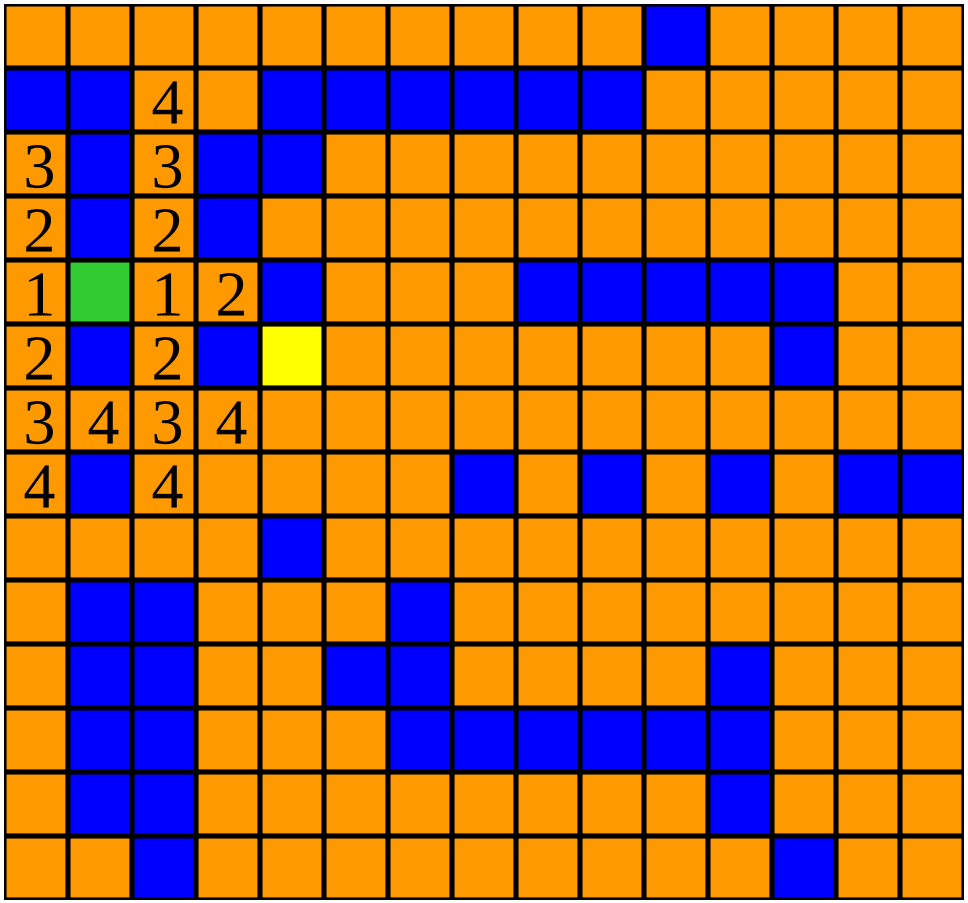



Label all reachable unlabeled squares 4 units from start.

# Lee's Wire Router

start pin

end pin

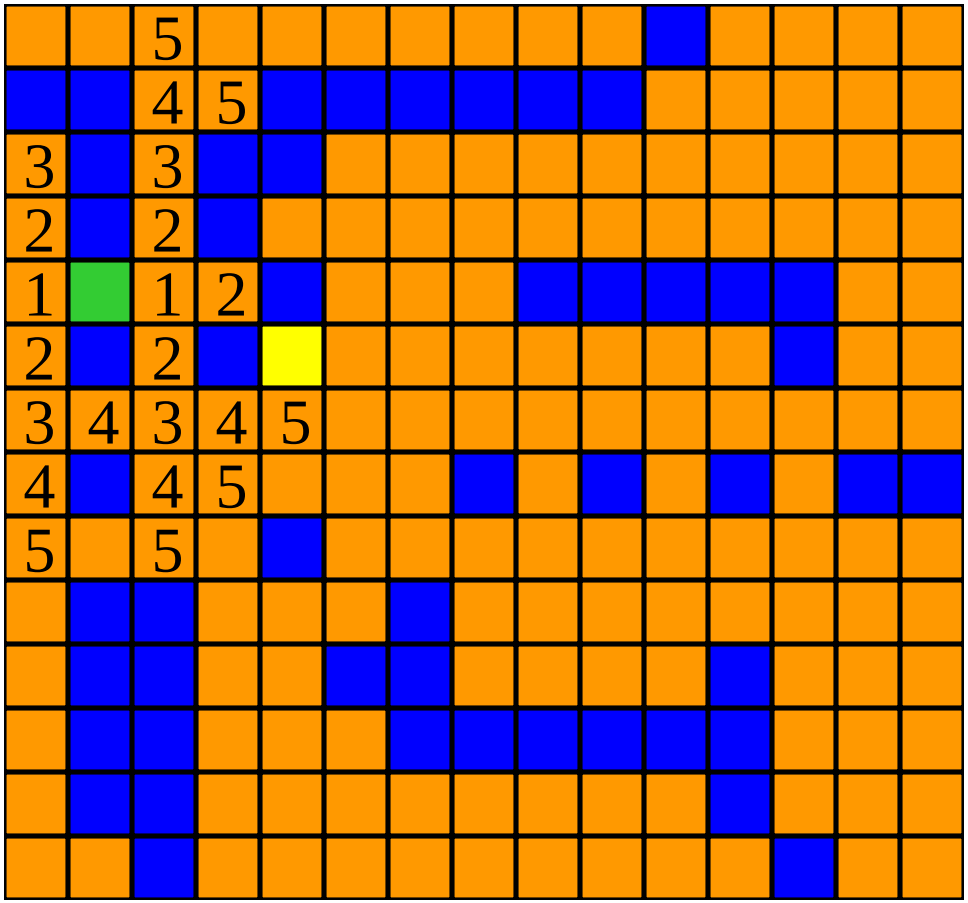



Label all reachable unlabeled squares 5 units from start.

# Lee's Wire Router

start pin

end pin

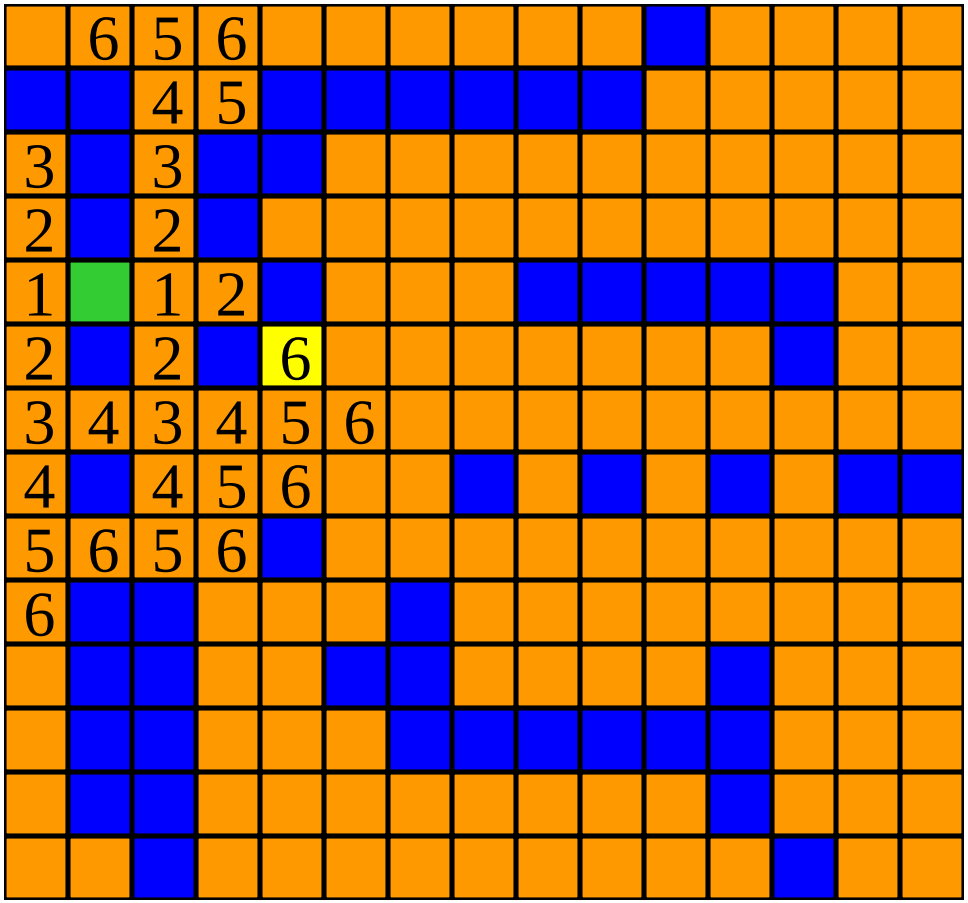



Label all reachable unlabeled squares 6 units from start.

# Lee's Wire Router

start pin

end pin

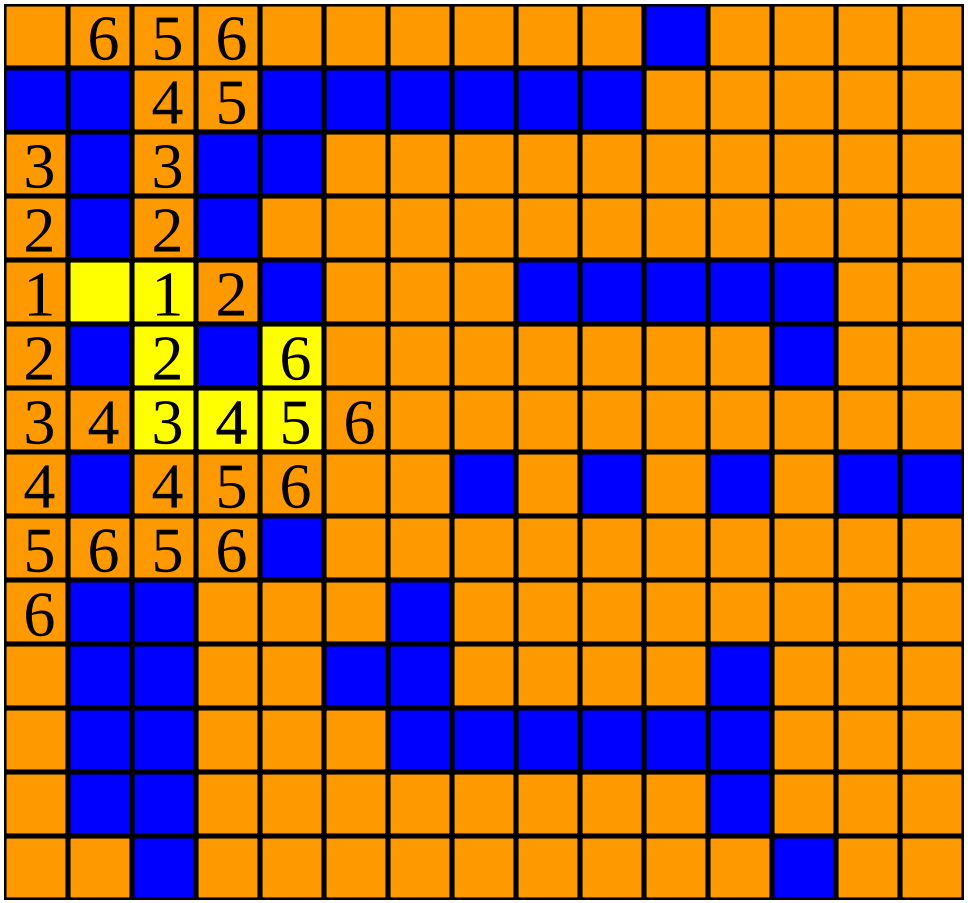



End pin reached. Traceback.

# Lee's Wire Router

start pin

end pin




End pin reached. Traceback.

# Queue Operations

- IsEmpty … return true iff queue is empty
- Front … return front element of queue
- Rear … return rear element of queue
- Push … add an element at the rear of the queue
  - (*Enqueue*)
- Pop … delete the front element of the queue
  - (*Dequeue*)

# Queue in an Array

- Use a 1D array to represent a queue.
- Suppose queue elements are stored with the front element in queue[0], the next in queue[1], and so on.

# Derive From arrayList

| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| a | b | c | d | e | | | | | | | | | | |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | | | | | | | | |

- Pop() => delete queue[0]
  - O(queue size) time
- Push(x) => if there is capacity, add at right end
  - O(1) time

# O(1) Pop and Push

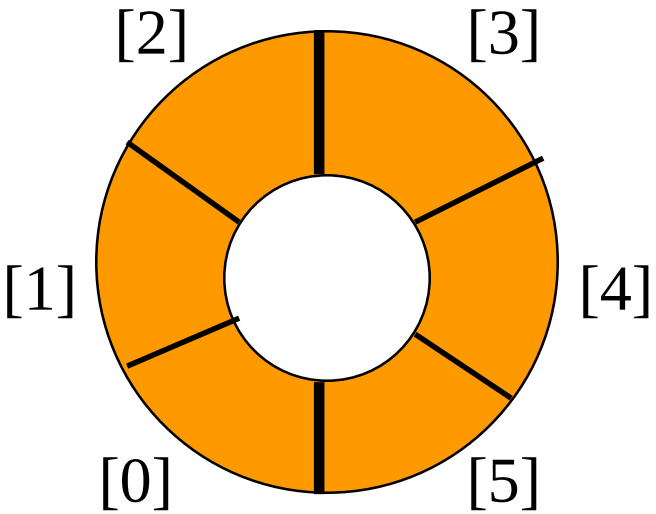
– to perform each opertion in O(1) time (excluding array doubling), we use a circular representation.

# Custom Array Queue

- Use a 1D array queue.

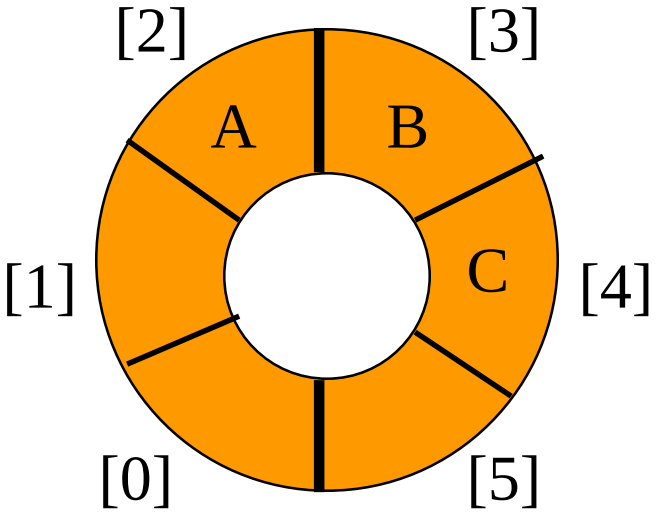
queue[]

- Circular view of array.



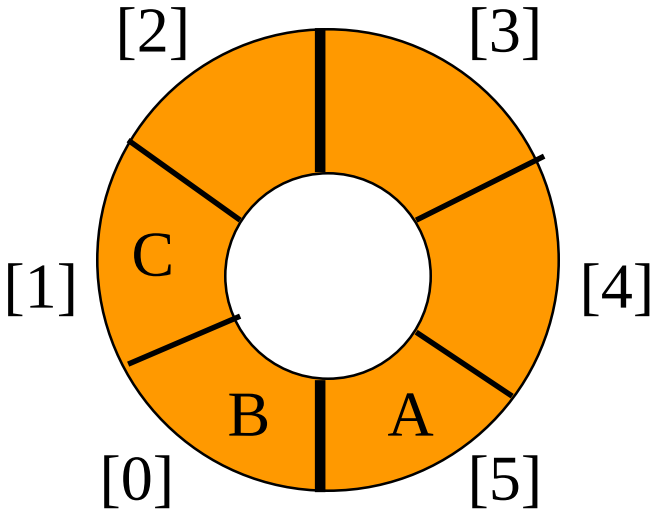
# Custom Array Queue

- Possible configuration with 3 elements.



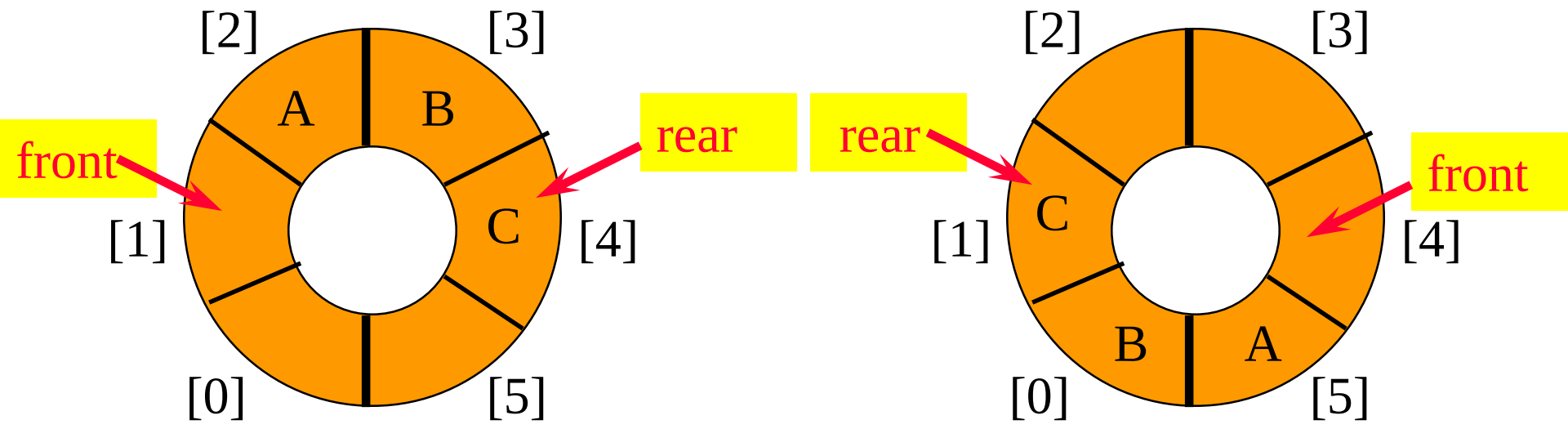
# Custom Array Queue

- Another possible configuration with 3 elements.



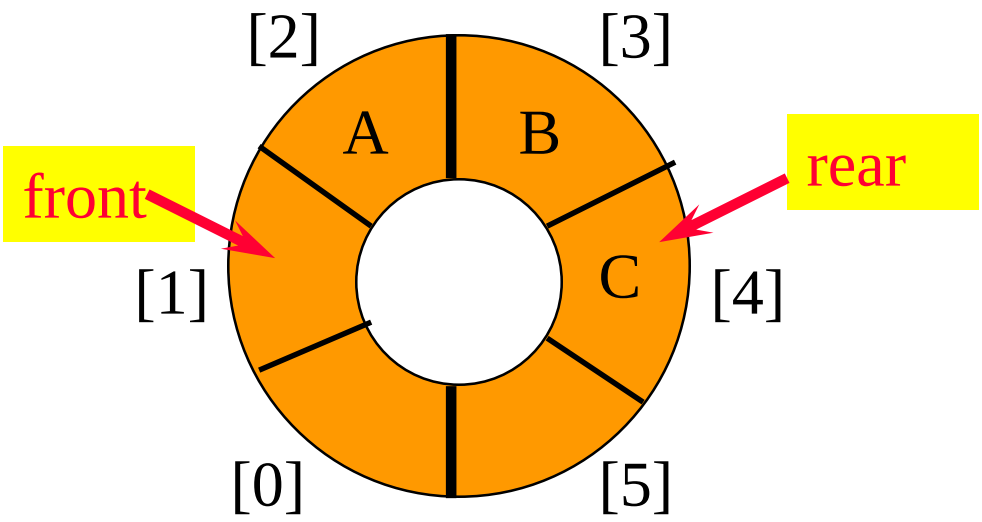
# Custom Array Queue

- Use integer variables front and rear.
  - front is one position counterclockwise from first element
  - rear gives position of last element



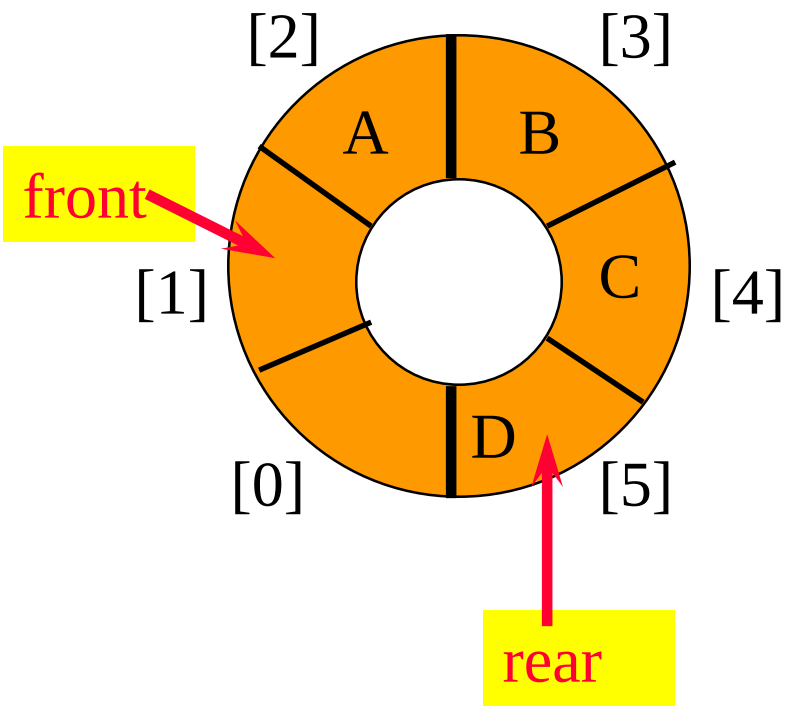
# Push An Element

- Move rear one clockwise.



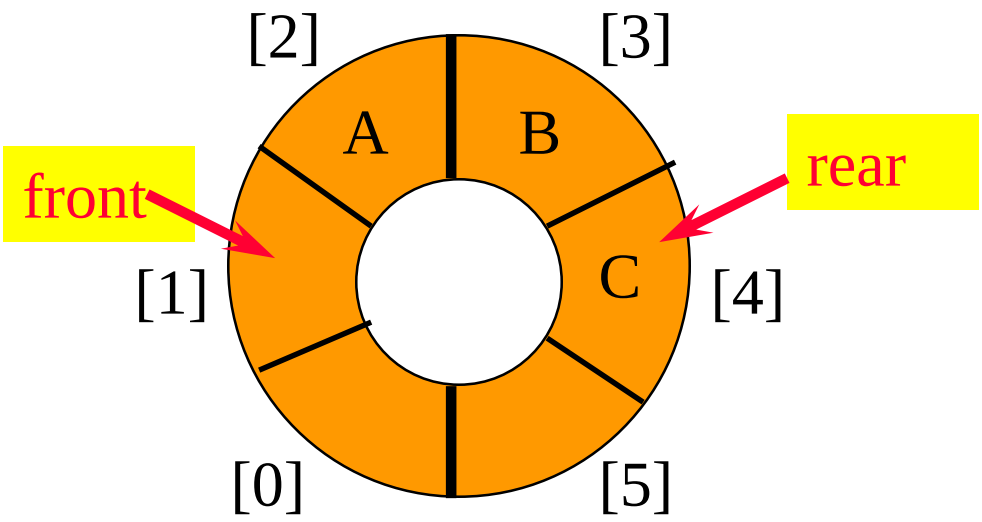
# Push An Element

- Move rear one clockwise.
- Then put into queue[rear].



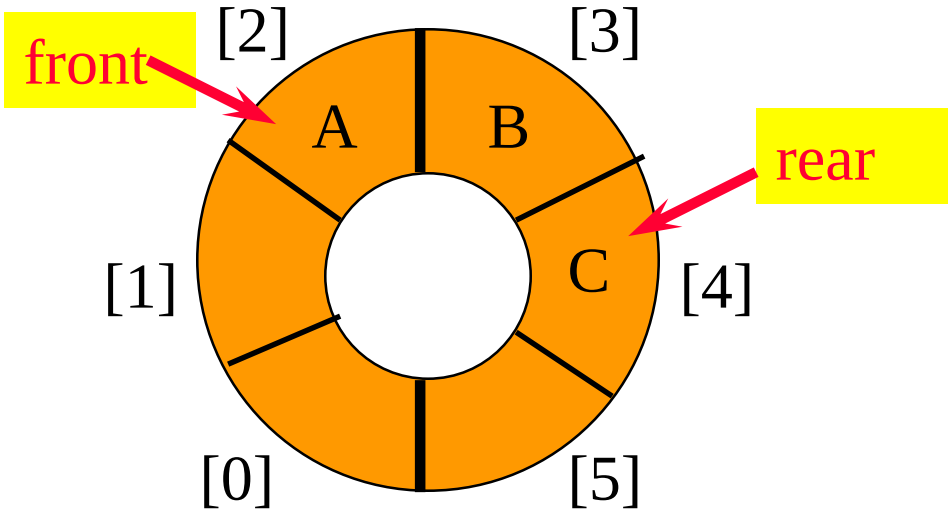
# Pop An Element

- Move front one clockwise.



# Pop An Element

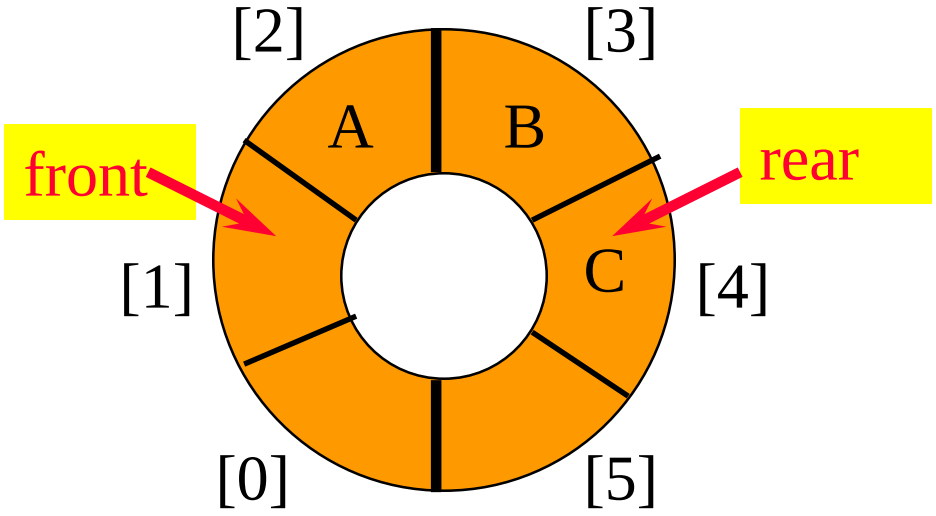
- Move front one clockwise.
- Then extract from queue[front].



# Moving rear Clockwise

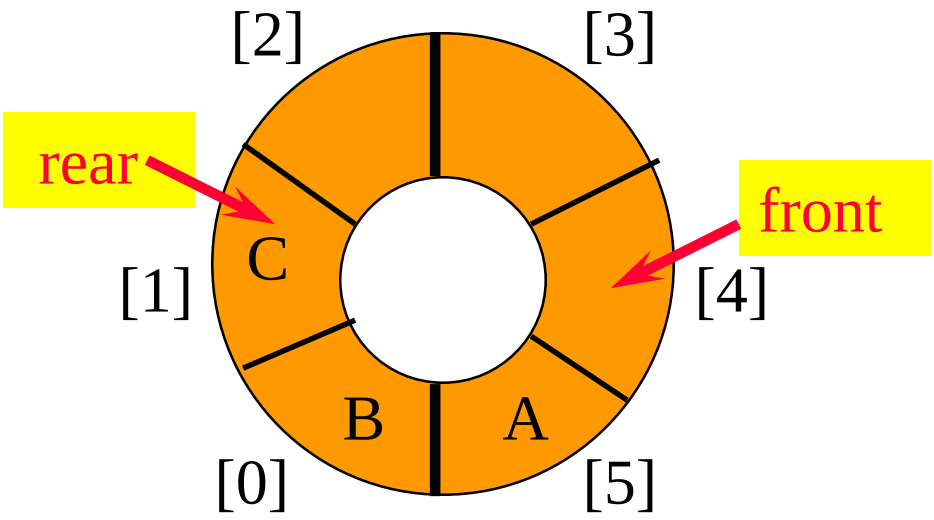
- rear++;

  if (rear = = capacity) rear = 0;

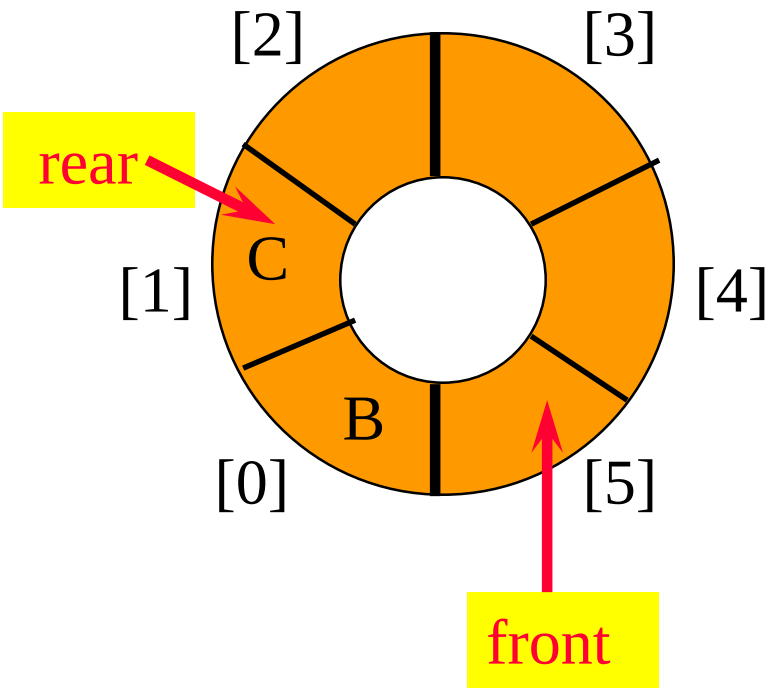



- rear = (rear + 1) % capacity;

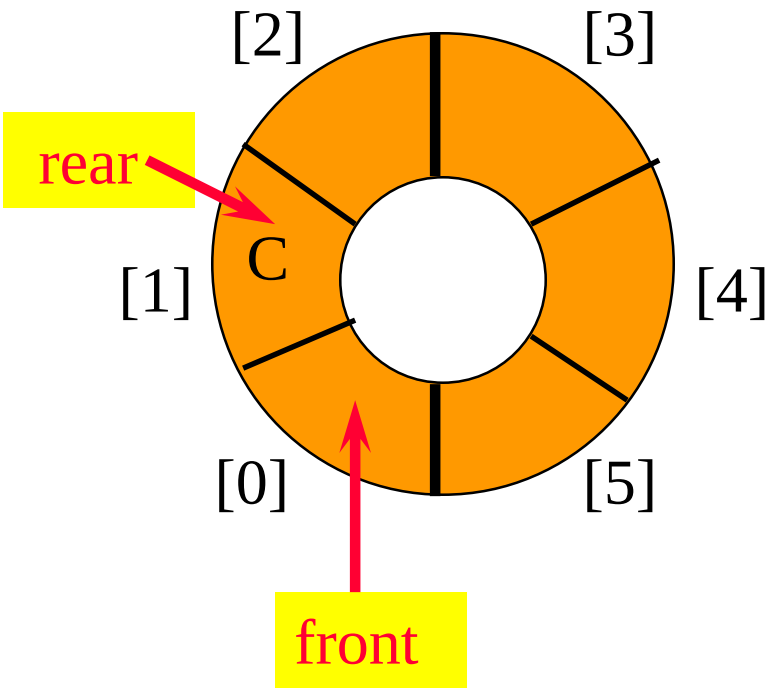
# Empty That Queue



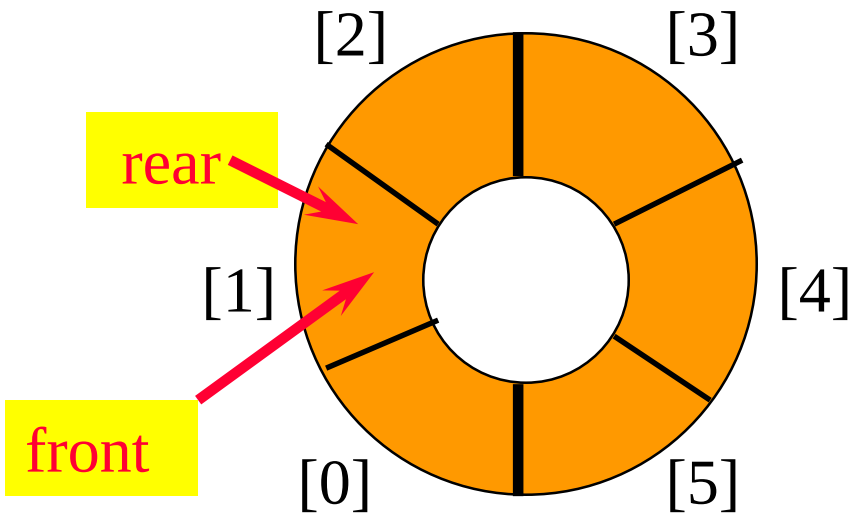
# Empty That Queue



# Empty That Queue

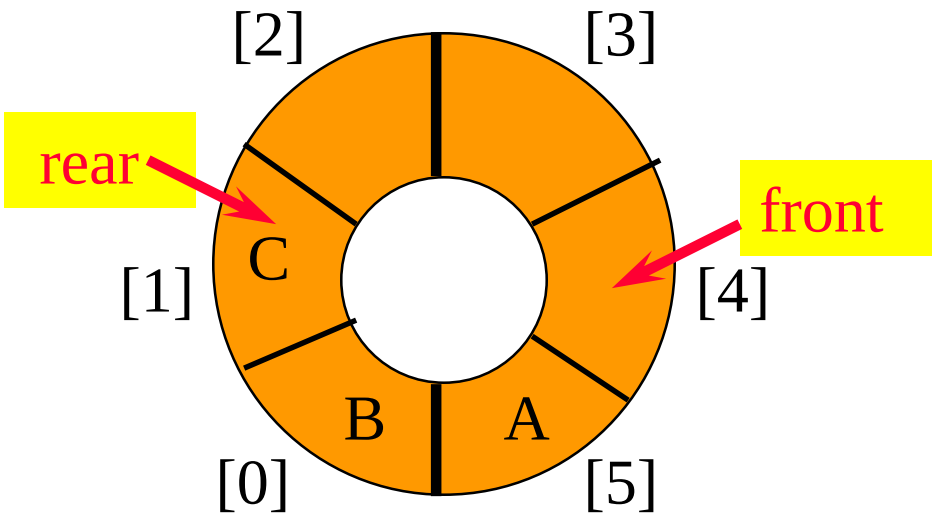


# Empty That Queue

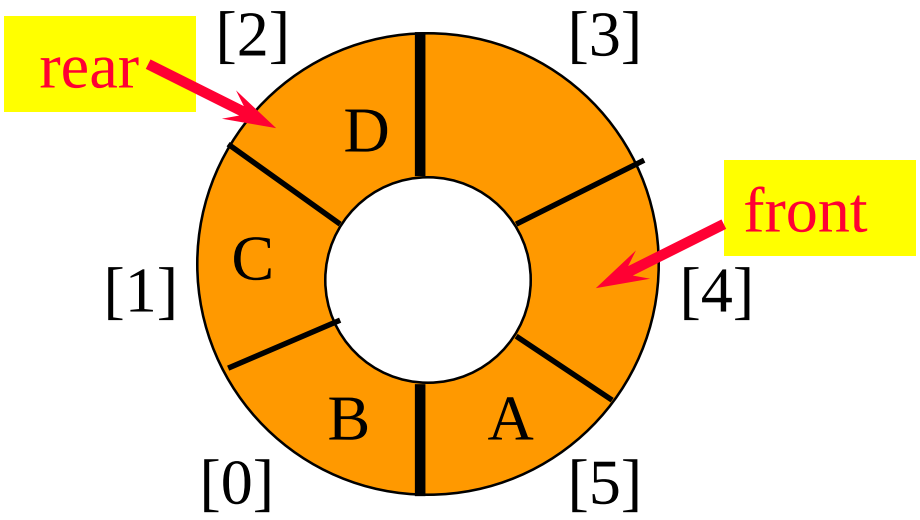



- When a series of removes causes the queue to become empty, front = rear.
- When a queue is constructed, it is empty.
- So initialize front = rear = 0.

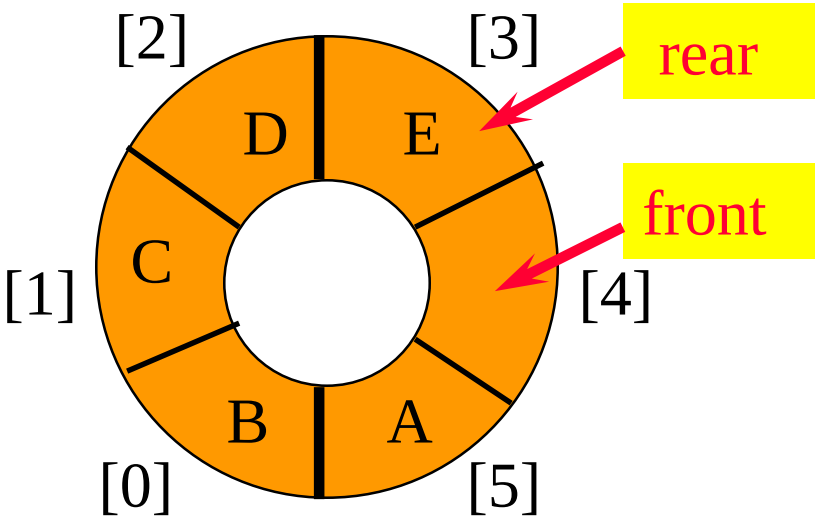
# A Full Tank Please



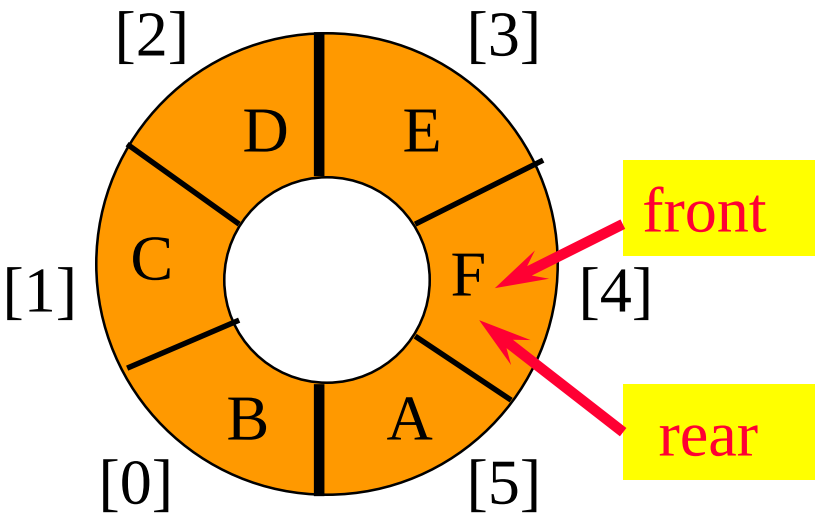
# A Full Tank Please



# A Full Tank Please



# A Full Tank Please




- When a series of adds causes the queue to become full, front = rear.
- So we cannot distinguish between a full queue and an empty queue!
- NOTE: If the queue is full after enqueuing, the queue is added a linear list of size of current queue and the elements from 0 to rear are moved to locations from number of current size to a new rear, (rear+current size), . The time complexity is the size of current queue size as described previously.

# Ouch!!!!!

- Remedies.
  - Don’t let the queue get full.
    - When the addition of an element will cause the queue to be full, increase array size.
    - This is what the text does.
  - Define a boolean variable lastOperationIsPush.
    - Following each push set this variable to true.
    - Following each pop set to false.
    - Queue is empty iff (front == rear) && !lastOperationIsPush
    - Queue is full iff (front == rear) && lastOperationIsPush

# Ouch!!!!!

- Remedies (continued).
  - Define an integer variable size.
    - Following each push do size++.
    - Following each pop do size--.
    - Queue is empty iff (size == 0)
    - Queue is full iff (size == arrayLength)
  - Performance is slightly better when first strategy is used.