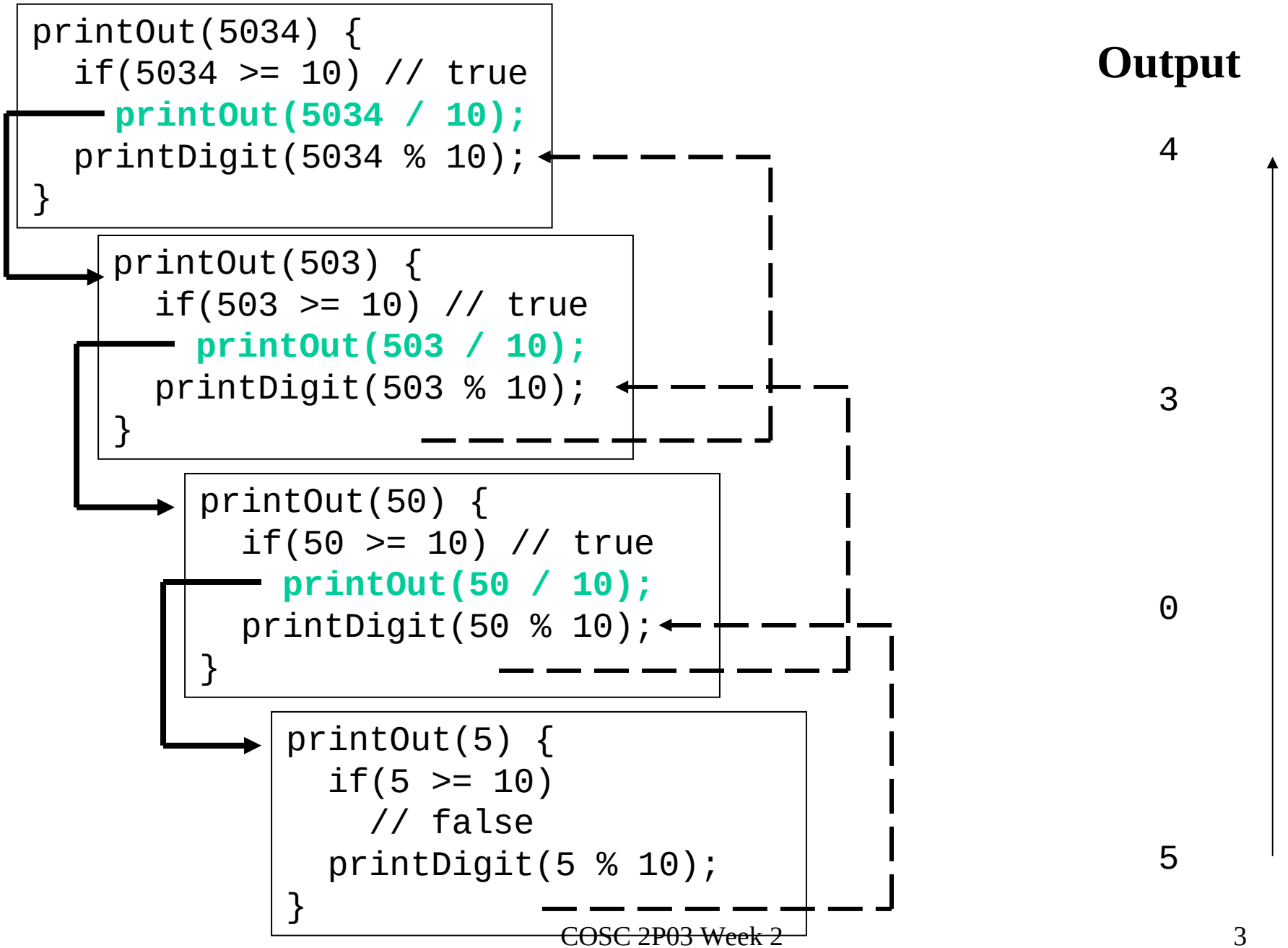
The Rules of Recursion

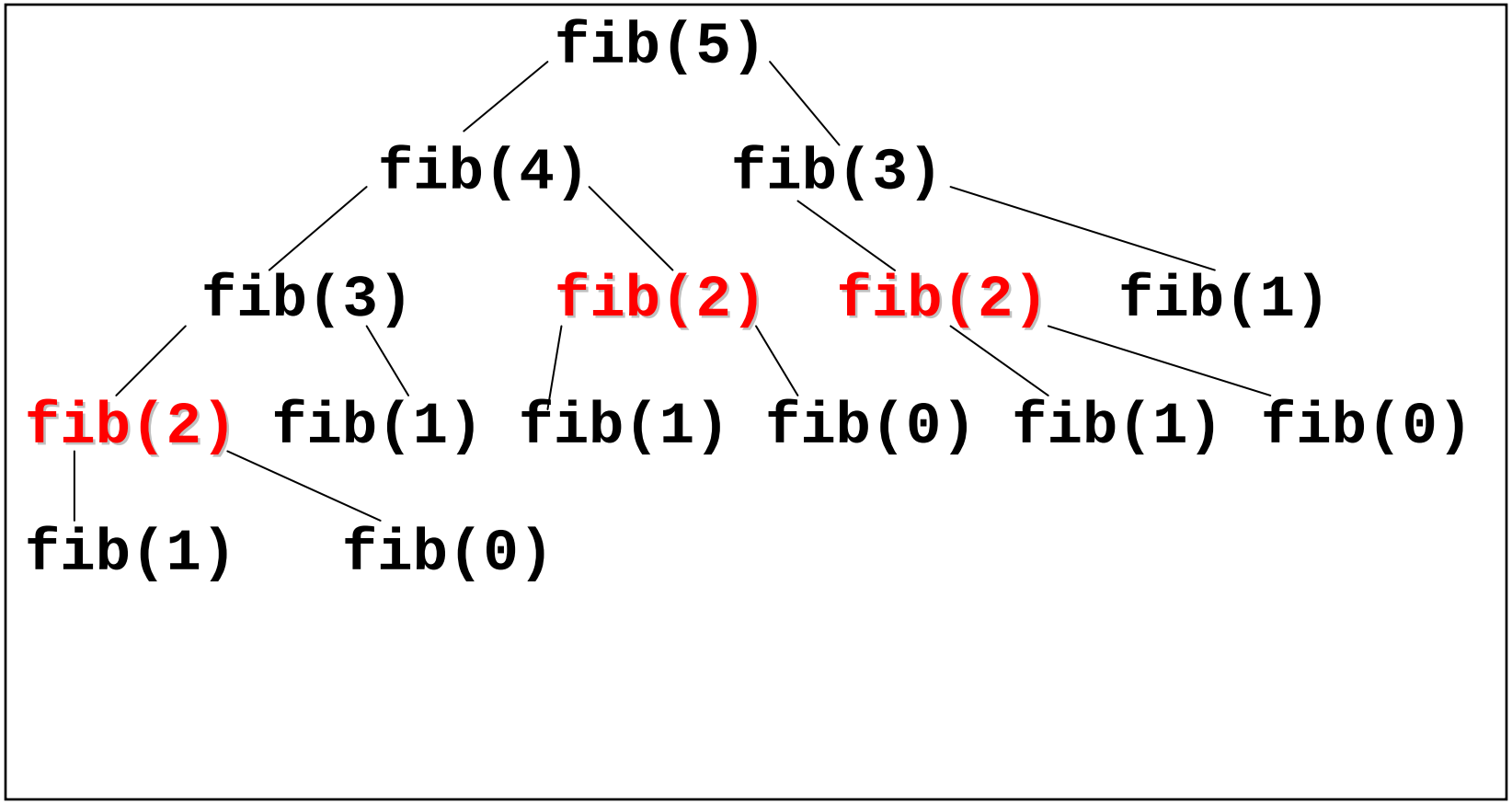
1. There must be **base cases** that can be solved without recursion
2. Recursive calls must always **make progress** towards a base case
3. Assume that **all recursive calls work**
4. **Never duplicate work** by solving the *same* instance of a problem in *separate* recursive calls

Recursive method `printOut`

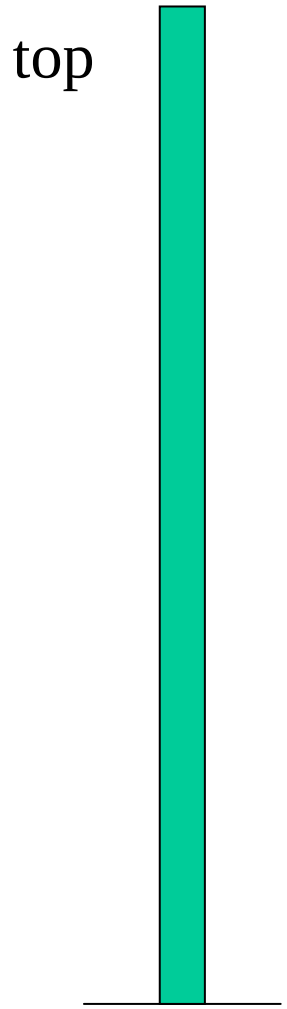
```
public static void printOut(int n)
{
  if(n >= 10)
    printOut(n/10);
  printDigit(n % 10);
}
```


printOut(5034) {
if(5034 >= 10) // true
printOut(5034 / 10);
printDigit(5034 % 10);
}
printOut(503) {
if(503 >= 10) // true
printOut(503 / 10);
printDigit(503 % 10);
}
printOut(50) {
if(50 >= 10) // true
printOut(50 / 10);
printDigit(50 % 10);
}
printOut(5) {
if(5 >= 10)
// false
printDigit(5 % 10);
}
Output
4
3
0
5

Sequence of calls to determine F_5



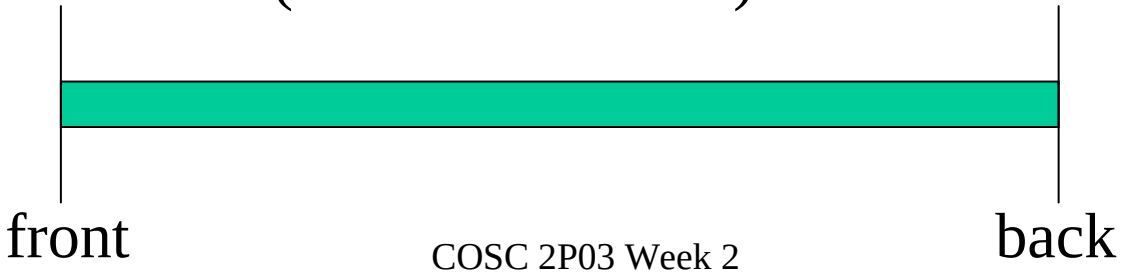
Stacks - review



- A Last-In First-Out (LIFO) structure
- Basic Operations:
  - push: insert data item onto top of stack
  - pop: remove data item from top of stack
- Additional Operations
  - isEmpty: determine if stack is empty
  - top: return (but don’t remove) top data item

Queues - review

- A First-In First-Out (FIFO) structure
- Basic Operations:
  - enqueue: add data item to back of queue
  - dequeue: remove data item from front of queue
- Additional Operations:
  - isEmpty: determine if queue is empty
  - isFull: determine if queue is full
  - front: return (but don’t remove) front data item



Double-Ended Queues (Deques)

- Items can be added or removed from either end
- Operations:
  - enqueueLeft
  - enqueueRight
  - dequeueLeft
  - dequeueRight

Special Deques

- *Input-Restricted Deque:* input restricted to one end (e.g. `EnqueueLeft` only)
- *Output-Restricted Deque:* output restricted to one end (e.g. `DequeueLeft` only)
- *Stack:* usage restricted to `EnqueueLeft` and `DequeueLeft`
- *Queue:* usage restricted to `EnqueueRight` and `DequeueLeft`

Priority Queues

- Each data item has a *key* or *priority*
- Ordering within queue is based on key
- Basic Operations:
  - insert: add data item according to its key
  - deleteMin: remove data item with smallest key

front – item with smallest key is here

back – item with largest key is here

Priority Queue Example

```
if(q.isEmpty())
   download(r);
else {
   t = q.front;
   if(r.level < t.level)
     insert r before t;
   else {
     while((t.next != null) &&
       (r.level >= t.next.level)
             t = t.next;
     insert r after t;
   }
}
```

Representation of a general tree

- No maximum number of children per node
- Keep children in something similar to a linked list

```
class TreeNode
{
  <declarations for info stored in node,
  e.g. int info;>
  TreeNode firstChild;
  TreeNode nextSibling;

   public void displayNode(){…}
      // display info stored in node
}
```

Representation of a binary tree

```
class BinaryNode
{
  < declarations for info stored in node,
  e.g. int info;>
  BinaryNode left;
  BinaryNode right;

  public void displayNode(){…}
     // display info stored in node
}
```

Tree Traversals

Breadth-first traversal:

- starting from root, visit all nodes on each level in turn, from left to right

Depth-first traversals:

- **Preorder:** visit root, traverse left, traverse right
  - General case: visit root, then traverse subtrees L→R
- **Postorder:** traverse left, traverse right, visit root
  - General case: traverse subtrees L→R, then visit root
- **Inorder:** traverse left, visit root, traverse right

Breadth-first traversal

We use an initially-empty queue, TQ, of type `BinaryNode`.

```
BFT(BinaryNode T)
{
  TQ.enqueue(T);
  while(!TQ.isEmpty())
  {
    temp = TQ.dequeue();
    temp.displayNode();
    if(temp.left != null)
      TQ.enqueue(temp.left);
    if(temp.right != null)
      TQ.enqueue(temp.right);
  }
}
```