# CMPT 225

Recursion-part 3

# Recursive Searching

- Linear Search
- Binary Search

- Find an element in an array, return its position (index) if found, or -1 if not found.

# Linear Search Algorithm (Java)

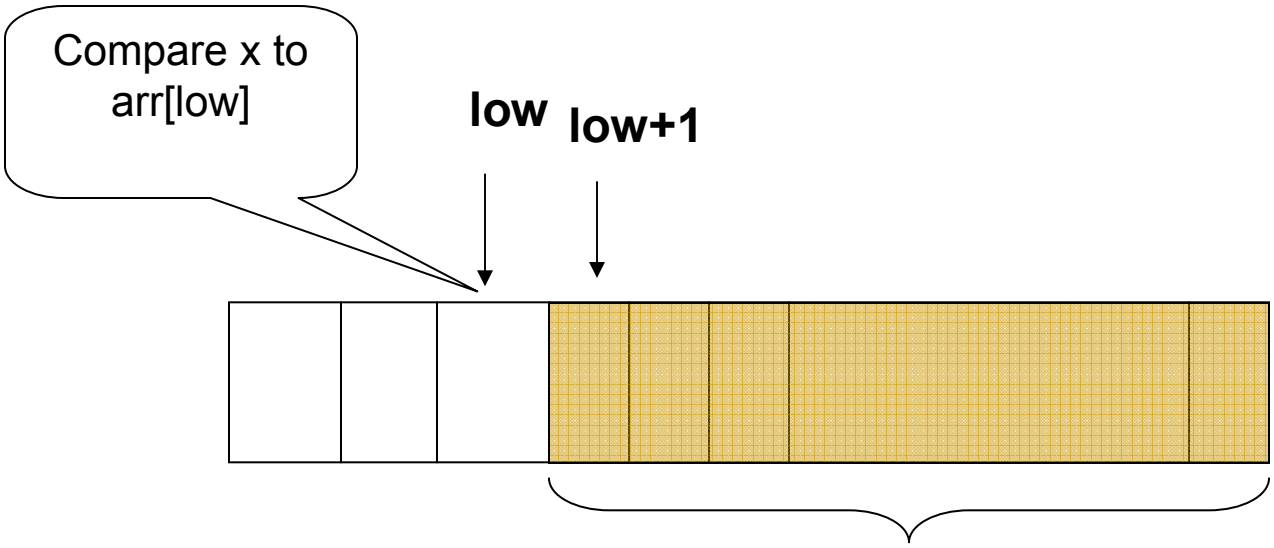
```
public int linSearch(int[] arr,
                     int target)
{
   for (int i=0; i<arr.size; i++) {
      if (target == arr[i]) {
         return i;
      }
   } //for
   return -1; //target not found
}
```

# Recursive Binary search

```
int recLinSearch (int[] arr, int low, int x)
```

To search the entire array for 2

```
index=recLinSearch(arr,0 ,2);
```



# Recursive Linear Search Algorithm

- Base case
  - Found the target or
  - Reached the end of the array
- Recursive case
  - Call linear search on array from the next item to the end

```
public int recLinSearch(int[] arr,int low,int x) {
    if (low >= arr.length) { // reach the end
        return -1;
    } else if (x == arr[low]){
        return low;
    } else
        return recLinSearch(arr, low + 1, x);
    }
}
```

# How many comparison does the recLinSearch perform in worst case?

- Suppose T(n) is the number of comparison where n is the size of the elements in the array.
- We have:

$T(0)=0$

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

- By expanding the T(n) n times we have

$T(n)=1+[1+T(n-2)]=2+T(n-2)=\ldots=k+T(n-k)=\ldots=n+T(0)=n=O(n)$

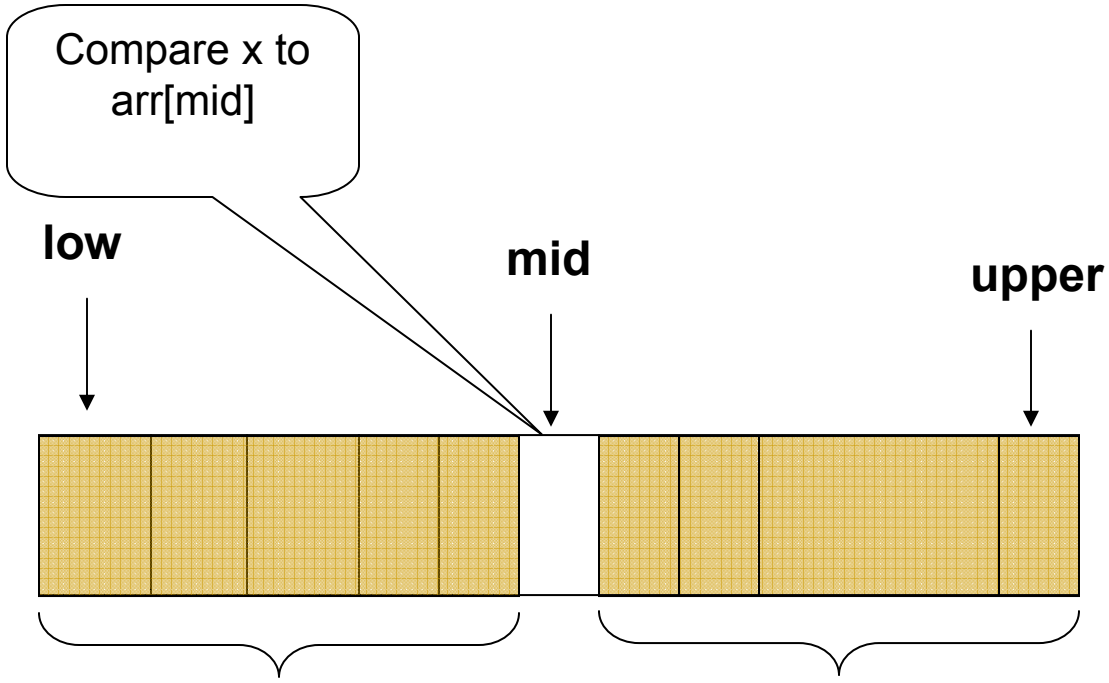
- Thus, the recLinSearch is not more efficient than the non-recursive linear search.

# Recursive linear search

```
int recBinSearch (int[] arr, int lower, int upper, int x)
```

To search the whole array for 2

```
index=recBinSearch(arr,0 , arr.length, 2);
```




**if x<arr[mid]** **if x>arr[mid]**

```
recBinSearch(arr, low, mid-1, x)   recBinSearch(arr, mid+1, upper, x)
```

# Recursive Binary Search Algorithm

```
public int binSearch(
        int[] arr, int lower, int upper, int x)
{
    int mid = (lower + upper) / 2;
    if (lower > upper) {// empty interval
        return - 1; // base case
    } else if(arr[mid] == x){
        return mid; // second base case
    } else if(arr[mid] < x){
         return binSearch(arr, mid + 1, upper, x);
    } else { // arr[mid] > target
         return binSearch(arr, lower, mid - 1, x);
    }
}
```

# How many comparison does the recBinSearch perform in worst case?

- Suppose T(n) is the number of comparison where n is the size of the elements in the array, and assume $\boldsymbol{n = 2^k}$ (e.g. if $n$ = 128, $k$ = 7)

- We have:

$T(1)=1$

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

- By expanding the T(n) n times we have

$T(n)=1+[1+T(n/2)]=2+T(n/2)=\ldots=k+T(n/2^k)=k+T(1)=k+1=O(\log_2 n)$

Because $\boldsymbol{n = 2^k}$, $\boldsymbol{k = \log_2 n}$

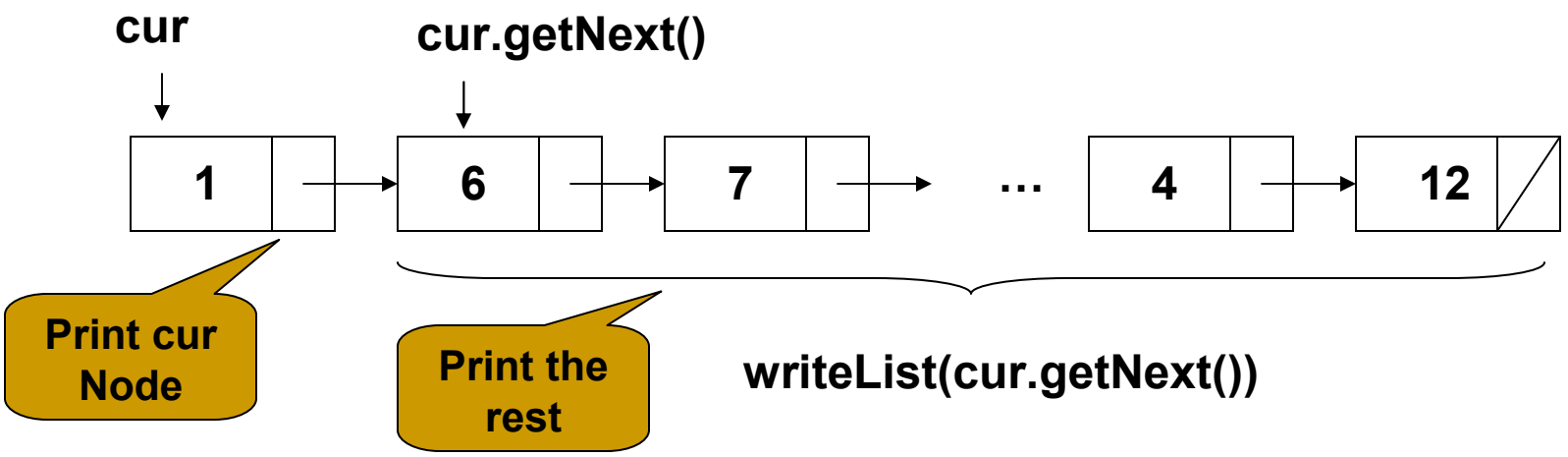
# Binary Search vs Linear Search

| | Linear | Binary |
|---|---|---|
| ***N*** | ***N*** | $\log_2(N)$ |
| 10 | 10 | 4 |
| 100 | 100 | 7 |
| 1,000 | 1000 | 10 |
| 10,000 | 10,000 | 14 |
| 100,000 | 100,000 | 17 |
| 1,000,000 | 1,000,000 | 20 |
| 10,000,000 | 10,000,000 | 24 |

# Processing linked list recursively.

- It's possible and sometimes desirable to process linked list recursively.
- Eg. Displaying the elements in the list recursively.

`writeList(Node cur)`: displays the elements from cur to the end of the list.



```
private void writeList(Node cur){
  if (cur==null)
     return;
  System.out.println(cur.getItem());
  writeList(cur.getNext());
}
```

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

$T(0)=0$

# Writing a list backward.

- A non-recursive solution.

```
private void wirteListBackward1(){
   Object temp[]= new Object[size];
   Node cur=head;
   for(int i=0; i<size; i++, cur=cur.getNext())
       temp[i]=cur.getItem();
   for(int i=size-1; i>=0; i--)
       System.out.println(temp[i]);
}
```

- Needs an extra array of the same size as list.
  - The space complexity is O(n) –where n is the size of the array

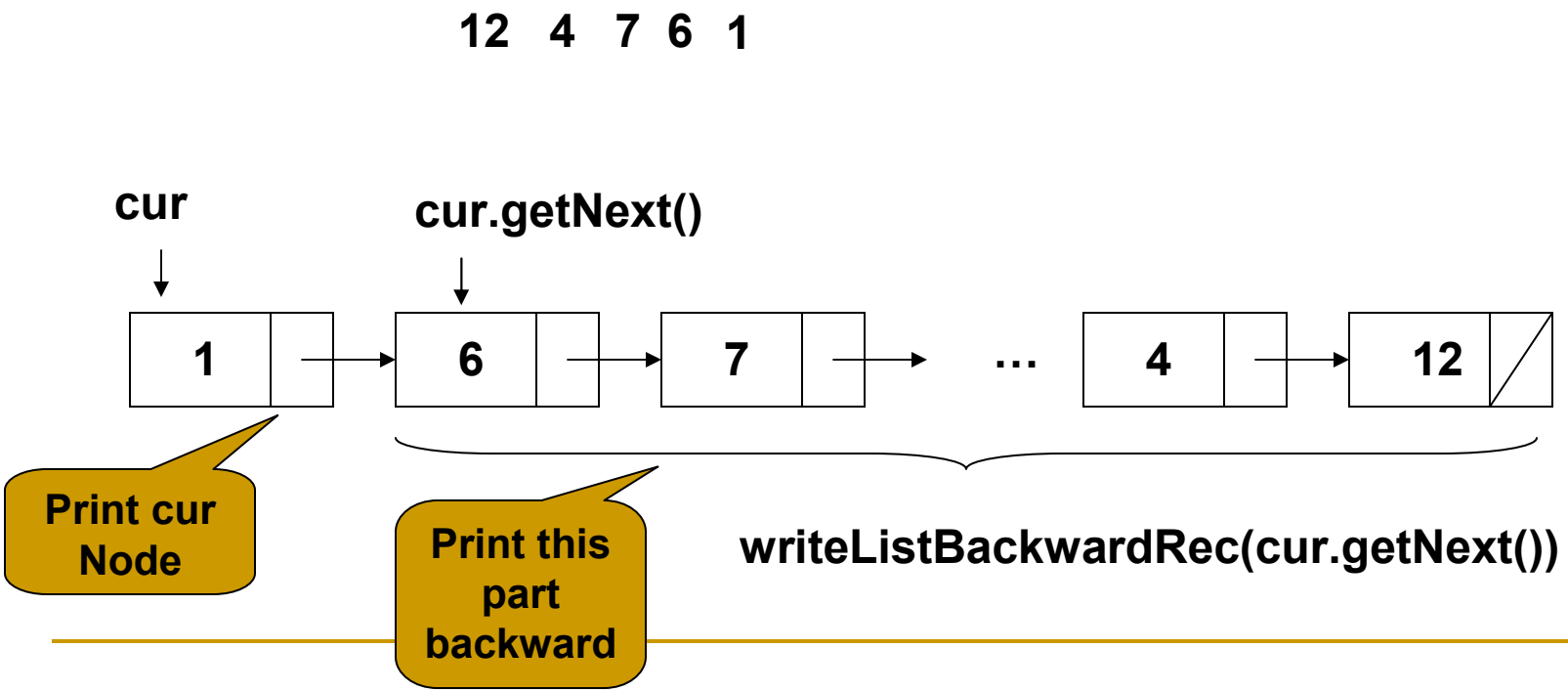
# Writing a list backward-a new solution.

```
private void wirteListBackward2(){
   for(int i=size-1; i>=0; i--){
        //returns the reference to the ith element in the
        //list.
        Node cur=get(i);
        System.out.println(cur.getItem());
   }
}
```

- Does not need extra space.
  - The space complexity is O(1)
- However, it's slow
  - The get(i) method requires i operations.
  - Therefore, the total number of operations: $(n-1)+(n-2)+\ldots+1=O(n^2)$

# Writing a list backward-a recursive solution.

`writeListBackwardRec(Node cur)`: displays the elements from cur to the end of the list in a reverse order.



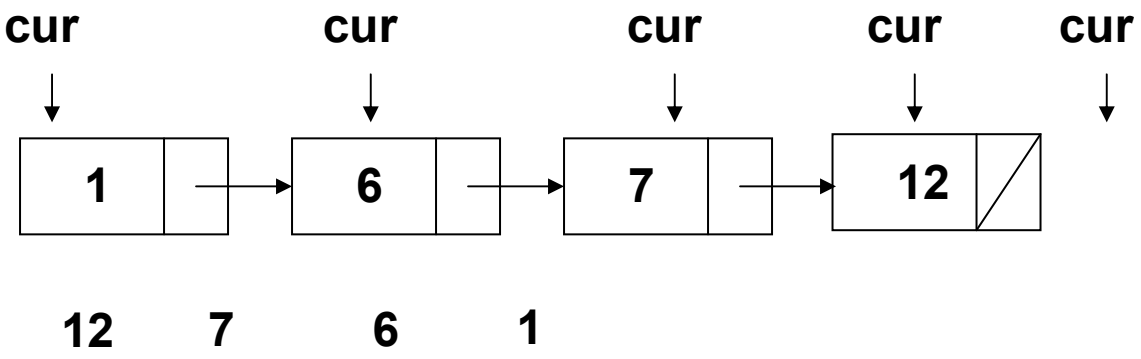
```
private void writeListBackwardRec(Node cur){
   if (cur == null)
        return;
   writeListBackwardRec(cur.getNext());
   System.out.println(cur.getItem());
}
```

The time complexity is O(n), therefore it's and optimal solution in terms of time.

What is the space complexity?

- Is it O(1)?
- No!!!. It requires a *stack* of size O(n) when it's executed.

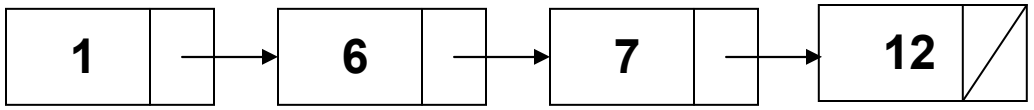
```
public void displayBackward(){
   writeListBackwardRec(head); //Z
}

private void writeListBackwardRec(Node cur){
   if (cur == null)
        return;
   writeListBackwardRec(cur.getNext()); //X
   System.out.println(cur.getItem()); //Y
}
```



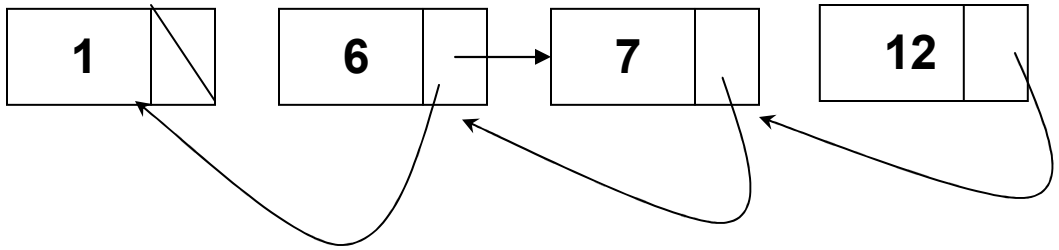

**Call Stack**

| |
|---|
| Cur:nul<br>Ret:x |
| Cur:12<br>Ret:X |
| Cur:7<br>Ret:X |
| Cur:6<br>Ret:X |
| Cur:1<br>Ret:Z |

# The optimal solution.

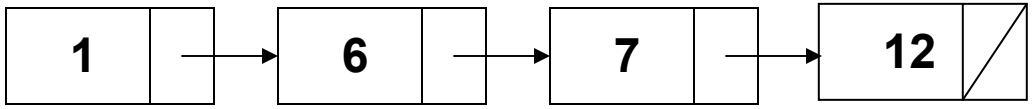



**Reverse the list.**




**Print the reversed list.**

**Restore the list to original order**




**O(n): time, O(1):space**

**O(n): time, O(1):space**

**O(n): time, O(1):space**

---

**Total O(n): time, O(1):space**

# Recursion Disadvantage 1

- Recursive algorithms have more overhead than similar iterative algorithms
  - Because of the repeated method calls (storing and removing data from call stack)
  - This may also cause a “stack overflow” when the call stack gets full
- It is often useful to derive a solution using recursion and implement it iteratively
  - Sometimes this can be quite challenging!

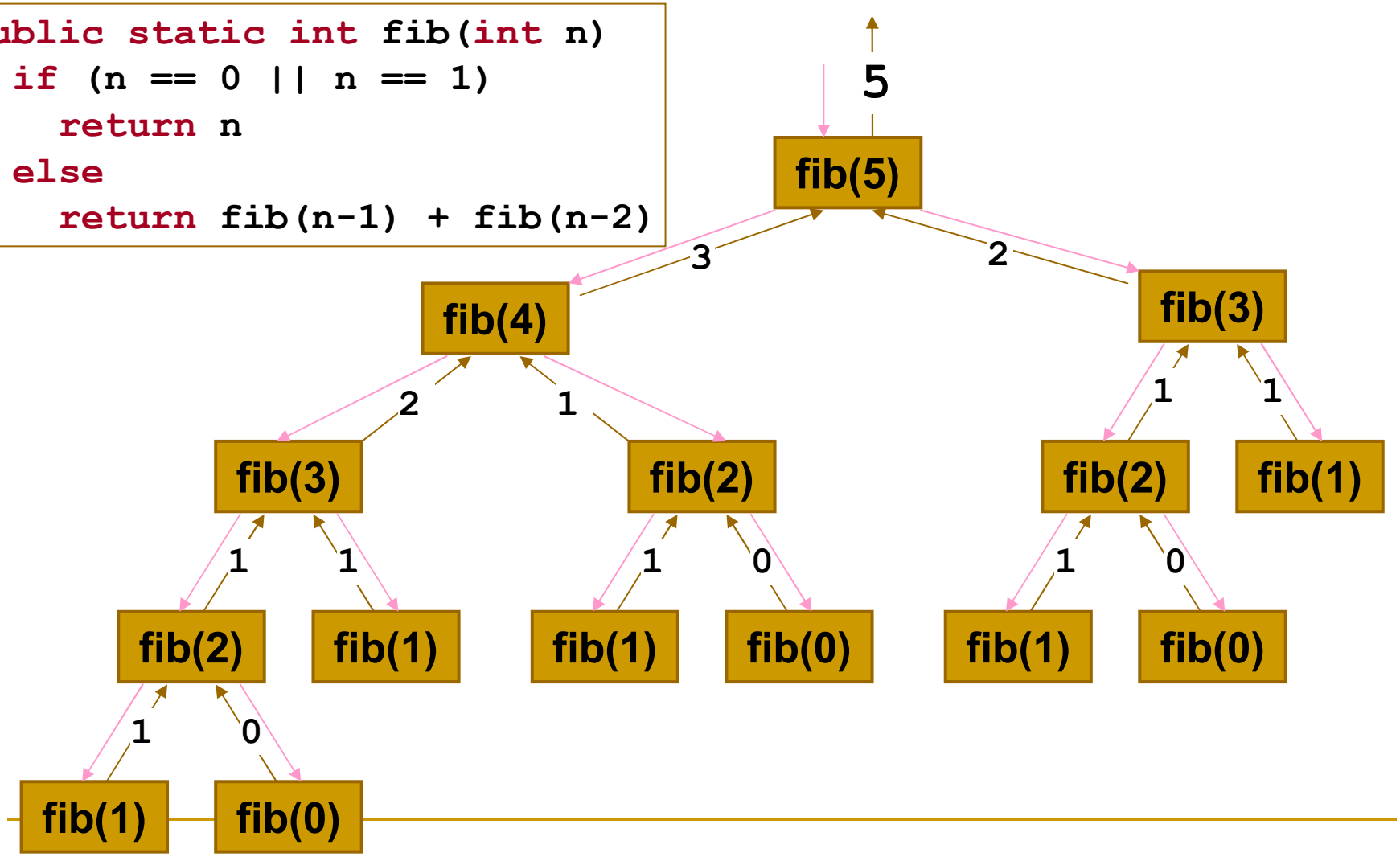
    (Especially, when computation continues after the recursive call -> we often need to remember value of some local variable -> stacks can be often used to store that information.)

# Recursion Disadvantage 2

- Some recursive algorithms are inherently inefficient
- An example of this is the recursive Fibonacci algorithm which repeats the same calculation again and again
  - Look at the number of times fib(2) is called
- Even if the solution was determined using recursion such algorithms should be implemented iteratively
- To make recursive algorithm efficient:
  - Generic method (used in AI): store all results in some data structure, and before making the recursive call, check whether the problem has been solved.
  - Make iterative version.

# Function Analysis for call `fib(5)`

```
public static int fib(int n)
  if (n == 0 || n == 1)
    return n
  else
    return fib(n-1) + fib(n-2)
```



# Iterative Fib solution

```
int itrFib(int n){
   int F0=0, F1=1, F2=0;
   for(int i=0; i<n; i++){
       F2=F0+F1;
       F0=F1;
       F1=F2;
   }
   return F2;
}
```

# Generic Fib Solution

```
int genericFib(int n){
   if(n==0 || n==1)
       return n;
   if(globalFib[n]>=0)
       return globalFib[n];
   globalFib[n]=genericFib(n-1)+genericFib(n-2);
   return globalFib[n];
}
```

- **The globalFib is a global array that is initially set to -1.**

# Analyzing Recursive Functions

- Recursive functions can be tricky to analyze
- It is useful to trace through the sequence of recursive calls
- This can be done using a recursion tree
  - As shown for the Fibonacci function
- Recursion trees can also be used to determine the running time (in number of operations) of algorithms
  - Annotate the tree to indicate how much work is performed at each level of the tree
  - Determine how many levels of the tree there are

# Recursion and Induction

- Recursion is similar to mathematical induction as recursion solves a problem by
  - Specifying a solution for the base case and
  - Using the recursive case to derive solutions of any size from the solutions to smaller problems
- Induction proves a property by
  - Proving it is true for a base case (which is often true by definition) and
  - Proving that it is true for some number, *n*, if it is true for all numbers less than *n*

# Recursive Factorial Algorithm

```
public int fact (int x){
    // Should check for negative values of x
    if (x == 0){
        return 1;
    } else
        return n * fact(n - 1);
    }
}
```

- Prove, using induction, that the algorithm returns the values:
  - fact(0) = 0! = 1
  - fact($n$) = $n!$ = $n$ * ($n$ – 1) * ($n$ – 2) * … * 1 if $n$ > 0

# Proof by Induction of `fact` Method

- **Basis:** Show that the property is true for $n = 0$, i.e. that fact(0) returns 1
  - This is true by definition as fact(0) is the base case of the algorithm and returns 1
- Now establish that the property is true for an arbitrary k implies that it is true for k + 1
- **Inductive hypothesis:** Assume that the property is true for n = k, that is assume that
  - fact($k$) = $k * (k - 1) * (k - 2) * \ldots * 2 * 1$

# Proof by Induction of `fact` Method

- **Inductive conclusion:** Show that the property is true for $n = k + 1$, i.e., that fact $(k + 1)$ returns
  - $(k + 1) * k * (k - 1) * (k - 2) * \ldots * 2 * 1$
- By definition of the function fact$(k + 1)$ returns
  - $(k + 1)$ * fact$(k)$ – the recursive case
- And by the inductive hypothesis fact(k) returns
  - $k * (k - 1) * (k - 2) * \ldots * 2 * 1$
- Therefore fact$(k + 1)$ must return
  - $(k + 1) * k * (k - 1) * (k - 2) * \ldots * 2 * 1$
- Which completes the inductive proof