Recursion

Amit Chopra

amit.chopra@lancaster.ac.uk

Recall….

- We studied the Stack ADT and some applications
- Implementing function invocation via the call stack

Today

- Recursion in computer programming
  - Function calling itself
  - Apply same logic to *progressively smaller pieces* of the input
- Exploit recursive organization of data structures
  - Subarrays
  - Sublists
  - Subtrees
  - Subgraphs
- Some hints already
  - The maze problem
  - Binary search

```
f(A){
    …
     f(B)
    …
}
```

Overview

- Whenever we need to traverse a set of values, or run the same piece of logic a given number of times, we can use **iteration** or **recursion** to do so

- We can write any recursive solution in an iterative way, and we can write any iterative solution in a recursive way

- Which style we use depends on how elegant the relative solution is, but also on their performance characteristics
  - Quickly understanding which style to use for a particular problem will make you a much more effective engineer

Overview

- An **iterative** function is one which *uses a loop* to execute the same logic multiple times

```
void counter(int n) {
   for (int i = 0; i < n; i++) {
      print("loop!")
      }
   }
```

Overview

- An **iterative** function is one which *uses a loop* to execute the same logic multiple times

```
void counter(int n) {
   for (int i = 0; i < n; i++) {
      print("loop!")
      }
   }
```

- A **recursive** function is one which *calls itself* to execute the same logic multiple times

```
void counter(int n) {
      if (n != 0) {
         print("loop!")
         counter(n - 1)
         }
   }
```

Order of execution

- The order in which we execute logic in a recursive function depends whether it's before / after the self-call

- Here's an iterative function to print the iterator value:

```
void counter(int n) {
   for (int i = 0; i < n; i++) {
      print("loop " + i)
      }
   }
```

```
loop 0
loop 1
loop 2
loop 3
```

Order of execution

- The order in which we execute logic in a recursive function depends on if it's before / after the self-call

logic-before-recursion

```
void counter(int n) {
   if (n != 0) {
      print("loop " + (n – 1))
      counter(n - 1)
      }
}
```

```
loop 3
loop 2
loop 1
loop 0
```

logic-after-recursion

```
void counter(int n) {
   if (n != 0) {
      counter(n - 1)
      print("loop " + (n – 1))
      }
}
```

```
loop 0
loop 1
loop 2
loop 3
```

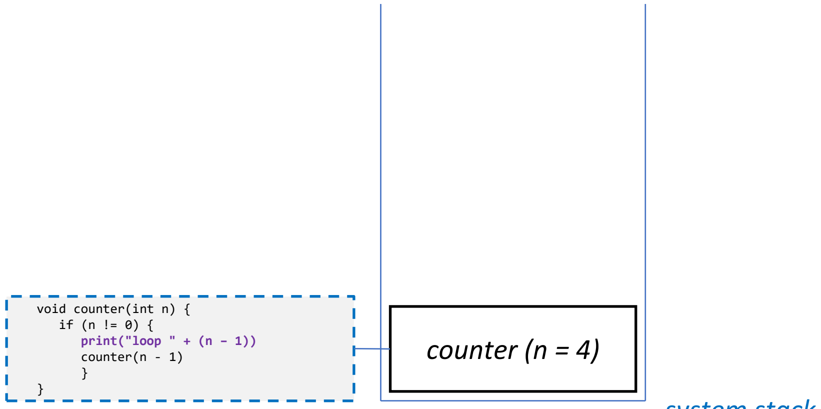
Order of execution

- The order in which we execute logic in a recursive function depends on if it's before / after the self-call

logic-before-recursion

```
void counter(int n) {
   if (n != 0) {
      print("loop " + (n – 1))
      counter(n - 1)
      }
}
```

```
loop 3
```



counter(4)

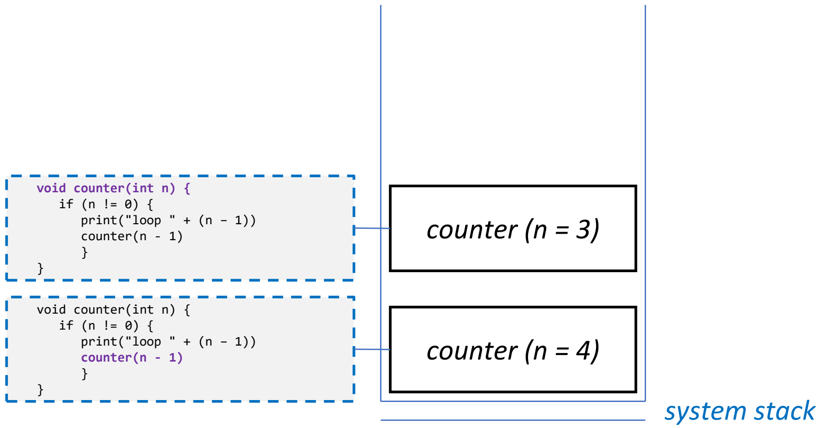
Order of execution

- The order in which we execute logic in a recursive function depends on if it's before / after the self-call

logic-before-recursion

```
void counter(int n) {
   if (n != 0) {
      print("loop " + (n – 1))
      counter(n - 1)
      }
}
```

```
loop 3
```



counter(4)

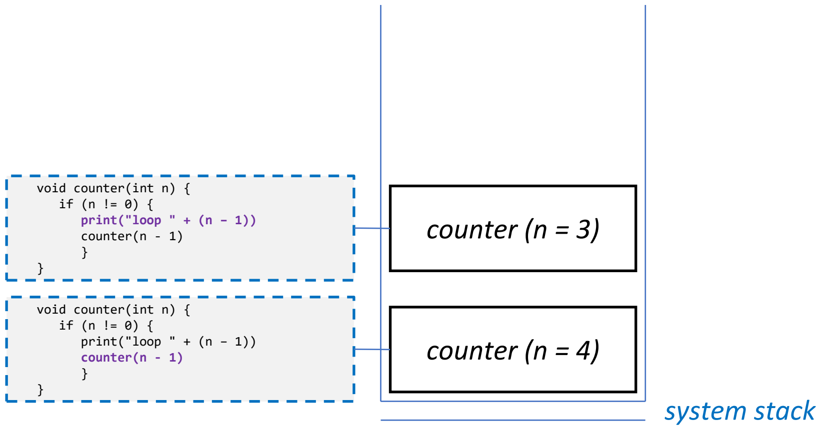
Order of execution

- The order in which we execute logic in a recursive function depends on if it's before / after the self-call

logic-before-recursion

```
void counter(int n) {
   if (n != 0) {
      print("loop " + (n – 1))
      counter(n - 1)
      }
}
```

```
loop 3
loop 2
```



counter(4)

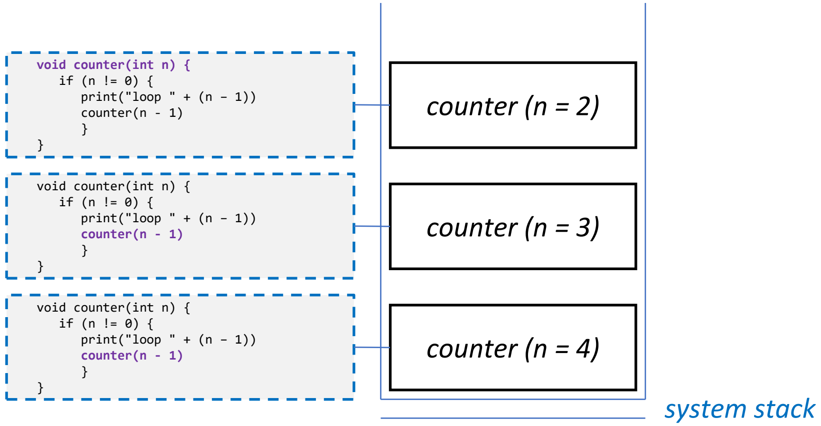
Order of execution

- The order in which we execute logic in a recursive function depends on if it's before / after the self-call

logic-before-recursion

```
void counter(int n) {
   if (n != 0) {
      print("loop " + (n – 1))
      counter(n - 1)
      }
}
```

```
loop 3
loop 2
```



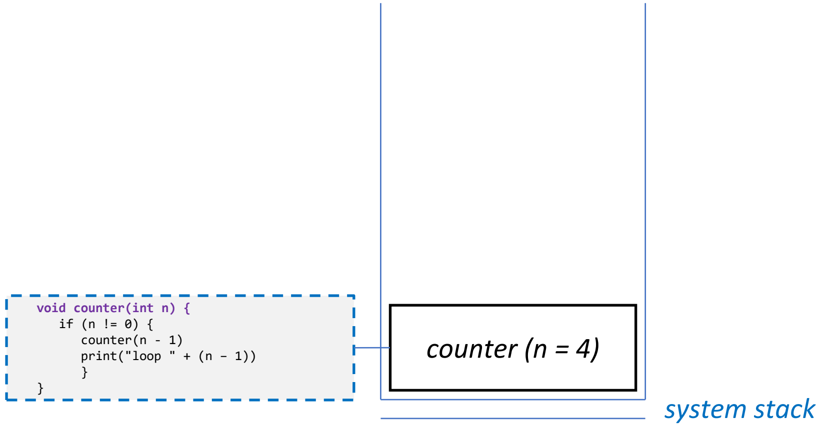
counter(4)

Order of execution

- The order in which we execute logic in a recursive function depends on if it's before / after the self-call

logic-after-recursion

```
void counter(int n) {
   if (n != 0) {
      counter(n - 1)
      print("loop " + (n – 1))
      }
}
```



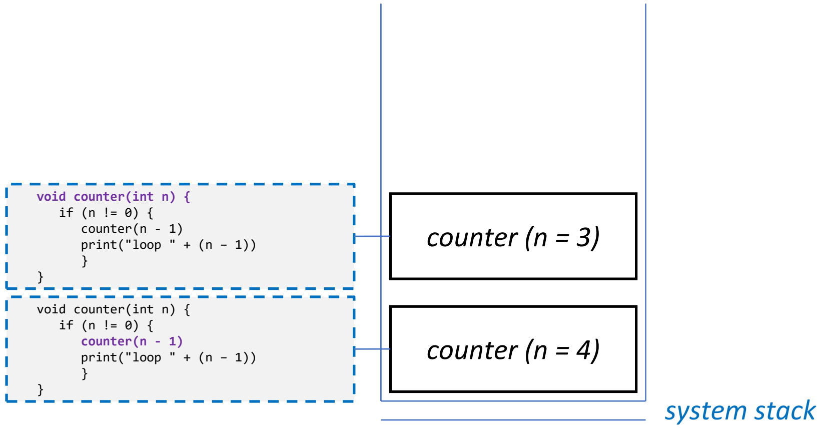
counter(4)

Order of execution

- The order in which we execute logic in a recursive function depends on if it's before / after the self-call

logic-after-recursion

```
void counter(int n) {
    if (n != 0) {
        counter(n - 1)
        print("loop " + (n – 1))
        }
}
```



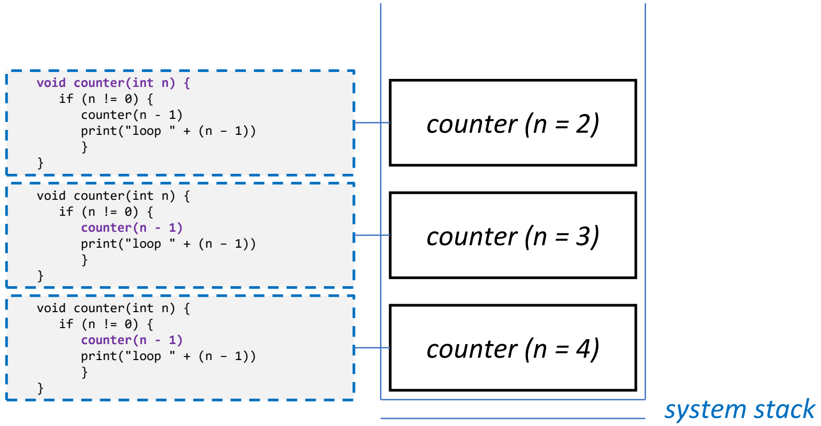
counter(4)

Order of execution

- The order in which we execute logic in a recursive function depends on if it's before / after the self-call

logic-after-recursion

```
void counter(int n) {
   if (n != 0) {
      counter(n - 1)
      print("loop " + (n – 1))
      }
}
```



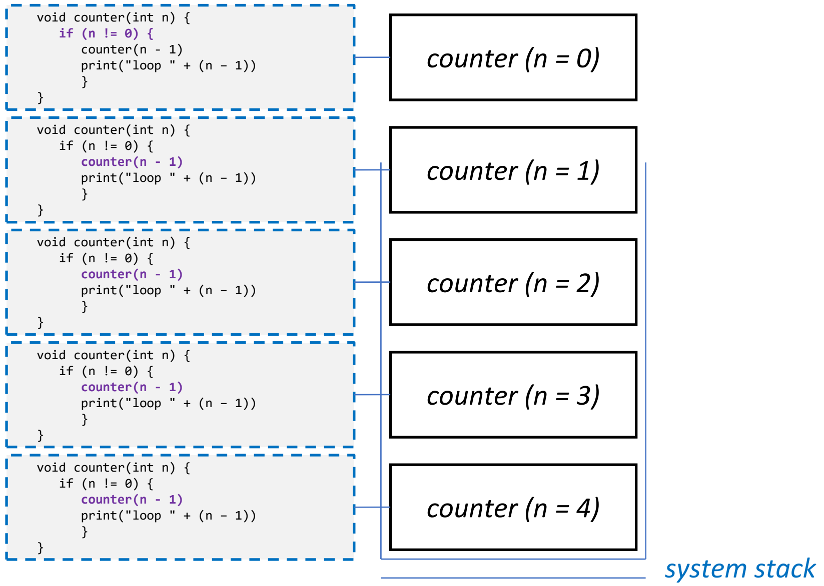
counter(4)

Order of execution

- The order in which we execute logic in a recursive function depends on if it's before / after the self-call

logic-after-recursion

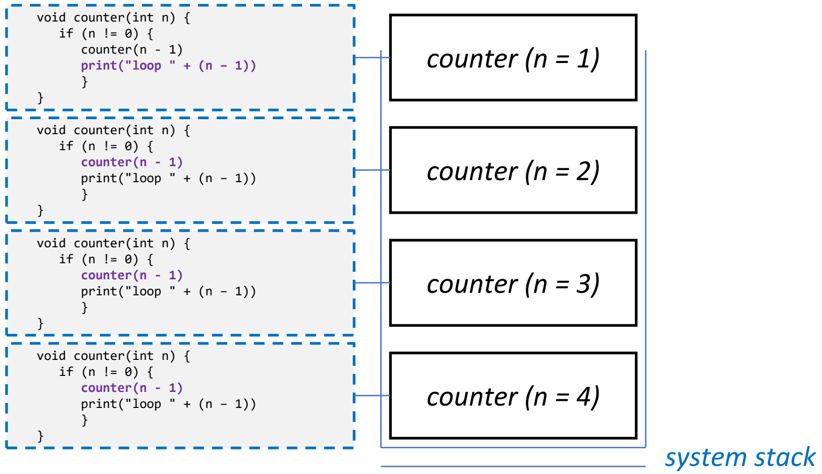
```
void counter(int n) {
   if (n != 0) {
      counter(n - 1)
      print("loop " + (n – 1))
      }
}
```



counter(4)

Order of execution

- The order in which we execute logic in a recursive function depends on if it's before / after the self-call



counter(4)

logic-after-recursion

```
void counter(int n) {
   if (n != 0) {
      counter(n - 1)
      print("loop " + (n – 1))
      }
}
```

```
loop 0
```

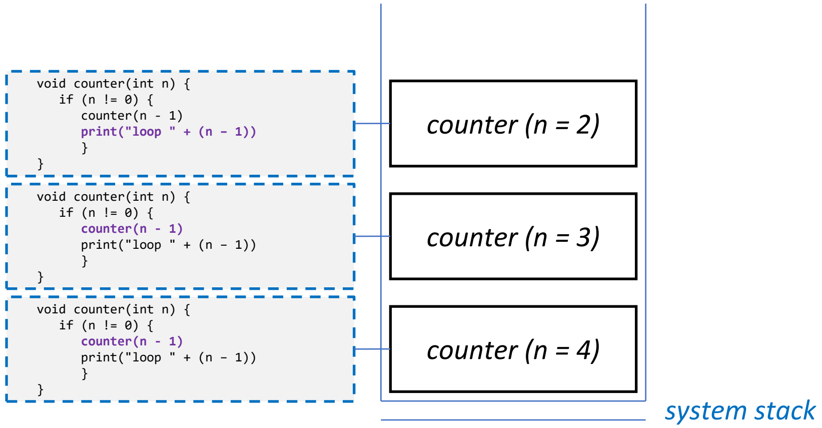
Order of execution

- The order in which we execute logic in a recursive function depends on if it's before / after the self-call

logic-after-recursion

```
void counter(int n) {
   if (n != 0) {
      counter(n - 1)
      print("loop " + (n – 1))
      }
}
```

```
loop 0
loop 1
```



counter(4)

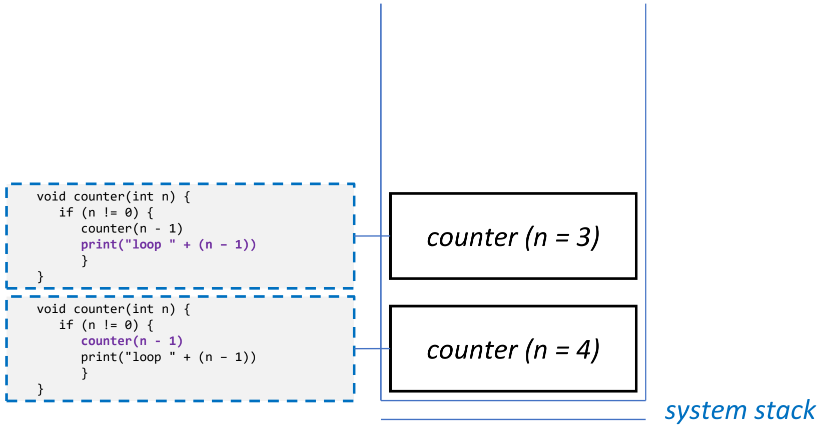
Order of execution

- The order in which we execute logic in a recursive function depends on if it's before / after the self-call

logic-after-recursion

```
void counter(int n) {
   if (n != 0) {
      counter(n - 1)
      print("loop " + (n – 1))
      }
}
```

```
loop 0
loop 1
loop 2
```



counter(4)

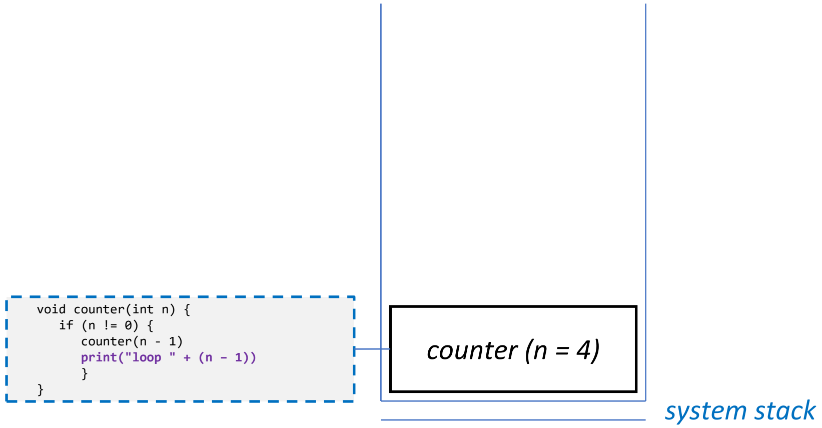
Order of execution

- The order in which we execute logic in a recursive function depends on if it's before / after the self-call

logic-after-recursion

```
void counter(int n) {
   if (n != 0) {
      counter(n - 1)
      print("loop " + (n – 1))
      }
}
```

```
loop 0
loop 1
loop 2
loop 3
```



counter(4)

Order of execution

- The order in which we execute logic in a recursive function depends on if it's before / after the self-call

logic-before-recursion

```
void counter(int n) {
    if (n != 0) {
        print("loop " + (n – 1))
        counter(n - 1)
        }
}
```

```
loop 3
loop 2
loop 1
loop 0
```

logic-after-recursion

```
void counter(int n) {
    if (n != 0) {
        counter(n - 1)
        print("loop " + (n – 1))
        }
}
```

```
loop 0
loop 1
loop 2
loop 3
```

Other reasons to choose...

iterative

```
void counter(int n) {
   for (int i = 0; i < n; i++) {
      print("loop!")
      }
   }
```

+ Memory usage is controlled explicitly by the programmer, so a "stack overflow“ less likely

+ Can executes more quickly, as there is no overhead from stack frame creation / destruction

- Naturally recursive functions can be harder to understand in an iterative form

recursive

```
void counter(int n) {
      if (n == 0) {
         return
         }
      print("loop!")
      counter(n - 1)
   }
```

+ Naturally recursive functions are much more concise when expressed this way

+ Languages which support tail recursion can eliminate some of the extra cost to performance, and stack overflows

Challenge 1

- Convert this **iterative** function to a **recursive** one
  - *Note: you are allowed to change the function's formal parameters*

```
int total(int array[]) {

   int result = 0

   for (int i = 0; i < array.length; i++)
      result += array[i]

   return result
}
```

Recursive Formulation

```
int total(int array[], int size) {
    if (size == 1)
        return array[size-1]
    return array[size-1] + total(array, size-1)
}
```

Say array = [5, 10, 15, 20]; size = 4

20 + total([5, 10, 15, 20], 3)

= 20 + (15 + total([5, 10, 15, 20], 2))

= 20 + (15 + (10 + total([5, 10, 15, 20], 1))

= 20 + (15 + (10 + (5)))

= 20 + (15 + (15))

= 20 + (30)

= 50

Factorial Computation

- Mathematically,
  - factorial(0) = 1
  - factorial(n) = n*factorial(n-1)*…*factorial(0)   *for* n = 1,2,…

```
//Recursive
int factorial(int n) {
   if (n == 0)
      return 1
   return n * factorial(n - 1)
}
```

```
//Iterative
int factorial(int n) {
    int total = 1
    for (int i = 1; i <= n; i++)
        total = total * i
    return total
}
```

More Difficult

- Convert this **recursive** function to an **iterative** one
  - *Note: you can use any ADT we've covered so far to help with this*

```
File {
   char name[]
   char path[]
   bool isDirectory
}
```

```
void printFiles(char dir[], int indent) {
     File f[] = getFileList(dir)
     for (int i = 0; i < f.length; i++) {
          printSpaces(indent)
          print(f.name)
          if (f.isDirectory)
               printFiles(f.path, indent + 3)
          }
     }
```

```
apple.txt
pictures
   img1.jpg
tv.zip
```

Solution (Details unimportant)

- Difficult because we need to keep track of the nested tree structure of the directory ourselves.
  - We need to *simulate* a call stack via the stack ADT

```
void printFiles(char dir[], int indent, Stack s) {
      int i = 0
      while (true) {
                  File f[] = getFileList(dir)
                  bool subScan = false
                  for (; i < f.length; i++) {
                        printSpaces(indent)
                        print(f.name)
                        if (f.isDirectory) {
                              s.push(new StackItem(dir, i, indent)
                              dir = f.path
                              i = 0
                              indent += 3
                              subScan = true
                              break
                              }
                        }

                  if (!subScan) {
                        if (s.peek()) {
                              StackItem si = s.pop()
                              dir = si.dir
                              i = dir.index + 1
                              indent = dir.indent
                              }
                              else {
                              break
                              }
                        }
            }
      }
```

```
StackItem {
   char dir[]
   int index
   int indent
}
```

Why was this one harder to write iteratively?

- We can classify problems as being singly-recursive or multiply-recursive

- A problem requires single-recursion if you only need to have one self-call for each function call; this kind of function is generally easy to write iteratively

```
int factorial(int n) {
   if (n == 0) return 1
   return n * factorial(n - 1)
 }
```

a single self-call

Why was this one harder to write iteratively?

- A problem requires multiple-recursion when you need to have more than one self-call for each function call; this kind of function is generally harder to write iteratively because we need to explicitly track state

a loop of self-calls

```
void printFiles(char dir[], int indent) {
      File f[] = getFileList(dir)
      for (int i = 0; i < f.length; i++) {
            printSpaces(indent)
            print(f.name)
            if (f.isDirectory)
                  printFiles(f.path, indent + 3)
            }
      }
```

Summary

- Any repeated-logic procedure can be written in an iterative or recursive form

- Which form to choose depends on the problem you are trying to solve, with varying elegance of expression, and varying performance characteristics
  - Singly-recursive problems are easy to write iteratively; multiply-recursive problems are harder to write iteratively

- Graphs and trees (coming later) feature extensive use of recursion in their analysis and traversal functions

Write binary search recursively

```
int lo = 0;
int hi = N-1;
int mid;
while (lo<=hi) {
      mid = (lo + hi)/2;
      if (A[mid]==X) return mid;
      if (X<A[mid]) hi = mid-1;
      else lo = mid+1;
 }
return -1  //not found
```