IT 179
15
Recursion

Have you read Chapter 5 (Recursion)?

No. Not at all

A few pages

The entire chapter

What is recurison?

What is Recursion?

The process in which a function calls itself directly or indirectly is called recursion

A function that calls itself (directly or indirectly) is called a recursive function.

Recursion

- Recursion can solve many programming problems that are difficult to conceptualize and solve linearly

- Recursive algorithms can
  - compute factorials
  - compute a greatest common divisor (GCD)
  - process data structures (strings, arrays, linked lists, etc.)
  - search efficiently using a binary search
  - find a path through a maze, and more

Recursive Thinking

Section 5.1

Recursive Thinking

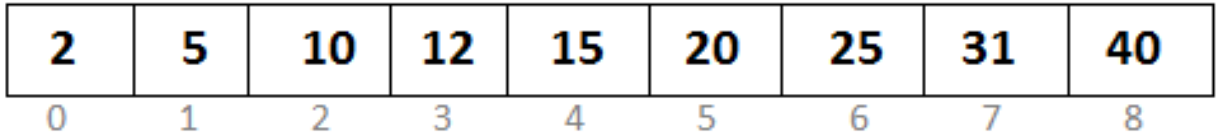
- Recursion is a problem-solving approach that can be used to generate simple solutions to certain kinds of problems that are difficult to solve by other means

- Recursion reduces a problem into one or more simpler versions of itself

Matryoshka dolls

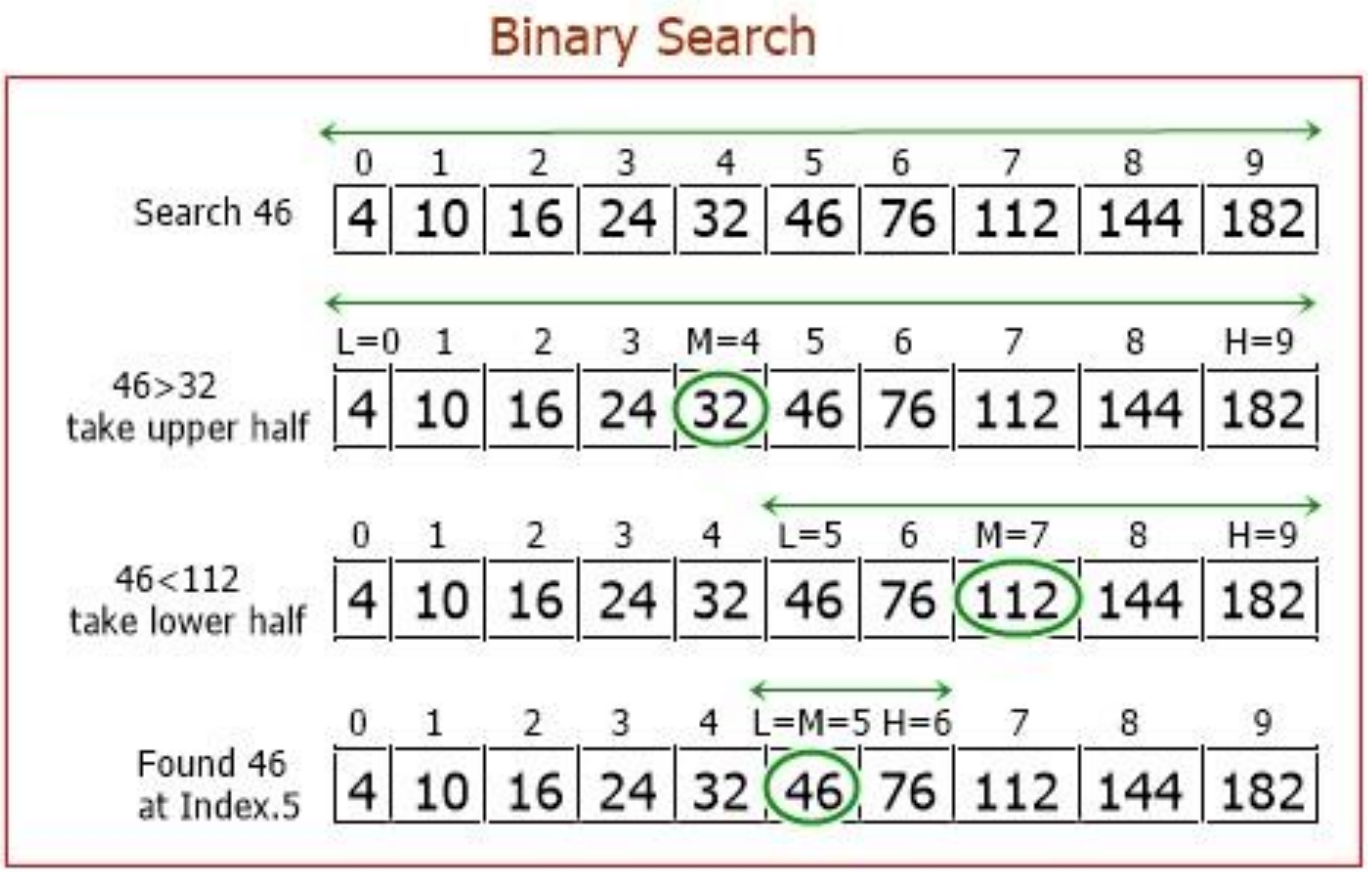
Recursive Thinking (cont.)

- Consider searching for a target value in an array
  - Assume the array elements are sorted in increasing order



  - We compare the target to the middle element and, if the middle element does not match the target, search either the elements before the middle element or the elements after the middle element
  - Instead of searching ***n*** elements, we search ***n*/2** elements

Recursive Thinking (cont.)



Recursive Thinking (cont.)

Recursive Algorithm to Search an Array

`if` the array is empty

    return -1 as the search result

`else if` the middle element matches the target

    return the subscript of the middle element as the result

`else if` the target is less than the middle element

    recursively search the array elements preceding the middle element and return the result

`else`

    recursively search the array elements following the middle element and return the result

Steps to Design a Recursive Algorithm

- There must be at least one case (the base case), for a small value of ***n***, that can be solved directly
- A problem of a given size *n* can be reduced to one or more smaller versions of the same problem (recursive case(s))
- Identify the base case(s) and solve it/them directly
- Devise a strategy to reduce the problem to smaller versions of itself while making progress toward the base case
- Combine the solutions to the smaller problems to solve the larger problem

Recursive Algorithm for Finding the Length of a String

`if` the string is empty (has no characters)

    the length is 0

`else`

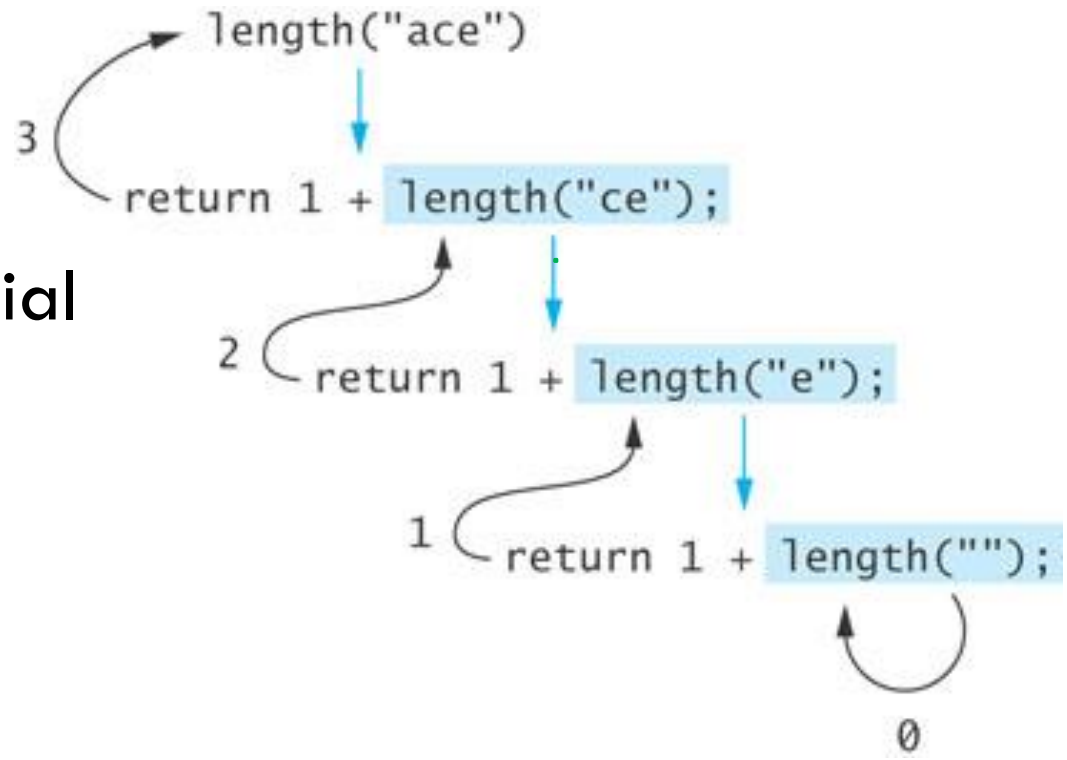
    the length is 1 plus the length of the string that excludes the first character

Recursive Algorithm for Finding the Length of a String (cont.)

```
/** Recursive method length
    @param str The string
    @return The length of the string
*/
public static int length(String str) {
  if (str == null || str.equals(""))
     return 0;
  else
     return 1 + length(str.substring(1));
}
```

Fig. 5.2 Tracing a Recursive Method

- The process of returning from recursive calls and computing the partial results is called *unwinding the recursion*



Two Examples to Warm Up

- Write a recursive method drawStar() that gives this output:

`drawStar(10)`:

```
**********
*********
********
*******
******
*****
****
***
**
*
```

Two Examples to Warm Up

- Write a recursive method **`printNumbers ()`** that gives this output:

`printNumbers(9)`:

```
9 9 9 9 9 9 9 9 9
8 8 8 8 8 8 8 8
7 7 7 7 7 7 7
6 6 6 6 6 6
5 5 5 5 5
4 4 4 4
3 3 3
2 2
1
```

Recursive Algorithm for Finding the Length of a String (cont.)

```
/** Recursive method length
    @param str The string
    @return The length of the string
*/
public static int length(String str) {
  if (str == null || str.equals(""))
     return 0;
  else
     return 1 + length(str.substring(1));
}
```

Recursive Algorithm for Printing String Characters

```
/** Recursive method printChars
    post: The argument string is
          displayed, one character per line
    @param str The string
*/
public static void printChars(String str) {
  if (str == null || str.equals(""))
    return;
  else {
    System.out.println(str.charAt(0));
    printChars(str.substring(1));
  }
}
```

```
printChars("Spring");


S
p
r
i
n
g
```

Recursive Algorithm for Printing String Characters

```
/** Recursive method printChars
    post: The argument string is
          displayed, one character per line
    @param str The string
*/
public static void printChars(String str) {
  if (str == null || str.equals(""))
    return;
  else {
     printChars(str.substring(1));
     System.out.println(str.charAt(0));
}
}
```

?

```
printChars("Spring");


g
n
i
r
p
S
```

Recursive Algorithm for Printing String Characters in Reverse

```
/** Recursive method printCharsReverse
    post: The argument string is displayed in reverse,
          one character per line
    @param str The string
*/
public static void printCharsReverse(String str) {
  if (str == null || str.equals(""))
    return;
  else {
    printCharsReverse(str.substring(1));
    System.out.println(str.charAt(0));
  }
}
```

What does the function fun1 () calculate?

```
static int fun1(int x, int y)
{
    if (x == 0)
        return y;
    else
        return fun1(x - 1, x + y);
}
```

fun1(4,7) = fun1(3 , 4+7)
= fun1(2, 3 + 4 + 7)
= fun1(1, 2 + 3 + 4 + 7)
= fun1(0, 1 + 2 + 3 + 4 + 7) = 17

What does the function `fun1()` calculate? - Generalize

```
static int fun1(int x, int y)
{
    if (x == 0)
        return y;
    else
        return fun1(x - 1, x + y);
}
```

fun1(x,y) = fun1(x-1 , x+y)

= fun1(x-2, (x-1) + x + y)

= fun1(x-3, (x-2) + (x-1) + x + y)

=…

=…

= fun1(0, 1 + 2 + … + (x-2) + (x-1) + x + y)

= 1 + 2 + … + (x-2) + (x-1) + x + y

= [x(x+1)/2] + y

Recursive Definitions of Mathematical Formulas

Section 5.2

Recursive Definitions of Mathematical Formulas

- Mathematicians often use recursive definitions of formulas that lead naturally to recursive algorithms
- Examples include:
  - factorials
  - powers
  - greatest common divisors (gcd)

Factorial of *n*: *n*!

- The factorial of *n*, or *n*! is defined as follows:

  $0! = 1$

  $n! = n \times (n-1)!$ $(n > 0)$

- The base case: *n* is equal to 0
- The second formula is a recursive definition

Factorial of *n*: *n*! (cont.)

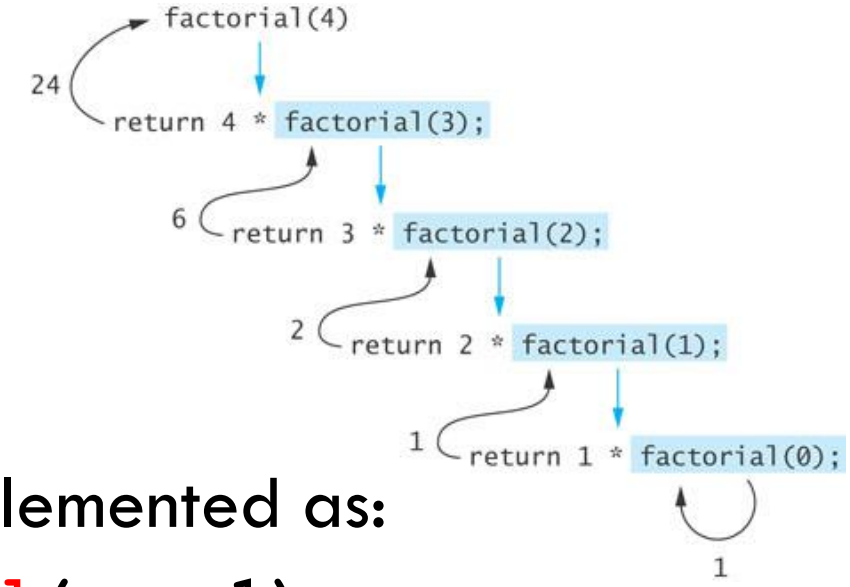
- The recursive definition can be expressed by the following algorithm:

  `if` *n* equals 0

  n! is 1

  `else`

  n! = *n* x (*n* – 1)!



- The last step can be implemented as:

```
return n * factorial(n - 1);
```

Recursive Algorithm for Calculating x^n

Recursive Algorithm for Calculating x^n ($n \geq 0$)

if *n is 0*

The result is 1

else

The result is $x \times x^{n-1}$

```
/** Recursive power method (in RecursiveMethods.java).
    pre: n >= 0
    @param x The number being raised to a power
    @param n The exponent
    @return x raised to the power n
*/
public static double power(double x, int n) {
    if (n == 0)
 return 1;
    else
 return x * power(x, n - 1);
}
```

Recursive Algorithm for Calculating gcd

- The greatest common divisor (gcd) of two numbers is the largest integer that divides both numbers
- The gcd of 20 and 15 is 5
- The gcd of 36 and 24 is 12
- The gcd of 38 and 18 is 2
- The gcd of 17 and 97 is 1

Recursive Algorithm for Calculating `gcd` (cont.)

- Given 2 positive integers m and n (m > n)

  `if` n is a divisor of m

  gcd(m, n) = n

  `else`

  gcd (m, n) = gcd (n, m % n)

gcd (38, 18) = gcd(18, 38 % 18) = gcd(18, 2) = 2

gcd (38, 16) = gcd(16, 38 % 16) = gcd(16, 6)
= gcd(6, 16 % 6) = gcd(6, 4)
= gcd(4, 6 % 4) = gcd(4, 2) = 2

Recursive Algorithm for Calculating gcd (cont.)

```
/** Recursive gcd method (in RecursiveMethods.java).
    pre: m > 0 and n > 0
    @param m The larger number
    @param n The smaller number
    @return Greatest common divisor of m and n
*/
public static double gcd(int m, int n) {
    if (m % n == 0)
 return n;
    else
 return gcd(n, m % n);
}
```

Recursion Versus Iteration

- There are similarities between recursion and iteration
- In iteration, a loop repetition condition determines whether to repeat the loop body or exit from the loop
- In recursion, the condition usually tests for a base case

- You can always write an iterative solution to a problem that is solvable by recursion

- A recursive algorithm may be simpler than an iterative algorithm and thus easier to write, code, debug, and read

Iterative `factorial` Method

```
/** Iterative factorial method.
    pre: n >= 0
    @param n The integer whose factorial is being computed
    @return n!
*/
public static int factorialIter(int n) {
    int result = 1;
    for (int k = 1; k <= n; k++)
 result = result * k;
    return result;
}
```

More Practice -1-

- Modify the previous recursive method **`printNumbers ()`** so that the output is:

`printNumbers(9)`:

```
1
2 2
3 3 3
4 4 4 4
5 5 5 5 5
6 6 6 6 6 6
7 7 7 7 7 7 7
8 8 8 8 8 8 8 8
9 9 9 9 9 9 9 9 9
```

More Practice -2-

- Modify the previous recursive method **`printNumbers ()`** so that the output is:

`printNumbers(9)`:

```
1
1 2
1 2 3
1 2 3 4
1 2 3 4 5
1 2 3 4 5 6
1 2 3 4 5 6 7
1 2 3 4 5 6 7 8
1 2 3 4 5 6 7 8 9
```

Problem Set 1

- Recursive sum of an array
- Count odd numbers in an array
- Recursive sum of digits of a number
- Find frequency of a word in a string
- Print 1 to n without using any loops
- Young Tableau

Consider the following recursive function fun(x, y). What is the value of fun(4, 3)?

```
int fun(int x, int y)
{
  if (x == 0)
    return y;
  return fun(x - 1,  x + y);
}
```

13

12

9

10

Answer:

(A)

What is the output of this program?

```c
#include <stdio.h>

int fun(int n)
{
    if (n == 4)
       return n;
    else return 2*fun(n+1);
}


int main()
{
   printf("%d ", fun(2));
   return 0;
}
```

4

8

16

Runtime Error

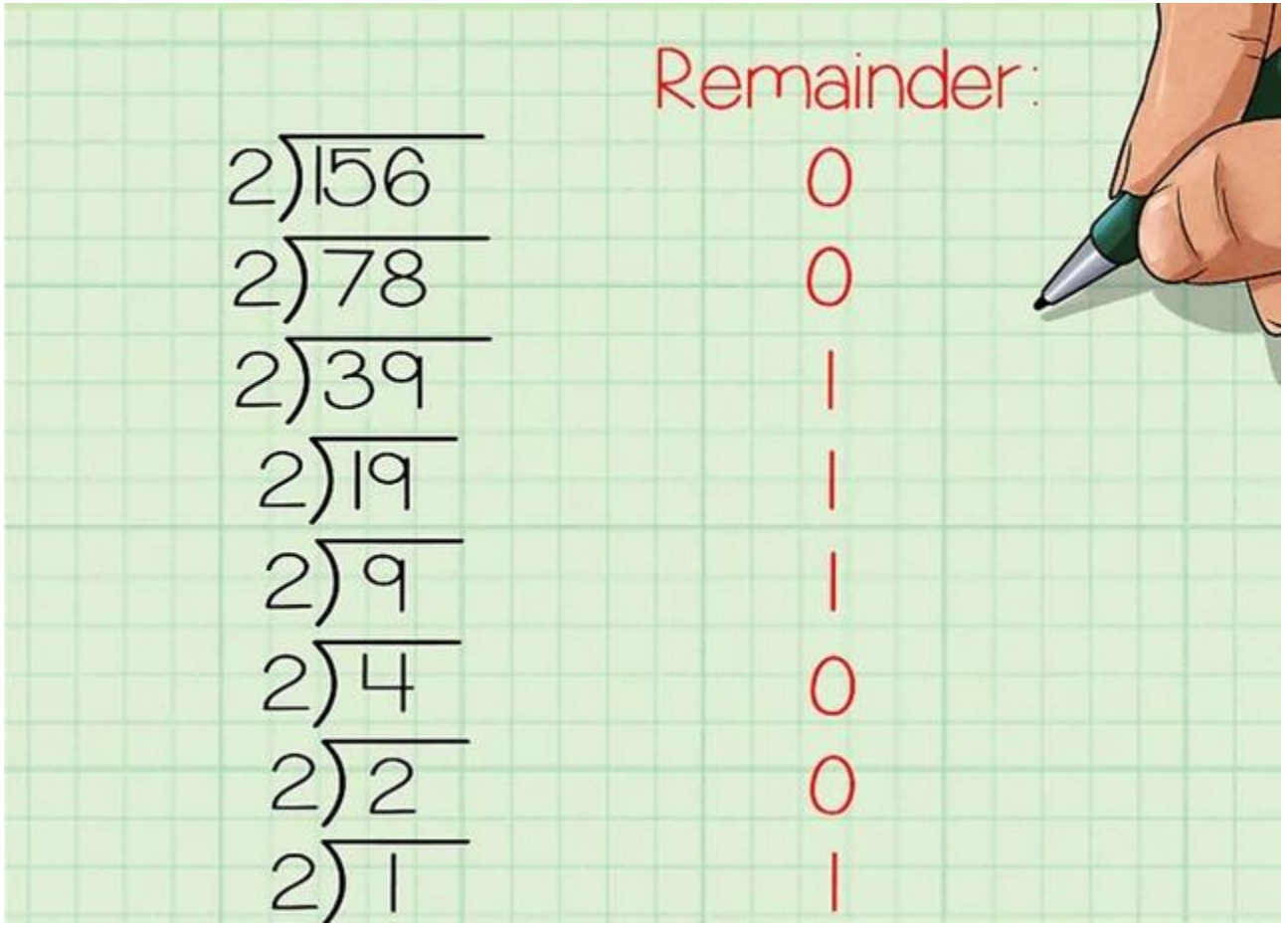
Answer:

(C)

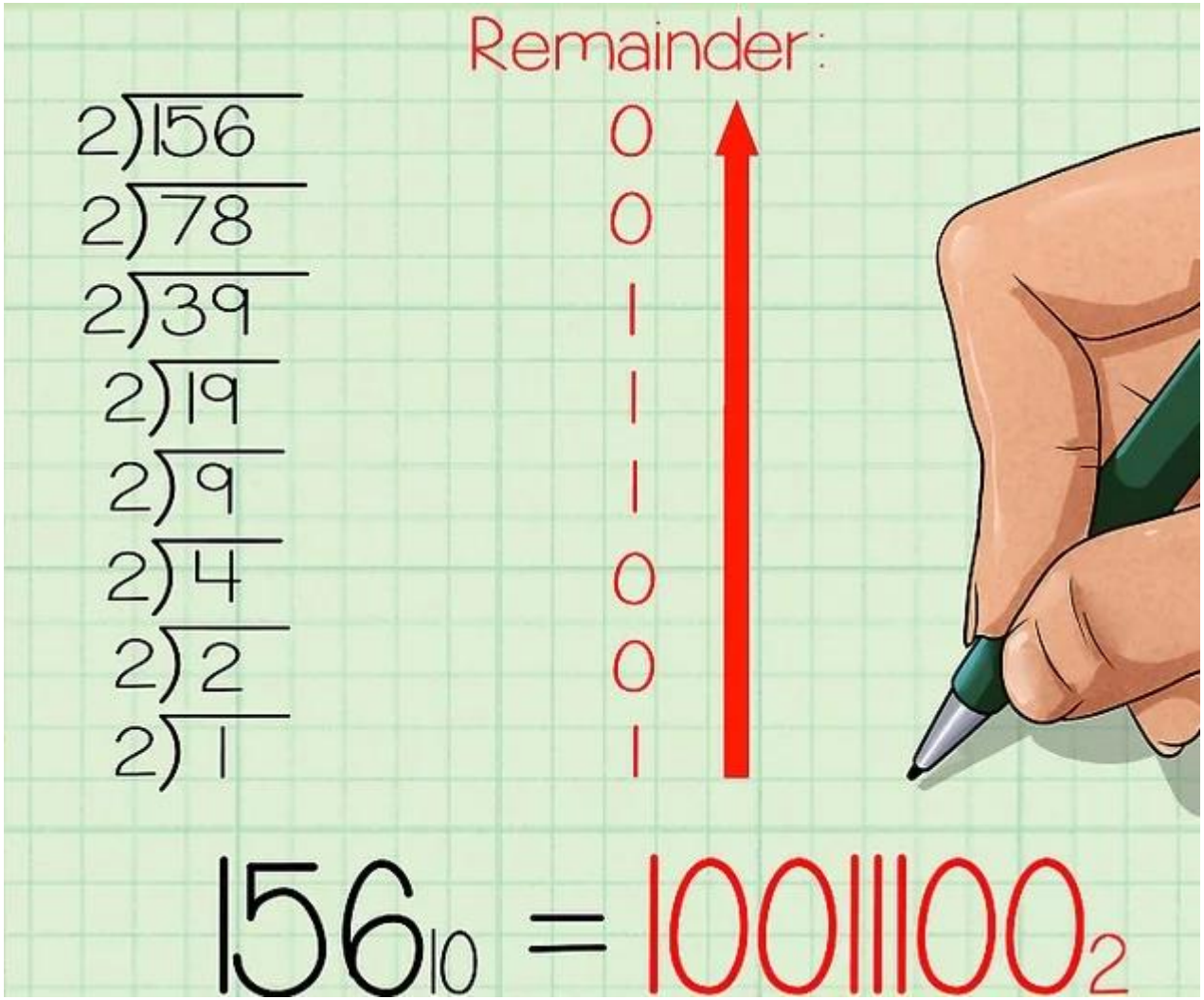
Problem Set 2

- Factorial (please read Section 5.2 in the textbook)
- Fibonacci (please read Section 5.2 in the textbook)
- Convert a decimal number into a binary number
- Convert a binary number into a decimal number
- Swap elements in an array
- Print out all permutations of a string
- Binary search

Convert a decimal number into a binary number



Convert a decimal number into a binary number



What does the following function print for n = 25?

```
void fun(int n)
{
  if (n == 0)
    return;

  printf("%d", n%2);
  fun(n/2);
}
```

- 11001
- 10011
- 11111
- 00000

Answer:

(B) The function mainly prints binary representation in reverse order.

Problem Set 2

- Convert a decimal number into a binary number
- Convert a binary number into a decimal number
- Swap elements in an array
- Print out all permutations of a string
- Binary search

Convert a binary number into a decimal number

convertBinToDec(11001) = 1 + 2* convertBinToDec(1100)

convertBinToDec(1100) = 0 + 2 * convertBinToDec(110)

convertBinToDec(110) = 0 + 2*convertBinToDec(11)

convertBinToDec(11) = 1 + 2*convertBinToDec(1)

convertBinToDec(1) = 1

Convert a binary number into a decimal number

convertBinToDec(11001) = 1 + 2* convertBinToDec(1100) = 25

convertBinToDec(1100) = 0 + 2 * convertBinToDec(110) = 12

convertBinToDec(110) = 0 + 2*convertBinToDec(11) = 6

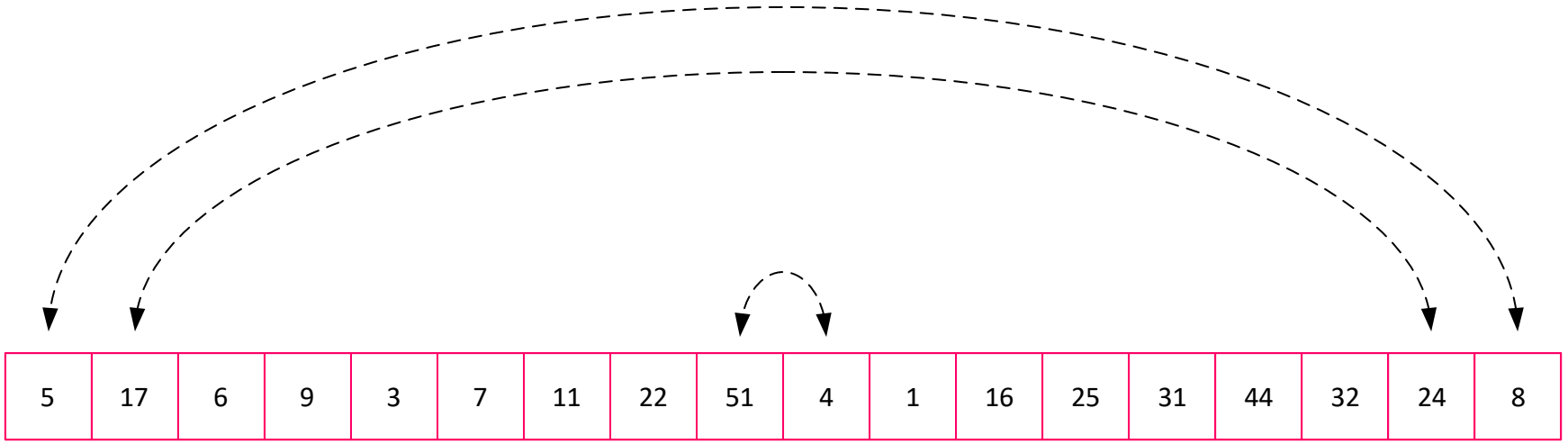
convertBinToDec(11) = 1 + 2*convertBinToDec(1) = 3

convertBinToDec(1) = 1

Problem Set 2

- Convert a decimal number into a binary number
- Convert a binary number into a decimal number
- Swap elements in an array
- Print out all permutations of a string
- Binary search

Swap elements in an array



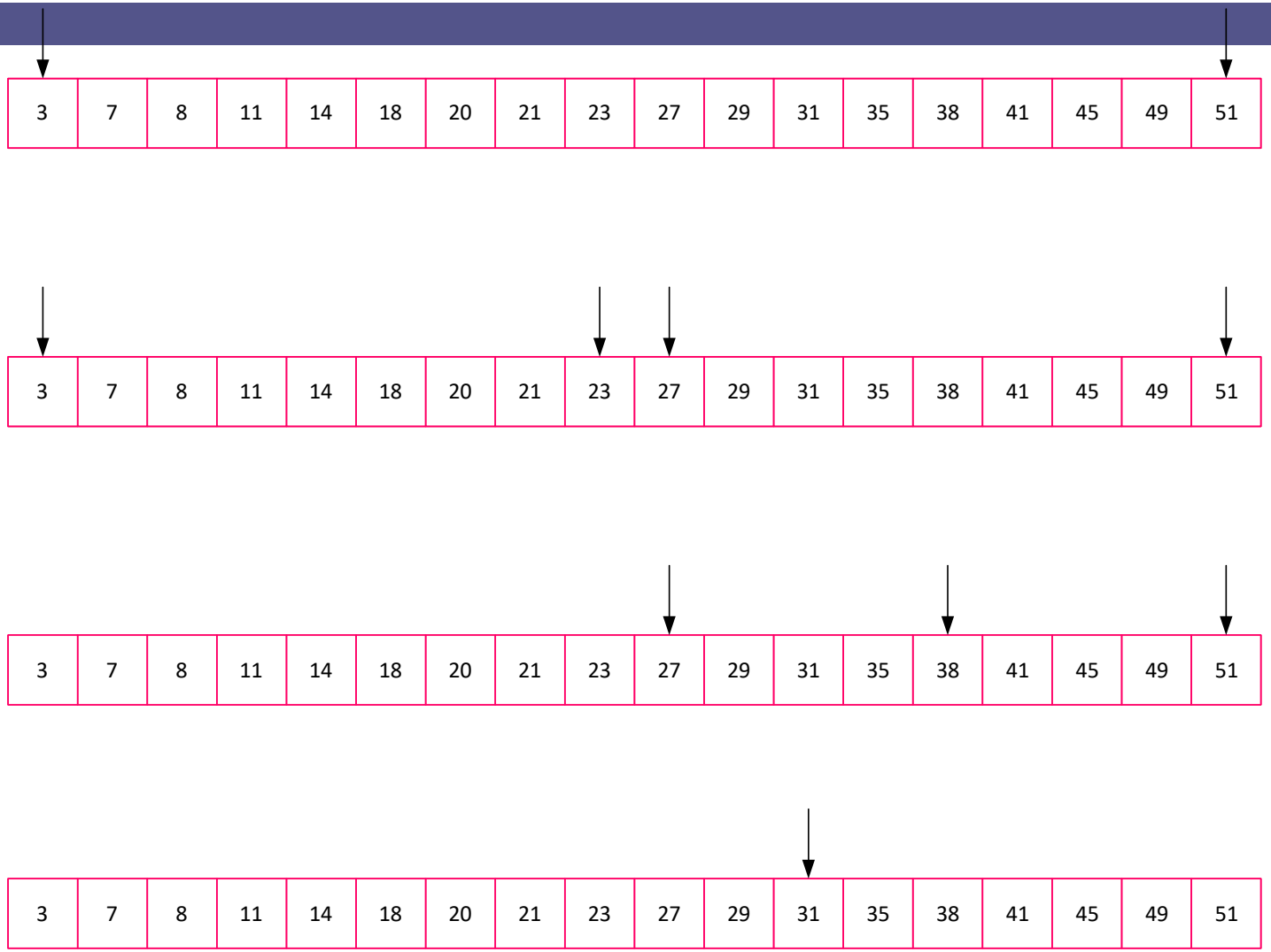
Problem Set 2

- Convert a decimal number into a binary number
- Convert a binary number into a decimal number
- Swap elements in an array
- Print out all permutations of a string
- ABCD
  - ABCD
  - ABDC
  - ACBD
  - ACDB

Problem Set 2

- Convert a decimal number into a binary number
- Convert a binary number into a decimal number
- Swap elements in an array
- Print out all permutations of a string
- Binary search

Binary search



Problem Set 3

- match()
  - match("([])") → true
  - match("({a[b]c})") → true
  - match("({[abc]]})") → false

- endX()
  - endX("xxre") → "rexx"
  - endX("xxhixx") → "hixxxx"
  - endX("xhixhix") → "hihixxx"