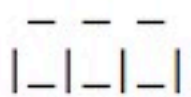
What output is generated by the program below?

```
def main() :
    showResult(10)

def showResult(value) :
    if value > 0 :
        print(value % 10, end="")
        showResult(value // 10)

main()
```

Consider the problem of arranging matchsticks so as to form a row of squares, as shown below.



Complete the recursive function below that is designed to return the number of matchsticks needed to form *n* squares.

```
def matchsticks(squares) :
    if squares == 1 :      # 1 square can be formed with 4 matchsticks
        return 4
    else :
        return _____________________________
```

Consider the function below, which implements the exponentiation operation recursively. What is the output of the function call `power(2, 3)`?

```
def power(base, exponent) :
    if exponent == 0 :
        result = 1.0
    else :
        result = base * power(base, exponent - 1)
    print(result + "", end="")
    return result
```

Recursive Helper functions

- Sometimes it is easier to find a recursive solution if you make a slight change to the original problem.
- Consider the palindrome test of previous section.
- It is a bit inefficient to construct new string objects in every step.

Substring Palindromes (1)

- Rather than testing whether the sentence is a palindrome, check whether a substring is a palindrome:

```
## Recursively tests whether a substring is
# a palindrome.
# @param text a string that is being checked
# @param start the index of the first character of the substring
# @param end the index of the last character of the substring
# @return True if the substring is a palindrome
#
def substringIsPalindrome(text, start, end) :
```

Substring Palindromes (2)

- Then, simply call the helper function with positions that test the entire string:

```
def isPalindrome(text) :
    return substringIsPalindrome(text, 0, len(text) - 1)
```

Recursive Helper function

```
def substringIsPalindrome(text, start, end) :
    # Separate case for substrings of length 0 and 1.
    if start >= end :
        return True
    else :
    # Get first and last characters, converted to lowercase.
        first = text[start].lower()
        last = text[end].lower()
```

Continued

Recursive Helper Function

```
if first.isalpha() and last.isalpha() :
    if first == last :
        # Test substring that doesn't contain the matching
        # letters.
        return substringIsPalindrome
           (text, start + 1, end - 1)
    else :
        return False
elif not last.isalpha() :
    # Test substring that doesn't contain the last character.
    return substringIsPalindrome(text, start, end - 1)
else :
    # Test substring that doesn't contain the first
    # character.
    return substringIsPalindrome(text, start + 1, end)
```

11.4 The Efficiency of Recursion

- Fibonacci sequence:
  Sequence of numbers defined by

$$f_1 = 1$$
$$f_2 = 1$$
$$f_n = f_{n-1} + f_{n-2}$$

- First ten terms:

1, 1, 2, 3, 5, 8, 13, 21, 34, 55

Recursivefib.py

```
1  ##
2  #  This program computes Fibonacci numbers using a recursive function.
3  #
4
5  def main() :
6     n = int(input("Enter n: "))
7     for i in range(1, n + 1) :
8        f = fib(i)
9        print("fib(%d) = %d" % (i, f))
10
11 ## Computes a Fibonacci number.
12 #  @param n an integer
13 #  @return the nth Fibonacci number
14 #
15 def fib(n) :
16    if n <= 2 :
17       return 1
18    else :
19       return fib(n - 1) + fib(n - 2)
20
21 # Start the program.
22 main()
```

Program Run

```
Enter n: 50
fib(1) = 1
fib(2) = 1
fib(3) = 2
fib(4) = 3
fib(5) = 5
fib(6) = 8
fib(7) = 13
. . .
fib(50) = 12586269025
```

Efficiency of Recursion

- Recursive implementation of `fib()` is straightforward.
- Watch the output closely as you run the test program.
- First few calls to `fib()` are quite fast.
- For larger values, the program pauses an amazingly long time between outputs.
- To find out the problem, let's insert **trace messages**.

Recursivefibtracer.py: 1

```
 1  ##
 2  #  This program prints trace messages that show how often the
 3  #  recursive function for computing Fibonacci numbers calls itself.
 4  #
 5
 6  def main() :
 7     n = int(input("Enter n: "))
 8     for i in range(1, n + 1) :
 9        f = fib(i)
10        print("fib(%d) = %d" % (i, f))
11
```

```
24
25  # Start the program.
26  main()
```

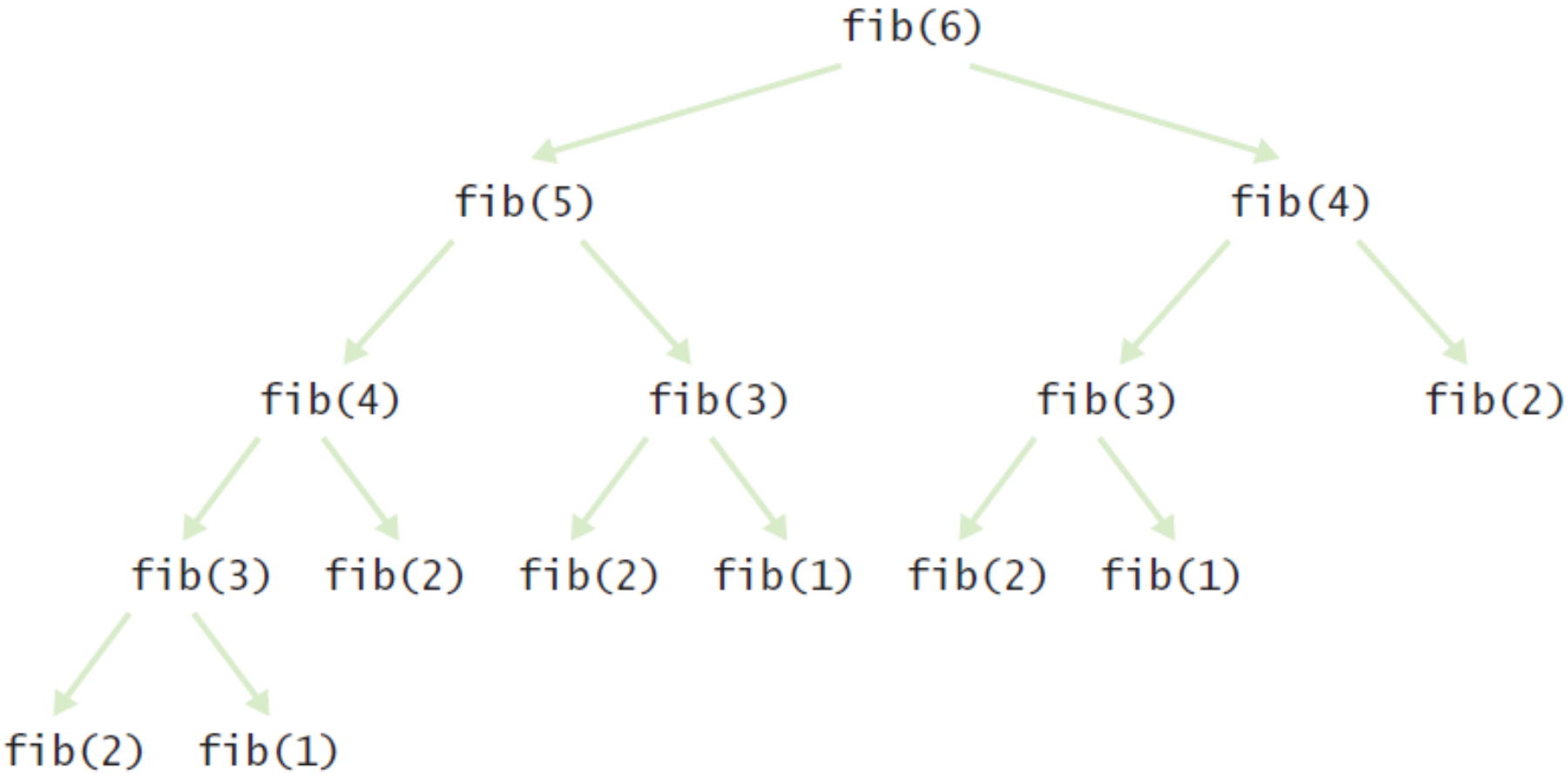
Recursivefibtracer.py: 2

```
12  ## Computes a Fibonacci number.
13  #  @param n an integer
14  #  @return the nth Fibonacci number
15  #
16  def fib(n) :
17     print("Entering fib: n =", n)
18     if n <= 2 :
19        f = 1
20     else :
21        f = fib(n - 1) + fib(n - 2)
22     print("Exiting fib: n =", n, "return value =", f)
23     return f
```

Program Run

```
Enter n: 6
Entering fib: n = 6
Entering fib: n = 5
Entering fib: n = 4
Entering fib: n = 3
Entering fib: n = 2
Exiting fib: n = 2 return value = 1
Entering fib: n = 1
Exiting fib: n = 1 return value = 1
Exiting fib: n = 3 return value = 2
Entering fib: n = 2
Exiting fib: n = 2 return value = 1
Exiting fib: n = 4 return value = 3
Entering fib: n = 3
Entering fib: n = 2
Exiting fib: n = 2 return value = 1
Entering fib: n = 1
Exiting fib: n = 1 return value = 1
Exiting fib: n = 3 return value = 2
```

Call Pattern of Recursive `Fib()` Function



Efficiency of Recursion

- The function takes so long because it computes the same values over and over.
- Computation of `fib(6)` calls `fib(3)` three times.
- Imitate the pencil-and-paper process to avoid computing the values more than once.

Efficiency of Recursion

- Occasionally, a recursive solution runs much slower than its iterative counterpart.
- In most cases, the recursive solution is only slightly slower.
- The iterative `isPalindrome()` performs only slightly better than recursive solution.
  - *Each recursive function call takes a certain amount of processor time*

Loopfib.py (1)

```
1  ##
2  #  This program computes Fibonacci numbers using an iterative function.
3  #
4
5  def main() :
6     n = int(input("Enter n: "))
7     for i in range(1, n + 1) :
8        f = fib(i)
9        print("fib(%d) = %d" % (i, f))
10
```

Loopfib.py (2)

```
11  ## Computes a Fibonacci number.
12  #  @param n an integer
13  #  @return the nth Fibonacci number
14  #
15  def fib(n) :
16     if n <= 2 :
17        return 1
18     else :
19        olderValue = 1
20        oldValue = 1
21        newValue = 1
22        for i in range(3, n + 1) :
23           newValue = oldValue + olderValue
24           olderValue = oldValue
25           oldValue = newValue
26
27        return newValue
28
29  # Start the program.
30  main()
```

Program Run

```
Enter n: 50
fib(1) = 1
fib(2) = 1
fib(3) = 2
fib(4) = 3
fib(5) = 5
fib(6) = 8
fib(7) = 13
. . .
fib(50) = 12586269025
```

Efficiency of Recursion

- Smart compilers can avoid recursive function calls if they follow simple patterns.
- Most compilers don't do that
- In many cases, a recursive solution is easier to understand and implement correctly than an iterative solution .
- '*To iterate is human, to recurse divine.*'
  - L. Peter Deutsch

Iterative `IsPalindrome()` Function

```
def isPalindrome(text) :
    start = 0
    end = len(text) - 1
    while start < end :
        first = text[start].lower()
        last = text[end].lower()
        if first.isalpha() and last.isalpha() :
            # Both are letters.
            if first == last :
                start = start + 1
                end = end - 1
            else :
                return False
        if not last.isalpha()
            end = end - 1
```

11.5 Permutations

- Design a class that will list all permutations of string, where a permutation is a rearrangement of the letters
- The string "eat" has six permutations:
  - "eat"
  - "eta"
  - "aet"
  - "ate"
  - "tea"
  - "tae"

Generate All Permutations (1)

- Generate all permutations that start with 'e', then 'a', then 't'
- The string "eat" has six permutations:
  - "eat"
  - "eta"
  - "aet"
  - "ate"
  - "tea"
  - "tae"

Generate All Permutations (2)

- Generate all permutations that start with 'e', then 'a', then 't'
- To generate permutations starting with 'e', we need to find all permutations of "at"
- This is the same problem with simpler inputs
- Use recursion

Implementing `Permutations()` Function

- Loop through all positions in the word to be permuted
- For each of them, compute the shorter word obtained by removing the i^{th} letter:

```
shorter = word[ : i] + word[i + 1 : ]
```

- Compute the permutations of the shorter word:

```
shorterPermutations = permutations(shorter)
```

Implementing `Permutations()` Function

- Add the removed letter to the front of all permutations of the shorter word:

```
for s in shorterPermutations :
    result.append(word[i] + s)
```

- Special case for the simplest string, the empty string, which has a single permutation - itself

Permutations.py (1)

```
1  ##
2  #  This program computes permutations of a string.
3  #

4
5  def main() :
6     for string in permutations("eat") :
7        print(string)
8

36
37 # Start the program.
38 main()
```

Permutations.py (2)

```
 9  ## Gets all permutations of a given word.
10  #  @param word the string to permute
11  #  @return a list of all permutations
12  #
13  def permutations(word) :
14     result = []
15
16     # The empty string has a single permutation: itself.
17     if len(word) == 0 :
18        result.append(word)
19        return result
20     else :
21        # Loop through all character positions.
22        for i in range(len(word)) :
23           # Form a shorter word by removing the ith character.
24
29           # Add the removed character to the front of each permutation
30           # of the simpler word.
31           for string in shorterPermutations :
32              result.append(word[i] + string)
33
34        # Return all permutations
35        return result
```

Program Run

```
eat
eta
aet
ate
tea
tae
```

Backtracking

- Backtracking examines partial solutions, abandoning unsuitable ones and returning to consider other candidates
- Can be used to
  - solve crossword puzzles
  - escape from mazes
  - find solutions to systems that are constrained by rules

Backtracking Characteristic Properties

1. A procedure to examine a partial solution and determine whether to:
    I. accept it as an actual solution or,
    II. abandon it (because it either violates some rules or can never lead to a valid solution)
2. A procedure to extend a partial solution, generating one or more solutions that come closer to the goal

Recursive Backtracking Algorithm

```
Solve(partialSolution)
     Examine(partialSolution).
     If accepted
          Add partialSolution to the list of solutions.
     Else if not abandoned
          For each p in extend(partialSolution)
              Solve(p)
```

Eight Queens Problem

- **Problem:** position eight queens on a chess board so that none of them attacks another according to the rules of chess
- A solution: