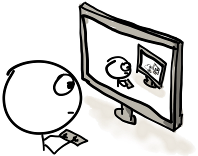
COMP 1023 Introduction to Python Programming

Recursion

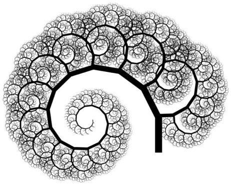
Dr. James Cheung, Dr. Wilfred Ng, Dr. Desmond Tsoi

Department of Computer Science & Engineering
The Hong Kong University of Science and Technology, Hong Kong SAR, China

Introduction



- Recursion is a technique that leads to elegant solutions for problems that are difficult to solve using simple loops.
- It is the process of defining a solution to a problem in terms of a simpler version of itself.
- Many natural phenomena exhibit recursion.
- Recursive functions are functions that call themselves. They consist of two parts: the base case(s) and the recursive case(s).
  - The base cases are the conditions that stop the recursion.
  - The recursive cases are the parts in which the function calls itself.



The Handshake Problem

Problem Statement

There are n people in the room. If each person shakes hands once with every other person, what will the total number of handshakes be?

- 1 person (A): 0 handshakes.
- 2 people (A and B): 1 handshake.
  - A 🤝 B
- 3 people (A, B, and C):
  - A 🤝 B, A 🤝 C, B 🤝 C
- 4 people (A, B, C, and D):
  - A 🤝 B, A 🤝 C, A 🤝 D
  - B 🤝 C, B 🤝 D, C 🤝 D

The Handshake Problem

- There is a trick to find the total number of handshakes:
  - If there are 2 people, there is only 1 handshake.
  - If there are 3 people, treat it as having 1 more person added to the 2 people, resulting in 2 extra handshakes.
  - If there are 4 people, treat it as having 1 more person added to the 3 people, resulting in 3 extra handshakes.
- You can generalize the total number into a formula.
  Let $h(n)$ be the total number of handshakes among n people:
  - $h(1) = 0$ (Base case)
  - $h(2) = h(1) + 1 = 0 + 1 = 1$
  - $h(3) = h(2) + 2 = 1 + 2 = 3$
  - $h(4) = h(3) + 3 = 3 + 3 = 6$

    $\ldots$
  - $h(n) = h(n-1) + (n-1)$ (General case)

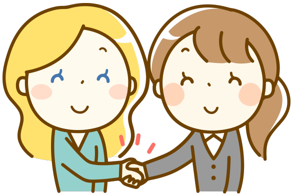
$$h(n) = \begin{cases} h(n-1) + (n-1), & \text{if } n \geq 2 \\ 0, & \text{otherwise} \end{cases}$$

The Handshake Problem

```python
# Filename: handshake.py

def handshake(n: int) -> int:
    if n <= 1:                      # Base case
        return 0                    # No one to shake hands
    return handshake(n-1) + (n-1)   # General case

def main() -> None:
    for i in range(5):
        print(f"{i} Person: {handshake(i)} handshake(s)")

if __name__ == "__main__":
    main()
```

Output:

```
0 Person: 0 handshake(s)
1 Person: 0 handshake(s)
2 Person: 1 handshake(s)
3 Person: 3 handshake(s)
4 Person: 6 handshake(s)
```



General Structure of Recursive Function

- The recursive function is implemented using an `if` or an `if-else` statement that leads to different cases.
- One or more base cases (the simplest cases) are used to stop recursion.
- Every recursive call reduces the original problem, bringing it increasingly closer to a base case until it becomes that case.

```
def <recursive_function_name>(<parameter list>):
    # Base case: condition to stop recursion
    if <base_case_condition>:
        return <base_case_value>

    # General case: call the function itself with modified parameters
    return <recursive_function_name>(modified_parameters)
```

Factorial Problem

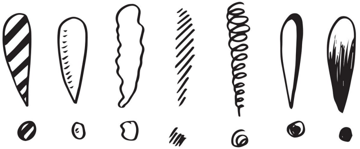
- The factorial of n is defined as $n! = n \times (n-1)!$, where n is a non-negative integer.
- Let $f(n)$ represent $n!$; it is computed as follows:

$$f(n) = \begin{cases} n \times f(n-1), & \text{if } n > 0 \\ 1, & \text{if } n = 0 \end{cases}$$

```python
# Filename: factorial_recursive.py
def factorial(n: int) -> int:
    if n == 0:                    # Base case
        return 1
    return n * factorial(n - 1)   # General case

def main() -> None:
    for i in range(6):
        print(f"{i}! = {factorial(i)}")

if __name__ == "__main__": main()
```

Output:

```
0! = 1
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
```



Factorial Problem - Demonstration

```python
def factorial(n: int) -> int:
    if n == 0:                    # Base case
        return 1
    return n * factorial(n - 1)  # General case
```

```
factorial(3):
  3 == 0 ?                          No
  fac(3) = 3 * fac(2)
    fac(2):
      2 == 0 ?                      No
      fac(2) = 2 * fac(1)
        fac(1):
          1 == 0 ?                  No
            fac(1) = 1 * fac(0)
              fac(0):
              0 == 0 ?              Yes
              return 1
            fac(1) = 1 * 1
            return 1
      fac(2) = 2 * 1 = 2
      return 2
  fac(3) = 3 * 2 = 6
  return 6

  fac(3) has the value 6
```

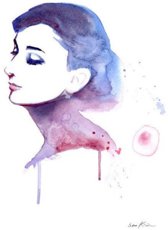
Factorial Problem

- It is simpler and more efficient to implement the factorial function using a loop. However, we use the recursive factorial function here to demonstrate the concept of recursion.

```python
def factorial(n: int) -> int:
    if n == 0:                    # Base case
        return 1
    return n * factorial(n - 1)   # General case
```

```python
def factorial(n: int) -> int:
    product: int = 1
    while n > 0:
        product *= n
        n -= 1
    return product
```

- For certain problems, a recursive solution often leads to short and elegant code.



Fibonacci's Bunnies Problem

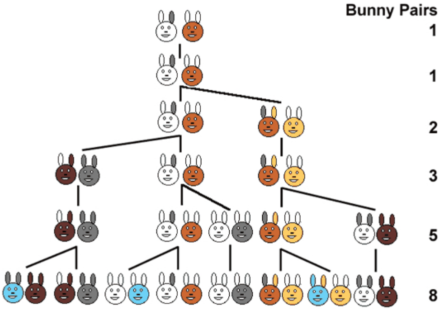
- Each new pair of bunnies becomes fertile at the age of one month.
- Each pair of fertile bunnies produces a new pair of offspring every month.
- None of the bunnies die during this time.

- Initially, God creates a pair of bunnies (let's call them A and B). **[1 pair]**
- After 1 month, the pair of bunnies becomes fertile. **[1 pair]**
- After 2 months, A and B produce a new pair of bunnies (let's call them C and D). So there will be 2 pairs of bunnies (A, B and C, D). **[2 pairs]**
- After 3 months, A and B produce a new pair of bunnies (let's call them E and F), and C and D become fertile. So there will be 3 pairs of bunnies (A, B; C, D; and E, F). **[3 pairs]**
- After 4 months, A and B produce a new pair of bunnies (let's call them G and H), C and D produce a new pair of bunnies (let's call them I and J), and E and F become fertile. So there will be 5 pairs of bunnies (A, B; C, D; E, F; G, H; and I, J). **[5 pairs]**

Question

After n months, how many pairs of bunnies are there?

Fibonacci's Bunnies



- The sequence (1, 1, 2, 3, 5, 8, ...) is called the Fibonacci sequence, proposed by Leonardo Fibonacci in 1202 in *The Book of the Abacus*. It is believed to model nature to a certain extent, such as Kepler's observation of leaves and flowers in 1611.
- Fibonacci sequence:

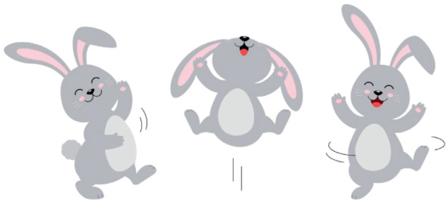
$$1, 1, 2, 3, 5, 8, 13, 21, 34, \ldots$$

where each number is the sum of the two preceding numbers.

Fibonacci's Bunnies Problem

- Recursive definition:
  Let $F(n)$ be the number of pairs of bunnies after n months; it is computed as follows:

$$F(n) = \begin{cases} 1 & \text{if } n = 0 \text{ (Number of pairs of bunnies after 0 months)} \\ 1 & \text{if } n = 1 \text{ (Number of pairs of bunnies after 1 month)} \\ F(n-1) + F(n-2) & \text{if } n \geq 2 \end{cases}$$



Fibonacci's Bunnies

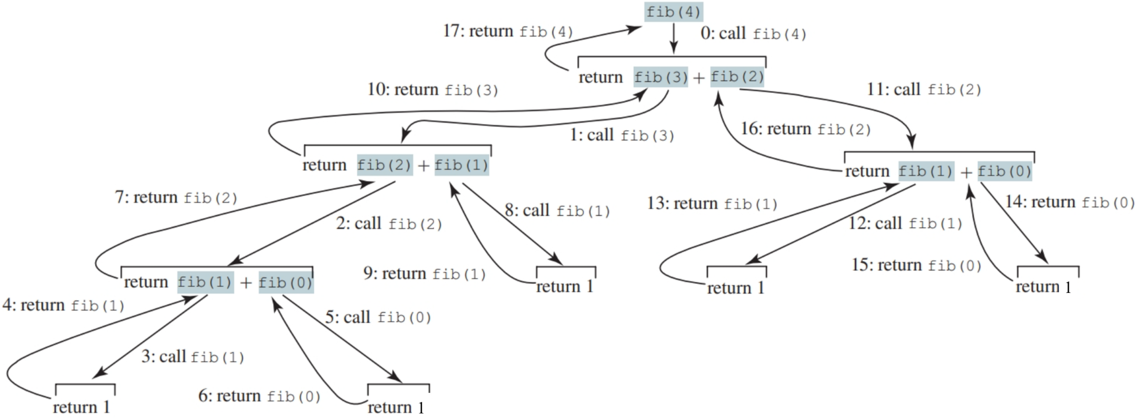
```python
# Filename: fib.py
def fib(n: int) -> int:
    if n == 0 or n == 1:        # Base cases
        return 1
    return fib(n - 1) + fib(n - 2)  # Recursive case

def main() -> None:
    m: int = int(input("Enter month: "))
    num_bunnies: int = fib(m)
    print(f"The number of pairs of bunnies after {m} \
          month(s) is equal to {num_bunnies} pair(s)")

if __name__ == "__main__": main()
```

Output:

```
Enter month: 5
The number of pairs of bunnies after 5 month(s) is equal to 8 pair(s)

Enter month: 10
The number of pairs of bunnies after 10 month(s) is equal to 89 pair(s)

Enter month: 20
The number of pairs of bunnies after 20 month(s) is equal to 10946 pair(s)
```

Fibonacci's Bunnies

- Calculating fib(4) using recursion:



Many intermediate steps are re-calculated.

Recursion and Loop

- Both recursion and looping are used to perform repetitive tasks in programming.
- They can be employed to solve the same problems, often yielding the same results, though the approach and implementation differ.
- Comparison:

	Loop	**Recursion**
Initialization	Initialize variables before the loop.	Pass initial values as parameters.
Condition	Check a condition at the beginning or end of each iteration.	Base case checks to stop further calls.
Iteration/ Recursion	Update variables within the loop.	Call the function with updated parameters.
Termination	Exit when the condition is false.	Return a value when the base case is reached.

Recursive Stack Overflow and Its Handling

- Stack overflow occurs when a recursive function exceeds the maximum call stack size.
- Can lead to a 'RecursionError' in Python.
- Strategies to Handle Stack Overflow:
  - Base Case:
    - Ensure that the base case is correctly defined and reachable.
    - Example: In a factorial function, ensure you return 1 when $n = 0$.
  - Tail Recursion:
    - Tail recursion is a specific type of recursion where the recursive call is the last operation in the function. This means that there is no additional computation after the recursive call.
    - Use it where possible, which can be optimized by some compilers.
    - Python does not optimize tail recursion, but it's a good practice in other languages.
  - Iterative Solutions:
    - Convert recursive algorithms to iterative ones using loops and stacks.
    - Example: Use a stack data structure to simulate recursion.
  - Increase Recursion Limit:
    - Use `sys.setrecursionlimit(limit)` to increase the maximum recursion depth.
    - Caution: Increasing the limit can lead to crashes if not managed properly.

Recursion

- Recursion enables you to create an intuitive, straightforward, and simple solution to a problem. However, there are trade-offs associated with its use:
  - Calling a function consumes more time and memory than adjusting a loop counter.
  - High-performance applications (such as graphics-intensive games or simulations of nuclear explosions) rarely use recursion.
- In less demanding applications, recursion can be an attractive alternative to iteration (for the right problems!).

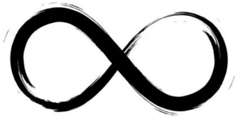
Caution with Loops and Recursion

- If we use a loop, we must be careful not to create an infinite loop by accident:

```
result = 1
while result > 0:
    ...
    result += 1  # Oops!
```

- Similarly, if we use recursion, we must be careful not to create an infinite chain of function calls:

```
def factorial(n: int) -> int:
    return n * factorial(n - 1)  # No base case (Oops!)

# ---------------------------------------------------------

def factorial(n: int) -> int:
    if n == 1:
        return 1
    return n * factorial(n + 1)  # factorial(n + 1) (Oops!)
```



Conclusion

- Recursion is one way to decompose a task into smaller subtasks.
- At least one of the subtasks is a simpler example of the same task:
- We must always ensure the following when designing recursive functions:
  - A recursive function must contain at least one non-recursive branch (i.e., base case).
  - The recursive calls must eventually lead to a non-recursive branch.

  An example is the factorial function:
  - $n! = n \times (n-1)!$ (Simpler subtask is $(n-1)!$)
  - $1! = 1$ (The simplest task is $n = 1$)

Key Terms

- base case
- infinite recursion
- recursive function
- stopping condition

Review Questions

Fill in the blanks in each of the following sentences about the Python environment.

1. A ________________ is one that directly or indirectly invokes itself. For a ________________ to terminate, there must be one or more ________________.
2. Recursion is an alternative form of program control. It is essentially repetition without a loop control. It can be used to write simple, clear solutions for inherently recursive problems that would otherwise be ________________.
3. Recursion bears substantial ________________. Each time the program calls a function, the system must allocate memory for all of the function's ________________ and ________________. This can consume considerable computer memory and requires extra time to manage the additional memory.

Answer: 1. recursive function; recursive function; base cases, 2. difficult to solve, 3. overhead; local variables; parameters

Further Reading

- Read Chapter 15 of “Introduction to Python Programming and Data Structures” textbook.

That's all!

Any questions?