13 – Recursion

COMP 125 Programming with Python

Recording Disclaimer

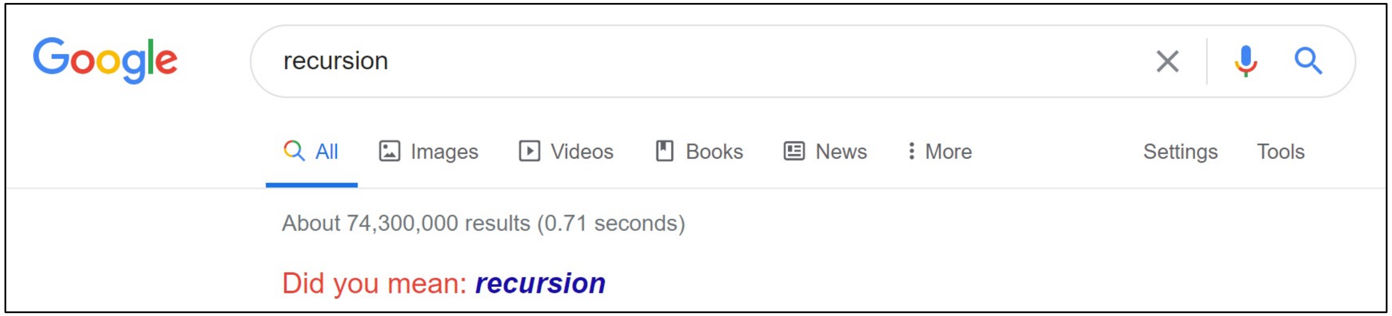
The synchronous sessions are recorded (audiovisual recordings). The students are not required to keep their cameras on during class.

The audiovisual recordings, presentations, readings and any other works offered as the course materials aim to support remote and online learning. They are only for the personal use of the students. Further use of course materials other than the personal and educational purposes as defined in this disclaimer, such as making copies, reproductions, replications, submission and sharing on different platforms including the digital ones or commercial usages are strictly prohibited and illegal.

The persons violating the above-mentioned prohibitions can be subject to the administrative, civil, and criminal sanctions under the Law on Higher Education Nr. 2547, the By-Law on Disciplinary Matters of Higher Education Students, the Law on Intellectual Property Nr. 5846, the Criminal Law Nr. 5237, the Law on Obligations Nr. 6098, and any other relevant legislation.

The academic expressions, views, and discussions in the course materials including the audio-visual recordings fall within the scope of the freedom of science and art.

Recursion



- Recursion is an extremely powerful problem-solving technique
  - It breaks a problem into smaller identical problems and uses **the same function** to solve these smaller problems
  - It is an alternative to iterative solutions, which use loops

"In order to understand recursion, you must first understand recursion."

- Facts about recursive solutions
  - A recursive function calls itself
  - Each recursive call solves an identical but a smaller problem
  - Base case must be defined (it enables to stop the recursive calls)
  - Eventually, one of the smaller problems must be the base case

Factorial function (*iterative solution*)

$$n! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot (n-1) \cdot n$$

$$f(n) = \begin{cases} 1 & \text{if n} = 0 \\ n \cdot f(n-1) & \text{otherwise} \end{cases}$$

```python
# assumes that n is non-negative
def iterative_factorial(n):
    if n == 0:
        return 1
    else:
        result = 1
        for i in range(1, n + 1):
            result *= i
        return result
```

Factorial function (*recursive solution*)

```python
# assumes that n is non-negative
def recursive_factorial(n):
    if n == 0:
        return 1
    else:
        return n * recursive_factorial(n - 1)
```

This is the base case, it stops recursive function calls

The function calls itself but a smaller argument

- Facts about recursive solutions
  - A recursive function calls itself
  - Each recursive call solves an identical but a smaller problem
  - Base case must be defined (it enables to stop the recursive calls)
  - Eventually, one of the smaller problems must be the base case

What happens if n is a negative integer?

Factorial function (*recursive solution*)

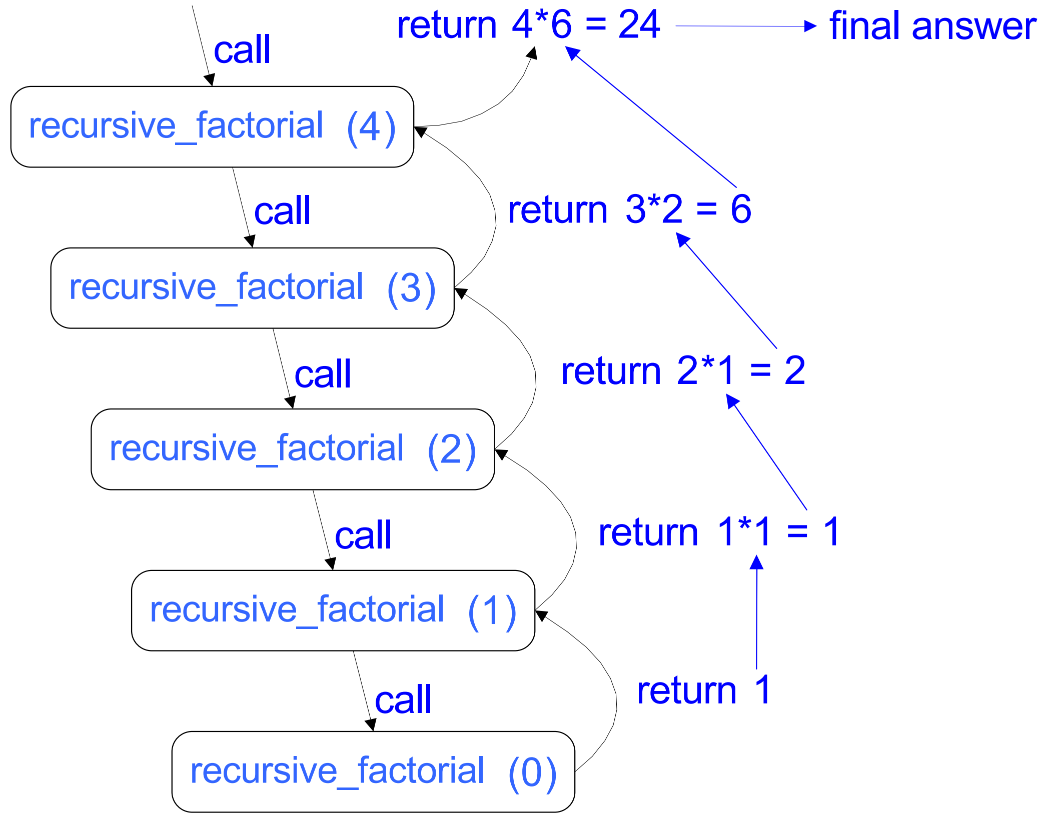
```
In [4]: recursive_factorial(-5)


  File "/Users/cigdem/Desktop/recursion.py", line 17, in recursive_factorial
    return n * recursive_factorial(n - 1)

  File "/Users/cigdem/Desktop/recursion.py", line 17, in recursive_factorial
    return n * recursive_factorial(n - 1)

  File "/Users/cigdem/Desktop/recursion.py", line 17, in recursive_factorial
    return n * recursive_factorial(n - 1)

  File "/Users/cigdem/Desktop/recursion.py", line 14, in recursive_factorial
    if n == 0:

RecursionError: maximum recursion depth exceeded in comparison
```

You have to be sure that the function eventually reaches the base case

The number of times that a function calls itself is known as the depth of recursion

Visualizing recursion

- Recursion trace
  - A box for each recursive call
  - An arrow from each caller to callee
  - An arrow from each callee to caller showing return value



Summing numbers from 1 to N

```
def iterative_sum(N):
    result = 0
    for i in range(N + 1):
        result += i
    return result
```

```
def recursive_sum(N):
    if N <= 0:
        return 0
    if N == 1:
        return 1
    return N + recursive_sum(N - 1)
```

It is possible to have more than one base case!

Fibonacci numbers

Recursive definition

$$F(N) = F(N - 1) + F(N - 2), \text{ for all } N > 2$$

Base cases

$$F(1) = 1$$

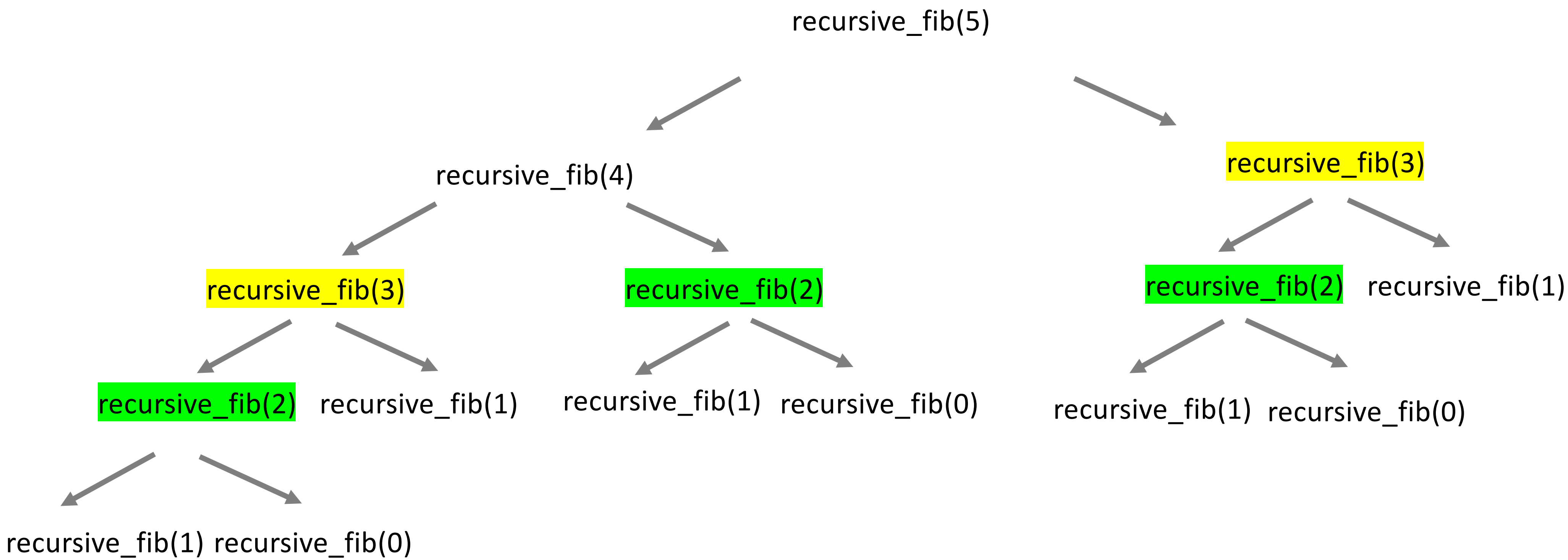
$$F(2) = 1$$

This is a bad example of recursion. It is too inefficient!!!

```
def iterative_fib(N):
    if N <= 0:
        return 0
    if N == 1 or N == 2:
        return 1
    first = 1
    second = 1
    for i in range(3, N + 1):
        result = first + second
        first = second
        second = result
    return result
```

```
def recursive_fib(N):
    if N <= 0:
        return 0
    if N == 1 or N == 2:
        return 1
    return recursive_fib(N - 1) + \
           recursive_fib(N - 2)
```

Visualizing recursion: fibonacci



Note the repeated calls!

What does it mean?

- Run the iterative and recursive functions for different values of N and measure the computational time

```python
import time

N = int(input('Enter N: '))

start = time.time()
res = iterative_fib(N)
end = time.time()
print(f"Iterative: {end - start:.6f} sec")

start = time.time()
res = recursive_fib(N)
end = time.time()
print(f"Recursive: {end - start:.6f} sec")
```

```
In [1]: runfile('/Users/cigdem/Des

Enter N: 5
Iterative: 0.000004 sec
Recursive: 0.000005 sec

In [2]: runfile('/Users/cigdem/Des

Enter N: 10
Iterative: 0.000006 sec
Recursive: 0.000045 sec

In [3]: runfile('/Users/cigdem/Des

Enter N: 20
Iterative: 0.000010 sec
Recursive: 0.005093 sec

In [4]: runfile('/Users/cigdem/Des

Enter N: 30
Iterative: 0.000011 sec
Recursive: 0.239757 sec

In [5]: runfile('/Users/cigdem/Des

Enter N: 40
Iterative: 0.000008 sec
Recursive: 29.496105 sec
```

What does it mean?

- Run the iterative and recursive functions for different values of N and measure the computational time

```
import time

N = int(input('Enter N: '))

start = time.time()
res = iterative_fib(N)
end = time.time()
print(f"Iterative: {end - start:.6f} sec")

start = time.time()
res = recursive_fib(N)
end = time.time()
print(f"Recursive: {end - start:.6f} sec")
```

```
In [5]: runfile('/Users/cigdem/Des

Enter N: 40
Iterative: 0.000008 sec
Recursive: 29.496105 sec

In [6]: runfile('/Users/cigdem/Des

Enter N: 41
Iterative: 0.000012 sec
Recursive: 53.246312 sec

In [7]: runfile('/Users/cigdem/Des

Enter N: 42
Iterative: 0.000013 sec
Recursive: 81.497787 sec

In [8]: runfile('/Users/cigdem/Des

Enter N: 43
Iterative: 0.000012 sec
Recursive: 131.133424 sec

In [9]: runfile('/Users/cigdem/Des

Enter N: 44
Iterative: 0.000012 sec
Recursive: 217.246881 sec
```

Recursion and efficiency

- Some recursive functions are so inefficient that they should not be used

- Factors contributing to this inefficiency
  - Inherent inefficiency of some recursive functions, such as the recursive_fib(…) function
  - Overhead associated with function calls

- Do not use a recursive function if it is inefficient and there is a clear and efficient implementation of the same algorithm using loops

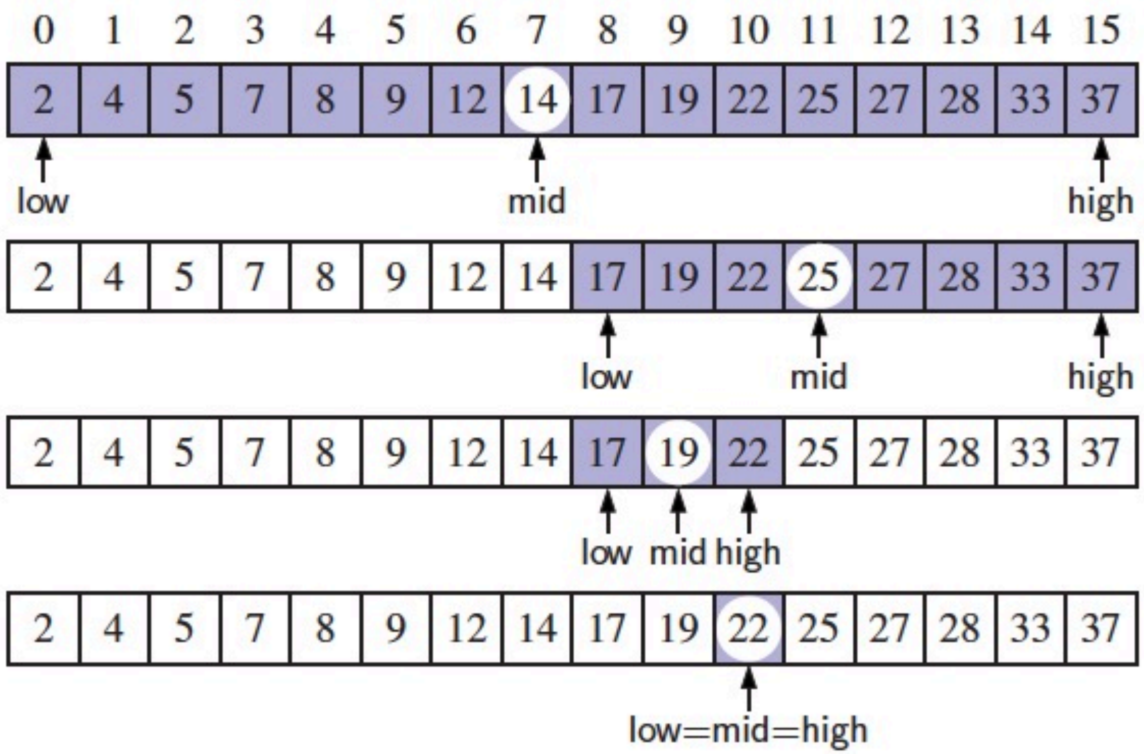
- **Efficiency (time complexity) analyses are discussed in more advanced CS courses**

Binary search

- This search algorithm works only on ***sorted sequences***
- It is a **very efficient search** algorithm

Find 22

- If the key equals data[mid], then we have found the target
- If key < data[mid], then we recur on the first half of the sequence
- If key > data[mid], then we recur on the second half of the sequence



```python
def binary_search(L, key, low, high):
    if low > high:
        return -1

    mid = (low + high) // 2 #integer division
    if L[mid] == key:
        return mid
    elif L[mid] > key:
        return binary_search(L, key, low, mid - 1)
    else:
        return binary_search(L, key, mid + 1, high)


def search(L, key):
    return binary_search(L, key, 0, len(L) - 1)

def main():
    D = [  3,  7, 10, 12, 14, 16, 19, 20, 23, 26, \
          32, 34, 37, 41, 43, 48, 52, 56, 59, 62, \
          67, 70, 72, 79, 81, 84, 89, 92, 95, 98 ]
    print(search(D, 16))
    print(search(D, 48))
    print(search(D, 77))

main()
```

```
In [1]: runfile('/Users/
5
15
-1
```

This is a very efficient search algorithm

It works very fast on even large sequences (e.g., containing millions of items)

It is much much faster than the linear search algorithm

Indirect recursion

- **Direct recursion**: function calls itself
  - e.g., binary_search(…) has a function call to binary_search(…)

- **Indirect recursion**: one function calls another function, which in turn calls the first function
  - e.g., function foo(…) calls function bar(…), and function bar(…) calls function foo(…)
  - Also called mutual recursion
  - Not restricted to two functions, can have a longer chain

Example

- Write two recursive functions that take an integer and return whether the integer is even or odd, respectively.

```
def isEven(N):
    if N == 0:
        return True
    return isOdd(N - 1)

def isOdd(N):
    if N == 0:
        return False
    return isEven(N - 1)
```

This is just to give you an indirect recursion example.

You know better/faster ways of implementing these functions

(N % 2 == 0 or N % 2 == 1)