COL1000
Introduction to Programming

Priyanka Golia

Most (if not all) of the content is borrowed from Prof. Subodh Kumar's slides

Recursion!

```python
n = int(input("enter a number"))
e = int(input("enter an exponent"))

def power(n,e):
    if e == 1:
        return n
    return n * power(n,e-1)

print(f"{n}^{e} is {power(n,e)}")
```

```
enter a number3
enter an exponent4
3^4 is 81
```

Recursion!

```python
n = int(input("enter a number"))
e = int(input("enter an exponent"))

def power(n,e):
    if e == 1:
        return n
    return n * power(n,e-1)

print(f"{n}^{e} is {power(n,e)}")
```

```
enter a number3
enter an exponent4
3^4 is 81
```

Did it cover all the cases?

Recursion!

```python
n = int(input("enter a number"))
e = int(input("enter an exponent"))

def power(n,e):
    if e == 1:
        return n
    return n * power(n,e-1)

print(f"{n}^{e} is {power(n,e)}")
```

```
enter a number3
enter an exponent4
3^4 is 81
```

Did it cover all the cases?

What if e is negative ?

Recursion!

```python
n = int(input("enter a number"))
e = int(input("enter an exponent"))

def power(n,e):
    if e == 1:
        return n
    return n * power(n,e-1)

print(f"{n}^{e} is {power(n,e)}")
```

```
enter a number3
enter an exponent4
3^4 is 81
```

Did it cover all the cases?

What if e is negative ?

What if e is zero?

Recursion!

```
1  n = int(input("enter a number (base)"))
2  e = int(input("enter an exponent"))
3
4  def power(n,e):
5      if e < 0:
6          if n == 0:
7              print("for negative exponent base cannot be zero")
8              return
9          return 1 / power(n,-e)
10     if e == 0:
11         return 1
12     return n * power(n,e-1)
13
14 print(f"{n}^{e} is {power(n,e)}")
```

Recursion!

```python
n = int(input("enter a number (base)"))
e = int(input("enter an exponent"))

def power(n,e):
    if e < 0:
        if n == 0:
            print("for negative exponent base cannot be zero")
            return
        return 1 / power(n,-e)
    if e == 0:
        return 1
    return n * power(n,e-1)

print(f"{n}^{e} is {power(n,e)}")
```

How many times function “power” will be called?

Recursion!

```python
n = int(input("enter a number (base)"))
e = int(input("enter an exponent"))

def power(n,e):
    if e < 0:
        if n == 0:
            print("for negative exponent base cannot be zero")
            return
        return 1 / power(n,-e)
    if e == 0:
        return 1
    return n * power(n,e-1)

print(f"{n}^{e} is {power(n,e)}")
```

How many times function “power” will be called?

This is what we called time complexity — for the worst possible input, how much time will the program take — it could be “how many times the loop will run”, “how many recursive call” .
We also have best case time complexity, avg. complexity. Similarly, we have space complexity.

Recursion!

```python
n = int(input("enter a number (base)"))
e = int(input("enter an exponent"))

def power(n,e):
    if e < 0:
        if n == 0:
            print("for negative exponent base cannot be zero")
            return
        return 1 / power(n,-e)
    if e == 0:
        return 1
    return n * power(n,e-1)

print(f"{n}^{e} is {power(n,e)}")
```

How many times function “power” will be called?

~ e many times. it makes one recursive call per unit decrease in exponent.

This is what we called time complexity — for the worst possible input, how much time will the program take — it could be “how many times the loop will run”, “how many recursive call” .
We also have best case time complexity, avg. complexity. Similarly, we have space complexity.

Recursion!

```python
n = int(input("enter a number (base)"))
e = int(input("enter an exponent"))

def power(n,e):
    if e < 0:
        if n == 0:
            print("for negative exponent base cannot be zero")
            return
        return 1 / power(n,-e)
    if e == 0:
        return 1
    return n * power(n,e-1)

print(f"{n}^{e} is {power(n,e)}")
```

How many times function “power” will be called?

~ e many times. it makes one recursive call per unit decrease in exponent.

Can we optimize it?

This is what we called time complexity — for the worst possible input, how much time will the program take — it could be “how many times the loop will run”, “how many recursive call” .
We also have best case time complexity, avg. complexity. Similarly, we have space complexity.

Recursion!

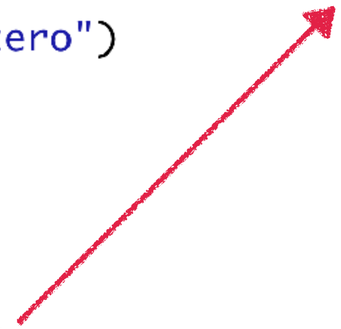
```python
n = int(input("enter a number (base)"))
e = int(input("enter an exponent"))
count = 0
def power(n,e):
    global count
    count += 1
    if e < 0:
        if n == 0:
            print("for negative exponent base cannot be zero")
            return
        return 1 / power(n,-e)
    if e == 0:
        return 1
    half = power(n, e // 2)
    return half * half if e % 2 == 0 else n * half * half

print(f"{n}^{e} is {power(n,e)}")
print(f"we called power {count} many times")
```

Recursion!

```python
n = int(input("enter a number (base)"))
e = int(input("enter an exponent"))
count = 0
def power(n,e):
    global count
    count += 1
    if e < 0:
        if n == 0:
            print("for negative exponent base cannot be zero")
            return
        return 1 / power(n,-e)
    if e == 0:
        return 1
    half = power(n, e // 2)
    return half * half if e % 2 == 0 else n * half * half

print(f"{n}^{e} is {power(n,e)}")
print(f"we called power {count} many times")
```



Simple concept:
exponentiation by squaring.

$n^6 = n^3 \times n^3$ — e is even

$n^5 = n \times n^2 \times n^2$ — e is odd.

Recursion!

```python
n = int(input("enter a number (base)"))
e = int(input("enter an exponent"))
count = 0
def power(n,e):
    global count
    count += 1
    if e < 0:
        if n == 0:
            print("for negative exponent base cannot be zero")
            return
        return 1 / power(n,-e)
    if e == 0:
        return 1
    half = power(n, e // 2)
    return half * half if e % 2 == 0 else n * half * half

print(f"{n}^{e} is {power(n,e)}")
print(f"we called power {count} many times")
```

Simple concept: exponentiation by squaring.

$n^6 = n^3 \times n^3$ — e is even

$n^5 = n \times n^2 \times n^2$ — e is odd.

```
enter a number (base)2
enter an exponent500
2^500 is 3273390607896141870013189696827599152216642046043064789483291368096133796404674554883270092325904157150886684127560071009217256545885393053328527589376
we called power 10 many times
```

Recursion!

```python
n = int(input("enter a number (base)"))
e = int(input("enter an exponent"))
count = 0
def power(n,e):
    global count
    count += 1
    if e < 0:
        if n == 0:
            print("for negative exponent base cannot be zero")
            return
        return 1 / power(n,-e)
    if e == 0:
        return 1
    half = power(n, e // 2)
    return half * half if e % 2 == 0 else n * half * half

print(f"{n}^{e} is {power(n,e)}")
print(f"we called power {count} many times")
```

Notice that we divide the exponent every time by 2, not by 1.
— we will reach base case quickly.
500 -> 250 -> 125 -> 62 -> 31-> 15 -> 7 -> 3 -> 2-> 1 -> 0

It takes ~ log e many call.

Recursion!

GCD (Greatest Common Divisor).

```
gcd(a, b) =
          a                   if b == 0
          gcd(b, a - b)       if a > b
          gcd(a, b - a)       otherwise
```

Recursion!

GCD (Greatest Common Divisor).

```
gcd(a, b) =
          a                  if b == 0
          gcd(b, a - b)      if a > b
          gcd(a, b - a)      otherwise
```

```python
count = 0
def gcd(a, b):
    global count
    count += 1
    if b == 0:
        return a
    elif a > b:
        return gcd(b, a - b)
    else:
        return gcd(a, b - a)

a = int(input("enter first number"))
b = int(input("enter second number"))
print(f"gcd of {a} and {b} is {gcd(a,b)}")
print(f"called gcd function {count} many times")
```

Recursion!

GCD (Greatest Common Divisor).

```
gcd(a, b) =
          a                    if b == 0
          gcd(b, a - b)        if a > b
          gcd(a, b - a)        otherwise
```

```python
count = 0
def gcd(a, b):
    global count
    count += 1
    if b == 0:
        return a
    elif a > b:
        return gcd(b, a - b)
    else:
        return gcd(a, b - a)

a = int(input("enter first number"))
b = int(input("enter second number"))
print(f"gcd of {a} and {b} is {gcd(a,b)}")
print(f"called gcd function {count} many times")
```

How many times we need to call gcd?

Recursion!

GCD (Greatest Common Divisor).

```
gcd(a, b) =
          a                    if b == 0
          gcd(b, a - b)        if a > b
          gcd(a, b - a)        otherwise
```

```
count = 0
def gcd(a, b):
    global count
    count += 1
    if b == 0:
        return a
    elif a > b:
        return gcd(b, a - b)
    else:
        return gcd(a, b - a)

a = int(input("enter first number"))
b = int(input("enter second number"))
print(f"gcd of {a} and {b} is {gcd(a,b)}")
print(f"called gcd function {count} many times")
```

```
enter first number4500
enter second number2345
gcd of 4500 and 2345 is 5
called gcd function 29 many times
```

How many times we need to call gcd?

Recursion!

GCD (Greatest Common Divisor).

```
gcd(a, b) =
          a                    if b == 0
          gcd(b, a - b)        if a > b
          gcd(a, b - a)        otherwise
```

```
count = 0
def gcd(a, b):
    global count
    count += 1
    if b == 0:
        return a
    elif a > b:
        return gcd(b, a - b)
    else:
        return gcd(a, b - a)

a = int(input("enter first number"))
b = int(input("enter second number"))
print(f"gcd of {a} and {b} is {gcd(a,b)}")
print(f"called gcd function {count} many times")
```

```
enter first number4500
enter second number2345
gcd of 4500 and 2345 is 5
called gcd function 29 many times
```

How many times we need to call gcd?

Worst case ~ max(a,b), because it subtracts one number from the other repeatedly.

Recursion!

GCD (Greatest Common Divisor).

Euclidean algorithm:

```
gcd(a, b) =
          a                    if b == 0
          gcd(b, a mod b)      otherwise
```

```python
count = 0
def gcd(a, b):
    global count
    count += 1
    a, b = abs(a), abs(b)
    if b == 0:
        return a
    return gcd(b, a % b)

a = int(input("enter first number"))
b = int(input("enter second number"))
print(f"gcd of {a} and {b} is {gcd(a,b)}")
print(f"called gcd function {count} many times")

```

```
enter first number4500
enter second number2345
gcd of 4500 and 2345 is 5
called gcd function 7 many times
```

~log(min(a,b)), which is much faster — it reduces the size of the numbers exponentially.

Quiz

```
x = “global"
def inner():
        global x
        x = "inner"
def outer():
    x = "outer"
    inner()
    print(x)
outer()
print(x)
```

```
def outer():
    def inner():
        print("I am inner")
    print("I am outer”)
    return inner

f = outer()
print(f())
```

```
def outer():
    def inner():
        print("I am inner")
    print("I am outer”)
    return inner

f = outer()
print(type(f))
print(type(outer))
```

```
def len(a):
    print("i am inside len")
    print(a)

x = [1,2,3,4]
len(x)
sum = 0
print(sum(x))
```

Quiz

```
x = “global"
def inner():
        global x
        x = "inner"
def outer():
    x = "outer"
    inner()
    print(x)
outer()
print(x)
```

outer
inner

```
def outer():
    def inner():
        print("I am inner")
    print("I am outer”)
    return inner

f = outer()
print(f())
```

```
def outer():
    def inner():
        print("I am inner")
    print("I am outer”)
    return inner

f = outer()
print(type(f))
print(type(outer))
```

```
def len(a):
    print("i am inside len")
    print(a)

x = [1,2,3,4]
len(x)
sum = 0
print(sum(x))
```

Quiz

```
x = “global"
def inner():
        global x
        x = "inner"
def outer():
    x = "outer"
    inner()
    print(x)
outer()
print(x)
```

outer
inner

```
def outer():
    def inner():
        print("I am inner")
    print("I am outer”)
    return inner

f = outer()
print(f())
```

I am outer
I am inner
None

```
def outer():
    def inner():
        print("I am inner")
    print("I am outer”)
    return inner

f = outer()
print(type(f))
print(type(outer))
```

```
def len(a):
    print("i am inside len")
    print(a)

x = [1,2,3,4]
len(x)
sum = 0
print(sum(x))
```

Quiz

```
x = “global"
def inner():
        global x
        x = "inner"
def outer():
    x = "outer"
    inner()
    print(x)
outer()
print(x)
```

outer
inner

```
def outer():
    def inner():
        print("I am inner")
    print("I am outer”)
    return inner

f = outer()
print(f())
```

Notice outer has return function as object.

I am outer
I am inner
None

```
def outer():
    def inner():
        print("I am inner")
    print("I am outer”)
    return inner

f = outer()
print(type(f))
print(type(outer))
```

```
def len(a):
    print("i am inside len")
    print(a)

x = [1,2,3,4]
len(x)
sum = 0
print(sum(x))
```

Quiz

```
x = “global"
def inner():
        global x
        x = "inner"
def outer():
    x = "outer"
    inner()
    print(x)
outer()
print(x)
```

outer
inner

```
def outer():
    def inner():
        print("I am inner")
    print("I am outer”)
    return inner

f = outer()
print(f())
```

Notice outer has return function as object.

I am outer
I am inner
None

```
def outer():
    def inner():
        print("I am inner")
    print("I am outer”)
    return inner

f = outer()
print(type(f))
print(type(outer))
```

<class 'function'>
<class 'function'>

```
def len(a):
    print("i am inside len")
    print(a)

x = [1,2,3,4]
len(x)
sum = 0
print(sum(x))
```

Quiz

```
x = “global"
def inner():
        global x
        x = "inner"
def outer():
    x = "outer"
    inner()
    print(x)
outer()
print(x)
```

outer
inner

```
def outer():
    def inner():
        print("I am inner")
    print("I am outer”)
    return inner

f = outer()
print(f())
```

Notice outer has return function as object.

I am outer
I am inner
None

```
def outer():
    def inner():
        print("I am inner")
    print("I am outer”)
    return inner

f = outer()
print(type(f))
print(type(outer))
```

<class 'function'>
<class 'function'>

```
def len(a):
    print("i am inside len")
    print(a)

x = [1,2,3,4]
len(x)
sum = 0
print(sum(x))
```

```
i am inside len
[1, 2, 3, 4]
Traceback (most recent call last):
  File "run.py", line 1, in <module>
    import lec_main
  File "/home/p10838/lec_main.py", line 3, in <module>
    import lec23
  File "/home/p10838/lec23.py", line 8, in <module>
    print(sum(x))
TypeError: 'int' object is not callable
```

A function can return another function

```
def outer():
    def inner():
        print("I am inner")
    print("I am outer”)
    return inner

f = outer()
print(f())
```

```
def power_factory(n):
    def power(x):
        return x ** n
    return power
```

```
square = power_factory(2)
cube = power_factory(3)

print(square(5))   # 25
print(cube(2))     # 8
```

The function power_factory *creates* and *returns* new functions — one that squares, one that cubes.
Each returned function remembers the value of n used when it was created.

A closure is a function that remembers the environment in which it was created, even after that environment is gone