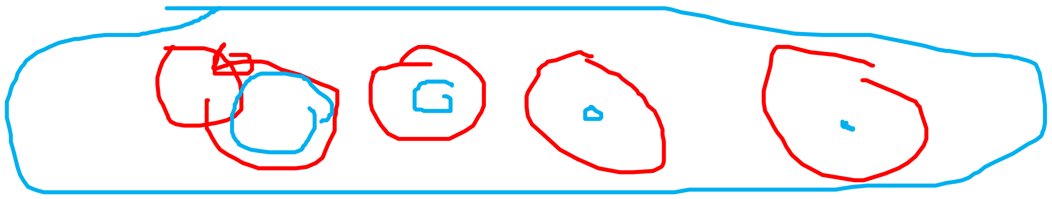
Recursion and Backtracking

Data structure, Computer science department, college of science for women, University of Baghdad.

Dr. Amer Al-Mahdawi

What is Recursion



- Any function which calls itself is called *recursive.*
- A recursive method solves a problem by calling a copy of itself to work on a smaller problem.
- This is called the recursion step.
- The recursion step can result in many more such recursive calls.
- It is important to ensure that the recursion terminates.
- Each time the function calls itself with a slightly simpler version of the original problem.
- The sequence of smaller problems must eventually converge on the base case.

Why Recursion?

- Recursion is a useful technique borrowed from mathematics.
- Recursive code is generally shorter and easier to write than iterative code.
- Generally, loops are turned into recursive functions when they are compiled or interpreted.
- Recursion is most useful for tasks that can be defined in terms of similar subtasks.
- for example, sort, search, and traversal problems often have simple recursive solutions.

Format of a Recursive Function

- A recursive function performs a task in part by calling itself to perform the subtasks.
- At some point, the function encounters a subtask that it can perform without calling itself.
- This case, where the function does not recur, is called the *base case.*
- The former, where the function calls itself to perform a subtask, is referred to as the *cursive case.*
- We can write all recursive functions using the format:

Format of a Recursive Function

- if(test for the base case)
-     return some base case value
- else if(test for another base case)
-     return some other base case value
-   // the recursive case
- Else
-   return (some work and then a recursive call)

Format of a Recursive Function

- As an example, consider the factorial function: n! is the product of all integers between n and 1.
- The definition of recursive factorial looks like:
- n! = 1 ,   if n = 0
- n! = n * (n - 1)! if *n* > o
- This definition can easily be converted to recursive implementation.
- Here the problem is determining the value of n!, and the subproblem is determining the value of (n - 1)!.

Format of a Recursive Function

- In the recursive case, when n is greater than 1 , the function calls itself to determine the value of(n - 1)! and multiplies that with n.
- In the base case, when *n* is 0 or 1 , the function simply returns 1. This looks like the following:
- // calculates factorial of n positive integer
- def factorial(n):
-     if n == 0: return 1
-     return n * factorial(n-1)

- Print (factorial(4))

Recursion and Memory (Visualization)

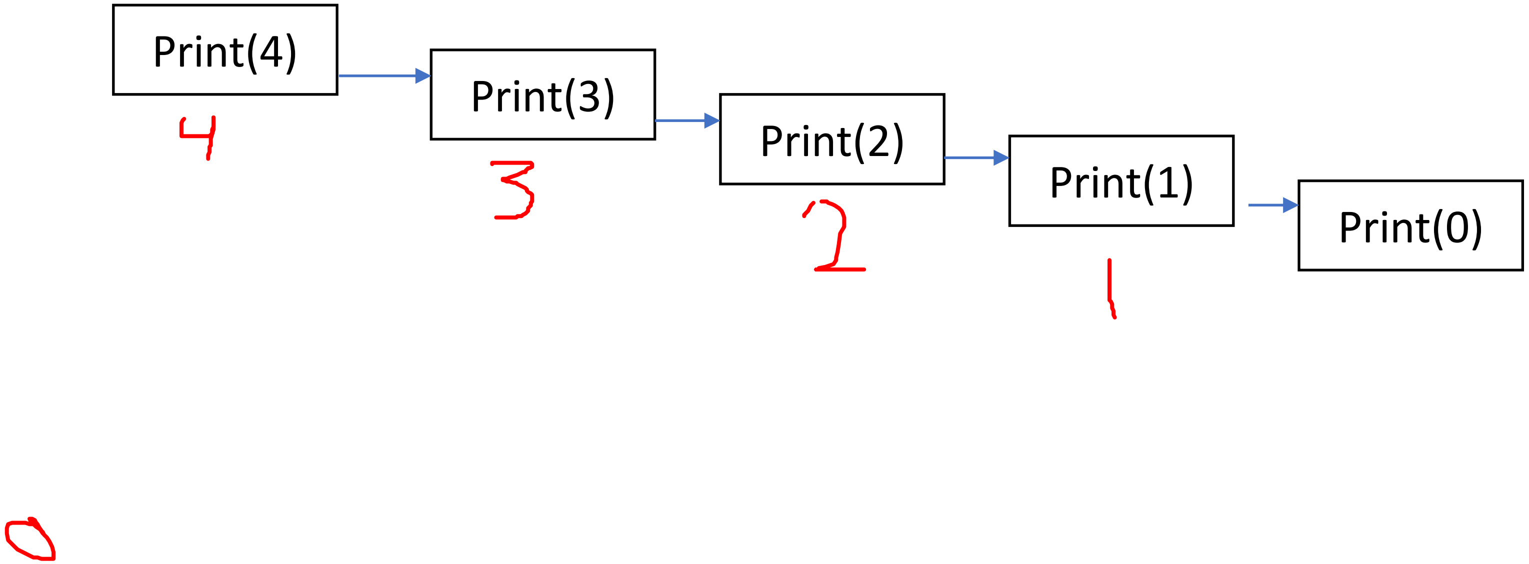
- Each recursive call makes a new copy of that method (only the variables) in memory.
- Once a method ends (that is, returns some data), the copy of that returning method is removed from memory.
- The recursive solutions look simple, but visualization and tracing takes time.
- For better understanding, let us consider the following example.

Recursion and Memory (Visualization)

- def Print(n):
- if n == 0: # this is the terminating base case
- return 0
- else:
- print n
- return Print(n-1) # recursive call to itself again
- print(Print(4))

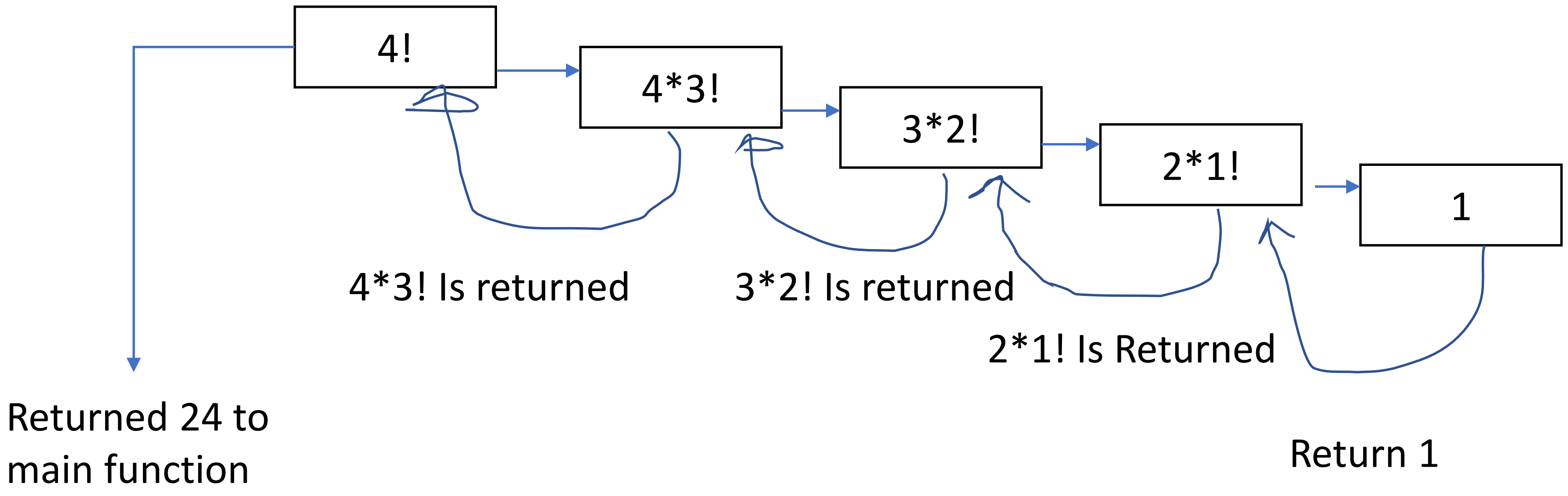
Recursion and Memory (Visualization)

- For this example, if we call the print function with n=4, visually our memory assignments may look like:



Recursion and Memory (Visualization)

- For this example, if we call the print function with n=4, visually our memory assignments may look like:



Recursion versus Iteration

- While discussing recursion, the basic question that comes to mind is: which way is better? - iteration or recursion?.
- The answer to this question depends on what we are trying to do.
- A recursive approach mirrors the problem that we are trying to solve.
- A recursive approach makes it simpler to solve a problem that may not have the most obvious of answers.
- But recursion adds overhead for each recursive call (needs space on the stack frame).

Recursion

- Terminates when a base case is reached.
- Each recursive call requires extra space on the stack frame (memory).
- If we get infinite recursion, the program may run out of memory and result in stack overflow.
- Solutions to some problems are easier to formulate recursively.

Iteration

- Terminates when a condition is proven to be false.
- Each iteration does not require extra space.
- An infinite loop could loop forever since there is no extra memory being created.
- Iterative solutions to a problem may not always be as obvious as a recursive solution.

Notes on Recursion

- Recursive algorithms have two types of cases, recursive cases and base cases.
- Every recursive function case must terminate at a base case.
- Generally, iterative solutions are more efficient than recursive solutions [due to the overhead of function calls].
- A recursive algorithm can be implemented without recursive function calls using a stack, but it’s usually more trouble than its worth. That means any problem that can be solved recursively can also be solved iteratively.
- For some problems, there are no obvious iterative algorithms.
- Some problems are best suited for recursive solutions while others are not.

What is Backtracking?

- Backtracking is a form of recursion.
- The usual scenario is that you are faced with several options, and you must choose one or these.
- After you make your choice, you will get a new set of options; just what set of options you get depends on what choice you made.
- This procedure is repeated over and over until you reach a final state.
- If you made a good sequence of choices, your final state is a goal state; if you didn't, it isn’t.
- Backtracking is a method of exhaustive search using divide and conquer.