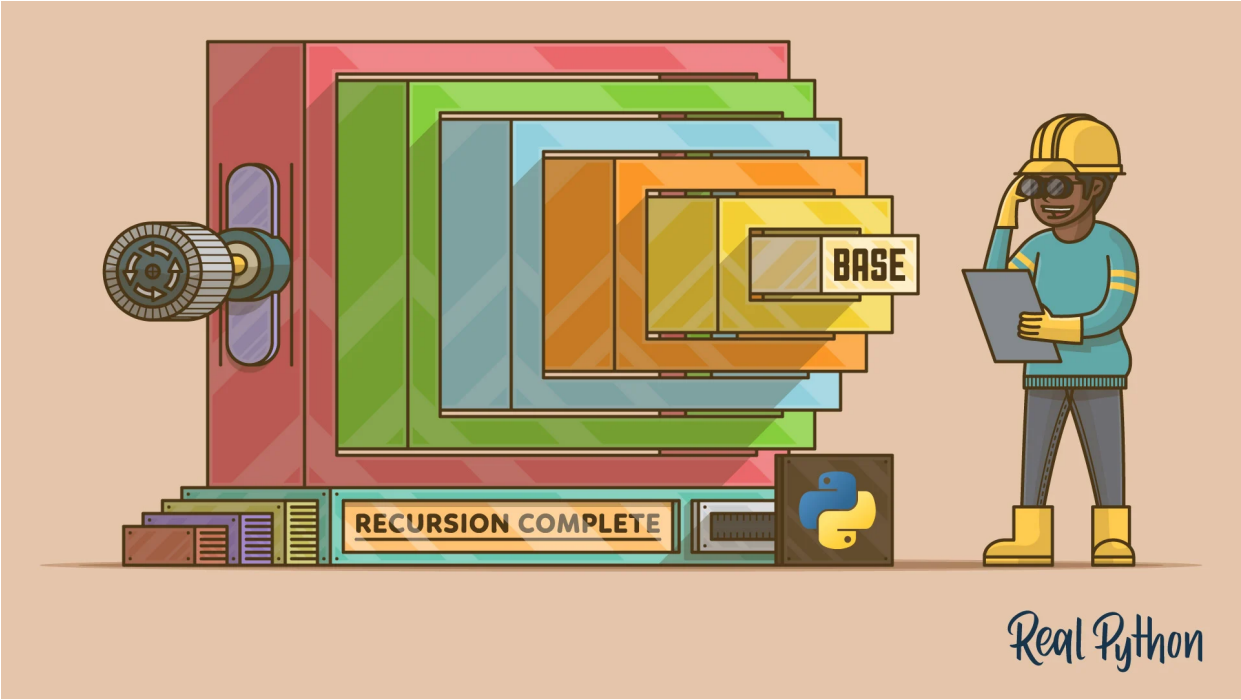
Lecture 21: Recursion and Backtracking

Date: 10/31/2023

Recursion In-Depth

Recursion is a way to solve a problem by solving smaller subproblems.



reference: https://realpython.com/python-thinking-recursively/

Generic Pseudocode

```
function recursion(big problem){
    if reached base state:
        return answer

    // solve smaller problems
```

```
    recursion(small problem)
    recursion(small problem)
    ...

    // based on the answers of smaller problems
    // figure out answer for bigger problem
    return answer
}
```

Types of Recursion

1. Tail Recursion

The recursive call is the last statement that is being executed by the function.

```
function tail_recursion(problem){
    if reached base state:
        return answer

    // do some task
    return tail_recursion(small problem)
}
```

2. Non-Tail Recursion

The recursive call is not the last statement that is being executed by the function.

```
function nontail_recursion(problem){
    if reached base state:
       return answer

    // do some task
    answer = tail_recursion(small problem)
    // do some task

    return answer*2
}
```

3. Tree Recursion

There are more than one recursive calls in one function.

```
function tree_recursion(problem){
    if reached base state:
        return answer

    tree_recursion(small problem1)
    tree_recursion(small problem2)

    return answer
}
```

⇒ **Which type is better?**

Examples

- Print 1 to N

⇒ **First step** is to figure out how you can build your answer using sub-problems

one way: “print all numbers from 1 to n-1” and then print n”

```
void print(int n){
    print(n-1);
    cout << n;
}
```

second way: “print starting number and then print all remaining numbers”

```
void print(int start, int n){
    cout << start;
    print(start+1, n);
}
```

⇒ **Second step** is to figure out the base case to stop the recursion

in the first way: we stop at number 1.

```
void print(int n){
    if n <= 1:
```

```
        return;

    print(n-1);
    cout << n;
}
```

in the second way: we stop when starting number reaches n

```
void print(int start, int n){
    if start >= n:
        return;

    cout << start;
    print(start+1, n);
}
```

⇒ Which one is faster?

- Print string in Reverse

first step: “print last character and reverse all characters from 1 to strlen-1”
second step: “stop when we don’t have any characters to reverse”

```
void reverse(string str){
    if len(str) == 0:
        return;

    cout << str[strlen-1];
    reverse(str.remove_last());
}
```

first step: “print characters from start+1 to strlen in reverse order and then print start character”
second step: “stop when start reaches n”

```
void reverse(string str, int start){
    if start == len(str):
        return;
```

```
    reverse(str, start + 1);
    cout << str[start];
}
```

⇒ Which one is faster?

- Print even indexed characters

DIY

Populating Answer

⇒ We need to have a proper way to populate the answer from sub-problems and build the answer for the bigger problem.
Types:

1. Use *return* value
2. Use shared variable to store the answer
    a. Pass it in the argument
    b. Use global variable

- Print Factorial of a Number

first step: “find factorial of n-1 and **multiply** that value with n”
second step: “stop when n reaches 0”

⇒ Using return value

```
int factorial(int n){
    if (n<=0)
        return 1;

    int fact_n_1 = factorial(n-1);
    return fact_n_1 * n;
}
```

⇒ Passing answer in argument

```
void factorial(int n, int& answer){
    if (n<=0)
        return 1;

    int fact_n_1 = factorial(n-1);
    answer = fact_n_1 * n;
}
```

⇒ Using global variable

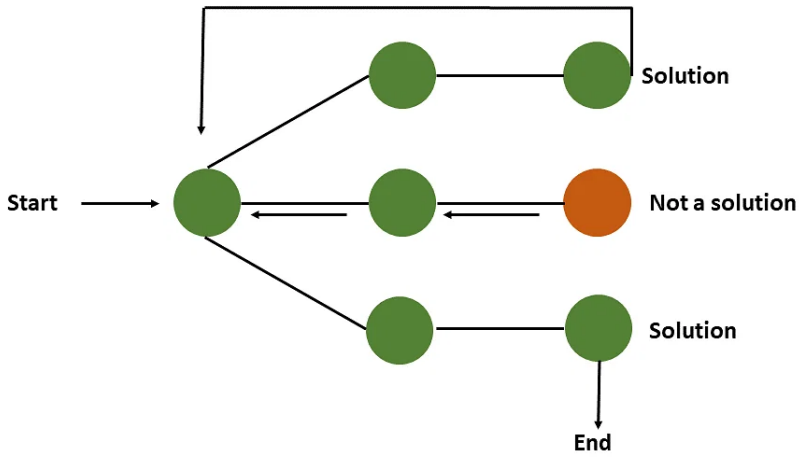
```
int answer;
void factorial(int n){
    if (n<=0)
        return 1;

    int fact_n_1 = factorial(n-1);
    answer = fact_n_1 * n;
}
```

- Sum of N numbers

DIY

Backtracking

Backtracking is a problem-solving algorithmic technique that involves finding a solution incrementally by trying different options and undoing them if they lead to a dead end.



reference:
https://www.simplilearn.com/tutorials/data-structure-tutorial/backtracking-algorithm

Difference

Recursion	Backtracking
Recursion does not always need backtracking	Backtracking always uses recursion to solve problems
A recursive function solves a particular problem by calling a copy of itself and solving smaller subproblems of the original problems.	Backtracking at every step eliminates those choices that cannot give us the solution and proceeds to those choices that have the potential of taking us to the solution.
Recursion is a part of backtracking itself and it is simpler to write.	Backtracking is comparatively complex to implement.
Applications of recursion are Tree and Graph Traversal, Towers of Hanoi, Divide and Conquer Algorithms, Merge Sort, Quick Sort, and Binary Search.	Application of Backtracking is N Queen problem, Rat in a Maze problem, Knight's Tour Problem, Sudoku solver, and Graph coloring problems.

reference:
https://www.geeksforgeeks.org/what-is-the-difference-between-backtracking-and-recursion/

Generic Pseudocode

```
function backtrack(state){
    if state is solution
        save the solution
        return

    for all available valid options from current state:
        go to new state
        backtrack(new_state)
```

```
        return to old state
```

Types

- **Decision** Problems: Here, we search for a feasible solution.
- **Optimization** Problems: For this type, we search for the best solution.
- **Enumeration** Problems: We find a set of all possible feasible solutions to the problems of this type.

Example

Find All Permutations of given array

Input: [1,2,3]
Output: [1,2,3], [1,3,2], [2,1,3], [2,3,1], [3,1,2], [3,2,1]