COL1000: Introduction to Programming

Functions

Subodh Sharma | Lec 22 | Oct 08

Co-Develop: Binary Search

- Given a **list of sorted** elements and an element **x**
  - Find whether x exists in the list and if so where?
- Algorithm:

1. Search x at the mid of the list: If found then return the index, else go to step (2)
2. If the element at mid > x, then lst = lst[mid+1:] and go to step 1
3. Else, lst = lst[:mid] and go to step 1

Non-linear Recursion: Tree Recursion

- Fibonacci
  - Fib (n) = Fib(n-1) + Fib(n-2)
  - **Recursive call stack grows in the form of a binary tree**
  - **Exponential Time q**
- Count the number of paths in a grid $m \times n$**,** moving right and down
  - P(m,n) = 1 if m = 1 or n = 1 else P(m-1, n) + P(m, n-1)

Non-linear Recursion: Tree Recursion

- **Fibonacci**
  - Fib (n) = Fib(n-1) + Fib(n-2)

```python
def recFib_slow(n: int) -> int:
    if n < 2:                # base
        return n
    return recFib_slow(n-1) + recFib_slow(n-2)

print(recFib_slow(100))
```

Non-linear Recursion: Tree Recursion

- **Fibonacci — cached version (more efficient)**
- **Fib (n) = Fib(n-1) + Fib(n-2)**

```python
from functools import lru_cache

@lru_cache(maxsize=None)
def fib_cached(n: int) -> int:
    if n < 2:
        return n
    return fib_cached(n-1) + fib_cached(n-2)

print(fib_cached(100))
```

Decorator for memoization

Non-linear Recursion: Tree Recursion

- **Fibonacci — cached version (more efficient) — manual memoization**
- **Fib (n) = Fib(n-1) + Fib(n-2)**

```
_cache = {0: 0, 1: 1}

def fib(n: int) -> int:
    if n < 0:
        raise ValueError("n must be non-negative")
    if n in _cache:
        return _cache[n]
    _cache[n] = fib(n - 1) + fib(n - 2)
    return _cache[n]
```

Dictionary serving as a cache
Also records the base case

Checking cache first

Non-linear Recursion: Tree Recursion

- **Paths in an mxn grid**
  - **Convince yourself of the base case and the induction step!**

```
def paths(m: int, n: int) -> int:
    if m < 1 or n < 1:
        raise ValueError("m, n must be >= 1")
    if m == 1 or n == 1:
        return 1
    return paths(m - 1, n) + paths(m, n - 1)
```

Tail Recursion

- Special form of recursion that is the **very last computation** performed in the function

- ```
  def factorial(n:int) -> int:
      if n == 0: return 1
      elif n < 0:
          raise Error("Not defined on neg values")
      else: return n * factorial(n-1)
  ```

Not Tail Recursive
  ```

Tail Recursion: Why it matters?

- Memory efficient (uses O(1) space)
    - Reuses the same stack frame
        - Therefore no stack overflow
- Unfortunately, Python **does NOT perform tail call optimisation**

Functions: Exceptions

- Exceptions in functions **denote errors or unexpected conditions**
- There are two ways of handling exceptions in the code:
    - Explicitly **raising the errors with an appropriated exception message**
    - **Catching** them to handle them gracefully
    - Let us see examples of each!