WORKSHEET WEEK 02: RECURSION

2.1 Introduction

There is a proverb: *To understand recursion, you must first understand recursion.* In many cases complex problems can be solved by breaking them down into subproblems multiple times.

2.1.1 Sequences defined by Recursion

Linear recursion example – Fibonacci sequence:

$$F(n) = \begin{cases} 0 & \text{if } n = 0, \\ 1 & \text{if } n = 1, \\ F(n-1) + F(n-2) & \text{if } n > 1. \end{cases}$$

It is known that the Fibonacci numbers (obtained by solving Linear recurrence) can be computed by the following closed formula:

$$F_m = \frac{1}{\sqrt{5}} \left(\left(\frac{1+\sqrt{5}}{2} \right)^m - \left(\frac{1+\sqrt{5}}{2} \right)^m \right).$$

Essentially, Fibonacci numbers are the sum of two geometric series. The common ratios in both geometric series are the roots of the *characteristic polynomial* $x^2 = x + 1$.

$$n! = \begin{cases} 1 & \text{if } n = 0, \\ n \cdot (n-1)! & \text{if } n > 0. \end{cases}$$

If you need to evaluate factorials for large n, you can apply Stirling's approximation:

$$n! \approx \sqrt{2\pi n} \left(\frac{n}{e} \right)^n.$$

The ratio of both expressions has limit 1 as $n \to \infty$.

$$S(n,x) = \begin{cases} x & \text{if } n = 0, \\ S(n-1,x) + \frac{(-1)^n \cdot x^{2n-1}}{(2n-1)!} & \text{if } n > 0. \end{cases}$$

For a constant x build the sequence $S(0,x), S(1,x), S(2,x), \ldots$ recursively. It has the limit (Taylor series for $y = \sin x$):

$$\lim_{n \to \infty} S(x,n) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \ldots = \sin x.$$

2.1.2 Structural Recursion

Recursion in mathematics is not necessarily adding 1 to a parameter n. It is possible to define valid mathematical expressions recursively using a grammar:

- Any number is a mathematical expression,
- Any variable $x, y, \ldots$ is a mathematical expression,
- If E is a mathematical expression, then (E) is also a mathematical expression,
- If E_1 and E_2 are mathematical expressions, then also $E_1 + E_2$, $E_1 - E_2$, $E_1 \cdot E_2$, and E_1/E_2 are mathematical expressions.

To compute mathematical expressions, one can proceed recursively: Find which is the "outermost" production to make the expression; break it down to one of the above cases, evaluate the subexpressions and finally compute the answer.

Even fancier recursive expressions (language statements) can be obtained defining programming languages with Context-free grammars. *Parsers* are responsible for breaking down such recursively built expressions (or programming language constructs) and converting them to some representation that can be executed.

2.2 Implementing the Recursion

2.2.1 Run-time stack

Every time one function calls another (in most imperative languages – both compiled or interpreted), a new activation record is created and pushed to the *run-time stack*. It consists of the following elements:

Activation Record:

- Parameters and local variables (everything passed to the function call or defined therein),
- Dynamic link (a pointer to the activation record of the caller),
- Access link (used by the function to access non-local data),
- Return value (will contain the value needed by the caller).

Activation records are quite large data structures, but programmer does not need to create them – they are maintained automatically as the functions call each other.

If a function is defined recursively, the run-time stack can contain multiple activation records of the same function (but with different parameters).

2.2.2 Situations when run-time stacks should be avoided?

Sometimes user can choose to implement recursion without explicit function calls (user-defined data structures). Here are some downsides to the run-time stacks:

Stack Overflow: The runtime stack has a limited amount of space, and if the recursion goes too deep, it can result in a stack overflow error. Explicit stack can be more efficient, as you can control the size of the stack and avoid the overflow error.

Performance: When using the runtime stack, there is overhead involved in pushing and popping function calls, as well as managing the stack pointers. For a large number of recursive calls, this overhead can be significant.

Memory: The runtime stack is typically stored in the memory with limited access time, so accessing it can be slow. In contrast, an explicit stack can be implemented as an array or a linked list, which can be stored in the heap with better access time.

Debugging: When using the runtime stack, it can be difficult to debug the recursive algorithm, as you have limited visibility into the stack.

2.3 Tail, Non-Tail and Excessive Recursion

There are several ways to implement factorial algorithmically. Consider these two Python examples:

```python
def factorial_tail(n, result):
    if n == 0:
        return result
    else:
        return factorial_tail(n - 1, n * result)
```

```python
def factorial_classic(n):
    if n == 0:
        return 1
    else:
        return n * factorial_classic(n - 1)
```

```python
# Non-Tail Recursive function to calculate nth Fibonacci number F(n)
def fibonacci_bad(n):
    if n <= 1:
        return n
    else:
        return fibonacci_bad(n - 1) + fibonacci_bad(n - 2)
```

This is much worse than the non-tail recursive factorial, since every call to `fibonacci_bad` leads to two more calls (the number of recursive calls grows exponentially!).

2.4 Solving Recursions

Recursive algorithms are somewhat harder to analyze in terms of running time. Unlike algorithms that use straightforward loops and data structures (where it is possible to add up elementary steps that are necessary to complete the function), recursive algorithms call themselves, but on different arguments.

The natural way to solve recursive algorithms is to write recursions on the runtime.

Binary Search Problem: There exists a *divide and conquer* algorithm to search for an item in an ordered list (see previous worksheet). Let $B(n)$ denote the running time of Binary search to find an item in an array of length n.

Express the number of comparisons needed to complete Binary search (your expression of $B(n)$ can use values of $B(m)$ for smaller input values $m < n$). Solve the recursion for $B(m)$ and find a closed formula for $B(n)$.

For some problems excessive recursion is necessary (just because the structure to be generated is rather large).

Hanoi Tower Problem: You need to move a set of disks (enumerated $1, 2, \ldots, n$ from smallest to largest) from one peg to another, one disk at a time, while obeying the rule that a larger disk cannot be placed on top of a smaller disk. You have altogether three pegs: `from_peg` is the peg, where all the disks are placed originally (smallest disk 1 at the top); `to_peg` is the peg, where these disks must end up at the very end. And there is also `aux_peg` – auxiliary peg that can be used during the movements, but should be freed at the end.

Input: The parameter n; *Output:* A valid schedule describing valid movements of the disks.

Let $H(n)$ denote the running time of Hanoi tower problem. Express $H(n)$ (the number of disk movements in the algorithm) in terms of previous values $H(m)$, where $m < n$. Solve the recursion and find a closed formula for $H(n)$.

```python
def tower_of_hanoi(n, from_peg, to_peg, aux_peg):
    if n == 1:
        print("Move disk 1 from peg {} to peg {}".format(from_peg, to_peg))
        return

    tower_of_hanoi(n-1, from_peg, aux_peg, to_peg)
    print("Move disk {} from peg {} to peg {}".format(n, from_peg, to_peg))
    tower_of_hanoi(n-1, aux_peg, to_peg, from_peg)
```

Karatsuba Multiplication Algorithm: Given two non-negative integer numbers of the same length n (written in binary), write an algorithm to multiply these numbers. We need an algorithm that is faster than the "school algorithm" that multiplies two numbers of length n in $O(n^2)$ time.

Note: Here multiplication of long numbers cannot be done in constant time; instead you can assume that operations on individual bits can be done in constant time (Boolean logic, bit arithmetic, checking bits for condition statements).

2.4.1 Master Theorem

Master Theorem: Let $f(n)$ be an increasing function that satisfies the recurrence relation:

$$f(n) = a \cdot f\left(\frac{n}{b}\right) + cn^d$$

Here we assume that $n = b^k$, where k is a positive integer, $a \geq 1$, $b > 1$ is an integer, c, d are real numbers (where $c > 0$ and $d \geq 0$). Then the asymptotic growth for $f(n)$ can be found like this:

$$f(n) \text{ is in } \begin{cases} O(n^d), & \text{if } a < b^d, \\ O(n^d \log n), & \text{if } a = b^d, \\ O(n^{\log_b a}), & \text{if } a > b^d. \end{cases}$$

2.4.2 Lindenmayer Systems

2.5 Backtracking

N-Queens Problem: The task is to place N queens on an $N \times N$ chessboard such that no two queens threaten each other. This is the best known example of backtracking; you can place queens one by one and backtrack if a placement causes a conflict.

Input: Parameter N; *Output:* Any chess-board of size $N \times N$ with all N queens placed. (In fact, N-Queens is solvable for all $N \geq 4$, and for any such N some solutions are easy to get without any backtracking.)

Related decision Problem: Completion problem is a variant, in which some queens are already placed and the solver is asked, if it is possible to place the rest (the output of the decision problems is Yes/No). This problem is NP-complete. See I.P.Gent Complexity of n-Queens Completion..

The Traveling salesman problem (TSP): There exists a connected graph of cities, some cities are connected with roads of known lengths. It asks for the shortest possible "tour" that visits every city from a given set of cities exactly once and returns to the origin city. For small input sizes, TSP can be solved using a brute-force backtracking, where all possible paths are generated and their lengths are compared to find the shortest one.

Input: The input graph $G(V, E)$

Related decision problem: Given the length, find, if there exists a route less or equal than the given length. This problem is NP-complete.

The Subset Sum Problem: It asks if a given set of numbers can be divided into two subsets such that the sum of numbers in one subset is equal to a given target. This problem can be solved using backtracking by generating all possible subsets and checking if any of them have the desired sum.

Sudoku Solver: The task is to fill in a 9×9 grid with digits so that each column, each row, and each of the nine 3×3 sub-grids contains all of the digits from 1 to 9. Backtracking can be used to solve this problem by trying each possible digit in a cell and backtracking if it leads to an invalid solution.

Input: Partially filled in array of size 9×9; *Output:* Completed array of size 9×9. (It is often assumed that the input array is such that there exists exactly one solution. For backtracking it does not matter – it is possible to find any feasible solution, or all feasible solutions, or find out that there is no solution.)

Generating Permutations: The task is to generate all possible permutations of a given set of elements. Backtracking can be used to generate permutations by fixing elements one by one and swapping them to generate new permutations.

Maze Generation: The task is to generate a random maze using backtracking. In this problem, you can start at a random cell and move to unvisited cells, marking them as visited, until you have visited all cells. If you reach a dead end, you backtrack to the previous cell.

Cryptarithmetic Puzzles: The task is to solve puzzles where a mathematical expression is written using words and each letter represents a unique digit. Backtracking can be used to solve these puzzles by trying different values for each letter and backtracking if a solution leads to a conflict.

Backtracking algorithms are not hard to implement – they do not do much more besides an exhaustive search in a large tree representing the space of potential solutions (potentially very inefficient). Nevertheless, it is desirable to

2.6 Solving Asymptotic Bounds Exercises

In C++ the computer program is easy to imagine being run on real hardware (measure the runtime with the calls to system time). For Python or pseudocode it is more complicated. For example, package *numpy* offers different integers (4 byte long) compared to Python's default integer numbers (unlimited size). All this can get complicated.

2.6.1 Model of computation

We often cannot list all the assumptions regarding the runtime, therefore we can state, how Python code can be analyzed:

- Start with the Word-RAM model. Machine word: block of w bits.
- Operations can be performed in $O(1)$ time – operations on words: Integer arithmetic: (`+`, `-`, `*`, `//`, `%`), logical operators, bitwise arithmetic, input/output.
- Memory address must be able to access every place in memory 32-bit words can address 4 GiB memory, 64-bit words can address 16 exabytes of memory. (One exabyte is 10^{18} or one quintillion bytes.)

C++, Python and other languages commonly use external calls (if we know the complexity of some library call such as "sort", we can apply it). There are some predefined data structures in Python (and STL data structures in C++):

- Arrays, Lists, Sets, Dictionaries are used to store non-constant data. Each data structure supports a set of operations. A collection of operations is called an *interface* (for well-known data structures also ADT - *Abstract Data Type*).
- Example data structure: Static Array – fixed width slots, fixed length of the array itself. Its functions supported in pseudocode:
  - $A = \textsc{Array}(n)$: allocate static array of size n in $\Theta(n)$ time
  - $\textsc{Array}.get(i)$: return word stored at array index i in $\Theta(1)$ time
  - $\textsc{Array}.set(i, x)$: write value x to array index i in $\Theta(1)$ time

  In many languages it is common to write "get" and "set" commmands with array notation $A[i]$.
- Example data structure: List – same as above, but no longer fixed length. If it is implemented as a physical array, the operation times do not change. But occasionally need to reallocate memory, if the number of elements exceeds the size of the current array.

2.7 Problems

Problem 1: Answer the following questions regarding the asymptotic behavior of functions.

(A) Have students generate 10 functions and order them based on asymptotic growth.

(B) Find a tight asymptotic bound for $\binom{n^2}{3168}$, and write it using the simplest notation possible.

(C) Find a simple, tight asymptotic bound for $f(n) = \log_2\left(\sqrt{n}^{\sqrt{n}}\right) - \log_{10}\left(\sqrt[3]{n}^{\sqrt[3]{n}}\right)$.

(D) Is 2^n in $\Theta(3^n)$? Is $2^{2^{n+1}}$ in $\Theta\left(2^{2^n}\right)$?

(E) Show that $(\log n)^a$ is in $O(n^b)$ for all positive constants a and b.

(F) Let $f(n) = (\log_2 n)^{\sqrt{n}}$ and $g(n) = (\log_{10} n)^{\sqrt{n}}$. Is $f(n)$ in $\Theta(g(n))$?

(G) Show that $(\log n)^{\log n}$ is in $\Omega(n)$.

(H) Is $(2n)!$ in $O(n!)$? Is $\sqrt{(2n)!}$ in $O(\sqrt{n!})$? Is $\sqrt{\log_2((2n)!)}$ in $\sqrt{\log_2(n!)}$

Problem 2: Consider Euclid algorithm to find the greatest common divisor (written around 300 B.C. in *Elements*):

```
EuclidGCD($a, b$)
1.  if $b$ == 0:
2.      return $a$
3.  else:
4.      return EuclidGCD($b, a$ mod $b$)
```

It is known that for a given input length n the worst-case running time is to run the algorithm on subsequent Fibonacci numbers: F_m and F_{m-1}, where F_m is the largest Fibonacci number of length not exceeding n.

Write a precise estimate (without using unknown constant factors as in Big-O notation) on how many calls of $\textsc{EuclidGCD}(a, b)$ are needed, if both inputs have length not exceeding n.

Note: Imagine that both arguments to the Euclid algorithm are two natural numbers a, b containing up to 100 digits each. Estimate the maximum number of recursive calls until the grater common divisor is found.

Problem 3: Given a sequence a_i ($i = 0, \ldots, n-1$) we call its element a_i a *peak* iff it is a local maximum (at least as big as any of its neighbors):

$$a_i \geq a_{i-1} \text{ and } a_i \geq a_{i+1}$$

(In case if $i = 0$ or $i = n-1$, one of these neighbors does not exist; and in such cases we only compare a_i with neighbors that do exist.)

(A) Suggest an algorithm to find some peak in the given array $A[0], \ldots, A[n-1]$ and find its worst-case running time.

(B) Suggest an algorithm that is faster than linear time to find peaks in an array. Namely, its worst-case running time should satisfy the limit:

$$\lim_{n \to \infty} \frac{T(n)}{n} = 0.$$

Question 4: Select the correct asymptotic complexity of an algorithm with runtime $T(n, n)$ where

$$\begin{cases} T(x,c) = \Theta(x) \text{ for } c \leq 2, \\ T(c,y) = \Theta(y) \text{ for } c \leq 2, \text{ and}, \\ T(x,y) = \Theta(x+y) + T(\lfloor x/2 \rfloor, \lfloor y/2 \rfloor) \text{ otherwise.} \end{cases}$$

a. $\Theta(\log n)$.

b. $\Theta(n)$.

c. $\Theta(n \log n)$.

d. $\Theta(nlog^2 n)$.

e. $\Theta(n^2)$.

f. $\Theta(2^n)$.

Question 5: Just like the tail-recursive factorial, write a tail-recursive Fibonacci program. This way you will also avoid excessive recursion – exponential increase of the number of recursive calls.

To achive this, you may need to pass multiple parameters in the recursive call to the recursive Fibonacci function.