WORKSHEET WEEK 02: RECURSION

There is a proverb: *To understand recursion, you must first understand recursion.* Recursion allows to analyze algorithms, to find their runtime even in complex cases.

2.1 Concepts and Facts

Definition: A Fibonacci sequence is defined by the following recurrent formula:

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

Statement: For every nonnegative integer n the Fibonacci number $F(n)$ can be computed by the following expression:

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

Such expressions are named *closed formulas*, since they can be evaluated directly, without repeating recurrent formula many times. See its proof – <https://brilliant.org/wiki/linear-recurrence-relations/>`_. The expression shows that (apart from an expression $((1-\sqrt{5})/2)^n$ that goes to 0) Fibonacci numbers grow as a geometric series.

Definition: A factorial for any non-negative integer is defined by the following recurrent formula:

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

Statement: Factorials for large n satisfy Stirlings approximation:

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

Note: The asymptotic equality $f(n) \sim g(n)$ for two positive functions $f(n), g(n)$ denotes that the ratio $f(n)/g(n)$ has limit 1 as $n \to \infty$. Such relation also would imply that $f(n)$ and $g(n)$ are in Big-Theta of each other. In fact, $f(n) \sim g(n)$ is stronger than Big-Theta, since $f(n) \in \Theta(g(n))$ does allow any finite and bounded ratios $f(n)/g(n)$.

Definition: Function code that invokes itself (with different arguments to avoid infinite loops) is called a *function implemented with recursion.*

Example: This is a straightforward recursive implementation of factorial:

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

For large n this implementation is problematic, since it keeps too many function calls on the activation stack. Therefore in practice factorial is usually implemented without recursion (one can multiply the numbers in a loop), or using a tail recursion.

Definition: Function code that contain recursive call in just one place and the recursive call is the entire return expression of the function is called *function implemented with tail-recursion*.

Example: Here is tail-recursive factorial:

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

Algorithms using divide-and-conquer paradigm (such as MergeSort, Quicksort) may call themselves multiple times. In these cases we must be very careful regarding the depth of recursion tree as the number of function calls grows exponentially. Here is an extremely inefficient algorithm to compute Fibonacci numbers: any call to fibonacci_bad(n) (where $n \geq 2$) leads to two more calls. Such implementations are called *excessive recursion*.

Example:

```
def fibonacci_bad(n):
    if n <= 1:
        return n
    else:
        return fibonacci_bad(n - 1) + fibonacci_bad(n - 2)
```

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:

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

This theorem can be used to find the asymptotic runtime for recursive algorithms.

Example (Binary Search): Binary search searches items in a sorted array of n elements: $A[0] < A[1] < \ldots < A[n-2] < A[n-1]$. At every point it maintains a search interval $[\ell, r]$ so that the searchable item w satisfies inequalities $D[\ell] < w < D[r]$. The initial call is $\textsc{BinarySearch}(A, 0, n-1, w)$. After that the binary search calls itself recursively on shorter intervals:

```
BINARYSEARCH($D, \ell, r, w$)
1.  if $\ell > r$:
2.      return NOT FOUND $w$
```

```
3.    $m = \lfloor (\ell + r)/2 \rfloor$
4.    **if** $w == D[m]$:
5.        **return** FOUND $w$ at location $m$
6.    **else if** $w < D[m]$:
7.        **return** BINARYSEARCH$(D, \ell, m-1, w)$
8.    **else**:
9.        **return** BINARYSEARCH$(D, m+1, r, w)$
```

Denote the runtime of this algorithm on an array of length n by $T(n)$. Denote by a constant K the upper bound of the time necessary to compute the middlepoint m on Line 3 and to do all the comparisons. Use the Masters theorem to find the time complexity for $T(n)$.

Answer:

The runtime $T(n)$ satisfies the following recurrence:

$$T(n) = \begin{cases} K, & \text{if } n = 1, \\ K + T(\lfloor n/2 \rfloor), & \text{if } n > 1. \end{cases}$$

In the Masters theorem $a = 1$, $b = 2$, $c = K$, $d = 0$. Since $a = b^d$, we have $T(n) \in O(n^d \log n) = O(\log n)$. We conclude that the BINARYSEARCH$(\ldots)$ algorithm runs in logarithmic time $O(\log n)$, where n is the length of the array.

Example (Hanoi Tower): 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.

```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)
```

The input of this algorithm is n (the number of disks), its output is a valid schedule describing valid movements of the disks. Let $H(n)$ denote the running time of this algorithm expressed as the number of `print()` statements. 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 $H(n)$.

Answer:

The runtime $H(n)$ satisfies the following recurrence:

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

In this case Masters theorem cannot be applied, since $H(n)$ is expressed via $H(n-1)$ rather than in terms of $H(n/b)$ for some constant b. Fortunately, $H(n)$ can be evaluated directly.

Observe that $H(1) = 1$, $H(1) = 1 + 2 \cdot H(1) = 3$, $H(3) = 1 + 2 \cdot H(2) = 7$, and so on. We observe that $H(n) = 2^n - 1$. This can be proven by induction.

Base: $n = 1$. In this case $H(1) = 1$, and also $H(1) = 2^1 - 1$; so the formula $H(n) = 2^n - 1$ is true in this case.

Inductive Hypothesis: Assume that for $n = k$ disks the number of print statements is indeed $H(k) = 2^k - 1$.

Induction Step: We now prove that the statement is also true for $n = k + 1$. In this case $H(k + 1) = 1 + 2 \cdot H(k)$ by the given recurrent formula. On the other hand, by inductive hypothesis,

$$H(k+1) = 1 + 2 \cdot (2^k - 1) = 1 + 2 \cdot 2^k - 2 = 2^{k+1} + 1 - 2 = 2^{k+1} - 1.$$

We now see that $H(k + 1) = 2^{k+1} - 1$, which means that the formula $H(n) = 2^n - 1$ is also true for $n = k + 1$. Inductive step is completed.

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

```
KARATSUBA($n_1, n_2$)
1.    if ($n_1 < 10$) **or** ($n_2 < 10$):
2.        **return** $n_1 \cdot n_2$      (*fall back to traditional multiplication*)
3.    $m = \max(\text{SIZE}(n_1), \text{SIZE}(n_2))$
4.    $m_2 = \lfloor m/2 \rfloor$
5.    $h_1, \ell_1 = \text{SPLITAT}(n_1, m_2)$
6.    $h_2, \ell_2 = \text{SPLITAT}(n_2, m_2)$
      (*Three recursive calls of Karatsubas algorithm.*)
7.    $z_0 = \text{KARATSUBA}(\ell_1, \ell_2)$
8.    $z_1 = \text{KARATSUBA}(\ell_1 + h_1, \ell_2 + h_2)$
9.    $z_2 = \text{KARATSUBA}(h_1, h_2)$
12.    $y = z_1 - z_2 - z_0$
13.    **return** $(z_2 \cdot 10^{2 \cdot m_2}) + (y \cdot 10^{m_2}) + z_0$
```

By $T(n)$ denote the runtime of Karatsubas algorithm on two numbers having length n each. (Assume that non-recursive parts take some constant time K.) Provide the asymptotic bound extimate for $K(n)$.

Note: We typically assume that addition and multiplication take $\Theta(1)$ time as they are CPU operations. But multiplication of very long numbers cannot be done in constant time. Instead assume that operations on individual bits are done in constant time. Things like Boolean operations, bit arithmetic, checking conditional statements.

Answer:

In this algorithm one call causes three recursive calls; each call has arguments that are half the size. It means that $T(n) = 3 \cdot T(n/2) + K$.

In Masters theorem we would have $a = 3$, $b = 2$, $c = K$, $d = 0$. In this case $a > b^d$, so $T(n) \in O\left(n^{\log_b a}\right) = O\left(n^{\log_2 3}\right)$. So the time complexity of this is $O(n^{1.58})$ which is significantly better than $O(n^2)$.

2.2 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($b, a \bmod b$)
```

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 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.

Answer:

```
# Tail Recursive function to calculate nth Fibonacci number
def fibonacci_tail(n, a, b) -> int:
    if n == 0:
        return a
    else:
        return fibonacci_tail_recursive(n - 1, b, a + b)

# Shows how to initialize the function's fibonacci_tail(...) arguments:
def fibonacci_tail(n: int) -> int:
    return fibonacci_tail_recursive(n, 0, 1)
```

Question 6: It is known that Taylor series for $y = \sin x)$ is given by formula:

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

The series converges for every $x \in \mathbb{R}$. Write a tail-recursive function that for any argument x computes the approximation for $\sin x$ by adding up the first 50 terms of the Taylor series. The use of global variables is not allowed – all data manipulation should be done with local variables function calls and their return values. Your solution should use as few multiplications and divisions as possible.

Answer:

As the global variables are not allowed, we can accumulate the partial sum as one of the arguments passed to the recursive method calls.

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

For a constant x build the sequence $S(0, x), S(1, x), S(2, x), \ldots$ using the recursive calls. You can write a recursive function in pseudocode that computes $S(50, x)$.

This answer is incomplete as the number of multiplications and divisions is not minimized. In fact, computing x^{2n-1} would require many multiplications; same as $(2n - 1)!$. So you can try to optimize this further.