Worksheet 2 – Algorithms

This worksheet has two parts:

Part one: Measuring time of execution and memory allocation

# Part one - Task 1

In this task we will use two functions to calculate Fibonacci number; one is iterative and the other one is recursive.

1. Revise the difference between iterative and recursive functions, here is a link to help you:

[What is the difference between iteration and recursion? (afteracademy.com)](https://afteracademy.com/blog/what-is-the-difference-between-iteration-and-recursion)

Or use your notes.

1. What are the advantages and disadvantages of using iteration?

* An advantage of using iteration is that the execution of the code is faster than a recursive program of the same type.
* A disadvantage of using iteration is that the code tends to be bigger in size.

1. What are the advantages and disadvantages of using recursion?

- An advantage of recursion is that the program is broken down into simpler parts therefore it is easier to find a solution.

- A disadvantage of recursion is that an infinite recursion will call a stack overflow or might crash the system.

1. Let us compare between a recursive and iterative functions using Fibonacci number. We will measure time of execution for each function and the memory allocation.
2. We need to change the value of Fibonacci number and measure time and memory allocation.

# Part one - Task 2

Use the code below

**import** tracemalloc

**import** timeit

*# this function calculates the nth Fibonacci Number using iteration*

**def** FibonacciItr(pos):

counter = 0

first = 0

second = 1

n = 0

**while** counter <= pos:

n = first + second

first = second

second = n

counter = counter + 1

**return** n

*# this function calculates the nth Fibonacci Number using recursion*

**def** FibonacciRec( pos ):

**if** pos <= 1 :

**return** 0

**if** pos == 2:

**return** 1

*#the function calls itself*

n = FibonacciRec( pos-1 ) + FibonacciRec( pos-2 )

**return** n

*# here are two functions to measure time and memory*

**def** memMesure(num):

tracemalloc.start()

*# call Fibonacci Function either recrsive or iterative*

nth\_fibo = FibonacciItr(num)

current, peak = tracemalloc.get\_traced\_memory()

**print**(f"Current memory usage is {current / 10\*\*3}KB; Peak was {peak / 10\*\*3}KB")

tracemalloc.stop()

**def** timeMeasure(num):

start = timeit.default\_timer()

nth\_fibo = FibonacciItr(num)

end = timeit.default\_timer()

t = (end -start)\*1000000 # time in microseconds. To convert it milliseconds multiply times 1000, and for second multiply by 1

**print**("Time of execution is " + str(t)+ " microseconds")

**print**("This program calculates the nth Faibonacci number and prints time it takes and memory used")

number = int(input("Enter the nth Fibonacci number "))

memMesure(number)

timeMeasure(number)

2. make sure **you understand the code** first before you start working.

3. Run the code so that you can get data to complete the following tables:

|  |  |  |
| --- | --- | --- |
| Iterative Fibonacci Function | | |
| number | Time in (microseconds) | Memory allocation in KB |
| 10 | 1.3 | 0.0KB |
| 20 | 2.1 | 0.038KB: Peak 0.084KB |
| 40 | 3.2 | 0.028 |
| 80 | 8.0 | 0.036 |
| 160 | 11.1 | 0.044 |
| 320 | 23.3 | 0.06 |
| 640 | 49.3 | 0.088KB |
| 1280 | 129.7 | 0.148KB |
| 2560 | 366.4 | 0.268KB |
| 5120 | 574.0 | 0.504KB |
| 10240 | 2006.4 | 0.976KB |
| 20480 | 6309.2 | 1.924KB |

|  |  |  |
| --- | --- | --- |
| Recursive Fibonacci Function | | |
| number | Time in (milliseconds / microseconds) delete | Memory allocation in KB |
| 5 |  |  |
| 10 |  |  |
| 15 |  |  |
| 20 |  |  |
| 25 |  |  |
| 30 |  |  |
| 35 |  |  |
| 40 |  |  |
|  |  |  |

Note using values more than 40 could crash the machine, make sure you use reasonable values.

# Part one - Task 3

Use Python, plot the graphs of number vs time and number vs memory allocation. Number is the x-axis always. Use this website to help you with the code

<https://www.geeksforgeeks.org/graph-plotting-in-python-set-1/>

# Part one - Task 4 – Analysis of the results

Analyse your graph by answering the following questions:

1. What kind of relationships between number and time?
2. What kind of relationship between number and memory allocation?
3. Why does the computer crash when number is in region of 40 when you used recursive function, while it didn’t when you use iterative function?
4. When do you recommend the use of iterative function ?
5. When do you recommend to use recursive function?

Part Two: Tower of Hanoi

The Tower of Hanoi is a mathematical puzzle. It consists of three poles and a number of disks of different sizes which can slide onto any poles. The puzzle starts with the disk in a neat stack in ascending order of size in one pole, the smallest at the top thus making a conical shape. The objective of the puzzle is to move all the disks from one pole (say ‘source pole’) to another pole (say ‘destination pole’) with the help of the third pole (say auxiliary pole).

The puzzle has the following two rules:  
      1. You can’t place a larger disk onto a smaller disk   
      2. Only one disk can be moved at a time



You can read more about Tower of Hanoi from: <https://www.mathsisfun.com/games/towerofhanoi.html>

There are two types of solutions to Tower Of Hanoi Problem; iterative solution and recursive solution. In this part we will compare between the two solutions.

**Iterative Algorithm:**

This solution is more difficult, and summarised as follows:

1. Calculate the total number of moves required i.e. "pow(2, n)

- 1" here n is number of disks.

2. If number of disks (i.e. n) is even then interchange destination

pole and auxiliary pole.

3. for i = 1 to total number of moves:

if i%3 == 1:

legal movement of top disk between source pole and

destination pole

if i%3 == 2:

legal movement top disk between source pole and

auxiliary pole

if i%3 == 0:

legal movement top disk between auxiliary pole

and destination pole

This solution is harder to program; therefore the program is given to you (2 pages of code). Note you are not expected to know how to write this program:

*# Python3 program for iterative Tower of Hanoi*

**import** sys

*# A structure to represent a stack*

**class** Stack:

*# Constructor to set the data of*

*# the newly created tree node*

**def** \_\_init\_\_(self, capacity):

self.capacity = capacity

self.top = -1

self.array = [0]\*capacity

*# function to create a stack of given capacity.*

**def** createStack(capacity):

stack = Stack(capacity)

**return** stack

*# Stack is full when top is equal to the last index*

**def** isFull(stack):

**return** (stack.top == (stack.capacity - 1))

*# Stack is empty when top is equal to -1*

**def** isEmpty(stack):

**return** (stack.top == -1)

*# Function to add an item to stack.*

*# It increases top by 1*

**def** push(stack, item):

**if**(isFull(stack)):

**return**

stack.top+=1

stack.array[stack.top] = item

*# Function to remove an item from stack.*

*# It decreases top by 1*

**def** Pop(stack):

**if**(isEmpty(stack)):

**return** -sys.maxsize

Top = stack.top

stack.top-=1

**return** stack.array[Top]

*# Function to implement legal*

*# movement between two poles*

**def** moveDisksBetweenTwoPoles(src, dest, s, d):

pole1TopDisk = Pop(src)

pole2TopDisk = Pop(dest)

*# When pole 1 is empty*

**if** (pole1TopDisk == -sys.maxsize):

push(src, pole2TopDisk)

moveDisk(d, s, pole2TopDisk)

*# When pole2 pole is empty*

**elif** (pole2TopDisk == -sys.maxsize):

push(dest, pole1TopDisk)

moveDisk(s, d, pole1TopDisk)

*# When top disk of pole1 > top disk of pole2*

**elif** (pole1TopDisk > pole2TopDisk):

push(src, pole1TopDisk)

push(src, pole2TopDisk)

moveDisk(d, s, pole2TopDisk)

*# When top disk of pole1 < top disk of pole2*

**else**:

push(dest, pole2TopDisk)

push(dest, pole1TopDisk)

moveDisk(s, d, pole1TopDisk)

*# Function to show the movement of disks*

**def** moveDisk(fromPeg, toPeg, disk):

*# recommended that you use pass instead of the print statement below*

**print**("Move the disk", disk, "from '", fromPeg, "' to '", toPeg, "'")

*# Function to implement TOH puzzle*

**def** tohIterative(num\_of\_disks, src, aux, dest):

s, d, a = 'S', 'D', 'A'

*# If number of disks is even, then interchange*

*# destination pole and auxiliary pole*

**if** (num\_of\_disks % 2 == 0):

temp = d

d = a

a = temp

total\_num\_of\_moves = int(pow(2, num\_of\_disks) - 1)

*# Larger disks will be pushed first*

**for** i **in** range(num\_of\_disks, 0, -1):

push(src, i)

**for** i **in** range(1, total\_num\_of\_moves + 1):

**if** (i % 3 == 1):

moveDisksBetweenTwoPoles(src, dest, s, d)

**elif** (i % 3 == 2):

moveDisksBetweenTwoPoles(src, aux, s, a)

**elif** (i % 3 == 0):

moveDisksBetweenTwoPoles(aux, dest, a, d)

*# Input: number of disks*

num\_of\_disks = int(input("Enter number of disks "))

*# Create three stacks of size 'num\_of\_disks'*

*# to hold the disks*

*# you need to call all of the functions below to get the solution*

src = createStack(num\_of\_disks)

dest = createStack(num\_of\_disks)

aux = createStack(num\_of\_disks)

tohIterative(num\_of\_disks, src, aux, dest)

# Part 2 - Task - 1

1. Copy the code into Python file, then test it.
2. Use the code from part 1 (Fibonacci number) , write two more functions; one to measure the time of execution, and the other to measure the memory allocation.
3. Measure the time and memory allocation for different value of disks, then complete the table below:

|  |  |  |
| --- | --- | --- |
| Tower of Hanoi – Iterative solution | | |
| Number of disks | Execution time | Memory allocation |
| 3 |  |  |
| 6 |  |  |
| 12 |  |  |
| 24 |  |  |
| 48 |  |  |
| 96 |  |  |
| 196 |  |  |
|  |  |  |
|  |  |  |

Be reasonable when you choose the number of disks;

**Recursive Algorithm:**

Tower of Hanoi can be solved much easier using recursion; the recursive algorithm is given below:

1. Shift 'n-1' disks from 'A' to 'B'.
2. Shift last disk from 'A' to 'C'.
3. Shift 'n-1' disks from 'B' to 'C'.

The function will call itself until we reach the last disks.

Here is Python Code:

*# Recursive Python function to solve tower of hanoi*

**def** TowerOfHanoi(n , from\_rod, to\_rod, aux\_rod):

**if** n == 0:

**return**

TowerOfHanoi(n-1, from\_rod, aux\_rod, to\_rod)

**print**("Move disk",n,"from rod",from\_rod,"to rod",to\_rod)

TowerOfHanoi(n-1, aux\_rod, to\_rod, from\_rod)

*# Driver code*

n = int(input("Enter number of disks "))

TowerOfHanoi(n, 'A', 'C', 'B')

*# A, C, B are the name of rods*

Clearly recursive solution is much easier to program. However we need to measure execution time and memory allocation so that we can compare the two solutions.

# Part 2- Task - 2

1. Copy the code into Python file, then test it.
2. Use the code from part 1 (Fibonacci number) or from part 2 task 1, write two more functions; one to measure the time of execution, and the other to measure the memory allocation.
3. Measure the time and memory allocation for different value of disks, then complete the table below:

|  |  |  |
| --- | --- | --- |
| Tower of Hanoi – Recursive solution | | |
| Number of disks | Execution time | Memory allocation |
| 3 |  |  |
| 6 |  |  |
| 12 |  |  |
| 24 |  |  |
| 48 |  |  |

Be reasonable when you choose the number of disks larger value can crash the computer ot takes too much time to complete;

# Part 2 – Task 3

Use Python, plot the graphs of number vs time and number vs memory allocation. Number is the x-axis always. Use this website to help you with the code

<https://www.geeksforgeeks.org/graph-plotting-in-python-set-1/>

# Part 2 – Task 4

Compare between the two algorithms

1. What kind of relationships between number of disks and time?
2. What kind of relationship between number of disks and memory allocation?
3. Why does some computers crash when number is in region of a specific value (e.g. 40) when you used recursive function, while it didn’t when you use iterative function?
4. When do you recommend the use of iterative function?
5. Explain why programmers prefer to use recursion when writing solutions to problems like Tower of Hanoi.