Cairo University
Faculty of Engineering
Credit Hour System

SBEN429 Biomedical Data Analytics
Assignment 1

Name: **Sandra Adel Aziz Gebraiel**

ID: **1180059**

Date: **14/11/2021**

Assignment 1.1 Fibonacci Number:

1) Explanation of Code:

Time Complexity of naive: O(n), with n up to 10^18

Optimized Algorithm: According to Pisano table, F(n) % m = F(n % Pisano period) % m, So, instead of finding Fn, we find F(n % pisano period). Pisano period is recognized by starting with 01, with being even (except for 2) and is up to m x m

OVERALL TIME COMPLEXITY: O(m^2), with m up to 10^3

Practically at maximum m identified, it is way less than that, as we break when the period is found, and the pisano period of m^3 = 1500, with stepping by 2, making the result 750 iterations

2) Pseudocode:

```
GetFibonacciHugeFast(n,m):

Initialize PisanoPeriod with zero

If m is 2:
    Assign PisanoPeriod with 3 (only odd Pisano period)

Otherwise:
    Loop on series of Fibonacci numbers modulus m
    with step 2 (searching for an even Pisano period):
        If a second occurrence of 0 and 1 is found (signifies start of new period):
            Assign PisanoPeriod with number of iterations from the first 0 to the last 1
            (as pisano period ends with 1), and break out of loop
            (no need to continue to m x m)

Assign remainder with n modulus PisanoPeriod

Loop on series of Fibonacci numbers until F(remainder)

Return F(remainder) modulus m
```

3) Naïve Algorithm and Stress Testing Code:

```python
def get_fibonacci_huge_naive(n, m):

    if n <= 1:
        return n

    previous = 0
    current  = 1

    for _ in range(n - 1):
        previous, current = current, previous + current

    return current % m

if __name__ == '__main__':
    #input_numbers = [int(x) for x in input().split()]
    #n, m = input_numbers
    #print(get_fibonacci_huge_fast(n,m))

    while 1:
            n = r.randint(1, 10000)
            m = r.randint(2, 1000)
            print( str(n) + ' ' + str(m) )

            res1 = get_fibonacci_huge_naive(n,m)
            res2 = get_fibonacci_huge_fast(n,m)

            if res1 != res2:
                print('wrong answer'+ ' ' + str(res1) + ' ' + str(res2))
            else:
                print('OK')
```

4) Resources:

Pisano period - Wikipedia

Pisano Period -- from Wolfram MathWorld

Pisano Period - Tables (liquisearch.com)

Assignment 1.2 Maximum Advertisement Revenue

1) Explanation of Code:

Naive Algorithm: Loop in n (number of advertisments), and in each iteration, loop to find the slot with the maximum average number of clicks per day and the maximum profit per one click on ad, and add their product to the total revenue to be found

Time Complexity of naive: O(n^2)

Optimized Algorithm: if the arrays of average number of clicks per day of slots, and profit per each click on ad are sorted descendingly, then the first element in both arrays will always be the maximum, as we loop on n (number of advertisments)

Complexity of greedy part of optimized algorithm: O(n)

Sorting: To sort, we use merge sort algorithm a it has a fairly better time complexity than other sorting algorithms, and a more stable one than quick sort, although the latter has better space complexity

Merge Sort: Break down array in smaller subarrays (of one element)
using recursion → O(logn)
Sort the subarray elements ascendingly while working the way up and merging them again → O(n)

Time Complexity of merge sort: O(nlogn)

OVERALL TIME COMPLEXITY OF OPTIMIZED ALGORITHM: O(nlogn)

2) Pseudocode:

```
MaxDotProductFast(a,b):

Assign SortedA with output of MergeSort(a) and SortedB with output of MergeSort(b)

Intialize Result with zero

Loop on SortedA and SortedB:

    Multiply each two opposite elements in arrays SortedA and SortedB
    and add the product to Result

Return Result
```

```
MergeSort(array):

If array has only one element:
    Return the array
Assign middle position of array to Median
Break down array into two subarrays (FirstHalf and SecondHalf) from the median
Assign MergedSortedArray with output of Merge(FirstHalf, SecondHalf)
Return MergedSortedArray


Merge(FirstArr, SecondArr):

Initialize FirstArrayPointer and SecondArrayPointer with zero
Initialize an empty array named SortedArray
While both pointers have not exceeded the last element of both arrays:
    If the element pointed to by FirstArrayPointer is less than or equal
    that pointed to by SecondArrayPointer:
        Copy element pointed to by FirstArrayPointer to the end of SortedArray
         Increment FirstArrayPointer
    Otherwise:
        Copy element pointed to by SecondArrayPointer to the end of SortedArray
         Increment SecondtArrayPointer
If FirstArrayPointer has not yet exceeded last element of FirstArr:
    Copy the remaining elements not exceeded in FirstArr to SortedArray
If SecondArrayPointer has not yet exceeded last element of SecondArr:
    Copy the remaining elements not exceeded in SecondArr to SortedArray
Return SortedArray
```

3) Calculation of Big-O:

MaxDotProductFast(a,b): → **O(nlogn)**

```
Assign SortedA with output of MergeSort(a) and SortedB
with output of MergeSort(b) → O(nlogn)
Intialize Result with zero → O(1)
Loop on SortedA and SortedB: → O(n)
    Multiply each two opposite elements in arrays SortedA and SortedB
    and add the product to Result → O(1)
Return Result → O(1)
```

MergeSort(Array): → **O(nlogn)**

The n elements of Array are looped, and each subarray/array is halved logn times

```
If Array has only one element:
    Return the array → O(1)
Assign middle position of array to Median → O(1)
Assign FirstHalf with output of MergeSort(Array[start to median]), and SecondHalf with
ouput of MergeSort(Array[median+1 to end]) recursive calls → O(nlogn)
Assign MergedSortedArray with output of Merge(FirstHalf, SecondHalf) → O(n)
Return MergedSortedArray → O(1)
```

Merge(FirstArr, SecondArr): → **O(n)**

The n elements of FirstArr + SecondArr are looped

```
Initialize FirstArrayPointer and SecondArrayPointer with zero → O(1)
Initialize an empty array named SortedArray → O(1)
While both pointers have not exceeded the last element of both arrays: → O(n)
    If the element pointed to by FirstArrayPointer is less than or equal
    that pointed to by SecondArrayPointer:
        Copy element pointed to by FirstArrayPointer to the end of SortedArray → O(1)
        Increment FirstArrayPointer → O(1)
    Otherwise:
        Copy element pointed to by SecondArrayPointer to the end of
        SortedArray → O(1)
        Increment SecondtArrayPointer → O(1)
If FirstArrayPointer has not yet exceeded last element of FirstArr:
    Copy the remaining elements not exceeded in FirstArr to SortedArray → O(n)
If SecondArrayPointer has not yet exceeded last element of SecondArr:
    Copy the remaining elements not exceeded in SecondArr to SortedArray → O(n)
Return SortedArray → O(1)
```

4) Naïve Algorithm and Stress Testing Code:

```python
def max_dot_product_naive(a, b):

    n = len( b )
    b_filled = []
    a_taken = []
    res = 0
    for i in range(n):

        max_b = float('-inf')
        max_b_index = -1
        j = 0
        for j in range(n):
            if b[j] >= max_b and (j not in b_filled):
                max_b = b[j]
                max_b_index = j
        b_filled += [max_b_index]

        max_a = float('-inf')
        max_a_index = -1
        k = 0
        for k in range(n):
            if a[k] >= max_a and (k not in a_taken):
                max_a = a[k]
                max_a_index = k
        a_taken += [max_a_index]

        res += max_a * max_b

    return res
```

```python
if __name__ == '__main__':
    '''
    input = sys.stdin.read()
    data = list(map(int, input.split()))
    n = data[0]
    a = data[1:(n + 1)]
    b = data[(n + 1):]
    print(max_dot_product_fast(a, b))
    '''


    while 1:
            a = []
            b = []
            n = r.randint(1, 100)
            print(n)
            for i in range(n):
                a.append(r.randint(-100000, 100000))
            print(a)
            for i in range(n):
                b.append(r.randint(-100000, 100000))
            print(b)

            res1 = max_dot_product_naive(a,b)
            res2 = max_dot_product_fast(a,b)

            if res1 != res2:
                print('wrong answer'+ ' ' + str(res1) + ' ' + str(res2))
                break
            else:
                print('OK')
```

5) Resources:

MOOC: Algorithmic Toolbox Slides - Week 4