Tutorial (week 3)

All the answers can be given with pseudocode or with Python. Remember that most of the time, many solutions are possible.

1. Order the following list of functions by the big-Oh notation.

$6n\log n$	2^{100}	$\log\log n$	$\log^2 n$	2^{logn}
2^{2^n}	$\lceil\sqrt{n}\rceil$	$n^{0.01}$	$1/n$	$4n^{3/2}$
$3n^{0.5}$	$5n$	$\lfloor 2n\log^2 n\rfloor$	2^n	$n\log_4 n$
4^n	n^3	$n^2\log n$	$4^{\log n}$	$\sqrt{\log n}$

Solution:
When in doubt about two functions $f(n)$ and $g(n)$, consider $\log f(n)$ and $\log g(n)$ or $2^{f(n)}$ and $2^{g(n)}$.

$$1/n,\ 2^{100},\ \log\log n,\ \sqrt{\log n},\ \log^2 n,\ n^{0.01},\ \lceil\sqrt{n}\rceil,\ 3n^{0.5},\ 2^{logn},\ 5n,\ n\log_4 n$$
$$6n\log n,\ \lfloor 2n\log^2 n\rfloor,\ 4n^{3/2},\ 4^{\log n},\ n^2\log n,\ n^3,\ 2^n,\ 4^n,\ 2^{2^n}$$

2. Give a big-Oh characterization, in terms of n, of the running time of the `Loop1`, `Loop2`, `Loop3`, `Loop4` and `Loop5` methods shown below.

Loop 1

```python
def loop1(n):
    s = 0
    for i in range(n):
        s = s + i
```

Loop 2

```python
def loop2(n):
    p = 1
    for i in range(2*n):
        p = p * i
```

Loop 3

```python
def loop3(n):
    p = 1
    for i in range(n**2):
        p = p * i
```

Loop 4

```python
def loop4(n):
    s = 0
    for i in range(2*n):
        for j in range(i):
            s = s + i
```

Loop 5

```python
def loop5(n):
    s = 0
    for i in range(n**2):
        for j in range(i):
            s = s + i
```

Solution:
The Loop1 and Loop2 method run in $O(n)$ time, Loop3 and Loop4 method run in $O(n^2)$ time, while Loop5 runs in $O(n^4)$ time.

3. Show that $(n+1)^5$ is $O(n^5)$.

Solution:
By the definition of big-Oh, we need to find a real constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $(n+1)^5 \leq c(n^5)$ for every integer $n \geq n_0$. Since $(n+1)^5 = n^5 + 5n^4 + 10n^3 + 10n^2 + 5n + 1$, $(n+1)^5 \leq c(n^5)$ for $c = 8$ and $n \geq n_0 = 2$.

4. Show that 2^{n+1} is $O(2^n)$.

Solution:
By the definition of big-Oh, we need to find a real constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $2^{n+1} \leq c(2^n)$ for $n \geq n_0$. One possible solution is choosing $c = 2$ and $n_0 = 1$, since $2^{n+1} = 2 \cdot 2^n$.

5. Describe a recursive algorithm for finding both the minimum and the maximum elements in an array A of n elements. Your method should return a pair (a, b), where a is the minimum element and b is the maximum. What is the running time of your method ?

Solution:

```python
def min_and_max(A):
    '''
    INPUT: an array A of n elementsn
    OUTPUT: the minimum and the maximum elements of A
    '''

    if len(A)==1: return [A[0],A[0]]
    else:
        [a,b] = min_and_max(A[1:])
        return [min(A[0],a),max(A[0],b)]


print(min_and_max([4,5,87,9,12,43,5,-7,5]))
```

The running time is $O(n)$.

6. Consider the following recurrence equation, defining a function $T(n)$:

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ T(n-1) + n & \text{otherwise} \end{cases}$$

Show, by induction, that $T(n) = n(n+1)/2$.

Solution:
For $n = 1$ and $n = 2$, this is clearly true. Now assume $T(n) = n(n+1)/2$. Then, $T(n+1) = T(n) + n + 1 = n(n+1)/2 + n + 1 = (n+1)(n+2)/2$.

7. An array A contains $n - 1$ unique integers in the range $[0, n-1]$; that is, there is one number from this range that is not in A. Design an $O(n)$-time algorithm for finding that number. You are allowed to use only $O(log(n))$ additional space besides the array A itself.

   Solution:
   First calculate the sum $\sum_{i=1}^{n-1} = n(n-1)/2$. Then, calculate the sum of all values in the array A. The missing element is the difference between these two numbers.

8. Suppose that each row of an $n \times n$ array A consists of 1's and 0's such that, in any row of A, all the 1's come before any 0's in that row. Assuming A is already in memory, describe a method running in $O(n)$ time (not $O(n^2)$ time) for finding the row of A that contains the most 1's.

   Solution:
   Start at the upper left of the matrix. Walk across the matrix until a 0 is found. Then walk down the matrix until a 1 is found. This is repeated until the last row or column is encountered. The row with the most 1's is the last row which was walked across. Clearly this is an $O(n)$-time algorithm since at most $2n$ comparisons are made.

9. Give a recursive algorithm to compute the product of two positive integers m and n using only addition.

   Solution:

```
def product(m,n):
    '''
    INPUT: two positive integers n and m
    OUTPUT: the product m*n
    '''

    if n==1: return m  #if n=1, then n*m=m
    else:
        return product(m,n-1) + m


print(product(5,7))
```

10. Given an array, A, describe an efficient algorithm for reversing A. For example, if A = [3, 4, 1, 5], then its reversal is A = [5, 1, 4, 3]. You can only use $O(1)$ memory in addition to that used by A itself. What is the running time of your algorithm?

    Solution:

```
def reversing(A):
    '''
    INPUT: A n-element array A of integers
    OUTPUT: A in reversed order
    '''

    # we initialize the two pointers to the first and last element of the array
```

```python
    p1 = 0
    p2 = len(A)-1

    # There are about n/2 swaps to do
    for i in range(round(len(A)/2)):
        # we swap the element pointed by p1 and the elements pointed by p2
        temp = A[p1]
        A[p1] = A[p2]
        A[p2] = temp

        # we move the two pointers by one position up for p1, down for p2
        p1 += 1
        p2 -= 1
    return A


A = [4,9,5,7,3,4,5,3,9,1,1]
print(reversing(A))
```

11. Given an array, A, of n integers, find the longest subarray of A such that all the numbers in that subarray are in sorted order. What is the running time of your method?

Solution:

```python
def longest_sorted_subarray(A):
    '''
    INPUT: A n-element array A of integers
    OUTPUT: the longest sorted subarray of A
    '''

    # build the table tab_length that will hold the currently found
    # longest sorted subarray when iterating the array from lest to right
    tab_length = [1]
    counter = 1
    for i in range(1,len(A)):
        if A[i] >= A[i-1]: counter += 1  # we continue the subarray
        else: counter = 1       # we start a new subarray

        tab_length = tab_length + [counter]

    # once tab_length built, we just have to run through it from
    # right to left to check what is the longest sorted subarray of A
    longest = 1
    start = 0
    for i in reversed(range(1,len(A))):
        if tab_length[i] > longest:
            longest = tab_length[i]
            start = i - longest + 1

    return A[start:start+longest]

A = [4,9,5,7,3,4,5,3,9,1,1]
print(longest_sorted_subarray(A))
```

12. Given an array, A, of n positive integers, each of which appears in A exactly twice, except for one integer, x, describe an $O(n)$-time method for finding x using only a single variable besides A and the iterating variable.

Solution:
Initialize a variable *temp* to 0 and then XOR all the values in A with *temp*. The result will be x.

```python
def outlier(A):
    '''
    INPUT: A n-element array A of positive integers
    each of which appears in A exactly twice, except for one integer x
    OUTPUT: the outlier value x
    '''

    temp=0  # we initialise the temporary value to 0
    for value in A:
        temp ^= value  # we XOR each value of the array to temp

    return temp


A = [4,9,5,7,3,4,5,3,9,1,1]
print(outlier(A))
```

Hints

- **Question 10:** reverse the array by using index pointers that start at the two ends.
- **Question 12:** consider using the XOR function.