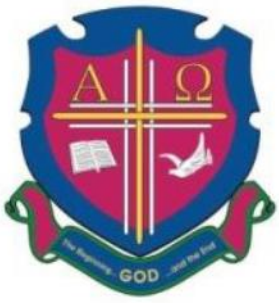

UGANDA CHRISTIAN UNIVERSITY

A Centre of Excellence in the Heart of Africa

FACULTY OF ENGINEERING DESIGN AND TECHNOLOGY

NAME: MOGA MUZAMIL ABDUL WAHAB

REG NO: S21B23/013

ACCESS NO: A94166

COURSE: BACHELOR OF SCIENCE IN COMPUTER SCIENCE (BSCS)

COURSE UNIT: DESIGN AND ANALYSIS OF ALGORITHMS

LECTURER: MR. WAMBETE JOSEPH NDAEBWA

1. An algorithm is a set of instructions for solving a problem or accomplishing a task.
   An example of an algorithm can be the process one does to get to class i.e waking up, praying, showering etc

2. A data structure refers to a group of data elements which provides an efficient way of storing and organizing data in the computer so that it can be used efficiently. An example of data structures can include a linked list

3. Algorithm complexity measures how many steps are required by the algorithm to solve a given problem

   a).
   Constant Complexity:
   It imposes a complexity of **O(1)**. It undergoes an execution of a constant number of steps like 1, 5, 10, etc. for solving a given problem.

   Logarithmic Complexity:
   It imposes a complexity of O(log(N)). It undergoes the execution of the order of log(N) steps. To perform operations on N elements, it often takes the logarithmic base as 2.

   Linear Complexity:
   It imposes a complexity of O(N). It encompasses the same number of steps as that of the total number of elements to implement an operation on N elements.

   Quadratic Complexity: It imposes a complexity of $O(n^2)$. For N input data size, it undergoes the order of N^2 count of operations on N number of elements for solving a given problem

Cubic Complexity: It imposes a complexity of $O(n^3)$. For N input data size, it executes the order of N^3 steps on N elements to solve a given problem.

Exponential Complexity: It imposes a complexity of $O(2^n)$, $O(N!)$, $O(n^k)$, For N elements, it will execute the order of count of operations that is exponentially dependable on the input data size.

4. Explain why analysis of algorithms is important?
   By analyzing different algorithms, we can compare them to determine the best one for use.
   To predict the behavior of an algorithm without implementing it on a specific computer.

Explain best case, worst case and average case complexity.

Best case: This is defined by the input for which algorithm takes less time or minimum time.

Worst Case: This is defined by the input for which algorithm takes a long time or maximum time

Average Case: This takes all random inputs and calculate the computation time for all inputs whereby;
Average case = all random case time / total no of case

5a) Pseudo code

1. Create a function called reverse
2. Create a variable called size and assign it to the length of the array
3. Create a for loop the iterates from 0 to half of the size
4. Swap the values of array at index I with array at index of size –i-1
5. Return the reversed array
6. Create a main function

7. Create a test array
8. Print the array
9. Call the main function

b) Deriving the complexity

Since we created a for loop to iterate from 0 to half, therefore our algorithm a worst case will have O(n).

```
#An algorithm that  reverses an array
 def reverse(array):
    size = len(array)
    for i in range(0,size//2):
        (array[i],array[size-i-1]) = (array[size-i-1],array[i])
    return array
def main():
    array = [2,6,9,10]
    print(reverse(array))
main()
```

6.

```
# using a linked list
 # Implementation of the Stack ADT using a singly linked list.
class Stack2 :
# Creates an empty stack.
    def __init__( self ):
        self._top = None
        self._size = 0
    # Returns True if the stack is empty or False otherwise.
    def isEmpty( self ):
        return self._top is None

    # Returns the number of items in the stack.
    def __len__( self ):
        return self._size

    # Returns the top item on the stack without removing it.
    def peek( self ):
        assert not self.isEmpty(), "Cannot peek at an empty stack"
        return self._top.item
```

```
    # Removes and returns the top item on the stack.
    def pop( self ):
        assert not self.isEmpty(), "Cannot pop from an empty stack"
        node = self._top
        self.top = self._top.next
        self._size -= 1
        return node.item

    # Pushes an item onto the top of the stack.
    def push( self, item ) :
        self._top = _StackNode( item, self._top )
        self._size += 1
#The private storage class for creating stack nodes.
class _StackNode :
    def __init__( self, item, link ) :
        self.item = item
        self.next = link

Class = Stack2()
Class.push(3)
print(Class.isEmpty())
print(Class.__len__())
```

a) Elements are pushed into the stack from element 1 to 5, but when we are printing the element at the peek of the stack, it is the last element that was pushed,
And on deleting (pop) it is the last element to be deleted . Therefore the above data Structure follows the concept of Last in, First Out.

7.Give some examples of Divide and Conquer algorithms

Merge sort

Binary search

Quick sort

Integer Multiplication

Matrix Multiplication

Maximal Subsequence.

8.

```
#Best case
#Best case scenario of a selection sort is when the elements in the list are
already in order or are sorted.
def Selection(list):
    n =len(list)
    for i in range(0,n-1):
        mini=i
        for j in range((i+1),n):
            if list[mini]>list[j]:

                mini=j
                if mini!=1:
                    list[mini],list[i] = list[i],list[mini]
def main():
    list=[3,4,5,6,7]
    print(list)
    Selection(list)
    print(list)
main()

#worst case
#The worst case is when the array is completely unsorted or sorted in descending
order.
def Selection(list):
    n =len(list)
    for i in range(0,n-1):
        mini=i
        for j in range((i+1),n):
            if list[mini]>list[j]:

                mini=j
                if mini!=1:
                    list[mini],list[i] = list[i],list[mini]
def main():
    list=[3,9,7,3,2,4,6,3,7,8]
    print(list)
    Selection(list)
    print(list)
main()
```

9. Arrange the following rates of growth in ascending order: N, n log n, n^2, 1, n, log n, n!, n^3

1,logn, n, n, nlogn, n^2, n^3, n!

10.

```
## sequential Search
# sequential Search has its worst case complexity as O(n)
def sequential_search(list,value):    # O(1)
    for i in range (len(list)):   # the for loop takes O(n)
        if value==list[i]:       # O(1)
            return i
    return -1

def main():           # O(1)
    list =[1,2,3,4,5,6,7,8,9]    # O(1)
    value = 9          # O(1)
    print(sequential_search(list,value))   # O(1)
    print((list,value))    # O(1)
main()      # O(1)


# the worst time complexity of the above sequential search is O(n)
# sequential search is at worst case complexity when either the last element was
the element being searched for or when the element being searched for is not in
the list
#in the above program we are searching for element 9 which is at index 8 and is
the last element thus worst case
```

11. illustrate the working of the quick sort on input instance:

25, 29, 30, 35, 42, 47, 50, 52, 60.

```
#Quick sort's average case time complexity is O(n*logn)
#Quick sort's best case time complexity is: O*log(n) when the partitions are as
evenly balanced as possible

#Worst Case Complexity:  n^2 is when the algorithm pick the largest or smallest
element as the pivot element every time.
#worst case is when the elements are sorted, reversed sorted or all elements are
same


def quickSort(data_list):
   quickSortHlp(data_list,0,len(data_list)-1)

def quickSortHlp(data_list,first,last):
   if first < last:

       splitpoint = partition(data_list,first,last)

       quickSortHlp(data_list,first,splitpoint-1)
       quickSortHlp(data_list,splitpoint+1,last)


def partition(data_list,first,last):
   pivotvalue = data_list[first]

   leftmark = first+1
   rightmark = last

   done = False
   while not done:

       while leftmark <= rightmark and data_list[leftmark] <= pivotvalue:
           leftmark = leftmark + 1

       while data_list[rightmark] >= pivotvalue and rightmark >= leftmark:
           rightmark = rightmark -1

       if rightmark < leftmark:
           done = True
       else:
```

```
            temp = data_list[leftmark]
            data_list[leftmark] = data_list[rightmark]
            data_list[rightmark] = temp

    temp = data_list[first]
    data_list[first] = data_list[rightmark]
    data_list[rightmark] = temp


    return rightmark

data_list = [25,29,30,35,42,47,50,52,60]
quickSort(data_list)
print(data_list)
```

12. Write a program/ algorithm of Merge Sort Method:

Sort this list [“Solomon”, “Daphine”, “Charles”, “Tracy”, “Jasper”, “Destiny”, “Nabil”] in alphabetic order. What is its Complexity.

COMPLEXITY:

```
#complexity of the Merge sort is: O(n) + O(n) + O(n) = O(3n)

# The complexity of this Merge Sort Method is O(n)
```

```
def merge(a,b):       # O(1)
    sorted_arr = []    # O(1)

    x = 0
    y = 0

    len_a = len(a)
    len_b = len(b)
    while x < len_a and y < len_b:      # the while loop takes O(n)
        if a[x] <= b[y]:                #the if statement takes O(1)
            sorted_arr.append(a[x])       # O(1)
```

```
            x=x+1                               # O(1)
        else:
            sorted_arr.append(b[y])             # O(1)

            y=y+1                                 # O(1)
    while x < len_a:                                  # While loop takes O(n)
since it is not nested
        sorted_arr.append(a[x])                       # O(1)
        x=x+1                                         # O(1)

    while y < len_b:                                  # while loop takes O(n) since it
is not nested
        sorted_arr.append(b[y])                       # O(1)
        y=y+1                                         # O(1)

    return sorted_arr                                 # O(1)

def merge_sort(arr):
    if len(arr) <= 1: #to check if the list has elements in it.   O(1)
        return arr       # O(1)

    mid = len(arr)//2  # finding the mid of the array

    arr_a = arr[:mid]        # O(1)
    arr_b = arr[mid:]        # O(1)

    arr_a = merge_sort(arr_a)       # O(1)
    arr_b = merge_sort(arr_b)     # O(1)


    return merge(arr_a,arr_b)       # O(1)

G = ["Solomon","Daphine","Charles","Tracy", "Jasper", "Destiny","Nabil"]

print("Array G before sorting.")
print(G)

print("Array G after sorting.")
print(merge_sort(G))


#complexity of the Merge sort is: O(n) + O(n) + O(n) = O(3n)

# The complexity of this Merge Sort Method is O(n)
```