International Institute of Professional Studies (IIPS) Devi Ahilya University
M.Tech. IT, 6th Semester, First Internal Test
Analysis and Design of Algorithms
13 February 2023
Dr. Shaligram Prajapat
https://sites.google.com/site/shaligramiipsdavvindore/

Q.1: Complete the table1.0 after simplification of corresponding recurrence relation

Table 1.0

Algorithm	Relation	Time Complexity	Space Complexity	Power Complexity
Linear Search	T(n)= T(n-1)+O(c)			
Binary Search	T(n)= T(n/2)+O(c)			
Factorial	T(n)= T(n-1)+O(c)			
Tower of Hanoi	T(n)=2 T(n-1)+O(c)			
Binary Tree Traversal	T(n)=2 T(n/2)+O(1)			
Merge Sort	T(n)=2 T(n/2)+O(n)			
Quick Sort	Best Case: T(n)=2 T(n/2)+O(n) Worst Case: T(n)= T(n-1)+O(n*n)			
Strassen's Matrix Multiplication	T(n)=7T(n/2)+O(n*n)			
Karatsuba Integer Multiplication	T(n)=3T(n/2)+O(n*n)			

Q.2 Write recursive algorithm for polynomial representation.

Q.3 Write iterative version of Tower of Hanoi Problem.

Solution1:
Time complexity may depend on various factors, such as the specific inputs to the algorithm, the efficiency of the implementation, and the use of any auxiliary data structures or algorithms. In practice, the actual running time of an algorithm may also be affected by various external factors, such as hardware limitations, operating system overhead, and the size of the input data.

- Linear Search: O(n)
- Binary Search: O(log n)
- Factorial: O(n)
- Tower of Hanoi: O(2^n)
- Binary Tree Traversal: O(n)
- Merge Sort: O(n log n)
- Quick Sort (best case): O(n log n)
- Quick Sort (worst case): O(n^2)
- Strassen's Matrix Multiplication: O(n^log2(7))
- Karatsuba Integer Multiplication: O(n^log2(3))

Space complexity may also depend on other factors, such as the implementation details and the specific inputs to the algorithm. In some cases, the space complexity may also be affected by the use of auxiliary data structures or temporary variables.

- Linear Search: O(1)
- Binary Search: O(1)
- Factorial: O(1)
- Tower of Hanoi: O(n)
- Binary Tree Traversal: O(h), where h is the height of the tree.
- Merge Sort: O(n)
- Quick Sort (best case): O(log n)
- Quick Sort (worst case): O(n)
- Strassen's Matrix Multiplication: O(n^2)
- Karatsuba Integer Multiplication: O(n^log2(3))

Power Complexity: The power complexity of an algorithm is a measure of how the algorithm's running time grows as the input size increases. It is typically expressed in terms of the exponent in the running time's big O notation.Note that the power complexities of the algorithms are not the same as their time or space complexities. The time and space complexities may have different constants or lower-order terms that affect the actual running time or memory usage of the algorithm.

- Linear Search: O(n)
- Binary Search: O(log n)
- Factorial: O(n)
- Tower of Hanoi: O(2^n)
- Binary Tree Traversal: O(n)
- Merge Sort: O(n log n)
- Quick Sort (best case): O(n log n)
- Quick Sort (worst case): O(n^2)
- Strassen's Matrix Multiplication: O(n^log2(7))
- Karatsuba Integer Multiplication: O(n^log2(3))

1.1. Linear Search
The recurrence relation for the linear search algorithm is:

$T(n) = T(n-1) + O(c)$
where n is the size of the input, and c is the constant time taken to compare an element in the list with the target element.

To solve this recurrence relation, we can use the recursive tree method. The recurrence relation can be written as:

$$\begin{aligned}T(n) &= T(n-1) + c\\ &= T(n-2) + c + c\\ &= T(n-3) + c + c + c\\ &= \ldots\\ &= T(1) + c + c + \ldots + c \quad \text{(n-1 times)}\\ &= O(n)\end{aligned}$$

Therefore, the time complexity of the linear search algorithm is O(n).
This is because in the worst case, the algorithm needs to compare every element in the list with the target element.

The space complexity of the linear search algorithm is O(1),
because it does not require any additional space beyond the input list and a few constant variables used for indexing and comparison.
The power complexity of linear search is O(n), which means that the running time of the algorithm grows linearly with the size of the input data. In other words, if the input size is doubled, the running time of the algorithm will also double. The power complexity is expressed as the exponent in the big O notation, which represents an upper bound on the growth rate of the running time as a function of the input size. The power complexity of linear search is relatively simple and easy to understand, but it may not be the most efficient algorithm for large datasets or complex search conditions.

1.2 Binary Search T(n)= T(n/2)+O(c) Find time and space complexity
The recurrence relation for the binary search algorithm is:
$T(n) = T(n/2) + O(c)$
where n is the size of the input, and c is the constant time taken to compare an element in the list with the target element.

To solve this recurrence relation, we can use the master theorem. The master theorem states that if a recurrence relation has the form:
$T(n) = aT(n/b) + f(n)$
where a >= 1 and b > 1 are constants, and f(n) is an asymptotically positive function, then:

If f(n) = O(n^log_b(a - ε)) for some constant ε > 0, then T(n) = Θ(n^log_b(a)).
If f(n) = Θ(n^log_b(a)), then T(n) = Θ(n^log_b(a) log n).
If f(n) = Ω(n^log_b(a + ε)) for some constant ε > 0, and if a f(n/b) <= c f(n) for some constant c < 1 and all sufficiently large n, then T(n) = Θ(f(n)).
In the case of the binary search algorithm, we have a = 1, b = 2, and f(n) = O(c). Therefore, we can apply case 1 of the master theorem:

T(n) = Θ(n^log_2(1)) = Θ(1)
Therefore, the time complexity of the binary search algorithm is O(1). This is because the algorithm divides the input size in half at each step, so the number of steps required is log n, which is a constant factor.

The space complexity of the binary search algorithm is O(1), because it does not require any additional space beyond the input list and a few constant variables used for indexing and comparison.
Power complexity of Binary search

The power complexity of binary search is O(log n), which means that the running time of the algorithm grows logarithmically with the size of the input data. In other words, if the input size is doubled, the running time of the algorithm will increase by a constant factor (i.e., one additional comparison on average). The power complexity of binary search is much better than that of linear search for large datasets, as it enables the algorithm to quickly narrow down the search range by discarding half of the remaining items at each step. This makes binary search a highly efficient algorithm for searching sorted arrays and other ordered data structures.

1.3 The recurrence relation for the factorial function is:

T(n) = T(n-1) + O(c)
where n is the input to the factorial function and c is the constant time taken to perform a multiplication operation.

To solve this recurrence relation, we can use the recursive tree method. The recurrence relation can be written as:

```
T(n) = T(n-1) + c
    = T(n-2) + c + c
    = T(n-3) + c + c + c
    = ...
    = T(1) + c + c + ... + c    (n-1 times)
    = O(n)
```

Therefore, the time complexity of the factorial function is O(n). This is because the function calls itself recursively n times, and performs a multiplication operation in each call.

The space complexity of the factorial function is O(n), because the function calls itself recursively n times, and each call creates a new stack frame on the call stack. The call stack therefore has a maximum depth of n, which is the space required by the function.

The power complexity of the recursive factorial function is O(n), which means that the running time of the algorithm grows linearly with the size of the input data. In this case, the input data is the integer n that represents the number whose factorial is to be computed. The recursive factorial function works by repeatedly multiplying n by the factorial of (n-1), until it reaches the base case of n=1, which has a factorial of 1. The total number of recursive calls made by the algorithm is proportional to n, which gives it a power complexity of O(n). Note that there are more efficient algorithms for computing factorials, such as iterative algorithms that use a simple loop to perform the multiplication, which have a power complexity of O(1). However, the recursive factorial function is a simple and intuitive example of a recursive algorithm, and it can be used to illustrate the basic principles of recursion and function calls.

1.4 The recurrence relation for the Tower of Hanoi problem is:

T(n) = 2T(n-1) + O(c)
where n is the number of disks to be moved, and c is the constant time taken to perform a disk move.

To solve this recurrence relation, we can use the recursive tree method. The recurrence relation can be written as:

```
T(n) = 2T(n-1) + c
    = 2(2T(n-2) + c) + c
    = 2^2 T(n-2) + 2c + c
    = 2^3 T(n-3) + 2^2 c + 2c + c
    = ...
    = 2^n T(0) + c(2^(n-1) + 2^(n-2) + ... + 1)
    = 2^n + c(2^n - 1)
    = O(2^n)
```

Therefore, the time complexity of the Tower of Hanoi problem is O(2^n). This is because in the worst case, the algorithm needs to make 2^n - 1 disk moves to complete the problem.

The space complexity of the Tower of Hanoi problem is O(n), because the function calls itself recursively n times, and each call creates a new stack frame on the call stack. The call stack therefore has a maximum depth of n, which is the space required by the function.

The power complexity of the Tower of Hanoi algorithm is O(2^n), which means that the running time of the algorithm grows exponentially with the size of the input data. In this case, the input data is the number of disks that need to be moved from the source peg to the destination peg, and the algorithm works by recursively moving smaller disks from the source peg to a temporary peg, and then moving larger disks from the source peg to the destination peg, and finally moving the smaller disks from the temporary peg to the destination peg. The total number of moves required to solve the problem for n disks is 2^n - 1, which gives the algorithm a power complexity of O(2^n). Note that the Tower of Hanoi algorithm is a classic example of a recursive algorithm with an exponential time complexity, and it is often used to teach the principles of recursion and algorithm analysis. However, for large values of n, the algorithm can become computationally expensive and impractical to use.

1.5 The recurrence relation for binary tree traversal (in-order, pre-order or post-order) is:

T(n) = 2T(n/2) + O(1)
where n is the number of nodes in the binary tree, and O(1) is the constant time taken to process a node.

To solve this recurrence relation, we can use the master theorem. The master theorem states that if a recurrence relation has the form:

T(n) = aT(n/b) + f(n)
where a >= 1 and b > 1 are constants, and f(n) is an asymptotically positive function, then:

If f(n) = O(n^log_b(a - ε)) for some constant ε > 0, then T(n) = Θ(n^log_b(a)).
If f(n) = Θ(n^log_b(a)), then T(n) = Θ(n^log_b(a) log n).
If f(n) = Ω(n^log_b(a + ε)) for some constant ε > 0, and if a f(n/b) <= c f(n) for some constant c < 1 and all sufficiently large n, then T(n) = Θ(f(n)).
In the case of binary tree traversal, we have a = 2, b = 2, and f(n) = O(1). Therefore, we can apply case 2 of the master theorem:

T(n) = Θ(n^log_2(2)) * log n = Θ(n log n)
Therefore, the time complexity of binary tree traversal is O(n log n).

The space complexity of binary tree traversal depends on the implementation of the traversal algorithm. If **the algorithm uses an explicit stack to maintain the traversal order, then the space complexity is O(h),** where h is the height of the binary tree. This is because the maximum number of nodes that can be stored on the stack at any given time is h, which is the height of the binary tree.

If the algorithm uses the call stack for recursion, then the space complexity is O(log n), where n is the number of nodes in the binary tree. This is because the maximum depth of the call stack is log n, which is the height of a balanced binary tree.

The power complexity of binary tree traversal is O(n^1) = O(n), because the algorithm needs to process each node in the binary tree once.
1.6 The recurrence relation for merge sort is:
T(n) = 2T(n/2) + O(n)
where n is the number of elements to be sorted, and O(n) is the time taken to merge two sorted arrays of size n/2.

To solve this recurrence relation, we can use the master theorem. The master theorem states that if a recurrence relation has the form:
$T(n) = aT(n/b) + f(n)$
where $a >= 1$ and $b > 1$ are constants, and f(n) is an asymptotically positive function, then:

If $f(n) = O(n^\wedge log\_b(a - \varepsilon))$ for some constant $\varepsilon > 0$, then $T(n) = \Theta(n^\wedge log\_b(a))$.
If $f(n) = \Theta(n^\wedge log\_b(a))$, then $T(n) = \Theta(n^\wedge log\_b(a) \log n)$.
If $f(n) = \Omega(n^\wedge log\_b(a + \varepsilon))$ for some constant $\varepsilon > 0$, and if $a f(n/b) <= c f(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$.
In the case of merge sort, we have a = 2, b = 2, and f(n) = O(n). Therefore, we can apply case 2 of the master theorem:
$T(n) = \Theta(n^\wedge log\_2(2)) * \log n = \Theta(n \log n)$
Therefore, the time complexity of merge sort is O(n log n).

The space complexity of merge sort is O(n), because the algorithm needs to create temporary arrays to store the elements being merged during the merge phase of the algorithm. The size of each temporary array is n, which gives a total space complexity of O(n).

The power complexity of merge sort is O(n^1 log n) = O(n log n), because the algorithm needs to perform n log n comparisons and swaps in the worst case.
1.7 Quick Sort
Best Case: T(n)=2 T(n/2)+O(n) Time space and power complexity
The recurrence relation for Quick Sort in its best case is:

$T(n) = 2T(n/2) + O(n)$
where n is the number of elements to be sorted, and O(n) is the time taken to partition the array into two parts.

To solve this recurrence relation, we can use the master theorem. The master theorem states that if a recurrence relation has the form:

$T(n) = aT(n/b) + f(n)$
where $a >= 1$ and $b > 1$ are constants, and f(n) is an asymptotically positive function, then:

If $f(n) = O(n^\wedge log\_b(a - \varepsilon))$ for some constant $\varepsilon > 0$, then $T(n) = \Theta(n^\wedge log\_b(a))$.
If $f(n) = \Theta(n^\wedge log\_b(a))$, then $T(n) = \Theta(n^\wedge log\_b(a) \log n)$.
If $f(n) = \Omega(n^\wedge log\_b(a + \varepsilon))$ for some constant $\varepsilon > 0$, and if $a f(n/b) <= c f(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$.
In the case of Quick Sort in its best case, we have a = 2, b = 2, and f(n) = O(n). Therefore, we can apply case 2 of the master theorem:

$T(n) = \Theta(n^\wedge log\_2(2)) * \log n = \Theta(n \log n)$
Therefore, the time complexity of Quick Sort in its best case is O(n log n).

The space complexity of Quick Sort depends on the implementation. In the simplest implementation, the space complexity is O(log n) due to the recursive calls, but in the worst case it can be O(n) due to the stack space used by the recursion. However, in the best case, the space complexity is O(log n).

The power complexity of Quick Sort in its best case is O(n^1 log n) = O(n log n), because the algorithm needs to perform n log n comparisons and swaps in the best case.

The recurrence relation for Quick Sort in its worst case is:

T(n) = T(n-1) + O(n^2)
where n is the number of elements to be sorted, and O(n^2) is the time taken to partition the array into two parts.

To solve this recurrence relation, we can use the iterative method. In the worst case, the partition operation always selects the largest or smallest element as the pivot, and the array is not divided into two equal parts. In this case, the depth of the recursion tree is n, and at each level i, the partition operation takes O(n-i) time. Therefore, the total time taken by the algorithm is:
T(n) = O(n) + O(n-1) + O(n-2) + ... + O(1)
    = O(n^2)
Therefore, the time complexity of Quick Sort in its worst case is O(n^2).

The space complexity of Quick Sort in its worst case is O(n) due to the recursive calls, since the depth of the recursion tree is n.

The power complexity of Quick Sort in its worst case is O(n^3), since the algorithm needs to perform n^2 comparisons and swaps, and the recursion tree has depth n.

1.8 Strassen's Matrix Multiplication T(n)=7T(n/2)+O(n*n) Time space and power complexity

Strassen's Matrix Multiplication is a matrix multiplication algorithm that uses a divide and conquer approach. The recurrence relation for the algorithm is given by:

T(n) = 7T(n/2) + O(n^2)
where n is the size of the matrices to be multiplied, and O(n^2) is the time taken to add or subtract the matrices.

To solve this recurrence relation, we can use the master theorem. The master theorem states that if a recurrence relation has the form:

T(n) = aT(n/b) + f(n)
where a >= 1 and b > 1 are constants, and f(n) is an asymptotically positive function, then:

If f(n) = O(n^log_b(a - ε)) for some constant ε > 0, then T(n) = Θ(n^log_b(a)).
If f(n) = Θ(n^log_b(a)), then T(n) = Θ(n^log_b(a) log n).
If f(n) = Ω(n^log_b(a + ε)) for some constant ε > 0, and if a f(n/b) <= c f(n) for some constant c < 1 and all sufficiently large n, then T(n) = Θ(f(n)).
In the case of Strassen's Matrix Multiplication, we have a = 7, b = 2, and f(n) = O(n^2). Therefore, we can apply case 3 of the master theorem:

f(n) = O(n^log_b(a + ε)) = O(n^log_2(7 + ε))
where we can choose ε such that log_2(7 + ε) > 2. Therefore, we have:

a f(n/b) = 7 O((n/2)^2) = (7/4) O(n^2)
and:
a f(n/b) <= c f(n)
for some constant c < 1 and all sufficiently large n. Therefore, by case 3 of the master theorem, we have:

T(n) = Θ(n^log_2(7))
Therefore, the time complexity of Strassen's Matrix Multiplication is O(n^log_2(7)), which is approximately **O(n^2.81).**

The space complexity of Strassen's Matrix Multiplication is O(n^log_2(7)), which is the same as the time complexity, since the algorithm uses a recursive approach that requires the creation of new matrices.

The power complexity of Strassen's Matrix Multiplication is O(n^log_2(7) log n), since the algorithm performs O(n^log_2(7)) scalar multiplications at each level of the recursion tree, and the depth of the recursion tree is log n.

1.9

Karatsuba Integer Multiplication T(n)=3T(n/2)+O(n*n) Time space and power complexity

Karatsuba Integer Multiplication is a divide and conquer algorithm for multiplying two integers. The recurrence relation for the algorithm is given by:

T(n) = 3T(n/2) + O(n^2)
where n is the number of digits in the integers being multiplied, and O(n^2) is the time taken to add or subtract the integers.

To solve this recurrence relation, we can use the master theorem. The master theorem states that if a recurrence relation has the form:

T(n) = aT(n/b) + f(n)
where a >= 1 and b > 1 are constants, and f(n) is an asymptotically positive function, then:

If f(n) = O(n^log_b(a - ε)) for some constant ε > 0, then T(n) = Θ(n^log_b(a)).
If f(n) = Θ(n^log_b(a)), then T(n) = Θ(n^log_b(a) log n).
If f(n) = Ω(n^log_b(a + ε)) for some constant ε > 0, and if a f(n/b) <= c f(n) for some constant c < 1 and all sufficiently large n, then T(n) = Θ(f(n)).
In the case of Karatsuba Integer Multiplication, we have a = 3, b = 2, and f(n) = O(n^2). Therefore, we can apply case 1 of the master theorem:

f(n) = O(n^log_b(a - ε)) = O(n^log_2(3 - ε))
where we can choose ε such that log_2(3 - ε) < 2. Therefore, we have:

T(n) = Θ(n^log_2(3))
Therefore, the **time complexity of Karatsuba Integer Multiplication is O(n^log_2(3)), which is approximately O(n^1.585).**

The space complexity of Karatsuba Integer Multiplication is O(n^log_2(3)), which is the same as the time complexity, since the algorithm uses a recursive approach that requires the creation of new integers.

The power complexity of Karatsuba Integer Multiplication is O(n^log_2(3) log n), since the algorithm performs O(n^log_2(3)) multiplications at each level of the recursion tree, and the depth of the recursion tree is log n.

Algorithm	Relation	Time Complexity	Space Complexity	Power Complexity
Linear Search	T(n)= T(n-1)+O(c)			
Binary Search	T(n)= T(n/2)+O(c)			
Factorial	T(n)= T(n-1)+O(c)			

Tower of Hanoi	T(n)=2 T(n-1)+O(c)			
Binary Tree Traversal	T(n)=2 T(n/2)+O(1)			
Merge Sort	T(n)=2 T(n/2)+O(n)			
Quick Sort	Best Case: T(n)=2 T(n/2)+O(n) Worst Case: T(n)= T(n-1)+O(n*n)			
Strassen's Matrix Multiplication	T(n)=7T(n/2)+O(n*n)			
Karatsuba Integer Multiplication	T(n)=3T(n/2)+O(n*n)			

Q.2 Write recursive algorithm for polynomial representation.
This algorithm defines a class Polynomial with a constructor that takes a list of coefficients. The evaluate method evaluates the polynomial for a given value of x.

The base case of the recursion is when the polynomial has degree 0, i.e., it is a constant polynomial. In this case, the method simply returns the single coefficient.

In the recursive case, the method creates a new polynomial p of degree n-1 by removing the highest degree coefficient from the list of coefficients. It then recursively evaluates p for the given value of x, and multiplies the result by x. Finally, it adds the highest degree coefficient to obtain the value of the polynomial for the given value of x.

The algorithm works by breaking down the problem of evaluating a polynomial of degree n into the problem of evaluating a polynomial of degree n-1. It uses the fact that a polynomial of degree n can be expressed as p(x) = x * q(x) + c, where q(x) is a polynomial of degree n-1 and c is the highest degree coefficient. This allows the evaluation to be performed recursively by computing q(x) and multiplying by x, and then adding the highest degree coefficient.

```
class Polynomial:
    def __init__(self, coefficients):
        self.coefficients = coefficients

    def evaluate(self, x):
        """
        Evaluates the polynomial for a given value of x.
        """
        n = len(self.coefficients)

        # Base case: constant polynomial
        if n == 1:
            return self.coefficients[0]

        # Recursive case: evaluate recursively for smaller degree
        else:
            # Compute the value of the polynomial of degree n-1
            p = Polynomial(self.coefficients[:-1])
            result = p.evaluate(x)
            # Add the term for the highest degree
            result = result*x + self.coefficients[-1]

            return result
```

Q.3 Write iterative version of Tower of Hanoi Problem.
This implementation uses a stack to keep track of the recursive calls that would be made in the recursive version. The stack starts with the initial call to move n disks from the from_rod to the to_rod, using the aux_rod as an intermediary.

In each iteration of the loop, the top call is popped from the stack and executed. If there is only one disk to move, it is moved and the loop continues. Otherwise, two recursive calls are pushed onto the stack: one to move n-1 disks from from_rod to aux_rod (using to_rod as intermediary), and another to move n-1 disks from aux_rod to to_rod (using from_rod as intermediary). Finally, the call to move the remaining disk from from_rod to to_rod is also pushed onto the stack.

This process continues until the stack is empty, at which point all the necessary moves have been made to solve the Tower of Hanoi problem.

```
def tower_of_hanoi(n: int, from_rod: str, to_rod: str, aux_rod: str) -> None:
    # Create a stack and push the initial values
    stack = [(n, from_rod, to_rod, aux_rod)]

    # Loop until the stack is empty
    while stack:
        n, from_rod, to_rod, aux_rod = stack.pop()

        # If there is only one disk to move, move it and continue
        if n == 1:
            print(f"Move disk 1 from {from_rod} to {to_rod}")
            continue

        # Otherwise, push the two recursive calls onto the stack
        stack.append((n - 1, aux_rod, to_rod, from_rod))
        stack.append((1, from_rod, to_rod, aux_rod))
        stack.append((n - 1, from_rod, aux_rod, to_rod))
```