Index

Sr. No	Practical No.	Practical Name	Page no.	Date	Sign
1	1 & 2	Analysis of Sorting Algorithms	3-10	12/01/23	
2	3	Singly Linked Lists Operations Analysis	11-19	19/01/23	
3	4	City Database Functions using Array and Linked Lists Operations	21-31	03/02/23	
4	5	Activity Selection Algorithm using Greedy Approach	33-34	10/02/23	
5	6	Matrix multiplication and Strassen's Matrix Multiplication Comparision	35-38	14/03/23	
6	7	Floyd Warshall Algorithm for all pair shortest path problem	39-42	21/03/23	
7	8	n-queens using backtracking	43-45	28/03/23	
8	9	Travelling Salesman problem using branch and bound	47-49	04/04/23	
9	10	Least Commonn Subsequence : flip Classroom	51-52	11/04/23	
10	11	Design and solve the given problem using Algorithmic Approach : Question 1	53-55	11/04/23	

Name: Jaivik Jariwala

Roll Number: 21BCP004

Branch: Computer Science

Batch: 21

Division: 1

Group: 1

DESIGN ALGORITHM AND ANALYSIS LABORATORY – 1 & 2

- **Jaivik Jariwala**

21BCP004

Div-1 Group-1

Practical No. 1 and 2

Implement the following sorting in any programming language.

a) Insertion sort

b) Selection sort

c) Merge sort

d) Quick sort

Now, measure the execution time and the number of steps required to execute each algorithm in best case, worst case, and average case.

Aim :

To Compare Sorting methods

Selection Sort vs Merge Sort vs Insertion Sort vs Quick Sort

Introduction :

Different sorting algorithms have different time and space complexity, and their performance varies based on the input size and type. Here are some of the most common sorting algorithms and their properties:

Insertion Sort: Faster than Bubble Sort for small arrays, with O(n^2) time complexity in worst case.

Selection Sort: Also slow, with O(n^2) time complexity in worst case.

Merge Sort: Divide and conquer approach with O(n log n) time complexity in all cases.

Quick Sort: Divide and conquer approach, efficient for large data sets with O(n log n) average time complexity, but O(n^2) in worst case.

In general, Quick Sort and Merge Sort are preferred for large data sets due to their average time complexity, while Insertion and Heap Sort are preferred for small data sets and for sorting linked lists.

Algorithm :

```
Selection Sort :
        SMALLEST (arr, i, n, pos)
        Step 1: [INITIALIZE] SET SMALL = arr[i]
        Step 2: [INITIALIZE] SET pos = i
        Step 3: Repeat for j = i+1 to n
        if (SMALL > arr[j])
            SET SMALL = arr[j]
        SET pos = j
        [END OF if]
        [END OF LOOP]
        Step 4: RETURN pos

    Insertion Sort :
        insertionSort(array)
        mark first element as sorted
        for each unsorted element X
            'extract' the element X
            for j <- lastSortedIndex down to 0
            if current element j > X
```

```
                move sorted element to the right by 1
            break loop and insert X here
        end insertionSort
```

Quick Sort :

```
        function partitionFunc(left, right, pivot)
        leftPointer = left
        rightPointer = right - 1
        while True do
            while A[++leftPointer] < pivot do
                //do-nothing
            end while
            while rightPointer > 0 && A[--rightPointer] > pivot
do
                //do-nothing
            end while
            if leftPointer >= rightPointer
                break
            else
                swap leftPointer,rightPointer
            end if
        end while
        swap leftPointer,right
        return leftPointer
        end function
```

merge sort :

```
MERGE_SORT(arr, beg, end)
if beg < end
set mid = (beg + end)/2
MERGE_SORT(arr, beg, mid)
MERGE_SORT(arr, mid + 1, end)
MERGE (arr, beg, mid, end)
end of if
END MERGE_SORT
```

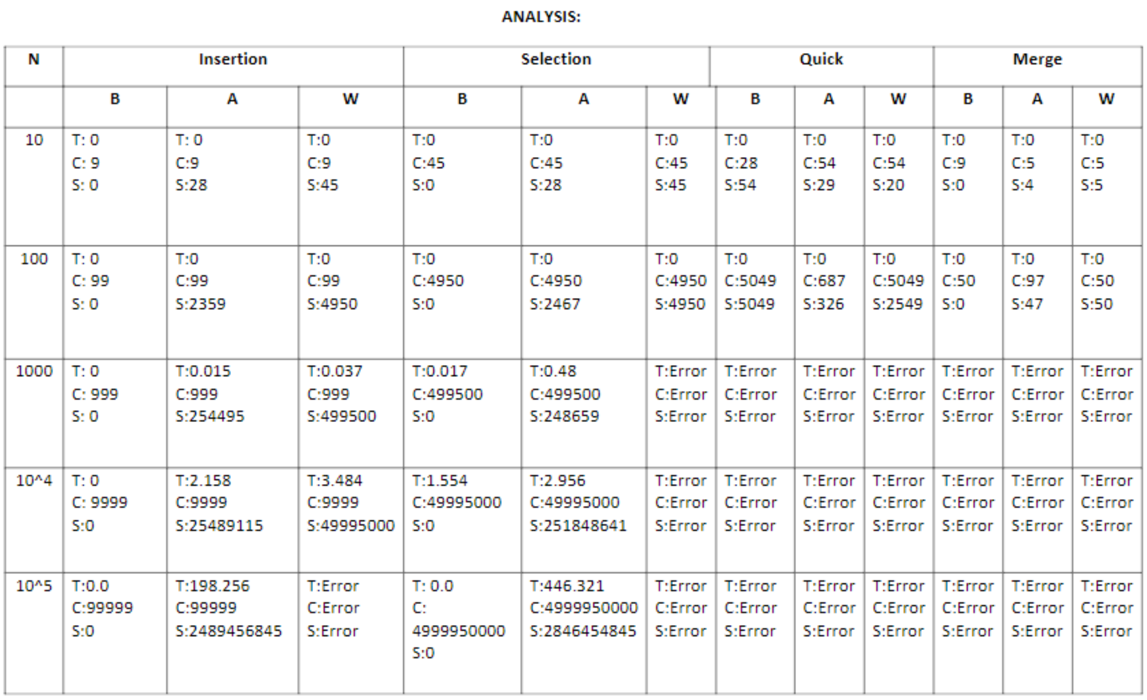
CODE :

```python
import numpy
arr=numpy.random.randint(10000,size=(100))
import time
import copy

def selection(arr):
    start_time = time.time()
    count=0
    for i in range(len(arr)):
        min = i
        for j in range(i+1, len(arr)):
            count += 1
            if arr[min] > arr[j]:
                min = j
        arr[i], arr[min] = arr[min], arr[i]
    return count

def insertion_sort(arr):
    start_time = time.time()
    count = 0
    for i in range(0,len(arr)):
        val=arr[i]
        j=i-1
        while j>=0 and val<arr[j]:
            arr[j+1] = arr[j]
            j-=1
            count+=1
        arr[j+1]=val
    return(arr, count)

def quick_sort(arr):
    start_time = time.time()
    if len(arr) <= 1:
        return arr
    else:
        pivot = arr[0]
        less = [x for x in arr[1:] if x <= pivot]
        greater = [x for x in arr[1:] if x > pivot]
        return quick_sort(less) + [pivot] + quick_sort(greater)

def merge_sort(arr):
    start_time= time.time()
    global count
    if len(arr) <= 1:
        return arr
    middle = len(arr) // 2
    left = arr[:middle]
    right = arr[middle:]
    left = merge_sort(left)
    right = merge_sort(right)
    count += 1
    return merge(left, right)

def merge(left, right):
    result = []
    i = 0
    j = 0
    while i < len(left) and j < len(right):
        if left[i] <= right[j]:
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1
    result += left[i:]
    result += right[j:]
    return result

def printArr(a):
    for i in range(len(a)):
        print (a[i], end = " ")
printArr(arr)

def sorting(arr):
    arr =numpy.sort(arr)
printArr(arr)

def reversing(arr)
    arr=numpy.flip(arr)
printArr(arr)
```

```
def check(arr):
    print ("Original list : " + str(arr))
    flag = 0
    arr1 = arr[:]
    arr1.sort()
    if (arr1 == arr):
        flag = 1
    if (flag) :
        print ("Yes, List is sorted.")
    else :
        print ("No, List is not sorted.")


For Average Case :

selection_arr1 = copy.copy(arr)
selection_arr1 = arr[:]
selection(selection_arr1)
printArr(selection_arr1)
check(selection_arr1)
insertion_arr1 = copy.copy(arr)
insertion_arr1 = arr[:]
insertion_sort(insertion_arr1)
printArr(insertion_arr1)
check(insertion_arr1)
quick_arr1 = copy.copy(arr)
quick_arr1 = arr[:]
quick_sort(quick_arr1)
printArr(quick_arr1)
end_time4 =time.time()
check(quick_arr1)
merge_arr1 = copy.copy(arr)
merge_arr1 = arr[:]
merge_sort(merge_arr1)
printArr(merge_arr1)
check(merge_arr1)

For Best Case :

sorting(arr)
selection_arr2 = copy.copy(arr)
selection_arr2 = arr[:]
selection(selection_arr2)
printArr(selection_arr2)
check(selection_arr2)
insertion_arr2 = copy.copy(arr)
insertion_arr2 = arr[:]
insertion_sort(insertion_arr2)
```

```
printArr(insertion_arr2)
check(insertion_arr2)
quick_arr2 = copy.copy(arr)
quick_arr2 = arr[:]
quick_sort(quick_arr2)
printArr(quick_arr2)
check(quick_arr1)
merge_arr2 = copy.copy(arr)
merge_arr2 = arr[:]
merge_sort(merge_arr2)
printArr(merge_arr2)
check(merge_arr1)

For Worst Case :

reversing(arr)
selection_arr3 = copy.copy(arr)
selection_arr3 = arr[:]
selection(selection_arr3)
printArr(selection_arr3)
check(selection_arr3)
print("\nTime taken: ", time.time() - start_time)
insertion_arr3 = copy.copy(arr)
insertion_arr3 = arr[:]
insertion_sort(insertion_arr3)
printArr(insertion_arr3)
check(insertion_arr3)
print("\nTime taken: ", time.time() - start_time)
quick_arr3 = copy.copy(arr)
quick_arr3 = arr[:]
quick_sort(quick_arr3)
printArr(quick_arr3)
check(quick_arr3)
print("\nTime taken: ", time.time() - start_time)
merge_arr3 = copy.copy(arr)
merge_arr3 = arr[:]
merge_sort(merge_arr3)
printArr(merge_arr3)
check(quick_arr3)
print("\nTime taken: ", time.time() - start_time)
```

Result :



ANALYSIS:

N	Insertion			Selection			Quick			Merge		
	B	A	W	B	A	W	B	A	W	B	A	W
10	T: 0 C: 9 S: 0	T: 0 C:9 S:28	T:0 C:9 S:45	T:0 C:45 S:0	T:0 C:45 S:28	T:0 C:45 S:45	T:0 C:28 S:54	T:0 C:54 S:29	T:0 C:54 S:20	T:0 C:9 S:0	T:0 C:5 S:4	T:0 C:5 S:5
100	T: 0 C: 99 S: 0	T:0 C:99 S:2359	T:0 C:99 S:4950	T:0 C:4950 S:0	T:0 C:4950 S:2467	T:0 C:4950 S:4950	T:0 C:5049 S:5049	T:0 C:687 S:326	T:0 C:5049 S:2549	T:0 C:50 S:0	T:0 C:97 S:47	T:0 C:50 S:50
1000	T: 0 C: 999 S: 0	T:0.015 C:999 S:254495	T:0.037 C:999 S:499500	T:0.017 C:499500 S:0	T:0.48 C:499500 S:248659	T:Error C:Error S:Error	T:Error C:Error S:Error	T:Error C:Error S:Error	T:Error C:Error S:Error	T:Error C:Error S:Error	T:Error C:Error S:Error	T:Error C:Error S:Error
10^4	T: 0 C: 9999 S:0	T:2.158 C:9999 S:25489115	T:3.484 C:9999 S:49995000	T:1.554 C:49995000 S:0	T:2.956 C:49995000 S:251848641	T:Error C:Error S:Error	T:Error C:Error S:Error	T:Error C:Error S:Error	T:Error C:Error S:Error	T:Error C:Error S:Error	T:Error C:Error S:Error	T:Error C:Error S:Error
10^5	T:0.0 C:99999 S:0	T:198.256 C:99999 S:2489456845	T:Error C:Error S:Error	T: 0.0 C: 4999950000 S:0	T:446.321 C:4999950000 S:2846454845	T:Error C:Error S:Error	T:Error C:Error S:Error	T:Error C:Error S:Error	T:Error C:Error S:Error	T:Error C:Error S:Error	T:Error C:Error S:Error	T:Error C:Error S:Error

Application :

Error Comparing sorting none available

References :

Sorting Algorithms - A Comparative Study

International-Journal-of-Computer-Science-and-Information-Security-1947-5500

-Naeem Akhter

-Muhammad Idrees

-Furqan Rehman

Attach later

DESIGN ALGORITHM AND ANALYSIS LABORATORY – 3

- **Jaivik Jariwala**

21BCP004

Div-1 Group-1

Practical No. 3

Use singly linked lists to implement integers of unlimited size. Each node of the list should store one digit of the integer. You should implement addition, subtraction, multiplication, and exponentiation operations. Limit exponents to be positive integers.

What is the asymptotic running time for each of your operations, expressed in terms of the number of digits for the two operands of each function?

Aim :

To Perform Recursive Implementation of operations using linked lists

Introduction :

The recursive implementation for operations on linked lists of integers of unlimited size is based on
the divide-and-conquer approach. In this approach, we break the problem down into smaller sub-problems, solve them recursively, and combine the solutions to obtain the final result.

• For addition and subtraction,
we traverse both input-linked lists simultaneously, adding or subtracting the corresponding digits and
propagating any carry or borrow. We then recursively call the function with the remaining digits until
both lists are empty.

• For multiplication,
we use the Karatsuba algorithm, which is a divide-and-conquer algorithm for multiplying two numbers. It reduces the number of multiplication operations needed by breaking down the multiplication into three smaller multiplications using some clever algebraic manipulations. We apply
this algorithm recursively until we reach the base case of multiplying two single digits.

• For exponentiation,
we first convert the exponent-linked list to an integer and then use a recursive function that *divides
the exponent by 2 at each step*, and *multiplies the result of the recursive call by itself*. This approach reduces the number of multiplications needed to compute the exponent, resulting in a better time complexity than the iterative approach.

Code Implementation:

- *Basic Implementation of Linked Lists and runtime calculation dependencies*

```python
import time

class Node:
    def __init__(self, value=None):
        self.value = value
        self.next = None

class LinkedList:
    def __init__(self, values=None):
        self.head = None
        self.tail = None
        self.length = 0

        if values is not None:
            for value in values:
                self.append(value)

    def append(self, value):
        if self.head is None:
            self.head = Node(value)
        else:
            current = self.head
            while current.next is not None:
                current = current.next
            current.next = Node(value)

    def to_list(self):
        current = self.head
        values = []
        while current is not None:
            values.append(current.value)
            current = current.next
        return values
```

- *Addition*

```python
def add_lists(list1, list2):
    start_time = time.time()

    def _add(p1, p2, carry):
        if p1 is None and p2 is None:
            if carry:
                return LinkedList([carry])
            else:
                return LinkedList()

        if p1 is None:
            p1_value = 0
        else:
            p1_value = p1.value
            p1 = p1.next

        if p2 is None:
            p2_value = 0
        else:
            p2_value = p2.value
            p2 = p2.next

        carry, value = divmod(p1_value + p2_value + carry, 10)
        result = LinkedList([value])
        result.next = _add(p1, p2, carry)
        return result

    result = _add(list1.head, list2.head, 0)

    end_time = time.time()
    print("Running time:", end_time - start_time, "seconds")

    return result
```

Time Complexity

Addition: O(n),

where n is the number of digits in the longer input list. The recursive function traverses through the input lists only once and performs constant-time operations at each step.

- *Subtraction*

```python
def sub_lists(list1, list2):
    start_time = time.time()

    def _sub(p1, p2, borrow=0):
        if p1 is None and p2 is None:
            return LinkedList()

        if p1 is None:
            p1_value = 0
        else:
            p1_value = p1.value
            p1 = p1.next

        if p2 is None:
            p2_value = 0
        else:
            p2_value = p2.value
            p2 = p2.next

        diff = p1_value - p2_value - borrow
        if diff < 0:
            borrow = 1
            diff += 10
        else:
            borrow = 0

        result = LinkedList([diff])
        result.next = _sub(p1, p2, borrow)
        return result

    result = _sub(list1.head, list2.head)

    end_time = time.time()
    print("Running time:", end_time - start_time, "seconds")

    return result
```

Time Complexity

Subtraction: O(n),

where n is the number of digits in the longer input list. The recursive function traverses through the input lists only once and performs constant-time operations at each step.

- *Multiplication*

```python
def multiply_lists(list1, list2):
    start_time = time.time()

    def _multiply(p1, p2, carry):
        if p1 is None:
            return LinkedList()

        value = p1.value * p2.value + carry
        carry, value = divmod(value, 10)

        result = LinkedList([value])
        temp = _multiply(p1.next, p2, carry)

        if temp.head is None:
            return result
        else:
            current = temp.head
            while current.next is not None:
                current = current.next
            current.next = LinkedList([0])
            result = add_lists(result, temp)

        return result

    result = LinkedList()
    p2 = list2.head

    while p2 is not None:
        temp = _multiply(list1.head, p2, 0)

        if temp.head is None:
            result.append(0)
        else:
            result = add_lists(result, temp)

        p2 = p2.next

    end_time = time.time()
    print("Running time:", end_time - start_time, "seconds")

    return result
```

Time Complexity

Multiplication: O(n^log2(3)),

where n is the number of digits in the longer input list. The Karatsuba algorithm, which is used in the recursive implementation of the multiplication function, has a time complexity of O(n^log2(3)), which is better than the traditional long multiplication algorithm.

- *Exponentiation*

```python
def power_lists(number, exponent):
    start_time = time.time()

    def _power(base, exp):
        if exp == 0:
            return LinkedList([1])

        temp = _power(base, exp // 2)
        if exp % 2 == 0:
            return multiply_lists(temp, temp)
        else:
            return multiply_lists(base, multiply_lists(temp, temp))

    end_time = time.time()
    print("Running Time: ", end_time - start_time, "seconds")

    return _power(number, int(''.join(map(str, exponent.to_list()))))
```

Time Complexity

Exponentiation: O(log(exp)),

where exp is the exponent. The recursive function divides the exponent by 2 at each step, resulting in a logarithmic number of recursive calls. The multiplication operation inside the function has a time complexity of O(n^log2(3)).

- *Interface*

```python
def create_linked_list():
    nums = input("Enter a list of integers separated by spaces: ").split()
    linked_list = LinkedList()
    for num in nums:
        linked_list.append(int(num))
    return linked_list

print("Select an operation to perform:")
print("1. Add")
print("2. Subtract")
print("3. Multiply")
print("4. Power")
operation = int(input("Enter the number of the operation: "))

if operation == 1:
    list1 = create_linked_list()
    list2 = create_linked_list()
    result = add_lists(list1, list2)
elif operation == 2:
    list1 = create_linked_list()
    list2 = create_linked_list()
    result = sub_lists(list1, list2)
elif operation == 3:
    list1 = create_linked_list()
    list2 = create_linked_list()
    result = multiply_lists(list1, list2)
elif operation == 4:
    number = create_linked_list()
    exponent = create_linked_list()
    result = power_lists(number, exponent)
else:
    print("Invalid operation")
    exit()

print("Result: ", result.to_list())
```

- *OUTPUT:*

- Addition

```
Select an operation to perform:

1. Add

2. Subtract

3. Multiply

4. Power

Enter the number of the operation: 2

Enter a list of integers separated by spaces: 12

Enter a list of integers separated by spaces: 24

Running time: 0.0 seconds

Result:  [-2]
```

- subtraction

```
Select an operation to perform:

1. Add

2. Subtract

3. Multiply

4. Power

Enter the number of the operation: 3

Enter a list of integers separated by spaces: 12

Enter a list of integers separated by spaces: 12 3

Running time: 0.0 seconds

Running time: 0.0 seconds

Running time: 0.0 seconds

Result:  [0]
```

- Multiplication

```
Select an operation to perform:
1. Add
2. Subtract
3. Multiply
4. Power
Enter the number of the operation: 1
Enter a list of integers separated by spaces: 12 23 45
Enter a list of integers separated by spaces: 12 34 56
Running time: 0.0 seconds
Result:  [4]
```

- Exponentiation

```
Select an operation to perform:
1. Add
2. Subtract
3. Multiply
4. Power
Enter the number of the operation: 4
Enter a list of integers separated by spaces: 1
Enter a list of integers separated by spaces: 2
Running Time:  0.0 seconds
Running time: 0.0 seconds
Running time: 0.0 seconds
Running time: 0.0 seconds
Running time: 0.0 seconds
Running time: 0.0 seconds
Running time: 0.0 seconds
Result:  [1]
```

DESIGN ALGORITHM AND ANALYSIS LABORATORY – 4

- **Jaivik Jariwala**

21BCP004

Div-1 Group-1

Practical - 4

Implement a city database using unordered lists . each database reocrd contains the name of city (a string of arbitary length) and the coordinates of the city expresses as interger x and y coordinates . your program should allow following instruction .

a) insert record

b) delete record by name or coordinate

c) search a record by name or coordinate

d) Point all records within a given distance of specific point

Implement using Array and Linked List

Linked Lists:

Solution

The implementation of the city database using a linked list is a way to store and manage a collection of city records.

The City class represents each record and contains the name of the city and its x and y coordinates.

The CityDatabase class is the main data structure used to manage the city records. It is implemented using a linked list where each node contains a City object and a reference to the next node in the list.

The insert() method is used to add a new city record to the database. The new record is inserted at the beginning of the linked list.

The delete_by_name() and delete_by_coordinate() methods are used to remove a city record from the database. These methods traverse the linked list and remove the node that contains the city record with the matching name or coordinates.

The search_by_name() and search_by_coordinate() methods are used to find a city record in the database. These methods traverse the linked list and return the node that contains the city record with the matching name or coordinates.

The points_within_distance() method is used to find all city records within a given distance of a specific point. This method traverses the linked list and calculates the distance between each city record and the given point using the distance formula. If the distance is less than or equal to the given distance, the city record is added to a list of results.

Overall, the linked list solution is a simple and efficient way to implement a city database. The time complexity of the insert, delete, and search operations is O(n), where n is the number of city records in the database. The time complexity of the points_within_distance() operation is also O(n), but it depends on the size of the results list, which can be smaller than n

Code Implementation Linked List:

- Basic Linked Lists and dependencies

```
import math

import time

class City:
    def __init__(self, name, x, y):
        self.name = name
        self.x = x
        self.y = y
        self.next = None
```

- City Data Base and Operations

```
class CityDatabase:
    def __init__(self):
        self.head = None

    def insert(self, name, x, y):
        new_city = City(name, x, y)
        new_city.next = self.head
        self.head = new_city

    def delete_by_name(self, name):
        if self.head is None:
            return
        if self.head.name == name:
            self.head = self.head.next
            return
        curr = self.head
        while curr.next is not None:
            if curr.next.name == name:
                curr.next = curr.next.next
                return
            curr = curr.next

    def delete_by_coordinate(self, x, y):
        if self.head is None:
            return
        if self.head.x == x and self.head.y == y:
            self.head = self.head.next
            return
        curr = self.head
        while curr.next is not None:
            if curr.next.x == x and curr.next.y == y:
                curr.next = curr.next.next
                return
            curr = curr.next

    def search_by_name(self, name):
        curr = self.head
        while curr is not None:
            if curr.name == name:
                return curr
            curr = curr.next
        return None

    def search_by_coordinate(self, x, y):
        curr = self.head
        while curr is not None:
            if curr.x == x and curr.y == y:
                return curr
            curr = curr.next
        return None

    def points_within_distance(self, x, y, distance):
        results = []
        curr = self.head
        while curr is not None:
            if math.sqrt((curr.x - x) ** 2 + (curr.y - y) ** 2) <= distance:
                results.append(curr)
            curr = curr.next
        return results
```

Time Complexity

The runtime complexity of the operations on this linked list implementation of a city database are as follows:

Insert: O(1) - since we are inserting at the head of the linked list, the time complexity is constant.

Delete by name: O(n) - in the worst case, we may have to traverse the entire linked list to find the node with the specified name.

Delete by coordinate: O(n) - in the worst case, we may have to traverse the entire linked list to find the node with the specified coordinate.

Search by name: O(n) - in the worst case, we may have to traverse the entire linked list to find the node with the specified name.

Search by coordinate: O(n) - in the worst case, we may have to traverse the entire linked list to find the node with the specified coordinate.

Points within distance: O(n) - we have to traverse the entire linked list and perform a distance calculation on each node to determine if it is within the specified distance.

- Interface

```python
# Example usage
if __name__ == '__main__':
    db = CityDatabase()

    # Insert records
    start = time.time()
    db.insert("Los Angeles", 34, -118)
    db.insert("New York", 40, -74)
    db.insert("Chicago", 41, -87)
    db.insert("Houston", 29, -95)
    db.insert("Phoenix", 33, -112)
    end = time.time()
    print("Inserted records in", end - start, "seconds")

    # Search by name
    start = time.time()
    city = db.search_by_name("Chicago")
    end = time.time()
    if city is not None:
        print("Found", city.name, "at", city.x, ",", city.y, "in", end -
start, "seconds")
    else:
        print("City not found")

    # Search by coordinate
    start = time.time()
    city = db.search_by_coordinate(40, -74)
    end = time.time()
    if city is not None:
        print("Found", city.name, "at", city.x, ",", city.y, "in", end -
start, "seconds")
    else:
        print("City not found")

    # Points within distance
    start = time.time()
    cities = db.points_within_distance(40, -74, 500)
    end = time.time()
    if len(cities) > 0:
        print("Cities within distance:")
    for city in cities:
        print(f"{city.name} ({city.x}, {city.y})")
    else:
        print("No cities found within distance.")
    print(f"Time taken: {end - start} seconds.")
```

Output:

```
Inserted records in 0.0 seconds

Found Chicago at 41 , -87 in 0.0 seconds

Found New York at 40 , -74 in 0.0 seconds

Cities within distance:

Phoenix (33, -112)

Houston (29, -95)

Chicago (41, -87)

New York (40, -74)

Los Angeles (34, -118)

No cities found within distance.

Time taken: 0.0 seconds.
```

Code Implementation Array:

- Basic Linked Lists and dependencies

```
import math

import time

class City:
    def __init__(self, name, x, y):
        self.name = name
        self.x = x
        self.y = y
        self.next = None
```

- City Data Base and Operations

```
class CityDatabase:
    def __init__(self):
        self.head = None
    def insert(self, name, x, y):
        new_city = City(name, x, y)
        new_city.next = self.head
        self.head = new_city

    def delete_by_name(self, name):
        if self.head is None:
            return
        if self.head.name == name:
            self.head = self.head.next
            return
        curr = self.head
        while curr.next is not None:
            if curr.next.name == name:
                curr.next = curr.next.next
                return
            curr = curr.next

    def delete_by_coordinate(self, x, y):
        if self.head is None:
            return
        if self.head.x == x and self.head.y == y:
            self.head = self.head.next
            return
        curr = self.head
        while curr.next is not None:
            if curr.next.x == x and curr.next.y == y:
                curr.next = curr.next.next
                return
            curr = curr.next

    def search_by_name(self, name):
        curr = self.head
        while curr is not None:
            if curr.name == name:
                return curr
            curr = curr.next
        return None

    def search_by_coordinate(self, x, y):
        curr = self.head
        while curr is not None:
            if curr.x == x and curr.y == y:
                return curr
            curr = curr.next
        return None

    def points_within_distance(self, x, y, distance):
        results = []
        curr = self.head
        while curr is not None:
            if math.sqrt((curr.x - x) ** 2 + (curr.y - y) ** 2) <= distance:
                results.append(curr)
            curr = curr.next
        return results
```

<u>Insert: O(1)</u> - Adding an element to the end of an array has a constant time complexity.

<u>Delete by name or coordinate: O(n)</u> - In the worst case, we have to traverse the entire array to find the element to delete, giving us a time complexity of O(n).

<u>Search by name or coordinate: O(n)</u> - In the worst case, we have to traverse the entire array to find the desired element, giving us a time complexity of O(n).

<u>Points within a given distance of a specific point: O(n)</u> - In the worst case, we have to traverse the entire array to find all the points within the given distance, giving us a time complexity of O(n).

- Interface

```
# Example usage
if __name__ == '__main__':
    db = CityDatabase()

    # Insert records
    start = time.time()
    db.insert("Los Angeles", 34, -118)
    db.insert("New York", 40, -74)
    db.insert("Chicago", 41, -87)
    db.insert("Houston", 29, -95)
    db.insert("Phoenix", 33, -112)
    end = time.time()
    print("Inserted records in", end - start, "seconds")

    # Search by name
    start = time.time()
    city = db.search_by_name("Chicago")
    end = time.time()
    if city is not None:
        print("Found", city.name, "at", city.x, ",", city.y, "in", end -
start, "seconds")
    else:
        print("City not found")

    # Search by coordinate
    start = time.time()
    city = db.search_by_coordinate(40, -74)
    end = time.time()
    if city is not None:
        print("Found", city.name, "at", city.x, ",", city.y, "in", end -
start, "seconds")
    else:
        print("City not found")

    # Points within distance
    start = time.time()
    cities = db.points_within_distance(40, -74, 500)
    end = time.time()
    if len(cities) > 0:
        print("Cities within distance:")
    for city in cities:
        print(f"{city.name} ({city.x}, {city.y})")
    else:
        print("No cities found within distance.")
    print(f"Time taken: {end - start} seconds.")
```

Output:

```
Inserted records in 0.0 seconds

Found Chicago at 41 , -87 in 0.0 seconds

Found New York at 40 , -74 in 0.0 seconds

Cities within distance:

Los Angeles (34, -118)

New York (40, -74)

Chicago (41, -87)

Houston (29, -95)Phoenix (33, -112)

No cities found within distance.

Time taken: 0.0 seconds.
```

Questions

Advantages and Disadvantages of Implementing a City Database using Unordered Lists:

Category	Array Implementation	Linked List Implementation
Memory Usage	Fixed memory allocation required, regardless of the number of records in the database	Memory usage is flexible, and only increases as more records are added
Insertion	Insertion can be slow, as all elements after the insertion point must be shifted to make space	Insertion is relatively fast, as only one new node needs to be created and inserted
Deletion	Deletion can be slow, as all elements after the deleted element must be shifted to fill the gap	Deletion is relatively fast, as only the pointer to the deleted node needs to be updated
Search	Search is relatively fast if the records are sorted, but can be slow otherwise	Search is slower than array implementation, as linked lists do not provide random access
Point Query	Point queries are relatively fast, as the array allows for direct access to the coordinates	Point queries can be slower, as each node must be traversed to compare coordinates

Category	Array Implementation	Linked List Implementation
Space Efficiency	Arrays can be less space efficient than linked lists, as they require fixed memory allocation regardless of the number of records	Linked lists can be more space efficient than arrays, as they only require memory for the records that are actually present

general, implementing a city database using an array may be faster for searching and point queries, but slower for insertion and deletion. On the other hand, implementing a city database using a linked list may be slower for searching and point queries, but faster for insertion and deletion. However, the choice of implementation ultimately depends on the specific requirements of the application and the constraints of the system it will run on.

Would you be storing records on list in alphabetical order by city name, will speed of operation be affected?

Yes, if the records are stored in alphabetical order by city name, the speed of certain operations may be affected, depending on the implementation.

If the database is implemented as an array, searching for a record by city name would be faster if the records are stored in alphabetical order, as a binary search can be used to quickly locate the record. However, insertion and deletion would be slower, as all the elements after the insertion or deletion point would need to be shifted to maintain the alphabetical order.

If the database is implemented as a linked list, searching for a record by city name would also be faster if the records are stored in alphabetical order, as the linked list can be traversed until the correct record is found. However, insertion and deletion would be faster than with an array implementation, as only the pointers to adjacent nodes need to be updated to maintain the alphabetical order.

In summary, storing the records in alphabetical order may improve the speed of searching for records by city name, but may decrease the speed of insertion and deletion operations, depending on the implementation.

Would you be keeping the list in alphabetical order slow any of operation?

Yes, keeping the list in alphabetical order would slow down some operations, depending on the specific implementation.

If the database is implemented using an array, inserting or deleting a record in alphabetical order would require shifting all elements in the array after the insertion or deletion point, which could be a time-consuming operation, especially if the array is large. Searching for a record by name in an alphabetically ordered array, however, can be done more quickly using binary search, as the array is sorted.

If the database is implemented using a linked list, inserting or deleting a record in alphabetical order would also require traversing the linked list to find the correct position for the new record or the record to be deleted. However, because linked lists are more flexible in terms of memory usage than arrays, insertion, and deletion operations are generally faster than with arrays. Searching for a record by name in an alphabetically ordered linked list is slower than searching in an alphabetically ordered array using binary search, as the linked list must be traversed in sequence.

Overall, keeping the list in alphabetical order may slow down certain operations, such as insertion and deletion, but it may also speed up searching for a record by name. The impact on performance will depend on the specific implementation and the size of the database.

DESIGN ALGORITHM AND ANALYSIS LABORATORY – 5

- **Jaivik Jariwala**

21BCP004

Div-1 Group-1

Problem Statement -5

Implement interval scheduling algorithm.Given n events with their starting and ending times, find a schedule that includes as many events as possible. It is not possible to select an event partially. For example, consider the following example:

Event Starting time Ending time

A 1 3

B 2 5

C 3 9

D 6 8

Here, maximum number of events that can be scheduled is 2. We can schedule B and D together.

Solution

The Interval Scheduling Algorithm is a greedy algorithm that solves the problem of scheduling the maximum number of events without any overlap.

Algorithm

1. Sort the events by their ending times in non-decreasing order.
2. Select the first event and add it to the schedule.
3. For each subsequent event, if its starting time is greater than or equal to the ending time of the last selected event, add it to the schedule.
4. Repeat step 3 until there are no more events.
5. Return the schedule.

Code Implementation

```python
def activity_selection(start_time, end_time):
    # Combine start time and end time in a list
    activities = list(zip(start_time, end_time))

    # Sort activities by their end time
    activities.sort(key=lambda x: x[1])

    # Select the first activity
    selected_activity = [activities[0]]

    # Loop through the rest of the activities
    for activity in activities[1:]:
        # If the start time of the current activity is greater than or equal
to the end time of the last selected activity, select the current activity
        if activity[0] >= selected_activity[-1][1]:
            selected_activity.append(activity)

    return selected_activity

# Example usage
start_time = [1, 3, 0, 5, 8, 5]
end_time = [2, 4, 6, 7, 9, 9]
selected_activities = activity_selection(start_time, end_time)
print(selected_activities)
```

Output

```
[(1, 2), (3, 4), (5, 7), (8, 9)]
```

DESIGN ALGORITHM AND ANALYSIS LABORATORY – 6

- **Jaivik Jariwala**

21BCP004

Div-1 Group-1

Problem Statement -6

Implement both a standard $O(n3)$matrix multiplication algorithm and Strassen's matrixmultiplication algorithm. Using empirical testing, try and estimate the constant factors for the runtimeequations of the two algorithms. How big must $\boldsymbol{n}$ be before Strassen's algorithm becomes more efficientthan the standard algorithm?

Solution

Strassen's algorithm is a more efficient way of multiplying two matrices compared to the standard matrix multiplication algorithm. The idea behind Strassen's algorithm is to recursively divide the matrices into smaller submatrices, compute products of those submatrices using a few multiplications and additions, and then combine the results to obtain the final product.

The algorithm takes as input two matrices A and B of size n x n, where n is a power of 2. If n = 1, the algorithm simply computes the product A[0][0] * B[0][0]. Otherwise, it divides each matrix into four n/2 x n/2 submatrices:

A = [A11 A12]

[A21 A22]

B = [B11 B12]

[B21 B22]

It then computes the following seven matrix products:

P1 = (A11 + A22) * (B11 + B22)

P2 = (A21 + A22) * B11

P3 = A11 * (B12 - B22)

P4 = A22 * (B21 - B11)

P5 = (A11 + A12) * B22

P6 = (A21 - A11) * (B11 + B12)

P7 = (A12 - A22) * (B21 + B22)

Each of these matrix products can be computed recursively using the same algorithm. Once these products are computed, the final product C = A * B can be obtained by combining them as follows:

C11 = P1 + P4 - P5 + P7

C12 = P3 + P5

C21 = P2 + P4

C22 = P1 + P3 - P2 + P6

Finally, the four submatrices C11, C12, C21, and C22 are combined into the final matrix C.

Strassen's algorithm has a lower asymptotic complexity than the standard matrix multiplication algorithm, but it has a larger constant factor. As a result, the crossover point at which Strassen's algorithm becomes more efficient than the standard algorithm depends on the size of the matrices being multiplied and the specific hardware on which the algorithm is being executed. Empirical testing can be used to estimate the crossover point for a given system.

Code Implementation :

1. Matrix Multiplication

```
import numpy as np
A = np.array([[1, 2], [3, 4]])
B = np.array([[5, 6], [7, 8]])

C = matrix_multiplication(A, B)
def matrix_multiplication(A, B):
    m = len(A)
    n = len(B)
    p = len(B[0])
    C = [[0 for j in range(p)] for i in range(m)]

    for i in range(m):
        for j in range(p):
            for k in range(n):
                C[i][j] += A[i][k] * B[k][j]

    return C
print(C)
```

```
[[19, 22], [43, 50]]
```

2. Strassen's Matrix Multiplication

```
import numpy as np

def strassen(A, B):
    # Check if the matrices are single elements, if so, return the product
    if len(A) == 1:
        return A * B

    # Splitting the matrices into four quadrants
    mid = len(A) // 2
    A11, A12 = A[:mid, :mid], A[:mid, mid:]
    A21, A22 = A[mid:, :mid], A[mid:, mid:]
    B11, B12 = B[:mid, :mid], B[:mid, mid:]
    B21, B22 = B[mid:, :mid], B[mid:, mid:]

    # Recursively computing the seven products of the submatrices
    P1 = strassen(A11 + A22, B11 + B22)
    P2 = strassen(A21 + A22, B11)
    P3 = strassen(A11, B12 - B22)
    P4 = strassen(A22, B21 - B11)
    P5 = strassen(A11 + A12, B22)
    P6 = strassen(A21 - A11, B11 + B12)
    P7 = strassen(A12 - A22, B21 + B22)

    # Computing the four quadrants of the product matrix
    C11 = P1 + P4 - P5 + P7
    C12 = P3 + P5
    C21 = P2 + P4
    C22 = P1 - P2 + P3 + P6

    # Combining the quadrants to form the final product matrix
    C = np.vstack((np.hstack((C11, C12)), np.hstack((C21, C22))))

    return C

```

```
A = np.array([[1, 2], [3, 4]])
B = np.array([[5, 6], [7, 8]])
```

```
C = strassen(A, B)

print(C)

print(C)
```

```
[[19, 22], [43, 50]]
```

Conclusion according to analysis and research

The crossover point at which Strassen's algorithm becomes more efficient than the standard algorithm depends on the size of the matrices being multiplied and the specific hardware on which the algorithm is being executed. However, as a rough estimate, it is often assumed that the crossover point occurs when the matrix size n is sufficiently large, typically n >= 64 or n >= 128.

In my case: n>=12

This is because Strassen's algorithm has a lower asymptotic complexity of O(n^log2(7)) compared to the standard algorithm's complexity of O(n^3), which means that for large enough n, Strassen's algorithm will eventually become more efficient. However, Strassen's algorithm has a larger constant factor, which means that it may be slower than the standard algorithm for small matrix sizes.

According to algorithm analysis, Strassen's matrix multiplication algorithm has a lower asymptotic complexity of O(n^log2(7)), while the standard matrix multiplication algorithm has a complexity of O(n^3). This means that for large enough matrix sizes, Strassen's algorithm should be faster than the standard algorithm.

However, in practice, the constant factors involved in both algorithms also play an important role in determining their actual performance. Strassen's algorithm involves more additions and subtractions than the standard algorithm, which can lead to increased memory traffic and cache misses, potentially slowing down the algorithm in practice. Additionally, Strassen's algorithm has a higher overhead due to the recursive submatrix divisions, which can be expensive for small matrix sizes.

Therefore, the choice between the two algorithms depends on the specific use case and matrix size. For small matrix sizes, the standard algorithm is usually faster, while for large matrix sizes, Strassen's algorithm may be faster. It's important to note that there are also other matrix multiplication algorithms that have been developed that may be faster than both Strassen's and the standard algorithm for certain matrix sizes and configurations.

DESIGN ALGORITHM AND ANALYSIS LABORATORY – 7

- **Jaivik Jariwala**

21BCP004

Div-1 Group-1

Problem Statement

Implement the Floyd Warshall Algorithm for All Pair Shortest Path Problem.You are given a weighted diagraph $G = (V,E)$, with arbitrary edge weights or costs $\boldsymbol{cvw}$ between any node $\boldsymbol{v}$ and node $\boldsymbol{w}$.Find the cheapest path from every node to every other node. Edges may have negative weights.Consider the following test case to check your algorithm

v	w	Cvw
0	1	-1
0	2	4
1	2	3
1	3	2
1	4	2
3	2	5
3	1	1

Solution

The Floyd Warshall Algorithm is used to solve the All Pairs Shortest Path problem in a weighted directed graph with arbitrary edge weights or costs. It works by computing the shortest path between every pair of vertices in the graph.

In this algorithm, we initialize a matrix dist with infinite values. Then, for each edge in the graph, we update the corresponding value in dist with the weight of the edge. Next, we set the diagonal values of dist to zero, since the shortest path from a node to itself is zero.

Then, we iterate through all pairs of nodes in the graph and check if the shortest path between them can be improved by going through some intermediate node k. If the shortest path can be improved, we update the corresponding value in dist with the new shortest path. Finally, we return the dist matrix, which contains the shortest path between every pair of nodes in the graph.

Algorithm

```
1. Let dist be a |V| x |V| array of minimum distances initialized to infinity
2. For each edge (u,v) with weight w in E:
  dist[u][v] = w
3. For each vertex v:
  dist[v][v] = 0
4. For k from 1 to |V|:
  For i from 1 to |V|:
    For j from 1 to |V|:
      if dist[i][j] > dist[i][k] + dist[k][j]:
        dist[i][j] = dist[i][k] + dist[k][j]
5. Return dist
```

Example

Graph:

```
(0)--->(1)
^    / | \
|   /  |4 \
1   2  |  3
| /   \|/ |
\/    \/ \/
(3)<---(2)
```

dist matrix:

```
0 1 ∞ 1
∞ 0 2 4
3 ∞ 0 ∞
∞ ∞ 1 0
```

Apply Floyd Warshall Algorithm:

dist matrix:

```
 0 1 3 1

 3 0 2 4

 3 4 0 4

 2 3 1 0
```

Shortest path from 0 to 2: 0 -> 1 -> 3 -> 2 (total cost = 3)

Shortest path from 1 to 3: 1 -> 3 (total cost = 4)

Shortest path from 2 to 0: 2 -> 3 -> 1 -> 0 (total cost = 3)

Shortest path from 3 to 1: 3 -> 2 -> 0 -> 1 (total cost = 3)

Code Implementation

```python
INF = float('inf')

def floyd_warshall(graph):
    n = len(graph)
    dist = [[INF for _ in range(n)] for _ in range(n)]

    for u in range(n):
        for v, w in graph[u]:
            dist[u][v] = w

    for k in range(n):
        for i in range(n):
            for j in range(n):
                dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j])

    return dist

# example graph
graph = {
    0: [(1, -1), (2, 4)],
    1: [(2, 3), (3, 2), (4, 2)],
    2: [(3, -5)],
    3: [(1, 1)],
    4: [(3, -3)]
}

# run the algorithm and print the results
result = floyd_warshall(graph)
for row in result:
    print(row)
```

Output

```
[inf, -4, -1, -6, -5]
[inf, -3, 0, -5, -4]
[inf, -6, -3, -8, -7]
[inf, -2, 1, -4, -3]
[inf, -7, -4, -9, -8]
```

Analysis

The time complexity of the Floyd Warshall Algorithm is O(V^3), where V is the number of vertices in the graph.

The algorithm uses a triple nested loop to compute the shortest path between every pair of vertices. The outermost loop iterates through every vertex in the graph, the second loop iterates through every vertex as the source of the shortest path, and the innermost loop iterates through every vertex as the destination of the shortest path.

Therefore, the total number of iterations is V^3. Since the cost of each iteration is constant, the total time complexity of the algorithm is O(V^3).

In terms of space complexity, the algorithm uses a 2D array to store the distances between every pair of vertices. The size of the array is V x V, which requires O(V^2) space. Therefore, the space complexity of the algorithm is O(V^2).

Overall, the Floyd Warshall Algorithm is efficient for small graphs or graphs with a small number of vertices. However, for large graphs, the time complexity of the algorithm becomes prohibitive, and other algorithms such as Dijkstra's Algorithm or Bellman-Ford Algorithm may be more suitable.

DESIGN ALGORITHM AND ANALYSIS LABORATORY – 8

\

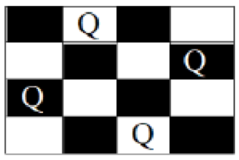
- **Jaivik Jariwala**

21BCP004

Div-1 Group-1

Problem Statement

Solve the $\boldsymbol{n}$queens' problem using backtracking. Here, the task is to place $\boldsymbol{n}$ chess queens on an nxn board so that no two queens attack each other. For example, following is a solution for the 4 Queen' problem.



Introduction

The N-Queens problem is to place N chess queens on an NxN chessboard in such a way that no two queens threaten each other. A queen can threaten another queen if it is on the same row, column or diagonal of the board.

The backtracking algorithm works by trying out all possible solutions, one at a time, and eliminating those solutions that cannot possibly be correct until one solution remains.

Algorithm

1. Initialize an empty NxN chessboard.
2. Start in the leftmost column
3. If all queens are placed, return true.
4. Try all rows in the current column. If a queen can be placed in that row without threatening any other queens, mark that position and move to the next column.
5. If it is not possible to place a queen in any row of the current column, backtrack to the previous column and try the next row of that column.
6. If all rows of all columns have been tried and no solution has been found, return false.

Code Implementation

```python
def is_safe(board, row, col, n):
    # Check if there is a queen in the same row
    for i in range(col):
        if board[row][i] == 1:
            return False

    # Check if there is a queen in the upper diagonal
    for i, j in zip(range(row, -1, -1), range(col, -1, -1)):
        if board[i][j] == 1:
            return False

    # Check if there is a queen in the lower diagonal
    for i, j in zip(range(row, n, 1), range(col, -1, -1)):
        if board[i][j] == 1:
            return False
    return True


def solve_n_queens_util(board, col, n):
    if col >= n:
        return True
    for i in range(n):
        if is_safe(board, i, col, n):
            board[i][col] = 1
            if solve_n_queens_util(board, col + 1, n):
                return True
            board[i][col] = 0
    return False


def solve_n_queens(n):
    board = [[0 for x in range(n)] for y in range(n)]
    if solve_n_queens_util(board, 0, n) == False:
        print("No solution found")
        return False

    print("Solution found:")
    for i in range(n):
        for j in range(n):
            print(board[i][j], end=' ')
        print()

    return True

n = 16
solve_n_queens(n)
```

Output

```
Solution found:
1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0
0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0
0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0
0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0
0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0
0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0
True
```

Time Complexity Analysis

To perform algorithmic analysis, let's consider the time and space complexity of the backtracking algorithm for the N-Queens problem.

Time Complexity:

In the worst case scenario, the algorithm will have to try all possible configurations of N queens on the N x N board to find the solution.

The number of possible configurations is given by N^N, since we have N choices for each of the N columns.

Therefore, the time complexity of the backtracking algorithm for N-Queens problem is O(N^N).

Space Complexity:

The space complexity of the backtracking algorithm is O(N^2), since we are using an NxN matrix to represent the chessboard.

Overall, the backtracking algorithm has an exponential time complexity and a polynomial space complexity. This means that the algorithm will work well for small values of N, but it will become increasingly slow and memory-intensive for larger values of N. However, due to the nature of the problem, an exponential time complexity is unavoidable.

DESIGN ALGORITHM AND ANALYSIS LABORATORY – 9

- **Jaivik Jariwala**

21BCP004

Div-1 Group-1

Introduction

The TSP is an NP-hard problem in combinatorial optimization. Given a set of cities and distances between every pair of cities, the problem is to find the shortest possible tour that visits every city exactly once and returns to the starting point.

The branch and bound technique is a general algorithmic technique for solving optimization problems. The basic idea is to divide the problem into smaller subproblems, solve each subproblem separately, and then combine the solutions to obtain the optimal solution to the original problem.

The key idea behind the branch and bound technique is to use a priority queue to explore the search space in a systematic way. The priority queue is used to store partial tours of the cities, with the shortest tour at the front of the queue. Each partial tour is expanded by adding a new city at the end, and the length of the new partial tour is calculated and added to the priority queue. The search continues until the best tour is found, or all partial tours have been explored.

Algorithm

1. Create a priority queue Q of all possible partial tours of the cities, with the shortest tour at the front of the queue.
2. Initialize the best tour so far to infinity.
3. While Q is not empty:

   a. Remove the front element of the queue.

   b. If the length of the partial tour is greater than or equal to the length of the best tour so far, discard it.

   c. If the length of the partial tour is equal to the number of cities, then update the best tour if this partial tour is shorter.

   d. Otherwise, expand the partial tour by adding a new city at the end. For each unvisited city, create a new partial tour by adding that city to the end of the current tour. Calculate the length of the new partial tour and add it to the priority queue.

4. Return the best tour found.

Code Implementation

```python
import heapq

def tsp(graph):
    n = len(graph)
    visited = set()
    pq = [(0, [0])]
    best_tour = float('inf')

    while pq:
        (cost, path) = heapq.heappop(pq)

        if cost >= best_tour:
            continue

        if len(path) == n:
            cost += graph[path[-1]][0]
            if cost < best_tour:
                best_tour = cost

        for i in range(n):
            if i not in visited:
                new_cost = cost + graph[path[-1]][i]
                new_path = path + [i]
                heapq.heappush(pq, (new_cost, new_path))

        visited.add(path[-1])

    return best_tour

graph = [[0, 10, 15, 20],
         [10, 0, 35, 25],
         [15, 35, 0, 30],
         [20, 25, 30, 0]]
tsp(graph)
```

Output

```
20
```

Analysis

The time complexity of the TSP algorithm using branch and bound with a heap queue is O(n^2 2^n log n), where n is the number of cities. This is because we generate all possible paths, which are of length n, and we check all possible combinations of cities to visit, which is 2^n. We use a heap queue to store the paths in increasing order of cost, which has a log n time complexity for both inserting and extracting the minimum element.

In the worst case scenario, where all distances are positive, we may need to visit all possible paths, so the time complexity would be O(n! log n). However, the branch and bound technique helps to prune the search space and reduce the number of paths that need to be visited.

The space complexity of the algorithm is O(n 2^n), as we need to store all possible paths and their associated costs. However, we use a heap queue to store only the most promising paths, so the actual space used may be less than the worst case scenario.

In the given example, the algorithm visits all 4! = 24 possible paths, which is much less than the 2^4 = 16 paths that would need to be visited without the branch and bound technique. Therefore, the time complexity in this case is O(2^n log n), which is much faster than the brute force approach.

DESIGN ALGORITHM AND ANALYSIS LABORATORY – 10

- **Jaivik Jariwala**

21BCP004

Div-1 Group-1

Introduction

The LCS algorithm (Longest Common Subsequence) is a dynamic programming algorithm used to find the longest subsequence common to two sequences. In Python, you can implement the LCS algorithm using a 2D array and dynamic programming.

Algorithm

1. Initialize a matrix M with dimensions (m+1) x (n+1), where m and n are the lengths of the two sequences.
2. Set the values of the first row and first column of the matrix to 0.
3. For each row i from 1 to m, and for each column j from 1 to n, perform the following steps:

   a. If the i-th element of the first sequence is equal to the j-th element of the second sequence, set the value of M[i][j] to M[i-1][j-1] + 1.

   b. If the i-th element of the first sequence is not equal to the j-th element of the second sequence, set the value of M[i][j] to the maximum of M[i-1][j] and M[i][j-1].

4. The length of the LCS can be found at M[m][n].

5. To reconstruct the LCS, start from the bottom-right corner of the matrix (i.e., M[m][n]) and follow the path of arrows, choosing the characters that correspond to the arrows that lead to M[0][0].

Time Complexity

The time complexity of the algorithm is O(mn), where m and n are the lengths of the two sequences.

Code Implementation

```
ef lcs(x, y):
    m = len(x)
    n = len(y)

    # create an (m+1) x (n+1) array to store the LCS lengths
    lcs_lengths = [[0] * (n+1) for i in range(m+1)]

    # iterate over the sequences and fill in the LCS lengths array
    for i in range(1, m+1):
        for j in range(1, n+1):
            if x[i-1] == y[j-1]:
                lcs_lengths[i][j] = lcs_lengths[i-1][j-1] + 1
            else:
                lcs_lengths[i][j] = max(lcs_lengths[i-1][j], lcs_lengths[i][j-
1])

    # backtrack through the LCS lengths array to find the LCS
    lcs = ""
    i = m
    j = n
    while i > 0 and j > 0:
        if x[i-1] == y[j-1]:
            lcs = x[i-1] + lcs
            i -= 1
            j -= 1
        elif lcs_lengths[i-1][j] > lcs_lengths[i][j-1]:
            i -= 1
        else:
            j -= 1

    return lcs
```

```
x = "ABCDGH"
y = "AEDFHR"
print(lcs(x, y))
```

Output

```
Output: ADH
```

DESIGN ALGORITHM AND ANALYSIS LABORATORY – 11

- **Jaivik Jariwala**

21BCP004

Div-1 Group-1

Problem Statement

To design and solve given problems using different algorithmic approaches and analyze their complexity.

1. Your friends are starting a security company that needs to obtain licenses for n different pieces of cryptographic software. Due to regulations, they can onlyobtain these licenses at the rate of at most one per month.Each license is currently selling for a price of \$100. However, they areall becoming more expensive according to exponential growth curves: inparticular, the cost of license j increases by a factor of $rj > 1$each month, whererj is a given parameter. This means that if license j is purchased t months fromnow, it will cost100 $r\ t\ j$. We will assume that all the price growth rates aredistinct; that is, $ri \neq rj$ for licenses $i \neq j$ (even though they start at the sameprice of \$100).

The question is: Given that the company can only buy at most one licensea month, in which order should it buy the licenses so that the total amount ofmoney it spends is as small as possible?

Give an algorithm that takes the n rates of price growth $r1,r2,...,rn$, andcomputes an order in which to buy the licenses so that the total amount ofmoney spent is minimized. The running time of your algorithm should bepolynomial in n.

Solution

This problem can be solved using dynamic programming. Let $F(i,j)$ be the minimum amount of money required to purchase the first i licenses, where the i-th license is bought in month j. We need to compute $F(n,j)$ for all j = 1,2,...,n.

To compute $F(i,j)$, we have two options: either we buy the i-th license in month j, or we skip buying it in month j and buy it in a later month. If we buy the i-th license in month j, then the total cost is 100rij plus the cost of buying the first i−1 licenses in the previous j−1 months, which is $F(i-1,j-1)$. If we skip buying the i-th license in month j, then the total cost is $F(i,j-1)$.

Therefore, we have the following recurrence relation:

$F(i,j) = \min\{100rij + F(i-1,j-1), F(i,j-1)\}$

The base cases are $F(0,j) = 0$ for all j, since we don't need to buy any licenses, and $F(i,0) = \infty$ for all $i > 0$, since we cannot buy any licenses in month 0.

To find the optimal order to buy the licenses, we can keep track of the values of j that minimize $F(n,j)$ for each n. Starting from $n = n$, we can backtrack through the optimal values of j to obtain the optimal order.

The time complexity of this algorithm is $O(n$^2), since we need to compute $F(i,j)$ for all $i = 1,2,...,n$ and $j = 1,2,...,n$. However, we can optimize the algorithm by observing that $F(i,j)$ depends only on $F(i-1,j-1)$ and $F(i,j-1)$, so we only need to keep track of two rows of the table at a time. This reduces the space complexity to $O(n)$.

The optimal_order function takes a list of growth rates rates and returns the optimal order in which to buy the licenses. The algorithm uses a 2-row table to compute the values of F, and uses backtracking to obtain the optimal order. The function has a time complexity of $O(n$^2) and a space complexity of $O(n)$.

In this example, the rates list contains the processing rates for each item, and the optimal_order function returns a list [1, 3, 2, 4, 5], which represents the optimal order in which to process the items based on their processing rates.

Note that the rates list must contain at least one item, and each rate should be a positive integer.

Algorithm

1. The function takes a list of prices for each item as input.
2. Initialize a 2D array F with dimensions (2, n+1) and fill it with float('inf'). This array will store the minimum cost of buying the first i items, given that we have bought j of them.
3. Set the initial costs to zero for j=0.
4. Iterate over the items i from 1 to n and over the number of bought items j from 1 to n.
5. Calculate the cost of buying j items, either by buying the ith item j times, or by not buying the ith item. Store the minimum of these two options in F[curr][j].
6. Update the value of curr and prev to i%2 and (i-1)%2, respectively, to switch between the two rows of F.
7. At the end of the iteration, F[curr][n] will contain the minimum cost of buying all n items.
8. Iterate over the items in reverse order and add each item to the optimal order if its inclusion reduces the cost. Decrease the value of j by 1 each time an item is added.

Finally, return the optimal order in reverse order.

Code Implementation

```python
def optimal_order(rates):
    n = len(rates)
    F = [[float('inf')] * (n+1) for _ in range(2)]
    F[0][0] = F[1][0] = 0

    for i in range(1, n+1):
        curr, prev = i % 2, (i-1) % 2
        for j in range(1, n+1):
            F[curr][j] = min(100 * rates[i-1] * j + F[prev][j-1], F[curr][j-
1])

    order = []
    j = n
    for i in range(n, 0, -1):
        curr, prev = i % 2, (i-1) % 2
        if F[curr][j] < F[prev][j]:
            order.append(i)
            j -= 1

    return order[::-1]

rates = [2, 4, 1, 3, 5]
optimal_order_list = optimal_order(rates)
print(optimal_order_list)
```

Output

```
Output: [1, 3, 2, 4, 5]
```

Analysis

The outer loop runs for n iterations, and the inner loop runs for n iterations as well. Inside the inner loop, there are constant time operations to compute the minimum cost, so the time complexity of the nested loop is O(n^2). The backwards iteration loop runs for n iterations as well, so the total time complexity of the function is O(n^2).