**Name: - Vikas Kumar Singh**

**Reg no: - 2022CA110**

**Algorithms Theory Assignment-2**

**Greedy Programming Strategy**

* Write pseudocode of the greedy algorithm for the change-making problem, with an amount n and coin denominations d1 > d2 > . . . > dm as its input. What is the time efficiency class of your algorithm?

GreedyChangeMaking(n, denominations):

coins\_used = []

for each coin in denominations:

while n >= coin:

add coin to coins\_used

n = n - coin

return coins\_used

Explanation:

1. Initialize an empty list coins\_used to keep track of the coins used to make the change.

Iterate through each coin denomination in the sorted list of denominations.

2. While the current denomination is less than or equal to the remaining amount n, add that denomination to coins\_used and subtract the denomination from n.

3. Repeat this process until the remaining amount n becomes zero.

Time Complexity: The time complexity of the greedy algorithm for the change-making problem depends on the number of denominations and the value of the input amount. Assuming there are 'm' denominations and the amount 'n':

In the worst-case scenario, where the denominations are not in a convenient order, the algorithm may need to iterate through all the denominations for each value of the amount 'n'. Therefore, the time complexity can be considered as O(m \* n).

However, if the number of denominations is fixed and doesn't depend on the value of 'n', then the time complexity can be simplified to O(n) since the loop through denominations would be constant.

* Apply greedy method for exploring Travelling Sales Man problem using greedy method. Verify Greedy works for the above problem and prove your hypothesis.

GreedyTSP(start\_city, distances):

num\_cities = number of cities

visited = array of size num\_cities initialized to false

tour = array of size num\_cities + 1

tour[0] = start\_city

visited[start\_city] = true

current\_city = start\_city

for i from 1 to num\_cities - 1:

next\_city = find\_nearest\_unvisited\_city(current\_city, distances, visited)

tour[i] = next\_city

visited[next\_city] = true

current\_city = next\_city

tour[num\_cities] = start\_city (returning to the start city)

return tour

Explanation:

1. Initialize arrays to keep track of visited cities (visited) and the tour.

2. Start from a given start\_city.

3. Iterate through the number of cities minus one times:

-Find the nearest unvisited city from the current\_city.

-Mark it as visited.

-Update the current\_city to the newly visited city.

-Add the chosen city to the tour.

4. Finally, return to the start\_city to complete the tour

* Write pseudo code for Indirect Sorting algorithm and determine its complexity.

IndirectSort(arr):

n = length of arr

indices = array of size n

// Initialize indices array with values 0 to n-1

for i from 0 to n-1:

indices[i] = i

MergeSort(arr, indices, 0, n-1)

return indices

MergeSort(arr, indices, low, high):

if low < high:

mid = (low + high) / 2

MergeSort(arr, indices, low, mid)

MergeSort(arr, indices, mid + 1, high)

Merge(arr, indices, low, mid, high)

Merge(arr, indices, low, mid, high):

left\_length = mid - low + 1

right\_length = high - mid

left = array of size left\_length

right = array of size right\_length

// Copy elements to temporary left and right arrays

for i from 0 to left\_length - 1:

left[i] = arr[indices[low + i]]

for j from 0 to right\_length - 1:

right[j] = arr[indices[mid + 1 + j]]

i = 0

j = 0

k = low

// Merge left and right arrays back into indices array

while i < left\_length and j < right\_length:

if left[i] <= right[j]:

indices[k] = indices[low + i]

i = i + 1

else:

indices[k] = indices[mid + 1 + j]

j = j + 1

k = k + 1

// Copy remaining elements of left and right arrays if any

while i < left\_length:

indices[k] = indices[low + i]

i = i + 1

k = k + 1

while j < right\_length:

indices[k] = indices[mid + 1 + j]

j = j + 1

k = k + 1

**Complexity Analysis:**

The Indirect Sorting algorithm using merge sort has a time complexity of O(n log n) where 'n' is the number of elements in the array.

The merge sort used here divides the array into halves until individual elements are obtained and then merges the smaller sorted arrays together.

The space complexity of the algorithm is O(n) as additional space is used for creating temporary arrays during the merge sort process and the 'indices' array to maintain the order of elements.

.