Hands-on Activity 12.1	
Algorithmic Strategies	
Course Code: CPE010	**Program:** Computer Engineering
Course Title: Data Structures and Algorithms	**Date Performed:** 10/28/25
Section: CPE21S4	**Date Submitted:** 10/28/25
Name(s): Francis Nikko I. Andoy	**Instructor:** Engr. Jimlord Quejado

A. Output(s) and Observation(s)

ILO A: Demonstrate an understanding of algorithmic strategies

Strategy	Algorithm	Analysis

Recursion	**Binary search**
	Algorithm BinarySearch(array, target, left, right)
	Input: sorted array of n elements, target value, left index, right index
	Output: index of target in array or -1 if not found
	Start
	if left > right then
	return -1 // base case: target not found
	mid ← (left + right) / 2
	if array[mid] == target then
	return mid
	else if array[mid] > target then
	return BinarySearch(array, target, left, mid - 1)
	else
	return BinarySearch(array, target, mid + 1, right)
	End
		A binary search is connected to recursive strategy. This is because of the way it operates by dividing the search array in half, using the sorted nature of the data. Since this is a recursive function that also has a base case when this is reached the algorithm will automatically exit the recursion program when the left index exceeds the right index, signaling that the target is not present. Each of the recursive calls will narrow the search size reducing the search logarithmically.

Brute force	**Linear search**
	Algorithm LinearSearch(array, target)
	Input: array of n elements, target value
	Output: index of target in array or -1 if not found
	
	Start
	for i from 0 to n - 1 do
	if array[i] == target then
	return i // target found at index i
	return -1 // target not found
	End
		Since the way linear search works is to check every element inside of the array, this only stops when the target is found. The connection between these two is there's no short cut or any optimization that speeds up the algorithm, just pure searching 1 by 1. That is how brute force works.

Backtracking	**Depth First**
	Algorithm DFS(graph, node, visited)
	  Input: graph as adjacency list, starting node, visited set
	  Output: none (prints or processes nodes in DFS order)
	
	  Start
	  if node is in visited then
	  return // already explored
	
	  mark node as visited
	  process node // e.g., print or store
	
	  for each neighbor in graph[node] do
	  DFS(graph, neighbor, visited)
	  End
		These 2 are connected because of how the DFS works, by using DFS we have to check the most left side of the tree then backtrack to the parent node. Just by explaining this we can see why it is connected. The nature of the DFS is always aligned to the backtracking strategy. A DFS also uses stack as when a path leads to a node that has already been visited or has no further options, the algorithm backtracks to the previous node. This allows DFS to explore alternative paths and ensures that all reachable nodes are eventually visited.

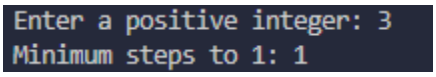
Greedy	**Selection sort**
	Algorithm SelectionSort(array)
	  Input: array of n elements
	  Output: sorted array in ascending order
	
	  Start
	  for i from 0 to n - 1 do
	  minIndex ← i
	
	  for j from i + 1 to n - 1 do
	  if array[j] < array[minIndex] then
	  minIndex ← j
	
	  if minIndex ≠ i then
	  swap array[i] with array[minIndex]
	  End
		Selection sort is related to the greedy strategy since it selects the most optimal local solution at each step with an optimistic view of getting a globally optimal solution. Each time, the algorithm picks the smallest element in the set which is not sorted and places it in its appropriate spot. This is a greedy decision in that a choice of a minimum at this stage would result in a sorted array at the decision point. This is a greedy principle of making optimal decisions here and now without looking in the future. Selection sort successfully goes through this till the entire array is sorted.

Divide-and-Conquer	Algorithm Merge(left, right)
	  Input: two sorted arrays
	  Output: merged sorted array
	
	  Start
	  result ← empty array
	
	  while left and right are not empty
		These two are connected because of their nature and how they operate. divide and conquer approach since it decomposes a problem into smaller problems, solves these smaller problems separately then assembles the solutions. The algorithm starts by partitioning the array in two parts,

	do
	if left[0] ≤ right[0] then
	append left[0] to result
	remove left[0] from left
	else
	append right[0] to result
	remove right[0] from right
	
	append any remaining elements from left to result
	append any remaining elements from right to result
	
	return result
	End
		irrespective of the data in the array. The two halves are then recursively sorted in the same fashion. This partitioning is repeated until one arrives at the base case, which is a subarray of a single element. This is also how a merge sort works, that is why they are connected.

Table 12-1. Algorithmic Strategies and Examples

Screen Shot	Analysis

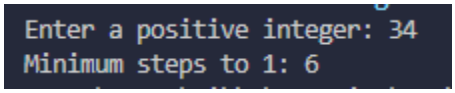

Code:
**#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
int getMinSteps(int n, vector<int>& memo) {
if (n == 1) return 0;
if (memo[n] != -1) return memo[n];
int steps = 1 + getMinSteps(n - 1, memo);
if (n % 2 == 0)
steps = min(steps, 1 + getMinSteps(n / 2, memo));
 	
if (n % 3 == 0)
steps = min(steps, 1 + getMinSteps(n / 3, memo));
return memo[n] = steps; // Store and return result
}
int main() {
int n;
cout << "Enter a positive integer: ";
cin >> n;
 	
if (n <= 0) {
cout << "Invalid input. Please enter a number greater than 0." << endl;
return 1;
}**
	Firstly, what is memoization? This memoization talks about reducing the time and effort when computing the problem. As for the given code we first have the base case which is used in the recursion function, since memoization is also connected to this strategy. Then after that the memoization will check if the new input has been computed already, without this it will recompute all the input of the user again and again. The computed input will be stored in the vector and that is why we need to use a vector when dealing with a memoization function. Next is just the solving part, after this it will store the computed value in steps to the memo[n], with this value stored the program can reuse the value if the value imputed is the same. Just like I said at first by doing memoization we can reduce the time and effort when computing because of this.

```
   vector<int> memo(n + 1, -1);      int result =
getMinSteps(n, memo);

   cout << "Minimum steps to 1: " << result << endl;
   return 0;
}
```

Table 12-2. Memoization Implementation

Screen Shot	Analysis


Code:
`#include <iostream>`
`#include <vector>`
`#include <algorithm>`
`using namespace std;`
 	
`int getMinSteps(int n) {`
`  vector<int> dp(n + 1);  // dp[i] stores min steps to reduce i to 1`
`  dp[1] = 0;  // base case: 1 requires 0 steps`
 	
`  for (int i = 2; i <= n; i++) {`
`  dp[i] = 1 + dp[i - 1];  // subtract 1`
 	
`  if (i % 2 == 0)`
`  dp[i] = min(dp[i], 1 + dp[i / 2]);  // divide by 2`
 	
`  if (i % 3 == 0)`
`  dp[i] = min(dp[i], 1 + dp[i / 3]);  // divide by 3`
`  }`
 	
`  return dp[n];`
`}`
 	
`int main() {`
`  int n;`
`  cout << "Enter a positive integer: ";`
`  cin >> n;`
 	
`  if (n <= 0) {`
`  cout << "Invalid input. Please enter a number greater than 0." << endl;`
`  return 1;`
`  }`
	In this code, it highlight the bottom up dynamic programming. This kind of program uses iterative instead of recursion. This first sets up a DP table as a vector to hold the minimum number of steps to each number on 1 to n. Base case The base case makes dp[1] = 0, which means that no steps are required to go to 1. The loop runs between 2 and n, in which the value of dp[i] is computed as the result of subtraction by 1 or conditional divisions by 2 or 3. In both cases, it assumes a subtraction of 1 followed by a test as to whether the division by 2 or 3 results in fewer steps using the min function. Input validation is dealt with in the main where any non-positive integers are rejected by the code. Output is also easy, and the result of the given n is printed. This approach is effective and scalable to regular constraints. It shows the iterative nature of dynamic programming, as opposed to recursive memoization.

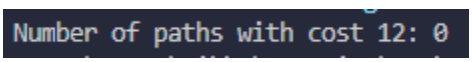
```
    int result = getMinSteps(n);
    cout << "Minimum steps to 1: " << result << endl;

    return 0;
}
```

Table 12-3. Bottom-Up Dynamic Programming Implementation

B. Answers to Supplementary Activity

Pseudocode	START Algorithm: Count Paths with Exact Cost Initialize: Define costMatrix as a 2D array: [[1,2,3], [4,6,5], [7,8,9]] Set targetCost = 12 Set rows = length of costMatrix Set cols = length of costMatrix[0] Call Function: result = countPaths(costMatrix, rows-1, cols-1, targetCost) Output: Print "Number of paths with cost " + targetCost + ": " + result END Algorithm Function: countPaths(costMatrix, m, n, cost) START Function If cost < 0, return 0 If m == 0 and n == 0: If cost == costMatrix[0][0], return 1 Else, return 0 If m < 0 or n < 0, return 0 Return countPaths(costMatrix, m-1, n, cost - costMatrix[m][n]) + countPaths(costMatrix, m, n-1, cost - costMatrix[m][n]) END Function
Solving problem by hand	

Working C++ Code	
Analysis of working code	This code actually uses a recursive solution to count the number of paths from the bottom-right to the top-left of the grid. It starts at the position (m, n) which indicate the last element inside the matrix What I mean by this is the cell at row 2 and column 2, which contains the number 9.The subtractive method of the current element is converted into a value and then subtracted in the remaining element cost, as shown in the matrix. Assuming that the remaining cost is the same as the value (0, 0), 1, then the program will deem the path valid. When m, n and cost are above zero, then the recursion takes place. For example, when the target cost is 12, the function will examine every possible route that will remove values in the matrix until the cost is 1. On each step, it either moves up or left and verifies the path to be valid. When the cost turns out to be negative then the function halts and gives 0 to that path. In case the indices become out of range, the function also gives 0. The base case is a scenario where the function goes to (0, 0); it is a scenario that tests whether the remaining cost is equal to the cell value.
Screenshots of demonstration	

```
Number of paths with cost 12: 0
```

C. Conclusion & Lessons Learned

In conclusion, I learnt how various algorithmic strategies are related to certain problem solving methods. One of the strategies that are of important is recursion, and I have observed its use in search, sort and pathfinding problems. It is applied in binary search, depth first search as well as solving optimization problems such as reducing a number to one.

Another thing I learned is that recursion is maximum when paired with memoization. Memoization is used to store already calculated values, hence recursive functions are quicker and more efficient. This saves time and the number of repetitive computations. This occurred to me in the minimum steps to one problem where memoization enabled the program to eliminate unnecessary work. It also facilitated the code and made it manageable and scalable. The knowledge of recursion assisted me to understand the principle of divide and conquer in sorting, in the case of merchant sort. It also demonstrated to me the depth-first search implementation with reversal on failure of a path by backtracking. These relationships confirm recursion is not only a tool, but a base in algorithm design. Now I also know brute force and greedy algorithms and their application to such problems as linear search and selection sort. Brute force is easy but tiresome whereas greedy algorithms select the best local option at a given step. I observed the greedy nature of selection sort which picks the smallest element that is present. I also got to know the concept of dynamic programming that is top-down and bottom-up. The bottom up version of the minimum steps problem depicted the way that we could develop solutions step by step. I have learnt the mechanism of cost based pathfinding through recursion as well as the way the function examines all available paths to arrive at a target cost. This made me understand how algorithms may be influenced by constraints such as cost, direction and structure. Overall, this activity allowed me to bridge the gap between theory and practices. It demonstrated to me the way various strategies can help to resolve various issues, and why the selection of the appropriate one should be important.

D. Assessment Rubric

E. External References