**LAB:01**

**Implement GCD algorithm and perform its analysis**

**THEORY:**

The Greatest Common Divisor (GCD) is the largest natural number that divides two numbers without leaving a remainder. GCD is also referred as Highest Common Factor (HFC) or Greatest Common Factor (GCF) or Greatest Common Measure (GCM). The Euclidean algorithm is the efficient algorithm to find GCD of two natural numbers.

The Euclidean algorithm is based on the principle that the greatest common divisor of two numbers does not change if the larger number is replaced by its difference with the smaller number.

**ALGORITHM:**

1. Start
2. Read any two number say x and y
3. If y==0, return the value of x as the answer and stop,
4. If x==0, return the value of y as the answer and stop,
5. Otherwise divide x by y and assign the value of the remainder to r
6. Assign the value of y to x and the value of r to y. Go to step 3.
7. Stop

**TIME COMPLEXITY:**

* Best case: O(1)
* Worst case: O(logn)
* Average case: O(logn)

**SPACE COMPLEXITY:**

* Space complexity: O(1)
* Auxiliary space complexity: O(1)

**SOURCE CODE:**

#include <stdio.h>

int main()

{

int n1, n2, i, gcd;

printf("Enter two integers: ");

scanf("%d %d", &n1, &n2);

for(i=1; i <= n1 && i <= n2; ++i)

{

if(n1%i==0 && n2%i==0)

gcd = i;

}

printf("G.C.D of %d and %d is %d", n1, n2, gcd);

return 0;

}

**OUTPUT:**

Enter two integers: 46

12

G.C.D of 46 and 12 is 2

**ANALYSIS:**

* Best Case: When one of the numbers is zero, the algorithm immediately returns the other number as the GCD. This takes constant time, O(1).
* Worst Case: The algorithm takes O(logn) time in the worst case. For example, when the two numbers are consecutive Fibonacci numbers, the algorithm has to perform multiple iterations.
* Average Case: Similar to the worst case, the average case time complexity is O(logn) since the number of steps required to reduce the numbers generally depends on their magnitude.

**LAB:02**

**Implement Fibonacci series and perform its analysis**

**THEORY:**

The Fibonacci numbers, commonly denoted by F(n) from sequence, called Fibonacci sequence, such that each number is the sum of two preceding ones, starting from 0 and 1. The Fibonacci sequence are  0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...so on.

Fibonacci numbers are generated b setting F(0) = 0 and F(1) = 1 and then using the recursive formula

F(n) = F(n-1) + F(n-2)

**ALGORITHM:**

1. Start
2. Set first=0, second=1
3. Read term of Fibonacci number say it be n
4. Set i=3
5. While(i<=n)

* Set temp = first + second
* Set first = second
* Set second = temp
* Increment i by 1 as , i++

1. Print temp as required Fibonacci number
2. Stop

**TIME COMPLEXITY:**

* Best case: O(1)
* Worst case: O(n)
* Average case: O(n)

**SPACE COMPLEXITY:**

* Space complexity: O(1)
* Auxiliary space complexity: O(1)

**SOURCE CODE:**

#include <stdio.h>

void printFibonacci(int n) {

    int t1 = 0, t2 = 1, nextTerm;

    printf("Fibonacci Series: %d, %d", t1, t2);

    for (int i = 3; i <= n; ++i) {

        nextTerm = t1 + t2;

        printf(", %d", nextTerm);

        t1 = t2;

        t2 = nextTerm;

    }

    printf("\n");

}

int main() {

    int n;

    printf("Enter the number of terms: ");

    scanf("%d", &n);

    if (n < 1) {

        printf("Please enter a positive integer.\n");

    } else {

        printFibonacci(n);

    }

    return 0;

}

**OUTPUT:**

Enter the number of terms: 7

Fibonacci Series: 0, 1, 1, 2, 3, 5, 8

**ANALYSIS:**

* Best Case: The best case occurs when n is 1 or 2, which results in O(1) time complexity.
* Worst Case: The worst-case time complexity is O(n), where n is the number of terms to be printed.
* Average Case: The average case time complexity is also O(n) since the algorithm must iterate through all the terms to generate the series.

**LAB:03**

**Implement Bubble Sort algorithm and perform its analysis**

**THEORY:**

Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in wrong order. Although the algorithm is simple, it is too slow and impractical for most problems even when compared to insertion sort. It can be practical if the input is usually in sort order but may occasionally have some out-of-order elements nearly in position. The complexity of bubble sort is O(n^2) in the worst and average case because the entire array needs to be iterated for every element. However, the complexity is O(n) in the best case because the array needs to be iterated only once when the array is already sorted. Bubble sort is adaptive.

**ALGORITHM:**

1. Start.
2. Read the key element from the user.
3. Repeat the following steps for (length of list - 1) times:
4. Iterate through the list from the beginning to the end.
5. For each pair of adjacent elements:
   1. If the first element is greater than the second element:
      1. Swap them.
6. If no swaps were made during a pass through the list, the list is sorted and you can exit early.
7. End.

**TIME COMPLEXITY:**

* Best case: O(n)
* Worst case: O(n2)
* Average case: O(n2)

**SPACE COMPLEXITY:**

* Space complexity: O(1)
* Auxiliary space complexity: O(1)

**SOURCE CODE:**

#include<stdio.h>

int main(){

int n, i,j, arr[50], temp;

printf("Enter total no. of elements:");

scanf("%d",&n);

printf("Enter %d numbers:",n);

for(i=0;i<n;i++){

scanf("%d",&arr[i]);

}

for(i=0;i<(n-1);i++){

for(j=0;j<(n-i-1);j++){

if(arr[j]>arr[j+1]){

temp=arr[j];

arr[j]=arr[j+1];

arr[j+1]=temp;

}

}

}

printf("Sorted list in ascending order: \n");

for(i=0;i<n;i++){

printf("%d ",arr[i]);

}

return 0;

}

**OUTPUT:**

Enter total no. of elements:5

Enter 5 numbers: 6 9 3 10 15

Sorted list in ascending order:

3 6 9 10 15

**ANALYSIS:**

* Best Case: The best case occurs when the array is already sorted, resulting in O(n) time complexity.
* Worst Case: The worst-case time complexity is O(n2) occurring when the array is sorted in reverse order.
* Average Case: The average case time complexity is O(n2) as the algorithm has to compare and possibly swap elements multiple times.

**LAB:04**

**Implement Selection Sort algorithm and perform its analysis**

**THEORY:**

Selection sort is a sorting algorithm, specifically an in-place comparison sort. It has O(n^2) time complexity, making it inefficient on large lists, and generally performs worse than the similar insertion sort. Selection sort is noted for its simplicity and has performance advantages over more complicated algorithms in certain situations, particularly where auxiliary memory is limited.

**ALGORITHM:**

1. Start.
2. Read the key element from the user.
3. Repeat the following steps for each element in the list (except the last one):
4. Assume the first unsorted element is the smallest.
5. For each of the remaining unsorted elements:
   1. If an element is smaller than the assumed smallest element, update the smallest element.
6. Swap the smallest element found with the first unsorted element.
7. End.

**TIME COMPLEXITY:**

* Best case: O(n2)
* Worst case: O(n2)
* Average case: O(n2)

**SPACE COMPLEXITY:**

* Space complexity: O(1)
* Auxiliary space complexity: O(1

**SOURCE CODE:**

#include <stdio.h>

void selectionSort(int arr[], int size);

int main()

{

int n;

printf("Enter the number of elements in the array: ");

scanf("%d", &n);

int arr[n];

printf("Enter the elements of the array: ");

for (int i = 0; i < n; i++)

{

scanf("%d", &arr[i]);

}

selectionSort(arr, n);

printf("The sorted array is: ");

for (int i = 0; i < n; i++)

printf("%d ", arr[i]);

return 0;

}

void selectionSort(int arr[], int size)

{

int minIndex, temp;

for (int i = 0; i < size - 1; i++)

{

minIndex = i;

for (int j = i + 1; j < size; j++)

{

if (arr[j] < arr[minIndex])

minIndex = j;

}

temp = arr[minIndex];

arr[minIndex] = arr[i];

arr[i] = temp;

}

}

**OUTPUT:**

Enter the number of elements in the array: 5

Enter the elements of the array: 64 25 12 22 11

The sorted array is: 11 12 22 25 64

**ANALYSIS:**

* Best Case: The best-case time complexity is as the algorithm always performs (n−1)/2 comparisons.
* Worst Case: The worst-case time complexity is O(n2), similar to the best case.
* Average Case: The average case time complexity is also O(n2) since the number of comparisons does not depend on the order of the elements.

**LAB:05**

**Implement Binary search algorithm and perform its analysis**

**THEORY:**

Binary search is a highly efficient algorithm for finding an item from a sorted list of items. It works by repeatedly dividing the search interval in half. If the value of the search key is less than the item in the middle of the interval, the search continues in the lower half, otherwise, it continues in the upper half. This process is repeated until the search key is found or the interval is empty.

**ALGORITHM:**

1. Start
2. Divide the search space into two halves by [finding the middle index “mid”](https://www.geeksforgeeks.org/problem-binary-search-implementations/).
3. Compare the middle element of the search space with the key.
4. If the key is found at middle element, the process is terminated.
5. If the key is not found at middle element, choose which half will be used as the next search space.
   * 1. If the key is smaller than the middle element, then the left side is used for next search.
     2. If the key is larger than the middle element, then the right side is used for next search.
6. This process is continued until the key is found or the total search space is exhausted.
7. Stop

**TIME COMPLEXITY:**

* Best case: O(1)
* Worst case: O(logn)
* Average case: O(logn)

**SPACE COMPLEXITY:**

* Space complexity: O(1)
* Auxiliary space complexity: O(1) for iterative, O(logn) for recursive

**SOURCE CODE:**

#include <stdio.h>

int binarySearch(int arr[], int n, int key) {

int low = 0;

int high = n - 1;

while (low <= high) {

int mid = (low + high) / 2;

if (arr[mid] == key) {

return mid;

} else if (arr[mid] < key) {

low = mid + 1;

} else {

high = mid - 1;

}

}

return -1;

}

int main() {

int arr[] = {2, 3, 4, 10, 40, 50};

int n = sizeof(arr) / sizeof(arr[0]);

int key = 10;

int result = binarySearch(arr, n, key);

if (result != -1) {

printf("%d found at index %d.\n", key, result);

} else {

printf("%d not found.\n", key);

}

return 0;

}

**OUTPUT:**

10 found at index 3.

**ANALYSIS:**

* Best Case: The best-case time complexity is O(1) when the middle element is the target.
* Worst Case: The worst-case time complexity is O(logn), as each step reduces the search interval by half.
* Average Case: The average-case time complexity is O(logn) due to the halving process.

**LAB:06**

**Implement Quick Sort and perform its analysis**

**THEORY:**

Quick Sort is a divide-and-conquer algorithm that selects a pivot element and partitions the array around the pivot such that elements less than the pivot are on its left and elements greater than the pivot are on its right. The process is repeated for the sub-arrays formed by the partition.

**ALGORITHM:**

1. Start.
2. Choose a pivot element from the array.
3. Partition the array into two sub-arrays: one with elements less than the pivot and one with elements greater than the pivot.
4. Recursively apply quicksort to both sub-arrays.
5. Combine the sorted sub-arrays.
6. End.

**TIME COMPLEXITY:**

* Best case: O(nlogn)
* Worst case: O(n2)
* Average case: O(nlogn)

**SPACE COMPLEXITY:**

* Space complexity: O(logn)
* Auxiliary space complexity: O(logn)

**SOURCE CODE:**

#include <stdio.h>

void swap(int arr[], int i, int j) {

int t = arr[i];

arr[i] = arr[j];

arr[j] = t;

}

int partition(int arr[], int low, int high) {

int pivot = arr[high];

int i = (low - 1);

for (int j = low; j < high; j++) {

if (arr[j] < pivot) {

i++;

swap(arr, i, j);

}

}

swap(arr, i + 1, high);

return (i + 1);

}

void quickSort(int arr[], int low, int high) {

if (low < high) {

int pi = partition(arr, low, high);

quickSort(arr, low, pi - 1);

quickSort(arr, pi + 1, high);

}

}

int main() {

int n, i;

printf("Enter total number of elements: ");

scanf("%d", &n);

int arr[n];

printf("Enter %d numbers: ", n);

for (i = 0; i < n; i++) {

scanf("%d", &arr[i]);

}

quickSort(arr, 0, n - 1);

printf("Sorted list in ascending order: \n");

for (i = 0; i < n; i++) {

printf("%d ", arr[i]);

}

return 0;

}

**OUTPUT:**

Enter total number of elements: 5

Enter 5 numbers: 64 34 25 12 22

Sorted list in ascending order:

12 22 25 34 64

**ANALYSIS:**

* Best Case: The best case occurs when the pivot element divides the array into two equal halves, resulting in O(nlogn) time complexity.
* Worst Case: The worst-case time complexity is O(n2), occurring when the pivot element is always the smallest or largest element.
* Average Case: The average case time complexity is O(nlogn) as the pivot element usually divides the array into reasonably balanced parts.

**LAB:07**

**Implement Fractional Knapsack and perform its analysis**

**THEORY:**

The Fractional Knapsack problem is a variant of the classic Knapsack problem, where the goal is to maximize the total value of items that can be placed in a knapsack of fixed capacity. Unlike the 0/1 Knapsack problem, where items must either be taken in whole or left out entirely, the Fractional Knapsack problem allows for items to be broken into smaller parts. This means that you can take any fraction of an item, rather than having to take the whole item.

**ALGORITHM:**

1. Start.
2. Input the number of items n, the values v[i], and weights w[i] of each item, and the capacity W of the knapsack.
3. Calculate the value-to-weight ratio for each item: r[i]=v[i]w[i].
4. Sort the items based on the value-to-weight ratio in descending order.
5. Initialize the total value of items in the knapsack V to 0 and the current weight Wk to 0.
6. For each item i in the sorted list:
7. If the item can fit entirely in the knapsack (Wk+w[i]≤W):
   1. Add the full value of the item to V.
   2. Update the current weight Wk​.
8. Else:
   1. Calculate the fraction of the item that can fit.
   2. Add the corresponding fraction of the value to V.
   3. Break the loop as the knapsack is now full.
9. Output the total value V.
10. Stop.

**TIME COMPLEXITY:**

* Time complexity: O(nlogn)
* Space complexity: O(1)

**SOURCE CODE:**

#include <stdio.h>

void sortItems(int values[], int weights[], double ratios[], int n) {

for (int i = 0; i < n; i++) {

ratios[i] = (double)values[i] / weights[i];

}

for (int i = 0; i < n - 1; i++) {

for (int j = 0; j < n - i - 1; j++) {

if (ratios[j] < ratios[j + 1]) {

double tempRatio = ratios[j];

ratios[j] = ratios[j + 1];

ratios[j + 1] = tempRatio;

int tempValue = values[j];

values[j] = values[j + 1];

values[j + 1] = tempValue;

int tempWeight = weights[j];

weights[j] = weights[j + 1];

weights[j + 1] = tempWeight;

}

}

}

}

double fractionalKnapsack(int W, int values[], int weights[], int n) {

double ratios[n];

sortItems(values, weights, ratios, n);

int curWeight = 0;

double finalValue = 0.0;

for (int i = 0; i < n; i++) {

if (curWeight + weights[i] <= W) {

curWeight += weights[i];

finalValue += values[i];

} else {

int remain = W - curWeight;

finalValue += values[i] \* ((double)remain / weights[i]);

break;

}

}

return finalValue;

}

int main() {

int W = 50; // Capacity of the knapsack

int values[] = {60, 100, 120};

int weights[] = {10, 20, 30};

int n = sizeof(values) / sizeof(values[0]);

printf("Maximum value in Knapsack = %.2f\n", fractionalKnapsack(W, values, weights, n));

return 0;

}

**OUTPUT:**

Maximum value in Knapsack = 240.00

**ANALYSIS:**

* Best Case: When all items can fit into the knapsack, the complexity is still O(nlogn) due to the sorting step.
* Worst Case: When only a fraction of one item can fit into the knapsack, the complexity is O(nlogn).

**LAB:08**

**Implement 0/1 Knapsack and perform its analysis**

**THEORY:**

The 0/1 Knapsack problem is a classic optimization problem where the objective is to maximize the total value of items that can be included in a knapsack of a given capacity. Each item can either be included (1) or excluded (0).

Suppose, a theif has a bag or knapsack that can contain maximum weight W of his loot. There are n i.e. xi is 0 or 1. Here the objective is to collect the items that maximize the total profit earned.

Let W = Capacity of Knapsack

n = No. of items

W = {w1, w2, …wn} = weights of items

V = {v1, v2, …vn} = value of items

C[i,w] = maximum profit earned with item I and knapsack of capacity w

Then the recurrence relation for 0/1 knapsack problem is given as,

0 if i=0 or w=0

C[i,w] = C[i-1,w] if wi > w

Max{vi + C[i-1,w-wi], C[i-1,w] if i>0 and wi <= w

**ALGORITHM:**

1. Create a 2D array dp where dp[i][w] represents the maximum value that can be obtained with the first i items and capacity w.
2. Initialize the first row and column to 0.
3. For each item i and capacity w:
4. If the item's weight is less than or equal to w, calculate the maximum value by including or excluding the item.
5. Update dp[i][w] with the maximum value.
6. The value at dp[n][W] is the maximum value for n items and capacity W.

**TIME COMPLEXITY:**

* Time complexity: O(nW)
* Space complexity: O(nW)

**SOURCE CODE:**

#include <stdio.h>

int max(int a, int b) {

return (a > b) ? a : b;

}

void knapsack(int W, int wt[], int val[], int n) {

int dp[n+1][W+1];

for (int i = 0; i <= n; i++) {

for (int w = 0; w <= W; w++) {

if (i == 0 || w == 0)

dp[i][w] = 0;

else if (wt[i-1] <= w)

dp[i][w] = max(val[i-1] + dp[i-1][w-wt[i-1]], dp[i-1][w]);

else

dp[i][w] = dp[i-1][w];

}

}

printf("Maximum value: %d\n", dp[n][W]);

}

int main() {

int n, W;

printf("Enter number of items: ");

scanf("%d", &n);

int val[n], wt[n];

printf("Enter values of items: ");

for (int i = 0; i < n; i++) {

scanf("%d", &val[i]);

}

printf("Enter weights of items: ");

for (int i = 0; i < n; i++) {

scanf("%d", &wt[i]);

}

printf("Enter capacity of knapsack: ");

scanf("%d", &W);

knapsack(W, wt, val, n);

return 0;

}

**OUTPUT:**

Enter number of items: 3

Enter values of items: 60 100 120

Enter weights of items: 10 20 30

Enter capacity of knapsack: 50

Maximum value: 220

**ANALYSIS:**

* Best Case: When the item weights are much smaller than the knapsack capacity, or when the optimal solution is straightforward.
* Worst Case: For large values of n and W, where the time and space requirements grow significantly.

**LAB:09**

**Implement Vertex Cover and perform its analysis**

**THEORY:**

A vertex Cover of a graph G is a set of vertices such that each edge in G is incident to at least one of these vertices. The decision vertex-cover problem was proven NPC. Now, we want to solve the optimal version of the vertex cover problem, i.e., we want to find a minimum size vertex cover of a given graph.

The Vertex Cover problem is a classical NP-complete problem in graph theory. Given an undirected graph, the objective is to find the smallest set of vertices such that each edge in the graph is incident to at least one vertex in the set.

**ALGORITHM:**

1. Initialize an empty set C to store the vertex cover.
2. While there are edges in the graph:
3. Pick any edge (u,v).
4. Add both vertices u and v to C.
5. Remove all edges incident to either u or v from the graph.
6. The set C is a vertex cover.

**TIME COMPLEXITY:**

* Time complexity: O(V+E)
* Space complexity: O(V)

**SOURCE CODE:**

#include <stdio.h>

#include <stdbool.h>

#define MAX 100

void addEdge(int graph[][MAX], int u, int v) {

graph[u][v] = 1;

graph[v][u] = 1;

}

void removeEdge(int graph[][MAX], int u, int v) {

graph[u][v] = 0;

graph[v][u] = 0;

}

void vertexCover(int graph[][MAX], int V) {

bool visited[MAX] = { false };

int uEdges[MAX], vEdges[MAX];

int edgeCount = 0;

for (int u = 0; u < V; u++) {

for (int v = u + 1; v < V; v++) {

if (graph[u][v]) {

uEdges[edgeCount] = u;

vEdges[edgeCount] = v;

edgeCount++;

}

}

}

for (int i = 0; i < edgeCount; i++) {

int u = uEdges[i];

int v = vEdges[i];

if (!visited[u] && !visited[v]) {

visited[u] = true;

visited[v] = true;

}

}

printf("Vertex Cover: ");

for (int i = 0; i < V; i++) {

if (visited[i])

printf("%d ", i);

}

printf("\n");

}

int main() {

int V = 7;

int graph[MAX][MAX] = { 0 };

addEdge(graph, 0, 1);

addEdge(graph, 0, 2);

addEdge(graph, 1, 3);

addEdge(graph, 3, 4);

addEdge(graph, 4, 5);

addEdge(graph, 5, 6);

vertexCover(graph, V);

return 0;

}

**OUTPUT:**

Vertex Cover: 0 1 3 4 5 6

**ANALYSIS:**

* Best Case: For specific graph structures where vertices are already well-distributed for minimal covers.
* Worst Case: For general graphs, the greedy algorithm does not always find the smallest vertex cover but provides an approximate solution.