**Data Structures Applications Lab (21EECF201) [0-0-2]**

**Term-work Report**

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Term-work** | *01* | | | | |  |  | | | | |
| **Student Name** | Bhushan Sonar | | | | |  |  | | | | |
| **SRN** | 01fe21bec312 | | | | **Roll Number** | | 561 | | **Division** | E | |
| **Code of ethics:**  I hereby declare that I am bound by ethics and have not copied any text/program/figure without acknowledging the content creators. I abide to the rule that upon plagiarized content all my marks will be made to zero.  Digital signature of the student | | | | | | | | | | | |
| **Apply Programming Skills**  **(5 marks)** | | **Identify Constraints and Implement**  **(10 marks)** | | **Integrate Modules**  **(3 Marks)** | | **Debugging and Tool usage**  **(2 marks)** | | **Remarks** | | | **Total**  **(20 Marks)** |
|  | |  | |  | |  | |  | | |  |
| **Problem Statement** | | | | | | | | | | | |
| Explain the operation of each algorithm type, take into account two examples of programmes for each algorithm type, and express the time complexity of each programme.   1. Iterative, 2. Recursive, 3. Back tracking, 4. Divide and conquer, 5. Dynamic programming, 2. Greedy, 7. Branch and Bound, 8. Brute force, 9. Randomized | | | | | | | | | | | |
| **Type of algorithm** | **Example No** | | **Which data structures are used?** | | | | | **What is the time complexity? O(n)** | | | |
| Iterative | **1** | | Array | | | | | O(logn) | | | |
| **2** | | Array | | | | | O(1) | | | |
| Recursive | **1** | | Array | | | | | O(n) | | | |
| **2** | | Array | | | | | O(1) | | | |
| Back tracking | **1** | | Array | | | | | O(n!) | | | |
| **2** | | Strings | | | | | O(n!) | | | |
| Divide and conquer | **1** | | Array | | | | | O(nlogn) | | | |
| **2** | | Array | | | | | O(n^2) | | | |
| Dynamic programming | **1** | | Array | | | | | O(n) | | | |
| **2** | | Array | | | | | O(n^2) | | | |
| Greedy | **1** | | Array | | | | | O(2^n) | | | |
| **2** | | Array | | | | | O(nlogn) | | | |
| Branch and bound | **1** | | Graph | | | | | O(n!) | | | |
| **2** | | Graph | | | | | O(n) | | | |
| Brute force | **1** | | Array | | | | | O(n) | | | |
| **2** | | Strings | | | | | O(n) | | | |
| Randomized | **1** | | Array | | | | | O(nlogn) | | | |
| **2** | | Array | | | | | O(n) | | | |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Code for example 1:** | | | | | | | |
| **Modularity** |  | **Documentation** |  | **Indentation** |  | **Programming practices** |  |
| **Type of Algorithm:**  *Iterative* | | | | | | | |
| **Details of the algorithm:** iterative algorithms are a type of algorithm that repeats a series of steps until a specific condition is met.  **Examples:**   1. **Binary Search:** 2. **Newtons method:** | | | | | | | |
| *#include <stdio.h>*  *int binarySearch(int arr[], int left, int right, int y) {*  *while (left <= right) {*  *int mid = left + (right - left) / 2;*  *if (arr[mid] == y)*  *return mid;*  *if (arr[mid] < y)*  *left = mid + 1;*  *else*  *right = mid - 1;*  *}*  *return -1;*  *}*  *int main() {*  *int arr[10];*  *int n = sizeof(arr) / sizeof(arr[0]);*  *printf("Enter %d elements for the array: ", n);*  *for (int i = 0; i < n; i++) {*  *scanf("%d", &arr[i]);*  *}*  *int x;*  *printf("Enter the value to search for: ");*  *scanf("%d", &y);*  *int result = binarySearch(arr, 0, n - 1, y);*  *if (result == -1)*  *printf("Element is not present in array.");*  *else*  *printf("Element is present at index %d.", result);*  *return 0;*  *}* | | | | | | | |
| **Sample Input:** | | | | | | | |
| *1 2 3 4 5 6 7 8 9 10* | | | | | | | |
| **Sample Output:** | | | | | | | |
| *Enter the element to be searched :5* | | | | | | | |
| **Time complexity calculation:O(logn)** | | | | | | | |
|  | | | | | | | |

|  |
| --- |
| **Code for example 2:** |
| *#include <stdio.h>*  *#include <math.h>*  *double sqrroot(double n) {*  *double z= n;*  *double root;*    *while (1) {*  *root = 0.5 \* (z+ (n / z));*    *if (fabs(root - z) < 0.000001) {*  *break;*  *}*    *z= root;*  *}*    *return root;*  *}*  *int main() {*  *double n;*    *printf("Enter a number to find its sqrroot: ");*  *scanf("%lf", &n);*    *double root = sqrroot(n);*    *printf("The square rootr”, n, root);*    *return 0;*  *}* |
| **Sample Input: 99** |
| *<Sample Input>* |
| **Sample Output:9.99** |
| *<Sample Output>* |
| **Time complexity calculation: O(1)** |
|  |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Code for example 1:** | | | | | | | |
| **Modularity** |  | **Documentation** |  | **Indentation** |  | **Programming practices** |  |
| **Type of Algorithm:**  *recursive* | | | | | | | |
| **Details of the algorithm:** Iterative algorithms are a type of algorithm that repeats a series of steps until a specific condition is met.  **Examples:**   1. **Factorial** 2. **Fabinocci method:** | | | | | | | |
| #include <stdio.h>  int fact(int n) {  if (n == 0) {  return 1;  } else {  return n \* fact(n-1);  }  }  int main() {  int n;    printf("Enter a number to find its factorial: ");  scanf("%d", &n);    int result = fact(n);    printf("%d! = %d", n, result);    return 0;  } | | | | | | | |
| **Sample Input: 5** | | | | | | | |
|  | | | | | | | |
| **Sample Output:120** | | | | | | | |
| *120* | | | | | | | |
| **Time complexity calculation: O(n)** | | | | | | | |
|  | | | | | | | |

|  |
| --- |
| **Code for example 2:** |
| *#include <stdio.h>*  *int fib(int n) {*  *if (n == 0 || n == 1) {*  *return n;*  *} else {*  *return fib(n-1) + fib(n-2);*  *}*  *}*  *int main()*  *{*  *int n;*    *printf("Enter the number of terms in the Fibonacci series: ");*  *scanf("%d", &n);*    *for (int i = 0; i < n; i++) {*  *printf("%d ", fib(i));*  *}*    *return 0;*  *}* |
|  |
|  |
| **Time complexity calculation: O(1)** |
|  |

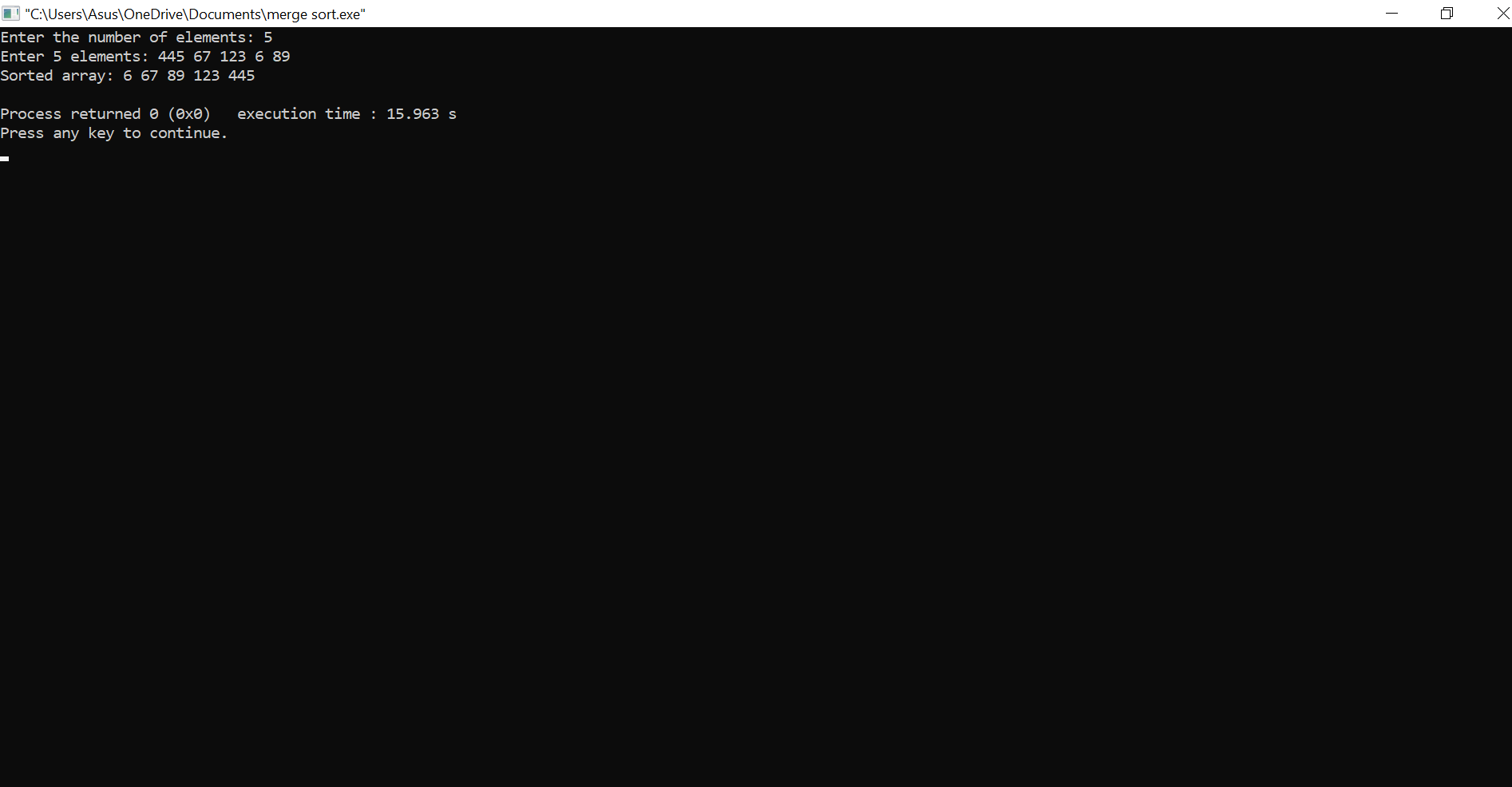
|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Code for example 1:** | | | | | | | |
| **Modularity** |  | **Documentation** |  | **Indentation** |  | **Programming practices** |  |
| **Type of Algorithm:**  *Back Tracking* | | | | | | | |
| **Details of the algorithm:** Backtracking is a technique used in algorithm design for solving problems recursively by trying out different possible solutions, and "backtracking" or undoing a solution that does not lead to the correct answer.  **Examples: 1.N Queen**  **2.permutation** | | | | | | | |
| *#include <stdio.h>*  *#include <stdbool.h>*  *#define N 8*  *void printSolution(int board[N][N]) {*  *for (int j= 0; j < N; j++) {*  *for (int j = 0; j < N; ++) {*  *printf("%d ", board[i][j]);*  *}*  *printf("\n");*  *}*  *}*  *bool isSafe(int board[N][N], int row, int col) {*  *int i, j;*    *for (i = 0; i < col; i++) {*  *if (board[row][i]) {*  *return false;*  *}*  *}*    *for (i = row, j = col; i >= 0 && j >= 0; i--, j--) {*  *if (board[i][j]) {*  *return false;*  *}*  *}*    *for (i = row, j = col; j >= 0 && i < N; i++, j--) {*  *if (board[i][j]) {*  *return false;*  *}*  *}*    *return true;*  *}*  *bool solveNQueens(int board[N][N], int col) {*  *if (col == N) {*  *printSolution(board);*  *return true;*  *}*    *bool res = false;*    *for (int i = 0; i < N; i++) {*  *if (isSafe(board, i, col)) {*  *board[i][col] = 1;*    *res = solveNueens(board, col+1) || res;*    *board[i][col] = 0;*  *}*  *}*    *return res;*  *}*  *int main() {*  *int board[N][N] = {0};*    *if (!solveNQueens(board, 0)) {*  *printf("Solution does not exist");*  *}*    *return 0;*  *}* | | | | | | | |
|  | | | | | | | |

**Time complexity calculation: O(n!)**

|  |
| --- |
| **Code for example 2:** |
|  |
|  |
| *#include <stdio.h>*  *#include <string.h>*  *void swap(char \*a, char \*b) {*  *char temp = \*x;*  *\*x = \*y;*  *\*y = temp;*  *}*  *void permute(char \*str, int left, int right) {*  *// Base case: when all characters are fixed*  *if (left == right) {*  *printf("%s\n", str);*  *return;*  *}*  *// Recursive case: try all possible permutations*  *for (int i = left; i <= right; i++) {*  *swap(&str[left], &str[i]);*  *permute(str, left + 1, right);*  *swap(&str[left], &str[i]);*  *}*  *}*  *int main() {*  *char str[] = "ABC";*  *int n = strlen(str);*  *permute(str, 0, n - 1);*  *return 0;*  *}* |
|  |

Time complexity: O(n!)

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Code for example 1:** | | | | | | | |
| **Modularity** |  | **Documentation** |  | **Indentation** |  | **Programming practices** |  |
| **Type of Algorithm:**  *Divide and Conquer* | | | | | | | |
| **Details of the algorithm:** Divide and conquer is a common algorithmic paradigm used to solve a wide variety of problems. The general idea behind divide and conquer is to break down a problem into smaller subproblems, solve them independently, and then combine the solutions to solve the original problem. This approach is typically more efficient than solving the problem directly.  **Example 1:Merge sort**  **2:Bubble Sort** | | | | | | | |
| #include <stdio.h>  #include <stdlib.h>  void merge(int arr[], int l, int m, int r) {  int i, j, k;  int n1 = m - l + 1;  int n2 = r - m;    // create temporary arrays  int L[n1], R[n2];    // copy data to temporary arrays  for (i = 0; i < n1; i++)  L[i] = arr[l + i];  for (j = 0; j < n2; j++)  R[j] = arr[m + 1 + j];    // merge the temporary arrays back into arr[l..r]  i = 0;  j = 0;  k = l;  while (i < n1 && j < n2) {  if (L[i] <= R[j]) {  a[k] = L[i];  i++;  }  else {  a[k] = R[j];  j++;  }  k++;  }    while (i < n1) {  a[k] = L[i];  i++;  k++;  }    while (j < n2) {  a[k] = R[j];  j++;  k++;  }  }    void mergeSort(int a[], int l, int r) {  if (l < r) {  // find the middle point to divide the array  int m = l + (r - l) / 2;    // recursively sort both halves  mergeSort(a, l, m);  mergeSort(a, m + 1, r);    // merge the sorted halves  merge(a, l, m, r);  }  }    int main() {  int n;  printf("Enter the number of elements: ");  scanf("%d", &n);  int a[n];  printf("Enter %d elements: ", n);  for (int i = 0; i < n; i++) {  scanf("%d", &a[i]);  }  mergeSort(arr, 0, n - 1);  printf("Sorted array: ");  for (int i = 0; i < n; i++) {  printf("%d ", a[i]);  }  printf("\n");  return 0;  } | | | | | | | |
| **Sample Input:** | | | | | | | |
| *1 2 5 82 6* | | | | | | | |
| **Sample Output:** | | | | | | | |
| *1 2 5 6 8*  **Time complexity calculation:O(nlogn)** | | | | | | | |



|  |
| --- |
| **Code for example 2:** |
| *#include <stdio.h>*  *#include <stdlib.h>*  *void bubbleSort(int arr[], int n) {*  *int i, j, temp;*  *for (i = 0; i < n - 1; i++) {*  *for (j = 0; j < n - i - 1; j++) {*  *if (arr[j] > arr[j + 1]) {*  *// swap arr[j] and arr[j+1]*  *temp = arr[j];*  *arr[j] = arr[j + 1];*  *arr[j + 1] = temp;*  *}*  *}*  *}*  *}*  *int main() {*  *int n, i;*  *printf("Enter the number of elements: ");*  *scanf("%d", &n);*  *int arr[n];*  *printf("Enter %d integers: ", n);*  *for (i = 0; i < n; i++) {*  *scanf("%d", &arr[i]);*  *}*  *bubbleSort(arr, n);*  *printf("Sorted array: ");*  *for (i = 0; i < n; i++) {*  *printf("%d ", arr[i]);*  *}*  *printf("\n");*  *return 0;*  *}* |
| **Sample Input:** |
|  |
| **Sample Output:**  **Time complexity calculation: O(n^2)** |
|  |

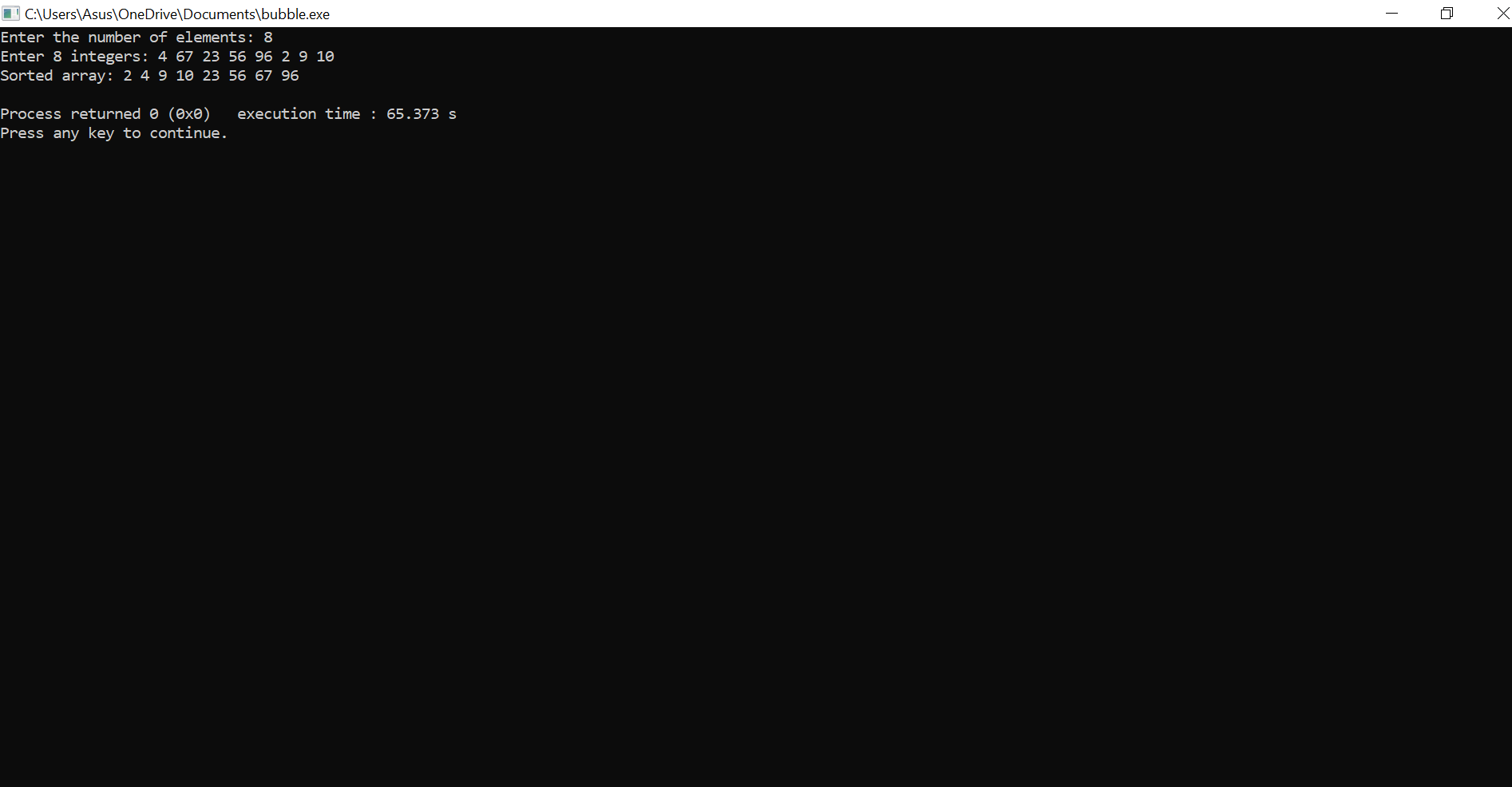
|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Code for example 1:** | | | | | | | |
| **Modularity** |  | **Documentation** |  | **Indentation** |  | **Programming practices** |  |
| **Type of Algorithm:**  *Dynamic Programming* | | | | | | | |
| **Details of the algorithm:** Dynamic programming is a powerful technique that can often lead to efficient solutions for complex optimization problems. However, it can also be challenging to apply correctly and efficiently, particularly for problems with a large number of subproblems or complex dependencies between them.  Examples: 1.knapsack  **2.length of common longest subset** | | | | | | | |
| #include <stdio.h>  #include <stdlib.h>  int main() {  int n, W;  printf("Enter the number of items: ");  scanf("%d", &n);  int wt[n], val[n];  for (int i = 0; i < n; i++) {  printf("Enter the weight and value of item %d: ", i + 1);  scanf("%d %d", &wt[i], &val[i]);  }  printf("Enter the capacity of the knapsack: ");  scanf("%d", &W);  int result = knapsack(W, wt, val, n);  printf("The maximum value that can be obtained is %d\n", result);  return 0;  }  int max(int a, int b) {  return (a > b) ? a : b;  }  int knapsack(int W, int wt[], int val[], int n) {  int K[n + 1][W + 1];  for (int i = 0; i <= n; i++) {  for (int w = 0; w <= W; w++) {  if (i == 0 || w == 0) {  K[i][w] = 0;  } else if (wt[i - 1] <= w) {  K[i][w] = max(val[i - 1] + K[i - 1][w - wt[i - 1]], K[i - 1][w]);  } else {  K[i][w] = K[i - 1][w];  }  }  }  return K[n][W];  } | | | | | | | |
| **Time complexity calculation: O(n)** | | | | | | | |

|  |
| --- |
| **Code for example 2:** |
| *#include <stdio.h>*  *#include <stdlib.h>*  *#include <string.h>*  *int main() {*  *char X[100], Y[100];*  *printf("Enter the first string: ");*  *scanf("%s", X);*  *printf("Enter the second string: ");*  *scanf("%s", Y);*  *int m = strlen(X);*  *int n = strlen(Y);*  *int result = lcs(X, Y, m, n);*  *printf("The length of the longest common subsequence is %d\n", result);*  *return 0;*  *}*  *int max(int a, int b) {*  *return (a > b) ? a : b;*  *}*  *int lcs(char\* X, char\* Y, int m, int n) {*  *int L[m + 1][n + 1];*  *for (int i = 0; i <= m; i++) {*  *for (int j = 0; j <= n; j++) {*  *if (i == 0 || j == 0) {*  *L[i][j] = 0;*  *} else if (X[i - 1] == Y[j - 1]) {*  *L[i][j] = L[i - 1][j - 1] + 1;*  *} else {*  *L[i][j] = max(L[i - 1][j], L[i][j - 1]);*  *}*  *}*  *}*  *return L[m][n];*  *}*    **Time complexity calculation:** |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Code for example 1:** | | | | | | | |
| **Modularity** |  | **Documentation** |  | **Indentation** |  | **Programming practices** |  |
| **Type of Algorithm:**  *Greedy Algorithm* | | | | | | | |
| **Details of the algorithm:** A greedy algorithm is an approach to problem-solving where the algorithm makes the locally optimal choice at each step in the hope of finding a globally optimal solution. In other words, the algorithm chooses the best available option at each step, without considering the long-term consequences or the possibility of making a better choice later on.  **Example 1.Coin change**  **2.Activity selection**   |  | | --- | | ***#include <stdio.h>***  ***int main() {***  ***int coins[] = {25, 10, 5, 1};***  ***int n = sizeof(coins) / sizeof(coins[0]);***  ***int amount = 48;***  ***printf("Coins used to make change for %d cents: ", amount);***  ***findCoins(coins, n, amount);***  ***return 0;***  ***}***  ***void findCoins(int coins[], int n, int amount) {***  ***int i, count = 0;***  ***for (i = n - 1; i >= 0; i--) {***  ***while (amount >= coins[i]) {***  ***printf("%d ", coins[i]);***  ***amount -= coins[i];***  ***count++;***  ***}***  ***}***  ***printf("\nMinimum number of coins required: %d\n", count);***  ***}*** | |  |   **Time complexity calculation: O(2^n)**   |  | | --- | | **Code for example 2:** |   **#include <stdio.h>**  **#include <stdlib.h>**  **typedef struct {**  **int start;**  **int end;**  **} Activity;**  **int compareActivities(const void\* a, const void\* b) {**  **return ((Activity\*)a)->end - ((Activity\*)b)->end;**  **}**  **void selectActivities(Activity activities[], int n) {**  **qsort(activities, n, sizeof(Activity), compareActivities);**  **int i, j;**  **i = 0;**  **printf("Selected activities: ");**  **printf("(%d,%d) ", activities[i].start, activities[i].end);**  **for (j = 1; j < n; j++) {**  **if (activities[j].start >= activities[i].end) {**  **printf("(%d,%d) ", activities[j].start, activities[j].end);**  **i = j;**  **}**  **}**  **}**  **int main() {**  **int n, i;**  **printf("Enter the number of activities: ");**  **scanf("%d", &n);**  **Activity activities[n];**  **for (i = 0; i < n; i++) {**  **printf("Enter start time and end time of activity %d: ", i + 1);**  **scanf("%d %d", &activities[i].start, &activities[i].end);**  **}**  **selectActivities(activities, n);**  **return 0;**  **}** | | | | | | | |
|  | | | | | | | |

**Time complexity calculation: O(nlogn)**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Code for example 1:** | | | | | | | |
| **Modularity** |  | **Documentation** |  | **Indentation** |  | **Programming practices** |  |
| **Type of Algorithm:**  *Branch and Bound* | | | | | | | |
| **Details of the algorithm:**  Branch and Bound is an algorithmic technique used to solve optimization problems. It works by exploring the solution space systematically, pruning branches that are unlikely to lead to optimal solutions. Branch and Bound is often used when exhaustive search is not feasible due to the size of the solution space.  **Example 1: Traveling salesman**  **2:Min Cost** | | | | | | | |
| *int main() {*  *int graph[N][N] = {*  *{ 0, 10, 15, 20 },*  *{ 10, 0, 35, 25 },*  *{ 15, 35, 0, 30 },*  *{ 20, 25, 30, 0 }*  *};*  *int path[N];*  *int cost = tsp(graph, 1, 0, 1, 0, path);*  *printf("Minimum cost: %d\nPath: 0 ", cost);*  *for (int i = 0; i < N - 1; i++) {*  *printf("%d ", path[i]);*  *}*  *printf("0\n");*  *return 0;*  *}*  **Time complexity calculation:O(n!)** | | | | | | | |



|  |
| --- |
| **Code for example 2:** |
| **#include <stdio.h>**  **#include <stdlib.h>**  **#include <limits.h>**  **#define N 4**  **int graph[N][N] = {**  **{ 0, 10, 15, 20 },**  **{ 10, 0, 35, 25 },**  **{ 15, 35, 0, 30 },**  **{ 20, 25, 30, 0 }**  **};**  **int min\_cost = INT\_MAX;**  **int visited[N] = {0};**  **void tsp(int curr\_city, int visited\_count, int cost\_so\_far) {**  **if (visited\_count == N) {**  **if (cost\_so\_far + graph[curr\_city][0] < min\_cost) {**  **min\_cost = cost\_so\_far + graph[curr\_city][0];**  **}**  **return;**  **}**    **for (int i = 0; i < N; i++) {**  **if (!visited[i]) {**  **visited[i] = 1;**  **tsp(i, visited\_count + 1, cost\_so\_far + graph[curr\_city][i]);**  **visited[i] = 0;**  **}**  **}**  **}**  **int main() {**  **visited[0] = 1;**  **tsp(0, 1, 0);**  **printf("Minimum cost: %d\n", min\_cost);**  **return 0;**  **}**  **Time complexity calculation: O(n)** |
|  |
| **Time complexity calculation: O(n)** |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Code for example 1:** | | | | | | | |
| **Modularity** |  | **Documentation** |  | **Indentation** |  | **Programming practices** |  |
| **Type of Algorithm:**  *Brute Force* | | | | | | | |
| **Details of the algorithm:** **:** Brute force is a straightforward algorithmic technique that involves systematically evaluating every possible solution to a problem until a satisfactory one is found. This approach is often used when the solution space is relatively small and the cost of evaluating each solution is relatively low  Example 1:Max element in aarray  **2:substring**   |  | | --- | | *#include <stdio.h>*  *int main() {*  *int arr[] = {10, 30, 20, 50, 40};*  *int n = sizeof(arr) / sizeof(arr[0]);*  *printf("The maximum element in the array is: %d", max(arr, n));*  *return 0;*  *}*  *int max(int arr[], int n) {*  *int i, max = arr[0];*  *for (i = 1; i < n; i++) {*  *if (arr[i] > max) {*  *max = arr[i];*  *}*  *}*  *return max;*  *}* | |  | | **Time complexity calculation:O(n)** |  |  | | --- | | **Code for example 2:** | | *#include <stdio.h>*  *#include <string.h>*  *int main() {*  *char str[] = "Hello, world!";*  *char substr[] = "world";*  *int index = findSubstring(str, substr);*  *if (index != -1) {*  *printf("The substring '%s' was found at index %d in the string '%s'", substr, index, str);*  *} else {*  *printf("The substring '%s' was not found in the string '%s'", substr, str);*  *}*  *return 0;*  *}*  *int findSubstring(char str[], char substr[]) {*  *int i, j, k, n, m;*  *n = strlen(str);*  *m = strlen(substr);*  *for (i = 0; i <= n - m; i++) {*  *j = 0;*  *while (j < m && str[i + j] == substr[j]) {*  *j++;*  *}*  *if (j == m) {*  *return i;*  *}*  *}*  *return -1;*  *}* | | | | | | | | |
|  | | | | | | | |

**Time complexity calculation** : O(n)

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Code for example 1:** | | | | | | | |
| **Modularity** |  | **Documentation** |  | **Indentation** |  | **Programming practices** |  |
| **Type of Algorithm:**  **Randomised** | | | | | | | |
| **Details of the algorithm:** Randomized algorithms are algorithms that make use of random numbers to solve problems. These algorithms often provide a faster solution than deterministic algorithms, and they are useful for solving problems in a probabilistic way.  Example:1.Randomised Quick Sort  **2:Randomised max element in array**   |  | | --- | | *#include <stdlib.h>*  *#include <time.h>*  *int main() {*  *int n, i;*  *printf("Enter the number of elements in the array: ");*  *scanf("%d", &n);*  *int arr[n];*  *printf("Enter the elements of the array:\n");*  *for (i = 0; i < n; i++) {*  *scanf("%d", &arr[i]);*  *}*  *randomizedQuickSort(arr, 0, n - 1);*  *printf("The sorted array is:\n");*  *for (i = 0; i < n; i++) {*  *printf("%d ", arr[i]);*  *}*  *printf("\n");*  *return 0;*  *}*  *void swap(int\* a, int\* b) {*  *int temp = \*a;*  *\*a = \*b;*  *\*b = temp;*  *}*  *int partition(int arr[], int low, int high) {*  *int pivot = arr[high];*  *int i = (low - 1);*  *for (int j = low; j <= high - 1; j++) {*  *if (arr[j] < pivot) {*  *i++;*  *swap(&arr[i], &arr[j]);*  *}*  *}*  *swap(&arr[i + 1], &arr[high]);*  *return (i + 1);*  *}*  *int randomizedPartition(int arr[], int low, int high) {*  *srand(time(NULL));*  *int random = low + rand() % (high - low);*  *swap(&arr[random], &arr[high]);*  *return partition(arr, low, high);*  *}*  *void randomizedQuickSort(int arr[], int low, int high) {*  *if (low < high) {*  *int pi = randomizedPartition(arr, low, high);*  *randomizedQuickSort(arr, low, pi - 1);*  *randomizedQuickSort(arr, pi + 1, high);*  *}*  *}* | |  | | **Time complexity calculation: O(nlogn)** |  |  | | --- | | **Code for example 2:** | | *#include <stdio.h>*  *#include <stdlib.h>*  *#include <time.h>*  *int main() {*  *int arr[] = {3,7,12,89};*  *int n = sizeof(arr) / sizeof(arr[0]);*  *srand(time(NULL));*  *int iterations = 100;*  *int max = arr[rand() % n];*  *for (int i = 1; i < iterations; i++) {*  *int random = arr[rand() % n];*  *if (random > max) {*  *max = random;*  *}*  *}*  *printf("Maximum element: %d\n", max);*  *return 0;*  *}*  *int random\_max(int arr[], int n) {*  *int max = arr[0];*  *for (int i = 1; i < n; i++) {*  *if (arr[i] > max) {*  *max = arr[i];*  *}*  *}*  *return max;*  *}*  *Sample input:3,7,12,89*    **Time complexity calculation: O(n)** | | | | | | | | |