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

**Term-work Report**

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Term-work** | *01* | | | | |  |  | | | | |
| **Student Name** | BHARATH Y R | | | | |  |  | | | | |
| **SRN** | 01FE21BEC123 | | | | **Roll Number** | | 317 | | **Division** | C | |
| **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** | | Arrays | | | | | O(n) | | | |
| **2** | | Linked list | | | | | O(n) | | | |
| Recursive | **1** | | Arrays | | | | | O(n) | | | |
| **2** | | Arrays | | | | | O(logn) | | | |
| Back tracking | **1** | | Arrays | | | | | O(n!) | | | |
| **2** | | Arrays | | | | | O(n^2) | | | |
| Divide and conquer | **1** | | Arrays | | | | | O(nlogn) | | | |
| **2** | | Arrays | | | | | O(nlogn) | | | |
| Dynamic programming | **1** | | Arrays | | | | | O(n) | | | |
| **2** | | Arrays | | | | | O(nw) | | | |
| Greedy | **1** | | Arrays | | | | | O(n) | | | |
| **2** | | Arrays | | | | | O(nlogn) | | | |
| Branch and bound | **1** | | Arrays | | | | | O(n\*n!) | | | |
| **2** | | Arrays | | | | | O(n^2) | | | |
| Brute force | **1** | | Linklist | | | | | O(n) | | | |
| **2** | | Arrays | | | | | O(n) | | | |
| Randomized | **1** | | Arrays | | | | | O(n) | | | |
| **2** | | Arrays | | | | | O(n) | | | |

Were you able to solve this problem? If not what where the challenges?

Most of them were solvable except few problems which needed a bit more time to intrigue them.

What assistance do you need to learn this term work better?

*Some of the websites and youtube videos to understand the concept.*

What are the areas you think you should work on to be able to make this solution better?

I think I need to work more on different date structures to make solution better.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Modularity** |  | **Documentation** |  | **Indentation** |  | **Programming practices** |  |
| **Type of Algorithm: Iterative** | | | | | | | |
| **Details of the algorithm:** | | | | | | | |
| *An iterative algorithm is an algorithm that repeats a set of instructions until a specific condition is met, without using recursion. In data structures, iterative algorithms are commonly used to traverse and manipulate data in a systematic manner.*  *Application:*   1. *Sorting* 2. *Searching* 3. *Data processing* 4. *Traversing*   *5. Modifying* | | | | | | | |
| **Code for example 1:** | | | | | | | |
| #include<stdio.h>  int main()  {  int n,i;  printf("enter the value of n");  scanf("%d",&n);  int arr[n];  printf("enter the %d values:\n",n); //reading of array elements  for(i=0;i<n;i++)  {  scanf("%d",&arr[i]);  }  printf("entered values are:"); //printing of array elements  for(i=0;i<n;i++)  {  printf("%d\n",arr[i]);  }  printf("\n");  return 0;  } | | | | | | | |
| **Sample Input:** | | | | | | | |
| Value of n :8  Entered numbers are : 10 20 30 40 50 60 70 80 | | | | | | | |
| **Sample Output:** | | | | | | | |
| Printed numbers are : 10 20 30 40 50 60 70 80 | | | | | | | |
| **Time complexity calculation:** | | | | | | | |
| O(n) | | | | | | | |

|  |
| --- |
| **Code for example 2:** |
| #include<stdio.h>  #include<stdlib.h>  struct Node  {  int data;  struct Node \*next;  };  void display (struct Node \*node)  {  //as linked list will end when Node is Null  while (node != NULL)  {  printf ("%d ", node->data);  node = node->next;  }  printf ("\n");  }  int searchElement (struct Node \*head, int item)  {  struct Node \*current = head; // Initialize current  int index = 0;  // traverse till then end of the linked list  while (current != NULL)  {  if (current->data == item)  {  return index;  }  current = current->next;  index++;  }  return -1;  }  int main ()  {  int item;  //creating 4 pointers of type struct Node  //So these can point to address of struct type variable  struct Node \*head = NULL;  struct Node \*node2 = NULL;  struct Node \*node3 = NULL;  struct Node \*node4 = NULL;  // allocate 3 nodes in the heap  head = (struct Node \*) malloc (sizeof (struct Node));  node2 = (struct Node \*) malloc (sizeof (struct Node));  node3 = (struct Node \*) malloc (sizeof (struct Node));  node4 = (struct Node \*) malloc (sizeof (struct Node));  head->data = 10; // data set for head node  head->next = node2; // next pointer assigned to address of node2  node2->data = 15;  node2->next = node3;  node3->data = 20;  node3->next = node4;  node4->data = 25;  node4->next = NULL;  printf ("Linked List: ");  display (head);  printf ("Enter element to search: ");  scanf ("%d", &item);  int index = searchElement (head, item);  if (index == -1)  printf ("Item not found");  else  printf ("Item found at position: %d", index + 1);  return 0;  } |
| **Sample Input:** |
| Linked List: 10 15 20 25  Element to search :20 |
| **Sample Output:** |
| Element found at position 3 |
| **Time complexity calculation:** |
| O(n) |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Modularity** |  | **Documentation** |  | **Indentation** |  | **Programming practices** |  |
| **Type of Algorithm: Recursive** | | | | | | | |
| **Details of the algorithm:** | | | | | | | |
| *Recursion in Data Structures and Algorithms is a process in which a function calls itself repeatedly until a certain condition is met. It is a programming technique that enables the problem to be solved by breaking it down into smaller sub-problems that are easier to solve.*  *Application:*   1. *Tree Traversal* 2. *Graph Traversal*   *3.Dynamic Programming* | | | | | | | |
| **Code for example 1:** | | | | | | | |
| #include <stdio.h>  #define MAX\_SIZE 100  int main()  {  int arr[MAX\_SIZE];  int i, min, size;  int fact=1;  printf("Enter size of the array: ");  scanf("%d", &size);  printf("Enter elements in the array: ");  for(i=0; i<size; i++)  {  scanf("%d", &arr[i]);  }  min = arr[0];  for(i=1; i<size; i++)  {  if(arr[i] < min)  {  min = arr[i];  }  }  printf("Minimum element = %d", min);  printf("\n");  if (min < 0)  printf("Error! Factorial of a negative number doesn't exist.");  else {  for (i = 1; i <= min; i++)  {  fact =fact\* i;  }  printf("Factorial of %d = %d", min, fact);  }  return 0;  } | | | | | | | |
| **Sample Input:** | | | | | | | |
| Number of inputs : 5  Array elements are : 10 20 30 40 50 | | | | | | | |
| **Sample Output:** | | | | | | | |
| Min element is : 10  Factorial of 10 is : 3628800 | | | | | | | |
| **Time complexity calculation:** | | | | | | | |
| O(n) | | | | | | | |

|  |
| --- |
| **Code for example 2:** |
| #include <stdio.h>  int recursiveBinarySearch(int array[], int start\_index, int end\_index, int element)  {  if (end\_index >= start\_index){  int middle = start\_index + (end\_index - start\_index )/2;  if (array[middle] == element)  {  return middle;  }  if (array[middle] > element)  return recursiveBinarySearch(array, start\_index, middle-1, element);  return recursiveBinarySearch(array, middle+1, end\_index, element);  }  return -1;  }  int main(void){  int array[] = {1, 4, 7, 9, 16, 56, 70};  int n = 7;  int element = 9;  int found\_index = recursiveBinarySearch(array, 0, n-1, element);  if(found\_index == -1 ) {  printf("Element not found in the array ");  }  else {  printf("Element found at index : %d",found\_index);  }  return 0;  } |
| **Sample Input:** |
| Element to search = 7 |
| **Sample Output:** |
| Element found at index : 3 |
| **Time complexity calculation:** |
| O( logn) |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Modularity** |  | **Documentation** |  | **Indentation** |  | **Programming practices** |  |
| **Type of Algorithm: Back tracking** | | | | | | | |
| **Details of the algorithm:** | | | | | | | |
| *Backtracking is a technique used in computer science and programming to solve problems by finding all possible solutions, testing them, and choosing the best one. It is especially useful when there are many possible solutions, but not all of them are valid. Backtracking involves moving backwards through the problem space, eliminating impossible solutions as they are encountered and continuing until a solution is found.*  *Application:*   1. *Subset sum problem* 2. *N-queens problem* 3. *Sudoku* 4. *Knapsack problem* | | | | | | | |
| **Code for example 1:** | | | | | | | |
| #include <stdio.h>  void swap(int \*a, int \*b) {  int temp = \*a;  \*a = \*b;  \*b = temp;  }  void permute(int arr[], int start, int end)  { int i ;  if (start == end)  {  for (i = 0; i <= end; i++)  {  printf("%d ", arr[i]);  }  printf("\n");  }  else {  for ( i = start; i <= end; i++) {  swap(&arr[start], &arr[i]);  permute(arr, start+1, end);  swap(&arr[start], &arr[i]); // backtrack  }  }  }  int main() {  int arr[] = {1, 2, 3};  int n = sizeof(arr)/sizeof(arr[0]);  permute(arr, 0, n-1);  return 0;  } | | | | | | | |
| **Sample Input:** | | | | | | | |
| Input numbers : 1 2 3 | | | | | | | |
| **Sample Output:** | | | | | | | |
| 1 2 3  1 3 2  2 1 3  2 3 1  3 2 1  3 1 2 | | | | | | | |
| **Time complexity calculation:** | | | | | | | |
| O(n!) | | | | | | | |

|  |
| --- |
| **Code for example 2:** |
| #include <stdio.h>  int subset\_sum(int set[], int n, int sum) {  if (sum == 0) {  return 1;  }  else if (n == 0 && sum != 0) {  return 0;  }  else {  int include = subset\_sum(set, n-1, sum-set[n-1]);  int exclude = subset\_sum(set, n-1, sum);  return include || exclude;  }  }  int main() {  int set[] = {3, 34, 4, 12, 5, 2};  int sum = 9;  int n = sizeof(set)/sizeof(set[0]);  if (subset\_sum(set, n, sum)) {  printf("Found a subset with given sum\n");  }  else {  printf("No subset with given sum\n");  }  return 0;  } |
| **Sample Input:** |
| Set[]= {3, 34, 4, 12, 5, 2}  sum = 9 |
| **Sample Output:** |
| Found a subset with given sum |
| **Time complexity calculation:** |
| O(n^2) |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Modularity** |  | **Documentation** |  | **Indentation** |  | **Programming practices** |  |
| **Type of Algorithm:** Divide and conquer | | | | | | | |
| **Details of the algorithm:** | | | | | | | |
| *Divide and conquer is a popular algorithmic paradigm in computer science that involves breaking down a problem into subproblems of the same type and solving each subproblem separately. The solutions to the subproblems are then combined to produce the final solution to the original problem*.  *Application:*   1. *Sorting* 2. *Binary Search* 3. *Maximum Subarray* 4. *Closest Pair of Points* | | | | | | | |
| **Code for example 1:** | | | | | | | |
| #include<stdio.h>  void quicksort(int number[25],int first,int last){  int i, j, pivot, temp;  if(first<last){  pivot=first;  i=first;  j=last;  while(i<j)  {  while(number[i]<=number[pivot]&&i<last)  i++;  while(number[j]>number[pivot])  j--;  if(i<j)  {  temp=number[i];  number[i]=number[j];  number[j]=temp;  }  }  temp=number[pivot];  number[pivot]=number[j];  number[j]=temp;  quicksort(number,first,j-1);  quicksort(number,j+1,last);  }  }  int main(){  int i, count, number[25];  printf("number of elements to enter");  scanf("%d",&count);  printf("Enter %d elements: ", count);  for(i=0;i<count;i++)  scanf("%d",&number[i]);  quicksort(number,0,count-1);  printf("Order of Sorted elements: ");  for(i=0;i<count;i++)  printf(" %d",number[i]);  return 0;  } | | | | | | | |
| **Sample Input:** | | | | | | | |
| 54 74 54 53 85 86 97 32 12 23 | | | | | | | |
| **Sample Output:** | | | | | | | |
| 12 23 32 53 54 54 74 85 86 97 | | | | | | | |
| **Time complexity calculation:** | | | | | | | |
| O(nlogn) | | | | | | | |

|  |
| --- |
| **Code for example 2:** |
| #include <stdio.h>  #define max 10  int a[11] = { 10, 14, 19, 26, 27, 31, 33, 35, 42, 44, 0 };  int b[10];  void merging(int low, int mid, int high) {  int l1, l2, i;  for(l1 = low, l2 = mid + 1, i = low; l1 <= mid && l2 <= high; i++) {  if(a[l1] <= a[l2])  b[i] = a[l1++];  else  b[i] = a[l2++];  }  while(l1 <= mid)  b[i++] = a[l1++];  while(l2 <= high)  b[i++] = a[l2++];  for(i = low; i <= high; i++)  a[i] = b[i];  }  void sort(int low, int high) {  int mid;  if(low < high) {  mid = (low + high) / 2;  sort(low, mid);  sort(mid+1, high);  merging(low, mid, high);  } else {  return;  }  }  int main() {  int i;  printf("List before sorting\n");  for(i = 0; i <= max; i++)  printf("%d ", a[i]);  sort(0, max);  printf("\nList after sorting\n");  for(i = 0; i <= max; i++)  printf("%d ", a[i]);  } |
| **Sample Input:** |
| 10, 14, 19, 26, 27, 31, 33, 35, 42, 44, 0 |
| **Sample Output:** |
| 0 10 14 19 26 27 31 33 35 42 44 |
| **Time complexity calculation:** |
| O(nlogn) |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Modularity** |  | **Documentation** |  | **Indentation** |  | **Programming practices** |  |
| **Type of Algorithm: Dynamic programming** | | | | | | | |
| **Details of the algorithm:** | | | | | | | |
| *Dynamic programming is a technique used to solve optimization problems that involve finding the best solution among a large number of subproblems. It involves breaking down a problem into smaller subproblems, solving each subproblem only once, and storing the solution to each subproblem to avoid redundant computation. The technique is particularly useful when the subproblems overlap, as is often the case with optimization problems.*  *Application:*   1. *Shortest path algorithms* 2. *Longest common subsequence* 3. *Edit distance* 4. *Maximum subarray* | | | | | | | |
| **Code for example 1:** | | | | | | | |
| #include <stdio.h>  int main() {  int limit, i;  printf("Enter the limit of the Fibonacci series: ");  scanf("%d", &limit);  int fib[limit];  fib[0] = 0;  fib[1] = 1;  for (i = 2; i < limit; i++) {  fib[i] = fib[i-1] + fib[i-2];  }  printf("Fibonacci series up to %d:\n", limit);  for (i = 0; i < limit; i++) {  printf("%d ", fib[i]);  }  return 0;  } | | | | | | | |
| **Sample Input:** | | | | | | | |
| limit of the Fibonacci series: 10 | | | | | | | |
| **Sample Output:** | | | | | | | |
| 0 1 1 2 3 5 8 13 21 34 | | | | | | | |
| **Time complexity calculation:** | | | | | | | |
| O(n) | | | | | | | |

|  |
| --- |
| **Code for example 2:** |
| #include <stdio.h>  #include <stdlib.h>  int max(int a, int b) {  return (a > b) ? a : b;  }  int knapsack(int W, int wt[], int val[], int n) {  int i, w;  int K[n+1][W+1];  for (i = 0; i <= n; i++) {  for (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];  } |
| **Sample Input:** |
| Va [] = {60, 100, 120};  wt [] = {10, 20, 30};  W = 50; |
| **Sample Output:** |
| Maximum value that can be obtained = 220 |
| **Time complexity calculation:** |
| O(nW) |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Modularity** |  | **Documentation** |  | **Indentation** |  | **Programming practices** |  |
| **Type of Algorithm: Greedy** | | | | | | | |
| **Details of the algorithm:** | | | | | | | |
| *Greedy is a technique used in algorithms and data structures that aims to solve optimization problems by making locally optimal choices at each step. The idea is to make the best possible decision at each step based on the current available information, with the hope that this will lead to an overall optimal solution*  *Application:*   1. *Huffman coding* 2. *Kruskal's algorithm* 3. *Dijkstra's algorithm* 4. *Activity selection problem* | | | | | | | |
| **Code for example 1:** | | | | | | | |
| #include <stdio.h>  int minCoins(int coins[], int n, int value)  { int i;  int count = 0;  for ( i = 0; i < n; i++) {  while (value >= coins[i]) {  value -= coins[i];  count++;  }  }  return count;  }  int main() {  int coins[] = {1, 5, 10, 25};  int n = sizeof(coins)/sizeof(coins[0]);  int value = 78;  int numCoins = minCoins(coins, n, value);  printf("Minimum number of coins required: %d\n", numCoins);  return 0;  } | | | | | | | |
| **Sample Input:** | | | | | | | |
| coins = {1, 5, 10, 25}  n = 4  value = 78 | | | | | | | |
| **Sample Output:** | | | | | | | |
| Minimum number of coins required: 78 | | | | | | | |
| **Time complexity calculation:** | | | | | | | |
| O(n) | | | | | | | |

|  |
| --- |
| **Code for example 2:** |
| #include <stdio.h>  #include <stdlib.h>  struct Item {  int weight;  int value;  };  int cmpfunc(const void\* a, const void\* b) {  struct Item\* item1 = (struct Item\*)a;  struct Item\* item2 = (struct Item\*)b;  double ratio1 = (double)item1->value / item1->weight;  double ratio2 = (double)item2->value / item2->weight;  if (ratio1 > ratio2) {  return -1;  }  else if (ratio1 < ratio2) {  return 1;  }  else {  return 0;  }  }  int maxItems(struct Item items[], int n, int capacity) {  qsort(items, n, sizeof(struct Item), cmpfunc);  int numItems = 0;  int currentWeight = 0;  for (int i = 0; i < n; i++) {  if (currentWeight + items[i].weight <= capacity) {  numItems++;  currentWeight += items[i].weight;  }  }  return numItems;  }  int main() {  struct Item items[] = {{10, 60}, {20, 100}, {30, 120}};  int n = sizeof(items)/sizeof(items[0]);  int capacity = 50;  int maxNumItems = maxItems(items, n, capacity);  printf("Maximum number of items that can be taken is: %d\n", maxNumItems);  return 0;  } |
| **Sample Input:** |
| items = [{10, 60}, {20, 100}, {30, 120}]  capacity = 50 |
| **Sample Output:** |
| Maximum number of items that can be taken is: 2 |
| **Time complexity calculation:** |
| O(nlogn) |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Modularity** |  | **Documentation** |  | **Indentation** |  | **Programming practices** |  |
| **Type of Algorithm: Branch and bound** | | | | | | | |
| **Details of the algorithm:** | | | | | | | |
| *Branch and Bound is a general algorithmic technique that is used to solve optimization problems, such as finding the shortest path in a graph, the minimum cost spanning tree, or the traveling salesman problem. The basic idea behind Branch and Bound is to partition the problem space into smaller subspaces, called "branches", and solve each branch individually using a "bound" or an estimate of the optimal solution. The algorithm then "backtracks" to the previous branch if a better solution is found in a later branch, in order to continue exploring other branches of the problem space*  *Application:*   1. *Traveling Salesman Problem* 2. *Knapsack Problem* 3. *Graph Coloring Problem* 4. *Integer Linear Programming* | | | | | | | |
| **Code for example 1:** | | | | | | | |
| #include <stdio.h>  #include <string.h>  /\* Function to swap values at two pointers \*/  void swap(char\* x, char\* y)  {  char temp;  temp = \*x;  \*x = \*y;  \*y = temp;  }  void permute(char\* a, int l, int r)  {  int i;  if (l == r)  printf("%s\n", a);  else {  for (i = l; i <= r; i++) {  swap((a + l), (a + i));  permute(a, l + 1, r);  swap((a + l), (a + i)); // backtrack  }  }  }  int main()  {  char str[] = "ABC";  int n = strlen(str);  permute(str, 0, n - 1);  return 0;  } | | | | | | | |
| **Sample Input:** | | | | | | | |
| ABC | | | | | | | |
| **Sample Output:** | | | | | | | |
| ABC ACB BAC BCA CBA CAB | | | | | | | |
| **Time complexity calculation:** | | | | | | | |
| O(n\*n!) | | | | | | | |

|  |
| --- |
| **Code for example 2 :** |
| #include <stdio.h>  int maxSum = 0; // to store maximum sum  void maxSubsetSum(int arr[], int n, int level, int sum) {  if (level == n) {    if (sum > maxSum) // update the maximum sum if current sum is greater  {  maxSum = sum;  }  return;  }  // check if the current node can be included in the subset  if (sum + arr[level] > maxSum) {  maxSubsetSum(arr, n, level + 1, sum + arr[level]);  }  // check if the current node can be excluded from the subset  if (sum + arr[level + 1] <= maxSum) {  return;  }  maxSubsetSum(arr, n, level + 1, sum);  }  int main() {  int arr[] = {2, 5, 8, 10};  int n = sizeof(arr) / sizeof(arr[0]);  maxSubsetSum(arr, n, 0, 0);  printf("Maximum sum of a subset: %d\n", maxSum);  return 0;  } |
| **Sample Input:** |
| the input array is 2, 5, 8, 10 |
| **Sample Output:** |
| Maximum sum of a subset: 15 |
| **Time complexity calculation:** |
| O(2^n) |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Modularity** |  | **Documentation** |  | **Indentation** |  | **Programming practices** |  |
| **Type of Algorithm: Brute force** | | | | | | | |
| **Details of the algorithm:** | | | | | | | |
| *Brute force is a straightforward approach to solving a problem by trying all possible solutions and selecting the best one. In data structures, brute force algorithms involve checking all possible combinations or permutations of data elements to find a solution*  *Application:*   1. *Searching* 2. *Sorting* 3. *String matching:* 4. *Optimization problems* | | | | | | | |
| **Code for example 1:** | | | | | | | |
| #include <stdio.h>  #include <stdlib.h>  struct Node {  int data;  struct Node\* next;  };  int find\_max(struct Node\* head) {  int max = head->data;  struct Node\* current = head->next;  while (current != NULL) {  if (current->data > max) {  max = current->data;  }  current = current->next;  }  return max;  }  int main()  {  int n, i, num;  struct Node\* head = NULL;  struct Node\* current = NULL;  printf("Enter the size of the array: ");  scanf("%d", &n);  printf("Enter %d elements: ", n);  for (i = 0; i < n; i++) {  scanf("%d", &num);  if (head == NULL) {  head = (struct Node\*) malloc(sizeof(struct Node));  head->data = num;  head->next = NULL;  current = head;  } else {  current->next = (struct Node\*) malloc(sizeof(struct Node));  current->next->data = num;  current->next->next = NULL;  current = current->next;  }  }  printf("Maximum element is %d\n", find\_max(head));  return 0;  } | | | | | | | |
| **Sample Input:** | | | | | | | |
| size of the array: 8  elements in the array : 23 67 12 89 45 96 54 43 | | | | | | | |
| **Sample Output:** | | | | | | | |
| Maximum element is : 96 | | | | | | | |
| **Time complexity calculation:** | | | | | | | |
| O(n) | | | | | | | |

|  |
| --- |
| **Code for example 2 :** |
| #include <stdio.h>  int linear\_search(int arr[], int n, int key) {  int i;  for (i = 0; i < n; i++) {  if (arr[i] == key)  return i;  }  return -1;  }  int main()  { int i;  int arr[100], n, key, index;  printf("Enter the size of the array: ");  scanf("%d", &n);  printf("Enter %d elements: ", n);  for (i = 0; i < n; i++)  scanf("%d", &arr[i]);  printf("Enter the element to search: ");  scanf("%d", &key);  index = linear\_search(arr, n, key);  if (index == -1)  printf("%d not found\n", key);  else  printf("%d found\n",key);  } |
| **Sample Input:** |
| size of the array :7  elements of array : 56 74 86 34 97 35 32  key to search : 86 |
| **Sample Output:** |
| 86 found |
| **Time complexity calculation:** |
| O(n) |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Modularity** |  | **Documentation** |  | **Indentation** |  | **Programming practices** |  |
| **Type of Algorithm: Randomised** | | | | | | | |
| **Details of the algorithm:** | | | | | | | |
| *Randomized algorithms in data structures are algorithms that use random numbers to achieve a desired output. The basic idea behind randomized algorithms is to use a probabilistic approach to solve a problem that is otherwise difficult or impossible to solve deterministically. Randomized algorithms are used to solve problems that involve a large number of inputs, such as sorting, searching, and graph algorithms*  *Application:*   1. *Randomized sorting* 2. *Hash tables* 3. *Skip lists* 4. *Bloom filters* | | | | | | | |
| **Code for example 1:** | | | | | | | |
| #include <stdlib.h>  #include <time.h>  int main() {  int min = 100, max = 1000, n = 5;  int i, random\_number;  int arr[n];  srand(time(NULL)); // seed the random number generator  for (i = 0; i < n; i++) {  random\_number = rand() % (max - min + 1) + min;  arr[i] = random\_number;  }  printf("Random numbers between %d and %d:\n", min, max);  for (i = 0; i < n; i++)  {  printf("%d ", arr[i]);  }  printf("\n");  return 0;  } | | | | | | | |
| **Sample Input:** | | | | | | | |
| value of n : 8  value of min : 10  value of max : 100 | | | | | | | |
| **Sample Output:** | | | | | | | |
| 59 12 22 23 80 59 82 83 | | | | | | | |
| **Time complexity calculation:** | | | | | | | |
| O(n) | | | | | | | |

|  |
| --- |
| **Code for example 2:** |
| #include <stdio.h>  #include <stdlib.h>  #include <time.h>  void shuffle(int arr[], int n)  {  int i;  srand(time(NULL));  for ( i = n - 1; i > 0; i--) {  int j = rand() % (i + 1);  int temp = arr[i];  arr[i] = arr[j];  arr[j] = temp;  }  }  int main()  {  int i;  int arr[] = {1, 2, 3, 4, 5};  int n = sizeof(arr) / sizeof(arr[0]);  printf("Original array: ");  for ( i = 0; i < n; i++) {  printf("%d ", arr[i]);  }  printf("\n");  shuffle(arr, n);  printf("Shuffled array: ");  for (i = 0; i < n; i++) {  printf("%d ", arr[i]);  }  printf("\n");  return 0;  } |
| **Sample Input:** |
| Original array: 1 2 3 4 5 |
| **Sample Output:** |
| Original array: 1 2 3 4 5  Shuffled array: 5 4 1 2 3 |
| **Time complexity calculation:** |
| O(n) |