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

**Term-work Report**

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Term-work** | *01* | | | | |  |  | | | | |
| **Student Name** | Akash Sainath Nayak | | | | |  |  | | | | |
| **SRN** | 01FE21BEC263 | | | | **Roll Number** | | 519 | | **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(n^2) | | | |
| **2** | | Array | | | | | O(n) | | | |
| Recursive | **1** | | Array | | | | | O(log n) | | | |
| **1** | | - | | | | | O(log n) | | | |
| Back tracking | **2** | | 2D Array | | | | | O(n!) | | | |
| **2** | | Array | | | | | O(2^n) | | | |
| Divide and conquer | **1** | | Array | | | | | O(nlog n) | | | |
| **2** | | Array | | | | | O(nlog n) | | | |
| Dynamic programming | **1** | | Array | | | | | O(n) | | | |
| **2** | | Array | | | | | O(n) | | | |
| Greedy | **1** | | Array | | | | | O(n) | | | |
| **2** | | Array | | | | | O(2^n) | | | |
| Branch and bound | **1** | | 2D Array | | | | | O(n^2) | | | |
| Brute force | **2** | | Array | | | | | O(2^n) | | | |
| **1** | | Array | | | | | O(n^4) | | | |
| Randomized | **2** | | Array | | | | | O(n^2) | | | |
| **1** | | Array | | | | | O(nlog n) | | | |
|  | **2** | | Array | | | | | O(n) | | | |

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

***Yes. I was successfully able to complete the term work. However, I did face some difficulty while learning new algorithms and calculating time complexity of the programs.***

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

***An explanation about calculating time complexity of complex algorithms would be very helpful.***

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

***Practicing more problems is a good way to improve at programming and grasping quickly how an algorithm works.***

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Modularity** |  | **Documentation** |  | **Indentation** |  | **Programming practices** |  |
| **Type of Algorithm: *Iterative*** | | | | | | | |
| **Details of the algorithm:** | | | | | | | |
| *Iterative algorithm in the process of looping over or iterating over a data structure until the required condition is satisfied. It iterates over all the elements and hence is called as iterative algorithm. Such algorithms are used in sorting elements in a Data Structure, searching for an element in a data structure etc. The key is to iterate over the data structure repeatedly till the required condition is met.* | | | | | | | |
| **Code for example 1:** | | | | | | | |
| // 01FE21BEC263 | 519 | E | AKASH S NAYAK  // ITERATIVE ANALYSIS  // PROGRAM TO SORT NUMBERS IN AN ARRAY USING BUBBLE SORT ALGORITHM  // DATA STRUCTURE USED: ARRAY  // TIME COMPLEXITY: O(n^2)    #include <stdio.h>  #include <stdlib.h>  // FUNCTION TO SWAP TWO NUMBERS  void swap(int \*a, int \*b){  int temp = \*a;  \*a=\*b;  \*b=temp;  }    // FUNCTION TO SORT THE GIVEN ARRAY USING BUBBLE SORT  void bubble\_sort(int arr[], int n){  int isSorted = 0;  for(int i=0; i<n-1; i++){  isSorted = 1;  for(int j=0; j<n-i-1; j++){  if(arr[j]>arr[j+1]){  swap(&arr[j], &arr[j+1]);  isSorted = 0;  }  }  }  }    int main(){  // INITIALIZING VARIABLES  int arr[100];  int n = 0;    // TAKING INPUT FROM USER  printf("Enter the number of elements :\n");  scanf("%d", &n);  for(int i=0; i<n; i++){  printf("Enter the element %d of the array :\n", i+1);  scanf("%d", &arr[i]);  }    // SORT THE GIVEN ARRAY USING BUBBLE SORT  bubble\_sort(arr, n);    // PRINT THE SORTED ARRAY  printf("Sorted array :\n");  for(int i=0; i<n; i++){  printf("%d ", arr[i]);  }    return 0;  } | | | | | | | |
| **Sample Input:** | | | | | | | |
| *5*  *12 32 11 31 1* | | | | | | | |
| **Sample Output:** | | | | | | | |
| *Sorted array:*  *1 11 12 31 32* | | | | | | | |
| **Time complexity calculation:** | | | | | | | |
| *The time complexity of the given code is O(n^2), where n is the number of elements in the array. This is because the algorithm uses nested loops to compare and swap elements until the array is fully sorted.*  **for(int i=0; i<n-1; i++){ -----------------------------> n+1**  **isSorted = 1;**  **for(int j=0; j<n-i-1; j++){------------------------> n+1**  **if(arr[j]>arr[j+1]){**  **swap(&arr[j], &arr[j+1]);------------------> (n+1)(n+1)**  **isSorted = 0;**  **}**  **}**  **}**  *The outer loop executes n-1 times and the inner loop executes (n-i-1) times for each i in the outer loop. Therefore, the total number of comparisons and swaps is (n-1)+(n-2)+...+2+1 = n(n-1)/2, which is O(n^2) in terms of time complexity.* | | | | | | | |

|  |
| --- |
| **Code for example 2:** |
| // 01FE21BEC263 | 519 | E | AKASH S NAYAK  // ITERATIVE ANALYSIS  // PROGRAM TO FIND MINIMUN ELEMENT IN AN ARRAY  // DATA STRUCTURE USED: ARRAY  // TIME COMPLEXITY: O(n)  #include <stdio.h>  #include <stdlib.h>  // FUNCTION TO FIND MINIMUM ELEMENT IN THE ARRAY  int find\_min(int arr[], int n){  int min = arr[0];  for(int i=1; i<n ;i++){  if(arr[i] < min){  min = arr[i];  }  }  return min;  }    int main(){  // INITIALIZING VARIABLES  int arr[100];  int n = 0;  // TAKING INPUT FROM USER  printf("Enter the number of elements :\n");  scanf("%d", &n);  for(int i=0; i<n; i++){  printf("Enter the element %d of the array :\n", i+1);  scanf("%d", &arr[i]);  }  // FINDING THE MINIMUM ELEMENT IN THE ARRAY  int min;  min = find\_min(arr, n);  // PRINT THE MINIMUM ELEMENT  printf("Minimum element in the array is : %d\n", min);  return 0;  } |
| **Sample Input:** |
| *5*  *12 32 11 1 2* |
| **Sample Output:** |
| *Minimum element in the array is: 1* |
| **Time complexity calculation:** |
| **for(int i=1; i<n ;i++){-------------------------------> n**  **if(ar[i] < min){--------------------------------> n-1**  **min = arr[i];** |
| *T(n) = 2n –1.*  *The time complexity of the given code is O(n).* |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Modularity** |  | **Documentation** |  | **Indentation** |  | **Programming practices** |  |
| **Type of Algorithm: Recursive** | | | | | | | |
| **Details of the algorithm:** | | | | | | | |
| *Recursive algorithm is an algorithm which calls itself to solve a problem by breaking down into subproblems which are similar to original problem. The name comes from the fact that the algorithm or the function calls itself recursively until a base condition is met with. These algorithms are used to minimize the size of the problem at each step, and it takes less time. For example, calculating factorial of a number, binary search etc. The steps are to identify the base condition for the problem and the recursive part of the problem and to combine them to break the problem into smaller parts until the base condition is met with.* | | | | | | | |
| **Code for example 1:** | | | | | | | |
| // 01FE21BEC263 | 519 | E | AKASH S NAYAK  // RECUSRIVE ALGORITHM  // PROGRAM TO FIND AN ELEMENT USING BINARY SEARCH  // DATA STRUCTURE USED: ARRAY  // TIME COMPLEXITY: O(logn)  #include <stdio.h>  #include <stdlib.h>  // FUNCTION TO FIND KEY IN THE ARRAY USING BINARY SEARCH  int binary\_search(int arr[], int start, int end, int target){  // BASE CONDITION  if(start>end){  return -1;  }  int mid = start + (end-start)/2;  // IF KEY IS FOUND, RETURN INDEX  if(arr[mid] == target){  return mid;  }  else if(arr[mid] > target){  // SEARCH FROM START TO MID-1  return binary\_search(arr, start, mid-1, target);  }  else{  // SEARCH FROM MID+1 TO END  return binary\_search(arr, mid+1, end, target);  }  }  int main(){  // INITIALIZING VARIABLES  int arr[100];  int n = 0;  // TAKING INPUT FROM USER  printf("Enter the number of elements :\n");  scanf("%d", &n);  for(int i=0; i<n; i++){  printf("Enter the element %d of the array :\n", i+1);  scanf("%d", &arr[i]);  }  // KEY TO BE SEARCHED IN ARRAY  printf("Enter the element to be searched :\n");  int key;  scanf("%d", &key);  // RESULT FROM BINARY SEARCH  int res;  res = binary\_search(arr, 0, n-1, key);  if(res == -1){  printf("Key not found in array\n");  }else{  printf("Key found at index %d\n", res);  }  return 0;  } | | | | | | | |
| **Sample Input:** | | | | | | | |
| *5*  *1 2 3 4 5*  *3* | | | | | | | |
| **Sample Output:** | | | | | | | |
| *Key found at index 2* | | | | | | | |
| **Time complexity calculation:** | | | | | | | |
| *The time complexity of the given code is O(log n).*  *- The time complexity of the binary search algorithm is logarithmic because at each step, the size of the search space is divided by two. Therefore, the worst-case time complexity of binary search is O(log n).* *- The maximum number of iterations required to find the target element is log(n) to the base 2.*  **T(n) = O(log n)** | | | | | | | |

|  |
| --- |
| **Code for example 2:** |
| // 01FE21BEC263 | 519 | E | AKASH S NAYAK  // RECUSRIVE ALGORITHM  // PROGRAM TO GENERATE FIND FACTORIAL OF A NUMBER USING RECURSION  // TIME COMPLEXITY: O(log n)  #include <stdio.h>  #include <stdlib.h>  // FUNCTION TO FIND FACTORIAL OF A NUMBER USING RECURSION  int factorial(int num){  if(num==0 || num==1){  return 1;  }  else{  return num \* factorial(num-1);  }  }  int main(){  // USER INPUT  int num;  printf("Enter the number whose factorial is to be found :\n");  scanf("%d",&num);  // FUNCTION CALL AND RESULT  printf("The factorial of %d is %d",num,factorial(num));  return 0;  } |
| **Sample Input:** |
| *5* |
| **Sample Output:** |
| *The factorial of 5 is 120* |
| **Time complexity calculation:** |
| *The time complexity of this code is O(n), where n is the input number for which the factorial is to be found.*   * *The function factorial is called recursively until the base case of num=0 or num=1 is reached.* * *In each recursive call, the function multiplies the current number with the factorial of the previous number.* * *Therefore, the number of recursive calls is equal to the input number.*   ***T(n) = O(n)*** |
|  |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Modularity** |  | **Documentation** |  | **Indentation** |  | **Programming practices** |  |
| **Type of Algorithm: Back tracking** | | | | | | | |
| **Details of the algorithm:** | | | | | | | |
| *Back Tracking is a problem-solving technique that involves trying different solutions to a problem and then discarding them if they don’t give a valid solution. This technique recursively explores all the solutions until a valid solution is found. It’s called backtracking because it backtracks to a previous step if it encounters a dead-end or a violation. These algorithms are used in solving puzzles like sudoku, chess etc. Steps involved are to generate a solution, test the solution and backtrack if necessary. This process is repeated until all solutions are explored.* | | | | | | | |
| **Code for example 1:** | | | | | | | |
| // 01FE21BEC263 | 519 | E | AKASH S NAYAK  // BACKTRACKING ALGORITHM  // PROGRAM TO PLACE N QUEENS ON A CHESSBOARD SUCH THAT NO QUEEN ATTACKS ANY OTHER QUEEN  // DATA STRUCTURE USED: 2D ARRAY  // TIME COMPLEXITY: O(n!)  #include <stdio.h>  #include <stdlib.h>  #define N 4  int chess\_board[N][N];  int isSafe(int row, int col){  int i,j;  // CHECKING FOR QUEENS IN THE SAME ROW  for(i=0; i<col; i++){  if(chess\_board[row][i]){  // NOT SAFE  return 0;  }  }  // CHECKING FOR QUEENS IN THE UPPER DIAGONAL  for (i=row, j=col; i>=0 && j>=0; i--, j--){  if(chess\_board[i][j]){  // NOT SAFE  return 0;  }  }  for (i=row, j=col; j>=0 && i<N; i++, j--){  if(chess\_board[i][j]){  // NOT SAFE  return 0;  }  }  // SAFE  return 1;  }  int solveNQueens(int col){  if(col == N){  return 1;  }  int res=0;  // CHECKING FOR EACH ROW  for(int i=0; i<N; i++){  if(isSafe(i,col)){  chess\_board[i][col] = 1;  // RECURSIVE CALL TO CHECK FOR THE NEXT COLUMN  res = solveNQueens(col+1);  // IF THE NEXT COLUMN IS NOT SAFE, THEN BACKTRACK  if(res){  return 1;  }  chess\_board[i][col] = 0;  }  }  return res;  }  int main(){  int i,j;  for(i=0; i<N; i++){  for(j=0; j<N; j++){  chess\_board[i][j] = 0;  }  }  if(solveNQueens(0)){  for(i =0; i<N; i++){  for(j=0; j<N; j++){  printf("%d ", chess\_board[i][j]);  }  printf("\n");  }  }else{  printf("No solution exists");  }  return 0;  } | | | | | | | |
| **Sample Input:** | | | | | | | |
| *(NONE)* | | | | | | | |
| **Sample Output:** | | | | | | | |
| *0 0 1 0*  *1 0 0 0*  *0 0 0 1*  *0 1 0 0* | | | | | | | |
| **Time complexity calculation:** | | | | | | | |
| *The time complexity is O(n!), where n is number of queens.*  *In the worst case, the function has to explore all possible combinations of queens on the board. There are N queens to be placed on N columns of the board, and each queen can be placed in N possible rows. Therefore, the total number of combinations to explore is N^N. But many of these combinations are invalid because they violate the condition that no two queens should threaten each other. In fact, the number of valid combinations is upper-bounded by N!. Therefore, the worst-case time complexity of the algorithm is O(N!).* | | | | | | | |

|  |
| --- |
| **Code for example 2:** |
| // 01FE21BEC263 | 519 | E | AKASH S NAYAK  // BACKTRACKING ALGORITHM  // PROGRAM TO FIND ALL POSSIBLE SUBSETS OF INTEGERS THAT SUM UP TO A GIVEN NUMBER  // DATA STRUCTURE USED: ARRAY  // TIME COMPLEXITY: O(2^n)    #include <stdio.h>  #include <stdlib.h>    int arr[100];  int subset[100];    // FUNCTION TO FIND ALL POSSIBLE SUBSETS  void subsetSum(int sum, int subsetSize, int currentIndex, int currentSum){  if(currentSum == sum){  // PRINTING THE SUBSET  printf("Subset : ");  for(int i=0; i<subsetSize; i++){  printf("%d ", subset[i]);  }  printf("\n");  return;  }  // IF THE CURRENT SUM IS LESS THAN THE REQUIRED SUM, CONTINUE ADDING ELEMENTS TO THE SUBSET  else if(currentSum < sum){  // ITERATING THROUGH THE ARRAY  for(int i=currentIndex; i>=0; i--){  if(currentSum + arr[i] <= sum){  // ADDING THE ELEMENT TO THE SUBSET  subset[subsetSize] = arr[i];  // RECURSIVE CALL TO CHECK FOR THE NEXT ELEMENT  subsetSum(sum, subsetSize+1, i-1, currentSum+arr[i]);  }  }  }  }  int main(){    int n, sum;    // TAKING INPUT FROM USER  printf("Enter the number of elements :\n");  scanf("%d", &n);  for(int i=0; i<n; i++){  printf("Enter the element %d of the array :\n", i+1);  scanf("%d", &arr[i]);  }    printf("Enter the sum :\n");  scanf("%d", &sum);    // FUNCTION CALL TO FIND ALL POSSIBLE SUBSETS  subsetSum(sum, 0, n-1, 0);    return 0;  } |
| **Sample Input:** |
| *6*  *1 2 3 4 5 6*  *10* |
| **Sample Output:** |
| *Subset : 6 4*  *Subset : 6 3 1*  *Subset : 5 4 1*  *Subset : 5 3 2*  *Subset : 4 3 2 1* |
| **Time complexity calculation:** |
| *The time complexity of the given code is O(2^n).*  *The function subsetSum is called recursively for each element in the array arr. In each recursive call, the function either adds the current element to the subset or not. Thus, there are two choices for each element, either to include it in the subset or not. Hence, the total number of subsets is 2^n (where n is the number of elements in arr). Therefore, the time complexity of the code is O(2^n).* |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Modularity** |  | **Documentation** |  | **Indentation** |  | **Programming practices** |  |
| **Type of Algorithm: Divide and Conquer** | | | | | | | |
| **Details of the algorithm:** | | | | | | | |
| *Divide and conquer is an algorithm that involves breaking down a complex problem into smaller sub-problems and solving them as they are easy to solve. The solutions are combined to solve the original problem. It is used in sorting, searching and matrix multiplication problems. The steps are to Divide the problem into smaller sub-problems and solve (Conquer) them separately and combine the solutions of sub-problems to solve original problem.* | | | | | | | |
| **Code for example 1:** | | | | | | | |
| // 01FE21BEC263 | 519 | E | AKASH S NAYAK  // DIVIDE AND CONQUER ALGORITHM  // PROGRAM TO SORT AN ARRAY USING MERGE SORT  // DATA STRUCTURE USED: ARRAY  // TIME COMPLEXITY: O(nlogn)    #include <stdio.h>  #include <stdlib.h>    // MERGE TWO SUBARRAYS OF arr[]  void merge(int arr[], int l, int m, int r)  {  int i, j, k;  int n1 = m - l + 1;  int n2 = r - m;    int L[n1], R[n2];    for (i = 0; i < n1; i++)  L[i] = arr[l + i];  for (j = 0; j < n2; j++)  R[j] = arr[m + 1 + j];    i = 0;  j = 0;  k = l;  while (i < n1 && j < n2) {  if (L[i] <= R[j]) {  arr[k] = L[i];  i++;  }  else {  arr[k] = R[j];  j++;  }  k++;  }    while (i < n1) {  arr[k] = L[i];  i++;  k++;  }    while (j < n2) {  arr[k] = R[j];  j++;  k++;  }  }    // MERGE SORT FUNCTION  void mergeSort(int arr[], int l, int r)  {  if (l < r) {  int m = l + (r - l) / 2;    mergeSort(arr, l, m);  mergeSort(arr, m + 1, r);    merge(arr, l, m, r);  }  }    int main(){  // INITIALIZING VARIABLES  int arr[100];  int n = 0;    // TAKING INPUT FROM USER  printf("Enter the number of elements :\n");  scanf("%d", &n);  for(int i=0; i<n; i++){  printf("Enter the element %d of the array :\n", i+1);  scanf("%d", &arr[i]);  }    // SORTING THE ARRAY  mergeSort(arr, 0, n-1);    // PRINT THE SORTED ARRAY  printf("Sorted array is : \n");  for(int i=0; i<n; i++){  printf("%d ", arr[i]);  }    return 0;  } | | | | | | | |
| **Sample Input:** | | | | | | | |
| *5*  *5 4 3 2 1* | | | | | | | |
| **Sample Output:** | | | | | | | |
| *Sorted array is :*  *1 2 3 4 5* | | | | | | | |
| **Time complexity calculation:** | | | | | | | |
| *The time complexity is O(nlogn), which means the time grows logarithmically with growth in n (size).*  *Using the Master Theorem, we can determine that the time complexity of the mergeSort function is O(nlogn). Therefore, the overall time complexity of the given code is O(nlogn).*  **T(n) = 2T(n/2) + O(n)** | | | | | | | |

|  |
| --- |
| **Code for example 2:** |
| // 01FE21BEC263 | 519 | E | AKASH S NAYAK  // DIVIDE AND CONQUER ALGORITHM  // PROGRAM TO SORT AN ARRAY USING QUICK SORT  // DATA STRUCTURE USED: ARRAY  // TIME COMPLEXITY: O(nlogn)  #include <stdio.h>  #include <stdlib.h>  // FUNCTION TO SWAP TWO ELEMENTS  void swap(int \*a, int \*b){  int temp = \*a;  \*a = \*b;  \*b = temp;  }  // PARTITION FUNCTION  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);  }  // QUICK SORT FUNCTION  void quickSort(int arr[], int low, int high){  if(low <high){  int part\_idx = partition(arr, low, high);  // SORT ELEMENTS BEFORE AND AFTER PARTITION  quickSort(arr, low, part\_idx-1);  quickSort(arr, part\_idx+1, high);  }  }  int main(){  // INITIALIZING VARIABLES  int arr[100];  int n = 0;  // TAKING INPUT FROM USER  printf("Enter the number of elements :\n");  scanf("%d", &n);  for(int i=0; i<n; i++){  printf("Enter the element %d of the array :\n", i+1);  scanf("%d", &arr[i]);  }  // SORTING THE ARRAY  quickSort(arr, 0, n-1);    // PRINT THE SORTED ARRAY  printf("Sorted array is : \n");  for(int i=0; i<n; i++){  printf("%d ", arr[i]);  }  return 0;  } |
| **Sample Input:** |
| *5*  *5 4 3 2 1* |
| **Sample Output:** |
| *Sorted array is :*  *1 2 3 4 5* |
| **Time complexity calculation:** |
| *The time complexity of the quick sort algorithm is O(nlogn) in the average and best case and O(n^2) in the worst case.*  *In this implementation, the worst case scenario can occur if the pivot chosen is always the smallest or largest element of the array, which would result in a partition of size n-1 and a recursive call on n-1 elements, leading to O(n^2) time complexity.*  *However, in the average and best case scenarios, the partition function divides the array into two halves of roughly equal size, leading to a balanced recursion tree, which results in an average time complexity of O(nlogn).*  ***T(n) = T(k) + T(n-k-1) + O(n)----> if pivot is chosen correctly***  *Thus, the time complexity of this implementation is O(nlogn).* |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Modularity** |  | **Documentation** |  | **Indentation** |  | **Programming practices** |  |
| **Type of Algorithm: Brute Force** | | | | | | | |
| **Details of the algorithm:** | | | | | | | |
| *Brute force is a straightforward algorithm which evaluates all the possible solutions and selects the best one. It is called brute force because it exhaustively examines every possible option without consideration for efficiency or optimization. Applications include password cracking, cryptographic attacks etc. Basics steps are to generate all possible solutions, evaluate each solution, select the best solution for the problem.* | | | | | | | |
| **Code for example 1:** | | | | | | | |
| // 01FE21BEC263 | 519 | E | AKASH S NAYAK  // BRUTE FORCE ALGORITHM  // PROGRAM TO CRACK A 4 DIGIT pin  // DATA STRUCTURE USED: ARRAY  // TIME COMPLEXITY: O(n^4)    #include <stdio.h>  #include <stdlib.h>    int main(){    // INITIALIZING VARIABLES  int i,j,k,l;  char pin[4];  char try[4];  char ch;    // TAKING INPUT  printf("Enter your pin: ");  for(i=0;i<4;i++){  scanf("%c",&pin[i]);  }    printf("\n");  // BRUTE FORCE ALGORITHM TO FIND ALL POSSIBLE COMBINATIONS AND CHECK IF IT MATCHES THE PIN  for(i=0;i<4;i++){  for(j=0;j<4;j++){  for(k=0;k<4;k++){  for(l=0;l<4;l++){  // TRYING ALL POSSIBLE COMBINATIONS  try[0]=pin[i];  try[1]=pin[j];  try[2]=pin[k];  try[3]=pin[l];    // CHECKING IF THE GUESS MATCHES THE PIN  if(try[0]==pin[0] && try[1]==pin[1] && try[2]==pin[2] && try[3]==pin[3]){  printf("The pin is: ");  for(int m=0;m<4;m++){  printf("%c",try[m]);  }  // IF THE GUESS MATCHES THE PIN, END THE PROGRAM  printf("\n");  printf("The pin is cracked\n");  return 0;  }  }  }  }  }  // IF THE GUESS DOES NOT MATCH THE PIN, PIN IS NOT CRACKED  printf("The pin is not cracked\n");  return 0;  } | | | | | | | |
| **Sample Input:** | | | | | | | |
| *Enter your pin: 9999* | | | | | | | |
| **Sample Output:** | | | | | | | |
| *The pin is: 9999*  *The pin is cracked* | | | | | | | |
| **Time complexity calculation:** | | | | | | | |
| *The time complexity of the code is O(n^4).*  **for(i=0;i<4;i+****+){-----------------------------> n+1**  **for(j=0;j<4;j++****){---------------------> n+1**  **for(k=0;k<4;k++){------------------> n+1**  **for(l=0;l<4;l++){--------------> n+1**  ***//statements--------------------------------------------------------------> n^4***  ***T(n) = O(n^4)*** | | | | | | | |

|  |
| --- |
| **Code for example 2:** |
| // 01FE21BEC263 | 519 | E | AKASH S NAYAK  // BRUTE FORCE ALGORITHM  // PROGRAM TO FIND ALL PAIRS OF ELEMENTS IN AN ARRAY WHOSE SUM IS EQUAL TO A GIVEN NUMBER  // DATA STRUCTURE USED: ARRAY  // TIME COMPLEXITY: O(n^2)    #include <stdio.h>  #include <stdlib.h>    void find\_pairs(int arr[], int n, int sum){  for(int i=0; i<n-1; i++){  for(int j=i+1; j<n; j++){  if(arr[i] + arr[j] == sum){  printf("The pair is (%d, %d)\n", arr[i], arr[j]);  }  }  }  }    int main(){  // INITIALIZING VARIABLES  int arr[100];  int n;    // TAKING INPUT FROM USER  printf("Enter the number of elements :\n");  scanf("%d", &n);  for(int i=0; i<n; i++){  printf("Enter the element %d of the array :\n", i+1);  scanf("%d", &arr[i]);  }    // TAKING INPUT OF THE SUM  int sum;  printf("Enter the sum :\n");  scanf("%d", &sum);    // CHECKING FOR PAIRS  find\_pairs(arr, n, sum);    return 0;  } |
| **Sample Input:** |
| *5*  *5 4 3 2 1*  *6* |
| **Sample Output:** |
| *The pair is (1, 5)*  *The pair is (2, 4)* |
| **Time complexity calculation:** |
| *The time complexity is O(n^2).*  *The function find\_pairs contains two nested loops that iterate through the entire array. Therefore, the time complexity of the function is O(n^2). The find\_pairs function is called once in the main function, so the overall time complexity of the code is O(n^2).*  **T(n) = a \* n^2 + b** |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Modularity** |  | **Documentation** |  | **Indentation** |  | **Programming practices** |  |
| **Type of Algorithm: Dynamic programming** | | | | | | | |
| **Details of the algorithm:** | | | | | | | |
| *Dynamic programming is a technique used to solve complex optimization problems by breaking them down to simple sub-problems of similar kind and storing the result to avoid redundant computations.* *It gets its name from the fact that it uses a dynamic (i.e., changing) programming table to store the solutions to sub-problems. It is used in optimization problems, game theory, economics etc. Steps are to develop a sub-problem, a recursive algorithm and design the dynamic programming table. Fill the table and solve the problem by finding solution.* | | | | | | | |
| **Code for example 1:** | | | | | | | |
| // 01FE21BEC263 | 519 | E | AKASH S NAYAK  // DYNAMIC PROGRAMMING  // PROGRAM TO FIND FIBONACCI SERIES USING DYNAMIC PROGRAMMING  // TIME COMPLEXITY: O(n)    #include<stdio.h>  #include<stdlib.h>    int fibbonacciDp(int n){  // INITIALIZE ARRAY TO STORE THE FIBONACCI SERIES  int fib[n+1];  int i;    // INITIALIZE THE FIRST TWO VALUES OF THE SERIES  fib[0] = 0;  fib[1] = 1;    // ITERATE THROUGH THE ARRAY TO FIND THE FIBONACCI SERIES  for(i=2; i<=n; i++){  fib[i] = fib[i-1] + fib[i-2];  }    return fib[n];  }    int main(){  int n;  printf("Enter the number : \n");  scanf("%d",&n);  printf("The nth fibonacci number is %d \n", fibbonacciDp(n));  return 0;  } | | | | | | | |
| **Sample Input:** | | | | | | | |
| *20* | | | | | | | |
| **Sample Output:** | | | | | | | |
| *The nth fibonacci number is 6765* | | | | | | | |
| **Time complexity calculation:** | | | | | | | |
| *The time complexity of this code is O(n).*  *This algorithm uses dynamic programming to calculate the Fibonacci number. Initializes an array of size n+1 to store the Fibonacci sequence. The first two values of the series are initialized to 0 and 1, respectively. The algorithm then traverses the array starting from the third element and i. Calculates the element as the sum of the (i-1)th and (i-2)th elements.*  *Since the algorithm traverses the sequence only once, the time complexity is O(n); where n is the number input.*  ***T(n) = O(n)*** | | | | | | | |

|  |
| --- |
| **Code for example 2:** |
| // 01FE21BEC263 | 519 | E | AKASH S NAYAK  // DYNAMIC PROGRAMMING  // PROGRAM TO FIND FACTORIAL OF A NUMBER USING DYNAMIC PROGRAMMING  // TIME COMPLEXITY: O(n)    #include<stdio.h>  #include<stdlib.h>    int factorialDp(int n){  // INITIALIZE ARRAY TO STORE THE FACTORIAL  int fact[n+1];  // INITIALIZE THE FIRST VALUE OF THE FACTORIAL  fact[0] = 1;    // ITERATE THROUGH THE ARRAY TO FIND THE FACTORIAL  for(int i=1; i<=n; i++){  fact[i] = i\*fact[i-1];  }    return fact[n];  }    int main(){  int n;  printf("Enter the number : \n");  scanf("%d",&n);  printf("The factorial of %d is : %d \n",n, factorialDp(n));  return 0;  } |
| **Sample Input:** |
| *25* |
| **Sample Output:** |
| *The factorial of 25 is : 2076180480* |
| **Time complexity calculation:** |
| *The time complexity of this code is O(n).*   *The number uses dynamic programming to find the factorial of a number. Initializes an array of size n+1 to store the factorial and sets the initial value to 1. It then iterates array till n and calculates the factorial of each number by multiplying the factorial of the previous number.*   *The time complexity of this code is O(n) because it iterates once through the array and performs a fixed number of tasks per iteration.*  ***T(n): T(n) = O(n)*** |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Modularity** |  | **Documentation** |  | **Indentation** |  | **Programming practices** |  |
| **Type of Algorithm: Greedy Algorithm** | | | | | | | |
| **Details of the algorithm:** | | | | | | | |
| *Greedy algorithm is an algorithmic pattern which focuses on finding a global optimum. It tends to find the best choice at each step. It cares about finding good choices in the current step, and does not worry about future consequences, which is why it is called greedy. This algorithm is used in network routing, data compression etc. General steps are to define the problem in a set of choices, determine a criterion for making a decision at a step, evaluate each choice based on the criteria, choose the best choice at the moment and repeat until the problem is solved.* | | | | | | | |
| **Code for example 1:** | | | | | | | |
| // 01FE21BEC263 | 519 | E | AKASH S NAYAK  // GREEDY ALGORITHM  // PROGRAM TO SOLVE THE COIN CHANGE PROBLEM USING GREEDY ALGORITHM  // DATA STRUCTURE USED: ARRAY  // TIME COMPLEXITY: O(n)    #include <stdio.h>  #include <stdlib.h>  // FUNCTION TO FIND MIN COINS REQUIRED  void coinChange(int coins[], int size, int amount){  int count=0;  for(int i=0; i<size; i++){    // USING THE GREEDY APPROACH  while(amount >= coins[i]){  // IF AMOUNT IS GREATER THAN THE COIN VALUE, SUBTRACT AND INCREASE COUNT  amount -= coins[i];  count++;  }    if(amount==0){  break;  }  }    if (amount==0)  {  printf("Minimum number of coins required: %d\n", count);  }  else  {  printf("Change not possible\n");  }  }    int main(){  // INITIALIZING THE COINS ARRAY (DENOMINATIONS IN DESCENDING ORDER)  int coins\_arr[] = {2000, 500, 200, 100, 50, 20, 10, 5, 2, 1};  int size = sizeof(coins\_arr)/sizeof(coins\_arr[0]);    // TAKING THE AMOUNT AS INPUT FROM USER  int amount;  printf("Enter the amount: ");  scanf("%d", &amount);    // FUNCTION CALL TO FIND THE MINIMUM NUMBER OF COINS REQUIRED  coinChange(coins\_arr, size, amount);    return 0;  } | | | | | | | |
| **Sample Input:** | | | | | | | |
| *264* | | | | | | | |
| **Sample Output:** | | | | | | | |
| *Minimum number of coins required: 5* | | | | | | | |
| **Time complexity calculation:** | | | | | | | |
| *The size of the coin string is n.*  *The loop is iterated n times.*  *A loop has a time loop that repeats up to n times.*  *Therefore, the time complexity of the* ***coinChange*** *function is O(n + n) = O(n).*  *The total time complexity of this program is also O(n) because the only function called in the program is* ***coinChange*** *which has a time complexity of O(n).* | | | | | | | |

|  |
| --- |
| **Code for example 2:** |
| // 01FE21BEC263 | 519 | E | AKASH S NAYAK  // GREEDY ALGORITHM  // PROGRAM TO SOLVE THE FRACTIONAL KNAPSACK PROBLEM  // DATA STRUCTURE USED: ARRAY  // TIME COMPLEXITY: O(2^n)  #include <stdio.h>  #include <stdlib.h>  // STRUCTURE TO STORE THE WEIGHT AND VALUE OF THE ITEMS  struct item{  int weight;  int value;  };  // FUNCTION TO SWAP TWO STRUCTURES  int swap(struct item \*a, struct item \*b){  struct item temp = \*a;  \*a = \*b;  \*b = temp;  }  // FUNCTION TO IMPLEMENT PARTITION IN QUICK SORT  int partition(struct item arr[], int low, int high){  int pivot = arr[high].value / arr[high].weight;  int i = low - 1;    for(int j = low; j < high; j++){  if(arr[j].value / arr[j].weight < pivot){  i++;  swap(&arr[i], &arr[j]);  }  }  swap(&arr[i + 1], &arr[high]);  return (i + 1);  }  // FUNCTION TO IMPLEMENT QUICK SORT TO SORT THE ARRAY OF STRUCTURES IN DESCENDING ORDER  void quickSort(struct item arr[], int low, int high){  if(low < high){  int pivot = partition(arr, low, high);    quickSort(arr, low, pivot - 1);  quickSort(arr, pivot + 1, high);  }  }  // FUNCTION TO COMPARE THE RATIO OF VALUE AND WEIGHT  int compare(const void \*a, const void \*b){  // TYPECASTING THE PARAMETERS  struct item \*item1 = (struct item \*)a;  struct item \*item2 = (struct item \*)b;  // CALCULATING THE RATIO  double ratio1 = (double)item1->value / item1->weight;  double ratio2 = (double)item2->value / item2->weight;  // COMPARING THE RATIO  if(ratio1 > ratio2)  return -1;  else if(ratio1 < ratio2)  return 1;  else  return 0;  }  // FUNCTION TO SOLVE THE FRACTIONAL KNAPSACK PROBLEM  double fractionalKnapsack(int weight, struct item arr[], int n){  quickSort(arr, 0, n - 1);  double totalValue = 0.0;  int i;  for(i=0; i<n; i++){  // USING THE GREEDY APPROACH  if(weight == 0){  return totalValue;  }  else if(arr[i].weight <= weight){  totalValue += arr[i].value;  weight -= arr[i].weight;  }else{  totalValue += ((double)arr[i].value / arr[i].weight) \* weight;  weight = 0;  }  }  return totalValue;  }  int main(){  // INITIALISING THE ARRAY OF STRUCTURES    struct item arr[] = {{10, 60}, {20, 100}, {30, 120}};  int size = sizeof(arr) / sizeof(arr[0]);  int weight;  printf("Enter the weight: ");  scanf("%d", &weight);  printf("Maximum value we can obtain = %lf", fractionalKnapsack(weight, arr, size));  return 0;  } |
| **Sample Input:** |
| *50* |
| **Sample Output:** |
| *Maximum value we can obtain = 220.000000* |
| **Time complexity calculation:** |
| *The time complexity of the given code is O(nlog n).*  *Code to first make a QuickSort on array of structures that will take O(nlogn).*  *Then iterates through the array of structures , which takes O(n) time.*  *So the total time complexity of the code is O(nlogn) + O(n) = O(nlogn).*  ***T(n) = O(nlogn)*** |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Modularity** |  | **Documentation** |  | **Indentation** |  | **Programming practices** |  |
| **Type of Algorithm: Branch and Bound Algorithm** | | | | | | | |
| **Details of the algorithm:** | | | | | | | |
| *Branch and bound is an optimization algorithm which is used to solve optimization problems by exploring the search space of favourable solutions. It is a technique to solve combinatorial optimization problems such as the traveling salesman problem (TSP) and the knapsack problem. The name "branch and bound" comes from the fact that the algorithm divides the problem into smaller sub-problems, called branches, and bounds the solutions obtained at each node to ensure optimality. The algorithm systematically searches through the search space by branching on each node of the search tree and assigning bounds to each sub-problem. Steps would be to search the tree with root node, expand current node by creating one or more child nodes, calculate lower or upper bound for each child node, discard a child node if its bound is worse than best solution so far, choose the child node with the best bound and repeat till search is complete.* | | | | | | | |
| **Code for example 1:** | | | | | | | |
| // 01FE21BEC263 | 519 | E | AKASH S NAYAK  // BRANCH AND BOUND ALGORITHM  // PROGRAM TO SOLVE THE TRAVELLING SALESMAN PROBLEM USING BRANCH AND BOUND ALGORITHM  // DATA STRUCTURE USED: 2D ARRAY  // TIME COMPLEXITY: O(n^2)  #include <stdio.h>  #include <stdlib.h>  int a[10][10], visited[10], n, cost = 0;  int least(int currCity){  int i, nearest = 999;  int min = 999, minKey;  // FINDING THE MINIMUM COST EDGE  for(i = 0; i < n; i++){  // CHECKING IF THE EDGE IS VISITED OR NOT  if((a[currCity][i] != 0) && (visited[i] == 0)){  // CHECKING IF THE EDGE IS MINIMUM OR NOT  if(a[currCity][i] < min){  // UPDATING THE MINIMUM COST EDGE  min = a[i][0] + a[currCity][i];  minKey = a[currCity][i];  nearest = i;  }  }  }  // UPDATING THE COST  if(min != 999){  cost += minKey;  }    return nearest;  }  // FUNCTION TO FIND THE MINIMUM COST  void minCost(int city){  int i;  int ncity;  visited[city] = 1;  printf("%d --> ", city + 1);  // FINDING THE NEXT NEAREST CITY  ncity = least(city);  if(ncity == 999){  // IF ALL THE CITIES ARE VISITED  ncity = 0;  printf("%d", ncity + 1);  cost += a[city][ncity];  return;  }  // RECURSION TO FIND THE MINIMUM COST  minCost(ncity);  }  int main(){  int i, j;  printf("Enter the number of cities: ");  scanf("%d", &n);  printf("Enter the cost matrix:\n");  for(i = 0; i < n; i++){  printf("Enter elements of row %d: ", i + 1);  for(j = 0; j < n; j++){  scanf("%d", &a[i][j]);  }  visited[i] = 0;  }  printf("\nThe cost matrix is:\n");  for(i = 0; i < n; i++){  printf("\n");  for(j = 0; j < n; j++){  printf("\t%d", a[i][j]);  }  }  printf("\nThe path is:\n");  minCost(0);  printf("\nMinimum cost is %d\n", cost);  return 0;  } | | | | | | | |
| **Sample Input:** | | | | | | | |
| *4*  *0 10 15 20*  *10 0 35 25*  *15 35 0 30*  *20 25 30 0* | | | | | | | |
| **Sample Output:** | | | | | | | |
| *The path is:*  *1 --> 4 --> 3 --> 2 --> 1*  *Minimum cost is 95* | | | | | | | |
| **Time complexity calculation:** | | | | | | | |
| *The time complexity is O(n^2).*  *The function least() has a for loop that iterates over all the cities to find the nearest unvisited city. This loop runs n times, where n is the number of cities, and therefore its time complexity is O(n).*  *The function minCost() calls least() function, which has a time complexity of O(n). It then recursively calls itself, with least() being called again, and so on until all cities are visited. In the worst-case scenario, the recursion goes through all the cities, and therefore minCost() has a time complexity of O(n^2).*  *The main function has two nested for loops that iterate over all the cities to read the cost matrix. These loops run n^2 times, and therefore their time complexity is O(n^2).*  ***T(n) = O(n^2) + O(n) + O(n^2) + O(n^2) = O(n^2)*** | | | | | | | |

|  |
| --- |
| **Code for example 2:** |
| // 01FE21BEC263 | 519 | E | AKASH S NAYAK  // BRANCH AND BOUND ALGORITHM  // PROGRAM TO SOLVE THE 0/1 KNAPSACK PROBLEM USING BRANCH AND BOUND ALGORITHM  // DATA STRUCTURE USED: ARRAY  // TIME COMPLEXITY: O(2^n)  #include <stdio.h>  #include <stdlib.h>  #define N 4  #define W 10  typedef struct{  int weight;  int value;  double ratio;  } item;    // FUNCTION TO SORT RATIOS IN DESCENDING ORDER  int compare\_items(const void\*a, const void\*b){  item \*item1 = (item \*)a;  item \*item2 = (item \*)b;  double diff = item2->ratio - item1->ratio;  if(diff > 0)  return 1;  else if(diff < 0)  return -1;  else  return 0;  }    // FUNCTION TO CALCULATE THE LOWER BOUND  double lower\_bound(int depth, int weight, int value, item\* items, int n){  double bound = value;  int w = weight;  for(int i = depth; i < n; i++){  if(w + items[i].weight <= W){  w += items[i].weight;  bound += items[i].value;  }  else{  int remain = W - w;  bound += items[i].ratio \* remain;  break;  }  }  return bound;  }    // BRANCH AND BOUND FUNCTION  int branch\_n\_bound(int depth, int weight, int value, item\* items, int n, int \*solution){  // BASE CASE  if(depth == n){  return value;  }    // COMPUTING THE LOWER BOUND  double bound = lower\_bound(depth, weight, value, items, n);  if(bound < solution[1]){  return -1;  }    // BRANCH ON NEXT ITEM  int include =0, exclude = 0;  if(weight + items[depth].weight <= W){  include = branch\_n\_bound(depth + 1, weight + items[depth].weight, value + items[depth].value, items, n, solution);  }  exclude = branch\_n\_bound(depth + 1, weight, value, items, n, solution);    // UPDATE THE SOLUTION  if(include > exclude){  solution[0] = depth;  solution[1] = include;  return include;  }else{  return exclude;  }    }  int main(){  // INITIALISING THE ITEMS  item items[N];  items[0].weight = 2;  items[0].value = 40;  items[1].weight = 3;  items[1].value = 50;  items[2].weight = 5;  items[2].value = 100;  items[3].weight = 7;  items[3].value = 150;    // COMPUTING THE RATIO  for(int i = 0; i < N; i++){  items[i].ratio = (double)items[i].value / items[i].weight;  }    // SORTING THE ITEMS IN DESCENDING ORDER OF RATIO  qsort(items, N, sizeof(item), compare\_items);    // CALLING THE BRANCH AND BOUND FUNCTION  int solution[2] = {-1, 0};  branch\_n\_bound(0, 0, 0, items, N, solution);    // PRINTING THE SOLUTION  printf("The selected items are : ");  for(int i = 0; i < N; i++){  if(solution[0] == i){  printf("1 ");  }else{  printf("0 ");  }  }    return 0;  } |
| **Sample Input:** |
| *-* |
| **Sample Output:** |
| *The selected items are: 1 0 0 0* |
| **Time complexity calculation:** |
| *Sorting the items in descending order of ratio using the qsort function: O(N\*log(N))*  *For each item, computing the ratio and assigning it to the corresponding item: O(N)*  *Calling the branch\_n\_bound function: O(2^N)*  *For each item in the branch\_n\_bound function, computing the lower bound: O(N)*  *For each item in the branch\_n\_bound function, checking if the bound is less than the current best solution: O(1)*  *For each item in the branch\_n\_bound function, branching on the next item and calling the function recursively: O(2^N)*  *Updating the solution if a better solution is found: O(1)*  Therefore, the total time complexity of the given code is **O(N*log(N)) + O(N) + O(2^N)*[O(N) + O(1) + O(2^N)\*[O(1)]] + O(1).**  **T(n) = O(2^n)** |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Modularity** |  | **Documentation** |  | **Indentation** |  | **Programming practices** |  |
| **Type of Algorithm: Randomized Algorithm** | | | | | | | |
| **Details of the algorithm:** | | | | | | | |
| *Randomized algorithm employs a degree of randomness in its logic to solve a problem. It gets the name because it used a random number generator for decision making during execution. Randomized algorithms have several applications, including cryptography, optimization, and machine learning. One of the main advantages of randomized algorithms is that they can often solve problems more efficiently than their deterministic counterparts. General steps would be to generate a random number or a set of random numbers to make decisions during algorithm execution, repeat this multiple times till accuracy improves and reduces chances of errors.* | | | | | | | |
| **Code for example 1:** | | | | | | | |
| // 01FE21BEC263 | 519 | E | AKASH S NAYAK  // RANDOMIZED ALGORITHM  // PROGRAM TO SORT AN ARRAY USING RANDOMIZED QUICK SORT  // DATA STRUCTURE USED: ARRAY  // TIME COMPLEXITY: O(nlogn)  // WORST CASE: O(n^2) ==> IMPLEMENATION OF RANDOMIZED QUICK SORT TO AVOID WORST CASE  #include <stdio.h>  #include <stdlib.h> #include <time.h>  // FUNCTION TO SWAP TWO ELEMENTS  void swap(int \*a, int \*b){  int temp = \*a;  \*a = \*b;  \*b = temp;  }  // FUNCTION TO FIND THE PARTITION  int partition(int arr[], int low, int high){  int pivot = arr[high];  int i = low - 1;  int j;  for(j = low; j < high; j++){  if(arr[j] < pivot){  i++;  swap(&arr[i], &arr[j]);  }  }  swap(&arr[i + 1], &arr[high]);  return (i + 1);  }  // FUNCTION TO IMPLEMENT RANDOMIZED PARTITION  int random\_partition(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);  }    // FUNCTION TO IMPLEMENT RANDOMIZED QUICK SORT  void randomized\_quick\_sort(int arr[], int low, int high){  if(low < high){  int r\_pivot = random\_partition(arr, low, high);  randomized\_quick\_sort(arr, low, r\_pivot - 1);  randomized\_quick\_sort(arr, r\_pivot + 1, high);  }  }  int main(){  // INITIALIZING VARIABLES  int arr[100];  int n = 0;    // TAKING INPUT FROM USER  printf("Enter the number of elements :\n");  scanf("%d", &n);  for(int i=0; i<n; i++){  printf("Enter the element %d of the array :\n", i+1);  scanf("%d", &arr[i]);  }  // SORTING THE ARRAY  randomized\_quick\_sort(arr, 0, n-1);  // PRINTING THE SORTED ARRAY  printf("The sorted array is :\n");  for(int i=0; i<n; i++){  printf("%d ", arr[i]);  }    return 0;  } | | | | | | | |
| **Sample Input:** | | | | | | | |
| *5*  *5 4 3 2 1* | | | | | | | |
| **Sample Output:** | | | | | | | |
| *The sorted array is :*  *1 2 3 4 5* | | | | | | | |
| **Time complexity calculation:** | | | | | | | |
| *Time complexity is O(n log n) in the average case and O(n^2) in the worst case.*  *The partition function takes O(n) time .*  *The* ***randomized\_partition*** *function time complexity of partitioning is O(log n).*  *The* ***randomized\_quick\_sort*** *function calls* ***randomized\_partition*** *and recursively calls itself twice, so its time complexity can be written as T(n) = 2T(n/2) + O(n) in the worst case and T(n) = 2T(n/2) + O(log n) in the average case.*  *Using the Master theorem, we can solve the recurrence relation to find that the time complexity of* ***randomized****\_****quick****\_****sort*** *is O(n log n) in the average case and O(n^2) in the worst case.* | | | | | | | |

|  |
| --- |
| **Code for example 2:** |
| // 01FE21BEC263 | 519 | E | AKASH S NAYAK  // RANDOMIZED ALGORITHM  // PROGRAM TO IMPLEMENT MONTE CARLO METHOD TO FIND THE VALUE OF PI  // DATA STRUCTURE USED: ARRAY  // TIME COMPLEXITY: O(n)  #include <stdio.h>  #include <stdlib.h>  #include <time.h>  int main(){  srand(time(NULL));  // INITIALIZING THE SEED  int n= 10000;  int count = 0;  // TAKING THE NUMBER OF POINTS  for(int i=0; i<n; i++){  // GENERATING RANDOM POINTS  double x = (double)rand() / RAND\_MAX;  double y = (double)rand() / RAND\_MAX;  // CHECKING IF THE POINTS LIE INSIDE THE CIRCLE  if(x\*x + y\*y <= 1){  count++;  }  }  // CALCULATING THE VALUE OF PI  double pi = (double)count / n \* 4;  printf("The value of pi is %lf\n", pi);  } |
| **Sample Input:** |
| *-* |
| **Sample Output:** |
| *The value of pi is 3.115600* |
| **Time complexity calculation:** |
| *The time complexity of this code is O(n), where n is the number of points generated in main function.*  ***T(n) = O(n)*** |