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

**Term-work Report**



|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Term-work** | **01** | | | | |  |  | | | | |
| **Student Name** | **Prajwal B** | | | | |  |  | | | | |
| **SRN** | **01FE20BEC196** | | | | **Roll Number** | | **564** | | **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** | | None | | | | | O(n) | | | |
| **2** | | Array | | | | | O(n) | | | |
| Recursive | **1** | | None | | | | | O(2^n) | | | |
| **2** | | Array | | | | | O(logn) | | | |
| Back tracking | **1** | | Array | | | | | O(2^n) | | | |
| **2** | | 2-D Array | | | | | O(n!) | | | |
| Divide and conquer | **1** | | Array | | | | | *O(n log n)* | | | |
| **2** | | Array | | | | | *O(n log n)* | | | |
| Dynamic programming | **1** | | Array | | | | | O(n) | | | |
| **2** | | Array | | | | | O(n) | | | |
| Greedy | **1** | | Array | | | | | O(n) | | | |
| **2** | | Array | | | | | O(n) | | | |
| Branch and bound | **1** | | Array | | | | | O(n!) | | | |
| **2** | | Array | | | | | O(2^n) | | | |
| Brute force | **1** | | Strings | | | | | O(1) | | | |
| **2** | | Strings | | | | | O(n!) | | | |
| Randomized | **1** | | Array | | | | | O(n) | | | |
| **2** | | Array | | | | | O(n \* sqrt(n)) | | | |

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

*<Write your answer here>*

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

*<Write your answer here>*

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

* Data structures,Pointers.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Modularity** |  | **Documentation** |  | **Indentation** |  | **Programming practices** |  |
| **Type of Algorithm:** *Iterative* | | | | | | | |
| **Details of the algorithm:** | | | | | | | |
| *An iterative algorithm is a type of algorithm that uses a loop to repeat a set of instructions until a certain condition is met. The name "iterative" comes from the fact that the algorithm iterates or repeats a process multiple times.*  *STEPS :*  *1.Initialization: Set initial values for variables, data structures, or parameters required by the algorithm.*  *2.Loop or iteration: Perform a series of iterations or loops to solve the problem incrementally. The number of iterations may be fixed or depend on certain conditions.*  *3.Update or modify: Within each iteration, update variables, data structures, or parameters based on the problem's requirements or constraints.*  *4.Termination condition: Define a condition that determines when to stop the iterations. This can be a fixed number of iterations, reaching a specific solution accuracy, or satisfying certain constraints.*  *5.Output or result: Once the iterations are complete, retrieve or compute the final result of the algorithm.*  *6.Validation or evaluation: Validate and evaluate the output of the algorithm to ensure it meets the desired objectives or requirements.* | | | | | | | |
| **Code for example 1:** | | | | | | | |
| /\* Factorial Of A Number. \*/  #include<stdio.h>  int factorial(int);  int main()  {      int num;      printf("Enter a number : ");      scanf("%d",&num);      int fact = factorial (num);      printf("%d! = %d ",num,fact);      return 0 ;  }  int factorial(int n)  {      int result = 1 ;      for(int i = 1 ; i <= n ; i++ )      {          result \*= i ;      }      return result ;  } | | | | | | | |
| **Sample Input:** | | | | | | | |
| *Enter a number : 5* | | | | | | | |
| **Sample Output:** | | | | | | | |
| *5! = 120* | | | | | | | |
| ***Time Complexity calculation :*** | | | | | | | |
| *Here the loop iterates from 1 to n, performing a constant amount of multiplication in each iteration. Therefore, the time complexity is proportional to the value of n.*  *Time Complexity = O(n)* | | | | | | | |

|  |
| --- |
| **Code for example 2:** |
| /\* Linear Search. \*/  #include<stdio.h>  int linear\_search(int arr[], int n, int key);  int main()   {      int num;      printf("Enter size of array : ");      scanf("%d",&num);      int arr[num];      printf("Enter the elements of array : ");      for(int i = 0 ; i < num;i++ )      {          scanf("%d",&arr[i]);      }      int n = sizeof(arr) / sizeof(arr[0]);      int key = 0;      printf("Enter the key : ");      scanf("%d",&key);      int index = linear\_search(arr, n, key);      if (index != -1) {          printf("Element %d found at index %d\n", key, index);      } else {          printf("Element %d not found\n", key);      }      return 0;  }  int linear\_search(int arr[], int n, int key) {      for (int i = 0; i < n; i++) {          if (arr[i] == key) {              return i;          }      }      return -1; // Element not found  } |
| **Sample Input:** |
| *Enter size of array : 6*  *Enter the elements of array : 2 3 5 7 9 11*  *Enter the key : 7* |
| **Sample Output:** |
| *Element 7 found at index 3* |
| **Time complexity calculation:** |
| *Here the algorithm iterates through the array elements sequentially, checking each element against the given key. In the worst case, the element being searched for may be located at the end of the array or may not be present at all. In this case, the algorithm would iterate through all n elements in the array.*  *Time Complexity : O(n)* |

|  |
| --- |
| **Modularity** |
| **Type of Algorithm:** *Recursive* | |
| **Details of the algorithm:** | |
| *A recursive algorithm is an algorithm that solves a problem by solving smaller instances of the same problem. In other words, it calls itself to break down a larger problem into smaller, more manageable subproblems until it reaches a base case where the problem can be directly solved. The name "recursive" comes from the concept of recursion, which is the process of a function or algorithm calling itself.*  *STEPS :*  *1.Base case(s): Define one or more base cases that represent the simplest form of the problem and provide a termination condition for the recursion. The base case(s) ensure that the recursion eventually ends.*  *2.Recursive case: Identify the recursive case(s) where the algorithm calls itself with a smaller or modified version of the original problem. This step breaks down the problem into smaller subproblems.*  *3.Parameters and arguments: Determine the necessary parameters or arguments to pass to the recursive function during each recursive call. These parameters typically represent the modified subproblem.*  *4.Recursion: Call the same algorithm recursively with the modified subproblem until the base case is reached.*  *5.Combine or process: If necessary, combine or process the results obtained from the recursive calls to solve the original problem.*  *6.Return value: Return the final result or accumulate the results obtained from recursive calls to the calling function.* | |
| **Code for example 1:** | |
| /\* Fibonacci Sequence. \*/  #include<stdio.h>  int fibonacci(int n) {      if (n <= 1) {          return n;      }      return fibonacci(n - 1) + fibonacci(n - 2);  }  int main()  {      int num ;      printf("Enter a number : ");      scanf("%d", &num);      int fib = fibonacci(num);      printf("Fibonacci number at position %d is %d\n", num, fib);      return 0;  } | |
| **Sample Input:** | |
| *Enter a number : 7* | |
| **Sample Output:** | |
| *Fibonacci number at position 7 is 13* | |
| **Time complexity calculation:** | |
| *In this example,* *the recursive function calls itself twice for each value of n greater than 1. As a result, the number of recursive function calls grows exponentially with the input size.*  *Time Complexity : O(2^n)* | |

|  |
| --- |
| **Code for example 2:** |
| /\* Binary Search. \*/  #include <stdio.h>  int binary\_search(int arr[], int low, int high, int target);  int main(){      int num;      printf("Enter the size of array : ");      scanf("%d", &num);      int arr[num];      printf("Enter the elements of array : ");      for (int i = 0; i < num; i++){          scanf("%d", &arr[i]);      }      int n = sizeof(arr) / sizeof(arr[0]);      int target;      printf("Enter target : ");      scanf("%d", &target);      int index = binary\_search(arr, 0, n - 1, target);      if (index != -1){          printf("Element found at index %d\n", index);      }      else{          printf("Element not found\n");      }      return 0;  }  int binary\_search(int arr[], int low, int high, int target){      if (low <= high){          int mid = low + (high - low) / 2;          if (arr[mid] == target){              return mid; // Return the index if found          }          else if (arr[mid] > target){              return binary\_search(arr, low, mid - 1, target); // Search in the left half          }          else{              return binary\_search(arr, mid + 1, high, target); // Search in the right half          }      }      return -1; // Return -1 if not found  } |
| **Sample Input:** |
| *Enter the size of array : 5*  *Enter the elements of array : 1 2 4 8 9*  *Enter target : 8* |
| **Sample Output:** |
| *Element found at index 3* |
| **Time complexity calculation:** |
| *In this example, the binary\_search function is recursively called to search for the target element in a sorted array. At each step, the search space is divided in half by calculating the middle index (mid) of the array. The target element is compared with the middle element to determine whether to search in the left half or the right half of the array.* |

|  |
| --- |
| **Modularity** |
| **Type of Algorithm:** *Back tracking* | |
| **Details of the algorithm:** | |
| *A backtracking algorithm is an algorithmic technique that systematically explores all possible solutions to a problem by incrementally building a solution candidate and then "backtracking" or undoing certain steps when they are found to be incorrect or not leading to a valid solution. It is named "backtracking" because the algorithm keeps track of its steps and "backtracks" or reverses certain decisions when necessary.*  *Backtracking algorithms are used in problems that can be formulated as finding a solution from a set of choices, where some choices lead to dead ends or invalid solutions. They are particularly useful for solving constraint satisfaction problems, puzzles, and combinatorial optimization problems.*  *STEPS:*  *1.Decision space: Define the space of possible decisions or choices at each step. This involves identifying the variables or parameters that define the problem and the constraints or conditions that guide the decision-making process.*  *2.Feasibility check: Perform a feasibility check to determine if the current partial solution satisfies the problem's constraints. If the partial solution is found to be invalid, backtrack to the previous decision point.*  *3.Base case: Define a base case that determines when a valid solution is found. This represents the end condition for the backtracking process.*  *4.Recursion: Make a decision and recursively explore the possible choices by moving to the next step. If the current choice leads to an invalid solution, backtrack to the previous step and try a different choice.*  *5.Pruning: Implement pruning techniques to optimize the search process by eliminating certain branches or paths that are guaranteed to lead to invalid or suboptimal solutions.*  *6.Backtrack: When a partial solution is found to be invalid or all choices have been exhausted, backtrack to the previous decision point and try a different choice or undo the previous decision.*  *7.Solution collection: Collect and store valid solutions as they are found.* | |
| **Code for example 1:** | |
| #include <stdio.h>  int isSubsetSum(int set[], int n, int target);  int main() {      int set[] = {3, 34, 4, 12, 5, 2};      int n = sizeof(set) / sizeof(set[0]);      int target = 9;      if (isSubsetSum(set, n, target))          printf("Subset with the given sum exists\n");      else          printf("No subset with the given sum exists\n");      return 0;  }  int isSubsetSum(int set[], int n, int target) {      if (target == 0)          return 1;      if (n == 0)          return 0;      if (set[n - 1] > target)          return isSubsetSum(set, n - 1, target);      return isSubsetSum(set, n - 1, target) || isSubsetSum(set, n - 1, target - set[n - 1]);  } | |
| **Sample Input:** | |
| *Sample Input:*  *Set = {3, 34, 4, 12, 5, 2}*  *Target = 9* | |
| **Sample Output:** | |
| *Sample Output:*  *Subset with the given sum exists* | |
| **Time complexity calculation:** | |
| *Here the algorithm explores all possible subsets of the given set, and the number of subsets is exponential in the size of the set.*  *Time Complexity : O(2^n)* | |

|  |
| --- |
| **Code for example 2:** |
| /\* N-Queens Problem \*/  #include <stdio.h>  #define N 4  void printSolution(int board[N][N]) {      for (int i = 0; i < N; i++) {          for (int j = 0; j < N; j++) {              printf("%d ", board[i][j]);          }          printf("\n");      }      printf("\n");  }  int isSafe(int board[N][N], int row, int col) {      int i, j;      // Check left side of the current row      for (i = 0; i < col; i++) {          if (board[row][i]) {              return 0;          }      }      // Check upper diagonal on the left side      for (i = row, j = col; i >= 0 && j >= 0; i--, j--) {          if (board[i][j]) {              return 0;          }      }      // Check lower diagonal on the left side      for (i = row, j = col; j >= 0 && i < N; i++, j--) {          if (board[i][j]) {              return 0;          }      }      return 1;  }  int solveNQueensUtil(int board[N][N], int col) {      if (col == N) {          printSolution(board);          return 1;      }      int res = 0;      for (int i = 0; i < N; i++) {          if (isSafe(board, i, col)) {              board[i][col] = 1;              res += solveNQueensUtil(board, col + 1);              board[i][col] = 0;          }      }      return res;  }  void solveNQueens() {      int board[N][N] = {0};      int res = solveNQueensUtil(board, 0);      if (res == 0) {          printf("No solution exists.\n");      }  }  int main() {      solveNQueens();      return 0;  } |
| **Sample Input:** |
| *Sample Input : N = 4* |
| **Sample Output:** |
| *Sample Output :*  *0 0 1 0*  *1 0 0 0*  *0 0 0 1*  *0 1 0 0*  *0 1 0 0*  *0 0 0 1*  *1 0 0 0*  *0 0 1 0* |
| **Time complexity calculation:** |
| *In the N-Queens problem, the goal is to place N queens on an NxN chessboard such that no two queens threaten each other. Specifically, no two queens should share the same row, column, or diagonal.* |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Modularity** |  | **Documentation** |  | **Indentation** |  | **Programming practices** |  |
| **Type of Algorithm:** *Divide and conquer* | | | | | | | |
| **Details of the algorithm:** | | | | | | | |
| *A divide and conquer algorithm is a problem-solving technique that involves breaking down a problem into smaller subproblems, solving each subproblem independently, and then combining the results to obtain a solution to the original problem. The name "divide and conquer" reflects the approach of dividing a complex problem into smaller, more manageable parts, conquering them individually, and merging the solutions to solve the overall problem.*  *Divide and conquer algorithms are particularly useful when the problem can be divided into independent subproblems that can be solved recursively. They are often used in problems that exhibit a recursive structure or can be naturally decomposed into smaller, similar instances.*  *STEPS :*  *1.Divide: Divide the problem into smaller subproblems that are similar to the original problem. This step involves breaking down the problem into smaller instances or subsets.*  *2.Conquer: Solve each subproblem independently. This step involves recursively applying the same divide and conquer algorithm to solve the smaller subproblems. If the subproblems are small enough, they are solved using a base case.*  *3.Combine: Combine the solutions of the subproblems to obtain the final solution to the original problem. This step involves merging or integrating the individual solutions obtained from the conquered subproblems.*  *4.Base case: Define a base case that specifies the simplest form of the problem that can be solved directly without further division. This base case ensures that the recursion eventually terminates.* | | | | | | | |
| **Code for example 1:** | | | | | | | |
| /\* Merge Sort.\*/  #include<stdio.h>  void printArray(int A[], int n)  {      for (int i = 0; i < n; i++)          printf("%d ", A[i]);        printf("\n");  }  void merge(int A[], int mid, int low, int high)  {      int i, j, k, B[100];      i = low;      j = mid + 1;      k = low;      while (i <= mid && j <= high)      {          if (A[i] < A[j])              B[k++] = A[i++];          else              B[k++] = A[j++];        }      while (i <= mid)          B[k++] = A[i++];        while (j <= high)          B[k++] = A[j++];        for (int i = low; i <= high; i++)          A[i] = B[i];  }  void mergeSort(int A[], int low, int high){      int mid;      if(low<high){          mid = (low + high) /2;          mergeSort(A, low, mid);          mergeSort(A, mid+1, high);          merge(A, mid, low, high);      }  }  int main()  {      int a[100],n;      printf("Enter the size: ");      scanf("%d",&n);      printf("Enter Numbers: ");      for(int i=0;i<n;i++)          a[i]=(rand()%10);        printArray(a, n);      mergeSort(a, 0, n-1);      printf("Sorted array is: ");      printArray(a, n);    } | | | | | | | |
| **Sample Input:** | | | | | | | |
| *Sample Input :*  *Enter the size: 5* | | | | | | | |
| **Sample Output:** | | | | | | | |
| *Sample Output :*  *Enter Numbers: 1 7 4 0 9*  *Sorted array is: 0 1 4 7 9* | | | | | | | |
| **Time complexity calculation:** | | | | | | | |
| *The time complexity of Merge Sort is O(n log n) in the average and worst cases. This is because the algorithm divides the array into halves at each step, and the merging step takes linear time. Hence, the time complexity can be approximated as O(n log n), where n is the size of the array.* | | | | | | | |

|  |
| --- |
| **Code for example 2:** |
| /\* Quick sort.\*/  #include<stdio.h>  #include<stdlib.h>  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);  }  void quickSort(int arr[], int low, int high) {      if (low < high) {          int pi = partition(arr, low, high);          quickSort(arr, low, pi - 1);          quickSort(arr, pi + 1, high);      }  }  int main() {      int n;      printf("Enter the size: ");      scanf("%d",&n);      int arr[n];      printf("Enter Numbers: ");      for(int i=0;i<n;i++){          arr[i]=rand()%50;          printf("%d ", arr[i]);      }      quickSort(arr, 0, n - 1);      printf("\nSorted array:");      for (int i = 0; i < n; i++) {          printf("%d ", arr[i]);      }      return 0;  } |
| **Sample Input:** |
| *Sample Input :*  *Enter the size: 5* |
| **Sample Output:** |
| *Sample Output :*  *Enter Numbers: 41 17 34 0 19*  *Sorted array:0 17 19 34 41* |
| **Time complexity calculation:** |
| *The time complexity of Quick Sort is O(n log n) in the average and best cases, and O(n^2) in the worst case. This is because the algorithm divides the array into subarrays based on a pivot element and recursively sorts the subarrays.* |

|  |
| --- |
| **Modularity** |
| **Type of Algorithm:** *Dynamic programming* | |
| **Details of the algorithm:** | |
| *Dynamic programming is a problem-solving technique used to solve complex problems by breaking them down into overlapping subproblems and solving each subproblem only once.*  *The word "dynamic" refers to the ability to make decisions based on changing circumstances or states.*  *STEPS:*  *1.Identify the subproblems: Analyze the problem and identify the overlapping subproblems that can be solved independently.*  *2.Define the recurrence relation: Express the solution to each subproblem in terms of solutions to smaller subproblems. This is usually done using a recursive equation or formula.*  *3.Formulate the base case: Identify the simplest subproblems that can be solved directly without any further recursion. These base cases provide the initial values for the dynamic programming table.*  *4.Build the table: Create a dynamic programming table or memoization array to store the intermediate results of subproblems. This table is populated in a bottom-up or top-down manner.*  *5.Fill in the table: Solve the subproblems iteratively by computing their solutions using the recurrence relation and populating the dynamic programming table.*  *6.Extract the final solution: Once the table is filled, extract the final solution to the original problem from the values stored in the dynamic programming table.* | |
| **Code for example 1:** | |
| /\* Fibonacci series numbers.\*/  #include <stdio.h>  #define MAX 100  #define NIL -1  int lookup[MAX];  void initialize() {      for (int i = 0; i < MAX; i++) {          lookup[i] = NIL;      }  }  int fibonacci(int n) {      if (lookup[n] == NIL) {          if (n <= 1)              lookup[n] = n;          else              lookup[n] = fibonacci(n - 1) + fibonacci(n - 2);      }      return lookup[n];  }  int main() {      int n ;      printf("Enter a number : ");      scanf("%d",&n);      initialize();      printf("Fibonacci Series up to %d terms: ", n);      for (int i = 0; i < n; i++) {          printf("%d ", fibonacci(i));      }      printf("\n");      return 0;  } | |
| **Sample Input:** | |
| *Sample Input : Enter a number : 10* | |
| **Sample Output:** | |
| *Sample Output :*  *Fibonacci Series up to 10 terms: 0 1 1 2 3 5 8 13 21 34* | |
| **Time complexity calculation:** | |
| *The time complexity of the Fibonacci algorithm using dynamic programming (memoization) is O(n). This is because the algorithm computes and stores the Fibonacci numbers from 0 to n in the lookup table. Each number is computed only once, and subsequent calculations for the same number are avoided by reusing the stored value. Therefore, the time complexity is linear with respect to the input size n.* | |

|  |
| --- |
| **Code for example 2:** |
| /\* Maximum subarray sum. \*/  #include <stdio.h>  int maxSubarraySum(int arr[], int n) {      int currentSum = arr[0];      int maxSum = arr[0];      for (int i = 1; i < n; i++) {          currentSum = (currentSum + arr[i] > arr[i]) ? currentSum + arr[i] : arr[i];          maxSum = (currentSum > maxSum) ? currentSum : maxSum;      }      return maxSum;  }  int main() {      int size;      printf("Enter the size of array : ");      scanf("%d",&size);      int arr[size];      printf("Enter the elements of array : ");      for(int i=0;i<size;i++)      {          scanf("%d",&arr[i]);      }      int n = sizeof(arr) / sizeof(arr[0]);      int maxSum = maxSubarraySum(arr, n);      printf("Maximum subarray sum: %d\n", maxSum);      return 0;  } |
| **Sample Input:** |
| *Enter the size of array : 9*  *Enter the elements of array : -2 1 -3 4 -1 2 1 -5 4* |
| **Sample Output:** |
| *Maximum subarray sum: 6* |
| **Time complexity calculation:** |
| *The time complexity of the Maximum Subarray Sum algorithm using dynamic programming is O(n), where n is the size of the array. The algorithm uses a single loop to iterate over the array elements, maintaining the current sum and the maximum sum seen so far. For each element, it compares whether adding the element to the current sum increases the current sum or starts a new subarray. It also updates the maximum sum if the current sum is greater. Since the algorithm iterates through the array once, the time complexity is linear with respect to the input size n.* |

|  |
| --- |
| **Modularity** |
| **Type of Algorithm:** *Greedy* | |
| **Details of the algorithm:** | |
| *Greedy algorithm is a problem-solving approach that makes locally optimal choices at each step with the aim of finding a global optimum. The name "greedy" comes from the idea that the algorithm always makes the choice that appears to be the best at the current moment without considering the potential consequences of that choice in the future.*  *Greedy algorithms are used in a wide range of applications, especially in optimization problems where an optimal solution is desired.*  *STEPS :*  *1.Define the problem: Clearly define the problem and the objective to be optimized or achieved.*  *2.Identify the choices: Determine the available choices or options at each step of the algorithm.*  *3.Determine the criteria: Establish a criterion or scoring function to evaluate the quality or value of each choice.*  *4.Select the best choice: At each step, select the choice that maximizes or minimizes the criterion value based on the greedy heuristic.*  *5.Update the solution: Incorporate the selected choice into the current solution.*  *6.Repeat: Repeat steps 2-5 until the problem is solved or a termination condition is met.*  *7.Validate the solution: Validate the final solution to ensure it meets the desired objectives or constraints.* | |
| **Code for example 1:** | |
| /\* Coin change problem.\*/  #include <stdio.h>  int coinChange(int coins[], int n, int amount) {      int count = 0;      for (int i = n - 1; i >= 0; i--) {          while (amount >= coins[i]) {              amount -= coins[i];              count++;          }      }      return count;  }  int main() {      int coins[] = {1, 5, 10, 25};      int n = sizeof(coins) / sizeof(coins[0]);      int amount = 47;      int result = coinChange(coins, n, amount);      printf("Minimum number of coins needed: %d\n", result);      return 0;  } | |
| **Sample Input:** | |
| *coins = {1, 5, 10, 25}*  *amount = 47* | |
| **Sample Output:** | |
| *Minimum number of coins needed: 5* | |
| **Time complexity calculation:** | |
| *The time complexity of this greedy algorithm is O(n), where n is the number of coin denominations. Since the number of coins required at each step is limited by the largest coin denomination, the algorithm runs in linear time* | |

|  |
| --- |
| **Code for example 2:** |
| #include <stdio.h>  int maxSumNonAdjacent(int arr[], int n) {      if (n == 0) {          return 0;      } else if (n == 1) {          return arr[0];      } else if (n == 2) {          return (arr[0] > arr[1]) ? arr[0] : arr[1];      }      int incl = arr[0];      int excl = 0;      for (int i = 1; i < n; i++) {          int temp = incl;          incl = (excl + arr[i] > incl) ? (excl + arr[i]) : incl;          excl = temp;      }      return (incl > excl) ? incl : excl;  }  int main() {      int arr[] = {4, 1, 1, 9, 5, 2};      int n = sizeof(arr) / sizeof(arr[0]);      int result = maxSumNonAdjacent(arr, n);      printf("Maximum sum of non-adjacent elements: %d\n", result);      return 0;  } |
| **Sample Input:** |
| *arr = {4, 1, 1, 9, 5, 2}* |
| **Sample Output:** |
| *Maximum sum of non-adjacent elements: 15* |
| **Time complexity calculation:** |
| *The time complexity of this greedy algorithm is O(n), where n is the number of elements in the array. The algorithm traverses the array once, calculating the maximum sum at each step. Therefore, it has a linear time complexity.* |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **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 by systematically exploring the search space and bounding the solutions.*  *STEPS :*  *1.* *Branching: It involves dividing the problem into smaller subproblems or branches. At each branching step, the algorithm explores different possibilities or choices that can lead to potential solutions. This process creates a tree-like structure, often called the search tree or state space tree.*  *2.Bounding: It involves establishing bounds or criteria to prune branches that are not promising or cannot lead to optimal solutions. The algorithm maintains upper and lower bounds on the optimal solution and uses them to determine which branches to explore further and which ones can be pruned.* | | | | | | | |
| **Code for example 1:** | | | | | | | |
| /\* Permutation problem .\*/  #include <stdio.h>  void swap(int\* a, int\* b) {      int temp = \*a;      \*a = \*b;      \*b = temp;  }  void permute(int arr[], int start, int end) {      if (start == end) {          for (int i = 0; i <= end; i++) {              printf("%d ", arr[i]);          }          printf("\n");      } else {          for (int i = start; i <= end; i++) {              swap(&arr[start], &arr[i]);              permute(arr, start + 1, end);              swap(&arr[start], &arr[i]); // backtrack          }      }  }  int main() {      int num;      printf("Enter the size of array : ");      scanf("%d",&num);      int arr[num];      printf("Enter the elements of array : ");      for (int i =0;i<num;i++)      {          scanf("%d",&arr[i]);      }      int n = sizeof(arr) / sizeof(arr[0]);      permute(arr, 0, n - 1);      return 0;  } | | | | | | | |
| **Sample Input:** | | | | | | | |
| *Enter the size of array : 3*  *Enter the elements of array : 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:** | | | | | | | |
| *The time complexity of the Branch and Bound algorithm for the Permutation Problem is factorial, as it generates all possible permutations. It is typically O(n!), where n is the number of elements in the set.* | | | | | | | |

|  |
| --- |
| **Code for example 2:** |
| /\* Knapsack Problem.\*/  #include <stdio.h>  #define MAX\_ITEMS 5  int max(int a, int b) {      return (a > b) ? a : b;  }  int knapsack(int values[], int weights[], int capacity, int n) {      if (n == 0 || capacity == 0)          return 0;      if (weights[n - 1] > capacity)          return knapsack(values, weights, capacity, n - 1);      int included = values[n - 1] + knapsack(values, weights, capacity - weights[n - 1], n - 1);      int excluded = knapsack(values, weights, capacity, n - 1);      return max(included, excluded);  }  int main() {      int values[MAX\_ITEMS] ;      printf("Enter values : ");      for(int i = 0;i<MAX\_ITEMS;i++)      {          scanf("%d",&values[i]);      }      int weights[MAX\_ITEMS] ;      printf("Enter weights : ");      for(int j = 0;j<MAX\_ITEMS;j++)      {          scanf("%d",&weights[j]);      }      int capacity = 100;      int maxValue = knapsack(values, weights, capacity, MAX\_ITEMS);      printf("Maximum value: %d\n", maxValue);      return 0;  } |
| **Sample Input:** |
| *Enter values : 60 100 120 40 70*  *Enter weights : 10 20 30 40 50* |
| **Sample Output:** |
| *Maximum value: 320* |
| **Time complexity calculation:** |
| *The time complexity of the Branch and Bound algorithm for the Knapsack Problem is exponential. It is typically O(2^n), where n is the number of items.* |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Modularity** |  | **Documentation** |  | **Indentation** |  | **Programming practices** |  |
| **Type of Algorithm:** *Brute force* | | | | | | | |
| **Details of the algorithm:** | | | | | | | |
| *Brute force algorithm is a straightforward problem-solving technique that involves trying all possible solutions to find the desired outcome. It gets its name from the approach of "brute-forcing" through every possible combination or permutation until the solution is found.*  *The steps involved in a brute force algorithm are as follows:*  *1.Identify the problem: Understand the problem and determine if a brute force approach is suitable.*  *2.Define the search space: Identify the range of possible solutions and determine the parameters that define each solution.*  *3.Generate all possible solutions: Use nested loops, recursion, or other constructs to systematically generate or iterate through all possible solutions.*  *4.Evaluate each solution: Apply the problem constraints or criteria to each solution and determine if it satisfies the desired conditions.*  *5.Analyze the results: Depending on the problem, you may need to analyze or compare the solutions to identify the best or optimal one.*  *6.Implement optimization: If the brute force approach is inefficient or impractical, consider implementing optimizations or heuristics to narrow down the search space or reduce the number of solutions to be evaluated.*  *7.Validate the solution: Finally, validate and verify the output of the algorithm to ensure it meets the desired requirements or objectives.* | | | | | | | |
| **Code for example 1:** | | | | | | | |
| /\* Password strength checker.\*/  #include <stdio.h>  #include <stdbool.h>  #include <string.h>  bool checkPassword(char\* password) {      char target[] = "kletech@2023";      if (strcmp(password, target) == 0) {          return true;      } else {          return false;      }  }  void bruteForcePassword() {      char password[50];      printf("Enter the password: ");      scanf("%s", password);      if (checkPassword(password)) {          printf("Password accepted\n");      } else {          printf("Incorrect password\n");      }  }  int main() {      bruteForcePassword();      return 0;  } | | | | | | | |
| **Sample Input:** | | | | | | | |
| *Enter the password: kletech@2023* | | | | | | | |
| **Sample Output:** | | | | | | | |
| *Password accepted* | | | | | | | |
| **Time complexity calculation:** | | | | | | | |
| *The time complexity of the brute force password checking algorithm is O(1), as it simply compares the entered password with the target password.* | | | | | | | |

|  |
| --- |
| **Code for example 2:** |
| /\* String permutation.\*/  #include <stdio.h>  #include <string.h>  void swap(char\* a, char\* b) {      char temp = \*a;      \*a = \*b;      \*b = temp;  }  void permute(char\* str, int start, int end) {      if (start == end) {          printf("%s\n", str);          return;      }      for (int i = start; i <= end; i++) {          swap(&str[start], &str[i]);          permute(str, start + 1, end);          swap(&str[start], &str[i]); // backtrack      }  }  int main() {      char str[30] ;      printf("Enter a string : ");      gets(str);      //printf("%s",str);      int n = strlen(str);      printf("Permutations of the string:\n");      permute(str, 0, n - 1);      return 0;  } |
| **Sample Input:** |
| *Enter a string : ABC* |
| **Sample Output:** |
| *Permutations of the string:*  *ABC*  *ACB*  *BAC*  *BCA*  *CBA*  *CAB* |
| **Time complexity calculation:** |
| *The time complexity of the string permutation algorithm is O(n!), where n is the length of the input string. This is because there are n! (n factorial) possible permutations of a string of length n.* |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Modularity** |  | **Documentation** |  | **Indentation** |  | **Programming practices** |  |
| **Type of Algorithm:** *Randomized* | | | | | | | |
| **Details of the algorithm:** | | | | | | | |
| *A randomized algorithm is an algorithm that incorporates an element of randomness in its execution.* *The name "randomized" comes from the fact that these algorithms make use of randomization to achieve their goals.*  *The steps involved in a randomized algorithm can vary depending on the specific problem and approach. However, some common steps include:*  *1.Initialize random seed: Set the seed for the random number generator to ensure reproducibility or randomness in subsequent random choices.*  *2.Randomized actions: Perform actions or make decisions based on randomly generated numbers. This can involve random sampling, random selection, or random perturbations.*  *3.Repeat or iterate: Repeat the randomized actions for a specified number of iterations or until a certain condition is met.*  *4.Analyze the outcomes: Analyze the outcomes of the randomized algorithm, which can include statistical analysis, estimating probabilities, or evaluating the quality of the solution.*  *5.Repeat or refine: If the initial outcome does not meet the desired criteria, repeat the randomized algorithm with different random choices or parameters to improve the result.* | | | | | | | |
| **Code for example 1:** | | | | | | | |
| /\* Randomized shuffle. \*/  #include <stdio.h>  #include <stdlib.h>  #include <time.h>  void swap(int\* a, int\* b) {      int temp = \*a;      \*a = \*b;      \*b = temp;  }  void randomizedShuffle(int arr[], int n) {      srand(time(NULL));      for (int i = n - 1; i > 0; i--) {          int j = rand() % (i + 1);          swap(&arr[i], &arr[j]);      }  }  void printArray(int arr[], int n) {      for (int i = 0; i < n; i++) {          printf("%d ", arr[i]);      }      printf("\n");  }  int main() {      int arr[] = {1, 2, 3, 4, 5};      int n = sizeof(arr) / sizeof(arr[0]);      printf("Original array: ");      printArray(arr, n);      randomizedShuffle(arr, n);      printf("Shuffled array: ");      printArray(arr, n);      return 0;  } | | | | | | | |
| **Sample Input:** | | | | | | | |
| *arr[] = { 1,2, 3, 4,5}* | | | | | | | |
| **Sample Output:** | | | | | | | |
| *Original array: 1 2 3 4 5*  *Shuffled array: 3 4 1 2 5* | | | | | | | |
| **Time complexity calculation:** | | | | | | | |
| *The time complexity of the randomized shuffle algorithm is O(n), where n is the number of elements in the array. This is because each element is swapped with another randomly chosen element exactly once.* | | | | | | | |

|  |
| --- |
| **Code for example 2:** |
| /\*Randomized Prime Number Generation.\*/  #include <stdio.h>  #include <stdlib.h>  #include <stdbool.h>  #include <time.h>  bool isPrime(int num) {      if (num <= 1)          return false;      for (int i = 2; i \* i <= num; i++) {          if (num % i == 0)              return false;      }      return true;  }  int generateRandomPrime(int lower, int upper) {      srand(time(NULL));      while (true) {          int num = (rand() % (upper - lower + 1)) + lower;          if (isPrime(num))              return num;      }  }  int main() {      int lower=10;      int upper=50;      // printf("Enter lower = ");      // scanf("%d",&lower);      // printf("Enter upper = ");      // scanf("%d",&upper);      int prime = generateRandomPrime(lower, upper);      printf("Random prime number between %d and %d: %d\n", lower, upper, prime);      return 0;  } |
| **Sample Input:** |
| *Lower = 10*  *Upper = 50* |
| **Sample Output:** |
| *Random prime number between 10 and 50: 29* |
| **Time complexity calculation:** |
| *The time complexity of the randomized prime number generation algorithm depends on the range of numbers and the probability of generating a prime number. In the worst case, it can be considered as O(n \* sqrt(n)), where n is the range of numbers. However, the actual time complexity can vary depending on the distribution of prime numbers within the range and the efficiency of the isPrime function.* |