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

**Term-work Report**

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Term-work** | *01* | | | | |  |  | | | | |
| **Student Name** | Amogh | | | | |  |  | | | | |
| **SRN** | 01FE21BEC122 | | | | **Roll Number** | | 316 | | **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)* | | | |
| **2** | | arrays | | | | | O(n). | | | |
| Recursive | **1** | | arrays | | | | | *O(2^n)* | | | |
| **2** | | arrays | | | | | *O(n + log n)* | | | |
| Back tracking | **1** | | arrays | | | | | *O(N^2)* | | | |
| **2** | | arrays | | | | | *O(n + 2^n)* | | | |
| Divide and conquer | **1** | | arrays | | | | | *O(n log n)* | | | |
| **2** | | arrays | | | | | *O(N)* | | | |
| Dynamic programming | **1** | | strings | | | | | *O(mn)* | | | |
| **2** | | arrays | | | | | *O(n)* | | | |
| Greedy | **1** | | arrays | | | | | O(n) = n | | | |
| **2** | | Linked list | | | | | O(n) = nlogn | | | |
| Branch and bound | **1** | | arrays | | | | | *O(n^2)* | | | |
| **2** | | arrays | | | | | *O(n\*2^n)* | | | |
| Brute force | **1** | | strings | | | | | *O(pq)* | | | |
| **2** | | arrays | | | | | *O(n^2)* | | | |
| Randomized | **1** | | arrays | | | | | *O(n^2)* | | | |
| **2** | | arrays | | | | | *O(n^2)* | | | |

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?

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **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>*  *void swap(int \*a, int \*b)*  *{*  *int temp = \*a;*  *\*a = \*b;*  *\*b = temp;*  *}*  *void Sort(int a[], int n)*  *{*  *int i, j;*  *for (i = 0; i < n-1; i++)*  *// Last i elements are already in place*  *for (j = 0; j < n-i-1; j++)*  *if (a[j] > a[j+1])*  *swap(&a[j], &a[j+1]);*  *}*  *int main()*  *{*  *int a[100], n, i;*  *printf("Enter the number of elements in the array: ");*  *scanf("%d", &n);*  *printf("Enter the elements of the array:\n");*  *for (i = 0; i < n; i++)*  *scanf("%d", &arr[i]);*  *Sort(a, n);*  *printf("Sorted array:\n");*  *for (i = 0; i < n; i++)*  *printf("%d ", a[i]);*  *return 0;* | | | | | | | |
| **Sample Input:** | | | | | | | |
| *1,2333,4,1,23* | | | | | | | |
| **Sample Output:** | | | | | | | |
| *1,1,4,23,2333* | | | | | | | |
| **Time complexity calculation:** | | | | | | | |
| *O(n^2)* | | | | | | | |

|  |
| --- |
| **Code for example 2:** |
| *#include <stdio.h>*  *#define MAX\_SIZE 100*  *int search(int a[], int n, int x) {*  *for (int i = 0; i < n; i++) {*  *if (a[i] == x) {*  *return i;*  *}*  *}*  *return -1;*  *}*  *int main()*  *{*  *int a[MAX\_SIZE];*  *int n, x;*  *printf("Enter the size of the array: ");*  *scanf("%d", &n);*  *printf("Enter the elements of the array:\n");*  *for (int i = 0; i < n; i++) {*  *scanf("%d", &a[i]);*  *}*  *printf("Enter the element to be searched: ");*  *scanf("%d", &x);*  *int index = search(a, n, x);*  *if (index == -1) {*  *printf("Element not found");*  *} else {*  *printf("Element found at index %d", index);*  *}*  *return 0;*  *}* |
| **Sample Input:** |
| *1,2333,43,232,14* |
| **Sample Output:** |
| *1* |
| **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>  int fibonacci(int n);  int main()  {  int n;  printf("Enter the value of n: ");  scanf("%d", &n);  printf("Fibonacci series up to %d:\n", n);  for(int i = 0; i <= n; i++) {  printf("%d ", fibonacci(i));  }  return 0;  }  int fibonacci(int n)  {  if(n == 0) {  return 0;  } else if(n == 1) {  return 1;  } else {  return fibonacci(n - 1) + fibonacci(n - 2);  }  } | | | | | | | |
| **Sample Input:** | | | | | | | |
| *5* | | | | | | | |
| **Sample Output:** | | | | | | | |
| *0 1 1 2 3 5* | | | | | | | |
| **Time complexity calculation:** | | | | | | | |
| *O(2^n)* | | | | | | | |

|  |
| --- |
| **Code for example 2:** |
| *#include <stdio.h>*  *int search(int a[], int low, int high, int x)*  *{*  *if (low > high) {*  *return -1;*  *}*  *int mid = low + (high - low) / 2;*  *if (a[mid] == x) {*  *return mid;*  *}*  *else if (a[mid] > x) {*  *return search(a, low, mid - 1, x);*  *}*  *else {*  *return search(a, mid + 1, high, x);*  *}*  *}*  *int main()*  *{*  *int n, x;*  *printf("Enter the size of array: ");*  *scanf("%d", &n);*  *int arr[n];*  *printf("Enter the elements of array in sorted order:\n");*  *for (int i = 0; i < n; i++) {*  *scanf("%d", &a[i]);*  *}*  *printf("Enter the element to search: ");*  *scanf("%d", &x);*  *int result = search(a, 0, n - 1, x);*  *if (result == -1) {*  *printf("Element not found.\n");*  *}*  *else {*  *printf("Element found at index %d.\n", result);*  *}*  *return 0;*  *}* |
| **Sample Input:** |
| *1,233,12,22,14* |
| **Sample Output:** |
| *1* |
| **Time complexity calculation:** |
| *O(n + log n)* |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **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>  #define N 4  void prMaze(int maze[N][N]) {  printf("\nMaze:\n");  for (int i = 0; i < N; i++) {  for (int j = 0; j < N; j++) {  printf("%d ", maze[i][j]);  }  printf("\n");  }  }  int isMove(int maze[N][N], int row, int col) {  if (row >= 0 && row < N && col >= 0 && col < N && maze[row][col] == 1) {  return 1;  }  return 0;  }  int sMaze(int maze[N][N], int row, int col, int solution[N][N]) {  if (row == N - 1 && col == N - 1) {  solution[row][col] = 1;  return 1;  }  if (isMove(maze, row, col) == 1) {  solution[row][col] = 1;  if (sMaze(maze, row, col + 1, solution) == 1) {  return 1;  }  if (sMaze(maze, row + 1, col, solution) == 1) {  return 1;  }  solution[row][col] = 0;  return 0;  }  return 0;  }  int main() {  int maze[N][N] = {{1, 0, 0, 0},  {1, 1, 0, 1},  {0, 1, 0, 0},  {1, 1, 1, 1}};  int solution[N][N] = {0};  pMaze(maze);  if (sMaze(maze, 0, 0, solution) == 1) {  printf("\nSolution:\n");  pMaze(solution);  } else {  printf("\nNo solution found\n");  }  return 0;  } | | | | | | | |
| **Sample Input:** | | | | | | | |
| *1 0 0 0*  *1 1 0 1*  *0 1 0 0*  *1 1 1 1* | | | | | | | |
| **Sample Output:** | | | | | | | |
| *1 0 0 0*  *1 1 0 0*  *0 1 0 0*  *0 1 1 1* | | | | | | | |
| **Time complexity calculation:** | | | | | | | |
| *O(N^2)* | | | | | | | |

|  |
| --- |
| **Code for example 2:** |
| *#include <stdio.h>*  *#define MAX\_SIZE 100*  *int set[MAX\_SIZE];*  *int subset[MAX\_SIZE];*  *int n;*  *int sum;*  *int count = 0;*  *void print\_subset() {*  *printf("{ ");*  *for (int i = 0; i < count; i++) {*  *printf("%d ", subset[i]);*  *}*  *printf("}\n");*  *}*  *void back(int index, int curr\_sum) {*  *if (curr\_sum == sum) {*  *print\_subset();*  *return;*  *}*  *if (curr\_sum > sum || index == n) {*  *return;*  *}*  *subset[count++] = set[index];*  *back(index + 1, curr\_sum + set[index]);*  *count--;*  *back(index + 1, curr\_sum);*  *}*  *int main() {*  *printf("Enter the number of elements in the set: ");*  *scanf("%d", &n);*  *printf("Enter the set elements:\n");*  *for (int i = 0; i < n; i++) {*  *scanf("%d", &set[i]);*  *}*  *printf("Enter the desired sum: ");*  *scanf("%d", &sum);*  *printf("Subsets with sum %d are:\n", sum);*  *back(0, 0);*  *return 0;*  *}* |
| **Sample Input:** |
| *1*  *2*  *2*  *3*  *4* |
| **Sample Output:** |
| *{ 1 3 }*  *{ 2 2 }*  *{ 4 }* |
| **Time complexity calculation:** |
| *O(n + 2^n)* |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **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 merge(int arr[], int left[], int right[], int nL, int nR) {  int i = 0, j = 0, k = 0;  while (i < nL && j < nR) {  if (left[i] <= right[j]) {  arr[k] = left[i];  i++;  }  else {  arr[k] = right[j];  j++;  }  k++;  }  while (i < nL) {  arr[k] = left[i];  i++;  k++;  }  while (j < nR) {  arr[k] = right[j];  j++;  k++;  }  }  void mergesort(int arr[], int n) {  if (n < 2) {  return;  }  int mid = n / 2;  int left[mid], right[n - mid];  for (int i = 0; i < mid; i++) {  left[i] = arr[i];  }  for (int i = mid; i < n; i++) {  right[i - mid] = arr[i];  }  mergesort(left, mid);  mergesort(right, n - mid);  merge(arr, left, right, mid, n - mid);  }  void printArray(int arr[], int size) {  for (int i = 0; i < size; i++) {  printf("%d ", arr[i]);  }  printf("\n");  }  int main() {  int n;  printf("Enter the size of the array: ");  scanf("%d", &n);  int arr[n];  printf("Enter the elements of the array: ");  for (int i = 0; i < n; i++) {  scanf("%d", &arr[i]);  }  printf("Original array: ");  printArray(arr, n);  mergesort(arr, n);  printf("Sorted array: ");  printArray(arr, n);  return 0;  } | | | | | | | |
| **Sample Input:** | | | | | | | |
| *1*  *2*  *3*  *444*  *33* | | | | | | | |
| **Sample Output:** | | | | | | | |
| *Original array: 1 2 3 444 33*  *Sorted array: 1 2 3 33 444* | | | | | | | |
| **Time complexity calculation:** | | | | | | | |
| *O(n log n)* | | | | | | | |

|  |
| --- |
| **Code for example 2:** |
| *#include <stdio.h>*  *void swap(int\* a, int\* b) {*  *int t = \*a;*  *\*a = \*b;*  *\*b = t;*  *}*  *int part(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 sort(int arr[], int low, int high) {*  *if (low < high) {*  *int pi = part(arr, low, high);*  *sort(arr, low, pi - 1);*  *sort(arr, pi + 1, high);*  *}*  *}*  *void printArray(int arr[], int size) {*  *for (int i = 0; i < size; i++) {*  *printf("%d ", arr[i]);*  *}*  *printf("\n");*  *}*  *int main() {*  *int n;*  *printf("Enter the size of the array: ");*  *scanf("%d", &n);*  *int arr[n];*  *printf("Enter the elements of the array: ");*  *for (int i = 0; i < n; i++) {*  *scanf("%d", &arr[i]);*  *}*  *printf("Original array: ");*  *printArray(arr, n);*  *sort(arr, 0, n - 1);*  *printf("Sorted array: ");*  *printArray(arr, n);*  *return 0;*  *}* |
| **Sample Input:** |
| *1*  *2*  *33*  *4*  *2* |
| **Sample Output:** |
| *Original array: 1 2 33 4 2*  *Sorted array: 1 2 2 4 33* |
| **Time complexity calculation:** |
| *O(N)* |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **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 length(char\* s) {  int i = 0;  while (s[i] != '\0') {  i++;  }  return i;  }  int lcs(char\* s1, char\* s2) {  int m = length(s1);  int n = length(s2);  int dp[m+1][n+1];  for (int i = 0; i <= m; i++) {  for (int j = 0; j <= n; j++) {  if (i == 0 || j == 0) {  dp[i][j] = 0;  } else if (s1[i-1] == s2[j-1]) {  dp[i][j] = dp[i-1][j-1] + 1;  } else {  dp[i][j] = (dp[i-1][j] > dp[i][j-1]) ? dp[i-1][j] : dp[i][j-1];  }  }  }  return dp[m][n];  }  int main() {  char s1[100], s2[100];  printf("Enter the first string: ");  scanf("%s", s1);  printf("Enter the second string: ");  scanf("%s", s2);  int len = lcs(s1, s2);  printf("The length of the longest common subsequence is %d.\n", len);  return 0;  } | | | | | | | |
| **Sample Input:** | | | | | | | |
| *Enter the first string: raaaam*  *Enter the second string: ram* | | | | | | | |
| **Sample Output:** | | | | | | | |
| *The length of the longest common subsequence is 3.* | | | | | | | |
| **Time complexity calculation:** | | | | | | | |
| *O(mn)* | | | | | | | |

|  |
| --- |
| **Code for example 2:** |
| *#include <stdio.h>*  *int fibonacci(int n) {*  *int fib[n+1];*  *fib[0] = 0;*  *fib[1] = 1;*  *for (int i = 2; i <= n; i++) {*  *fib[i] = fib[i-1] + fib[i-2];*  *}*  *return fib[n];*  *}*  *int main() {*  *int n;*  *printf("Enter the number of terms of the Fibonacci sequence to be generated: ");*  *scanf("%d", &n);*  *printf("The Fibonacci sequence is:\n");*  *for (int i = 0; i < n; i++) {*  *printf("%d ", fibonacci(i));*  *}*  *printf("\n");*  *return 0;*  *}* |
| **Sample Input:** |
| *Enter the number of terms of the Fibonacci sequence to be generated: 5* |
| **Sample Output:** |
| *The Fibonacci sequence is:*  *0 1 1 2 3* |
| **Time complexity calculation:** |
| *O(n)* |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **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) = 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(n) = 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>  typedef struct Job {  int id;  int profit;  int deadline;  }node;  void swap(node \*a, node \*b) {  node temp = \*a;  \*a = \*b;  \*b = temp;  }  void bubbleSort(node a[], int n) {  for (int i = 0; i < n - 1; i++) {  for (int j = 0; j < n - i - 1; j++) {  if (a[j].profit < a[j + 1].profit) {  swap(&a[j], &a[j + 1]);  }  }  }  }  int MaxDeadline(node a[], int n) {  int maxDeadline = -1;  for (int i = 0; i < n; i++) {  if (a[i].deadline > maxDeadline) {  maxDeadline = a[i].deadline;  }  }  return maxDeadline;  }  int jobSequencing(node a[], int n) {  bubbleSort(a, n);  int maxDeadline = MaxDeadline(a, n);  int jobSequence[maxDeadline];  for (int i = 0; i < maxDeadline; i++) {  jobSequence[i] = -1;  }  int profit = 0;  for (int i = 0; i < n; i++) {  int j = a[i].deadline - 1;  while (j >= 0 && jobSequence[j] != -1) {  j--;  }  if (j >= 0) {  jobSequence[j] = a[i].id;  profit += a[i].profit;  }  }  return profit;  }  int main() {  int n;  printf("Enter the number of jobs: ");  scanf("%d", &n);  node a[n];  for (int i = 0; i < n; i++) {  printf("Enter the profit and deadline of job %d: ", i + 1);  scanf("%d %d", &a[i].profit, &a[i].deadline);  a[i].id = i + 1;  }  int maxProfit = jobSequencing(a, n);  printf("Maximum profit is %d", maxProfit);  return 0;  } | | | | | | | |
| **Sample Input:** | | | | | | | |
| *Enter the number of jobs: 5*  *Enter the profit and deadline of job 1: 12*  *122*  *Enter the profit and deadline of job 2: 12333*  *2123*  *Enter the profit and deadline of job 3: 123*  *1233*  *Enter the profit and deadline of job 4: 12312*  *12344*  *Enter the profit and deadline of job 5: 123321*  *122221* | | | | | | | |
| **Sample Output:** | | | | | | | |
| *Maximum profit is 148101* | | | | | | | |
| **Time complexity calculation:** | | | | | | | |
| *O(n^2)* | | | | | | | |

|  |
| --- |
| **Code for example 2:** |
| *#include <stdio.h>*  *float upper\_bound(int n, int w, float p[], float wt[], int i)*  *{*  *float u = p[i];*  *int j = i;*  *float x[n];*  *for (int k = 0; k < n; k++) {*  *x[k] = 0.0;*  *}*  *while (wt[j] <= w && j < n) {*  *x[j] = 1.0;*  *w -= wt[j];*  *u += p[j];*  *j++;*  *}*  *if (j < n) {*  *x[j] = w / wt[j];*  *u += p[j] \* x[j];*  *}*  *return u;*  *}*  *float knapsack(int n, int W, float p[], float wt[])*  *{*  *float maxprofit = 0.0;*  *int queue[1000];*  *int qft = 0, qbk = 0;*  *queue[qbk++] = -1;*  *float u = upper\_bound(n, W, p, wt, -1);*  *while (qft < qbk) {*  *int i = queue[qft++];*  *if (i == -1) {*  *if (qft == qbk) {*  *break;*  *}*  *queue[qbk++] = -1;*  *u = upper\_bound(n, W, p, wt, i);*  *continue;*  *}*  *float lb = 0.0;*  *float w = 0.0;*  *for (int j = 0; j <= i; j++) {*  *if (w + wt[j] <= W) {*  *lb += p[j];*  *w += wt[j];*  *}*  *else {*  *lb += (W - w) \* (p[j] / wt[j]);*  *break;*  *}*  *}*  *if (lb > maxprofit) {*  *maxprofit = lb;*  *}*  *if (u > maxprofit) {*  *queue[qbk++] = i + 1;*  *u = upper\_bound(n, W - wt[i + 1], p, wt, i + 1);*  *}*  *queue[qbk++] = i + 1;*  *}*  *return maxprofit;*  *}*  *int main()*  *{*  *int n, W;*  *printf("Enter the number of items: ");*  *scanf("%d", &n);*  *float p[n], wt[n];*  *printf("Enter the profits and weights of %d items:\n", n);*  *for (int i = 0; i < n; i++) {*  *scanf("%f%f", &p[i], &wt[i]);*  *}*  *printf("Enter the capacity of the knapsack: ");*  *scanf("%d", &W);*  *float maxprofit = knapsack(n, W, p, wt);*  *printf("The maximum profit that can be obtained is: %f\n", maxprofit);*  *return 0;*  *}* |
| **Sample Input:** |
| *<Sample Input>* |
| **Sample Output:** |
| *<Sample Output>* |
| **Time complexity calculation:** |
| *O(n\*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<string.h>  int strmat(int \*line,int \*pattern)  {  int p=strlen(line);  int q=strlen(pattern);  for (int i = 0; i <= p - q; i++) {  int k = i,j;  for (j = 0; j < q; j++) {  if (pattern[j] == line[k])  k++;  else  break;  }  if (j == q)  return 1;  }  return -1;  }  int main()  {  char line[1000],pattern[1000];  printf("enter the line: \n");  scanf("%s",line);  printf("enter the pattern: \n");  scanf("%s",pattern);  int temp=strmat(line,pattern);  if(temp==-1)  {  printf("the pattern is not matched\n");  }  else  {  printf("the pattern is not matched\n");  }  return 0;  } | | | | | | | |
| **Sample Input:** | | | | | | | |
| *enter the line:*  *ram*  *enter the pattern:*  *rm* | | | | | | | |
| **Sample Output:** | | | | | | | |
| *the pattern is not matched* | | | | | | | |
| **Time complexity calculation:** | | | | | | | |
| *O(pq)* | | | | | | | |

|  |
| --- |
| **Code for example 2:** |
| *#include <stdio.h>*  *void select\_sort(int a[], int n) {*  *int i, j, min;*  *for (i = 0; i < n-1; i++) {*  *min= i;*  *for (j = i+1; j < n; j++) {*  *if (a[j] < a[min]) {*  *min= j;*  *}*  *}*  *int temp = a[i];*  *a[i] = a[min];*  *a[min] = temp;*  *}*  *}*  *int main() {*  *int n;*  *printf("Enter the number of elements in the array: ");*  *scanf("%d", &n);*  *int a[n];*  *printf("Enter the elements of the array: ");*  *for (int i = 0; i < n; i++) {*  *scanf("%d", &a[i]);*  *}*  *select\_sort(a, n);*  *printf("Sorted array: ");*  *for (int i = 0; i < n; i++) {*  *printf("%d ", a[i]);*  *}*  *return 0;*  *}* |
| **Sample Input:** |
| *Enter the number of elements in the array: 5*  *Enter the elements of the array: 1*  *2*  *3*  *33*  *22* |
| **Sample Output:** |
| *Sorted array: 1 2 3 22 33* |
| **Time complexity calculation:** |
| *O(n^2)* |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Modularity** |  | **Documentation** |  | **Indentation** |  | **Programming practices** |  |
| **Type of Algorithm:** *Randomized* | | | | | | | |
| **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 <stdio.h>*  *#include <stdlib.h>*  *#include <time.h>*  *void swap(int\* a, int\* b) {*  *int temp = \*a;*  *\*a = \*b;*  *\*b = temp;*  *}*  *int partition(int arr[], int low, int high) {*  *int pivot\_idx = low + rand() % (high - low + 1);*  *int pivot = arr[pivot\_idx];*  *swap(&arr[pivot\_idx], &arr[high]);*  *int i = low - 1;*  *for (int j = low; j < high; 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 pivot\_idx = partition(arr, low, high);*  *quicksort(arr, low, pivot\_idx - 1);*  *quicksort(arr, pivot\_idx + 1, high);*  *}*  *}*  *int main() {*  *srand(time(NULL));*  *int n;*  *printf("Enter the size of the array: ");*  *scanf("%d", &n);*  *int arr[n];*  *printf("Enter the array elements:\n");*  *for (int i = 0; i < n; i++) {*  *scanf("%d", &arr[i]);*  *}*  *quicksort(arr, 0, n - 1);*  *printf("Sorted array:\n");*  *for (int i = 0; i < n; i++) {*  *printf("%d ", arr[i]);*  *}*  *printf("\n");*  *return 0;*  *}* | | | | | | | |
| **Sample Input:** | | | | | | | |
| *Enter the size of the array: 5*  *Enter the array elements:*  *1*  *23*  *2*  *1*  *444* | | | | | | | |
| **Sample Output:** | | | | | | | |
| *Sorted array:*  *1 1 2 23 444* | | | | | | | |
| **Time complexity calculation:** | | | | | | | |
| *O(n^2)* | | | | | | | |

|  |
| --- |
| **Code for example 2:** |
| *#include <stdio.h>*  *int partition(int arr[], int low, int high, int pivot\_index) {*  *int pivot\_value = arr[pivot\_index];*  *int temp = arr[high];*  *arr[high] = arr[pivot\_index];*  *arr[pivot\_index] = temp;*  *int i = low - 1;*  *for (int j = low; j < high; j++) {*  *if (arr[j] < pivot\_value) {*  *i++;*  *temp = arr[i];*  *arr[i] = arr[j];*  *arr[j] = temp;*  *}*  *}*  *i++;*  *temp = arr[i];*  *arr[i] = arr[high];*  *arr[high] = temp;*  *return i;*  *}*  *int randomized\_selection(int arr[], int low, int high, int k) {*  *while (low <= high) {*  *int pivot\_index = rand() % (high - low + 1) + low;*  *int pivot\_pos = partition(arr, low, high, pivot\_index);*  *if (pivot\_pos == k) {*  *return arr[k];*  *} else if (pivot\_pos > k) {*  *high = pivot\_pos - 1;*  *} else {*  *low = pivot\_pos + 1;*  *}*  *}*  *return -1;*  *}*  *int main() {*  *int n, k;*  *printf("Enter the size of array: ");*  *scanf("%d", &n);*  *int arr[n];*  *printf("Enter the elements of array: ");*  *for (int i = 0; i < n; i++) {*  *scanf("%d", &arr[i]);*  *}*  *printf("Enter the kth smallest element to find: ");*  *scanf("%d", &k);*  *int result = randomized\_selection(arr, 0, n-1, k-1);*  *if (result == -1) {*  *printf("Error: k is out of bounds\n");*  *} else {*  *printf("The %dth smallest element is: %d\n", k, result);*  *}*  *return 0;*  *}* |
| **Sample Input:** |
| *Enter the size of array: 6*  *Enter the elements of array: 1*  *2*  *3*  *333*  *2*  *1* |
| **Sample Output:** |
| *Enter the kth smallest element to find: 2*  *The 2th smallest element is: 1* |
| **Time complexity calculation:** |
| *O(n^2)* |