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

**Term-work Report**

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Term-work** | *01* | | | | |  |  | | | | |
| **Student Name** | Vivek Kengalagutti | | | | |  |  | | | | |
| **SRN** | 01FE21BEC153 | | | | **Roll Number** | | 343 | | **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** | | Array | | | | | O(n) | | | |
| **2** | | Linked list | | | | | O(n) | | | |
| Recursive | **1** | | Array | | | | | O(2^n) | | | |
| **2** | | Link list | | | | |  | | | |
| Back tracking | **1** | | Array | | | | | O(n!) | | | |
| **2** | | Array | | | | | O(2^n) | | | |
| Divide and conquer | **1** | | Array | | | | | O(nlogn) | | | |
| **2** | | Array | | | | | O(logn) | | | |
| Dynamic programming | **1** | | Array | | | | | O(n) | | | |
| **2** | | Array | | | | | O(n) | | | |
| Greedy | **1** | | Array | | | | | O(n) | | | |
| **2** | | stacks | | | | | O(nlogn) | | | |
| Branch and bound | **1** | | Queues | | | | | O(1) | | | |
| **2** | | Array | | | | | O(n^2) | | | |
| Brute force | **1** | | Strings | | | | | O(m\*n) | | | |
| **2** | | Array | | | | | O(n^k) | | | |
| Randomized | **1** | | Array | | | | | O(logn) | | | |
| **2** | | Array | | | | | O(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?

*<Write your answer here>*

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Modularity** |  | **Documentation** |  | **Indentation** |  | **Programming practices** |  |
| **Type of Algorithm: Iterative** | | | | | | | | |
| **Details of the algorithm:** | | | | | | | | |
| An iterative algorithm is a type of algorithm that repeats a set of instructions multiple times, with each repetition referred to as an iteration. In iterative algorithms, the repetition of the set of instructions continues until a specific condition is met or a specific number of iterations have been completed. Iterative algorithms can be contrasted with recursive algorithms, which call themselves to solve subproblems until they reach a base case.  Iterative algorithms are often used in situations where the problem being solved requires a large number of repeated operations, such as in numerical methods for solving differential equations or in optimizing algorithms like gradient descent. They are generally more efficient than recursive algorithms since they avoid the overhead associated with recursive function calls. However, iterative algorithms can be more difficult to write and understand, particularly when the problem being solved has a complex structure or requires non-trivial bookkeeping. | | | | | | | | |
| **Code for example 1:** | | | | | | | | |
| #include <stdio.h>  int main()  {  int arr[5];  int sum=0,n;  scanf("%d",&n);  for (int i = 0; i < n; i++)  {  scanf("%d",&arr[i]);  sum=sum+arr[i];  }  printf("%d\n", sum);  return 0;  } | | | | | | | | |
| **Sample Input:** | | | | | | | | |
| *4*  *2 3 4 5* | | | | | | | | |
| **Sample Output:** | | | | | | | | |
| 14 | | | | | | | | |
| **Time complexity calculation:** | | | | | | | | |
| O(n) | | | | | | | | |

|  |
| --- |
| **Code for example 2:** |
| #include <stdio.h>  #include <stdlib.h>  struct Node {  int data;  struct Node\* next;  };  int main()  {  // create a linked list with three nodes  struct Node\* head = (struct Node\*) malloc(sizeof(struct Node));  scanf("%d",&head->data);  head->next = (struct Node\*) malloc(sizeof(struct Node));  scanf("%d",&head->next->data);  head->next->next = (struct Node\*) malloc(sizeof(struct Node));  scanf("%d",&head->next->next->data);  head->next->next->next = NULL;  // traverse the linked list and print its contents  printf("Linked List: ");  printList(head);  return 0;  }  void printList(struct Node\* head)  {  struct Node\* current = head;  while (current != NULL) {  printf("%d ", current->data);  current = current->next;  }  } |
| **Sample Input:** |
| *45 78 92* |
| **Sample Output:** |
| *linked list:45 78 92* |
| **Time complexity calculation:** |
| O(n) |

|  |
| --- |
|  |
| **Type of Algorithm: Recursive** | |
| **Details of the algorithm:2** | |
| Recursive algorithm is a type of algorithm that solves a problem by breaking it down into smaller subproblems of the same type, and recursively solving these subproblems. It is based on the idea of solving a problem by reducing it to a simpler version of the same problem.  In a recursive algorithm, a function calls itself with smaller input values until a base case is reached, at which point the recursion stops and the solution is returned. The base case is the simplest possible input for the problem, and it is what terminates the recursive calls.  Recursive algorithms are commonly used in many fields such as computer science, mathematics, and engineering. They are particularly useful when the problem can be broken down into smaller subproblems that are of the same type, and when the solution to the problem can be expressed in terms of the solutions to the subproblems.  However, it is important to note that recursive algorithms can be less efficient than other types of algorithms, as they may involve multiple recursive calls and repeated calculations. Careful design and optimization is necessary to ensure that the algorithm is both correct and efficient. | |
| **Code for example 1:** | |
| //Find Fibonacci of a number  #include <stdio.h>  int fibonacci(int n, int memo[]);  int main() {  int n;  printf("Enter a non-negative integer: ");  scanf("%d", &n);  int memo[n+1]; // array to store previously calculated Fibonacci numbers  for (int i = 0; i <= n; i++) {  memo[i] = -1; // initialize all elements to -1  }  printf("The %dth Fibonacci number is %d.\n", n, fibonacci(n, memo));  return 0;  }  int fibonacci(int n, int memo[]) {  if (memo[n] != -1) { // if Fibonacci number already calculated, return it  return memo[n];  }  int result;  if (n == 0 || n == 1) {  result = n;  } else {  result = fibonacci(n - 1, memo) + fibonacci(n - 2, memo);  }  memo[n] = result; // store calculated Fibonacci number in memo array  return result;  } | |
| **Sample Input:** | |
| 8 | |
| **Sample Output:** | |
| The 8th Fibonacci number is 21 | |
| **Time complexity calculation:** | |
| O(2^n) | |

|  |
| --- |
| **Code for example 2:** |
| #include <stdio.h>  #include <stdlib.h>  typedef struct Node {  int value;  struct Node\* next;  } Node;  Node\* insert(Node\* head, int value) {  if (head == NULL) {  head = malloc(sizeof(Node));  head->value = value;  head->next = NULL;  return head;  }  head->next = insert(head->next, value);  return head;  }  void print\_list(Node\* head) {  if (head == NULL) {  printf("\n");  return;  }  printf("%d ", head->value);  print\_list(head->next);  }  void delete\_list(Node\* head) {  if (head == NULL) {  return;  }  delete\_list(head->next);  free(head);  }  int main() {  Node\* head = NULL;  int n, value;  printf("Enter the number of elements: ");  scanf("%d", &n);  printf("Enter the elements: ");  for (int i = 0; i < n; i++) {  scanf("%d", &value);  head = insert(head, value);  }  printf("The list is: ");  print\_list(head);  delete\_list(head);  return 0;  } |
| **Sample Input:** |
| 2  5 10 |
| **Sample Output:** |
| 5 10 |
| **Time complexity calculation:** |
| *O(n)* |

|  |
| --- |
|  |
| **Type of Algorithm: Back tracking** | |
| **Details of the algorithm:2** | |
| Backtracking is an algorithmic technique for solving computational problems that involve searching for all possible solutions by incrementally building candidates to the solutions, and abandoning a candidate ("backtracking") as soon as it determines that the candidate cannot possibly be completed to a valid solution.  The approach starts with a possible solution to a problem and tries to incrementally build the solution until the desired state is achieved. However, if the current candidate solution is found to be invalid or does not lead to the desired state, the algorithm abandons it and "backtracks" to the previous valid state to explore other possible solutions.  Backtracking is particularly useful for solving problems in which there are multiple solutions, and the most appropriate solution needs to be identified. It is commonly used for solving problems in combinatorics, such as the N-Queens problem, Sudoku, and solving mazes.  One of the key aspects of backtracking is that it uses a depth-first search approach to explore the possible solutions, which means that the algorithm will explore each possible path until it reaches the end of the path, and then backtrack to the previous state to explore another path. This process continues until all possible paths have been explored, and the optimal solution is found.  Backtracking can be implemented using various data structures such as arrays, stacks, and trees, and it is often used in conjunction with other algorithms, such as dynamic programming and greedy algorithms, to solve complex computational problems. | |
| **Code for example 1:** | |
| #include <stdio.h>  #include <stdbool.h>  #define MAX\_SIZE 50  int solution[MAX\_SIZE];  bool used[MAX\_SIZE] = {false};  int n;  void print\_solution() {  printf("Solution: ");  for (int i = 0; i < n; i++)  {  printf("%d ", solution[i]);  }  printf("\n");  }  void backtrack(int pos) {  if (pos == n)  {  print\_solution();  return;  }  for (int i = 1; i <= n; i++) {  if (!used[i]) {  solution[pos] = i;  used[i] = true;  backtrack(pos + 1);  used[i] = false;  }  }  }  int main() {  printf("Enter num: ");  scanf("%d", &n);  backtrack(0);  return 0;  } | |
| **Sample Input:** | |
| 2 | |
| **Sample Output:** | |
| 1 2 | |
| **Time complexity calculation:** | |
| O(n!) | |

|  |
| --- |
| **Code for example 2:** |
| // Subset Sum problem  #include <stdio.h>  #include <stdbool.h>  #define N 100  void print\_subset(int subset[N], int len)  {  int i;  printf("{");  for (i = 0; i < len; i++)  {  printf("%d", subset[i]);  if (i < len - 1) {  printf(", ");  }  }  printf("}\n");  }  bool subset\_sum(int set[N], int subset[N], int target, int len, int start, int n)  {  int i;  if (target == 0)  {  print\_subset(subset, len);  return true;  }  for (i = start; i < n; i++)  {  if (set[i] <= target)  {  subset[len] = set[i];  if (subset\_sum(set, subset, target - set[i], len + 1, i + 1, n))  {  return true;  }  }  }  return false;  }  int main()  {  int set[N];  int subset[N];  int n, target, i;  printf("Enter the size of the set: ");  scanf("%d", &n);  printf("Enter the set of integers: ");  for (i = 0; i < n; i++)  {  scanf("%d", &set[i]);  }  printf("Enter the target sum: ");  scanf("%d", &target);  if (!subset\_sum(set, subset, target, 0, 0, n))  {  printf("No subset found.\n");  }  return 0;  } |
| **Sample Input:** |
| 4  23 45 34 18  52 |
| **Sample Output:** |
| {34,18} |
| **Time complexity calculation:** |
| O(2^n) |

|  |
| --- |
|  |
| **Type of Algorithm: Divide and conquer** | |
| **Details of the algorithm:4** | |
| Divide and conquer is a problem-solving technique that involves breaking down a complex problem into smaller, more manageable sub-problems, solving each sub-problem separately, and then combining the solutions of the sub-problems to arrive at a solution for the original problem.  The divide and conquer approach involves three basic steps: divide, conquer, and combine. In the divide step, the problem is broken down into smaller sub-problems. In the conquer step, each sub-problem is solved recursively using the same algorithm. Finally, in the combine step, the solutions to the sub-problems are combined to form a solution to the original problem.  This approach is often used in problems where a large input can be broken down into smaller pieces that can be processed independently. It is widely used in various areas such as computer science, mathematics, engineering, and science. Common examples of algorithms that use divide and conquer include binary search, quicksort, mergesort, and the Karatsuba algorithm for multiplication of large numbers.  The divide and conquer algorithm can be very efficient and scalable, especially when dealing with large datasets, as the algorithm can be easily parallelized. However, the approach may not always be the most optimal or efficient for all types of problems. In some cases, it may be more appropriate to use other techniques such as dynamic programming or greedy algorithms. | |
| **Code for example 1:** | |
| //Merge Sort  #include <stdio.h>  void merge(int arr[], int left[], int leftSize, int right[], int rightSize)  {  int i = 0, j = 0, k = 0;  while (i < leftSize && j < rightSize)  {  if (left[i] <= right[j])  {  arr[k] = left[i];  i++;  }  else {  arr[k] = right[j];  j++;  }  k++;  }  while (i < leftSize)  {  arr[k] = left[i];  i++;  k++;  }  while (j < rightSize)  {  arr[k] = right[j];  j++;  k++;  }  }  void merge\_sort(int arr[], int n)  {  if (n < 2) {  return;  }  int mid = n / 2;  int left[mid];  int right[n - mid];  int i;  for (i = 0; i < mid; i++)  {  left[i] = arr[i];  }  for (i = mid; i < n; i++)  {  right[i - mid] = arr[i];  }  merge\_sort(left, mid);  merge\_sort(right, n - mid);  merge(arr, left, mid, right, n - mid);  }  int main() {  int n;  printf("Enter the number of elements: ");  scanf("%d", &n);  int arr[n];  printf("Enter the elements:\n");  for (int i = 0; i < n; i++) {  scanf("%d", &arr[i]);  }  printf("\n");  merge\_sort(arr, n);  for (int i = 0; i < n; i++)  {  printf("%d ", arr[i]);  }  printf("\n");  return 0;  } | |
| **Sample Input:** | |
| 5  23 67 43 13 90 | |
| **Sample Output:** | |
| 13 23 43 67 90 | |
| **Time complexity calculation:** | |
| O(n log n) | |

|  |
| --- |
| **Code for example 2:** |
| BINARY SEARCH:  #include <stdio.h>  #define MAX\_SIZE 200  int binary\_search(int array[], int left, int right, int x) {  if (left > right) {  return -1;  }  int mid = (left + right) / 2;  if (array[mid] == x) {  return mid;  } else if (array[mid] < x) {  return binary\_search(array, mid + 1, right, x);  } else {  return binary\_search(array, left, mid - 1, x);  }  }  int main() {  int n;  printf("Enter the size of the array: ");  scanf("%d", &n);  int array[MAX\_SIZE];  printf("Enter the elements of the array in sorted order: ");  for (int i = 0; i < n; i++) {  scanf("%d", &array[i]);  }  int x;  printf("Enter the element to search for: ");  scanf("%d", &x);  int index = binary\_search(array, 0, n - 1, x);  if (index != -1) {  printf("%d is found at index %d.\n", x, index);  } else {  printf("%d is not found in the array.\n", x);  }  return 0;  } |
| **Sample Input:** |
| 3  12 5 36  7 |
| **Sample Output:** |
| Element not found |
| **Time complexity calculation:** |
| O(log n) |

|  |
| --- |
| **Type of Algorithm: Dynamic programming** |
| **Details of the algorithm:5** |
| Dynamic programming (DP) is a problem-solving technique that involves breaking down a complex problem into smaller subproblems and solving each subproblem once, storing the results of each subproblem and reusing them as needed.  DP is useful when there are overlapping subproblems and optimal substructure, which means that the solution to a problem can be constructed from the solutions of its subproblems. By storing the results of the subproblems in a table or an array, DP avoids redundant computations and reduces the time complexity of the algorithm.  DP can be used to solve a wide range of problems, including optimization problems, path-finding problems, and combinatorial problems. Examples of problems that can be solved using DP include the Knapsack problem, the Fibonacci sequence, and the longest common subsequence problem.  DP can be implemented using various data structures, such as arrays, tables, and memoization. Bottom-up and top-down approaches are the two common approaches to implementing DP. Bottom-up DP involves solving subproblems in a specific order and using the results to solve larger problems, while top-down DP involves solving larger problems by recursively solving smaller subproblems. |
| **Code for example 1:** |
| //Fibonacci number in the array  #include <stdio.h>  int fibonacci(int n) {  int fib[n+1];  int i;  fib[0] = 0;  fib[1] = 1;  for (i = 2; i <= n; i++) {  fib[i] = fib[i-1] + fib[i-2];  }  return fib[n];  }  int main() {  int n;  printf("Enter the index of the Fibonacci number you want to find: ");  scanf("%d", &n);  printf("The %dth Fibonacci number is %d\n", n, fibonacci(n));  return 0;  } |
| **Sample Input:** |
| 24 |
| **Sample Output:** |
| 46368 |
| **Time complexity calculation:** |
| O(n) |

|  |
| --- |
| **Code for example 2:** |
| MAXIMUM SUB ARRAY SUM:  #include <stdio.h>  #define MAX\_SIZE 200  int array[MAX\_SIZE];  int max\_subarr(int n) {  int max\_ending\_here = 0;  int max\_so\_far = 0;  for (int i = 0; i < n; i++) {  max\_ending\_here = (max\_ending\_here + array[i] > 0) ? max\_ending\_here + array[i] : 0;  max\_so\_far = (max\_so\_far < max\_ending\_here) ? max\_ending\_here : max\_so\_far;  }  return max\_so\_far;  }  int main() {  int n;  printf("Enter the size of the arr\n ");  scanf("%d", &n);  printf("Enter the elements of the arr\n ");  for (int i = 0; i < n; i++) {  scanf("%d", &array[i]);  }  int result = max\_subarr(n);  printf("The maximum subarr sum is %d.\n", result);  return 0;  } |
| **Sample Input:** |
| 3  54 3 16 |
| **Sample Output:** |
| 73 |
| **Time complexity calculation:** |
| O(n) |

|  |
| --- |
| **Type of Algorithm: Greedy** |
| **Details of the algorithm:6** |
| Greedy algorithm is a problem-solving approach that involves making the locally optimal choice at each step with the hope of finding a global optimum. It is called "greedy" because it makes choices based on the current best option, without considering the future consequences of that choice.  In general, greedy algorithms have the following characteristics:  - They are iterative.  - They make a sequence of decisions, one at a time.  - At each decision point, they choose the best option available at that moment.  - Once a decision is made, it is never reconsidered.  - They aim to find the global optimum, but may not always succeed.  Greedy algorithms are often used in optimization problems where the goal is to find the best solution among many possible choices. Some common examples include finding the shortest path in a graph, scheduling activities to maximize profit or minimize completion time, and making change with the fewest number of coins.  While greedy algorithms can be very efficient and easy to implement, they may not always produce the best possible solution. In some cases, a more complex approach may be needed to guarantee the optimal solution. |
| **Code for example 1:** |
| //Coin Changing Problem  #include <stdio.h>  #include <stdlib.h>  void coinChange(int coins[], int n, int amount) {  int count = 0;  printf("Coins used to make amount %d are:\n", amount);  for (int i = 0; i < n; i++) {  while (amount >= coins[i]) {  printf("%d ", coins[i]);  amount -= coins[i];  count++;  }  }  printf("\nTotal number of coins used = %d\n", count);  }  int main() {  int n, amount;  printf("Enter the number of coins: ");  scanf("%d", &n);  int coins[n];  printf("Enter the value of each coin:\n");  for (int i = 0; i < n; i++) {  scanf("%d", &coins[i]);  }  printf("Enter the amount to be changed: ");  scanf("%d", &amount);  coinChange(coins, n, amount);  return 0;  } |
| **Sample Input:** |
| 3  5 2 1  33 |
| **Sample Output:** |
| 5 5 5 5 5 5 2 1  8 |
| **Time complexity calculation:** |
| O(n) |

|  |
| --- |
| **Code for example 2:** |
| //Interval Scheduling Problem  #include <stdio.h>  #include <stdlib.h>  #define MAX\_SIZE 100  typedef struct interval {  int start;  int end;  } interval;  int compare(const void\* a, const void\* b) {  interval\* interval1 = (interval\*)a;  interval\* interval2 = (interval\*)b;  return interval1->end - interval2->end;  }  void intervalScheduling(interval intervals[], int n) {  int i, j;  int count = 1;  interval stack[MAX\_SIZE];  stack[0] = intervals[0];  for (i = 1, j = 0; i < n; i++) {  if (intervals[i].start >= stack[j].end) {  stack[++j] = intervals[i];  count++;  }  }  printf("Max number of non-overlapping intervals = %d\n", count);  printf("Selected intervals:\n");  for (i = 0; i < count; i++) {  printf("[%d, %d] ", stack[i].start, stack[i].end);  }  }  int main() {  int n, i;  printf("Enter the number of intervals: ");  scanf("%d", &n);  interval intervals[n];  printf("Enter the start and end times for each interval:\n");  for (i = 0; i < n; i++) {  scanf("%d %d", &intervals[i].start, &intervals[i].end);  }  qsort(intervals, n, sizeof(interval), compare);  intervalScheduling(intervals, n);  return 0;  } |
| **Sample Input:** |
| 3  1 3  4 6  8 9 |
| **Sample Output:** |
| 3  [1,3] [4,6] [8,9] |
| **Time complexity calculation:** |
| O(nlogn) |

|  |
| --- |
| **Type of Algorithm: Branch and Bound** |
| **Details of the algorithm:7** |
| Branch and Bound is an optimization algorithm used to solve combinatorial optimization problems. It is a systematic way of exploring all possible solutions to the problem by dividing the solution space into smaller sub-spaces, and then eliminating branches of the search tree that cannot possibly contain the optimal solution. The algorithm proceeds by constructing a search tree, where each node in the tree corresponds to a partial solution to the problem, and the root node represents the empty solution.  The algorithm evaluates each node in the search tree using an upper bound and a lower bound on the objective function of the problem. Nodes with upper bounds lower than the current best solution are pruned, and the search continues in the promising sub-trees. The algorithm iteratively divides the solution space into smaller and smaller sub-spaces until the optimal solution is found or all sub-spaces have been explored.  Branch and Bound algorithm can be used for a wide range of optimization problems, including shortest path, knapsack, maximum flow, scheduling, and many others. It is particularly useful for large-scale problems where an exhaustive search is not feasible, and the problem exhibits a structure that can be exploited to reduce the search space. |
| **Code for example 1:** |
| //minimum cost of traversing all cities  #include <stdio.h>  #include <stdlib.h>  #include <limits.h>  #include <stdbool.h>  #include <string.h>  #include <ctype.h>  #include <math.h>  #define N 3  int findMin(int graph[N][N], bool visited[N], int u)  {  int min = INT\_MAX;  for (int v = 0; v < N; v++) {  if (graph[u][v] < min && !visited[v]) {  min = graph[u][v];  }  }  return min;  }  int findSecondMin(int graph[N][N], bool visited[N], int u)  {  int min = INT\_MAX, secondMin = INT\_MAX;  for (int v = 0; v < N; v++) {  if (graph[u][v] <= min && !visited[v]) {  secondMin = min;  min = graph[u][v];  }  else if (graph[u][v] <= secondMin && !visited[v]) {  secondMin = graph[u][v];  }  }  return secondMin;  }  int bound(int graph[N][N], bool visited[N], int u)  {  return findMin(graph, visited, u) + findSecondMin(graph, visited, u);  }  int tsp(int graph[N][N], bool visited[N], int currPos, int cost, int level, int \*minCost)  {  if (level == N) {  if (graph[currPos][0] && cost + graph[currPos][0] < \*minCost) {  \*minCost = cost + graph[currPos][0];  }  return 0;  }  int boundValue = bound(graph, visited, currPos);  if (boundValue + cost >= \*minCost) {  return 0;  }  int min = INT\_MAX;  for (int i = 0; i < N; i++) {  if (graph[currPos][i] && !visited[i]) {  visited[i] = true;  int temp = tsp(graph, visited, i, cost + graph[currPos][i], level + 1, minCost);  if (temp < min) {  min = temp;  }  visited[i] = false;  }  }  return min;  }  int main()  {  int graph[N][N];  printf("Enter the distance matrix:\n");  for (int i = 0; i < N; i++) {  for (int j = 0; j < N; j++) {  scanf("%d", &graph[i][j]);  }  }  bool visited[N];  memset(visited, false, sizeof(visited));  visited[0] = true;  int minCost = INT\_MAX;  tsp(graph, visited, 0, 0, 1, &minCost);  printf("Minimum cost: %d\n", minCost);  return 0;  } |
| **Sample Input:** |
| 3 5 7 4 7 5 1 2 9 |
| **Sample Output:** |
| 11 |
| **Time complexity calculation:** |
| O(1) |

|  |
| --- |
| **Code for example 2:** |
| MAXIMUM SUBSET SUM:  #include <stdio.h>  #include <stdlib.h>  #include <limits.h>  #define MAX\_SIZE 60  int array[MAX\_SIZE];  int best\_solution[MAX\_SIZE];  int current\_solution[MAX\_SIZE];  int best\_sum = INT\_MIN;  int current\_sum = 0;  int n;  void branch\_and\_bound(int level) {  if (level == n) {  if (current\_sum > best\_sum) {  best\_sum = current\_sum;  for (int i = 0; i < n; i++) {  best\_solution[i] = current\_solution[i];  }  }  return;  }  if (current\_sum + array[level] > best\_sum) {  current\_sum += array[level];  current\_solution[level] = 1;  branch\_and\_bound(level + 1);  current\_solution[level] = 0;  current\_sum -= array[level];  }  branch\_and\_bound(level + 1);  }  int main() {  printf("Enter the size of the array: ");  scanf("%d", &n);  printf("Enter the elements of the array: ");  for (int i = 0; i < n; i++) {  scanf("%d", &array[i]);  }  branch\_and\_bound(0);  printf("The maximum subset sum is %d.\n", best\_sum);  printf("The maximum subset is: { ");  for (int i = 0; i < n; i++) {  if (best\_solution[i]) {  printf("%d ", array[i]);  }  }  printf("}\n");  return 0;  } |
| **Sample Input:** |
| 5  20 3 2 5 1 |
| **Sample Output:** |
| 31  {20 3 2 5 1} |
| **Time complexity calculation:** |
| O(n^2) |

|  |
| --- |
| **Type of Algorithm: Brute force** |
| **Details of the algorithm:8** |
| Brute force algorithm is a straightforward, exhaustive and systematic approach to solving a problem by examining every possible solution and choosing the best one. This algorithm does not involve any optimization techniques, but instead generates all possible solutions to a problem and checks which one meets the given constraints.  In this approach, the algorithm systematically generates all possible solutions and checks if each solution satisfies the given problem constraints. The main disadvantage of the brute force approach is that it is not efficient for larger problem instances because the number of possible solutions grows exponentially as the size of the input increases.  Despite its limitations, brute force algorithm is sometimes the only approach available for solving certain problems, or it may be used as a benchmark for comparing the performance of more efficient algorithms. It is commonly used in computer science, mathematics, physics, and engineering. |
| **Code for example 1:** |
| //String matching problem  #include <stdio.h>  #include <string.h>  int bruteForce(char\* pattern, char\* text) {  int n = strlen(text);  int m = strlen(pattern);  int i, j;  for (i = 0; i <= n-m; i++) {  j = 0;  while (j < m && text[i+j] == pattern[j]) {  j++;  }  if (j == m) {  return i;  }  }  return -1;  }  int main() {  char pattern[100], text[100];  printf("Enter the text: ");  fgets(text, 100, stdin);  printf("Enter the pattern to search: ");  fgets(pattern, 100, stdin);  text[strcspn(text, "\n")] = 0;  pattern[strcspn(pattern, "\n")] = 0;  int index = bruteForce(pattern, text);  if (index != -1) {  printf("Pattern found at index %d\n", index);  } else {  printf("Pattern not found\n");  }  return 0;  } |
| **Sample Input:** |
| I am freaking Gaurdian of the Galaxy  Gaurdian |
| **Sample Output:** |
| Pattern found at index 14 |
| **Time complexity calculation:** |
| O(m \* n) |

|  |
| --- |
| **Code for example 2:** |
| //finding the maximum number in an array using brute force algorithm  #include <stdio.h>  int main() {  int n, i, max;  printf("Enter the number of elements in the array: ");  scanf("%d", &n);  int arr[n];  printf("Enter the elements of the array:\n");  for(i = 0; i < n; i++) {  scanf("%d", &arr[i]);  }  max = arr[0];  for(i = 1; i < n; i++) {  if(arr[i] > max) {  max = arr[i];  }  }  printf("The maximum number in the array is: %d", max);  return 0;  } |
| **Sample Input:** |
| 5  76 54 98 73 18 |
| **Sample Output:** |
| 98 |
| **Time complexity calculation:** |
| O(N^k) |

|  |
| --- |
| **Type of Algorithm: Randomized** |
| **Details of the algorithm:9** |
| A randomized algorithm is an algorithm that employs a degree of randomness as part of its logic in order to solve a problem. Randomized algorithms are often used in situations where deterministic algorithms may fail or may not exist. They are particularly useful when the input data size is too large, and a deterministic algorithm takes too long to run or is not efficient enough to provide an exact solution. In these cases, a randomized algorithm can provide an approximate solution in a reasonable amount of time. |
|  |
| //Randomized Select  #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) {  srand(time(NULL));  int random = low + rand() % (high - low);  swap(&arr[random], &arr[high]);  int pivot = 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;  }  int randomized\_select(int arr[], int low, int high, int k) {  if (low == high) {  return arr[low];  }  int pivot = partition(arr, low, high);  int length = pivot - low + 1;  if (k == length) {  return arr[pivot];  } else if (k < length) {  return randomized\_select(arr, low, pivot-1, k);  } else {  return randomized\_select(arr, pivot+1, high, k-length);  }  }  int main() {  int n, k;  printf("Enter the number of elements in 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("Enter the value of k: ");  scanf("%d", &k);  int kth\_smallest = randomized\_select(arr, 0, n-1, k);  printf("The %dth smallest element is %d\n", k, kth\_smallest);  return 0;  } |
| **Sample Input:** |
| 6  3 8 6 1 0 3 2  2 |
| **Sample Output:** |
| 2 |
| **Time complexity calculation:** |
| O(n log n) |

|  |
| --- |
| **Type of Algorithm: Randomized** |
| **Details of the algorithm:9** |
| A randomized algorithm is an algorithm that employs a degree of randomness as part of its logic in order to solve a problem. Randomized algorithms are often used in situations where deterministic algorithms may fail or may not exist. They are particularly useful when the input data size is too large, and a deterministic algorithm takes too long to run or is not efficient enough to provide an exact solution. In these cases, a randomized algorithm can provide an approximate solution in a reasonable amount of time. |
|  |
| //Randomized Select  #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) {  srand(time(NULL));  int random = low + rand() % (high - low);  swap(&arr[random], &arr[high]);  int pivot = 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;  }  int randomized\_select(int arr[], int low, int high, int k) {  if (low == high) {  return arr[low];  }  int pivot = partition(arr, low, high);  int length = pivot - low + 1;  if (k == length) {  return arr[pivot];  } else if (k < length) {  return randomized\_select(arr, low, pivot-1, k);  } else {  return randomized\_select(arr, pivot+1, high, k-length);  }  }  int main() {  int n, k;  printf("Enter the number of elements in 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("Enter the value of k: ");  scanf("%d", &k);  int kth\_smallest = randomized\_select(arr, 0, n-1, k);  printf("The %dth smallest element is %d\n", k, kth\_smallest);  return 0;  } |
| **Sample Input:** |
| 6  3 8 6 1 0 3 2  2 |
| **Sample Output:** |
| 2 |
| **Time complexity calculation:** |
| O(n log n) |

|  |
| --- |
| **Code for example 2:** |
| //Random permutation  #include <stdio.h>  #include <stdlib.h>  #include <time.h>  void swap(int\* a, int\* b) {  int temp = \*a;  \*a = \*b;  \*b = temp;  }  void random\_permutation(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]);  }  }  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: ");  for (int i = 0; i < n; i++) {  printf("%d ", arr[i]);  }  printf("\n");  random\_permutation(arr, n);  printf("Random permutation: ");  for (int i = 0; i < n; i++) {  printf("%d ", arr[i]);  }  printf("\n");  return 0;  } |
| **Sample Input:** |
| 6  6 8 5 3 1 7 |
| **Sample Output:** |
| 1 8 6 3 5 7 |
| **Time complexity calculation:** |
| O(n) |