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

**Term-work Report**

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Term-work** | *01* | | | | |  |  | | | | |
| **Student Name** | OHIL HOSAMANE | | | | |  |  | | | | |
| **SRN** | 01FE21BEC358 | | | | **Roll Number** | | 408 | | **Division** | D | |
| **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** | | Factorial(Array) | | | | | O(n) | | | |
| **2** | | Bubble sort(Array) | | | | | O(n^2) | | | |
| Recursive | **1** | | Binary search(Array) | | | | | O(1) | | | |
| **2** | | Fibonacci | | | | | O(2^n) | | | |
| Back tracking | **1** | | Generate all binary strings of length N | | | | | O(2^n) | | | |
| **2** | | find all permutations of a given set of numbers | | | | | O(n!) | | | |
| Divide and conquer | **1** | | Power function | | | | | O(log n) | | | |
| **2** | | Merge sort | | | | | O(n log n) | | | |
| Dynamic programming | **1** | | computes the longest common subsequence of two strings | | | | | O(m \* n). | | | |
| **2** | | Fibonacci | | | | | O(1) | | | |
| Greedy | **1** | | finds the minimum number of coins required to make change for a given amount of money | | | | | O(n) | | | |
| **2** | | find the minimum number of platforms required so that no train has to wait for another train to leave | | | | | O(1) | | | |
| Branch and bound | **1** | | Knapsack | | | | | O(2^n \* n). | | | |
| **2** | | traveling salesman problem | | | | | O(n^2) | | | |
| Brute force | **1** | | find the maximum sum of a subarray | | | | | O(n^3) | | | |
| **2** | | find the maximum product of two numbers in an array | | | | | O(n^3) | | | |
| Randomized | **1** | | Randomized Quick Sort | | | | | O(n log n) | | | |
| **2** | | finding the k-th smallest element in an array using the quickselect algorithm | | | | | O(n)/O(n^2) | | | |

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

Yes, I was able to solve this.

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

Browser .

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

None.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Modularity** |  | **Documentation** |  | **Indentation** |  | **Programming practices** |  |
| **Type of Algorithm:** Iterative | | | | | | | |
| **Details of the algorithm:** | | | | | | | |
| An iterative algorithm is a type of algorithm that repeatedly applies a sequence of steps to approximate a solution to a problem. The name "iterative" comes from the fact that the algorithm involves a repeated process or iteration to converge towards a solution.  Applications: numerical analysis, optimization, machine learning, signal processing, and image processing.  The steps involved in an iterative algorithm typically include:  Initialization  Iteration  Convergence  Termination  Output | | | | | | | |
| **Code for example 1:** | | | | | | | |
| #include <stdio.h>  void factorial(int n, int\* result) {  result[0] = 1;  for (int i = 1; i <= n; i++) {  int carry = 0;  for (int j = 0; j < 100; j++) {  int product = result[j] \* i + carry;  result[j] = product % 10;  carry = product / 10;  }  }  }  int main() {  int n = 100;  int result[100] = {0};  factorial(n, result);  printf("%d! = ", n);  int i = 99;  while (result[i] == 0) {  i--;  }  for (; i >= 0; i--) {  printf("%d", result[i]);  }  printf("\n");  return 0;  } | | | | | | | |
| **Sample Input:** | | | | | | | |
| 5,10,3 | | | | | | | |
| **Sample Output:** | | | | | | | |
| 120,3628800,6 | | | | | | | |
| **Time complexity calculation:** | | | | | | | |
| result[0] = 1; - This operation takes constant time, or O(1).  for (int i = 1; i <= n; i++) { - This loop iterates n times, so its time complexity is O(n).  int carry = 0; - This operation takes constant time, or O(1).  for (int j = 0; j < 100; j++) { - This loop iterates 100 times, so its time complexity is O(1).  int product = result[j] \* i + carry; - This operation takes constant time, or O(1).  result[j] = product % 10; - This operation takes constant time, or O(1).  carry = product / 10; - This operation takes constant time, or O(1).  since it is nested inside an outer loop that runs O(n) times, the overall time complexity of the factorial function is O(n). | | | | | | | |

|  |
| --- |
| **Code for example 2:** |
| #include <stdio.h>  void bubble\_sort(int arr[], int n) {  int i, j;  for (i = 0; i < n; i++) {  for (j = 0; j < n-i-1; j++) {  if (arr[j] > arr[j+1]) {  int temp = arr[j];  arr[j] = arr[j+1];  arr[j+1] = temp;  }  }  }  }  int main() {  int arr[] = {5, 2, 1, 8, 4, 7};  int n = sizeof(arr) / sizeof(arr[0]);  bubble\_sort(arr, n);  printf("Sorted array: ");  for (int i = 0; i < n; i++) {  printf("%d ", arr[i]);  }  return 0;  } |
| **Sample Input:** |
| 1. arr = [5, 2, 1, 8, 4, 7] 2. arr = [3, 6, 2, 9, 1, 5, 4, 7, 8] |
| **Sample Output:** |
| 1. [1, 2, 4, 5, 7, 8] 2. [1, 2, 3, 4, 5, 6, 7, 8, 9] |
| **Time complexity calculation:** |
| The outer loop iterates n times.  On each iteration of the outer loop, the inner loop iterates n-i-1 times.  Therefore, the total number of iterations is:  n + (n-1) + (n-2) + ... + 1 = (n \* (n+1)) / 2  This gives us a time complexity of O(n^2) for Bubble Sort. |

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

Yes, I was able to solve this.

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

Browser .

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

None.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Modularity** |  | **Documentation** |  | **Indentation** |  | **Programming practices** |  |
| **Type of Algorithm:**  Recursive | | | | | | | |
| **Details of the algorithm:** | | | | | | | |
| A recursive algorithm is a type of algorithm that solves a problem by dividing it into smaller subproblems, solving each subproblem recursively, and then combining the solutions to the subproblems to obtain a solution to the original problem. The name "recursive" comes from the fact that the algorithm calls itself recursively to solve the subproblems.  Examples of applications include tree traversal, graph traversal, sorting algorithms, searching algorithms, and numerical analysis.  The steps involved in a recursive algorithm are:  Identify the base case.  Identify the recursive case.  Call the algorithm recursively.  Combine the solutions. | | | | | | | |
| **Code for example 1:** | | | | | | | |
| int binary\_search(int arr[], int low, int high, int x) {  if (high >= low) {  int mid = low + (high - low) / 2;  if (arr[mid] == x)  return mid;  if (arr[mid] > x)  return binary\_search(arr, low, mid - 1, x);  return binary\_search(arr, mid + 1, high, x);  }  return -1;  } | | | | | | | |
| **Sample Input:** | | | | | | | |
| int arr[] = {2, 5, 7, 9, 12, 15, 19};  int n = sizeof(arr) / sizeof(arr[0]);  int x = 9;  int result = binary\_search(arr, 0, n - 1, x);  printf("Index of %d in array: %d\n", x, result); | | | | | | | |
| **Sample Output:** | | | | | | | |
| Index of 9 in array: 3 | | | | | | | |
| **Time complexity calculation:** | | | | | | | |
| int binary\_search(int arr[], int low, int high, int x) {  if (high >= low) { // O(1)  int mid = low + (high - low) / 2; // O(1)  if (arr[mid] == x) // O(1)  return mid;  if (arr[mid] > x) // O(1)  return binary\_search(arr, low, mid - 1, x); // T(n/2)  return binary\_search(arr, mid + 1, high, x); // T(n/2)  }  return -1; // O(1)  } | | | | | | | |

|  |
| --- |
| **Code for example 2:** |
| #include <stdio.h>  int fibonacci(int n) {  if (n <= 1) {  return n;  } else {  return fibonacci(n - 1) + fibonacci(n - 2);  }  }  int main() {  int n = 10;  printf("Fibonacci sequence up to %d:\n", n);  for (int i = 0; i <= n; i++) {  printf("%d ", fibonacci(i));  }  printf("\n");  return 0;  } |
| **Sample Input:** |
| 6  10 |
| **Sample Output:** |
| 8  55 |
| **Time complexity calculation:** |
| // The time complexity of this algorithm is O(2^n), which is exponential.  int fibonacci(int n) {  if (n <= 1) {  return n; // This is the base case, which takes constant time.  } else {  // This line of code makes two recursive calls to the function with input sizes n-1 and n-2.  // Each of these recursive calls results in two more recursive calls until the base case is reached.  return fibonacci(n - 1) + fibonacci(n - 2);  }  } |

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

Yes, I was able to solve this.

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

Browser .

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

None.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Modularity** |  | **Documentation** |  | **Indentation** |  | **Programming practices** |  |
| **Type of Algorithm:** Back tracking | | | | | | | |
| **Details of the algorithm:** | | | | | | | |
| Backtracking is a problem-solving algorithmic technique that involves incrementally building a candidate solution and testing whether it satisfies all constraints. If it does not satisfy a constraint, the algorithm backtracks and tries a different solution. The name "backtracking" comes from the idea that the algorithm moves forward by making choices, but when it encounters a dead end, it backtracks to the last decision point and tries a different choice.  Backtracking algorithms are used in a variety of applications, such as finding a path in a maze, solving puzzles, generating all permutations or combinations of a set of elements, and solving optimization problems.  The steps involved in a backtracking algorithm are as follows:  Initialization  Choose  Constraint  Solution  Recurse  Backtrack  Termination | | | | | | | |
| **Code for example 1:** | | | | | | | |
| #include <stdio.h>  void generate(char \*str, int i, int N)  {  if (i == N) {  printf("%s\n", str);  return;  }  str[i] = '0';  generate(str, i+1, N);  str[i] = '1';  generate(str, i+1, N);  }  int main()  {  int N = 3;  char str[N+1];  str[N] = '\0';  generate(str, 0, N);  return 0;  } | | | | | | | |
| **Sample Input:** | | | | | | | |
| set[] = {1, 2, 3} | | | | | | | |
| **Sample Output:** | | | | | | | |
| {}  {3}  {2}  {2, 3}  {1}  {1, 3}  {1, 2}  {1, 2, 3} | | | | | | | |
| **Time complexity calculation:** | | | | | | | |
| The generate () function is called 2^n times, where n is the number of elements in the input set.  The print\_subset () function is called once for each subset, which is O(2^n) times.  The time complexity of the entire program is O(2^n) . | | | | | | | |

|  |
| --- |
| **Code for example 2:** |
| #include <stdio.h>  void swap(int \*a, int \*b)  {  int temp = \*a;  \*a = \*b;  \*b = temp;  }  void permute(int arr[], int l, int r)  {  if (l == r) {  for (int i = 0; i <= r; i++) {  printf("%d ", arr[i]);  }  printf("\n");  } else {  for (int i = l; i <= r; i++) {  swap(&arr[l], &arr[i]);  permute(arr, l+1, r);  swap(&arr[l], &arr[i]);  }  }  }  int main()  {  int arr[] = {1, 2, 3};  int n = sizeof(arr) / sizeof(arr[0]);  permute(arr, 0, n-1);  return 0;  } |
| **Sample Input:** |
| {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 permute function is O(n!), where n is the length of the array.  The reason for this is that the number of permutations of an array of length n is n!.  This is because there are n choices for the first element, n-1 choices for the second element, n-2 choices for the third element, and so on, down to 1 choice for the last element.  The permute function generates all possible permutations of the array by recursively permuting the remaining elements after fixing the current element.  This generates n! permutations in total.  The time complexity of the swap function is O(1), since it only performs a constant number of operations.  The time complexity of the main function is O(n!), since it calls the permute function which has a time complexity of O(n!). |

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

Yes, I was able to solve this.

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

Browser .

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

None.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Modularity** |  | **Documentation** |  | **Indentation** |  | **Programming practices** |  |
| **Type of Algorithm:** Divide and conquer | | | | | | | |
| **Details of the algorithm:** | | | | | | | |
| The divide and conquer algorithm gets its name from the three steps involved in its operation, which are "divide", "conquer", and "combine".  The steps involved in the divide and conquer algorithm are:  Divide: The problem is divided into smaller sub-problems that are easier to solve.  Conquer: The sub-problems are solved recursively using the same algorithm.  Combine: The solutions to the sub-problems are combined to obtain a solution to the original problem.  Some applications of the divide and conquer algorithm include:  Sorting algorithms: Merge Sort and Quick Sort are examples of sorting algorithms that use the divide and conquer approach.  Searching algorithms: Binary search is an example of a searching algorithm that uses the divide and conquer approach.  Computational geometry: Algorithms such as the closest pair of points and convex hull problems can be solved using the divide and conquer approach. | | | | | | | |
| **Code for example 1:** | | | | | | | |
| #include <stdio.h>  int power(int base, int exponent) {  if (exponent == 0)  return 1;  int result = power(base, exponent / 2);  if (exponent % 2 == 0)  return result \* result;  else  return base \* result \* result;  }  int main() {  int base = 2;  int exponent = 5;  int result = power(base, exponent);  printf("%d^%d = %d\n", base, exponent, result);  return 0;  } | | | | | | | |
| **Sample Input:** | | | | | | | |
| base = 2  exponent = 5  base = 3  exponent = 0 | | | | | | | |
| **Sample Output:** | | | | | | | |
| 2^5 = 32  3^0 = 1 | | | | | | | |
| **Time complexity calculation:** | | | | | | | |
| The recursion in the power function divides the exponent by 2 in each recursive call, leading to a depth of log base 2 of n (i.e., log₂ n) recursive calls.  Each recursive call involves a single multiplication operation.  Therefore, the total number of multiplication operations is proportional to the depth of the recursion, which is log base 2 of n.  Hence, the time complexity of the power function is O(log n). | | | | | | | |

|  |
| --- |
| **Code for example 2:** |
| #include <stdio.h>  void merge(int arr[], int left, int mid, int right) {  int i, j, k;  int n1 = mid - left + 1;  int n2 = right - mid;  int left\_arr[n1], right\_arr[n2];  for (i = 0; i < n1; i++)  left\_arr[i] = arr[left + i];  for (j = 0; j < n2; j++)  right\_arr[j] = arr[mid + 1 + j];  i = 0;  j = 0;  k = left;  while (i < n1 && j < n2) {  if (left\_arr[i] <= right\_arr[j]) {  arr[k] = left\_arr[i];  i++;  }  else {  arr[k] = right\_arr[j];  j++;  }  k++;  }  while (i < n1) {  arr[k] = left\_arr[i];  i++;  k++;  }  while (j < n2) {  arr[k] = right\_arr[j];  j++;  k++;  }  }  void mergeSort(int arr[], int left, int right) {  if (left < right) {  int mid = left + (right - left) / 2;  mergeSort(arr, left, mid);  mergeSort(arr, mid + 1, right);  merge(arr, left, mid, right);  }  }  int main() {  int arr[] = {12, 11, 13, 5, 6, 7};  int n = sizeof(arr)/sizeof(arr[0]);  printf("Original array: ");  for (int i = 0; i < n; i++)  printf("%d ", arr[i]);  printf("\n");  mergeSort(arr, 0, n - 1);  printf("Sorted array: ");  for (int i = 0; i < n; i++)  printf("%d ", arr[i]);  printf("\n");  return 0;  } |
| **Sample Input:** |
| arr = {4, 2, 7, 1, 9, 3}  arr = {5, 2, 1, 8, 9, 4, 6} |
| **Sample Output:** |
| Sorted array: 1 2 3 4 7 9  Sorted array: 1 2 4 5 6 8 9 |
| **Time complexity calculation:** |
| The main function initializes an array arr of size 6 and calculates its length n. Time complexity: O(1).  The main function then prints the original array arr using a for loop. Time complexity: O(n).  The main function calls mergeSort function to sort the array. Time complexity: O(1).  The mergeSort function recursively calls itself twice until the left pointer is greater than or equal to the right pointer. Time complexity: O(log n).  The merge function initializes variables i, j, and k, and calculates the length of two sub-arrays left\_arr and right\_arr. Time complexity: O(1).  The merge function then copies elements from the original array arr to left\_arr and right\_arr. Time complexity: O(n).  The merge function then uses a while loop to merge the two sub-arrays left\_arr and right\_arr back into the original array arr. Time complexity: O(n).  The main function prints the sorted array arr using a for loop. Time complexity: O(n).  Therefore, the overall time complexity of the Merge Sort algorithm implemented in this code is O(n\*log n). |

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

Yes, I was able to solve this.

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

Browser .

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

None.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Modularity** |  | **Documentation** |  | **Indentation** |  | **Programming practices** |  |
| **Type of Algorithm:** Dynamic Programming | | | | | | | |
| **Details of the algorithm:** | | | | | | | |
| Dynamic programming is a technique used in algorithm design to efficiently solve problems by breaking them down into smaller subproblems and then solving those subproblems in a bottom-up manner.  Applications:  The Knapsack problem: Given a set of items with weight and value, determine the maximum value that can be obtained by selecting a subset of the items that fit within a given weight limit.  The Longest Common Subsequence problem: Given two sequences of characters, find the longest subsequence that is common to both sequences.  The Shortest Path problem: Given a graph with weighted edges, find the shortest path between two nodes.  The steps involved in a dynamic programming algorithm are typically as follows:  Define the problem and determine the base case(s) that can be solved directly.  Break the problem down into smaller subproblems, each of which can be solved independently.  Define a recurrence relation that expresses the solution to the problem in terms of the solutions to the subproblems.  Determine the order in which to solve the subproblems.  Implement the algorithm using an appropriate data structure to store intermediate results.  Use the results of the subproblems to compute the solution to the original problem. | | | | | | | |
| **Code for example 1:** | | | | | | | |
| #include <stdio.h>  #include <string.h>  int max(int a, int b) {  return (a > b) ? a : b;  }  int lcs(char X[], char Y[], int m, int n) {  int L[m + 1][n + 1];  int i, j;  for (i = 0; i <= m; i++) {  for (j = 0; j <= n; j++) {  if (i == 0 || j == 0) {  L[i][j] = 0;  } else if (X[i - 1] == Y[j - 1]) {  L[i][j] = L[i - 1][j - 1] + 1;  } else {  L[i][j] = max(L[i - 1][j], L[i][j - 1]);  }  }  }  return L[m][n];  }  int main() {  char X[] = "AGGTAB";  char Y[] = "GXTXAYB";  int m = strlen(X);  int n = strlen(Y);  printf("Length of the longest common subsequence is %d\n", lcs(X, Y, m, n));  return 0;  } | | | | | | | |
| **Sample Input:** | | | | | | | |
| char X[] = "AGGTAB";  char Y[] = "GXTXAYB"; | | | | | | | |
| **Sample Output:** | | | | | | | |
| Length of the longest common subsequence is 4 | | | | | | | |
| **Time complexity calculation:** | | | | | | | |
| The initialization of the L array takes O(m \* n) time, since it has (m+1) rows and (n+1) columns.  The two nested loops used to fill in the values of L take O(m \* n) time, since each loop iterates m+1 and n+1 times, respectively.  The max function is called at most O(m \* n) times, once for each element of the L array. | | | | | | | |

|  |
| --- |
| **Code for example 2:** |
| #include <stdio.h>  int memo[100];  int fibonacci(int n) {  if (n <= 2) {  return 1;  }  if (memo[n] != 0) {  return memo[n];  }  memo[n] = fibonacci(n-1) + fibonacci(n-2);  return memo[n];  }  int main() {  int n = 6;  printf("Fibonacci number at index %d is %d\n", n, fibonacci(n));  return 0;  } |
| **Sample Input:** |
| n = 6 |
| **Sample Output:** |
| Fibonacci number at index 6 is 8 |
| **Time complexity calculation:** |
| In the main function, the integer variable n is assigned the value 6. This operation takes constant time, or O(1).  The printf function call takes constant time, or O(1).  The fibonacci function is called with argument n, which is 6.  The base case is checked and if it is met, the function returns 1. This operation takes constant time, or O(1).  If the base case is not met, the function checks if memo[n] is not zero, which takes constant time, or O(1).  If memo[n] is not zero, the function returns memo[n], which takes constant time, or O(1).  If memo[n] is zero, the function recursively calls itself with arguments n-1 and n-2.  This step involves two recursive function calls, which both take O(n) time.  The results of the recursive calls are added together, which takes constant time, or O(1).  The sum is stored in memo[n], which takes constant time, or O(1).  The function returns the sum, which takes constant time, or O(1). |

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

Yes, I was able to solve this.

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

Browser .

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

None.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Modularity** |  | **Documentation** |  | **Indentation** |  | **Programming practices** |  |
| **Type of Algorithm:** Greedy | | | | | | | |
| **Details of the algorithm:** | | | | | | | |
| A greedy algorithm is called "greedy" because it makes the locally optimal choice at each step without considering the global optimal solution. In other words, it chooses the best option available at the moment without thinking ahead.  The steps involved in a greedy algorithm are:  Define the problem and determine the objective function to be optimized.  Identify the set of choices available at each step.  Evaluate each choice using the objective function.  Choose the best option available at the moment and update the solution.  Repeat steps 2-4 until the desired solution is obtained.  Greedy algorithms are used in a wide range of applications such as scheduling, routing, and optimization problems. For example, the Huffman coding algorithm used in data compression is a famous example of a greedy algorithm. | | | | | | | |
| **Code for example 1:** | | | | | | | |
| #include <stdio.h>  int main() {  int coins[] = {25, 10, 5, 1};  int num\_coins = 4;  int amount = 68;  int count = 0;    for (int i = 0; i < num\_coins; i++) {  while (amount >= coins[i]) {  amount -= coins[i];  count++;  }  }    printf("Minimum number of coins required: %d", count);    return 0;  } | | | | | | | |
| **Sample Input:** | | | | | | | |
| coins[] = {25, 10, 5, 1}  num\_coins = 4  amount = 68 | | | | | | | |
| **Sample Output:** | | | | | | | |
| Minimum number of coins required: 5 | | | | | | | |
| **Time complexity calculation:** | | | | | | | |
| We have an array of coin denominations, which we iterate through once in the outer for loop. Therefore, the time complexity of this loop is O(n), where n is the number of coin denominations available.  For each denomination, we perform a while loop that repeats a number of times proportional to the value of the amount divided by the denomination. The time complexity of this while loop is O(amount/denomination), which represents the number of times the loop will run.  However, since the value of the amount decreases with each iteration, the total time taken by all the while loops put together is less than O(n\*amount) because the amount is typically smaller than the number of denominations.  Therefore, the overall time complexity of the program is O(n + amount), which simplifies to O(n) since amount is typically much smaller than n. | | | | | | | |

|  |
| --- |
| **Code for example 2:** |
| #include <stdio.h>  #include <stdlib.h>  int minPlatforms(int arrival[], int departure[], int n) {  qsort(arrival, n, sizeof(int), compare);  qsort(departure, n, sizeof(int), compare);  int plat\_needed = 1, result = 1;  int i = 1, j = 0;    while (i < n && j < n) {  if (arrival[i] <= departure[j]) {  plat\_needed++;  i++;  } else {  plat\_needed--;  j++;  }  if (plat\_needed > result) {  result = plat\_needed;  }  }  return result;  }  int compare(const void\* a, const void\* b) {  return \*(int\*)a - \*(int\*)b;  }  int main() {  int arrival[] = {900, 940, 950, 1100, 1500, 1800};  int departure[] = {910, 1200, 1120, 1130, 1900, 2000};  int n = sizeof(arrival)/sizeof(arrival[0]);    int min\_platforms = minPlatforms(arrival, departure, n);    printf("Minimum number of platforms required: %d", min\_platforms);    return 0;  } |
| **Sample Input:** |
| arrival = {900, 940, 950, 1100, 1500, 1800}  departure = {910, 1200, 1120, 1130, 1900, 2000}  n = 6 |
| **Sample Output:** |
| Minimum number of platforms required: 3 |
| **Time complexity calculation:** |
| Sorting the arrival and departure arrays takes O(n log n) time each, since the implementation of the qsort function is O(n log n) on average.  Initializing variables takes constant time, O(1).  Iterating through the arrays and updating variables takes O(n) time, since we are traversing both arrays once.  Returning the result takes constant time, O(1). |

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

Yes, I was able to solve this.

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

Browser .

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

None.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Modularity** |  | **Documentation** |  | **Indentation** |  | **Programming practices** |  |
| **Type of Algorithm:** Branch and Bound | | | | | | | |
| **Details of the algorithm:** | | | | | | | |
| The Branch and Bound algorithm is a commonly used optimization technique that involves breaking down a problem into smaller subproblems and then systematically eliminating parts of the search space that cannot contain an optimal solution.  The Branch and Bound algorithm is commonly used in combinatorial optimization problems such as integer programming, traveling salesman problem, and job scheduling problems.  The steps involved in the Branch and Bound algorithm are as follows:  Initialization: Define the problem, create an initial subproblem, and set up a priority queue.  Branch: Choose one subproblem from the queue and partition it into two or more subproblems by making an arbitrary decision about one of the variables. This creates a tree structure of subproblems.  Bound: Compute a lower bound on the objective function for each of the subproblems. If the lower bound of a subproblem is worse than the current best solution, prune the subproblem.  Update: If a subproblem has a better solution than the current best solution, update the best solution.  Add subproblems to the priority queue: Add any remaining subproblems that have not been pruned to the priority queue, ordered by their lower bounds.  Repeat: Continue branching, bounding, updating, and adding subproblems to the queue until either all subproblems have been pruned or the best solution has been found.  Termination: Return the best solution found. | | | | | | | |
| **Code for example 1:** | | | | | | | |
| #include <stdio.h>  #include <stdlib.h>  #define INF 99999999  typedef struct {  int\* weight;  int\* value;  int n;  } Item;  int bound(Item item, int W, int\* weight, int\* value, int i, int v) {  int bound = v;  int w = 0;  for (int j = i; j < item.n; j++) {  if (w + weight[j] > W) {  bound += (W - w) \* value[j] / weight[j];  break;  }  w += weight[j];  bound += value[j];  }  return bound;  }  int knapsack(Item item, int W) {  int maxprofit = 0;  int\* weight = item.weight;  int\* value = item.value;  int n = item.n;  typedef struct {  int profit;  int level;  int weight;  int bound;  int\* solution;  } Node;  Node u, v;  u.solution = (int\*)calloc(n, sizeof(int));  v.solution = (int\*)calloc(n, sizeof(int));  u.level = -1;  u.profit = 0;  u.weight = 0;  int max = 0;  int i = 0;  while (1) {  if (u.level == -1) {  v.level = 0;  } else if (u.level == n - 1) {  break;  } else {  v.level = u.level + 1;  }  v.weight = u.weight + weight[v.level];  v.profit = u.profit + value[v.level];  for (i = 0; i <= u.level; i++) {  v.solution[i] = u.solution[i];  }  v.solution[v.level] = 1;  if (v.weight <= W && v.profit > maxprofit) {  maxprofit = v.profit;  max = v.level;  }  v.bound = bound(item, W, weight, value, v.level + 1, v.profit);  if (v.bound > maxprofit) {  Node temp = u;  u = v;  v = temp;  } else {  i = u.level;  }  while (i >= 0 && (u.profit + bound(item, W, weight, value, u.level + 1, u.profit) <= maxprofit)) {  i--;  u.weight -= weight[i+1];  u.profit -= value[i+1];  }  if (i < 0) {  break;  }  u.weight -= weight[i+1];  u.profit -= value[i+1];  u.level = i;  u.solution[i] = 0;  }  printf("Selected items: ");  for (int i = 0; i <= max; i++) {  printf("%d ", u.solution[i]);  }  printf("\n");  return maxprofit;  }  int main() {  Item item;  item.n = 4;  item.weight = (int\*)malloc(item.n \* sizeof(int));  item.value = (int\*)malloc(item.n \* sizeof(int));  item.weight[0] = 2;  item.weight[1] = 3;  item.weight[2] = 4;  item.weight[3] = 5;  item.value[0] = 3;  item.value[1] = 4;  item.value[2] = 5;  item.value[3] = 6;  int W = 8;  int maxprofit = knapsack(item, W);  printf("Max profit: %d\n", maxprofit);  free(item.weight);  free(item.value);  return 0; | | | | | | | |
| **Sample Input:** | | | | | | | |
| item.weight[0] = 2;  item.weight[1] = 3;  item.weight[2] = 4;  item.weight[3] = 5;  item.value[0] = 3;  item.value[1] = 4;  item.value[2] = 5;  item.value[3] = 6; | | | | | | | |
| **Sample Output:** | | | | | | | |
| Selected items: 1 1 0 0  Max profit: 7 | | | | | | | |
| **Time complexity calculation:** | | | | | | | |
| The time complexity of the provided knapsack algorithm is O(2^n) and the total number of nodes in the tree is 2^n-1.  In addition to the branch-and-bound approach, the algorithm calculates the upper bound of the knapsack problem at each node using the bound function, which has a time complexity of O(n), where n is the number of remaining items to be considered. Therefore, the overall time complexity of the algorithm is O(2^n \* n). | | | | | | | |

|  |
| --- |
| **Code for example 2:** |
| #include <stdio.h>  #include <stdlib.h>  #include <limits.h>  #define V 4  int graph[V][V] = {{0, 10, 15, 20},  {10, 0, 35, 25},  {15, 35, 0, 30},  {20, 25, 30, 0}};  int min(int a, int b) {  return (a < b) ? a : b;  }  int tsp(int mask, int pos, int count) {  if (count == V && graph[pos][0]) {  return graph[pos][0];  }  if (count == V) {  return INT\_MAX;  }  int ans = INT\_MAX;  for (int city = 0; city < V; city++) {  if ((mask & (1 << city)) == 0 && graph[pos][city]) {  int newmask = mask | (1 << city);  int currAns = graph[pos][city] + tsp(newmask, city, count + 1);  ans = min(ans, currAns);  }  }  return ans;  }  int main() {  int ans = tsp(1, 0, 1);  printf("Minimum cost: %d\n", ans);  return 0;  } |
| **Sample Input:** |
| 4  2 10  3 20  4 30  5 40  10 |
| **Sample Output:** |
| The maximum value that can be obtained is: 70 |
| **Time complexity calculation:** |
| The function bound is called n times in the worst-case scenario, where n is the number of items. The time complexity of the bound function is O(n), which means the total time complexity of calling this function is O(n^2).  The knapsack function initializes two nodes u and v, each with an array of size n. Allocating memory for these arrays takes O(n) time.  The while loop in the knapsack function executes n + 1 times in the worst-case scenario, where n is the number of items. For each iteration of the loop, the bound function is called, which takes O(n) time. Therefore, the total time complexity of the while loop is O(n^2).  The for loop inside the while loop executes n times in the worst-case scenario, and it copies the solution from node u to node v, which takes O(n) time.\  The if statement inside the for loop checks if the weight of the new node v is less than or equal to the maximum weight W, and if the profit of v is greater than the maximum profit maxprofit. This operation takes constant time.  The bound function is called again to calculate the bound of the new node v. This operation takes O(n) time.  The if statement inside the while loop checks if the bound of node v is greater than the maximum profit maxprofit. This operation takes constant time.  The while loop inside the while loop executes at most n times. For each iteration of the loop, the bound function is called, which takes O(n) time. Therefore, the total time complexity of the while loop inside the while loop is O(n^2). |

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

Yes, I was able to solve this.

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

Browser .

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

None.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Modularity** |  | **Documentation** |  | **Indentation** |  | **Programming practices** |  |
| **Type of Algorithm:** Brute force | | | | | | | |
| **Details of the algorithm:** | | | | | | | |
| The brute force algorithm is so named because it exhaustively searches all possible solutions to a problem in a systematic manner, without using any heuristic or optimization techniques. It is also sometimes called the exhaustive search algorithm.  Applications:  Password cracking: a brute force algorithm can be used to try all possible combinations of characters until the correct password is found.  Cryptography: brute force algorithms can be used to try all possible keys until the correct one is found.  Optimization problems: a brute force algorithm can be used to try all possible solutions to an optimization problem and find the best one.  The steps involved in the brute force algorithm are:  Generate all possible solutions to the problem.  Evaluate each solution and keep track of the best one found so far.  Repeat steps 1 and 2 until all possible solutions have been evaluated. | | | | | | | |
| **Code for example 1:** | | | | | | | |
| #include <stdio.h>  #include <stdlib.h>  #define MAX\_SIZE 100  int array[MAX\_SIZE] = {-1, 2, 4, -3, 5, 2, -5, 2};  int brute\_force\_max\_subarray(int n) {  int max\_sum = 0;  for (int i = 0; i < n; i++) {  for (int j = i; j < n; j++) {  int current\_sum = 0;  for (int k = i; k <= j; k++) {  current\_sum += array[k];  }  if (current\_sum > max\_sum) {  max\_sum = current\_sum;  }  }  }  return max\_sum;  }  int main() {  int n = 8;  int max\_sum = brute\_force\_max\_subarray(n);  printf("Maximum sum of a subarray: %d\n", max\_sum);  return 0;  } | | | | | | | |
| **Sample Input:** | | | | | | | |
| n = 8  array = {-1, 2, 4, -3, 5, 2, -5, 2} | | | | | | | |
| **Sample Output:** | | | | | | | |
| Maximum sum of a subarray: 11 | | | | | | | |
| **Time complexity calculation:** | | | | | | | |
| The outer loop iterates over all possible starting indices of the subarray. The loop runs n times, where n is the size of the array.  The inner loop iterates over all possible ending indices of the subarray, starting from the current starting index. The loop also runs n times at most.  The innermost loop computes the sum of the subarray, which can have at most n elements. | | | | | | | |

|  |
| --- |
| **Code for example 2:** |
| #include <stdio.h>  #define MAX\_SIZE 10  int array[MAX\_SIZE] = {-3, 4, 6, -2, 8, -9, 1, 3, -5, 7};  int brute\_force\_max\_product(int n) {  int max\_product = -1000000;  for (int i = 0; i < n; i++) {  for (int j = i + 1; j < n; j++) {  int current\_product = array[i] \* array[j];  if (current\_product > max\_product) {  max\_product = current\_product;  }  }  }  return max\_product;  }  int main() {  int n = 10;  int max\_product = brute\_force\_max\_product(n);  printf("Maximum product of two numbers in an array: %d\n", max\_product);  return 0;  } |
| **Sample Input:** |
| Enter the number of elements in the array: 5  Enter the elements of the array: 4 5 -6 7 8 |
| **Sample Output:** |
| The maximum sum of a subsequence is 24 |
| **Time complexity calculation:** |
| We are iterating over all possible pairs of elements in the array, so the time complexity of the brute force algorithm is O(n^2), where n is the size of the array.  The outer loop iterates over the n elements of the array, so its time complexity is O(n).  The inner loop iterates over the remaining n-1, n-2, ..., 1 elements for each element in the outer loop, so its time complexity is the sum of the first n-1 integers, which is O(n^2).  Therefore, the overall time complexity of the algorithm is O(n) \* O(n^2) = O(n^3). |

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

Yes, I was able to solve this.

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

Browser .

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

None.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Modularity** |  | **Documentation** |  | **Indentation** |  | **Programming practices** |  |
| **Type of Algorithm:** Randomized | | | | | | | |
| **Details of the algorithm:** | | | | | | | |
| Randomized algorithms are named so because they incorporate randomness in their decision-making process. These algorithms make use of a source of randomness, such as a random number generator, to make certain decisions in the algorithm.  Some examples of applications include:  Monte Carlo simulations in physics and finance  Randomized algorithms for graph problems in computer science and operations research  Randomized algorithms for optimization problems in machine learning and data analysis  Randomized algorithms for cryptography and security  Randomized algorithms for numerical linear algebra and scientific computing  The steps involved in a randomized algorithm can vary depending on the specific algorithm and application. However, a common framework for designing randomized algorithms involves the following steps:  Define the problem: Clearly define the problem you want to solve using a randomized algorithm.  Choose a source of randomness: Select a source of randomness, such as a random number generator, to incorporate randomness in the algorithm.  Analyze the probability of success: Analyze the probability that the algorithm succeeds in solving the problem, as a function of the randomness used.  Implement the algorithm: Implement the randomized algorithm using the chosen source of randomness.  Analyze the complexity: Analyze the computational complexity of the algorithm in terms of the problem size and the amount of randomness used.  Test and refine: Test the algorithm on various inputs and refine the algorithm as needed to improve its performance. | | | | | | | |
| **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 = 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);  }  int randomized\_partition(int arr[], int low, int high) {  srand(time(NULL));  int random = low + rand() % (high - low);  swap(&arr[random], &arr[high]);  return partition(arr, low, high);  }  void randomized\_quicksort(int arr[], int low, int high) {  if (low < high) {  int pi = randomized\_partition(arr, low, high);  randomized\_quicksort(arr, low, pi - 1);  randomized\_quicksort(arr, pi + 1, high);  }  }  int main() {  int arr[] = {10, 7, 8, 9, 1, 5};  int n = sizeof(arr) / sizeof(arr[0]);  randomized\_quicksort(arr, 0, n - 1);  printf("Sorted array: ");  for (int i = 0; i < n; i++)  printf("%d ", arr[i]);  return 0;  } | | | | | | | |
| **Sample Input:** | | | | | | | |
| arr[] = {10, 7, 8, 9, 1, 5}; | | | | | | | |
| **Sample Output:** | | | | | | | |
| Sorted array: 1 5 7 8 9 10 | | | | | | | |
| **Time complexity calculation:** | | | | | | | |
| The time complexity of randomized QuickSort is O(n log n). Here are the steps involved in the algorithm and their time complexities:  Choose a random pivot - O(1)  Partition the array around the pivot - O(n)  Recursively sort the sub-arrays - O(log n)  Step 2 has a time complexity of O(n) because we need to scan the entire array once to partition it around the pivot.  The recurrence relation for the time complexity of randomized QuickSort is:  T(n) = 2T(n/2) + O(n)  Using the Master Theorem, we can solve this recurrence relation and get a time complexity of O(n log n). | | | | | | | |

|  |
| --- |
| **Code for example 2:** |
| #include <stdio.h>  #include <stdlib.h>  #include <time.h>  int partition(int\* arr, int left, int right, int pivot) {  int pivotValue = arr[pivot];  int temp = arr[right];  arr[right] = arr[pivot];  arr[pivot] = temp;  int storeIndex = left;  for (int i = left; i < right; i++) {  if (arr[i] < pivotValue) {  temp = arr[i];  arr[i] = arr[storeIndex];  arr[storeIndex] = temp;  storeIndex++;  }  }  temp = arr[right];  arr[right] = arr[storeIndex];  arr[storeIndex] = temp;  return storeIndex;  }  int quickselect(int\* arr, int left, int right, int k) {  if (left == right) {  return arr[left];  }  int pivotIndex = left + rand() % (right - left + 1);  pivotIndex = partition(arr, left, right, pivotIndex);  if (k == pivotIndex) {  return arr[k];  } else if (k < pivotIndex) {  return quickselect(arr, left, pivotIndex - 1, k);  } else {  return quickselect(arr, pivotIndex + 1, right, k);  }  }  int main() {  int arr[] = {5, 1, 9, 3, 7, 2, 8, 4, 6};  int n = sizeof(arr) / sizeof(int);  int k = 2;  srand(time(NULL));  int kthSmallest = quickselect(arr, 0, n - 1, k - 1);  printf("The %d-th smallest element is %d\n", k, kthSmallest);  return 0;  } |
| **Sample Input:** |
| Array: [3, 8, 2, 5, 1, 4, 7, 6]  k: 3 |
| **Sample Output:** |
| 3 |
| **Time complexity calculation:** |
| The time complexity of quickselect is O(n) in the average case and O(n^2) in the worst case, where n is the number of elements in the array. |