COMPULSORY FORMATIVE ASSESSMENT 2019/20

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **ANSWER SHEET**  (leave space on the right for comments) | | | | | | | | |
| **Assessment ID:** | | **PASS-CSP1** | | **Section:** | | **COMP1201** | | |
| Student ID: | 30824966 | | Question: | COMP1201\_1 | Page number: | | | 1 |
| (a)  The actual Java code for the given algorithm is as follows:  public int[] proc1(int[] A, int n) {  if(n > 0) {  proc1(A, n-1);  int x = A[n];  int i = n - 1;  while(i >= 0 && A[i] > x) {  A[i + 1] = A[i];  i = i - 1;  }  A[i+1] = x;  }  return A; }  The algorithm is known as *Insertion sort*.  The recurrence relation of recursive insertion sort is:  T(n) = T(n-1) + n.  It can be solved by the method of substitution and is found to be equal to n2.  The worst-case time complexity of the algorithm is Big Theta of n2 (Θ(n2)) and it occurs when the input array is sorted in reverse order (e.g. input\_array = {5, 4, 3, 2, 1}).  What the algorithm is computing: The algorithm is given as input an array and the size of the array. Then the input is sorted in ascending order (e.g. inputting {5, 4, 3, 2, 1} will output {1, 2, 3, 4, 5}).  (b)  (i)  A comparison-based sorting algorithm is a type of sorting algorithm that can only gain information about items in the input sequence - (a1, a2, …, an) - by performing pairwise-comparisons. The only requirement for applying a comparison-based sorting algorithm is that the operator forms a total preorder over the data. This means:   * If ai <= aj and aj <= az then ai <= az * For all ai and aj, ai <= aj or aj <= ai   Simply put, a sorting algorithm can be applied when the input data can be ordered. | | | | | | |  | |

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **ANSWER SHEET**  (leave space on the right for comments) | | | | | | | | |
| **Assessment ID:** | | **PASS-CSP1** | | **Section:** | | **COMP1201** | | |
| Student ID: | 30824966 | | Question: | COMP1201\_1 | Page number: | | | 2 |
| (ii)  The Java code for the given algorithm is as follows:  public int[] proc2(int[] array) {  for(int j = 0; j < array.length - 1; j++) {  int min = j;  for(int i = j + 1; i < array.length; i++) {  if(array[i] < array[min]) {  min = i;  }  }  int temp = array[j];  array[j] = array[min];  array[min] = temp;  }  return array; }  This algorithm is known as *Selection sort*.  Selection sort *is* in place and *not* stable. Its worst-case time complexity is Big Theta of n2 (Θ(n2)).  For an algorithm to be in-place, the memory used must be O(1). Selection sort is in-place because it does not require extra space in order to sort the input data. To prove that the algorithm is in-place we need to go through how the algorithm sorts its input.  First, the algorithm divides the input into two parts – sorted and unsorted. Initially, the sorted sublist is empty and the unsorted sublist is the whole input.  Then the algorithm repeatedly finds the smallest element from the unsorted part and swaps it with the leftmost unsorted element. That element is now “added” to the sorted sublist and “removed” from the unsorted sublist. Of course, both sublists do not physically exist. The whole input is divided into sublists with the use of pointers which indicate where the sublists start and end.  This means that using the Selection sort algorithm to sort an input does not require extra space in order to correctly sort the input data.  For an algorithm to be stable, the order of the elements that have the same values should not change when the data structure that they are in is sorted.  Selection sort in *not* stable. As described above, the way that Selection sort works is by swapping elements from the front of the array into the spot vacated by the minimum element. | | | | | | |  | |

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **ANSWER SHEET**  (leave space on the right for comments) | | | | | | | | |
| **Assessment ID:** | | **PASS-CSP1** | | **Section:** | | **COMP1201** | | |
| Student ID: | 30824966 | | Question: | COMP1201\_1 | Page number: | | | 3 |
| Because of the way it sorts the input, the sorted order is not correct, making Selection sort *not* stable.  An example showing than Selection sort is *not* stable:  Input array: {4, 4, 2}.  Let the two 4’s be shown as ‘a’ and ‘b’ to be able to distinguish between them. This makes the input array look like: {a, b, 2}.  If we try to sort the example input using Selection sort we get as an output:  {2, 4, 4}, which is {2, b, a}  From this example we can clearly see that, while the output is valid (it is correctly sorted), that the order of the numbers with the same values is changed. This example proves that Selection sort is *not* stable.  (iii)  We can modify Selection sort by using an implicit heap data structure. Using the heap data structure in conjunction with Selection sort, we greatly improve the base algorithm. This “new” algorithm is known as Heapsort.  Heapsort, like Selection sort, divides its input into a sorted and unsorted part. The difference is that Heapsort does not linearly scan the unsorted part. Instead, it holds the unsorted part in a heap data structure which allows the algorithm to more quickly find the sough-after element in each step.  Implementing the heap data structure in the Selection sort algorithm improves the worst-case time complexity of the algorithm from Big Theta of n2 **(Θ(n2))** to Big Theta of (n\*logn) **(Θ(n\*logn))**.  Implementing the heap data structure does not change the space complexity of the Selection sort algorithm (which is O(1)). This is because the heap data structure is not physically created but merely the data in the unsorted part of the input is rearranged into a heap, in place. | | | | | | |  | |

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **ANSWER SHEET**  (leave space on the right for comments) | | | | | | | | |
| **Assessment ID:** | | **PASS-CSP1** | | **Section:** | | **COMP1201** | | |
| Student ID: | 30824966 | | Question: | COMP1201\_1 | Page number: | | | 4 |
| (iv)  In order to improve the performance of the original Quicksort algorithm, we could:   * Use multiple pivots to partition its input into some *x* number of subarrays using *x-1* pivots.   In the original Quicksort algorithm, only a single pivot is used to partition the input into two subarrays. Using multiple (more than one) pivots improves the performance of the algorithm because it takes advantage of modern caching, and therefore demonstrates better practical performance.   * Choose the median of the first, middle and last element of the input as a pivot.   In the original Quicksort algorithm, the last element of the input sequence is selected as a pivot. But this causes worst-case behaviour on already sorted arrays.  When no information about the ordering of the input sequence is known, then setting the median of the first, middle and last element of the input as a pivot prevents the worst-case performance of having a sorted array as an input and increases the likelihood of the pivot being close to the median of the whole array. The closer the selected pivot is to the median of the input, the better performance we get. This is because when we divide the input into two parts, the closer to being equal the parts are, the faster the input array can be sorted. | | | | | | |  | |

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **ANSWER SHEET**  (leave space on the right for comments) | | | | | | | | |
| **Assessment ID:** | | **PASS-CSP1** | | **Section:** | | **COMP1201** | | |
| Student ID: | 30824966 | | Question: | COMP1201\_2 | Page number: | | | 5 |
| (a)  Kruskal’s algorithm is a minimum spanning-tree (MST) computing algorithm. It works by creating a forest in which edges are added by:   * Finding an edge with the least possible weight. * Checking that the edge does not form a cycle with the already added edges.   How Kruskal’s algorithm will create a MST with the given problem:  **Sorted edges by weight Subtrees**  1: (H, I) (H, I), (A), (B), (C), (D), (E), (F), (G), (J)  2: (D, B) (H, I), (D, B), (A), (C), (E), (F), (G), (J)  3: (A, C) (H, I), (D, B), (A, C), (E), (F), (G), (J)  4: (H, B) (H, I, D, B), (A, C), (E), (F), (G), (J)  5: (J, C) (H, I, D, B), (A, C, J), (E), (F), (G)  6: (G, H) (H, I, D, B, G), (A, C, J), (E), (F)  7: (C, D) (H, I, D, B, G, A, C, J), (E), (F)  9: (J, E) (H, I, D, B, G, A, C, J, E), (F)  10: (C, B) REJECTED  11: (E, F) (H, I, D, B, G, A, C, J, E, F) DONE  12: (A, B)  …  This is how the MST formed by Kruskal’s algorithm will look: | | | | | | |  | |

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **ANSWER SHEET**  (leave space on the right for comments) | | | | | | | | |
| **Assessment ID:** | | **PASS-CSP1** | | **Section:** | | **COMP1201** | | |
| Student ID: | 30824966 | | Question: | COMP1201\_2 | Page number: | | | 6 |
| (b)  The problem given is known as 0-1 Knapsack problem.  We are given:   * A maximum allowed weight of 7. * Four items that we can choose from (we can choose only one item from each): * (weight, value) -> (1, 1), (3, 4), (4, 5), (5,7)   We need to choose such items that will *maximise* the total value, while not going over the weight limit.  To solve this problem, we can either use the recursive approach or the bottom-up approach. For this problem, we will use the bottom-up approach.  The Java code for solving this problem using Dynamic Programming:  public class Solution {  //Returns the bigger of the two input integers.  static int getMax(int a, int b) {  return (a > b) ? a : b;  }   // Returns the maximum value  static int calculateMaxValue(int maxWeight, int weights[], int values[], int length)  {  int table[][] = new int[length + 1][maxWeight + 1];   // Build the table in a bottom up manner  for (int i = 0; i <= length; i++) {  for (int j = 0; j <= maxWeight; j++) {   if (i == 0 || j == 0)  table[i][j] = 0;   else if (weights[i - 1] <= j)  table[i][j] = *getMax*(  values[i - 1] + table[i - 1][j - weights[i - 1]],  table[i - 1][j]);  else  table[i][j] = table[i - 1][j];  }  }   return table[length][maxWeight];  } | | | | | | |  | |

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **ANSWER SHEET**  (leave space on the right for comments) | | | | | | | | |
| **Assessment ID:** | | **PASS-CSP1** | | **Section:** | | **COMP1201** | | |
| Student ID: | 30824966 | | Question: | COMP1201\_2 | Page number: | | | 7 |
| public static void main(String args[])  {  // Gives the input for solving the problem  // The input number must be sorted (ascending order)  int values[] = new int[] { 1, 4, 5, 7 };  int weights[] = new int[] { 1, 3, 4, 5 };  int maxWeight = 7;   // Calculates the maximum value  int maxValue = *calculateMaxValue*(maxWeight, weights, values, values.length);   // Displays the maximum value  System.*out*.println("Max value: " + maxValue);  } }  The output of the following code is: Max value: 9.  Clearly the output is correct because we can get the maximum value by choosing the second (3, 4) and the third (4, 5) element.  How the solution works:  We have two arrays – *values*, containing the values, and *weights*, containing the weights. The first element of the *weights* array has its value stored in the first element of the *values* array. This is valid for all the other elements in the arrays. (the second element of the *weights* array corresponds to the second element of the *values* array and so on)  In order to solve this problem using Dynamic Programming we will create a temporary array which will hold all possible *maximum* values. This is done so we can avoid recomputations of the same subproblems, thus making our algorithm better timewise. (in comparison to using a basic recursive functions).  We create a two-dimensional array in which the *columns* are all the possible weights and the *rows* are the weights given to us in our problem. | | | | | | |  | |

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **ANSWER SHEET**  (leave space on the right for comments) | | | | | | | | |
| **Assessment ID:** | | **PASS-CSP1** | | **Section:** | | **COMP1201** | | |
| Student ID: | 30824966 | | Question: | COMP1201\_2 | Page number: | | | 8 |
| The table containing the solution to our problem should look like this:  Total weight    0 1 2 3 4 5 6 7    0  1  3  4  5  0 0 0 0 0 0 0 0  0 1 1 1 *1* 1 1 1  Given weights  0 1 1 4 **5** 5 5 5  0 1 1 4 5 6 6 9  0 1 1 4 5 7 8 9  For a chosen *value* (table[i][j]), it will denote the maximum value for a given weight (j) using only the values until the i-thelement. (i-th element included).  To explain how the algorithm works we will use an example:  table[2][4] -> the element we have chosen is element ‘**5**’ (denoted in bold in the example).  What this element shows is the *maximum value* that we can get using the first two given weights – (1, 1), (3, 4) – while maintaining a maximum weight of 4. The maximum value is calculated by choosing the bigger of the two:   * table[i-1][j] which is equal to 1 * *values*[i] + table[i-1, j-i] which is equal to 4 + 1 = 5   So, the formula for populating the table from the “bottom-up” can be condensed to the following:  **Max (table[i-1][j] OR *values*[i] + table[i-1, j-i])** | | | | | | |  | |

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **ANSWER SHEET**  (leave space on the right for comments) | | | | | | | | |
| **Assessment ID:** | | **PASS-CSP1** | | **Section:** | | **COMP1201** | | |
| Student ID: | 30824966 | | Question: | COMP1201\_2 | Page number: | | | 9 |
| Note: if, when calculating the bigger of the two elements, we get an undefined value, we just choose the defined value. It is guaranteed that at least one of the values will be defined.  Also, when constructing the table, if *i* or *j* are equal to zero, we set the corresponding value in the table to zero. This is done so the formula can function properly.  When the table is constructed and populated, we just return the final element. This is the maximum value that we can obtain, given our maximum weight and items that we can choose from. Using this “bottom-up” method we get a time complexity of O(N\*W) where ‘N’ is the number of items we can choose from and ‘W’ is the maximum weight. We also use some space for constructing the array, meaning that our algorithm is not in-place. | | | | | | |  | |

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **ANSWER SHEET**  (leave space on the right for comments) | | | | | | | | |
| **Assessment ID:** | | **PASS-CSP1** | | **Section:** | | **COMP1201** | | |
| Student ID: | 30824966 | | Question: | COMP1201\_3 | Page number: | | | 10 |
|  | | | | | | |  | |

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **ANSWER SHEET**  (leave space on the right for comments) | | | | | | | | |
| **Assessment ID:** | | **PASS-CSP1** | | **Section:** | | **COMP1201** | | |
| Student ID: | 30824966 | | Question: | COMP1201\_3 | Page number: | | | 11 |
|  | | | | | | |  | |