# CS463 Project#2

|  |
| --- |
| **STUDENT NAME:** Matthew Krieger  **DATE:** March 22, 2017  For coding questions submit Java application with all classes and input and output files and/or pictures of program run outcomes in the cell for the question. |
| 1. Consider the problem of counting, in a given text, the number of substrings that start with an "A" and end with a "B". (For example, there are five such substrings in "CABAAXYAZBZXA".) Implement a brute-force algorithm in Java that accepts string and starting and ending character, and returns number of above specified substrings. Determine the efficiency class for the algorithm.   count = 0;  Analyze text from left to right;  For each character except the last{  If the (desired starting character aka “A”) is encountered{  count the number of the (desired ending characters aka “B”) following it  add the count of the previous line to the overall count of substrings  }  }  return count  public int Bruteforce\_Algorithm(T[0…n-1], startChar, endChar){  // Input: Array of n characters, starting character, ending character  // Output: Number of substrings in the string  count = 0;  for (int i = 0; i < n-1; i++){  if (T[i] == startChar){  for (int j = 0; j < n; j++){  if (T[j] == endChar){  count++;  }  }  }  }  return count;  )  Cworst (n) =  *=*  *=*  *= (n + n – 1 + n – 2 + … + 3 + 2)*  *= (n + n – 1 + n – 2 + … + 3 + 2 + 1) – 1*  *= - 1*  ∈ θ(n2) |
| 2) There are several alternative ways to define a distance between two points P1 = (x1, y1) and P2=(x2,y2) in the Cartesian plane. The Manhattan distance is defined as:  DistManhattan (P1,P2) = |x1-x2| + |y1-y2|  Sketch all the points in the x,y coordinate plane whose Manhattan distance to the origin (0,0) is less or equal to 7.  *../Library/Containers/com.apple.mail/Data/Library/Mail%20Downloads/CB011657-DE98-4483-B5D6-8112F58B0507/2.pdf* |
| 3) Sketch all the points in the x,y coordinate plane whose Euclidian distance to the origin (0,0) is equal to 7.  DistEuclid(P1,P2) = ( (x1-x2)2 + (y1-y2)2 )1/2  ***SEE ANSWER FOR #2*** |
| 4) Trace DFS algorithm on page142 ex 1a) Start from node d. Assign to each node a pair of numbers. The first number should be the order when that node is first visited (or put on the stack), and the second number is order when that node is finished (or removed from the stack). In addition, mark the directed edges with T, B, F, or C (tree, back, forward or cross). Specify if topological ordering based on DFS is possible. If YES provide the topological order of nodes. |
| 5) Trace the algorithm to produce topological sorting based on source removal for the graph on page142 ex 1a). The trace should show the picture of the graph after each step (each deletion). List resulting order of vertices for this topological sorting algorithm.  ../../Library/Containers/com.apple.mail/Data/Library/Mail%20Downloads/C74606D3-606A-4868-9094-4C7883B36EDE/5.pdf |
| 6-7) How many comparisons (both successful and unsuccessful) will be made by the brute-force String matching algorithm (page 105) in searching for each of the following patterns in the binary text of ten thousand zeros? Show work how you calculated result. A number only is not sufficient as an answer.  n – m + 1 = 10000 – 4 + 1 = 9997 tries  6) 0001  For the pattern “0001”, there will be 3 successful comparisons (“0”) , and 1 unsuccessful comparisons (“1”). After that, the pattern will shift 1 position to the right.  0 0 0 0 0  0 0 0 1  0 0 0 1  etc…  Cumulative Number of Character Comparisons: C = 4 x 9997 = ***39988 comparisons***    7) 1000  0 0 0 0 0  1 0 0 0  1 0 0 0  etc…  For the pattern “1000”, there will be 1 unsuccessful comparison (“1”) for each trial. After that, the pattern will shift 1 position to the right.  Cumulative Number of Character Comparisons: C = 1 x 9997 = **9997 comparisons** |
| 8-9) Consider the following small instance of linear programming problem: maximize 3x+2y subject to the following constraints: x + y <= 8, y - x <= 0, x >= 0, y >= 0  8) Sketch, in the Cartesian plane, the problem's feasible region, defined as a set of points satisfying all the problem's constraints. Use grey color to indicate feasible region.  ../../Library/Containers/com.apple.mail/Data/Library/Mail%20Downloads/419138CB-8911-4C31-B769-52518DBF7566/8%20%209.pd  9) Solve the optimization problem specified above. Use the following theorem: a linear programming problem with a nonempty bounded feasible region always has a solution, which can be found at one of the extreme points of its feasible region.  **SEE ANSWER FOR #8** |
| 10) (2 points ) Consider the sum-of-subsets problem: Given n integers, partition them into two disjoint subsets with the same sum of their elements. (Of course, the problem doesn't always have a solution.) Design a brute-force algorithm for this problem. Input n integers from an input file named "indata.txt" and resulting partitions (or reporting that problem has no solution) should be sent to an output file named "outdata.txt".  Test program with the input file with integers: {1, 2, 3, 4, 6, 7, 8, 9} Hint: The brute-force approach would compute the sum of the elements in each subset and compare it with the sum of the elements in its complement.  *// input: n positive integers*  *// output: Sums*  */\*Subset(n)*  *S = 0;*  *For (int i=0; i<n;i++){*  *S = S + a[i];*  *If (S mod 2 == 1){*  *Stop because problem has no solution*  *}*  *else {*  *Generate subsets until there are no more subsets left or a subset whose elements’ sum is S/2*  *}*  *}*  *\*/*  import java.io.\*;    class sumOfSubsets  {      static boolean isSubsetSum (int arr[], int n, int sum)      {          // Base Cases          if (sum == 0)              return true;          if (n == 0 && sum != 0)              return false;            // If last element > sum, ignore it          if (arr[n-1] > sum)              return isSubsetSum (arr, n-1, sum);            /\* else, check if sum can be obtained by any of             the following          (a) including the last element          (b) excluding the last element          \*/          return isSubsetSum (arr, n-1, sum) ||                 isSubsetSum (arr, n-1, sum-arr[n-1]);      }        // Returns true if arr[] can be partitioned in two      // subsets of equal sum, otherwise false      static boolean findPartition (int arr[], int n)      {          // Calculate sum of the elements in array          int sum = 0;          for (int i = 0; i < n; i++)              sum += arr[i];            // If sum is odd, there cannot be two subsets          // with equal sum          if (sum%2 != 0)              return false;            // Find if there is subset with sum equal to half          // of total sum          return isSubsetSum (arr, n, sum/2);      }        public static void main (String[] args)      {            int arr[] = {1, 2, 3, 4, 6, 7, 8, 9};          int n = arr.length;          if (findPartition(arr, n) == true){              System.out.println("Can be divided into two "+                                  "subsets of equal sum");  }          else {              System.out.println("Can not be divided into " +                                  "two subsets of equal sum");  )      }  } |
| 11) EXTRA CREDIT Improve the example 10) algorithm in the following way. Do not generate all the subsets. Instead, generate only subsets of size ⎣ n/2⎦ and check whether the sum of the elements in a subset is equal to totalSum/2 where totalSum is the sum of all given input numbers. (Of course, it will still be an untractable algorithm but the amount of work will be cut in half.)  import java.io.\*;    class sumOfSubsets  {      static boolean isSubsetSum (int arr[], int n, int sum)      {          // Base Cases          if (sum == 0)              return true;          if (n == 0 && sum != 0)              return false;            // If last element > sum, ignore it          if (arr[n-1] > sum)              return isSubsetSum (arr, n-1, sum);            /\* else, check if sum can be obtained by any of             the following          (a) including the last element          (b) excluding the last element          \*/          return isSubsetSum (arr, n-1, sum) ||                 isSubsetSum (arr, n-1, sum-arr[n-1]);      }        // Returns true if arr[] can be partitioned in two      // subsets of equal sum, otherwise false      static boolean findPartition (int arr[], int n)      {          // Calculate sum of the elements in array          int sum = 0;          for (int i = 0; i < n; i++)              sum += arr[i];            // If sum is odd, there cannot be two subsets          // with equal sum          if (sum%2 != 0)              return false;            // Find if there is subset with sum equal to half          // of total sum          return isSubsetSum (arr, n, sum/2);      }        public static void main (String[] args)      {            int arr[] = {1, 2, 3, 4, 6, 7, 8, 9};          int n = arr.length;          if (findPartition(arr, n) == true){              System.out.println("Can be divided into two "+                                  "subsets of equal sum");  }          else {              System.out.println("Can not be divided into " +                                  "two subsets of equal sum");  )      }  } |