# CS463 Spring 2017 Project#1

|  |
| --- |
| **STUDENT NAME: Matthew A. Krieger DATE: February 20, 2017**  Project should be submitted via Blackboard Learn P1 assignment tool. Do not forget to press submit button at the end. Modify this MS word document by including your solutions and rename it to P1cs463YourlastName.docx. Keep this table with questions and insert your answers in the same cell where the question is specified. For coding questions insert java code for all classes in "Courier" or "Courier New" font and insert also input and output files, if any, and pictures of program run with desired outcomes. |

|  |  |
| --- | --- |
| 1) Consider the following algorithm for finding the distance between the two closest elements in an array of numbers. The pseudocode notation is the same as the notation used in the textbook except that assignment uses = symbol instead of an arrow. Make as many improvements as you can in the given algorithmic solution to the problem   |  | | --- | | ALGORITHM MinDistance (A[0..n-1])  // Input: Array A[0..n-1] of numbers  //Output: Minimum distance between two of its elements Distance is defined as absolute value of their difference)  dMin = MaxInt  for I =0 to n-1 do  for j = 0 to n-1 do  if i != j and | A[i] – A[j] | <dMin  dMin = |A[i] – A[j]|  return dMin |   .  **My Solution:**  /Scanner library allowing the user to input data import java.lang.Math.\*;  public class ArrayTester{ //algorithm for finding the distance between the two closest elements in an array of numbers  public int MinDistance(int [] ar){  int [] a = ar;  int aSize = a.length;  int dMin = 0;//MaxInt  for(int i=0; i< aSize; i++)  {  for(int j=i+1; j< aSize;j++)  {  dMin = Math.min(dMin, Math.abs( a[i]-a[j] );  )  }  return dMin;  } |
| 2) Design an application for computing floor(squareRoot(n)) for any positive integer n. Besides assignment and comparison, your algorithm may only use the four basic arithmetic operations (+, -, \*, /).   |  | | --- | | public class Question2  {  public static int floorSqrt(int x)  {  if (x == 0 || x == 1)  {  return x;  }  int start = 1, end = x, solution=0;  while (start <= end)  {  int mid = (start + end) / 2;  if (mid\*mid == x)  return mid;  // Since we need floor, we update answer when mid\*mid is  // smaller than x, and move closer to sqrt(x)  if (mid\*mid < x)  {  start = mid + 1;  solution = mid;  }  else // If mid\*mid is greater than x  end = mid - 1;  }  return solution;  }  // Test Method  public static void main (String args[])  {  int x = 16;  System.out.println(floorSqrt(x));  }  } | |  | |
| 3) Four persons p1, p2, p5 and p10 require 1, 2, 5, 10 minutes, respectively, to get to the other side of the bridge. Solve example 2 on page 17. For each step specify set of persons on the left bank, set of persons on the right bank before crossing, who is crossing, and which way LeftToRight or RightToLeft.  *Step 1:*   * Left Bank: p1, p2, p5, p10 & Flashlight * Right Bank: Nobody * p1 & p2 : LeftToRight * Time Used: 2 minutes   *Step 2:*   * Left Bank: p5, p10 * Right Bank: p1, p2, flashlight * p2: RightToLeft * Time Used: 2 minutes   *Step 3:*   * Left Bank: p2, p5, p10, flashlight * Right Bank: p1 * p5 & p10: LeftToRight * Time Used: 10 minutes   *Step 4:*   * Left Bank: p2 * Right Bank : p1, p5, p10, flashlight * P1: RightToLeft * Time Used: 1minutes   *Step 5:*   * Left Bank: p1, p2, flashlight * Right Bank: p5, p10 * P1 & p2: LeftToRight * Time Used: 2 minutes   *Total Time: 17 minutes* |
| 4) Give an example of a problem of your choice. Provide two different algorithms to solve the problem. Use Java to implement the two algorithms. (Searching, sorting, and GCD are excluded.)  Tower of Hanoi – Iterative vs Recursive   |  | | --- | | import java.lang.Math;  public class Hanoi {    static int n; //number of disks    public static void main (String[] args) {  n = Integer.parseInt(args[0]);  int limit = (1 << n) - 1;  for (int i = 0; i < limit; i++) {  int d = disk(i); //disk to be moved  int source = (movements(i,d)\*direction(d))%3;  int dest = (source + direction(d))%3;  }  }  static int disk(int i) {  int g, x = i+1;  for (g = 0; x%2 == 0; g++) x /= 2;  return g;  }    //how many times disk d is moved before stage i  static int movements(int i, int d) {  return ((i >> d) + 1) >> 1;  }    static int direction(int d) {  return 2 - (n + d)%2;  }    static void out(int d, int source, int dest) {  System.out.println("Moving disk " + d + " from tower " + source + " to tower " + dest);  }  }  //--------------------------------------------------------------------------------  public class TowersOfHanoi {     public void solve(int n, String start, String auxiliary, String end) {        if (n == 1) {            System.out.println(start + " -> " + end);        } else {            solve(n - 1, start, end, auxiliary);            System.out.println(start + " -> " + end);            solve(n - 1, auxiliary, start, end);        }    }     public static void main(String[] args) {        TowersOfHanoi towersOfHanoi = new TowersOfHanoi();        System.out.print("Enter number of discs: ");        Scanner scanner = new Scanner(System.in);        int discs = scanner.nextInt();        towersOfHanoi.solve(discs, "A", "B", "C");    } } | |
| 5) Design and implement in Java a recursive algorithm for computing 2n for any nonnegative integer n based on the formula 2n= 2n-1 + 2n-1. Perform run time analysis. Use addition as basic operation. Set up a recurrence relation for the number of additions made by the algorithm and solve it. Specify the asymptotic run time performance as big theta.  //Computes 2n recursively by the formula 2n =2n-1 +2n-1  //Input: A nonnegative integer n  //Output: Returns 2n if n =0 return 1 else return Power(n−1) + Power(n−1)   |  | | --- | | public class Problem5 {  public static int PowerOfTwo(int n){  if (n == 0){  return 1;  }  else {  return PowerOfTwo(n-1) + PowerOfTwo(n-1);  }  }  public static void main(String[] args) {  int n = 2;  System.out.println(PowerOfTwo(n));  }  } |   Let C(n) denote the number of additions performed by the above algorithm when given an input n > -1. If the condition in the if statement is true then 0 additions are performed. Otherwise 1 addition is performed.  Recurrence Base Case:  T(0) = 0  T(n) = 2T(n-1) + 1, n>1  2(2T(n-2)+1)+1  22T(n-2)+2+1  22(2T(n-3)+1)  23T(n-3)+22+2+1  2kT(n-k)+2k-1 + 22+2+1  2kT(n-k)+  2nT(n-n) + 2n-1  2nT(0) + 2n -1  2n-1  T(n) => exponential and its in Θ(2n) |
| 6) Provide different algorithm to calculate 2n that is the most efficient, and provide its time analysis. Assume that n is of the form 2k.   |  | | --- | | public class Problem6 {  public static double PowerOfI(double i,int n){  if (n == 0){  return 1;  }  else if (n == 1){  return i;  }  double temp = PowerOfI(i, n/2);  if (n % 2 == 1){  return i \* temp \* temp;  }  else {  return temp \* temp;  }  }  public static void main(String[] args) {  int n = 2;  int i = 2;  System.out.println(PowerOfI(i,n));  }  } |   T(0) = 0  T(1) = 0  T(n) = T(n/2) + 1, n> 1  n = 2k  k = lg n  T(2k) = T(2K-1) + 1  (T(2k-2)+1) +1 = T(2k-2) + 2  (T(2k-3)+1)+2 = T(2k-3) + 3  …  T(2k-k) + k = T(20) + k  T(1) +k  T(2k) = k  T(n) = lg n  T(n) = Θ(log n)  Log running time |
| For the algorithms 7) -10) specify the following:   1. what is input, 2. what is output, 3. what algorithm does, 4. provide code or pseudo code, 5. provide detailed algorithm time performance analysis.  * For recursive algorithms time performance analysis specify problem size n, basic operation, and recurrences, and then solve the recurrences to determine T(n). * For iterative algorithms time performance analysis specify problem size n, basic operation, and determine how many times is basic operation performed as a function of problem size n.   Algorithms analyzed in the textbook, textbook supplement, and class examples provided by the instructor are excluded. |
| 7) Give an example of an iterative algorithm that has Ө(n2) running time. Provide detailed time performance analysis.   |  | | --- | | Algo(A[0…n-1,0…n-1]  {  for (int i = 0 to (n-2))  {  for (int j = (i+1) to (n-1))  {  if (A[i,j] != A[j,i]  {  return false;  }  }  }  return true;  } | |  |   Algorithm – determines if matrix is symmetric or assymetric  Basic Operation: comparison on 2 matrix elements  Input: Matrix  Output: true: symmetric / false: assymetric  Cworst(n) =        (n-1)  (n-2)2 – ((n-2)(n-1) / 2 )  (n(n-1)) / 2  ½ n2 ∈ Θ(n2) |
| 8) Give an example of either iterative or recursive algorithm that has Ө( log n) running time. Give detailed time performance analysis.  Practice(int n)  {  if (n == 1)  {  return 1;  }  else  {  return Practice([n/2]) + 2  }  }  Algorithm – # of digits in binary version  Basic Operation: addition  Input: + integer  Output: true: # of digits in binary version  A(2K) = A(2k-1) +1, k>0  A(20) = 0  A(2k) = A(2k-1) + 1  A(2K-2)+1 + 1 =  A(2K-2)+2  [A(2k-3)+1]+2 = A(2K-3+3)  ...  =A(2k-i)+i  ...  A(2K-K)+k  A(2k) = A(1) +K = K  N = 2k  K = log2n  A(n) = log2n∈Θ(log n) |
| 9) Give example of an algorithm that has Ө(2n) running time. Use recursive approach. Give detailed time performance analysis. (recurrence relation)  Han(int N, string from\_peg, string to\_peg, string spare\_peg)  {  if (N < 1){  return;  }  if (N> 1){  Han(N-1, from\_peg, spare\_peg, to\_peg)  }  print ”move from ” + from\_peg + ” to” + to\_peg  if (N> 1){  {  Han(N-1, spare\_peg, to\_peg, from\_peg)  }  }  Algorithm: prints all the moves necessary to solve the famous “Towers of Hanoi” problem for n disks  Basic Operation: print for disk move  Size: n  T(1) = 1  T(n) = 2T(n-1)+1  T(n) = 2(2T(n-2)+1)  T(n) = 2^2(2T(n-2)+2+1  T(n) = 22(2T(n-3)+1 = 23T(n-3)+22+2+1  T(n) = 2kT(n-k)+2k-1+…2+1  T(n) = 2 2n-1-1  T(n) = 2n-1  T(n) is in Θ(2n) |
| 10) List the following functions based on their asymptotic order of growth from the lowest to highest. Those that have the same asymptotic order of growth should be listed in the same line. Those that have lower asymptotic growth should be above those with higher growth. For each line provide simplest representative with same asymptotic growth.  ~~2n lg n~~ , ~~n log~~~~3~~ ~~n~~, ~~log n~~~~2~~ , ~~n!~~, ~~3n~~~~2~~ ~~+ 7n~~, ~~8 n~~~~7~~ ~~– 300n~~~~3~~, ~~2~~~~n~~, ~~5~~~~n~~, ~~n~~~~2~~, ~~n~~~~5~~~~+ 2~~~~n~~, ~~3n + 3n~~~~2~~ ~~+ 2~~~~n~~, ~~log~~~~7~~ ~~n~~, ~~7 log~~~~3~~ ~~n~~~~4~~.  limn🡪∞(n\*log3 n) / (n \* log n) = 1 / ln(3) = 0.91024 🡺 n log3 n ∈ Θ(n log n)  limn🡪∞(2n\*log n) / (n \* log n) = 2 🡺 2n log n ∈ Θ(n log n)  limn🡪∞(log(n2) / (log(n)) = 2 🡺 log(n2) ∈ Θ(log(n))  limn🡪∞(n!) / (n!) = 1 🡺 n! ∈ Θ(n!)  limn🡪∞(3n2 + 7n) / (n2) = 3 🡺 3n2 + 7n ∈ Θ(n2)  limn🡪∞(8n7-300n3) / (n7) = 8 🡺 8n7-300n3 ∈ Θ(n7)  limn🡪∞(2n) / (2n) = 1 🡺 2n ∈ Θ(2n)  limn🡪∞(5n) / (5n) = 1 🡺 5n ∈ Θ(5n)  limn🡪∞(n2) / (n2) = 1 🡺 n2 ∈ Θ(n2)  limn🡪∞(n5 + 2n) / (2n) = 1 🡺 n5 + 2n ∈ Θ(2n)  limn🡪∞(3n + 3n2 + 2n) / (2n) = 1 🡺 3n + 3n2 + 2n ∈ Θ(2n)  limn🡪∞(log7n / (log n) = 1 / ln(7) = 0.51390 🡺 log7n ∈ Θ(log n)  limn🡪∞(7\*log3n4 / (log n) = 28 / ln(3) = 25.48670 🡺 7\*log3n4 ∈ Θ(log n)  log(n) (logarithmic): log7(n)  log(n) (logarithmic): log(n2)  log(n) (logarithmic): 7\*log3n4  n log n (linearithmic): (2n\*log n)  n log n (linearithmic): (n log3n)  n2 (quadratic): (n2)  n2 (quadratic): (3n2 + 7n)  n7 : (8n7 – 300n3)  2n (exponential): (2n)  2n (exponential): (3n + 3n2 + 2n)  2n (exponential): (n5 + 2n)  5n (exponential): (5n)  n! (factorial): (n!)  OR(Not sure how you want it)  log(n) 🡺log7n, log(n2), 7\*log3n4  n log n 🡺 (2n\*log n), (n log3n)  n2 🡺 (n2) , (3n2 + 7n)  n7 🡺 (8n7 – 300n3)  2n 🡺 (2n), (3n + 3n2 + 2n), (n5 + 2n)  5n 🡺 (5n)  n! 🡺 (n!) |