P1PartA

n=100

Input value x = 100

Sum-of-squares y1 = 338350

Divide-last y2 = 338350

Divide-early y3 = 338350

n=500

Input value x = 500

Sum-of-squares y1 = 41791750

Divide-last y2 = 41791750

Divide-early y3 = 41791750

n=1000

Input value x = 1000

Sum-of-squares y1 = 333833500

Divide-last y2 = 333833500

Divide-early y3 = 333833500

n=1500

Input value x = 1500

Sum-of-squares y1 = 1126125250

Divide-last y2 = -305530515

Divide-early y3 = 1126125250

n=2000

Input value x = 2000

Sum-of-squares y1 = -1626300296

Divide-last y2 = -194644530

Divide-early y3 = -1626300296

n=60000

Input value x = 60000

Sum-of-squares y1 = 968259856

Divide-last y2 = 252431973

Divide-early y3 = 368249856

For each of the three functions, what is the *largest* input that gives the *correct* output?

n=1000 is the last correct of divide last

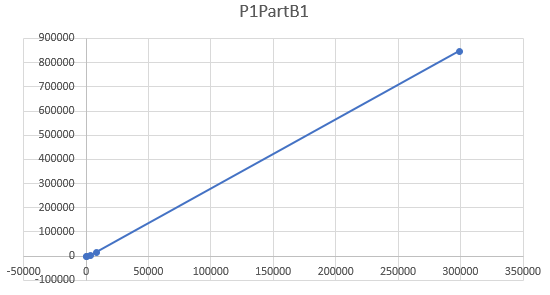
n=1500 is the last correct of sum of squares and divide early

Which algorithm is "best"? Why?

Sum of squares and divide early are the best. They are both equally accurate. I think sum of squares is an easier algorithm so it would be easiest.

**// Program to calculate the sum of squared integers  
// using 3 algorithms.  
public class P1PartA  
{  
 public static void main(String args[])  
 {  
 int x, y1, y2, y3;  
 if(args.length >= 0){  
 x = Integer.parseInt(args[0]);  
 System.out.println("\n Input value x = " + x);  
 if (x >= 1){  
 // Sum-of squares algorithm  
 y1 = f1(x);  
 System.out.println(" Sum-of-squares y1 = " + y1);  
 // Divide-last algorithm  
 y2 = f2(x);  
 System.out.println(" Divide-last y2 = " + y2);  
 // Divide-early algorithm  
 y3 = f3(x);  
 System.out.println(" Divide-early y3 = " + y3);  
 }  
 }  
 return;  
 } // end main  
  
 private static int f1(int x)  
 {  
 // Sum-of-squares algorithm  
 int y = 0;  
 for (int k = 1; k <= x; k++){  
 y = y + k\*k;  
 }  
 return y;  
 }  
  
 private static int f2(int x)  
 {  
 // Divide-last algorithm  
 int y = 0;  
 y = x\*(x + 1)\*(2\*x + 1)/6;  
 return y;  
 }  
  
 private static int f3(int x)  
 {  
 // Divide-early algorithm  
 int y = 0;  
 int A = x\*(x+1)/2;  
 int B = 2\*x + 1;  
 if(A % 3 == 0)  
 y = (A/3)\*B;  
 else  
 y = A\*(B/3);  
 return y;  
 }  
  
} // end class**

P1PartB1

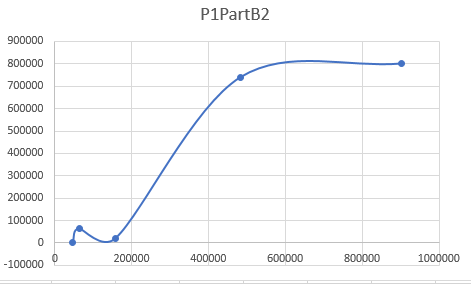


This function **is** obviously one to one. It will never have two of the same values.

|  |  |
| --- | --- |
| X = 6 | Y = 12 |
| X = 277 | Y= 278 |
| X = 3443 | Y= 3768 |
| X = 8128 | Y= 16256 |
| X = 298760 | Y= 846720 |

**// Program to calculate the sum of the divisors  
// of a positive integer.  
public class P1PartB  
{  
 public static void main(String args[])  
 {  
 long X, Y;  
 if(args.length >= 1){  
 X = Long.parseLong(args[0]);  
 if(X >= 1){  
 Y = SumOfDivisors(X);  
 System.out.println("\n X = " + X + " Y = " + Y);  
 }  
 }  
 return;  
 }  
  
 public static long SumOfDivisors(long x)  
 {  
 long y = 1;  
   
   
  
 for(int i = 2;i \* i <= x; ++i)  
 {  
 int p = 1;  
   
 while(x % i == 0)  
 {  
 p = p \* i + 1;  
 x /= i;   
 }  
  
 y \*= p;  
 }  
  
 if(x > 1)   
 y \*= 1 + x;  
   
   
 return(y);   
 }  
 }**

P1PartB2

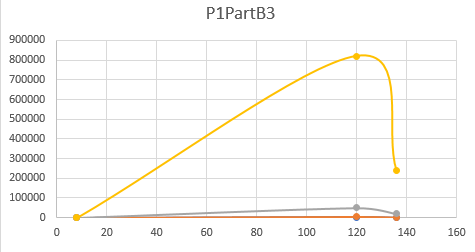


This function **is not** one to one. You can see that there are many points that will have the same value.

|  |  |
| --- | --- |
| X = 45600 | Y= 654 |
| X = 62826 | Y= 62826 |
| X = 157910 | Y= 19751 |
| X = 483047 | Y= 740384 |
| X = 901108 | Y= 801109 |

**public class P1PartB2  
{  
 public static void main(String args[])  
 {  
 long X, Y;  
 if(args.length >= 1){  
 X = Long.parseLong(args[0]);  
 if(X >= 1){  
 Y = ReverseNumber(X);  
 System.out.println("\n X = " + X + " Y = " + Y);  
 }  
 }  
 return;  
 }  
  
 public static long ReverseNumber(long x)  
 {  
 long y =0;  
   
 while( x != 0 )  
 {  
 y = y \* 10;  
 y = y + x%10;  
 x = x/10;  
 }  
  
 return(y);   
 }  
 }**

P1PartB3

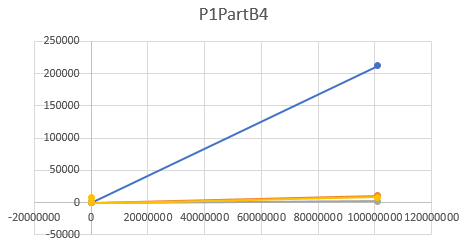


This function **is not** one to one. You can see there are many points that will have the same value.

|  |  |  |
| --- | --- | --- |
| X1 = 136 | X2 = 120 | Y = 8 |
| X1 = 512 | X2 = 2217 | Y= 1 |
| X1 = 1595 | X2 = 4255 | Y= 5 |
| X1 = 18684 | X2 = 49878 | Y= 18 |
| X1 = 237732 | X2 = 819918 | Y= 66 |

**public class P1PartB3  
{  
 public static void main(String args[])  
 {  
 long X1, X2, Y;  
 X1 = Long.parseLong(args[0]);  
 X2 = Long.parseLong(args[1]);  
   
 Y = GreatesCommonDivisor(X1,X2);  
 System.out.println("\n X1, X2 = " + X1 + ", " + X2 + " Y = " + Y);  
   
   
 return;  
 }  
  
 public static long GreatesCommonDivisor(long x1,long x2)  
 {  
 long y = 1;  
 long i = 0;  
   
 for (i = 1;i<= x1 && i <= x2; i++)  
 {  
 if (x1 % i == 0 && x2 % i == 0)  
 {  
 y =i;  
 }  
 }   
 return(y);  
 }  
}**

P1PartB4

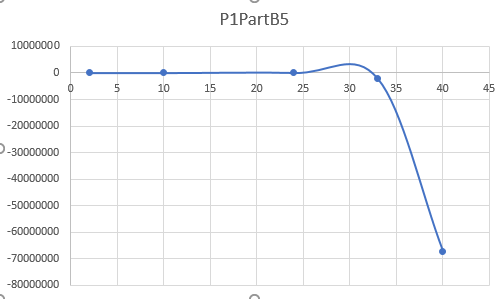


This function **is** obviously one to one. It is never going to have the same value more than once.

|  |  |  |
| --- | --- | --- |
| X = 359 | B = 2 | Y = 101100111 |
| X = 632 | B = 3 | Y = 212102 |
| X = 757 | B = 5 | Y = 11012 |
| X = 981 | B = 8 | Y = 1725 |
| X = 8046 | B = 10 | Y = 8046 |

**public class P1PartB4  
{  
 public static void main(String args[])  
 {  
 long X, B, Y;  
 X = Long.parseLong(args[0]);  
 B = Long.parseLong(args[1]);  
   
 Y = BaseB(X,B);  
 System.out.println("\n X, New Base = " + X + ", " + B + " Y = " + Y);  
   
   
 return;  
 }  
  
 public static long BaseB(long x,long b)  
 {  
 long y = 0, z = 0, i = 0;  
 double d = 0;  
   
 while (x != 0)  
 {  
 i++;  
 z = x%b;  
 x = x/b;  
 d = d + (z\*Math.pow(10,i -1));  
 }  
   
   
 y = (new Double(d)).longValue();  
   
  
 return(y);   
  
 }  
}**

P1PartB5



This function **is not** one to one. You can see that there is a small blip is the graph showing that there are going to be a few spots with the same value.

|  |  |
| --- | --- |
| X = 2 | Y = -2 |
| X = 10 | Y = -64 |
| X = 24 | Y = 32768 |
| X = 33 | Y = -2097152 |
| X = 40 | Y = -67108864 |

**public class P1PartB5  
{  
 public static void main(String args[])  
 {  
 long X, Y;  
 X = Long.parseLong(args[0]);  
   
 Y = Recursion(X);  
 System.out.println("Y= " + Recursion(X));  
   
   
 return;  
 }  
  
  
 public static long Recursion2(long x)  
 {  
   
 if (x <= 1)  
 return 1;  
 else  
 return (2\* Recursion(x-1));  
   
  
 }  
   
 public static long Recursion(long x)  
 {  
   
 if (x <= 1)  
 return 1;  
 else  
 return (-2\*Recursion2(x-2));  
   
  
 }  
  
}**