1.**Write a function that will iterate through an array *a* as follows.** Start at a[0]. If a[0] is -1 return -1. If a[0] is less than -1 or greater than or equal to the length of the array (i.e., it can't be used to index an element of the array), return 1. Otherwise visit a[a[0]] and repeat these steps. This could potentially result in an infinite loop. If an infinite loop is detected the function should return a 0.

To summarize:

1. iterate through the array using the value of an element as the index to the next element (like in a linked list)
2. return -1 if a -1 encountered
3. return 1 if a value less than -1 or greater than or equal to the size of the array is encountered.
4. return 0 if an infinite loop is detected.

If you are programming in Java or C#, the function signature is int isInfinite(int[ ] a)

If you are programming in C or C++, the function signature is int isInfinite(int a[ ], int len) where len is the number of elements in the array

Examples

if the input array is	traversal	return
{1, 2, -1, 5}	visit a[0], a[1], a[2]	-1 (because -1 is encountered before the 5 is encountered)
{1, 2, 4, -1}	visit a[0], a[1], a[2]	1 (because 4, which is too big to be an index, is encountered before the -1)
{5, 3, 4, -1, 1, 2}	visit a[0], a[5], a[2], a[4], a[1], a[3]	-1 (because a[3] is -1)
{3}	visit a[0]	1 (because 3, which is too big to be an index, is encountered.)
{3, 2, 3, 1}	visit a[0], a[3], a[1], a[2], a[3], ...	0
{0}	visit a[0], a[0], ...	0
{-1}	visit a[0]	-1

2. Define an array to be *cumulative* if the nth (n > 0) element of the array is the sum of the first n elements of the array. So {1, 1, 2, 4, 8} is cumulative because

1. a[1] == 1 == a[0]
2. a[2] == 2 == a[0] + a[1]
3. a[3] == 4 == a[0] + a[1] + a[2]
4. a[4] == 8 == a[0] + a[1] + a[2] + a[4]

And {1, 1, 2, 5, 9} is not cumulative because a[3] == 5 != a[0] + a[1] + a[2]

Write a function named isCumulative that accepts an array of integers and returns 1 if the array is cumulative and 0 otherwise.

If you are programming in Java or C#, the function signature is int isCumulative(int[ ] a)

Some other examples:

if the input array is	isCumulative should return
{1}	0 (array must contain at least 2 elements)
{0,0,0,0,0,0}	1
{1, 1, 1, 1, 1, 1}	0
{3, 3, 6, 12, 24}	1
{-3, -3, -6, -12, -24}	1
{-3, -3, 6, 12, 24}	0

```
public static void main (String []args){

            System.out.println(isCommulative(new int[]{1, 1, 2, 4, 8}));
            System.out.println(isCommulative(new int[]{}));

      }
  public static int isCommulative(int a[]){

        if (a.length<=1)
              return 0;

        for (int i=1;i<a.length;i++){
              int sum=0;
              for (int j=0;j<i;j++) {

                          sum+=a[j] ;

                                                  }

              if (sum != a[i])
                    return 0;

                                  }
return 1;


  }
```

3. **Write a function that takes two arguments, an array of integers and a positive, non-zero number n. It sums n elements of the array starting at the beginning of the array.** If n is greater than the number of elements in the array, the function loops back to the beginning of the array and continues summing until it has summed n elements. You may assume that the array contains at least one element and that n is greater than 0.

If you are programming in Java or C#, the function signature is int loopSum(int[ ] a, int n)

Examples

If a is	and n is	then function returns
{1, 2, 3}	2	3 (which is a[0] + a[1])
{-1, 2, -1}	7	-1 (which is a[0] + a[1] + a[2] + a[0] + a[1] + a[2] + a[0])
{1, 4, 5, 6}	4	16 (which is a[0] + a[1] + a[2] + a[3])

```
public static void main (String args[]){

      System.out.println(f((new int[]{1,2,3}),8));


}
      public static int f(int a[],int n){
            int sum=0;

            if (a.length >= n){
                  for (int i=0;i<n;i++){
                        sum+=a[i];
                                    }

                                    }
            else
                              {
                  int times=n/a.length;
                  int remainder=n % a.length;
                    for (int y=0;y<a.length;y++)
                          sum+=a[y];
                    sum=sum * times;
                    for (int z=0;z < remainder;z++)
                          sum+=a[z];

                                    }

             return sum;
      }
```

4 Write a function **int hasNFollowingComposites(int n, int count)** that returns 1 if *n* is a prime number and the next *count* numbers are composite(non-prime). Otherwise, it returns 0. Recall that a prime number is a number whose only factors are 1 and itself. You may assume that *n* and *count* are greater than zero.

Examples:

if n is	and count is	Return
23	5	1 because 23 is prime and the next 5 numbers, 24, 25, 26, 27 and 28 are composite

23	6	0 because 23 is prime but the 6th number following it (29) is prime, i.e., the next six numbers are not **all** composite.
89	6	1 because 89 is prime and the next 6 numbers, 90, 91(13*7), 92, 93(31*3), 94, 95 are composite.
24	4	0 because 24 is **not** prime (it doesn't matter that the next 4 numbers are composite)

```
public static void main(String args[]){

		System.out.println(f(23,5));
	}


	public static int f(int n,int count){
		if ((isPrime(n)==false) || (n==0))
			return 0;

		else
			for(int i=n+1; i <= n + count; i++){
				if (isComposite(i)==false)
					return 0;
			}

		return 1;

	}
	public static boolean isPrime(int number){
		for (int i=2;i<number;i++){
			if (number % i == 0)
				return false;
		}
		return true;

	}
	public static boolean isComposite(int number){
		for (int i=2;i<number;i++){
			if (number % i == 0)
				return true;
		}
		return false;
}
```

5, Write a function named **equivalentArrays** that has two array arguments and returns 1 if the two arrays contain the same values (but not necessarily in the same order), otherwise it returns 0. Your solution must **not** sort either array or a copy of either array! You may assume that both arrays have the same number of elements.

If you are programming in Java or C#, the function prototype is int equivalentArrays(int[ ] a1, int[ ] a2)

Examples:

if a1 is	**and a2 is**	**return**
{0, 1, 2}	{2, 0, 1}	1

{1, 1, 1, 1, 1, 1}	{1, 1, 1, 1, 1, 2}	0 because every element of a1 is in a2 but not vice versa.
{}	{3, 1, 1, 1, 1, 2}	0 because every element of a1 (show me one that isn't) is in a2 but not vice versa.

```
public static void main (String args[]){
            int arr1[]={2,0,1};
            int arr2[]={0,2,1};
            System.out.println(f(arr1,arr2));
  }

      public static int f(int a1[],int a2[]){

            if ((a1.length==0) ||( a2.length==0))
                  return 0;
            if (a1.length !=a2.length)
                  return 0;
            for (int i=0;i<a1.length;i++){
                  if (count(a1,a1[i])!=count(a2,a1[i]))
                      return 0;
                        }


            return 1;
      }

      public static int count(int a[],int x){
            int count=0;
            for(int y:a){
                  if (y==x)
                   count++;

            }
            return count;


      }
```

6, Write a function named **hasTwoValues** which takes an array as an argument. It returns 1 if all the elements of the array are one of two different values, otherwise it returns 0. Your solution must make exactly one pass through the array. It must not sort the array or a copy of the array.. Furthermore, your solution must not have any nested loops in it! We are looking for a solution that minimizes time. Note that an element of the array can be any value including 0, negative numbers, the maximum integer and the minimum integer.

If you are writing in Java or C#, the function signature is int hasTwoValues(int[ ] a)

If you are writing in C or C++, the function signature is int hasTwoValues(int a[ ], int len) where len is the length of a

Examplesm m

if a is	**Return**
{1, 2, 2, 1}	1 (because there are 2 different element values)

{1, 1, 1, 8, 1, 1, 1, 3, 3}	0 (because there are 3 different element values, 1, 3, 8, not two as required.)
{1, 1, 1, 1, 1, 1, 1, 1, 1}	0 (because there is only one element value, 1, not two as required.)
{}	0 (because there are 0 different element values, not two as required.)

Hint: One possible solution uses the following local variables

```
  int cnt1 = 0, cnt2 = 0;
  int n1=0, n2=0;

public static void main (String args[]){

            System.out.println(f(new int[]{1,1,1,1,1,2,2}));
      }
   public static int f(int a[]){
         int n1=0,n2=0;
         int cnt1=0,cnt2=0;
         n1=a[0];

         if (a.length==0)
                     return 0;

         for(int x: a){
               if (n1!=x)
                 n2=x;

                        }

         cnt1=count(a,n1);
         cnt2=count(a,n2);

         if (cnt2==0)
               return 0;
         if (a.length== cnt1 + cnt2)
               return 1;
         else
               return 0;

   }
   public static int count(int a[],int x){
            int count=0;
            for(int y:a){
                  if (y==x)
                   count++;

            }
            return count;


      }
```

7. Write a function named *isSorted* that accepts an integer array and returns 1 if its elements are in ascending or descending order, otherwise it returns 0.

If you are programming in Java or C#, the function signature is int isSorted(int[ ] a)

If you are programming in C or C++, the function signature is int isSorted(int a[ ], int len) where len is the number of elements in the array

Examples:

if the input array is	**return**
{1, 2, 5, 6}	1
{12, 3, 2, 1}	1
{1, 2, 6, 3}	0 (because it is in neither ascending or descending order)
{}	1
{2}	1

ANSWER

```
// Don't have a compiler

int isSorted(int a[ ], int len)
{
 int sort = 0;
 int ascending = 0;
 int descending = 0;
 for (int i = 0; i < (len - 1); i++)
  {
   if (a[i]< a[i + 1])
    {
     ascending ++;
    }
   else if (a[i] > a[i + 1])
    {
     descending ++;
    }
  }
 if ((ascending == (len - 1))||(descending == (len - 1)))
  {
   sort = 1;
  }
return sort;
}
```

8. **A normal number is defined to be one that has no odd factors, except for 1 and possibly itself.** Write a method named **isNormal** that returns 1 if its integer argument is normal, otherwise it returns 0.

The function signature is int isNormal(int n)

Examples

if the number is	return
0	1
1	1
2	1
3	1
4	1
5	1
6	0 (3 is a factor)
7	1
8	1
9	0 (3 is a factor)
10	0 (5 is a factor)
11	1
12	0 (3 is a factor)
13	1
14	0 (7 is a factor)
15	0 (3 and 5 are factors)
16	1
17	1
18	0 (3 is a factor)
19	1
20	0 (5 is a factor)

ANSWER
// Don't have a compiler

```
int isNormal(int n)
{
int normal = 1;
if (((n%3) == 0)||((n%5) == 0)||((n%7) == 0))
{
normal = 0;
}
return normal;
}
```

9. **An array is said to be dual** if it has an even number of elements and each pair of consecutive even and odd elements sum to the same value. Write a function named **isDual** that accepts an array of integers and returns 1 if the array is dual, otherwise it returns 0.

If you are programming in Java or C#, the function signature is int isDual(int[ ] a)

If you are programming in C or C++, the function signature is int isDual(int a[ ], int len) where len is the number of elements in the array

Examples

if the input array is	**return**
{1, 2, 3, 0}	1 (because 1+2 == 3+0 == 3)
{1, 2, 2, 1, 3, 0}	1 (because 1+2 == 2+1 == 3+0 == 3)
{1, 1, 2, 2}	0 (because 1+1 == 2 != 2+2)
{1, 2, 1}	0 (because array does not have an even number of elements)
{}	1

```
public static int f(int a[]){
            int value=0;
            if (a.length % 2!=0)
                  return 0;
            if (a.length > 0)
                  value=a[0] + a[1];

            for (int i=0; i < a.length - 1;i+=2){
                  if (a[i]%2==0)
                        if(a[i + 1]%2==0)
                              return 0;
                  if (a[i]%2!=0)
                        if(a[i + 1]%2!=0)
                              return 0;
                  if (value!=a[i] + a[i +1])
                              return 0;

                                    }
        return 1;
                  }
```

10. **A non-empty array *a* of length n is called an array of all possiblities** if it contains all numbers between 0 and a.length-1 inclusive. Write a method named **isAllPossibilities** that accepts an integer array and returns 1 if the array is an array of all possiblities, otherwise it returns 0.

If you are programming in Java or C#, the function signature is int isAllPossibilities(int[ ] a)

If you are programming in C or C++, the function signature is int isAllPossibilities(int a[ ], int len) where len is the number of elements in the array

Examples

if the input array is	**return**
{1, 2, 0, 3}	1
{3, 2, 1, 0}	1
{1, 2, 4, 3}	0 (because 0 not included and 4 is too big)

{0, 2, 3}	0 (because 1 is not included)
{0}	1
{}	0

```
public static int f(int a[]){
            int count=0;

            for (int i=0;i< a.length;i++){
                  for(int x:a){
                        if (i==x)
                              count++;
                              }
                        }
            if (count!=a.length)
                  return 0;
            return 1;
      }
```

11. **An array is called *layered*** if its elements are in ascending order and each element appears two or more times. For example, {1, 1, 2, 2, 2, 3, 3} is layered but {1, 2, 2, 2, 3, 3} and {3, 3, 1, 1, 1, 2, 2} are not. Write a method named **isLayered** that accepts an integer array and returns 1 if the array is layered, otherwise it returns 0.

If you are programming in Java or C#, the function signature is int isLayered(int[ ] a)

If you are programming in C or C++, the function signature is int isLayered(int a[ ], int len) where len is the number of elements in the array

Examples:

if the input array is	**return**
{1, 1, 2, 2, 2, 3, 3}	1
{3, 3, 3, 3, 3, 3, 3}	1
{1, 2, 2, 2, 3, 3}	0 (because there is only one occurence of the value 1)
{2, 2, 2, 3, 3, 1, 1}	0 (because values are not in ascending order)
{2}	0
{}	0

```
public static int f(int a[]){
            int count=0;

            if ((ascending(a)==true) && (a.length > 1)){
            for (int i=0;i< a.length;i++){
                  count=0;
                  for(int x:a){
                        if (a[i]==x)
```

```
                        count++;
                        }
                if (count < 2)
                        return 0;
                        }
                }
        else
                return 0;

return 1;
}

public static boolean ascending(int a[]){
        for (int i=0;i< a.length-1;i++)
                if (a[i] > a[i +1])
                        return false;
         return true;

}
```

```
int isDual(int a[ ], int len)
{
int dual=0;
int sum;
  if(len%2==0)
   {
    for (int i=0;i<len-3;i+2)
          if (a[i]+a[i+1]==a[i+2]+a[i+3])
          {
           sum++;
            }
  }
    if (sum==len/2-1)
      dual=1;

return dual;nc


int isAllPossibilities(int a[ ], int len)
{
 unique=0;
 int x=0;
for(int j=0;j<len&&x=j;j++)
  {
   for(int i=0;i<len;i++)
     {
      if(a[i]==j)
        x++;
```

```
    }
  }
if(x==len)
unique=x;
return unique;
}
```

```
int isLayered(int a[ ], int len)
  int  layer=0;
  int x=0;
if (len>1)
{
  for(int i=1;i<len;i++)
    {
      if(a[i]>=a[i-1])
        x++;
    }
}
if (x=len-1)
   layered=1;

reurn layered;
```

11. Write a function that accepts an array of non-negative integers and returns the second largest integer in the array. Return -1 if there is no second largest. The signature of the function is **int f(int[ ] a)**

Examples:

if the input array is	**return**
{1, 2, 3, 4}	3
{{4, 1, 2, 3}}	3
{1, 1, 2, 2}	1
{1, 1}	-1
{1}	-1
{}	-1

12. Write a function that takes an array of integers as an argument and returns a value based on the sums of the even and odd numbers in the array. Let X = the sum of the odd numbers in the array and let Y = the sum of the even numbers. The function should return X - Y
The signature of the function is: **int f(int[ ] a)**

Examples

if input array is	return
{1}	1
{1, 2}	-1
{1, 2, 3}	2
{1, 2, 3, 4}	-2
{3, 3, 4, 4}	-2
{3, 2, 3, 4}	0
{4, 1, 2, 3}	-2
{1, 1}	2
{}	0

13. Write a function that accepts a character array, a zero-based start position and a length. It should return a character array containing containing *length* characters starting with the *start* character of the input array. The function should do error checking on the start position and the length and return null if the either value is not legal. The function signature is: **char[ ] f(char[ ] a, int start, int len)**

Examples

if input parameters are	return
{'a', 'b', 'c'}, 0, 4	null
{'a', 'b', 'c'}, 0, 3	{'a', 'b', 'c'}
{'a', 'b', 'c'}, 0, 2	{'a', 'b'}
{'a', 'b', 'c'}, 0, 1	{'a'}
{'a', 'b', 'c'}, 1, 3	null
{'a', 'b', 'c'}, 1, 2	{'b', 'c'}
{'a', 'b', 'c'}, 1, 1	{'b'}
{'a', 'b', 'c'}, 2, 2	null
{'a', 'b', 'c'}, 2, 1	{'c'}
{'a', 'b', 'c'}, 3, 1	null
{'a', 'b', 'c'}, 1, 0	{}
{'a', 'b', 'c'}, -1, 2	null
{'a', 'b', 'c'}, -1, -2	null
{}, 0, 1	null

Answers

First answer

```
public static void main()
{
  a1(new int[]{1, 2, 3, 4});
  a1(new int[]{4, 1, 2, 3});
```

```
  a1(new int[]{1, 1, 2, 2});
  a1(new int[]{1, 1});
  a1(new int[]{1});
  a1(new int[]{});
}

static int a1(int[] a)
{
  int max1 = -1;
  int max2 = -1;

  for (int i=0; i<a.length; i++)
  {
    if (a[i] > max1)
    {
      max2 = max1;
      max1 = a[i];
    }
    else if (a[i] != max1 && a[i] > max2)
      max2 = a[i];
  }

  return max2;
}
```

Second answer

```
public static void main()
{
  a2(new int[] {1});
  a2(new int[] {1, 2});
  a2(new int[] {1, 2, 3});
  a2(new int[] {1, 2, 3, 4});
  a2(new int[] {3, 3, 4, 4});
  a2(new int[] {3, 2, 3, 4});
  a2(new int[] {4, 1, 2, 3});
  a2(new int[] {1, 1});
  a2(new int[] {});
}

static int a2(int[] a)
{
  int sumEven = 0;
  int sumOdd = 0;

  for (int i=0; i<a.length; i++)
  {
    if (a[i]%2 == 0)
      sumEven += a[i];
    else
      sumOdd += a[i];
  }

  return sumOdd - sumEven;
}
```

Third answer

```
public static void main()
{
  a3(new char[]{'a', 'b', 'c'}, 0, 4);
  a3(new char[]{'a', 'b', 'c'}, 0, 3);
  a3(new char[]{'a', 'b', 'c'}, 0, 2);
  a3(new char[]{'a', 'b', 'c'}, 0, 1);
  a3(new char[]{'a', 'b', 'c'}, 1, 3);
  a3(new char[]{'a', 'b', 'c'}, 1, 2);
  a3(new char[]{'a', 'b', 'c'}, 1, 1);
  a3(new char[]{'a', 'b', 'c'}, 2, 2);
  a3(new char[]{'a', 'b', 'c'}, 2, 1);
  a3(new char[]{'a', 'b', 'c'}, 3, 1);
  a3(new char[]{'a', 'b', 'c'}, 1, 0);
  a3(new char[]{}, 0, 1);
  a3(new char[]{'a', 'b', 'c'}, -1, 2);
  a3(new char[]{'a', 'b', 'c'}, -1, -2);
}

static char[] a3(char[] a, int start, int length)
{
  if (length < 0 || start < 0 || start+length-1>=a.length)
  {
    return null;
  }

  char[] sub = new char[length];
  for (int i=start, j=0; j<length; i++, j++)
  {
    sub[j] = a[i];
  }

  return sub;
}
```

14. **An array is called *balanced* if its even numbered elements (a[0], a[2], etc.) are even and its odd numbered elements (a[1], a[3], etc.) are odd.** Write a function named *isBalanced* that accepts an array of integers and returns 1 if the array is balanced, otherwise it returns 0.

If you are programming in Java or C#, the function signature is int isBalanced(int[ ] a)

If you are programming in C or C++, the function signature is int isBalanced(int a[ ], int len) where len is the number of elements in the array

Examples:

if the input array is	**return**	**reason**
{2, 3, 6, 7}	1	a[0] and a[2] are even, a[1] and a[3] are odd.
{6, 3, 2, 7}	1	a[0] and a[2] are even, a[1] and a[3] are odd.
{6, 7, 2, 3, 12}	1	a[0], a[2] and a[4] are even, a[1] and a[3] are odd.
{6, 7, 2, 3, 14, 95}	1	a[0], a[2], and a[4] are even, a[1], a[3] and a[5] are odd.
{7, 15, 2, 3}	0	a[0] is odd
{16, 6, 2, 3}	0	a[1] is even

{2}	1	a[0] is even
{3}	0	a[0] is odd
{}	1	true vacuously

```
public static int isbalanced(int a[]){


            for (int i=0;i <a.length;i+=2){
                  if (a[i] % 2!=0)
                        return 0;
                  }
            for (int i=1;i <a.length;i+=2){
                        if (a[i] % 2==0)
                              return 0;
            }

      return 1;
            }
```

15. Write a function named *eval* that returns the value of the polynomial $a_nx^n + a_{n-1}x^{n-1} + ... + a_1x^1 + a_0$.

If you are programming in Java or C#, the function signature is double eval(double x, int[ ] a)

If you are programming in C or C++, the function signature is double eval(double x, int a[ ], int len) where len is the number of elements in the array

Examples:

if x is	**if the input array is**	**this represents**	**eval should return**
1.0	{0, 1, 2, 3, 4}	$4x^4 + 3x^3 + 2x^2 + x + 0$	10.0
3.0	{3, 2, 1}	$x^2 + 2x + 3$	18.0
2.0	{3, -2, -1}	$-x^2 - 2x + 3$	-5.0
-3.0	{3, 2, 1}	$x^2 + 2x + 3$	6.0
2.0	{3, 2}	$2x + 3$	7.0
2.0	{4, 0, 9}	$9x^2 + 4$	40.0
2.0	{10}	10	10.0
10.0	{0, 1}	x	10.0

```
public static double eval(double x,int a[]){
```

```
double result=0;

for (int i=a.length-1;i >=0 ;i--){
        result=result + a[i]*(Math.pow(x, i));
                                    }
return result;
 }
```

16. A palindrome is a word or phase that reads the same backwards or forwards. Write a function named *isPalindrome* that returns true or false if the input array is a palindrome.

If you are programming in Java or C#, the function signature is boolean isPalindrome(char [] arr)

If you are programming in C or C++, the function signature is bool isPalindrome(char a[ ], int len) where len is the number of elements in the array.

Examples:

if the input array is	**output is**
{'t', 'o', 'p', 's', 'p', 'o', 't'}	true
{'t','o','t','o'}	false
{'d','o','t','s','e','e','s','t','o', 'd'}	true
{ }	false
{'a'}	true
{4, 0, 9}	false
the char string "ipreferpi"	true
{0, 1, 0}	true

```
public static boolean palindrome(char a[]){
            if (a.length==0)
                  return false;
      for (int i=0;i<a.length;i++)

                  if (a[i]!=a[(a.length -1 ) - i])

                        return false;

           return true;

            }
```

From Internet

import java.util.*; class Palindrome { public static void main(String args[]) { String original, reverse=""; Scanner in = new Scanner(System.in);

System.out.println("Enter a string to check if it is a palindrome"); original = in.nextLine(); int length = original.length(); for (int i = length - 1 ; i >= 0 ; i--) reverse = reverse + original.charAt(i); if (original.equals(reverse)) System.out.println("Entered string is a palindrome."); else System.out.println("Entered string is not a palindrome."); } }

```
public static boolean isPrime(int n)
{
        if (n <= 1) // Not prime
                return false;
        if (n == 2) // Prime
                return true;
        if (n % 2 == 0) // Divisible by 2 means it's always not a prime
                return false;

        // For all other numbers, test by checking the divisibility of the square root of the number
        int m = (int)Math.round(Math.sqrt(n));

        for (int i = 3; i <= m; i += 2)
        {
                if (n % i == 0)
                {
                        return false;
                }
        }
        return true;
}
```

17. Define a **dimple** of an array of integers to be an element whose value is strictly less than the value of the element before it and strictly less than the value the element that follows it in the array. So 5 is a dimple of {10, 5, 8} and -1 and 83 are dimples of the array {3, -1, 0, 1000, 83, 84}. Write a function named **countDimples** that returns the count of the number of dimples in its array argument.
21. If you are programming in Java or C#, the function signature is int countDimples(int[ ] n)
22. Some more examples

if input array is	return the score	comment
{1, 1, 2, 2, 3}	0	To be a dimple an element must be strictly less than its neighbors
{1}	0	An array must have at least 3 elements to have a dimple
{1, 0, 1, 0, 1}	2	Both 0s are dimples. Note: the maximum number of dimples that an array can have is (a.length-1)/2
{0, 1, 0, 1, 0}	1	The second 0 is the only dimple
{-1, -18736, 123817}	1	Do the math

```
public static int countDimples(int a[]){
            int count=0;
             for (int i=1;i<a.length-1;i++){
                if((a[i-1] > a[i]) && (a[i] < a[i+1]))
                   count++;
                               }

                  return count;

            }
```

18. The following method of multiplying numbers was used in ancient times. Put the numbers to be multiplied side by side. Call them n1 and n2. If n1 is odd put an asterisk next to n2. Next, divide n1 by 2, discarding any remainder and multiply n2 by 2. If the new value of n1 is odd put an asterisk next to the new value of n2. Continue dividing n1 by 2 and discarding the remainder and doubling n2 until n1 is 0. Mark each value of n2 that is associated with an odd value of n1 with an asterisk. Then sum all the n2 values that have asterisks next to them. The result is the product of n1 * n2.

Here is an example

n1	n2	mark
17	25	* (because 17 is odd)
8	50	
4	100	
2	200	
1	400	* (because 1 is odd)
0		

Summing the n2s that are marked by an asterisk gives 425 which is 17*25. Reversing n1 and n2 gives the same result

n1	n2	mark
25	17	* (because 25 is odd)
12	34	
6	68	
3	136	* (because 3 is odd)
1	272	* (because 1 is odd)
0		

Note that 17 + 136 + 272 = 425 = 25*17

Notice that there is no provision for negative numbers in the above algorithm. Perhaps the omission was not important thousands of years ago but today we use negative numbers. Therefore your version of the algorithm should handle negative numbers. Specifically,

n1	n2	result
-17	25	-425
17	**-25**	**-425**
-17	**-25**	**+425**

Write a function named **ancientMultiplication** that implements this algorithm. Its signature is *int ancientMultiplication(int n1, int n2)* This function should return n1*n2 computed using the above algorithm modified to handle negative numbers.

```
public static int f(int n1,int n2){

            int total=0;
            int originalN1=n1;
            int originalN2=n2;
            n1=Math.abs(n1);
            n2=Math.abs(n2);
            do {
                  if (n1%2!=0)
```

```
            total=total + n2;
          n1=n1/2;
          n2=n2*2;

        }while (n1 != 0);

          if ((originalN1 > 0)     && (originalN2 < 0))
                total= -total;
          if ((originalN1 < 0)     && (originalN2 > 0))
                total= -total;

return total;
    }
```

19. An array is defined to be **n-zero-packed** if it contains two or more non-zero elements and exactly *n* zeroes separate all non-zero elements that would be adjacent if the zeroes were removed. For example, the array {1, 0, 0, 18, 0, 0, -8, 0, 0} is 2-zero-packed because there are two zeroes between the 1 and the 18 and two zeroes between the 18 and the -8 and this accounts for all the non-zero elements. The array {0, 1, 0, 0, 0, 6, 0, 8, 0, 0, 4} is not 2-zero-packed because there are three zeroes between the 1 and the 6 and only one zero between the 6 and the 8

Write a function named **isNZeroPacked** with the following signature

If you are programming in Java or C#, the function signature is *int isNZeroPacked(int[ ] a, int n)*

If you are programming in C++ or C, the function signature is *int isNZeroPacked(int a[ ], int len, int n)* where len is the length of the array.

The function returns 1 if its array argument is n-zero-packed (note that n is passed as one of the arguments of the function)

Examples

a is	**and n is**	**then function returns**	**reason**
{0, 0, 0, 2, 0, 2, 0, 2, 0, 0}	1	1	because exactly 1 zero separates all the non-zero elements of the array
{12, 0, 0, 0, 0, 0, 0, 0, -8}	7	1	because exactly 7 zeroes separate all the non-zero elements of the array
{0, 0, 0, 0, 5, 0, 0, 4, 0, 0, 6}	2	1	because exactly 2 zeroes separate all the non-zero elements of the array
{0, 0, 0, 0, 5, 0, 0, 4, 0, 0, 0, 6}	2	0	because there are three zeroes between the 4 and the 6.
{0, 0, 0, 0, 5, 0, 4, 0, 0, 6}	2	0	because there is only one zero between the 5 and the 4.
{0, 0, 0, 0}	3	0	because array must have at least two non-zero elements
{0, 0, 1, 0, 0}	2	0	because array must have at least two non-zero elements

Note: zeroes at the beginning and end of the array should be ignored.