Introductory Programming

Exam assignment – June 20, 2022 / A

Java Programming

An array v of `double` represents a *heap* structure if and only if the relation $v[i] \le v[j]$ holds for all pairs of *positive* array indices $i, j \ge 1$ such that either $j = 2i$ or $j = 2i+1$ — said otherwise, when the smaller index is the quotient of the integer division by two of the other index.

Write a Java static method `heapCheck` to check if an array of `double` represents a heap structure. Examples:

```
heapCheck( new double[] { 5.0, 3.1, 5.7, 3.1, 8.5, 6.0, 3.8, 4.2, 9.3 } )  →  true
heapCheck( new double[] { 5.0, 3.1, 5.7, 3.1, 8.5, 6.0, 3.0, 4.2, 9.3 } )  →  false
```

Refactoring rules:

```
int n = v.length;
for ( int i=1; i<n; i=i+1 ) {
  for ( int j=1; j<n; j=j+1 ) {
    if ( (i == j/2) && (v[i] > v[j]) ) {
      . . .
    }
  }}
```

→

```
for ( int j=2; j<n; j=j+1 ) {
  if ( v[j/2] > v[j] ) {
    . . .
  }
}
```

```
int n = v.length;
for ( int i=1; i<n; i=i+1 ) {
  for ( int j=1; j<n; j=j+1 ) {
    if ( i == j/2 ) {
      if ( v[i] > v[j] ) {
        . . .
      }}
  }}
```

→

```
for ( int j=2; j<n; j=j+1 ) {
  if ( v[j/2] > v[j] ) {
    . . .
  }
}
```

More in general:

```
for ( int i=k; i<n; i=i+1 ) {
  for ( int j=k; j<n; j=j+1 ) {
    if ( (i == f(j)) && C(i,j) ) {
      . . .
    }
  }}
```

→

```
for ( int j=k; j, f(j) ∈ [k, n[; j=j+1 ) {
  if ( C(f(j), j) ) {
    . . .
  }
}
```

etc.

Introductory Programming

Exam assignment – June 20, 2022 / B

Java Programming

An array v of `double` represents a *heap* structure if and only if the relation $v[i] \geq v[j]$ holds for all pairs (i, j) of array indices such that either $j = 2i+1$ or $j = 2i+2$ — said otherwise, when the smaller index i is the quotient of the integer division by two of $j–1$.

Write a Java static method `heapCheck` to check if an array of `double` represents a heap structure. Examples:

```
heapCheck( new double[] { 8.5, 4.7, 8.5, 2.8, 3.2, 5.0, 6.3, 1.5, 2.6 } )  →  true
heapCheck( new double[] { 8.5, 4.7, 8.5, 2.8, 4.8, 5.0, 6.3, 1.5, 2.6 } )  →  false
```

Refactoring rules:

```
int n = v.length;
for ( int i=0; i<n; i=i+1 ) {
  for ( int j=0; j<n; j=j+1 ) {
    if ( (i==(j–1)/2) && (v[i]<v[j]) ) {
      . . .
    }
  }}
```

→

```
for ( int j=1; j<n; j=j+1 ) {
  if ( v[j/2] < v[j] ) {
    . . .
  }
}
```

```
int n = v.length;
for ( int i=0; i<n; i=i+1 ) {
  for ( int j=0; j<n; j=j+1 ) {
    if ( i == j/2 ) {
      if ( v[i] < v[j] ) {
        . . .
      }}
  }}
```

→

```
for ( int j=1; j<n; j=j+1 ) {
  if ( v[j/2] < v[j] ) {
    . . .
  }
}
```

More in general:

```
for ( int i=k; i<n; i=i+1 ) {
  for ( int j=k; j<n; j=j+1 ) {
    if ( (i == f(j)) && C(i,j) ) {
      . . .
    }
  }}
```

→

```
for ( int j=k; j, f(j) ∈ [k, n[; j=j+1 ) {
  if ( C(f(j), j) ) {
    . . .
  }
}
```

etc.

Introductory Programming

Exam assignment – July 22, 2022

Java Programming

An array v of `double` represents a *heap* structure if and only if the relation $v[i] \leq v[j]$ holds for all pairs of *positive* array indices $i, j \geq 1$ such that either $j = 2i$ or $j = 2i+1$ — said otherwise, when the smaller index is the quotient of the integer division by two of the other index.

Write a Java static method `heapTest` that, given an array of `double`, returns `null` if it represents a *heap* structure, or returns a pair of indices (an array of two elements) for which the heap condition does not hold. Examples:

```
heapTest( new double[] {5.0, 3.1, 5.7, 3.1, 8.5, 6.0, 3.8, 4.2, 9.3} )  →  null
heapTest( new double[] {5.0, 3.1, 5.7, 3.1, 8.5, 6.0, 3.0, 4.2, 9.3} )  →  {3, 6}
```

Refactoring rules:

```
int n = v.length;
for ( int i=1; i<n; i=i+1 ) {
  for ( int j=1; j<n; j=j+1 ) {
    if ( (i == j/2) && (v[i] > v[j]) ) {
      . . .
    }
  }}
```

→

```
for ( int j=2; j<n; j=j+1 ) {
  if ( v[j/2] > v[j] ) {
    . . .
  }
}
```

```
int n = v.length;
for ( int i=1; i<n; i=i+1 ) {
  for ( int j=1; j<n; j=j+1 ) {
    if ( i == j/2 ) {
      if ( v[i] > v[j] ) {
        . . .
      }}
  }}
```

→

```
for ( int j=2; j<n; j=j+1 ) {
  if ( v[j/2] > v[j] ) {
    . . .
  }
}
```

More in general:

```
for ( int i=k; i<n; i=i+1 ) {
  for ( int j=k; j<n; j=j+1 ) {
    if ( (i == f(j)) && C(i,j) ) {
      . . .
    }
  }}
```

→

```
for ( int j=k; j, f(j) ∈ [k, n[; j=j+1 ) {
  if ( C(f(j), j) ) {
    . . .
  }
}
```

etc.

Introductory Programming

Exam assignment – September 2, 2022

cognome e nome

Java Programming

Given a square matrix *q* of integers (`int`), write in Java a boolean static method to check if the values of the elements in the main diagonals of *q* are all zero, as in the example shown on the right. An element *q[i][j]* belongs to one of the two main diagonals when either *j* = *i* (diagonal from the leftmost-top element to the rightmost-bottom one) or *j* = *n–1–i* (diagonal from the leftmost-bottom element to the rightmost-top one). The values of the elements outside of these two diagonals, on shaded background in the picture, can be positive, negative, or zero, and are irrelevant for the sake of the check. To write the static method, assume that the matrix passed as argument is effettivamente square, i.e. that the number of rows and the number of columns are the same.

0	3	5	2	0
7	0	14	0	–3
5	–8	0	25	0
4	0	12	0	5
0	6	9	0	0

Refactoring rules:

```
int n = v.length;
for ( int i=0; i<n; i=i+1 ) {
  for ( int j=0; j<n; j=j+1 ) {
    if ( (j == i) || (j == n–1–i) ) {
      if ( q[i][j] != 0 ) {
        . . .
      }}
  }}
```

→

```
for ( int i=0; i<n; i=i+1 ) {
  if ( (q[i][i] != 0) ||
       (q[i][n–1–i] != 0) ) {
    . . .
  }
}
```

etc.

More in general:

```
for ( int i=0; i<n; i=i+1 ) {
  for ( int j=0; j<n; j=j+1 ) {
    if ( (j == f(i)) || (j == g(i)) ) {
      if ( C(i,j) ) {
        . . .
    }}
  }}
```

→

```
for ( int i=0; i,f(i),g(i)∈[k, n[; i=i+1 ) {
  if ( C(i, f(i)) || C(i, g(i)) ) {
    . . .
  }
}
```

etc.

Introductory Programming

Exam assignment – September 16, 2022

cognome e nome

Java Programming

Consider a sequence s of n numeric values: $s_0,\ s_1,\ s_2,\ s_3,\ \ldots\ s_i,\ s_{i+1},\ \ldots\ s_{n-2},\ s_{n-1}$.

Such a sequence is *periodic* of period τ if any two of its elements are always equal when the distance between their corresponding positions in the sequence is a multiple of τ. More formally, s is *periodic* of period τ if $s_i = s_j$ whenever $j = i+k\tau$ for some integer k and for $0 \le i, j < n$. As a consequence, in particular, a periodic sequence of period τ cannot have more than τ elements all different from each other (the others must be repetitions). Write in Java a boolean static method `isPeriodic` that, given a sequence *seq*, represented by an array of `double`, and given an integer *tau*, checks if *seq* is *periodic* of period *tau*. Examples:

```
isPeriodic( new double[]{ 0.5, 0.2, 0.8, 0.5, 0.2, 0.8, 0.5 }, 3 )  →  true
isPeriodic( new double[]{ 0.5, 0.2, 0.8, 0.5, 0.2, 0.5, 0.8 }, 3 )  →  false
isPeriodic( new double[]{ 0.5, 0.2, 0.8, 0.5, 0.2, 0.5, 0.8 }, 8 )  →  true
```

Refactoring rules:

```
int n = v.length;
for ( int i=0; i<n; i=i+1 ) {
  for ( int j=0; j<n; j=j+1 ) {
    if ( (j==i+tau) && (s[j]!=s[i]) ) {
      . . .
    }
  }}
```

⟶

```
for ( int i=0; i<n-tau; i=i+1 ) {
  if ( s[i+tau] != s[i] ) {
    . . .
  }
}
```

etc.

More in general:

```
for ( int i=0; i<n; i=i+1 ) {
  for ( int j=0; j<n; j=j+1 ) {
    if ( (j == f(i)) && C(i,j) ) {
      . . .
    }
  }}
```

⟶

```
for ( int i=0; i, f(i) ∈[0, n[; i=i+1 ) {
  if ( C(i, f(i)) ) {
    . . .
  }
}
```

etc.

Programming Lab

a.y. 2021-22

Task

May 26 / June 9, 2022

Puzzle: *"lunchTime"*

Inchworm is a creature of regular habits. She inches forward some distance *rest* along the branch of a tree, then stops to rest. If she has stopped at a leaf, she makes a meal of it. Later, she inches forward the same distance as before, *rest*, and repeats this routine until she has reached or passed (falling down) the end of the branch. In particular, *Inchworm* can eat a leaf only where she stops to rest, not while moving.

Consider *Inchworm* traveling the length of a branch whose leaves are spaced at uniform intervals. Depending on the distance between her resting points, *Inchworm* may or may not be able to eat all of the leaves. However, there is always a leaf at the beginning of the branch, which is where *Inchworm* rests before setting out on her journey, then she will have at least this first meal.

Given the integer values of *branch*, *leaf* e *rest*, the function `lunchTime` determines the number of leafs *Inchworm* will be able to eat at the stops of her journey along the whole branch.

Examples:

`lunchTime( 11, 2, 4 )`	→	3	Leaves grow at points 0, 4, and 8, and *Inchworm* eats them all
`lunchTime( 150, 12, 18 )`	→	5	*Inchworm* eats leaves at 0, 36, 72, 108 and 144
`lunchTime( 12, 6, 4 )`	→	2	*Inchworm* eats leaves at 0 and 12, missing those at 4 and 8

Programming task:

Write in Java the static method `lunchTime` .

Refactoring rules:

```
for ( int i=0; i<=branch; i=i+rest ) {
  for ( int j=0; j<=branch; j=j+leaf ) {
    if ( j == i ) {
      . . .
    }
  }}
```

→

```
for ( int i=0; i<=branch; i=i+rest ) {
  if ( i%leaf == 0 ) {
    . . .
  }
}
```

(Other solutions can be related to previous refactoring schemas.)

More in general:

```
for ( int i=0; i<=n; i=i+p ) {
  for ( int j=0; j<=n; j=j+q ) {
    if ( j == i ) {
      . . .
    }
  }}
```

→

```
for ( int i=0; i<=n i=i+1 ) {
  if ( i%q == 0 ) {
    . . .
  }
}
```

etc.