# DD1339 Introduktion till datalogi 2013/2014

# Uppgift nummer: 8

# Namn: Marcus Larsson

# Grupp nummer: 5

# Övningsledare: Marcus Dicander

# Betyg: ..... Datum: .............. Rättad av: .......................................

# Exercise 7.17

Man kan klicka på det misslyckade testet och få se vad det blev för error. Först skrivs vad som förväntades och sedan vad som resultatet blev. Den skriver även ut var i koden felet uppstod, på vilken rad. Man kan även klicka "Show Source" som tar en till raden i koden där felet uppstod.

# Exercise 7.18

import static org.junit.Assert.\*;

import org.junit.After;

import org.junit.Before;

import org.junit.Test;

/\*\*

\* The test class CommentTest.

\*

\* @author Marcus Larsson

\* @version 2013.11.16

\*/

public class CommentTest

{

/\*\*

\* Default constructor for test class CommentTest

\*/

public CommentTest()

{

}

/\*\*

\* Sets up the test fixture.

\*

\* Called before every test case method.

\*/

@Before

public void setUp()

{

}

/\*\*

\* Tears down the test fixture.

\*

\* Called after every test case method.

\*/

@After

public void tearDown()

{

}

@Test

public void testNewComment()

{

Comment comment1 = new Comment("Marcus", "test", 5);

assertEquals("Marcus", comment1.getAuthor());

assertEquals(5, comment1.getRating());

}

@Test

public void testUpvote()

{

Comment comment1 = new Comment("Marcus", "test", 3);

assertEquals(0, comment1.getVoteCount());

comment1.upvote();

assertEquals(1, comment1.getVoteCount());

}

@Test

public void testDownvote()

{

Comment comment1 = new Comment("Marcus", "test", 2);

comment1.upvote();

comment1.upvote();

assertEquals(2, comment1.getVoteCount());

comment1.downvote();

assertEquals(1, comment1.getVoteCount());

comment1.downvote();

comment1.downvote();

assertEquals(-1, comment1.getVoteCount());

}

}

# Exercise 7.15, 7.16, 7.19

import static org.junit.Assert.\*;

import org.junit.After;

import org.junit.Before;

import org.junit.Test;

import java.util.Random;

/\*\*

\* The test class SalesItemTest.

\*

\* @author mik

\* @version 0.1

\*/

public class SalesItemTest

{

/\*\*

\* Default constructor for test class SalesItemTest

\*/

public SalesItemTest()

{

}

/\*\*

\* Sets up the test fixture.

\*

\* Called before every test case method.

\*/

@Before

public void setUp()

{

}

/\*\*

\* Tears down the test fixture.

\*

\* Called after every test case method.

\*/

@After

public void tearDown()

{

}

/\*\*

\* Test that a comment can be added, and that the comment count is correct afterwards.

\*/

@Test

public void testAddComment()

{

SalesItem salesIte1 = new SalesItem("Java for complete Idiots", 21998);

assertEquals(true, salesIte1.addComment("James Duckling", "This book is great. I learned all my Java from it.", 4));

assertEquals(1, salesIte1.getNumberOfComments());

}

/\*\*

\* Test that a comment using an illegal rating value is rejected.

\*/

@Test

public void testIllegalRating()

{

SalesItem salesIte1 = new SalesItem("Java For Complete Idiots, Vol 2", 19900);

assertEquals(false, salesIte1.addComment("Joshua Black", "Not worth the money. The font is too small.", -5));

}

/\*\*

\* Test that a sales item is correctly initialised (name and price).

\*/

@Test

public void testInit()

{

SalesItem salesIte1 = new SalesItem("test name", 1000);

assertEquals("test name", salesIte1.getName());

assertEquals(1000, salesIte1.getPrice());

}

@Test

public void testTwoComments()

{

SalesItem salesIte1 = new SalesItem("Book", 35);

assertEquals(true, salesIte1.addComment("Marcus", "nice!", 3));

assertEquals(true, salesIte1.addComment("JOhan", "WOW!", 5));

assertEquals(2, salesIte1.getNumberOfComments());

}

@Test

public void testAddCommentSameAuthor()

{

SalesItem salesIte1 = new SalesItem("phone", 35);

salesIte1.addComment("Marcus", "test", 4);

assertEquals(false, salesIte1.addComment("Marcus", "testing", 2));

}

@Test

public void testRaitingValues()

{

SalesItem salesIte1 = new SalesItem("test", 50);

assertEquals(false, salesIte1.addComment("Marcus", "test", 0));

assertEquals(false, salesIte1.addComment("author", "testing", 6));

}

@Test

public void testFindMostHelpfulComment()

{

Random r = new Random();

SalesItem salesIte1 = new SalesItem("test", 346);

//create test comments

salesIte1.addComment("MArcus", "Test", 4);

salesIte1.addComment("Test", "THis is helpful", 2);

//randomly vote for comments

int first = r.nextInt(10);

int second = r.nextInt(10);

for(int i=0;i<first;i++){

salesIte1.upvoteComment(0);

}

for(int i=0;i<second;i++){

salesIte1.upvoteComment(1);

}

//get all comments and see which one has highest vote

int comments = salesIte1.getNumberOfComments();

Comment highest = salesIte1.getComment(0);

for(int i=0;i<comments;i++){

Comment comment = salesIte1.getComment(i);

if(comment.getVoteCount()>highest.getVoteCount()){

highest=comment;

}

}

//Compare with findMostHelpfulComment()

Comment comment1 = salesIte1.findMostHelpfulComment();

assertEquals(highest, comment1);

}

}

# Exercise Insertionsort implementation

import java.util.Arrays;

/\*\*

\* A collection of sorting algorithms for arrays of integers.

\*

\* @author Stefan Nilsson & Marcus Larsson

\* @version 2013-11-20

\*/

public class Sort

{

private static final boolean DEBUGGING = true;

private void debugPrint(String s) {

if (DEBUGGING) {

System.err.println("Sort: " + s);

}

}

/\*\*

\* Sort the elements in ascending order.

\* This algorithm has time complexity Theta(n\*n), where n is

\* the length of the array.

\*

\* @param v An array of integers.

\* @return The same array sorted in ascending order.

\*/

public void selectionSort(int[] v) {

int n = v.length;

debugPrint("selection sort, n=" + n);

for (int i = 0; i < n - 1; i++) {

// find index m of min element in v[i..n-1]

int m = i;

for (int j = i + 1; j < n; j++) {

if (v[j] < v[m]) {

m = j;

}

}

if (DEBUGGING && n < 10) {

debugPrint(Arrays.toString(v));

debugPrint("i=" + i + ", m=" + m);

}

// swap v[i] and v[m]

int temp = v[i];

v[i] = v[m];

v[m] = temp;

}

}

/\*\*

\* Sort the elements in ascending order.

\*

\* @param v An array of integers.

\*/

public void insertionsort(int[] unsortedArray){

for (int i=1;i<unsortedArray.length;i++){

for(int j=0;j<i;j++){

if(unsortedArray[i]<unsortedArray[j]){

//swap index i and j

int temp = unsortedArray[j];

unsortedArray[j]=unsortedArray[i];

unsortedArray[i]=temp;

}

}

}

}

}

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

/\*\*

\*This is another implementation of insertionsort that is faster due to that there is less operations.

\* Sort the elements in ascending order.

\*

\* @param v An array of integers.

\*/

public void insertionsort(int[] unsortedArray){

for (int i=1;i<unsortedArray.length;i++){

for(int j=0;j<i;j++){

if(unsortedArray[i]<unsortedArray[j]){

//move elements in array, save element i and then move all elements before i up one //position in list.

int temp = unsortedArray[i];

for(int a=i;a>j;a--){

unsortedArray[a] = unsortedArray[a-1];

}

//insert element i in the correct place.

unsortedArray[j]=temp;

break;

}

}

}

}

## SortTest

import java.util.Arrays;

import java.util.Random;

/\*\*

\* Test class for Sort.

\*

\* @author Stefan Nilsson & Marcus Larsson

\* @version 2013-11-20

\*/

public class SortTest extends junit.framework.TestCase

{

/\*\*

\* t: test case, s: expected solution.

\*/

private int[] t0, s0, t1, s1, t2, s2, t7, s7;

/\*\*

\* Big array of random numbers.

\* tr: test case, sr: expected solution.

\* R\_SIZE is the size of the array.

\*/

private static final int R\_SIZE = 10000;

private int[] tr, sr;

private Random rand;

/\*\*

\* Constructs a new test case.

\*/

public SortTest() {

rand = new Random();

}

/\*\*

\* Sets up the test fixture.

\* Called before every test case method.

\*/

protected void setUp() {

t0 = new int[0];

s0 = new int[0];

t1 = new int[] {1};

s1 = new int[] {1};

t2 = new int[] {2, 1};

s2 = new int[] {1, 2};

t7 = new int[] {9, 5, 2, 7, 1, 6, 6};

s7 = new int[] {1, 2, 5, 6, 6, 7, 9};

tr = new int[R\_SIZE];

sr = new int[R\_SIZE];

for (int i = 0; i < R\_SIZE; i++) {

tr[i] = sr[i] = rand.nextInt();

}

Arrays.sort(sr);

}

/\*\*

\* Tears down the test fixture.

\* Called after every test case method.

\*/

protected void tearDown() {

}

public void testInsertionsort()

{

Sort sort = new Sort();

sort.insertionsort(t0);

assertTrue(Arrays.equals(t0, s0));

sort.insertionsort(t1);

assertTrue(Arrays.equals(t1, s1));

sort.insertionsort(t2);

assertTrue(Arrays.equals(t2, s2));

sort.insertionsort(t7);

assertTrue(Arrays.equals(t7, s7));

sort.insertionsort(tr);

assertTrue(Arrays.equals(tr, sr));

}

public void testSelectionSort() {

Sort sort = new Sort();

sort.selectionSort(t0);

assertTrue(Arrays.equals(t0, s0));

sort.selectionSort(t1);

assertTrue(Arrays.equals(t1, s1));

sort.selectionSort(t2);

assertTrue(Arrays.equals(t2, s2));

sort.selectionSort(t7);

assertTrue(Arrays.equals(t7, s7));

sort.selectionSort(tr);

assertTrue(Arrays.equals(tr, sr));

}

}

# Exercise Reverse list

Ett exempel på en algoritm som ändrar ordningen i en lista så att den blir baklänges är:

Algorithm reverseList(A):

Input: an array A storing n elements

output: the same array with the elements in reversed order.

for i = 0 to n/2

if(A[i]!=A[n-i])

swap(A[i], A[n-i])

Vad den gör är att den loopar genom endast halva listan och byter sista elementet med det första. Nästa varv i loopen byter den element 2 med näst sista osv. Tills den kommer till mitten och "byter plats på det mittersta elementet med det mittersta elementet om n är udda" Alltså ingenting den sista gången. Om elementen som ska byta plats har samma siffra så byter man inte eftersom det är onödiga operationer, resultatet blir ju desamma.

# Exercise lista i växande ordning

f4(n)=n+100

f3(n)=nlogn

n^1,5

2^n

10^n

# Exercise sanna påståenden.

* *n*(*n* + 1) / 2 = *O*(*n*3) ----- Sant.O(n3) är bara en väldigt dålig approximation. n3 är väldigt mycket större för stora n.
* *n*(*n* + 1) / 2 = *O*(*n*2) -------- Sant. n(n+1)/2 = (n2+n)/2. För stora n är (n2'n)/2 alltid mindre än n2
* *n*(*n* + 1) / 2 = Θ(*n*3) --------- Falskt. Detta uttryck är alltid mindre än n3 och och därför kan inte n3 vara "genomsnitt"
* *n*(*n* + 1) / 2 = Ω(*n*) -------- Sant. (n2+n)/2 kommer alltid vara större än n för n>1.

# Excercise Beräkna O(f(n))

Indata är en heltalsvektor *A* med *n* element. Vi vill beräkna en vektor *B*, där *B*[i] =*A*[0] + *A*[1] + ... + *A*[i]. Här är en enkel algoritm som löser problemet.

**for** i = 0 **to** n-1

Add the numbers A[0] thru A[i].

Store the result in B[i].

* Beräkna tidskomplexiteten för denna algoritm och uttryck den på formen *O*(*f(n)*), där funktionen *f(n)* är så liten och enkel som möjligt.
* Visa att tidskomplexiteten också är Ω(*f(n)*).

Vad man gör är att man loopar igenom en lista n gånger. Och för varje gång loopar man igenom bara en bit av listan. Första gången 1 gång, nästa gång 2 gånger osv.

Detta bildar en aritmetisk summa. Formeln för en aritmetisk summa är: (n(a0+an))/2 där a0 är första talet och an är sista talet. I detta fallet är a0=1och an=n så därför har vi: (n(n+1))/2=(n2+n)/2

Ordo för detta är O(n2).

Eftersom man måste titta på alla element i listan går det inte att göra färre gånger. Därför är minsta gränsen samma.

* + Hitta på en algoritm med bättre asymptotisk tidskomplexitet. Beskriv algoritmen i pseudokod och ange dess tidskomplexitet med *O*-notation.

Algorithm sumArray(A):

Input: an array A storing n elements

output: a new array B with B[0]=A[0], B[1]=A[0]+A[1]...

B[0]=A[0]

for i=1 to n

B[i]=B[i-1]+A[i]

Ordo blir i denna algoritm O(n). Man loopar igenom listan A endast en gång och utnyttjar att man redan har räknat ut dem tidigare värdena.

* Ge ett exempel på en positiv funktion *f(n)* sådan att *f(n)* varken är *O*(*n*) eller Ω(*n*).

Vi måste hitta en funktion som pendlar genom att gå över n och under n.

Exempelvis n2|sin(n)|. Den kommer pendla mellan n2 och 0.

Man skulle också kunna tänka sig en funktion som n2|cos(n)|+logn|sin(n)| denna blir negativ för små n men för stora n pendlar den mellan log(n) och n2

Man skulle också kunna ha en diskontinuerlig funktion som F(x) = n2 om n är jämn och log n om n är jämt. Detta skulle också alltid pendla mellan gradtalen och därför inte kunna vara O(n) eller Omega(n).