PG4200: Algorithms And Data Structures

Lesson 04:
Recursion and
Test Driven Development (TDD)

Prof. Andrea Arcuri

Sorting Overview

- Many different kinds of sorting algorithms

- Most language APIs provides good defaults

  - Unless very large data, default will be fine 99% of the cases

- Sorting is very popular in programming

  - Important to understand how it works under the hood

- Tractable mathematically

  - So good example to show how to analyze algorithms

Sorting Algorithms

- Insertion Sort

- Merge Sort

- Quick Sort

- There are more, but those are the most famous that you need to know (besides Bubble Sort)

- Good way to see a problem been solved in many different ways

Insertion Sort

- An array of size 0 or 1 is always considered sorted

- From left to right, till length N

- K-leftmost values are sorted

- Position K+1 is not sorted, insert it in the first K

  - by swapping adjacent elements, like in Bubble Sort

| 4 | 5 | 1 | 3 | 2 | 6 | K=0

| 4 | 5 | 1 | 3 | 2 | 6 | K=1

| 4 | 5 | 1 | 3 | 2 | 6 | K=2

| 4 | 5 | 1 | 3 | 2 | 6 | K=3

| 4 | 1 | 5 | 3 | 2 | 6 | swap

| 1 | 4 | 5 | 3 | 2 | 6 | swap

Cont.

| 1 | 4 | 5 | 3 | 2 | 6 | K=4

| 1 | 4 | 3 | 5 | 2 | 6 | swap

| 1 | 3 | 4 | 5 | 2 | 6 | swap

| 1 | 3 | 4 | 5 | 2 | 6 | K=5

| 1 | 3 | 4 | 2 | 5 | 6 | swap

| 1 | 3 | 2 | 4 | 5 | 6 | swap

| 1 | 2 | 3 | 4 | 5 | 6 | swap

| 1 | 2 | 3 | 4 | 5 | 6 | K=6

- Best case: already sorted, e.g., 1-2-3-4-5-6, need to do N-1 comparisons, so $\Omega(N)$

- Worst case: opposite order, e.g, 6-5-4-3-2-1, each element needs to be compared and swapped with all previous K ones, so O(N^2)

Merge Sort

- *Divide and Conquer*

4	5	1	3	2	6

  - Recursive implementation

- Split the array in two

4	5	1	3	2	6

- Sort the two parts

1	4	5	2	3	6

- Merge the two parts once sorted

1	2	3	4	5	6

Recursion

- Writing *mergeSort(T[] array)*

- How to sort the 2 halves if my goal was to write a sort algorithm???

- Recursion: reuse function you are writing, but on smaller data

mergeSort(T[] array){

    mergeSort(array, 0, array.length / 2)

    mergeSort(array, array.length /2 , array.length)

    mergeHalves(array)

}

Recursion End

- To avoid infinite loop, you need to define a stopping condition

- When do I know for certain that an array is sorted?

- Answer: when its size is at most 1

- So, stop recursion when reaching an half of size 1 or less

Merge of Halves

- By recursion, I can sort the 2 halves, but array with 2 sorted halves is not sorted

- Scan the 2 halves, and copy min to a new array

- Between N/2 and N comparisons: once reached end of one half, copy over the other

1	4	5

2	3	6

1					

1	4	5

2	3	6

1	2				

1	4	5

2	3	6

1	2	3			

1	4	5

2	3	6

1	2	3	4		

Cost

- Cost based on 2 recursive calls and then merge

- C(1) = O(1)

- C(n) >= C(n/2) + C(n/2) + n/2
  - Best case for merge, only have to look at one of the halves
  - Eg, all elements in half A are lower than first element in half B

- C(n) <= C(n/2) + C(n/2) + n
  - Worst case for merge, need to look at whole of both halves

- $\dots C(n) = \Theta(n \log n)$

Considerations

- Asymptotically, does not exist comparison-based sorting better than O(n log n)

- Merge-sort is therefore asymptotically optimal

  - Ie, no instance for which get worse than O(n log n)

- But… more memory, need extra array buffer

- … might be not best on *average*

Wikipedia

- Besides book, Wikipedia is good source to read about algorithms and data structures in layman terms
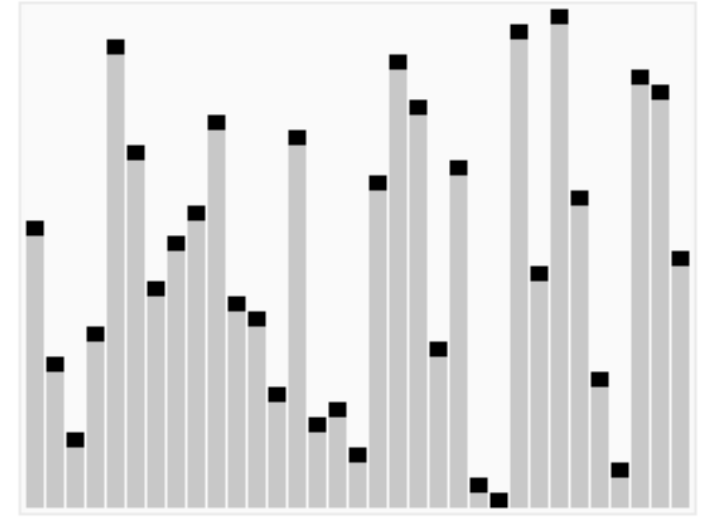
    - https://en.wikipedia.org/wiki/Merge_sort

6 5 3 1 8 7 2 4

Quick Sort

- Most used sorting algorithm

  - Ie, usually it is the default in the language APIs

- *Fast on average*, usually "n log n"

  - Better "constant" compared to Merge Sort (eg, no moving data to buffer)

  - But can go till n^2 in worst case

- Minimal memory overhead

- Lot of variants studied during the years

Recursion

- Still Divide and Conquer algorithm, like Merge Sort

- Choose a value X (pivot)

- Move values <X before X, and >X after it

- After one step X is in the correct position

- Apply recursion on subarrays before and after X

| 6 | 3 | 7 | 5 | 4 | 2 | 1 |

| 1 | 3 | 7 | 5 | 4 | 2 | 6 |

| 1 | 3 | 7 | 5 | 4 | 2 | 6 |

| 1 | 3 | 2 | 5 | 4 | 7 | 6 |

| 1 | 3 | 2 | 5 | 4 | 7 | 6 |

| 1 | 3 | 2 | 4 | 5 | 7 | 6 |

| 1 | 3 | 2 | 4 | 5 | 7 | 6 |

- Choose a pivot, eg value 5

- Scan from left till > 5, from right till < 5

- Swap (eg 6 with 1), and continue

- Note how 3 is not touched, as < 5

- At the end, the pivot 5 is in the right position

- On left side, all values < 5

- On right side, all values > 5

- Apply recursively on left and right of pivot

Choice of Pivot

- Choice of pivot is crucial for performance of Quick Sort
  - pivot = array[i]

- For performance, would like a pivot value that gives the 2 partitions of equal size

- How to choose "i"?
  - An option is to take "i" at random
  - Another option is to take middle, which is good when array nearly sorted

1 (and 7) is worst choice

4	3	7	1	5	2	6

4 is best choice

4	3	7	1	5	2	6

Which Sorting Algorithm to Use?

- Unless you have very specific, advanced cases, you use the *default* of what is given by the standard library of the language/framework you use

- Most of the time, it will be a variant of Quick Sort

 - Fast on average, often "n log n" cost, no extra memory

Test Driven Development (TDD)

TDD in One Slide

First write the test cases, which will fail, and then write code to make the test cases pass

What TDD is Not

- It is not a fancy tool you can buy/download

- No cutting edge novel technology

- No silver bullet solving all your software development problems

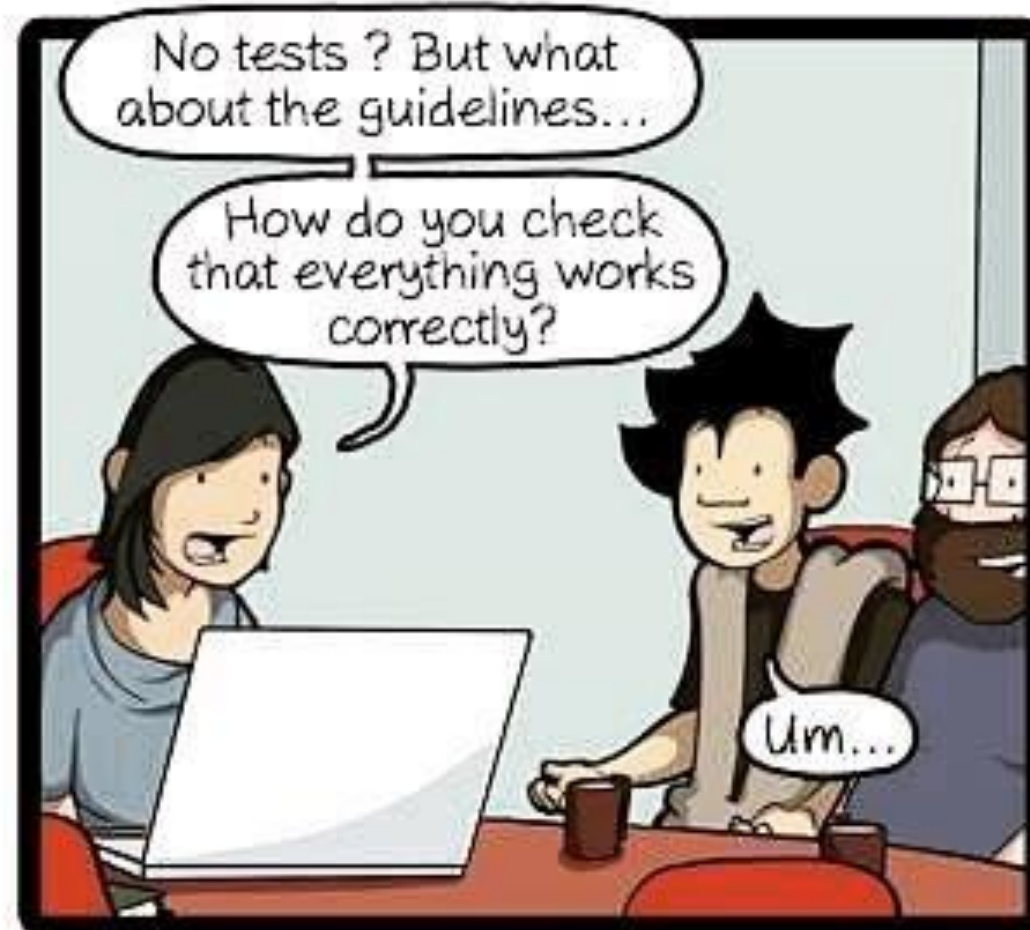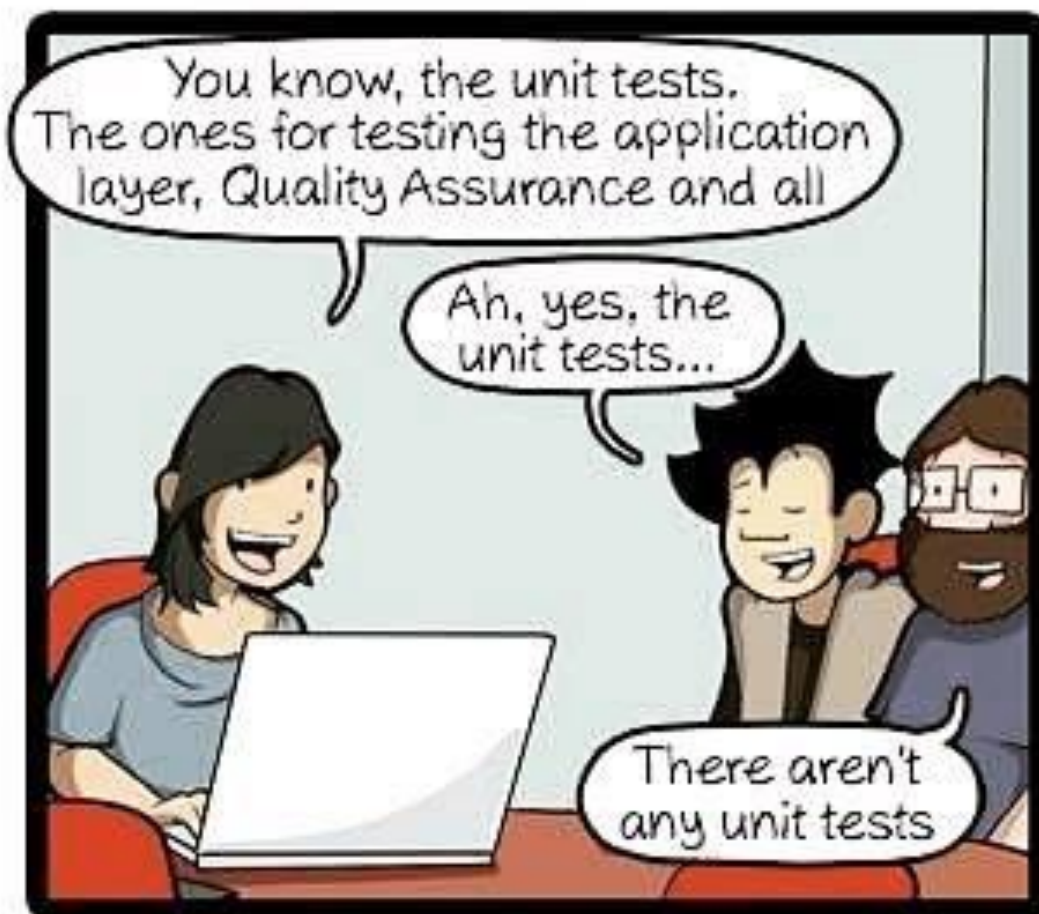- *It is just a (relatively simple) design methodology to improve productivity*
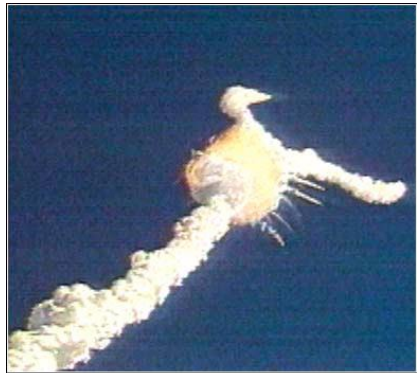
Why TDD???

- Bugs in the field can be extremely expensive
  - Ariane 5 explosion, Toyota braking system (100k recalled cars), etc

- Aim at improving *quality* ("testing") within acceptable *cost*

- Lot of **hype** on TDD as possible solution

CommitStrip.com

Some Facts on TDD

- There isn't much *scientific* evidence that it helps on *large scale* systems
  - Small improvements in *quality*, but also small decrease in *productivity*

- But many anecdotal, success stories

- My *opinion* might be different from what you might hear/read on TDD

The Effects of Test-Driven Development on External Quality and Productivity: A Meta-Analysis

Yahya Rafique and Vojislav B. Mišić, *Senior Member*, *IEEE*

Abstract—This paper provides a systematic meta-analysis of 27 studies that investigate the impact of Test-Driven Development (TDD) on external code quality and productivity. The results indicate that, in general, TDD has a small positive effect on quality but little to no discernible effect on productivity. However, subgroup analysis has found both the quality improvement and the productivity drop to be much larger in industrial studies in comparison with academic studies. A larger drop of productivity was found in studies where the difference in test effort between the TDD and the control group's process was significant. A larger improvement in quality was also found in the academic studies when the difference in test effort is substantial; however, no conclusion could be derived regarding the industrial studies due to the lack of data. Finally, the influence of developer experience and task size as moderator variables was investigated, and a statistically significant positive correlation was found between task size and the magnitude of the improvement in quality.

Index Terms—Test-driven development, meta-analysis, code quality, programmer productivity, agile software development

TDD and Algorithms

- Algorithms is a good a place to start with TDD, as having clear functions with clear expected outputs

- Lot of hype on TDD

- However, here we deal with small functions and classes, and not whole applications

Do You Practice What You Preach?

- TDD can be useful, especially for students / junior developers

- But to be honest, I am not using TDD...
  - TDD is a bottom-up approach to software design, whereas I prefer top-down

- As any technique, its success depends on the context
  - eg, type of software, stage in which TDD is introduced (at the beginning or in a mature project), etc

- Can test software even without TDD

- Success of TDD depends also on *management*

General Principles of TDD

Example: A List

```
//Start from an "empty" stub that compiles

class List{

    int size(){return -1;  //wrong value}

}
```

First Unit Test

```
void testEmptyList(){

    List list = new List();

    Assert.equals( 0 , list.size());

    // expected empty list

}
```
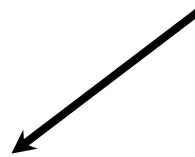
This will fail!

Fix The Code

```
class List{

    int size(){return 0;}

}
```

Now the test case does not fail

Refactor Step

Improve code

```
class List{

    int counter = 0;

    int size(){return counter;}

}
```

Integration Step

- As all test cases do pass, make changes *permanent*

- Example: commit to repository

  - Git, CVS, SVN, Mercurial, etc.

- Do you really need to at each TDD step??? I do not think so...

  - overhead, tedious, side effects if email generated for each commit, etc.

Back to First Step

```
class List{

    int counter = 0;

    void insert(int value){ /* do nothing */}

    int size(){return counter; }

}
```

Write A Second Test

```
void testInsertOneElement(){

    List list = new List();

    list.insert(42);

    Assert.equals( 1 , list.size());

}
```

This will fail!

Minimal Fix

Does not work:
- *testInsertOneElement* does pass
- but *testEmptyList* will fail

```
class List{

    int counter = 1;

    void insert(int value){ }

    int size(){return counter; }

}
```

Still need all passing tests

```
class List{

    int counter = 0;

    void insert(int value){ counter++;}

    int size(){return counter; }

}
```

Refactoring Step

```
class List{

    int[] data = new int[16];

    int counter = 0;

    void insert(int value){ data[counter]=value;

                            counter++;}

    int size(){return counter; }

}
```

Do not do that! You have no test case checking for internal state!

New Empty Stub

```
class List{

    int counter = 0;

    void insert(int value){ counter++;}

    int size(){return counter; }

    int getLast(){return 0;}

}
```

Write A Third Test

```
void testInsertAndGetOneElement(){

    List list = new List();

    int value = 42;

    list.insert(value);

    Assert.equals( value , list.getLast());

}
```

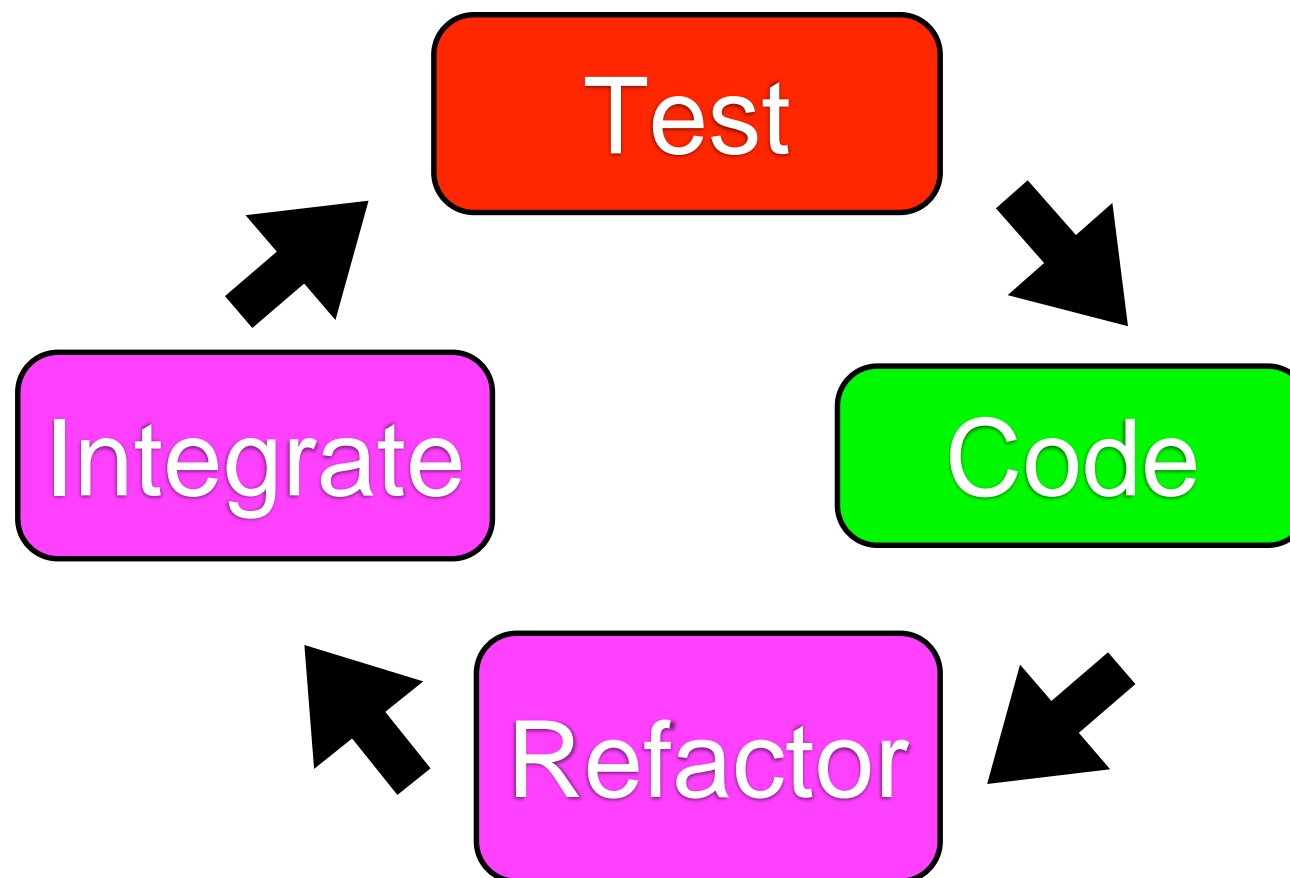Note: we are not testing for "size" here

This will fail!

Minimal Fix Before Refactoring

```
class List{

    int counter = 0;

    void insert(int value){ counter++;}

    int size(){return counter; }

    int getLast(){return 42;}

}
```
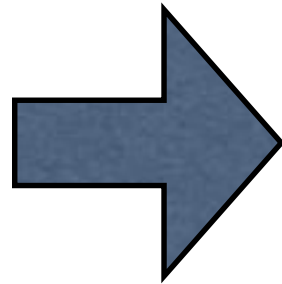
Measuring Test Effectiveness

- Big issue, regardless of TDD

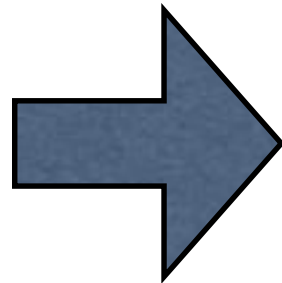- #bugs found in QA is a tricky measure

- *Code coverage* is often used

TDD and Coverage

- TDD is not about more code coverage

- Managers can just set a target (eg >50%) and let engineers decide how to achieve it

- TDD is about *design*

Coverage Does Not Tell You Much

```
void testInsertOneElement(){

    List list = new List();

    list.insert(42);

    int size = list.size();

// Assert.equals( 1 , size );

}
```

Let's say 15% coverage

Coverage does not change!!!

How Good Are Your Test Assertions?

- Current tools cannot tell you that

  - However, "Mutation Testing" is starting to get usable

- TDD helps you because *force* you to write effective assertions

  - *tests first need to fail*, and then pass

TDD is A Design Methodology

- You can write tests even without TDD

- Incremental, *small* steps, each one *verified*

- Can be useful for complex code

- Can apply TDD only on subset of a project

- TDD might lead to write very different code
  - impact on architecture

Homework

- Study Book Chapter 2.2 and 2.3

- Study code in the *org.pg4200.les04* package

- Do exercises in *exercises/ex04*

- Extra: do exercises in the book