DEVELOPMENT AND TESTING
DEVELOPMENT (ONE OUT OF MANY PERSPECTIVES)

1. Solve
2. Implement
   1. Write test
   2. Write code
   3. Repeat
3. Integrate
4. Release
TEST DRIVEN DEVELOPMENT (TDD)
I’m trying to understand test-driven development.

That’s easy. First you make a test that fails, then you do the least amount of work possible to make it pass.

So, if I’m going to build a bridge...

...step one would be to drive your car over a cliff.

I don’t want to be on your project anymore.

We can discuss that later. Take these keys.
PRACTICAL EXAMPLE

• Let's practice some TDD on the following example

Your project manager at BusyBody Inc says he needs a feature implemented which determines the total amount of time a worker at the company spends at their desk. He says the number of hours each day is already being measured and is stored in an internal array in the code base.
PRACTICAL EXAMPLE

• How do we solve this?
  • Compute the sum!
PRACTICAL EXAMPLE

• First we write a test
  • in other words, set up the scaffolding of the code instead of a function which you don’t know if it works or not – and continue to struggle finding bugs

```java
public static double sum(double[] arr) {
    return Double.POSITIVE_INFINITY; //note this clearly does not work and is thus failing
}

public static void main() {
    double[] arr = {0, 1, 1, 2, 3, 5, 8};
    if(sum(arr) != 20)
        System.out.println("Test failed?!?!?! I suck!"); //you don't, its supposed to fail!
}
```
**PRACTICAL EXAMPLE**

- Before we continue, let’s review
  - Positives
    - Scaffolding, function interface, and test all implemented
    - We know it is good design
    - Tests to tell if the code is correct, before we struggle with debugging many lines of code
  - Negatives
    - Code isn’t written until later.....but is that really that bad? NO

- In fact, with TDD you code FASTER and more EFFECTIVELY than without it
PRACTICAL EXAMPLE

• Now the code – and then run the test!

```java
public static double sum(double[] arr) {
    double s = 0;
    for (double x : arr)
        s += x;
    return s;
}
```
THINGS TO REMEMBER

• Always have code that compiles
• Test writing is an art that takes practice (and more learning!)
• Compile and test often!
TESTING FRAMEWORKS

• Many frameworks exist CppUnit, JUnit, etc.

• We will be using a much more simple unit testing framework developed by me
  • A unit test is a check of one behavior of one “unit” (e.g., function) of your code
  • If you have downloaded the lab zip for today open it and look there
  • Follows SETT – unit testing paradigm
    • Setup – create data for input and predetermine the output
    • Execute – call the function in question
    • Test – analyze correctness and determine true/false for test
    • Teardown – cleanup any data, close buffers, etc
public static boolean testSum() {
    //setup
    double[] arr = {0, 1, 1, 2, 3, 5, 8};
    double ans = 20;

    //execute
    double s = sum(arr);

    //test
    return s == ans;

    //teardown - here is empty
}
TDD - EXERCISE

• Write a Java function to find the minimum of an array of integers
  • Do test driven development, starting with a good unit test
  • After test is created and checked, code the function

• Pair program
CH 4
ALGORITHM ANALYSIS

ACKNOWLEDGEMENT: THESE SLIDES ARE ADAPTED FROM SLIDES PROVIDED WITH DATA STRUCTURES AND ALGORITHMS IN JAVA, GOODRICH, TAMASSIA AND GOLDWASSER (WILEY 2016)
ANALYSIS OF ALGORITHMS (CH 4.2-4.3)
RUNNING TIME

• Most algorithms transform input objects into output objects.

• The running time of an algorithm typically grows with the input size.

• We focus on the worst case running time.
  • Easier to analyze
  • Crucial to applications such as games, finance, and robotics
LIMITATIONS OF EXPERIMENTS

• It is necessary to implement the algorithm, which may be difficult.

• Results may not be indicative of the running time on other inputs not included in the experiment.

• In order to compare two algorithms, the same hardware and software environments must be used.
THEORETICAL ANALYSIS

• Uses a high-level description of the algorithm instead of an implementation
• Characterizes running time as a function of the input size, $n$
• Takes into account all possible inputs
• Allows us to evaluate the speed of an algorithm independent of the hardware/software environment
**BIG-OH NOTATION**

- Given functions $f(n)$ and $g(n)$, we say that $f(n)$ is $O(g(n))$ if there are positive constants $c$ and $n_0$ such that $f(n) \leq cg(n)$ for $n \geq n_0$
  - $f(n)$ - real computation time (measured time, if you will)
  - $g(n)$ - approximation function

- Example: $2n + 10$ is $O(n)$
  - $2n + 10 \leq cn$
  - $\frac{10}{c-2} \leq n$
  - Pick $c = 3$ and $n_0 = 10$

- Determining the function – one way is to count the operations

- To reduce: Strip constants, and take highest order terms
  - Constants do no matter because of limits as $n$ goes to infinity
**EXAMPLE**

**ADDING TO AN ARRAY**

- To add an entry $e$ into array $A$ at index $i$, we need to make room for it by shifting forward the $n - i$ entries $A[i], ..., A[n - 1]$

---

**Add**

**Input:** Array $A$, index $i$, element $e$

1. for $k \leftarrow n, n-1, ..., i+1$
2. $A[k] \leftarrow A[k-1]$
3. $A[i] \leftarrow e$
4. $n \leftarrow n + 1$
EXAMPLE
ADDING TO AN ARRAY

• Best case
  • Add at the end of the array
  • One comparison, one copy, one increment
  • $3 = O(1)$, by removal of constants

• Worst case
  • Add at the beginning of the array
  • $n$ comparisons, $n$ copies, $2n$ increments
  • $4n = O(n)$, by removal of constants

• Average case?
EXERCISES

• Removing from an array
  • Best, average, worst cases
• Inserting at head or tail of linked list
• Removing head of tail of doubly-linked list
• Removing head of singly-linked list
• Removing tail of singly-linked list
SEVEN IMPORTANT FUNCTIONS

- Seven functions that often appear in algorithm analysis:
  - Constant \( \approx 1 \)
  - Logarithmic \( \approx \log n \)
  - Linear \( \approx n \)
  - N-Log-N \( \approx n \log n \)
  - Quadratic \( \approx n^2 \)
  - Cubic \( \approx n^3 \)
  - Exponential \( \approx 2^n \)

- In a log-log chart, the slope of the line corresponds to the growth rate.
BIG-OH ANALYSIS APPLIES TO TIME AND MEMORY

• How about recursion?
  • Each function call uses memory!
COMMON PROOF TECHNIQUES FOR THIS CLASS

• Direct proof – using knowledge of axioms and definitions
  • Used for determining theoretical complexity
  • **Loose example**
    • Copying takes one operation. My loop runs \( n \) times and performs \( n \) copies. Therefore the total runtime is \( O(n) \)

• Contradiction – assume the opposite and reach an impossibility
  • We will see this later in the course, in proving properties of structures
  • **Loose example**
    • Prove: if \( ab \) is odd, then \( a \) is odd and \( b \) is odd. Proof: Assume \( a \) is even, then \( a = 2j \) for some integer \( j \). Thus \( ab = 2(jb) \), implying \( ab \) is even. This is a contradiction to our original assumption, thus \( a \) cannot be even.

• Induction – not on a test or homework, only for my lectures

• Counterproof by example
RUNTIME ANALYSIS
BIG-OH

• Given functions $f(n)$ and $g(n)$, we say that $f(n)$ is $O(g(n))$ if there are positive constants $c$ and $n_0$ such that $f(n) \leq cg(n)$ for $n \geq n_0$
  
  • We need to know how to determine $f(n)$, $c$, and $n_0$
  
  • This is all done through experiments
DETERMINING $f(n)$

- Vary the size of the input and then determine runtime using `System.nanoTime()`

```java
1. for(int n = 2; n < MAX; n*=2) {
2.    int r = max(10, MAX/n); //number of repetitions
3.    long start = System.nanoTime();
4.    for(int k = 0; k < r; ++k)
5.       executeFunction();
6.    long stop = System.nanoTime();
7.    double elapsed = (stop - start)/1.e9/r;
8.}
```
DETERMINE \( c \) AND \( n_0 \)

• First plot \( f(n) \) – time vs size

• Second plot \( \frac{f(n)}{g(n)} \) or \( \frac{\text{time}}{\text{theoretical time}} \) vs size

• Look for where the data levels off. This will be \( n_0 \)

• Look for the largest value to the right of \( n_0 \), this will be \( c \)
TOGETHER – TIME LINEAR SEARCH

• We will download and modify Prog01.zip for this activity
ACTIVITY

- Determine big-oh constants for Arrays.sort();
- Theoretical complexity will be $O(n \log n)$
WHY GO THROUGH THIS ANALYSIS?

• If two algorithms have the same theoretical analysis, we must compare them experimentally

• Determining the $c$ value informs us:
  • Which algorithm is cheaper in practice on the experimental data

• Determining the $n_0$ informs us:
  • When the theoretical complexity begins holding true

• If you reach the memory limit of the machine, you will see "odd" effects…