Unit Testing

Eric McCreath

Black-Box Testing

Tests can be created based purely on the functional requirements of the code.

There are three types of possible test cases including:

- normal functioning of the program,
- boundary cases, and
- testing cases outside the requirements & robustness testing.

White-Box Testing

A set of tests can be constructed based on the code.

This enables the tester to target the test cases.

White-box testing would normally involve generating test cases that have good "code coverage"

Test cases can also be constructed based on boundary cases based on the code (these tests can be different than the boundary cases based on requirements).

It can be difficult to create test cases for some exception conditions.

Code Coverage

Code coverage is a measure of the amount of code that is "covered" when a set of test cases are run over the code. There are a number of different code coverage measures including:

- the proportion of methods executed,
- the proportion of statements executed,
- if the branches within the code have had test cases which checked both the alternatives, or
- if all possible paths through the code have been checked.

Code Coverage

Given the code below what is a minimum set of test cases that would be code complete, branch complete, and path complete?

```
void method(boolean bool1, boolean bool2, boolean bool3) {
    if (bool1) {
        statement1;
    } else {
        statement2;
        if (bool2) statement3;
    }
    if (bool3) statement4;
}
```

Example Exam Questions

- What is the difference between White-Box and Black-Box testing? Explain using an example how boundary test cases may differ between white-box and black-box testing.
- Why is it generally impossible to prove the correctness of a program using testing?
- Given the code below, create a minimal test set that is path complete.

```
void method(int x, int y) {
   if (x < 7) {
      if (y >= 5) statement1;
      statement2;
   } else {
      statement3;
   }
}
```

JUnit Testing

Eric McCreath

JUnit Testing

The JUnit testing library provides a standard way of creating tests. Eclipse facilitates JUnit testing letting you build up a collection of tests for your program. All these test can be re-run easily.

There are different versions of JUnit. The approach given here uses version 4.

You would normally create another class to hold your tests. Each test would be done within a method of the class. The method is given the compiler directive "@Test", and you would normally run part of your code and check whether the results are as expected using "assertTrue" (or other JUnit assert methods like assertEquals)

JUnit Testing

Say in class MyMath we have the code:

```
static double square(double x) {
   return x * x;
}
```

We could write a simple test class as follows:

```
import static org.junit.Assert.assertEquals;
import org.junit.Test;

public class MyMathTest {
    @Test
    public void testSquare() {
        assertEquals(4.0, MyMath.square(2.0), 0.001);
        assertEquals(0.0, MyMath.square(0.0), 0.001);
        assertEquals(9.0, MyMath.square(-3.0), 0.001);
    }
}
```

For each class we wish to test we could create a test class which has test methods for different aspects of the class.

Testing - Introduction

Eric McCreath

Overview

In this lecture we will:

- consider the utility of testing,
- review different types of testing,
- checkout the code review process, and
- discuss debugging approaches.

Why bother testing your code?

Software testing encompasses a wide range of activities a software engineer might engage in to check if their software works correctly. These activities range from just compiling your code to system testing that runs software under realistic loads.

Testing will not "prove" code to be correct, however, it does provide confidence that the code is correct and it will often uncover problems within code.

Only an extremely confident or foolish programmers would not test the programs they develop.

Why bother testing your code?

Testing has many benefits, including:

- testing provides a way of checking that the code (or part of the code) is working correctly in a particular situation (we can use many test cases and inductive reasoning to provide confidence in the correctness of the code),
- testing enables us to find and isolate bugs early in the development cycle (this often saves development time),
- testing is a process we can tell clients that we have done which given them confidence that the software will work (such testing may also be a contractual requirement),
- the process of developing tests helps us reflect on the correctness of the design and implementation.

Testing will take time, and the amount of testing that is done will depend on the type of software developed.

Validation

- Validation certifies that the system meets the requirements (Building the right thing).
- Validation matches functions within the implementation back to the requirements specification and checks if the specifications are met.
- This checks whether the developer is building the correct product.
- Testing can focus on validation, so the test cases go all the way back to requirement specifications.

Verification

- Verification checks whether each function within the implementation is working correctly. (Building the thing right.)
- It will check the system against the design.
- Verification certifies the quality of the system.

The focus of testing in this course is on verification.

Unit Testing

- Each of the modules or units that make up the system should be tested in isolation. This simplifies the process of tracking down problems. It also enables more complete testing.
- Test cases and expected results will be constructed. Usually the cases will include:
  - Standard use of the code.
  - Checking boundary cases.
  - Cases that exercise all parts of the code.
  - Tests that check exception/error conditions.

Integration Testing

- Integration testing verifies that the components of the system work together correctly.
- This is like unit testing but on the entire system.
- Unit and integration testing are performed in a test area where the tests can be controlled and repeated.
- JUnit tests can also be created for integration testing.

System Testing

- System testing checks that the entire system works within the environment that it will be placed in. That is, it checks that all the hardware and software parts work with other external components in the context of normal loads.

- Ideally this would involve a duplication of the hardware/software resources. However in some cases this is not feasible so system testing is performed "live".

Test Driven Development

- If you are going to create test cases it may be worth constructing them BEFORE you develop your code. This helps you understand the requirements and reflect on your design.

- This also provides a way of verifying your testing. As you write your test cases and check that they all fail and then develop the code and check that they all pass.

- This approach is known as test driven development.

Code Review

- "All non-trivial code contains bugs". =>

"If code contains no bugs then it is trivial"

- There are many types of errors including:
  - syntax - compile time
  - runtime
  - functional
  - robustness
- The aim of testing is to uncover errors. Testing does not prove a program to be 100% correct. Invariably, your software development process will involve "Code Reviews" alongside testing.

Code Review

- A good way to uncover errors in your code is to get someone else to read over it and check for correctness, formatting, documentation, etc. This process is known as a code review.
- Code reviews have a number of advantages, including:
  - they will uncover problems and find ways of fixing them,
  - someone else can often see different ways of achieving a goal,
  - they spread the responsibility of having correct code, and
  - knowing that someone else will read over your code encourages you to write better code.
- Often organizations will have a formal process that requires code reviews before code is added to the master branch.

Code Walk Through

The idea of a code walk through is that you would present an overview of your code to a group of colleagues. You walk through the code and attempt to explain and justify your design.

This is useful in three ways:

- Firstly, as you plan/execute your walk through you can uncover errors.
- Secondly, it can also help brainstorm simpler designs and implementations.
- Thirdly, people who hear your walk through can spot errors.

Debugging & Tracking Down Errors

Tracking down errors is an iterative task. It involves:

- locating problematic code,
- generating hypotheses of what the problem might be,
- removing and refining your hypotheses of the problem, and
- fixing the code and re-testing it.

Sometimes locating problematic code can be tricky. There will usually be a difference between what you expect the code to do and what it is actually doing. So to track down these problems we can often "step" through the code to locate issues.

Performance Evaluation and Profiling

Eric McCreath

Measuring Performance

Debuggers are great at tracking down run-time problems within your implementation. With these you would add breakpoints and step through your code to work out the difference between expected and actual execution. However they are not as helpful in terms of tracking down performance problems.

The two main ways of measuring performance are:

- Adding a timer into your code so you can time how long key events take. This is particularly useful if you are comparing different approaches or if you wish to understand the variance of how long a critical activity takes.
- The other approach is to make use of profiling tools. These, like debugger, enable you to quickly track down the performance "hot spots" within your code.

Focusing Your Effort

Focus your effort ONLY on aspects that will make a significant difference. As with Amdahl's rule you will only obtain a signicant performance improvement if you improve the performance of the parts of your code that are taking a large proportion of the overall time.

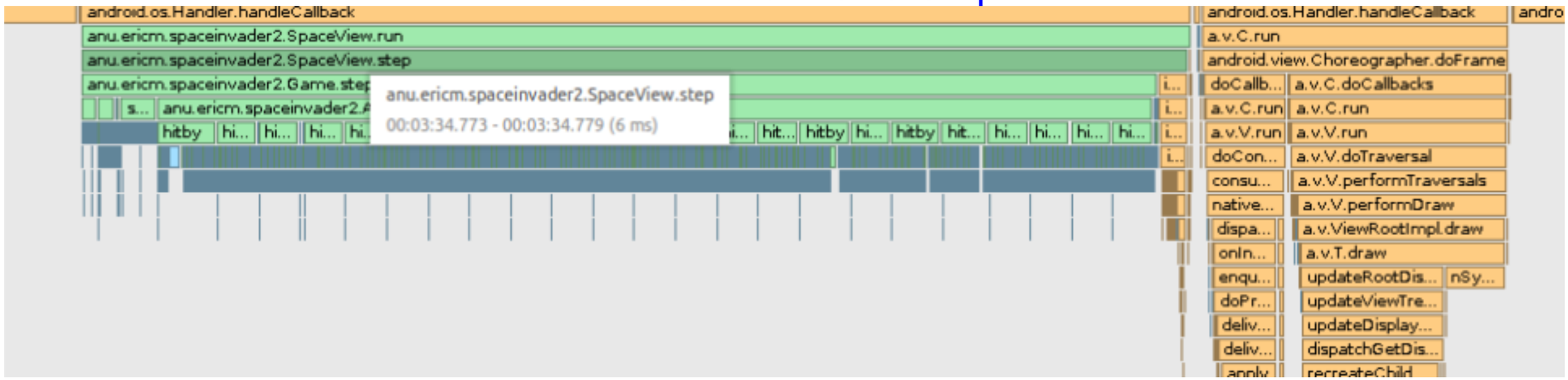
Modern CPUs run the GHz range so as a rule of thumb you can do 10^9 primitive instructions per-second. So for any method you implement consider, for your given data size, how many times per-second is the method is likely to be called and how many primitive instructions will the method take. Often this number will be many orders of magnitude less than the 10^9 , in which case implement code that is simple and robust.

Profilling Tools

A SDK will often include set of performance profiling tools. This will help you quickly evaluate and track down problems with memory, network or cpu usage.

To do the profiling you will run your application with via the "profiling" option (similar to debugger). This will record a profiling trace which you can then explore to work out what the amount of time different parts of your code are taking.

Android Studio Profile Output



Introduction to Performance Analysis

Eric McCreath

Producting good solutions

- Correctness is a key characteristic normally required in software developed. However there is often many other constraints and characteristic desirable in software development. These include: verifiable, valid, robust, tested, portable, simple to use, maintainable, documented, reusable, extendable, scaleable, efficient, and fast (in terms of time performance).
- Often different implementations perform differently in term of memory footprint and cpu usage. Applications will generally be required to work within both memory limitations and cpu time limitations. Also the less memory and cpu time a program takes the better.

Producting good solutions

- We can use complexity analysis to compare and evaluate the performance of a particular design. However there is often little scope to improve complexity. Also there can be significant differences in performance of functionally identical implementations.
- So to develop good software we may need to take performance into consideration in all the software design and development stages.

Performance

There is normally two measures we wish to focus on in terms of time performance. They are:

- response time - how long it takes from the first input to the result is complete.
- throughput - the overall rate at which work can be completed.

These need to be considered in terms of expected workloads and the scaleability of the software.

The software engineering approach you take to performance may be either reactive or proactive. The approach taken will depend on the context.

Measuring Performance

Performance constraints may be elicited from the client and incorporated into the specification and design documentation. A performance model may also be used to better understand and create an appropriate design.

Experience within a domain can be helpful in understanding key parts that will effect performance. Also experience can be useful in understanding performance characteristics of different components used.

Measuring Performance

In a similar way to testing a developer can create a framework that will systematically measure the performance of their implementation. This gives them the ability better understand and fine tune the design and implementation.

Performance analysis may be:

- best case,
- average case, or
- worst case.

Note that actual performance can be highly dependent on both the hardware that you are executing on and also dependent on the particular data used in the evaluations.

Design and Implementation Approach

One approach would be to not consider performance issues during the design and development and then to fix any problems that emerge once the system is working. This approach has a number of advantages including:

- you don't unnecessarily over complicate your design and implementation,
- you only spend time on addressing performance problems when they occur.

Although clearly there are some down sides to this approach including:

- although you can fine turn an implementation to improve on performance, often significant improvements in performance require fundamental architectural/design changes. This will involve significant cost.

Design and Implementation Approach

A more systematic approach would be to clearly state the performance requirements within the specification. These performance constraints can be included into the design such that as components are combined so the overall performance requirements are satisfied. Thus in a similar way to testing, performance can be measured and checked at different levels (unit, integration, and system).

Algorithmic approaches to solve problems

Eric McCreath

Optimization Problems

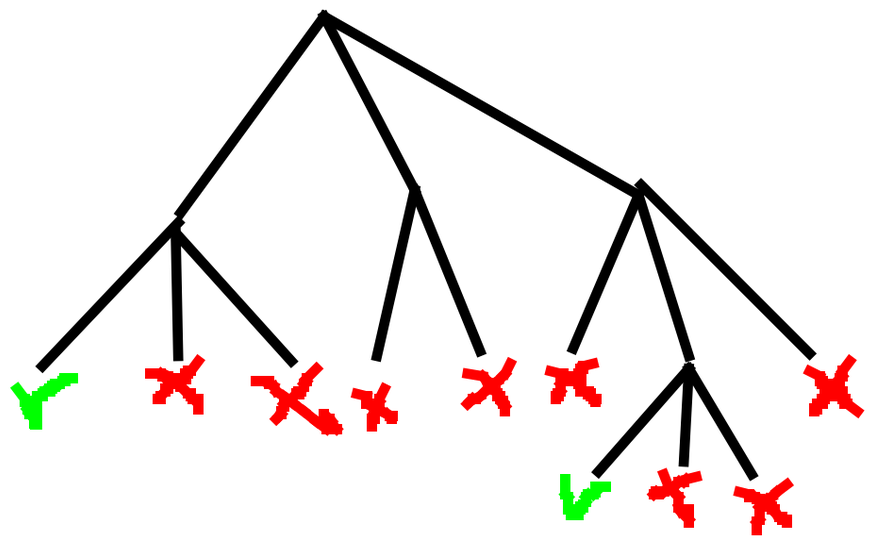
There are many problems within computing which can be categorized as optimization or search problems. So you have a set of possible solutions to a problem and you search over these possibilities finding a one (or the one) that optimizes some measure. In some situations "good", rather than optimal ones, are sufficient.

Examples include:

- timetabling students into different classes,
- finding a path between two locations,
- identifying objects within an image,
- printed circuit board design,
- determining crytographic keys,
- inducing rules for medical diagnosis, and
- planning robotic movements.

Search Tree

Searching for the best solution(s) can often be formulated as searching a rooted tree with the inner nodes being possible configuration decision points and leaf nodes measurable candidate configurations. Now to improve on search performance we can remove branches that we know will not be fruitful (or "prune" the search tree).



Brute-Force

A Brute-Force (or exhaustive) approach will consider all possible situations or combinations, evaluate the measure on these situations, and return the best (or list of best) results.

Advantages include:

- simple to implement
- simple to parallelize
- will generally have a small memory footprint
- easy to show the correctness of the approach

Disadvantages include:

- often the approach will be intrackably slow
- will not work with infinite search spaces

Greedy

A greedy approach will follow a path down the search tree by selecting the best child branch to follow based on some local heuristic. Greedy approaches will produce results quickly. In some cases greedy approaches can be shown to produce optimal results, this will require very particular characteristics of the search tree and the heuristic.

Often a greedy approach can be used to come up with "okay" solutions quickly.

Branch and bound

The *branch and bound* approach maintains the value of the best solution so far and prunes branches that it knows can not be better than the best solution so far. So as it is searches if, at an inner node, it finds that the bound of the best possible solution of children from that node are worse than best solution so far then it can "prune" these children from the search.

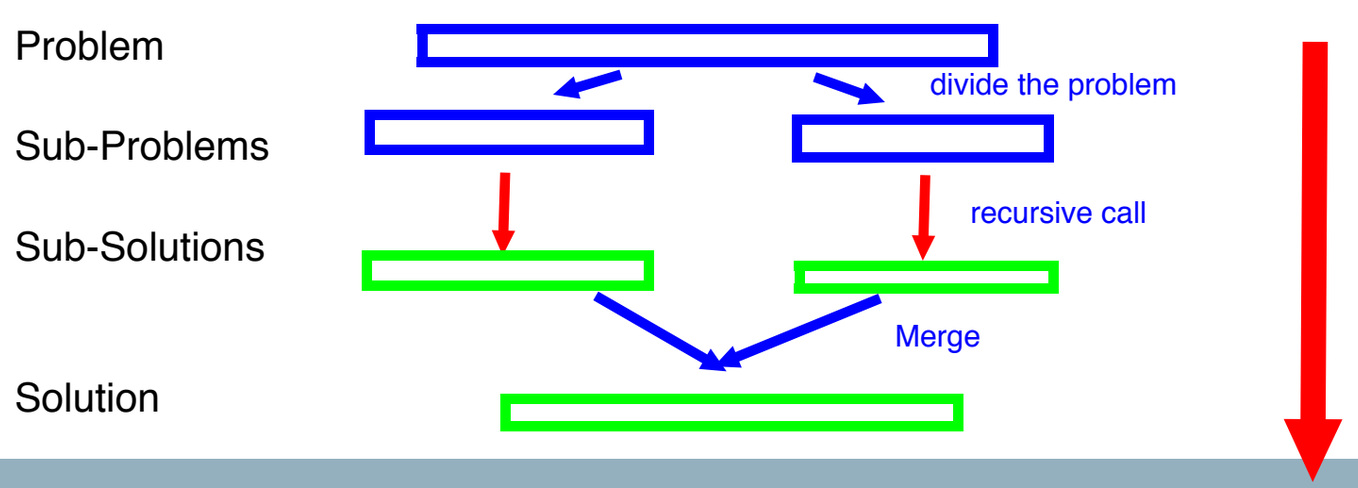
This approach will still find optimal results and will be considerably faster the a brute force approach.

This approach requires a fast way of determining bounds from inner nodes of the search.

Divide-and-Conquer

The idea of divided-and-conquer is to brake the problem into two smaller section (hopefully about the same size), recursively solve the problems for these two sub-sections. Then merge the two results into a result for the entire problem. For divide-and-conqurer to work you need:

- to be able to divide the problem into 2 section such that optimal solutions of these subsection will help you create an optimal solution for the entire problem.
- you need a merging approach, and
- also a base case (solution for a problem of size 1).



Dynamic Programming

The basic idea of dynamic programming is that you tabulate (or memorize) partial solutions for sub-problems. The results of these sub-problem solutions are combined to form larger sub-problem solutions, this is repeated until the initial problem is solved.

This can be done either using a **bottom-up** approach where you will often use loop structures to solving all the sup-problem of size 1 then using these to solve all the problems of size 2, etc .... until you solve the given problem. Or it can be done using a **top-down** approach, this would normally be done recursively where you "memorize" solutions for sub-problems, saving the repeated computation.

Often the big performance gain is made when these partial sub-problems are overlapping.

Others

There is a large array of other approaches including:

- Recursive Backtracking - Use a depth first approach on the search tree, however, the children of inner nodes are constrained to possible situations based on the configuration so far.
- A * - In this approach in each step the "fringe" of the search tree is expanded. The selection about which node on the fringe to expand is based the cost to get to that node along with the a heuristic that estimates the minimum cost from that node to a possible solution.

Complexity - Introduction

Eric McCreath

Overview

In this lecture we will:

- motive the use of complexity analysis,
- define the Big-O notation, and
- show how you can formally prove a function is an element of a Big-O order.

Motivation for Big-O

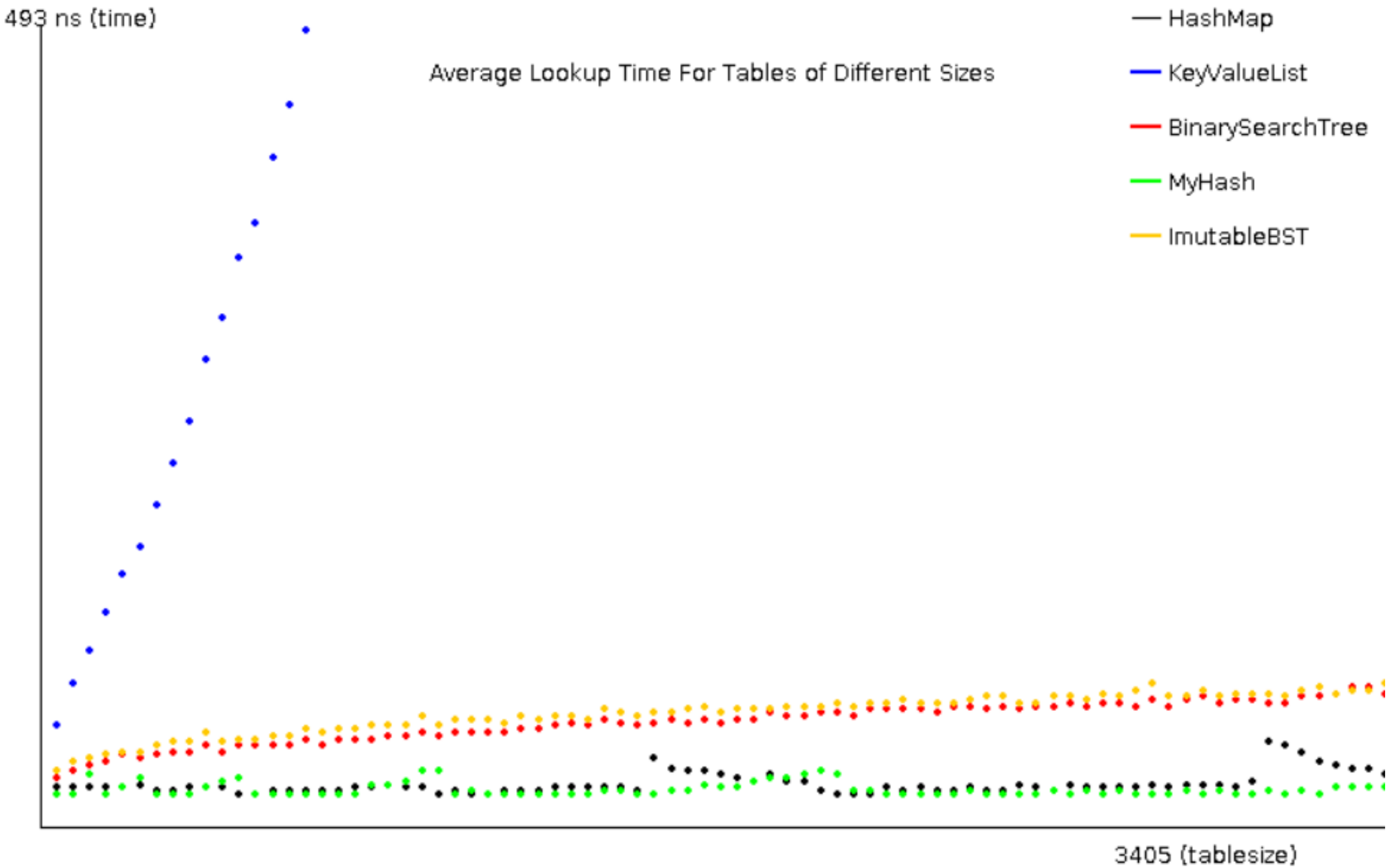
- Different programs take different amounts of time to execute. Also different programs will require different amounts of main memory. A satisfactory implementation of any problem not only must be valid, it also needs to execute within time and memory constraints of the system it is run on.

- Thus as you design a solution you need to keep in mind how that solution will scale. Will your solution cope under the expected load?

- More than anything else the 'order' of a program/method will determine how well it will scale. This lecture will introduce the idea of the 'order' of a program, more formally this is know as complexity analysis.

Motivation for Big-O

- You could implement the different approaches and compare their performance. This would give a good indication of what is the best approach. However, a good software engineer will be able to estimate the performance of a program directly from it's design and compare different approaches without implementing anything!


493 ns (time)
Average Lookup Time For Tables of Different Sizes
HashMap
KeyValueList
BinarySearchTree
MyHash
ImutableBST
3405 (tablesize)

Type of Evaluation

- The amount of time a program will take to run will not only depend on the size of the input data but also the values of the data provided.

- So you could evaluate a programs performance in terms of:
  - the best possible performance,
  - average or expected performance, or
  - the worst possible performance,

over the possible input data it is run with.

- If your program runs fast enough in the "worst case" then it will also be okay in other (better) cases. So often programmers will evaluated the implementations/algorithms in terms of worst case analysis. This is generally the simplest form of analysis.

Count the primative statements

- A simple approximation of how long a program will take to execute is to count the number of primitive statements executed.
- Each primitive statement will take a different amount of time to execute. However as there is a limited number of primitive statements, all these statements will execute in less than some fixed maximum amount of time. Let us denote this fixed maximum amount of time as t_{max} . Now if our program takes TS primitive statements to execute then the time taken to execute the program will be less than: $t_{max} \times TS$.

Constant Time

- This method measures the distance between two xy-points.

```
public Double distance(XYPoint p) {
    Double xdiff = p.x - x;
    Double ydiff = p.y - y;
    return Math.sqrt(xdiff*xdiff + ydiff*ydiff);
}
```

- Now if we assume maths operators count as one primitive statement and also the assignment and return statements count as one primitive statement then we would always have:

$$TS_{distance} = 9$$

- This method will not get any slower for different xy-points.

Mean method

- The following method calculates the mean of an ArrayList of Doubles.

```
public class MyList extends ArrayList<Double> {
  public Double mean() {
    Double sum = 0.0;
    for (Double d : this) {
        sum += d;
    }
    return sum/size();
  }
}
```

- The number of statement executed will depend of the size of the list.
- Let $n = \text{the size of the list}$.
- We can now count the number of statements in terms of different values of n .

Mean method

- In addition to the primative statements in the previous slide, If we also assume
  - the "foreach" statement takes one primitive statement to move to and obtain the next element in the list, and
  - olbtaining the "size" of the list is also just 1 primitive statement.

In the code below I have added comments at the end of each line with the primitive statement count for that line.

```
public class MyList extends ArrayList<Double> {
  public Double mean() {
    Double sum = 0.0;                    // 1
    for (Double d : this) sum += d;      // 3*n
    return sum/size();                   // 3
  }
}
```

If we add all this up we end up with $TS_{mean}(n) = 4 + 3n$

Focus on the biggest contributor

- Consider the number of statements required for the method for calculating the mean:

$$TS_{mean}(n) = 4 + 3n$$

- As n gets bigger the constant component becomes insignificant in terms of how long this method takes to run.
- The idea of the Big-O notation is to focus on the most significant component of the time required to execute the method/program. Also to not worry about constant factors.
- You would say the mean method is of order n . Or $TS_{mean}(n) \in O(n)$
- The distance method is order 1 or $TS_{distance} \in O(1)$

Big-O Definition

- Suppose n is the size of the input data the method is given.
- Let $TS(n)$ be the number of statements executed in the method.
- Then we would say:

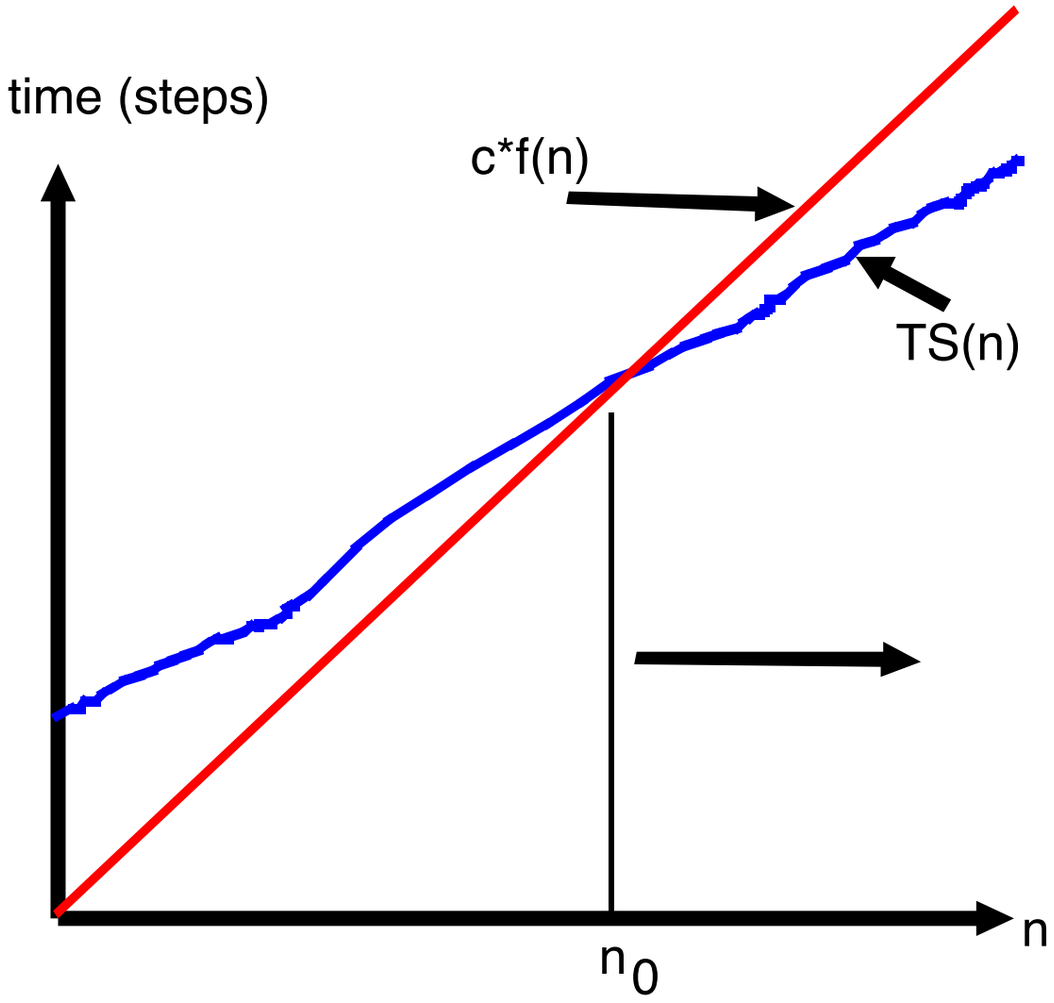
$$TS(n) \in O(f(n))$$

if and only if there exists a constant $c > 0$ and a constant $n_0 > 0$ such that for all $n > n_0$:

$$TS(n) \leq c \times f(n)$$

Big-O

So we are looking for constants c and n_0 such that for values larger than n_0 the blue line (TS(n)) is below the red line (c * f(n)). In the case below we have f(n)=n



Proving a function is a member of a particular order

- So if we considered the number of steps the mean method took to execute:

$$TS_{mean}(n) = 4 + 3n$$

we know:

$$4 + 3n \in O(n)$$

however, to formally prove this we need to find constants c and n_o such that for all values of n greater than n_0 we have the inequality in the definition hold true. In this case $n_0 = 1$ and $c = 7$ would work.

Proving a function is a member of a particular order

- To prove:

$$4 + 3n \in O(n)$$

we "guess" the constants and deductively build from simple inequality statement we know to be true until we get to the inequality as required by the definition.

So lets set $f(n) = n$, $n_0 = 1$ and $c = 7$.

We know:

$$\begin{aligned}
1 &< n && \text{for all } n > 1 \\
4 &< 4n && \text{for all } n > 1 \\
4 + 3n &< 4n + 3n && \text{for all } n > 1 \\
4 + 3n &< 7n && \text{for all } n > 1 \\
4 + 3n &< cf(n) && \text{for all } n > n_0 \quad \text{QED}
\end{aligned}$$

A few things to note

- The big-O notation describes a set of functions, this is why the set notation is often used with the big-O notation. More traditionally people used the $=$ symbol rather then the $\in$ symbol, however, this is a misuse of the normal way we use equals.

- Say we have a program that executes $7n^4 + 100$ steps, for a particular n . We would normally just say it is $O(n^4)$ as $7n^4 + 100 \in O(n^4)$. Yet, it is also the case that $7n^4 + 100 \in O(7n^4)$. or even $7n^4 + 100 \in O(7n^4 + 100)$.

However, as the idea of the big-O notation is to remove constant and less significant factors thus you would not normally include them in your statement of the complexity.

A few things to note (cont.)

- The big-O notation acts as a maximum. So for example it is true that $7n + 2 \in O(n^4)$. However, one would normally give the lowest possible maximum for a particular method or program. So if the number of primitive statements was $7n$ then you would normally say such a program is $O(n)$.
- The big-O notation provides a natural way of ranking approaches. So method or program that is $O(n)$ would run faster than one that is $O(n^2)$ (at least for all n larger then some fixed constant)

A few things to note (cont.)

Different complexities can be ordered:

$$O(1) \subset O(\log n) \subset O(n) \subset O(n \log n) \subset O(n^2) \subset$$

$$O(n^{2.376}) \subset O(n^3) \subset O(n^4) \subset O(2^n) \subset O(n!) \subset$$

$$O(n^n) \subset O(2^{2^n})$$

Tractable and intractable problems

- Problems for which there is an algorithm, yet, the implementation of the algorithm would take too long to run are said to be intractable. To clarify this division Computer scientists have said that problems for which there is a polynomial solution are considered tractable. And problems for which there is no polynomial solutions are said to be intractable.

- So sorting a list can be done in $O(n^2)$ where n is the number of elements to sort. So the task of sorting is said to be tractable.

- Whereas, the task of finding a variable assignments that makes a Boolean formula true (this is known as the Boolean satisfiability problem) does not have a polynomial solution (at least no one has found one yet!). This task of Boolean satisfiability is said to be intractable.

Calculating the Complexity of Some Code

Eric McCreath

Overview

In this lecture we will:

- show how the complexity of some code can be quickly evaluated, and
- explain how we can evaluated the complexity of a recursive method.

Evaluating the complexity

- To calculate the complexity of some code we could carefully sum all the primitive statements, then delete the less important factors and constants producing the big-O complexity.
- In the example below we calculate the biggest gap between any two integers in the a list of size n .

```
static Integer biggestGap(ArrayList<Integer> list) {
   Integer biggestGap = null; // 1
   for (int i = 0; i < list.size(); i++) { // 2n + 1
      for (int j = 0; j < list.size(); j++) { // 2n*n + n
         int gap = Math.abs(list.get(i) - list.get(j)); // 5n*n
         if (biggestGap == null || biggestGap < gap) { // 3n*n
            biggestGap = gap; // n*n
         }                    // given we are doing worst case
      }                       // analysis we assume the if
   }                          // contition always evaluates to true
   return biggestGap; // 1
}
```

This gives us: $TS_{\text{biggestGap}} = 11n^2 + 3n + 3 \in O(n^2)$

Evaluating the complexity (cont)

As we get more practice at doing this we can just note the order of each line. This is because we can just ignore constant values.

```
static Integer biggestGap(ArrayList<Integer> list) {
   Integer biggestGap = null; // O(1)
   for (int i = 0; i < list.size(); i++) { // O(n)
      for (int j = 0; j < list.size(); j++) { // O(n*n)
         int gap = Math.abs(list.get(i) - list.get(j)); // O(n*n)
         if (biggestGap == null || biggestGap < gap) { // O(n*n)
            biggestGap = gap; // O(n*n)
         }
      }
   }
   return biggestGap; // O(1)
}
```

These can be quickly combined to give $O(n^2)$.

Evaluating the complexity (cont)

Aspects to be careful of:

- if you summed together n lots of $O(1)$ it becomes $O(n)$,
- if a method calls another method it may not be "primitive". So in the previous example if we just changed " ArrayList" to "LinkedList" then the complexity of biggestGap would become $O(n^3)$. This is because the get method goes from $O(1)$ to $O(n)$.

Finding the Complexity of a Recursive Method

We may wish to calculate the complexity of a recursive method. We can do this by working out the formula for the number of primitive operations that the method would execute for a given input size. This formula would be recursive also. Now once this is done we can solve (or just look up) the analytic solution for the recursive formula.

Say we had the recursive method for calculating the factorial of a number:

```
public int factorial(int n) {
   return (n==0?1:n*factorial(n-1));
}
```

So the formula for the number of primitive operations is:

$$TS_{\text{factorial}}(n) = \begin{cases} 2 + TS_{\text{factorial}}(n-1) & ,\text{if } n > 0 \\ 1 & ,\text{otherwise.} \end{cases}$$

Finding the Complexity of a Recursive Method (cont)

By repeated substitution we can work out the formula:

$$TS_{\text{factorial}}(n) = 2n + 1$$

From this we know the complexity is

$$TS_{\text{factorial}}(n) \in O(n)$$

There is a number of "common" formulas that we see in recursive methods. With these we can look up the solution. They include:

$$\begin{aligned}
TS(n) &= TS(n-1) + O(n) && \Rightarrow O(n^2) \\
TS(n) &= TS(n/2) + O(1) && \Rightarrow O(\lg n) \\
TS(n) &= 2TS(n/2) + O(1) && \Rightarrow O(n) \\
TS(n) &= 2TS(n/2) + O(n) && \Rightarrow O(n \lg n)
\end{aligned}$$

Amdahl's Law

Eric McCreath

Definition

Amdahl's Law was proposed by Gene Amdahl in 1967 and is a simplistic formula for estimating the expected speedup as more resources are added to a part of a fixed workload. So a program could be partitioned into a serial stages and parallel stages (or stage you can add resources to make run faster). The the parallel stages are assumed to be perfectly parallizable (n processors means n times faster). Whereas the serial stages are not helped by adding more processors.

Let p denote is the percent of time a program spends in parallelizable stages and n denote the number of processors then Amadah's Law states:

$$speedup(n) = \frac{1}{(1-p) + \frac{p}{n}}$$

Derivation

Say T_1 is the total time to complete a fixed task using 1 processor. This time could be divided into the time for the parallelizable and serial stages:

$$T_1 = (1-p)T_1 + pT_1$$

Now if we had n processors then the time to complete the task is:

$$T_n = (1-p)T_1 + \frac{pT_1}{n}$$

So the speed up can be calculated:

$$speedup(n) = T_1/T_n = \frac{1}{(1-p) + \frac{p}{n}}$$

Examples

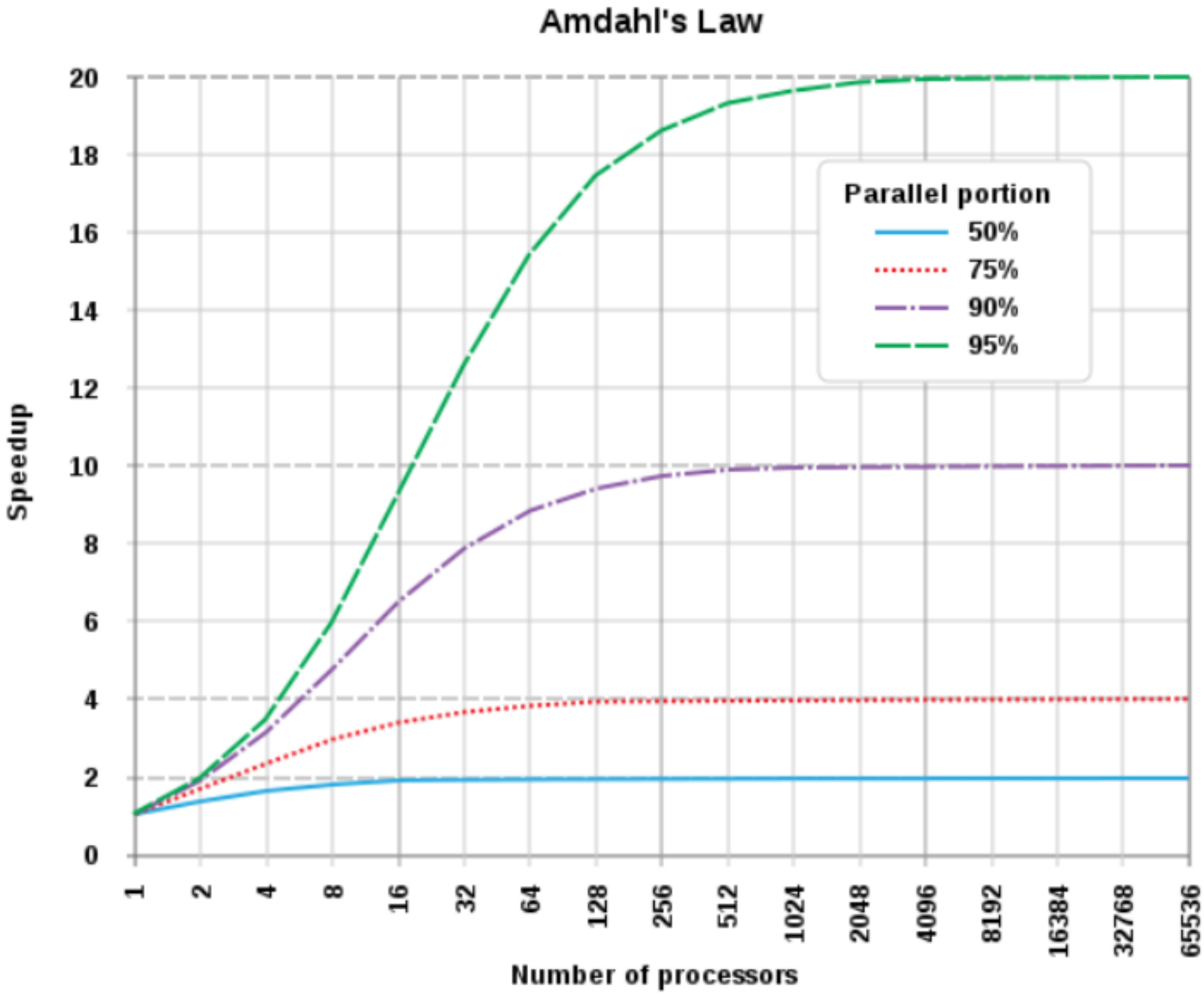
Lets say we had a program that had 10% parallelizable code. And we used 20 processors to help improve speed. Then according to Amdahl's Law the speed up would be:

$$\frac{1}{(1-0.1)+0.1/20} = 1.105$$

Whereas, if we had a program that had 80% parallelizable code. And we only use 10 processors to help improve speed. Amdahl's Law would give us:

$$\frac{1}{(1-0.8)+0.8/10} = 3.571$$

Graphs



Implications and limitations

- As you add more processors you will start to get diminishing returns. So even with an "infinite" number of processors your speedup will be limited.
- You need to add more processors to the code that takes a large proportion of your execution time.
- It is pessimistic as it doesn't capture the ability for a programer to change the algorithm/approach taken which would increase the proportion of the execution time that is parallelizable.
- It is optimistic in that at does not capture synronization overheads association parallel approachs (or other bottlenecks like memory throughput).