# The Hitchhiker's Guide to Interviews*

How to apply in interviews what you've learned in INFO6205

* **With apologies to *The Hitchhiker's Guide to the Galaxy,* by Douglas Adams**

## Part One

## DON'T PANIC!

- This is the famous catch-phrase from *HG2G*
- But, it applies especially to interviews:
  - When you are asked a question in an interview, you will be very aware that the clock is ticking.
  - If you don't answer right away, you'll look like an idiot, right?
  - Wrong! If you like, tell them that you're going to think the problem through. They will respect you for it.
  - Software Engineering is not a video game where you have to make split-second decisions!
  - It's a discipline in which thought and planning are valued!

# Strategy

- Is the question asking something you're expected to know? Or is it a problem/challenge?
  - If it's something you should know, make sure you recall the answer to *this* question before answering. Don't, for example, mix up different types of sort.
  - If it's a problem, start out by thinking the problem through…
  - … and asking clarifying questions—these questions, far from making you look like a fool, make you look really smart.
- What is the questioner looking for?
  - They want to know how you *think!* In particular, they want to know how you approach problems.
  - Let them in on the secret by talking through your solution.

# Strategy (1)

- Is it a trick question?
  - Almost certainly not.
  - But it will be a question designed to find out how you solve it, so it probably won't reveal all its secrets at once.
  - You may have to ask quite a few clarifying questions to figure out what's really going on.
- Start with the most obvious approach (usually brute-force), explaining what you're doing.
  - Now, think through the kinds of optimizations that we have learned in class.
  - Remember: premature optimization is the root of all evil—especially in the coding interview!

# Strategy (2)

- Logic
  - Stay with logic. Try not to take flights of fancy or guess the answer. Cold logic will get you to the right place.
  - Don't give the answer you *think* they are looking for—again, usually these will not be trick questions.
  - *Do* try to understand what they're looking for by examining any unusual conditions or rules they have specified.
  - Don't jump to conclusions and don't make assumptions. Question your assumptions—"is it OK to assume that…?" Remember: ASSUME makes an ass* out of you and me!

*** ass = donkey**

# Strategy (3)

- Stop digging!
  - Unfortunately, sometimes you will find yourself in a rat-hole. Or you paint yourself into a corner. Your approach is failing to reduce the search space.
  - Admit it to yourself (and your interviewer) and say that you're going to take a different approach. That's OK. What's not OK is to keep trying to force the wrong approach into a solution. When you're in a rat-hole, don't keep digging!

## Strategy (4)

- Invariants
  - Look for invariants that might help you to understand what's going on.
    - Is there a common theme to different aspects or phases of the problem as stated?
    - Is there something you can *take advantage of?*
  - For a good example of looking for an invariant, see this YouTube video from 3Blue1Brown.

# Solution Types

- Brute Force
  - Your first line of attack should normally be brute force.
  - You should only give up on a brute force approach if the performance will be too costly.
- Reduction
  - Your next thought will be to reduce the problem into a set of simpler problems (see other slides on Reduction).
  - An example is merge sort (divide-and-conquer).
  - A reduction technique will necessarily introduce a set of *invariants*. In the case of merge sort, the invariant is that, after recursively sorting each partition—and before merging—we can state that each partition is ***in order***.
- What if there is no way to reduce complexity?

# General Principles

- You should be confident of the basics before your interview—the general principles of programming.
- What drives processing time? What causes complexity?
  - If the time taken for some solution is, say, $c\ N^k$, then what is that really saying? What is it that takes up all the time?
  - The short (and over-simplified answer) is *array lookups*. Or, more generally, memory references.
- What about compares, equals, hashCode, etc.?
  - Well, much of the time taken for one of those operations is just for getting the value(s) from memory into place.
  - This is why the behavior of the caching system is so important.

# General Principles (1)

- The Dictionary Principle
    - Usually, the simplest and most effective optimization to reduce complexity from, say, $c\ N^{k}$, to $c\ logN\ N^{k-1}$, is the application of "The Dictionary Principle."
    - First sort the elements and then use some sort of search-space-halving strategy to find the solution (typically binary search with/without binary tree).
    - **If the number of searches you expect to do pays off the extra effort you put in to do the sorting, then you come out ahead!**

# General Principles (2)

- Can you explain where this *log* term actually comes from?
  - Let's say that you are searching in an unordered list: every element you add to the list increases the subsequent search time according to the same marginal rate: $\Delta t \sim \Delta N$. Thus the total time will be $\sim N$.
  - But suppose you are searching in an ordered list: each additional element in the list increases the time by what fraction of the total list size it represents: $\Delta t \sim \Delta N/N$. Thus the total time will be $\sim \log N$.

# General Principles (3)

- Entropy—the degree of disorder of a collection:
  - Entropy is, loosely, how much randomness there is in a dataset. The role of algorithms and data structures is, usually, to try to *reduce entropy* to make information more readily available. Reducing entropy requires work!
  - In general, the minimum number of compares required to sort a collection of *N* elements is *lg (N!)* which is to say *~N lg N*.
- Invariants:
  - Relationships which must hold true while a data structure is in a steady state (i.e. not undergoing a transformation).
  - Example: in a binary heap, elements are in *heap order,* except while actually undergoing a deletion or insertion.
- Inversions:
  - In a list that you would like to be ordered, an inversion is when *any* two elements are out of order (they don't have to be adjacent).

# General Principles (4)

- Reduction—a fundamental approach to solving problems:
  - Reduction is breaking up hard problems into a set of smaller/easier problems and transforming the solution(s) back into the domain of the original problem.
  - Perhaps the most common type of reduction uses recursion—we do some work, test if we’re finished and, if not, try to solve the smaller problem using the same method.
  - Code looks better with recursion rather than iteration. But it also uses extra memory (on the “stack”). The stack has finite capacity and with complex problems you can easily suffer a “stack overflow.”
  - Therefore, if you *can* do it with equivalent elegance, or if you think you may overflow the stack, use iteration instead. More on this later. Example: binary search.

# General Principles (5)

- Assumptions and arbitrary decisions
  - Always carefully examine your assumptions and any *arbitrary decisions* that you have made in coding. Choosing the order to do two operations can make a profound difference to the effectiveness of your code. Remember *ASP**.
  - Don't guess. Don't make unnecessary or unjustified assumptions—these will, unfortunately, always come back to embarrass you later.
  - Take care also of your implicit assumptions—for example, did you declare something as an *int* when perhaps it should be a *long?*

* arbitrary substitution principle

## The conditions and requirements

- Examine the *conditions* and *requirements* of the exercise. Most probably, they have been chosen carefully. Think about what the problem setter wants you to do. But don't assume everything is a trick.

- Do a back-of-the-envelope calculation to check the domain of the problem: for example, if it involves the cartesian product of classes of size 10^6 and 10^5, then an enumeration of the resulting domain cannot fit in a 32-bit integer.

# Visualize the problem and any answer you may get

- A picture, as they say, is worth 1000 words. If possible sketch the problem as you see it before trying to solve it.
- Use your common sense: does your answer *look and feel* right? Can you draw the solution as part of your sketch of the problem?

# Complexity

- In general, the time and memory required to solve a problem will grow as ~$f(N)$ where $f(N)$ is a polynomial in $N$:
  - i.e. $f(N) = c_0 + d_0 \ln N + c_1 N + d_1 N \ln N + c_2 N^2 + d_2 N^2 \ln N + \ldots$

- You can of course express this in tilde (~) notation by ignoring all but the highest powers of $N$:
  - For example: $c_2 N^2$
  - or: $d_2 N^2 \ln N$

# Estimation

- Engineers are expected to be good at estimation.
  - If you are asked to estimate the number of compares using merge sort, for example, you know it's ~N lg N.
  - If the question says that N = 1,000,000 then you should know right away that *lg N* is 20. No need for a calculator. It will just make you look like you don't get it.
  - Similarly, if the question is about quicksort where the average number of compares is *2 N ln N*, you should know that *2 ln N* is about 28 (i.e. 2 x 2 x 7).

# Attacking complexity

- Let's suppose that the complexity of an algorithm is $c\,N^k$.
- You can lower this by making any of the following smaller: *c*, *N* or *k*.
  - **c**: minimizing *c*, for example, running on a faster computer, will help a bit. But it won't do much for you as *N* gets larger and larger.
  - **N**: you can't do much about *N* other than use divide-and-conquer (see next slide).
  - **k**: you can't do anything *directly* about *k*—but you might be able to optimize the algorithm (see next slide).

# Optimizations

- Reduction: divide-and-conquer
  - Can we transform the given problem A into a set of easier problems **B** while being able to transform the solutions **B*** back into the original problem domain, i.e. A*.
  - If, for example, we can divide a problem of $N$ elements into $r$ independent partitions, solve each one, then do a *linear* amount of work merging the results, the time complexity will be ~ $N \log_r N$.

- Reduction: the dictionary principle (using order)
  - If we can replace a linear search on $N$ elements, for instance, with a binary search (which requires ordering the elements), we can reduce ~$c\ N$ to ~$d\ \lg N$.
  - Sort once—search many: amortizes the cost of searching.

# Optimizations (2)

- Reduction: Memoization (caching)
- Sometimes we can use extra memory to hold results which would otherwise need to be recalculated each time:
  - This is the basis of dynamic programming, BTW;
  - Internal memoization, for example, caching the hash value of a *String* in Java;
  - Symbolic memoization: setting up a symbol table (cache) of values which can be retrieved (or updated) using a key;
  - An example of the latter is when we solved the *3-sum* problem, we set up a symbol table of element pairs, using their sum as the key.

# Fake Optimizations

- All optimizations need to be tested!
- Try each optimization on its own. Never combine.
- Don’t start out by thinking that something will need optimization (“premature optimization”). Get the logic right first, and *then* think about optimization.
- The compiler will perform a lot of optimizations for you—don’t make its job harder with silly things like writing *x*+*x* when you mean *x**2. That just obfuscates the code for the next person.

# Coding Challenges

- Write code that is simple, obvious and elegant (SOE).
  - It’s unlikely that the code they want you to write requires any obscure cases or anything like that—but you must consider the usual situations like upper vs. lower case.
  - The more elegant your code:
    - the more likely it will work; and
    - the more likely that if it doesn’t work, you’ll be able to see the problem; and
    - the more the interviewer will be able to see what you’re doing (and hopefully be impressed).
  - Elegant coders are the kind of coders they are looking for!
  - For bonus points—once done, explain (or comment) what assumptions you made and what limitations your solution will have.

# Coding Challenges (2)

- So, you’ve been asked to code quicksort and the test doesn’t run. Where did you go wrong?
    - If there are multiple tests and some are right, try to understand why those particular ones succeeded while others failed.
    - For example, are the successful tests working with empty or singleton collections?
    - Or.. is it only the even-handed collections that work, not the odd ones (i.e. *N%2==0 or 1).*
    - Could your index be off by one?
    - If you have the opportunity to write your own tests, use them to separate working cases from failing cases.

# Coding Challenges (3)

- Checklist for debugging:
  1. *For* loops, *While/until/do* loops: are the starting value (if any) and the “while” condition valid? Could any of them be off by one?
  2. Recursion: is the terminating condition arising at the correct point? Could it be off by one?
  3. When comparing or assigning array values: are the indexes correct? Could one of them be off by one?
  4. When using *compareTo* or *less*: do you have the operands (*this* and *other*) in the correct order?
  5. Incrementing variables: if you are using ++ or --, are you putting it on the correct side of the variable (i.e. pre- or post-)? Have you remembered to increment/decrement every variable that should be changed?

# Coding Challenges (4)

- So, you looked for a bug and you found something.
  1. You “fixed” what you found;
  2. Only, it didn’t solve the problem 😢
  3. Did you *clearly* get closer to the solution?
     1. Example: the algorithm never terminated before but now it does
     2. Then you can leave the “fix” that you made.
  4. Otherwise,
     1. Always *revert* the change you made because, if it didn’t fix the problem, why would you want to keep it?

# Techniques to avoid Stack Overflows

- Stack Overflow:
  - While recursion is usually the most natural and elegant manner in which to implement an algorithm, it is unfortunately not safe when the depth of recursion *h* gets large.
  - As mentioned previously, it’s possible to “unroll” a recursion and replace it with an iteration. If done properly, this will avoid stack overflows.
  - The basic technique is called *tail call optimization.* You ensure that the result of any recursive call is passed back immediately to the caller, without any further processing, thus requiring nothing on the stack.
  - Scala (and other FP languages) handle this properly. Unfortunately, Java doesn’t
  - However, you can transform your tail-recursive code into a *while* loop.

## Part Two

## 20 Questions

- The following questions are samples of the kind of thing I would ask if I were interviewing you. You should be able to talk intelligently about all of these subjects. I have not provided answers because, generally, the answers are not simple short phrases.
- If you don’t feel confident of talking about the subject, review it!

# 20 Questions (1)

1. Why might you use Heapsort instead of Quicksort?
2. In the context of sort algorithms, explain what is meant by “inversion.”
3. How many possible permutations of a collection of *N* items are there?
4. What is the essential difference between a binary heap and a binary search tree? Hint: it’s not about arrays versus pointers.
5. Why is it so much more efficient to use quick-select (or heap-select) to find the median of a collection versus doing a sort and then picking the middle item?

# 20 Questions (2)

6. What is the average number of inversions in a randomly-ordered collection of *N* items?
7. Why might it be possible to search a tree more efficiently than a list? In other words, what is it about the *structure* of a tree that allows for more efficiency?
8. In the study of data structures and algorithms, what is the significance of an *invariant*?
9. Insertion sort and selection sort each use a different kind of swap (exchange) operation. Explain why selection sort requires only *N* swaps while insertion sort requires (on average) $N^2/4$ swaps.
10. A divide-and-conquer algorithm partitions a problem of size *N* into *k* sub-problems and the number of operations required to combine the *k* sub-solutions together is *N*. A problem of size 1 requires no work at all. What's the total number of operations required to solve the original problem?

# 20 Questions (3)

11. What is the difference between a tree and a graph?
12. Which algorithm is generally more useful for traversing a graph: breadth-first search or depth-first search?
13. Why is a *bag* a suitable data structure in which to store the adjacent vertices of a vertex in a graph?
14. When considering orders of growth, is it possible to have an order of growth intermediate between linear and logarithmic?
15. Arrays are excellent for implementing lookup tables where the key is an integer in a finite range: how can they be extended to be useful for symbol tables in general?

# 20 Questions (4)

16. What is a greedy algorithm?
17. Why is recursion sometimes considered bad coding practice and what can you do about it?
18. Which is more beneficial: improving an algorithm from O(N) to O(log N) or improving from O(log N) to O(1)?
19. Explain the concept of a “collision” as it relates to symbol tables, and what are the two principal methods used to deal with collisions.
20. Why does Java use (dual-pivot) Quicksort for primitives but Timsort for objects?

## Part Three

# Case Study: Reaching Points

- LeetCode problem #780 (Reaching Points)
  - Looks simple but is not (it is marked "hard").
  - In a two-dimensional integer grid ($0 < x, y < 1{,}000{,}000{,}000$), you are given two points, *s* (start) and *t* (target).
  - You are to determine if there is a path from *s* to *t* where there are only two legal moves from a point *p* to point *q*: **Visualize this!**
    - $q_1 = (p_x+p_y, p_y)$ and $q_2 = (p_x, p_x+p_y)$

## Reaching Points #1

- First, let's check for *invariants*:
    - We are given one invariant that defines the transition from $p_i$ to $p_{i+1}$.
    - Is there an invariant which governs the transition from $p_{i+1}$ to $p_i$?
    - While you're thinking about that, let's get started with the most obvious, brute-force approach…
- Start out by, essentially, rewriting the problem in (pseudo-)code

    -
      ```
      public boolean valid (Point p) =
                                    p.equals(t) || (valid(q1) || valid(q2))
      ```

# Reaching Points #2

- Let's *visualize* the problem:
  - In this case, we start at 1,1 and show the various squares reachable within the 5x5 grid. The subscript *x* means we maintain *x* and replace *y* by *x+y*. The subscript *y* means we maintain *y* and replace *x* by *x+y*.
  - One step: 2 possibilities; two steps: 4; three steps: 8; four steps: 16 (only 4 visible).
  - Note well: there are six unreachable squares.

| | **1** | **2** | **3** | **4** | **5** |
|---|---|---|---|---|---|
| **5** | $t_{xxxx}$ | $t_{yxx}$ | $t_{xyx}$ | $t_{yyyx}$ | — |
| **4** | $t_{xxx}$ | — | $t_{yyx}$ | — | $t_{xxxy}$ |
| **3** | $t_{xx}$ | $t_{yx}$ | — | $t_{xxy}$ | $t_{yxy}$ |
| **2** | $t_x$ | — | $t_{xy}$ | — | $t_{xyy}$ |
| **1** | S | $t_y$ | $t_{yy}$ | $t_{yyy}$ | $t_{yyyy}$ |

# Reaching Points #3

- Let’s just think about the complexity of the problem— How many possible moves could there be?
    - The only measure of complexity we have is the size of the allowable grid, i.e. N = 1 Billion.
    - Each move goes in either the horizontal or the vertical direction and it’s length is (more or less) proportional to its current distance from the origin.
    - In other words, successive lengths are double the sum of the previous lengths. This suggests that the maximum length of a path is *lg N*.
    - Unfortunately, the number of possible routes is exponential, i.e. $2^{\lg N}$, i.e. N, that’s to say 1 Billion. That’s not a trivial problem.
    - The search space of 1 Billion possible has to be reduced to one.

# Reaching Points #4

- The test conditions are:
    - `1,1->1,1: true`
    - `1,1->2,2: false`
    - `1,1->3,5: true`
    - `9,5->12,8: false`
    - `1,1->99,100: true`
    - `35,13->455955547,420098884: false`
- The first four of these are easy to accomplish with the code already discussed.
- But, in order to satisfy the fifth test, you must transform your recursion into an iteration because you will surely blow the stack.

# Reaching Points #5

- At this point in the interview (if you’re in one):
  - you will say that your algorithm is correct but in order to work with more realistic problems, you must transform it to an iteration.
  - The way you do that is first to see if your algorithm is *tail-recursive*—it isn’t (when you have two recursive calls it never can be).
  - So, what we have to do is to separate the work that still needs to be done (in this case, points to be tested) from the result (normally this will be accumulated call-by-call but in this case, once we’ve found a path, we are done).

# Reaching Points #6

- So, our algorithm will look something like this:

```java
public boolean valid(int x, int y) {
    Queue<Point> points = new Queue_Elements<>();
    points.enqueue(new Point(x,y));
    return inner(points, false);
}
private boolean inner(Queue<Point> points, boolean result)
{
    if (points.isEmpty()) return result;
    Point x = points.dequeue();
    if (x.equals(t)) return true;
    if (x.x>t.x || x.y>t.y) return inner(points, false);
    points.enqueue(new Point(x.x,x.x+x.y));
    points.enqueue(new Point(x.x+x.y, x.y));
    return inner(points, result);
}
```

## Reaching Points #7

- This algorithm is a big improvement:
  - It solves the 5th test, although not without some definite running time.
  - It still gets nowhere near running the 6th test.
  - At this point, you begin to suspect that you are in a rat hole and you should stop digging. But you see a couple of rays of hope.
  - Did you notice a certain *arbitrariness* in the last algorithm? (ASP)
  - You had to put both possible successors into the queue and you gave no particular thought which one should come first (and be acted on first). But, if you think about it, it could make a world of difference which path is followed first.
  - Therefore, you resolve to choose first the path that gets you closest to the target.

# Reaching Points #8

- In the rat hole with a couple of rays of hope:
  - What’s the other hope? It’s a slender chance and, even as you code it, you know it won’t solve the problem: memoization of points previously visited and eliminated. A simple cache where all we care about is finding or not finding the key (i.e. no value), is a *Set*. The *contains* method uses binary search.
  - For caching (memoization) to save much time, you have to have a high frequency of duplicates. I don’t think that’s the case.
  - As predicted, this makes no noticeable difference in performance and so we hope that choosing the most direct path (#5) will do it (but our instinct is that it won’t work).

## Reaching Points #9

- This algorithm (choosing most direct path) is a moderate improvement:
    - The 5th test now runs instantaneously.
    - It still gets nowhere near running the 6th test. Why? It must have to do with the exponential nature of the solution space. After *N* moves, you could be at any one of $2^N$ points (not adjusting for duplicates)
    - At this point, you *know* that you are in a rat hole and you *must* stop digging. This is the point at which you tell your interviewer that you've come as far as you can down this path and it's time to back up and cast around for a radically different solution.

# Reaching Points #10

- You’re stuck!
  - You have exhausted all possibilities for improving a strategy where you begin at the start and work your way to the target.
  - Go back and think about the problem conditions:
    - Why did they specify that the domain of the problem is the positive integers? Could this be a clue?
    - Why did they choose the particular type of move that they did? It’s like a very strange knight’s move in chess.
    - Is there anything about the move that jumps out at you?
    - What if you were to begin at the *target* and work your way back to the *start*?

# Reaching Points #11

- Invariant-based solution:
  - Did you come up with a useful invariant for going backwards?
  - How could it make any difference if you began at the target and worked towards the start? You still have two possible moves from each successive point, right?
  - **No, you don’t!** If you think about it, there’s only one possible point from which the target can be reached. It is at *either* ($t_x$-$t_y$, $t_y$) *or* ($t_x$, $t_y$-$t_x$) (<u>but it cannot be both</u>).
  - Why not? Because *either* $t_x$-$t_y$ *or* $t_y$-$t_x$ is negative and therefore invalid (or both are zero which means we have no moves).
  - Therefore (**key point**) as we progress from target to start, there can only be one point to choose from, not two. And this simple fact makes **all the difference** because now, our algorithm can be written as a very simple iteration with no exponential growth!

# Reaching Points #12

- A solution:
  - It solves all my test cases, even the last one, quickly.
  - Here is the code for Leet #780:

```
class Solution {
    public boolean reachingPoints(int sx, int sy, int tx, int ty) {
        while (true) {
            if (tx==sx && ty==sy) return true;
            if (tx<sx || ty<sy) return false;
            if (ty > tx) ty = ty - tx;
            else tx = tx - ty;
        }
    }
}
```

  - However, this does *not* satisfy the Leet challenge. :) It takes too long (only getting to 154 out of 189 tests). Since the tests get steadily harder, I suspect that this solution is still a long way from being correct. Is it possible that it's stuck in an infinite loop, i.e. didn't properly terminate? No. Careful consideration of the possible outcomes rejects that possibility.

# Reaching Points #13

- The final push:
    - Let’s look again graphically (top). Recall that we begin at the target and try to get to the start. Let’s say we begin at 4,5. The route is shown in claret (dark red).
    - Now, let’s try beginning at 2,4 (green). Only one move is possible (as always): to 2,2. This “solution” can be quickly eliminated.
    - Now (bottom) we change *S* to 2,2. The light blue squares are impossible (they missed), the aligned squares (medium blue) are easy to calculate (see below). The dark blue squares need to move into one of the other regions to get a definite answer.
    - In the aligned (medium blue) regions, the move rule tells us that, e.g. if sx==tx, then we are [effectively] home if (ty-sy) % sx == 0. That modulo calculation can save quite a few moves.

| | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 5 | $t_{xxxx}$ | $t_{yxx}$ | $t_{xyx}$ | $t_{yyyx}$ | — |
| 4 | $t_{xxx}$ | — | $t_{yyx}$ | — | $t_{xxxy}$ |
| 3 | $t_{xx}$ | $t_{yx}$ | — | $t_{xxy}$ | $t_{yxy}$ |
| 2 | $t_{x}$ | — | $t_{xy}$ | — | $t_{xyy}$ |
| 1 | S | $t_{y}$ | $t_{yy}$ | $t_{yyy}$ | $t_{yyyy}$ |

| | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 5 | | — | | | |
| 4 | | $t_{x}$ | | | |
| 3 | | — | | | |
| 2 | | S | — | $t_{y}$ | — |
| 1 | | | | | |

## Reaching Points #14

- Success at last!
  - Based on that last bit of intuition, I have solved the LeetCode Problem #780 and beaten 23% of all other solutions (5 milliseconds).
  - Sorry, but I'm not sharing the code with you.
  - Actually, I was able to make a very small improvement to this solution which beats 100% (!!) of other solutions. To get the best solution, you will have to think about it a little bit more.
  - Good luck!

## The Real Answer to Life, the Universe and Everything?

- Everything in this universe is based on the concept of trial and error—random variations and their consequences:
  - Evolution of life itself is driven by heritable *genes*—with some randomness—which make an organism slightly better (or slightly worse) at surviving and reproducing in a particular environment.
  - Human achievement is driven by accumulated *memes*—with some randomness—which improve a process/idea or make it worse. This is the basis of *technology*. Since the invention of the printing press and, especially, the computer, the accumulation of memes has accelerated. Note that science works in a different way: by modeling a system and using the model to make predictions.
  - Your own personal accomplishments are also frequently affected by some randomness—this is especially true when you study and test algorithms.

# Further Reading

- Cracking the Coding Interview
- A Step-by-Step Guide to Passing a Programming Interview
- Avery Lieu's Interview Guide (Python)
- Recursion and Iteration (see Blackboard)
- The Sherlock Holmes Guide to Programming, Debugging and Performance Tuning (from my Software blog).
- The Hitchiker's Guide to the Galaxy.

## How can you get better at coding challenges?

- Practice
- Practice
- Practice