The Hitchhiker's Guide to Interviews*

How to apply in interviews what you've learned here

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

42

The Answer to Life, the Universe, and Everything

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?
  - If it's a problem, start out by thinking the problem through—don't open your mouth for at least 60 seconds, unless you need clarification.
  - Otherwise, make sure you recall the answer to *this* question before saying anything. Don't, for example, mix up different types of sort.
- What is the questioner looking for?
  - They want to know how you *think!*
- Is it a trick question?
  - Almost certainly not.
- 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.

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.
  - 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!
- Stop digging
  - Unfortunately, some times 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!

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. $f(N) = c_0 + c_1 N + c_2 N^2 + \ldots + d_0 \ln N + d_1 N \ln N + d_2 N^2 \ln N + \ldots$
- 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: $d_2 N^2 \ln N$

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 problem domain, i.e. A*.
  - If, for example, we can divide the problem 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, 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:
  - 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.

General Principles

- Entropy—the degree of disorder of a collection:
  - 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.
- Recursion—a fundamental approach to solving problems:
  - In many tree or graph algorithms, recursion is the natural way to code. Don't be afraid to use it.
  - Code looks better with recursion rather than iteration. But it uses extra memory (on the "stack") so, if you *can* do it with equivalent elegance, use iteration. Example: binary search.

Coding Challenges

- Write code that is simple, obvious and elegant (SOE).
  - It’s extremely unlikely that the code they want you to write requires any special cases (corner cases) or anything like that.
  - The more elegant your code a) the more likely it will work; b) the more likely that if it doesn’t work, you’ll be able to see the problem; and c) 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 succeeded.
  - 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?

Coding Challenges (3)

- Checklist for debugging:
  1. For loops: are the starting value and the “while” condition valid? Could they be off by one?
  2. While/until/do loops: is the “while/until” condition valid? Could it be off by one?
  3. Recursion: is the terminating condition arising at the correct point? Could it be off by one?
  4. When comparing or assigning array values: are the indexes correct? Could one of them be off by one?
  5. When using *compareTo* or *less*: do you have the operands (*this* and *other*) in the correct order?
  6. 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?

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 *N* 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.
  - Scala (and other FP languages) will handle this properly. Unfortunately, Java doesn't
  - However, you can transform your tail-recursive code into a while loop.

Case Study: Reaching Points

- LeetCode problem #780 (Reaching Points)
  - Looks simple but is not (it is marked “hard”).
  - In a two-dimensional integer grid ($0 < x, y < 1000000000$), 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*:
    - $q_1 = (p_x+p_y, p_y)$ and $q_2 = (p_x, p_x+p_y)$
  - Start out by, essentially, rewriting the problem in (pseudo-)code
    - `public boolean valid (Point p) = p.equals(t) || p.valid() && (valid(q1) || valid(q2))`

Reaching Points #1

- Let’s look at the problem graphically:
  - 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*.
- 1 step: 2 squares; 2 steps: 4 squares; 3 steps: 8 squares; 4 steps: 4 (visible) squares.
- 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 #2

- 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 likely blow the stack.

Reaching Points #3

- 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 #4

- So, our algorithm will look something like this:

```
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 #5

- 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?
  - 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 #5A

- 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 work, you have to have a high frequency of duplicates. It 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 we are losing faith).

Reaching Points #6

- 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.
- You have exhausted all possibilities for improving a strategy where you begin at the start and work your way to the target.
- What if you were to begin at the target and work your way to the start?

Reaching Points #7

- A new approach:
  - 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.
  - **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) (but not both).
  - 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 #8

- My (current) “final” version:
  - 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, that 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 #9

- 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					

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

- The Sherlock Holmes Guide to Programming, Debugging and Performance Tuning.

- The Hitchiker's Guide to the Galaxy.