Technical Interview Workshop Resources

Kenny Yu

November 4, 2012

DISCLAIMER. This document is simply a guide on how to prepare for interviews. Your interview preparation should NOT be limited to this list. Furthermore, each company has its own distinct recruiting process and interview style which may or may not be reflected in this document.

1 How to Prepare for the Interview

1. **Practice coding on the whiteboard.** Coding on the whiteboard without a text editor and a compiler/interpreter is a very different experience than coding on your computer. Practice penmanship, using whiteboard space effectively, learning to diagram components of your code, and presenting your work on the whiteboard.

2. **Know your preferred language very well.** Whatever language you use for your interviews, know the ins and outs of the language.

   Is the language strongly/weakly typed? Compiled/interpreted? Mutable/immutable data structures? How does memory allocation work? Is it garbage collected? What are the properties of your language and its impact on performance (time and space)?

   Know the idioms of your language, e.g. understand code design in your language and when and why to use a class, abstract class, interface, function, method, module, etc. Should I use a loop, or use functional aspects like map and reduce? How should I iterate over collections? Know the standard variable naming conventions.

   Know the standard libraries of your language (e.g. for Java, know how to use the `java.util.Collections` package; for C++, know how to use the STL collections; for Python, know how to use the built-in lists, dictionaries, and other data structures). How are these collections implemented? You should be comfortable using the standard libraries of your language.

3. **Review your algorithms and data structures.** Many questions may require creative use of one or multiple of the basic data structures. Also, many questions may also require creative use of basic algorithms (binary search, DFS, BFS, etc).

4. **Review the material that you have learned from your classes.** In addition to solving the questions, it is just as important to show that you have deep understanding of the topics you have learned in your classes.

5. **The interview is typically a conversation between you and the interviewer.** Pay attention to cues that your interviewer may give you. Not only is the interview your opportunity to showcase your problem solving abilities, but it is also an opportunity for you to learn more about the company. Go to your interview prepared with questions about the company–what do you want to learn about the company?

2 General Tips During the Interview

1. **Always be thinking out loud**. The point of the interview is to gauge how you think through a problem. If you are silent, the interviewer will learn nothing about your thinking process.

2. Repeat the question back in your own words.

3. Make sure you understand the problem by working through a few small and simple test cases. This will give you time to think and get some intuition on the problem. Your test cases should cover all normal and boundary cases (null, negatives, fractions, zero, empty, etc.).

4. Write down the function header/interface/class definition first and validate it with your interviewer to make sure you understand the problem.

5. Quoted from a friend: "If you are ever stuck, don't be frustrated. Being visibly frustrated shows a sign of weakness and an inability to work through tough problems. The questions wont be answered from the first go. Approach it as a research problem, take examples and work it out till you see a pattern. Be patient, and try out multiple hypothesis to show your creativity. Be able to realize when a path isn't working and backtrack and try something different."

6. **Don't try to come up with the most efficient algorithm from the first go.** Propose the simplest (slow, but correct) algorithm you can think of and then start thinking of better solutions. This could mean brute forcing the problem (trying all cases). Point out the inefficiencies of this solution (e.g. time and/or space complexity). This will also give you a starting point from which to find a more efficient solution.

7. Quoted from a friend: "After you have thought of a solution and described it verbally, start thinking about code design. Do you need a helper class to represent some data (e.g. Points, Pair, Dates,)? Does this class need any methods? Do you need any other helper methods? This step is your transition between algorithm and code, take the time to design your classes (if any) and functions properly."

8. When you are done writing your code, validate your code on your test cases.

9. Quoted from a friend: "If you find a mistake don't be frustrated. Its not typical to get it all right the first (or second) time, just go back and update your algorithm/code!"

3 Approaching Problems

1. **Reduction.** Does this problem look familiar? Can I reduce this problem to a problem I have already solved?

2. **Data Structures.** What data structures do I need? Stacks, queues, heaps, priority queues, linked lists, arrays, sets, maps, binary search trees, graphs, etc.?

3. **Algorithms.** What algorithms do I know might be helpful? Binary search, depth-first-search, breadth-first-search, shortest path, minimum spanning tree, max flow, etc.

4. **Greedy.** Does choosing the *best* choice at every step lead to the overall optimal solution?

5. **Divide & Conquer.** Can I divide this problem into smaller subproblems and then merge their solutions back together? Think merge sort.

6. **Dynamic Programming.** Does the solution to the current problem rely on solutions to smaller subproblems? What is my recurrence?

7. **General Problem Solving.** What patterns am I seeing? Do these patterns continue for larger inputs? What is the behavior for large inputs? Small inputs? In the long run? Can I solve a simplified version of the problem first? Can I use randomness? Can I achieve an approximation to my answer?

4 Resources for Further Reference

4.1 Books

1. **Cracking the Coding Interview** by Gayle Laakmann McDowell.

2. **Algorithm Design** by Jon Kleinberg & Eva Tardos.

   This book is great at teaching you the algorithms and techniques if you are learning it for the first time. Has a great collection of exercise problems.

3. **Introduction to Algorithms** by Cormen, Leiserson, Rivest, Stein (CLRS).

   This book is a bit dense and can be difficult to follow if you are learning the material for the first time. I recommend using the Tardos book to learn the material, and using this book as a reference and for more problems.

4. **Design Patterns** by Gamma, Helm, Johnson, Vlissides (Gang of Four).

    Great reference book for commonly used design patterns for problems that occur over and over again in software engineering.

4.2   Online Resources

1. **Hacking a Google Interview**,
   `http://courses.csail.mit.edu/iap/interview/materials.php`

2. **Top Coder Algorithm Tutorials**,
   `http://www.topcoder.com/tc?d1=tutorials&d2=alg_index&module=Static`

3. **Glassdoor Interviews**,
   `http://www.glassdoor.com/Interview/engineering-interview-questions-SRCH_KO0,11.htm`

4. **Career Cup**, authors of Cracking the Coding Interview
   `http://www.careercup.com/`

5. **Interview Street**, `https://www.interviewstreet.com/challenges/`

6. `http://poj.org/`

7. **Project Euler**, `http://projecteuler.net/`

8. **Code forces**, `http://codeforces.com`

9. **Searching online.**