Coding Interview Workshop

Annie Gao (BR ‘21)
General Guidelines

● Ask clarifying questions!
  ○ Make sure you understand the problem completely before beginning to discuss the solution
  ○ Many interview questions are slightly vague on purpose
  ○ Check input constraints, edge cases
    ■ negative/floating point numbers, empty lists/strings, etc.

● Explain thought processes and decision making throughout the interview
  ○ Think out loud - engineers want to see how you break down a potentially unfamiliar problem
  ○ Comment your code (especially on technical assessments with no moderator)
General Guidelines

- Listen to the interviewer’s feedback
  - Verbally explain your approach and get confirmation before beginning to code
  - Pay attention to questions the interviewer asks back - usually they lead you to a bug or optimization in your code
    - Don’t be nervous if this happens - it’s more important that you realize and fix it

- Optimizing code
  - Again, think aloud. Discuss options and their tradeoffs, and don’t be scared to start from the brute force solution (sometimes that’s actually the solution)
  - Consider runtime and space complexities, reusability, readability
General Guidelines

- **Language choice**
  - Pick (or learn) a language where you know how to call different library functions/data structures
    - Java, Python, C++ are very popular and are also widely used in industry
  - Know the syntax of your language, and maybe even some language features that impact code performance
    - Is length of an array retrieved in $O(1)$ or $O(n)$ time?
    - Pass function arguments by value or by reference?
    - Memory management?
  - It’s okay if you blank on the specific function name. Ask your interviewer if you could quickly look this up during the interview, or if it’s okay to just name the function a similar name that still conveys what it does (i.e. remove() vs delete())
General Guidelines

● Test your code!
  ○ After implementing a solution, run it on a sample input (out loud) for a sanity check
    ■ If you find a bug, fix it before the interviewer has a chance to speak :)
  ○ After basic test passes, think about edge cases
    ■ Come up with a list of test inputs and their expected outputs, and make sure your code produces the right output
  ○ Only tell your interviewer that this is your final solution after running them through your tests
    ■ At this point, they will ask you about runtime and space complexity (again, think aloud)

● Ending questions
  ○ Ask questions about the company and/or the interviewer’s work/company life
  ○ Thank the interviewer for their time
  ○ Take a deep breath - you made it! :)
Technical Preparation

- **Data Structures:**
  - Arrays, linked lists, stacks/queues, hash tables, priority queues, trees/graphs
  - Know the runtime and space complexity of operations on these structures to discuss tradeoffs
  - To understand them more, try implementing them yourself from scratch (*not* during the interview though; use built-in data structures for that)

- **Algorithms:**
  - Big O analysis for runtime/space complexity
  - Sorting, searching, hashing, recursion, divide and conquer, dynamic programming

- **Coding:**
  - Practice with friends on paper/whiteboard, mock interview each other to familiarize yourself with the interview setting
  - Pay attention to neatness and syntax (you need to show that you actually know how to code)
Technical Preparation: Resources

- Cracking the Coding Interview book (the holy grail)
- LeetCode
- Project Euler
- Google Code Jam questions
- ACM-ICPC Live Archive
- UVa Online Judge
- UVA Toolkit
- Interview Cake
- CodingBat
- Coderbyte
- Programming Challenges (text with solutions)
- Topcoder (launch the "Arena" widget and then go to the practice rooms)
Technical Preparation: Practice Question 1

Given a string containing just the characters '(', ')', '{', '}', '[' and ']', determine if the input string is valid.

An input string is valid if:

1. Open brackets must be closed by the same type of brackets.
2. Open brackets must be closed in the correct order.

Example 1:
Input: "()"
Output: true

Example 2:
Input: "()[{}]"
Output: true

Example 3:
Input: "(()"
Output: false

Example 4:
Input: "([)]"
Output: false

Example 5:
Input: "{[]}"
Output: true

Clarifying Questions?

Approaches to Problem?

Implementation?
Practice Question 1: Algorithm

Algorithm

1. Initialize a stack S.
2. Process each bracket of the expression one at a time.
3. If we encounter an opening bracket, we simply push it onto the stack. This means we will process it later, let us simply move onto the sub-expression ahead.
4. If we encounter a closing bracket, then we check the element on top of the stack. If the element at the top of the stack is an opening bracket of the same type, then we pop it off the stack and continue processing. Else, this implies an invalid expression.
5. In the end, if we are left with a stack still having elements, then this implies an invalid expression.

What is the runtime and space complexity?
Practice Question 1: Implementation

```java
public boolean isValid(String s) {
    Stack<Character> stack = new Stack<>();
    HashMap<Character, Character> map = new HashMap<>();
    map.put(')', '(');
    map.put(']', '[');
    map.put('}', '{');
    for (int i = 0; i < s.length(); i++) {
        char c = s.charAt(i);
        if (c == '(' || c == '[' || c == '{') {
            stack.push(c);
        } else { // closing bracket
            if (stack.isEmpty()) { // illegal closing bracket
                return false;
            }
            char opening = stack.pop(); // illegal character
            if (map.get(opening) != c) { // wrong matching
                return false;
            }
        }
    }
    // If the stack still contains elements, then invalid
    return stack.isEmpty();
}```
Technical Preparation: Practice Question 2

Implement $\text{pow}(x, n)$, which calculates $x$ raised to the power $n$ ($x^n$).

Note:
- $-100.0 < x < 100.0$
- $n$ is a 32-bit signed integer, within the range $[-2^{31}, 2^{31} - 1]$

Example 1:
Input: 2.00000, 10
Output: 1024.00000

Example 2:
Input: 2.10000, 3
Output: 9.26100

Example 3:
Input: 2.00000, -2
Output: 0.25000

Questions, Approaches, and Implementations? Work with a friend! :)
Problem 2 Approach 1: Brute Force

Algorithm

We can use a straightforward loop to compute the result.

Complexity?

Optimizations / tradeoffs?

```java
public double myPow(double x, int n) {
    long N = n;
    if (N < 0) {
        x = 1 / x;
        N = -N;
    }
    double ans = 1;
    for (long i = 0; i < N; i++)
        ans = ans * x;
    return ans;
}
```
Problem 2: Recursive Solution Algorithm

Algorithm

Assume we have got the result of $x^{n/2}$, and now we want to get the result of $x^n$. Let $A$ be result of $x^{n/2}$, we can talk about $x^n$ based on the parity of $n$ respectively. If $n$ is even, we can use the formula $(x^n)^2 = x^{2n}$ to get $x^n = A \times A$. If $n$ is odd, then $A \times A = x^{n-1}$. Intuitively, We need to multiply another $x$ to the result, so $x^n = A \times A \times x$. This approach can be easily implemented using recursion. We call this method "Fast Power", because we only need at most $O(\log n)$ computations to get $x^n$.

Complexity?
Problem 2: Recursive Solution Implementation

```java
private double fastPow(double x, long n) {
    if (n == 0) {
        return 1.0;
    }
    double half = fastPow(x, n / 2);
    if (n % 2 == 0) {
        return half * half;
    } else {
        return half * half * x;
    }
}

public double myPow(double x, int n) {
    long N = n;
    if (N < 0) {
        x = 1 / x;
        N = -N;
    }
    return fastPow(x, N);
}
```

Optimizations / tradeoffs?
Problem 2: Iterative Solution Algorithm

Algorithm

We can use the binary representation of $n$ to better understand the problem. Let the binary representation of $n$ to be $b_1, b_2, ..., b_{\text{length\_limit}}$, from the Least Significant Bit (LSB) to the Most Significant Bit (MSB). For the $i$th bit, if $b_i = 1$, it means we need to multiply the result by $x^{2^i}$.

It seems to have no improvement with this representation, since $\sum_i b_i * 2^i = n$. But using the formula $(x^n)^2 = x^{2^n}$ we mentioned above, we can see some differences. Initially $x^1 = x$, and for each $i > 1$, we can use the result of $x^{2^{i-1}}$ to get $x^{2^i}$ in one step. Since the number of $b_i$ is at most $O(\log n)$, we can get all $x^{2^i}$ in $O(\log n)$ time. After that, for all $i$'s that satisfy $b_i = 1$, we can multiply $x^{2^i}$ to the result. This also requires $O(\log n)$ time.
Problem 2: Iterative Solution Implementation

```java
public double myPow(double x, int n) {
    long N = n;
    if (N < 0) {
        x = 1 / x;
        N = -N;
    }
    double ans = 1;
    double current_product = x;
    for (long i = N; i > 0; i /= 2) {
        if ((i % 2) == 1) {
            ans = ans * current_product;
        }
        current_product = current_product * current_product;
    }
    return ans;
}
```

Complexity?
Questions?
Good Luck! :)