The Silicon Valley coding interview

Nicolo Valigi   nicolovaligi.com

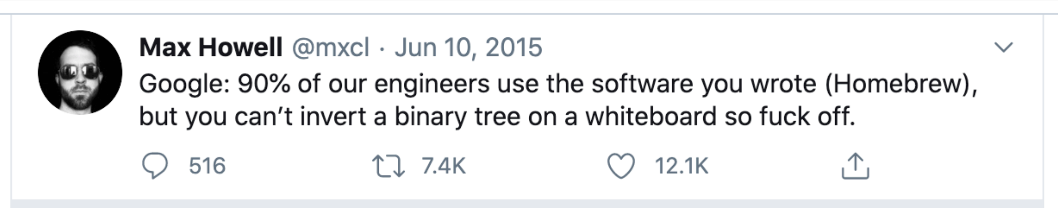
June 13, 2020

Silicon Valley

The Silicon Valley tech ecosystem is huge, and has its own peculiar way of interviewing people for jobs.

Is the coding interview too hard? Or just wrong?

Not everybody is excited about being tested for coding before getting a job..



A common complaint is that the typical coding interview is not representative of the actual day-to-day job.

Goals for the talk

Who is this talk for?

- You're thinking about applying to one of the tech companies
- You want to laugh about how ridiculous people are
- You just like code puzzles and algorithms

This talk will not:

- replace a degree in CS
- land you a job at Google tomorrow
- teach you anything about C++

Plan for the talk

- Recruiting process and timeline
- What interview questions look like
- Families of algorithmic questions
- Tips for preparation

Example question

Some examples of questions I actually got during interviews:

> *Given an array of numbers and an integer* `n`*, find the number of triplets* `(a, b, c)` *such that* `a+b+c` *is less than* `n`*.*

or:

> *Implement a Tic-Tac-Toe game.*

The recruiting process

- You call up a recruiter (or a recruiter calls you)
- Short call with the recruiter (non-technical)
- 1-hour programming test remotely (eg coderpad.io)
- 5 1-hour programming tests on-site
- Feedback and maybe a job offer

The technical interview

What questions to expect

- Most interview questions are about **algorithms** and **data structures**.
- Explained in just a few minutes in very abstract terms.
- Little domain knowledge needed for most of them.

Families of questions

Most questions will fall into one of these categories:

- *linear things*: arrays, (linked) lists, searching, and sorting
- *stacks* and *queues*
- *tree-ish things*: binary search trees, heaps
- *graphs*: search, connected components
- *dynamic programming and recursion*
- *system design*

Array problems

Arrays (1/)

Given `int[] A` *and* `int N`*, find two indices* `i` *and* `j` *in* `A` *such that* `A[i] + A[j] == N`*. Assume that the solution exists and is unique. Each element of* `A` *can only be used once.*

Example:

```
TwoSum({2, 7, 11, 15}, 9) -> {0, 1}
```

Arrays (2/)

A trivial solution uses a nested for loop:

```
for (int i = 0; i < A.size(); i++) {
    for (int j = i+1; j < A.size(); j++) {
        if (A[i] + A[j] == N) {
            return {i, j};
        }
    }
}
```

Works but has poor **asymptotic complexity** (ie, it becomes slow with large arrays).

Intermission: asymptotic complexity

Measures how computational cost (or memory) grows with increasing input size.

- **constant** complexity: same amount of CPU/memory no matter the size of the input. *Example*: get the first element of the array.
- **linear** complexity: grows linearly with the input size. *Example*: sum all the elements of the array.
- **quadratic** complexity
- **exponential** complexity: usually, this means you have to do better..

The interviewer will often ask you what the a.c. of your solution is, and maybe to find one with lower a.c.

Arrays (3/)

- What's the time and space complexity of the trivial solution?
- Can we write a better solution?

Arrays (3/)

- What's the time and space complexity of the trivial solution?
- Can we write a better solution?

We can go from quadratic to linear time complexity if we take advantage of extra memory:

```
std::unordered_map<int, size_t> valueToIndex;
for (int i = 0; i < A.size(); i++) {
    const int rem = N - A[i];
    if (auto it = valueToIndex.find(rem); it != valueToIndex.end()) {
        return {it->second, i};
    }
    valueToIndex[A[i]] = i;
}
```

Stack problems

Stacks (1/)

Stacks are the prototypical *first-in, first-out* datastructure.

> *Different UNIX pathnames can represent the same directory on disk, Write a function that returns the shortest possible equivalent pathname to the input.*

Example:

```
/home/niko//code/../code/./project -> /home/niko/code/project
```

Stacks (2/)

The idea is to `push` each path fragment to a stack, and `pop` on "`..`".

Stacks (2/)

The idea is to push each path fragment to a stack, and pop on "`..`".

```
stack<string> fragments;

stringstream ss(path);
string fragment;

while (getline(ss, fragment, '/')) {
    if (fragment == "." || fragment == "") { }
        continue;
    } else if (fragment == "..") {
    } else if (fragment == "..") {
        if (fragments.empty()
            || fragments.back() == "..") {
            fragments.push(fragment);
        } else {
            fragments.pop();
        }
    } else {
        fragments.push(fragment);
    }
```

Stacks (3/)

And finally we assemble the result back together from the stack:

```
string result;
while (!fragments.empty()) {
    result += fragments.top();
    fragments.pop();
}
return result;
```

- Stacks are a good choice for problems that can be solved incrementally.

Tree problems

Trees (1/)

A (binary) tree is a hierarchical data structure, where each *node* holds references to a left and right children.

```
template <typename T>
struct Node {
    T value;
    Node* left;
    Node* right;
};
```

Many operations on trees are naturally recursive.

Trees (2/)

Determine if a binary tree is symmetric around its center.

This tree is symmetric:

```
    1
   / \
  2   2
 / \ / \
3  4 4  3
```

This tree is *not* symmetric:

```
  1
 / \
2   2
 \   \
 3    3
```

Trees (3/)

```
bool IsSymmetric(Node* tree) {
    return IsMirror(tree->left, tree->right);
}
```

Trees (3/)

```
bool IsSymmetric(Node* tree) {
    return IsMirror(tree->left, tree->right);
}

bool IsMirror(Node* subtree1, Node* subtree2) {
    if (subtree1 == nullptr && subtree2 == nullptr) {
        return true;
```

Trees (3/)

```
bool IsSymmetric(Node* tree) {
    return IsMirror(tree->left, tree->right);
}

bool IsMirror(Node* subtree1, Node* subtree2) {
    if (subtree1 == nullptr && subtree2 == nullptr) {
        return true;

    } else if (subtree1 != nullptr && subtree2 != nullptr) {
        return subtree1->value == subtree2->value &&
```

Trees (3/)

```
bool IsSymmetric(Node* tree) {
    return IsMirror(tree->left, tree->right);
}

bool IsMirror(Node* subtree1, Node* subtree2) {
    if (subtree1 == nullptr && subtree2 == nullptr) {
        return true;

    } else if (subtree1 != nullptr && subtree2 != nullptr) {
        return subtree1->value == subtree2->value &&

            IsMirror(subtree1->left, subtree2->right) &&
            IsMirror(subtree1->right, subtree2->left);
```

Trees (3/)

```
bool IsSymmetric(Node* tree) {
    return IsMirror(tree->left, tree->right);
}

bool IsMirror(Node* subtree1, Node* subtree2) {
    if (subtree1 == nullptr && subtree2 == nullptr) {
        return true;

    } else if (subtree1 != nullptr && subtree2 != nullptr) {
        return subtree1->value == subtree2->value &&

            IsMirror(subtree1->left, subtree2->right) &&
            IsMirror(subtree1->right, subtree2->left);

    } else {
        return false;
    }
}
```

Graph problems

Graph search (1/)

> *You're given two words (`beginWord` and `endWord`) and a list of words `wordList`. Find the length of a sequence of transformations from `beginWord` to `endWord` where only one letter can be changed at a time, and all words must be in `wordList`. Return 0 if no such sequence exists.*

```
findSequence("hit", "cog", {"hot", "dot", "lot", "log", "cog"})
    -> 5
```

Example transformation sequence:

```
"hit" -> "hot" -> "dot" -> "dog" -> "cog"
```

Graph search (2/)

```
queue<string> q;
q.push(beginWord);
int seqLen = 1;
```

Graph search (2/)

```
queue<string> q;
q.push(beginWord);
int seqLen = 1;

while (!q.empty()) {
    const int n = q.size();
    for (int i=0; i < n; i++) {
        const string cur = q.front();
        q.pop();
        dictionary.erase(word);  // Don't use this word again.
        if (cur == endWord) { return seqLen; }  // Done.
        // List neighbors and push them into q...          <=======
    }
    seqLen++;
}
```

Graph search (3/)

Here's how we generate new candidates for exploration. Remember `cur` is the current word we're looking at.

```
for (int j = 0; j < cur.size(); j++) {
    char c = cur[j];
    for (int k = 0; k < 26; k++) {
        string cand = cur;
        // Overwrite the j-th character
        cand[j] = 'a' + k;
        if (dictionary.find(cand) != dictionary.end()) {
            q.push(cand);
        }
    }
}
```

Dynamic Programming problems

Dynamic Programming (1/)

> *You're given a list of coin denominations `values` and an integer `amount`. Find the minimum number of coins needed to add up to that amount. If no combination of `values` can add up to `amount`, return -1.*

Example:

`NumCoinsRequired(11, {1, 2, 5}) -> 3`

made up by `5 + 5 + 1 = 11`

Dynamic Programming (2/)

This problem satisfies the *optimal substructure* property: the optimal solution $X(\text{amount})$ satisfies the following relation:

$$X(\text{amount}) = X(\text{amount} - \text{coin}) + 1$$

where coin is the value of *some* coin, but we don't know which yet. So we try all of them and pick the minimum:

$$X(\text{amount}) = \min_{i=0..n-1} X(\text{amount} - c_i) + 1$$

This allows us to start from small values of amount and reuse lots of computation as we move towards bigger values.

Dynamic Programming (3/)

```
vector<int> dp(amount+1, amount+1);
dp[0] = 0;

for (int i = 1; i <= amount; i++) {
    for (int j = 0; j < coins.size(); j++) {
        if (coins[j] <= i) {
            dp[i] = min(dp[i], dp[i - coins[j]] + 1);
        }
    }
}

return dp[amount] > amount ? -1 : dp[amount];
```

System design problems

System design (1/)

You have to sort more data that fits in the memory of any single machine. What do you suggest to get the job done quickly?

- Open-ended question with several potential solutions and tradeoffs to explore.
- Higher level topics than pure datastructures (eg networking, computer architecture, etc..)

System design (2/)

So, what do we do?

System design (2/)

So, what do we do?

- Spill data to disk (order of magnitudes slower than memory or network).

System design (2/)

So, what do we do?

- Spill data to disk (order of magnitudes slower than memory or network).
- Distribute the data to multiple machines, but how?

System design (2/)

So, what do we do?

- Spill data to disk (order of magnitudes slower than memory or network).
- Distribute the data to multiple machines, but how?
- At first glance, *merge sort* might be a good algorithm (just merge the data to/from multiple machines at the same time).

System design (2/)

So, what do we do?

- Spill data to disk (order of magnitudes slower than memory or network).
- Distribute the data to multiple machines, but how?
- At first glance, *merge sort* might be a good algorithm (just merge the data to/from multiple machines at the same time).
- If the data is segmented by a natural key, use *bucketing* techniques to spread the load across multiple machines.

Wrapping up

Recap of problems

- *arrays*: usually the goal is to minimize time/space complexity.
- *stacks*: applies to problems with an incremental/sequential structure.
- *trees*: the trick is to identify a common property across levels and write it down recursively.
- *graphs*: the trick is to write down a problem in terms of edges, nodes, and connectivity between the two.
- *Dynamic Programming*: the trick is to find a repeating structure of smaller problems.
- *System Design*: good opportunity to show off your real-world experience.

Things we left out:

- (Linked) lists and queues.
- Backtracking with recursion.
- Searching and sorting.

Wrapping up

- Algorithms are fun. Not so much when you're under pressure in front of a whiteboard.
- The interviewer wants you to do well (so they'll have one more team member to help out).
- Start with simple solutions, then iterate to improve the time/space complexity.
- Some interviewers give lots of tips if you're stuck, some are dead silent. Asking helps.
- Practice is everything.

References

- *Aziz, Lee, Prakash*, Elements of Programming Interviews: The Insiders' Guide
- EPI judge github.com
- leetcode.com
- *Steven S. Skiena*, The Algorithm Design Manual

Thanks