Behdad Esfahbod Follow
Apr 12 · 6 min read

On a great interview question

Between 2010 and 2019 I interviewed dozens of Software Engineer candidates at Google. Almost always I asked the same interview question. Moreover, this question happened to be on the banned list at Google, because it was publicly available on Glassdoor and other interview websites, but I continued to use it because I got good signal from the candidates.

Daniel Tunkelang had a similar positive experience with this problem but retired it back in 2011: "Retiring a Great Interview Problem". That is an excellent read by itself, but since my experience with this problem has been more extensive, I decided to write it down.

Why this is a good interview problem

The problem starts simple. *Really* simple, to the extent that some candidates think they have not understood it correctly. From there it explores the candidates understanding of various basic algorithms and data structure concepts.

Moreover, the problem as I present it, has enough range to sift out bad candidates from good ones even if the candidates have seen the problem before, which is why I successfully used it even though it was banned at Google. Twice I asked the hiring committee if I should stop using this and they had no problem with it.

The interview

This is a 45-minute technical interview for Software Engineering positions, from junior to mid-level candidates. Maybe surprisingly, junior candidates sometimes rank better at this problem because their Data Structure & Algorithms knowledge is more fresh. But this pattern is not universal. Good senior candidates have no problem with the interview at all.

A few anecdotes that have stayed with me:

- I once had a candidate with 15-years work experience give me lots of attitude for asking them such a simple question, while at the same time struggling at it.
- The fastest I had a candidate solve the entirety of the problem with all bells and whistles was in 20 minutes, by a *freshman* from the University of Waterloo, the type who did high-school competitions.
- The most depressing failure I saw was a PhD graduate from University of Toronto who could not produce working code for the first section of the problem in 45 minutes.
- A candidate upfront told me that they have seen the problem. I grinned and said "great, so you should know how to solve it!" and they still struggled with it.
- Another candidate had clearly memorized the solution code and jumped to writing it. I asked them to step back and go through the steps I asked them to. They also struggled. They were also dishonest about having seen it before.

I start by telling the candidate that I like to hear their thought process, and I do *not* want them to write code until I ask them to do so. I then tell them that there are two parts to the problem.

Part 1

The problem

"Write a function that given a string, determines if it's a concatenation of two dictionary words."

When questioned, I explain that the dictionary is available as a function to query a string for membership.

The solution

Any good candidate should be able to quickly work out the brute-force / trivial solution to the problem, which is: split the input string at all possible locations, and query both sides in the dictionary.

You would be surprised how many candidates struggle to arrive at this solution though. Some come up with a *greedy* algorithm which finds the first prefix that is in the dictionary, and then returns whether the rest of the string is in the dictionary. Some might deem the brute-force solution too simple / *inefficient* and seek a faster solution right off the bat. I try to steer them to suggest *any* solution first.

At this point I ask them to write code for it. Here's a one-liner solution code in Python, using generators:

```
def splitString2(s):
  return any(inDict(s[:i]) and inDict(s[i:]) for i in range(1, len(s)))
```

Most candidates though write a for loop, which is equally acceptable.

Running-time

I then ask the candidate to analyze the running time of this solution.

Most candidates suggest O(n) as the answer. (Let's assume that making substrings s[:i] and s[i:] is constant-time.) I will ask them how would they implement the dictionary function such that this solution is O(n). They mostly suggest that if implemented as a hash function, it would be O(1) lookup. I then push them on this until they realize that any implementation of the dictionary lookup function of a string would be o(n) at best, and as such the total runtime of the solution is $O(n^2)$.

Faster solution

If the candidate was fast to get this far (ie. less than 20 minutes had passed), I then ask the candidate if they can think of a way to speed up on this solution. Most candidates would struggle a bit. I help by telling them that if dictionary lookup is using a function as is, then this algorithm is asymptotically optimal (I don't ask them to prove, but can you prove this?). So any speedup would require a different representation of the dictionary.

Very good candidates realize that the dictionary lookup of prefixes is doing a lot of repetitive work walking the strings, which can be avoided by representing the dictionary in form of a tree, to be more exact, a *trie*. I had a couple of candidates reinvent tries on the spot, while others vaguely remembered them. At any rate, representing the dictionary as a trie would make checking all prefixes in the dictionary O(n) total. We still have the problem of checking all the suffixes. By way of symmetry, we can also keep another trie that contains words in the dictionary *reversed*. At that point, checking all suffixes for dictionary membership would also be O(n). We just need an array of o(n) size to save this. Then we just need to intersect the prefix and suffix results and find an overlap. The total algorithm is O(n), and hence optimal.

I do *not* ask candidates to write code for this. If they get stuck for too long, I skip to part 2.

Part 2

The problem

"Write a function that given a string, determines if it's a concatenation of a number of dictionary words."

Again, we start with the dictionary being represented as a query function. When asked whether a string that is in the dictionary by itself should pass the test, the answer is yes.

The solution

I emphasize again that I like to see *any* solution first. Good candidates recognize that they can simply modify their solution from part one into a recursive one. For example:

```
def splitString(s):
  if inDict(s):
    return True
  return any(inDict(s[:i]) and splitString(s[i:]) for i in range(1, len(s)))
```

Many candidates struggle with the recursive solution though.

Running time

Analyzing the running-time of the recursive solution can be a challenge for some. Again, assuming that the string splitting operations are O(1), it comes down to recognizing that the string is, at worst case, split into all possible splits, and there are o(2^n) of them, which combined with the dictionary lookup cost, results in O(n.2^n) runtime.

Faster solution

I then ask for a faster solution, and this time I expect the candidate to arrive at one. Good candidates observe that the splitString function is being called recursively again and again for the same input, and as such arrive at memoization or dynamic-programming as a remedy.

I do *not* ask them to write code for this, and just jump to the runtime analysis.

Running time

The key to correct analysis here is to recognize that we ever only recurse on the splitString function with suffixes of the original string, which there are n of. For each suffix, we perform O(n) operations, which considering the dictionary lookup cost become $O(n^2)$. The total runtime as such is $O(n^3)$.

I do *not* ask candidates to show this, but see if you can prove this to be optimal assuming that the dictionary is provided as a lookup function.

Even faster solution

With most candidates we have run out of time at this point, but if there is time, I ask the candidate if they can come up with a faster solution. If they have arrived at the trie solution to part 1, they will suggest the trie approach here, which would make for a combined $O(n^2)$ runtime.

See if you can prove this to be optimal.

Update: Assume that the number of words in the dictionary is much larger than n.

Software Engineering Interview Questions Dynamic Programming Google Glassdoor