General Coding Skills Evaluation Framework

Technical Brief

CodeSignal

Skills Evaluation Lab

Published July 2019; Updated April 2023

Introduction

In today's business landscape, digital transformation, especially in the context of software development, has become a primary source of competitive advantage for organizations. This is reflected in the widely-held belief that "every company is now a software company" [1]. To thrive in this environment, organizations must find ways to identify top technical talent better and faster than the competition. Accordingly, demand for technical talent, particularly software developers or engineers, has grown rapidly. In fact, the demand for software engineer jobs is projected to grow at an accelerated rate of more than 25% between 2021 to 2031 [2].

Unfortunately, current hiring processes tend to be inefficient and ineffective, either by requiring too much time for senior engineers to manually vet candidates through time-intensive interviews or through the use of inefficient evaluations that do not accurately or consistently capture candidates' skills. As a result, organizations are becoming increasingly concerned with the structure, consistency, and scalability of their hiring processes for software engineers. Such concerns call for a standardized approach to creating automated evaluations, which will allow organizations to evaluate the skills of software engineering candidates with a high degree of accuracy, consistency, and fairness while enabling them to scale to meet the growing demand for technical talent.

This paper describes a framework for developing simulation-based evaluations that accurately capture high-quality signals of the technical skills held by candidates applying to software engineering jobs at scale. Framework-based evaluations are expertly designed and highly structured, allowing engineering and talent teams to efficiently scale their hiring process and make effective hiring decisions while providing a fair and engaging experience for candidates.

The General Coding Skills Evaluation Framework described in this paper can be used to create Certified Evaluations to measure core coding skills. This Framework was developed based on researching software engineering jobs and consultation with subject matter experts. Certified Evaluations powered by this framework are designed to assess the key knowledge and skills that are: 1) generally taught in computer science programs (including coding boot camps) and 2) commonly required for software engineering roles across a wide variety of organizations and industries.

Framework Specifications

The purpose of this framework is to provide a blueprint for developing valid and reliable evaluations of candidates' role-relevant skills for software engineering and related roles at scale. The framework can be utilized to create evaluations that span across different delivery methods, such as pre-screen assessments or technical interviews, while providing objective signals by generating scores to quantify candidates' skills.

Evaluations based on this framework consist of four modules, with one question each, that require candidates to write code based on specified requirements. Each module has a slightly different focus, but all modules are designed to capture one or more of these core coding skills:

1. Basic Coding
2. Data Manipulation
3. Implementation Efficiency
4. Problem Solving

Candidates are given an opportunity to demonstrate their skills by effectively solving questions within the modules. For example, solving Basic Coding questions demonstrates skill in writing basic code to conduct basic operations such as working with numbers, strings, and arrays. Similarly, solving Problem Solving questions demonstrates understanding of challenging computing problems and knowledge of algorithms that can efficiently solve those problems (e.g., greedy, two pointers, etc.).

To balance the breadth and depth of the evaluation content with the goals of fostering a positive candidate experience, **the maximum allowed time for this framework is 70 minutes (for 4 code writing questions)**. Longer evaluations allow for increased measurement precision and improve the quality of signal–however, the more time-intensive evaluations become, the more reluctant candidates are to complete them. Moreover, solving the questions in the given timeframe is an important indicator of skill and a key factor in differentiating between candidate skill levels. This time-constrained process simulates on-the-job demands, as software engineers often balance multiple tasks simultaneously. Additionally, offering a limited, 70-minute timeframe helps prevent candidates from engaging in behaviors such as spending time searching for answers online, further promoting the validity of evaluations powered by the framework.

The following sections outline specifications for each module within the General Coding Framework at a high level. These specifications can be used to create variations of questions while ensuring evaluation results are comparable across candidates.

Module 1 – Basic Coding

This module contains **one coding question** focusing on basic coding concepts and operations. On average, candidates are expected to write **5-10 lines of code** and solve this within **10 minutes**.

Expected Knowledge

- Basic operations with numbers
- Basic string manipulation, such as splitting a string into substrings or modifying the elements of a string
- Basic array manipulation, such as iterating over an array

Can Include

- Questions that require a combination of 2 to 3 basic concepts, such as conditionally iterating over an array, or conditionally splitting a string
- Questions that should generally be solvable using a single loop
- Clear descriptions of implementation with step-by-step instructions

Should Exclude

- Questions that require noticing or proving patterns
- Questions that require knowledge of basic algorithms or optimization
- Questions that require designing or figuring out implementation details

Module 2 – Data Manipulation

This module contains **one coding question** focusing on manipulating data structures. On average, candidates are expected to write **10-20 lines of code** and solve this within **15 minutes**.

Expected Knowledge

- Working with numbers, including
    - Basic operations with numbers
    - Splitting numbers into digits
- Basic string manipulation
    - Splitting a string into substrings
    - Comparing strings
- Modifying elements of a string
    - Concatenating strings
    - Reversing strings
- Basic array manipulation
    - Iterating over an array
    - Modifying the elements of an array
    - Reversing an array
    - Concatenating two arrays

Can Include

- Questions that require a combination of 3 to 5 basic concepts, for example:
    - Splitting a string into substrings, then modifying each substring and comparing each with other substrings
    - Iterating over an array to split into two arrays, then modifying the second array and appending it to the first array
- Questions that should generally be solvable using 1 to 2 nested loops
- Clear descriptions of implementation with step-by-step instructions

Should Exclude

- Questions that require noticing or proving patterns
- Questions that require knowledge of basic algorithms or optimization

Module 3 – Implementation Efficiency

This module contains **one coding question** focusing on implementing solutions that can run efficiently and adheres to execution time limits. On average, candidates are expected to write **25-40 lines of code** and solve this within **20 minutes**.

Expected Knowledge

- Includes everything from module 1 and module 2
- Splitting overall requirements into subtasks or functions
- Manipulating multidimensional arrays, for example:
    - Iterating over elements within nested arrays in a given order
    - Transposing or pivoting the rows and columns values in a 2D array
- Using built in hashmaps to store strings or integers as keys

Can Include

- Implementing a specific comparator for strings
- Implementing a specific merge function for arrays
- Other implementation challenges which require translating step-by-step instructions into code

Should Exclude

- Questions that require noticing or proving patterns
- Questions that require algorithms with advanced data structures, such as binary indexed trees
- Questions that require complex topics, such as graphs, number theory, or dynamic programming

Module 4 – Problem Solving

This module contains **one coding question** focusing on applying algorithmic techniques to implement optimal solutions. On average, candidates are expected to write **25-35 lines of code** and solve this within **30 minutes**.

Expected Knowledge

- Includes everything from module 1, module 2, and module 3
- Implementing common algorithms to optimize solutions, such as greedy, divide and conquer, and two pointers
- Implementing abstract data types such as hashmaps within solutions
- Discrete mathematics fundamentals

Can Include

- Questions that require implementing an appropriate algorithm, data structure, or technique
- Questions that require optimizing queries using data structures like hashmaps or sets

Should Exclude

- Questions designed like brain teasers
- Questions that require knowledge of specialized or advanced algorithms, such as Dijkstra, Kruskal, or Fast Fourier transform (FFT)
- Questions with complicated or time-consuming implementation steps that would be difficult to optimize

Framework Example Content

Below are example questions for each module within the framework. Similar questions are developed in accordance with framework specifications on an ongoing basis to minimize the impact of leaks that could result in cheating or plagiarism, as well as provide relevant and fair candidate experiences through changing industry standards.

Module 1 – Basic Coding

Given an array a, your task is to output an array b of the same length by applying the following transformation:

- For each `i` from 0 to `a.length - 1 inclusive`, `b[i] = a[i - 1] + a[i] + a[i + 1]`
- If an element in the `sum a[i - 1] + a[i] + a[i + 1]` does not exist, use `0` in its place
- For instance, `b[0] = 0 + a[0] + a[1]`

Example

For `a = [4, 0, 1, -2, 3]`:

```
•  b[0] = 0 + a[0] + a[1] = 0 + 4 + 0 = 4
•  b[1] = a[0] + a[1] + a[2] = 4 + 0 + 1 = 5
•  b[2] = a[1] + a[2] + a[3] = 0 + 1 + (-2) = -1
•  b[3] = a[2] + a[3] + a[4] = 1 + (-2) + 3 = 2
•  b[4] = a[3] + a[4] + 0 = (-2) + 3 + 0 = 1
```

So, the output should be `solution(a) = [4, 5, -1, 2, 1].`

Sample Solution (python)

```python
def solution(a):
    n = len(a)
    b = [0 for _ in range(n)]
    for i in range(n):
        b[i] = a[i]
        if i > 0:
            b[i] += a[i - 1]
        if i < n - 1:
            b[i] += a[i + 1]
    return b
```

Module 2 – Data Manipulation

You are given two strings: `pattern` and `source`. The first string pattern contains only the symbols `0` and `1`, and the second string source contains only lowercase English letters.

Your task is to calculate the number of substrings of source that match pattern.

We'll say that a substring `source[l..r]` matches pattern if the following three conditions are met:

- The pattern and substring are equal in length.
- Where there is a `0` in the pattern, there is a vowel in the substring.
- Where there is a `1` in the pattern, there is a consonant in the substring.

Vowels are `["a", "e", "i", "o", "u", "y"]`. All other letters are consonants.

Example
For `pattern = "010"` and `source = "amazing"`, the output should be `solution(pattern, source) = 2`.

- `"010"` matches `source[0..2] = "ama"`. The pattern specifies "vowel, consonant, vowel". `"ama"` matches this pattern: `0` matches `a`, `1` matches `m`, and `0` matches `a`.
- `"010"` doesn't match `source[1..3] = "maz"`
- `"010"` matches `source[2..4] = "azi"`
- `"010"` doesn't match `source[3..5] = "zin"`
- `"010"` doesn't match `source[4..6] = "ing"`

So, there are `2` matches.

For `pattern = "100"` and `source = "codesignal"`, the output should be `solution(pattern, source) = 0`.

- There are no double vowels in the string `"codesignal"`, so it's not possible for any of its substrings to match this pattern.

Guaranteed constraints:

- `1 ≤ source.length ≤ 103`
- `1 ≤ pattern.length ≤ 103`

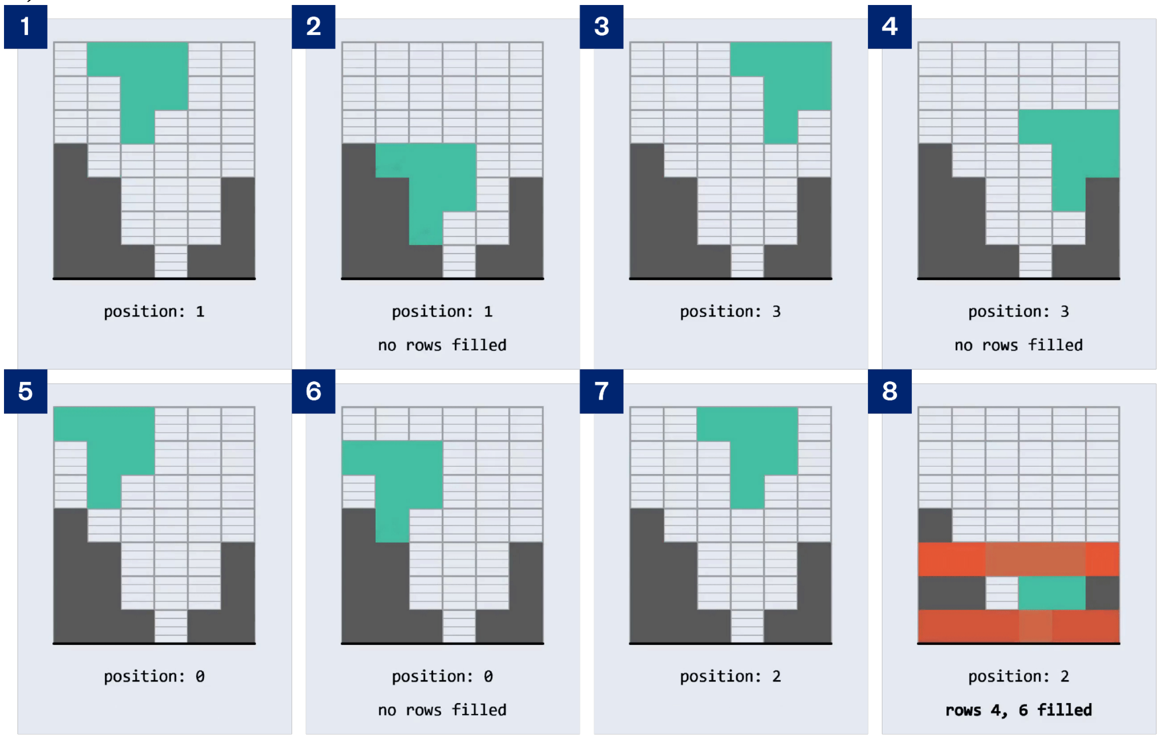
Example Solution (python)

```python
vowels = ['a', 'e', 'i', 'o', 'u', 'y']

def check_for_pattern(pattern, source, start_index):
   for offset in range(len(pattern)):
       if pattern[offset] == '0':
           if source[start_index + offset] not in vowels:
               return 0
       else:
           if source[start_index + offset] in vowels:
               return 0
   return 1
  def solution(pattern, source):
   answer = 0
   for start_index in range(len(source) - len(pattern) + 1):
       answer += check_for_pattern(pattern, source, start_index)
   return answer
```

Module 3 – Implementation Efficiency

You are given a matrix of integers field of size height × width representing a game field, and also a matrix of integers figure of size 3 × 3 representing a figure. Both matrices contain only 0s and 1s, where 1 means that the cell is occupied, and 0 means that the cell is free.



[1]

You choose a position at the top of the game field where you put the figure and then drop it down. The figure falls down until it either reaches the ground (bottom of the field) or lands on an occupied cell, which blocks it from falling further. After the figure has stopped falling, some of the rows in the field may become fully occupied.

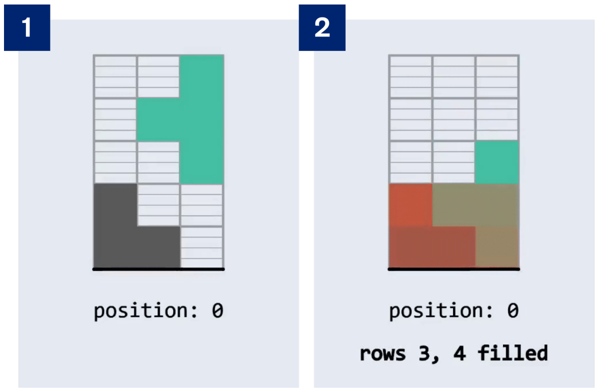
[1]The actual image is presented to candidates in an animated gif format, which can be viewed here.

Your task is to find the dropping position such that at least one full row is formed. As a dropping position, you should return the column index of the cell in the game field which matches the top left corner of the figure's 3 × 3 matrix. If there are multiple dropping positions satisfying the condition, feel free to return any of them. If there are no such dropping positions, return -1.

Note: The figure must be dropped so that its entire 3 × 3 matrix fits inside the field, even if part of the matrix is empty.

Examples

For
```
field = [[0, 0, 0],
         [0, 0, 0],
         [0, 0, 0],
         [1, 0, 0],
         [1, 1, 0]]
```
and
```
figure = [[0, 0, 1],
          [0, 1, 1],
          [0, 0, 1]]
```
The output should be `solution(field, figure) = 0`.
Because the field is a `3 x 3` matrix, the figure can be dropped only from position `0`. When the figure stops falling, two fully occupied rows are formed, so dropping position 0 satisfies the condition.



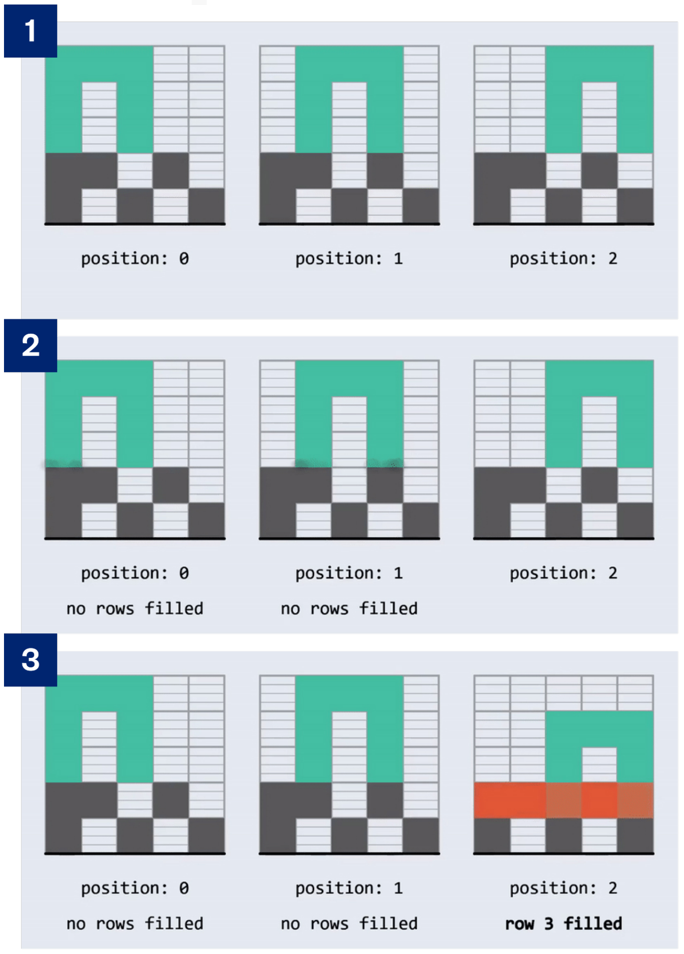
[2]

For
```
field =  [[0, 0, 0, 0, 0],
          [0, 0, 0, 0, 0],
          [0, 0, 0, 0, 0],
          [1, 1, 0, 1, 0],
          [1, 0, 1, 0, 1]]
```
and
```
figure = [[1, 1, 1],
          [1, 0, 1],
          [1, 0, 1]]
```

the output should be `solution(field, figure) = 2`.

[2] The actual image is presented to candidates in an animated gif format, which can be viewed here.

The figure can be dropped from three positions: `0`, `1`, and `2`. As you can see below, a fully occupied row will be formed only when dropping from position `2`:



[3]

Sample Solution (python)

```python
def solution(field, figure):
   height = len(field)
   width = len(field[0])
   figure_size = len(figure)

   for column in range(width - figure_size + 1):
       row = 1
       while row < height - figure_size + 1:
           can_fit = True
           for dx in range(figure_size):
               for dy in range(figure_size):
                   if field[row + dx][column + dy] == 1 and figure[dx][dy] == 1:
                       can_fit = False
           if not can_fit:
               break
           row += 1
       row -= 1

       for dx in range(figure_size):
           row_filled = True
```

[3] The actual image is presented to candidates in an animated gif format, which can be viewed here.

```python
            for column_index in range(width):
             if not (field[row + dx][column_index] == 1 or
                     (column <= column_index < column + figure_size and\
                   figure[dx][column_index - column] == 1)):
                 row_filled = False
            if row_filled:
                return column
    return -1
```

Module 4 – Problem Solving

Given an array of unique integers numbers, your task is to find the number of pairs of `indices (i, j)` such that `i ≤ j` and the sum `numbers[i] + numbers[j]` is equal to some power of `2`. Note: The numbers 20 = 1, 21 = 2, 22 = 4, 23 = 8, etc. are considered to be powers of 2.

Examples

For `numbers = [1, -1, 2, 3]`, the output should be `solution(numbers) = 5`.

- There is one pair of indices where the sum of the elements is 20 = 1: (1, 2): `numbers[1] + numbers[2] = -1 + 2 = 1`
- There are two pairs of indices where the sum of the elements is 21 = 2: `(0, 0)` and `(1, 3)`
- There are two pairs of indices where the sum of the elements is 22 = 4: `(0, 3)` and `(2, 2)`
- In total, there are `1 + 2 + 2 = 5` pairs summing to powers of `2`.

For `numbers = [2]`, the output should be `solution(numbers) = 1`.

- The only pair of indices is `(0, 0)` and the sum is equal to 22 = 4. So, the answer is `1`.

For `numbers = [-2, -1, 0, 1, 2]`, the output should be `solution(numbers) = 5`.

- There are two pairs of indices where the sum of the elements is 20 = 1: `(2, 3)` and `(1, 4)`
- There are two pairs of indices where the sum of the elements is 21 = 2: `(2, 4)` and `(3, 3)`
- There is one pair of indices where the sum of the elements is 22 = 4: `(4, 4)`
- In total, there are 2 + 2 + 1 = 5 pairs summing to powers of `2`.

Guaranteed constraints:

- `1 ≤ numbers.length ≤ 105`
- `-106 ≤ numbers[i] ≤ 106`

Sample Solution (python)

```python
from collections import defaultdict

 def solution(numbers):
   counts = defaultdict(int)
   answer = 0
   for element in numbers:
       counts[element] += 1
       for two_power in range(21):
           second_element = (1 << two_power) - element
           answer += counts[second_element]
   return answer
```