ART+LOGIC
SOFTWARE INNOVATION

# The Art+Logic Programming Challenge

Thanks for your interest in Art+Logic.

This file contains the instructions for the first part of the Art+Logic Programming Challenge.

This test is an important part of our hiring process; over the years, we have learned that there's not always a strong correlation between the contents of a candidate's resume and their ability to design and implement code at the level that we demand. The only way to determine how well someone writes code is to ask them to roll up their sleeves and write some. Rather than follow the current trend of asking someone to freestyle some algorithm code from memory onto a whiteboard under pressure in an interview, it's better to approximate actual working conditions.

## About Us and What We Do

There are a lot of different kinds of companies that hire developers, so it might be useful to explain our niche in the world as a distributed custom software development firm.

1. **Distributed:** Our development team works from all across North and South America; typically from our home offices. Art+Logic was one of the first completely distributed development firms in the world, since our founding in 1991.

2. **Custom:** We execute software projects for clients. Those clients end up representing every kind of industry and every level of growth and sophistication between Fortune 500 companies and a lone inventor with an idea and some money to make it real. Sometimes those clients have strong opinions about the languages, technologies, frameworks, and approaches used for their projects, and we endeavor to adapt ourselves to those needs. We need to work with developers who bring (or are excited to develop) a broad skill set and the flexibility to move quickly between those skills.

3. **Software Development:** this is perhaps a more subtle point. Many different kinds of companies hire software developers, but not all of those companies are *software companies*. Even within the universe of companies that think of themselves as software companies, there's a continuum between what we'll call a 'software development firm' that develops software which *is* their clients' business, and an 'agency' that develops software *about* their clients' business. Both companies use similar tools and techniques, but toward different ends, and facing different challenges. While Art+Logic is experienced at both ends of that spectrum, our projects tend more toward the 'software development' end.

Completing this test will give you an opportunity to display your abilities on some tasks that are not too different from those that are routinely encountered on Art+Logic projects. It's fairly typical on a project that a developer will be handed a fairly high-level specification for a piece of functionality and be asked to design and implement a clean, elegant, flexible solution to the problem in a reasonably short period of time. We also end up needing to jump into projects that are already in progress. Sometimes this means stepping into an existing Art+Logic project, but it might also mean inheriting a pile of sometimes broken and undocumented code written by a client (and typically, the original programmer responsible for this code is long gone, so debugging skills and detective work are required).

Please bear in mind while completing this exercise that your goal should not be just to demonstrate your competence. We evaluate many more applicants than we have the capacity to hire; this test is key to identifying the most exceptional.

## Who We're Looking For

We'll be considering how your skills and experience match up against the types of work that we are most commonly hired to do. Our clients come to us with a wide variety of needs and requirements, and our goal in recruiting is to find great developers who we would be able to staff onto as many different types of projects as possible. It's difficult for us to keep some sorts of specialists busy—for example, we don't do a lot of legacy J2EE or Perl work, so an amazing developer with those skills but not much else would be difficult for us to keep busy, and therefore that developer would be unlikely to proceed very far into our recruiting process.

Our ideal candidates have a good mix of:

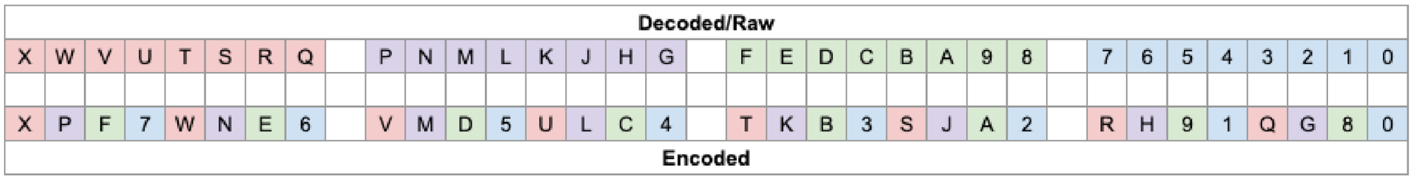
- **'Modern' web development:** HTML5 and its associated APIs, heavy client-side JavaScript (we also make use of JavaScript libraries/frameworks like jQuery, Angular, Backbone/Marionette, React, Vue) and server-side development, most often using Python and C#.
- **Mobile development:** Most commonly we are engaged to do iOS development, but we also do Android work.
- **'Other':** we still do a fair amount of Windows and Mac desktop development, especially applications and plug-ins for audio/music applications and in other domains where realtime performance or interactions with hardware require it. Experience with embedded development or other very low-level programming like device drivers is a plus.

# Part 1: The Weird Text Format (8-bit)

A recurring theme in Art+Logic projects is the need to process data that is provided in unusual or proprietary formats, regardless of whether a project is targeting desktop, mobile, embedded, or the web, so asking applicants to demonstrate their ability to work with low-level data formats is a traditional initial step.

For this challenge, we'll use a made-up format that encodes bundles of 4 characters by scrambling them into 32-bit integer values for transmission, then reverses the operation on the receive end to reconstitute the original text.

The below image shows how the individual bits in the original raw text (top row) are mixed together into the 32-bit output value on the bottom row:



| | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Decoded/Raw** | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| X | W | V | U | T | S | R | Q | | P | N | M | L | K | J | H | G | | F | E | D | C | B | A | 9 | 8 | | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| X | P | F | 7 | W | N | E | 6 | | V | M | D | 5 | U | L | C | 4 | | T | K | B | 3 | S | J | A | 2 | | R | H | 9 | 1 | Q | G | 8 | 0 |
| **Encoded** | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |

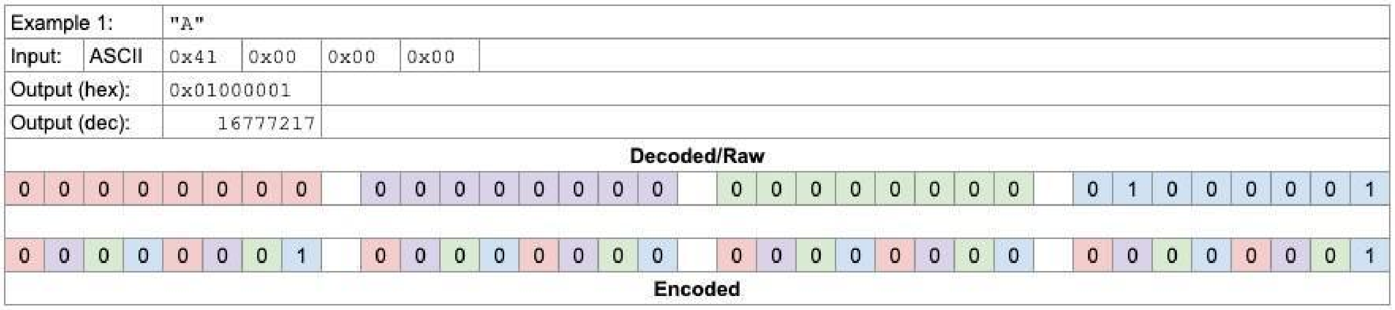
*NOTE that in this image, the `0-9` and `A-X` are only used to identify individual bit positions within the 32-bit data. The position labeled `0` is the least significant bit of the data, `X` is the most significant bit.*

If this exercise seems contrived based on your experience, consider that it's at least vaguely similar to real-world things like Reed-Solomon error correction codes or some interleaved data formats that might be transmitted by IoT edge devices.

Some examples should make things more clear.

### Example 1: Single Character

When encoding a chunk with fewer than four characters, the input should be zero-padded to a length of four before encoding. The first character in the input is treated as the least significant byte in the example images:

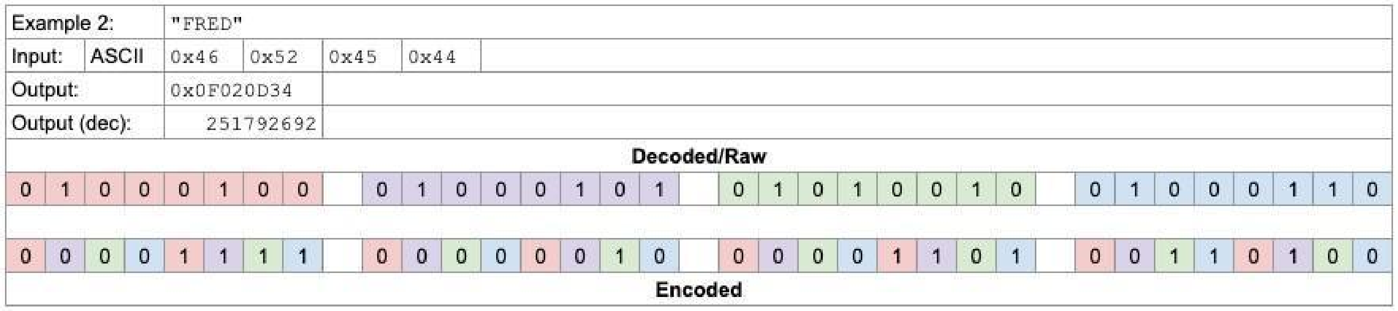


| | | | | | |
|---|---|---|---|---|---|
| Example 1: | | `"A"` | | | |
| Input: | ASCII | `0x41` | `0x00` | `0x00` | `0x00` |
| Output (hex): | | `0x01000001` | | | |
| Output (dec): | | `16777217` | | | |

| | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Decoded/Raw** | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 |
| | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| **Encoded** | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |

## Example 2: Full Bundle

Here we see a complete chunk of 4-characters—refer to a table of ASCII character values to see how the incoming text data is shifted into the Decoded/Raw buffer, and follow the individual bits into their final positions.



| Example 2: | | "FRED" | | | |
|---|---|---|---|---|---|
| Input: | ASCII | 0x46 | 0x52 | 0x45 | 0x44 |
| Output: | | 0x0F020D34 | | | |
| Output (dec): | | 251792692 | | | |

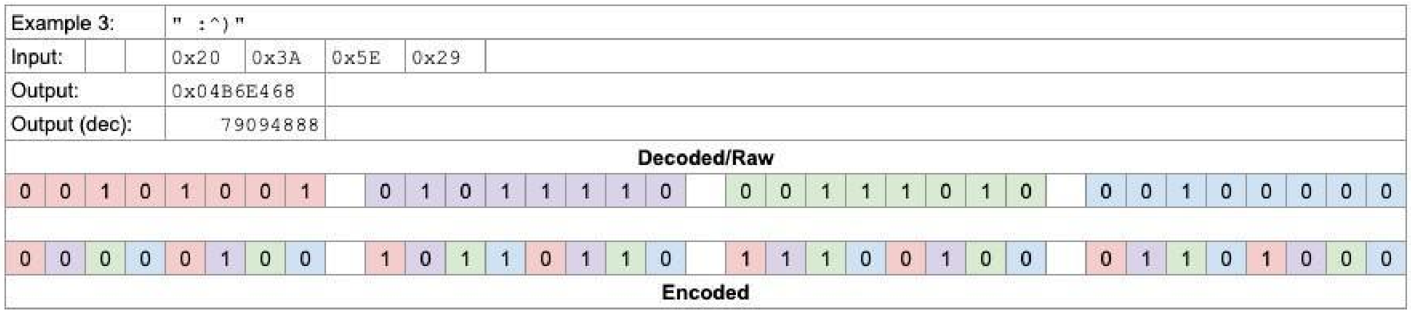
Decoded/Raw

| | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 0 |
| 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 0 |

Encoded

## Example 3: Non-Alphanumerics

Another simple example with some characters outside of the regular alphanumeric range:



| Example 3: | | | " :^)" | | | |
|---|---|---|---|---|---|---|
| Input: | | | 0x20 | 0x3A | 0x5E | 0x29 |
| Output: | | | 0x04B6E468 | | | |
| Output (dec): | | | 79094888 | | | |

Decoded/Raw

| | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | | 0 | 1 | 0 | 1 | 1 | 1 | 1 | 0 | | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 0 | | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 0 |

Encoded

## More examples

| Raw Characters | Encoded Value (dec) |
|---|---|
| "foo" | 124807030 |
| "  foo" | 250662636 |
| "foot" | 267939702 |
| "BIRD" | 251930706 |
| "...." | 15794160 |
| "^^^^" | 252706800 |
| "Woot" | 266956663 |
| "no" | 53490482 |

## The Task

The task for part 1 of this challenge is to write just enough code to unlock the rest of it. We expect that a skilled developer should be able to implement the encoding step of this in an hour or so using the above diagrams and example data.

To move forward in our recruiting process, create a bit of code that can encode a single 4-character block of text as described above. If you have an urge to implement your solution by converting input into a text string containing "binary" data, resist that urge and find a different approach. See below for more details on how to submit materials to complete part 1.

### Guidelines, Requirements, and Other Constraints

For part 1 we won't ask to see your code, but since part 2 will build on part 1, developers who make it to that stage will want to prepare for some constraints that will apply there.

Your program must be written using one of our preferred programming languages:

- C++
- C#
- JavaScript
- Objective-C
- Python*
- Swift
- Kotlin

* We are only accepting Python solutions targeting Python 3.x.

Avoid the use of any third-party libraries or frameworks where possible—remember that the goal here is to help us see how awesome your code can be when you're working on a problem where none of those libraries exist yet.

In production, we also require that code written for Art+Logic conform to the conventions outlined in the Art+Logic Programming Style Guide (at https://styleguide.artandlogic.com).

Your completed part 2 submission will be evaluated by a senior Art+Logic developer, who will be looking for several things, but most importantly:

- Does this code work correctly?
- Is it well designed and cleanly implemented?
- Was it completed in a reasonable amount of time?
- If I had to maintain this code, would that make me happy or angry?
- How closely does it follow our Style Guide?
- Did the applicant read the instructions fully?

## Submitting Your Response

To continue with your application, create a plain text file named with your name in the format `{firstname}_{lastname}.txt` that includes the following data:

**Time taken to complete part 1:** *(in hours)*

**Email address:**

**Challenge response:** *(integer value)*

The Challenge response value should be the encoded integer value that your code calculates for the first 4 characters of your email address as entered in this document.

Enter this value as a decimal number, so a user with the email address `a@b.com` would enter `131107009`.

Upload this file to the form located at the URL specified in the email that this challenge document was attached to.

*File version 2022-02-21*