INSTRUCTIONS

The purpose of this assessment is to complete a simple programming assignment. You are required to produce working and tested source code to solve the problem below.

There are no design constraints placed on you other than your code must be fully self-contained and need no external libraries other than the core language of your choice.

However you choose to compile / build your code, please do not make any assumptions about tooling. You should submit two files:

- a zip file containing the complete code
- a README.txt explaining how to build / run the code from the command line

You do not need 100% test coverage, we just want to see that you know how to write testable code. A few unit tests will suffice. Equally, not all error conditions need to be dealt with – we just want to see some examples of how you deal with them. Logging is similar – you do not need to log everything, but we want to see you demonstrate that you know how to log with forethought regarding future troubleshooting / debugging.

Do not spend any more than a few hours on the exercise. **We are interested in how you have mentally modelled the problem - particularly which data structures you chose.**

Be prepared to walk through your code with the assessor, answering questions about why you chose (or didn't choose) particular things, justifying the design decisions you made and generally talking through your thought process. There's not really any 'right' or 'wrong' way to solve this problem, we're interested in seeing how you approached it and thought about it and modelled it.

At interview, we will use your submission as the basis for discussion. **We will expand the scope of the exercise and throw in a couple of NFRs and strange requirements**. You will be expected to use the whiteboard to help explain how you would deal with these new requirements. Something you will definitely be asked to discuss is how you would scale the code you've written, and what some of the tradeoffs would be.

You can choose any language(*) you like, but our preference is in this order:

Golang
Python
Java
C / C++

(Anything else you choose)

(*) Javascript is not a language. Please do not choose this.

CODING ASSIGNMENT / FUNCTIONAL REQUIREMENTS

Write a program and associated unit tests that can price a basket of goods, accounting for special offers.

The goods that can be purchased, which are all priced in GBP, are:

Soup – 65p per tin
Bread – 80p per loaf
Milk – £1.30 per bottle
Apples – £1.00 per bag

Current special offers are:

Apples have 10% off their normal price this week
Buy 2 tins of soup and get a loaf of bread for half price

The program should accept a list of items in the basket and output the subtotal, the special offer discounts and the final price.

Input should be via the command line in the form PriceBasket item1 item2 item3 ...

For example: PriceBasket Apples Milk Bread

Output should be to the console, for example:

```
Subtotal: £3.10
Apples 10% off: -10p
Total: £3.00
```

If no special offers are applicable, the code should output:

```
Subtotal: £1.30
(no offers available)
```

Total: £1.30

NON-FUNCTIONAL REQUIREMENTS

The code and design should meet the functional requirements but be sufficiently flexible to allow for future extensibility. Particular attention will be paid to how well (or how much) you've considered how things will be at large-scale (e.g. if there were ten of thousands of products and offers, or if there were thousands of baskets to be priced) - particularly with regard to:

- thread safety
- resource usage
- speed

You don't need to have solved every edge case for these things, we're just looking to see that you've structured your code in a way that shows you've considered them. In some cases, it's good enough to just comment something as being inefficient or not thread-safe. We can then discuss in your tech interview what the alternatives are and some of the pros/cons around those alternatives.

The code should be well structured, suitably commented, have error handling and unit tests (neither of which needs to be exhaustive). The comments should help give context about *why* you've done (or not done) a particular thing, as opposed to just saying what the code already says. Any log messages you include should indicate forethought about supporting / debugging in a live environment.

Bear in mind that the whole exercise is for us to get to know how you think about problems, and give us things to talk about in a tech interview rather than an exam question with a right or wrong answer.