Emurgo R&D : Software Engineer Test Tasks (v3.6)

The followin tasks are more or less ordered by complexity, but for different people different tasks might appear easier than others. You can solve them in any order and if you feel like a certain task is out of scope of your professional specialisation - just leave a comment about it.

Solutions are preferred in JS (Flow is a bonus) or TS languages, alternatively you can send it in any language, but it will then be reviewed in accordance to the specifics of the position you are applying for.

1. Permutations

Input: a string of unknown length that can contain only three characters: `0`, `1`, or `*`. For example: `"101*011*1"`

Output: an array of strings where each string is a permutation of the input string with the `*` character replaced by a `0` or a `1`. For example: for input `"*1"` the result array is `["01", "11"]`

Tasks: Implement the function/program that will give correct output for all possible inputs within given restrictions. Write a documenting comment explaining the algorithmic complexity of the program.

2. Data-model

Intro: we are creating a software system for a pizza restaurant, one of the modules is supposed to handle the management of various pizza recipes and how the orders are put together, and a big part of the module will be the control of food types, the potential allergens in recipes, and calories counting.

Requirements: you are commissioned to write a data-model API (classes, interfaces, etc) which will then be used by the entire module to exchange the data about the recipes, ingredients, and all that. And also couple helper function that demonstrate the usage of your data-model.

Tasks: Create a data-model to represent pizza ingredients and pizza recipes, including the food types and allergens. Create a function `hasAllergens` where we can pass a certain pizza recipe and an array of allergens and receive `true` if the recipe contains any of the allergens. Create a function `hasFoodTypes` where we can pass a certain pizza recipe and an array of food types and receive `true` if this recipe contains any of the food types. Create functions `removeAllergens` and `removeFoodTypes` with similar parameters but they should return the same pizza recipe but with the allergens and food types removed. Create function `removeIngredients` where we can pass a certain pizza recipe and an array of ingredients and receive the pizza recipe without the specified ingredients. Create function `doubleIngredients` with the same parameters but it returns the recipe with the specified ingredients in double amount. Create function `getCalories` where we can pass a certain pizza recipe and it will return the number of total calories in that recipe.

Example of usage for the functions:

1. `hasFoodTypes` when user checks they want to only see pizza with mushrooms
2. `hasAllergens` when user wants to highlight any pizzas that might contain soy in it
3. `removeIngredients` when user checks they don't want Jalapeño pepper
4. `doubleIngredients` when user checks they want double cheese
5. `removeFoodTypes` when user wants to check what the selected pizza will look like with no meat
6. `getCalories` when user wants to check what the number will be for the resulting recipe

3. Building a transaction

Intro: we are creating a software solution for a bank and you are being commissioned to build one function out of the entire system, but this function will construct an internal transaction object, which moves the funds when some accounts are getting closed.

Requirements: from times to times the bank will want to run a big internal operation which closes multiple reserve accounts and moves all the funds into multiple recipient accounts according to the corresponding credit, and the task of the function is to detail how exactly the funds will move between the accounts and also calculate the cost of the operation itself.

Tasks: implement function `newRebalancingTx(closingAccounts, recipientAccounts)`. Where `closingAccounts` is an array of objects like `{ accountId: string, amount: number }`, `recipientAccounts` is an array of objects like `{ accountId: string, credit: number }`. And the function is supposed to return an object of this format:

```
{
  transfers: [
    [fromAccountId, toAccountId, value],
  ],
  operationalFee: number,
}
```

1. The `transfers` array must contain a listing of how the money must be moved from all closed accounts into all new accounts. For example, if we have one closing account with ID `acc1` and amount `1000` and two receipients with IDs `rec1` and `rec2` and credit values `500` and `400` - the first two elements in the transfers array will be: `[acc1, rec1, 500], [acc1, rec2, 400]`.
2. A transfer cannot have a zero or negative value, such transaction will not be accepted and will cause an error.
3. If there are more money on a closed account than required for one receipient - then the remainder should go into the next receipient account.
4. If there are not enough money in total in all closed accounts to fulfill total credit in recipients - an error must be raised: “`not enough funds for rebalance`“.
5. If there is more money available in closed accounts than total recipients credit - the remainder should be transferred from each account with a remainder to a new reserve which is added to transfers with account ID `null`. For example, if closed `acc1` and `acc2` with amounts `500` and `500` and one recipient `rec1` with credit `400` - then transfers would be `[[acc1, rec1, 400], [acc1, null, 100], [acc2, null, 500]]`.

Here’s the catch, though, for any rebalancing transaction a special operational-fee must be included, which depends on the number of transfers and is calculated as `transfers.length * 10`. This amount must be subtracted from the remainder transferred into reserve. For example, if closed `acc1` and `acc2` with amounts `500` and `500` and one recipient `rec1` with credit `400` - then the full correct response will be:

```
{
  transfers: [[acc1, rec1, 400], [acc1, null, 100], [acc2, null, 480]],
  operationalFee: 20,
}
```

So the end goal of the produced tx is to make sure every single bit of money on the input matches exactly to all the outputs plus the operational fee.

If total value of closed accounts is not enough to fulfill the total recipient credit AND the operational fee - an error must be raised: “`not enough funds for rebalance`“.