## **قسمت اول:** بررسی درست بودن شماره‌ی کارت‌های شبکه‌ی شتاب

شبکه‌ی شتاب برای بررسی درست بودن شماره‌ی کارت‌ها از یک چک‌سام ساده استفاده می‌کند. نحوه‌ی محاسبه‌ی این چک‌سام به این صورت است:

۱. با شروع از راست رقم‌ها را یکی در میان دو برابر می‌کنیم. (یعنی رقم اول از راست عوض نمی‌شود، رقم دو برابر می‌شود، رقم سوم عوض نمی‌شود...)
۲. اکنون لیستی شامل اعداد یک یا دو رقمی داریم. همه‌ی ارقام موجود را جمع می‌کنیم. (یعنی برای اعداد دورقمی جمع ارقام حساب می‌شود نه خود عدد).
۳. اگر عدد به دست آمده بر ۱۰ بخش‌پذیر بود، شماره‌ی کارت صحیح است. در غیر این‌صورت شماره‌ی کارت غلط است.

مثال:

2345 6789 =>

4 3 8 5 12 7 16 9 =>

4+3+8+5+1+2+7+1+6+9 =>

46

یعنی 2345 6789 یک شماره‌ی کارت صحیح نیست.

با پیاده‌سازی توابع مورد نیاز بررسی صحت شماره‌ی کارت‌ها را پیاده‌سازی کنید.

**تمرین اول:** ابتدا لازم است که ارقام یک عدد را به دست بی‌آوریم. توابع زیر را تعریف کنید:

```
toDigits :: Integer -> [Integer]
toDigitsRev :: Integer -> [Integer]
```

تابع toDigits باید اعداد صحیح مثبت را به لیست ارقام آن‌ها تبدیل کند. برای ورودی صفر یا اعداد منفی باید یک لیست خالی بازگردانده شود. تابع toDigitsRev مانند تابع toDigits است با این تفاوت که ترتیب ارقام برعکس شده‌باشد.

```
toDigits 1234 == [1,2,3,4]
toDigitsRev 1234 == [4,3,2,1]
toDigits 0 == []
toDigits (-17) == []
```

**تمرین دوم:** بعد از به‌دست‌آوردن ارقام لازم است که آن‌ها را یکی‌درمیان دوبرابر کنیم. تابع زیر را تعریف کنید:

```
doubleEveryOther :: [Integer] -> [Integer]
```

به یاد داشته‌باشید که این تابع باید ارقام را یکی‌درمیان **از راست** دوبرابر کند.

```
doubleEveryOther [8,7,6,5] == [16,7,12,5]
doubleEveryOther [1,2,3] == [1,4,3]
```

**تمرین سوم:** خروجی تابع doubleEveryOther شامل اعداد یک‌رقمی و دورقمی است. تابع زیر را تعریف کنید:

```
sumDigits :: [Integer] -> Integer
```

این تابع جمع همه‌ی **ارقام موجود در لیست** را محاسبه می‌کند.

```
sumDigits [16, 7, 12, 5] == 22
```

**تمرین چهارم:** تابع زیر را تعریف کنید:

```
validate :: Integer -> Bool
```

این تابع با استفاده از توابع قبل مشخص می‌کند که شماره‌ی کارت داده‌شده صحیح است یا نه.

```
validate 5171251432710904 == True
validate 5171251432710905 == False
```

## قسمت دوم: مساله‌ی برج‌های هانوی

با مساله‌ی برج‌های هانوی آشنایی دارید. در این مساله تعدادی دیسک با اندازه‌های مختلف به ترتیب از بزرگ‌ترین دیسک در زیر تا کوچک‌ترین دیسک در بالایی میله، بر روی یک میله (مثلا میله‌ی ***الف***) قرار دارند. هدف مساله انتقال همه‌ی این دیسک‌ها به یک میله‌ی دیگر (مثلا میله‌ی ***ب***) با استفاده از یک میله‌ی کمکی (مثلا میله‌ی ***پ***) است. قانون عملیات انتقال این است که هیچ دیسکی نمی‌تواند بر روی یک دیسک کوچک‌تر قرار بگیرد.

برای انتقال n دیسک از میله‌ی ***الف*** به میله‌ی ***ب*** با استفاده از میله‌ی ***پ*** یک راه‌حل بازگشتی به شرح زیر وجود دارد:

۱. به صورت بازگشتی n-1 دیسک را با استفاده از میله‌ی ***ب*** به میله‌ی ***پ*** منتقل کنید.
۲. دیسک nام باقی‌مانده در میله‌ی ***الف*** را به میله‌ی ***ب*** منتقل کنید.
۳. n-1 دیسک منتقل شده به میله‌ی ***پ*** را به صورت بازگشتی با استفاده از میله‌ی ***الف*** به میله‌ی ***ب*** منتقل کنید.

**تمرین پنجم:** تابع hanoi را با type مشخص شده در زیر پیاده‌سازی کنید.

```
type Peg = String
type Move = (Peg, Peg)
hanoi :: Integer -> Peg -> Peg -> Peg -> [Move]
```

این تابع باید تعداد دیسک‌ها و نام سه میله را گرفته و حرکت‌های لازم برای انتقال تعداد مورد نظر از میله‌ی اول به میله‌ی دوم با استفاده از میله‌ی سوم را بازگرداند.

توجه کنید که کلمه‌ی کلیدی type در زبان Haskell یک نام جدید برای یک type تعریف می‌کند. به عبارت دیگر **Peg** و **String** هر دو یک چیز هستند، اما استفاده از type synonymها باعث می‌شود کد خواناتر و اسم typeها بامفهوم‌تر باشد.

```
hanoi 2 “a” “b” “c” == [(“a”,”c”), (“a”,”b”), (“c”,”b”)]
```