Dobrze wiedzieć

W trakcie rozwiązywania zadań na rozmowach rekrutacyjnych umiejętność szacowania złożoności obliczeniowej jest niezwykle ważna. Przyda się też umiejętność pisania testów jednostkowych. Dzięki nim bardzo łatwo przetestujesz działanie algorytmu.

- Podstawy złożoności obliczeniowej,
- Test Driven Development na przykładzie,
- Testy jednostkowe z Junit 5.

Podstawą jest oczywiście znajomość języka programowania, ja używał będę Javy jednak możesz użyć dowolnego języka programowania. W opisie problemu będę używał “pseudo kodu”. Przykładowe rozwiązania będą w języku Java.

- Kurs programowania w języku Java.

Kilka wskazówek dotyczących rozwiązywania zadań

Jedno z tych zadań sam miałem na rozmowie kwalifikacyjnej. Rozmowa była przeprowadzana na stanowisko starszego programisty. Sam problem nie jest dość trudny. Zauważyłem, że jest to częsta praktyka. Problemy do rozwiązania na rozmowach kwalifikacyjnych przeważnie służą tylko do tego, żeby zweryfikować czy kandydat zna składnię danego języka. Dodatkowo zadania tego typu sprawdzają umiejętność analizowania problemu i szukania rozwiązania.

Na rozmowie rozwiązałem ten problem algorytmem o najlepszej złożoności czasowej, jednak nie był on optymalny pod kątem użycia pamięci. Mimo tego, że moje rozwiązanie nie było idealne dostałem ofertę pracy. Tutaj chcę Ci pokazać, że nawet jeśli nie rozwiążesz problemu w idealny sposób, a będziesz sensownie kombinować, uda się.

Zacznij od najprostszego rozwiązania. Zacznij od czegokolwiek i później głośno zastanawiaj się nad minusami Twojego rozwiązania. Prowadzący rozmowę widząc Twoje zaangażowanie często pomogą i nakierują Cię na lepsze rozwiązanie problemu.

Zadanie 1. - czy liczba jest cykliczna

Czym jest liczba cykliczna

Zanim przejdę do treści zadania musisz wiedzieć czym jestliczba cykliczna. Liczba cykliczna to liczba całkowita, której cykliczne permutacje cyfr są możliwe do uzyskania przez mnożenie liczby przez kolejne liczby naturalne. Przykładową liczbą cykliczną jest 142857. Wyniki mnożenia tej liczby przez pierwsze 6 liczb naturalnych dają jej permutacje cykliczne:

```
142857 * 1 = 142857
142857 * 2 = 285714
142857 * 3 = 428571
142857 * 4 = 571428
142857 * 5 = 714285
142857 * 6 = 857142
```

Permutacja cykliczna może brzmieć jak coś skomplikowanego. W praktyce powstaje ona przez wstawianie pierwszego elementu danego łańcucha na koniec. Na przykład permutacjami cyklicznymi łańcucha znaków `abcd` są:

- `abcd`
- `bcda`
- `cdab`
- `dabc`

Zadanie do wykonania

Napisz funkcję `isCyclic`, która jako argument dostaje dowolnie dużą dodatnią liczbę całkowitą w postaci łańcucha znaków. Liczba może być poprzedzona zerami, więc `"0123"` jest poprawnym wejściem programu. Zadaniem jest napisanie funkcji `isCyclic`, która sprawdzi czy dana liczba jest liczbą cykliczną.

```
isCyclic("142857") == true
isCyclic("012233") == false
```

W przykładzie pierwsza liczba jest liczbą cykliczną. Druga linijka pokazuje przykład, dla którego `isCyclic` powinna zwrócić wartość `false`.

Od czego zależy złożoność

W przypadku tego zadania danymi wejściowymi jest łańcuch znaków. W zadaniach tego typu długość takiego łańcucha używana jest do szacowania złożoności obliczeniowej i pamięciowej. Zatem `n` użyte poniżej odnosi się właśnie do długości wejściowego łańcucha znaków.

Najprostsze rozwiązanie problemu

Zacznę od najprostszego rozwiązania problemu. Parametrem funkcji jest łańcuch znaków reprezentujący liczbę. Żeby zobaczyć czy ta liczba jest cykliczna wygeneruję wszystkie jej permutacje cykliczne i będę sprawdzał czy mnożąc liczbę przez kolejne wartości od 1 do N

wynik będzie znajdował się we wcześniej przygotowanych permutacjach. Proszę spójrz na przykładowe rozwiązanie:

```
public boolean isCyclicNaive(String number) {
    String[] permutations = new String[number.length()];

    for (int index = 0; index < permutations.length; index++) {
        permutations[index] = number.substring(index) + number.substring(0, index);
    }

    BigInteger value = new BigInteger(number);
    String formatString = "%0" + number.length() + "d";

    outerLoop: for (int multiplicator = 2; multiplicator <= number.length(); multiplicator++) {
        BigInteger multiplication = value.multiply(BigInteger.valueOf(multiplicator));
        String formattedResult = String.format(formatString, multiplication);
        for (String permutation : permutations) {
            if (formattedResult.equals(permutation)) {
                continue outerLoop;
            }
        }
        return false;
    }

    return true;
}
```

Pierwsza pętla odpowiedzialna jest za tworzenie permutacji cyklicznych. Wewnątrz drugiej pętli sprawdzam czy mnożąc liczbę przez kolejne wartości od 2 do N uzyskam jedną z wcześniej przygotowanych permutacji. Posługuję się tutaj typem `BigInteger` aby móc pracować na liczbach większych niż te, które mogę przechowywać w zmiennej typu `long`.

Złożoność obliczeniowa

Pierwsza pętla ma złożoność `O(n^2)`. Dzieje się tak ponieważ metoda `substring` ma złożoność `O(n)`. Kolejna pętla jest zagnieżdżona i ma złożoność `O(n^3)`. Tym razem złożoność "psuje" operacja `multiply`, która ma złożoność obliczeniową `O(n^2)`. Więc finalnie złożoność obliczeniowa tego algorytmu to `O(n^3)`.

Złożoność pamięciowa

W przypadku tego algorytmu przechowuję listę permutacji w tablicy. Tablica zawiera `N` permutacji. Każda permutacja ma długość `N`. Zatem finalna złożoność pamięciowa to `O(n^2)`.

Rozwiązanie bazujące na właściwościach liczb cyklicznych

Czytając o liczbach cyklicznych dowiedziałem się kilku istotnych rzeczy:

- liczby cykliczne tworzone są na podstawie liczb pierwszych,
- długość liczby cyklicznej jest o jeden większa niż liczba pierwsza użyta do generowania liczby cyklicznej,
- liczba cykliczna jest cyklicznym rozwinięciem ułamka `1/liczba pierwsza do generacji`.

Mając takie informacje podszedłem do problemu od drugiej strony. Zamiast sprawdzić czy dana liczba jest cykliczna wygenerowałem liczbę, która powstałaby na podstawie dzielenia `1/liczba pierwsza do generacji`. Następnie porównuję tak uzyskaną liczbę z tą przekazaną jako argument metody. Jeśli są sobie równe wówczas przekazana liczba jest liczbą cykliczną. Proszę spójrz na implementację:

```
public boolean isCyclicGeneration(String number) {
    int base = 10;
    int generatingPrime = number.length() + 1;

    StringBuilder representation = new StringBuilder();

    int step = 0;
    int reminder = 1;
    do {
        step++;
        int currentValueToDivide = reminder * base;
        int currentDigit = currentValueToDivide / generatingPrime;
        reminder = currentValueToDivide % generatingPrime;
        representation.append(currentDigit);
    } while (reminder != 1 && step < generatingPrime);

    return number.equals(representation.toString());
}
```

Na początku ustawiam niezbędne zmienne. Następnie wewnątrz pętli obliczam kolejne wartości ułamka. Tak uzyskane liczby dodaję do bufora `representation`, który następnie porównuję z przekazaną liczbą.

Warunek `reminder != 1` wykrywa wystąpienie okresu w rozwinięciu dziesiętnym ułamka. Więcej na temat “okresu ułamka” przeczytasz w artykule opisującym liczby zmiennoprzecinkowe.

Warunek `step < generatingPrime` jest potrzebny, aby uniknąć nieskończonej pętli. Taki przypadek mógłby mieć miejsce jeśli metoda jako parametr otrzymałaby liczbę, która nie jest cykliczna.

Złożoność obliczeniowa

W przypadku tego rozwiązania występuje wyłącznie jedna pętla. Zatem złożoność obliczeniowa tego algorytmu to `O(n)`.

Złożoność pamięciowa

Algorytm do działania potrzebuje kilku zmiennych. Jedna z nich, `representation`, urośnie do długości `N`. Zatem w tym przypadku złożoność pamięciowa tego algorytmu to `O(n)`.

Bardziej wydajne rozwiązanie problemu

Masz jakiś pomysł? Z chęcią poznam Twoje podejście do rozwiązania tego problemu. Wrzuć swój kod na githuba i podziel się rozwiązaniem. Pamiętaj, żeby przetestować poprawność swojego rozwiązania. Możesz to zrobić przy pomocy testów jednostkowych, które przygotowałem.

Zadanie 2. - znajdź brakujący element

Zadanie do wykonania

Napisz funkcję `findMissing`, która jako argument przyjmuje tablicę N liczb całkowitych z zakresu od 0 do N. W tablicy wszystkie elementy są unikalne. Liczb z zakresu `<0, N>`, jest `N + 1`. Tablica ma długość N. W tablicy brakuje jednego elementu z zakresu. Funkcja `findMissing` powinna zwrócić brakujący element:

```
tablica = [0, 2, 1, 4]
findMissing(tablica) == 3
```

W przykładzie wejściem jest tablica, która ma 4 elementy. Brakuje w niej liczby 3, i właśnie taką liczbę zwraca funkcja `findMissing`.

Najprostsze rozwiązanie problemu

Zacznę od najprostszego rozwiązania problemu. Na wejściu mamytablicę[1], w której brakuje jednego elementu z zakresu `<0, N>`. Żeby znaleźć brakujący element można sprawdzić czy każdy z elementów występuje w tablicy. Pierwsza iteracja sprawdzi czy w tablicy występuje `0`, kolejna `1` i tak dalej aż dojdziemy do `N`. Proszę spójrz na przykładowe rozwiązanie:

```
private static int naiveFindMissing(int... array) {
    int missing = 0;
    boolean elementFound;
    for (int elementToFind = 0; elementToFind <= array.length; elementToFind++) {
        elementFound = false;
        for (int elementInArray : array) {
            if (elementToFind == elementInArray) {
                elementFound = true;
                break;
            }
        }
        if (!elementFound) {
            missing = elementToFind;
            break;
        }
    }
    return  missing;
}
```

Złożoność obliczeniowa

Ten algorytm działa i znajduje brakujący element. Aby znaleźć brakujący element algorytm używa zagnieżdżonych pętli. Ilość iteracji w każdej z nich zależna jest od liczby elementów. Złożoność obliczeniowa tego algorytmu to `O(n^2)`

Złożoność pamięciowa

Algorytm ten używa stałej liczby zmiennych, jest ona niezależna od wielkości danych wejściowych. W związku z tym złożoność pamięciowa tego algorytmu jest stała, wynosi ona `O(1)`.

Pamięciożerne rozwiązanie

Teraz spróbuję ugryźć problem z innej strony. Z opisu problemu wiem, że brakuje jednej liczby z zakresu `<0, N>`. Pomysł jest taki, aby stworzyć tablicę wartości logicznych (flag), które będą wskazywały czy dany element znajduje się w tablicy. Iterując po elementach tablicy wejściowej można oznaczać każdą z flag. Następnie iterując po tablicy z flagami można znaleźć tę, która ma wartość `false`. To ona będzie wskazywała brakujący element:

```java
private static int memoryGreedyFindMissing(int... array) {
    boolean[] foundElements = new boolean[array.length + 1];

    for (int element : array) {
        foundElements[element] = true;
    }

    for (int index = 0; index < foundElements.length; index++) {
        if (!foundElements[index]) {
            return index;
        }
    }
    throw new IllegalStateException("At least one flag should be equal false!");
}
```

Złożoność obliczeniowa

Algorytm wymaga przejścia przez wszystkie elementy tablicy wejściowej. Dodatkowo iterujemy po tablicy flag. Rozmiar tej tablicy jest także zależy od wielkości danych wejściowych. Uzyskujemy zatem lepszą złożoność obliczeniową niż w poprzednim przypadku `O(2n) = O(n)`.

Złożoność pamięciowa

Aby algorytm działał konieczna jest inicjalizacja tablicy z flagami. Jej rozmiar zależny jest od wielkości danych wejściowych. W związku z tym algorytm ten ma złożoność pamięciową `O(n)`. Zatem kosztem pamięci udało mi się poprawić wydajność algorytmu. W większości przypadków taka zmiana jest akceptowalna, jednak problem ten można rozwiązać jeszcze wydajniej.

Optymalne rozwiązanie

Z opisu zadania wiesz, że szukamy brakującej liczby z zakresu `<0, N>`. Liczby 0, 1, 2, …, N to ciąg arytmetyczny. Istnieje wzór, który pozwala na obliczenie sumy elementów ciągu arytmetycznego:

$$S_n = \frac{n(2a_1 + (n-1)r)}{2}$$

Załóżmy, że tablica wejściowa ma 100. Wiemy zatem, że szukamy liczby z zakresu 0 do 100. Liczb w tym zakresie jest 101, pierwszy element ma wartość 0 a różnica pomiędzy elementami wynosi 1. Podstawiając te wartości pod wzór otrzymujemy sumę elementów:

$$S_n = \frac{101 * (2 * 0 + (101 - 1) * 1)}{2} = \frac{101 * 100}{2} = 5050$$

Skoro znamy oczekiwaną sumę możemy zsumować zawartość przekazanej tablicy i odjąć tę wartość od oczekiwanej sumy. Wynik odejmowania to brakujący element:

```java
private static int optimalFindMissing(int... array) {
    int expectedSum = (array.length + 1) * array.length / 2;
    int actualSum = 0;

    for (int element : array) {
        actualSum += element;
    }

    return expectedSum - actualSum;
}
```

Złożoność obliczeniowa

W tym przypadku w rozwiązaniu jest wyłącznie jedna pętla o złożoności $O(n)$. Obliczenie sumy elementów ma złożoność $O(1)$. Więc finalnie złożoność obliczeniowa tego algorytmu to $O(n)$.

Złożoność pamięciowa

Algorytm ten wymaga stałej liczby zmiennych. Liczba ta nie jest zależna od wielkości danych wejściowych. Zatem złożoność pamięciowa tego algorytmu to $O(1)$.

Wyślij mi swoje zadanie

Jeśli chcesz abym spróbował omówić zadanie, które trafiło się Tobie na rozmowie daj znać. Zastrzegam jednak, że nie jestem alfą i omegą. Potrafię sobie wyobrazić problemy, na które nie znajdę najlepszego rozwiązania. Niemniej jednak postaram się rozwiązać to zadanie w najlepszy znany mi sposób. Zadania możesz wysłać na mój adres email marcin@samouczekprogramisty.pl.

> Często firmy zastrzegają sobie to, żeby nie rozpowszechniać zadań, które były na rozmowie kwalifikacyjnej. Jeśli tak było w Twoim przypadku proszę uszanuj wolę danej firmy i nie przesyłaj mi takiego zadania.

1. Właściwie jest to argument, który akceptuje zmienną liczbę argumentów. Taka sygnatura metody pozwala także na przekazanie tablicy. ↩