Magazine

Kombinacje, permutacje czyli kombinatoryka dla testera

Autor: Jacek Okrojek

O autorze: Absolwent Wydziału Fizyki Technicznej, Informatyki i Matematyki Stosowanej Politechniki Łódzkiej, specjalizacja Sieci i Systemy Teleinformatyczne. Tester, test leader, freelance developer, od ponad 6 lat zajmuje się testowaniem i tworzeniem oprogramowania w kraju i za granicą w firmie Ericpol Telecom i jako niezależny konsultant. Uczestniczył i nadzorował testy na poziomie podstawowym, funkcyjnym, integracyjnym i systemowym, prowadził szkolenia z zakresu testowania oprogramowania. Kontakt: jacek.okrojek@gmail.com

Sformatowano: Polski (Polska)

Intermediate
Level

5
Magazine Number

Testowanie oprogramowania
Section in the magazine

Wprowadzenie
W poniższym tekście zgromadziłem podstawowe informacje z zakresu kombinatoryki, przydatne przy planowaniu i tworzeniu przypadków testowych. Skupiłem się na przedstawieniu wygodnych i prostych w zastosowaniu algorytmów oraz praktycznym zastosowania każdego z nich. Implementacje algorytmów przedstawione są w języku Java, ale zamieszczone opisy pomogą wykorzystać je też w innych językach programowania.

Istnieje kilka bibliotek i narzędzi, które mogą pomóc testerom przy tworzeniu przypadków testowych, szczególnie gdy istnieje duża liczba parametrów wejściowych dla testowanych systemów. Nie zawsze możemy z nich skorzystać w naszym środowisku testowym lub nie do końca spełniają one nasze wymagania, dlatego uważam, iż warto przypomnieć sobie i poszerzyć wiedzę z zakresu kombinatoryki. Wykorzystanie jej przy automatyzacji testów pomoże nam zaoszczędzić czas. Żmudne sprawdzanie czy nie przeoczyliśmy jednej z możliwości jaka mogłaby zaistnieć nie jest zajęciem pasjonującym. Lepiej pozostawić to maszynom i zająć się bardziej interesującymi zadaniami.

Permutacje

Do ilustracji tego pojęcia posłużę się przykładem funkcji wykorzystywanej w telekomunikacji. Przy zestawianiu połączenia funkcja ta określa, jaki rodzaj kodowania będzie używany przy transmisji. Wykorzystuje przy tym listę kodeków obsługiwanych przez nadawcę, a kolejność elementów na liście określa priorytet sposobu kodowania. Sposób kodowania o niższym indeksie ma wyższy priorytet, czyli jest bardziej preferowany niż ten o indeksie wyższym. Mamy tu do czynienia z uporządkowaniem elementów. Oto przykładowa tablica z danymi wejściowymi dla naszej funkcji:

Indeks tablicy (priorytet)	0	1	2
Identyfikator sposobu kodowania	1	2	0

Sposoby kodowania

0 – 16kbps

1 – 8kbps

2 – 22kbps

Uszeregowane dane w takiej tablicy (może to być także lista) moglibyśmy nazwać permutacją bez powtórzeń sposobów kodowania (określony sposób kodowania może pojawić się w tablicy tylko raz). Zgodnie z definicją, permutacją bez powtórzeń zbioru złożonego z n-różnych elementów nazywamy każdy n-wyrazowy ciąg utworzony ze wszystkich wyrazów zbioru.

Gdyby nasz przykładowy system obsługiwał tylko 3 sposoby kodowania, łatwo przewidzieć, iż istnieje tylko 6 możliwości ustawienia elementów w tablicy. Przyporządkowując sposobom kodowania liczbowe identyfikatory wszystkie możliwości przedstawione są na przykładzie 1.

Przykład 1

{0, 1, 2} {0, 2, 1} {1, 0, 2} {1, 2, 0} { 2, 0, 1} {2, 1, 0}

Dla niewielu elementów określenie wszystkich permutacji jest proste, jednak ile jest możliwych tablic i jak one wyglądają, gdy sposobów kodowania jest 9? Określenie, ile jest wszystkich możliwości, czyli ilości permutacji bez powtórzeń sprowadza się do wyznaczenia wartości funkcji silnia (ang. factorial). Przypomnę jej matematyczną definicję:

$$P_n = n!$$

$$n! = \begin{cases} 1 & n = 0 \\ n*(n-1)! & n > 0 \end{cases}$$

$$n! = \prod_{k=1}^{n} k$$

Jak widzimy funkcję silnia możemy zdefiniować na dwa sposoby: rekurencyjny i iteracyjny. Mimo, że implementacja obu definicji wydaje się prosta, należy zwrócić uwagę na kilka szczegółów. Podstawowym ograniczeniem, którego trzeba być świadomym jest maksymalna wartość typu zmiennej, którą wykorzystamy w obliczeniach. Przepełnienie zmiennej można łatwo przeoczyć. Korzystając z poniższej iteracyjnej implementacji dla n = 25 otrzymamy wynik 2076180480. Proszę zerknąć do kodu następnej proponowanej implementacji by sprawdzić czy wartość ta jest poprawna. Problemem dodatkowym przy rekurencyjnej implementacji jest dość trudne do przewidzenia zachowanie systemu przy zagnieżdżonym wywoływaniu funkcji dla dużych n.

```
public static int factorial(int m){
    int ans = 1;
    for (int i = m; i >= 1; --i){
        ans = ans * i;
    }
    return ans;
```

```
}
```

Przy małych wartościach n, rozsądnym i zarazem najbardziej efektywnym rozwiązaniem okazuje się proste wpisanie wartości do tablicy:

```
private static long[] factorials = new long[] {

    1, 1, 2, 6, 24,

    120, 720, 5040, 40320, 362880,

    3628800, 39916800, 479001600, 6227020800L, 87178291200L,

    1307674368000L, 20922789888000L, 355687428096000L, 6402373705728000L,
    121645100408832000L,

    2432902008176640000L };


public static long factorial(int n) throws IllegalArgumentException{

    return factorials[n];

}
```

Jak poradzić sobie z problemem generowania wszystkich możliwych permutacji dla większych zbiorów? Pokażę dwa spośród wielu możliwych algorytmów.

Generowanie permutacji w porządku leksykograficznym

Porządek leksykograficzny jest najprostszym i najbardziej intuicyjnym sposobem, w jakim możemy przedstawić kolejno wszystkie permutacje. Układając ciągi w porządku leksykograficznym analizujemy ich kolejne elementy. Większy jest ten ciąg, którego pierwszy nierówny element jest większy. Traktując kolejne permutacje jako kilkucyfrowe liczby poukładamy je więc w porządku rosnącym. Właśnie w takim porządku przedstawione były permutacje w przykładzie 1.

W przedstawionym poniżej algorytmie kolejno będziemy generować permutacje. Pierwszą (najmniejszą) permutację, składającą się z kolejnych cyfr od 0 do n uszeregowanych rosnąco wygenerujemy w konstruktorze:

```
public Permutation(int order){

    element = new int[order];

    for (int i = 0; i< order;i++)
```

```
        element[i] = i;
    this.order = order;
}

private Permutation(int[] element) {
    this.element = new int[element.length];
    System.arraycopy(element, 0, this.element, 0, element.length);
    order = element.length;
}

public Permutation successor() {

    Permutation result = new Permutation(this.element);
    int left, right;

    left = result.order - 2;
    while ((result.element[left] > (result.element[left + 1])) && (left >= 1)){
        --left;
    }

    if ((left == 0) && (this.element[left] > (this.element[left + 1]))) {
        return null;
    }

    right = result.order - 1;
    while (result.element[left] > (result.element[right])){
        --right;
    }

    int temp = result.element[left];
    result.element[left] = result.element[right];
    result.element[right] = temp;

    int i = left + 1;
    int j = result.order - 1;
```

```
        while (i < j){
            temp = result.element[i];
            result.element[i++] = result.element[j];
            result.element[j--] = temp;
        }

        return result;
    }
```

W celu wygenerowania następnej permutacji modyfikujemy ostatnio wygenerowany ciąg w następujący sposób; znajdujemy lewy indeks przeszukując tablice od przedostatniego prawego elementu i zmniejszamy indeks do momentu, aż element po prawo (element o indeksie większym o 1) jest mniejszy od elementu pod aktualnym indeksem. W przykładzie poniżej będziemy poruszać się aż do elementu o indeksie 1. Prawy indeks znajdziemy rozpoczynając poszukiwania od skrajnie prawego elementu. Tym razem zmniejszamy indeks do momentu, w którym element pod aktualnym indeksem będzie większy od elementu pod indeksem lewym. W tym przypadku będzie to indeks 3. Następnie zamieniamy miejscami elementy pod lewym i prawym indeksem i odwracamy kolejność elementów na prawo od indeksu lewego.

Przykład 2

Krok 1 { 1, 3, 5, 4, 2, 0 } => lewy indeks = 1, prawy indeks = 3

Krok 2 { 1, 4, 5, 3, 2, 0 } => zamiana miejscami elementów

Krok 3 { 1, 4, 0, 2, 3, 5 } => odwracamy kolejność elementów na prawo od indeksu 1

Podany algorytm ma pewną wadę - permutacje musimy generować kolejno. Przy ich dużej liczbie może to być operacja czasochłonna i często niepotrzebna. Możemy tego uniknąć korzystając z innego sposobu.

Generowanie permutacji wykorzystując system factoradic

Przetestowanie tylko niektórych, wybranych permutacji zapewni nam często wystarczającą jakość. Istnieje kilka strategii, jeśli chodzi o wybór przypadków do testowania. Dość skuteczne jest testowanie losowe. Należy przy tym jednak pamiętać by losowo wygenerowane przypadki znacząco się od siebie różniły.

W celu generowania losowych permutacji możemy wykorzystać właściwości zapisu liczb naturalnych w systemie factoradic. W systemie dziesiętnym podstawą pozycji są kolejne potęgi liczby tzn.

wartości kolejnych pozycjach mnożymy przez liczbę 10 podniesionej do kolejnej potęgi. System factoradic to system, w którym mnożymy wartości kolejnych pozycji przez wartości funkcji silnia dla kolejnych liczb naturalnych. Ilustruje to poniższy przykład.

Przykład 3

System dziesiętny

$1 * 10^2 + 9 * 10^1 + 9 * 10^0 = 1 * 100 + 9 * 10 + 9 * 1 = 199$

System factoradic

$1 * 5! + 2 * 4! + 3 * 3! + 4 * 2! + 5 * 1! = 1 * 120 + 2 * 24 + 3 * 6 + 4 * 2 + 5 * 1 = 199$

Wykorzystamy jednoznaczne odwzorowanie liczby zapisanej w systemie factoradic na odpowiadającą permutację. Oznacza to, że liczbę zapisaną w systemie factoradic będziemy zamieniać na odpowiadającą jej dokładnie jedna permutację.

Przykład 4

indeks	zapis factoradic	odpowiadająca permutacja
0	{ 0, 0, 0 }	{ 0, 1, 2 }
1	{ 0, 1, 0 }	{ 0, 2, 1 }
2	{ 1, 0, 0 }	{ 1, 0, 2 }
3	{ 1, 1, 0 }	{ 1, 2, 0 }
4	{ 2, 0, 0 }	{ 2, 0, 1 }
5	{ 2, 1, 0 }	{ 2, 1, 0 }

Do zamiany wykorzystamy następującą procedurę: początkowo do liczby zapisanej w systemie factoradic dopisujemy na końcu 0 i kopiujemy ją do pomocniczej tablicy. Następnie zwiększamy każdy element tej tablicy. Do tablicy wynikowej wpisujemy jako ostatni element wartość 1. Rozpoczynając od elementu o przedostatnim indeksie kopiujemy element z tablicy pomocniczej do tablicy

wynikowej zwiększając o 1 wszystkie elementy, które są większe lub równe skopiowanemu elementowi. W ten sposób postępujemy kopiując kolejne elementy o coraz mniejszych indeksach. Finalnie, po skopiowaniu wszystkich pól zmniejszamy wartości elementów w wynikowej tablicy o 1.

```
public Permutation getKPermutation(int k){

    int[] factoradic = new int[order];

    for (int j = 1; j <= order; ++j){
        factoradic[order-j] = k % j;
        k /= j;
    }

    int[] temp = new int[order];

    for (int i = 0; i < order; ++i){
        temp[i] = ++factoradic[i];
    }

    this.element[order-1] = 1;

    for (int i = order-2; i >= 0; --i){
        this.element[i] = temp[i];
        for (int j = i+1; j < order; ++j){
            if (this.element[j] >= this.element[i])
            ++this.element[j];
        }
    }

    for (int i = 0; i < order; ++i){
        --this.element[i];
    }
    return new Permutation(element);
```

```
}
```

Kombinacje

Zgodnie z definicją, kombinacją k-elementową zbioru elementowego A, nazywa się każdy k-elementowy podzbiór zbioru A (0≤k≤n). Istotną informacją jest to, że w kombinacji nie ma znaczenia porządek (kolejność) elementów. Często mamy z nią do czynienia w różnych grach. Grając w pokera nie ma znaczenia, w jakiej kolejności ułożymy karty, ważne jest natomiast, jakie są to karty.

Gdyby w naszej przykładowej funkcji kolejność elementów nie miała znaczenia, musielibyśmy przetestować tylko jedną kombinację (istnieje tylko jedna kombinacja 3-elementowa z 3 - elementowego zbioru). Jaki wymiar miałoby zwiększenie liczby sposobów kodowania do 5 przy stałym rozmiarze tablicy wejściowej? Ilość możliwości, jakie powinniśmy wówczas przetestować wynosi 10 i możemy je obliczyć korzystając z poniższych wzorów:

$$C_n^k = \frac{n!}{k!*(n-k)!}$$

$$C_n^k = \frac{n*(n-1)*(n-2)*...*(n-k+1)}{k!}$$

$$C_n^k = C_n^{n-k}$$

Wykorzystując drugi wzór nie będziemy musieli wyliczać wartości funkcji silnia dla dużych argumentów i zmniejszymy wartość mianownika. Trzeciego wzoru warto użyć kiedy n-k<k.

```
public static long c(long n, long k) {

    if (n < k) return 0;
    if (n == k) return 1;

    long delta, iMax;

    if (k < n-k){
        delta = n-k;
        iMax = k;
    }else {
        delta = k;
```

```
            iMax = n-k;
        }

        long ans = delta + 1;

        for (long i = 2; i <= iMax; ++i){
            ans = (ans * (delta + i)) / i;
        }

        return ans;
    }
```

Chcąc poznać liczbę kombinacji możemy również wykorzystać trójkąt Pascala i jego właściwości.

Pozostaje jeszcze problem generowania kombinacji. Najlepiej robić to w znanym nam porządku leksykograficznym. Tym razem również będziemy modyfikować ostatnio wygenerowaną kombinację. Zaczynamy od ostatniego jej elementu i poszukujemy elementu równego n-k poruszając się w lewo. Następnie zwiększamy o 1 znaleziony element i elementy o indeksach wyższych.

```
    public Combination successor(){

        if (this.element.length == 0 ||
            this.element[0] == this.n - this.k)
            return null;

        Combination ans = new Combination(this.n, this.k);

        int i;
        for (i = 0; i < this.k; ++i)
            ans.element[i] = this.element[i];

        for (i = this.k - 1; i > 0 && ans.element[i] == this.n - this.k + i; --i) ;

        ++ans.element[i];
```

```
        for (int j = i; j < this.k - 1; ++j)
            ans.element[j+1] = ans.element[j] + 1;

        return ans;
    }
```

Wariacje

Znając już pojęcie permutacji i kombinacji możemy przejść do wariacji. Wariacją bez powtórzeń k-wyrazową zbioru n-elementowego A nazywamy każdy k-wyrazowy ciąg k różnych elementów tego zbioru. Zgodnie z definicją mamy tu do czynienia z ciągami, istotna jest więc kolejność elementów. Zwróćmy uwagę, że gdy k=n, wariację bez powtórzeń nazywa się permutacją.

Przy wyliczaniu liczby wariacji możemy skorzystać z poniższych wzorów i tu również lepiej wykorzystać drugi wzór z powodów opisanych powyżej:

$$V_n^k = \frac{n!}{(n-k)!}$$
$$V_n^k = n*(n-1)*...*(n-k)$$

Wracając do pierwszej wersji naszej przykładowej funkcji przy 5 sposobach kodowania i 3 elementowej tablicy liczba wszystkich możliwości wynosiłaby:

$$V_5^3 = \frac{5!}{(5-3)!} = 5*4*3 = 60$$

Przy generowaniu wariacji możemy wykorzystać ciekawą zależność:

$$V_n^k = C_n^k \cdot P_k$$

Oznacza to, że w pierwszym etapie możemy wygenerować kombinację i znaleźć dla niej wszystkie możliwe permutacje. Implementacja z wykorzystaniem przedstawionych wyżej funkcji jest już prosta i pozostawię ją czytelnikowi.

Powtórzenia elementów

Do tej pory zakładaliśmy, iż elementy w generowanych ciągach nie będą się powtarzać. Co zrobić gdy nasz system dopuszcza taką możliwość? Generowanie wariacji z powtórzeniami nie przysparza kłopotów. Kolejne przypadki generować można w zagnieżdżonych pętlach przebiegających po wszystkich elementach zbiorów.

```
for (int i; i < n; i++){
    ...
    for (int j; j < n; j++){
        element[i] = i;
        ...
        element[j] = j;
    }
}
```

Ilość możliwości, jakich możemy się spodziewać przedstawiona jest poniżej i jest to największa liczba spośród prezentowanych.

$$\overline{V}_n^k = n^k$$

Ograniczenia jakie posiada nasz system mogą spowodować zmniejszenie liczby możliwości. Można w takim wypadku zastosować zabieg generujący interesujący nas zbiór. Gdybyśmy dopuścili możliwość, by kodowanie 16kbps mogło wystąpić w tablicy parametrów wejściowych naszej przykładowej funkcji dwa razy, należy dla potrzeb generowania przypadków przypisać mu drugi, pomocniczy identyfikator np. 3. W ten sposób zwiększamy nasz zbiór elementów i generujemy permutacje bądź kombinacje jak dla przypadku bez powtórzeń. Po wygenerowaniu zbioru pozostaje zamiana pomocniczego identyfikatora na właściwy. Należy pamiętać, że dopuszczając powtórzenia zwiększa się liczba możliwości, która wynosi dla permutacji i kombinacji odpowiednio:

$$\overline{P}_n = \frac{n!}{n_1!*n_2!*...*n_k!}$$

gdzie $n = n_1 + n_2 + ... + n_k$ i n_k oznacza ilość wystąpień k-tego elementu zbioru.

$$\overline{C}_n^k = \frac{(k+n-1)!}{k!*(n-1)!}$$

Jeśli użyjemy przedstawionego powyżej sposobu, otrzymamy niestety większą liczbę ciągów. Uważam jednak, iż prostota rozwiązania rekompensuje tą niedogodność i w większości przypadków może być on stosowany.

Podsumowanie

Testowanie oprogramowania jest dziedziną, która opiera się formalizacji i bardzo trudno znaleźć w niej uniwersalne rozwiązania. Mam nadzieję, że przedstawione informacje pomogą osobom rozpoczynającym przygodę z testowaniem oprogramowania w wypracowaniu własnych, dostosowanych do indywidualnych potrzeb, metod.

W codziennej praktyce testera problemy związane z testowaniem oprogramowania są znacznie bardziej skomplikowane niż te przedstawione w artykule. Często jednak wykorzystanie takich metod, jak podział na klasy równoważności, pozwala sprowadzić je do postaci, w której można wykorzystać prezentowane sposoby. Mogą one też posłużyć do wstępnego oszacowania liczby przypadków testowych i stworzenia ich bazy. Podobnie bardziej skomplikowane struktury jak grafy czy tablice wielowymiarowe można uprościć do postaci tablic jednowymiarowych.