Nekaj algoritmov za generiranje permutacij

↓↓↓
Aleksander Vesel

→ **Denimo, da imamo številčno ključavnico s tremi številkami. Koliko gesel moramo preizkusiti, če smo geslo pozabili, a vemo, da je sestavljeno iz številk 1, 2 in 3? (Vseh gesel je seveda šest: 123, 132, 213, 231, 312, 321.)**

Zgornja naloga je preprost primer razvrščanja elementov neke množice v vsa možna zaporedja. Zanimajo nas torej urejene izbire vseh elementov (pri čemer ponavljanje elementov ni dovoljeno), neko tako izbiro pa imenujemo *permutacija*. Število permutacij v množici z n elementi je enako $n! = n \cdot (n-1) \cdot \ldots \cdot 2 \cdot 1$, pri čemer z zapisom $n!$ označimo *fakulteto* naravnega števila n. Opazimo lahko, da število permutacij zelo hitro narašča glede na število elementov v množici. V primeru pozabljene PIN številke, ob predpostavki, da poznamo vse štiri številke, ne pa tudi njihovega vrstnega reda, je tako potrebno preizkusiti že $4! = 24$ različnih gesel. Število permutacij za množice z največ 10 elementi je prikazano v tabeli 1.

V tem prispevku nas bodo zanimali algoritmi za konstruiranje vseh permutacij množice z n elementi. Brez izgube splošnosti bomo pri tem privzeli, da permutiramo množico prvih n naravnih števil $\{1, 2, \ldots, n\}$. Za vsako konstruirano permutacijo bomo izvedli algoritem *obišči permutacijo*, ki predstavlja poljubno operacijo (na primer izpis) nad permutacijo. V nadaljevanju bomo uporabili tudi algoritem *zamenjaj*(x, y), ki zamenja vrednosti spremenljivk x in y.

Osnovni rekurzivni algoritem

V izrazu za število permutacij množice z n elementi

- $n! = n \cdot (n-1) \cdot \ldots \cdot 2 \cdot 1$

opazimo, da ga lahko za $n > 0$ zapišemo kot število permutacij množice z $n-1$ elementi pomnoženi z n oziroma

- $n! = n \cdot (n-1)!$. (1)

Pri tem velja $0! = 1$. Za podajanje vrednosti fakultete naravnega števila n smo torej uporabili fakulteto števila $n-1$. Izraz (1) je zato primer *rekurzivne formule*. Rekurzija je močno orodje tudi pri razvoju algoritmov. Pravimo, da je algoritem *rekurziven*, če kliče samega sebe.

Ker smo število permutacij izrazili z rekurzivno formulo, lahko intuitivno pričakujemo, da bo možno rekurzijo uporabiti tudi pri konstrukciji permutacij. Rekurzivna zveza $n! = n \cdot (n-1)!$ pomeni, da lahko permutacije množice $\{1, 2, \ldots, n\}$ pridobimo iz permutacij množice $\{1, 2, \ldots, n-1\}$, pri čemer vsaki permutaciji množice $\{1, 2, \ldots, n-1\}$ dodamo element n na vsa možna mesta (teh je ravno n).

Poglejmo si primer permutacij množice $\{1, 2, 3\}$, ki jih skonstruiramo iz permutacij množice $\{1, 2\}$.

Elementa 1 in 2 lahko razvrstimo na dva načina: 12 in 21. Sedaj vsako od permutacij dopolnimo s številom 3. Za permutacijo 12 tako dobimo

- **3**12, 1**3**2, 12**3**,

za 21 pa

- **3**21, 2**3**1, 21**3**.

n	1	2	3	4	5	6	7	8	9	10
$n!$	1	2	6	24	120	720	5040	40320	362880	3628800

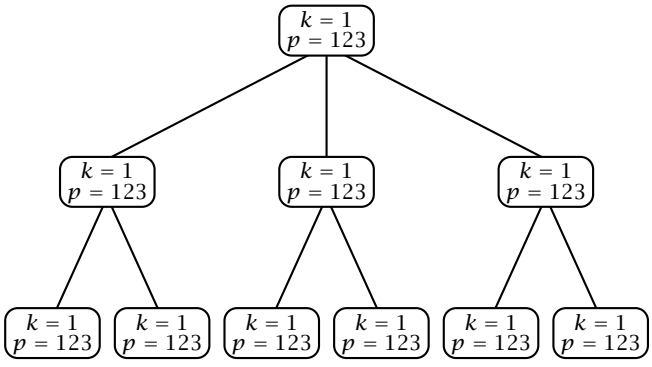
TABELA 1.
Število permutacij za množice velikosti največ 10

Algoritem 1: permutacije

Vhod: *Naravni števili n in k, zaporedje $p = (p_1, p_2, \ldots, p_n)$.*
Izhod: *Vse permutacije elementov zaporedja.*
begin
  if k=1 **then**
    obišči permutacijo(n, p)
  else
    for $i := 1$ **to** n **do**
      zamenjaj(p_i, p_k);
      permutacije($k-1$,n,p);
      zamenjaj(p_i, p_k);
    end
  end
end

Postopek se zdi enostaven, a je pri zapisu algoritma potrebno nekaj previdnosti. Predvsem se je potrebno izogniti nepotrebnemu shranjevanju permutacij, še posebej zato, ker njihovo število glede na vrednost n hitro narašča.

Postopek je predstavljen v Algoritmu 1 (permutacije). Algoritem vzame za osnovo poljubno permutacijo dolžine n, ki je predstavljena kot zaporedje $p = (p_1, p_2, \ldots, p_n)$. Pri prvem klicu algoritma je vrednost parametra k enaka n. Algoritem za vse vrednosti i med 1 in k zamenja i-ti in k-ti element zaporedja, rekurzivno pokliče samega sebe s parametrom $k-1$ ter nato spet zamenja i-ti in k-ti element zaporedja. Algoritem se zaključi z obiskom permutacije, ko k doseže vrednost ena.



SLIKA 1.
Delovanje algoritma permutacije za $n = 3$

Predstavimo delovanje Algoritma 1 s parametri $n = 3$, $k = 3$ in $p = (1, 2, 3)$ (glej sliko 1). Algoritem sproži tri rekurzivne klice za $k = 2$ ter $p = (3, 2, 1)$, $p = (1, 3, 2)$ ter $p = (1, 2, 3)$. Vsak od teh rekurzivnih klicev sproži še dva rekurzivna klica za $k = 1$, ob vsakem od teh rekurzivnih klicev algoritem obišče trenutno permutacijo shranjeno v p. Vrstni red obiskanih permutacij je: 231, 321, 312, 132, 213, 123.

Urejeno zaporedje permutacij

Včasih je zaželeno, da algoritem vrne urejeno zaporedje permutacij. Ko govorimo o urejenosti, ponavadi mislimo *leksikografsko urejenost*. Če gre za permutacije števk, je to kar običajna urejenost po velikosti, saj npr. permutacija $p = (1, 2, 3)$ na naraven način predstavlja število 123. Če pa so elementi množice črke, je leksikografska urejenost kar abecedna urejenost.

Predstavljeni algoritem tokrat ne bo rekurziven. Osnova algoritma je postopek, predstavljen v Algoritmu 2, ki za dano permutacijo p poišče naslednjo permutacijo v leksikografski ureditvi.

Algoritem najprej poišče najbolj desni element zaporedja, ki je manjši od svojega desnega soseda in ga označi z indeksom k. Če takšen element ne obstaja, je permutacija p največja, zato naslednja permutacija ne obstaja. V tem primeru dobi k vrednost 0 in algoritem se zaključi z vrednostjo *obstaja* = *false*. Če je $k > 0$, algoritem poišče najbolj desni element zaporedja, ki je večji od p_k, ter ju zamenja. Elemente z indeksi od $k+1$ do n nato preuredi tako, da predstavljajo najmanjšo možno vrednost. Algoritem se zaključi z vrednostjo *obstaja* = *true*.

Kot primer si poglejmo potek algoritma za permutacijo $p = (3, 4, 2, 1)$. Algoritem najprej ugotovi, da se najbolj desni element zaporedja p, ki je manjši od svojega desnega soseda, nahaja na prvem mestu, zato je $k = 1$ in $p_k = 3$. Najbolj desni element zaporedja p, ki je večji od 3, se nahaja na drugem mestu, zato je $j = 2$ in $p_j = 4$. Algoritem zamenja elementa 3 in 4, nato pa preuredi elemente podzaporedja (3,2,1) v (1,2,3). Permutacija, ki jo vrne algoritem, je tako $p = (4, 1, 2, 3)$.

Algoritem 3, ki poišče vse permutacije v urejenem vrstnem redu, generiranje permutacij začne z najmanjšo permutacijo $p = (1, 2, \ldots, n)$. Permutacijo p v zanki spreminja tako dolgo, dokler obstaja večja permutacija.

Algoritem 2: naslednja

Vhod: *Naravno število n, zaporedje $p = (p_1, p_2, \ldots, p_n)$.*
Izhod: *Boolova vrednost $obstaja$, nova vrednost p.*
begin
  $k := n - 1$;
  while $p_k > p_{k+1}$ **do**
    $k := k - 1$;
  end
  if $k = 0$ **then**
    $obstaja := false$
  else
    $obstaja := true$;
    $j := n$;
    while $p_k > p_j$ **do**
      $j := j - 1$;
    end
    zamenjaj(p_k,p_j);
    $r := n$; $s := k + 1$;
    while $r > s$ **do**
      zamenjaj(p_r,p_s);
      $r := r - 1$; $s := s - 1$;
    end
  end
end

Algoritem 3: leksikografsko

Vhod: *Naravno število n.*
Izhod: *Zaporedje $p = (p_1, p_2, \ldots, p_n)$.*
begin
  $p := (1, 2, \ldots, n)$;
  $obstaja := true$;
  obišči permutacijo(n, p);
  while $obstaja$ **do**
    naslednja(p,n,$obstaja$);
    obišči permutacijo(n, p);
  end
end

Hitro generiranje permutacij

Generiranje permutacij pogosto uporabljamo pri reševanju številnih pomembnih kombinatoričnih problemov. Zelo znan je *problem trgovskega potnika*, ki je bil v Preseku že opisan. Ponovimo na kratko definicijo problema. Dana je množica mest $C = \{c_1, c_2, \ldots, c_n\}$. Za vsak par mest c_i, c_j je znana cena povezave od mesta c_i do mesta c_j, ki jo označimo z $d_{i,j}$. Trgovski potnik mora začeti pot v enem od mest, obiskati vsa preostala mesta s seznama ter se vrniti v izhodišče tako, da bo skupna cena poti čim manjša. Poiskati torej želimo takšno zaporedje mest $(c_{\pi_1}, c_{\pi_2}, \ldots, c_{\pi_n})$ iz C, da bo vrednost izraza $d_{\pi_1,\pi_2} + d_{\pi_2,\pi_3} + \ldots + d_{\pi_{n-1},\pi_n} + d_{\pi_n,\pi_1}$ najmanjša možna.

Problem trgovskega potnika spada med probleme, ki jih ne znamo rešiti s hitrimi algoritmi, torej z algoritmi, ki bi omogočali izračun rešitve v sprejemljivem času tudi za večje število vhodnih podatkov. Osnova reševanja problema je tako postopek, s katerim poiščemo vse permutacije množice C in izračunamo ceno pripadajočega krožnega obhoda.

Kot smo že povedali, število permutacij zelo hitro narašča glede na število elementov v množici, zato je problem rešljiv le za primere, ko je množica mest razmeroma majhna. Zato je zelo pomebno, da je delovanje algoritma kolikor je le mogoče hitro, kar pa je v precejšnji meri odvisno tudi od hitrosti generiranja permutacij.

Prvi predstavljeni algoritem, algoritem permutacije, je res enostaven, a ne spada med najhitrejše. Nova permutacija je generirana ob klicu algoritma za vrednost parametra $k = 1$, pred in po klicu algoritma pa se izvede algoritem zamenjaj. Število zamenjav, ki jih izvede algoritem, je zato vsaj dvakratnik števila generiranih permutacij. Algoritem, ki generira leksikografsko urejene permutacijem je nekoliko hitrejši, a ne bistveno. Med najhitrejše pa spada Heapov algoritem (glej Algoritem 4).

Spet je osnova poljubna permutacija $p = (p_1, p_2, \ldots, p_n)$. Pri prvem klicu algoritma je vrednost parametra k enaka n. Algoritem za vse vrednosti i med 1 in k rekurzivno pokliče samega sebe s parametrom $k-1$ ter nato zamenja k-ti element zaporedja bodisi s prvim, če je k sod, bodisi z i-tim, če k lih. Algoritem se zaključi z obiskom permutacije, ko k doseže vrednost ena.

Algoritem deluje podobno kot algoritem pemutacije, le da se ob vsakem rekurzivnem klicu izvede samo ena zamenjava vrednosti. Skupno število zamenjav je tako približno enako številu generiranih permutacij. Predstavimo delovanje Algoritma 4 za parametre $n = 3$, $k = 3$ in $p = (1, 2, 3)$. Ker je k lih,

Algoritem 4: Heap

Vhod: *Naravni števili n in k, zaporedje $p = (p_1, p_2, \ldots, p_n)$.*
Izhod: *Vse permutacije elementov zaporedja.*

```
begin
    if k=1 then
        obišči permutacijo(n, p)
    else
        for i := 1 to n do
            heap(k − 1,n,p);
            if k je sod then
                zamenjaj(p_i, p_k);
            else
                zamenjaj(p_1, p_k);
            end
        end
    end
end
```

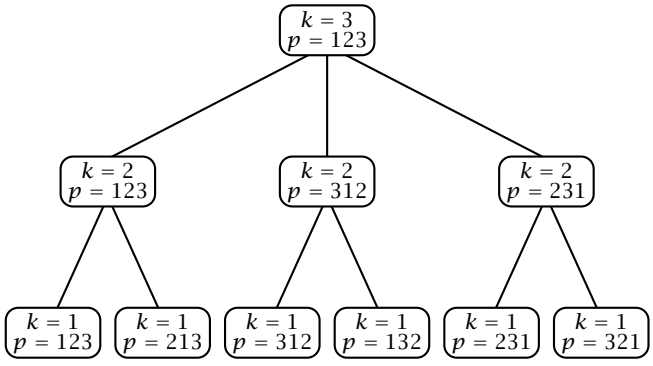
algoritem sproži tri rekurzivne klice za $k = 2$ ter po vsakem klicu zamenja prvi in zadnji element zaporedja. Vsak od teh rekurzivnih klicev sproži še dva rekurzivna klica za $k = 1$ ter po vsakem klicu zamenja drugi in i-ti element zaporedja. Delovanje algoritma za opisani primer je predstavljeno na sliki 2.

Za konec si v tabeli 2 poglejmo primerjavo časa izvajanja vseh predstavljenih algoritmov, realiziranih v programskem jeziku C++. Primerjava kaže, da je algoritem Heap skoraj dvakrat hitrejši od algoritma permutacije, medtem ko je algoritem leksikografsko nekje na sredini med njima. V vseh primerjanih programih je algoritem obišči permutacijo izpuščen.

Omenimo za konec še to, da bi bila algoritma permutacija in Heap nekoliko hitrejša, če bi ju zapisali nerekurzivno.

algoritem	čas (sekunde)
permutacije	10,107
leksikografsko	8,726
Heap	5,348

TABELA 2.
Čas izvajanja algoritmov za permutacije z 12 elementi



SLIKA 2.
Delovanje algoritma Heap za $n = 3$.

Literatura

[1] S. B. Maurer in A. Ralston, *Discrete Algorithmic Mathematics*, A K Peters/CRC Press, 2005.

[2] S. Pemmaraju in S. Skiena, *Computational Discrete Mathematics: Combinatorics and Graph Theory with Mathematica*, Cambridge University Press, 2009.

× × ×

Križne vsote

REŠITEV S STRANI 29

↓ ↓ ↓

	12	10			
16	7	9	11		
9	5	1	3	7	
		10	8	2	13
			11	4	7
			7	1	6

× × ×