НАЛАЖЕЊЕ ПОДСКУПОВА БЕКТРЕКОМ

др Ђура Паунић, Природно-математички факултет, Нови Сад

Један од често коришћених комбинаторних објеката су комбинације без понављања. Опишимо један поступак за генерисање свих комбинација k-те класе од n елемената тако да га је лако испрограмирати.

Ако се са S означи скуп од n елемената од кога треба да направимо комбинације k-те класе, тада су комбинације без понављања од n елемената k-те класе у ствари онај подскуп T партитивног скупа $\mathcal{P}(S)$, који се састоји од свих k-елементних подскупова скупа S. Ради једноставности претпоставимо да је скуп $S = \{1, 2, \ldots, n\}$ и нека је једна комбинација подскуп $\{a_1, a_2, \ldots, a_k\}$. Ову комбинацију ћемо писати једноставно $a_1a_2 \ldots a_k$ и претпоставићемо да је $a_1 < a_2 < \ldots < a_k$, а овако уређене елементе a_i називати компоненте. Ако су $a_1a_2 \ldots a_k$ и $b_1b_2 \ldots b_k$ две комбинације, тада ћемо рећи да је $a_1a_2 \ldots a_k < b_1b_2 \ldots b_k$ ако и само ако је $a_1 < b_1$ или је за неко $j, 1 \leq j \leq k$, задовољено $a_i = b_i$ за $i = 1, \ldots, j-1$ и $a_j < b_j$. Овакво уређење се назива лексикографско.

Ако се комбинације лексикографски уреде тада се на пример за $n = 6$ и $k = 3$ добија следећи уређени низ комбинација:

$$123 < 124 < 125 < 126 < 134 < 135 < 136 < 145 < 146 < 156 <$$

$$< 234 < 235 < 236 < 245 < 246 < 256 < 345 < 346 < 356 < 456.$$

При оваквом уређењу свака следећа комбинација се добија тако што се повећава последња компонента која може да се повећа. Очигледно је да се последња компоненета може повећавати до n, претпоследња до $n-1$ итд. прва до $n-k+1$.

Дакле, поступак генерисања лексикографског низа комбинација је следећи:

За сваку компоненту узимамо најпре најмањи могући елемент, тако да је најмања, почетна комбинација $12 \ldots k$, а следеће комбинације се добијају тако да се последња компонента $a_k = k$ повећава један по један до n. Затим се $a_{k-1} = k-1$ повећа за један. Добија се $a_{k-1} = k$, а последња компонента стави да је $a_k = k+1$, за један већа од a_{k-1}, јер је свака комбинација растући низ бројева. Затим се опет повећава последња компонента док је то могуће итд.

Генерисање комбинација овим поступком могуће је испрограмирати и итеративно и рекурзивно. Напишимо итеративну процедуру у Pascalu која реализује овај поступак и коју је лако могуће превести у било који програмски језик. Претпоставимо да су компоненте елементи низа тј. да је декларисано:

```
const
  maxn = 20;
type
  niz = array[0 .. maxn] of integer;
```

При том је узето да низ почиње од 0 да би се у неким случајевима избегла провера дефинисаности индекса низа при његовом смањивању.

Процедура има три параметра: n, величину скупа, $1 \leq n \leq maxn$, k величину подскупа, $1 \leq k \leq n$, а логичка променљива ok служи за проверу да ли су параметри n и k у дозвољеном опсегу. Коначно се добија процедура.

```
procedure iterkombinacije(n, k : integer;
                          var ok : boolean);
var
  a : niz;
  i, j : integer;
begin
  if (1 <= n) and (k <= maxn) and (1 <= k) and (k <= n) then
    begin
      ok := true;
      i := 1;
      a[1] := 0;
      while i > 0 do
        begin
          while a[i] < n - k + i do
            begin
              a[i] := a[i] + 1;
              if i < k then
                begin
                  a[i+1] := a[i];
                  i := i + 1
                end
              else
                begin
                  for j := 1 to k do
                    write(a[j]:4);
                  writeln
                end
            end;
          i := i - 1
        end
    end
  else
    ok := false
end;    (*  iterkombinacije  *)
```

Рекурзиван поступак је сличан. При том треба нагласити да се рекурзивне процедуре најједноставније реализују тако да се у процедури иницијализују и провере почетне вредности променљивих и по потреби израчуна неки специјални случај који се не уклапа у општи поступак, а направи се унутрашња рекурзивна процедура или функција у којој се изводи само израчунавање. У овом случају је унутрашња процедура процедура `nadji`.

```
procedure rekkombinacije(n, k : integer
                         var ok : integer);
var
  a : niz;
```

```
  i : integer;

  procedure nadji(i : integer);

  var
    j : integer;
  begin
    if i <= k then
      begin
        a[i] := a[i-1];
        while a[i] < n - k + i do
          begin
            a[i] := a[i] + 1;
            nadji(i+1)
          end
      end
    else
      begin
        for j := 1 to k do
          write(a[j]:4);
        writeln
      end
  end;  (*  nadji  *)

begin
  if (1 <= n) and (n <= maxn) and (1 <= k) and (k <= n) then
    begin
      ok := true;
      a[0] := 0;
      nadji(1)
    end
  else
    ok := false;
end;   (*  rekkombinacije  *)
```

Овим поступком је могуће генерисати и комбинације са понављањем, јер се комбинације са понављањем лако добијају од комбинација без понављања. Свака комбинација са понаваљањем од n елемената k-те класе се прави од комбинације без понављања од $n+k-1$ елемента тако што се од комбинације без понављања одузму редом бројеви од 0 до $k-1$. Дакле за генерисање комбинација са понављањем може да се користи поступак као у процедури за комбинације без поначања, али од $n+k-1$ елемента k-те класе, а при испису комбинације уместо `a[j]` штампати `a[j] - j + 1` за $j = 0, 1, \ldots, k-1$. Наиме лако се доказује да се оваквим одузимањем различитим комбинацијама без понављања од $n+k-1$-ног елемента k-те класе придружују различите комбинације са понављањем од n елемената k-те класе и да се свака комбинација са понављањем добија као слика једне комбинације без понављања тј. да је описана конструкција бијекција.

Аналогно као што се генеришу комбинације, могу да се генеришу и варијације са понављањем. Ако је $\{1, 2, \dots, n\}$ скуп над којим се генеришу варијације са понављањем, тада је n горња граница за сваку компоненту, а почетна варијација је $11\dots1$. Следеће варијације се добијају, као и код комбинација, повећавањем последње компоненте до n, а затим се повећа претпоследња компонента за 1, а последња се стави да је 1 и настави се са повећававањем последње компоненте за по 1. У j-том кораку се повећава за један прва компонента гледано са десне стране, која може да се повећа, а све компоненте десно од повећане компоненте се стављају на 1.

Интересантно је да се варијације са понављањем доста често јављају у програмирању. Наиме генерисање свих варијација са понављањем k-те класе је сличан проблему генерисању свих индекса k петљи које се налазе једна у другој, тј. за задат променљив број k треба написати програмски фрагмент следећег типа

```
for i1 := dgr[1] to ggr[1] do
  for i2 := dgr[2] to ggr[2] do
    . . . . . . . . . . . . .

      for ik := dgr[k] to ggr[k] do
        obrada(k, i1, i2, ..., ik);
```

у коме се појављује k петљи. Решимо овај проблем када је k произвољан број из неког интервала, а индекси су елементи низа `indeks` и мењају се у интервалима од доње до горње границе које су задате у низовима `dgr` и `ggr`.

Наведимо итеративну верзију процедуре која генерише све индексе за k петљи, а за илустрацију узмимо да процедура `obrada` само штампа вредности индекса:

```
procedure obrada(var k : integer;
                 var indeks : niz);
var
  j : integer;
begin
  for j := 1 to k do
    write(indeks[j]:4);
  writeln
end;  (*  obrada  *)

procedure indeksit(k : integer;
                   var dgr, ggr, indeks : niz);
var
  i : integer;
begin
  i := 1;
  indeks[1] := dgr[1] - 1;
  while i > 0 do
    begin
      while indeks[i] < ggr[i] do
        begin
```

```
          indeks[i] := indeks[i] + 1;
          if i < k then
            begin
              i := i + 1;
              indeks[i] := dgr[i] - 1
            end
          else
            obrada(k, indeks)
        end;
      i := i - 1
    end
end;   (*  indeksit  *)
```

Сада лако може да се напише процедура која генерише све варијације са понављањем од n елемената k-те класе, што се оставља за вежбу.

Сви ови примери су специјални случајеви следећег општег проблема:

Назовимо решење n-торку $(a_1, a_2, \ldots)$, коначне, али неодређене дужине, при чему компоненте a_i припадују подскуповима T_i коначних скупова S_i, $a_i \in T_i \subset S_i$, који за сваку компоненту могу бити различити. У скупу S, унији коначних Декартових производа скупова S_i,

$$S = S_1 \bigcup S_1 \times S_2 \bigcup S_1 \times S_2 \times S_3 \bigcup \cdots = \bigcup_{i=1}^{\infty} \prod_{k=1}^{i} S_k,$$

треба наћи подскуп T, скуп решења – бар једног или свих, такав да за сваки елемент скупа T важи да свака компонента зависи само од претходних компонената.

За решавање овог проблема се користи општи алгоритам који се назива бектрек или претраживање са враћањем (енг. backtrack), чији су неки специјални случајеви коришћени у претходним процедурама. Бектрек се састоји у следећем:

На основу постојећих ограничења израчуна се скуп T_1, допустивих елемената за прву компоненту, $T_1 \subseteq S_1$. У скупу T_1 се бира елемент a_1 и изабрани елемент a_1 се избацује из T_1, T_1 постаје $T_1 \setminus \{a_1\}$, и формира се делимично решење (a_1). Затим се проверава да ли је добијени низ (a_1) решење. Ако јесте тада је готово – решење се штампа, а ако није тада се на основу делимичног решења (a_1) из скупа свих могућих елемената за другу компоненту S_2 одређује подскуп T_2, скуп допустивих елемената за другу компоненту. Из скупа T_2 се бира елемент a_2, a_2 се избацује из скупа T_2, T_2 постаје $T_2 \setminus \{a_2\}$, и образује се делимично решење (a_1, a_2). Ако делимично решење (a_1, a_2) није решење прелази се на одређивање треће компоненте решења итд. На основу делимичног решења $(a_1, a_2, \ldots, a_i)$ у скупу S_{i+1} одређује се подскуп допустивих елемената T_{i+1} за $i+1$-ву компоненту делимичног решења. Из скупа T_{i+1} се бира $i+1$-компонента, a_{i+1}, и проверава се да ли је $(a_1, \ldots, a_i, a_{i+1})$ решење. Уколико је нађено решење тада се решење штампа. Поступак се наставља све док скуп T_{i+1} не постане празан, или се нађе неко решење. Ако у неком кораку скуп T_{i+1} постане празан тада се враћа на i-ту компоненту, одбацује се елемент a_i из делимичног решења и за i-ту компоненту се бира неки други елемент из скупа T_i.

Овај алгоритам је описан следећом псеудо процедуром у којој се користе процедуре: dopustiv, која за дати скуп S_i налази скуп допустивих елемената T_i за i-ту компоненту, print која штампа решење, izaberi која из допустивог скупа T_i бира један елемент a_i, izbaci која из скупа T_i избацује елемент a_i и логичких функција prazan, којом се проверава да ли је скуп празан и proveriresenje, којом се проверава да ли је низ елемената $(a_1, \ldots, a_i)$ решење. Претпоставићемо да је низ елемената a дат као тип niz, при чему тај тип може да буде стваран паскалски низ довољне дужине за решавање проблема или листа. Елементе низа а означаваћемо са a[i] иако се за тип низа не мора подразумевати array.

```
procedure backtrack;

var
  i : integer;
begin
  i := 1;
  dopustiv(T[1], S[1]); (*  nalazi podskup T1 od S1  *)
  while  i > 0  do
    begin
      while not prazan(T[i]) do
        begin
          izaberi(a[i], T[i]);
          izbaci(a[i], T[i]);
          if  proveriresenje(a) then
            print(a);
          i := i + 1;
          dopustiv(T[i], S[i])
        end;
      i := i - 1
    end
end;  (*  backtrack  *)
```

Бектрек је по својој природи рекурзиван поступак тако да се врло лако може формулисати и рекурзивна процедура, што се оставља за вежбу.

За бектрек алгоритам су врло честе следеће две варијације које се не искључују.

Прва модификација се састоји у томе да се уместо целих скупова допустивих елемената T_i користе најмањи елементи тих скупова. Наиме сваки коначан скуп може да буде линеарно уређен, а функција избора елемента a_i, скупа T_i, izaberi(a[i],T[i]), може да буде избор минималног елемента скупа T_i, $a_i = \min(T_i)$. У овом случају је могуће реализовати бектрек тако да се израчунава само најмањи елемент a_i скупа T_i. При том се најчешће за скупове S_i користе подскупови скупа целих бројева, јер је тада функцију избора лако испрограмирати. Дакле уместо процедура dopustiv(T[i],S[i]) и izaberi(a[i],T[i]) потребна је функција nadjimin којом се налази минималан допустив елемент S_i, па се тада претходне две наредбе реализују са a[i]:= nadjimin(S[i]). Када је елемент a_i изабран, тада се одређује следећи најмањи елемент b_i,

$$b_i = \min(T_i \setminus \{a_i\}) = \min(x \in S_i \mid x > a_i).$$

Често се користи модификација ове идеје тако да се користи функција

```
b[i] := nadjisledeci(a[i],S[i]),
```

која налази минималан допустив елемент скупа S_i који је већи од елемента a_i. Тада се минимални елемент добија када се за a_i узме нека погодна вредност (најчешће број мањи од минимума скупа $T_i \subseteq S_i$).

У случају када се бирају елементи тада за сваки скуп T_i постоји природно ограничење одгоре тј. скуп T_i постаје празан уколико следећи елемент, b_i, треба да буде већи од неке природне границе, `granica[i]`, за сваки скуп T_i. Тиме се низ скупова замењује низом елемената, за које се најчешће користе цели бројеви, што у неким програмским језицима значајно поједностављује реализацију програма.

```
procedure ebacktrack;

var
  i : integer;
begin
  i := 1;
  a[1] := nadjisledeci(- maxint, S[1]);
  while  i > 0  do
    begin
      while a[i] <= granica[i] do
        begin
          resenje[i] := a[i];
          a[i] := nadjisledeci(a[i],S[i]);
          if  proveriresenje(resenje) then
            print(resenje);
          i := i + 1;
          a[i] := nadjisledeci( - maxint,S[i])
        end;
      i := i - 1
    end
end;  (*  ebacktrack  *)
```

Други чест случај је аутоматска провера да ли је конструисани низ решење. У великом броју проблема се зна да је решење низ од тачно n допустивих елемената, $(a_1, \ldots, a_n)$, где је n фиксан број. Тада отпада процедура за проверу решења, јер се са $i = n$ једноставно проверава да ли је решење нађено. У овом случају бектрек процедура се модификује на следећи облик:

```
procedure nbacktrack;

var
  i : integer;
```

```
begin
  i := 1;
  dopustiv(T[1],S[1]);
  while  i > 0  do
    begin
      while not prazan(T[i]) do
        begin
          izaberi(a[i],T[i]);
          izbaci(a[i],T[i]);
          if  i < n then
            begin
              i := i + 1;
              dopustiv(T[i],S[i])
            end
          else
            print(a[1],...,a[n])
        end;
      i := i - 1
    end
end;  (*  nbacktrack  *)
```

Бектрек може успешно да се примени за решавање проблема 8 краљица на шаховској табли. Проблем се састоји у томе да се пронађе бар један распоред или сви распореди 8 краљица на шаховској табли, тако да нема две краљице које се узајамно нападају, тј. ни један пар краљица није у истој врсти, истој колони и на истој дијагонали.

Процедура `ndama` налази и штампа све могуће распореде дама за таблу $n \times n$ где је $3 < n < 20$. Ова процедура је добијена потребним модификацијама процедуре `ebektrek`, коришћена је и аутоматска провера решења из процедуре `nbektrek`. Унутрашња процедура `nadjipolje` налази прво слободно поље у колони `kolona` почев од поља `mesto` и оно се добија у променљивој `mesto`. Процедура `nadjipolje` одговара функцији `nadjisledeci` у процедури `ebektrek`. Неко поље није слободно ако већ постоји краљица на претходној вертикали i или у истој врсти, што се проверава са `resenje[i] = mesto` или на истој дијагонали што се проверава са `abs(resenje[i] - mesto) = abs(i - kolona)`. Граница величине скупа је увек иста n.

```
procedure ndama(var resenje : niz;
                n : integer);
var
  br, i, kolona : integer;
  t : niz;

  procedure nadjipolje(var n, kolona, mesto : integer;
                       var resenje : niz);
  var
    slobodno : boolean;
    i : integer;
```

```
  begin
    if mesto <= n then
      repeat
        slobodno := true;
        i := 1;
        while (i < kolona) and slobodno do
          if (resenje[i] = mesto) or
             (abs(resenje[i] - mesto) = abs(i - kolona)) then
            slobodno := false
          else
            i := i + 1;
        if (mesto <= n) and not slobodno then
          mesto := mesto + 1
      until slobodno or (mesto > n)
  end;  (*  nadjipolje  *)
begin
  if (3 < n) and (n < 20) then
    begin
      br := 0;
      t[1] := 1;
      kolona := 1;
      while kolona > 0 do
        begin
          while t[kolona] <= n do
            begin
              resenje[kolona] := t[kolona];
              t[kolona] := t[kolona] + 1;
              nadjipolje(n, kolona, t[kolona], resenje);
              if kolona < n then
                begin
                  kolona := kolona + 1;
                  t[kolona] := 1;
                  nadjipolje(n, kolona, t[kolona], resenje)
                end
              else
                begin
                  br := br + 1;
                  write('br ',br:3,'  ');
                  for i := 1 to n do
                    write(resenje[i]:3);
                  writeln
                end
            end;
          kolona := kolona - 1
        end
    end
  else
    writeln('broj n mora da bude u intervalu (3 < n < ',maxn)
end;  (*  ndama  *)
```

Статијата прв пат е објавена во списанието Тангента на ДМ на Србија во 1995/96 година