МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное учреждение
высшего профессионального образования
«Ижевский государственный технический университет
имени М.Т. Калашникова»

Кафедра «Программное обеспечение»

О.Л. Макарова

Дискретная математика

методические указания для выполнения лабораторной работы №3

«Генерирование комбинаторных объектов»

направление 231000 «Программная инженерия»

Ижевск, 2013 г.

МЕТОДИЧЕСКИЕ УКАЗАНИЯ
к выполнению лабораторных работ по курсу
«Дискретная математика»

Общие указания

Целью лабораторных работ является:

- закрепление теоретических знаний по дисциплине «Дискретная математика»;
- приобретение практических навыков по моделированию объектов, изучаемых в курсе, а также операций над ними.

Предлагаемые лабораторные работы проводятся со студентами всех форм обучения кафедры «Программное обеспечение» в ходе изучения курса «Дискретная математика». На первом лабораторном занятии студенты получают номер варианта на весь семестр, в соответствии с которым выполняют индивидуальные задания.

В ходе лабораторной работы студент должен ознакомиться с теоретическим материалом (см. список литературы [1-10]) и написать программу в соответствии с заданием *своего номера варианта*. За каждую выполненную работу студенту начисляется определенное количество баллов. Помимо лабораторных работ студент может выполнить дополнительные задания, за что получает соответственно *дополнительные баллы*.

Программа может быть написана на языке *Pascal* (по желанию студент может использовать языки *C, C++,* возможно использование среды программирования *Delphi*). Текст программы должен в *обязательном порядке содержать комментарии*.

Предварительно представить отчет по лабораторной работе можно в электронном виде по почте ol@istu.ru (в формате .doc, .docx или .pdf; дополнительно к сообщению необходимо прикрепить тексты исходных и exe-файлов). Окончательный вариант представляется студентом преподавателю в *печатном виде.*

Содержание отчета

Отчет должен соответствовать требованиям оформления документов такого типа и содержать следующие пункты:

1. Титульный лист с указанием темы лабораторной работы и номера варианта
2. Постановку задачи
3. Алгоритм решения задачи (математическая постановка)
4. Описание программы:
   - структура входных данных;
   - структура выходных данных;
   - алгоритм программы.
5. Листинг текста программы
6. Тестовые примеры
7. Выводы

Примечание: если в задании не указан метод решения задачи или структура входных данных, то выбор метода и структуры данных должен быть сделан самостоятельно студентом, обоснование выбора должно быть отражено в отчете.

ЛАБОРАТОРНАЯ РАБОТА №3
Генерация комбинаторных объектов

3.1. Цель работы

Цель работы состоит в изучении методов решения перечислительных задач комбинаторики. Например, требуется рассмотреть все двухэлементные подмножества заданного множества. К примеру для множества {1,2,3} число таких подмножеств равно $C_3^2 = 3$, и это можно считать в ряде случаев кратким ответом. Однако полным ответом будут служить все указанные подмножества, т.е. {1,2}, {1,3}, {2,3}. Вряд ли кто-нибудь захочет создавать перечень всех 5-элементных подмножеств из множества мощности 20 ($C_{20}^5 = \frac{20!}{5!\cdot 15!}$), печатая их на тысячах страницах или выводя в огромный файл. Что действительно нужно, так это научиться представлять их в виде некоторой структуры данных, чтобы программа могла последовательно анализировать каждую комбинацию.

Вот почему говорят *о генерации всех необходимых комбинаторных объектов*.

3.2. Теоретические сведения

3.2.1. Основные определения[1,2,3]

Правило суммы: пусть A и B — конечные множества такие, что $A \cap B = \emptyset$, $|A| = n$ и $|B| = m$, тогда $|A \cup B| = n + m$. Это значит, что если элемент $a \in A$ можно выбрать n способами, а элемент $b \in B$ — m способами, то выбор элемента $x \in A \cup B$ можно осуществить $(n + m)$ способами.

Правило прямого произведения: пусть A и B — конечные множества, $|A| = n$ и $|B| = m$, тогда $|A \times B| = nm$. Это значит, что если элемент $a \in A$ можно выбрать n способами и если после каждого такого выбора элемент $b \in B$ можно выбрать m способами, то выбор пары $(a, b) \in A \times B$ в указанном порядке можно осуществить $|A \times B| = nm$ способами.

Пусть $A = \{a_1, a_2, \dots, a_n\}$. Будем выбирать из этого множества объекты и записывать их номера (объект после этого может быть возвращен назад, причем, заметим, что с номерами объектов работать гораздо проще, чем с их именами, поэтому в дальнейшем часто множества будем задавать номерами объектов, т. е. $A = \{1, 2, \dots, n\}$). Повторяем эту процедуру k раз. В результате этой процедуры получим последовательность $(i_1, i_2, \dots, i_k)$, которая называется *выборкой* (комбинацией) из n элементов по k. Если объекты не возвращались в исходное множество, то имеем выборку *без повторений*, иначе имеем выборку *с повторениями*.

Одна из задач комбинаторики состоит в переборе всех таких комбинаций (перечислительные задачи). Для этого необходимо определить, какие выборки считаются различными.

Если две выборки, не отличающиеся по составу, но имеющие различный порядок элементов в записи, считаются различными, то говорят о *размещении из n объектов по k*.

В противном случае (порядок выбора элемента считается несущественным) говорят о *сочетании из n объектов по k*. Более того, следует заметить, что такая комбинация является *подмножеством рассматриваемого множества объектов*.

Размещение из n элементов по n называется *перестановкой n элементов*.

Пример. Выбирая два элемента из имеющихся трех $\{a, b, c\}$, получаем девять размещений с повторениями, шесть размещений без повторений, шесть сочетаний с повторениями и три сочетания без повторений. Если рассматривать перестановки из двух элементов $\{a, b\}$, то с повторениями их будет ровно четыре, а без повторений – только две. Все возможные размещения, сочетания и перестановки приведены в табл. 1.

Таблица 1

	Размещения	Сочетания	Перестановки
С повторением	(a,a), (a,b), (a,c), (b,a), (b,b), (b,c), (c,a), (c,b), (c,c)	{a, a}, {a, b}, {a, c}, {b, b}, {b, c}, {c, c}	(a, a), (a, b), (b, a), (b, b)
Без повторения	(a,b), (a,c), (b,a), (b,c), (c,a), (c,b)	{a, b}, {a, c}, {b, c}	(a, b), (b, a)

Упорядоченным разбиением конечного множества S, где $|S| = n$, на k различных блоков называется система *упорядоченных* попарно не пересекающихся подмножеств $S = S_1 \cup S_2 \cup \ldots \cup S_k$, $|S_i| = n_i$, $i = 1..k$ и $\sum_{i=1}^{k} n_i = n$. Последовательность различных $S_1, S_2, \ldots, S_k$ рассматривается как упорядоченная последовательность подмножеств.

В случае, когда порядок подмножеств не имеет значения, мы имеем дело с так называемыми *неупорядоченными разбиениями*. Например, следующие упорядоченные разбиения множества $S = \{1,2,3,4,5\}$ вида

$$\{1,3\},\{4\},\{2,5\}$$
$$\{1,3\},\{2,5\},\{4\}$$
$$\{2,5\},\{1,3\},\{4\}$$
$$\{4\},\{1,3\},\{2,5\}$$

в случае рассмотрения их как неупорядоченных считаются одинаковыми.

Особое место занимают задачи представления целого числа в виде суммы целых неотрицательных или положительных слагаемых: композиции, разложения и разбиения.

Композицией целого числа n называется последовательность $(p_1, \ldots, p_k)$, где $p_i > 0$ и $\sum_{i=1}^{k} p_i = n$.

Пример. Все композиции числа 4: 1+1+1+1, 1+1+2, 1+3, 4, 1+2+1, 2+2, 2+1+1, 3+1

Разложением целого числа n на k частей называется последовательность $(p_1, \ldots, p_k)$, где $p_i \geq 0$ и $\sum_{i=1}^{k} p_i = n$.

Разложения в отличии от композиций могут обладать нулевыми компонентами. Поэтому количество частей в разложении может быть неограниченным и всегда должно указываться.

Пример. Все разложения числа 4 на 2 части: 0+4, 1+3, 2+2, 4+0, 3+1

Разбиением целого числа n называется мультимножество $[{p_1}^{n_1}, {p_2}^{n_2}, \dots, {p_k}^{n_k}]$ такое, что $p_i > 0$, $n_i \geq 0$ и $\sum_{i=1}^{k} p_i n_i = n$.

В отличие от композиций и разложений *порядок элементов в разбиении не важен*! Учитывая это для организации корректного перебора разбиений упорядочим их элементы так, что $p_1 > p_2 > \dots > p_k$.

Пример. Разбиения числа 5: 1+1+1+1+1 2+1+1+1 3+1+1 4+1 5
2+2+1 3+2

3.3. Генерирование комбинаторных объектов[1,4,5,8,9]

При порождении комбинаторных объектов существуют две возможные цели - систематическое порождение всех возможных конфигурация и порождение равномерно распределенных случайных конфигураций.

Алгоритм *систематического порождения*

```
выбрать начальную конфигурацию;
while (not условие окончания) do
  вывести объект;
  преобразовать текущий объект в следующий;
end;
```

При такой схеме необходимо учитывать, что алгоритм предпринимает еще одно преобразование после того, как выведен последний объект.

На практике иногда желательно, чтобы изменения текущего объекта при переходе к следующему были, возможно, малыми. Такие алгоритмы называются *алгоритмами минимального изменения*.

При втором подходе можно задать взаимно однозначное соответствием между целыми числами $0,1,\dots,n-1$ и n объектами, тогда случайное порождение осуществляется случайным порождением целого числа от 0 до $n-1$ и обращением его в объект. В этом случае систематическое порождение сводится к перечислению целых чисел от 0 до $n-1$ и обращением каждого числа в объект (с данной схемой вы уже ознакомились при выполнении второй лабораторной работы).

3.3.1. Алгоритм поиска с возвращением

Одним из известных методов позволяющих провести поиск множества решений на конечных множествах является метод поиска с возвращением. Этот метод постоянно пытается расширить частичное решение. Если расширение текущего частичного решения невозможно, то возвращаются к более короткому частичному решению и пытаются снова его продолжить.

В общем случае решение задачи можно выразить как вектор $(a_1, a_2, a_3, \dots)$ конечной, но неопределенной длины, удовлетворяющий некоторым ограничениям. Каждое $a_i \in A_i$, где A_i — конечное линейно упорядоченное множество. Таким образом, при исчерпывающем поиске должны рассматриваться элементы множества $(a_1, a_2, a_3, \dots, a_i) \in A_1 \times A_2 \times \dots \times A_i$, где $i = 0,1,2,\dots$ в качестве возможных решений. В качестве исходного частичного решения примем пустой вектор $()$ и на основании имеющихся ограничений выясним, какие элементы из A_1 являются кандидатами в a_1. Обозначим это подмножество кандидатов через $S_1 \subseteq A_1$. В результате имеем частичное решение (a_1). В общем случае для расширения частичного решения от $(a_1, a_2, \dots, a_{k-1})$ до $(a_1, a_2, \dots, a_{k-1}, a_k)$ кандидаты на роль a_k выбираются из $S_k \subseteq A_k$. Если частичное решение $(a_1, a_2, \dots, a_k)$не представляет возможности для выбора элемента a_k, то $S_k =$

$\emptyset$; возвращаемся и выбираем новый элемент a_{k-1}. Если новый элемент a_{k-1} выбрать нельзя, возвращаемся еще дальше и выбираем новый элемент a_{k-2} и т.д. Этот процесс удобно представлять в терминах прохождения дерева поиска в глубину. Процедура поиска с возвращением для нахождения всех решений формально представлена в следующем алгоритме.

Алгоритм 3.1. *Общий алгоритм поиска с возвращениями*

```
S1=A1;                                          {Выделить кандидатов}
k=1;                                            {Длина частичного
решения}
while k>0 do begin
  while Sk≠∅ do begin
        ak∈Sk;                                  {Расширить частичное решение}
        Sk=Sk\{ak};                             {Удалить выбранного кандидата}
        if (a1,a2,...,ak)—решение then Сохранить решение;
        k=k+1;                                  {Расширить частичное решение}
        Sk⊂Ak;                                  {Выделить кандидатов}
   end;
   k=k-1;                          {Вернуться, уменьшить частичное решение}
end.
```

Поиск с возвращением представляет собой только общий метод. Непосредственное его применение обычно ведет к алгоритмам, время работы которых недопустимо велико. Поэтому, чтобы метод был полезен, к нему нужно относиться как к схеме, с которой следует подходить к задаче.

3.3.2. Генерирование k-элементных подмножеств[5,8,9]

Как уже было сказано ранее между -разрядными двоичными наборами и подмножествами некоторого множества мощности n существует взаимно однозначное соответствие. Т.е. порождение подмножеств множества $S = \{s_1, s_2, \ldots, s_n\}$ эквивалентно порождению n-разрядными двоичных наборов.

Наиболее прямым способом порождения всех двоичных наборов длины n является счет в системе счисления с основанием 2, что реализует алгоритм 2.

Алгоритм 3.2. *Порождения всех -разрядных наборов в двоичной системе счисления*

```
Вход: n≥0 - число разрядов
Выход: последовательность n-разрядных наборов

for i=0 to n do bi=0;
while bn  ≠1 do
   Print(bn-1,bn-2,...,b0);
   i=0;
   while bi=1 do
      bi=0;
      i=i+1;
   end;
   bi=1
end.
```

Алгоритм 3.3. *Порождение подмножеств счетом в двоичной системе счисления*

```
Вход: S - множество
Выход: последовательность подмножеств

S=∅;
while an+1∉S do
  Print(S);
  i=0;
```

```
  while aᵢ∈S do
     S=S\{aᵢ}
     i=i+1;
  end;
  S=SU{aᵢ}
end.
```

Существует алгоритм порождения подмножеств в порядке минимального изменения. Минимальным возможным изменением при переходе от одного подмножества к другому является добавление или удаление одного элемента множества. В терминах двоичных наборов это означает, что последовательные наборы должны различаться в одном разряде. Такие последовательности двоичных наборов называются *кодами Грея*; более точно, -разрядный код Грея есть упорядоченная циклическая последовательность 2^n n-разрядных наборов (кодовых слов), такая, что последовательные слова отличаются в одном разряде.

Алгоритм 3.4. *Построения бинарного кода Грея*

```
Вход: n≥0 - мощность множества
Выход: последовательность кодов подмножествS
S:array[1..n] of 0..1      {битовая шкала для представления подмножеств}

for i=1 to n do S[i]=0;              {инициализация}
S=∅;                                 {пустое подмножество}
for i=1 to 2^(n-1) do
   k=Q(i);        {определяем номер элемента для добавления или удаления}
   S[k]=1-S[k];      {изменяем подмножество добвляя или удаляя элемент}
   S;                {получаем новое подмножество}
end;
```

Функция Q определения номера разряда, подлежащего изменению

```
Вход: i-номер подмножества
Выход: номер изменяемого разряда

p=1;
j=i;
while j четно do
   j=j/2;
   p=p+1;
end;
return p;
```

Для сравнения в таблице приведены протоколы выполнения алгоритм 3.2 и алгоритма 3.4.

	Код		
i	Двоичный	p	Грея
1.	**000**		**000**
2.	**001**	1	**001**
3.	**010**	2	**011**
4.	**011**	1	**010**
5.	**100**	3	**110**
6.	**101**	1	**111**
7.	**110**	2	**101**
8.	**111**	1	**100**

Коды Грея удобно задавать начальным словом и последовательностью переходов, т.е. упорядоченным списком номеров разрядов (пронумерованных справа налево), которые меняются при переходе от одного кодового слова к другому. Так для приведенного в таблице кода начальное слово (000), а последовательность переходов будет иметь вид T_3=(1,2,1,3,1,2,1).

Существует много -разрядных кодов Грея. Здесь был приведен *двоично-отраженный* код.

3.3.3. Генерирование перестановок[1,5,8,9]

Опишем три разных метода генерирования всех $n!$ перестановок -элементного множества. Разместим элементы нашего множества в массив $P: array[1..n]\ of\ type$. Во

всех трех методах элементарной операцией, которая применяется к массиву P, является поэлементная транспозиция, т.е. обмен значениями переменных $P[i]$ и $P[j]$, где $1 < i, j < n$. Обозначим эту операцию $P[i] \leftrightarrow P[j]$.

Не нарушая общности задачи договоримся, что элементами множества являются целые числа от $1..n$. Определим на множестве всех перестановок или на множестве всех последовательностей длины n лексикографический порядок:

$$(x_1, x_2, \ldots, x_n) < (y_1, y_2, \ldots, y_n) \Leftrightarrow \exists k \geq 1 : x_k \leq y_k \text{ и } \forall i < k \Rightarrow x_i = y_i.$$

Заметим, что лексикографический порядок определяет стандартную последовательность, в которой слова длины n появляются в словаре. Подобным же образом определяется антилексикографический порядок, обозначаемый через $<'$, с той разницей, что как очередность позиций в последовательности, так и упорядочение элементов множества P обратны по отношению к исходным, т.е.

$$(x_1, x_2, \ldots, x_n) <' (y_1, y_2, \ldots, y_n) \Leftrightarrow \exists k \geq 1 : x_k > y_k \text{ и } \forall i < k \Rightarrow x_i = y_i.$$

Для примера приведем перестановки множества $P = \{1,2,3\}$ в лексикографическом и антилексикографическом порядке:

Лексикографический порядок	Антилексикографический порядок
123, 132, 213, 231, 312, 321	123, 213, 132, 312, 231, 321

Можно заметить, что последовательность перестановок на множестве $\{1,2,\ldots,n\}$ представлена в лексикографическом порядке, если она записана в порядке возрастания получающихся чисел. Порождать такую последовательность можно следующим образом.

Алгоритм 3.5. *Генерирование всех последовательностей в лексикографическом порядке*

```
Вход: n – количество элементов
Выход: перестановки множества {1,…,n} в лексикографическом порядке

for i=1 to n do P[i]=i;
i=1;
while i≠0 do
   Print(P[1],P[2],…,P[n]);
   i=n-1;
   while P[i]>P[i+1] do i=i-1;
   j=n;
   while P[i]>P[j] do j=j-1;
   P[i]↔P[j];
   k=n;
   s=i+1;
      while k>s do
         P[k]↔P[s];
         k=k-1;
         s=s+1;
      end;
end;
```

В случае генерации перестановок в антилексикографическом порядке заметим последняя перестановка — обращение первой. Во-вторых, последовательность можно разделить на n блоков длины $(n-1)!$, соответствующих убывающим значениям элемента в последней позиции. Первые $n-1$ позиций блока, содержащего элемент p в последней позиции, определяют последовательность перестановок множества $\{1,\ldots,n\}\backslash\{p\}$ в антилексикографическом порядке.

Алгоритм 3.6. *Генерирование перестановок в антилексикографическом порядке*

```
Вход: n – количество элементов
Выход: перестановки множества {1,...,n} в антилексикографическом порядке

for i=1 to n do P[i]=i;                          {инициализация}
ANTYLEX(n)
```

Процедура *ANTYLEX*

```
Вход: n – количество элементов
Выход: перестановки множества {1,...,n} в антилексикографическом порядке

if m=1 then Print(P[1],...,P[n])                  {новая перестановка}
       else
          for i=1 to m do
             ANTYLEX(m-1);                        {рекурсивный вызов}
             if i<m then
                P[i]↔P[m];                       {следующий элемент}
                REVERSE(m-1);    {изменение порядка элементов}
             end;
          end;
end;
```

Вспомогательная процедура *REVERSE*

```
Вход: m – номер элемента, задающий отрезок массива, подлежащий
перестановке в обратном порядке
Выход: первые m элементов массива переставлены в обратном порядке

i=1;                            {задаем нижнюю границу изменяемого отрезка}
while i<m do
   P[i]↔P[m];                     {меняем местами элементы}
   i=i+1;                         {увеличиваем нижнюю границу}
   m=m-1;                         {уменьшаем верхнюю границу}
end;
```

Существует алгоритм, в котором каждая следующая перестановка получается из предыдущей с помощью выполнения *только* одной транспозиции. Это может оказаться существенным в ситуации, когда с каждой перестановкой связаны некоторые вычисления и когда существует возможность использования частичных результатов, полученных для предыдущей перестановки, если последовательные перестановки мало отличаются друг от друга.

Основная схема этого алгоритма заключается в рекурсивной процедуре *PERM*.

```
Вход: m – количество элементов
Выход: последовательность перестановок множества {1,...,n}

if m=1 then Print(P[1],...,P[n])                    {новая перестановка}
       else
         for i=1 to m do
           PERM(m-1);                               {рекурсивный вызов}
           if i<m then P[B[m,i]]↔P[m];
         end;
end;
```

Задачей этой процедуры является генерирование всех перестановок элементов $P[1],\ldots,P[n]$ путем последовательное генерирование всех перестановок элементов $P[1],\ldots,P[n-1]$ и замену элемента $P[n]$ на один из элементов $P[1],\ldots,P[n-1]$, определяемый из массива $B[m,i], 1 \le i < m \le n$. Чтобы эта схема работала правильно

необходимо определить массив B так, чтобы гарантировать, чтобы каждая транспозиция $P[B[m,i]] \leftrightarrow P[m]$ вводила бы новый элемент в $P[n]$. Представим теперь алгоритм, в котором значение $B[m,i]$ вычисляется динамически как функция от m и i.

Алгоритм 3.7. *Генерирование всех перестановок за минимальное число транспозиций*

```
Вход: n-количество элементов
Выход: перестановки множества {1,...,n}

procedure B(m,i);
if (m mod 2=0) and (m>2) then
        if i<m-1 then B=i
                 else B=m-2
                         else B=m-1;
end;

procedure PERM(m);
if m=1 then Print(P[l],...,P[n])  {новая перестановка}
       else
         for i=1 to m do
            PERM(m - 1);
            if i<m then P[B(m,i)]↔P[m];
         end;
end;

begin                                {главная программа}
   for i=1 to n do P[i] =i;
   PERM(n);
end.
```

Существует еще более быстрый алгоритм генерирования перестановок. В этом алгоритме каждая следующая перестановка получается из предыдущей с помощью однократной транспозиции *соседних* элементов. Суть алгоритма заключается в следующем. Пусть последовательность перестановок элементов $2,3,\ldots,n$, (например, $(2\ 3)$, $(3\ 2)$ для $n=3$) уже построена. Тогда требуемую последовательность перестановок элементов $1,2,\ldots,n$ получим, вставляя элемент 1 всеми возможными способами в каждую перестановку элементов $2,3,\ldots,n$. В нашем случае получаем (123), (213), (231), (321), (312), (132).

В общем виде элемент 1 помещается между первой и последней позициями попеременно вперед и назад $(n-1)!$ раз. На основе этой конструкции получается рекурсивный алгоритм, генерирующий требуемую последовательность перестановок для произвольного n. Однако этот метод имеет недостаток: последовательность перестановок строится «целиком» и только после окончания всего построения ее можно считывать. Очевидно, что решение такого типа потребовало бы огромного объема памяти.

Разберем нерекурсивный вариант этого алгоритма. Для каждого $i, 1 \le i < n$, булева переменная $PR[i]$ содержит информацию о том, переносится ли элемент i вперед ($PR[i] = $ истина) или же назад ($PR[i] = $ ложь), переменная же $C[n]$ показывает, какую из возможных $n-i+1$ позиций элемент 1 занимает относительно элементов $i+1,\ldots,n$ на своем пути вперед или назад. Позицию элемента 1 в таблице P определяем на основании его позиции в блоке, содержащем $i, i+1,\ldots,n$, а также на основании числа элементов из $1,2,\ldots,i-1$, которые находятся слева от этого блока. Это число, будучи значением переменной x, вычисляется как число элементов $j < i$, которые, двигаясь назад, достигли бы своего крайнего левого положения ($C[j] = n - j + 1, PR[j] = $ ложь). Каждая новая перестановка образуется транспозицией самого

меньшего из элементов j, который не находится в граничном положении (т.е., $C[i] < n - j + 1$) с его левым или правым соседом. Это реализует приведенный ниже алгоритм.

Алгоритм 3.8. *Генерирование всех перестановок транспозицией соседних элементов*

```
Вход: n – количество элементов
Выход: перестановки множества {1,...,n}

for i=1 to n do
   P[i]=i;
   C[i]=1;
   PR[i]:=истина;
end;
C[n]=0;
Print(P[l],..., P[n]);
i=1;
while i<n do
   i =1;
   x=0;
   while C[i]=n-i+1 do
      PR[i]=not PR[i];
      C[i] := 1;
      if PR[i] then x=x+1;
      i:=i + l
   end;
   if i<n then                    {выполнение транспозиции}
     if PR[i] then k=C[i]+x;
          else k=n-i+1-C[i]+x;
     P[k]↔P[k+1];
     Print(P[i],...,P[n]);
     C[i]=C[i]+1;
   end;
end.
```

3.3.4. Генерирование сочетаний

Иногда в задачах не о всем множестве $\{a_1, a_2, \ldots, a_n\}$, а только тех подмножествах, которые удовлетворяют некоторым ограничениям. Особый интерес представляют подмножества фиксированной длины k, (C_n^k - сочетания из n предметов по k). Например, сочетания из шести элементов по три ($C_6^3 = 20$) в лексикографическом порядке запишутся следующим образом: 123, 124, 125, 126, 134, 135, 136, 145, 146, 156, 234, 235, 236, 245, 246, 256, 345, 346, 356, 456.

Сочетания в лексикографическом порядке можно порождать следующим образом. Начиная с сочетания $\{1,2,\ldots,k\}$ следующее сочетание находится просмотром текущего справа налево с тем, чтобы определить место самого правого элемента, который не достиг еще своего максимального значения. Этот элемент увеличивается на единицу, и всем элементам справа от него присваиваются новые наименьшие возможные значения.

Алгоритм 3.9. *Порождение сочетаний в лексикографическом порядке*

```
c[0]=1;
for i=1 to k do c[i]=i;
j=1;
while j≠0 do
   Print(c[1],c[2],…,c[k]);
   j=k;
```

```
   while c[j]≥n-k+j do j=j-1;
   c[j]=c[j]+1;
   for j=j+1 to k do c[i]=c[i-1]+1;
end;
```

3.3.5. Генерирование размещений

Так как размещения - это упорядоченные k -элементные подмножества множества из n элементов (упорядоченные сочетания из n по k), то алгоритм их порождения можно получить путем комбинации алгоритмов порождения сочетаний и перестановок.

3.3.6. Генерирование композиций и разбиений

При решении задачи о нахождении композиций следует заметить, что каждой композиции числа n из k частей при $z_i \geq 0$ взаимно однозначно соответствует двоичный набор, такой, что первое слагаемое равно числу единиц, стоящих перед первым нулем в наборе, второе - числу единиц, стоящих между первым и вторым нулями, и т.д.

Пример. Композиции числа $n = 4$ из $k = 3$ частей:

№	Композиция	Двоичный набор
1	0+0+4	0 0 1 1 1 1
2	0+1+3	0 1 0 1 1 1
3	0+2+2	0 1 1 0 1 1
…	…	…
13	3+0+1	1 1 1 0 0 1
14	3+1+0	1 1 1 0 1 0
15	4+0+0	1 1 1 1 0 0

Длина двоичного набора равна $n + k - 1$, число нулей равно $k - 1$, следовательно, число наборов (искомых композиций) равно числу способов выбора $k - 1$ мест для нулей из $n + k - 1$ мест (C_{n+k-1}^{k-1}) или, то же самое, числу способов выбора n мест для единиц из $n + k - 1$ мест (C_{n+k-1}^{n}).

Разбиения n отличаются от композиций n тем, что порядок компонент не важен. Каждое разбиение удовлетворяет условию $z_1 > z_2 > \ldots > z_k$ (см. пример п.3.2.1). Генерировать разбиения наиболее эффективно в лексикографическом порядке. Для этого, начав с $[1^n]$, будем переходить от одного разбиения к следующему, рассматривая самый правый элемент разбиения ${p_k}^{n_k}$. Если $p_k \cdot n_k$ достаточно велико ($n_k > 1$), можно исключить два p_k для того, чтобы добавить еще одно $p_k + 1$ (или включить еще одно $p_k + 1$, если в текущий момент его нет). Если $n_i = 1$, то $n_{k-1} \cdot p_{k-1} + n_k \cdot p_k$ достаточно велико для того, чтобы добавить $p_{k-1} + 1$. В любом случае то, что остается, превращается в соответствующее число единиц и формируется новое разбиение.

Алгоритм 3.10. *Порождение разбиений в лексикографическом порядке*

```
k=1;
p₋₁=0;
n₋₁=0;
p₀=n+1;
n₀=0;
p₁=1;
n₁=n;
while k≠0 do
   Print[p1,];
```

```
    sum=p_k n_k;
    if n_k=1 then
        k=k-1;
        sum=sum+p_k n_k;
    end;
    if p_{k-1}=pk+1 then
        k=k-1;
        n_k=n_k+1
                  else
           p_k=p_k+1;
           n_k=1;
    end;
    if sum>p_k then
        p_{k+1}=1;
        n_{k+1}=sum-p_k;
        k=k+1;
end;
```

3.4. Порядок выполнения работы

Изучить теоретический материал, включая определения и алгоритмы.

Решить следующие задачи:

1. (1 балл) Вывести все двоичные наборы длины n.
2. (1 балл) Напечатать все перестановки длины n в лексикографическом порядке.
3. (1 балл) Напечатать все перестановки длины n, используя минимальное число транспозиций.
4. (1 балл) Перечислить все вложения (функции, переводящие разные элементы в разные) множества $\{1..k\}$ в $\{1..n\}$ (предполагается, что $k \leq n$).
5. (1 балл) Представляя разбиения как невозрастающие последовательности, перечислить их в лексикографическом порядке (для n=4, например, должно получиться $1+1+1+1$, $1+1+2$, $2+2$, $1+3$, 4).

3.5. Дополнительные задачи

1. (1 балл) Используя код Грея по заданной мощности n множества $\{1..n\}$ вывести все его подмножества.
2. (1 балла) Перечислить все числа-палиндромы (перевертыши)длины n.
3. (1 балл) Представляя разбиения как невозрастающие последовательности, перечислить их в порядке, обратном лексикографическому (для n=4, например, должно получиться 4, $3+1$, $2+2$, $2+1+1$, $1+1+1+1$).
4. (1 балл) Представляя разбиения как неубывающие последовательности, перечислить их в порядке, обратном лексикографическому. Пример для $n = 4$: 4, $2+2$, $1+3$, $1+1+2$, $1+1+1+1$.
5. (1 балл) Напечатать все последовательности из k положительных целых чисел, у которых -ый член не превосходит i.
6. (1 балла) Перечислить все последовательности длины n из чисел $1..k$ в таком порядке, чтобы каждая следующая отличалась от предыдущей в единственной цифре, причем не более чем на 1.
7. (2 балла) Перечислить все последовательности длины $2n$, составленные из n единиц и n минус единиц, у которых сумма любого начального отрезка

неотрицательна, т.е. число минус единиц в нем не превосходит числа единиц (число таких последовательностей называют *числом Каталана*).

8. (2 балла) Перечислить все расстановки скобок в произведении n сомножителей. Порядок сомножителей не меняется, скобки полностью определяют порядок действий. (Например, для $n = 4$ есть 5 расстановок $((ab)c)d$, $(a(bc))d$, $(ab)(cd)$, $a((bc)d)$, $a(b(cd))$.
9. (2 балла) *«Счастливый билет»*. Дано n ($n > 1$) произвольных отличных от нуля цифр: $a_1, a_2, \dots, a_n$ и произвольное число M. Написать программу, которая расставляла бы между каждой парой цифр знаки $+$ и $-$ так, чтобы значение получившегося выражения было равно числу M.
10. (2 балла) *«Выпуклый многоугольник»*. Дано множество пар целых чисел $\{(x_i, y_i)\}_{i=1}^{n}$ – координаты точек на плоскости. Написать программу выделения тех точек, которые являются вершинами выпуклого многоугольника, содержащего все остальные точки.

3.6. Контрольные вопросы

1. Понятия множества, подмножества, мощности конечного множества, мультимножества.
2. Способы задания множеств. Число подмножеств конечного множества.
3. Основные схемы порождения комбинаторных объектов.
4. · Понятие кода Грея. Рекурсивное определение двоично-отраженного кода Грея и его последовательности переходов.
5. Понятие перестановки. Число перестановок -элементного множества.
6. Понятие перестановки с повторениями. Число таких перестановок. Понятие полиномиального коэффициента.
7. Понятие последовательности перестановок в лексикографическом порядке.
8. Понятие последовательности перестановок в порядке минимального изменения. Рекурсивное определение такой последовательности.
9. Понятие сочетания. Число сочетаний из n элементов по k. Свойства сочетаний.
10. Понятие сочетания с повторениями. Число сочетаний с повторениями. Предложить алгоритм генерации таких сочетаний.
11. Понятие последовательности сочетаний в возрастающем лексикографическом порядке.
12. Понятие размещения. Число размещений.
13. Понятие композиции. Число композиций числа n. Число композиций n из k частей при $\mathrm{p}_i > 0$. Число композиций n из k частей при $\mathrm{p}_i \geq 0$. Связь таких композиция с сочетаниями из k элементов по n с повторениями.
14. Понятие разбиения и его отличие от композиции.

1.1. Литература

1. *Новиков, Ф. А.* Дискретная математика для программистов : учеб. для вузов. – 2-е изд. – СПб. : Питер, 2005. – 364 с.
2. *Хаггарти, Р.* Дискретная математика для программистов : учеб. пособие. – 2-е изд. – М. : Техносфера, 2005. – 400 с.
3. *Судоплатов, С. В.* Дискретная математика : учебник / С. В. Судоплатов, Е. В. Овчинникова. – 2-е изд. – Москва ; ИНФРА-М ; Новосибирск : НГТУ, 2005. – 256 с.
4. *Дейкстра. Э.* Дисциплина программирования. – М.: Мир, 1978. – 280 с.
5. *Кнут Д.* Искусство программирования, том 1, выпуск 2. Основные алгоритмы. : Пер. с англ. - М.: ООО "И.Д. Вильямс", 2007. – 682 с.

6. *Грэхем Р., Кнут Д., Паташник О.* Конкретная математика. Основание информатики/ Пер. с англ. – М.: Мир, 1998. – 703 с.
7. *Кнут Д.* Искусство программирования, том 2, выпуск 2. Получисленные алгоритмы. : Пер. с англ. - М.: ООО "И.Д. Вильямс", 2007. – 788 с.
8. *Иванов Б.Н.* Дискретная математика. Алгоритмы и программы : учеб. пособие. – М. : Лаборатория Базовых Знаний, 2001 – 288 с.
9. *Липский В.* Комбинаторика для программистов. – М.: Мир, 1988. – 200 с.
10. *Холл М.* Комбинаторика. – М.: Мир, 1970. – 424 с.