Министерство образования Республики Беларусь

Учреждение образования

«Гомельский государственный университет им. Ф. Скорины»

Математический факультет

Кафедра МПУ

Реферат

**Комбинаторные задачи**

Исполнитель:

Студентка группы М-42 Макарченко А.Ю.

Научный руководитель: Долинский М.С.

Гомель 2005

**Содержание**

Введение

Генерация k-элементных подмножеств

Генерация всех подмножеств данного множества

Генерация всех перестановок n-элементного множества

Разбиения множества

Заключение

Литература

**Введение**

Задачи дискретной математики, к которым относится большинство олимпиадных задач по информатике, часто сводятся к перебору различных комбинаторных конфигураций объектов и выбору среди них наилучшего, с точки зрения условия той или иной задачи. Поэтому знание алгоритмов генерации наиболее распространенных комбинаторных конфигураций является необходимым условием успешного решения олимпиадных задач в целом. Важно также знать количество различных вариантов для каждого типа комбинаторных конфигураций, так как это позволяет реально оценить вычислительную трудоемкость выбранного алгоритма решения той или иной задачи на перебор вариантов и, соответственно, его приемлемость для решения рассматриваемой задачи, с учетом ее размерности. Кроме того, при решении задач полезным оказывается умение для каждой из комбинаторных конфигураций выполнять следующие операции: по имеющейся конфигурации получать следующую за ней в лексикографическом порядке; определять номер данной конфигурации в лексикографической нумерации всех конфигураций; и, наоборот, по порядковому номеру выписывать соответствующую ему конфигурацию.

# **Генерация k-элементных подмножеств**



В комбинаторике такие подмножества называют сочетаниями из *n* элементов по *k* элементов и обозначают *Cnk* . Их количество выражается следующей формулой:

Однако при программировании гораздо удобнее использовать следующие рекуррентные соотношения:

Объясняется это тем, что в формуле (1) числитель и знаменатель растут очень быстро, поэтому в силу особенностей компьютерной арифметики не всегда возможно точно вычислить значение *Cnk*, даже когда последнее не превосходит максимально представимое целое число.

При фиксированном значении *n* максимального значения число сочетаний достигает при *k* = *n*/2 (вернее, для четного *n* максимум один и он указан, а для нечетного — максимум достигается на двух соседних значениях *k*: [*n/*2] и



[*n*/2]+1). Поэтому особенно полезной оказывается следующая оценка для четных *n* [4] (очевидно, что при нечетных *n* отличия будут минимальными), основанная на формуле Стирлинга:

Если допустить, что за время, отведенное для решения задачи, мы можем перебрать около 106 вариантов, то из формулы (3) следует, что генерацию всех сочетаний из *n* элементов для любого фиксированного *k* можно проводить для *n* ≤ 24.

Обычно генерацию всех *k*-элементных подмножеств проводят в лексикографическом порядке, тем более что в данном случае это не приводит ни к усложнению алгоритма, ни к увеличению его вычислительной трудоемкости. Напомним, что порядок подмножеств называется лексикографическим, если для любых двух подмножеств справедливо, что раннее должно быть сгенерировано то из них, из индексов элементов которого можно составить меньшее *k*-значное число в *n*-ричной системе счисления (или в десятичной, для *n* < 10). Так, для *n* = 6 и *k* = 3 сочетание из третьего, первого и пятого элемента должно быть сгенерировано раньше, чем из второго, третьего и четвертого, так как 135 < 234.

Рассмотрим рекурсивный алгоритм решения данной задачи. Идея сведения данной задачи к задаче меньшей размерности следующая. Первым элементом подмножества может быть любой элемент, начиная с первого и заканчивая (*n* – *k* + 1)-м элементом. После того, как индекс первого элемента подмножества зафиксирован, осталось выбрать *k* – 1 элемент из элементов с индексами, большими, чем у первого. Далее поступаем аналогично. Когда выбран последний элемент, то мы достигли конечного уровня рекурсии и выбранное подмножество можно обработать (проанализировать или распечатать). В предлагаемой ниже программе массив a содержит значения элементов исходного множества и может быть заполнен произвольным образом. В массиве p будем формировать очередное сочетание из *k* элементов.

*const nmax = 24;*

*type list = array[1..nmax] of integer;var k,i,j,n,q : integer;a,p : list;procedure print(k : integer);*

*var i:integer;beginfor j:=1 to k dowrite(p[j]:4);writelnend;{print}*

*procedure cnk(n,k : integer);procedure gen(m,L : integer);var i:integer;beginif m=0 then print(k) elsefor i:=L to n-m+1 dobegin*

*p[k-m+1]:=a[i];*

*gen(m-1,i+1)*

*end*

*end;{gen}*

*begin {cnk}gen(k,1)end;{cnk}begin {main}readln(n,k);for i:=1 to n doa[i]:=i;{заполнить массив можно и по-другому}*

*cnk(n,k)*

*end.*

Заметим, что собственно генерация сочетаний производится в рекурсивной подпрограмме gen. Она имеет следующие параметры: *m* - сколько элементов из множества нам еще осталось выбрать и *L* - начиная с какого элемента исходного множества, следует выбирать эти *m* элементов. Обратите внимание, что именно вложенная структура описания процедур cnk и gen позволяет не передавать при рекурсивных вызовах значения *n* и *k*, а из основной программы обращаться к процедуре cnk с параметрами, соответствующими постановке задачи, не вдаваясь в подробности ее решения. Такой способ будем применять и в дальнейшем.

# **Генерация всех подмножеств данного множества**

При решении олимпиадных задач чаще всего заранее неизвестно, сколько именно элементов исходного множества должно входить в искомое подмножество, то есть необходим перебор всех подмножеств. Однако, если требуется найти минимальное подмножество, то есть состоящее как можно из меньшего числа элементов (или максимальное подмножество), то эффективнее всего организовать перебор так, чтобы сначала проверялись все подмножества, состоящие из одного элемента, затем из двух, трех и т. д. элементов (для максимального подмножества — в обратном порядке). В этом случае, первое же подмножество, удовлетворяющее условию задачи и будет искомым, и дальнейший перебор следует прекратить. Для реализации такого перебора можно воспользоваться, например, процедурой cnk, описанной в предыдущем разделе. Введем в нее еще один параметр: логическую переменную flag, которая будет обозначать, удовлетворяет текущее сочетание элементов условию задачи или нет. При получении очередного сочетания вместо его печати обратимся к процедуре его проверки check, которая и будет определять значение флага. Тогда начало процедуры gen следует переписать так:

*procedure gen(m,L:integer);*

*var i:integer;*

*begin*

*if m=0 then*

*begin*

*check(p,k,flag);*

*if flag then exit*

*end*

*else ...*

Далее процедура дословно совпадает с предыдущей версией. В основной же программе единственное обращение к данной процедуре следует заменить следующим фрагментом:

*k:=0;*

*flag:=false;*

*repeat*

*k:=k+1;*

*cnk(n,1,flag)*

*until flag or (k=n);*

*if flag then print(k)*

*else writeln('no solution');*

Очевидно также, что в основной программе запрос значения переменной k теперь не производится.

Существует также альтернативный подход к перебору всех подмножеств того или иного множества. Каждое подмножество можно охарактеризовать, указав относительно каждого элемента исходного множества, принадлежит оно данному подмножеству или нет. Сделать это можно, поставив в соответствие каждому элементу множества 0 или 1. То есть каждому подмножеству соответствует *n*-значное число в двоичной системе счисления (строго говоря, так как числа могут начинаться с произвольного количества нулей, которые значащими цифрами не считаются, то следует заметить, что в соответствие ставятся *n*- или менее - значные числа). Отсюда следует, что полный перебор всех подмножеств данного множества соответствует перебору всех чисел в двоичной системе счисления. Теперь легко подсчитать и количество различных подмножеств данного множества. Оно равно 2*n* – 1 (или 2*n*, с учетом пустого множества). Таким образом, сопоставляя два способа перебора всех подмножеств данного множества, мы получили следующую формулу:



То есть, в рамках сделанной выше оценки на количество допустимых вариантов в переборе, мы можем рассмотреть все подмножества исходного множества только при *n* ≤ 20.

Прежде, чем перейти к рассмотрению программ, соответствующих второму способу перебора, укажем, когда применение этих программ целесообразно. Во-первых, данные программы легко использовать, когда необходимо в любом случае перебрать все подмножества данного множества (например, требуется найти все решения удовлетворяющие тому или иному условию). Во-вторых, когда с точки зрения условия задачи не имеет значения, сколько именно элементов должно входить в искомое подмножество. На примере такой задачи мы и напишем программу генерации всех подмножеств исходного множества в лексикографическом порядке. Задача взята из книги [5].

*Условие*. Дан целочисленный массив a[1..N] (N ≤ 20) и число M. Найти подмножество элементов массива a[i1], a[i2], ...a[ik] такое, что 1 ≤ i1 < i2 < i3 < ... < ik ≤ N и a[i1] + a[i2] + ... + a[ik] = M.

*Решение*. В качестве решения приведем процедуру генерации всех подмножеств, которые можно составить из элементов массива и функцию проверки конкретного подмножества на соответствие условию задачи.

*function check(j:longint):boolean;*

*var k:integer; s:longint;*

*begin*

*s:=0;*

*for k:=1 to n do*

*if ((j shr (k-1))and 1)=1 {данное условие означает, что в*

*k-й справа позиции числа j, в 2-й системе, стоит 1}*

*then s:=s+a[k];*

*if s=m then*

*begin*

*for k:=1 to n do*

*if ((j shr (k-1))and 1)=1 then write(a[k]:4);*

*writeln*

*end*

*end;*

*procedure subsets(n:integer);*

*var q,j:longint;*

*begin*

*q:=1 shl n; {таким образом мы помещаем в q число 2^n}*

*for j:=1 to q-1 do {цикл по всем подмножествам}*

*if check(j) then exit*

*end;*

Заметим, что если все элементы в массиве положительные, то, изменив порядок рассмотрения подмножеств, решение приведенной выше задачи можно сделать более эффективным. Так, если сумма элементов какого-либо подмножества уже больше, чем *M*, то рассматривать подмножества, включающие его в себя уже не имеет смысла. Пересчет же сумм можно оптимизировать, если каждое следующее сгенерированное подмножество будет отличаться от предыдущего не более, чем на один элемент (такой способ перечисления подмножеств показан в [2]). Приведенная же программа черезвычайно проста, но обладает одним недостатком: мы не можем ни в каком случае с ее помощью перебирать все подмножества множеств, состоящих из более, чем 30 элементов, что обусловлено максимальным числом битов, отводимых на представление целых чисел в Турбо Паскале (32 бита). Но, как уже было сказано выше, на самом деле, перебор всех подмножеств у множеств большей размерности вряд ли возможен за время, отведенное для решения той или иной задачи.

# 

# **Генерация всех перестановок *n*-элементного множества**

Количество различных перестановок множества, состоящего из *n* элементов равно *n*!. В этом нетрудно убедиться: на первом месте в перестановке может стоять любой из n элементов множества, после того, как мы на первом месте зафиксировали какой-либо элемент, на втором месте может стоять любой из *n* – 1 оставшегося элемента и т.д. Таким образом, общее количество вариантов равно *n*(*n* – 1)(*n* – 2)...3⋅2⋅1 = *n*!. То есть рассматривать абсолютно все перестановки мы можем только у множест, состоящих из не более, чем 10 элементов.

Рассмотрим рекурсивный алгоритм, реализующий генерацию всех перестановок в лексикографическом порядке. Такой порядок зачастую наимболее удобен при решении олимпиадных задач, так как упрощает применение метода ветвей и границ, который будет описан ниже. Обозначим массив индексов элементов — p. Первоначально он заполнен числами 1, 2, ..., *n*, которые в дальнейшем будут меняться местами. Параметром i рекурсивной процедуры Perm служит место в массиве p, начиная с которого должны быть, получены все перестановки правой части этого массива. Идея рекурсии, в данном случае следующая: на *i*-ом месте должны побывать все элементы массива p с *i*-го по *n*-й и для каждого из этих элементов должны быть получены все перестановки остальных элементов, начиная с (*i*+1)-го места, в лексикографическом порядке. После получения последней из перестановок, начиная с (*i*+1)-го места, исходный порядок элементов должен быть восстановлен.

*{описание переменных совпадает с приведенным выше}*

*procedure Permutations(n:integer);*

*procedure Perm(i:integer);*

*var j,k:integer;*

*begin*

*if i=n then*

*begin for j:=1 to n do write(a[p[j]],' '); writeln end*

*else*

*begin*

*for j:=i+1 to n do*

*begin*

*Perm(i+1);*

*k:=p[i]; p[i]:=p[j]; p[j]:=k*

*end;*

*Perm(i+1);*

*{циклический сдвиг элементов i..n влево}*

*k:=p[i];*

*for j:=i to n-1 do p[j]:=p[j+1];*

*p[n]:=k*

*end*

*end;{Perm}*

*begin {Permutations}*

*Perm(1)*

*end;*

*begin {Main}*

*readln(n);*

*for i:=1 to n do p[i]:=i;*

*a:=p; {массив a может быть заполнен произвольно}*

*Permutations(n)*

*end.*

Заметим, что в данной программе массив p можно было и не использовать, а переставлять непосредственно элементы массива a.

# **Разбиения множества**

Число разбиений *n*-элементного множества на *k* блоков произвольного размера, но таких, что каждый элемент множества оказывается “приписан” к одному из блоков, выражается числом Стирлинга второго рода *S*(*n*,*k*) [6,7]. Очевидно, что *S*(*n*,*k*) = 0 для *k* > *n*. Если согласиться, что существует только один способ разбиения пустого множества на нулевое число непустых частей, то *S*(0,0) = 1 (именно такая договоренность, как и в случае с факториалом, приводит в дальнейшем к универсальным формулам). Так как при разбиении непустого множества нужна, по крайней мере, одна часть, *S*(*n*,0) = 0 при *n* > 0. Отдельно интересно также рассмотреть случай *k* = 2. Если непустое множество разделили на две непустые части, то в первой части может оказаться любое подмножество исходного множества, за исключением подмножеств, включающих в себя последний элемент множества, а оставшиеся элементы автоматически попадают во вторую часть. Такие подмножества можно выбрать 2*n*-1 – 1 способами, что и соответствует *S*(*n*,2) при *n* > 0.

Для произвольного *k* можно рассуждать так. Последний элемент либо будет представлять из себя отдельный блок в разбиении и тогда оставшиеся элементы можно разбить уже на *k* – 1 частей *S*(*n* – 1,*k* – 1) способами, либо помещаем его в непустой блок. В последнем случае имеется *kS*(*n* – 1,*k*) возможных вариантов, поскольку последний элемент мы можем добавлять в каждый блок разбиения первых *n* - 1элементов на *k* частей. Таким образом

*S*(*n*,*k*) = *S*(*n* – 1, *k* – 1) + *kS*(*n* – 1, *k*), *n* > 0. (5)

Полезными могут оказаться также формулы, связывающие числа Стирлинга с биномиальными коэффициентами, определяющими число сочетаний:



Если же значение *k* теперь не фиксировать и рассмотреть все разбиения *n*-элементного множества, то их количество выражается числом Белла



По формулам (7) можно подсчитать, что в рамках принятых выше допущений можно построить все разбиения множества, состоящего не более чем из 15 элементов (*B*15=1382958545).

Перейдем теперь к рассмотрению способа генерации всех разбиений исходного множества. Прежде всего, следует договориться о том, как обозначать текущее разбиение. Так как в каждом из разбиений участвуют все элементы исходного множества, будем в массиве индексов p записывать в какой блок попадает каждый из элементов в текущем разбиении. Параметр i в рекурсивной процедуре part означает, что на текущем шаге мы именно i-ый элемент будет размещать в каждом из допустимых для него блоков, а j как раз и определяет максимальный номер допустимого блока. После того, как i-ый элемент помещен в один из блоков, рекурсивно решается такая же задача уже для следующего элемента (в данном случае фактически работает универсальная схема перебора с возвратом [8]).

*procedure partition(n : integer; var p:list);*

*procedure part(i, j: integer);*

*var l: integer;*

*begin*

*if i > n then print(n, p) else*

*for l := 1 to j do*

*begin*

*p[i] := l;*

*if l = j then part(i+1, j+1)*

*else part(i+1, j)*

*end*

*end; {part}*

*begin {partition}*

*part(1,1)*

*end;*

Как ни странно, в данном случае процедура print оказывается совсем не тривиальной, если требуется печатать (или анализировать) элементы каждого из блоков разбиения в отдельности. Поэтому приведем возможный вариант ее реализации (как и ранее, распределяли по блокам мы индексы, а печатаем или анализуруем сами элементы исходного массива a):

*procedure print(n:integer; var p:list);*

*var i,j,imax:integer;*

*begin*

*imax:=1;{определяем количество блоков в разбиении}*

*for i:=2 to n do*

*if p[i]>imax then imax:=p[i];*

*for i:=1 to imax do {цикл по блокам}*

*begin*

*for j:=1 to n do*

*if p[j]=i then write(a[j]:4);*

*write(' |') {блок напечатан}*

*end;*

*writeln {разбиение напечатано}*

*end;*

Вложенного цикла можно избежать, если требуется, например, подсчитать сумму элементов в каждом из блоков. Тогда, используя дополнительный массив, мы, просматривая элементы массива a последовательно, будем увеличивать значения суммы для блока, соответствующего рассматриваемому элементу (аналогично операции, осуществляемой в алгоритме сортировки подсчетом).

Если при этом рассматривать массив p как *n*-значное число *n*-ричной системе счисления, то можно ввести понятие лексикографического порядка для разбиений множества и ставить задачи определения номера разбиения и обратную ей. Как и ранее (см. [1-3]), они решаются методом динамического программирования и не используют непосредственную генерацию всех разбиений.

Для полноты рассмотрения данной темы самостоятельно измените процедуру partition так, чтобы она генерировала все разбиения, состоящие не более чем из *k* блоков. После этого напишите процедуру разбиения множества уже на ровно *k* непустых частей.

# Олимпиадные задачи, использующие комбинаторные конфигурации

*Пример* 1. На одном острове Новой Демократии каждый из его жителей организовал партию, которую сам и возглавил. Отметим, что к всеобщему удивлению даже в самой малочисленной партии оказалось не менее двух человек. К сожалению, финансовые трудности не позволили создать парламент, куда вошли бы, как предпологалось по Конституции острова, президенты всех партий. Посовещавшись, Островитяне решили, что будет достаточно, если в парламенте будет хотя бы один член каждой партии. Помогите Островитянам организовать такой, как можно более малочисленный парламент, в котором будут представлены члены всех партий.

Исходные данные: каждая партия и, соответственно, ее президент имеют одинаковый порядковый номер от 1 до *N* (4 ≤ *N* ≤ 150). Вам даны списки всех *N* партий Острова Новой Демократии. Выведите предлагаемый Вами парламент в виде списка номеров его членов. (*Олимпиада стран СНГ*, г. Могилев, 1992 г.)

*Решение*: с теоретической точки зрения, данная задача совпадает с задачей генерации всех подмножеств из множества жителей острова новой демократии. Причем наиболее подходящим кажется первый подход к решению данной задачи, связанный с генераций различных сочетаний, начиная с одного жителя. Для полноты изложения этого подхода, опишем функцию сheck, которую следует применить в данной задаче. Исходные данные следует записать в массив s:array[1..150] of set of 1..150, заполнив каждый из *n* первых элементов этого массива множеством тех партий, в которых состоит тот или иной житель. Тогда функция проверки сочетания из элементов этого массива примет следующий вид:

*function check(var p:list;k:integer): boolean;*

*var i:integer; ss:set of 1..150;*

*begin*

*ss:=[];*

*for i:=1 to k do ss:=ss+s[p[i]];*

*check:=(ss=[1..n])*

*end;*

Как видно из описания функции, использование типа “множество”, позволяет не только упростить данную программу, но и существенно ускорить ее выполнение, по сравнению с работой с массивами. Однако большая размерность данной задачи не позволяет считать приведенное решение удовлетворительным во всех случаях. Так, уже для *n* = 100, перебор всех сочетаний из 4-х и менее жителей приводит к рассмотрению около 4-х миллионов вариантов. Для построения кода, пригодного для решения данной задачи при любых входных данных, несколько изменим описание массива s:

*s: array[1..150] of*

*record name,number:integer;*

*partys: set of 1..150*

*end;*

Здесь поле partys совпадает по смыслу с первоначальным описанием массива s, поле name cоответствует номеру (то есть фактически имени) жителя и первоначально данное поле следует заполнить числами от 1 до *n* cогласно индексу элемента в массиве записей, и поле number соответствует количеству элементов во множестве из поля partys, то есть имеет смысл сразу подсчитать, в каком количестве партий состоит тот или иной житель. Теперь следует отсортировать массив s по убыванию значений поля number. Верхнюю оценку для числа членов парламента (*kmax*) подсчитаем, построив приближенное решение данной задачи следующим образом: во-первых, включим в парламент жителя, состоящего в максимальном количестве партий, затем исключим эти партии из остальных множеств и заново найдем в оставшемся массиве элемент с максимальным значением поля number (уже пересчитанного) и включим его в парламент, и так далее, до тех пор, пока сумма значений пересчитанных полей number у жителей, включенных в парламент, не будет равна *n*. Найденное количество членов парламента и будет *kmax*.

Теперь следует рассматривать сочетания из (*kmax* – 1) элемента, затем из (*kmax* – 2) и т. д. элементов. Если для сочетаний из какого-то рассматриваемого количества элементов *k*, решение найдено не будет, то это означает, что точным является решение с количеством членов парламента *k*+1. Так как массив s упорядочен, то, если решение для того или иного значения *k* существует, то, скорее всего, оно будет найдено при рассмотрении в лексикографическом порядке первых же сочетаний по *k* элементов. Поэтому временная трудоемкость в переборе возникает, только если решения c данным значением *k* не существует. В такой ситуации можно воспользоваться следующим “ненаучным” приемом: на поиск решения для каждого *k*, меньшего, чем *kmax* отведем фиксированное количество времени, скажем, 2-3 секунды (точнее данное время стоит определить экспериментальным путем). Если за отведенное время решение не найдено, то следует считать полный перебор невозможным и закончить выполнение программы. В любом случае, мы будем иметь какое-либо решение исходной задачи: точное или приближенное, то есть, возможно, содержащее больше членов парламента, чем минимально возможно. Однако, на практике такой подход почти всегда приводит к точному решению, в силу перебора “с предпочтением”, проводимого для каждого *k* (невозможность проведения полного перебора для какого-либо *k* на практике соответствует отсутствию решения для данного *k*).

*Пример* 2. Дан автобусный билет с номером, состоящим из *N* цифр. Расставить между цифрами знаки арифметических операций '+', '-', '\*', '/' (целочисленное деление) и скобки таким образом, чтобы значение полученного выражения было равно 100. Можно образовывать многозначные числа из стоящих рядом цифр. Выражение должно быть корректным с точки зрения арифметики. Допустимы лишние скобки, не нарушающие корректности выражения. При вычислениях используется стандартный приоритет операций, число цифр *N* в номере билета не больше 6. (*5-ая Всероссийская олимпиада по информатике*, г. Троицк, 1993 г.)

*Решение*. для построения универсального алгоритма решения данной задачи будем считать слияние двух соседних цифр одной из операций. Тогда между каждой парой соседних цифр может стоять одна из 5 операций. Для *N* цифр получаем 5*N*-1 различных вариантов расстановки операций. Перебирать все варианты расстановки операций удобнее всего с помощью рассмотрения всех чисел в 5-ричной системе счисления, состоящих не более чем из *N* – 1 цифры, то есть для *N* = 6 от 00000 до 44444. Для перебора таких чисел необходимо написать процедуру прибавления 1 в 5-ричной системе счисления. Для каждого из вариантов расстановки операций перейдем от исходного массива из *N* цифр билета, к массиву из *К* чисел, где *K* = (*N* – количество операций слияния цифр в рассматриваемом варианте). Теперь мы должны рассмотреть все перестановки из *K* – 1 арифметической операции в данном варианте. Каждая перестановка соответствует одному из порядков выполнения арифметических операций. Так, для 4-х чисел, перестановка номеров операций 3, 1, 2 означает, что сначала нужно выполнить арифметическое действие между 3-м и 4-м числом, затем между 1-м и 2-м и затем оставшееся. Если результат выполнения действий данного варианта в порядке, соответствующем текущей перестановке, равен искомому числу 100, то задача решена и можно перейти к печати результата. Для данной задачи возможны и более эффективные решения, но в силу ее небольшой размерности, комбинаторный перебор оказывается вполне приемлемым.

*Пример 3*. Губернатор одной из областей заключил с фирмой "*HerNet*" контракт на подключение всех городов области к компьютерной сети. Сеть создается следующим образом: в области устанавливается несколько станций спутниковой связи, и затем от каждого города прокладывается кабель до одной из станций. Технология, используемая компанией требует при увеличении расстояния увеличения толщины кабеля. Таким образом, стоимость кабеля, соединяющего город со станцией, при используемой компанией технологии будет равна *kL*2, где *L* - расстояние от города до станции, а *k* - некий коэффициент. Вам требуется написать программу, определяющую минимальные затраты компании на установку сети.

Входные данные. Во входном файле записано число *n* (1 ≤ *n* ≤ 10) - количество городов в области. Затем идут положительные вещественные числа: *k* - коэффициент стоимости кабеля и *P* - стоимость постройки одной станции. Далее следует *n* пар вещественных чисел, задающих координаты городов на плоскости.

Выходные данные. В первую строку выходного файла нужно вывести минимальные затраты на установку сети (с тремя знаками после десятичной точки), во вторую - количество устанавливаемых станций. Далее вывести координаты станций (с тремя знаками после десятичной точки), а затем - список из *n* целых чисел, в котором *i*-ое число задает номер станции, к которой будет подключен *i*-ый город. (*Кировский командный турнир по программированию*, 2000 г.)

*Решение*. В силу небольшой размерности мы можем рассмотреть все возможные варианты разбиения городов на группы, подразумевая что для каждой группы будет установлена своя станция, причем оптимальным образом (найти оптимальное местонахождение станции для одной группы городов можно по формуле, аналогичной формуле нахождения центра масс). Затем нужно из всех разбиений выбрать то, для которого общая сумма затрат на установку сети будет минимальной.

# **Заключение**

Перечисленные подзадачи в программировании обычно рассматривают для следующих комбинаторных конфигураций: перестановки элементов множества, подмножества множества, сочетания из *n* элементов множества по *k* элементов (*k*-элементные подмножества множества, состоящего из *n*≥*k* элементов), размещения (упорядоченные подмножества множества, то есть отличающиеся не только составом элементов, но и порядком элементов в них), разбиения множества (множество разбивается на подмножества произвольного размера так, что каждый элемент исходного множества содержится ровно в одном подмножестве), разбиения натуральных чисел на слагаемые, правильные скобочные последовательности (различные правильные взаимные расположения *n* пар открывающихся и закрывающихся скобок).

Большинство указанных конфигураций были подробно рассмотрены в [1-3]. Однако при генерации различных конфигураций использовались в основном нерекурсивные алгоритмы. Опытные же участники олимпиад в подобных случаях при программировании используют в основном именно рекурсию, с помощью которой решение рассматриваемых задач зачастую можно записать более кратко и прозрачно. Поэтому для полноты изложения данной темы приведем ряд рекурсивных комбинаторных алгоритмов и рассмотрим особенности применения рекурсии в комбинаторике.

**Литература**

1. *Окулов С.М.* Перестановки. “Информатика”, №7, 2000.
2. *Окулов С.M.* Комбинаторные задачи. “Информатика”, №10, 13, 2000.
3. *Усов Б.Б.* Комбинаторные задачи. “Информатика”, №39, 2000.
4. *Кормен Т., Лейзерсон Ч., Ривест Р.* Алгоритмы. Построение и анализ. М.: МЦНМО, 2000.
5. *Брудно A.Л., Каплан Л.И.* Московские олимпиады по программированию. М.: Наука, 1990.
6. *Кнут Д.* Конкретная математика. Основание информатики. М.: “Мир”, 1998.
7. *Липский В.* Комбинаторика для программистов. М.: “Мир”, 1988.
8. *Андреева Е.В.* Еще раз о задачах на полный перебор вариантов. “Информатика”, №45, 2000.