**Лекция 2**

**КОМБИНАТОРНЫЕ МЕТОДЫ РЕШЕНИЯ ОПТИМИЗАЦИОННЫХ ЗАДАЧ**

1:

1. **Генерация подмножеств заданного множества**:

- разработка генератора подмножеств на С++;

- решение задачи о рюкзаке.

1. **Генерация сочетаний:**

- разработка генератора сочетаний на С++;

- решение задачи об оптимальной загрузке.

1. **Генерация перестановок:**

- разработка генератора перестановок на С++;

- решение задачи о коммивояжере.

1. **Генерация размещений:**

- разработка генератора сочетаний на С++;

- решение задачи об оптимальной загрузке (с центровкой)

**Особенность алгоритмов:**

1. Невозможно использовать для задач большой размерности.
2. Применяются тогда, когда требуется:

- точное решение или достаточное решение;

- размерность задачи небольшая.

1. Сложность этих алгоритмов:

- является верхней оценкой сложности решения задач;

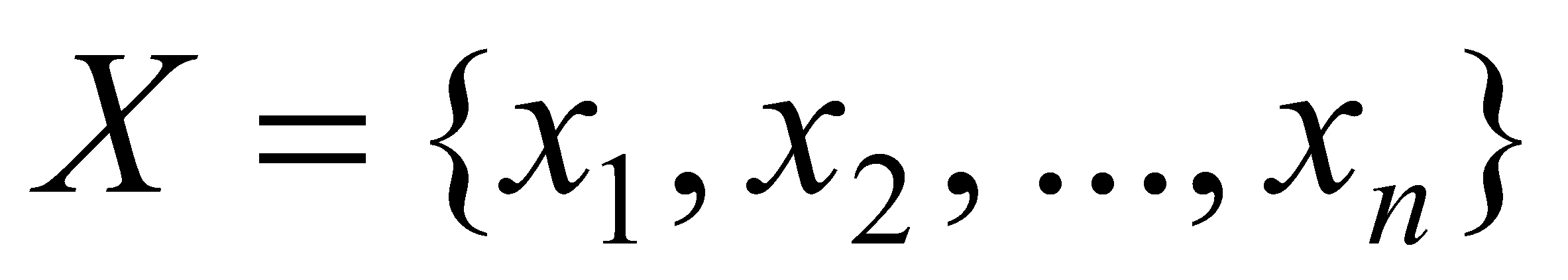
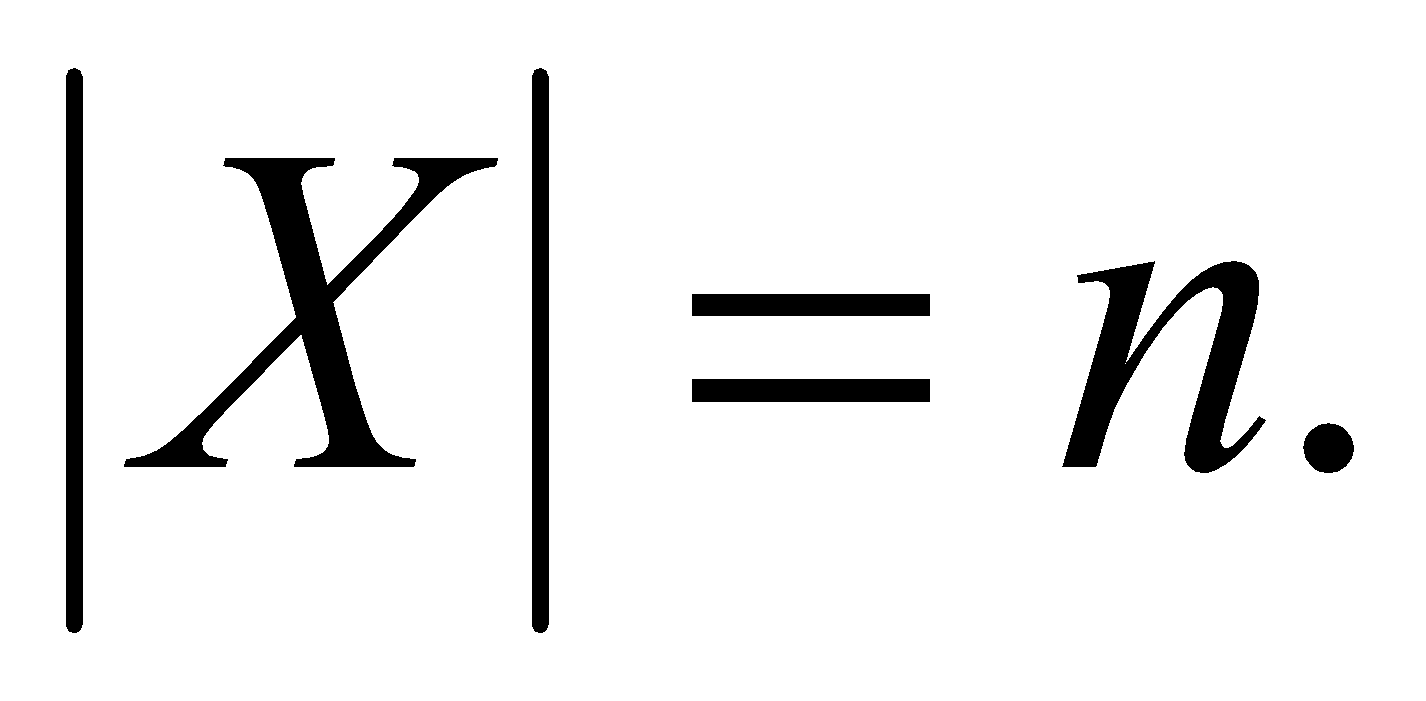
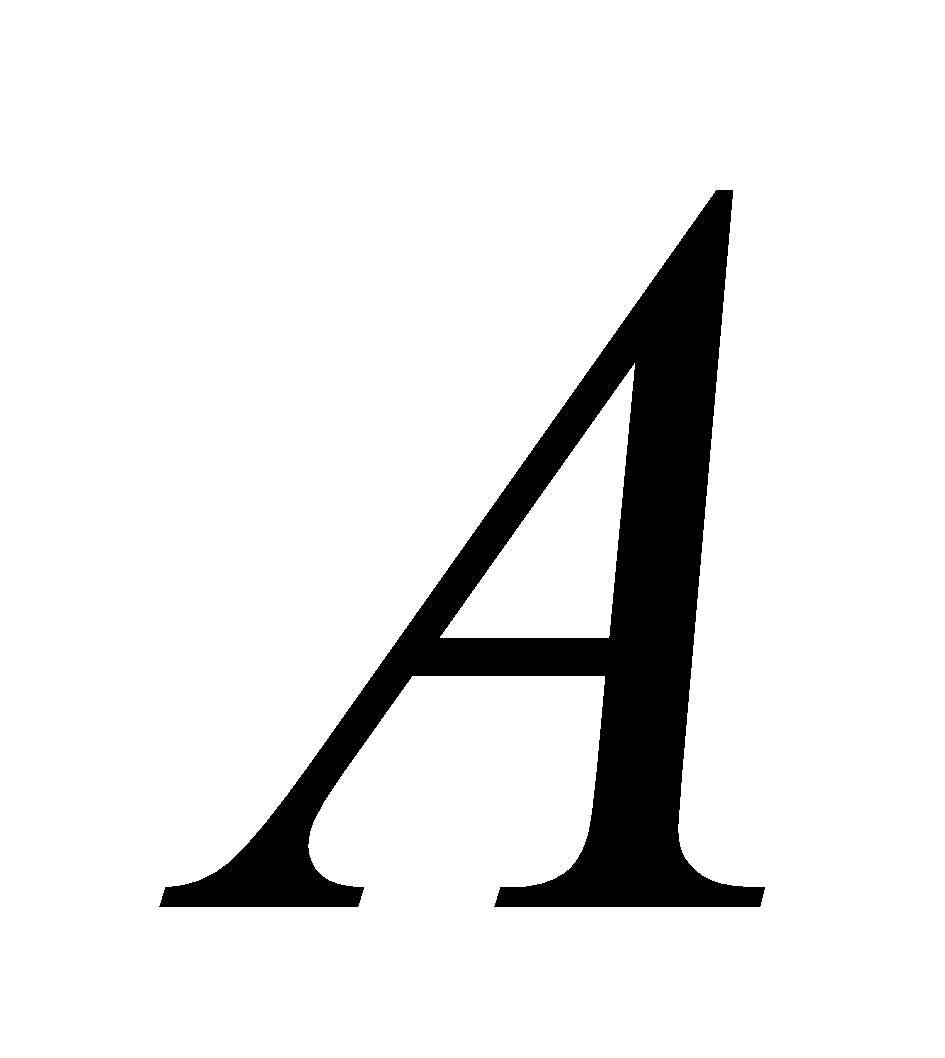
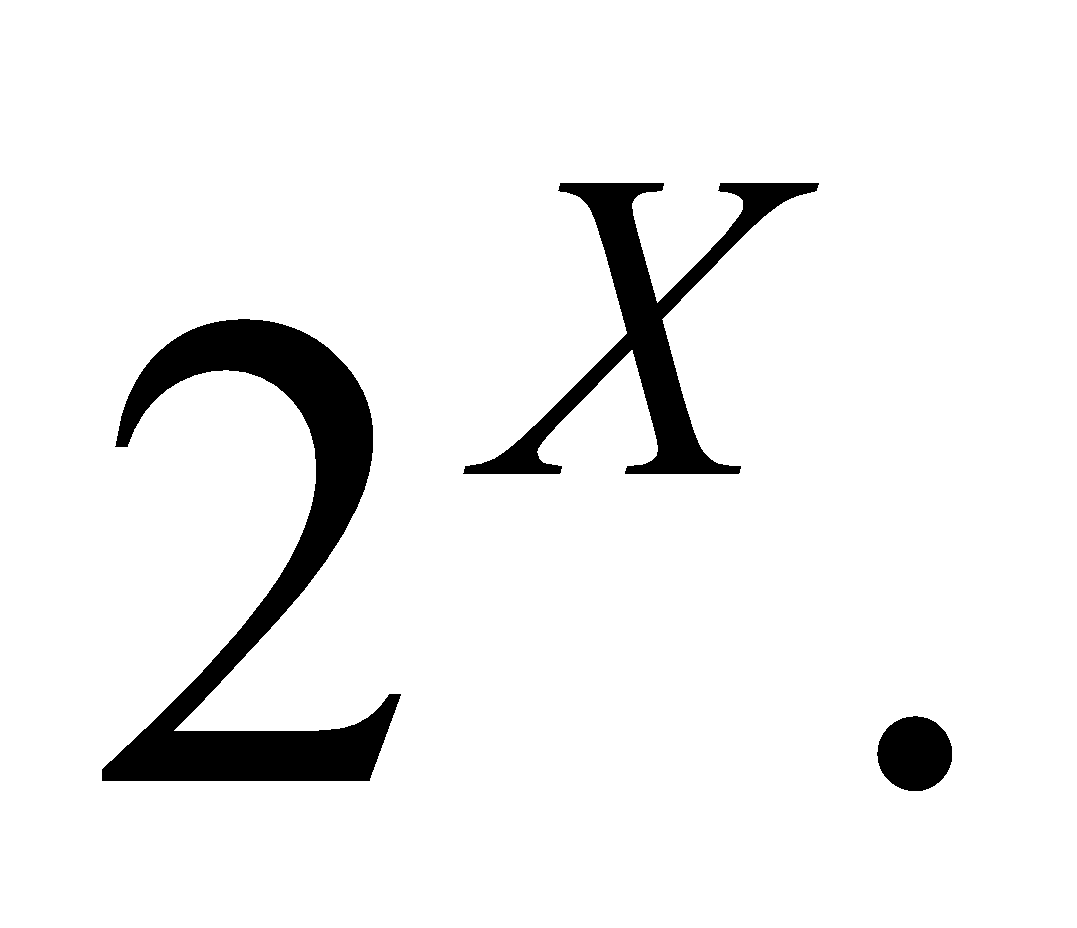
- не зависит от данных;

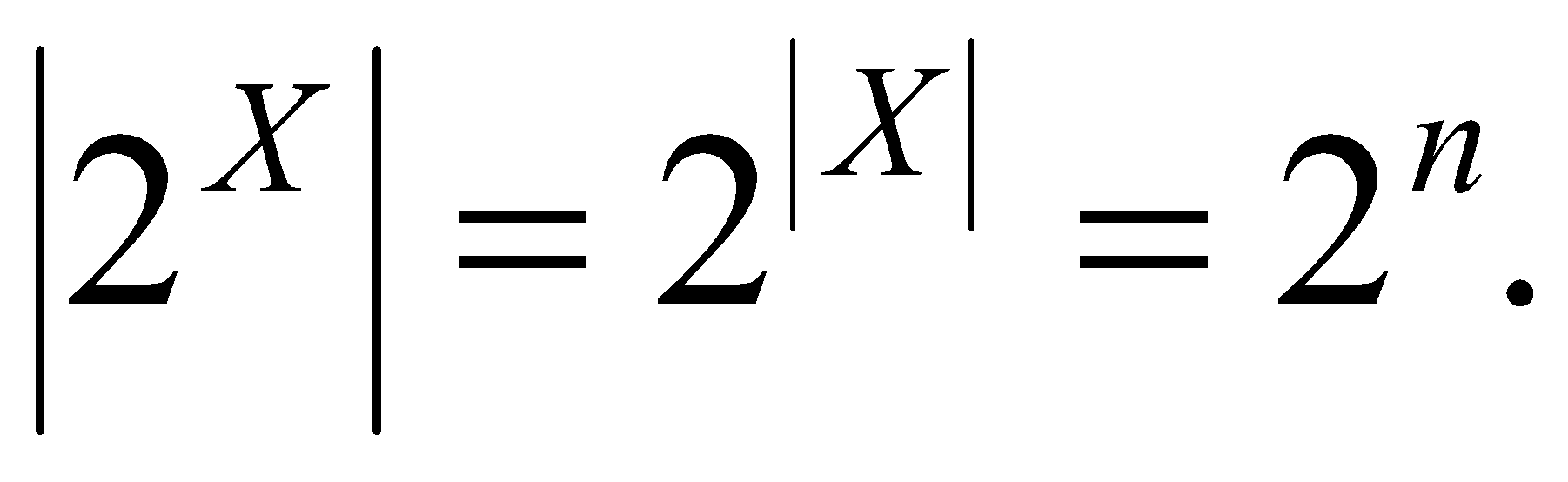
- улучшение возможно только в статистическом смысле.

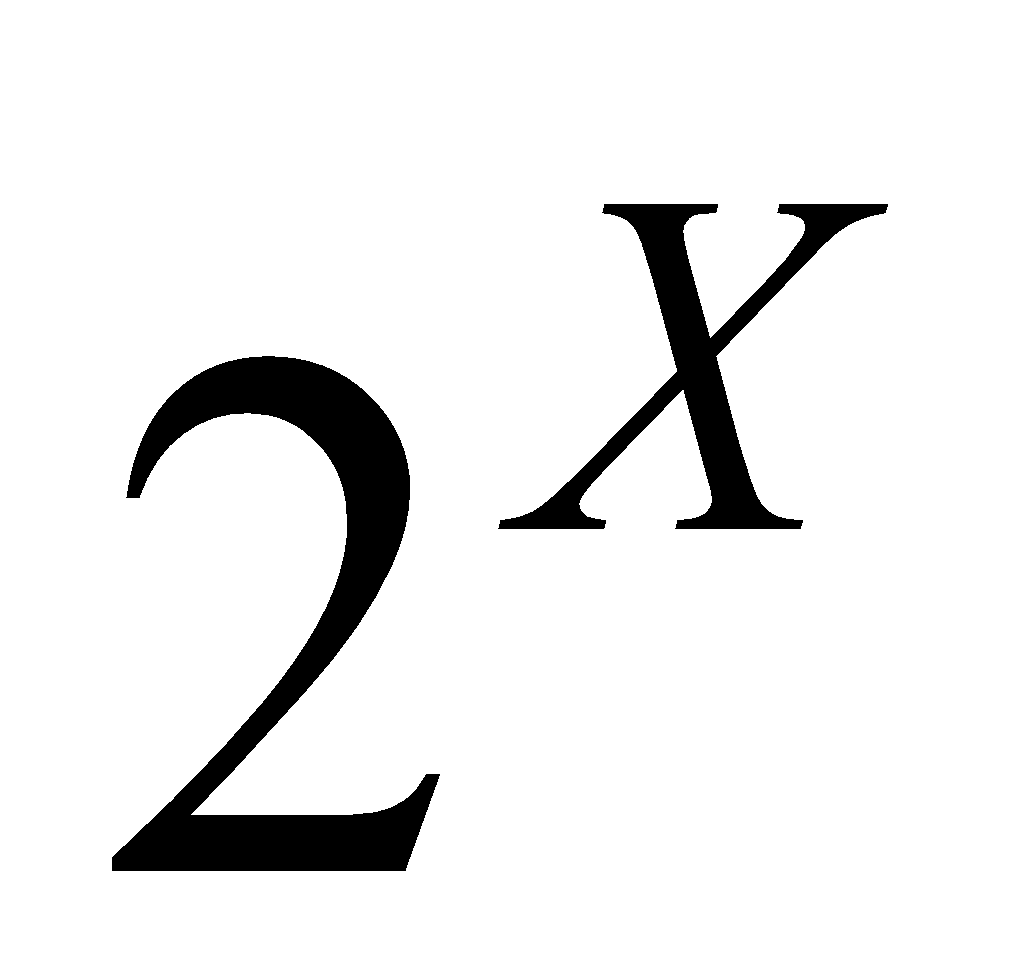
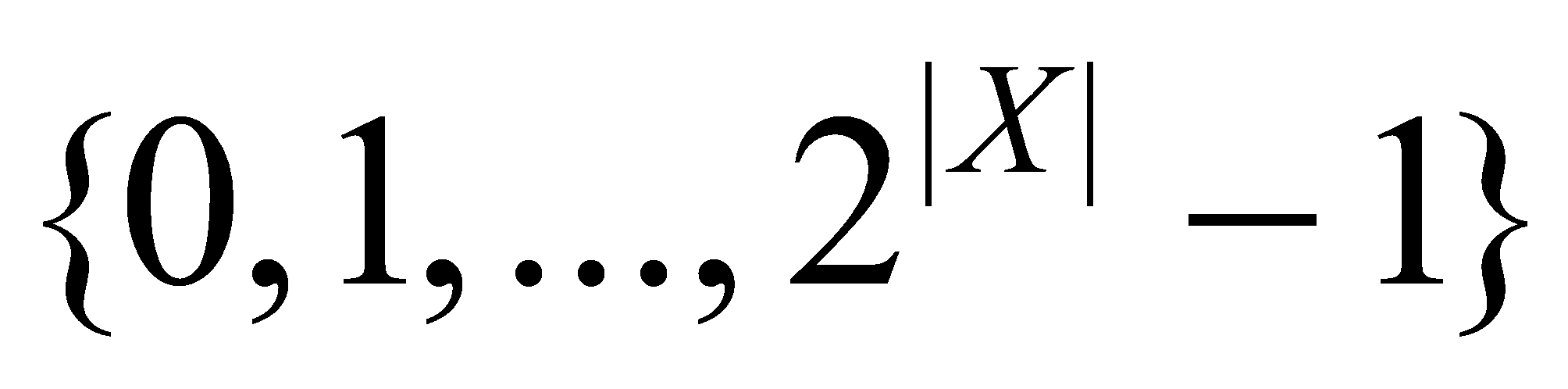
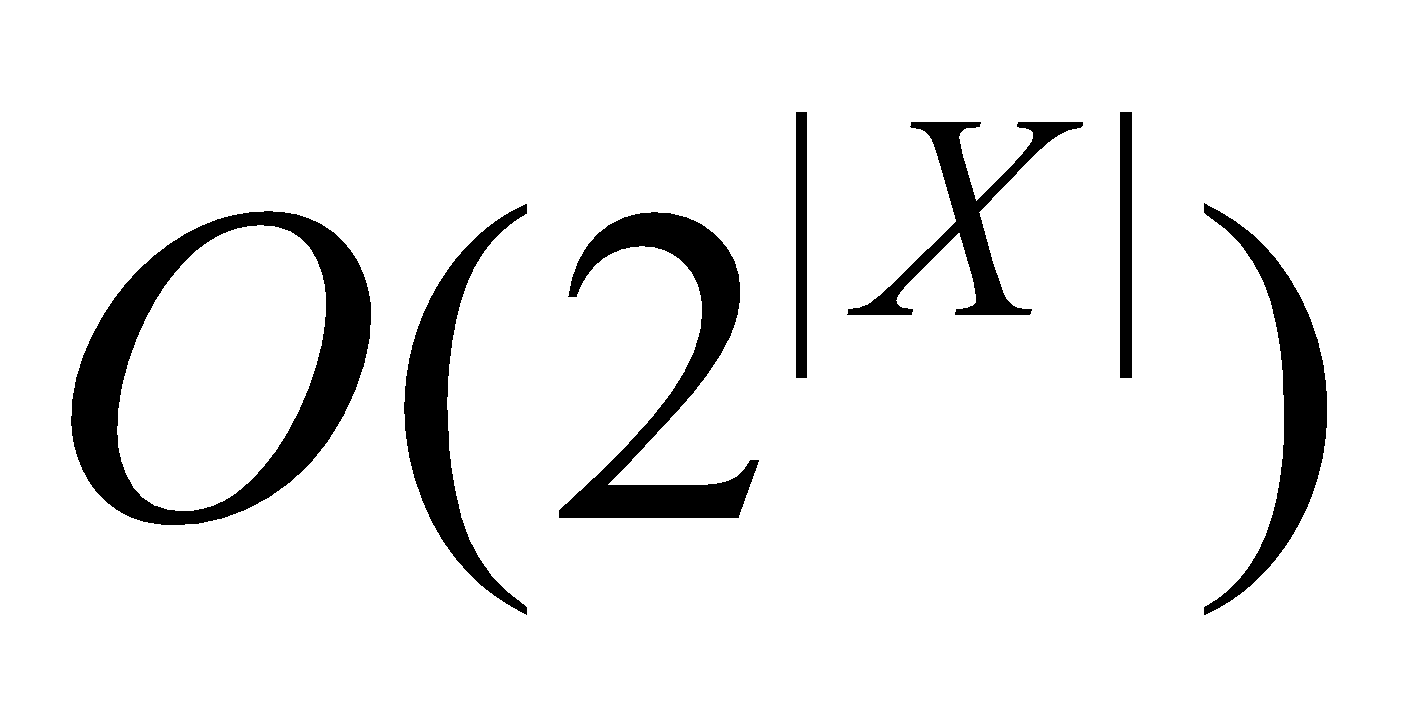
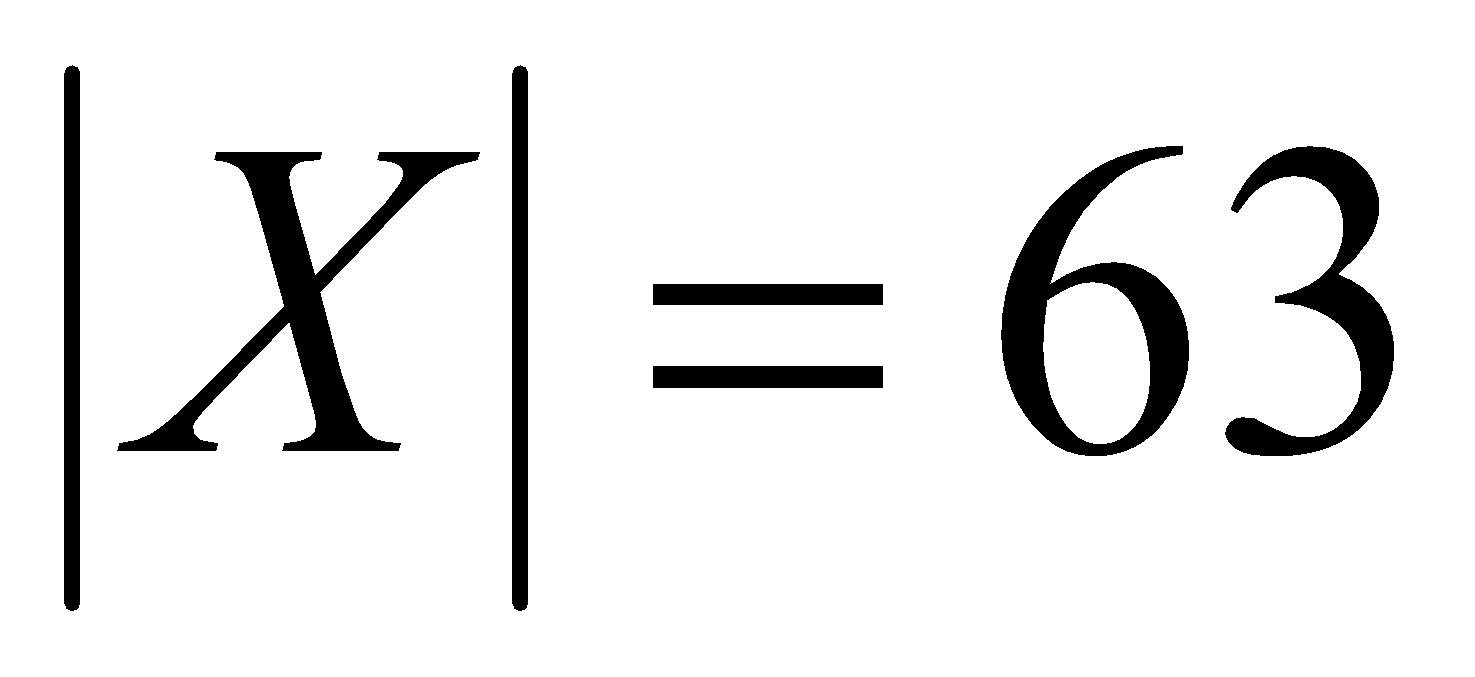
2:

***Комбинаторный анализ (комбинаторика, комбинаторная математика)*** – раздел математики, посвященный решению задач выбора и расположения элементов некоторого, обычно конечного, множества в соответствии с заданными правилами.

1. **Генерация подмножеств заданного множества**

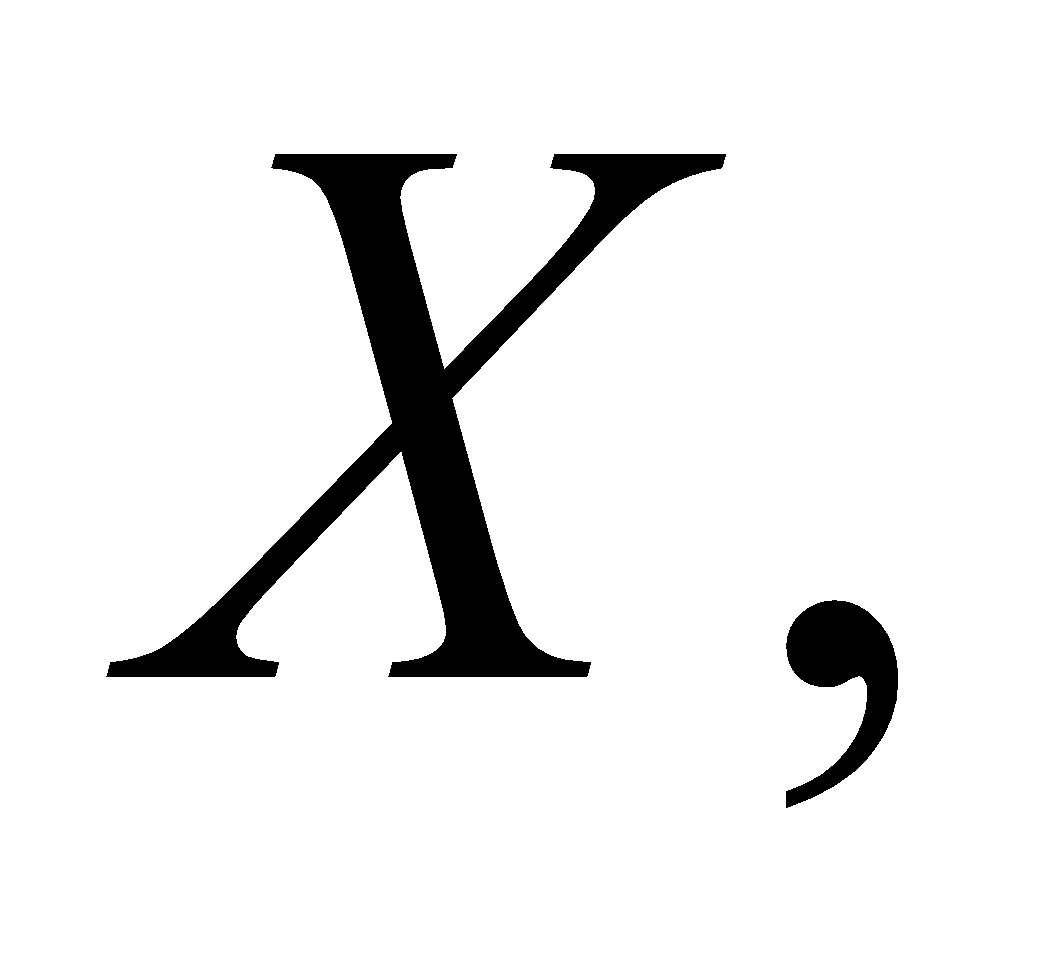
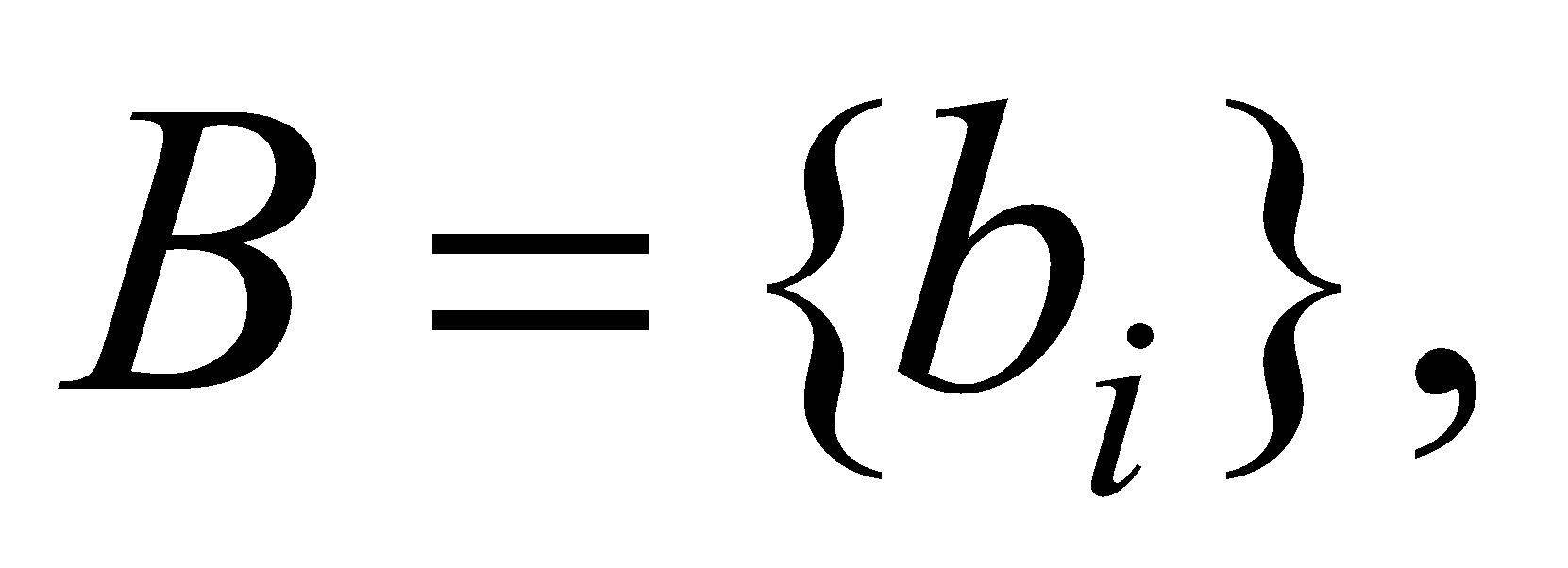
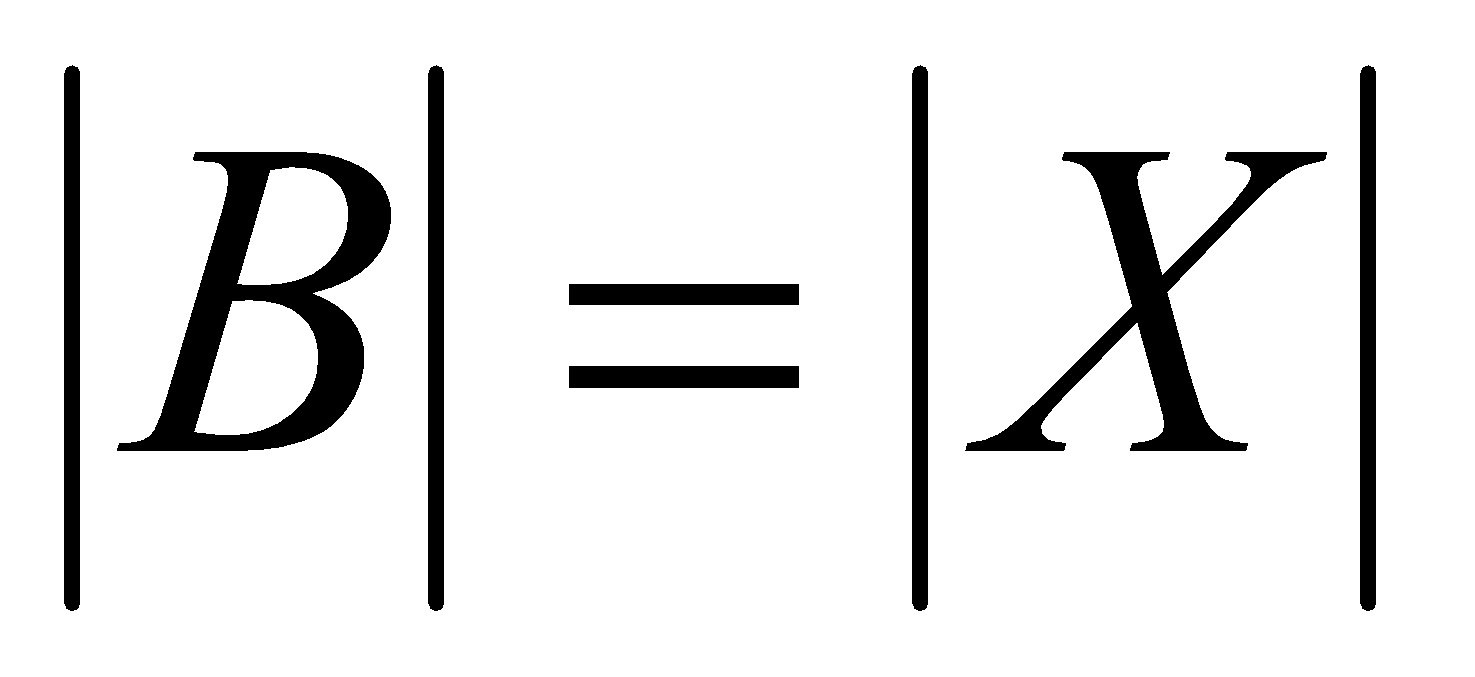
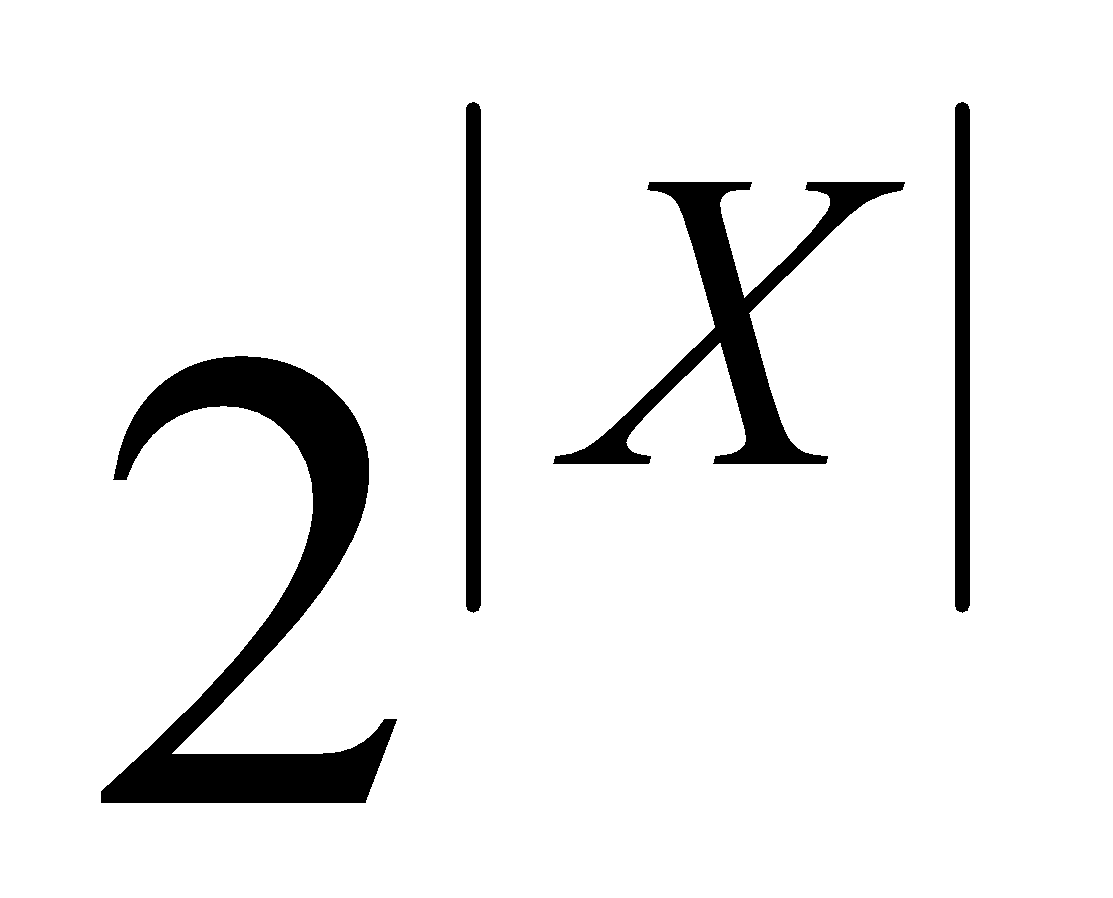
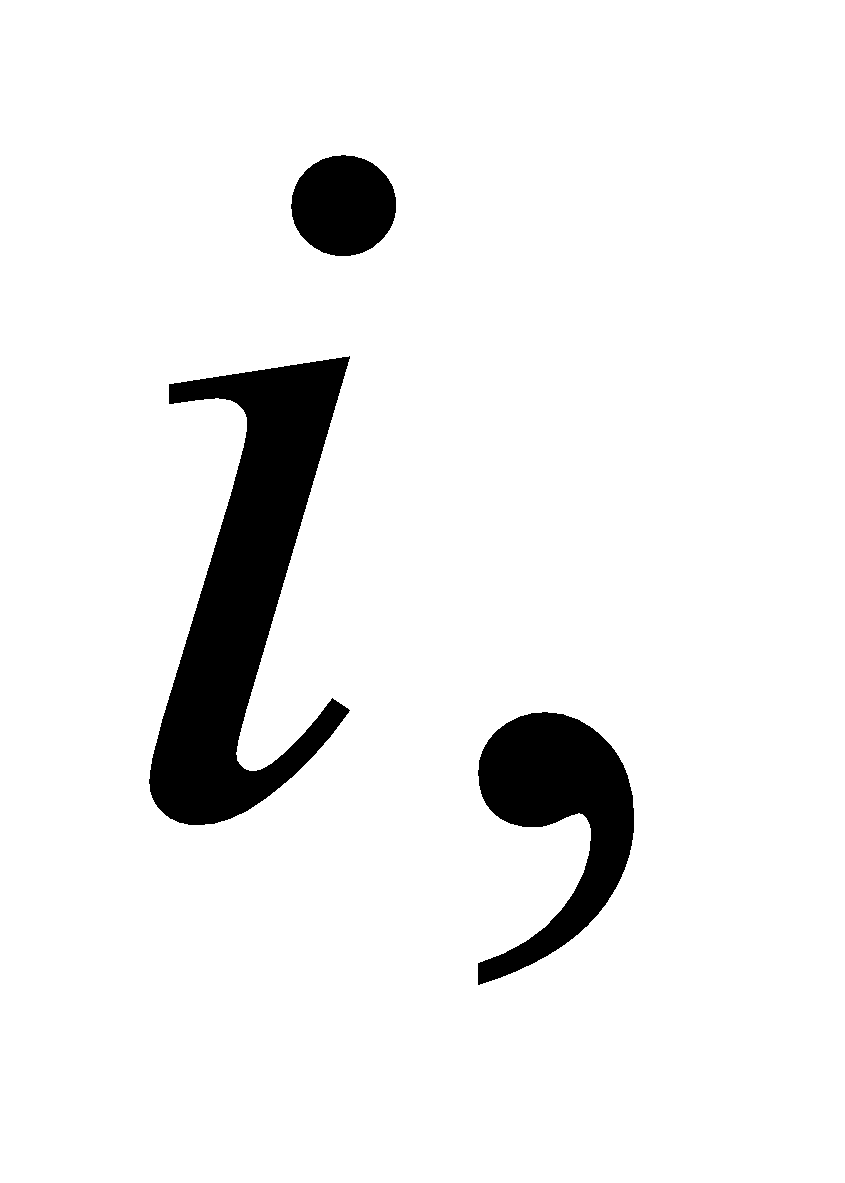
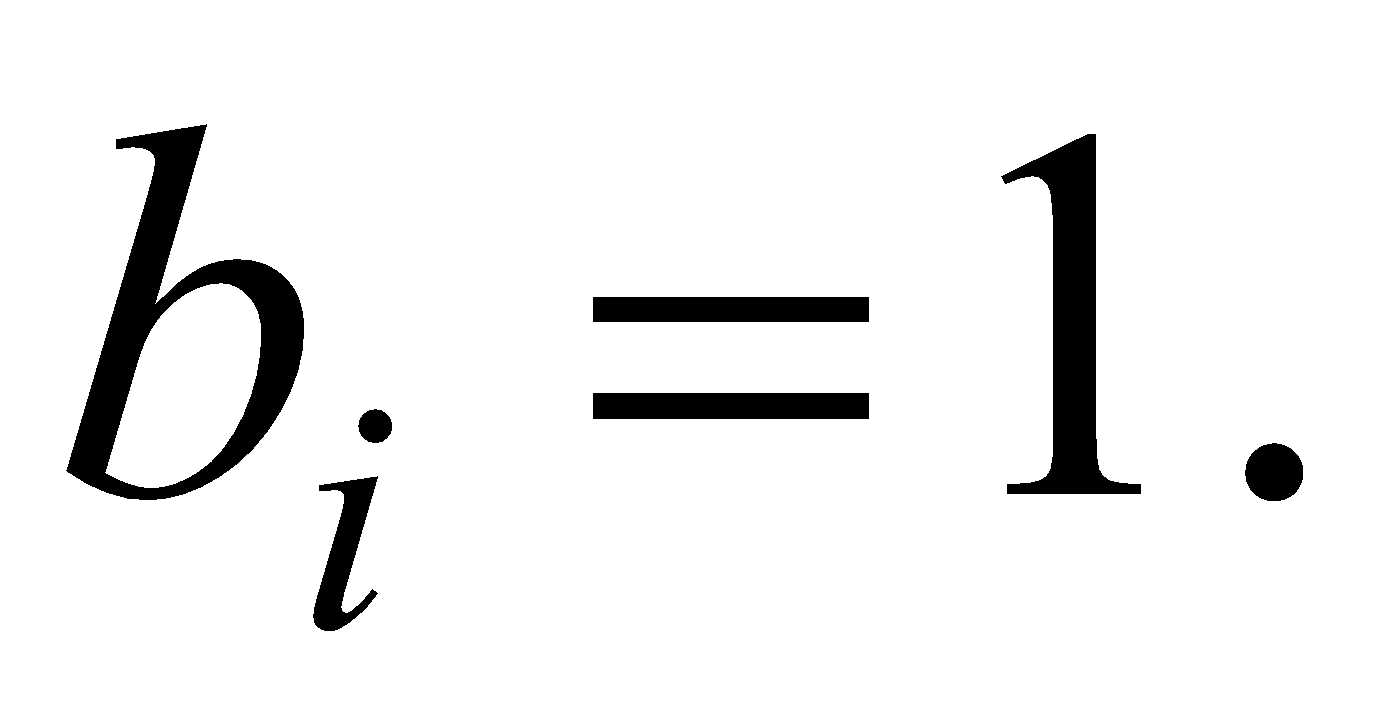
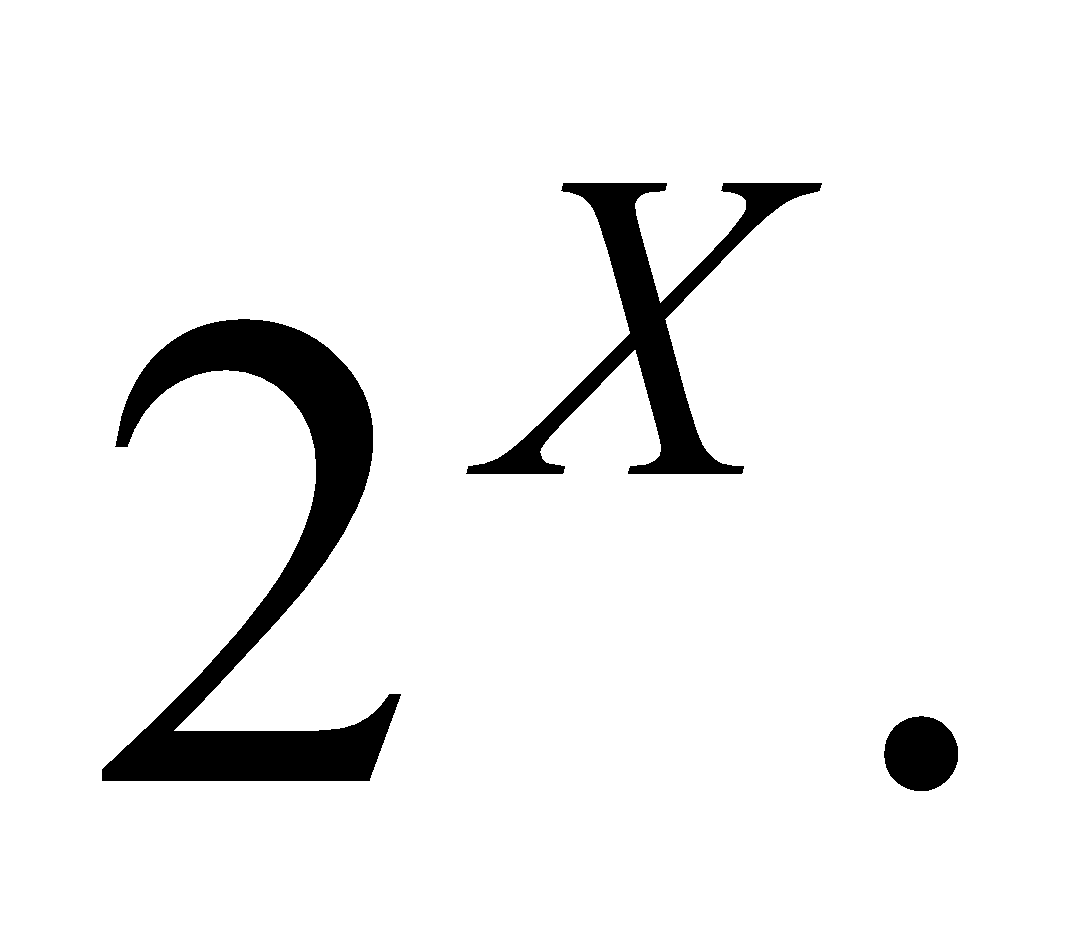
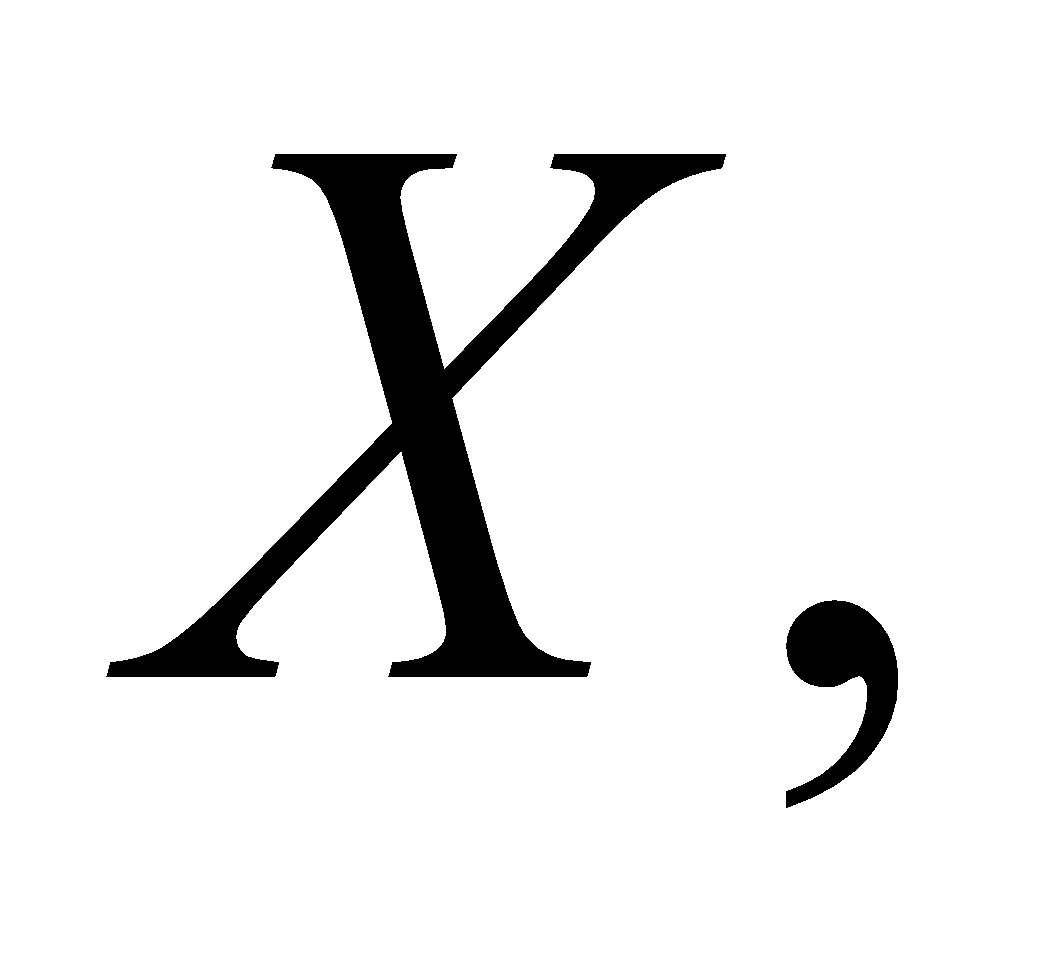
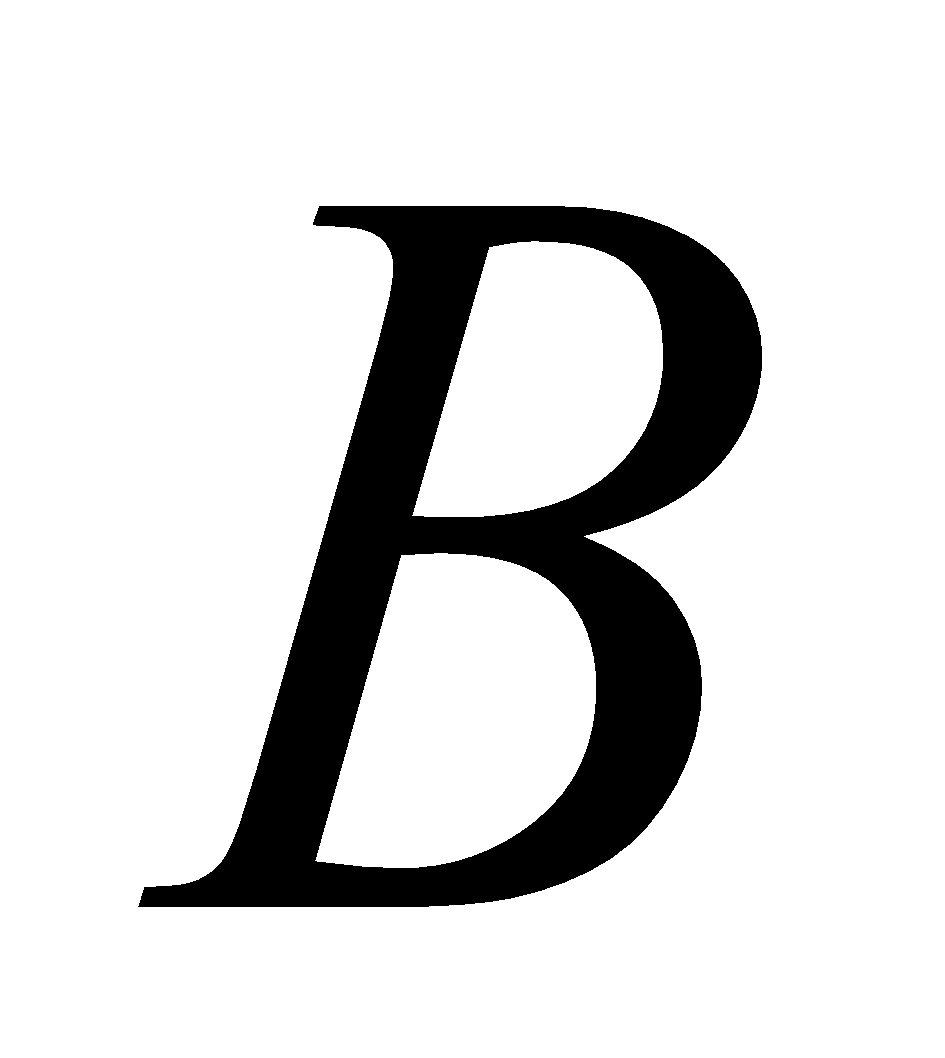
Пусть  – конечное множество мощности  **Множество всех подмножеств** множества  называется ***булеаном***  и обозначается  Количество элементов булеана множества  вычисляется по формуле

 (1)

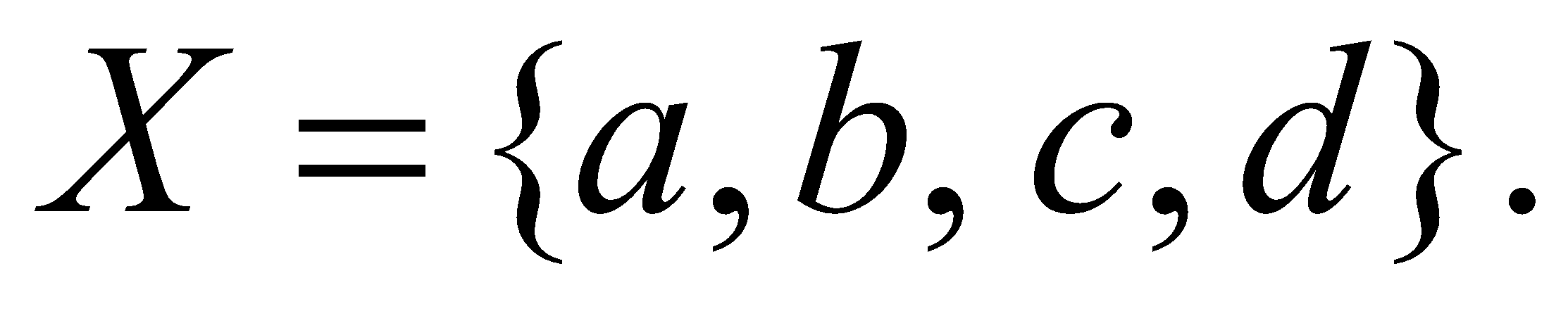
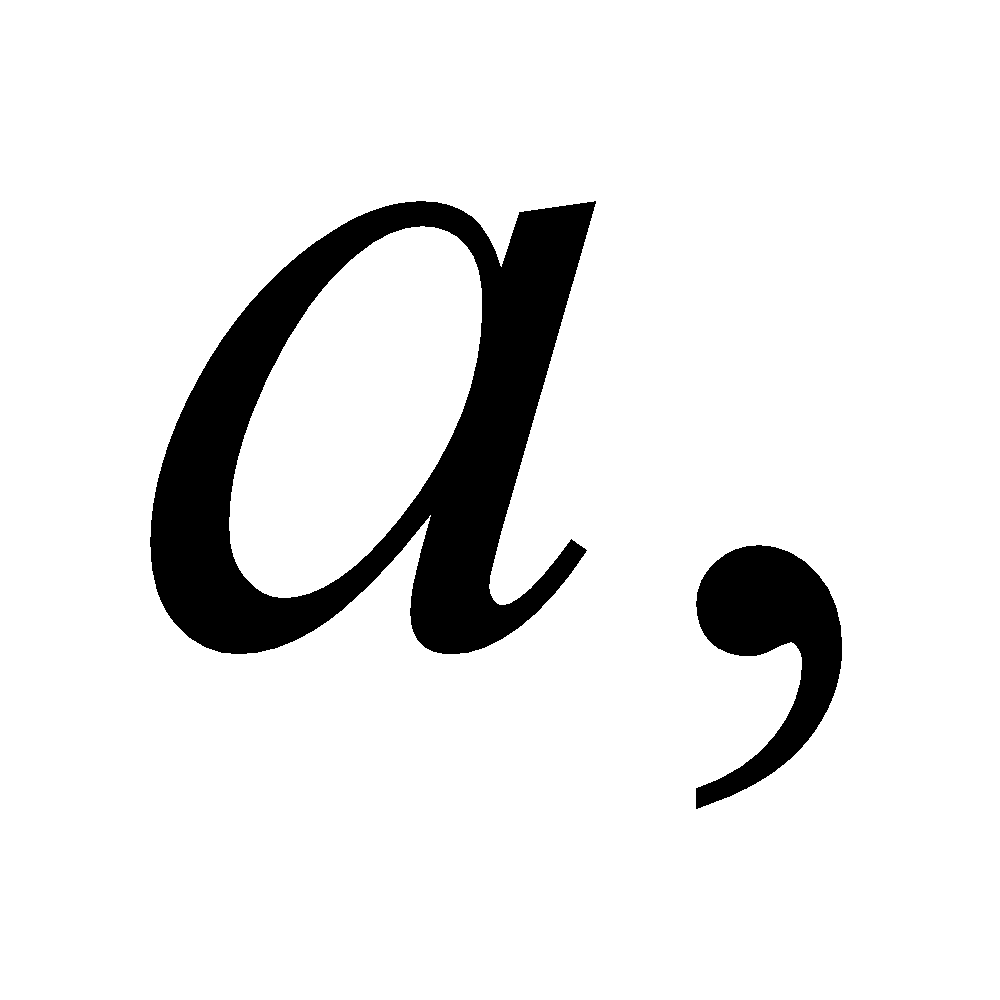
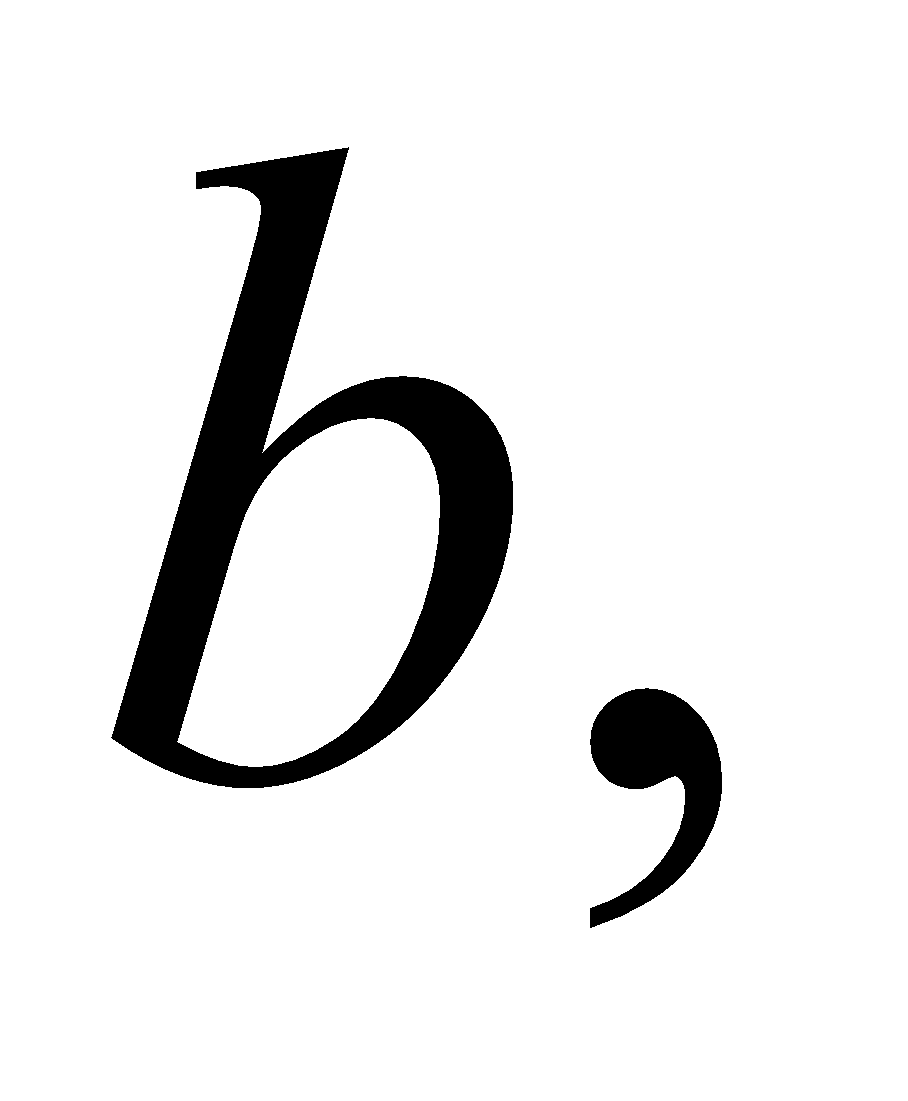
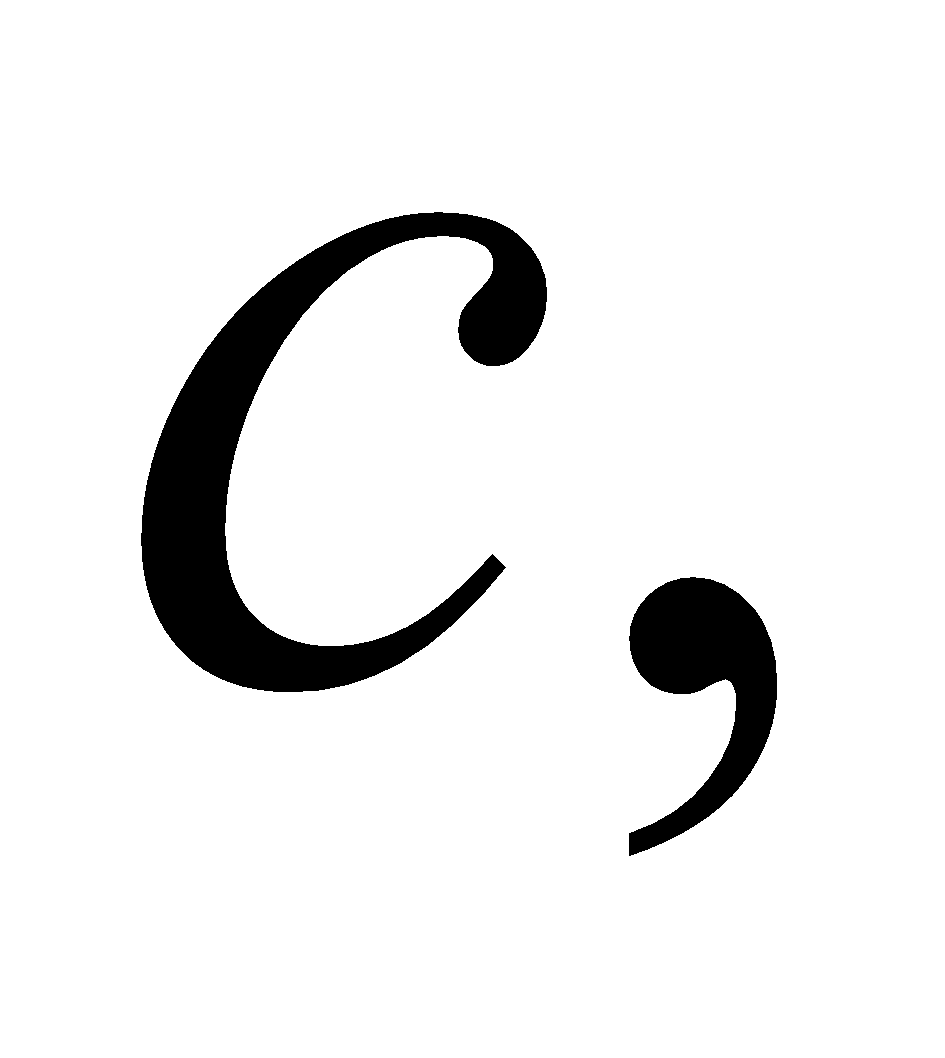
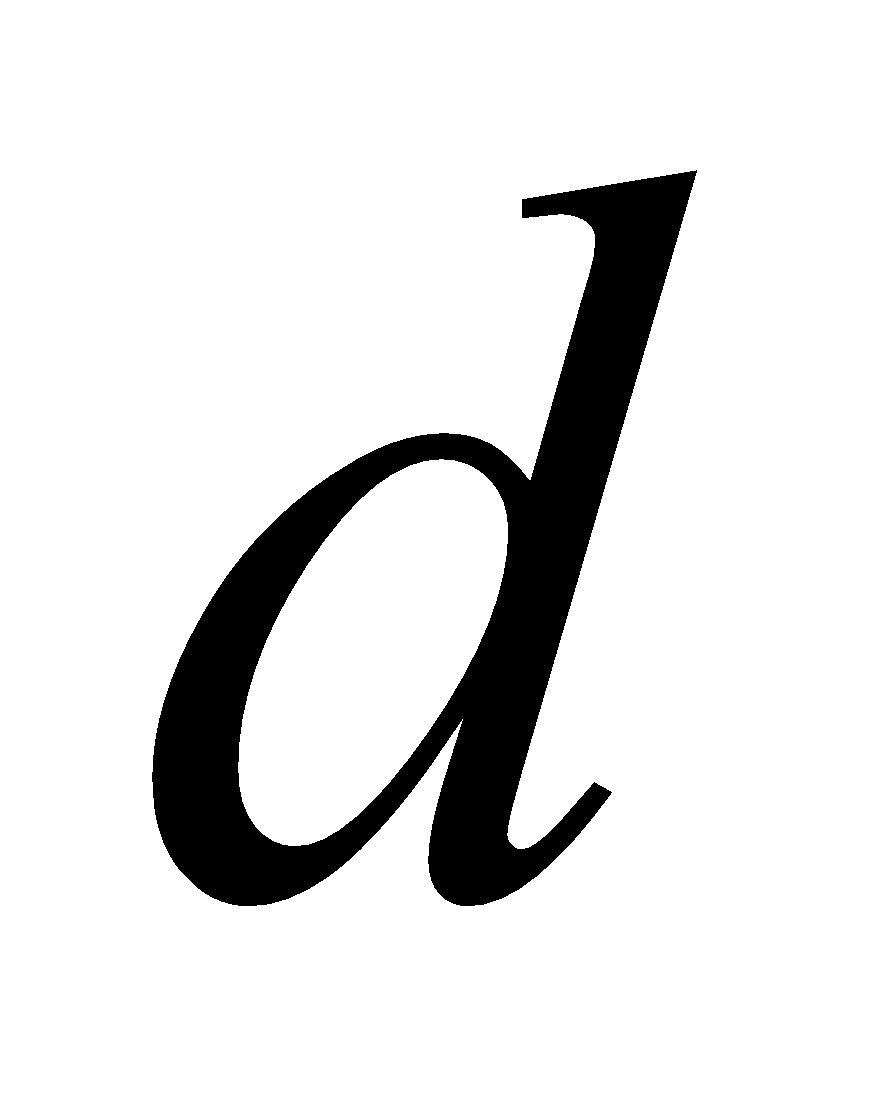
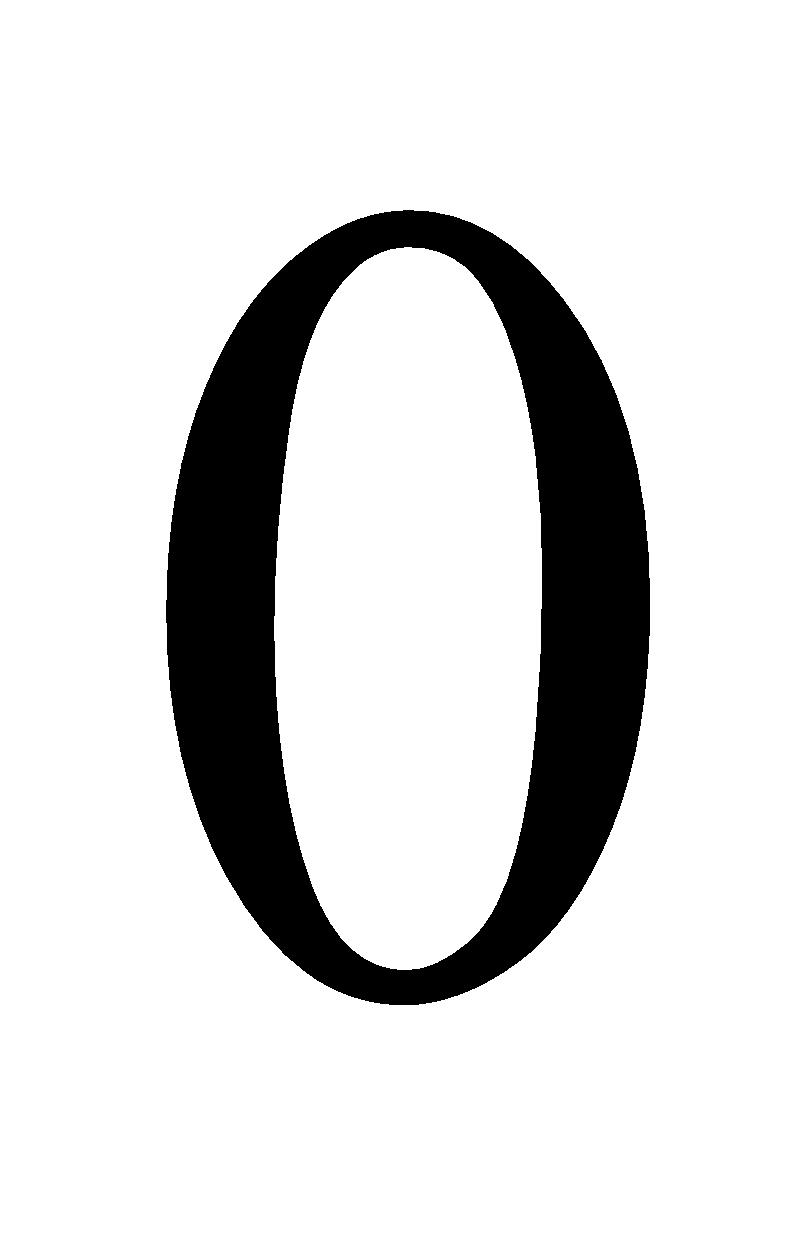
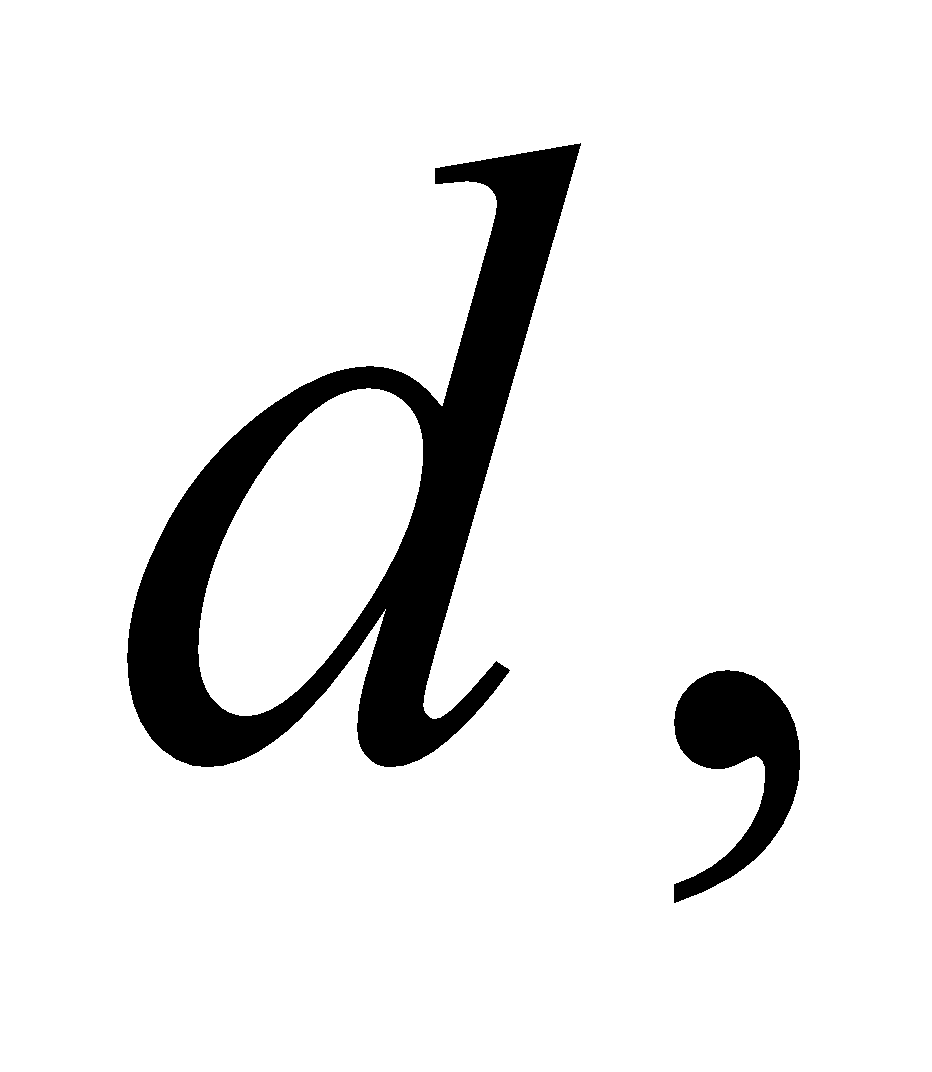
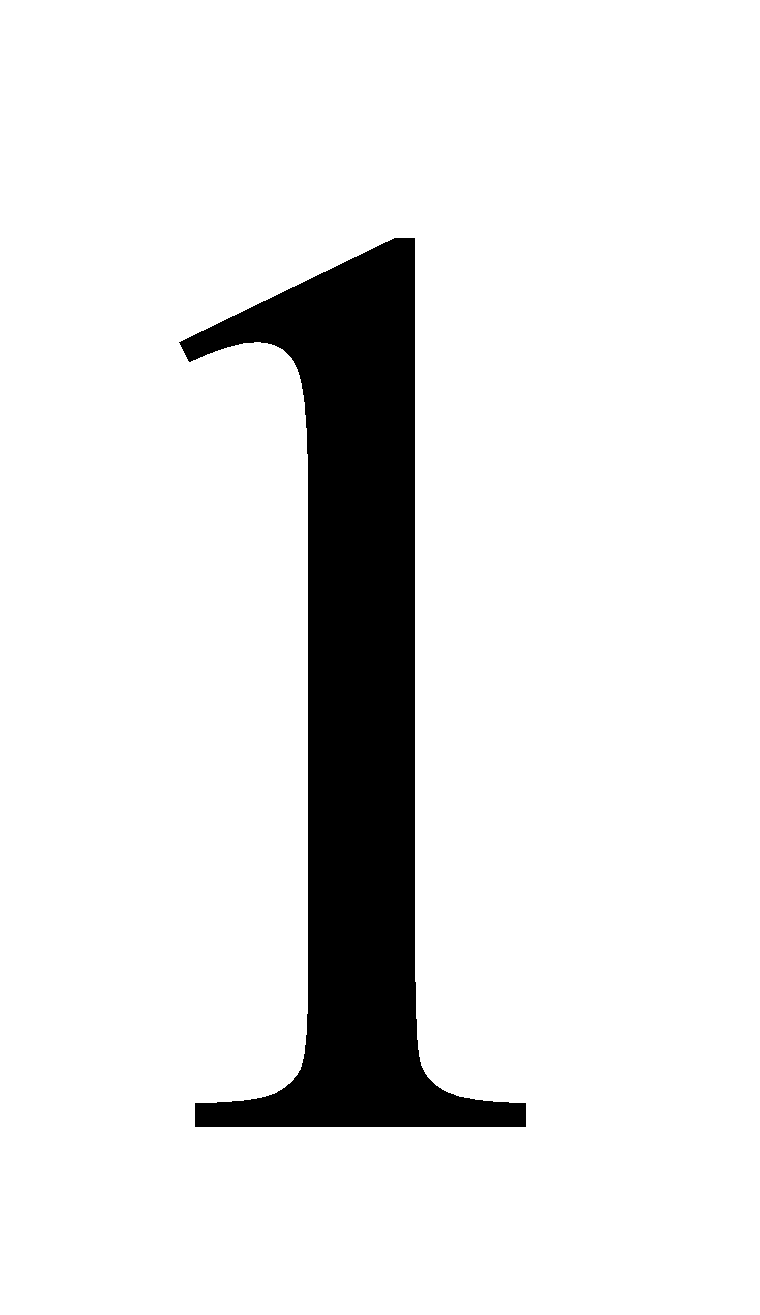
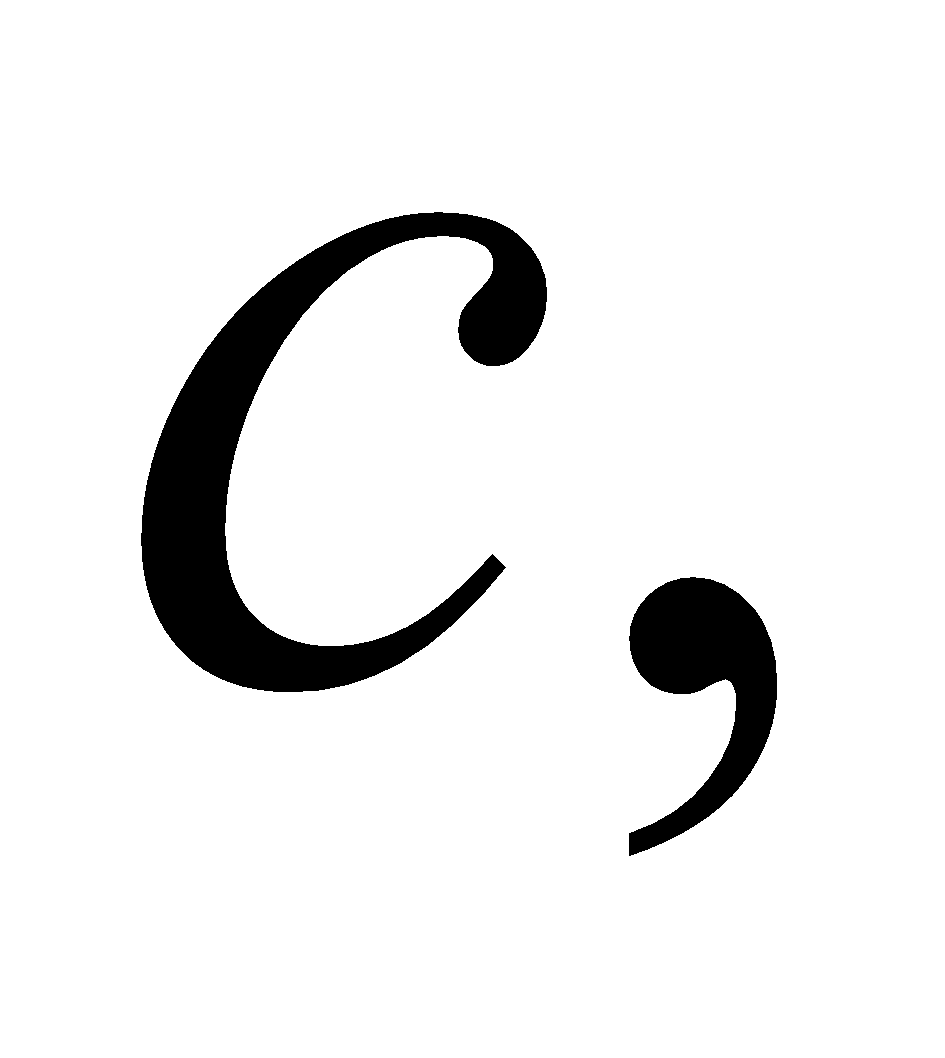
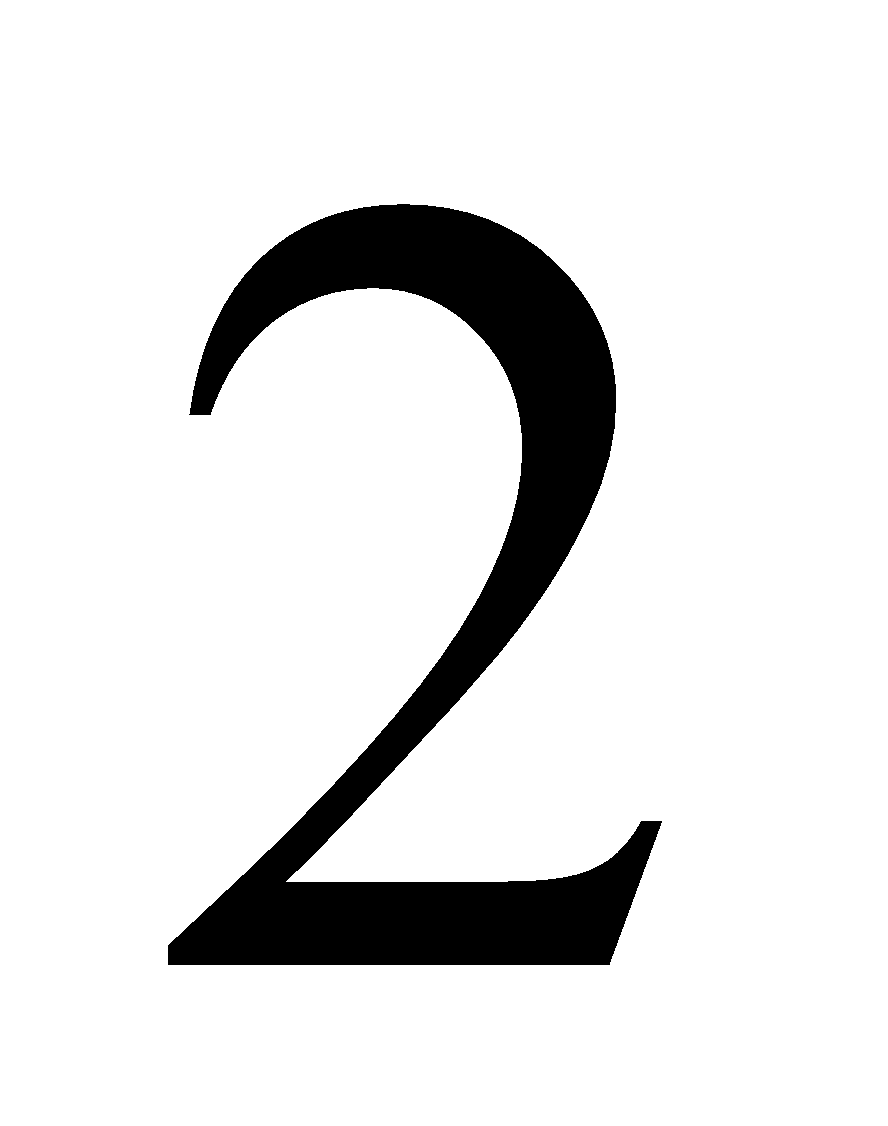
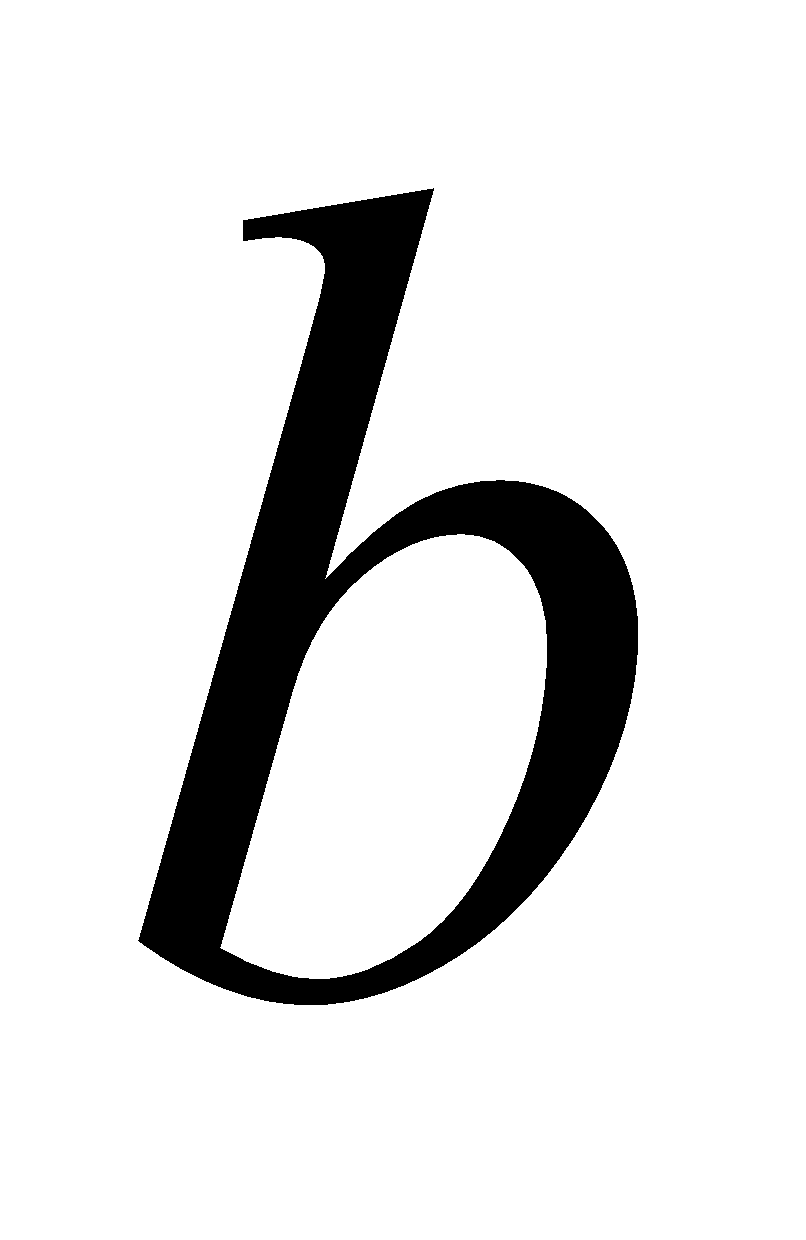
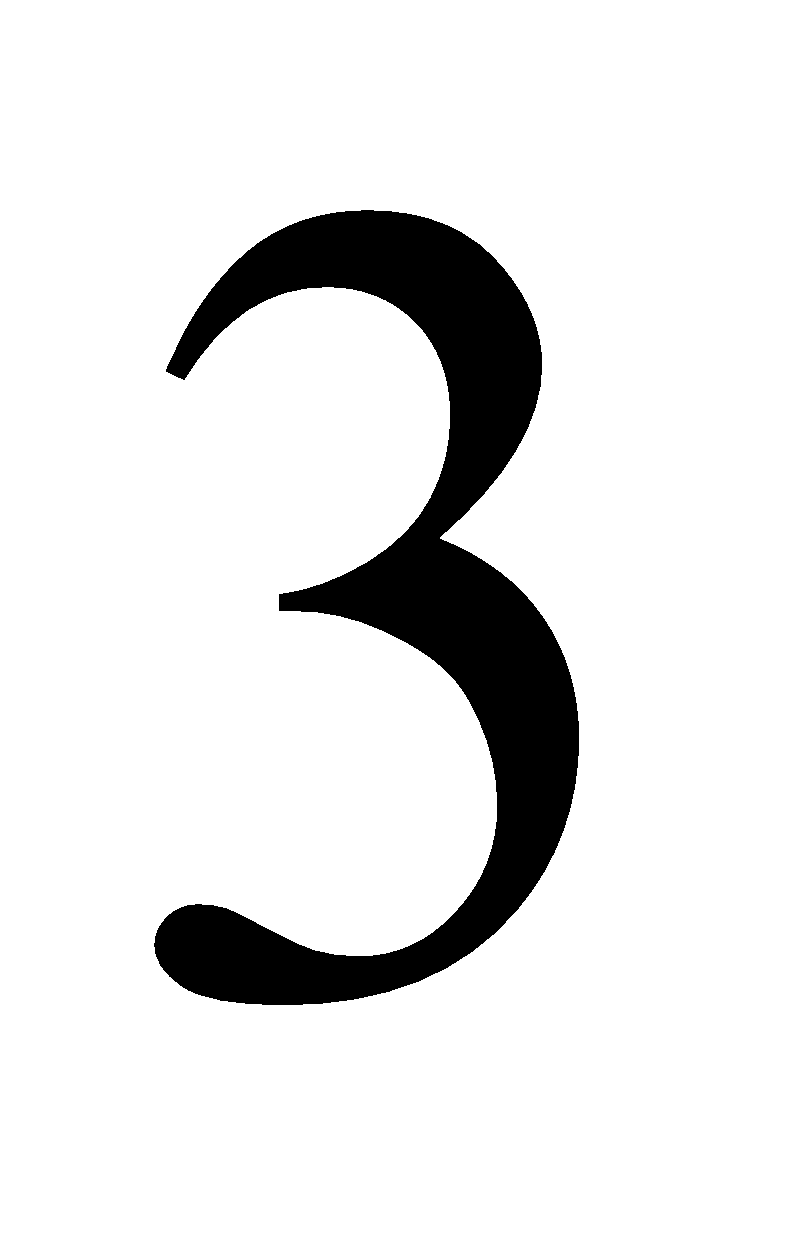
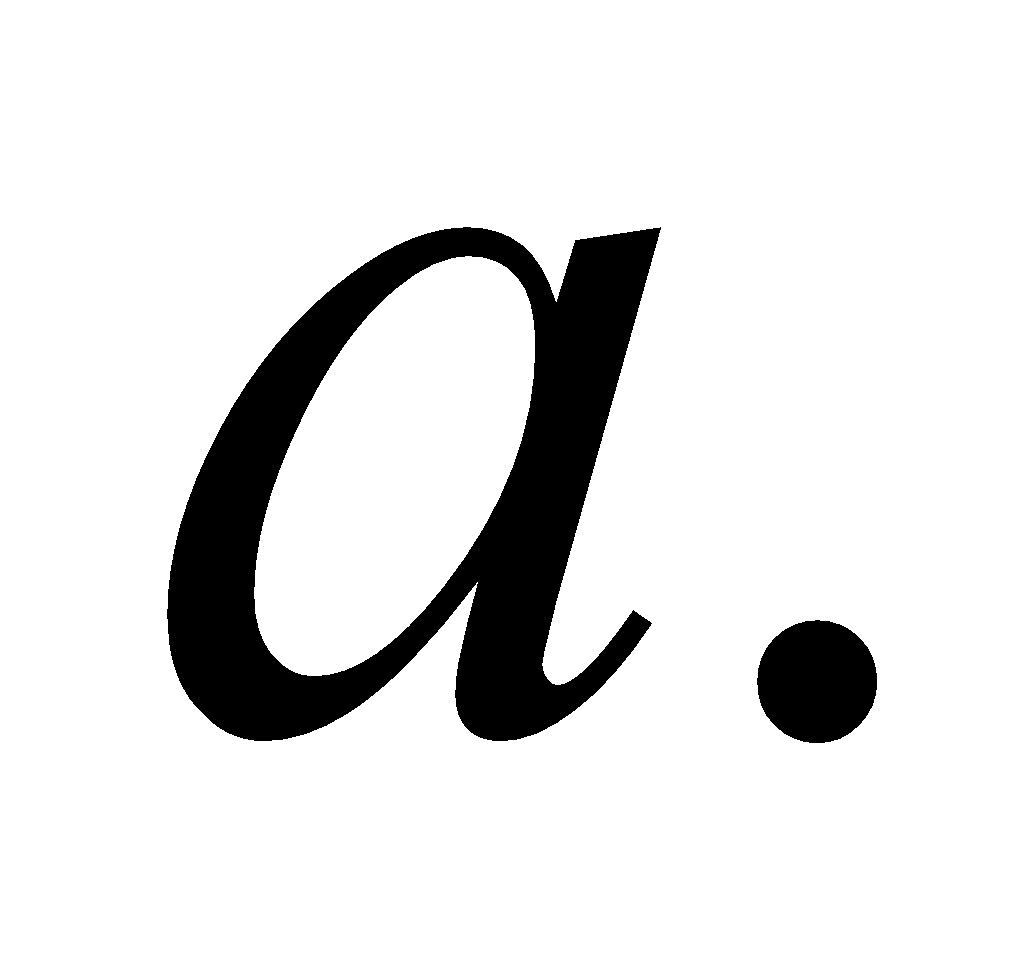
Алгоритм генерации множества всех подмножеств основывается на взаимно однозначном соответствии между элементами булеана  множества  и всеми целыми числами множества , записанными в двоичном виде. Следует сказать, что алгоритм имеет сложность  и поэтому реально может применяться для множеств с небольшой мощностью. Например, генерация булеана множества мощностью  на компьютере с тактовой частотой процессора 3 ГГц займет более 100 лет.

3:

Построение элементов булеана множества  сводится к следующему алгоритму:

1. Пронумеровать элементы заданного множества  начиная с нуля.
2. Сформировать битовую последовательность  состоящую из  двоичных нулей. Пронумеровать элементы этой последовательности справа налево, начиная с нуля.
3. Последовательно выполнить шаги 4 и 5 алгоритма  раз.
4. Выбрать из множества  элементы с номерами  для которых  Полученное подмножество будет являться элементом булеана  В первом случае не будет выбран ни один элемент (пустое подмножество) множества  так как исходная последовательность  состоит только из нулей.
5. Интерпретируя битовую последовательность как целое положительное число, увеличить это число на единицу.

4:

На рис. 1 представлен пример построения булеана для множества  Во втором сверху прямоугольнике изображены элементы     множества  Над ними указаны их номера:  для   для   для  и  для 

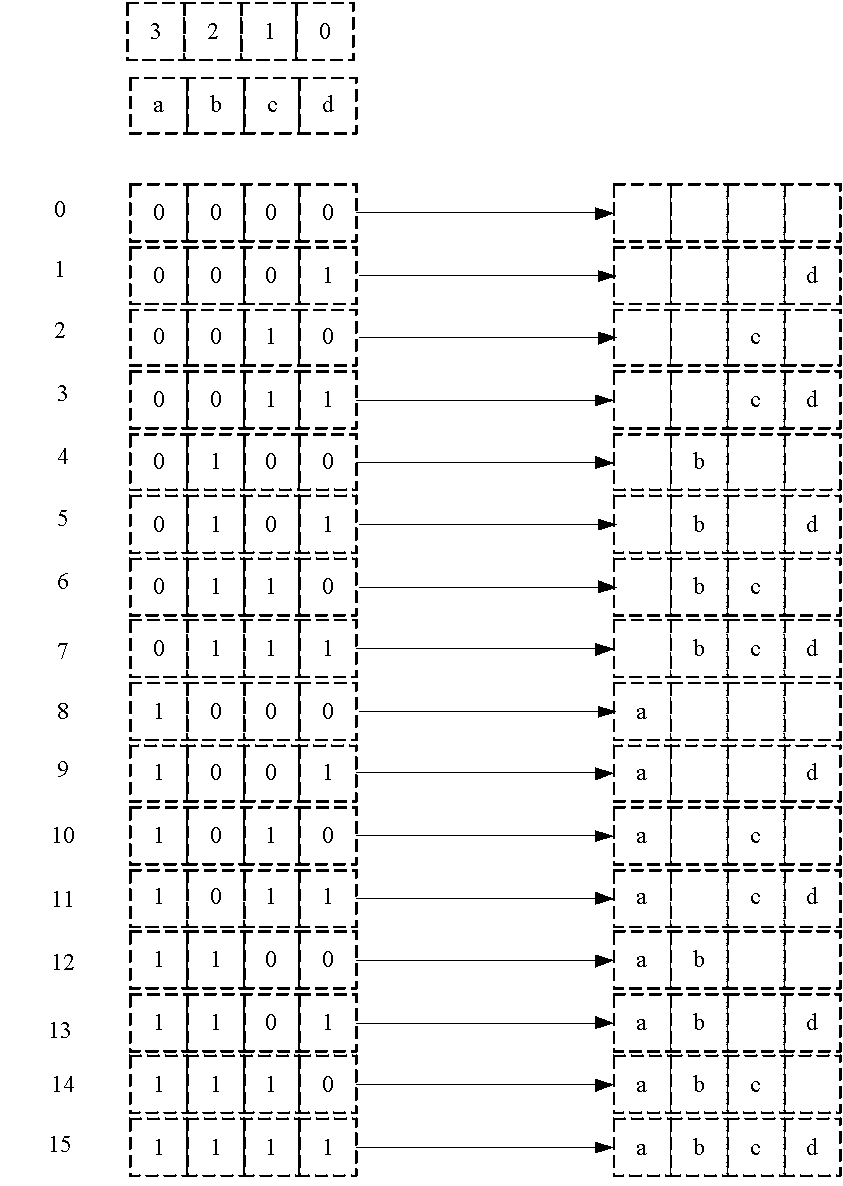
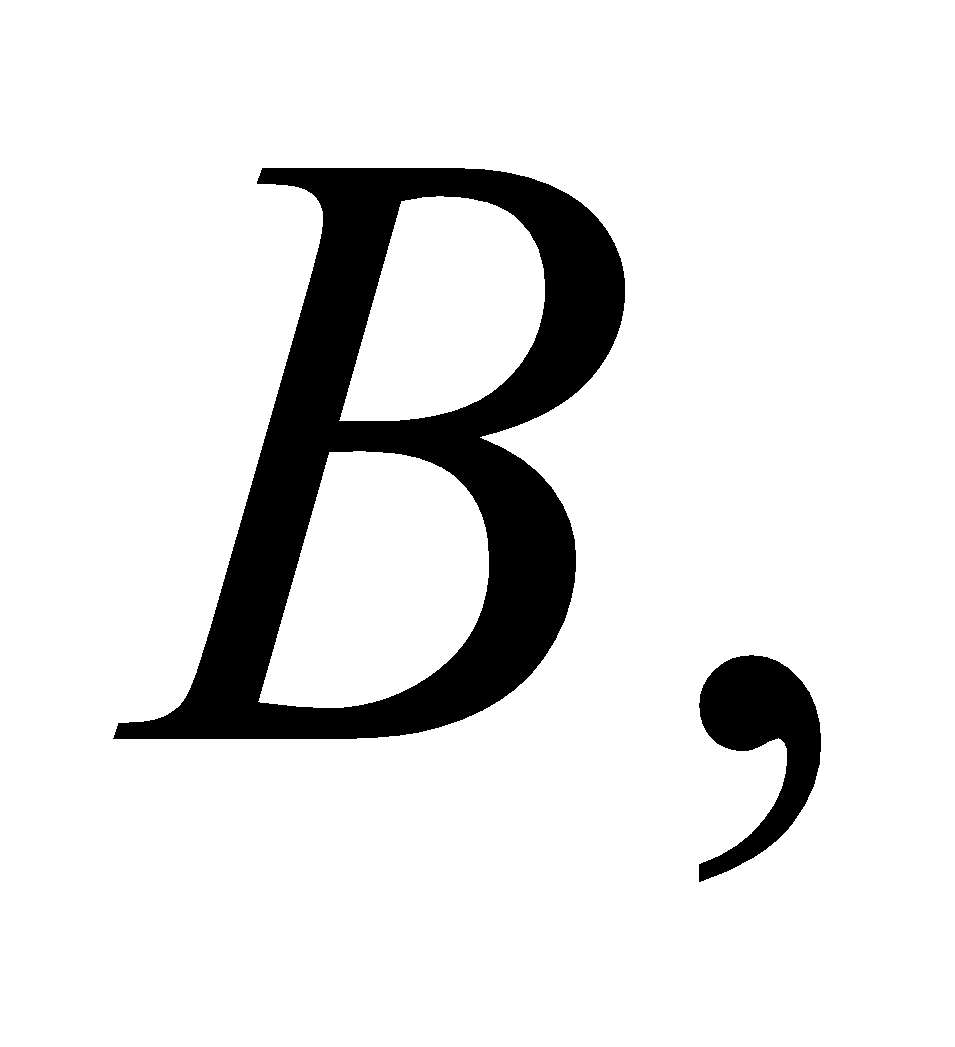
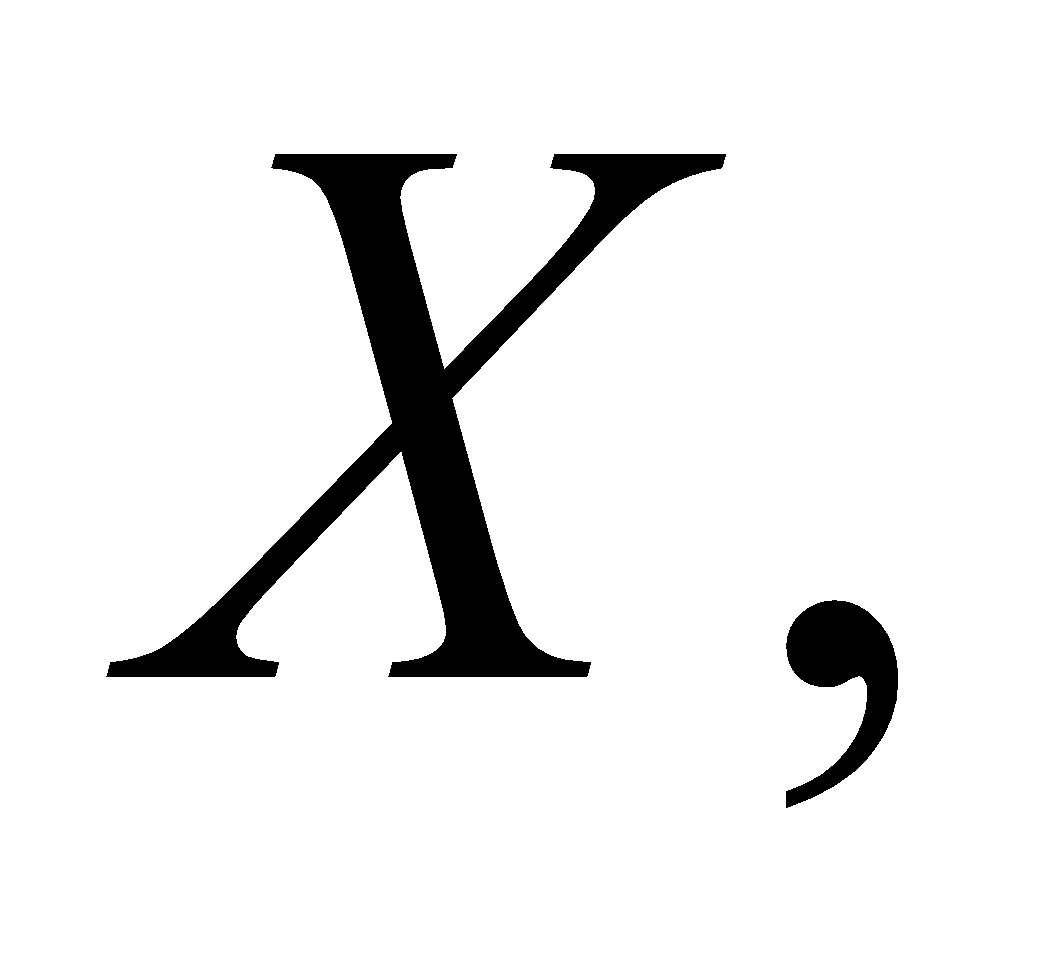
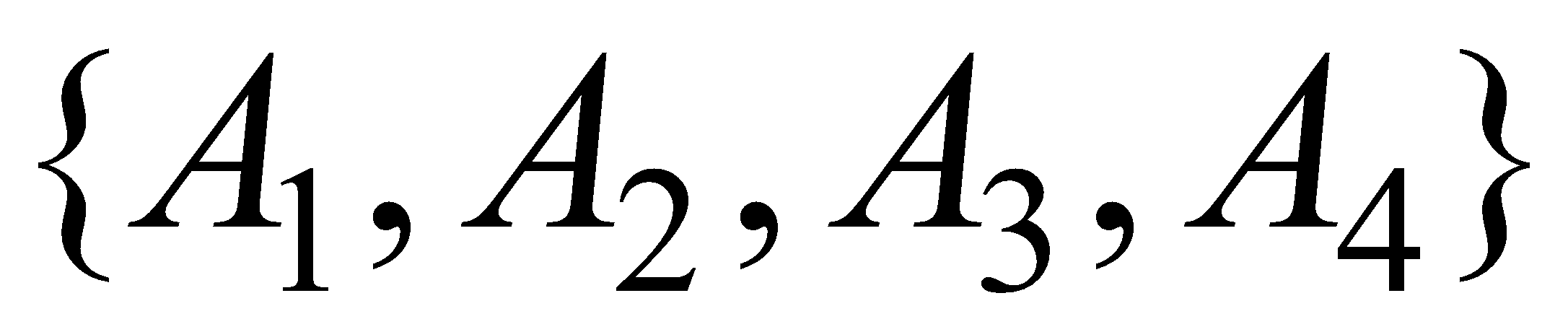


Рис. 1. Генерация множества всех подмножеств

Под элементами множества  изображен столбец битовых последовательностей. Каждая строка столбца соответствует состоянию последовательности  при очередном прохождении шага 4 алгоритма. Слева от столбца указаны целые числа, которые являются интерпретацией битовых последовательностей как целых чисел в десятичной системе счисления.

Стрелки на рис. 2.1. указывают на строки другого столбца, который содержит элементы множества  соответствующие битовым последовательностям. Каждая строка этого столбца содержит элементы одного из подмножеств множества 

Разберем еще один пример: Требуется получить множество всех подмножеств: . По формуле 1 узнаем количество всех подмножеств – 24 = 16.

5:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Порядковый номер | Запись в двоичном коде | | | | Запись в общем виде |
| 0 | 0 | 0 | 0 | 0 |  |
| 1 | 0 | 0 | 0 | 1 |  |
| 2 | 0 | 0 | 1 | 0 |  |
| 3 | 0 | 0 | 1 | 1 |  |
| 4 | 0 | 1 | 0 | 0 |  |
| 5 | 0 | 1 | 0 | 1 |  |
| 6 | 0 | 1 | 1 | 0 |  |
| 7 | 0 | 1 | 1 | 1 |  |
| 8 | 1 | 0 | 0 | 0 |  |
| 9 | 1 | 0 | 0 | 1 |  |
| 10 | 1 | 0 | 1 | 0 |  |
| 11 | 1 | 0 | 1 | 1 |  |
| 12 | 1 | 1 | 0 | 0 |  |
| 13 | 1 | 1 | 0 | 1 |  |
| 14 | 1 | 1 | 1 | 0 |  |
| 15 | 1 | 1 | 1 | 1 |  |

6:

**Принципы реализации генератора на С++**

1. Все генераторы должны иметь одинаковый интерфейс
2. Должна быть возможность применения нескольких генераторов одновременно.
3. Функции должны легко встраиваться в другие.

На рис. 2 и 3 представлена реализация генератора множества всех подмножеств на языке C++. Генератор реализован в виде структуры **subset**. Применение структуры позволяет создавать несколько экземпляров одного генератора и упростить его вызов в других программах.

7:

// Combi.h

**#pragma once**

**namespace combi**

**{**

**struct subset** // генератор множества всех подмножеств

**{**

**short n,** // количество элементов исходного множества < 64

**sn,** // количество элементов текущего подмножества

**\*sset;** // массив индексов текущего подмножества

**unsigned \_\_int64 mask;** // битовая маска

**subset(short n = 1);** // конструктор(количество элементов исходного множества)

**short getfirst();** // сформормировать массив индексов по битовой маске

**short getnext();** // ++маска и сформировать массив индексов

**short ntx(short i);** // получить i-й элемент массива индексов

**unsigned \_\_int64 count();** // вычислить общее количество подмножеств

**void reset();** // сбросить генератор, начать сначала

**};**

**};**

Рис. 2. Шаблон структуры генератора множества всех подмножеств

8-9:

// Combi.cpp

**#include "stdafx.h"**

**#include "Combi.h"**

**#include <algorithm>**

**namespace combi**

**{**

**subset::subset(short n)**

**{**

**this->n = n;**

**this->sset = new short[n];**

**this->reset();**

**};**

**void subset::reset()**

**{**

**this->sn = 0;**

**this->mask = 0;**

**};**

**short subset::getfirst()**

**{**

**\_\_int64 buf = this->mask;**

**this->sn = 0;**

**for (short i = 0; i < n; i++)**

**{**

**if (buf & 0x1) this->sset[this->sn++] = i;**

**buf >>= 1;**

**}**

**return this->sn;**

**};**

**short subset::getnext()**

**{**

**int rc = - 1;**

**this->sn = 0;**

**if (++this->mask < this->count()) rc = getfirst();**

**return rc;**

**};**

**short subset::ntx(short i)**

**{return this->sset[i];};**

**unsigned \_\_int64 subset::count()**

**{return (unsigned \_\_int64)(1<<this->n);};**

**};**

Рис. 3. Реализация методов структуры **subset**

Структура **subset**  имеет единственный конструктор, параметром которого является мощность исходного множества.

Для хранения текущего состояния генератора используются три переменные: **n** (мощность исходного множества, которая не должна превышать 63), **sn** (текущееколичество элементов массива индексов), **sset** (адрес нулевого элемента массива индексов) и **mask** (битовая последовательность). Первоначальное значение всех переменных инициализируются конструктором. Причем значения **n** и **sset** послеинициализацииостаются неизменными, а значения остальных переменных меняются в процессе работы генератора.

Помимо конструктора, структура содержит еще пять функций-членов.

Функция **getfirst** позволяет заполнить массив индексов в соответствии с текущим значением битовой последовательности **mask**. Функция устанавливает значение **sn**, равным количеству двоичных единиц в последовательности **mask**, а в массив **sset**,начиная с нулевого элемента, записывает номера единичных позиций (позиции нумеруются справа налево) в последовательности **mask**. Вызов функции сразу после создания структуры приводит к установке значения **sn** в нуль, что соответствует битовой последовательности **mask**, состоящей из одних нулей (конструктор инициализирует **mask** нулевым значением). Функция **getfirst** всегда возвращает значение **sn**. Обычно пользователь эту функцию вызывает один раз – для формирования первого (пустого) подмножества. Для получения остальных подмножеств применяется функция **getnext**.

Функция **getnext** увеличивает значение **mask** на единицу и вызывает функцию **getfirst**, которая формирует новый массив индексов и **sn** в соответствии с новым значением **mask**. Функция **getnext** возвращает значение **sn**, если массив индексов сформирован. Последний вызов функции возвращает значение – 1.

Функция **ntx** возвращает значение элемента массива индексов по индексу этого элемента и служит для сокращения записи при переборе элементов массива.

Функция **count** вычисляет и возвращает количество элементов булеана заданного множества.

Для перевода генератора в начальное состояние служит функция **reset**. После вызова **reset** снова могут вызываться **getfirst** и **getnext**.

На рис. 4 и 5 приведен пример применения генератора множества всех подмножеств и результат выполнения программы.

9-10:

// Main

**#include "stdafx.h"**

**#include <iostream>**

**#include "Combi.h"**

**int \_tmain(int argc, \_TCHAR\* argv[])**

**{**

**setlocale(LC\_ALL, "rus");**

**char AA[][2]= {"A", "B", "C", "D"};**

**std::cout<<std::endl<<" - Генератор множества всех подмножеств -";**

**std::cout<<std::endl<<"Исходное множество: ";**

**std::cout<<"{ ";**

**for (int i = 0; i < sizeof(AA)/2; i++)**

**std::cout<<AA[i]<<((i< sizeof(AA)/2-1)?", ":" ");**

**std::cout<<"}";**

**std::cout<<std::endl<<"Генерация всех подмножеств ";**

**combi::subset s1(sizeof(AA)/2);** // создание генератора

**int n = s1.getfirst();** // первое (пустое) подмножество

**while (n >= 0)** // пока есть подмножества

**{**

**std::cout<<std::endl<<"{ ";**

**for (int i = 0; i < n; i++)**

**std::cout<<AA[s1.ntx(i)]<<((i< n-1)?", ":" ");**

**std::cout<<"}";**

**n = s1.getnext();** // cледующее подмножество

**};**

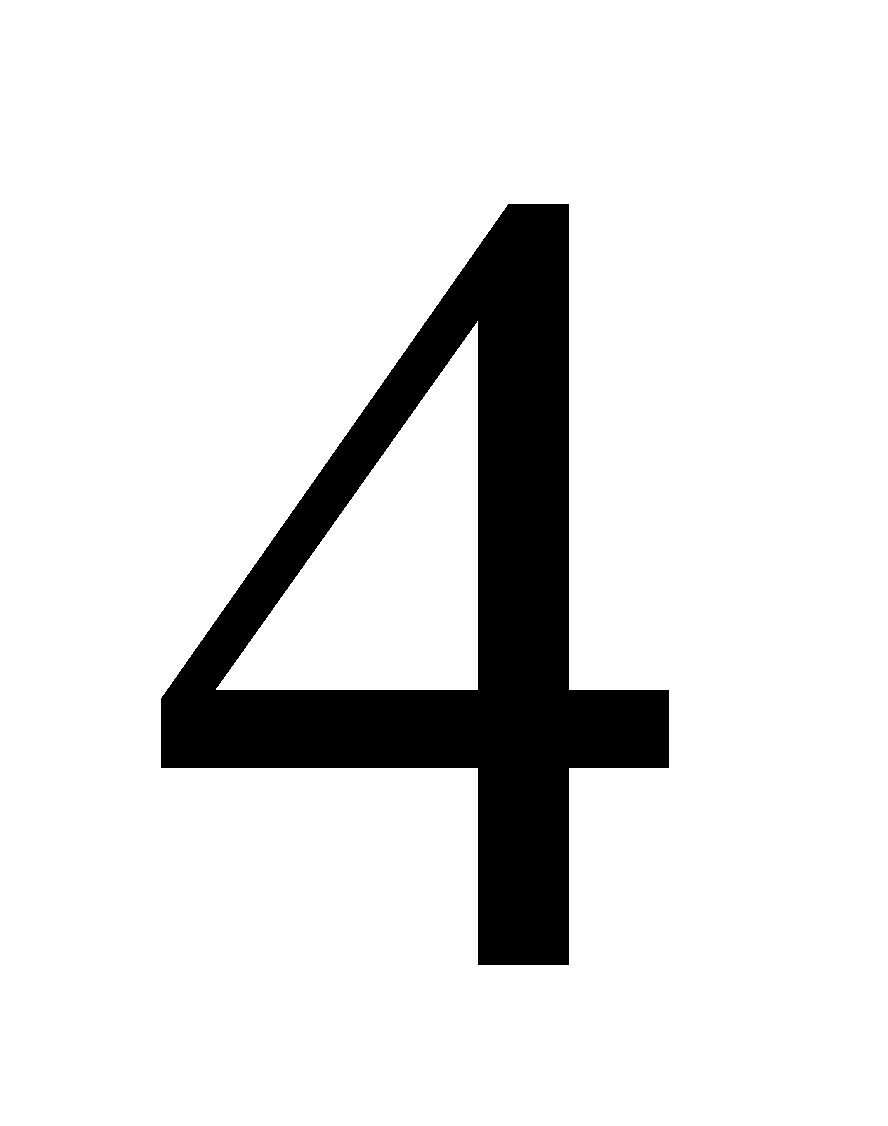
**std::cout<<std::endl<<"всего: " << s1.count()<<std::endl;**

**system("pause");**

**return 0;**

**}**

Рис. 4. Пример применения генератора множества всех подмножеств

В качестве исходного множества в примере используется строковый массив, состоящий из четырех строк. Вначале этот массив распечатывается. Затем объявляется структура **subset** и ей в качестве параметра передается количество элементов исходного множества (значение ).

Цикл формирования подмножеств начинается функцией **getfirst**,которая формирует первый (пустой) массив **sset**, соответствующий пустому подмножеству. Все остальные массивы индексов (для непустых подмножества) формируются функцией **getnext**. Выход из цикла происходит после того, как **getnext** возвращает отрицательное значение (получены все подмножества). Выбор элементов исходного массива осуществляется с помощью функции **ntx**.

11:

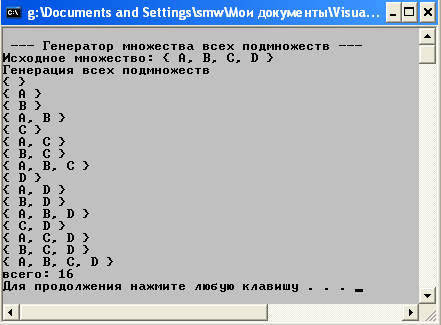
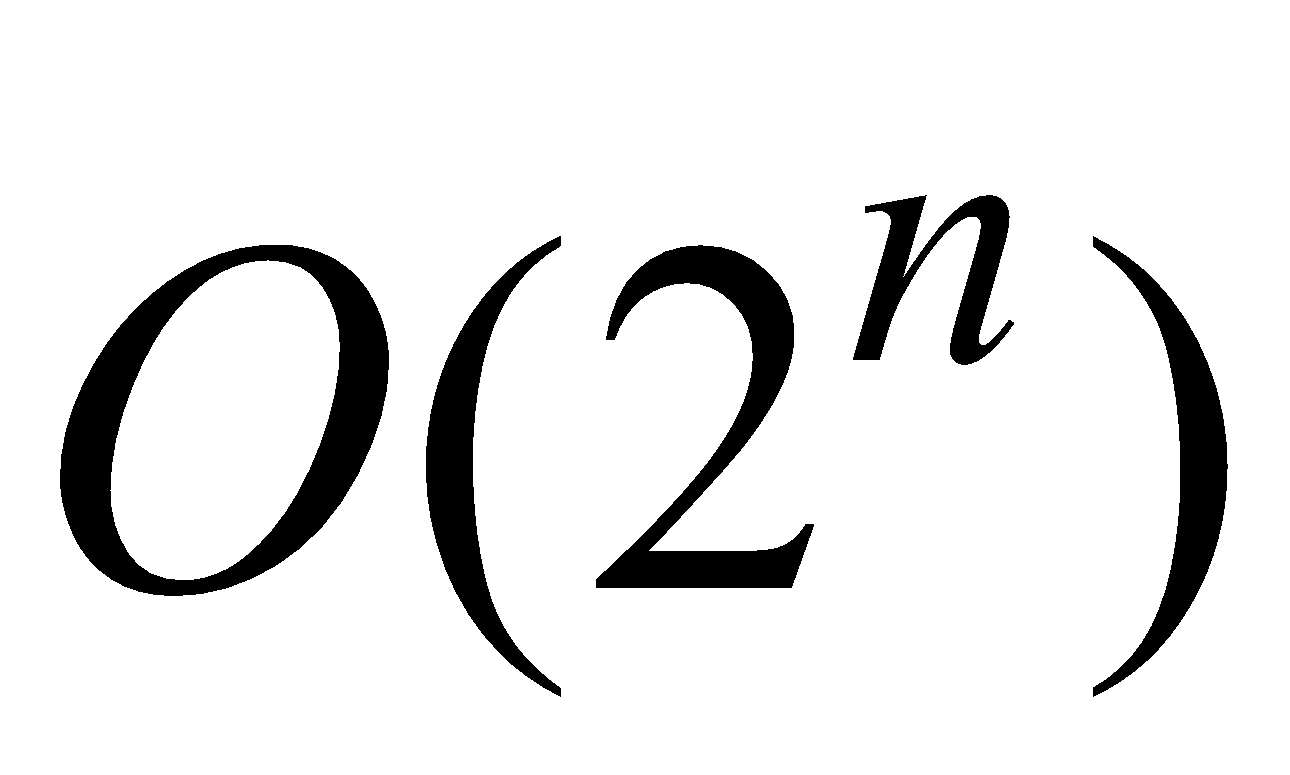


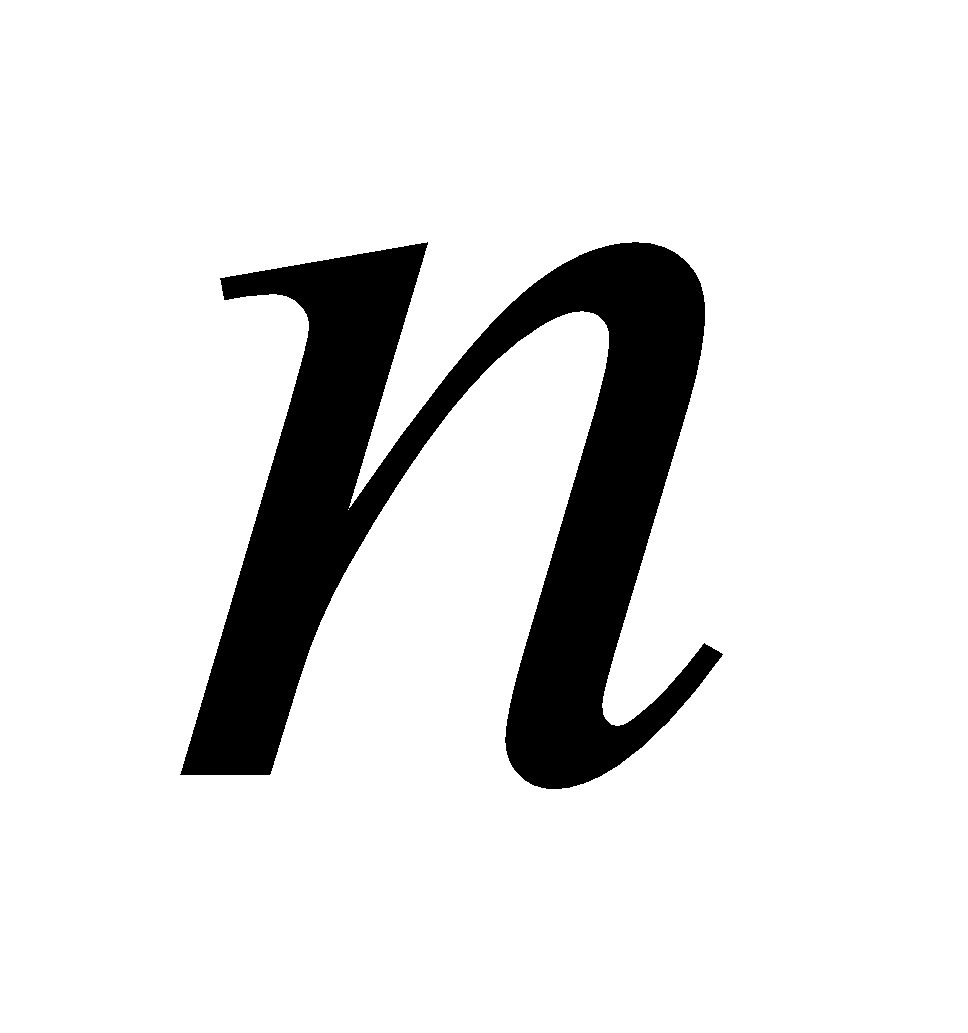
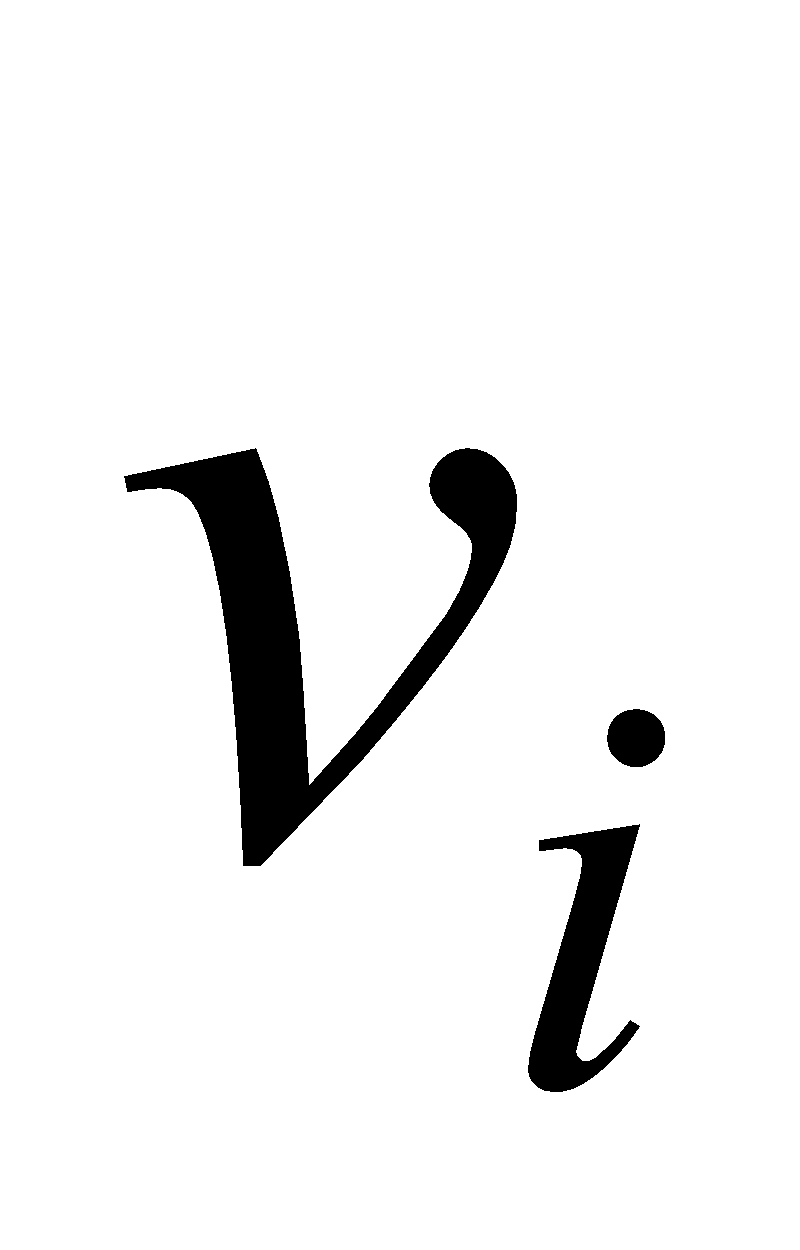
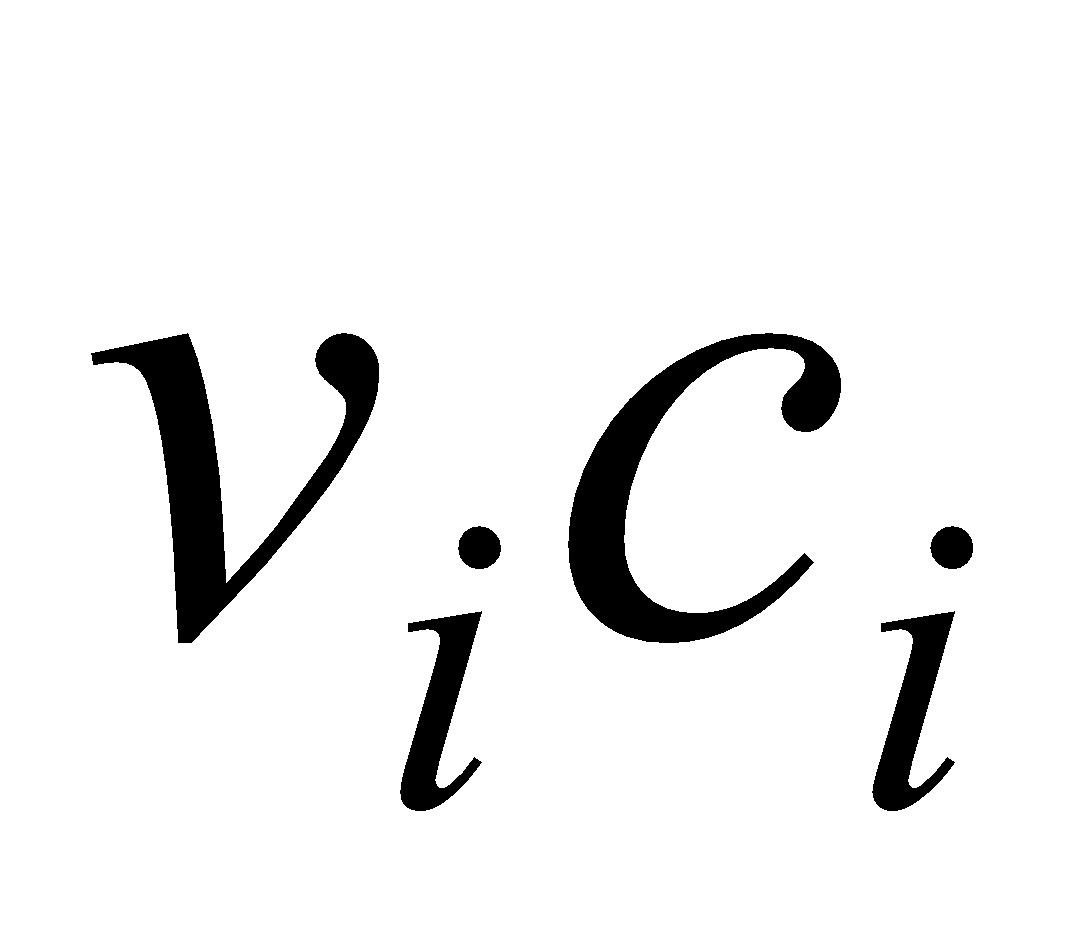
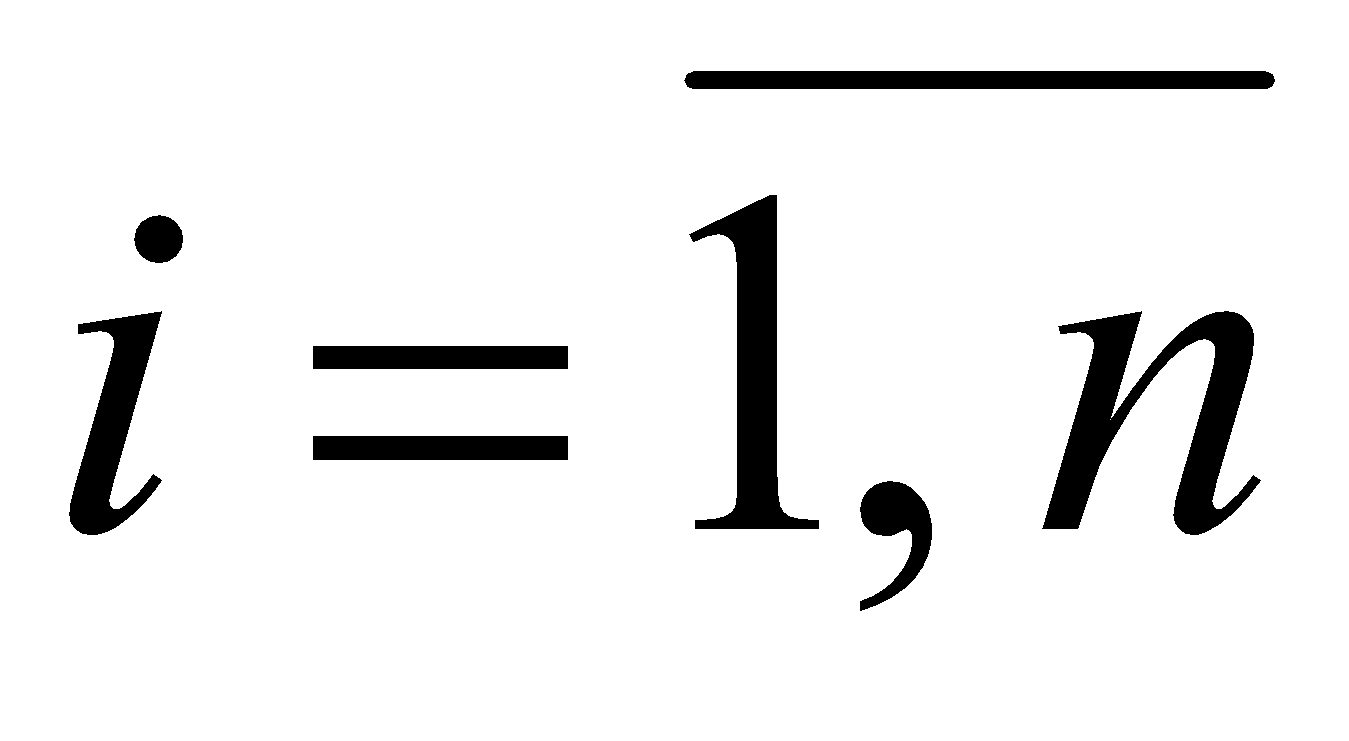
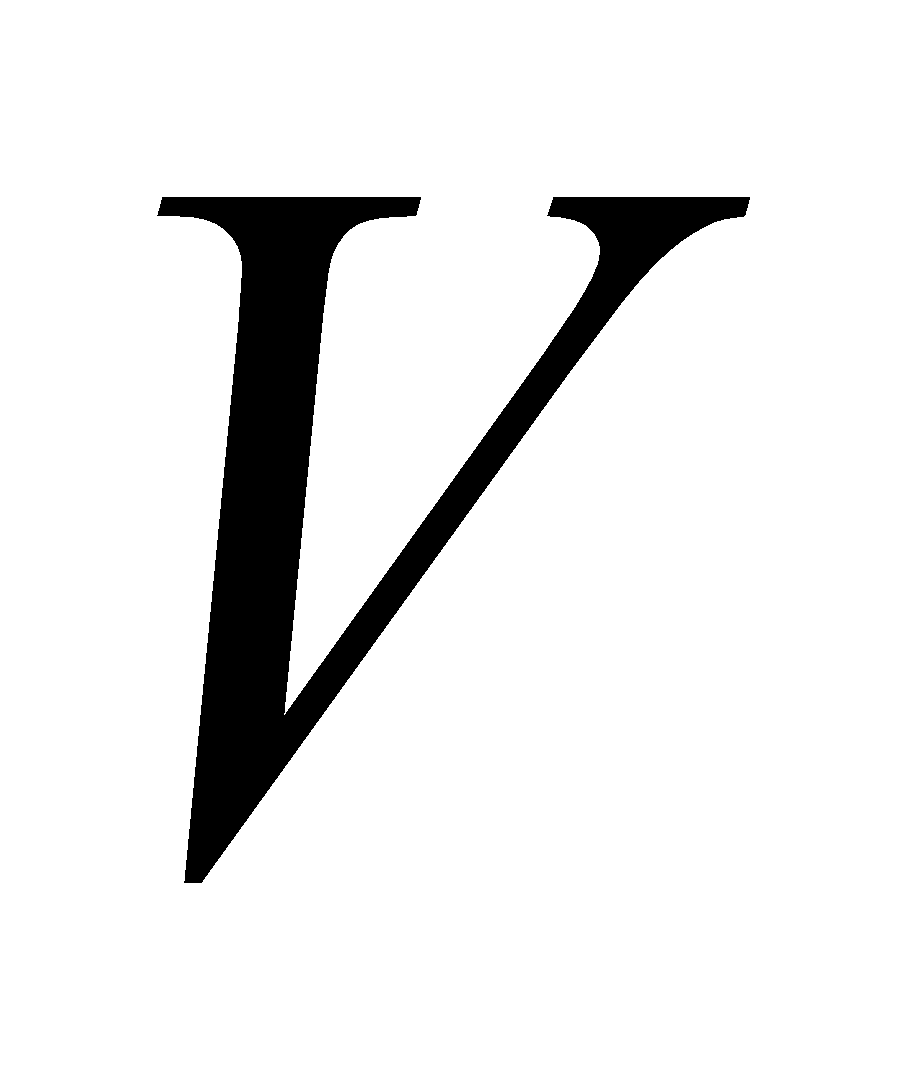
Рис. 5. Результат выполнения программы, представленной на рис. 4

**Сложность: **

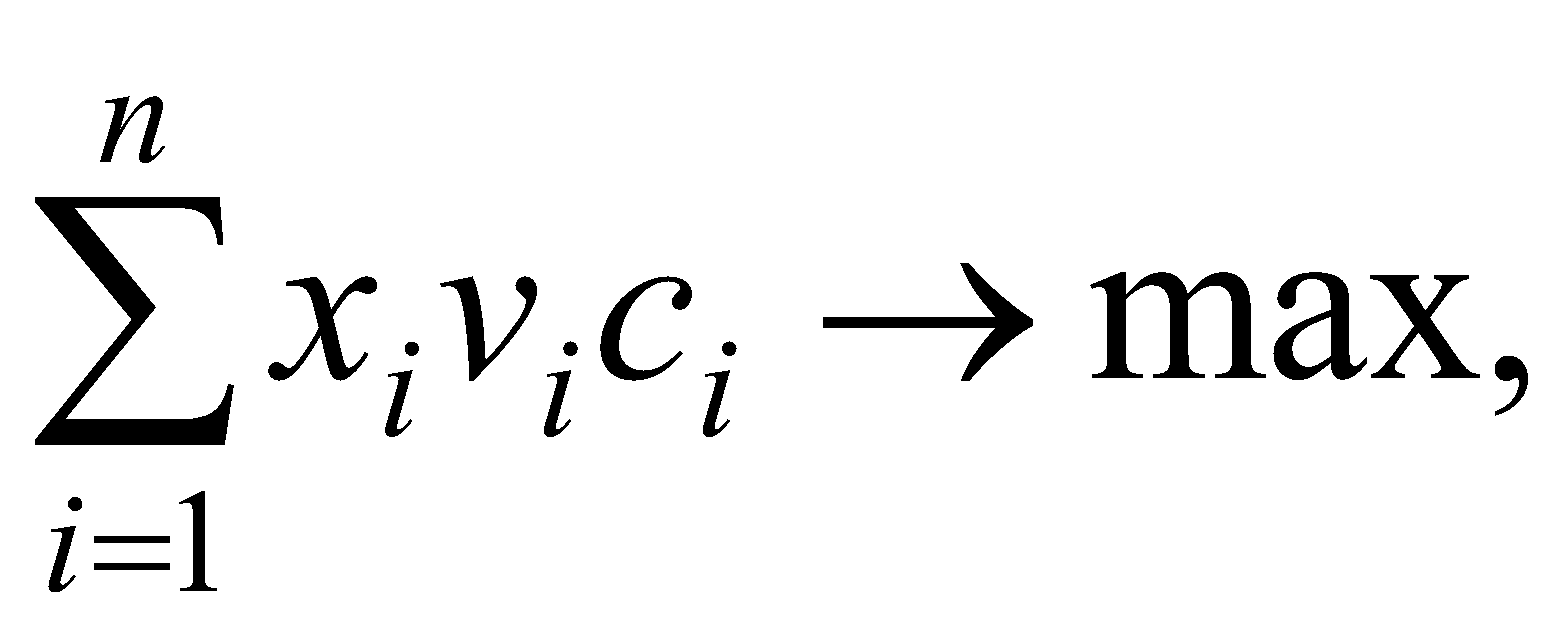
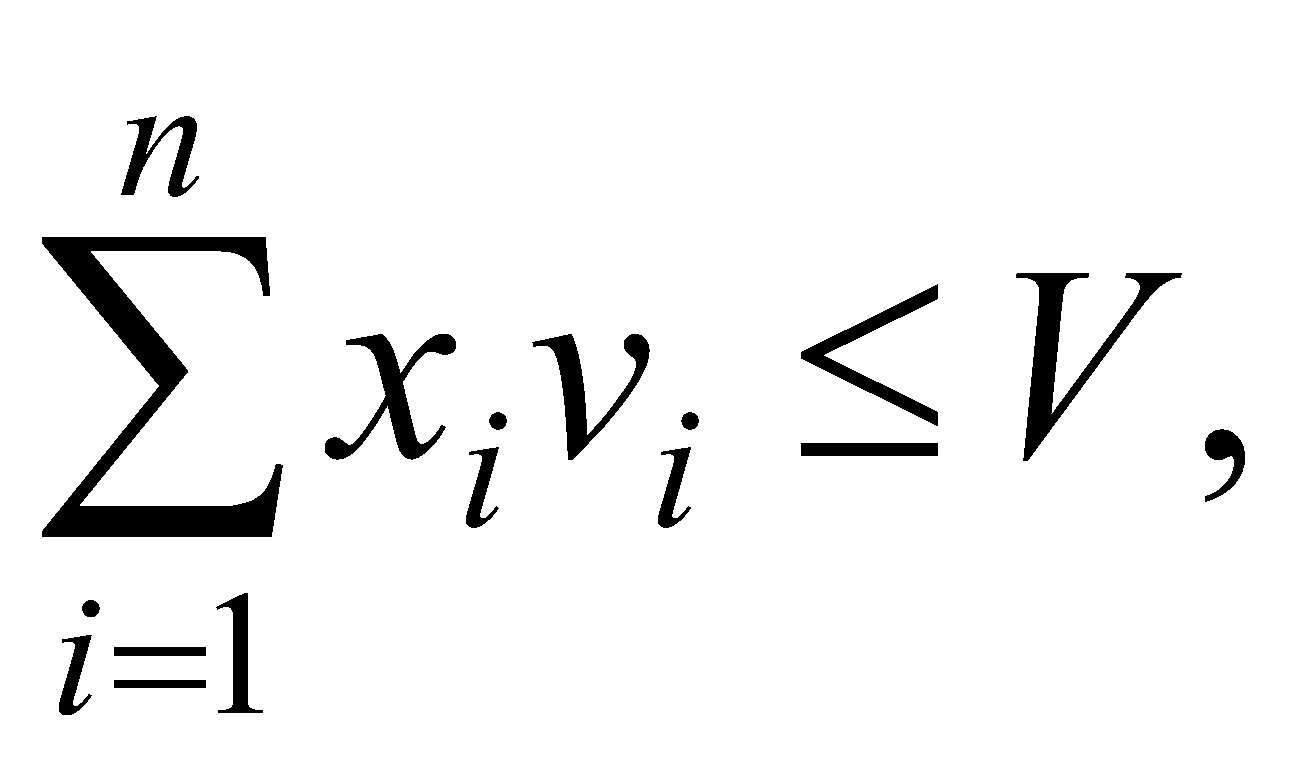
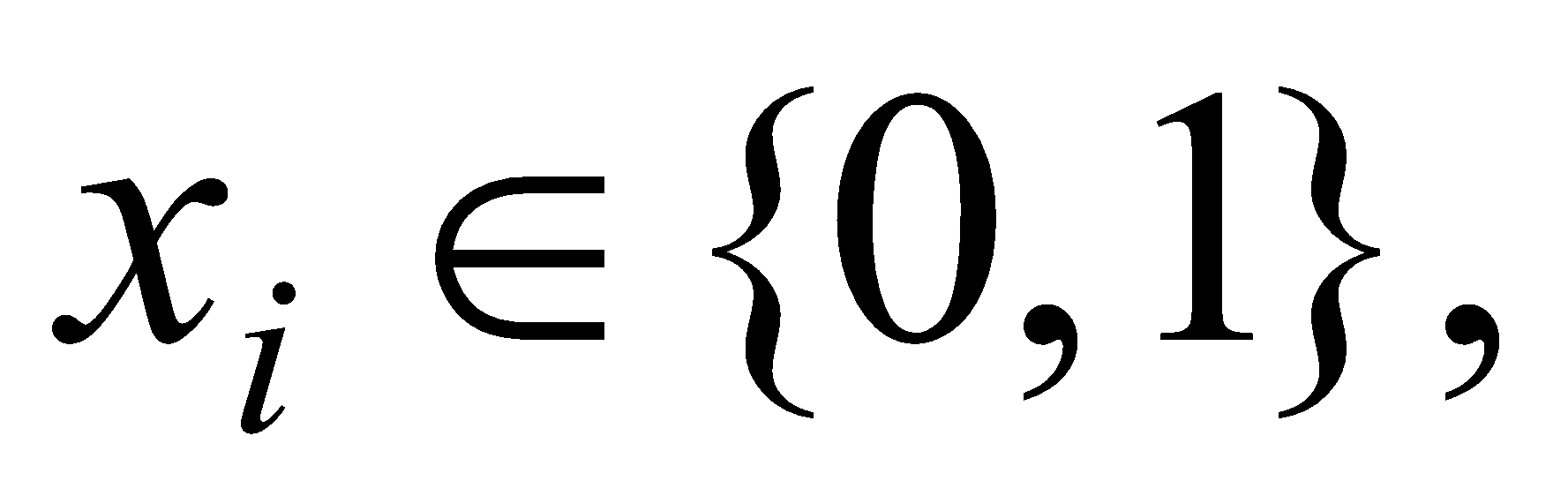
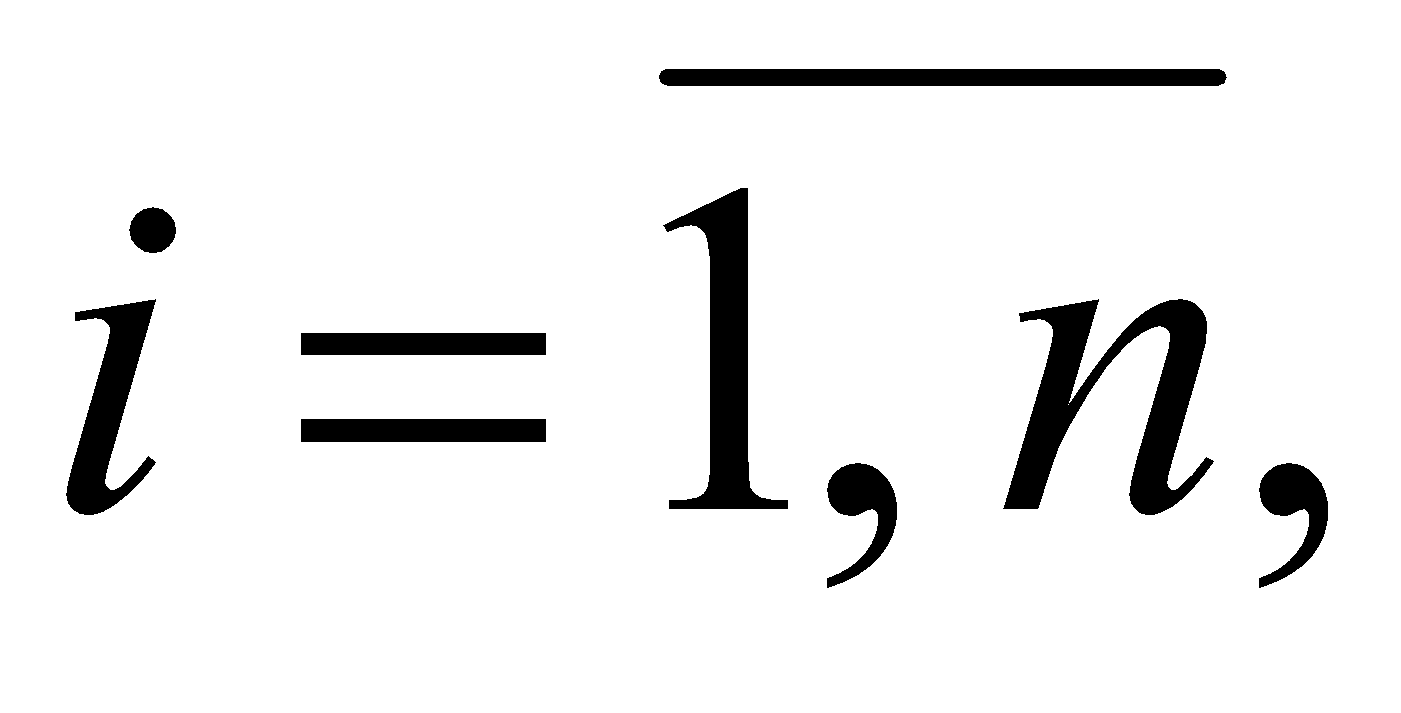
12:

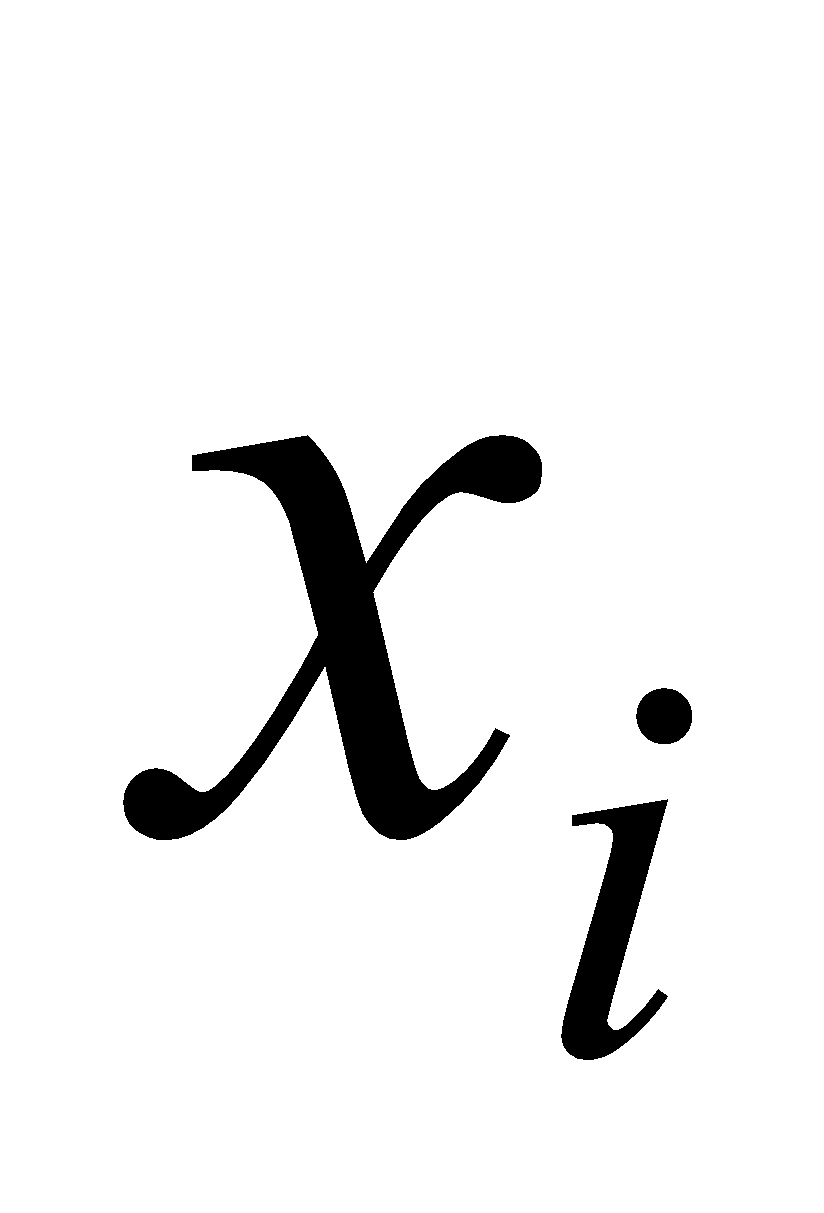
**Решение упрощенной задачи о рюкзаке с помощью генератора множества всех подмножеств**

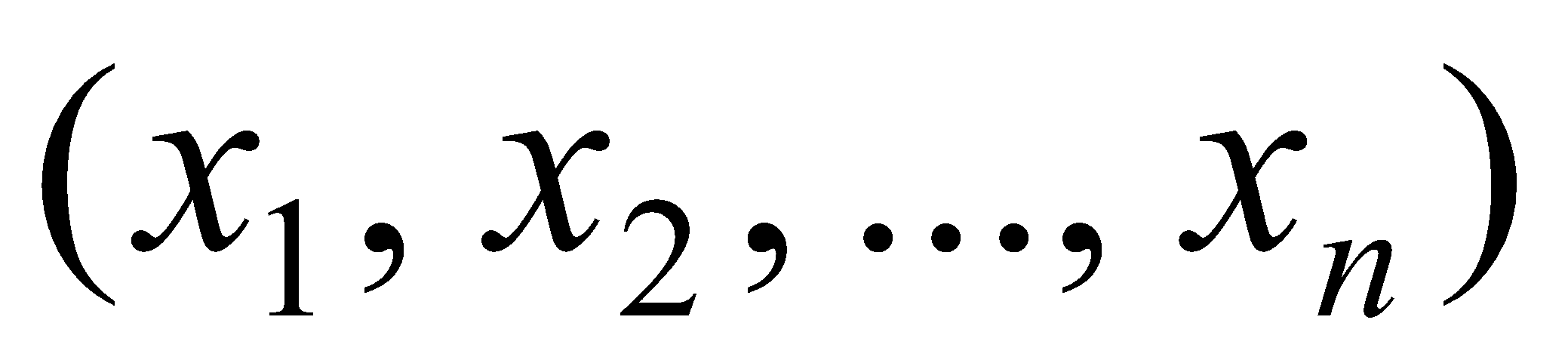
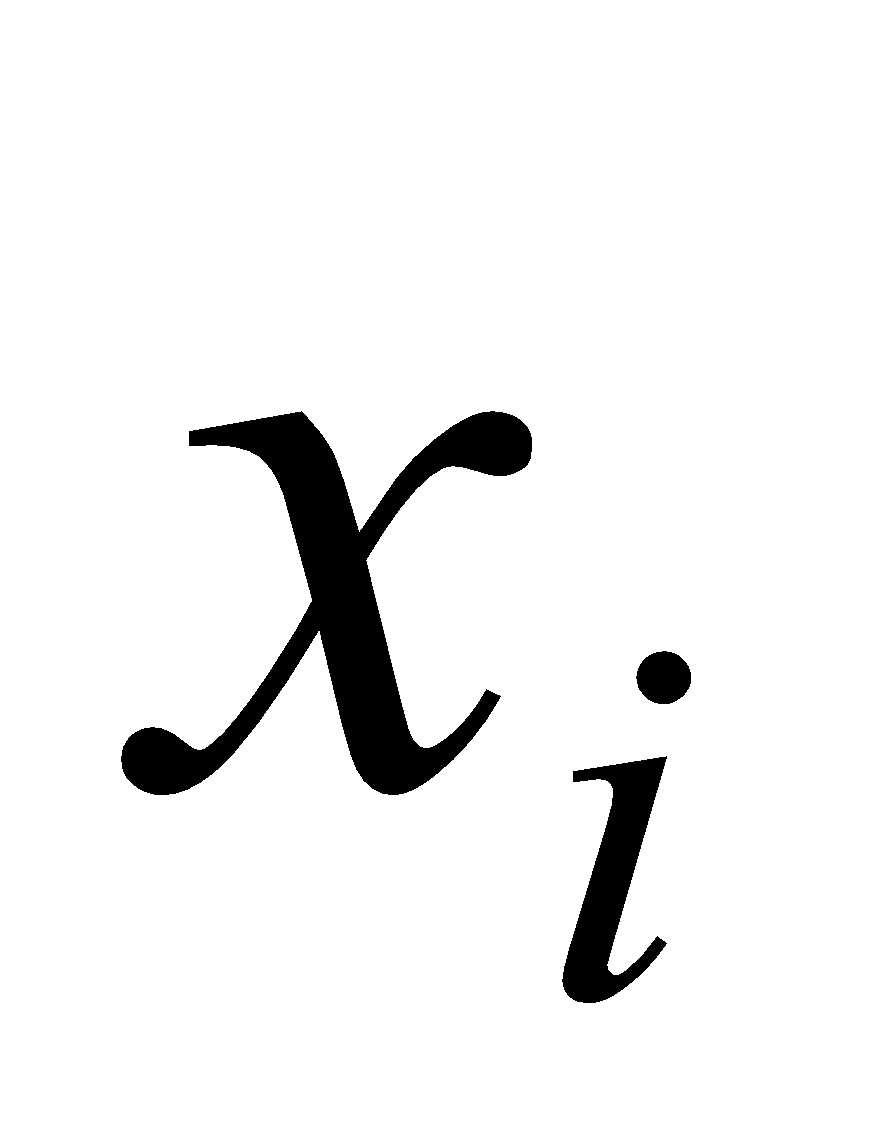
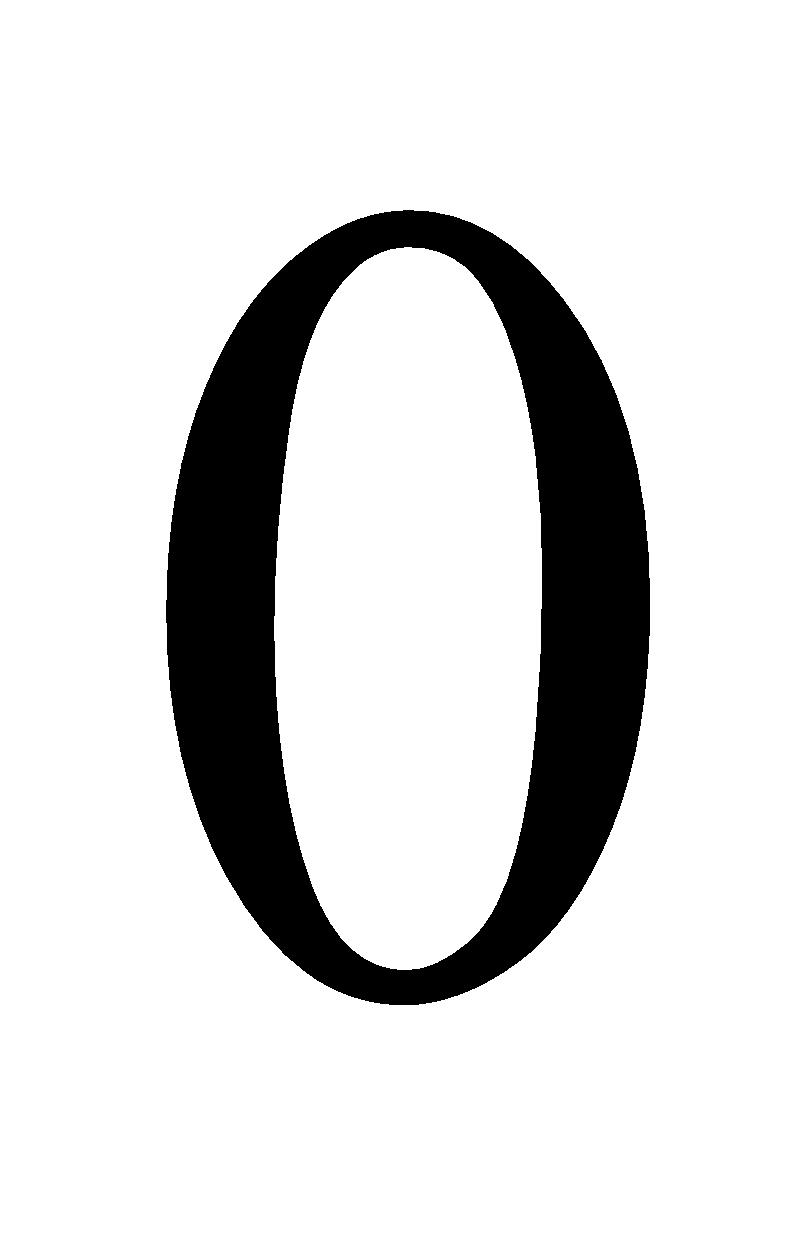
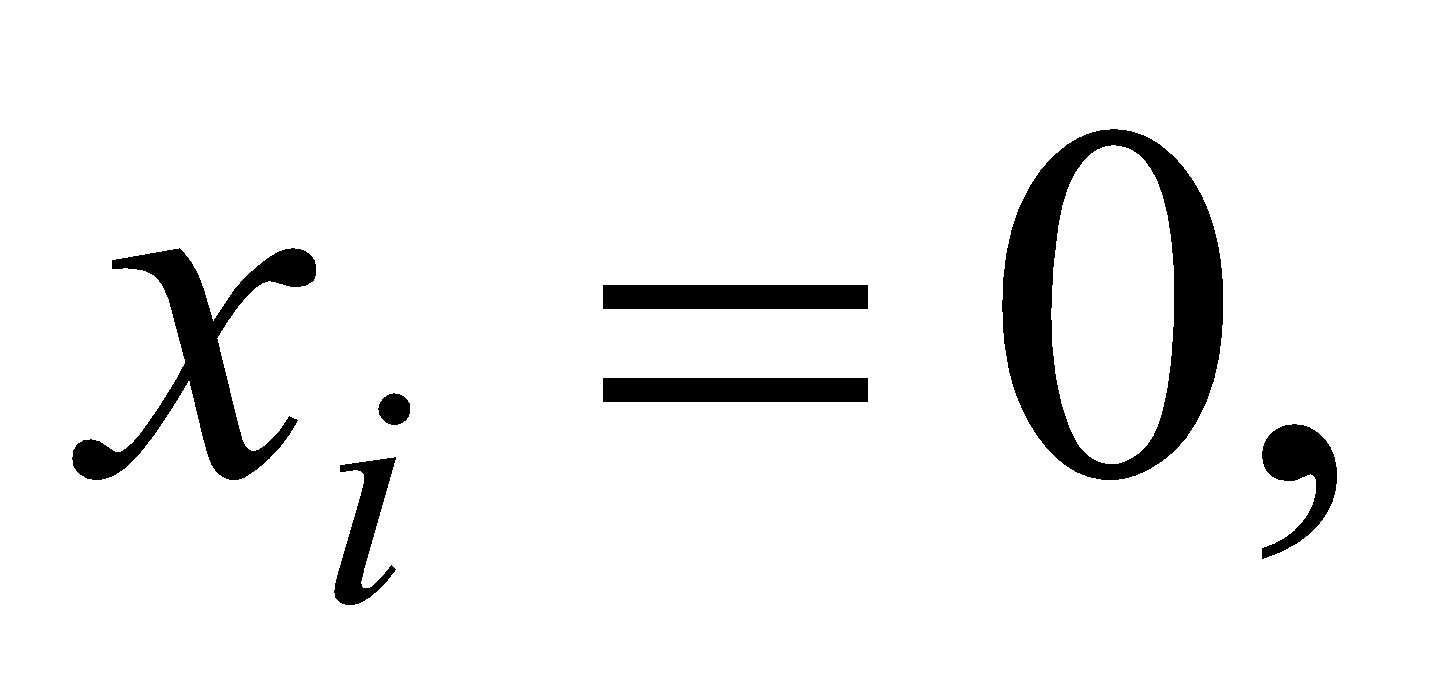
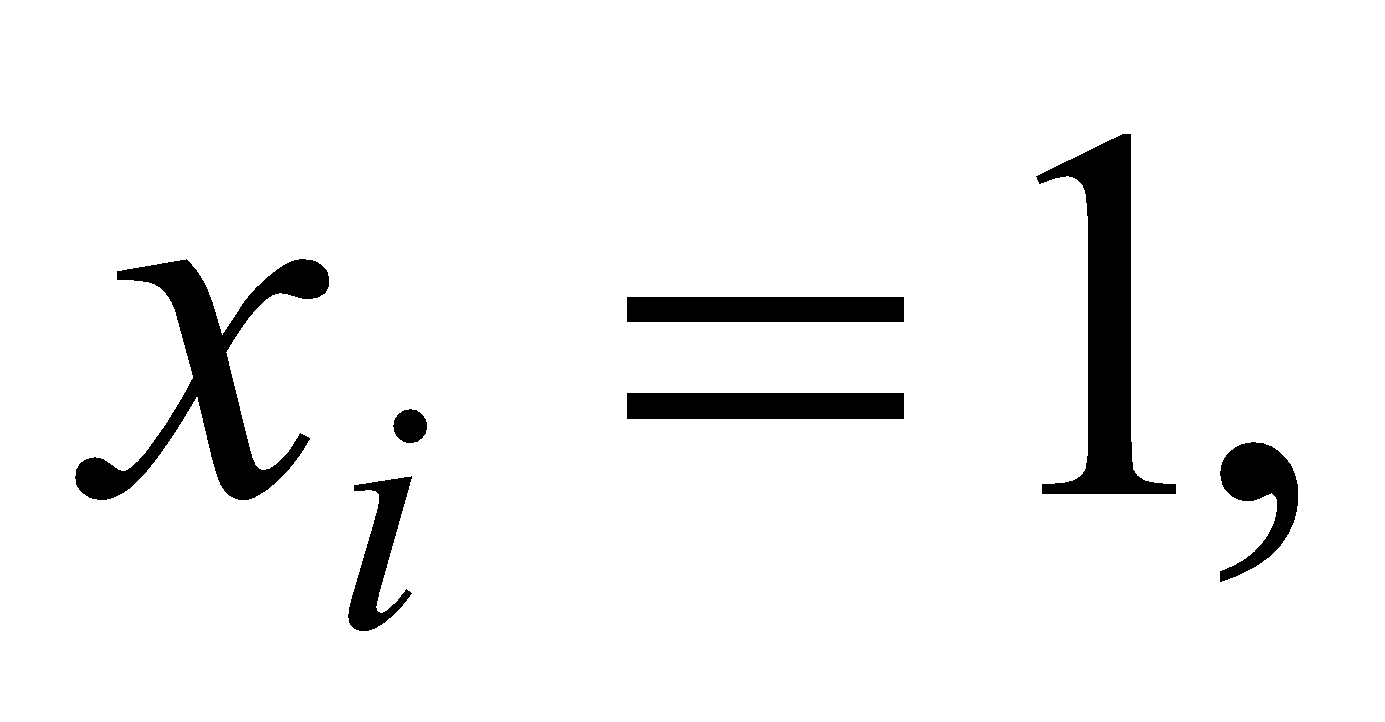
Задача о рюкзаке является одной из известных задач, решаемых методом перебора. Сформулируем упрощенную задачу о рюкзаке.

Существует  различных предметов, характеризующихся объемом  и стоимостью , . Необходимо выбрать несколько разных предметов таким способом, чтобы они поместились в рюкзаке объемом  и при этом их суммарная стоимость была максимальной.

Математическая модель задачи может быть записана следующим образом:

где  – неизвестные, которые требуется найти.

Решением задачи при такой постановке будет вектор . Каждый элемент  вектора может принимать значение  или 1. При этом если  то -ый предмет не выбран, и если  то -й предмет выбран для размещения в рюкзаке.

13:

Задача имеет следующие исходные данные:

 – вместимость (объем) рюкзака;

 – количество предметов;

 – вектор объемов предметов;

 – вектор стоимостей предметов.

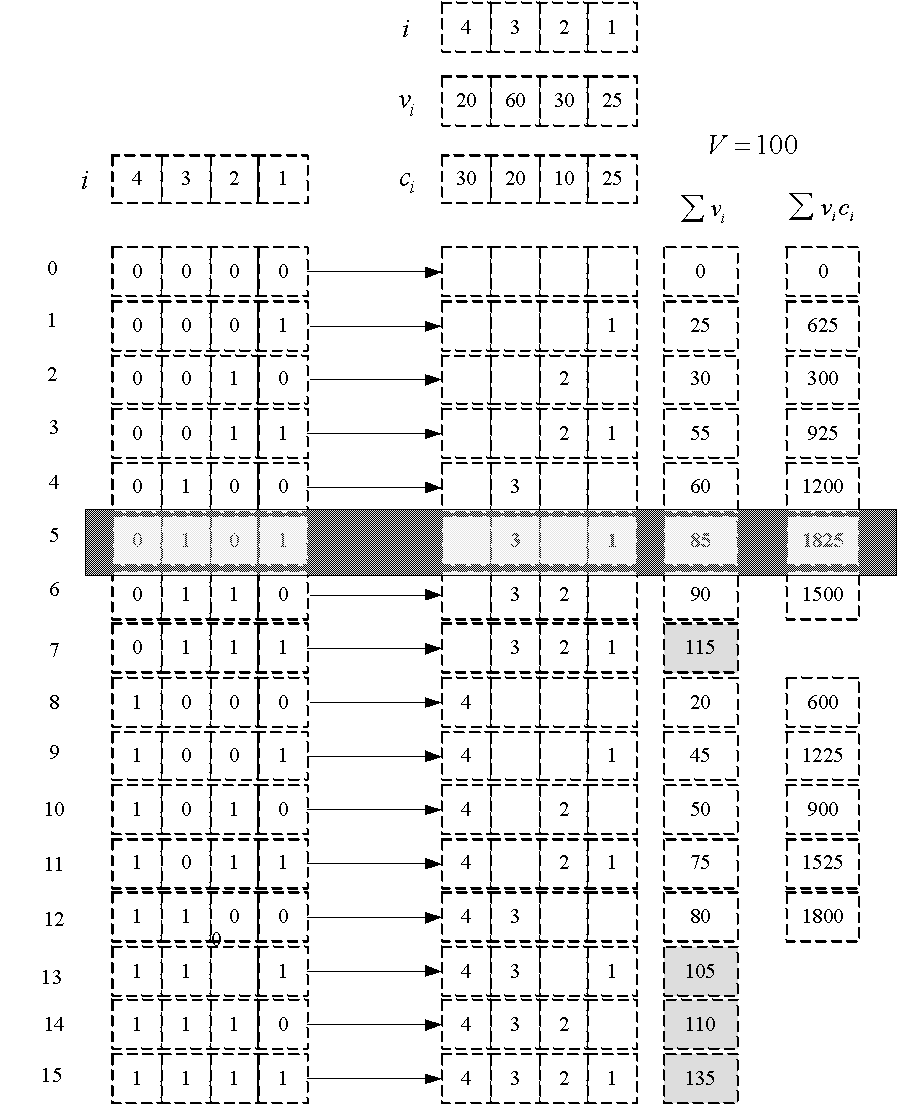
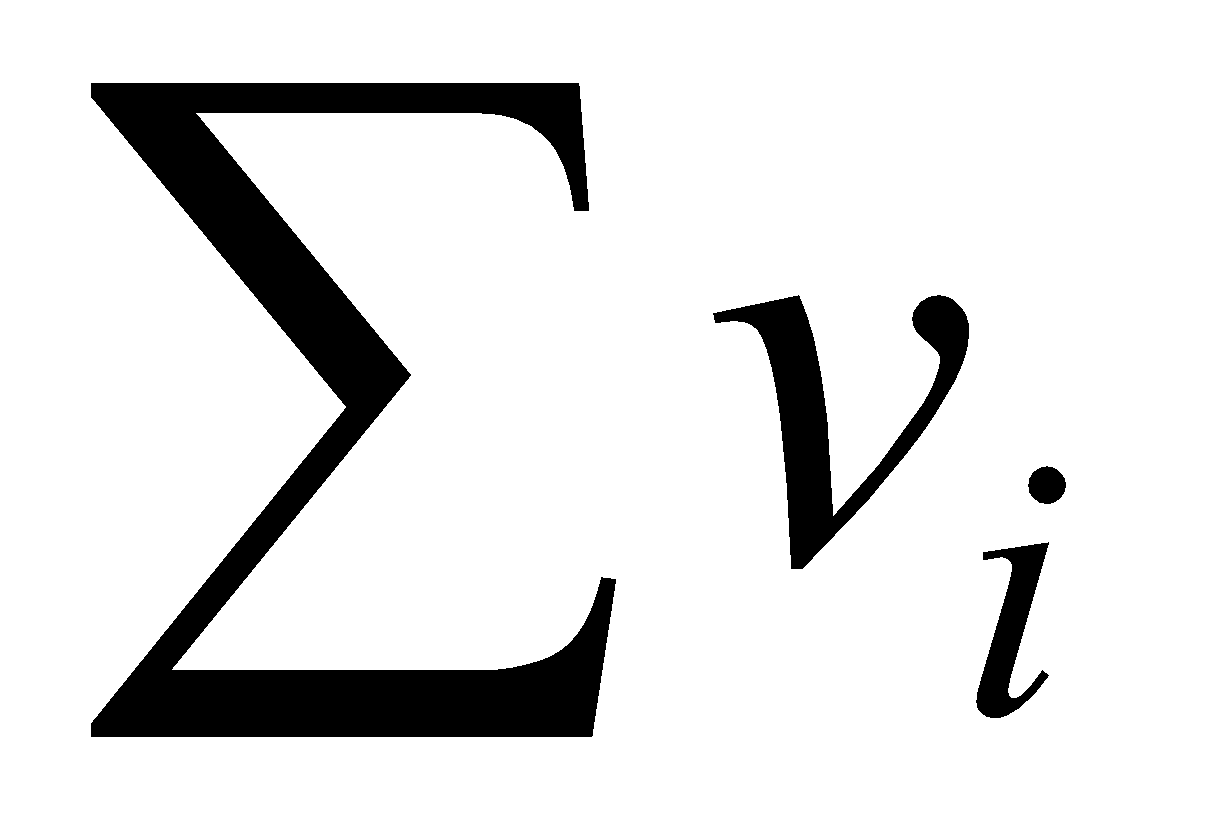
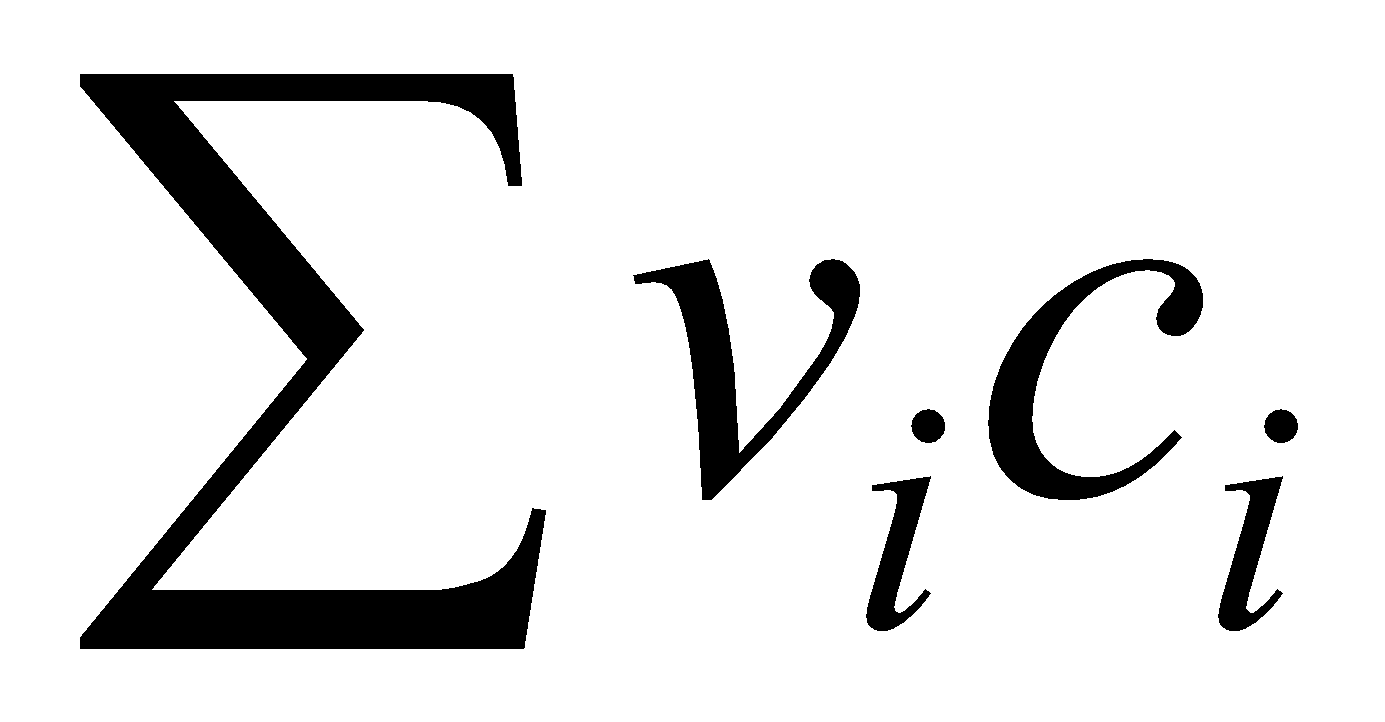


Рис. 6. Схема решения задачи о рюкзаке с применением генератора множества всех подмножеств

С помощью генератора всех подмножеств последовательно генерируются все массивы индексов (на рис. 6 второй слева столбец), соответствующие битовой последовательности (первый столбец слева). На основе массивов индексов формируются все возможные подмножества предметов. Для каждого подмножества вычисляется суммарный объем (столбец с заголовком) и сверяется с вместимостью рюкзака. Если выбранное подмножество предметов помещается в рюкзак (суммарный объем предметов не превышает вместимости рюкзака ), то для выбранных предметов рассчитывается суммарная стоимость (столбец ). Окончательным решением задачи будет подмножество предметов, имеющее максимальную суммарную стоимость при допустимом суммарном объеме. На рис. 6 строка, соответствующая решению, отмечена рамкой.

14:

Пример реализации функции **knapsack\_s** на языке C++, которая решает задачу о рюкзаке.

// Knapsack.h

**#pragma once**

**#include "Combi.h"**

**int knapsack\_s(**

**int V,** // [in] вместимость рюкзака

**short n,** // [in] количество типов предметов

**const int v[],** // [in] размер предмета каждого типа

**const int c[],** // [in] стоимость предмета каждого типа

**short m[]** // [out] количество предметов каждого типа

**);**

Рис.7. Прототип функции **knapsack\_s**

15-16:

// Knapsack.cpp

**#include "stdafx.h"**

**#include "Knapsack.h"**

**#define NINF 0x80000000** // самое малое int-число

**int calcv(combi::subset s, const int v[])** // объем в рюкзаке

**{**

**int rc = 0;**

**for (int i = 0; i < s.sn; i++) rc += v[s.ntx(i)];**

**return rc;**

**};**

**int calcc(combi::subset s, const int v[], const int c[])** //стоимость в рюкзаке

**{**

**int rc = 0;**

**for (int i = 0; i < s.sn; i++) rc += (v[s.ntx(i)]\*c[s.ntx(i)]);**

**return rc;**

**};**

**void setm(combi::subset s, short m[])** //отметить выбранные предметы

**{**

**for (int i = 0; i < s.n; i++) m[i] = 0;**

**for (int i = 0; i < s.sn; i++) m[s.ntx(i)] = 1;**

**};**

**int knapsack\_s(**

**int V,** // [in] вместимость рюкзака

**short n,** // [in] количество типов предметов

**const int v[],** // [in] размер предмета каждого типа

**const int c[],** // [in] стоимость предмета каждого типа

**short m[]** // [out] количество предметов каждого типа {0,1}

**)**

**{**

**combi::subset s(n);**

**int maxc = NINF, cc = 0;**

**short ns = s.getfirst();**

**while (ns >= 0)**

**{**

**if (calcv(s, v) <= V)**

**if ((cc = calcc(s,v,c)) > maxc)**

**{**

**maxc = cc;**

**setm(s,m);**

**}**

**ns = s.getnext();**

**};**

**return maxc;**

**};**

Рис.8. Реализация функции **knapsack\_s**

Функция **knapsack\_s** имеет четыре входных параметра, задающих условие задачи: **V** (объем рюкзака), **n** (количество предметов), **v** (массив размерностью **n**, содержащий объемы всех предметов), **c** (массив размерностью **n**, содержащий стоимости всех предметов), а также один выходной параметр **m** (массив размерностью **n**). Каждый элемент массива **m** может быть только единицей или нулем. Единица указывает, что соответствующий предмет включен, а ноль – не включен в оптимальный перечень предметов. В результате выполнения функция возвращает оптимальную стоимость рюкзака, т. е. максимальную суммарную стоимость предметов, которые можно одновременно поместить в рюкзак заданной вместимости.

В процессе своей работы функция **knapsack\_s** использует генератор множества всех подмножеств (**combi::subset**) и вызывает три вспомогательные функции: **calcv**,  **calcc** и **setm**.

Для подключения генератора в заголовочный файл функции **knapsack\_s** (рис. 7)добавлена директива **include**, которая включает файл **Combi.h**, содержащий шаблон структуры **combi::subset**.

Функция **calcv** предназначена для вычисления суммарного объема текущего подмножества предметов. Она принимает два входных параметра: **s** (структуру **combi::subset**) и **v** (массив размерностью **n**, содержащий объемы всех предметов), а также возвращает суммарный объем текущего подмножества предметов.

Функция **calcс** позволяетвычислить суммарную стоимость текущего подмножества предметов. Онапринимает два входных параметра: **s** (структуру **combi::subset**) и **c** (массив размерностью **n**, содержащий стоимости всех предметов), а возвращает суммарную стоимость текущего подмножества предметов.

Функция **setm** принимает два параметра: входной **s** (структуру **combi::subset**) и возвращаемый **m (**массив размерностью **n**, содержащий нули и единицы).

Функция **knapsack\_s** перебирает все подмножества множества предметов (функции **getfirst** и **getnext** генератора), вычисляет суммарный объем (функция **calcv**) каждого подмножества, для подмножества с суммарным объемом, меньшим **V**, рассчитывает суммарную стоимость (функция **calcc**) и запоминает (и возвращает) оптимальный вариант (формирует выходной массив **m** спомощьюфункции **setm**).

17-19:

Пример вызова функции **knapsack\_s** для решения задачи о рюкзаке с исходными данными для схемы, представленной на рис. 6.

// Main

**#include "stdafx.h"**

**#include <iostream>**

**#include "Combi.h"**

**#include "Knapsack.h"**

**#define NN 4**

**int \_tmain(int argc, \_TCHAR\* argv[])**

**{**

**setlocale(LC\_ALL, "rus");**

**int V = 100,** // вместимость рюкзака

**v[] = {25, 30, 60, 20},** // размер предмета каждого типа

**c[] = {25, 10, 20, 30};** // стоимость предмета каждого типа

**short m[NN];** // количество предметов каждого типа {0,1}

**int maxcc = knapsack\_s(**

**V,** // [in] вместимость рюкзака

**NN,** // [in] количество типов предметов

**v,** // [in] размер предмета каждого типа

**c,** // [in] стоимость предмета каждого типа

**m** // [out] количество предметов каждого типа

**);**

**std::cout<<std::endl<<"-------- Задача о рюкзаке --------- ";**

**std::cout<<std::endl<<"- количество предметов : "<< NN;**

**std::cout<<std::endl<<"- вместимость рюкзака : "<< V;**

**std::cout<<std::endl<<"- размеры предметов : ";**

**for(int i = 0; i < NN; i++) std::cout<<v[i]<<" ";**

**std::cout<<std::endl<<"- стоимости предметов : ";**

**for(int i = 0; i < NN; i++) std::cout<<v[i]\*c[i]<<" ";**

**std::cout<<std::endl<<"- оптимальная стоимость рюкзака: " << maxcc;**

**std::cout<<std::endl<<"- вес рюкзака: ";**

**int s = 0; for(int i = 0; i < NN; i++) s+= m[i]\*v[i];**

**std::cout<<s;**

**std::cout<<std::endl<<"- выбраны предметы: ";**

**for(int i = 0; i < NN; i++) std::cout<<" "<<m[i];**

**std::cout<<std::endl<<std::endl;**

**system("pause");**

**return 0;**

**}**

Рис. 9. Пример использования функции **knapsack\_s**

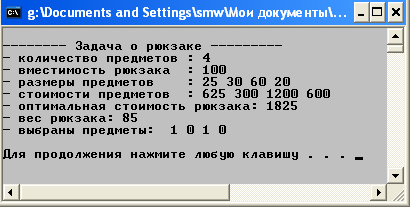


Рис. 10. Результат выполнения программы, представленной на рис. 9

20-21:

Оценить зависимость продолжительности вычисления оптимальной комбинации предметов от их общего количества можно с помощью следующей программы.

// Main

**#include "stdafx.h"**

**#include <iostream>**

**#include "Combi.h"**

**#include "Knapsack.h"**

**#include <time.h>**

**#include <iomanip>**

**#define NN (sizeof(c)/sizeof(int))**

**int \_tmain(int argc, \_TCHAR\* argv[])**

**{**

**setlocale(LC\_ALL, "rus");**

**int V = 600,** // вместимость рюкзака

**v[] = {25, 56, 67, 40, 20, 27, 37, 33, 33, 44, 53, 12,**

**60, 75, 12, 55, 54, 42, 43, 14, 30, 37, 31, 12},**

**c[] = {15, 26, 27, 43, 16, 26, 42, 22, 34, 12, 33, 30,**

**12, 45, 60, 41, 33, 11, 14, 12, 25, 41, 30, 40};**

**short m[NN];**

**int maxcc = 0;**

**clock\_t t1, t2;**

**std::cout<<std::endl<<"-------- Задача о рюкзаке --------- ";**

**std::cout<<std::endl<<"- вместимость рюкзака : "<< V;**

**std::cout<<std::endl<<"-- количество ------ продолжительность -- ";**

**std::cout<<std::endl<<" предметов вычисления ";**

**for (int i = 14; i <= NN; i++)**

**{**

**t1 = clock();**

**maxcc = knapsack\_s(V, i, v, c, m );**

**t2 = clock();**

**std::cout<<std::endl<<" "<<std::setw(2)<<i**

**<<" "<<std::setw(5)<<(t2-t1);**

**}**

**std::cout<<std::endl<<std::endl;**

**system("pause");**

**return 0;**

**}**

Рис. 11. Вычисление продолжительности решения задачи о рюкзаке при различном количестве предметов

Для вычисления продолжительности выполнения функции **knapsack\_s** в программе используется стандартная функция **clock**, возвращающая количество условных единиц времени, прошедших с момента запуска программы. Разница между возвращаемыми значениями функции **clock**,полученными до вызова **knapsack\_s** и после, позволяет оценить продолжительность выполнения этой функции.

22:

Результат вычисления продолжительности выполнения функции **knapsack\_s** в зависимости от общего количества предметов в задаче о рюкзаке.

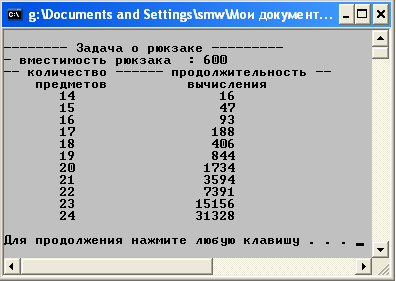


Рис. 12. Результат выполнения программы, представленной на рис. 11

23:

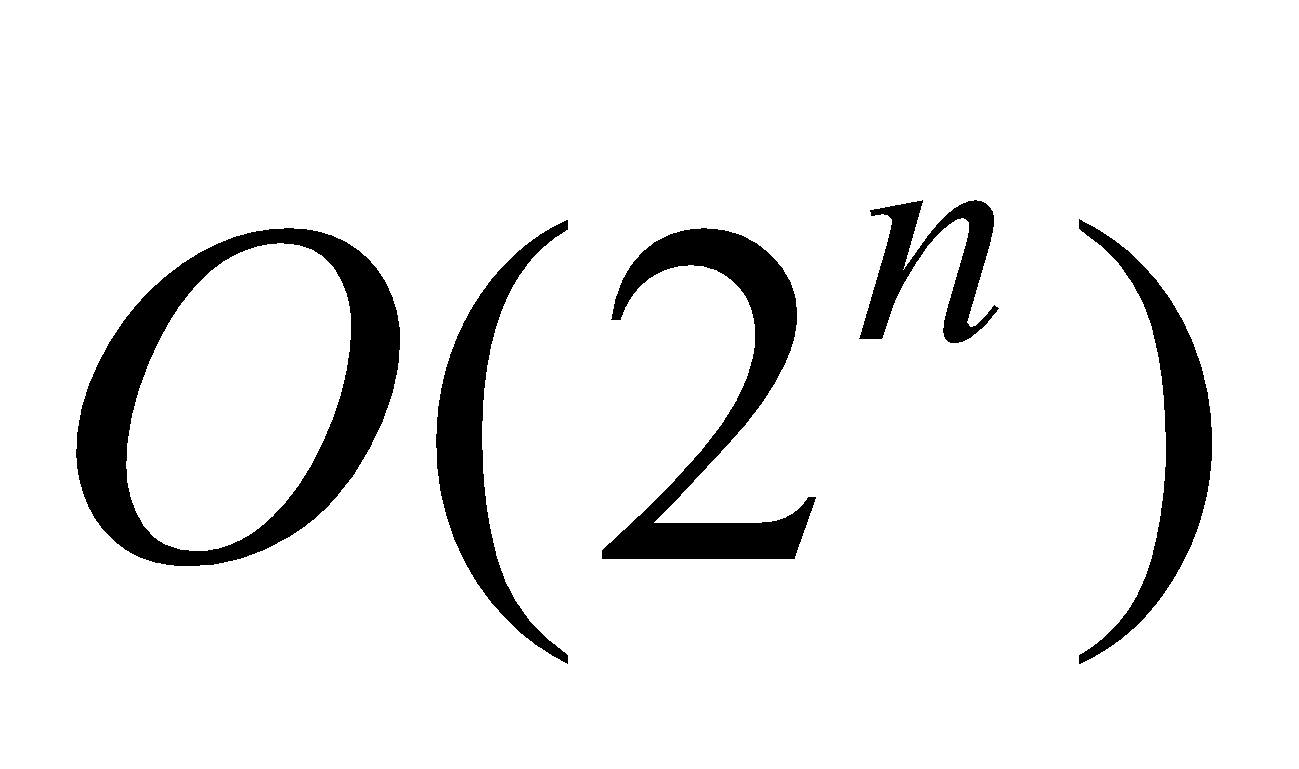
График, построенный по результатам выполнения программы. Несложно заметить, что с увеличением количества предметов (параметр **n** функции **knapsack\_s**) наединицу продолжительность выполнения функции **knapsack\_s** удваивается. Такой результат согласуется с оценкой  сложности алгоритма генерации булеана множества мощности **n**.

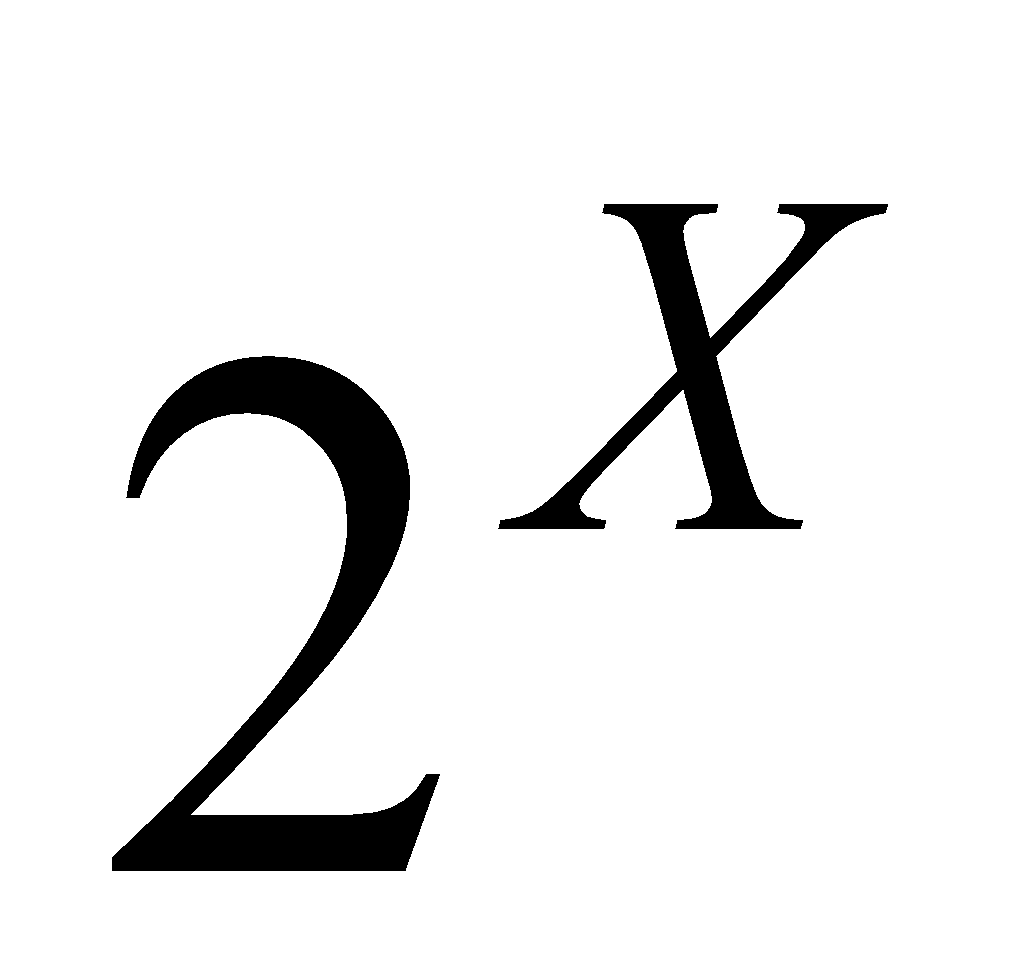
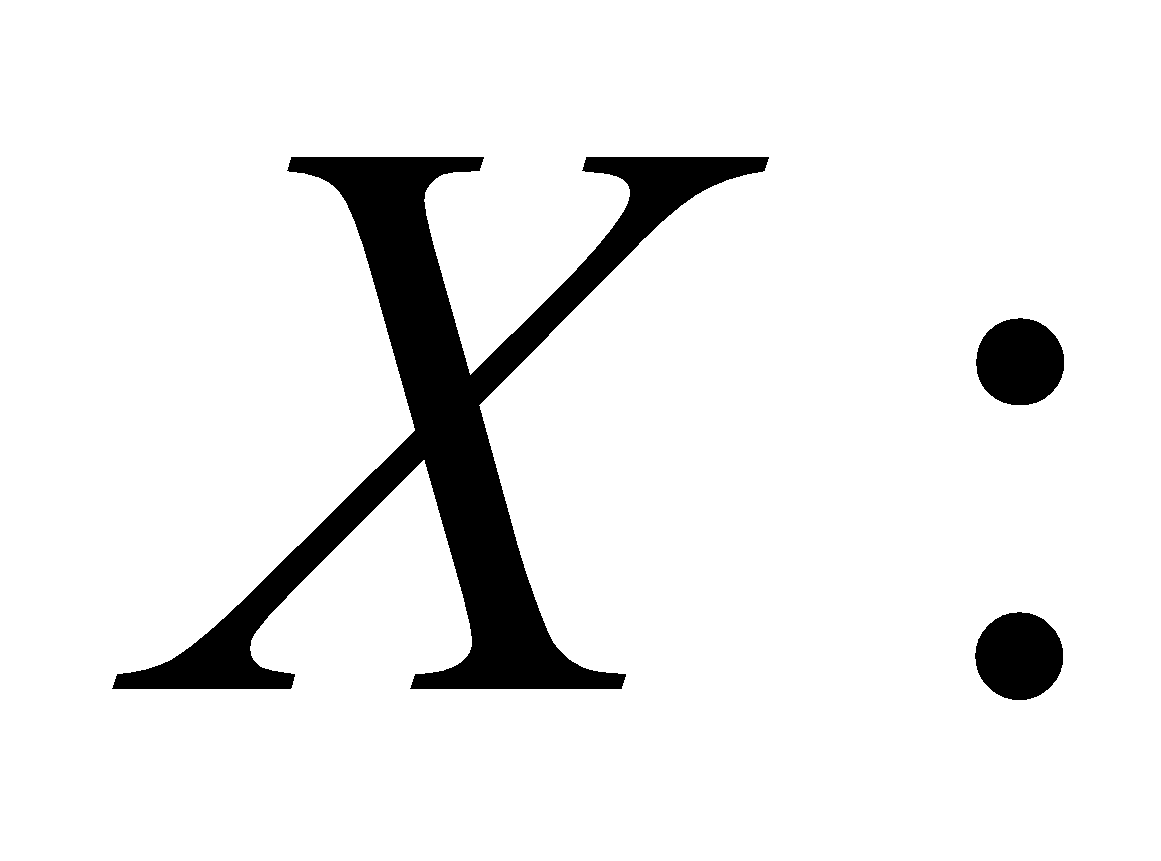
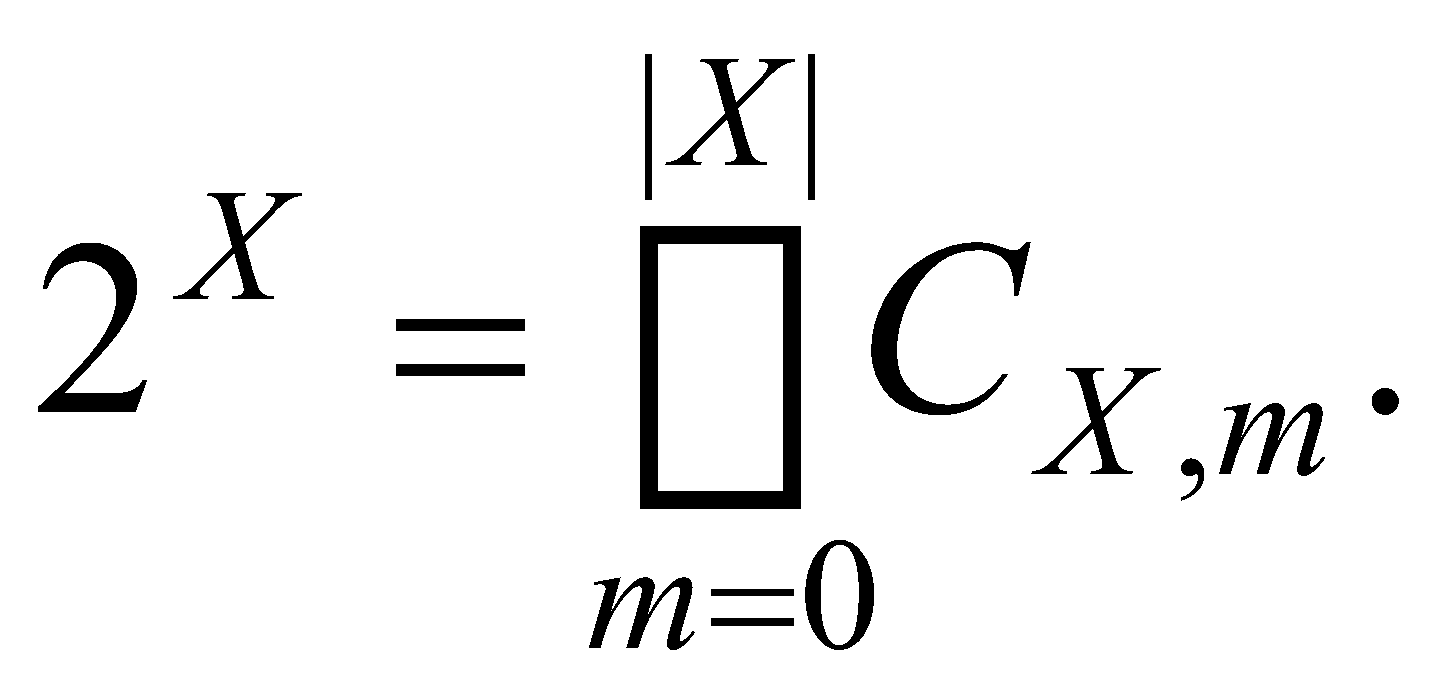
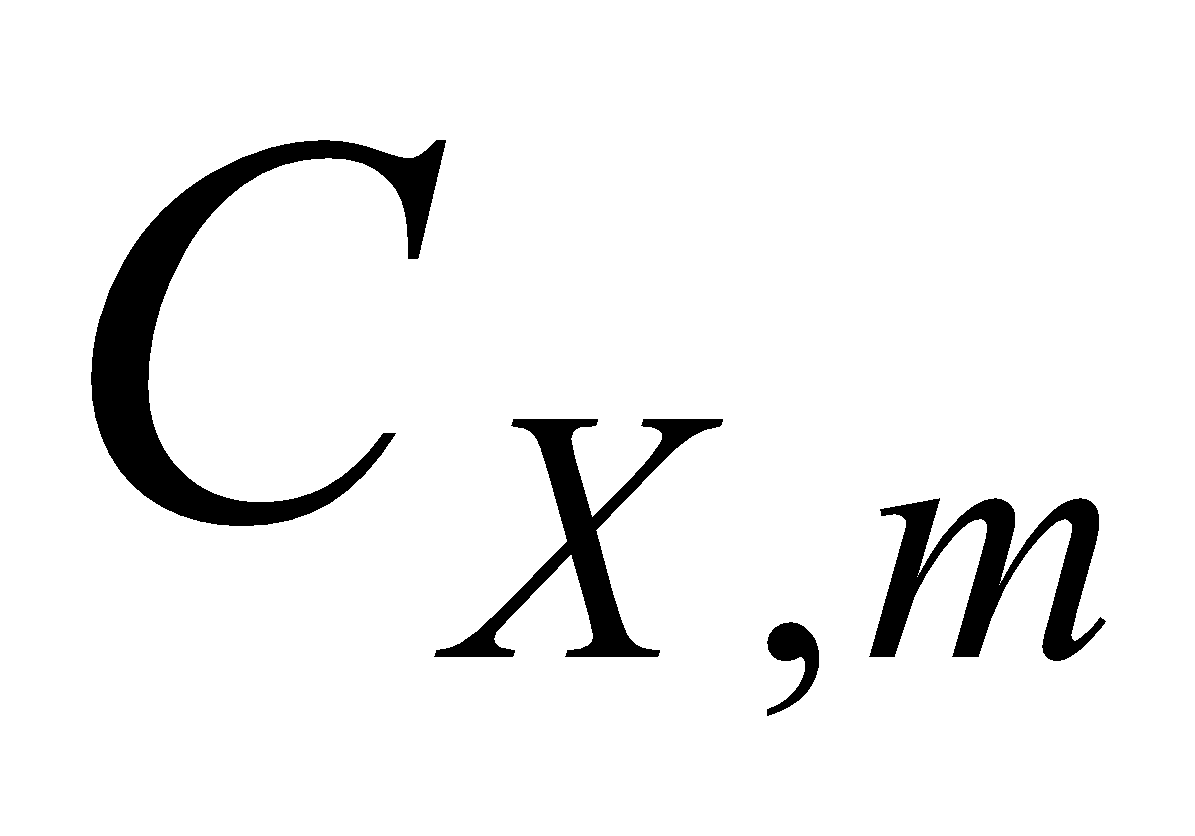
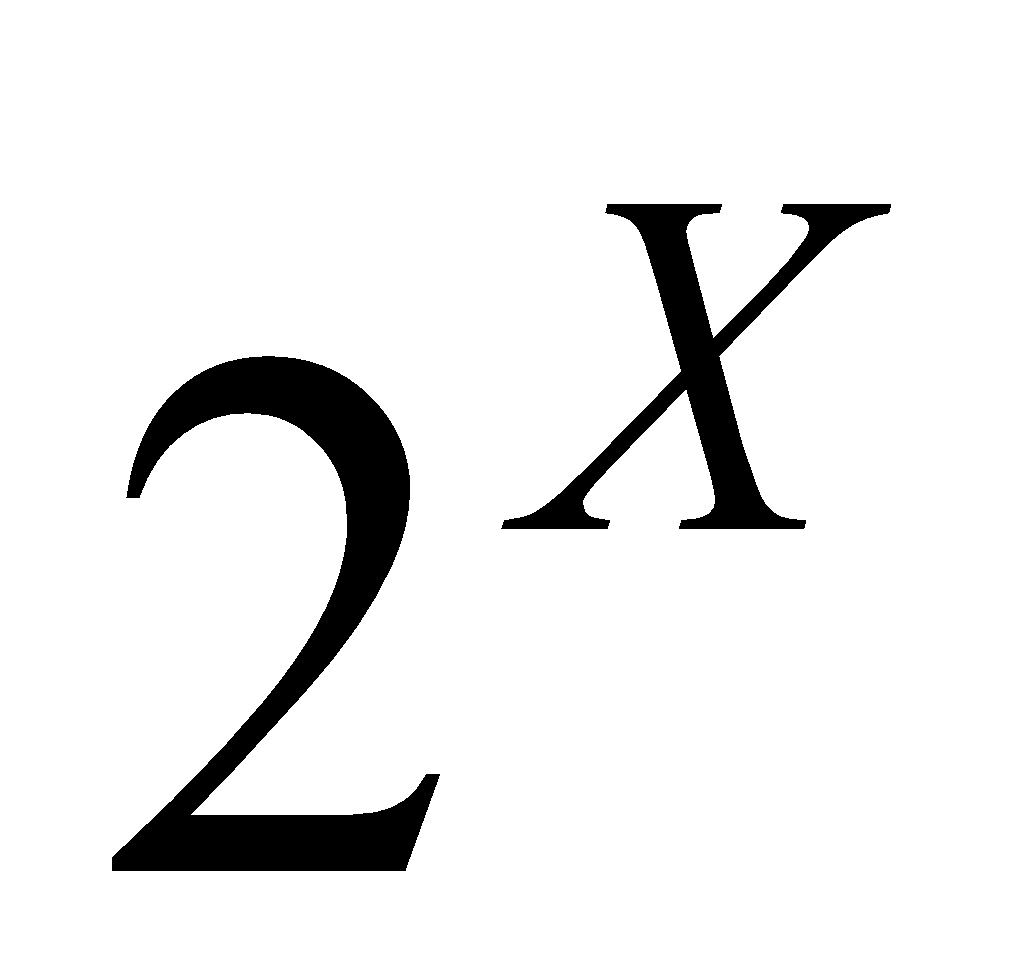
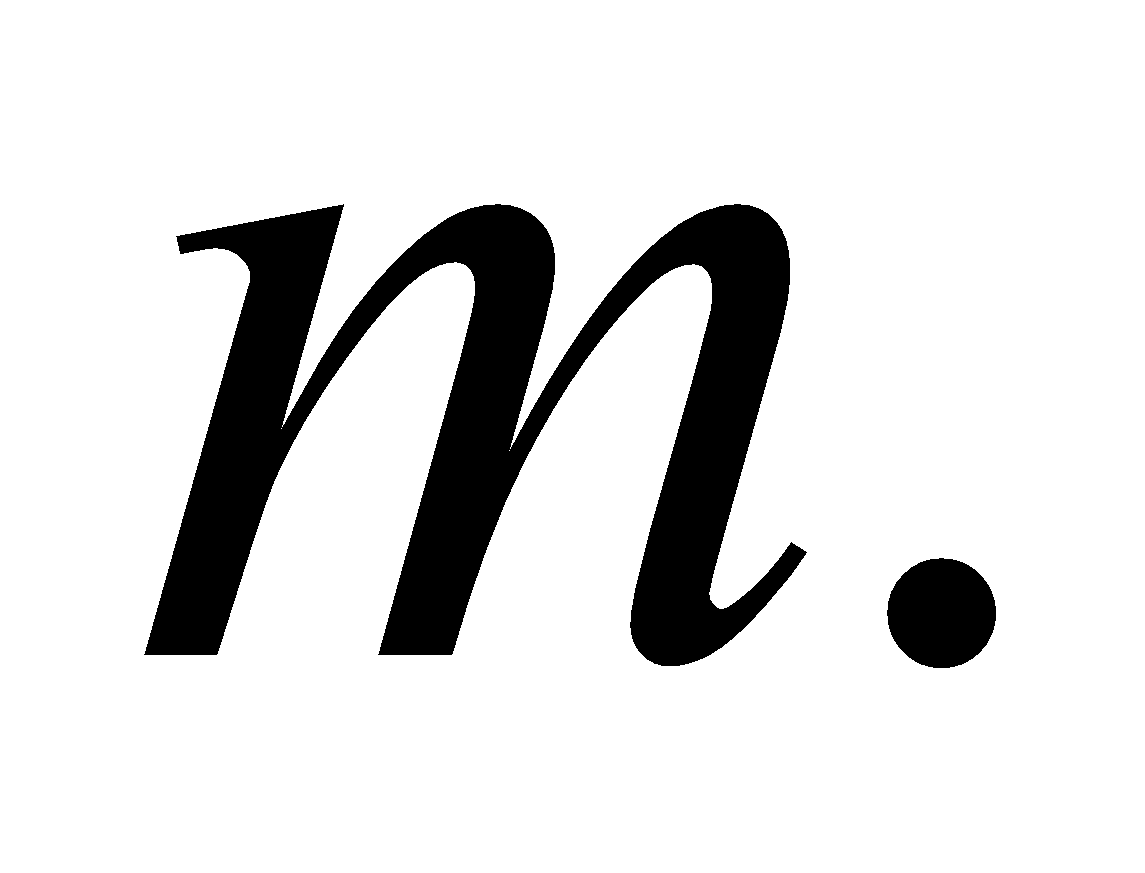


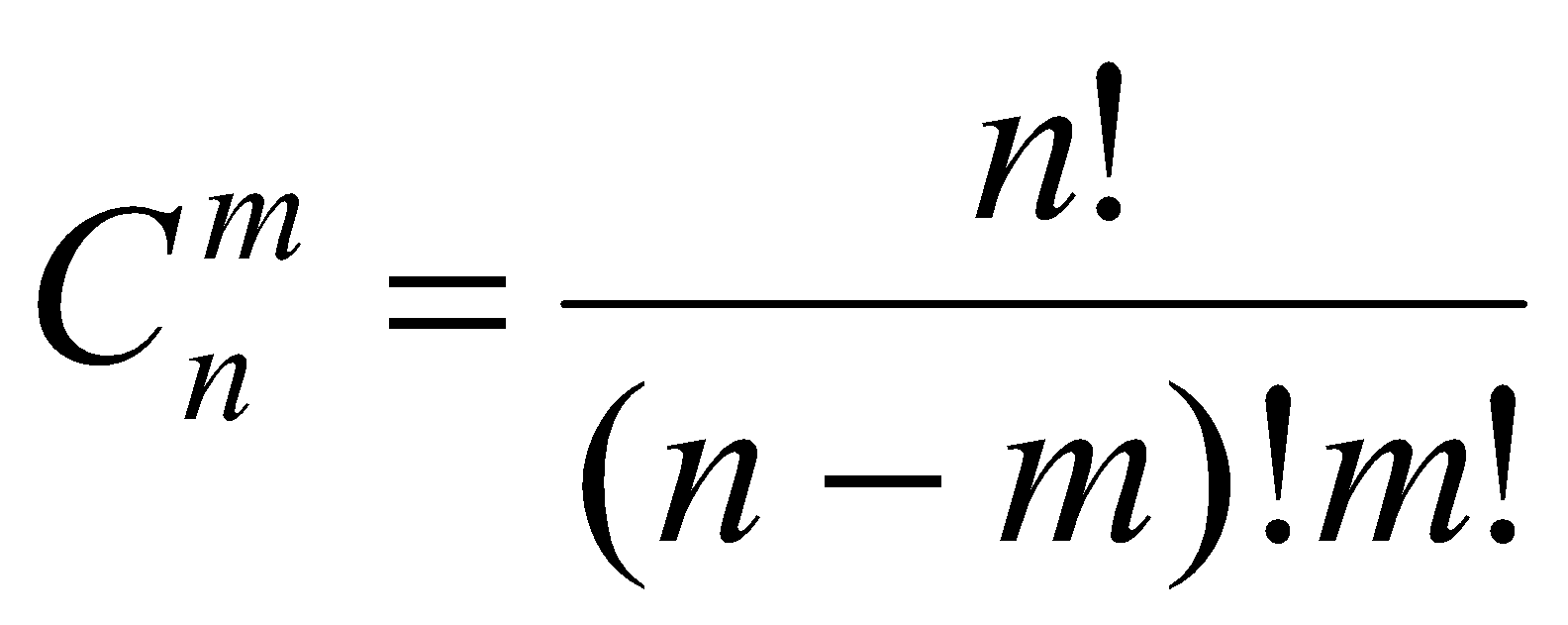
Рис. 13. Зависимость продолжительности выполнения функции **knapsack\_s** от значения ее параметра n

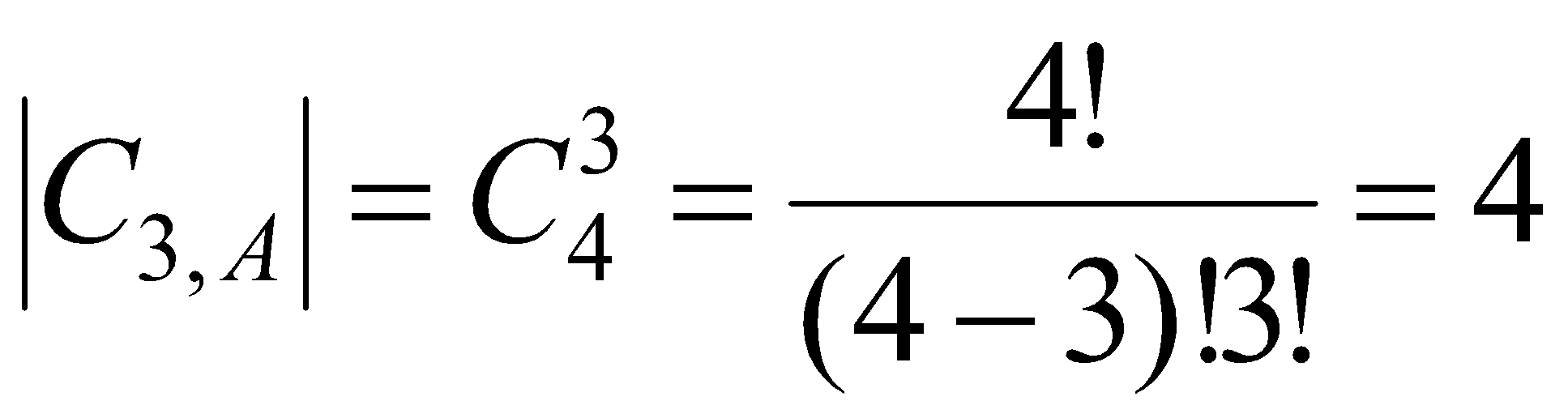
Эксперимент (рис. 12 и 13) проводился на компьютере с процессором Pentium V с тактовой частотой 3,2 ГГц. Простая экстраполяция результатов этого эксперимента дает следующие результаты: для 30 предметов продолжительность вычисления будет составлять более 24 мин., для 40 – более 16 сут., а для 45 – почти два года.

24:

1. **Генерация сочетаний**

Булеан  можно рассматривать как объединение всевозможных сочетаний, построенных из элементов множества   Поэтому генерация множества  может быть сведена к генерации булеана и выбору из него всех подмножеств с мощностью 

**** (2)

** **

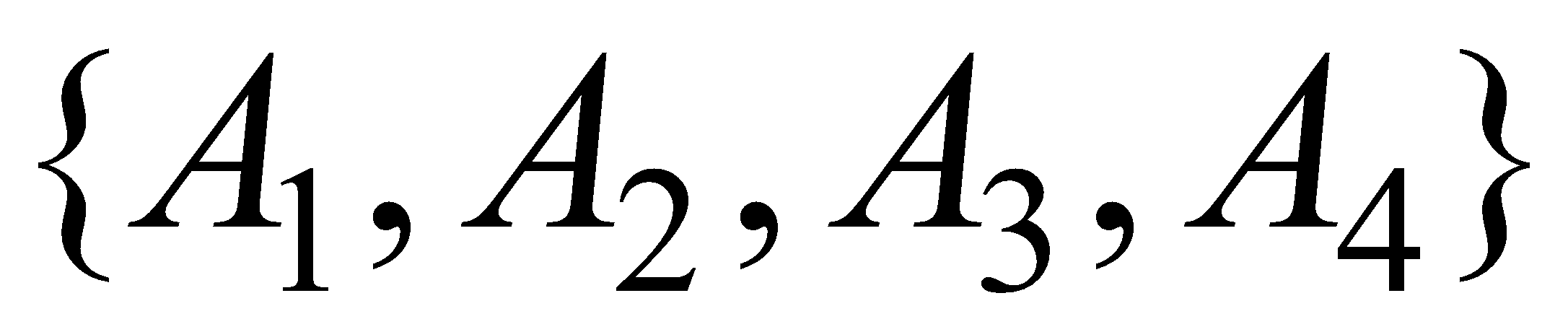
На рис. 14 представлена схема построения множества сочетаний  из элементов множества  Закрашенным прямоугольником на рисунке обозначены номера (индексы) элементов битовых последовательностей   и элементов множества  Стрелки связывают битовые последовательности, содержащие три двоичные единицы и сгенерированные сочетания множества  Для каждой стрелки указаны индексы единичных позиций соответствующих битовых последовательностей. Эти индексы используются для выбора элементов из множества для включения в соответствующее сочетание. Очевидно, что такой алгоритм генерации сочетаний имеет сложность  как и алгоритм генерации множества всех подмножеств.

25:



Рис.14. Схема генерации сочетаний на основе множества всех подмножеств

26:

Рассмотрим еще один пример. Имеется множество всех подмножеств: . Всего подмножеств 16. Требуется найти все сочетания по три.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Порядковый номер | Запись в двоичном коде | | | | Запись подмножеств в общем виде | Требуемые сочетания по три |
| 0 | 0 | 0 | 0 | 0 |  |  |
| 1 | 0 | 0 | 0 | 1 |  |  |
| 2 | 0 | 0 | 1 | 0 |  |  |
| 3 | 0 | 0 | 1 | 1 |  |  |
| 4 | 0 | 1 | 0 | 0 |  |  |
| 5 | 0 | 1 | 0 | 1 |  |  |
| 6 | 0 | 1 | 1 | 0 |  |  |
| 7 | 0 | 1 | 1 | 1 |  |  |
| 8 | 1 | 0 | 0 | 0 |  |  |
| 9 | 1 | 0 | 0 | 1 |  |  |
| 10 | 1 | 0 | 1 | 0 |  |  |
| 11 | 1 | 0 | 1 | 1 |  |  |
| 12 | 1 | 1 | 0 | 0 |  |  |
| 13 | 1 | 1 | 0 | 1 |  |  |
| 14 | 1 | 1 | 1 | 0 |  |  |
| 15 | 1 | 1 | 1 | 1 |  |  |

27-29:

На рис. 15 и 16 представлена реализация генератора сочетаний на языке С++. Генератор реализован в виде структуры **xcombination**.

// Combi.h

**#pragma once**

**namespace combi**

**{**

**struct xcombination** // генератор сочетаний (эвристика)

**{**

**short n,** // количество элементов исходного множества

**m,** // количество элементов в сочетаниях

**\*sset;** // массив индексов текущего сочетания

**xcombination (**

**short n = 1,** //количество элементов исходного множества

**short m = 1** // количество элементов в сочетаниях

**);**

**void reset();** // сбросить генератор, начать сначала

**short getfirst();** // сформировать первый массив индексов

**short getnext();** // сформировать следующий массив индексов

**short ntx(short i);** // получить i-й элемент массива индексов

**unsigned \_\_int64 nc;** // номер сочетания 0,..., count()-1

**unsigned \_\_int64 count() const;** // вычислить количество сочетаний

**};**

**};**

Рис. 15. Шаблон структуры генератора сочетаний

// Combi.cpp

**#include "stdafx.h"**

**#include "Combi.h"**

**#include <algorithm>**

**namespace combi**

**{**

**xcombination::xcombination (short n, short m)**

**{**

**this->n = n;**

**this->m = m;**

**this->sset = new short[m+2];**

**this->reset();**

**}**

**void xcombination::reset()** // сбросить генератор, начать сначала

**{**

**this->nc = 0;**

**for(int i = 0; i < this->m; i++) this->sset[i] = i;**

**this->sset[m] = this->n;**

**this->sset[m+1] = 0;**

**};**

**short xcombination::getfirst()**

**{ return (this->n >= this->m)?this->m:-1; };**

**short xcombination::getnext()** // сформировать следующий массив индексов

**{**

**short rc = getfirst();**

**if (rc > 0)**

**{**

**short j;**

**for (j = 0; this->sset[j]+1 == this->sset[j+1]; ++j)**

**this->sset[j] = j;**

**if (j >= this->m) rc = -1;**

**else {**

**this->sset[j]++;**

**this->nc++;**

**};**

**}**

**return rc;**

**};**

**short xcombination::ntx(short i)**

**{ return this->sset[i]; };**

**unsigned \_\_int64 fact(unsigned \_\_int64 x){return(x == 0)?1:(x\*fact(x-1));};**

**unsigned \_\_int64 xcombination::count() const**

**{**

**return (this->n >= this->m)?**

**fact(this->n)/(fact(this->n-this->m)\*fact(this->m)):0;**

**};**

**};**

Рис. 16. Реализация функций генератора сочетаний

Структура **xcombination** имеет один конструктор с двумя параметрами. Первый параметр определяет количество элементов в исходном множестве, второй – количество элементов в генерируемых сочетаниях.

Для хранения текущего состояния генератора используются три переменные: **n** (мощность исходного множества), **m** (количество элементов в генерируемых сочетаниях), **sset** (адрес нулевого элемента массива индексов) и **nc** (номер текущего сочетания). Все переменныеинициализируются в конструкторе. Значение **nc** увеличивается на единицу после формирования очередного сочетания, а значение остальных переменных остается постоянным.

Кроме конструктора, структура **xcombination** содержит еще пять функций.

Функция **getfirst** (рис. 16) не имеет параметров и предназначена для проверки корректности параметров, заданных в конструкторе. Эта функция не формирует массива индексов, как это происходило в структуре **subset** (генератор множества всех подмножеств). В основном она существует для унификации интерфейсов всех генераторов. Функция возвращает отрицательное значение, если параметры генератора заданы неверно.

Функция **getnext** формирует массив индексов следующего сочетания и увеличивает значение переменной **nc** на единицу. При каждом вызове функции для текущего массива индексов вычисляется новое значение *j*-индекса и, если оно не превышает **m**, строится новый массив индексов. При достижении *j*-индексом значения, равного или превышающего **m**, функция возвращает отрицательное значение, в других случаях возвращается положительное значение.

Функция **ntx** возвращает значение элемента массива индексов по индексу этого элемента и служит для сокращения записи при переборе элементов массива.

Функция **count** вычисляет и возвращает общее количество сочетаний из **n** по **m**.

Как и в генераторе множества всех подмножеств, для сброса генератора сочетаний в начальное состояние служит функция **reset**. После вызова **reset** снова могут вызываться **getfirst** и **getnext**.

30-32:

Пример использования генератора сочетаний и результат выполнения программы.

// Main

**#include "stdafx.h"**

**#include <iostream>**

**#include "Combi.h"**

**int \_tmain(int argc, \_TCHAR\* argv[])**

**{**

**setlocale(LC\_ALL, "rus");**

**char AA[][2]= {"A", "B", "C", "D", "E"};**

**std::cout<<std::endl<<" --- Генератор сочетаний ---";**

**std::cout<<std::endl<<"Исходное множество: ";**

**std::cout<<"{ ";**

**for (int i = 0; i < sizeof(AA)/2; i++)**

**std::cout<<AA[i]<<((i< sizeof(AA)/2-1)?", ":" ");**

**std::cout<<"}";**

**std::cout<<std::endl<<"Генерация сочетаний ";**

**combi::xcombination xc(sizeof(AA)/2, 3);**

**std::cout<<"из "<<xc.n<< " по "<< xc.m;**

**int n = xc.getfirst();**

**while (n >= 0)**

**{**

**std::cout<<std::endl<<xc.nc <<": { ";**

**for (int i = 0; i < n; i++)**

**std::cout<<AA[xc.ntx(i)]<<((i< n-1)?", ":" ");**

**std::cout<<"}";**

**n = xc.getnext();**

**};**

**std::cout<<std::endl<<"всего: " << xc.count()<<std::endl;**

**system("pause");**

**return 0;**

**}**

Рис. 17. Пример применения генератора сочетаний

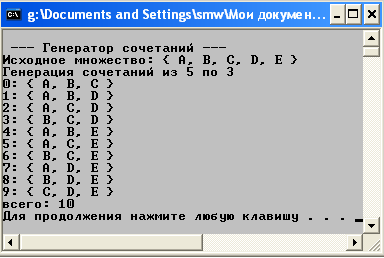


Рис. 18. Результат выполнения программы, представленной на рис. 17

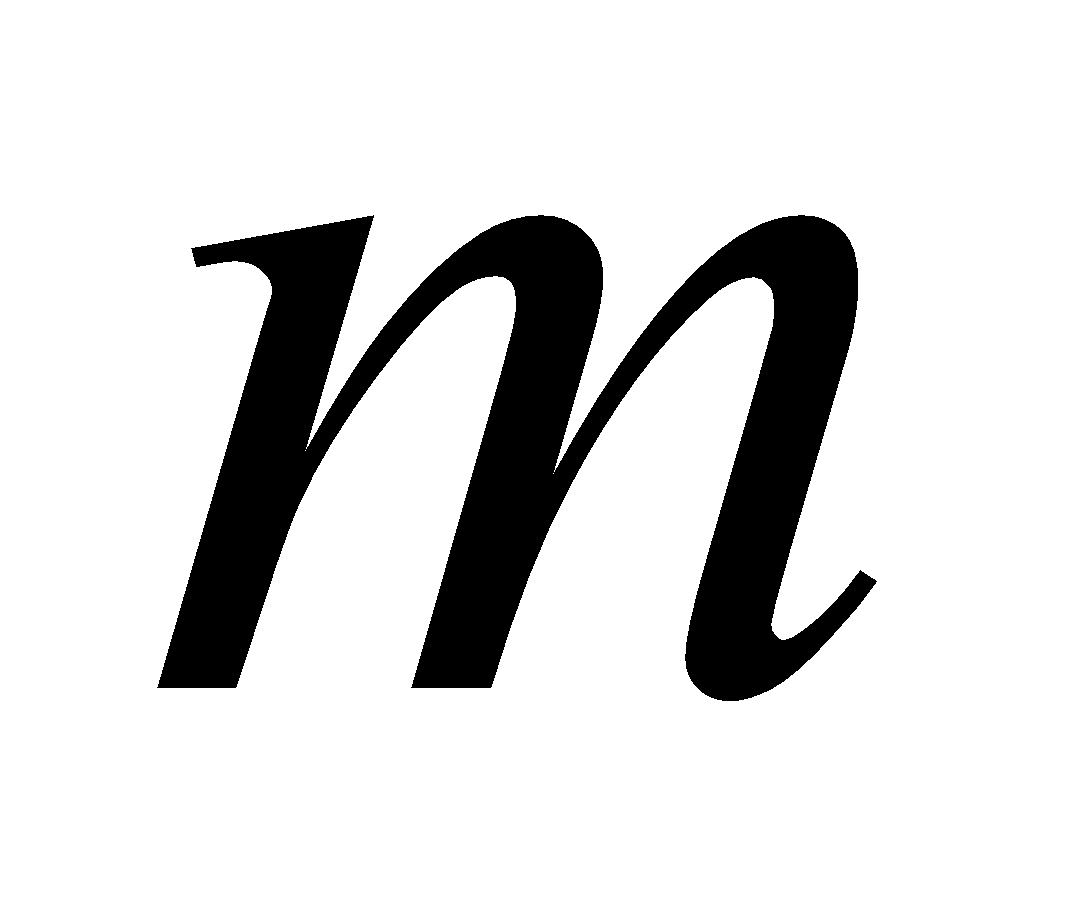
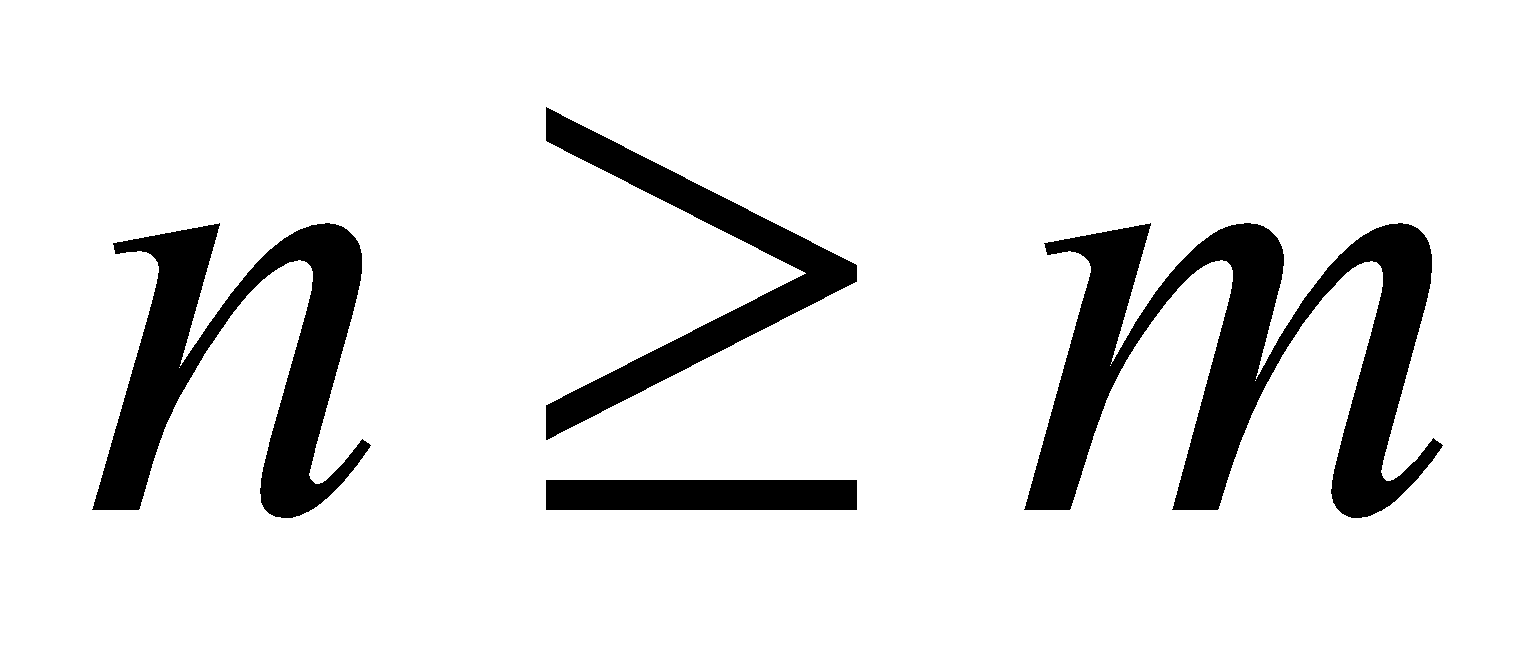
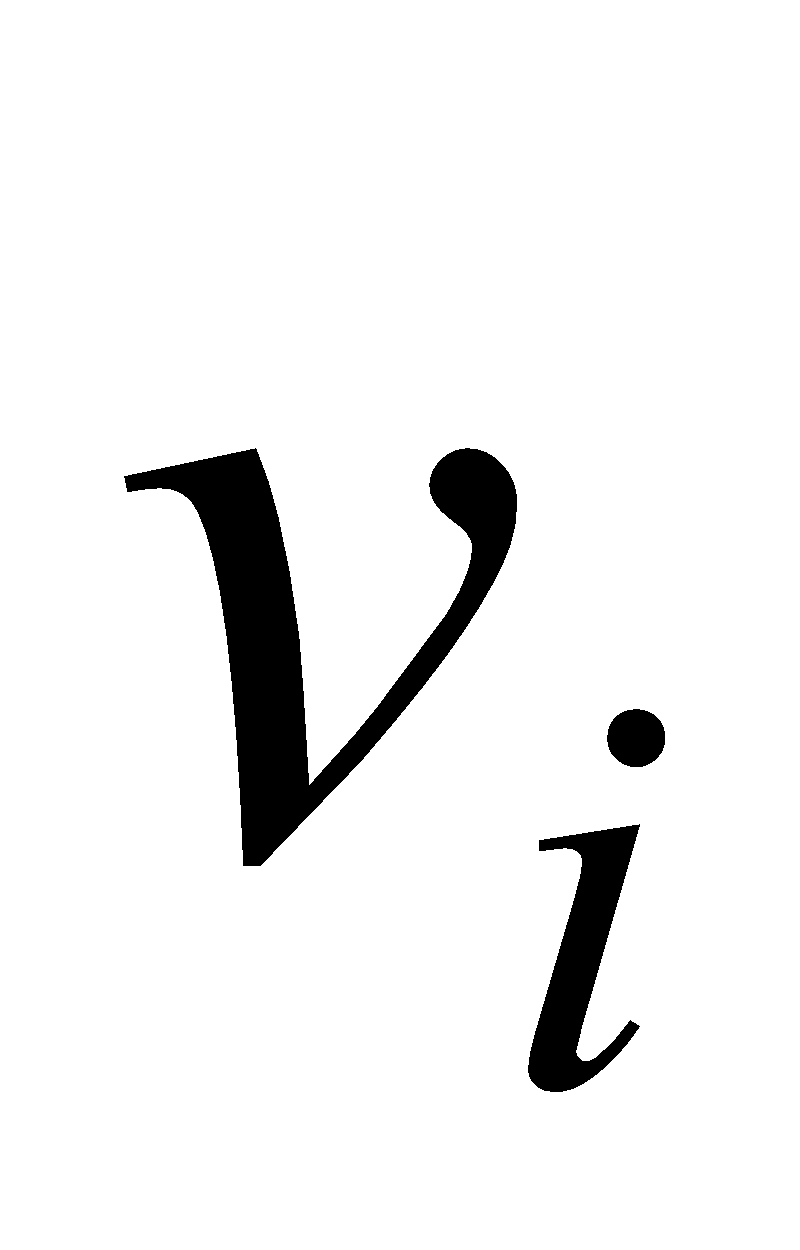
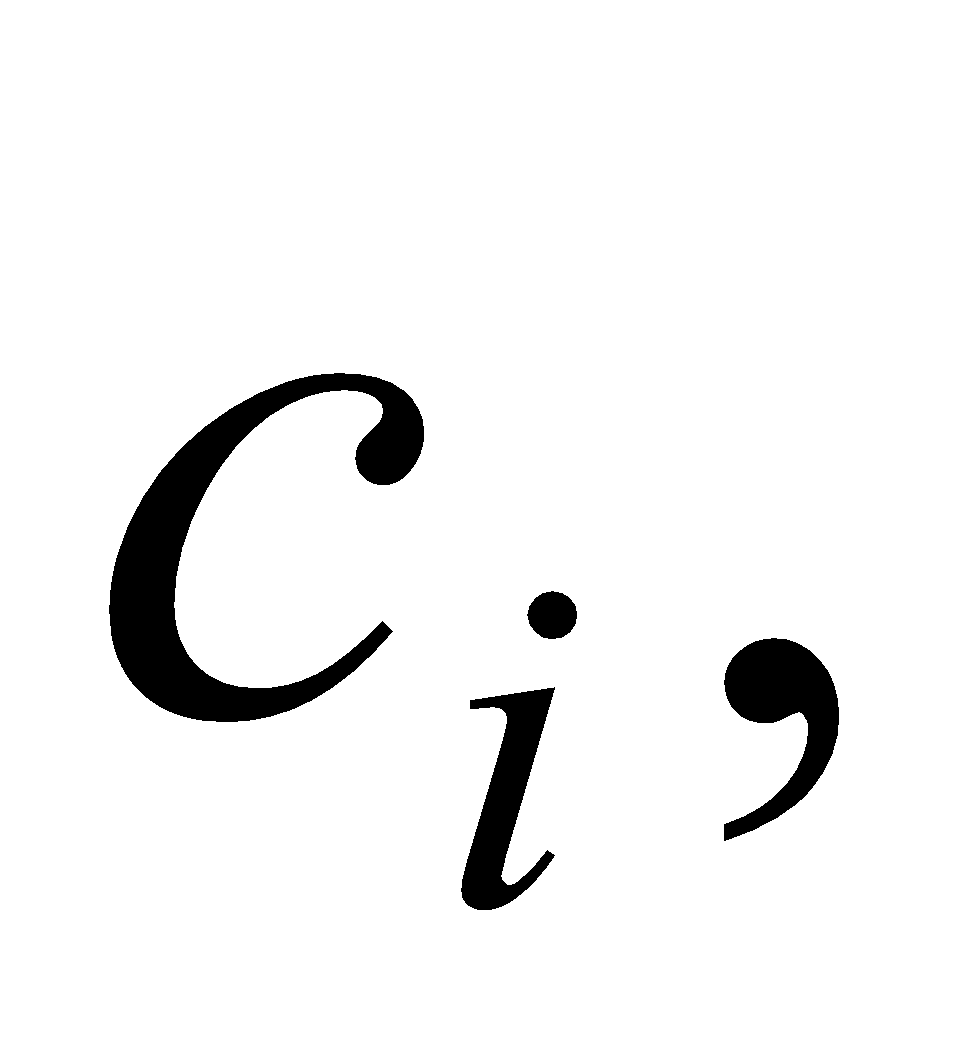
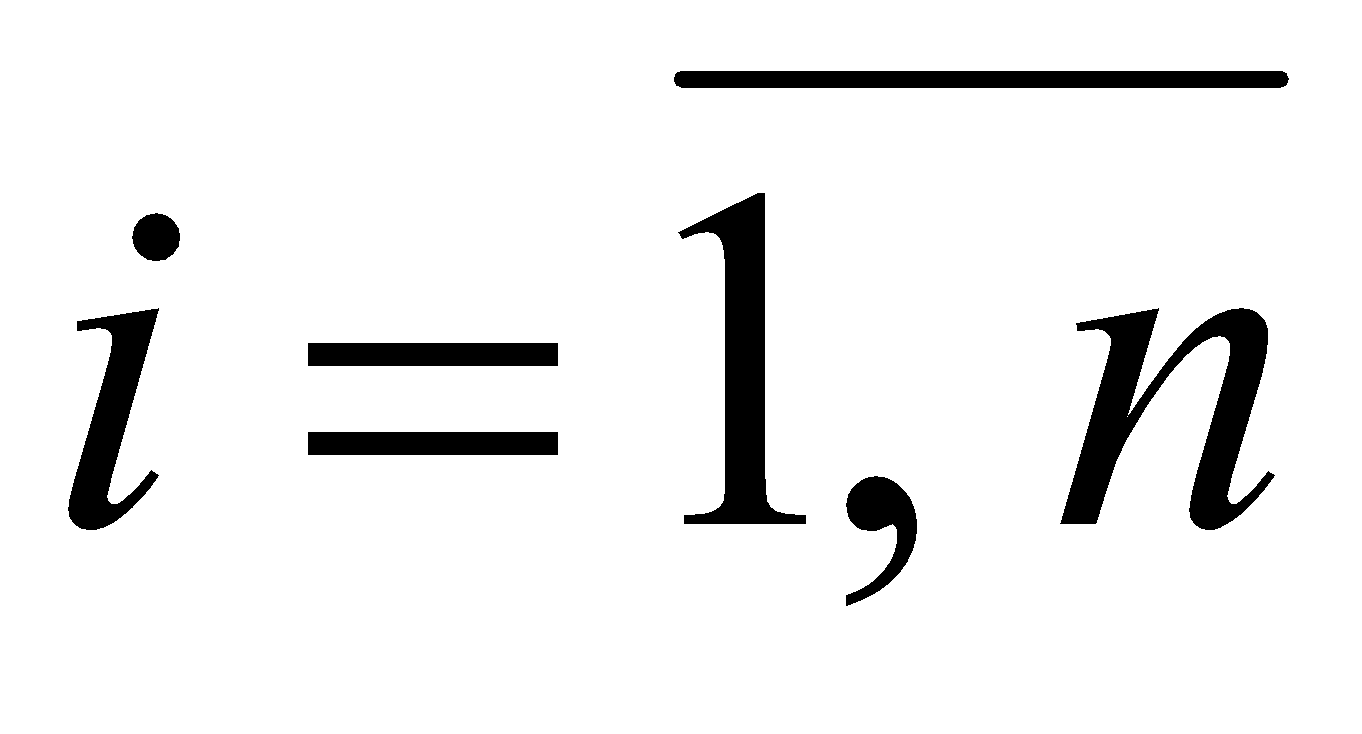
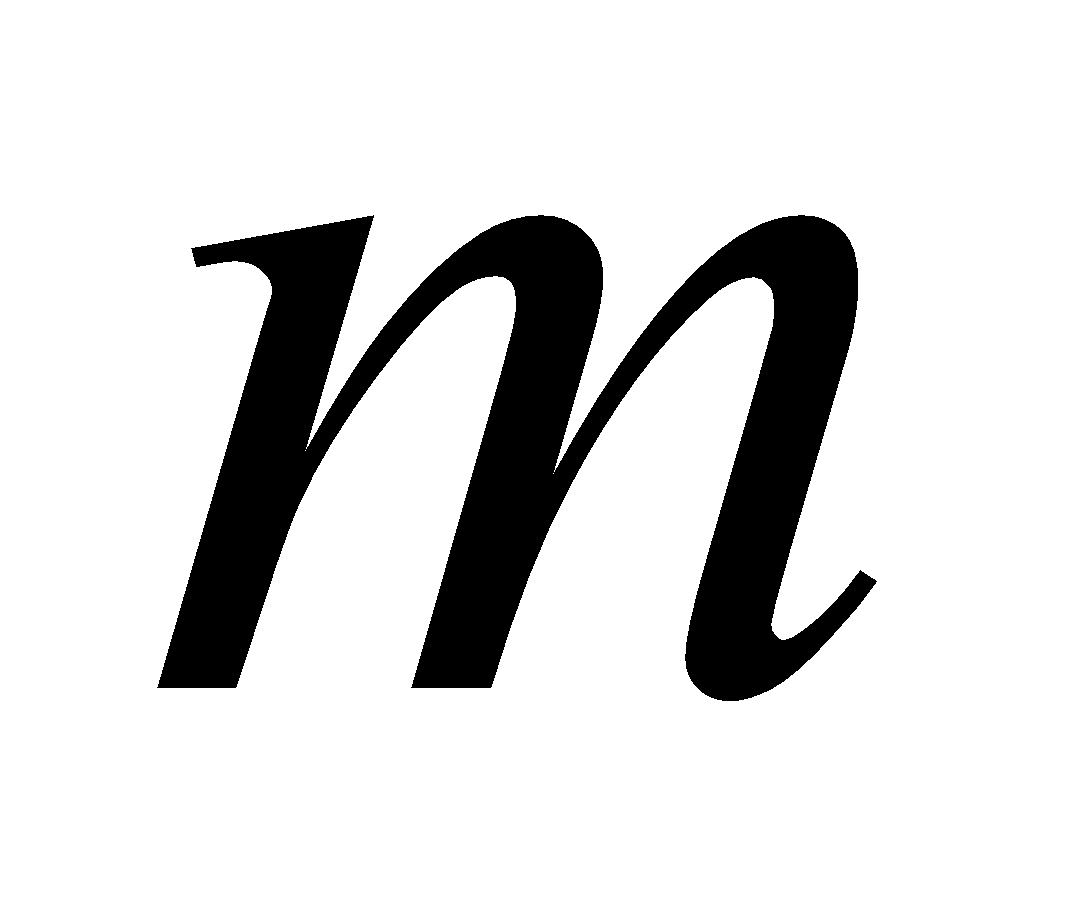
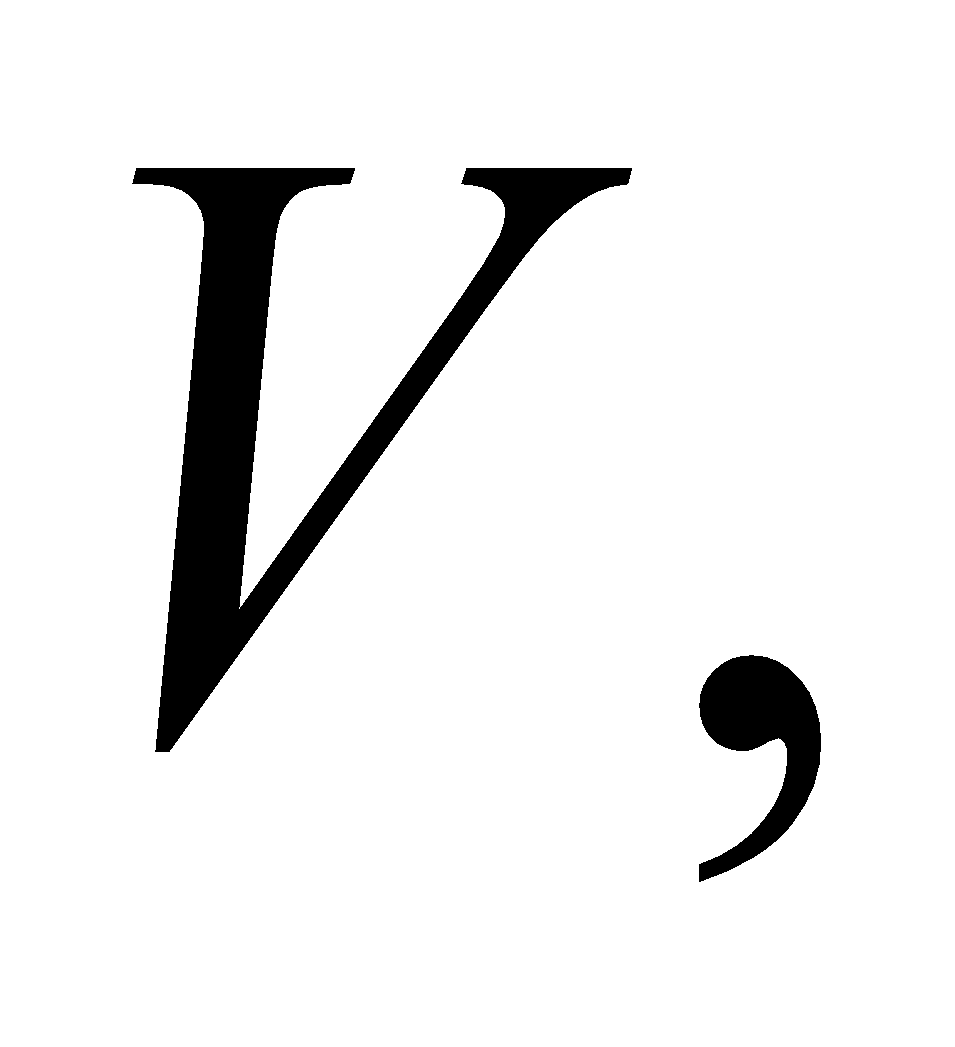
В качестве исходного множества в примере используется строковый массив, состоящий из пяти элементов. Вначале программы этот массив распечатывается. Далее объявляется структура **xcombination** и ее конструктору в качестве параметров передаются количество элементов исходного множества и размерность сочетаний.

Генерация сочетаний начинается функцией **getfirst**.Если функция возвращает положительное значение, то сформирован индекс массивов первого сочетания. Массив индексов каждого следующего сочетания формируется функцией **getnext**. Выбор элементов исходного массива осуществляется с помощью функции **ntx**. Признаком завершения цикла генерации является отрицательное значение функции **getnext**.

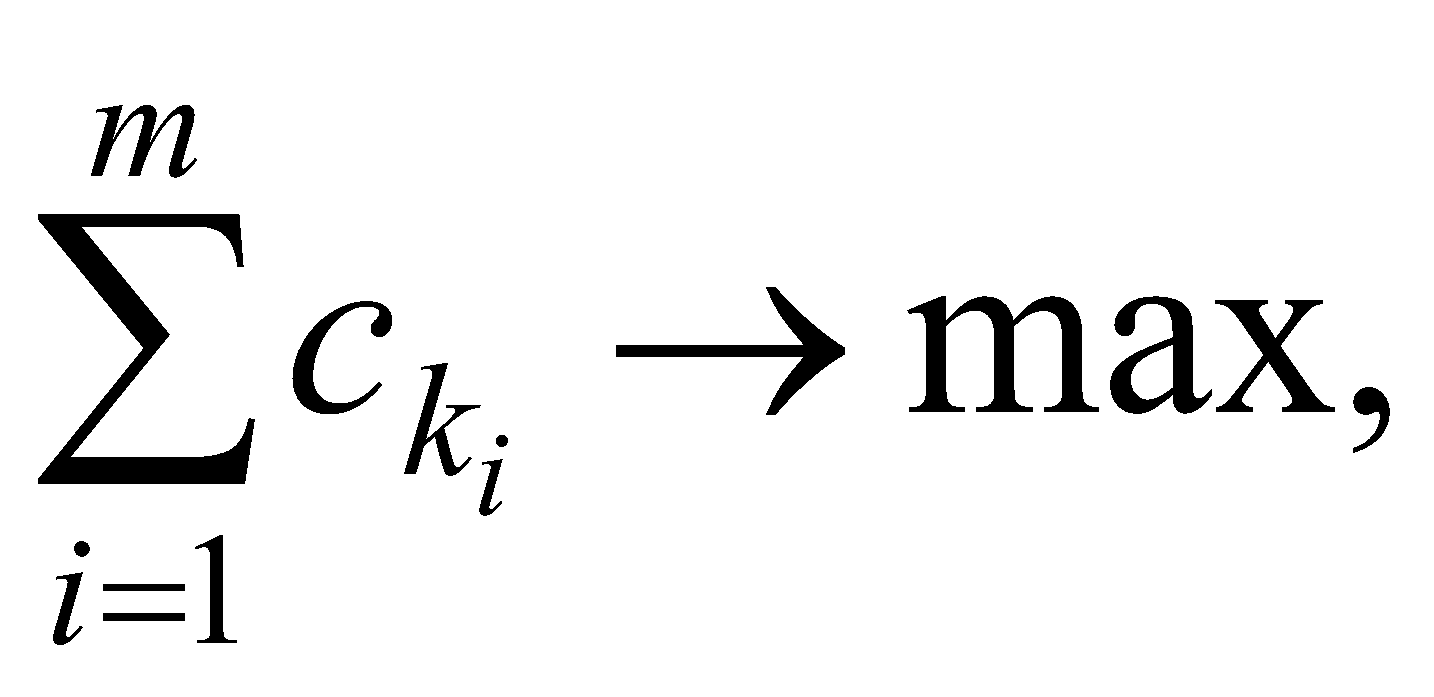
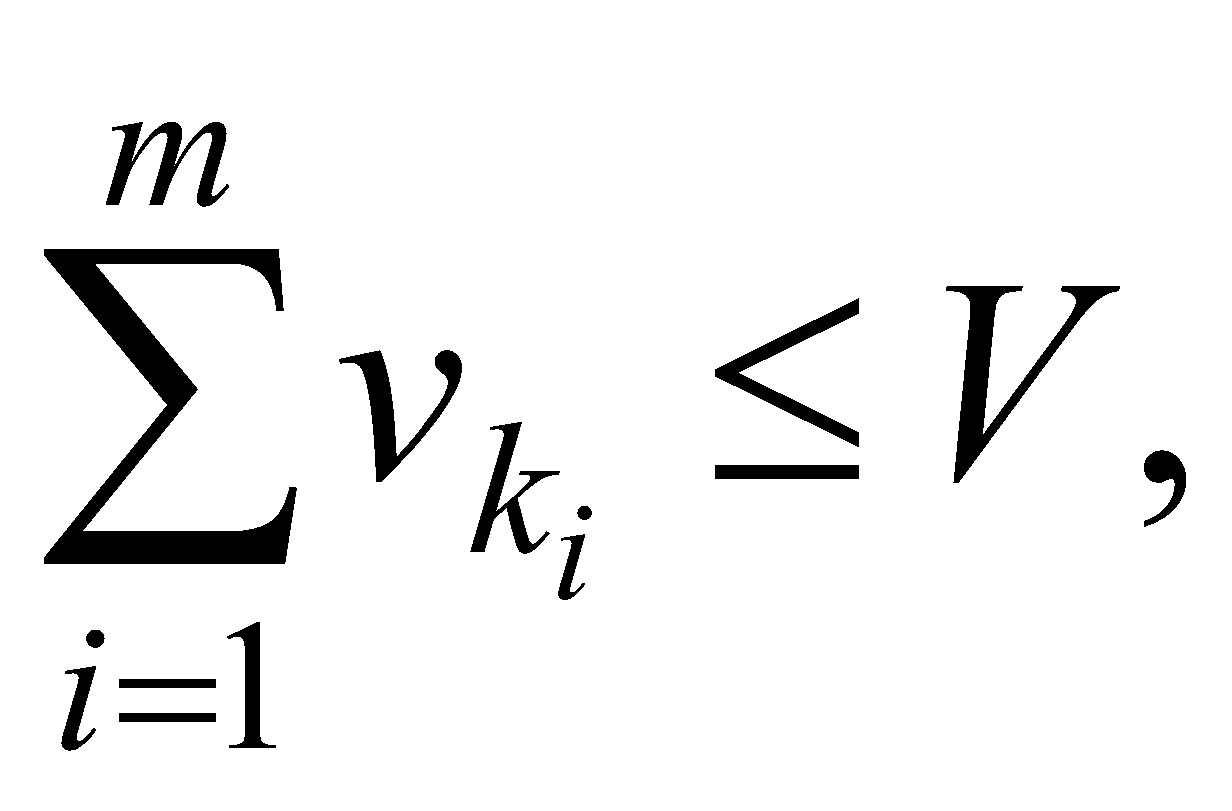
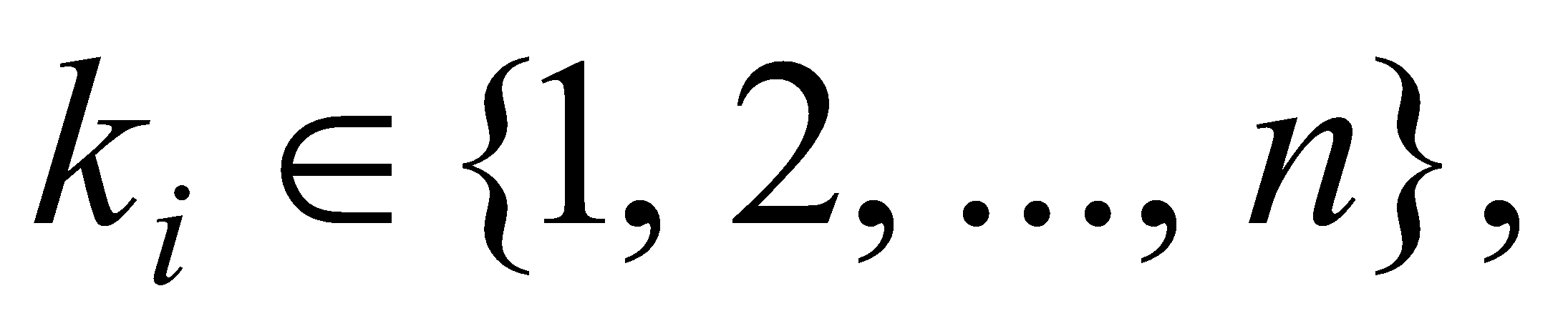
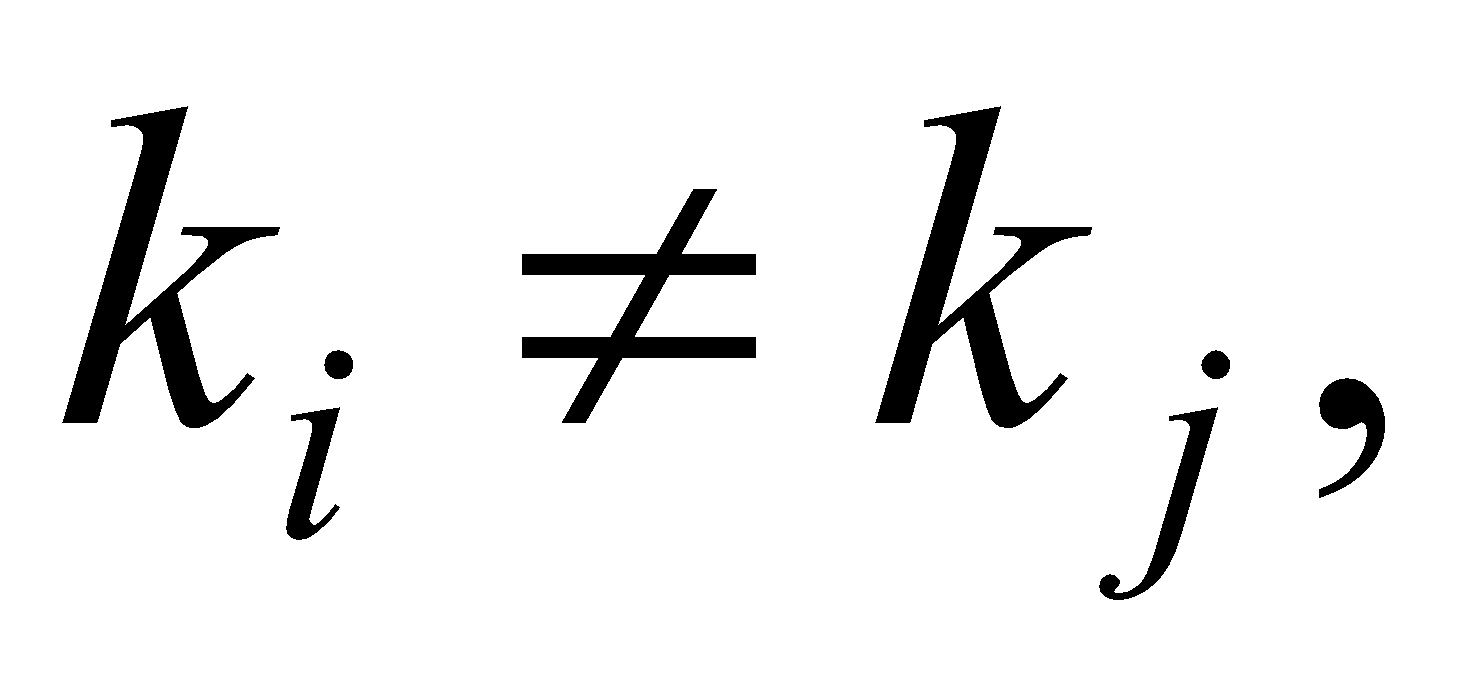
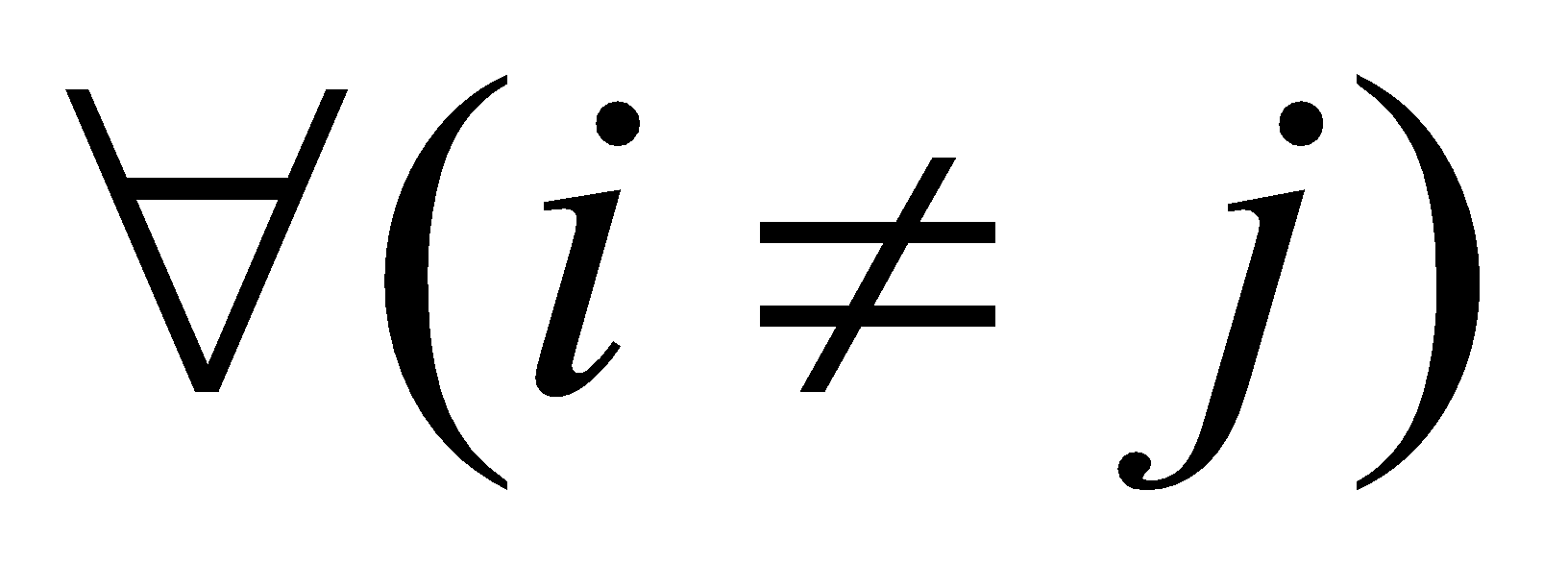
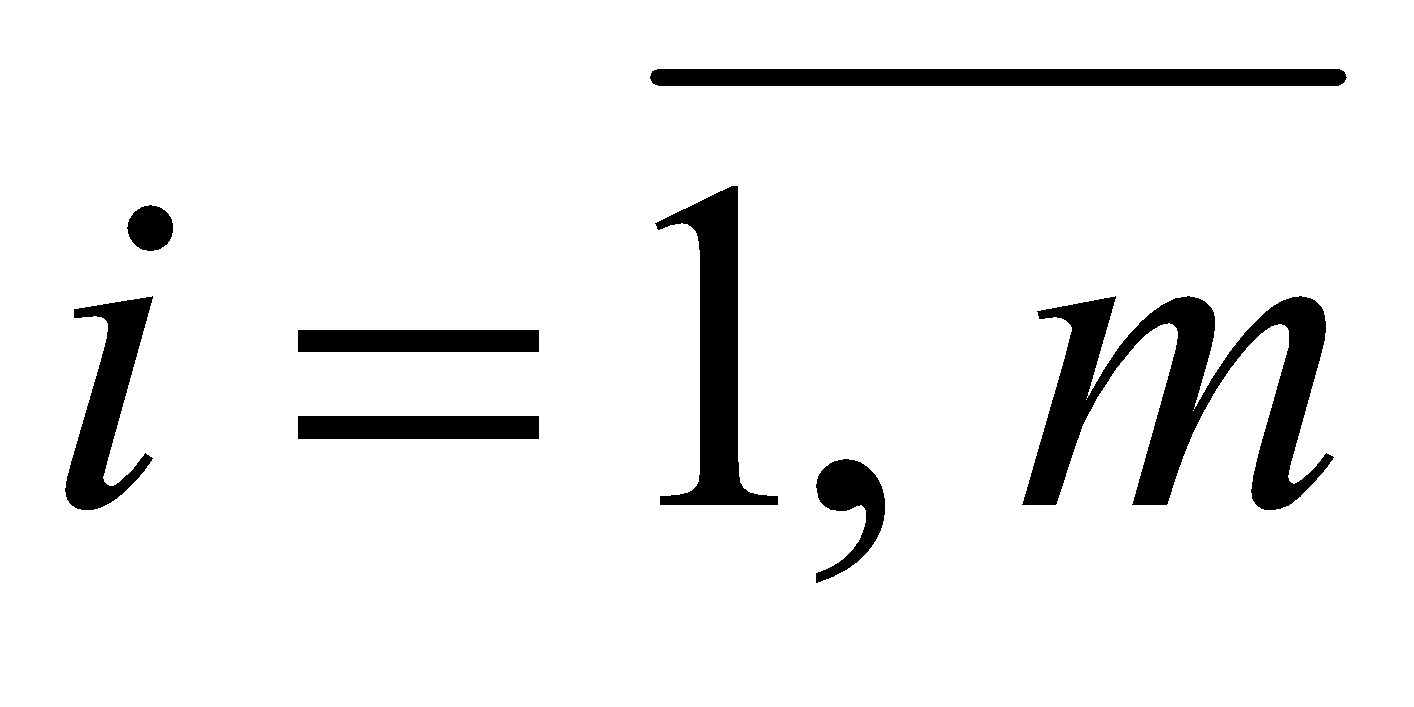
33:

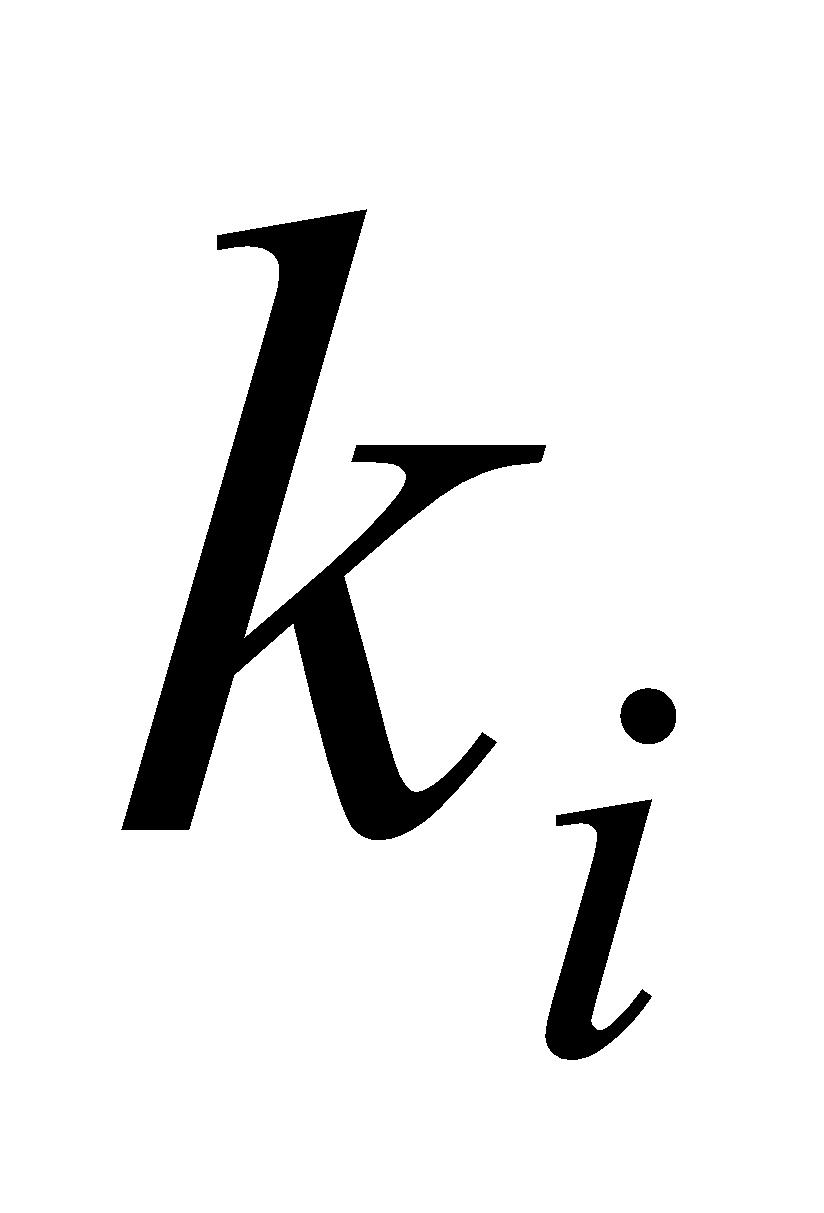
**Решение задачи об оптимальной загрузке судна на основе генератора сочетаний**

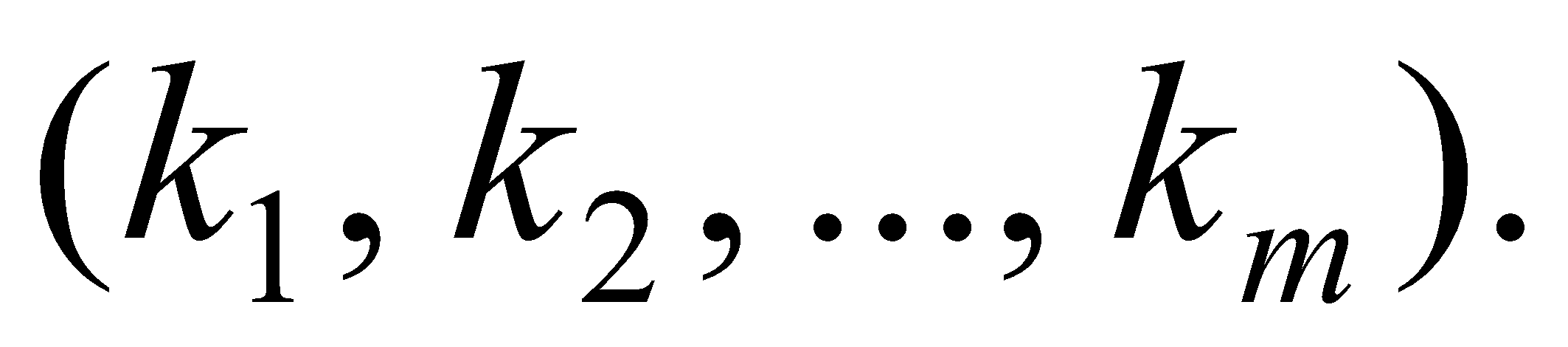
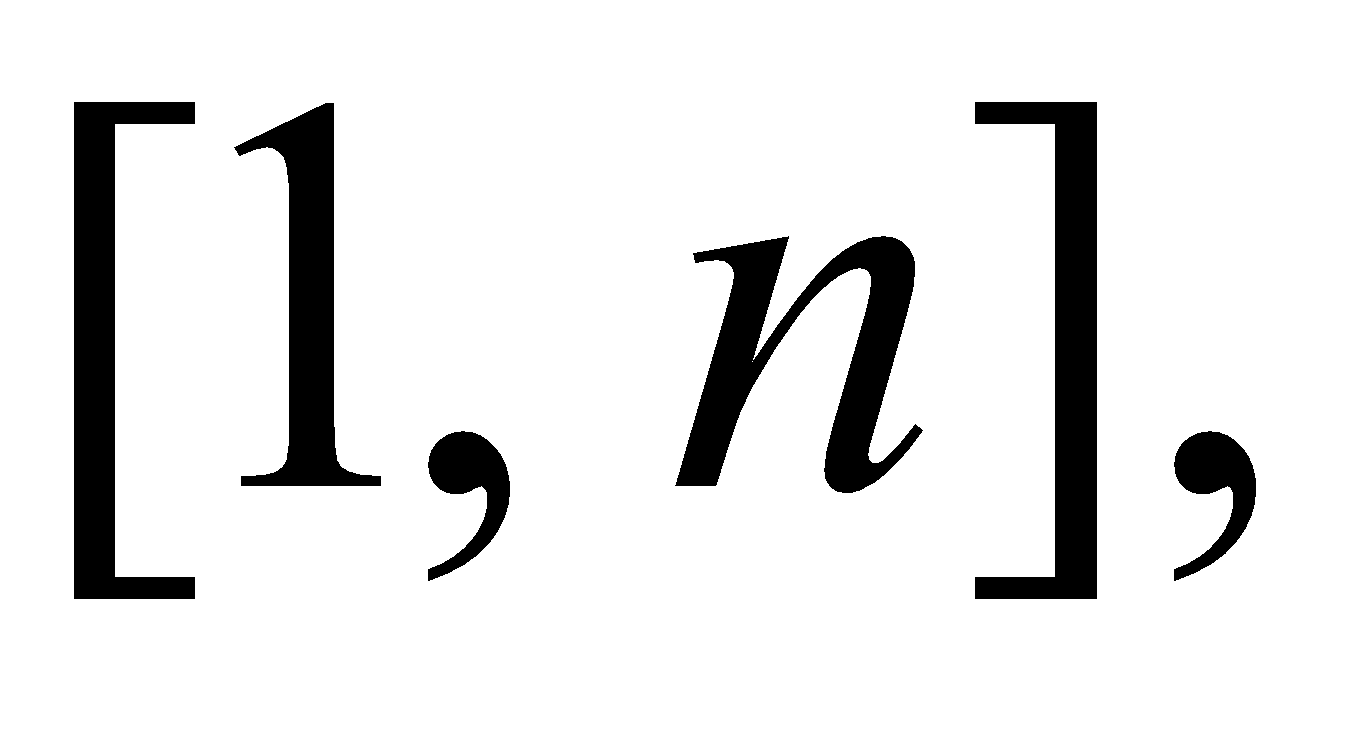
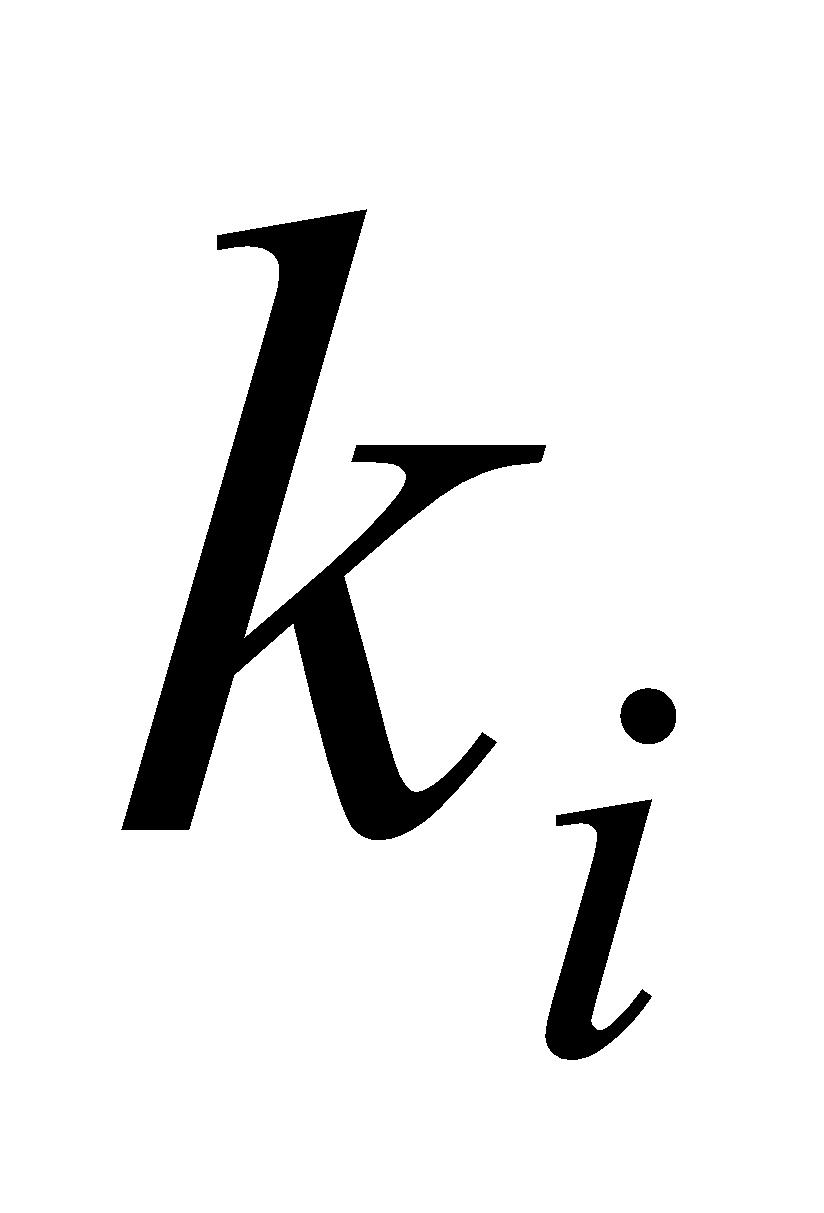
Применение генератора сочетаний продемонстрируем на решении задачи об оптимальной загрузке судна. Сформулируем условие задачи.

На палубе судна имеется  мест для размещения стандартных контейнеров. Выбрать *n* контейнеров для погрузки на судно можно из  имеющихся в наличии. Каждый контейнер  характеризуется весом  и доходом   от его перевозки. Необходимо выбрать  контейнеров таким образом, чтобы их общий вес не превышал  но при этом доход от перевозки был максимально возможным.

Математическая модель задачи может быть записана следующим образом:

    ,

где  – неизвестные (номера выбранных контейнеров), которые требуется найти.

Решением задачи будет вектор  Каждый элемент этого вектора может принимать целое значение из отрезка  и при этом все значения  должны быть разными.

34:

Схема решения задачи с применением генератора подмножеств. Задача имеет следующие исходные данные:

 – ограничение по общему весу контейнеров;

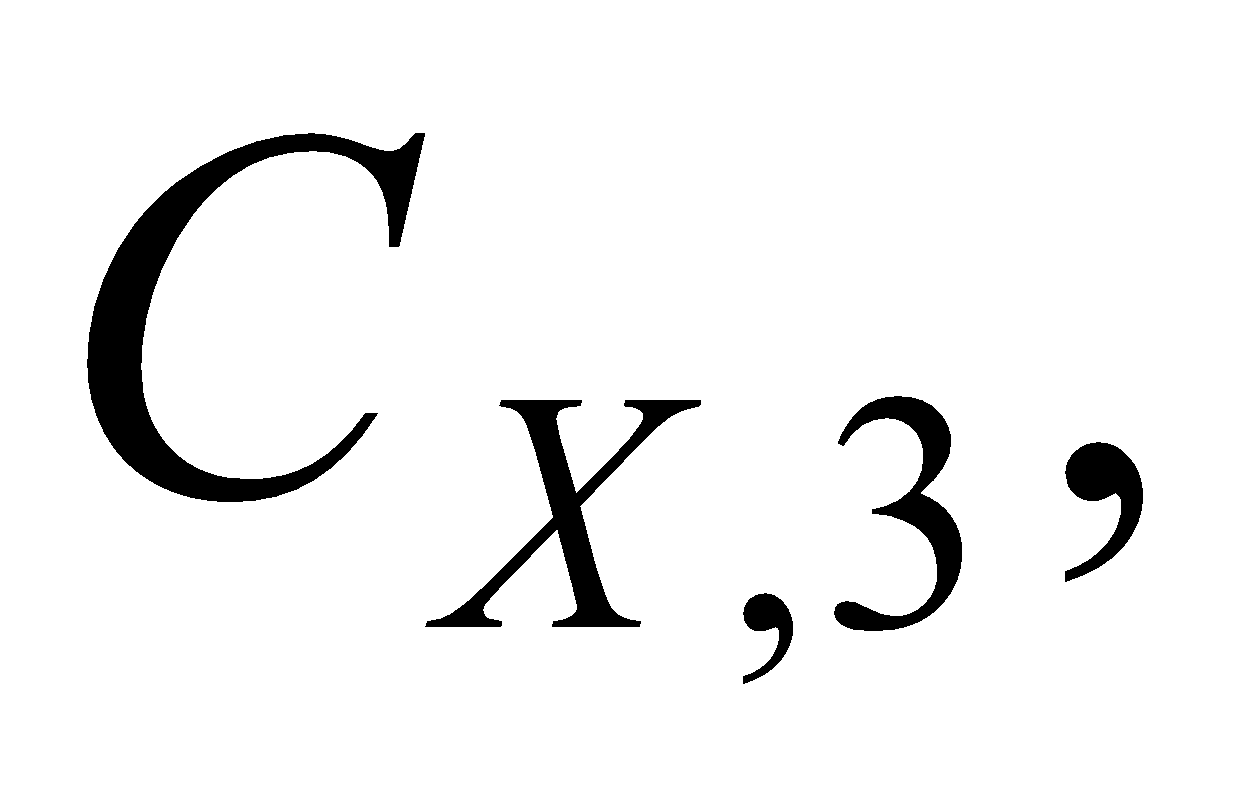
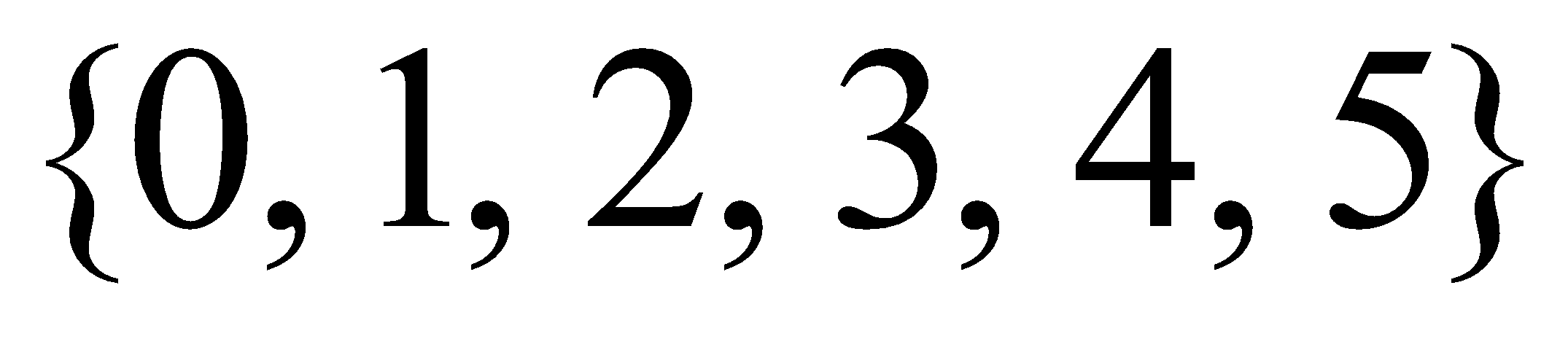
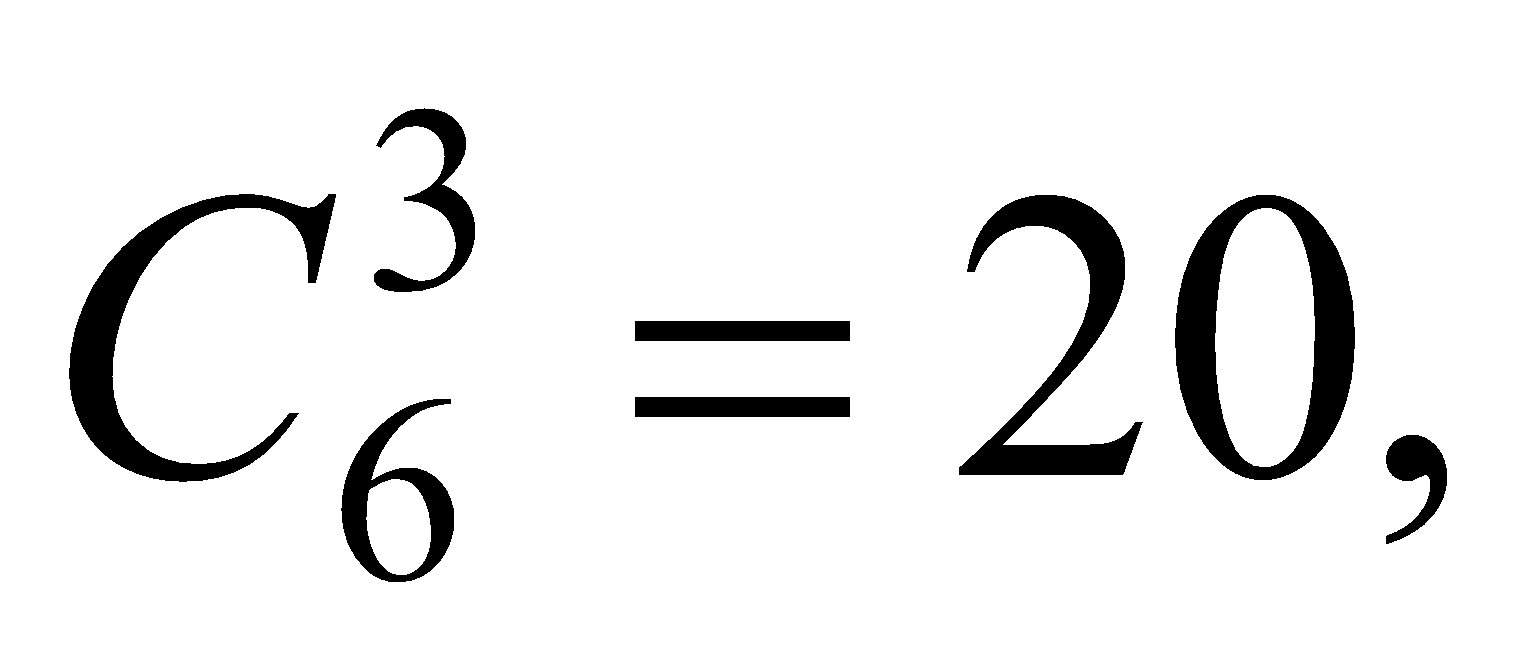
 – количество контейнеров;

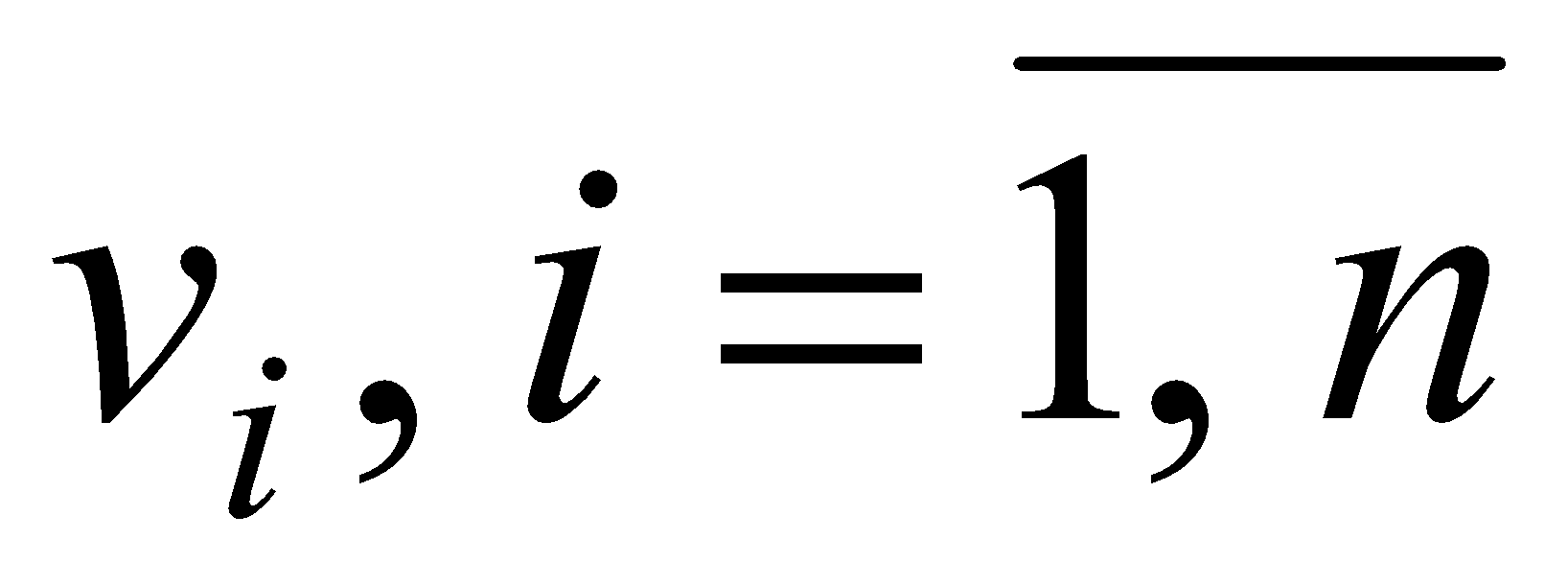
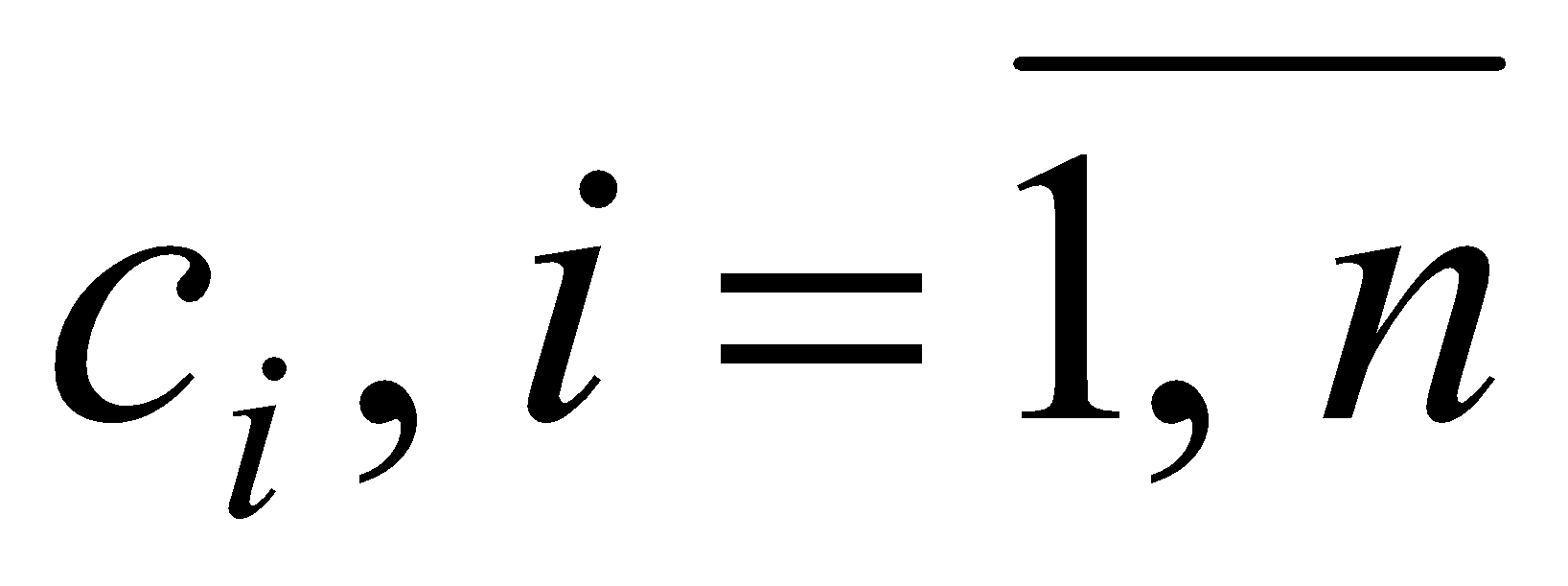
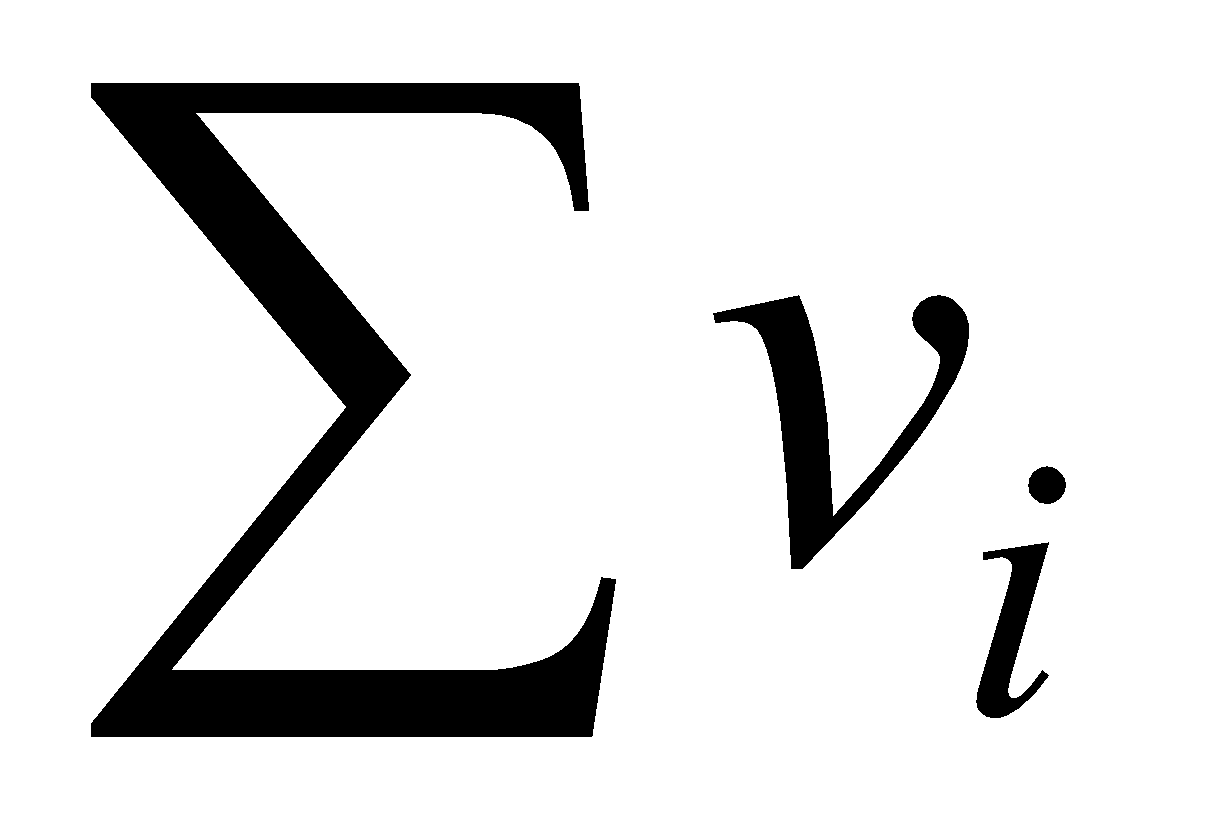
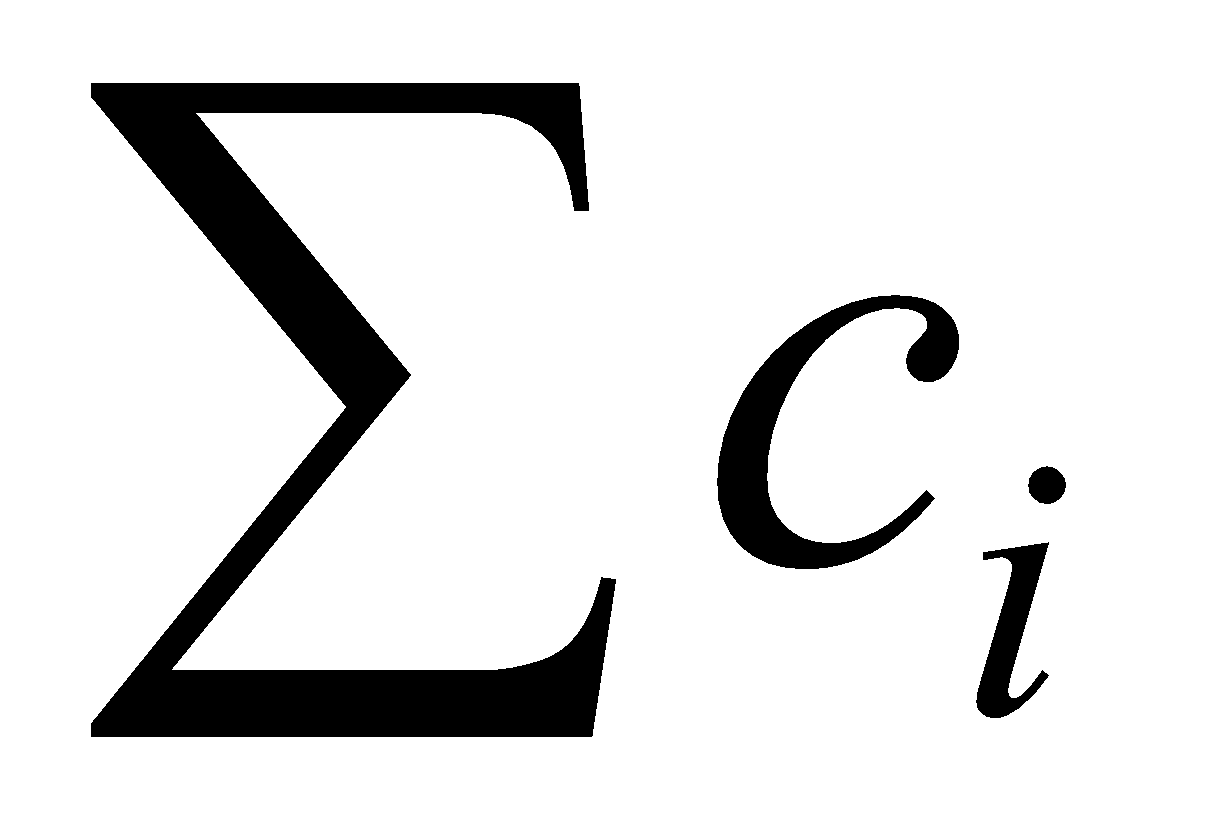
 – количество свободных мест на палубе;

 – вес контейнеров;

 – доход от перевозки контейнеров.

**** Рис. 19. Схема решения задачи об оптимальной загрузке судна

Строки таблицы, озаглавленной на рис. 19 символом  представляют собой все сочетания по три из множества . Эти сочетания могут быть получены с помощью соответствующего генератора. Несложно убедиться, что количество строк составляет  а порядок их перечисления соответствует порядку генерации сочетаний, рассмотренным выше алгоритмом.

Используя элементы сгенерированных сочетаний в качестве индексов для массивов  (вес каждого контейнера) и  (доход от перевозки), осуществляется выбор соответствующих значений (вторая таблица слева на рис. 19) , что позволяет рассчитать вес (столбец ) и доход от перевозки (столбец ) комбинации контейнеров. Решением задачи будет сочетание контейнеров, имеющее максимальный суммарный доход при допустимом суммарном весе. На рис. 19 строка, соответствующая решению, отмечена рамкой.

35:

пример реализации на языке С++ функции **boat**,решающей задачу об оптимальной загрузке судна.

// --- Вoat.h

// -- решение задачи об оптимальной загрузке судна

// функция возвращает доход от перевози выбранных контейнеров

**#pragma once**

**#include "Combi.h"**

**int boat(**

**int V,** // [in] максимальный вес груза

**short m,** // [in] количество мест для контейнеров

**short n,** // [in] всего контейнеров

**const int v[],** // [in] вес каждого контейнера

**const int c[],** // [in] доход от перевозки каждого контейнера

**short r[]** // [out] результат: индексы выбранных контейнеров

**);**

Рис. 20. Функция **boat**, решающая задачу об оптимальной загрузке судна

Функция **boat** имеет пять входных параметров, определяющих условие задачи: **V** (максимальный допустимый суммарный вес контейнеров), **m** (количество мест на палубе для установки контейнеров), **n** (общее количество контейнеров), **v** (массив размерностью **n**, содержащий вес каждого контейнера), **c** (массив размерностью **n**, содержащий доход от перевозки каждого контейнера), а также один возвращаемый параметр **r** (массив размерностью **m**, содержащий номера выбранных контейнеров). В том случае, если решение существует, функция **boat** возвращает положительное значение, иначе – нуль.

36-37:

В процессе своей работы функция **boat** использует генератор сочетаний (**combi::xcombination**) и вызывает три вспомогательные функции: **boatfnc::calcv** (расчет веса текущего сочетания контейнеров), **boatfnc::calcс** (расчет дохода от транспортировки текущего сочетания контейнеров) и **boatfnc::copycomb** (копирование текущей компинации).

// --- Вoat.cpp

**#include "stdafx.h"**

**#include "Boat.h"**

**namespace boatfnc**

**{**

**int calcv(combi::xcombination s, const int v[])** // вес

**{**

**int rc = 0;**

**for (int i = 0; i < s.m; i++) rc += v[s.ntx(i)];**

**return rc;**

**};**

**int calcc(combi::xcombination s, const int c[])** // доход

**{**

**int rc = 0;**

**for (int i = 0; i < s.m; i++) rc += c[s.ntx(i)];**

**return rc;**

**};**

**void copycomb(short m, short \*r1, const short \*r2)** // копировать

**{ for (int i = 0; i < m; i++) r1[i] = r2[i];**

**};**

**}**

**int boat(**

**int V,** // [in] максимальный вес груза

**short m,** // [in] количество мест для контейнеров

**short n,** // [in] всего контейнеров

**const int v[],** // [in] вес каждого контейнера

**const int c[],** // [in] доход от перевозки каждого контейнера

**short r[]** // [out] результат: индексы выбранных контейнеров

**)**

**{**

**combi::xcombination xc(n, m);**

**int rc = 0, i = xc.getfirst(), cc = 0;**

**while (i > 0)**

**{**

**if (boatfnc::calcv(xc,v)<= V)**

**if ((cc = boatfnc::calcc(xc,c)) > rc)**

**{rc = cc; boatfnc::copycomb(m, r, xc.sset);}**

**i = xc.getnext();**

**};**

**return rc;**

**};**

Рис. 21. Реализация функции **boat**

Функция **boat** последовательно генерирует все возможные сочетания по **m** контейнеров, вычисляет для каждого сочетания суммарный вес(функция **boatfnc::calcv**), для сочетаний с весом, не превышающим допустимое значение **V**,вычисляетдоход от перевозки этих контейнеров (**boatfnc::calcс**), фиксирует оптимальную комбинацию контейнеров (**boatfnc::copycomb**)и возвращает оптимальную доходность или нуль, если решения нет.

38-40:

Пример вызова функции **boat** для решения задачи.

// --- Main

**#include "stdafx.h"**

**#include <iostream>**

**#include <iomanip>**

**#include "Boat.h"**

**#define NN (sizeof(v)/sizeof(int))**

**#define MM 3**

**int \_tmain(int argc, \_TCHAR\* argv[])**

**{**

**setlocale(LC\_ALL, "rus");**

**int V = 1000,**

**v[] = {100, 200, 300, 400, 500, 150},**

**c[NN] = { 10, 15, 20, 25, 30, 25};**

**short r[MM];**

**int cc = boat(**

**V,** // [in] максимальный вес груза

**MM,** // [in] количество мест для контейнеров

**NN,** // [in] всего контейнеров

**v,** // [in] вес каждого контейнера

**c,** // [in] доход от перевозки каждого контейнера

**r** // [out] результат: индексы выбранных контейнеров

**);**

**std::cout<<std::endl<<"- Задача о размещении контейнеров на судне";**

**std::cout<<std::endl<<"- общее количество контейнеров : "<< NN;**

**std::cout<<std::endl<<"- количество мест для контейнеров : "<< MM;**

**std::cout<<std::endl<<"- ограничение по суммарному весу : "<< V;**

**std::cout<<std::endl<<"- вес контейнеров : ";**

**for(int i = 0; i < NN; i++) std::cout<<std::setw(3)<<v[i]<<" ";**

**std::cout<<std::endl<<"- доход от перевозки : ";**

**for(int i = 0; i < NN; i++) std::cout<<std::setw(3)<<c[i]<<" ";**

**std::cout<<std::endl<<"- выбраны контейнеры (0,1,...,m-1): ";**

**for(int i = 0; i < MM; i++) std::cout<<r[i]<<" ";**

**std::cout<<std::endl<<"- доход от перевозки : " << cc;**

**std::cout<<std::endl<<"- общий вес выбранных контейнеров : ";**

**int s = 0; for(int i = 0; i < MM; i++) s+= v[r[i]]; std::cout<<s;**

**std::cout<<std::endl<<std::endl;**

**system("pause");**

**return 0;**

**}**

Рис. 22. Пример решения задачи об оптимальной загрузке судна

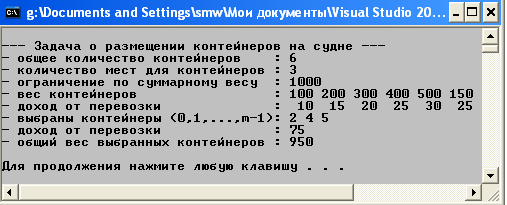


Рис. 23. Результат выполнения программы, представленной на рис. 3.7

41-42:

Представлена программа, с помощью которой можно оценить продолжительность решения задачи о загрузке судна при разном количестве контейнеров. В программе фиксируется значение параметра **m** (количество мест для контейнеров) и вычисляется продолжительность работы функции boat в зависимости от параметра n (общее количество контейнеров).

// --- Main

#include "stdafx.h"

#include <iostream>

#include <iomanip>

#include "Boat.h"

#include <time.h>

#define NN (sizeof(v)/sizeof(int))

#define MM 6

#define SPACE(n) std::setw(n)<<" "

int \_tmain(int argc, \_TCHAR\* argv[])

{

setlocale(LC\_ALL, "rus");

int V = 1000,

v[] = {250, 560, 670, 400, 200, 270, 370, 330, 330, 440, 530, 120,

200, 270, 370, 330, 330, 440, 700, 120, 550, 540, 420, 170,

600, 700, 120, 550, 540, 420, 430, 140, 300, 370, 310, 120} ;

int c[NN]={15,26, 27, 43, 16, 26, 42, 22, 34, 12, 33, 30,

42,22, 34, 43, 16, 26, 14, 12, 25, 41, 17, 28,

12,45, 60, 41, 33, 11, 14, 12, 25, 41, 30, 40};

short r[MM];

int maxcc = 0;

clock\_t t1, t2;

std::cout<<std::endl<<"-- Задача об оптимальной загрузке судна -- ";

std::cout<<std::endl<<"- ограничение по весу : "<< V;

std::cout<<std::endl<<"- количество мест : "<< MM;

std::cout<<std::endl<<"-- количество ------ продолжительность -- ";

std::cout<<std::endl<<" контейнеров вычисления ";

for (int i = 24; i <= NN; i++)

{

t1 = clock();

int maxcc = boat(V, MM, i, v, c, r);

t2 = clock();

std::cout<<std::endl<<SPACE(7)<<std::setw(2)<<i

<<SPACE(15)<<std::setw(5)<<(t2-t1);

}

std::cout<<std::endl<<std::endl;

system("pause");

return 0;

}

Рис. 24. Вычисление продолжительности решения задачи о загрузке судна при разном количестве контейнеров

Как и в задаче о рюкзаке, для вычисления продолжительности выполнения функции **boat** в программе, применяется стандартная функция **clock**.

43-44:

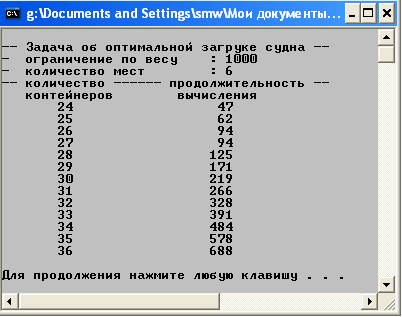
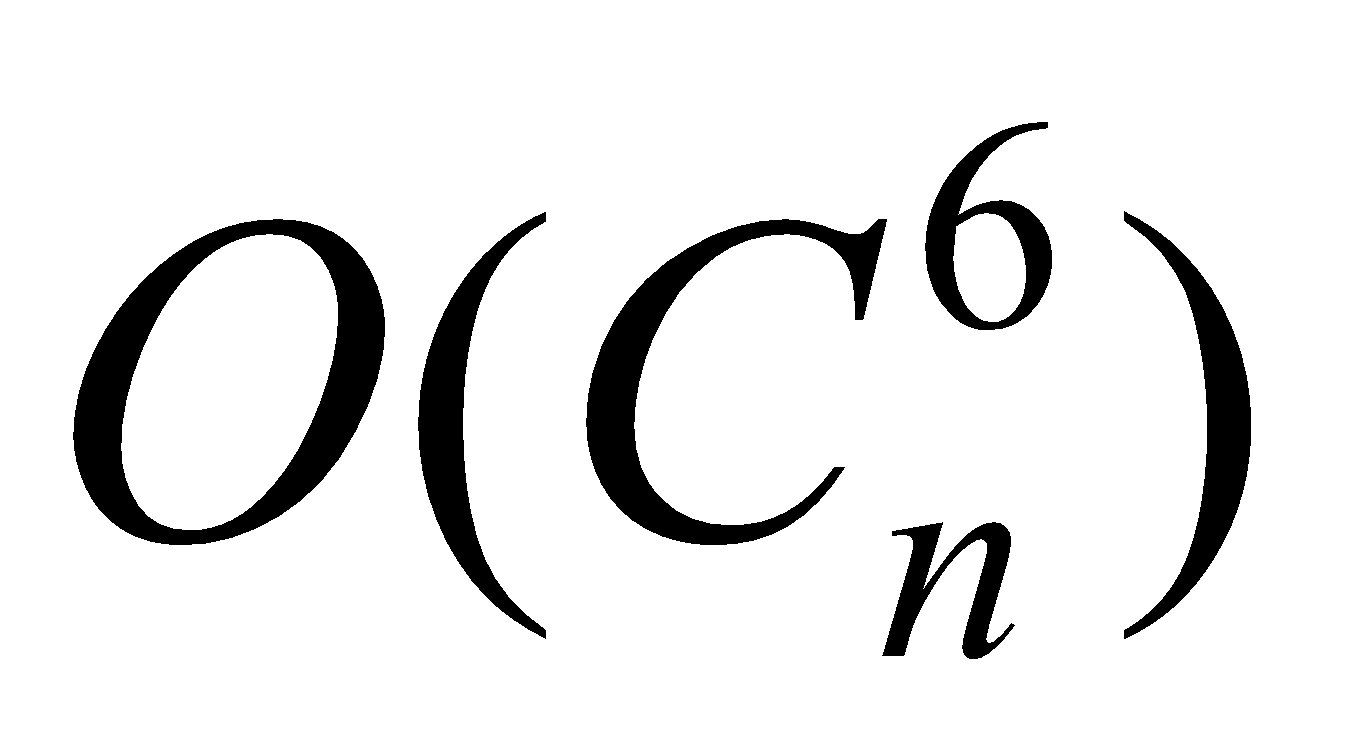
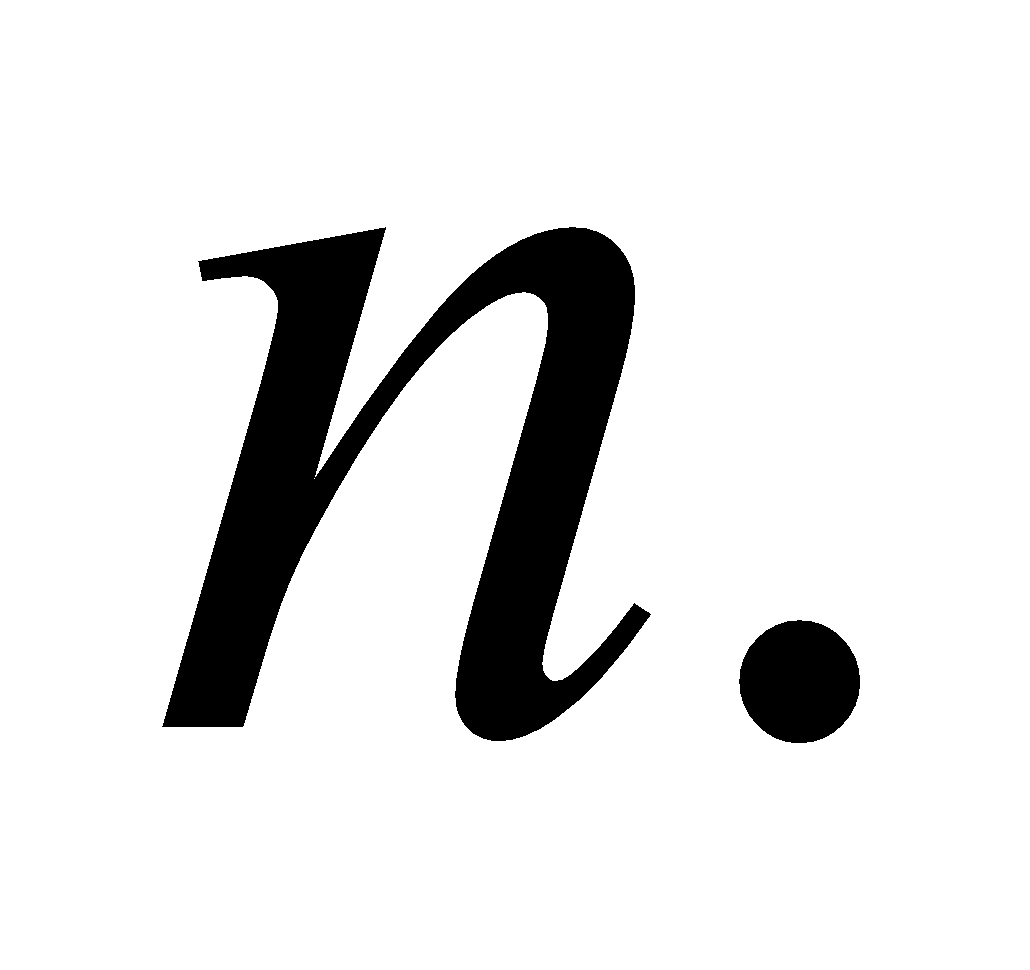
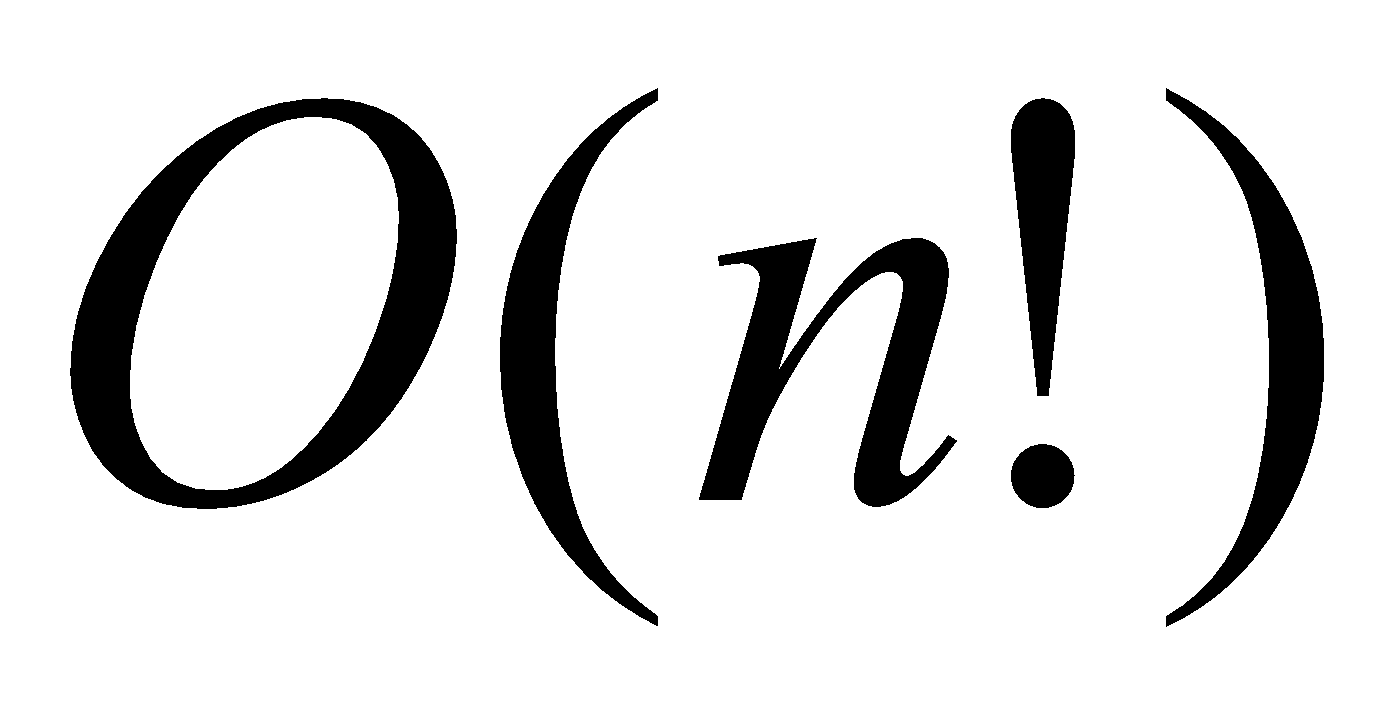


Рис. 25. Результат работы программы, представленной на рис. 24



Рис. 26. Зависимость продолжительности выполнения функции **boat** от значения параметра n

Вид графика на рис. 26 согласуется с оценкой  сложности алгоритма генерации сочетаний по 6 элементов из множества мощности 

**Сложность: в общем случае **