МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ
УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«БЕЛГОРОДСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ им. В. Г. Шухова»
(БГТУ им. В. Г. Шухова)

Кафедра программного обеспечения вычислительной
техники и автоматизированных систем

Лабораторная работа №2.1
по дисциплине: «Дискретная математика»
по теме: «**Алгоритмы порождения комбинаторных объектов**»

Выполнил/а: ст. группы ПВ-231
Чупахина София Александровна
Проверили:
Островский Алексей Мичеславович
Рязанов Юрий Дмитриевич

Белгород, 2024

Цель работы: изучить основные комбинаторные объекты, алгоритмы их порождения, программно реализовать и оценить временную сложность алгоритмов.

Задание 1.

Задание: реализовать алгоритм порождения подмножеств.

В дальнейшем для выполнения заданий этой работы нам понадобится неоднократно писать программный код, совершающий операции над множествами. За основу, как и в предыдущих лабораторных работах, возьмем библиотеку ordered_array_set с заголовочным файлом **ordered_array_set.h:**

```
#ifndef INC_ORDERED_ARRAY_SET_H
#define INC_ORDERED_ARRAY_SET_H

#include <stdint.h>
#include <assert.h>
#include <memory.h>
#include <malloc.h>
#include <stdio.h>
#include <stdbool.h>

typedef struct ordered_array_set {
    int *data; //элементы множества
    size_t size; //количество элементов в множестве
    size_t capacity; //максимальное количество элементов в
множестве
} ordered_array_set;

//возвращает пустое множество, в которое можно вставить capacity
элементов
ordered_array_set ordered_array_set_create (size_t capacity);

//возвращает множество, состоящее из элементов массива a размера
size
ordered_array_set ordered_array_set_create_from_array (const int
*a, size_t size);

//Уменьшает емкость capacity множества по адресу a до его размера
size
//При этом перераспределяет память, отведенную под массив value,
чтобы он занимал минимальное ее количество
static void ordered_array_set_shrinkToFit (ordered_array_set *a);

//Увеличивает емкость capacity множества по адресу a на slots
единиц
static void ordered_array_set_increaseCapacity (ordered_array_set
*a, size_t slots);

//возвращает значение позицию элемента в множестве,
```

```
//если значение value имеется в множестве set,
//иначе - n
size_t ordered_array_set_in (ordered_array_set *set, int value);

//возвращает значение 'истина', если элементы множеств set1 и set2
равны
//иначе - 'ложь'
bool isEqual (ordered_array_set set1, ordered_array_set set2);

//возвращает значение 'истина', если subset является подмножеством
set
//иначе - 'ложь'
bool isSubset (ordered_array_set subset, ordered_array_set set);

//возбуждает исключение, если в множество по адресу set
//нельзя вставить элемент
void ordered_array_set_isAbleAppend (ordered_array_set *set);

//добавляет элемент value в множество set
void ordered_array_set_insert (ordered_array_set *set, int value);

//удаляет элемент value из множества set
void ordered_array_set_deleteElement (ordered_array_set *set, int
value);

//возвращает объединение множеств set1 и set2
ordered_array_set join (ordered_array_set set1, ordered_array_set
set2);

//возвращает пересечение множеств set1 и set2
ordered_array_set intersection (ordered_array_set set1,
ordered_array_set set2);

//возвращает разность множеств set1 и set2
ordered_array_set difference (ordered_array_set set1,
ordered_array_set set2);

//возвращает симметрическую разность множеств set1 и set2
ordered_array_set symmetricDifference (ordered_array_set set1,
ordered_array_set set2);

//возвращает дополнение до универсума universumSet множества set
ordered_array_set complement (ordered_array_set set,
ordered_array_set universumSet);
```

```
//вывод множества set
void ordered_array_set_print (ordered_array_set set);

//освобождает память, занимаемую множеством set
void ordered_array_set_delete (ordered_array_set *set);
#endif
```

...И файлом реализации **ordered_array_set.c:**

```
#ifndef INC_ORDERED_ARRAY_SET_C
#define INC_ORDERED_ARRAY_SET_C

#include "ordered_array_set.h"
#include "C:\Users\sovac\Desktop\ОП, преимущественно
лабы\second_semester\libs\algorithms\array\array.c"
#include "C:\Users\sovac\Desktop\ОП, преимущественно
лабы\second_semester\libs\algorithms\math_basics\math_basics.c"

ordered_array_set ordered_array_set_create (size_t capacity) {
   return (ordered_array_set) {
       malloc(sizeof(int) * capacity),
       0,
       capacity
   };
}

ordered_array_set ordered_array_set_create_from_array (const int
*a, size_t size) {
    ordered_array_set result = ordered_array_set_create(size);
    for (size_t i = 0; i < size; i++) {
        ordered_array_set_insert(&result, a[i]);
    }
    qsort(result.data, size, sizeof(int), compare_ints);
    ordered_array_set_shrinkToFit(&result);
    return result;
}

static void ordered_array_set_shrinkToFit (ordered_array_set *a) {
    if (a->size != a->capacity) {
        a->data = (int *) realloc(a->data, sizeof(int) * a->size);
        a->capacity = a->size;
    }
}

static void ordered_array_set_increaseCapacity (ordered_array_set
*a, size_t slots) {
    a->data = (int *) realloc(a->data, sizeof(int) * (a->size +
```

```
slots));
    a->capacity += slots;
}

size_t ordered_array_set_in (ordered_array_set *set, int value) {
    size_t index = binarySearch_(set->data, set->size,value);
    return (index != SIZE_MAX) ? index : set->size;
}

bool isEqual (ordered_array_set set1, ordered_array_set set2) {
    if (set1.size == set2.size) {
        return memcmp(set1.data, set2.data, sizeof(int) *
set1.size) == 0;
    } else {
        return 0;
    }
}

bool isSubset (ordered_array_set subset, ordered_array_set set) {
    bool is_subset = 1;
    for (size_t i = 0; i < subset.size; i++) {
        if (ordered_array_set_in(&set, subset.data[i]) ==
set.size) {
            is_subset = 0;
            break;
        }
    }
    return is_subset;
}

void ordered_array_set_isAbleAppend (ordered_array_set *set) {
    assert(set->size < set->capacity);
}

void ordered_array_set_insert (ordered_array_set *set, int value)
{
    ordered_array_set_isAbleAppend(set);
    if (set->size == 0 || value > set->data[(set->size)-1]) {
        append_(set->data, &(set->size), value);
    } else if (ordered_array_set_in(set, value) == set->size) {
        size_t start_index = binarySearchMoreOrEqual_(set->data,
set->size, value);
        insert_(set->data, &(set->size), start_index, value);
    }
}
```

```
void ordered_array_set_deleteElement (ordered_array_set *set, int
value) {
    if (ordered_array_set_in(set, value) != set-> size) {
        deleteByPosSaveOrder_(set->data, &(set->size),
ordered_array_set_in(set, value));
    }
}

ordered_array_set join (ordered_array_set set1, ordered_array_set
set2) {
    ordered_array_set result =
ordered_array_set_create(set1.capacity + set2.capacity);
    size_t index1 = 0;
    size_t index2 = 0;
    while (index1 < set1.size && index2 < set2.size) {
        if (set1.data[index1] == set2.data[index2]) {
            ordered_array_set_insert(&result, set1.data[index1]);
            index1++;
            index2++;
        } else if (set1.data[index1] < set2.data[index2]) {
            ordered_array_set_insert(&result, set1.data[index1]);
            index1++;
        } else {
            ordered_array_set_insert(&result, set2.data[index2]);
            index2++;
        }
    }
    while (index1 < set1.size) {
        ordered_array_set_insert(&result, set1.data[index1]);
        index1++;
    }
    while (index2 < set2.size) {
        ordered_array_set_insert(&result, set2.data[index2]);
        index2++;
    }
    ordered_array_set_shrinkToFit(&result);
    return result;
}

ordered_array_set intersection (ordered_array_set set1,
ordered_array_set set2) {
    ordered_array_set result =
ordered_array_set_create(set1.capacity);
    size_t index1 = 0;
```

```
    size_t index2 = 0;
    while (index1 < set1.size && index2 < set2.size) {
        if (set1.data[index1] == set2.data[index2]) {
            ordered_array_set_insert(&result, set1.data[index1]);
            index1++;
            index2++;
        } else if (set1.data[index1] < set2.data[index2]) {
            index1++;
        } else {
            index2++;
        }
    }
    ordered_array_set_shrinkToFit(&result);
    return result;
}

ordered_array_set difference (ordered_array_set set1,
ordered_array_set set2) {
    ordered_array_set result =
ordered_array_set_create(set1.capacity);
    size_t index1 = 0;
    size_t index2 = 0;
    while (index1 < set1.size && index2 < set2.size) {
        if (set1.data[index1] == set2.data[index2]) {
            index1++;
            index2++;
        } else if (set1.data[index1] < set2.data[index2]) {
            ordered_array_set_insert(&result, set1.data[index1]);
            index1++;
        } else {
            index2++;
        }
    }
    while (index1 < set1.size) {
        ordered_array_set_insert(&result, set1.data[index1]);
        index1++;
    }
    ordered_array_set_shrinkToFit(&result);
    return result;
}

ordered_array_set symmetricDifference (ordered_array_set set1,
ordered_array_set set2) {
    ordered_array_set result =
ordered_array_set_create(set1.capacity + set2.capacity);
```

```
    size_t index1 = 0;
    size_t index2 = 0;
    while (index1 < set1.size && index2 < set2.size) {
        if (set1.data[index1] == set2.data[index2]) {
            index1++;
            index2++;
        } else if (set1.data[index1] < set2.data[index2]) {
            ordered_array_set_insert(&result, set1.data[index1]);
            index1++;
        } else {
            ordered_array_set_insert(&result, set2.data[index2]);
            index2++;
        }
    }
    while (index1 < set1.size) {
        ordered_array_set_insert(&result, set1.data[index1]);
        index1++;
    }
    while (index2 < set2.size) {
        ordered_array_set_insert(&result, set2.data[index2]);
        index2++;
    }
    ordered_array_set_shrinkToFit(&result);
    return result;
}

ordered_array_set complement (ordered_array_set set,
ordered_array_set universumSet) {
    return difference(universumSet, set);
}

void ordered_array_set_print (ordered_array_set set) {
    if (set.size == 0) {
        printf("Empty set\n");
    } else {
        printf("{");
        for (int i = 0; i < set.size; i++) {
            printf("%d ", set.data[i]);
        }
        printf("\b}\n");
    }
}

void ordered_array_set_delete (ordered_array_set *set) {
    free(set->data);
```

```
    set->size = 0;
    set->capacity = 0;
}

#endif
```

Далее поступим следующим образом. Каждому подмножеству некоторого множества M можно поставить в соответствие |M|-разрядный двоичный вектор (или двоичное число), где i-тый его разряд будет равен единице, если i-тый элемент множества входит в подмножество, и нулю, если i-тый элемент множества не входит в подмножество. Задача порождения всех подмножеств сводится к задаче порождения всех двоичных векторов и генерации подмножеств, соответствующих каждому из них. Для того, чтобы получить подмножество, соответствующее некоторому двоичному вектору, зададим вне функции main() следующую функцию:

```
//Возвращает подмножество множества set, заданное двоичным
вектором vector
ordered_array_set create_subset_by_binary_vector(ordered_array_set
set, int vector) {
    ordered_array_set result =
ordered_array_set_create(set.capacity);
    for (int i = 0; i < set.capacity; i++) {
        if (vector >> i & 1) {
            ordered_array_set_insert(&result, set.data[i]);
        }
    }
    ordered_array_set_shrinkToFit(&result);
    return result;
}
```

Непосредственно для генерации объявим следующие функции, в основе которых лежит алгоритм поиска с возвращением (более подробное объяснение каждого этапа дано в комментариях):

```
//Выводит на экран все подмножества множества set с помощью
алгоритма поиска с возвращением
//Рекурсивная функция без обертки; на каждой итерации она
опирается на текущее значение формируемого двоичного вектора
cur_vector
//и номер обрабатываемого разряда (слева направо) cur_digit
void print_all_subsets_(int cur_vector, int cur_digit,
ordered_array_set set) {
    //В цикле for перебираются возможные значения для разряда
двоичного вектора
    for (int i = 0; i <= 1; i++) {
        //В конец вектора добавляется новый разряд, обращенный в 0
или 1 в зависимости от итерации цикла for
```

```
        int plus_cur_vector = (cur_vector << 1) + i;
        //Если полученный вектор - решение, на его основе
генерируется подмножество, выводимое на экран
        if (cur_digit == set.size) {
            ordered_array_set cur_subset =
create_subset_by_binary_vector(set, plus_cur_vector);
            ordered_array_set_print(cur_subset);
        //Если полученный вектор - не решение, этой же функцией
обрабатывается его следующий разряд
        } else {
            print_all_subsets_(plus_cur_vector, cur_digit+1, set);
        }
    }
}
```

```
//Выводит на экран все подмножества множества set с помощью
алгоритма поиска с возвращением
//Обертка функции print_all_subsets_, начинающая работу с пустым
стартовым вектором cur_vector и номером обрабатываемого разряда 1
//Также здесь обрабатывается случай, когда длина исходного
множества равна 0
void print_all_subsets(ordered_array_set set) {
    if (set.size == 0) {
        ordered_array_set_print(ordered_array_set_create(0));
    }
    else {
        print_all_subsets_(0, 1, set);
    }
}
```

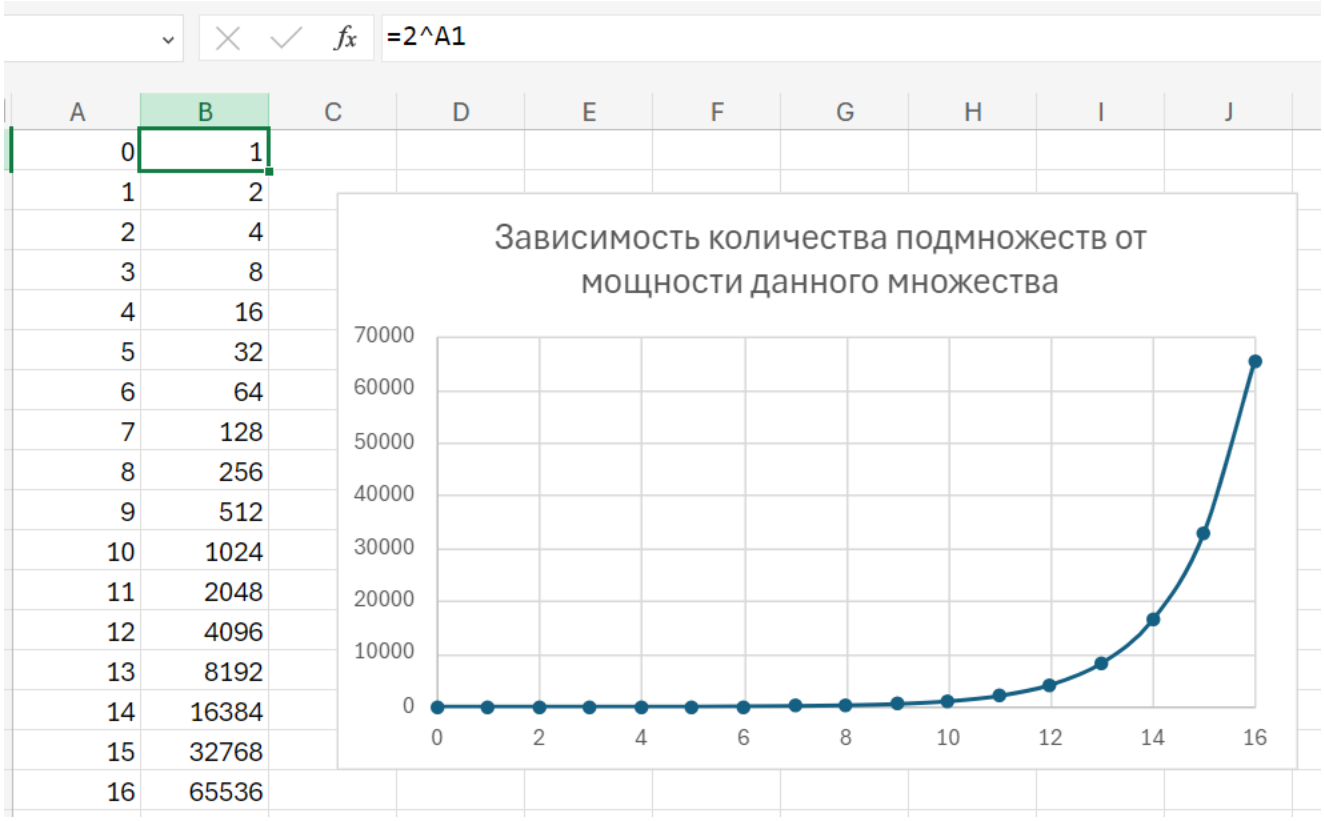
Задание 2.

Задание: построить график зависимости количества всех подмножеств от мощности множества.

Для пустого множества вариант подмножества будет единственным: пустое множество. Для остальных случаев справедливы следующие рассуждения. Если некое множество обладает мощностью |M|, то для каждого из |M| элементов можно выбрать один из двух вариантов: элемент либо присутствует, либо не присутствует в подмножестве. Выбор одного из двух вариантов для каждого элемента множества M – отдельное действие, и действия выбора для каждого отдельного элемента не зависят друг от друга. Следовательно, можно применить правило произведения. Для составления подмножеств из элементов множества с мощностью |M| нужно совершить |M| действий, и количество вариантов, которым можно совершить эту цепочку действий, будет равно $2*2*...*2 = 2^{|M|}$.

Значит, график зависимости количества всех подмножеств от мощности множества будет эквивалентен графику функции $y = 2^x$, экспоненте. Составив

электронную таблицу со следующими опорными точками, получим график, выглядящий следующим образом:



Задание 3.

Задание: построить графики зависимости времени выполнения алгоритмов п.1 на вашей ЭВМ от мощности множества.

Для того, чтобы отслеживать время выполнения алгоритма, нам потребуется написать несколько новых функций. Для этого воспользуемся встроенной библиотекой <sys/time.h>, из которой нам понадобится структура timeval и функция gettimeofday(). Структура timeval имеет два поля типа long, tv_sec и tv_usec, хранящие значение времени в секундах и микросекундах соответственно. Функция gettimeofday() записывает в переданную ей структуру timeval время, прошедшее с 1 января 1970 года (все так же в секундах и микросекундах). Для отслеживания времени выполнения алгоритма напишем следующие функции:

```
//Выводит на экран структуру timeval time
void print_timeval(struct timeval time) {
    printf("%ld,%.6ld", time.tv_sec, time.tv_usec);
}
```

```
//Выводит на экран все подмножества множества set и сообщение о
том, сколько времени выполнялся алгоритм
void print_subsets_and_time_of_generating (ordered_array_set set)
{
    //Заранее объявляем структуры timeval для хранения времени
начала, конца и длительности процесса генерации
```

```
    struct timeval start, end, difference;
    //Задаем значение временной метки перед генерацией и после нее
    gettimeofday(&start, NULL);
    print_all_subsets(set);
    gettimeofday(&end, NULL);
    //Высчитываем разницу в секундах и микросекундах и сохраняем в
структуру difference
    difference.tv_sec = end.tv_sec - start.tv_sec - (end.tv_usec <
start.tv_usec);
    difference.tv_usec = (end.tv_usec + (end.tv_usec <
start.tv_usec) * 1000000)-start.tv_usec;

    //Вывод сообщения на экран
    printf("|M| = %llu; time of generation = ", set.size);
    print_timeval(difference);
    printf("\n");
}
```

И наконец, прописав в функции main() следующее, мы получим вывод – время, за которое генерируются все подмножества, в нашем случае, пустого множества A:

```
int main() {
    int a[0] = {0};
    ordered_array_set A = ordered_array_set_create_from_array(a,
0);
    print_subsets_and_time_of_generating(A);
}
```

```
"C:\Users\sovac\Desktop\дискретная математика\ДИСКРЕТКА ЛАБА 2.1\subset_creation.exe"
Empty set
|M| = 0; time of generation = 0.000340

Process finished with exit code 0
```

Меняя функцию main(), увеличивая длину и количество элементов множества A, и запуская программу заново, можно получить время генерации для множеств разных мощностей.

```
int main() {
    int a[1] = {1};
    ordered_array_set A = ordered_array_set_create_from_array(a,
1);
    print_subsets_and_time_of_subsets_generating(A);
}
```

```
"C:\Users\sovac\Desktop\дискретная математика\ДИСКРЕТКА ЛАБА 2.1\subset_creation.exe"
Empty set
{1}
|M| = 1; time of generation = 0.000620

Process finished with exit code 0
```

```
int main() {
    int a[2] = {1, 2};
    ordered_array_set A = ordered_array_set_create_from_array(a,
2);
    print_subsets_and_time_of_subsets_generating(A);
}
```

```
"C:\Users\sovac\Desktop\дискретная математика\ДИСКРЕТКА ЛАБА 2.1\subset_creation.exe"
Empty set
{1}
{2}
{1 2}
|M| = 2; time of generation = 0.001602

Process finished with exit code 0
```

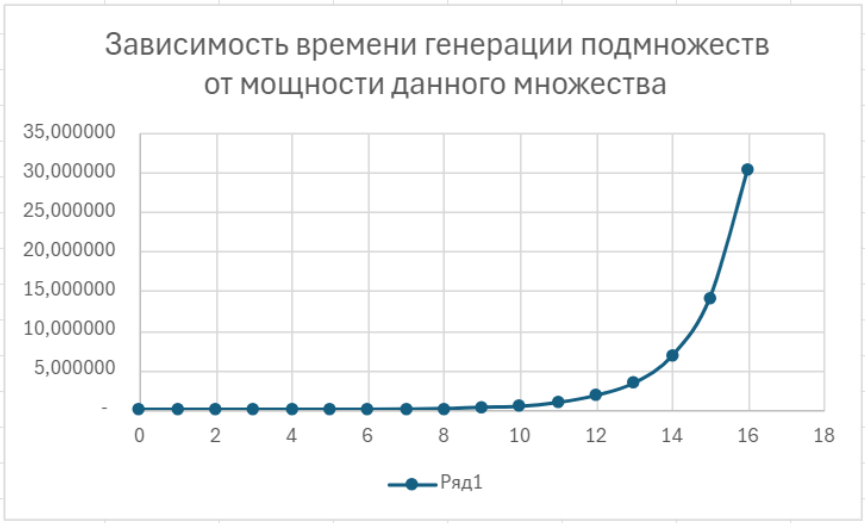
```
int main() {
    int a[3] = {1, 2, 3};
    ordered_array_set A = ordered_array_set_create_from_array(a,
3);
    print_subsets_and_time_of_subsets_generating(A);
}
```

```
"C:\Users\sovac\Desktop\дискретная математика\ДИСКРЕТКА ЛАБА 2.1\subset_creation.exe"
Empty set
{1}
{2}
{1 2}
{3}
{1 3}
{2 3}
{1 2 3}
|M| = 3; time of generation = 0.002097

Process finished with exit code 0
```

И так далее. Проделав это еще несколько раз, занесем значения в таблицу и составим график по ее данным:

L	M	N
0	0,000340	**Проверка**
1	0,000620	0,000680
2	0,001602	0,001240
3	0,002097	0,003204
4	0,004000	0,004194
5	0,009606	0,008000
6	0,018830	0,019212
7	0,045291	0,037660
8	0,096880	0,090582
9	0,269672	0,193760
10	0,385659	0,539344
11	0,873461	0,771318
12	1,761764	1,746922
13	3,331655	3,523528
14	6,743418	6,663310
15	14,100449	13,486836
16	30,499124	28,200898



Как видим, график все так же представляет собой линию, похожую на экспоненту. К тому же, в таблице имеется графа «проверка», каждое значение которой – умноженное на 2 значение генерации подмножеств при мощности, меньшей на 1. За редкими исключениями, значения времени фактической генерации не сильно отклоняются от удвоенного времени предыдущей генерации, то есть экспоненциальная зависимость сохраняется – каждое новое значение больше предыдущего вдвое.

Задание 4.

Задание: определить максимальную мощность множества, для которого можно получить все подмножества не более чем за час, сутки, месяц, год на вашей ЭВМ.

Возьмем как опорное значение времени генерации то, которое мы получили при максимальной мощности исходного множества, то есть при $|M| = 16$ (будем считать, что при большем времени генерации погрешность будет меньшей). Время генерации всех подмножеств при $|M| = 16$ будет составлять 28,200898 секунд. Тогда время генерации всех подмножеств при $|M| = 17$ будет составлять около 2*28,200898 секунд, при $|M| = 18$ – около 2^2*28,200898 секунд, при $|M| = 19$ – около 2^3*28,200898 секунд.

Можно сделать вывод, что при любой $|M| > 16$ время генерации можно вычислить как $2^{|M|-16}$*28,200898 секунд. И если мы хотим найти максимальную мощность множества, при которой генерация подмножеств происходит за время, не превышающее t, то нужно найти максимальное целое решение неравенства $2^{|M|-16}$*28,200898 <= t. Разделив обе части на 28,200898, получим $2^{|M|-16} <= t/28,200898$, а прологарифмировав их по основанию 2, получим $|M|-16 <= \log_2(t/28,200898)$ и, как следствие, $|M| <= \log_2(t/28,200898) + 16$. Для нахождения же максимального целого |M| нужно просто отбросить дробную часть полученного выражения.

Таким образом, максимальная мощность множества, подмножества которого будут генерироваться меньше чем за час, равна целой части выражения $\log_2(3600/28{,}200898) + 16$. (1 час = 60*60 = 3600 секунд). $|M| = \log_2(3600/28{,}200898) + 16 = \log_2(127{,}65550941) + 16 = 6.996111 + 16 = 22.996111$, значит, наибольшее значение $|M| = 22$. (а так как значение дробной части близится к единице, можно предположить, что для множества мощностью 23 генерация будет происходить незначительно дольше часа).

Далее, максимальная мощность множества, подмножества которого будут генерироваться меньше чем за сутки, равна целой части выражения $\log_2(86400/28{,}200898) + 16$. (1 сутки = 60*60*24 = 86400 секунд). $|M| = \log_2(86400/28{,}200898) + 16 = \log_2(3\,063{,}73222583) + 16 = 11.581074 + 16 = 27.581074$, значит, наибольшее значение $|M| = 27$.

Максимальная мощность множества, подмножества которого будут генерироваться меньше чем за месяц, равна целой части выражения $\log_2(2\,592\,000/28{,}200898) + 16$. (1 месяц = 30 дней = 60*60*24*30 = 2 592 000 секунд). $|M| = \log_2(2\,592\,000/28{,}200898) + 16 = \log_2(91911.96677) + 16 = 16.487965 + 16 = 32.487965$, значит, наибольшее значение $|M| = 32$.

Максимальная мощность множества, подмножества которого будут генерироваться меньше чем за год, равна целой части выражения $\log_2(31\,536\,000/28{,}200898) + 16$. (1 год = 60*60*24*365 = 31 536 000 секунд). $|M| = \log_2(31\,536\,000/28{,}200898) + 16 = \log_2(1118262.262) + 16 = 20.092827 + 16 = 36.092827$, значит, наибольшее значение $|M| = 36$.

Задание 5.

Задание: определить максимальную мощность множества, для которого можно получить все подмножества не более чем за час, сутки, месяц, год на ЭВМ, в 10 и в 100 раз быстрее вашей.

Для определения максимальной мощности множества на более мощных ЭВМ мы можем воспользоваться той же логикой, однако для мощностей в 10 и 100 раз больше исходной время генерации подмножеств для $|M| = 16$ будет соответственно 28,200898/10 = 2,8200898 и 28,200898/100 = 0,28200898 секунд.

Таким образом, **при мощности ЭВМ в десять раз больше,** $|M| <= \log_2(t/2{,}8200898) + 16$. Меньше часа будут генерироваться подмножества множества с мощностью $|M| <= \log_2(3600/2{,}8200898) + 16 = 26.318040$; $|M| = 26$. Меньше суток будут генерироваться подмножества множества с мощностью $|M| <= \log_2(86400/2{,}8200898) + 16 = 30.903002$; $|M| = 30$. Меньше месяца будут генерироваться подмножества множества с мощностью $|M| <= \log_2(2\,592\,000/2{,}8200898) + 16 = 35.809893$; $|M| = 35$. Меньше года будут генерироваться подмножества множества с мощностью $|M| <= \log_2(31\,536\,000/2{,}8200898) + 16 = 39.414755$; $|M| = 39$.

В свою очередь, **при мощности ЭВМ в сто раз больше,** $|M| <= \log_2(t/0{,}28200898) + 16$. Меньше часа будут генерироваться подмножества множества с мощностью $|M| <= \log_2(3600/0{,}28200898) + 16 = 29.639968$; $|M| = 29$.

Меньше суток будут генерироваться подмножества множества с мощностью |M| <= $\log_2(86400/0{,}28200898) + 16 = 34.224930$; |M| = 34. Меньше месяца будут генерироваться подмножества множества с мощностью |M| <= $\log_2(2\,592\,000/0{,}28200898) + 16 = 39.131821$; |M| = 39. Меньше года будут генерироваться подмножества множества с мощностью |M| <= $\log_2(31\,536\,000/0{,}28200898) + 16 = 42.736683$; |M| = 42.

Примечание: здесь мы опустили процесс вычисления, чтобы не слишком растягивать лабораторную работу.

Задание 6.

Задание: реализовать алгоритм порождения сочетаний.

Так же, как и в случае порождения подмножеств, для порождения сочетаний мы воспользуемся алгоритмом поиска с возвращением. Более подробное пояснение каждого шага дано в комментариях получившейся функции:

```
#include "ordered_array_set\ordered_array_set.c"
#include <assert.h>

//Выводит на экран все сочетания множества set по
combination_power с помощью алгоритма поиска с возвращением
//Рекурсивная функция без обертки; на каждой итерации она
опирается на текущее значение формируемого сочетания
cur_combination
//И значения "левой границы" подходящих на данной итерации
элементов
void print_all_combinations_(ordered_array_set cur_combination,
int left_border, int combination_power, ordered_array_set set) {
    //В цикле for перебираются возможные значения для текущего
элемента
    for (int i = left_border; i <= (set.size - combination_power +
left_border) && (i < set.size); i++) {
        //В подмножество-сочетание добавляется новый элемент,
выбранный в цикле for
        ordered_array_set_insert(&cur_combination, set.data[i]);
        //Если полученное сочетание - решение, оно выводится на
экран
        if (cur_combination.size == combination_power) {
            ordered_array_set_print(cur_combination);
        //Если полученное сочетание - не решение, этой же функцией
добавляется его следующий элемент
        //При этом "левая граница" - элемент, следующий после
добавленного на текущей итерации
        } else {
            print_all_combinations_(cur_combination, i+1,
```

```
combination_power, set);
        }
        //Независимо от результата итерации, только что
добавленный элемент удаляется, откатывая "заготовку" для сочетания
к исходному варианту
        ordered_array_set_deleteElement(&cur_combination,
set.data[i]);
    }
}

//Выводит на экран все сочетания множества set по
combination_power с помощью алгоритма поиска с возвращением
//Обертка функции print_all_combinations_, начинающая работу с
пустым стартовым сочетанием cur_combination и индексом левой
границы 0
//Также здесь обрабатывается случай, когда длина требуемого
сочетания равна 0
void print_all_combinations(int combination_power,
ordered_array_set set) {
    assert(combination_power <= set.size);
    if (combination_power == 0) {
        ordered_array_set_print(ordered_array_set_create(0));
    } else {

print_all_combinations_(ordered_array_set_create(combination_power
), 0, combination_power, set);
    }
}
```

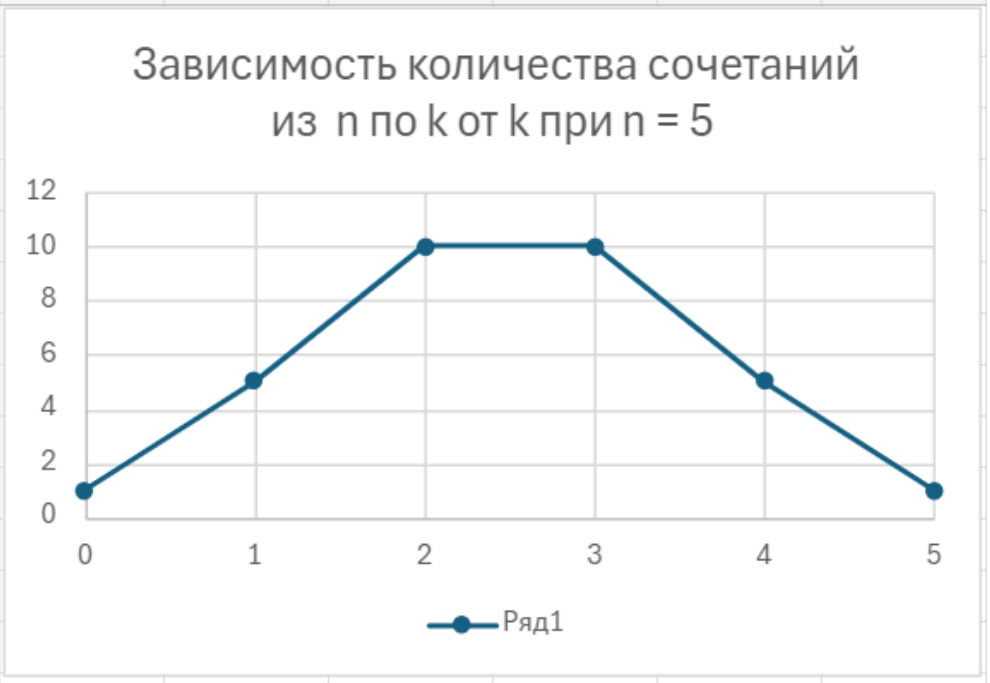
Задание 7.

Задание: построить графики зависимости количества всех сочетаний из n по k от k при n = (5, 6, 7, 8, 9).

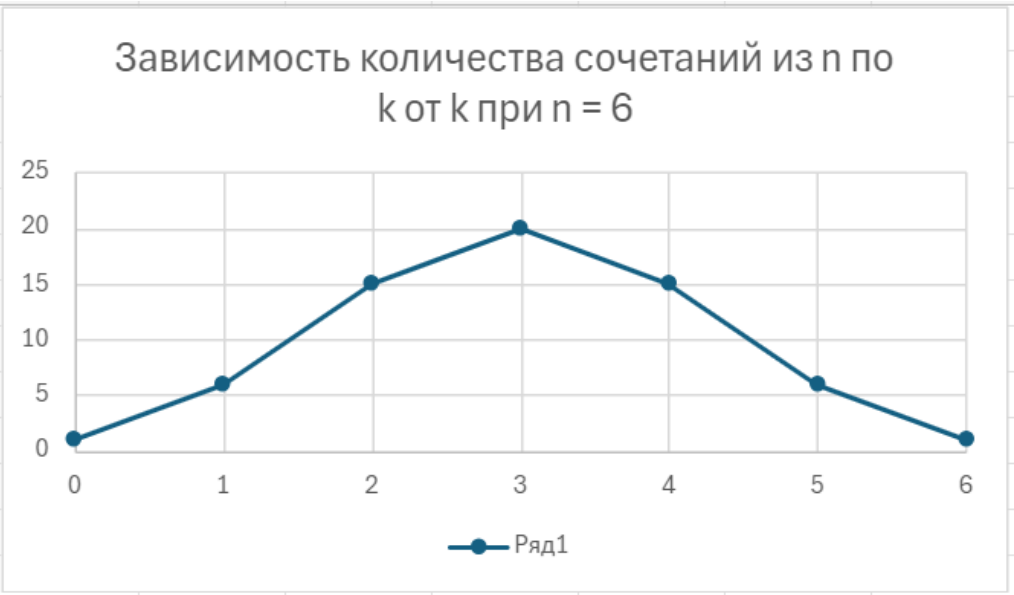
Как мы уже знаем, количество различных сочетаний из n по k описывается формулой $\frac{n!}{k!(n-k)!}$. Процесс ее вывода можно описать так: пусть нам нужно выбрать k неповторяющихся элементов из n элементов исходного множества. Если первый элемент можно выбрать n способами, то второй – (n-1) способами, третий – (n-2) способами, ..., k-тый – (n-k+1) способами. Произведение этих скобок описывается выражением $\frac{n!}{(n-k)!}$. Однако при таком способе подсчета вариантов не учитывается, что будут посчитаны также и варианты с повторяющимся набором элементов, но разным порядком их вхождения. Чтобы понять, сколько «дублей» будет у каждого сочетания, воспользуемся тем же правилом произведений. Первый элемент из k можно выбрать k способами, второй – (k-1) способами, ..., k-тый – 1 способом, и произведение этих скобок даст k!. Значит, разделив предыдущее выражение на k!, получим количество уникальных сочетаний.

Воспользовавшись этой формулой, можно с помощью электронной таблицы автоматически рассчитать количество сочетаний по k в зависимости от значения k для любого n из множества {5, 6, 7, 8, 9}, построив график по этим данным.

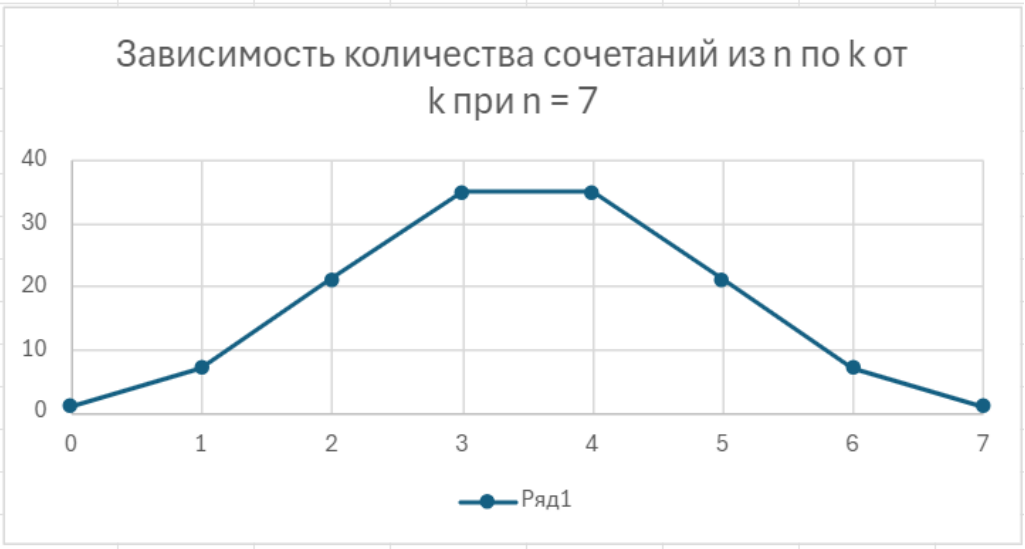
n =	5
0	1
1	5
2	10
3	10
4	5
5	1



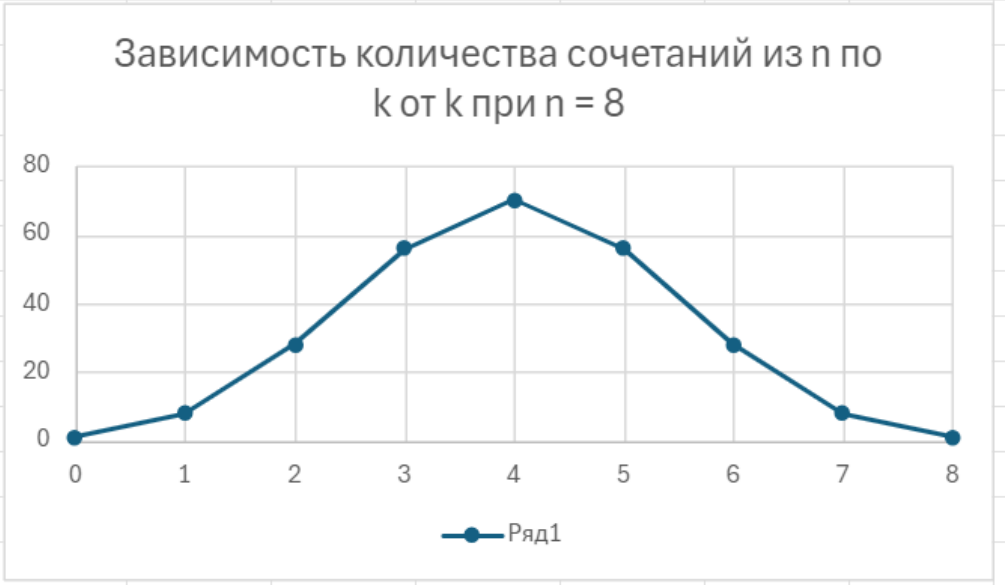
n =	6
0	1
1	6
2	15
3	20
4	15
5	6
6	1



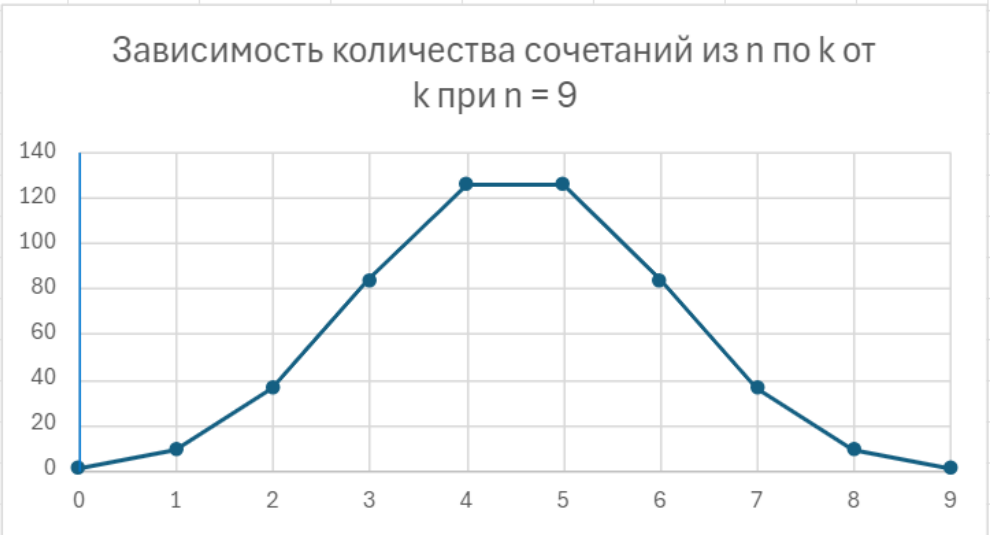
n =	7
0	1
1	7
2	21
3	35
4	35
5	21
6	7
7	1



n =	8
0	1
1	8
2	28
3	56
4	70
5	56
6	28
7	8
8	1



n =	9
0	1
1	9
2	36
3	84
4	126
5	126
6	84
7	36
8	9
9	1



Задание 8.

Задание: реализовать алгоритм порождения перестановок.

Для порождения перестановок точно так же используется общий алгоритм поиска с возвращением, но с его реализацией в этот раз возникают дополнительные трудности. Этот алгоритм предполагает, что рекурсивной функции будет также передан массив чисел, из которых будет выбираться i-тый элемент перестановки, и из этого массива постепенно убираются уже использованные в перестановке элементы. Но в C мы передаем функции указатель на область памяти, то есть не можем создать новый массив без какого-то элемента, а должны менять уже существующий. Полученная функция учитывает эту особенность (пояснения также даны в комментариях).

```
#include "ordered_array_set\ordered_array_set.c"
#include <sys/time.h>

//Выводит на экран все перестановки множества set с помощью
алгоритма поиска с возвращением
//Рекурсивная функция без обертки; на каждой итерации она
опирается на текущее значение формируемой перестановки
cur_arrangement
//ее длину cur_len и множество вариантов для текущего элемента
set_of_variants
void print_all_arrangements_(int *cur_arrangement, size_t cur_len,
ordered_array_set set_of_variants, ordered_array_set set) {
    //В цикле for перебираются возможные значения для текущего
элемента
    for (int i = 0; i < set_of_variants.size; i++) {
        //В перестановку добавляется новый элемент, выбранный в
цикле for, он же удаляется из множества вариантов
        append_(cur_arrangement, &cur_len,
set_of_variants.data[i]);
        ordered_array_set_deleteElement(&set_of_variants,
cur_arrangement[cur_len - 1]);
        //Если полученная перестановка - решение, она выводится на
экран
        if (cur_len == set.size) {
            outputArray_(cur_arrangement, set.size);
        //Если полученная перестановка - не решение, этой же
функцией добавляется ее следующий элемент
        //Значение cur_len е увеличивается на 1, потому что это
уже выполняет функция append_
        } else {
            print_all_arrangements_(cur_arrangement, cur_len,
set_of_variants, set);
        }
        //Независимо от результата итерации, только что
добавленный элемент удаляется из перестановки и добавляется
обратно в массив вариантов
```

```
        ordered_array_set_insert(&set_of_variants,
cur_arrangement[cur_len - 1]);
        deleteByPosSaveOrder_(cur_arrangement, &cur_len, cur_len -
1);
    }
}

//Выводит на экран все перестановки множества set с помощью
алгоритма поиска с возвращением
//Обертка функции print_all_arrangements_, начинающая работу с
пустой стартовой перестановкой, ее длиной 0 и множеством
вариантов, равным исходному; все это создается в теле функции
//Также здесь обрабатывается случай, когда длина исходного
множества равна 0
void print_all_arrangements(ordered_array_set set) {
    if (set.size == 0) {
        ordered_array_set_print(ordered_array_set_create(0));
    } else {
        ordered_array_set start_set_of_variants =
ordered_array_set_create_from_array(set.data, set.size);
        int start_arrangement[set.size];
        int start_len = 0;
        print_all_arrangements_(start_arrangement, start_len,
start_set_of_variants, set);
    }
}
```

Задание 9.

Задание: построить график зависимости количества всех перестановок от мощности множества.

Как мы уже знаем, количество перестановок множества мощности n можно найти как n!. Доказать это можно следующим образом: первый элемент перестановки можно выбрать n способами, второй – (n-1) способами, третий – (n-2) способами, ..., последний – 1 способом, и произведение этих скобок даст n!.

Воспользовавшись этой формулой, можно с помощью электронной таблицы автоматически рассчитать количество перестановок в зависимости от мощности исходного множества, построив график по этим данным.

0	1
1	1
2	2
3	6
4	24
5	120
6	720
7	5040
8	40320
9	362880

Задание 10.

Задание: построить графики зависимости времени выполнения алгоритма п.8 на вашей ЭВМ от мощности множества.

Мы уже писали функции для определения времени генерации подмножеств; для того, чтобы вычислить время генерации перестановок некоторого множества, нужно немного их отредактировать, изменив названия самих функций и название функции, вызываемой в теле print_arrangements_and_time_of_generating:

```
//Выводит на экран структуру timeval time
void print_timeval(struct timeval time) {
    printf("%ld,%.6ld", time.tv_sec, time.tv_usec);
}

//Выводит на экран все перестановки множества set и сообщение о
том, сколько времени выполнялся алгоритм
void print_arrangements_and_time_of_generating (ordered_array_set
set) {
    //Заранее объявляем структуры timeval для хранения времени
начала, конца и длительности процесса генерации
    struct timeval start, end, difference;
    //Задаем значение временной метки перед генерацией и после нее
    gettimeofday(&start, NULL);
    print_all_arrangements(set);
    gettimeofday(&end, NULL);
    //Высчитываем разницу в секундах и микросекундах и сохраняем в
структуру difference
    difference.tv_sec = end.tv_sec - start.tv_sec - (end.tv_usec <
start.tv_usec);
    difference.tv_usec = (end.tv_usec + (end.tv_usec <
start.tv_usec) * 1000000)-start.tv_usec;

    //Вывод сообщения на экран
```

```
    printf("|M| = %llu; time of generation = ", set.size);
    print_timeval(difference);
    printf("\n");
}
```

После чего, каждый раз задавая в теле функции main исходное множество новой длины и вызывая для него функцию print_arrangements_and_time_of_generating, мы будем видеть, в течение какого времени были выведены на экран все возможные перестановки, и сможем составить таблицу на основе этих данных.

```
int main() {
    int a[0] = {};
    ordered_array_set A = ordered_array_set_create_from_array(a,
0);
    print_arrangements_and_time_of_generating(A);
}
```

```
"C:\Users\sovac\Desktop\дискретная математика\ДИСКРЕТКА ЛАБА 2.1\arrangement_creation.exe"
Empty set
|M| = 0; time of generation = 0,000387

Process finished with exit code 0
```

```
int main() {
    int a[1] = {1};
    ordered_array_set A = ordered_array_set_create_from_array(a,
1);
    print_arrangements_and_time_of_generating(A);
}
```

```
"C:\Users\sovac\Desktop\дискретная математика\ДИСКРЕТКА ЛАБА 2.1\arrangement_creation.exe"
1
|M| = 1; time of generation = 0,000219

Process finished with exit code 0
```

```
int main() {
    int a[2] = {1, 2};
    ordered_array_set A = ordered_array_set_create_from_array(a,
2);
    print_arrangements_and_time_of_generating(A);
}
```

```
"C:\Users\sovac\Desktop\дискретная математика\ДИСКРЕТКА ЛАБА 2.1\arrangement_creation.exe"
1 2
2 1
|M| = 2; time of generation = 0,000452

Process finished with exit code 0
```

```c
int main() {
    int a[3] = {1, 2, 3};
    ordered_array_set A = ordered_array_set_create_from_array(a,
3);
    print_arrangements_and_time_of_generating(A);
}
```

```
"C:\Users\sovac\Desktop\дискретная математика\ДИСКРЕТКА ЛАБА 2.1\arrangement_creation.exe"
1 2 3
1 3 2
2 1 3
2 3 1
3 1 2
3 2 1
|M| = 3; time of generation = 0,001404

Process finished with exit code 0
```

Полученная таблица будет выглядеть следующим образом:

0	0,000387	**Проверка:**
1	0,000219	0,000387
2	0,000452	0,000438
3	0,001404	0,001356
4	0,006719	0,005616
5	0,040222	0,033595
6	0,281186	0,241332
7	1,753842	1,968302
8	13,437992	14,030736
9	135,388735	120,941928

Как видим, график похож на тот, что мы получили упражнением ранее. Также, графа «проверка» (каждое значение в которой – время предыдущей генерации, умноженное на количество элементов в текущем множества), пусть и показывает значительные отклонения от реальных результатов на множествах с малым количеством элементов, подтверждает закономерность на множествах с большей мощностью.

Задание 11.

Задание: определить максимальную мощность множества, для которого можно получить все перестановки не более чем за час, сутки, месяц, год на вашей ЭВМ.

Итак, для того, чтобы погрешность в наших вычислениях была минимальной, опять возьмем множество с наибольшим временем генерации перестановок; |M| = 9, t = 135,388735. Тогда для множества с |M| = 10, t = 135,388735*10; для множества с |M| = 11, t = 135,388735*10*11. Из чего можно сделать вывод, что для множества с любой |M| > 9, $t = \frac{135{,}388735 * |M|!}{9!}$. И чтобы найти максимальную |M|,

для которого все перестановки генерируются за время не больше t, нужно найти наибольшее целое решение неравенства $t <= \frac{135{,}388735 * |M|!}{9!}$

Меньше чем за час сгенерируются перестановки множества с |M| таким, что $3600 <= \frac{135{,}388735 * |M|!}{9!}$. При |M| = 10, t ≈ 1353; при |M| = 11, t ≈ 14892. 1353 < 3600 < 14892; следовательно, |M| = 10.

Меньше чем за сутки сгенерируются перестановки множества с |M| таким, что $86400 <= \frac{135{,}388735 * |M|!}{9!}$. При |M| = 11, t ≈ 14892; при |M| = 12, t ≈ 178713. 14892 < 86400 < 178713; следовательно, |M| = 11.

Меньше чем за месяц сгенерируются перестановки множества с |M| таким, что $2592000 <= \frac{135{,}388735 * |M|!}{9!}$. При |M| = 13, t ≈ 2323270; при |M| = 14, t ≈ 32525789. 2323270 < 2592000 < 32525789; следовательно, |M| = 13.

Меньше чем за год сгенерируются перестановки множества с |M| таким, что $31536000 <= \frac{135{,}388735 * |M|!}{9!}$. Обратимся к предыдущему абзацу: 2323270 < 31536000 < 32525789; следовательно, |M| = 13, как и в предыдущем случае.

Задание 12.

Задание: определить максимальную мощность множества, для которого можно получить все перестановки не более чем за час, сутки, месяц, год на ЭВМ, в 10 и в 100 раз быстрее вашей.

Закономерности, указанные выше, справедливы и для ЭВМ с большими мощностями, однако «контрольное время» при |M| = 9 будет для них равно 135,388735/10 = 13,5388735 и 135,388735/100 = 1,35388735 секунд соответственно.

Для ЭВМ с мощностью в 10 раз больше нашей, меньше чем за час сгенерируются перестановки множества с |M| таким, что $3600 <= \frac{13{,}5388735 * |M|!}{9!}$. При |M| = 11, t ≈ 1489; при |M| = 12, t ≈ 17871. 1489 < 3600 < 17871; следовательно, |M| = 11.

Меньше чем за сутки сгенерируются перестановки множества с |M| таким, что $86400 <= \frac{13{,}5388735 * |M|!}{9!}$. При |M| = 12, t ≈ 17871; при |M| = 13, t ≈ 232327. 17871 < 86400 < 232327; следовательно, |M| = 12.

Меньше чем за месяц сгенерируются перестановки множества с |M| таким, что $2592000 <= \frac{13{,}5388735 * |M|!}{9!}$. При |M| = 13, t ≈ 232327; при |M| = 14, t ≈ 3252578. 232327 < 2592000 < 3252578; следовательно, |M| = 13.

Меньше чем за год сгенерируются перестановки множества с |M| таким, что $31536000 <= \frac{13{,}5388735 * |M|!}{9!}$. При |M| = 14, t ≈ 3252578; при |M| = 15, t ≈ 48788684. 3252578 < 31536000 < 48788684, следовательно, |M| = 14.

Для ЭВМ с мощностью в 100 раз больше нашей, меньше чем за час сгенерируются перестановки множества с |M| таким, что $3600 <= \frac{1{,}35388735 * |M|!}{9!}$. При |M| = 12, t ≈ 1787; при |M| = 13, t ≈ 23232. 1787 < 3600 < 23232; следовательно, |M| = 12.

Меньше чем за сутки сгенерируются перестановки множества с |M| таким, что $86400 <= \frac{1{,}35388735 * |M|!}{9!}$. При |M| = 13, t ≈ 23232; при |M| = 14, t ≈ 325257. 23232 < 86400 < 325257; следовательно, |M| = 13.

Меньше чем за месяц сгенерируются перестановки множества с |M| таким, что $2592000 <= \frac{1{,}35388735 * |M|!}{9!}$. При |M| = 14, t ≈ 325257; при |M| = 15, t ≈ 4878868. 325257 < 2592000 < 4878868; следовательно, |M| = 14.

Меньше чем за год сгенерируются перестановки множества с |M| таким, что $31536000 <= \frac{1{,}35388735 * |M|!}{9!}$. При |M| = 15, t ≈ 4878868; при |M| = 16, t ≈ 78061895. 4878868 < 31536000 < 78061895; следовательно, |M| = 16.

Задание 13.

Задание: реализовать алгоритм порождения размещений.

Алгоритм порождения размещений по k почти ничем не отличается от алгоритма порождения перестановок, за исключением того, что решение считается найденным, когда длина формируемой перестановки равна заданному k, а не длине исходного множества. Поэтому нам остается внести в алгоритм порождения перестановок лишь небольшие коррективы.

```
#include "ordered_array_set\ordered_array_set.c"
#include <assert.h>

//Выводит на экран все размещения множества set по placement_power
с помощью алгоритма поиска с возвращением
//Рекурсивная функция без обертки; на каждой итерации она
опирается на текущее значение формируемого размещения
cur_placement
//ее длину cur_len и множество вариантов для текущего элемента
set_of_variants
void print_all_placements_(int *cur_placement, size_t cur_len,
ordered_array_set set_of_variants, int placement_power,
ordered_array_set set) {
    for (int i = 0; i < set_of_variants.size; i++) {
        append_(cur_placement, &cur_len, set_of_variants.data[i]);
        ordered_array_set_deleteElement(&set_of_variants,
cur_placement[cur_len - 1]);
        //Полученное размещение считается решением, если его длина
равна placement_power
        if (cur_len == placement_power) {
            outputArray_(cur_placement, placement_power);
```

```
        } else {
            print_all_placements_(cur_placement, cur_len,
set_of_variants, placement_power, set);
        }
        ordered_array_set_insert(&set_of_variants,
cur_placement[cur_len - 1]);
        deleteByPosSaveOrder_(cur_placement, &cur_len, cur_len -
1);
    }
}
```

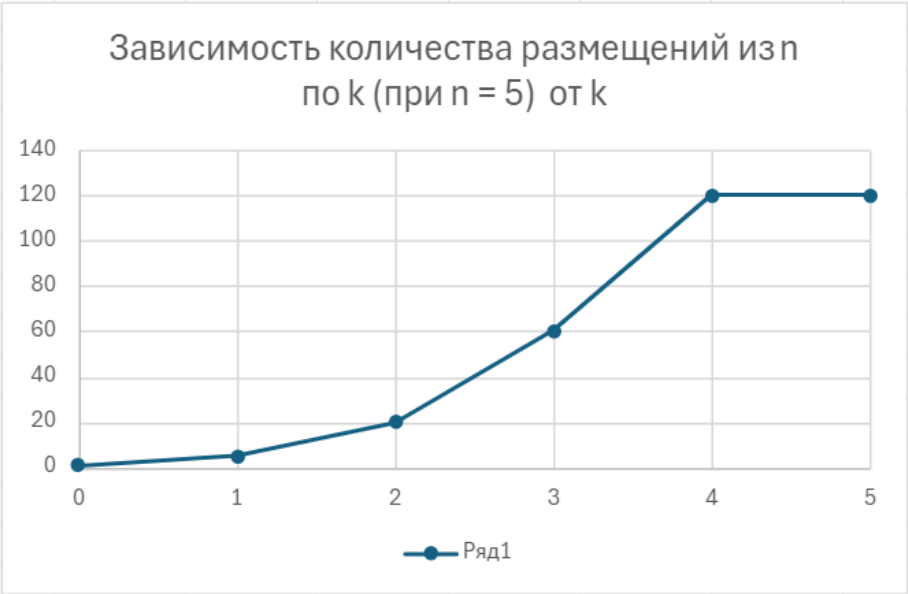
```
//Выводит на экран все размещения множества set с помощью
алгоритма поиска с возвращением
//Обертка функции print_all_placements_, начинающая работу с
пустым стартовым размещением, его длиной 0 и множеством вариантов,
равным исходному; все это создается в теле функции
//Также здесь обрабатывается случай, когда длина требуемого
размещения равна 0
void print_all_placements(int placement_power, ordered_array_set
set) {
    assert(placement_power <= set.size);
    if (set.size == 0) {
        ordered_array_set_print(ordered_array_set_create(0));
    } else {
        ordered_array_set start_set_of_variants =
ordered_array_set_create_from_array(set.data, set.size);
        int start_placement[set.size];
        int start_len = 0;
        print_all_placements_(start_placement, start_len,
start_set_of_variants, placement_power, set);
    }
}
```

Задание 14.

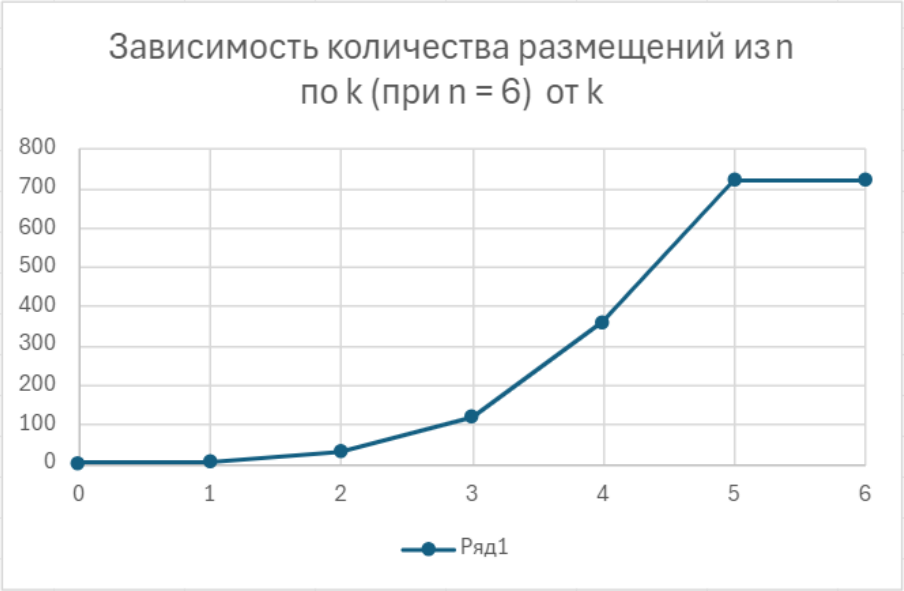
Задание: построить графики зависимости количества всех размещений из n по k от k при n = (5, 6, 7, 8, 9).

Как мы уже знаем, количество размещений из n по k описывается формулой $\frac{n!}{(n-k)!}$. Процесс ее вывода можно описать так: пусть нам нужно выбрать k неповторяющихся элементов из n элементов исходного множества. Если первый элемент можно выбрать n способами, то второй – (n-1) способами, третий – (n-2) способами, ..., k-тый – (n-k+1) способами. Произведение этих скобок и описывается выражением $\frac{n!}{(n-k)!}$. Воспользовавшись этой закономерностью, можно составить электронную таблицу и построить графики искомой зависимости.

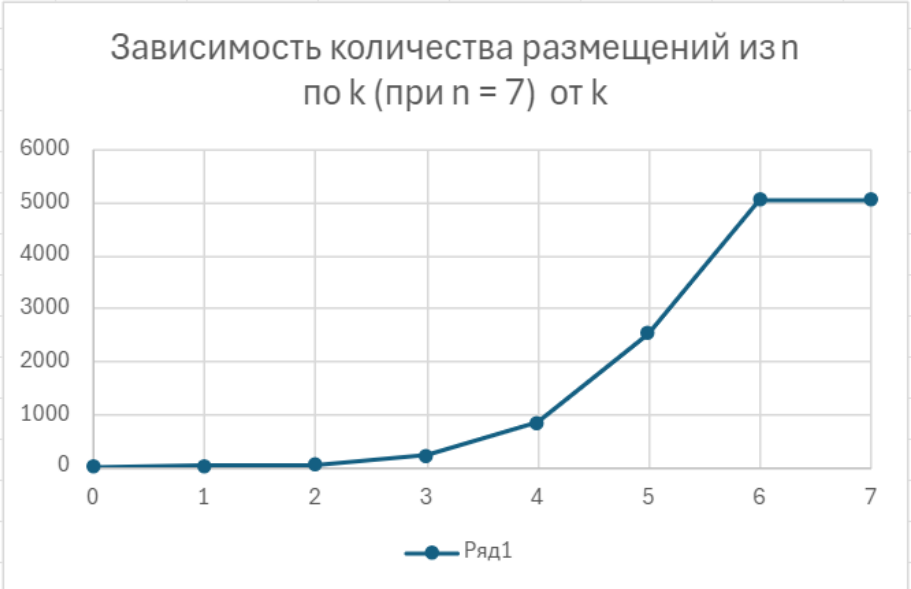
n = 5	
0	1
1	5
2	20
3	60
4	120
5	120



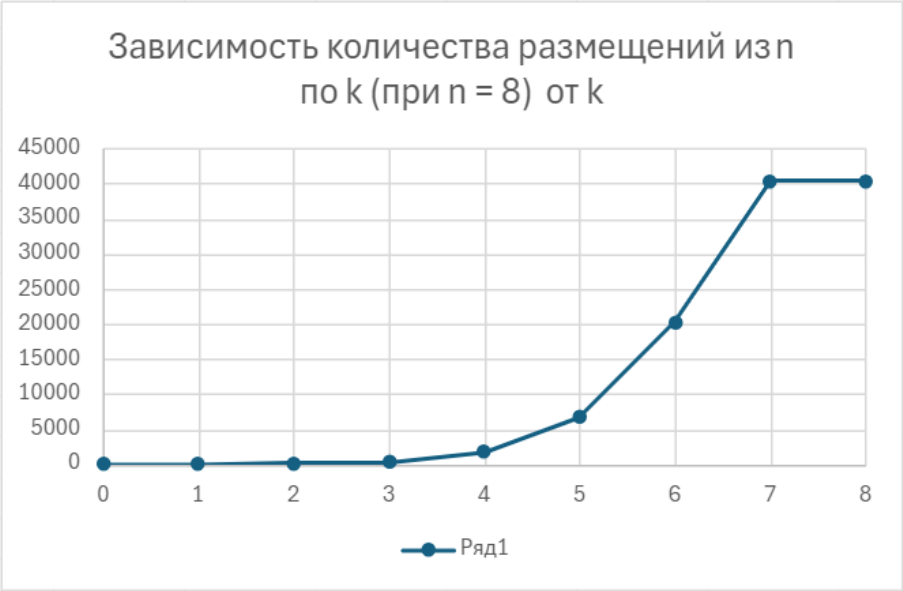
n = 6	
0	1
1	6
2	30
3	120
4	360
5	720
6	720



n = 7	
0	1
1	7
2	42
3	210
4	840
5	2520
6	5040
7	5040



n = 8	
0	1
1	8
2	56
3	336
4	1680
5	6720
6	20160
7	40320
8	40320



n = 9	
0	1
1	9
2	72
3	504
4	3024
5	15120
6	60480
7	181440
8	362880
9	362880

Вывод:

В ходе выполнения лабораторной работы изучены основные комбинаторные объекты – наборы, составленные из элементов множества и обладающие определяющим для этого типа комбинаторных объектов свойством.

Универсальный способ генерации комбинаторных объектов – алгоритм поиска с возвращением. Для некоего i-го объекта последовательности выполняется рекурсивный алгоритм: выбирается один из элементов, который может быть i-ым элементом этого объекта. Если полученная после этого выбора последовательность является решением, то оно возвращается/сохраняется/выводится на экран, и перебор продолжается для i-го объекта. Если последовательность не является решением, то рекурсивный алгоритм вызывается вновь для (i+1)-го элемента этой же последовательности. Если все возможные варианты для i-го объекта перебраны, то алгоритм возвращается к итерации, обрабатывающей (i-1)-ый объект. Если i = 1 и возвращение на шаг невозможно, это значит, все возможные решения найдены.

Количество комбинаторных объектов от какого-либо множества стремительно возрастает при увеличении мощности этого множества, и генерация всех комбинаторных объектов быстро становится невозможной задачей, независимо от мощности ЭВМ, на которой она производится. Исключение – комбинаторные объекты, которые помимо мощности зависят от числа k, по которому они берутся, и это число является очень малым (для сочетаний и размещений) или очень большим (для размещений).