

План-конспект к занятию № 11

Аннотация

№ занятия	11
Тема занятия	Оценка сложности алгоритмов. Поиск и сортировка данных
Продолжительность занятия	100 минут
Используемое оборудование	Компьютер с установленным компилятором g++

Цель занятия

Сформировать знания об оценке сложности алгоритмов, поиске и сортировке данных. Изучить алгоритмы линейного поиска и сортировки подсчетом.

Результат работы

Разработанный комплекс программ, использующих поиск и сортировку данных.

Программа занятия

1. Логарифм
2. Оценка сложности алгоритма. Нотация O -большое
3. Константные алгоритмы. $O(1)$
4. Линейные алгоритмы. $O(N)$
5. Логарифмические алгоритмы. $O(k \log N)$
6. Сравнение $O(1)$, $O(N)$, $O(\log N)$ и $O(N \log N)$
7. Полиномиальные алгоритмы. $O(N^p)$
8. Сравнение $O(N \log N)$ и $O(N^2)$
9. Экспоненциальные алгоритмы. $O(k^N)$
10. Сравнение $O(N^3)$ и $O(2^N)$
11. Факториальные алгоритмы. $O(N!)$
12. Сравнение $O(2^N)$ и $O(N!)$
13. Поиск и сортировка данных
14. Линейный поиск
15. Сортировка подсчетом
16. Случайные числа. Функции `rand()` и `srand()`
17. Практические задания

Методы обучения

На достижение целей занятия направлены используемые педагогом методы. Используются следующие методы учебно-познавательной деятельности: словесный, методы и приёмы самостоятельной работы.

Выбранные методы позволяют реализовать цели системно-деятельностного подхода. Форма проведения занятий выбрана с учетом возрастных особенностей и способствует формированию личностных и регулятивных универсальных учебных действий.

Ход занятия

1. Организационный момент

Приветствие. Определение отсутствующих. Проверка готовности обучающихся к занятию.

2. Структура изучения нового материала

1. Объяснение нового материала (теория)
2. Описание результата занятия (описание работы программы)
3. Разработка программы
4. Тестирование (испытание программы)
5. Подведение итогов занятия (рефлексия / выводы)

Логарифм

ЛОГАРИФМ положительного числа B по основанию A (обозначается, как $\log_A B$) — это степень, в которую нужно возвести A , чтобы получить B .

$$B > 0$$

$$A > 0$$

$$A \neq 1$$

Допустим, что $\log_A B = C$, тогда $A^C = B$

Оценка сложности алгоритма. Нотация O - большое

Существует огромное множество алгоритмов для выполнения совершенно различных задач. Разные алгоритмы могут решать одну и ту же задачу; алгоритмы могут комбинироваться друг с другом и представлять собой новый алгоритм. И среди этого разнообразия необходимо выбрать алгоритм, наиболее подходящий для решения вашей задачи. Важным качеством хорошего программиста является умение еще на этапе разработки программы (или даже планирования) находить самое подходящее для задачи решение, которое будет работать максимально эффективно (выполняться как можно меньшее количество времени и занимать как можно меньше памяти компьютера).

Для того, чтобы понять насколько эффективен алгоритм, не запуская программу, необходимо уметь анализировать, так называемую, **сложность алгоритма**, которая показывает какое количество времени и памяти компьютера занимает определенный алгоритм.

В оценке сложности алгоритма время и память рассматриваются, как ресурс, показывая лишь численное значение без привязки к физическим свойствам. Это можно представить так, как если бы время и память были баллами, количеством которых можно определить "стоимость" определенной услуги.

Одним из постулатов языка C++ является "Не плати за то, что не используешь". Здесь под платой подразумевается именно процессорное время и объем занимаемой памяти.

Существует множество способов оценки сложности алгоритма. Рассмотрим один из наиболее часто употребляемых методов — нотацию O -большое.

O -БОЛЬШОЕ — функция из высшей математики, работающая с пределом последовательности.

Поэтому для простоты мы не будем изучать ее формулу, но будем рассматривать наиболее часто встречающиеся ее примеры.

Опуская сложную математику, мы будем смотреть на алгоритм как на некоторую функцию $g(N)$ и будем сравнивать ее с $O(C_0 \times f(N) + C_1)$, где N — количество входных данных нашего алгоритма, а C_0 и C_1 — некоторые постоянные значения, которые опускаются для простоты понимания полученной оценки. Проводя такое сравнение, мы примерно представляем, что скорость роста сложности нашего алгоритма $g(N)$ будет не больше, чем скорость роста некоторой функции $f(N)$. Это можно представить, как график функции $g(N)$, значения которого всегда лежат ниже или равны графику функции $f(N)$. Таким образом, O -большое показывает как работает функция в наихудшем для нее сценарии и может быть использовано для оценки необходимого времени работы или использования памяти некоторым алгоритмом.

В устной речи, например, $O(N)$ будет произноситься как "О-большое от N " или просто "О от N ". Далее мы будем рассматривать примеры различных функций $f(N)$ относительно времени исполнения, но это так же равнозначимо переносится и на занимаемую память.

Константные алгоритмы. $O(1)$

$O(1)$ описывает алгоритм, который всегда будет исполняться за одно и то же (постоянное или константное) время, независимо от количества входных данных.

```
bool isNull(int element) {  
    return element == null;  
}
```

Ключ к пониманию нотации O -большое лежит в понимании скорости роста функции. Предположим, что у нас есть следующий фрагмент кода:

```
int N = 1000;  
cout << "N = " << N << '\n';
```

Неважно какое значение содержит N — фрагмент кода выше всегда будет выполняться за константное время.

Допустим, теперь мы имеем следующий фрагмент кода:

```
int N = 1000;  
cout << "N = " << N << '\n';  
cout << "N = " << N << '\n';  
cout << "N = " << N << '\n';
```

Второй пример так же работает за константное время, даже несмотря на то, что это занимает в 3 раза больше времени, потому что этот код не зависит от размера N . Мы говорили, что константы C_0 и C_1 в функции $O(C_0 \times f(N) + C_1)$ опускаются. Здесь $f(N) = 1$, поэтому $O(2)$, $O(3)$ или даже $O(100)$ эквивалентны $O(1)$ и значат одно и то же. Для нас не имеет значения как долго будет происходить выполнение — имеет значение лишь то, что это время будет константным.

Линейные алгоритмы. $O(N)$

$O(N)$ описывает алгоритм, чья скорость работы линейно и прямопропорционально зависит от размера входных данных. Пример ниже демонстрирует, как O -большое показывает наихудший сценарий производительности: совпадение заданного символа и символа внутри строки может произойти на любой итерации цикла и на этом функция может закончить свою работу, но нотация O -большое всегда рассматривает алгоритм так, будто он выполнит максимальное количество действий внутри него.

```
bool isContainingValue(string text, char letter) {
    for (size_t i = 0; i < text.length(); ++i) {
        if (text.at(i) == letter) {
            return true;
        }
    }

    return false;
}
```

Когда мы говорим, что что-то растет с линейной скоростью, мы имеем ввиду, что оно растет прямопропорционально количеству его элементов (например, линейная функция $f(x) = kx + b$).

```
for (int i = 0; i < N; ++i) {
    for (int j = 1; j <= 5; ++j) {
        cout << i << ' ';
    }
    cout << '\n';
}
```

Время выполнения линейно зависит от количества входных данных — от N . Нас не интересует какое количество времени займет выполнение одной итерации — нас интересует лишь то, что все итерации выполняются за N единиц времени. И мы знаем, что при увеличении N , например, на **100**, алгоритм будет выполняться за $N + 100$ единиц времени (то есть время работы будет расти линейно). Также нас не интересует, что полное представление оценки работы выглядит как $O(5N + N) = O(6N)$, так как мы опускаем константы. Поэтому мы оцениваем этот алгоритм с помощью $O(N)$.

Логарифмические алгоритмы. $O(k \log N)$

$O(\log N)$

$O(\log N)$ описывает алгоритмы, в которых происходит циклическое деление количества входных элементов. График зависимости скорости работы таких алгоритмов от количества входных элементов сначала быстро растет, а затем становится все более плоским.

Допустим, число, на которое делится входное множество, равно 10 . Тогда при $N = 10$, алгоритм сработает за 1 единицу времени. При $N = 100$ — за 2 . При $N = 1000$ — за 3 . Таким образом, при увеличении количества входных данных в 10 раз, время работы увеличивается на 1 единицу времени. Поэтому, такие алгоритмы крайне эффективны на больших объемах данных.

```
for (int i = 1; i < N; i *= 2) {  
    cout << i << '\n';  
}
```

Основанием логарифма в примере выше является 2 , так как мы выполняем цикл с умножающим шагом 2 (что, по своей сути, является циклическим располовиниванием входных данных).

Если $N = 8$, то вывод будет следующим:

```
1  
2  
4
```

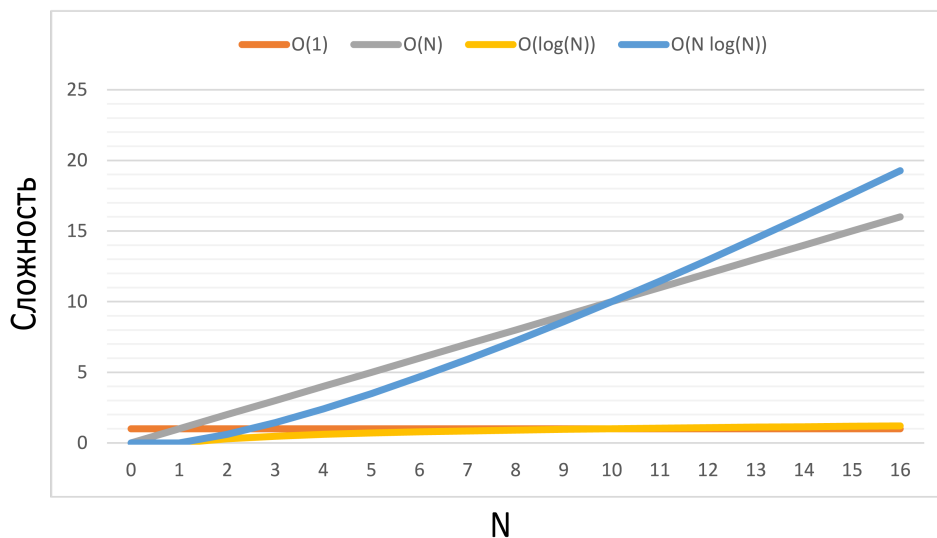
Таким образом, наш алгоритм выполнен за $\log_2 8 = 3$ единицы времени.

$O(N \log N)$

```
for (int i = 1; i <= N; ++i) {  
    for (int j = 0; j < N; j *= 2) {  
        cout << i << ' ' << j << '\n';  
    }  
}
```

Пусть $N = 8$, тогда алгоритм будет выполняться за $8 \log_2 8 = 8 \times 3 = 24$ единицы времени.

Сравнение $O(1)$, $O(N)$, $O(\log N)$ и $O(N \log N)$



Как и ожидаемо, график:

- $O(1)$ постоянен
- $O(N)$ растет линейно
- $O(\log N)$ до некоторого N растет медленнее константы, но затем "обгоняет" ее
- $O(N \log N)$ до некоторого N растет медленнее константы, но быстрее обычного логарифма. Затем "обгоняет" константу, но растет медленнее линейного, после чего "обгоняет" и его

Вспомним, что нас больше интересует то, как выглядит разница между различными оценками при больших значениях N . Отсюда вытекает важное примечание: необходимо выбирать алгоритм, работающий наиболее эффективно, исходя из входных данных, так как разница между одними и теми же оценками может меняться в зависимости от объема входных данных.

Полиномиальные алгоритмы. $O(N^p)$

Термин **полиномиальный** — это обобщающий термин, включающий в себя квадратные (N^2) функции, кубические (N^3), квадратические (N^4) и так далее. Наиболее важно то, что $O(N^2)$ быстрее, чем $O(N^3)$, который, в свою очередь, быстрее, чем $O(N^4)$ и так далее.

$O(N^2)$

$O(N^2)$ описывает алгоритм, чья производительность прямопропорциональна квадрату размера входных данных. Обычно, это алгоритмы, включающие в себя вложенные циклы по всему объему входных данных. Более глубокие уровни вложения отражаются на оценке в виде $O(N^3)$, $O(N^4)$ и так далее.

```
bool containsDuplicates(string text) {
    for (size_t outer = 0; outer < text.length(); ++outer) {
        for (size_t inner = 0; inner < text.length(); ++inner) {
            if (outer == inner) {
                continue;
            }

            if (text.at(outer) == text.at(inner)) {
                return true;
            }
        }
    }

    return false;
}
```

Допустим, что строка `text` содержит 8 символов. Тогда алгоритм будет выполнен за $8^2 = 64$ единицы времени.

Сравнение $O(N \log N)$ и $O(N^2)$

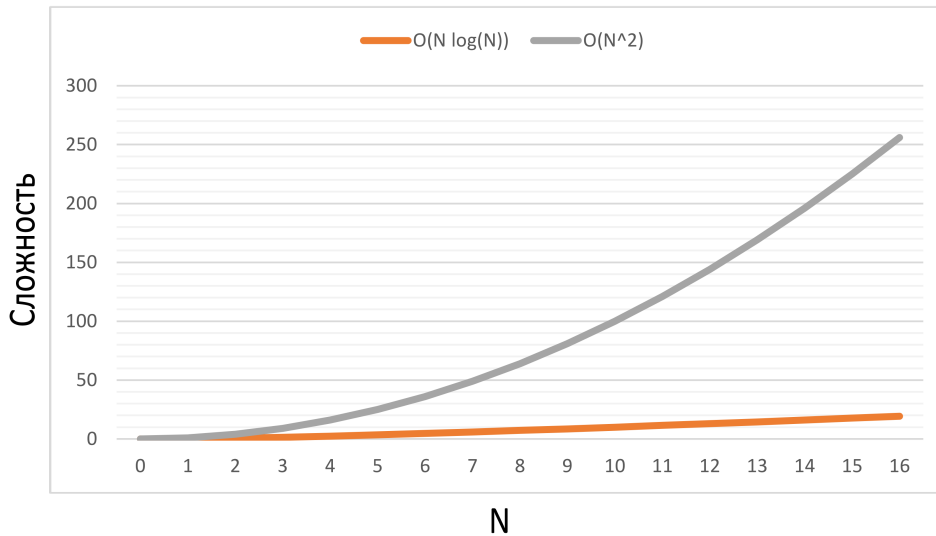


График $O(N \log N)$ растет медленнее, чем график $O(N^2)$.

Экспоненциальные алгоритмы. $O(k^N)$

Время исполнения этих алгоритмов возрастает в k раз при каждом дополнительном входном значении. Например, алгоритмы с оценкой $O(2^N)$ удваивают затрачиваемое на них время с каждым увеличением N ; с оценкой $O(N^3)$ — утраивают.

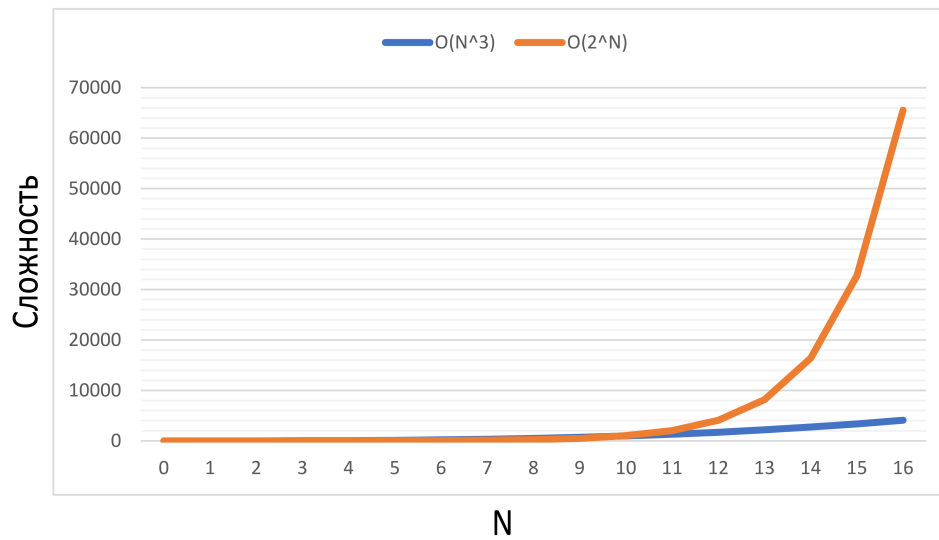
$O(2^N)$

$O(2^N)$ описывает алгоритм, чей рост удваивается с каждым добавлением в множество входных данных. Примером такого алгоритма является рекурсивное вычисление n -ного числа Фибоначчи:

```
int fibonacci(int number) {  
    if (number <= 1) {  
        return number;  
    }  
  
    return fibonacci(number - 2) + fibonacci(number - 1);  
}
```

Если `number` = 8, то алгоритм выполнится за $2^8 = 256$ единиц времени.

Сравнение $O(N^3)$ и $O(2^N)$



При малых значениях N графики $O(N^3)$ и $O(2^N)$ примерно одинаковы, но, начиная с $N = 10$, график $O(2^N)$ начинает стремительно расти.

Факториальные алгоритмы. $O(N!)$

Этот класс алгоритмов выполняется за время, прямопропорциональное факториалу количества входных данных. Классическим примером таких алгоритмов является алгоритм решения задачи коммивояжера, использующий полный перебор.

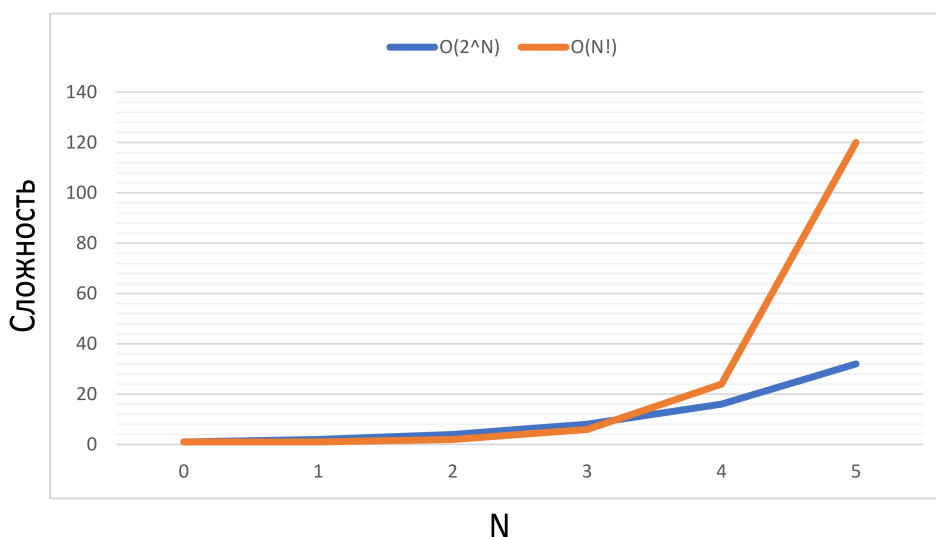
Объяснения задачи коммивояжера и ее решения не входят в рамки текущего занятия.

Более простым примером подобных алгоритмов будет следующий фрагмент кода:

```
for (int i = 0; i < factorial(N); ++i) {  
    cout << i << '\n';  
}
```

Где `factorial(N)` — функция, вычисляющая факториал числа N . Допустим, $N = 8$, тогда алгоритм выполнится за $8! = 40320$ единиц времени.

Сравнение $O(2^N)$ и $O(N!)$



При малых значениях N графики $O(2^N)$ и $O(N!)$ примерно одинаковы, но, начиная с $N = 3$, график $O(N!)$ начинает стремительно расти.

Поиск и сортировка данных

КОЛЛЕКЦИЯ ДАННЫХ — специальным образом организованное множество элементов.

Самой простой коллекцией данных является массив.

В программировании часто требуется искать определенные данные в коллекции. например, поиск данных об аккаунте по его логину; проверка наличия товара на складе по его артикулу и так далее. В сложных системах количество данных, обычно, очень велико и вопрос эффективного поиска и хранения является одним из критически важных мест в этих системах.

ПОИСК ДАННЫХ — процесс поиска элемента коллекции по заданному параметру поиска.

СОРТИРОВКА ДАННЫХ — процесс упорядочивания коллекции по заданному критерию сравнения (по какому принципу следует располагать элементы коллекции).

Существует множество алгоритмов, решающих задачи поиска на коллекциях определенного типа. Так же, как и существует множество алгоритмов, решающих задачи сортировки данных на коллекциях определенного типа. На этом занятии мы рассмотрим самые простые алгоритмы поиска и сортировки данных.

Линейный поиск

Суть **линейного поиска** заключается в обходе коллекции от ее начала до ее конца с одновременной проверкой необходимых свойств элементов этой коллекции.

Пусть имеется целочисленный массив размера N . Необходимо найти все индексы, по которым в массиве расположено число 4. Используем линейный поиск, сравнивая каждый элемент массива с числом 4 и выводя все индексы, в которых было встречено совпадение:

```
for (int i = 0; i < N; ++i) {  
    if (arr[i] == 4) {  
        cout << i << '\n';  
    }  
}
```

Время выполнения

Время выполнения линейного поиска оценивается, как $O(N)$, потому что каждый элемент массива проверяется единожды.

Сортировка подсчетом (Counting sort)

СОРТИРОВКА ПОДСЧЁТОМ — алгоритм сортировки коллекции C , содержащей целые числа в диапазоне от 0 до некоторой константы K ($K = \max(C)$).

Шаги сортировки подсчетом

1. Найдем K — максимальное число в коллекции C
2. Создадим дополнительную коллекцию T размера $K + 1$
3. Пройдемся по коллекции C . Будем прибавлять единицу к элементам коллекции T с индексами, равными значениям элементов коллекции C .

```
for(size_t i = 0; i < N; ++i) {  
    ++T[C[i]];  
}
```

Таким образом мы подсчитали количество вхождений каждого числа из диапазона $[0, K]$ в коллекцию C .

4. Пройдемся по коллекции T . Теперь вернем в коллекцию C каждый элемент из T столько раз, сколько он встречался в C .

```
for(size_t i = 0; i <= K; ++i) {  
    int currentIndex = 0;  
    for(size_t j = 0; j < T[i]; ++j) {  
        C[currentIndex++] = i;  
    }  
}
```

Сложность алгоритма

- Время выполнения — $O(N)$
- Используемая память — $O(N)$

Случайные числа. Функции `rand()` и `srand()`

Часто в программах необходимо использовать случайные числа. Для человека выдумка последовательности случайных чисел кажется чем-то простым, но в действительности каждое следующее случайное число подчиняется некоторой ассоциативной логике конкретного человека. Для компьютера же такая задача становится еще сложнее, так как ему необходимо указать алгоритм, по которому нужно вычислять следующее случайное число. Однако, там, где имеет место определенный заранее алгоритм, нет места истинной случайности. Получается, что последовательность случайных чисел, полученная с помощью компьютера, вовсе не случайна, а **псевдослучайна** — с первого взгляда последовательность действительно выглядит случайной, хотя таковой не является.

Для получения случайных чисел на компьютере используются алгоритмы, именуемые **генераторами псевдослучайных чисел**. И одной из преобладающих концепций таких генераторов является использование некоторого числа, используемого в качестве отправной точки для всех последующих чисел. Такое число называется **зерном** (англ. *a seed*).

В C++ имеется встроенная функция `rand()`, возвращающая целое число из отрезка `[0, INT_MAX]`, где `INT_MAX` — максимальное значение переменной типа `int` для конкретной ЭВМ или конкретного компилятора).

Для того, чтобы получить "случайное" число из определенного отрезка, используется следующая конструкция:

```
a + rand() % (b - a + 1);
```

Где `a` — левая граница отрезка, а `b` — правая граница отрезка.

По умолчанию, в каждой программе на языке C++ используется одно и то же зерно, если оно не задано пользователем. Таким образом, если пользователь не задал зерно самостоятельно, то при каждом запуске программы, последовательность генерируемых случайных чисел будет одной и той же. Для того, чтобы задать зерно вручную, используется встроенная функция `srand()`. Она принимает целое число, но если оно константно задано в программе, то это логически не будет отличаться от использования значения зерна по умолчанию. Такое поведение может быть полезно, например, при тестировании программы для того, чтобы проверить не изменилось ли поведение алгоритма после его модификации на одном и том же наборе данных. Но чтобы получать различные последовательности случайных чисел при независимых запусках программы, в качестве зерна используется текущее время, которое легко получить, используя функцию `time()` из библиотеки `<time.h>` с аргументом `NULL`: `time(NULL)`.

```
#include <time.h>

...

srand(time(NULL));
int r = a + rand() % (b - a + 1);
```

Практические задания

1. Линейный поиск

Массив заполняется N случайными числами из отрезка $[a, b]$. Необходимо найти все индексы массива, в которых находится значение k .

Пользователю предлагается ввести N , k , a и b .

Вывести на экран сгенерированный массив и все индексы элементов массива, в которых находится число k . Вывести -1 , если число не было найдено в массиве.

Пример 1:

```
Enter N, k, a, b:
5 8 -10 10

-1 -8 -3 8 -5
Indices of 8: 3
```

Пример 2:

```
Enter N, k, a, b:
5 8 -10 10

5 -7 -6 -3 9
Indices of 8: -1
```


2. Сортировка подсчетом

Массив заполняется N случайными числами из отрезка $[a, b]$. Необходимо отсортировать массив по возрастанию и по убыванию значений.

Пользователю предлагается ввести N , a и b .

Вывести на экран сгенерированный массив, отсортированный по возрастанию массив и отсортированный по убыванию массив.

Пример 1:

```
Enter N, a, b:
```

```
5 50 100
```

```
59 98 93 68 74
```

```
59 68 74 93 98
```

```
98 93 74 68 59
```

Пример 2:

```
Enter N, a, b:
```

```
5 -50 50
```

```
'a' must be bigger than 0!
```