Семинар №1

Сложность алгоритмов

1. **Инструментарий:**

Блок содержит ссылки на презентации, тесты, инструкции и др. полезную информацию к семинару

---

2. **Цели семинара №1:**
    - Научиться определять сложность алгоритмов
    - Закрепить знание признаков, характерных для той или иной сложности алгоритма
    - Попрактиковаться в навыках написания алгоритмов определенной сложности
    - Сравнить производительность различных вариантов алгоритмов для решения одинаковой задачи

    По итогам семинара №1 слушатель должен **знать**:
    - Знать такие сложности, как: линейная, квадратичная, экспоненциальная
    - Признаки указанных сложностей

    По итогам семинара №1 слушатель должен **уметь**:
    - Писать алгоритмы различной сложности
    - Уметь определять сложность алгоритма по признакам
    - Выбирать оптимальную сложность из возможных

Формат проведения семинара

1. Ученики разбиваются на группы в сессионные комнаты зум
2. Один из учеников расшаривает экран
3. Обсуждают задание и как его лучше выполнить
4. Ученик, расшаривший экран пишет код, другие ученики участвуют в обсуждении и подсказывают что нужно еще добавить
5. После выполнения первого задания, ученики скидывают архивы с работой для проверки

3. План Содержание:

Этап урока	Тайминг, минуты	Формат
Знакомство со студентами	5	Модерирует преподаватель
Вопросы по лекции, подготовка к семинару	10	Преподаватель спрашивает, есть ли вопросы по пройденному материалу и отвечает на них
Подготовка алгоритма линейной сложности (сумма всех чисел от 1 до N)	5	Преподаватель дает задачу реализовать алгоритм, напоминает свойства линейной сложности
Подготовка алгоритма квадратичной сложности (поиск простых чисел от 1 до N)	10	Преподаватель дает задачу реализовать алгоритм, напоминает свойства квадратичной сложности
Пишем алгоритм экспоненциальной сложности (поиск всех возможных комбинаций игральных кубиков K (4) с количеством граней N (6)	15	Преподаватель дает задачу реализовать алгоритм, на выбор 2 варианта – полноценный алгоритм для K кубиков с N граней, либо упрощенный вариант для 4 кубиков с N граней. Напоминает свойства экспоненциальной сложности
Пишем алгоритм поиска чисел Фибоначчи рекурсивным способом (экспоненциальная сложность), от последнего числа к первому	10	Преподаватель дает задачу реализовать алгоритм, дает пояснение, как именно можно определить его сложность, почему она будет считаться экспоненциальной
Пишем алгоритм поиска чисел Фибоначчи линейным алгоритмом, от первого числа к последнему	10	Преподаватель дает задачу реализовать алгоритм, указывает на возможность решить одну и ту же задачу разными способами
Делаем сравнение времени выполнения поиска чисел Фибоначчи двумя написанными алгоритмами, оцениваем какой из них быстрее, выбирает оптимальный по производительности вариант	10	Преподаватель предлагает измерить скорость выполнения алгоритмов и сделать выводы, какой из них будет оптимальнее
Длительность:	**85**	

4. Блок 1.

Задание:
Необходимо написать алгоритм, считающий сумму всех чисел от 1 до N. Согласно свойствам линейной сложности, количество итераций цикла будет линейно изменяться относительно изменения размера N.

Пример решения:

```
public static int sum(int lastNumber) {
    int sum = 0;
    for (int i = 1; i <=lastNumber; i++){
        sum+=i;
    }
    return sum;
}
```

5. Блок 1.

Задание:
Написать алгоритм поиска простых чисел (делятся только на себя и на 1) в диапазоне от 1 до N. В алгоритме будет использоваться вложенный for, что явно говорит о квадратичной сложности, на этом стоит акцентировать внимание

Пример решения:

```
public static List<Integer> findSimpleNumbers(int lastNumber) {
    List<Integer> result = new ArrayList<>();
    boolean simple = true;
    for (int i = 1; i <= lastNumber; i++) {
        simple = true;
        for (int j = 2; j < i; j++) {
            if (i % j == 0) {
                simple = false;
                break;
            }
        }
        if (simple) {
            result.add(i);
        }
    }
    return result;
}
```

Часто встречающиеся ошибки:
Не стоит включать во вложенный цикл граничные значения – деление на 1 и на само себя выполняется в любом случае. Требуется проверить только значения больше единицы и меньше самого числа

6. Блок 1.

Задание:
Необходимо написать алгоритм поиска всех доступных комбинаций (посчитать количество) для количества кубиков K с количеством граней N. При этом, стоит предложить студентам 2 варианта на выбор – количество кубиков может быть строго ограничено (4 кубика, например),

либо их количество будет динамическим. Какой вариант алгоритма они захотят реализовать – тот и реализуют.
Если студент реализует простой вариант, обращает внимание, что данное решение имеет сложность $O(n^4)$, но если количество кубиков сделать переменной, то она трансформируется в $O(n^k)$, что будет представлять собой экспоненциальную сложность. Для второго решения очевидно, что его сложность $O(n^k)$ с самого начала.

Пример решения:

```
/**
 * Простой вариант (количество кубиков 4)
 */
public static int combinationCount(int faces) {
    int count = 0;
    for (int i = 1; i <= faces; i++) {
        for (int j = 1; j <= faces; j++) {
            for (int k = 1; k <= faces; k++) {
                for (int l = 1; l <= faces; l++) {
                    count++;
                }
            }
        }
    }
    return count;
}

/**
 * Сложный вариант - количество кубиков задается условием
 */
public static int combinationCount(int count, int faces) {
    if (count > 0) {
        return recursiveCounter(1, count, faces);
    } else {
        return 0;
    }
}

private static int recursiveCounter(int depth, int maxDepth, int faces) {
    int count = 0;
    for (int i = 1; i <= faces; i++) {
        if (depth == maxDepth) {
            count++;
        } else {
            count += recursiveCounter(depth + 1, maxDepth, faces);
        }
    }
    return count;
}
```

7. Блок 1.

Задание:

Пишем алгоритм поиска нужного числа последовательности Фибоначчи. Считаем, что 1 и 2 значения последовательности равны 1. Искать будем по формуле $O_n=O_{n-1}+O_{n-2}$ что предполагает использовать рекурсивного алгоритма. Можно объяснить на псевдокоде, чтобы не писать готовое решение вместо студентов. После написания стоит показать схему расчета сложности из методических материалов, где явно видно, что подобного рода рекурсия имеет экспоненциальную сложность.

```
                          n
                        /     \
                      n-1      n-2      -------- 2^1 операций
                     /  \     /   \
                   n-2  n-3 n-3 n-4    -------- 2^2 операций
                  /   \
                n-3 n-4                -------- 2^3 операций
                ........               ........
               n-n-1  n-n-2            -------- 2^(n-1) операций
```

Пример решения:

```
public static int fb(int num) {
    if (num <= 2) {
        return 1;
    } else {
        return fb(num-1) + fb(num-2);
    }
}
```

8. Блок 1.

Задание:
Пишем алгоритм поиска нужного числа последовательности Фибоначчи, но в этот раз откажемся от рекурсии и воспользуемся обычным алгоритмом, что даст нам линейную сложность, т.к. вычисление каждого из чисел последовательности будет происходить ровно 1 раз. Вариантов решения может быть несколько, но нужно предложить студентам считать последовательность с первого числа и далее до тех пор, пока не доберемся до нужного номера. При этом значение предыдущих элементов нужно сохранять, чтобы не приходилось вычислять заново, как это происходило при рекурсивном методе.

Пример решения:

```
public static int fb(int num) {
    if (num <= 2) {
        return 1;
    } else {
        final int[] numbers = new int[num];
        numbers[0] = 1;
        numbers[1] = 1;
        for (int i = 2; i < numbers.length; i++) {
            numbers[i] = numbers[i - 1] + numbers[i - 2];
        }
        return numbers[num - 1];
    }
}
```

9. Блок 1.

Задание:
Необходимо сравнить скорость работы 2 алгоритмов вычисления чисел Фибоначчи и определить, какой из них работает быстрее. Так различия начнутся уже с 40 члена последовательности.

Пример решения:

```
public static void test() {
    for (int i = 10; i <= 50; i = i + 10) {
        Date startDate = new Date();
        fbLine(i);
        Date endDate = new Date();
        long lineDuration = endDate.getTime() - startDate.getTime();
        startDate = new Date();
        fbRecursive(i);
        endDate = new Date();
        long recursiveDuration = endDate.getTime() - startDate.getTime();
        System.out.printf("i: %s, line duration: %s, recursive duration:
%s%n", i, lineDuration, recursiveDuration);
    }
}
```

Пример результата:
i: 10, line duration: 0, recursive duration: 0
i: 20, line duration: 0, recursive duration: 0
i: 30, line duration: 0, recursive duration: 0
i: 40, line duration: 0, recursive duration: 190
i: 50, line duration: 0, recursive duration: 21950

Как итог, можно сделать вывод, что сложность алгоритма напрямую влияет на скорость выполнения задачи и что выбирать нужно алгоритмы с меньшей сложностью, если это возможно