**Приложение №10**

|  |  |
| --- | --- |
| **Дата** | 19.12.2023 |
| **Тема** | 1. Анализ, разработка и реализация алгоритма  2. Проверка на правильность работы отдельных модулей программы |
| **Задания** | 1. Проанализировать, разработать и реализовать алгоритм  2. Проверить на правильность работу отдельных модулей программы |

**1. Анализ, разработка и реализация алгоритма:**

**Что такое анализ?**

Анализируя алгоритм, можно получить представление о том, сколько времени займет решение данной задачи при помощи данного алгоритма. Одну и ту же задачу можно решить с помощью различных алгоритмов. Анализ алгоритмов дает нам инструмент для выбора алгоритма.

Результат анализа алгоритмов — не формула для точного количества секунд или компьютерных циклов, которые потребует конкретный алгоритм. Нужно понимать, что разница между алгоритмом, который делает N + 5 операций, и тем, который делает N + 250 операций, становится незаметной, как только N становится очень большим.  
**Классы входных данных**

При анализе алгоритма выбор входных данных может существенно повлиять на его выполнение. Скажем, некоторые алгоритмы сортировки могут работать очень быстро, если входной список уже отсортирован, тогда как другие алгоритмы покажут весьма скромный результат на таком списке. А вот на случайном списке результат может оказаться противоположным. Поэтому не будем ограничиваться анализом поведения алгоритмов на одном входном наборе данных. Практически нужно искать такие данные, которые обеспечивают как самое быстрое, так и самое медленное выполнение алгоритма. Кроме того, полезно оценивать и среднюю эффективность алгоритма на всех возможных наборах данных.  
**Наилучший случай**

Время выполнения алгоритма в наилучшем случае очень часто оказывается маленьким или просто постоянным, поэтому подобный анализ проводится редко.  
**Наихудший случай**

Анализ наихудшего случая чрезвычайно важен, поскольку он позволяет представить максимальное время работы алгоритма. При анализе наихудшего случая необходимо найти входные данные, на которых алгоритм будет выполнять больше всего работы.  
**Средний случай**

Анализ среднего случая является самым сложным, поскольку он требует учета множества разнообразных деталей. В основе анализа лежит определение различных групп, на которые следует разбить возможные входные наборы данных. На втором шаге определяется вероятность, с которой входной набор данных принадлежит каждой группе. На третьем шаге подсчитывается время работы алгоритма на данных из каждой группы. Время работы алгоритма на всех входных данных одной группы должно быть одинаковым, в противном случае группу следует подразбить.

**Разработка алгоритма**— особый метод для создания математического способа решения проблемы.

**Разработка алгоритма** — это отождествление и объединение во множество решений теорий [исследования операций](https://ru.wikipedia.org/wiki/%D0%98%D1%81%D1%81%D0%BB%D0%B5%D0%B4%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5_%D0%BE%D0%BF%D0%B5%D1%80%D0%B0%D1%86%D0%B8%D0%B9), например [динамическое программирование](https://ru.wikipedia.org/wiki/%D0%94%D0%B8%D0%BD%D0%B0%D0%BC%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5) и [разделяй и властвуй](https://ru.wikipedia.org/wiki/%D0%A0%D0%B0%D0%B7%D0%B4%D0%B5%D0%BB%D1%8F%D0%B9_%D0%B8_%D0%B2%D0%BB%D0%B0%D1%81%D1%82%D0%B2%D1%83%D0%B9_(%D0%B8%D0%BD%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0)). Методиками разработки и реализации разработки алгоритма будут шаблоны[[1]](https://ru.wikipedia.org/wiki/%D0%A0%D0%B0%D0%B7%D1%80%D0%B0%D0%B1%D0%BE%D1%82%D0%BA%D0%B0_%D0%B0%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC%D0%BE%D0%B2#cite_note-1), такие как [шаблонные методы](https://ru.wikipedia.org/wiki/%D0%A8%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD%D0%BD%D1%8B%D0%B9_%D0%BC%D0%B5%D1%82%D0%BE%D0%B4_(%D1%88%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD_%D0%BF%D1%80%D0%BE%D0%B5%D0%BA%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F)) и [декораторы](https://ru.wikipedia.org/wiki/%D0%94%D0%B5%D0%BA%D0%BE%D1%80%D0%B0%D1%82%D0%BE%D1%80_(%D1%88%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD_%D0%BF%D1%80%D0%BE%D0%B5%D0%BA%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F)), использование [структуры данных](https://ru.wikipedia.org/wiki/%D0%A1%D1%82%D1%80%D1%83%D0%BA%D1%82%D1%83%D1%80%D0%B0_%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%D1%85), а также имя и сортировка списков. Сейчас использование разработки алгоритма можно найти в поисковых процессах сканирования Интернета, маршрутизации пакетов и кэшировании.

Одним из наиболее важных качеств алгоритма является его эффективность по времени выполнения и по используемой памяти.

**2. Проверка на правильность работы отдельных модулей программы**

**Тестирование программного обеспечения (Software Testing)** - проверка соответствия между реальным и ожидаемым поведением программы, осуществляемая при конечном наборе тестов, выбранном определенным образом. [IEEE Guide to Software Engineering Body of Knowledge, SWEBOK, 2004].

В более широком смысле, **тестирование** - это одна из техник контроля качества, включающая в себя действия по планированию работ (Test Management), проектированию тестов (Test Design), выполнению тестирования (Test Execution) и анализу полученных результатов (Test Analysis).

Для разработчика же интересней следующее определение:

**Тестирование** — это проверка соответствия программы требованиям, осуществляемая путём наблюдения за её работой в специальных, искусственно созданных ситуациях, выбранных определённым образом.

Хочется обратить внимание на слова “искусственно созданных ситуациях, выбранных определённым образом”. Это означает, что не имеет смысла тестировать вообще все ситуации, стоит выбирать критически важные места и сценарии.

## Уровни тестирования

Тестирование для разработчика состоит из написания тестов. Тест – это такой же программный код, который пишется аналогично коду для реализации бизнес-логики. Тесты проверяют сценарии работы программы (test-case).

Тест-кейсы встречаются самые различные, один от другого может резко отличаться. По желанию можно тестировать ВСЕ возможные и невозможные ситуации. Однако стоит соблюдать адекватность и покрывать код тестами ровно настолько, насколько требуется для уверенного понимания, что бизнес-логика работает как задумано.

Например, функция, которая возвращает числовое значение от 0 до 100. В тестах стоит проверить не только правильность значений из этого диапазона, но и то, что других значений не возникает.

Негласное правило – если на участке кода проявилась ошибка, то стоит написать тест на этот случай.

Тестирование смело делится на несколько уровней глубины. Наиболее показательная классификация по уровням тестирования в данном случае такая:

* Системное тестирование – тестирование полностью интегрированного программного приложения
* Интеграционное тестирование – тестирование интегрированных групп программных модулей
* Модульное тестирование или юнит-тестирование – тестирование отдельных модулей исходного кода приложения

Когда говорят "тестирование", не выделяя конкретный тип, то говорят скорее всего о модульном тестировании.

Пойдем с низу вверх (от 3 к 1)

### Модульное тестирование

Модульное тестирование, или юнит-тестирование (англ. unit testing) — процесс в программировании, позволяющий проверить на корректность отдельные модули исходного кода программы.

Идея состоит в том, чтобы писать тесты для каждой нетривиальной функции или метода. Это позволяет достаточно быстро проверить, не привело ли очередное изменение кода к регрессии, то есть к появлению ошибок в уже протестированных местах программы и облегчает обнаружение и устранение таких ошибок.

#### **Инструменты**

В Python для модульного тестирования применяют

* PyUnit
* PyTest
* Nose

Об этих инструментах будет описано в следующих статьях.

### Интеграционное тестирование

Интеграцио́нное тестирование или функциональное тестирование предназначено для проверки связи между компонентами, а также взаимодействия с различными частями системы (операционной системой, оборудованием либо связи между различными системами).

#### **Подходы к интеграционному тестированию**

* Снизу вверх (Bottom Up Integration)
* Все низкоуровневые модули, процедуры или функции собираются воедино и затем тестируются. После чего собирается следующий уровень модулей для проведения интеграционного тестирования. Данный подход считается полезным, если все или практически все модули разрабатываемого уровня готовы.
* Сверху вниз (Top Down Integration)
* Вначале тестируются все высокоуровневые модули, и постепенно, один за другим добавляются низкоуровневые. Все модули более низкого уровня симулируются заглушками с аналогичной функциональностью, затем по мере готовности они заменяются реальными активными компонентами.
* Большой взрыв ("Big Bang" Integration)
* Все или практически все разработанные модули собираются вместе в виде законченной системы или ее основной части, и затем проводится интеграционное тестирование. Такой подход очень хорош для сохранения времени. Однако, если тест кейсы и их результаты записаны не верно, то сам процесс интеграции сильно осложнится, что станет преградой для команды тестирования при достижении основной цели интеграционного тестирования

#### **Инструменты**

Для автоматизации интеграционного тестирования применяются системы непрерывной интеграции (Continuous Integration System, CIS). Принцип действия таких систем состоит в следующем:

1. CIS производит мониторинг системы контроля версий;
2. При изменении исходных кодов в репозитории производится обновление локального хранилища;
3. Выполняются необходимые проверки и модульные тесты;
4. Исходные коды компилируются в готовые выполняемые модули;
5. Выполняются тесты интеграционного уровня;
6. Генерируется отчет о тестировании.

Примеры инструментов:

* Bamboo
* Hudson и Jenkins
* CruiseControl
* TeamCity
* BuildBot
* Travis CI
* Team Foundation Server

### Системное тестирование

Основной задачей системного тестирования является проверка как функциональных, так и не функциональных требований в системе в целом. При этом выявляются дефекты:

* Неверное использование ресурсов системы
* Непредусмотренные комбинации данных пользовательского уровня
* Несовместимость с окружением
* Непредусмотренные сценарии использования
* Отсутствующая или неверная функциональность
* Неудобство использования
* И другие

Для минимизации рисков, связанных с особенностями поведения в системы в той или иной среде, во время тестирования рекомендуется использовать окружение максимально приближенное к тому, на которое будет установлен продукт после выдачи.

Стоит выделить два подхода к системному тестированию:

* на базе требований (requirements based). Для каждого требования пишутся тестовые случаи (test cases), проверяющие выполнение данного требования.
* на базе случаев использования (use case based). На основе представления о способах использования продукта создаются случаи использования системы (Use Cases).

По конкретному случаю использования можно определить один или более сценариев. На проверку каждого сценария пишутся тест кейсы (test cases), которые реализуются в виде тестов.

**Выводы:** Проанализировал, разработал и реализовал алгоритм. Проверил на правильность работу отдельных модулей программы

Студент: / Ульянов Никита Анатольевич