Лабораторная работа №1. Основы Node.js

Цель работы:

Теоретическая часть

Node.js – среда выполнения JavaScript вне браузера.

В основе Node.js – виртуальная машина V8. Она была создана компанией Google для браузера Chrome. V8 умеет выполнять JavaScript быстрее и экономнее, чем любая другая.

Node.js используется для создания веб-серверов, самостоятельных приложений, программ для компьютеров и мобильных устройств, а также программирования микроконтроллеров.

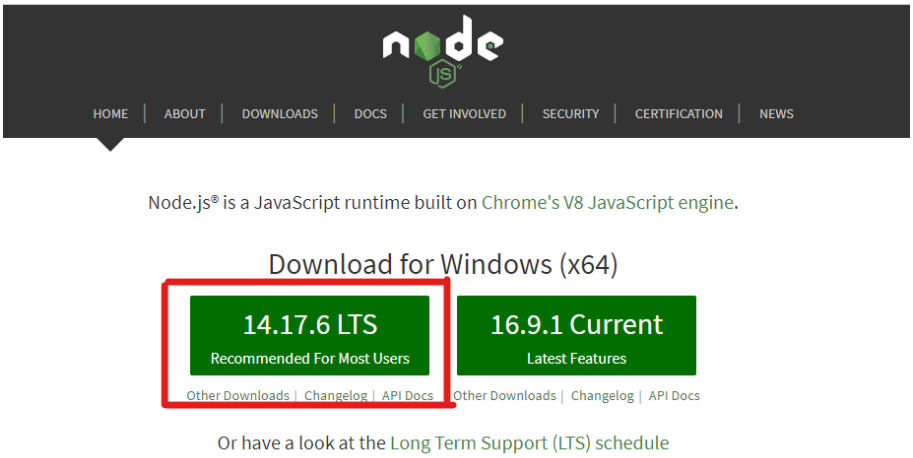
У Node.js только одна платформа, в которой в результате постоянного улучшения появляется поддержка новейших возможностей Javascript. В отличие от браузерного JS, не нужно заботиться о поддержке устаревших браузеров, практически никогда не нужно транспилировать код под старые версии платформы (однако, как и в случае с браузерным JS, перед использованием новейших возможностей языка рекомендуется проверить, что они уже поддерживаются выбранной версией Node.js).

Компании выбирают Node.js за скорость и простоту разработки, возможность использовать один язык при создании фронтенда и бэкенда, скорость работы созданных приложений, кроссплатформенность, экономию ресурсов.

К относительным недостаткам Node.js можно отнести то, что он несколько хуже подходит для решения задач, требующих интенсивных вычислений (хотя с появлением ***worker threads*** ситуация значительно улучшилась).

Устанавливаем Node.js:

Ссылка для скачивания https://nodejs.org/en/



Скачиваем и устанавливаем последнюю LTS версию (Recommended For Most Users)

Проверяем что именно установили. Для этого проверяем версию Node.js.

Открываем, например, Git Bash: клик правой кнопкой по рабочему столу, в контекстном меню выбираем Git Bash Here. Если такого пункта нет, скачайте и установите Git https://git-scm.com/downloads.

Выполняем команду **node -v**

Если отображается версия Node.js, значит, с первым пунктом мы справились и Node.js установили.

Где писать код

В терминале писать не очень удобно. Как и в блокноте. Выбираем привычный VS Code, открываем терминал (вкладка Терминал на панели вверху, пункт Создать терминал), проверяем, что терминал работает. Для этого выполняем команду **node -v**.

Режим REPL

Код можно писать и выполнять прямо в терминале.

Такой режим называется REPL (от англ. **Read-Eval-Print-Loop** – цикл чтение – вычисление – вывод)

Чтобы в него перейти, выполните в терминале команду node

Теперь код можно писать непосредственно в терминале, REPL вычислит введенное выражение и выведет результат. К примеру, если ввести 2 + 2 и нажать Enter, REPL выведет 4.

Также можно делать явный вывод в консоль при помощи уже знакомых нам методов. Так, вы можете написать

```
console.log("Hello, world!");
```

REPL имеет некоторые полезные команды, получить информацию о которых можно, отправив команду ***.help***

Чтобы выйти из режима REPL, отправьте команду ***.exit***(также для более грубого завершения процесса можно применить стандартное для используемого терминала сочетание клавиш наподобие Ctrl + C).

Очистить терминал позволят такие команды, как ***cls*** (для стандартной командной строки Windows и Powershell), ***clear*** (для Bash).

Как запустить файл

Режим REPL используется достаточно редко.
Как правило, при помощи Node.js запускают код, размещенный в файлах.
Создадим файл *test.js* и напишем в нём команду

```
console.log("Hello, world!");
```

Откроем этот файл при помощи VS Code, в терминале выполним команду:
node test.js

Операции ввода/вывода

I/O (input/output) означает ввод/вывод

- **input** - получение информации от сетевых ресурсов, или чтение с диска или файла, или ввод с клавиатуры
- **output** - вывод информации, например, сохранение на диск или запись в файл, или вывод в консоль

Это самые затратные по времени этапы работы программы. Сравните:

Операция	Количество CPU тактов
CPU Registers	3 такта
L1 Cache	8 тактов
L2 Cache	12 тактов
RAM	150 тактов
Disk	30,000,000 тактов
Network	250,000,000 тактов

Блокирующий I/O

Операции input/output происходят синхронно, одна за другой

```
const connection = db.connect();  // подключаемся к базе
 данных
const users = connection.query('SELECT * FROM users');
// делаем запрос
console.log(users); // выводим информацию в консоль
```

Это простой в реализации, но очень затратный по времени вариант: программа ждёт 250 миллионов тактов процессора, пока не произойдёт подключение к базе данных.

Вторая проблема синхронного I/O - он не отказоустойчив. Если программа не сможет подключиться к базе данных, или если в базе данных не найдётся затребованной информации, программа остановит свою работу.

Синхронный или блокирующий I/O в Node.js используется очень редко:

- если необходимо получить данные, без которых работа программы не может начаться. Например, информацию о настройках
- может использоваться в консольных приложениях, у которых только один пользователь

Неблокирующий I/O

Неблокирующий I/O происходит асинхронно

```
db.connect((error, connection) => {
  if (error) throw error;
  connection.query('SELECT * FROM users', users => {
    console.log(users);
  });
});
```

В данном примере программа вызывает **`db.connect()`**, ставит его колбэк в очередь и переходит к выполнению оставшейся синхронной части кода. Когда весь синхронный код выполнен, программа возвращается к выполнению колбэка **`db.connect()`**.

Первым аргументом колбэка, переданного в **`db.connect()`**, является ошибка. Если ошибка имеется, то в нашем примере происходит "проброс исключения", в результате которого эта ошибка либо будет обработана в коде выше, либо вывалится в рантайм и "повалит" приложение. Если ошибки нет - **`error === null`** (в логическом контексте ***false***) - функция переходит к работе с базой данных: выполняет **`connection.query()`** и **`console.log()`**.

В основе работы Node.js лежат **неблокирующий ввод/вывод** и **асинхронность**. Благодаря этому приложения на Node.js работают быстро и могут обрабатывать большое количество клиентских запросов в единицу времени.

Стандартные потоки ввода/вывода

Для ввода и вывода информации (I/O - input/output) в Node.js существуют стандартные потоки ввода и вывода:

- **`process.stdin`** - поток ввода
- **`process.stdout`** - поток вывода
- **`process.stderr`** - поток ошибки как разновидность потока вывода

Например, уже известный нам **`console.log()`** для вывода информации использует **`process.stdout`**.

Стандартный поток вывода

Выведем информацию в консоль при помощи **`process.stdout`**.

```
const { stdout } = process;
stdout.write('Node.js');
```

Метод **`stdout.write()`** принимает в качестве аргумента строку и выводит её в консоль. В отличие от **`console.log()`** он не добавляет автоматический перенос в конце строки. При необходимости перенос строки **`\n`** можно добавить вручную.

Стандартный поток ввода

В файле **test.js** напишем и запустим код:

```
const { stdin, stdout } = process;
stdin.on('data', data => stdout.write(data));
```

При помощи метода **`.on()`** мы подписываемся на событие **`'data'`** объекта **`stdin`**.

Метод **`.on()`** принимает два параметра - название события **`'data'`** и стрелочную функцию-обработчик **`data => stdout.write(data)`**, которая выводит в консоль переданные данные.

Теперь, когда мы вводим в консоль какой-то текст и нажимаем клавишу Enter, **`stdout.write()`** возвращает введённый нами текст.

Метод process.exit()

Остановить выполнение программы можно нажав комбинацию славиш **Ctrl + C** или использовав метод **`process.exit()`**.

Метод **`process.exit()`** при запуске эмитит событие **`'exit'`**, подписавшись на которое мы можем выполнить определенные действия перед завершением программы:

```
process.on('exit', () => stdout.write('Удачи в изучении
Node.js!'));
```

`process.exit()` принимает необязательный аргумент **`exitCode`**, представленный целым числом. По умолчанию данный метод запускается с параметром **`exitCode === 0`**. Такое завершение процесса означает, что программа выполнена

успешно и отработала без ошибок. Завершение процесса с любым другим **exitCode**, что работа программы завершилась ошибкой. Благодаря этому можно передать разные сообщения на выходе в зависимости от того, сработала программа как нужно, или нет.

```
    const { stdin, stdout } = process;

    process.on('exit', code => {
        if (code === 0) {
            stdout.write('Всё в порядке');
        } else {
            stderr.write(`Что-то пошло не так. Программа
завершилась с кодом ${code}`);
        }
    });
```

Аргументы командной строки

В Node.js есть возможность запустить файл с определёнными аргументами командной строки. При запуске файла аргументы передаются после его имени. Например, при запуске:

node test 1 2 3

1, 2, 3 - это аргументы. Как внутри кода получить доступ к переданным при запуске файла аргументам? Для этого используется свойство глобального объекта **process** - **process.argv**

В файле **test.js** напишем код:

console.log(process.argv);

В терминале выполним команду **node test 1 2 3.**

В консоли отображается массив, первые два элемента которого - путь к файлу node.exe и путь к запущенному файлу. Дальше идут переданные аргументы.

Если нужно получить только аргументы, выполним код:

```
    console.log(process.argv.slice(2));
```

Метод **process.argv.slice(2)** возвращает новый массив, который начинается с элемента с индексом "2".

Флаги

Чтобы иметь возможность отправлять аргументы в любом порядке или пропускать какие-то из них, аргументы командной строки можно пометить. Для этого используются флаги - слова или символы, которые указывают, что за ними следует

аргумент командной строки. Перед флагами, как правило, ставят один или два дефиса, чтобы не перепутать их с аргументами. Например,

node test -m Hello

Чтобы получить аргумент с указанным флагом, напишем код:

```
const flagIndex = process.argv.indexOf('-m');
if (flagIndex !== -1) {
  const message = process.argv[index + 1];
  console.log(message);
}
```

Можно этот код преобразовать в функцию, получающую флаг аргумента и возвращающую его значение:

```
function getValue(flag) {
  const flagIndex = process.argv.indexOf(flag);
  return flagIndex !== -1 ? process.argv[flagIndex + 1] : null;
}
const message = getValue('-m');
console.log(message);
```

Практическое применение

На практике в случае, если вы пишете код для работы с аргументами командной строки самостоятельно, необходимо корректно обработать всевозможные ситуации – аргумент может отсутствовать, флаг может быть не передан, либо передан без значения, либо само наличие флага является булевым значением и т.д.

Для повышения удобства работы с аргументами командной строки, а также минимизации вероятности возникновения ошибок удобно использовать готовые решения, такие как minimist, commander, yargs и другие.

Доступ к файловой системе

В отличие от браузерного JavaScript, у Node.js есть доступ к файловой системе.

Например, мы легко можем узнать абсолютный путь к директории, в которой находится наш файл. Для этого откроем файл test.js и напишем в нём код:

`console.log(__dirname);`

Откроем терминал и запустим файл:

node test

В консоль выведется абсолютный путь к директории с файлом **test.js**. Теперь выведем абсолютный путь к файлу. Для этого в файле **test.js** добавим строку

`console.log(__filename);`

Теперь в консоль выводится абсолютный путь к файлу **test.js** вместе с его именем.

Модули

Node.js любой файл воспринимает как модуль. В Node.js на данный момент используются 2 системы модулей: **CommonJS** (модули, используемые в Node.js по умолчанию, появились раньше) и **ECMAScript модули** (реализуют функционал JS, впервые появившийся в спецификации ECMAScript 2015). Они имеют существенные отличия друг от друга. В рамках данных материалов мы используем только CommonJS модули. Глобальных переменных вроде **`__dirname`**, **`__filename`**, **`process`** в Node.js не так много. Остальной функционал реализован в виде подключаемых модулей.

Преимущества использования модулей в Node.js:

- Лучшее структурирование кода – код, разбитый на модули, гораздо легче для понимания, поддержки, тестирования
- Облегчение переиспользования кода – правильно написанный и документированный модуль легко может быть использован в нескольких местах в одном проекте, а также в разных проектах
- Инкапсуляция – содержимое модуля инкапсулировано, т.е. доступно исключительно внутри модуля. Разработчик сам решает, что импортировать в модуль и что экспортировать за пределы модуля
- Благодаря кэшированию модулей их многократный импорт не приводит к дополнительным издержкам производительности

Упрощенно, модули в Node.js можно разделить на 3 типа:

1. Стандартные модули (core modules), которые мы получаем "из коробки", устанавливая Node.js на компьютер. Примеры стандартных модулей:
   - модуль **path**
   - модуль **fs**
   - модуль **os**
   - модуль **http**
   - модуль **events**
2. Модули-пакеты
3. Модули, которые разработчик создаёт самостоятельно

Стандартные модули

Они уже скомпилированы в двоичный код и описаны в документации. Перечень стандартных модулей. Стандартные модули достаточно подключить и можно с ними работать.

Для подключения модуля используется функция **`require()`**

Примеры подключения модулей:

```
const path = require('path');
const fs = require('fs');
const os = require('os');
```

Модули-пакеты (Packages)

К модулям-пакетам относятся папки с кодом, описываемые при помощи находящегося в них файла **package.json**. С модулями-пакетами удобно работать при помощи менеджеров пакетов, таких как **npm** или **yarn**. Если мы хотим использовать уже написанные кем-то модули-пакеты (частый способ использования кода других разработчиков), их нужно установить, затем подключить, затем использовать.

Установка модуля-пакета при помощи **npm** осуществляется командой:

npm install <имя модуля>

Установленные модули добавляются в папку **node_modules**, а информация о них добавляется в файл **package.json**. Кроме того, автоматически создается файл **package.lock.json**, гарантирующий идентичность пакетов у различных пользователей, а также выполняющий ряд других полезных функций. Если удалить папку **node_modules** и выполнить команду **npm install**, папка **node_modules** восстановится вместе со всеми добавленными модулями на основе записей в файле **package.json**. Проекты, написанные на Node.js, добавляются на GitHub без папки **node_modules**, но с файлом **package.json**, а также **package.lock.json**. После скачивания такого проекта необходимо выполнить в терминале команду **npm install**, чтобы восстановить все установленные через **npm** модули.

Модули, которые разработчик создаёт самостоятельно

Создание модуля начинается с создания отдельного js-файла, в котором пишется код. Если модуль должен экспортировать что-либо наружу, это делается при помощи записи экспортируемого значения в качестве свойства специального объекта `module.exports`, либо его перезаписи. Экспортируемые из одних модулей значения мы можем импортировать в других модулях. Как и в случае с другими типами модулей, импорт осуществляется при помощи функции `require()`, только в качестве аргумента функции вместо имени модуля указываем путь к файлу.

Создание Node.js-приложения

Создадим новый проект. Для этого создадим папку проекта, откроем её в VS Code и в терминале выполним команду

npm init -y

Параметр **-y** (Yes) означает, что мы соглашаемся со всеми настройками проекта по умолчанию.

В папке проекта появляется файл **package.json**, который описывает созданное приложение.

Streams (потоки, стримы)

Если нам нужно работать с достаточно большим объёмом данных, работать с ним целиком означает загрузить оперативную память и остановить работу программы на все время операции.

Вместо этого считывание и запись данных можно осуществлять по частям, небольшими фрагментами - чанками (**chunk**). Это позволяет работать с очень большими объемами данных, не повышая объем потребляемой памяти пропорционально их размеру.

Стримы используют интерфейс работы с событиями, унаследованный от **`EventEmitter`**.

Помимо использования готовых стримов, мы можем создавать свои собственные стримы, отнаследовавшись от базовых классов и реализовав некоторые обязательные методы.

Для работы с потоковыми данными в Node.js есть абстрактный интерфейс – **streams** (потоки, стримы).

В Node.js есть 4 основных вида потоков:

Readable – поток чтения, используется для чтения данных;

Writable – поток записи, используется для записи данных;

Duplex – поток, который может быть использован как для чтения, так и для записи данных;

Transform – разновидность Duplex, используемая для преобразования данных.

Поток чтения (Readable stream)

Поток чтения, как понятно из его названия, используется для чтения данных. Источником данных может быть что угодно: ввод пользователя, файл, входящий запрос пользователя при обработке на сервере, другой поток, асинхронный итератор и т.д.

Создадим программу, которая будет читать достаточно большой файл и выводить его содержимое в консоль. Для этого используем модуль **fs**, но вместо метода **`readFile()`** используем метод **`createReadStream()`**, параметром которого укажем название файла **source.txt**, из которого будем читать информацию. Так как файл лежит в той же директории, что и файл с кодом, путь к файлу прописывать не обязательно.

```
    const fs = require('fs');
    const readableStream = fs.createReadStream('source.txt');
```

У потока чтения есть событие **data**, которое генерируется, когда стрим прочитал порцию данных и готов отдать ее потребителю этих данных. При наступлении этого события выведем поступившую часть данных в консоль:

```
    const fs = require('fs');
    const readableStream = fs.createReadStream('source.txt');
readableStream.on('data', chunk => console.log(chunk));
```

В консоли вместо текста объекты **Buffer**. Мы можем решить эту проблему при помощи метода **`data.toString()`**, но преобразовать **Buffer** в строку можно и другим способом, указав вторым параметром метода **`createReadStream()`** кодировку **`'utf-8'`**.

Как убедиться, что данные приходят по частям?

Выведем в консоль не сами данные, а длину каждой пришедшей части данных:

```
    const fs = require('fs');
    const readableStream = fs.createReadStream('source.txt');
readableStream.on('data', chunk => console.log(chunk.length));
```

Если файл с данными достаточно большой, видно, что приходят они частями (чанками) размером 64кБ.

Чтобы все эти части собрать вместе, определим переменную **data**. Её значением укажем пустую строку. Каждую пришедшую часть данных будем присоединять к **data**.

```
    const fs = require('fs');
    const stream = fs.createReadStream('source.txt', 'utf-8');
    let data = '';
    stream.on('data', chunk => data += chunk);
```

Так как мы имеем дело с потоком данных, нам нужно знать когда поток завершится. Для этого у стрима есть событие **`'end'`**. Это событие срабатывает, когда все данные уже переданы.

При наступлении события **'end'** выведем в консоль сообщение и длину полученных данных:

```
    const fs = require('fs');

    const stream = fs.createReadStream('source.txt', 'utf-8');

    let data = '';

    stream.on('data', chunk => data += chunk);
    stream.on('end', () => console.log('End', data));
```

Обработаем возможную ошибку. При возникновении ошибки будет сгенерировано событие **error**. При наступлении ошибки выведем в консоль сообщение и текст ошибки. Чтобы вызвать ошибку, укажем несуществующее имя файла:

```
    const fs = require('fs');

    const stream = fs.createReadStream('source2.txt', 'utf-8');

    let data = '';

    stream.on('data', chunk => data += chunk);
    stream.on('end', () => console.log('End', data));
    stream.on('error', error => console.log('Error',
error.message));
```

Поток записи (Writable stream)

Поток записи, является противоположностью потока чтения. Он используется для записи данных. Записывать данные можно, к примеру: в стандартный поток вывода, файл, **response** при обработке на сервере, другой поток и т.д.

Если мы читаем данные по частям, логично записывать их тоже по частям.

Для этого создадим поток записи **output:**

```
const fs = require('fs');
const output = fs.createWriteStream('destination.txt');
```

Если не создать файл, который указан в качестве пункта назначения наших данных, **destination.txt**, перед началом записи он будет создан автоматически.

Поток чтения назовём **input** и каждую часть данных, которую он отдает, будем записывать в файл при помощи метода **output.write():**

Сравните полученный код потока записи с кодом потока чтения - они создаются и используются сходным образом.

```
    const fs = require('fs');

    const input = fs.createReadStream('source.txt', 'utf-8');
    const output = fs.createWriteStream('destination.txt');

    input.on('data', chunk => output.write(chunk));
    input.on('error', error => console.log('Error',
error.message));
```

Объединение потоков чтения-записи

Код выше можно сделать ещё проще и лучше:

```
    const fs = require('fs');

    const input = fs.createReadStream('source.txt', 'utf-8');
    const output = fs.createWriteStream('destination.txt');

    input.pipe(output);
```

Несмотря на то, что кода стало меньше, работает он точно так же, как прежде.

Метод **pipe()**, имеющийся у каждого потока, можно использовать для объединения одних потоков с другими. Такие цепочки могут объединять несколько потоков.

Эту особенность метода **pipe()** используют, например, для сжатия файлов.

Есть довольно удобный способ объединения нескольких потоков, позволяющий использовать один обработчик ошибок – функция **pipeline**:

```
const fs = require('fs');
const zlib = require('zlib');
const { pipeline } = require('stream');

const input = fs.createReadStream('source.txt', 'utf-8');
const output = fs.createWriteStream('destination.txt.gz');
const gzip = zlib.createGzip();

pipeline(
    input,
    gzip,
    output,
    err => {
        if (err) {
            // обрабатываем ошибки
        }
    }
);
```

Практическая часть

Внедрите инструмент CLI для решения представленных задач.
Инструмент CLI должен принимать 2 опции (короткий псевдоним и полное имя):

1. **-i**, **--input**: для входных файлов
2. **-o**, **--output**: для выходных файлов

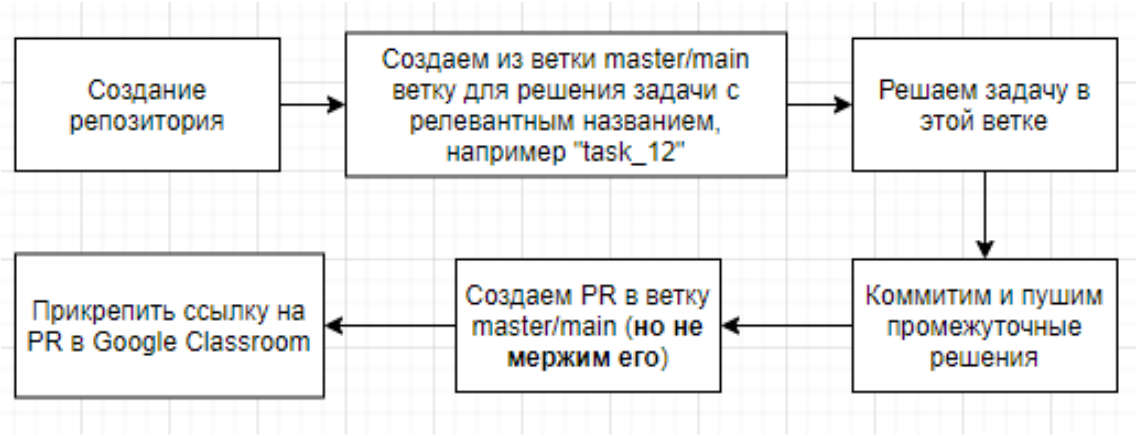
Основная часть:

1. В ***README.md*** должно быть описано, как можно запустить программу из командной строки, описаны аргументы, которые можно передать приложению.
2. Для аргументов командной строки можно использовать один из
   - https://www.npmjs.com/package/commander
   - https://www.npmjs.com/package/minimist или любой другой модуль.
3. Предусмотреть возможность ввода входных данных через консоль используя **`stdin`**, если не задан входной файл.
4. Если входные данные вводится из консоли, то программа не должна завершаться после выполнения задачи, т.е. должна быть возможность ввести еще данные.
5. Предусмотреть возможность вывода данных через консоль используя **`stdout`**, если не задан выходной файл.
6. Если входной и/или выходной файл задан, но не существует или вы не можете его прочить (например, из-за разрешений или это каталог) – должна вывестись соответствующая ошибка в консоль (используя **`stderr`**) и процесс завершается с кодом, отличным от 0.
7. Если переданные параметры в порядке, вывод (файл или **`stdout`**) должен содержать решение представленной задачи.
8. Использование **`streams`** для чтения, записи и преобразования **обязательно**.
9. Кодовая база не находится в одном файле, а разделена на файлы в соответствии с выполняемыми задачами

Подсказка: в качестве предлагаемого решения для повышения надежности кода и повышения эффективности использования памяти рассмотрите возможность использования метода **`pipeline`**. Структура может быть следующая:

```
pipeline(
  input_stream, // input file stream or stdin stream
  transform_stream, // Transform stream
  output_stream // output file stream or stdout stream
)
.then(success and error callbacks)
```

Схемы выполнения задачи



Примеры реализации и описания приложений

[Пример реализации](#) (Калинин М.А.) [Пример описание](#)
[Пример реализации](#) (Гардейчик С.М.) [Пример описания](#)

Передача данных из CLI в функцию решения задачи

Считывание данных с файла:
Флаг -i, --input (необязательный): путь к входному файлу

Запись данных в файл:
Флаг -o, --output (необязательный): путь к выходному файлу

Примеры обработки входных данных из консоли (из файла аналогично):

Функция ожидает один параметр (строка):

```
node task_cli -o "./output.txt"
bash > text
```

```
// ...
const payload = process.stdin; // -> "text"
// Вызов функции с переданными параметрами
const result = task_fn(payload);
// ...
```

Функция ожидает два и более параметра (строки):

```
node task_cli -o "./output.txt"
bash > text1:text2
```

```
// ...
const payload = process.stdin; // -> "text1:text2"
const [arg1, arg2] = payload.split(":");
// -> [text1, text2]
// Вызов функции с переданными параметрами
const result = task_fn(arg1, arg2);
// ...
```

Функция ожидает один параметр (массив):

```
node task_cli -o "./output.txt"
bash > [1, 2, 3]
```

```
// ...
const payload = process.stdin; // -> "[1, 2, 3]"
const arr = JSON.parse(args); // -> [1, 2, 3]
// вызов функции с переданными параметрами
const result = task_fn(arr);
// ...
```

Функция ожидает два и более параметр (массив):

```
node task_cli -o "./output.txt" -a "[1,2,3]:[4,5,6]"
```

```
// ...
const payload = process.stdin; // -> "[1, 2, 3]:[4, 5, 6]"
const [arg1, arg2] = payload.split(":");
const arr1 = JSON.parse(arg1); // -> [1, 2, 3]
const arr2 = JSON.parse(arg2); // -> [4, 5, 6]
// вызов функции с переданными параметрами
const result = task_fn(arr1, arr2);
// ...
```

!!! Можно придумать свою схему передачи данных.

Примеры CLI команд:

Команда:

```
node task_cli -i "./input.txt" -o "./output.txt"
```

Данные считываются из файла, преобразуются, результат записывается в файл.

Команда:

```
node task_cli -i "./input.txt"
```

Данные считываются из файла, преобразуются, результат выводится в консоль.

Команда:

```
node task_cli -o "./output.txt"
```

Данные считываются из консоли, преобразуются, результат записывается в файл.

Команда:

```
node task_cli
```

Данные считываются из консоли, преобразуются, результат выводится в консоль.

Задачи на преобразование строк

Вариант 1

Реализовать функцию, которая принимает массив из 10 целых чисел (от 0 до 9), который возвращает строку этих чисел в форме номера телефона.

```
// Пример
createPhoneNumber([1, 2, 3, 4, 5, 6, 7, 8, 9, 0])
// => returns "(123) 456-7890"
```

```
createPhoneNumber([1, 2, 3, 4, 5, 6, 7, 8, 9, 0])
// -> "(123) 456-7890"

createPhoneNumber([1, 1, 1, 1, 1, 1, 1, 1, 1, 1])
// -> "(111) 111-1111"

createPhoneNumber([1, 2, 3, 4, 5, 6, 7, 8, 9, 0])
// -> "(123) 456-7890"
```

Вариант 2

Вам дается строка слов (x), для каждого слова в строке вам нужно вывернуть слово «наизнанку». Внутренние буквы будут двигаться наружу, а внешние буквы двигаться к центру. Если слово четной длины, все буквы переместятся. Если длина нечетная, то «средняя» буква слова останется на месте.

```
// Пример
insideOut('taxi') // -> 'atix'
insideOut('taxis') // -> 'atxsi'
```

```
// Дополнительные тесты
insideOut('man i need a taxi up to ubud')
// -> 'man i ende a atix up to budu'

insideOut('what time are we climbing up the volcano')
// -> 'hwta item are we milcgnib up the lovcona'

insideOut('take me to semynak')
// -> 'atek me to mesykan'
```

Вариант 3

Реализовать функцию, которая принимает массив строк и удаляет все последовательные повторяющиеся буквы из каждой строки в массиве. Строки должны быть только строчными, без пробелов.

```
// Пример
dup(["abracadabra","allottee","assessee"])
// -> ["abracadabra","alote","asese"]

dup(["kelless","keenness"]) // -> ["keles","kenes"]
```

```
// Дополнительные тесты
dup([
    "ccooddddddewwwaaaaarrrrsssss",
    "piccaninny",
    "hubbubbubboo"
])
// -> ['codewars','picaniny','hubububo']

dup(["abracadabra","allottee","assessee"])
// -> ['abracadabra','alote','asese']

dup(["kelless","keenness"]) // -> ['keles','kenes']

dup(["adanac","soonness","toolless","ppellee"])
// -> ['adanac','sones','toles','pele']

dup(["callalloo","feelless","heelless"])
// -> ['calalo','feles','heles']

dup(["kelless","voorraaddoosspullen","achcha"])
// -> ['keles','voradospulen','achcha']
```

Вариант 4

Реализовать функцию, которая преобразует строку в новую строку, где каждый символ в новой строке равен «(», если этот символ появляется только один раз в исходной строке, или «)», если этот символ встречается более одного раза в исходной строке. Игнорируйте использование заглавных букв при определении дубликата символа.

```
// Пример
duplicateEncode("din") // -> "((("
duplicateEncode("recede") // -> "()()()"
duplicateEncode("Success") // -> ")())())"
duplicateEncode("(( @") // -> "))(("
```

Вариант 5

Реализуйте функцию, которая получает две строки и возвращает n, где n равно количеству символов, на которое нужно сдвинуть первую строку вперед, чтобы соответствовать второй. Проверка должна быть чувствительна к регистру.

Если вторая строка не является допустимым поворотом первой строки, метод возвращает -1.

```
// Пример
shiftedDiff("coffee", "eecoff") // -> 2
shiftedDiff("eecoff", "coffee") // -> 4
shiftedDiff("moose", "Moose") // -> -1
```

```
// Дополнительные тесты
shiftedDiff("eecoff","coffee") // -> 4
shiftedDiff("Moose","moose") // -> -1
shiftedDiff("isn't","'tisn") // -> 2
shiftedDiff("Esham","Esham") // -> 0
shiftedDiff(" "," ") // -> 0
shiftedDiff("hoop","pooh") // -> -1
shiftedDiff("  "," ") // -> -1
```

Вариант 6

Создайте функцию, которая принимает целое число n и возвращает формулу для $(a+b)^2$ в виде строки.

```
// Пример
formula(0)  // -> "1"
formula(1)  // -> "a+b"
formula(2)  // -> "a^2+2ab+b^2"
formula(-2) // -> "1/(a^2+2ab+b^2)"
formula(3)  // -> "a^3+3a^2b+3ab^2+b^3"
formula(5)  // -> "a^5+5a^4b+10a^3b^2+10a^2b^3+5ab^4+b^5"
```

Вариант 7

Реализовать функцию, которая принимает строку английских цифр, «склеенных» вместе, например: «zeronineoneoneeighttwoseventhreesixfourtwofive».

Необходимо разбить строку на отдельные цифры: «zero nine one one eight two seven three six four two five».

```
// Пример
uncollapse("eightsix") // -> "eight six"
```

```
// Дополнительные тесты
uncollapse("three") // -> "three"
uncollapse("eightsix") // -> "eight six"
uncollapse("fivefourseven") // -> "five four seven"
uncollapse("ninethreesixthree")
// -> "nine three six three"

uncollapse("foursixeighttwofive")
// -> "four six eight two five"
```

Вариант 8

Считываем переменную n,

если n - целое число, возвращать строку с дефисом '-' перед каждым нечетным целым числом и после него, но не начинать и не заканчивать строку знаком тире. Если n отрицательное, знак минус следует удалить.

Если n не является целым числом, вернуть пустое значение.

```
// Пример
dashatize(274)
 //-> '2-7-4'

dashatize(6815)
 //-> '68-1-5'
```

```
// Дополнительные тесты
dashatize(NaN)
  // -> Should return NaN

dashatize(0)
  // -> Should return 0

dashatize(-1)
  // -> Should return 1

dashatize(-28369)
  // -> Should return 28-3-6-9
```

Вариант 9

Отсортируйте данный массив строк в алфавитном порядке, без учета регистра. Например:

```
// пример
sortme(["Hello", "there", "I'm", "fine"])
 //-->  ["fine", "Hello", "I'm", "there"]

sortme(["C", "d", "a", "B"])
 //-->  ["a", "B", "C", "d"]

sortme(["CodeWars"])
 //--> ["CodeWars"]
```

Вариант 10

Здесь вам нужно проделать некоторые математические операции с «грязной строкой».

Ввод: строка, состоящая из двух положительных чисел (двойных) и ровно одного оператора, например +, -, * или / всегда между этими числами. Строка грязная, а это значит, что внутри разные символы, а не только числа и оператор. Вы должны объединить все цифры слева и справа, возможно, с "." внутри (удваивается), и для вычисления результата, который должен быть округлен до целого числа и преобразован в строку в конце.

```
// пример
strictEqual("gdfgdf234dg54gf*23oP42")
  // "54929268" (because 23454*2342=54929268)
```

```
// Дополнительные тесты
strictEqual(";$%§fsdfsd235??df/sdfgf5gh.000kk0000")
 // -> Should return "47"

strictEqual("sdfsd23454sdf*2342")
 // -> Should return "54929268"

strictEqual("fsdfsd235???34.4554s4234df-sdfgf2g3h4j442")
 // -> Should return "-210908"
```

Задачи на преобразование массивов

Вариант 1

Реализовать функцию, которая вычитает один список (массив) из другого и возвращает результат. Функция должна удалить все значения из списка A, которые присутствуют в списке B, сохраняя их порядок.

```
// Примеры
arrayDiff([1,2], [1]) // -> [2]
arrayDiff([1,2,2,2,3], [2]) // -> [1,3]
```

```
// Дополнительные тесты
arrayDiff([], [4,5])) // -> []
arrayDiff([3,4], [3])) // -> [4]
arrayDiff([1,8,2], [])) // -> [1,8,2]
arrayDiff([1,2,3], [1,2])) // -> [3]
```

Вариант 2

Дан треугольник из последовательных нечетных чисел:

```
                1
            3       5
        7       9       11
    13      15      17      19
21      23      25      27      29
            ..........
```

Найти строку треугольника, зная его индекс (индексация строк начинается с 1), например:

```
// Пример
oddRow(1) // -> [1]
oddRow(2) // -> [3, 5]
oddRow(3) // -> [7, 9, 11]
```

Примечание: код должен быть оптимизирован для обработки больших данных.

```
// Дополнительные тесты
oddRow(13)
// -> [157, 159, 161, 163, 165, 167, 169, 171, 173, 175, 177, 179, 181]

oddRow(19)
// -> [343, 345, 347, 349, 351, 353, 355, 357, 359, 361, 363, 365, 367, 369, 371, 373, 375, 377, 379]

oddRow(41)
// -> [1641, 1643, 1645, 1647, 1649, 1651, 1653, 1655, 1657, 1659, 1661, 1663, 1665, 1667, 1669, 1671, 1673, 1675, 1677, …]
```

Вариант 3

Реализовать функцию, которая отсортирует нечетные числа в порядке возрастания, оставив четные числа на своих исходных позициях.

```
// Пример
sortArray([7, 1]) // -> [1, 7]
sortArray([5, 8, 6, 3, 4]) // -> [3, 8, 6, 5, 4]
sortArray([9, 8, 7, 6, 5, 4, 3, 2, 1, 0])
// -> [1, 8, 3, 6, 5, 4, 7, 2, 9, 0]
```

```
// Дополнительные тесты
sortArray([5, 3, 2, 8, 1, 4]) // -> [1, 3, 2, 8, 5, 4]
sortArray([5, 3, 1, 8, 0]) // -> [1, 3, 5, 8, 0]
sortArray([]) // -> []
```

Вариант 4

Реализуйте функцию, которая создает таблицу умножения NxN размера, указанного в параметре.

```
// Пример
multiplicationTable(3) // -> [[1,2,3],[2,4,6],[3,6,9]]
```

Вариант 5

Учитывая отсортированный массив различных целых чисел, напишите функцию, которая возвращает наименьший индекс, для которого:

```
массив[index] == index
```

Верните -1, если такого индекса нет. Алгоритм должен быть очень производительным.

```
// Пример
indexEqualsValue([-8,0,2,5]) // -> 2 (array[2] === 2)
indexEqualsValue([-1,0,3,6]) // -> -1 (array[i] !== i)
```

```
// Дополнительные тесты
indexEqualsValue([-8,0,2,5]) // -> 2
indexEqualsValue([-1,0,3,6]) // -> -1
indexEqualsValue([-3,0,1,3,10]) // -> 3
indexEqualsValue([-5, 1, 2, 3, 4, 5, 7, 10, 15]) // -> 1
indexEqualsValue([9,10,11,12,13,14]) // -> -1
indexEqualsValue([0]) // -> 0
```

Вариант 6

Учитывая массив из n целых чисел, найдите минимальное число для вставки в список, чтобы сумма всех элементов списка была равна ближайшему простому числу.

Примечания

1. Размер списка не менее 2.
2. Элементы списка только положительные (n > 0) и целые числа.
3. Возможно повторение чисел в списке.
4. Сумма нового списка должна равняться ближайшему простому числу.

```
// Пример
minimumNumber([3, 1, 2]) // -> 1

// Поскольку сумма элементов списка равна (6), минимальное число, которое нужно вставить для преобразования суммы в простое число, равно (1), что сделает сумму списка равной ближайшему простому числу (7).
```

```
// Дополнительные тесты
minimumNumber([3,1,2]) // -> 1
minimumNumber([5,2]) // -> 0
minimumNumber([1,1,1]) // -> 0
minimumNumber([2,12,8,4,6]) // -> 5
minimumNumber([50,39,49,6,17,28]) // -> 2
```

Вариант 7

Напишите функцию, которая принимает две квадратные матрицы (N x N двумерных массивов) и возвращает их сумму. Обе матрицы, передаваемые в функцию, будут иметь размер N x N (квадрат) и содержать только целые числа.

Как сложить две матрицы:

Возьмите каждую ячейку [n][m] из первой матрицы и добавьте ее с той же ячейкой [n][m] из второй матрицы. Это будет ячейка [n][m] матрицы решения.

Визуализация:

```
|1 2 3|     |2 2 1|     |1+2 2+2 3+1|     |3 4 4|
|3 2 1|  +  |3 2 3|  =  |3+3 2+2 1+3|  =  |6 4 4|
|1 1 1|     |1 1 3|     |1+1 1+1 1+3|     |2 2 4|
```

```
// Пример
matrixAddition(
    [[1, 2], [1, 2]],
    [[2, 3], [2, 3]]
)
// -> [[3, 5], [3, 5]]
```

```
// Дополнительный тест
matrixAddition(
    [[1, 2, 3], [3, 2, 1], [1, 1, 1]],
    [[2, 2, 1], [3, 2, 3], [1, 1, 3]]
)
// -> [[3, 4, 4], [6, 4, 4], [2, 2, 4]]

matrixAddition( [[1]], [[2]] ) // -> [[3]]
```

Вариант 8

Инверсия массива указывает, насколько далеко массив от сортировки. Инверсия массива – это пара элементов, которые расположены «вне своего естественного порядка».

```
// пример
strictEqual([1, 2, 3, 4])
 // ->  0 inversions

strictEqual([1, 3, 2, 4])
// ->  1 inversion: 2 and 3

strictEqual([4, 1, 2, 3])
// ->  3 inversions: 4 and 1, 4 and 2, 4 and 3
```

Цель:

Цель состоит в том, чтобы придумать функцию, которая может вычислять инверсии для любого произвольного массива.

```
// Дополнительный тест
strictEqual(([]) // -> 0

strictEqual( [6,5,4,3,2,1] ) // -> 15
```

Вариант 9

Формулу расстояния можно использовать для определения расстояния между двумя точками. Что, если бы мы пытались пройти из точки А в точку Б, но на пути были здания? Нам понадобится какая-то другая формула .. но какая?

Манхэттенская метрика

Манхэттенское расстояние – это расстояние между двумя точками в сетке (например, в сетке уличной географии района Нью-Йорка на Манхэттене), рассчитанное только путем взятия вертикального и/или горизонтального пути.

Завершите функцию, которая принимает две точки и возвращает Манхэттенское расстояние между двумя точками.

Точки представляют собой массивы или кортежи, содержащие координаты **x** и **y** в сетке. Вы можете представить **x** как строку в сетке, а **y** как столбец.

```
// пример
manhattanDistance( [1, 1], [1, 1] )
 // => returns 0

manhattanDistance( [5, 4], [3, 2] )
 // => returns 4

manhattanDistance( [1, 1], [0, 3] )
 // => returns 3
```

```
 // Дополнительные тесты
 manhattanDistance([1,1],[1,1])
   // -> Should return 0

 manhattanDistance([5,4],[3,2])
   // -> Should return 4

 manhattanDistance([1,1],[0,3])
   // -> Should return 3
```

Вариант 10

Цель состоит в том, чтобы вернуть все пары целых чисел из заданного массива целых чисел, которые имеют разность 2.

Массив результатов следует отсортировать в порядке возрастания значений.

Предположим, что в массиве нет повторяющихся целых чисел. Порядок целых чисел во входном массиве не имеет значения.

```
// пример
twosDifference([1, 2, 3, 4])
 // should return [[1, 3], [2, 4]]

twosDifference([4, 1, 2, 3])
 //-->  should also return [[1, 3], [2, 4]]

twosDifference([1, 23, 3, 4, 7])
 //--> should return [[1, 3]]
```

```
// Дополнительные тесты
 twosDifference([1,2,3,4])
  // -> Should return [[1,3],[2,4]]
 twosDifference([1,3,4,6])
  // -> Should return [[1,3],[4,6]]
```