|  |  |
| --- | --- |
| Untitled-1.jpg | **Coding Dojo Academy**  Algorithms I  Feb 5 - Mar 2 M/W/F @ 9:00 am PST  Syllabus |

*Algorithms I is Coding Dojo Academy’s first live lecture course covering Algorithms, from the foundational topics of variables, conditionals, loops and functions, to algorithms related to strings, arrays and linked lists. This course is designed to use the JavaScript language for Algorithm practice. We use JavaScript because it’s easy to setup and get started, and it’s one of the most utilized programming language on the web. The below links are essential tools utilized during this course, all class communication is handled through the* [*Slack*](https://join.slack.com/t/codingdojo-algorithms/shared_invite/enQtMzEwMDQ2NDM4Nzg2LTNlMDc2NTBhYjcyNWZhMTk0MmQxMTM3ODYxYTYxNDYyZDZjN2RkOTM5OGMwNjRmNjM5ZGRlMjUyZjg2MjJiMjA) *channel, be sure to join your classmates in chat, where you may ask and answer questions.*

*Be Humble,*

*Be Present*

*Be on time, be attentive and responsive*

|  |  |
| --- | --- |
| ***Instructor:*** *Speros Misirlakis*  *Head of Curriculum and Dojo Academy*  ***Slack:****[@Speros - Coding Dojo](https://codingdojo-algorithms.slack.com/messages/D9530R7BR/)*  ***Email:***[*speros@codingdojo.com*](mailto:speros@codingdojo.com)  [*LinkedIn*](https://www.linkedin.com/in/speros-misirlakis-38474328/) | [*Github Link*](https://github.com/keephopealive) | ***Moderator****: Jason Franz*  *Online Instructor*  ***Slack****:* [*@Jason - Coding Dojo*](https://codingdojo-algorithms.slack.com/messages/D942XA1MY/) |

* [**Algorithm App** *(click this link to visit our platform)*](http://algorithm.codingdojo.com/)**:***Algorithm App is Coding Dojo’s algorithm platform designed to help new developers learn the very fundamentals of algorithms using JavaScript.*
* [***Zoom*** *(click this link to download Zoom)*](https://support.zoom.us/hc/en-us/articles/201362233-Where-Do-I-Download-The-Latest-Version-): *Zoom is the software required for Coding Dojo Academy’s live lectures.*
* [**Webinar** *(click this link to join our live lecture)*](https://codingdojo.zoom.us/j/528244900?pwd=mhv9A0IpinY)*: Use this link to join live lectures at the scheduled date/times.*
* [**Slack** *(click this link to join our chatroom)*](https://join.slack.com/t/codingdojo-algorithms/shared_invite/enQtMzEwMDQ2NDM4Nzg2LTNlMDc2NTBhYjcyNWZhMTk0MmQxMTM3ODYxYTYxNDYyZDZjN2RkOTM5OGMwNjRmNjM5ZGRlMjUyZjg2MjJiMjA)***:*** *Slack is the chatroom service used for Coding Dojo Academy’s courses and the primary method of communication.*
  + [*General Channel Link*](https://codingdojo-algorithms.slack.com/messages/C93TURWJF/) *- General Q&A, anything related to the academy and algorithms.*
  + [*Basic 13 Channel Link*](https://codingdojo-algorithms.slack.com/messages/C945W6UEP/) *- In depth discussions related to the basic 13 algorithms.*
* [**JS Bin** (click this link to practice Algorithms with JavaScript)](https://jsbin.com/?js,console): *JS Bin is a JavaScript Playground tool which helps us experiment with javascript code on the browser, without the need of creating our own local file, and launching that file on our browser.*
* [**Homework Submission** *(click this link to view our class G-Drive)*](https://drive.google.com/open?id=1uDX_f7h42tjdPuVhziCJfgt8P4y6HZ_D): *We will be using this* [*Google Drive*](https://drive.google.com/open?id=1uDX_f7h42tjdPuVhziCJfgt8P4y6HZ_D) *to upload homework, to receive an invite to the G-Drive please join the* [*Slack*](https://join.slack.com/t/codingdojo-algorithms/shared_invite/enQtMzEwMDQ2NDM4Nzg2LTNlMDc2NTBhYjcyNWZhMTk0MmQxMTM3ODYxYTYxNDYyZDZjN2RkOTM5OGMwNjRmNjM5ZGRlMjUyZjg2MjJiMjA) *channel and private message* [*@Speros*](https://codingdojo-algorithms.slack.com/messages/D9530R7BR/team/U93TURHK5/) *your gmail address. Once you have access, please create a folder with your full first name and initial last name:*

*e.g. Tom Jones = Tom J*

*e.g. Diana Mills = Diana M*

*Please be respectful and keep the folder pattern as shown above. Do not use usernames, please use your real first name and real last name initial. Use your own folder to upload the homework files into.*

*Warning: Any malicious use of the google drive will get you ejected from the Academy programs and blacklisted from all Coding Dojo services, indefinitely.*

**Week 1** - Fundamentals

**(2/5/18) Monday**

*Lecture Recording:* [*https://vimeo.com/254394409/0ccb9f41d6*](https://vimeo.com/254394409/0ccb9f41d6)

* Course Overview
* Run/Test JS Code
* Conditionals, Loops, Functions
* Algorithm App
* Extra Resources
* Assignment: [**Algorithm App**](http://algorithm.codingdojo.com/) *(Complete Lessons, Challenges and Assessment) This assignment will not require submission to your G-Drive’s personal folder.*

**(2/7/18) Wednesday**

*Lecture Recording:* [*https://vimeo.com/254748701/e00e6b980c*](https://vimeo.com/254748701/e00e6b980c)

* Recap Algorithm App & Q&A
* [**Basic 13**](#ecrlxm6cj8an) Introduction / Review

**(2/9/18) Friday**

*Lecture Recording:* [*https://vimeo.com/255116985/0c5c8c11f8*](https://vimeo.com/255116985/0c5c8c11f8)

* Open Q&A
* Review Basic 13 - Conceptual questions will be prioritized before demos.
* Assignment (due Monday 2/12 9am PST): [**Basic 13**](#ecrlxm6cj8an)*(Complete all 13 algorithms in under 26 minutes, practice by solving the algorithms over and over until you reach the time limit)*

**Week 2** - Arrays

**(2/12/18) Monday**

*Lecture Recording:* [*https://vimeo.com/255435200/40acdde6f6*](https://vimeo.com/255435200/40acdde6f6)

* Basic 13 Q&A / Recap (10 min)
* [Arrays Overview](#qicrz7ex0z8w) (20 min)
* [Arrays Challenges](#ayrnfdrcb2s) Introduced to Monday’s algorithm challenges
* Open Q&A on array manipulation.
* Built-in methods allowed to be used: .**push**(), .**pop**(), do not use built-in methods such as .~~shuffle~~, .~~splice~~, .~~split~~
* Assignment (due Sunday 2/18 11:59 pm local): **PushFront**, **PopFront**, **InsertAt**, **RemoveAt** (found in the [Arrays Basic Challenges](#ayrnfdrcb2s), turn in a text file or google doc file within your personal G-Drive folder)
* *Optional Assignment: Second-to-Last, Nth-to-Last, Second-Largest, Nth-Largest*

**(2/14/18) Wednesday**

*Lecture Recording:* [*https://vimeo.com/255783281/40d3b1a7a1*](https://vimeo.com/255783281/40d3b1a7a1)

* Recap Monday’s algorithms (Q&A and/or demo).
* Introduction to Wednesday algorithms found [here](#bsetivjosxje).
* Build-in methods allowed to be used: .**push**(), .**pop**()
* Assignment (due Sunday 2/18 11:59 pm local): **arrConcat**, **fasterFactorial**, **shuffle** (found in [Arrays Intermediate Challenges](#bsetivjosxje), turn in a text file or google doc file within your personal G-Drive folder)

**(2/16/18) Friday**

*Lecture Recording:* [*https://vimeo.com/256176353/79fe665bb7*](https://vimeo.com/256176353/79fe665bb7)

* Recap on Wednesday’s algorithms (Q&A and/or demo).
* Fundamental Knowledge Check
* Introduction to Friday algorithms TBD.
* Build-in methods allowed to be used: .**push**(), .**pop**()
* OPTIONAL: **smarterSum**, **fabulousFibonacci**, **trickyTribonacci** (found in [Arrays Intermediate 2 Challenges](#16b37m6r8kb7))

**(2/17/18) Saturday** - Q&A Discussion

*Lecture Recording:* [*https://vimeo.com/256602451/91cd21323e*](https://vimeo.com/256602451/91cd21323e)

**Week 3** - Strings

**(2/19/18) Monday**

*Lecture Recording:* [*https://vimeo.com/256602555/42b1a4c65a*](https://vimeo.com/256602555/42b1a4c65a)

* Recap/Open Q&A for array algorithms (10 min)
* Introduction to String algorithms (15 min)
* Assignment (due Sunday 2/25 11:59 pm local): **Arrs2Map, InvertHash, myJoin, mySplit, ReverseString** ([link to challenges](#hrobw08xr9tb))

**(2/21/18) Wednesday**

*Lecture Recording:* [*https://vimeo.com/256832593/8cd3dbb1be*](https://vimeo.com/256832593/8cd3dbb1be)

* Mini discussion for fasterFactorial via Slack request.
* Recap/Open Q&A for Monday algorithms (10 min)
* Discussion over String algorithms
* Assignment (due Sunday 2/25 11:59 pm local): **removeBlanks, getStringDigits, acronyms** ([link to challenges](#8h5zd08ttmz4))

**(2/23/18) Friday**

*Lecture Recording:* [*https://vimeo.com/257183316/b1e8d825b4*](https://vimeo.com/257183316/b1e8d825b4)

* Recap/Open Q&A for Wednesday’s Algorithms (15min)
* Discussion over String algorithms
* OPTIONAL: **Parens Valid**, **Braces Valid**, **Is Palindrome**, **Longest Palindrome** ([link to challenges](#s4l4o1a3lj7o))

**Week 4** - Linked Lists (Bonus: Recursion)

**(2/26/18) Monday**

*Lecture Recording:* [*https://vimeo.com/257572352/e7f4471f15*](https://vimeo.com/257572352/e7f4471f15)

* Recap/Open Q&A for Week 3 (string algorithms) (10 min)
  + What are methods
* Introduction to Linked Lists (overview)
* Assignment **addFront**, **removeFront**, **contains**, **front** ([link to challenges](#xfeqwrsqqp8p))

***Bonus (pre-recorded) Lecture*** *- Linked Lists Overview*

*Part 1* [*https://vimeo.com/257819631/38f951b91d*](https://vimeo.com/257819631/38f951b91d)

*(video was abruptly cut off, part 2 picks up from where part 1 ends)*

*Part 2* [*https://vimeo.com/257821893/0e65f5c0c7*](https://vimeo.com/257821893/0e65f5c0c7)

**(2/28/18) Wednesday**

*Lecture Recording:* [*https://vimeo.com/258033256/753d89ad4b*](https://vimeo.com/258033256/753d89ad4b)

* Recap Linked Lists Q&A
* Assignment **length**, **average**, **min/max**, **display** ([link to challenges](#arpw9x853eo9))

**(3/2/18) Friday**

*Lecture Recording:* [*https://vimeo.com/258332887/5023b7db3d*](https://vimeo.com/258332887/5023b7db3d)

* Recap Wednesday algorithms.
* Review of any algorithm from the month.
* Open Q&A
* Exam Overview (how it will be implemented)

**(3/3/18) Saturday** - Algorithm Exam

* First Exam: 10am PST - 6pm UTC
* Second Exam: 7pm PST - 3am UTC

*Time zone conversion:* [*https://www.worldtimebuddy.com/?pl=1&lid=8,100&h=100*](https://www.worldtimebuddy.com/?pl=1&lid=8,100&h=100)

**Resources**

**The “Basic 13”**

The foundation “Basic 13” algorithm challenges.

**Print 1-255**

*Print all the integers from 1 to 255.*

**Print Sum 0-255**

*Print integers from 0 to 255, and with each integer print the sum so far.*

**\*Find and Print Max**

*Given an array, find and print its largest element.*

**Array with Odds**

*Create an array with all the odd integers between 1 and 255 (inclusive).*

**\*Greater Than Y**

*Given an array and a value Y, count and print the number of array values greater than Y.*

**Max, Min, Average**

*Given an array, print the max, min and average values for that array.*

**\*Swap String For Array Negative Values**

*Replace any negative array values with 'Dojo'.*

**Print Odds 1-255**

*Print all odd integers from 1 to 255.*

**\*Iterate and Print Array**

*Iterate through a given array, printing each value.*

**Get and Print Average**

*Analyze an array’s values and print the average.*

**Square the Values**

*Square each value in a given array, returning that same array with changed values.*

**Zero Out Negative Numbers**

*Return the given array, after setting any negative values to zero.*

**\*Shift Array Values**

*Given an array, move all values forward by one index, dropping the first and leaving a '0' value at the end.*

**Arrays (Overview)**

This chapter, we explore the *array* data structure: reading, changing, adding and removing elements (hence changing the array’s length). Before chapter’s end, we will touch upon associative arrays as well. At this point we expect you to quickly complete the 13 mandatory algorithm challenges. Also, building Array functions such as min(), max(), sum() and average() should be easy and rapid.

Arrays store multiple values, which are accessed by specifying the *index* (the offset from the front of the array) in square brackets. This *random-access* characteristic makes arrays well-suited for accessing values in a different order than they were added. Arrays are less suitable (but still commonly used) in scenarios with many insertions and removals. Each value must be moved to create space for an insertion (or to fill a vacancy caused by a removal). JS array values can be different data types: one array can contain numbers, booleans, strings, etc.

Arrays are zero-based: an array’s first value is located at index 0. Accordingly, the array attribute *length* means “one more than the last index.” As with other interpreted languages, JavaScript arrays are not fixed-length; they automatically grow as values are set beyond the current length.

Tracking variables with T-diagrams is extremely beneficial. Use a t-diagram for challenges this chapter. Below are some of the constructs we’ll use this chapter. Remember these basic building blocks!

**Declaring a new array:**

**var myArr = [];**

**console.log(myArr.length); // -> "0"**

**Setting and accessing array values:**

**myArr[0] = 42; // myArr = [42], myArr.length == 1**

**console.log(myArr[0]); // -> "42"**

**Array.length is determined by the largest index:**

**myArr[1] = "hello"; // myArr == [42,"hello"], myArr.length == 2**

**myArr[2] = true; // myArr == [42,"hello",true], myArr.length == 3**

**Overwriting array values:**

**myArr[0] = 101; // myArr == [101, "hello", true]**

**Arrays can be sparsely populated:**

**myArr[myArr.length+2]=0.2; // myArr == [101,"hello",true,undefined,0.2]**

**console.log(myArr.length); // -> "5"**

**Shorten an array with the pop() method:**

**myArr.pop(); // myArr == [101,"hello",true,undefined]**

**myArr.pop(); // myArr == [101,"hello",true], myArr.length==3**

**Arrays (Basic Challenges)**

This chapter you will familiarize yourself with basic array manipulation. From your work with the Basic 13 challenges, we assume that you already know how to read from numerical arrays, and that you can easily create JavaScript functions to get the **minimum** or **maximum** value, the **sum** of all values in the array, or the **average** of all values in the array. If this is not the case, then definitely review those implementations before continuing to today’s challenges.

Here is a list of concepts to consider; some or all will be used in this chapter.

*for / while loops array.pop() & push()*

*if / else statements*

*can contain different data types arrs are objects, passed by reference (ptr)*

|  |  |
| --- | --- |
| **PushFront** Given an array and an additional value, *insert this value* at the beginning of the array. Do this without using any built-in array methods. **PushFront(arr,val)** **InsertAt** Given an array, index, and additional value, *insert the value into the array* at the given index. Do this without using built-in array methods. You can think of **PushFront(arr,val)** as equivalent to **InsertAt(arr,0,val)**. | **PopFront** Given an array, *remove and return the value* at the beginning of the array. Do this without using any built-in array methods except **pop()**. **RemoveAt** Given an array and an index into the array, *remove and return the array value* at that index. Do this without using any built-in array methods except **pop()**. Think of **PopFront(arr)** as equivalent to **RemoveAt(arr,0)**. |

Familiarize yourself with basic array manipulation. Here is a list of methods to study; some or all will be used in this chapter’s challenges.

*for / while loops array.pop() & push()*

*if / else statements*

*can contain different data types arrs are objects, passed by reference (ptr)*

|  |  |
| --- | --- |
| **Second-to-Last** Return the second-to-last element of an array.  Given: [1,2,3,4]  Returned: 3 **Second-Largest** Return the second-largest element of an array.  Given: [1,2,3,4,5,6,7]  Return: 6 | **Nth-to-Last** Return the element that is N-from-array’s-end.  Arr, nth  Given [1,2,3,4], Nth to last => 2  Return: 3  **Nth-Largest** Given an array, return the Nth-largest element: there should be (N - 1) elements that are larger.  Given: [1,4,3,2,5,7,6], nth => 3  Return: 5 |

### 

**Arrays 2 (Intermediate Challenges)**

## **Time-space Tradeoff**

Good engineering is all about tradeoffs: knowing what tradeoffs are available, and knowing when to use them. In software engineering, one important tradeoff is *time vs. space*. If you know you will be asked to solve a certain formula repeatedly, you can keep track your previous answer and simply provide that answer rather than recomputing it. For certain problems, whether in algorithms class or in the workplace, *caching* (saving) the results does not make the function any faster when it is first called, but it can make subsequent calls *much* faster. Use this concept in today’s algorithm challenges!

This chapter you will familiarize yourself with basic array manipulation. Here is a list of methods to study; some or all will be used in this chapter’s challenges.

*for / while loops array.pop() & push()*

*if / else statements*

*can contain different data types arrs are objects, passed by reference (ptr)*

|  |  |
| --- | --- |
| **arrConcat** Replicate JavaScript’s **concat()**. Create a standalone function that accepts two arrays. Return a *new* array containing the first array’s elements, followed by the second array’s elements. Do not alter the original arrays. Ex.:  **arrConcat( ['a','b'], [1,2] )** should return **['a','b',1,2]**. **Shuffle** Recreate the **shuffle()**built into JavaScript, to efficiently shuffle a given array’s values. Do you need to return anything from your function? | **Faster Factorial** Remember iFactorial from last chapter? Take that implementation and use a time-space tradeoff to accelerate the average running time. Recall that iFactorial(num) returns the product of positive integers from 1 to the given num. For example: **fact(1)** = 1, **fact(2)** = 2, **fact(3)** = 6. For these purposes, **fact(0)** = 1. |

**Arrays 2 (Intermediate 2 Challenges)**

This chapter you will familiarize yourself with basic array manipulation. Some or all of these were used in this chapter’s challenges.

*for / while loops array.pop() & push()*

*arrays grow: arr.length == lastIdx-1 if / else statements*

*can contain different data types arrs are objects, passed by reference (ptr)*

For extra array practice at the end of this chapter, work on these additional challenges:

|  |  |
| --- | --- |
| **Smarter Sum** Use a time-space tradeoff to accelerate the average running time of an iSigma(num) function that returns the sum of all positive integers from 1 to num. Recall: sig(1) = 1, sig(2) = 3, sig(3) = 6, sig(4) = 10. **Fabulous Fibonacci** Use a time-space tradeoff to accelerate the average running time of an iFibonacci(num) function that returns the ‘num’th number in the Fibonacci sequence. Recall: fib(0) = 0, fib(1) = 1, fib(2) = 1, fib(3) = 2. | **Tricky Tribonacci** Why stop with fibonacci? Create a function to retrieve a “tribonacci” number, from the sum of the previous 3. Tribonaccis are {0, 0, 1, 1, 2, 4, 7, 13, 24, 44, 81, ...}. Again, use a time-space tradeoff to make this fast. |

# **Strings - Monday**

This chapter explores associative arrays and then *strings* – a special case of the basic indexed array. By now you should be able to complete “Basic 13” algorithm challenges in less than 2 minutes each.

Remember best practices mentioned previously. Ask clarifying questions before rushing to write code. Think about and note any special-case situations (corner cases). Verify understanding by restating problems and intended outputs for simple input. *Then* start coding. Write T-diagrams to track variables.

## **Associative Arrays**

Associative arrays (dictionaries, maps, key-value pairs) are JavaScript objects. Think of them as arrays indexed by strings. Associative arrays are initialized as {}, not [ ]. Access them as object attributes (**arr.attrib**) or array indices (**arr["index"]**).

Associative arrays / objects / maps / dictionaries

**var myAssocArr = {};**

**myAssocArr.IQ = 116;**

**myAssocArr["fun"] = "Martin honks on a tenor saxophone";**

**console.log(myAssocArr);**

**// { IQ: 116; fun: "Martin honks on a tenor saxophone" }**

### **Arrs2Map**

Given two arrays, create an associative array (map) containing keys of the first, and values of the second. For

**arr1 = ["abc", 3, "yo"]** and

**arr2 = [42, "wassup", true]**,

return **{"abc": 42, 3: "wassup", "yo": true}**.

### **InvertHash**

Create invertHash(assocArr) that converts a hash’s keys to values and values to corresponding keys.

Example:

given **{"name": "Zaphod", "numHeads": 2}**,

return **{"Zaphod": "name", 2: "numHeads"}**.

You will need to learn and use a JavaScript **for ... in**here!

### 

**.myJoin method (arr) => return string**

Given an array of strings, return a string

Given [‘a’,’b’,’c’]

Return ‘abc’

**.mySplit method (str) => return array**

Given ‘abc’

Return [‘a’,’b’,’c’]

OPTIONAL: **.mySplit2 method (str) => return array**

Given ‘aa bb cc’, ‘ ’

Return [‘aa’,’bb’,’cc’]

Given ‘aa bb cc’, ‘bb’

Return [‘aa ’,’ cc’]

Given ‘aa bb cc bb dd’, ‘bb’

Return [‘aa ’,’ cc ’, ‘ dd’]

### **ReverseString**

Implement a function reverseString(str) that, given a string, will return the string of the same length but with characters reversed. Example:

given **"creature"**,

return **"erutaerc"**.

Do not use the built-in **reverse()** function!

Strings are arrays of characters (more accurately, arrays of one-character *strings*). Once a string is defined, individual characters can be referenced by [ ] but *not changed*. Strings are *immutable*: they can be completely replaced in their entirety, but not changed piecewise. To manipulate string characters, you must split the string to an *array*, make individual changes, then join it.

**Strings**

**console.log(typeof myAssocArr.fun); // "string"**

**var myChar = myAssocArr.fun[26]; // "x"**

**console.log(typeof myChar); // "string"**

**.length method**

**console.log(myAssocArr.fun.length); // 33**

**console.log("".length); // 0**

**.split method**

**myArray = myAssocArr["fun"].split(" ");**

**// ["Martin","honks","on","a","tenor","saxophone"]**

**console.log(myArray[5].split("");**

**// ["s","a","x","o","p","h","o","n","e"];**

**.join method**

**console.log(myArray.join());**

**// "Martin, honks, on, a, tenor, saxophone"**

**console.log(myArray.join("-"));**

**// "Martin-honks-on-a-tenor-saxophone"**

**Challenge:** what is displayed by the following? Why?

**console.log(1 + 2 + "3" + "4" + 5 + 6);**

### 

**String Wednesday**

### **Remove Blanks**

Create a function that, given a string, returns the string, without blanks.

Given **" play that Funky Music "**,

returns a string containing **"playthatFunkyMusic"**.

### 

### **Get String Digits**

Create a JavaScript function that given a string, returns the integer made from the string’s digits.

Given **"0s1a3y5w7h9a2t4?6!8?0"**, the function should

return the number 1357924680.

### 

### **Acronyms**

Create a function that, given a string, returns the string’s acronym (first letters only, capitalized).

Given **"there's no free lunch - gotta pay yer way"**,

return **"TNFL-GPYW"**.

Given **"Live from New York, it's Saturday Night!"**,

return **"LFNYISN"**.

**OPTIONAL:**

How to retrieve the Ascii numerical value of

<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String/charCodeAt>

Ascii value of “a”

Ascii value of “A”

# **String Friday (Bonus)**

## **Switch/case statements**

Think of switch statements as a series of if statements. From the **switch(VAL)**, execution jumps forward to the **case:** that matches the VAL (or **default:** if no match is found), continuing from there until an optional **break;**.

This chapter we explore associative arrays then strings. Some or all of these will be useful:

*.length .split .join .concat for...in loops switch/case*

|  |  |
| --- | --- |
| **Parens Valid** Create a function that, given an input string, returns a boolean whether parentheses in that string are valid. Given input **"y(3(p)p(3)r)s"**, return true. Given **"n(0(p)3"**, return **false**. Given **"n)0(t(0)k"**, return **false**. | **Braces Valid** Given a string, returns whether the sequence of various parentheses, braces and brackets within it are valid. For example, given the input string **"w(a{t}s[o(n{c}o)m]e)h[e{r}e]!"**, return **true**. Given **"d(i{a}l[t]o)n{e"**, return **false**. Given **"a(1)s[O(n]0{t)0}k"**, return **false**. |

This chapter we focus on *strings*. Some or all of these might be useful:

*.length .split .join .concat for...in loops switch/case*

|  |  |
| --- | --- |
| **Is Palindrome** Strings like **"Able was I, ere I saw Elba"** or **"Madam, I'm Adam"** could be considered *palindromes*, because (if we ignore spaces, punctuation and capitalization) the letters are the same from front and back.  Create a function that returns a boolean whether the string is a *strict* palindrome. For **"a x a"** or **"racecar"**, return **true**. Do **not** ignore spaces, punctuation and capitalization: if given **"Dud"** or **"oho!"**, return **false**. | **Longest Palindrome** For this challenge, we will look not only at the entire string, but also substrings within it.  For a string, return the longest palindromic substring. Given **"what up, dada?"**, return **"dad"**. Given **"not much"**, return **"n"**. *Include spaces* as well (i.e. be strict, as in the “Is Palindrome” challenge): given **"My favorite racecar erupted!"**, return **"e racecar e"**. |

### 

### 

# **Linked Lists**

This chapter we explore linked lists, a data structure used widely in lower layers such as backends, frameworks, runtimes or operating systems. You should be familiar with object oriented ideas, including the *reference* concept: not a local copy of a value, but a pointer to the value in shared memory.

How does your operating system keeps track of the files in a directory? Modern systems do not do this with an array. They use a data structure called a *linked list*. Linked lists are easily reordered and well-suited for large data collections because (unlike arrays) they store data in small pieces of memory that “fit in the holes” between variables, rather than requiring a large chunk of contiguous memory. Linked lists are the first data structure we discuss as an *object* and introduce us to the concept of a *reference*.

A **class definition** is like a blueprint of a complex machine, from which many copies can be made. Actually constructing a machine is a separate step. Likewise, *declaring* a class merely informs us of that blueprint; actual objects must be individually constructed. In JavaScript, class declarations take the form of functions called *object constructors* – when called, they create an **object** for the caller. An object is an instance of the class, brought to life, just like a physical copy of the ideas in the blueprint.

Not all machines are complex, and not all objects are complicated. However, code can add and remove attributes of objects on-the-fly, so this makes them different than a boolean or number which always occupies the same amount of memory space. Why does this matter? Well, if you have debugged your JavaScript code in the browser, you may understand the idea of a call stack. This is the series of function calls that led the computer to where it is right now. Whenever the currently running function returns, the JavaScript runtime will look to the call stack to help it “remember” which function it came from, as well as the state of all its local variables at the time when it called into another function. The runtime stores local variables in the call stack while changing the execution to another function. Setting aside call stack space for booleans and numbers is easy -- regardless of value, numbers occupy a 64-bit chunk of memory. However, objects are tricky: the JavaScript runtime cannot determine *a priori* how much space to set aside for your objects. So how can it quickly construct a call stack?

The answer is that objects are created using a common chunk of memory set aside for variable-sized allocations. This memory is called the *heap*, and it is used for any unpredictable memory needs. When the system looks at your ‘blueprint’ and constructs a ‘machine’ corresponding to those plans, it goes to the heap and sets aside space for all that object’s attributes and functions. If the object needs more space, it expands into adjacent heap memory. During normal operation, the heap is wide-open for large and small allocations. The call stack is apartment space in a high-rise tower; the heap is Montana.

When you create an object and store it in a local **var**, the system doesn’t put the object in that memory slot the way it does for a number or a boolean. It puts a *reference* to that heap location into your local **var**. References (called pointers) are fixed-size, so this enables the runtime do its stack magic. A pointer represents an object’s location in memory, but you can think of it as an object’s contact info: its email address. True to its name, a pointer *points* to where the object is found. If you have information to retrieve from (or store to) an object, you “go there” by dereferencing that pointer, followed by the attribute you want within the object. This could look like **myProject.name** or **myQuizzes[3]** or even **getAverage(myArr)**. Yes, arrays, strings and even functions are objects – dereferenced by **.** or **[** or **(** .

**Week 4 Monday – Linked Lists**

Over the chapter’s course, we’ll coalesce a considerable collection of concepts to contemplate. Some or all of these will be used in this chapter’s challenges.

*classes* and *objects object constructors local vars vs. heap allocations pointers*

*reference* vs. *value private* vs. *public ===* vs. *== push( ) & pop( )*

|  |  |
| --- | --- |
| To the left is a definition of a **node** object. A node object simply holds a value, as well as a *pointer* that links it to the next node in the sequence, if there is one. A sequence of node objects is called a *linked list*. | **function listNode(value)**  **{**  **this.val = value;**  **this.next = null;**  **}** |

|  |  |
| --- | --- |
| **addFront** Given a pointer to the **first node** in a list, and a **value**, create a new node, connect it to the head of the list, and return a pointer to the list’s new head node.  **addFront(firstNode, val) => newFirstNode** **contains** Given a pointer to a **listNode** and a **value**, return whether value is found in any node within the list. | **removeFront** Given a pointer to the **first node** in a list, remove the head node and return the new list. If list is empty, return null.  **removeFront(firstNode) => newFirstNode** **front** Return the *value* (not the node) at the **head** of the list. If list is empty, return null.  **front(firstNode) => val or null** |

**contants(firstNode, value) => boolean**

### 

## Linked Lists Wednesday

This chapter we will familiarize ourselves with basic manipulation of the *singly linked list* data structure. Why is it referred to as a *singly* linked list? Well, there are many other ways to arrange node objects, and some of them feature more than one linkage between nodes. For example, *doubly linked list* nodes each connect to *two* others: the next one as well as the previous. *Singly* linked list nodes contain only a *next* pointer. Here are some of the concepts used in this chapter’s challenges.

*classes* and *objects object constructors local variables vs. heap allocations push( ) & pop( )* *pointers private* vs. *public ===* vs. *== reference* vs. *value*

For the following challenges, use this listNode definition as a starting point. Note: *singly linked lists* are sometimes referred to as **sLists**.

### **length**

Create a function that accepts a pointer to first list node, and returns number of nodes in sList.

**length(node) => counter**

### **average**

Create a standalone function **average(node)** that returns (…wait for it … ) the *average* of all values contained in that list.

**average(node) => value**

**function listNode(value)**

**{**

**this.val = value;**

**this.next = null;**

**}**

### **min, max**

Create function**min(node)** and **max(node)** to returning smallest and largest values in the list.

**min(node) => minValue**

**max(node) => maxValue**

### **display**

Create **display(node)** for debugging that returns a string containing all list values. Build what you wish **console.log(myList)** did!

### **display(node) => String (contains list values)**

Additional Resources:

**Zen Programmer** - Great article about programming and your mindset. <https://www.zenprogrammer.org/en/10-rules-of-a-zen-programmer.html>

**Web 3.0** - New way of using the web.

<https://medium.com/@matteozago/why-the-web-3-0-matters-and-you-should-know-about-it-a5851d63c949>