Table of Content

Concepts

JavaScript Questions

90. Custom cookie
91. Create an immutability helper - part 1
92. Create an immutability helper - part 2
93. Make high priority API call
94. Convert JSON to HTML
95. Convert HTML to JSON
96. Concurrent history tracking system
97. Implement an in-memory search engine
98. Implement a fuzzy search function
99. Cancel an API request
100. Highlight words in the string

React Questions

1. usePrevious() hook
2. useIdle() hook
3. useAsync() hook
4. useDebounce() hook
5. useThrottle() hook
6. useResponsive() hook
7. useWhyDidYouUpdate() hook
8. useOnScreen() hook
9. useScript() hook
10. useOnClickOutside() hook
11. useHasFocus() hook
12. useToggle() hook
13. useCopy() hook
14. useLockedBody() hook
15. Number Increment counter
16. Capture product visible in viewport
17. Highlight text on selection
18. Batch API calls in sequence
19. Time in human readable format
20. Detect overlapping circles

System design Questions

1. Maker checker flow
2. Online coding judge

Retry promises N number of times

Problem Statement -

Implement a function in JavaScript that retries promises N number of times with a delay between each call.

Example

```
Input:
retry(asyncFn, retries = 3, delay = 50, finalError = 'Failed');

Output:
... attempt 1 -> failed
... attempt 2 -> retry after 50ms -> failed
... attempt 3 -> retry after 50ms -> failed
... Failed.
```

In short, we have to create a retry function that Keeps on retrying until the promise resolves with delay and max retries.

We will see two code examples one with thenable promise and the second with async...await.

Let us first create the delay function so that we can take a pause for a specified amount of time.

Delay function

We can create a delay function by creating a new promise and resolve it after given time using setTimeout.

```
//delay func
const wait = ms => new Promise((resolve) => {
  setTimeout(() => resolve(), ms);
});
```

Approach 1 - Using then...catch

To retry the promise we have to call the same function recursively with reduced max tries, if the promise failed that is in the catch block. Check if there are a number of tries left then recursively call the same function or else reject with the final error.

Implementation

```
const retryWithDelay = (
  operation, retries = 3,
  delay = 50, finalErr = 'Retry failed') => new Promise((resolve, reject)
=> {
  return operation()
    .then(resolve)
    .catch((reason) => {
      //if retries are left
      if (retries > 0) {
        //delay the next call
        return wait(delay)
          //recursively call the same function to retry with max retries -
1
          .then(retryWithDelay.bind(null, operation, retries - 1, delay,
finalErr))
          .then(resolve)
          .catch(reject);
      }

      // throw final error
      return reject(finalErr);
    });
});
```

Test Case

To test we can create a test function that keeps throwing errors if called less than 5 times. So if we call it 6 or more times, it will pass.

```
Input:
```

```
// Test function
const getTestFunc = () => {
  let callCounter = 0;
  return async () => {
    callCounter += 1;
    // if called less than 5 times
    // throw error
    if (callCounter < 5) {
      throw new Error('Not yet');
    }
  }
}

// Test the code
const test = async () => {
  await retryWithDelay(getTestFunc(), 10);
  console.log('success');
  await retryWithDelay(getTestFunc(), 3);
  console.log('will fail before getting here');
}

// Print the result
test().catch(console.error);

Output:
"success" // 1st test
"Retry failed" //2nd test
```


Approach 2 - Using async...await.

We can implement the same login using async-await. When using async-await, we need to wrap the code inside try..catch block to handle the error, thus in the catch block, we can check if the max retries are still left then recursively call the same function or else throw the final error.

Implementation

```
const retryWithDelay = async (
  fn, retries = 3, interval = 50,
  finalErr = 'Retry failed'
```

```
) => {
  try {
    // try
    await fn();
  } catch (err) {
    // if no retries left
    // throw error
    if (retries <= 0) {
      return Promise.reject(finalErr);
    }

    //delay the next call
    await wait(interval);

    //recursively call the same func
    return retryWithDelay(fn, (retries - 1), interval, finalErr);
  }
}
```

Test Case

```
Input:
// Test function
const getTestFunc = () => {
  let callCounter = 0;
  return async () => {
    callCounter += 1;
    // if called less than 5 times
    // throw error
    if (callCounter < 5) {
      throw new Error('Not yet');
    }
  }
}

// Test the code
const test = async () => {
  await retryWithDelay(getTestFunc(), 10);
  console.log('success');
  await retryWithDelay(getTestFunc(), 3);
  console.log('will fail before getting here');
}
```

```
// Print the result
test().catch(console.error);

Output:
"success" // 1st test
"Retry failed" //2nd test
```

Alternatively, you can also update the code and keep on retrying the API call based on some test.

Implement mapLimit async function

Problem Statement -

Implement a mapLimit function that is similar to the Array.map() but returns a promise that resolves on the list of output by mapping each input through an asynchronous iteratee function or rejects it if any error occurs. It also accepts a limit to decide how many operations can occur at a time.

The asynchronous iteratee function will accept an input and a callback. The callback function will be called when the input is finished processing, the first argument of the callback will be the error flag and the second will be the result.

Example

```
Input:
let numPromise = mapLimit([1, 2, 3, 4, 5], 3, function (num, callback) {

  // i am async iteratee function
  // do async operations here
  setTimeout(function () {
    num = num * 2;
    console.log(num);
    callback(null, num);
  }, 2000);
});

numPromise
  .then((result) => console.log("success:" + result))
  .catch(() => console.log("no success"));

Output:
/// first batch
2
4
6
/// second batch
```

```
8
10
/// final result
"success: [2, 4, 6, 8, 10]
```

To implement this function we will have to use the combination of both Async.parallel and Async.series.

- First chop the input array into the subarrays of the given limit. This will return us an array of arrays like [[1, 2, 3], [4, 5]].
- The parent array will run in series, that is the next subarray will execute only after the current subarray is done.
- All the elements of each sub-array will run in parallel.
- Accumulate all the results of each sub-array element and resolve the promise with this.
- If there is any error, reject.

```
// helper function to chop array in chunks of given size
Array.prototype.chop = function (size) {
  //temp array
  const temp = [...this];

  //if the size is not defined
  if (!size) {
    return temp;
  }

  //output
  const output = [];
  let i = 0;

  //iterate the array
  while (i < temp.length) {
    //slice the sub-array of a given size
    //and push them in output array
    output.push(temp.slice(i, i + size));
    i = i + size;
  }
```

```
  return output;
};

const mapLimit = (arr, limit, fn) => {
  // return a new promise
  return new Promise((resolve, reject) => {
    // chop the input array into the subarray of limit
    // [[1, 2, 3], [1, 2, 3]]
    let chopped = arr.chop(limit);

    // for all the subarrays of chopped
    // run it in series
    // that is one after another
    // initially it will take an empty array to resolve
    // merge the output of the subarray and pass it on to the next
    const final = chopped.reduce((a, b) => {

      // listen on the response of previous value
      return a.then((val) => {

        // run the sub-array values in parallel
        // pass each input to the iteratee function
        // and store their outputs
        // after all the tasks are executed
        // merge the output with the previous one and resolve
        return new Promise((resolve, reject) => {

          const results = [];
          let tasksCompleted = 0;
          b.forEach((e) => {

            // depending upon the output from
            // async iteratee function
            // reject or resolve
            fn(e, (error, value) => {

              if(error){
                reject(error);
              }else{
                results.push(value);
                tasksCompleted++;
                if (tasksCompleted >= b.length) {
```

```
                resolve([...val, ...results]);
              }
            }
          });
        });
      });

    }, Promise.resolve([]));

    // based on final promise state
    // invoke the final promise.
    final
      .then((result) => {
        resolve(result);
      })
      .catch((e) => {
        reject(e);
      });
  });
};
```

Test Case 1: All the inputs resolve.

```
Input:
let numPromise = mapLimit([1, 2, 3, 4, 5], 3, function (num, callback) {
  setTimeout(function () {
    num = num * 2;
    console.log(num);
    callback(null, num);
  }, 2000);
});

numPromise
  .then((result) => console.log("success:" + result))
  .catch(() => console.log("no success"));

Output:
// first batch
2
4
6
// second batch
```

```
8
10
"success:2,4,6,8,10"
```

Test Case 2: Rejects.

```
Input:
let numPromise = mapLimit([1, 2, 3, 4, 5], 3, function (num, callback) {
  setTimeout(function () {
    num = num * 2;
    console.log(num);

    // throw error
    if(num === 6){
      callback(true);
    }else{
      callback(null, num);
    }

  }, 2000);
});

numPromise
  .then((result) => console.log("success:" + result))
  .catch(() => console.log("no success"));

Output:
// first batch
2
4
6
"no success"
```

Create Pausable auto incrementer

Problem Statement -

Create a pausable auto incrementor in JavaScript, which takes an initial value and steps as input and increments the initial value with given steps every second. The incrementer can be paused and resumed back.

It is one of the classic problems which use two of the trickiest concepts of JavaScript.

1. Timers.
2. Closure.

Use the setInterval to auto increment the values, whereas wrapping the start and stop function inside the closure and returning them, so that the incrementor can be controlled while still maintaining the value.

Defining the function body

Our incrementor function will take two inputs, initial value, and steps. And within the function, we will need two variables,

1. To track the value.
2. For storing the IntervalId, so that it can be cleared to stop the timer and resume when needed.

```
const timer = (init = 0, step = 1) => {
    let intervalId;
    let count = init;
}
```

Start function

In the start function, setInterval will be invoked at an interval of 1 second, and in each interval call, the initial value will be increased by the given step and it will be logged in the console.

setInterval's id will be stored In the intervalId variable.

```
const startTimer = () => {
    if (!intervalId){
      intervalId = setInterval(() => {
        console.log(count);
        count += step;
      }, 1000);
    }
  }
```

There is a condition to check if intervalId is having any value or not, just to make sure we don't start multiple intervals.

Stop function

Inside the stop function we stop the increment by invoking the clearInterval by passing the intervalId to it and also updating the intervalId to null.

```
const stopTimer = () => {
  clearInterval(intervalId);
  intervalId = null;
}
```

At the end return the startTimer and stopTimer.

```
return {
 startTimer,
 stopTimer,
};
```

Implementation

```
const timer = (init = 0, step = 1) => {
  let intervalId;
  let count = init;

  const startTimer = () => {
    if (!intervalId){
      intervalId = setInterval(() => {
        console.log(count);
        count += step;
      }, 1000);
    }
  }

  const stopTimer = () => {
    clearInterval(intervalId);
    intervalId = null;
  }

  return {
    startTimer,
    stopTimer,
  };
}
```

Test Case

```
Input:
const timerObj = timer(10, 10);
//start
timerObj.startTimer();

//stop
setTimeout(() => {
    timerObj.stopTimer();
}, 5000);

Output:
10
20
30
40
```

Implement stack with max and min function

Problem Statement -

Implement a stack data structure in which we can get the max and min value through function in O(1) time.

Example

```
Input:
2 5 17 23 88 54 1 22

Output:
max: 88
min: 1
```

Approach

- The idea is very simple, instead of storing a single value in the stack we will store an object with current, max and min values.
- While adding the new value in the stack we will check if the stack is empty or not.
- If it is empty then add the current value as current, max and min values.
- Else get the previous value and compare it with the current item, if it is greater than the existing then replace the max, If it is less than the existing then replace the min.

Implementation

```
function stackWithMinMax(){
    let items = [];
    let length = 0;

    this.push = (item) => {
      //Check if stack is empty
      //Then add the current value at all place
      if(length === 0){
```

```
      items[length++] = {value: item, max: item, min: item};
    }else{
      //Else get the top data from stack
      const data = this.peek();
      let {max, min} = data;

      //If it is greater than previous then update the max
      max = max < item ? item : max;

      //If it is lesser than previous then update the min
      min = min > item ? item : min;

      //Add the new data
      items[length++] = {value: item, max, min};
    }
  }

  //Remove the item from the stack
  this.pop = () => {
    return items[--length];
  }

  //Get the top data
  this.peek = () => {
    return items[length - 1];
  }

  //Get the max value
  this.max = () => {
    return items[length - 1].max;
  }

  //Get the min value
  this.min = () => {
    return items[length - 1].min;
  }

  //Get the size
  this.size = () => {
    return length;
  }

  //Check if its empty
```

```
    this.isEmpty = () => {
      return length === 0;
    }

    //Clear the stack
    this.clear = () => {
      length = 0;
      items = [];
    }
}
```

Test Case

```
Input:
let SM = new stackWithMinMax();
SM.push(4);
SM.push(7);
SM.push(11);
SM.push(23);
SM.push(77);
SM.push(3);
SM.push(1);
SM.pop();
console.log(`max: ${SM.max()}`, `min: ${SM.min()}`);

Output:
"max: 77" "min: 3"
```

Remove cycle from the object

Problem Statement -

Given an object with a cycle, remove the cycle or circular reference from it.

Example

```
Input:
const List = function(val){
  this.next = null;
  this.val = val;
};

const item1 = new List(10);
const item2 = new List(20);
const item3 = new List(30);

item1.next = item2;
item2.next = item3;
item3.next = item1;

// this form a cycle, if you console.log this you will see a circular
object,
// like, item1 -> item2 -> item3 -> item1 -> so on.

Output:
// removes cycle
// item1 -> item2 -> item3
```

If you see the above example, we have created a list object, that accepts a value and pointer to the next item in the list, similar to a linked list, and using this we have created the circular object.

We have to create a function that will break this cycle, in this example to break the cycle we will have to delete the next pointer of the *item3*.

There are two places where this cycle removal can take place.

1. For normal use of Object.
2. While creating the JSON string.

Normal use

We can use WeakSet which is used to store only unique object references and detect if the given object was previously detected or not, if it was detected then delete it.

This cycle removal always takes in-place.

```
const removeCycle = (obj) => {
    //set store
    const set = new WeakSet([obj]);

    //recursively detects and deletes the object references
    (function iterateObj(obj) {
        for (let key in obj) {
            // if the key is not present in prototype chain
            if (obj.hasOwnProperty(key)) {
                if (typeof obj[key] === 'object'){
                    // if the set has object reference
                    // then delete it
                    if (set.has(obj[key])){
                      delete obj[key];
                    }
                    else {
                      //store the object reference
                        set.add(obj[key]);
                      //recursively iterate the next objects
                        iterateObj(obj[key]);
                    }
                }
            }
        }
    })(obj);
}
```

Test Case

```
Input:
const List = function(val){
  this.next = null;
  this.val = val;
};

const item1 = new List(10);
const item2 = new List(20);
const item3 = new List(30);

item1.next = item2;
item2.next = item3;
item3.next = item1;

removeCycle(item1);
console.log(item1);

Output:
/*
{val: 10, next: {val: 20, next: {val: 30}}}
*/
```

Using JSON.stringify while creating JSON

JSON.stringify accepts a replacer function that can be used to alter the value of the stringification process.

We can use the same function to detect and remove the cycle from the object.

```
const getCircularReplacer = () => {
    //form a closure and use this
    //weakset to monitor object reference.
    const seen = new WeakSet();

    //return the replacer function
    return (key, value) => {
        if (typeof value === 'object' && value !== null) {
            if (seen.has(value)) {
```

```
                return;
            }
            seen.add(value);
        }
        return value;
    };
};
```

Test Case

```
Input:
const List = function(val){
  this.next = null;
  this.val = val;
};

const item1 = new List(10);
const item2 = new List(20);
const item3 = new List(30);

item1.next = item2;
item2.next = item3;
item3.next = item1;

console.log(JSON.stringify(item1, getCircularReplacer()));

Output:
"{'next':{'next':{'val':30},'val':20},'val':10}"
```

Method chaining - Part 2

Problem Statement -

Showcase a working demo of method chaining in JavaScript by implementing the following example.

Example

```
Input:
computeAmount().lacs(15).crore(5).crore(2).lacs(20).thousand(45).crore(7).value();

Output:
143545000
```

In the previous problem we have already seen an example of method chaining in which we used object-based and function-based implementations.

Similarly we will solve this problem with different approaches as well.

Method 1: Using function as constructor

We will create a constructor function that will help us to create a new instance every time and perform the operations.

```
const ComputeAmount = function(){
  this.store = 0;

  this.crore = function(val){
    this.store += val * Math.pow(10, 7);
    return this;
  };

  this.lacs = function(val){
    this.store += val * Math.pow(10, 5);
    return this;
  }
```

```
  this.thousand = function(val){
    this.store += val * Math.pow(10, 3);
    return this;
  }

  this.hundred = function(val){
    this.store += val * Math.pow(10, 2);
    return this;
  }

  this.ten = function(val){
    this.store += val * 10;
    return this;
  }

  this.unit = function(val){
    this.store += val;
    return this;
  }

  this.value = function(){
    return this.store;
  }
}
```

Test Case

```
Input:
const computeAmount = new ComputeAmount();
const amount =
computeAmount.lacs(15).crore(5).crore(2).lacs(20).thousand(45).crore(7).val
ue();

console.log(amount === 143545000);

Output:
true
```

Method 2: Using function as closure

We will form closure and return a new object from it that will have all the logic encapsulated. This way we won't have to create a constructor everytime and the data won't duplicate too.

```
const ComputeAmount = function(){

  return {
    store: 0,
    crore: function(val){
      this.store += val * Math.pow(10, 7);
      return this;
    },

    lacs: function(val){
      this.store += val * Math.pow(10, 5);
      return this;
    },

    thousand: function(val){
      this.store += val * Math.pow(10, 3);
      return this;
    },

    hundred: function(val){
      this.store += val * Math.pow(10, 2);
      return this;
    },

    ten: function(val){
      this.store += val * 10;
      return this;
    },

    unit: function(val){
      this.store += val;
      return this;
    },
```

```
    value: function(){
      return this.store;
    }
  }
}
```

Test Case

```
Input:
const amount =
ComputeAmount().lacs(9).lacs(1).thousand(10).ten(1).unit(1).value();
console.log(amount === 1010011);

const amount2 =
ComputeAmount().lacs(15).crore(5).crore(2).lacs(20).thousand(45).crore(7).v
alue();
console.log(amount2 === 143545000);

Output:
true
true
```

Currying - Problem 3

Problem Statement -

Write a function that satisfies the following.

```
add(1)(2).value() = 3;
add(1, 2)(3).value() = 6;
add(1)(2)(3).value() = 6;
add(1)(2) + 3 = 6;
```

This is a little tricky question and requires us to use and modify the valueOf() method.

When JavaScript wants to turn an object into a primitive value, it uses the function valueOf() method. JavaScript automatically calls the function valueOf() method when it comes across an object where a primitive value is anticipated, so you don't ever need to do it yourself.

Example

```
function MyNumberType(n) {
  this.number = n;
}

MyNumberType.prototype.valueOf = function () {
  return this.number + 1;
};

const myObj = new MyNumberType(4);
myObj + 3; // 8
```

Thus we can form a closure and track the arguments in an Array and return a new function everytime that will accept new arguments.

We will also override the valueOf() method and return the sum of all the arguments for each primitive action, also add a new method

value() that will reference the valueOf() thus when invoked will return the sum of arguments.

```
function add(...x){
    // store the current arguments
    let sum = x;

    function resultFn(...y){
        // merge the new arguments
        sum = [...sum, ...y];
        return resultFn;
    }

    // override the valueOf to return sum
    resultFn.valueOf = function(){
      return sum.reduce((a, b) => a + b, 0);
    };

    // extend the valueOf
    resultFn.value = resultFn.valueOf;

    // return the inner function
    // on any primitive action .valueOf will be invoked
    // and it will return the value
    return resultFn;
}
```

Test Case

```
Input:
console.log(add(1)(2).value() == 3);
console.log(add(1, 2)(3).value() == 6);
console.log(add(1)(2)(3).value() == 6);
console.log(add(1)(2) + 3);

Output:
true
true
true
6
```

HTML encoding of a string.

Problem Statement -

Given a string and an array representing the HTML encoding of the string from and to with the given tag. Return the HTML encoded string.

Example

```
Input:
const str = 'Hello, world';
const styleArr = [[0, 2, 'i'], [4, 9, 'b'], [7, 10, 'u']];

Output:
'<i>Hel</i>l<b>o, w<u>orl</u></b><u>d</u>'
```

Note – <u> tag gets placed before the <b> tag and after it as the insertion index overlaps it.

It is one of the most common features that is implemented in the WYSIWYG editors, where you write normal text and apply styles to it and it will be converted to HTML encoding at the end.

There are two ways to solve this question.

- First one is extremely simple as most of the work is done by an inbuilt method from the javaScript.
- In the second one, we write the complete logic for encoding.

Let us first see the second method as most of the times in the interview you will be asked to implement this way only.

Custom function for HTML encoding of the string

The styles can be overlapping thus one tag can overlap the other and in such scenarios we have to close the overlapping tags and re-open it again.

```
Input:
const str = "Hello, World";
const style =  [0, 2, 'i'] , [1, 3, 'b'];

Output:
<i>H<b>el</b></i><b>l</b>o, World

// b is starting from 1 and ending at 3, i is in between b.
```

As you can see in the above example b is overlapping thus it is closed at 2nd character of the string and re-opened at 3.

We can solve this with the help of a priority queue and a stack.

- Aggregate the styles on the starting index in the order of priority of the range difference between them. (endIndex – startIndex), this way the longest tag is created first.
- Now using a stack, track the opening and closing of tags.
- If the current style's ending index is greater than the previous one, then create a new tag as it is overlapping and push this new entry back to the stack and in the next iteration create this tag.
- At the end return the string of the top most stack entry.

Priority queue

We are not going to use the complete implementation of Priority Queue, rather a helper function that adds a new item in the array and sorts it in the ascending or descending order depending upon the order.

```
// helper function works as Priority queue
// to add tags and sort them in descending order
// based on the difference between the start and end
function addAndSort(track, index, data) {
  if (!track[index]) track[index] = [];
  track[index] = [...track[index], data];
```

```
  track[index].sort((a, b) => a.getRange() > b.getRange());
};
```

Stack

A simple implementation of the Stack data structure with the required methods.

```
function Stack() {
  let items = [];
  let top = 0;

  //Push an item in the Stack
  this.push = function (element) {
    items[top++] = element;
  };

  //Pop an item from the Stack
  this.pop = function () {
    return items[--top];
  };

  //Peek top item from the Stack
  this.peek = function () {
    return items[top - 1];
  };
};
```

Tag method

A helper function to create a new tag for each entry of styles and its corresponding, also helps to get the range for easier processing in the priority queue. We push this in the priority queue and sort it.

```
// helper function to form a tag
// and trace the string
function Tag(start, end, tag) {
  this.start = start;
  this.end = end;
  this.tag = tag;
  this.text = "";
```

```
  this.getRange = () => {
    return this.end - this.start;
  };
};
```

Encoder (main) method

In this method we perform all the encoding step by step.

- Create an empty array trace of the size of the input string.
- Iterate the styles and form a new tag for each entry.
- On the start index of the styles aggregate the common tags in the priority queue based on the difference of their range in descending order.
- Add these priority queues at the start indexes of the styles in the trace.
- Create a new stack and add an initial Tag with maximum range i.e Tag(start = 0, end = Number.MAX_VALUE, tag = "").
- Iterate till input string length, get the tags from the trace at each index, iterate the prioritized tags if they are present.
- Create the opening HTML tag for each tag in the queue and push it in the stack. Check if the current end Index of the tag is greater than the previous one in the stack, then create a new tag and push it in the priority queue.
- At the end close all the HTML tags at the given index in the loop.

```
function parse(str, markups) {
  // create an empty array for all the indexes of the string
  const track = new Array(str.length).fill(null);

  // add the tag at the starting point
  // of each text mentioned in the markups
  for (let markup of markups) {
    const [start, end, tag] = markup;
    addAndSort(track, start, new Tag(start, end, tag));
  }
```

```
// create a new stack
const html = new Stack();

// initialize with a new Tag that has max range and empty string
html.push(new Tag(0, Number.MAX_VALUE, ""));

// iterate each character of the string
for (let i = 0; i < str.length; i++) {

  // check for opening tags and add them
  while (track[i] && track[i].length > 0) {
    const cur = track[i].shift();
    cur.text = `<${cur.tag}>`;

    // for example in [0, 2, 'i'] , [1, 3, 'b']
    // b is starting from 1 and ending at 3, i is in between b.
    // <i> <b> </b> </i> <b> </b>
    // if the end of the nested tag is larger than the parent,
    // split the tag
    // and insert the remaining split to the bucket after its parent
    if (cur.end > html.peek().end) {
      const split = new Tag(html.peek().end + 1, cur.end, cur.tag);
      cur.end = html.peek().end;
      addAndSort(track, html.peek().end + 1, split);
    }

    // push the new tag
    html.push(cur);
  }

  // add the current character to the currently topmost tag
  html.peek().text += str[i];

  // Check for closing tags and close them.
  while (html.peek().end === i) {
    html.peek().text += `</${html.peek().tag}>`;
    const temp = html.pop().text;
    html.peek().text += temp;
  }
}

// return the topmost
return html.pop().text;
```

```
};
```

Test Case

```
Input:
const encoded = parse('Hello, World',
[[0, 2, "i"],
[7, 10, "u"],
[4, 9, "b"],
[2, 7, "i"],
[7, 9, "u"]]);

console.log(encoded);

Output:
"<i>He<i>l</i></i><i>l<b>o,
<u><u>W</u></u></b></i><b><u><u>or</u></u></b><u>l</u>d"
"Hello, World"
```

Using DOMParser()

DOMParser() parses the HTML source code from the string and the good thing is it appropriately places the opening and closing tags if it is missing.

Thus all we have to do is traverse the styles and add tags at the mentioned opening and closing positions in the input string.

Then pass this string to the DOMParser() and it will generate the HTML from the string.

```
function parse(string, markups) {
  // place the opening and closing tags at the appropriate indexes
  const fragments = markups.reduce((chars, [start, end, tag]) => {
    chars[start] = `<${tag}>` + chars[start];
    chars[end] += ``;
    return chars;
```

```
  }, [...string]);

  // pass this string to DOMParser()
  // to convert it to HTML
  return new DOMParser()
      .parseFromString(fragments.join(''), 'text/html')
      .body.innerHTML;
}
```

Test Case

```
Input:
const encoded = parse('Hello, World',
[[0, 2, "i"],
[7, 10, "u"],
[4, 9, "b"],
[2, 7, "i"],
[7, 9, "u"]]);

console.log(encoded);

Output:
"<i>He<i>l</i>l<b>o,
<u><u>W</u></u></b></i><b><u><u>or</u></u></b><u>l</u>d"
"Hello, World"
```

Create your custom cookie

Problem Statement -

Create your custom cookie that is available on the document object with the only max-age option.

Example

```
document.myCookie = "blog=learnersbucket";
document.myCookie = "name=prashant;max-age=1"; // this will expire after 1 second

console.log(document.myCookie);
// "blog=learnersbucket; name=prashant"

setTimeout(() => {
  console.log(document.myCookie);
}, 1500);
// "blog=learnersbucket"
// "name=prashant" is expired after 1 second
```

As our custom cookie is available on the document object, we will extend the document using Object.defineProperty() and add the custom property "myCookie", this will give us the addition methods like get() and set() that could be used behind the scene to make the cookie entry and return it.

To implement this, we will create a function with a map to persist the entry of each cookie.

- As the cookie string is colon separated values with data and options, in the set() method we will extract the data and separate it on the key and value and also the options (max-age) and store them in the map.
- While getting the cookie, get all the entries of the map and for each entry check if it has expired or not. If it is expired, delete the

entry. Send the remaining entry as a string with a colon-separated.

To parse the input and separate the data and options, we will be using a helper function.

```
// helper function to parse the
// key-value pairs
function parseCookieString(str) {
  // split the string on ;
  // separate the data and the options
  const [nameValue, ...rest] = str.split(';');

  // get the key value separated from the data
  const [key, value] = separateKeyValue(nameValue);

  // parse all the options and store it
  // like max-age
  const options = {};
  for(const option of rest) {
    const [key, value] = separateKeyValue(option);
    options[key] = value;
  }

  return {
    key,
    value,
    options,
  }
}

// helper function
// to separate key and value
function separateKeyValue(str) {
  return str.split('=').map(s => s.trim());
}
```

For the main logic we can extend the document object with Object.defineProperty().

```
// enable myCookie
function useCustomCookie() {

  // to store the key and value
  // of each cookie
  const store = new Map();

  Object.defineProperty(document, 'myCookie', {
    configurable: true,
    get() {

      const cookies = [];
      const time = Date.now();

      // get all the entries from the store
      for(const [name, { value, expires }] of store) {
        // if the entry is expired
        // remove it from the store
        if (expires <= time) {
          store.delete(name);
        }
        // else push the key-value pair in the cookies array
        else {
          cookies.push(`${name}=${value}`);
        }
      }

      // return all the key-value pairs as a string
      return cookies.join('; ');
    },

    set(val) {
      // get the key value of the data
      // and option from the string
      const { key, value, options } = parseCookieString(val);

      // if option has max-age
      // set the expiry date
      let expires = Infinity;
      if(options["max-age"]) {
        expires = Date.now() + Number(options["max-age"]) * 1000;
      }
```

```
      // add the entry in the store
      store.set(key, { value, expires });
    }
  })
};
```

Test Case

```
Input:
useCustomCookie();
document.myCookie = "blog=learnersbucket";

// this will expire after 1 second
document.myCookie = "name=prashant;max-age=1";

console.log(document.myCookie);

setTimeout(() => {
  console.log(document.myCookie);
}, 1500);

Output:
"blog=learnersbucket; name=prashant"
"blog=learnersbucket"
```

Reactjs Questions

Detect overlapping circles

Problem Statement -

Draw circles on the screen on the click and whenever two circles overlap change the color of the second circle.

- When a user clicks anywhere on the DOM, create a circle around it of a radius of 100px with a red background.
- If two or more circles overlap, change the background of the later circle.

Let us understand the logic of creating the circle first.

As we have to create the circle with a radius of 100px (200px diameter), rather than generating the circle on the click, we will store the coordinates of the position where the circle should be generated when the user clicks and then create circles out of these coordinates.

As all the circles will be in absolute position so that they can be freely placed on the screen, we will calculate the top, bottom, left, and right positions that will help in placement as well as detecting if two circles are colliding.

Get the clientX and clientY coordinates when the user clicks and align the circle around with a simple calculation so that it is placed in the center. Also before updating the state check if the current circle is overlapping with the existing circles then update the background color of the current.

```
// helper function to gather configuration when user clicks
const draw = (e) => {
  // get the coordinates where user has clicked
  const { clientX, clientY } = e;

  // decide the position where circle will be created and placed
```

```
  // as the circle is of 100 radius (200 diameter), we are subtracting the
values
  // so that circle is placed in the center
  // set the initial background color to red
  setElementsCoordinates((prevState) => {
    const current = {
      top: clientY - 100,
      left: clientX - 100,
      right: clientX - 100 + 200,
      bottom: clientY - 100 + 200,
      background: "red",
    };

    // before making the new entry
    // check with the existing circles
    for (let i = 0; i < prevState.length; i++) {
      // if the current circle is colliding with any existing
      // update the background color of the current
      if (elementsOverlap(current, prevState[i])) {
        current.background = getRandomColor();
        break;
      }
    }

    return [...prevState, current];
  });
};
```

Assign the event listener and draw the circle on the click.

```
// assign the click event
  useEffect(() => {
    document.addEventListener("click", draw);
    return () => {
      document.removeEventListener("click", draw);
    };
  }, []);
```

Helper function to detect collision and generate random colors.

```
// helper function to generate a random color
const getRandomColor = () => {
  const letters = "0123456789ABCDEF";
  let color = "#";
  for (let i = 0; i < 6; i++) {
    color += letters[Math.floor(Math.random() * 16)];
  }
  return color;
};

// helper function to detect if two elements are overlapping
const elementsOverlap = (rect1, rect2) => {
  const collide = !(
    rect1.top > rect2.bottom ||
    rect1.right < rect2.left ||
    rect1.bottom < rect2.top ||
    rect1.left > rect2.right
  );

  return collide;
};
```

Generate the circles from the coordinates which we have stored after the user has clicked. As the detection is done before making entry into the state, the circles are generated with different colors if they collide.

```
// circle element
const Circle = ({ top, left, background }) => {
  return (
    <div
      style={{
        position: "absolute",
        width: "200px",
        height: "200px",
        borderRadius: "50%",
        opacity: "0.5",
        background,
        top,
```

```
        left,
      }}
    ></div>
  );
};

return (
  <div>
    {/* render each circle */}
    {elementsCoordinates.map((e) => (
      <Circle {...e} key={e.top + e.left + e.right} />
    ))}
  </div>
);
```

Putting everything together.

```
import { useEffect, useState } from "react";

// helper function to generate a random color
const getRandomColor = () => {
  const letters = "0123456789ABCDEF";
  let color = "#";
  for (let i = 0; i < 6; i++) {
    color += letters[Math.floor(Math.random() * 16)];
  }
  return color;
};

// helper function to detect if two elements are overlapping
const elementsOverlap = (rect1, rect2) => {
  const collide = !(
    rect1.top > rect2.bottom ||
    rect1.right < rect2.left ||
    rect1.bottom < rect2.top ||
    rect1.left > rect2.right
  );

  return collide;
};
```

```
const Example = () => {
  // store the configuration of each circle
  const [elementsCoordinates, setElementsCoordinates] = useState([]);

  // helper function to gather configuration when user clicks
  const draw = (e) => {
    // get the coordinates where user has clicked
    const { clientX, clientY } = e;

    // decide the position where circle will be created and placed
    // as the circle is of 100 radius (200 diameter), we are subtracting
the values
    // so that circle is placed in the center
    // set the initial background color to red
    setElementsCoordinates((prevState) => {
      const current = {
        top: clientY - 100,
        left: clientX - 100,
        right: clientX - 100 + 200,
        bottom: clientY - 100 + 200,
        background: "red",
      };

      // before making the new entry
      // check with the existing circles
      for (let i = 0; i < prevState.length; i++) {
        // if the current circle is colliding with any existing
        // update the background color of the current
        if (elementsOverlap(current, prevState[i])) {
          current.background = getRandomColor();
          break;
        }
      }

      return [...prevState, current];
    });
  };

  // assign the click event
  useEffect(() => {
    document.addEventListener("click", draw);
    return () => {
      document.removeEventListener("click", draw);
```

```
    };
  }, []);

  // circle element
  const Circle = ({ top, left, background }) => {
    return (
      <div
        style={{
          position: "absolute",
          width: "200px",
          height: "200px",
          borderRadius: "50%",
          opacity: "0.5",
          background,
          top,
          left,
        }}
      ></div>
    );
  };

  return (
    <div>
      {/* render each circle */}
      {elementsCoordinates.map((e) => (
        <Circle {...e} key={e.top + e.left + e.right} />
      ))}
    </div>
  );
};

export default Example;
```

Output