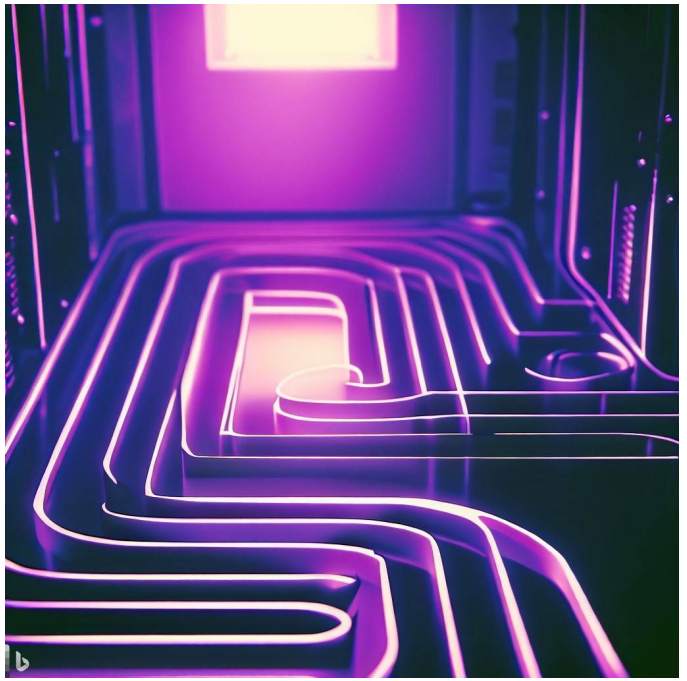


CSE 331

Servers & Routes

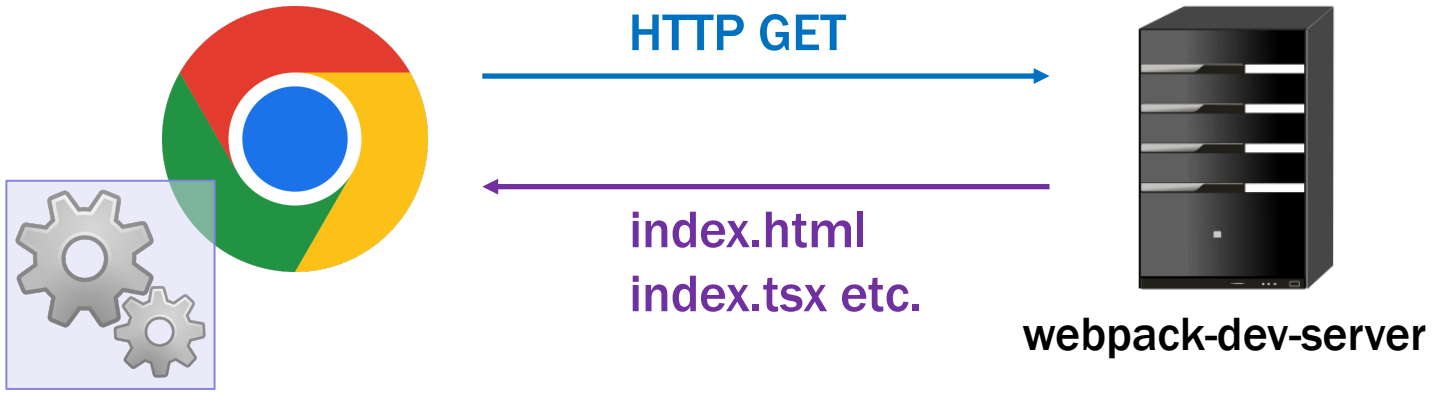
Kevin Zatloukal

Servers & Routes

Client-Side JavaScript

- Code so far has run inside the browser
  - webpack-dev-server handles HTTP requests
  - sends back our code to the browser

- In the browser, executes the code of `index.tsx`
  - calls `root.render` to produce the UI

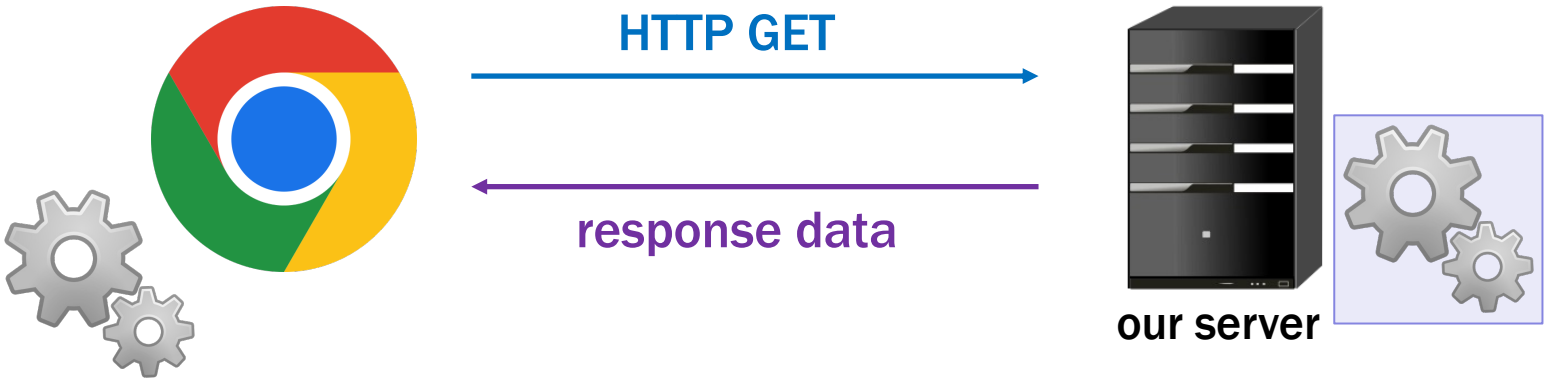


Server-Side JavaScript

- Can run code in the server as well
  - return different data for each request

    data could be HTML, JSON, etc.
  - "node" executes the code of `index.ts`

- Will have code in *both* browser and server
  - only writing server-side code in HW6



Custom Server

- Create a custom server as follows:

```
function F(req: Request, res: Response): void {
 …
}

const app = express();
app.get("/foo", F);
app.listen(8080);
```

  - request for `http://localhost:8080/foo` will call `F`
  - mapping from "`/foo`" to `F` is called a "route"
  - can have as many routes as we want (with different URLs)

Custom Server

- **Query parameters (e.g., `?name=Fred`) in `Request`**

```
function F(req: Request, res: Response): void {
  const name: string|undefined = req.query.name;
  if (name === undefined) {
    res.status(400).send("Missing 'name'");
    return;
  }
  …  // name was provided
}
```

  - set status to 400 to indicate a client error (Bad Request)
  - set status to 500 to indicate a server error
  - default status is 200 (OK)

Custom Server

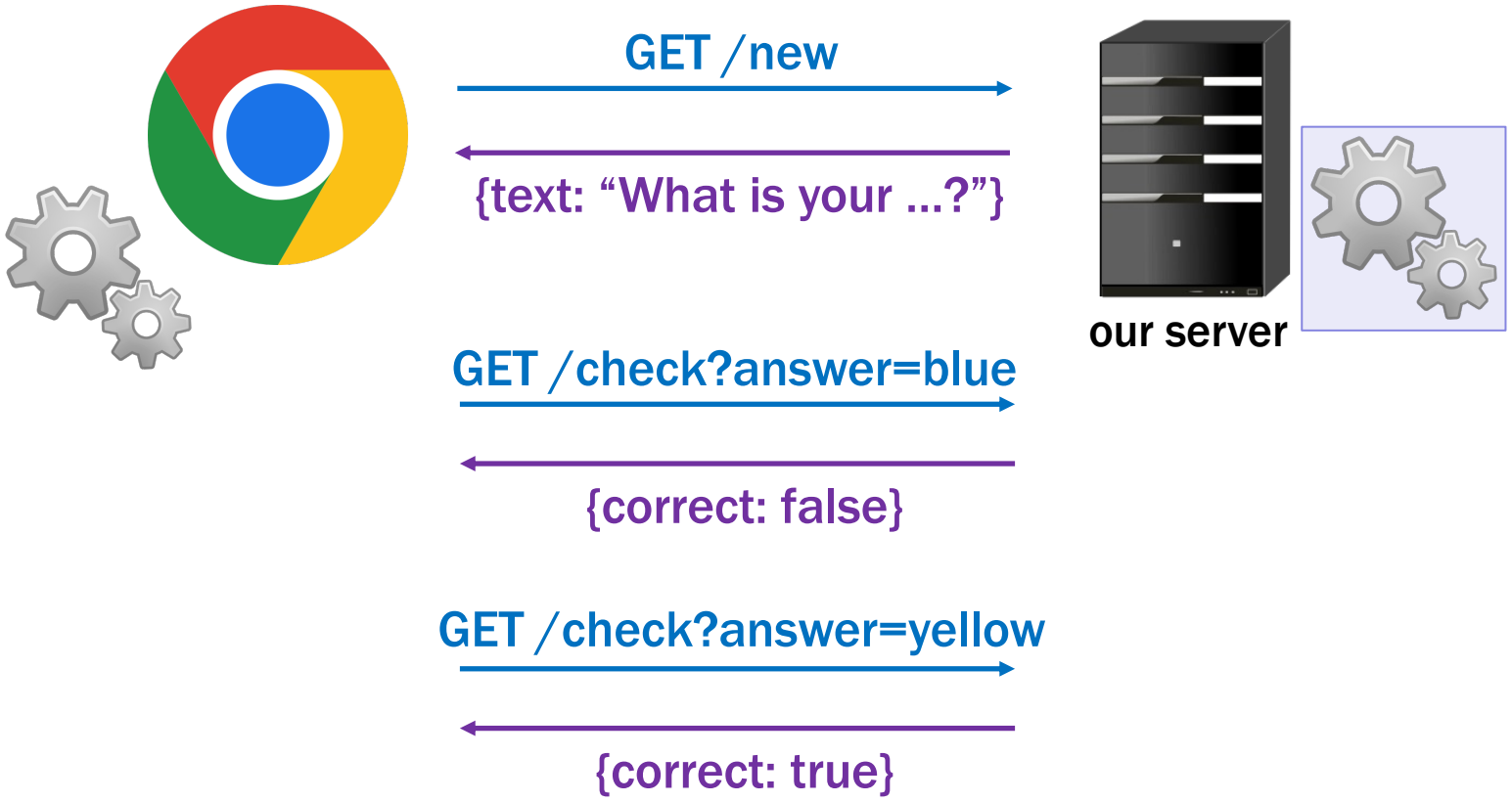
- **Query parameters (e.g., `?name=Fred`) in `Request`**

```
function F(req: Request, res: Response): void {
  const name: string|undefined = req.query.name;
  if (name === undefined) {
    res.status(400).send("Missing 'name'");
    return;
  }
  res.send({message: `Hi, ${name}`});
}
```

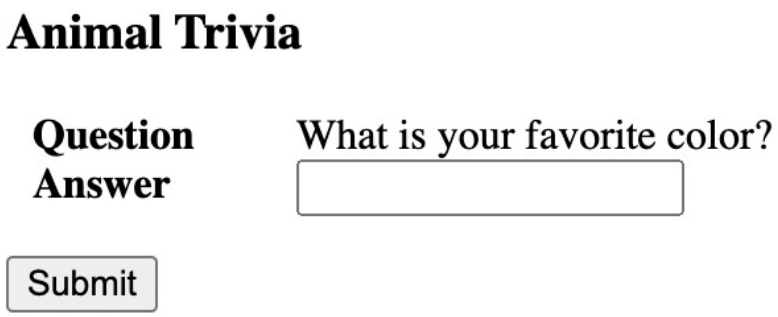
  - `send` **of string returned as text/HTML**
  - `send` **of record returned as application/JSON**

Server-Side JavaScript

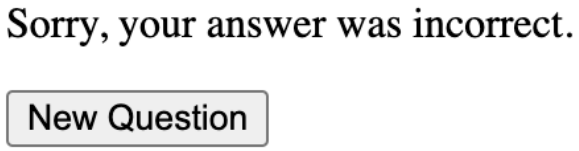
- Apps will make sequence of requests to server
  - e.g., in HW6:



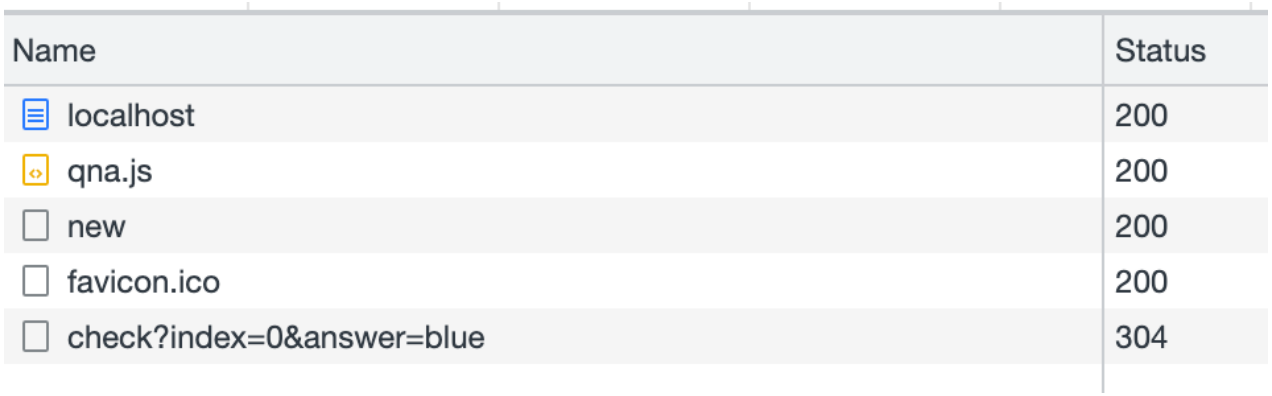
Example App



User types “blue” and presses “Submit”...



“Network” Tab Shows Requests

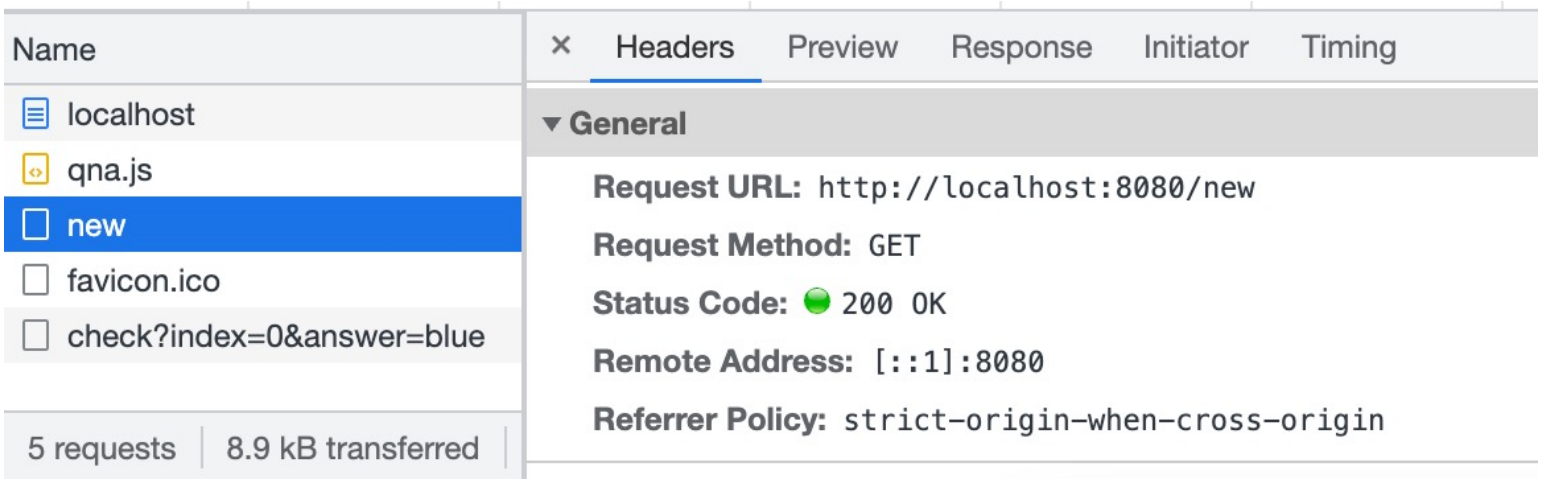


Name	Status
localhost	200
qna.js	200
new	200
favicon.ico	200
check?index=0&answer=blue	304

- Shows every request to the serve
  - first request loads the app (as usual)
  - “new” is a request to get a question
  - “check?answer=blue” is a request to check answer

- Click on a request to see details…

“Network” Tab Shows Request & Response

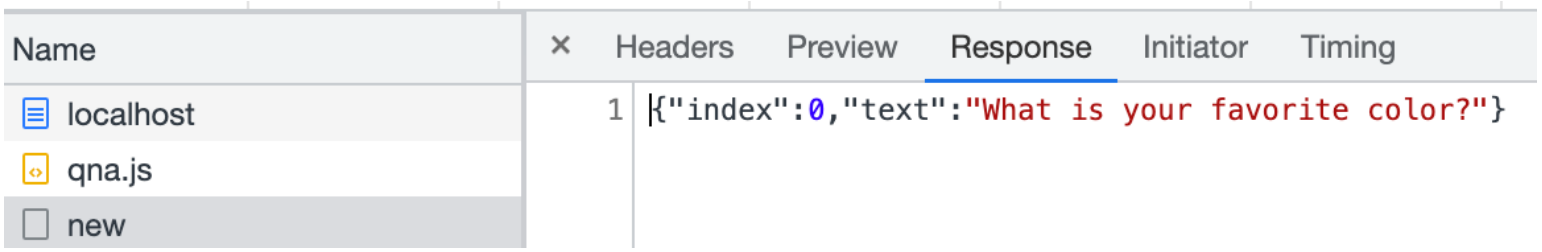


JSON

- **JavaScript Object Notation**
  - **text description of JavaScript object**
  - **allows strings, numbers, null, arrays, and records**
    - no undefined and no instances of classes
    - no '..' (single quotes), only ".."
    - requires quotes around keys in records
  - **another tree!**

- **Translation into string done *automatically* by send**

```
res.send({index: 0, text: 'What is your …?'});
```



Testing Server-Side TypeScript

- A route calls an ordinary function
- Testing is the same as on the client side
  - write unit tests in `X_test.ts` files
  - run then using `npm run test`
- Libraries help set up Request & Response for tests
  - can check the status returned was correct

    e.g., 200 or 400
  - can check the response body was correct

    e.g., “Missing ‘name’” or {message: “Hi, Fred”}

Testing Server-Side TypeScript

- A route calls an ordinary function
- Client- and server-side code is made up of functions
  - server functions handles requests for specific URLs
  - client functions draw data, create requests, etc.
  - test (and code review) each one
- Key Point: unit test each function thoroughly
  - often hard to figure which part caused the failure

    e.g., did the server return an error because of a server bug or a bad request?
  - *much easier* to debug failing tests than errors in the app

Functions with Mutations

Specifying Functions that Mutate

- Our functions so far have not mutated anything

  that makes things *much* simpler!

- Cannot yet write a spec for sorting an array
  - could return a sorted version of the array
  - but cannot say that we change the array to be sorted

- Need some new tags to describe that…

Specifying Functions that Mutate

- By default, no parameters are mutated
  - must *explicitly* say that mutation is possible (default not)

```
/**
 * Reorders A so the numbers are in increasing order
 * @param A array of numbers to be sorted
 * @modifies A
 * @effects A contains the same numbers but now in
 *   increasing order
 */
quickSort(A: number[]): void { .. }
```

  - anything that might be changed is listed in `@modifies`

    not a promise to modify it — `A` could already be sorted!

    a shorter modifies list is a **stronger** specification

Specifying Functions that Mutate

- By default, no parameters are mutated
  - must *explicitly* say that mutation is possible (default not)

```
/**
 * Reorders A so the numbers are in increasing order
 * @param A array of numbers to be sorted
 * @modifies A
 * @effects A contains the same numbers but now in
 *   increasing order
 */
function quickSort(A: number[]): void { .. }
```

  - `@effects` gives promises about result after mutation

    like `@returns` but for mutated values, not return value

    this returns void, so no `@returns`

Mutating Arrays

- Assigning to array elements changes known state

{{ A[j – 1] < A[j] for any 1 ≤ j ≤ 5 }}

```
A[0] = 100;
```

{{ A[0] = 100 and A[j – 1] < A[j] for any 2 ≤ j ≤ 5 }}

- Can add to the end of an array

```
A.push(100);
```

{{ $A = A_0 \# [100]$ }}

- Can remove from the end of an array

```
A.pop();
```

{{ $A = A_0[0 .. n - 2]$ }}

A has one fewer element than before

Example Mutating Function

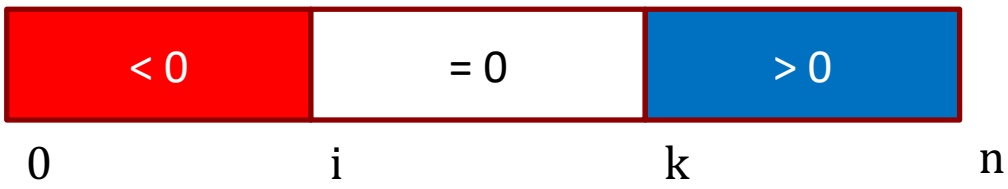
- Reorder an array so that
  - negative numbers come first, then zeros, then positives

    (not necessarily sorted)

```
/**
 * Reorders A into negatives, then 0s, then positive
 * @modifies A
 * @effects leaves same numbers in A but with
 *   A[j] < 0 for 0 <= j < i
 *   A[j] = 0 for i <= j < k
 *   A[j] > 0 for k <= j < n
 * @returns the indexes (i, k) above
 */
function sortPosNeg(A: number[]): [number,number]
```

Example: Sorting Negative, Zero, Positive

```
// @effects leaves same numbers in A but with
//   A[j] < 0 for 0 <= j < i
//   A[j] = 0 for i <= j < k
//   A[j] > 0 for k <= j < n
```



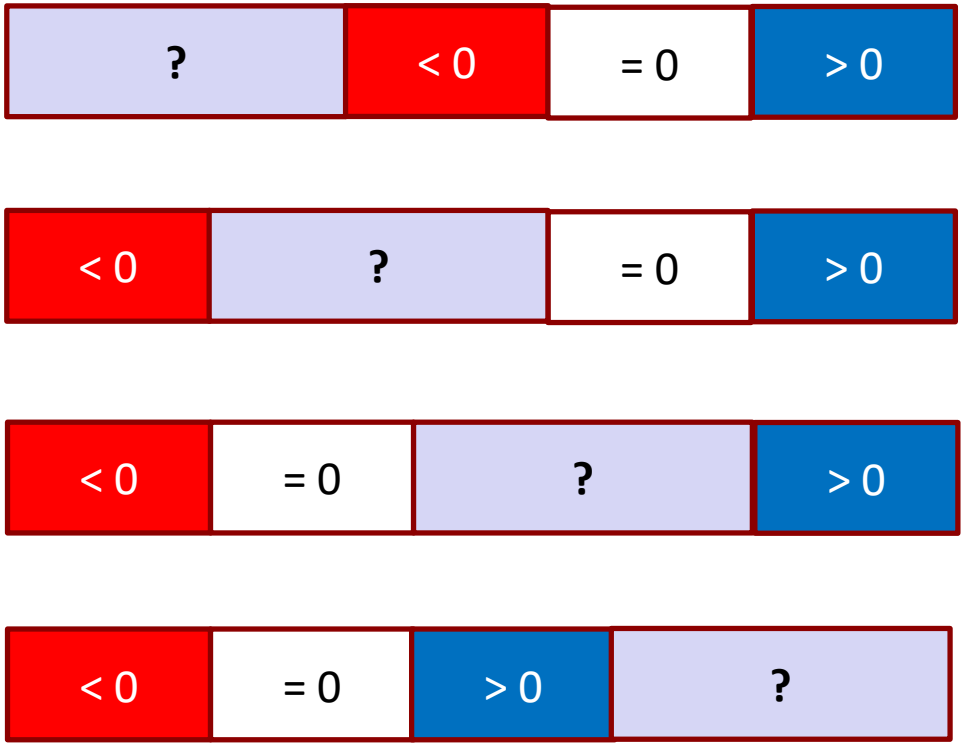
Let's implement this...

Example: Sorting Negative, Zero, Positive

How should we weaken this for the invariant?

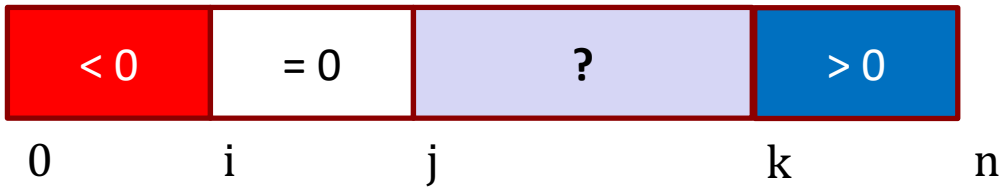
- **needs allow elements with *unknown* values**

  initially, we don't know anything about the array values



Example: Sorting Negative, Zero, Positive

Our Invariant:



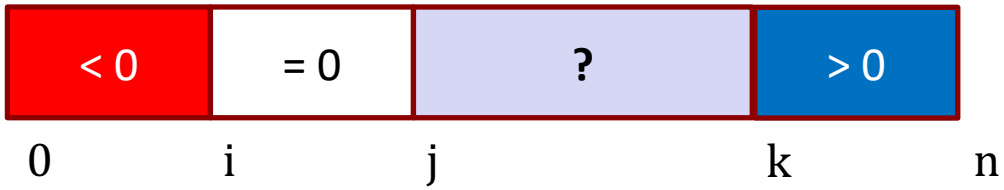
$A[\ell] < 0$ for any $0 \leq \ell < i$

$A[\ell] = 0$ for any $i \leq \ell < j$

(no constraints on $A[\ell]$ for $j \leq \ell < k$)

$A[\ell] > 0$ for any $k \leq \ell < n$

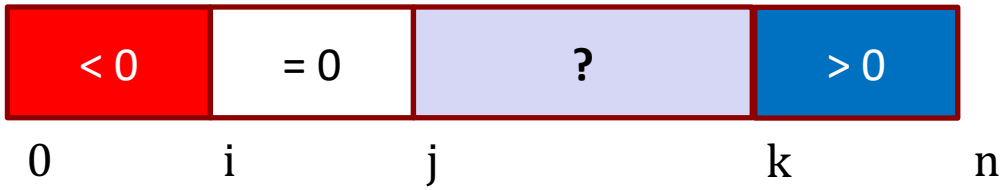
Example: Sorting Negative, Zero, Positive



- Let's try figuring out the code (problem type 2)
  - on homework, this would be type 3 (check correctness)

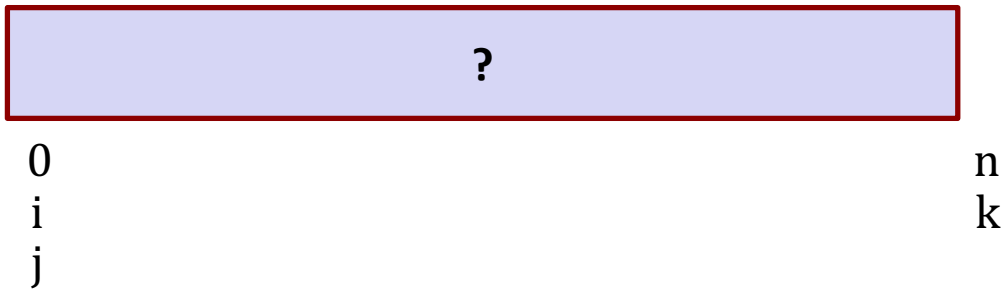
- Figure out the code for
  - how to initialize
  - when to exit
  - loop body

Example: Sorting Negative, Zero, Positive

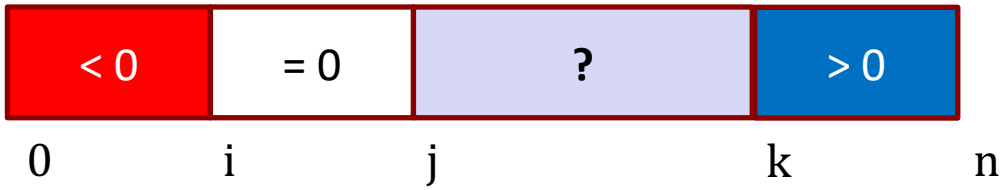


- Will have variables i, j, and k with $i \leq j < k$

- How do we set these to make it true initially?
  - we start out not knowing anything about the array values
  - set $i = j = 0$ and $k = n$

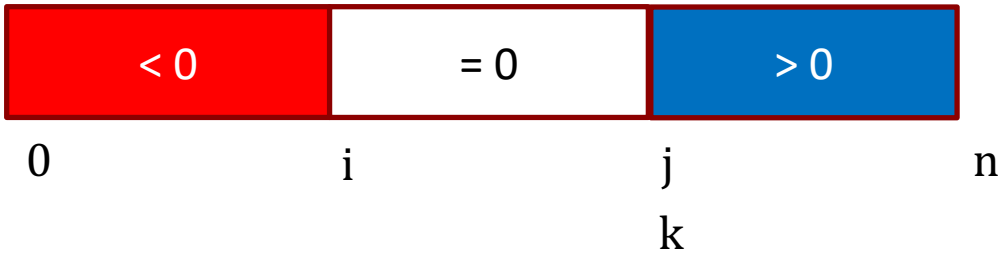


Example: Sorting Negative, Zero, Positive



- **Set $i = j = 0$ and $k = n$ to make this hold initially**

- **When do we exit?**
  - **purple is empty if $j = k$**



Sort Positive, Zero, Negative

```
let i: number = 0;
let j: number = 0;
let k: number = A.length;
```

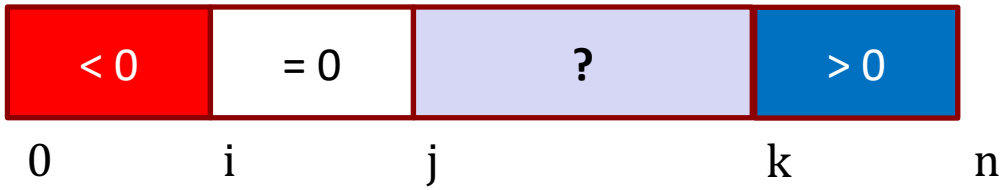
{{ **Inv**: $A[\ell] < 0$ for any $0 \leq \ell < i$ and $A[\ell] = 0$ for any $i \leq \ell < j$
$A[\ell] > 0$ for any $k \leq \ell < n$ }}

```
while (j !== k) {
  ...
}
```

{{ $A[\ell] < 0$ for any $0 \leq \ell < i$ and $A[\ell] = 0$ for any $i \leq \ell < j$
$A[\ell] > 0$ for any $j \leq \ell < n$ }}

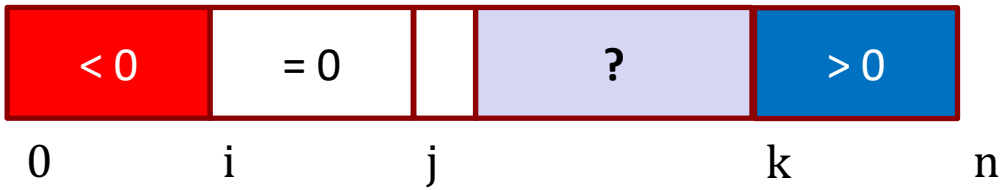
```
return [i, j];
```

Example: Sorting Negative, Zero, Positive

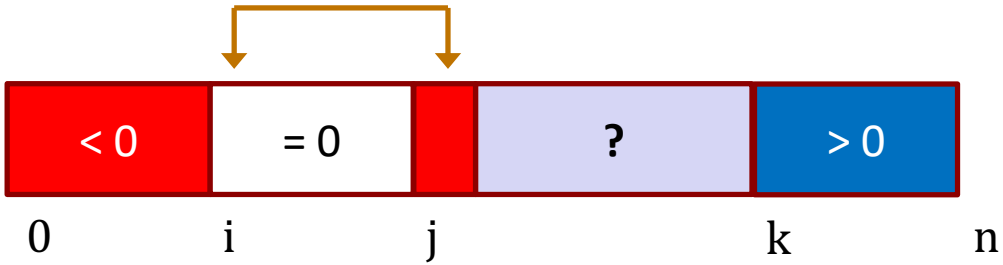


- How do we make progress?
  - try to increase j by 1 or decrease k by 1
- Look at $\mathrm{A}[\mathrm{j}]$ and figure out where it goes
- What to do depends on $\mathrm{A}[\mathrm{j}]$
  - could be < 0, $= 0$, or > 0

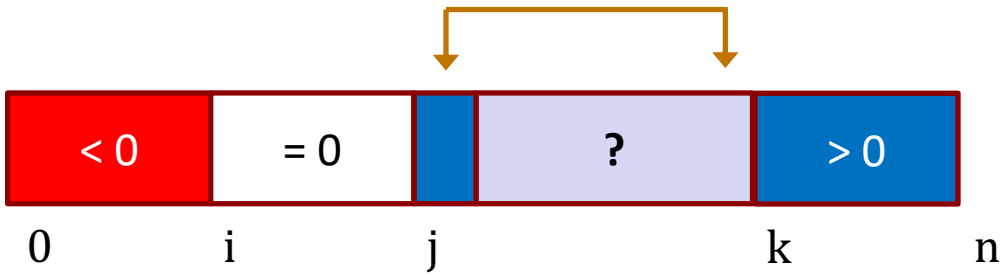
Example: Sorting Negative, Zero, Positive



Set $j = j_0 + 1$



Swap A[i] and A[j]
Set $i = i_0 + 1$
and $j = j_0 + 1$



Swap A[j] and A[k–1]
Set $k = k_0 - 1$

Sort Positive, Zero, Negative

{{ **Inv**: $A[\ell] < 0$ for any $0 \leq \ell < i$ and $A[\ell] = 0$ for any $i \leq \ell < j$
$A[\ell] > 0$ for any $k \leq \ell < n$ }}

```
while (j !== k) {
  if (A[j] === 0) {
    j = j + 1;
  } else if (A[j] < 0) {
    swap(A, i, j);
    i = i + 1;
    j = j + 1;
  } else {
    swap(A, j, k);
    k = k - 1;
  }
}
```

Combine forward and backward reasoning to double check correctness.

Sort Positive, Zero, Negative

{{ **Inv**: $A[\ell] < 0$ for any $0 \le \ell < i$ and $A[\ell] = 0$ for any $i \le \ell < j$
$A[\ell] > 0$ for any $k \le \ell < n$ }}

```
while (j !== k) {
  …
  } else if (A[j] < 0) {
```

{{ $A[\ell] < 0$ for any $0 \le \ell < i$ and $A[\ell] = 0$ for any $i \le \ell < j$
$A[\ell] > 0$ for any $k \le \ell < n$ and $A[j] < 0$ }}

```
    swap(A, i, j);
    i = i + 1;
    j = j + 1;
```

{{ $A[\ell] < 0$ for any $0 \le \ell < i$ and $A[\ell] = 0$ for any $i \le \ell < j$
$A[\ell] > 0$ for any $k \le \ell < n$ }}

```
  }
  …
```

Sort Positive, Zero, Negative

{{ **Inv**: $A[\ell] < 0$ for any $0 \le \ell < i$ and $A[\ell] = 0$ for any $i \le \ell < j$
$A[\ell] > 0$ for any $k \le \ell < n$ }}

```
while (j !== k) {
```

…

```
} else if (A[j] < 0) {
```

{{ $A[\ell] < 0$ for any $0 \le \ell < i$ and $A[\ell] = 0$ for any $i \le \ell < j$
$A[\ell] > 0$ for any $k \le \ell < n$ and $A[j] < 0$ }}

```
swap(A, i, j);
```

{{ $A[\ell] < 0$ for any $0 \le \ell < i+1$ and $A[\ell] = 0$ for any $i+1 \le \ell < j+1$
$A[\ell] > 0$ for any $k \le \ell < n$ }}

```
i = i + 1;
j = j + 1;
```

{{ $A[\ell] < 0$ for any $0 \le \ell < i$ and $A[\ell] = 0$ for any $i \le \ell < j$
$A[\ell] > 0$ for any $k \le \ell < n$ }}

```
}
```

…

Sort Positive, Zero, Negative

{{ $A[\ell] < 0$ for any $0 \le \ell < i$ and $A[\ell] = 0$ for any $i \le \ell < j$
$A[\ell] > 0$ for any $k \le \ell < n$ and $A[j] < 0$ }}

```
swap(A, i, j);
```

{{ $A[\ell] < 0$ for any $0 \le \ell < i+1$ and $A[\ell] = 0$ for any $i+1 \le \ell < j+1$
$A[\ell] > 0$ for any $k \le \ell < n$ }}

Easiest to stop here since this is a function call. (Need to use its spec.)

Step 1: What facts are new in the bottom assertion?

New facts are $A[i] < 0$ and $A[j] = 0$

Initially have $A[i] = 0$ and $A[j] < 0$

Swapping them gives what we want.

Other 2 cases are similar... (Exercise)