Solution

Firstly, we should know the following APIs and what they do:

- `Window.getComputedStyle()` which returns an object containing the values of all CSS properties of an element, after applying active stylesheets and resolving any basic computation those values may contain.
- `Element.children` which returns a live `HTMLCollection` of the child elements. We use this over `Node.childNodes` which returns a live `NodeList` of child `Node`s because `childNodes` will include non-element nodes like text and comment nodes, which are not relevant in this question.
  - However `HTMLCollection` does not have `.forEach`, so we have to iterate through it using traditional `for` loops.

`Window.getComputedStyle()` vs `Element.style`

While both APIs return `CSSStyleDeclaration`s, `getComputedStyle()` returns an object that represents the final resolved styles of an element after all styles have been applied, including styles from CSS files, inline styles, and browser defaults. The `style` property on elements allows you to access and modify inline styles directly on the element. If an element is not styled using inline styles, the values of all the keys on the `style` property is empty.

Since the function is meant to match the style rendered by the browser, `getComputedStyle()` should be used instead of `element.style`.

```
// Assuming a typical <body> element with no inline styles specified.

console.log(document.body.style.fontSize); // '' (empty string)
console.log(getComputedStyle(document.body).getPropertyValue('font-size')); //
```

What are computed styles and why are they important?

Let's take a closer look at `Window.getComputedStyle()`. According to MDN, it returns the property values **after applying active stylesheets and resolving any basic computation those values may contain**. Obtaining resolved styles is important because:

1. **Styling can be done in many ways**: There are many ways to style an element on a webpage:
   1. **Inline**: Directly within the HTML tag.

    2. **Internal**: With a `<style>` tag.
    3. **External**: Linking to a separate CSS file.
2. **Styles follow cascading and inheritance rules**: CSS works by cascading styles – rules from different sources combine and potentially override each other. Elements can also inherit styles from their parent elements.
3. **Multiple ways of defining styles properties**: Other than using the raw final values when styling elements:
    1. `color: white` `color: #fff` `color: rgb(255, 255, 255)` t color.
    2. `margin-top: 10px`
    3. **CSS variables**: Properties can also be written using CSS variables or more officially known as CSS custom properties, e.g. `color: var(--text-color)`. The final color value is not known until the browser resolves the value of the `--text-color` variable.
4. **Styles have to be resolved**: The `getComputedStyle()` API gives you a snapshot of the final, calculated styles applied to an element after all the cascading and inheritance rules have been applied. This is incredibly valuable because it reflects how the element is actually rendered in the browser.

The implication of using `Window.getComputedStyle()` is that we can only match based on the element's resolved values. Font sizes, paddings, margins, can be defined using `px`, `rem`, `em`, etc but the resolved value unit for these properties obtained from `getComputedStyle()` is `px`. Colors can be defined using named colors, HSL, RGB (and more) formats but the resolved style format for colors is RGB hexadecimal. This is a limitation that you should mention during your interviews if you have the opportunity.

```
element.style.color = 'white';
console.log(getComputedStyle(element).getPropertyValue('color')); // 'rgb(255, 255, 255)'
```

While it is possible to write your own conversion/resolution logic within your `getElementsByStyle()` function so that the value argument is resolved before comparing against the element's resolved styles, it is only achievable for certain properties that do not rely on properties of other elements. Properties like `inherit`, `rem` which rely on properties of other elements due to CSS cascading cannot be matched easily and accurately.

Recursive traversal

Recursive traversal

Both depth-first traversal and breadth-first traversals can be used but it's easier to write depth-first traversals using recursions. When visiting each element:

1. Call `getComputedStyle(element)` to get a `CSSStyleDeclaration` object containing the resolved styles of the specified element. `getPropertyValue(property)` is used to retrieve the value of a specific CSS property such as `color`, `font-size`, `margin`, etc. Then check against the property and value arguments.
2. Recursively traverse the element's children via `element.children`.

Maintain an `elements` array to collect the matching elements while recursively traversing starting from the root element. Remember that the root element argument itself is not included in the results and it should not be added to the `elements` array.

JavaScript TypeScript

```
/**
 * @param {Element} element
 * @param {string} property
 * @param {string} value
 * @return {Array<Element>}
 */
export default function getElementsByStyle(element, property, value) {
  const elements = [];

  function traverse(el) {
    if (el == null) {
      return;
    }

    const computedStyles = getComputedStyle(el);
    if (computedStyles.getPropertyValue(property) === value) {
      elements.push(el);
    }

    for (const child of el.children) {
      traverse(child);
    }
  }

  for (const child of element.children) {
```

```
    traverse(child);
  }

  return elements;
}
```

Edge cases

- Element argument is not included in the results even if it matches the specified style.
- Value arguments have to be in the format of the resolved styles, e.g. colors have to be in `rgb()` / `rgba()` format in order to match.

Techniques

- Recursion
- DOM APIs
  - How to get an `Element`'s computed style
  - How to traverse an `Element`'s children