# JavaScript Word Break II

## Challenge

Given a string `s` and a dictionary of strings `wordDict`, add spaces in `s` to construct a sentence where each word is a valid dictionary word. Return all such possible sentences in any order.

> **Note**
>
> The same word in the dictionary may be reused multiple times in the segmentation.

### 1st Example

```
Input: s        = 'catsanddog',
       wordDict = ['cat','cats','and','sand','dog']
Output: ['cats and dog','cat sand dog']
```

### 2nd Example

```
Input: s        = 'pineapplepenapple',
       wordDict = ['apple','pen','applepen',
                   'pine','pineapple']
Output: ['pine apple pen apple',
         'pineapple pen apple','pine applepen apple']
Explanation: You are allowed to reuse a dictionary word.
```

## 3rd Example

```
Input: s        = 'catsandog',
       wordDict = ['cats','dog','sand','and','cat']
Output: []
```

## Constraints

- `1 <= s.length <= 20`
- `1 <= wordDict.length <= 1000`
- `1 <= wordDict[i].length <= 10`
- `s` and `wordDict[i]` consist of only lowercase English letters.
- All the strings of `wordDict` are unique.
- Input is generated in a way that the length of the answer doesn't exceed $10^5$.

## Solution

```
const wordBreak = (s, wordDict) => {
    let from = [];

    from[0] = [0];
    wordDict = new Set(wordDict);

    for (let i = 1; i<=s.length; i++) {
        from[i] = [];

        for (let j=0; j<i;j++) {
            if (from[j].length) {
                let mySubstr = s.substring(j, i);
```

Solution continues on next page...

```
                if (wordDict.has(mySubstr)) {
                    from[i].push(j);
                }
            }
        }
    }

    let res = [];

    const build = (idx, suffix) => {
        if (idx === 0) {
            return res.push(suffix);
        }

        from[idx].forEach((startsAtIndex) => {
            let mySubstr = s.substring(startsAtIndex, idx);

            if (suffix === '') {
                build(startsAtIndex, mySubstr);
            } else {
                build(startsAtIndex,
                      mySubstr + ' ' + suffix);
            }
        })
    }

    build(s.length, '');

    return res;
};
```

## Explanation

I've built a function called `wordBreak` that takes in a string `s` and an array `wordDict`. The purpose of this function is to find all possible combinations of words from `wordDict` that can be formed by splitting the string `s` into multiple words.

Inside the function, an empty array called `from` is initialized to keep track of the starting indices of substrings that can form words from `wordDict`. The first element of `from` is set to be an array containing only the index `0`, indicating that the first character of `s` can be a word itself.

The `wordDict` array is converted into a `Set` for faster lookup. A loop is then used to iterate from `1` to the length of the string `s`. At each index `i`, an empty array `from[i]` is initialized.

Another loop is used to iterate from `0` to `i-1`. For each index `j`, it checks if `from[j]` is not empty. If it is not empty, a substring `mySubstr` is created from `s` starting at index `j` and ending at index `i`. It then checks if `mySubstr` exists in the `wordDict` Set. If it does, the index `j` is added to `from[i]`, indicating that a word can be formed from index `j` to index `i`.

After the nested loops, an empty array called `res` is initialized to store the resulting combinations of words.

A recursive function called `build` is then defined, which takes in an index `idx` and a string `suffix`. If the index `idx` is `0`, it means we have reached the beginning of the string, so the `suffix` is pushed to the `res` array.

For each starting index `startsAtIndex` in `from[idx]`, a substring `mySubstr` is created from `s` starting at `startsAtIndex` and ending at `idx`. If the `suffix` is an empty string, the `build` function is recursively called with `startsAtIndex` as the new index and `mySubstr` as the new suffix. If the `suffix` is not empty, the `build` function is recursively called with `startsAtIndex` as the new index and `mySubstr + ' ' + suffix` as the new suffix.

After the recursive calls, the `build` function is called with `s.length` as the initial index and an empty string as the initial suffix.

Finally, the function returns the `res` array containing all possible combinations of words formed from `wordDict` in the string `s`.