139. Wor

下記のコードにおいて、s.length = N, max(wordDict.length) = M, wordDict.length = W とした場合、

buildTrieTreeの時間計算量はO(MW)、findWordsの時間計算量がO(MW)、wordBreakのwhileの部分は、sの各文字事にfindWordsを実行しているので、最終的な時間計算量はO(NMW) であっていますか

```
type Trie = {
    child: {
        [key: string]: Trie;
    },
    eow: boolean;
}
function wordBreak(s: string, wordDict: string[]): boolean {
    const wordTree: Trie = buildTrieTree(wordDict);
    const reached: boolean[] = new Array(s.length + 1).fill(false);

    const q: number[] = [0];
    while (q.length > 0) {
        const i = q.shift()!;
        if (reached[i]) {
            continue;
        }
        reached[i] = true;

        const words = findWords(wordTree, s, i);
        for (const word of words) {
            q.push(i + word.length);
        }
    }

    return reached[s.length];
};
function findWords(root: Trie, s: string, start: number): string[] {
    const words: string[] = [];
    let parent = root;
    let path: string[] = [];
    for (let i = start; i < s.length; i++) {
        const char = s[i];
        if (!(char in parent.child)) {
            break;
        }
        parent = parent.child[char];
        path.push(char);
        if (parent?.eow) {
            words.push(path.join(''));
```

```
      }
    }
    return words;
}

function buildTrieTree(wordDict: string[]): Trie {
    const root: Trie = {
        child: {},
        eow: false,
    };

    for (const word of wordDict) {
        let parent = root;
        for (const char of word) {
            if (!(char in parent.child)) {
                parent.child[char] = {
                    child: {},
                    eow: false,
                };
            }
            parent = parent.child[char];
        }
        parent.eow = true;
    }

    return root;
}
```

結論から言うと、ご質問にある「最終的な時間計算量は O(N M W) になる」という分析は**必ずしも正しくありません**。少なくとも提示された実装を見る限り、`findWords` 呼び出し部分で「全単語数 W」がそのまま掛かるような処理にはなっていないため、BFS 部分が W を含むオーダーになるわけではありません。

以下では、

- $N = s.\text{length}$（入力文字列の長さ）
- $W = \text{wordDict.length}$（単語数）
- $M = \max(\text{各単語の長さ})$

という前提で、各関数の計算量を整理します。

---

1. `buildTrieTree` の計算量

```ts
function buildTrieTree(wordDict: string[]): Trie {
    const root: Trie = { child: {}, eow: false };

    for (const word of wordDict) {
        let parent = root;
```

```
            for (const char of word) {
                if (!(char in parent.child)) {
                    parent.child[char] = {
                        child: {},
                        eow: false,
                    };
                }
                parent = parent.child[char];
            }
            parent.eow = true;
        }

        return root;
    }
```

- 単語数 W のそれぞれについて、最大長さ M ぶん Trie に挿入している。
- 1 文字挿入するごとに子ノードをたどる処理はおおむね $O(1)$ (連想配列のキーアクセス) と考えられる。

よってビルド処理全体の計算量は

$$O(M \times W)$$

で問題ありません。

---

2. `findWords` の計算量

```ts
ts


    function findWords(root: Trie, s: string, start: number): string[] {
        const words: string[] = [];
        let parent = root;
        let path: string[] = [];
        for (let i = start; i < s.length; i++) {
            const char = s[i];
            if (!(char in parent.child)) {
                break;
            }
            parent = parent.child[char];
            path.push(char);
            if (parent?.eow) {
                words.push(path.join(''));
            }
        }
        return words;
    }
```

- `start` から始まり、`s[i]` をキーとして Trie をたどります。
- **最大でも**「単語の最大長 M」ぶんしかたどれません**（もっと長い単語があっても、途中で `child` が途切れるか、または `s` の末尾に到達するから）」。
- ループ自体は「最長で M 回」になりますが、各ループで `path.join('')` をしているところがやや重い操作になり、そこを素直に評価すると 1 回あたり $O(\text{現在の長さ}) \to O(M)$ となる可能性があります。

したがって、

- 単純に「Trie をたどるだけ」という観点で見るなら $O(M)$
- 文字列連結（`path.join('')`）のコストまで細かく考慮すると $O(M^2)$
  くらいが妥当です。

ポイントは、この関数内に「単語数 W」に応じたループがないことです。
Trie によって「文字列を 1 文字ずつたどる」形になっており、「全単語を総当たりでチェックする」という実装にはなっていないため、ここに W がそのまま掛かることはありません。

---

3. `wordBreak` の計算量

```ts
function wordBreak(s: string, wordDict: string[]): boolean {
    const wordTree: Trie = buildTrieTree(wordDict);
    const reached: boolean[] = new Array(s.length + 1).fill(false);

    const q: number[] = [0];
    while (q.length > 0) {
        const i = q.shift()!;
        if (reached[i]) {
            continue;
        }
        reached[i] = true;

        const words = findWords(wordTree, s, i);
        for (const word of words) {
            q.push(i + word.length);
        }
    }

    return reached[s.length];
};
```

3.1 BFS (while ループ) 全体の挙動

- キュー `q` に、切り分けが可能なインデックス（次に探索すべき文字列の始点）を入れて BFS しているイメージです。
- `i` は最大で 0 から N（＝`s.length`）まであります。
- `reached[i]` がいったん `true` になると二度と処理しないので、**同じインデックス i を重複して探索することはありません。**
  つまり、`while` ループの本体が呼ばれるのは高々 $N+1$ 回です。

3.2 BFS 1 回ごとのコスト

- `const words = findWords(wordTree, s, i);` を呼び出すところがメインのコストです。
- 上で見たように `findWords` は $O(M) \sim O(M^2)$ 程度。

よって、BFS 全体のコストは大まかに言うと

$$O(N) \times O(\text{findWords})$$

ですから、

- `findWords` を $O(M)$ とみなせば、BFS は $O(N \times M)$
- `findWords` の文字列連結コストを考慮して $O(M^2)$ とみなせば、BFS は $O(N \times M^2)$

と表せます。いずれにせよ、**「BFS のループで W が掛かる」ような実装にはなっていない**ので、ここに W は出てきません。

---

4. 全体の時間計算量

以上をまとめると、全体の計算量は

1. Trie 構築: $O(MW)$
2. BFS（`wordBreak` のメイン部分）: 最大で $O(N \times M^2)$ 程度

したがって合計としては

$$O(MW) + O(N \times M^2)$$

くらいになると考えるのが自然です。

- 「最終的に $O(N \cdot M \cdot W)$ になる」というのは、`findWords` が各呼び出しで「全単語数 W」ぶんの処理をしているか、もしくは BFS が「各インデックスごとに全単語に相当する探索をする」ような場合に出てくる表現です。しかし今回のコードでは **Trie により単語探索を 1 文字ずつたどる作り** になっており、実際に「全単語を総当たり」でチェックしているわけではありません。
- そのため、BFS 側で W が乗算されることはなく、$O(N \cdot M \cdot W)$ というオーダーにはなりません。

---

5. まとめ

- **`buildTrieTree` の計算量**: $O(MW)$ は正しい
- **`findWords` の計算量**:
  - せいぜい「最大 M 回のループ」＋「文字列連結」コストを考慮すると $O(M^2)$ 程度
  - 「全単語数 W」ぶんかかるわけではないので $O(MW)$ ではない
- **`wordBreak` (BFS 部) の最終計算量**: 1 インデックスにつき最大 $O(M^2)$ を呼ぶので $O(N \times M^2)$ 程度

以上より、ご質問にあるような「最終的に $O(N \cdot M \cdot W)$ になる」という結論は**誤り**で、正しくは

$$O(MW) + O(N \times M^2)$$

程度と見なすのが妥当です。実装のロジック上、BFS 部分に W が乗ってくる構造にはなっていない点にご注意ください。