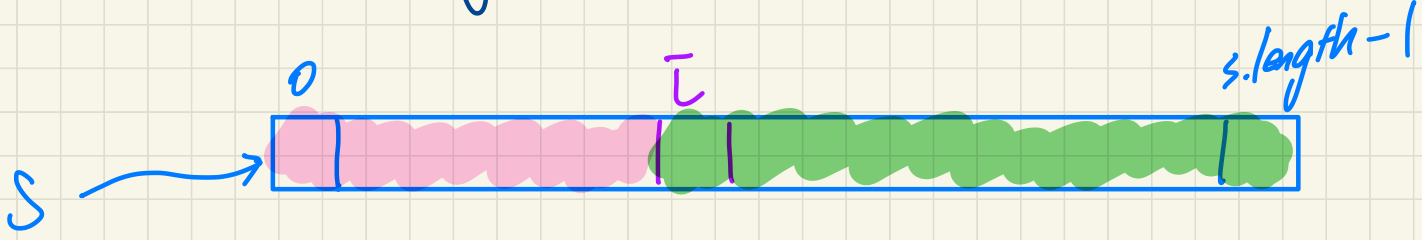
For any non-null string s,

given int. i s.t. $0 \leq i \leq s.length - 1$

$s.equals(s.substring(0, i) + s.substring(i, s.length()))$



Recursions on Strings

ex "abcd"

Palindrome

"racecar"

"aracecars"

"raceacar"

isP("racecar")

↳ r == r && isP("aceca")

Reversal

"abcd"

Number of Occurrences

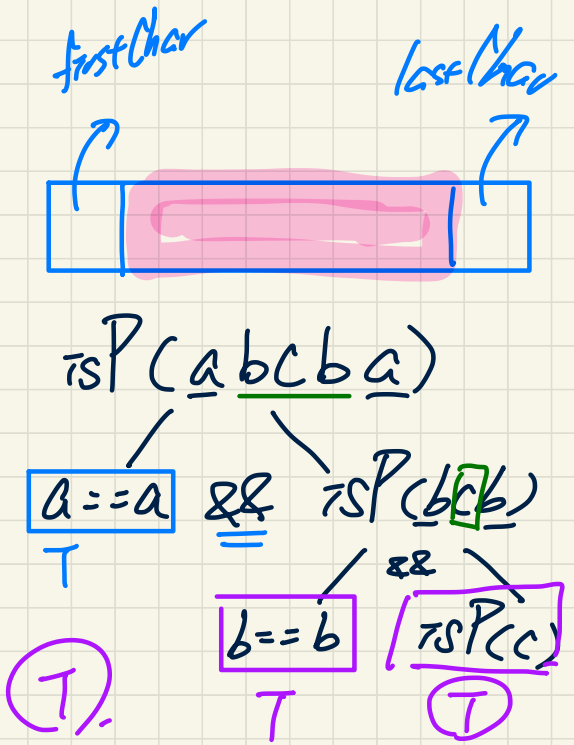
"abca"

'a'

'b'

Problem: Palindrome

```
boolean isPalindrome (String word) {
  if(word.length() == 0 || word.length() == 1) {
    /* base case */
    return true;
  }
  else {
    /* recursive case */
    char firstChar = word.charAt(0);
    char lastChar = word.charAt(word.length() - 1);
    String middle = word.substring(1, word.length() - 1);
    return
        firstChar == lastChar
        /* See the API of java.lang.String.substring. */
        && isPalindrome (middle);
  }
}
```



ex. isP(racecar)

ex. isP(racebar)

Recursions on Strings

ex.

occ("abca", 'a')

Palindrome

isP("racecar")

↳ r == r && isP("aceca")

"racecar"

"aracecars"

"raceacar"

Reversal

reverseOf("abcd") ← strictly smaller subproblem

reverseOf(bcd) + "a"

dcb

Number of Occurrences

occ("abca", 'a')

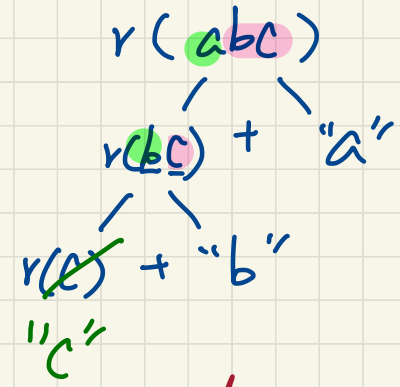
↳ 1 + occ(bca)

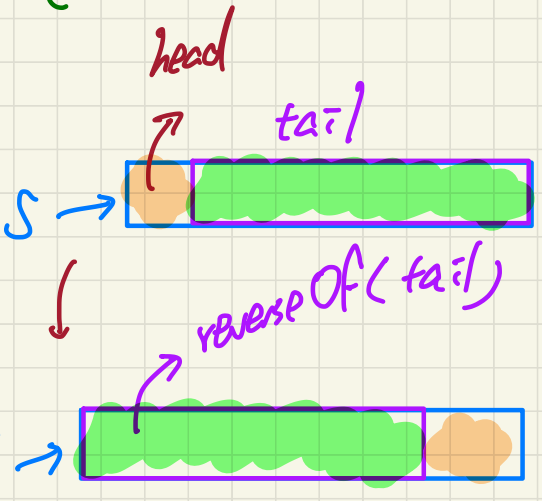
occ("abca", 'b')

↳ 0 + occ(bca)

Problem: Reverse of a String

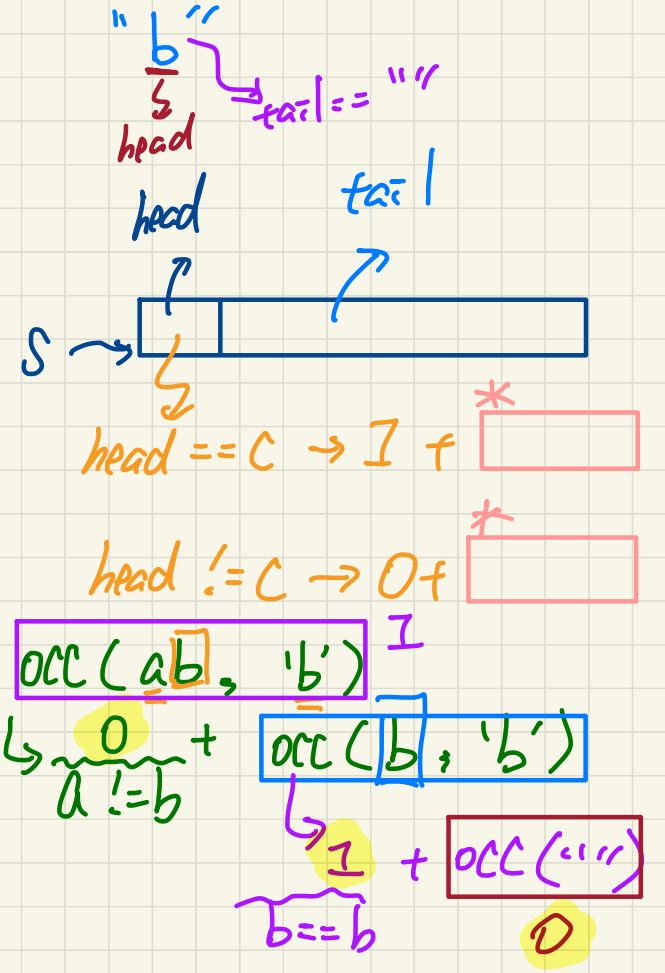
```
String reverseOf (String s) {
  if(s.isEmpty()) { /* base case 1 */
    return "";
  }
  else if(s.length() == 1) { /* base case 2 */
    return s;
  }
  else { /* recursive case */
    String tail = s.substring(1, s.length());
    String reverseOfTail = reverseOf(tail);
    char head = s.charAt(0);
    return reverseOfTail + head;
  }
}
```





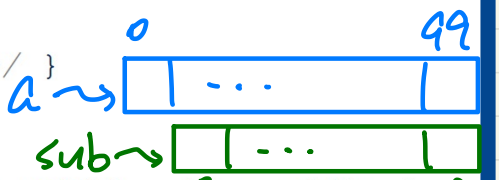
Problem: Number of Occurrences

```
int occurrencesOf(String s, char c) {
  if(s.isEmpty()) {
    /* Base Case */
    return 0;
  }
  else {
    /* Recursive Case */
    char head = s.charAt(0);
    String tail = s.substring(1, s.length());
    if(head == c) {
      return 1 + occurrencesOf(tail, c);
    }
    else {
      return 0 + occurrencesOf(tail, c);
    }
  }
}
```



Recursion on an Array: Passing new Sub-Arrays

```
void m(int[] a) {
  if(a.length == 0) { /* base case */ }
  else if(a.length == 1) { /* base case */ }
  else {
    int[] sub = new int[a.length - 1];
    for(int i = 1; i < a.length; i ++) { sub[i - 1] = a[i]; }
    m(sub) } }
```

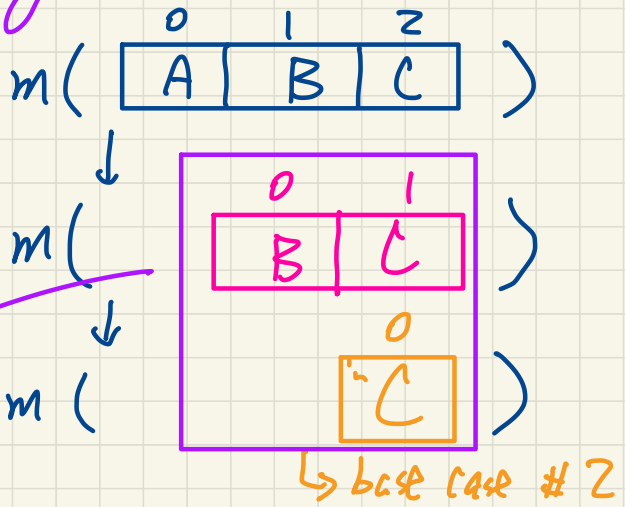


recursive call on a new, strictly smaller array

Say a1 = {}, consider m(a1)

↳ base case # 1

Say a2 = {A, B, C}, consider m(a2)



Recursion on an Array: Passing Same Array Reference

```
void m(int[] a, int from, int to) {
  if(from > to) { /* base case */ }
  else if(from == to) { /* base case */ }
  else { m(a, from + 1, to) } }
```

for non-empty range: from ≤ to

to - from + 1

form a range of elements to look at in the current method call

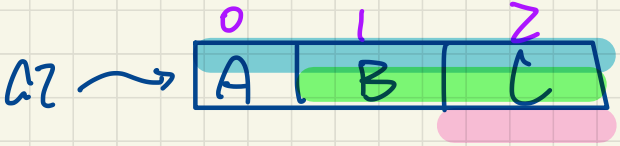
call by value

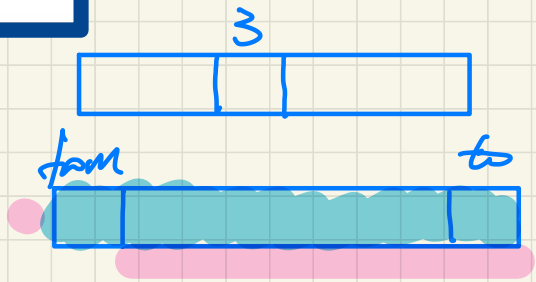
0 - 1 = -1

Say a1 = {}, consider m(a1, 0, a1.length - 1)

↳ base case # 1

Say a2 = {A, B, C}, consider m(a2, 0, a2.length - 1)





m(a2, 0, 2)

m(a2, 1, 2)

base case # 2

m(a2, 2, 2)

Problem: Are All Numbers Positive?

Say a = {}

Say a = {4}

Say a = {4,7,3,9}

Say a = {5,3,-2,9}

$\forall x \cdot P(x) \equiv True$] there's no witness to show some violation

$x \in \phi$

$\exists x \cdot P(x) \equiv False$] there's no witness to show some satisfaction

$x \in \phi$

allP(a, 0, 3) — a.length - 1

↳ a[0] > 0 && allP(a, 1, 3)

if a[0] ≤ 0 is by pass RHS

Problem: Are All Numbers Positive?

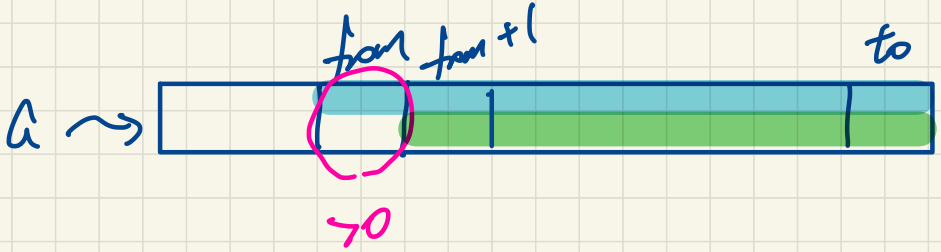
```
boolean allPositive(int[] a) {
  return allPositiveHelper(a, 0, a.length - 1);
}

boolean allPositiveHelper(int[] a, int from, int to) {
  if (from > to) { /* base case 1: empty range */
    return true;
  }
  else if(from == to) { /* base case 2: range of one element */
    return a[from] > 0;
  }
  else { /* recursive case */
    return a[from] > 0 && allPositiveHelper(a, from + 1, to);
  }
}
```

public

private

recursive helper method



Tracing Recursion: allPositive

Say a = {}

allPositive(a)

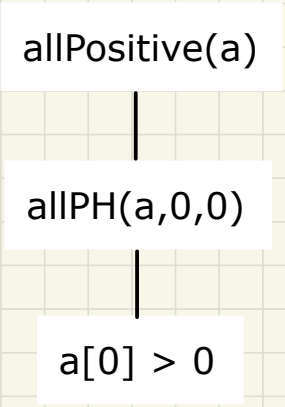
allPH(a,0,-1)

```
boolean allPositive(int[] a) {
  return allPositiveHelper(a, 0, a.length - 1);
}

boolean allPositiveHelper(int[] a, int from, int to) {
  if (from > to) { /* base case 1: empty range */
    return true;
  }
  else if(from == to) { /* base case 2: range of one element */
    return a[from] > 0;
  }
  else { /* recursive case */
    return a[from] > 0 && allPositiveHelper(a, from + 1, to);
  }
}
```

Tracing Recursion: allPositive

Say a = {4}

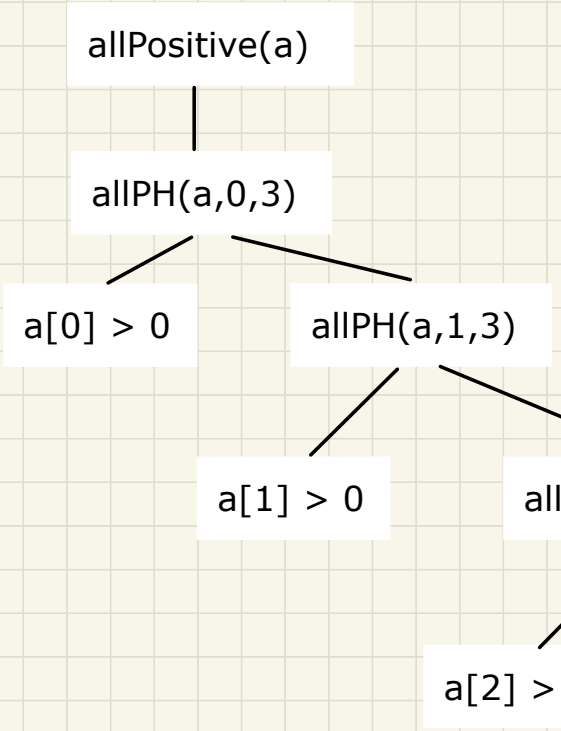



```
boolean allPositive(int[] a) {
  return allPositiveHelper(a, 0, a.length - 1);
}

boolean allPositiveHelper(int[] a, int from, int to) {
  if (from > to) { /* base case 1: empty range */
    return true;
  }
  else if(from == to) { /* base case 2: range of one element */
    return a[from] > 0;
  }
  else { /* recursive case */
    return a[from] > 0 && allPositiveHelper(a, from + 1, to);
  }
}
```

Tracing Recursion: allPositive

Say a = {4,7,3,9}

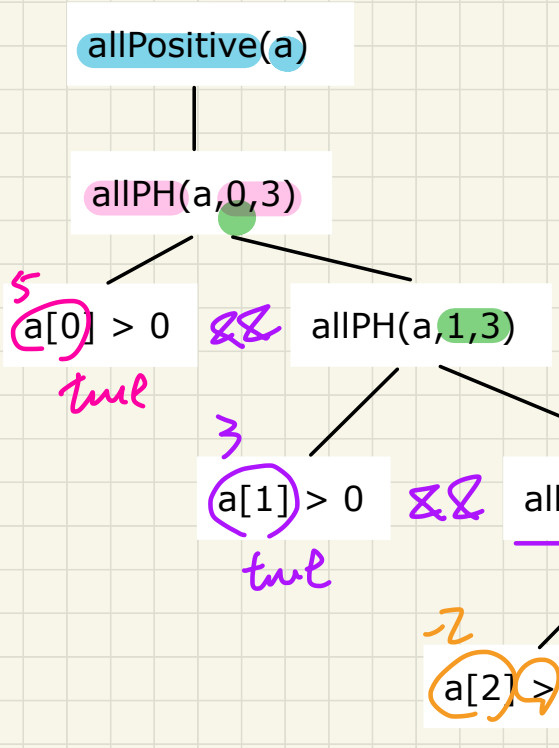



```
boolean allPositive(int[] a) {
  return allPositiveHelper(a, 0, a.length - 1);
}

boolean allPositiveHelper(int[] a, int from, int to) {
  if (from > to) { /* base case 1: empty range */
    return true;
  }
  else if(from == to) { /* base case 2: range of one element */
    return a[from] > 0;
  }
  else { /* recursive case */
    return a[from] > 0 && allPositiveHelper(a, from + 1, to);
  }
}
```

Tracing Recursion: allPositive

Say a = {5,3,-2,9}

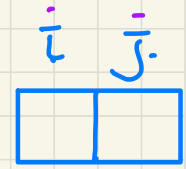



```
boolean allPositive(int[] a) {
  return allPositiveHelper(a, 0, a.length - 1);
}

boolean allPositiveHelper(int[] a, int from, int to) {
  if (from > to) { /* base case 1: empty range */
    return true;
  }
  else if(from == to) { /* base case 2: range of one element */
    return a[from] > 0;
  }
  else { /* recursive case */
    return a[from] > 0 && allPositiveHelper(a, from + 1, to);
  }
}
```

non-ascending ?

non-descending



$$\equiv \neg\left(a[i] \geq a[j]\right)$$

$$\equiv a[i] \leq a[j]$$

Problem: Are Numbers Sorted?

Say a = {}

Say a = {4}

Say a = {3,6,6,7}

Say a = {3,6,5,7}

isSorted(a, 0, 3)

from

a[0] ≤ a[1] && isSorted(a, 1, 3)

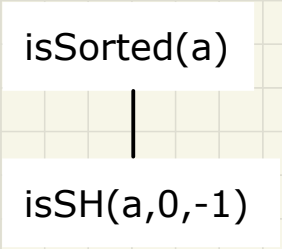
Problem: Are Numbers Sorted?

```
boolean isSorted(int[] a) {
  return isSortedHelper(a, 0, a.length - 1);
}

boolean isSortedHelper(int[] a, int from, int to) {
  if (from > to) { /* base case 1: empty range */
    return true;
  }
  else if(from == to) { /* base case 2: range of one element */
    return true;
  }
  else {
    return a[from] <= a[from + 1]
      && isSortedHelper(a, from + 1, to);
  }
}
```

Tracing Recursion: isSorted

Say a = {}

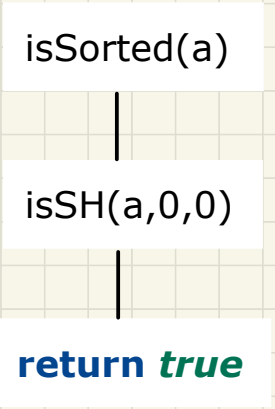



```
boolean isSorted(int[] a) {
  return isSortedHelper(a, 0, a.length - 1);
}

boolean isSortedHelper(int[] a, int from, int to) {
  if (from > to) { /* base case 1: empty range */
    return true;
  }
  else if(from == to) { /* base case 2: range of one element */
    return true;
  }
  else {
    return a[from] <= a[from + 1]
      && isSortedHelper(a, from + 1, to);
  }
}
```

Tracing Recursion: isSorted

Say a = {4}

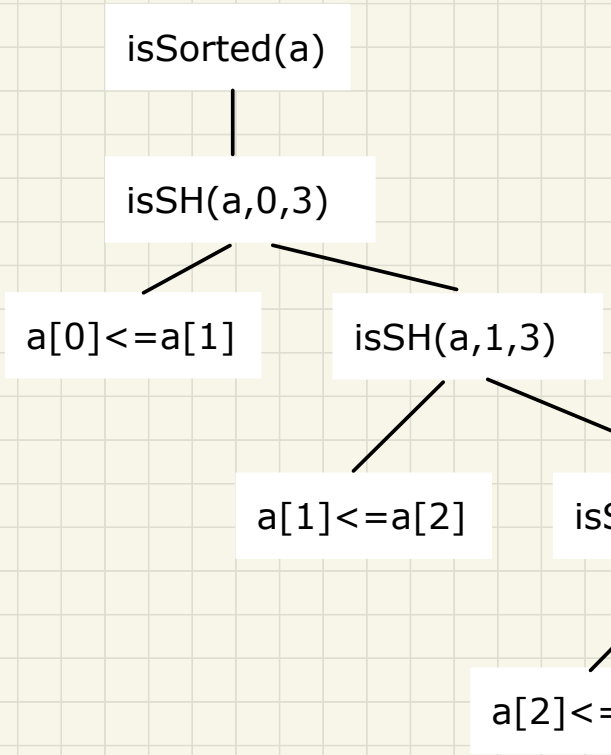



```
boolean isSorted(int[] a) {
  return isSortedHelper(a, 0, a.length - 1);
}

boolean isSortedHelper(int[] a, int from, int to) {
  if (from > to) { /* base case 1: empty range */
    return true;
  }
  else if(from == to) { /* base case 2: range of one element */
    return true;
  }
  else {
    return a[from] <= a[from + 1]
      && isSortedHelper(a, from + 1, to);
  }
}
```

Tracing Recursion: isSorted

Say a = {3,6,6,7}

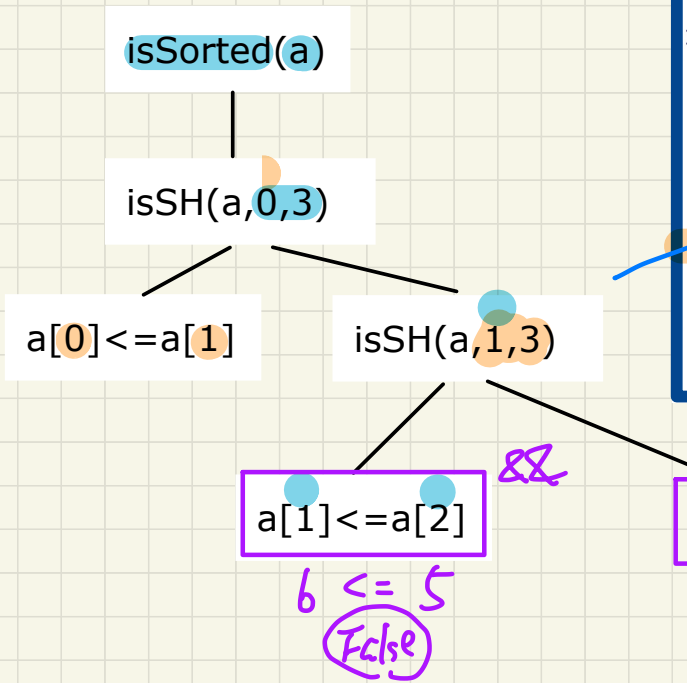



```
boolean isSorted(int[] a) {
  return isSortedHelper(a, 0, a.length - 1);
}

boolean isSortedHelper(int[] a, int from, int to) {
  if (from > to) { /* base case 1: empty range */
    return true;
  }
  else if(from == to) { /* base case 2: range of one element */
    return true;
  }
  else {
    return a[from] <= a[from + 1]
      && isSortedHelper(a, from + 1, to);
  }
}
```

Tracing Recursion: isSorted

Say a = {3,6,5,7}




```
boolean isSorted(int[] a) {
  return isSortedHelper(a, 0, a.length - 1);
}

boolean isSortedHelper(int[] a, int from, int to) {
  if (from > to) { /* base case 1: empty range */
    return true;
  }
  else if(from == to) { /* base case 2: range of one element */
    return true;
  }
  else {
    return a[from] <= a[from + 1]
      && isSortedHelper(a, from + 1, to);
  }
}
```