# Problem: Are All Numbers Positive?

Say a = {}

Say a = {4}

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

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

# 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);
  }
}
```

# Tracing Recursion: allPositive

Say a = {}

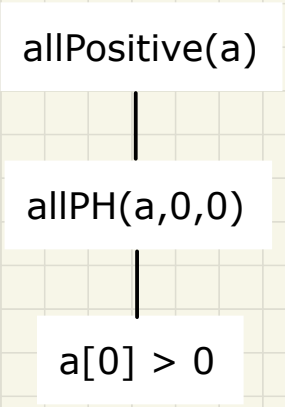
allPositive(a)

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

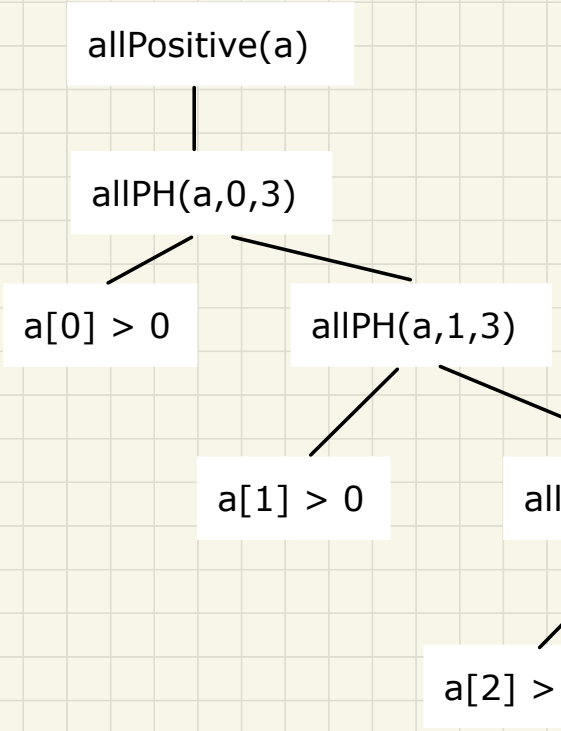
Say a = {4}




```
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

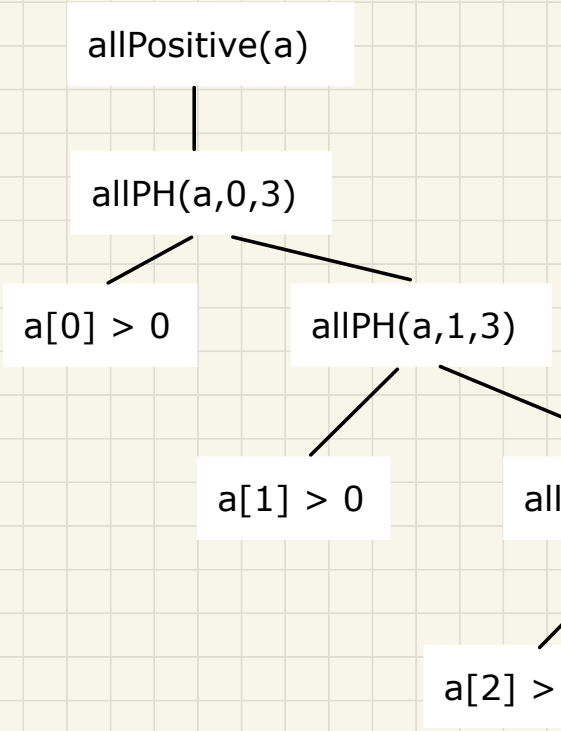
Say a = {4,7,3,9}




```
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);
  }
}
```

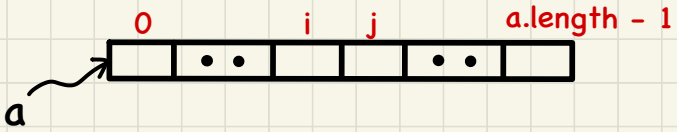
# Tracing Recursion: allPositive

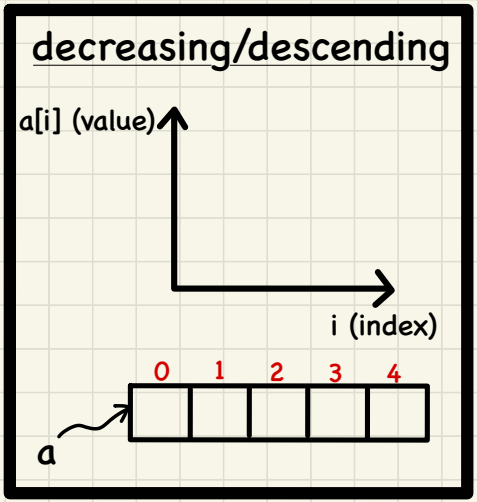
Say a = {5,3,-2,9}

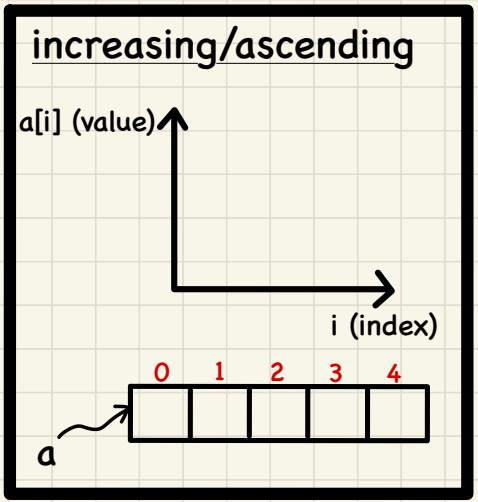


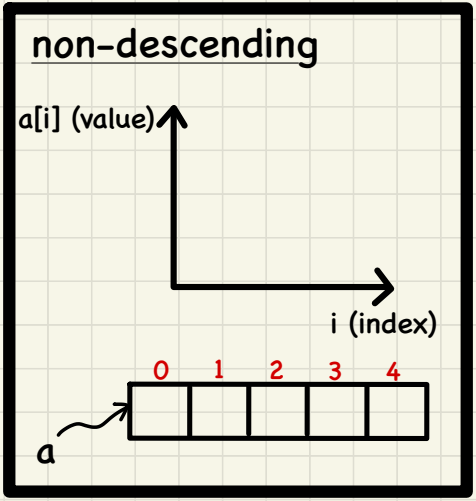

```
boolean allPositive(int[] a) {
  return allPositiveHelper(a, 0, a.length - 1);
}

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);
  }
}
```

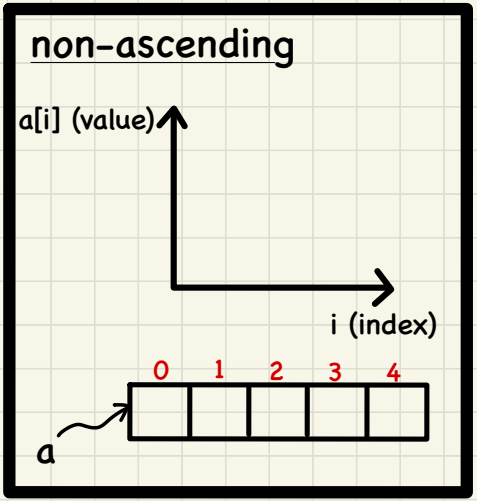
# Sorting Orders of Arrays



# Problem: Are Numbers Sorted?

Say a = {}

Say a = {4}

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

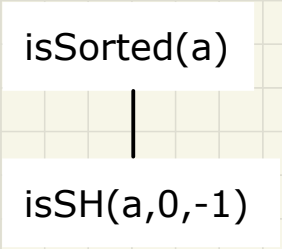
Say a = {3,6,5,7}

# 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

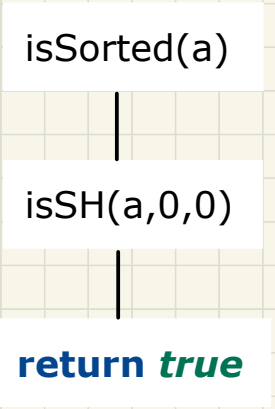
Say a = {}




```
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

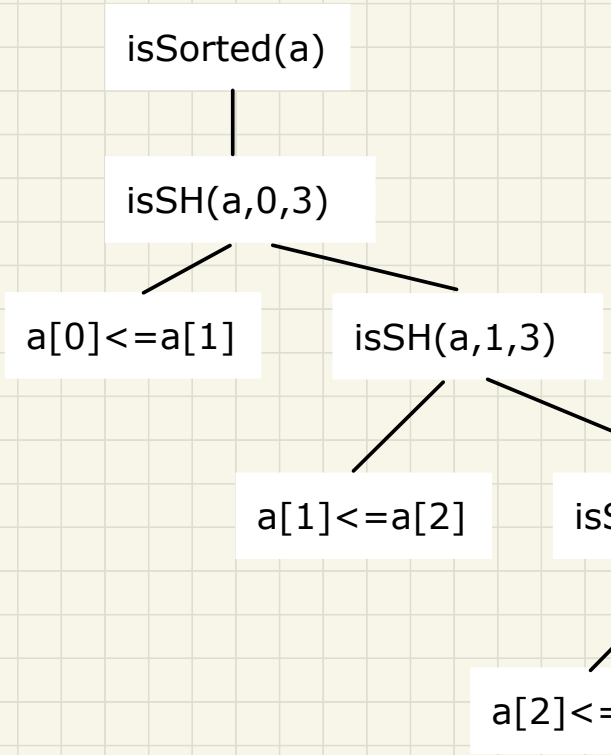
Say a = {4}




```
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

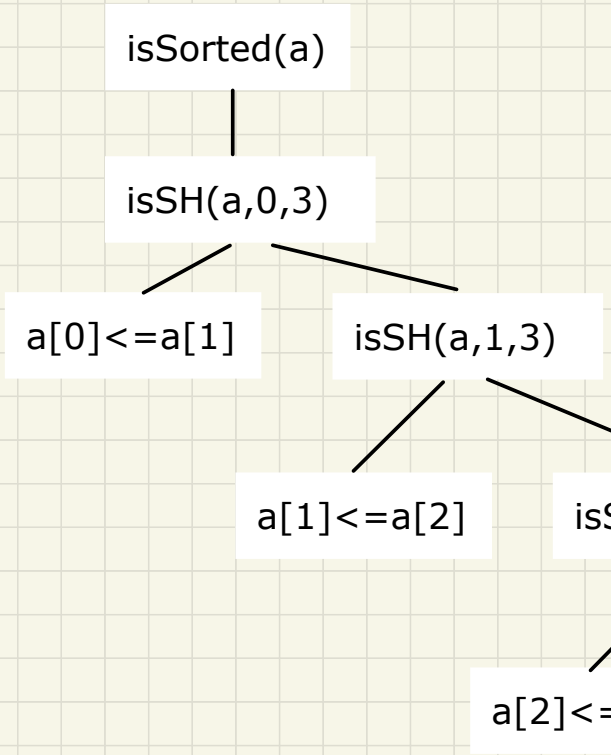
Say a = {3,6,6,7}




```
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);
  }
}
```

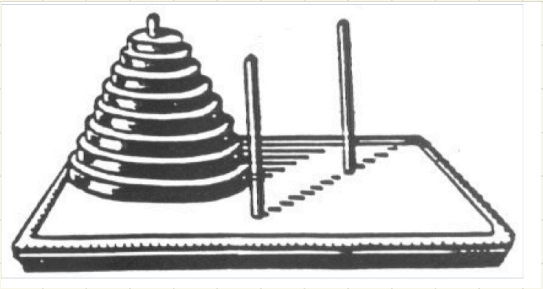
# Tracing Recursion: isSorted

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



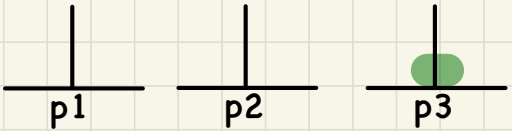

```
boolean isSorted(int[] a) {
  return isSortedHelper(a, 0, a.length - 1);
}

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);
  }
}
```

# Tower of Hanoi: Strategy



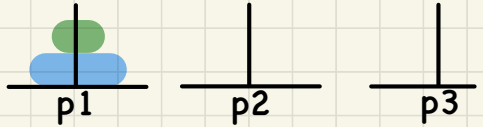
## Consider 1 disk: A

move from p1 to p3



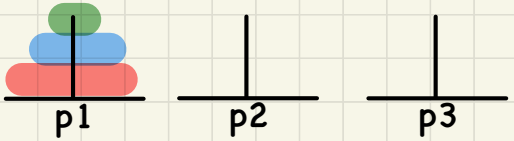

## Consider 2 disks: A < B

move from p1 to p3

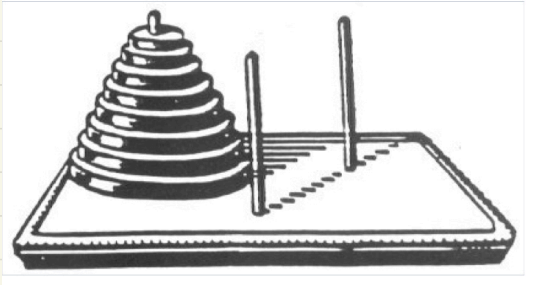



## Consider 3 disks: A < B < C

move from p1 to p3




# Tower of Hanoi: Strategy



Consider 3 disks A < B < C

move from p1 to p3

move from p1 to p2

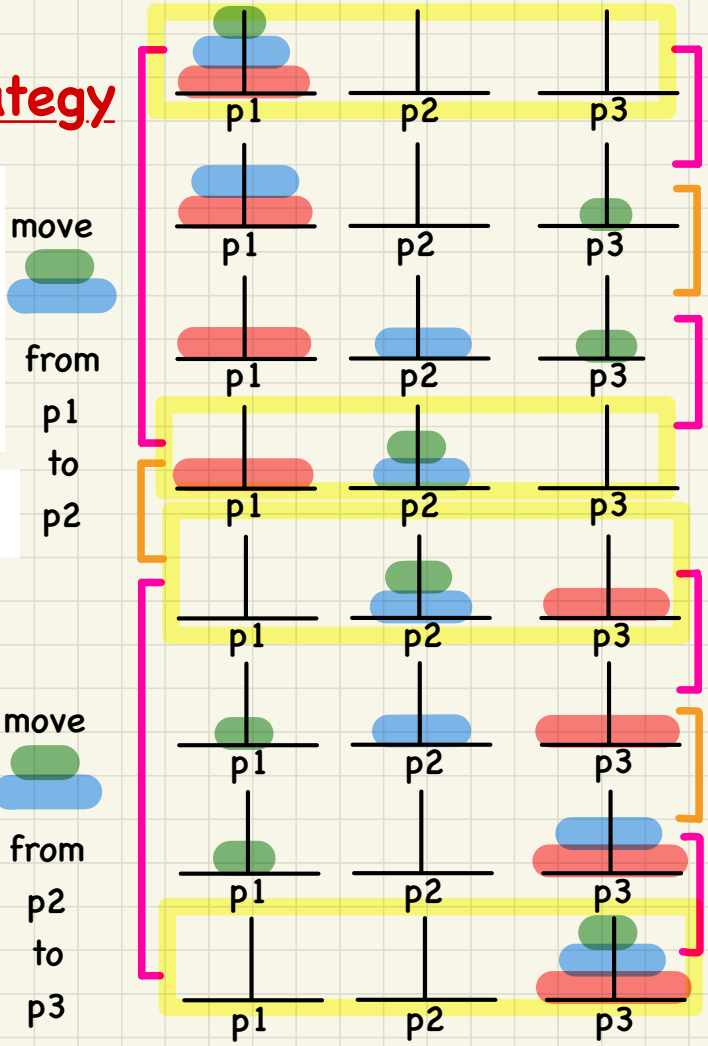
move from p3 to p2

move from p1 to p3

move from p2 to p1

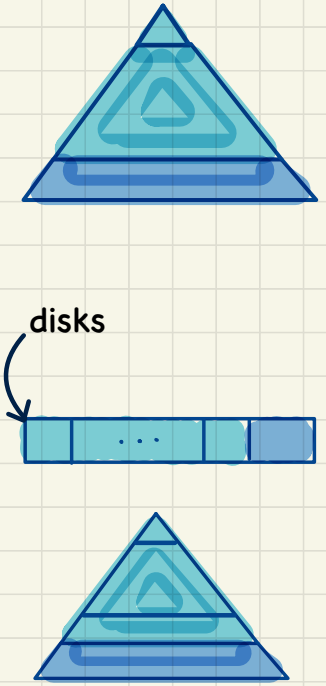
move from p2 to p3

move from p1 to p3



# Tower of Honoi in Java

```
void towerOfHanoi(String[] disks) {
  tohHelper(disks, 0, disks.length - 1, 1, 3);
}
void tohHelper(String[] disks, int from, int to, int ori, int des){
  if(from > to) { }
  else if(from == to) {
    print("move " + disks[to] + " from " + ori + " to " + des);
  }
  else {
    int intermediate = 6 - ori - des;
    tohHelper(disks, from, to - 1, ori, intermediate);
    print("move " + disks[to] + " from " + ori + " to " + des);
    tohHelper(disks, from, to - 1, intermediate, des);
  }
}
```

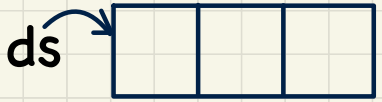


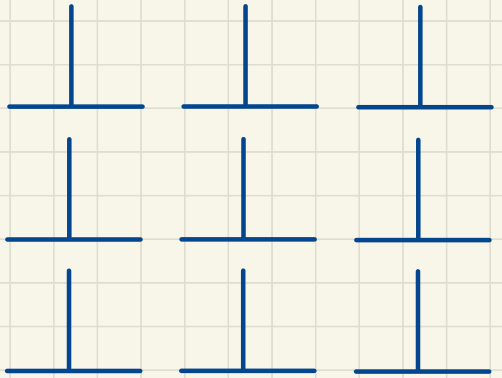

Stage 1

Stage 2

Stage 3

# Tower of Hanoi: Tracing



# Tower of Hanoi: Tracing

Say *ds* (disks) is $\{A, B, C\}$, where $A < B < C$.

$$
tohH(ds, \underbrace{0,2}_{\{A,B,C\}}, p1, p3) = \begin{cases}
tohH(ds, \underbrace{0,1}_{\{A,B\}}, p1, p2) = \begin{cases}
tohH(ds, \underbrace{0,0}_{\{A\}}, p1, p3) = \{ \texttt{Move A: p1 to p3} \\
\texttt{Move B: p1 to p2} \\
tohH(ds, \underbrace{0,0}_{\{A\}}, p3, p2) = \{ \texttt{Move A: p3 to p2}
\end{cases} \\
\texttt{Move C: p1 to p3} \\
tohH(ds, \underbrace{0,1}_{\{A,B\}}, p2, p3) = \begin{cases}
tohH(ds, \underbrace{0,0}_{\{A\}}, p2, p1) = \{ \texttt{Move A: p2 to p1} \\
\texttt{Move B: p2 to p3} \\
tohH(ds, \underbrace{0,0}_{\{A\}}, p1, p3) = \{ \texttt{Move A: p1 to p3}
\end{cases}
\end{cases}
$$

# Tower of Hanoi: Running Time

```
void towerOfHanoi(String[] disks) {
  tohHelper(disks, 0, disks.length - 1, 1, 3);
}
void tohHelper(String[] disks, int from, int to, int ori, int des){
  if(from > to) { }
  else if(from == to) {
    print("move " + disks[to] + " from " + ori + " to " + des);
  }
  else {
    int intermediate = 6 - ori - des;
    tohHelper(disks, from, to - 1, ori, intermediate);
    print("move " + disks[to] + " from " + ori + " to " + des);
    tohHelper(disks, from, to - 1, intermediate, des);
  }
}
```

## Running Time as a Recurrence Relation