Lecture 26

Sequence Algorithms (Continued)
Announcements for This Lecture

Assignment & Lab

- A6 is not graded yet
  - Done early next week
- A7 due **Sun, Dec. 3**
  - But extensions possible
  - Just ask for one!
  - But make good effort
- Lab Today: Office Hours
  - Get help on A7 aliens
  - Anyone can go to any lab

Next Week

- Last Week of Class!
  - Finish sorting algorithms
  - Special final lecture
- Lab held, but is optional
  - Can use lab as extra credit
  - Also use lab time on A7
- Details about the exam
  - Multiple review sessions

11/21/17
**Recall: Horizontal Notation**

<table>
<thead>
<tr>
<th>0</th>
<th>k</th>
<th>len(b)</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>&lt;= sorted</td>
<td>&gt;=</td>
</tr>
</tbody>
</table>

Example of an assertion about an sequence b. It asserts that:

1. b[0..k−1] is sorted (i.e. its values are in ascending order)
2. Everything in b[0..k−1] is \( \leq \) everything in b[k..len(b)−1]

Given index \( h \) of the first element of a segment and index \( k \) of the element that follows that segment, the number of values in the segment is \( k − h \).

\[ b[h .. k − 1] \text{ has } k − h \text{ elements in it.} \]
Partition Algorithm

• Given a sequence b[h..k] with some value x in b[h]:

  \[ h \quad x \quad ? \quad k \]

  **pre:** b

  **post:** b

  \[ \leq x \quad x \quad \geq x \]

• Swap elements of b[h..k] and store in j to truthify post:

  h  i  i+1  k

  **pre:** b

  **post:** b

  \[ h \quad j \quad k \]

  \[ \leq x \quad x \quad > x \]

  **inv:** b

  \[ \leq x \quad x \quad ? \quad \geq x \]

• Agrees with precondition when \( i = h, j = k+1 \)

• Agrees with postcondition when \( j = i+1 \)
def partition(b, h, k):
    """Partition list b[h..k] around a pivot x = b[h]"""
    i = h; j = k+1; x = b[h]
    # invariant: b[h..i-1] < x, b[i] = x, b[j..k] >= x
    while i < j-1:
        if b[i+1] >= x:
            # Move to end of block.
            swap(b,i+1,j-1)
            j = j - 1
        else:  # b[i+1] < x
            swap(b,i,i+1)
            i = i + 1
    # post: b[h..i-1] < x, b[i] is x, and b[i+1..k] >= x
    return i

partition(b,h,k), not partition(b[h:k+1])
Remember, slicing always copies the list!
We want to partition the original list
def partition(b, h, k):
    """Partition list b[h..k] around a pivot x = b[h]""
    i = h; j = k + 1; x = b[h]
    # invariant: b[h..i-1] < x, b[i] = x, b[j..k] >= x
    while i < j - 1:
        if b[i + 1] >= x:
            # Move to end of block.
            swap(b, i + 1, j - 1)
            j = j - 1
        else:
            # b[i+1] < x
            swap(b, i, i + 1)
            i = i + 1
    # post: b[h..i-1] < x, b[i] is x, and b[i+1..k] >= x
    return i
def partition(b, h, k):
    """Partition list b[h..k] around a pivot x = b[h]"""
    i = h; j = k+1; x = b[h]
    # invariant: b[h..i-1] < x, b[i] = x, b[j..k] >= x
    while i < j-1:
        if b[i+1] >= x:
            # Move to end of block.
            swap(b,i+1,j-1)
            j = j - 1
        else:  # b[i+1] < x
            swap(b,i,i+1)
            i = i + 1
    # post: b[h..i-1] < x, b[i] is x, and b[i+1..k] >= x
    return i
Partition Algorithm Implementation

```python
def partition(b, h, k):
    """Partition list b[h..k] around a pivot x = b[h]""
    i = h; j = k+1; x = b[h]
    # invariant: b[h..i-1] < x, b[i] = x, b[j..k] >= x
    while i < j-1:
        if b[i+1] >= x:
            # Move to end of block.
            swap(b, i+1, j-1)
            j = j - 1
        else:
            # b[i+1] < x
            swap(b, i, i+1)
            i = i + 1
    # post: b[h..i-1] < x, b[i] is x, and b[i+1..k] >= x
    return i
```

```
1        2   <= x  x  ?  >= x
1  2  3  1  5  0  6  3  8
     h   i  i+1   j  k

1  2  1  3  5  0  6  3  8
     h   i  i+1  j  k
```

```
1  2  1  3  0  5  6  3  8
     h   i   j     k
```

11/21/17
Sequences (Continued)
def partition(b, h, k):
    """Partition list b[h..k] around a pivot x = b[h]""
    i = h; j = k + 1; x = b[h]
    # invariant: b[h..i-1] < x, b[i] = x, b[j..k] >= x
    while i < j - 1:
        if b[i + 1] >= x:
            # Move to end of block.
            swap(b, i + 1, j - 1)
            j = j - 1
        else:
            # b[i+1] < x
            swap(b, i, i + 1)
            i = i + 1
    # post: b[h..i-1] < x, b[i] is x, and b[i+1..k] >= x
    return i
Dutch National Flag Variant

- Sequence of integer values
  - ‘red’ = negatives, ‘white’ = 0, ‘blues’ = positive
  - Only rearrange part of the list, not all

<table>
<thead>
<tr>
<th>pre: b</th>
<th>h</th>
<th>?</th>
<th>k</th>
</tr>
</thead>
<tbody>
<tr>
<td>post: b</td>
<td>&lt; 0</td>
<td>= 0</td>
<td>&gt; 0</td>
</tr>
<tr>
<td>inv: b</td>
<td>&lt; 0</td>
<td>?</td>
<td>= 0</td>
</tr>
</tbody>
</table>
# Dutch National Flag Variant

- **Sequence of integer values**
  - ‘red’ = negatives, ‘white’ = 0, ‘blues’ = positive
  - Only rearrange part of the list, not all

**Pre:**

<table>
<thead>
<tr>
<th>h</th>
<th>k</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>?</td>
</tr>
</tbody>
</table>

**Post:**

<table>
<thead>
<tr>
<th>h</th>
<th>t</th>
<th>i</th>
<th>j</th>
<th>k</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt; 0</td>
<td>= 0</td>
<td>&gt; 0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Inv:**

<table>
<thead>
<tr>
<th>h</th>
<th>t</th>
<th>i</th>
<th>j</th>
<th>k</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt; 0</td>
<td>?</td>
<td>= 0</td>
<td>&gt; 0</td>
<td></td>
</tr>
</tbody>
</table>

**Pre:**

- $t = h$
- $i = k+1$
- $j = k$

**Post:**

- $t = i$
def dnf(b, h, k):
    '''Returns: partition points as a tuple (i,j)'''
    t = h; i = k+1, j = k;
    # inv: b[h..t-1] < 0, b[t..i-1] > 0, b[i..j] = 0, b[j+1..k] > 0
    while t < i:
        if b[i-1] < 0:
            swap(b, i-1, t)
            t = t+1
        elif b[i-1] == 0:
            i = i-1
        else:
            swap(b, i-1, j)
            i = i-1; j = j-1
    # post: b[h..i-1] < 0, b[i..j] = 0, b[j+1..k] > 0
    return (i, j)
**Dutch National Flag Algorithm**

```
def dnf(b, h, k):
    # Returns: partition points as a tuple (i,j)"
    t = h; i = k+1, j = k;
    # inv: b[h..t-1] < 0, b[t..i-1] ?, b[i..j] = 0, b[j+1..k] > 0
    while t < i:
        if b[i-1] < 0:
            swap(b, i-1, t)
            t = t+1
        elif b[i-1] == 0:
            i = i-1
        else:
            swap(b, i-1, j)
            i = i-1; j = j-1
    # post: b[h..i-1] < 0, b[i..j] = 0, b[j+1..k] > 0
    return (i, j)
```

<table>
<thead>
<tr>
<th></th>
<th>h</th>
<th>t</th>
<th>i</th>
<th>j</th>
<th>k</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt; 0</td>
<td>-1</td>
<td>-2</td>
<td>3</td>
<td>-1</td>
<td>0</td>
</tr>
<tr>
<td>?</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>= 0</td>
<td></td>
<td></td>
<td>i</td>
<td>j</td>
<td>k</td>
</tr>
<tr>
<td>&gt; 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>6</td>
</tr>
</tbody>
</table>
Dutch National Flag Algorithm

def dnf(b, h, k):
    """Returns: partition points as a tuple (i,j)""
    t = h; i = k+1, j = k;
    # inv: b[h..t-1] < 0, b[t..i-1] > 0, b[i..j] = 0, b[j+1..k] > 0
    while t < i:
        if b[i-1] < 0:
            swap(b, i-1, t)
            t = t+1
        elif b[i-1] == 0:
            i = i-1
        else:
            swap(b, i-1, j)
            i = i-1; j = j-1
    # post: b[h..i-1] < 0, b[i..j] = 0, b[j+1..k] > 0
    return (i, j)
def dnf(b, h, k):
    # Returns: partition points as a tuple (i,j)
    t = h; i = k+1, j = k;
    # inv: b[h..t-1] < 0, b[t..i-1] ?, b[i..j] = 0, b[j+1..k] > 0
    while t < i:
        if b[i-1] < 0:
            swap(b,i-1,t)
            t = t+1
        elif b[i-1] == 0:
            i = i-1
        else:
            swap(b,i-1,j)
            i = i-1; j = j-1
    # post: b[h..i-1] < 0, b[i..j] = 0, b[j+1..k] > 0
    return (i, j)
Changing the Invariant

- Different invariants = different code
  - Need to change how we initialize, stop
  - Also need to change the body of the loop

<table>
<thead>
<tr>
<th>pre:</th>
<th>b</th>
<th>?</th>
</tr>
</thead>
<tbody>
<tr>
<td>h</td>
<td></td>
<td>k</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>post:</th>
<th>b</th>
<th>&lt; 0</th>
<th>= 0</th>
<th>&gt; 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>h</td>
<td></td>
<td>k</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>inv:</th>
<th>b</th>
<th>&lt; 0</th>
<th>= 0</th>
<th>?</th>
<th>&gt; 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>h</td>
<td>t</td>
<td>i</td>
<td>j</td>
<td>k</td>
<td></td>
</tr>
</tbody>
</table>
Changing the Invariant

- Different invariants = different code
  - Need to change how we initialize, stop
  - Also need to change the body of the loop

<table>
<thead>
<tr>
<th>pre:</th>
<th>b</th>
<th>h</th>
<th>k</th>
</tr>
</thead>
<tbody>
<tr>
<td>post:</td>
<td>b</td>
<td>&lt; 0</td>
<td>= 0</td>
</tr>
</tbody>
</table>

| inv:  | b | < 0 | = 0 | ? | > 0 |

<table>
<thead>
<tr>
<th>pre:</th>
<th>t = h, i = h, j = k</th>
</tr>
</thead>
<tbody>
<tr>
<td>post:</td>
<td>t = j+1</td>
</tr>
</tbody>
</table>
Changing the Invariant

```python
def dnf(b, h, k):
    """Returns: partition points as a tuple (i,j)""
    t = h; i = h, j = k;
    # inv: b[h..t-1] < 0, b[i..t-1] = 0, b[t..j] ?, b[j+1..k] > 0
    while t < j+1:
        if b[???] < 0:
            ???
        elif b[???] == 0:
            ??? == 0:
        else:
            ???

    # post: b[h..i-1] < 0, b[i..j] = 0, b[j+1..k] > 0
    return (i, j)
```

<table>
<thead>
<tr>
<th>h</th>
<th>i</th>
<th>t</th>
<th>j</th>
<th>k</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>-2</td>
<td>0</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>-1</td>
<td>0</td>
<td>6</td>
<td>3</td>
</tr>
</tbody>
</table>

<0 | =0 | ? | >0

11/21/17
Sequences (Continued)
def dnf(b, h, k):
    
    """Returns: partition points as a tuple (i,j)""
    
    t = h; i = h, j = k;
    # inv: b[h..t-1] < 0, b[i..t-1] = 0, b[t..j] ?, b[j+1..k] > 0
    
    while t < j+1:
        if b[t] < 0:
            ???

        elif b[t] == 0:
            ???

        else:
            ???

        # post: b[h..i-1] < 0, b[i..j] = 0, b[j+1..k] > 0
    
    return (i, j)
Changing the Invariant

```python
def dnf(b, h, k):
    """Returns: partition points as a tuple (i,j)""
    t = h; i = h, j = k;
    # inv: b[h..t-1] < 0, b[i..t-1] = 0, b[t..j] ?, b[j+1..k] > 0
    while t < j+1:
        if b[t] < 0:
            ???
        elif b[t] == 0:
            ???
        else:
            swap(b,t,j)
            j = j-1
    # post: b[h..i-1] < 0, b[i..j] = 0, b[j+1..k] > 0
    return (i, j)
```

```
11/21/17
Sequences (Continued)
20
```
def dnf(b, h, k):
    """Returns: partition points as a tuple (i,j)"""
    t = h; i = h, j = k;
    # inv: b[h..t-1] < 0, b[i..t-1] = 0, b[t..j] ?, b[j+1..k] > 0
    while t < j+1:
        if b[t] < 0:
            ???
        elif b[t] == 0:
            t = t+1
        else:
            swap(b,t,j)
            j = j-1
    # post: b[h..i-1] < 0, b[i..j] = 0, b[j+1..k] > 0
    return (i, j)
def dnf(b, h, k):
    
    # Returns: partition points as a tuple (i,j)
    t = h; i = h, j = k;
    # inv: b[h..t-1] < 0, b[i..t-1] = 0, b[t..j] ?, b[j+1..k] > 0
    while t < j+1:
        if b[t] < 0:
            swap(b,t,i)
            i = i+1; t = t+1;
        elif b[t] == 0:
            t = t+1
        else:
            swap(b,t,j)
            j = j-1
    # post: b[h..i-1] < 0, b[i..j] = 0, b[j+1..k] > 0
    return (i, j)
Changing the Invariant

def dnf(b, h, k):
    """Returns: partition points"""
    t = h; i = h, j = k;
    # b[h..t-1] <, b[t..i-1] =, b[i..j] ?, b[j+1..k] >
    while t < j+1:
        if b[t] < 0:
            swap(b,t,i)
            i = i+1; t = t+1;
        elif b[t] == 0:
            t = t+1
        else:
            swap(b,t,j)
            j = j-1
    # b[h..i-1] <, b[i..j] =, b[j+1..k] >
    return (i, j)

def dnf(b, h, k):
    """Returns: partition points"""
    t = h; i = k+1, j = k;
    # b[h..t-1] <, b[t..i-1] =, b[i..j] ?, b[j+1..k] >
    while t < i:
        if b[i-1] < 0:
            swap(b,i-1,t)
            t = t+1
        elif b[i-1] == 0:
            i = i-1
        else:
            swap(b,i-1,j)
            i = i-1; j = j-1
    # b[h..i-1] <, b[i..j] =, b[j+1..k] >
    return (i, j)
Flag of Mauritius

• Now we have four colors!
  - Negatives: ‘red’ = odd, ‘purple’ = even
  - Positives: ‘yellow’ = odd, ‘green’ = even

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>h</td>
<td>k</td>
<td></td>
</tr>
<tr>
<td>pre:</td>
<td>b</td>
<td>?</td>
</tr>
</tbody>
</table>

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>h</td>
<td>k</td>
<td></td>
<td></td>
</tr>
<tr>
<td>post:</td>
<td>b</td>
<td>&lt; 0 odd</td>
<td>&lt; 0 even</td>
</tr>
</tbody>
</table>

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>h</td>
<td>r</td>
<td>s</td>
<td>i</td>
<td>t</td>
</tr>
<tr>
<td>inv:</td>
<td>b</td>
<td>&lt; 0, o</td>
<td>&lt; 0, e</td>
<td>≥ 0, o</td>
</tr>
</tbody>
</table>
Flag of Mauritius

<table>
<thead>
<tr>
<th>&lt;0, o</th>
<th>&lt;0, e</th>
<th>≥0, o</th>
<th>?</th>
<th>≥0, e</th>
</tr>
</thead>
<tbody>
<tr>
<td>h</td>
<td>r</td>
<td>s</td>
<td>i</td>
<td>t</td>
</tr>
<tr>
<td>-1</td>
<td>-3</td>
<td>-2</td>
<td>-4</td>
<td>7</td>
</tr>
<tr>
<td>-5</td>
<td>-6</td>
<td>1</td>
<td>0</td>
<td>2</td>
</tr>
</tbody>
</table>

One swap is not good enough
Flag of Mauritius

<table>
<thead>
<tr>
<th>&lt; 0, o</th>
<th>&lt; 0, e</th>
<th>≥ 0, o</th>
<th>?</th>
<th>≥ 0, e</th>
</tr>
</thead>
<tbody>
<tr>
<td>h</td>
<td>r</td>
<td>s</td>
<td>i</td>
<td>t</td>
</tr>
<tr>
<td>-1</td>
<td>-3</td>
<td>-2</td>
<td>-4</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>-5</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2</td>
</tr>
</tbody>
</table>

Need two swaps for two spaces
Flag of Mauritius

<table>
<thead>
<tr>
<th>&lt; 0, o</th>
<th>&lt; 0, e</th>
<th>≥ 0, o</th>
<th>?</th>
<th>≥ 0, e</th>
</tr>
</thead>
<tbody>
<tr>
<td>h</td>
<td>r</td>
<td>s</td>
<td>i</td>
<td>t</td>
</tr>
<tr>
<td>-1</td>
<td>-3</td>
<td>-2</td>
<td>-4</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>5</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>-5</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>-6</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>4</td>
</tr>
</tbody>
</table>

And adjust the loop variables
Flag of Mauritius

<table>
<thead>
<tr>
<th>&lt; 0, o</th>
<th>&lt; 0, e</th>
<th>≥ 0, o</th>
<th>?</th>
<th>≥ 0, e</th>
</tr>
</thead>
<tbody>
<tr>
<td>h</td>
<td>r</td>
<td>s</td>
<td>i</td>
<td>t</td>
</tr>
<tr>
<td>-1</td>
<td>-3</td>
<td>-2</td>
<td>-4</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>-5</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2</td>
</tr>
</tbody>
</table>

See algorithms.py for Python code

11/21/17
Sequences (Continued)
Flag of Mauritius

<table>
<thead>
<tr>
<th>&lt; 0, o</th>
<th>&lt; 0, e</th>
<th>≥ 0, o</th>
<th>?</th>
<th>≥ 0, e</th>
</tr>
</thead>
<tbody>
<tr>
<td>h</td>
<td>r</td>
<td>s</td>
<td>i</td>
<td>t</td>
</tr>
<tr>
<td>-1</td>
<td>-3</td>
<td>-2</td>
<td>-4</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>-5</td>
</tr>
</tbody>
</table>

See algorithms.py for Python code
Linear Search

- **Vague**: Find first occurrence of $v$ in $b[h..k-1]$. 
Linear Search

- **Vague**: Find first occurrence of $v$ in $b[h..k-1]$.
- **Better**: Store an integer in $i$ to truthify result condition post:
  
  post:  
  1. $v$ is not in $b[h..i-1]$ 
  2. $i = k$ OR $v = b[i]$
Linear Search

- **Vague:** Find first occurrence of \( v \) in \( b[h..k-1] \).
- **Better:** Store an integer in \( i \) to truthify result condition post:
  
  \[
  \text{post:} \quad 1. \ v \text{ is not in } b[h..i-1] \\
  \quad 2. \ i = k \ OR \ v = b[i]
  \]

\[
\begin{array}{c}
\text{pre: } b \\
\hline
h & ? & k
\end{array}
\]

\[
\begin{array}{ccc}
\text{post: } b \\
\hline
h & i & k
\end{array}
\]

\[
\begin{array}{c|c|c}
\text{v not here} & v & ?
\end{array}
\]
• **Vague**: Find first occurrence of \( v \) in \( b[h..k-1] \).

• **Better**: Store an integer in \( i \) to truthify result condition post:

  - 1. \( v \) is not in \( b[h..i-1] \)
  - 2. \( i = k \) OR \( v = b[i] \)

\[
\begin{array}{|c|c|}
\hline
h & k \\
\hline
{\text{pre: } b} & ? \\
\hline
\end{array}
\]

\[
\begin{array}{|c|c|c|}
\hline
h & i & k \\
\hline
{\text{post: } b} & \text{v not here} & v & ? \\
\hline
\end{array}
\]

\[
\begin{array}{|c|c|}
\hline
h & k \\
\hline
i & {\text{b}} \\
\hline
\end{array}
\]

\[
\begin{array}{|c|}
\hline
\text{v not here} \\
\hline
\end{array}
\]

11/21/17 Sequences (Continued)
Linear Search

pre: b

post: b

OR

inv: b

Sequences (Continued)
def linear_search(b,v,h,k):
    """Returns: first occurrence of v in b[h..k-1]""
    # Store in i index of the first v in b[h..k-1]
    i = h

    # invariant: v is not in b[0..i-1]
    while i < k and b[i] != v:
        i = i + 1

    # post: v is not in b[h..i-1]
    #        i >= k or b[i] == v
    return i if i < k else -1

Analyzing the Loop

1. Does the initialization make inv true?

2. Is post true when inv is true and condition is false?

3. Does the repetend make progress?

4. Does the repetend keep the invariant inv true?
Binary Search

- **Vague**: Look for $v$ in sorted sequence segment $b[h..k]$. 
Binary Search

- **Vague:** Look for v in sorted sequence segment b[h..k].

- **Better:**
  - **Precondition:** b[h..k-1] is sorted (in ascending order).
  - **Postcondition:** b[h..i] <= v and v < b[i+1..k-1]

- Below, the array is in non-descending order:

```plaintext
pre: b

| h | ? | k |

post: b

| <= v | > v |
```

11/21/17 Sequences (Continued) 37
Binary Search

- **Vague:** Look for \( v \) in sorted sequence segment \( b[h..k] \).
- **Better:**
  - **Precondition:** \( b[h..k-1] \) is sorted (in ascending order).
  - **Postcondition:** \( b[h..i] \leq v \) and \( v < b[i+1..k-1] \)
- **Below, the array is in non-descending order:**

```
\begin{array}{c|c|c|c}
  h & i & j & k \\
  \hline
  \text{pre: } b & ? & & \\
  \hline
  \text{post: } b & \leq v & > v & \\
  \hline
  \text{inv: } b & \leq v & ? & > v \\
\end{array}
```

Called binary search because each iteration of the loop cuts the array segment still to be processed in half
Extras Not Covered in Class
Loaded Dice

- Sequence $p$ of length $n$ represents $n$-sided die
  - Contents of $p$ sum to 1
  - $p[k]$ is probability die rolls the number $k$

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>0.1</td>
<td>0.1</td>
<td>0.1</td>
<td>0.3</td>
<td>0.3</td>
<td>weighted d6, favoring 5, 6</td>
</tr>
</tbody>
</table>

- Goal: Want to “roll the die”
  - Generate random number $r$ between 0 and 1
  - Pick $p[i]$ such that $p[i-1] < r \leq p[i]$

<table>
<thead>
<tr>
<th></th>
<th>0.1</th>
<th>0.1</th>
<th>0.1</th>
<th>0.1</th>
<th>0.3</th>
<th>0.3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>0.2</td>
<td>0.3</td>
<td>0.4</td>
<td>0.7</td>
<td>1.0</td>
<td></td>
</tr>
</tbody>
</table>
**Loaded Dice**

- **Want:** Value $i$ such that $p[i-1] < r <= p[i]$

$$
\begin{array}{cccc}
0 & \quad & \quad & n \\
\text{pre: } b & \quad & ? \\
0 & i & \quad & n \\
\text{post: } b & r > \text{sum} & \quad & r \leq \text{sum}
\end{array}
$$

- **Same as precondition if $i = 0$**
- **Postcondition is invariant + false loop condition**
def roll(p):
    
    """Returns: randint in 0..len(p)-1; i returned with prob. p[i]
    Precondition: p list of positive floats that sum to 1."""
    r = random.random()  # r in [0,1)
    # Think of interval [0,1] divided into segments of size p[i]
    # Store into i the segment number in which r falls.
    i = 0;  sum_of = p[0]
    # inv: r >= sum of p[0] .. p[i-1]; pEnd = sum of p[0] .. p[i]
    while r >= sum_of:
        sum_of = sum_of + p[i+1]
        i = i + 1
    # post: sum of p[0] .. p[i-1] <= r < sum of p[0] .. p[i]
    return i

Analyzing the Loop
1. Does the initialization make inv true?
2. Is post true when inv is true and condition is false?
3. Does the repetend make progress?
4. Does the repetend keep inv true?
### Reversing a Sequence

<table>
<thead>
<tr>
<th>pre: b</th>
<th>not reversed</th>
</tr>
</thead>
<tbody>
<tr>
<td>h</td>
<td>k</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>post: b</th>
<th>reversed</th>
</tr>
</thead>
<tbody>
<tr>
<td>h</td>
<td>k</td>
</tr>
</tbody>
</table>

#### Change:

<table>
<thead>
<tr>
<th>change: b</th>
<th>1 2 3 4 5 6 7 8 9 9 9 9</th>
</tr>
</thead>
<tbody>
<tr>
<td>h</td>
<td>k</td>
</tr>
</tbody>
</table>

#### Into:

<table>
<thead>
<tr>
<th>into b</th>
<th>9 9 9 9 8 7 6 5 4 3 2 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>h</td>
<td>k</td>
</tr>
</tbody>
</table>

#### Inv:

<table>
<thead>
<tr>
<th>inv: b</th>
<th>swapped</th>
<th>not reversed</th>
<th>swapped</th>
</tr>
</thead>
<tbody>
<tr>
<td>h</td>
<td>i</td>
<td>j</td>
<td>k</td>
</tr>
</tbody>
</table>