**Binary Search**

- Look for value \( v \) in **sorted** segment \( b[h..k] \)

  pre: \( b \) ?
  post: \( b < v \) \( i \) ? \( j \) ? \( k \)

  inv: \( b < v \) ? \( i \) ? \( j \) ? \( k \)

  New statement of the invariant guarantees that we get leftmost position of \( v \) if found

  * if \( v \) is 3, set \( i \) to 0
  * if \( v \) is 4, set \( i \) to 5
  * if \( v \) is 5, set \( i \) to 7
  * if \( v \) is 8, set \( i \) to 10

**Example \( b \)**

\[
\begin{array}{cccccccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
\end{array}
\]

Looking at \( b[i] \) gives linear search from left.
Looking at \( b[j-1] \) gives linear search from right.
Looking at middle: \( b[(i+j)/2] \) gives binary search.

**Flag of Mauritius**

\[
\begin{array}{cccccccccc}
<0, o & <0, e & ? & \approx & t & \approx & c & \approx & k \\
\end{array}
\]

Need two swaps for two spaces

BUT NOT ALWAYS!

Have to check if second swap is okay

**Sorting: Arranging in Ascending Order**

pre: \( b \) ? \( n \)
post: \( b \) ? \( n \)

**Insertion Sort**

\[
\begin{array}{cccccccccc}
i = 0 & 0 & \text{sorted} & ? \\
\end{array}
\]

\[
\begin{array}{cccccccccc}
i = 0 & 0 & 1 & 2 & 4 & 4 & 6 & 6 & 7 & 8 & 9 \\
\end{array}
\]

**Insertion Sort: Moving into Position**

\[
\begin{array}{cccccccccc}
i = 0 & 0 & \text{sorted} & ? \\
\end{array}
\]

\[
\begin{array}{cccccccccc}
i = 0 & 0 & 1 & 2 & 4 & 4 & 6 & 6 & 7 & 8 & 9 \\
\end{array}
\]

**Insertion Sort: Performance**

\[
\begin{array}{cccccccccc}
\text{Total Swaps: } 0 + 1 + 2 + 3 + \ldots (n-1) = (n-1)n/2 \\
\end{array}
\]
## Algorithm “Complexity”

- **Given:** a list of length n and a problem to solve
- **Complexity:** *rough* number of steps to solve worst case
- Suppose we can compute 1000 operations a second:

<table>
<thead>
<tr>
<th>Complexity</th>
<th>n=10</th>
<th>n=100</th>
<th>n=1000</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>0.01 s</td>
<td>0.1 s</td>
<td>1 s</td>
</tr>
<tr>
<td>n log n</td>
<td>0.016 s</td>
<td>0.32 s</td>
<td>4.79 s</td>
</tr>
<tr>
<td>n²</td>
<td>0.1 s</td>
<td>10 s</td>
<td>167 m</td>
</tr>
<tr>
<td>n³</td>
<td>1 s</td>
<td>167 m</td>
<td>11.6 d</td>
</tr>
<tr>
<td>2ⁿ</td>
<td>1 s</td>
<td>4x10¹⁰ y</td>
<td>3x10³⁸⁹ y</td>
</tr>
</tbody>
</table>

### Sorting: Changing the Invariant

#### Selection Sort:

- **Inv:** sorted, ≤ b[0..i-1] ≥ b[i..n-1]
- **Pre:** h ≤ b[i] ≥ k
- **Post:** h ≤ b[i] ≥ k
- **Change:** b[h, i, i+1, k]

#### Insertion Sort:

1. Set `i = 0`
2. While `i < n`:
   - Find minimum index `j` of `b[i..n-1]`
   - Swap `b[i, j]`
   - Increment `i`

#### Selection Sort:

- First segment always contains smaller values
- Selection sort also is an $n^2$ algorithm

### Partition Algorithm

- **Given:** a list segment $b[h..k]$ with some value $x$ in $b[h]$
- **Pre:** $b[h..k]$
- **Post:** $b[h..k]$ into $b[h..j-1]$ and $b[j..k+1]$

### Sorting with Partitions

- **Given:** a list segment $b[h..k]$ with some value $x$ in $b[h]$
- **Pre:** $b[h..k]$
- **Post:** $b[h..k]$ into $b[h..j-1]$ and $b[j..k+1]$

#### QuickSort

```python
def quick_sort(b, h, k):
    """Sort the array fragment b[h..k]"""
    if b[h..k] has fewer than 2 elements:
        return
    j = partition(b, h, k)
    # b[j-1] < b[j] < b[j+1]
    quick_sort(b, h, j-1)
    quick_sort(b, j+1, k)
```

### Final Word About Algorithms

- **Algorithm:**
  - Step-by-step way to do something
  - Not tied to specific language
- **Implementation:**
  - An algorithm in a specific language
  - Many times, not the “hard part”
- **Higher Level Computer Science courses:**
  - We teach advanced algorithms (pictures)
  - You learn on your own