**Binary Search**

- Look for value v in sorted segment b[h..k]

```plaintext
pre: b | h k
post: b | <= v

inv: b | h i k
```

**Example b**

<table>
<thead>
<tr>
<th>0 1 2 3 4 5 6 7 8</th>
<th>h k</th>
</tr>
</thead>
<tbody>
<tr>
<td>3 3 3 3 4 4 6 7 7</td>
<td></td>
</tr>
</tbody>
</table>

* 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

**New statement of the invariant guarantees that we get leftmost position of v if found**

```plaintext
h i j k
```

- 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**

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

**Insertion Sort: Moving into Position**

```python
i = 0
while i < n:
    push_down(b, i)
    i = i+1

def push_down(b, i):
    if j > 0:
        if b[j-1] > b[j]:
            swap(b[j-1], b[j])
            j = j+1
```

**Insertion Sort: Performance**

```python
def push_down(b, i):
    """Push value at position i into sorted position in b[0..i-1]""
    j = i
    while j > 0:
        if b[j-1] > b[j]:
            swap(b[j-1], b[j])
            j = j+1
```

- b[0..i-1]: i elements
- Worst case:
  - i = 0: 0 swaps
  - i = 1: 1 swap
  - i = 2: 2 swaps
- Pushdown is in a loop
- Called for i in 0..n
- i swaps each time

**Total Swaps:** 0 + 1 + 2 + 3 + … (n-1) = (n-1)*n/2
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>n^2</td>
<td>0.1 s</td>
<td>10 s</td>
<td>16.7 m</td>
</tr>
<tr>
<td>n^3</td>
<td>1 s</td>
<td>16.7 m</td>
<td>11.6 d</td>
</tr>
<tr>
<td>2^n</td>
<td>1 s</td>
<td>4x10^9 y</td>
<td>3x10^9 y</td>
</tr>
</tbody>
</table>

**Major Topic in 2110: Beyond scope of this course**

Sorting: Changing the Invariant

pre: b
post: b sorted

Selection Sort:

\[
i = 0 \\
\text{while } i < n: \\
j = \text{index of min of } b[i..n-1] \\
\text{swap}(b, i, j) \\
i = i + 1
\]

Sorting with Partitions

Given a list segment [b[h..k]] with some value x in b[h]:

- Swap elements of [b[h..k]] and store in j to truthify post:
  - pre: b[h..k] <= x, x >= b[h..k]
  - post: b[j] <= x, x >= b[j+1..k]
-  x is called the pivot value
-  x is not a program variable
-  denotes value initially in b[h]

QuickSort

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

- **Worst Case:** array already sorted
  - Or almost sorted
  - n^2 in that case
- **Average Case:** array is scrambled
  - n log n in that case
  - Best sorting time!

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)
  - Implementation you learn on your own

List Diagrams

Demo Code