Lecture 27

Sorting
Announcements for This Lecture

Prelim/Finals

- Prelims in **handback room**
  - Gates Hall 216
  - Open 12-4pm each day

- **Final: Dec 9th 7:00-9:30pm**
  - Study guide is posted
  - Announce reviews on Thurs.

- **Conflict with Final time?**
  - Submit to conflict to CMS by this THURSDAY!

Assignments/Lab

- **A6** is now graded.
  - **Mean**: 89, **Median**: 94
  - **Std Deviation**: 14.2
  - Mean/Median **Time**: 12 hrs

- **A7** is due next **Dec. 11**
  - Will grade if turn in Sun.

- **Lab 13** is **optional**
  - Good study for the final
  - Consultant hours still open
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 \( v \leq b[i+1..k] \)

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

<table>
<thead>
<tr>
<th>h</th>
<th>?</th>
<th>k</th>
</tr>
</thead>
</table>
  **pre:** \( b[h..i] < v \) and \( v \leq b[i+1..k] \)

<table>
<thead>
<tr>
<th>h</th>
<th>i</th>
<th>k</th>
</tr>
</thead>
</table>
  **post:** \( b[h..i] < v \) and \( v \leq b[i+1..k] \)
Binary Search

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

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

<table>
<thead>
<tr>
<th>h</th>
<th>i</th>
<th>k</th>
</tr>
</thead>
<tbody>
<tr>
<td>post: $b$</td>
<td>$&lt; v$</td>
<td>$\geq v$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>h</th>
<th>i</th>
<th>j</th>
<th>k</th>
</tr>
</thead>
<tbody>
<tr>
<td>inv: $b$</td>
<td>$&lt; v$</td>
<td>?</td>
<td>$\geq v$</td>
</tr>
</tbody>
</table>

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 $b$**

| 3 | 3 | 3 | 3 | 3 | 4 | 4 | 6 | 7 | 7 |

$12/1/15$  Sorting  $5$
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]

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

<table>
<thead>
<tr>
<th>h</th>
<th>?</th>
<th>k</th>
</tr>
</thead>
<tbody>
<tr>
<td>pre: b</td>
<td>&lt; v</td>
<td>&gt;= v</td>
</tr>
</tbody>
</table>

Called binary search because each iteration of the loop cuts the array segment still to be processed in half.

<table>
<thead>
<tr>
<th>h</th>
<th>i</th>
<th>j</th>
<th>k</th>
</tr>
</thead>
<tbody>
<tr>
<td>inv: b</td>
<td>&lt; v</td>
<td>?</td>
<td>&gt; v</td>
</tr>
</tbody>
</table>
Binary Search

<table>
<thead>
<tr>
<th>h</th>
<th>k</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>j</td>
</tr>
</tbody>
</table>

pre: \( b \) ?

post: \( b \) < v \( >= v \)

inv: \( b \) < v ? \( >= v \)

\( i = h; j = k+1; \)

\textbf{while} \( i \neq j: \)

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

New statement of the invariant guarantees that we get \textit{leftmost} position of \( v \) if found
Flag of Mauritius

- Now we have four colors!
  - Negatives: ‘red’ = odd, ‘purple’ = even
  - Positives: ‘yellow’ = odd, ‘green’ = even
## One swap is not good enough

<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></td>
</tr>
<tr>
<td>-5</td>
<td>-6</td>
<td>1</td>
<td>0</td>
<td></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>

One swap is not good enough
Flag of Mauritius

Need two swaps for two spaces

<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>
</tbody>
</table>

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

BUT NOT ALWAYS!
### 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</td>
<td>r=s</td>
<td>i</td>
<td>t</td>
</tr>
<tr>
<td>-1</td>
<td>-3</td>
<td>-7</td>
<td>-5</td>
</tr>
</tbody>
</table>

Have to check if second swap is okay

**BUT NOT ALWAYS!**
Sorting: Arranging in Ascending Order

Insertion Sort:

\[ \text{i} = 0 \]
\[ \text{while} \ i < n: \]
\[ \quad \# \text{Push b[i] down into its} \]
\[ \quad \# \text{sorted position in b[0..i]} \]
\[ \quad i = i + 1 \]
**Insertion Sort: Moving into Position**

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

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

*swap shown in the lecture about lists*
The Importance of Helper Functions

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

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

i = 0
while i < n:
    j = i
    while j > 0:
        if b[j-1] > b[j]:
            temp = b[j]
            b[j] = b[j-1]
            b[j-1] = temp
        j = j - 1
    i = i + 1

Can you understand all this code below?
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, 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>$4 \times 10^{19}$ y</td>
<td>$3 \times 10^{290}$ y</td>
</tr>
</tbody>
</table>

**Major Topic in 2110**: Beyond scope of this course
**Sorting: Changing the Invariant**

pre: \( b \) \(?\) \hspace{1cm} post: \( b \) \(\text{sorted}\)

<table>
<thead>
<tr>
<th>pre:</th>
<th>post:</th>
</tr>
</thead>
<tbody>
<tr>
<td>( b ) (?)</td>
<td>( b ) (\text{sorted})</td>
</tr>
</tbody>
</table>

**Selection Sort:**

*inv:* \( b \) \(\text{sorted, } \leq b[i..] \) \( \geq b[0..i-1]\)

\( i = 0 \)

\( \text{while } i < n: \)
- \# Find minimum in \( b[i..] \)
- \# Move it to position \( i \)
- \( i = i + 1 \)

First segment always contains smaller values

12/1/15

Sorting 19
Sorting: Changing the Invariant

Selection Sort:

\[
\begin{align*}
\text{inv: } & \quad b_{\text{sorted, } \leq b[i..]} \quad \geq b[0..i-1] \\
\text{i = 0} & \\
\text{while } i < n: & \\
& \quad j = \text{index of min of } b[i..n-1] \\
& \quad \text{swap}(b, i, j) \\
& \quad i = i + 1
\end{align*}
\]

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]:

  h                       k
pre:  b
      x [ ?

  h       i     i+1          k
post:  b
      <= x   x   >= x

  h       k
change:  b
      3 5 4 1 6 2 3 8 1

  h      i    k
into  b
      1 2 1 3 5 4 6 3 8

  h      i    k
or  b
     1 2 3 1 3 4 5 6 8

  h    k
• x is called the **pivot value**
  ▪ x is not a program variable
  ▪ denotes value initially in b[h]
Sorting with Partitions

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

h
pre: b x ? k

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

h i i+1 k
post: b <= y y >= y x >= x

Partition Recursively

Recursive partitions = sorting
- Called QuickSort (why???)
- Popular, fast sorting technique
**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[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