Lecture 27

Sorting
Announcements for This Lecture

Prelim/Finals

- Prelims in **handback room**
  - Gates Hall 216
  - See Piazza for hours

- **Final: Dec 7th 9:00-11:30am**
  - Study guide is posted
  - Announce reviews on Thurs.

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

This Week

- **Lab 13** is optional, final lab
  - Due day before final exam
  - Not part of mandatory 12
  - Best way to study for final

- But **Lab 12** is NOT optional

- **A7** is due **SUNDAY**
  - Extensions to Dec 8 possible
  - Have been granting if ask
  - S/U students get by default

11/28/17
Let’s Talk about Assignment 6

• An extensive redesign of a 2011 assignment
  ▪ Last offered when class was very different
  ▪ Back then majority were engineers & less of them

• We saw a WIDE variety of scores/difficulty
  ▪ Grades: Mean 80, Median 84, SDev 15
  ▪ Time: Mean 16.8 hrs, Median 15 hrs, SDev 7.3 hrs

• Most common rating: Pretty Good
  ▪ But enough students hated to drop to Ok
  ▪ Students who took longer rated lower
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:** $b[h..i-1] < v$ and $v \leq b[i..k]$
- Below, the array is in non-descending order:

```
| h | ? | k |
---|---|---|
pre: b |     |
```

```
| h | i | k |
---|---|---|
post: b | < v | >= v |
```
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:</td>
<td>b</td>
<td></td>
</tr>
<tr>
<td></td>
<td>h</td>
<td>i</td>
</tr>
<tr>
<td>post:</td>
<td>b</td>
<td>&lt; v</td>
</tr>
<tr>
<td></td>
<td>h</td>
<td>i</td>
</tr>
<tr>
<td>inv:</td>
<td>b</td>
<td>&lt; 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

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>6</td>
<td>7</td>
<td>7</td>
</tr>
</tbody>
</table>

11/28/17
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: \( b[h..i-1] < v \) and \( v \leq b[i..k] \)

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

  \[
  \begin{array}{cccc}
  h & & k \\
  \text{pre: } b & & ? \\
  h & i & k \\
  \text{post: } b & < v & \geq v \\
  h & i & j & k \\
  \text{inv: } b & < v & ? & > v
  \end{array}
  \]

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

Pre: \( b \)

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

Post: \( b \)

\[
\begin{array}{c|c|c|c|c}
\text{h} & < v & \geq v \\
\hline
\text{h} & \text{i} & \text{j} & \text{k} \\
\end{array}
\]

Inv: \( b \)

\[
\begin{array}{c|c|c|c|c}
\text{h} & < v & ? & \geq v \\
\hline
\end{array}
\]

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

while \( i \neq j: \)

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.

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

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

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

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

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

One swap is not good enough
Flag of Mauritius

<table>
<thead>
<tr>
<th>h</th>
<th>r</th>
<th>s</th>
<th>i</th>
<th>t</th>
<th>k</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>-3</td>
<td>-2</td>
<td>-4</td>
<td>7</td>
<td>5</td>
</tr>
<tr>
<td>-5</td>
<td>-6</td>
<td>1</td>
<td>0</td>
<td></td>
<td></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>-5</td>
<td>-6</td>
<td>1</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

And adjust the loop variables
Flag of Mauritius

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

### BUT NOT ALWAYS!

Have to check if second swap is okay
**Sorting: Arranging in Ascending Order**

<table>
<thead>
<tr>
<th>pre: b</th>
<th>?</th>
<th>n</th>
</tr>
</thead>
<tbody>
<tr>
<td>post: b</td>
<td>sorted</td>
<td>n</td>
</tr>
</tbody>
</table>

**Insertion Sort:**

- \( i = 0 \)
- while \( i < n \):
  - # Push \( b[i] \) down into its sorted position in \( b[0..i] \)
  - \( i = i + 1 \)

```python
while i < n:
    # Push b[i] down into its sorted position in b[0..i]
    i = i + 1
```
Insertion Sort: Moving into Position

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

0 2 4 4 6 6 7

i

0 2 4 4 6 6 5

i

0 2 4 4 5 6 6

i

0 2 4 5 6 6 7

i
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?
**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, 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:** $0 + 1 + 2 + 3 + \ldots + (n-1) = \frac{(n-1)n}{2}$

Insertion sort is an $n^2$ algorithm
## 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: $b$ ?

post: $b$ sorted

Selection Sort:

inv: $b$ sorted, $\leq b[i..]$ $\geq b[0..i-1]$

First segment always contains smaller values

\[
i = 0
\]

while $i < n$:

# Find minimum in $b[i..]$

# Move it to position $i$

\[
i = i + 1
\]
Sorting: Changing the Invariant

<table>
<thead>
<tr>
<th>pre:</th>
<th>post:</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>b sorted</td>
</tr>
</tbody>
</table>

Selection Sort:

\[
\begin{array}{c|c|c|c}
0 & i & n \\
\hline
b & \text{sorted, } \leq b[i..] & \geq b[0..i-1] \\
\end{array}
\]

\[
i = 0
\]

\[
\text{while } i < n:
\]

\[
j = \text{index of min of } b[i..n-1]
\]

\[
\text{swap}(b,i,j)
\]

\[
i = i + 1
\]

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 $b[h]$

  $$
  \begin{array}{c|c|c}
  \text{pre: } & b & x \\
  \end{array}
  $$

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

  $$
  \begin{array}{c|c|c|c|c|c|c|c|c|c|c}
  \text{post: } & b & <= x & x & >= x \\
  \end{array}
  $$

- $x$ 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]:

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

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

  \[
  \begin{array}{cccc}
  h & i & i+1 & k \\
  \text{post: } b & \leq y & y & \geq y & x & \geq x \\
  \end{array}
  \]

**Partition Recursively**

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

<table>
<thead>
<tr>
<th>h</th>
<th>i</th>
<th>i+1</th>
<th>k</th>
</tr>
</thead>
</table>
| <= x | x | >= x

pre: b

post: b
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