Lecture 23

Sorting
Announcements for This Lecture

Assignments

• A5 is now graded
  ▪ Mean: 95, Median: 97
  ▪ Average Time: 4-5 hours
  ▪ Longer than I expected…
• A6 is due **Friday**
  ▪ Just activated in CMS
  ▪ Should be on stenography
• A7 due Monday, Dec. 3
  ▪ Week **after** classes

Next Two Weeks

• Reading
  ▪ **Chapter 19**: Tkinter
  ▪ Alternative to Kivy
  ▪ But similar concepts
• Next Tue is important!
  ▪ Will need it for A7
• No lab next week
  ▪ This week is “last lab”
  ▪ Lab final week is optional

11/13/12
# Announcements for This Lecture

## Assignments

- **A6** is due **Tomorrow**
  - Hopefully you are close
  - Trying to add consultants
  - Keep reading Piazza
- **A7** due Monday, Dec. 3
  - Week **after** classes
  - Online Saturday
  - Do not need lecture until the paddle task

## Next Two Weeks

- **Reading**
  - **Chapter 19**: Tkinter
  - Alternative to Kivy
  - But similar concepts
- **Next Tue is important!**
  - Will need it for A7
  - Unifies attribute invariants and loop invariants
  - Last major topic of course

---

11/13/12

Sorting
Binary Search

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

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

11/13/12
Binary Search

h        k
pre:     b
h    i    k
i = h; j = k+1;
while i != 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
Sorting: Arranging in Ascending Order

Insertion Sort:

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

\[ i = 0 \]

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

\[ \quad \text{push\_down}(b, i) \]

\[ i = i + 1 \]

\[ \text{def push\_down}(b, i): \]

\[ \quad j = i \]

\[ \quad \text{while } j > 0: \]

\[ \quad \quad \text{if } b[j-1] > b[j]: \]

\[ \quad \quad \quad \text{swap}(b, j-1, j) \]

\[ \quad j = j - 1 \]

---

**Explanation:**

- **Initial Array:** \[ [2, 4, 4, 6, 6, 7] \]
- **Step 1:**
  - \[ i = 0 \]
  - \[ \text{push\_down}(b, 0) \]
  - \[ j = 0 \]
  - Loop until \( j = 0 \):
    - \( b[0-1] = 4 \) \( > \) \( b[0] = 2 \)
    - Swap \( 4 \) and \( 2 \)
    - \( j = 0 - 1 = -1 \) (Invalid index)
  - Update \( j = 0 \)

- **Step 2:**
  - \[ i = 0 + 1 = 1 \]
  - \[ \text{push\_down}(b, 1) \]
  - \[ j = 1 \]
  - Loop until \( j = 0 \):
    - \( b[1-1] = 4 \) \( = \) \( b[1] = 4 \)
    - \( j = 1 - 1 = 0 \)
  - Update \( j = 0 \)

- **Step 3:**
  - \[ i = 1 + 1 = 2 \]
  - \[ \text{push\_down}(b, 2) \]
  - \[ j = 2 \]
  - Loop until \( j = 0 \):
    - \( b[2-1] = 6 \) \( = \) \( b[2] = 6 \)
    - \( j = 2 - 1 = 1 \)
    - Loop until \( j = 0 \):
      - \( b[1-1] = 4 \) \( = \) \( b[1] = 4 \)
      - \( j = 1 - 1 = 0 \)
  - Update \( j = 0 \)

- **Step 4:**
  - \[ i = 2 + 1 = 3 \]
  - \[ \text{push\_down}(b, 3) \]
  - \[ j = 3 \]
  - Loop until \( j = 0 \):
    - \( b[3-1] = 6 \) \( = \) \( b[3] = 6 \)
    - \( j = 3 - 1 = 2 \)
    - Loop until \( j = 0 \):
      - \( b[2-1] = 6 \) \( = \) \( b[2] = 6 \)
      - \( j = 2 - 1 = 1 \)
      - Loop until \( j = 0 \):
        - \( b[1-1] = 4 \) \( = \) \( b[1] = 4 \)
        - \( j = 1 - 1 = 0 \)
  - Update \( j = 0 \)

- **Step 5:**
  - \[ i = 3 + 1 = 4 \]
  - \[ \text{push\_down}(b, 4) \]
  - \[ j = 4 \]
  - Loop until \( j = 0 \):
    - \( b[4-1] = 7 \) \( = \) \( b[4] = 7 \)
    - \( j = 4 - 1 = 3 \)
    - Loop until \( j = 0 \):
      - \( b[3-1] = 6 \) \( = \) \( b[3] = 6 \)
      - \( j = 3 - 1 = 2 \)
      - Loop until \( j = 0 \):
        - \( b[2-1] = 6 \) \( = \) \( b[2] = 6 \)
        - \( j = 2 - 1 = 1 \)
        - Loop until \( j = 0 \):
          - \( b[1-1] = 4 \) \( = \) \( b[1] = 4 \)
          - \( j = 1 - 1 = 0 \)
    - Update \( j = 0 \)

- **Final Array:** \[ [2, 4, 4, 5, 6, 6] \]

---

**Note:**

`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 = i + 1

Can you understand all this code below?

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

11/13/12
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

<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:

inv:  

\[
\begin{align*}
\text{b} & \quad \text{sorted, } \leq b[i..] \quad \geq b[0..i-1] \\
i & \quad \text{while } i < n: \\
& \quad \# \text{ Find minimum in } b[i..] \\
& \quad \# \text{ Move it to position } i \\
& \quad i = i + 1
\end{align*}
\]

First segment always contains smaller values
**Sorting: Changing the Invariant**

**Selection Sort:**

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

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

\( i = 0 \)

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

11/13/12  Sorting
Partition Algorithm

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

  \[
  \begin{array}{c|c|c}
  \text{pre:} & b & x \\
  \text{post:} & b & \leq x \quad x \quad \geq x
  \end{array}
  \]

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

  \[
  \begin{array}{c|c|c|c}
  \text{change:} & b & 3 & 5 & 4 & 1 & 6 & 2 & 3 & 8 & 1 \\
  \text{into} & b & 1 & 2 & 1 & 3 & 5 & 4 & 6 & 3 & 8 \\
  \text{or} & b & 1 & 2 & 3 & 1 & 3 & 4 & 5 & 6 & 8
  \end{array}
  \]

- \( x \) 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 \(b[h]\):

  \[
  \begin{array}{c|c|c|c|c|c|c|c|c|c}
  h & & & & & & k \\
  \hline
  \text{pre: } b [ & x & ? & ] \\
  \hline
  \text{post: } b [ & \leq y & y & \geq y & x & \geq x & ]
  \end{array}
  \]

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

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

Partition Recursively

Recursive partitions = sorting

- Called **QuickSort** (why???)
- Popular, fast sorting technique
QuickSort

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!

pre: b

\[
\begin{array}{c|c|c}
\hline
h & i & i+1 \\
\hline
\end{array}
\]

post: b

\[
\begin{array}{c|c|c}
\hline
\leq x & x & \geq x \\
\hline
\end{array}
\]
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