Lecture 27

Sorting
Announcements for This Lecture

Prelim/Finals

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

• Final: Dec 8th 2:00-4:30pm
  ▪ 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**: 90, **Median**: 94
  ▪ **Std Deviation**: 15
  ▪ **Mean/Median Time**: 11 hrs

• **A7** is due **SUNDAY**
  ▪ But ask for an extension…

• **Lab 13** is final lab
  ▪ Due by the final exam
  ▪ Optional if you did others
Binary Search

- Vague: Look for $v$ in sorted sequence segment $b[h..k]$. 

11/29/16 Sorting 3
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 <= b[i..k]`

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

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

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

<table>
<thead>
<tr>
<th>pre:</th>
<th>b</th>
<th>?</th>
</tr>
</thead>
<tbody>
<tr>
<td>post:</td>
<td>b</td>
<td>&lt; v</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: 3 3 3 3 3 4 4 6 7 7
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:

  \[
  \text{pre: } b \begin{array}{c}
h \\hline\hline i \\hline k \end{array} \quad ?
  \]

  \[
  \text{post: } b \begin{array}{cc}
h \hline i \hline j \hline k \end{array} < v \hline \geq v
  \]

  \[
  \text{inv: } b \begin{array}{ccc}
h \hline i \hline j \hline k \end{array} < v \hline ? \hline \geq v
  \]

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

\[ i = h; \quad j = k + 1; \]  
\[ \text{while } i \neq j:\]  
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

```
pre:  b  ?
     h  k

post: b  < 0 odd  < 0 even  ≥ 0 odd  ≥ 0 even
       h  k

inv:  b  < 0, o  < 0, e  ≥ 0, o  ?  ≥ 0, e
      h  r  s  i  t  k
```
One swap is not good enough
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></td>
<td></td>
<td></td>
<td></td>
<td>5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-5</td>
<td>-6</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>4</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></td>
<td></td>
<td></td>
<td></td>
<td>-5</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2</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></td>
<td></td>
<td>i</td>
<td></td>
</tr>
<tr>
<td>h</td>
<td>r=s</td>
<td></td>
<td>t</td>
</tr>
<tr>
<td>-1</td>
<td>-3</td>
<td>-7</td>
<td>-4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>i</td>
<td></td>
</tr>
<tr>
<td>h</td>
<td>r=s</td>
<td></td>
<td>t</td>
</tr>
<tr>
<td>-1</td>
<td>-3</td>
<td>-7</td>
<td>-5</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>
<tr>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>4</td>
</tr>
</tbody>
</table>

BUT NOT ALWAYS!

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

Insertion Sort:

\[ i = 0 \]

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

\[ \quad \# \text{ Push } b[i] \text{ down into its} \]

\[ \quad \# \text{ sorted position in } b[0..i] \]

\[ i = i + 1 \]
Insertion Sort: Moving into Position

\[ i = 0 \]

\[ \text{while } i < n:\]
\[ \quad \text{push\_down}(b, i) \]
\[ \quad i = i + 1 \]

\[ \text{def push\_down}(b, i): \]
\[ \quad j = i \]
\[ \quad \text{while } j > 0:\]
\[ \quad \quad \text{if } b[j-1] > b[j]: \]
\[ \quad \quad \quad \text{swap}(b, j-1, j) \]
\[ \quad \quad j = j - 1 \]

```
2 4 4 6 6 7 5
```

\[ 2 4 4 6 6 5 7 \]

\[ 2 4 4 5 6 6 7 \]

\[ \text{swap shown in the lecture about lists} \]
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

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
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

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

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

- $i = 0$

- **while** $i < n$:
  - # Find minimum in $b[i..]$  
  - # Move it to position $i$
  - $i = i + 1$

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

\[ \text{pre: } b \quad ? \quad \text{post: } b \quad \text{sorted} \]

**Selection Sort:**

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

\[ i = 0 \]

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

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

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

\[ i = i + 1 \]

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]:

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

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

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

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

- 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 $b[h]$:

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

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

  \[
  \begin{array}{c}
  \text{h} \\
  \text{pre: } b \\
  \text{<=} y \quad y \quad >= y \quad x \\
  \text{?} \\
  \text{post: } b \\
  \text{<=} y \quad y \quad >= y \quad x \quad >= x \\
  \text{k}
  \end{array}
  \]

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!

```
11/29/16
```

---

**Pre:**

<table>
<thead>
<tr>
<th>b</th>
<th>x</th>
<th>?</th>
</tr>
</thead>
</table>

**Post:**

<table>
<thead>
<tr>
<th>b</th>
<th>&lt;= x</th>
<th>x</th>
<th>&gt;= x</th>
</tr>
</thead>
</table>

**Diagram:**

```
h                   k
x
```

```
h   i   i+1   k
< x  x  >= x
```
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