SEARCHING AND SORTING
HINT AT ASYMPTOTIC COMPLEXITY

Lecture 10
CS2110 – Spring 2016
Miscellaneous


- Pinned Piazza note on Supplemental study material. @472. Contains material that may help you study certain topics. It also talks about how to study.
Search as in problem set: b is sorted

<table>
<thead>
<tr>
<th>pre: b</th>
<th>post: b</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>h</td>
<td>h &lt;= v</td>
</tr>
<tr>
<td>t</td>
<td>t &gt; v</td>
</tr>
<tr>
<td>b.length</td>
<td>b.length</td>
</tr>
</tbody>
</table>

Methodology:
1. Draw the invariant as a combination of pre and post
2. Develop loop using 4 loopy questions.

```
    h = -1; t = b.length;
    while (h+1 != t) {
        if (b[h+1] <= v) h = h+1;
        else t = h+1;
    }
```

Practice doing this!
Search as in problem set: b is sorted

pre: \( b \)  

\[
\begin{array}{c|c|c}
0 & b.\text{length} & ? \\
\end{array}
\]

post: \( b \)  

\[
\begin{array}{c|c|c}
0 & h & t \\
\leq v & ? & > v \\
\end{array}
\]

inv: \( b \)  

\[
\begin{array}{c|c|c|c}
0 & h & t & b.\text{length} \\
\leq v & ? & > v & \end{array}
\]

\[ h = -1; \ t = b.\text{length}; \]

while (h+1 != t) {

\[ \text{if} \ (b[h+1] \leq v) \ h = h+1; \]

\[ \text{else} \ t = h+1; \]

}

\[
\begin{array}{c|c|c|c}
b[0] & > v? \\
\end{array}
\]

one iteration.

\[
\begin{array}{c|c|c|c}
b[b.\text{length}-1] & \leq 0? \\
b.\text{length} \text{ iterations} \\
\end{array}
\]

Worst case: time is proportional to size of b

Since b is sorted, can cut \( ? \) segment in half. As a dictionary search
Search as in problem set: b is sorted

<table>
<thead>
<tr>
<th>pre: b</th>
<th>post: b</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0 h</td>
</tr>
<tr>
<td>b.length</td>
<td>b.length</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>inv: b</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;= v</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>h= −1; t= b.length;</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>while (h != t−1) {</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>int e= (h + t) / 2;</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>// h &lt; e &lt; t</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>if (b[e] &lt;= v) h= e;</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>else t= e;</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>}</th>
</tr>
</thead>
</table>
**Binary search: an O(log n) algorithm**

Initial state:

<table>
<thead>
<tr>
<th>0</th>
<th>h</th>
<th>t</th>
</tr>
</thead>
</table>

**inv:** \( b \)  

\[ \begin{array}{c|c|c|c}
\leq v & ? & > v \\
\end{array} \]

\( h = -1; \ t = b.\text{length}; \)

**while** \((h \neq t-1)\) {

\[ e = (h+t)/2; \]

**if** \((b[e] \leq v)\) \(h = e;\)

**else** \(t = e;\)

}

Each iteration cuts the size of the ? segment in half.

\[ b.\text{length} = n \]

\[ \begin{array}{c|c|c|c|c}
0 & h & e & t \\
\leq v & ? & ? & > v \\
\end{array} \]

\( n = 2^{**k} \) ? About k iterations

Time taken is proportional to k, or \( \log n \).

**A logarithmic algorithm**

Write as \( O(\log n) \)

[explain notation next lecture]
Looking at execution speed  
Process an array of size $n$

2n+2, n+2, n are all “order n” $O(n)$
Called linear in $n$, proportional to $n$

Number of operations executed

$\begin{align*}
\text{size n} & \\
0 & \quad 1 \quad 2 \quad 3 \quad \ldots
\end{align*}$

$\begin{align*}
\text{Constant time} & \\
2n + 2 \text{ ops} & \\
n + 2 \text{ ops} & \\
n \text{ ops} & \\
n^2 \text{ ops} &
\end{align*}$
InsertionSort

A loop that processes elements of an array in increasing order has this invariant:

**Inv:**

| b[0..i-1] is sorted | ? |

**Pre:**

| b | ? |

**Post:**

| b | sorted |

for (int i = 0; i < b.length; i = i + 1) { maintain invariant }
Each iteration, $i = i + 1$; How to keep inv true?

<table>
<thead>
<tr>
<th>inv:</th>
<th>b</th>
<th>sorted</th>
<th>?</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>i</td>
<td>b.length</td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>2   5   5   5   7</td>
<td>3</td>
<td>?</td>
</tr>
<tr>
<td>e.g.</td>
<td>b</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>i</td>
<td>b.length</td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>2   3   5   5   5</td>
<td>7</td>
<td>?</td>
</tr>
</tbody>
</table>

Push $b[i]$ down to its shortest position in $b[0..i]$, then increase $i$

Will take time proportional to the number of swaps needed
What to do in each iteration?

**inv:**

<table>
<thead>
<tr>
<th>0</th>
<th>i</th>
<th>b.length</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>b</th>
<th>sorted</th>
<th>?</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**e.g.:**

<table>
<thead>
<tr>
<th>0</th>
<th>i</th>
<th>b.length</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>b</th>
<th>2 5 5 5 7</th>
<th>3</th>
<th>?</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Loop body (inv true before and after)

**Push b[i] to its sorted position in b[0..i], then increase i**

<table>
<thead>
<tr>
<th>0</th>
<th>i</th>
<th>b.length</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>b</th>
<th>2 5 5 5</th>
<th>3</th>
<th>7</th>
<th>?</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>b</th>
<th>2 5 5 3</th>
<th>5</th>
<th>7</th>
<th>?</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| b | 2 5 3 5 5 | 7 | ? |
|---|---|---|---|---|
| 0 |   |   |   |   |

| b | 2 3 5 5 5 | 7 | ? |
|---|---|---|---|---|
| 0 |   |   |   |   |
// sort b[], an array of int
// inv: b[0..i-1] is sorted
for (int i = 0; i < b.length; i = i+1) {
    Push b[i] down to its sorted position in b[0..i]
}

Note English statement in body.

**Abstraction.** Says *what* to do, not *how*.

This is the best way to present it. We expect you to present it this way when asked.

Later, show how to implement that with a loop.

Many people sort cards this way
Works well when input is *nearly sorted*
InsertionSort

// Q: b[0..i-1] is sorted
// Push b[i] down to its sorted position in b[0..i]
int k = i;
while (k > 0 && b[k] < b[k-1]) {
    Swap b[k] and b[k-1]
    k = k–1;
}
// R: b[0..i] is sorted

invariant P: b[0..i] is sorted
except that b[k] may be < b[k-1]

start? stop? progress?
maintain invariant?

example
How to write nested loops

// sort b[], an array of int
// inv: b[0..i-1] is sorted
for (int i = 0; i < b.length; i = i + 1) {
  Push b[i] down to its sorted position in b[0..i]
}

Present algorithm like this

If you are going to show implementation, put in the "WHAT TT DO" as a comment
### InsertionSort

// sort b[], an array of int
// inv: b[0..i-1] is sorted
for (int i = 0; i < b.length; i = i + 1) {
    Push b[i] down to its sorted position in b[0..i]
}

Pushing b[i] down can take i swaps.
Worst case takes
\[1 + 2 + 3 + \ldots + (n-1) = \frac{(n-1) \times n}{2}\] Swaps.

- **Worst-case:** $O(n^2)$
  (reverse-sorted input)
- **Best-case:** $O(n)$
  (sorted input)
- **Expected case:** $O(n^2)$

$O(f(n))$ : Takes time proportional to $f(n)$.
Formal definition later
### SelectionSort

<table>
<thead>
<tr>
<th>pre</th>
<th>post</th>
<th>inv</th>
</tr>
</thead>
<tbody>
<tr>
<td>b[0..b.length]</td>
<td>b[0..b.length] sorted</td>
<td>b[0..i] (\leq) b[i..] (\geq) b[0..i-1]</td>
</tr>
</tbody>
</table>

**Additional term in invariant**

Keep invariant true while making progress?

| e.g. | |
|------|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| b[0..b.length] | i | b.length |
| 1 2 3 4 5 6 9 9 9 7 8 6 9 |

Increasing i by 1 keeps inv true only if b[i] is min of b[i..]
SelectionSort

Another common way for people to sort cards

Runtime
- Worst-case $O(n^2)$
- Best-case $O(n^2)$
- Expected-case $O(n^2)$

//sort b[], an array of int
// inv: b[0..i-1] sorted AND
//       b[0..i-1] <= b[i..]
for (int i = 0; i < b.length; i = i + 1) {
    int m = index of minimum of b[i..];
    Swap b[i] and b[m];
}

sorted, smaller values         larger values
<table>
<thead>
<tr>
<th>b</th>
<th>0</th>
<th>i</th>
<th>length</th>
</tr>
</thead>
</table>
Each iteration, swap min value of this section into b[i]
Swapping b[i] and b[m]

// Swap b[i] and b[m]

int t = b[i];
b[i] = b[m];
b[m] = t;
Partition algorithm of quicksort

pre: 

\[
\begin{array}{ccc}
  h & h+1 & k \\
  x & ? & \\
\end{array}
\]

post: 

\[
\begin{array}{ccc}
  h & j & k \\
  <= x & x & >= x \\
\end{array}
\]

x is called the pivot

Swap array values around until b[h..k] looks like this:
pivot

partition

j

Not yet sorted

these can be in any order

Not yet sorted

these can be in any order

The 20 could be in the other partition

Not yet sorted

these can be in any order
## Partition algorithm

<table>
<thead>
<tr>
<th></th>
<th>h</th>
<th>h+1</th>
<th>k</th>
</tr>
</thead>
<tbody>
<tr>
<td>pre:</td>
<td>b</td>
<td>x</td>
<td>?</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>h</th>
<th>j</th>
<th>k</th>
</tr>
</thead>
<tbody>
<tr>
<td>post:</td>
<td>b</td>
<td>&lt;= x</td>
<td>x</td>
</tr>
</tbody>
</table>

Combine pre and post to get an invariant

<table>
<thead>
<tr>
<th></th>
<th>h</th>
<th>j</th>
<th>t</th>
<th>k</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>b</td>
<td>&lt;= x</td>
<td>x</td>
<td>?</td>
</tr>
</tbody>
</table>
Partition algorithm

Initially, with \( j = h \) and \( t = k \), this diagram looks like the start diagram:

\[
\begin{array}{ccc|c|c}
  & h & j & t & k \\
 b & \leq x & x & ? & \geq x \\
\end{array}
\]

\( j = h; \ t = k; \)
\[
\text{while } (j < t) \ {\}
  \ \text{if } (b[j+1] \leq b[j]) \ {\}
    \ \text{Swap } b[j+1] \text{ and } b[j]; \ j = j+1;
  \ \text{else } \ {\}
    \ \text{Swap } b[j+1] \text{ and } b[t]; \ t = t-1;
\]

Terminate when \( j = t \), so the “?” segment is empty, so diagram looks like result diagram:

Takes linear time: \( O(k+1-h) \)
QuickSort procedure

/** Sort b[h..k]. */

public static void QS(int[] b, int h, int k) {
    if (b[h..k] has < 2 elements) return;  // Base case

    int j = partition(b, h, k);
    // We know b[h..j–1] <= b[j] <= b[j+1..k]

    //Sort b[h..j-1] and b[j+1..k]
    QS(b, h, j-1);
    QS(b, j+1, k);
}
Quicksort developed by Sir Tony Hoare (he was knighted by the Queen of England for his contributions to education and CS).

81 years old.

Developed Quicksort in 1958. But he could not explain it to his colleague, so he gave up on it.

Later, he saw a draft of the new language Algol 58 (which became Algol 60). It had recursive procedures. First time in a procedural programming language. “Ah!,” he said. “I know how to write it better now.” 15 minutes later, his colleague also understood it.
Worst case quicksort: pivot always smallest value

```java
/** Sort b[h..k]. */
public static void QS(int[] b, int h, int k) {
    if (b[h..k] has < 2 elements) return;
    int j = partition(b, h, k);
    QS(b, h, j-1);   QS(b, j+1, k);
}
```
Best case quicksort: pivot always middle value

depth 0. 1 segment of size ~n to partition.

Depth 2. 2 segments of size ~n/2 to partition.

Depth 3. 4 segments of size ~n/4 to partition.

Max depth: about log n. Time to partition on each level: ~n
Total time: O(n log n).

Average time for Quicksort: n log n. Difficult calculation
QuickSort procedure

/** Sort b[h..k]. */

public static void QS(int[] b, int h, int k) {
    if (b[h..k] has < 2 elements) return;
    int j = partition(b, h, k);
    // We know b[h..j–1] <= b[j] <= b[j+1..k]
    // Sort b[h..j–1] and b[j+1..k]
    QS(b, h, j-1);
    QS(b, j+1, k);
}

Worst-case: quadratic
Average-case: O(n log n)

Worst-case space: O(n*n)! --depth of recursion can be n
Can rewrite it to have space O(log n)
Average-case: O(n * log n)
Partition algorithm

Key issue: How to choose a pivot?

Choosing pivot

- Ideal pivot: the median, since it splits array in half

But computing median of unsorted array is $O(n)$, quite complicated

Popular heuristics: Use

- first array value (not good)
- middle array value
- median of first, middle, last, values GOOD!
- Choose a random element
Quicksort with logarithmic space

Problem is that if the pivot value is always the smallest (or always the largest), the depth of recursion is the size of the array to sort.

Eliminate this problem by doing some of it iteratively and some recursively.
Quicksort with logarithmic space

Problem is that if the pivot value is always the smallest (or always the largest), the depth of recursion is the size of the array to sort.

Eliminate this problem by doing some of it iteratively and some recursively. We may show you this later. Not today!
QuickSort with logarithmic space

/** Sort b[h..k]. */

public static void QS(int[] b, int h, int k) {
    int h1 = h; int k1 = k;
    // invariant b[h..k] is sorted if b[h1..k1] is sorted
    while (b[h1..k1] has more than 1 element) {
        Reduce the size of b[h1..k1], keeping inv true
    }
}
QuickSort with logarithmic space

/** Sort b[h..k]. */

public static void QS(int[] b, int h, int k) {
    int h1 = h; int k1 = k;
    // invariant b[h..k] is sorted if b[h1..k1] is sorted
    while (b[h1..k1] has more than 1 element) {
        int j = partition(b, h1, k1);
        // b[h1..j-1] <= b[j] <= b[j+1..k1]
        if (b[h1..j-1] smaller than b[j+1..k1])
            { QS(b, h, j-1); h1 = j+1; }
        else
            { QS(b, j+1, k1); k1 = j-1; }
    }
}

Only the smaller segment is sorted recursively. If b[h1..k1] has size n, the smaller segment has size < n/2. Therefore, depth of recursion is at most log n
Binary search: find position $h$ of $v = 5$

**pre:** array is sorted

$h = -1$

- $1 \ 4 \ 4 \ 5 \ 6 \ 6 \ 8 \ 8 \ 10 \ 11 \ 12$

$h = -1$

- $1 \ 4 \ 4 \ 5 \ 6 \ 6 \ 8 \ 8 \ 10 \ 11 \ 12$

$h = 2$

- $1 \ 4 \ 4 \ 5 \ 6 \ 6 \ 8 \ 8 \ 10 \ 11 \ 12$

$h = 3$

- $1 \ 4 \ 4 \ 5 \ 6 \ 6 \ 8 \ 8 \ 10 \ 11 \ 12$

$h = 3$  $t = 4$

- $1 \ 4 \ 4 \ 5 \ 6 \ 6 \ 8 \ 8 \ 10 \ 11 \ 12$

**post:**  $\leq v \quad h \quad > v$

**Loop invariant:**

$b[0..h] \leq v$

$b[t..] > v$

B is sorted