SEARCHING AND SORTING
HINT AT ASYMPTOTIC COMPLEXITY

Lecture 10
CS2110 – Fall 2016

Developing methods

We use Eclipse to show the development of A2 function evaluate. Here are important points to take away from it.

1. If similar code will appear in two or more places, consider writing a method to avoid that duplication.
2. If you introduce a new method, write a specification for it.
3. Before writing a loop, write a loop invariant for it.
4. Have a loop exploit the structure of the data it processes.
5. Don’t expect your first attempt to be perfect. Just as you rewrite and rewrite an essay, we rewrite programs.

Search as in problem set: b is sorted

pre: b[0] ≤ v <= t <= e <= b.length
post: b[0] ≤ v <= e ≤ t ≤ b.length

inv: b[0] ≤ v <= e <= t ≤ b.length

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

Practice doing this!

Since b is sorted, can cut segment in half. As a dictionary search

Miscellaneous

- Pinned Piazza note on Supplemental study material. @281. Contains material that may help you study certain topics. It also talks about how to study.
Binary search: an $O(\log n)$ algorithm

```
inv: \( h \leq v \Rightarrow t \geq v \)
\( h = -1; \quad t = b.length; \)
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.

Looking at execution speed

```
Number of operations executed
\( n^2 + 2 \), \( n^2 + 2 \), \( n \) are all "order \( n \)" \( O(n) \)
```

A logarithmic algorithm

Write as \( \mathcal{O}(\log n) \)

[explain notation next lecture]

InsertionSort

```
pre: b ? b.length
post: b sorted b.length
inv: b sorted \( i \) b.length
or: \( b[0..i-1] \) is sorted
\( 0 \) \( i \) b.length
inv: b processed \( i \) b.length
for (int \( i = 0 \); \( i < b.length \); \( i = i + 1 \)) { maintain invariant }
```

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

Each iteration, \( i = i + 1 \); How to keep inv true?

```
inv: b \( \quad i \quad b.length \)
\( 0 \) \( i \) b.length
or: \( b[0..i-1] \) is sorted
\( 0 \) \( i \) b.length
inv: b processed \( i \) b.length
```

Push \( b[i] \) 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: b \( \quad i \quad b.length \)
\( 0 \) \( i \) b.length
or: \( b[0..i-1] \) is sorted
\( 0 \) \( i \) b.length
```

Loop body (inv true before and after)

Push \( b[i] \) to its sorted position in \( b[0..i] \), then increase \( i \)

Many people sort cards this way

Works well when input is nearly sorted

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
### InsertionSort

**start?**

```
while (k > 0  &&  b[k] < b[k-1]) {
    Swap b[k] and b[k-1];
    k= k–1;
}
```

**stop?**

```
int k= i;
```

**progress?**

```
for (int i= 0; i < b.length; i= i+1) {
    Push b[i] down to its sorted position in b[0..i]
}
```

\[ \text{invariant P: } b[0..i] \text{ is sorted except that } b[k] \text{ may be } b[k-1] \]

Example:

<table>
<thead>
<tr>
<th>k</th>
<th>2</th>
<th>3</th>
<th>5</th>
<th>5</th>
<th>?</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>?</td>
<td>?</td>
<td>?</td>
<td>?</td>
<td>?</td>
</tr>
</tbody>
</table>

### SelectionSort

**pre:**

```
b[0..i-1] is sorted
// inv: b[i] <= b[i+1]  AND  b[i+1] <= b[i+2] ...
```

**post:**

```
b[0..i-1] sorted  AND  b[0..i] <= b[i+1]  AND  b[i+1] <= b[i+2]
```

\[ \text{increasing } i \text{ by 1 keeps inv true only if } b[i] \text{ is min of } b[i..i+1] \]

**worst-case:**

\[ O(n^2) \]

**best-case:**

\[ O(n) \]

**average-case:**

\[ O(n^2) \]

**runtime**

```
int m= index of minimum of b[i..i+1];
Swap b[i] and b[m];
```

### How to write nested loops

**start?**

```
for (int i= 0; i < b.length; i= i+1) {
    Push b[i] down to its sorted position in b[0..i]
}
```

**stop?**

```
```

**progress?**

```
if you are going to show implementation, put in “WHAT IT DOES” as a comment
```

### 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

**Partition algorithm**

pre: \( h \) \( h+1 \) \( k \)
post: \( b \)
\( x \) \( ? \) \( \) \( x \) \( j \) \( k \)
\( \leq x \) \( x \) \( \geq x \)

Swap array values around until \( b[h..k] \) looks like this:

pre:
post:
\( x \) is called the pivot

QuickSort procedure

```java
/** 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

\[
\begin{align*}
x_0 & \rightarrow x_0 & \text{partioning at depth 0} \\
j & \rightarrow x_1 & \text{partioning at depth 1} \\
x_0 & \rightarrow x_1 & \rightarrow x_2 & \text{partioning at depth 2}
\end{align*}
\]

/** 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

<table>
<thead>
<tr>
<th>j</th>
<th>x0</th>
<th>x1</th>
<th>x2</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;= x0</td>
<td>x0</td>
<td>&gt;= x0</td>
<td></td>
</tr>
<tr>
<td>&lt;= x1</td>
<td>x1</td>
<td>&gt;= x1</td>
<td></td>
</tr>
<tr>
<td>&lt;= x2</td>
<td>x0</td>
<td>&lt;= x2</td>
<td>x2</td>
</tr>
<tr>
<td>&lt;= x2</td>
<td>x2</td>
<td>&gt;= x2</td>
<td></td>
</tr>
</tbody>
</table>

\[
\begin{align*}
0 & \rightarrow \text{depth 0. 1 segment of size ~n to partition.} \\
0 & \rightarrow \text{depth 1. 2 segments of size ~n/2 to partition.} \\
0 & \rightarrow \text{depth 2. 4 segments of size ~n/4 to partition.} \\
\end{align*}
\]

Depth 0. 1 segment of size ~n to partition.
Depth 1. 2 segments of size ~n/2 to partition.
Depth 2. 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);
}

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

```java
/** Sort b[h..k]. */
public static void QS(int[] b, int h, int k) {
    int h1 = h, 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

```java
/** Sort b[h..k]. */
public static void QS(int[] b, int h, int k) {
    int h1 = h, 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
post: <= v    h    > v
```

<table>
<thead>
<tr>
<th>h = -1</th>
<th>t = 11</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 4 4 5 6 6 8 8 10 11 12</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>h = -1</th>
<th>t = 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 4 4 5 6 6 8 8 10 11 12</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>h = 2</th>
<th>t = 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 4 4 5 6 6 8 8 10 11 12</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>h = 1</th>
<th>t = 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 4 4 5 6 6 8 8 10 11 12</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>h = 1</th>
<th>t = 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 4 4 5 6 6 8 8 10 11 12</td>
<td></td>
</tr>
</tbody>
</table>
```

Loop invariant:
- b[0..h] <= v
- b[t..] > v
- B is sorted
h = 2
b[1..] <= b[2] <= b[3..]
```