We will not cover all this material.

**SEARCHING AND SORTING**

**HINT AT ASYMPTOTIC COMPLEXITY**

**Lecture 9**

CS2110 – Fall 2014

We will not cover all this material.

Last lecture: binary search

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

inv: b[0] <= v

b = –1; t = b.length;
while (h != t–1) {
  int e = (h+t)/2;
  if (b[e] <= v) h = e;
  else t = e;
}

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

Practice doing this!

Binary search: a O(log n) algorithm

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>h</td>
<td>&lt;= v</td>
<td>&gt; v</td>
</tr>
</tbody>
</table>

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

Practice doing this!

**Binary search:** a O(log n) algorithm

Search array with 32767 elements, only 15 iterations!

Bsearch:

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

inv: b[0] <= v

h = –1; t = b.length;
while (h != t–1) {
  int e = (h+t)/2;
  if (b[e] <= v) h = e;
  else t = e;
}

Bsearch executes \(\log n\) iterations for an array of size n. So the number of operations and if-tests made is proportional to \(\log n\). Therefore, Bsearch is called an order \(\log n\) algorithm, written \(O(\log n)\). We formalize this notation later.

Linear search: Find first position of v in b (if in)

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>h</td>
<td>b.length</td>
<td></td>
</tr>
</tbody>
</table>

store in h to truthify:

pre: b[0] <= v

post: b[h] <= v and h = b.length or b[h] = v

inv: b[0] <= v

h = 0;
while (h <= b.length && b[h] != v) h = h+1;

Expected or average time? \(n/2\) iterations. \(O(n/2)\) — is also \(O(n)\)

Looking at execution speed

Process an array of size n

| Number of operations executed |
|------------------------------|---|---|---|
| \(2n^2\) | \(n^2\) | \(n\) |

\(n^2\) ops

\(n^2 + 2\) ops

\(n + 2\) ops

\(n\) ops
**InsertionSort**

| pre: | b | ? |
| post: | b | sorted |
| inv: | b | sorted |
| or: | b[0..i-1] is sorted |

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

**What to do in each iteration?**

| pre: | 0 | i | b.length |
| inv: | b | sorted | ? |
| or: | b | sorted |

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.

**SelectionSort**

| pre: | 0 | ? |
| post: | b | sorted |
| inv: | b | sorted, <= b[i..]  => b[0..i-1] |

Keep invariant true while making progress?

Additional term in invariant

**SelectionSort**

| pre: | b | 1 2 3 4 5 6 9 9 7 8 6 9 |
| post: | b | sorted, smaller values  larger values |

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

<table>
<thead>
<tr>
<th>Runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Worst-case: O(n^2) (reverse-sorted input)</td>
</tr>
<tr>
<td>• Best-case: O(n) (sorted input)</td>
</tr>
<tr>
<td>• Expected-case: O(n^2)</td>
</tr>
</tbody>
</table>

**InsertionSort**

| pre: | b | ? |
| post: | b | sorted |
| inv: | b | sorted |

This is the best way to present it. Later, show how to implement that with a loop.

Many people sort cards this way.

Works well when input is nearly sorted.
Partition algorithm of quicksort

**Idea** Using the pivot value x that is in b[h]:

- **pre:** Swap array values around until b[h...k] looks like this:
  - x                          ?
  - h   h+1                                                 k
  - <= x                x           >= x
  - j                           k

- **post:** x is called the pivot

---

Partition algorithm

**pre:** b[k] <= x <= b[j]

- **h** <= x <= b[j] <= b[t] <= k
- **post:** Combine pre and post to get an invariant

---

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 procedure

```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);
    // 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);
    // Function does the partition algorithm and
    // returns position j of pivot
    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)
```
Worst case quicksort: pivot always smallest value

<table>
<thead>
<tr>
<th>j</th>
<th>x0</th>
<th>&gt;= x0</th>
<th>partitioning at depth 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>j</td>
<td>x0 x1</td>
<td>&gt;= x1</td>
<td>partitioning at depth 1</td>
</tr>
<tr>
<td>j</td>
<td>x0 x1 x2</td>
<td>&gt;= x2</td>
<td>partitioning at depth 2</td>
</tr>
</tbody>
</table>

Best case quicksort: pivot always middle value

<table>
<thead>
<tr>
<th>0</th>
<th>j</th>
<th>n</th>
<th>partitioning at depth 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;= x0</td>
<td>x0</td>
<td>&gt;= x0</td>
<td>depth 0: 1 segment of size ~n to partition</td>
</tr>
<tr>
<td>&lt;= x1</td>
<td>x1</td>
<td>&gt;= x1</td>
<td>Depth 2: 2 segments of size ~n/2 to partition</td>
</tr>
<tr>
<td>&lt;= x2</td>
<td>x2</td>
<td>&gt;= x2</td>
<td>Depth 3: 4 segments of size ~n/4 to partition</td>
</tr>
</tbody>
</table>

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

Quicksort developed by Sir Tony Hoare (he was knighted by the Queen of England for his contributions to education and CS).
Will be 80 in April.
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 68 (which became Algol 60). It had recursive procedures. First time in a programming language. “Ah!,” he said. “I know how to write it better now.” 15 minutes later, his colleague also understood it.

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
QuickSort with logarithmic space

```java
/** 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.