We may not cover all this material

Last lecture: binary search

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

<table>
<thead>
<tr>
<th>inv: b</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>h</td>
<td>t</td>
</tr>
<tr>
<td>&lt;= v</td>
<td>?</td>
<td>&gt; v</td>
</tr>
</tbody>
</table>

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

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

Practice doing this!

Binary search: find position h of v = 5

<table>
<thead>
<tr>
<th>pre: array is sorted</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td>h = -1</td>
</tr>
<tr>
<td>1 4 4 5 6 6 8 8 10 11 12</td>
</tr>
<tr>
<td>t = 11</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>h = -1</td>
</tr>
<tr>
<td>1 4 4 5 6 6 8 8 10 11 12</td>
</tr>
<tr>
<td>t = 5</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>h = 3</td>
</tr>
<tr>
<td>1 4 4 5 6 6 8 8 10 11 12</td>
</tr>
<tr>
<td>t = 5</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>h = 1</td>
</tr>
<tr>
<td>1 4 4 5 6 6 8 8 10 11 12</td>
</tr>
<tr>
<td>t = 4</td>
</tr>
</tbody>
</table>

post: <= v | h | > v

<table>
<thead>
<tr>
<th>Loop invariant:</th>
</tr>
</thead>
<tbody>
<tr>
<td>entries h and below are &lt;= v</td>
</tr>
<tr>
<td>entries t and above are &gt; v</td>
</tr>
<tr>
<td>entries between h and t are sorted</td>
</tr>
</tbody>
</table>

Binary search: an O(log n) algorithm

Search array with 32767 elements, only 15 iterations!

Bsearch:

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

Each iteration takes constant time (a few assignments and an if).

Bsearch executes log n iterations for an array of size n. So the number of assignments and if-tests made is proportional to log n.

Therefore, Bsearch is called an \textbf{order log n algorithm}, written \textbf{O(log n)}. (We'll formalize this notation later.)

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

<table>
<thead>
<tr>
<th>pre: b</th>
<th>post: b[v not here]</th>
<th>n = b.length</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>?</td>
<td>n</td>
</tr>
<tr>
<td></td>
<td>v not here</td>
<td>?</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>inv: b</th>
</tr>
</thead>
<tbody>
<tr>
<td>h</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>v not here</td>
</tr>
</tbody>
</table>

Find 5

<table>
<thead>
<tr>
<th>h = 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 4 4 5 6 6 8 8 10 11 12</td>
</tr>
</tbody>
</table>

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

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

<table>
<thead>
<tr>
<th>h = 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 4 4 5 6 6 8 8 10 11 12</td>
</tr>
</tbody>
</table>
**Linear search: Find first position of v in b (if present)**

- **pre:**
  - b: ?
  - 0 ≤ h < n

- **inv:**
  - v not here

- **post:**
  - b: ?
  - 0 ≤ h < n

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

Worst case: for array of size n, requires n iterations, each taking constant time. Worst-case time: O(n).

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

**Looking at execution speed**

- **Number of operations executed**
  - n^2 ops
  - n + 2 ops
  - n ops
  - Constant time

![Graph showing execution time vs array size](image)

**InsertionSort**

- **pre:**
  - b: ?
  - 0 ≤ i < n

- **inv:**
  - b[0..i-1] is sorted

- **post:**
  - b: sorted

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

**What to do in each iteration?**

- **inv:**
  - b: ?
  - 0 ≤ i < n

- **e.g.:**
  - b: 2 5 5 5 7 3

Loop body (inv true before and after)

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

**InsertionSort**

- **// sort b[], an array of int
  // inv: b[0..i-1] is sorted
  for (int i = 0; i < b.length; i++) {
    Push b[i] to its sorted position in b[0..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. Later, we can figure out how to implement it with a loop

**InsertionSort**

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

Takes time proportional to number of swaps. Finding the right place for b[i] can take i swaps.

Worst case takes

\[
1 + 2 + 3 + \ldots + n-1 = \frac{(n-1)n}{2}
\]

swaps.
### SelectionSort

**pre:**
- $b[0..i-1]$ sorted
- $b[0..i-1] \leq b[i..]$

**post:**
- $b[0..i-1]$ sorted
- $b[0..i-1] \leq b[i..]$

**inv:**
- Keep invariant true while making progress?

**e.g.:**
- $b[0..i-1]$ sorted
- $b[0..i-1] \leq b[i..]$

**increasing i by 1 keeps inv true only if b[i] is min of b[i..]**

---

### QuickSort: a recursive algorithm

**pre:**
- $b[0..i-1]$ sorted
- $b[0..i-1] \leq b[i..]$

**post:**
- $b[0..i-1]$ sorted
- $b[0..i-1] \leq b[i..]$

**partition step**
- $\leq x \quad x \quad \geq x$

**recursion step**
- [QuickSort] $x$ [QuickSort]

**e.g.:**
- $20 \quad 31 \quad 24 \quad 19 \quad 45 \quad 56 \quad 4 \quad 20 \quad 5 \quad 72 \quad 14 \quad 99$

---

### Partition algorithm of QuickSort

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

**pre:**
- $x$

**post:**
- $\leq x \quad x \quad \geq x$

---

### Partition algorithm

**pre:**
- $b[0..i-1]$ sorted
- $b[0..i-1] \leq b[i..]$

**post:**
- $\leq x \quad x \quad \geq x$

**Combine pre and post to get an invariant**

---

### SelectionSort

Another common way for people to sort cards

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

---

**Partition algorithm of QuickSort**

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

**pre:**
- $x$

**post:**
- $\leq x \quad x \quad \geq x$

---

### Partition algorithm

**pre:**
- $b[0..i-1]$ sorted
- $b[0..i-1] \leq b[i..]$

**post:**
- $\leq x \quad x \quad \geq x$

**Combine pre and post to get an invariant**
Partition algorithm

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

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);
}  // Function does the partition algorithm and returns position j of pivot

Worst case quicksort: pivot always smallest value

Best case quicksort: pivot always middle value

\[ \begin{align*}
\text{j} & \quad \begin{cases} 
0 & \quad \text{x0} \geq x0 \\
0 & \quad \text{x0} \geq x1 \\
0 & \quad \text{x0} \geq x2 
\end{cases} \\
\text{partioning at depth 0} \\
\text{partioning at depth 1} \\
\text{partioning at depth 2}
\end{align*} \]
QuickSort

QuickSort was developed by Sir Tony Hoare, who received the Turing Award in 1980.

He developed QuickSort in 1958, but could not explain it to his colleague, and gave up on it.

Later, he saw a draft of the new language Algol 68 (which became Algol 60). It had recursive procedures, for the first time in a programming language. “Ah!” said Sir Tony. “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

```
/** 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

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;
        }
    }
}
```