SEARCHING AND SORTING
HINT AT ASYMPTOTIC COMPLEXITY

Lecture 9
CS2110 – Fall 2015

We may not cover all this material

Last lecture: binary search

<table>
<thead>
<tr>
<th>pre</th>
<th>post</th>
</tr>
</thead>
<tbody>
<tr>
<td>h</td>
<td>?</td>
</tr>
<tr>
<td>t</td>
<td>&gt; v</td>
</tr>
</tbody>
</table>

inv:
- h = -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: find position h of v = 5

pre: array is sorted
post: <= v > v

inv:
- b[0] <= v
- ? > v
- t = b.length
- h = -1; t = 2^k - 1
- while (h != t - 1) {
  - int e = (h + t) / 2;
  - if (b[e] <= v) h = e;
  - else t = e;
}

Methodology:
- Set e to (h + t) / 2 = (2^k - 1) / 2 = 2^k - 1.
- Set t to e, i.e. t = 2^k - 1.
- Each iteration takes constant time (a few assignments and an if).

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 e = (h + t) / 2;
  - if (b[e] <= v) h = e;
  - else t = e;
}

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 order log n algorithm, written O(log n). (We’ll formalize this notation later.)
Linear search: Find first position of v in b (if in)

Store in h to truthify:

pre: \[ b \]

post: \[ b \]

inv: \[ b \]

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

\[ B: h != b.length \&\& b[h] != v \]

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
Process an array of size n

Number of operations executed

\[ \begin{align*}
2n &+ 2, n^2 &+ 2, n &\text{ are all } \text{“order n” } O(n) \\
\text{Called linear in n, proportional to n} &
\end{align*} \]

\[ n^2 \text{ ops} \]
\[ n \text{ ops} \]
\[ n + 2 \text{ ops} \]
\[ 2n + 2 \text{ ops} \]

Constant time

size n

InsertionSort

pre: \[ b \]

post: \[ b \]

inv: \[ b \]

\[ b[0..i-1] \text{ sorted} \]
\[ b[i] \text{ ?} \]

\[ b \]

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: \[ b \]

\[ \begin{align*}
\text{e.g. } b &
\end{align*} \]

\[ 0 \]
\[ 2 \]
\[ 3 \]
\[ 5 \]
\[ 5 \]
\[ 5 \]
\[ ? \]

\[ b \]

\[ 2 \]
\[ 3 \]
\[ 5 \]
\[ 5 \]
\[ 5 \]
\[ ? \]

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: \[ b \]

\[ \begin{align*}
\text{e.g. } b &
\end{align*} \]

\[ 0 \]
\[ 2 \]
\[ 5 \]
\[ 5 \]
\[ 5 \]
\[ 7 \]
\[ ? \]

\[ 0 \]
\[ 2 \]
\[ 3 \]
\[ 5 \]
\[ 5 \]
\[ 5 \]
\[ 7 \]

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

```java
// sort b[], an array of int
// inv: b[0..i-1] is sorted
for (int i = 1; i < b.length; i++) {
    Push b[i] down 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, show how to implement that with a loop

**SelectionSort**

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

Swapping b[i] and b[m]

```java
// Swap b[i] and b[m]
int t = b[i];
b[i] = b[m];
b[m] = t;
```

**Partition algorithm of quicksort**

```java
// Swap array values around until b[h..k] looks like this:
pre:  h  h+1... k  x  x         x is called the pivot
post: h  j   x   k   >= x
```
QuickSort

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.

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.

\[
\begin{array}{cccccc}
\phantom{\text{b}} & \text{h} & \text{j} & \text{t} & \text{k} \\
\text{b} & \leq x & x & ? & \geq x \\
\end{array}
\]

\[
\begin{array}{cccccc}
\phantom{\text{b}} & \text{h} & \text{j} & \text{t} & \text{k} \\
\text{b} & \leq x & x & ? & \geq x \\
\end{array}
\]

\[
\begin{array}{cccccc}
\phantom{\text{b}} & \text{h} & \text{j} & \text{t} & \text{k} \\
\text{b} & \leq x & x & ? & \geq x \\
\end{array}
\]

QuickSort procedure

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

\[
\begin{array}{cccccc}
\phantom{\text{b}} & \text{h} & \text{h+1} & \text{k} \\
\end{array}
\]

pre: \( \text{b} \)

\[
\begin{array}{cccccc}
\phantom{\text{b}} & \text{x} & ? & \text{x} \\
\end{array}
\]

post: \( \text{b} \)

\[
\begin{array}{cccccc}
\phantom{\text{b}} & \text{h} & \text{j} & \text{k} \\
\text{b} & \leq x & x & \geq x \\
\end{array}
\]

Combine pre and post to get an invariant.

invariant needs at least 4 sections

\[
\begin{array}{cccccc}
\phantom{\text{b}} & \text{h} & \text{j} & \text{t} & \text{k} \\
\text{b} & \leq x & x & ? & \geq x \\
\end{array}
\]

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

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

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

Worst case quicksort: pivot always smallest value

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

\[
\begin{array}{cccccc}
\phantom{\text{b}} & \text{j} & \text{x0} & \geq x0 \\
\end{array}
\]

partitioning at depth 0

\[
\begin{array}{cccccc}
\phantom{\text{x0} x1} & \text{x1} & \geq x1 \\
\end{array}
\]

partitioning at depth 1

\[
\begin{array}{cccccc}
\phantom{\text{x0} x1 x2} & \text{x2} & \geq x2 \\
\end{array}
\]

partitioning at depth 2
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

```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);
}
```

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

Worst-case space: \(O(n^2)\) -- depth of recursion can be \(n\)
Can rewrite it to have space \(O(\log n)\)

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

```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) {
        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; 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