Recitation 5

Loop Invariants and Prelim Review

Four loopy questions

//Precondition
Initialization:
// invariant: P
while ( B ) { S }

1. Does it start right?
   Does initialization make invariant P true?

2. Does it stop right?
   Does P and !B imply the desired result?

3. Does repetend S make progress toward termination?

4. Does repetend S keep invariant P true?

Add elements backwards

Precondition

Invariant

Postcondition

Add elements backwards

INV: b

INV: b

INV: b

INV: b

Add elements backwards

int s = 0;
int h = b.length-1;
while (h > 0) {
    s= s + b[h];
    h--;
}

int s = 0;
int h = b.length-1;
while (h >= 0) {
    s= s + b[h];
    h = h - 2;
}
Add elements backwards

```
int s = 0;
int h = 0;
while (h >= 0) {
    s = s + b[h];
    h--;
}
```

**INV:** \( s = \sum h \)

1. Does it start right?
2. Does it stop right?
3. Does it keep the invariant true?
4. Does it make progress toward termination?

---

**Linear search time**

Linear search for \( v \) in an array \( b \) of length \( n \)

```
b[h] ???
```

**worst-case time.** \( v \) is not in \( b[0..n-1] \), so linear search has to look at every element. Takes time proportional to \( n \).

**expected (average) case time.** If you look at all possibilities where \( v \) could be and average the number of elements linear search has to look at, you would get close to \( n/2 \). Still time proportional to \( n \).

---

**Binary search time (b[0..n-1] is sorted)**

```
h= -1; t= n;
// invariant: P (below)
while (h < t-1) {
    e= (h+t)/2;
    if (b[e] <= v) h= e;
    else t= e;
}
```

**inv P:** \( b[h..t-1] \) starts out with \( n \) elements in it.

Each iteration cuts size of \( b[h+1..t-1] \) in half.

**worst-case and expected case time:** \( \log n \)

---

**Selection sort of b[0..n-1]**

```
h= 0;
// invariant: P (below)
while (h < n) {
    Swap b[h] with min value in b[h..n-1];
    h= h+1;
}
```

To find the min value of \( b[h..n-1] \) takes time proportional to \( n - h \).

\[ n + (n-1) + \ldots + 3 + 2 + 1 = n (n-1) / 2 \]

**worst-case and average case time:** proportional to \( n^2 \)
Quicksort of b[0..n-1]

** Prelim Review **

`partition(b, h, k)` takes time proportional to size of `b[h..k]`

Best-case time: `partition` makes both sides equal length

- time `n` to partition
- time `n` to partition
- time `n` to partition

Depth: proportional to log `n`  
Therefore: time `n log n`

Worst-case time: `partition` makes one side empty

- time `n` to partition
- time `n-1` to partition
- time `n-2` to partition

Depth: proportional to `n`  
Therefore: time `n^2`

Someone proved that the average or expected time for quicksort is `n log n`

```
/** Sort b[h..k] */
void QS(int[] b, int h, int k) {
    if (b[h..k] size < 2) return;
    j = partition(b, h, k);
    // b[h..j-1] <= b[j] <= b[j+1..k]
    QS(b, h, j-1);
    QS(b, j+1, k)
}
```

What method calls are legal

Animal an;  an.m(args);

The . is computation.
Stores something in an.

Legal ONLY if Java can guarantee that method `m` exists. How to guarantee?

- `n` must be declared in Animal or inherited. Why?
- Someone might write a subclass `C` of Animal that does not have `m` declared in it, create an object of C, store it in an. Then method `m` would not exist.
- You know already from lecture 4 on class `Object`, overriding `toString()`, and the bottom-up/overriding rule that the overriding method is called.

Exception handling

```
private static double m(int x) {
    int y = x;
    try {
        y = 5/x;
        return 5/(x+2);
    } catch (NullPointerException e) {
        System.out.println("null");
    } catch (RuntimeException e) {
        y = 5/(x+1);
    } return 1/x;
}
```
Quicksort

0 1 2 3 5 4 6 7
0 1 2 3 4 5 6 7
0 1 2 3 4 5 6 7
0 1 2 3 4 5 6 7