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	b	???
Invariant	b	??? | h: s = sum
Postcondition	b	h: s = sum

Add elements backwards

INV: b | 0: ??? | h: s = sum |

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

Add elements backwards

INV: b | 0: ??? | h: s = sum |

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

Add elements backwards

INV: b | 0: ??? | h: s = sum |

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

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

Add elements backwards

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

	0	h	
INV: b	???	s = sum	

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

Add elements backwards

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

	0	h	
INV: b	???	s = sum	

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

	0	n
b	???	

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) {
   int e= (h+t)/2;
   if (b[e] <= v) h= e;
   else  t= e;
}
// b[0..h] <= v < b[t..n-1]
```

	0	h	t	n
inv P: b	<= v	?	> v	

b[h+1..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

Insertion sort of b[0..n-1]

```
h= 0;
// invariant: P (below)
while (h < n) {
   Push b[h] down into
     its sorted position
     in b[0..h];
   h= h+1;
}
```

Worst-case time for Push: h swaps

Average case time for Push: h/2 swaps

1 + 2 + 3 + … + n-1 = n (n-1) / 2

Worst-case and average case time: proportional to n^2

	0	h	n
inv P: b	sorted	?	

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) + … + 3 + 2 + 1 = n (n-1) / 2

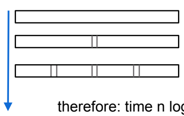
Worst-case and average case time: proportional to n^2

	0	h	n
inv P: b	sorted	?	

Quicksort of b[0..n-1]

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

Quicksort of b[0..n-1]

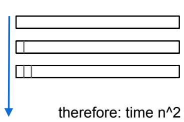
```
/** Sort b[h..k] */
void QS(int[] b, int h, int k) {
  if (b[h..k] size < 2)
     return;
  j= partition(b, h, k);
  // b[h..j-1] <= b[j] <= b[j+1..k]
  QS(h, j-1);
  QS(j+1, k)
}
```

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

Quicksort of b[0..n-1]

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

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

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;
}
```


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?

m 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

Quicksort

pivot

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

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