CS1110 7 April 2010
insertion sort, selection sort, quick sort

Do exercises on pp. 311-312 to get familiar with concepts and develop skill. Practice in DrJava! Test your methods!

A5 times
median 6.5
mean 7.6

10-14: 23
15-19: 2
20-21: 4

Comments on A5

Liked not having to write test cases!

Recursion:
Make requirements/descriptions less ambiguous, clearer; give more direction.
Need optional problem with more complicated recursive solution would have been an interesting challenge, more recursive functions. They make us think!
Make task 5 easier. I could not finish it.

Needed too much help, took too long

Add more methods; it did not take long

Allow us to do recursive methods with loops rather than recursively.

I had intended here to erupt in largely incoherent rage over that wretched concept of recursion, which I came to hate like an enemy: like a sentient being who, knowing the difference between right and wrong, had purposely chosen to do me harm. However, I then figured out how it works, and it is actually quite elegant, so now I suppose I have learned something against my will.

Good time drinking beer while watching the demo after I was done.

Sorting:

"sorted" means in ascending order

pre: b [0 ? n]   post: b [0 sorted n]

insertion sort inv: b [0 sorted i ? n]

for (**int** i= 0; i < n; i= i+1) {

}

0 i
2 4 4 6 6 7 | 5 3 5 2 8

Draw a diagram here to show what the array has to look like before i can be incremented so that inv is true after incrementing i

0 i
2 4 4 5 6 6 | 7 3 5 2 8

0 i
2 4 4 5 6 6 7 | 3 5 2 8

State in English what you did.
Don't mention loops or anything like that.

Sorting:

"sorted" means in ascending order

pre: b [0 ? n]   post: b [0 sorted n]

insertion sort inv: b [0 sorted i ? n]

for (**int** i= 0; i < n; i= i+1) {
Push b[i] down into its sorted position in b[0..i];
}

0 i
2 4 4 6 6 7 | 5

0 i
2 4 4 5 6 6 | 7

Iteration i makes up to i swaps.
In worst case, number of swaps needed is
0 + 1 + 2 + 3 + … (n-1) = (n-1)*n / 2.

Called an "n-squared", or n^2, algorithm.

b[0..i-1]: i elements

in worst case:

Iteration 0: 0 swaps
Iteration 1: 1 swap
Iteration 2: 2 swaps
…

pre: b [0 ? n]   post: b [0 sorted n]

insertion sort invariant: b [0 sorted i ? n]

Add property to invariant: first segment contains smaller values.

selection sort invariant: b [0 ≤ b[i..], sorted i ≥ b[0..i-1], ? n]

for (**int** i= 0; i < n; i= i+1) {

}

i n
2 4 4 6 6 | 8 9 9 7 8 9

Draw diagram to show what array has to look like so that incrementing i makes inv true

i n
2 4 4 6 6 | 7 9 9 8 8 9

i n
2 4 4 6 6 8 9 | 9 7 8 9

State in English what you did.

pre: b [0 ? n]   post: b [0 sorted n]

insertion sort invariant: b [0 sorted i ? n]

Add property to invariant: first segment contains smaller values.

selection sort invariant: b [0 ≤ b[i..], sorted i ≥ b[0..i-1], ? n]

for (**int** i= 0; i < n; i= i+1) {
int j= index of min of b[i..n-1];
Swap b[j] and b[i];
}

i n
2 4 4 6 6 | 8 9 9 7 8 9

i n
2 4 4 6 6 | 7 9 9 8 8 9

Also an "n-squared", or n^2, algorithm.

Partition algorithm: Given an array b[h..k] with some value x in b[h]:

P: b [h: **x** | ? :k]

Swap elements of b[h..k] and store in j to truthify P:

Q: b [h: <= **x** | j: **x** | >= **x** :k]

change: b [h] **3** 5 4 1 6 2 3 8 1 [k]

into b [h] 1 2 1 **3** [j] 5 4 6 3 8 [k]

or b [h] 1 2 3 1 **3** [j] 4 5 6 8 [k]

x is called the pivot value.

x is not a program variable; x just denotes the value initially in b[h].

7

Quicksort

```
/** Sort b[h..k] */
public static void qsort(int[] b, int h, int k) {
    if (b[h..k] has fewer than 2 elements)
        return;
    int j= partition(b, h, k);
    // b[h..j–1] <= b[j] <= b[j+1..k]
    // Sort b[h..j–1] and b[j+1..k]
    qsort(b, h, j–1);
    qsort(b, j+1, k);
}
```

[h] 6 3 2 5 1 [j] 6 [ ] 7 8 6 9 [k]

Write a diagram that shows this arrays sorted

Write down in English what you did

pre: b [h: **x** | ? :k]

j= partition(b, h, k);

post: b [h: <= **x** | j: **x** | >= **x** :k]

8

Quicksort

```
/** Sort b[h..k] */
public static void qsort(int[] b, int h, int k) {
    if (b[h..k] has fewer than 2 elements)
        return;
    int j= partition(b, h, k);
    // b[h..j–1] <= b[j] <= b[j+1..k]
    // Sort b[h..j–1] and b[j+1..k]
    qsort(b, h, j–1);
    qsort(b, j+1, k);
}
```

To sort array of size n. e.g. 2^{15}

Worst case: n^2 e.g. 2^{30}

Average case: n log n. e.g. $15 * 2^{15}$

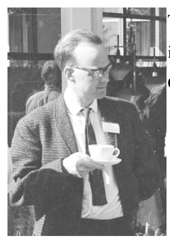
$2^{15} = 32768$

pre: b [h: **x** | ? :k]

j= partition(b, h, k);

post: b [h: <= **x** | j: **x** | >= **x** :k]

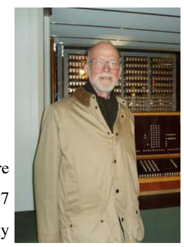
9



Tony Hoare, in 1968

Quicksort author

Tony Hoare in 2007 in Germany



Thought of Quicksort in ~1958. Tried to explain it to a colleague, but couldn't.
Few months later: he saw a draft of the definition of the language Algol 58 –later turned into Algol 60. It had recursion.
He went and explained Quicksort to his colleague, using recursion, who now understood it.

10

The NATO Software Engineering Conferences

homepages.cs.ncl.ac.uk/brian.randell/NATO/

7-11 Oct 1968, Garmisch, Germany
27-31 Oct 1969, Rome, Italy

Download Proceedings, which have transcripts of discussions.
See photographs.

Software crisis:
Academic and industrial people. Admitted for first time that they did not know how to develop software efficiently and effectively.

Software Engineering, 1968

Next 10-15 years: intense period of research on software engineering, language design, proving programs correct, etc.

12

Software Engineering, 1968

13

During 1970s, 1980s, intense research on
How to prove programs correct,
How to make it practical,
Methodology for developing algorithms

The way we understand recursive methods is based on that methodology.
Our understanding of and development of loops is based on that methodology.

Throughout, we try to give you thought habits to help you solve programming problems for effectively

Mark Twain: Nothing needs changing so much as the habits of **others**.

14

The way we understand recursive methods is based on that methodology.
Our understanding of and development of loops is based on that methodology.

Throughout, we try to give you thought habits to help you solve programming problems for effectively

Simplicity is key:
Learn not only to simplify, learn not to complify.

Don't solve a problem until you know what the problem is (give precise and thorough specs).

Separate concerns, and focus on one at a time.

Develop and test incrementally.

Learn to read a program at different levels of abstraction.

15