University of British Columbia
CPSC 111, Intro to Computation
2009W2: Jan-Apr 2010
Tamara Munzner

Yet More Array Practice

Lecture 24, Mon Mar 15 2010

borrowing from slides by Kurt Eiselt

http://www.cs.ubc.ca/~tmm/courses/111-10

Reading: CORRECTION!

- This week: Sorting and Searching
  - 14.1 and 14.3 in 3rd edition
  - (19.1 and 19.3 in 2nd edition)
  - so weekly question is indeed required!
- course web page has also been updated

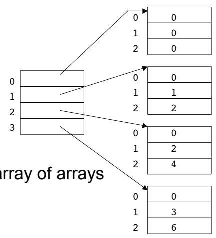
Midterm 2

- Midterm 2: Mon Mar 22, 6:30pm
  - FSC 1005 again
  - hour-long exam, reserve 6:30-8 time slot
    - for buffer in case of fire alarms etc
- coverage: through arrays (Chap 8)
  - includes/builds on material covered in previous midterm
- study tips - same as before!
  - write and test programs, not just read book
    - try programming exercises from book

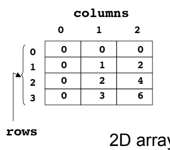
Reading through Midterm, 3rd edition

- 1.1-1.8
- 2.1-2.10
- 3.1-3.8
- 4
- 5.1-5.4
- 6.1-6.5
- 7.1,7.5-7.7
- 14.1,14.3
  - see course page for 2nd edition list

Recap: Arrays of Arrays = 2D Arrays



array of arrays



2D array

- 2D array often easier to think about
- Internally, 2D arrays implemented as arrays of arrays in Java
  - they're equivalent

Recap: 2D Array Access Patterns

scores

cols: quizzes

	0	1	2	3
0	95	82	13	96
1	51	68	63	57
2	73	71	84	78
3	50	50	50	50
4	99	70	32	12

rows: students

- Print average score for each student
  - for each row of scores
    - add up scores
    - divide by number of quizzes
      - length of row

```
for (int row = 0; row < scores.length; row++) {
  double average = 0;
  for (int col = 0; col < scores[row].length; col++) {
     average = average + scores[row][col];
  }
  average = average / scores[row].length;
}
```

- Print average score for each quiz
  - for each column of scores
    - add up scores
    - divide by number of students
      - length of column

```
for (int col = 0; col < scores[0].length; col++) {
  double average = 0;
  for (int row = 0; row < scores.length; row++) {
    average = average + scores[row][col];
  }
  average = average / scores.length;
}
```


Recap: Per-Student Averages

```
public class ArrayEx4
{
  public static void main(String[] args)
  {
    double[][] scores = {{95, 82, 13, 96},
      {51, 68, 63, 57}, {73, 71, 84, 78}, {50, 50, 50, 50},
      {99, 70, 32, 12}};
    double average;

    // here's where we control looping row by row (student by student)
    for (int row = 0; row < scores.length; row++)
    {
      average = 0;
      // and here's where we control looping through the columns
      // (i.e., quiz scores) within each row
      for (int col = 0; col < scores[row].length; col++)
      {
        average = average + scores[row][col];
      }
      average = average / scores[row].length;
      System.out.println("average of row " + row + " is " + average);
    }
  }
}
```


Recap: Selection Sort

0	3
1	8
2	19
3	16
4	12

- Start at beginning
- Consider unsorted array elements: beyond current spot
  - Find smallest element
  - Swap with current spot
  - Move down by one

The smallest value so far is 12

Its index is 4

Multidimensional Arrays

- now that we know 2D, we can do nD!
  - any number of dimensions: 3D, 4D…
    - up to 127D, actually
- example: student quiz scores over multiple terms
  - row: students
  - col: quiz scores
  - stack: term
    - let's try it!

05-06 Term2

05-06 Term1

04-05 Term2

95	82	13	96
51	68	63	57
73	71	84	78
50	50	50	50
99	70	32	12

04-05 Term1

Now for the good stuff

Computer science folks don't spend all their time writing programs. They're also concerned with the efficiency of those programs and their underlying algorithms. Efficiency can be expressed in terms of either time or memory needed to complete the task. In the case of sorting algorithms, we're typically interested in how much time it takes to sort.

So let's try to get some sense of the time requirements of selection sort. We don't use a stopwatch...instead, we use mathematics. The fundamental operation in sorting is the comparison to see if one value is less than the other, and the time required to sort corresponds to the number of comparisons that must be made to complete the sorting.

Estimating time required to sort

0	16
1	3
2	19
3	8
4	12

We can go back to the selection sort example and count the comparisons. The first pass through the array of 5 elements started with 16 being compared to 3, then 3 was compared to 19, 8, and 12. There were 4 comparisons. The value 3 was moved into the location at index 0.

Estimating time required to sort

0	3
1	16
2	19
3	8
4	12

We can go back to the selection sort example and count the comparisons. The first pass through the array of 5 elements started with 16 being compared to 3, then 3 was compared to 19, 8, and 12. There were 4 comparisons. The value 3 was moved into the location at index 0. Then the second pass through the array began, starting with index 1. 16 was compared to 19, then 16 was compared to 8, which became the new minimum and was compared to 12. So among 4 elements in the array, there were 3 comparisons.

Estimating time required to sort

0	3
1	8
2	12
3	16
4	19

It takes 4 passes through the array to get it completely sorted. There are 4 comparisons on the first pass, 3 comparisons on the second pass, 2 comparisons on the third pass, and 1 comparison on the last pass. That is, it takes 4 + 3 + 2 + 1 = 10 comparisons to sort an array of five values.

If you do this same computation on an array with six values, you'll find it takes 5 + 4 + 3 + 2 + 1 = 15 comparisons to sort the array. Do you see a pattern?

With a little math, you can figure out that the number of comparisons required to perform selection sort on an array of N values is given by the expression: $N*(N-1)/2$ or $(N^2-N)/2$

Estimating time required to sort

Either way, it should be easy to see that as N, the number of values in the array gets very big, the number of comparisons needed to sort the array grows in proportion to N^2, with the other terms becoming insignificant by comparison.

So sorting an array of 1,000 values would require approximately 1,000,000 comparisons. Similarly, sorting an array of 1,000,000 values would take approximately 1,000,000,000,000 comparisons.

As the number of values to be sorted grows, the number of comparisons required to sort them grows much faster. Fortunately, there are other sorting algorithms that are much less time-consuming, but we won't be talking about them in this class. In the meantime, here are some real numbers to help you think about just how long it might take to sort some really big arrays...

Estimating time required to sort

Let's assume that your computer could make 1 billion (1,000,000,000) comparisons per second. That's a lot of comparisons in a second. And let's say your computer was using selection sort to sort the names of the people in the following hypothetical telephone books. Here's some mathematical food for thought.

phone book	number of people (N)	N^2	number of seconds needed to sort

Estimating time required to sort

Let's assume that your computer could make 1 billion (1,000,000,000) comparisons per second. That's a lot of comparisons in a second. And let's say your computer was using selection sort to sort the names of the people in the following hypothetical telephone books. Here's some mathematical food for thought.

phone book	number of people (N)	N^2	number of seconds needed to sort	
Vancouver	544,320	296,284,262,400	296	or 5 minutes

Estimating time required to sort

Let's assume that your computer could make 1 billion (1,000,000,000) comparisons per second. That's a lot of comparisons in a second. And let's say your computer was using selection sort to sort the names of the people in the following hypothetical telephone books. Here's some mathematical food for thought.

phone book	number of people (N)	N^2	number of seconds needed to sort	
Vancouver	544,320	296,284,262,400	296	or 5 minutes
Canada	30,000,000	900,000,000,000,000	900,000	or 10.4 days

Estimating time required to sort

Let's assume that your computer could make 1 billion (1,000,000,000) comparisons per second. That's a lot of comparisons in a second. And let's say your computer was using selection sort to sort the names of the people in the following hypothetical telephone books. Here's some mathematical food for thought.

phone book	number of people (N)	N^2	number of seconds needed to sort	
Vancouver	544,320	296,284,262,400	296	or 5 minutes
Canada	30,000,000	900,000,000,000,000	900,000	or 10.4 days
People's Republic of China	1,000,000,000	1,000,000,000,000,000,000	1,000,000,000	or 31.7 years

Estimating time required to sort

Let's assume that your computer could make 1 billion (1,000,000,000) comparisons per second. That's a lot of comparisons in a second. And let's say your computer was using selection sort to sort the names of the people in the following hypothetical telephone books. Here's some mathematical food for thought.

phone book	number of people (N)	N^2	number of seconds needed to sort	
Vancouver	544,320	296,284,262,400	296	or 5 minutes
Canada	30,000,000	900,000,000,000,000	900,000	or 10.4 days
People's Republic of China	1,000,000,000	1,000,000,000,000,000,000	1,000,000,000	or 31.7 years
World	6,000,000,000	36,000,000,000,000,000,000	36,000,000,000	or 1142 years

Favorite Colors

- record everybody's favorite color
- how can we do "averages" per row?