Computer Science E214

Tutorial 8

2014

Do the following exercises. Note that this tut combines both performance (Chapter 4.1) and sorting/searching (Chapter 4.2).

1 Book 3.3.15

Add code to `Counter` (Program 3.3.2 in the textbook) to throw a `RuntimeException` if the client tries to construct a `Counter` object using a negative value for `max`.

You can read about exceptions in the textbook towards the end of Chapter 3.3. These are extremely important to know about, since most error handling in current programming methodologies make use of exception throwing and handling.

2 Book 4.1.7

Use tilde notation to simplify each of the following formulas, and give the order of growth of each:

1. $N(N-1)(N-2)(N-3)/24$
2. $(N-2)(\lg N-2)(\lg N+2)$
3. $N(N+1)-N^2$
4. $N(N+1)/2+N\lg N$
5. $\lg((N-1)(N-2)(N-3))^2$

3 Book 4.1.9

Determine whether the following code fragment is linear, quadratic, or cubic (as a function of N). You should answer this question analytically and explain your reasoning. You can use `StopWatch.java` to measure the runtime of the code.

```
for (int i = 0; i < N; i++) {
   for (int j = 0; j < N; j++) {
      if (i == j) c[i][j] = 1.0;
      else        c[i][j] = 0.0;
   }
}
```

HINT: You may run into trouble with having too little memory for large N. Google for `java adjust heap size`.

4 Book 4.1.14

Apply the scientific method to develop and validate a hypothesis about order of growth of the running time of `Markov.java` (PROGRAM 1.6.3) as a function of the input parameters T and N. Use the methods discussed in SECTION 4.1 of the textbook. Use the data files given: `tiny.txt`, `medium.txt`, `large.txt`, and `extralarge.txt`.

Note again, the resulting formula will be a function of both T and N. For a given N (this is chosen by choosing a file), you have to run your program for a variety of T values. This will form a two-dimensional table, which you can use to develop your hypothesis.

Remember to verify your result! Predict the time for `extralarge.txt` for various values of T using the model you developed.

5 Book 4.1.17

The following code fragment (adapted from a Java programming book) creates a random permutation of the integers from 0 to $N - 1$. Determine the order of growth of its *average* running time as a function of N. Compare its order of growth with `Shuffle.java` from SECTION 1.4. (Note the similarities between this approach to shuffling and the coupon collector problem of SECTION 1.4.)

```
int[] a = new int[N];
boolean[] taken = new boolean[N];
int count = 0;

while (count < N) {
  int r = StdRandom.uniform(N);
  if (!taken[r]) {
    a[r] = count;
    taken[r] = true;
    count++;
  }
}
```

Hint: if you repeatedly do something with probability of success p, the expected (i.e. average) number of repetitions until you succeed is $1/p$.

6 Book 4.1.35

Array rotation. Given an array of N elements, give a linear time algorithm to rotate the array k positions in-place (i.e. using at most a constant amount of extra memory except for that used by the array given). That is, if the array contains $a_0, a_1, \ldots, a_{N-1}$, the rotated array is $a_k, a_{k+1}, \ldots, a_{N-1}, a_0, \ldots, a_{k-1}$.

Hint: you can do this by reversing three subsections of the array in-place. The solution to EXERCISE 4.1.20 might help you understand how to do this. However, note that that solution does not reverse the string in-place because Java strings are immutable. How could you do it in-place if you were instead provided with a `char[]`?

7 Book 4.2.7

Modify PROGRAM 4.2.3 `BinarySearch` so that if the search key is in the array, it returns the smallest index i for which $a[i]$ is equal to *key*, and otherwise, it returns $-i$, where i is the smallest index such that $a[i]$ is greater than *key*. (If there is no $a[i]$ greater than the key return $-N$, where N is the size of the array.)

What is the danger of testing whether the returned value is negative to say whether an element is in the array or not?

8 Book 4.2.8

Figure out what happens if you apply binary search to an *unordered* array. What is the running time now? Why shouldn't you check whether the array is sorted before each call to binary search? Could you check that the elements binary search examines are in ascending order? Modify the binary search code in the book to do so — without changing the order complexity of the solution.

9 Book 4.2.26

Finding a majority. Consider an array of elements. An element is a majority if it appears more than $N/2$ times in the array (and hence there is at most one such element). Write a static method that takes an array of N strings as argument and identifies a majority (if it exists) *in linear time.*

Hint (if you like a challenge, don't read on!): sweep through the array, maintaining a pair consisting of the current candidate and a counter. Initially, the current candidate is unknown and the counter is 0. When we move the pointer forward over an element e

- if the counter is 0, we set the current candidate to e and we set the counter to 1.
- if the counter is not 0, we increment or decrement the counter according to whether or not e is the current candidate.

When we are done, the current candidate is the majority element, if there is a majority. You must take one more linear pass through the data to confirm that the chosen element is indeed the majority

It is a useful exercise to convince yourself why this approach works.

10 Book 4.2.31

Partitioning. Write a static method that sorts a `Comparable` array that is known to have a most two different values *in linear time.*

Hint: Maintain two indices, one starting at the left end and moving right, the other starting at the right end and moving left. Maintain the invariant that all elements to the left of the left pointer are equal to the smaller of the two values and all the elements to the right of the right pointer are equal to the larger of the two values.

11 Book 4.2.32

Dutch national flag. Write a static method that sorts a `Comparable` array that is known to have at most three different values. (Edsgar Dijkstra named this the Dutch-national-flag problem because the result is three "stripes" of values like the three stripes in the flag.)

Hint: You can use the solution to Exercise 4.2.31, by first partitioning the array into two parts with all elements having the smallest value in the first part and all other elements in the second part, then partition the second part.

12 Book 4.2.33

Quicksort. Write a recursive program that sorts an array of randomly ordered distinct `Comparable` elements. Make use of a method like that described in Exercise 4.2.32: First, partition the array into a left part with all elements less than some array element v, then v itself, followed by a right part with all elements greater than v. Then, recursively sort the two parts. Modify your method to work properly when the elements are not necessarily distinct.

What are the best and worst case input array patterns for this approach?

Investigate the order of growth of the running time of your solution experimentally in the following cases:

- the input array is in a random order;
- the input array is in increasing order; and
- the input array is in decreasing order.

13 Book 4.2.37

Rhyming words Write a program `Rhymer.java` that tabulates a list that you can use to find words that rhyme. Use the following approach

- read in a dictionary of words into an array of strings;
- reverse the letters in each word (*confound* becomes *dnuofnoc*, for example);
- sort the resulting array;
- reverse the letters in each word back to their original order.

For example, *confound* is adjacent to words such as *astound* and *surround* in the resulting list.

A sample input file `words.txt` can be found in the resources section for this tutorial.