COMP 250: Introduction to Computer Science
Assignment 3

Posted Tuesday, March 4, 2014
Due Monday, March 17, 2014

Please submit the homework through myCourses before midnight on the day it is due.

1. [30 points] **Pascal's triangle**

   The following pattern:

   1
   1 1
   1 2 1
   1 3 3 1
   1 4 6 4 1
   1 5 10 10 5 1

   is called Pascal's triangle. The leftmost column and the diagonal consist of 1s. Each cell contains the sum of the number immediately above, and immediately above and to the left. For example, the number 6 in the 4th row is the sum of the 3 and 3 in row 3.

   (a) [20 points] Write a Java class, called Pascal, with one static recursive method, called pascalTriangle, which takes as arguments two integers, m and n and returns the number appearing in the mth position of the nth row. For example, Pascal(4,0)=1, Pascal(4,1)=4, Pascal(4,2)=6, etc. Your class should also have a main method, in which you should print the triangle above by calling pascalTriangle appropriately.

   Solution: The solution is in the file Pascal.java

   (b) [5 points] Prove by induction that the sum of all elements in the nth row of the triangle is 2^n.

   Solution:

   Base case: For $n = 0$, $1 = 2^0$.

   Induction step: Let $P_{i,j}$ denote the element in the ith row and jth column of the triangle. In the $(i + 1)$st row, we have:

$$\begin{aligned}\sum_{j=0}^{i+1} P_{i+1,j} &= 1 + \sum_{j=1}^{i+1} P_{i+1,j} = 1 + \sum_{j=1}^{i}(P_{i,j} + P_{i,j-1}) + 1 \\ &= P_{i,0} + \sum_{j=1}^{i} P_{i,j} + \sum_{j=1}^{i} P_{i,j-1} + P_{i,i} = \sum_{j=0}^{i} P_{i,j} + \sum_{j=0}^{i} P_{i,j} = 2^n + 2^n = 2^{n+1}\end{aligned}$$

where the second-to-last equality comes form the induction hypothesis.

(c) [5 points] Prove that if you start at any leftmost 1 (in a row i) in the triangle and take a diagonal of any length $(i, 0)$, $(i + 1, 1)$, $\ldots$, $(i + j, j)$ you obtain the element at location $(i + j + 1, j)$.

Solution: We will prove this by induction on j, considering a fixed, arbitrary starting row i.

Base case: for $j = 0$, we have $P_{i,0} = P_{i+1,0} = 1$.

Induction step: Suppose $\sum_{k=0}^{j} P_{i+k,k} = P_{i+j+1,j}$ (this is the induction hypothesis). Then:

$$\sum_{k=0}^{j+1} P_{i+k,k} = \sum_{k=0}^{j} P_{i+k,k} + P_{i+j+1,j+1} = P_{i+j+1,j} + P_{i+j+1,j+1} = P_{i+j+2,j+1}$$

where the second-to-last equality is from the induction hypothesis and the last one is from the definition of the Pascal triangle.

2. [30 points] **Stacks and Queues**

You can solve this problem either in pseudocode or in Java. If you use Java, please use the generic classes provided by the API.

(a) Write a method reverseQueue which takes as argument a queue and modifies it to have the content reversed. You may use one stack as an additional data structure. Give the $O()$ for the running time of your method.

Solution: We will take all elements from the queue, put them on the stack, then put them on the queue again. Since the stack is LIFO, the last element that came out from the queue will end up on top of the stack, and hence be the first one popped. Enqueueing will then give us the desired reverse order. In Java-ish pseudocode:

```
Algorithm reverseQueue (Queue q)
Stack s
Object x
while (!q.isEmpty()) do
    x = q.dequeue()
    s.push(x)
while (!s.isEmpty()) do
    x = s.pop()
    q.enqueue(x)
```

or alternatively:

```
Algorithm reverseQueue (Queue q)
Stack s
while (!q.isEmpty()) do
    s.push(q.dequeue())
while (!s.isEmpty()) do
```

```
    q.enqueue(s.pop())
```

This is $O(n)$, where n is the size of the queue, since we have a sequence of two loops each of which is $O(n)$.

(b) Write a method copyStack that takes as argument a stack s and returns a new stack containing the same elements and in the same order as s. Before the method finishes, it must restore the contents of s to its original state (same contents in the same order). Besides the new stack that the method returns, the only additional data structure that it can use is a single queue. The method may also use O(1) additional space. Give the $O()$ for the running time of your method.

Solution: We will take the elements from the first stack and copy them onto both the queue and the second stack Note that this will reverse them. We then go from the copy on the stack back to the original, to restore it. We now have to copy from the queue to the stack, back to the queue and back to the stack again (in order to make sure the order it correct).

```
Algorithm copyStack (Stack s)
Stack sCopy
Object x
Queue q
while (!s.isEmpty()) do
    x = s.pop()
    q.enqueue(x)
    sCopy.push(x)
while (!sCopy.isEmpty()) do
    s.push(sCopy.pop())
while (!q.isEmpty()) do
    sCopy.push(q.dequeue())
while (!sCopy.isEmpty()) do
    q.enqueue(sCopy.pop())
while (!q.isEmpty()) do
    sCopy.push(q.dequeue())
```

The algorithm is $O(n)$ since we have a sequence of loops each of which is $O(n)$.

3. [40 points] **Sorting**

Consider the Sorting package, available on the lectures web page. In this problem, you will work on adding one more algorithm to this package, as well as on benchmarking the code. Benchmarking measures the actual running time of the algorithm (and is very useful in empirical studies). The purpose of this exercise is three-fold:

- To get you to implement some of the algorithms we discuss (and make the leap from pseudocode to code)
- To get you used to looking at the Java API (so you know how to find information there on your own)

- To show an example of (simple) performance evaluation in practice (rather than in theory)

(a) [10 points] Add a class called quickSort, with a method which implements the QuickSort algorithm we discussed in class. You should pick as pivot the median of the first, last and middle elements in the array. Note that you may use extra methods as needed.

Solution The solution is in QuickSort.java

(b) [20 points] Write a new class called SortBenchmark. In this class, you will write a main function which initializes an array of Integer objects of a size that is read from the command line. You will use the java.util.Random class to generate the random values. Please read the API documentation for this class (you will mainly be interested in the constructor that uses a seed, and in the nextInt() method.

Once you create the array, call your quick sort algorithm to sort it, and measure its running time. To do this, a method that will help you is currentTimeMillis() from the System class in the java.lang package. Again, please look at the API to figure out what it does, and how to call it. Print the running time of your algorithm.

Once you have sorted the array, re-initialize it with *the same* random integers. You can do this by setting the seed of the random number generator to the same seed with which you constructed it. Now call the mergeSort() algorithm which is provided to you, measuring and printing its running time as above. Repeat this process again with the selectionSort algorithm that is provided.

Solution: The solution is in SortBenchmark.java

(c) [10 points] Run an experiment with arrays of size 16, 256, 1024, 4096. If possible, keep increasing the size until you get errors for the memory size being too big. Repeat this experiment 5 times.

Draw a graph (in Excel or your favourite graphing program) showing the running times you obtained as a function of the size of the array (one line for each of the 5 repetitions, and for each algorithm). Write a little report including the graph and a brief description of what you found.

Solution: The solution is in experiment.pdf