CS2110 Assignment 4 — Recurrences, Runtime Analysis, Sorting, Stacks, Summer 2008
Due July 23, 2008, 6:00PM

0 Introduction

0.1 Goals

The goal of this assignment is to familiarize you with asymptotic complexity (big-O), run-time analysis, basic data structures, and sorting.

0.2 Submission

Follow the Submission Requirements on the course website. The last section of this assignment summarizes the files that you need to submit on CMS.

0.3 Grading

As before, solutions will be graded on both correctness and style. For correctness, at the very least your program must compile without errors or warnings. For style, we like concise, clear, easy-to-read code. Use mnemonic variable names. Use proper indentation. Comment where necessary for clarification, but do not overcomment. Be concise; a two-page solution to a problem that can be done in a few lines will lose points, even if correct. The assignment will be graded out of 100 points and is worth 10% of your course grade.

1 Big-Oh

Purpose: Understanding the growth of functions, complexity
Points: 25

1. How are the return values of the following functions growing with n? Give your answer in Big-Oh notation. Explain the reasoning behind each answer.

   (a)
   ```
   int sharp(int n){
       int r = 0;
       for(int i=1; i<n; i++)
           for(int j=i+1; j<=n; j++)
               for(int k=j+1; k<=n; k++)
                   r++;
       return r;
   }
   ```
   (b)
   ```
   int cool(int n){
       int r = 0;
       for(int i=1; i<=n; i++)
         for(int j=1; j<=i; j++)
            for(int k=j; k<=i+j; k++)
               for(int l=1; l<=i+j-k; l++)
                    r++;
       return r;
   }
   ```
   (c)
   ```
   int awesome(int n){
       if(n<3) return 1;
   ```

```
        return 4*awesome(n/3)+2*n+1;
    }
```

(d)
```
int sneaky(int n){
        if(n<=1) return n;
        return 5*sneaky(n-1)-6*sneaky(n-2);
    }
```

2. Given an array of N real values, find the pair of numbers such that the difference has the minimum value.
   (a) Give a O(N^2) algorithm.
   (b) Give a O($N \log N$) algorithm.
3. Given an array of N real values, find the pair of numbers such that the difference has the maximum value. Give a O(N) algorithm.

Write your answers in a plain text file called `Runtime.txt`.

2 Data Structures

Purpose: To become familiar with the uses of various data structures
Points: 20

Answer the following questions. You do not have to write working Java code for this problem. You only have to describe the implementation in words and/or in pseudocode.

1. How can we implement a queue using a priority queue? How can we implement a stack using a priority queue?
2. Show how to implement a queue using two stacks. Analyze the running times of the `enQueue` and `deQueue` operations. Also explain how to implement a stack using two queues. Analyze the running times of the `push` and `pop` operations.
3. What are the minimum and maximum number of elements in a heap of height h? Is any array of n elements in sorted (ascending) order a heap? Prove it or give a counterexample.
4. Modify a binary search tree so that it supports the following operation in $O(h)$ time where h is the height of the tree. The operation is `delete(k)` which deletes the kth smallest element from the tree. Also explain how the methods `insert(x)` and `find(x)` need to be modified in order to keep the extra information stored in the tree consistent. Show that your modifications do not affect the running time of these operations. You may assume that all keys in the tree are unique.

Submit your answers in `DataStructures.txt`.

3 Sorting

Purpose: To illustrate some ideas about sorting
Points: 20

Your friend Ada Byron is working on a program that will automatically create an index of a given text. She wants the index to list each word in the text in alphabetical order. For each word, she wants the page number(s) that the word appears on to be listed in ascending order. She has written most of the program, but she is having some trouble getting the sorting to work properly. In her implementation, the index is stored as an array of `IndexEntry` objects. She has defined the `IndexEntry` class as follows:

```
enum KeyType{TERM, PAGE };

class IndexEntry implements Comparable<IndexEntry> {
    private static KeyType key;
    private String term;
    private int page;
    public IndexEntry(String s, int p) {
        term = s;
        page = p;
    }
    public static void setKey(KeyType k) {
        key = k;
    }
    public int compareTo(IndexEntry e) {
        switch(key){
            case TERM:
                return term.compareTo(e.term);
            case PAGE:
                return page - e.page;
            default:
                return 0;
        }
    }
}
```

Ada wants to sort the array so that the `IndexEntry` objects are ordered alphabetically by term, and then sorted by ascending page number for objects having the same term. To do this she uses her favorite algorithm, heapsort, in the following way:

```
SortingAlgorithm h = new Heapsort();
IndexEntry.setKey(KeyType.PAGE);
h.sort(index);
IndexEntry.setKey(KeyType.TERM);
h.sort(index);
```

Unfortunately, she has found that this code does not sort the index entries properly. She has verified that after the first call to heapsort the entries are in ascending order of page number. After the second call to heapsort, the terms are sorted alphabetically but order of page numbers for objects having the same term are all mixed up. Ada has thought of some fixes for the problem, like performing a separate page number sort operation on objects having the same term once the first sort is complete. But these solutions seem computationally expensive and not very elegant.

After some thought, you reply that she probably needs a *stable* sorting algorithm. A stable sorting algorithm is guaranteed to maintain the order of elements having the same key. Some of the algorithms we have discussed in class are stable, but some are not. Heapsort is not a stable algorithm, which is why Adas code didnt work.

To demonstrate this, implement **two** (or more) sorting algorithms of your choice. At least one algorithm must be unstable, and at least one must be stable. Heapsort may *not* be one of the algorithms you implement. Your program will take two command line arguments: a filename specifying an (unsorted) input index, and a number indicating which sorting algorithm the user would like to use. If either argument is omitted, the program should print a helpful error message that includes which sorting algorithms are implemented and how to specify them (e.g. 1=quick sort, 2 = shell sort, etc.).

Each line of the input file has a word followed by a page number. For example,

```
autism 1025
algae 47
Algeria 13
```

```
amoeboid 63
Algeria 101
anarchism 5
autism 57
algae 68
Algeria 25
anarchism 581
autism 831
```

Your program should load the file into an array of `IndexEntry` objects, create an instance of your sorting algorithm, sort by page number, and then sort by term, using one of the sorting routines you have written.

Then it should pretty print the sorted index to the screen, like this:

```
algae 47, 68
Algeria 13, 25, 101
amoeboid 63
anarchism 5, 581
autism 57, 831, 1025
```

Submit your program in `IndexSort.java`. You can find some skeleton code and some test input files on CMS. Also provide a plain text file `IndexSortNotes.txt` that describes which sorting algorithms you chose to implement, and explain why each algorithm is or is not stable.

4 The 2110 Arboretum

Purpose: Stacks
Points: 35

In this exercise, we will explore an interesting application that lets you implement and appropriately use stacks, as well as reinforce your understanding of recursion and grammars.

4.1 Recursion

As we learnt in class, recursion is a powerful technique for solving problems based on the observation that sub-problems have a structure similar to that of the bigger problem. A recursive function can therefore call itself to solve its sub-problems, and then use these solutions to solve the current problem.

Recursion is a concept that manifests itself even outside the realm of CS2110! Snowflakes, clouds, coastlines and plants like ferns exhibit some level of structural self-similarity: when these are viewed at different levels of resolution, you see the same repetitive patterns. Such shapes are examples of *fractals*. Using our grasp of recursion, we can exploit the self-similarity in these shapes to render realistic images of fractals.

For example, in Figure 1 we see that the pattern visible in the big square is scaled down and replicated in four smaller squares within the square, which is further replicated in four squares in those squares and so on. The self-similarity in this image naturally lends itself to a recursive implementation, where in the base case (when the square is too small), we can imagine just drawing a simple square. Such self-similarity is also evident in the bush-like structures we see in Table 1.

4.2 Grammars and L-systems

In class, a grammar was introduced as a set of recursive rules that help to generate the sentences in a language. Such recursive rules can also be used to model the structure of fractals. For this exercise, we look at a variant of grammars, known as L-systems. These too contain symbols and recursive rules. A sentence is derived by multiple applications of the rules of a grammar, until ultimately all the non-terminals have been replaced by terminals. Likewise, some fractal images can be created by repeatedly applying the rules of an L-system.

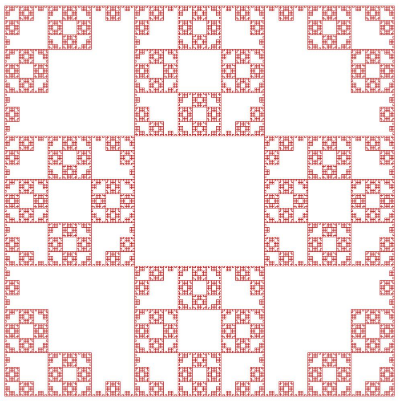
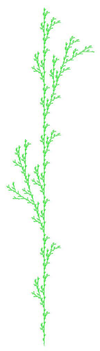
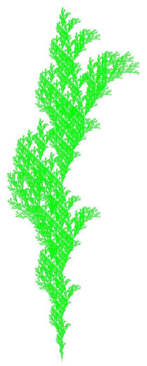
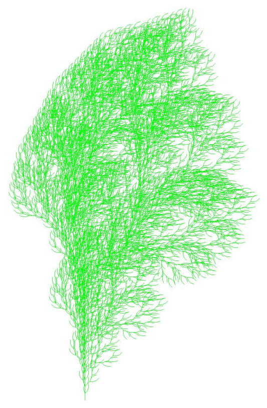


Figure 1: A fractal image with tiled self-similarity

Table 1: The bushes in the arboretum, and their corresponding L-systems

		
A	B	C
$\delta = 25.7°$	$\delta = 20°$	$\delta = 22.5°$
F → F[+F]F[-F]F	F → F[+F]F[-F][F]	F → FF-[-F+F+F]+[+F-F-F]

The difference between grammars and L-systems is the number of rules applied during each step. To produce a sentence using a grammar, one production rule is applied during each derivation step. In other words, one non-terminal symbol is replaced (or rewritten) by applying an appropriate rule (that has the symbol on the left-hand side of the rule). L-systems, on the other hand, apply all possible rules simultaneously during each step. As a result, *all* non-terminals are replaced using appropriate rewrite rules.

To make things more precise, the following L-system was used to synthesize image A in Table 1:

$$\delta = 25.7°$$

F → F[+F]F[-F]F

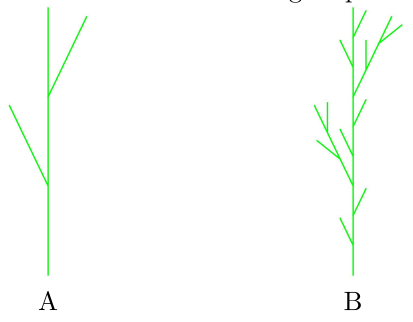
Before you get bewildered by the syntax, note that each of the symbols in this L-system have a simple interpretation as rules for drawing on a canvas. Consider a turtle (or cursor) positioned on a canvas at co-ordinates (x, y), with an orientation α. Note that for these images, the origin is located at the top-left corner of the image: the x-coordinate increases from left to right, and the y-coordinate increases from top to bottom. Let d be a parameter known as the step-size.

Initially the turtle is positioned near the base of the image and faces up, i.e. the north direction ($\alpha = 90°$). The symbol F means that the turtle draws a line of length d in its current direction α; after drawing the line the turtle's new position is

$$x' = x + d * cos(\alpha), \quad y' = y - d * sin(\alpha)$$

Until we reach the base case (for example $d < 3$) , we replace this line of length d by applying the rewrite rule for the system using a reduced value of d. The symbol '-' indicates that the turtle turns right by the angle δ, i.e. $\alpha = \alpha - \delta$; while '+' indicates that the turtle turns left by δ ($\alpha = \alpha + \delta$). The symbol '[' indicates that you save the current state of the turtle onto a stack (symbolizing the beginning of a new branch), while ']' means that you replace the current state of the turtle with one popped from the stack. Figure A in Table 2 demonstrates a single application of the rewrite rule F → F[+F]F[-F]F, while Figure B shows the result of replacing each line in A using the same rule. Repeated applications of this rule produce a bush-like shape as shown in Table 1.

Table 2: Demonstration of Edge-replacement



4.3 Implementation Details: What you need to do

For this exercise, we'd like you to implement a program that produces the three fractal images in Figure 1, using the provided L-system definitions.

We provide a class `TState`, whose objects contain information about the turtle state, namely its position and orientation. We require you to implement the class `TStack`, which basically implements a `Stack` interface like the one presented in class, and is used to maintain a stack of `TState` objects. You may use either an array or linked-list based implementation, but we require you to explain your choice. Also note that you may not directly use the `Stack` class in the Java API — we require you to implement the stack.

In the file `Fractal.java`, we have provided some backbone code that deals with creating and saving images. We have also provided some sample values for parameters you would need to draw the fractal images, primarily the reduction factor(the amount by which the step-size d is reduced at each level of recursion), and the size of the base case. You may change the values if you like, they are not central to the definition of the L-system, and are merely provided to help you get started.

Your file `Fractal.java` will contain a recursive function `recF()` that implements the rewrite rule. This function will process the characters on the right-hand-side of the rule, and either call itself (if the character being processed is 'F'), or take the appropriate action, like modify the turtle state, or save it on the stack. `recF()` will thus make use of the `TStack` class you implement. Note that unlike previous assignments where we supplied function signatures and restricted the interface of the class you implemented, here you get to make your own design decisions. You may define auxiliary helper methods if you need them, and the choice of inputs to the `recF()` function is yours. This increased freedom comes at a price though. Your code must include sufficient comments that precisely explain the purpose of each function. Also, as in the earlier assignments, ensure your code design is meaningful, and that you avoid unnecessary code duplication (make use of functions and loops where possible).

Note that you do not need to know anything about the Java Image API, beyond the fact that you can use the `drawLine()` function of the `Graphics2D` class to draw lines onto the canvas. All the same, we encourage you to experiment with the Image API. For example, you could be creative and modify the turtle state such that different branches of the tree are drawn using different shades of colors, or with different line widths. In addition to the code, we require you to submit the *three best images* you were able to synthesize, one for each of the bush-like structures A, B and C in Table 1 (along with the values of any parameters you used to create those images).

5 What to Submit

Submit the following files:

1. `Runtime.txt`
2. `DataStructures.txt`
3. `IndexSort.java, IndexSortNotes.txt`
4. `TState.java, TStack.java, Fractal.java`
5. 3 output images

6 Extra Credit, Extra Fun!

If you have finished everything above and are craving some more programming, here are a couple extra credit items you can try your hand at. *We will award extra credit based on how well they are done. However, extra credit is always worth less than the main part of the homework. Therefore, your time is best spent finishing the homework well and only working on extra credit afterwards.*

Make sure you keep a pristine backup copy of the solutions before you start.

6.1 Sorting with Compound Keys

Points: 5

An alternative way to solve problem 3 is to sort using a *compound key* and only sort the data once. In a compound key with two components (for example), entries are compared using the first component and ties are broken by using the second component. In the sorting problem above, the first component would be term comparisons and the second would be page number comparisons.

Extend your solution to problem 3 by adding a command line option to enable sorting with a compound key. In particular, you should get the same sort results after sorting the data only once. You will need to modify the `main()` method; make sure that we can still run the non-extra-credit part of the solution.

Start Early and Good Luck! Have Fun!