COMP611 – Algorithm Design and Analysis

Assignment 01

Due: 11.59pm, 19 August 2016

Submission requirements

- You may collaborate with (at most) one other student to complete this assignment.
- Each team must submit a `.zip` file.
- For the code questions, submit only source files (i.e. `.java` files). Do not submit projects or any auxilliary project files.
- For the theory questions, submit a PDF file. We recommend using LaTeX for this, but this is not compulsory.
- Your answer to Question 3 should include your plots. Please provide image files, *not* screenshots.
- Your `.zip` file should contain a `README.txt` file clearly stating your name(s) and student ID(s). This file should also describe which file(s) contains your `main` method(s).

Assignment tasks

Question 1 (12 marks)

State, for each of the following, whether $f(n)$ is $O(g(n))$, $\Omega(g(n))$, $\Theta(g(n))$ or all of the above.

(a)
$$f(n) = 2n + 2^n$$
$$g(n) = 3n + 3^n$$

(b)
$$f(n) = n^{\frac{1}{2}}$$
$$g(n) = n^2 + 3$$

(c)
$$f(n) = n - 100$$
$$g(n) = \left(\frac{1}{n}\right)^2$$

(d) $f(n) = n^3 + n^2 \log(n)$
$g(n) = 2n^2 \log(n)$

(e) $f(n) = \log_2(n)$
$g(n) = (\log(n))^2 + \log(n) + 2$

(f) $f(n) = 1 + 2 + 3 + \cdots + n$
$g(n) = 3n^2 + n \log(n)$

Question 2 (16 marks)

Definition 1. *Let f, g be functions. If there exist $c, n_0 > 0$ such that, for all $n > n_0$, $f(n) \leq c \cdot g(n)$, then $f(n)$* is $O(g(n))$.

Let $f(n), g(n), h(n) : \mathbb{N} \to \mathbb{N}$. Using the above definition, prove the following:

(a) **Scaling Property:** For any constant $c > 0$, $c \cdot f(n)$ is $O(f(n))$.

(b) **Transitive Property:** If $f(n)$ is $O(g(n))$ and $g(n)$ is $O(h(n))$, then $f(n)$ is $O(h(n))$.

(c) **Sum Property:** If $f(n), g(n)$ are both $O(h(n))$, then $f(n) + g(n)$ is also $O(h(n))$.

(d) **Product Property:** If $f(n), g(n)$ are both $O(h(n))$, then $f(n) \cdot g(n)$ is $O(h^2(n))$.

Question 3 (32 marks)

Suppose you have k sorted arrays, each with n elements, and you want to combine them into a single sorted array of kn elements. There are two possible strategies:

Strategy One: Using the `merge` procedure for mergesort, merge the first two arrays, then 'merge in' the third, then the fourth, etc.

Strategy Two: Divide and conquer.

(a) **(4 marks)** Analyze the time complexity of the algorithm described in Strategy One.

(b) **(4 marks)** Design and describe in detail an algorithm based on Strategy Two.

(c) **(4 marks)** Analyze the time complexity of your algorithm in (b), which should be more efficient than the algorithm in (a).

(d) **(12 marks)** Implement both algorithms. Your program should generate a 2-dimensional array `data` with dimensions $k \times n$ storing k sorted arrays of randomly-generated integers, with each array being length n. Both implementations should take `data` as input, and merge all k arrays into one single array of length kn.

(e) **(8 marks)** Plot the running times of both algorithms for different values of n (in the range of $100, 200, \ldots, 1000$), and for different values of k (in the range of $100, 200, \ldots, 1000$). Your plot should clearly indicate the difference in time complexity for both algorithms.

Question 4 (40 marks)

A *maze* contains $n \times n$ positions, indexed by pairs (x, y), where x is the row position, and y the column position. A maze is surrounded by walls, and some positions may also be separated by walls. You can move up, down, left or right from any position, provided that there's no wall between you and your destination. A maze is *connected*: you can get from any position in the maze to any other position in the maze (although it may take many steps). The *starting position* of a maze is $(0, 0)$. Additionally, a random position in every maze contains a *treasure*.

Implement a program, containing the following features:

(a) **(16 marks)** A way to randomly generate a maze, given a dimension n. Use *depth-first search* to generate the maze as follows:

  (a) Assume all pairs of positions have walls between them.

  (b) Start at a random position.

  (c) At each step, randomly choose a direction, and determine the next position along this direction. If it has not been visited, break the wall between where you are now and this position and move there; otherwise, backtrack to the previous position and repeat the operation.

(b) **(8 marks)** A way to display a maze in the terminal. Use `#` to indicate a wall, `s` to indicate the starting position, and $\star$ to indicate the treasure location.

(c) **(12 marks)** A 'solver' for a maze. This should solve the maze (by finding a path from the starting position to the treasure), and then display the maze in the terminal, with the path printed using `.`.