General Notes for Example Classes 2 – 4

The next three example classes intend to reinforce your understanding of algorithms through hands-on experience of implementation and empirical analysis.

Before an example class, each group should have finished the coding and testing of program (in Java, C or C++) on your own computers, and have written a report describing the implementation and experiments. The first 30 – 40 minutes of the example class may be used to test and finalize the programs and presentation materials. Then, representatives are chosen by each group to give a demo of their program and experimental results. By the end of the example class, each group should submit to lab tutor: (1) the report (in softcopy or hardcopy); (2) source code, executable and testing data (in softcopy).

Each member of the group should know the program and presentation materials well enough to be able to act as a backup presenter or to help answer questions in the class.Thus if someone is suddenly ill, someone else should be able to step forward and do the presentation for their sick group mate. Otherwise, the performance of the group will suffer and therefore the grade.

Example Class 2 (Week 6 – Week 7)

Applications and Empirical Analysis of Searching Algorithms

Study and identify an example application, where hashing algorthms are used in the process to solve a a real world problem. For example, the problem can be storing and searching of employee information based on NRIC numbers, storing and searching of vehicle information in a taxi company based on car plate registration numbers, storing and searching of book information in a library, or storing and searching of product information in a warehouse. If real world data sets are not available, you may generate synthetic data sets for use in your experiments.

Implement the algorithms of hashing using a programming language of your choice (either Java, C, or C++). Your implementation should include at least two variations of the hashing algorithms so that you can compare, say between closed and open address hashing schemes, between linear probing and double hashing, or betwee choices of hashing functions for storing the data items. Your programs should include a counter that tracks the number of comparisons in each search operation.

Your presentation and report should include

1. Description of the problem scenario and the selected data set;
2. Description of your chosen hashing algorithms;
3. Demo of your implementation of hashing algorithms to search for an entity in the data set; (Show a few searches for both successful and unsuccessful cases.)
4. Based on a **fixed hash table size of 1000**, statistics on the average CPU time and number of comparisons taken to search in data sets of various sizes, ranging from 100, 300, 500, 700 to 900. (Be sure to repeat your search many times to obtain a statistically meaningful performance and report statistics for both successful and unsuccessful cases);
5. Explanation of the results obtained and conclusion on time complexity comparison between your chosen variants of hashing algorthms.

Example Class 3 (Week 9 – Week 10)

Performance comparison between Mergesort and Quicksort

The objective for this class is to perform empirical comparion of time efficiency between the two sorting algorithms, namely Mergesort and Quicksort. For simplicity, suppose the input data are arrays of integers and the algorithms should sort the integers into ascending order.

In this project, the following steps need to be carried out:

1. **Algorithm implementation:** Implement Mergesort and Quicksort in the same programming language (choose one from Java, C or C++). For Mergesort, in order to achieve high speed, you can use an auxiliary array to store the result of merging. For Quicksort, you can follow the pseudo code given in the lecture notes, and choose the pivot element as you see fit.

2. **Generating input data:** Generate arrays of sizes n = 2000, 4000, 6000, 8000, and 10000. For each of the sizes, generate the following three types of data:
   (1) Randomly generated datasets of integers in the range [1 ... n].
   (2) Integers 1, 2, ..., n sorted in ascending order.
   (3) Integers n, n−1, ...,1 sorted in descending order.
   Save each dataset into a text file for record.

3. **Measuring time complexity:** Run your program of each of the two sorting algorithms on the datasets generated in Step 2. Count the number of key comparisons (i.e. comparisons between array elements) for each run. Also record the CPU time for each run. The statistical results (i.e. numbers of key comparisons and CPU times) should be recorded into a table.

4. **Analysis of results:** Draw 6 scatter plots to visualize the running times of the 2 algorithms on the 3 types of data (i.e. integers in random, ascending, and descending orders). The x-axis is the size of input array, and the y-axis is the running time. In each plot, use two sets of dots (with a trend line fitting each set of dots) to represent how the running time increases with the input size. One set of dots represents the numbers of key comparisons and the other the CPU time. Compare your empirical results with theoretical analysis of time complexity.

In your report and presentation, please describe the above steps, as well as results.

Example Class 4 (Week 11 – Week 12)

All-pairs shortest paths

Write a program (in Java, C or C++) that supports query of shortest-path in an undirected unweighted graph G. That is, given two vertices u and v, return a shortest path and its length measured by the number of edges between u and v in G. If there is no path between two query vertices, the program should return special values to indicate that (say, length equal to −1). The program should be able to answer each query promptly, after preprocessing to compute all shortest paths in the graph. The preprocessing can be achieved by running Breadth-first search (BFS) for each vertex of G, since BFS can solve the single-source shortest path problem.

The project can be divided into the following steps:

1. **Generate random graphs of various sizes**: Write a program to generate two sets of random graphs. Each graph in the first set contains 5000 vertices, and each graph in the second set 10000 vertices. Each set contains 5 graphs, with the numbers of edges equal to 1000, 5000, 10000, 50000, and 100000, respectivelly. (*Hint*: To generate a random graph with n vertices, you can repeatedly generate pairs of random numbers between 0 and n–1 to represent edges. However, you should exclude self-loops and parallel edges, i.e. multiple edges between the same pair of vertices.)

2. **Implement BFS algorithm**: Implement the BFS algorithm running on graphs represented by adjacency lists, following the pseudo code in the lecture notes. If the input graph is not connected (i.e. consisting of multiple connected components), repeat calling BFS on different components (with a similar idea as the dfsSweep in the lecture notes).

3. **Preprocessing to compute all-pairs shortest paths**: Run BFS on every vertex as the starting vertex. Then, save the results into some data structure, so that a query can be answered by looking up the records of shortest paths. (*Hint*: You may save the lengths and parent-links of the shortest paths into two matrices, say, L and P, where L[i, j] contains the length of a shortest path between vertices i and j, and P[i, j] contains the index of predecessor of vertex j in the shortest path from vertex i. You are also encouraged to try your own method of saving the results.)

4. **Analysis of results**: For each set of input random graphs with a fixed number of vertices (5000 or 10000), draw a scatter plot with a trend line of best fit, where the x-axis is the number of edges and y-axis the CPU time for preprocessing. Justify the plots by theoretical analysis of time complexity. During the presentation, give a demo of a few queries.

In your report and presentation, please describe the above steps and present the results of empirical analysis.