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 programs (in Java, C, C++, or Python) on your own computers, and have written a report describing the implementation and experiments. The first **30** 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 presentation and demo. By the end of the example class, each group should submit to lab tutor: (1) the project report (in softcopy); (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, for example, if someone is not feeling well, someone else should be able to step forward and do the presentation as a substitute. Otherwise, the performance of the group will suffer and therefore the grade.

Note:

- Make sure that each presentation/demo is limited to **10 minutes**. Having too many PowerPoint slides is not encouraged.
- Write your report concisely. Each report should be **at most 5 pages** long.

Example Class 2 (Week 6 – Week 7)

Applications and Empirical Analysis of Searching Algorithms

Study and identify an example application, where hashing algorithms are used in the process to solve 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. Each group should have implementation and comparative experiments of at least **two variations of the hashing methods**, as follows:

1. Using closed address hashing, compare between different hashing functions (Group 1).
2. Using closed address hashing, compare between hash tables of prime and non-prime sizes (Group 2).
3. Using open address hashing and linear probing, compare between different hashing functions (Group 3).
4. Using open address hashing and double hashing, compare between different rehashing functions (Group 4).
5. Using open address hashing, compare between linear probing and double hashing (Group 5).
6. Using the same hashing function and table size, compare between closed and open address hashing (Group 6).

Your programs should report the **average CPU time** as well as the **average number of key comparisons** for searching in the hash tables using the chosen methods. Note that you may need to use data with a specific distribution (e.g. all data items are multiples of some values) to contrast the performance of different hashing methods under various circumstances.

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, including searches for both successful and unsuccessful cases;
4. Statistics on the average CPU time and the number of key comparisons taken to search in data sets with various load factors (e.g. 0.25, 0.50 and 0.75). Be sure you repeat the searches sufficiently many times so that the performance figures are stable 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 algorithms.

Example Class 3 (Week 9 – Week 10)

Please choose **one** of the following two projects (Project 3A or 3B):

Project 3A: Empirical comparison between Insertion Sort and Mergesort

The objective for this project is to perform empirical comparison of time efficiency between the two sorting algorithms, namely Insertion Sort and Mergesort. 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 the algorithms of Insertion Sort and Mergesort in the same programming language. For Mergesort, in order to achieve high speed, you can use an auxiliary array to store the result of merging.

2. **Generating input data:** Generate arrays of increasing sizes, say in a range from 1000 to 1 million. For each of the sizes, generate the following 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.

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

4. **Analysis of results:** Draw scatter plots to visualize the running times of the two algorithms on the different types of data (i.e. integers in random, ascending, and descending orders). The variable on x-axis is the size of the input array, and the variable on y-axis is the running time. In each plot, show how the running time increases with the input size. Compare your empirical results with theoretical analysis of time complexity.

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

Project 3B: Integration of Mergesort and Insertion Sort

As a divide-and-conquer algorithm, Mergesort breaks the input array into sub-arrays and recursively sort them. When the sizes of sub-arrays are small, the overhead of many recursive calls makes the algorithm inefficient. This problem can be remedied by choosing a small value of **S** as a threshold for the size of sub-arrays. When the size of a sub-array in a recursive call is less than or equal to the value of **S**, the algorithm will switch to Insertion sort, which is efficient for small input. A pseudocode of the modified Mergesort is given below:

```
void mergeSort(Element E[], int first, int last, int S)
{
     if (last - first > S) {
          int mid = (first + last)/2;
          mergeSort(E, first, mid, S);
          mergeSort(E, mid + 1, last, S);
          merge(E, first, mid, last);
     } else {
          insertionSort(E, first, last);
     }
}
```

Implement the original version of Mergesort (as learned in lecture) and the above modified version of Mergesort, using a programming language of your choice. Compare their performances in the numbers of key comparisons and CPU times. The value of **S** is to be empirically decided by trial and error.

For simplicity, suppose the input elements are integers and the goal is to sort input numbers in ascending order. Your report and presentation should include a description and results of the following steps:

1. Generate your own input data sets of various sizes, say, ranging from 1000 to 1,000,000 random integers.

2. Count the numbers of key comparisons and CPU times taken by your program on the data sets. Describe how running times increases with input sizes when running the two versions of Mergesort algorithm.

3. Carry out experiments to study how the different values of **S** will affect the performance of the modified algorithm.

Example Class 4 (Week 11 – Week 12)

Please choose **one** of the following two projects (Project 4A or 4B):

Project 4A: Statistical and topological analysis of social networks

Write a program to analyze the statistical properties of a social network. For a node in a network (or graph), its *degree* is the number of other nodes connected with it. In this project, we will calculate and analyze the distribution of node degrees for a graph *G* that represents a social network, to see if *G* represents a "scale-free" network.

The following definition of "scale-free network" is from Wikipedia, and it is used in this project:

"A scale-free network is a network whose degree distribution follows a power law, at least asymptotically. That is, the fraction $P(k)$ of nodes in the network having k connections to other nodes goes for large values of k as $P(k) \propto k^{-\gamma}$, where γ is a parameter whose value is typically in the range $2 < \gamma < 3$, although occasionally it may lie outside these bounds."

The project can be divided into the following steps:

1. **Retrieve a real social network dataset of big size:** You may use APIs of some publicly available social networks. For example:
    a. Facebook: https://developers.facebook.com/docs/reference/api/
    b. Twitter: https://dev.twitter.com/overview/api

    Also, you can choose a social network from SNAP:
    https://snap.stanford.edu/data/

    A recommended network size is around 5000 nodes. But you may also choose other network sizes. Use either an adjacency matrix or an array of adjacency lists to represent the network.

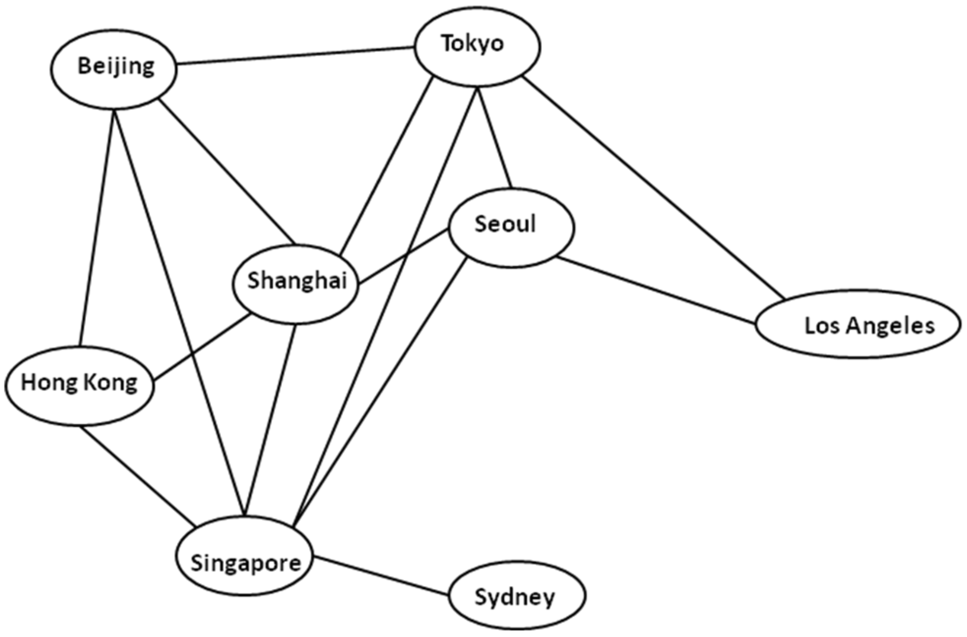
2. **Calculate node degrees**: Design and implement an algorithm to calculate the degrees of all the nodes in the network.

3. **Study the distribution of node degrees:** Plot a histogram for the node degrees in your network, where the variable on x-axis is k (i.e. the node degree), and the variable on y-axis is $P(k)$ (i.e. the fraction of nodes having degree equal to k). Do curve fitting (e.g. using Microsoft Excel or R) of the power-law function to the data, choosing a value for γ that fits the data well.

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

Project 4B: Application of BFS to flight scheduling

Construct an undirected graph to represent non-stop airline flights between cities in the world (a hypothetical graph for Asia Pacific is given below). In your graph, please add more cities and flights to make it more realistic. It should contain at least one pair of cities, between which there is no non-stop flight, but there is a route (or path) between them.



Implement the Breadth-first search (BFS) algorithm to find a route between two cities with the minimum number of stops. That is, when user inputs the names of two cities, your program should return **one** route with the smallest possible number of stop(s). If a non-stop flight is available, it will just return the departure city and arrival city.

In your report and presentation, please include a description and results of the following steps:

1. Measure the CPU times of your program on graphs of different sizes, and analyze how the running times depend on the numbers of cities and non-stop flights.

2. Explain whether Depth-first search (DFS) algorithm can be used in place of BFS, why or why not.

Note for Example class 4
Any student who has not done any presentation for his/her group must present.