CSE 5311: PROGRAMMING PROJECT TOPICS

Several programming projects are briefly described below. I am willing to consider other programming proposals that you may have on your own, provided they have a very strong relation to the contents of this course. If you wish, send me the proposal modelled according to the descriptions below. There are totally **8** projects described below. The first four involves simple implementation of concepts/ data structures described in the class while the latter four asks you to solve some practical problem using algorithms/data structures that you learned in the class.

Logistics: The project will be done in teams of 2-4 students. Exceptions will be granted as appropriate. If you plan to use the same team as Anki project, that is perfectly fine. In order to ensure some variety of projects, the project assignment will be done as follows. Update the list of **3 (three)** projects ordered based on your preference in the Google document for Project signup sheet (where you entered your Anki project preferences). If you have any trouble, send the list to instructor/GTA. As an example, your preference might look like - P_1, P_4, P_2 (where P_i means the title in the document below of Project #i). This list means that your first preference is project P_1 followed by P_4 and P_2. Please finalize your project team by **October 23, 2014** and inform the instructor/GTA. The final allocation will be announced on **October 24**. We will try to assign projects based on your preference as much as possible.

Final Project Demonstration: The project presentation will be scheduled during the **first week of December**. During the presentation, you will demonstrate your project to the instructor/GTA in which you show the various features of your system, such as its correctness, efficiency, etc. You should be prepared to answer detailed questions on the system design and implementation during this demo. We will also examine your code to check for code quality, code documentation, etc. *We might also use some external data test files to verify the correctness of your algorithm.* Plagiarism will be treated very seriously. We will use some popular academic tools to check for **plagiarism** that can look past simple variable name changes, moving code blocks etc :). Additional details will be communicated as necessary.

Communication: We have created a separate folder in Piazza for the final project. Post your queries/clarification there.

General Advice: While we would be thrilled to have some dazzling UI or flashy animations, our primary aim is to make you understand the various design choices that go into the algorithms and the necessary trade offs. The scalability of your algorithm is very essential make sure you test your algorithm with (hundreds of) thousands of elements. Your project will be evaluated based on correctness, efficiency, scalability and

most importantly, the experimental analysis. In both the project demonstration and the project report, we are very interested in your analysis of the experimental results. While we might give some initial parameters to evaluate, it is incumbent upon you to thoroughly evaluate the algorithms and present them in the report/demonstration.

Deliverables: The following are the expected project deliverables that must be sent to the instructor/GTA as "CSE5311-Project" in the subject.

1. A completed project report for the entire team which contains details about your project, such as main data structures, main components of the algorithm, design of the user-interface for input/output (if applicable), experimental results, e.g. charts of running time versus input size, etc.

2. You should also turn in your code and associated documentation (e.g. README files) so that everything can be backed up for future reference.

3. An individual report from each of the team members specifying (a) what are their responsibilities in the project and (b) what is the break-up in terms of effort by each student. Notice that this is an individual report.

During project demonstration, we might ask you to test the algorithm on some external file provided by us. The input and output format will be specified clearly in the project description.

Project Topics

1 Order Statistics and Sorting

In this project, you will evaluate various algorithms and strategies for the related problems of order statistics and sorting. Your project must be able to perform the following tasks:

1. Given an array and a number k, return the k-th smallest element
2. Given an array and a number k, return the top-k elements
3. Given an array, sort it in ascending order.

Order Statistics: You will implement and evaluate the following algorithms for computing order statistics: (remember that k can be arbitrary and not necessarily the median). Please refer Chapter 9 of CLRS for additional details.

1. Order statistics in worst case linear time (via median of median algorithm)
2. Order statistics in expected linear time

Sorting: You will implement and evaluate the following algorithms for sorting.

1. **Elementary Sorting Algorithms:** Bubble, Selection and Insertion sort.
2. **Heap sort:** You can use existing implementation of heap.
3. **Quicksort** and the following variants:
   (a) *Classical* quick sort that always takes the first element as pivot
   (b) *Randomized* quick sort that takes the pivot randomly
   (c) *Median of 3 heuristic*: Instead of taking a single pivot randomly, this heuristic picks 3 elements randomly. Then choose the median of these three elements as the pivot.
   (d) *Quicksort with insertion sort*: This variant is based on the observation that insertion sort is extremely fast for small arrays. So given a number l, your algorithm should use quick sort until the sub-array is of size l or less when insertion sort is used instead. Notice that this will give a runtime of $O(nl + n \log \frac{n}{l})$. How should l be chosen in practice?
   (e) Implementations of Quicksort in popular languages Java/C++/C#) etc use a combination of the heuristics above. Compare your previous heuristics with a real world implementation of quick sort (For eg, Java's implementation can be found in the references - specifically look for the sort1 function).

Input/Output format: For simplicity, you can assume that all the elements are float. For order statistics, the first line of your input file will contain k and n,the total number of elements in the array separated by a space. Then the actual elements in the array itself will be provided one per line. For sorting, the first line will provide the array size followed by array content one per line. The output is a file with the

correct answer (k-th element, top-k elements, sorted array etc in a one element per line format).

Evaluation: While this project might look "easy", it also requires extensive analysis of the algorithm behavior. Your program should be able to handle array sizes in the range of millions. Some "suggested" (but non exhaustive) evaluation include

1. Behavior of median of median algorithm for groups of 3,5 and 7.
2. Comparison of deterministic and probabilistic algorithm for order statistics.
3. Comparison of elementary sorting algorithms among themselves and with faster algorithms (heap/quick sort)
4. Comparison of running time of heap sort and quick sort which is faster?
5. Comparison of various quick sort heuristics wrt to running time, stack (recursion) depth etc
6. How does the distribution of input data (uniform, normal etc) affect the algorithm?

References:
1. Java's implementation of Quick sort (see sort1 function) `http://grepcode.com/file/repository.grepcode.com/java/root/jdk/openjdk/6-b14/java/util/Arrays.java` .

2 Large Number and Matrix Multiplication

In this project, you will implement algorithms to multiply really large numbers and matrices. Note that Gauss and Strassen's algorithm will be covered at the end of the class. If you plan to implement this project, do read the relevant chapters in textbook.

Large Number Multiplication: In this task, you will implement and evaluate various algorithms for multiplying very large numbers. The algorithms are the traditional long multiplication (the one learned in grade school!), Gauss's algorithm (the one covered in class) and Karatsuba's algorithm (see reference below). You can assume that all numbers are provided in base 10 and are always integers (albeit with lot of digits) and the number of digits is a power of 2.

Large Matrix Multiplication: In this task, you will implement algorithms for large matrix multiplication. You can assume that the matrices are square, the elements are integers and the dimensions are a power of 2. Evaluate the traditional $O(n^3)$ algorithm and Strassen's algorithm.

Input/Output format: The input file for large number multiplication will contain two lines one for each number. For matrix multiplication, the matrix will be specified in the "sparse" format. The first line will provide the dimension while the subsequent lines will be of the format (row, column, value). For eg, an identity matrix

of size 3 will be specified as 3 \n (1,1,1) \n (2,2,1) \n (3,3,1).

Evaluation: This project is "easy" wrt to implementation per se. However getting the internal data structures right is a bit tricky! Your project should be able to multiply two numbers with up to 10 thousand digits each and matrices of dimension up to 2048. This means that you can no longer use any of the available data types. Think carefully about how you represent large numbers as this will have a significant impact on the running time. Evaluate the running time of the various algorithms.

References:

1. Gauss' (also covered in class) and Karatsuba's algorithm for multiplication - `http://en.wikipedia.org/wiki/Multiplication_algorithm#Fast_multiplication_algorithms_for_large_inputs`
2. Strassen's algorithm : Chapter 4 of CLRS.

3 Shortest Path Algorithms

In this project, you will implement various algorithms for Shortest path over weighted graphs. Specifically, you will implement two single source shortest path algorithms (Dijkstra and Bellman-Ford) and two multi source shortest path (Floyd-Warshall and Johnson). We will not be covering Johnson's algorithm in the class - So refer the textbook for details. Once you have implemented Dijkstra and Bellman-Ford algorithms, implementing Johnson's algorithm is actually quite easy. You could use external library implementations for the required data structures such as heaps.

Input/Output format: The input graph will be specified through adjacency matrix. The matrix itself will be specified in the "sparse" format. The first line will provide the number of nodes, the second line contains the number of edges while the subsequent lines will be of the format (row, column, weight). For eg, an identity matrix of size 3 will be specified as 3 \n (1,1,1) \n (2,2,1) \n (3,3,1). You can assume that weights are integers that could be negative.

Evaluation: This is another "easy" project wrt implementation. However, in order to get good credit, you would have to do extensive experiments. Your algorithm must also run on very large graphs with thousands of nodes. Some "suggested" (but non exhaustive) evaluation include

1. Comparison of performance of single source shortest path algorithms (Dijkstra and Bellman-Ford)
2. Comparison of multiple source shortest path algorithms (Floyd-Warshall and Johnson)
3. Compute all-pair shortest paths by running single source shortest path algorithms (D and B-F) on all nodes. Compare the performance with directly invoking all pair shortest path algorithms (F-W and J)
4. Compare the performance on dense and sparse graphs

References:

1. The algorithm descriptions can be found in Chapters 24 and 25 of CLRS
2. A sample input file can be found in `http://algs4.cs.princeton.edu/44sp/NYC.txt` . Our test files will be larger.

4 Data Structures for Dynamic Set ADT

In this project, you will implement four popular data structures for representing dynamic set ADT. Specifically, you will implement BST, RBT, Min-Heap and Hash Table. You cannot use any external libraries as part of your implementation. For each data structure, you will implement the following operations: Insert, Delete, Search and Minimum For Hash table, implement a simple modular hash function $h(k) = k \mod m$ and use linear probing for collision resolution.

Input/Output format: The first line of the input file will contain the space budget for hash table. For other data structures, this can be safely ignored. This will be followed by a list of tuples of the format (opcode, value). Valid opcodes include I (insert), D (delete), S (search), M (minimum) and P (print). You can also assume that value will be an integer and all elements are distinct. When delete opcode is invoked on the heap, you need to delete the root node (i.e. ignore the value). Ensure that your code works on edge cases (e.g. deleting or searching for an element that is not in the data structure). When the print opcode is issued, output the content of ADT through pre-order traversal (for BST, RBT and Heap) - one element per line. For Hash table, print the content of each bucket one line at a time. If the bucket is empty, output the special string None.

Evaluation: This is another "easy" project wrt implementation. However, in order to get good credit, you would have to do extensive experiments. Your algorithm must be able to handle large sets. Some "suggested" (but non exhaustive) evaluation include

1. Comparison of performance of various data structures for the 4 operations
2. Comparison of performance of BST and RBT for same set of inputs
3. Performance on worst case examples - sorted elements, elements that will cause large collisions etc

5 Task Scheduler using Red-Black Trees

Red-black trees are very powerful data structures and find numerous applications. For eg, *TreeMap* in Java/C++ STL is usually implemented using it. In this project, you will implement another high profile application - a simplified version of Linux's scheduling algorithm - **Completely Fair Scheduler(CFS)**.

Here is the intuition behind the scheduler. The primary objective is to ensure that each task that is currently active has access to CPU as fairly as possible. So we associate an unfairness score with each task. The scheduler maintains the list of active tasks as a red-black tree and the nodes are ordered in descending order of unfairness hence the task that was treated most unfairly is the left most node. The scheduler

runs a task till it is no longer the most unfairly treated. Please see the references for additional details. Of course, you have to insert a node for a task when it arrives and delete it once it is completed.

Input/Output format: Your input will be a single file of the following format. The first line contains the total number of tasks and the number of time periods to run the algorithm separated by a space. It is followed by a list of tasks. Each task will be triple <task id, start time, number of seconds it takes to complete>. For eg <20, 10, 100>means that task-20 arrives at 10th time unit of simulation and requires 100 seconds in the CPU to complete (not necessarily consecutively).

The output will contain two things:

1. Given some time unit (say time unit 100), the snapshot of the red-black tree including the tasks, their color and their unfairness values. Do an in-order traversal so that the tasks are ordered based on their unfairness.

2. The list of tasks that ran during the time period of the simulation. For eg, suppose the simulation ran for 5 time units <1,2,3,2,2>means that task 1 was run by the scheduler at the start. Then task-2 ran for 1 time-unit followed by task-3 and then task-2 ran again for 2 consecutive time periods.

Evaluation:

1. Compare this implementation with that of a heap (you can use a priority queue from existing library).

2. How fair is the scheduler actually?

3. Does this scheduler maximizes throughput?

References: The algorithm for this scheduler is quite simple once you get the red black tree working. For further details see:

1. `http://cs.unm.edu/~eschulte/classes/cs587/data/bfs-v-cfs_groves-knockel-schulte.pdf`

2. `http://en.wikipedia.org/wiki/Completely_Fair_Scheduler` and the references therein

3. `http://www.ibm.com/developerworks/library/l-completely-fair-scheduler/`

6 MST and TSP for Metric Graphs with MST Heuristic

In this project you will implement two algorithms for MST and evaluate how the choice of the internal data structure impacts the running time. Then you will use the MST algorithm to provide an approximate solution to the famous Traveling Salesman problem.

Minimum Spanning Trees: In this task, you will implement the following algorithms:

1. **Kruskal's algorithm:** You will implement Kruskal's algorithm and evaluate the following variants that differ in how they check if two vertices are in two different trees.

   (a) The naive method where you use DFS to check if two vertices are in same or different trees.

   (b) Union-find without path compression and union-by-rank heuristic (you can use some existing implementation of Union-Find)

   (c) Union-find with path compression and use union-by-rank heuristic (you can use some existing implementation of Union-Find)

2. **Prim's algorithm** - Learn about Prim's algorithm from CLRS. You will implement the algorithm and evaluate the following variants that differ in how they choose the next minimum weight edge.

   (a) Storing edges as an unsorted array

   (b) Storing edges as a sorted array

   (c) Storing edges as a binary min-heap (you can use an external library for heaps)

Traveling Salesman Problem: Traveling salesman problem is a very famous and hard problem that we will briefly explore later in the course. For the purpose of this project, you can get a primer from Chapters 34 and 35 of CLRS. We will look at using approximation algorithms based on MST for a specific type of graph *metric graph* (where the distances obey triangle inequality). Assume that the input graph is guaranteed to be metric.

You can choose to implement either the *2-approximation* algorithm (defined in Chapter 35 of CLRS) or the *1.5-approximation* algorithm (aka Christofides algorithm). Compare it with the optimal answer. You can use some external library to find the optimal solution to the TSP problem.

Input/Output format: The input graph is a *complete* graph where each node is connected to each other node. It is specified as follows. The first line contains the number of nodes. The subsequent lines contains the nodes as triple <node id, x,y>. i.e. the input is a geometric graph where the nodes are points in the 2-d plane. The distance between any two points is the euclidean distance between them. Notice that this graph is also metric.

Evaluation: Here are some simple evaluations to start with.

1. How does the various data structures impact the performance of individual MST algorithms

2. How does Prim's and Kruskal's algorithm compare in performance?

3. How good are the answers provided by the approximation algorithm with that of optimal answers provided by external solver?

References:

1. TSP : Chapter 34 of CLRS
2. MST based Approximation algorithm for TSP: Chapter 35 of CLRS.
3. Christofides algorithm - `http://en.wikipedia.org/wiki/Christofides_algorithm` or `http://www.cs.cornell.edu/courses/cs681/2007fa/Handouts/christofides.pdf` .

7 Network Flow: Resource Allocation and Project Scheduling

In this project showcases the power of network flow algorithms by allowing you to solve three different yet related problems. Evaluate each of the problem for both Ford-Fulkerson and Edmond-Karp algorithms.

7.1 Simple Resource Allocation via Maximum Bipartite Matching

You are given a set of n tasks and m resources (or people). Each task can be completed by a single person. For each task, the subset of workers qualified to perform it are also provided to you. For each person, you are also provided with a maximum number of projects he/she can work on. Your objective is to find an assignment of tasks to people such that as many jobs are completed and no person is overloaded. Model this as a maximum bipartite graph matching problem and solve it.

7.2 Resource Allocation with Constraints

Here is a slightly more complex version of the problem. The setting is similar to above. For each person we are provided with a bound [a,b] which means that the person has to work atleast 'a' tasks and atmost 'b' tasks. Similarly, each task is also provided with a bound [c,d] which means it requires atleast 'c' workers and atmost 'd' workers. We also know which persons are eligible to work on which projects. Your objective is to find an assignment such that as many tasks are completed while not violating the resource constraints.

7.3 Task Scheduling with Profit Constraints

Here is yet another variant. Let us forget about the people and focus only on the tasks. We are provided with a bunch of tasks. Each task t_i has a profit p_i associated with it. If $p_i > 0$ then we make a profit while if $p_i < 0$ we lose money. To make things harder, we are also provided with some constraints within projects. Intuitively, model the problem as a graph where each task is a node. An edge exists between tasks t_i and t_j if t_j is a prerequisite of t_i. In other words, if we want to do t_i , we must also do t_j. Your objective is to identify a subset of tasks such that your profit is maximized and all the inter-dependencies are satisfied.

Input/Output Format: For the first two problems, the input is specified as two files. One for workers and one for tasks. The first line of workers file stores the total

number of workers followed by a triple <workerid, min tasks, max tasks>. Of course for problem 1, min task=max task. The first line of task file specifies the total number of tasks followed by a tuple <task id, min requirement, max requirement, list of qualified worker ids>. For the third project, the input is provided as follows. The first line provides the total number of tasks. The subsequent line describe each task per line as a tuple <task id, task profit, list of projects dependent on>. The last parameter can be empty! The output consists of one line per task (ordered by task id) where you specify the task id followed by the resource allocated. For the final problem, the list of tasks chosen (ordered by id) suffices.

References: You might want to check this notes for some useful tips on modeling the problems `http://courses.engr.illinois.edu/cs473/sp2011/lectures/19_add_notes.pdf` .

8 DeDuplication and Music Identification using Hashing Functions

In this project, you will use the magic of hashing functions to perform two non-trivial applications. The first is near duplicate identification where the objective is to identify items that are very similar to each other. The second is to identify music (or video if you are ambitious) from a small music snippet.

Near Duplicate Identification: In the class, we briefly discussed about how lot of services such as Dropbox use hashing function to identify duplicates so that each file is uploaded only once. However, very few scenarios allow for such exact duplicates. Typically, two files will be near duplicates. To take a web example, there might two documents that have identical content but with different timestamps. Search engines such as Google would like to store only one of them so that the storage cost is minimized.

Please see references for additional details about how to implement the algorithm. You can use any popular hashing function such as SHA-1, SHA-256, MD5 etc.

Music Identification: Many of you must be aware of the music identification tool Shazam. It uses hashing functions to identify a song from a 10 second fingerprint. A key difference from previous problem is that two songs can sound identical to human ear even if their bits are very different. Hence, we have to use a hashing function that "mimics" human ear. This is called as acoustic fingerprinting. Fortunately, there exist a number of open source hashing functions for audios. In this project, you will design an algorithm to identify the music. If you are ambitious, you could extend it to video (ContentId of YouTube works in a similar fashion).

High Level Idea: While the two projects might be in two domains, the underlying approach to solve them is quite similar. You will split your project into two parts. In the first part, you will be given a set of input files (text or audio) and you will convert the content into fingerprints (of Shingles and using appropriate hashing function) and store them in some database (file based such as SQLite or MySQL/PostgreSQL). In the second part, given an input (either a duplicate text file or short audio snippet), you want to identify if it exists in the database and output relevant details. With careful design, most of your code (almost 80% in my case) could be shared across the

two tasks (deduplication and music identification).

Input/Output format: The input to both projects will be two directories. The first directory will contain a set of files that are distinct. In the second directory, there will be additional files that will be used for evaluation. For task 1, the files could potentially be (near) duplicates of some existing file in directory 1. For task 2, the files will short audio files that are potential snippets of files from directory 1 (you can use a tool like Audacity to create audio snippets). Read the files in second directory in sorted order. For both tasks, each line of the output file will contain two information : (filename, duplicatename). Filename is the name of the file in second directory while duplicate name is the file in first directory that it is closest to. If there are multiple duplicates choose the closest one. If there are no duplicates, write as (filename, Distinct).

References:

- Near Duplicate Detection: `http://nlp.stanford.edu/IR-book/pdf/19web.pdf`
- Music Identification: Do a Google search for query like "shazam fingerprint". Here is an old non technical introduction `http://www.slate.com/articles/technology/technology/2009/10/that_tune_named.html`.
- My favorite database of acoustic fingerprints for testing is `https://acoustid.org/database` . There are many others available in net.
- There are a number of open source acoustic fingerprinting services. You can use any one of AcoustID (such as Chromaprint), EchoPrint, MusicBrainz etc.