CSE 5311: PROGRAMMING PROJECT TOPICS

Several programming projects are briefly described below. However, in exceptional circumstances we are 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.

Logistics: The project will be done in teams of 3-4 students.

Final Project Demonstration: The project presentation will be scheduled during the **first week of May**. During the presentation, you will demonstrate your project to the 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 also 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 Blackboard 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 your 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.

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. **Heap sort:** You would have to implement heap yourself.
2. **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 the order statistics, 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 running time of heap sort and quick sort which is faster?
4. Comparison of various quick sort heuristics wrt to running time, stack (recursion) depth etc
5. 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.

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.

Large Matrix Multiplication with Large Numbers: Combine your previous implementations to allow multiplication of two large matrices with large numbers as its elements!

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 upto a million digits each and matrices of dimension of upto 10,000. 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. I would also urge you to consider the sparse representation of matrix which can save significant amount of space for matrices. 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 String Matching: Plagiarism detection

Plagiarism is a serious problem in research. In this project, you will implement a very simple plagiarism detector. Your input will be a corpus of existing documents and a potentially plagiarized document. Your output will be the set of documents from which the document was plagiarized from. The definition of plagiarism and threshold are left to your discretion. Implement the following basic detectors and compare their performance :

1. **LCSS:** Run the LCSS algorithm for each paragraph from the test file and existing corpora.
2. **Naive String Search:** Implement the nave search algorithm that treat each sentence in the test file as a potential pattern and searches the pattern in all existing documents.
3. **KMP:** Given a test file, treat each sentence in the test file as a potential pattern. Search for the pattern in the existing documents and find the matches.
4. **Boyer-Moore algorithm:** Perform string matching using the Boyer-Moore algorithm that is quite similar in philosophy to KMP.

Input/Output format: The input will be a directory containing a bunch of existing documents (for eg, submissions from all students) and a single file that has to be evaluated. Your output will be the set of documents and sentences from which the document was plagiarized from. For eg, sentence x from input file is same as sentence y from file z and so on.

Evaluation: Potential evaluation parameters include the running time of various algorithms for matching a single pattern.

References:

1. Naive search, KMP: Chapter 32 of CLRS.

2. LCSS: Chapter 15 of CLRS.

3. Boyer-Moore : See Wikipedia page and references therein.

4 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/`

5 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. Think carefully about how you will implement union-find! A bad implementation renders the speedup moot.
   (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 (use union-by-rank heuristic)
   (c) Union-find with path compression (use union-by-rank heuristic)
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` .

6 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.

6.1 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.

6.2 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.

6.3 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` .

7 Convex Hulls, Skylines and Triangulations

In this project you will implement different algorithms for convex hulls and then use them for triangulation of geometric points.

Convex Hull: Implement the following convex hull algorithms.

1. Divide and Conquer
2. Graham Scan
3. Jarvis March.

Skylines: Given a set of points $p_1, p_2, \ldots, p_N$, the skyline is a subset of points (lets call it P) such that each point in P is not dominated by any other point in the dataset. The definition of domination is simple: p_1 dominates p_2 if p_1 is is not worse than p_2 in all dimensions and p_1 is better than p_2 in at least one dimension. For e.g., <10,20>dominates <8,2>but not <15,8>. How can you use your convex hull algorithms to find the set of skyline points? (Note: not all points in convex hull are skyline!).

Onion Convex Hulls: In this task, you will use some of the algorithms above to implement onion convex hulls. For a given set of points, you can create a set of

concentric convex hulls. Let S be the set of original points. Now find the convex hull using the algorithms you implemented in the previous task. Let S' be the set of points from S that are not in the convex hull. Find the convex hull for S' . Repeat the process till no points remain within the convex hull.

Triangulation: Implement Onion triangulations for any two consecutive convex hull using Rotating calipers method. For details refer `http://cgm.cs.mcgill.ca/~orm/ontri.html`.

Input/Output format: The first line of the input file contains the total number of points. The subsequent lines contain a triple <point id, x, y>where id is a unique number while x and y are its co-ordinates in 2D plane. The output for convex hulls is a list of points on the hull ordered by their id.F The output for skyline is similar just output the skyline points ordered by their id. For triangulation, output the list of triangles one per line. The 3 points involved in the triangle are ordered based on their ids. For eg, <10, 20, 30>is a valid order but <20,10,30>is not. The list of triangles must be ordered based on their points. I.e sort them based on first point in triangle, followed by second point and third point. For eg, <10,20,80>, <20,30,40>and <20,50,70>is a valid order.

Evaluation:

1. Compare the performance of various convex hull algorithms
2. Compare the naive algorithm for Skyline with the one based on convex hull.