CSE 101 Homework 1

Background (Order and Recurrence Relations), correctness proofs, time analysis, and speeding up algorithms with restructuring, preprocessing and data Structures.
Due April 29
100 points total = 10%

Solve each problem. For algorithm problems, if the problem only specifies that you need to give a proof of correctness, then no time analysis is required. If it specifies that you need to give an efficient implementation, then you do not need to give a correctness proof for the basic strategy (just explain why your version actually carries out the strategy). If it says to do both, or doesn’t specify what parts you need, you need to give both a proof of correctness and time analysis.

**Order (10 points)** Is it always the case that \( f(2n) \in O(f(n)) \)? If so, give a proof; if not, give a counter-example.

**Triangles (10 points)** Let \( G \) be an undirected graph with nodes \( v_1 \ldots v_n \). The *adjacency matrix* representation for \( G \) is the \( n \times n \) matrix \( M \) given by: \( M_{i,j} = 1 \) if there is an edge from \( v_i \) to \( v_j \), and \( M_{i,j} = 0 \). The *adjacency list* representation is an array, which for each node, contains the head of a list of (pointers to) all of the node’s neighbors in the graph.

A triangle in the graph is a set of three nodes that are all adjacent. The problem is, given a graph, does it contain a triangle? On the calibration homework, you were asked to give and analyze an \( O(nm + n^2) \) time algorithm for this problem in the adjacency matrix representation, where \( n \) is the number of nodes and \( m \) the number of edges. Can you give an equally fast algorithm if the input is presented in the adjacency list representation? You can use the existence of the algorithm for adjacency matrix representation as a subroutine, without needing details or proof. (Just cite the answer key.)

**Proof of correctness - 10 pts** Below is an algorithm, that, given two integers, \( x,y \), computes \( y^x \). Only integer operations are performed on \( y \), but it is important that \( x \) be represented in binary, as an array of bits \( x = x[n-1] \ldots x[0] \). After the algorithm, there is a proof of correctness, with some parts missing. Fill in the missing sections in the proof to get a complete proof of correctness. You do not need to give a time analysis.

\[
\text{Exp}(y: \text{integer}; x=x[n-1..0]: \text{array of bits (binary integer))}: \text{integer}
\]

1. \( p \leftarrow 1. \)
2. FOR \( I = 1 \) TO \( n \) do:
3. \( p \leftarrow p^2; \)
4. IF $x[n - I] = 1$ THEN $p \leftarrow p * y$;
5. Return $p$.

Proof of correctness: For $t \geq 1$, let $x_t$ be the integer represented by the first $t$ most significant bits of $x$, i.e., in binary, $x_t$ is . We'll also define $x_0 = 0$. Let $p_t$ be the value of $p$ after the loop when $I = t$, and $p_0$ be its initial value, i.e., $p_0 =$. We'll prove correctness via the following loop invariant: $p_t = y^{x_t}$. Then the correctness of the algorithm will follow, from the case $t =$, since then we output $p_n$, which by the invariant, is equal to . Since $x_n =$, this means the output is , as desired.

We'll prove this invariant by on . First we prove the , that the invariant is true when $t = 0$, i.e., that . This is true because $p_0 =$ and $x_0 =$, so $p_0 = y^0 =$ as we wanted to show.

Next we'll prove the . Assume that the invariant holds for $t$, i.e., that . We want to show that it is still true after the next iteration, i.e., that 

There are two cases: If $x[n - (t+1)] = 0$, then in binary $x_{t+1}$ is $x_t$ followed by a , so $x_{t+1} =$. In this case, we do not perform the multiplication in line , so $p_{t+1} =$. By the loop invariant, $p_t =$. Substituting, $p_{t+1} = (y^{x_t})^2 = y^{x_t+1}$, which is what we wanted to prove.

In the other case, if $x[n - (t+1)] =$, then in binary $x_{t+1}$ is $x_t$ followed by a , so $x_{t+1} =$. In this case, we the multiplication in line , so $p_{t+1} =$. By the loop invariant, $p_t =$. Substituting, $p_{t+1} = y * (y^{x_t})^2 = y^{x_t+1}$, which is what we wanted to prove.

Thus, in either case, we have shown that if the invariant holds for , then it also holds for . Thus, by induction, the invariant holds for all , and we have seen that this implies that the algorithm outputs the correct value .

**Business plan (15 points):** Consider the following problem. You are designing the business plan for a start-up company. You have identified $n$ possible projects for your company, and for each project, $Project_i$, you have calculated the minimum capital you need to start the project, $RC_i > 0$, and the profit you will get after you complete the project, $Profit_i > 0$. (Note that profit is revenue - expenditures, so it is the amount you get in addition to the amount of capital you use.) You also know your initial capital $C_0 > 0$. You want to perform at most $k$, $1 \leq k \leq n$, projects before the IPO and want to maximize your total capital at the IPO. Your company cannot perform the same project twice.
In other words, you want to pick a list of up to \( k \) distinct Projects, \( \text{Project}_1, ..., \text{Project}_{k'} \) with \( k' \leq k \). Your accumulated capital after project \( i \) will be \( C_i = C_0 + \sum_{h=1}^{i} \text{Profit}_h \). The sequence must satisfy the constraint that you have sufficient capital to start \( \text{Project}_{i+1} \) after completing the first \( j \) jobs, i.e., \( C_j \geq RC_{i+1} \) for each \( j = 0..k' - 1 \). You want to maximize the final amount of capital, \( C_{k'} \).

AbstractBusinessPlan[Project[1..n]: array of pairs of positive reals, [RC, Profit], C_0: real, K:integer]: List of Projects.

Let \( S \) represent the set of projects that we have enough accumulated capital to start, and haven’t yet performed.

1. Initialize \( \text{List} \) as an empty list of projects.
2. Initialize \( \text{Accumulated} \leftarrow C_0 \).
3. Initialize \( S \) to be the set of all projects \( I \) with \( \text{Project}[I].RC \leq \text{Accumulated} \).
4. Iterate \( k \) times, or until \( S \) is empty.
5. Find the largest profit project in \( S \), \( \text{Project}[J] \).
6. Append \( \text{Project}[J] \) to \( \text{List} \);
7. Delete \( \text{Project}[J] \) from \( S \).
8. Add all projects \( \text{Project}[K] \) with \( \text{Accumulated} < \text{Project}[K].RC \leq \text{Accumulated} + \text{Project}[J].Profit \) to \( S \).
9. \( \text{Accumulated} \leftarrow \text{Accumulated} + \text{Project}[J].Profit \)
10. Return \( \text{List} \)

Describe an efficient algorithm that carries out this strategy. Your description should mention which data structures you use, and any pre-processing steps. Give a time analysis.

Example:

- Project 1: Design the CSE home page; \( RC_1 = 10K, Profit_1 = 30K \).
- Project 2: Write an automatic homework grader; \( RC_2 = 50K, Profit_2 = 100K \).
- Project 3: Automate the STOC conference submission process; \( RC_3 = 40K, Profit_3 = 50K \).
- Project 4: Design a scheduling algorithm for the UCSD elevators; \( RC_4 = 100K, Profit_4 = 200K \).
- Project 5: Write a baseball-card trading program; \( RC_5 = 90K, Profit_5 = 500K \).
- Project 6: Design secure, on-line GREs; \( RC_6 = 650K, Profit_6 = 1,000K \).
• Project 7: Embedded GPS chips for pencils: \(RC_7 = 2000K, \text{Profit}_7 = 10,000K.\)

You can complete up to \(k=4\) projects, and have an initial capital of \(C_0=50K.\)

\(S\) starts at 1, 2, 3. The max profit project in \(S\) is project 2, making accumulated capital 50+100=150. That adds Project 4 and 5, making \(S = 1, 3, 4, 5.\) Of these, the max profit is project 5, raising capital to 650, just enough to do project 6. Now \(S = 1, 3, 4, 6,\) and project 6 is the largest profit, raising capital to 1650. We can’t do any new projects, so our last project is the best of 1, 2, 4, project 4. Thus, we do projects 2, 5, 6, and 4, and go IPO with capital 1850. Pencils will have to wait.

**DAG:** **(15 points)** A directed graph has edges going from nodes to other nodes, and we write \(x \to y\) if there is an edge from \(x\) to \(y.\) A directed cycle is a list of edges \(x_1 \to x_2 \to x_3 \ldots \to x_k \to x_1.\) A DAG (Directed Acyclic Graph) is a directed graph with no directed cycles. Design and analyze an efficient algorithm that determines if a directed graph (given in adjacency list format, where there is an array \(N(x)\) of lists of nodes with edges to them from each starting node \(x\)) is a DAG. A good algorithm has time approximately \(O(n + m)\) where the graph has \(n\) nodes and \(m\) edges.

**Merge lists (20 points)** You are given an array of \(k\) sorted, non-empty lists, \(L[1..k],\) where each \(L[I] = a[I, 1] < a[I, 2] < \ldots < a[I, n].\) Let \(n\) be the total sizes of all the lists (so in particular, \(n > k\)). Give an \(O(n \log k)\) time algorithm that returns a sorted list containing exactly the elements in the union of the \(k\) lists.

**Implementation** **20 points** Implement bubble-sort and heap-sort. You can use heaps from a standard library to implement heap-sort. Plot their performance on random arrays of \(n\) integers with values between 1 and \(n,\) for \(n = 2^6, 2^8, 2^{10}, 2^{12}, 2^{14}, 2^{16}.\) Plot their performance on a log-log scale. Is heap-sort always better than bubble-sort? Why or why not?