Homework 4

ECE 358 Foundations of Computing
Fall Semester, 2023

Due: Friday, November 24, 5:00pm

- All page numbers are from the 2022 edition of Cormen, Leiserson, Rivest and Stein.
- For each algorithm you asked to design you should give a detailed *description* of the idea, proof of algorithm correctness, termination, analysis of time and space complexity. If not, your answer will be incomplete and you will miss credit. You are allowed to refer to pages in the textbook.
- Do not write C code! When asked to describe an algorithm give analytical pseudocode.
- **Read the handout with the instructions on how to submit your Homework online. Failure to adhere to those instructions may disqualify you and you may receive a mark of zero on your HW.**
- Write *clearly*, if we cannot understand what you write you may not get credit for the question. Be as formal as possible in your answers. Don't forget to include your name(s) and student number(s) on the front page!
- No Junk Clause: For any question, if you don't know the answer, you may write "I DON'T KNOW" in order to receive 20% of the marks.

---

1. **[Minimum Spanning Trees, 10+10 points]**

   Consider a connected, undirected graph $G = (V, E)$ with positive edge weights $w_G(e) > 0$. Consider another graph $H = (V, E)$ with the same vertex and edge sets, but $w_H(e) = w_G(e)+1$ for every edge $e \in E$.

   (a) Do G and H share the same set of minimum spanning trees? Prove that they do or provide a counter-example demonstrating that they do not.

   (b) Do G and H share the same shortest paths between all pairs of vertices? Prove that they do or provide a counter-example demonstrating that they do not.

2. **[Shortest Paths, 15 points]**

   North America and Europe each have separate computer networks, represented by strongly connected directed graphs where vertices are computers on the network, and edges are connections between the nodes (note that the graph is directed, so the connections are not bidirectional unless there is an edge going in either direction). The weights of each edge is the time it takes a packet to travel over the connection. The combination of their networks is represented by $G = (V, E)$.

   A new transatlantic link, $e_{\text{transatlantic}} = (v_{\text{NA1}}, v_{\text{E1}})$ connects the North American and European networks, the only link between them. Devise an $O((|V|+|E|) \lg |V|)$ algorithm that will enable any query for the length of the shortest path from a North American node to a European node to be answered in $O(1)$ time.

3. **[Graph Algorithms, 10 points]**
   You are a frequent traveler who visits various countries with different currencies. However, you often encounter difficulties in converting currencies accurately. To overcome this challenge, you aim to develop an efficient algorithmic solution.

   You are provided with a set of currency exchange rates, where each rate represents the conversion factor between two currencies. Your goal is to design an algorithm that can quickly handle multiple currency conversion queries.

   Devise an algorithm that takes the exchange rates and conversion queries as input and produces the converted values as output. If a conversion is not possible, the algorithm should indicate it appropriately.

   Input:

   Exchange rates: A collection of currency pairs with their corresponding conversion rates. Conversion queries: An array of queries, each consisting of a source currency and a target currency.

   Output:

   Results: An array of converted values for each query. If a conversion is not possible, the message "conversion not possible" should be provided.

   Example:

   Consider the following scenario:

Exchange rates:	Conversion queries:
USD to EUR: 0.85	(a) USD to GBP
EUR to GBP: 0.88	(b) GBP to EUR
GBP to JPY: 147.25	

   Expected output:
   Query (a): 0.85 * 0.88 = 0.748 (GBP)
   Query (b): 1 / 0.88 = 1.136 (EUR)

4. **[Maximum Flow, 20 points]**

   You are helping a medical consulting firm to set up vacation schedule of doctors in a large hospital. Because of the nature of the hospital, they want to make sure there is at least one doctor covering each vacation day.

During the year, there are k vacation periods (*e.g.*, the week of Christmas, the Easter weekend, *etc.*), each of which consists of multiple contiguous days. Let D_j be the set of days included in the j^{th} vacation period, and let $N = \sum_{i=1}^{k} |D_j|$ be the total number of vacation days in the year. There are n doctors at the hospital, each with a set of vacation days when they are available.

Give a polynomial-time algorithm that takes this information and determines whether or not one doctor can be assigned on each vacation day under the following constraints:

(1) Each doctor should only be assigned to work on the days when they are available.

(2) For a given parameter c, each doctor should be assigned to work at most c vacation days in total.

(3) For each vacation period j, any given doctor should be scheduled at most one of the days in D_j.

Your algorithm *must* work by constructing a flow network. In particular, you must create a flow network with a *source node* (one that only has outward capacity) labeled s and a *sink node* (one that only has inward capacity) t. Your algorithm should return "yes" (indicating that a feasible schedule exists) if and only if the maximum s–t flow is at least N.

5. **[Skewed Heaps, 15+5 points]**

In this problem you are required to develop a data structure similar to that of the leftist heap from Homework 2. In *leftist heaps*, the `Merge` operation preserved the heap ordering and the balance (leftist bias) of the underlying tree. *Skewed heaps* use the same idea for merging heap-ordered trees. `SkewHeapMerge` is performed by merging the rightmost paths of two trees without keeping any explicit balance conditions. This means that there's no need to store the rank of the nodes in the tree. This is similar to self-adjusting trees, since no information is kept or updated.

Good performance of those data structures is guaranteed by a "rebalancing step"—like the splay in self-adjusting trees, only simpler. At each step of the merging along the rightmost paths of the two heaps, we swap *all* of the left and right children of nodes along this path, except for the last one. The modified procedure for merging two skewed heaps looks as follows:

function SkewedHeapMerge(h,h') : heap

if h is empty then **return** h'
else if h' is empty then **return** h

if the root of $h' \preceq$ the root of h then
    exchange h and h' (* h holds smallest root *)

if right(h) = **nil** then
    right(h) := h' (* last node, we don't swap *)
else
    right(h) := SkewedHeapMerge(right(h), h')

swap left and right children of h

return h

The above recursive routine can also be done iteratively. In fact, it can be done more efficiently than the leftist heap `Merge` (by a constant factor), because everything can be done in one pass, while moving down the rightmost path. In the case of leftist heaps, we go down the rightmost path, and then back up to recompute ranks. In leftist heaps, that also requires either a recursive algorithm or pointers from nodes to their parents [1].

Since there is no balance condition, there's no guarantee that these trees will have $\mathcal{O}(\log n)$ worst-case performance for the merge (and hence all of the other operations). But they do have good *amortized performance*.

Here's the intuition for the above: In the previous merge algorithm we only had to swap children when the right one had a larger rank than the left. In this merge algorithm, we always swap children, so we might actually replace a right child of "small" rank with one of "large" rank. But the *next time* we come down this path, we will correct that error, because the right child will be swapped onto the left.

(a) Show that a `SkewedHeapMerge` of two skewed heaps uses amortized time $\mathcal{O}(\log_2 n)$ by the use of the *accounting method*.

(b) Show that `Insert` and `DeleteMin` have the same amortized time bounds.

(*Hint*: Use weights instead of ranks. Define the *weight* of a node to be the number of nodes in the subtree rooted at that node (below that node, including the node itself). Let node x be a *heavy* node if its weight is more than half of the weight of its parent. Otherwise it is a *light* one. What can we say about light and heavy nodes in any binary tree? Look at any root–leaf path in the tree. *How many* light nodes can you encounter in such path? Why? The best case is when the light nodes are also right children of their parents. For the accounting method, every time we swap, we must pay \$1. Moreover, if it happens to swap a heavy child to a right child position, then you must deposit a "penalty" amount.)

[1] Just a note on implementation here. Sometimes we may want to be able to move up a tree as easily as we move down; so every node will also include a pointer to its parent. That means that a node has a pointer to its left child, a pointer to its right child, and a pointer to its parent, increasing the amount of space needed to store trees. To decrease the *space* requirement, you can do this. Look at three nodes, a parent p and its two children r and l. Now p will have a pointer to l, but not to r; l has a pointer to r and r has a pointer back to p. So, all the left children in a tree have pointers to their right siblings (brothers) and to their own left children. All the right children have pointers to their parents and to their right children. If you draw a picture, this should make more sense. Now every node still has two pointers, and you can visit left nodes, right nodes and parent nodes easily.