CS 3510: Design and Analysis of Algorithms
Section A, Spring 2017

Homework 5
Due: Monday 11:55PM, April 17th, 2017

Name: Alan Chiang

Notes:

1. Please don't write a program when you describe an algorithm. You should give an idea of what the algorithm does and then discuss the details of how your algorithm does specific tasks. If you give pseudo-code, you should explain your code.

2. For every algorithm that you are asked to describe, please clearly list the steps of your algorithm and explain its correctness and runtime behavior. We recommend separating the algorithm, correctness argument, and runtime analysis into three paragraphs.

3. You can assume that basic operations on integers which are not described to have a variable length (addition, multiplatication, subtraction, floor, ceiling, absolute value, comparison of less than, greater than, or equals, and anything you can form out of a constant number of these) can be done in constant time. You can also assume that, given i, you can read a_i from a list $a_1, ...a_n$ in constant time.

1. (10 points) *Dijstra's Algorithm*

   In order for Dijkstra's algorithm to work, the input graph G must have positive edge weights. Your friend says "I think we can modify the algorithm to make it work on graphs with negative edge weights by just adding some large value N to the weight of each edge to make them all positive." Explain to your friend why he or she is wrong by giving an example of when this algorithm fails.

   ANSWER: They are wrong because total cost of any/every path is shared by the number of edges between source and destination. Adding a large constant to every edge would throw off the cost-per-edge ratio between longer and shorter paths and so the cheapest path would not be guaranteed to stay the same from the original graph to the adjusted graph.

   Example: Two paths from source to destination, one costing 8 through 4 nodes and the other costing 12 through 2 nodes. If a large constant 10 were added to every edge, then the adjusted cost would be 48 for 4 nodes and 32 for 2 nodes. In this example, the cheapest path in the altered graph does not correspond to the cheapest in the original graph.

2. (10 points) *Fibonacci Heap*

Professor Pinocchio claims that the height of an n-node Fibonacci heap is $O(\log n)$. Show that the professor is mistaken by exhibiting, for any positive integer n, a sequence of Fibonacci-heap operations that creates a Fibonacci heap consisting of just one tree that is a linear chain of n nodes.

ANSWER: Add n individual nodes and meld them as their degrees are equal until a B_{n-1} is created (meld two single nodes into a B_1, repeat, meld two B_1 trees into a B_2, repeat, etc, etc until B_{n-1}). Then choose a child of the root (any child not part of the longest subtree) and decrease-key until it is less than the root. This will cut that subtree from the B_{n-1} and cause the subtree to become a new tree with one child. Now we can simply decrease-key the root of the new tree until it is the min and delete-min. This creates a new tree with one child, which is one node smaller than the previous new tree. Repeat the decrease-key and delete-min operations until the entire new tree has been removed. Then return to the our original B_{n-1} heap (now a B_{n-1} minus the one subtree) and cut another child and that child's subtree by decreasing its key. Repeat the decrease-key and delete-min operations until this subtree is completely deleted. Then we repeat this cut-child-subtree-then-decrease-key-delete-min procedure until the original B_{n-1} binomial heap has only one child (the longest subtree child) left. At that point the original B_{n-1} binomial heap will have become a linked list of length n.

Basically, we insert nodes and meld them until we get a B_{n-1} binomial tree and then remove all the subtrees except for the longest, one node at a time. What's left is a linked list with size n.

3. (10 points) *Amortized runtime analysis.*

Suppose you have a stack of items, where you can push an item onto the stack or pop an item from the stack in constant time. Suppose there is only one operation defined for this stack, called `OP`, which involves applying zero or more pops followed by one push. Also suppose that m such operations have been performed on the stack which was initially empty.

```
OP(k) {
        // requires at least k items on the stack.
        pop k items from stack
        push 1 item onto stack
}
```

(a) What is the *non-amortized* cost of performing these m operations (simple worst-case analysis)? Hint: what is the worst-case running time for a single `OP` operation in the sequence?

(b) What is the *amortized* cost of performing these m operations?

ANSWER:

a. Worst-case runtime of a single `OP` operation is when we first perform many `OP` operations in which we pop 0 items, so the size of the stack grows by one each time. Then we perform one `OP` operation in which we pop the entire stack before pushing one item. If the size of the stack is denoted s, this single `OP` operation would run in $O(s)$ time. Since we are using simple worst-case analysis, we treat all `OP` operations as though they cost this much, and so the non-amortized worst-case runtime would be $O(m * s)$.

b. Using amortized analysis, we can realize that we start with an empty stack, so if a single `OP` operation can remove s items, this means that there must have been at least s prior `OP` operations that removed 0 items. Amortizing the cost allows us to spread the single removal of s items across s `OP` operations, which is to say, that every previous `OP` operation effectively popped 1 and pushed 1 node. The cost of this is amortized $O(1)$ per `OP` operation, so to perform m of them would cost amortized $O(m)$.

4. (10 points) *Minimum Spanning Tree.*

A spanning tree T of an undirected graph G is a tree subgraph which includes all of the vertices of G, with minimum possible number of edges. A minimum spanning tree is a spanning tree whose sum of edge weights is as small as possible. Use this information to solve the following questions:

(a) Find a valid MST in Fig 1.(Your answer should be in terms of a list of edges)

(b) Let there exist an edge weighted graph G with V vertices and E edges. Let the weight of the MST be for this graph be W. Now I add a small amount x to the weight of all the edges. What is the new weight W' of the MST?

ANSWER:

a. Using Kruskal's algorithm gives us (a, b), (d, f), (c, d), (b, c), (d, e) Kruskal's chooses the lowest-cost edge in the entire graph and adds it to the MST. It then chooses the next lowest-cost edge that does not create a cycle in the MST. It repeats this step until the MST is of size $V - 1$ at which point the MST is complete.

b. All MSTs have $V - 1$ edges, so each edge weighs on average $W/(V - 1)$ in a weighted graph. Every edge increases by weight x, so every edge in the MST now weighs $W/(V - 1) + x$. Thus $W' =$(W / (V - 1) + x) * (V-1). This simplifies to $W' = W + (V - 1) * x$.