Algorithms in the Real World (15-853), Spring 2018
Assignment #4 Due: Sunday, March 25, 2018

Complete all problems.

You are not permitted to look at solutions of previous years' assignments. You can work together in groups, but all solutions must be written up individually, and please list your collaborators. If you get information from sources other than the course notes and slides, please cite the information, even if from Wikipedia or a textbook.

For algorithm design problems, you should try to provide pseudocode and a well organized verbal description. Then you should analyze the complexity (work and span) of your algorithms based on the pseudocode. Please make your solutions as clear as possible, and they will be graded by the correctness and clearness. If only some "high-level" concepts are provided, you will get no more than half the credit even if they are correct.

Problem 1: List-Ranking[10 pts]

The *list-ranking problem* is to compute for each node in the list, its distance to the tail of the linked list. Despite seeming inherently sequential at first glance, list-ranking can actually be solved in $O(n \log n)$ work and $O(\log n)$ depth with the following algorithm due to Wylie. P gives for each node in the linked list, i, the id of its parent ($P[i]$), (initially the next node in the list). The tail points to itself ($P[i] = i$).

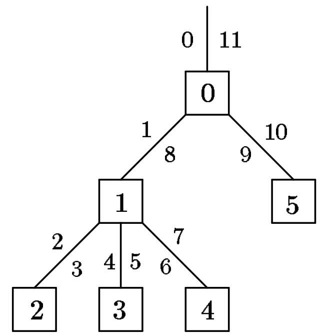
```
list_rank(P) =
  D = array(|P|, lambda i.if (P[i] == i) then 0 else 1);
  for j in [1, ceil(log(|P|))]
    D = array(|P|, lambda i.D[i] + D[P[i]]);
    P = array(|P|, lambda i.P[P[i]]);
  return D;
```

[10 pts] Argue that this algorithm solves the list-ranking problem in $O(n \log n)$ work and $O(\log n)$ depth on the MP-RAM. Can this algorithm be used to compute distances to the head of the list?

In fact, list-ranking can be solved in $O(n)$ work and $O(\log n)$ depth using an idea similar to the random-mate connectivity algorithm described in class. When using list-ranking in the rest of this problem set you may assume that you have a black-box list-ranking algorithm that runs in $O(n)$ work and $O(\log n)$ depth.

Problem 2: Euler Tours[35 pts]

An Euler tour of a rooted tree is a path that starts at the root and goes around the tree visiting each edge twice, once on the way down and once on the way up as in the following figure:



Assume that the Euler tour is represented as a linked list, where each link consists of a pointer to the tree node below the edge (e.g. 1 and 8 both point to tree node 1 above) and a pointer to the next list element. The tree nodes consist of just a value and a pointer to their parent.

1. [10 pts] Using list ranking, describe how to calculate for every node the sum of values stored at and below it. Your solution must run in linear work and $O(\log n)$ span. You can assume list ranking runs in linear work and $O(\log n)$ span and that an Euler tour (in the form described above) is already created.

2. [25 pts] Your second task is to "line break" in parallel. You are given a sequence of integers W representing word lengths in a paragraph, and a line length L, with the condition that all word lengths are at most L. Your job is to describe a parallel algorithm that generates the same output `B` as the following serial code.

```
int s = W[0];  B[0] = 1;
for (i=i; i < n; i++) {
  s += W[i];
  B[i] = 0;
  if (s > L) {
    B[i] = 1;
    s = W[i];
  }
}
```

   In particular this code simply marks in `B` the start of every new line when filling text up to width L (for simplicity, we are ignoring the width of the spaces). Your algorithm must use at most linear work and $O(\log n)$ span. You cannot assume L is a constant (i.e. the work and span must be independent of L). You might find the two problems you already solved useful. Please make your solution as clear as possible.

Problem 3: Parallel SSSP [45pts]

The single-source shortest path (SSSP) problem is to compute the distance between a source node and all other nodes in a graph. Sequentially, SSSP on directed weighted graphs can be solved in $O(m + n \log n)$ using Dijkstra's algorithm with Fibonacci heaps. But what about the parallel complexity of SSSP? about In this problem, you will design and analyze the work and depth of several different parallel SSSP algorithms for *unweighted directed graphs*.

1. [3 pts] What algorithm have we discussed in class that solves this problem? What is the worst-case work and depth of this algorithm?
2. Analyze the work and depth of the following algorithm based on *repeated squaring*. The algorithm converts the graph into an adjacency matrix, A, where $a_{ij} = w_{ij}$ if $(i, j, w_{ij}) \in E$, and ∞ otherwise, and repeatedly squares this matrix over the min-plus semiring. The ij'th entry of the product, C, of two $n \times n$ matrices, A and B over the min-plus semiring is given by

$$c_{ij} = \min_{k=1}^{n} A_{ik} + B_{kj}$$

   (a) [2 pts] What is the work and depth of computing A^2 on the MP-RAM?

   (b) [5 pts] Prove that A^k_{ij} computes the shortest path distance using $\leq k$ edges in G.

   (c) [5 pts] Show how to compute A^n in $O(n^3 \log n)$ work and $O(\log^2 n)$ depth on the MP-RAM. Note that this method solves the more general problem of weighted, directed SSSP.

3. We set out to find an *single-source* shortest-path algorithm, but ended up computing *all-pairs shortest paths (APSP)*! Let's rethink our algorithm and see we can avoid computing shortest-path information between all pairs of vertices, and thereby improve the work. Consider the following algorithm for unweighted, directed SSSP. The algorithm below takes a parameter k that you will determine. The algorithm consists of a *preprocessing phase* which runs in $\tilde{O}(m\sqrt{n})$ work and $O(k \log n)$ depth, and an *query* phase, where (s, t) shortest path queries can be answered w.h.p. in $O(m+n)$ work and $O(k \log n)$ depth. We assume that $m = O(n)$ in this problem. [1]

---

1. Sample a set S of $\sqrt{n}$ *hop nodes* at random from V.
2. From each of the hop nodes, perform a k-limited breadth-first search (a k-limited BFS is just a standard BFS that runs no more than k rounds). During the BFS

[1] $\tilde{O}$ ignores logarithmic factors in n, i.e. $O(n \cdot \mathsf{polylog}(n)) = \tilde{O}(n)$.

from some hop node h_i if any other hop node h_j is encountered, add an edge with weight $(h_i, h_j, dist(h_i, h_j))$ to a weighted graph on just the hop-nodes, H.

3. Compute all-pairs shortest paths on H using the APSP algorithm above.

(a) [10 pts] Argue that if $k = O(\sqrt{n} \log n)$ then any path of length k will have at least one hop-node w.h.p. (recall that w.h.p. means with probability $1 - 1/n^c$ for any constant c). (Hint: you may wish to review the Chernoff bound for this problem.)

(b) [10 pts] Given two arbitrary vertices, $s, t \in V$ how can we compute the shortest path distance from s to t using H? Prove that your method will return the shortest path distance between any two pairs of vertices w.h.p. (Hint: start by performing a k-step BFS from s and t. To prove the high-probability bound you may wish to analyze the probability that a shortest path of length $\geq k$ is preserved between two arbitrary vertices, and take a union bound over all pairs of vertices).

(c) [5 pts] Argue that the work and depth of the preprocessing algorithm is $O(m\sqrt{n})$ and $O(k \log n)$. Argue that the work and depth of a single (s, t) shortest path query is $O(m + n)$ and $O(k \log n)$. What is the work and depth for computing all (s, t) shortest path distances $\forall t \in V$ using this approach?

(d) [5 pts] Suppose that instead of sampling $\sqrt{n}$ vertices, we sample $|S| = \rho$ vertices. What should k be set to as a function of n and ρ in the k-limited BFSs so that shortest path queries are answered correctly w.h.p.? What is the work and depth of preprocessing and the work and depth of an (s, t) shortest path query when $\rho = O(\log^c n)$? What about when $\rho = n^\epsilon, 0 < \epsilon < 1$? (no proof necessary, just state the bounds).