Matthew Feng November 2, 2018

**6.046 Problem Set 7**

Collaborators: *James Lin*

## Problem 1

### (a)

For our deterministic algorithm to work as desired, the best we can do is return "SORTED" immediately if more than $\frac{9}{10}n$ elements are indeed sorted (since we don't care what we output if the array is mostly sorted but not fully sorted), and return "UNSORTED" immediately if we find more than $\frac{1}{10}n$ elements to be unsorted. This is the best because if we don't check $\frac{9}{10}n$ elements, the array could possibly be unsorted, yet we would return "SORTED". Thus, we need to check $\frac{9}{10}n$ elements for a sorted array $A$, which is $\Omega(n)$ runtime.

### (b)

Consider the array $A = [1, 2, 3, ..., n]$ that is rotated by $\frac{n}{2}$ indices, so that the resulting array $A$ is $[n/2, n/2 + 1, ..., n - 2, n - 1, n, 1, 2, 3, ...n/2 - 1]$. $A$ is not mostly sorted, since only by removing (at least) $n/2$ elements can we obtain a sorted array. For $n - 4$ indices, $A[i] < A[i+1] < A[i+2]$. Only for two triples, namely $(n-1, n, 1)$ and $(n, 1, 2)$, will checking three consecutive indices reveal that $A$ is unsorted.

For this adversarial example, if we randomly select an index $i$ from 0 to $n-3$, the probability that $A[i], A[i + 1]$, and $A[i + 2$ will be sorted is $\frac{n-4}{n-2}$. Our algorithm fails if it outputs "SORTED". The probability that this occurs is $p = \Pr[fail] = (\frac{n-4}{n-2})^k$, since we have $k$ independent iterations of testing indices, and all must pass. Our goal is to have $p < \frac{1}{3}$ for all values of $n$. This is not possible unless $k = \Omega(n)$. To see this, suppose the contrary: that $k = o(n)$ can satisfy that $p < \frac{1}{3}$. This means that $\forall \epsilon > 0, \exists N$ such that $\forall n > N$, $k < \epsilon n$, and $(\frac{n-4}{n-2})^k = (1 - \frac{2}{n-2})^k < \frac{1}{3}$. However, this is false. $(1 - \frac{2}{n-2})^k$ is minimized when $k$ is maximized; that is, $p$ is lower-bounded by $(1 - \frac{2}{n-2})^{\epsilon n} \approx (\frac{1}{e^2})^{\epsilon}$. For very small $\epsilon$, this value is larger than $\frac{1}{3}$. Thus, $k$ cannot be $o(n)$, which means it must be $\Omega(n)$.

## (c)

Let us consider the point at which the execution of $i_1 \leftarrow$ BINARY-SEARCH($A$, $x_1$, 0, $n$) diverges from $i_2 \leftarrow$ BINARY-SEARCH($A$, $x_2$, 0, $n$). The only point at which this is possible is if $x_1 < A[median]$ and $x_2 \geq A[median]$, or vice versa. Since $i_1 < i_2$, however, it must be the case that $x_1 < A[median]$ and $x_2 \geq A[median]$, or $x_1 < A[median] \leq x_2 \implies x_1 < x_2$.

## (d)

Suppose (for the sake of contradiction) that $A$ is **not** mostly sorted, yet more than $\frac{9}{10}n$ indices can pass BINARY-SEARCH-TEST. We can order these indices in increasing order $\{i_1, i_2, i_3, ..., i_{9n/10}, i_{9n/10+1}, ...\}$. Then, by the property demonstrated in part (c), we know that $A[i_1] < A[i_2] < A[i_3] < ... < A[i_{9n/10}] < A[i_{9n/10+1}]$. But that means that the array $A$ is mostly sorted. This is a contradiction, and thus no more than $\frac{9}{10}n$ can pass BINARY-SEARCH-TEST for an array $A$ that is not mostly sorted.

## (e)

We can now use BINARY-SEARCH-TEST as a subroutine for our algorithm. The algorithm is as follows:

**Algorithm 1** Check if an array is sorted using randomization

```
function CHECK-SORTED($A$)
    for 100 iterations do
        generate random index $i \in [0, n-1]$
        success ← BINARY-SEARCH-TEST($A$, $i$)
        if success $\neq$ True then
            return "UNSORTED"
        end if
    end for
    return "SORTED"
end function
```

The main idea behind the algorithm is that if BINARY-SEARCH-TEST fails for any index $i \in [0, n-1]$, then $A$ is not sorted. Similarly, the more times BINARY-SEARCH-TEST succeeds, the more probable that $A$ is sorted.

The algorithm calls BINARY-SEARCH-TEST with a random index; if BINARY-SEARCH-TEST fails, then we know that the array $A$ cannot be sorted, and so we return "UNSORTED." However, if our tests succeed a sufficient number of times, then we can be fairly confident that the array is sorted, and thus we return "SORTED" with some probability that we will be incorrect.

Formally, we can argue the correctness of this algorithm through casework. There are three cases:

**Case 1.** $A$ is sorted.

Since $A$ is sorted, Binary-Search-Test will always succeed, and thus we will always return "SORTED", as desired.

**Case 2.** $A$ is mostly sorted, but not (completely) sorted.

In this case, our algorithm may return "SORTED" or "UNSORTED", but we don't care what the output is; thus, this case is satisfactory as well.

**Case 3.** $A$ is not mostly sorted.

First, if $A$ is not mostly sorted, our algorithm fails if it outputs "SORTED." This means that our algorithm fails if all 100 calls to Binary-Search-Test succeed.

Next, since $A$ is not mostly sorted, by the fact shown in (d) that Binary-Search-Test will succeed on no more than $\frac{9}{10}n$ indices, we can say that for a not-mostly-sorted array, each individual call to Binary-Search-Test will succeed **with probability** $\leq \frac{9}{10}$.

Thus, the chance that *every call* of Binary-Search-Test on $A$ succeeds, for $k$ calls, is bounded above by $(\frac{9}{10})^k$. For a large enough constant $k$ (in particular, $k = 22$), this probability drops below $\frac{1}{10}$. In other words, the probability that our algorithm fails for this case is less than $\frac{1}{10}$, for all $k \geq 22$. Since we set $k = 100$, the probability that our algorithm returns the incorrect output is well below $\frac{1}{10}$.

Since $A$ must fall in one of these three cases, and that our algorithm succeeds all the time for cases 1 and 2, and fails with very little probability for case 3, our algorithm fits the specifications. Furthermore, since we make a constant number of calls to Binary-Search-Test, which runs in $O(\log n)$ time, our algorithm runs in $O(1) \times O(\log n)$ time complexity, which remains $O(\log n)$, as desired.

# Problem 2

## (a)

For both of Melon's algorithms, if the length of the random walk is long enough so that every node is visited, then the edges that are selected edges must form a spanning tree.

For both algorithms, we notice that the number of edges we admit is $|V| - 1$.

Then, for algorithm 1, every node $u$ other than the start node $s$ is the destination in exactly one edge $(\cdot, u)$ in the edges we select ($s$ must be the source for at least one of these edges, i.e. $(s, \cdot)$). Since we visit every vertex, there must be exactly $|V|$ distinct nodes adjacent to the edges we have selected. Since there are $|V|$ nodes and $|V| - 1$ edges, the edges we have selected must form a tree. Additionally, since all $|V|$ nodes in $G$ are accounted for, the edge-set we have selected via algorithm 1 must form a spanning tree.

A similar analysis works for algorithm 2. In algorithm 2, every node $u$ other than the final node $s$ must be the source in exactly one edge $(u, \cdot)$, and $s$ must be the destination of one these edges $(\cdot, s)$. Again, all $|V|$ distinct nodes are adjacent to our selected edge-set, which contains $|V| - 1$ edges. Thus our edge must be a tree, and it spans all $|V|$ vertices in $G$.

## (b)

We notice that for any particular random walk $w = (u_0, u_1, ..., u_k)$ that covers all the nodes in $G$, the spanning tree constructed using algorithm 1 is the same as the spanning tree constructed using algorithm 2, if the random walk was performed in reverse as $w_{rev} = (u_k, u_k - 1, ..., u_1, u_0)$. To see this, for algorithm 1, we exclude $u_0$ from the set of vertices we perform our union over; we exclude $u_0$ in algorithm 2 as well. Next, if $u_i \to u_{i+1}$ is the first transition in $w$ where $u_{i+1}$ is visited, then $u_{i+1} \to u_i$ must be the last transition from $u_{i+1}$ in $w_{rev}$. Thus, the set of edges we select from algorithm 1 and 2, using $w$ and $w_{rev}$ respectively, will be the same. What remains to show is that the probability that $w_{rev}$ is generated from a random walk is the same as the probability that $w$ is generated from a random walk. The probability that we generate $w$ is $\Pr[u_0] \times \prod_{i=1}^{k} \Pr[u_i|u_{i-1}] = 1/|V| \times (1/d)^k$. The probability that we generate $w_{rev}$ is $\Pr[u_k] \times \prod_{i=0}^{k-1} \Pr[u_i|u_{i+1}] = 1/|V| \times (1/d)^k$. We see that the probabilities are equivalent, and thus the distribution of spanning trees generated by algorithms 1 and 2 must be the same after the same number of steps.

## (c)

To show that the uniform distribution $u = [1/n, 1/n, ..., 1/n]$ ($u$ is a row vector) is stationary for a Markov chain with $n \times n$ size transition matrix $W$, we need to show that $uW = u$. If we adopt the "row $\times$ matrix" perspective of matrix multiplication, we see that $uW$ is equivalent to $\sum_{i=1}^{n} u_i \times (\text{row } i \text{ of } W) = \sum_{i=1}^{n}(1/n) \times (\text{row } i \text{ of } W) = (1/n)\sum_{i=1}^{n}(\text{row } i \text{ of } W) = (1/n)[1, 1, ..., 1] = u$.

## (d)

First, view every edge as two different halves: an outgoing, source "half," and an incoming, destination "half." Every edge must have both parts. For a digraph $G = (V, E)$, if the out-degree of every node is exactly $d$, then there are exactly $|V|d$ outgoing "halves". Consequently, there must be exactly $|V|d$ incoming "halves". Since the in-degree of any node must be $\leq d$, the total number of incoming "halves" that can exist in $G$ is $\leq |V|d$, with equality if and only if every node has an in-degree of $d$. Thus, since we have exactly $|V|d$ incoming halves, this implies that every node has an in-degree of $d$.

## (e)

If we can show that any node in the Markov chain as described above (i.e. on the node set $V(G) \times \mathcal{T}$) can have in-degree at most $d$, and we know that every node in the Markov chain has out-degree of $d$ (since every node $v \in V(G)$ is connected to $d$ neighbors), then by Part (d) we know that the in-degree of every node must be $d$.

Suppose, for the sake of contradiction, that $(u, T)$ has an in-degree $\hat{d}$ greater than $d$. We note that we can "reverse" any transition $(v_i, T_i) \to (u, T), \forall i \in [1, \hat{d}]$ (the $v_i$'s may not be distinct, but the $T_i$'s will be); that is, we can delete $(u, v_i)$ in $E(T)$ to form $T_u$ and $T_{v_i}$, two trees rooted at $u$ and $v_i$, respectively, and reattach $u$ to its original parent $\pi_i(u)$ (which may be $v_i$) in $T_i$. This means that $u$ has $\hat{d}$ distinct valid parents, one for each $T_i$ rooted at $v_i$. However, in order for $\pi_i(u)$ to be a parent, $\pi_i(u)$ must have been connected to $u$ in $G$, i.e. $(\pi_i(u), u) \in E(G)$. But this means that $u$ has $\hat{d} > d$ neighbors in $G$, which is impossible.

Thus, the number of transitions that end at $(u, T)$ is cannot be greater than $d$. This is exactly what we wanted to show; as such, this fact implies, by Part (d), that the in-degree of every node in the Markov chain is exactly $d$.

Now, we need to show that the transition/walk matrix $W$ formed by this Markov chain is doubly stochastic, that is, every row and every column sums to 1. $W_{ij}$ may represent the transition from node $i$ to node $j$ (we are using left matrix multiplication). Then every row $i$ in $W$ represents the transition distribution across all other nodes, starting at $i$. Since there are $d$ outgoing edges from $i$, each with probability $1/d$, for every $i$, the sum of every row is 1. Likewise, the column $j$ in $W$ represents all the edges incoming to $j$ and their respective probabilities. We showed that the in-degree for any node $j$ must be $d$, and each transition always has probability $1/d$, and thus all the columns must also sum to 1.

Thus, since $W$ is doubly-stochastic and converges to a unique, stationary distribution, that distribution must be uniform across all nodes $V(G) \times \mathcal{T}$, which yields a uniform random distribution over $\mathcal{T}$, as desired.