COS 226 **Algorithms and Data Structures** **Fall 2024**

Final Solutions

1. **Initialization.**

   Don't forget to do this.

2. **Graph search algorithms.**

   (a) 0 2 6 8 5 3 4 9 1 7

   (b) 0 2 5 3 6 8 4 9 1 7

   (c) 3 5 2 6 1 7 9 4 8 0

   (d) *yes*
   For every edge $u \rightarrow v$, u appears before v in this ordering.

3. **Minimum spanning trees.**

   (a) 0.4 0.2 0.1 0.3 0.55 0.6 0.25

   (b) A B C E F G

   (c) 2
   The edges (C, D) and (G, H) cross the cut.

   (d) *yes*
   Since there is no cut with a single crossing edge and this cut has two, it is a mincut.

4. **Shortest paths.**

   (a) 60

   (b) 0 5 4 1 2 3
   The shortest path lengths are 0, 110, 120, 145, 90, 70.

   (c) 0 110 130 155 90 70

   (d) *yes*
   The second pass of Bellman–Ford updates the distances of `2` *and* `4`*.*

   (e) *no*
   The `distTo[]` *values after the second pass match the* `distTo[]` *values obtained from Dijkstra execution in part (b). Since these latter values are optimal and since* `distTo[]` *values can only decrease, no further changes will occur after the second pass.*

5. **Maxflows and mincuts.**

   (a) $36 = 14 + 22 = 36 + 0$

   (b) $113 = 35 + 24 + 18 + 36$

   (c) $A \to F \to G \to C \to I \to J$

   (d) $42 = 28 + 14$

   Verify that the value of the maximum flow matches the capacity of the minimum cut, which is $42 = 40 + 2$.

   (e) A B C F G H

   (f) *42*

   The flow can only increase. However, increasing the capacity of the edge $I \to C$ *does not affect the capacity of the minimum cut found in part (e).*

   (g) *36*

   The capacity of the cut found in part (e) decreases by 6. *Since no other cut decreases by more than* 6, *this cut remains the minimum cut.*

6. **Properties of graph algorithms.**

   T T F F F

   (a) T: A spanning tree contains exactly $V - 1$ edges. Adding any extra edge will create a cycle.

   (b) T: At the time when `dfs(u)` is called, since there is a path from u to v and v has not been visited before (as u is marked before v), `dfs(v)` will be called before `dfs(u)` finishes and will therefore return before `dfs(u)` does.

   (c) F: Consider two graphs, each with three vertices: s, t, and v, and two edges: $s \to v$ and $v \to t$. In the first graph, the edge $s \to v$ has capacity 1, and $v \to t$ has capacity 3. In the second graph, the edge $s \to v$ has capacity 3, and $v \to t$ has capacity 1. In both graphs, the value of the maximum flow is 1, but in the union graph, the value of the maximum flow is 3.

   (d) F: Consider a graph with five vertices: s, t, u, v, and w, and edges $s \to u \to v \to w \to u \to t$, each with a capacity of 1. Sending 1 unit of flow through all edges achieves the maximum flow, yet the cycle $u \to v \to w \to u$ carries a positive flow.

   (e) F: The vertex with an out-degree of 0 must be the first in any topological ordering. The remaining vertices can be arranged in any order. Thus, there are $n!$ possible orderings.

7. **Dynamic programming.**

```
static int minCandies(int[] candies) {
     int n = candies.length;
     int[] opt = new int[n];

     opt[0] = candies[0];
     opt[1] = candies[1];

     for (int i = 2; i < n; i++)
         opt[i] = candies[i] + Math.min(opt[i - 1], opt[i - 2]);

     return Math.min(opt[n - 1], opt[n - 2]);
}
```

8. **Multiplicative weights.**

F F T T F

(a) F: Partition the experts into three groups: S_1, S_2, and S_3, with $|S_1| = 0.3n$, $|S_2| = 0.4n$, and $|S_3| = 0.3n$. Assume that on day 1, the experts in S_1 make a mistake; on day 2, the experts in S_2 make a mistake; and from day 3 onward, all experts predict correctly. It is evident that both algorithms are correct on all days except, possibly, day 2. The elimination algorithm makes a mistake on day 2 because $0.4n$ out of the remaining $0.7n$ experts are wrong. In contrast, the multiplicative weights algorithm is correct on day 2: at that point, the total weight of S_1 is $0.15n$, the weight of S_2 is $0.4n$, and the weight of S_3 is $0.3n$. Thus, the total weight of correct experts ($0.45n$) exceeds the total weight of incorrect experts ($0.4n$).

(b) F: For any day $t < T$, it is possible for all but one expert to make their first mistake on day $t + 1$. The combined weight of those experts is $n - 1$, which exceeds the weight of the single correct expert.

(c) F: This modification does not impact the analysis of the multiplicative weights algorithm.

(d) T: Execute 7 independent instances of the elimination algorithm (or the multiplicative weights algorithm), one for each day.

(e) F: Consider the scenario where, in each iteration, the decision stump correctly labels all points except the first one. This stump correctly labels at least 90% of the total weight as, ignoring normalization, after $T = 100$ iterations, the weight of the first point becomes 2^{100}, while the total weight of all other points is $n - 1$. For sufficiently large n, $n - 1$ is more than 90% of the total weight, which is $n - 1 + 2^{100}$. Furthermore, since the algorithm determines the label of each point based on the majority output of the stumps, it will mislabel the first point.

9. **Intractability.**

T T F F F

(a) T: Since X is in **P**, it has a $p(n)$-time algorithm for some polynomial $p(n)$. To prove that X poly-time reduces to Y, it is enough to show that if Y has a $T(n)$-time algorithm for some $T(n) \geq n$, then X has a $T(p(n))$-time algorithm. However, since X already has a $p(n)$-time algorithm, it follows that X also has a $T(p(n))$-time algorithm for any $T(n) \geq n$ (regardless of the running time of any algorithm for Y). A symmetric argument shows that Y poly-time reduces to X.

(b) T: Since Y is in **P**, it is also in **NP**. Because 3-SAT is **NP**-complete and poly-time reduces to Y, it follows that Y is **NP**-complete. However, if any **NP**-complete problem has a polynomial-time algorithm, then **P** = **NP**.

(c) F: The first algorithm is polynomial time because a boolean array of length n is represented using $\Theta(n)$ bits, and the algorithm runs in $\Theta(n)$ time, which is linear in the input size. In contrast, the second algorithm is not polynomial time. The integer x is represented using $\Theta(\log x)$ bits, but the algorithm has a running time of $\Theta(x)$, which is exponential in the input size.

(d) F: Repetitions of a Monte Carlo randomized algorithm can reduce the error probability, but the issue here is improving the approximation factor. (In fact, unless **P** = **NP**, no polynomial-time approximation algorithm for 3-SAT can satisfy more than $\frac{7}{8}$ of the equations).

(e) F: The verification algorithm does not run in polynomial time because, for even n, half of the subsets $S \subseteq [n]$ have size $\leq \frac{n}{2}$, resulting in 2^{n-1} such subsets.

10. **Randomness.**

F T T T F

(a) F: `majA` is Monte Carlo (deterministic running time, not guaranteed to be correct). `majB` is Las Vegas (guaranteed to be correct, running time depends on randomness).

(b) T: A majority element is found in each iteration of the loop with a probability greater than $\frac{1}{2}$.

(c) T: The expected number of loop iterations is $\Theta(1)$ because the probability of halting after each iteration is at least $\frac{1}{2}$. Since each iteration takes $\Theta(n)$ time, the total runtime is $\Theta(n)$.

(d) T: As the frequency of the majority element increases, the probably of one iteration of the loop finding the majority element also increases. This reduces the expected number of iterations. (The number of array accesses per iteration does not change.)

(e) F: The number of iterations and the number of array accesses per iteration remain the same. As a result, the overall number of array accesses stays unchanged.

11. **Design: majority element.**

(a) **Full credit.** For full credit, observe that if x is the majority element of `arr`, then x must be the majority element of either the left or right subarray of `arr`. The reasoning is that if x appears in at most half of the entries in each subarray, it cannot appear in more than half of the entries in `arr` and, therefore, cannot be its majority element. This leads to a divide-and-conquer algorithm that recursively computes the majority elements of the two subarrays and checks whether either of the obtained candidates is the majority of the entire array. The running time satisfies the recurrence $T(n) = 2T(n/2) + \Theta(n)$, which resolves to $T(n) = \Theta(n \log n)$.

Partial credit. For partial credit, sort the array and check if `arr[n/2]` is a majority element. Note that if a majority element exists, it will appear at least $n/2+1$ consecutive times after sorting, which guarantees that it appears at the position $n/2$. Another option is to count how many times each element appears in the sorted array.

(b) Use the *frequency counter* we implemented in the lecture on symbol tables, utilizing a hash table. Then, check whether the maximum frequency exceeds $n/2$.

Recall that the frequency counter treats array objects as keys, with values representing their counts. It processes the array by iterating through each index i, checking if `arr[i]` is already a key in the hash table and, if so, retrieving the associated counter c. If the key does not exist, it is added with an initial count value of 1. Otherwise, it is re-added with the associated counter $c+1$. Finally, the algorithm scans the hash table to identify the highest counter value.

Remark: There are algorithms that find the majority element in $\Theta(n)$ time in the worst case without making any assumptions. One such approach uses an auxiliary array `aux`. For every $0 \le k < n/2$, compare entries $2k$ and $2k+1$ using the `equals()` method. If they are equal, copy one of them to `aux`; otherwise, copy neither. The number of elements copied to `aux` is at most $n/2$. Replace `arr` with `aux` and repeat the process to find the majority element of the shorter array. This algorithm requires $\Theta(n)$ space and runs in $\Theta(n)$ time, as the total work is proportional to $n + \frac{n}{2} + \frac{n}{4} + \cdots + 1$, which forms a geometric sum summing to $\Theta(n)$.

Another $\Theta(n)$-time algorithm is based on the *Boyer–Moore majority vote algorithm.* this algorithm is in-place and scans the array only twice. Below is a Java implementation:

```
static Object maj(Object[] arr) {
     Object maj = null;
     int count = 0;

     for (int i = 1; i < arr.length; i++) {
          if (count == 0) {
               maj = arr[i];
               count = 1;
          }
          else if (arr[i].equals(maj))
               count++;
          else
               count--;
     }

     if ((maj != null) && isMaj(arr, maj))
          return maj;

     return null;
}
```

12. **Design: reachability with even-length paths.**

(a) *yes:* Consider the directed graph with the edges $s \to u \to v \to s \to t$. While there is no simple even-length path from s to t, the path that includes all the edges forms a (non-simple) even-length path from s to t.

(b) **Full credit via reduction.** A possible full-credit solution involves constructing a new graph that contains an "even" and an "odd" copy of each vertex in the original graph G. For every vertex v in G, the new graph includes two vertices, $(v, \texttt{even})$ and $(v, \texttt{odd})$. For every edge $u \to v$ in G, the new graph includes the edges $(u, \texttt{even}) \to (v, \texttt{odd})$ and $(u, \texttt{odd}) \to (v, \texttt{even})$. Notice that after taking on odd number of steps starting from $(s, \texttt{even})$, one always ends up at an `odd` vertex. So, the algorithm then runs either BFS or DFS on this new graph, starting from $(s, \texttt{even})$, to check whether $(t, \texttt{even})$ is reachable. This approach is applicable to any graph (not just DAGs) and requires $\Theta(E + V)$ memory, as it involves creating two copies of the graph.

We note, however, that explicitly constructing the new graph is not strictly necessary. For example, the DFS algorithm can be modified to take an additional parity bit as input: `dfs(v, parity)`. Instead of a standard marked array, a marked array indexed by pairs *(v, parity)* is used. The algorithm starts with `dfs(s, 0)`. When `dfs(u, 0)` is called, it marks `(u, 0)` and recursively calls `dfs(v, 1)` for each vertex v connected by an edge $u \to v$. Similarly, when `dfs(u, 1)` is called, it marks `(u, 1)` and recursively calls `dfs(v, 0)` for each vertex v connected by an edge $u \to v$.

Full credit via dynamic programming. An alternative full-credit approach leverages the fact that G is a DAG. First, perform a topological sort starting from s, and maintain

two arrays of length V: `even[]` and `odd[]`. Then, iterate over the vertices v in topological order. For each vertex v with an incoming edge $u \to v$, update the arrays as follows: if `even[u] = true`, set `odd[v] = true`; if `odd[u] = true`, set `even[v] = true`.

This is effectively a dynamic programming solution, where the subproblem `(i, even)` determines whether vertex i in the topological order has a path of even length from s, and `(i, odd)` determines if vertex i in the topological order has a path of odd length from s. Since the vertices are processed in topological order, the subproblems for vertex i in the topological order depend only on the solutions to the subproblems for vertices 0 through $i-1$. Moreover, by the time the algorithm processes vertex i, the subproblems for vertices 0 through $i-1$ have already been resolved.

Partial credit. While the graph contains at least one edge incident to s, run a BFS or DFS from s to search for a path from s to t. If no path exists, return `NO`. If an even-length path is found, return `YES`. Otherwise, if an odd-length path is found, remove all edges on this path from the graph and restart the process. Since each iteration either terminates the algorithm or removes at least one edge, the loop executes at most E times, with each iteration running a BFS or DFS in $\Theta(E+V)$ time.