COMPSCI 311: Introduction to Algorithms

Second Midterm Exam, November 14, 2018

Name: ______________________________ ID: ____________________

- Answer the questions directly on the exam pages.
- Show all your work for each question. More detail including comments and explanations can help with assignment of partial credit.
- Questions have different point values. Move on to another question if you get stuck.
- If you need extra space, use the back of a page.
- No books, notes, or electronic devices are allowed. Any cheating will result in a grade of 0.
- If you have questions during the exam, raise your hand.

1. (5p) In a binary tree with integer-valued nodes, we call a node v *well ordered* if no node in the left subtree of v is greater than v and no node in the right subtree of v is less than v. Design an efficient algorithm that counts the well-ordered nodes in a binary tree and analyze its complexity.

Solution: Since a node must be compared with *all* descendants, we write a recursive function that returns a triple: next to a count of well-ordered nodes, also the minimum and maximum node value in a tree. The helper function also takes as parameter the parent value, to uniformly handle the case of one or both empty subtrees. If a node value is between the left maximum and the right minimum, it is well-balanced and 1 is added to the counts from the subtrees.

```
function minmaxroot(t, parent)
if t = nil then
  return (0, parent, parent)
else
  (lcnt, lmin, lmax) = minmaxroot(t.left, t.val);
  (rcnt, rmin, rmax) = minmaxroot(t.right, t.val);
  return (lcnt + rcnt + (t.val ≥ lmax && t.val ≤ rmin), min(lmin, t.val, rmin),
      max(lmax, t.val, rmax));
function countwellordered(t)
return first(minmaxroot(t, 0)); // extract first of triple; arg. 0 does not matter
```

■

The algorithm traverses all nodes doing constant work at each, thus is has complexity $\Theta(n)$.

Comments: It is important that the minimum and maximum are computed and returned for use by the parent node. Calling separate minimum and maximum functions repeats work and leads to higher complexity: $\Theta(n \log n)$ for a balanced tree, $\Theta(n^2)$ if the tree degenerates into a list.

2. (4p) Derive asymptotic upper and lower bounds for the following recurrences:

a) $T(n) = 3T(n/9) + \sqrt{n}$

Solution: The easiest way to solve this is to use the Master Theorem: here $a = 3$, $b = 9$, and $f(n) = \sqrt{n}$. Note that $n^{\log_b a} = n^{\log_9 3} = n^{1/2}$, so

$$f(n) = \sqrt{n} = \Theta(n^{\log_b a}),$$

and so Case 2 of the Master Theorem applies, and we have $T(n) = \Theta(n^{\log_b a} \log n) = \Theta(\sqrt{n} \log n)$. You can solve this using recursion trees as well. ■

b) $T(n) = T(n-2) + \log n$

Solution: We can "unroll" as usual (abusing notation slightly to not worry about whether n is even or odd) to get

$$T(n) = T(n-2) + \log n = T(n-4) + \log(n-2) + \log n = \ldots = \sum_{k=1}^{n/2} \log(2k).$$

Using $\log(2k) = \log 2 + \log k = 1 + \log k$, we get

$$T(n) = \sum_{k=1}^{n/2}(1 + \log k) = \frac{n}{2} + \sum_{k=1}^{n/2} \log k.$$

We have seen numerous times in this course that $\sum_{k=1}^{n/2} \log k = \log((n/2)!) = (n/2)\log(n/2) = \Theta(n \log n)$, which absorbs the $n/2$ additive factor as well, so we are left with $T(n) = \Theta(n \log n)$. ■

3. (2p) While running sequence alignment experiments with mismatch cost 1, Max noticed that for positive real gap penalty values δ that are sufficiently small, the aligned letters of the two strings would form a longest common subsequence. If Max is right, state and prove a sufficient condition on δ (possibly in terms of the string lengths) for this to occur, otherwise give a counterexample.

Solution: Let L_{tot} be the sum of the two string lengths. Without mismatches, the best alignment matches a longest common subsequence (of length L_{cs}) and has cost $\delta \cdot (L_{tot} - 2L_{cs})$. If $\delta < 1/L_{tot}$, the cost is < 1, the cost of a single mismatch, so this is indeed the optimal alignment.

In fact, $\delta < 1/2$ suffices. For x mismatches and m matches, the cost is $x + \delta \cdot (L_{tot} - 2 \cdot (m + x)) = x(1 - 2\delta) + \delta(L_{tot} - 2m)$. Since $m \leq L_{cs}$, this will be larger than $\delta \cdot (L_{tot} - 2L_{cs})$ when $\delta < 1/2$. ■

Comments: A common answer was $\delta < 1/\max(l_1, l_2)$, from a mistaken maximum number of gaps, compared to the cost of a single mismatch. These were graded 1.5 out of 2 points.

4. (7p) Like in the homework, you and the evil mathematician in turn take a coin from either end of a row of $2n$ distinct coins. Both of you play optimally for maximum gain. If in your turn, both choices would lead to the same total gain at the end, greedily take the larger coin. This gives a unique strategy. Design an algorithm that computes in linear space your best move at each step, and analyze its time complexity.

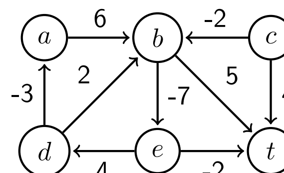
Solution: We can use precisely the same solution as in the homework problem, where we used the max-min recurrence $M_{i,j} = \max\{\min\{M_{i+2,j}, M_{i+1,j-1}\} + X[i], \min\{M_{i+1,j-1}, M_{i,j-2}\} + X[j]\}$ with initial conditions $M_{i,i+1} = \max\{X[i], X[i+1]\}$. In addition, we can also store a pointer at cell $[i,j]$ to which of $X[i]$ or $X[j]$ the maximum came from (this is where we use the condition here for breaking ties – if $X[i]$ and $X[j]$ both lead to the same finishing value, we greedily pick the larger of the two). The solution here is to do the same thing at each subproblem $[i,j]$, except to store only a linear number of entries at any point. Remember that in the homework problem, the solution is an induction on the quantity $j - i + 1$, with step size 2. Each fixed value of $j - i + 1$ corresponds to a specific size of a subproblem, where we can fix all the entries with $j - i + 1 = 2$, and then use them to fix all the entries with $j - i + 1 = 4$, and so on. Importantly, for any *fixed* value of $j - i + 1$, there are only at most n subproblems that correspond to that size. The idea here is to only store entries corresponding to two consecutive even values of $j - i + 1$. Once we compute one such value, we can forget and overwrite the lower value of $j - i + 1$. We can keep going until we reach the cell corresponding to the current value of $[i,j]$, at which point we specify our move.

The tradeoff here is that since we overwrite most smaller entries of the table, we lose information about the pointers going back through the dynamic array. Therefore, once the mathematician makes his move, we have to recompute all of this (in linear space) once again to determine our move. We can keep doing this, never using more than linear space.

Each computation takes $O(n^2)$ time, since we fill out up to $O(n^2)$ values of the dynamic table (including overwriting). There are $n/2$ moves made by us, and so the running time is $O(n^3)$. It is not asymptotically smaller than this, since the first $n/4$ moves can be shown to take $\Omega(n^3)$ time.

This is the point of *time-space tradeoff*, a concept central to a number of dynamic programming problems. If we optimize one of time and space, we would have to compromise on the other. ∎

5. (6p) For the given graph, find shortest distances to t using the Bellman-Ford algorithm with a check to detect negative cycles. Explain how and when the cycle is detected and whether you can still claim valid shortest paths from some nodes.

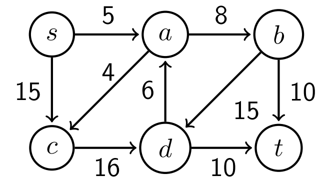


Solution: A simple implementation updates at each iteration $d[u] = \min_v(d[u], c(u,v) + d[v])$ for all neighbors u of nodes v whose value has **changed** in the previous iteration.

	a	b	c	d	e	t
0 :	∞	∞	∞	∞	∞	**0**
1 :	∞	**5**	**4**	∞	**−2**	0
2 :	**11**	**−9**	**3**	**7**	−2	0
3 :	**3**	−9	**−11**	**−7**	−2	0
4 :	3	−9	−11	−7	**−3**	0
5 :	3	**−10**	−11	−7	−3	0
6 :	**−4**	−10	**−12**	**−8**	−3	0
succ :	b	e	b	b	d	–

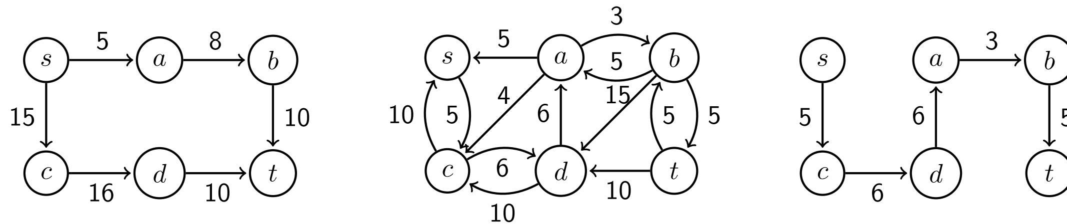
If values still change at iteration n (= 6), there is a negative cycle. We can find such cycles by keeping track of each node's successor on the path to t (the neighbor that determined the last update). Any node updated at iteration n (e.g., a) must reach the cycle; we can mark the nodes and use the pointers to print it (b–e–d–b) once we reach a marked node (b) again. Any node reaching a negative cycle cannot have a shortest path to t: we can find these by reversing the successor graph and doing a bread-first search. The only remaining node that has a shortest path to t is t itself (distance 0).
We could find the cycle earlier, at iteration 4, by checking for cycles when the successor pointers are updated (here, for e). ∎

6. (5p) For the given graph, find a maximum flow by always choosing a shortest augmenting path and updating the level graph when needed. Explain how the algorithm works.



Solution:
A breadth-first search in the original flow graph (which is also the first residual graph, for flow 0) produces the level graph below (where we keep only edges going one level down).

Searching the level graph for paths from s to t produces two paths, $s \to a \to b \to t$ with flow 5, and $s \to c \to d \to t$ with flow 10. After deleting the two saturated edges, no $s - t$ paths in the level graph remain. The residual flow graph and the resulting level graph are shown further right.



There is a single path in the level graph, with flow value 3, for a total of 18. The resulting residual graph has no $s - t$ paths (only a, c and d are reachable from s). This is the maximum flow.

Comments: The problem asked *how* the algorithm finds paths of increasing length by updating the level graph. Solutions that gave the paths with no indication of how they were found (breadth-first search or level graph) got 4.5p. Stating the flow without individual paths got 4p. ■

7. (5p) Consider a flow network which in addition to edge capacities has *node capacities*: each node v has a limit $\ell(v)$ on how much flow can pass through v. Show how to reduce this problem to a regular network flow problem (without node capacities) and prove your reduction correct.

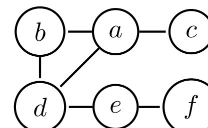
Solution: Given a graph $G = (V, E)$ with the usual edge capacities, together with node capacities $\ell(v)$ for each v, we can construct G' from it as follows. "Duplicate" each $v \in V$ into two vertices, v^- and v^+. The edges incident to and from v in G are separated into two sets: all incoming edges into v are now incoming into v^-, and all outgoing edges from v are now outgoing from v^+. None of these edges change their capacities. In addition, there is a single edge of capacity $\ell(v)$ directed from v^- to v^+.

We can do this for all vertices $v \in V$. This forms our new network G'. Note that G' has edge capacities defined for all edges, but no vertex capacities.

Now, note that this is a correct reduction: finding an edge and vertex-capacity constrained flow on G is equivalent to finding a "normal" edge-capacity-constrained flow on G'. This is because any flow entering a vertex v^- in G *must* pass through the edge (v^-, v^+), which has capacity $c(v)$, and conversely, any flow passing through an edge in G' corresponds either to a flow passing through the same edge in G constrained by its edge capacity, or to a vertex in G constrained by its vertex capacity. Therefore, the reduction is correct.

The new graph G' has exactly $2V$ vertices, and $V + E$ edges. Therefore, the problems are of comparable size. The running time of Ford-Fulkerson is also the same (assuming $V = O(E)$, which is always true for all connected networks, which is a reasonable assumption for flow networks). ■

8). (6p) A *dominating set* of an undirected graph is a set of nodes D such that each graph node is either in D or has an edge to a node in D. The DOMINATINGSET problem asks whether a graph has a dominating set of size $\leq k$, for given k.



For example, in the given graph, b, c and d have edges to a, and f has an edge to e, thus $\{a, e\}$ is a dominating set.

a) Show how to reduce DOMINATINGSET to SETCOVER and prove your reduction correct.

Solution: Given an arbitrary instance of DOMINATINGSET, we wish to construct an instance of SETCOVER so that the first instance has a dominating set of size $\leq k$ if and only if the second has a set cover of size $\leq k$. Furthermore, this construction needs to take polynomial time.

Given a graph G, we can define our universe to be the set of vertices of the graph G, and our collection of subsets to be $S = \{S_v := v \cup N_v : N_v \text{ is the neighborhood of } v \text{ in } G\}$. Clearly, S covers the entire universe. The construction of this subset system given G clearly takes polynomial time.

Suppose G has a dominating set of size $\leq k$. Let the vertices of this dominating set be $\{v_1, \ldots, v_r\}$ for $r \leq k$. Then, $S_{v_1}, \ldots, S_{v_r}$ is a subcollection of r subsets in S whose union covers the entire universe. Conversely, if we have a collection of subsets $S_{v_1}, \ldots, S_{v_r}$ in S whose union covers the entire universe, then consider the set of vertices $D = \{v_1, \ldots, v_r\}$. Since the collection covers the universe, any arbitrary $v \in V$ is in some subset, say S_{v_m}. This means by construction that $v \in \{v_m \cup N_{v_m}\}$, so v is either in the subset D or has an edge into some vertex in D, and so D is a dominating set. This proves the reduction. ■

b) What does this reduction tell you about the NP-completeness of DOMINATINGSET ?

Solution: Nothing. If the reduction had been in the other direction, we would be able to conclude that DOMINATINGSET is also NP-complete (since SETCOVER is), but this direction just tells us that DOMINATINGSET is at *most* as hard as SETCOVER. Interestingly, DOMINATINGSET *is* known to be NP-complete, but this particular reduction does not enable us to conclude that fact; the usual way to prove NP-completeness for DOMINATINGSET is to reduce VERTEXCOVER to it, and use the fact that VERTEXCOVER is NP-complete. ■

Question	Value	Points Earned
1	5	
2	4	
3	2	
4	7	

Question	Value	Points Earned
5	6	
6	5	
7	5	
8	6	
Total	40	