Cheat-sheet #2 for CS201

**GRAPHS**:

Directed graph: edges are uni-directional **|** Weighted graph: Each edge has associated weight **|** A Euler trail is always the longest trail in an undirected graph: True **|** All undirected graphs have a Euler trail: False **|** A Hamiltonian path is always the longest path in an undirected graph: True **|** All undirected graphs have a Hamiltonian path: False **|** The lonest path in an undirected graph is bounded by: minimum degree of graph. **|** An undirected graph with a min-degree of 2 must have cycle: True **|** All connected, undirected graphs have a spanning tree: True. **|** Kruskal’s algorithm is used to find: minimum spanning tree **|** Dijkstra’s algorithm is used to find: The shortest path between a source vertex and the other vertices. **|** Prim’s algorithm is used to find: minimum spanning tree. **|** Floyd-Warshall algorithm is used to find: all-pairs shortest paths. **|** Suppose E = ω(V). What is the asymptotic running time of Kruskal’s algorithm? Θ(E lg E). **|** Suppose E = Θ(V). What is the asymptotic running time of Prim’s algorithm? Θ(E lg E). **|** Suppose you keep track of the level number for each vertex w in a breadth-first search of a simple, undirected, unweighted graph, starting from vertex v. The level numbers of each w would correspond to the shortest path distance: True. **|**  For a simple, undirected graph, Θ(E log E) = Θ(E log V): True. **|** # of edges in a COMPLETE graph: n(n-1)/2, where n is the # of vertices. **|** Consider running Kruskal’s algorithm on the complete graph KN, processing the first i edges. What is the smallest value of i that could yield the final results?: N-1 **|** Consider running Dijkstra’s algorithm using a linked-list (with a tail pointer) as the basis for a priority queue. What is the asymptotic running time for the algorithm? Answer was not listed…

**MEMOIZATION:**

function F1(x) { function F2(x, items, y) {

if(x == 0) return 0; if(x == 0) return 1; // base case 1

if(x == 1) return 1; if(x < 0) return 0; // base case 2

return F1(x-2) + F1(x-1); if(y == items.size) return 0; // base case 3

} return min(F2(x-items[y], items, y), F2(x, items, y+1));

}

Questions for F1:

What would be memoization table’s largest index/indeces? “x” **|** To remove the two base cases, how would the memorization table be initialized? memo[0] = 0; memo[1] = 1; **|**  What base case would replace the two original base cases, assuming a properly initialized memoization table? “if(memo[x] != EMPTY) return memo[x];” **|**

Questions for F2:

Assume all three original base cases are retained and the smallest memoization table possible—what would be the memo table’s largest index\indeces, where x refers to the value of x? x & items.size – 1 **|** Which of the original base cases cannot be removed, given no knowledge of the values in items? Base case 2. **|** Removing all possible base cases, what would be the memoization table’s largest index, indeces? Assume no knowledge of the values in items. x and items.size.

**DYNAMIC PROGRAMMING:**

function f(a, b, c, d, e) }

if(a == 0) return 0;

if(a < 0) return –INFINITY;

if(d == e) return 0;

return max( f(a, b, c, **d + 1**, e), f(**a-b[d],** b, c, d, e) + c[d] );

}

Questions for f:

What would be the dimensionality of the dynamic programming table? **b & c & e never change, so don’t have to track those, but a & d DO change/are updated, so dimensionality = 2** **|** How would the dynamic programming table be filled, using **a** as an index? **smaller to larger. |** How would dynamic programming table be filled using **b** as an index? **b is not used as an index |** How would the dyn. Programming table be filled using **d** as an index? **largest to smallest**. **|**

**LINEAR SELECTION:**

🡺 Suppose you wish to find both the min and max values in an array of n values, n being odd. Consider this algorithm: i) make pass through the array and find the min ii) make a pass through the array and find the max iii) report the min and max. What is the min **# of comparisons** (excluding termination check) that this alg. needs to perform? **2n-2**

🡺Suppose you wish to find both the min and max values in an array A of n values, n being odd. Consider the alg.: i) set the max and min to the first value in the array ii) make a pass through the array (i = 1; i < n; i += 2) iii) compare A[i] with A[i+1] iv) update the min if the smaller of the two is less v) update the max if the larger of the two is greater vi) report the min and max at loop termination. What is the minimum **# of comparisions** (excluding termination check) that this algorithm needs to perform? You compare A[i] and A[i+1] and if A[i] < A[i+1], smaller = A[i] and larger = A[i+1], else smaller = A[i+1] and larger = A[i], so that’s ONE comparison done. Then, if smaller < min, set min, and if larger > max, set max. There’s TWO MORE comparisons for a total of **THREE PER PAIR OF VALUES.** Now, the # of pairs evaluated is (n-1)/2; therefore, total # of comps = 3[(n-1)/2]

🡺Consider running the linear selection algorithm on an array of n unique elements. What is a tight lower bound on the # of elements less than the median of medians?

[\*\*\*OO] [\*\***M**OO] [OOOOO], \* = lower than MOM, O = NOT lower than MOM, M = MOM

Assume the MOM is found with groups of five and that there an odd # of groups. Total # of groups n/5. Then, the number of groups < the MOM group: (n/5 – 1)/2. Then, the # of elements in the groups < mom group guaranteed to be < MOM: 3(n/5 -1)/2. Mom group has 2 extra: 3(n/5 – 1)/2 + 2. Then, simplified tight lower bound 3(n/5 – 1)/2 + 2 = **(3n + 5)/10**

**🡺** Consider running the linear selection algorithm on an array with n = 7K unique elements. What is a reasonable recurrence equation to describe the running time of the algorithm? Assume MOM is found with groups of 7.

[OOOO\*\*\*] [OOOM\*\*\*] [\*\*\*\*\*\*\*]

Groups < MOM group have 3 elements > MOM & groups > MOM group have 7 elements > MOM. MOM group has 3 extra. **Groups below: 3(n/7 -1)/2 Groups above: 7(n/7 -1)/2 Extra: 3** We’re talking about runtimes here, so ignore the extra 3: **3(n/7 -1)/2 + 7(n/7 -1)/2** This simplifies to **5n/7 – 5**, but again we’re talking about runtimes so ignore the constant: **5n/7.** Plug this into the formula:

T(n) = T(n/grouping #) + T(calculated polynomial) + Theta(n)

Then, **T(n) = T(n/7) + T(5n/7) + Theta(n)**

🡺 If the linear selection algorithm uses groups of three to find the MOM, the asymptotic run time is still O(n): False (b/c of MRT)

🡺 If the linear selection algorithm uses groups of seven to find the MOM, the asymptotic run time is still O(n): True (b/c of MRT)

**DECISION TREES:**

**🡺** In proving a tight lower bound for a class of algorithms, one tries to establish, over all possible algorithms: the best possible worst-case.

🡺 In an efficient decision tree for comparison sorts of n numbers, what is the smallest possible depth of a leaf, in the worst case? Assume the root is at depth 0: **n – 1**

**🡺** In an efficient decision tree for comparison sorts of n numbers, what does the shortest path from the root to a leaf represent? **The best case behavior of the sort**.

🡺 An efficient decision tree for comparison sorts of n numbers has how many leaves? **n!**

🡺 Deriving a tight lower time bound for comparison sorts, based upon an efficient decision tree, yields: **Ω(n log n)**

**EXTRA STUFF:**



