Solutions to Final Exam Review Problems

Problem 1. Finding two integers

Let X be an unordered set of n distinct integers. For each of the following parts, briefly describe your algorithm and state its running time. Your grade will depend on both the clarity of your description and the efficiency of your algorithm. There is no need to prove the correctness of the algorithm or prove your stated running time.

(a) Give an algorithm to determine distinct $x, y \in X$ that maximize $|x - y|$.

Answer:

1. Find largest element M in X.
2. Find smallest element m in X.
3. Return M and m.

Running time is $\Theta(n)$.

(b) Give an algorithm to determine $x, y \in X$ that minimize $|x - y|$.

Answer: This is the same as finding closest pair in one-dimension.

1. Sort X.
2. Perform a linear scan through the sorted list, maintaining the pair of items in consecutive indices in the sorted list that have the least difference.
3. Return the pair that has the least difference.

Running time is $\Theta(n \log n)$.

(c) Let M and m denote the maximum and minimum element of X, respectively. Give an algorithm to determine any two distinct elements $x, y \in X$, such that $|x-y| \leq (M-m)/(n-1)$. (That is, find distinct x and y whose difference is at most the average difference between consecutive elements in the list containing the elements of X in sorted order.)

Answer:

Note that the list X is not sorted. The parenthetical remark gives you the meaning of the expression $(M - m)/(n - 1)$.
An $O(n^2)$ time algorithm is to consider all pairs and return the first one encountered with difference smaller than or equal to the average difference between consecutive elements.
An $O(n\ log\ n)$ time algorithm is to first sort the list and make a linear scan and return the first two consecutive numbers that have difference smaller than or equal to the average difference between consecutive elements.

Here is an $O(n)$ time algorithm. First we find the median of the n numbers in $O(n)$ time. Let the median have value μ. We partition the list into two parts, a list L that has all integers

that are at most the median, and a list H that has all integers that are at least the median. (We can put the median in both lists.) Then, the average gap for L is $(\mu - m)/(|L| - 1)$, and that for H is $(M - \mu)/(|H| - 1)$. We claim that the minimum of these two average gaps is at most $g = (M - m)/(n - 1)$. Otherwise, we have

$$(\mu - m) > (|L| - 1)g$$

$$(M - \mu) > (|H| - 1)g$$

Adding the above two, we have

$$(M - m) > (n - 1)g$$

a contradiction.

Since one of the two average gaps is at most the average gap of the whole list, we recurse on the sub-list that has minimum average gap. For the base case we consider a list with 2 elements, and return the pair. The running time is given by the recurrence: $T(n) = T(n/2) + O(n)$, which solves to $T(n) = O(n)$.

Problem 2. Out Tree of a Directed Graph

(a) Give an example of a directed graph $G = (V, E)$, two vertices $s, t \in V$, and a DFS run on the graph G such that:

1. G has five vertices, so $|V| = 5$.
2. For every vertex v in V , there exists a path in G from s to v.
3. In the DFS forest there is no path from s to t.

(By DFS forest, we refer to the set of trees obtained by running the DFS. Note that there are several possible runs of DFS, depending on how the vertices are chosen in the DFS. You need to provide one run that satisfies the desired property.)

Answer: A simple example is a cycle with 5 vertices: $s \to u \to v \to w \to t \to s$. Clearly, for every vertex v, s has a path to v. For the DFS run, if we start the DFS from u, we would obtain the DFS forest as the single tree $u \to v \to w \to t \to s$, satisfying the last requirement.

(b) An *out-tree* of a directed graph G is a rooted tree in which there is a path from the root to every vertex in G. Design an efficient algorithm to determine whether a given directed graph has an out-tree. State the worst-case running time of the algorithm, in terms of n and m, the number of vertices and the number of edges, respectively, of G.

Answer: Here are two algorithms.

Algorithm 1:

1. Run DFS.
2. Let C be last DFS component, and let v be the vertex in C from where the DFS was initiated (i.e., with the lowest discovery time).
3. Run DFS from v to see if every vertex is reachable from v. If yes, then return "YES", else return "NO".

Algorithm 2:

1. Find SCCs and DAG D on SCCs.

2. Compute topological sort on D. Let v be the first node of this topological sort.
3. Run DFS from v to see if every node in D is reachable from v. If yes, then return "YES" else return "NO".

Both the above algorithms take time $O(m+n)$.
A more ineffiicient algorithm is to run DFS from every vertex (separately) to check if any of them could possibly be a root of the desired out-tree.

Problem 3. MST, Shortest Paths, and Network Flow

(a) Let G be an undirected connected graph with distinct weights on the edges, and let T be a minimum spanning tree of G, and let T' be any spanning tree of G (not necessarily of minimum weight). Let e be the edge with maximum weight in T, and let e' be the edge with maximum weight in T'. Prove that the weight of e is at most the weight of e'.

Answer: Consider the cut formed in T when e is removed from T. Since T' is a spanning tree, it has at least one edge crossing the cut; let e_1 be one such edge. Then $w(e_1) \geq w(e)$ since if $w(e_1) < w(e)$, then the spanning tree $T - \{e\} \cup \{e_1\}$ has weight smaller than that of T, a contradiction. Since e' is the edge of maximum weight in T', we obtain $w(e') \geq w(e1) \geq w(e)$, yielding the desired claim.

(b) Let G be any connected undirected graph G with distinct positive weights on edges, and let T be the unique minimum spanning tree of G. **True or False?**: For every vertex u in G, there exists a vertex v such that the unique path from u to v in T has minimum weight among all paths from u to v in G. If your answer is True, then give a brief proof; otherwise, give a counterexample.

Answer: True.
Note that the statement is not about all u and v; it is about whether for all u there exists a certain v. Let u be any vertex of G. Let v be any adjacent node of u in T. The proof is by contradiction. If (u, v) is not the shortest path from u to v , then let P denote the shortest path u to v. Then, the cycle made by (u, v) and path P has (u, v) as its heaviest edge. By the cycle property, there is an MST that does not contain (u, v). Since the MST is unique, (u, v) should not be in MST, a contradiction.

(c) Consider a network flow (directed) graph G with a positive integer capacity on each edge, a source s, and a sink t. We say that an edge e is *crucial* if *decreasing* its capacity by one will *decrease* the maximum flow from s to t by one. Recall that an edge e is a *bottleneck* if *increasing* its capacity by one will *increase* the maximum flow from s to t by one.

True or False?: If an edge is crucial, then it is also a bottleneck. If you claim that the statement is true, give a brief proof; otherwise, give a counterexample.

Answer: False.
Let network be defined as $s \to a \to t$, with both edges having capacity 1. Both edges are crucial, but neither is a bottleneck.

Problem 4. Minimum bandwidth

Let us model a data network by a weighted directed graph G, in which the vertices of G represent the nodes of the network, the edges of G represent the data links of the network and the weight of

an edge is the communication capacity of the link (say, in Mbps). We define the bandwidth of a path p as the capacity of the minimum-capacity link in p.

Design and analyze an efficient algorithm to determine a path with the largest bandwidth from a given node s to a given node t. If no path from s to t exists, then your algorithm must indicate so. You may assume that the bandwidth of every link is positive. (Hint: Modify Dijkstra's algorithm.)

Answer: We can modify Dijkstra's algorithm by mainly replacing the "min" operation by "max", the "+" operation by "min", and the ">" operation by "<". These entail three kinds of changes in the algorithm.

First, consider the initialization routine. We will set $d[v]$ to 0 for all $v \neq s$ and we set $d[s]$ to 1. Second, the operation in which we explore a given edge (u, v) will change to the following code.

```
1 if d[v] < min{d[u], w(u,v)} then
2 |   d[v] = min{d[u], w(u,v)}
3 |   π[v] = u
4 end
```

Finally, in DIJKSTRA(G, w, s), instead of extracting the vertex with the minimum d value, we extract the vertex with the maximum d value.

Problem 5. Decomposing concatenated strings

A string x is said to be a *concatenation* of strings $y_1, y_2, \ldots, y_k$, for some k, if x is obtained by appending y_2 after y_1, then appending y_3 after y_2, and so on until appending y_k after y_{k-1}; that is x can be written as $y_1 y_2 y_3 \ldots y_k$.

Design an efficient algorithm that takes as input a set S of n binary strings and a string σ of m bits and determines whether σ is a concatenation of a subset of strings from S. Assume that each string in S has at most m bits. Your algorithm must run in time polynomial in m and n. State the worst-case running time of your algorithm.

Example: Suppose S is the set $\{1, 101, 111, 00, 100, 10, 110\}$. If σ is 10111100, then your algorithm should return "yes" since σ can be written as 101 + 111 + 00, where "+" stands for append or concatenation. Note that there are multiple ways that σ can be written as a concatenation of words in S; for instance, 10 + 111 + 100 and 10 + 1 + 1 + 1 + 1 + 00. If σ is 00000, then your algorithm should return "no" since σ cannot be written as any concatenation of the words in S.

Answer: Let $OPT[i]$ denote whether $\sigma[i \ldots n]$ is a concatenation of words in S. Let $s_1, \ldots, s_n$ be the strings in S. Let T denote true and F denote false. Below $\bigvee$ represents an OR.

$$OPT[i] = \begin{cases} \bigvee_{1 \leq k \leq n : s_k = \sigma[1 \ldots j-1} OPT[j], & \text{if there exists a string in } S \text{ which is a prefix of } \sigma[i \ldots n] \\ F & \text{otherwise} \end{cases} \tag{1}$$

The base case is $OPT[n+1] = T$, since $\sigma[n+1 \ldots n]$ is the empty string. Running time is $O(m^2 n)$ assuming substring determination takes time $O(m)$.

Problem 6. Task Rabbit

Task Rabbit is trying to allocate its rabbits among tasks on a given day. There are n types of tasks to complete on the day, with t_i tasks of type i, for $1 \leq i \leq n$. There are m rabbits, where rabbit j has the time to complete at most c_j tasks, and is capable of only completing tasks of types drawn from a subset $S_j \subset \{1, 2, \ldots, n\}$ of types.

Give a polynomial-time algorithm to determine if Task Rabbit can assign its rabbits to the tasks so as to complete all of them. If it is possible to do so, your algorithm should also return an assignment that indicates how many tasks of each type is executed by each rabbit. There is no need to prove the correctness of the algorithm or state a running time for your algorithm.

Answer: We form the following flow network.

We have a source s, a sink t, a set R of nodes consisting of one node for each rabbit, a set T of nodes consisting of one node for each task type. We have an edge (s, j) from s to rabbit node j of capacity c_j , for $1 \leq j \leq m$, and an edge (i, t) from task type node i to t of capacity t_i, for $1 \leq i \leq n$. We have an edge from rabbit node j to task type node i of capacity ∞ if $i \in S_j$.

The algorithm constructs the above network and computes a maximum flow f. If the value of f equals $\Sigma_{1 \leq i \leq n} t_i$, then all the tasks can be completed; otherwise, the algorithm returns "No". In the former case, the algorithm indicates that rabbit j does $f(j, i)$ tasks of type i, where $f(j, i)$ is the flow along edge from rabbit node j to task type node i.