Practice Quiz 1

- The following practice quiz is a compilation of relevant problems from previous semesters.
- This practice quiz may not be taken as a strict gauge for the difficulty level of the actual quiz.
- The quiz contains multiple problems, several with multiple parts. You have 120 minutes to complete this quiz.
- Write your solutions in the space provided. If you run out of space, continue on a scratch page and make a notation.
- Do not waste time deriving facts that we have studied. Just cite results from class.
- When we ask you to *give an algorithm* in this quiz, describe your algorithm in English or pseudocode, and provide a short argument for correctness and running time. You do not need to provide a diagram or example unless it helps make your explanation clearer.
- Do not spend too much time on any one problem.
- **Good Luck!**

Problem-1: T/F Questions

Answer *true* or *false* to each of the following. **No justification is required for your answer**. Space is provided for scratch work.

(a) The solution to the recurrence $T(n) = T\left(\frac{n}{6}\right) + T\left(\frac{7n}{9}\right) + O(n)$ is $O(n)$. You may assume $T(n) = 1$ for n smaller than some constant c.

Solution: True. Using the substitution method:

$$\begin{aligned} T(n) &\leq \frac{cn}{6} + \frac{7cn}{9} + an \\ &\leq \frac{17cn}{18} + an \\ &\leq cn - \left(\frac{cn}{18} - an\right) \end{aligned}$$

This holds if $\frac{c}{18} - a \geq 0$, so it holds for any constant c such that $c \geq 18a$. Full credit was also given for solutions that uses a recursion tree, noting that the total work at level i is $\frac{17}{18}$ in, which onverges to $O(n)$.

(b) Given a set of points $\{x_1, ..., x_n\}$, and a degree n bounded polynomial $P(x) = a_0 + a_1x + ... + a_{n-1}x_{n-1}$, we can evaluate $P(x)$ at $\{x_1, ..., x_n\}$ in $O(n \log n)$ time using the FFT algorithm presented in class.

Solution: False. Using FFT, we can efficiently evaluate a polynomial at the roots of unity, but not any n points.

(c) In a weighted connected graph $G = (V, E)$, each edge with the minimum weight belongs to some minimum spanning tree of G.

Solution: True. Consider a MST T . If it doesn't contain the minimum edge e, then by adding e to T , we get a cycle. By removing a different edge than e from the cycle, we

get a spanning tree T' whose total weight is no more than the weight of T. Thus, T' is a MST that contains e.

(d) For a flow network with an *integer* capacity on every edge, the *Ford-Fulkerson algorithm* runs in time $O((|V| + |E|)|f|)$, where $|f|$ is the *maximum* flow.

Solution: True. There may be $O(|f|)$ iterations as each iteration increases the flow by at least 1.

(e) If the primal Linear Program is feasible and has a bounded optimum, then the dual Linear Program is feasible.

Solution: True. By the *LP Duality Theorem*.

(f) In a weighted, connected graph G, if s is a starting node in *Prim's algorithm*, then for any other vertex v, the path on the resulting minimum spanning tree from s to v is the shortest path.

Solution: False. Consider graph G with vertices $\{s, v, w\}$ and edge weights $e(s, v) = 3$, $e(s, w) = 2$ and $e(w, v) = 2$. Path on the MST from s to v is of cost 4, while the shortest path is 3. There are many examples possible.

(g) Let $G = (V, E)$ and let H be the subgraph of G induced by some set of vertices $V' \subset V$ – i.e., $H = (V', E')$ where E' consists of all edges both of whose endpoints are in V'. Then every minimum spanning tree of H is a subgraph of some minimum spanning tree of G.

Solution: False. The subgraph induced by two vertices is a single edge and it is obvious that an arbitrary edge of G is not necessarily a subgraph of any minimum spanning tree of G.

(h) When performing competetive analysis on an *online* algorithm, we must know an *optimal, offline* algorithm relative to which we are assessing our *online* algorithm.

Solution: False. We do not need find an *optimal, offline* algorithm. Often we can prove what the optimal algorithm will have done over a sequence of operations and compare the *online* algorithm relative to that.

(i) Consider an implementation of the $ACCESS$ operation for self-organizing lists, which, whenever it accesses an element x of the list, transposes x with its predecessor in the list (if it exists). This heuristic is 2-competetive.

Solution: False. Suppose that we have a list of length n in which the last two elements are x and y, with y coming after x. Now consider a length-n sequence of $ACCESS$ operations: $[y, x, y, x, \ldots, y, x]$. It will have a total cost of $O(n^2)$.

Now consider if we had instead used the *MOVE-TO-FRONT* heuristic: both x and y would be moved to the beginning of the list after the first two accesses and each subsequent access would have cost $O(1)$, giving us a total cost of $O(n)$.

Since the *MOVE-TO-FRONT* heuristic outperforms the implementation under consideration by a factor of $O(n)$, this heuristic cannot be 2-competitive.

Problem-2: Well Spaced Triples

Suppose we are given a bit string $B[1 \ldots n]$. A triple of distinct indices $1 \leq i < j < k \leq n$ is called a *well-spaced triple* in B if $B[i] = B[j] = B[k] = 1$ and $k - j = j - i$.

(a) Describe an algorithm to determine whether B has a well-spaced triple in $O(n^2)$ time.
Solution: The algorithm is the following:

WELL-SPACED-TRIPLE($B[1 \ldots n]$):
  For each $i \in \{1, ..., n\}$
    For each $k \in \{1, ..., n\}$
      If $i + k$ is even AND $B[i] = 1$ AND $B[k] = 1$ AND $B[\frac{i+k}{2}] = 1$
        return yes
  return no

The running time of the above algorithm is $O(n^2)$, since we need to check all of the pairs of indices (i, k).

By the definition of well-spaced triples, B has a well-spaced triple if and only if there exists a pair (i, k) such that the following criterion are met:

i. $i + k$ is even.
ii. $B[i] = 1$.
iii. $B[k] = 1$.
iv. $B[\frac{i+k}{2}] = 1$.

Thus, the algorithm returns **yes** if and only if B has a well-spaced triple.

(b) Describe an algorithm to determine the number of well-spaced triples in B in $O(n \log n)$ time. (**Hint:** Use FFT)

Solution: The algorithm is as follows: Let $B[i]$ represent the i^{th} entry of the bit string B.

FAST-WELL-SPACED-TRIPLE($B[1 \ldots n]$):

Construct the polynomial $P(x) = \sum_i B[i]x^i$.
Use FFT to compute $Q(x) = P(x) \cdot P(x)$.
Let us denote the coefficients of $Q(x)$ by $(q_0, q_1, \ldots, q_{2n})$.
For each $j = 0, 1, ..., n$
  If $B[j] = 1$, $result = result + (q_{2j} - 1)$
return result/2

The running time of the above algorithm is $O(n \log n)$. We (a) use the *FFT* to multiply two polynomials of degree n in $O(n \log n)$ time and then (b) perform $O(n)$ comparisons and operations.

Since $P(x)$ has degree n, we know that $Q(x)$ has degree less than or equal to $2n$. The $2j$-th coefficient of $Q(x)$ can be written as

$$q_{2j} = \sum_{i+k=2j} B[i]B[k]$$

Thus, $q_{2j} \neq 0$ if and only if there exists (i, k) such that $B[i] = B[k] = 1$. Note that this condition does not guarantee that B has a well-spaced triple. That is because we have not enforced the condition $i < k$. In fact, if $B[j] = 1$, then $B[j]B[j] = 1$, which means that $q_{2j} \geq 1$.

However, if $q_{2j} > 1$, then there exists is a pair (i, k) such that i, j and k are distinct and $B[i] = B[k] = 1$. Furthermore, by symmetry, if (i, k) satisfies $B[i] = B[k] = 1$ then (k, i) does as well. So, the value $(q_{2j} - 1)/2$ gives us exactly how many distinct pairs (i, k) exist such that $i < k$ and $B[i] = B[k] = 1$. Note that we only want to consider such pairs if $B[j] = 1$, which is handled by the final if-statement in the code.

Problem-3: Minimum Spanning Trees

(a) In the graph depicted below, prove that:

i. The edge between B and E is not in *any* minimum spanning tree.

Solution: (B, E) is the heaviest edge in the cycle $B \to E \to D \to C \to B$, and cannot be in the MST by the cycle rule.

ii. In the graph shown in Figure 1, prove that the edge between E and G is in *every* minimum spanning tree.

Solution: (G, E) is the minimum weight edge across the cut $\{G\}$, and therefore must be in any MST by the cut rule.

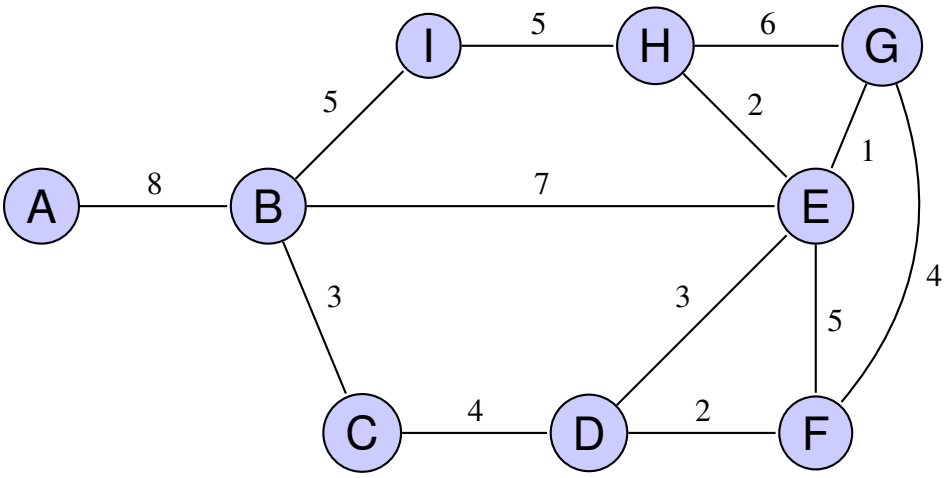

Figure 1: Example graph for part (a).

(b) Suppose that we are given a minimum spanning tree T of a graph $G = (V, E)$ that is connected, undirected, and has distinct positive integer weights (i.e. all edge weights are different positive integers). We want to design an algorithm that updates the minimum spanning tree when a new edge $e = (u, v)$ is added to E, for some $u, v \in V$.

Solution: We will first show that only the edges in the set $T'' = T \cup \{e\}$ can be in the minimum spanning tree of the modified graph $G' = (V, E \cup \{e\})$. Consider an edge $e' \in E \setminus T$. Since e' is not the MST of G, there is a cycle including it in G where it is the heaviest edge. Since we only added an edge to G to form G', this cycle still exists in G'. Therefore e' is not in the MST of G'.

The heaviest edge of the only cycle in T'' cannot be in the minimum spanning tree by the cycle rule. $T \cup \{e\}$ has n edges, removing this one edge from the cycle leaves $n-1$ edges that may be on the MST. Since the MST has at least $n-1$ edges, this is the tree.

Alyssa P. Hacker suggests the following algorithm: to compute the minimum spanning tree T' of graph $G' = (V, E \cup \{e\})$ from T, we **(1)** add edge e to T, and then **(2)** delete the heaviest edge in the cycle that is created in T with the addition of edge e.

Prove that this algorithm constructs a minimum spanning tree for G'. (i.e. give a proof of correctness.)

(c) After observing Alyssa's algorithm, Ben Bitdiddle wants to design an algorithm for updating the minimum spanning tree of a graph when new vertices are inserted. Specifically, Ben wants an algorithm that takes as input:

- A minimum spanning tree T of some undirected graph $G = (V, E)$,
- A new vertex, v'.
- A set of new (weighted) edges, E', between v' and V,

and computes a minimum spanning tree T' of the graph $G' = (V \cup \{v'\}, E \cup E')$.

Design an $O(V \log V)$ time algorithm for Ben's problem. As before, assume that all edge weights are distinct integers.

Solution: Part (c) shows that only the edges in $R = T_E \cup E'$ can be in the minimum spanning tree for $G' = (V \cup \{v'\}, E \cup E')$. Therefore the minimum spanning trees of the graphs

$$G'' = (T_V \cup \{v'\}, T_E \cup E')$$

and

$$G' = (V \cup \{v'\}, E \cup E')$$

are the same. Since G'' has $2V = O(V)$ edges, running *Kruskal's algorithm* on it then produces the MST in $O(V \log V)$ time.

Problem-4: 6.046 Carpool

The n people in your dorm want to carpool to 34-101 during the m days of 6.046. On day i, some subset S_i of people actually want to carpool (i.e., attend lecture), and the driver d_i must be selected from S_i. Each person j has a limited number of days ℓ_j they are willing to drive.

Give an algorithm to find a driver assignment $d_i \in S_i$ for each day i such that no person j has to drive more than their limit ℓ_j. (The algorithm should output "no" if there is no such assignment.)

For example, for the following input with $n = 3$ and $m = 3$, the algorithm could assign Penny to Day 1 and Day 2, and Leonard to Day 3.

Person	Day 1	Day 2	Day 3	Driving limit
1 (Penny)	X	X	X	2
2 (Leonard)	X		X	1
3 (Sheldon)		X	X	0

Hint: Use network-flow.

Solution: First, we create a graph with following vertices:

(a) a super source s and a super sink t

(b) vertex p_i for each person who wants to carpool

(c) vertex d_j for each day of the class.

Then create the following edges:

(a) s to p_i with capacity of ℓ_j

(b) p_i to d_j with capacity of 1 if person i needs to carpool on day j

(c) d_j to t with weight 1 for all j.

Finally, run max flow from s to t, and find f. If $|f| = m$, return that person i will drive on day j if the edge (p_i, d_j) has non-zero flow. If $|f| < m$, then return no valid assignment.

At a high level, the graph represents a matching between the driver and the days he/she will be driving. The capacity from s to p_i will ensure that no p_i drives more than l_i days (flow conservation). The non-zero flows from p_i to d_j means that p_i will drive on day d_j. The capacity of d_j to t will ensure that no more than 1 driver will be assigned to a particular day (flow conservation). If $|f| = m$, then all vertices d_j has an incoming flow, and therefore all d_j has a valid driver assignment. If $|f| < m$, then there is at least one d_j that has no incoming flow, and therefore does not have a driver assigned to it. By maximality of the flow, this means that there does not exist a valid driver assignment.

Ford-Fulkerson algorithm would run in $O(nm^2)$ since there are at most $nm + 2$ edges (bi-partite graph), and the max flow is bounded by m.

Problem-5: Candy Land.

The Candy Corporation of Candy Land owns 3 candy factories f_1, f_2, and f_3 that manufacture 20 kinds of candy $c_1, c_2, \ldots, c_{20}$. Each factory f_i manufactures m_{ij} pounds of each candy c_j. The Candy Corporation has 103 local stores $s_1, s_2, \ldots, s_{103}$ that sell candy. Each store s_k requires d_{kj} pounds of candy c_j. The factories only produce what is needed, and hence we have

$$\sum_{i=1}^{3}\sum_{j=1}^{20} m_{ij} = \sum_{k=1}^{103}\sum_{j=1}^{20} d_{kj}$$

A directed graph $G = (V, E)$ represents the road map of Candy Land, where V represents road intersections and E represents the roads themselves. Factories and stores are located at intersections: $f_i \in V$ for $i = 1, 2, 3$ and $s_k \in V$ for $k = 1, 2, \ldots, 103$. King Kandys tax collector, Lord Licorice, has designated a tax rate $c(e)$ for each road $e \in E$: if x pounds of candy (of any kind) are shipped along road e, Lord Licorice must be paid $x \cdot c(e)$ gumdrops.

Princess Lolly, the supply manager of the Candy Corporation, wants to determine how to deliver the candy from the factories to the local stores while minimizing the tax paid to Lord Licorice. Give a linear-programming formulation of this problem.

Solution: Let $x_j(u, v)$ be the amount of candy c_j delivered through the road $(u, v) \in E$ (for convenience, define $x_j(u, v) = 0$ if $(u, v) \notin E$). The delivery of candy is optimized by the linear program:

$$\text{minimize} \quad \sum_{(u,v)\in E} [t(u,v) \sum_{j=1}^{20} x_j(u,v)] \qquad \text{(total tax)}$$

$$\text{subject to} \quad \sum_{u\neq v} x_j(v,u) - x_j(u,v) = \begin{cases} m_{ij} & : \quad v \in \{f_i\} \quad \text{(supply constraints)} \\ -d_{kj} & : \quad v \in \{s_k\} \quad \text{(demand constraints)} \\ 0 & \quad \text{otherwise} \quad \text{(no buildup of candy)} \end{cases} \qquad \forall v \in V, j \in 1..20$$

$$x_j(u,v) \geq 0 \qquad \forall v \in V, j \in 1..20$$

Problem-6: Majority Element Suppose we have a list of n objects, where $n = 2^k$ for some positive integer k. The only operation that we can use on these objects is the following: $isEqual(i, j)$, which runs in constant time and returns True if and only if two objects i and j are equal. *The objects are neither hashable nor comparable.*

Our goal is to find if there exists a majority (more than $n/2$ occurrences) of the same element and if so, return it.

Design an $O(n)$ time algorithm that finds whether a majority element exists, and if so, returns it. Provide arguments for your algorithm's correctness and runtime.

Hint: Consider implementing and using a subroutine $Reduce(L)$ which converts the list L into an $n/2$ sized list of each adjacent pairs.

For example, consider the list:

$$L = [\ddagger, \oslash, \star, \ddagger, \sqcap, \sqcap, \ddagger, \ddagger, \ddagger, \oslash, \ddagger, \bigcirc, \ddagger, \uplus, \ddagger, \ddagger]$$

A single iteration of $Reduce(L)$ would create a list L' as follows:

$$L' = [(\ddagger, \oslash), (\star, \ddagger), (\sqcap, \sqcap), (\ddagger, \ddagger), (\ddagger, \oslash), (\ddagger, \bigcirc), (\ddagger, \uplus), (\ddagger, \ddagger)]$$

Solution: We want to use divide and conquer to solve this problem - given our list L, we want to create a list M where if there is a majority element $x \in L$, x will be a majority element in M.

For a linear time algorithm, we can do the following:

We assume that our original list L has an even number of elements. If it doesn't (which might occur in some subproblems), we remove the last element and check if it's the majority. If it is, then we return that element. If not, we can remove the element from our list to get a list with an even number of elements, without affecting the majority. This can be done in $O(n)$ time.

Run the operation $Reduce(L)$, which reduces our size n list to an $n/2$ list of object pairs L'.

For a given object pair (i, j), we run $isEqual(i, j)$. If $i = j$, then we include i in our new list M. If not, we discard both elements.

Since we discard at least half the elements in L, M has size $\leq \frac{n}{2}$.

We keep recursing until we reach a list of size at most 2, where we can find the majority in constant time. If the list is of size one, then we return the single element. If the list is of size 2, we return the first element iff the two elements in the list are equal. If not, we return None, since there is no majority.

If we find that M (our subproblem) returns a candidate majority element y, in linear time, we scan L to see if it's truly a majority element in L. If so, we return y - if not, we return None.

Since M has size at most $n/2$, our recurrence becomes:

$$T(n) = T\left(\frac{n}{2}\right) + O(n)$$

which is O(n) time.

Correctness: To show that our algorithm is correct, we have to show that if there is a majority element $x \in L$, x is a majority element in M. To do this, we look at the subroutine $Reduce$.

$Reduce$ creates object pairs (i, j). If $i \neq j$, we discard both i and j. Let's assume there are p such elements which form pairs where $i \neq j$. At most half of these elements can be x since for any given pair (i, j), $i \neq j$. Thus, we discard at most $p/2$ occurrences of x.

For x to be a majority element, there must be $\frac{n}{2} + c$ occurrences of it, for some $c > 0$. If we lose $\frac{p}{2}$ of these occurrences, there are still $\frac{n}{2} + c - \frac{p}{2}$ occurrences of x in tuples where $i = j$. Since there are $n - p$ elements which make up tuples where $i = j$, there are still $\frac{n+2c-p}{2}$ occurrences of x in this sublist and x remains a majority.

Since M has size at most $\frac{n}{2}$, x will still be a majority element in M.

Problem-7: Amortized Analysis.

Design a data structure to maintain a set S of n distinct integers that supports the following two operations:

(a) INSERT(x, S): insert integer x into S.

(b) REMOVE-BOTTOM-HALF(S): remove the smallest $\lceil \frac{n}{2} \rceil$ integers from S.

Describe your algorithm and give the worse-case time complexity of the two operations. Then carry out an amortized analysis to make INSERT(x, S) run in amortized $O(1)$ time, and REMOVE-BOTTOM-HALF(S) run in amortized 0 time.

Solution: Use a singly linked list to store the integers. To implement INSERT(x, S), we append the new integer to the end of the linked list. This takes $\Theta(1)$ time.

To implement REMOVE-BOTTOM-HALF(S), we use the median finding algorithm taught in class to find the median number, and then go through the list again to delete all the numbers smaller or equal than the median. This takes $\Theta(n)$ time.

Amortized Analysis: We will use the potential method. Suppose the runtime of REMOVE-BOTTOM-HALF(S) is bounded by cn for some constant c. If we use $\Phi = 2cn$ as our potential function, then the amortized cost of an insertion is:

$$1 + \Delta\Phi = 1 + 2c = \Theta(1)$$

and the amortized cost of REMOVE-BOTTOM-HALF(S) is:

$$cn + \Delta\Phi = cn + \left(-2c \times \frac{n}{2}\right) = 0$$