Alex Havrilla

Hwk 1

Task 1:

---

a)

Choose $c(n) = \lceil log_2(n) \rceil + 1$. Since all array elements are distinct we know the median will be the unique element with n elements larger than it(or n-1 smaller). Consider the following algorithm. Let a_1 be the median of array A and b_1 the median of array B(which we can access in O(1) time since they are sorted). Compare these. If $n = 1$ return the smaller. Now assume $n > 1$ and without loss of generality suppose $a_1 < b_1$. If this is not case we simply reverse the roles of A and B in the following. We know a_1 is less than at least $\lfloor n/2 \rfloor + 1 + \lfloor n/2 \rfloor$ elements. Then we know all the elements less than a_1 in A cannot be the median because they are less than at least $1 + 1 + \lfloor n/2 \rfloor + \lfloor n/2 \rfloor > n$. Similarly we know b_1 is greater than at least $n - \lfloor n/2 \rfloor - 1 + n - \lfloor n/2 \rfloor - 1 + 1 = 2n - 2\lfloor n/2 \rfloor - 1$ so all the elements greater than it in B are greater than at least $2n - 2\lfloor n/2 \rfloor > n - 1$ elements, ie. they cannot be the median. Recursively apply this algorithm to the subarrays $A[n - \lfloor n/2 \rfloor - 1, n - 1]$ and $B[0, n - \lfloor n/2 \rfloor - 1]$. Note if n is even a_1 is less than $n + 1$ elements, ie. cannot be the median, so we instead consider the range $A[n - \lfloor n/2 \rfloor, n - 1]$. So in both cases recurse on subarrays of equal size(of around half the input size). Thus we expect a logarithmic number of steps, ie. comparisons. Formally we prove $c(n)$ correct via induction. Clearly the base case is correct($log_2(1) + 1 = 1$). Then if we consider inputs of size n, we know the recursive case will cost $\lceil log_2(2(n - \lfloor n/2 \rfloor)) \rceil = log_2(n)$. Then one comparison at the top level gives $\lceil log_2(n) \rceil + 1$.

Note that this must correctly find the median, as the median of the subarrays must be the median of the top level input(as we discard an equal number of elements on the "tails"). Formally we can show this inductively. Clealry the base case is correct. Then if we consider some arbitrary array inputs of size n, we wlog half of the small terms from A and the large

terms from B and recurse, ie. apply the inductive hypothesis. The returned median will be the true median since the number of small terms dropped is the same as the number of larger terms dropped at the top level.

b)

We use an information theoretic argument to show a lower bound. Fix $n \in \mathbb{N}$. We know there are 2n possible outputs for an arbitrary algorithm solving this(any of the indices in A or B). Furthermore for ith output of 2n outputs we can find arrays A and B whose joint median is at the ith index(starting with A[0] the first output and B[n-1] the last output) since the median is unique. Wlog suppose the median is in A(we do the symmetric thing in the case it is in B). Fix n at $A[i-1]$. Then pick $i-1$ numbers less than n and place these ordered in A. Pick $n-i$ numbers larger than n and place these orderd in A. Then for B pick i numbers larger than n and $n-i$ numbers smaller than n and order these in B. Then we know in total there are $n-i+i=n$ numbers larger than n in total, ie. n at index i is the median. Since we know distinct inputs exist for each possible output, we may apply a lower bound of $\lceil log_2(2n) \rceil = \lceil log_2(2n) \rceil + 1$.

Task 2:

We analyze the expeced running time of the QuickSelect variant.

We case on events. We know $L = k-1$ with probability $1/n$. $L < n/3$ or $L > 2n/3$ with probability $2/3$ since they are disjoint events(and since the probabililty of each alone is $1/3$ via uniformity of the pivot). Finally the probability $n/3 \leq L \leq 2n/3$ is $1/3$ via uniformity of the pivot. Then we may bound the expected running time $ET(n)$ as

$\mathbb{E}T(n) \leq P(L = k-1)T(1) + P(L < n/3 \vee L > 2n/3)\mathbb{E}T(n) + P(n/3 \leq L \leq 2n/3)\mathbb{E}T(2n/3) + O(n) = T(1)/n + \frac{2}{3}\mathbb{E}T(n) + \frac{1}{3}\mathbb{E}T(2n/3) + O(n)$ via the following argument. If we get $L = k-1$ we terminate immediately. Otherwise if we get L too large or too small we repeat by picking a new pivot with probability $P(n/3 < L \vee L > 2n/3)$. Otherwise we know we picked a

"good" pivot and both L and its complement are at most $2n/3$ in size and we recurse on the appropriate one. $O(n)$ work comes from comparing each element to the pivot. Notice the $T(1)/n$ term is abosrbed by $O(n)$.

This expression simplifies to

$$\frac{1}{3}\mathbb{E}T(n) \leq \frac{1}{3}\mathbb{E}T(2n/3) + O(n) \implies \mathbb{E}T(n) \leq \mathbb{E}T(2n/3) + 3O(n) \implies$$

$$\mathbb{E}T(n) \leq \mathbb{E}T(2n/3) + O(n)$$

which solves to $\mathbb{E}T(n) \in O(n)$ via brick-method.

Task 3:

---

a)

Claim: $\binom{n}{2}$ queries are necessary in the worst case to test whether G contains a cycle.

Proof. Fix $n \in \mathbb{N}$. Heuristically, we aim to construct a tree or almost-tree(a tree with one additional edge). Consider some algorithm A determining cyclicness. When A queries $G[u, v]$ answer 1 without introducing a cycle until n-1 edges are revealed. Then anwer 0 for the rest of the queries. Then for any unqueried edge (u,v) there could exist a cycle containing u,v, and other vertices. Note that there are $\binom{n}{3}$ possible 3 cycles in G and to check all of them A would need to query at least $3 * \binom{n}{3}/(n-2) > n-1$ since in order to check each triangle A would need to check each of 3 edges in all of triangles(we divide by n-2 since revealing one edge is missing in a triangle eliminates n-2 triangles simultaneously). Thus if A returns an answer before we can reveal n-1 edges, it means it has not checked every triangle. If A returns cyclic, we reveal all other edges as 0. Otherwise we reveal a triangle as a cycle. Suppose A does check enough edges to reveal n-1 edges. Then we know acyclicness the revealed graph is connected. Thus if A does not check the remaining edges, we may reveal one as existing and produce a cycle. Thus A must query all edges.

□

b)

Claim: At least $\binom{n}{2} - n + 1$ are necessary in worst case to test whether a graph G is bipartite.

Proof. Fix $n \in \mathbb{N}$. Heuristically, we aim to construct a bipartite graph or almost bipartite graph with one vertex in a partion and n-1 in the other. Consider some algorithm A. Pick a vertex v. When A queries G[u,v] return 1(the edge exist), otherwise return 0. Then for an unqueried pair (u,w), with neither equal to v, A cannot know if they are in the same partition. If A does not query some specific (u,w) and returns bipartite, we reveal this edge as existing, so the graph cannot be bipartite as we have a triangle (u,w,v). Otherwise if it returns not bipartite we reveal all the edges not incident to v as missing. Then the parition with n-1 vertices not v and 1 vertex v works to show bipartiteness. Note that there are $\binom{n}{2} - (n-1)$ edges to check not incident to v. Hence at least $\binom{n}{2} - (n-1)$ checks are required.

□