CS 161: Final Exam — Solution Sketches

Question 1

Part (a)

There are three cases depending on the value of k that correspond to the three different cases in the master theorem. Using the version of the master theorem where $f(n) = n^d$, we have $a = 2, b = 2, d = k$.

1. If $0 < k < 1$, then $a = 2 > 2^k = b^d$. Hence, $T(n) = O(n)$.
2. If $k = 1$, then $a = 2 = b^d$. Hence, $T(n) = O(n \log n)$.
3. If $k > 1$, then $a = 2 < 2^k = b^d$. Hence, $T(n) = O(n^k)$.

Part (b)

Assume $n = 2^k$ and consider the recursion tree. We assume that the base case is $T(2) = 1$.

At level 0, we have one problem of size 2^k, leading to work $\frac{2^k}{\log 2^k} = \frac{2^k}{k}$.

At level 1, we have two problem of size 2^{k-1}, leading to total work of $2 \cdot \frac{2^{k-1}}{k-1} = \frac{2^k}{k-1}$.

In general, at level j, we have 2^j problems of size 2^{k-j}, leading to total work of $2^j \cdot \frac{2^{k-j}}{k-j} = \frac{2^k}{k-j}$.

At the last level k, we have 2^k problems of size 1, leading to a total work of 2^k (since $T(1) = 1$).

Summing over the $k + 1$ levels, we get that the total work is

$$\sum_{j=0}^{k-1} \frac{2^k}{k-j} + 2^k = 2^k \left(1 + \sum_{j=1}^{k} \frac{1}{j}\right) = O(2^k \log k).$$

Recall $\sum_{i=k}^{k} \frac{1}{j} = O(\log k)$ was proven in homework 1. Since $k = \log n$, we conclude that $T(n) = O(n \log \log n)$.

Question 2

True. We prove by induction on n.

Base case: If $n = 1$, then the tree contains only one node, and there is only one way to assign the value to the node.

Inductive step: Assume that for all trees with at most n nodes, there is only one way to assign a set of values to the nodes. Consider a tree with $n + 1$ nodes. Denote by $(a_1, \ldots, a_{n+1})$ the set of keys in sorted order $(a_i < a_{i+1})$.

Let r denote the root. Denote by n_l the number of nodes in the left subtree of the root. Since the nodes in the left subtree must store a key that is less than the root and the nodes in the right subtree must have keys that are greater than the root, we conclude that the root must store the key a_{n_l+1}.

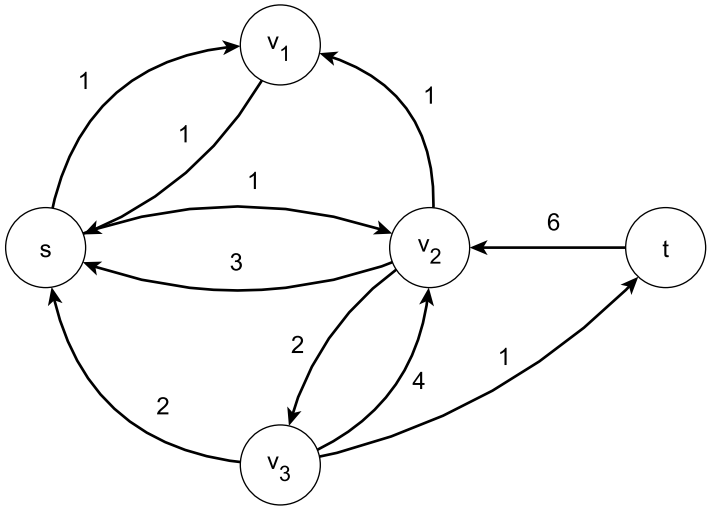
In addition, the left subtree of the root must contain the keys $\{a_1, \ldots, a_{n_l}\}$, and the right subtree must contain the keys $\{a_{n_l+2}, \ldots, a_{n+1}\}$. Each of these subtrees is a binary search tree with at most n nodes,

and from the induction hypothesis, we get that there is only one way to assign the values to the nodes. We conclude that there is only one way to assign the $n+1$ values to the nodes of the tree in a way that will result in a valid binary search tree.

Question 3

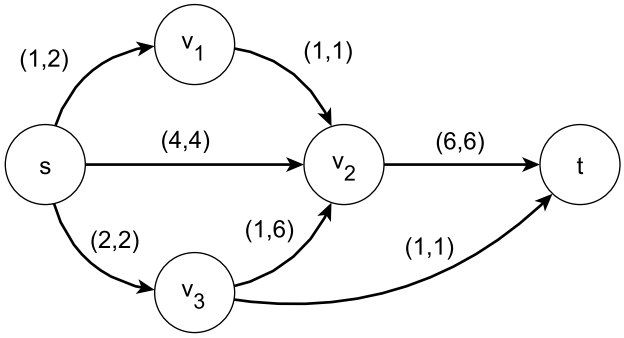
Part (a)

The residual network (edges with residual capacity 0 are omitted):



Part (b)

The augmenting path is (s, v_2, v_3, t). The resulting flow is:



Part (c)

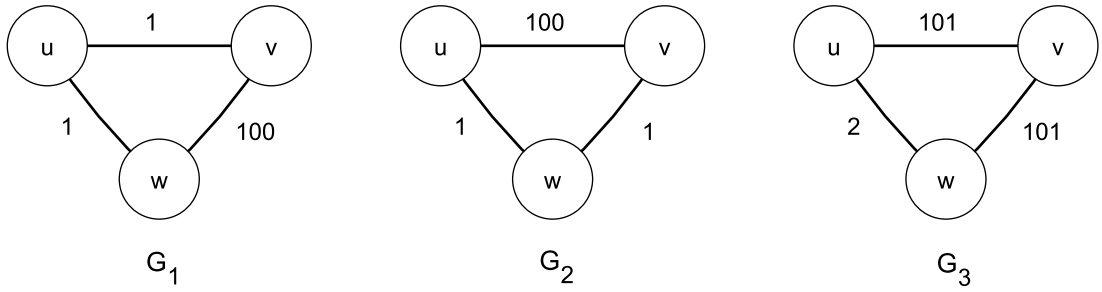
One possible minimum s-t cut in the graph is $S = \{s, v_1\}, V \setminus S = \{v_2, v_3, t\}$. There are other minimum s-t cuts.

The value of the cut is 7, which is also the value of the flow described in part (b). From the max-flow min-cut theorem, we conclude that this cut has minimum value.

Question 4

(a) $cost(T_3) = \sum_{e \in E(T_3)} w_3(e) = \sum_{e \in E(T_3)} (w_1(e) + w_2(e)) = \sum_{e \in E(T_3)} w_1(e) + \sum_{e \in E(T_3)} w_2(e) = cost(S_1) + cost(S_2)$ for some spanning trees S_1 and S_2 of G_1 and G_2, respectively. Because $cost(T_1) \leq cost(S_1)$ and $cost(T_2) \leq cost(S_2)$, $cost(T_1) + cost(T_2) \leq cost(S_1) + cost(S_2)$. So, $cost(T_1) + cost(T_2) \leq cost(T_3)$.

(b) Consider the following examples:



T_1 includes the edges $(u, v), (u, w)$ and costs 2. T_2 includes the edges $(u, w), (w, v)$ and costs 2. T_3 includes the edges $(u, v), (u, w)$ and costs 103. Hence, $cost(T_1) + cost(T_2) < cost(T_3)$.

Question 5

Algorithm: Sort the red and blue points separately.
Let's renumber the points so that $r_1 < r_2 < \ldots r_n$ and $b_1 < b_2 < \ldots b_n$. Now match r_i to b_i.

Running Time: Sorting the two sets of points takes $O(n \log n)$ time. Producing the matching takes $O(n)$ time (actually $O(1)$ time given the sorted orders, since we don't need to do any additional work to produce the matching).

Correctness: We will assume that points are renumbered in sorted order. We will prove that the greedy matching that matches r_i to b_i minimizes the cost of the matching.

Lemma 1: There is an optimal matching that matches r_1 to b_1.
Proof: Without loss of generality, assume that $r_1 \leq b_1$ (if not, swap the red and blue points). Consider an optimal matching M. If r_1 is matched to b_1 in M then we are done. Suppose this is not the case. We will show that we can modify M to obtain a different optimal matching that matches r_1 to b_1. Suppose M matches r_1 to b_j and r_i to b_1. Consider the alternate matching M' where r_1 is matched to b_1, r_i is matched to b_j. All points other than r_1, b_1, r_i, b_j are matched identically in M and M'. We will prove that the cost of M' is at most the cost of M. Hence M' must also be an optimal matching, and r_1 is matched to b_1 as required.

To show that the cost of M' is at most the cost of M, we need to show that
$|r_1 - b_1| + |r_i - b_j| \leq |r_1 - b_j| + |r_i - b_1|$ (*)
We prove this by considering various cases for the configuration of these points on the line. We know that $r_1 < r_i$ and $b_1 < b_j$. Also, by assumption, $r_1 \leq b_1$ Hence $r_1 \leq b_1 < b_j$. There are 3 cases for the possible location of r_i:
Case 1: $r_1 < r_i \leq b_1 < b_j$
Suppose that $r_i - r_1 = x > 0$, $b_1 - r_i = y \geq 0$ and $b_j - b_1 = z > 0$.
Then $|r_1 - b_1| = x + y$, $|r_i - b_j| = y + z$, $|r_1 - b_j| = x + y + z$, $|r_i - b_1| = y$. Hence the inequality (*) holds.

Case 2: $r_1 \leq b_1 < r_i \leq b_j$
Suppose that $b_1 - r_1 = x \geq 0$, $r_i - b_1 = y > 0$ and $b_j - r_i = z \geq 0$.
Then $|r_1 - b_1| = x$, $|r_i - b_j| = z$, $|r_1 - b_j| = x + y + z$, $|r_i - b_1| = y$. Hence the inequality (*) holds.
Case 3: $r_1 \leq b_1 < b_j < r_i$
Suppose that $b_1 - r_1 = x \geq 0$, $b_j - b_1 = y > 0$ and $r_i - b_j = z > 0$.
Then $|r_1 - b_1| = x$, $|r_i - b_j| = z$, $|r_1 - b_j| = x + y$, $|r_i - b_1| = y + z$. Hence the inequality (*) holds.

This concludes the proof of Lemma 1.

Lemma 2: There is an optimal matching that matches r_i to b_i for all $i = 1, \ldots n$.
Proof: We will prove by induction that there is an optimal matching that matches r_i to b_i for all $1 \leq i \leq j$.

Base Case: Follows from Lemma 1.
Inductive Step: By the inductive hypothesis there is an optimal matching M that matches r_i to b_i for all $1 \leq i \leq j$. Note that the cost of the matching is $\sum_{i=1}^{j} |r_i - b_i|$ plus the cost of the matched pairs on the remaining points. If we remove the points $r_1, \ldots r_j$ and $b_1, \ldots b_j$, then M restricted to the remaining points $R_j = \{r_{j+1}, \ldots r_n\}$ and $B_j = \{b_{j+1}, \ldots b_n\}$ does indeed give a valid red-blue matching on R_j and B_j. This R_j-B_j matching must be an optimal matching.

Let M' refer to the partial matching that matches r_i to b_i for all $1 \leq i \leq j$. Any R_j-B_j matching M_j can be combined with M' to produce a valid matching on the entire set of points. The cost this matching is the sum of the costs of M' and M_j. Hence any optimal R_j-B_j matching can be combined with M' to produce an optimal matching on the entire set of points.

By Lemma 1, there is an optimal R_j-B_j matching M_j that matches r_{j+1} to b_{j+1}. Combining this with M' gives an optimal matching on the entire set of points that matches r_i to b_i for all $1 \leq i \leq j + 1$.

Note: This proof can shortened in the following way. Let M be an optimal matching such that i is the lowest index such that r_i is not matched to b_i. Then, r_i is matched to some b_j where $j > i$, and b_i is matched to r_k where $k > i$. Using similar arguments to those applied in the proof of Lemma 1, we can show that if we match r_i to b_i and r_k to b_j (and all other points are matched as in M), we still get an optimal matching. We can continue applying this until we get that the matching where r_i is matched b_i for all i is optimal.

Question 6

For $1 \leq i \leq n$, $1 \leq j \leq \min(k, i)$, let $C(i, j)$ denote the minimum cost for clustering $x_1, \ldots x_i$ into exactly j clusters. In the optimum clustering of $x_1, \ldots x_i$ into j clusters, suppose that the minimum point in the cluster that includes x_i is x_a. We define $s(i, j)$ to be this index a.

For convenience, we define $C(0, 0) = 0$.

We compute the values of $C(i, j)$ for $i \geq 1$, $1 \leq j \leq \min(k, i)$ as follows:

$$C(i, j) = \min_{j \leq a \leq i} \{c(a - 1, j - 1) + (x_i - x_a)^2\}$$

$$s(i, j) = \arg\min_{j \leq a \leq i} \{c(a - 1, j - 1) + (x_i - x_a)^2\}$$

We apply this for i increasing from 1 to n. For each value of i, we compute all entries $C(i, j)$ before proceeding to $i + 1$.

The final value reported is $C(n, k)$, i.e., the cost of clustering all n points into k clusters. The optimum clustering can be reported by backtracking, using the values $s(i, j)$ as described by this pseudocode:

```
OutputClustering(n,k) {
    If k = 0 return;
    OutputClustering(s(n,k)-1,k-1);
    Output "S_k = {", s(n,k), s(n,k)+1, ..., n, "}";
}
```

Running Time: Each entry of the table takes $O(n)$ time to compute. There are $O(nk)$ entries, so overall running time is $O(n^2k)$.

Correctness: We claim that this dynamic program correctly computes the costs $C(i,j)$. We need to show that the recurrence relation $C(i,j)$ is computing the minimum cost of clustering points $x_1, \ldots, x_i$ into $j \leq i$ clusters. If in the optimum clustering, the last cluster j contains the points $x_a, \ldots, x_i$, the cost of the optimal solution is the cost of the optimum clustering of $x_1, \ldots, x_{a-1}$ into $j-1$ clusters, and the cost of the last cluster $(x_i - x_a)^2$. The recurrence considers all possible values of a and picks the one that results in the lowest total cost (every value of a that the recurrence considers corresponds to the cost of a clustering, so the minimum cannot be lower than the optimum clustering). In addition, we compute the values in increasing order of i, j in a way that ensures that we never access the cost $C(a-1, j-1)$ of a subproblem before it has been computed.

Question 7

The main observation is that the most difficult up-and-down path should consist of a shortest "ascending" path from a to the node, say v, of the max height on p and then a shortest "descending" path from v to b. Consider the following algorithm:

Algorithm 1: FindPath$(G(V,E), w, h)$

1. Construct a directed graph $G' = (V, E')$ where for each $e \in E$ where $e = (u, v)$, there is a directed edge $e' = (u, v)$ in E' only if $h(u) < h(v)$.
2. Run Dijkstra's algorithm to find single-source shortest paths from a on G'. Let d_1 and π_1 be the shortest path distances and predecessors.
3. Run Dijkstra's algorithm to find single-source shortest paths from b on G'. Let d_2 and π_2 be the shortest path distances and predecessors.
4. If $d_1(v) = \infty$ or $d_2(v) = \infty$ for all $v \in V \setminus \{a, b\}$
5. Return "No up-and-down path from a to b"
6. Else
7. Let $v' = \arg\max_{v \in V \setminus \{a,b\}} \frac{h(v)}{d_1(v)+d_2(v)}$
8. Construct the shortest paths p_1 and p_2 to v' using π_1 and π_2, respectively.
9. Return $p_1 \cup \text{reverse}(p_2)$

Correctness: Assume an up-and-down path from a to b exists and let p^* be the most difficult up-and-down path among such up-and-down paths. We show that the algorithm FindPath returns an up-and-down path with difficulty at least difficulty(p^*). By the definition of p^*, it would follow that the returned path has difficulty exactly equal to difficulty(p^*).

We note that p^* consists of a shortest "ascending" path p^+ from a to the node, say v^*, of the max height on p^* and a shortest "descending" path p^- from v^* to b, where ascending (descending) paths are on edges in the direction of increasing (decreasing) node heights. Otherwise, we can improve the difficulty of p^* by replacing p^+ or p^- with a shorter path.

By the construction of G' in Step 1 and Dijkstra's algorithm in Steps 2 and 3, we compute shortest ascending paths from a to v^* and from b to v^*. Reversing the direction of the shortest ascending path from b to v^*, we get a shortest descending path from v^* to b. It follows that FindPath finds an up-and-down path through v^* with the same difficulty as p^*. Then, FindPath returns an up-and-down path with difficulty at least difficulty(p^*) in Steps 7 - 9.

If no up-and-down path exists from a to b, it must be that for any node v, either no ascending path from a to v exists or no ascending path from b to v exists. Equivalently, $d_1(v) = \infty$ or $d_2(v) = \infty$ for all $v \in V \setminus \{a, b\}$. The algorithm reports correctly that there is no up-and-down path in this case.

Runtime: The construction of G' takes $O(|E| + |V|)$ time in Step 1. Steps 2 and 3 take $O(|E| + |V| \log |V|)$ time, which is the running time of Dijkstra's algorithm with the Fibonacci heap implementation for the priority queue. Steps 4-9 can be completed in $O(|V|)$ time. Overall, FindPath has running time $O(|E| + |V| \log |V|)$.