# CS300 Algorithms
# 2008 Fall semester Mid-term Solution

## 1

1.1 Since our algebraic decision tree $T$ is linear, (i.e. $f(x_1, x_2, \cdots, x_n)$ for all nodes $i$ is linear), $L_r$ node of $T$ divides the space into two convex spaces (ex. $f > 0$, $f \leq 0$). Let $S_j$ be the selected space from $j^{th}$ $L_r$ node which you pass through until the leaf node $l_i$. Then,

$$D_i = S_1 \cap S_2 \cap \cdots \cap S_d,$$

where $d$ is the depth of $T$.

Since each $S_j$ is convex and the intersection of a finite number of convex sets is also convex, $D_i$ is also convex.

1.2 We show that if $Y(w_i)$, $1 \leq i \leq p$, are distinct then $||W|| > ||L||$: We know that if an algorithm $\mathcal{A}$ represented by a linear decision tree solves a problem $\mathcal{D}$ then for all $i$, $w_i \cap D_j \neq \emptyset$ for some $1 \leq j \leq d$. Therefore, by the pigeonhole principal, it holds.

1.3 Suppose that $Y(w_i)$, $1 \leq i \leq p$, are not distinct. Then, $Y(w_i) = Y(w_j) = h$, for $1 \leq i < j \leq p$, $1 \leq h \leq d$. By definition,

$$Y(w_i) = h \Rightarrow w_i \cap D_h \neq \emptyset.$$

$$Y(w_j) = h \Rightarrow w_j \cap D_h \neq \emptyset.$$

Take two points $q_1, q_2 \in D_h$ such that $q_1 \in w_i \cap D_h$ and $q_2 \in w_j \cap D_h$. Since $D_h$ is convex, the line segment joining $q_1$ and $q_2$ must be contained in $D_h$. However, $w_i \cap w_j = \emptyset$ since $w_i$ are disjoint connected components. Therefore, the line segment joining $q_1$ and $q_2$ exits $w_i$ and enters $w_j$ again, which means that first it gives an answer "yes", then "no", and "yes" again. But since the segment is contained in a domain $D_h$, it is supposed to give only one answer. A contradiction!

1.4 Because $||L|| \geq ||W||$ from 1.2 and 1.3, the time complexity for solving $\pi$ of the input size $n$ is

$$T(n) \geq c \log_2 ||L|| \geq c \log_2 ||W||.$$

Therefore, $T(n) = \Omega(\log_2 ||W||)$.

## 2

2.1 Since $N$ elements are all distinct, the space of input is divided by $N!$ subsets. Each subset represents a disjoint connected component of $W$ which eventually leads to "halt and yes, all the elements are unique". Therefore, the number of disjoint connected components of $W$ is $N!$. i.e., $|W| = N!$.

2.2 A lower bound is determined by $|W|$, i.e.,

$$c \log_2 |W| = c \log_2 N! \in \Omega(N \log N).$$

This bound is tight (i.e. $\Theta(N \log N)$) because there is an algorithm to solve EUP in $O(N \log N)$; we first sort $N$ elements in $O(N \log N)$ and then compare whether consecutive elements are equal in $O(N)$.

2.3 First, we transform the input of $EUP$, $I_{EUP} = \{x_1, x_2, \cdots, x_N\}$, into the one of $2CPP$, $I_{2CPP} = \{(x_1, 0), (x_2, 0), \cdots, (x_N, 0)\}$, in $O(N)$. Then compute the solution for $2CPP$. If the solution of $2CPP$ is a pair with distance 0, then we can answer "No, they are not unique" to $EUP$, otherwise, "yes, they are unique."

$$\therefore EUP \propto_{O(N)} 2CPP.$$

Therefore, the lower bound for solving $2CPP$ is

$$L_{2CPP} = \Omega(N \log N) - O(N) = \Omega(N \log N).$$

## 3

3.1 If you look at the first element only, it is already sorted. Therefore, you can assume that when the $i^{th}$ element is examined, the first $i-1$ elements have already been sorted. Thus, to find a correct position of $i^{th}$ element, we need to compare it with $i-1$ elements in the worst case. For $n$ elements with $k$ inversions, the number of comparisons and swapping is bounded by $O(k) + (n-1)$. (You need $n-1$ comparisons because even though the elements are already sorted, you need to compare $i^{th}$ element with the previous $(i-1)^{th}$ element.)

3.2 By assumption, the probability of the $i^{th}$ element is in the $j^{th}$ position is $1/i$, where $j = 1, 2, \cdots, i$. Let $A_i(n)$ be the average number of comparisons for the $i^{th}$ element and $A(n)$ be the sum of average numbers of comparisons for all $i$ (i.e. $A(n) = \sum_{i=1}^{n} A(i)$).

$$A_i(n) = \begin{cases} 0 & \text{if } i = 1 \\ \left(\sum_{j=1}^{i-1} \frac{1}{i} \times j\right) + \left(\frac{i-1}{i}\right) = \frac{i+1}{2} - \frac{1}{i}. & \text{otherwise} \end{cases}$$

Therefore,

$$A(n) = \sum_{i=2}^{n} \left(\frac{i+1}{2} - \frac{1}{i}\right) = \frac{n^2}{4} + \frac{3n}{4} - 1 - \sum_{i=1}^{n} \frac{1}{i} \cong \frac{n^2}{4} \in O(n^2),$$

since $\sum_{i=1}^{n} \frac{1}{i} \in O(\ln n)$.

3.3 To show that the insertion sort is optimal in the average case, we need to compute the average number of inversions. Remind that an inversion of the permutation $\Pi$ is an ordered pair $(\Pi(i), \Pi(j))$ such that $i < j$ and $\Pi(i) > \Pi(j)$.

First note that we have $n!$ possible permutations with given $n$ elements. For each $(i, j)$, $(\Pi(i), \Pi(j))$ is an inversion in either $\Pi_k = (\Pi_k(1), \Pi_k(2), \cdots, \Pi_k(n))$ or a transpose of $\Pi_k$, which we denote by $\Pi_k^T = (\Pi_k(n), \Pi_k(n-1), \cdots, \Pi_k(1))$. Among $n!$ permutations, the number of $(\Pi_k, \Pi_k^T)$ pairs is $n!/2$. Since we have $\binom{n}{2}$ $(i, j)$ pairs of elements, the total number of inversions among all permutations is

$$\binom{n}{2} \cdot \frac{n!}{2} = \frac{n(n-1)}{2} \cdot \frac{n!}{2}.$$

If we assume that the probability of $\Pi = \Pi_k$ is equally likely (i.e. $P(\Pi = \Pi_k) = 1/n!$), then the average number of inversions is

$$\frac{1}{n!}\left(\frac{n(n-1)}{2} \cdot \frac{n!}{2}\right) = \frac{n(n-1)}{4} \in \Omega(n^2).$$

Since in the case that only local comparisons/moves are allowed, you can compare adjacent keys only and move the current pair of compared key locally, you actually need the amount of time to move all the inversions. Therefore, it takes $\Theta(n^2)$ time in the average case.

## 4

4.1 First, Kruskals algorithm sorts edge set in the non-decreasing order of weights. Then, the algorithm chooses edges from the sorted list as long as the current selected edge does not form a cycle with the previously-chosen edges.

Kruskals algorithm is a greedy algorithm, because at each step it adds to the tree an edge of least possible weight.

4.2 To sort the edge set, it takes $O(|E| \log n)$, where $|E|$ is the number of edges and $n$ is the number of vertices. If we use the Union-Find data structure, it takes $G(n)$ amortized time to detect whether the selected edge forms a cycle. Thus, it takes $O(|E|G(n))$ time to detect a cycle for all edges. Since $G(n) \in O(\log n)$, the total time complexity of the algorithm is $O(|E| \log n)$.

4.3 One of the approximation algorithms might be using the following simple observations of a tour: A tour $t$ is a simple cycle containing all vertices in $V$. i.e.,

(a) $degree(t) \leq 2$.

(b) it does not form a cycle until $|V|$ edges in $E$ are picked up.

Therefore, at each stage, we need to choose an edge of minimum weight satisfying above conditions. So, we take Kruskals algorithm and modify it. We sort the edge set as usual. And for each edge from the sorted list, we check whether vertices of it have more than 2 incident edges. And if yes, we discard this edge, otherwise, we keep it. We do this until the cycle consists of $|V|$ edges.

## 5

5.1
$$G(i,h) = \begin{cases} 0 & \text{if } i = 0 \\ \sum_{j=1}^{i} f_j(0) & \text{if } h = 0 \\ \max_{0 \le k \le h} f_i(k) + G(i-1, h-k) & \text{otherwise} \end{cases}$$

(For the initialization values ($i = 0$ and $h = 0$), it's okay not to state them.)

5.2 $G(0,h) = 0$ for $0 \le h \le H$ and $G(i,0) = \sum_{j=1}^{i} f_j(0)$ for $1 \le i \le n$. (For $i = 0$, $G(0,0)$ is already filled in with 0.)

For $G(0,h)$, you can interpret this as when you take no course, so that you get no points. For $G(i,0)$, you can think this as when you take first $i$ courses and spend no time on them, so that you get the base points added up to $i$ courses.

5.3 You first fill in the table with the initialization values $G(0,h) = 0$ for $0 \le h \le H$ and $G(i,0) = \sum_{j=1}^{i} f_j(0)$ for $1 \le i \le n$. Then, whenever $i^{th}$ course is added, you fill in $G(i,h)$ entry for all $1 \le h \le H$. To fill in $G(i,h)$ for one $h$, you need to compute the maximum value of $f_i(k) + G(i-1, h-k)$ over $0 \le k \le h$. Therefore, the total time to fill in each entry $G(i,h)$ is $O(H)$, and there are $O(nH)$ entries, for a total time of $O(nH^2)$. Since you only need to use $(i-1)^{th}$ row to fill in $i^{th}$ row, the space complexity is $O(H)$.

5.4 First, note that to maximize your average score is the same as to maximize your total score. Then, you can use above algorithm: While you are filling up $G(i,h)$ entry, you also record the value of $k$ that produces this maximum. (Here, you actually need $O(nH)$ space.) Then, you can backtrack through the entries using the recorded values to produce the optimal distribution of time like the following: You first look at $G(n,H)$ entry and find out how many hours you have spent on $n^{th}$ project, which is the recorded hour $k$ for $G(n,H)$ entry. Let us say that $h'$ is used. Then you look up the entry $G(n-1, H-h')$ and find out how many hours you have spent on $(n-1)^{th}$ project. You go on continuously until you reach the first project.

## 6

6.1

$$A^{(1)} = \begin{pmatrix} 1 & 1 & 0 \\ 1 & 1 & 1 \\ 0 & 0 & 1 \end{pmatrix}$$

$$A^{(2)} = \begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 0 & 0 & 1 \end{pmatrix}$$

$$A^{(3)} = \begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 0 & 0 & 1 \end{pmatrix}$$

6.2

$$A^{(k)} = A^{(k-1)}A^{(1)} \qquad \text{for } k = 2, 3, \cdots n$$

where each element $a_{ij}^{(k)}$of $A^{(k)}$ is computed as the following:

$$\begin{aligned} a_{ij}^{(k)} &= \sum_{m=1}^{n} a_{im}^{(k-1)} a_{mj}^{(1)} \\ &= a_{i1}^{(k-1)} a_{1j}^{(1)} + a_{i2}^{(k-1)} a_{2j}^{(1)} + \cdots + a_{in}^{(k-1)} a_{nj}^{(1)} \\ &= \bigvee_{m=1}^{n} (a_{im}^{(k-1)} \wedge a_{mj}^{(1)}) \end{aligned}$$

Note that the last equation is from that each element can have only 0 or 1 value.

## 7

7.1 We have to show that $m$ must be the joint median. Since $m = \max\{X[i], Y[j]\}$ and $i + j = N$, there are $N$ or more elements, which are smaller than or equal to $m$. But, by the assumption that $Y[j] \leq X[i+1]$ and $X[i] \leq Y[j+1]$, there are exact $N$ elements smaller than or equal to $m$. Thus, m is the joint median.

7.4 The joint median is 5. You can verify it by drawing a table of $(i, j)$ and see 5 is the only value which satisfies the given condition;

| $(0,5)$ | $(1,4)$ | $(2,3)$ | $(3,2)$ | $(4,1)$ | $(5,0)$ |
|---|---|---|---|---|---|
| $Y[j] = 10$<br>$X[i+1] = 1$<br>$X[i] = -\infty$<br>$Y[j+1] = \infty$ | $Y[j] = 8$<br>$X[i+1] = 3$<br>$X[i] = 1$<br>$Y[j+1] = 10$ | $Y[j] = 6$<br>$X[i+1] = 5$<br>$X[i] = 3$<br>$Y[j+1] = 8$ | $Y[j] = 4$<br>$X[i+1] = 7$<br>$X[i] = 5$<br>$Y[j+1] = 6$ | $Y[j] = 2$<br>$X[i+1] = 9$<br>$X[i] = 7$<br>$Y[j+1] = 4$ | $Y[j] = -\infty$<br>$X[i+1] = \infty$<br>$X[i] = 9$<br>$Y[j+1] = 2$ |
| x | x | x | o | x | x |

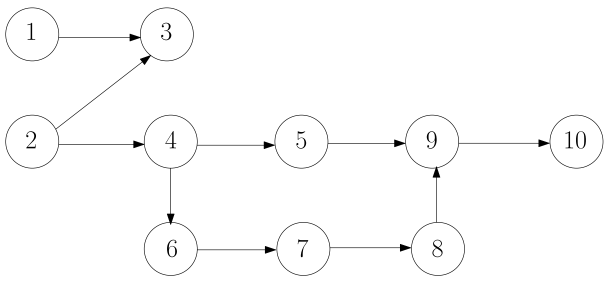
7.3 Each time, we compare the middle elements, $X[N/2]$ and $Y[N/2]$. If $X[N/2] > Y[N/2]$, then $M$ must be in $X[1 \cdots N/2]$ or in $Y[N/2 \cdots N]$. Otherwise, $M$ must be in $X[N/2 \cdots N]$ or in $Y[1...N/2]$. Thus, we have reduced the problem size into half of the original one.

Similarly, for the next step, in the case of $X[N/2] > Y[N/2]$, we pick up the middle elements, $X[N/4]$ and $Y[3N/4]$, from $X[1 \cdots N/2]$ and $Y[N/2 \cdots N]$. Otherwise, we pick up the middle elements, $X[3N/4]$ and $Y[N/4]$, from $X[N/2 \cdots N]$ and $Y[1...N/2]$.

We iterate this process until each one element remains. Since for each step, we reduce the size of problem into half, the total number of comparisons is $O(\log N)$.

## 8

8.1 There can be many solutions. The following figure is one of them.



8.2 One solution might be to compute the in-degree of all nodes and keep it as a list, which takes $O(|V|+|E|)$. Remove the node of in-degree 0 first and label it as 1. (This is possible since in every acyclic directed graph, there is a node with no incoming edges.) When you remove a node of in-degree 0, you have to reduce the in-degree of it's adjacent nodes as well. Repeat this process of labeling the node of in-degree 0 and reducing the in-degree of it's adjacent nodes until you label all the vertices. Since you can think reducing the in-degree of a vertex as removing some of its edges, after all you end up with no edges. Therefore, it takes $O(|E|)$. In total, the complexity of this algorithm is $O(|V|+|E|)$.

8.3 From the rule, if there exists an edge from a vertex $k$ to vertex $j$, $k < j$. Equivalently, $k \geq j$, there does not exists an edge from a vertex $k$ to a vertex $j$. Thus, for the case $k \geq j$, $a_{kj} = \infty$. Therefore, the following holds:

$$\min_{k \neq j}(v_k + a_{kj}) = \min_{k<j}(v_k + a_{kj}).$$

8.4 Use Bellman's formula defined in 8.3. To find the longest path, we need to negate the edge weights for all $a_{kj} \neq \infty$.