Prob. 1	Prob. 2	Prob. 3

Problem 1.

1. The idea to check for intimate pairs is to create a Voronoi diagram, then create a Delaunay diagram. If we have an edge of the Delaunay diagram, linking two points p_i and p_j, and this edge intersects with the edge of the Voronoi diagram separating those two points, then p_i and p_j are intimate.
   To prove it first it's obvious that for a given point, the only possible intimate points are the direct neighbors of it in the Voronoi diagram. The other points being further away, at least one of the direct neighbor would be inside the circle described in the definition of intimate points.
   Then why do we need the edge in the Delaunay diagram intersecting the edge in the Voronoi diagram? Well if we have only two points (p_1 and p_2) in the Voronoi diagram, then obviously the edge linking the two points (in the Delaunay diagram) intersects the edge separating those points in the Voronoi diagram. Now as we can see on Figure 1, if we add a point p_3 in the *intimate circle* then the edge separating p_1 and p_2 is going to be shorten where it intersects with the edge separating p_1 and p_3 and the edge separating p_2 and p_3. After this change in the Voronoi diagram the edge linking p_1 and p_2 will not intersect with the edge in the Voronoi diagram anymore. Yet if p_3 is added outside this *intimate circle* then the edge separating p_1 and p_2 is gonna be shorten but less and would still intersect the Delaunay diagram.

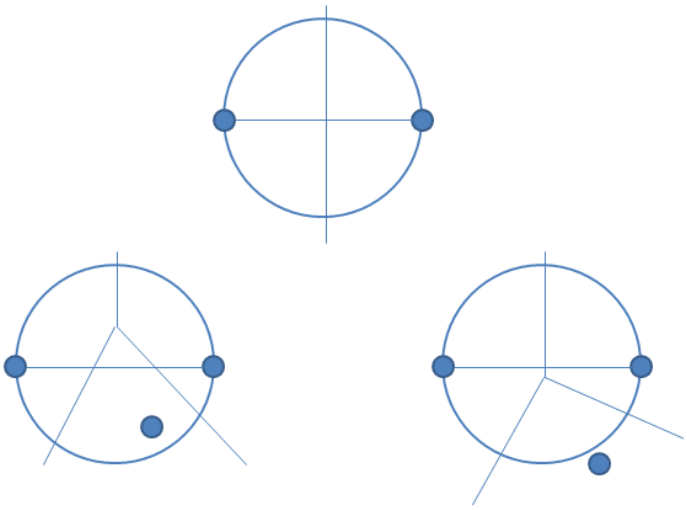

Figure 1: Illustration of the proof

For the running time, as we've seen in class, the Voronoi diagram can be build in $O(n \log n)$. The Delaunay diagram can be created in $O(n \log n)$ as well. After creating the Voronoi diagram we can have a data structure where we have the neighbors of each point and their separating edge and can access them in O(1). And then while we run the algorithm for the Delaunay diagram, every time we add an edge for the diagram we can check if it intersect the edge in the Voronoi diagram and if it does we can keep those points in a data structure. Those calcalations are done in O(1) so it doesn't add complexity in the algorithm. Finally the running time is then: $O(n \log n)+O(n \log n) \leq O(n \log n)$.
(source for Delaunay diagram: http://en.wikipedia.org/wiki/Delaunay_triangulation)

2. We can prove it by contradiction, so let's say there exists a minimum spanning tree where at least one edge is linking two non-intimate points (p_i and p_j). So this means there's a point p_k closer to p_j and to p_i. This p_k is either not connected to neither of those two points or it's connected to only one of them. p_k cannot be connected to both of them otherwise we have a loop.
Now if it's connected to one of them, the subtree with vertices $\{p_i,p_j,p_k\}$ should be a minimum spanning tree too. But it is not because the only minimum spanning tree is the one with path p_i-p_k-p_j, yet p_k cannot be connected to both points.
If p_k is not connected to either p_i or p_j, this means there is a subtree with p_k and this subtree is connected to the subtree p_i-p_j with an edge from a vertex p_x to either p_i or p_j and $p_x \neq p_k$. Let's say p_x is connected to p_i it's not a minimum spanning tree because if we remove the edge between p_i and p_j and add an edge p_k-p_j we have a smaller spanning tree. Respectively if p_x is connected to p_j and we remove the edge between p_i and p_j and add an edge p_k-p_i we have a smaller spanning tree (for a better understanding, see Figure 2. So we cannot end with a minimum spanning tree if p_i and p_j are directly connected and not intimate.

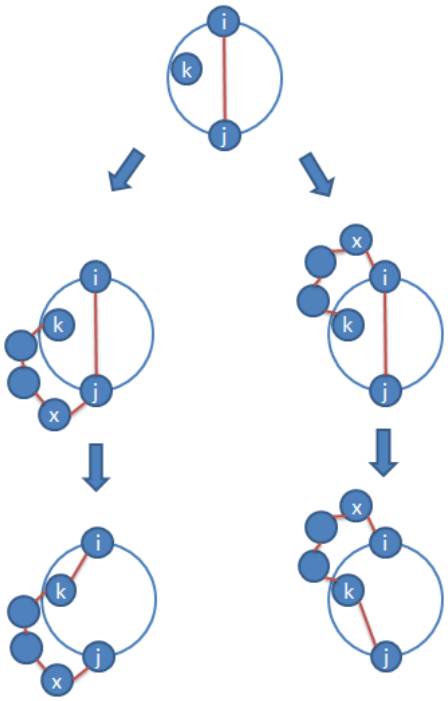

Figure 2: Illustration of the proof

Problem 2.

Firstly, we draw attention onto some important characteristics of the problem:

- Points must be traversed one by one because turning the direction (other than points) only increases the total delay.
- Visiting points is costless when we get there. Thus, if we are passing a non-visited node, we should note it as visited. As a result of this property, we can define an interval as visited. That's why next candidate point is the one right or left of the interval.

We need differences between points so we create a matrix of $n \times n$ to store differences. This matrix is created in $O(n^2)$. After we get matrix, we can lookup differences in $O(1)$.

Since we could see visiting process as an interval, we want a $n - length$ interval. We define our function(states) as $delay(i, j, k)$ where it is the total cost at point i with leftmost point j and rightmost point k. Since in an interval all points are visited, visiting a new point can extend the interval. Therefore, $delay(i, j, k)$ seems to have n^3 states but i actually can be equal to only j or k which means i is just a binary flag. As a result, we have only n^2 states.

With defined $delay(i, j, k)$ function, we can easily devise the minimum cost of $n - length$ path by

$$\min(delay(1,1,n), delay(n,1,n))$$

In each step, (with explained i) we will chop off leftmost or rightmost point and then take their min. Base case of our recursive algorithm is the interval with length of 1 when we simply return 0.

$$delay(i,j,k) = \begin{cases} min(& \\ \quad dist(j, j+1) + 2 \cdot delay(j+1, j+1, k), & \\ \quad dist(k, k-1) + 2 \cdot delay(k-1, j, k-1)) & \text{if } k - j + 1 > 1 \\ 0 & \text{if } k - j + 1 = 1 \end{cases}$$

Multiplier 2 in the above algorithm comes from the nature of the delay function: while visiting point i, all visiting costs prior to i are added into delay of i.

$$\sum_{i=2}^{n} \sum_{j=2}^{i} dist(i, j)$$

In analysis, our $delay$ function has n^2 states because j and k parameters can take any value from $[1, n]$, precisely $\sum_{j=1}^{n-1} \sum_{k=j+1}^{n} 1 \leq O(n^2)$. Moreover, we need to run algorithm for different starting points since we have $O(n)$ points and each run is $O(n^2)$, we get $O(n) \cdot O(n^2) \leq O(n^3)$ overall. Other costs such as finding actual path after getting minimum(optimal) delay or creating $n \times n$ matrix for differences will be dominated by $O(n^3)$.

Next, we will define the procedure to get the actual path since question asks us for it. We can modify the above algorithm (adding $O(1)$ time work) to make bookkeeping where it found the optimal. Therefore, we add predecessor pointer which is only the number of the point in the list. For $delay(i, j, k)$, if we came from $j + 1$, we add $j + 1$ for i. Otherwise, we add $k - 1$. Therefore, following these pointers in the reverse order gives us the actual path.

One final remark is that we assumed that points are given in sorted order. However, it isn't a problem because we can sort by ourselves unless they are in sorted order and it will also be dominated.

Problem 3.