Problem 1 (15 points).

You are given a matrix $A_{n\times n}$ with n^2 distinct integers. Matrix A satisfies a property that the numbers are increasing along any of its $(2n-1)$ diagnals, i.e., $A[i,k] < A[i+1,k+1]$, for all $1 \le i < n$ and $1 \le k < n$. Design an algorithm to sort all elements in A, i.e., to output a sorted array with all n^2 elements in A. Your algorithm should run in $O(n^2 \cdot \log n)$ time.

Solution. This problem essentially asks to merge $(2n-1)$ sorted arrays, each of which corresponds to a diagnal as it is sorted. We now describe a divide-and-conquer algorithm that merges m sorted arrays. Define recursive function merge-multi-sorted-arrays ($M[1\cdots m]$) returns the sorted array after merging the given m sorted arrays; here we assume $M[i]$ actually stores an array (instead of a single number), and the lengths of these m sorted arrays could be different.

```
function merge-multi-sorted-arrays ($M[1\cdots m]$)
    if $m = 1$: return $M[1]$;
    $A_1$ = merge-multi-sorted-arrays ($M[1\cdots m/2]$);
    $A_2$ = merge-multi-sorted-arrays ($M[m/2+1\cdots m]$);
    return merge-two-sorted-arrays ($A_1$, $A_2$);
end function
```

Let $T(m,L)$ be the running time of merge-multi-sorted-arrays ($M[1\cdots m]$) where L represents the maximum length of the m input sorted arrays. The size of A_1 and A_2 is at most $mL/2$; the merge-two-sorted-arrays used then takes $O(|A_1|+|A_2|) = O(mL)$. Then we have the recurrence $T(m,L) = 2\cdot T(m/2,L) + O(mL)$. Applying the master's theorem gives $T(m,L) = O(mL\cdot\log m)$.

Back to the original problem, we will call merge-multi-sorted-arrays with the $(2n-1)$ sorted arrays (transcribed from the diagnals) as input, and in this case $L = n$ (the length of the main diagnal). Hence, the overall running time is $O((2n-1)\cdot n\cdot\log(2n-1)) = O(n^2\cdot\log n)$.

Note: in fact, you may also simply sort all n^2 elements, which also takes $O(n^2\cdot\log n)$ time.

Problem 2 (15 points).

Given n distinct points $P = \{p_1, p_2, \cdots, p_n\}$ that are sorted along one line, design an algorithm to pick m of them, $m < n$, such that the sum of the distance of each of the n points to its nearest picked point is minimized. Your algorithm should run in $O(mn^2)$ time. (*Hint:* consider the simpliest case with $m = 1$, then the optimal strategy is to pick the $(n+1)/2$-th point if n is odd, and to pick $(n/2)$-th or $(n/2+1)$-th point if n is even; try to use this observation in your algorithm.)

Solution. Define $F(i,j)$ as the optimal objective value of the subproblem with the first i points on the line among which j of them should be marked. To compute $F(i,j)$, suppose that the last marked point covers the last $(i-k)$ points, i.e., the last marked point is the nearest marked point for $p_{k+1}, p_{k+2}, \cdots, p_i$. (We don't know k and therefore we will enumerate all the possible values in the recurrence.) Once we know k (through enumeration), those heading k points $p_1, p_{i+1}, \cdots, p_k$ form a subproblem of $F(k, j-1)$. Let $T(k+1,i)$ be the optimal objective value to cover points $p_{k+1}, p_{k+2}, \cdots, p_i$ using one marked point. We then have the recureence: $F(i,j) = \min_{k=j-1}^{i-1}(F(k,j-1) + T(k+1,i))$.

Now we compute $T(i,j)$, for all $1 \le i \le j \le n$. According to the hint, we have the recursion of $T(i,j) = T(i+1, j-1) + d(p_i, p_j)$. Therefor, computing all $T(i,j)$ takes $O(n^2)$ time using a (simple) dynamic programming algorithm. This serves as the preprocessing the above main algorithm.

Back to the main algorithm: solving each subproblem takes $O(n)$ time; Therefore, the main algorithm takes $O(m \cdot n^2)$ time as we need to solve $O(mn)$ subproblems. The overall running time is $O(n^2 m)$.

Bonus Problem (5 points).

You are given an array A with n distinct positive integers. Each element must be pushed into a stack and then poped out from the stack. When we push, the element with the smallest index is removed from array A and then pushed into the top of the stack. When we pop (of course, the stack must be not empty), the element on the top of the stack is removed. In the beginning, the stack is empty. The push and pop can be operated in any order until all elements are poped out. All elements are then naturally sorted according to their order that is poped out. Let $P(i)$ be the index of $A[i]$ in this new rearranged array. Design an algorithm to compute one push and pop sequence such that $\sum_{k=1}^{n} A[i] \cdot P(i)$ is minimized. Your algorithm should run in $O(n^3)$ time. For example, let $A = (5,8,4)$. One possible sequence of operations are push (5), push (8), pop (8), pop (5), push (4), pop (4). Thus, under this scenario, $P(1) = 2$, $P(2) = 1$ and $P(3) = 3$, and $\sum_{k=1}^{3} A[i] \cdot P(i) = 5 \cdot 2 + 8 \cdot 1 + 4 \cdot 3 = 30$. This is also the optimal scenario.

Solution. Consider the first element $A[1]$. Suppose that $P(1) = k$, i.e., $A[1]$ is the kth element that is poped out. Who are the $(k-1)$ elements that are poped before $A[1]$? They must be $A[2,\cdots,k]$! In other words, when the order of $A[1]$ is fixed to k, we then have two independent subproblems, formed by $A[2,\cdots,k]$ and $A[k+1,\cdots,n]$. Let $F(2,k)$ and $F(k+1,n)$ be the optimal objective value of these two subproblems. Thus, the objective value of the whole problem under the condition of $P(1) = k$ can be computed as $A[1] \cdot k + F(2,k) + F(k+1,n) + k \cdot \sum_{i=k+1}^{n} A[i]$. The existence of the last item is because the poping order of each element in subproblem $A[k+1,\cdots,n]$ should be added k to be equal to its poping order in the whole problem.

Following this, we can first preprocess the array to compute $S(i,j) = \sum_{k=i}^{j} A[k]$, for all $1 \le i \le j \le n$. Second, define $F(i,j)$ as the minimum objective value of of the subproblem formed by $A[i,\cdots,j]$. We compute $F(i,j)$ using the recursion of $F(i,j) = \min_{k=1}^{j-i+1}(A[i] \cdot k + F(i+1,i+k-1) + F(i+k,j) + k \cdot S(i+k,j))$. Finally, $F(1,n)$ is the optimal objective value of the whole problem.

The preprocessing costs $O(n^2)$ and the main step of the algorithm clearly costs $O(n^3)$, which dominates the running time of the whole algorithm.