1. (a) In the best case, Line #2 ("if $A[i] = v$") is executed once, which means for any input (A, v), $A[1] = v$.

   (b) The probability that the best case occurs is $\frac{1}{n+1}$, since for any input v, $1 \leq v \leq n$, the chances that $A[1] = v$ is $\frac{1}{n+1}$.

   (c) In the worst case, for input (A, v), Line #2 is executed n times, which means $v \notin A$ or $A[n] = v$ (if $v = 1$). For instance $A[1] = A[2] = ... = A[n] = 0$.

   (d) For input (A, v), assume $v = n - i$, where $0 \leq v \leq n - 1$,

$$\textit{probability that the worst case occurs is} \begin{cases} \frac{v}{n+1} & \text{if } v > 1 \\ \frac{2}{n+1} & \text{if } v = 1 \end{cases}$$

$i = 0, v = n$, then probability that the worst case occurs is $\frac{n}{n+1}$.
$i = 1, v = n - 1$, then probability that the worst case occurs is $\frac{n}{n+1} \times \frac{n-1}{n} = \frac{n-1}{n+1}$.
$i = 2, v = n - 2$, then probability that the worst case occurs is $\frac{n}{n+1} \times \frac{n-1}{n} \times \frac{n-2}{n-1} = \frac{n-2}{n+1}$.
...
$i = k, v = n - k$, then probability that the worst case occurs is $\frac{n}{n+1} \times \frac{n-1}{n} \times ... \times \frac{n-k}{n+k-1} = \frac{n-k}{n+1}$.
...
$i = n - 1, v = 1$, then probability that the worst case occurs is $\frac{n}{n+1} \times \frac{n-1}{n} \times ... \times \frac{2}{3} \times \frac{2}{2} = \frac{2}{n+1}$.

   (e) In the average case, for input (A, v), assume $v = n - i$, where $0 \leq v \leq n - 1$, Line #2 expected to be executed $\frac{(n-v)^2+3(n-v)+2nv+2}{2(n+1)}$ times.
   DENOTE : N to be the number that Line #2 is executed.

   For $v = n - i$ we have:

$N = 1$, probability is $\frac{1}{n+1} = \frac{1}{n+1}$.
$N = 2$, probability is $\frac{n}{n+1} \times \frac{1}{n} = \frac{1}{n+1}$.
$N = 3$, probability is $\frac{n}{n+1} \times \frac{n-1}{n} \times \frac{1}{n-1} = \frac{1}{n+1}$.
...
$N = i + 1$, probability is $\frac{n}{n+1} \times \frac{n-1}{n} \times ... \times \frac{1}{n-i+1} = \frac{1}{n+1}$.
$N = n$, probability is $\frac{n}{n+1} \times \frac{n-1}{n} \times ... \times \frac{n-i}{n-i+1} = \frac{n-i}{n+1}$.

   Therefore, for the average case with input (A, v), assume $v = n - i$, Line #2 is expected to be executed $(1 + 2 + 3 + ... + (i + 1)) \times \frac{1}{n+1} + n \times \frac{n-i}{n+1} = \frac{(i+1)(i+2)}{2} \times \frac{1}{n+1} + n \times n - in + 1$ substitute $i = n - v$ then we get $\frac{(n-v)^2+3(n-v)+2nv+2}{2(n+1)}$.

2. (a) Let, L be an empty list.
Suppose every node is a tuple of the form (v, i), where v is the actual value of the node and i is the index of the list where it belongs to.

   i. First, using the last node(biggest) in every list to build a max heap based on first component. Then. remove the last nodes in all lists. Clearly, this heap contains k nodes and building this heap has worst case runtime $\mathcal{O}(\ k)$.
   ii. Then, using *Extract Max* on this heap, and append the returned node to the front of list L. Also, clearly, the *Extract Max* method has worst case runtime $\mathcal{O}(log\ k)$.
   iii. Case 1: If the list which this node belongs is not empty then insert the last node from that list to heap and remove the that node from the list. Same as previous, *Insert* also has worst case runtime $\mathcal{O}(logk)$.
   iv. Case 2: If the list which this node belongs is empty, repeat step ii, until list where the returned node belongs is not empty or heap is empty.
   v. return *result* when heap is empty.

   Based on the algorithm the heap always contains the current biggest nodes of each list, then when we do *Extract Max* we get the current biggest node among all k lists and append that node to the font of L. Therefore, *result* is sorted in non-descending order. Since, after we built the heap, every time we de *Extract Max* or *Insert* worst case runtime is in order $\mathcal{O}(logk)$ and there are n nodes in total so the worst case runtime is in order $\mathcal{O}(nlog\ k)$.

(b) Assume that the number of solutions is $\mathcal{O}(n^2 log(n))$.
First let's define (x_1, x_2, x_1, x_2) as a trivial solution and other solutions which don't have that form is non-trivial.
Our goal here is find all non-trivial solutions, since it's really easy to find all trivial solutions, just do a combination of $n \times n$, and every node it a trivial solution. The worst case runtime is in order $\mathcal{O}(n^2)$.
Now, lets find all non-trivial solutions.
Define *solution* and *track* to be two empty lists, and *current* to be the node $(0, 0, 0)$.

   i. Image there are n lists, where $list_i$ $(1 \le i \le n)$ is $\{(i^7 + a^7, i, a)$ and a is increasing from 1 to n.
   For example $n = 3$
   $i = 1$, $[(1^7 + 3^7, 1, 3), (1^7 + 2^7, 1, 2), (1^7 + 1^7, 1, 1)]$
   $i = 2$, $[2^7 + 3^7, 2, 3), (2^7 + 2^7, 2, 2), (2^7 + 1^7, 2, 1)]$
   $i = 3$, $[(3^7 + 3^7, 3, 3), (3^7 + 2^7, 3, 2), (3^7 + 1^7, 3, 1)]$
   Note: every $list_i$ is strictly increasing.
   ii. First we build a max heap (based on the first component of a node) initially containing $(1^7 + n^7, 1, n), (2^7 + n^7, n), (3^7 + n^7, 3, n), ..., (n^7 + n^7, n, n)$ which are the first elements in those lists.
   iii. Then we do merge sort on those lists based on the first component. However, since we don't have the lists this time, so what we actually do is, suppose we extract a node $(i^3 + j^3, i, j)$ then we insert $(i^3 + (j-1)^3, i, j-1)$ if $j - 1 > 0$ or we keep extract like $2(a)$ does.
   iv. So the basic the merge algorithm is like $2(a)$, but the differences are this time we keep tracking the node we extracted.
   Each time we exact a node from heap:
   if the node $(i^3 + j^3, i, j)$ we extracted has the same first component with *current*.

i combine this node with each nodes in list *track* we get several solutions;

ii append those solutions to list *solution*;

iii append *current* to list *track*.

Note that each solution we get take constant time $\mathcal{O}(1)$

If the node $(i^3 + j^3, i, j)$ we extracted does not has the same first component *current*,

i empty the *track*;

ii append newly extracted node to *track*;

iii set current to $(i^3 + j^3, i, j)$.

v. return *solution* when the heap is empty.

Actually, what the algorithm basically does is sorted those n imaginary lists in non-descending order based on the first component. And clearly, all the nodes with same first component are together in the list. So for each of those subsequence which has the same first component, we calculate all possible combinations using list *track* and node *current*. Hence, *solution* gives us all possible solutions.

Since, to get a single solution takes constant time $\mathcal{O}(1)$ and the number of solutions is $\mathcal{O}(n^2 log(n))$;therefore, to get all solution we need $\mathcal{O}(n^2 log(n))$. And from $2(a)$ we know the worst case runtime for the heap part is also in order $\mathcal{O}(n^2 log(n))$. Hence, the algorithm has worst-case runtime $\mathcal{O}(n^2 log(n))$.