1. (a) In the best case, $A[1]$, the first element we check, is v, so we find v after one comparison, i.e., Line #2 is executed only once.

   (b) The probability of the best case is the probability that we choose v for $A[1]$, out of $n+1$ candidates ($\{0,\ldots,n\}$). Since it's chosen uniformly at random, this probability is $1/(n+1)$.

   (c) In the worst case, v does not exist in A or only $A[n]=v$, and the algorithm will go through the whole array, i.e., executing Line #2 for n times.

   (d) The probability of the worst case is the probability that v is not chosen by any of the entries $A[1]$, $A[2]$, …, $A[n]$. For each one entry, the probability of not choosing v is different.

   For entries after $A[n-v+1]$, the probability of not choosing v is simply 1, because for $i > n-v+1$, $A[i] \le n-i+1 < n-(n-v+1)+1 = v$.

   For entries before $A[n-v+1]$ (including $A[n-v+1]$), v is a candidate. Thus for $A[i]$, $i \le n-v+1$, there are $n-i+2$ total choices so the probability of not choosing v is $(n-i+1)/(n-i+2)$.

   Since all choices are made independently, we just take the product of probabilities of v not chosen by each entry to get the probability that v is not chosen by any entry, i.e.,

$$\Pr(v \text{ not in } A) = \prod_{i=1}^{n} \Pr(v \text{ not chosen by } A[i]) = \prod_{i=1}^{n-v+1} \frac{n-i+1}{n-i+2} = \frac{v}{n+1}$$

   However, there is a special case when $v=1$, because the worst-case (n comparisons) happens in two possible ways. One is that v does not exist in A; the other is that v is found at $A[n]$ but we do n comparisons anyway. The probability of the former way is $1/(n+1)$ by plugging in $v=1$ to the above equation; the probability of the latter way is also $1/(n+1)$ because $A[n]$ chooses 0 and 1 with equal probabilities, so the probability of 1 not being in the array is the same as the probability that 1 only appears in the last position of the array. So overall the probability of worst-case when $v=1$ is $2/(n+1)$.

   The final answer to the worst-case probability is

$$\Pr(\text{worst-case}) = \begin{cases} v/(n+1) & \text{if } 2 \leqslant v \leqslant n \\ 2/(n+1) & \text{if } v = 1 \end{cases}$$

   (e) For the average case, we need to determine the probability that we need to perform t ($1 \leqslant t \leqslant n$) comparisons until finding (or not finding) v. Let t_n be a random variable that denotes the number of comparisons (Line #2) executed. The first thing to note is that only certain values are possible for t_n, which are $1, 2, \ldots, n-v+1$ and n. That is, either you find v within $n-v+1$ steps, or you never find it (after $n-v+1$ steps, v would not be a candidate any more).

   Now let's figure out the probability of $t_n = 1, 2, \ldots, n-v+1$, when v is found. The probability that $t_n = t$ is the probability that v is not chosen by $A[1]$, $A[2]$, …, $A[t-1]$ and is chosen by $A[t]$, i.e.,

$$\Pr(t_n = t) = \frac{1}{n-t+2} \cdot \prod_{i=1}^{t-1} \frac{n-i+1}{n-i+2} = \frac{1}{n+1}$$

   Note that, this probability is independent of t as long as we are given that v is found. Then the probability that v is not found is simply 1 minus the sum of probabilities of all v-found cases (also the result of Part (d)), which is

$$\Pr(v \text{ not found}) = 1 - (n-v+1) \cdot \frac{1}{n+1} = \frac{v}{n+1}$$

Thus the complete description of t_n's probability distribution is

$$\Pr(t_n = t) = \begin{cases} 1/(n+1) & 1 \leqslant t \leqslant n-v+1 \ \ (v \text{ found}) \\ v/(n+1) & t = n \ \ (v \text{ not found}) \\ 0 & \text{otherwise} \end{cases}$$

Note that when $v = 1$, t_n could be n for two reasons: found at the last comparison or not found. Finally, the average running time of FINDLAST, with the given input distributions, is

$$E[t_n] = n \cdot \frac{v}{n+1} + \sum_{i=1}^{n-v+1} i \cdot \frac{1}{n+1} = \frac{2nv + (n-v+2)(n-v+1)}{2(n+1)}$$

2. (a) When we merge two arrays into one sorted array, we use two pointers, initially pointing at the beginning of each array. Then we compare the two values pointed by the two pointers and increment the one that is pointing to the smaller value (after appending to smaller value to the output array). We keep incrementing the pointers until one of them reaches the end of its array. Then, append to the output whatever is left in the other array, and we will get a merged sorted array. The running time of this algorithm is $\mathcal{O}(n)$ since we visit each element at most once. The thing to note here is that, at each iteration, we needed to determine the minimum of *two* values being pointed to, which takes constant time.

   Merging k sorted arrays is essentially the same as merging two sorted arrays, except that at each iteration we need to determine the minimum of k values being pointed to, instead of 2. If we find the min by just going through the k values linearly, that will result in a $\mathcal{O}(nk)$ overall running time. We can do it more efficiently using a min-heap. We maintain a heap with k elements where each element corresponds to a pointer, and the key is the value that the pointer points to. At each iteration, we do the following:

   i. Get *minptr* by calling EXTRACTMIN on the heap.
   ii. Append the value pointed by *minptr* to the output array.
   iii. Increment *minptr*.
   iv. Insert *minptr* to the heap, with the new pointed value being its key.

   Since EXTRACTMIN and INSERT both take $\mathcal{O}(\log k)$ time, the overall running time of the merging is then $\mathcal{O}(n \log k)$.

   The heap initially contains the smallest value from each array which hasn't been appended into the output array. Since the arrays are in non-descending order, this property is maintained every time *minptr* is incremented and the next value is inserted. Therefore, every time the minimum value is extracted from the heap, it is the smallest value from all of the arrays which has not been added to the output array. This ensures that the values are extracted in non-descending order and that the algorithm is correct.

   (b) This problem can be solved by performing a merge of n wisely-defined arrays. Let $R_{x,y}$ denote the triple $(x, y, x^7 + y^7)$. We have in total n^2 triples from $R_{1,1}$ up to $R_{n,n}$.

   Now we define the n sorted input arrays, namely, A_1 to A_n. Let A_i contain $R_{i,1}, R_{i,2}, \ldots, R_{i,n}$, i.e., A_1 contains $R_{1,1}, R_{1,2}, \ldots, R_{1,n}$, A_2 contains $R_{2,1}, R_{2,2}, \ldots, R_{2,n}$, etc. Based on this definition, the triples $(x, y, x^7 + y^7)$ in each array have the same x, have y in increasing order, and also have $x^7 + y^7$ in increasing order.

   Then we merge the n sorted arrays of triples $A_1, \ldots, A_n$, using the algorithm described in Part (a). We use the value of $x^7 + y^7$ as the comparison key while merging. As a result, the

output will be a single array of triples $(x, y, x^7 + y^7)$ sorted according to $x^7 + y^7$. If the equation $a^7 + b^7 = c^7 + d^7$ has any solution, we would have **adjacent** triples in the output array that have the same $x^7 + y^7$, and the x's and y's in these triples are exactly the solutions a, b, c, d that we are looking for.

We can find all possible solutions to the equation using the following search strategy, which iterates through the sorted output array. For each position i in the array, first add (a, b, a, b) to the list of solutions, where (a, b) is the (x, y) pair at position i. Next, consecutively iterate through positions directly after i which have the same $v = x^7 + y^7$ value, until you have reached an entry with a different v. At each position j after i with the same value v, add (a, b, c, d) and (c, d, a, b) to the solution list, where (c, d) is the (x, y) pair at position j.

Now suppose that (a, b, c, d) was a solution to $v = a^7 + b^7 = c^7 + d^7$. Then (a, b) and (c, d) are in the sorted output array and all entries between the two have the same value v. In all cases - when $(a, b) = (c, d)$, (a, b) appears before (c, d) or (a, b) appears after (c, d) - the search procedure would have added (a, b, c, d) to the solutions list. Therefore, the solutions list contains all possible solutions. Since only integer pairs with the same v value were added to the solutions list, the solutions list only contains all possible solutions, which proves the algorithm is correct.

The worst-case running time of the merging, according to the analysis in Part (a), is $\mathcal{O}(n^2 \log n)$ since there are n arrays and n^2 elements in total. We note that the search procedure performed a constant amount of work for each pair of positions which resulted in a unique solution. Since there are $\mathcal{O}(n^2 \log n)$ solutions, the search took $\mathcal{O}(n^2 \log n)$. Therefore, the algorithm has a worst-case run time of $\mathcal{O}(n^2 \log n)$.

Note: Google "taxicab number" for more insights into this problem.