a) The worst-case time-efficiency in Θ[·] notation of BINSEARCHRECURSIVE is the same with BINSEARCH. For BINSEARCH, its time-efficiency can be characterized as the number of loop iterations x the time for each iteration. For iterations that we don't return in Line(3), the time to compare lo ≤ hi in Line(1), the time to compute and assign mid in Line(2), the time to compare A[mid] = i in Line(3), the time to compare A[mid] < i in Line(4), and the assignment of lo or hi in Line(4) or Line(5), they all take the same time, and we could argue that for each iteration, the time is the same, some constant $c \times logn$. Where n is number of entries in array A. And let t(k) be the number of loop iterations that happen in the worst-case in BINSEARCH, where k = hi-lo+1, then t(k) = O(log k). And for worse-case time-efficiency of BINSEARCH, its time-efficiency is Θ (log n). For BINSEARCHRECURSIVE, the time-efficiency could be written down as a recurrence. Suppose T(lo,hi) is the worst-case number of times Line(4) or (5) runs with Line(3) not returning true.

So, $T(lo, hi) = \begin{cases} \Theta(1) & if\ lo > hi\ (1) \\ T(mid+1, hi) + \Theta(c) & if\ A[mid] < i\ (2), \\ T(lo, mid-1) + \Theta(c) & if\ A[mid] > i\ (3) \end{cases}$

where $mid = \left\lfloor \frac{lo+hi}{2} \right\rfloor$, $c\ is\ some\ constant, and\ i\ is\ the\ element\ we\ want\ to\ find.$

No matter if (2) or (3) gets executed, the total number of iterations in the worst case are the same. Suppose we adopt c in place of Θ(c), and 1 for Θ(1).

T(lo, hi) → T(mid+1, hi) + c

→ T($\left\lfloor \frac{mid+hi}{2} \right\rfloor$ + 1, hi) + c + c

→ T(mid+1, mid) + c +… + c

→1 + ($\log_2$n)c

= Θ ($\log_2$n)

Where n is the number of entries in array A. This is the same as BINSEARCH.

b) The worst-case space-efficiency in Θ[·] notation of BINSEARCHRECURSIVE is worse than BINSEARCH. BINSEARCH can be characterized as highly space-efficient. The only space it needs, in addition to that needed to encode the input, is those for lo, hi and mid, and the space-efficiency of BINSEARCH can be characterized as $3(1 + \lfloor log_2 n \rfloor)$. But, if we choose the call-stack depth as a characterization of the space-efficiency of this algorithm, then its worst-case space-efficiency is just Θ(1), as only 1 call stack will be allocated for this algorithm.

But for BINSEARCHRECURSIVE, it's completely different. For each recursive call that happened in the algorithm, a new call stack is needed and will be allocated for it. So, for this algorithm, the max depth of the call stack is Θ($\log_2$n), where n is the number of entries in array A. Hence, if we choose the call-stack depth as a characterization of the space-efficiency of this algorithm, then its worst-case space-efficiency is Θ(n).

c) BINSEARCHCEIL does possess the termination property. The termination property is: for every legal input, <A, lo, hi, i>, BINSEARCHCEIL is guaranteed to halt. By legal input, we mean one that meets the constraints we impose on the invoker of BINSEARCHCEIL. That is, A must be a finite array of finite sorted, non-decreasing integers, of $n \geq 1$ finitely many entries, and lo and hi must be integers that satisfy $0 \leq lo \leq hi \leq n-1$, and i must be a finite integer.

So, we first observe that if BINSEARCHCEIL is invoked with lo > hi, then we return immediately in Line(16) as the while condition evaluates to false. For the case that BINSEARCHCEIL is invoked with $lo \leq hi$, if we return in Line(13) in that iteration, the algorithm has terminated. Now suppose that when we enter an iteration of the while loop, we do so with lo, hi values of lo_1, hi_1, respectively, and suppose in that iteration, we do not return in Line(13). Thus, we are guaranteed to check the while condition again. This second time, suppose that the lo, hi values are lo_2, hi_2, respectively.

We claim that $hi_2 - lo_2 < hi_1 - lo_1$

To prove this, let $m = \left\lceil \frac{lo_1 + hi_1}{2} \right\rceil$. We observe that we need to consider 2 cases.

(1) $lo_2 = m + 1$, $hi_2 = hi_1$, and , (2) $lo_2 = lo_1$, $hi_2 = m - 1$

In case (1):

$$hi_2 - lo_2 = hi_1 - m - 1$$

$$= hi_1 - \left\lceil \frac{lo_1 + hi_1}{2} \right\rceil - 1$$

$$< hi_1 - \left(\frac{lo_1 + hi_1}{2} - 1\right) - 1 \qquad \because \forall x \in R,\ x < \lceil x \rceil$$

$$= \frac{hi_1 - lo_1}{2}$$

$$< hi_1 - lo_1$$

Similarly, in case (2):

$$hi_2 - lo_2 = m - 1 - lo_1$$

$$= \left\lceil \frac{lo_1 + hi_1}{2} \right\rceil - 1 - lo_1$$

$$< \left(\frac{lo_1 + hi_1}{2} + 1\right) - 1 - lo_1$$

$$= \frac{hi_1 - lo_1}{2}$$

$$< hi_1 - lo_1$$

Thus, we have proven that $hi_2 - lo_2 < hi_1 - lo_1$. Thus, if we start out with $lo \leq hi$, then we either return in Line(13) in some iteration, or, if we do not, eventually $hi - lo < 0 \rightarrow lo > hi$, and we exit the while loop and terminate.

d) From Line(12) of the algorithm, mid = $\left\lceil \frac{lo+h}{2} \right\rceil$. We carry out the following case-analysis.

Case 1:*lo* + *hi* is even. In this case, $\left\lceil \frac{lo+hi}{2} \right\rceil = \left\lfloor \frac{lo+hi}{2} \right\rfloor$, and the proof would be the same.

Case 2:*lo* + *hi* is odd. In this case:

$$\text{mid} = \frac{lo+hi+1}{2}$$
$$\text{mid} \geq \frac{lo+lo}{2} \qquad \because \text{lo} \leq \text{hi}$$
$$\text{mid} \geq \text{lo} + 1/2 \geq \text{lo} \qquad \because \text{lo} + 1/2 \geq \text{lo}$$

Similarly, we have:

$$\text{mid} = \frac{lo+hi+1}{2}$$
$$\text{mid} \leq \frac{hi+hi+1}{2} \qquad \because \text{lo} \leq \text{hi}$$
$$\text{mid} \leq \text{hi} + 1/2$$
$$\text{mid} \leq \text{hi} \qquad \because \text{both mid and high are integers}$$