4) Run time analysis:

1)

i) it is necessary to do the search and return action only $\frac{n}{2}$ times because on the $\frac{n}{2}+1$ iteration we will go over all the array so there is no need to do the step back.

The number of forward steps is $\sum_{i=1}^{\frac{n}{2}+1} i$ and the number of backward steps is $\sum_{i=1}^{\frac{n}{2}}(i-1)$.

This two sums are arithmetic progression sums :

- $\sum_{i=1}^{\frac{n}{2}+1} i = \frac{\left(1+\frac{n}{2}+1\right)*(\frac{n}{2}+1)}{2} = \frac{1}{8}(n+4)(n+2)$
- $\sum_{i=1}^{\frac{n}{2}}(i-1) = \frac{\left(0+\frac{n}{2}\right)*(\frac{n}{2})}{2} = \frac{1}{8}(n-2)*n$

$$\sum_{i=1}^{\frac{n}{2}}(i-1) + \sum_{i=1}^{\frac{n}{2}+1} i = \frac{1}{8}(n-2)*n + \frac{1}{8}(n+4)(n+2) = \frac{2n^2+4n+8}{8}$$

Lower bound :

We need to show that there are c, n0 such that for n≥n0 :

$\frac{2n^2+4n+8}{8} \geq cn^2$

$\frac{n^2+2n+4}{4n^2} \geq$ c the expression on the left side will be the smallest when n →∞.

We will choose c = $\frac{1}{8}$ and n0=2

Therefore T(n) = $\Omega(n^2)$

Upper bound :

We need to show that there are d, n0 such that for n≥n0 :

$\frac{2n^2+4n+8}{8} \leq dn^2$

$\frac{n^2+2n+4}{4n^2} \leq$ d the expression on the left side will be the greatest when n=1.

We will choose c = 5 and n0=1

Therefore T(n) = $O(n^2)$

In conclusion, T(n) = $\Theta(n^2)$

ii)

we have notice that on the first iteration we will call the functions 3 times (2 seps and 1 backtrack) and that the i^{th} iteration is depends on its previous calls.

We have managed to come up with the following equation that describe the number of calls to the functions step and backtrack for the i^{th} iteration (**specifically**):

T(i) = T(i − 1) + 2(i-2) , T(1)=3

Use the iteration method:

T(i) = T(i − 1) + 2(i-2) = T(i − 2) + 2(i-1-2) + 2(i-2) = T(i − 2) + 2(2i-5)

= T(i − 3) +2 (i-2-2) + (4i-10) = T(i − 3) + 6i - 18

= … = $T(i-k) + 2(k*i - \sum_{j=2}^{k+1} j\ )$

For k = i-1 (the worst case) we get:

$T(i) = T(1) + 2[i*(i-1) - \frac{1}{2}(i^2+i-2)] = T(1) + i^2 - 3i + 2 = i^2 - 3i + 5$

(i in its worst case equals to n)

Now we will show in induction on i=n (worst case):

<u>Lower bound:</u>

Induction assumption - Assume T(k) ≥ c$(k^2 - 3k + 5)$ for all 1 ≤ k < n
Induction step – for k=n we will prove T(n) ≥ c$k^2 - 3k + 5$:
T(n) = T(n − 1) + 2(n-2) ≥ $c[(n-1)^2 - 3(n-1) + 5] + 2(n-2) =$

$d(n^2 - 5n + 9) + 2n - 4$

For c=1 and n_0=1 T(n) is **greater/equals** to n^2.

Therefor T(n) = $\Omega(n^2 - 5n + 9)$

<u>Upper bound:</u>

Induction assumption - Assume T(k) ≤ d$(k^2 - 3k + 5)$ for all 1 ≤ k < n
Induction step – for k=n we will prove T(n) ≤ d$n^2 - 3n + 5$:
T(n) = T(n − 1) + 2(n-2) ≤ $d[(n-1)^2 - 3(n-1) + 5] + 2(n-2) =$

$d(n^2 - 5n + 9) + 2n - 4$

For d=1 and n_0=1 T(n) is **less/equals** to n^2.

Therefor T(n) = $O(n^2 - 5n + 9)$

T(n) = $\Omega(n^2 - 5n + 9)$ and T(n) = O($n^2 - 5n + 9$) → T(n) = $\Theta(n^2 - 5n + 9) \approx \Theta(n^2)$

In the worst case we will need to go over the entire array (n) but by the instruction we can understand that n-1 iterations it enough because that the first step of the n^{th} iteration is a step and we will finish scan the array.

So no we can analyze the final result depending on the calculation above:

$$\sum_{i=1}^{n-1} i^2 = \frac{1}{6}(2n^3 - 3n^2 + n)$$

the expression $a * n^3 + b * n^2 + c * n + d$, the terms $c * n^2$b*n and d become negligible compared to $a * n^3$ when n approaches infinity.

Thus, the time complexity of insertion sort is ≈ $a * n^3$.

We are not interested in the constant a. Thus, we say that the running time of insertion sort grows like n when n approaches infinity which means Θ(n^3)

$$\begin{pmatrix} the\ precise\ amount\ of\ calls\ to\ step\ and\ backtrack\ is: \\ \sum_{i=1}^{n-1} i^2 - 3i + 5 = \frac{1}{3}(n^3 - 6n^2 + 20n - 15) \end{pmatrix}$$

iii)

backtrackingSearch (arr, x, n1, n2, myStack)	cost	times
For i = 1 to arr.length (include) do:	C1	n
for j=1 to n1 (include) do:	C2	(n-1)*n_1
if i+j ≥ arr.length	C3	(n-1)*(n_1-1)
return -1	C4	1
else	C5	(n-1)*(n_1-1)
push(myStack,arr[i+j])	C6	(n-1)*(n_1-1)
if arr[i+j] = x	C7	(n-1)*(n_1-1)
return i+j	C8	1
for int k = n2 to 1 (include) do:	C9	(n-1)*n_2
pop(myStack)	C10	(n-1)*(n_2-1)
i ← i + n1 – n2	C11	n-1
return -1	C12	1

$$
\begin{aligned}
T(n) = {} & c1 * n + c2 * n1n + c3 * n1(n-1) + c4 + (c5 + c6 + c7) * n1(n-1) + c8 \\
& + c9 * n2(n-1) + c10 * (n-1)(n2-1) + c11 * (n-1) + c12 \\
= {} & \quad n * n1(c2 + c3 + c5 + c6 + c7) + n * n2(c9 + c10) \\
& + n(c1 - c10 + c11) - n2 * (c9 + c10) - n1(c3 + c5 + c6 + c7) \\
& + c4 + c8 + c10 - c11 + c12 \\
= {} & a * n1 * n + b * n2 * n + c * n + d * n2 + e * n1 + f
\end{aligned}
$$

We will focus on the first part of the equation: $an1 * n + bn2 * n$ (the other part is negligible).

Lower Bound $\Omega(n)$:

In this case, the value we're searching for will be in the last index or won't appear at all. Furthermore, n2 = 0 so there won't be any backwards steps.
our equation now is: $n1 * n$.
we know that n1>0 and now we will prove that there are c1>0 and n0∈ N such that $an1 * n \geq c1 * n$ for all n>n0.
a is a specific constant therefore we can take c1 = $\frac{a}{4}$, n0 = 1.
$an1 * n \geq c1 * n \leftrightarrow an1 \geq c1 \rightarrow an1 \geq \frac{a}{4}$
Therefore the lower bound is $\Omega(n)$.

Upper Bound O(n*n1):

We have told that $A.length = \text{n} > n_1 > n_2 \geq 0$ therefore n2 at his worst case is n_1-1.

For that case we will get

$$T(n) = a * n1 * n + b * (n1 - 1) * n + c * n + d * (n1 - 1) + e * n1 + f =$$

$$(a + b) * n1 * n + (c - b) * n + (e + d) * n1 + f - d$$

the expression $d1 * n * n1 + d2 * n + d3 * n1 + d4$, the terms $d2 * n$, d3*n1 and d4 become negligible compared to $d1 * n1 * n$ when n approaches infinity.
Thus, the equation is: $dn1 * n$

We will prove that there are c2>0 and n0∈ N such that $d1 * n1 * n \leq c2 * n1 * n$ for all n>n0.
d1 is a specific constant therefore we can take c2 = d1+1 , n0 = 1.
$d1 * n1 * n \leq c2 * n1 * n \leftarrow\rightarrow d1 \leq c2 \rightarrow d1 \leq d1 + 1 \rightarrow 0 \leq 1$

Therefore, the upper bound is O(n1*n).

In conclusion, the lower bound of this function is $\Omega(\text{n})$ and the upper bound is O(n1*n).

2)

i)

We choose an implementation which save the first empty index on the fields section of the class (when inserting a value we add 1 to it and when deleting reduce 1).
Every action of insertion or deletion we push an object to the stack (which defined in another class) that indicate the action made , the index where the change happened and the value on this index.
When calling a backtrack action first of all we pop the head object on the stack.

- Insert backtrack (undo insertion):
  We know that the implementation of the insert method (on unsorted array) the value will be added in the first empty index (which we save).
  When backtracking the insertion we know that we need to remove the last value on our array.
  Each action of the algorithm access a specific index on the array (O(1)) , therefore this backtracking case is – O(1).

- Delete Backtrack (undo deletion):
  In the delete method, when we delete a value we put in its index the value at the end of the array.
  From the object we pulled from the stack we have the value we need to return and its previous index.
  When backtracking we replace the location of the value in the index we pulled from the stack to the first free index on the array (which we save). Now we take the value that have been deleted and return it in its previous index (which now is free).
  Each action of the algorithm access a specific index on the array (O(1)) , therefore this backtracking case is – O(1).

ii)

We choose an implementation which save the first empty index on the fields section of the class (when inserting a value we add 1 to it and when deleting reduce 1).
Every action of insertion or deletion we push an object to the stack (which defined in another class) that indicate the action made , the index where the change happened and the value on this index.
When calling a backtrack action first of all we pop the head object on the stack.

- Insert backtrack (undo insertion):
  We know that the implementation of the insert method (in sorted array) the value will be added in the right index of the array according to its value so the array will remain sorted.
  When backtracking the insertion we know that we need to move each value that is bigger than the value we inserted one index back. Then, we remove the firs free index (which we save).
  In the worst case, the value was inserted at the array's first index. Thus, we will have to move all (n-1) values one index back.
  Therefore , in the worst case the run time will be O(n).

- Delete Backtrack (undo deletion):
  We know that the implementation of the delete method (in sorted array) all the value greater than the one we deleted will move one index back. Then, we remove the first free index (which we save).
  When backtracking the deletion we know that we need to move each value that is bigger than the value we inserted one index forward, including the first free index.
  In the worst case, the value was deleted from the array's first index. Thus, we will have to move all (n-1) values one index forward.
  Therefore , in the worst case the run time will be O(n)

iii) we will choose an implementation that push the stack an object which save the node that have been deleted/inserted and an indicator that defined what action has been made.

•Insert backtrack (undo insertion):
When inserting a node to a BST it will always located as a leaf. Therefore when doing backtrack action we will need to delete a leaf.
When pulling the object that have been inserted from the stack all we need to do is change his parent's son (according to its key we will know which side) to null.
We get directly from the stack a pointer so no search is necessary and also to change a pointer is O(1).
Therefore this backtrack action is O(1).

•Delete Backtrack (undo deletion):
in this backtrack action we have three cases:

1) return a leaf that has been deleted:
   The node that will pop from the stack has a pointer to his parent- exactly where we need to put it back. All we have to do is to change the parent's son back to the node (according to its key we will know on which side).
   We get directly from the stack a pointer so no search is necessary and also to change a pointer is O(1).
   Therefore this backtrack case is O(1).

2) return a node that had 1 son:
   The node that will pop from the stack has one pointer to his parent and one pointer to his son. First, just like the previous case, we will change his parent's son back to the node popped from the stack (O(1)). Then, we will also change his son's parent to the same node (O(1)).
   Therefore this backtrack case is O(1).

3) Return a node that had 2 sons:
   in this delete action, we replace the node with its successor and push both of them into the stack.
   in the backtrack action: first the node will pop from the stack, and we replace his parent and 2 sons to point back at it. Then, the successor will pop. We will place it back in its previous location (case 1 or 2).
   Therefore this backtrack case is O(1).

iv) we will choose an implementation that push the stack an object which save the node that have been deleted/inserted and another node with indicator that defined its rotation's direction.

- Insert backtrack (undo insertion):
  just like in BST, When inserting a node to an AVL tree, it will always located as a leaf. Therefore when doing backtrack action we will need to delete a leaf.
  in this backtrack action we have two cases:

  1) no rotation is needed (the tree remains balanced)- we pop the node from the stack and only have to change his parent to point to null O(1).
  then, in the worst case, we have to update each node's height in the course to the root – h nodes. This action will "cost" O(logn).
  Therefore this backtrack case is O(1)+O(logn) → O(logn).

  2) rotation is needed (the tree isn't balanced after the insertion)- in this case there were two rotations (on the deepest non-balanced node).
  in the backtrack action- first, the node that has been rotated will pop from the stack and we will rotate it (according to the direction we saved), and again on the node we want to delete O(1).
  then, in the worst case, we have to update each node's height in the course to the root – h nodes. This action will "cost" O(logn).
  in the end, we only have to change the node's pointers to null O(1).
  Therefore this backtrack case is O(1)+O(logn)+O(1) → O(logn).

- Delete Backtrack (undo deletion):
  in this backtrack action we have two cases:

  1) no rotation is needed (the tree remains balanced):
  first, we will change the parent of the node that will pop from the stack to point back to the node. If it has sons, we will change their parent to be the node. Then, in the worst case, we have to update each node's height in the course to the root – h nodes. This action will "cost" O(logn).
  therefore this backtrack case is O(1)+O(logn) → O(logn).

  2) rotation is needed (the tree isn't balanced after the deletion)-
  in the worst case, we will have to make two rotations on the nodes from the stack.
  first, we will have to change the pointers to the nodes from the stack →O(1). Then, the "cost" of the rotation actions depends on the tree height – O(logn+logn) → O(logn)
  now we will return on the same actions that we have mentioned in case 1.
  Therefore this backtrack case is O(logn)+O(1) → O(logn).