```
1 if n = 1 then
2    Print the message "Mystery solved"
3 else
4    j←1;
5    while j ≤ n - 1 do
6         if a_{j-1} > a_j then
7              swap a_{j-1} and a_j
8         Print n, j and the array elements
9         j←j + 1;
10   Mystery(A, n-1);
```

(a) Using A = (9,5,11,3,2) as the input array, hand-trace Mystery, showing the output.

Line	A	n	j	Recursive Call	Boolean Result	Output
1	(9,5,11,3,2)	5	–	–	False	–
4	(9,5,11,3,2)	5	1	–	–	–
5	(9,5,11,3,2)	5	1	–	True	–
6	(9,5,11,3,2)	5	1	–	True	–
7	(5,9,11,3,2)	5	1	–	–	–
8	(5,9,11,3,2)	5	1	–	–	5 1 [5, 9, 11, 3, 2]
9	(5,9,11,3,2)	5	2	–	–	–
5	(5,9,11,3,2)	5	2	–	True	–
6	(5,9,11,3,2)	5	2	–	False	–
8	(5,9,11,3,2)	5	2	–	–	5 2 [5, 9, 11, 3, 2]
9	(5,9,11,3,2)	5	3	–	–	–
5	(5,9,11,3,2)	5	3	–	True	–
6	(5,9,11,3,2)	5	3	–	True	–
7	(5,9,3,11,2)	5	3	–	–	5 3 [5, 9, 3, 11, 2]
8	(5,9,3,11,2)	5	3	–	–	–
9	(5,9,3,11,2)	5	4	–	–	–
5	(5,9,3,11,2)	5	4	–	True	–
6	(5,9,3,11,2)	5	4	–	True	–
7	(5,9,3,2,11)	5	4	–	–	–
8	(5,9,3,2,11)	5	4	–	–	5 4 [5, 9, 3, 2, 11]
9	(5,9,3,2,11)	5	5	–	–	–
5	(5,9,3,2,11)	5	5	–	False	–
10	(5,9,3,2,11)	5	5	Mystery((5,9,3,2,11), 4)	–	–
Now Processing: Mystery((5, 9, 3, 2, 11), 4)						
1	(5,9,3,2,11)	4	–	–	False	–
4	(5,9,3,2,11)	4	1	–	–	–
5	(5,9,3,2,11)	4	1	–	True	–
6	(5,9,3,2,11)	4	1	–	False	–
8	(5,9,3,2,11)	4	1	–	–	4 1 [5, 9, 3, 2, 11]

9	(5,9,3,2,11)	4	2	–	–	–
5	(5,9,3,2,11)	4	2	–	True	–
6	(5,9,3,2,11)	4	2	–	True	–
7	(5,3,9,2,11)	4	2	–	–	–
8	(5,3,9,2,11)	4	2	–	–	4 2 [5, 3, 9, 2, 11]
9	(5,3,9,2,11)	4	3	–	–	–
5	(5,3,9,2,11)	4	3	–	True	–
6	(5,3,9,2,11)	4	3	–	True	–
7	(5,3,2,9,11)	4	3	–	–	–
8	(5,3,2,9,11)	4	3	–	–	4 3 [5, 3, 2, 9, 11]
9	(5,3,2,9,11)	4	4	–	–	–
5	(5,3,2,9,11)	4	4	–	False	–
10	(5,3,2,9,11)	4	4	Mystery((5,3,2,9,11), 3)	–	–
Now Processing: **Mystery**((**5**, **3**, **2**, **9**, **11**), **3**)						
1	(5,3,2,9,11)	3	–	–	False	–
4	(5,3,2,9,11)	3	1	–	–	–
5	(5,3,2,9,11)	3	1	–	True	–
6	(5,3,2,9,11)	3	1	–	True	–
7	(3,5,2,9,11)	3	1	–	–	–
8	(3,5,2,9,11)	3	1	–	–	3 1 [3, 5, 2, 9, 11]
9	(3,5,2,9,11)	3	2	–	–	–
5	(3,5,2,9,11)	3	2	–	True	–
6	(3,5,2,9,11)	3	2	–	True	–
7	(3,2,5,9,11)	3	2	–	–	–
8	(3,2,5,9,11)	3	2	–	–	3 2 [3, 2, 5, 9, 11]
9	(3,2,5,9,11)	3	3	–	–	–
5	(3,2,5,9,11)	3	3	–	False	–
10	(3,2,5,9,11)	3	3	Mystery((3,2,5,9,11), 2)	–	–
Now Processing: **Mystery**((**3**, **2**, **5**, **9**, **11**), **2**)						
1	(3,2,5,9,11)	2	–	–	False	–
4	(3,2,5,9,11)	2	1	–	–	–
5	(3,2,5,9,11)	2	1	–	True	–
6	(3,2,5,9,11)	2	1	–	True	–
7	(2,3,5,9,11)	2	1	–	–	–
8	(2,3,5,9,11)	2	1	–	–	2 1 [2, 3, 5, 9, 11]
9	(2,3,5,9,11)	2	2	–	–	–
5	(2,3,5,9,11)	2	2	–	False	–
10	(2,3,5,9,11)	2	2	Mystery((2,3,5,9,11), 1)	–	–
Now Processing: **Mystery**((**2**, **3**, **5**, **9**, **11**), **1**)						
1	(2,3,5,9,11)	1	–	–	True	–
2	(2,3,5,9,11)	1	–	–	–	Mystery Solved

Final state of A: (2, 3, 5, 9, 11)

(b) Determine the running time T(n) as a function of the number of comparisons made on line 6, and then indicate its time and space efficiency (i.e., O and Ω bounds).

Line 6: if $a_{j-1} > a_j$
Line 6 is inside of a while loop:

$$\text{while } j \leq n - 1 \text{ do}$$

Since j is always initialized to 1 we can set this as the smallest possible value that j can take on. Moreover, from the while loop condition we see that line 6 is executed as long as $j \leq n - 1$ is true. Knowing that j increments by 1 always we can therefore deduce that j runs from 1 to $n - 1$ (inclusive).

Furthermore, we can model the number of function calls by n.
The maximum number of function calls (recursive call stack size) is always n.
This is because the recursive call is in the else statement of the function.

Therefore, we see that the number of times line 6 is executed is as follows:

$$(n - 1) + (n - 2) + (n - 3) + \cdots 1$$

Where each operand is a recursive call

To sum all the terms up:

$$\sum_{k=1}^{n} k = \frac{1}{2}(n^2 - n)$$

Time complexity

$$\mathbf{T(n) = \frac{1}{2}(n^2 - n)}$$

$$O(T(n)) = \mathbf{O(n^2)}$$
$$\Omega(T(n)) = \mathbf{\Omega(n^2)}$$

Space complexity

Maximum size of the recursive call stack = n
$$\mathbf{O(n)}$$

(c) Determine the running time T(n) as a function of the number of swaps made on line 7, and indicate its time and space efficiency.

Line 7: swap a_{j-1} and a_j
Line 7 is inside of a while loop:

$$\text{while } j \leq n - 1 \text{ do}$$

Swaps are only made if $a_{j-1} > a_j$. Hence the order of elements in list A matters.
The worst case scenario is when the array A is in descending order
The best case scenario is when the array A is in ascending order

Best Case:
In the best case scenario, the if condition on line 6 will always fail and so no comparisions will be made. Line 7 will be executed 0 times in the best case.

Worst Case:
If the array is in descending order, the if condition on line 6 will always return true Since it is nested in a while statement running from 1 to $n - 1$. There will be $n - 1$ swaps made per function call. Knowing that there are always n function calls we can model the number of swaps made in the worst case as follows:

$$(n - 1) + (n - 2) + (n - 3) + \cdots 1$$

Where each operand is a recursive call

To sum all the terms up:

$$\sum_{k=1}^{n} k = \frac{1}{2}(n^2 - n)$$

Time complexity

$$\mathbf{T(n) = \frac{1}{2}(n^2 - n)}$$

$$\text{Worst Case} = O(T(n)) = \mathbf{O(n^2)}$$
$$\text{Best Case} = 0 = \mathbf{\Omega(1)}$$

Space complexity

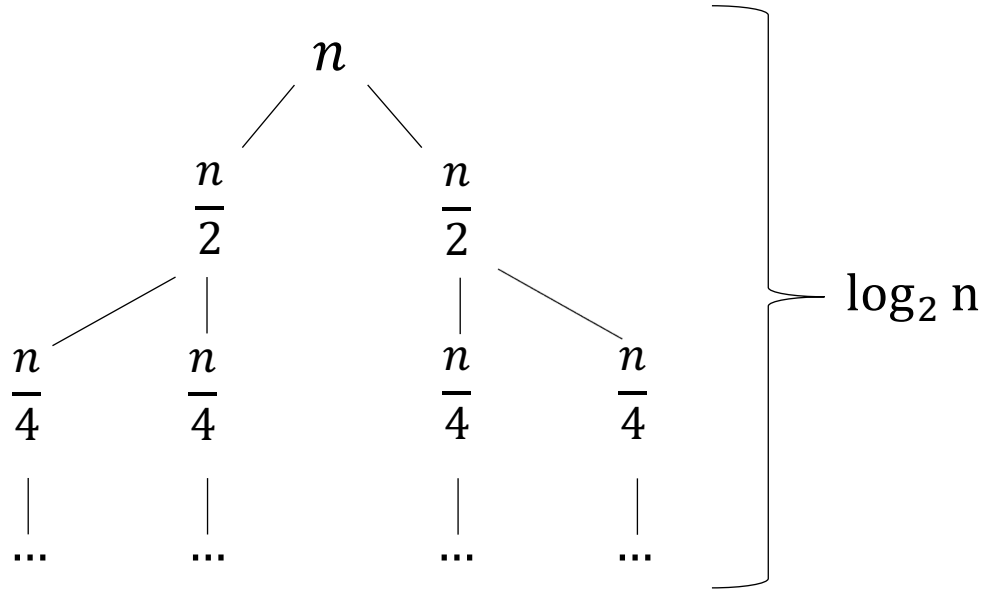
$$\text{Maximum size of the recursive call stack} = n$$
$$\mathbf{O(n)}$$

(d) Suggest an improvement, or a better algorithm altogether, and indicate its time and space efficiency. If you cannot do it, explain why it cannot be done?

Any attempt to make the algorithm more efficient will be redundant as the time and space complexity will again be $O(n^2)$ and $O(n)$ respectively.

Instead, a new sorting algorithm will be used. Merge Sort. This algorithm will divide the array into smaller pieces – sort the smaller pieces – then merge them

```
def mergeSort(arr):

     //Spliting the array into smaller pieces
     if arr.length > 1:
          middle = int(arr.length / 2)
          left = arr[0:mid]
          right = arr[mid:arr.length]
          mergeSort(left)
          mergeSort(right)

          i = j = k = 0


          //Add the elements from the left and right
            sub arrays in order
          while i < left.length and j < right.length:
               if left [i] < right[j]:
                    arr[k] = left[i]
                    i += 1
               else:
                    arr[k] = right[j]
                    j += 1
               k += 1

          //Add remaining elements of the subarrays
          while i < left.length:
               arr[k] = left[i]
               i += 1
               k += 1

          //Add remaining elements of the subarrays
          while j < right.length:
               arr[k] = right[j]
               j += 1
               k += 1
```



The tree above shows the reccurence relation of the merge sort algorithm. In addition to this we have to factor in the time taken to merge the subarrays together. This comes out to be n.

Therefore the reccurence relation can be modelled as follows:

$$T(n) = T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) + n$$

$$T(n) = 2T\left(\frac{n}{2}\right) + n \quad \text{Equation: } \mathbf{1}$$

$$T\left(\frac{n}{2}\right) = 2T\left(\frac{n}{4}\right) + \frac{n}{2} \quad \text{Equation: } \mathbf{2}$$

$*$ Sub 2 in 1 $*$

$$T(n) = 2\left[2T\left(\frac{n}{4}\right) + \frac{n}{2}\right] + \frac{n}{2}$$

$$T(n) = 2^2T\left(\frac{n}{2^2}\right) + 2n \quad \text{Equation: } \mathbf{4}$$

$$T\left(\frac{n}{4}\right) = 2T\left(\frac{n}{8}\right) + \frac{n}{4} \quad \text{Equation: } \mathbf{3}$$

$*$ Sub 3 in 4 $*$

$$T(n) = 2^3T\left(\frac{n}{2^3}\right) + 3n$$

$$T(n) = 2^i T\left(\frac{n}{2^i}\right) + in \quad \text{Equation: } \mathbf{5}$$

Base Case: $T(1) = 1$

$$\frac{n}{2^i} = 1$$

$$\boldsymbol{n = 2^i}$$
$$\boldsymbol{i = \log_2 n}$$

$*$ Sub in equation 5 $*$

$$T(n) = n * 1 + \log_2 n * n$$
$$T(n) = n + n\log_2 n$$

Therefore, the algorithm's time complexity is $\mathbf{O(n \log_2 n)}$

Since 2 sub-arrays are used the space complexity of this sorting algorithm is of linear order. The maximum size of a single sub-array is $\frac{n}{2}$ and since 2 sub-arrays are used we have $2 * \frac{n}{2} = n$

Therefore, the algorithm's space complexity is $\mathbf{O(n)}$

(e) Can Mystery be converted into an iterative algorithm? If it cannot be done, explain why; otherwise, do it and determine its running time complexity in terms of the number of comparisons of the array elements.

```
def func(arr):

    n = arr.length

    FOR i 0 to n:
        FOR j 0 to n-i-1:
            if arr[j] > arr[j+1]:
               swap(arr[j], arr[j+1])
```

The worst case for this algorithm is when the array passed is in descending order. The table below will be used to model the time complexity of this iterative algorithm. Note: the table below shows descending order arrays of size n.

n	Comparisions
2	1
3	3
4	6
5	10
6	15
7	21
8	28
9	36
10	45

$$\mathrm{T(n)} = \sum_{i=1}^{n} i = \frac{n(n-1)}{2}$$

Therefore, the time complexity of this algorithm is $\mathbf{O(n^2)}$

Limit Asymptotic Theorem

$$\lim_{n\to\infty}\frac{f(n)}{g(n)}=L\quad\begin{cases}if\ L=0\ then\ f(n)\in O(g(n))\\ if\ L=c\ then\ f(n)\in\theta(g(n))\\ if\ L=\infty\ then\ f(n)\in\Omega(g(n))\end{cases}$$

2a) $\mathbf{f(n)=4nlog(n)+n^2, g(n)=log\,(n)}$

$$\lim_{n\to\infty}\frac{4n\log(n)+n^2}{\log(n)}=\lim_{n\to\infty}\frac{\log(n)\left[4n+\frac{n^2}{\log(n)}\right]}{\log(n)}=\lim_{n\to\infty}4n+\frac{n^2}{\log(n)}$$

$$\lim_{n\to\infty}4n+\lim_{n\to\infty}\frac{n^2}{\log(n)}=\infty+\infty$$

$\lim_{n\to\infty}\frac{f(n)}{g(n)}=\infty$ Therefore, $\mathbf{f(n)\ is\ \Omega(g(n))}$

2b) $\mathbf{f(n)=8log(n^2)\,, g(n)=(log(n))^2}$

$$\lim_{n\to\infty}\frac{8\log(n^2)}{(\log(n))^2}=\lim_{n\to\infty}16*\frac{\log(n)}{(\log(n))^2}=\lim_{n\to\infty}16*\frac{1}{\log\,(n)}=0$$

$\lim_{n\to\infty}\frac{f(n)}{g(n)}=0$ Therefore, $\mathbf{f(n)\ is\ O(g(n))}$

2c) $\mathbf{f(n)=log(n^2)+n^3\,, g(n)=log(n)+3}$

$$\lim_{n\to\infty}\frac{\log(n^2)+n^3}{\log(n)+3}=\lim_{n\to\infty}\frac{2\log(n)+n^3}{\log(n)+3}=\lim_{n\to\infty}\frac{\log\,(n)\left(2+\frac{n^3}{\log(n)}\right)}{\log(n)\left[1+\frac{3}{\log(n)}\right]}$$

$$\lim_{n\to\infty}\frac{2+\frac{n^3}{\log(n)}}{1+\frac{3}{\log(n)}}=\frac{\lim_{n\to\infty}2+\frac{n^3}{\log(n)}}{\lim_{n\to\infty}1+\frac{3}{\log(n)}}=\frac{\lim_{n\to\infty}2+\lim_{n\to\infty}\frac{n^3}{\log(n)}}{1}=\frac{\infty}{1}=\infty$$

$$\left[\lim_{n\to\infty}\frac{n^3}{\log(n)}=\lim_{n\to\infty}\frac{3n^2}{\frac{1}{n}}=\infty\right]$$

$\lim_{n\to\infty}\frac{f(n)}{g(n)}=\infty$ Therefore, $\mathbf{f(n)\ is\ \Omega(g(n))}$

2d) $\mathbf{f(n) = n\sqrt{n} + \log(n)\,, g(n) = \log(n^2)}$

$$\lim_{n\to\infty} \frac{n\sqrt{n} + \log(n)}{\log(n^2)} = \lim_{n\to\infty} \frac{n^{1.5} + \log(n)}{2\log(n)} = \lim_{n\to\infty} \frac{\log(n)\left[\frac{n^{1.5}}{\log(n)} + 1\right]}{2\log(n)}$$

$$\lim_{n\to\infty} \frac{\frac{n^{1.5}}{\log(n)} + 1}{2} = \frac{1}{2}\lim_{n\to\infty} \frac{n^{1.5}}{\log(n)} + 1 = \frac{1}{2}\lim_{n\to\infty} \frac{n^{1.5}}{\log(n)} + \lim_{n\to\infty} 1$$

$$\lim_{n\to\infty} \frac{n^{1.5}}{\log(n)} * \text{L'Hopital Rule} * = \lim_{n\to\infty} \frac{1.5n^{0.5}}{\frac{1}{n}} = \infty$$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty \qquad \text{Therefore, } \mathbf{f(n)\ is\ \Omega(g(n))}$$

2e) $\mathbf{f(n) = 2^n + 10^n\,, g(n) = 10n^2}$

$$\lim_{n\to\infty} \frac{2^n + 10^n}{10n^2} = \frac{1}{10}\lim_{n\to\infty} \frac{2^n + 10^n}{n^2} = \frac{1}{10}\lim_{n\to\infty} \frac{2^n}{n^2} + \lim_{n\to\infty} \frac{10^n}{n^2}$$

$$\frac{1}{10}\lim_{n\to\infty} \frac{2^n}{n^2} + \lim_{n\to\infty} \frac{10^n}{n^2}$$

$$\lim_{n\to\infty} \frac{2^n}{n^2} * \text{L'Hopital Rule} * = \lim_{n\to\infty} \frac{2^n \ln(2)}{2n} = \infty$$

$$\lim_{n\to\infty} \frac{10^n}{n^2} * \text{L'Hopital Rule} * = \lim_{n\to\infty} \frac{10^n \ln(10)}{2n} = \infty$$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty \qquad \text{Therefore, } \mathbf{f(n)\ is\ \Omega(g(n))}$$

2g) $\mathbf{f(n) = \log^2(n)\,, g(n) = \log(n)}$

$$\lim_{n\to\infty} \frac{\log^2(n)}{\log(n)} = \lim_{n\to\infty} \frac{\left(\log(n)\right)^2}{\log(n)} = \lim_{n\to\infty} \frac{\left(\log(n)\right)^2}{\log(n)} = \lim_{n\to\infty} \log(n) = \infty$$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty \qquad \text{Therefore, } \mathbf{f(n) \text{ is } \Omega(g(n))}$$

2h) $\mathbf{f(n) = n\,, g(n) = \log^2(n)}$

$$\lim_{n\to\infty} \frac{n}{\log^2(n)} = \lim_{n\to\infty} \frac{n}{(\log(n))^2} * \text{L'Hopital Rule} *$$

$$\lim_{n\to\infty} \frac{n}{\log^2(n)} = \lim_{n\to\infty} \frac{1}{\frac{2\ln(n)}{n}} = \lim_{n\to\infty} \frac{\frac{1}{2}}{n} = \infty$$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty \qquad \text{Therefore, } \mathbf{f(n) \text{ is } \Omega(g(n))}$$

2i) $\mathbf{f(n) = \sqrt{n}\,, g(n) = \log(n)}$

$$\lim_{n\to\infty} \frac{\sqrt{n}}{\log(n)} = \lim_{n\to\infty} \frac{0.5n^{-0.5}}{\frac{1}{n}} * \text{L'Hopital Rule} *$$

$$\lim_{n\to\infty} 0.5n^{0.5} = \infty$$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty \qquad \text{Therefore, } \mathbf{f(n) \text{ is } \Omega(g(n))}$$

2j) $\mathbf{f(n) = 4^n\,, g(n) = 5^n}$

$$\lim_{n\to\infty} \frac{4^n}{5^n} = \lim_{n\to\infty} \left(\frac{4}{5}\right)^n = 0$$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \qquad \text{Therefore, } \mathbf{f(n) \text{ is } O(g(n))}$$

2k) $\mathbf{f(n) = 3^n}$, $\mathbf{g(n) = n^n}$

$\lim_{n\to\infty} \frac{3^n}{n^n} = 0$ * Growth Rate of $n^n \gg 3^n$ *

$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$ Therefore, $\mathbf{f(n)}$ **is** $\mathbf{O(g(n))}$