1 a)
This algorithm searches through every element in the matrix in order until it has found a match.

Pseudo code:

```
FindElementD(A,n, p)
        For i = 0; i <= n; i++
                For j = 0; j <= n; j++
                        If A[i][j] == p return true
        Return false
```

Fundamental operation: If A[i][j] == p return true
This is checking if the element against a search term.

The worst case is when the element is not in the matrix.
In the worst case the outside loop is executed n times and the inner loop is executed n times

$$t(n) = \Sigma n \times \Sigma n$$

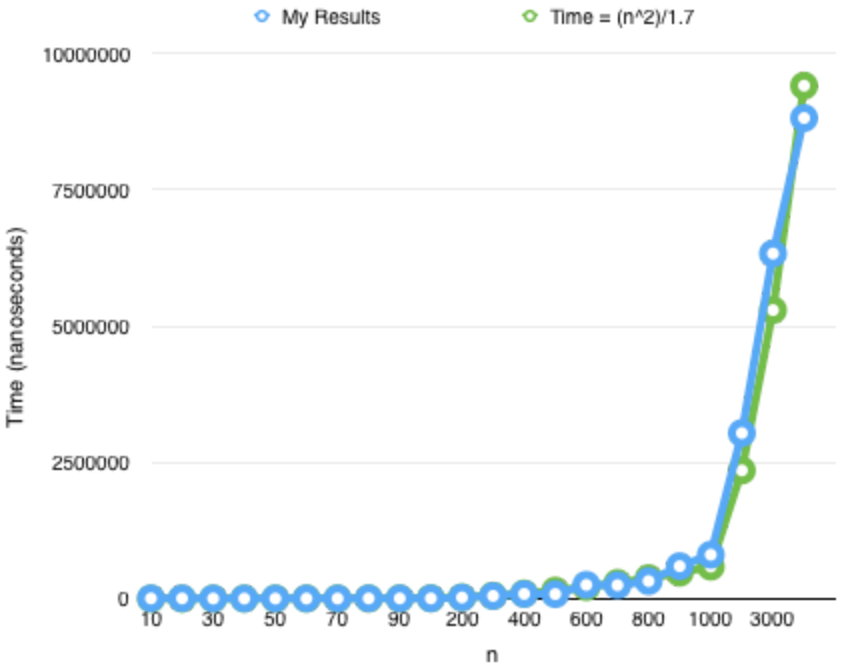
So

$$O(n^2)$$

This is an example of a worst case of size 7x7 if the search term is = 10:

1 2 3 4 5 6 7
1 2 3 4 5 6 7
1 2 3 4 5 6 7
1 2 3 4 5 6 7
1 2 3 4 5 6 7
1 2 3 4 5 6 7
1 2 3 4 5 6 7

A worst case can be generated by simply not including the search term in the matrix.



This graph plots the results from my implementation and test of the worst case, the blue line represents my results and the green line is a function of order n^2. They follow the same shape which shows that the calculated worst case was correct.

b)
This algorithm checks each row in the matrix to see if the search term is in the range of the row.
If the search term is in the row a binary search is run on the row, otherwise it moves on to check the range of the next row.
The binary search finds which half of the row could possibly contain the search term and checks the middle element in this half. The other half is eliminated from the search, this is repeated until the result is found.
If the search term is not found it moves on to the next row. This continues until the correct element is found or the last row is reached.

Pseudo code:

```
FindElementD1(A, n, p)
        For int i = 0; i < n; i++
                If A[i][0] <= p && p <= A[i][n-1]
                        int first = 0
                        int last = n - 1
                        While last >= first
                                int centre = (first + last) / 2
                                If A[centre] == p
                                        return true
                                If A[centre] < p
                                        first = centre + 1
                                If A[centre] > p
                                        last = centre - 1
        return false
```

Fundamental operation: if A[centre] == p
This is checking if the element against a search term.

The worst case is when the element is not in the matrix but the search value is between the highest and lowest value of each row. Since the worst case means each row is checked the binary search is ran n times:

$$t(n) = \Sigma n$$

The binary search halves the size each time so how many times do you divide by two before you only have one element is how many times the fundamental operation is ran:

$$1 = \frac{n}{2^{t(n)}}$$

$$2^{t(n)} = n$$

$$t(n) = \Sigma log_2 n$$

Bringing the two together gives:

$$t(n) = \Sigma n \times \Sigma log_2 n$$

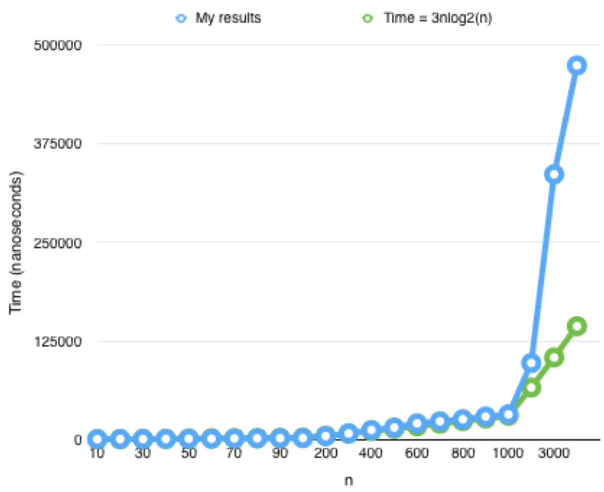
So

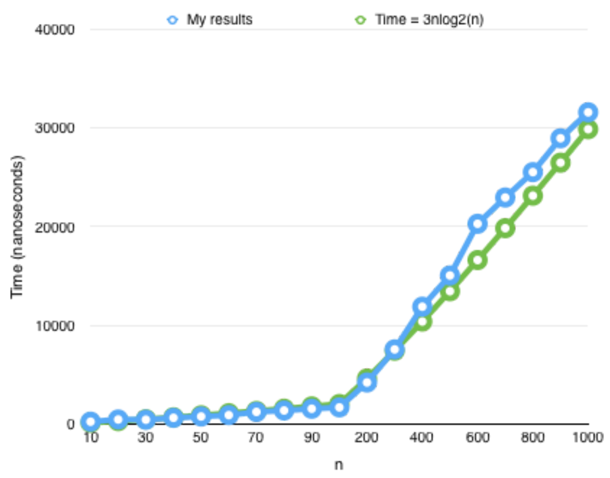
$$O(nlog_2 n)$$

A worst case can be generated by not including the search term in the matrix but making sure each row is searched. I did this by filling each row of the matrix with its index number but for the final value of each row I added one. I then made the search term the last index.

An example of the array built for the tests of algorithm D1 this example is of size n=10 the search term will be 9:

0	1	2	3	4	5	6	7	8	10
0	1	2	3	4	5	6	7	8	10
0	1	2	3	4	5	6	7	8	10
0	1	2	3	4	5	6	7	8	10
0	1	2	3	4	5	6	7	8	10
0	1	2	3	4	5	6	7	8	10
0	1	2	3	4	5	6	7	8	10
0	1	2	3	4	5	6	7	8	10
0	1	2	3	4	5	6	7	8	10
0	1	2	3	4	5	6	7	8	10





These graphs plot the results from my implementation and test of the worst case, the blue lines represents my results and the green lines represent a function of order $nlog_2 n$. The graph on the left shows all the results, my results where n > 2000 don't quite fit with the other

line. The reason for the line not fitting could be something that is going on in the background on my computer messing up the results. Because of this I've shown another graph with results where n > 1000 removed on the right. It can clearly be seen that the two lines are very similar which shows that the calculated worst case was correct.

c)
This algorithm begins by checking the element in the bottom left hand corner, if the search term is lower than the checked element the next element to check will be the one above it. If the checked element is smaller than the search term the next element to check will be the one to the right.
This continues until the top or right edge of the matrix is reached and the algorithm moves in the direction of the reached edge.

Pseudo code:

```
FindElementD1(A,n, p)
        i = 0;
        j = n;
        While i <= n >= j
                If p == A[i][j]
                        Return true
                If p < A[i][j]
                        j -= 1
                Else
                        i += 1
        Return false
```

Fundamental operation: If p == A[i][j]
This is checking if the element against a search term.

The worst case is when the element is one less than that of the top right corner of the matrix but is not in the matrix. This means the fundamental operation of the algorithm is the length of the bottom and right edge so:

$$t(n) = \Sigma n + n$$

So

$$O(2n)$$

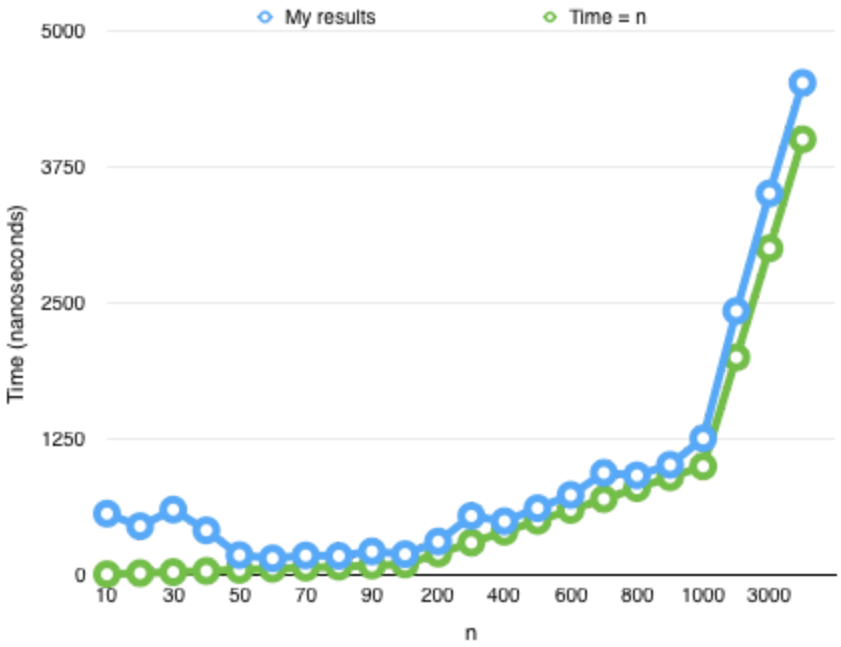
$$O(n)$$

A worst case can be generated by making each element equal to the element column multiplied by ten and then adding the element row. Then add one to the element in the top

right hand corner. The search term then has to be equal to what the top right corner would have been before adding the one.

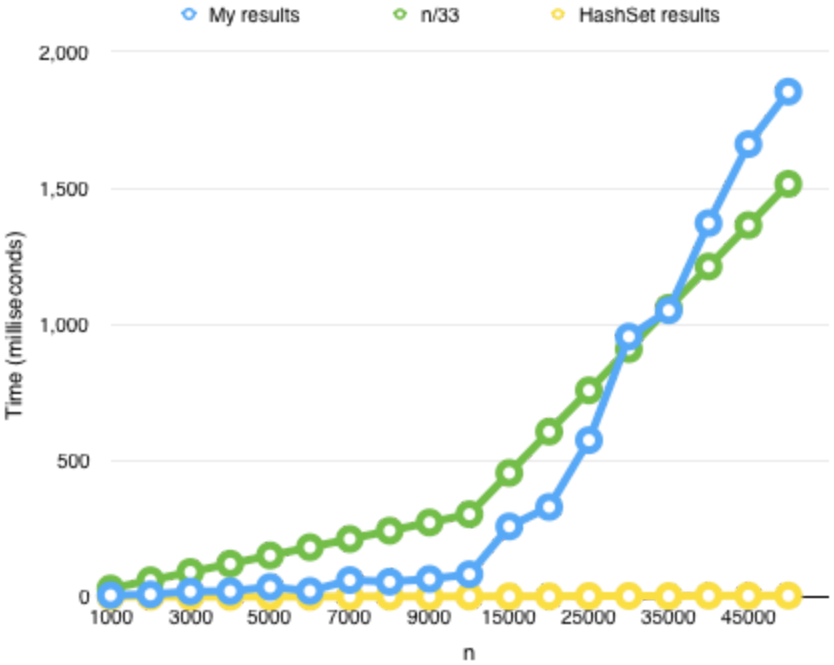
An example of the array built for the test of algorithm D2 this example is of size n=10 the search term will be 90:

0	10	20	30	40	50	60	70	80	91
1	11	21	31	41	51	61	71	81	91
2	12	22	32	42	52	62	72	82	92
3	13	23	33	43	53	63	73	83	93
4	14	24	34	44	54	64	74	84	94
5	15	25	35	45	55	65	75	85	95
6	16	26	36	46	56	66	76	86	96
7	17	27	37	47	57	67	77	87	97
8	18	28	38	48	58	68	78	88	98
9	19	29	39	49	59	69	79	89	99



This graph plots the results from my implementation and test of the worst case, the blue line represents my results and the green line is a function of order n . The lines are a very similar shape which shows that the calculated worst case was correct. The beginning of the graph does not match so well, this is due to Java optimising the function call after it recognises it is being ran so many times.

2)5)
Timing the ArrayHashTable against HashSet.



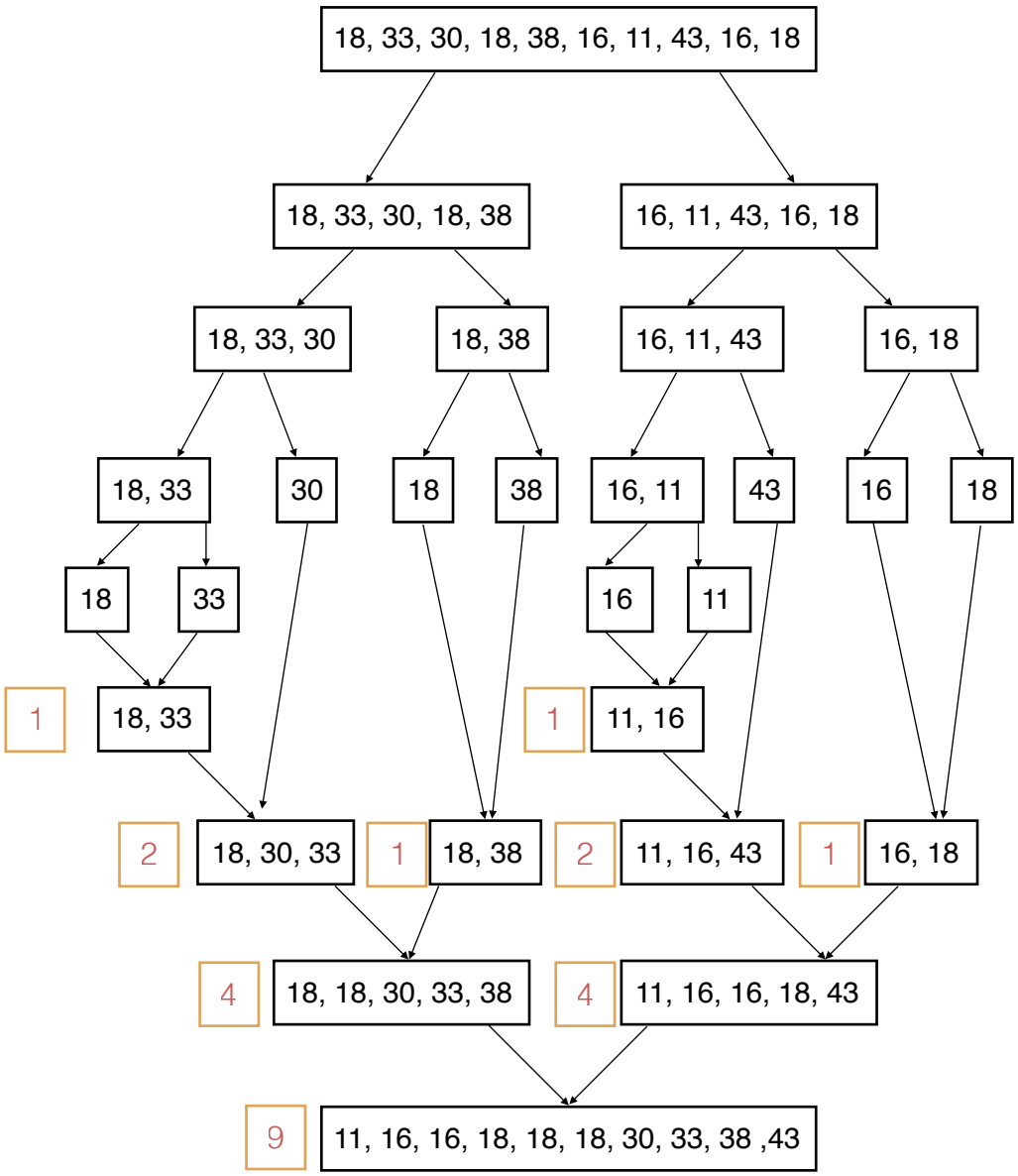
On the graph above you can see that My results loosely follow a line that is of order n (the green line being of order n).

The built in Java class HashSet (represented by the yellow line) is clearly a lot faster than my implementation of a HashTable, this is because HashSet uses a linked list instead of an array meaning we don't have to do all the array manipulation such as doubling or halving the array size (which requires relatively expensive copying of arrays) because linked lists are so much more flexible as they don't have a predetermined size.

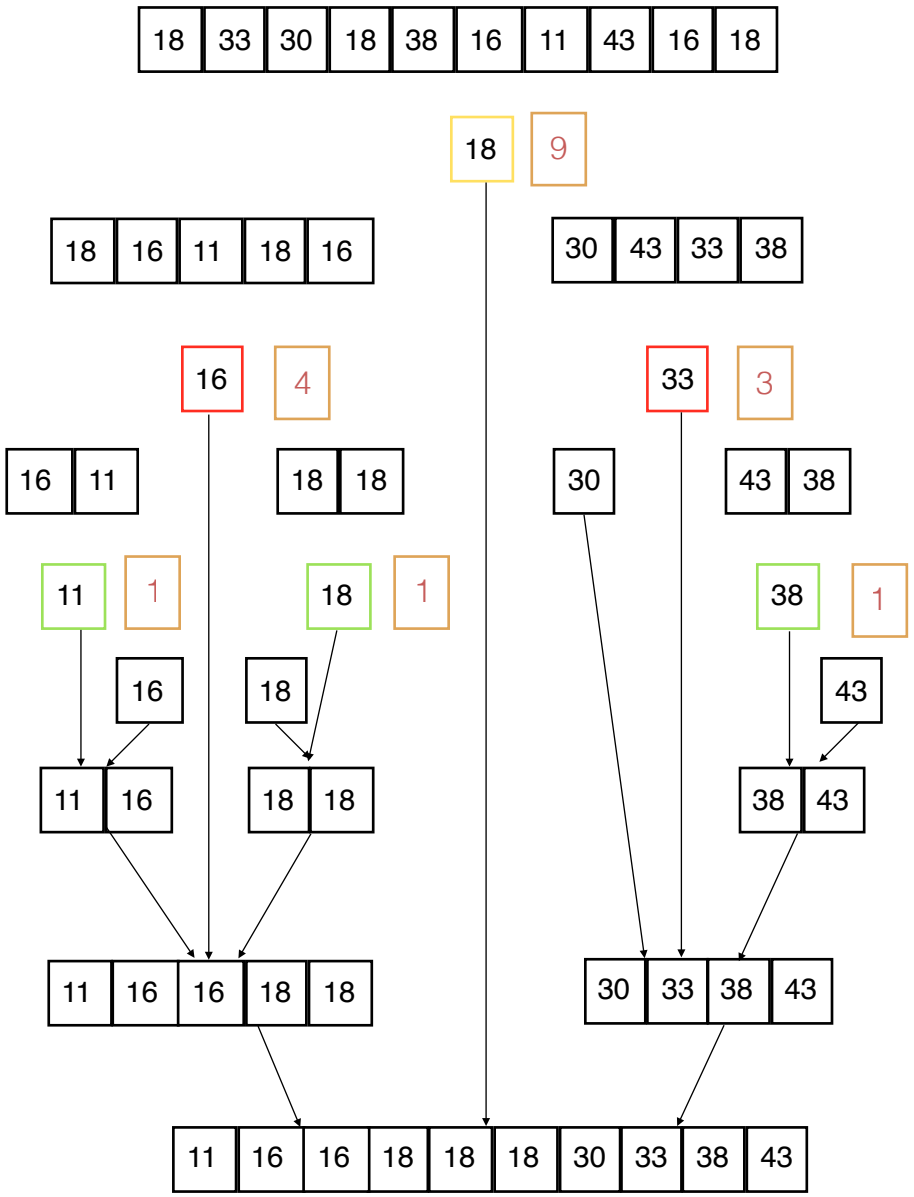
Merge Sort
Q3 1)

Total comparisons:

25



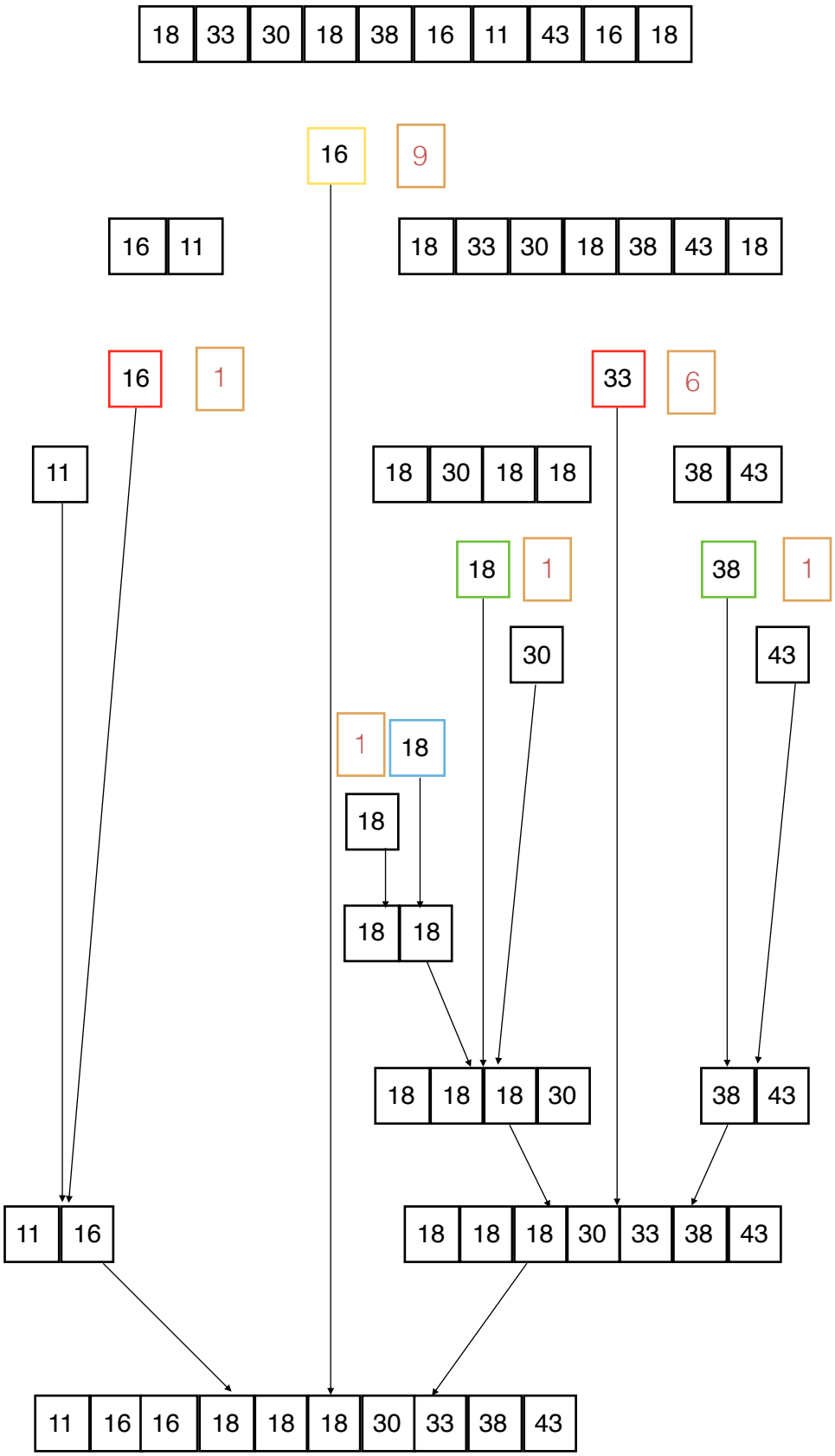
Quick Sort 2a)



Total comparisons:

19

2b)



Median of 3 pivots:
(3 elements
chosen at random)

33 16 11

16 11 16

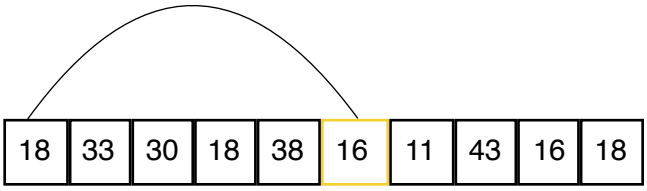
33 18 38

30 18 18

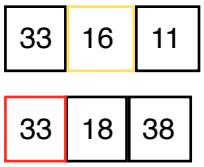
Total comparisons:

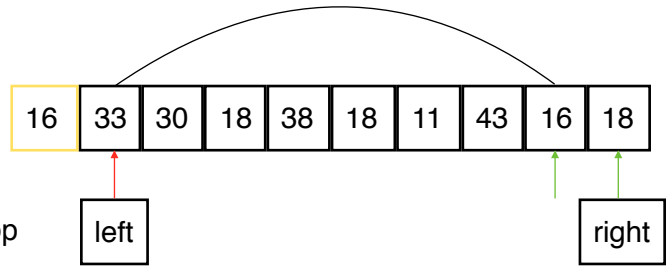
19

Partitioning algorithm:



Median of 3 pivots:
(3 elements
chosen at random)



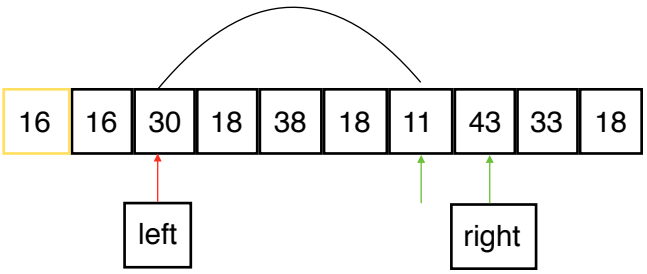


33>16 so stop

18>16 so go
16<16 so stop

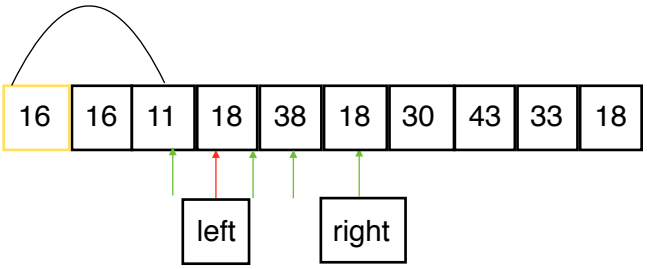
Total comparisons:

23



30>16 so stop

43>16 so go
11<16 so stop

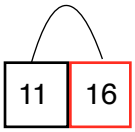


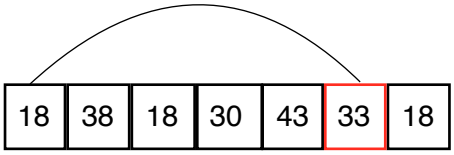
18>16 so stop

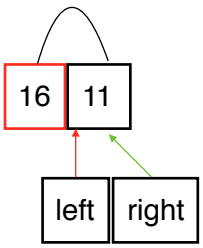
18>16 so go
38>16 so go
18>16 so go

crossed over so stop
and swap with right

16

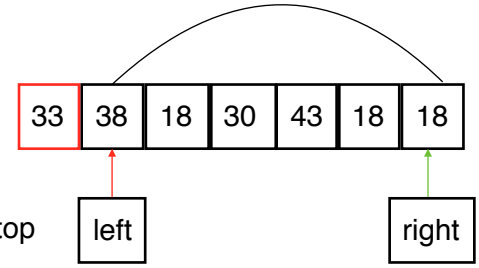






11<16 so go

crossed over so stop
and swap with right

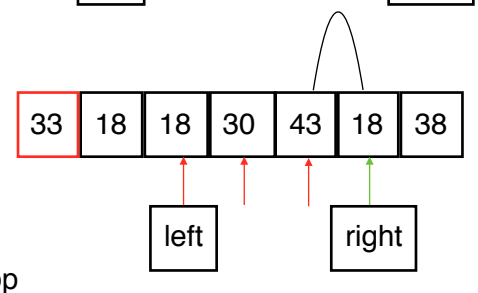


38>33 so stop

18<33 so stop

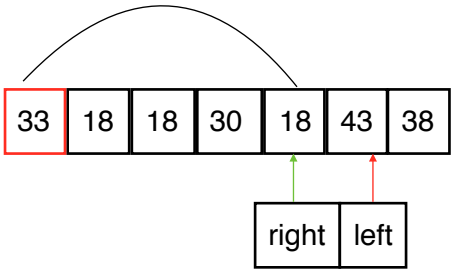
16

11



18<33 so go
30<33 so go
43>33 so stop

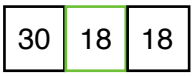
18<33 so stop

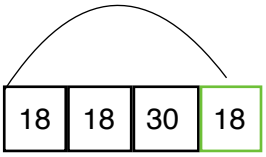


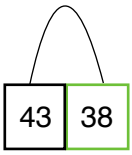
crossed over so stop
and swap with right

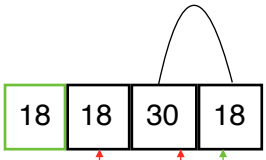
33

Median of 3 pivots:
(3 elements
chosen at random)







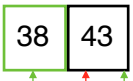


18<18 so go
30>18 so stop

left

right

18<18 so stop

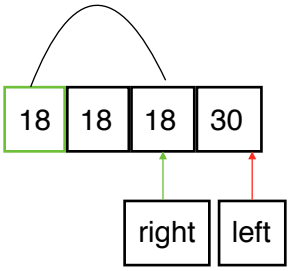


43>38 so stop

left

ght

43>38 so go



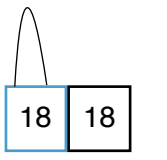
crossed over so stop
and swap with right

38

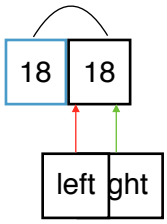
43

crossed over so stop
and swap with right

18



30

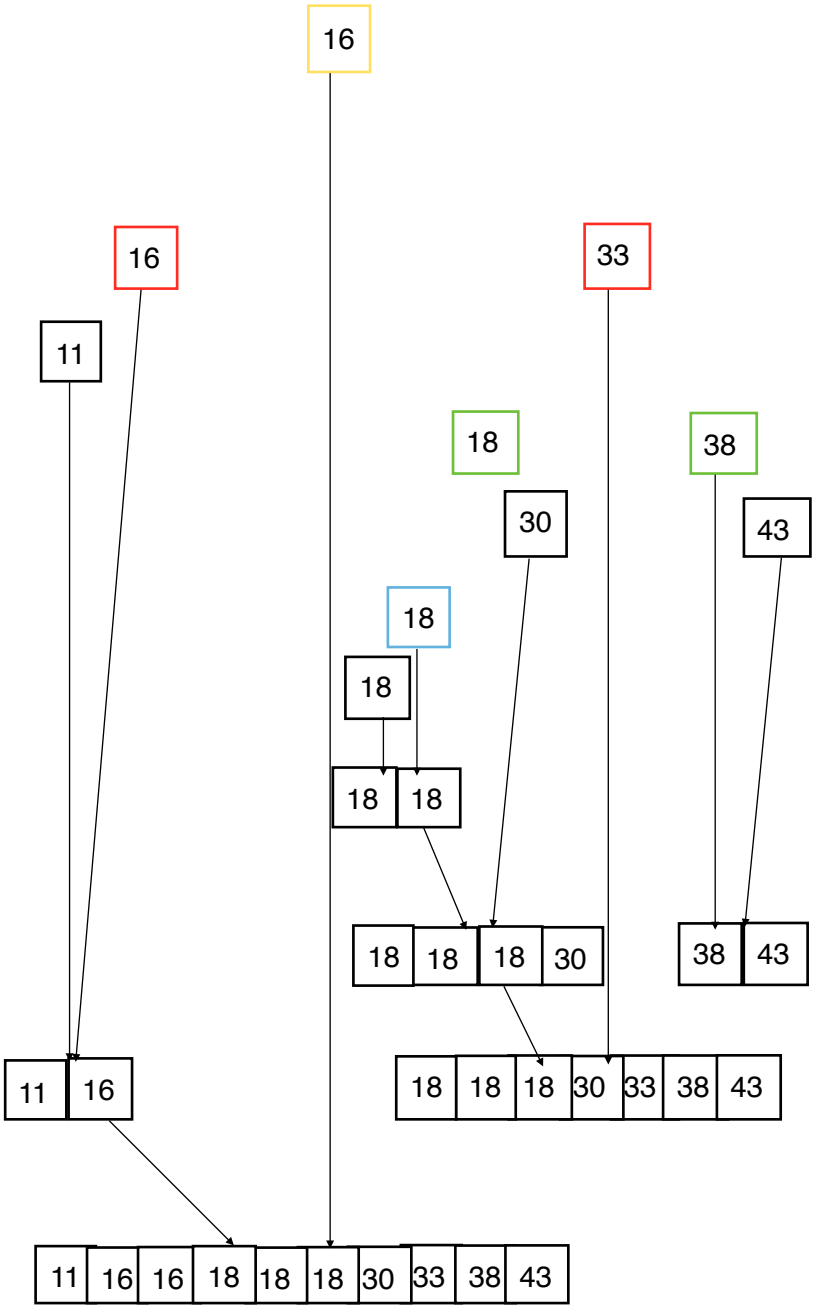


18<18 so go

crossed over so stop
and swap with right

18

18


16
16
33
11
18
38
30
43
18
18
18
18
18
18
18
30
38
43
11
16
18
18
18
30
33
38
43
11
16
16
18
18
18
30
33
38
43