index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26
array	2	5	5	7	9	15	20	20	27	32	37	38	40	43	51	53	64	64	71	77	77	77	80	84	86	91	95

Answer the following question in a `hw7.pdf` file.

Question 1: On paper, using the above array, trace the execution of the ternary search algorithm starting with *target* = 80, *first* = 0, and *last* = 26. At each step, show how *last* and *first* change, and which positions and values of the array are examined, and what comparisons are being performed. Compute the index of the position one third of the way along the array using the formula

$$\text{oneThird} = (\text{first} + (\text{last} - \text{first})/3)$$

and compute the index of the item 2/3 of the way along the array using the formula

$$\text{twoThirds} = (\text{first} + 2 * (\text{last} - \text{first})/3).$$

oneThird $= (0+(26-0)/3) = 8.6666\overline{6} = 8$

twoThirds $= (0+2\cdot(26-0)/3) = 17.3333\overline{3} = 17$

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26
array	2	5	5	7	9	15	20	20	27	32	37	38	40	43	51	53	64	64	71	77	77	77	80	84	86	91	95

1) looks at the item at oneThird (27) and compares it to target (80). if it matches, return oneThird.
2) compares target and 1/3 item again, this time checking if its smaller than target. if so, it returns recursively with only the first 1/3 of the array.
3) compares target and 1/3 item again, this time checking if its bigger than target. if so, it goes through three more if statements:
    a) compares target with the item at twoThirds, if they are equal, return twoThirds.
    b) compares target with 2/3 item, checking if its smaller than target. if so, it returns recursively with only the last 1/3 of the array.
    c) lastly compares target with the 2/3 item, this time checking if it's bigger than target. if so, it returns recursively, searching the middle 1/3 of the array.

if the target is not found under any of these conditions, it returns -1 (an invalid index)

oneThird = 8
twoThirds = 17
target = 80
using these steps:
1) 80 == 27? false
2) 80 < 27? false
3) 80 > 27? true:
    a) 80 == 64? false
    b) 80 > 64? true
    return

oneThird = 20
twoThirds = 23
target = 80
1) 80 == 77? false
2) 80 < 77? false
3) 80 > 77? true:
    a) 80 == 84? false
    b) 80 > 84? false
    c) 80 < 84? true
    return

oneThird = 21
twoThirds = 22
target = 80
1) 80 == 77? false
2) 80 < 77? false
3) 80 > 77? true:
    a) 80 == 80? true
    return 22

```
./sort SelectionSort 10
```

```
./sort MergeSort 10
```

```
./sort MeanSort 10
```

Use the **plot.sh** script we have provided to make a graph of the execution time of each of these sorting algorithms, for a variety of list lengths. You can generate a graph like this:

```
./plot.sh ./sort SelectionSort 1000 2000 5000 7000 10000
```

This runs the given program for each of the sizes you specify (which must be in ascending order), and graphs the execution time results. Use enough sizes, and at least five, to get a nice curve over a wide range of list sizes and use the same lists sizes for each fo the three algorithms. These graphs, `SelectionSort.png`, `MergeSort.png`, and `MeanSort.png` should be submitted with your project.

Note: There will be natural variation in execution time, due to noise, the specific random numbers generated, etc., so use input sizes large enough to see very significant changes in the execution time (try values of `n` of at least several thousand).

`Question 2:` Comment on your three graphs and if they match the expected running times of each of these sorting algorithms.

SelectionSort: the graph depicts an inefficient execution time, like $O(n^2)$, which is expected.
MergeSort: this graph shows a run time similar to $O(n)$, which is expected.
MeanSort: this graph portrays an execution time similar to $O(\log_3 n)$, which is expected.