CS 315 - Day 12, Selection Sort and Timing

Our first incremental algorithm, Insertion sort, was iterative (not recursive), and had a cost of $\theta(n^2)$. Now we consider another iterative sort, Selection Sort. The entire array/list is searched for the smallest element, then the next smallest, and so on, until it is sorted.

	`Selection-Sort(A,n)`	***constant * # executions***
1	`n = length[A]`	$c_1 * 1$
2	`for i=1 to n-1:`	$c_2 * n$
3	`    smallest = i`	$c_3 * (n-1)$
4	`    for j=i+1 to n:`	$c_4 * (\sum_{i=1}^{n-1} t_i)$
5	`        if A[smallest] > A[j]`	$c_5 * (\sum_{i=1}^{n-1} t_i - 1)$
6	`            then smallest = j`	c_6 * ?
7	`    Swap A[i] with A[smallest]`	$c_7 * (n-1)$

<u>In-class w/***assigned groups of four (4)***, write 0. and 1. on board as ground rule(s):</u>

0. No devices.
1. Intro: Name, POE, Icebreaker.
2. Pick someone on the group who will report out.
3. Work through Selection sort, using the cards and following the pseudocode above.
4. Will the sort be faster (or slower) depending on the ordering? Is there a best (or worst) case for the data?
5. In the pseudocode above, what is the most frequently executed line?
6. Attempt to determine the values of t_i and then the asymptotic efficiency, in terms of θ, O, or Ω.

Line 4, which is the inner for-loop, will be executed most. Since we are dealing with nested for-loops, our intuition says that this will be an n^2 algorithm. But lets show this.

What is t_i ?
when $i = 1,\ t_i = t_1 = n$
when $i = 2,\ t_2 = n - 1$
when $i = 3,\ t_3 = n - 2$
and so on…
when $i = n - 1,\ t_{n-1} = 2$

which leads to $t_i = n - i + 1$, and this is unaffected by any ordering of inputs, best/worst case, etc.

We substitute this, $\sum_{i=1}^{n-1} t_i = \sum_{i=1}^{n-1} n - i + 1 = \sum_{i=1}^{n-1} n - \sum_{i=1}^{n-1} i + \sum_{i=1}^{n-1} 1$

$$= n \cdot (n-1) - \frac{(n-1) \cdot n}{2} + (n-1)$$

We could stop here and note that the highest order term is n^2, giving us $\theta(n^2)$, but lets do the algebra for completeness:

$$= n^2 - n - \frac{n^2}{2} - \frac{n}{2} + n - 1 = \frac{n^2}{2} - \frac{n}{2} - 1 = \theta(n^2)$$

Since t_i is the same regardless of input ordering, best, worst, and average cases are all $\theta(n^2)$.

Starting with `selectionPoC.ipynb`, which includes the pseudocode above, write a `selection` function. Step through the code and output below:

```
N=8
A = list(range(N))
random.shuffle(A)
print(A)
selection(A,True)
print(A)
```

```
[5, 4, 0, 6, 3, 2, 1, 7]
[0, 4, 5, 6, 3, 2, 1, 7]
[0, 1, 5, 6, 3, 2, 4, 7]
[0, 1, 2, 6, 3, 5, 4, 7]
[0, 1, 2, 3, 6, 5, 4, 7]
[0, 1, 2, 3, 4, 5, 6, 7]
[0, 1, 2, 3, 4, 5, 6, 7]
[0, 1, 2, 3, 4, 5, 6, 7]
[0, 1, 2, 3, 4, 5, 6, 7]
```

Note how the output for Insertion-Sort and Selection Sort differ. Insertion just takes the next element and inserts it in sorted order into the list so far, while Selection surveys the entire remaining list and inserts it at the end of the sorted list so far.

The invariant for Insertion-Sort is that the list sorted so far, of length i, consists of just the first i elements in the given list, and the invariant for Selection Sort is that the list sorted so far, of length i, consists of the i smallest elements in the entire given list.

So, how do we measure performance, to confirm that our code is running at the expected time?

First, we add timing, which in Python would require:

```
import time
start = time.time()
# call sort function - nothing else
end = time.time()
print("N= ",N,"time= ",end-start)
```

We need to make sure to only time the sort, and to turn off the flag to output intermediate results. Output will dramatically skew results.

We can then run on larger problem sizes, capturing the problem size n and associated elapsed time.

While Floating Point Operations Per Second, or FLOPS, is interesting to calculate, we're more interested in confirming that we are meeting our asymptotic expectations, which is the code is running at $\theta(n^2)$ or $O(n \lg(n))$ as n gets large.

In this case we use the Ratio Test:

$$\frac{actual\ running\ time}{expected\ running\ time}$$

And we consider the trend as n gets large.

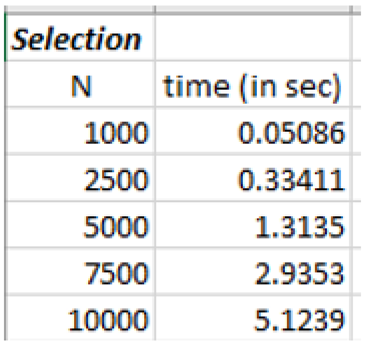
For example, this could be $\frac{running\ time\ for\ size\ n}{n^2}$ or $\frac{running\ time\ for\ size\ n}{n\lg(n)}$.

As n gets larger, the ratios will converge, assuming the function for the expected running time in the denominator is correct.

This works because as n gets larger, the algorithm's the highest-order term will dominate the running time, and any fixed-cost constants.

In a Homework assignment you will be asked to add timings to your sorts and confirm through the ratio test that the sorts are exhibiting the expected performance.

Calculate the ratios for the given Selection Sort timings:



Selection	
N	time (in sec)
1000	0.05086
2500	0.33411
5000	1.3135
7500	2.9353
10000	5.1239

$$\frac{0.05086}{1000^2} = 5.086 \cdot 10^{-08}$$
$$\frac{0.33411}{2500^2} = 5.35 \cdot 10^{-08}$$
$$\frac{1.3135}{5000^2} = 5.254 \cdot 10^{-08}$$
$$\frac{2.9353}{7500^2} = 5.218 \cdot 10^{-08}$$
$$\frac{5.1239}{10000^2} = 5.1239 \cdot 10^{-08}$$

How can you use the Ratio Test to estimate runtime for a larger problem?

For example, based on the data above, how long would Selection sort run for N= 20,000?

$$\frac{running\ time\ for\ size\ n}{n^2} = ratio$$

$$\frac{running\ time\ for\ size\ n}{20000^2} = 5.1 \cdot 10^{-08}$$

$$running\ time\ for\ size\ n = 20000^2 \quad \cdot \quad 5.1 \cdot 10^{-08} = 20.4 \text{ seconds}$$

What about Mergesort? What would the ratio be for a problem of size $n = 10000$ which takes 0.0625 sec?

$$\frac{running\ time\ for\ size\ n}{n \cdot \lg(n)} = ratio$$

$$\frac{0.0625}{10000 \cdot \lg(1000)} = \frac{0.0625}{10000 \cdot 13.288} = 4.703 \cdot 10^{-07}$$

Where $\lg(10000) = \frac{log_{10}(10000)}{log_{10}(2)} = 13.288$