Analysis of Selection Sort

Selection sort loops over indices in the array; for each index, selection sort calls **indexOfMinimum** and swap. If the length of the array is ***n***, there are ***n*** indices in the array.

Since each execution of the body of the loop runs two lines of code, you might think that **2n** lines of code are executed by selection sort. But it's not true! Remember that **indexOfMinimum** and swap are functions: when either is called, some lines of code are executed.

How many lines of code are executed by a single call to swap? In the usual implementation, it's three lines, so that each call to swap takes constant time.

How many lines of code are executed by a single call to **indexOfMinimum**? We have to account for the loop inside indexOfMinimum. How many times does this loop execute in a given call to **indexOfMinimum**? It depends on the size of the subarray that it's iterating over. If the subarray is the whole array (as it is on the first step), the loop body runs **n** times. If the subarray is of size 6, then the loop body runs 6 times.For example, let's say the whole array is of size 8 and think about how selection sort works.

1. In the first call of indexOfMinimum, it has to look at every value in the array, and so the loop body in **indexOfMinimum** runs 8 times.
2. In the second call of **indexOfMinimum**, it has to look at every value in the subarray from indices 1 to 7, and so the loop body in **indexOfMinimum** runs 7 times.
3. In the third call, it looks at the subarray from indices 2 to 7; the loop body runs 6 times.
4. In the fourth call, the subarray from indices 3 to 7; the loop body runs 5 times.

5. ...

6. In the eighth and final call of **indexOfMimimum**, the loop body runs just 1 time.

If we total up the number of times the loop body of **indexOfMinimum** runs, we get 8 + 7 + 6 + 5 + 4 + 3 + 2 + 1 = 36 times.

Side note: Computing summations from 1 to n

How do you compute the sum 8 + 7 + 6 + 5 + 4 + 3 + 2 + 1 quickly? Here's a trick. Let's add the numbers in a sneaky order. First, let's add 8 + 1, the largest and smallest values. We get 9. Then, let's add 7 + 2, the second-largest and second-smallest values. Interesting, we get 9 again. How about 6 + 3? Also 9. Finally, 5 + 4. Once again, 9! So what do we have?

$$(8+1)+(7+2)+(6+3)+(5+4)=9+9+9+9$$

$$=4 * 9$$

$$=36$$

There were four pairs of numbers, each of which added up to 9. So here's the general trick to sum up any sequence of consecutive integers:

1. Add the smallest and the largest number.
2. Multiply by the number of pairs.

What if the number of integers in the sequence is odd, so that you cannot pair them all up? It doesn't matter! Just count the unpaired number in the middle of the sequence as half a pair. For example, let's sum up 1 + 2 + 3 + 4 + 5. We have two full pairs (1 + 5 and 2 + 4, each summing to 6) and one "half pair" (3, which is half of 6), giving a total of 2.5 pairs. We multiply **2.5 * 6 = 15**, and we get the right answer.

What if the sequence to sum up goes from 1 to **n**? We call this an arithmetic series. The sum of the smallest and largest numbers is **n + 1**. Because there are **n** numbers altogether, there are n/2n/2 pairs (whether **n** is odd or even). Therefore, the sum of numbers from 1 to **n** is **(n + 1)(n / 2)**, which equals $n^2/2$

+ n/2. Try out this formula for **n** = **5** and **n** = **8**.

Asymptotic running-time analysis for selection sort

The total running time for selection sort has three parts:

1. The running time for all the calls to **indexOfMinimum**.
2. The running time for all the calls to swap.
3. The running time for the rest of the loop in the **selectionSort** function.

Parts 2 and 3 are easy. We know that there are **n** calls to **swap**, and each call takes constant time. Using our asymptotic notation, the time for all calls to swap is **Θ(n)**. The rest of the loop in **selectionSort** is really just testing and incrementing the loop variable and calling **indexOfMinimum** and swap, and so that takes constant time for each of the nn iterations, for another **Θ(n)** time.

Adding up the running times for the three parts, we have $\mathbf{\Theta(n^2)}$ for the calls to **indexOfMinimum**, **Θ(n)** for the calls to swap, and **Θ(n)** for the rest of the loop in selectionSort. The $\mathbf{\Theta(n^2)}$ term is the most significant, and so we say that the running time of selection sort is $\mathbf{\Theta(n^2)}$. Notice also that no case is particularly good or particularly bad for selection sort. The loop in indexOfMinimum will always make n^2 + n/2 iterations, regardless of the input. Therefore, we can say that selection sort runs in $\Theta(n^2)$ time in all cases.

Let's see how the $\mathbf{\Theta(n^2)}$ running time affects the actual execution time. Let's say that selection sort takes approximately $\mathbf{n^2/10^6}$ seconds to sort **n** values. Let's start with a fairly small value of **n**, let's say **n** = **100**. Then the running time of selection sort is about $\mathbf{100^2/10^6}$ = **1/100** seconds. That seems pretty fast. But what if **n** = **1000**? Then selection sort takes about $\mathbf{1000^2/10^6}$ =**1** second. The array grew by a factor of 10, but the running time increased 100 times. What if **n**=**1,000,000**? Then selection sort takes $\mathbf{1{,}000{,}000^2/10^6}$=**1,000,000** seconds, which is a little more than 11.5 days. Increasing the array size by a factor of 1000 increases the running time a million times!

1000 increases the running time a million times.