CS50 Insertion Sort

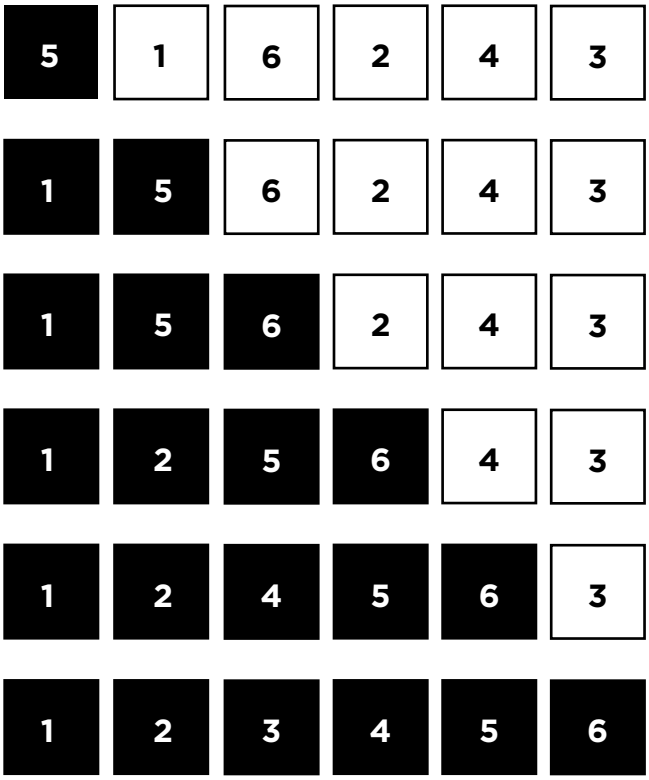
Overview

Insertion sort is yet another algorithm to sort arrays, but this time it does not require multiple iterations over the **array**. Like usual, optimizations usually force the programmer to sacrifice something else. However these sacrifices are nearly negligible, in the case of insertion sort, when the array is small or the array is nearly sorted. Similar to selection sort, the array of elements will be split into two parts: a sorted portion and an unsorted portion.

Key Terms

- insertion sort
- array
- pseudocode

Step-by-step process for insertion sort



Implementation

In the case of having two elements in the array, the implementation is relatively simple. Consider the first element to be automatically in the sorted portion of the list. Look at the next element in the array, and determine where it fits in the sorted list. This can be applied to a larger array with the following **pseudocode**:

```
for each unsorted element, n, in the array

    determine where in the sorted portion of the
    array to insert n

    shift sorted elements rightwards as necessary
    to make room for n

    insert n into sorted portion of the list
```

When this is implemented on the example array, the program would start at `array[1]`, which is 1, since an array of size one (`array[0]`) is already sorted. 5 would get shifted over to the right, and 1 would be moved to `array[0]`. Next, the program looks at 6. 6 is greater than 5 so no elements need to be shifted to make room for it. And so on and so forth. Eventually with this procedure, the entire array will be sorted.

Sorted Arrays

There is no guarantee that after any step in the implementation, any elements are in the correct location in the array. Even if an element did happen to be in its correct location in the initial array, it is likely that it would be moved to a different location within the sorted array before it would be shuffled back into its final location. Also note that in insertion sort, all the elements to the right of the newest element in the sorted list have to be shifted over one space. So while it may seem like insertion sort involves n steps, we are increasing the amount of times an element is being moved because elements are being shifted over to accomodate new elements rather than just being swapped. This is, however, dependent on the order of the initial array. It is also worth considering that sorting algorithms need to address cases in which two elements are equal. When dealing with few unique elements the key is just to be consistent. If, in the implementation of insertion sort, there was a line that checked if the current element was equal to an element in the sorted array, it should always have the same outcome, whether it is to the right or the left of that element is irrelevant.