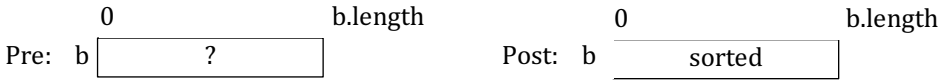
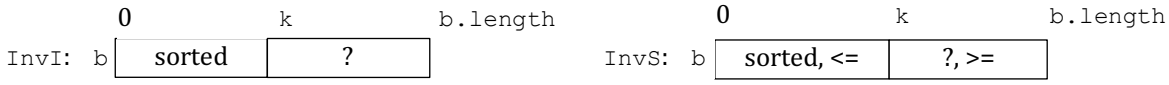
Insertion sort and selection sort

We *develop* two algorithms to sort an array `b`: insertion sort and selection sort. The pre- and post-conditions are:



To the left below, we draw invariant `InvI` in the natural way, generalizing the pre- and post-conditions and introducing variable `k` to mark the boundary between the two sections. `InvI` is the invariant for insertion sort. To the right, we add one property to `InvI` to get invariant `InvS` for selection sort: the <= in the left section and >= in right section are shorthand for `b[0..k-1] <= b[k..]` —every value in the left section is at most every value in the right section. If you remember this one extra property, developing selection sort is a piece of cake.



The invariants have the same shape. Both say that `b[0..k-1]` is processed —sorted— and the second section `b[k..]` is not completely processed. Therefore, the loops that we develop from them will look similar. The only difference will be in how processing is done in the repetend to maintain the invariant.

To truthify each invariant initially, makes the left section empty by setting `k` to `0`. The postcondition is true when the invariant is true and the second section is empty, i.e. when `k = b.length`. Therefore the loop condition can be written as `k != b.length` or `k < b.length`. Finally, the way to get closer to terminaton is to increase `k`. So, we write the algorithms like this:

```
// insertion sort
k= 0;
while (k != b.length) {
   Ensure that InvI will be true after k= k+1;

   k= k+1;
}
```

```
// selection sort
k= 0;
while (k != b.length) {
   Ensure that InvS will be true after k= k+1;

   k= k+1;
}
```

Ensuring InvI will be true

Look at invariant `InvI`. Before `k` can be incremented, `b[0..k]` has to be sorted —in ascending order. But it won't be if `b[k-1] > b[k]`.

Here's an example, with `b[k]` shown in red:

`b[0..k]` = (1, 3, 3, 5, 7, 8, **4**)

Here, `b[k]` is smaller than several values in `b[k-1]`. So, `b[k]` must be pushed down into its sorted position in a number of swaps, as shown below:

`b[0..k]` = (1, 3, 3, 5, 7, **4**, 8)
`b[0..k]` = (1, 3, 3, 5, **4**, 7, 8)
`b[0..k]` = (1, 3, 3, **4**, 5, 7, 8)

Thus, to ensure that `InvI` will be true after incrementing `k`:

Push `b[k]` down to its sorted position in `b[0..k]`.

We leave this as a high-level statement.

Ensuring InvS will be true

Look at invariant `InvS`. Because `b[0..k-1] <= b[k..]`, we know that `b[0..k]` is in ascending order. But before `k` can be incremented, we need `b[k] <= b[k+1..]`.

Here's an example where this is not the case:

`b[k..]` = (8, 9, 8, 7, 5, 10)

Here, `b[k]` = 8 is larger than the 5 and the 7.

Evidently, before `k` can be incremented, the smallest value of `b[k..]` must be in `b[k]`. So, to ensure that `InvS` will be true after incrementing `k`:

Swap `b[k]` with the min value of `b[k..]`

We leave this as a high-level statement

Our two algorithms, insertion sort and selection sort, are as shown below. This is how to first present them.

Insertion sort and selection sort

```
// insertion sort
k= 0;
//invariant: InvI
while ( k != b.length) {
   Push b[k] down to its sorted position
        in b[0..k];
   k= k+1;
}
```

Push b[k] down to its sorted position in b[0..k]

Pushing `b[k]` down is easily done with a loop:

```
j= k;
// inv: b[0..k] is sorted except b[j] may be
//      greater than b[j-1]
while (0 < j  &&  b[j-1] > b[j]) {
   swap b[j-1] and b[j];
   j= j - 1;
}
```

Execution time of insertion sort

Let `n = b.length`. If array `b` is already sorted, the loop of *Push* `b[k]` *down to its sorted position* executes 0 iterations, i.e. it takes time `O(1)`. Because the main loop requires `n` iterations, the algorithm takes time `O(n)`.

If `b` is in descending order, then each execution of *Push* `b[k]` *down to its sorted position* requires `k` swaps because `b[k]` is smaller than every element in `b[0..k-1]`. The statement *Push* `b[k]` ... is executed for `k = 0, 1, 2, 3, …, n-1`. Therefore the number of swaps is

$$0 + 1 + 2 + \ldots + n-1 = (n-1)(n)/2$$

Therefore $O(n^2)$ swaps are executed and the time is $O(n^2)$.

What is the average/expected execution time? On the average, at iteration `k` of the main loop, `b[k]` will have to be swapped halfway down to position `k/2`. This is still linear in `k`, and because of this, the average/expected time is also $O(n^2)$.

```
// selection sort
k= 0;
// Invariant: InvS
while (k != b.length) {
   Swap b[k] with the minimum value of b[k..]

   k= k+1;
}
```

Swap b[k] with min of b[k..]

First, find the index of the minimum value of `b[k..]`, then swap. You have seen an algorithm to find the minimum before.

```
j= k+1; i= k;
// inv: b[i] is the minimum of b[k..j-1]
for (j= k+1; j < b.length; j= j+1) {
   if (b[j] < b[i]) i= j;
}
swap b[k] and b[i]
```

Execution time of selection sort

Let `n = b.length`. The section `b[k..]` can be written as `b[k..n-1]`. It has size `n-k`. Looking at the loop of the implementation of *Swap* `b[k]` *with the min of* `b[k..]`, we see that it makes `n-k-1` array comparisons.

Since *Swap* `b[k]` *with the min of* `b[k..]` is executed for `k = 0, 1, 2, …, n-1`, the total number of array comparisons made is

$$n-1 + n-2 + \ldots + 0 = (n-1)(n-2)/2$$

Therefore, $O(n^2)$ array comparisons are made. The number of swaps is about `n`. So the worst-case time *and* the expected time is $O(n^2)$.

Selection sort always makes the same number of array comparisons, no matter what values are in the array. And, it always makes `O(n)` swaps. Therefore, the worst-case time, expected time, and best-case time are the same: $O(n^2)$.