Here is selection sort again:

In [1]:
```
def max_i(L1):
    '''Return the index of a largest element in L. If there is more than
    one maximal element, return the index of the leftmost one'''
    cur_max = L[0] #the largest element up to index i
    cur_max_i = 0  #the index of the largest element up to index i

    for i in range(1, len(L1)):
        if L1[i] > cur_max:
            cur_max = L1[i]
            cur_max_i = i
    return cur_max_i

def selection_sort(L):
    '''Modify L so that it's sorted in non-decreasing order

    Arugment:
    L -- a list of ints
    '''
    for j in range(len(L)):
        ind_of_max = max_i(L[:len(L)-j])
        L[ind_of_max], L[-1-j] = L[-1-j], L[ind_of_max]
```

Here is how we can estimate the worst-case runtime complexity of `selection_sort()`.

We first observe the loop in `selection_sort()`. It repeats `len(L)` times, but the complexity is not $\mathcal{O}(len(L))$. That's because different iterations take different amounts of time: if `j` is small, `max_i(L[:len(L)-j]))` will take longer than if `j` is large, because the list `L[:len(L)-j]` will be larger.

So what's the runtime complexity of `max_i(L[:len(L)-j])`? Two things happen there. First, we create a new list, `L[:len(L)-j]`. That actually takes time proportional to the length of the new list, `len(L)-j`, but we'll ignore this (see below for a discussion) and focus on the call to `max_i()`.

What's the complexity of `max_i(L1)`? Well, the loop in there runs for `len(L1)-1` iterations. The loop in `max_i(L1)` *does* take time that doesn't depend on the index `i`, which means that `max_i(L1)` runs in $\mathcal{O}(k)$ time, where `k=len(L1)`.

Back to the complexity of `selection_sort(L)`. Let `n = len(L)`.

```
When j = 0, there are n-1 iterations in max_i() performed.

When j = 1, there are n-2 iterations in max_i() performed.

...

When j = n-1, there are 0 iterations in max_i() performed.
```

So in total, there are `1+2+3+...+(n-1)` iterations performed in `max_i()`. Now,

$\sum_{i=1}^{m} = m(m+1)/2$, so $1+2+3+...+(n-1) = n(n-1)/2$.

So in total we perform $n(n-1)/2 = n^2/2 - n/2$ iterations. That means Selection Sort runs in $\mathcal{O}(n^2)$ time.

(Now, we mentioned that creating the list `L[:len(L)-j]` also takes time. It was (sort of) OK to ignore that, since we then processed the new list by going through it. Technically, in addition to the iterations that we counted, we'd also create new lists of total length $n(n-1)/2$ (a list of length $n-1$, then a list of length $n-2$, and so on.) This will also take $\mathcal{O}(n^2)$ time, so the total runtime will be $\mathcal{O}(n^2)$.)