Recall from last lecture:
Bubble Sort Algorithm

1 Input the numbers $x(1), \ldots, x(n)$ (as a vector).

2 Input (or compute) $n$.

3 for $k = 1, \ldots, n - 1$
   for $j = 1, \ldots n - k$
     if ($x(j) > x(j + 1)$)
       swop $x(j)$ and $x(j + 1)$
     end if
   end for j
end for k

4 Output the sorted list
Programs: bubble.m, complexity2.m

Time for execution grows with $O(n^2)$.

We showed this by counting the number of comparisons and swops: $n(n - 1)/2$.

Can we do better?
Selection Sort Algorithm

Items 1, 2 and 4 are as for bubble sort, but item 3 is replaced by the following

3 for $k = 1, \ldots, n - 1$

    assign $l = k$

    for $j = k + 1, \ldots, n$
        if($x(j) < x(l)$)
            assign $l = j$
        end if
    end for $j$

    if($l \neq k$)
        swop $x(k)$ and $x(l)$
    end if

end for $k$

Whole thing coded as selection.m
The idea is to swap the first element of the input vector $x$ with the smallest element, then the second with the second smallest, and so on. The $j$-loop finds the position $l$ of a smallest entry of the sub-vector $[x(k), \ldots, x(n)]$, after the sub-vector $[x(1), \ldots, x(k - 1)]$ has already been sorted.
Example: Selection Sort

Let $n = 5$, $x = [27, 13, 9, 5, 3]$.

First passage through $k$-loop: $k = 1$.
Start by putting $l = k (= 1)$, but then (in the $j$-loop) look to see if there are any smaller elements at later positions in the vector; if there are, change $l$ to mark their position. At the end of the $j$-loop, $l = 5$, marking the position of the smallest element in $x$.
Now swap $x(k) = x(1)$ and $x(l) = x(5)$: $x$ becomes $[3, 13, 9, 5, 27]$. 
Second passage through $k$-loop: $k = 2$. Start by putting $l = k (= 2)$, and then look to see if there are any smaller elements later. (We have already placed the smallest element at $k = 1$, and we’re looking to place the second smallest at $k = 2$.) The smallest is $x(4)$, so at end of $j$-loop $l = 4$. Swop $x(k) = x(2)$ and $x(l) = x(4)$: $x$ becomes $[3, 5, 9, 13, 27]$.

Third passage through $k$-loop: $k = 3$. Start by putting $l = 3$; there are no smaller elements later, so it never changes.

Nor does it for the fourth and last passage through the $k$-loop, so we’re done.
function y = selection(x)

n = length(x);

for k = 1:n-1

    l = k; % by the time we get here
    % we’ve already ordered
    % the elements before k,
    % and we only need to worry
    % about those from k on

    for j = k+1:n
        if(x(j)<x(l)) % we’ve found
            % a smaller element,
            l = j; % so we mark its position
        end % if
    end % for

end % for
if(l~=k) % if the k-th element
    % is not the smallest from k on,
    % swop it with the element that is
    oldxl = x(l);
    x(l) = x(k);
    x(k) = oldxl;
end % if

end % for

y = x; % output is the sorted vector
Analysis of Selection Sort

Number of comparisons:
\((n - 1) + (n - 2) + \ldots + 2 + 1 = n(n - 1)/2 - 1\)

Number of assignments (at most):
\((n - 1) + [(n - 1) + (n - 2) + \ldots + 1]\
\equiv (n - 1) + n(n - 1)/2 .

At most \(n - 1\) swops.

Again \(O(n^2)\) - same as Bubble Sort.
complexity3.m compares these two algorithms:

We use each algorithm to sort five random vectors with $n$ entries and find the mean of the computation times.

This is repeated for increasing $n$.

We see that on average selection.m wins over bubble.m.

In general, they are both $O(n^2)$, so to explain this we would need a more detailed analysis. It turns out that the coefficient of $n^2$ for selection.m is smaller than that for bubble.m for most vectors.
There are even faster algorithms - e.g. the MATLAB function `sort`.

See `complexity4.m` for an evaluation of this.

Compared to selection sort, it sorts vectors 100 times as long in a slightly shorter time!