L34: Sorting

Different sorting methods

Lucas A. J. Bastien

E7 Spring 2017, University of California at Berkeley

April 17, 2017

Version: release

Announcements

Lab 12 is due on April 21 at 12 pm (noon)

Today:

- Sorting

Next week:

- Sorting, searching, linked lists

Remember: You are responsible for keeping up with:

- bCourses announcements
- bCourses messages
- email

(read these communications carefully)

A catalog of methods

For part 2 of the class, I would recommend you to create for yourself (with pen and paper) a catalog of the numerical methods that we talked about. For each chapter of part 2:

- **Write the “what” and the “why”** *i.e.* what is the chapter about, and why is it relevant?
  - Example for root finding: find approximate solutions to $f(x) = 0$, useful when the equation is too hard to solve by hand
- **List the methods that we learned**. For each method, write:
  - How does the method work?
  - What are its strengths and limitations?
  - What is the order of the method (when we can describe the order)?
  - Two examples that you can solve by hand

Matlab's built-in sort function

Arrange data in ascending or descending order
(each column is sorted separately)

```
>> rng(1)
>> array = randi([0, 10], [4, 3])
array =
     4     1     4
     7     1     5
     0     2     4
     3     3     7
>> % By default, sorting is done in ascending order
>> sort(array)
ans =
     0     1     4
     3     1     4
     4     2     5
     7     3     7
>> % But it can be changed
>> sort(array, 'descend')
ans =
     7     3     7
     4     2     5
     3     1     4
     0     1     4
```

Matlab's built-in sort function

```
>> rng(1)
>> array = randi([0, 10], [4, 3])
array =
     4     1     4
     7     1     5
     0     2     4
     3     3     7

>> % Get the re-organized lists of indices
>> [sorted, indices] = sort(array)
sorted =
     0     1     4
     3     1     4
     4     2     5
     7     3     7
indices =
     3     1     1
     4     2     3
     1     3     2
     2     4     4
```

Matlab's built-in sortrows function

Arrange other columns in the same order as a designated "sorting column" (default: sort based on first column)

```
>> beatles = {'John', 'Lennon', 1940; ...
       'Ringo', 'Starr', 1940; 'Paul', 'McCartney', ...
       1942; 'George', 'Harrison', 1943}
beatles =
  4x3 cell array
    'John'       'Lennon'       [1940]
    'Ringo'      'Starr'        [1940]
    'Paul'       'McCartney'    [1942]
    'George'     'Harrison'     [1943]

>> sortrows(beatles)
ans =
  4x3 cell array
    'George'     'Harrison'     [1943]
    'John'       'Lennon'       [1940]
    'Paul'       'McCartney'    [1942]
    'Ringo'      'Starr'        [1940]
```

Matlab's built-in sortrows function

Arrange other columns in the same order as a designated "sorting column" (default: sort based on first column)

```
>> beatles = {'John', 'Lennon', 1940; 'Ringo', 'Starr', 1940; ...
        'Paul', 'McCartney', 1942; 'George', 'Harrison', 1943};

>> sortrows(beatles, 2)
ans =
  4x3 cell array
    'George'      'Harrison'       [1943]
    'John'        'Lennon'         [1940]
    'Paul'        'McCartney'      [1942]
    'Ringo'       'Starr'          [1940]

>> sortrows(beatles, 3)
ans =
  4x3 cell array
    'John'        'Lennon'         [1940]
    'Ringo'       'Starr'          [1940]
    'Paul'        'McCartney'      [1942]
    'George'      'Harrison'       [1943]
```

Basic sorting algorithm

- Maintain two lists:
  - The values sorted so far
  - The values that remain to be sorted
- As long as there remain values to be sorted:
  - Find the minimum in the values that remain to be sorted
  - Add this value at the end of the sorted list
  - Remove this value from the values that remain to be sorted

```
function [sorted] = my_sort_basic(vector)

% Sorts a vector using a basic sorting algorithm.

sorted = zeros(size(vector));
for i = 1:numel(vector)
    [minimum, index] = min(vector);
    sorted(i) = minimum;
    vector(index) = [];
end

end
```

Selection sorting algorithm

For each index *i* in the vector:

- Find the minimum element between this index and the end of the vector
- Swap this minimum element with the element at index *i*

```
function [vector] = my_sort_selection(vector)

% Sorts a vector using the selection sort algorithm.

for i = 1:numel(vector)
    [minimum, index] = min(vector(i:end))
    vector([i, i-1+index]) = vector([i-1+index, i]);
end

end
```

Selection sorting algorithm

Example of what happens at each step for the following vector;

[2, 3, 8, 5, 1, 10, 6, 9, 4, 7]

2	**3**	**8**	**5**	**1**	**10**	**6**	**9**	**4**	**7**
1	**3**	**8**	**5**	**2**	**10**	**6**	**9**	**4**	**7**
1	**2**	**8**	**5**	**3**	**10**	**6**	**9**	**4**	**7**
1	**2**	**3**	**5**	**8**	**10**	**6**	**9**	**4**	**7**
1	**2**	**3**	**4**	**8**	**10**	**6**	**9**	**5**	**7**
1	**2**	**3**	**4**	**5**	**10**	**6**	**9**	**8**	**7**
1	**2**	**3**	**4**	**5**	**6**	**10**	**9**	**8**	**7**
1	**2**	**3**	**4**	**5**	**6**	**7**	**9**	**8**	**10**
1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**

Insertion sorting algorithm

- Divide the list into two parts: **sorted** and **unsorted**
- Start with the **sorted** part (the first element only)
- Move second, third, fourth, ... elements one at a time from the **unsorted** into the **sorted** part of the list
- Each element transferred to the **sorted** part of list must be inserted in correct order
  - we may need to shift some already sorted elements over to make room at the correct insertion point

Insertion sorting algorithm

Example of what happens at each step for the following vector;

[2, 3, 8, 5, 1, 10, 6, 9, 4, 7]

2	**3**	**8**	**5**	**1**	**10**	**6**	**9**	**4**	**7**
2	**3**	**8**	**5**	**1**	**10**	**6**	**9**	**4**	**7**
2	**3**	**8**	**5**	**1**	**10**	**6**	**9**	**4**	**7**
2	**3**	**5**	**8**	**1**	**10**	**6**	**9**	**4**	**7**
1	**2**	**3**	**5**	**8**	**10**	**6**	**9**	**4**	**7**
1	**2**	**3**	**5**	**8**	**10**	**6**	**9**	**4**	**7**
1	**2**	**3**	**5**	**6**	**8**	**10**	**9**	**4**	**7**
1	**2**	**3**	**5**	**6**	**8**	**9**	**10**	**4**	**7**
1	**2**	**3**	**4**	**5**	**6**	**8**	**9**	**10**	**7**
1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**

Insertion sorting algorithm

```
function [vector] = my_sort_insertion(vector)

% Sorts a vector using the insertion sorting algorithm.

for i = 2:numel(vector)

    % Find where to insert vector(i)
    j = i - 1;
    while j >= 1 && vector(j) > vector(i)
        j = j - 1;
    end

    % Insert vector(i) in the list of sorted elements in
    % its place, shifting elements of the sorted part of
    % the list if necessary
    vector(j+1:i) = [vector(i), vector(j+1:i-1)];

end

end
```

Quicksort algorithm

This algorithm uses a **divide and conquer approach**

- Choose a pivot element from the list to use in partitioning the data into three groups:
  - **group 1: smaller** (elements < pivot)
  - group 2: middle (elements == pivot)
  - **group 3: larger** (elements > pivot)

- Since data within groups 1 and 3 may not be in correct order, use recursive quicksort function (qs) calls to sort them:

$$\text{sorted} = [\textbf{qs(group 1)}, \text{group 2}, \textbf{qs(group 3)}]$$

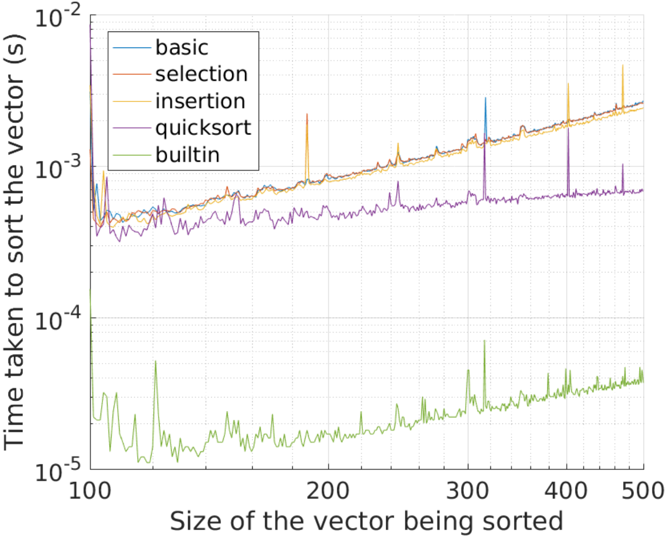
How to choose the pivot? There are different methods *e.g.*,

- First or last element of the vector
- A random element from the vector
- The median between the first, middle, and last element of the vector

Quicksort algorithm

```
function [vector] = my_sort_quicksort(vector)

% Sorts a vector using the quicksort sorting algorithm.

% Stopping criterion of the recursion
n = numel(vector);
if n < 2
    return
end

% The pivot is chosen randomly
pivot = vector(ceil(rand()*n));
smaller = vector < pivot;
greater = vector > pivot;
equal = ~(smaller | greater);

% Recursively sort values that are around the pivot
vector = [my_sort_quicksort(vector(smaller)), vector(equal), ...
    my_sort_quicksort(vector(greater))];

end
```

Efficiency of the methods



Merging two sorted lists

Concatenate the vectors and use Matlab's built-in function `sort`:

$$\text{c} = \text{sort}([\text{vector1}, \text{vector2}])$$

- Does not take advantage of "`vector1`" and "`vector2`" already being sorted

Alternative approach: use merge-sort

- Initialize counters `i1`=1, `i2`=1
- `while i1 <= numel(vector1) && i2 <= numel(vector2)`
  - Move `vector1(i1)` or `vector2(i2)` (whichever is smaller) to c
  - Increment `i1` or `i2` (whichever was moved to `c`)
- Move any remaining elements from `vector1` or `vector2` to c

Merging two sorted lists

```
function [sorted] = my_merge_sort(vector1, vector2)

% Merges sorted vectors vector1 and vector2 to produce a sorted vector.

n1 = numel(vector1);
n2 = numel(vector2);
sorted = zeros(1, n1+n2);
i1 = 1;
i2 = 1;
i = 1;

% While we still have unmerged elements in both vector1 and vector2...
while i1 <= n1 && i2 <= n2
    if vector1(i1) > vector2(i2)
        sorted(i) = vector2(i2);
        i2 = i2 + 1;
    else
        sorted(i) = vector1(i1);
        i1 = i1 + 1;
    end
    i = i + 1;
end

% At this point, we have added to "sorted" all the elements of one of the
% vectors. Add the remaining values from the other vector to the sorted
% merged vector
if i1 > n1
    sorted(i:end) = vector2(i2:end);
else
    sorted(i:end) = vector1(i1:end);
end

end
```

Merging two sorted lists

```
>> vector1 = sort(rand(1, 1e5));
>> vector2 = sort(rand(1, 1e5));

>> tic();
>> for i = 1:500; sort([vector1, vector2]); end
>> toc
Elapsed time is 6.758573 seconds.

>> tic();
>> for i = 1:500; my_merge_sort(vector1, vector2); end
>> toc
Elapsed time is 2.846680 seconds.
```

`my_merge_sort` is faster!!