L35: Sorting and searching

And linked lists

Lucas A. J. Bastien

E7 Spring 2017, University of California at Berkeley

April 19, 2017

Version: release

Announcements

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

Today:

- Sorting and searching

Next week:

- Special topics

Why sorting?

Searching through a large array is **MUCH** faster if the array is sorted

Unsorted array

- Start at beginning of list
- Look through the list one element at a time until you find a match or reach the end of the list
- On average, look at half of the list
- Complexity: $\mathcal{O}(n)$

Sorted array

- Start at the middle of list
- **Bisection method:** Eliminate half of the list (all elements either before or after the current element) at each step $\rightarrow$ **Divide and Conquer**
- At each step, divide the size of the problem by 2
- Complexity: $\mathcal{O}(\log_2(n))$

n: size of the list to search.

Why sorting?

Average number of comparisons needed to search unsorted and sorted versions of the same list:

	$n/2$	$\log_2(n)$
Search E7 roaster for an SID	≈ 210	≈ 9
Search DMV records for a California driver's license or ID number[1]	$\approx$ 16.5 millions	≈ 25
Search IRS records for a U.S. 2016 individual tax return[2]	$\approx$ 75 millions	≈ 27

n: size of the list to search

[1]: based on numbers from State of California DEPARTMENT OF MOTOR VEHICLES STATISTICS FOR PUBLICATION JANUARY THROUGH DECEMBER 2015, https://www.dmv.ca.gov/portal/dmv/detail/about/dmvinfo, retrieved April 19th 2017

[2] based on numbers from https://www.irs.gov/uac/soi-tax-stats-tax-stats-at-a-glance, retrieved April 19th 2017.

Sorting struct arrays

```
>> scientists
scientists =
  1×3 struct array with fields:
    fn
    ln
    dob
    pob
>> scientists(1)
ans =
  struct with fields:

     fn: 'Pierre'
     ln: 'Currie'
    dob: 1859
    pob: 'Paris'
>> scientists(2)
ans =
  struct with fields:

     fn: 'Marie'
     ln: 'Currie'
    dob: 1867
    pob: 'Warsaw'
>> scientists(3)
ans =
  struct with fields:

     fn: 'Isaac'
     ln: 'Newton'
    dob: 1643
    pob: 'Woolsthorpe-by-Colsterworth'
>>
```

```
>> [~, indices] = sort([scientists(:).dob]);
>> for i = indices
       scientists(i)
   end
ans =
  struct with fields:

     fn: 'Isaac'
     ln: 'Newton'
    dob: 1643
    pob: 'Woolsthorpe-by-Colsterworth'
ans =
  struct with fields:

     fn: 'Pierre'
     ln: 'Currie'
    dob: 1859
    pob: 'Paris'
ans =
  struct with fields:

     fn: 'Marie'
     ln: 'Currie'
    dob: 1867
    pob: 'Warsaw'
>>
```

Sort vector of dates of birth (dob), which are doubles

Sorting struct arrays

```
>> scientists
scientists =
  1×3 struct array with fields:
    fn
    ln
    dob
    pob
>> scientists(1)
ans =
  struct with fields:

     fn: 'Pierre'
     ln: 'Currie'
    dob: 1859
    pob: 'Paris'
>> scientists(2)
ans =
  struct with fields:

     fn: 'Marie'
     ln: 'Currie'
    dob: 1867
    pob: 'Warsaw'
>> scientists(3)
ans =
  struct with fields:

     fn: 'Isaac'
     ln: 'Newton'
    dob: 1643
    pob: 'Woolsthorpe-by-Colsterworth'
>>
```

```
>> [~, indices] = sort({scientists(:).pob});
>> for i = indices
       scientists(i)
   end
ans =
  struct with fields:

     fn: 'Pierre'
     ln: 'Currie'
    dob: 1859
    pob: 'Paris'
ans =
  struct with fields:

     fn: 'Marie'
     ln: 'Currie'
    dob: 1867
    pob: 'Warsaw'
ans =
  struct with fields:

     fn: 'Isaac'
     ln: 'Newton'
    dob: 1643
    pob: 'Woolsthorpe-by-Colsterworth'
>>
```

Sort cell array of places of birth (pob), which are character strings

Sorting struct arrays

```
>> scientists
scientists =
  1×3 struct array with fields:
    fn
    ln
    dob
    pob
>> scientists(1)
ans =
  struct with fields:

     fn: 'Pierre'
     ln: 'Currie'
    dob: 1859
    pob: 'Paris'
>> scientists(2)
ans =
  struct with fields:

     fn: 'Marie'
     ln: 'Currie'
    dob: 1867
    pob: 'Warsaw'
>> scientists(3)
ans =
  struct with fields:

     fn: 'Isaac'
     ln: 'Newton'
    dob: 1643
    pob: 'Woolsthorpe-by-Colsterworth'
>>
```

```
>> names = transpose({scientists(:).ln; scientists(:).fn})
names =
  3×2 cell array
    'Currie'    'Pierre'
    'Currie'    'Marie'
    'Newton'    'Isaac'
>> [~, indices] = sortrows(names, [1, 2]);
>> for i = transpose(indices)
       scientists(i)
   end
ans =
  struct with fields:

     fn: 'Marie'
     ln: 'Currie'
    dob: 1867
    pob: 'Warsaw'
ans =
  struct with fields:

     fn: 'Pierre'
     ln: 'Currie'
    dob: 1859
    pob: 'Paris'
ans =
  struct with fields:

     fn: 'Isaac'
     ln: 'Newton'
    dob: 1643
    pob: 'Woolsthorpe-by-Colsterworth'
>>
```

Use sortrows on the cell array of last and first names

Searching sorted vectors

Approach 1: Use Matlab's built-in `find` function *e.g.,*

```
>> rng(0); a = sort(randi([1, 1e5], [1, 1e5]));
>> indices = find(a==102)
indices =
    96    97    98    99
>> indices = find(a==104)
indices =
  1x0 empty double row vector
```

Approach 2: Use user-defined function (see `my_find_in_sorted.m`) *e.g.,*

```
>> rng(0); a = sort(randi([1, 1e5], [1, 1e5]));
>> [~, indices] = my_find_in_sorted(a, 102)
indices =
    96    97    98    99
>> [~, indices] = my_find_in_sorted(a, 104)
indices =
     []
```

Note: this approach uses an algorithm similar to bisection

Searching sorted vectors: efficiency

```
>> rng(0)
>> a = sort(randi([1, 1e5], [1, 1e5]));
>> a(93:102)
ans =
   100   101   101   102   102   102   102   103   105   105

>> % Without using the fact that "a" is sorted
>> % Matlab looks at all of the elements in the array!
>> % Complexity: O(n)
>> tic(); for i = 1:1000; find(a==102); end; toc()
Elapsed time is 0.220739 seconds.

>> tic(); for i = 1:1000; find(a==104); end; toc()
Elapsed time is 0.167810 seconds.

>> % Using the fact that "a" is sorted. Complexity: O(log_2(n))
>> tic(); for i = 1:1000; my_find_in_sorted(a, 102); end; toc()
Elapsed time is 0.004417 seconds.

>> tic(); for i = 1:1000; my_find_in_sorted(a, 104); end; toc()
Elapsed time is 0.001637 seconds.
```

Linked lists

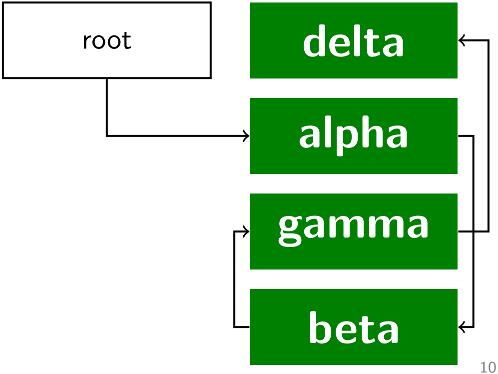
A linked list is a data structure with a root element and nodes

- The root is a reference to the first node in the list
- Nodes contain data and a reference to the next node in the list

The logical order of the data (as defined by the references in the nodes) may be different from the physical order of the data (here: the order of the nodes in the `struct` array)

```
>> list.root = 2;
>> list.node(1).value = 'delta';
>> list.node(1).next = 0;
>> list.node(2).value = 'alpha';
>> list.node(2).next = 4;
>> list.node(3).value = 'gamma';
>> list.node(3).next = 1;
>> list.node(4).value = 'beta';
>> list.node(4).next = 3;
>> list
list =
  struct with fields:

    root: 2
    node: [1×4 struct]
>>
```



Traversing a linked list

To traverse a linked list, we do not look at the elements in the order in which they appear in the `struct` array. Rather, we follow the order from one element to the next, as indicated by the reference “next” of each node. Here, we use a zero to indicate that there is no “next” element (other choices are possible *e.g.*, -1)

```
function [output] = my_ll_traverse(linked_list)

% Traverses the linked list, and gathers the corresponding
% values (in the logical order of the list) in the cell
% array "output".

output = {};
current = linked_list.root;

while current ~= 0
    output{end+1} = linked_list.node(current).value;
    current = linked_list.node(current).next;
end

end
```

Traversing a linked list

```
>> list.root = 2;
>> list.node = struct('value', 'delta', 'next', 0);
>> list.node(2) = struct('value', 'alpha', 'next', 4);
>> list.node(3) = struct('value', 'gamma', 'next', 1);
>> list.node(4) = struct('value', 'beta', 'next', 3);

>> % Physical order of the data
>> {list.node(:).value}
ans =
  1x4 cell array
    'delta'     'alpha'     'gamma'     'beta'

>> % Logical order of the data
>> values = my_ll_traverse(list)
values =
  1x4 cell array
    'alpha'     'beta'     'gamma'     'delta'
```

Deleting nodes from a linked list

We can "delete" nodes without physically removing them from memory (watch out for memory leaks!), by re-setting the "next" values of the appropriate nodes to skip the "deleted" nodes

```
function [linked_list] = my_ll_delete(linked_list, value)

% Delete all nodes whose value is "value" from the linked list.

previous = 0;
current = linked_list.root;

while current ~= 0
    next = linked_list.node(current).next;
    if strcmp(linked_list.node(current).value, value)
        linked_list.node(current).value = '';
        linked_list.node(current).next = 0;
        if previous == 0
            % We just deleted the first node
            linked_list.root = next;
        else
            linked_list.node(previous).next = next;
        end
    else
        previous = current;
    end
    current = next;
end

end
```

Deleting nodes from a linked list

```
>> list.root = 2;
>> list.node = struct('value', 'delta', 'next', 0);
>> list.node(2) = struct('value', 'alpha', 'next', 4);
>> list.node(3) = struct('value', 'gamma', 'next', 1);
>> list.node(4) = struct('value', 'beta', 'next', 3);

>> [list] = my_ll_delete(list, 'gamma');

>> % Physical order of the data
>> {list.node(:).value}
ans =
  1x4 cell array
    'delta'    'alpha'    ''    'beta'

>> % Logical order of the data
>> values = my_ll_traverse(list)
values =
  1x3 cell array
    'alpha'    'beta'    'delta'
```

Linked lists

Advantages:

- We can sort, insert, and remove elements from a linked list without having to move data around in memory (as opposed to, for example, inserting an element in the middle of a large vector of class `double`)
- Thus, it makes it easier to keep the list sorted as nodes are added and/or removed

Disadvantages:

- Need to come up with alternate ways to search the list; we cannot easily apply the bisection method used earlier for sorted vectors of class `double`

Examples of additional functions for maintaining linked lists:

- Add an element to the list (in correct order)
- Compress linked list to get rid of unused nodes