L36: More Sorting

And queues and stacks

Lucas A. J. Bastien

E7 Spring 2017, University of California at Berkeley

April 21, 2017

Version: release

Announcements

Project is due on Friday April 28 at 11:59 pm (midnight)

- **Your code MUST work on the computers in 1109 Etcheverry Hall**
  (It is where we will do the grading)
  - Do not use functions from Toolboxes not installed on these computers
  - Your personal computer may be faster than these computers
- **It is your responsibility to make sure that your code works on these computers**
  - **We will NOT debug your code before grading it**

Today:

- More sorting, queues, and stacks

Next week:

- Special topics, guest lecture(s), teaching evaluations

Time complexity of sorting algorithms

What is the theoretical “minimum” (“minimum” as in “more advantageous”) time complexity of any given sorting algorithm?

(A) $\mathcal{O}(\log(n))$

(B) $\mathcal{O}(n \log(n))$

(C) [✓ $\mathcal{O}(n)$]

(D) $\mathcal{O}(n^2)$

(E) $\mathcal{O}(\mathrm{constant}^n)$, with $\mathrm{constant} > 1$

At the very least, we have to look at each value at least once

Time complexity of sorting algorithms

Selection sort: at each iteration, look at the values that remain to be sorted, find the minimum, and move the minimum to the end of the part of the list that is already sorted

Example:

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

Time complexity of sorting algorithms

Selection sort to sort a list of n elements

First step: look at n elements to find the minimum
Second step: look at $(n-1)$ elements to find the remaining minimum
Third step: look at $(n-2)$ elements to find the remaining minimum
...

Number of operations:

$$\approx n + (n-1) + (n-2) + \cdots + 2 + 1$$
$$= \frac{n+1}{2} \times n = \frac{n^2+n}{2}$$

The time complexity of selection sort is $\mathcal{O}(n^2)$, no matter how ordered the list is to start with

Time complexity of sorting algorithms

Insertion sort: at each iteration, take the next unsorted value, and insert it in its place in the part of the list that is already sorted

Example:

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

Time complexity of sorting algorithms

Selection sort to sort a list of n elements

Worst case scenario: the values are in reverse order. At each step, we shift the entire sorted part of the list
Time complexity: $\mathcal{O}(n^2)$
(similar “proof” as with selection sort)

Best case scenario: the values are already sorted. We go through the list once, and we do not do any shifting
Time complexity: $\mathcal{O}(n)$

On average: **Time complexity:** $\mathcal{O}(n^2)$

Time complexity of sorting algorithms

Quick sort: **divide and conquer**. At each step, choose a pivot and divide the elements of the list into three (unsorted) piles

1. Smaller than the pivot
2. Equal to the pivot
3. Greater than the pivot

and sort piles 1 and 3 separately

Worst-case scenario: at each step, all the values are in one single pile

- First step: there remain $(n-1)$ values to sort
- Second step: there remain $(n-2)$ values to sort
- Third step: there remain $(n-3)$ values to sort
- ...

Time complexity: $\mathcal{O}(n^2)$

Time complexity of sorting algorithms

Quick sort: **divide and conquer**. At each step, choose a pivot and divide the elements of the list into three (unsorted) piles

1. Smaller than the pivot
2. Equal to the pivot
3. Greater than the pivot

and sort piles 1 and 3 separately

Best-case scenario: at each step, piles 1 and 3 are of equal size. At each step:

- We have to look at all the values to separate values in piles ($\mathcal{O}(n)$)
- We divide the size of the problem by 2 ($\mathcal{O}(\log_2(n)$)

Time complexity: $\mathcal{O}(n \log_2(n))$

On average: **Time complexity:** $\mathcal{O}(n \log_2(n))$

Queues and Stacks

Queues and Stacks: Structures that provide sequencing for the order in which data elements/tasks are processed

Queues

"FIFO": First-In First-Out

Requests/data at the front of the line get handled first

New data are added to the end of the queue

Examples: line at the register

Stacks

"FILO": First-In Last-Out

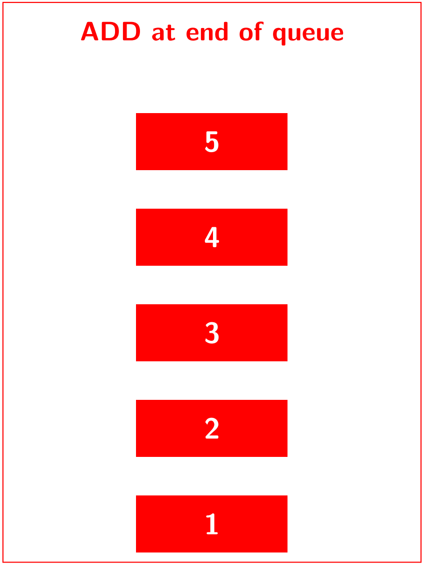
Process data off the top of the stack first

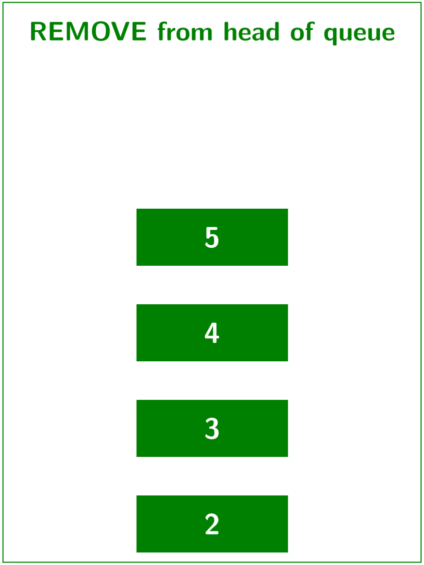
New data are added to the top of the stack

Examples: packed elevator, moving boxes

Queue operations

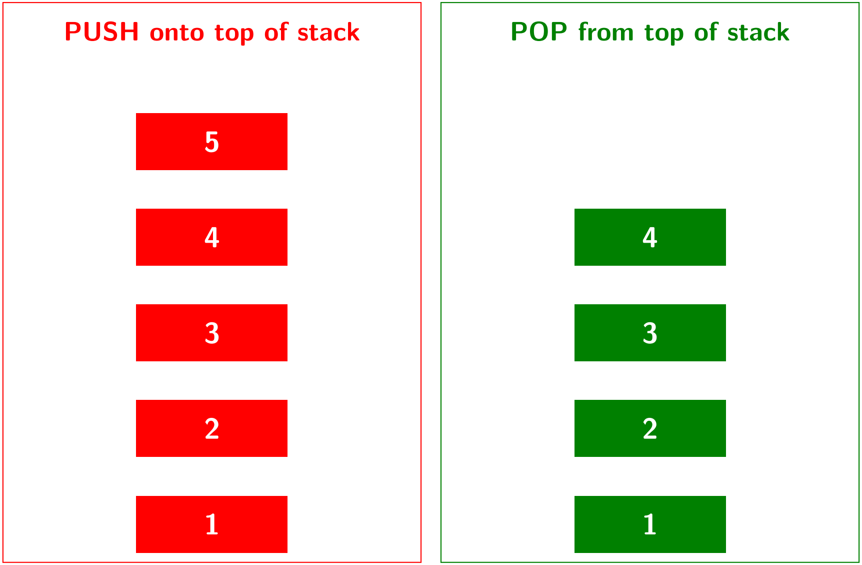
Basic operations are **ADD** and **REMOVE**





Stack operations

Basic operations are **PUSH (ADD)** and **POP (REMOVE)**



Quicksort algorithm (recursive)

```
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
```

Quicksort algorithm, using a stack

See `my_sort_quicksort_stack.m`

- This implementation does not use recursion
- **Use a stack to keep track of which sub-sections of the vector remain to be sorted.** Each entry in the stack contains the "start" and "end" indices of a sub-section of the vector that remains to be sorted
- Unlike the recursive implementation, **this implementation sorts values "in place"** *i.e.* values are moved around in the original vector, without creating temporary vectors that contain sub-sections of the vector

Quicksort algorithm: recursion versus stack

```
>> rng(0)
>> a = randi([1, 1e4], [1, 1e4]);

>> % Quicksort with recursive implementation
>> tic(); for i = 1:100; my_sort_quicksort(a); end; toc()
Elapsed time is 2.949985 seconds.

>> % Quicksort using a stack but no recursion
>> tic(); for i = 1:100; my_sort_quicksort_stack(a); end; toc()
Elapsed time is 0.512825 seconds.
```

Using a stack to keep track of the sections of the vector that remain to be sorted, and sorting the values in place (thereby reducing memory usage) gives results many times faster than the original recursive quicksort implementation

Swapping values in an array: efficiency

```
% Create a large vector with pseudo-random values
rng(0)
vector = randi([1, 1e4], [1, 1e4]);

% Swap two arbitrary values in the vector
index_1 = 100;
index_2 = 500;

% The following method to swap values is relatively slow
tic();
n_repeat = 1000000;
for i = 1:n_repeat
    vector([index_1, index_2]) = vector([index_2, index_1]);
end
toc()

Elapsed time is 1.674436 seconds.

% The following method is much faster!
tic();
for i = 1:n_repeat
    temporary = vector(index_1);
    vector(index_1) = vector(index_2);
    vector(index_2) = temporary;
end
toc()

Elapsed time is 0.012809 seconds.
```