Sorting

Lecture 17

Menu

- Binary Search
- Sorting
  - approaches
  - selection sort
  - insertion sort
  - bubble sort
  - analysis
  - fast sorts

Binary Search: Cost

- What is the cost of searching if *n* items in set?
  - key step = ?

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

- Iteration / Size of range / Cost of iteration

Iteration	Size of range	Cost of iteration
1	n	
2		
k	1	

$\text{Log}_2(n)$ or $\log(n)$:

The number of times you can divide a set of *n* things in half.

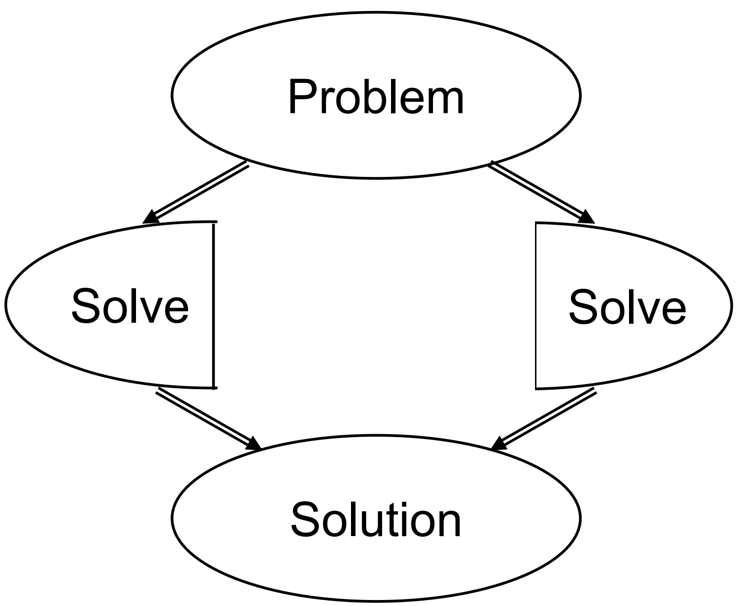
log(1000) =10, log(1,000,000) = 20, log(1,000,000,000) =30

Every time you double *n*, you add one step to the cost!

$$2^{\log(x)} = x$$

- Arises all over the place in analysing algorithms

  Especially "Divide and Conquer" algorithms:



SortedArraySet algorithms

Assume data is sorted

Contains(value):

index = findIndex(value),
return (data[index] equals value)

Add(value):

index = findIndex(value),
if index == count or data[index] not equal to value,
double array if necessary
move items up, from position count-1 down to index
insert value at index
increment count

Remove(value):

index = findIndex(value),
if index < count and data[index] equal to value
move items down, from position index+1 to count-1
decrement count

Sets with Binary Search

ArraySet: unordered

- contains: O(n)
- add: O(n)
- remove: O(n)

⇒ All the cost is in the searching: O(n)

SortedArraySet: with Binary Search

- Binary Search is fast: O(log(n))
  - contains: O(log(n))
  - add:
  - remove:

⇒ All the cost is in keeping it sorted!!!!

Making SortedArraySet fast

- If you have to call add() and/or remove() many items, then SortedArraySet is no better than ArraySet!
  - Both O(*n*)
  - Either pay to search
  - Or pay to keep it in order

- If you construct the set once but many calls to contains(), then SortedArraySet is much better than ArraySet.
  - SortedArraySet contains() is O(log(*n*))

- But, how do you construct the set fast?

Why Sort?

- Constructing a sorted array one item at a time is very slow :
  - add(item) is n/2 on average [O(n)]
    $\Rightarrow$ 1/2 + 2/2 + 3/2 + …. n/2 is $n^2/4$ [$O(n^2)$]
  - $\approx$ 2,500,000,000,000,000 steps for 100,000,000 items
    $\Rightarrow$ 25,000,000 seconds = 289 days at 10ns per step.

- There are sorting algorithms that are much faster if you can sort whole array at once: O(n log(n))
  - $\approx$ 2,700,000,000 steps for 100,000,000 items
    $\Rightarrow$ 27 seconds at 10ns per step.

Alternative Constuctor

- Sort the items all at once

```
public SortedArraySet(Collection<E> colln){
    // Make space
    count=colln.size();
    data = (E[]) new Object[count];
    // Put items from collection into the data array.
    colln.toArray(data);
    // sort the data array.
    Arrays.sort(data);
}
```

- How do you sort?

Sort them!

Rat	Pig	Owl	Kea	Fox	Hen	Yak	Ant_1	Tui	Dog	Cat	Man	Jay	Bee	Eel	Gnu	Ant_2	Sow

73	3	6931	427	5	45	463	941	7273	64	9731	61	873	44	74	465	6929	75

How to do it?

Ways of sorting

- Selecting sorts:
  - Find the next largest/smallest item and put in place
  - Builds the correct list in order
- Inserting Sorts:
  - For each item, insert it into an ordered sublist
  - Builds a sorted list, but keeps changing it
- Compare and Swap Sorts:
  - Find two items that are out of order, and swap them
  - Keeps “improving” the list
- Radix Sorts
  - Look at the item and work out where it should go.
  - Only works on some kinds of values.
- …

Analysing Sorting Algorithms

- Efficiency
  - What is the (worst-case) order of the algorithm?
  - Is the average case much faster than worst-case?
- Requirements on Data
  - Does the algorithm need random-access data? (*vs* streaming data)
  - Does it need anything more than “compare” and “swap”?
- Space Usage
  - Can the algorithm sort in-place? or does it need extra space?

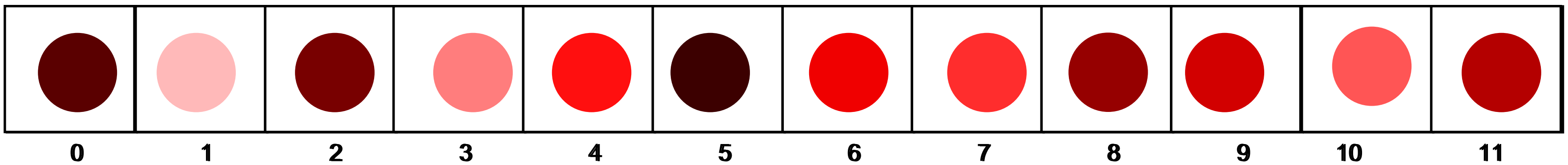
Selection Sort – Example

- sort (34, 10, 64, 51, 32, 21) in ascending order

Sorted part	Unsorted part	Swapped
	34 **10** 64 51 32 21	10, 34
10	34 64 51 32 **21**	21, 34
10 21	64 51 **32** 34	32, 64
10 21 32	51 64 **34**	51, 34
10 21 32 34	64 **51**	51, 64
10 21 32 34 51	**64**	--
10 21 32 34 51 64		

In-place sorting

Inserting Sorts



- Insertion Sort (slow)
- Merge Sort (fast) (Divide and Conquer)

Insertion Sort

look at elements one by one

build up sorted list by *inserting* the element at the correct location

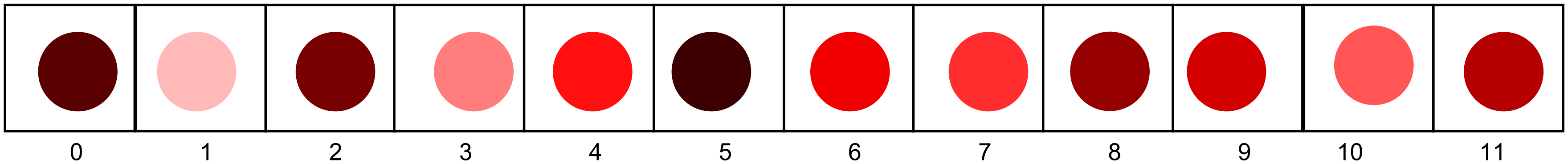
Example

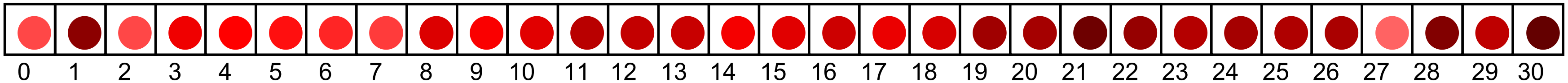
- sort (34, 8, 64, 51, 32, 21) in ascending order

Sorted part	Unsorted part	int moved
	34 8 64 51 32 21	
34	**8** 64 51 32 21	-
8 34	**64** 51 32 21	34
8 34 64	**51** 32 21	-
8 34 51 64	**32** 21	64
8 32 34 51 64	**21**	34, 51, 64
8 21 32 34 51 64		32, 34, 51, 64

In-place sorting

Compare and Swap Sorts





- Bubble Sort (terrible)
- QuickSort (the fastest) (Divide and Conquer)

Bubble Sort

- starting from the last element, swap adjacent items if they are not in ascending order
- when first item is reached, the first item is the smallest
- repeat the above steps for the remaining items to find the second smallest item, and so on

In-place sorting

Bubble Sort – Example

round	(34	10	64	51	32	21)
	34	10	64	51	**32**	**21**
1	34	10	64	**51**	**21**	32
	34	10	**64**	**21**	51	32
	34	**10**	**21**	64	51	32
	34	**10**	21	64	51	32
	10	34	21	64	**51**	**32**
2	*10*	34	21	**64**	**32**	51
	10	34	**21**	**32**	64	51
	10	**34**	**21**	32	64	51
	10	*21*	34	32	64	51

Bubble Sort – Example (2)

round						
	10	*21*	34	32	**64**	**51**
3	*10*	*21*	34	**32**	**51**	64
	10	*21*	**34**	**32**	51	64
	10	*21*	*32*	34	**51**	**64**
4	*10*	*21*	*32*	**34**	**51**	64
	10	*21*	*32*	*34*	**51**	**64**
5	*10*	*21*	*32*	*34*	*51*	64

bubble sort

- also called as sinking sort
- smaller values bubble their way up during the process
- need several passes through the data (how many?)
- During each pass, successive pairs of elements are compared
  - if a pair is in order, leave them as they are
  - if a pair not in order, swap

Implementing Sorting Algorithms

- Could sort Lists
  - ⇒ general and flexible
  - but efficiency depends on how the List is implemented
- Could sort Arrays.
  - ⇒ less general
  - but efficiency is well defined
  - easy to convert any Collection to an array:
    toArray() method.
- Comparing items:
  - require items to be comparable (natural order)
  - provide comparator (prescribed order)
  - handle both.

Sort methods

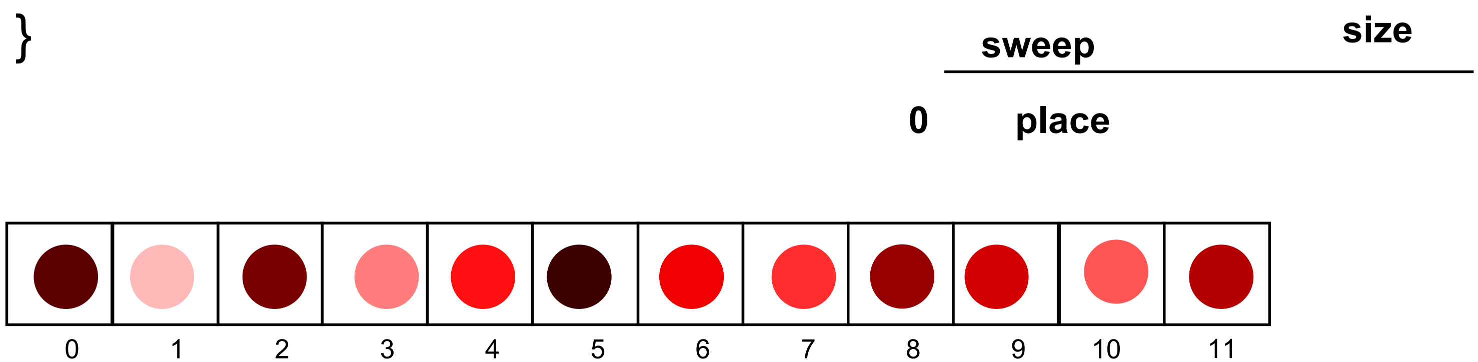
We will use:

public void …Sort(E[] data, int size, Comparator<E> comp)

- sorts first *size* elements of an array of some type, given a Comparator for that type.
- Could be used inside SortedArraySet, or standalone.
- Designing very flexible code that can be used in many different places is tricky!!

Selection Sort

```
public void selectionSort(E[ ] data, int size, Comparator<E> comp){
    // for each position, from 0 up, find the next smallest item
    // and swap it into place
    for (int place=0; place<size-1; place++){
        int minIndex = place;
        for (int sweep=place+1; sweep<size; sweep++){
            if (comp.compare(data[sweep], data[minIndex]) < 0)
                minIndex=sweep;
            }
        swap(data, place, minIndex);
    }
}
```



Selection Sort Analysis

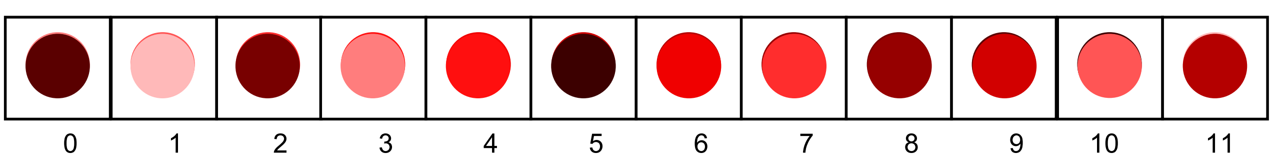
- Cost:
  - step?
  - best case:
    - what is it?
    - cost:
  - worst case:
    - what is it?
    - cost:
  - average case:
    - what is it?
    - cost:

Selection Sort Analysis

- Efficiency
  - worst-case: **$O(n^2)$**
  - average case exactly the same.
- Requirements on Data
  - Needs random-access data, but easy to modify for files
  - Needs compare and swap
- Space Usage
  - in-place

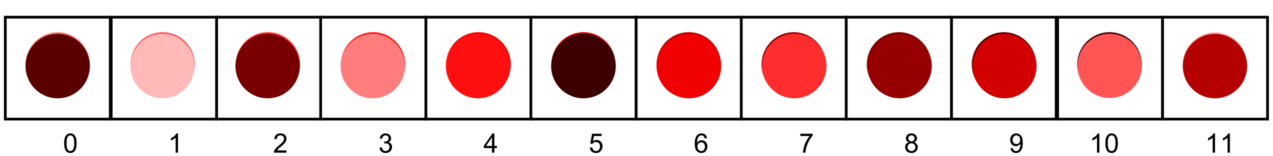
Exercise: Bubble Sort Implementation

```
public  void bubbleSort(E[] data, int size, Comparator<E> comp){
    // Repeatedly scan array, swapping adjacent items if out of order
    // Builds a sorted region from the end








}
```



Bubble Sort

```
public  void bubbleSort(E[] data, int size, Comparator<E> comp){
    // Repeatedly scan array, swapping adjacent items if out of order
    // Builds a sorted region from the end
    for (int top=size-1; top>0; top--){
        for (int sweep=0; sweep<top; sweep++)
            if (comp.compare(data[sweep], data[sweep+1]) >0)
                swap(data, sweep, sweep+1);
    }
}
```



Bubble Sort Analysis

- Cost:
  - step?
  - best case:
    - what is it?
    - cost:
  - worst case:
    - what is it?
    - cost:
  - average case:
    - what is it?
    - cost:

Bubble Sort Analysis

- Efficiency
  - worst-case: **$O(n^2)$**
  - average case: **$O(n^2)$,** no better than worst
- Requirements on Data
  - Needs random-access data, but can modify for files
  - Needs compare and swap
- Space Usage
  - in-place

Slow Sorts

- Insertion sort, Selection Sort, Bubble Sort:
  - All slow (except Insertion sort on almost sorted lists)
  - Insertion sort is better than the others
- Problem:
  - Insertion and Bubble
    - only compare adjacent items
    - only move items one step at a time
  - Selection
    - compares every pair of items –
      - ignores results of previous comparisons.
- Solution:
  - **Must be able to compare and swap items at a distance**
  - **Must not perform redundant comparisons**

Summary

- Binary Search
- Sorting
  - approaches
  - selection sort
  - insertion sort
  - bubble sort
  - analysis
  - fast sorts

Readings

- [Mar07] Read 7.1, 7.2, 7.3
- [Mar13] Read 7.1, 7.2, 7.3