COSC 222 Data Structure

Sorting - Part 1

Sorting

- **Sorting**: Rearranging the values in an array or collection into a specific order (usually into their "natural ordering").
  - one of the fundamental problems in computer science
  - can be solved in many ways:
    there are many sorting algorithms
    some are faster/slower than others
    some use more/less memory than others
    some work better with specific kinds of data
    some can utilize multiple computers / processors, ...

Selection sort

- **selection sort**: Orders a list of values by repeatedly putting the smallest or largest unplaced value into its final position.

- The algorithm:
  - Look through the list to find the smallest value.
  - Swap it so that it is at index 0.
  - Look through the list to find the second-smallest value.
  - Swap it so that it is at index 1.
  - ...
  - Repeat until all values are in their proper places.

Selection sort example

- Initial array:

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
value	22	18	12	-4	27	30	36	50	7	68	91	56	2	85	42	98	25

- After 1st pass:

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
value	-4	18	12	22	27	30	36	50	7	68	91	56	2	85	42	98	25

- After 2nd pass:

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
value	-4	2	12	22	27	30	36	50	7	68	91	56	18	85	42	98	25

- After 3rd pass:

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
value	-4	2	7	22	27	30	36	50	12	68	91	56	18	85	42	98	25

Selection sort code

```
public static void selectionSort(int[] a) {
   for (int pass = 0; pass < a.length; pass++) {
       int min = pass;
       for (int j = pass + 1; j < a.length; j++) {
          if (a[j] < a[min]) {
              min = j;
          }
       }
       swap(a, pass, min);
   }
}
```

Running time (# comparisons) for input size *N*: $O(N^2)$

Best/worst/average case?

Bubble sort

- **bubble sort**: orders a list of values by repetitively comparing neighboring elements and swapping their positions if necessary
- Traverse a collection of elements
  - Move from the front to the end
  - "Bubble" largest value to end using pair comparisons and swapping

"Bubbling" largest element

- What can you assume about the array's state afterward?

index	0	1	2	3	4	5
value	42	77	35	12	91	8

42 77

35 77

12 77

77 91

8 91

value	42	35	12	77	8	91

"Bubbling" largest element

- What can you assume about the array's state afterward?

index	0	1	2	3	4	5
value	42	77	35	12	91	8

42 77

35 77

12 77

77 91

8 91

value	42	35	12	77	8	91

index	0	1	2	3	4	5
value	42	35	12	77	8	91

35 42

12 42

42 77

8 77

77

value	35	12	42	8	77	91

After 5 passes: 8 12 35 42 77 91

Bubble sort code

```
public static void bubbleSort(int[] a) {
   for (int pass = 0; pass < a.length - 1; pass++) {
       boolean changed = false;
       for (int i = 0; i < a.length - 1 - pass; i++) {
           if (a[i] > a[i + 1]) {
               swap(a, i, i + 1);
               changed = true;
            }
       }
       if (!changed) {     // exit early if in sequence
              return;
       }
   }
}
```

Bubble sort time complexity

- Worst case scenarios:
  - The array is in reverse order
  - n-1 comparisons need in the 1st pass
  - n-2 in the 2nd pass
  - n-3 in the 3rd pass

  The total number of comparisons will be

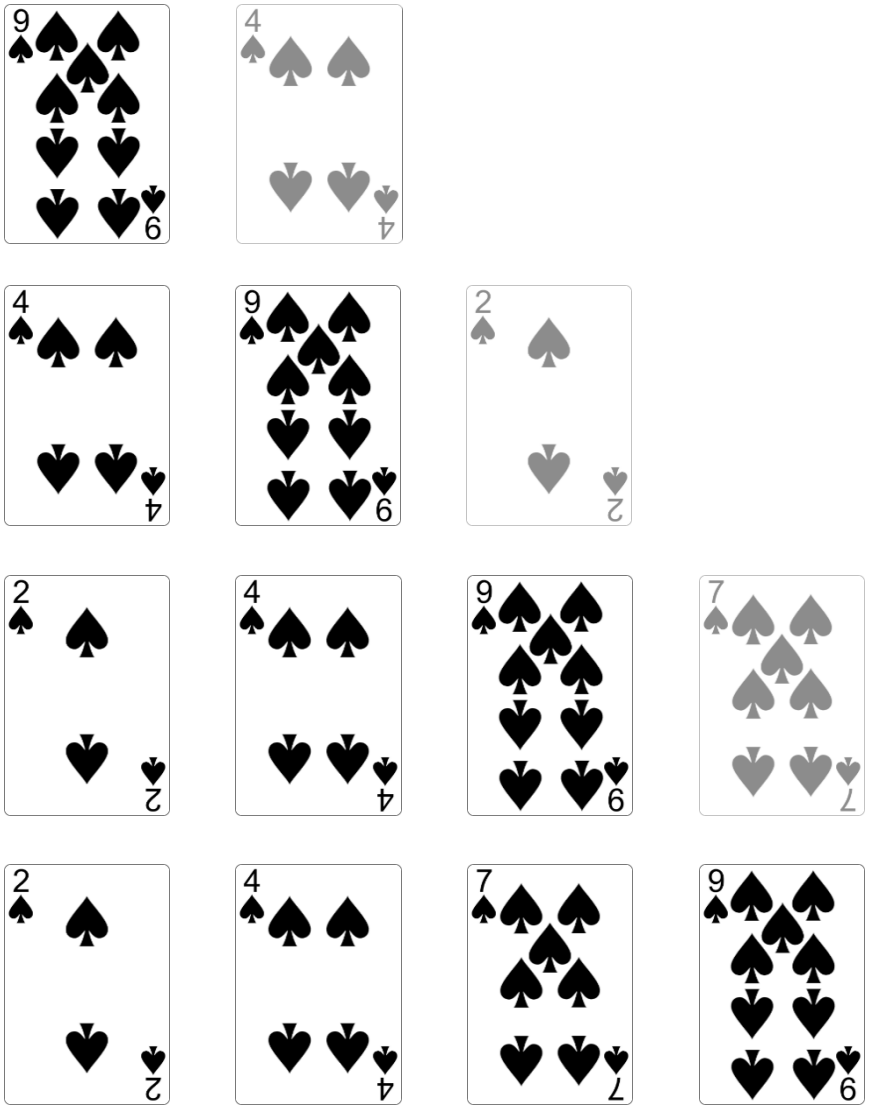
  $(n-1) + (n-2) + (n-3) + ... + 3 + 2 + 1$

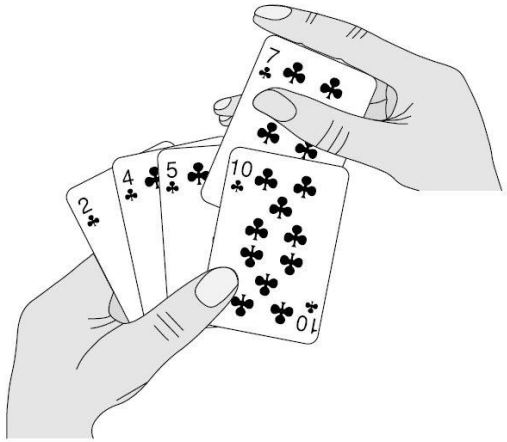
  $= n(n-1)/2$

  i.e., $O(n^2)$

- What is the best case?
- What is the average case?

Insertion sort





Insertion sort

- **insertion sort**: orders a list of values by repetitively inserting a particular value into a sorted subset of the list

- more specifically:
  - consider the first item to be a sorted sublist of length 1
  - insert second item into sorted sublist, shifting first item if needed
  - insert third item into sorted sublist, shifting items 1-2 as needed
  - ...
  - repeat until all values have been inserted into their proper positions

Insertion sort

- Makes N-1 passes over the array.
- At the end of pass i, the elements that occupied A[0]...A[i] originally are still in those spots and in sorted order.

index	0	1	2	3	4	5	6	7
value	15	2	8	1	17	10	12	5
pass 1	**2**	**15**	8	1	17	10	12	5
pass 2	**2**	**8**	**15**	1	17	10	12	5
pass 3	**1**	**2**	**8**	**15**	17	10	12	5
pass 4	**1**	**2**	**8**	**15**	**17**	10	12	5
pass 5	**1**	**2**	**8**	**10**	**15**	**17**	12	5
pass 6	**1**	**2**	**8**	**10**	**12**	**15**	**17**	5
pass 7	**1**	**2**	**5**	**8**	**10**	**12**	**15**	**17**

Insertion sort code

```
// Rearranges the elements of a into sorted order.
public static void insertionSort(int[] a) {
    for (int i = 1; i < a.length; i++) {
        int temp = a[i];

        // slide elements right to make room for a[i]
        int j = i;
        while (j >= 1 && a[j - 1] > temp) {
            a[j] = a[j - 1];
            j--;
        }
        a[j] = temp;
    }
}
```

Insertion sort time complexity

- *worst case:* reverse-ordered elements in array.

  The total number of comparisons will be

  $(n-1) + (n-2) + (n-3) + ... + 3 + 2 + 1$

  $= n(n-1)/2$

  i.e., $O(n^2)$

  - What is the best case?
  - What is the average case?

Shell sort

- **shell sort**: orders a list of values by comparing elements that are separated by a gap of >1 indexes
  - a generalization of insertion sort
  - invented by computer scientist Donald Shell in 1959

- based on some observations about insertion sort:
  - insertion sort runs fast if the input is almost sorted
  - insertion sort's weakness is that it swaps each element just one step at a time, taking many swaps to get the element into its correct position

  - Runs a lot faster on already-sorted ascending input than random input (no shifting needed). Also works well on descending input.

Shell sort example

- For some sequence of gaps $g_1, g_2, g_3, \ldots, 1$:
  - Sort all elements that are g_1 indexes apart
  - Then sort all elements that are g_2 indexes apart, ...
  - Then sort all elements that are 1 index apart

- An example that sorts by gaps of 8, then 4, then 2, then 1:

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
start	27	88	92	-4	22	30	36	50	7	18	11	76	2	65	56	3	85
gap 8	7	18	11	-4	2	30	36	3	27	88	92	76	22	65	56	50	85
gap 4	2	18	11	-4	7	30	36	3	22	65	56	50	27	88	92	76	85
gap 2	2	-4	7	3	11	18	22	30	27	50	36	65	56	76	85	88	92
gap 1	-4	2	3	7	11	18	22	27	30	36	50	56	65	76	85	88	92

Shell sort code

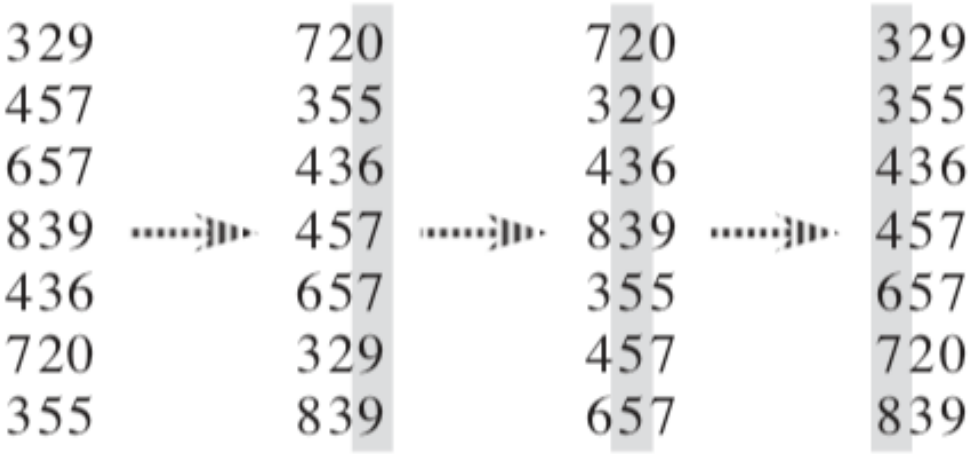
```
// Rearranges the elements of a into sorted order.
// Uses a shell sort with gaps that divide by 2:
// length/2, length/4, ..., 4, 2, 1
public static void shellSort(int[] a) {
    for (int gap = a.length / 2; gap >= 1; gap /= 2) {
        for (int i = gap; i < a.length; i++) {
            // slide elements right by 'gap'
            // to make room for a[i]
            int temp = a[i];
            int j = i;
            while (j >= gap && a[j - gap] > temp) {
                a[j] = a[j – gap];
                j -= gap;
            }
            a[j] = temp;
        }
    }
}
```

Shell sort time complexity

- worst case: $O(n^2)$
- best case: $O(n \log n)$
- average case: between $O(n \log n)$ and $O(n^2)$
  - depends on gap size chosen

Radix sort

- **radix sort**:
  - Sorts elements by grouping the individual digits of the same place value
  - also called ‘bucket sort’, as items are distributed into a set of ‘buckets’
  - is useful for non-numerical data, such as words or punchcards



Radix sort example

- input array *a*:

index	0	1	2	3	4	5	6	7	8	9	10	11
value	714	128	206	34	722	8	142	533	646	29	240	373

- sort by last digit, then by tens digit, then by hundreds digit:

index	0	1	2	3	4	5	6	7	8	9	10	11
value	240	722	142	533	373	714	034	206	646	008	128	029

index	0	1	2	3	4	5	6	7	8	9	10	11
value	206	008	714	722	128	029	533	034	240	142	646	373

index	0	1	2	3	4	5	6	7	8	9	10	11
value	008	029	034	128	142	206	240	373	533	646	714	722

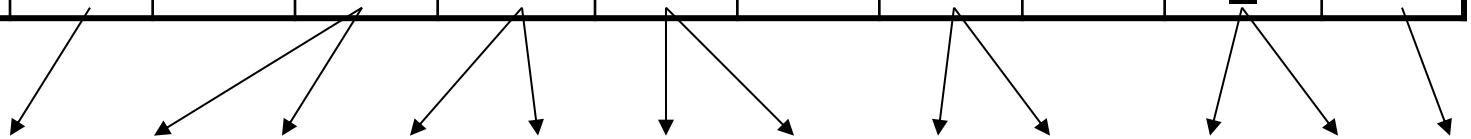
Radix sort, detailed

- input array *a*:

index	0	1	2	3	4	5	6	7	8	9	10	11
value	71**4**	12**8**	20**6**	3**4**	72**2**	**8**	14**2**	53**3**	64**6**	2**9**	24**0**	37**3**

- create array of queues, ordered by last digit:

index	0	1	2	3	4	5	6	7	8	9
value	24**0**		72**2**, 14**2**	53**3**, 37**3**	71**4**, 3**4**		20**6**, 64**6**		12**8**, **8**	2**9**



index	0	1	2	3	4	5	6	7	8	9	10	11
value	240	722	142	533	373	714	34	206	646	128	8	29

- put elements back into *a*, sorted by last digit. ...

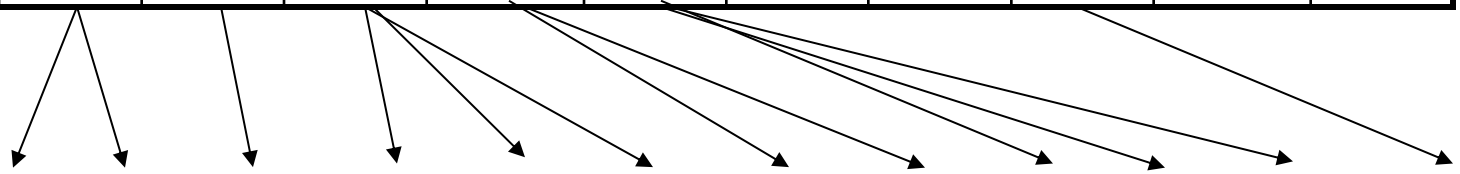
Radix sort, detailed – second pass

- input array *a*:

index	0	1	2	3	4	5	6	7	8	9	10	11
value	240	722	142	533	373	714	34	206	646	128	08	29

- insert into queues, ordered by second last digit:

index	0	1	2	3	4	5	6	7	8	9
value	206 08	714	722 128 29	533 34	240 142 646			373		



index	0	1	2	3	4	5	6	7	8	9	10	11
value	206	8	714	722	128	29	533	34	240	142	646	373

- put elements back into a, sorted by last two digits. ...

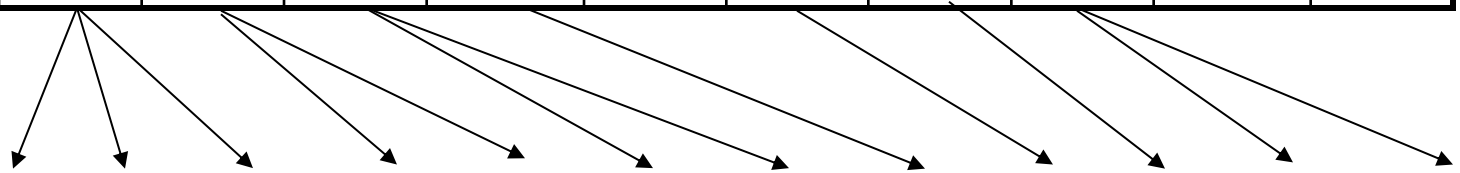
Radix sort, detailed – thrid pass

- input array *a*:

index	0	1	2	3	4	5	6	7	8	9	10	11
value	206	008	714	722	128	029	533	034	240	142	646	373

- insert into queues, ordered by thrid last digit:

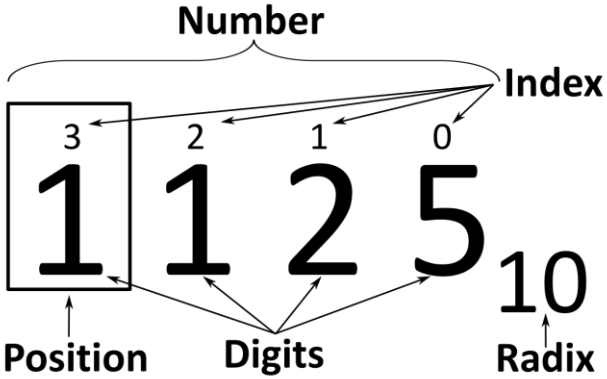
index	0	1	2	3	4	5	6	7	8	9
value	008 029 034	128 142	206 240	373		533	646	714 722		



index	0	1	2	3	4	5	6	7	8	9	10	11
value	8	29	34	128	142	206	240	373	533	646	714	722

- put elements back into a, sorted by last three digits. ...

Radix sort



The algorithm:

- Given an array of numbers, *a*:
- Create an array of queues *C* of size *k*, where *k* is the radix (base).

  i.e. create *k* 'buckets', one for each numerical position
- For each position *i* from least to most significant:

  For each element in *a:*

  if its digit at position *i* has value is *k*, add it to queue *C*[*k*].

  Dequeue the items in *C* to place *a*'s contents back in sorted order.

Radix sort time complexity

- worst: O(k N) for N elements with k digits each
- best/average case?
- performs best with fewer digits
- Requires additional space for the buckets and storage of sorted output

Sorting Visualisation Tools

- Cool visualisation tool here:
  - https://www.cs.usfca.edu/~galles/visualization/ComparisonSort.html
- Loads of others online

Thank you