CSE 143
Lecture 26

quick sort

slides adapted from Marty Stepp
http://www.cs.washington.edu/143/

Quick sort

- **quick sort**: Orders a list of values by partitioning the list around one element called a *pivot*, then sorting each partition.
  - invented by British computer scientist C.A.R. Hoare in 1960

- Quick sort is another divide and conquer algorithm:
  - Choose one element in the list to be the pivot.
  - *Divide* the elements so that all elements less than the pivot are to its left and all greater (or equal) are to its right.
  - *Conquer* by applying quick sort (recursively) to both partitions.

- Runtime: $O(N \log N)$ average, $O(N^2)$ worst case.
  - Generally somewhat faster than merge sort.

Choosing a "pivot"

- The algorithm will work correctly no matter which element you choose as the pivot.
  - A simple implementation can just use the first element.

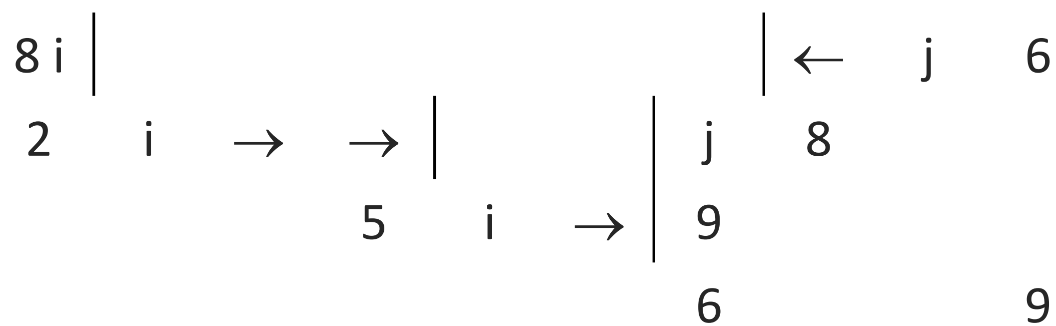
- But for efficiency, it is better if the pivot divides up the array into roughly equal partitions.
  - What kind of value would be a good pivot? A bad one?

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

Partitioning an array

- Swap the pivot to the last array slot, temporarily.
- Repeat until done partitioning (until *i*,*j* meet):
  - Starting from *i* = 0, find an element a[*i*] ≥ pivot.
  - Starting from *j* = *N*-1, find an element a[*j*] ≤ pivot.
  - These elements are out of order, so swap a[*i*] and a[*j*].
- Swap the pivot back to index *i* to place it between the partitions.

index	0	1	2	3	4	5	6	7	8	9
value	6	1	4	9	0	3	5	2	7	8



2	1	4	5	0	3	6	8	7	9

Quick sort example

index	0	1	2	3	4	5	6	7	8	9	
value	**65**	23	81	43	92	39	57	16	75	32	choose pivot=65
	32	23	81	43	92	39	57	16	75	**65**	swap pivot (65) to end
	32	23	**16**	43	92	39	57	**81**	75	65	swap 81, 16
	32	23	16	43	**57**	39	**92**	81	75	65	swap 57, 92
	32	23	16	43	57	39	92	81	75	65	
	32	23	16	43	57	39	**65**	81	75	**92**	swap pivot back in

recursively quicksort each half

32	23	16	43	57	39	pivot=32
39	23	16	43	57	**32**	swap to end
16	23	**39**	43	57	32	swap 39, 16
16	23	**32**	43	57	**39**	swap 32 back in

...

81	75	92	pivot=81
92	75	**81**	swap to end
75	92	81	swap 92, 75
75	**81**	**92**	swap 81 back in

...

Quick sort code

```
public static void quickSort(int[] a) {
    quickSort(a, 0, a.length - 1);
}
private static void quickSort(int[] a, int min, int max) {
    if (min >= max) {  // base case; no need to sort
        return;
    }

    // choose pivot; we'll use the first element (might be bad!)
    int pivot = a[min];
    swap(a, min, max);        // move pivot to end

    // partition the two sides of the array
    int middle = partition(a, min, max - 1, pivot);

    swap(a, middle, max);    // restore pivot to proper location

    // recursively sort the left and right partitions
    quickSort(a, min, middle - 1);
    quickSort(a, middle + 1, max);
}
```

Partition code

```
// partitions a with elements < pivot on left and
// elements > pivot on right;
// returns index of element that should be swapped with pivot
private static int partition(int[] a, int i, int j, int pivot) {
    while (i <= j) {
        // move index markers i,j toward center
        // until we find a pair of out-of-order elements
        while (i <= j && a[i] < pivot) { i++; }
        while (i <= j && a[j] > pivot) { j--; }

        if (i <= j) {
            swap(a, i, j);
            i++;
            j--;
        }
    }

    return i;
}
```

Choosing a better pivot

- Choosing the first element as the pivot leads to very poor performance on certain inputs (ascending, descending)
  - does not partition the array into roughly-equal size chunks

- Alternative methods of picking a pivot:
  - *random:* Pick a random index from [*min* .. *max*]
  - *median-of-3:* look at left/middle/right elements and pick the one with the medium value of the three:
    - `a[min]`, `a[(max+min)/2]`, and `a[max]`
    - better performance than picking random numbers every time
    - provides near-optimal runtime for almost all input orderings

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
value	8	18	91	-4	27	30	86	50	65	78	5	56	2	25	42	98	**31**

Stable sorting

- **stable sort**: One that maintains relative order of "equal" elements.
  - important for secondary sorting, e.g.
    - sort by name, then sort again by age, then by salary, ...

- All of the N^2 sorts shown are stable, as is shell sort.
  - bubble, selection, insertion, shell
- Merge sort is stable.
- Quick sort is *not* stable.
  - The partitioning algorithm can reverse the order of "equal" elements.
  - For this reason, Java's Arrays/Collections.sort() use merge sort.

Unstable sort example

- Suppose you want to sort these points by Y first, then by X:
  - `[(4, 2), (5, 7), (3, 7), (3, 1)]`
- A stable sort like merge sort would do it this way:
  - `[(3, 1), (4, 2), (5, 7), (3, 7)]` sort by y
  - `[(3, 1), (3, 7), (4, 2), (5, 7)]` sort by x
  - Note that the relative order of (3, 1) and (3, 7) is maintained.
- Quick sort might leave them in the following state:
  - `[(3, 1), (4, 2), (5, 7), (3, 7)]` sort by y
  - `[(3, 7), (3, 1), (4, 2), (5, 7)]` sort by x
  - Note that the relative order of (3, 1) and (3, 7) has reversed.