CS 142

Lecture 31: sorting

Thanks to Marty Stepp and Stuart Reges for parts of these slides

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, 2nd, and 3rd passes:

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

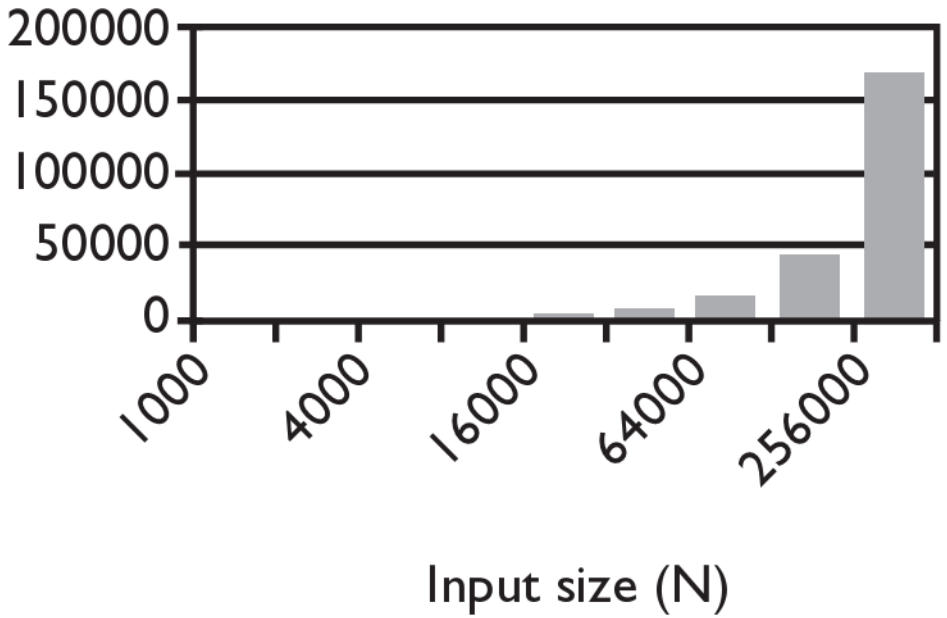
index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
value	-4	**2**	12	22	27	30	36	50	7	68	91	56	**18**	85	42	98	25

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 runtime (Fig. 13.6)

- What is the complexity class (Big-Oh) of selection sort?

N	Runtime (ms)
1000	0
2000	16
4000	47
8000	234
16000	657
32000	2562
64000	10265
128000	41141
256000	164985



Similar algorithms

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

- **bubble sort**: Make repeated passes, swapping adjacent values
  - slower than selection sort (has to do more swaps)

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

22 → 50 → 91 → 98 →

- **insertion sort**: Shift each element into a sorted sub-array
  - faster than selection sort (examines fewer values)

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

sorted sub-array (indexes 0-7) ← 7

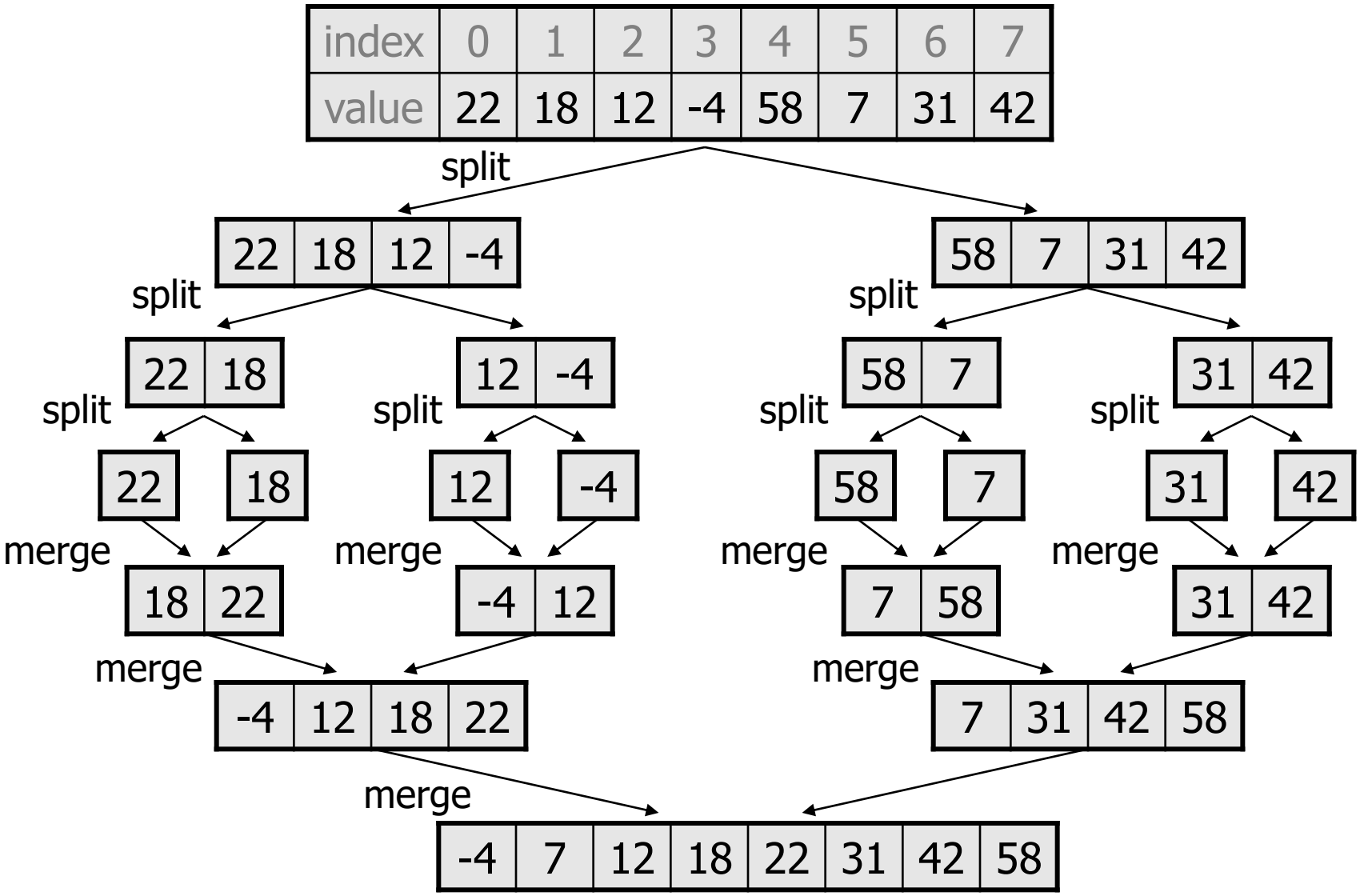
Merge sort

- **merge sort**: Repeatedly divides the data in half, sorts each half, and combines the sorted halves into a sorted whole.

  The algorithm:
  - Divide the list into two roughly equal halves.
  - Sort the left half.
  - Sort the right half.
  - Merge the two sorted halves into one sorted list.

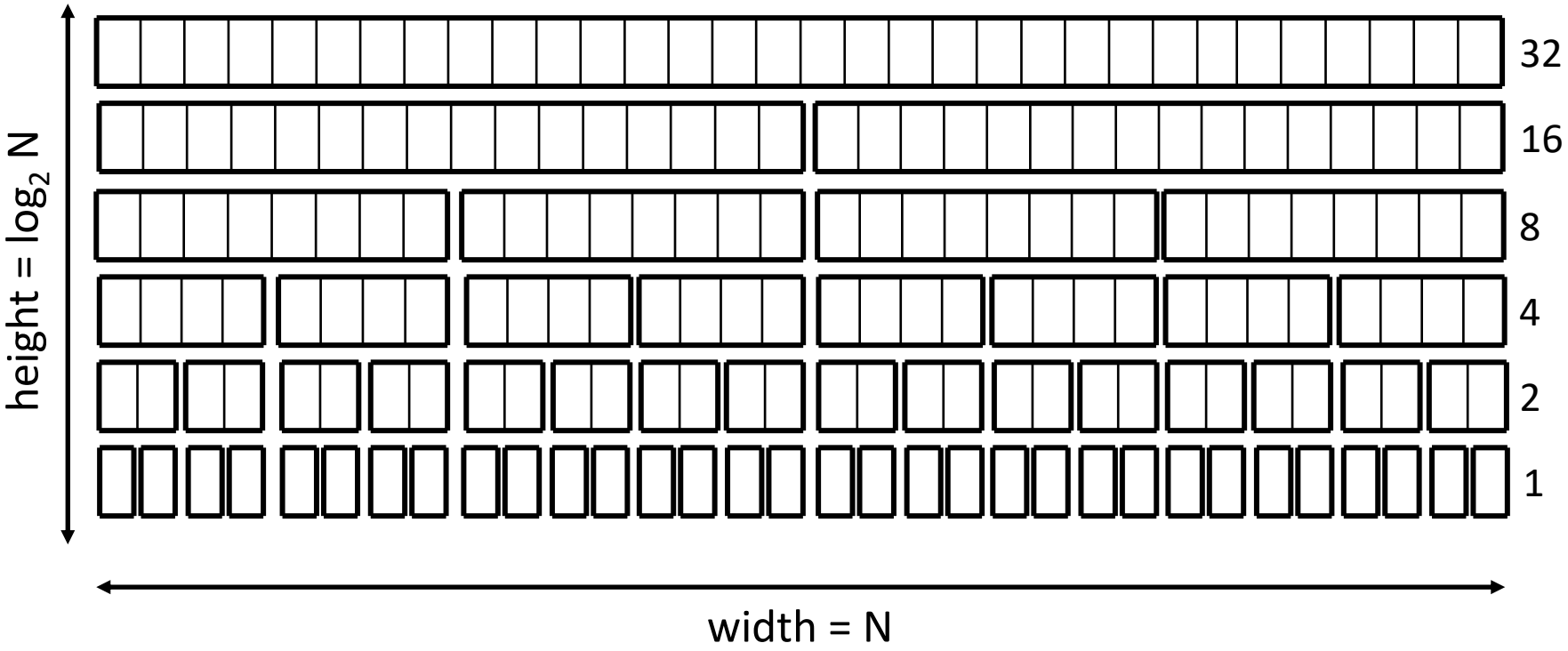
  - Often implemented recursively.
  - An example of a "divide and conquer" algorithm.
    - Invented by John von Neumann in 1945

Merge sort example

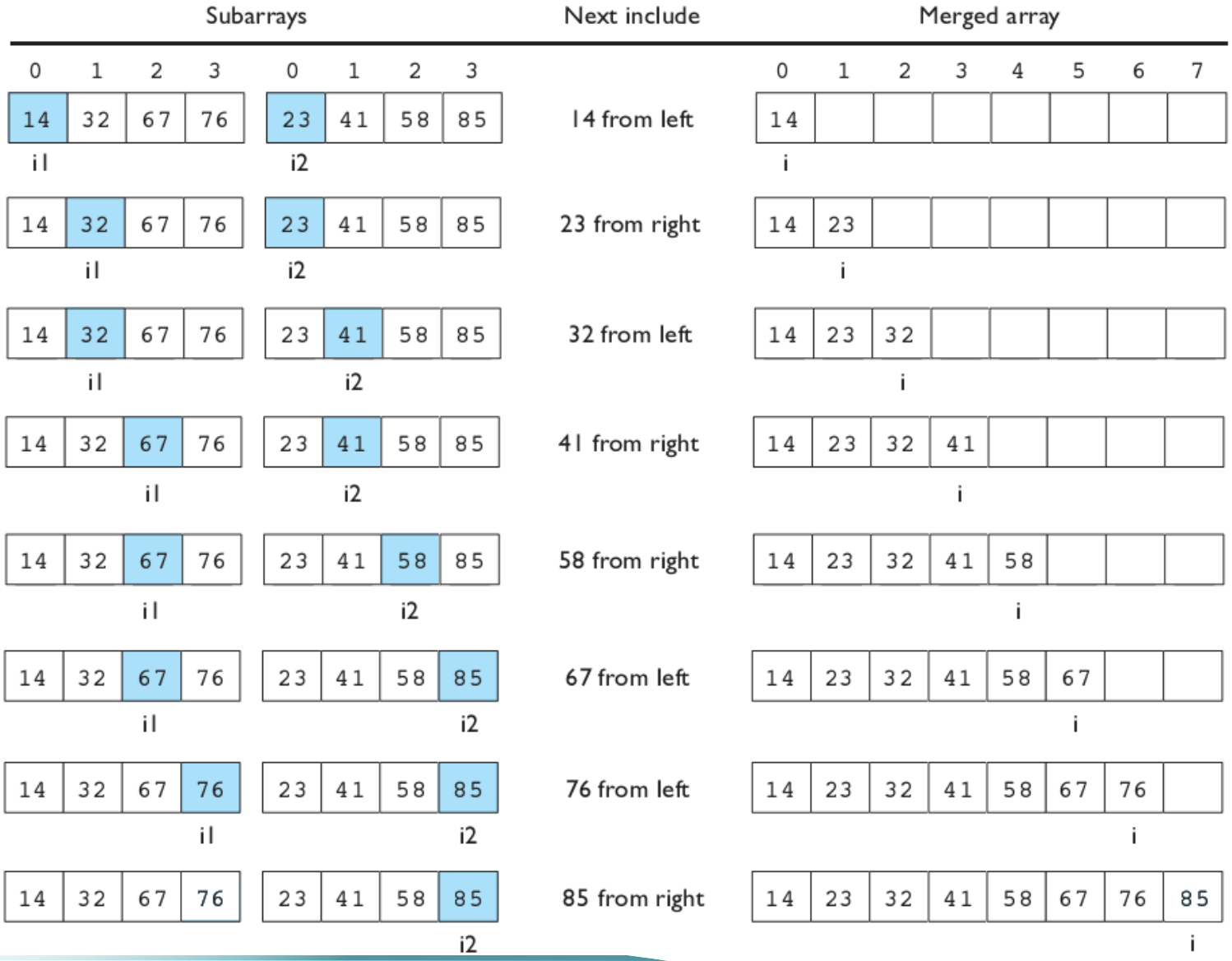


More runtime intuition

- Merge sort performs O(*N*) operations on each level. *(width)*
  - Each level splits the data in 2, so there are $\log_2 N$ levels. *(height)*
  - Product of these = $N * \log_2 N$ = O(*N* log *N*). *(area)*
  - Example: *N* = 32. Performs ~ $\log_2 32$ = 5 levels of *N* operations each:



Merging sorted halves



Solution: merge halves code

```
// Merges the left/right elements into a sorted result.
// Precondition: left/right are sorted
public static void merge(int[] result, int[] left,
                                       int[] right) {
    int i1 = 0;   // index into left array
    int i2 = 0;   // index into right array

    for (int i = 0; i < result.length; i++) {
        if (i2 >= right.length ||
            (i1 < left.length && left[i1] <= right[i2])) {
            result[i] = left[i1];    // take from left
            i1++;
        } else {
            result[i] = right[i2];   // take from right
            i2++;
        }
    }
}
```

Solution: merge sort code 2

```
// Rearranges the elements of a into sorted order using
// the merge sort algorithm (recursive).
public static void mergeSort(int[] a) {
    if (a.length >= 2) {
        // split array into two halves
        int[] left  = Arrays.copyOfRange(a, 0, a.length/2);
        int[] right = Arrays.copyOfRange(a, a.length/2, a.length);

        // sort the two halves
        mergeSort(left);
        mergeSort(right);

        // merge the sorted halves into a sorted whole
        merge(a, left, right);
    }
}
```

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.

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

...

Choosing a "pivot"

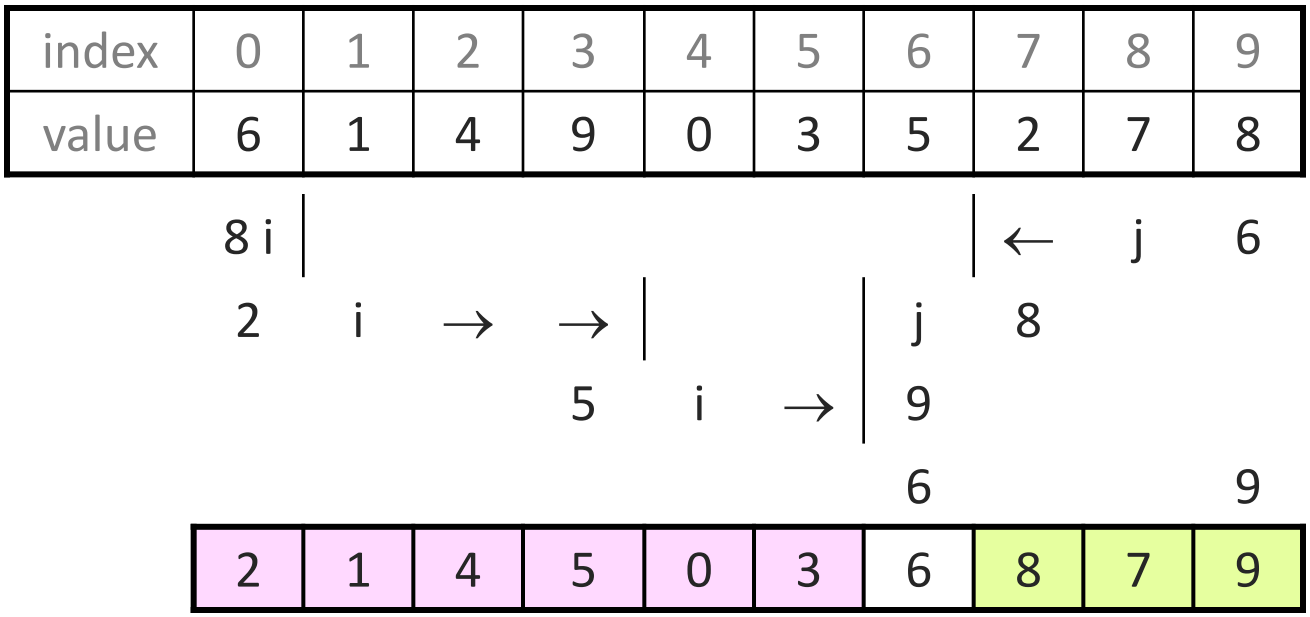
- The algorithm will work correctly no matter which element you choose as the pivot.
  - A simple implementation can just use the first element.

- 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.



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.
  - 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.

In-place sorting

- **In-place**: not needing to copy the array/list, just needing space for constant sized variables

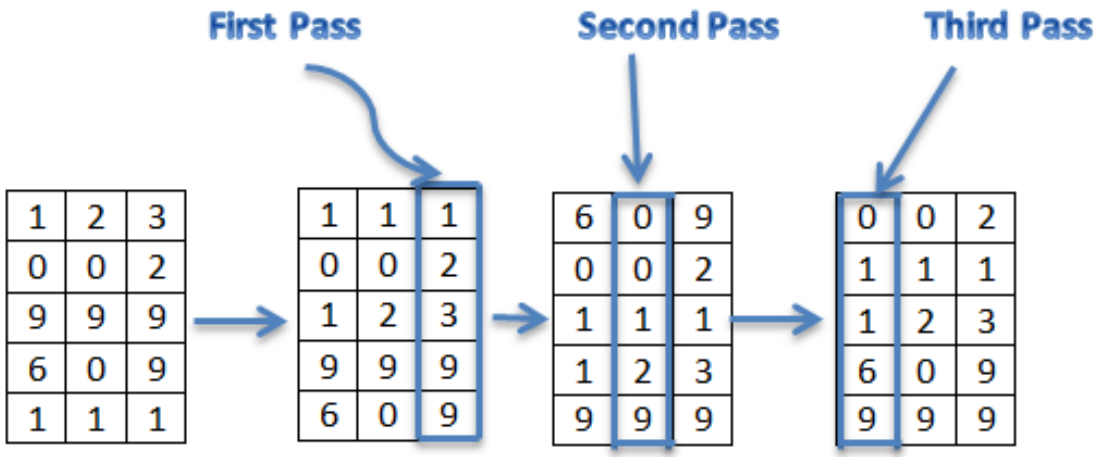
  - Which of the sorts that we have seen so far can we do in-place?
    - selection sort
    - insertion sort
    - bubble sort
    - quick sort (usually considered in-place even though variable space is needed for stack pointers to recursive calls)

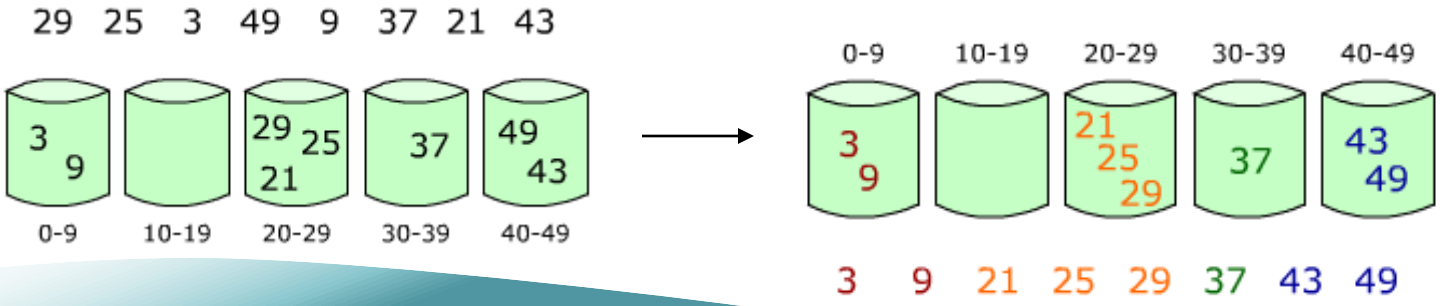
Sorting in the real world

- What sort does Java's built-in sort use?
  - a hybrid of multiple sorts
    - Insertion sort
    - Quick sort
    - Merge sort
- Most real-world sorts are hybrids as the setup cost for the more efficient sorts is really high when the amount of data is small.
- If you are asked to implement a sort in an interview, I would generally recommend going for merge sort
  - efficient, good sort that is commonly used
  - stable
  - much easier to get right in an interview than quick sort

Bucket/radix sort

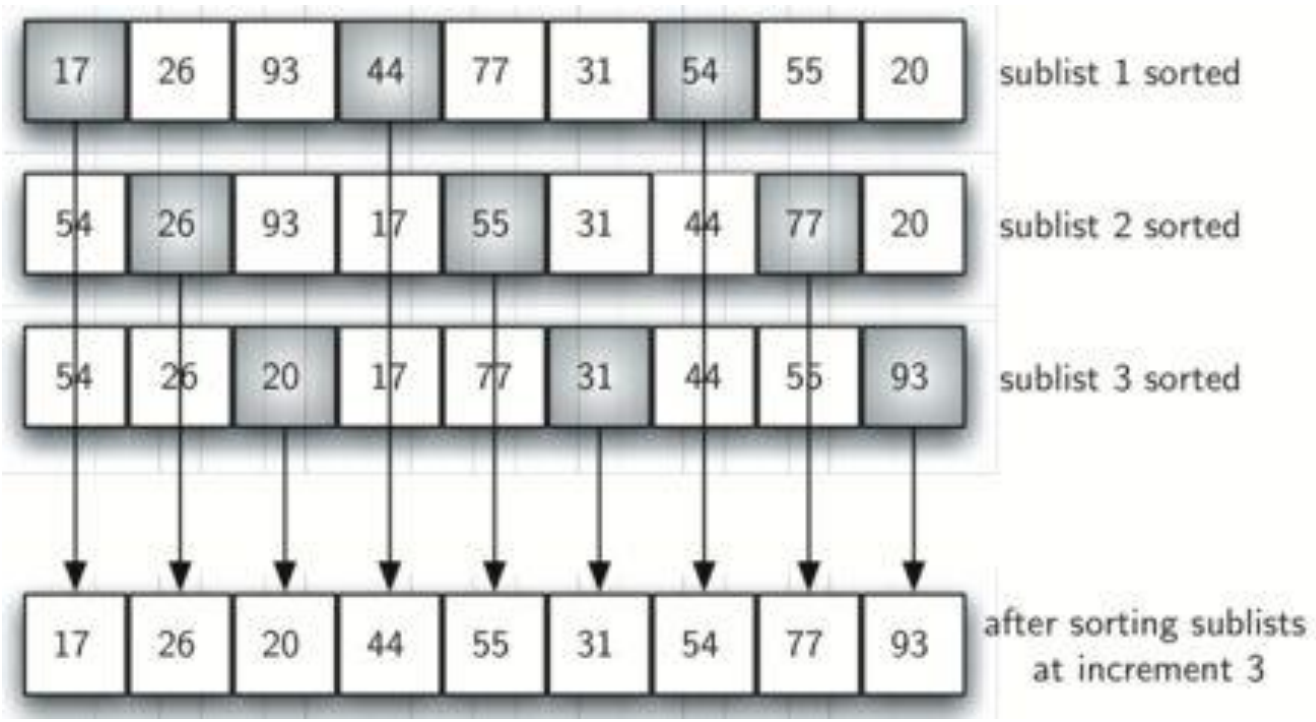
- **bucket sort**: arrange items into buckets or bins repeatedly
- **radix sort**: sort integers by 1s, then 10s, then 100s, ...
  - O(*N*) when used with data in a known fixed range (!)





Shell sort

- **shell sort**: Sort every *k* th element, repeatedly, then reduce *k*
  - not as fast as merge/quick sort, but better than selection/insertion
  - O(N $^{3/2}$) or O(N $^{4/3}$) depending on gaps used



Parallel sorts

- **parallel sorting algorithms**: modify existing algorithms to work with multiple CPUs/cores
- common example: **parallel merge sort**.
  - general algorithm idea:
    - Split array into two halves.
    - One core/CPU sorts each half.
    - Once both halves are done, a single core merges them.