CSc 110, Autumn 2017

Lecture 38: Sorting

Adapted from slides by Marty Stepp and Stuart Reges

Sorting

- **sorting**: Rearranging the values in a list 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, ...

  - *comparison-based sorting* : determining order by comparing pairs of elements:
    - <, >, …

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 list:

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

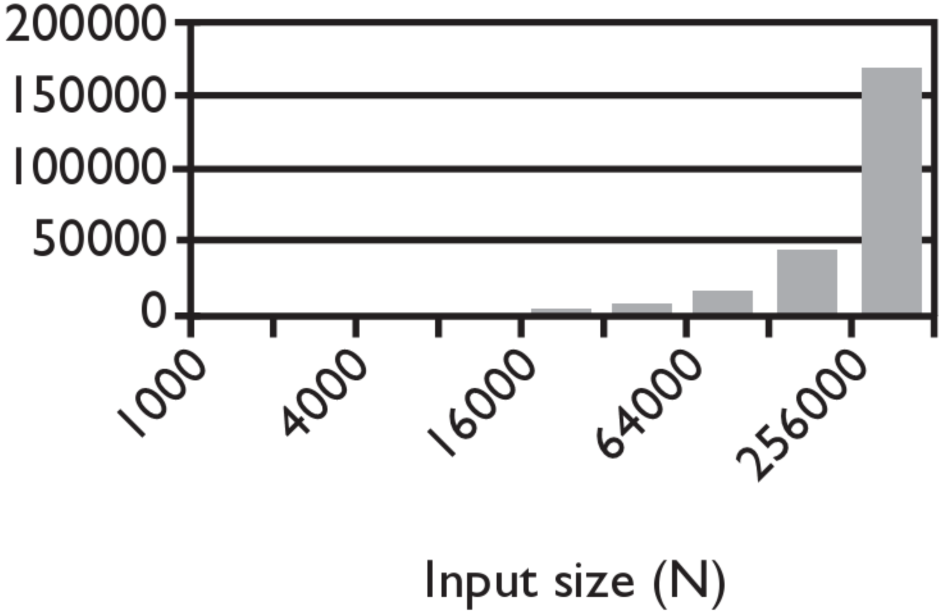
Selection sort code

```
# Rearranges the elements of a into sorted order using
# the selection sort algorithm.
def selection_sort(a):
    for i in range(0, len(a) - 1):
        # find index of smallest remaining value
        min = i
        for j in range(i + 1, len(a)):
            if (a[j] < a[min]):
                min = j
        # swap smallest value its proper place, a[i]
        swap(a, i, min)
```

Selection sort runtime (Fig. 13.6)

- How many comparisons does selection sort have to do?

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



Activity

- Initial list:

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
value	2	8	-2	4	7	0	-6	50	70	-8	1	6	21	5	42	9	-5

- 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	**-8**	8	-2	4	7	0	-6	50	70	**2**	1	6	21	5	42	9	-5

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
value	-8	**-6**	-2	4	7	0	**8**	50	70	2	1	6	21	5	42	9	-5

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
value	-8	-6	**-5**	4	7	0	8	50	70	2	1	6	21	5	42	9	**-2**

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-list
  - 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-list (indexes 0-7)

← 7

Merge sort

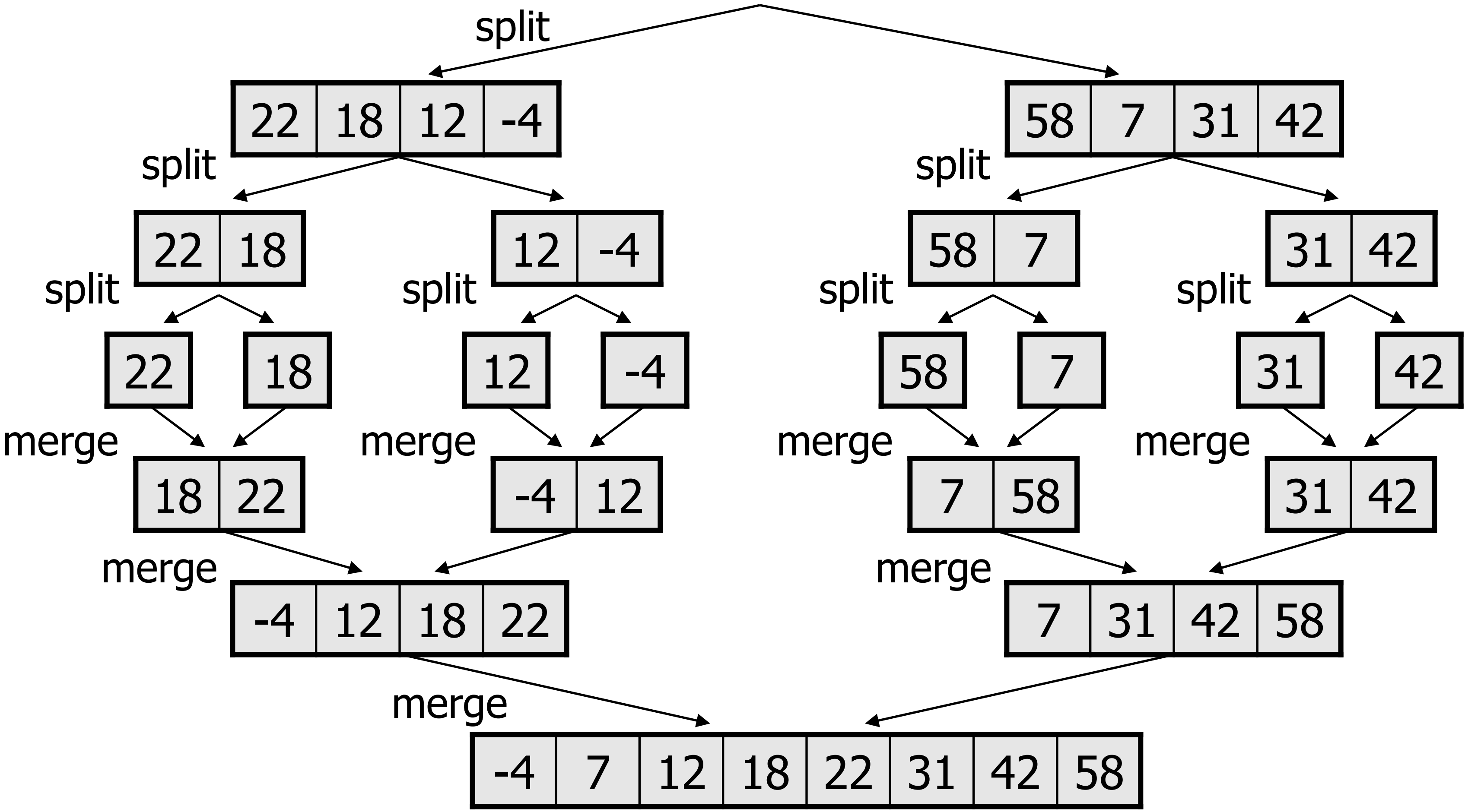
- **merge sort**: Repeatedly divides the data in half, sorts each half, and combines the sorted halves into a sorted whole.

  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

index	0	1	2	3	4	5	6	7
value	22	18	12	-4	58	7	31	42



split

22 | 18 | 12 | -4

58 | 7 | 31 | 42

split

22 | 18

12 | -4

58 | 7

31 | 42

split

22

18

12

-4

58

7

31

42

merge

18 | 22

-4 | 12

7 | 58

31 | 42

merge

-4 | 12 | 18 | 22

7 | 31 | 42 | 58

merge

-4 | 7 | 12 | 18 | 22 | 31 | 42 | 58

Merge halves code

```
# Merges the left/right elements into a sorted result.
# Precondition: left/right are sorted
def merge(result, left, right):
    i1 = 0   # index into left list
    i2 = 0   # index into right list

    for i in range(0, len(result)):
        if i2 >= len(right) or (i1 < len(left) and left[i1] <= right[i2]):
            result[i] = left[i1]    # take from left
            i1 += 1
        else:
            result[i] = right[i2]   # take from right
            i2 += 1
```

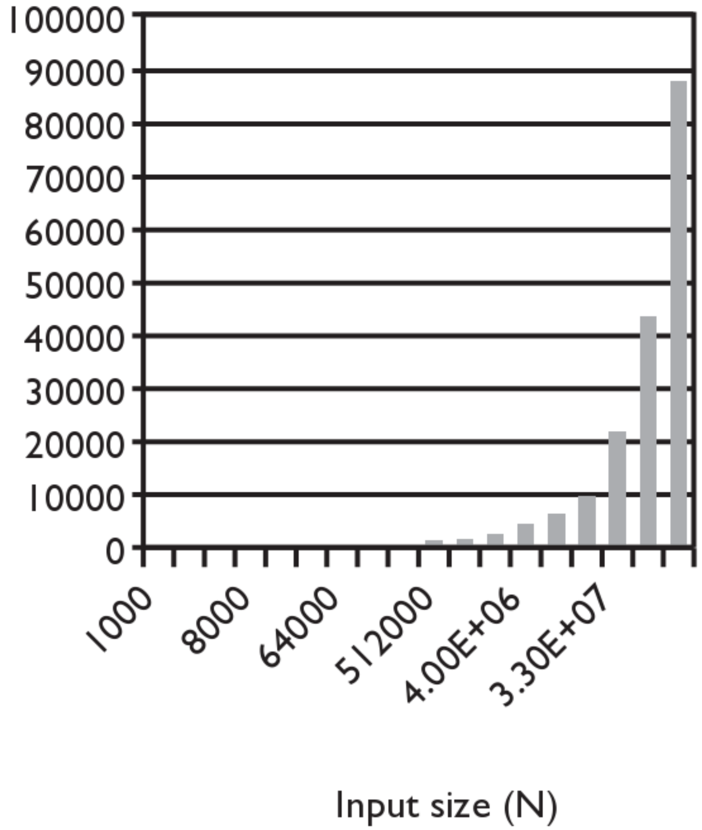
Merge sort code

```
# Rearranges the elements of a into sorted order using
# the merge sort algorithm.
def merge_sort(a):
    if len(a) >= 2:
        # split list into two halves
        left  = a[0, len(a)//2]
        right = a[len(a)//2, len(a)]

        # sort the two halves
        merge_sort(left)
        merge_sort(right)

        # merge the sorted halves into a sorted whole
        merge(a, left, right)
```

Merge sort runtime

- How many comparisons does merge sort have to do?

N	Runtime (ms)
1000	0
2000	0
4000	0
8000	0
16000	0
32000	15
64000	16
128000	47
256000	125
512000	250
1e6	532
2e6	1078
4e6	2265
8e6	4781
1.6e7	9828
3.3e7	20422
6.5e7	42406
1.3e8	88344



Activity

merge sort the following list:

index	0	1	2	3	4	5	6	7
value	2	11	6	4	-8	7	3	42

Sorting algorithms

- **bogo sort**: shuffle and pray
- **bubble sort**: swap adjacent pairs that are out of order
- **selection sort**: look for the smallest element, move to front
- **insertion sort**: build an increasingly large sorted front portion
- **merge sort**: recursively divide the list in half and sort it
- **heap sort**: place the values into a sorted tree structure
- **quick sort**: recursively partition list based on a middle value

other specialized sorting algorithms:

- **bucket sort**: cluster elements into smaller groups, sort them
- **radix sort**: sort integers by last digit, then 2nd to last, then ...
- ...