Data Structures & Algorithms

Lecture 10: Searching and Sorting

Searching

Sequential search

- **sequential search**: Locates a target value in a list by examining each element from start to finish. Used in `index`.
  - How many elements will it need to examine?
  - Example: Searching the list below for the value **42**:

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
value	-4	2	7	10	15	20	22	25	30	36	42	50	56	68	85	92	103

i

Sequential search

- How many elements will be checked?

```
def index(value):
    for i in range(0, size):
        if my_list[i] == value:
            return i
    return -1   # not found
```

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
value	-4	2	7	10	15	20	22	25	30	36	42	50	56	68	85	92	103

- On average how many elements will be checked?

Binary search

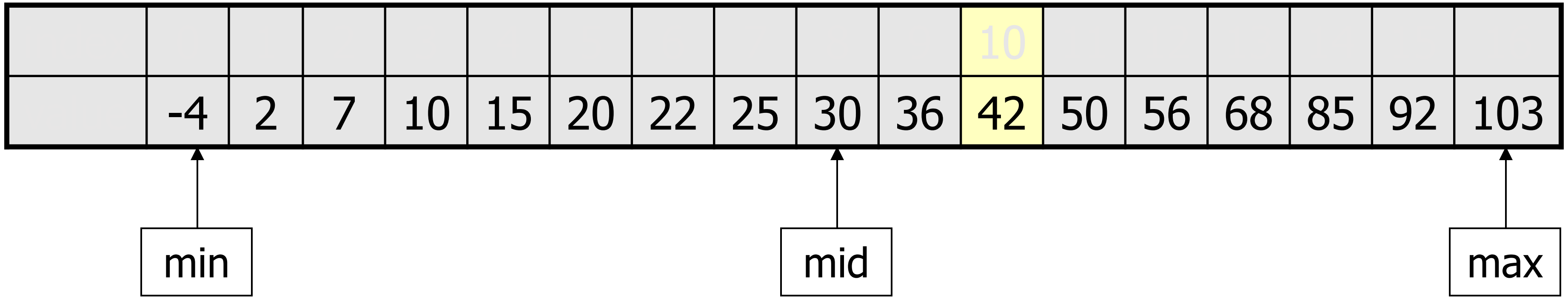
- **binary search**: Locates a target value in a *sorted* list by successively eliminating half of the list from consideration.
  - How many elements will it need to examine?
  - Example: Searching the list below for the value **42**:

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
value	-4	2	7	10	15	20	22	25	30	36	42	50	56	68	85	92	103

min mid max

Binary search

- **binary search**: Locates a target value in a *sorted* array/list by successively eliminating half of the array from consideration.
  - How many elements will it need to examine?
  - Example: Searching the array below for the value **42**:



Binary search runtime

- For any list of size N, it eliminates ½ until 1 element remains.

  N, N/2, N/4, N/8, ..., 4, 2, 1

  - How many divisions does it take?

- Think of it from the other direction:
  - How many times do I have to multiply by 2 to reach N?

    1, 2, 4, 8, ..., N/4, N/2, N
  - Call this number of multiplications "x".

    $2^x = N$

    $\mathbf{x = \log_2 N}$

- Binary search looks at a **logarithmic** number of elements

binary_search

Write the following two functions:

```
# searches an entire sorted list for a given value
# returns the index the value should be inserted at to maintain sorted order
# Precondition: list is sorted
binary_search(list, value)

# searches given portion of a sorted list for a given value
# examines min_index (inclusive) through max_index (exclusive)
# returns the index of the value or -(index it should be inserted at + 1)
# Precondition: list is sorted
binary_search(list, value, min_index, max_index)
```

Using binary_search

```
# index 0   1   2   3    4    5    6    7    8    9   10   11   12   13   14   15
a   =  [-4, 2, 7, 9, 15, 19, 25, 28, 30, 36, 42, 50, 56, 68, 85, 92]

index1 = binary_search(a, 42)
index2 = binary_search(a, 21)
index3 = binary_search(a, 17, 0, 16)
index2 = binary_search(a, 42, 0, 10)
```

- `binary_search` returns the index of the number

or

- (index where the value should be inserted + 1)

Binary search code

```
# Returns the index of an occurrence of target in a,
# or a negative number if the target is not found.
# Precondition: elements of a are in sorted order
def binary_search(a, target, start, stop):
    min = start
    max = stop - 1

    while min <= max:
        mid = (min + max) // 2
        if a[mid] < target:
            min = mid + 1
        elif a[mid] > target:
            max = mid - 1
        else:
            return mid   # target found

    return -(min + 1)    # target not found
```

Sorting

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:
    - <, >, …

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 ...
- ...

Random

Bogo sort

- **bogo sort**: Orders a list of values by repetitively shuffling them and checking if they are sorted.
  - name comes from the word "bogus"

  The algorithm:
  - Scan the list, seeing if it is sorted. If so, stop.
  - Else, shuffle the values in the list and repeat.

- This sorting algorithm (obviously) has terrible performance!

Bogo sort code

```
# Places the elements of a into sorted order.
def bogo_sort(a):
    while (not is_sorted(a)):
        shuffle(a)

# Returns true if a's elements
#are in sorted order.
def is_sorted(a):
    for i in range(0, len(a) - 1):
        if (a[i] > a[i + 1]):
            return False
    return True
```

```
# Swaps a[i] with a[j].
def swap(a, i, j):
    if (i != j):
        temp = a[i]
        a[i] = a[j]
        a[j] = temp

# Shuffles a list by randomly swapping each
# element with an element ahead of it in the list.
def shuffle(a):
    for i in range(0, len(a) - 1):
        # pick a random index in [i+1, a.length-1]
        range = len(a) - 1 - (i + 1) + 1
        j = (random() * range + (i + 1))
        swap(a, i, j)
```

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

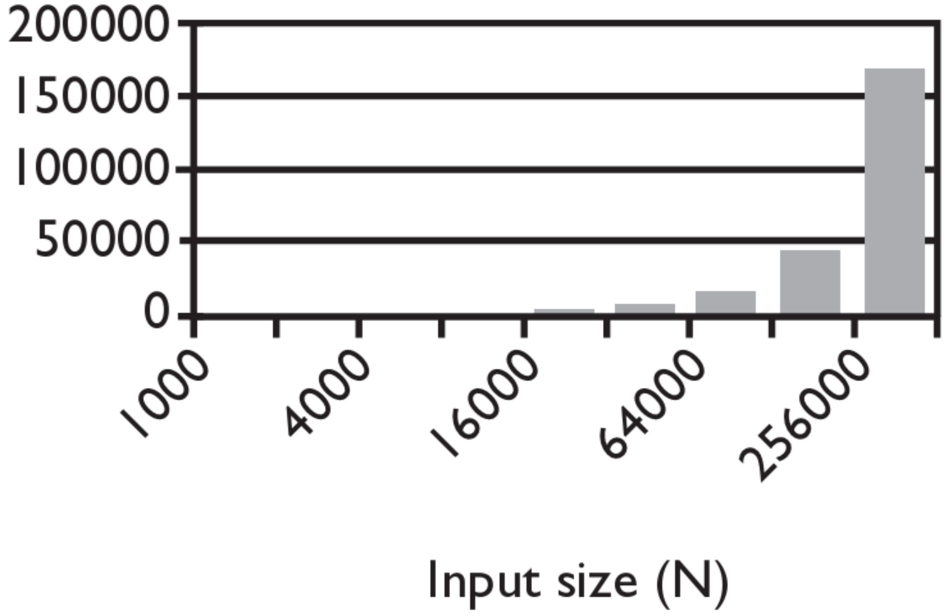
Selection sort code

```
# 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

- 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



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

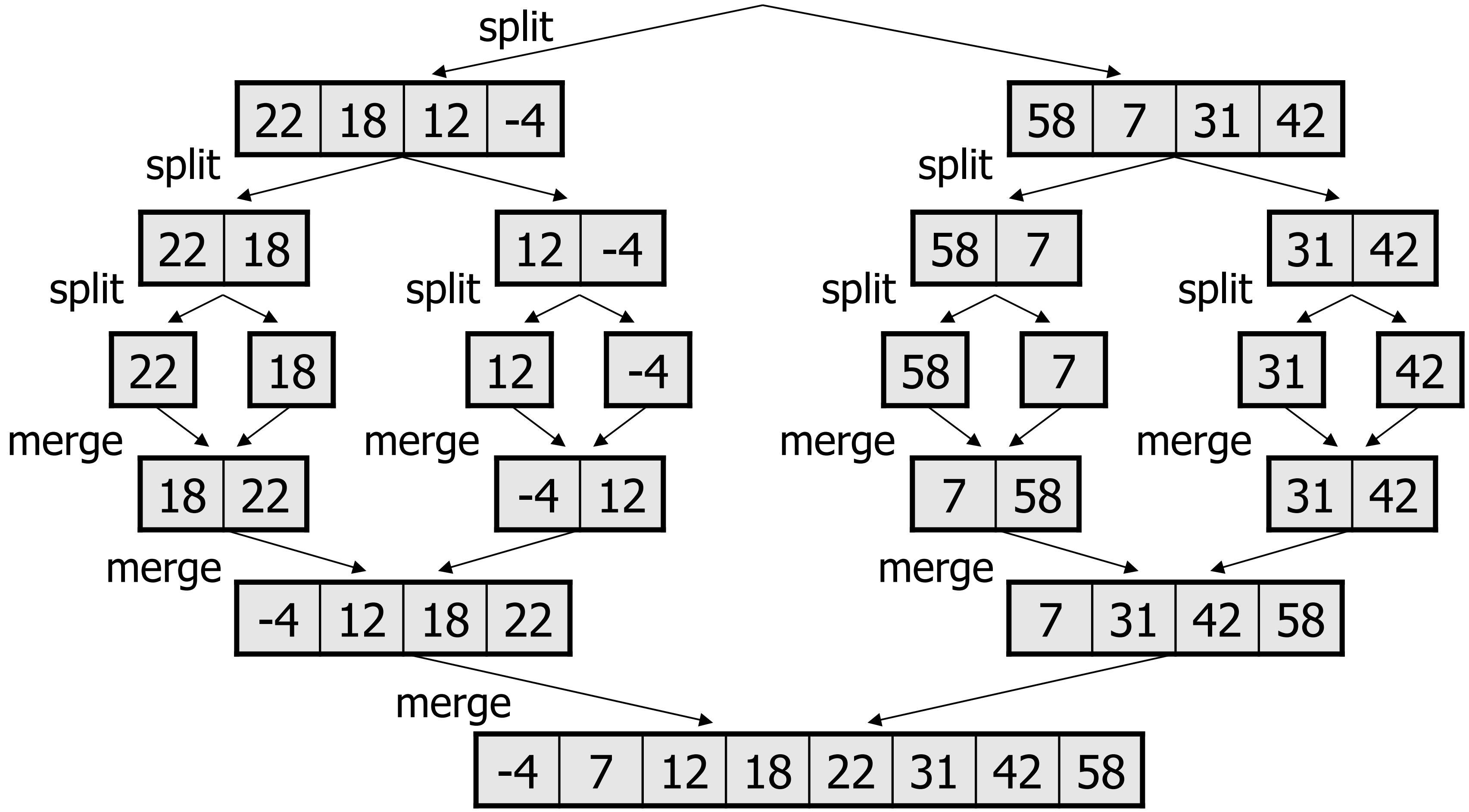
- **merge sort**: Repeatedly divides the data in half, sorts each half, and combines the sorted halves into a sorted whole.

  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



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
```

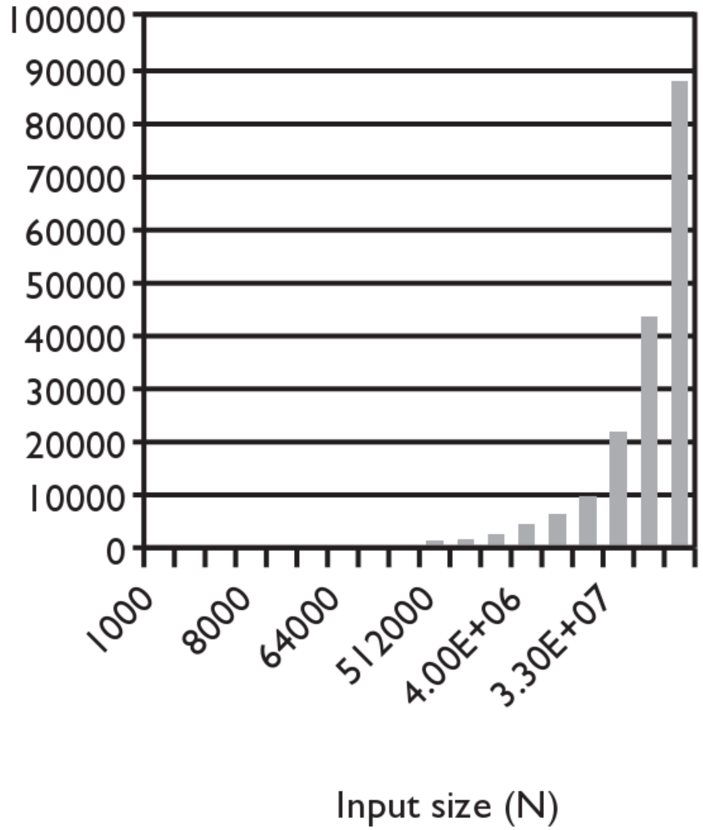
Merge sort code

```
# 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



Exercise 10

- Question 1

Merge sort the following list:

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

- Question 2

Write a Python console application that implements various search algorithms.

- Question 3

Write a Python console application that implements various sort algorithms.