Introduction to Problem Solving in Python

COSI 10A

Class objectives

- File Processing (6.3)
  - File output
- Searching (10.3)
- Sorting (10.3)

Output to file

- Open a file in write or append mode
  - 'w' - write mode – replaces everything in the file
  - 'a' – append mode – adds to the bottom of the file preserving what is already in it

```
name = open("filename", "w")      # write
name = open("filename", "a")      # append
```

- Once you have opened the file for writing, you can send output to it
- If no such file already exists, the program will create it
- If such file exists, the computer will overwrite the current version

Output to file

```
with open("hello.txt", "w") as file:
      print("Hello, world!", file= file)
      print("", file= file)
      print("This program produces four", file= file)
      print("lines of output in a file.", file= file)
```

- Remember to wrap any file-writing code in a with statement, or else make sure to explicitly call close on the file

Prompting for a File

- Prompting for a file name involves reading a string with the `input` function

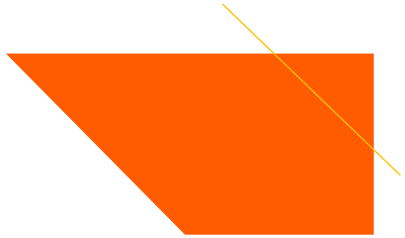
```
filename = input("file to open:")
```

- … but if the user types a name of a file that does not exist the program crashes `FileNotFoundError`
  - Possible ways to avoid this error:
    - `if` statement ?
    - `try/except` statement ?
    - Ask whether the given file exists before trying to open it

Prompting for a File

- Need to use a library called `os.path`
- Call the function `isfile`, which returns True if a given file exists and False if not

```
import os.path

def prompt_for_file(message):
    filename = input(message)
    while not os.path.isfile(filename):
        print("File not found. Try again")
        filename = input(message)
    return filename

def main():
    filename = prompt_for_file("File to open? ")
    with open(filename) as file:
        filetext = file.read()
        print(filetext)

main()
```

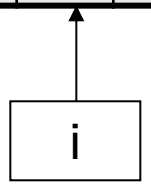


Searching

Sequential search

- **Sequential search**: Locates a target value in a list by examining each element from start to finish. Used in index.
- Example: Searching the list below for the value 42:
- index(value, my_list)

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



Sequential search

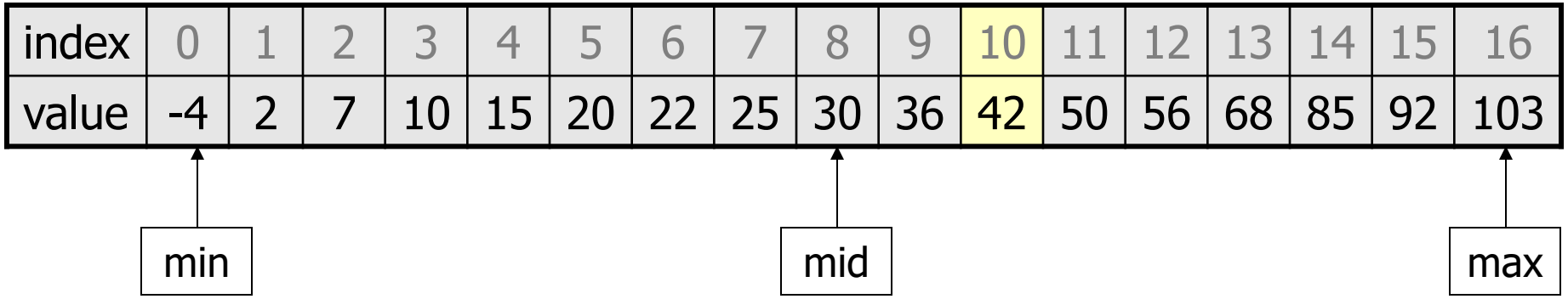
```
def index(value, my_list):
    for i in range(0, len(my_list)):
        if my_list[i] == value:
            return i
    return -1   # not found
```

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?

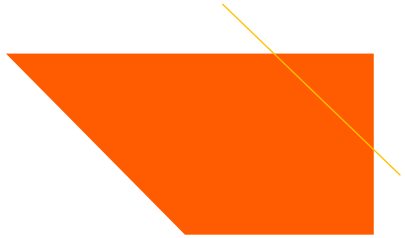
Binary search

- **Binary search**: Locates a target value in a sorted list by successively eliminating half of the list from consideration
- Example: Searching the list below for the value 42:



Binary search

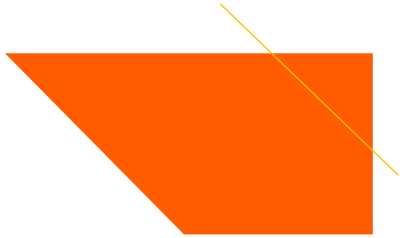
```
# Returns the index of an occurrence of target in a,
# or a negative number if the target is not found.
# Precondition: elements of a are in sorted order
def binary_search(a, target):
    min = 0
    max = len(a)-1

    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 -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
  - 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 ...
- ...

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

- 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

```
# Rearranges the elements of a list 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)
```