What we will cover today

- Algorithms and their analysis
- Image
- Command Line Arguments
- Debugger
An algorithm specifies a series of steps that perform a particular computation or task.

An algorithm may receive an input, perform its instructions, and may produce an output.

Algorithms are essential to the way computers process data.
Algorithm – a simple example

- Finding the max value in a given list:
  1. Set a new variable `max` to 0
  2. For each number in the list if it is larger than `max` set `max` to the current value
  3. At the end of the run `max` will have the highest value

```python
def find_max(lst):
    max_num = 0
    for x in lst:
        if max_num < x:
            max_num = x
    return max_num
```
 Algorithms

There are two things that must be considered when we design an algorithm:

1. Running time
2. Memory usage
Runtime Analysis

- It is like “counting” the number of operations needed to be done, given an input of size $n$.
  - How many iterations or recursive calls
  - How many operations are done in each iteration or recursive call
\( f(n) = O(g(n)) \) if and only if there exists a positive real number \( c \) and a real number \( n_0 \) such that

\[
f(n) \leq c \cdot g(n) \text{ for all } n \geq n_0
\]

basically, it tells you how fast a function grows
O notation

- For typical programs, $g(n)$ is a function such as $\log(n)$, $n$, $n\log(n)$, $n^2$, or $n^3$. 
Constant time:

- Arithmetic operations (+, -, *, /, %)
- Comparison operators (<, >, ==, !=)
- Variable declaration
- Assignment statement
- Invoking a method or function
Let \( f(x) = 6x^4 - 2x^3 + 5 \)

Of each of the three terms of \( f(x) \), \( 6x^4 \), \( -2x^3 \) and \( 5 \), \( 6x^4 \) has the highest growth rate as \( x \) becomes very large.

So we can say that \( g(x) = 6x^4 \) is “big O” of \( f(x) \):

\[ f(x) = O(x^4) \]
Let \( f(x) = 6x^4 - 2x^3 + 5 \) and \( g(x) = x^4 \)

We will show that \( f(x) \leq c \cdot g(x) \) for all \( x \geq x_0 \) and a positive integer \( c \)

Let \( x_0 = 1 \) and \( c = 13 \)

\[
f(x) = 6x^4 - 2x^3 + 5 \leq 6x^4 + 2x^3 + 5 \\
    \leq 6x^4 + 2x^4 + 5x^4 \\
    \leq 13x^4
\]

\( f(x) = O(x^4) \)
Proof of Correctness

- Two main conditions:
  - The algorithm is complete/correct: the post-condition is respected on all possible inputs satisfying the pre-condition
    - Precondition: a predicate $I$ on the input data
    - Postcondition: a predicate $O$ on the output data
    - Correctness: proving $I \Rightarrow O$
  - The algorithm terminates: for all possible input, the algorithm exits
Back to the example:

- Define \( n = \text{len(lst)} \)
- There are \( n \) iterations
- In each we use comparison and assignment operations — this is constant:

\[
\Rightarrow \text{The runtime is } O(n)
\]
A sorting algorithm is an algorithm that puts elements of in a certain order:
- Lexicographic
- Numeric
- Chronologic

Sorting is important since many search algorithms are much more efficient (lower running time complexity) when running on a sorted input.

We will talk about lists during the presentation, but it is valid to any type of sequence that keeps the order of its elements.
Example

- Find the youngest person

  - **Unsorted**: go over n rows: worst case *takes n steps*

- Find the person who is 10 years old

  - **Unsorted**: go over n rows: worst case *takes n steps*

<table>
<thead>
<tr>
<th>Name</th>
<th>Age</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bart</td>
<td>10</td>
</tr>
<tr>
<td>Homer</td>
<td>37</td>
</tr>
<tr>
<td>Lisa</td>
<td>8</td>
</tr>
<tr>
<td>Marge</td>
<td>35</td>
</tr>
<tr>
<td>Meggie</td>
<td>1</td>
</tr>
</tbody>
</table>
Example

- Find the youngest person
- Sorted (by age): get the youngest person in the table: \textit{takes 1 step}
- Find the person who is 10 years old
- How long would it take on a sorted list?

<table>
<thead>
<tr>
<th>Name</th>
<th>Age</th>
</tr>
</thead>
<tbody>
<tr>
<td>Meggie</td>
<td>1</td>
</tr>
<tr>
<td>Lisa</td>
<td>8</td>
</tr>
<tr>
<td>Bart</td>
<td>10</td>
</tr>
<tr>
<td>Marge</td>
<td>35</td>
</tr>
<tr>
<td>Homer</td>
<td>37</td>
</tr>
</tbody>
</table>
Binary Search

- Given a **sorted** list and a key to be found:
  - Get the middle element
  - If(key == middle) we are done.
  - If(key < middle) do binary search on the left half of the list
  - If(key > middle) do binary search on the right half of the list
Binary Search - example

- A list: [1,1,2,3,5,8,13,21,34,55]
- We want to find the index of 3 in the list
  1. middle = 8
  2. 3 < 8, therefore we will look for it in the left half: [1,1,2,3,5]
  3. Search again on the updated list
Binary Search

- Find the index of 3:

```
1 1 2 3 5 8 13 21 34 55
```

- Middle:

```
1 1 2 3 5 8 13 21 34 55
```

- Middle:

```
1 1 2 3 5 8 13 21 34 55
```

- Middle:

```
1 1 2 3 5 8 13 21 34 55
```
def binary_search_iterative(lst, key):
    left = 0
    right = len(lst)
    idx = -1
    while (left <= right):
        middle_idx = (left + right) // 2
        middle = lst[middle_idx]
        if (middle == key):
            idx = middle_idx
            break
        elif (middle > key):
            right = middle_idx - 1
        else:
            left = middle_idx + 1
    return idx
In each iteration the search is done on a list that is half the size of the previous one.

Assume the list is of size $n$

After first iteration the list is of size $n/2$

After the second iteration the list is of size $n/2 \cdot (1/2) = n/2^2$

After $k$ iterations the list is of size $n/2^k$

The algorithm is done when the list is of size 1, therefore:

$$\frac{n}{2^k} = 1$$

Take log:

$$k = \log(n)$$
Sorting Motivation:

- In many applications we would like to get information on a given input, for example to get the grades of a student.
- Given list of data of size $n$:
  - Searching on an unsorted list requires $n$ steps.
  - Searching on a sorted list requires $\log(n)$ steps when we use binary search.
Bubble Sort

- The idea: bubble up the largest element
- **Iteration 1**: Compare each element with its neighbor to the right from index 0 to index n-1
  - If they are out of order, swap them
  - At the end of this iteration, the largest element is at the end of the list
  - If there were no swaps after going over all elements, we are done
Bubble Sort

- **Iteration 2**: Compare each element with its neighbor to the right from index 0 to index $n-2$

... 

- Continue until there are no more elements to swap
Bubble Sort - example

Iteration 1: 7 2 8 5 4
Iteration 2: 2 7 5 4 8
Iteration 3: 2 5 4 7 8
Iteration 4: 2 4 5 7 8

(done)
def bubble_sort(lst):
    for i in range(len(lst)):
        swap = False
        for j in range(len(lst)-i-1):
            if(lst[j] > lst[j+1]):
                lst[j], lst[j+1] = lst[j+1], lst[j]
                swap = True
        if(not swap):
            break
    return lst
For a list of size $n$, the outer loop does $n$ iterations.

In each iteration $i$ of the outer loop, the inner loop does $n-i$ iterations:

$$(n-1) + (n-2) + \ldots + 1 = \frac{n(n-1)}{2}$$

Note that each operation within the inner loop (comparing and swapping) takes constant time $c$.

Number of steps is a polynomial of $2^{nd}$ degree: $O(n^2)$
What would be the number of steps in the best case? (i.e., the input list is sorted)

- Start with the first iteration, going over all the list
- The list is sorted so there are no swaps

$n$ steps are required
Loop invariant is a property that holds before (and after) each iteration

The invariant is:

At iteration \( i \), the sub-array \( A[n-i:] \) is sorted and any element in \( A[n-i:] \) is greater or equal to any element in \( A[:n-i] \)
Bubble Sort – Proof of Correctness

1. **Initializing:** \( i = 0 \), the invariant holds (trivial for empty list)

2. **Proof by induction:**
   
   Assume \( A[n-i-1:] \) is sorted.

   Iteration \( i \) inserts at position \( n-i \) the largest of the remaining unsorted elements of \( A[:i+1] \), as computed by the inner loop.

   \( A[:n-i] \) contains only elements smaller than \( A[n-i:] \), and \( A[n-i] \) is smaller than any element in \( A[n-i:] \), then \( A[n-i:] \) is sorted and the invariant is preserved
**Termination:** at the last iteration, A[1:] is sorted, and all elements in A[1:] are larger than the elements in A[0]. Therefore the list is sorted.
Radix Sort

- Sorts integers by grouping keys by the individual digits which share the same significant position and value

- Assumption: the input has \( d \) digits ranging from 0 to \( k \):
  
  For each position there is a finite number of possible digits, depends on the base of the number (for decimal representation there are 10)
Radix Sort (assume decimal representation)

- Divide all in integers into 10 groups according the least significant digit
- For two numbers with the same digit keep the original order
- Repeat the above steps with the next least significant digit
Radix Sort

- **Input list:** [493, 812, 715, 710, 195, 437, 582, 340, 385]

Divide by the least significant digit:

<table>
<thead>
<tr>
<th>digit</th>
<th>sublist</th>
<th>digit</th>
<th>sublist</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>710,340</td>
<td>5</td>
<td>715,195,385</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>812,582</td>
<td>7</td>
<td>437</td>
</tr>
<tr>
<td>3</td>
<td>493</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>9</td>
<td></td>
</tr>
</tbody>
</table>

[710, 340, 812, 582, 493, 715, 195, 385, 437]
Radix Sort

- **Current list:** [710, 340, 812, 582, 493, 715, 195, 385, 437]

Divide by the second least significant digit:

<table>
<thead>
<tr>
<th>digit</th>
<th>sublist</th>
<th>digit</th>
<th>sublist</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>710, 812, 715</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>437</td>
<td>8</td>
<td>582, 385</td>
</tr>
<tr>
<td>4</td>
<td>340</td>
<td>9</td>
<td>493, 195</td>
</tr>
</tbody>
</table>

[710, 812, 715, 437, 340, 582, 385, 493, 195]
Radix Sort

- Current list: [710, 812, 715, 437, 340, 582, 385, 493, 195]

Divide by the third least significant (and last) digit:

<table>
<thead>
<tr>
<th>digit</th>
<th>sublist</th>
<th>digit</th>
<th>sublist</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>5</td>
<td>582</td>
</tr>
<tr>
<td>1</td>
<td>195</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>7</td>
<td>710,715</td>
</tr>
<tr>
<td>3</td>
<td>340,385</td>
<td>8</td>
<td>812</td>
</tr>
<tr>
<td>4</td>
<td>437,493</td>
<td>9</td>
<td></td>
</tr>
</tbody>
</table>

[195, 340, 385, 437, 493, 582, 710, 715, 812]

and we are done!
def radixsort(lst, radix=10):
    max_val = max(lst)
    power = 0
    # getting the number of digits of the largest number
    while radix**power < max_val:
        power += 1
    for p in range(power):
        # splitting into buckets according to the current digit order
        factor = radix**p
        buckets = [list() for d in range(radix)]
        for val in lst:
            tmp = val / factor
            buckets[int(tmp % radix)].append(val)
        # appending the buckets to one list according the last sorting
        lst = []
        for b in range(radix):
            buck = buckets[b]
            for val in buck:
                lst.append(val)
    return lst
Radix Sort Runtime Analysis

- Let's note:
  - $n$ the number of elements in the input
  - $k$ the number of digits in the largest number
  - $b$ the base of the number
- The outer loop would iterate $k$ times
- In each iteration, we go over $n$ elements and rebuild a list from $b$ buckets

   It would take about $k \times (n+b)$ steps

   For $b$, $k$ small enough it is linear runtime
A digital image is typically encoded as a \( k \)-by-\( l \) rectangle, or matrix, \( M \), of either grey-level or color values.

The matrix can be presented as list of lists.

For \( k \)-by-\( l \) matrix the we will have a list of \( k \) lists, each of length \( l \).
Matrix - example

- Lets look at this matrix:
- Present it as a list of lists:
  \[ M = [[1,2,3], [4,5,6], [7,8,9]] \]
- When we want to get the \( i,j \) element, we would like to get the value of the element at the \( i \)th row and at the \( j \)th column:
  \[ x = M[i][j] \]
Each element of the image matrix is called a *pixel*, shorthand for picture element.

For grey level images, each pixel is a nonnegative real number (in the range 0-255), representing the light intensity at the pixel.
For standard (RGB) color images, each pixel is a triplet of values, representing the Red, Green, and Blue components (in the range 0-255) of the light intensity at the pixel.
We will use the module Pillow (which was forked from PIL).

- It is installed at CS environment, and can be installed at home.
- Adds image processing capabilities to the Python interpreter.
- Supports many file formats, and provides powerful image processing and graphics capabilities.
>>> from PIL import Image
>>> img = Image.open("Eiffel.jpg")
>>> img.show()

This opens a new, graphical window.
Image modes

- ‘1’ - Binary images (black and white or 0 and 255 only)
- ‘L’ - Grayscale images
- ‘RGB’ - colored images with 3 brandes
- ‘RGBA’ - colored images with alpha channel (for transparency)
(some) Image attributes

```python
>>> img.format
'JPEG'

>>> img.size # (width, height) of pixels
(1024, 768)

>>> img.mode
'RGB'
```
getpixel

- getpixel method gets a tuple (column, row)
- Top left pixel coordinates are (0,0)

#RGB image:

```python
>>> img.getpixel((0,0))
(206, 237, 255)
>>> grey_img.getpixel((0,0))
229
```

#Binary images: pixel can be 0 or 255

```python
>>> bw_img.getpixel((0,0))
255
```
for i in range(400,600):
    for j in range(300):
        grey_img.putpixel((i,j), 200)
grey_img.show()
s_img = Image.new(mode='L',
size=(40,20), color=0)
for i in range(40):
    for j in range(0, 20, 2):
        s_img.putpixel((i,j), 255)
s_img.show()
Command Line Arguments

- Every Python script can be run as a command line from the terminal.
- In Windows we can open the command prompt (cmd).
- The command: python test.py
We can pass argument in the command line.

Arguments come after the script name.

There should be a space in the command line between each of the arguments.
Command Line Arguments

- The Python `sys` module provides access to any command-line arguments via the `sys.argv`. We must import `sys` in order to use it.
- `sys.argv` is the list of command-line arguments.
- Each element in that list is a string.
- The first element is the name of the script.
- All other elements are the given arguments.
import sys

NUMBER_OF_ARGUMENTS = 2

if __name__ == "__main__":
    if len(sys.argv) == NUMBER_OF_ARGUMENTS+1:
        script_name = sys.argv[0]
        word = sys.argv[1]
        num = int(sys.argv[2])
Run -> Edit configurations..
Add the arguments to Script parameters
Many of the IDE (integrated development environment) applications contain a debugger.

Debugger is a very useful tool that can help in debugging our code.

It enables one to “stop” the run at a given point and examine the current variables values.

One disadvantage – running in debug mode might be very slow.
Do we really understand the code? What really happens in each step?

def radixsort(lst, radix=10):
    max_val = max(lst)
    power = 0
    # getting the number of digits of the largest number
    while radix**power < max_val:
        power += 1
    for p in range(power):
        # splitting into buckets according to the current digit order
        factor = radix**p
        buckets = [list() for d in range(radix)]
        for val in lst:
            tmp = val/factor
            buckets[int(tmp % radix)].append(val)
        # appending the buckets to one list according the last sorting
        lst = []
        for b in range(radix):
            buck = buckets[b]
            for val in buck:
                lst.append(val)
    return lst

if __name__ == "__main__":
    lst = [49, 812, 71500, 195, 4317, 582, 34, 3856]
    sorted_lst = radixsort(lst)
    print(sorted_lst)
Let's debug!

Add breakpoint by clicking next to the line we would like to stop and check variables.
Running in debug mode

- Run -> Debug
Getting variables values at breakpoint

You can see current variable values in grey.
The variables values are present at the debug tub (at the bottom of the window)
Continuing execution

Click here to get to the next line in the code

Click here to reach next breakpoint
Continuing execution

Data is updated as we continue