Programming for Engineers

Lecture 2 - matrices and array processing

Radoslav Škoviera

Table of contents

2D arrays (matrices), array processing (search, cumulative sum)

Prelude: Visualizing (printing) arrays

For debugging purposes, it is useful to see what is in an array. Smaller arrays are easy to print "whole":

```
import numpy as np
a = np.random.permutation(10)
print(a)
```

```
[5 9 3 1 2 0 4 7 6 8]
```

You can use the `join` method of strings to make the output nicer. More on that later, when we get to strings.

```
def pretty_print_array(a):
    print('[' + ', '.join([str(x) for x in a]) + ']')

pretty_print_array(a)
```

```
[5, 9, 3, 1, 2, 0, 4, 7, 6, 8]
```

For larger arrays, you need to be creative - the solution will depend on what you need. For example, it might be enough to print the first or last few items. Print the array as rows (Python kinda does that but "uncontrollably"):

```
def print_as_rows(a, row_length=10):
    for i in range(0, len(a), row_length):
        print(a[i:i+row_length])

a = np.random.permutation(100)
print_as_rows(a)
```

```
[70 41 91  6 81 46 94 99 60 32]
[21 65 67 29 55 18 82 88 45 84]
[51 17 10 73 50 77 96 97 12 40]
[66 64 79  2 23 38 80 76 93 14]
[26  3 86 16 92 78 15 49 59 85]
[63  8  0 36 44 25 56 39 62 47]
[83 24 71 19 58 52 72 13  1 28]
[61 98 53 87 74 11 33  9 95 69]
[37 89 31 42 75 27 30 43 20  4]
[68 90 35 48 57 22 34  7  5 54]
```

You can even improve it to have nicer output. Again, more on that next time, when we get to strings.

```
def pretty_print_as_rows(a, row_length=10):
    print('[')
    for i in range(0, len(a), row_length):
        print(', '.join([str(x) for x in a[i:i+row_length]]), end=',\n')
    print(']')
```

```
a = np.random.permutation(100).tolist()
pretty_print_as_rows(a)
```

```
[
99, 82, 36, 90, 49, 71, 74, 8, 22, 88,
54, 20, 66, 93, 55, 59, 47, 77, 32, 31,
76, 25, 11, 48, 18, 30, 69, 89, 2, 0,
91, 64, 6, 43, 23, 58, 67, 41, 70, 13,
61, 15, 78, 44, 19, 81, 39, 63, 87, 33,
50, 56, 27, 26, 53, 92, 86, 5, 45, 17,
65, 4, 35, 10, 34, 97, 42, 1, 83, 16,
73, 96, 38, 79, 3, 80, 52, 75, 24, 9,
12, 68, 51, 94, 14, 62, 57, 98, 7, 85,
60, 40, 37, 29, 46, 84, 28, 21, 72, 95,
]
```

Printing matrices:

(remember to define/run cell with the `pretty_print` function above)

Visualizing a matrix using the build-in `print` function might not be the best approach:

```
mat = np.random.randint(0, 20, size=(17, 11)).tolist()
print(mat)
```

```
[[5, 10, 4, 11, 19, 2, 14, 5, 9, 7, 5], [5, 4, 14, 16, 17, 17, 13, 10, 0, 4, 4], [16, 11, 0,
```

```
def print_matrix(m):
    print('[', end='')
    n_rows = len(m)
    for ri, row in enumerate(m):
        print(row, end=',\n' if ri < n_rows - 1 else '')
    print(']')

def formatted_print_matrix(m):  # works only for integers
    n_rows = len(m)
    maximum_value = abs(max([max(row) for row in m])) + 1e-6
    if maximum_value > 0:
        max_digits =  int(np.ceil(np.log10(maximum_value)))
    else:  # all zeros
```

```
        max_digits = 1
    print('[', end='')
    for ri, row in enumerate(m):
        row_prefix = ('' if ri == 0 else ' ') + '['
        row_numbers = ', '.join([f"{x:{max_digits}d}" for x in row])
        row_end='],\n' if ri < n_rows - 1 else ']'
        print(row_prefix + row_numbers, end=row_end)
    print(']')

mat = np.random.randint(0, 20, size=(17, 11)).tolist()
print("Row-by-row print:")
print_matrix(mat)
print()
print("Formatted print:")
formatted_print_matrix(mat)
```

```
Row-by-row print:
[[14, 7, 10, 16, 14, 18, 18, 18, 13, 15, 13],
[16, 0, 8, 2, 18, 0, 3, 8, 11, 13, 10],
[17, 12, 13, 7, 1, 1, 7, 5, 5, 7, 19],
[8, 14, 1, 2, 10, 8, 1, 5, 1, 3, 2],
[18, 6, 17, 5, 1, 10, 1, 10, 9, 16, 4],
[14, 18, 14, 10, 3, 1, 2, 18, 17, 12, 13],
[15, 16, 0, 5, 19, 17, 15, 4, 15, 3, 12],
[4, 3, 8, 0, 15, 4, 13, 7, 5, 1, 9],
[13, 6, 19, 8, 15, 7, 17, 12, 5, 11, 16],
[1, 8, 11, 5, 8, 10, 16, 16, 8, 0, 11],
[15, 16, 7, 3, 7, 15, 5, 13, 18, 16, 12],
[17, 8, 18, 4, 11, 16, 3, 6, 0, 19, 11],
[6, 13, 5, 16, 5, 0, 7, 1, 8, 4, 18],
[2, 0, 9, 9, 18, 2, 12, 13, 18, 16, 19],
[15, 19, 12, 13, 7, 0, 15, 11, 0, 14, 16],
[7, 9, 12, 5, 13, 1, 0, 2, 15, 19, 19],
[15, 18, 13, 19, 18, 4, 14, 16, 1, 10, 4]]

Formatted print:
[[14,  7, 10, 16, 14, 18, 18, 18, 13, 15, 13],
 [16,  0,  8,  2, 18,  0,  3,  8, 11, 13, 10],
 [17, 12, 13,  7,  1,  1,  7,  5,  5,  7, 19],
 [ 8, 14,  1,  2, 10,  8,  1,  5,  1,  3,  2],
 [18,  6, 17,  5,  1, 10,  1, 10,  9, 16,  4],
 [14, 18, 14, 10,  3,  1,  2, 18, 17, 12, 13],
```

```
[15, 16,  0,  5, 19, 17, 15,  4, 15,  3, 12],
[ 4,  3,  8,  0, 15,  4, 13,  7,  5,  1,  9],
[13,  6, 19,  8, 15,  7, 17, 12,  5, 11, 16],
[ 1,  8, 11,  5,  8, 10, 16, 16,  8,  0, 11],
[15, 16,  7,  3,  7, 15,  5, 13, 18, 16, 12],
[17,  8, 18,  4, 11, 16,  3,  6,  0, 19, 11],
[ 6, 13,  5, 16,  5,  0,  7,  1,  8,  4, 18],
[ 2,  0,  9,  9, 18,  2, 12, 13, 18, 16, 19],
[15, 19, 12, 13,  7,  0, 15, 11,  0, 14, 16],
[ 7,  9, 12,  5, 13,  1,  0,  2, 15, 19, 19],
[15, 18, 13, 19, 18,  4, 14, 16,  1, 10,  4]]
```

Matrices

Matrices are 2D arrays. They have two dimensions: rows and columns. The number of rows is often called the height and the number of columns is often called the width. This is similar to image, although, images have typically flipped dimensions: x-axis is width (columns) and y-axis is height (rows).

```python
m = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
formatted_print_matrix(m)
also_m = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]
formatted_print_matrix(also_m)
m_as_well = [[j + i * 3 for j in range(1, 4)] for i in range(3)]
formatted_print_matrix(m_as_well)
```

```
[[1, 2, 3],
 [4, 5, 6],
 [7, 8, 9]]
[[1, 2, 3],
 [4, 5, 6],
 [7, 8, 9]]
[[1, 2, 3],
 [4, 5, 6],
 [7, 8, 9]]
```

Creating a matrix in a loop (the following two functions are equivalent):

```
def create_counting_matrix(height, width):
    return [[j + i * width for j in range(1, width + 1)] for i in range(height)]

def create_counting_matrix_unpacked(height, width):
    mat = []
    for i in range(height):
        row = []
        for j in range(1, width + 1):
            row.append(j + i * width)
        mat.append(row)
    return mat

formatted_print_matrix(create_counting_matrix(3, 4))
```

```
[[ 1,  2,  3,  4],
 [ 5,  6,  7,  8],
 [ 9, 10, 11, 12]]
```

Matrix pre-allocation:

```
def create_empty(n_rows, n_cols):
    return [[0] * n_cols for _ in range(n_rows)]
M, N = 3, 4
mat = create_empty(M, N)
print(f"Empty {M} by {N} matrix:")
formatted_print_matrix(mat)
```

```
Empty 3 by 4 matrix:
[[     0,      0,      0,      0],
 [     0,      0,      0,      0],
 [     0,      0,      0,      0]]
```

Filling matrices with values

Filling matrix with a constant value. *Warning*, the following function might fail in case of "jagged" matrices - this is why it's best to only use 2D arrays (matrices) with equal column lengths (second dimension). There is a simple workaround - get `n_col` at each row-loop. Nonetheless, the best approach is to avoid jagged matrices in the first place.

```python
def fill_matrix_with_value(matrix, value):
    n_row = len(matrix)
    if n_row == 0:  # let's check if matrix is empty to avoid some errors
        print("Matrix is empty!")
        return
    n_col = len(matrix[0])
    for r in range(n_row):
        for c in range(n_col):
            matrix[r][c] = value


m = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
fill_matrix_with_value(m, 11)
formatted_print_matrix(m)
```

```
[[11, 11, 11],
 [11, 11, 11],
 [11, 11, 11]]
```

Random fill:

```python
import random

def fill_matrix_with_randint(matrix, min_value=0, max_value=100):
    n_row = len(matrix)
    if n_row == 0:  # let's check if matrix is empty to avoid some errors
        print("Matrix is empty!")
        return
    n_col = len(matrix[0])

    for r in range(n_row):
        for c in range(n_col):
            matrix[r][c] = random.randint(min_value, max_value)

mat = [[1] * 5 for _ in range(4)]
fill_matrix_with_randint(mat)
formatted_print_matrix(mat)
```

```
[[30, 12, 68, 88, 33],
 [91, 87, 30,  9,  6],
 [60, 65, 70, 55, 55],
 [34, 23, 41, 74,  7]]
```

Shuffle matrix elements. There are few ways to do it. Here, we will create "flattened" indices, shuffle them and assign values to them from the original matrix.

```
def shuffle_matrix(matrix):
    n_row = len(matrix)
    m_cols = len(matrix[0])
    flat_indices = list(range(n_row * m_cols))
    random.shuffle(flat_indices)
    shuffled_matrix = create_empty(n_row, m_cols)
    for r in range(n_row):
        for c in range(m_cols):
            flat_index = r * m_cols + c
            shuffled_matrix[r][c] = matrix[int(flat_indices[flat_index]
            / m_cols)][flat_indices[flat_index] % m_cols]

    return shuffled_matrix

m = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
sm = shuffle_matrix(m)
print("Original matrix:")
formatted_print_matrix(m)
print("Shuffled matrix:")
formatted_print_matrix(sm)
```

```
Original matrix:
[[1, 2, 3],
 [4, 5, 6],
 [7, 8, 9]]
Shuffled matrix:
[[1, 6, 2],
 [9, 7, 4],
 [3, 5, 8]]
```

Matrix operations

First, let's create some matrices:

```
mat_A = create_empty(3, 4)
fill_matrix_with_randint(mat_A, 0, 20)
mat_B = create_empty(3, 4)
fill_matrix_with_randint(mat_B, 0, 20)
```

```
print("Matrix A:")
formatted_print_matrix(mat_A)
print("Matrix B:")
formatted_print_matrix(mat_B)
```

```
Matrix A:
[[19, 19, 14, 20],
 [19, 16,  0,  5],
 [17, 11,  0, 19]]
Matrix B:
[[19, 14,  1,  0],
 [ 0, 18, 16,  6],
 [ 9,  7,  5,  7]]
```

Addition

Compute $C = A + B$. This one is easy, we just need to loop through elements of the two matrices and add them.

```
def add_two_matrices(mat_A, mat_B):
    n_row = len(mat_A)
    n_col = len(mat_A[0])
    mat_C = create_empty(n_row, n_col)
    for r in range(n_row):
        for c in range(n_col):
            mat_C[r][c] = mat_A[r][c] + mat_B[r][c]
    return mat_C

mat_C = add_two_matrices(mat_A, mat_B)
print("Result of addition of two matrices:")
formatted_print_matrix(mat_C)
```

```
Result of addition of two matrices:
[[38, 33, 15, 20],
 [19, 34, 16, 11],
 [26, 18,  5, 26]]
```

Transposition

We want to compute A^T. We just swap the indices for the output matrix.

```python
def transpose(mat):
    n_row = len(mat)  # get the matrix dimensions
    n_col = len(mat[0])
    mat_T = create_empty(n_col, n_row)  # create a new matrix
    for r in range(n_row):
        for c in range(n_col):
            mat_T[c][r] = mat[r][c]  # swap the row and column indices
    return mat_T

mat_T = transpose(mat_A)
print("Transposed matrix:")
formatted_print_matrix(mat_T)
```

```
Transposed matrix:
[[19, 19, 17],
 [19, 16, 11],
 [14,  0,  0],
 [20,  5, 19]]
```

Multiplication

Compute $C = A \times B$. It is not simple, we need to loop through the elements of the matrices and compute their product. The result will be a new matrix with dimensions $n \times m$.

```python
def multiply_two_matrices(mat_A, mat_B):
    n_row_A = len(mat_A)
    n_col_A = len(mat_A[0])
    n_row_B = len(mat_B)
    n_col_B = len(mat_B[0])

    if n_col_A != n_row_B:  # the matrices need to have the correct shapes
        raise ValueError("Matrices A and B cannot be multiplied. "
                         "Matrix B needs to have the same number of columns "
                         f"(has {n_col_B}) as matrix A has rows (has {n_row_A}).")
    # this function will not be able to deal with broadcasting!
    mat_C = create_empty(n_row_A, n_col_B)
    for r in range(n_row_A):
        for c in range(n_col_B):
            mat_C[r][c] = 0
            for k in range(n_col_A):
                mat_C[r][c] += mat_A[r][k] * mat_B[k][c]
```

```
    return mat_C

mat_C = multiply_two_matrices(mat_A, transpose(mat_B))
print("Result of multiplication of two matrices:")
formatted_print_matrix(mat_C)
```

```
Result of multiplication of two matrices:
[[641, 686, 514],
 [585, 318, 318],
 [477, 312, 363]]
```

Miscellaneous operations on matrices

Comparison

Compare two matrices "plain" - just iterate over the rows & columns and compare corresponding elements in both matrices. Sometimes, we might want to compare elements in two matrices, regardless of their position. In that case, it is easier to "flatten" one of the matrices - it is easier to iterate over a 1D array. The 'unique' version of the second function requires us to remove duplicates from the flattened matrix first (you can remove them from any matrix but again, it is easier to do it for a 1D array). Otherwise, we would need to remove them at the end, when comparing the values.

```
def compare_two_matrices(mat_A, mat_B):
    n_row = len(mat_A)
    n_col = len(mat_A[0])
    if n_row != len(mat_B) or n_col != len(mat_B[0]):
        return False  # matrices don't have the same shape => cannot be equal
    for r in range(n_row):
        for c in range(n_col):
            if mat_A[r][c] != mat_B[r][c]:
                return False  # "early" termination for efficiency
    return True

def remove_duplicates(arr):
    i = 0
    while i < len(arr) - 1:
        j = i + 1
        while j < len(arr):
            if arr[i] == arr[j]:
                del arr[j]
```

```
            j += 1
        i += 1

def find_equal_values(mat_A, mat_B, unique=False):
    n_row_A = len(mat_A)
    n_col_A = len(mat_A[0])
    n_row_B = len(mat_B)
    n_col_B = len(mat_B[0])
    flat_values_B = []
    for r in range(n_row_B):
        flat_values_B.extend(mat_B[r])

    if unique:  # remove duplicates,
        # otherwise the `remove` method below might not be enough
        remove_duplicates(flat_values_B)

    equal_values = []
    for r in range(n_row_A):
        for c in range(n_col_A):
            item_A = mat_A[r][c]
            if item_A in flat_values_B:
                equal_values.append(item_A)
                if unique:
                    flat_values_B.remove(item_A)
    return equal_values

mat_A = create_empty(3, 4)
fill_matrix_with_randint(mat_A, 0, 10)
mat_B = create_empty(3, 4)
fill_matrix_with_randint(mat_B, 0, 10)
print("Matrix A:")
formatted_print_matrix(mat_A)
print("Matrix B:")
formatted_print_matrix(mat_B)

print("Matrices are equal:", compare_two_matrices(mat_A, mat_B))
print("Equal values:", find_equal_values(mat_A, mat_B))
print("Uniquely equal values:", find_equal_values(mat_A, mat_B, unique=True))
```

```
Matrix A:
[[ 1,  1,  1, 10],
 [ 7,  1,  8,  6],
```

```
 [ 0,  7,  5,  3]]
Matrix B:
[[10,  1,  9,  6],
 [ 9,  5,  8, 10],
 [ 7,  5,  1,  8]]
Matrices are equal: False
Equal values: [1, 1, 1, 10, 7, 1, 8, 6, 7, 5]
Uniquely equal values: [1, 10, 7, 8, 6, 5]
```

All different

Find whether the matrix contains only unique values. One option is to use the function above. Then, the number of uniquely equal values with itself must equal number of elements.

```python
# print("Matrix contains only unique values:", len() == 0)
mat_U = create_empty(3, 4)
fill_matrix_with_randint(mat_U, 0, 50)
formatted_print_matrix(mat_U)
num_equal_values = len(find_equal_values(mat_U, mat_U, unique=True))
if num_equal_values == len(mat_U) * len(mat_U[0]):
    print("Matrix contains only unique values.")
else:
    print("Matrix does not contain only unique values.")
```

```
[[49, 27,  4, 20],
 [19,  1, 42, 46],
 [ 9, 43, 31, 22]]
Matrix contains only unique values.
```

Alternatively, a custom method with similar approach can be made. The easiest approach is to flatten the matrix and then basically compare "two" arrays.

```python
def all_different(mat):
    # flatten the matrix
    flat_values = []
    for r in range(len(mat)):
        flat_values.extend(mat[r])
    n = len(flat_values)
    for i in range(n - 1):
        for j in range(i + 1, n):
            if flat_values[i] == flat_values[j]:
```

```
                return False
    return True

mat_U = create_empty(3, 4)
fill_matrix_with_randint(mat_U, 0, 50)
formatted_print_matrix(mat_U)
if all_different(mat_U):
    print("Matrix contains only unique values.")
else:
    print("Matrix does not contain only unique values.")
```

```
[[49, 46, 48, 34],
 [13, 47, 38, 36],
 [32,  8, 26,  8]]
Matrix does not contain only unique values.
```

"Graphical" matrices (images)

Raw images are basically matrices, where each element represents an intensity value of the corresponding pixel. Color images are 3D matrices, where each element is a 3-tuple of intensity values of the color channels but we will not go there.

However, we can have a little fun with gray-scale images.

```
img = [
    [100, 100, 100, 100, 100, 100, 100, 100, 100, 100],
    [100,   0 ,  0 ,  0,   0 ,  0 ,  0,   0,   0, 100],
    [  0,   0, 128, 128,   0,   0, 128, 128,   0,   0],
    [  0,   0, 255, 255,   0,   0, 255, 255,   0,   0],
    [  0,   0, 255, 255,   0,   0, 255, 255,   0,   0],
    [  0,   0, 128, 128,   0,   0, 128, 128,   0,   0],
    [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0],
    [  0,   0,   0,   0,   0, 200,   0,   0,   0,   0],
    [  0,   0,   0,   0,   0, 200,   0,   0,   0,   0],
    [  0,   0,   0,   0, 200, 200,   0,   0,   0,   0],
    [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0],
    [  0, 255,   0,   0,   0,   0,   0,   0, 255,   0],
    [  0, 255, 255, 255, 255, 255, 255, 255, 255,   0],
    [  0,   0, 255, 128, 128, 128, 128, 255,   0,   0],
    [  0,   0,   0, 255, 255, 255, 255,   0,   0,   0],
    [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0],
```

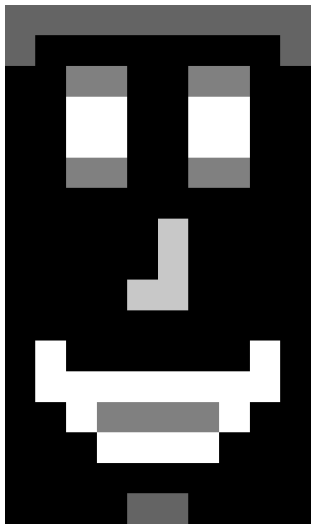
```
    [  0,   0,   0,   0, 100, 100,   0,   0,   0,   0],
]

print(pretty_print_as_rows(img))

from matplotlib import pyplot as plt

plt.imshow(img, cmap='gray')
plt.axis('off')
plt.show()
```

```
[
[100, 100, 100, 100, 100, 100, 100, 100, 100, 100], [100, 0, 0, 0, 0, 0, 0, 0, 0, 100], [0, 0
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0], [0, 255, 0, 0, 0, 0, 0, 0, 255, 0], [0, 255, 255, 255, 255, 2
]
None
```



Array & matrix processing

Search

Linear search

Requires iterating over the array, until the item with the correct value is found.

```
import numpy as np
N = 100
a = np.random.permutation(N)
searched_item = 50

def linear_search(a, x):
    for i in range(len(a)):
        if a[i] == x:
            return i
    return None

print("Linear search run time:")
%timeit linear_search(a, searched_item)
```

```
Linear search run time:
742 ns ± 32.8 ns per loop (mean ± std. dev. of 7 runs, 1,000,000 loops each)
```

Computational complexity side-quest

The asymptotic time of the linear search is $O(n)$, the 'actual' time depends on the 'properties' of the array - where the item is located in the array.

```
N = 100
a = np.random.permutation(N)
searched_item = a[N // 2]
print("Searching for an item in the middle of the array:")
%timeit linear_search(a, searched_item)

searched_item = a[0]
print("Searching for an item at the beginning of the array:")
%timeit linear_search(a, searched_item)

searched_item = a[-1]
print("Searching for an item at the end of the array:")
%timeit linear_search(a, searched_item)
```

```
Searching for an item in the middle of the array:
4.64  s ± 123 ns per loop (mean ± std. dev. of 7 runs, 100,000 loops each)
Searching for an item at the beginning of the array:
201 ns ± 8.16 ns per loop (mean ± std. dev. of 7 runs, 1,000,000 loops each)
Searching for an item at the end of the array:
9.07  s ± 231 ns per loop (mean ± std. dev. of 7 runs, 100,000 loops each)
```

Binary search

Requires pre-sorted array but asymptotic runtime is $O(\log n)$ - we only look which "half" of the array might contain the item, then discard the other half.

```
def binary_search_loop(a, x):
    low = 0  # lower-bound index for search
    high = len(a) - 1  # upper-bound index for search
    loop_count = 1
    while low <= high:  # while there are indices to search
        mid = (low + high) // 2  # compute the middle index
        if a[mid] < x:  # if the middle point is less than the item
            low = mid + 1  # "discard" the left half (search from the middle + 1)
        elif a[mid] > x:  # if the middle point is greater than the item
            high = mid - 1  # "discard" the right half (search from the middle - 1)
        else:  # otherwise, the middle point is the item
            return mid, loop_count
        loop_count += 1
    return None, loop_count  # we did not find the item

def binary_search_recursion(a, x):
    if len(a) == 0:  # we got an empty array
        return None  # the item is for sure not in it
    mid_idx = len(a) // 2  # compute the middle point index
    if a[mid_idx] == x:  # if the middle point is the item
        return mid_idx  # we found it, job done
    elif a[mid_idx] < x:  # if the middle point is less than the item
        # we look at the right half of the array
        return binary_search_recursion(a[mid_idx + 1:], x)
    else:
        # otherwise, we look at the left half of the array
        return binary_search_recursion(a[:mid_idx], x)

# recursion is slower in this case, as we are copying the whole array,
# instead of just using the index.
sa = np.sort(a)
print("Binary search run time:")
%timeit binary_search_loop(sa, 5)
%timeit binary_search_recursion(sa, 5)

print("Buuuuuut...")
print("'Fair' binary search run time (includes sorting):")
%timeit binary_search_loop(np.sort(a), 5)
```

```
%timeit binary_search_recursion(np.sort(a), 5)
```

```
Binary search run time:
1.11  s ± 30.9 ns per loop (mean ± std. dev. of 7 runs, 1,000,000 loops each)
2.21  s ± 54.4 ns per loop (mean ± std. dev. of 7 runs, 100,000 loops each)
Buuuuuut...
'Fair' binary search run time (includes sorting):
2.54  s ± 98.6 ns per loop (mean ± std. dev. of 7 runs, 100,000 loops each)
3.72  s ± 115 ns per loop (mean ± std. dev. of 7 runs, 100,000 loops each)
```

The advantage of binary search is relatively constant search time. As you can see from the following, the search time is shorter than "average" search time for linear search: $O(\log n) < O(n)$. This includes even sorting the array.

```
print("Searching for an item in the middle of the array, "
"using binary search:")
searched_item = sa[N // 2]
%timeit binary_search_loop(np.sort(a), searched_item)

print("Searching for an item at the beginning of the array, "
"using binary search:")
searched_item = sa[0]
%timeit binary_search_loop(np.sort(a), searched_item)

print("Searching for an item at the end of the array, "
"using binary search:")
searched_item = sa[-1]
%timeit binary_search_loop(np.sort(a), searched_item)
```

```
Searching for an item in the middle of the array, using binary search:
2.67  s ± 66.3 ns per loop (mean ± std. dev. of 7 runs, 100,000 loops each)
Searching for an item at the beginning of the array, using binary search:
2.83  s ± 61.9 ns per loop (mean ± std. dev. of 7 runs, 100,000 loops each)
Searching for an item at the end of the array, using binary search:
2.48  s ± 150 ns per loop (mean ± std. dev. of 7 runs, 100,000 loops each)
```

Interpolation search

Even more restrictive requirements than binary search - the array needs to be 'uniformly' (equal distribution of values) sorted. It uses similar approach as binary search but uses "an

educated guess" of where the item might be (this is where the uniformity comes into play). The asymptotic run time is $O(\log(\log n))$.

```
def interpolation_search(a, x):
    low = 0  # lower-bound index for search
    high = len(a) - 1  # upper-bound index for search
    loop_count = 1
    while low <= high and a[low] <= x <= a[high]:  # while there are indices to search
        mid = low + int(((x - a[low]) * (high - low)) / (a[high] - a[low]))
        if a[mid] < x:  # if the middle point is less than the item
            low = mid + 1  # "discard" the left half (search from the middle + 1)
        elif a[mid] > x:  # if the middle point is greater than the item
            high = mid - 1  # "discard" the right half (search from the middle - 1)
        else:  # otherwise, the middle point is the item
            return mid, loop_count
        loop_count += 1
    return None, loop_count  # we did not find the item

print("Searching for an item in the middle of the array, using interpolation search:")
searched_item = sa[N // 2]
%timeit interpolation_search(np.sort(a), searched_item)

print("Searching for an item at the beginning of the array, using interpolation search:")
searched_item = sa[0]
%timeit interpolation_search(np.sort(a), searched_item)

print("Searching for an item at the end of the array, using interpolation search:")
searched_item = sa[-1]
%timeit interpolation_search(np.sort(a), searched_item)
```

```
Searching for an item in the middle of the array, using interpolation search:
2.27  s ± 37.4 ns per loop (mean ± std. dev. of 7 runs, 100,000 loops each)
Searching for an item at the beginning of the array, using interpolation search:
2.35  s ± 63.4 ns per loop (mean ± std. dev. of 7 runs, 100,000 loops each)
Searching for an item at the end of the array, using interpolation search:
2.33  s ± 134 ns per loop (mean ± std. dev. of 7 runs, 100,000 loops each)
```

The awesome speed of interpolation search:

```
a = np.random.permutation(N)
sa = np.sort(a)
searched_item = sa[N // 2 + 1]
```

```
print("Interpolation search run time:")
%timeit interpolation_search(sa, searched_item)
v, i_loop_count = interpolation_search(sa, searched_item)
print(f"Interpolation search 'step' count: {i_loop_count}")

print("Binary search run time:")
%timeit binary_search_loop(sa, searched_item)
b, b_loop_count = binary_search_loop(sa, searched_item)
print(f"Binary search 'step' count: {b_loop_count}")
```

```
Interpolation search run time:
907 ns ± 22.8 ns per loop (mean ± std. dev. of 7 runs, 1,000,000 loops each)
Interpolation search 'step' count: 1
Binary search run time:
1.44  s ± 22.9 ns per loop (mean ± std. dev. of 7 runs, 1,000,000 loops each)
Binary search 'step' count: 7
```

Unfortunately, the "single step" search only works if the array is completely uniform.

```
a = np.random.choice(N**3, size=N, replace=False) / 100  # possibly non-uniform array
sa = np.sort(a)
searched_item = sa[N // 2 + 1]

print("Interpolation search run time:")
%timeit interpolation_search(sa, searched_item)
v, i_loop_count = interpolation_search(sa, searched_item)
print(f"Interpolation search 'step' count: {i_loop_count}")
print("Binary search run time:")
%timeit binary_search_loop(sa, searched_item)
v, b_loop_count = binary_search_loop(sa, searched_item)
print(f"Binary search 'step' count: {b_loop_count}")
```

```
Interpolation search run time:
1.83  s ± 90.8 ns per loop (mean ± std. dev. of 7 runs, 100,000 loops each)
Interpolation search 'step' count: 2
Binary search run time:
1.49  s ± 40.2 ns per loop (mean ± std. dev. of 7 runs, 1,000,000 loops each)
Binary search 'step' count: 7
```

Find all

The search finds the first occurrence of the searched item. Which one will depend on the search method. Sometimes, it is required to find all occurrences of the searched item in the array.

```
import numpy as np
def find_all(a, x):
    indices = []
    for i in range(len(a)):
        if a[i] == x:
            indices.append(i)
    return indices

N = 100
a = np.random.choice(N**3, size=N, replace=True)
searched_item = a[N // 2]
occurrences = find_all(a, searched_item)
print("The value", searched_item, "occurs", len(occurrences),
    "times in the array at indices", occurrences)
```

```
The value 896518 occurs 1 times in the array at indices [50]
```

'Statistical' computations on arrays

Simple operations

Minimum, maximum

Python has built-in `min()` and `max()` functions. Here, we wil see simple implementation of these functions, simply to practice working with arrays. In practice, you can use the built-in functions.

```
import numpy as np
my_list = np.random.permutation(10).tolist()
print("Input array:", my_list)

def minimum(a):
    min_value = a[0]
    for i in range(1, len(a)):
        if a[i] < min_value:
            min_value = a[i]
```

```
        # Alternatively:
        # min_value = min(min_value, a[i])
    return min_value

def maximum(a):
    max_value = a[0]
    for i in range(1, len(a)):
        if a[i] > max_value:
            max_value = a[i]
        # Alternatively:
        # max_value = max(max_value, a[i])
    return max_value

print("Minimum: ", minimum(my_list))
print("Maximum: ", maximum(my_list))
```

```
Input array: [4, 3, 6, 7, 9, 0, 5, 8, 1, 2]
Minimum:  0
Maximum:  9
```

Mean

First, we need to compute the sum:

```
import numpy as np
my_list = np.random.permutation(10).tolist()
print("Input array:", my_list)

def sum(a):
    s = 0
    for i in range(len(a)):
        s += a[i]
    return s

print("Sum: ", sum(my_list))
```

```
Input array: [1, 7, 6, 3, 0, 8, 2, 5, 4, 9]
Sum:  45
```

Mean is then simply the sum divided by the number of items:

```
def mean(a):
    return sum(a) / len(a)

print("Mean: ", mean(my_list))
```

```
Mean:  4.5
```

Cumulative sum (prefix sum)

Summing items between two indices is useful for many tasks. For example, computing average temperature between two specified dates. The following method computes sum between two indices in an array (inclusive of the start and the end indices).

```
import numpy as np
temp_measurements = (np.random.rand(200) * 10).tolist()
print("Input array: " + ', '.join([f"{x:.2f}"
    for x in temp_measurements[:10]]) + ", ...")

def range_sum(a, start, end):
    s = 0
    for i in range(start, end + 1):  # +1 to include the last element
        s += a[i]
    return s

print("Sum between indices 5 and 15:", range_sum(temp_measurements, 5, 15))
```

```
Input array: 5.71, 7.48, 5.20, 1.36, 2.26, 7.49, 6.49, 2.26, 8.65, 5.27, ...
Sum between indices 5 and 15: 64.07973638424976
```

This is fine, if we need to to this once, but what if we need to do it many times?

```
def generate_range_queries(a, n, query_len):
    numel_a = len(a)
    assert query_len <= numel_a, "Query length must be less than the array length"
    max_pos = numel_a - query_len
    queries = []
    for _ in range(n):
        start = np.random.randint(0, max_pos)
        end = start + query_len - 1
        queries.append((start, end))
```

```
    return queries

print("Summing once run time:")
%timeit range_sum(temp_measurements, 50, 150)
print("Summing 100 times run time:")
queries = generate_range_queries(temp_measurements, 100, 50)
%timeit [range_sum(temp_measurements, start, end) for start, end in queries]
```

```
Summing once run time:
2.43 s ± 70 ns per loop (mean ± std. dev. of 7 runs, 100,000 loops each)
Summing 100 times run time:
128 s ± 1.59 s per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
```

Is there a better way? Why yes! We will use **cumulative sum** (also known as prefix sum). It is done bone adding all the previous values to the current value in the array.

```
def cumulative_sum(a):
    cs = [0] * len(a)  # pre-allocate
    for i in range(1, len(cs)):
        cs[i] = a[i] + cs[i - 1]  # add previous sum to the current element
    return cs
my_list = list(range(10))
print("Cumulative sum of ordered numbers from 0 to 9: ")
print("Input array:   [" + ', '.join([f"{x:2d}" for x in my_list]) + "]")
print("Cumulative sum: " + str(cumulative_sum(my_list)))
print()
my_list = np.random.permutation(10).tolist()
print("Cumulative sum of randomly shuffled numbers: ")
print("Input array:   [" + ', '.join([f"{x:2d}" for x in my_list]) + "]")
print("Cumulative sum: " + str(cumulative_sum(my_list)))
```

```
Cumulative sum of ordered numbers from 0 to 9:
Input array:   [ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9]
Cumulative sum: [0, 1, 3, 6, 10, 15, 21, 28, 36, 45]

Cumulative sum of randomly shuffled numbers:
Input array:   [ 4,  5,  9,  0,  7,  3,  2,  1,  8,  6]
Cumulative sum: [0, 5, 14, 14, 21, 24, 26, 27, 35, 41]
```

Table 1: “Visualizing” cumulative sum

index	0	1	2	3	4	5	6	7	8	9
value	4	2	0	8	1	5	7	9	3	6
cumsum	4	6	6	14	15	20	27	36	39	45
	4	4+2	4+2+0	4+2+0+8	...					

If we have the cumulative sum precomputed, to get sum of elements between two indices a and b, we can simply compute $cumsum[b] - cumsum[a-1]$. Therefore, instead of $b-a$ additions, we only compute one addition.

```
my_list = np.random.permutation(10).tolist()

def range_sum_cs(a_cumsum, start, end):
    return a_cumsum[end] - a_cumsum[start - 1]

print(my_list)
start, end = 3, 7
print(f"Sum between indices {start} and {end}:",
    range_sum_cs(cumulative_sum(my_list), start, end))
# sanity check with the "simple" method:
print("This is the same as with the simple `range_sum` method:",
    range_sum(my_list, start, end) == range_sum_cs(cumulative_sum(my_list), start, end))
```

```
[1, 9, 3, 7, 5, 0, 6, 2, 8, 4]
Sum between indices 3 and 7: 20
This is the same as with the simple `range_sum` method: True
```

Now, let’s say we have the cumulative sum precomputed and we want to compute the range sum for 100 different ranges. How long will it take?

```
print("Run time of cumulative sum computation:")
%timeit cumulative_sum(temp_measurements)

cumsum_measurements = cumulative_sum(temp_measurements)
print("Summing 100 times run time:")
%timeit [range_sum_cs(cumsum_measurements, start, end) for start, end in queries]
```

```
Run time of cumulative sum computation:
8.38  s ± 116 ns per loop (mean ± std. dev. of 7 runs, 100,000 loops each)
```

```
Summing 100 times run time:
7.4  s ± 146 ns per loop (mean ± std. dev. of 7 runs, 100,000 loops each)
```

Of course, the actual effectiveness of the cumulative / prefix sum depends on the number of queries and the length of the queries vs. the length of the array. For a few short queries in a long array, it might take longer to just compute the cumulative sum than to compute all the queries.

```
short_queries = generate_range_queries(temp_measurements, 10, 5)
print("Run time of simple range sum:")
%timeit [range_sum(temp_measurements, start, end) for start, end in short_queries]
print("Run time of cumulative sum & range sum:")
%timeit cumsum_measurements = cumulative_sum(temp_measurements); [range_sum_cs(cumsum_measure
```

```
Run time of simple range sum:
2.67  s ± 82.7 ns per loop (mean ± std. dev. of 7 runs, 100,000 loops each)
Run time of cumulative sum & range sum:
9.7  s ± 639 ns per loop (mean ± std. dev. of 7 runs, 100,000 loops each)
```

Remember to always use the right tool for the job!

Now, we can compute average values over varying ranges:

```
def range_average(csa, start, end):
    return range_sum_cs(csa, start, end) / (end - start + 1)

a = list(range(10))
csa = cumulative_sum(a)
print("Input array:", a)
print("Average value of elements between indices 2 and 5 (inclusive):",
    range_average(csa, 2, 5))
```

```
Input array: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
Average value of elements between indices 2 and 5 (inclusive): 3.5
```

Integral image (summed-area table, cumulative sum in 2D)

In computer vision, a useful extension of the cumulative sum to 2 dimensions is used. It is called the **integral image** (or summed-area table).

```
matrix = create_empty(11, 10)
fill_matrix_with_randint(matrix, 0, 255)

def integral_image(mat):
    ii = create_empty(len(mat), len(mat[0]))
    for i in range(len(mat)):
        for j in range(len(mat[0])):
            ii[i][j] = mat[i][j]
            if i > 0:
                ii[i][j] += ii[i - 1][j]
            if j > 0:
                ii[i][j] += ii[i][j - 1]
            if i > 0 and j > 0:
                ii[i][j] -= ii[i - 1][j - 1]
    return ii

ii = integral_image(matrix)
formatted_print_matrix(matrix)
formatted_print_matrix(ii)
```

```
[[ 66,  18,  60, 222, 191,   0, 136,  25, 104,  87],
 [ 72, 225, 126,  35, 127, 149, 171, 160, 144, 138],
 [ 46, 166,  79, 129,  64, 250, 250,  42,  76,  32],
 [210, 170, 210, 111,  93, 207, 119,  30,  13, 213],
 [ 86, 197, 189, 188, 181,  60, 146,  60, 148, 192],
 [213, 139, 172,  14,  41,  51,  68,  80, 176, 229],
 [204,  29, 199, 207,  72,  10,  55,  34,  74, 185],
 [  9,  88,  78, 141, 254, 156, 116,  33, 164, 137],
 [255,  96, 163,  17,  59, 153, 209, 208, 169, 106],
 [212, 255, 149, 248, 250, 183, 122,  36,  50, 143],
 [253, 191,  28, 135,  26, 138, 252, 207, 240, 118]]
[[   66,    84,   144,   366,   557,   557,   693,   718,   822,   909],
 [  138,   381,   567,   824,  1142,  1291,  1598,  1783,  2031,  2256],
 [  184,   593,   858,  1244,  1626,  2025,  2582,  2809,  3133,  3390],
 [  394,   973,  1448,  1945,  2420,  3026,  3702,  3959,  4296,  4766],
 [  480,  1256,  1920,  2605,  3261,  3927,  4749,  5066,  5551,  6213],
 [  693,  1608,  2444,  3143,  3840,  4557,  5447,  5844,  6505,  7396],
 [  897,  1841,  2876,  3782,  4551,  5278,  6223,  6654,  7389,  8465],
 [  906,  1938,  3051,  4098,  5121,  6004,  7065,  7529,  8428,  9641],
 [ 1161,  2289,  3565,  4629,  5711,  6747,  8017,  8689,  9757, 11076],
 [ 1373,  2756,  4181,  5493,  6825,  8044,  9436, 10144, 11262, 12724],
 [ 1626,  3200,  4653,  6100,  7458,  8815, 10459, 11374, 12732, 14312]]
```

Now, to compute the sum of a sub-matrix, the same principle as in the cumulative "range sum" is applied. The "only" difference is that now we are in 2D. Therefore, the computation is slightly more complicated. To compute the sum over the "area" (sub-matrix) demarked by the indices (a, b) and (c, d), we need to compute: $ii[c][d] - ii[a-1][d] - ii[c][b-1] + ii[a-1][b-1]$. Let's visualize the integral image and this computation. Let us consider the problem to be defined as follows:

```
top_left = (5, 3)  # a, b
bottom_right = (7, 6)  # c, d
```

Let's break it down: We want to compute the sum of the gray area - area of interest (AOI) in Figure 1. To do that, we need to take the "total sum" (delimited by red rectangle in Figure 1) of the area from $[0, 0]$ to $[c, d]$. That is, the sum from the origin of the image to the bottom_right corner of the AOI. This sum has the value at $ii[c][d]$. Then, we subtract the two areas from origin to the top-right corner of the AOI (delimited by blue rectangle in Figure 1) and the area from the origin to the bottom-left corner of the AOI (delimited by green rectangle in Figure 1). The sums of these areas are located at $ii[a-1][d]$ and $ii[c][b-1]$. This way, we subtracted twice the sum of the area from the origin to just above the top-left corner of the AOI. We need to add it once back-in. Therefore, the last step is to add the sum of this area (delimited by orange rectangle in Figure 1) that is located at $ii[a-1][b-1]$. To recap, the full equation is:

$$area\_sum[[a, b], [c, d]] = ii[c][d] - ii[a-1][d] - ii[c][b-1] + ii[a-1][b-1]$$

Specifically, in our case:

$$area\_sum[[5, 3], [7, 6]] = ii[7][6] - ii[4][6] - ii[7][3] + ii[4][3]$$

general term	current case term	color in Figure 1
$+ii[c][d]$	$+ii[7][6]$	red
$-ii[a-1][d]$	$-ii[4][6]$	blue
$-ii[c][b-1]$	$-ii[7][3]$	green
$+ii[a-1][b-1]$	$+ii[4][3]$	orange

	0	1	2	3	4	5	6	7	8	9
0	66	84	144	366	557	557	693	718	822	909
1	138	381	567	824	1142	1291	1598	1783	2031	2256
2	184	593	858	1244	1626	2025	2582	2809	3133	3390
3	394	973	1448	1945	2420	3026	3702	3959	4296	4766
4	480	1256	1920	2605	3261	3927	4749	5066	5551	6213
5	693	1608	2444	3143	3840	4557	5447	5844	6505	7396
6	897	1841	2876	3782	4551	5278	6223	6654	7389	8465
7	906	1938	3051	4098	5121	6004	7065	7529	8428	9641
8	1161	2289	3565	4629	5711	6747	8017	8689	9757	11076
9	1373	2756	4181	5493	6825	8044	9436	10144	11262	12724
10	1626	3200	4653	6100	7458	8815	10459	11374	12732	14312

Figure 1: Integral image with colored regions.

Let's first compute the sum "manually":

```
Summed elements:
[5, 3] + [5, 4] + [5, 5] + [5, 6] + [6, 3] + [6, 4] + [6, 5] + [6, 6] + [7, 3] +
 [7, 4] + [7, 5] + [7, 6]
Summed values:
14 + 41 + 51 + 68 + 207 + 72 + 10 + 55 + 141 + 254 + 156 + 116
Sum ("manual" approach): 1185
```

Now, let's compute the sum from the integral image:

```
sum_value = ii[c][d] - ii[a-1][d] - ii[c][b-1] + ii[a-1][b-1]

print("Sum (integral image approach):\n\t"
    f"{ii[c][d]} - {ii[a-1][d]} - {ii[c][b-1]} + {ii[a-1][b-1]}"
    f" = {sum_value}")
```

```
Sum (integral image approach):
    7065 - 4749 - 3051 + 1920 = 1185
```