

hw0_code

January 10, 2026

1 2

1.1 a

```
[3]: import numpy as np

# matrix A, 5 by 4, values 1 to 20 left to right, top to bottom
A = np.arange(1, 21).reshape(5, 4)
print("Matrix A:")
print(A)

# matrix B, 4 by 3, values 1 to 12 top to bottom, left to right
B = np.arange(1, 13).reshape(3, 4).T
print("Matrix B:")
print(B)

def matrix_multiply(A, B):
    return A @ B

print("Matrix Product AB:")
print(matrix_multiply(A, B))
```

Matrix A:

```
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]
 [13 14 15 16]
 [17 18 19 20]]
```

Matrix B:

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

Matrix Product AB:

```
[[ 30  70 110]
 [ 70 174 278]
 [110 278 446]
 [150 382 614]
 [190 486 782]]
```

1.2 b

```
[4]: import scipy.sparse as sp

# worst time complexity of multiplying  $n \times n$  dense matrix with dense vector  $\mathbb{R}^n$ 
# is  $O(n^2)$  because we need to compute  $n$  dot products, each taking  $O(n)$  time.

# worst time complexity of multiplying  $n \times n$  sparse matrix with dense vector  $\mathbb{R}^n$ ,
# where the sparse matrix has  $\text{nnz}(A)$  non-zero entries, is  $O(\text{nnz}(A))$  because we
# only need to compute dot products for the non-zero entries.

# Create sparse matrix  $A$  of size  $(n-1) \times n$  such that for any  $x$  in  $\mathbb{R}^n$ ,
#  $Ax$  gives the vector of differences between consecutive elements of  $x$ .
def p2b(n):
    row_indices = np.arange(n - 1)
    col_indices = np.arange(n - 1)
    data = np.ones(n - 1)

    row_indices = np.concatenate([row_indices, np.arange(n - 1)])
    col_indices = np.concatenate([col_indices, np.arange(1, n)])
    data = np.concatenate([data, -data])

    A = sp.coo_matrix((data, (row_indices, col_indices)), shape=(n - 1, n))
    return A.toarray()

n = 5
A_sparse = p2b(n)
print(f"Sparse matrix A of size ({n-1}, {n}):")
print(A_sparse)

# random vector  $x$  of integers in  $\mathbb{R}^n$ 
x = np.random.randint(1, 10, size=n)
print("Vector x:")
print(x)
print("Ax:")
print(A_sparse @ x)
```

Sparse matrix A of size (4, 5):

```
[[ 1. -1.  0.  0.  0.]
 [ 0.  1. -1.  0.  0.]
 [ 0.  0.  1. -1.  0.]
 [ 0.  0.  0.  1. -1.]]
```

Vector x:

```
[9 5 7 1 8]
```

Ax:

```
[ 4. -2.  6. -7.]
```

1.3 c

```
[5]: # read in text from 'data_example.txt' line by line

with open('data_example.txt', 'r') as f:
    lines = f.readlines()

# naive unique word count by just splitting on spaces and keeping track of
# seen words and their count using a dictionary

words = {}
for line in lines:
    for word in line.strip().split():
        words[word] = words.get(word, 0) + 1
print(f"Naive unique word count: {len(words)}")

# the time complexity of the naive approach is  $O(n)$ , where  $n$  is the total number
# of words in the text, because we need to process each word once.
# the space complexity is  $O(2m)$ , where  $m$  is the number of unique words,
    ↪ multiplied
# by 2 because for each unique word, we store the word itself and its count.
```

Naive unique word count: 1394

1.4 d

```
[6]: # naive Fibonacci
def fib_naive(n):
    if n <= 1:
        return n
    return fib_naive(n - 1) + fib_naive(n - 2)

# recursive Fibonacci w/ bookkeeping using a global dictionary
fib_dict = {0: 0, 1: 1}

def fib(n):
    if n not in fib_dict:
        fib_dict[n] = fib(n - 1) + fib(n - 2)
    return fib_dict[n]

# time complexity of book keeping Fibonacci is  $O(n)$  because each Fibonacci
    ↪ number
# from 0 to  $n$  is computed only once and stored in the dictionary.

print("Naive Fibonacci of 10:", fib_naive(10))
print("Bookkeeping Fibonacci of 10:", fib(10))
```

Naive Fibonacci of 10: 55

Bookkeeping Fibonacci of 10: 55