Contents

1 Introduction

The main objective of this assignment is to learn about algorithm complexity.

2 Programming problems

2.1 Remove Duplicates – rdmup

The code we use is:

```
# removes duplicates from data.
# This function keeps the last occurence of each element
# and preserves order.
# So rmdup([1,2,3,2,1,4,2]) should return [3,1,4,2]
def rmdup(data):
    my_dict = {}
    for i in range(len(data)):
        my_dict[data[i]] = i
    res = []
    seen = set()
    for i in range(len(data)):
        if i == my_dict[data[i]] and data[i] not in seen:
            seen.add(data[i])
            res.append(data[i])
    return res
```

We analyze the runtime in Figure 1. As is shown at the bottom of Figure 1, the runtime is **O(N)**, where N is the length of input data.

Function	How many times?	How long?
def rmdup(data):		
my_dict = {}	O(1)	O(1)
for i in range(len(data)):	O(N)	O(1)
my_dict[data[i]] = i	O(N)	O(1)
res = []	O(1)	O(1)
seen = set()	O(1)	O(1)
for i in range(len(data)):	O(N)	O(1)
if i == my_dict[data[i]] and data[i] not in seen:	O(N)	O(1)
seen.add(data[i])	O(N)	O(1)
res.append(data[i])	O(N)	O(1)
return res	O(1)	O(1)

Runtime is O(1+N+N+1+1+N+N+N+N+1) = O(N)

Figure 1: Runtime analysis of **rdmup**

We now measure the runtime for various data sizes. Specifically, we measure the performance of **rdmup** with data sizes from 4096 (4K) to 4194304 (4M) elements, in steps of multiplication by 2, on three types of random data:

1. **Many duplicates**
   Each data item is chosen by `random.randrange(0, size/2048)`

2. **Moderate duplication**
   Each data item is chosen by `random.randrange(0, size/16)`

3. **Rare duplication**
   Each data item is chosen by `random.randrange(0, size*4)`

The results are shown in Table 1. Runtime is shown in milliseconds, and we round those floats to integers for us to see more easily.

We note that to facilitate data analysis, we add three columns at last to show the tendency of runtime growth. Specifically, we use the runtime for data 4096 as the base time to indicate the time growth.

Table 1: Actual runtime for **rdmup** in reality

runtime (ms) / number of elements	Duplication			log2($\frac{\text{new time}}{\text{base time}}$)		
	many	moderate	rare	many	moderate	rare
4096 (base case)	1	1	2	0	0	0
8192	2	2	6	1	1	1
16384	3	4	10	2	2	2
32768	7	9	26	3	3	3
65536	13	19	45	4	4	4
131072	26	40	94	5	5	5
262144	54	88	214	6	7	6
524288	108	200	479	7	8	7
1048576	250	553	1045	8	9	9
2097152	501	1376	2231	9	10	10
4194304	1073	3494	4721	10	12	11

We observe that while the data size grows to the power of two, the corresponding runtime also grows to the power of two, which is in linear relationship with the growth of data size.

This practice validates our theory analysis: the time complexity is indeed O(N).

Moreover, we note that the the less the duplication, the more drastically the runtime grows (that is to say, the bigger the constant coefficient of N in O(N)) as the data size grows. We think this might be due to the following reasons:

- The less the duplicates, the more numbers Python has to insert to the list with **append** method, which takes time.
- The less the duplicates, the more numbers in **seen**, which makes the **in** method deal with bigger set, which takes more time.

2.2 Matrix Multiplication - matrixmul

The code we use is:

```
# This function takes 2 matricies (as lists of lists)
# and performs matrix multiplication on them.
```

```python
# Note: you may not use any matrix multiplication libraries.
# You need to do the multiplication yourself.
# For example, if you have
#     a=[[1,2,3],
#        [4,5,6],
#        [7,8,9],
#        [4,0,7]]
#     b=[[1,2],
#        [3,4],
#        [5,6]]
#  Then a has 4 rows and 3 columns.
#  b has 3 rows and 2 columns.
#  Multiplying a * b results in a 4 row, 2 column matrix:
#  [[22, 28],
#   [49, 64],
#   [76, 100],
#   [39, 50]]
def matrix_mul(a, b):
    numRows_a = len(a)
    numCols_a = len(a[0])
    numRows_b = len(b)
    numCols_b = len(b[0])

    ans = [[0 for i in range(numCols_b)] for j in range(numRows_a)]

    for row in range(numRows_a):
        for col_b in range(numCols_b):
            for col_a in range(numCols_a):
                ans[row][col_b] += a[row][col_a] * b[col_a][col_b]

    return ans
```

We analyze the runtime in Figure 2. As is shown at the bottom of Figure 2, the runtime is $O(N^2M)$, where N is the number of rows of matrix a, and M the number of columns.

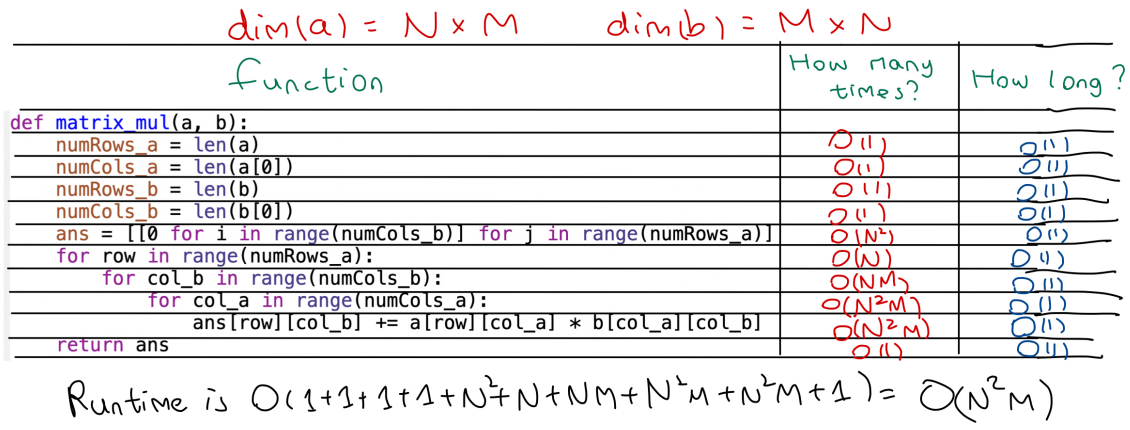

Figure 2: Runtime analysis of **matrixmul**

We now measure the runtime for various data sizes. Specifically, we measure the performance of **matrixmul** with data sizes from 4 to 512 elements, in steps of multiplication by 2, on three types of random data in the instruction.

The results are shown in Table 2. Runtime is shown in microseconds (10^{-6}), and we round those floats to integers for us to see more easily.

We note that to facilitate data analysis, we add three columns at last to show the tendency of runtime growth. Specifically, we use the runtime for data size 4 as the base time to indicate the time growth.

Table 2: Actual runtime for **matrixmul** in reality

runtime (10^{-6}s) / num elements	Matrix Type			log2($\frac{\text{new time}}{\text{base time}}$)		
	many row	square	less row	many row	square	less row
4 (base case)	4	24	12	0	0	0
8	12	155	98	1	3	3
16	103	1221	315	4	6	5
32	509	7776	2157	6	8	8
64	3800	61621	15590	9	11	10
128	28336	500348	129033	12	14	13
256	246935	3932992	915544	15	17	16
512	1893985	31882395	7859280	18	20	19

We observe that while the data size grows to the power of two, the corresponding runtime grows as our theory analysis: the time complexity $O(N^2M)$, where $M = N//4$ in the case of many rows, and $M = N * 4$ in the case of less rows.

Moreover, we note that the many row and column's type is way faster than the many column and less row's type, we guess it's mainly because that number arrays are stored consecutively in address, so take numbers in the first style will make the code fetch data faster.

2.3 Matching length sub string

The code we use is:

```
# Write a function, which when iven one string (s) and two characters
# (c1 and c2), computes all pairings of contiguous ranges of c1s
# and c2s that have the same length.  Your function should return
# a set of three-tuples.  Each element of the set should be
# (c1 start index, c2 start index, length)
#
# Note that s may contain other characters besides c1 and c2.
# Example:
#  s = abcabbaacabaabbbb
#      0123456789011111  <- indices for ease of looking
#                1123456
#  c1 = a
#  c2 = b
#  Observe that there are the following contiguous ranges of 'a's (c1)
#  Length 1: starting at 0, 3, 9
#  Length 2: starting at 6, 11
#  And the following contiguous ranges of 'b's (c2)
#  Length 1: starting at 1, 10
#  Length 2: starting at 4
#  Length 4: starting at 13
```

```
#  So the answer would be
#  { (0, 1, 1), (0, 10, 1), (3, 1, 1), (3, 10, 1), (9, 1, 1), (9, 10, 1),
#    (6, 4, 2), (11, 4, 2)}
#  Note that the length 4 range of 'b's does not appear as there are no
#  Length 4 runs of 'a's.

def matching_length_sub_strs(s, c1, c2):
    def find_sequences(s, c):
        i = 0
        while i < len(s):
            if s[i] == c:
                start = i
                while i < len(s) and s[i] == c:
                    i += 1
                yield (start, i - start)
            i += 1

    matches = set()
    c1_sequences = list(find_sequences(s, c1))
    c2_sequences = list(find_sequences(s, c2))

    for c1_start, c1_length in c1_sequences:
        for c2_start, c2_length in c2_sequences:
            if c1_length == c2_length:
                matches.add((c1_start, c2_start, c1_length))

    return matches


# Makes a random string of length n
# The string is mostly comprised of 'a' and 'b'
# So you should use c1='a' and c2='b' when
# you use this with matching_length_sub_strs
def rndstr(n):
    def rndchr():
        x=random.randrange(7)
        if x==0:
            return chr(random.randrange(26)+ord('A'))
        if x<=3:
            return 'a'
        return 'b'
    ans=[rndchr() for i in range(n)]
    return "".join(ans)
```

We analyze the runtime in Figure 3. As is shown at the bottom of Figure 3, the runtime is $O(N^2)$, where N is the length of string s.

We now measure the runtime for various data sizes.

The results are shown in Table 3. Runtime is shown in milliseconds, and we round those floats to integers for us to see more easily.

We note that to facilitate data analysis, we add three columns at last to show the tendency of runtime growth. Specifically, we use the runtime for data 512 as the base time to indicate the time growth.

Analysis:

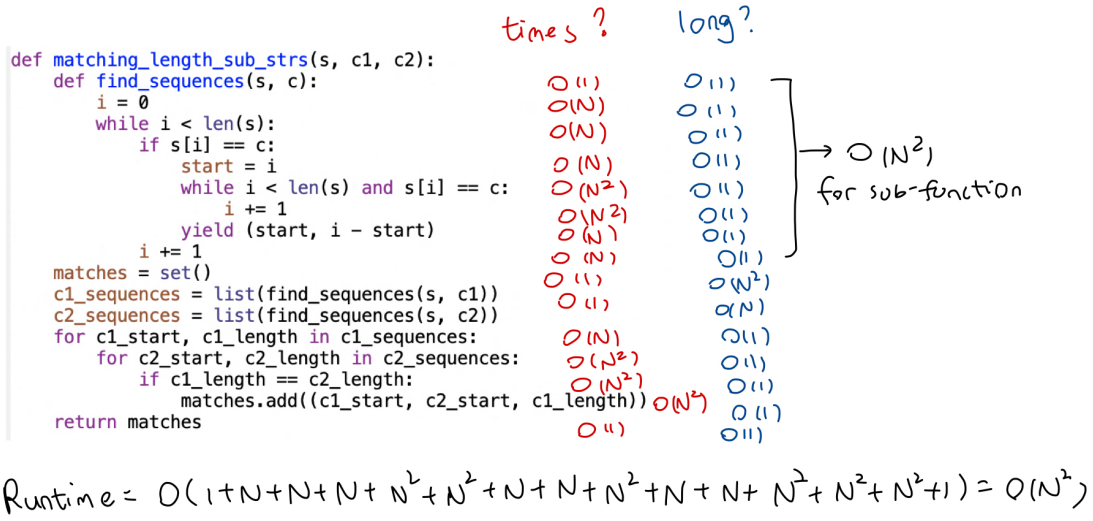

Figure 3: Runtime analysis of **eqsubstr**

Table 3: Actual runtime for **eqsubstr** in reality

Number of elements	Time (ms)	log2(time/base)
512 (base case)	2	0
1024	9	2
2048	47	4
4096	160	6
8192	771	8
16384	3199	10

3 Time Complexity True or False

By mathematics, we know that $\log(n^2)$ is just $2\log(n)$, so they grow at the same rate, it's just that the former one is scaled a constant coefficient of 2 of the latter one.

For the following statements:

(a) $2n^3 - 8n^2 + 32n + 9 \in O(n^3)$ This is **True** because the upper bound here is just the term with the highest degree, which is indeed n^3.

(b) $2n^3 - 8n^2 + 32n + 9 \in \Omega(n^3)$ This is **True** because the Omega describes a lower bound and n^3 is the term with the highest degree that will dominate the growth rate for large ns.

(c) $n^p \in O(e^n)$, where $p \in \mathbb{R}$ and $p \geq 0$ This is **True** because exponential functions grows explosively and is proved to grow way faster than polynomial functions for all positive real numbers p, so the polynomial one will always be bounded by the exponential one if n is large enough.

(d) $e^n \in O(n^p)$, where $p \in \mathbb{R}$ and $p \geq 0$ This is **False**, because after the point N where $N = p * ln(N)$, exponential function grows strictly faster than exponential ones, for

any real positive p.

(e) $\sqrt{n} \in O(1)$ This is **False** as $\sqrt{n}$ is not independent of n, it grows as n grows.

4 Conclusion

In this assignment, we learned about algorithm complexity.