ECE 590 HW 5

Chelsea (Jiyuan) Lyu

Nov. 7 2023

1 Problem 1

Here is my function which removes duplicates from the data:

```
1 def rmdup(data):
2     unique = {}
3     last_occur = []
4
5     for element in reversed(data):
6         if element not in unique:
7             last_occur.append(element)
8             unique[element] = True
9
10     last_occur.reverse()
11    return last_occur
```

Here is the O of runtime for each line.
N is the input data size.

Line	How Many Times?	How Long?
1	$O(1)$	$O(1)$
2	$O(1)$	$O(1)$
3	$O(1)$	$O(1)$
4		
5	$O(N)$	$O(1)$
6	$O(N)$	$O(1)$
7	$O(N)$	$O(1)$
8	$O(N)$	$O(1)$
9		
10	$O(1)$	$O(N)$
11	$O(1)$	$O(1)$

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

Here I measure the runtime for various data sizes three times and calculate the average. The average runtime is shown below in Graph 1 (Figure 1) and Table 2 (Figure 2).

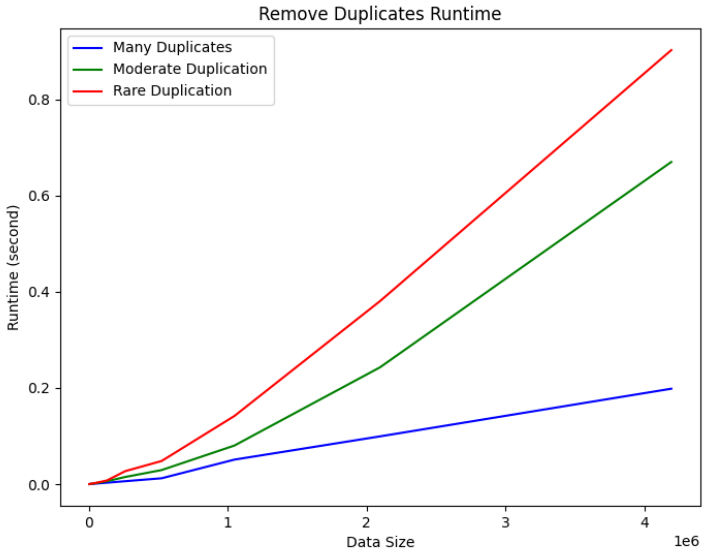

Figure 1: Graph 1

	Many Duplicates (s)	Moderate Duplication (s)	Rare Duplication (s)
4096	0.00017062822977701822	0.00012095769246419 27	0.00024040540059407553
8192	0.00029118855794270 83	0.00039402643839518 23	0.00043511390686035156
16384	0.00046539306640625	0.00070500373840332 03	0.00083533922831217 44
32768	0.00083295504252115 88	0.00135517120361328 12	0.00166479746500651 05
65536	0.00156402587890625	0.00281023979187011 7	0.00355609258015950 54
131072	0.00307909647623697 9	0.00611027081807454 4	0.00770537058512369 8
262144	0.00612998008728027 34	0.01491085688273112	0.02699017524719238 3
524288	0.01220202445983886 7	0.02926731109619140 6	0.04810476303100586
1048576	0.05100687344868978	0.08023039499918 62	0.14166895548502603
2097152	0.09932255744934082	0.24282511075337 73	0.38041130701700 85
4194304	0.19834701220194498	0.66958308219909 67	0.90206511815389

Figure 2: Table 1: Runtime (seconds) for three categories

According to Graph 1 and Table 1, with the data size increase, the runtime increase linear, which match my theory runtime $O(N)$.

However, there are different behaviors for different categories of input. When there are fewer duplicates in the input, the runtimes increase faster when data size increases. This might be due to fewer duplicates leading to more elements needing to be stored in the output array.

2 Problem 2

Here is my function which multiplies the matrix:

```
1 def matrix_mul(a,b):
2     c = []
3     for i in range(len(a)):
4         row = []
5         for j in range(len(b[0])):
6             element = 0
7             for k in range(len(b)):
8                 element += a[i][k] * b[k][j]
9             row.append(element)
10        c.append(row)
11    return c
```

Here is the O of runtime for each line.
Let the input matrix a be $X * Y$ and the input matrix b be $Y * Z$.

Line	How Many Times?	How Long?
1	$O(1)$	$O(1)$
2	$O(1)$	$O(1)$
3	$O(X)$	$O(1)$
4	$O(X)$	$O(1)$
5	$O(X * Y)$	$O(1)$
6	$O(X * Y)$	$O(1)$
7	$O(X * Y * Z)$	$O(1)$
8	$O(X * Y * Z)$	$O(1)$
9	$O(X * Y)$	$O(1)$
10	$O(X)$	$O(1)$
11	$O(1)$	$O(1)$

Runtime is $O(1 + 1 + X + X + X * Y + X * Y + X * Y * Z + X * Y * Z + X * Y + X + 1) = O(X * Y * Z)$

Since three categories of the shape of the input matrix are $(N * 4) \times N * N \times (N/4)$ for many rows by few columns, $N \times N * N \times N$ for squares, and $(N/4) \times N * N \times (N * 4)$ for few rows by many columns, the runtime $O(X * Y * Z) = N^3$ for three shapes.

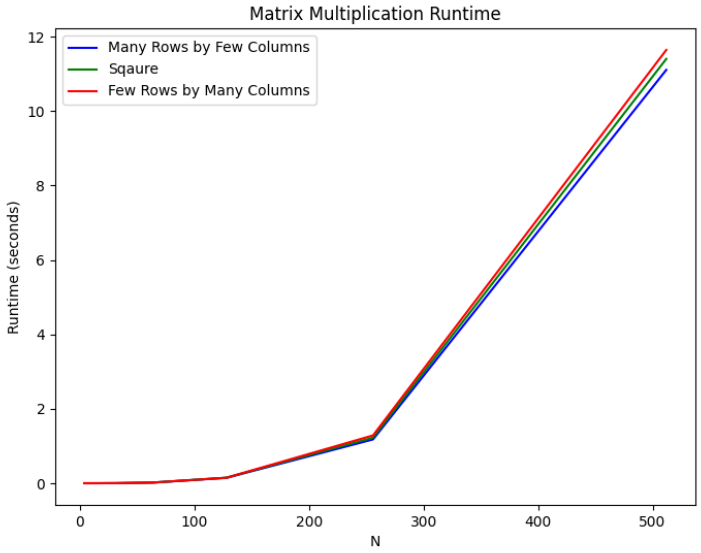

Figure 3: Graph 2

	Many Rows by Few Columns (s)	Sqaure (s)	Few Rows by Many Columns (s)
4	1.5099843343098959e-05	8.742014567057291e-06	7.947285970052084e-06
8	5.2928924560546875e-05	4.760424296061198e-05	4.951159159342448e-05
16	0.0003352959950764974	0.0003286202748616536 3	0.0003146330515543 62
32	0.0024193922678629556	0.002315998077392578	0.002392371495564779
64	0.018148501714070637	0.01879294713338216	0.018306811650594074
128	0.1446088949839274	0.14953200022379556	0.14765691757202148
256	1.1776947180430095	1.2255309422810872	1.2862345377604167
512	11.11037834485372	11.405304431915283	11.646107832590738

Figure 4: Table 2: Runtime (seconds) for three categories

Here I measure the runtime for various data sizes three times and calculate the average. The average runtime is shown above in Graph 2 (Figure 3) and Table 2 (Figure 4).

According to Graph 2 and Table 2, with the data size increase, the runtime increases match my theory runtime $O(N^3)$. The lines do not look like curves due to there are not enough different data sizes and runtimes.

The behaviors for different categories of matrix shapes are quite similar. But we can observe that when there are fewer rows, it'll take slightly more runtime, which could be due to the ignoring part in the former calculation.

3 Problem 3

Here is my function which matches the length of the substrings:

```
1 def matching_length_sub_strs(s, c1, c2):
2     c1_dict = {}
3     c2_dict = {}
4     ans = set()
5
6     count1 = 0
7     count2 = 0
8     for i in range(len(s)):
9         if s[i] == c1:
10            count1 = count1 + 1
11        if s[i] != c1 or i == len(s)-1:
12            index1 = i - count1
13            if i == len(s)-1:
14                index1 = i - count1 + 1
15            if count1 != 0:
16                if count1 in c1_dict:
17                    c1_dict[count1].append(index1)
18                else:
19                    c1_dict[count1] = [index1]
20                count1 = 0
21
22        if s[i] == c2:
23            count2 = count2 + 1
24        if s[i] != c2 or i == len(s)-1:
25            index2 = i - count2
26            if i == len(s)-1:
27                index2 = i - count2 + 1
28            if count2 != 0:
29                if count2 in c2_dict:
30                    c2_dict[count2].append(index2)
31                else:
32                    c2_dict[count2] = [index2]
33                count2 = 0
34    for key in c1_dict:
35        if key in c2_dict:
36            for c1_value in c1_dict[key]:
37                for c2_value in c2_dict[key]:
38                    tuple = (c1_value, c2_value, int(key))
39                    ans.add(tuple)
40    return ans
```

Here is the O of runtime for each line.
N is the length of the input string.

Line	How Many Times?	How Long?
1	$O(1)$	$O(1)$
2	$O(1)$	$O(1)$
3	$O(1)$	$O(1)$
4	$O(1)$	$O(1)$
5	$O(1)$	$O(1)$
6	$O(1)$	$O(1)$
7	$O(1)$	$O(1)$
8	$O(N)$	$O(1)$
9	$O(N)$	$O(1)$
10	$O(N)$	$O(1)$
11	$O(N)$	$O(1)$
12	$O(N)$	$O(1)$
13	$O(N)$	$O(1)$
14	$O(N)$	$O(1)$
15	$O(N)$	$O(1)$
16	$O(N)$	$O(1)$
17	$O(N)$	$O(1)$
18	$O(N)$	$O(1)$
19	$O(N)$	$O(1)$
20	$O(N)$	$O(1)$
21	$O(N)$	$O(1)$
22	$O(N)$	$O(1)$
23	$O(N)$	$O(1)$
24	$O(N)$	$O(1)$
25	$O(N)$	$O(1)$
26	$O(N)$	$O(1)$
27	$O(N)$	$O(1)$
28	$O(N)$	$O(1)$
29	$O(N)$	$O(1)$
30	$O(N)$	$O(1)$
31	$O(N)$	$O(1)$
32	$O(N)$	$O(1)$
33	$O(N)$	$O(1)$
34	$O(len(c1\_dict))$	$O(1)$
35	$O(len(c1\_dict))$	$O(1)$
36	$O(len(c1\_dict) * len(c1\_dict[key]))$	$O(1)$
37	$O(len(c1\_dict) * (len(c1\_dict[key]) + len(c2\_dict[key])))$	$O(1)$
38	$O(len(c1\_dict) * (len(c1\_dict[key]) + len(c2\_dict[key])))$	$O(1)$
39	$O(len(c1\_dict) * (len(c1\_dict[key]) + len(c2\_dict[key])))$	$O(1)$
40	$O(1)$	$O(1)$

According to the table of Big O above, since the maximum of $len(c1\_dict)$ is N, assume that $len(c1\_dict[key]) + len(c2\_dict[key]))$ is M, the runtime is

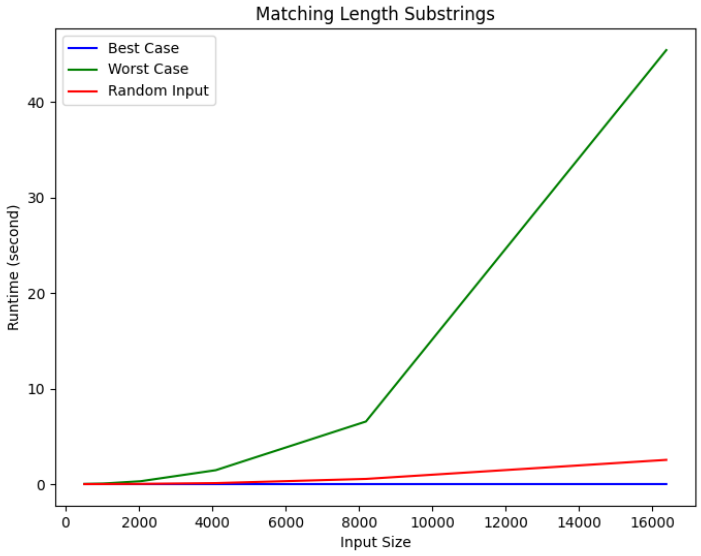

Figure 5: Graph 3

$O(N * M)$.
Here are three cases of the input:

1. The best case is that the input string only has $c1 = a$ and $c2 = b$. Here I assume there only exists $c1 = a$ in the string, so the regular expression of best case is $s = (a^*b^*) \bigcup (b^*a^*)$, and N is the length of the string.

2. The worst case is that the input string does not have a contiguous character $c1 = a$ or $c2 = b$, so the regular expression of the worst case is $s = (ab)^*$.

3. The random input is a string of length N which is approximately 3/7 "a"s, 3/7 "b"s, and 1/7 a capital letter.

I measure the runtime for various data sizes three times and calculate the average. The average runtime is shown above in Graph 3 (Figure 5) and Table 3 (Figure 6).

According to Graph 3 and Table 3, with the data size increase by the polynomial, the runtimes increase matching my theory runtime $O(N * M)$.

	Best Case(s)	Worst Case(s)	Random Input(s)
512	0.00017340977986653647	0.016974608103434246	0.0017744700113932292
1024	0.0003096262613932292	0.05750107765197754	0.006915569305419922
2048	0.000629266103108724	0.30259402592976886	0.022995710372924805
4096	0.0012320677439371746	1.4569799900054932	0.10441001256306966
8192	0.002499103546142578	6.555830955505371	0.5429200331370035
16384	0.0051546891530354815	45.46240504582723	2.537724335988363

Figure 6: Table 3: Runtime (seconds) for three categories

4 Problem 4

(a) T

(b) T

(c) T

(d) F

(e) F