DS- II Assignment

Introduction:

Ethereum (Cryptocurrency) is open-source service that works on blockchain technology which offers smart contracts and transactions securely without need of any third-party application. Our assignment aims in analyzing the data of crypto transactions over a month of data. We perform various operations on data like sorting, searching and grouping of the data. We analyzed the data on 10 files using python pandas package, an efficient way to analyze the data.

Required Modules:

- Pandas
- Numpy
- Os
- Glob
- CSV

Command To Run Before Running The Python File To Install The Module:

pip install pandas

Commands To Execute The File:

>>> pip install pandas
>>> python file.py

Project Report:

Note : This project report has the following format:

- Code snippet is shown as the reference.
- Analysis of every line on our approach.
- Identifying major data structure.
- Finding time complexity for each line

*- Dominant time complexity from the graph and theoretical stand point.

*- Output of the question with the Execution time.

First Question:

```
### First Question ###

final_df = tr_data.drop_duplicates(subset=['tx_hash']) # blk no is dropped duplicates dataframe
dff_hash = final_df.groupby(["block_number","block_time"])
dff = dff_hash.gas.sum().reset_index()
dff = dff.sort_values(by = "gas", kind='heapsort')
dff.rename(columns={'gas': 'block_gas'}, inplace=True) # dff = first q
# dff.to_csv('/Users/srikaramara/Desktop/first_q.csv')
print(dff)
print("--- First Question Ran in %s seconds ---" % (time.time() - start_time))
```

- **Line - 1 :**

Here we are dropping duplicate transactions

Method Signature:

df.drop_duplicates(subset:union[Hashable , sequence[hash],NoneType]= None;)

Parameters used:

Subset : column label or sequence of labels

- In line 1 we gave one column as input for searching that particular column in a data frame. Let's say we have 'm 'columns then in worst case our given subset can be mth column. M will be worst case time complexity.
- Now after getting particular column or column from signature if passed values is single value on that value is used if it is collection or sequence of values it takes as order set for hash-able.
- Objects takes as ordered set for hash-able objects.
- In our case **tx_hash** is string which is **hashable**, therefore internally these values are stored in (keys, value) pairs where keys are unique which are rows, value is a column name to process all rows it need to check whole column which is 'N'.
- N is number of rows with duplicates.

So the time complexity of line 1 is "m+N"

Key Data structure- "Dictionary or Hash map"

(Reasons is due to its unique property of storing unique keys)

- **Line 2 and 3:**

Here we are grouping by "block_number","block_time" cloumns.

- We used groupby function takes all rows of given columns initially checks every column names. It iteratively checks every column name if it is equal to given one then takes all rows. If multiple columns are given then it takes common rows.
- Let total number of columns be m, our required column in worse case can be at m'th position. Time complexity involved will be m.
- But inside columns as rows are taken, worst case will be n.
- n is number of rows after dropping duplicates.
- **Therefore m+n time complexity for group by.**
- **Next**- reset index is applied which adds a column and gives S no from 0 to n-1
- **Time complexity for Reset_ index() – n**
- **Total time complexity for Line 2 and 3:**

- **Time complexity of groupby()+ time complexity of sum() + time complexity of reset_index()**
- **m+n+n+n = m+3n**

- **Line 4:**

Now after grouping we apply "heapsort" which sorts gas value in ascending order

Key data_structure: Heapsort

Reason : doesn't have any limitations like count sort, it is inplace algorithm which is better than merge sort and have efficient time complexity of nlogn in worst cases scenarios.

Disadvantage -it is unstable algorithm meaning changes indices of original data.

Time complexity – nlog(n)

- **Line 5:**

This function is used to rename given column name in a iteration. Let there be k columns after group-by.

In worst case scenario given column rename can be at k'th position

Time complexity is K

CONCLUSION:-

Total code time complexity : m+N +m+3n +nlgn+k

2m+4n+nlgn+k

As in our data no of rows are very more than columns

n>>m

dominant term will be **nlgn.**

O(nlgn)

All data structures used from line 1 are:- Hashset, Heap sort.

Dominant data structure:-heap sort.

Output with runtime:

Asymptotic time = O(n log(n)) = O(14,033,192*log(14,033,192))

Actual Execution Time = 164.5035

```
--------------------------------------------------------------------------------------------------------------
---------------
        block_number               block_time  block_gas
63483       11032004  2020-10-11 04:25:38 UTC      37195
344184      11318874  2020-11-24 04:40:04 UTC      38853
133009      11103134  2020-10-22 01:50:50 UTC      39139
82191       11051169  2020-10-14 02:43:11 UTC      39599
267031      11240044  2020-11-12 02:14:30 UTC      39686
...              ...                        ...        ...
211088      11182894  2020-11-03 07:38:25 UTC   60527433
210985      11182791  2020-11-03 07:15:56 UTC   62735686
247631      11220248  2020-11-09 01:07:55 UTC   66670776
214903      11186781  2020-11-03 22:01:44 UTC   68392671
210818      11182620  2020-11-03 06:38:54 UTC   73189043

[387560 rows x 3 columns]
--- First Question Ran in 164.44302988052368 seconds ---
```

Second question

```
### Second Question ###

start_time = time.time()
tmp_1 = dff_hash.size().reset_index(name='transactions')
tmp_1 = tmp_1.sort_values(by = "transactions", kind='heapsort')
print(tmp_1)
print("--- Second Question Ran in %s seconds ---" % (time.time() - start_time))
```

- **Line -1:**
  - Line 1: tmp 1= dff_hash.size().reset_index(name=transactions)
  - Here dff_hash is taken from question 1 as it is already computed and we are adding those count of all blocks.
- **Line 2:**

we applied “heapsort” which sorts transaction values in ascending order

Key data_structure: Heapsort

Reason : doesn’t have any limitations like count sort, it is inplace algorithm which is better than merge sort and have efficient time complexity of nlogn in worst cases scenarios.

Disadvantage -it is unstable algorithm meaning changes indices of original

Time complexity – nlog(n).

Conclusion:

The key data structure in the second question is Heap sort.

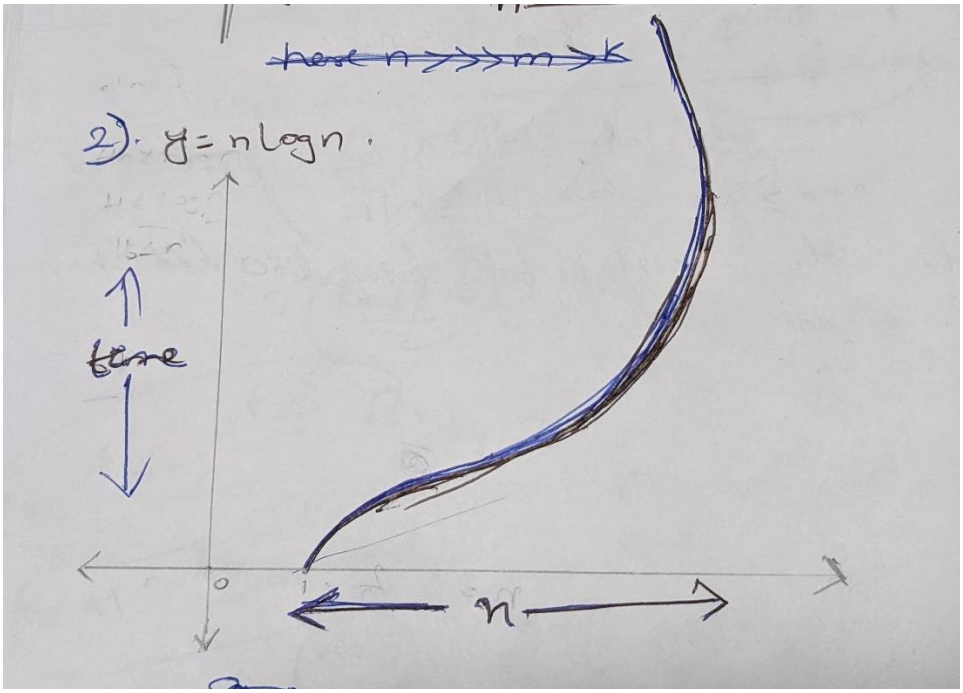
Dominant time complexity is nlog(n)

output with runtime:-

Asymptotic execution time:- O(nlog(n)) = O(387,560*log(387,560))

Actual execution time:- 7.45 seconds, we got the less execution time, because here we used already framed dataframe from first question.

```
                                    transactions
block_number block_time
10997519     2020-10-05 19:03:39 UTC             1
11220525     2020-11-09 02:14:25 UTC             1
11181431     2020-11-03 02:12:07 UTC             1
11036366     2020-10-11 20:15:48 UTC             1
11292306     2020-11-20 02:37:15 UTC             1
...                                            ...
11163138     2020-10-31 06:54:36 UTC           144
11160763     2020-10-30 22:13:25 UTC           144
11044715     2020-10-13 03:01:23 UTC           147
11163137     2020-10-31 06:54:34 UTC           152
11028451     2020-10-10 15:20:08 UTC           153

[387560 rows x 1 columns]
--- Second Question Ran in 7.456465244293213 seconds ---
```



(Since only line2 is dominant part which is heapsort .It's best case and worst case are represented by blue pen and black pen that overlap in graph.)

Third Question

```
### Third Question ###

start_time = time.time()
fff = final_df['total_gas'].subtract(final_df['gas'])
final_df['transaction_fee'] = fff
thir_q = final_df.sort_values(by = "transaction_fee", kind='heapsort')
print(thir_q[['tx_hash','block_number','block_time','transaction_fee']])
print("--- Third Question Ran in %s seconds ---" % (time.time() - start_time))
```

Line 1 :

Here, for every row, we are subtracting gas from total_gas. As it iterativeley subracts till last row.

That implies the time complexity of line 1 is "n"

Line 3 :

we apply "heapsort" which sorts "transaction_fee" value in ascending order

Key data_structure: Heapsort

Reason : doesn't have any limitations like count sort, it is inplace algorithm which is better than merge sort and have efficient time complexity of nlogn in worst cases scenarios.

Disadvantage -it is unstable algorithm meaning changes indices of original

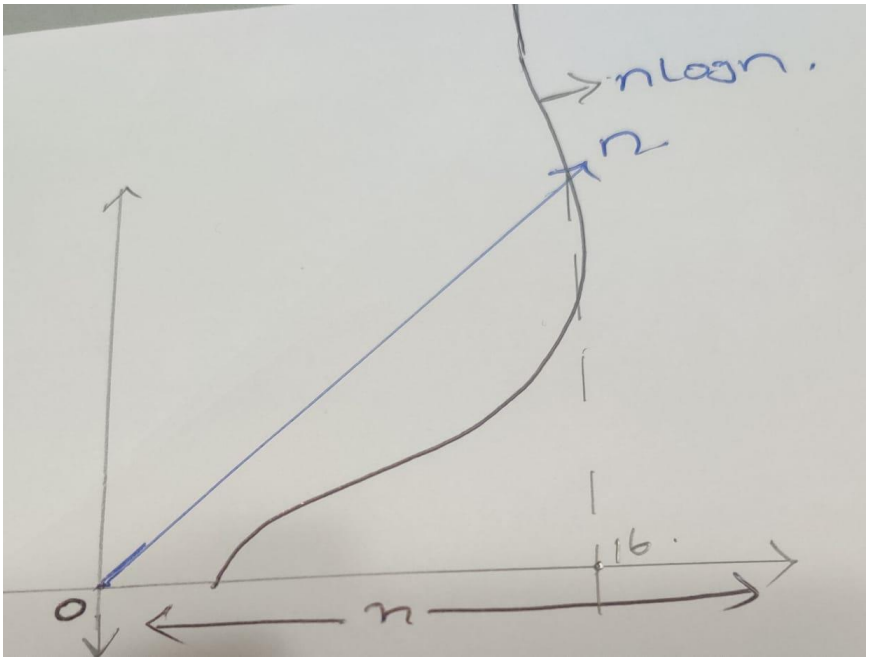
Time complexity of this block of code is nlog(n) + n.

Dominant Data Structure : Heapsort.

Asymptotic execution time:- O(nlog(n)) = O(14,033,192*log(14,033,192))

Actual execution time:- 89.84 seconds

```
                                          tx_hash  block_number              block_time  transaction_fee
11207429  0x914aedb8c7bb96b010ae594d6e0d15f7171f94512e34...      11036135  2020-10-11 19:27:03 UTC        -12427471
6155611   0x262e12790edb9f0855612822f988f9b81689a0aa66c9...      11247640  2020-11-13 06:11:16 UTC        -12417738
2065738   0xf875527f25271663ba38dafa9ea3f15739b621115954...      11343674  2020-11-28 00:15:46 UTC        -12417662
15397228  0x25c0eb7ad8347e5c94b27648af19dca10d4eac36ed44...      11167872  2020-11-01 00:13:04 UTC        -12415905
5709185   0xc8254e71c6949263fd1d331732cc5dc622c0ec200021...      11076627  2020-10-18 00:12:56 UTC        -12411563
...                                                  ...           ...                       ...              ...
12924653  0x10750cb42ec3c9c0f2a1ba484b02a84e6b76a359bd69...      11180376  2020-11-02 22:30:21 UTC         12494554
9533080   0x608e0c5fbba26b9f06ef24f904d4f7ef70b80321e383...      11324781  2020-11-25 02:11:32 UTC         12495477
2514717   0x5ae0047a55bd380448f2a4b8cd8b69a70fc8b39a38a1...      11200825  2020-11-06 01:39:21 UTC         12496683
15467375  0x094a9a6e6aebdd1957fb5cec060bad60e6efc52d2316...      11080389  2020-10-18 14:15:12 UTC         12497848
7991896   0xe6cc8910f09b78d9c0beba162207429a9da8b331c10a...      11180371  2020-11-02 22:29:05 UTC         12500295

[14033192 rows x 4 columns]
--- Third Question Ran in 89.84499788284302 seconds ---
```



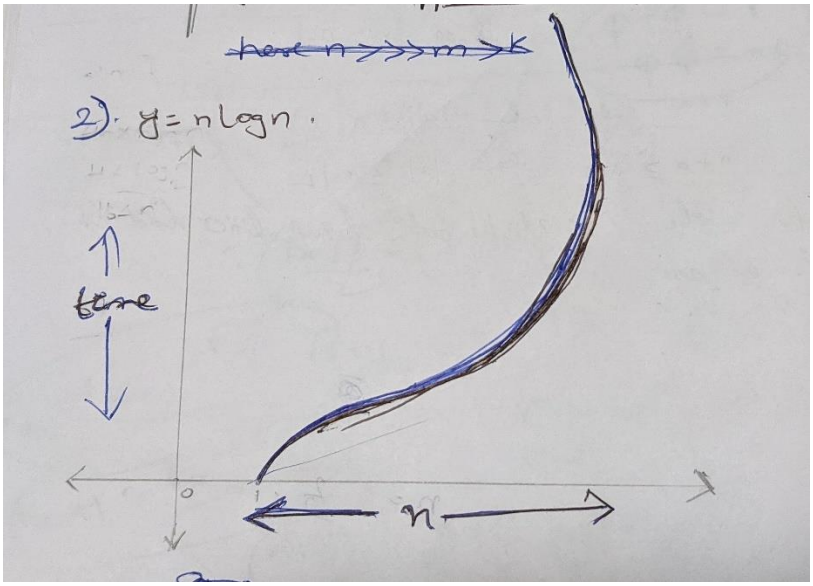
(It is time vs n graph where nlogn will be upper bound for n>=16.;)

Fourth question

```
### Fourth Question ###
start_time = time.time()

fourth_q = final_df.sort_values(["block_number", "gas_price"],ascending = (True, True), kind = 'heapsort')[['tx_hash','gas_price','block_number','block_time']]
#fourth_q.to_csv('/Users/srikaramara/Desktop/four.csv')
print(fourth_q)
print("--- Fourth Question Ran in %s seconds ---" % (time.time() - start_time))
```

- **Here we are using final_df which the unique transactions from complete data.**
- **We are using sort_values function to sort "block_number", "gas_price" columns**
- **Key data_structure used in the function : Heapsort**
- **Reason : doesn’t have any limitations like count sort, it is inplace algorithm which is better than merge sort and have efficient time complexity of nlogn in worst cases scenarios.**
- **Disadvantage -it is unstable algorithm meaning changes indices of original**
- **Time complexity – nlog(n).**
- **Asymptotic execution time:- O(nlog(n)) = O(14,033,192*log(14,033,192))**
- **Actual execution time:- 108.99 seconds**

```
                                                 tx_hash     gas_price  block_number              block_time
2739380   0x84a02c4eda95f489d1ece1507521cd82d6892a2a26bc...   77000000000      10966874  2020-10-01 00:00:10 UTC
8835224   0xc95e8098d92f39110a55f1069892dd6a0e251c26f568...   78000000000      10966874  2020-10-01 00:00:10 UTC
10923856  0x129b29609a41974333923a8af2af4180f7534fc31032...   78000001482      10966874  2020-10-01 00:00:10 UTC
1041221   0xf577e69df7ef2211f86d6cad04083157f95a6cc5986c...   80000000000      10966874  2020-10-01 00:00:10 UTC
9169012   0x4ecbc08bd02b9471f71d4532c5ffa70632bbfc1914b3...   80000000000      10966874  2020-10-01 00:00:10 UTC
...                                                     ...           ...           ...                       ...
17125092  0x5df70bc59da18d4544cb47b845b7c657a577ab311abe...   67500000000      11363269  2020-11-30 23:59:56 UTC
16739834  0xba1c627a494608de56628485a5b5f3e1d736c601d96c...   80000000000      11363269  2020-11-30 23:59:56 UTC
9633270   0xaec25a6e269f0415868f6590a680c139140a194882e9...   80800000000      11363269  2020-11-30 23:59:56 UTC
1202437   0xb795beb1c061b94e394a8444b38b6f7f65f719087762...   90000000000      11363269  2020-11-30 23:59:56 UTC
747382    0xa486c1a0f5df63ff7d3db1be672a04799c7c57a2d82e...  161460826569      11363269  2020-11-30 23:59:56 UTC

[14033192 rows x 4 columns]
--- Fourth Question Ran in 108.99601793289185 seconds ---
```



(Since only line2 is dominant part which is heapsort .It's best case and worst case are represented by blue pen and black pen that overlap in graph.)

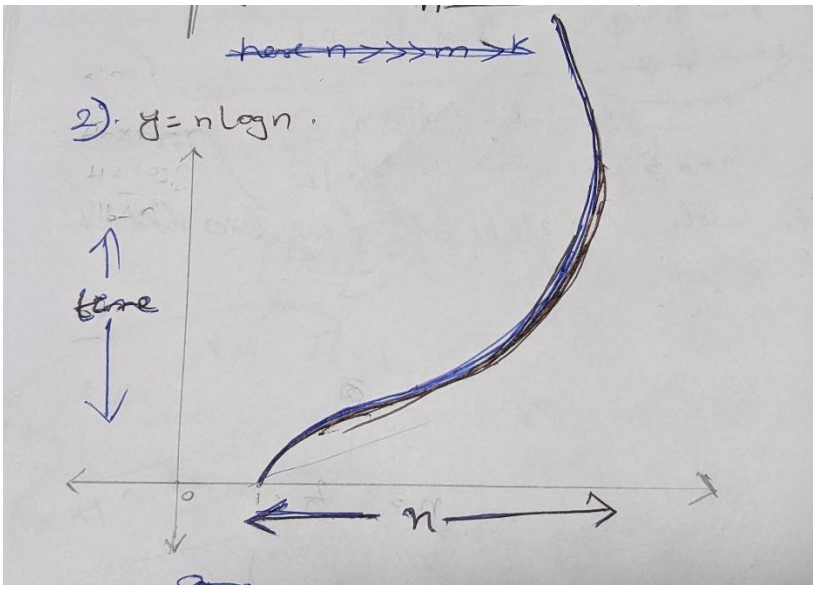
Fifth question

```
### Fifth Question ###

start_time = time.time()

fifth_q = tr_data.sort_values(["block_number", "from_addr","to_addr"],ascending = (True, True,True), kind = 'heapsort')[['from_addr','to_addr','tx_hash','block_number','block_time']]
print(fifth_q)

print("--- Fifth Question Ran in %s seconds ---" % (time.time() - start_time))
```

- **Here we are using all transactions from complete data.**
- **We are using sort_values function to sort "block_number", "from_addr", "to_addr" columns**
- **Now after grouping we apply "heapsort" which sorts gas value in ascending order**
- **Key data_structure: Heapsort**

- **Reason : doesn’t have any limitations like count sort, it is inplace algorithm which is better than merge sort and have efficient time complexity of nlogn in worst cases scenarios.**
- **Disadvantage -it is unstable algorithm meaning changes indices of original**
- **Time complexity – nlog(n)**
- **Asymptotic execution time:- O(nlog(n)) = O(17,406,410*log(17,406,410))**
- **Actual execution time:- 108.99 seconds**

```
                                    from_addr                                  to_addr  ... block_number           block_time
2851117   0x0000000000000000000000000000000000000000  0x150d7f02cdc552301095cdd09a90adfb1819c0e0  ...     10966874  2020-10-01 00:00:10 UTC
8835224   0x026d6fb910cb6ab7ab63128c82f252590f771f0b  0xc97a4ed29f03fd549c4ae79086673523122d2bc5  ...     10966874  2020-10-01 00:00:10 UTC
9500479   0x11ef34572ccab4c85f0baf03c36a14e0a9c8c7ea  0x7872eb21639ae8193f1aaf495b967202713c8fe5  ...     10966874  2020-10-01 00:00:10 UTC
6318273   0x11ef34572ccab4c85f0baf03c36a14e0a9c8c7ea  0x95b00984a9c38417cdba9653ce4f4a0a3e54451c  ...     10966874  2020-10-01 00:00:10 UTC
13853991  0x11ef34572ccab4c85f0baf03c36a14e0a9c8c7ea  0x992ef8145ab8b3dbfc75523281dad6a0981891bb  ...     10966874  2020-10-01 00:00:10 UTC
...                                              ...                                        ...  ...          ...                      ...
15110991  0xd291328a6c202c5b18dcb24f279f69de1e065f70  0x956c4de7cf44395dc0470671e723cebfffc2a8a5  ...     11363269  2020-11-30 23:59:56 UTC
1342182   0xda0c7a50171da3d76db313db0b059c165db2f62c  0x22273e4eddd80b1c4b77a88571e4389e8a4dbc35  ...     11363269  2020-11-30 23:59:56 UTC
12778601  0xde26d72fcf376a8a98560c8d57b1465715037851  0x0d0d65e7a7db277d3e0f5e1676325e75f3340455  ...     11363269  2020-11-30 23:59:56 UTC
11468551  0xf7a1f8b59d2c5901790954cae448186e32898921  0x20e3a9cf9f2d4773de24fb50b1eff25fed70acc3  ...     11363269  2020-11-30 23:59:56 UTC
563740    0xfb7a3112c96bbcfe4bbf3e8627b0de6f49e5142a  0x0000000000000000000000000000000000000000  ...     11363269  2020-11-30 23:59:56 UTC

[17406410 rows x 5 columns]
--- Fifth Question Ran in 215.24014592170715 seconds ---
```



(Since only line2 is dominant part which is heapsort .It’s best case and worst case are represented by blue pen and black pen that overlap in graph.)

Sixth question

```
### Sixth Question ###
start_time = time.time()

six_pd = thir_q.loc[thir_q['block_number'] == 11344115] # O(n) Simple Search
# six_pd.to_csv('/Users/srikaramara/Desktop/sixth.csv')
print(six_pd[['tx_hash','transaction_fee','tx_index_in_block','block_number','block_time']])

print("--- Sixth Question Ran in %s seconds ---" % (time.time() - start_time))
```

➔ Line 1:

- six_pd = thir_q.loc[thir_q['block_number'] == 11147095]
- Here we used third questions dataframe to find the block number.
- The key data structure used is **linear search**, where 11147095 block number is searched linearly for all rows, the rows with this number is set as true if present, or false if not present, at the same time thir_q.loc checks like a Boolean true or false and displays only the rows which are true.
- Time complexity is n.

Why linear search?

Actually, binary search is optimal compared to linear search, but inorder to binary search, we must have rows in sorted order, so for sorting, it is nlogn, searching logn, total nlogn + logn, but where as in linear search, it is simply n.

Therefore, for the above reason we are using linear search.

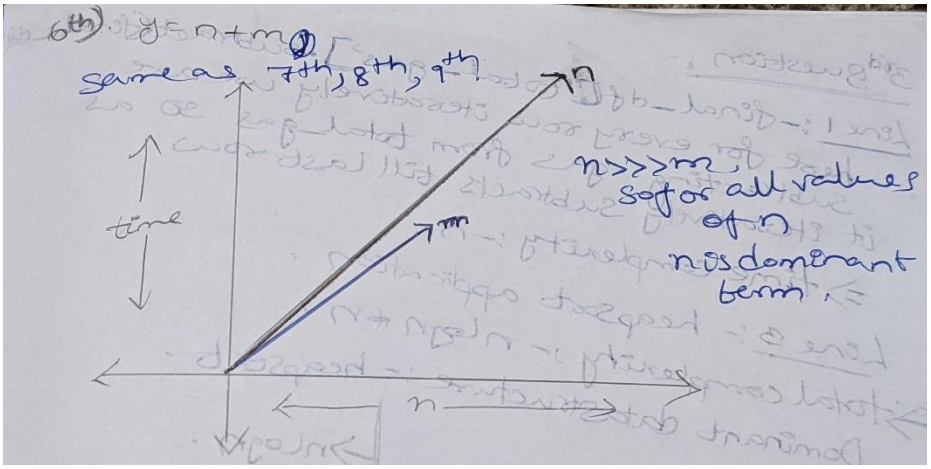
- Line 2:
- It iteratively checks the given columns and prints it by taking simultaneously.
- Time complexity:- m for columns

Total code time complexity : m+n

Asymptotic execution time:- O(n) = O(14,033,192)

Actual execution time:- 0.26 seconds

```
                                                tx_hash  transaction_fee  tx_index_in_block  block_number              block_time
16726655  0x8d21a430de88f00c5be62994bab4ccaa57bdf35b6b06...           124839                  5      11344115  2020-11-28 01:46:20 UTC
11211507  0xff2c43d2fe047733d656d2bd26cdbf919d00d9e370ac...           214773                  6      11344115  2020-11-28 01:46:20 UTC
16315777  0xb14eca574efa924592876365b63dfbc18aaedff48249...           223321                  9      11344115  2020-11-28 01:46:20 UTC
4202957   0x45e2952e825f0fbff6f249718b450c11d9736b41f26e...           816675                 13      11344115  2020-11-28 01:46:20 UTC
9711048   0xc29728cf7fde0fc25efaea288078bdc3201f09395d00...         11899580                 17      11344115  2020-11-28 01:46:20 UTC
878302    0xc2f67612140e23fd6446588c8a4e9e4b01a48a383c98...         12450007                 24      11344115  2020-11-28 01:46:20 UTC
--- Sixth Question Ran in 0.26903414726257324 seconds ---
```



(as n>>>m , so, far all values of n, n is dominant term which we can see from graph)

Seventh question

```
### Seventh Question ###

start_time = time.time()

sev_pd = thir_q.loc[thir_q['tx_hash'] == '0xd9da28fefdcd33f0bfee00b4b159c092c8e1f627d224c2856216d5ccedfdbdf3']

# sev_pd.to_csv('/Users/srikaramara/Desktop/seventh.csv')
print(sev_pd [['tx_hash','transaction_fee','tx_index_in_block','block_number','block_time']])

print("--- Seventh Question Ran in %s seconds ---" % (time.time() - start_time))
```

➔ Line 1:

sev_pd = thir_q.loc[thir_q[‘tx_hash’] ==
“0xd9da28fefdcd33f0bfee00b4b159c092c8e1f627d224c2856216d5ccedfdbdf3”]

- Here we used third questions dataframe to find the transaction hash.
- The key data structure used is **linear search**, where 0xd9da28fefdcd33f0bfee00b4b159c092c8e1f627d224c2856216d5ccedfdbdf3 hash is searched linearly for all rows, the rows with this number is set as true if present, or false if not present, at the same time thir_q.loc checks like a Boolean true or false and displays only the rows which are true.
- Time complexity is n.

Why linear search?

Actually, binary search is optimal compared to linear search, but inorder to binary search, we must have rows in sorted order, so for sorting, it is nlogn, searching logn, total nlogn + logn, but where as in linear search, it is simply n.

Therefore, for the above reason we are using linear search.

➔ Line 2:

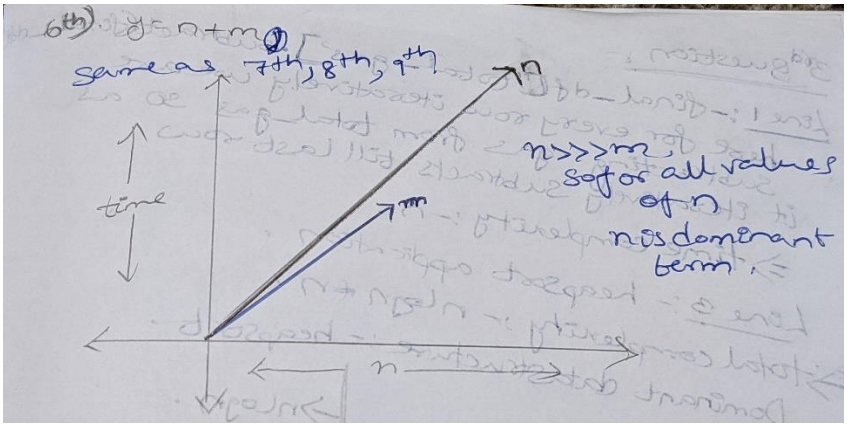
- It iteratively checks the given columns and prints it by taking simultaneously.
- Time complexity: - O(m) for columns

Total code time complexity : m+n

Asymptotic execution time:- O(n) = O(14,033,192)

Actual execution time:- 22.92 seconds

```
                                          tx_hash  transaction_fee  tx_index_in_block  block_number             block_time
83203  0xd9da28fefdcd33f0bfee00b4b159c092c8e1f627d224...          7243996                101      11240377  2020-11-12 03:27:42 UTC
--- Seventh Question Ran in 22.929280042648315 seconds ---
```



(as n>>>m , so, far all values of n, n is dominant term which we can see from graph)

Eight question

```python
### Eigth Question ###

start_time = time.time()

eight_df = final_df.loc[final_df['from_addr'] == '0x47ddfdddff875851ba18526cb30e0d35868c8c79a']

print(eight_df[['from_addr','tx_hash','block_number','block_time','transaction_fee']])

# eight_df.to_csv('/Users/srikaramara/Desktop/eigth.csv')
print("--- Eigth Question Ran in %s seconds ---" % (time.time() - start_time))
```

Line 1:

- Here we used third questions dataframe to find the transaction hash.

- The key data structure used is **linear search**, where 0x47ddfddff875851ba18526cb30e0d35868c8c79a from_addr is searched linearly for all rows, the rows with this number is set as true if present, or false if not present, at the same time final_df checks like a Boolean true or false and displays only the rows which are true.
- Time complexity is n.

Why linear search?

Actually, binary search is optimal compared to linear search, but inorder to binary search, we must have rows in sorted order, so for sorting, it is nlogn, searching logn, total nlogn + logn, but where as in linear search, it is simply n.

Therefore, for the above reason we are using linear search.

➔ Line 2:

- It iteratively checks the given columns and prints it by taking simultaneously.
- Time complexity: - O(m) for columns

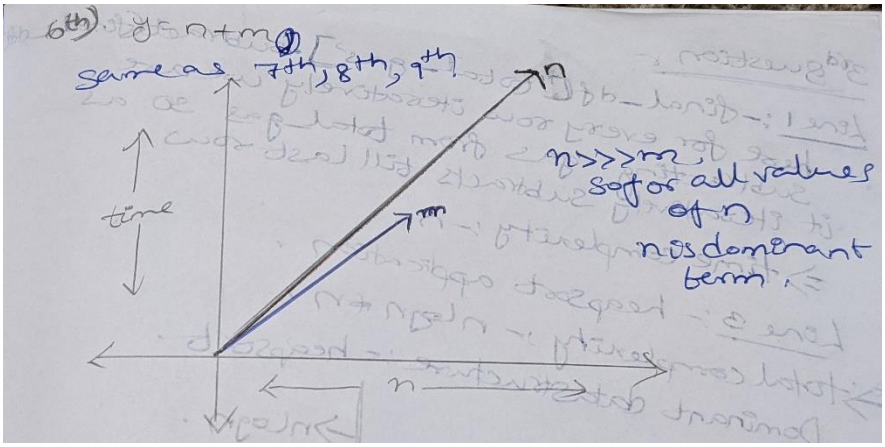
Total code time complexity : m+n

Asymptotic execution time:- O(n) = O(14,033,192)

Actual execution time:- 3.72 seconds

	from_addr	tx_hash	block_number	block_time	transaction_fee
1159796	0x47ddfddff875851ba18526cb30e0d35868c8c79a	0xbe80207e326dc48e3477944853c96eb04e7386712947...	11098177	2020-10-21 07:34:21 UTC	1871539
3892775	0x47ddfddff875851ba18526cb30e0d35868c8c79a	0x0900ea7b43ca6a906373b25b9dea06e8320123a50fee...	11108589	2020-10-22 21:53:11 UTC	2728604
4270561	0x47ddfddff875851ba18526cb30e0d35868c8c79a	0xe859f99ec7be0f2c57a64860cf8852064b2457fa7a14...	11271443	2020-11-16 21:51:30 UTC	1062398
4298058	0x47ddfddff875851ba18526cb30e0d35868c8c79a	0xda21288efb4cd8acc09487f072c3b45b4a17a184c21a...	11359650	2020-11-30 10:36:29 UTC	1012645
4316543	0x47ddfddff875851ba18526cb30e0d35868c8c79a	0x8b8c05429d74cbe77b725636a71d80a45e10fbef2f8f...	11099533	2020-10-21 12:31:13 UTC	1665303
4569898	0x47ddfddff875851ba18526cb30e0d35868c8c79a	0xc6c29afae14d310a38d61beeea7fdbc6f16e17109ea6...	11228372	2020-11-10 06:56:26 UTC	-37779
4646379	0x47ddfddff875851ba18526cb30e0d35868c8c79a	0xcb7a0c71d3db93779c5358f6a304b051f4e1b133b633...	11128253	2020-10-25 22:17:19 UTC	1163762
5881663	0x47ddfddff875851ba18526cb30e0d35868c8c79a	0x054246a889eae08fb174a32f35e833a44de3de1374b1...	11110500	2020-10-23 04:56:13 UTC	3531716
6728643	0x47ddfddff875851ba18526cb30e0d35868c8c79a	0x2ded2ee13c5db64113efbddc936455022f38898516d5...	11012711	2020-10-08 04:02:30 UTC	4284602
7625012	0x47ddfddff875851ba18526cb30e0d35868c8c79a	0x57c762e37540fbe11f399e31e897e5b438bf427a3bc7...	11016972	2020-10-08 20:17:19 UTC	204676
8665641	0x47ddfddff875851ba18526cb30e0d35868c8c79a	0x1c20a6d93573e393c9178d7ecbadfabee82663db415d...	11030383	2020-10-10 22:17:25 UTC	1529752
8868888	0x47ddfddff875851ba18526cb30e0d35868c8c79a	0x5c16e1da7dff2f22560ede274e8cdd60c0eb7720133b...	11247424	2020-11-13 05:20:07 UTC	1213273
9003738	0x47ddfddff875851ba18526cb30e0d35868c8c79a	0x1fcef9e5767b1ba3992d980f207aa324498315ddefa9...	11174891	2020-11-02 02:13:09 UTC	793859
9236985	0x47ddfddff875851ba18526cb30e0d35868c8c79a	0xd258a2f7d2938b0c6f35bc23695221b7371198de6c3f...	11089420	2020-10-19 23:16:35 UTC	1389841
10074041	0x47ddfddff875851ba18526cb30e0d35868c8c79a	0x29ec6b6abbd60ad4b0a97a4f98c83c704ab48b88ee03...	11359134	2020-11-30 08:50:45 UTC	777159
11204388	0x47ddfddff875851ba18526cb30e0d35868c8c79a	0x9d30917bce6459c75d6e3042e419332f037bc04a95ca...	11197128	2020-11-05 12:09:15 UTC	414276
11474978	0x47ddfddff875851ba18526cb30e0d35868c8c79a	0x700246d9eed699a3df17232089c9be545a95d4a889c4...	11242658	2020-11-12 11:56:31 UTC	8812528
11562249	0x47ddfddff875851ba18526cb30e0d35868c8c79a	0x5af40fa012d208b6e5554e151a013a019ed7bfbcb1a3...	11187270	2020-11-03 23:50:10 UTC	631669
11696615	0x47ddfddff875851ba18526cb30e0d35868c8c79a	0x7ac31767f6cddf0d267f31de192c642fa8d42f312809...	11086386	2020-10-19 12:23:42 UTC	3900099
12043153	0x47ddfddff875851ba18526cb30e0d35868c8c79a	0x4ce3a73595dcad81b4bd81afc5c1a8dbf9971dd62af7...	11291538	2020-11-19 23:45:13 UTC	272856
12112752	0x47ddfddff875851ba18526cb30e0d35868c8c79a	0x01b44d48d3e4882ff549d72d22e905519da488deebeb...	11082818	2020-10-18 23:12:42 UTC	5333654
12923104	0x47ddfddff875851ba18526cb30e0d35868c8c79a	0x8532e7f83034e5b71467e2f2ba448c8ff28ac90d71e1...	11144215	2020-10-28 09:02:17 UTC	670605
12949078	0x47ddfddff875851ba18526cb30e0d35868c8c79a	0xbd436fd364119224f34edbfecebd6172b98998b3817e...	11320039	2020-11-24 08:51:41 UTC	673412
13955922	0x47ddfddff875851ba18526cb30e0d35868c8c79a	0xe2e38780609b0761d2e326833ea093685bf1efca69a6...	11143010	2020-10-28 04:44:06 UTC	84750
13959737	0x47ddfddff875851ba18526cb30e0d35868c8c79a	0x9ba17c61913758725fab16014c96bbdf24e808f895af...	11188962	2020-11-04 06:03:27 UTC	534639
14617248	0x47ddfddff875851ba18526cb30e0d35868c8c79a	0xa9b498f12e2aba331d5c1045a25da16f2434e52219fe...	11110890	2020-10-23 06:17:15 UTC	2379883
15814898	0x47ddfddff875851ba18526cb30e0d35868c8c79a	0xd39d1bb3736103414fbe6547e66135ee0929e1b06ad5...	11110877	2020-10-23 06:15:03 UTC	571225
16003358	0x47ddfddff875851ba18526cb30e0d35868c8c79a	0xd85af7182b6f2d351eb9cc8c70f0614bbd3b8160522d...	11115289	2020-10-23 22:39:13 UTC	59615
16968955	0x47ddfddff875851ba18526cb30e0d35868c8c79a	0x7c6f096a66fce2cd43365a18e7be6b29d7098f3da76f...	11018617	2020-10-09 02:33:06 UTC	844585
17006156	0x47ddfddff875851ba18526cb30e0d35868c8c79a	0x4fb23eff6be72e9dc6311d4faab31c0bacd1c6d1e1ad...	11136474	2020-10-27 04:46:01 UTC	2107722
17093493	0x47ddfddff875851ba18526cb30e0d35868c8c79a	0xbb036414da53acdb39c7b304e629d5bfa99a2dcf7bc1...	11220705	2020-11-09 02:55:50 UTC	289123

--- Eigth Question Ran in 3.7239410877227783 seconds ---



(as n>>>m , so, far all values of n, n is dominant term which we can see from graph)

Ninth question

```
### Nineth Question ###

start_time = time.time()

nineth_df = final_df.loc[final_df['to_addr'] == '0x7a250d5630b4cf539739df2c5dacb4c659f2488d']

print(nineth_df[['from_addr','tx_hash','block_number','block_time','transaction_fee']])

# nineth_df.to_csv('/Users/srikaramara/Desktop/nineth_df.csv')

print("--- Nineth Question Ran in %s seconds ---" % (time.time() - start_time))
```

Line 1:

nineth_df = final_df.loc[final_df['to_addr'] == '0x7a250d5630b4cf539739df2c5dacb4c659f2488d']

- Here we used third questions dataframe to find the transaction hash.
- The key data structure used is **linear search**, where '0x7a250d5630b4cf539739df2c5dacb4c659f2488d' to_addr is searched linearly for all rows, the rows with this number is set as true if present, or false if not present, at the same time final_df checks like a Boolean true or false and displays only the rows which are true.
- Time complexity is n.

Why linear search?

Actually, binary search is optimal compared to linear search, but inorder to binary search, we must have rows in sorted order, so for sorting, it is nlogn, searching logn, total nlogn + logn, but where as in linear search, it is simply n.

Therefore, for the above reason we are using linear search.

➔ Line 2:

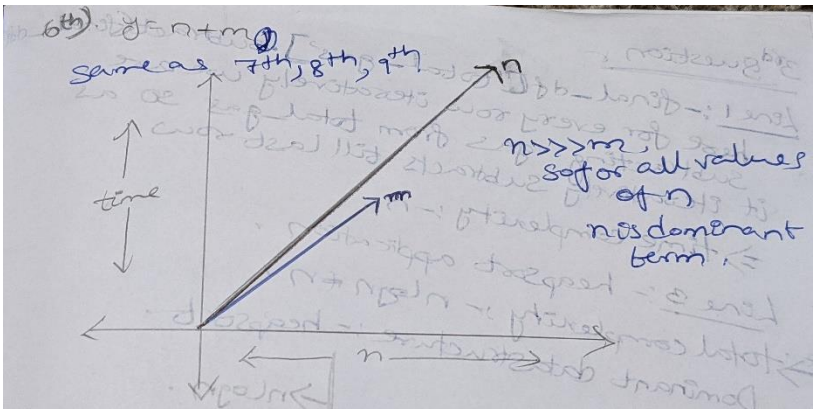
- It iteratively checks the given columns and prints it by taking simultaneously.
- Time complexity: - O(m) for columns

Total code time complexity : m+n

Asymptotic execution time:- O(n) = O(14,033,192)

Actual execution time:- 21.89 seconds

```
                                  from_addr                                                tx_hash  block_number             block_time  transaction_fee
14        0x608903534527b0623fe0b0bd81a2f29bc5b50d32  0x575ed73a647d0254ca9d5713733940b6ed6527a1370d...      10974876  2020-10-02 05:54:30 UTC          9314579
46        0xb7864c708ad58af75c756c26b1ba155bfa0e2307  0x85e02ec9c5058c3a49f0104399667 83bbe328b741f33...      11236004  2020-11-11 11:17:15 UTC          6751100
69        0x705dffd844f9252cf85a26bc178e59212914f2ef  0x012863fea8ce7c33b70b8be86f6f19475f18e0dc09ab...      11244261  2020-11-12 17:55:13 UTC          9317677
71        0x0d4a11d5eeaac28ec3f61d100daf4d40471f1852  0xf3368fca1250c5a2d542df9ad3764bc59dad1598eba2...      11262630  2020-11-15 13:18:37 UTC          5437891
72        0x86fef14c27c78deaeb4349fd959caa11fc5b5d75  0xce5ae1a8221a18495c41c85c5ccc7d3cadd3a03f06f9...      11181409  2020-11-03 02:08:22 UTC          1955573
...                                              ...                                                ...           ...                       ...              ...
17406205  0x92699aaad95da504d52feb80a4c113f7105f9339  0xa0031382da3ed96b5fd844f9878ddd8ea28c501a7131...      11159396  2020-10-30 17:05:04 UTC          7230514
17406210  0xa2107fa5b38d9bbd2c461d6edf11b11a50f6b974  0x88c25a59173bcc6a4bf1fc4f8891a21a0d225d194cf5...      11045652  2020-10-13 06:24:13 UTC          3732384
17406292  0xffa98a091331df4600f87c9164cd27e8a5cd2405  0xbd9e58b14b54b7f5dedc1d4db607449f51a194379d7b...      11359422  2020-11-30 09:51:33 UTC          4437525
17406331  0xd3d2e2692501a5c9ca623199d38826e513033a17  0xc004516c2a2875f4fd741da69693db18ec1b89fb6759...      11158231  2020-10-30 12:54:54 UTC          3765720
17406379  0x960d228bb345fe116ba4cba4761aab24a5fa7213  0x7075cbae17ff93d546c2803302b5cf9c06028b8cb513...      11082382  2020-10-18 21:36:25 UTC           545736

[566996 rows x 5 columns]
--- Nineth Question Ran in 21.809725999832153 seconds ---
```



(as n>>>m , so, far all values of n, n is dominant term which we can see from graph)

Tenth question

```python
### Tenth Question ###
start_time = time.time()

from_addr = '0xcfa770b0f8970286c839724f94d46db8a71be39a'
mx = tr_data.loc[tr_data['from_addr'] == from_addr]['token_qty'].max()
mn = tr_data.loc[tr_data['from_addr'] == from_addr]['token_qty'].min()

d = {'from_addr': from_addr, 'max_token_transfer': [mx], 'min_token_tranfer': [mn]}
tenth_pd = pd.DataFrame(data=d)

# tenth_pd.to_csv('/Users/srikaramara/Desktop/tenth_df.csv')
print(tenth_pd)

print("--- Tenth Question Ran in %s seconds ---" % (time.time() - start_time))
```

➔ Line 1:

- mx = tr_data.loc[tr_data['from_addr'] == from_addr]['token_qty'].max()
- Here similar to line 1 in 6th question, it perform linear search and produces rows which have that from_address

Why linear search:

Actually, binary search is optimal compared to linear search, but inorder to binary search, we must have rows in sorted order, so for sorting, it is nlogn, searching logn, total nlogn + logn, but where as in linear search n.

Therefore, for the above reason we are using linear search.

Time complexity:- n.

In worst case nth element can be max. So a **linear traversal** to find max is produced by function .max();

Why linear traversal? Why not getmax from heap?

N heap root element is max and building heap takes n time complexity. But there are some other factors or computations which is more than n say n+k

But in linear traversal only one iteration is enough

Total line complexity:- n+n =2n

Line 2: here we use same operation as max but we find minimum element.

Time complexity:- n

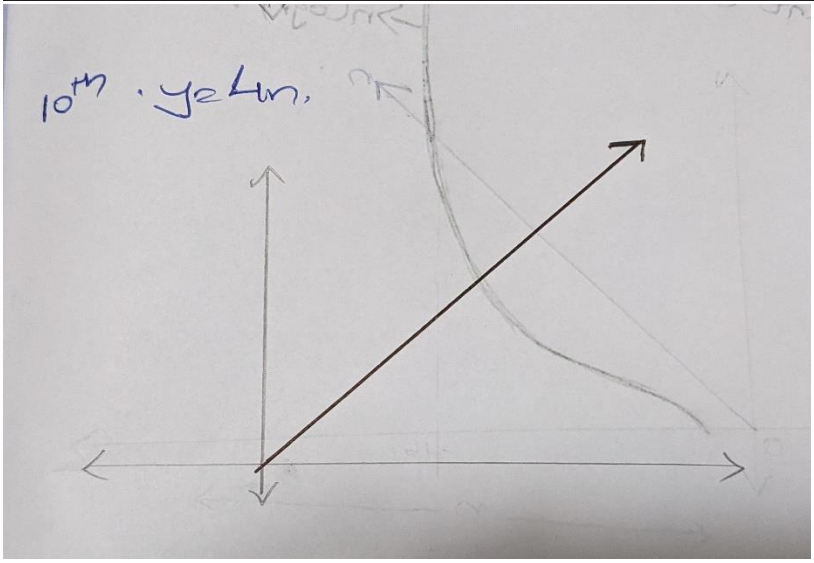
Total line complexity:- n+n =2n

Total code complexity:-4n.

Asymptotic execution time:- O(n) = O(17,406,410)

Actual execution time:- 2.743686199188 seconds

```
                                  from_addr max_token_transfer min_token_tranfer
0  0xcfa770b0f8970286c839724f94d46db8a71be39a                 4                 0
--- Tenth Question Ran in 2.7436861991882324 seconds ---
```



(from the graph as total code time complexity is 4n ,it's best case is omega(1).)

OBSERVATIONS:-

Theoretically we know that for larger n we would get more time value in substituting in dominant term ,but actual execution time depends on servers, GPU's which might be varying with theoretical aspect.

Group – 5 :

Srikar Amara

Sai Srikanth Sarabu

Srinath Sai Tripuraneni

Sai Koushik Reddy Chityala

Chaitanya Madisetty