Partition problem

Introduction

The background to the partition problem is described in **MOD007357 Coursework Brief 2021-22 (Task C)**, including the description of six simple splitting approaches (labelled A-F).

Implementation summary

The splitting algorithms have been implemented in the *python* programming language, and can be conveniently applied to an input array (*multiset*) by initialising an `ArraySplitter` class with the array, and calling the class' `split()` method - providing a string key for the required method.

The described methods A-F have been extended with method 'G' - an implementation of the KK splitting algorithm[1].

ZIP archive

The code and output of running the `main()` method can be found in the associated ZIP archive (`partition-problem.zip`). Contents:

```
partition-problem.zip
  - Partition problem.html
  - Partition problem.ipynb
  - partition_problem.py
  - README.md
  - environment.yml
  results/
    - comparison_pandas.png
    - comparison_seaborn.png
    - testing_df.csv
    - walkthrough_examples.txt
```

Dependencies

The `conda` environment required to run the `partition_problem.py` code, and this notebook, can be recreated from the *environment.yml* file in the ZIP archive, and activated as follows:

```
conda env create -f environment.yml
conda activate partition-problem
```

We can use a jupyter 'magic' command to generate the *environment.yml* file.

In [ ]:
```
!conda env export --from-history > environment.yml
```

Notebook

This notebook was created within the same python `conda` environment used to run the `partition_problem.py`.

To generate the PDF version, the notebook was exported to HTML format and then printed to PDF format from Google Chrome.

Imports

In [ ]:
```
import pandas as pd
from matplotlib import pyplot as plt
import seaborn as sns

from partition_problem import ArraySplitter
from partition_problem import generate_random_array, run_tests, seaborn_plot, pandas_plot

pd.set_option('display.max_rows', 10)  # Keep table displays short
sns.set_context('talk')  # Make seaborn plots have good font sizes, line widths, etc
```

Split method G - the Karmarkar-Karp heuristic (KK)

The KK method - described in section 2.3 of [1] - was implemented as an additional method of the `ArraySplitter` class, and is reproduced below for reference.

In [ ]:
```
# Approach G - time complexity O(nlogn)
def _split_kk(self, array):
    """Sorts high to low, then removes the first two members of the array.
    The 2nd value is subtracted from the first value, and the difference is appended to the
    end of the list. The list is then resorted, and the process is repeated until the list
    length is 1.

    Finally the list is passed back as s1_, and an empty list is passed back as s2_.
    This is a different approach to the other algorithms, hence the different variable
    naming for clarity.

    Passing back the two lists allows downstream calculation of the absolute subset sum
    difference (even though we already know the answer)
    """
    s1_ = []
    s2_ = []

    reverse_sorted_array = sorted(array, reverse=True)  # O(nlogn)
    while len(reverse_sorted_array) > 1:  # O(n)
        elem_0 = reverse_sorted_array.pop(0)
        elem_1 = reverse_sorted_array.pop(0)
        reverse_sorted_array.append(elem_0 - elem_1)
        reverse_sorted_array.sort(reverse=True)  # O(nlogn)
    # We know the difference at this point, but we are not using or returning it
    s1_ = reverse_sorted_array
    return s1_, s2_
```

Walkthrough examples

The splitting applied by each method can be seen below for a single input array of cardinality 32.

Generate the array

In [ ]:
```
array = generate_random_array(min_val=1, max_val=320, cardinality=32)
print(array)
```

```
[147, 218, 164, 206, 7, 90, 169, 295, 290, 26, 194, 166, 106, 91, 77, 141, 310, 300, 147, 13
2, 281, 236, 103, 318, 74, 317, 78, 131, 115, 163, 320, 234]
```

Apply each splitting method in turn

In [ ]:
```
splitter = ArraySplitter(array)
results = splitter.compare_methods(verbose=True)
```

```
Method: A
Time complexity: O(n)
  Subset 1: [147, 218, 164, 206, 7, 90, 169, 295, 290, 26, 194, 166, 106, 91, 77, 141]
  Subset 2: [310, 300, 147, 132, 281, 236, 103, 318, 74, 317, 78, 131, 115, 163, 320, 234]
  Absolute partition difference: 872

Method: B
Time complexity: O(n)
  Subset 1: [218, 164, 206, 90, 290, 26, 194, 166, 106, 310, 300, 132, 236, 318, 74, 78, 320,
234]
  Subset 2: [147, 7, 169, 295, 91, 77, 141, 147, 281, 103, 317, 131, 115, 163]
  Absolute partition difference: 1278

Method: C
Time complexity: O(n)
  Subset 1: [147, 164, 7, 90, 169, 290, 166, 91, 310, 147, 281, 103, 74, 317, 115, 320]
  Subset 2: [218, 206, 295, 26, 194, 106, 77, 141, 300, 132, 236, 318, 78, 131, 163, 234]
  Absolute partition difference: 64

Method: D
Time complexity: O(n)
  Subset 1: [218, 290, 310, 281, 318, 320, 147, 7, 90, 26, 166, 91, 141, 132, 74, 131]
  Subset 2: [206, 295, 194, 300, 236, 317, 234, 164, 169, 106, 77, 147, 103, 78, 115, 163]
  Absolute partition difference: 162

Method: E
Time complexity: O(nlogn)
  Subset 1: [320, 317, 300, 290, 236, 218, 194, 166, 163, 147, 132, 115, 103, 90, 77, 26]
  Subset 2: [318, 310, 295, 281, 234, 206, 169, 164, 147, 141, 131, 106, 91, 78, 74, 7]
  Absolute partition difference: 142

Method: F
Time complexity: O(nlogn)
  Subset 1: [320, 310, 300, 290, 234, 206, 194, 164, 147, 141, 132, 115, 91, 90, 74, 7]
  Subset 2: [318, 317, 295, 281, 236, 218, 169, 166, 163, 147, 131, 106, 103, 78, 77, 26]
  Absolute partition difference: 16

Method: G
Time complexity: O(nlogn)
  Subset 1: [0]
  Subset 2: []
  Absolute partition difference: 0
```

Input parameters

The parameters below can be varied and will control the subsequent code blocks. These are the same parameters accepted by the `main()` function of `partition_problem.py`.

In [ ]:
```
CARDINALITIES = [32, 64, 128, 256, 512, 1024]
REPEATS = 10000
MIN_ARRAY_VAL = 1
MAX_VAL_FACTOR = 10
```

Running tests

For each *cardinality* value we will run *number_of_tests* tests.

Each test involves the following steps:

1. Make a new random array
2. Make an `ArraySplitter` instance for the random array
3. Use the ArraySplitter's `compare_methods()` function to get the absolute difference of the subset sums for each different split method
4. Add the results from the iteration to a list that keeps track of all of the results

In [ ]:
```
testing_results = []  # a list to keep each iteration's results in
for cardinality in CARDINALITIES:  # iterate over cardinalities
    for test in range(REPEATS):  # then iterate over n tests (eg 100)
        # 1. Make a new random array
        array = generate_random_array(min_val=MIN_ARRAY_VAL,
                                      max_val=MAX_VAL_FACTOR * cardinality,
                                      cardinality=cardinality)
        # 2. Use an ArraySplitter to get the results for each available split method
        splitter = ArraySplitter(array)
        results = splitter.compare_methods()
        # 3. Add the cardinality info to the result
        results['cardinality'] = cardinality
        # 4. add the result to the testing_results list
        testing_results.append(results)
```

In `partition_problem.py` the above steps can be conveniently run by simply calling the `run_tests()` method.

```
testing_results = run_tests(cardinalities=CARDINALITIES, repeats=REPEATS,
                            min_array_val=MIN_ARRAY_VAL,
max_val_factor=MAX_VAL_FACTOR)
```

An example result in the list (showing the dictionary structure):

In [ ]:
```
testing_results[0]
```

Out[ ]:
```
{'A': 195,
 'B': 249,
 'C': 115,
 'D': 139,
 'E': 153,
 'F': 21,
 'G': 1,
 'cardinality': 32}
```

If we convert the whole `testing_results` list of dictionaries into a DataFrame, we get the following shape.

In [ ]:
```
testing_df = pd.DataFrame(testing_results)
testing_df
```

Out[ ]:

	A	B	C	D	E	F	G	cardinality
0	195	249	115	139	153	21	1	32
1	90	1914	286	42	220	4	0	32
2	463	1405	207	7	201	3	1	32
3	322	1444	20	94	196	2	0	32
4	227	277	107	69	133	5	1	32
...	...	...	...	...	...	...	...	...
59995	202326	142858	1382	88	5142	20	0	1024
59996	16631	56079	5151	4215	5515	25	1	1024
59997	188464	243190	5762	30	5094	12	0	1024
59998	23092	12468	3900	2384	4818	10	0	1024
59999	181439	116555	3389	2041	5261	11	1	1024

60000 rows × 8 columns

Plotting with Seaborn

The above shape of the data is convenient for readability, but not so good for certain plotting approaches. The `seaborn` package makes plotting data (including data from a `pandas.DataFrame`) easy, but it is best to provide the data in a "tall, skinny" format.

Convert raw results into tall-skinny DataFrame

The raw data can be unpivotted using `pandas.melt()` . This gives us the tall, skinny version of the output data where each method appears not as a column, but as a value in a single 'method' column.

In [ ]:

```
tall_df = pd.melt(testing_df, id_vars=['cardinality'],
                  value_vars=splitter.func_dict.keys(),
                  var_name='method', value_name='absolute_diff')
tall_df
```

Out[ ]:

	cardinality	method	absolute_diff
0	32	A	195
1	32	A	90
2	32	A	463
3	32	A	322
4	32	A	227
...	...	...	...
419995	1024	G	0
419996	1024	G	1
419997	1024	G	0
419998	1024	G	0
419999	1024	G	1

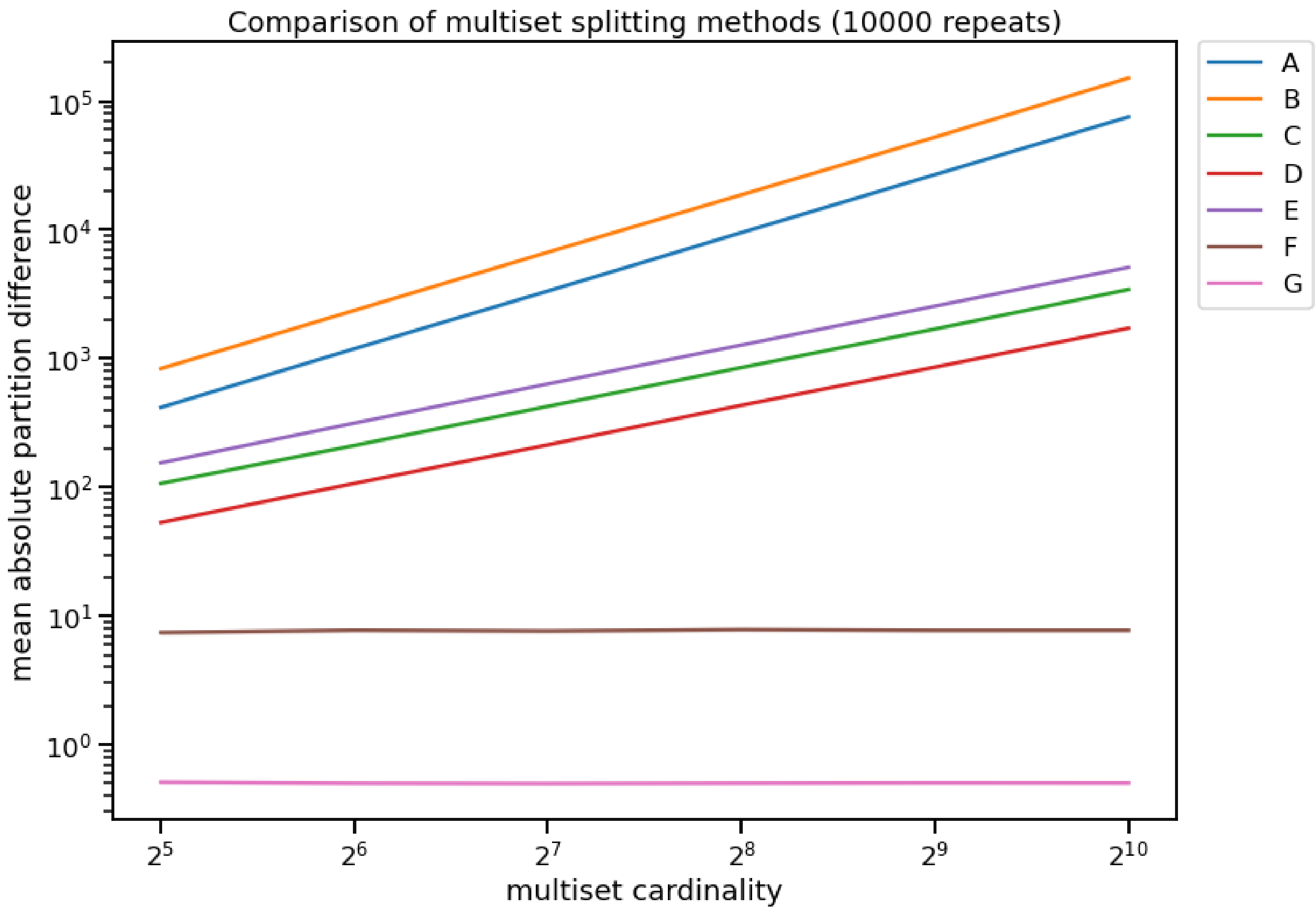
420000 rows × 3 columns

Plot

Seaborn's `lineplot()` method is an appropriate choice. This method will plot the mean of the data, and can automatically show variation from the mean in a number of ways. As we are probably most interested in how confident we are in the predicted mean values, 95% confidence intervals have been chosen (rather than standard deviations).

In [ ]:

```
fig, ax = plt.subplots(figsize=(12, 9), facecolor='white')
ax = sns.lineplot(data=tall_df, x='cardinality', y='absolute_diff', hue='method', ci=95)
ax.set_xlabel('multiset cardinality')
ax.set_xscale('log', base=2)
ax.set_ylabel('mean absolute partition difference')
ax.set_yscale('log')
# Put legend outside plot
ax.legend(bbox_to_anchor=(1.02, 1), loc='upper left', borderaxespad=0)
# Underscore assignment to supress Text object output
_ = ax.set_title(f'Comparison of multiset splitting methods ({REPEATS} repeats)')
```



Both of the above steps can be conveniently run by calling the `seaborn_plot()` method from `partition_problem.py`. The inputs to this function are simply the raw result dictionary list, and the list of cardinality values used to generate the results:

```
fig = seaborn_plot(results=testing_results, repeats=REPEATS)
```

Plotting with pandas

This is a bit more 'DIY' than using Seaborn.

We will use the `groupby()` method of the `pandas.DataFrame` to aggregate the results by cardinality, and generate corresponding statistical values (*mean*, *std*).

This grouped form can then be easily used to plot with.

Group the data

In [ ]:

```
# Group by cardinality and method
group = tall_df.groupby(['cardinality', 'method'])
# Dropping level 0 of axis 1 allows us to use 'mean' and 'std' as the column names
# instead of ('absolute_diff', 'mean') and ('absolute_diff', 'std')
stats_df = group.agg(['mean', 'std']).droplevel(axis=1, level=0)
# Some of the seaborn code above
stats_df
```

Out[ ]:

cardinality	method	mean	std
32	A	417.1434	315.032901
	B	835.5898	632.621641
	C	107.0974	75.873346
	D	53.2330	38.639325
	E	154.6836	27.448644
...	...	...	...
1024	C	3442.7974	2437.622184
	D	1725.2598	1211.461576
	E	5116.2232	162.388149
	F	7.7386	8.492083
	G	0.5042	0.500007

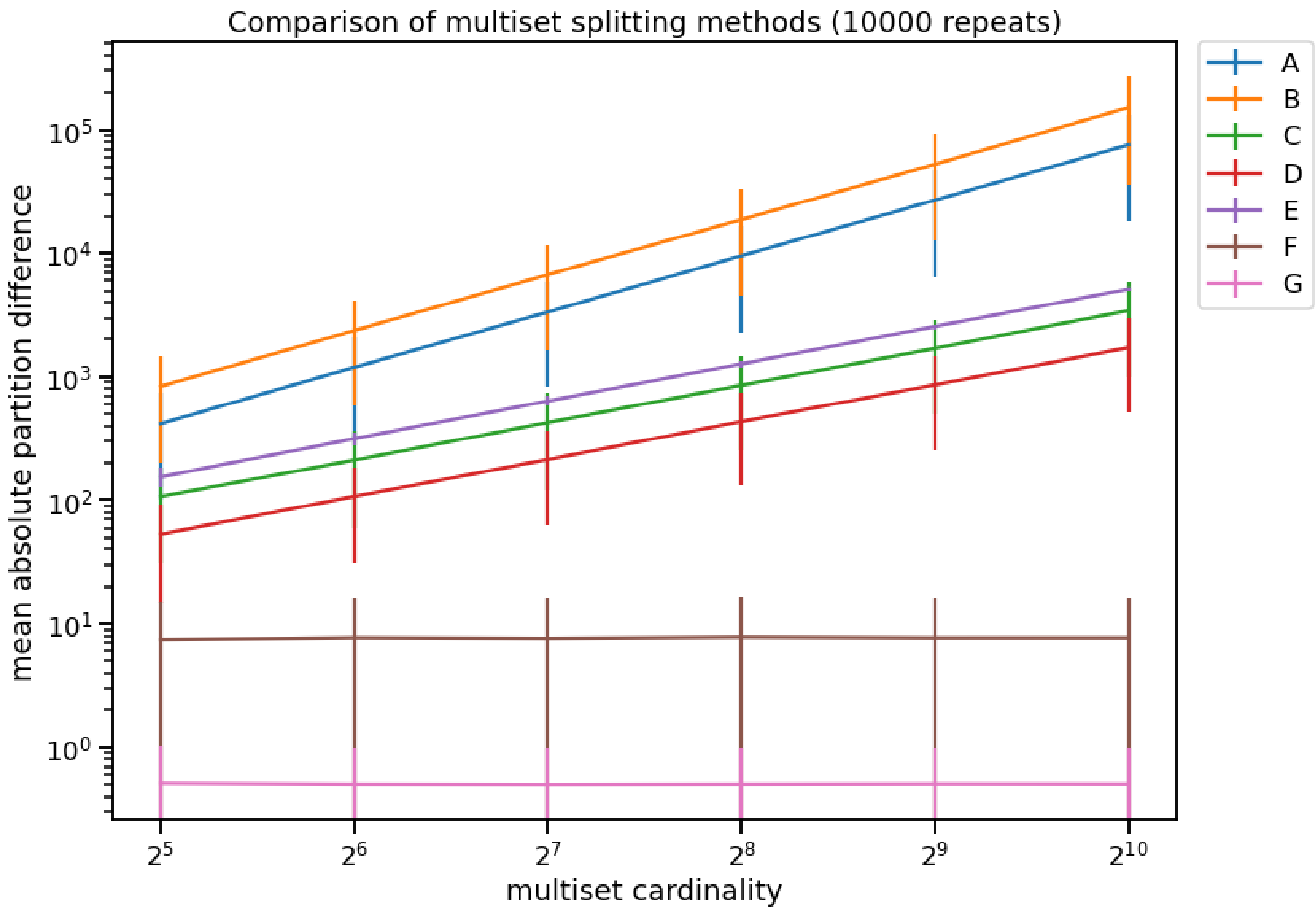
42 rows × 2 columns

Plot

We will iterate over (using `group()` again) each method, and add a plot to the axis. The `pandas` plot will show error bars (not area) by default and, as we have calculated standard deviation in the initial aggregation, we will plot that instead of the 95% confidence interval.

In [ ]:

```
fig, ax = plt.subplots(figsize=(12, 9), facecolor='white')
# Iterate over each method and add the data to the plot
for key, group in stats_df.groupby('method'):
    group.reset_index(inplace=True)
    group.plot('cardinality', 'mean', yerr='std', label=key, ax=ax)
ax.set_xlabel('multiset cardinality')
ax.set_xscale('log', base=2)
ax.set_ylabel('mean absolute partition difference')
ax.set_yscale('log')
# Put legend outside plot
ax.legend(bbox_to_anchor=(1.02, 1), loc='upper left', borderaxespad=0)
# Underscore assignment to supress Text object output
_ = ax.set_title(f'Comparison of multiset splitting methods ({REPEATS} repeats)')
```



Again, this functionality can be easily accessed by using the `pandas_plot()` method from `partition_problem.py`:

```
fig = pandas_plot(results=testing_results, repeats=REPEATS)
```

References

[1] https://www.ijcai.org/Proceedings/09/Papers/096.pdf

Resources used but not directly referenced

- https://en.wikipedia.org/wiki/Partition_problem
- https://pandasguide.readthedocs.io/en/latest/
- https://peps.python.org/pep-0008/
- https://seaborn.pydata.org/api.html
- https://en.wikipedia.org/wiki/Timsort