Solution CS Problems

A few words before the reader begins.

In general, all code solutions are broken down into three files:

1. `common.py`, which is a quasi-module of classes and functions that remains the same throughout this whole solution for the problem-dependent code of the other files to import and use. The classes are solely used for the purpose of inheritance, to avoid rewriting many utility methods. This is not really a module, as that would require installation via a `setup.py`, and the problem set does not allow for such an operation, so I just ended up copying and pasting the same file to all problem solutions. A `Sorter` is a class capable of reading, storing, and writing data, and contains instances of `Data`.
2. `x_sort.py`, in which the specific child class of `Sorter` is implemented. The child class specifies what kind of data the class handles, how the sorting happens, and how to verify that the result of a sorting is correct. The convention is adopted that the sorting algorithms all return a list of integers, each being an index to the data held in the `Sorter`. By enumerating through the returned indices in order and accessing the corresponding child, one obtains the sorted list. The timing of each sorting ends when the sorted indices are produced, as, in my opinion, sorted indices and instances of the data are equilvalent, and algorithms can be easily modified to do one or the other without much difference in performance speed. I adopted the indices simply to provide more flexibility.
3. `test.py`, where all the test functions are provided. The `method` chosen in `test_output` is the final submission sorting method, while `test_all` provides capability to see all or some of the sorting methods that I implemented in action--to understand why I preferred one over another.

I apologize for the difficulties my choice of language cause to the analysis of memory and runtime. I started off with Python, my default language of choice, and now is too late to scrap the whole thing and rewrite the solutions in a lower level language. Python does not have good support for fixed-size arrays, so the default list is used, which fails to show off many of the benefits of non-dynamically sized arrays in memory usage. For example, in `P3`, linear radix sort fails to compete with the builtin $O(n \log n)$ Timsort mostly due to the expensive operations of list operations; if the same comparison were made in C, radix sort would definitively be faster than Timsort for any reasonably big dataset. Regardless, I'll be submitting the `fastest` solutions I could achieve in pure Python (no non-standard packages used, since packages like Numpy "cheat" by compiling some of the code directly in C).

For each solution, `cd` into the directory of the problem-specific solution (e.g. `P1`) and run `test.py` from there. There are functions supplied to generate and/or test files in the directory `tests`, so put whatever test file the reader would like to try under `tests` and check for outputs in the directory `output`.
All code snippets given in this solution set should be run in `test.py` of the appropriate problem folder (paste the code at the end and comment out any earlier code that's not function definition, then run `python test.py`). The `test.py` file itself always has some test code and some commented out test code invoking the defined test functions already in there, but it's meant as examples.

As a general rule, the `test_output` function is the one you would want to run. It calls the final solution method I decided on (the other methods not used in the final solution are left in to show the thinking process/comparison between different approahces) and sorts all the test files in the directory `tests`, outputting the sorted results and their runtime into files in the directory `output`. By supplying `tests` with your own test files, you can test out my final solution.

Since the test files are massive, I have deleted them before uploading. Just use the `generate_testing_data` function to generate them again.

Citations are made as comments in the code, usually just in the form of a link.

P1

See the folder `P1`. My `sort_counting` method yielded a ~0.0002 ms/item sorting speed (since it's linear), while the naive implementation took ~0.0005 ms/item. The edge cases only sped up the result a bit.

P2

(a) I considered a number of approaches:

1. one-pass sort on the ordered tuple (graduation, GPA) using Timsort
2. two-pass sort, first of the graduation year (counting sort), then, within each year, the GPA (Timsort)
3. counting sort by breaking down graduation year and GPA into individual bins (total of 5*401 possible bins). This obviously works better for larger datasets, since only then would the linear runtime shine over O(nlogn) and compensate for its expensive setup
4. radix sort: first graduation year, then last digit of the GPA, then second-to-last, then first digit

I decided to not use radix sort since ultimately it works similarly to the two-pass sort, just with a few extra steps.

After testing all three remaining methods,

```
test_all(["sort_builtin", "sort_counting", "sort_twopass"], verify=False)
```

the two-pass and counting sort definitively end up being definitively faster than the builtin method for $n > 1000$, but the builtin method is faster for small datasets since it does not require the slow initializations of the other two methods. For $1000000 > n > 1000$, the two-pass method is a lot faster than counting by a small margin (usually less than 10%). For datasets larger than 1000000, the counting sort works the best. Therefore, my final submission is a combination of them, depending on the data size. See the method `sort_combined` for my final submission.

(b) It is difficult to analyze the runtime of my algorithm, since it's a combination of everything, but I can analyze each component part.

One Pass

Since it's just Timsort, the best case scenario is $O(n)$, worst case scenario is $O(n \log n)$, and the average is

$O(n \log n)$.

Two Pass

Likewise, since the second step is ultimately is just Timsort, and the first step is linear runtime, the best case scenario is $O(n)$, the worst case scenario is $O(n \log n)$. The average requires some contemplation: it's just the average runtime of each component step on a dataset size of $\frac{n}{4}$ summed plus the linear runtime of the first step. Ultimately it is still roughly $O(n \log n)$.

Counting Sort

It is $O(n)$ all across, since it's linear.

Clearly, our final algorithm has the best case runtime $O(n)$ and worst case runtime $O(n \log n)$. The average case runtime is somewhere between $O(n \log n)$ and $O(n)$, depending on the size of the data. Since for $n > 1000000$, the algorithm becomes linear, it can be said that the $O(n \log n)$ runtime of the other two methods are roughly $O(n \log 1000000)$, which is just $O(n)$. But this is just technicality, since real-world data sets, especially for school records, rarely exceed that size.

(c) Again, the memory varies depending on the method employed. Also, note that it is significantly harder to get an exact memory count for Python than for a language like C, and the builtin `Timsort` is a bit of a blackbox that simply has $O(N)$ memory usage. Let's call the memory usage of a `Timsort` of N elements $T(N)$, and just take for granted that it's some linear function of N.

Timsort

As stated earlier, $T(N)$.

Two Pass

The list `l` is a length 5 list, which is 128 bytes. A Python integer on the other hand costs 28 bytes, so storing N of them uses $28N$ bytes, and since each Timsort happens after the one before is done, the Timsort uses on average $T(\frac{N}{5})$ bytes. So in total, the sorting uses $T(\frac{N}{5}) + 28N + 128$ bytes.

Counting Sort

A list of 2005 empty lists is 16560 bytes, and storing N integers use $28N$. The sorting uses $28N + 16560$ bytes.

P3

(a) It's definitely possible (by having each student ID as a unique bin), but certainly not very efficient. We'd need 9^{10} different bins, and each bin at best would have only one element, since the ids are all unique. Each element would be placed in its corresponding bin (e.g. ID 100000000 would be placed in the 100000000'th bin). The memory cost and time to arrays of such size is incredibly high. Since the maximum input dataset is 9^{10}, the efficiency of an algorithm like radix sort is $O(9N)$, but is much more memory efficient, only using $O(N)$ for memory, which is always going to be smaller than the memory used by counting sort.

(b) This is definitely better than counting sort. Radix sort would run through the data set 9 times, one for each digit. For time efficiency, each run takes $O(N)$ time, where N is the dataset's size. In total, the runtime estimate is $O(9N)$. (In an orthodox implementation) for memory efficiency, each digit's counting sort requires a size 10 array storing the count of each possible digit, and an output array is needed to store the temporary sorted array for each digit. In total, the memory usage is $O(N + 10)$, where the 10 is negligible enough to round it as $O(N)$. It is certainly a good approach, managing to both linear time and memory usage.

(c) The difference is that only 3 runs are needed now, but each run requires the creation of a size 1000 array storing the different possible keys. So the time used decreases to $O(3N)$, while memory grows to $O(N + 1000)$. Whether this new partition is better than the old one depends on the dataset's size. In general, the bigger the dataset, the more time this new partition would save compared to the one described in (b), and the more negligible the extra size-1000 array is compared to the dataset itself. But since the extra array is barely any memory on a modern computer, the 1/3 amount of time is almost always preferable. So the partition in (c) is better.

(d) See the folder `P3`. Note that `sort_builtin` is the final choice.

(e) I tried both partitions of radixsort (groups of 3 or 1), a standard 2-way mergesort, and the builtin Timsort. After some painful debugging,

```
test_all(["sort_merge", "sort_radix_1", "sort_radix_3", "sort_builtin"], verify=False)
```

it turns out that Python's list operations slow down the lienar time radixsort too much to make it worthwhile. Theoretically, as the dataset upscales, radixsort should be better in runtime due to its linearity, and that indeed is what has been observed. However, the upscaling is so slow (see this)

```
Testing 40.test
sort_radix_1: 915.3809547424316ms for 100000 lines of input, avg: 0.009153809547424317
sort_radix_3: 613.1720542907715ms for 100000 lines of input, avg: 0.006131720542907715
sort_builtin: 111.12308502197266ms for 100000 lines of input, avg: 0.0011112308502197266
...
Testing 45.test
sort_radix_1: 11507.652997970581ms for 1000000 lines of input, avg: 0.011507652997970581
sort_radix_3: 7954.3280601501465ms for 1000000 lines of input, avg: 0.007954328060150147
sort_builtin: 2309.2689514160156ms for 1000000 lines of input, avg: 0.0023092689514160156
...
Testing 50.test
sort_radix_1: 113530.2209854126ms for 10000000 lines of input, avg: 0.01135302209854126
sort_radix_3: 82253.83996963501ms for 10000000 lines of input, avg: 0.0082253839969635
sort_builtin: 32005.70011138916ms for 10000000 lines of input, avg: 0.003200570011138916
```

that it is doubtful whether or not radixsort would ever catch up, before the maximum dataset of 10^9 entries is reached. I attribute this slow upscaling to Python's implementation of list: since Python does not implement an array, but rather a vector-like List, the repeated allocation of memory slows down the whole sorting process and dulls the faster theoretical runtime of radixsort. Mergesort happens to be even more disappointing, being by far the slowest of all the methods. This is not surprising, as the builtin `Timsort` is an highly optimized form of $O(n \log n)$ sorting, and radixsort has the appeal of linear runtime. In conclusion, and rather unfortunately, my final submission is the `sort_builtin` function (which is just a wrapper of the builtin `sorted` function), as it is written in C and avoids the expensive list operations. If code directly written in C is disallowed, then `sort_radix_3` is the second best. The reason why it is faster than `sort_radix_1` is provided in part (c).

P4

To come up with the final design, I'm trying out a couple methods:

1. Call the default Timsort to sort first by number of occurrences of the most common character (which I shall refer to as the `count` of the string), then by the character itself (smallest of them if there are multiple) (let's refer to it as the `char` of the string), then by the strings themselves (`string` of the string).
2. Divide the data into bins based on the tuple pair (`count`, `char`), and then each bin becomes a fixed-length string alphabetical sorting problem. Recombine all the bins in the correct order at the end.

Regardless of the method adopted, we need a function to extract the smallest (with the ordering A < B < C < ... < Z) most commonly occurring character in string. Since the string only most likely lacks some of the 26 possible letters (its length is 1 to 40), the hashtable (a Python `Counter` in this case) would end up with fewer than 26 keys. I went through a few designs for this function

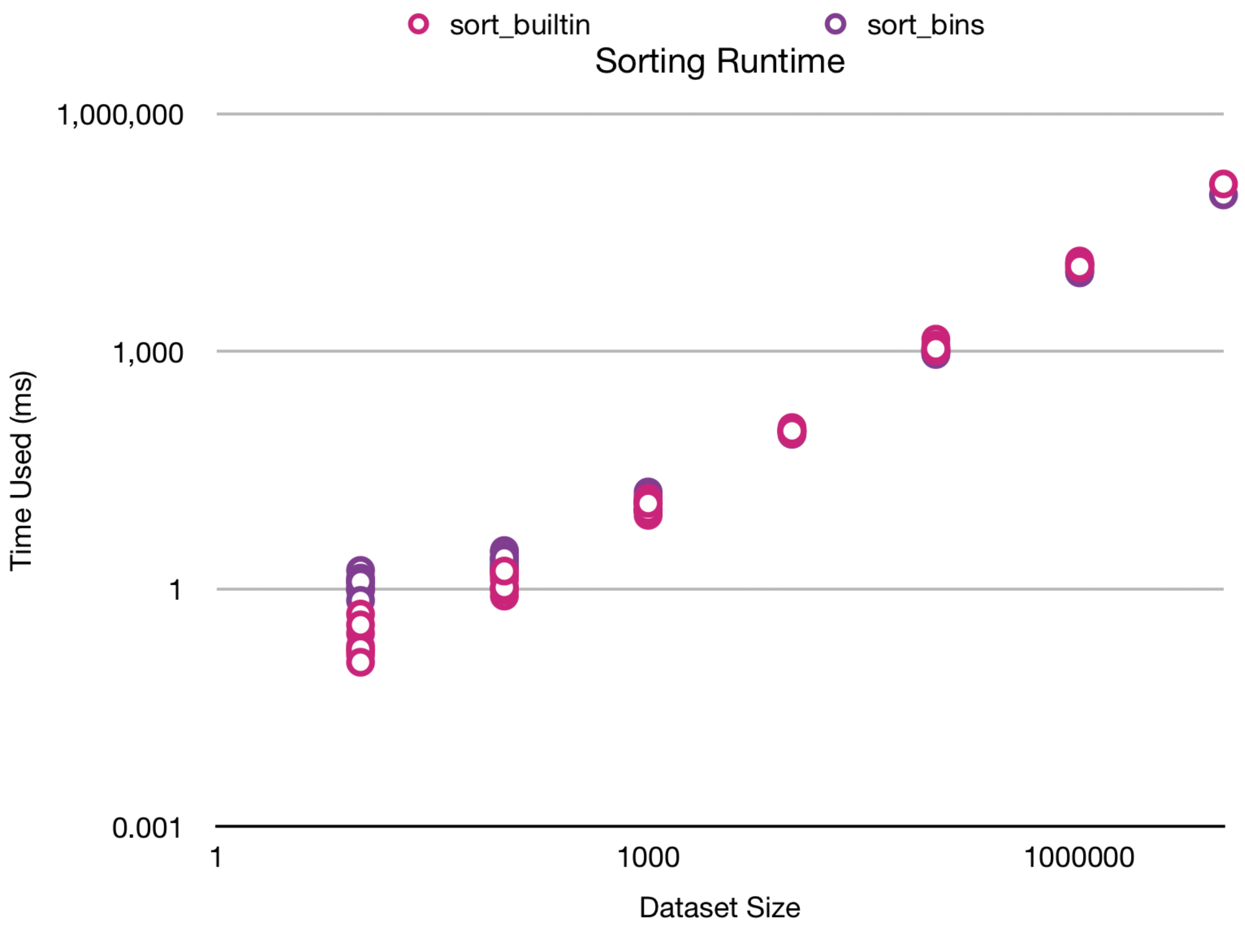
```python
# This is the first design (and what I ended up using)
def max_occur(d):
    from collections import Counter
    result = Counter(d)
    highest = 0
    current = ""
    for i in sorted(result.keys()):
        if result[i] > highest:
            highest = result[i]
            current = i
    return highest, current
# Note the need to sort the keys so that the smallest
# character is selected if multiple most common
# occurrences exist

def max_occur(d):
    from collections import Counter
    result = Counter(d)
    highest = 0
    current = "{" # ascii bigger than "Z"
    for i in result.keys(): # avoid sorting the keys
        if result[i] > highest or (result[i] == highest and ascii(i) < ascii(current)):
            highest = result[i]
            current = i
    return highest, current

def max_occur(d):
    from collections import Counter
    # first adding all the letters in order to ensure
    # if multiple most common occurrences exist
    # the smallest character is chosen
    result = Counter(string.ascii_uppercase)

    result.update(d)
    a,b = result.most_common(n=1)[0]
    return a,b-1
```

After a bit of testing, the first design proves to have the fastest runtime.

Now we move on to the fixed-length string sorting problem. We have a few options in approaching it: radix sort and its variants (e.g. MSD or LSD sorting) give great theoretical runtime, and we can always resort to the builtin Timsort for good practical runtime. Again, Timsort in any other language would be significantly less advantageous with its $O(n \log n)$ runtime, but Python cheats by having the builtin Timsort be written in C.

An experiment between the various methods can be found in `test_strings.py`. The result is that even though LSD radixsort is faster than Timsort for big sets of length 1 strings, it is quickly outcompeted by Timsort for any longer strings, as LSD radixsort's runtime linearly scales up with string length, while evidently the Timsort runtime increases minimally as string length increases. Although the MSD sort does not scale up nearly as much after length 20 is reached, the many list creations are so slow that the runtime is not worthwhile. In conclusion, the best method in Python would be to use the builtin Timsort.

Then we proceed to determine which of the sorting strategies (bin method or builtin) is better for various dataset sizes. The result from some rigorous testing is graphed here.



As the dataset size increases, `sort_bins` becomes faster than `sort_builtin`. After some close examination of these results, the cutoff seems to be at $n = 10000$. Therefore, the final submission first checks the input data size. If it's greater than 10000, we'll use `sort_bins`, and if it's smaller than the cutoff, we'll use `sort_builtin`.

The final submission method is `sort_combined`, which implements the above logic.

P5

Like `P4`, whatever the final method is, we'll need to write an `is_prime` function somehow. This is done in the `primes_adventure` folder. We consider a naive classic square root runtime (deterministic) test, Miller-Rabin (probabalistic) test, Solovay–Strassen (probabalistic) test, and AKS (deterministic) test. For this exploration, after reading up on the math, I just found preexisting implementations (cited inline) and modified them moderately to Pythonize, optimize, and tailor them to my needs. As it turns out, AKS is way too slow in actual implementation (having only nice theoretical runtime). Solovay-Strassen is eliminated after reading this paper, which pitches the two probabalistic tests considered here against each other and demonstrates that Miller-Rabin is faster and more accurate.

I proceeded to compare the runtime of the naive algorithm and Miller-Rabin (and its false-positive rates). We consider integers larger than a certain threshold N. Check out `primes_adventure/test.py`. Naturally, due to the difference in theoretical runtime, as the input number n grows, Miller-Rabin would become on average faster than the naive test. The cutoff found is 10000000. The false-positive rate of Miller-Rabin test by running it once is negligible enough the complications brought by it, though it needs to be addressed, would not be an significant impediment to runtime, since we can apply the process of elimination by confirming whether the rest of the integers are prime--most of the time they will be reported as composite (and Miller-Rabin's composite results are deterministic). Only when there are multiple candidates for the prime would a deterministic test be necessary. However, the naive test is much faster for large composites with small factors, and blindly applying Miller-Rabin, though it is faster on average after $N = 10000000$, is clearly not optimal.

The next step, of course, is to implement a preprocess function for a data entry to find the position of its prime in the 5 integers. Many possible approaches, such as prime testing numbers smaller than the threshold with naive test and ones bigger than the threshold with Miller-Rabin, and optimizations, including quasi-simultaneous checking of all numbers at once with naive test, prioritizing smaller numbers for the sake of elimination, switching out of naive test after a certain factor is tested, sorting the numbers and starting with small ones, etc.

As a good first step, however, let's implement a simple algorithm (`preprocess_pure_miller`) that employs the process of elimination and exploits the deterministic nature of composites of the Miller-Rabin test:

> Run one round of Miller-Rabin test on all five numbers. Eliminate those returned as Composite. If there is only one number left, it is the Prime. Otherwise, repeat this step.

Now, let's try out using the naive approach (`preprocess_naive_simul`), but quasi-simulatenously process all candidates. Naturally this will still be slow for cases with both large primes and large composites, but it's good to verify.

As always, let's run some tests.

```
log_preprocess = range(30) test_all_preprocess(["preprocess_pure_miller",
"preprocess_naive_simul"], verify=False, files=[str(i).zfill(2) for i in tests])
```

Before attempting to optimize preprocessing any further, let's get an idea of what the limiting factor is:

preprocessing or sorting.

For a fast-and-easy attempt at sorting, let's define a `sort_builtin` calling the default Timsort on the tuple `(prime_index, prime, ints)`, which would sort based on the first item of the tuple, and the second should the first be equal, and so on.

After some testing,

```python
# modify the range at will for some fun
# I tried range(20), range(60,62), range(90,91)
tests = range(60,62)
log_preprocess = test_all_preprocess(["preprocess_pure_miller"], verify=False, files=
[str(i).zfill(2) for i in tests])
log_sort = test_all(["sort_builtin"], verify=False, files=[str(i).zfill(2) for i in tests])
print(log_preprocess)
print(log_sort)
```

it is clear that the majority of the runtime is spent on the task of preprocessing: sorting takes less than 10% of the runtime. Though it should be noted that as the dataset size increases, sorting becomes more and more significant (due to sorting's $O(n \log n)$ runtime as compared to preprocessing's $O(n)$ runtime).

So we shall return to improving the preprocess function. Using the cutoff found earlier, define `preprocess_split` so that numbers are checked by naive test if they are smaller than $N = 10000000$, and those greater than that are checked (only for composites) by Miller-Rabin. Some tests

```python
log_preprocess = test_all_preprocess(["preprocess_pure_miller", "preprocess_split"],
verify=False, files=[str(i).zfill(2) for i in tests])
```

show that this approach shaves off about 20% of the runtime. Also, let's shave off some more time by preloading the primes less than $\sqrt{N}$. Next up, we can try the get-lucky method first on all the numbers, in case they are multiples of 2 or 3. All the optimizations result in a preprocess less than half the pure_miller runtime with no complex composite.

With preprocess handled, let's quickly move back to optimizing the actual sorting. A counting sort algorithm would be great, with bins based on the prime positions. Then within each bin, the sorting remains the same (use tuple `(prime, ints)`). This turns out to only speed up the result marginally. I will opt for it for the sake of scalability.