Dalveer Singh Dosanjh

301397398

dsd6@sfu.ca

Comparative Programming Languages

CMPT 383

Fall

D100

The differences in working with Python, Racket, and Haskell

Implementation

Python

First, lets talk about implementation. In my mainstream language implementation (python) I first needed to get my text to parse the file contents so its readable by my code. So to do this, I split on every space (A B) -> (“A”,”B”), now that I have a readable line, I basically have a list with each item in the list being a vote for a candidate. Second, I needed to remove any possible duplicates so that I don’t need to worry about counting twice (“A”,”B”,”A”) -> (“A”, “B”). Now that everything is ready, I am ready to count.

To count, I needed to find what I was counting, so I created a list that had every possible vote in it so this list would only have 1 of every vote, to do this, I just needed to create a list, if a letter appears that is not in the list, put it in the list. Now this list I have just created are all my keys to my dictionary. Now my dictionary can use those same keys to have a variable and increment it each time it encounters the key. While I am counting my votes for each candidate, I am also counting the number of “none” votes. If there is no other vote being casted, I assume it’s none, incrementing my none each time. Then, I just print all my data, except full, as I must calculate it still. To count full votes, I just check how many times the list of all possible votes show up in the file, then I print it. To print my data in a sorted order, I just sort my dictionary just before I print the whole dictionary.

Racket

Trying to stay loyal to my python implementation was insanely difficult. As it was possible, parsing the data into readable lists into (“A B”)->((“A”,”B”)). Also, in python, I did everything as I read line by line, this wasn’t possible as I read the whole file at once in Racket. In racket, implementing it the same as my python gave exponential time and performance issues. It would take more than 60 seconds to execute the “example100000.txt” and I was never able to finish the “example1000000.txt” even though I waited hours. The problem was parsing and changing (“A B”, “C”) into ((“A”,”B”),(“C”)), and a for loop was just too inefficient. So, I decided to change my for-loop into a recursion, and this was also slow. After many different implementations of for loops and recursions, I finally just switched to mapping, which was different from my mainstream implementation. Map worked wonders in racket, I had many ideas to make it extremely efficient If I were to change everything to use map and filter, but in-order to stay loyal to my mainstream implementation, I stuck with for-loops. The reset of the implementation was pretty similar logically, but this time I had to manually make a dictionary which was just a list that had the candidate and value ((“A”,3)(“B”,2)), to do this it was similar to making my dictionary, instead I had my list of all possible votes (“A”,”B”,”C”) and I had the list of all my values (3,2,4) and I paired them. Then to count my votes, I just checked how many times the candidate appeared, for none, I checked how many “none” appeared. Finally, I sorted and printed it, however, this time I counted my full votes just before I printed, but I already had all my data.

Haskell

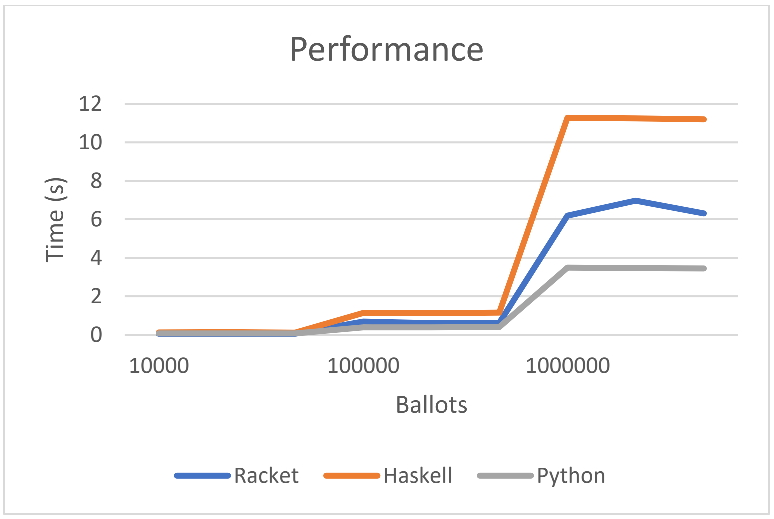
For-loops in Haskell are non-existent, so unlike racket, where I tried to keep some for-loops like my mainstream implementation, I had zero for-loops, and recursion instead. I used map and filter to parse my data, and I used zip, but since my sort function needed the integer to be first my theoretical dictionary was [[1,”A”] [3,”B”]] instead. Basically, I logically followed my mainstream and racket implementation, but it utilized maps, filters and, lambdas instead of for loops. Then, to print my data, I used recursion.

Performance

Python was 88% faster than racket and 225% faster than Haskell. Python was tested in a native IDE by a time function that returned the current time, then calculating this time at the end. Racket was also tested in a similar way, but it had a built in current-millisecond function, it was also run on a native IDE. My Haskell was run on the windows 11 subsystem for Linux, which could have negatively affected its run, but this was tested using the ghci's built in :set s+ function. Based on this data, python is the most efficient in parsing files.

Performance of Languages in seconds

Ballots	10000			100000			1000000		
Racket	0.0596	0.062	0.055	0.68	0.61	0.62	6.19	6.97	6.31
Haskell	0.12	0.13	0.11	1.14	1.12	1.15	11.28	11.25	11.19
Python	0.071	0.076	0.072	0.38	0.39	0.4	3.49	3.46	3.44



Ease of development

Python was the most enjoyable to program in, but this is heavily affected by my experience with python and other similar programming languages. I have a lot more experience with these languages, and I have done similar file-parsing, so this is a massive reason why I enjoyed it so much. I probably could have implemented it faster, but I haven't used python in a while. For racket and Haskell however, I have only so much experience with it, and as my experience was recent, I was getting lost on what functions exist and what I could do with it. I searched up and looked through notes for Racket and Haskell a lot longer than I did for python. For these reasons, working with Racket and Haskell where not as enjoyable.

Python only took me a couple of hours due to my experience, but for racket and Haskell, it was very different. First, when I initially implemented my racket, it was done in about 3 hours, but when I tested it on 1,000,000, it would take too long. Narrowing down this issue took me an additional 3 hours, I changed my for-loop to a recursion, this gave me the same issue, then finally I switched to map, which fixed the issue. After I learned of map, I wanted to redo my entire implementation and make it use map and other higher order functions, but I wanted to stay loyal to my mainstream implementation. Then, Haskell took me a bit long as every for-loop had to become a recursion, but I already had learned of map through Racket, and this saved me a considerable amount of time, making it shorter than Racket.

I felt like I could do a lot more with python. I could work with strings in any way, and I was just working with my output. In Racket and Haskell, I felt more restricted, I couldn't create simple loops, and mindlessly implement a simple rendition of my python implementation, instead I was forced to use higher order functions and functional programming. During my implementation of both Racket and Haskell, I was very unhappy, but afterwards I felt like I learned a lot from Racket and Haskell, and it showed me that I could make my python implementation even better! Another issue was researching my issues, in python I only had a single issue, and that was how to sort a dictionary, I was instantly met with a quick solution, and it helped me complete my solution. Racket and Haskell were not the same story, learning how to read files was confusing and it was hard to find examples online.

	Racket	Haskell	Python
Ease	6	5	8
Knowledge	6	6	8
Flexibility	6	7	9
Community	6	7	10
Readability	2	5	6
Total	5.2	6.2	8.4

	Racket	Haskell	Python
Time (hrs)	6.5	3.5	2
Debugging (hrs)	3	0.3	0.2
Lines of Code	79	43	47
Functions	4	3	0

Quality of source code

I always considered python to have terrible readability, because it was hard to read indents, so looking at what function nested what code was very difficult. However, this changed when Racket came along. With the most lines of code, most functions, and the least readability, it was very difficult to read racket functions. Even though I was more understanding of Rackets notation by now, I still found it very difficult to re-read my program. Rackets brackets just made everything extremely unreadable. I tried to treat them as braces like other mainstream languages, but it was still unreadable due to the brackets within brackets. For complicated programs like this, Racket is just unreadable. Finally, Haskell was not as bad as Racket since it was not littered with brackets, but I was unexperienced with its indents and functions, so it was weird to read which made me score it lower for readability.

Conclusion

To conclude, Python is the language I would recommend. Although Racket and Haskell can give you a new perspective on coding and how to confront problems, Python can do a lot more and is just more flexible. Even though Python is not my go-to language for problems, I give it full credibility when it comes to files. Python is extremely easy to use when it comes to parsing files and text. Not only is Python exponentially faster than Racket and Haskell for files, it's like other mainstream languages, so you can think the same way as you use other languages. I must give credit to Racket and Haskell though, once I truly understood the map and filter function, I wanted to write everything using map and filter functions. There is a learning curve when it comes to using Racket and Haskell which makes me unable to recommend this as much as Python. Finally, Python's versatility, Ease of use, and simple structure makes it an easy recommendation over Racket and Haskell.