In 2020, I did a co-op at Purefacts Financial Solutions in our machine learning R&D team. One problem we spent a lot of time talking about was the memory footprint of our algorithm, which was so big that in it's first test with a real client's data, 64 GB of ram hadn't been enoguh, and the program didn't work. Because of this we spent a lot of time optimizing our code. And yet, the code was in Python, a language well-known for being rather inefficient. We made our program work, but I always wondered how much performance we lost by our choice of programming language.

The question I want to answer in my project is a generalization of the above: how much does one's choice of programming language affect performance? Everyone knows that C is faster than Python, but how much faster? And where do other languages fit in?

1 Languages

The following languages will be evaluated:

- Python, because it's very popular, and because it's the language that started my interest in the question I'm answering. It represents languages that are not particularly designed for performance.
- C, because it's supposedly good for high-performance applications. It represents non-garbage-collected languages.
- Java, because it supposedly has very good performance, while still having garbage collection. It's also one of the most popular languages in the world.
- JavaScript, **for extra credit only**, because it is a very popular language. It's also the only mainstream language this is more closely related to Lisp than to C.

2 Benchmarks

I want to evaluate each language on general-purpose tasks: things that all languages can do, and that no language particularly specializes in. All inputs will be random, and all benchmarks will be averaged over multiple runs.

2.1 Naïve Matrix Multiplication

Matrix multiplication is a common benchmark in high-performance computing and is a useful task in many applications. It's memory-intensive but has somewhat good caching perfermance due to the storage of data in 2d arrays.

I will implement this multiplication myself, based on the following psuedocode:

```
matmul(matrix a, matrix b, matrix c) {
    for (i = 0; i < N; i++) {
        for (j = 0; j < N; j++) {
            c[i][j] = 0;
            for (k = 0; j < N; k++) {
                c[i][j] += a[i][k] * b[k][j];
            }
        }
    }
}
```

No manual optimizations will be applied, but all automatic optmizations will be turned on. For example, C code will be compiled with `-O3`.

2.2 Optimized Matrix Multiplication

In addition to seeing what a language can do in typical use, I'm also interest in what it can do when it has been optimized to the highest extent possible. I'm again using matrix multiplication for this benchmark, since it's common enough to have highly-optimized libraries in each language I'm using. Specifically:

Python Numpy

C OpenBLAS

Java EJML

2.3 Sorting

Another common task in software is sorting a list. This is a performance-intensive task, and is included in the standard libraries of each language I am evaluating. This allows me to evaluate the efficiency of standard library functions (as opposed to 3rd-party libraries evaluated in matrix multiplication). And because the functions under evaluation *are* standard library functions, I expect the cost (in development hours) of implementing this additional benchmark to be very small.

2.4 Heap

The above benchmarks all use data organized in arrays. Arrays are good for caching since they have high spatial locality. But I'm interested in what happens when operating on a pointer-heavy data structure, where closely related blocks of data may be far apart in physical memory address. I will do this by implementing a heap as a binary tree.

Each node in this binary tree will have a value and up to two children, referenced by pointers. When new values are added to or removed from the heap, many of these pointers will have to be changed to maintain the invariant of the heap.[1] This should result in a rather chaotic memory allocation pattern.

To exercise this data structure, I will use it to sort an list. I will generate a list of integers, insert each integer into the heap, then pop them all into a new list.

3 Analysis

For each program, I will analyze runtime and memory usage. Runtime will be evaluated using wall-clock time, measured by the program under test itself. Memory usage, including peak memory usage and cache hit rate, will be evaluated using Valgrind and possibly other language-specific tools if doing so presents an apparent benefit.

Note that runtime and memory usages will never be evaluated simultaneously. Valgrind adds overhead, which will make time measurements inaccurate.

Runtime will be evaluated on only part of each program's operation. Specifically, setup such as generating the random inputs will not be measured. Since Valgrind is an external program, this selectivity is not an option for memory analysis.

4 Deliverables by Grade

F φ

D Raw results computed for 1 languages, only for some benchmarks.

D+ All of the above, but with benchmarks.

C- Raw results computed for all languages[2] for all benchmarks.

C All of the above, plus results formatted in readable visuals and tables.

C+ All of the above, plus a written summary.

B+ All of the above, plus superficial analysis of what the results mean.

B All of the above, plus analysis of what the results mean.

B+ All of the above, plus thoughtful analysis of what the results mean.

[1] The invariant of a heap is that each node's value is smaller than that of each of it's children.
[2] Except extra credit languages.

A- All of the above, plus analysis of at least one unexpected result if one was present, such as theorized reason.

A All of the above, but speculation on unexpected result (if one was present) is backed up by reference to external documents (i.e. papers or language documentation).

5 Expected Results

My above deliverables mention how I will respond to unexpected results, so I think it's important I write down my *expected* results before I begin.

- The runtimes and peak memory usage of different languages on the same benchmark will differ by less than an order of magnitude.
- C will be the fastest on all benchmarks. This is because of all languages evaluated, only C is pre-compiled to raw machine code which does not require a software interpreter (like the Python Interpreter or the JVM[3]). These interpreters add a layer of indirection between the compiled code and the hardware, which will take time to evaluate.

  Additionally, Java and Python both use garbage collection, which makes development easier at the cost of runtime overhead.
- Python will be the slowest on all benchmarks, slower than C by at least a factor of 2. Like Java, Python will be slowed by its interpreter and by garbage collection. However, Java is compiled to it's own bytecode, which should mean it needs to do less parsing at runtime than Python's interpreter.
- Java will be closer in speed to C than to Python.
- The ratio of the runtime of one language to another will be roughly consistent between benchmarks.
- As an exception to some of the above, Python and Numpy will perform very well in the optimized matrix multiplication. Numpy, though a Python library, is implemented partially in C++, which I expect to result in a considerable speedup. I believe it is more likely than not that Python will be faster than Java in this benchmark. I believe it is possible, but not particularly likely, that Python and Numpy will approach the performance of C and OpenBLAS.
- C will have better memory performance (lower peak memory usage and higher cache hit rates) in all benchmarks than the garbage-collected languages (Java and Python). This is due to the overhead introduced by garbage collection.

[3] Java Virtual Machine

- Java and Python will have similar memory performance.
- Regarding memory performance, the garbage-collected languages (Java and Python) will fare particularly poorly in the heap benchmark. This is because the pointer-heavy nature of the data will require more garbage-collection overhead.