EVALUATING LANGUAGES FOR BIOINFORMATICS: PERFORMANCE, EXPRESSIVENESS, AND ENERGY

Thesis Defense

Randy J. Ray
December 2022

The UNIVERSITY of OKLAHOMA

Evaluating Languages for Bioinformatics

- One of the fastest growing concerns in the technology sector is the increased demand for power in the world's data centers
- As the data center industry continues to expand, so to will power usage, and therefore the need for increased energy efficiency in software development
- A methodology that evaluates a set of programming languages based on three key metrics: performance, expressiveness, and energy use is needed
- The framework presented takes a collection of string-matching algorithms used on DNA sequences to demonstrate the capabilities of each language, and draw out their distinctiveness
- DNA sequencing was chosen due to its growing uses and applications as technology evolves and makes such sequencing faster and less expensive
- The methodology presented here will show that using a newer language like Rust has advantages that help it balance speed, ease of use, and power consumption when used for advanced scientific computing

Motivations & Real World Applications

- Having started out with the BASIC language on a personal computer, the introduction of Pascal showed me that there were many other languages and that their syntax and structure could vary wildly
- Pascal led to C, with academic diversions into Lisp and Fortran during my undergraduate study
- Working in the software industry brought about experience with a long list of other languages, ranging from interpreted scripting languages (Perl, Python, Tcl, Ruby), to web-development (PHP, JavaScript), to Java and languages built on the Java Virtual Machine (Clojure, Scala)
- Along the way, one constant remained: an interest in comparing languages not only on speed but also on readability, expressiveness, and capability

Real World Implications: Power Consumption

- We are rapidly approaching an inflection point in computer science
- We have been relying on being able to increase computing capacity and capability endlessly
- Limits are going to be placed on our computing power purely from environmental concerns — which means efficiency has to extend beyond hardware into our software programming practices as well
- Performance alone is not enough — we have to consider energy use and efficiency

Real World Implications: Security Concerns

- A large percentage of security vulnerabilities discovered in programs are traced back to memory related issues
- Google & Microsoft software engineers are quoted as attributing roughly 70% of serious security bugs to memory management and safety
- Even the NSA recommends developers switch to programming languages that feature greater memory safety
- But in terms of evaluating memory safety, it is hard to quantify other than whether vulnerabilities have been fixed or not

Evaluation Methodology

- Evaluation of the languages was focused into three areas:
    - Performance
    - Expressiveness
    - Energy Efficiency
- Hypothesis was that these areas were distinct enough to provide a broad and fair assessment of each language
- Performance & Energy Efficiency were evaluated with direct experiments
- Expressiveness was evaluated by analysis of the code
- A dedicated machine was set up and a harness application was developed

Languages & Algorithms Selected

- C & C++
  - Foundational compiled languages in widespread use
- Perl & Python
  - Interpreted languages;
    Perl used early on in the Human Genome Project and Python is popular within Bioinformatics
- Rust
  - Newer compiled language focused on safety

- Knuth, Morris, and Pratt
  - Foundational to the development of string matching algorithms
- Boyer and Moore
  - Performance improvements on KMP and longevity in use
- Bitap
  - Treats both pattern and sequence as strings of bits and never actually does character comparisons
- Aho and Corasick
  - Introduces multiple pattern matching and uses a DFA approach

Implementation

- Each algorithm was written as similarly as possible in each language
- Implementation stressed readability as an important factor in software engineering
- For C and C++ three different compilers were used to see how the performance compared to each other
- Experiments did not produce as much source code as would have benefited expressiveness measurements, but provided a reasonable “apples to apples” comparison

Creating the Novel Algorithm - DFA Gap

- Much of DNA sequence alignment is approximate matching
- Inspiration for the novel algorithm comes from a combination of regular expression experience and the Aho-Corasick algorithm
- Used a Deterministic Finite Automaton (DFA) with extra states as an optimized regular expression
- Created an approximate matching method with the ability to hold the gap between characters to a specific maximum

DFA Gap - How it works

- DFA involves two stages-
  - Building the DFA from the pattern
  - Applying it to the target sequence
- Building can be done in linear time with a predictable number of states
- Applying it takes similar time complexity as other approximate matching algorithms
- The DFA finds approximate matches in a non-greedy manner without backtracking
- It provides the starting point and specific ending point of its matches with no further calculation

Regular Expression Variation

- Perl and Python performed extremely slowly in the DFA Gap
- A regular expression variation was implemented to compare with the hand-crafted DFAs
- This improved the interpreted languages performance
- Surprisingly, the performance declined for all the compiled languages
- Provided an important comparative performance test

Evaluating Performance

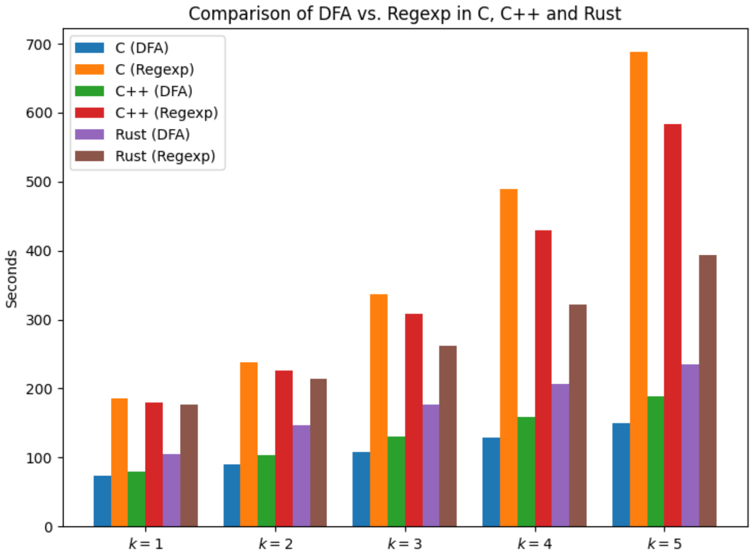
- Performance was measured by raw execution time
- Two values were taken for each experiment
  - The total execution time
  - Time spent on the algorithm (shown)
- A total of 14 experiments were run
- DFA and Regular Expression Gap Algorithms were run with a k=1 to 5

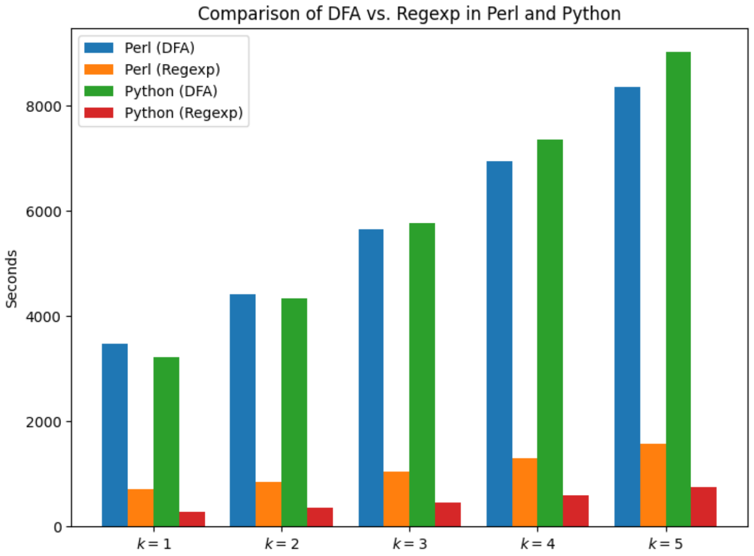
Combined Runtimes of all Experiments

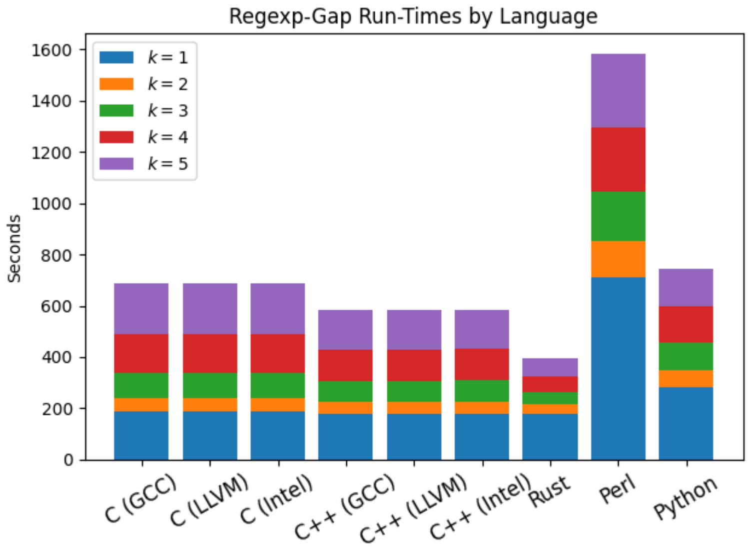
Language	Runtime	Score
Rust	2321.58	1.0000
C++ (GCC)	2451.74	1.0561
C++ (LLVM)	2482.64	1.0694
C++ (Intel)	2499.37	1.0766
C (GCC)	2541.39	1.0947
C (LLVM)	2566.72	1.1056
C (Intel)	2606.27	1.1226
Python	36503.46	15.7236
Perl	38234.67	16.4693

Runtime measured in seconds

Regexp Performance Results



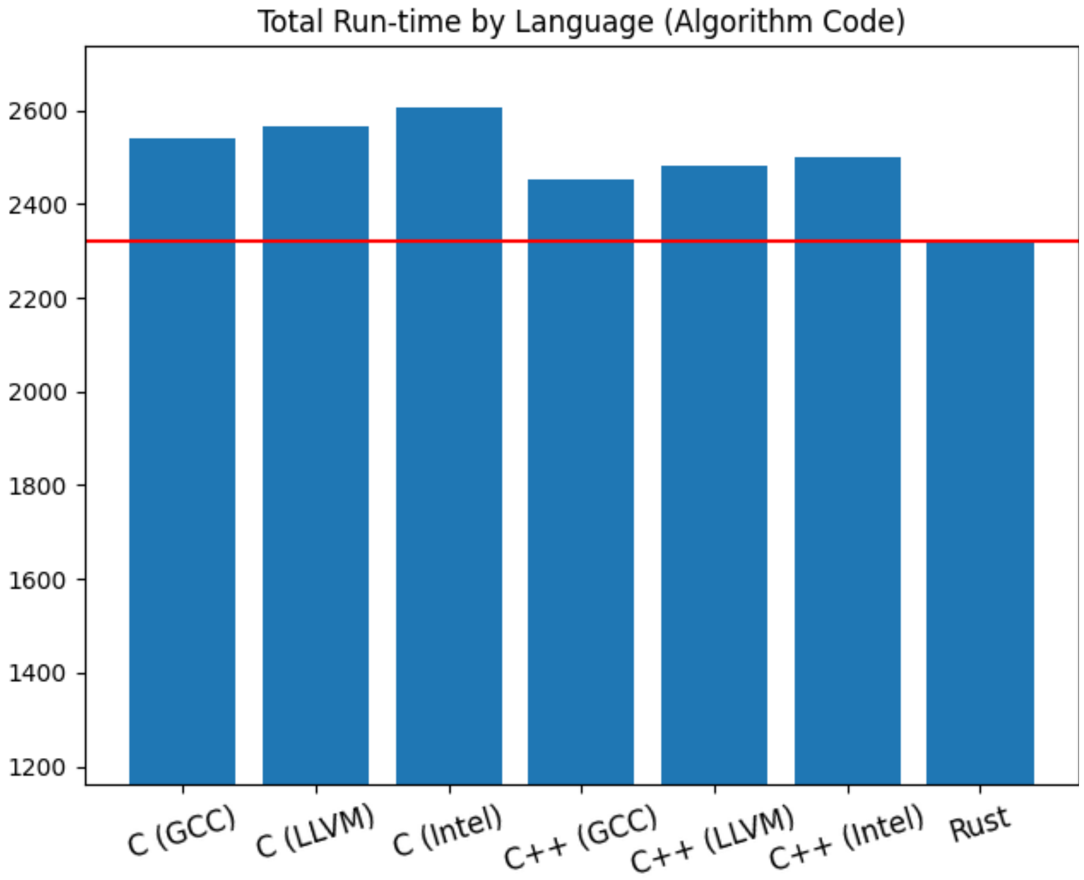




- Reverse trend in performance between compiled and interpreted languages
- Regular expression helped Python approach the compiled languages run times
- In all the compiled languages, the DFAs outperformed the regular expressions

Performance Results

- Rust was consistent across all experiments
- Considerable variation in compilers used for C and C++
- Interpreted languages were slower than their compiled counterparts by an order of magnitude



Evaluating Expressiveness

- Expressiveness was the most challenging of the metrics to measure and evaluate
- Three comparison points:
  - Source Lines of Code: Total SLOC values for each language’s files were combined. In the case of C and C++, this includes relevant lines from the header files for the runner and input modules.
  - Complexity: Each file’s cyclomatic complexity values were computed on a per-function basis. Each module’s function scores were averaged, and all modules’ averages for a given language were summed together.
  - Conciseness: Each source code file for a given language was stripped of comments and then merged into a sort of “archive” using the standard “cat” command available on Linux. Each such resulting file was then compressed with the “xz” compression utility and the compression ratio recorded.

Source Lines of Code

- Here Python is the clear leader, with Perl being 40% larger
- C++ maintains the third-place ranking across all three sub-tables
- Rust edges out C in both the framework measurement and the total lines measurement

(a) Algorithm lines

Language	Code	Score
Python	272	1.0000
Perl	376	1.3824
C++	403	1.4816
C	528	1.9412
Rust	543	1.9963

(b) Framework lines

Language	Support	Score
Python	148	1.0000
Perl	211	1.4257
C++	269	1.8176
Rust	272	1.8378
C	353	2.3851

(c) Total of lines

Language	All	Score
Python	420	1.0000
Perl	587	1.3976
C++	672	1.6000
Rust	815	1.9405
C	881	2.0976

Cyclomatic Complexity

- The value is based on measurement of control structures such as conditional statements, loops and similar means of changing the path of execution through the program or function being measured
- Here the contest between the first and second rankings was between Python and C++

(a) Algorithms complexity

Language	Total	Avg
Python	76	19.57
C++	81	16.83
Perl	106	26.97
C	114	18.30
Rust	132	17.97

(b) Framework complexity

Language	Total	Avg
C++	43	10.75
Python	47	9.90
Rust	58	5.43
Perl	61	12.90
C	76	19.00

(c) Total complexity

Language	Total	Avg
Python	123	29.47
C++	124	27.58
Perl	167	39.87
C	190	37.30
Rust	190	23.40

Conciseness: Comparison of Compressibility

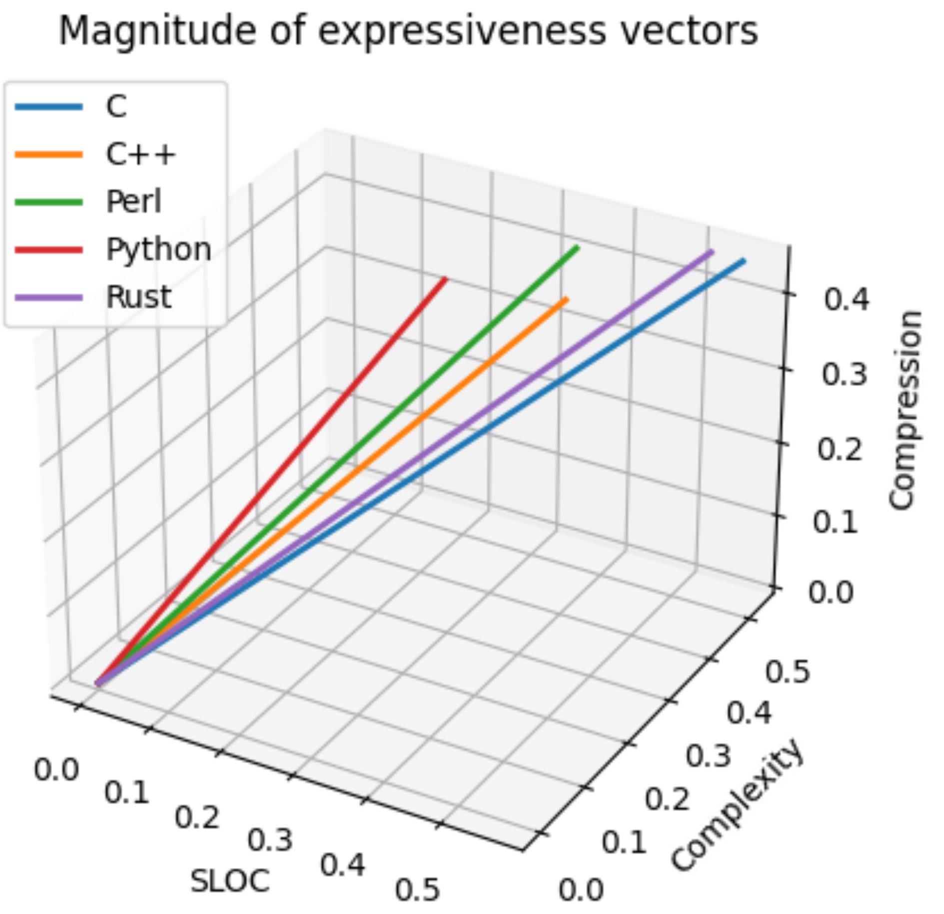
- The ranking of Python and Perl as the first two is expected
- However, the placing of C ahead of both Rust and C++ came as a surprise given the presence of highly repetitive calls to library routines for the manual memory management

Language	Ratio	Score
Python	78.50%	1.0000
Perl	80.50%	1.0255
C	80.60%	1.0268
Rust	80.80%	1.0293
C++	81.00%	1.0318

Overall Expressiveness

Language	SLOC	Complexity	Compression	Score
Python	1.0000	1.0000	1.0000	1.0000
C++	1.6000	1.0081	1.0318	1.1565
Perl	1.3976	1.3577	1.0255	1.2109
Rust	1.9405	1.5447	1.0293	1.4086
C	2.0976	1.5447	1.0268	1.4503

Language	SLOC	Compression	Score
Python	1.0000	1.0000	1.0000
Perl	1.3976	1.0255	1.1406
C++	1.6000	1.0318	1.2159
Rust	1.9405	1.0293	1.3446
C	2.0976	1.0268	1.4070



Evaluating Energy Efficiency

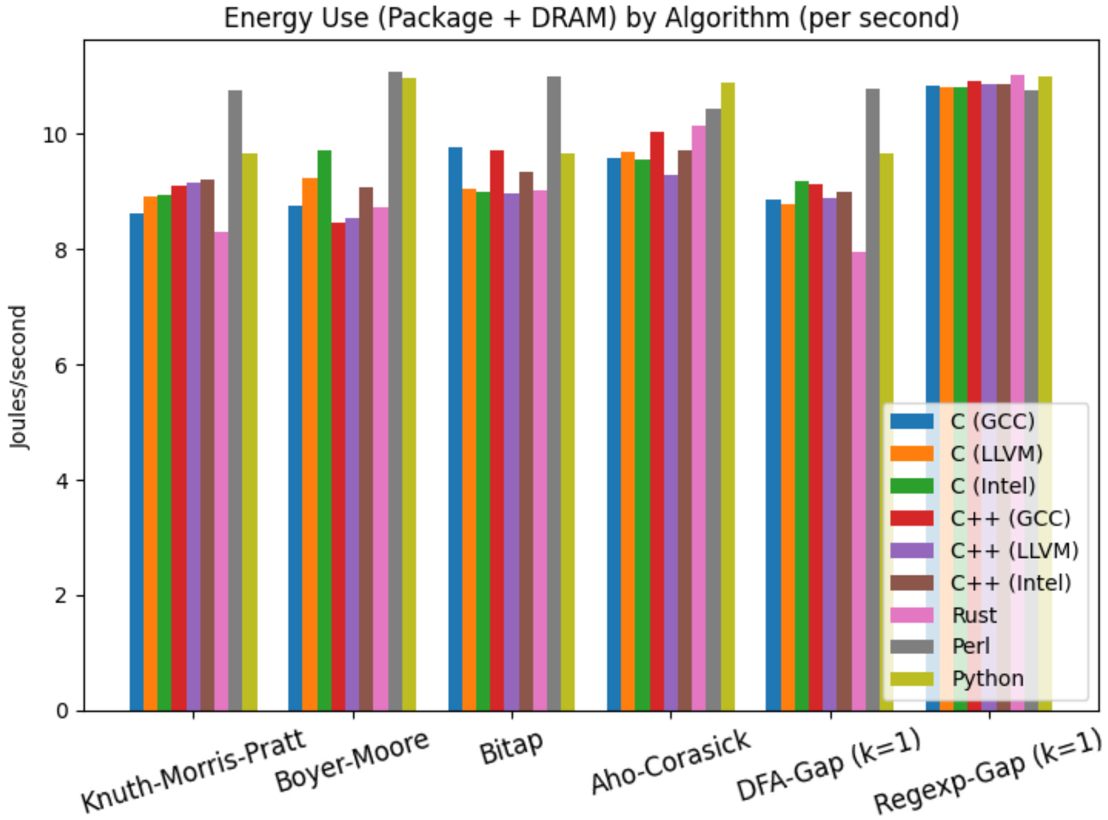
- The Running Average Power Limit (RAPL) measurement system was used to measure power consumption
- RAPL is a feature of select Intel CPUs
- The only significant barrier to overcome was ensuring that the machine used to run the experiments would be sufficiently isolated so as to have the least amount of interference possible from other running processes
- The RAPL system ended up proving robust-enough to handle measuring programs whose run-times ranged from a few seconds to nearly three hours

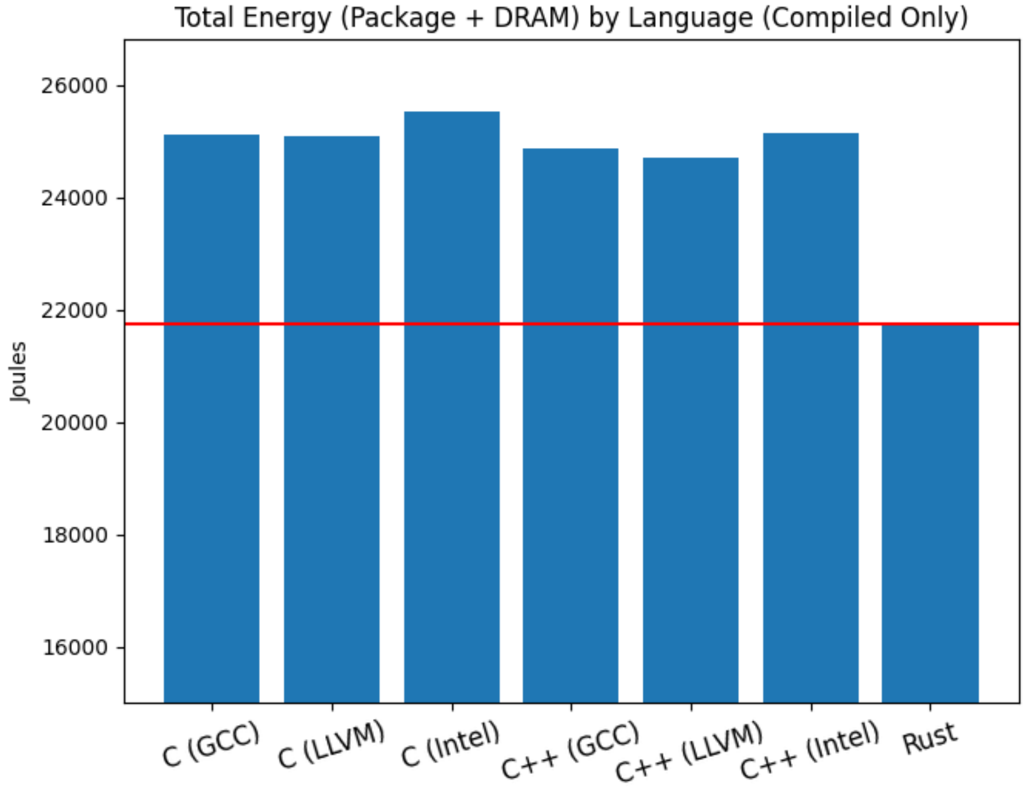
Combined Power Consumption of all Experiments

Language	Energy	Score
Rust	21756.11	1.0000
C++ (LLVM)	24691.83	1.1349
C++ (GCC)	24878.42	1.1435
C (LLVM)	25074.22	1.1525
C (GCC)	25115.48	1.1544
C++ (Intel)	25130.75	1.1551
C (Intel)	25528.86	1.1734
Python	357076.12	16.4127
Perl	392680.44	18.0492

Runtime measured in Joules

Energy Efficiency Results





- Rust scored the lowest power consumption for all five variations of the DFA-Gap algorithm, making it the most-efficient language for 6 of the 14 distinct groups of experiments

Final Rankings

- The three metrics were combined, similar to how expressiveness was measured
- Rust outperformed in performance and energy efficiency
- Python outperformed in expressiveness
- C++ outperformed Rust when expressiveness was added
- Without complexity, Rust was only 2.9% behind C++ in the final ranking

Rank, with Complexity

Language	Score
C++	1.0000
Rust	1.2247
Python	1.6583
C	1.8930
Perl	2.2174

Rank, without Complexity

Language	Score
C++	1.0000
Rust	1.0290
Python	1.3933
C	1.5904
Perl	1.7823

The End Result

- The expectation was that this research would show how testing and evaluating via performance, expressiveness and energy could be done in a consistent and reproducible manner
- Across three complete runs of the experiments, the difference between run times and energy readings were consistently within 1% from run to run
- It is my hope that this methodology enables developers to make educated choices when selecting a language for a project
- Everything described here would be applicable to other languages as well, given similar data to work with

Potential of Rust

- Why Rust?
- Performance and power consumption were concrete measurements; language expressiveness needs development
- While memory security was not a metric used, Rust never had any memory errors when the code was checked with Valgrind
- As the youngest language, Rust is still developing; increased usage and experience with the language will drive improvements
- Rust shows great promise for scientific applications

Future Development of DFA Gap Algorithm

- The DFA Gap Algorithm was shown to be effective at approximate-matching while also being simple to implement
- Comparing it to other approximate matching algorithms would be an interesting exercise
- Using the structure of Aho-Corasick, it could potentially be extended to include multiple-pattern matching
- With the distinction of its approach to defining and constraining gaps, it can be further developed as an additional tool for researchers to use

Future Research Potential

- Additional algorithms would tune the methodology
- Larger source code examples would improve expressiveness measurements
- Given larger datasets, run times and energy use could be more closely correlated
- Examining the programs at the machine code level could give insights into energy usage patterns
- Adding a memory security metric would impact the evaluation

Major Takeaways

- Software engineers need to start planning for future energy budgets, environmental constraints, and challenges of scale
- Small energy savings are significant at data center scale
- By looking at performance, expressiveness and energy efficiency side by side, developers can consider the weights and balances of all three metrics to make decisions for future programming needs
- When energy efficiency is as important as performance, this methodology has shown its ability to clearly evaluate the suitability of a programming language

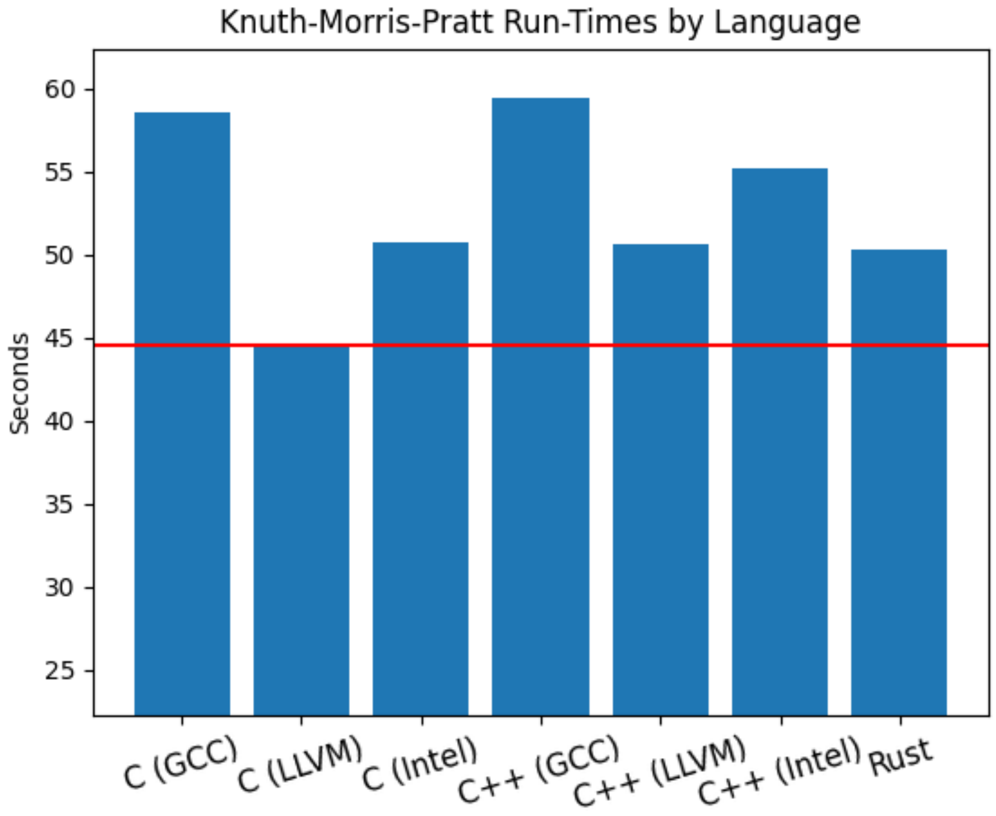
Q & A

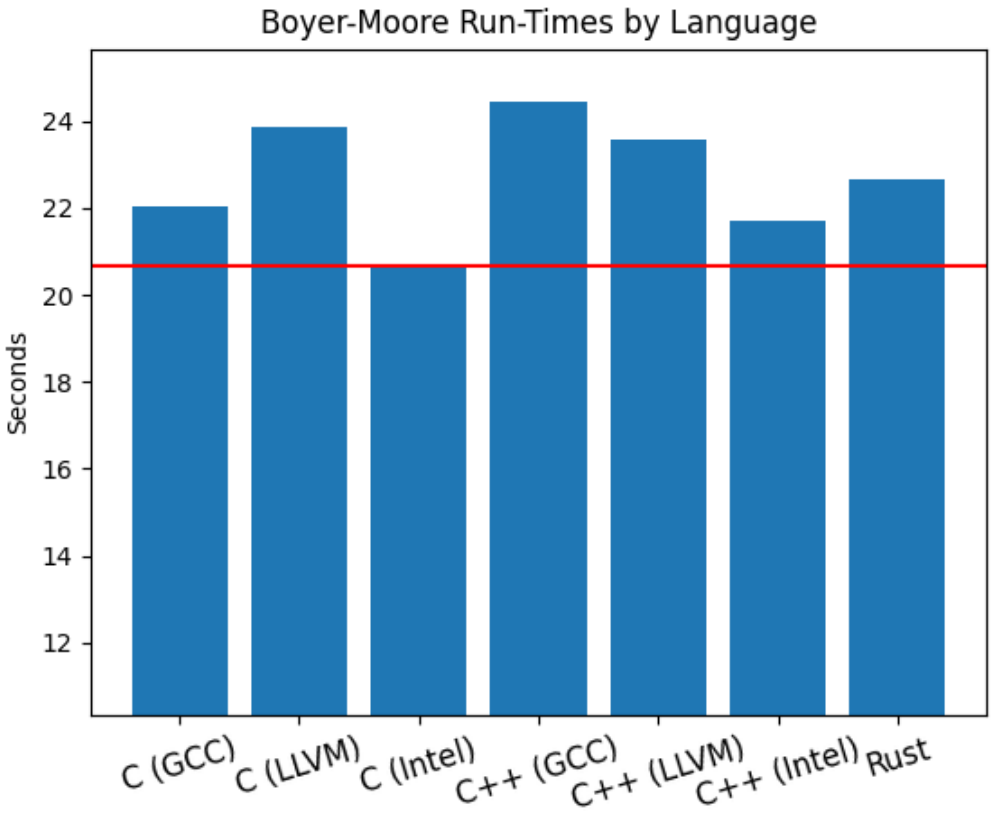
Thank You

https://github.com/rjray/mscs-thesis-project

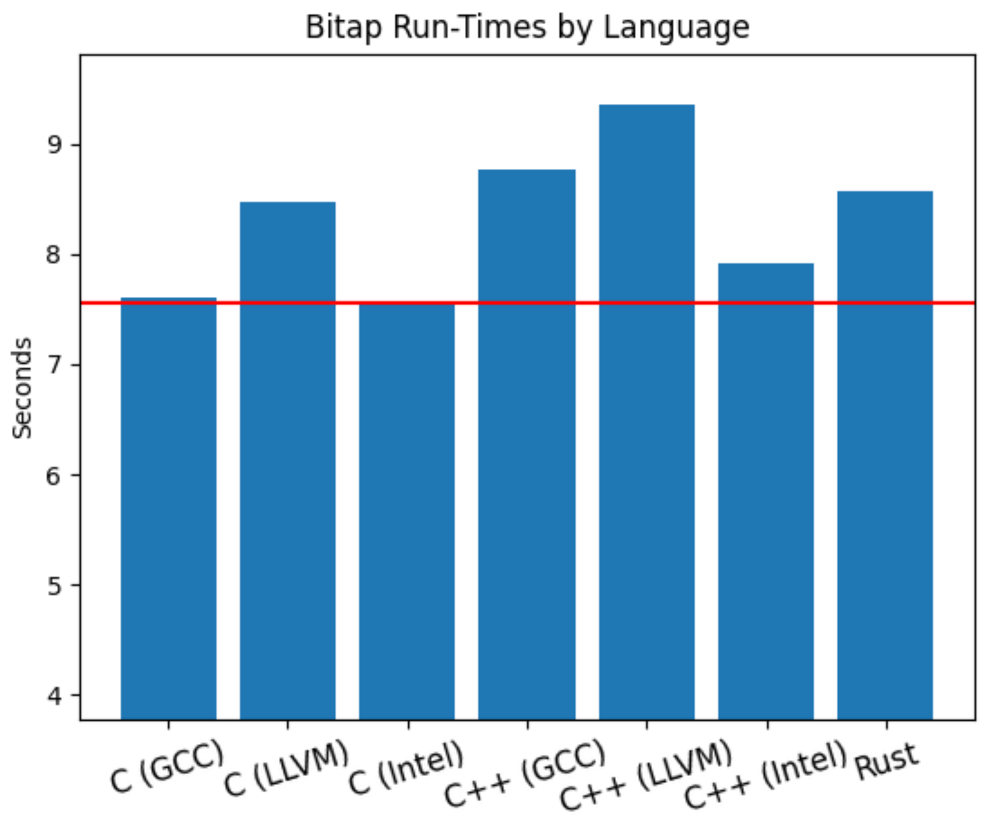
Appendices

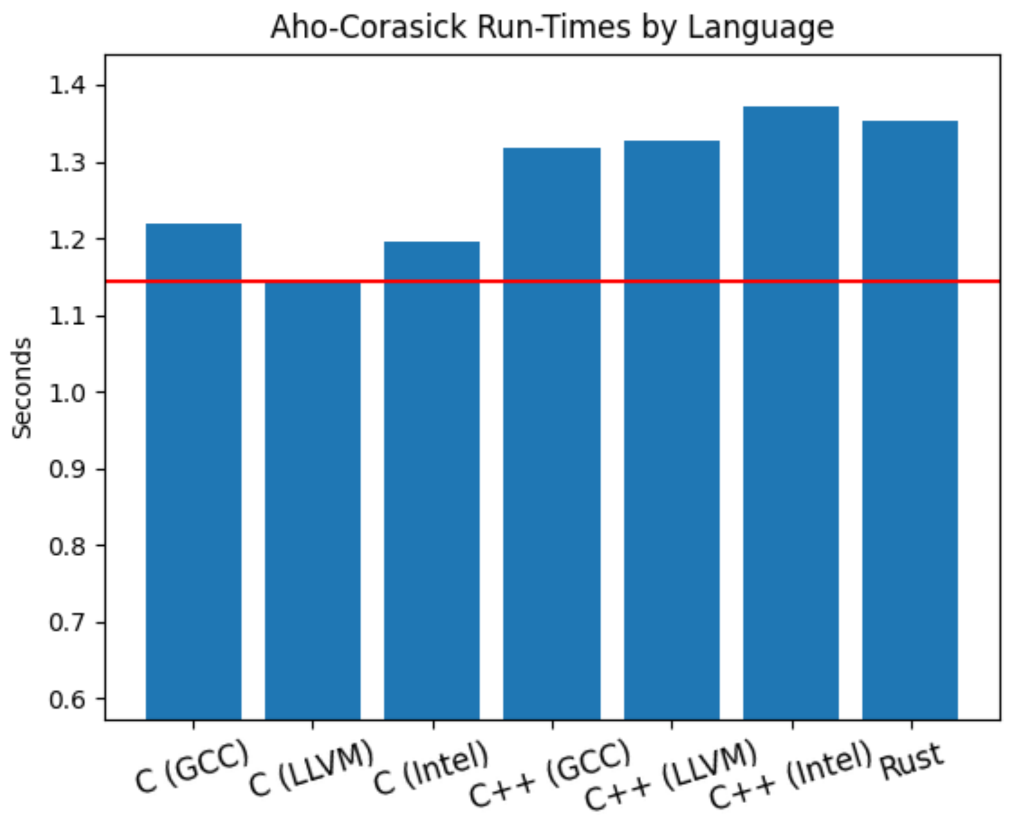
Exact Match Algorithm Run Times



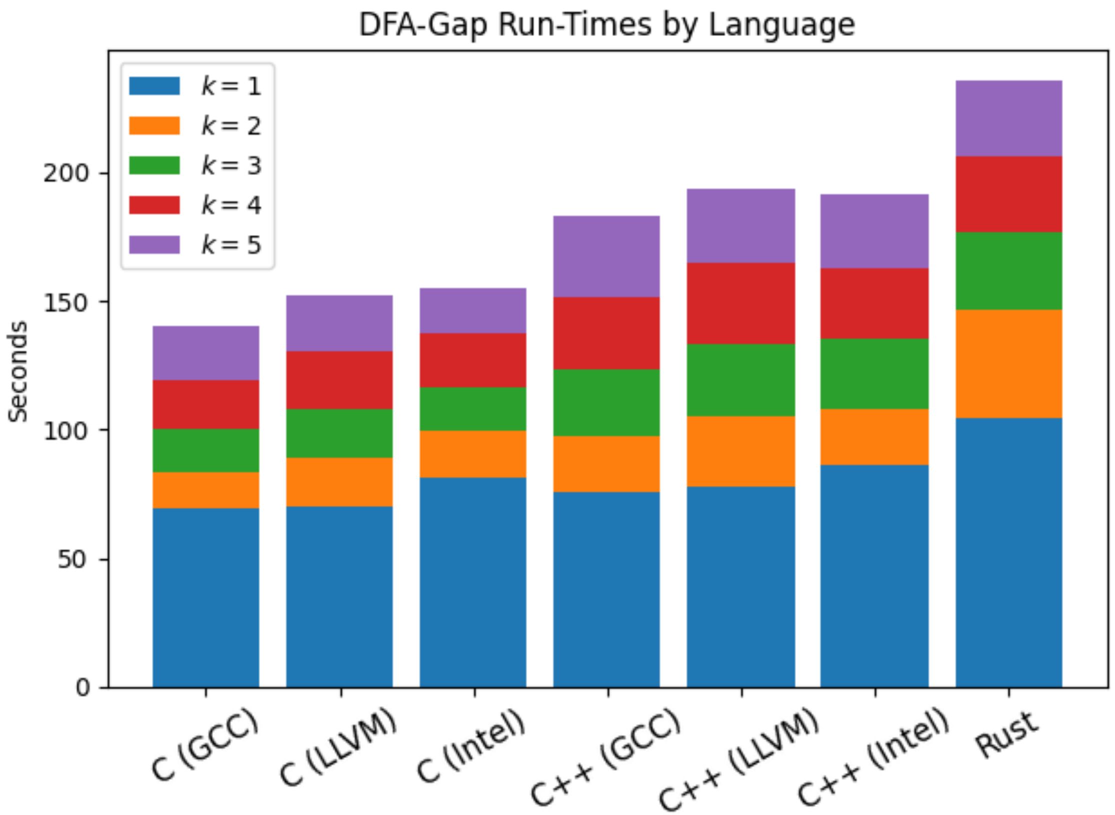


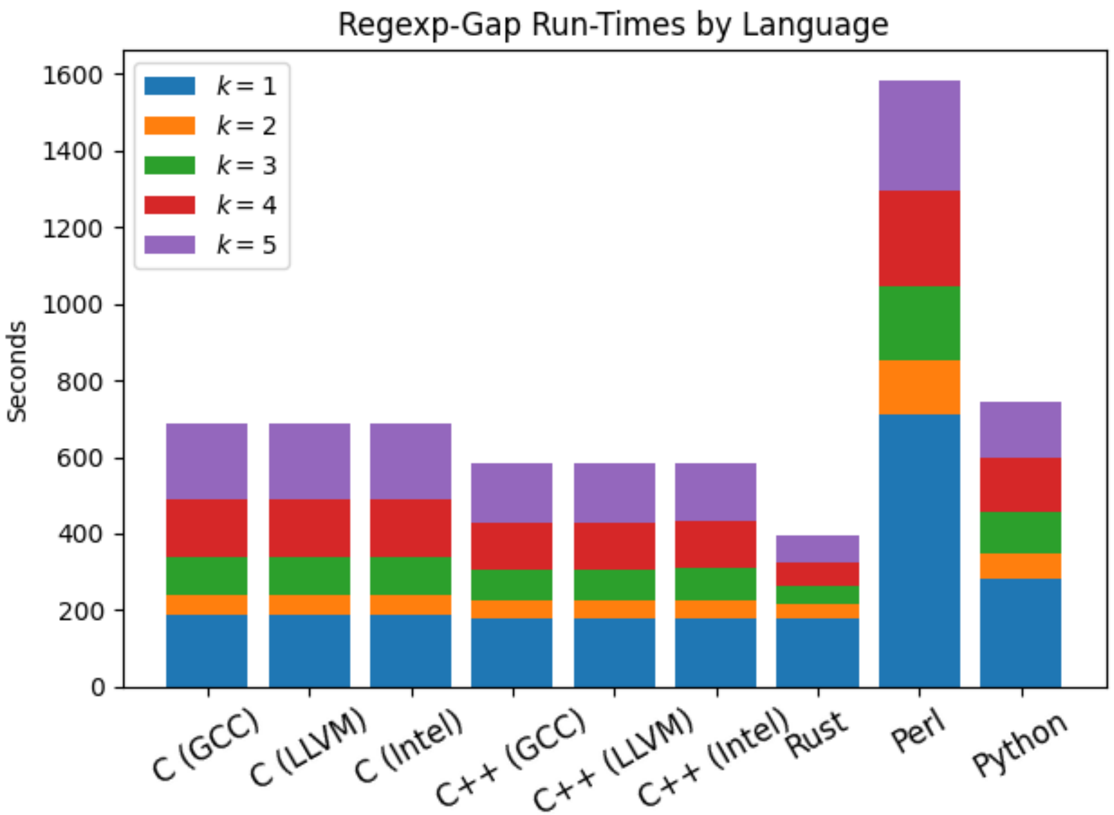
Exact Match Algorithm Run Times



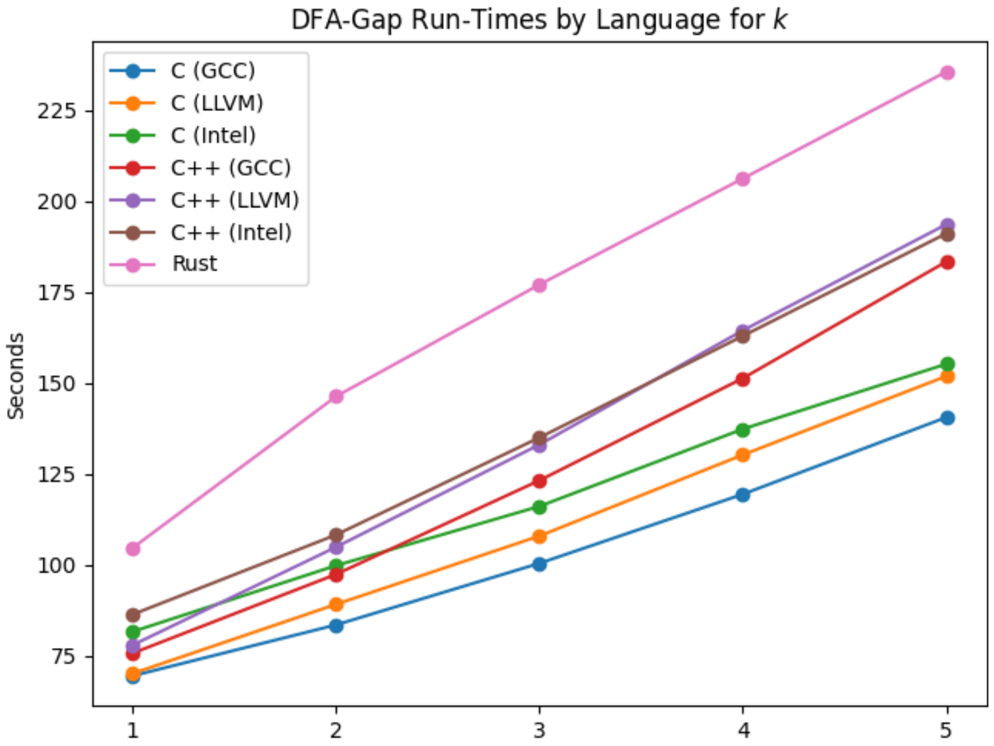


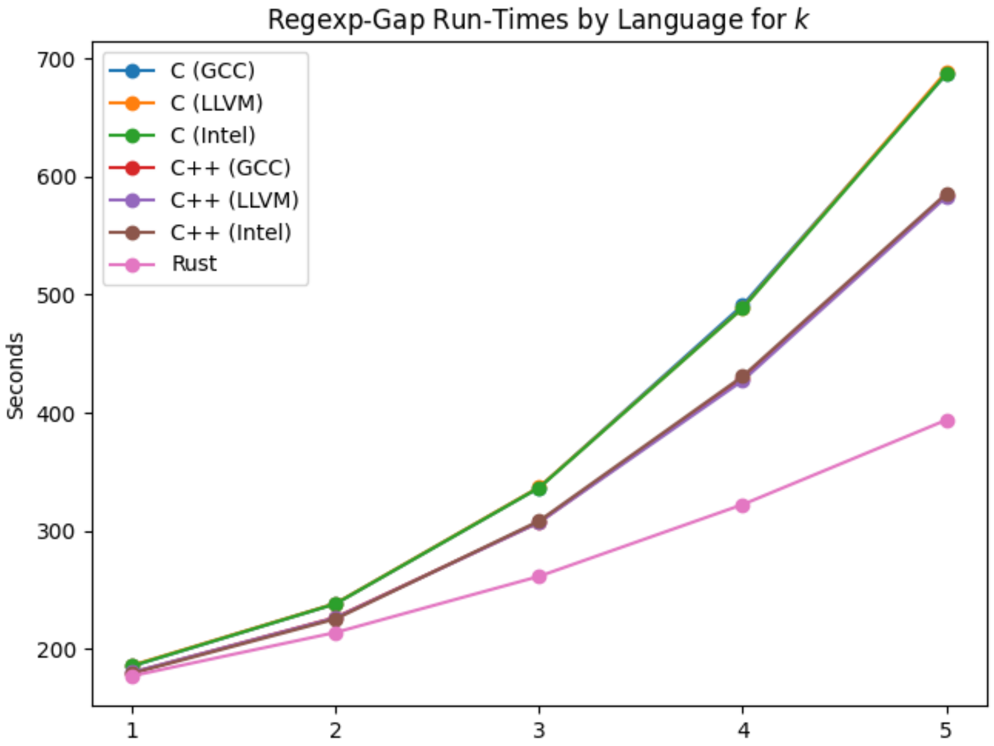
Novel Algorithm Run Times - DFA & Regexp





Novel Algorithm Run Times - DFA & Regexp





Final Rankings, with detail

(a) Score by scale, with complexity

Language	Score
C++ (GCC)	1.0000
C++ (LLVM)	1.0001
C++ (Intel)	1.0007
Rust	1.2058
C (GCC)	1.2446
C (LLVM)	1.2447
C (Intel)	1.2454
Python	3.2200
Perl	3.4881

(b) Score by scale, no complexity

Language	Score
C++ (GCC)	1.0000
C++ (LLVM)	1.0001
C++ (Intel)	1.0006
Rust	1.0987
C (GCC)	1.1521
C (LLVM)	1.1523
C (Intel)	1.1529
Python	3.0424
Perl	3.2800

(c) Score by ranks, with complexity

Language	Score
C++ (GCC)	1.0000
C++ (LLVM)	1.0000
Rust	1.4951
C++ (Intel)	1.8150
C (GCC)	2.4132
C (LLVM)	2.4375
Python	2.7547
C (Intel)	2.9406
Perl	3.3166

(d) Score by ranks, no complexity

Language	Score
C++ (GCC)	1.0000
C++ (LLVM)	1.0000
Rust	1.3143
C++ (Intel)	1.6652
C (GCC)	2.1213
C (LLVM)	2.1426
Python	2.4215
C (Intel)	2.5849
Perl	2.7469

How Final Rankings Grew

- Differences in score based ranking and placement ranking