Programming language comparisons are always interesting and rife with opinion, and this one from author Naser Tamimi is no exception. We can learn a great deal from considering his article. He wrote an excellent article about considering C++ for data science problems (see the article at [Medium - How fqast is C++](https://towardsdatascience.com/how-fast-is-c-compared-to-python-978f18f474c7). )

He provided an algorithm, and “identically” coded implementations, which he ran in both Python and C++ and timed the result. It was no surprise that C++ was faster running the algorithm. I was moderately surprised with his finding that it was *twenty-five times* faster than Python. I wanted to investigate this finding and include my own observations. I also wanted to add the Java programming language for consideration by those seeking performance in Data Science computing.

The Python and Java source code supporting this article is available in my GitHub repository at [DemoDev K-mer Algorithm](https://github.com/DonaldET/DemoDev/tree/master/dev-topics-algorithms/dev-topics-kmer).

# Recommendations for Language Performance Evaluation

Changing an implementation language can be a significant risk and should be undertaken with more rigor than emphasized in Naser’s article. My modifications to his recommendations are as follows:

1. Choose the test algorithm carefully, not just any simple algorithm will do.
2. Make sure the algorithm is “clean” and understandable.
3. Code the algorithm for each language as an experienced programmer in that language would code it (e.g., “Pythonic” and idiomatically for Python.)
4. Make use of well-known coding paradigms to present a well thought out algorithm to represent in the target language.
5. Compare how well simple optimizations in each target language

Spoiler alert: we can easily achieve a *184X speed improvement over Python*. Read on to see the process.

# Language Performance Testing Algorithm

Naser chose to compute K-MERs, a concept from computational genetics that is a combinatorial generation problem. He does an excellent job of presenting the problem domain in plain language, but I will simplify it even further (see a detailed explanation at [Wiki K-mer explaination](https://en.wikipedia.org/wiki/K-mer).)

Compute all possible strings of length k where each character of the string is drawn from the sequence ‘A’, ‘C’, ‘G’, ‘T’. For example, for k = 2, we have sixteen possible K-mer strings:

1. AA 9. GA

2. AC 10. GC

3. AG 11. GG

4. AT 12. GT

5. CA 13. TA

6. CC 14. TC

7. CG 15. TG

8. CT 16. TT

The algorithm Naser chose to test is basically a simple “odometer”. You can see the right-most position of the K-mer above cycle through [‘A’, ‘C’, ‘G’, ‘T’] for each “digit” of the left-most character in the generated string. We will use a K-mer of length 13 in our performance timing.

# Choose Wisely

Naser has made the choice for us and we will stick with the K-mer domain problem, tested using the odometer algorithm, to compare Python and Java.

# Make sure the algorithm is “clean” and understandable

Naser’s sample code was unfortunately flawed with unnecessary code in the original version, which we cleaned up. Here is the comparison of the inner portion of the code.

|  |  |  |
| --- | --- | --- |
| **Original Article Algorithm (kmer\_raw.py)** |  | **Cleaned up Algorithm (kmer\_raw\_fix.py)** |
| Variable s is the current K-mer iteration, variable s\_last is the ending value for the iterations (TTTTTTTTTTTTT)  Variable endr is the last nucleotide (T), marking the last “digit” in the odometer | | |
| pos = 0  counter = 1  while s != s\_last:  counter += 1  *change\_next = True*  for i in range(len\_str):  if *change\_next*:  if s[i] == opt[-1]:  s = s[:i] + convert(s[i]) + s[i + 1:]  *change\_next = True*  else:  s = s[:i] + convert(s[i]) + s[i + 1:]  break |  | pos = 0  counter = 1  while s != s\_last:  counter += 1  for i in range(len\_str):  done = s[i] != ender  s = s[:i] + convert(s[i]) + s[i + 1:]  if done:  break |
| 68.828 seconds |  | 67.140 seconds |

The cleaned up version is also 2.5% faster than the less clear version as a bonus.

# Code the algorithm for each language idiomatically

The original algorithm Nasier proposed for comparison used dynamic string slicing support in Python syntax. There is no such string slicing support in Java. One must use methods of a String instance to accomplish concatenation. To again achieve similar code patterns in both Python and Java, we refactor the list construction component to use similar coding patterns (see method build\_by\_append.)

|  |  |  |
| --- | --- | --- |
| **Python Article Algorithm (kmer\_article.py)** |  | **Java version (KmerLists.java)** |
| Variable s is the current K-mer iteration, variable s\_last is the ending value for the iterations (TTTTTTTTTTTTT)  Variable first\_base is the first nucleotide (A), marking the first “digit” in the odometer  Variables nc and newBase are the new nucleotide from the generation mechanism | | |
|  | | |
| def \_build\_by\_append(s, pos, new\_nucleotide):  last = len(s) - 1  if pos > 0:  if pos < last:  s\_new = s[0:pos]+new\_nucleotide+s[pos+1:]  else:  s\_new = s[0:last]+new\_nucleotide  else:  s\_new = new\_nucleotide + s[1:]  return s\_new |  | private static String build\_by\_append(String s,  int pos, String newBase, StringBuilder sb) {  sb.delete(0, sb.length());  if (pos > 0) {  sb.append(s.substring(0, pos));  sb.append(newBase);  if (pos < s.length() - 1) {  sb.append(s.substring(pos + 1));  }  } else {  sb.append(newBase);  sb.append(s.substring(1));  }  return sb.toString();  } |
|  | | |
| count = 1  while s != s\_last:  count += 1  pos = len\_str - 1  while pos >= 0:  nc = nucleotides\_rotation[s[pos]]  s = \_build\_by\_append(s, pos, nc)  if nc != first\_base:  break  pos -= 1 |  | int count = 1;  while (!s.equals(slast)) {  count += 1;  int pos = kmerLength - 1;  while (pos >= 0) {  String newBase = nucleotidesRotation.get(s.substring(pos, pos + 1));  s = build\_by\_append(s, pos, newBase, sb);  if (!newBase.equals(firstBase)) {  break;  }  pos -= 1;  } |
| 39.703 seconds |  | 5.359 seconds |

We are test on approximately the same basis as the Python/C++ test in Naser’s article. We note that Java is only seven times faster than Python in this case. This was suspicious to me because two decades of Java development have lead to close performance parity. Closer examination of the C++ code in Naser’s article reveals that C++ example is not dynamically creating strings with every operation like the Python and Java examples here (build\_by\_append.) This speeds up C++ significantly.

## Remove dynamic string creation

We replace dynamic string creation with static list creation for both implementations, and considerable improvements are seen.

|  |  |  |
| --- | --- | --- |
| **Python Odometer Algorithm (kmer\_article.py)** |  | **Java version (KmerLists.java)** |
| Variable s is the current K-mer iteration, variable s\_last is the ending value for the iterations (TTTTTTTTTTTTT)  Variable first\_base is the first nucleotide (A), marking the first “digit” in the odometer  Variables nc and newBase are the new nucleotide from the generation mechanism | | |
|  | | |
| count = 1  while s != s\_last:  count += 1  pos = len\_str - 1  while pos >= 0:  nc = nucleotides\_rotation[s[pos]]  s = \_build\_by\_append(s, pos, nc)  if nc != first\_base:  break  pos -= 1 |  | int count = 1;  while (!s.equals(slast)) {  count += 1;  int pos = kmerLength - 1;  while (pos >= 0) {  String newBase = nucleotidesRotation.get(s.substring(pos, pos + 1));  s = build\_by\_append(s, pos, newBase, sb);  if (!newBase.equals(firstBase)) {  break;  }  pos -= 1;  } |
| 39.703 seconds |  | 5.359 seconds |

Algorithm Cleanup Validity Test Run

|  |  |
| --- | --- |
| **Raw Algorithm from Medium Article** | **Mild rework** |
| Start article algorithm  1 AA  2 CA  3 GA  4 TA  5 AC  6 CC  7 GC  8 TC  9 AG  10 CG  11 GG  12 TG  13 AT  14 CT  15 GT  16 TT  Number of generated k-mers: 16, elapsed time: 0.0 seconds.  Finish! | Start KMER Computation by Constructing Lists  Nucleotides: ACGT; Sequence Length: 2; K-MERs expected: 16  First: AA  Last : TT  1. AA  2. AC  3. AG  4. AT  5. CA  6. CC  7. CG  8. CT  9. GA  10. GC  11. GG  12. GT  13. TA  14. TC  15. TG  16. TT  Number of generated k-mers: 16  Elapsed time: 0.0 secs  Finished! |