Table of Contents

[Recommendations for Language Performance Evaluation 1](#_Toc63527517)

[A. Language Performance Testing Algorithm 2](#_Toc63527518)

[Choose Wisely 2](#_Toc63527519)

[B. Make sure the algorithm is “clean” and understandable 3](#_Toc63527520)

[C. Code the algorithm for each language idiomatically 3](#_Toc63527521)

[Remove dynamic string creation 5](#_Toc63527522)

[Make use of well-known coding paradigms to represent the algorithm in the target language 6](#_Toc63527523)

[D. Compare how well simple optimizations perform in each target language 6](#_Toc63527524)

[Performance Analysis 7](#_Toc63527525)

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

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

The complete 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). This article only shows the main algorithm sections of the code.

# 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. Therefore, my modifications to his recommendations are as follows:

1. Choose the test algorithm carefully, not just any simple algorithm will properly evaluate your language comparison.
2. Make sure the algorithm is “clean” and understandable to begin your language comparison.
3. Code the implementation of the algorithm for each language as an experienced programmer in that language would code it (e.g., “Pythonic” and idiomatically for Python.)
   * Make use of well-known coding paradigms to represent the algorithm in the target language.
4. Compare how well simple optimizations are used in each target language.

Spoiler alert: we can easily achieve a *187X speed improvement over Python*. Read on to see the process and qualifying conditions.

# 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’, each representing a base. 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 Python program was unfortunately flawed with unnecessary code, which we cleaned up as shown below. 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 original version as a bonus.

# Code the algorithm for each language idiomatically

The original algorithm Naser proposed used dynamic string slicing support in Python syntax, but there is no 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 in Python and Java 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 base nucleotide from the generation mechanism | | |
|  | | |
| def ***\_build\_by\_append***(s, pos, new\_nucleotide):  if pos > 0:  last = len(s) - 1  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 testing on approximately the same basis as the Python/C++ test in Naser’s article. We note that Java is only 7x faster than Python in our case. This was suspicious to me because two decades of Java JVM development have led to close parity with C++, so I would have expected Java to be 25x faster as well. Closer examination of the C++ code in Naser’s article reveals that the C++ example is not dynamically creating strings and appending them like the Python and Java examples. Using a static array speeds up C++ significantly. Additionally, the refactored Python approach above, where we avoid concatenating empty strings at the beginning and end of each pass, is 2.3x faster than Naser’s published version.

## Remove dynamic string creation

We replace dynamic string creation with static arrays for both Python and C++ implementations, and considerable improvements are seen. In addition, the original example uses a function call (convert) to cycle through each base in the nucleotide sequence. We replace this this function with a dictionary/map instance that is often used to represent a state machine transitions (nucleotides\_rotation/nucleotidesRotation.)

|  |  |  |
| --- | --- | --- |
| **Python Odometer Algorithm (kmer\_article\_odometer.py)** |  | **Java version (KmerOdometer.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 nucleotides\_rotation and nucleotidesRotation are dictionary/map structure for the new nucleotide from the generation mechanism | | |
| nucleotides\_rotation = {'A': 'C', 'C': 'G', 'G': 'T', 'T': 'A'} |  | public static final Map<Character, Character> nucleotidesRotation = new HashMap<>();  static {  nucleotidesRotation.put('A', 'C');  nucleotidesRotation.put('C', 'G');  nucleotidesRotation.put('G', 'T');  nucleotidesRotation.put('T', 'A');  } |
|  |  |  |
| count = 1 while s != s\_last:  count += 1  pos = len\_str - 1  while pos >= 0:  s[pos] = nucleotides\_rotation[s[pos]]  if s[pos] != first\_base:  break  pos -= 1 |  | int count = 1;  while (!Arrays.equals(s, sLast)) {  count += 1;  int pos = kmerLength - 1;  while (pos >= 0) {  s[pos] = (byte) (nucleotidesRotation.get((char) (s[pos] & 0xFF)) & 0x00FF);  if (s[pos] != firstBase) {  break;  }  pos -= 1;  }  } |
| 15.422 seconds |  | 0.453 seconds |

With both Python and Java using the same static array approach, we see that *Java is thirty four times faster than Python*. It is interesting to note that this minor cleanup sped up the original Python code by a factor of 2.6x.

## Make use of well-known coding paradigms to represent the algorithm in the target language

Static allocation is a common approach taken by experienced coders in either language. Based on both experience and testing, a dictionary/map lookup is faster than a function full of “if-then” tests and is a standard representation for state transitions.

# Compare how well simple optimizations perform in each target language

We can now apply some simple C-like optimizations to both Python and Java code implementations at this stage. We can change the representation of a base to be {0, 1, 2, 3} instead of characters ‘A’, ‘C’, ‘G’, and ‘T’. We replace the dictionary/map representation for generating the next base with an array lookup instead of a hash lookup.

|  |  |  |
| --- | --- | --- |
| **Python Optimized Algorithm (kmer\_article\_c.py)** |  | **Java version (KmerC.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 nucleotides\_rotation and nucleotidesRotation are arrays providing new nucleotide from the generation mechanism | | |
| nucleotides = "ACGT"  nc = len(nucleotides)  nucleotides\_rotation = (1, 2, 3, 0) |  | public static final byte[] nucleotideLabels = new byte[] { 'A', 'C', 'G', 'T' };  public static final byte[] nucleotides = new byte[] { 0, 1, 2, 3 };  public static final byte[] nucleotidesRotation = new byte[] { 1, 2, 3, 0 }; |
|  |  |  |
| count = 1  while s != s\_last:  count += 1  pos = len\_str - 1  while pos >= 0:  s[pos] = nucleotides\_rotation[s[pos]]  if s[pos] != first\_base:  break  pos -= 1 |  | int count = 1;  while (!Arrays.equals(s, sLast)) {  count += 1;  int pos = kmerLength - 1;  while (pos >= 0) {  s[pos] = nucleotidesRotation[s[pos] & 0xFF];  if (s[pos] != firstBase) {  break;  }  pos -= 1;  }  } |
| 14.391 seconds |  | 0.212 seconds |

The final optimizations show the Java version is 68x faster than the equivalent Python version. We have come a long way. This optimized version is more than 187x faster than the original Python version from Naser’s article

# Performance Analysis

Here are the collected performance statistics for our three stages of optimization and testing of Python verses Java:



Our first observation is that Java responds well to optimization, and becomes increasing faster than Python as we tune the code. Let’s look at the improvement each stage of optimization achieves compared to the prior un-optimized version:



Our second observation is both Python and Java benefited greatly from removing dynamic memory management and using static storage. Additionally, Java tended to respond better to performance optimizations.