Appendix

In its problem domain, the Catenator provides increases in speed and compactness that are so overwhelmingly large as to be almost beyond definition. Almost, but not quite. For those in search of more precision, this appendix discusses the computational complexity of common operations within the Catenator.

Let *m* be the number of different lists of strings contained in the Catenator. Let k_0 to k_m be the number of strings contained in the respective lists. Then the number of different strings contained in the Catenator is $k_0 \times k_1 \times \ldots \times k_m = z$. Let

$$k = \frac{1}{m}\sum_{i=0}^{m} k_i$$

then k^m approximates *z*, depending on the degree of volatility of the *k*'s and in particular whether any of them is much smaller than the others (as the accuracy of the approximation depends only on the distribution of the *k*'s and not the problem size, it may be treated as a constant parameter when determining complexity). In other words, provided $m > 1$, and $k_p >> 1$ for $0 \le p \le m$, k^m is a reasonable approximation for the number of different strings in the Catenator. A Catenator fulfilling these assumptions will be termed a *non-pathological Catenator* and one not fulfilling these assumptions will be termed a *pathological Catenator*.

The problem size (*n*) for the operations under review is the size of the input lists of strings, i.e., $k_0 + k_1 + \ldots k_m$, which is precisely $m \times k$. Note that, for *m* approximately equal to *k*,

$$n^{\frac{\sqrt{n}}{2}} \approx k^m .$$

Creation

For a plain vector, creation by combining all possible lists of sub-strings requires k^m operations, i.e., it is approximately $O(n^{\frac{\sqrt{n}}{2}})$.

For a Catenator, creation requires copying $k \times m$ individual strings, together with a small overhead proportional to *m*. For a non-pathological Catenator, this is $O(km) = O(n)$, i.e., linear time.

operator[]

For a plain vector, array-access is a constant time operation, i.e. it is $O(1)$.

For a Catenator, array-access requires m combinations of substrings, together with a small constant overhead proportional to m. For a non-pathological Catenator, this will be effectively $O(1)$, but for larger values of m, this will grow to approximately $O(\sqrt{n})$.

Matching Strings

For a plain vector, finding a match is an $O(n^{\frac{\sqrt{n}}{2}})$ operation – as the vector is unsorted; matching involves a linear search through k^m strings. If the vector were sorted, this would improve to, say, $O(\sqrt{n} x \log n)$, for a binary search. Obviously, this involves the additional cost of sorting the vector in the first place.

For a non-pathological Catenator, finding a match involves $k \times m$ comparisons, i.e. it is $O(n)$. Again, this could be improved by sorting the strings in the Catenator and any such sort will be much cheaper than sorting the equivalent vector (it involves sorting m lists of k members, as opposed to one list of k^m members).

Conclusion

When used as intended, the Catenator will improve most operations from exponential to linear time. Memory usage improves to a similar extent.