Introduction

This report provides a description of Stefano Costa's code focusing mainly on the computational complexity of all functions.

All graphs reported were done in Python with the library `matplotlib.pyplot`, while the other images were done by the author with an online software. The codes used to generate the graphs have not been submitted, but can be upon request.

1 `number2base_rep(n, b)`

Explanation

Let $n \in \mathbb{N}$ and $b \in \mathbb{N}_{\geq 2}$. To obtain $(n)_b$ the following steps are taken:

1. Divide n by b. Obtain q_0 quotient, r_0 remainder.
2. Divide q_0 by b. Obtain q_1 quotient, r_1 remainder.
3. Continue dividing q_i by b until $q_i = 0$.
4. Calling the obtained reminaders: $r_0, ..., r_n$, the result is computed as $(n)_b = r_n r_{n-1} ... r_0$

An example is shown in table 1.

Division	Quotient	Remainder
24/4	6	0
6/4	1	2
1/4	0	1

Table 1: Example of base change: $(24)_{10} = (120)_4$

Code

The cases $b = 10$ and $n = 0$ are treated in an `if` to avoid unnecessary computations.

Complexity

The idea of the algorithm is to divide by b until the quotient obtained is zero, hence $\log_b(n)$ divisions are performed. Therefore the overall asymptotic complexity is $O(\log_b(n))$ or equivalently $O(\log(n))$.

2 `admissible(n, b)`

Code

The function `admissible(n, b)` calls `number2base_rep(n, b)` and relies on another function that checks whether a string is admissible, called `checkAdmissibleString(s)`.
The function `checkAdmissibleString(s)` accepts a string and returns `True` if it is acceptable, otherwise returns `False`. To check the equality of adjacent substrings, it is checked character by character.

Explanation of `checkAdmissibleString(s)`

Starting at `i=0`, equality of all adjacent characters is checked, as shown in figure 1.

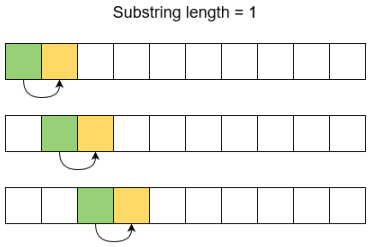

Figure 1: How substring of length one are checked

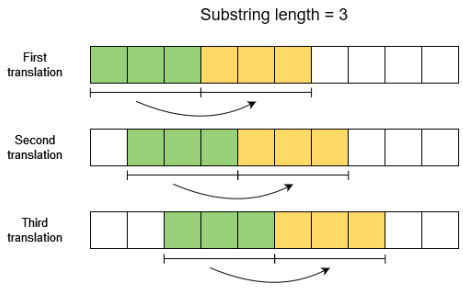

Figure 2: How substring of length three are checked

If two adjacent characters are equal, the algorithm returns `False`. Otherwise it continues checking 2-element substrings in a similar manner. The only difference is that firstly the algorithm checks the equality starting at `i=0`, after passing the whole string it reperforms the check starting at `i=1` as shown in figure 3.

If two adjacent 2-element substrings are equal, the algorithm return `False`. Otherwise it continues checking 3-element substrings in a similar manner. It is clear that searching for adjacent k-element substrings requires reperforming the check at k different starting indexes. In figure 2 is shown the 3-element substrings case.

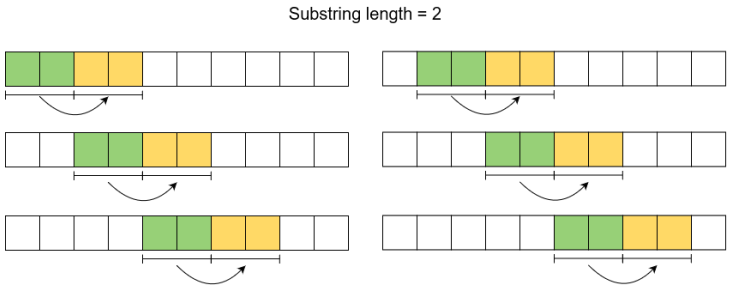

Figure 3: How substring of length two are checked

Only substrings of length $l \leq \lfloor \frac{n}{2} \rfloor$ are checked. If no adjacent equal substrings are found, the algorithm returns `True`.

Complexity

Complexity of `checkAdmissibleString(s)`

The check of 1-element adjacent substrings requires $n-1$ operations, where n is the length of the string.
The check of 2-element adjacent substrings is the same as the check of adjacent elements for a list consisting of pairs of string characters. This concept is shown in figure 4.

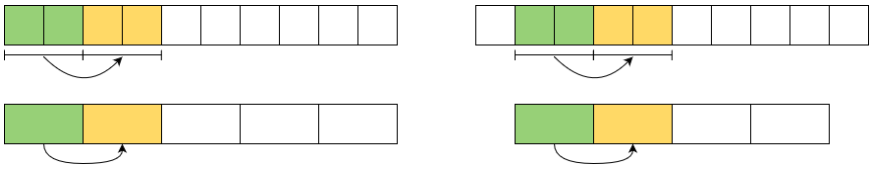

Figure 4: Checking 2-element adjacent substring in the string is the same as checking 1-element adjacent sublist in a list of pairs of characters.

A list composed of pairs of string characters has $\frac{n}{2}$ elements if n is even, $\frac{n-1}{2}$ elements if n is odd, hence it has $\lfloor \frac{n}{2} \rfloor$ elements. Checking all adjacent elements in this list requires, as in the case of 1-element substrings, as many checks as the length of the list minus one, i.e. $\lfloor \frac{n}{2} \rfloor - 1$.

This argument so far has not taken into account checking 2-element adjacent substrings starting at index `i=1`. For this case the reasoning is similar, the only difference being that starting from `i=1` the first element of the string is not considered. Consequently, the list will have $\lfloor \frac{n-1}{2} \rfloor$ elements and the operation will require $\lfloor \frac{n-1}{2} \rfloor - 1$ checks. Overall the control of 2-element adjacent substrings requires $\lfloor \frac{n}{2} \rfloor - 1 + \lfloor \frac{n-1}{2} \rfloor - 1$ checks.

It is now clear how to extend this argument to the check of 3-element adjacent substrings, where there are three addends: $\lfloor \frac{n}{3} \rfloor - 1$ for the check starting at `i=0`, $\lfloor \frac{n-1}{3} \rfloor - 1$ for the check starting at `i=1`, $\lfloor \frac{n-2}{3} \rfloor - 1$ for the check starting at `i=2`.

Since for a string of n characters it is necessary to check adjacent substrings of at most $\lfloor \frac{n}{2} \rfloor$ elements, the total number of checks is: $\sum_{k=1}^{\lfloor \frac{n}{2} \rfloor} \sum_{i=0}^{k-1} \left(\lfloor \frac{n-i}{k} \rfloor - 1 \right)$. That sum represents the worst case, in which all substrings are checked. Taking into account the check of individual characters of substrings, the worst case of checking all characters of all adjacent substrings requires $\sum_{k=1}^{\lfloor \frac{n}{2} \rfloor} \sum_{i=0}^{k-1} k \cdot \left(\lfloor \frac{n-i}{k} \rfloor - 1 \right)$ operations. With the help of the graph shown in figure 5, it turns out that the total number of operations behaves as n^c with $c \in [2.3, 2.5]$

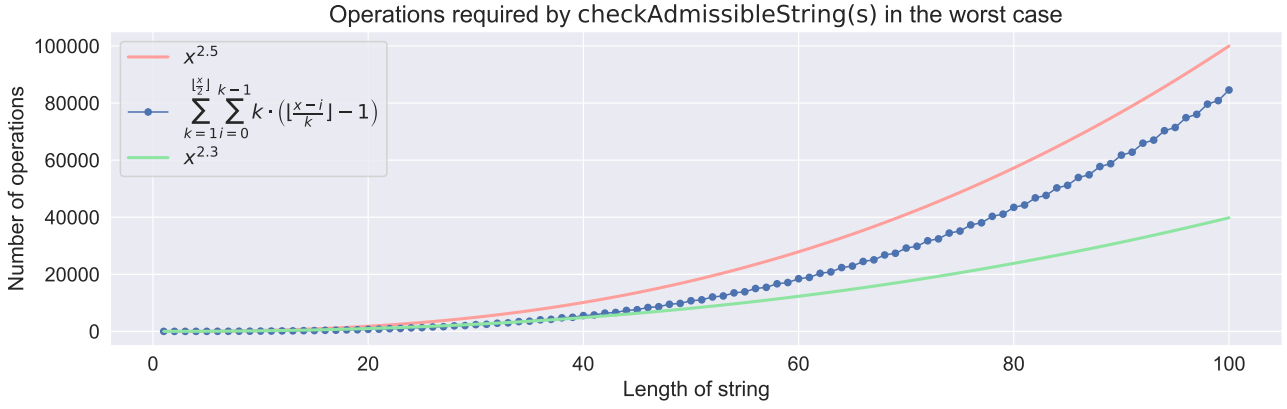

Figure 5: Behavior of the total number of operations compared with $n^{2.3}$ and $n^{2.5}$

Average operations of `checkAdmissibleString(s)`

Note that the worst case is unlikely, since it requires all substrings to be different only in the last character. To investigate the average number of operations, benchmarks have been made. Tests were done with strings up to 96 characters in length (using all python printable characters: `string.printable`) with 50.000 random strings for each length. The number of operations and time required to execute the function `checkAdmissibleString(s)` were measured.

The mean and standard deviation of the empirical number of operations is shown in figure 6. The average number of steps grows approximately as $n^{1.5}$. Evidently, the standard deviation increases as the length of the string increases. This

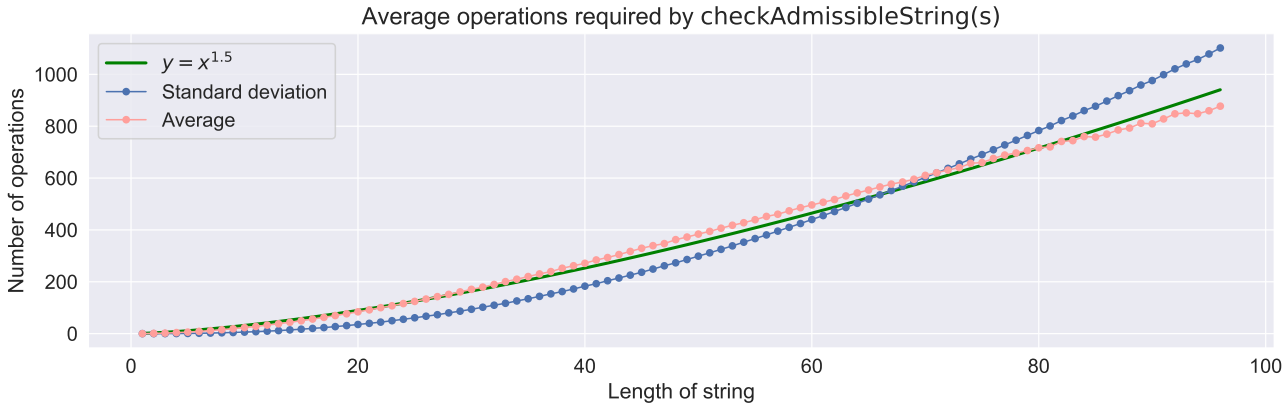

Figure 6: Behavior of the average and standard deviation of the number of operations compared with $n^{1.5}$.

is reasonable thinking about the fact that 50.000 strings represent an insignificant number of all possible strings of length $k > 10$.

The average time required by `checkAdmissibleString(s)` roughly follows the trend of the average number of operations, as is shown in figures 7 and 8. Evidently there are fluctuations that could be avoided by increasing the number of random strings per length. This problem is also noticeable in the standard deviation, which fluctuates greatly even for small n.

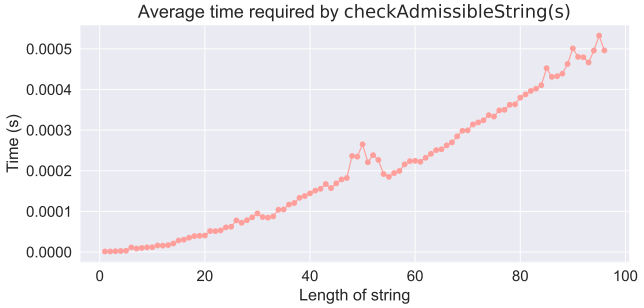

Figure 7: Average time required by `checkAdmissibleString(s)` for different values of n.

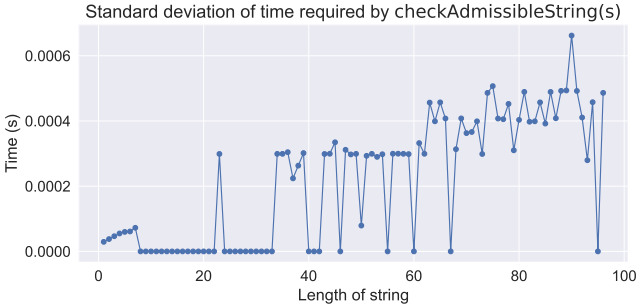

Figure 8: Standard deviation of time required by `checkAdmissibleString(s)` for different values of n.

Complexity of `admissible(n, b)`

To calculate the number of operations required by the function `admissible(n, b)`, it is necessary to take into account the operations required by `number2base_rep(n, b)` and `checkAdmissibleString(s)`.

As noted above, `number2base_rep(n, b)` requires $\log_b(n)$ operations. Since the number n converted to base b has $\lceil \log_b(n) \rceil$ digits (which in terms of asymptotic complexity can be treated as $\log_b(n)$) and is passed to `checkAdmissibleString(s)` as the string `s`, the number of operations performed by `admissible(n, b)` in the worst case is $\log_b(n)^{2.5}$ while for the average case is $\log_b(n)^{1.5}$.

Overall, $\log_b(n) + \log_b(n)^{2.5}$ operations are taken in the worst case, asymptotically $O(\log_b(n)^{2.5})$. For the average case one has $O(\log_b(n)^{1.5})$.

3 `count_admissible(b, start, end)`

Explanation

The function calls `admissible(n,b)` once for each element in `[start, end)` and updates a counter.

Complexity

Because `admissible(n,b)` is called `end` $-$ `start` times, by averaging over the length of $(n)_b$ the asymptotic complexity is $O((\text{end} - \text{start}) \cdot \log_b(\frac{\text{end}-\text{start}}{2})^{2.5})$ in the worst case while in the average case is $O((\text{end} - \text{start}) \cdot \log_b(\frac{\text{end}-\text{start}}{2})^{1.5})$.

Comparing the empirical time required by `count_admissible(5,10**k,10**(k+1))`, as shown in figure 9, with the hypothesized average number of operations, as shown in figure 10, it can be seen that the behaviors are similar

Sequence of results of `count_admissible(5,10**k,10**(k+1))`

The result of the call `count_admissible(5,10**k,10**(k+1))` generates the sequence: `[8, 60, 370, 2715, 17238, 111465, 776004, 4829962, ...]`. A search for this sequence on the OEIS site did not yield any results. All data can be found in table 2.

4 `count_admissible_width(b,width)`

Explanation

The function `count_admissible(b, start, end)` is used by setting appropriate `start` and `end` values. In particular, the following are set: `start` $= b^{(\texttt{width}-1)}$, `end` $= b^{\texttt{width}}$.

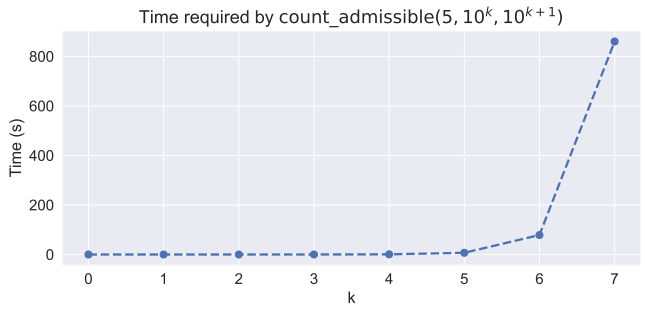

Figure 9: Empirical time required by `count_admissible(5,10**k,10**(k+1))` for different values of k.

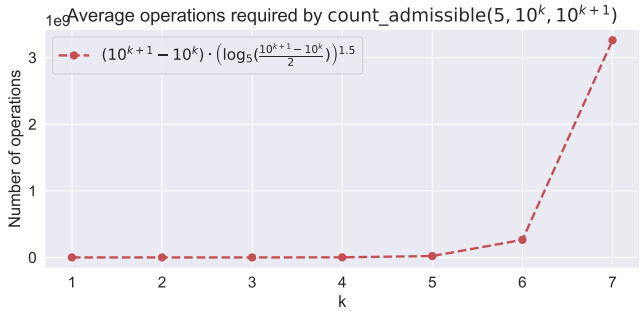

Figure 10: Hypothesized average number of operations required by `count_admissible(5,10**k,10**(k+1))` for different values of k

Complexity

Using the complexity of `count_admissible(b, start, end)`, the average case of the call `count_admissible_width(3,k)` is asymptotically $O([(3^k-1)-3^{k-1}]\cdot\left(\log_3\left(\frac{[(3^k-1)-3^{k-1}]}{2}\right)\right)^{1.5})$ while the worst case is $O([(3^k-1)-3^{k-1}]\cdot\left(\log_3\left(\frac{[(3^k-1)-3^{k-1}]}{2}\right)\right)^{2.5})$. Comparing the empirical time shown in figure 11 with the average case hypothesized in figure 12, it may be seen that the behaviors are similar.

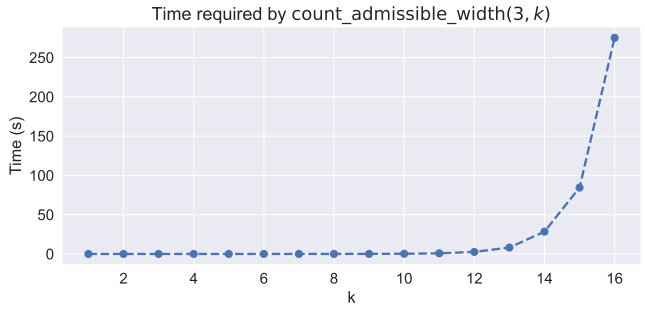

Figure 11: Empirical time required by `count_admissible_width(3,k)` for different values of k.

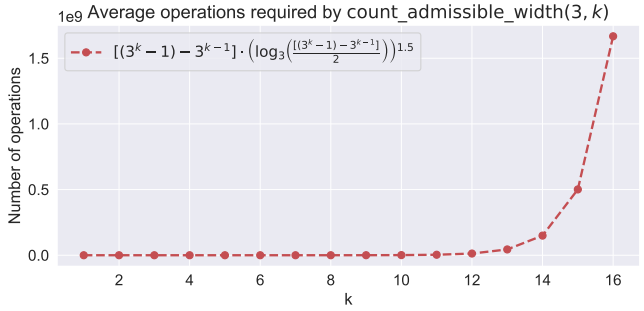

Figure 12: Hypothesized average number of operations required by `count_admissible_width(3,k)` for different values of k.

Sequence of results of `count_admissible_width(3,k)`

The sequence generated by `count_admissible_width(3,k)` is: `[3, 4, 8, 12, 20, 28, 40, 52, 72, 96, 136, 176, 228, 304, 412, 532, ...]` and this coincides with the OEIS succession: A088953 representing (quoting the description from the site): "*Number of numbers that are ternary squarefree words of length n*". All data can be found in table 3.

5 `largest_multi_admissible(L, start, end)`

Explanation

The algorithm calls `admissible(n,b)` for each base in the list `L` and for each element in `[start, end)` starting with the largest. Since it starts from the largest in the interval, when a number is admissible for each base, the function returns it.

Complexity

The complexity is the same as `count_admissible(b,start,end)` multiplied by the length of `L` (to avoid writing it as a sum, the logarithm in base 10 is used). Hence the average case is $O(\texttt{len(L)}\cdot(\text{end}-\text{start})\cdot\log(\frac{\text{end}-\text{start}}{2})^{1.5})$, while the worst case is $O(\texttt{len(L)}\cdot(\text{end}-\text{start})\cdot\log(\frac{\text{end}-\text{start}}{2})^{2.5})$.

The average number of operations required by `largest_multi_admissible([3, 5, 7, 10], 1, 10**k)` can be seen in figure 14, while the empirical time can be found in figure 13.

Sequence of results of `largest_multi_admissible([3, 5, 7, 10], 1, 10**k)`

The sequence generated by `largest_multi_admissible([3, 5, 7, 10], 1, 10**k)` is: `[7, 96, 923, 8165, 70921, 657984, 8428271, 92045829]`. A search for this sequence on the OEIS site did not yield any results. All data can be found in table 4.

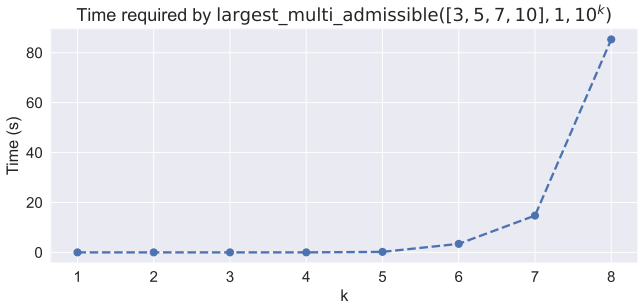

Figure 13: Empirical time required by `largest_multi_admissible([3, 5, 7, 10], 1, 10**k)` for different values of k.

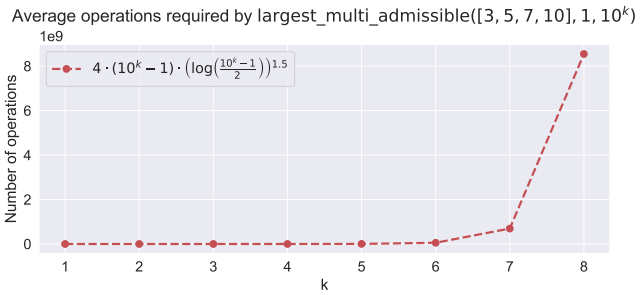

Figure 14: Hypothesized average number of operations required by `largest_multi_admissible([3, 5, 7, 10], 1, 10**k)` for different values of k.

6 Appendix

This section reports all the results of function calls: `count_admissible(5,10**k,10**(k+1))`, `count_admissible_width(3, k)`, `largest_multi_admissible([3, 5, 7, 10], 1, 10**k)`.

k	`count_admissible(...)`	time (s)
0	8	0.00000
1	60	0.00000
2	370	0.00497
3	2715	0.08477
4	17238	0.63103
5	111465	6.91480
6	776004	78.59082
7	4829962	860.05345

Table 2: Results and CPU time of `count_admissible(5,10**k,10**(k+1))` for $k \in [0, 7]$.

k	`count_admissible_width(3,k)`	time (s)
1	3	0.00075
2	4	0.00000
3	8	0.00023
4	12	0.00000
5	20	0.00267
6	28	0.00450
7	40	0.01293
8	52	0.03344
9	72	0.10543
10	96	0.26590
11	136	0.81404
12	176	2.61641
13	228	8.16544
14	304	28.43366
15	412	84.44320
16	532	275.12542

Table 3: Results and CPU time of `count_admissible_width(3, k)` for $k \in [1, 16]$.

k	`largest_multi_admissible(...)`	time (s)
1	7	0.00000
2	96	0.00000
3	923	0.00000
4	8165	0.00896
5	70921	0.20950
6	657984	3.41669
7	8428271	14.74276
8	92045829	85.33897

Table 4: Results and CPU time of `largest_multi_admissible([3, 5, 7, 10], 1, 10**k)` for $k \in [1, 8]$.