Project 1 - Scientific Computing

Letizia D'Achille

Description of the implementations

`number2base_rep`

The representation of a number n with respect to a given base b is obtained by repeatedly dividing the number by the base, and appending all the reminders in the reversed order. Therefore, while the number is non-zero, the remainder of the division by b is appended in front of the string, and the quotient becomes the next value to use as n. We add the same computation before the `while` loop to support the representation of 0.

`admissible`

We compute $s = (n)_b$ using the function `number2base_rep`. We first iterate over the possible lengths i of the single substring, that can be at most $\lfloor \texttt{len}(s)/2 \rfloor + 1$, since it has to be repeated twice. Then we compare pairs of characters at a distance of i, iterating over the possible positions j of the first one. We save in the variable `ind` the starting position of the second substring to compare. To verify whether there are two neighbouring identical substrings of length i, it is sufficient to check whether characters at a distance i one from another are equal, and check if we can reach i subsequent characters with this property. The comparisons have to be carried out until we reach the index preceding starting position of the second substring ($\texttt{ind} - 1$). As soon as we find a character that is not equal to the one at a distance i, we update `ind` to $i + j + 1$, since the second substring necessarily has to start one position after the latest checked character. When we find two neighbouring identical substrings we immediately return the value `False`. If the algorithm terminates without having found any witness substrings, we return `True`. Note that each pair of characters is compared once and only once, since each pair is uniquely characterized by i and j.

`count_admissible`

We initialize a counter to 0, then run a cycle over all the numbers in the range `[start, end)` checking whether they are admissible using the previous function. If a number is admissible, we add 1 to the counter.

`count_admissible_width`

The smallest number with a representation of `width` characters is $b^{\texttt{width}-1}$, the greatest is $b^{\texttt{width}} - 1$. Therefore, we apply the function `count_admissible` to these parameters, excluding the non-admissible strings "`(`b`-1)(`b`-1)[0-(`b`-1)]*`" and "`[0-(`b`-1)]*00`".

`largest_multi_admissible`

We iterate over all the numbers from the greatest to the smallest checking whether they are admissible or not with respect to every element in the list. With the help of a flag, we can end the verification as soon as we find an element in L with respect to which the number is not admissible. Moreover, if the flag remains `True` we can immediately return the current number, since it is admissible with respect to every element in the list and it is the greatest with this property (the numbers are checked starting from the greatest). If the algorithm terminates without having returned any number, we return `None` since no number in the range is admissible with respect to every element in L.

Analysis of timings and results

k	1	2	3	4	5	6
res	60	370	2715	17238	111465	776004
tim	\	0.001	0.013	0.140	1.604	17.500

Table 1: Timings (tim) and results (res) of `count_admissible`$(5, 10^k, 10^{k+1})$

k	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
res	2	4	8	12	20	28	40	52	72	96	136	176	228	304	412
tim	\	\	\	\	\	0.001	0.001	0.006	0.017	0.053	0.168	0.531	1.724	5.491	17.185

Table 2: Timings (tim) and results (res) of `count_admissible_width`$(3, k)$

k	1	2	3	4	5	6	7	8
res	7	96	923	8165	70921	657984	8428271	92045829
tim	\	\	\	0.003	0.057	0.766	3.784	22.379

Table 3: Timings (tim) and results (res) of `largest_multi_admissible`$([3, 5, 7, 10], 1, 10^k)$

Looking at the tables above we can observe that the time depends exponentially on k in all three cases. In particular, regarding the first and the third algorithm, there appears to be a relation of the type $T(k+m) \sim T(k) * 10^m$, while for the second one the relation appears to be $T(k+m) \sim T(k) * 3^m$. A more precise computation tells us that `number2base_rep`(n, b) has a complexity of $\mathrm{O}(\log_b(n))$ - the number of iterations coincides with the number of characters in the representation $(n)_b$. The algorithm `admissible`(n, b) iterates at most $\sum_{i=1}^{\lfloor \frac{\log_b(n)}{2} \rfloor} (\log_b(n) - i) \sim \log_b^2(n)$ times a constant time code, therefore it has complexity $\mathrm{O}(\log_b^2(n))$.
As a consequence, `count_admissible`$(5, 10^k, 10^{k+1})$, which calls the previous function $10^{k+1} - 10^k \sim 10^k$ times over elements with order of magnitude of $\sim 10^k$, has a complexity of $\mathrm{O}(10^k * \log_b^2(10^k)) \sim \mathrm{O}(10^k * k^2)$. Similarly, `count_admissible_width`$(3, k)$ has a complexity of $\mathrm{O}(3^k * k^2)$ and `largest_multi_admissible`$([3, 5, 7, 10], 1, 10^k)$ has again a complexity of $\mathrm{O}(10^k * k^2)$ (the length of the list L is negligible).
Neither the first nor the third sequence of numbers that we obtained are known sequences. The second one, divided by 4, corresponds to the number of dissimilar ternary square-free words of length $n+1$ (see A060688), that is indeed correlated to our problem.