Computational Complexity

The complexity of an algorithm is measured in the performance and the memory consumption.

Performance

The performance of an algorithm is the time it takes to run in relation to the input. It is usually measured using the O-Notatation ‘

Example: $\exists n_0, c \forall n > n_0 : RT(n) < c * f(n)$

The O-Notation always has the following elements:

- $\exists n_0, c \forall n > n_0$
- Constant c (The constant should be low as this indicates a good performance. In general a constant value of 1 for a linear performance is desired.)

Memory consumption

This denotes how much memory the algorithm consumes. The lower the memory consumption the lower is the computational complexity.

Exact Matching approaches

Example:

```
            1  i            n
Text:       ATTCACTATTCGGCTAT
Pattern:    GCAT
```

Question: Does pattern exists in text?

1. Approach: Look for the first pattern symbol in the text and go from there

Start with the first character of the text and the first character of the pattern and compare each character individually. If a mismatch occurs move the pattern one character and repeat the procedure. Terminate when the remaining characters in the text are less than the characters in the pattern since a match is impossible.

Algorithm:

```
for i=1,n-k
      for j=1,k
            if P[j] != T[i+j-1]
                  goto NOMATCH
      found match @
NOMATCH
```

Running time: Order of $O(n^k)$. Worst case $(n-k+1)^k = nk - k^2 + k$

The algorithm is called naïve as it is very simple but suffers poor performance.

2. Approach: Pre-processing

The idea is that information that is collected in a pre-processing step can speed up the subsequent matching process. In this particular approach so called Z-Boxes are used to identify reoccurrences of the prefix of the string later in the string.

Example:

Z[i]: length of longest common prefix of T[i..h], T[0..n]

Text:	A	T	T	C	A	C	T	A	T	T	C	G	G	C	T	A	T
Z[i]	0	0	0	0	1	0	0	4	0	0	0	0	0	0	0	2	0

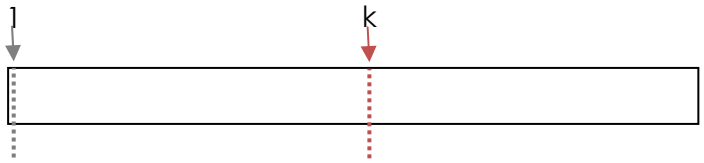
Question: *How can we compute the Z values efficiently?*

Z-Algorithm

for k := 2,…,n either case 1 or case 2 applies:

1. if **k > r** then compare the characters starting at k with the characters starting at 1. If a mismatch occurs then set Z_K to the number of characters that matched.

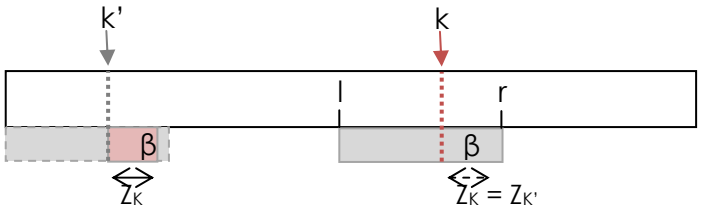
   If $Z_k > 0$, set l := k and r := k + Z_k - 1; else l,r :=0



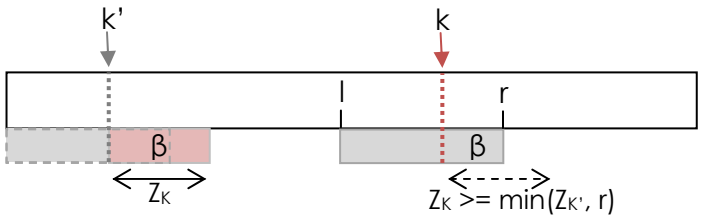
2. if **k ≤ r**, the position k is inside a Z-box S[l…r] = S[1…Z_l]. Thus S[k…r] = S[k'…Z_l].

   Let β = S[k…r];

   a) If **$Z_{k'}$ < |β|**, then the substring that contributes to ZK' lies inside the box and we can simply set Z_k := $Z_{k'}$.



a) If $Z_{K'} \geq |\beta|$, then the substring that contributes to ZK' goes beyond the current Z-Box (>r) and we can set ZK to the right side of the current box (r) and explore the succeeding characters by individually comparing it with the prefix.



Exact matching algorithm

In order to identify occurrences of the pattern in the text define

S = P$T,

where $ is a symbol that doesn't occur in the pattern and in the text. An occurrence is detected if there is a Z[i] with i equal to the length of the pattern. By using this approach detecting the pattern is done by finding the prefixes of the string S.

This algorithm does matching in linear time and it does not depend on the alphabet that we are using. If every character we see is something we haven't seen before then we get the worst computation time!

Boyer-Moore

One of the problems is that we don't know anything about the future. If we move backwards maybe we can learn something about the future along the way.

```
Text :    X P B C T B X A B P Q X C T B P Q

                  X | | | |

Pattern:      T P A B X A B
```