# 1 Zadanie

Mamy dany ciąg s składający się z liter $a_1..a_n$ o długości $n$. Tezą jest stwierdzenie, że ciąg $s$ zawiera $O(n)$ unikalnych podciągów palindromicznych.

Potraktujmy każdy znak ciągu jako osobny palindrom - jeśli wszystkie znaki są różne, teza jest prawdziwa, ponieważ mamy $n$ unikalnych podciągów palindromicznych (dalej w dowodzie będę to skracał do UPP). Kolejnym podmiotem zainteresowania są palindromy o ekwiwalentnej treści występujące w różnych częściach ciągu (co neguje ich unikalność) - w tym wypadku jednak dalej pozostajemy w granicach wyznaczonych przez tezę - co najwyżej n palindromów, więc możemy założyć dla maksymalnego wyniku unikalność wszystkich wieloliterowych rozważanych palindromów. Załóżmy istnienie palindromu o długości $i$. W takim palindromie, $b_1 = b_i, b_2 = b_{i-1}$ (zakładając, że mówimy o literach palindromu $b$ o długości $i$). W takim wypadku, $b_i$ oraz $b_{i-1}$ nie są UPP jednoliterowymi, ponieważ pokryły je odpowiedniki na początku palindromu. Z drugiej strony, palindromy zagnieżdżają się tak, że $b_2..b_(j-1)$ dla długości $j$ jest osobnym palindromem, a $b_1..b_j$ osobnym. Z tej przyczyny, podpalindromy pokrywają zanegowaną unikalność liter w drugiej połowie palindromu.

QED

# 2 Zadanie

Wejściem jest ciąg $s$, np. `bn#aan`

Algorytm:

- Sortujemy ciąg s jako tablicę znaków (tak, że `#` trafia na koniec) i traktujemy wynik tej operacji jako r. (NB. Mergesort, O(n log n))
- Przyporządkujemy każdemu znakowi z ciągu s, indeks pod którym dany znak występuje. Następnie, indeks umieszczamy w kolejce znaków w tablicy o nazwie w, na pozycji w równej kodowi ASCII znaku. (NB: `std::vector<>` `operator[]` - O(1), `std::queue<> push` - O(n log n) )
- Przyporządkujemy każdemu znakowi z ciągu r wartość z frontu kolejki z w i przypisujemy ją kolejnym elementom tablicy l. Następnie, zdejmujemy element początkowy kolejki. (NB: `std::vector<> operator[]` - O(1), `std::queue<> pop` - O(n log n) )
- Zaczynając od znaku o pozycji równej oryginalnej pozycji znaku `#` w ciągu w s, wyświetlamy jego treść znak po znaku, co iterację zmieniając wskaźnik na wartość pod jego indeksem w tablicy l.

Przykład:

- Dla ciągu `s='bn#aan'`, `r='aabnn#'`.
- w = 'a': 3, 4, 'b': 0, 'n': 1, 5, '`#`': 2

- l = 3, 4, 0, 1, 5, 2
- # jest na pozycji 2, więc zaczynamy od 0 - 0 - 3 - 1 - 4 - 5 - 2 (koniec ciągu), co odpowiada znakom: b - a - n - a - n - #, czyli ciągowi wejściowemu.

```
// Implementacja algorytmu w C++

#include <iostream>
#include <vector>
#include <algorithm>
#include <functional>
#include <queue>
#include <iterator>

// Z przyczyn technicznych, algorytm zakłada ~ zamiast #.
void print_inverted_bwt(std::string bwt) {
    // Wstępnie obliczona długość wejścia, indeks w pętli, początkowy
    // wiersz z tabeli BWT
    int input_length = bwt.length(), idx, val = bwt.find('~');

    // Dla każdego znaku przyporządkować numery wierszy z lnodes,
    // które zaczynają się zadanym znakiem.
    std::vector<std::queue<char>> corresp(256);

    // Połączenia takich samych ciągów między kolejnymi szeregami
    // transformacji Burrowsa-Wheelera i posortowanego wektora.
    std::vector<int> lnodes(input_length);

    // Posortowany klon wejścia.
    std::string bwt_sorted = bwt;

    // Zakładamy 1 bajt = 8 bitów.
    // Nie dokonujemy założeń związanych z rozmiarem char,
    // ponieważ według standardu sizeof(char) == 1.
    corresp.reserve(256);

    // Zakładając użycie mergesort, gwarantowana jest złożoność
    // wielomianowa O(n log n); w przeciwieństwie do quicksort
    // z najgorszym O(n^2).
    std::sort(std::begin(bwt_sorted), std::end(bwt_sorted));

    // W kolejce przyporządkowanej znakowi pod aktualnym indeksem,
    // umieścić aktualny indeks.
    for(idx = 0; idx < input_length; idx++) {
        corresp[bwt[idx]].push(idx);
    }
```

```
    // Stopniowo wypełnia się wektor połączeń zgodnie ze znakami
    // z posortowanej tablicy, zdejmując pierwsze elementy kolejek.
    for(idx = 0; idx < input_length; idx++) {
        lnodes[idx] = corresp[bwt_sorted[idx]].front();
        corresp[bwt_sorted[idx]].pop();
    }

    // Wyświetla się znaki z ciągu wejściowego o indeksach równym
    // wartościom z lnodes, zaczynając od input_length.
    for(idx = 0; idx < input_length; idx++) {
        std::cout << bwt[val = lnodes[val]];
    }
}

int main(void) {
    print_inverted_bwt("bn~aan");
}
```

# 3 Zadanie, podpunkt A

Założenie: jest dany ciąg $p$ o długości $n$. Teza: $p$ można podzielić na co najwyżej $n$ zbiorów przedziałów podciągów kończących się we wspólnym punkcie.

Ilość palindromów w ciągu może sięgnąć nawet $n^2$, ale punkty końcowe ograniczające paczki dalej będą te same, więc, jako że ciąg ma długość $n$, to można stworzyć $n$ zbiorów przedziałów podciągów, każdy kończący się na kolejnej literze tak, że wszystkie poprzednie litery są elementami przedziału, tj. dla ciągu $a_1..a_n$ i punktu $m$, w przedziale zawarte będą elementy $a_1..a_m$. QED

# 4 Uzasadnienie

```
; reszta znajduje się w submisji na Szkopule (program) ;)
; złożoność: O(n^2)
levenshtein (s1, s2, s):
    x, y, matrix[s+1][s+1]
    matrix[0][0] = 0;
    for (x = 1; x <= s; x++)
        matrix[x][0] = matrix[x-1][0] + 1;
    for (y = 1; y <= s; y++)
        matrix[0][y] = matrix[0][y-1] + 1;
    for (x = 1; x <= s; x++)
        for (y = 1; y <= s; y++)
            matrix[x][y] = MIN3(
                matrix[x-1][y] + 1,
```

```
                matrix[x][y-1] + 1,
                matrix[x-1][y-1] + (s1[y-1] == s2[x-1] ? 0 : 1)
            );
    matrix[s][s];

; bazowane na O(n^2) w pętli do (prawie) końca ciągu.
; złożoność wielomianowa i kwadratowa.
main:
    tl, pl, m;
    vector, text, pattern;

    read tl;
    read pl;

    text = string(tl + 2);
    pattern = string(pl + 2);

    read text;
    read pattern;

    for idx = 0 to tl - pl + 1, step 1:
        if levenshtein(text + idx, pattern, pl) <= 1:
            vector add idx
            m = m + 1

    vector foreach(e -> e + 1)

    print m
    print vector
```