4 Algorytm Karpa-Millera-Rosenberga

Algorytm Karpa-Millera-Rosenberga (w skrócie KMR) jest algorytmem wyszukiwania wzorca w tekście. Dzięki swojej interesującej budowie, znajduje on także zastosowanie w wielu innych problemach tekstowych. KMR opiera się na utworzeniu dla słowa pewnej struktury zwanej **słownikiem podsłów bazowych**. Struktura ta ma rozmiar $\Theta(n \log n)$ (gdzie n jest długością słowa) i może być zbudowana w takim samym czasie.

Jak zwykle w analizie algorytmów tekstowych zakładać będziemy, że alfabet ma stały rozmiar.

4.1 Słownik podsłów bazowych

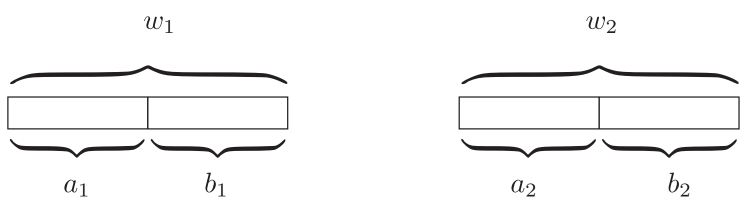
Konstrukcja słownika podsłów bazowych polega na obliczeniu haszy dla wszystkich podsłów wyjściowego słowa o długości 2^p dla $p = 0, 1, \ldots, \lfloor \log p \rfloor$. Tym razem jednak nie możemy użyć haszowania, takiego jak w rozdziale 3, gdyż będziemy wymagali od haszy nieco mocniejszych własności. Chcemy bowiem nie dopuścić do sytuacji, w której różne słowa tej samej długości otrzymają te same hasze.

Przyjmijmy, że budujemy strukturę słownika podsłów bazowych dla słowa $S = s_0 s_1 \ldots s_{n-1}$. Przez $id_p[t]$ oznaczmy hasz podsłowa długości 2^p zaczynającego się na pozycji t (jeśli takie słowo nie istnieje to przyjmujemy, że $id_p[t] = \infty$). Tablice $id_p[]$ będziemy konstruować dla kolejnych p zaczynając od $p = 0$. Tablica $id_0[]$ reprezentuje słowa długości 1 czyli pojedyncze litery. Doskonałym haszem dla pojedynczej litery jest jej pozycja w alfabecie (tzn. **a** = 0, **b** = 1, etc.). A zatem $id_0[t] = s_t$. Takie identyfikatory spełniają oczywiście założenie unikalności.

Zastanówmy się teraz, jak obliczyć tablicę id_{p+1} mając obliczoną tablicę id_p. Korzystać będziemy z następującego, oczywistego faktu:

> **Fakt 1.** Słowa $s[i \ldots i+2^{p+1}-1]$ i $s[j \ldots j+2^{p+1}-1]$ są równe wtedy i tylko wtedy, gdy są równe słowa $s[i \ldots i+2^p-1]$ i $s[j \ldots j+2^p-1]$ oraz są równe słowa $s[i+2^p \ldots i+2^{p+1}-1]$ i $s[j+2^p \ldots j+2^{p+1}-1]$.

Innymi słowy, dwa słowa są równe gdy ich lewe połowy są równe oraz ich prawe połowy są równe. Ilustruje to poniższy rysunek:



$$w_1 = w_2 \Leftrightarrow a_1 = a_2 \wedge b_1 = b_2$$

Na podstawie Faktu 1 można zauważyć, że dobrym haszem $id_{p+1}[t]$ jest para $(id_p[t], id_p[t+2^p])$. My jednak nie chcemy przechowywać par tylko zamienić je na nowe identyfikatory w postaci kolejnych liczb naturalnych. Aby to osiągnąć, musimy posortować leksykograficznie wszystkie takie pary identyfikatorów a następnie, przeglądając je od najmniejszych, przydzielać nowe hasze w postaci kolejnych liczb naturalnych. Oczywiście takie same pary muszą otrzymać ten sam hasz.

Warto w tym miejscu zaznaczyć, że ważne jest aby na każdym etapie sortować pary identyfikatorów w kolejności niemalejącej. Jak wkrótce zobaczymy, będzie to miało kluczowe znaczenie przy porównywaniu leksykograficznym podsłów. Zauważmy, że hasze dla każdej długości słów są liczbami z przedziału $[0, max(n, |\Sigma|)]$, przez co możemy sortowanie par zaimplementować w czasie liniowym np. używając sortowania kubełkowego.

Oto (nieco okrojona) implementacja funkcji `build_KMR(s)` konstruującej słownik podsłów bazowych dla słowa s i zapisującej go w tablicy `ids[]`.

KAPITAŁ LUDZKI NARODOWA STRATEGIA SPÓJNOŚCI
Warszawska Wyższa Szkoła Informatyki
UNIA EUROPEJSKA EUROPEJSKI FUNDUSZ SPOŁECZNY
Projekt współfinansowany ze środków Unii Europejskiej w ramach Europejskiego Funduszu Społecznego

```
/* Para identyfikatorów, których zbiór będziemy sortowali.
   Aby móc zapisać wynikowy hasz musimy pamiętać też indeks
   słowa, któremu odpowiada dana para */

struct Pair {
    int p1, p2, index;
};

/* tablica (dwuwymiarowa) identyfikatorów */
vector<vector<int> > ids;

void build_KMR(string &s){
    vector<int> curr_id(s.size());
    vector<Pair> pairs;

    /* hasze słów długości 1 to kody symboli alfabetu */
    for(int i = 0; i < s.size(); ++i)
        curr_id[i] = (int) s[i];
    ids.push_back(curr_id);

    int p = 2;
    while(p < s.size()){   /* dla kolejnych potęg dwójki p */
        curr_id.resize(s.size() - p + 1);
        pairs.resize(s.size() - p + 1);
        for(int i = 0; i <= s.size() - p; ++i){
            pairs[i].p1 = ids.back()[i];
            pairs[i].p2 = ids.back()[i + p/2];
            pairs[i].index = i;
        }

        /* posortuj kubełkowo pary identyfikatorów */
        bucket_sort(pairs);

        /* przydziel nowe identyfikatory */
        int new_id = 0;
        for(int i = 0; i < pairs.size(); ++i){
            if(i > 0 && pairs[i-1] != pairs[i])
                ++new_id;
            curr_id[pairs[i].index] = new_id;
        }
        ids.push_back(curr_id);
        p *= 2;
    }
}
```

W powyższym fragmencie kodu pominięty został algorytm sortowania kubełkowego.

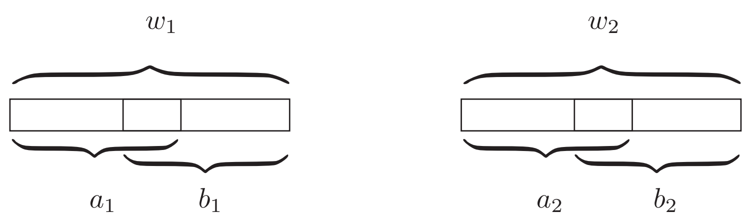
Ćwiczenie 8. Jak zmieni się złożoność funkcji `build_KMR()` gdy zamiast sortowania kubełkowego zastosujemy sortowanie przez scalanie? Zaimplementuj obie wersje algorytmu i porównaj czasy ich działania dla długich słów (w drugiej wersji możesz wykorzystać funkcję `sort()` z STL).

KAPITAŁ LUDZKI
NARODOWA STRATEGIA SPÓJNOŚCI
Warszawska Wyższa Szkoła Informatyki
UNIA EUROPEJSKA
EUROPEJSKI FUNDUSZ SPOŁECZNY
Projekt współfinansowany ze środków Unii Europejskiej w ramach Europejskiego Funduszu Społecznego

4.2 Porównywanie podsłów

Mając zbudowany słownik podsłów bazowych możemy w czasie stałym sprawdzić, czy dwa podsłowa są równe. Oczywiście jest sens sprawdzać równość tylko słów tej samej długości. Jeśli długość porównywanych słów jest potęgą dwójki, sprawdzenie równości sprowadza się do porównania haszy z odpowiedniej tablicy. Co, jeśli tak nie jest? Można skorzystać z prostej obserwacji, podobnej do Faktu 1:

Fakt 2. Niech dane będą słowa s_1 i s_2 długości n. Niech p będzie największą liczbą całkowitą, taką, że $2^p \leqslant n$. Słowa s_1 i s_2 są równe wtedy i tylko wtedy, gdy są równe słowa $s_1[0 \ldots 2^p - 1]$ i $s_2[0 \ldots 2^p - 1]$ oraz są równe słowa $s_1[n - 2^p \ldots n - 1]$ i $s_2[n - 2^p \ldots n - 1]$.



$$w_1 = w_2 \Leftrightarrow a_1 = a_2 \wedge b_1 = b_2$$

Korzystając z Faktu 2 wystarczy porównać najdłuższe prefiksy oraz najdłuższe sufiksy, których długości są potęgami dwójki. Jeśli założymy, że na początku stablicowaliśmy dla każdej możliwej długości podsłowa największą potęgę dwójki nie większą niż ta długość, porównywanie podsłów można zrealizować w czasie stałym.

Jeśli przy konstrukcji słownika podsłów bazowych, za każdym razem sortowaliśmy pary identyfikatorów niemalejąco, to możemy w analogiczny sposób zrealizować w czasie stałym porównywanie leksykograficzne podsłów. W terminach oznaczeń na powyższej ilustracji oznacza to, że wystarczy porównać leksykograficznie parę (a_1, b_1) z parą (a_2, b_2). Co zrobić w przypadku słów różnej długości? Najpierw porównać krótsze słowo z prefiksem dłuższego o tej samej długości. W przypadku, gdy te okażą się równe, mniejsze leksykograficzne będzie słowo krótsze.

4.3 Najdłuższe powtarzające się podsłowo

Zastanówmy się najpierw, jak sprawdzić, czy w słowie istnieją dwa równe podsłowa ustalonej długości. Każdemu takiemu podsłowu odpowiada para identyfikatorów dwóch krótszych podsłów (jak na ilustracji do Faktu 2). Możemy wszystkie takie pary posortować kubełkowo w czasie liniowym a następnie sprawdzić czy nie ma dwóch takich samych. Jeśli dodatkowo, obok pary zapamiętamy indeks w tablicy, będziemy mogli takie powtarzające się podsłowo łatwo odtworzyć.

Jeśli w słowie istnieją dwa równe podsłowa długości d, to istnieją również dwa równe podsłowa każdej długości mniejszej od d (są nimi na przykład prefiksy podsłów długości d). Wykorzystując tę obserwację możemy znaleźć najdłuższe powtarzające się podsłowo w czasie $O(n \log n)$ stosując wyszukiwanie binarne po jego długości.

Ćwiczenie 9. Jak znaleźć najdłuższe podsłowo, które występuje w tekście **dokładnie** k razy?

4.4 Najdłuższe wspólne podsłowo

Algorytm znajdowania najdłuższego wspólnego podsłowa dwóch słów s_1 i s_2 jest analogiczny do algorytmu szukającego powtarzającego się podsłowa. Wystarczy zbudować słownik podsłów bazowych dla słowa $s_1 \# s_2$ (gdzie „$\#$” jest symbolem nie występującym w alfabecie), a następnie znaleźć najdłuższe powtarzające się podsłowo. Aby zagwarantować, że znalezione podsłowo występuje zarówno w s_1 jak i w s_2 (a nie na przykład dwukrotnie w s_1) musimy przy sortowaniu par identyfikatorów pamiętać dodatkowo, z którego słowa pochodzi dana para i akceptować tylko powtórzenie pary z różnych słów.