WYDZIAŁ MATEMATYKI I INFORMATYKI

UNIWERSYTET ŁÓDZKI

KHARYBINA YULIIA

NR albumu: 377939

Kierunek: Informatyka

**„Algorytmy kompresji danych”**

Praca licencjacka

Napisana w Katedrze Informatyki Stosowanej

Pod Kierunkiem dr

Wojciecha Horzelskiego

**Łódź 2020**

# Spis treści

[Spis treści 2](#_Toc47537913)

[Wstęp 3](#_Toc47537914)

["Pojęcie kompresja” 3](#_Toc47537915)

[Cel 3](#_Toc47537916)

[Kompresja stratna i bezstratna 3](#_Toc47537917)

[Opis kompresji 4](#_Toc47537918)

[Współczynnik redundancji 4](#_Toc47537919)

[Opis aplikacji 5](#_Toc47537920)

[Architektura aplikacji jako wyjściowego produktu 5](#_Toc47537921)

[Algorytmy 5](#_Toc47537922)

[RLE+BWT 5](#_Toc47537923)

[Kody bez pamięci 7](#_Toc47537924)

[Huffman Encoding 7](#_Toc47537925)

[Integer encodings 9](#_Toc47537926)

[Kod unarny 9](#_Toc47537927)

[Elias gamma code 9](#_Toc47537928)

[Elias delta code 10](#_Toc47537929)

[Elias omega code 11](#_Toc47537930)

[String RePair 12](#_Toc47537931)

[IRT – aktualizowana wersja BWT 14](#_Toc47537932)

[Kompresja genomu 15](#_Toc47537933)

[Mierzenie czasu 19](#_Toc47537934)

[Bibliografia 19](#_Toc47537935)

# Wstęp

W dzisiejszym szybko rozwijającym się informacyjnym świecie dość ostro stoi pytanie przechowywania i przekazania informacji. Nie zważając na bez ustanku rosnącą objętość nośników, czasami konieczne jest przechowywanie dużej ilości danych w pamięci o małej pojemności (na przykład na pen drive). Aby zmniejszyć rozmiar, stosuje się specjalne algorytmy — tak zwane algorytmy kompresji.

## "Pojęcie kompresja”

**Kompresja danych** (ang. *data compression*) – polega na zmianie sposobu zapisu informacji tak, aby zmniejszyć redundancję i tym samym objętość zbioru. Innymi słowy, aby bez utraty treści pliku oraz jakości zmniejszyć liczbę wykorzystywanych bitów. Służy do bardziej racjonalnego wykorzystania urządzeń do przechowywania i przesyłania danych.

Działaniem przeciwnym do kompresji jest dekompresja.

## Cel

Zdobyć trochę doświadczenia w ocenie wydajności i porównaniu różnych technik kompresji. Zdobyć trochę doświadczenia w skalowaniu różnych technik kompresji i ocenie ich zachowania podczas skalowania.

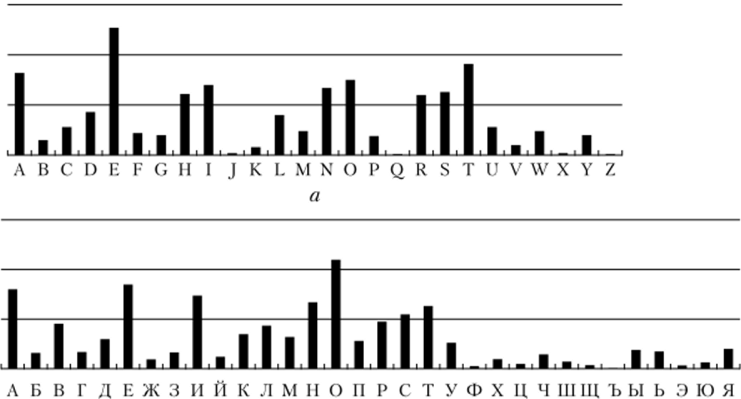
Główne cechy techniczne procesów kompresji i wyniki ich pracy to:

* stopień kompresji (compress rating) lub stosunek (ratio) objętości źródła i wyniku;
* stopień kompresji - czas kompresji pewnej ilości informacji;
* jakość kompresji - wartość pokazująca, ile zapakowany jest wynik poprzez zastosowanie do niego ponownej kompresji przy użyciu tego samego lub innego algorytmu.

## Kompresja stratna i bezstratna

Kompresja bezstratna – kompresja w której z postaci skompresowanej można odzyskać identyczną postać pierwotną. Stratna -w której takie odzyskanie jest niemożliwe, jednak główne właściwości zostają zachowane. Podczas przesyłania informacji graficznych lub audio stosuje się kompresję stratną, biorąc pod uwagę niedoskonałość narządów słuchu i wzroku, które nie zauważają pogorszenia jakości związanego z tymi stratami.

Bezstratna kompresja informacji odbywa się za pomocą kodowania statystycznego lub na podstawie wcześniej utworzonego słownika. Algorytmy statystyczne (np. Schemat kodowania Huffmana) przypisują określony kod do każdego znaku wejściowego. W tym przypadku najczęściej używanemu symbolowi przypisywany jest najkrótszy kod, a najrzadziej - dłuższy.  Rozkład częstotliwości poszczególnych liter alfabetu angielskiego pokazano na rys. 1. Taki rozkład można zbudować dla innych języków.



Wielkość kompresji zależy od nadmiarowości przetwarzanej tablicy bitów. Każdy z języków naturalnych ma pewną redundancję. Spośród języków europejskich rosyjski ma jeden z najwyższych poziomów nadmiarowości. Można to ocenić na podstawie rosyjskiego tłumaczenia tekstu angielskiego. Jeśli chodzi o tekst poetycki, nadmiarowość może być nawet dwa razy wyższa.

Są algorytmy zastosowania ogólnego, ale najczęściej do każdego typu danych zastosujemy różne algorytmy kompresji, ponieważ dla różnych typów danych koniecznie jest zastosowanie różnych właściwości. Na przykład, jeśli dokonujemy kompresji na obrazku, nie widać znaczących różnic w stosunku do oryginału. Ale to już nie możemy wykorzystywać do wydruku, bo w tym przypadku wymaga się zachowanie innych właściwości.

Przykłady algorytmów kompresji bezstratnej: BWT, kodowanie Shannona, Shannona-Fano, Huffmana, arytmetyczne, LZ77, RLE.

Przykłady algorytmów kompresji stratnej: JPEG, DivX/XviD, zmodyfikowana transformata kosinusowa, transformat falkowy.

## Opis kompresji

Kompresja zmniejsza ilość miejsca wymaganego do przechowywania plików i czas potrzebny na przesłanie danych. Kompresja jest formą kodowania. Dzięki usunięciu nadmiarowości z tekstu kompresja ułatwia szyfrowanie, co utrudnia znalezienie szyfru za pomocą metody statystycznej dostępnej dla hackera. Istnieje dość duża liczba ich różnych odmian. W prace pokażę podstawowe informacje na temat typów archiwizacji i kompresji, a także implementacji algorytmów BWT, RLE, LZ77 i td w języku Python.

## Współczynnik redundancji

Współczynnik redundancji komunikatu A jest określony wzorem

r = (Imax - I) / Imax,

gdzie I to ilość informacji w komunikacie A, Imax to maksymalna możliwa ilość informacji w wiadomości o tej samej długości co A.

Przykład redundancji podano w wiadomościach w językach naturalnych, na przykład w języku rosyjskim r jest w zakresie 0,3 ... 0,5. Obecność redundancji pozwala nam podnieść kwestię kompresji informacji bez utraty jej w przesyłanych wiadomościach.

Kompresja danych, które nie mają właściwości redundancji, jest niemożliwa bez strat. Ponadto kompresowanie zaszyfrowanych informacji zwykle nie jest możliwe.

Wymagane właściwości kompresji ciągów dla baz danych:

* musi być w stanie skutecznie czytać n-te słowo
* dobry współczynnik kompresji
* bezstratna dekompresja
* wyszukiwanie prefiksów, sufiksów i podciągów
* wstawia, usuwa, aktualizuje bez pełnej dekompresji, np. powinien być w stanie usunąć n-te słowo.

# Opis aplikacji

Aplikacja służy do przedstawienia roboty algorytmów BWT, RLE, Huffmana, RePiar, IRT (ulepszony BWT), także porównaina algorytmów, czasu ich działania, wyników działania. Applikacja pobiera dane z pliku alice.txt za pomocą metody random\_text(), która generuje text. Funkcja przyjmuje tekst z pliku tekstowego „alice.txt” i losowo wybiera spośród pliku kawalek (100 znaków każdy). W rezultacie, kawałki są połączone i zwracane. Można wybrać jakikolwiek inny plik. W wyniku pracy algorytmów możemy zobaczyć skompresoane dane, rozmiar danych wejściowych i wyjsciowych oraz czas dzałania ałgorytmu z tymi danymi.

GUI aplikacji jest zrobiony przy pomocy PyQt5.

# Architektura aplikacji jako wyjściowego produktu

## Opis klas:

**Class BWT** - klasa służąca do kompresowania danych przy pomocy algorytmu BWT ta RLE. Konstruktor: def \_\_init\_\_(self, s). Metody: transform(self), ibwt(self, bwt), rle\_encode(self, data), rle\_decode(self, data). Naipierw wywołana jest metoda transform (), w której tworzę tablicę obrotów danych i wyciągam ostatnią kolumnę tej tablicy. To jest wynik pracy algorytmu BWT. Po wykorzystaniu algorytmu BWT posługujemy metodą rle\_decode(data), która w pętli liczy iiłość powtarzających się znaków i generuję wynik. W tej klasie są także zaimplementowane funkcji decodowania BWT ta RLE.

**Class Huffman** -klasa do realizoania algorytmu Huffmana. Konstruktor: def \_\_init\_\_(self, s). Metody: Huffman\_code(self, h), computeFrequencises(self, s), encode(self, h), Huffman\_decode(self, enStr, code). Działanie całego algorytmu zaczyna się od metody computeFrequencises(s) która liczy numer wystąpienia każdego symbolu z ciągu.

Zatem korzystam z funkcji huffman\_code(h), która koduje dane przy pomocy struktury heap (heapq w python3). Do naszej listy h dodajemy przy pomocy append() freq- częstość wystąpina danego symolu, len(h) – unikalny licznik, Leaf(ch) – liść. Heapq.heapify(h) – funkcja, która transformuje list w heap. W pętli while budujemy drzewo huffmana, usuwając lewy i prawy element i dodawając węzeł. W następnej części budujemy kod Huffmana, wykorzystając metodę walk(). Także jest funkcja huffman\_decode(), która dekoduje kod huffmana przy pomocy pętli while. Pętla while obhodzi ciąg i generuje tekst za pomocą słownika.

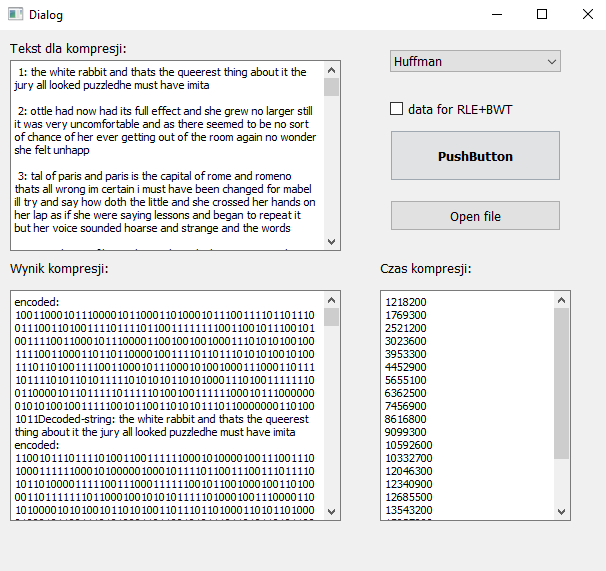
**Class Node, Leaf**  - pomocnicze klasy do realizoania drzewa Huffmana. Metody: def walk(self, code, acc), walk(self, code, acc).

**Class RePair** - klasa realizująca algorytm RePair. Konstruktor: def \_\_init\_\_(self, s). Metody: findMFD(self, s), replaceMFD(self, digram, s, nonT), repair(self, s, ch, d), encode(self, rules, symb), func(self, s), decomp(self, rules, s). W metodzie RePair są tworzone digramy(bigramy) za pomocą funkcji zip(). Dalej liczymy wystąpienia digramów w ciągu (Counter(bigrams)) i zwracamy najczęsty element counts.most\_common(). Zamieniamy digram na nonTerminal. Zapisujemy w słownik digram i odpowiedni mu nonTerminal.

**Class binaryBWT** klasa służąca do kompresowania danych DNA(A,C,G,T) przy pomocy ulepszonego algorytmu BWT. Konstruktor: def \_\_init\_\_(self, s). Metody: DNA2binary(self, s), binary2DNA(self, bString), rank(self, char, int, list), count(self, char, rank, list), bin2BWTs(self, binaryInput), BWTs2bin(self, E1, BWT1, BWT2).

**Class random\_text** – klasa, która przyjmuje tekst z pliku tekstowego „alice.txt” i losowo wybiera spośród pliku kawalek (100 znaków każdy). W rezultacie, kawałki są połączone i zwracane. Konstruktor: def \_\_init\_\_(self, file\_name). Metody: makeRandomText(self, i).

Interfejs programu składa się z textarea1, który pokazuje tekst do kompresji. ComboBox wyświetła algorytm, który chcemy wykorzystać do kompresji danych. Przy pomocy przycisku „Open file” możemy wybrać potrzebny nam plik. Przycisk „PushButton” zaczyna działanie programu. Textarea2 wyświetla skompresowane dane. Textarea3 wyświetla czas kompresji.



# Algorytmy

Aby wszystkie nasze algorytmy działały poprawnie, wystarczy przypisać dowolny znak do końca linii, który jest oczywiście mniejszy niż jakikolwiek znak, z którego linia może się składać (na przykład może to być dolar lub ostry; w C można w tym celu użyć istniejącego znaku zerowego) . W danej prace znak końca linii „$”.

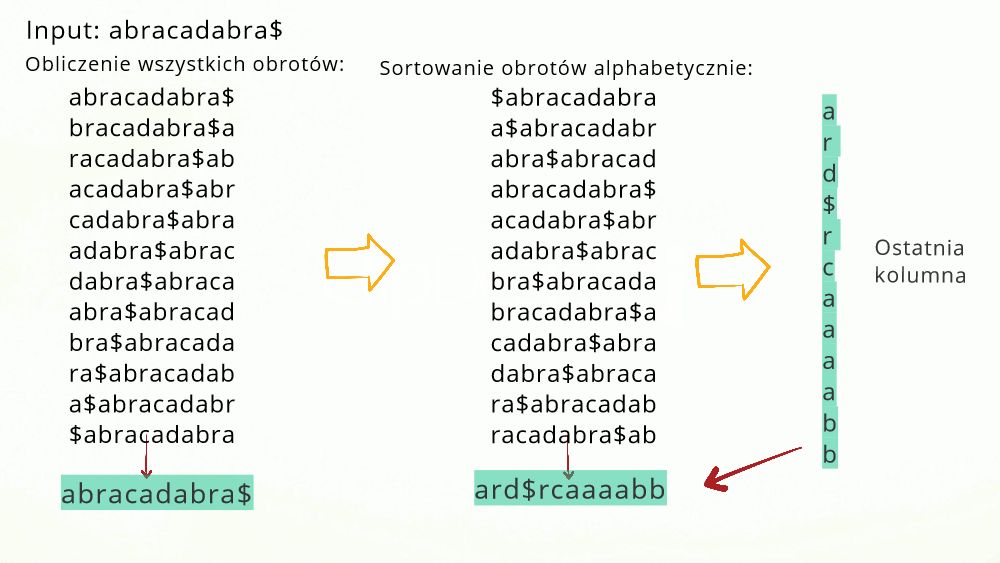
## RLE+BWT

Znakomita większość plików posiada w swej strukturze pewne powtarzające się wyrażenia, czy fragmenty wyrażeń. Żeby uniknąć redundancji, w takim przypadku stosujemy algorytm RLE.  
RLE — algorytm kompresji danych, który zastępuję powtarzające się znaki (serie) w słowie, jednym znakiem i liczbą jego powtórzeń. Jeżeli w pliku mamy: DDDDDAAAAABBBBBBBBB, to wykorzystując algorytm RLE, otrzymujemy znacznie prostszą formę: 5D5A9B. Długość pierwszego słowa – 19, długość wyniku – 6.

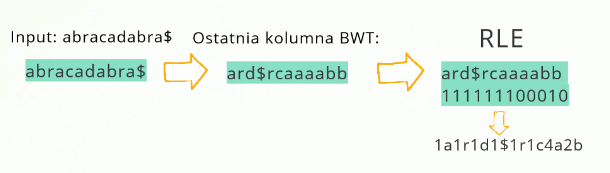
Inne przykady:



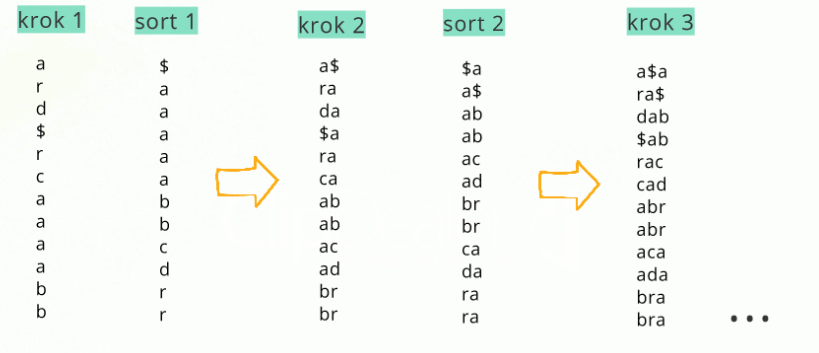
Algorytm jest wydajny, kiedy jest wiele powtórzeń po kolei. W przypadku ze słowem abracadabra otrzymujemy wynik: 1a1b1r1a1c1a1d1a1b1r1a. Długość wstępnych danych – 11, długość danych na wyjściu 22. Czyli w danym przypadku przed zastosowaniem RLE wykorzystujemy BWT, żeby otrzymać lepsze wyniki.  
  
BWT - algorytm, który jest wykorzystywany w różnych technikach kompresji do przekształcania danych.  
Ideą BWT jest zbudowanie tablicy, której wiersze są cyklicznymi przesunięciami łańcucha wejściowego w kolejności słownikowej i zwrócenie ostatniej kolumny tablicy, która zwykle ma długie serie identycznych znaków. Dokładnie po której wykorzystujemy RLE.



Wykorzystanie RLE+BWT:



Dekompresja. Żeby odtworzyć słowo, musimy posortować ostatnią kolumną, zatem dodać do posortowanej kolumny ostatnią kolumną na początek i znów posortować. Powtarzać etapy. Do ostatniego kroku proces jest identyczny. Na wyjściu otrzymujemy abracadabra$ w tym wierszu, który kończy się znakiem „$”. Tekst kończy się specjalnym znakiem „$”, który jest unikalna i leksykograficznie mniejszy niż jakikolwiek inny znak.



Ta metoda jest skuteczna podczas przesyłania obrazów rastrowych, ale jest mało przydatna podczas przesyłania tekstu.

## Kody bez pamięci

Najprostsze kody, na podstawie których można przeprowadzić kompresję danych, to kody bez pamięci (*code without memory*).

W kodzie bez pamięci każdy znak w zakodowanym wektorze danych jest zastępowany słowem kodowym z zestawu prefiksów sekwencji binarnych lub słów.  
Przykładami kodów bez pamięci są kodowanie Huffmana i kodowanie Shannon-Fano.

Zalety kodów bez pamięci:  
Te kody są prefiksami, co upraszcza dekodowanie, dlatego często są preferowane.  
Przy użyciu tej metody kodowania można uzyskać krótsze kody niż przy użyciu kodu o stałej długości. Można odkodować wiadomość, gdy dotrze, nie odbierając jej w całości.

Wady kodów bez pamięci:

Kody Huffmana i Shannona-Fano są optymalne, ale wciąż mają szereg wad.  
Podczas kodowania przy użyciu metod Huffmana lub Shannona wykorzystywane są prawdopodobieństwa wystąpienia znaków alfabetu w tekście. To znaczy, aby zbudować kod, musimy mieć te informacje. Dlatego musimy wcześniej znać całą zakodowaną sekwencję.  
Aby dekoder mógł odszyfrować plik, tabela częstotliwości używana przez koder powinna zostać zapisana w skompresowanym pliku. Dlatego długość skompresowanego komunikatu jest zwiększana o długość tabeli częstotliwości, która powinna być wysłana przed danymi, co może nie uzasadniać kompresji. Chociaż w przypadku kodów Huffmana można optymalnie przenieść tabelę.  
Konieczność posiadania pełnych statystyk częstotliwości przed rozpoczęciem samego kodowania wymaga dwóch przejść w komunikacie: jeden do budowy modelu komunikatu (tabela częstotliwości i drzewo kodowania Huffmana), a drugi do samego kodowania.

## Huffman Encoding

Kod jest regułą przekształcania fragmentu informacji (na przykład litery, słowa lub wyrażenia) w inny — zwykle skróconą forma lub reprezentacja (jeden znak w inny znak), niekoniecznie tego samego typu. W niektórych zastosowaniach głównie przy przesyłaniu informacji podlegających utajnieniu, zwany jest szyfrem. Kody są stosowane m.in. w telegrafii, telefonii, w technice cyfrowej.

Niektóre przyczyny kodowania:

* Sytuacje, w których „prosty język” jest trudny (komunikacja na odległość, różne języki, analfabetyzm)
* Zapewnienie poufności komunikacji (kody kryptograficzne)
* Kody są krótsze, niż oryginalne informacje. Oszczędzaj koszty transferu i / lub koszty przechowywania
* Kody korekcji błędów

Przykłady kodów: kod Morse’a, kod genetyczny, kod NATO, Kodowanie Huffmana.

**Kodowanie Huffmana** (ang. Huffman coding) – jedna z najprostszych i łatwych w implementacji metod kompresji bezstratnej. Została opracowana w 1952 roku przez Amerykanina Davida Huffmana.

Idea leżąca u podstaw kodowania Huffmana opiera się na częstotliwości występowania znaku w sekwencji. Znak występujący najczęściej w sekwencji otrzymuje nowy, bardzo mały kod, a znak, który występuje najrzadziej, otrzymuje, wręcz przeciwnie, bardzo długi kod.

Algorytm Huffmana nie należy do najefektywniejszych obliczeniowo systemów bezstratnej kompresji danych, dlatego też praktycznie nie używa się go samodzielnie. Często wykorzystuje się go jako ostatni etap w różnych systemach kompresji, zarówno bezstratnej, jak i stratnej, np. MP3 lub JPEG. Pomimo że nie jest doskonały, stosuje się go ze względu na prostotę oraz brak ograniczeń patentowych. Jest to przykład wykorzystania algorytmu zachłannego.

Cechy **kodowanie Huffmana:**

* Algorytm bezstratnego kodowania
* Kody o zmiennej długości
* Kody bez prefiksów

Rodzaje kodowania Huffmana:

* Algorytm dynamicznego kodowania Huffmana
* Algorytm statycznego kodowania Huffmana

Statyczne kodowanie Huffmana (nie adaptacyjne) wymaga częstotliwości wszystkich liter. Alternatywa: użyć „ogólnej” tabeli częstotliwości (np. Średnich częstotliwości języka) - Wada: jeśli rozkład częstotliwości tekstu różni się, kompresja nie jest „optymalna”

Adaptacyjny Huffman to drzewo Huffman, które jest obliczane „na bieżąco”. Przy każdym nowym symbolu zmieniają się względne częstotliwości liter – w razie potrzeby po każdym symbolu należy dostosować drzewo. Podczas dekompresji należy wykonać te same czynności na drzewie Huffman, co przy kompresji – dekompresor zna ten sam stan drzewa Huffman. Kilka algorytmów do obliczenia takiego adaptacyjnego drzewa Huffmana - FGK: Faller (1973), Gallagher (1978) i Knuth (1985) - Algorytm Jeffreya Scotta Vittera (1987) daje bardziej zrównoważone drzewa.

Przykład statycznego (nie adopcyjnego) kodowania Huffmana:



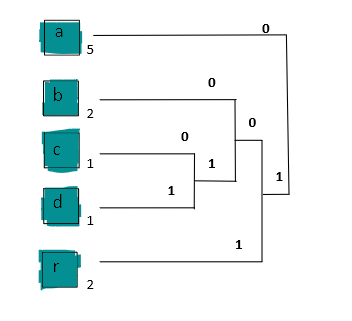
Jak obliczyć tabelę Huffmana:

S = a b r a c a d a b r a

1 Krok: Oblicz tabelę częstości występowania liter(znaków)

2 Krok: Oblicz drzewo huffmana

* Połącz dwa „węzły” z najniższą całkowitą częstotliwością
* Przypisz do każdej górnej gałąźi „0” bit, a do każdej dolnej gałęzi „1”



S = a b r a c a d a b r a

Wynik: Huffman(S) = 0 100 101 0 110 0 111 0 100 101 0

## Integer encodings

Jedna z najwcześniejszych technik (1949). Tworzy drzewo binarne, które reprezentuje prawdopodobieństwo wystąpienia każdego ze znaków i nawzajem.  
  
Kod symbolu jest uzyskiwany przez przeszukanie drzewa i dodanie 0 lub 1, w zależności od tego, gdzie pójdziemy. Na przykład ścieżka do „A” - jego kod to „110”. Algorytm nie zawsze podaje optymalne kody do konstruowania drzewa od podstaw. Dlatego używany jest teraz algorytm Huffmana, odpowiedni dla różnych danych.

### Kod unarny

**Kod unarny** to jest kod prefiksowy, w którym słowa kodowe składają się z ciągu bitów tej samej wartości zakończonymi bitem o wartości przeciwnej. Czyli oznacza każdą liczbę całkowitą n z n-1-„0” bitów, a następnie jedną „1” bitową (koduje n>0).

Inne reprezentacje:

* n-„0” bitów, po których następuje jeden 1 bit (koduje n>=0)
* n-1 „1”-bitowy, po którym następuje jeden „0”-bitowy bit
* n „1”bity, a następnie jeden bit „0”

Przykład:

* U(1) = 1
* U(14) = 00000000000001
* Dla listy L=[1,4,4,3,5,4,2,4]:

U(L) = 10001001001000010001010001

Nawet jeśli skoczyć losowo do kodu, można odczytać bieżący symbol: Kody unarne są synchronizowane automatycznie.

Problem jest w tym, że kody unarne rosną bardzo szybko. Czyli bardzo niewydajne w przypadku kodowania dużych liczb.

### Elias gamma code

**Kodowanie Eliasa** – sposób kodowania liczb całkowitych większych od zera, za pomocą słów kodowych o zmiennej długości; liczba bitów jest proporcjonalna do kodowanej wartości. Istnieją trzy sposoby kodowania: gamma(γ) ( γ ) , {\displaystyle (\gamma ),}delta(δ) ( δ ) {\displaystyle (\delta )}oraz omega (ω).

Kodowanie jest używane m.in. w różnych metodach kompresji, wymagających zapisywania liczb z szerokiego przedziału wartości (np. przesunięć w metodzie LZR, pochodnej LZ77).

Pomysł Elias-gamma-kodowania jest taki, żeby dodać informacje o długości w formie kodów unarnych. To powoduje, że informacje te mogą być zdekodowane w sposób niepowtarzalny.

Elias-𝛾-kod reprezentuje liczbę całkowitą n jako parę selektora (selector) i przesunięcia (offset):

- Przesunięcie to minimalny kod binarny n bez jego wiodącego 1-bitu

- Selektor to długość jako kod unarny (lub długość przesunięcia 1 jako unarny kod)

Przykład:

* B(1) = 1
* B(4) = 00100
* B(5) = 00101
* L=[1,4,4,3,4,5,2]

B(L) = 100100001000110010000101010

To, co jest podkreślono to jest długość liczby w formie dwójkowej w zapisie unarnym.

Ale kodowanie Elias gamma wciąż zawiera nadmiarowe informacje, które zmniejszają wydajność kompresji.

Przykłady:

|  |  |  |
| --- | --- | --- |
| n |  | (n) |
| **1** | **1** | **0** |
| **2** | **10** | **01 0** |
| **3** | **11** | **01 1** |
| **4** | **100** | **001 00** |
| **5** | **101** | **001 01** |
| **6** | **110** | **001 10** |
| **7** | **111** | **001 11** |
| **8** | **1000** | **0001 000** |
| **9** | **1001** | **0001 001** |
| **10** | **1010** | **0001 010** |

Żeby ulepszyć kod i zmniejszyć jego długość, w części selktora możemy kod unarny zamienić na gamma(γ) kod.

### Elias delta code

Elias-𝛾-kod reprezentuje liczbę całkowitą n jako parę selektora (selector) i przesunięcia (offset):

- Przesunięcie to minimalny kod binarny n bez jego wiodącego 1-bitu

- Selektor to długość jako γ-kod (lub długość przesunięcia+1 jako γ-kod)

Przykłady:

|  |  |  |
| --- | --- | --- |
| n |  |  |
| **1** | **1** | **0** |
| **2** | **10** | **01 0 0** |
| **3** | **11** | **01 0 1** |
| **4** | **100** | **01 1 00** |
| **5** | **101** | **01 1 01** |
| **6** | **110** | **01 1 10** |
| **7** | **111** | **01 1 11** |
| **8** | **1000** | **001 00 000** |
| **9** | **1001** | **001 00 001** |
| **10** | **1010** | **001 00 010** |

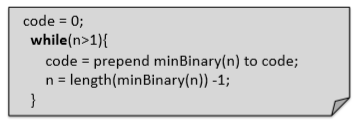
### Elias omega code

Elias-ω-kod reprezentuje liczbę całkowitą n jako parę selektora (selector) i przesunięcia (offset):

- Przesunięcie to minimalny kod binarny n z dodatkowym „0”-bitem na końcu

- Selektor to długość -1 jako δ-kod

Kodowanie:



Przykłady:

|  |  |  |
| --- | --- | --- |
| n |  |  |
| **1** | **1** | **0** |
| **2** | **10** | **10 0** |
| **3** | **11** | **11 0** |
| **4** | **100** | **10 100 0** |
| **5** | **101** | **10 101 0** |
| **6** | **110** | **10 110 0** |
| **7** | **111** | **10 111 0** |
| **8** | **1000** | **11 1000 0** |
| **9** | **1001** | **11 1001 0** |
| **10** | **1010** | **11 1010 0** |

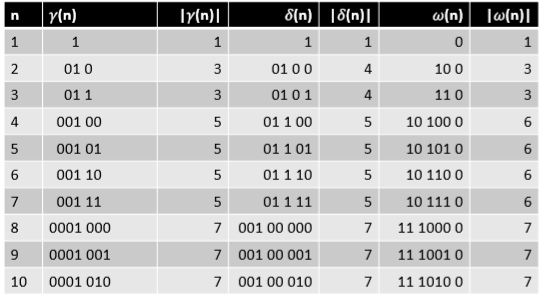


Figure 1. Porównanie kodu gamma, delta i omega

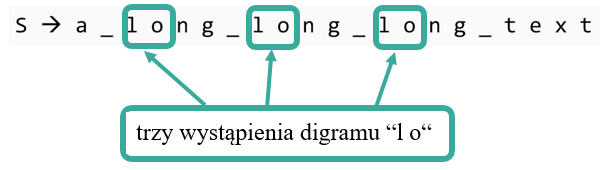
## String RePair

Opracowane przez N. Jespera Larssona i Alistaa Moffa w 1999 r. Pomysł - zastępowanie powtarzających się wzorów przez nieterminalne symbole. Wzory podstawione mogą również zawierać nieterminale. Najpierw należy zamienić najmniejsze wzory w ciągu wejściowym. Należy rozważyć tylko „digramy”, tj. pary 2 sąsiadujących symboli. Dalej także usuwamy nieterminale (reguly), które są nazywane tylko raz (poprzez inlining).

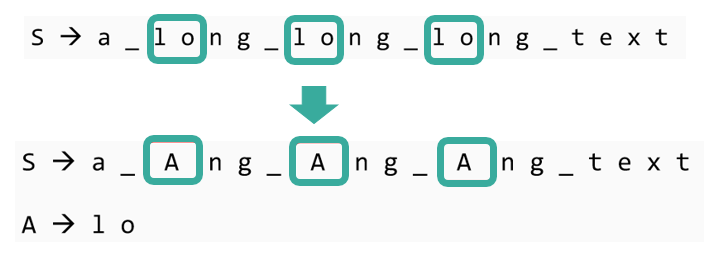
Przykład:



digram = dwa sąsiadujące znaki w ciągu, np. „l o”

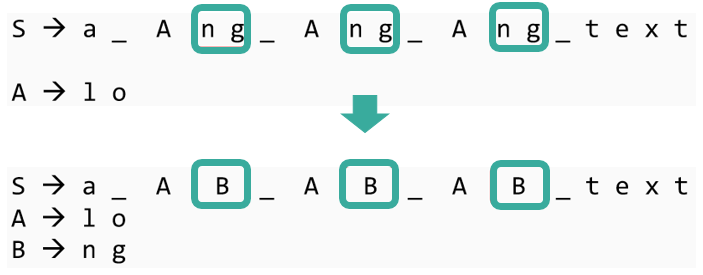


Zamieniamy digram „lo” nieterminałem A. To znaczy, że zastępujemy wszystkie jego wystąpienia w S i generujemy nową regułę gramatyczną:

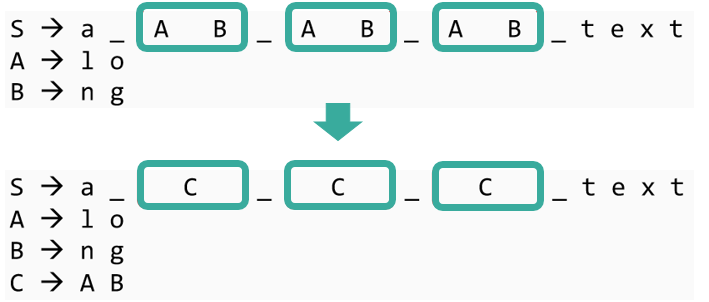


Gramatyczny rozmiar w pierwszym przypadku = 21, w drugim – 18+2.

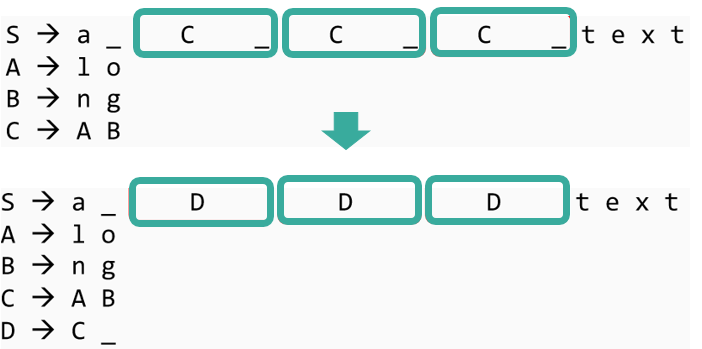
Przetwarzamy dalej:



Gramatyczny rozmiar = 15+2+2



Gramatyczny rozmiar = 12+2+2+2



Gramatyczny rozmiar = 9+2+2+2+2

Zastąpienie digramów nie jest już przydatne, ponieważ nowe terminale i dodatkowe zasady zwiększyłyby rozmiar gramatyki.

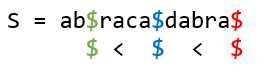
## IRT – aktualizowana wersja BWT

Aktualizujemy BWT, ponieważ dekompresja bezstratna BWT nie spełnia wszystkich wymaganych właściwości kompresji ciągów:

* Brak skutecznego dostępu do n-tego słowa
* Usunięcie nie działa w BWT

BWT nie obsługuje tekstowe bazy danych, drzewa XML, pobieranie dokumentów. To wszystko wymaga innej techniki (np. IRT).

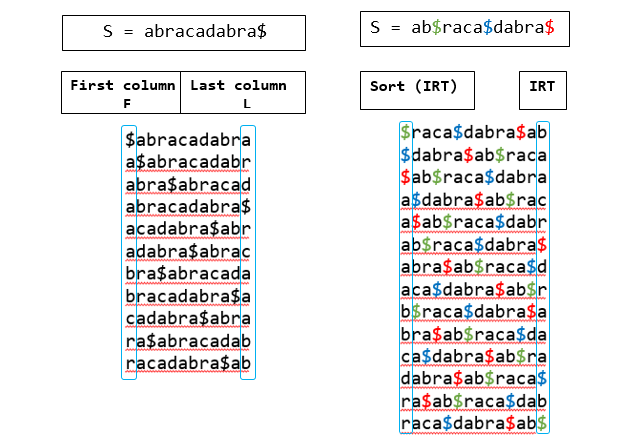
IRT = Indexed Reversible Transformation (Indeksowana Odwracalna Transformacja)



IRT na odmianu od BWT:

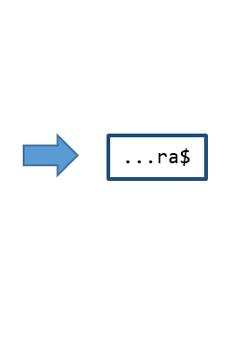
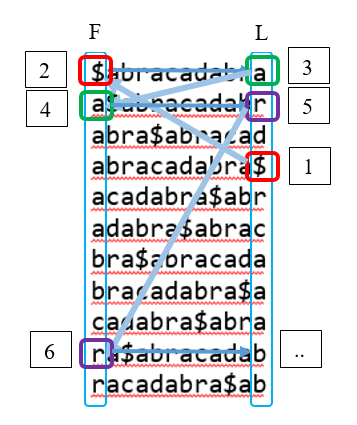
* Symbol $ wśród siebie, określa ich wystąpienie w tekście wejściowym (lexicographic order of $among themselves determined by their occurrence in the input text)
* Obroty w IRT są sortowane według kolejności prefiksów zamiast sufiksów, jak w BWT (rotations in IRT are sorted according to Prefix order instead of Suffix order as in BWT)

Różnica między IRT oraz BWT:

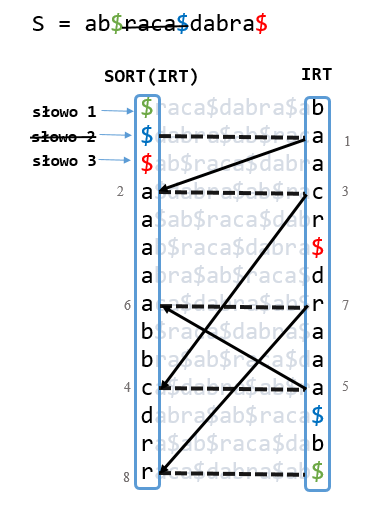


Jak jest opisane w rozdziale BWT, ostatnia kolumna wszystkich przesunięć łańcucha wejściowego w postaci alfabetycznej to jest pierwsza kolumna. Kiedy mamy pierwszą i ostatnią kolumnę, możemy dekompresować ciąg. Zaczynamy od ostatniego symbolu „$” z L-kolumny (ostatniej), nachodzimy w F-kolumnie (pierwszej) właściwy symbol w tej kolejności co i z L-kolumny:

Dalej postępujemy tak, jak pokazano niżej:



Wyjaśniliśmy, że IRT jets lepej niż BWT, bo IRT obsługuje operacje wstawianie i usuwanie. Jeżeli mamy słowo ab$raca$dabra$. Usuwanie, można realizować przy pomocy LF-mapowania. Zaczynamy z symbolu $ i po koleje usuwamy znaki słowa raca$ z L,F-kolumn:



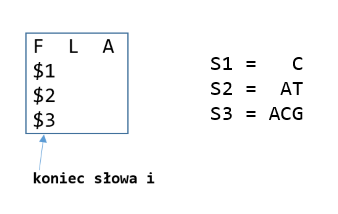
Kolejność słowa i kolejność IRT wszystkich innych znaków pozostają takie same. Wstawienie słowa jest zgodne z tą samą zasadą co i usuwanie.

BWT to kluczowe struktury danych dla wyszukiwania, sekwencjonowania genomu.

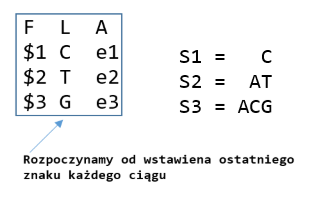
## Kompresja genomu

Równoległa budowa BWT.

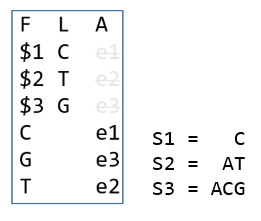
Podany ciąg k, np. k=3. W tym przykładzie ciągi zawierają tylko symbole A,C,G,T (związki chemiczne DNA). F = pierwsza kolumna matrycy BWT, L = ostatnia kolumna matrycy BWT, A = pozycje, w których dodano następny znak ciągu



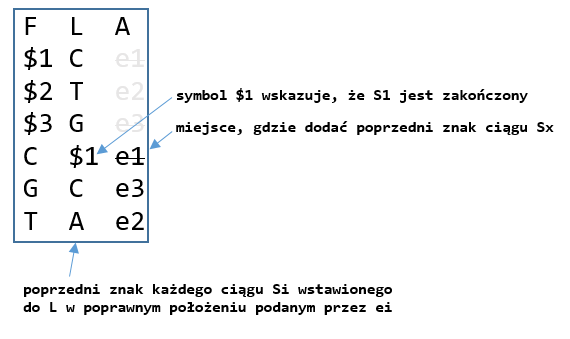
Rozpocząć trzeba od wstawienia ostatniego znaku każdego ciągu symboli.



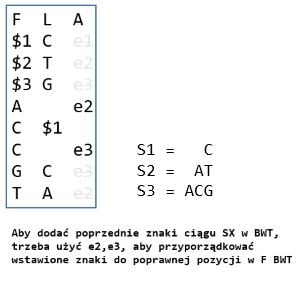
W następnym kroku do kolumny A dodajemy poprzedni znak ciągu. Do kolumny F wstawiamy symbole z kolumny L, ale w alfabetycznym porządku. W kolumnie A zmieniamy porządek żeby dopasować do znaków kolumny F.



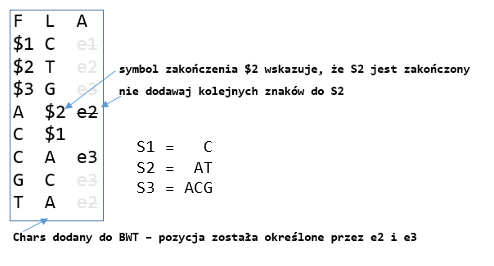
Dalej do kolumny L dodajemy następne znaki ciągów S1, S2, S3. W przypadku S1 wstawiony jest symbol $1, dlatego że S1 ma tylko jedeт znak.



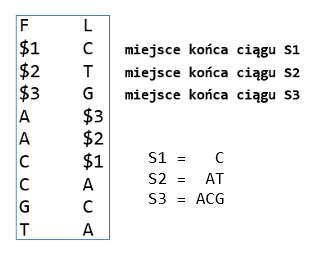
Powtarzamy działania I do kolumny F wstawiamy znaki (następne znaki ciągów S3, S2). Sortujemy kolumnę F. Dopasujemy kolumnę L ta A.



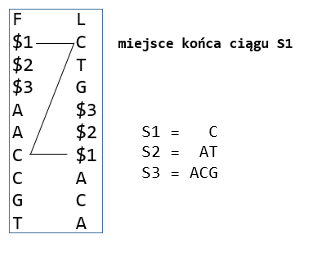
Dodajemy poprzednie znaki ciągów S1, S2, S3. Dla S2 to jest $2 (symbol zakończenia $2 wskazuje, że S2 jest zakończony), dla S3 – A. Dodajemy symbole do kolumny Last.



Sortujemy kolumnę F. Dopasujemy kolumnę L ta A. Wynik:



Dekompresja S1 przez mapowanie LF:



Przetwarzanie danych genomu. Powodem do tego jest to, że jest uważane za klucz do badań medycznych, farmacji, biologii. Sekwencjonowanie genomu — surowa sekwencja nukleotydowa DNA pojedynczego organizmu. Biologiczne i medyczne znaczenie i wpływ tych sekwencji - w jaki sposób można zastosować sekwencjonowanie genomu w celu zapobiegania chorobie?  
Badania medyczne wymagają algorytmów szybkiego wyszukiwania sekwencji genomu.

Aby znaleźć mutacje lub różnice, potrzebujemy dość dużej kolekcji ludzkich genomów, np. tysięcy genomów.

Spodziewamy się, że genomy mają bardzo podobne wzorce, tzn. niewiele różnic.

* FM-Index (=BWT-based Self-index): Wyszukiwanie może być korzystne z podobieństw (alternatywne podejście: Matryca z pełnym sufiksem)

Wzory wyszukiwania:

1. Wyszukiwanie substringów ACGT (gdzie w ciągu odniesienia występuje podciąg)

2. Wyrównanie ciągów (jak można wyrównać wiele warstw w dużym ciągu)

3. Wyszukiwanie lub wyrównanie nie pasujące

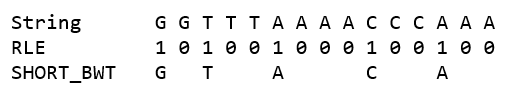
Typowe podejścia:

1. Pełna tablica sufiksów ciągów ACGT

2. BWT tablicy sufiksów ACGT smyczkowych à bardzo długie biegi.

Wymagane: Używane miejsce powinno mieć wartość o(#runs), a nie o( size(BWT) )

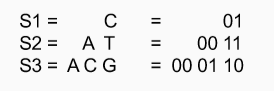
Przykad:



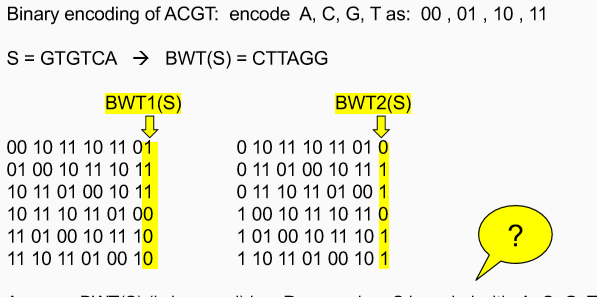
Jak zrobić lepiej? Możemy wykorzystać podejście: ilość powtórzeń znaku (int) + znak. Ale podobne podejście ma 2 różne typy danych.

Teraz ja będę wykorzystywać binarne BWT dla danych genomu. Czyli zamiast zwykłego BWT-kodowania ACGT-stringu jako ciągu, robimy ciąg binarny i generujemy 2 binarne BWT-tablicy w następujący sposób: obroty zaczynające się z pierwszego bitu i obroty zaczynające się z drugiego bitu. Generujemy dwie tablicy: BWT1i BWT2. Będziemy także wykorzystywać mapowanie LF.

Możemy przedstawić genom jak: A=00, C=01, G=10, T=11.



Do konstuwania wykorzystujemy IRT.



## Mierzenie czasu

Żeby wymierzyć czas wykonania realizacji algorytmów RePair, Huffman, BWT, musicie wymierzyłam czas CPU, czyli czas, zużyty CPU na obróbkę instrukcji realizacji. W python można mierzyć czas przy pomocy funkcji time.process\_time(), clock(), perf\_counter().Ja mierzę czas przy pomocy metody evaluate() ta funkcji wbudoanej perf\_counter\_ns.

Każde wywołanie mojej implementacji jest otoczone pomiarem początkowego czasu procesora startTime i końcowego czasu procesora stopTime. Powtarzaj te wywołania 20 raz. (aż do osiągnięcia całkowitego czasu działania procesora wynoszącego 30 sekund) (tj. Suma endTime-startTime dla wszystkich przebiegów jest większa lub równa 30 s) i zwiększam rozmiar losowego pliku o 500 znaków dla każdej iteracji (tj. zwiększyć parametr wejściowy i o 5).

W ramach każdej iteracji wprowadziłam parametr i, czas procesora, który upłynął dla bieżącej iteracji oraz sumę wszystkich czasów procesora, które upłynęły we wszystkich iteracjach. Porównaj wartości ostatnich iteracji dla wszystkich 3 implementacji.

|  |  |  |
| --- | --- | --- |
| bwt | huffman | repair |
| 263 600  552 000  700 700  1 059 600  1 666 000  1 989 700  2654700  2985300  3857800  4745800  5105400  5577100  7459000  8731900  8605600  8691000  10541500  11112800  11 970 900 | 2 955 900  3 434 100  3 056 600  3 586 200  3 940 400  3 511 800  4579200  5511000  3950400  7219100  3040300  4370300  3573200  4289100  5974800  3102600  5509900  4307300  3 500 500 | 1 383 200  4077600  8103700  13094900  16011300  19669300  24320000  31505400  48445200  60012600  55237200  70138300  66908500  81100200  100296100  93863300  116941400  102821600  115 793 800 |

# Bibliografia

<https://mafiadoc.com/offline-dictionary-based-compression-jesper-larsson_59d665411723dd7e65450aa7.html>