# Kompresja tekstu

dr inż. Marcin Ciura
mgc@agh.edu.pl

Wydział Informatyki, Akademia Górniczo-Hutnicza

## Plan wykładu

- Zwycięzcy wyzwań
- Definicje
- Kodowanie Shannona-Fano
- Kodowanie Huffmana
- Kodowanie arytmetyczne
- Kodowanie statyczne i dynamiczne
- Kodowanie słownikowe
- Transformata Burrowsa-Wheelera

# Zwycięzcy wyzwań

## Zwycięzcy wyzwań

# Definicje

# Definicje

Kompresja tekstu to przekształcenie naturalnej reprezentacji tekstu w inną reprezentację, która zajmuje mniej bitów

Każda metoda kompresji musi mieć zdefiniowaną fazę dekompresji, w której rekonstruuje się naturalną reprezentację tekstu

# Definicje

Kod binarny to taki kod, w którym każdemu symbolowi odpowiada pewien ustalony ciąg bitów, zwany słowem kodowym

Przykład kodu binarnego: $\{\mathtt{m} \to \mathtt{01}, \mathtt{a} \to \mathtt{1}\}$

# Definicje

Kod prefiksowy to taki kod, w którym żadne słowo kodowe nie jest prefiksem innego słowa kodowego

Przykład kodu prefiksowego: $\{\texttt{t} \to \texttt{01}, \texttt{a} \to \texttt{11}\}$

Kodowanie w kodzie prefiksowym polega na łączeniu ze sobą słów kodowych

Dekodowanie kodu prefiksowego jest jednoznaczne

# Definicje

Kod o zmiennej długości to taki kod, w którym słowa kodowe mają różną długość :-)

Gdy częstszym symbolom odpowiadają krótsze słowa kodowe, a rzadszym symbolom – dłuższe słowa kodowe, to przy użyciu kodu o zmiennej długości można lepiej kompresować tekst niż przy użyciu kodu o stałej długości

# Przykład kodu binarnego o zmiennej długości

| Symbol | a | n | s |
|---|---|---|---|
| Słowo kodowe o stałej długości | 00 | 01 | 10 |
| Słowo kodowe o zmiennej długości | 0 | 11 | 10 |

Kodujemy łańcuch ananas

Kod ASCII: $6 \times 8 = 48$ bitów

Kod o stałej długości: `00·01·00·01·00·10` 12 bitów

Kod o zmiennej długości: `0·11·0·11·0·10` 9 bitów

# Claude Shannon (30.4.1916–24.2.2001)

Amerykański matematyk, elektrotechnik, informatyk i kryptograf. Profesor MIT. W 1948 roku opublikował artykuł *The Mathematical Theory of Communication*, w którym zajął się kodowaniem komunikatów. Oprócz tego zbudował maszynę żonglującą, rakietowe frisbee i urządzenie układające kostkę Rubika.

Entropia to średnia ilość informacji, która przypada na 1 symbol ze źródła informacji

Jednostką entropii jest bit

Entropia $H(X)$ zmiennej losowej $X$, która przyjmuje wartości $\{x_1, x_2, \ldots x_n\}$ z prawdopodobieństwami $p(x_1), p(x_2), \ldots p(x_n)$ wynosi

$$H(X) = -\sum_{i=1}^{n} p(x_i) \log_2 p(x_i)$$

## Definicje

Przykład: Dana jest zmienna losowa $X$, która przyjmuje trzy wartości: a, n i s z prawdopodobieństwami

$p(\mathrm{a}) = 1/2$
$p(\mathrm{n}) = 1/3$
$p(\mathrm{s}) = 1/6$

$$H(X) = -(1/2 \log_2(1/2) + 1/3 \log_2(1/3) + 1/6 \log_2(1/6)) \approx$$

$$-(1/2 \cdot (-1) + 1/3 \cdot (-1{,}585) + 1/6 \cdot (-2{,}585)) \approx$$

$$0{,}5 + 0{,}5283 + 0{,}4308 =$$

$$1{,}4591$$

# Definicje

Podstawowe twierdzenie Shannona: średnia długość jednoznacznie dekodowalnego kodu nie może być mniejsza niż entropia rozkładu prawdopodobieństwa występowania słów kodowych

Przykład: średnia długość kodu $\{\mathtt{a} : \mathtt{0}, \mathtt{n} : \mathtt{11}, \mathtt{s} : \mathtt{10}\}$, gdy $p(\mathtt{a}) = 1/2$, $p(\mathtt{n}) = 1/3$, $p(\mathtt{s}) = 1/6$, wynosi

$$1/2 \cdot 1 + 1/3 \cdot 2 + 1/6 \cdot 2 = 1{,}5$$

czyli więcej niż entropia

$$H(X) = 1{,}4591$$

# Kodowanie Shannona-Fano

# Robert Fano (11.11.1917–13.7.2016)

Włosko-amerykański elektrotechnik i informatyk. Profesor MIT. W 1948 roku wraz z Claude'em Shannonem odkrył, jak zbudować kod prefiksowy na podstawie prawdopodobieństwa występowania symboli.

# Kodowanie Shannona-Fano

Symbole alfabetu mają przypisane wagi – zwykle częstość lub prawdopodobieństwo wystąpienia. Tworzymy kod metodą zstępującą:

- Posortuj symbole według rosnących wag
    1. Jeśli lista symboli jest pusta lub zawiera 1 symbol, nie wykonuj kroków 2–4
    2. Podziel listę symboli na dwie części w takim miejscu, by suma wag lewej strony i suma wag prawej strony były jak najbliższe
    3. Dopisz 0 na końcu kodów tych symboli, które leżą po lewej stronie listy; dopisz 1 na końcu kodów tych symboli, które leżą po prawej stronie listy
    4. Wykonaj kroki 1–4 dla lewej strony listy; wykonaj kroki 1–4 dla prawej strony listy

# Kodowanie Shannona-Fano – przykład

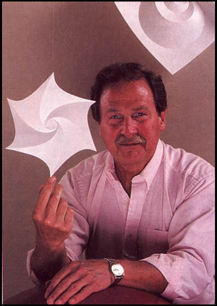
Kodowany łańcuch: `madagaskar`

| Symbol | m | d | g | s | k | r | a |
|---|---|---|---|---|---|---|---|
| Waga | 1 | 1 | 1 | 1 | 1 | 1 | 4 |
| 1. podział | 0 | 0 | 0 | 0 | 0 | 1 | 1 |
| 2. podziały | 00 | 00 | 01 | 01 | 01 | 10 | 11 |
| 3. podziały | 000 | 001 | 010 | 011 | 011 | | |
| 4. podział | | | | 0110 | 0111 | | |

`000·11·001·11·010·11·0110·0111·11·10` 27 bitów

# Kodowanie Huffmana

# David A. Huffman (9.8.1925–7.10.1999)



Amerykański informatyk. W 1951 roku, kiedy był doktorantem na MIT, Robert Fano, wykładowca teorii informacji, dał studentom wybór: pisać egzamin albo opracować wydajniejsze kodowanie niż kodowanie Shannona-Fano. Huffman opracował optymalny kod prefiksowy. Potem zajmował się między innymi teorią origami.

# Kodowanie Huffmana

Tworzymy kod jako drzewo binarne metodą wstępującą. Każdy liść drzewa odpowiada jednemu symbolowi alfabetu. Każdy węzeł ma wagę równą sumie wag tego poddrzewa, którego jest korzeniem

- Utwórz po jednym węźle dla każdego symbolu alfabetu
- Dodaj wszystkie węzły do kolejki priorytetowej
- Dopóki w kolejce jest co najmniej 1 węzeł, powtarzaj kroki 1–3:
    1. Usuń z kolejki dwa węzły o najniższych wagach
    2. Utwórz nowy węzeł, którego dziećmi są te dwa węzły, a wagą jest suma wag tych dwóch węzłów
    3. Dodaj ten nowy węzeł do kolejki
- Pozostały w kolejce 1 węzeł to korzeń drzewa

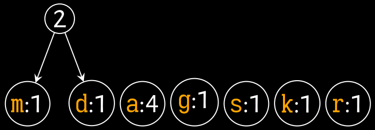
# Kodowanie Huffmana – przykład

Kodowany łańcuch: `madagaskar`

`m`:1 `a`:4 `d`:1 `g`:1 `s`:1 `k`:1 `r`:1

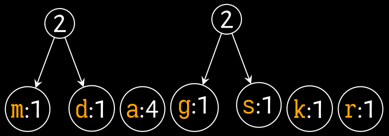
# Kodowanie Huffmana – przykład

Kodowany łańcuch: `madagaskar`



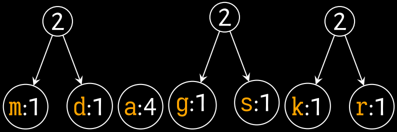
# Kodowanie Huffmana – przykład

Kodowany łańcuch: `madagaskar`



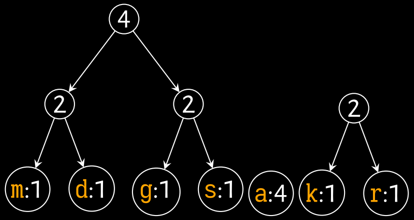
# Kodowanie Huffmana – przykład

Kodowany łańcuch: `madagaskar`



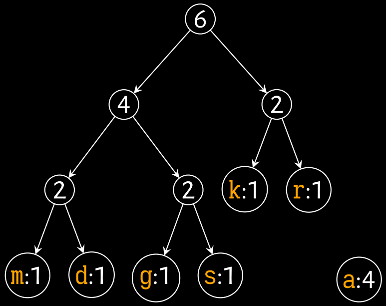
# Kodowanie Huffmana – przykład

Kodowany łańcuch: `madagaskar`



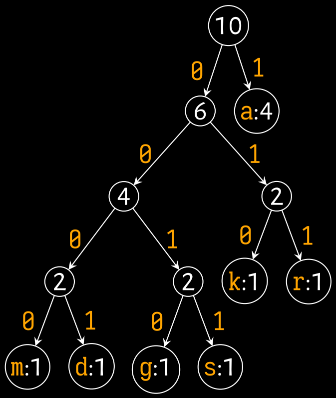
## Kodowanie Huffmana – przykład

Kodowany łańcuch: `madagaskar`



# Kodowanie Huffmana – przykład

Kodowany łańcuch: `madagaskar`




`0000·1·0001·1·0010·1·0011·010·1·011` 26 bitów

# Kodowanie Huffmana

Kodowanie Huffmana to optymalna metoda kodowania spośród tych metod, które kodują każdy symbol osobno, zawsze jako to samo słowo kodowe

# Kodowanie arytmetyczne

# Jorma Rissanen (20.10.1932–9.5.2020)

Fiński informatyk. W 1975 roku, kiedy pracował w IBM Research w San Jose, opublikował algorytm kompresji, który nie korzysta ze słów kodowych.

# Kodowanie arytmetyczne

- Odwzoruj wszystkie symbole alfabetu ($s_i$) w rozłączne podprzedziały $P(s_i)$ przedziału [0; 1), których długości są proporcjonalne do wag tych symboli
- $Q := [0; 1)$
- Powtarzaj kroki 1–2 dopóki w kodowanym tekście są symbole:
    1. Pobierz kolejny symbol kodowanego tekstu do zmiennej $s$
    2. $Q := [\inf Q + |Q| \inf P(s); \inf Q + |Q| \sup P(s))$
- Wyślij dowolną liczbę z przedziału Q jako ułamek dwójkowy

# Dekodowanie arytmetyczne – idea algorytmu

- Odwzoruj wszystkie symbole alfabetu ($s_i$) w rozłączne podprzedziały $P(s_i)$ przedziału [0; 1), których długości są proporcjonalne do wag tych symboli
- Pobierz liczbę $q$
- Powtarzaj kroki 1–2 dopóki nie zdekodujesz wszystkich symboli:
  1. Wyślij taki symbol $s$, dla którego $q \in P(s)$
  2. $q := (q - \inf P(s))/|P(s)|$

# Kodowanie arytmetyczne – przykład

`m` [0,0; 0,1)

`a` [0,1; 0,5)

`d` [0,5; 0,6)

`g` [0,6; 0,7)

`s` [0,7; 0,8)

`k` [0,8; 0,9)

`r` [0,9; 1,0)

$P$ [0; 1)

`m` [0; 0,1)

# Kodowanie arytmetyczne – przykład

`m` [0,0; 0,1)

`a` [0,1; 0,5)

`d` [0,5; 0,6)

`g` [0,6; 0,7)

`s` [0,7; 0,8)

`k` [0,8; 0,9)

`r` [0,9; 1,0)

`m` [0,0; 0,1)

`ma` [0,01; 0,05)

# Kodowanie arytmetyczne – przykład

`m` [0,0; 0,1)

`a` [0,1; 0,5)

`d` [0,5; 0,6)

`g` [0,6; 0,7)

`s` [0,7; 0,8)

`k` [0,8; 0,9)

`r` [0,9; 1,0)

`ma` [0,01; 0,05)

`mad` [0,03; 0,034)

## Kodowanie arytmetyczne – przykład

m [0,0; 0,1)

a [0,1; 0,5)

d [0,5; 0,6)

g [0,6; 0,7)

s [0,7; 0,8)

k [0,8; 0,9)

r [0,9; 1,0)

mad [0,03; 0,034)

mada [0,0304; 0,032)

# Kodowanie arytmetyczne – przykład

`m` [0,0; 0,1)

`a` [0,1; 0,5)

`d` [0,5; 0,6)

`g` [0,6; 0,7)

`s` [0,7; 0,8)

`k` [0,8; 0,9)

`r` [0,9; 1,0)

`mada` [0,0304; 0,032)

`madag` [0,03136; 0,03152)

# Kodowanie arytmetyczne – przykład

`m` [0,0; 0,1)

`a` [0,1; 0,5)

`d` [0,5; 0,6)

`g` [0,6; 0,7)

`s` [0,7; 0,8)

`k` [0,8; 0,9)

`r` [0,9; 1,0)

`madag` [0,03136; 0,03152)

`madaga` [0,031376; 0,03144)

## Kodowanie arytmetyczne – przykład

m [0,0; 0,1)

a [0,1; 0,5)

d [0,5; 0,6)

g [0,6; 0,7)

s [0,7; 0,8)

k [0,8; 0,9)

r [0,9; 1,0)

madaga [0,031376; 0,03144)

madagas [0,0314208; 0,0314272)

# Kodowanie arytmetyczne – przykład

`m` [0,0; 0,1)

`a` [0,1; 0,5)

`d` [0,5; 0,6)

`g` [0,6; 0,7)

`s` [0,7; 0,8)

`k` [0,8; 0,9)

`r` [0,9; 1,0)

`madagas` [0,0314208; 0,0314272)

`madagask` [0,03142592; 0,03142656)

# Kodowanie arytmetyczne – przykład

`m` [0,0; 0,1)

`a` [0,1; 0,5)

`d` [0,5; 0,6)

`g` [0,6; 0,7)

`s` [0,7; 0,8)

`k` [0,8; 0,9)

`r` [0,9; 1,0)

`madagask` [0,03142592; 0,03142656)

`madagaska` [0,031425984; 0,03142624)

# Kodowanie arytmetyczne – przykład

`m` [0,0; 0,1)

`a` [0,1; 0,5)

`d` [0,5; 0,6)

`g` [0,6; 0,7)

`s` [0,7; 0,8)

`k` [0,8; 0,9)

`r` [0,9; 1,0)

`madagaska` [0,031425984; 0,03142624)

`madagaskar` [0,0314262144; 0,03142624)

# Kodowanie arytmetyczne – przykład

`madagaskar` [0,0314262144; 0,03142624) =

$[0{,}000010000000101110001100 01\ldots_2;\ 0{,}00001000000010111000110011\ldots_2)$

Wystarczy wysłać kod dwójkowy dowolnej liczby z tego przedziału, na przykład

`00001·00000·00101·11000·11001` 25 bitów

# Dekodowanie kodu arytmetycznego – przykład

`m` [0,0; 0,1)

`a` [0,1; 0,5)

`d` [0,5; 0,6)

`g` [0,6; 0,7)

`s` [0,7; 0,8)

`k` [0,8; 0,9)

`r` [0,9; 1,0)

$q = 0{,}03142623$

`m`

$q := (0{,}03142623 - 0)/0{,}1 = 0{,}3142623$

# Dekodowanie kodu arytmetycznego – przykład

`m` [0,0; 0,1)

`a` [0,1; 0,5)

`d` [0,5; 0,6)

`g` [0,6; 0,7)

`s` [0,7; 0,8)

`k` [0,8; 0,9)

`r` [0,9; 1,0)

$q = 0{,}3142623$

`ma`

$q := (0{,}3142623 - 0{,}1)/0{,}4 = 0{,}5356558$

# Dekodowanie kodu arytmetycznego – przykład

`m` [0,0; 0,1)

`a` [0,1; 0,5)

`d` [0,5; 0,6)

`g` [0,6; 0,7)

`s` [0,7; 0,8)

`k` [0,8; 0,9)

`r` [0,9; 1,0)

$q = 0{,}5356558$

`mad`

$q := (0{,}5356558 - 0{,}5)/0{,}1 = 0{,}356558$

# Dekodowanie kodu arytmetycznego – przykład

`m` [0,0; 0,1)

`a` [0,1; 0,5)

`d` [0,5; 0,6)

`g` [0,6; 0,7)

`s` [0,7; 0,8)

`k` [0,8; 0,9)

`r` [0,9; 1,0)

$q = 0{,}356558$

`mada`

$q := (0{,}356558 - 0{,}1)/0{,}4 = 0{,}64139$

# Dekodowanie kodu arytmetycznego – przykład

`m` [0,0; 0,1)

`a` [0,1; 0,5)

`d` [0,5; 0,6)

`g` [0,6; 0,7)

`s` [0,7; 0,8)

`k` [0,8; 0,9)

`r` [0,9; 1,0)

$q = 0{,}64139$

`madag`

$q := (0{,}64139 - 0{,}6)/0{,}1 = 0{,}4139$

# Dekodowanie kodu arytmetycznego – przykład

`m` [0,0; 0,1)

`a` [0,1; 0,5)

`d` [0,5; 0,6)

`g` [0,6; 0,7)

`s` [0,7; 0,8)

`k` [0,8; 0,9)

`r` [0,9; 1,0)

$q = 0{,}4139$

`madaga`

$q := (0{,}4139 - 0{,}1)/0{,}4 = 0{,}7848$

# Dekodowanie kodu arytmetycznego – przykład

`m` [0,0; 0,1)

`a` [0,1; 0,5)

`d` [0,5; 0,6)

`g` [0,6; 0,7)

`s` [0,7; 0,8)

`k` [0,8; 0,9)

`r` [0,9; 1,0)

$q = 0{,}7848$

`madagas`

$q := (0{,}7848 - 0{,}7)/0{,}1 = 0{,}848$

# Dekodowanie kodu arytmetycznego – przykład

`m` [0,0; 0,1)

`a` [0,1; 0,5)

`d` [0,5; 0,6)

`g` [0,6; 0,7)

`s` [0,7; 0,8)

`k` [0,8; 0,9)

`r` [0,9; 1,0)

$q = 0{,}848$

`madagask`

$q := (0{,}848 - 0{,}8)/0{,}1 = 0{,}48$

# Dekodowanie kodu arytmetycznego – przykład

`m` [0,0; 0,1)

`a` [0,1; 0,5)

`d` [0,5; 0,6)

`g` [0,6; 0,7)

`s` [0,7; 0,8)

`k` [0,8; 0,9)

`r` [0,9; 1,0)

$q = 0{,}48$

`madagaska`

$q := (0{,}48 - 0{,}1)/0{,}4 = 0{,}95$

# Dekodowanie kodu arytmetycznego – przykład

`m` [0,0; 0,1)

`a` [0,1; 0,5)

`d` [0,5; 0,6)

`g` [0,6; 0,7)

`s` [0,7; 0,8)

`k` [0,8; 0,9)

`r` [0,9; 1,0)

$q = 0{,}95$

`madagaskar`

# Kodowanie arytmetyczne

W kodzie arytmetycznym nie ma słów kodowych, więc na 1 symbol może przypadać niecałkowita liczba bitów kodu, nawet mniejsza niż 1 bit

Zatem kod arytmetyczny zajmuje mniej bitów niż kod Huffmana, zwłaszcza gdy częstość występowania jakiegoś symbolu jest bliska 1, bo wtedy:

- Długość słowa kodowego w kodzie Huffmana nie może być mniejsza niż 1 bit
- Średnia długość kodu Huffmana jest wyraźnie większa od entropii rozkładu prawdopodobieństwa symboli

# Kodowanie arytmetyczne

Kod arytmetyczny można kodować i dekodować, korzystając tylko z arytmetyki liczb całkowitych, traktowanych jako ułamki, czyli liczby stałoprzecinkowe

Implementacje takiego algorytmu działają wolniej niż kodowanie Huffmana, chyba że użyje się asymetrycznych systemów liczbowych, opracowanych przez Jarosława Dudę z UJ :-)

# Kodowanie statyczne i dynamiczne

# Kodowanie statyczne

Wprowadziłem państwa w błąd: nikt nie kompresuje tak, jak to opisałem, czyli statycznie :-)

Żeby po statycznej kompresji działała dekompresja, trzeba by przesyłać wraz ze skompresowanym komunikatem:

- Listę słów kodowych lub listę częstości występowania symboli
- Długość komunikatu lub dodatkowy kod końca komunikatu

Do tego kompresja musiałaby być dwuprzebiegowa:

- W pierwszym przebiegu sumowałaby częstość występowania symboli
- W drugim przebiegu kodowałaby symbole komunikatu

# Kodowanie dynamiczne, czyli adaptacyjne

Przy kodowaniu dynamicznym koder i dekoder synchronizują swój stan, czyli model prawdopodobieństwa występowania symboli

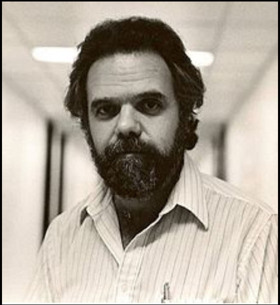
## Kodowanie dynamiczne – schemat algorytmu kodera

- Inicjalizuj model
- Dopóki jest co kodować, wykonuj kroki 1–2:
    1. Zakoduj kolejny symbol
    2. Zaktualizuj model na podstawie tego symbolu

# Kodowanie dynamiczne – schemat algorytmu dekodera

- Inicjalizuj model
- Dopóki jest co dekodować, wykonuj kroki 1–2:
    1. Zdekoduj kolejny symbol
    2. Zaktualizuj model na podstawie tego symbolu

# Newton Faller (25.1.1947–9.10.1996)



Brazylijski elektrotechnik i informatyk. W 1973 roku, jako pracownik IBM w Rio de Janeiro opublikował algorytm dynamicznego kodowania Huffmana.

# Robert Gray Gallager (29.5.1931–)

Amerykański elektrotechnik z dziedziny teorii informacji. W 1978 roku opublikował poprawki do algorytmu Fallera.

# Donald Knuth (10.1.1938–)

W 1985 roku opublikował poprawki do algorytmu Fallera-Gallagera.

## Dynamiczne kodowanie Huffmana – algorytm FGK

Pomocnicza struktura danych: dwukierunkowa lista węzłów, zawierająca węzły poziomami od korzenia do liści, a na każdym poziomie od prawej do lewej. Każdy węzeł zawiera swój licznik. Lista węzłów jest posortowana nierosnąco względem liczników

# Dynamiczne kodowanie Huffmana – szkic algorytmu FGK

- Zacznij od drzewa z 1 węzłem (\), który reprezentuje takie symbole, które jeszcze nie wystąpiły w kodowanym tekście
- Jeśli w tekście wystąpi nowy symbol, prześlij kod \, a po nim prześlij niezakodowany symbol. Potem zastąp węzeł \ nowym węzłem, będącym rodzicem węzła \ i węzła nowego symbolu
- Inkrementuj liczniki w węzłach na ścieżce do korzenia. Jeśli licznik któregoś węzła na ścieżce do korzenia jest większy niż licznik jego poprzednika na liście węzłów, zamień odpowiednie dwa węzły miejscami tak, żeby lista węzłów znowu była posortowana

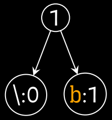
# Dynamiczne kodowanie Huffmana – algorytm FGK


węzły poziomami: 0

b

001100010

# Dynamiczne kodowanie Huffmana – algorytm FGK

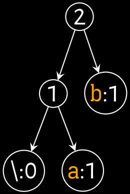



węzły poziomami: 1 1 0

ba

001100010·0·001100001

# Dynamiczne kodowanie Huffmana – algorytm FGK

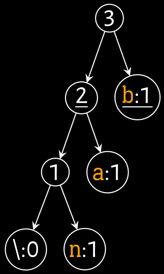



węzły poziomami: 2 1 1 1 0

ban

001100010·0·001100001·00·001101110

# Dynamiczne kodowanie Huffmana – algorytm FGK

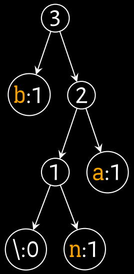



węzły poziomami: 3 <u>1</u> <u>2</u> 1 1 1 0

ban

`001100010·0·001100001·00·001101110`

# Dynamiczne kodowanie Huffmana – algorytm FGK

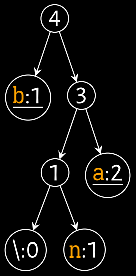



węzły poziomami: 3 2 1 1 1 1 0

bana

`001100010·0·001100001·00·001101110·11`

# Dynamiczne kodowanie Huffmana – algorytm FGK

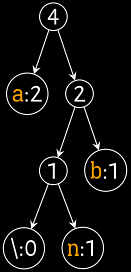



węzły poziomami: 4 3 1 2 1 1 0

bana

001100010·0·001100001·00·001101110·11

# Dynamiczne kodowanie Huffmana – algorytm FGK

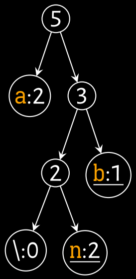



węzły poziomami: 4 2 2 1 1 1 0

banan

`001100010·0·001100001·00·001101110·11·101`

# Dynamiczne kodowanie Huffmana – algorytm FGK

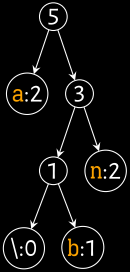



węzły poziomami: 5 3 2 1 2 2 0

banan

`001100010·0·001100001·00·001101110·11·101`

# Dynamiczne kodowanie Huffmana – algorytm FGK



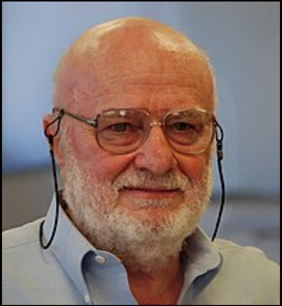

węzły poziomami: 5 3 2 2 2 1 0

banan<EOF>

`001100010·0·001100001·00·001101110·11·101·100·1` 39 bitów

# Kodowanie słownikowe

# Abraham Lempel (10.2.1936–4.2.2023)



Izraelski informatyk. Urodził się we Lwowie. W latach 1977 i 1978 wraz z Jacobem Zivem opublikował algorytmy LZ77 i LZ78, stanowiące podstawę współczesnych metod bezstratnej kompresji tekstu i obrazów.

# Jacob Ziv (27.11.1931–25.3.2023)

Izraelski elektrotechnik i informatyk. W latach 1977 i 1978 wraz z Abrahamem Lempelem opublikował algorytmy LZ77 i LZ78.

# Kodowanie słownikowe

- Kodowanie słownikowe zastępuje taki ciąg symboli, który wcześniej wystąpił w kodowanym tekście, przez pozycję jego poprzedniego wystąpienia i długość tego ciągu
- Do sprawdzania, czy ciąg symboli wystąpił w tekście, służy słownik

# Algorytm LZ77

- Tekst przepływa przez bufor `buf[N]byte`
- Większa część bufora zawiera tę część tekstu, która już została zakodowana: `prev = buf[:N-F]`
- Ostatnie pozycje bufora zawierają te symbole, które jeszcze nie są zakodowane: `ahead = buf[N-F:]`
- Praktyczne wartości: `N = 8192`, `F = 20`
- `Słownik`: wszystkie słowa w buforze `buf`, które zaczynają się w części `prev`

# Algorytm LZ77

- Inicjalizuj pusty bufor; inicjalizuj pusty słownik
- Wczytaj co najwyżej `F` symboli do części `ahead`
- Powtarzaj kroki 1–3, dopóki w części `ahead` są symbole:
  1. Znajdź największe takie `i`, że `slices.Equal(ahead[:i], buf[N-F-j:N-F-j+i])`
  2. Jeśli kod pary <`i, j`> jest krótszy niż `i` kodów poszczególnych symboli `ahead[0:i]`, to wyślij go, przesuń bufor w lewo o `i` symboli i wczytaj co najwyżej `i` symboli do części `ahead`
  3. W przeciwnym przypadku wyślij kod symbolu `ahead[0]`, przesuń bufor w lewo o 1 symbol i wczytaj co najwyżej 1 symbol do części `ahead`

# Algorytm LZ77 – przykład

`To␣niedzwiedz␣czy␣moze␣dzwiedz?␣Chyba␣nie␣dzwiedz.`

`To␣niedzw`<4,5>`␣czy␣moze␣`<6,17>`?␣Chyba`<4,34><8,19>`.`

Sposób kodowania symboli i par zależy od implementacji

# Algorytm LZ77 – przykład

ananas

an<3,2>s

Pierwszy element pary może być większy od drugiego, gdy kodowany tekst zawiera podłańcuchy okresowe

# Algorytm LZ77 – słownik

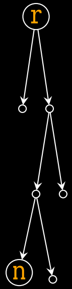
- Drzewo binarne przechowywane w tablicy `N-F` węzłów, po jednym węźle na każdy element części `prev`
- `k`-ty węzeł reprezentuje podłańcuch `buf[k:k+F]`
- Atrybuty węzła: `left`, `right`, `parent`
- Aktualizacja słownika po zakodowaniu `i` symboli:
  - usuń pierwsze `i` węzłów
  - przesuń bufor i słownik w lewo o `i` pozycji
  - dodaj do słownika wierzchołki o indeksach `N-F-i:N-F`
  - wczytaj co najwyżej `i` symboli do części `ahead`
- W praktyce tego drzewa binarnego nie trzeba równoważyć

# Algorytm LZ77 – słownik

Żeby znaleźć najdłuższy podłańcuch zawarty w buforze:

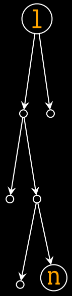
- Szukaj w drzewie binarnym łańcucha `ahead[0:F]`; pamiętaj ostatni węzeł `l`, z którego wybrano gałąź `left` i ostatni węzeł `r`, z którego wybrano gałąź `right`
- Jeśli znaleziono łańcuch `ahead[0:F]` w węźle `n`, to zwróć parę <`F`, `n`>

- Jeśli nie znaleziono łańcucha `ahead[0:F]`, ostatni przeszukany węzeł ma numer `n` i `slices.Compare(ahead[0:F], buf[n:n+F]) < 0`, to najdłuższy prefiks łańcucha `ahead[0:F]` leży leksykograficznie między węzłami `n` i `r`. Między tymi węzłami nie ma więcej węzłów, więc prefiks zaczyna się albo w węźle `n`, albo w węźle `r`. Sprawdź obie możliwości, zwróć długość dłuższego prefiksu i pozycję jego początku



# Algorytm LZ77 – słownik

- Jeśli nie znaleziono łańcucha `ahead[0:F]`, ostatni przeszukany węzeł ma numer `n` i `slices.Compare(ahead[0:F], buf[n:n+F]) > 0`, to najdłuższy prefiks łańcucha `ahead[0:F]` leży leksykograficznie między węzłami `l` i `n`. Między tymi węzłami nie ma więcej węzłów, więc prefiks zaczyna się albo w węźle `l`, albo w węźle `n`. Sprawdź obie możliwości, zwróć długość dłuższego prefiksu i pozycję jego początku



# Algorytm LZ77 – słownik

- Albo jeszcze prościej: szukaj w drzewie binarnym łańcucha `ahead[0:F]`; pamiętaj najdłuższy prefiks tego łańcucha znaleziony na ścieżce wyszukiwania

## Kodowanie słownikowe – zastosowania

LZ77 + Huffman = DEFLATE: zip, gzip, PNG, PDF,...

LZ77 + Huffman + predefiniowany słownik = Brotli

# Transformata Burrowsa-Wheelera

# Michael Burrows (1963–)

Brytyjski informatyk. W 1994 roku wraz ze swoim promotorem Davidem Wheelerem opublikował tak zwaną transformatę Burrowsa-Wheelera, stosowaną między innymi w kompresorze bzip2. Jako pracownik Google opracował Chubby'ego – system blokowania dostępu do zasobów rozproszonych.

# David Wheeler (9.2.1927–13.12.2004)

Brytyjski informatyk, profesor Uniwersytetu w Cambridge. W artykule z 1952 roku wprowadził pojęcie podprogramu. W 1994 roku wraz z Michaelem Burrowsem opublikował transformatę Burrowsa-Wheelera.

Dla danego tekstu:

1. Podziel tekst na bloki
2. Utwórz tablicę, której wiersze to wszystkie możliwe rotacje bloku `s + '$'`
3. Posortuj leksykograficznie wiersze tej tablicy
4. Wyślij ostatnią kolumnę tej tablicy

# Transformata Burrowsa-Wheelera – przykład

wejście: ananas

| | |
|---|---|
| ananas$ | $ananas |
| nanas$a | ananas$ |
| anas$an | anas$an |
| nas$ana | as$anan |
| as$anan | nanas$a |
| s$anana | nas$ana |
| $ananas | s$anana |

wyjście: s$nnaaa

# Transformata Burrowsa-Wheelera – przykład

wejście: `To␣niedzwiedz␣czy␣moze␣dzwiedz?␣Chyba␣nie␣dzwiedz.`

wyjście: `?zeeyao.zz␣$by␣eeee␣␣iziiiiCnwwwn␣␣␣Tmzzzzhdddodddc`

Wyjście BWT dobrze się kompresuje

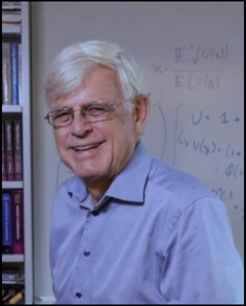
## Transformata Burrowsa-Wheelera – algorytmy

- Drzewo sufiksów
- Tablica sufiksów
- Sortowanie szybkie łańcuchów (Bentley i Sedgewick)

# Jon Bentley (20.2.1953–)

Amerykański informatyk. Autor książki Perełki programowania. Wraz z Robertem Sedgewickiem opisał drzewa trójkowe i oparty na drzewach trójkowych algorytm sortowania szybkiego łańcuchów.

# Robert Sedgewick (20.12.1946–)



Amerykański informatyk. Był doktorantem Donalda Knutha. Odkrył drzewa czerwono-czarne, drzewa trójkowe i kopce parujące. Napisał książkę Algorytmy w C++.

# Sortowanie szybkie łańcuchów

```go
// Sortuje `s`, którego elementy są jednakowej
// długości i mają jednakowe prefiksy [:d].
// Nie wykonuje zbędnych porównań bajtów
func qsort(s [][]byte, d int) {
	if len(s) <= 1 || d > len(s[0]) {
		return
	}
	// Wybierz klucz `v`
	// Podziel `s` wokół klucza `v` według
	// `d`-tego bajtu na trzy części:
	// `sLess`, `sEqual` i `sGreater`
	qsort(sLess, d)
	qsort(sEqual, d+1)
	qsort(sGreater, d)
}
```

# Odwrotna transformata Burrowsa-Wheelera – przykład

wejście: `s$nnaaa`

Wpisz wejście w kolumnie koniec

| wiersz | początek | koniec | `leftShift` |
|---|---|---|---|
| 0 | | `s` | |
| 1 | | `$` | |
| 2 | | `n` | |
| 3 | | `n` | |
| 4 | | `a` | |
| 5 | | `a` | |
| 6 | | `a` | |

# Odwrotna transformata Burrowsa-Wheelera – przykład

wejście: s$nnaaa

Wpisz posortowane wejście w kolumnie początek

| wiersz | początek | koniec | leftShift |
|---|---|---|---|
| 0 | $ | s | |
| 1 | a | $ | |
| 2 | a | n | |
| 3 | a | n | |
| 4 | n | a | |
| 5 | n | a | |
| 6 | s | a | |

# Odwrotna transformata Burrowsa-Wheelera – przykład

wejście: `s$nnaaa`

`leftShift[0] = 1`, bo w obu kolumnach jest tylko 1 symbol `$`

| wiersz | początek | koniec | `leftShift` |
|---|---|---|---|
| 0 | `$` | `s` | 1 |
| 1 | `a` | `$` | |
| 2 | `a` | `n` | |
| 3 | `a` | `n` | |
| 4 | `n` | `a` | |
| 5 | `n` | `a` | |
| 6 | `s` | `a` | |

# Odwrotna transformata Burrowsa-Wheelera – przykład

wejście: `s$nnaaa`

`leftShift[6] = 0`, bo w obu kolumnach jest tylko 1 symbol `s`

| wiersz | początek | koniec | `leftShift` |
|---|---|---|---|
| 0 | `$` | `s` | 1 |
| 1 | `a` | `$` | |
| 2 | `a` | `n` | |
| 3 | `a` | `n` | |
| 4 | `n` | `a` | |
| 5 | `n` | `a` | |
| 6 | `s` | `a` | 0 |

# Odwrotna transformata Burrowsa-Wheelera – przykład

wejście: `s$nnaaa`

`leftShift[1] = 4` lub 5 lub 6, bo w obu kolumnach są po 3 symbole a

| wiersz | początek | koniec | leftShift |
|---|---|---|---|
| 0 | $ | s | 1 |
| 1 | a | $ | |
| 2 | a | n | |
| 3 | a | n | |
| 4 | n | a | |
| 5 | n | a | |
| 6 | s | a | 0 |

# Odwrotna transformata Burrowsa-Wheelera – przykład

wejście: `s$nnaaa`

Wypełniamy `leftShift[1]`, `leftShift[2]` i `leftShift[3]` w kolejności rosnącej

| wiersz | początek | koniec | `leftShift` |
|---|---|---|---|
| 0 | `$` | `s` | 1 |
| 1 | `a` | `$` | 4 |
| 2 | `a` | `n` | 5 |
| 3 | `a` | `n` | 6 |
| 4 | `n` | `a` | |
| 5 | `n` | `a` | |
| 6 | `s` | `a` | 0 |

# Odwrotna transformata Burrowsa-Wheelera – przykład

wejście: `s$nnaaa`

Wypełniamy `leftShift[4]` i `leftShift[5]` w kolejności rosnącej

| wiersz | początek | koniec | `leftShift` |
|---|---|---|---|
| 0 | `$` | `s` | 1 |
| 1 | `a` | `$` | 4 |
| 2 | `a` | `n` | 5 |
| 3 | `a` | `n` | 6 |
| 4 | `n` | `a` | 2 |
| 5 | `n` | `a` | 3 |
| 6 | `s` | `a` | 0 |

# Odwrotna transformata Burrowsa-Wheelera – przykład

wejście: s$nnaaa

Przechodzimy po tablicy `leftShift`: 1-4-2-5-3-6-0, kopiujemy symbole z odpowiednich elementów tablicy początek: ananas$ :-)

| wiersz | początek | koniec | `leftShift` |
|---|---|---|---|
| 0 | $ | s | 1 |
| 1 | a | $ | 4 |
| 2 | a | n | 5 |
| 3 | a | n | 6 |
| 4 | n | a | 2 |
| 5 | n | a | 3 |
| 6 | s | a | 0 |

# Odwrotna transformata Burrowsa-Wheelera

Reguła: jeśli początek[i] = początek[j] oraz `i < j`,
to `leftShift[i] < leftShift[j]`

rotacja[1] < rotacja[2] < rotacja[3] oraz rotacja[4] < rotacja[5] < rotacja[6]

Symbole `? ? ? ? ?` w wierszach 4, 5 i 6 odpowiadają kolejno Symbolom `a ? ? ? ?` w wierszach 1, 2 i 3

| wiersz | rotacja | leftShift |
|---|---|---|
| 0 | `$ ? ? ? ? ? s` | |
| 1 | `a ? ? ? ? ? $` | 4 |
| 2 | `a ? ? ? ? ? n` | 5 |
| 3 | `a ? ? ? ? ? n` | 6 |
| 4 | `n ? ? ? ? ? a` | |
| 5 | `n ? ? ? ? ? a` | |
| 6 | `s ? ? ? ? ? a` | |

# Transformata Burrowsa-Wheelera – zastosowania

BWT + Move to Front + Huffman = bzip2

# Podsumowanie

# Podsumowanie

- Kodowanie Shannona-Fano
- Kodowanie Huffmana
- Kodowanie arytmetyczne
- Kodowanie statyczne i dynamiczne
- Kodowanie słownikowe
- Transformata Burrowsa-Wheelera

# Pomysły, uwagi, pytania, sugestie

Proszę wysyłać podpisane pomysły, uwagi, pytania, sugestie na temat wykładów lub laboratoriów na adres `mgc@agh.edu.pl`

lub wpisywać anonimowe pomysły, uwagi, pytania, sugestie pod adresem `https://tiny.cc/algorytmy-tekstowe`

**Dziękuję państwu za udział w wykładach**
**Życzę państwu powodzenia w dalszej nauce**

# Źródła ilustracji

- https://commons.wikimedia.org/wiki/File:Fireworks_at_the_2010_Celebration_of_Light_in_Vancouver,_BC_02.jpg
- https://en.wikipedia.org/wiki/Claude_Shannon
- https://en.wikipedia.org/wiki/Robert_Fano
- https://www.huffmancoding.com/my-uncle/scientific-american
- https://cml.rhul.ac.uk/people/rissanen/
- https://en.wikipedia.org/wiki/Newton_Faller
- https://en.wikipedia.org/wiki/Robert_G._Gallager
- https://www-cs-faculty.stanford.edu/~knuth/
- https://en.wikipedia.org/wiki/Abraham_Lempel
- https://en.wikipedia.org/wiki/Jacob_Ziv
- https://royalsociety.org/people/michael-burrows-11174/
- https://en.wikipedia.org/wiki/David_Wheeler_(computer_scientist)
- https://engineering.lehigh.edu/dac/jon-bentley
- https://en.wikipedia.org/wiki/Robert_Sedgewick_(computer_scientist)