

takiej analizy istnieje np. znany program hashcat używany przez hakerów (www.hashcat.net). Jego najnowsze wersje pozwalają na wykorzystanie zarówno CPU, jak i GPU naszego urządzenia, co zapewnia szybkość działania oraz wspiera setki algorytmów hashujących.

Podstawową metodą łamania haseł jest użycie metody słownikowej. Polega ona na wykorzystaniu powszechnie znanych haseł (ludzie są leniwi i używają imion dzieci lub zwierząt domowych, prostych haseł typu 123456...). Wystarczy zatem pobierać z takiego słownika hasła „kandydatów” i wyliczać na nich kody hash, porównując je z wykradzionym kodem hash prawdziwego hasła. Owszem, jest to żmudne, ale właśnie taką nudną robotę wykona za nas program. Jeśli tylko program dopasuje wygenerowaną wartości hash do kodu, który chcemy złamać, to ze słownika możemy odczytać hasło w postaci tekstowej!

Techniki kompresji danych

Przejdzmy wreszcie do omówienia metod kompresji danych, czyli algorytmów kodowania pozwalających na zmniejszenie objętości przesyłanych danych.

Jednym z historycznie pierwszych przykładów kompresji danych był *alfabet Morse'a*. Samuel Morse zauważył, że pewne litery alfabetu występują częściej niż inne, i wykorzystał to w swoim systemie kodowania, opartym na dwóch znakach: kreska (-) i kropka (.), łatwych do przesyłania za pomocą telegrafu. Zmniejszenie długości przesyłanych tekstów uzyskane zostało poprzez zakodowanie częściej występujących znaków krótszym ciągiem kodowym, a znaków występujących rzadziej — dłuższym⁸ (tabela 18.3).

Sygnal pozwalający na przesłanie kropki określa też umowną jednostkę czasu umożliwiającą rozróżnianie znaków „na słuch”⁹, np. przesłanie kreski powinno trwać tyle co przesłanie trzech kropek; podobnie ściśle określony jest odstęp pomiędzy elementami znaku (tyle samo co kropka). Odstęp pomiędzy poszczególnymi znakami trwa tyle co trzy kropki, a odstęp pomiędzy grupami znaków (słowniami) — siedem kropek. Konsepcyjnie podobny system kodowania jest zastosowany w alfabetie Braille'a, gdzie znak (wyraz) reprezentowany jest przez dwuwymiarową tabliczkę o rozmiarze 2×3 , zawierającą punkty wypukłe lub płaskie (zw. sześciopunkt, zwany znakiem tworzącym)¹⁰. Ponieważ poszczególnych wartości jest $2^6 = 64$, co przekracza liczbę znaków alfabetu (26), pozostałych kodów można użyć do reprezentowania często występujących słów — w tym celu jeden

⁸ Pretendentów przyznających się do wynalezienia tego systemu kodowania było wielu, ale do historii przeszedł Samuel Morse, który pierwszą depeszę napisaną tym alfabetem przesłał z Waszyngtonu do Baltimore w 1844 r.

⁹ Wielu radioamatorów znajomość tego kodu traktuje ciągle jako punkt honoru.

¹⁰ Patrz <http://www.braille.pl/index.php?body=system>.

TABELA 18.3. Alfabet Morse'a

SYMBOL	KOD	SYMBOL	KOD	SYMBOL	KOD	SYMBOL	KOD
A	· -	J	· - - -	T	-	3	· · -
B	- · · ·	K	- - -	U	· · -	4	- · · -
C	- - · -	L	· - - ·	V	· · · -	5	· · · -
D	- - ·	M	- -	W	· - -	6	- · · -
E	·	N	- ·	Y	- - - -	7	- - - -
F	· · - -	O	- - -	Z	- - · ·	8	- - - -
G	- - -	P	- - - ·	0	- - - -	9	- - - -
H	· · · ·	R	· - ·	1	· - - -		
I	· ·	S	· · ·	2	· · - -		

znak jest przeznaczony do poinformowania odbiorcy, że za nim wystąpi całkowita sŁowa, a nie znak¹¹.

W ogólnym modelu matematycznym kompresji danych mamy — podobnie jak dla kodowania i dekodowania — do czynienia z dwoma algorytmami: kompresującym i rekonstruującym. Co się tyczy samych algorytmów, to w zasadzie dwie najpopularniejsze metody klasyfikacji dzielą algorytmy kompresji na:

- *bezstratne* (po zakodowaniu i zrekonstruowaniu uzyskujemy wynik zgodny w 100% z oryginałem);
- *stratne* (po zakodowaniu i zrekonstruowaniu wynik może się nieco różnić od oryginału).

W kompresji stratnej dopuszcza się pewną utratę informacji, oczywiście w zasztowaniach, które to umożliwiają, np. kompresji mowy lub obrazu. Na pewno kompresji stratnej nie można użyć w systemach, w których liczy się 100% wiarygodności, np. bankowych lub innych systemach tej klasy odpowiedzialności biznesowej.

Algorytmów kompresji jest bardzo dużo i na dodatek dziedzina ta jest w trakcie ciągłego rozwoju, wymuszonego potrzebami współczesnej technologii. Bez wątpienia w poboczne klasyfikacje można powiedzieć, że głównymi kryteriami oceny algorytmów kompresji są:

¹¹ Alfabet został opublikowany w 1837 r. przez Louisa Braille'a, francuskiego pedagoga, niewidomego od dziecka, który pracując jako nauczyciel w zakładzie dla niewidomych, chciał umożliwić niewidzącym dzieciom czytanie.

- szybkość działania,
- stopień kompresji, tj. współczynnik, w jakim uległ zmniejszeniu rozmiar danych wejściowych po przetworzeniu ich przez algorytm kompresji.

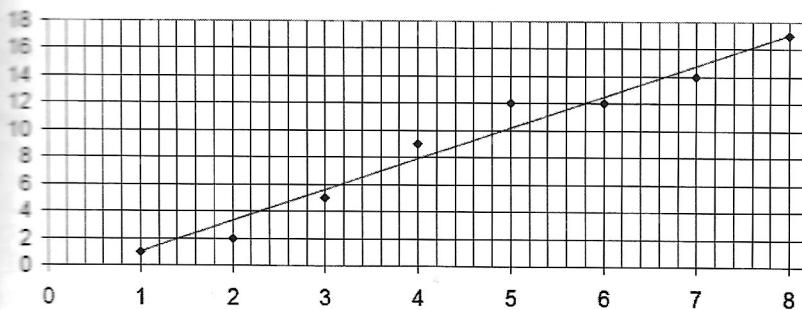
Wszelkie algorytmy kompresji wykorzystują to, że w plikach z danymi znajduje się więcej danych, niż jest to faktycznie potrzebne do przekazania tej samej zawartości informacyjnej. Taką właściwość zbioru danych nazywamy fachowo redundancją — pełną definicję matematyczną sobie darujemy, gdyż nie chciałbym zamieścić jej książki w zbiór teorii informacji. Zresztą samo pojęcie nie jest egzotyczne, w językoznawstwie jest to występowanie w języku elementów funkcjonalnych zbędnych.

Kompresja za pomocą modelowania matematycznego

Kompresja przy użyciu modelowania matematycznego polega na próbie odtworzenia redundancji zbioru danych. Metoda ta daje się dość łatwo wytlumaczyć na prostym przykładzie, który niestety ma to do siebie, że nie jest w praktyce zbyt użyteczny. Jednak ma on swoje walory edukacyjne i pozwala opisać typowy schemat działania algorytmów tego rodzaju.

Załóżmy, że podczas transmisji danych przekazaliśmy następujący ciąg liczb: 1, 2, 5, 9, 12, 12, 14, 17. Jest to fragment przekazu i naszym zadaniem będzie sprawdzenie, czy w celu przekazania tego ciągu trzeba koniecznie przekazać aż 8 = 40 bitów (każda z liczb od 1 do 17 może zostać zakodowana binarnie na 5 bitach) — być może ta sama informacja mogłaby zostać przekazana nieco zwięźlej?

Jednym ze sposobów jest wydedukowanie funkcji matematycznej, która na podstawie numeru sekwencyjnego próbki pozwoli wyliczyć wartość przekazywanego znaku. Aby odtworzyć taką funkcję, spróbujemy narysować prosty wykres danych (rysunek 18.3).



RYSUNEK 18.3. Kompresja wykorzystująca matematyczne modelowanie zbioru danych

Już na pierwszy rzut oka widać, że dane układają się z dobrym przybliżeniem w kształcie prostej (jest nałożona na wykres). Założymy, że zamodelujemy ten ciąg danych funkcją $(7n-4)/3$, gdzie n jest numerem próbki. Porównajmy teraz ciąg kodowy z naszym modelem matematycznym:

Zastanówmy się zatem, jakie powinno być kryterium wyboru, czy dane kodować, czy nie.

Wbrew pozorom nie jest to specjalnie skomplikowane, jeśli uświadomimy sobie, czemu w ogóle służy kompresja. W nawale prezentowanych algorytmów mogło nam to umknąć, zatem przypomnę, że naszym celem jest zmniejszenie rozmiaru bloku danych bez utraty jego zawartości informacyjnej.

Czy warto zakodować jeden znak Z? Oczywiście nie, bo zamiast efektu kompresji doprowadzimy nawet do wzrostu objętości pliku danych:

Z → *1Z (trzy znaki w miejsce jednego).

Podobnie ma się sprawia z dwoma znakami, a nawet trzema, gdzie nie ma ani kompresji, ani wzrostu objętości pliku danych:

ZZZ → *3Z (trzy znaki w miejsce... trzech).

Kompresja LRE staje się zatem efektywna, gdy przepuszczamy bez zmian sekwencje od jednego do trzech znaków, a koncentrujemy się na kompresji powtarzalnych bloków o rozmiarze od czterech znaków wzwyż.

 W katalogu *Dodatki* w archiwum ZIP dostępnym na FTP, w pliku *rle8_sc.zip*, znajdują się przykłady procedur kodujących i dekodujących RLE autorstwa Shauna Case'a. Są to programy na licencji *public domain*. Oryginalnie są napisane w języku C dla kompilatora Borland C++ 2.0, ale przy pewnym wysiłku można je skompilować w innych kompilatorach (przeczytaj porady w *rle.txt*).

Kompresja danych metodą Huffmmana

Pierwszym algorytmem, który opiszę szczegółowo, jest *algorytm Huffmmana*; to wręcz akademicki przykład, jak utworzyć dobry algorytm dający się łatwo implementować za pomocą współczesnych języków programowania. Sam algorytm należy do licznej klasy tzw. *algorytmów prefiksowych*, jego opis poprzedzę jednak wstępem, który ma na celu wytłumaczenie głównej idei, na której ten algorytm bazuje.

Kod, którego zdecydujemy się używać, może się znacznie różnić od znanego kodu ASCII. Jak już pisałem, kod ASCII jest tabelą 8-bitowych znaków tekstu (nie wszystkie są co prawda używane w języku polskim, ale nie ma to tutaj większego znaczenia). Jego podstawową cechą jest równa długość każdego słowa kodowego odpowiadającego danemu znakowi: 8 bitów. Czy jest to obowiązkowe? Otóż nie — spójrz na przykład kodowania znaków pewnego alfabetu 5-znakowego (tabela 18.4).

Gdzieś w dalekiej dżungli żyje lud, który potrafi za pomocą kombinacji tych pięciu znaków wyrazić wszystko: wypowiedzenie wojny, rozejm, prośbę o żywność, prognozę pogody itd.

TABELA 18.4. Przykład kodowania znaków pewnego alfabetu 5-znakowego

ZNAK	KOD BITOWY
Ѡ	000
Ѽ	001
Ѽ	01
Ѽ	10
Ѽ	11

Teksty zapisywane są na liściach pewnej odpornej na działanie pogody rośliny. W celu szybkiej komunikacji został wymyślony system szybkiego przesyłania wiadomości za pomocą sygnałów trąb niosących dźwięk na bardzo długie dystanse.

Dwa krótkie sygnały oznaczają znak ☺, krótki i długi oznaczają ☺ itd., zgodnie z tabelą 18.4 (0 — sygnał dlugi, 1 — krótki). Jest godne docenienia, że mamy przed sobą niewątpliwie kod... binarny! (Nawet jeśli ów tajemniczy lud nie zdaje sobie z tego sprawy).

Założymy, że pewnego dnia odebrano następujący sygnał: 011110000001 (nadawca przekazał wiadomość: ☺☺☺☺☺, czyli „doślijcie świeże melony”). Czy możliwe jest nieprawidłowe odtworzenie wiadomości, tzn. ewentualne pomyłenie jednego znaku z innym? Spróbujmy:

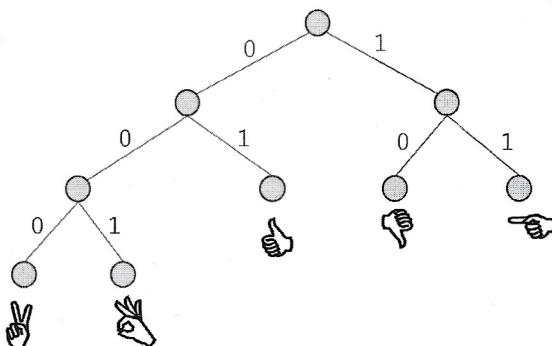
- 0 — znakiem może być ☺ lub ☺, lub ☺.
 - 01 — już wiemy, że jest to ☺!
 - 01 1 — znakiem może być ☺ lub ☺.
 - 01 11 — już wiemy, że jest to ☺!
 - 01 11 1 — znakiem może być ☺ lub ☺
- itd.

Pomyłki są, jak to wyraźnie widać, niemożliwe, gdyż żaden znak kodowy nie jest przedrostkiem (prefiksem) innego znaku kodowego. Dotarliśmy do istotnej cechy kodu: ma on być jednoznaczny, tzn. nie może być wątpliwości, czy dana sekwencja należy do znaku X, czy też może do znaku Y.

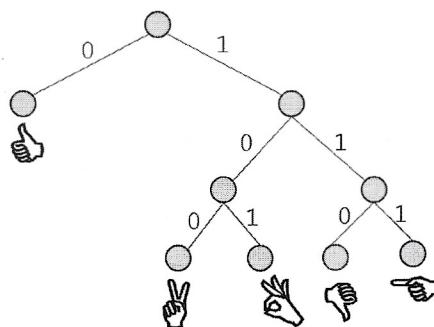
Konstrukcja kodu o powyższej własności jest dość łatwa w przypadku reprezentacji alfabetu w postaci tzw. drzewa kodowego. Dla naszego przykładu wygląda ono tak jak na rysunku 18.4.

Przechadzając się po drzewie (od jego korzenia aż do liści), odwiedzamy gałęzie oznaczone etykietami 0 (lewe) lub 1 (prawe). Po dotarciu do danego liścia ścieżka, po której szliśmy, jest jego binarnym słowem kodowym. Zasadniczym problemem drzew kodowych jest ich... nadmiar. Dla danego alfabetu można skonstruować cały las drzew kodowych, o czym świadczy przykład na rysunku 18.5.

ZRZUNEK 18.4.
Przykład drzewa kodowego do analizy prefiksowej (1)



ZRZUNEK 18.5.
Przykład drzewa kodowego do analizy prefiksowej (2)



Powstaje więc pytanie: „Które drzewo jest najlepsze?”. Oczywiście kryterium jakości drzewa kodowego jest związane z naszym głównym celem, czyli kompresją. Kod, który zapewni największy stopień kompresji, będzie uznany za najlepszy. Warto zwrócić uwagę, że długość słowa kodowego nie jest stała (w naszym przykładzie wynosiła dwa lub trzy znaki). Jeśli w jakiś magiczny sposób sprawimy, że znaki występujące w kodowanym tekście najczęściej będą miały najkrótsze słowa kodowe, a znaki występujące sporadycznie — najdłuższe, to uzyskana reprezentacja bitowa będzie miała najmniejszą długość w porównaniu z innymi kodami binarnymi.

Na tym spostrzeżeniu bazuje kod Huffmana, który służy do uzyskania optymalnego drzewa kodowego. Jak nietrudno się domyślić, potrzebuje on danych na temat częstotliwości występowania znaków w tekście. Mogą to być wyniki uzyskane od leksykografów, którzy policzyli prawdopodobieństwo występowania określonych znaków w danym języku, lub po prostu nasze własne wyliczenia bazujące na wstępnej analizie tekstu, jaki ma zostać zakodowany. Sposób postępowania zależy od tego, co zamierzamy kodować (i ewentualnie przesyłać): teksty języka mówionego, dla którego prawdopodobieństwa występowania liter są znane, czy też losowe w swojej treści pliki „binarne” (np. obrazy, programy komputerowe).

Dla zainteresowanych podaję tabelkę zawierającą dane na temat języka polskiego (przytaczam za serwisem internetowym PWN¹² — tabela 18.5).

¹² Patrz <https://sjp.pwn.pl/poradnia/haslo/frekwencja-liter-w-polskich-tekstach;7072.html>.

TABELA 18.5. Prawdopodobieństwa występowania liter w języku polskim

PRAWDO-LITERA PODOBIEŃSTWO	PRAWDO-LITERA PODOBIEŃSTWO	PRAWDO-LITERA PODOBIEŃSTWO	PRAWDO-LITERA PODOBIEŃSTWO	PRAWDO-LITERA PODOBIEŃSTWO
a 8,91%	w 4,65%	p 3,13%	g 1,42%	ć 0,40%
i 8,21%	s 4,32%	M 2,80%	ę 1,11%	f 0,30%
o 7,75%	t 3,98%	U 2,50%	h 1,08%	ń 0,20%
e 7,66%	c 3,96%	J 2,28%	ą 0,99%	q 0,14%
z 5,64%	y 3,76%	L 2,10%	ó 0,85%	ż 0,06%
n 5,52%	k 3,51%	ł 1,82%	ż 0,83%	v 0,04%
r 4,69%	d 3,25%	B 1,47%	ś 0,66%	x 0,02%

Algorytm Huffmana korzysta w bezpośredni sposób z takich właśnie tabelek. Wyszukuje i grupuje rzadziej występujące znaki, tak aby w konsekwencji przypisać im najdłuższe słowa binarne, natomiast znakom występującym najczęściej przypisuje słowa najkrótsze. Może operować prawdopodobieństwami lub częstotliwościami występowania znaków. Poniżej podam klasyczny algorytm konstrukcji kodu Huffmana, który następnie przeanalizujemy na konkretnym przykładzie obliczeniowym.

FAZA REDUKCJI (kierunek: w dół)

1. Uporządkuj znaki kodowanego alfabetu według malejącego prawdopodobieństwa.
2. Zredukuj alfabet poprzez połączenie dwóch najmniej prawdopodobnych znaków w jeden znak zastępczy o prawdopodobieństwie równym sumie prawdopodobieństw łączonych znaków.
3. Jeśli zredukowany alfabet zawiera dwa znaki (zastępcze), skocz do punktu 4., w przeciwnym razie powróć do punktu 2.

FAZA KONSTRUKCJI KODU (kierunek: w góre)

1. Przyporządkuj 0 i 1 dwóm znakom zredukowanego alfabetu słów kodowych.
2. Dopusz 0 i 1 na najmłodszych pozycjach słów kodowych odpowiadających dwóm najmniej prawdopodobnym znakom zredukowanego alfabetu.
3. Jeśli powróciłeś do alfabetu pierwotnego, zakończ algorytm, w przeciwnym razie skocz do punktu 5.

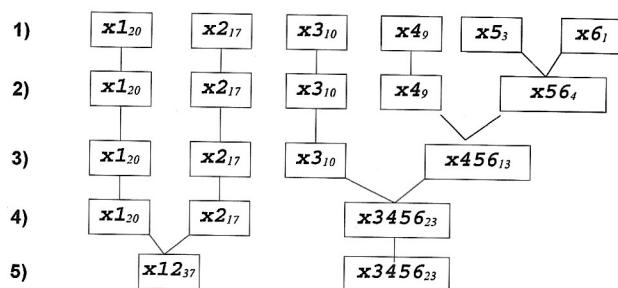
Zdaję sobie sprawę, że algorytm może się wydawać niezbyt zrozumiały, ale wszystkie jego ciemne strony powinien rozjaśnić konkretny przykład obliczeniowy, który zaraz wspólnie przeanalizujemy.

Założymy, że dysponujemy alfabetem składającym się z sześciu znaków: x_1, x_2, x_3, x_4, x_5 i x_6 . Otrzymujemy do zakodowania tekst o długości 60 znaków, których częstotliwości występowania są następujące: 20, 17, 10, 9, 3 i 1. Aby zakodować

6 różnych znaków, potrzeba minimum 3 bitów ($6 < 2^3$), zatem zakodowany tekst zająłby $3 \cdot 60 = 180$ bitów. Popatrzmy teraz, jaki będzie efekt zastosowania algorytmu Huffmmana w celu otrzymania optymalnego kodu binarnego.

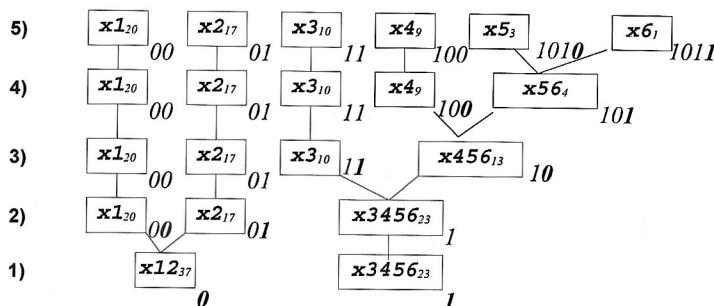
Postępując według reguł zacytowanych w powyższym algorytmie, otrzymamy następujące redukcje (rysunek 18.6).

RYSUNEK 18.6.
Konstrukcja kodu Huffmmana — faza redukcji



Na rysunku przedstawiono kolejne etapy redukcji kodowanego alfabetu (nie sugeruj się postacią rysunku, to jeszcze nie jest drzewo binarne!). Znaki x_5 i x_6 występują najrzadziej, zatem redukujemy je do zastępczego znaku, który oznaczmy jako x_{56} . Podobnie czynimy w każdym kolejnym etapie, aż dojdziemy do momentu, w którym zostają tylko dwa znaki alfabetu (zastępce). Faza redukcji została zakończona i możemy przejść do fazy konstrukcji kodu. Popatrzmy w tym celu na rysunek 18.7.

RYSUNEK 18.7.
Konstrukcja kodu Huffmmana — faza tworzenia kodu



Bitły 0 i 1, które są dokładane na danym etapie do zredukowanych znaków alfabetu, zostały wytłuszczone. Mam nadzieję, że nie będziesz miał zbytnich kłopotów zbudowaniem sposobu konstruowania kodu, tym bardziej że przykład bazuje na krótkim alfabetie.

Przy klasycznej metodzie kodowania binarnego komunikat o długości 60 znaków (napisanego z użyciem 6-znakowego alfabetu) zakodowalibyśmy za pomocą $60 \cdot 3 = 180$ bitów. Przy użyciu kodu Huffmmana ten sam komunikat zająłby $20 \cdot 2 + 17 \cdot 2 + 11 \cdot 2 + 9 \cdot 3 + 3 \cdot 4 + 1 \cdot 4 = 137$ znaków (zysk wynosi ok. 23%).

Wiemy już, jak konstruować kod, warto zatem zastanowić się nad implementacją programową. Nie chciałem prezentować gotowego programu kodującego, gdyż zajęłby zbyt dużo miejsca. Dobrą metodą byłoby skopiowanie struktury graficznej przedstawionej na dwóch ostatnich rysunkach. Jest to przecież tablica 2-wymiarowa o rozmiarze maksymalnym 6×5 . W jej komórkach trzeba by zapamiętywać dość złożone informacje: kod znaku, częstotliwość jego występowania, kod bitowy. Z zapamiętaniem tego ostatniego nie byłoby problemu, możliwości jest wiele: tablica 0-1, liczba całkowita, której reprezentacja binarna byłaby tworzonym kodem itp. Podczas kodowania nie należy również zapominać, aby wraz z kodowanym komunikatem wysłać jego... kod Huffmana! (Inaczej odbiorca miałby pewne problemy z odczytaniem wiadomości).

Problemów technicznych jest zatem dość sporo. Oczywiście zaproponowany powyżej sposób wcale nie jest obowiązkowy. Bardzo interesujące podejście (wraz z gotowym kodem C++) zaprezentowano w [Sed92]. Autor używa tam kolejki priorytetowej do wyszukiwania znaków o najmniejszych prawdopodobieństwach, ale to podejście z kolei komplikuje nieco proces tworzenia kodu binarnego na podstawie zredukowanego alfabetu. Zaletą algorytmów bazujących na „stertopodobnych” strukturach danych jest jednak niewątpliwie ich efektywność, gdyż operacje na stercie są klasy $\log N$, co ma wymierne znaczenie praktyczne! Popatrzmy zatem, jak wyrazić algorytm tworzenia drzewa kodowego Huffmana właśnie przy użyciu tych struktur danych:

```
def Huffman(s, f):
    # s - kodowany binarny ciąg znaków
    # f - tablica częstotliwości występowania znaków w alfabetie
    Wstaw wszystkie znaki ciągu s do sterty H stosownie do ich częstotliwości
    while H nie jest pusta:
        if H zawiera tylko jeden znak X:
            X staje się korzeniem drzewa Huffmana T
        else:
            Weź dwa znaki X i Y z najmniejszymi częstotliwościami
            f1 i f2 i usuń je ze sterty H
            Zastąp X i Y znakiem zastępczym Z, którego częstotliwość
            występowania wynosi f=f1 + f2
            Wstaw znak Z do kolejki H
            Wstaw X i Y do drzewa T jako potomków Z

    return drzewo T
```

Algorytm ten jest oczywiście równoważny podanemu wcześniej, zmieniłem tylko formę zapisu.

Zachęcam do głębszych studiów nad teoriami kodowania i informacji, gdyż są to bardzo ciekawe zagadnienia o dużym znaczeniu praktycznym. Z braku miejsca nie mogłem podjąć wielu interesujących wątków, poza tym pewne zagadnienia trudno przełożyć na łatwy do zrozumienia i zwięzły przykładowy kod. Potraktuj zatem ten rozdział jako wstęp, za którym kryje się bardzo rozległa i ciekawa dziedzina wiedzy!