בס"ד

סיכום - דחיסת נתונים א'

תשפ"ב סמסטר א'

סוכם ע"י: אלמוג יעקב
מרצה: פרופ' דנה שפירא

תוכן עניינים

משפטים והגדרות

קוד UD: קוד שניתן לפענוח בצורה יחידה
קוד מידי: קוד שבו כל מילת קוד מפוענחת בסיום קריאתה

- קוד מידי אינו הכרחי עבור תכונת UD (למשל במחרוזת סופית נשים מחיצות בסיום קריאתה)

קוד שלם: קוד שהוא אינסופי למחצה כך שכשנפענח אותו משמאל לימין הוא תמיד יפוענח באופן UD וגם לא נתקע כאשר נפענח אותו.
אם קוד הוא שלם אזי הוא UD.
עבור כל קוד שהוא שלם מתקיים $K(C) = 1 \leftrightarrow$. הוא לא מבוזבז ואין צמתים פנימיים בעלי בן אחד.
אורך מילת קוד ממוצעת: $E(C,P) = \sum_{i=1}^{n} p_i \cdot |c_i|$
קוד חסר רישות: לא קיימת מילת קוד שהיא רישה של מילת קוד אחרת

- אם קוד חסר רישות ניתן לבנות לו עץ
- אם קוד חסר רישות אז הוא UD
- קוד מידי $\leftrightarrow$ קוד חסר רישות

קוד בעל יתירות מינימלית: קוד C הוא קוד בעל יתירות מינימלית אם מתקיים $E(C,P) \leq E(C',P)$ עבור כל קוד C' חסר רישות בעל n אותיות.
אינפורמציה: $I(s_i) = -\log_2(p_i)$

- צריך לתכנן את הקוד כך שמילת קוד עבור s_i מכילה $I(s_i)$ תווים
  לדוגמה, עבור $p_i = \frac{1}{4}$ יהיה מוצדק לתת למילה 2 סיביות.
- אם מתקיים $p_i = 1$ אז מתקיים $I(s_i) = 0$.
- אם הרצף $s_i s_j$ מופיע בהסתברות $p_i p_j$ אז $I(s_i s_j) = I(s_i) + I(s_j)$.

אנטרופיה: $H(P) = -\sum_{i=1}^{n} p_i \cdot \log_2(p_i)$

- אנטרופיה מהווה חסם תחתון (של סיביות) לאורך מילות הקוד
- את האנטרופיה מעגלים כלפי מעלה כי חייב לקחת מספר שלם של סיביות
- אנטרופיה זה בעצם ממוצע משוקלל של כמויות האינפורמציה
- לכל קוד UD מתקיים $H(P) \leq E(C,P)$
- קוד חסר רישות הוא קוד UD (הפוך לא בהכרח)

קראפט: $K(C) = \sum_{i=1}^{n} 2^{-|c_i|}$

- אם $K(C) \leq 1$ אזי קיים קוד חסר רישות C' כך ש: $E(C,P) = E(C',P)$ וגם $|C| = |C'|$
- אם $K(C) > 1$ אזי לא קיים קוד חסר רישות עם אותם אורכים
- אם קוד הוא UD אז $K(C) \leq 1$
- קיים קוד מידי $\leftrightarrow K(C) \leq 1$

UD TEST:
נגדיר: Dangling Suffix = המשלים של רישה של מילה מסוימת.
לדוגמה, עבור המילים 011,01 המשלים הינו 1. כי הוא משלים את הרישה 01 להיות 011.
אלגוריתם:

- הכנס את כל מילות הקוד לקבוצה (נקרא למילים אלו "מילים מקוריות").
  חזור על התהליך כל עוד שלב 1 מתקיים:
  - בדוק אם קיימת מילה שהיא רישה של מילה אחרת
  - אם קיימת מילה כזו, הוסיף לקבוצה את הDangling Suffix
  - אם הDangling Suffix שהוספנו היא מילת קוד מקורית אזי הקוד לא UD

דחיסות

Static

ההנחה היא שההסתברויות בלתי תלויות ואחידות.
הקוד הוא סטנדרטי ולכן אין prelude.
כמו כן, השימוש הוא בתווי האסקי ולכן ההסתברות לכל תו הינה $\frac{1}{256}$
נשים לב כי האנטרופיה היא 8.

Semi-Static

כאן ההגבלה היא רק על התווים שמופיעים בטקסט במקום להשתמש בכל התווים באסקי.
לכן, לאחר מעבר על הטקסט וספירת כמות התווים (ללא ספירת ההופעות של כל תו כי מניחים שההסתברות לכל תו היא $\frac{1}{n}$) ובהנחה שמצאנו n תווים, אזי, preludeה יראה כך:

- 8 סיביות ראשונות: יהיו מספר התווים
- 8*n הסיביות הבאות: יהיו קודי האסקי של התווים
- שאר הסיביות: מילות הקוד באורך קבוע

$$E(C,P) = H(P) + \frac{8n+8}{|text\ size|}$$

Semi-Static (Self-probabilities)

בשיטה זו עוברים על הטקסט וסופרים את כמות התווים + כמות ההופעות של כל תו.
כל הסתברות של תו תהיה $\frac{number\ of\ occurrences\ in\ the\ text}{text\ size}$
בהנחה שמצאנו n תווים, אזי, preludeה יראה כך:

- 8 סיביות ראשונות: יהיו מספר התווים
- 8*n הסיביות הבאות: יהיו קודי האסקי של התווים
- שאר הסיביות: ביטים המייצגים את ההסתברויות של כל תו

נניח שמצאנו n תווים ונניח שצריך p סיביות על מנת לקודד הסתברות, אזי:

$$E(C,P) = H(P) + \frac{8n+pn+8}{|text\ size|}$$

נעביר את התווים עם ההסתברויות השכיחות תחילה.
כאשר ישנן הסתברויות עצמיות נוכל להשתמש בקודים מתקדמים כדוגמת קוד אריתמטי/קוד הופמן/קוד אליאס.

Unary Code

ממספרים את התווים לפי ההסתברויות (כך שההסתברות השכיחה יותר תקבל אינדקס נמוך)
כל קידוד של תו במיקום i יהיה $1^{i-1}0$
נשים לב כי נקבל עץ משוך ימינה אך המילת קוד האחרונה מייצרת בן בודד
אם מספר התווים סופי נוכל להוריד את ה0 במילת קוד האחרונה (עדיין נוכל לזהות אותה כי 2 מילות הקוד האחרונות נבדלות בסופן ע"י 0 או 1) ואז נקבל עץ מלא – קוד שלם
מקרה פרטי: כאשר ההסתברויות הן חזקות של $\frac{1}{2}$ (כך שההסתברות של המילה הראשונה היא חצי והשנייה רבע וכו') נקבל קוד בעל יתירות מינימלית.

Binary Code

<u>קוד בינארי פשוט</u>: נותנים את מספר הסיביות המינימלי עבורו ניתן לייצג את כמות האלף בית
כלומר עבור אלף בית בגודל n ניתן $\lceil \log_2 n \rceil$ סיביות.
<u>קוד בינארי מינימלי</u>:
נאפשר לחלק ממילות הקוד להיות באורך $\lceil \log_2 n \rceil$ ולחלק להיות באורך $\lceil \log_2 n \rceil - 1$

כמות מילים	אורך
$2n - 2^{\lceil \log_2 n \rceil}$	$\lceil \log_2 n \rceil$
$2^{\lceil \log_2 n \rceil} - n$	$\lfloor \log_2 n \rfloor$

נחשב את הטבלה ונבנה איתה עץ כאשר מילות הקוד הקצרות יהיו בצד שמאל של העץ.
לאחר מכן נחשב ע"פ העץ את מילות הקוד.

Elias

משתמשים בשיטה זו כאשר נתונות לנו ההסתברויות העצמיות.

קוד C_γ:

נסדר את האותיות לפי ההסתברויות שלהן (הכי נפוץ יהיה ראשון עם אינדקס 1).

כל מילת קוד תראה כך: XY

כאשר X הוא **מספר הסיביות** הנדרש על מנת לייצג את האינדקס באונארי

וY הוא האינדקס עצמו בבינארי ללא ה1 המוביל (הכי שמאלי).

לדוגמה,

מילת הקוד עבור אינדקס 1 היא 0

מילת הקוד עבור אינדקס 3 היא 101

מילת הקוד עבור אינדקס 6 היא 11010

קוד C_δ:

זהה לחלוטין לקוד C_γ פרט לכך שX מקודד בעזרת C_γ במקום באונארי.

לדוגמה,

מילת הקוד עבור אינדקס 6 היא 10110 (כי 3 בקוד C_γ הוא 101)

קוד C_γ ארוך יותר מקוד אונארי רק עבור אינדקסים 2,4

קוד C_δ ארוך יותר מקוד C_γ רק עבור אינדקסים 2,3,...,15

אם נרצה להמשיך לייעל לקוד C_ε אז זה יהיה מורגש רק במספרים גבוהים

Golomb + Rice

קוד Golomb הוא קוד שמקודד כל מילת קוד כך שהרישא שלה היא קוד אונארי והסיפא היא קוד בינארי **מינימלי**. ישנם כמה ורסיות של קוד Golomb בהתאם לפרמטר b הנקרא "דלי".

נשים לב שRice הוא מקרה פרטי של Golomb כאשר הדלי הוא חזקה של 2. כלומר $b=2^k$.

אלגוריתמים פשוטים:

Encode	Decode
$Golom\_encode(x,b)$: $q = \lfloor (x-1)/b \rfloor$ $r = x - q * b$ $p1 = Unary\_encode(q+1)$ $p2 = Minimal\_binary\_encode(r,b)$ $return\ p1 \cdot p2$ p2 היא מילת הקוד הr בא"ב בינארי **מינימלי** בגודל b. (כאשר המילים מסודרות בסדר לקסיקוגרפי ומהקצרה לארוכה). לדוגמה: אם r=1, b=5 נקבל p2=00 אם r=2, b=5 נקבל p2=01 אם r=4, b=5 נקבל p2=110	$Golom\_decode(xy,b)$: $q = Unary\_decode(x) - 1$ $r = Minimal\_binary\_decode(y)$ $return\ r + q * b$ כאשר xy היא מילת הקוד אותה מפענחים. x הוא הרישה (האונארית) של מילת הקוד (ניתן לזיהוי כי מסתיים ב0) y זה שאר מילת הקוד

לדוגמה,

עבור דלי b=5 קידוד הספרה 9 הינו 10110

עבור דלי b=5 קידוד הספרה 7 הינו 1001

עבור קידוד Rice כאשר k=2. כלומר, דלי בגודל b=4 קידוד הספרה 7 הינו 1010

Fibonacci Code

כל מספר ניתן לייצוג בעזרת מספר פיבונאצ'י.

דוגמה: (בסדרת פיבונאצ'י יש פעמיים 1 אז נתעלם מאחד מהם כמו בטבלה)

binary		128	64	32	16	8	4	2	1
73		0	1	0	0	1	0	0	1
fibo	55	34	21	13	8	5	3	2	1
73	1	0	0	1	0	1	0	0	0

בקוד זה לא ניתן לקבל 1 פעמיים, אחרת היינו יכולים לקחת את מספר הבא בסדרה.
לכן, כשנראה 1 פעמיים נדע שהגענו למילת קוד חדשה.
מכיוון שאנו רוצים שהקוד יהיה מידי (וכאן הוא לא מידי כי רק בתחילת המילה הבאה נדע שסיימנו את הקודמת) אז נהפוך את המילים כך ש11 יהיה בסוף.
תכונות:

- קוד פיבונאצ'י הוא קוד שלם (הוכחנו בעזרת קראפט)
- יש F_k מילות קוד באורך k+1 (כולל ה1 שמוסיפים בסוף) כאשר $F_1 = F_2 = 1$.
  מספר j המקיים $F_{k+1} \le j \le F_{k+2}$ צריך k+1 סיביות $F_{k+2} - F_{k+1} = F_k$

חיסרון בקוד הנ"ל הוא שאין מילת קוד באורך 1 (כי הקצרה ביותר היא 11)
ואם יש לנו מילה עם הסתברות ממש גבוהה אז זה יהיה מבוזבז.
<u>ורסיות נוספות הפותרות את הבעיה</u> (של החסרון שאין מילה באורך 1):

1. ורסיה ראשונה:
   - ניקח רק את המילים שמתחילות ב1 (לאחר שהפכנו)
   - נוריד את הסיבית האחרונה (שהיא 1 כי מסתיים ב11)
2. ורסיה שניה (שקולה לראשונה):
   - נוריד את הסיבית האחרונה (שהיא 1 כי מסתיים ב11)
   - נוסיף 10 להתחלה של כל מילת קוד
   - לבסוף נוסיף את המילה 1 כמילת הקוד הראשונה

<u>תכונות של הוורסיות הנוספות</u>:

- מכילות מילת קוד אחת בלבד באורך 1.
- פרט למילה הראשונה, ישנן F_{k-2} מילות קוד באורך k.
- ניתן להוכיח שתמיד יותר טוב מקוד C_δ וקוד C_γ.

Shannon Code (Coding blocks of symbols, Shannon-Fano codes)

שאנון הציע שאם נעגל את כמות האינפורמציה (עבור כל אות) כלפי מעלה נוכל לבנות קוד הקרוב לאנתרופיה בלכל היותר סיבית בודדת. (כלומר, נפסיד סיבית בכל מילת קוד [בכל מופע בטקסט]).
אנו לוקחים את כמות האינפורמציה של כל אות ומעגלים כלפי מעלה (כי לא יתכנו שברי ביטים)
אם ננסה לחשב את הקראפט עבור האילוץ הזה (עיגול כלפי מעלה) נקבל קראפט קטן-שווה ל1
שזה אומר שקיים קוד חסר רישות עבור אותם אורכים.
עבור עיגול של האינפורמציה כלפי מעלה. כלומר: $\log_2 \frac{1}{p} \le \left\lceil \log_2 \frac{1}{p} \right\rceil \le \log_2 \frac{1}{p} + 1$
ניתן להוכיח כי $H(P) \le L \le H(P) + 1$

<u>Coding blocks of symbols</u>:

נניח שהסתברות של זוג תווים הוא בלתי תלוי.
הוכחנו כי אנתרופיה על 2 תווים שווה לפעמיים אנתרופיה על תו אחד. $H\left(p(s_i, s_j)\right) = 2H(p(s))$
לכן, אם נחשב את האנתרופיה עבור זוגות של תווים כמו שעשינו לעיל נקבל שנתרחק מהאנתרופיה בסיבית אחת. אבל זה עבור זוג תווים ולכן עבור על תו אנו מתרחקים מהאנתרופיה בחצי סיבית.
אם נעשה זאת עבור שלשות נתרחק מהאנתרופיה ב1/3 סיבית. וכו'.

החולשה של שיטה זו שאנו יכולים לקבל עץ שאינו מלא.

encoding full binary tree:

תזכורת: בעץ מלא עם n עלים יש n-1 צמתים פנימיים
לכן, נצטרך 2n-1 ביטים כדי לקודד עץ בינארי עם n עלים (n עלים + n-1 צמתים פנימיים).
בנוסף, נזכיר כי כל קוד חסר רישות ניתן לייצוג ע"י עץ
על מנת לקודד את העץ נשלח מחרוזת שאיתה נוכל לשחזר את העץ. נבצע זאת בצורה הבאה:

- 8 סיביות ראשונות: מספר האלף בית
- 8*n סיביות הבאות: קידודי האסקי עבור כל אות
(נניח שהאותיות המתקבלות מסודרות מהעלה השמאלי לעלה הכי ימני לפי הסדר)
- נעביר מחרוזת המייצגת את העץ כך שכל סיבית היא אינדיקטור עם צומת מסוים הוא צומת פנימי (1) או עלה (0).

לדוגמא עבור המחרוזת 11110010000:
רמה ראשונה: הספרה הכי שמאלית היא 1 ולכן השורש הוא צומת פנימי.
רמה שניה: הספרות הבאות הן 11 ולכן הבנים של השורש גם צמתים פנימיים.
רמה שלישית: הספרות הבאות הן 1001 ולכן הצומת הכי ימני והכי שמאלי הם צמתים פנימיים אבל הצמתים האמצעיים הם עלים ולכן אין להם בנים. כלומר, אנו יודעים שברמה הבאה מצפים לנו 4 צמתים ולא 8. לכן, צריך לפענח את 4 הביטים הבאים (לא יותר ולא פחות).
רמה רביעית: הספרות הן 0000 ולכן נקבל 4 עלים כפי שהסברנו לעיל.

Shannon-Fano codes:

אלגוריתם:
נסדר את ההסתברויות מהגדול לקטן.
כל פעם נחלק את ההסתברויות לשני קבוצות כך שההפרש בניהם יהיה מינימלי ככל הניתן (אם החלוקה אינה ודאית ויש התלבטות עבור הסתברות מסוימת אז ניקח אותה לקבוצה השמאלית שבה האורך קצר יותר)
בכל חלוקה לקבוצה השמאלית ניתן את הספרה 0 ולימנית את הספרה 1.

Prob.	0.67	0.11	0.07	0.06	0.05	0.04
	0	1				
		0		1		
		0	1	0	1	
					0	1
Code	0	100	101	110	1110	1111

קוד זה אינו תמיד אפקטיבי.

Huffman Code (Minimum Redundancy Coding)

אלגוריתם:

```
Huffman(Σ):
  n = |Σ|
  Q1 = Σ // min heap
  For i=1 to n-1:
    Do allocate-node(Z) // new node
    z.left = x = Extract_min(Q1)
    z.right = y = Extract_min(Q1)
    z.weight = x..weight + y.weight
    insert(Q1,z)
return Extract_min(Q1)
```

סיבוכיות: O(nlogn)
אם אנו מקבלים את השכיחויות בצורה ממוינת ניתן לבצע את האלגוריתם בסיבוכיות של O(n) ע"י שימוש בשני תורים (התור הראשון הוא הקלט הממוין ואת הסכומים המתקבלים שמים בתור השני) בכל פעם שנשלוף 2 מינימליים נחפש אותם ב2 המקומות האחרונים בכל תור.

משפט: בהינתן משקלים $w_1, \dots, w_n$ הופמן מקצה אורכי קוד $l_1, \dots, l_n$ כך ש $\sum_{i=1}^{n} w_i l_i$ הוא מינימלי.
למה 1: קוד אופטימלי עבור קובץ תמיד מיוצג ע"י עץ בינארי מלא כך שלכל צומת פנימית יש 2 ילדים.
למה 2: בעץ אופטימלי ה2 ילדים עם המשקלים הכי נמוכים נמצאים ברמה הנמוכה ביותר.
למה 3: בעץ אופטימלי ה2 ילדים עם המשקלים הכי נמוכים יכולים להיות אחים.
בהחלפת 0/1 עבור צמתים פנימיים הופמן יכול לתת 2^n קודים שונים.

Huffman (Canonical)

עץ הופמן קנוני הוא עץ הופמן משוך שמאלה/ימינה

נראה האלגוריתם שבהינתן קוד הופמן פולט עץ הופמן קנוני משוך שמאלה.
בהינתן אותיות $\{i\}$ ואורך אופטימלי $\{l_i\}$

```
maxlength=max{l_i}
// find the number of codewords of each length
For l = 1 to maxlength
  num[l]=0
For i = 1 to n
  num[l_i]++
// store the first codeword
Firstcode[maxlength]=0
For l=maxlength-1 downto 1
  firstcode[l] = (firstcode[l+1]+num[l+1])/2
For l=1 to maxlength
  nextcode[l] = firstcode[l]
For i=1 to n
  codeword[i]=nextcode[l_i]
  symbol[l_i,nextcode[l_i]-firstcode[l_i]] = i
  nextcode[l_i]++
```

לדוגמה, נפעיל את האלגוריתם עבור קוד ההופמן הבא (שאינו קנוני):

מילת קוד	תו	מילת קוד	תו
0110	a	101	e
0111	b	110	f
010	c	111	g
100	d	11	h

num[4]=2, num[3]=5, num[2]=1

מילת הקוד הראשונה באורך 4 הינה 0000 ע"פ שורה 6 באלגוריתם. 0 ב-4 סיביות.
מילת הקוד הראשונה באורך 3 הינה 001. ע"פ שורה 7 באלגוריתם. 1=2/(0+2) ב-3 סיביות.
מילת הקוד הראשונה באורך 2 הינה 11. ע"פ שורה 7 באלגוריתם. 3=2/(1+5) ב-2 סיביות.
מכאן על פי הלולאה השנייה והשלישית באלגוריתם נקבל:

מילת קוד	תו	מילת קוד	תו
0000	a	011	e
0001	b	100	f
001	c	101	g
010	d	11	h

וסיימנו.

על מנת לייצג עץ הופמן קנוני (כלומר, להעביר אותו למפענח) נצטרך 2 דברים:
1. את האותיות לפי הסדר (מהארוך לקצר ובסדר לקסיקוגרפי)
2. טבלה שבה יש את רק את האותיות שבתחילת כל בלוק (כלומר, המילה הראשונה בכל אורך [מהארוכה אל הקצרה]) כך שהטבלה מכילה את האותיות הנ"ל ואת הקידוד שלה.

עבור הדוגמה לעיל נצטרך להעביר את הסדר הבא: abcdefgh וגם את הטבלה הבא:

s	canonical
a	0000
c	001
h	11

<u>אלגוריתם לפענוח הודעה בינארית</u> (משתמשים במטריצת symbol שבנינו לעיל):

```
maxlength=max{l_i}
v=nextInputBit()
l = 1
while v < firstcode[l] do
  v = 2v+nextInputBit()
  l++
// the integer v is a valid codeword of l bits
return symbol[l,v-firstcode[l]]
// the index of the decoded symbol
```

<u>הגדרה שקולה לעצי הופמן קנוני</u>:

ראינו עץ הופמן קנוני משוך שמאלה. נראה אלגוריתם לבניית עץ הופמן קנוני משוך ימינה.

- תחילה, נפעיל את אלגוריתם הופמן הרגיל
- לאחר מכן, נמיין את מילות הקוד שקיבלנו מהאלגוריתם הרגיל ע"פ האורכים של מילות הקוד
- לכל מילת קוד ניתן הסתברות 2^{-l_i} כאשר l_i היא אורך מילת הקוד
- לכל מילה נגדיר מספר $k = \sum_{j=1}^{i-1} 2^{-l_i}$. כלומר, עבור מילת קוד i אנו סוכמים את ההסתברויות של מילות הקוד שהאינדקס שלהן קטן ממש מהאינדקס i
- מילת הקוד החדשה תהיה l_i הספרות שאחרי הנקודה העשרונית של המספר k הנ"ל

דוגמה: (לאחר שקיבלנו מאלגוריתם הופמן הרגיל את האורכים)

i	s_i	l_i	2^{-l_i}	$\sum_{j=1}^{i-1} 2^{-l_i}$
1	B	2	0.01	0.**00**000
2	D	2	0.01	0.**01**000
3	A	3	0.001	0.**100**00
4	E	3	0.001	0.**101**00
5	F	3	0.001	0.**110**00
6	C	4	0.0001	0.**1110**0
7	G	4	0.0001	0.**1111**0

<u>Huffman (D-ary)</u>

לעץ הופמן די-ארי עם n צמתים פנימיים יש $(D-1)\cdot n+1$ עלים

- לפחות 2 צמתים נמצאים ברמה התחתונה.
- ה2 צמתים עם המשקל הנמוך ביותר נמצאים ברמה התחתונה.
- ה2 צמתים עם המשקל הנמוך ביותר יכולים להיות אחים.

בעקרון, בניית עץ די-ארי שקולה לחלוטין לבניית הופמן רגיל פרט לכך שיתכן שיהיו צמתים עם פחות מD ילדים וזה מבוזבז כי יתכן שלא ניצלנו אורכים קצרים יותר ולכן "נקפיץ" למעלה אותיות (אך ניזהר לשמור על סדר הסתברויות תקין המתקבל מאלגוריתם הופמן [מהעולה ליורד]).

ניתן גם להשתמש באלגוריתם הבא המטפל בבעיה זו:

- חשב מה מספר העלים n_0 שצריך להוסיף כך שמתקיים $n+n_0=(D-1)\cdot n'+1$ לדוג', D=3, n=10 נקבל $10+n_0=2\cdot n'+1$. כלומר, עבור $n_0=1$ נקבל מספר אי-זוגי
- הפעל הופמן עבור D עלים בכל שלב
- תן ערך $0,\dots,D-1$ לכל קשת (הספירה מתחילה מ0)
- התאם עלה להסתברות שלו (וזרוק את העלים עם ה0)

הרעיון הוא שמילות הקוד שנזרוק יהיו הכי ארוכות ובכך לא בזבזנו מילים קצרות.

Huffman (Adaptive)

Sibling property:

- משקל צומת שווה לסכום המשקל של שני ילדיו.
- הצמתים ניתנים לסידור ע"פ משקלם בסדר ≥ כך שמתקיים שהקודקודים 2j-1 וגם 2j אחים.

בקידוד הופמן דינאמי ניתן משקלים לכל אות ע"פ המופעים שלה **תוך כדי הקידוד**. כלומר, נשנה את העץ תמיד כך שהאותיות עם התדירויות הנפוצות יהיו יותר קרובים לשורש.
במילים אחרות, העץ משתנה תמיד.
אין צורך לשמור עץ בסיום הקידוד כיוון שעל סמך ההודעה המקודדת בלבד המפענח יודע לשחזר את העצים בכל שלב בדיוק כמו בצעדים של המקודד אבל בדיליי של צעד אחד.
קידוד האותיות יכול להיות סטנדרטי (אסקי עם 8 סיביות לאות) או אחרת עם prelude.

אינטואיציה:
נגדיר: NYT = הצומת שהמסלול אליו הוא אינדיקציה לכך שקוראים אות חדשה שלא הופיעה לפני.
אנו מתחילים עם שורש שהוא NYT. נשים לב שבתחילה המסלול אליו ריק ולכן בתחילת ההודעה מתחילים ישר לקרוא אות חדשה.
בכל פעם שמגיעים לצומת NYT אנו מוסיפים לו שני בנים. הבן הימני יהפוך לצומת הNYT והבן השמאלי לאות החדשה עם תדירות 0 שנוספה ולאחר מכן נעדכן את העץ.
גם אם הגענו לעלה שאינו NYT אנו מעלים את התדירות שלו ב1 ואז נעדכן את העץ.
חשוב לזכור: תמיד מקודדים ורק אז מעדכנים את העץ. אחרת, המפענח לא יוכל לפענח.
הכנסה לעץ היא עם תדירות 0.

אלגוריתם העדכון: (נניח שהעלה x הוא העלה שהגענו אליו באחד משלבי הקידוד וכעת עלינו לעדכן)

```
q = leaf(x)
if (q is the 0-node)
  replace q by a parent 0-node with two 0-node children;
  q = left child;
if (q is a sibling of a 0-node)
  interchange q with the highest numbered leaf of the same weight;
  increment q weight by 1;
  q = parent of q;
while q!= root
  interchange q with the highest numbered node of the same weight;
  increment q weight by 1;
  q = parent of q;
increment q weight by 1;
```

Huffman (Affix Codes)

לא בחומר?

Huffman (Skeleton)

אינטואיציה: כאשר אנו קוראים חלק ממילת קוד נוכל לפעמים לדעת כמה סיביות נותר לנו לקרוא ואז נקרא אותם בבת אחת. מצב זה קורה כאשר אנו מגיעים לבלוק מסוים (גודלו קבוע). כלומר, זה שקול לכך שבהינתן עץ המתאר את הקוד אם אנו מגיעים לתת עץ שלם אז נדע בדיוק כמה נותר לקרוא.

על מנת ליצור עץ שלד, ניקח את הקוד הופמן הקנוני (המשוך ימינה) ונוריד לו את כל תתי עצים השלמים שגובהם לכל הפחות 1.
כל צומת כנ"ל (שהוא שורש של תת העץ שמחקנו) יהפוך לעלה בעץ שלד, והערך שלו יהיה אורך מילות הקוד מהשורש (הכללי) עד לעלים (של תת העץ שמחקנו). כלומר, אורך מילת הקוד.
הערך של הצמתים הפנימיים יהיה 0.

למפענח שולחים את האלף בית לפי הסדר של העלים בעץ המקורי משמאל לימין (בקנוני תמיד עושים זאת).

הגדרות:
n_i = מספר מילות הקוד באורך i
m = $min\{i|n_i > 0\}$. כלומר, זאת המילה עם האינדקס המינימלי.
$base(i)$ = ערך מספרי של מילת הקוד הראשונה באורך i
- $base(m)$ = 0
- $base(i)$ = $2(base(i-1) + n_{i-1})$

$B_s(k)$ = התצוגה של המספר k בs ביטים בדיוק. (מוסיפים אפסים משמאל אם צריך).
- מילת הקוד באינדקס j באורך i (עבור $j = 0,1, \dots n_i - 1$) היא $B_i(base(i) + j)$

$seq(i)$ = מספר האינדקס של המילה הראשונה באורך i
- $seq(m)$ = 0
- $seq(i)$ = $seq(i-1) + n_{i-1}$

$I(w)$ = הערך המספרי של הייצוג הבינארי w
- אם w באורך l אז מתקיים $w = B_l(I(w))$

מכאן ניתן להסיק:
$I(w) - base(l)$ = האינדקס הפנימי של מילת הקוד w **בתוך הבלוק** של מילות הקוד באורך l
ולכן:
$seq(l) + I(w) - base(l)$ = האינדקס הכללי של מילת הקוד w (כללי = ביחס לכלל מילות הקוד)
אפשר לכתוב זאת כך:
$I(w) - diff(l)$ כאשר $diff(l) = base(l) - seq(l)$
ואז בשביל למצוא אינדקס כללי של מילת קוד מסוימת נצטרך **רק את** $\boldsymbol{diff(l)}$ כי $I(w)$ ניתן לחישוב.

דוגמה:

l	n_l	$base(l)$	$seq(l)$	$diff(l)$
3	1	0	0	0
4	3	2	1	1
5	4	10	4	6
6	8	28	8	20
7	15	72	16	56
8	32	174	31	143
9	63	412	63	349
10	74	950	126	824

0	000
1	0010
2	0011
3	0100
4	01010
5	01011
6	01100
7	01101
8	011100
9	011101
10	011110
11	011111
12	100000
13	100001
14	100010
15	100011
16	1001000
17	1001001
18	1001010
	...
29	1010101
30	1010110
31	10101110
32	10101111
	...
62	11001101
63	110011100
64	110011101
	...
125	111011010
126	1110110110
127	1110110111
	...
199	1111111111

אלגוריתם פענוח:

```
tree_pointer=root
i=1
start=1
while i  < length_of_string
    if string[i]=0
        tree_pointer=left(tree_pointer)
    else
        tree_pointer=right(tree_pointer)
    if value(tree_pointer)>0
        codeword=string[start…(start+value(tree_pointer)-1)]
        output=table[I(codeword)-diff[value(tree_pointer)]]
        tree_pointer=root
        start=start+value(tree_pointer)
        i=start
    else i++
```

Huffman (Reduced Skeleton)

כפי שראינו, בשביל לקצץ תת עץ (על מנת לייצר שלד) היינו צריכים לקצץ תתי עצים שלמים בלבד.
עבור Reduced Skeleton נאפשר לקצץ סוג נוסף של תת עץ:
במידה ומצאנו שישנו תת עץ שההפרש בין אורכי הבנים שלו הם לכל היותר 1 נאפשר לקצץ אותו.
אבל איך נדע בשלב הפענוח מה אורך הבן של תת עץ זה? (הרי ישנם 2 אורכים אפשריים)
אופציה 1: לקרוא עוד כמה תווים עד שנוכל להבדיל בין האורכים (העץ קנוני ולכן ניתן למצוא הפרדה)
אופציה 2: לקחת את הערך של עד איפה שקרנו ולהשוות עם האינדקסים וכך להחליט מה האורך

value(v) = 0 אם v הוא צומת פנימי. אחרת, value(v)=lower(v)
flag(v) = 0 אם v הוא עלה "מיוחד".
table(j) = האלמנט הj כאשר האלמנטים מסודרים בסדר עולה לפי התדירויות.

אלגוריתם פענוח:

```
tree_pointer=root
i=1
start=1
while i  < length_of_string
    if string[i]=0
        tree_pointer=left(tree_pointer)
    else
        tree_pointer=right(tree_pointer)
    if value(tree_pointer)>0
        len=value(tree_pointer)
        codeword=string[start…(start+len-1)]
        if flag(tree_pointer)=1 && 2 I(codeword)≥ base(len+1)
            codeword=string[start…(start+len)
            len++
        output=table[I(codeword)-diff[len]]
        tree_pointer=root
        start=start+len
        i=start
    else i++
```

Arithmetic Code (Static)

- בקוד אריתמטי נוכל לייצג את כל הטקסט הגדול במספר בודד אחד.
- בקוד אריתמטי נוכל להשתמש ב"שברי ביטים" מה שנותן את האנטרופיה.
- מתחילים מאינטרבל חצי פתוח [0,1)
  ונחלק את הקטע בצורה פרופורציונלית לפי ההסתברויות של כל תו.
  לאחר מכן, נעשה כמין zoom-in ונחזור על התהליך.

אלגוריתם קידוד:

```
low = 0.0
high = 1.0
while input symbols remain {
    range = high – low
    get symbol
    high = low + high_bound(symbol)*range
    low = low + low_bound(symbol)*range
}
output any value in [low,high)
```

לדוגמה, נניח אלף בית {a,b,c,d,e,f} והסתברויות {0.2, 0.3, 0.1, 0.2, 0.1, 0.1}
טבלת האינטרבלים תהיה:

	Low bound	High bound
a	0	0.2
b	0.2	0.5
c	0.5	0.6
d	0.6	0.8
e	0.8	0.9
f	0.9	1

אם נרצה לקודד את baccf:

	low	high	range
	0	1	1
b	0.2	0.5	0.3
a	0.2	0.26	0.06
c	0.23	0.236	0.006
c	0.233	0.2336	0.0006
f	0.23354	0.2336	0.00006

והקידוד יהיה מספר כלשהו x בטווח $0.23354 \leq x < 0.2336$

<u>אלגוריתם פענוח (עבור מספר n):</u>

```
Find symbol whose range contains n
Output the symbol
Range = high(symbol) – low(symbol)
Encoded = (encoded-low(symbol))/range
```

לדוגמה n=0.23354:
המספר 0.23354 בטווח של b ולכן נפלוט "b". נקבל range=0.3 וגם n=(0.23354-0.2)/0.3=0.1118
המספר 0.1118 בטווח של a ולכן נפלוט "a". נקבל range=0.2 וגם n=(0.1118-0)/0.2=0.559
המספר 0.559 בטווח של c ולכן נפלוט "c". נקבל range=0.1 וגם n=(0.559-0.5)/0.1=0.59
המספר 0.59 בטווח של c ולכן נפלוט "c". נקבל range=0.1 וגם n=(0.59-0.5)/0.1=0.9
המספר 0.9 בטווח של f ולכן נפלט "f". נקבל range=0.1 וגם n=(0.9-0.9)/0.1=0
סיימנו. קיבלנו את ההודעה "baccf".

- גודל האינטרבל הוא כפל ההסתברויות של כל אות בהודעה שקודדנו.

Arithmetic Code (Adaptive)

בקוד אריתמטי אדפטיבי לא צריכים לשלוח למפענח את הא"ב וההתפלגויות – המפענח קורא ותוך כדי מעבד את הנתונים.

נניח שיש לנו אלף בית {a,b,c}.

נגדיר:

$N(a)$ = מספר התדירויות שהאות a הופיעה.

$P(a)$ = ההסתברות לקבל את a.

כאשר: $P(a) = \frac{N(a)+1}{N(a)+N(b)+N(c)+3}$

וכן לכל אות באלף בית.

נראה דוגמה לקידוד של ההודעה "bccb":

Interval = [0,1)
p(a)=p(b)=p(c)=1/3
N(a)=N(b)=N(c)=0
Encode b:
Interval = [0.3333,0.6667)
N(a)=0, N(b)=1, N(c)=0
p(a)=1/4, p(b)=1/2, p(c)=1/4
Encode c:
Interval = [0.5834,0.6667)
N(a)=0, N(b)=1, N(c)=1
p(a)=1/5, p(b)=2/5, p(c)=2/5
Encode c:
Interval = [0.6334,0.6667)
N(a)=0, N(b)=1, N(c)=2
p(a)=1/6, p(b)=2/6, p(c)=1/2
Encode b:
Interval = [0.639,0.6501)

כלומר, הקידוד יהיה מספר כלשהו x בטווח $0.639 \le x < 0.6501$. לדוגמה, 0.64.

נשים לב גם שמכפלת ההסתברויות שווה לגודל האיטרבל הסופי 0.6501-0.639=0.111

$$p(b)\cdot p(c)\cdot p(c)\cdot p(b) = \frac{1}{3}\cdot\frac{1}{4}\cdot\frac{2}{5}\cdot\frac{2}{6} = \frac{1}{90} = 0.01111111111$$

חסרונות לקידוד אריתמטי:

- אם נרצה לקודד הודעה בגודל *125KB* אז נצטרך 2^{20} שזה בערך 10^6. כלומר מיליון סיביות אחרי הנקודה. (נצטרך לדמות *floating point*).
- צריך לסיים את הקידוד ורק לבסוף לשלוח למפענח. (לא באמת נכון כי ניתן לפתור זאת).
- איטי יותר מהופמן כי באריתמטי אנו מבצעים פעולת חילוק שקצת יותר יקרה מהפעולות של הופמן
- אין אפשרות לגשת באופן ישיר לאמצע קובץ מבלי לפענח הכל כי הפיענוח מתבצע על מספר. בניגוד לכך בהופמן ניתן לגשת לאמצע הקובץ ואז בשלב מסוים נסתנכרן עם מילות הקוד (במידה ולא מדובר בקוד אפיקסי).

Dictionary Code (Static VS Adaptive)

הבדלים של דחיסות מילוניות:
גישה סטטית: המילון סטטי, נבנה לפני הקידוד על סמך ידע קודם על הקובץ.
גישה אדפטיבית (כמו בשיטת LZSS): המילון נבנה בצורה דינאמי כמו בהופמן/אריתמטי דינאמי
גישת סטטיסטיקה (התפלגות): ניסיון חיזוי של התו הבא. כמו בהופמן/אריתמטי/פיבונצ'י/אליאס
גישה תחליפית: שימוש ברצפים של תווים שקודדנו קודם. בדחיסות מילוניות נשתמש בגישה זו.

נניח ונתונות לנו מילות קוד עבור תווים מסוימים + מילות קוד עבור מילים (רצף תווים)
ונניח שקיימת לנו מחרוזת אותה נרצה לקודד.
קידוד גרידי (חמדני): מקודד כרגיל (משמאל לימין) וכל פעם המילה הראשונה שקיים לה קידוד הוא יקודד אותה ישר.
קידוד " השבר הארוך ביותר": מקודד את המחרוזת כך שנותן עדיפות ראשונה לקודד מילים ארוכות יותר במחרוזת.
קידוד "מינימום מילים": מקודד את המחרוזת בצורה בה נקבל כמה שפחות מילים לקודד (בלי קשר לאורכי מילות הקוד)
קידוד אופטימלי: מקודד בצורה בה נקבל בסוף הודעה מקודדת הכי קצרה

ניתן לבנות קוד אופטימלי בצורה הבאה:
בונים גרף כך:
תחילה, עבור כל אות במחרוזת אותה נרצה לקודד נוסיף קודקוד.
כעת, עוברים על המחרוזת כרגיל (משמאל לימין) וכל פעם שמגיעים לאות מחפשים את כל המילים (שקיימות עבורן מילות קוד) שמתחילות באות זו, עבור כל מילה כזו שמצאנו מוסיפים צלע מהאות שעליה אנו מצביעים במחרוזת לאות האחרונה של המילה שמצאנו ומשקל הצלע יהיה אורך מילת הקוד
לבסוף נחפש את המסלול הקצר ביותר מהקודקוד של האות הראשונה במחרוזת לקודקוד של האות האחרונה במחרוזת.

LZ77

אדפטיבי.
כל פוינטר מיוצג באמצעות שלשה
Offset = כמה תווים צריך ללכת אחורה כדי להגיע למחרוזת קודמת אותה נרצה להעתיק
Length = אורך המחרוזת אותה נרצה להעתיק
Symbol = התו הבא (התו הנוסף שנקודד לאחר שהעתקנו את המחרוזת הקודמת)
אלגוריתם קידוד:

```
p=1 // next char to be coded
while (there is text to be coded):
    search for the longest match for S[p…] in s[p-w…p-1]
      // suppose match at pos m with len = 1
    output the triple (p-m,l,s,s[p+1])
    p = p+l+1
```

דוגמה עבור המחרוזת: A_walrus_in_Spain_is_a_walrus_in_vain
ההודעה המקודדת תהיה:

```
(0,0,"A")(0,0,"_")(0,0,"w")(0,0,"a")(0,0,"l")(0,0,"r")(0,0,"u")
(0,0,"s")(7,1,"i")(0,0,"n")(3,1,"S")(0,0,"p")
(11,1,"i")(6,2,"i")(12,2,"a")(21,11,"v")(20,3,".")
```

הצבעה חוזרת:
אם יש תת מחרוזת אותה נרצה להעתיק מספר פעמים והיא עוקבת לפוינטר שלנו אז נוכל לעשות כך:
לדוגמה, עבור a^{100} ההודעה המקודדת תהיה: (0,0,"a")(1,98,"a")

חסרונות של קידוד זה היא העלות של הביטים עבור כל שלשה כזו.
כי עבור תו בודד נצטרך לקודד שלשה שלמה ויתכן שזה יקר מידי.

LZSS

שיפור לקוד LZ77. בקוד LZSS החיפוש יתבצע באמצעות עץ בינארי מאוזן.
במקום להשתמש בשלשה נשתמש בזוג סדור (offset,length) או בתווים בודדים.
נבדיל בין זוג סדור לתו בודד ע"י ביט שיהיה אינדיקטור.
אם הביט הוא 0 אז מצפה לנו תו בודד (ולכן נקבל שקידוד תו בודד הוא 9bit)
אם הביט הוא 1 אז מצפה לנו מחרוזת (ולכן נקבל שקידוד הזוג הוא 1bit + גודל החלון בביטים)

לפעמים הoffset ו/או הlength גדולים מידי ואז יהיה עדיף לנו לקודד גם מחרוזות בגודל שתיים או שלושה כבודדים מאשר באמצעות זוג סדור.
בחיים האמיתיים בודקים מחרוזת ארוכה בעזרת hashing ולא בעזרת עץ בינארי (לא בהכרח נקבל דחיסה מקסימלית אבל הדחיסה תהיה מהירה הרבה יותר)

LZS

אותו רעיון כמו בקוד LZSS אבל כאן נרצה לאפשר 2 סוגים של חלונות בגדלים שונים.
עבור תו בודד הביט האינדיקטור יהיה 0 ואז קוראים 9 ביטים
עבור זוג סדור יהיו 2 ביטים שיהיו אינדיקטור: הביט השמאלי יהיה 1 (על מנת לזהות שמדובר בזוג סדור ולא בתו בודד) והביט הימני יהיה אינדיקטור באיזה גודל חלון נשתמש.
כלומר, נגדיר ככה את סוג הזוג הסדור:

- עבור האינדיקטור 11 נגדיר חלון לטווח 0-128 כלומר בגודל 7bit. סה"כ נקבל 9bit.
- עבור האינדיקטור 10 נגדיר חלון לטווח 129-2048 כלומר בגודל 11bit. סה"כ נקבל 13bit.

חלוקה נוספת:

- עבור האינדיקטור 11 נגדיר חלון לטווח 0-64 כלומר בגודל 6bit. סה"כ נקבל 8bit.
- עבור האינדיקטור 101 נגדיר חלון לטווח 65-320 כלומר בגודל 8bit. סה"כ נקבל 11bit.
- עבור האינדיקטור 100 נגדיר חלון לטווח 321-2368 כלומר בגודל 11bit. סה"כ נקבל 14bit.

LZ78

בדר"כ LZ77/LZSS נותן תוצאה טובה יותר של דחיסה. כיוון שבLZ78 כדי להכניס מילה מסוימת למילון נצטרך שכל הרישות שלה יהיו במילון כפי שנראה למטה.
בקוד LZ78 נראה מילון מפורש. ולא מרומז (window) כמו בLZ77.
כלומר, ניתן ישר את האינדקס – את הכניסה במילון (במקום לחפש אחורה עם window)
נבנה מילון D שיכיל בהתחלה רק את המילה הריקה (שהאינדקס שלה הוא 0) ונכניס אליו תתי מחרוזות (של המחרוזת שנרצה לקודד) וכל תת מחרוזת כזו תקבל את האינדקס הבא בתור שפנוי.
כעת, כל פעם שנקודד תת מחרוזת ננסה "לתפוס" את המילה הכי ארוכה שכבר במילון ונקודד את האינדקס שלה + הקידוד של התו הבא. במקביל, נוסיף למילון את אותה הרישה בשרשור התו הבא שמקודדים (וככה הוספנו מילה קצת יותר ארוכה למילון).
לדוגמה, נניח שיש לנו אלף בית {a,b,c,d}={00,01,10,11}. ונרצה לקודד את הטקסט badadadabaab

index	phrase	encoding	num of bits
0	ε		
1	b	(0,b)	0+2
2	a	(0,b)	1+2
3	d	(0,d)	2+2
4	ad	(2,d)	2+2
5	ada	(4,a)	3+2
6	ba	(1,a)	3+2
7	ab	(2,b)	3+2

נקבל: (ε,01)(0,00)(00,11)(10,11)(100,00)(001,00)(010,01)
שיטה לדעת כמה ביטים צריך לקידוד מס': נשאל – "כמה ביטים צריך בשביל לקודד x מספרים" כאשר x הוא האינדקס באותה שורה. למשל בשורה 2 צריך 0 ביטים בשביל לקודד 1 מספרים.
שימושים LZ77 של נמצאים ב: GZip, Rar
שימושים LZ78 של נמצאים ב: PDF, GIF (כי קוד LZ78 יהיה יעיל אם המחרוזות חוזרות הרבה)

LZW

preluden מקבל את האלף בית ההתחלתי.
נייצר טבלה של כל תו באלף בית עם אינדקס.
גודל הטבלה הוא בחזקות של 2 והטבלה צריכה להיות גדולה ממש מגודל האלף בית כי נרצה להשאיר מקום במילון עבור אחסונים של הרצפים הבאים. לדוג' במקרה סטנדרטי של כל האסקי (שיש בו 256 תווים) המילון יהיה בגודל 512 שזה אומר שכל כניסה מקודדת ב9 סיביות.

אלגוריתם קידוד:

```
Dictionary=single characters
w=first char of input
repeat {
  k=next char
  if(EOF)
      output code(w)
  else if (w · k)∈Dictionary
      w = w · k
  else
      output code(w)
      Dictionary.add(w · k)
      w = k
}
```

קודם פולטים ורק אז מקודדים.
חשוב מאוד: נגדיל את המילון **רק** בשעה שנרצה להכניס מילת קוד חדשה אך המילון מלא.
בכל פעם שפולטים מילת קוד נפלוט אותה במספר הסיביות המתאים לגודל הטבלה.
כלומר, אם הטבלה בגודל 8 אז נקודד את מילת הקוד ב3 סיביות.
לדוגמה, עבור אלף בית {a,b} אם נרצה לקודד את ההודעה abababababab
הטבלה הראשונית תהיה:

code	
0	"a"
1	"b"
2	
3	

ולבסוף נקבל:

w	k
"a"	"b"
"b"	"a"
"a"	"b"
"ab"	"a"
"a"	"b"
"ab"	"a"
"aba"	"b"
"b"	"a"
"ba"	"b"
"b"	"a"
"ba"	"b"
"bab"	

code	
0	"a"
1	"b"
2	"ab"
3	"ba"
4	"aba"
5	"abab"
6	"bab"
7	

וההודעה הינה: 00 01 10 100 011 110

אלגוריתם פענוח:
חשוב מאוד: המפענח תמיד מגדיל את המילון באותו רגע שהמילון מתמלא.

```
Initialize table with single character strings
OLD = first input code
Output translation of OLD
While not end of input stream {
    NEW = next input code
    If new is not in the string table
        S = translation of OLD
        S = S·C
    else
        S = translation of NEW
    Output S
    C = first character of S
    Translation(OLD)·C to the string table
    OLD = NEW
}
```

לדוגמה, עבור אלף בית {a,b} אם נרצה לפענח את ההודעה 110 011 100 10 01 00
הטבלה הראשונית תהיה:

code	
0	"a"
1	"b"
2	
3	

נשים לב כי אנו פולטים בתחילה את המילה הראשונה (במקרה שלנו את a)

OLD	NEW	S	C	Translation(OLD)·C
00	01	"b"	"b"	"ab"
01	10	"ab"	"a"	"ba"
10	100	"aba"	"a"	"aba"
100	011	"ba"	"b"	"abab"
011	110	"bab"	"b"	"bab"

code	
0	"a"
1	"b"
2	"ab"
3	"ba"
4	"aba"
5	"abab"
6	"bab"
7	

נקבל את ההודעה: abababababab

Dictionary Based Compressions - Compress (UNIX), GIF, PNG, Sequitur algorithm

הדוחס של UNIX השתמש בLZW אבל המילון היה מוגבל עד לתקרה של 2^{16} והמילון נשאר סטטי.
ניתן היה לאתחל את המילון כשמזהים שאין ניצול.
הדחיסה של הGIF היא אותו רעיון אבל גדל עד 4096 ומשם נשאר קבוע.
הGIF לא היה חינמי ולכן יצא מתחרה חינמי שהוא PNG המבוסס על LZ77

Run length Code

קוד פשוט המקודד על פי רצף של תווים.
לדוגמה, הרצף acccbbaaabb מקודד כך: (a,1)(c,3)(b,2)(a,3)(b,2)
את האותיות נקודד ב8 סיביות ואת המספרים בfixed-length code או variable
כלומר, ניתן להפעיל על קידוד המספרים את הופמן או אחרת.

Binary Run length Code:

אם נרצה לקודד באותה צורה מחרוזת בינארית מספיק לקודד רק את מספרי ההופעות ללא התווים מכיוון שיש רק 2 תווים – 0 או 1. בנוסף, נצטרך לתת מוסכמה שכל מחרוזת מתחילה תמיד ב0.
עבור ההודעה 00111001 נקודד ככה: 2,3,2,1
בהנחה ומקודדים את המספרים בעזרת fixed-length code אז ישנה בעיה בקידוד מספר מספיק גדול. לדוגמה, עבור קידודים בגודל 3 סיביות, איך נקודד את המספר 10?
נוכל להשתמש בescape codeword. כלומר, אם ניתן את הספרה המקסימלית ב3 סיביות לא נחליף בין התווים (מ0 ל1 או הפוך)
דוגמה (בהנחה ואנו מקודדים הופעות של 1):
קידוד הספרה 10 ב3 סיביות: 111011
קידוד הספרה 7 ב3 סיביות: 111000
קידוד הספרה 14 ב3 סיביות: 111111000

The Burrows Wheeler Transform

דחיסה המשומשת ע"י bzip2.
בהנחה שהטקסט בגודל n אז על מנת לקודד אותו, נבנה תחילה מטריצה מסדר nxn
השורות יהיו כל הטרנספורמציות הציקליות של הטקסט.
כלומר, השורה הראשונה תתחיל באות הראשונה של הטקסט
השורה השניה תתחיל מהאות השניה של הטקסט וכו'
(ממשיכים לשכתב את הטקסט בצורה מעגלית בסוף השורה)
לאחר מכן, נמיין את השורות בסדר לקסיקוגרפי (לפי האלף בית abcdefghijklmnopqrstuvwxyz)
לבסוף מה שנצטרך זה העמודה האחרונה במטריצה + אינדקס של השורה המקורית של הטקסט.
כאשר נעביר את הטקסט של העמודה מלמעלה למטה וכאשר הספירה של האינדקסים מתחילה מ0.

ישנה תכונה בה אם ניקח את העמודה האחרונה ונשרשר אותה להיות העמודה הראשונה של המטריצה אז נקבל שכל השורות המתחילות בתו מסוים יהיו מסודרות בסדר לקסיקוגרפי.
פענוח:
תחילה נשחזר את העמודה הראשונה ע"י כך שנמיין את העמודה האחרונה שקיבלנו.
כעת, נגדיר מערך שיפנה בין תו בעמודה הראשונה לתו המתאים לו בעמודה האחרונה שקיבלנו.
כאן אנו מסתמכים על התכונה לעיל שהשורות עם אותם תווים מסודרות בסדר לקסיקוגרפי.
לדוגמה, עבור הקלט BWT(T)=(pssmipissii,4) נקבל:

	S
0	4
1	6
2	9
3	10
4	3
5	0
6	5
7	1
8	2
9	7
10	8

F	
0	i
1	i
2	i
3	i
4	m
5	p
6	p
7	s
8	s
9	s
10	s

L	
0	p
1	s
2	s
3	m
4	i
5	p
6	i
7	s
8	s
9	i
10	i

אלגוריתם פענוח (הוסבר לעיל פרט ללולאה עצמה) [בדוגמה לעיל הפענוח הוא mississippi]

```
BWT(L, index):
    F = sort(L)
    M = Index
    For (i=0; i < n; i++)
        T[i] = F[M]
        M = S[M]
```

Move To Front

הרעיון בקידוד זה הוא לקודד את האינדקס של כל תו ואז להעביר אותו לראש הרשימה.
היתרון בכך הוא שכאשר ישנם הרבה חזרות של תו אז נקבל רצפים של אינדקס 0 (כי התו עבר להתחלה) ובכללי נוכל לקבל אינדקסים קטנים.
לדוגמה, עבור אלף בית {d,e,h,l,o,r,w} אז כאשר נקודד את המילה helloworld נקבל: 2230461...
נשים לב כי בכל קריאה של תו סדר האלף בית משתנה.
שלב 1: {d,e,h,l,o,r,w}. שלב 2: {h,d,e,l,o,r,w} שלב 3: {e,h,d,l,o,r,w} שלב 4: {l,e,h,d,o,r,w} וכו'

בחזרה לאלגוריתם BWT:
מכיוון שאלגוריתם BWT פולט טקסט שהוא "כמעט ממוין" נוכל לדחוס הודעה בצורה הבאה:

- נפעיל על ההודעה את אלגוריתם BWT
- לאחר מכן נפעיל על העמודה האחרונה (שאנו מעבירים) את אלגוריתם Move To Front
- על הפלט של Move To Front נפעיל דחיסה סטטיסטית הופמן/אליאס/RunLength/אריתמטי
- לבסוף נוכל להעביר את האינדקס שהאלגוריתם BWT פלט + הקידוד (הסופי) של העמודה

PPM (+EP)

הוכחנו בהרצאה כי אנטרופיה מותנת קטנה-שווה לאנטרופיה רגילה.
כלומר, $H(X|Y) \leq H(X)$
הסתברות של תו בהקשר של תו אחר: P(x|s)=c(x|s)/c(x) כאשר c(x) הוא מספר ההופעות של x.
לדוגמה, אם בטקסט ישנם 12 הופעות של e לבד וגם 7 הופעות של e **לאחר th** אז p(e|th)=7/12.
נגדיר k-max = אורך המחרוזות המקסימלי עבורם בודקים הקשרים.
נרצה להימנע מהסתברות 0 כי אז נצטרך אינסוף ביטים (חישוב הביטים הוא ע"י לוג).
לכן, נוסיף לטבלה תו $ שערכו יהיה מספר האלף בית באותו הקשר ונקודד אותו על מנת לרדת רמה.
אם הגענו לרמה 1- זה אומר שצריכים לקרוא תו חדש ולכן אנו מצפים לקידוד שלו.

לדוגמה, נניח אלף בית {a,b,$} התו $ ישמש כEOF. נקודד (עם k-max=2) את ההודעה aabb כך:
תווי הקידוד הסופיים הינם: $$$a$$a$$$b$$b
נשים לב כי עבור קידוד של תו חדש ההסתברות היא $1/3$ כי ישנם 3 תווים באלף בית.
עבור קידוד של תו קיים ההסתברות לקידוד תהיה אחידה על פני יחס ההופעות של תו מתוך ההופעות של שאר התווים באותה רמה.
בכל שלב נסמן בסוגריים מה ההסתברות לקידוד התו. מספר הסיביות הדרוש יהיה $\left\lceil \log_2 \frac{1}{p} \right\rceil$.
חשוב לזכור: תמיד נעדכן את כל הרמות לאחר קריאת תו.

<u>שלב 1</u>: [נקודד את a].
הסתברויות הקידוד לאחר ההוספה: (1)(1)$(1)a(1/3). הטבלה לאחר העדכון:

-1	empty	1	2
	a=1 $=1		

<u>שלב 2</u>: [נקודד את a].
הסתברויות הקידוד לאחר ההוספה: (1)(1)a(1/2). הטבלה לאחר העדכון:

-1	empty	1		2
	a=2 $=1	a	a = 1 $ = 1	

<u>שלב 3</u>: [נקודד את b]. b לא מופיע בהקשר aa ולא בהקשר a.
הסתברויות הקידוד לאחר ההוספה: (1)(1/2)$(1/2)b(1/3). הטבלה לאחר העדכון:

-1	empty	1		2	
	a=2 b=1 $=2	a	a = 1 b = 1 $ = 2	aa	b = 1 $ = 1

שלב 4: [נקודד את b]. b לא מופיע בהקשר ab ולא בהקשר b.
הסתברויות הקידוד לאחר ההוספה: $(1/5)b(1)$(1)$. הטבלה לאחר העדכון:

-1	empty	1		2	
	a=2 b=2 $=2	a	a = 1 b = 1 $ = 2	aa	b = 1 $ = 1
		b	b = 1 $ = 1	ab	b = 1 $ = 1

Exclusion Principle (EP):
לפעמים אנו יכולים להגדיל את ההסתברות של תו מסוים ברמה k כאשר $0 \le k < kmax$.
מצב זה קורה כאשר אנו מגיעים לרמה מסוימת ואנו יודעים בוודאות שהתו שאנו רוצים לקודד אנו תו מסוים. ולכן אפשר להתעלם מהתו האחר (ואז ההסתברות תגדל).
לדוגמה, אם נביט למעלה בטבלה של שלב 4 אז אם היינו רוצים לקודד את a במחרוזת bba אז מכיוון שהתו a אינו מופיע בהקשר bb וגם לא בהקשר b אז ברמה 0 המפענח יסיק את המסקנה הבאה:
התו שאנו רוצים לקודד (a) בהכרח אינו b כי אם הוא היה b אז כבר ברמה 1 יכולנו לקודד את b בהקשר b. לכן, ברמה 0 נתעלם מb=2 ונקבל a=2,b=0,$=2.
אלגוריתם לחישוב הסתברויות ברמה מסוימת:

```
estimate_prob(k) {
    sum=count[$];
    for (i=0; i<=|Σ|; i++)
        if k==kmax
            count[i]=count[a_i | previous k symbols]
        else if (k>=0)
            count[i]=(count[a_i | previous k+1 symbols]==0)*count[a_i | previous k symbols]
        else
            count[i]=(count[a_i]==0)
        Sum+=count[i]
    for (i=1; i<=|Σ|; i++)
        p(encode a_i) = count[i]/sum
    p(encode $) = count[$]/sum
}
```

Grammer Compression
מתקשר לדחיסות מילוניות.
הדחיסה מתבצעת ע"י גזירות (כללי יצירה) כמו באוטמטים. לדוגמה $A \to a$
המשתנים הם אותיות גדולות. הטרמינלים אותיות קטנות.
הדקדוק חייב להיות דטרמיניסטי (אין הסתעפויות בגזירה) וגם אסור שיהיה מעגל בגזירות.

Sequiter:
קורא תווים משמאל לימין ומעבד.
חוקים:
- תווים עוקבים לא יופיעו יותר מפעם אחת
- חוק צריך להופיע לפחות פעמיים

אלגוריתם:
כל עוד לא הגענו אל EOF:
- אם ישנם 2 תווים צמודים שחוזרים על עצמם – צור כלל חדש
- אם ישנו חוק שמשתמשים בו פעם אחת – הסר אותו והכנס את התוכן למופע היחיד שלו

דוגמה לקידוד המחרוזת abcdbcabcdbc:

Step 1	Step 2	Step 3	Step 4	Step 5
abcdbcabcdbc	**abcdbc**abcdbc	**abcdbca**bcdbc	**abcdbcab**cdbc	**abcdbcabc**dbc
$S \to abcdb$	$S \to aAdA$ $A \to bc$	$S \to aAdAa$ $A \to bc$	$S \to aAdAab$ $A \to bc$	$S \to aAdAa\boldsymbol{bc}$ $A \to \boldsymbol{bc}$
Step 6	Step 7	Step 8	Step 9	Step 10
Enforce rule 1	Enforce rule 1	**abcdbcabcd**bc	Enforce rule 1	Enforce rule 2
$S \to \boldsymbol{aA}dA\boldsymbol{aA}$ $A \to bc$	$S \to \boldsymbol{B}dA\boldsymbol{B}$ $A \to bc$ $B \to \boldsymbol{aA}$	$S \to \boldsymbol{Bd}A\boldsymbol{Bd}$ $A \to bc$ $B \to aA$	$S \to \boldsymbol{C}A\boldsymbol{C}$ $A \to bc$ $B \to aA$ $C \to \boldsymbol{Bd}$	$S \to \boldsymbol{C}A\boldsymbol{C}$ $A \to bc$ $C \to aAd$

Re-Pair

אלגוריתם:

- מצא זוג תווים הכי נפוץ. לדוגמה, ab.
- צור כלל. לדוגמה, $A \to ab$.
- החלף את כל מופעי הזוג בכלל שיצרת. לפי הדוגמה הנ"ל החלף את מופעי הזוג בכלל A.
- חזור ל1 כל עוד מצאת זוג שמופיע **יותר** מפעם אחת (בחר את הכי נפוץ).