סיכום אלגורתמים – דחיסת נתונים:

הגדרות וסימונים:

<u>אלף בית בקובץ מקור</u> - $S = [s_1, s_2, \dots, s_n]$.
<u>הסתברויות של האלפבית</u> - $P = [p_1, p_2, \dots, p_n]$, כאשר $p_1 \geq p_2 \geq \dots \geq p_n$ ובנוסף $\sum_{i=1}^{n} p_i = 1$.
<u>קידודי המילים</u> - $C = [c_1, c_2, \dots, c_n]$.
<u>אורכי הקידודים</u> - $|C| = [|c_1|, |c_2|, \dots, |c_n|]$.
<u>אורך מילת קוד ממוצעת (Excepted codeword length)</u> - $E(C,P) = \sum_{i=1}^{n} p_i * |c_i|$.

<u>קוד חסר רישות (prefix free)</u> – אף מילת קוד היא לא תחילית של מילת קוד אחרת.
<u>קוד UD</u> – כל מחרוזת של קידודי מילים ניתנת לפיענוח בצורה יחידה.
<u>קוד שלם</u> – כל מחרוזת קידודים אינסופית למחיצה ניתנת לפיענוח בצורה ייחודית.
<u>קוד מיידי</u> – מחרוזת קידודים שהמפענח יודע לפענח גל תו ברגע שהוא סיים לקרוא את מילת הקוד שלו.
<u>קוד חסר יתרות (Minimum Redundancy Code)</u> – עבור הסתברויות נתונות, אין קידוד אחר שמביא לexpected codeword length קטן יותר, פורמלית: יהי $E(C,P)$ אורך מילת קוד ממוצעת עבור קידוד C, C ייקרא קוד חסר יתרות עבור סדרת התפלגויות P אם $E(C,P) \leq E(C',P)$ לכל קוד חסר רישות C'.
<u>אינפורמציה</u> – כמות האינפורמציה המכולת בתו s_i עם הסתברות p_i היא $I(s_i) = -\log(p_i)$, קוד יכול להיות מתוכנן כך שמילת הקוד של s_i מורכבת מ$I(s_i)$ ביטים.
<u>אנטרופיה</u> - $H(P) = -\sum_{i=1}^{n} p_i * \log(p_i)$, ובנוסף לכל קידוד C מתקיים $H(P) \leq E(C,P)$.
<u>אי שוויון קראפט</u> – עבור קוד C בעל n תווים, אם C הוא UD אז מתקיים: $K(C) = \sum_{i=1}^{n} 2^{-|c_i|} \leq 1$.
<u>קידוד Fixed length</u> – קידוד תווים שבו כל קידוד הוא עם אורך זהה, למשל קידוד ASCII, מייצג כל תו ב8 ביטים.

משפטים:

קוד UD אינו בהכרח מיידי, יכול להיות קוד UD שמפוענח בצורה יחידה רק אחרי שסיימנו לקרוא את כל המחרוזת.
קוד מיידי אמ״מ קוד חסר רישות.
אם $K(C) \leq 1$ עבור קידוד C, אז קיים קוד C' כך ש$E(C,P) = E(C',P)$ ובנוסף $|C'| = |C|$ וC' הוא קוד חסר רישות.
אם $K(C) > 1$ אזי הקוד אעם האורכים הנתונים לא יכול להיות חסר רישות.
קוד מיידי עם מילות קוד באורכים $l_1 \leq l_2 \leq \dots \leq l_n$ קיים אמ״מ $\sum_{i=1}^{n} 2^{-l_i} \leq 1$.
תנאי הכרחי עבור קוד להיות UD הוא שמתקיים $\sum_{i=1}^{n} 2^{-l_i} \leq 1$.
קוד הוא שלם אם $\sum_{i=1}^{n} 2^{-l_i} = 1$.

אלגוריתם לבדיקת UD:

1. בנה רשימה של כל מילות הקוד.
2. אם קיימת מילת קוד שהיא תחילית של מילת קוד אחרת, תוסיף את הdangling suffix לרשימה (במידה ואין ברשימה מחרוזת זהה) עד ש:
   a. נקבל dangling suffix שהוא מילת קוד מקורית – החזר שהקוד אינו UD.
   b. אין יותר dangling suffix ייחודיים (שלא מופיעים כבר ברשימה) – החזר שהקוד הוא UD.

- Codewords {0,01,11}
- 0 is a prefix of 01 → dangling suffix is 1
- List - {0,01,11,1}
- 1 is a prefix of 11 → dangling suffix is 1 - already there
- → the code is UD

- Codewords {0,01,10}
- 0 is a prefix of 01 → dangling suffix is 1
- List - {0,01,10,1}
- 1 is a prefix of 10 → dangling suffix is 0 - which is an original codeword!
- → the code is not UD

מערכת דחיסה סטטית (מודל מסדר 0):
ההסתברויות לתו אינן תלויות בהודעה שיש לדחוס (נניח בASCII אם בהודעה אין a, מבחינתנו ההסברות לראות a היא עדיין 1/256). לכן, $p_i = \frac{1}{256}$, $H(P) = -\sum_{i=1}^{256} \frac{1}{256} * \log\left(\frac{1}{256}\right) = 8.0$.

מערכת דחיסה סמי-סטטית (מודל מסדר 0):
נבחן את ההודעה כדי להבין מי הם התווים שמופיעים בה – במערכות כאלו נדרש *prelude* להודעה שמכיל תיאור של התווים הנ״ל.
למשל עבור ההודעה *"Bring me my bow of burning gold!"* נגדיר $p_i = 1/25$ למרות שהתו *b* מופיע יותר מפעם אחת, כלומר נברר כמה תווים שונים יש וניתן להם התפלגות אחידה. מכאן $H(P) = -\sum_{i=1}^{25} \frac{1}{25} * \log\left(\frac{1}{25}\right) = 4.64$.
מה מכיל ה*prelude*?
- 8 ביטים של גודל האלפבית (מספר בין 1 ל256 ולכן במקרה הגרוע נצטרך 8 ביטים).
- $8 * n$ ביטים עבור התיאור של כל אחד מהתווים ($n = 25$ בדוגמה שלנו).

אורך מילת הקוד הממוצעת הוא בסה״כ $4.64 + \frac{208}{|message|}$.

מערכת דחיסה סמי-סטטית עם הסתברויות עצמיות (מודל מסדר 0):
במקרה זה נגדיר $p_i = \frac{v_i}{m}$ כאשר v_i הוא מספר המופעים של s_i בהודעה וm זה גודל הטקסט.
מה מכיל ה*prelude*?
- 8 ביטים של גודל האלפבית (מספר בין 1 ל256 ולכן במקרה הגרוע נצטרך 8 ביטים).
- $8 * n$ ביטים עבור התיאור של כל אחד מהתווים ($n = 25$ בדוגמה שלנו).
- $4 * n$ ביטים עבור תיאור ההסתברויות לכל תו.

מודל מסדר 1:
ההסתברות לכל תו היא בהקשר של התו הקודם.
למשל: ההסתברות שנראה *i* אחרי *n* היא 5/8.

קוד *Unary* (סוג של קוד סטטי):
נייצג את המספר x ע״י $x - 1$ ביטים של "1" וביט של "0" בסוף.
<u>יתרון</u>: קוד אינסופי (ללא דרישה שגודל האלפבית יהיה ידוע מראש).

Binary	*Unary*	*Decimal*
01	*0*	***1***
10	*10*	***2***
11	*110*	***3***
100	*1110*	***4***

קוד בינארי (סוג של קוד סטטי):
<u>קידוד בינארי פשוט</u> - כל תו מקבל מילת קוד באורך $\lceil \log_2 n \rceil$ ביטים (כאשר *n* זה מספר המילים).
<u>קידוד בינארי מינימלי</u> – עבור אלפבית בגודל *n*, קוד מינימלי מכיל $2^{\lceil \log_2 n \rceil} - n$ מילות קוד המורכבות מ$\lfloor \log_2 n \rfloor$ ביטים, ושאר $2n - 2^{\lceil \log_2 n \rceil}$ מילות הקוד יהיו מורכבות מ$\lceil \log_2 n \rceil$ ביטים.
לדוגמה: עבור *n=5, S=[1,2,3,4,5]* ולכן נקבל 3 מילות קוד באורך 2 ו2 מילות קוד באורך 3: $C = [00,01,10,110,111]$.

קודי *Elias* – C_γ:
מילת הקוד של *x* היא באורך $O(\log(x))$ ביטים.
הקידוד מורכב מ2 חלקים:
- חלק ראשון – קידוד *Unary* עבור מספר הביטים של *x* בקידוד בינארי:
  $Unary(num\ of\ bits\ in\ binary\ representation\ of\ x)$.
- חלק שני – הקידוד הבינארי של *x* ללא ה1 המוביל.

Decimal	Binary	C_γ		
		First part	Second part	Total
1	1	0	-	0
2	10	10	0	100
3	11	10	1	101
4	100	110	00	11000
5	101	110	01	11001

קודי Elias – C_δ:

הקידוד מורכב מ2 חלקים:

- חלק ראשון – קידוד C_γ עבור מספר הביטים של *x* בקידוד בינארי:
  $Unary(num\ of\ bits\ in\ binary\ representation\ of\ x)$.
- חלק שני – הקידוד הבינארי של *x* ללא ה1 המוביל.

Decimal	Binary	C_δ		
		First part	Second part	Total
1	1	0	-	0
2	10	100	0	1000
3	11	100	1	1001
4	100	101	00	10100
5	101	101	01	10101

<u>הערה</u>: C_γ ארוך יותר מקידוד *Unary* רק עבור *x=2,4*.

C_δ ארוך יותר מC_γ רק עבור *x=2,3,8,...,15*.

עבור ערכים גבוהים יותר קודי *Alias* קצרים יותר אקספוננציאלית מקידוד *Unary*.

Golomb Code:

```
Golomb_encode(x,b){
    q←(x-1) div b;
    r←x-q·b;
    Unary_encode(q+1);
    Minimal_binary_encode(r,b);
}
Golomb_decode(b){
    q←Unary_decode()-1;
    r←Minimal_binary_decode(b);
    return r+q·b;
}
```

קוד עם "דליי" בגודל קבוע *b*.

<u>דוגמה לקידוד</u>: נרצה לקודד את $x = 8$, כאשר $b = 5$.

כלומר, בכל ריישא של הקוד ה*Unary* יש 5 מילות קוד.

ו$x = 8$, כלומר נרצה לקודד את מילת הקוד השמינית.

$q = (8-1)div\ 5 = 1$

$r = 8 - 1 * 5 = 3$

$Unary(1+1) = 10$

$MBE(3{,}5) = 10$ – כלומר מילת הקוד השלישית באלפבית בגודל 5 *(0,01,**10**,110,111)*.

נקבל $C_8 = 1010$.

<u>דוגמה לפיענוח:</u>

$q = Unary_{decode}(10) - 1 = 2 - 1 = 1$: המפענח יקרא את החלק ה*Unary* של המילה (עד ל0 מפריד) ויפענח.

$r = MBD(10, 5) = 3$: שאר המילה היא *10*, וזוהי המילה השלישית באלפבית מגודל 5.

$return\ r + qb = 3 + 1 * 5 = 8$.

:Rice Code

מקרה פרטי של *Golomb*, כל דלי הוא בגודל חזקה של 2 : $b = 2^k$.

במקרה זה, החלק השני של הקידוד (ה*MBE*) מתלכד עם קידוד בינארי פשוט.

בחלק הראשון נעשה *shift right* ל$(x-1)$ בk ביטים ונוסיף 1 (פעולת חיבור) ואז מקודדים ב*Unary*.

דוגמה:

לפי *Golomb* כאשר $k = 4 = 2^2$, $x = 8$.

$$q = (8-1)div\ 4 = 1$$
$$r = 8 - 1 * 4 = 4$$
$$Unary(1+1) = 10$$
$$MBE(4,4) = 11$$
$$X_{Golomb-4} = 1011$$

לפי *Rice* כאשר $k = 4 = 2^2$, $x = 8$.

עבור החלק הראשון: $x = 8 \xrightarrow{x-1} 7 \xrightarrow{(7)_2} 111 \xrightarrow{shift\ right\ k=2\ bits} 1 \xrightarrow{add\ 1} 2 \xrightarrow{Unary(2)} 10$

עבור החלק השני: במקום לעשות *MBE* ניקח את $k = 2$ הביטים הנמוכים ביותר של $(x-1)$, אותם ביטים שזרקנו ב*shift right* ← *11*.

סה״כ קיבלנו: *1011*.

:Fibonaccci Code

בדיוק כמו בקידוד בינארי ניתן לקודד מספרים בפיבונאצ׳י כאשר הבסיסים הם *...1,2,3,5,8,13*:

למשל: $101001 = 19$ כיוון ש$19 = \mathbf{13} * 1 + 8 * 0 + \mathbf{5} * 1 + 3 * 0 + 2 * 0 + \mathbf{1} * 1$.

התכונה המיוחדת שהקידוד מקיים הוא שאף פעם לא יהיו לנו 2 אחדות צמודות.

לכן נוסיף 1 בהתחלה ונהפוך את הקידוד וכך נקבל קוד מיידי.

$$(19)_{fib-code} \rightarrow 101001 \xrightarrow{add\ front\ 1} \mathbf{1}101001 \xrightarrow{reverse} 100101\mathbf{1}$$

<u>תכונות</u>: קוד חסר רישות, קוד מיידי, יש F_k מילות קוד באורך $k-1$ ביטים.

:Fibonaccci2 Code

ב*Fib1* אין מילת קוד באורך 1 (הקידוד של המילה הראשונה (1) הוא 11) ולכן הציעו שינוי קטן.

האלגוריתם:

1. עבור כל מילת קוד ב*Fib1* נמחק את ה1 הימני ביותר.
2. מחק את כל מילות הקוד ב*Fib1* שמתחילות ב0.

אלגוריתם זהה:

1. עבור כל מילת קוד ב*Fib1* נמחק את ה1 הימני ביותר.
2. הוסף תחילית 10 לכל מילת קוד.
3. קבע את 1 להיות מילת הקוד הראשונה.

decimal	***Fib1***	*Delete left most 1*	*Add 10 prefix*	*Fix 1 as first cw + dropdown all cw*	***Fib2***
1	***11***	*1*	*101*	*1*	***1***
2	***011***	*01*	*1001*	*101*	***101***
3	***0011***	*001*	*10001*	*1001*	***1001***
4	***1011***	*101*	*10101*	*10001*	***10001***
5	***00011***	*0001*	*100001*	*10101*	***10101***
6	***10011***	*1001*	*101001*	*100001*	***100001***
7	***01011***	*0101*	*100101*	*101001*	***101001***

<u>תכונות</u>: יש מילת קוד אחת באורך 1, יש F_{k-2} מילות קוד באורך *k*.

:Shanon Code

מספר הביטים של כל מילת קוד c_i הוא $\left\lceil \log_2 \frac{1}{p_i} \right\rceil$.

זה מבטיח לנו אורך מילת קוד ממוצע שהוא בין האנטרופיה לבין האנטרופיה+1.

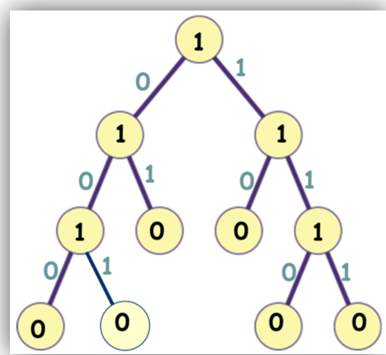
קידוד של עץ בינארי:

כדי להעביר את הקוד למפענח לא תמיד נצטרך להעביר את כל ההסתברויות, אפשר פשוט להעביר את העץ עצמו.

עבור עץ עם n עלים, נייצג ב1 כל צומת פנימית וב0 עלה.

נשים לב שיש לנו n-1 אפסים וn אחדות ולכן מספר הביטים שנצטרך הוא 2n-1.

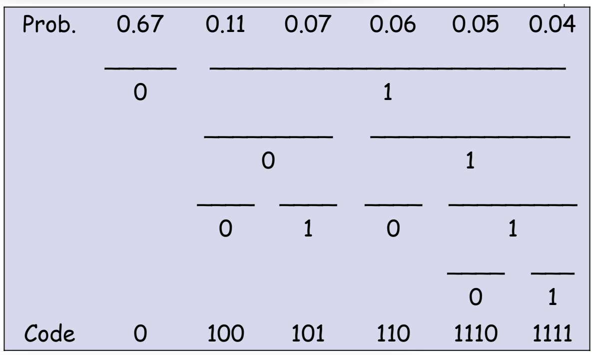
<u>לדוגמה</u>: 0000 1001 11 1 מייצג את העץ הבא:



:Shanon-Fano Code

1. נסדר את התווים בסדר יורד לפי ההסתברויות.
2. כל עוד יש לנו יותר מתו אחד:
   a. נחלק את הקבוצה ל2 כך שסכום ההתפלגויות בכל קבוצה כמעט שווה.
   b. נקצה 1 עבור הקבוצה הראשונה ו0 עבור הקבוצה השנייה.

<u>הערה</u>: הקידוד לא תמיד יעיל.



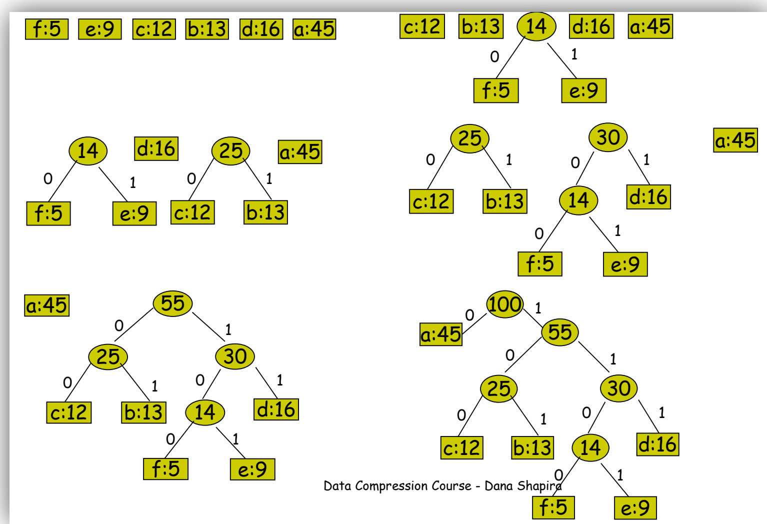
:($O(n * log_2(n))$ זמן ריצה) *Huffman Encoding*

1. נסדר את התווים בסדר יורד לפי ההסתברויות.
2. כל עוד יש יותר מתו אחד:

נבצע איחוד ל2 התווים עם ההסתברויות הנמוכות ביותר תחת שורש אחד ונקצה 0 עבור הבן השמאלי ו1 עבור הבן הימני ונוסיף את סכום ההסתברויות (שורש העץ) לקבוצה.

```
HUFFMAN(Σ)
n ←|Σ|
Q ← Σ
for i ←1 to n - 1
    do  ALLOCATE-NODE(z)
        left[z] ← x ← EXTRACT-MIN(Q)
        right[z] ← y ← EXTRACT-MIN(Q)
        w[z] ←w[x] + w[y]
        INSERT(Q, z)
return EXTRACT-MIN(Q)
```

<u>דוגמה:</u>



<u>תכונות:</u>

בהינתן משקלים $w_1, w_2, \dots, w_n$ אלגוריתם Huffman מקצה קודים באורכים $l_1, l_2, \dots, l_n$ כך ש$\sum_{i=1}^{n} w_i l_i$ הוא מינימלי.

Huffman מייצר קוד שלם (אופטימלי) – קוד אופטימלי מיוצג תמיד ע״י עץ מלא (לכל צומת פנימית יש בדיוק 2 בנים).

אי שוויון קראפט מקיים עבורו שוויון $K(C) = 1$.

ניתן להחליף בין האפסים והאחדות בכל אחד מהבנים של הקודקודים הפנימיים וכך אפשר לייצר 2^{n-1} קידודים שונים.

Canonical Huffman:

בקידוד האפמן קנוני, נעשה שימוש באורכי הסיביות של קודי האפמן הסטנדרטיים שנוצרו עבור כל סמל.
הסמלים ממוינים תחילה לפי אורכי הסיביות שלהם בסדר עולה ולאחר מכן עבור כל אורך סיביות, הם ממוינים באופן לקסיקוגרפי.
הסמל הראשון מקבל קוד המכיל את כל האפסים ובאורך זהה לזה של אורך הביט המקורי.
עבור הסמלים הבאים, אם לסמל יש אורך סיביות השווה לזה של הסמל הקודם, הקוד של הסמל הקודם מוגדל באחד ומוקצה לסמל הנוכחי.
אחרת, אם לסמל יש אורך סיביות גדול מזה של הסמל הקודם, לאחר הגדלה של הקוד של הסמל הקודם יתווספו אפסים עד שהאורך ישתווה לאורך הסיביות של הסמל הנוכחי והקוד יוקצה לסמל הנוכחי.
תהליך זה נמשך עבור שאר הסמלים.

Character	Frequency
a	10
b	1
c	15
d	7

Character	Huffman Codes	Bit lengths
a	11	2
b	100	3
c	0	1
d	101	3

Character	Bit lengths
c	1
a	2
b	3
d	3

Character	Canonical Huffman Codes
c	0
a	10
b	110
d	111

<u>הסבר לדוגמה:</u>
בשלב בראשון קיבלנו את התווים עם ההסתברויות.
בשלב השני הפעלנו אלגוריתם Huffman כדי לקבל את אורכי הקידודים לכל תו.
בשלב השלישי נסדר את התווים בסדר עולה לפי אורכי הקידודים שקיבלנו מHuffman.
בשלב הרביעי תחיל להקצות קידודים:

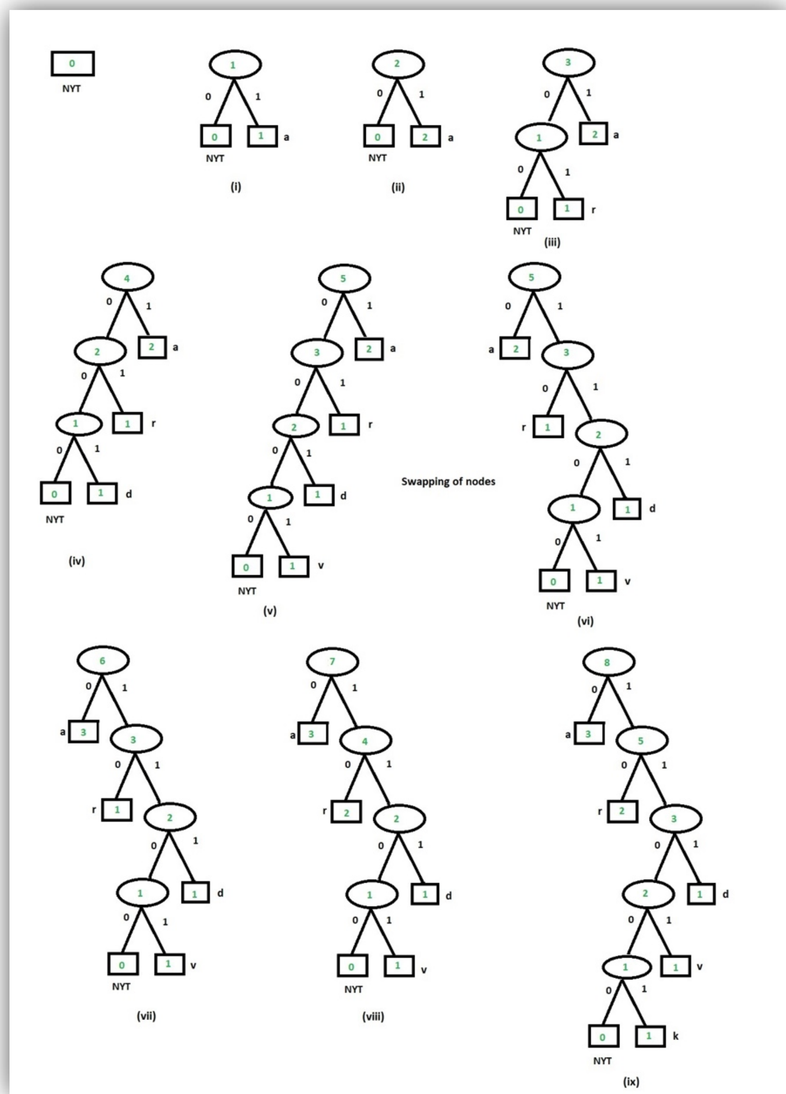
- c הוא התו הראשון, צריך להקצות לו קידוד באורך 1 – לפי האלגוריתם ניתן לו אפסים בכל הביטים, הקידוד הוא בגודל 1 ולכן נקצה 0.
- נעבור לa, a לא באותו אורך כמו c ולכן נוסיף 1 $(0+1=1)$ ונרפד באפסים מימין עד שנגיע לאורך הרצוי, הקידוד הוא בגודל 2 ולכן נקצה 10.
- נעבור לb, b הוא לא באותו אורך כמו a ולכן נוסיף 1 $(10+1=11)$ ונרפד באפסים מימין עד שנגיע לאורך הרצוי, הקידוד הוא בגודל 3 ולכן נקצה 110.
- נעבור לd, d הוא באותו אורך כמו b, כל מה שנצטרך לעשות הוא להוסיף 1 $(110+1=111)$ ולכן נקצה 111.

D-ary Huffman Code:

במקום Huffman בינארי נבצע Huffman עם D בנים.
בדיוק אותו אלגוריתם רק שבכל פעם נאחד את הD תווים עם ההתפלגויות הכי נמוכות.
הבעיה: אם מספר התווים הוא לא כפולה של D, לשורש יהיו פחות מD בנים ולכן בקוד שנקבל יהיו ביטים מיותרים.
פתרון: נוסיף x קודקודים עם הסתברות 0 לרשימה ההתחלתית כך שיתקיים $x+n=1+(D-1)k$.

Dynamic (adaptive) Huffman:

נבנה את העץ בריצה אחת, כלומר בלי לדעת מראש את ההסתברויות.



<u>דוגמה</u>: המחרוזת היא *"aardvark"*.

- נתחיל מקודקוד *NYT*.
- ראינו *a* – נוסיף אותו בתור אח של *NYT* ונבנה שורש מעליהם.
- ראינו *a* נוסף – נעדכן את ההסתברות של *a* מ1 ל2 ולכן נעדכן את השורש להיות *0+2*.
- ראינו *r* – נוסיף אותו בתור אח של *NYT* ונבנה שורש מעליהם (*1+0=1*), השורש החדש הוא האח החדש של *a*. נעדכן את השורש להיות *1+2*.
- ראינו *d* - נוסיף אותו בתור אח של *NYT* ונבנה שורש מעליהם (*1+0=1*), השורש החדש הוא האח החדש של *r*. נעדכן את האבא של *r* ושל השורש החדש להיות *1+1*, הוא האח של *a* ונעדכן את השורש הראשי.
- ראינו *v* - נוסיף אותו בתור אח של *NYT* ונבנה שורש מעליהם (*1+0=1*), השורש החדש הוא האח החדש של *d*. נעדכן את כל העץ כמו קודם.
- נשים לב שעד כה תמיד הבנים הימניים היו גדולים או שווים מהבנים השמאליים. ברגע שעקרון זה מופר נצטרך לתקן את העץ.
- נבצע החלפה בין כל האחים שהעיקרון הזה מופר אצלם.

תכונות אחים:

1. $w(v) = w(right(v)) + w(left(v))$.
2. הקודקודים יכולים להיות ממוספרים בסדר עולה לפי המשקלים כך ש2j-1 ו2j הם אחים.

עץ שמקיים את תכונת האחים אמ״מ הוא יכול להיות עץ Huffman.

Sekelton Tree:

מוטבציה – לא צריך לשמור את כל העץ (חוסך מקום).

צריך להשלים

Arithmetic Code:

השוואה בין קוד אריתמטי לקוד האפמן:

קוד האפמן:

- מחליף תו למילת קוד.
- צריך לקבל כקלט את ההסתברויות למופעים של התווים.
- קשה לו להסתגל לסטטיסטיקות משתנות.
- צריך לאחסן את טבלת מילות הקוד.
- מילת הקוד המינימלית היא באורך ביט 1.

קוד אריתמטי: לוקח לו יותר זמן לדחוס ולפרק ולכן על אף היתרונות שלו הזמן שמושקע בדחיסה לא שווה את זה.

- מחליף את כל המחרוזת במספר עשרוני יחיד (*floating-point number*).
- לא צריך לקבל את ההסתברויות.
- מסתגל בקלות.
- לא צריך לאחסן ולשלוח את טבלת הקידודים.
- אורכי מילות קוד שבריים.
- מגיע לאנטרופיה.

קידוד אריתמטי:

1. נגדיר *low=0, high=1*.
2. כל עוד נותרו תווים לקרוא:
   a. *range=high-low*.
   b. קריאת התו הבא.
   c. עדכון high לפי החסם עליון של התו.
   d. עדכון low לפי חסם תחתון של התו.
3. פלוט מספר כלשהו בטווח $[low, high)$.

```
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)
```

<u>דוגמה:</u> קלט – "ABAAAEAABA".

לבסוף נרצה לפלוט מספר בטווח $[0.46940461, 0.46944128)$.
נעביר את הlow ואת הhigh לייצוג בינארי:
$low = \mathbf{0.0111100000101}010111010$
$high = \mathbf{0.0111100000101}101010011$
נחזיר את המספר הבינארי הקטן ביותר שנמצא ביניהם:
$return\ \mathbf{0.0111100000101}100$

M[i]	Low	High	Range
-	0.0000	1.0000	1.0000
A	0.0000	0.67	0.67
B	0.4489	0.5226	0.0737
A	0.4489	0.498279	0.049379
A	0.4489	0.48198393	0.03308393
A	0.4489	0.47106623	0.02216623
E	0.46907127	0.47017958	0.00110831
A	0.46907127	0.46981384	0.00074257
A	0.46907127	0.46956879	0.00049752
B	0.46940461	0.46945934	0.00005473
A	0.46940461	0.46944128	0.00003667

S_i	Low bound	High bound
A	0.0	0.67
B	0.67	0.78
C	0.78	0.85
D	0.85	0.91
E	0.91	0.96
F	0.96	1.0

פיענוח קידוד אריתמטי:

קלט: מספר עשרוני.

1. נגדיר encoded ששווה למספר שקיבלנו מהמקודד.
2. כל עוד לא הגענו לEOF:
   a. מצא את התו s שנמצא בטווח של המספר.
   b. פלוט את s.
   c. עדכון את הrange להיות $range = high(s) - low(s)$.
   d. עדכון את הencoded להיות $encoded = \frac{encoded - low(s)}{range}$.

```
encoded ← Get (encoded number)
do{
  Find symbol whose range contains encoded
  Output the symbol
  range ← high(symbol) - low(symbol)
  encoded ← (encoded - low(symbol))/range
}until (EOF)
```

<u>דוגמה:</u>

בהינתן טבלת הlow והhigh של התווים נפענח את המספר העשרוני 0.109375.

- B נמצא בטווח ולכן נפלוט B.
  $range = 0.25 - 0 = 0.25$
  $encoded = \frac{0.109375 - 0}{0.25} = 0.4375$
- I נמצא בטווח ולכן נפלוט I.
  מעדכן את הencoded וכן הלאה...
  נעצור כאשר $encoded = 0$.

E = .109375
between 0.0, 0.25; output 'B'
E = (.109375 - 0.0) / 0.25 = .4375
.4375 between 0.25, 0.5; output 'I'
E = (.4375 - 0.25) / 0.25 = 0.75
0.75 between 0.5, 1.0; output 'L'
E = (0.75 - 0.5) / 0.5 = 0.5
0.5 between 0.5, 1.0; output 'L'
E = (0.5 - 0.5) / 0.5 = 0.0 -> STOP

Symbol	Low	High
B	0	0.25
I	0.25	0.50
L	0.50	1.00

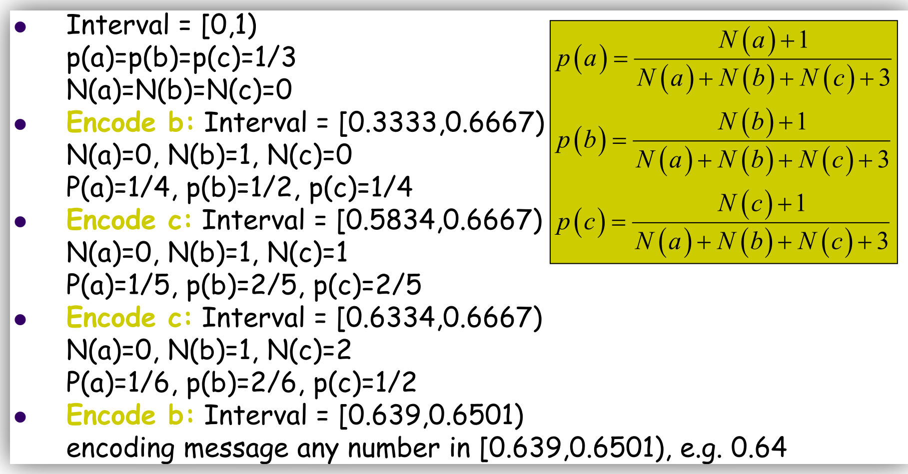
:Adaptive Arithmetic Coding

<u>דוגמה:</u>

קלט: אלפבית $\{a,b,c\}$, מחרוזת *bccb* לקידוד, *N(x)* – מספר המופעים של התו *x* עד נקודה נוכחית.

האלגוריתם משתמש בנוסחה:

$$p(x) = \frac{N(x)+1}{\sum_{i=1}^{n} \mathrm{N}(\mathrm{s_i}) + |\Sigma|}$$



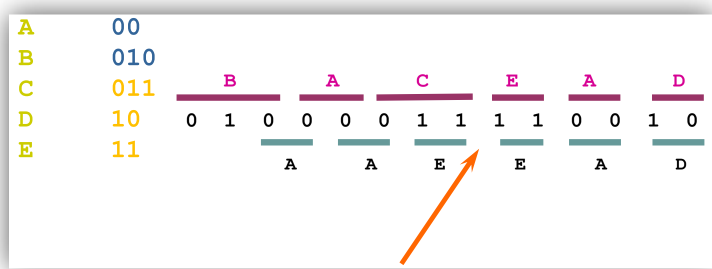
- מתחילים מאינטרוול שלם $[0,1)$.
  כאשר $N(a)=N(b)=N(C)=0$.
  ולכן $p(a)=p(b)=p(c)=1/3$.
- נקודד את התו *b*:
  b מוגדר בקטע *[1/3,2/3)* ולכן נעדכן את האינטרוול להיות $[0.33,0.66)$.
  עתה, $N(a)=0, N(b)=1, N(c)=0$.
  ולכן $p(a)=\frac{1}{4}, p(b)=\frac{1}{2}, p(c)=\frac{1}{4}$.
- נקודד את התו *c*:
  c מוגדר ברבע האחרון של האינטרוול.
  אם נחלק את הקטע $\left[\frac{4}{12},\frac{8}{12}\right)$ לרבעים נראה
  ש*a* מוגדר ברבע הראשון שזה $\left[\frac{4}{12},\frac{5}{12}\right) =$
  $[0.33,0.416)$.
  b מוגדר באופן זהה בחצי האמצעי שזה $\left[\frac{5}{12},\frac{7}{12}\right) = [0.416,0.583)$.
  c מוגדר באופן זהה ברבע האחרון של האינטרוול זה $\left[\frac{7}{12},\frac{8}{12}\right) = [0.583,0.667)$.
  לכן נעדכן את האינטרוול להיות $[0.583,0.667)$.
  עתה, $N(a)=0, N(b)=1, N(c)=1$ ולכן $p(a)=\frac{1}{5}, p(b)=\frac{2}{5}, p(c)=\frac{2}{5}$.
- נמשיך את שאר השלבים באופן זהה עד שנסיים לקודד את המחרוזת ולבסוף נפלוט מספר שנמצא בטווח של האינטרוול הסופי.

<u>חסרונות:</u>

1. דיוק – כיוון שעובדים עם אינטרוול יכול להיות שהמספר שנקודד יהיה מאוד ארוך וכידוע מחשבים לא מצליחים לדייק מספרים גדולים.
2. איטי יותר מ*Huffman*.
3. לא תומך בגישה אקראית.

סנכרון:

במידה ואיבדנו ביטים נרצה לדעת האם המפענח יצליח לחזור בשלב מסויים לתווים הנכונים (כלומר יצליח לפענח החל משלב מסויים למרות איבוד הביטים).
כאשר יש קידוד למילה מסויימת שהוא סייפא של מילת קוד אחרת שנמצאת בחלק שאיבד את הביטים, נצליח להסתנכרן ברגע שתופיע מילת הקוד שהיא סייפא.
בדוגמה *E* היא סייפא של *C* ולכן הסנכרון קורה החל מקריאת *E*.
אם נשתמש ב*Fixed length codes* או ב*Affix codes* לא יהיה סנכרון.



מילון סטטי *vs* מילון אדפטיבי:

גישה סטטית (*Static approach*):

- המילון נבנה לפני הקידוד.
- נדרש ידע מוקדם על המקור.

גישה אדפטיבית (*Adaptive approach*): *LZSS*

- המילון נבנה דינמית תוך כדי הקידוד.

גישה סטטיסטית (*Statistical approach*):

- ננסה לחזות את התו הבא.

גישת ההחלפה(*Substitutional approach*):

- החלפת בלוקים של טקסט בהפניות למופעים קודמים של טקסט זהה.

Dictionary

c_i	symbol
10	a
1111110	b
111110	c
0	aa
11111110	aaaa
11111111	ab
110	baa
1110	bccb
11110	bccba

בהינתן מילון סטטי ומחרוזת, איך נשבור אותה לבלוקים שנקודד?

נניח שקיבלנו מחרוזת *"aaabccbaaaa"*.

- <u>גישה חמדנית (*Greedy*):</u> מסתכלת על המחרוזת מההתחלה ותיקח בכל פעם את המחרוזת הארוכה ביותר שניתן לתפוס.
  בדוגמה: *aa-ab-c-c-b-aa-aa* (*30 bits*).
- <u>גישת ה*longest fragment*:</u> מסתכלת על כל המחרוזת ומוצאת את המחרוזת הארוכה ביותר שניתן לתפוס, מסתכלת שוב על המחרוזת שנותרה ומחפשת את תת המחרוזת הארוכה ביותר שניתן לתפוס וכן הלאה...
  בדוגמה:*aa-a-bccba-aa-a* (*11 bits*).
- גישת *Min-words*: נרצה למצוא את מספר המילים המינימליות שניתן לפרק את המחרוזת.
  בדוגמה: *aa-a-bccb-aaaa* (*15 bits*).
- אופטימלי: *aa-a-bccb-aa-aa* (*9 bits*).

***LZ77 Compression*:**

נחליף ביטוי בטקסט למצביע למילון כדי להשיג את הדחיסה.

Window	Look Ahead Buffer

חלון הטקסט (*window*): הקוד שקודדנו עד כה.

look ahead buffer: התווים שותר לקודד.

<u>הערה</u>: המילון כאן הוא מרומז, זה למעשה החלון.

פלט האלגוריתם יהיה שלשות כאשר כל שלשה מורכבת מ:

- *offset* – כמה אחורה צריך ללכת.
- *length* – אורך הביטוי שיש להעתיק.
- הביטוי עצמו.

<u>אלגוריתם קידוד:</u>

1. נגדיר מצביע *p* שיצביע למקום הראשון.
2. כל עוד יש עוד תווים בטקסט שצריך לקודד:
   a. חפש את המחרוזת הארוכה ביותר ב*look ahead* שקיימת ב*window*.
   b. נוציא את השלשה המתאימה.
   c. נקדם את הפוינטר שיצביע על המקום הבא.

```
1. p←1 // The next character to be coded
2. while there is text remaining to be
   coded{
   1. search for the longest match for S[p…]
      in S[p-W…p-1] suppose that the match occurs
      at position m with length ℓ
   2. Output the triple (p-m, ℓ,S[p+ℓ])
   3. p←p+ℓ+1
}
```

<u>בדוגמה</u> נרצה לקודד את המחרוזת הנ"ל.

כל שלשה בקידוד אומרת כמה ללכת אחורה? כמה להעתיק? ואיזה תו לשים בסוף?

למשל השלשה *(21,11,'v')* אומרת לך אחורה 21 תווים תעתיק 11 תווים ותוסיף בסוף *v*.

String:
A_walrus_in_Spain_is_a_walrus_in_vain.

Encoded String
(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,'.')

בפועל לא נשלח למפענח את השלשות בצורה כזו, נשלח 6 ביטים עבור ה*offset* (ה*offset* הגדול ביותר הוא 21 וזה דורש 6 ביטים), 4 ביטים עבור ה*length* (ה*length* הגדול ביותר הוא 11 וזה דורש 4 ביטים) ו8 ביטים עבור התו (*ASCII*) – סה"כ 18 ביטים לכל שלשה.

אלגוריתם פענוח:

1. נגדיר מצביע *p* שיצביע למקום הראשון.
2. לכל שלשה בקלט:
   a. לך אחורה *offset* צעדים.
   b. תעתיק *length* תווים ותוסיף את התו.
   c. נקדם את הפוינטר שיצביע על המקום הבא.

```
p←1 // The next character to be decoded
For each triple (f,ℓ,c) in the input{
   S[p…p+ℓ-1]←S[p-f…p-f+ℓ-1]
   S[p+ℓ]←c
   p←p+ℓ+1
}
```

בדוגמה המפענח מקבל את השלשות ומתחיל לפענח כל שלשה.
בדוגמה כל השלשות הראשונות הן עם *offset* 0 וגם *length* 0 ולכן המפענח פשוט ירשום את התווים *A_walrus*. לאחר מכן יפענח את השלשה *(7,1,'i')* יילך 7 תווים אחורה, יעתיק תו 1 ('_') וישים *'i'* בסוף.
סה"כ נקבל *A_walrus_i*. נמשיך ככה עד שייגמרו השלשות.

Encoded String
(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,'.')

String:
A_walrus_in_Spain_is_a_walrus_in_vain.

חסרונות של *LZ77*:

- צוואר בקבוק - בעת הקידוד, עליו לבצע השוואות של מחרוזות אל מול ה*look-ahead buffer* עבור כל מיקום בחלון הטקסט.
- שימוש בשלשה כאשר אין התאמה – כשלא נמצא התאמה ב*window* נכתוב (0,0,'c') וזה תוספת מיותרת של ביטים.

נניח שהחלון בגודל 4096 תווים וה look-ahead buffer בגודל 15 תווים. כדי לקודד את השלשה הנ"ל נצטרך 12 סיביות לoffset ($2^{12} = 4096$), 4 סיביות לlength ו8 תווים לתו עצמו. למעשה אנחנו מעבירים 24 סיביות במקום להעביר רק 8.

LZSS:

גרסה משופרת ל*LZ77*.
מאפשר לערבב מצביעים (שלשות) ותווים בקידוד.
חיפוש המחרוזות הזהות נעשה באמצעות בניית עץ מאוזן.
כדי שהמפענח ידע לפענח את הקידוד נוסיף ביט זיהוי לפני קידוד כל תו. במידה והקידוד הוא של תו נוסיף בהתחלה 0 ובמקרה וזה מצביע (שלשה) נוסיף בהתחלה 1.
לכן, במקרה שהחלון שלנו בגודל 4096 תווים וה look-ahead buffer בגודל 16 תווים, נצטרך 9 ביטים לקודד תו 0+character, ו17 ביטים לשלשה 1+offset(12bit)+length(4bit).

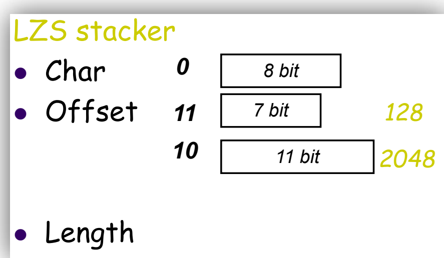
דוגמה:
נשים לב שלא משתלם לנו לכתוב שלשה עבור תו בודד אך לעיתים גם לא משתלם לנו לכתוב שלשה בשביל להעתיק תו או 2 כי הקידוד יכו להיות יותר ביטים מאשר אם נקודד את התווים בASCII ולא נשלח הצבעה.
לדוגמה לא משתלם לנו להעתיק את ה'_' השני ע"י מצביע לראשון כי זה ייקח יותר ביטים מאשר לכתוב את התו עצמו.
במידה וגודל החלון וה look-ahead הם כמו למעלה, העתקה של 2 ביטים תיקח לנו 18 ביטים אם נכתוב אותם כתווים בASCII ו17 ביטים עם מצביעים – לכן נבנה שלשה עבור העתקה של 2 תווים ומעלה.

String:
A_walrus_in_Spain_is_a_walrus_in_vain

Encoded String
A_walrus_in_Spa(6,3)is_a(21,11)v(20,3)

LZS:

חידוש ל*LZ77*.
נגדיר מספר אופציות לoffset, לפעמים נרצה ללכת הרבה תווים אחורה ולפעמים נרצה ללכת מעט תווים אחורה.
אין סיבה ששתי האופציות ייקחו אותם כמות תווים ולכן נוסיף flag שידע כמה הוא צריך לקרוא מהoffset.
אם אני רוצה ללכת מעט תווים אחורה נגביל את החלון ל7 ביטים ונוסיף flag של 10 כדי שהמפענח ידע כמה תווים לקרוא.
אם נרצה ללכת יותר תווים אחורה ניתן חלון בגודל 11 ביטים ונוסיף flag של 11.



ניתן להוסיף יותר גדלים (ולכן flagים יותר ארוכים) ולשחק עם גודלי הoffset.

:LZ78

בLZ77 מילון הביטויים הוגדר על ידי חלון קבוע של טקסט שנראה בעבר – window.
בLZ78 מילון הוא רשימה בלתי מוגבלת של ביטויים שנראו בעבר.
כל אסימון ב LZ78 מורכב מקוד שבוחר ביטוי נתון ותו בודד שאחריו הביטוי.
שלא כמו LZ77, אורך הביטוי אינו עובר מאחר שהמפענח יודע זאת.

<u>אלגוריתם קידוד:</u>

1. מתחילים ממילון ריק – בכניסה 0 יש את המילה הריקה ϵ.
2. מצא את ההתאמה הארוכה ביותר בין הlook-ahead לבין הכניסה הארוכה ביותר שקיימת במילון שפותחת בתו הראשון בlook-ahead.
3. פלוט את הזוג (id, c) כאשר $c = next\ char$.
4. אם $c = EOF$ עצור.
5. אחרת, הוסף למילון $(\max id + 1, T(id) * c)$.
6. חזור ל2.

<u>דוגמה:</u> $\Sigma = \{a, b, c, d\} \xrightarrow{binary} \{00{,}01{,}10{,}11\}$

- מתחילים ממילון ריק עם כניסה 0 שמכילה את המילה הריקה.
- ההתאמה הארוכה ביותר במילון היא המילה הריקה ולכן נפלוט את הזוג $(0, b)$.
ונוסיף למילון בכניסה ה1 את b.
- נקפוץ לאיטרציה הרביעית: עד כה קידדנו את bad.
מסתכלים בlook-ahead והמחרוזת הארוכה ביותר במילון שפותחת את המחרוזת ad היא a (כניסה 2).
לכן, נפלוט את הזוג $(2, d)$ ונוסיף למילון $(4, ad)$.

- **T = badadadabaab**

index	Phrase	Encoding	# of bits
0	ε		
1	b	(0,b)	0+2
2	a	(0,a)	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

<u>כמה ביטים זה לוקח?</u>
תחילה נציין שלא מעבירים את המילון כיוון שהמפענח ידע לבנות אותו לבד.
המפענח יודע שהכניסה ה0 היא המילה הריקה ולכן לא נצטרך לקודד אותו (ייקח לנו *0* ביטים).
בכניסה ה1 המפענח יודע שהכניסה שהאיבר הראשון בזוג הוא הכניסה ה0 ולכן לא נצטרך לקודד אותה, נצטרך לקודד רק את התו *b*, יש 4 תווים באלפבית ולכן הקידוד של *b* יעלה לנו 2 ביטים.
כרגע במילון של המקודד (ושל המפענח) יש רק 2 כניסות ולכן נוכל לציין את מספר הכניסה ע״י ביט בודד + 2 ביטים עבור התו עצמו בבינארי – סה״כ 3 ביטים.
עכשיו יש במילון 3 כניסות ולכן קידוד של כניסה יעלה לנו 2 ביטים ומסיבה זו הקידוד של הזוג הבא יעלה 4 ביטים, 2 עבור הכניסה ו2 עבור התו עצמו וכן הלאה.

בסופו של דבר נקבל את הקידוד שמופיע משמאל.
ניתן לראות את הקידוד בביטים.

$$\left\{\begin{array}{c} \underline{encoding:} \\ \underbrace{(0,b)}_{b}\underbrace{(0,a)}_{a}\underbrace{(0,d)}_{d}\underbrace{(2,d)}_{ad}\underbrace{(4,a)}_{ada}\underbrace{(1,a)}_{ba}\underbrace{(2,b)}_{ab} \\ \underline{in\ bits:} \\ \underbrace{(\varepsilon^*,01)}_{b}\underbrace{(0,00)}_{a}\underbrace{(00,11)}_{d}\underbrace{(10,11)}_{ad}\underbrace{(100,00)}_{ada}\underbrace{(001,00)}_{ba}\underbrace{(010,01)}_{ab} \end{array}\right\}$$

:LZW

<u>אלגוריתם קידוד:</u>

```
1. Dictionary ← single Characters
2. w ← first char of input
3. repeat{
   1. k ← next char
   2. if(EOF)
      1. output code(w)
   3. else if (w · k)∈ Dictionary
      1. w ← w · k
   4. else
      1. output code(w)
      2. Dictionary ← w · k
      3. w ← k
}
```

Data Compression Course - Dana Shapira

1. נאתחל את המילון עם כל התווים הבודדים שבאלפבית.
2. נאתחל את w להיות התו הראשון בטקסט.
3. כל עוד לא הגענו לסוף הטקסט:
   a. נאתחל את k להיות התו הבא.
   b. אם wk נמצא במילון: $w = wk$.
   c. אחרת:
      i. נפלוט את הקידוד של w.
      ii. נכניס למילון תחת כניסה חדשה את wk.
      iii. וw יקבל את התו הבא בטקסט - $w = k$.

<u>דוגמה:</u>

T=wabba_wabba_wabba_wabba_woo_woo

init ⇒

code	phrase
0	–
1	a
2	b
3	o
4	w

⇒

w	k	$\in$	$update\,(n, wk)$	out
w	a	×	$(5, wa)$	4
a	b	×	$(6, ab)$	1
b	b	×	$(7, bb)$	2
b	a	×	$(8, ba)$	2
a	_	×	$(9, a\_)$	1
_	w	×	$(10, \_w)$	0
w	a	✓	–	
wa	b	×	$(11, wab)$	5
b	b	✓	–	
bb	a	×	$(12, bba)$	7
⋮				

⇒

code	phrase
0	_
1	a
2	b
3	o
4	w
5	wa
6	ab
7	bb
8	ba
9	$a\_$
10	$\_w$
11	wab
12	bba
⋮	
15	

- נרצה לדחות את המחרוזת T.
- תחילה המילון ריק (כניסה 0 עם ביטוי ϵ).
- נאתחל את המילון באלפבית שלנו (כניסות 1-4 יקבלו את התווים a,b,o,w בהתאמה).
- נאתחל את w להיות האות הראשונה בטקסט - $w = w$.
  ואת k להיות האות הבאה – $k = a$.
- הביטוי wa לא מופיע במילון ולכן נפלוט 4 (הקידוד של w), ונכניס לכניסה 5 את הביטוי wa, ונגדיר את w להיות שווה לk (כלומר $w = a$).
- נמשיך באותה דרך עד שנגיע לאיטרציה השביעית.
- עתה, $w = w, k = a$.
- הביטוי wa כן מופיע בטבלה בכניסה 5 ולכן נעדכן את w: $w = wa$.
- k מקבל את התו הבא: $k = b$.
- הביטוי wab לא מופיע בטבלה ולכן נפלוט את הקידוד של $w = wa$ שהוא 5, נכניס את wab למילון לכניסה הבאה (כניסה 11) ונגדיר את w להיות שווה לk.
- נמשיך כך עד שנסיים לקרוא את כל הקובץ.

בסופו של דבר הקידוד שנקבל הוא *...41221057*.

```
1. Initialize table with single character strings
2. OLD = first input code
3. output translation of OLD
4. while not end of input stream{
   1. NEW = next input code
   2. if NEW is not in the string table
      1. S = translation of OLD
      2. S = S · C
   3. else
      1. S = translation of NEW
   4. output S
   5. C = first character of S
   6. Translation(OLD)· C to the string table
   7. OLD = NEW
 }
```

Data Compression Course - Dana Shapira

<u>אלגוריתם מפענח:</u>

1. נאתחל את המילון בכל התווים הבודדים שבאלפבית.
2. נגדיר את OLD להיות התו הראשון בקלט.
3. נפלוט את התרגום של OLD.
4. כל עוד לא הגענו לסוף הטקסט:
   a. נגדיר את NEW להיות הקידוד הבא.
   b. אם NEW לא מופיע במילון: נגדיר את S להיות שווה לפירוש של OLD משורשר עם C.
   c. אחרת, נגדיר את S להיות שווה לפירוש של NEW.
   d. פלוט את S.
   e. נגדיר את C להיות התו הראשון של S.
   f. נכניס בכניסה הבאה במילון את הפירוש של OLD משורשר עם C.
   g. נגדיר OLD=NEW.

<u>דוגמה:</u>
נניח שהמפענח קיבל את הקידוד הבא: 0,1,3,2,4,7,0,9,10,0.

$code$	$phrase$
0	a
1	b
2	c

$\Rightarrow$

old	new	$\in$	s	out	c	$update\left(n, T(old)\cdot c\right)$
0	$-$	✓	$-$	a	$-$	$-$
0	1	✓	b	b	b	$(4, T(0)\cdot b) = (3, ab)$
1	3	✓	ab	ab	a	$(4, T(1)\cdot a) = (4, ba)$
3	2	✓	c	c	c	$(5, T(3)\, c) = (5, abc)$
2	4	✓	ba	ba	b	$(6, T(2)\, b) = (6, cb)$
4	7	$\times$	$ba\cdot b$	bab	b	$(7, T(4)\, b) = (7, bab)$
7	0	✓	a	a	a	$(8, T(7)\, b) = (8, baba)$
0	9	$\times$	$a\cdot a$	aa	a	$(9, T(0)\, a) = (9, aa)$
9	10	$\times$	$aa\cdot a$	aaa	a	$(10, aaa)$
10	0	✓	a	a	a	$(11, aaaa)$

$\Rightarrow$

$code$	$phrase$
0	a
1	b
2	c
3	ab
4	ba
5	abc
6	cb
7	bab
8	$baba$
9	aa
10	aaa
11	$aaaa$

$$\text{deocded} = \text{a b ab c ba bab a aa aaa a}$$

Run-Length Codes:
Runs – בלוקים של תווים שחוזרים על עצמם ברצף.
נייצג את המחרוזת ע״י האורכים של תווים רצופים.
<u>דוגמה</u>: בהינתן המחרוזת: acccbbaaabb נקודד אותה ל:$(a,1)(c,3)(b,2)(a,3)(b,2)$.

Binary Run-Length Codes:
במידה והמחרוזת היא בינארית, יש רק שני תווים יהיה לנו קל יותר לקודד.
<u>דוגמה</u>: בהינתן המחרוזת *0111100111000000101011* נקודד אותה ל: *1423511112*.
<u>הערה</u>: נניח שתמיד נפתח מחרוזת ב0, במידה והמחרוזת מתחילה ב1 אז יש רצף של אפס אפסים בהתחלה ולכן הקידוד של התו הראשון יהיה 0. <u>דוגמה</u>: בהינתן המחרוזת 111001 נקודד אותה ל:0321.

בדר״כ יהיה מוגדר מספר $k > 0$, שיהיה מספר הביטים המקסימלי שאפשר לקודד בו run בודד.
איך נטפל בrun גדול שהקידוד שלו דורש יותר מk ביטים, כלומר הrun גדול יותר מ2^k?
עבור המקרים הנדירים האלה נשתמש בescape code שיהיה רצף של אחדות על כל הk ביטים וכך המפענח ידע לא להחליף צד, כלומר, אנחנו נשארים בקידוד של אותו מספר.
עכשיו יש לנו בעיה חדשה, נניח שהתו 0 מופיע $2^k - 1$ פעמים, איך נקודד אותו? הרי אם נקודד אותו ע״י k אחדות המפענח יחשוב שזה escape code, כדי לפתור את זה נוסיף 0 אחרי רצף האחדות.
<u>דוגמה</u>: נניח ש*k=3* ואנו רוצים לקודד את *0111100111000000101011*.
נקודד את זה ל:*1423511112*.
נכתוב את זה ע״י 3 סיביות לכל קידוד ונקבל את הקידוד הסופי: *010 001 001 001 101 011 010 100 001*.
<u>דוגמה נוספת</u>: נניח ש*k=3* ואנו רוצים לקודד את *0000000* (7 אפסים).
הקידוד יהיה: *000 111*.
<u>דוגמה נוספת</u>: נניח ש*k=3* ואנו רוצים לקודד 14 אפסים.
הקידוד יהיה: *000 111 111*.
<u>דוגמה נוספת</u>: נניח ש*k=3* ואנו רוצים לקודד 16 אפסים.
הקידוד יהיה: *010 111 111*.
<u>דוגמה נוספת</u>: נניח ש*k=3* ואנו רוצים לקודד $m(2^k - 1)$ אפסים.
הקידוד יהיה: *m* פעמים *111* ובסוף *000*.

<u>כדי לשפר אפשר לבצע בחירה אדפטיבית של k:</u>
- המקודד יכול לבצע תחילה מעבר על הקלט כדי לבחור את הערך של k הממזער את האורך הכולל של קידוד אורך הריצה.
- אפשר להשתמש בקודים בעלי אורך משתנה לספירות: להשתמש בקודי Huffman וכו' לספירה.

אלגוריתם מקודד:

```
MaxCount ← 2^k-1; count ←0; PrevBit ←"0";
while input remains
   CurBit ←next input bit;
   if PrevBit = CurBit and count < MaxCount
       count++
   else
       output count using k bits
       if count = MaxCount and PrevBit ≠ CurBit
             output 0 using k bits
       count ← 1;
   PrevBit ←CurBit
output count using k bits.
```

אלגוריתם מפענח:

```
MaxCount ← 2^k-1
parity ← 0
while input remains
   read k bits from the input stream to get the
   integer x.
   if parity = 0
       output x "0" bits
   else
       output x "1" bits
   if x < MaxCount
       parity := 1 – parity
```

BWT:

אלגוריתם שמבצע טרנספורמציה לטקסט ומחזיר טקסט שניתן לדחוס אותו בצורה יותר טובה.

אלגוריתם המקודד:

1. נבנה מטריצה של כל הטרנספורמציות הציקליות של הטקסט T.
2. נמיין את שורות המריצה לפי סדר לקסיקוגרפי.
3. נפלוט את העמודה האחרונה במטריצה ואת מספר השורה שבה נמצא הטקסט המקורי במטריצה הממויינת.

1. Generate a matrix of all the cyclic shifts of T
2. Sort the matrix rows in lexicographic order
3. The output of BWT:
   - The final column of the matrix
   - The number of the row corresponding to the original input

דוגמה:

נניח T=mississippi.

בשלב הראשון נבנה את המטריצה עם כל הטרנספורמציות הציקליות של T ונקבל את המטריצה השמאלית.

בשלב השני נמיין את המטריצה לקסיקוגרפית ונקבל את המטריצה הימנית.

לבסוף נפלוט את העמודה האחרונה ואת האינדקס שבו נמצא הטקסט המקורי:

$BWT(T) = (pssmipissii, 4)$

```
mississippi
ississippim
ssissippimi
sissippimis
issippimiss
ssippimissi
sippimissis
ippimississ
ppimississi
pimississip
imississipp
```

```
F         L
imississipp
ippimississ
issippimiss
ississippim
mississippi
pimississip
ppimississi
sippimissis
sissippimis
ssippimissi
ssissippimi
```

תכונות המטריצה הממויינת:

1. מיון של L (העמודה האחרונה) יביא לנו את F (העמודה הראשונה).
2. תווים זהים בL מסודרים באותו סדר בF.

אלגוריתם המפענח:

1. נגדיר את F להיות הקלט L בצורה ממויינת.
2. נגדיר את I להיות האינדקס שקיבלנו.
3. עבור $i = 0, \dots, n$:
   a. $T[i] = F[I]$
   b. $I = S[I]$.

```
BWT(L, Index){
      F ← sort(L)
      I ← Index
      for (i=0; i<n; i++){
            T[i] ← F[I];
            I    ← S[I];
      }
}
```

<u>דוגמה לפיענוח:</u> קלט: $(pssmipissii, 4)$
נגדיר את L להיות הטקסט שקיבלנו ואת F להיות L ממויין.
נגדיר $I = 4$ ונגדיר את T להיות וקטור בגודל n, בסוף נחזיר את T.
לאחר מכן נבנה את S להיות מיפוי של כל תו בF לתו המתאים בL.
נתחיל מאינדקס 4, ולכן נוסיף לT את F[4]: T=m.
S ממפה את 4 בL ל3 בF ולכן נוסיף לT את F[3]: T=mi.
S ממפה את 3 בL ל10 בF ולכן נוסיף לT את F[10]: T=mis.
S ממפה את 10 בL ל8 בF ולכן נוסיף לT את F[8]: T=miss.
S ממפה את 8 בL ל2 בF ולכן נוסיף לT את F[2]: T=missi.
וכן הלאה.

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

<u>דוגמה נוספת ללא שימוש בS:</u>
קלט: $(nnbaaa, 3)$.

F	
0	a
1	a
2	a
3	b
4	n
5	n

L	
0	n
1	n
2	b
3	a
4	a
5	a

התחלנו מאינדקס 3 בF, נפלוט b.
נעבור לb המתאים בL, נמצא באינדקס 2, ומשם נחזור לL לאינדקס הזהה ונפלוט את a (נמצא באינדקס 2 בF).
נעבור לb המתאים בL, זהו הa השלישי בF ולכן נעבור לa בשלישי בL, נמצא באינדקס 5, ומשם נחזור לL לאינדקס הזהה ונפלוט את n (נמצא באינדקס 5 בF) וכן הלאה...

MoveToFront Compression:
כל תו בקלט מקודד לפי מספר התווים השונים שהופיעו מאז הופעתו האחרונה של תו זה.
מיושם באמצעות רשימה של תווים ממוינים לפי עדכניות השימוש.
הפלט מכיל מספרים קטנים רבים אם הטקסט הומוגני מקומי – BWT פולט טקסט שהוא הומוגני מקומי.
<u>דוגמה:</u>
קלט: $T = helloworld$, $\Sigma = \{d, e, h, l, o, r, w\}$.
לאורך כל הקידוד נתחזק את רשימת התווים ונעדכן אותה בכל איטרציה.
קידוד התו הראשון:
$MTF - List = \{d, e, h, l, o, r, w\}$, $char = 'h'$
התו $'h'$ מופיע במקום השני ברשימה ולכן נפלוט 2 ונבצע הזזה קדימה לתו $'h'$ ברשימה.
קידוד התו השני:
$MTF - List = \{h, d, e, l, o, r, w\}$, $char = 'e'$
התו $'e'$ מופיע במקום השני ברשימה ולכן נפלוט 2 ונבצע הזזה קדימה לתו $'e'$ ברשימה.
קידוד התו השלישי:
$MTF - List = \{e, h, d, l, o, r, w\}$, $char = 'l'$
התו $'l'$ מופיע במקום השלישי ברשימה ולכן נפלוט 3 ונבצע הזזה קדימה לתו $'l'$ ברשימה.
קידוד התו הרביעי:
$MTF - List = \{l, e, h, d, o, r, w\}$, $char = 'l'$
התו $'l'$ מופיע במקום האפס ברשימה ולכן נפלוט 0 ונבצע הזזה קדימה לתו $'l'$ ברשימה – בפועל הוא כבר נמצא שם.
נמשיך כך ובסוף נקבל את הקידוד: *2230461...*.

<u>האלגוריתם הכללי לדחיסה:</u>
קלט – מחרוזת *T*.
1. *BWT(T)* – נבצע טרנספורמציה לטקסט.
2. *MTF* – נבצע קידוד לטקסט שקיבלנו מהBWT.
3. נבצע דחיסה סטטיסטית לקידוד ע״י *Huffman* או *Arithmetic Coding*.