פרוייקט באסמבלי – קידוד הופמן

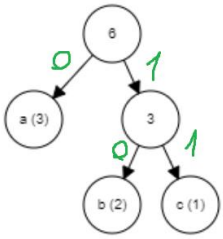
מגיש: תומר רובינשטיין.
כיתה: י'3.
מנחה: אליהו גולדשטיין.

פרק א' – תיאור הפרוייקט

בחרתי לממש את אלגוריתם הקידוד של הופמן לצורך דחיסת קבצים.
באמצעות יצירת עץ הופמן[1], ניתן לקבוע **קוד הופמן** ייחודי עבור כל תו (המופיע בתור עלה בעץ ההופמן) ע"י הדרך מהשורש לאותו תו (עלה) בכך שבכל פעם שאנו פונים לבן ימני בעץ אנו נוסיף לקוד ההופמן את הסיבית 1 ועבור פנייה לבן שמאלי נוסיף 0, כסיבית[2].
הליך בניית עץ הופמן מבטיח שעבור התו בעל התדירות הגבוהה היותר, מס' הסיביות בקוד ההופמן שלו הוא הקטן ביותר, וההפך.

ניזכר שכל תו בקובץ טקסטואלי לוקח בית אחד. לכן, אם ישנם k תווים בתוכן הקובץ, נצטרך להקצות בזיכרון סה"כ $8k$ סיביות.
לכן אם נחליף כל תו בקובץ שיש בו הרבה חזרות של אותם תווים בקוד ההופמן שלהם, בהכרח נקבל שעבור קבצים עם הרבה תווים שחוזרים על עצמם, הקובץ הדחוס יהיה קטן יותר. למשל:

בהינתן הטקסט: "aaabbc". יש בו 6 תווים אשר כל תו עולה בית בזיכרון ולכן כקובץ טקסט נקבל $6 * 8 = 48$ סיביות.
ניצור עץ הופמן עבור הטקסט:



נקבל את ספר הקודים הבא:

$$(*)\begin{bmatrix} a \equiv_{hf} & 0 \\ b \equiv_{hf} & 10 \\ c \equiv_{hf} & 11 \end{bmatrix}$$

נציב במקום כל תו בטקסט המקורי את קוד ההופמן שלו (כבינארי!) ונקבל:

$$000101011$$

קיבלנו 9 סיביות בטקסט הדחוס (לעומת 48)!
אך אין זו המציאות, עלינו גם לשמור את ספר הקודים הנידון $(*)$ בקובץ הדחוס, לצרכי קריאת הקובץ הדחוס..
נשמור כל תו כמקורו (כבית שלם) ולאחר מכן את קוד ההופמן שלו (כבית שלם) ונקבל שנצטרך להוסיף לקובץ הדחוס עוד $3 * 2 = 6$ בתים ולכן סה"כ הקובץ הדחוס יכיל $(6 * 8) + 9 = 57$ סיביות.

בשביל לקרוא את הקובץ הדחוס, ניקרא את קוד ההופמן משמאל לימין וכאשר נקבל התאמה בספר הקודים, נרשום את התו המתאים.
משום שקוד הופמן הוא קוד תחיליות, אין קוד הופמן מסוים המכיל תת-רצף (התמיד מתחיל משמאל) המהווה קוד הופמן בספר הקודים.
למשל התת-רצף `1` בקוד ההופמן של c ב-$(*)$ אינו מייצג אף תו בספר הקודים.

►

מסקנה: קידוד הופמן יעיל עבור קבצים בהם ישנם תווים עם הרבה חזרות.

דוד (אלברט) הופמן הוכיח את נכונות האלגוריתם בשנת 1951 ב-MIT כאשר הוטל על כיתתו לתורת האינפורמציה הבחירה לעשות מבחן סיום או עבודת מחקר על מציאת הקוד הבינארי היעיל ביותר.
הופמן לא צלח ברעיונותיו לעבודת המחקר, וכמעט החליט לוותר וללמוד למבחן הסיום, אך לפתע הכה בו הרעיון לעצים בינאריים המסודרים לפי תדירויות (עץ הופמן) והוכיח ששיטה זו הייתה היעילה ביותר.

---

[1] עץ הופמן הוא עץ בינארי הנבנה מלמטה למעלה.
[2] המחשה: https://cmps-people.ok.ubc.ca/ylucet/DS/Huffman.html.

פרק ב' – רפלקציה

בהתחלה שהייתי צריך לבחור רעיון לפרוייקט, חשבתי על רעיונות פשוטים למדי, שלא עניינו אותי ולכן ידעתי שלא תהיה לי די מוטיבציה לעבוד עליהם. שנתקלתי ברעיון של דחיסת קבצים, התלהבתי, כי לא ידעתי בדיוק מה זה ואיך לממש את זה.

בהתחלה שלמדתי איך האלגו' פועל, ישר רציתי לראות את המימוש שלו בקוד, ונתקלתי באתר[3]. ראיתי שהמימושים השונים שהם עשו בשפות השונות תמכו במבנים, אך ב-x86 לא מצאתי אף תמיכה עבור מבנים!

נזכרתי שאפשר לייצג עץ באמצעות מערך ולכן מתוך אינסטינקט קפצתי לרעיון של tree traversal, אבל כל פתרון שחשבתי עליו היה מסורבל מדי או לא פעל היטב (בדיעבד, אפשר היה גם לממש את הפרוייקט באמצעות max heap).

בסופו של דבר, הלכתי על הדרך הנאיבית שרעיונה הוא להגדיר שני מערכים מסונכרנים, אחד מייצג את התווים ואחד את התדירות של אותו תו.
קרי: עבור התו ה-i במערך A, התדירות שלו היא $B[i]$ (כאשר A הוא מערך התווים ו-B הוא מערך התדירויות), ובאמצעות שני המערכים לבנות את קודי ההופמן הרצויים.

אילו הייתי יכול להמשיך הייתי משפר את יעילות הפרוצדורות בקוד (מבחינת מקום וזמן ריצה), לוודא את בטיחות הקוד (לוודא שלא יהיו גלישות זיכרון ואחר) ולעשות תמיכה לקבצים יותר גדולים (להגדיל את מס' הסיביות המוגדרות עבור כל בלוק מידע בספר הקודים).

[3] https://www.geeksforgeeks.org/huffman-coding-greedy-algo-3/

פרק ג' – תכנון הקוד

<u>פרוצ' printNum</u>
קלט: מספר כלשהו.
פלט: הדפסת המספר למסך ספרה לאחר ספרה בסדר הנקוב עם 3 אפסים מרפדים.

```
proc printNum
        mov bp, sp
        mov ax, [bp+2]

        mov bh, 3
        mov cx, 10

        printNum1:
                xor dx, dx
                div cx
                push dx

                sub bh, 1
                cmp bh, 0
                jg printNum1

        mov bh, 3
        xor dx, dx
        printNum2:
                pop dx

                add dx, '0'
                mov ah, 2
                int 21h

                sub bh, 1
                cmp bh, 0
                jg printNum2
        ret 2
endp printNum
```

פרוצ' buildFreqArr

קלט: אין.

פלט: המערך freqArr(word, 128 cells) יהי למערך מונים המכיל את התדירות של כל תו (אסקי) בתוכן קובץ נתון (inputFilename), כך שהתדירות של תו c הינו $freqArr[c]$.

```
proc buildFreqArr
        ; open file
        mov ah, 3Dh
        xor al, al
        lea dx, [inputFilename]
        int 21h
        jc openError
        mov [filehandle], ax

        ; read file and store content in filecontent
        mov ah, 3Fh
        mov bx, [filehandle]
        mov cx, 1200
        mov dx, offset filecontent
        int 21h

        ; incrementing the element in freqArr corresponding to the current ASCII char value as index
        ; i.e. to increment the appearences of the char 'a', increment freqArr[97] by 1.
        mov si, 0
        loop_genFreqArr:
                mov bx, 0
                mov bl, [filecontent+si]
                inc [freqArr+bx]

                inc si
                cmp [filecontent+si], 0 ; read until the null terminator
                jne loop_genFreqArr
        mov [byteCount], si
```

```
        ; stdout the byte count in the given file
        mov dx, offset log_OrgByteCount
        mov ah, 9
        int 21h
        push [byteCount]
        call printNum
        ; newline
        mov dl, 0Ah
        mov ah, 2
        int 21h
        ret
        ; log error msg if the file couldn't be opened
        openError:
                mov dx, offset log_OpenError
                mov ah, 9
                int 21h
                jmp exit
        ret
endp buildFreqArr
```

פרוצ' splitFreqArr

קלט: אין.

פלט: פיצול המערך freqArr לשני מערכים מסונכרנים באופן הבא:

- המערך freqChars מכיל את כל התווים המופיעים בתוכן הקובץ בסדר רציף (מסקנה: משום שאנו עוברים על המערך freqArr באופן רציף, נסיק שfreqChars ממויין בסדר לקסיקוגרפי).
- המערך freqCharsCount מכיל את התדירות של כל תו ב-freqChars בהתאמה.

קרי: התדירות של $freqChars[i]$ הוא $freqCharsCount[i]$.

```
proc splitFreqArr
        mov bp, sp

        xor si, si
        xor bx, bx

        splitChars:
                xor cx, cx
                mov cl, [(offset freqArr)+si]
                cmp cx, 0
                jnz addChar

                ;loop iteration & condition
                splitCharsIter:
                        inc si
                        cmp si, 128
                        jne splitChars
                        je endSplitArr

                addChar:
                        ;insert the current char to freqChars(dw, 128)
                        mov [freqChars+bx], 0
                        mov [freqChars+bx], si
                        ;insert the current NOA of this char to freqCharsCount(dw,
128)
                        mov [freqCharsCount+bx], 0
                        mov [freqCharsCount+bx], cx
                        add bx, 2
                        jmp splitCharsIter

        endSplitArr:
        ret
endp splitFreqArr
```

<u>פרוצ' findMins</u>

הפרוצ' מחזירה את האינדקסים של שני הערכים המינימליים ב-$freqCharsCount$

קלט: אין.

פלט:

- ax = האינדקס של המספר המינימלי במערך $freqCharsCount$.
- cx = האינדקס של המספר השני המינימלי במערך $freqCharsCount$.

```
proc findMins
    mov bp, sp

    xor si, si
    xor bx, bx
    xor di, di

    firstMinLoop:
        cmp [freqCharsCount+si], -1 ; end-of-freqCharsCount
        je endFirstMin

        mov cx, [freqCharsCount+si]
        mov dx, [freqCharsCount+bx]

        cmp cx, dx
        jl newFirstMin

        add si, 2 ; word indexing
        jmp firstMinLoop
    endFirstMin:

    ; bx now holds the minimal value's index
    cmp di, 0
    je firstResult
    jne secondResult
```

```
    firstResult:
        mov ax, [freqCharsCount+bx]
        push ax
        mov ax, bx ; ⟹ ax - first result
        mov [freqCharsCount+bx], 199Ah ; max value
        ; find the second minimal value
        xor si, si
        xor bx, bx
        inc di
        jmp firstMinLoop

    secondResult:
        mov cx, bx ; ⟹ cx - second result
        ; retrieve the first minimal value to freqCharsCount
        pop dx
        mov si, ax
        mov [freqCharsCount+si], dx
    ret
    newFirstMin:
        mov bx, si
        add si, 2
        jmp firstMinLoop
endp findMins
```

פרוצ' addCells

קלט:

- [bp+2], אינדקס של תא אחד
- [bp+4], אינדקס של תא שני

פלט:

הפרוצ' סוכמת שני תאים נתונים ב-$freqCharsCount$ ומאפסת את התא בעל התדירות הגבוהה יותר.
יתרה מזאת, היא גם משנה את $freqChars$ בכך שהיא מאפסת את התא בעל התדירות הגבוהה יותר וגם משנה את ערך תא הסכום ל-$parentCount$ בהתאם (כל פעם שהפרוצ' נקראת, $parentCount$ עולה באחד).

```
proc addCells

        mov bp, sp


        mov bx, [bp+2] ; i

        mov si, [bp+4] ; j


        ; make it so: i < j

        cmp bx, si

        je exitAddCells

        jl skipSwap

        ; i > j, swap

        mov bx, [bp+4] ; i

        mov si, [bp+2] ; j


        skipSwap:

        ; if we are summing a parent node(s)

        cmp [freqChars+bx], 128

        jge incParentCount


        cmp [freqChars+si], 128

        jge incParentCount
```

```
        ; sum freqCharsCount cells
        mov dx, [freqCharsCount+si]
        add [freqCharsCount+bx], dx
        mov [freqCharsCount+si], 199Ah

        ; change freqChars
        mov [freqChars+si], 0
        xor dx, dx
        mov dl, [parentCount]
        mov [freqChars+bx], dx
        jmp exitAddCells
        incParentCount:
                xor dx, dx
                mov dl, [parentCount]
                mov [freqChars+bx], dx
                mov [freqChars+si], 0
                mov dx, [freqCharsCount+si]
                add [freqCharsCount+bx], dx
                mov [freqCharsCount+si], 199Ah
                jmp exitAddCells

        exitAddCells:
        ret 4
endp addCells
```

פרוצ' insertCodebook

הפרוצ' מכניסה\מעדכנת בלוק מידע בספר הקודים.

נגדיר בלוק מידע:

parentCount	hf code bit	…	hf code bit	hf code bit	char

char – התו שאנו מעוניינים להראות את הייצוג הופמן שלו בספר הקודים.

hf code bit – הסיביות שמייצגות את קוד ההופמן של התו (כל אחת עולה בית בזיכרון). תחום הערכים הוא 0, 1 או 2. 0 ו-1 מייצגים סיביות תוצאה אולם 2 מייצגת ערך null (כלומר, אם קוד ההופמן לא עולה על blockSize[4]).

parentCount[5] – האב של אותו "צומת" בעץ ההופמן (אנו מייצגים צומת באמצעות בלוק המידע הנ"ל).

קלט:

- [bp+2], parentCount (האבא של הצומת)
- [bp+4], סיבית תוצאה שאנו מעוניינים להוסיף לקוד ההופמן של אותו תו.
- [bp+6], התו שאנו מעוניינים להוסיף\לעדכן בספר הקודים.

פלט: מילוי בלוק מידע בספר הקודים, כמצוין בדיון דלעיל.

```
proc insertCodebook

        mov bp, sp


        mov ax, [bp+2] ; new parentCount = al

        xor ah, ah

        mov bx, [bp+4] ; result bit (0 or 1) = ah

        mov ah, bl

        xor bx, bx

        mov bx, [bp+6] ; (ASCII) char = bl


        cmp bl, 80h

        jae incChilds


        xor si, si

        searchCodebook:

                cmp si, 2048

                jge initDataBlock ; else init a new data block
```

[4] blockSize – מספר הבתים שאנו מקציבים ל-hf code bit (במימוש הספציפי הזה – 8).

[5] ערך התחלתי: 80h. מדוע? – משום ש-80h אינו מייצג אף תו אסקי, נוכל להגדיר שאם char=parentCountA (קרי ערכו גדול מ-80h) ו-parentCount ערך אחר, נוכל לעדכן את כל הבנים של char ([bp+6]) לערך של parentCount ([bp+2]).

```
        cmp [codebook+si], bl
        je updateDataBlock ; if char in codebook, update data block

        xor dx, dx
        mov dl, [blockSize]

        add si, dx
        add si, 2
        jmp searchCodebook

updateDataBlock:
push si ; head of data block
updateDataBlockLoop:
        cmp [codebook+si], 2 ; only 12 bit-space
        jne udb_iteration
        mov [codebook+si], ah ; insert the new result bit

        ; going to the parentCount cell's index
        pop si
        xor dx, dx
        mov dl, [blockSize]
        add si, dx
        add si, 1

        ; replacing the old parentCount with the new one
        mov [codebook+si], al

        jmp end_insertCodebook
        udb_iteration:
                inc si
                jmp updateDataBlockLoop
```

```
jmp end_insertCodebook
; ----------
initDataBlock:
xor si, si

; finding the first available data block to insert to
searchPlace:
        cmp [codebook+si], 2
        je insertDataBlock

        xor dx, dx
        mov dl, [blockSize]

        add si, dx
        add si, 2
        jmp searchPlace

insertDataBlock:
        mov [codebook+si], bl ; char
        inc si
        mov [codebook+si], ah ; result bit

        xor dx, dx
        mov dl, [blockSize]

        add si, dx
        mov [codebook+si], al ; new parent count

end_insertCodebook:
ret 6
```

```
; ----------
incChilds:
        mov si, -1
        incChildsLoop:
                xor dx, dx
                mov dl, [blockSize]

                ; looping through the parents of each data block
                add si, dx
                add si, 2

                ; end of codebook
                cmp si, 2048
                jge end_insertCodebook

                ; found a match, add the result bit to the huffman code of that data block
                cmp [codebook+si], bl
                je addResultBit

                jmp incChildsLoop

; found a match for the parent, inserting the result bit in the correct cell of the result huffman code.
addResultBit:
        push si ; saving si to search for other matches later on
        mov [codebook+si], al

        ; iterating backwards (dec index) in the codebook
        addResultBitLoop:
                cmp [codebook+si-1], 2 ; searching for an available cell
                je ARB_iter
```

```
        ; found an available cell
        mov [codebook+si], ah ; insert the result bit
        pop si
        jmp incChildsLoop ; searching for other matches...
        ARB_iter:
        dec si
        jmp addResultBitLoop
endp insertCodebook
```

<u>פרוצ' getFreqLength</u>

קלט: אין.

פלט: dx – אורך המערך $freqCharsCount$.

```
proc getFreqLength
        mov bp, sp

        xor dx, dx
        xor si, si
        GFL_loop:
                cmp [freqCharsCount+si], -1 ; end-of-freqCharsCount
                je GFL_end

                cmp [freqCharsCount+si], 199Ah ; “null” value
                je GFL_iter

                inc dx

                GFL_iter:
                        add si, 2
                        jmp GFL_loop

        GFL_end:
        ret
endp getFreqLength
```

פרוצ' buildCodebook

קלט: אין.

פלט: ספר קודים עבור תוכנו של הקובץ הנתון.

```
proc buildCodebook
        mov bp, sp

        BC_loop:
                ; until 1 node is left
                call getFreqLength
                cmp dx, 1
                je end_buildCodebook

                ; finding the two minimums
                call findMins
                mov bx, ax ; i
                mov si, cx ; j

                ; inserting to the codebook
                ; finding the cell with the higher frequency and assign it's index to si
                ; the other cell goes to bx
                mov dx, [freqCharsCount+bx]
                cmp dx, [freqCharsCount+si]
                jle BC_loop_continue
                mov dx, bx ; tmp
                mov bx, si
                mov si, dx

                BC_loop_continue:
                push ax
                push bx
                push cx
```

```
; bx gets 0 as result bit
push si
mov ax, [freqChars+bx]
xor bx, bx
mov bx, 0
xor cx, cx
mov cl, [parentCount]
push ax ; [bp+6] = bl = char
push bx ; [bp+4] = ah = bit
push cx ; [bp+2] = al = count
call insertCodebook
pop si


; si gets 1 as result bit
mov ax, [freqChars+si]
xor bx, bx
mov bx, 1
xor cx, cx
mov cl, [parentCount]
push ax ; [bp+6] = bl = char
push bx ; [bp+4] = ah = bit
push cx ; [bp+2] = al = count
call insertCodebook

; adding the cells
pop cx
pop bx
pop ax
push ax
push cx
```

```
            call addCells
            inc [parentCount]

            jmp BC_loop
        end_buildCodebook:
        ret
endp buildCodebook
```

פרוצ' tidyCodebook

קלט: אין.

פלט: היפוך קודי ההופמן של כל תו בספר הקודים.

```
proc tidyCodebook
        mov bp, sp

        ; reverse the huffman code of each data block
        xor si, si
        l1:
            cmp [codebook+si], 2
            je end_reverseHuffman

            mov bx, si
            ; find the tail index of the huffman code >> bx
            l2:
                cmp [codebook+bx+1], 2
                je end_l2

                inc bx
                jmp l2
            end_l2:
            push si
            inc si
            ; si < bx
            l3:
```

```
                cmp si, bx
                jge end_l3
                ; exchange start, end
                mov al, [codebook+bx]
                mov ah, [codebook+si]
                mov [codebook+bx], ah
                mov [codebook+si], al

                inc si
                dec bx
                jmp l3

            end_l3:
            pop si

            xor ax, ax
            mov al, [blockSize]
            add si, ax
            add si, 2
            jmp l1

        end_reverseHuffman:
        ret
endp tidyCodebook
```

פרוצ' outputByte

קלט: אין.

פלט: הדפסת $[byteToWrite]$ מה-*dataseg* לקובץ התוצאה (לקובץ הדחוס הסופי).

```
proc outputByte
        mov bp, sp

        ;output [byteToWrite] to the compressed file
        mov ah, 40h
        mov bx, [newFilehandle]
        mov cx, 1
        mov dx, offset byteToWrite
        int 21h

        ret
endp outputByte
```

פרוצ' outputHuffmanCode

קלט: [bp+2], תו מתוכן הקובץ.

פלט: הדפסת קוד ההופמן של אותו תו מספר הקודים לקובץ התוצאה.

```
proc outputHuffmanCode
        mov bp, sp
        mov cx, [bp+2] ; cl - holds the char

        xor si, si ; index

        ; lookup cl char in the codebook
        xor bx, bx
        t2:
                cmp [codebook+bx], cl
                je end_t2

                add bx, 2
                add bl, [blockSize]
```

```
        jmp t2
    end_t2:

    ; now read the huffman code and append to [byteToWrite]
    inc bx
    t3:
        cmp [bitsCount], 8 ; byteToWrite is full
        je writeByte
        cmp [codebook+bx], 2
        je end_t3 ; done writing the huffman code to [byteToWrite]

        ; append huffman code bit-by-bit to [byteToWrite]
        mov al, [codebook+bx]
        shl [byteToWrite], 1
        add [byteToWrite], al

        inc [bitsCount] ; bits count
        inc bx
        jmp t3
    end_t3:
    ret 2
    writeByte:
        inc [byteCount]
        push bx
        call outputByte
        mov [byteToWrite], 0
        mov [bitsCount], 0
        pop bx
        jmp t3
endp outputHuffmanCode
```

פרוצ' outputCodebook

קלט: אין.

פלט: הדפסת ספר הקודים לקובץ התוצאה כך שכל תו מודפס (כבית) ומיד אחריו את קוד ההופמן שלו (כבית שלם).

```
proc outputCodebook
        mov bp, sp

        xor bx, bx
        oc_loop:
                xor ch, ch
                mov cl, [codebook+bx]
                mov [byteToWrite], cl
                push bx
                push cx
                call outputByte ; write char as *byte* to output file
                mov [bitsCount], 0
                mov [byteToWrite], 0
                call outputHuffmanCode

                cmp [bitsCount], 0
                je skip_output_byte
                call outputByte

                skip_output_byte:
                mov [byteToWrite], 0
                 ;skip to the next char in the codebook
                pop bx
                add bx, 2
                add bl, [blockSize]
                cmp [codebook+bx], 2
                jne oc_loop

        ret
endp outputCodebook
```

<u>פרוצ' compress</u>

קלט: אין.

פלט: קובץ דחוס לפי קידוד הופמן ושמו יהי לשם הקובץ הנתון אך החלפת הסיומת שלו ל-*"hf"*.

בנוסף, הפרוצ' מוציאה לקובץ התוצאה 3 בתים המהווים *header*:

- 2 בתים ראשונים בקובץ – מס' הבתים בקובץ המקורי, בכדי שנוכל לדעת מתי להפסיק לקרוא בדחיסה
- בית שלישי – סך כל הבתים בספר הקודים הדחוס (דהיינו, בקובץ התוצאה), בכדי שנוכל לדלג על ספר הקודים הדחוס וישירות לקרוא את קודי ההופמן.

```
proc compress
        mov bp, sp

        call buildFreqArr
        call splitFreqArr
        call buildCodebook
        call tidyCodebook

        ; create new file named after [filename]
        mov ah, 3Ch
        xor cx, cx
        mov dx, offset filename
        int 21h
        mov [newFilehandle], ax         ; save new file handle

        ; output [byteCount] to the compressed file
        mov bx, [byteCount]
        mov [byteToWrite], bh
        call outputByte
        mov bx, [byteCount]
        mov [byteToWrite], bl
        call outputByte

        mov [byteToWrite], 0
        mov [byteCount], 0
```

```
; get the codebook's length
xor bx, bx
xor si, si
lc_loop:
        cmp [codebook+si], 2
        je end_lc_loop

        inc bx
        add si, 2
        xor dx, dx
        mov dl, [blockSize]
        add si, dx
        jmp lc_loop
end_lc_loop:
; bx now holds the number of different chars in the given text
mov ax, bx
xor bx, bx
mov bl, 2
mul bl
; ax now holds the length of the codebook
mov ah, 0
push ax
mov [byteToWrite], al
call outputByte ; output the length of the "compressed" codebook as the third byte header
mov [byteCount], 3 ; 3 bytes for the header
pop ax
add [byteCount], ax

; output the codebook
call outputCodebook
```

```
        ; output huffman codes
        mov [bitsCount], 0
        mov [byteToWrite], 0
        xor si, si
        ofc_loop:
                xor cx, cx
                mov cl, [filecontent+si]
                push si
                push cx
                call outputHuffmanCode
                pop si
                inc si
                cmp [filecontent+si], 0
                jne ofc_loop

        ; output the final byte
        mov ax, 8
        sub al, [bitsCount]
        _finalByte:
                shl [byteToWrite], 1
                dec ax
                cmp ax, 0
                jne _finalByte
        call outputByte
        inc [byteCount]
        ret
endp compress
```

<u>פרוצ' findPattern</u>

קלט: *[bp+2]*, קוד הופמן כלשהו כבית שלם.

פלט:

ax – המכיל את התו המותאם באופן חח"ע (לפי נכונות אלגו' הקידוד של הופמן), ו-0 אם קוד הופמן שכזה לא קיים בספר הקודים.

```
proc findPattern
        mov bp, sp
        mov ax, [bp+2]

        xor si, si
        fp_loop:
                cmp [codebook+si+1], al
                je found
                cmp [codebook+si], 2 ; end-of-codebook
                je notFound
                ; iteration
                xor bx, bx
                mov bl, [blockSize]
                add si, bx
                add si, 2
                jmp fp_loop
        found:
                xor ax, ax
                mov al, [codebook+si]
                cmp ax, 2
                je notFound
        ret 2
        notFound:
                mov ax, 0
        ret 2
endp findPattern
```

פרוצ' decompress

קלט: אין.

פלט: הקובץ המקורי שמחולץ מקובץ דחוס נתון. בנוסף, הפרוצ' בונה את ספר הקודים מהקובץ הדחוס כך שכל בלוק מידע ייראה כך:

char	hf code as byte	2	…	2	80h

```
proc decompress
        mov bp, sp

        ; create new file named after [outputFilename]
        mov ah, 3Ch
        xor cx, cx
        mov dx, offset outputFilename
        int 21h
        mov [newFilehandle], ax          ; save new file handle

        ; open file
        mov ah, 3Dh
        xor al, al
        lea dx, [filename]
        int 21h
        jnc continue_decompress

        ; log error msg if the file couldn't be opened & exit
        mov dx, offset log_OpenError
        mov ah, 9
        int 21h
        jmp exit

        continue_decompress:
        mov [filehandle], ax

        ; read file and store content in filecontent
```

```
mov ah, 3Fh
mov bx, [filehandle]
mov cx, 1200
mov dx, offset filecontent
int 21h

xor dx, dx
mov dh, [filecontent+0]
mov dl, [filecontent+1]
mov [org_filecontent_len], dx

xor dx, dx
mov dl, [filecontent+2] ; dl := length[codebook]
add dl, 3 ; to ignore the compressed file's headers

xor si, si
add si, 3 ; for reading the codebook
; insert the codebook from the file to [codebook]
df_loop:
        cmp si, dx ; end-of-codebook
        jae end_df_loop
        push dx
        push si

        xor ax, ax
        xor bx, bx
        xor cx, cx
        mov al, [filecontent+si]
        mov bl, [filecontent+si+1]
        mov cl, [parentCount]
        push ax
```

```
        push bx
        push cx
        call insertCodebook

        pop si
        pop dx
        add si, 2
        jmp df_loop
end_df_loop:

; re-calculate the start index of the huffman code
xor bx, bx
mov bl, [filecontent+2] ; length of the codebook
add bx, 3 ; to ignore the header bytes
xor cx, cx ; curr pattern
dc_loop:
        ; lookup current huffman code
        mov al, [filecontent+bx]
        xor dx, dx ; bit count
        f1:
                ; done writing all original bytes
                mov si, [byteCount]
                cmp [org_filecontent_len], si
                je end_dc_loop

                cmp dx, 8 ; done with this byte
                je end_f1

                ; add 1 or 0 to the LSB of cx
                shl cx, 1
                clc
```

```
shl al, 1
push ax
jnc f1_continue

add cx, 1
f1_continue:
        push bx
        push dx
        push cx

        ; lookup the pattern cx represents in the codebook
        push cx
        call findPattern

        ; pattern not found, need more bits to determine
        cmp ax, 0
        jz f1_iter

        ; pattern found, output to the result file
        mov [byteToWrite], al
        call outputByte
        ; search for a new pattern
        inc [byteCount]
        pop cx
        xor cx, cx
        jmp keep_cx
f1_iter:
        pop cx
        keep_cx:
        pop dx
        pop bx
```

```
                    pop ax

                    inc dx
                    jmp f1
            end_f1:
            inc bx
            jmp dc_loop
        end_dc_loop:
        ret
endp decompress
```

תיאור הפתרון

<u>דחיסה</u>:

לאחר קבלת המידע הדרוש מהמשתמש, נפתח את הקובץ שברצוננו לדחוס, נקרא את תוכן הקובץ ונכניס את התדירות של כל תו למערך מונים המכיל 127 תאים (למשל הערך באינדקס ה-97 הוא התא המכיל את מס' הפעמים שהתו *'a'* [לפי אסקי] מופיע בתוכן הקובץ).
נניח לדוגמה שאנו דוחסים את הטקסט *"aaabbc"* אזי לאחר ביצוע שלב זה נקבל:

Value:		3	2	1	
Index:	...	97	98	99	...

נפצל את מערך המונים לשני מערכים נפרדים (המתחילים מ-0): האחד מכיל את כל התווים המופיעים בתוכן הקובץ והאחר מכיל את מס' הפעמים שכל תו מופיע. בדוגמה שלנו זה ייראה כך:

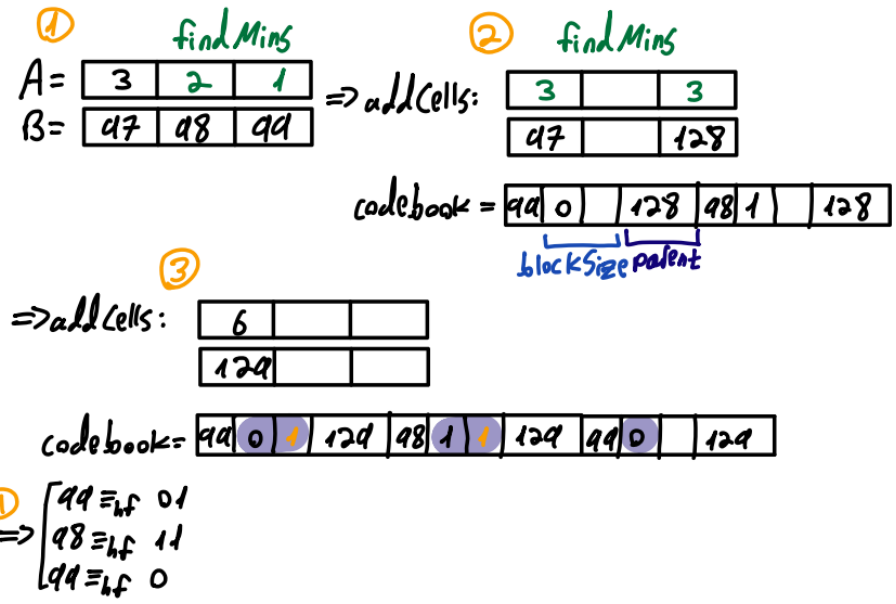
$$A = [3,2,1,\dots]$$
$$B = [97,98,99,\dots]$$

(שני המערכים מסונכרנים, קרי: התדירות של האות *'a'* ב-B הוא $A[0]$)

כעת נבנה את ספר הקודים.
נמצא את שני התווים עם מס' התדירות הנמוך ביותר במערך A (הפרוצ' *findMins*), **נחבר** אותם ונוסיף את הסיבית 0 בספר הקודים לתו עם התדירות הקטנה יותר ו-1 לאחר. נעשה זאת עד שכאשר המערך *A* מכיל תא אחד בלבד.

נגדיר את **פעולת החיבור** (הפרוצ' *addCells*) על שני תאים במערך:
בתא עם התדירות הנמוכה יותר, נוסיף לתדירות שלו את התדירות של התא האחר ונאפס את התא האחר (גם במערך *A* וגם במערך *B*).
במקום התו של התא החדש, נציב ערך שנקרא *parentCount* (המאותחל ל-128 וגדל כל פעם ב-1 כאשר מתבצעת פעולת החיבור).
אם מתבצעת פעולת חיבור על תא אשר ערך התו שלו גדול ממש מ-127 (קרי לא אסקי, אזי הוא אב) נוסיף את התדירויות כמקודם, נאפס את התא הגדול יותר, נוסיף (בספר הקודים) לכל הבנים אשר אביהם הוא ה-*parentCount* את הסיבית שקיבל אותו תא ולאחר מכן נעדכן את ערך "התו" (קרי לשנות ל-*parentCount*)

נמשיך את הדוגמה:

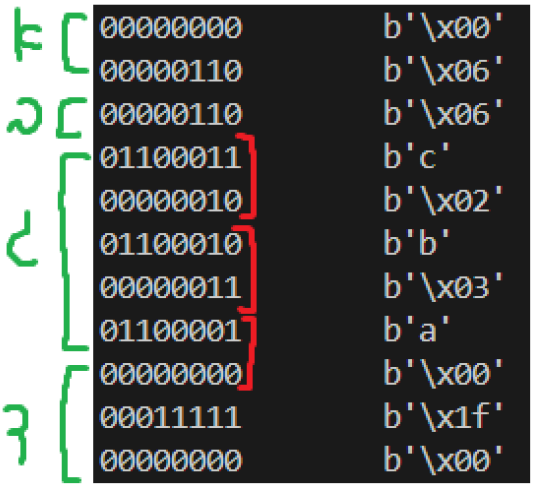


נשים לב כי קיבלנו קודי הופמן הפוכים (מעבר 4 בהמחשה), נרצה להפוך אותם (הפרוצ' *tidyCodebook*) ונקבל:

$$codebook := \begin{bmatrix} 99 & \equiv_{hf} & 10 \\ 98 & \equiv_{hf} & 11 \\ 97 & \equiv_{hf} & 0 \end{bmatrix}$$

כעת נרשום את הקובץ הדחוס עם ספר הקודים שבנינו.
ראשית נדפיס את שלושת בתי ה-*header*, הם יעזרו לנו בדחיסה:
שני הבתים הראשונים בקובץ מהווים מספר *word* ביחד (הבית הראשון זה ה-*high* והשני ה-*low*) המייצג את מס' הבתים בקובץ המקורי, כך נדע מתי בדיוק להפסיק את הליך הדחיסה.
הבית השלישי מהווה את סך כל הבתים בספר הקודים הדחוס, הוא יהי למס' התווים השונים $*$ 2 (כי כל תו ייקח בית וקוד ההופמן שלו ייקח גם בית). בזכותו, נוכל לדלג בנקל על ספר הקודים בקובץ הדחוס ולקרוא את קודי ההופמן.
לאחר מכן נדפיס את ה-*codebook* לקובץ התוצאה כך שכל תו מודפס כבית שלם וקוד ההופמן שלו מודפס מיד אחריו כבית שלם.
ובסופו של דבר, נדפיס את תוכן הקובץ בצורת ההופמן שלו:
נעבור תו-תו על תוכן הקובץ המקורי ונחפש את ערך התו הנוכחי ב-*codebook*, נוסיף את קוד ההופמן שלו ל-*byteToWrite* (משתנה כללי מסוג בית) ולאחר ספירה שהוספנו 8 סיביות (סה"כ, לא בהכרח אותו תו) ל-*byteToWrite*, נדפיס אותו לקובץ התוצאה, נאפס אותו ונמשיך בלולאה עד לקבלת תו *EOF*.

הקובץ הדחוס ייראה בדוגמתנו כך:



(החלק השמאלי בתמונה וגם ה*newlines* אינם קיימים בקובץ הדחוס ומשמשות לצרכי המחשה)
חלק א' – שני בתים המהווים מספר מסוג *word* המייצגים את מס' הבתים בקובץ המקורי. דהיינו נקבל את המספר:

$$0000000000000110 \equiv_{dec} 6$$

חלק ב' – בית המציין את מס' הבתים בספר הקודים הדחוס (חלק ג').
חלק ג' – ספר הקודים הדחוס.
חלק ד' – הטקסט המקורי (*"aaabbc"*) לאחר קידוד הופמן.

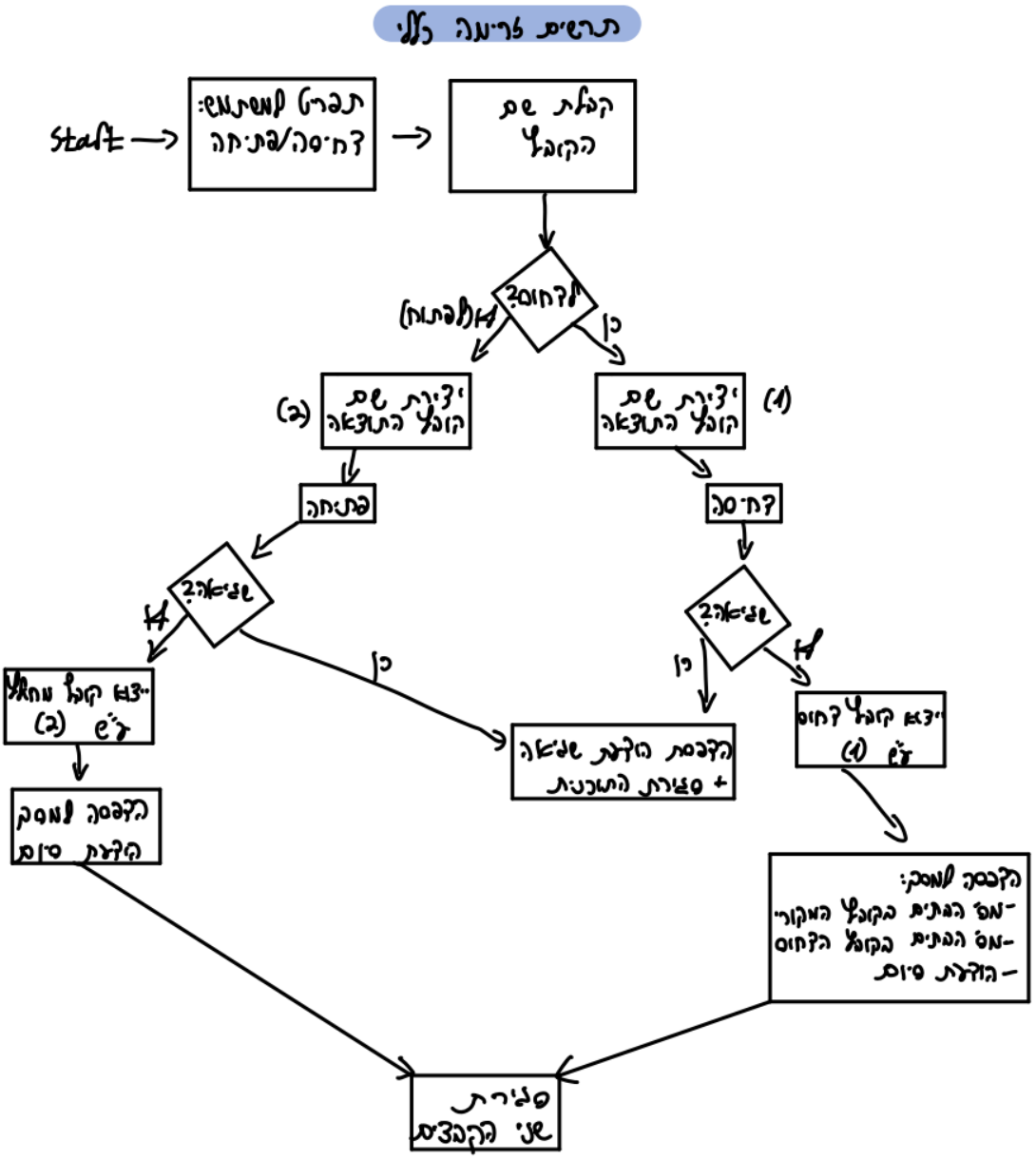
<u>פתיחה:</u>

ראשית ניצור את קובץ התוצאה, שמו יהיה כשם של הקובץ הדחוס (קיבלנו מהמשתמש) שנרצה לחלץ אך עם סיומת של *"txt"* במקום *"hf"*.

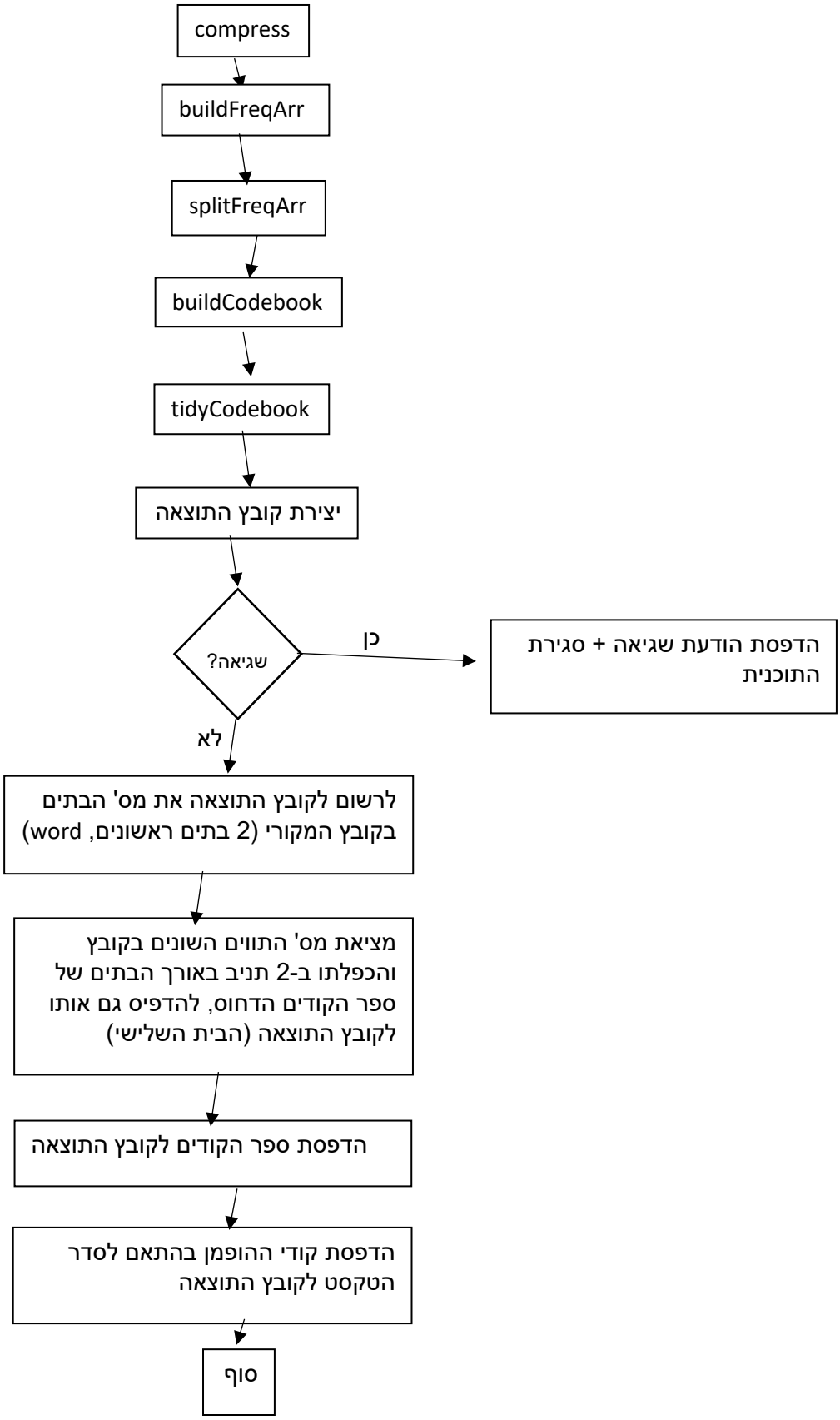
נפתח את הקובץ הדחוס ונקרא ראשית את ה-*header*.
נכניס את **שני הבתים הראשונים** (המייצגים את אורך הטקסט המקורי) למשתנה *[org_filecontent_len]* מסוג *word* (נשתמש בו מאוחר יותר).
בנוסף, נטען את ספר הקודים הדחוס למערך של ספר הקודים ב-*dataseg* (נדע מתי לעצור את קריאת ספר הקודים הדחוס באמצעות **הבית השלישי** בקובץ הדחוס – מס' הבתים בספר הקודים הדחוס).

כעת נתחיל את אלגו' הדחיסה, נדלג לחלק של קודי ההופמן בקובץ הדחוס (באמצעות הבית השלישי) ונקרא את תוכן הקובץ בית-בית.
נוסיף את הסיבית הנוכחית של הבית הנוכחי לאוגר עד כאשר ונתקל בהתאמה לתו מסויים (לפי קוד ההופמן שהאוגר מייצג) בספר הקודים (באמצעות הפרוצ' *findPattern*).
אם נתקלנו בהתאמה לתו כלשהו, נדפיסו לקובץ התוצאה. אחרת, נמשיך בלולאה עד לקבלת התאמה של קוד הופמן לתו מסויים בספר הקודים.
הלולאה תתבצע עד כאשר סך כל הבתים שרשמנו לקובץ התוצאה שווה בגודלו ל-*[org_filecontent_len]* (דהיינו, מס' הבתים בקובץ המקורי).

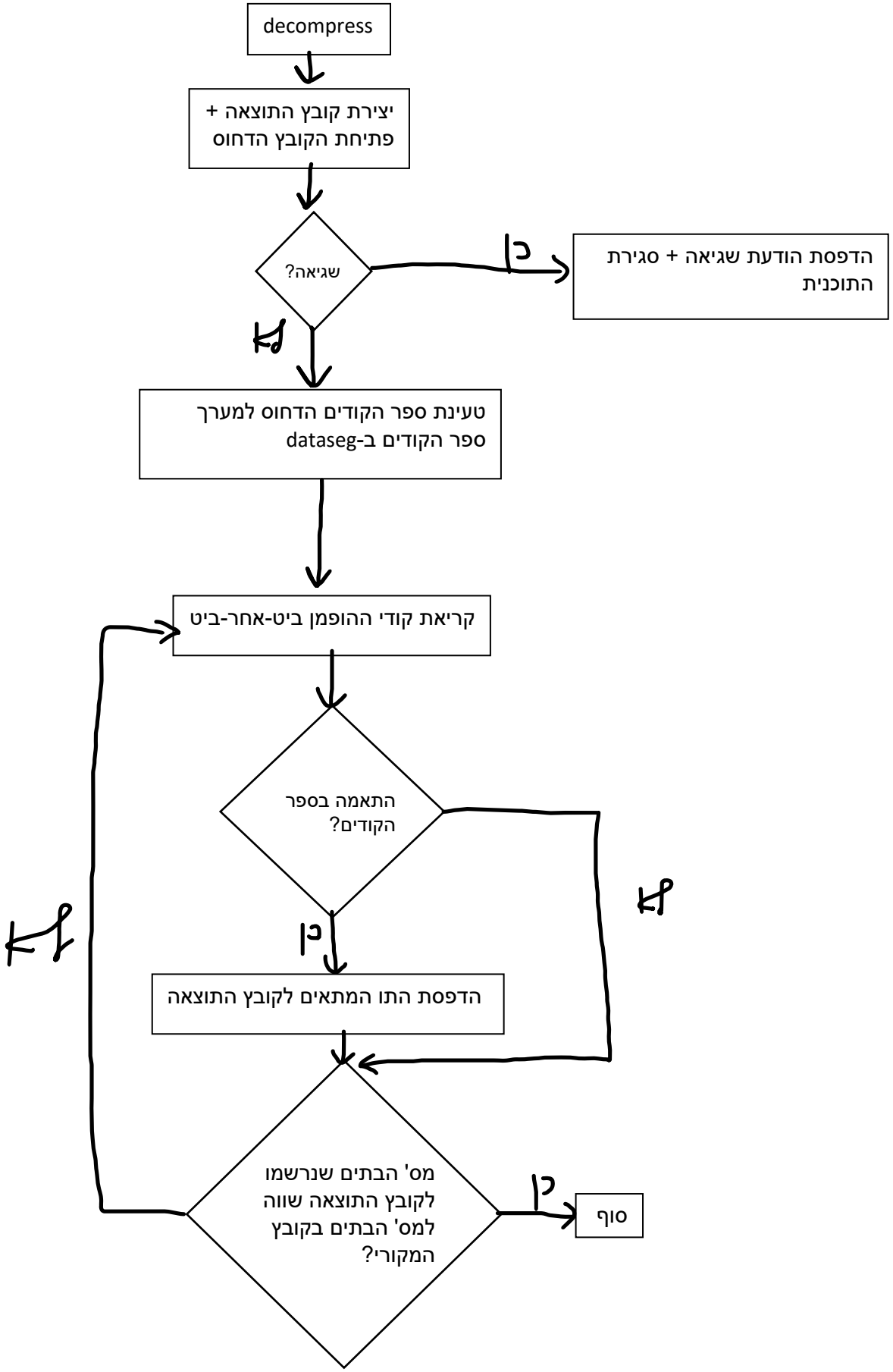
תרשימי זרימה



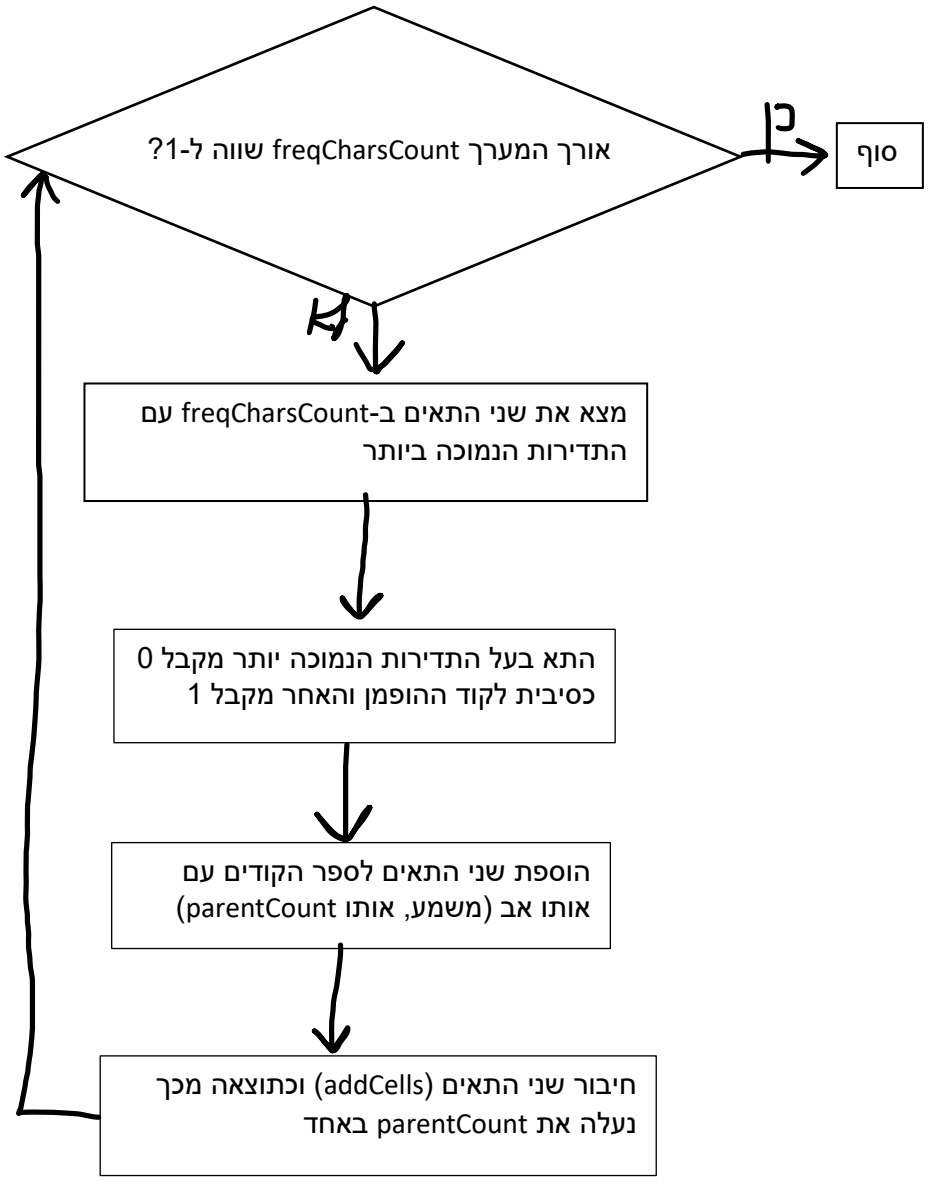
תרשים זרימה לפרוצ' *compress*



תרשים זרימה לפרוצ' *decompress*



תרשים זרימה לפרוצ' buildCodebook



הוראות הפעלה

לאחר התקנת הפרוייקט[6], נריץ מה-*DOSBox*:

cd x86-Lossless-Compression

(ייתכן ו-*DOSBox* ישנה את שם התיקייה לשם אחר, לכן כדאי לרשום *x86* ואז ללחוץ *tab* להשלמה אוטומטית)

רשמתי סקריפט *batch* (*auto.bat*) המקל על הליך הקומפילציה (בדומה ל-*Makefile*). במקום לרשום 2 או 3 פקודות ב-*DOSBox* לצרכי קימפול\דיבוג\הרצה\ניקוי, ניתן לרשום:

$\boldsymbol{auto\ FILENAME}$	$(compile)$
$\boldsymbol{auto\ FILENAME\ run}$	$(compile\ \&\ run)$
$\boldsymbol{auto\ FILENAME\ debug}$	$(compile\ \&\ debug)$
$\boldsymbol{auto\ FILENAME\ clean}$	$(delete\ all\ output\ files)$

אז נקמפל ונריץ בנקל את תוכנית הדחיסה:

$$auto\ compress\ run$$

המשתמש יתבקש להזין 1 או 2 (כאשר 1 משמעותו לדחוס קובץ ו-2 לחלץ קובץ דחוס).
לאחר מכן יתבקש להזין את שם הקובץ שברצונו לבצע את הפעולה שנבחרה.
קובץ התוצאה יהיה עם אותו שם של קובץ הקלט אך עם סיומת שונה (בהתאם לפעולה שנבחרה).

בשביל לבדוק את נכונות הדחיסה, רשמתי תוכנית פייטון (*test.py*) המדפיסה שורה-שורה את כל הבתים בקובץ הדחוס בבינארי ובהקסא (או אסקי).
התוכנית מקבלת כארגומנט משורת הפקודה את שם הקובץ הדחוס שברצוננו לקרוא את תוכנו. לדוגמה:

$$python\ test.py\ compressed.hf$$

אציין שאת התוכנית אני מריץ מה-*"host machine"* (במקרה שלי, *Windows 10*) המכיל את ההתקנה של *Python3.7* ולא מה-*DOSBox*.

[6] ניתן לשכפל מכאן: https://github.com/Tomer-Rubinstein/x86-Lossless-Compression.

דוגמת הרצה

נרצה לדחוס את הטקסט file2.txt (מצורף בתיקיית הפרוייקט) שתוכנו:

```
hello hello hello hello hello hello hello hello hello hello hello hello hello hello hello hello hello hello hello hello hello hello hello
hello hello hello hello hello hello hello hello hello hello hello hello hello hello hello hello hello hello hello hello hello hello hello
```

נריץ ונקמפל את קוד התוכנית באמצעות הסקריפט auto:

```
C:\X86-LO~1>auto compress run
Turbo Assembler  Version 4.1  Copyright (c) 1988, 1996 Borland International

Assembling file:   compress.asm
Error messages:    None
Warning messages:  None
Passes:            1
Remaining memory:  449k

Turbo Link  Version 7.1.30.1. Copyright (c) 1987, 1996 Borland International

Huffman Coding File Compression Program!
[1] Compress
[2] Decompress
(1 or 2) >> _
```

נקיש 1 לדחיסה ונזין את שם הקובץ:

```
Huffman Coding File Compression Program!
[1] Compress
[2] Decompress
(1 or 2) >> 1
Filename (include extension): file2.txt

Original file byte count: 264
Compressed file byte count: 100
[DONE] Compression
```

קיבלנו שהדחיסה שלנו חסכה כ-164 בתים ביחס לקובץ המקורי!
בנוסף קיבלנו את הקובץ הדחוס בשם: file2.hf

נניח ומחקנו את file2.txt.
נרצה לחלץ את תוכנו המקורי של file2.txt מתוך הקובץ הדחוס (file2.hf).
נריץ שוב את התוכנית, נבחר 2 לדחיסה ונזין את שם הקובץ הדחוס:

```
C:\X86-LO~1>compress

Huffman Coding File Compression Program!
[1] Compress
[2] Decompress
(1 or 2) >> 2
Filename (include extension): file2.hf

[DONE] Decompression
```

הפלא ופלא, קיבלנו את הקובץ המקורי!

```
hello hello hello hello hello hello hello hello hello hello hello hello hello hello hello hello hello hello hello hello hello hello hello
hello hello hello hello hello hello hello hello hello hello hello hello hello hello hello hello hello hello hello hello hello hello hello
```

פרק ד' – פירוט ה-interrupts

<u>ah=02, int 21h (הדפסת תו למסך)</u>
ב-start (לא פרוצ'):
להדפסת שורה חדשה ובחירת המשתמש (דחיסה-1 או פתיחה-2).
ב-buildFreqArr:
להדפסת שורה חדשה.

<u>ah=09, int 21h (הדפסת מחרוזת למסך)</u>
ב-start (לא פרוצ'):
להדפסת מחרוזת של מס' הבתים בקבצים והודעות סיום (של דחיסה ופתיחה)
ב-buildFreqArr:
להדפסת הודעת שגיאה בעת פתיחת קובץ.

<u>ah=3Eh, int 21h (סגירת קובץ בהינתן ה-handle באוגר bx)</u>
ב-start (לא פרוצ'):
סגירת שני הקבצים שנפתחו בפרוצ' compress או decompress.

<u>ah=3Dh, int 21h (פתיחת קובץ)</u>
ב-buildFreqArr:
פתיחת הקובץ שברצוננו לדחוס
ב-decompress:
פתיחת הקובץ הדחוס

<u>ah=3Fh, int 21h (קריאת קובץ)</u>
ב-buildFreqArr:
בשביל לשמור את תוכן הקובץ שברצוננו לדחוס ב-dataseg ולהשתמש בו בהמשך.
ב-decompress:
בשביל לשמור את תוכן הקובץ הדחוס ב-dataseg ולהשתמש בו בהמשך.

<u>ah=0Ah, int 21h (קבלת קלט ל-buffer)</u>
ב-start (לא פרוצ'):
לקבלת שם הקובץ שברצוננו לבצע עליו את הפעולה הנבחרה(דחיסה\פתיחה) ע"י המשתמש.

<u>ah=4Ch, int 21h (סגירת התוכנית)</u>
ב-exit (לא פרוצ'):
בשביל שנוכל לצאת מהתוכנית ולא להישאב בריקוד אל האינסוף ובשביל שנוכל לקפוץ ל-label של exit אם ניתקל בשגיאה שתמנע מאיתנו את המשך הרצת התוכנית.

<u>ah=40h, int 21h (לרשום לקובץ)</u>
ב-outputByte:
בשביל לייצא לקובץ byte **יחיד** מה-dataseg בשם byteToWrite.

<u>ah=3Ch, int 21h (יצירת קובץ)</u>
ב-compress:
בשביל ליצור את קובץ התוצאה (הקובץ הדחוס)
ב-decompress:
בשביל ליצור את קובץ התוצאה (הקובץ המקורי, המחולץ מקובץ דחוס נתון)