7

נושא 2 - אלגוריתמים חמדניים

אלגוריתמים חמדניים הם אלגוריתמים שמנסים למצוא פתרון אופטימלי לבעיה כלשהי. באלגוריתמים אלו אנו עוברים על כל מבנה הנתונים כך שבכל איטרציה (אפשר גם ברקורסיה) הבעיה מצטמצמת מעט. האסטרטגיה המרכזית היא שבכל צומת ניקח את האפשרות הטובה ביותר מבחינה מקומית, בלי להתחשב בהשפעה של ההחלטה על המשך הפתרון. כדי להוכיח אופטימליות באלגוריתם חמדני צריך להראות שכל בחירה (חמדנית) היא חלק מפתרון אופטימלי כלשהו. פעמים שהבחירה החמדנית אכן תהיה הפתרון האופטימלי ופעמים שלא. נציג ונדון בארבעה אלגוריתמים חמדניים, וננתח האם הם אופטימליים.

1) החלפת עודף

א) תיאור הבעיה: נתון סכום n שאנו צריכים להחזיר בעודף, כאשר יש לנו סט S של מטבעות קבועים מהם אנו יכולים להשתמש. מטרתנו היא להחזיר את העודף במדויק עם המספר הנמוך ביותר של מטבעות.

ב) הבחירה החמדנית: מכיוון שאנו רוצים את המספר הכי קטן, נבחר בכל צעד את המטבע בעל הערך הגדול ביותר, ונקטין את n בגודל זה. כאשר לא נוכל כבר להכניס מטבע זה, ננסה להכניס את המטבע הבא הכי גדול. וכן הלאה עד שנקבל את הסכום n.

```
coinChagne(n) {
    S = set of all coins
    A = 0    // number of coins we use
    while n>0 {
        c = maximus coin in S
        A = A + ⌊n/c⌋      // ⌊⌋ - round downward
        n = n - ⌊n/c⌋
        S = S - {c}
    }
} return A
```

ג) אופטימליות: אלגוריתם זה לא אופטימלי. לדוגמא, עבור $n=8$ ו-$S=\{5,4,2,1\}$ הפתרון שיחזיר האלגוריתם הוא $A=3$ ($5+2+1$), כאשר הפתרון האופטימלי הוא $A=2$ ($4+4$).

(8)

2)

בעיית תרמיל הגב

א) **תיאור הבעיה:** נתונים n עצמים $G_1, G_2, \ldots, G_n$, כאשר לכל עצם G_i מוגדר משקל העצם w_i וערכו v_i. לנו יש תרמיל גב שמסוגל להכיל עד משקל של C. נסמן ב-f_i את אחוז הכמות שלקחנו מ-G_i, כלומר $0 \le f_i \le 1$ כאשר $f_i = 0$ אומר שלא לקחנו כלל את העצם ו- $f_i = 1$ אומר שלקחנו את כל העצם.
מטרתנו היא למלא את השק בעצמים כך שלא נחרוג מהקיבולת וערכו יהיה מקסימלי.
או בניסוח מתמטי - ש-$\sum_{i=1}^{n} f_i \cdot v_i$ מקסימלי וגם $\sum_{i=1}^{n} f_i \cdot w_i \le C$.

ב) **הבחירה החמדנית:** עבור כל עצם G_i נחשב את הערך חלקי המשקל $\frac{v_i}{w_i}$. בכל בחירה נבחר את העצם שלו $\frac{v_i}{w_i}$ הוא הגדול ביותר, עד שהשק יתמלא ולא יהיה ניתן להכניס עוד עצמים.

```
KnapSack Problem (c) {
    S = set of all vi/wi
    V = 0        // value of the knapsack
    while c > 0 {
        i = index of maximus value in S
        if (wi ≤ c)    V += vi ;  c -= wi
        else           V += vi · c/wi ,  c = 0
    }
    return V
}
```

ג) **אופטימליות:** האופטימליות של אלגוריתם זה תלויה בשאלה האם ניתן לקחת חלק מעצם (כמו לדוגמא בנוזלים) או שמוכרחים לקחת את כולו, כלומר $f_i = 0 \backslash 1$.
במידה וניתן לקחת חלק מעצם אזי אלגוריתם זה אופטימלי, כי בכל בחירה אנו לוקחים את המקסימום האפשרי מהעצם עם הערך הכי גבוה ביחס למשקל. אך אם לא ניתן לקחת חלק, אלגוריתם זה לא אופטימלי. נביא דוגמא נגדית:
עבור $C = 50$ ועבור שלושת האיברים הנתונים בטבלה:

	Item 1	Item 2	Item 3
Value	60	100	120
Weight	10	20	30
V/W	6	5	4

האלגוריתם יחזיר $V = 160$ (איברים 1 ו-2) בעוד הפתרון האופטימלי הוא $V = 220$ (איברים 2 ו-3).

בס"ד

(9)

3) בע"ת הפעילויות

א) <u>תיאור הבעיה:</u> נתונים n פעילויות $S=\{1,2,\dots,n\}$ כך שלכל פעולות יש זמן התחלה s_i וזמן סיום f_i, המקיימים $s_i<f_i$. מטרתנו לבחור מספר מקסימלי של פעילויות כך שאף פעולות לא חופפת פעולות אחרת, כלומר לכל שני פעילויות i,j מתקיים (בלי הגבלת הכלליות): $[s_i,f_i)<[s_j,f_j)$.

ב) <u>הבחירה החמדנית:</u> נמיין את כל הפעילויות לפי זמן הסיום הכי מוקדם לזמן הסיום הכי מאוחר, בכל בחירה נרצה לבחור את הפעילות עם זמן הסיום הכי מוקדם, ולאחר מכן למחוק את כל הפעילויות שחופפות עם פעולות זו. לשם כך קודם נכניס את הפעילות הראשונה ב-S (כשהוא ממוין) אל מערך A, ונסמן $j=1$. לאחר מכן נעבור על כל שאר הפעילויות ב-S, אם נמצא פעולות i כך שזמן ההתחלה שלה גדול מזמן הסיום של הפעילות האחרונה שהכנסנו $s_i \geq f_j$, נכניס גם פעולות זו ל-A, ונקדם את j.

```
Activity_selector(S) {
    sort S from early finish time to latest
    A = {1}
    j=1
    for (i=2 to n) {
        if (si ≥ fj)
            A = A U {i}
            j = i
    }
    return A
}
```

ג) <u>אופטימליות:</u> נוכיח שאלגוריתם זה אכן אופטימלי. יהי G הפתרון של האלגוריתם $G=\{1,\dots\}$ ויהי A פתרון אופטימלי כלשהי $A=\{k,\dots\}$. מטרתנו להוכיח $|G|=|A|$, אך קודם נוכיח שהבחירה הראשונה 1 לא מפריעה לאופטימליות. במידה ו-$k=1$ סיימנו, לכן נניח $k\neq 1$. נגדיר פתרון חדש B שבואה ל-A אך יש בו 1: $B=\{A-\{k\}\}\cup\{1\}$. בהכרח $|B|=|A|$ וגם אין בו חפיפות מכיוון ש-k הפעילות הראשונה ב-A, ו-1 בוודאי נגמר לפני k. כעת נצמצם את הבעיה: אם A פתרון ל-S נגדיר $A'=A-\{k\}$ פתרון אופטימלי ל-$S'=\{i\in S: s_i\geq f_1\}$. באופן דומה נוכל להחליף את הפעילות הראשונה ב-A' בפעילות השנייה ע-G, כך שכמות הפעילויות תישאר ולא יהיו חפיפות. אם נעשה זאת עבור כל פעולות ב-A נקבל לבסוף את G. נמצא כי $|G|=|A|$ ולכן גם G אופטימלי.

<u>חשוב:</u>
כדי להוכיח אופטימליות באלגוריתם חמדני, צריך להראות שכל בחירה היא חלק מפתרון אופטימלי כלשהו.

12/11 (10) בס"ד

4) אלגוריתם הפמן

א) <u>תיאור הבעיה</u>: בקובץ טקסט כאשר אנו רוצים לשמור את המידע יש להמיר את כל התווים בקובץ לקוד בינארי. הדרך הפשוטה היא להמיר את כל התווים באמצעות טבלת ASCII, אמנם שיטה זו אינה יעילה מבחינת נפח האחסון שהיא דורשת, שכן כל תו מקבל 8 ספרות, ועדיף היה להקצות קוד קצר לתווים היותר נפוצים, ובכך לדחוס את המידע באחסון קטן יותר.

אנו מחפשים אלגוריתם שעבור n תווים, כאשר לכל תו i ישנה שכיחות w_i, האלגוריתם יחזיר קוד c_i באורך l_i שייצג בינארית את אותו תו. קידוד כזה צריך לקיים את התכונות הבאות:

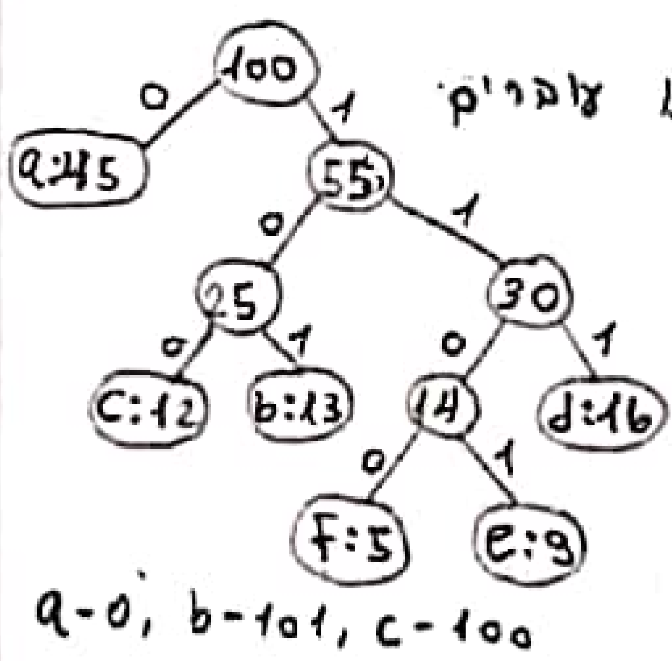
(1) <u>אין איבוד מידע</u> - כל תו מקבל קידוד.

(2) <u>קוד חסר רישות</u> - הוא קוד שבו אין שני תווים שהקידוד של אחת מהם הוא רישא של הקידוד של השנייה. לדוגמא: $e=110$, $f=1101$, e הוא רישא של f ולכן זהו לא קוד חסר רישות. קוד חסר רישות ניתן לפענח אותו בצורה יחידה ומהירה, מפני שעוברים על כל הקוד, וכשמוצאים קידוד ממירים אותו, וממשיכים במעבר על הקוד מה שלא ניתן לעשות בקוד שאינו חסר רישות.

$1100 = f$ או $1100 = ea$

(3) <u>חיסכון באחסון</u> - התווים עם השכיחות הכי גבוהה יהיו עם הקידוד הכי קצר, ואילו התווים עם שכיחות נמוכה יהיו עם קוד ארוך. ניתן לייצג דרישה זו בנוסחה מתמטית בכך שהסכום $\sum_{i=1}^{n} w_i \cdot l_i$ יהיה מינימלי. כלומר שסכום שכיחות כל תו כפול אורך הקידוד יהיה מינימלי. קוד כזה נקרא "קוד בעל יתירות מינימלית" (minimal redundancy code).

ב) <u>תיאור הקידוד בעץ בינארי</u>: ניתן לייצג קוד חסר רישות בעץ בינארי. אנו עוברים על הקידוד והעץ המתקבל. בכל פעם שמופיע 0 - פונים שמאלה, וכשמופיע 1 - פונים ימינה, כך עד שנגיע לעלה. כל עלה מייצג קידוד של תו אחר הנוצר מהדרך שעברנו מהשורש ועד העלה. בכל פעם שנגיע לעלה נמיר את הקוד עד כה לתו שבאותו עלה, ונחזור לשורש. שאר הצמתים בעץ שאינם עלים מחזיקים את סכום השכיחויות של ילדיהם.



a-0, b-101, c-100
d-111, e-1101, f-1100

כדי שהעץ יהיה אופטימלי הוא חייב להיות מלא, כלומר לכל צומת פנימית יש שני ילדים. מפני שאם היה צומת פנימית עם ילד אחד, ניתן היה להוריד אחד מהם ובכך לחסוך ביט אחד בקידוד. עץ כזה לא מאוזן, מפני שאנחנו לא מעוניין גובה העץ אלא רק שהסכום $\sum_{i=1}^{n} w_i \cdot l_i$ יהיה מינימלי.

(11)

1) <u>תיאור אלגוריתם הפמן:</u> נסתכל על התווים כאילו כל תו הוא עץ בינארי עם איבר אחד בלבד. נכניס את כל התווים (שורשי העצים) למערך ונמיין אותם לפי השכיחות הקטנה לשכיחות הגבוהה, כך שהמערך שלנו יהיה עץ ערימה מינימלי (MinHeap). אלגוריתם הפמן הוא אלגוריתם חמדני. הבחירה החמדנית היא שבכל פעם אנו לוקחים את שני התווים מהערימה עם השכיחות הכי נמוכה ויוצרים צומת חדש כך ששני התווים יהיו בניו, כעת נתייחס לצומת החדש כאל תו חדש שמספר המופעים שלו הוא סכום המופעים של שני בניו, ונחזיר צומת זאת לערימה במקום המתאים. כך קיבלנו צמצום של הבעיה מ-n תווים ל-n-1 תווים. נחזור על התהליך עד שנישאר עם צומת אחת בלבד. אנחנו בעצם בונים את העץ מלמטה למעלה כך שהתווים עם השכיחות הנמוכה יהיו בתחתית העץ ולהם יהיו קידוד ארוך, ואילו התווים עם השכיחות הגבוהה יהיו בראש העץ עם קידוד קצר.

במצגת 3 יש שקופית 9 הדגמה שלב אחר שלב של בניית עץ בינארי לפי אלגוריתם הפמן.

```
huffman(Σ){
    n = |Σ|   // number of charecters
    Q = MinHeap(Σ)                        // O(n)
    For(i=1 to n-1){
        create Node(z)
        z.left = x = Extract-Min(Q)       // O(log n)
        z.right = y = Extract-Min(Q)      // O(log n)
        z.key = x.key + y.key
        Q.insert(z)                       // O(log n)
    }
    return Extract-Min(Q)
}
```