שאלה 1 סעיף ב'

```
50  ###Part B###
51  def reverse_k_first_elements(A=list,k=int):
52      if len(A) == 0: #if the list is empty, for every k the list will remain the same
53          return A
54      if k>len(A):
55          raise ValueError('k larger than number of values in A')
56      if k==0:
57          return A #no need to make changes when k is 0
58      q_lst = Queue()
59      for val in A[:k]:
60          q_lst.enqueue(val) #putting first k elements in queue
61          A.remove(val) #removing them from list
62      while not q_lst.empty():
63          cur = q_lst.dequeue() #removing FIFO method from queue
64          A.insert(0,cur) #inserting in reverse order
65      return A #List with the defined order
```

(2) **שורות 52-58** פעולות בסיסיות (הצבה, השוואה), פעולות לוגיות: עבור C קבוע

$$C(\theta(1))$$

שורות 59-61 לולאת *for* אשר רצה מהאיבר הראשון עד האיבר ה*K* , נשים לב שבמקרה הכי גרוע, *k= len(A)* כלומר יש מעבר על כל איברי הרשימה, סה"כ:

$$O(n)$$

בתוך הלולאה, נשתמש בפעולות שהגדרנו במחלקה, שכל אחת מהן מורכבת מפעולות בסיסיות ולוגיות, לכן סה"כ בתוך כל איטרציה מתבצעת:

$$C(\theta(1))$$

שורות 62-64 לולאת *while* אשר רצה על כל האיברים בתור (*K* איברים), נשים לב שבמקרה הכי גרוע, *k= len(A)* כלומר התור מכיל את כל איברי הרשימה, סה"כ:

$$O(n)$$

בתוך הלולאה, נשתמש בפעולות שהגדרנו במחלקה, שכל אחת מהן מורכבת מפעולות בסיסיות ולוגיות, לכן סה"כ בתוך כל איטרציה מתבצעת:

$$C(\theta(1))$$

בסה"כ אם נסכום את כל הסיבוכיויות, נקבל שזמן הריצה הוא-

$$O(n)$$

שאלה 2

(2)

הרעיון מאחורי האלגוריתם שבנינו הוא לסכום תמיד את האיבר המינימלי לסכום שכבר קיים לנו. כלומר, בהתחלה נסכום את שני האיברים המינימליים, לסכום זה נוסיף את האיבר המינימלי הנוכחי (לאחר הורדת השניים הראשונים) וכך הלאה.

זה מבטיח לנו את הפתרון האופטימלי משום שכל פעם מתבצעת סכימה של ערכים מינימליים. הקפדה על סכימה בצורה זו תוביל לכך שכל פעם הסכום הבא יהיה המינימלי. (עד שלבסוף נגיע לסכימה הסופית שתהיה אופטימלית- הסכום הקטן ביותר).

על מנת לשמר את הזמן שלקח לשושנה לאגד את הסכומים עד לרגע זה, עלינו לשים לב לחוקיות הבאה:

ברשימת מטבעות של 1,2,3,4 מתבצעות הפעולות:

$$1+2$$

$$1+2+3$$

$$1+2+3+4$$

כלומר, האיחוד הראשון (שני המינימליים הראשונים) נספרים 3 פעמים (אורך הרשימה המקורית פחות 1), האיחוד השני (+3) מתבצע פעמיים (אורך הערימה לאחר שהורדנו ממנה את 2 המינימליים), והאיחוד האחרון (+4) מתבצע פעם אחת (אורך הערימה לאחר שהורדנו את כל המינימליים למעט האחרון).

ולכן באלגוריתם הכפלנו את הסכום הראשוני (שני המינימליים) באורך הרשימה המקורית פחות אחד, וכל אחד מהחיבורים הבאים הכפלנו באורך הערימה (לפני ההוצאה של החיבור הנוכחי).

(3)

```
65  def optimal_coin_collector(coins=list):
66      n = len(coins)
67      heap = Heap(coins)
68      cur = heap.delete_min()
69      cur += heap.delete_min() #first union
70      cur *= n-1
71      while heap.size > 0:
72          cur += heap.size * heap.delete_min()
73      return cur #The time is takes to create al the merges
```

שורות 66,70,73: פעולות בסיסיות, סה"כ $O(1)$

שורה 67: בנייה של ערימה מינימלית (*min heap*), בתוך ה *init* מתבצע *build heap* שסיבוכיותו היא $O(n)$

שורות 68,69: קריאה למתודה *delete min* פעמיים, $2 \times O(\log(n))$

שורות 71,72: הלולאה תתבצע n פעמים (עד שהערימה תהיה ריקה) ובתוך הלולאה יש קריאה למתודה *delete min* – סה"כ $O(n\log(n)) + O(1)$ (פעולה בסיסית בכניסה ללולאה)

סה"כ הסיבוכיות היא $O(n\log(n))$.

(4)

קוד האפמן הינו שיטה לקידוד סימנים, ללא אובדן נתונים (מכונה אלגוריתם "חמדני")

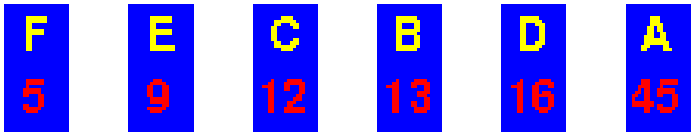
קוד המספק דחיסת נתונים מירבית - כלומר, מאחסן את הסימנים במספר מזערי של *bits*.

האלגוריתם מתבסס על אורך משתנה לכל סימן, עפ״י שכיחותו, כך שסימן נפוץ יותר ייוצג ע״י מספר קטן של *bits*.

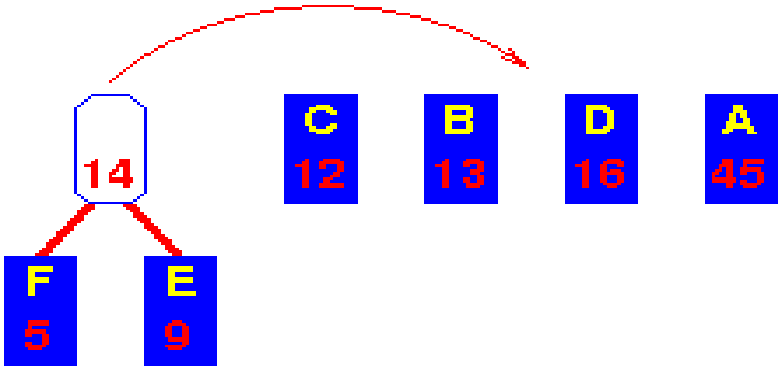
בין השימושים שלו: דחיסת טקסטים, תמונות או קבצי קול.

<u>דוגמה לשימוש באלגוריתם: דחיסת טקסט</u>

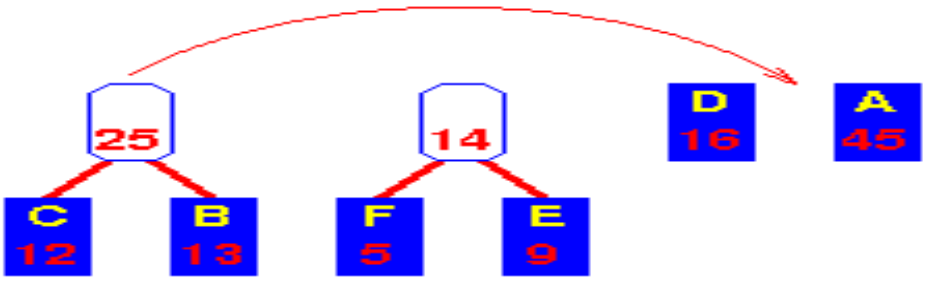
הנתונים ההתחלתיים ממויינים על פי סדר שכיחות:



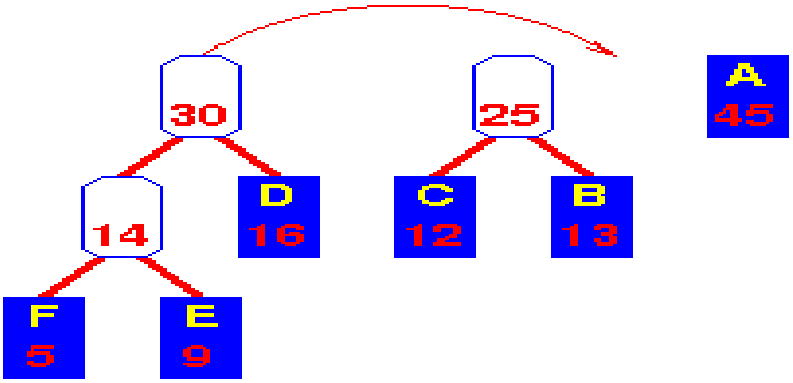
בכל צעד נשלב יחד את שני העצים בעלי המשקל הנמוך ביותר. *E* ו-*F* משולבים יחד ויוצרים תת עץ במשקל 14.



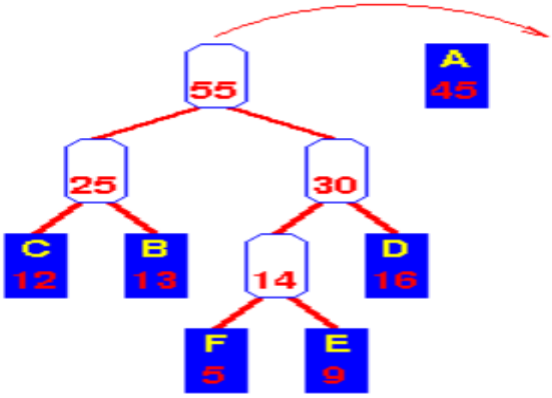
בעלי המשקל הנמוך ביותר, *B* ו-*C*, משולבים יחד ויוצרים תת עץ במשקל 25.



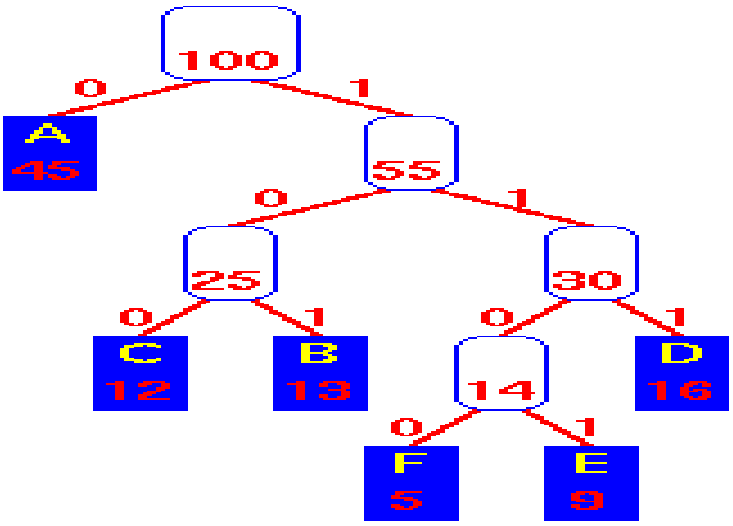
כעת, תת העץ במשקל 14 (המינימלי מבין תתי העצים הקיימים) ו-D משולבים יחד ויוצרים תת עץ במשקל 30.



תתי העצים במשקל 25,30 הם אלו בעלי המשקל הנמוך ביותר. לכן, נשלבם ונקבל את המשקל 55.



לבסוף, נשלב את תת העץ במשקל 55 עם העלה במשקל 45 ובכך ניצור עץ במשקל 100.



הערות: 1. נשים לב כי בכל שלב בו יוצרים תת עץ חדש האלגוריתם מעבירו למקומו הנכון, כלומר בצורה שבה ישמר סדר השכיחות.

2. בכל שלב הענפים מחולקים כך שהענף המוביל לבן הימני יסומן ב-1 והשמאלי ב-0.
באמצעות הקידוד המסודר בצורה בינארית, כל תו יקבל קידומת ייחודית. הקידוד עבור כל תו מתקבל על ידי מעבר על העץ מהשורש ועד לתו (לדוגמה התו E מתקבל על ידי ימינה, ימינה, שמאלה, ימינה- כלומר 1101).

שאלה 3

	Sheet 1	Sheet 2	Sheet 3
Mod, Chain	1.0672268907563025	1.0	9.470588235294118
Mod, Quadratic Probing	2.0	1.0	14.487394957983193
Mod, Double Hashing	1.773109243697479	1.0	3.2016806722689077
Multiplication, Chain	1.0756302521008403	1.0	1.0168067226890756
Multiplication, Quadratic Probing	2.319327731092437	3.2941176470588234	1.7647058823529411
Multiplication, Double Hashing	2.0840336134453783	1.7899159663865547	1.7563025210084033

מצ"ב פלט של "מדד היעילות"

(ה)

נשים לב שמספר הרשומות תמיד שווה, בכל אחד מהסט נתונים יש לנו 119 רשומות. כלומר מספר הפעולות הקטן ביותר ייתן את היעילות הטובה ביותר. ניתן לראות כי ערך מדד היעילות הטוב ביותר (כלומר, מספר הפעולות הקטן ביותר) הוא 1.0.

התייחסנו לעמודות תעודות הזהות כ keys ולעמודות השמות כ values.

Sheet 1: נשים לב כי טווח המפתחות נע בין 217785088 לבין 334906799 בסדר לא קבוע אך עולה לכל מפתח (חוץ מחריגה אחת של מפתח 556720506). נבחין כי כאשר בחרנו בשיטת גיבוב מסוג שרשור, בין אם בחלוקה או כפל, מדד היעילות נשמר פחות או יותר וכן הינו מדד היעילות הטוב ביותר עבור סט נתונים זה. ניתן להסביר זאת על ידי כך שגם בשיטה הריבועית וגם בשיטת הגיבוב הכפול מתבצעות פעולות כגון כפל והעלאה בחזקה לעומת שיטת השרשור שלא משתמשת בפעולות כאלה- זאת בדומה לסדר עלייתם של המפתחות (עלייה מתונה ולא קיצונית למרות היותה לא קבועה).

Sheet 2: ניתן לראות כי טווח המפתחות עולה בסדר קבוע (בקפיצות של אחד). רואים באופן מובהק, שלרוב טבלאות הגיבוב זהו סט הנתונים בעל מדד היעילות הטוב ביותר. נסיק מכאן, שיש השפעה משמעותית של אופי הנתונים הנקלטים על היעילות של הגיבוב. כשמשתמשים בחלוקה, ובמקרה שלנו במודולו 149- כל עוד מכניסים מפתח הקטן יותר מ-149 נקבל את המפתח עצמו. כלומר, בכל מקרה שבו משתמשים בסט נתונים זה עם חלוקה, לא תהיינה התנגשויות כלל. משמע, מספר הפעולות יהיה מינימלי. בנוסף, בדומה להסבר על Sheet 1,

השוני בין ערכי היעילות של סט נתונים זה, מוסברת על ידי כך ששימוש בשרשור מתאים ויעיל ביחס למפתחות ללא שינויים קיצונים בין אחד לשני (כמו במקרה זה) ולכן אנחנו רואים את ערך היעילות הטוב ביותר גם כאשר משתמשים בהכפלה עם שרשור.

Sheet 3: נשים לב שהמפתחות מחולקים לשישיות, בכל שישייה יש עליה בסדר קבוע (בקפיצות של אחד) וכן בין כל שישייה קפיצה של בדיוק 149. כלומר, כאשר עושים מודולו 149 עם האיבר הראשון בכל אחת מהשישיות נקבל את אותו הערך (כלומר תהיה התנגשות ונשתמש בעוד פעולות) וכן לגבי שאר האיברים בכל אחת מהשישיות בהתאמה. התנגשויות אלו מסבירות מדוע ערכי מדד היעילות גבוהים יותר כאשר משתמשים בפונקציה גיבוב חלוקה בהשוואה לsheet1, sheet 2. כלומר פונקציות הגיבוב בשיטת גיבוב הכפל יעילה יותר עבור סט נתונים זה.

(ו)

Sheet 1: נשנה את הערך *m* להיות 211, וכן את גודל המערך. שינוי זה יוביל לשיפור ערך מדד היעילות כיוון ש211 הוא מספר ראשוני גדול מ149, כלומר יש טווח ערכים הרבה יותר גדול לערכו של [מפתח כלשהו%*m*], דבר המקטין את ההצטברויות המשניות.

Sheet 2: נשים לב כי ערך מדד היעילות הכי טוב בסט נתונים זה הוא 1.0. כמו כן, מדד היעילות הכי טוב באופן כללי גם הוא 1.0. זאת מכיוון שמדד היעילות מחושב ע"פ (מספר הפעולות / מספר הרשומות) ומספר הרשומות הוא מספר קבוע. כלומר, עבור הכנסת 119 איברים, 119 פעולות ייתן לנו את מדד היעילות הטוב ביותר (הערך המינימלי), כלומר פעולה אחת עבור כל הכנסת איבר. זאת אומרת, לא ניתן לשפר את פעולת הגיבוב היעילה ביותר עבור סט נתונים זה, כי היא היעילה ביותר באופן כללי.

Sheet 3: בדומה ל*sheet 1*, נשנה את הערך של *m* להיות מספר ראשוני גדול ממנו (לדוגמה 211). הסבר שקול לסט נתונים הראשון.