השוואת הפתרונות השונים שראינו לבעיית ה repeating substring

קטע הקוד הבא משווה את זמני הריצה של repeat_naive, repeat_hash1, repeat_hash2 על מחרוזות אקראיות ואורכי תת מחרוזת שונים.

(למשל בדוגמא הבאה משווה את זמני הריצה של הפונקציות על מחרוזת אקראית באורך 1000 ואורך תת מחרוזת l=10)

```
# "competition" between the 3 solutions
import time
str_len=1000
st=gen_str(str_len)
l=10
print("str_len=",str_len, "repeating substring len=",l)
for f in [repeat_naive,repeat_hash1,repeat_hash2]:
   t0=time.clock()
   res=f(st, l)
   t1=time.clock()
   print(f.__name__, t1-t0, "found?",res)
```

ראינו בכיתה שעבור מחרוזת באורך n וגודל תת מחרוזת l, סיבוכיות הזמן של repeat_naive הינה $O(l(n-l)^2)$. סיבוכיות הזמן הממוצעת של repeat_hash1 הינה $O(l(n-l))$, אך ב worst-case סיבוכיות הזמן של repeat_hash1 היא $O(l(n-l)^2)$, כלומר כמו של הפתרון הנאיבי.

סיבוכיות repeat_hash2 שקולה לזו של repeat_hash1. ההבדל בין שני פתרונות אלה הוא ש repeat_hash1 משתמש בטבלאות hash מהטיפוס שראינו בהרצאה, בעוד שהפתרון השני משתמש ב set של פייתון (מבנה יעיל שמבוסס על טבלת hash ומטפל בהתנגשויות בגישת open addressing).

מהרצת הקוד הנ"ל מתקבל הפלט הבא:

```
>>>
str_len= 1000 repeating substring len= 10
repeat_naive 0.29123701471846164 found? False
repeat_hash1 0.0026285363761566205 found? False
repeat_hash2 0.0007428901426530521 found? False
```

באופן לא מפתיע זמן הריצה של repeat_naive גדול משמעותית משל שני הפתרונות האחרים. כמו כן זמן הריצה של repeat_hash1 ארוך משל repeat_hash2.

נריץ שוב עבור מחרוזת אקראית באורך גדול פי 2 (כלומר באורך 2000).
מאחר ש n גדל פי 2, נצפה שזמן הריצה של הפתרון הנאיבי יגדל פי 4 בערך בעוד שזמן הריצה של repeat_hash1, repeat_hash2 יגדל פי 2 בהשוואה להרצה הקודמת.

```
>>>
str_len= 2000 repeating substring len= 10
repeat_naive 1.0280356160955322 found? False  #גדל פי 4 בערך
repeat_hash1 0.0055479920058389975 found? False  #גדל פי 2 בערך
repeat_hash2 0.001547358851859748 found? False  #גדל פי 2 בערך
```

כעת נריץ עבור n= 8000 ו l=2. סביר שבמקרה כזה תמצא תת מחרוזת באורך 2 שחוזרת על עצמה די מהר.

```
>>>
str_len= 8000 repeating substring len= 2
repeat_naive 0.0005139441900819824 found? True
repeat_hash1 0.0028720875015641363 found? True
repeat_hash2 4.934180012307332e-05 found? True
```

שימו לב לתוצאות שהתקבלו כעת. repeat_hash1 איטי יותר מהפתרון הנאיבי משום שrepeat_hash1 יוצר בכל מקרה טבלה ריקה (בגודל n-l) ורק לאחר יצירת הטבלה מתחיל לעבור על תת המחרוזות ומגלה די מהר שיש חזרה. repat_hash2 מצד שני מהיר משמעותית משני הפתרונות האחרים (set נוצר בגודל התחלתי קטן מאד, וגדל על פי צורך בהתאם לכמות האיברים שמוכנסים לתוכו).

<u>השפעת גודל הטבלה על הפתרון שמשתמש במחלקה Hashtable (כלומר על repeat_hash1)</u>

הפונקציה הבאה מריצה את הפתרון של repeat_hash1, אך יוצרת טבלה בגודל נתון m.

```
def repeat_hash1_var_size(st, l, m=0):
    if m==0: #default hash table size is ~number of substrings to be inserted
        m=len(st)-l+1
    htable = Hashtable(m)
    for i in range(len(st)-l+1):
        if htable.find(st[i:i+l])==False:
            htable.insert(st[i:i+l])
        else:
            return True
    return False
```

עבור מחרוזת אקראית באורך 1000 ואורך תת מחרוזת 20, נריץ את repeat_hash1_var_size עם גדלי טבלה משתנים : 1, 10, 100, 1000, 1500, 10000, 100000

```
import time
str_len=1000
st=gen_str(str_len)
l=20
print("str_len=",str_len, "repeating substring len=",l)
for m in [1, 10, 100, 1000, 1500, 10000, 100000]:
    t0=time.clock()
    res=repeat_hash1_var_size(st, l, m)
    t1=time.clock()
    print(t1-t0, "found?",res, "table size=",m)
```

והנה התוצאות:

```
str_len= 1000 repeating substring len= 20
0.05543575782622579 found? False table size= 1
```

0.006257112373427881 found? False table size= 10
0.002820279328043296 found? False table size= 100
0.002444110510740921 found? False table size= 1000
0.0024362160969465807found? False table size= 1500
0.0038797096592494695 found? False table size= 10000
0.024170721434951803 found? False table size= 100000

ניתן לראות שעבור גודל טבלה ~ 1500 התקבל זמן הריצה המינימלי (זה קורה כאשר גודל הטבלה הוא בערך כמספר האיברים השונים שמוכנסים אליה, כלומר m~n).

עבור גדלים נמוכים מאוד של טבלה (למשל m=1, 10) זמן הריצה יהיה גבוה יחסית, כי יהיו הרבה התנגשויות (אורך הרשימה הממוצע יהיה גבוה).

עבור גדלים גבוהים מאוד של טבלה (למשל m=100000) זמן הריצה שוב יהיה גבוה, משום שכעת זמן יצירת הטבלה יהיה דומיננטי.