פרויקט סיום בקורס "תכנות פונקציונלי בסביבת ה - JVM":

gitHub:

הפרויקט נמצא כאן (https://github.com/ofirc/SuffixTrees/).

תיאור הבעיה:

א. מימוש עץ סייפות (suffix tree) עבור מחרוזת T בשפת Clojure.

ב. מתן מענה לשאילתות הבאות באמצעות עץ הסיפות:

1. האם תבניות P1,P2,… מוכלת במחרוזת T?

<u>סיבוכיות רצויה:</u> O(m) עבור כל תבנית Pi באורך m.

2. האם ואיפה מופיעות תבניות P1,P2,… במחרוזת T?

<u>סיבוכיות רצויה:</u> O(m) עבור כל תבנית Pi באורך m.

שאלות אלו עולות בהקשר יומיומי של שימוש אישי במחשב (ctrl+F) ולכן חשיבותה ברורה.

3. מציאת מספר ההופעות של דפוס מסוים: כמה פעמים מופיע תבנית P במחרוזת T?

<u>סיבוכיות רצויה:</u> O(m) עבור תבנית P באורך m.

4. מבין תתי המחרוזת שחוזרות מספר רב ביותר של פעמים בטקסט T, מיהי המחרוזת הארוכה ביותר?

למשל במילה "nanana" תתי המחרוזות שחוזרות הכי הרבה פעמים בטקסט (3 פעמים) הן: 'a', 'n', 'na' והארוכה ביותר מבינן היא "na" .

<u>סיבוכיות רצויה:</u> טובה מהטריוויאלית (שהיא n^3), ליניארית בגודל העץ (שחסום ב-n^2 עבור טקסט Tבאורך n).

הוראות הרצה:

1. יש לייבא את הפרויקט עצמו suffix.zip בעזרת eclipse .

הפרויקט מכיל את הקבצים הבאים: testing.clj , suffix.clj , agents.clj

2. טסטים:

2.1. תפעול קובצי הבדיקות:

- "testing.clj": קרא ל-REPL מתוך הקובץ
- "agents.clj": קרא ל-REPL והרץ (agent_example)

(<u>הערה</u>: הקובץ מדגים שימוש ב agents על מספר pattern'ים בו-זמנית)

2.2 הפרויקט עצמו: יש להפעיל את ה REPL מהקובץ הראשי suffix.clj

2.3 כדי לצפות ב AOT, ראה את הסעיף שמופיעה בהמשך שמפרט כיצד להשתמש ב jar files שמצורפים.

הקבצים הרלוונטיים: AOT.java וה- jar file.

<u>הערה</u>: ה jar file הוא קומפילציה של הפרויקט (באמצעות lein compile;lein uberjar)

3. שימוש בפונקציות: (אם ברצונך להפעיל את הפונקציות בעצמך מתוך הקובץ הראשי suffix.clj)

3.1 שימוש בפונקציות עבור תבנית בודדת:

- נסמן את הטקסט שעליו נבצע את השאילתות ב-word
- נסמן את התבנית הרצויה ב-sub
- נסמן את תוצאת הקריאה לפונקציה build_tree על word ב root

(ניתן גם לצפות בעץ ויזואלית בעזרת שמירות וקריאה לפונקציה inspect-tree כפי שהוסבר קודם לכן).

קרא לפונקציה הרצויה (מתוך הפונק' שמפורטות בעמ' 3-4) עם root ו-sub ותוכל לראת כפלט את התוצאה.

3.2 שימוש לפונקציה עבור מספר תבניות בשאילתא אחת:

- נסמן את הטקסט שעליו נבצע את השאילתות ב-word
- נסמן את אוסף התבניות הרצויות ב-seq

קרא לפונקציה הרצויה (מתוך הפונק' שמפורטות בעמ'4-5) ושמור תוצאתה: def ans = (some_lazy_query…)

לשם הצגת התוצאה ה-i יש להשתמש בפונקציה (nth i ans) (לחלופין שימוש בפונקציה ?realize ע"מ לדעת אילו מהתוצאות שוערכו ואילו טרם שוערכו).

תיאור הפתרון:

א. מימוש עץ הסיפות

עץ הסיפות בנוי מרשומות מקוננות (records):

```
(defrecord node [next idx_start cnt])
```

כאשר כל צומת בעץ ממומש בעזרת רשומה שמורכבת משני שדות :

next: hash-map שמכיל את כל המצביעים מהצומת הזה לכל הילדים שלו בעץ. המפתח ב-map הוא האות שצריך לקרוא בעץ ה-trie על מנת לעבור לצומת הבא, והערך הוא ה-node הבא.

idx_start: vector שמכיל את האינדקסים של כל הסיפות שניתן להגיע אליהן על ידי טיול בתת העץ של הצומת הזה. (האינדקס של סיפא כלשהי הוא אינדקס ההתחלה שלה בטקסט). דבר זה יאפשר לנו לענות על השאלה היכן בטקסט מופיעה תבנית כלשהי.

cnt: מונה שמחזיק את מספר הילדים בתת העץ של צומת כלשהו.

בניית העץ מתבצעת בפונקציה הבאה:

```
(defn build_tree [word] (…))
```

המקבלת את המחרוזת T ובונה את העץ על ידי הוספת כל הסיפות של T החל מהסיפא הארוכה ביותר (=המילה כולה) ועד הסיפא הקצרה ביותר (שמכילה אות אחת בלבד). הוספת סיפא S_i מהצורה $S_i=a_1a_2a_3\ldots a_n$ לעץ פירושה להתחיל מהשורש ולהסתכל בשדה ה-next שלו האם ה-key a_1 קיים במפה- אם הוא קיים, מחליפים את המפה שנמצאת ב-value של a_1 במפה שהיא מיזוג בין המפה הנוכחית ומפה שמכילה את המסלול עבור הסיפא $a_2a_3\ldots a_n$, בצורה רקורסיבית. אם הוא לא קיים, מוסיפים לכניסה של a_1 את תת העץ שמתאים לסיפא $a_2a_3\ldots a_n$, בצורה רקורסיבית .

בנוסף, בעת בניית העץ, עבור כל סיפא שמתווספת לעץ, מפעפעים את האינדקס של הסיפא במורד המסלול שלה מהשורש עד לעלה המתאים לה (מוסיפים אותו לוקטור ה-`idx_start` של הצמתים). זה יעזור לנו לענות על השאלה היכן בטקסט מופיעה תבנית כלשהי- אם נצליח לקרוא תבנית ונסיים בצומת כלשהו בעץ, כל האינדקסים שיופיעו בצומת זה בשדה idx_start הם האינדקסים של שהחל מהם בטקסט ניתן לקרוא את התבנית- למעשה התבנית היא רישא משותפת של כל הסייפות שמתחילות באינדקסים אלו.

בצורה זהה לכך מעדכנים את שדה cnt- בעת הוספת סיפא מגדילים ב-1 את ערכו של ה-cnt בכל הצמתים בדרך. דבר זה יאפשר לנו להחזיר את מספר המופעים של דפוס כלשהו P שמופיע בטקסט בזמן בלתי תלוי במספר המופעים (לא נצטרך לספור את מספר האיברים במערך idx_start אלא נענה ב-O(1) מרגע שהגענו לצומת המתאים ל-P).

ב. מענה על השאילתות:

הערה: השאילתות להלן מוסברות על קלט של תבנית אחת אולם קיימת לכל אחת מהן מקבילה בגרסת lazy-evaluation שמתאימה לקבלת מספר תבניות וטקסט בשאילתא אחת.

שאילתות עבור דפוס בודד:

שאילתא 1:

על מנת לענות על השאילתא "האם P מוכלת במחרוזת T" יש לקרוא לפונקציה הבאה:

```
(defn is_sub [root sub] (…))
```

עם הקלטים: שורש עץ הסיפות של T וה-sub שהוא התבנית P. הפונקציה עונה על השאילתא בעזרת קריאה לפונקציה build_tree שהוצגה מקודם, ואז מתבצע טיול על העץ בניסיון לקרוא את התבניות שקיבלנו על ידי הפונקציה הבאה:

```
(defn which_ind [root sub] (…) )
```

פונקציה זו מקבלת את שורשו של העץ ש-build_tree החזירה ותבנית כלשהי ב-sub, נניח בה"כ את P, כאשר $P=a_1a_2a_3\ldots a_n$ ומנסה לקרוא אותה בעץ על ידי חיפוש המפתח a_1 במפה בשדה ה-next של השורש- אם הוא לא קיים מחזירה false. אם הוא קיים ממשיכה לעקוב אחרי ה-value של מפתח זה. נמשיך כך עבור הצמתים הבאים בדרך, ואם סיימנו לקרוא את כל האותיות של P אזי נחזיר את הוקטור idx_start של הצומת האחרונה שביקרנו בה.

עץ סייפות הינו מקרה פרטי של trie ולכן קריאת כל Pi בעץ נעשית **בזמן ליניארי באורך התבנית**.

שאילתא 2:

על מנת לענות על השאילתא "האם ואיפה מופיעה תבנית P במחרוזת T? " נקרא לפונקציה שהוצגה קודם לכן:

```
(defn which_ind [root sub] (…) )
```

שאילתא 3:

על מנת לענות על השאילתא "כמה פעמים מופיע תבנית P במחרוזת T? " נקרא לפונקציה:

```
(defn how_many [root sub] (…))
```

עם הקלטים root- שורש העץ וsub- התבנית שאנו מחפשים. הפונקציה מתקדמת על העץ בניסיון לקרוא את התבנית וכשהיא מסיימת לקרוא אותה היא מחזירה את המונה cnt של הצומת האחרונה בטיול. אם לא הצלחנו לקרוא כל התבנית מחזירים 0. כפי שהוזכר קודם לכן, תחזוקת המונה cnt בכל אחד מהצמתים מאפשר לנו להחזיר את מספר המופעים של דפוס כלשהו בזמן בלתי תלוי במספר מופעיו בטקסט (לא נצטרך לספור את מספר האיברים במערך idx_start אלא נענה ב-O(1) מרגע שהגענו לצומת המתאים).

שאילתא 4:

על מנת לענות על השאילתא " מבין תתי המחרוזות שחוזרות מספר רב ביותר של פעמים בטקסט T, מיהי המחרוזת הארוכה ביותר? " נקרא לפונקציה הבאה:

```
(defn longest_repeating_substring [word] (…) )
```

עם הקלט word - המילה בה נחפש את תת המחרוזת החוזרת על עצמה באופן מקסימאלי . פונקציה זו בונה את העץ (קוראת ל-build_tree) ואז קוראת לפונקציית העזר get_max_child root . הפונקציה פועלת לפי הלוגיקה הבאה: עבור כל בן של השורש נמצא את הצומת (בן מדרגה כלשהי שלו) עם המספר המקסימאלי של ילדים. עם המצא צומת כזה נשחזר את המסלול שלו בחזרה אל השורש תוך כדי קריאת התווים שהובילו אליו (מציאת ה sub-string עצמו).

נשים לב כי התהליך רץ בלולאה הנעצרת באחד מן המקרים הבאים –

1. לבן של הצומת הנוכחי יש פחות ילדים מאשר לצומת הנוכחית.
2. לצומת הנוכחי יש יותר מבן אחד (ואז סימן שיש פיצול והמערך של העלים מצטמצם)

הערך המוחזר הינו ה substring שמופיע הכי הרבה פעמים בטקסט , אם ישנם כמה כאלה יוחזר זה שאורכו הוא הגדול ביותר , ואם לא קיימת תבנית שחוזרת על עצמה בטקסט יוחזר הטקסט עצמו.

<u>שאילתות מקבילות עבור מספר דפוסים בשאילתא אחת (כולן משתמשות ב-lazy evaluation):</u>

הקלטים עבור הפונקציות הבאות הם word – הטקסט, ו-col- הcollection שכל איבר בו הוא תבנית P_i:

1. (defn sub_lazy [word col] ())

הפונקציה יוצרת lazy sequence של ערכים בוליאניים עם התשובה האם התבנית ה- i מוכלת בטקסט.

2. (defn idx_lazy [word col] ())

הפונקציה יוצרת lazy sequence כך שכל אלמנט בה מכיל – אם התבנית נמצאת בעץ את אינדקסי ההתחלה שלה בטקסט, ואחרת nil.

3. (**defn** how_many_lazy [word col] ())

הפונקציה יוצרת lazy sequence כך שהערך ה-i בו מכיל את מספר ההופעות של התבנית ה-i בטקסט.

4. (**defn** longest_repeating_substring_lazy [word col] ())

הפונקציה יוצרת lazy sequence כך שהערך ה-i בו מכיל את המחרוזת הארוכה ביותר שחוזרת על עצמה הכי הרבה פעמים בתבנית ה-i בקלט. (כאן word מתפקד כ-dummy).

פתרונות חלופיים אפשריים שלא בחרנו בהם:

1. **בניית העץ כשכל צומת הוא vector :**

בחרנו שלא להשתמש בפתרון זה מאחר ששימוש בווקטורים היה מכריח אותנו לנקוט באחת מהגישות הבאות:

א. להקצות בכל צומת וקטור כגודל האי"ב –יקר מבחינת סיבוכיות מקום. בנוסף, שימוש ב-map מאפשר גם נזילות בבחירת האי"ב שעמו עובדים- כלומר העץ שלנו מותאם לא"ב עברי, מספרי, סיני,Unicode , ascii וכו' כאחד (ללא תלות בגודל האי"ב).

ב. להקצות בכל צומת וקטור בגודל הנדרש (כמספר הילדים של הצומת)- יקר מבחינת סיבוכיות זמן , כלומר שימוש בווקטור לא היה מאפשר לנו גישה לכל מידע על הצומת תתאפשר ב $O(1)$. בנוסף, כדי לשמור על מבנה נתונים תקין , בכל שינוי שהיה מבוצע בצומת היה עלינו לדאוג על שמירת הסדר הלקסיקוגרפי , דבר יקר גם הוא מבחינת סיבוכיות הזמן.

2. **מימוש העץ בעזרת zippers :** בחרנו שלא להשתמש באופציה זו מאחר וע"פ מאמרים שקראנו היה נראה כי שימוש ב zippers ידרוש סיבוכיות מקום הרבה יותר גדולה עבור הצמתים בעץ מאשר הפתרון שאנו הצענו (ניתן לראות דוגמה לכך במאמר בלינק הבא: http://yquem.inria.fr/~huet/PUBLIC/zip.pdf).

טכניקות של פרדיגמת התכנות הפונקציונלי שהפתרון משתמש בהן:

- ✓ **Immutability** (שורה מספר 82, פונק' build_tree, קובץ suffix.clj): ב-Clojure עיקרון זה בא לידי ביטוי על ידי השימוש במבנים שאינם ניתנים לשינוי שהשפה מציעה- בפרויקט שלנו, עץ הסיפות מומש תוך שימוש במבנים אלו- map, vector. הליך בניית העץ נעשה פעם אחת עבור טקסט מסוים T והעץ אינו משתנה לאחר שלב הבנייה, וכן העץ מתאים עבור שאילתא לכל מחרוזת P עליה נרצה לפעול. לכן מימוש העץ תוך שימוש במבנים ה-immutables שהשפה מציעה (map, vector) התאימה מאוד לפתרון.
- ✓ **Laziness** (שורה מספר 151, פונק' lazy_query, קובץ suffix.clj): עקרון זה בשפה מבטיח שלא ישוערכו חישובים שאין משתמשים בתוצאתם ובכך מאפשר יעילות חישובית. הפתרון שהצענו מוחזר כ-lazy sequence , כלומר שיערוך התשובה מתבצע רק מתי שבאמת צריך אותה. לפיכך, לאחר הכנסת שאילתא למערכת, התשובה עבור התבנית הראשונה תינתן באופן מידי ואילו התשובות עבור התבניות הבאות תינתן רק לאחר בקשה מפורשת מצד המשתמש (עבור התבנית ה-i המשתמש יבקש את ans[i]).
- ✓ **Referential transparency** & **Function purity**: כדי להבטיח שפונק' היא טהורה יש לוודא שני דברים: 1) עבור אותו סט של ערכים, ערך ההחזרה של הפונקציה יהיה זהה (כלומר לא קוראים state שהינו חיצוני לפונקציה), 2) אין תופעות-לוואי (לא משנים state שהינו חיצוני לפונקציה) אצלנו בקוד כל הפונקציה הן טהורות, שכן הן מסתמכות אך ורק על סימבולים שהם מקומיים לפונקציה. ידוע לנו ש pure --> referential transparency לכן הן גם referential transparent. הערה: כל הפונקציה הן pure מלבד הפונקציה is_sub_agt שמועברת לagents שאנחנו יוצרים, וזאת בכוונה כדי להדגים בקלות את

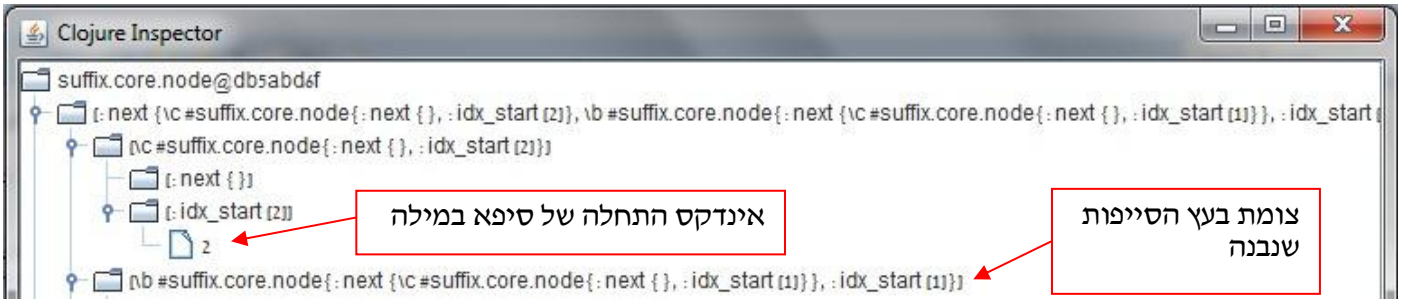
השימוש ב agents, שסדר ההדפסה אכן יכול להיות שונה, כתלות בזמן שבו הורצו ה agents. (שכן לו היינו מחכים לשערוך של כולם ואז היינו מדפיסים, הפלט היה דטרמיניסטי ולא מעניין לכשעצמו).

- ✓ **TCO- Tail Call Optimization** (הפונקציה get_str_path, מופיעה במספר שורה 182, קובץ suffix.clj): פונק' זו היא פונק' עזר למציאת "תת-הרצף שחוזר הכי הרבה בטקסט והארוך מבניהם" . הפונקציה משתמשת באופטימיזצית TCO : השורה האחרונה שהיא מבצעת היא החזרת הערך של הקריאה הבאה. דבר זה חוסך את המקום שה stack-frame'ים היו תופסים אילולא היו נפרשים בנפרד :

```
(get_str_path next_node cnt (str word ch_cur))
```

- ✓ **Clojure.Inspector:** מאפשר הצגה של עץ הסייפות שנבנה באופן יותר אינטואיטיבי מבחינה ויזואלית. לשם הפעלה יש לבצע ב REPL את התהליך המוצג :

כאשר תוצאת ההרצה תפתח חלון של javaScript מהצורה הבאה :



כמו כן, השתמשנו בספריה זו, בפונקציה inspect-tree בקובץ -"testing.clj", שורה 24.

- ✓ **High-order functions & Reusability** (שורה 151, פונק' lazy_query, קובץ suffix.clj): כל ארבעת השאילתות עבור sequences של pattern'ים מתבצעות בצורה יעילה. השאילתות הן : idx_lazy, sub_lazy, how_many_lazy, longest_repeating_substring_lazy. הן מתבצעות בצורה יעילה כי כולן מעבירות את הפונקציה שיש להפעיל על כל אחד מה-pattern-ים, לפונקציה lazy_query שהיא Higher-order-function, והיא יודעת כבר כיצד לבצע את הבאים :
  - בדיקות תקינות הקלט
  - יצור lazy sequence של הפעלת הפונקציה על כל אחת מן האיברים

ואז למעשה במקום לשכפל קוד 4 פעמים, בנינו את lazy_query פעם אחת בצורה חכמה, שתדע איזו פונק' יש להפעיל.

- ✓ **Concurrency using Agents:** נעשה כדי לחשב האם patterns הנמצאות ב sequence הן substring ב tree של מילה נתונה באופן מקבילי (מאחר וחישוב עבור כל patterns הינו בלתי תלוי בחישובי ה patterns האחרים).

בדומה ל Refs שמאפשרים לנו לבצע שינוי בצורה סינכרונית מתואמת של מספר אלמנטים,

הAgents מאפשרים לנו ביצוע של שינויים **אסינכרוניים** עבור אלמנטים בודדים. נדגים את השימוש בעזרתם כדי לבצע חיפוש מקבילי של מספר pattern'ים בתוך text.

```
=> (agent_example)
Demonstrating agents in action on the following pattern:
 hiha
=====================================
 Pattern        Matched

hi:            true
ih:            true
hj:            false
hiha:            true
```

שורות ההרצה: בתוך הקובץ testing.clj יש לרשום (agent_example)

✓ **Ahead-of-time compilation: קריאה לפונקציה Clojure מתוך Java.**

השימוש הוא לצורך Proof-of-concept בלבד, כדי להדגים את היכולת שלנו לקרוא לפונקציה שכתבנו ב Clojure מתוך Java (מתוך קובץ Clojure שקומפל מראש, Ahead of time). מבחינת test'ים של הפרויקט, יש כיסוי נרחב יותר בחלק של ה Clojure.לא העמסנו על הקובץ Java כדי "להעביר את המסר", באופן דומה ניתן לקרוא לכל אחת מהפונקציות שלנו.

הקבצים הדרושים:

1) `suffix-1.0.0-SNAPSHOT-standalone.jar` (זה ה jar של הפרויקט שלנו)

2) `AOT.java`

שורות ההרצה:

```
javac -cp ".;suffix-1.0.0-SNAPSHOT-standalone.jar" AOT.java
java -cp ".;suffix-1.0.0-SNAPSHOT-standalone.jar" AOT
```

הפלט המתקבל:

```
C:\Temp\j\java>javac -cp ".;suffix-1.0.0-SNAPSHOT-standalone.jar" AOT.java

C:\Temp\j\java>java -cp ".;suffix-1.0.0-SNAPSHOT-standalone.jar" AOT
Demonstrating AOT, calling Clojure functions from Java
Input text: This is my input text
Pattern This was found
Pattern is was found
Pattern ab was not found

C:\Temp\j\java>_
```

פירוט הבדיקות שנעשו לתוכנה

שימוש בתמיכה המובנית של Clojure ב Unit testing , זאת בעזרת המילה השמורה deftest .

שימוש ב deftest מאפשר להגדיר פונקציות testing אותן ניתן להריץ באופן מרוכז , ולקבל פלט לגבי כישלון או הצלחה שלהן (מבחינה מספרית). יש לציין כי יחידות ה testing משמשות גם כתיעוד חי של הקוד , מאחר והן מאפשרות לאדם שאינו מכיר את הקוד להתנסות באופן מידי ואינטואיטיבי במהות ושימוש הקוד.

הבדיקות לכל הפונקציות מופיעות בקובץ 'testing' במיקומים הבאים:

עבור מחרוזת בודדת:

Build_tree – line 19, is_sub -line 35, which_ind –line 48, how_many- line 64, longest_repeating_substring- line 84

עבור קבוצת מחרוזות (lazy):

Sub_lazy- line 118, idx_lazy- line 132, how_many_lazy- line 148, longest_repeating_substring_lazy- line 168

הרחבות עתידיות אפשריות לפרויקט

מתן מענה לשאילתות הבאות:

1. מציאת תת המחרוזת הארוכה ביותר שחוזרת על עצמה בטקסט מסוים: מחפשים צומת שיש לו לפחות 2 עלים- אפשר לעבור על העץ ולראות מהם כל הצמתים שיש להם שני עלים, וכמה עמוקה כל צומת (כמה היא רחוקה מהשורש באותיות).
2. בהינתן K טקסטים ומחרוזת P, מציאת כל הטקסטים שמכילים את המחרוזת P: נשרשר את הטקסטים ונפריד ביניהם בעזרת תווים מיוחדים (שונים) T1$1T2$2...TK$K. נבנה עץ סיפות לשרשור.