תקציר זה אינו נועד להחליף את החומר שנלמד בהרצאות ובתרגולים. הוא נועד לסכם בקצרה את המוטיבציה לאלגוריתם. הפרטים המלאים של האלגוריתם נמצאים בשקפי ההרצאות.

אלגוריתם Karp-Rabin

נתונות לנו מחרוזת text (אורכה n) ומחרוזת pattern (אורכה m) והמטרה היא למצוא את כל המופעים של pattern בתוך text. נסמן מופע ע"י אינדקס ההתחלה שלו בתוך text.

למשל:

Text = "abcbcbc", pattern= "bcb"

המופעים הם: [1,3] (כלומר "abcbcbc", "abcbcbc")

האלגוריתם הנאיבי יעבור על כל אינדקס התחלה אפשרי ב text ויבדוק האם תת המחרוזת שמתחילה ממנו ואורכה m (אם יש כזו) זהה ל pattern. השוואה של תת מחרוזת ל pattern (ע"י השוואה של תו לתו) עולה O(m) ויש O(n-m) אינדקסים שהם נק' התחלה פוטנציאליות למחרוזות באורך m ב text ולכן סה"כ נקבל סיבוכיות O(m*n) = O(m(n-m)).

המטרה שלנו היא למצוא אלגוריתם לינארי באורך הקלט – כלומר סיבוכיות הזמן שלו צריכה להיות O(m+n).

הערה: אם במקום השוואה של תו לתו נפעיל פונקציית hash ונבדוק האם ההפעלה של הפונקציה על תת המחרוזת החזירה אותו ערך כמו הפעלתה על pattern אז אנו עלולים לקבל false positives כלומר תת מחרוזות שאינן זהות ל pattern שפונקציית ה hash החזירה עבורן ערך זהה לזה שהחזירה עבור pattern. בנוסף גם להפעיל פונקציית hash על מחרוזת באורך m לוקחת O(m) זמן, כך שלא שיפרנו.

מאחר שברור שעלינו לבדוק את כל O(n-m) תתי המחרוזות הפוטנציאליות באורך m ב text הדבר היחיד שניתן לצמצם על מנת לקבל זמן לינארי הוא ההשוואה של שתי תת מחרוזות באורך m. אם נצליח לבצע אותה בזמן O(1) אז נקבל את הסיבוכיות המבוקשת.

כיצד נעשה זאת?

ראשית נסתכל על כל מחרוזת כעל מספר בבסיס 2^16. למה ניתן לעשות זאת? ניתן לייצג כל תו (או לפחות כמות מספיקה של תווים) בבינארית ע"י 16 ביטים. לכן מספר התווים השונים בסה"כ הוא לכל

היותר 2^16. לכל תו קיים מספר (עשרוני) בין 0 ל- 2^16 - 1 שמייצג אותו (את המספר הזה ניתן לכתוב בבינארית תוך שימוש בלכל היותר 16 ביטים). המספר הזה הוא ערך ה Unicode של התו.

כלומר בהינתן מחרוזת "abcd" , למשל, נמיר אותה למספר עשרוני שמייצג אותה (ורק אותה) באופן הבא:

aritmetization("abcd") =

ord("a")*(2^16)^3 + ord("b")*(2^16)^2 + ord("c")*(2^16)^1 + ord("d")*(2^16)^0

זה מאוד דומה להמרה מבסיס כלשהו b לבסיס עשרוני (במקרה הזה b=2^16).

ההמרה הזו לוקחת O(m) זמן עבור מחרוזת באורך m וגם יוצרת מספרים גדולים מאוד שלא ניתנים לייצוג ע"י מספר קבוע של "מילים" בזיכרון – לכן גם השוואה של שני מספרים גדולים כל כך לא תיקח זמן קבוע.

לכן נעשה מודולו עם מספר ראשוני אקראי כלשהו r. נגדיר את הפעולה הבאה:

fingerprint("abcd") = aritmetization("abcd") % r.

ברור שכל הערכים שנקבל כתוצאה מ fingerprint יהיו בין 0 ל r-1 ולכן יתפסו מספר קבוע של "מילים" בזיכרון (והשוואה ביניהם תיקח זמן קבוע). אבל – יש לשים לב שכעת ייתכנו false positives כי בעוד שaritmetization של שתי מחרוזות שונות תמיד החזירה ערכים שונים fingerprint עלולה להחזיר את אותו הערך.

נותר עוד לפתור את בעיית החישוב של ה fingerprint שלוקח O(m) עבור מחרוזת באורך m (כי aritmetization לוקחת O(m)). לשם כך נשים לב שאם חישבנו את fingerprint("abcd") אז ניתן לחשב בזמן קבוע את fingerprint("bcde") באופן הבא:

fingerprint("bcde") =

((fingerprint("abcd") – (ord("a") * (2^16)^3)) *(2^16) + ord("e")) % r

שימו לב שהמודולו r נעשה במהלך החישוב ולא רק על התוצאה הסופית כדי שלא נעבוד בשום שלב עם מספרים גדולים מ r (לפי חוקי האריתמטיקה המודולרית שפגשתם כאשר למדתם על פרוטוקול דיפי-הלמן זה חוקי).

כלומר לאחר שחישבנו את ה fingerprint של תת המחרוזת הראשונה באורך m בזמן O(m), לחשב את הfingerprint של הבאה אחריה יקח O(1). השוואה של שני fingerprints תיקח זמן קבוע כי שני הערכים בגודל לכל היותר r. בסה"כ האלגוריתם יקח זמן O(m+n), אך ייתכנו false positives.