**Smith Waterman algorithm**

**הקדמה:**

a - מחרוזת בעלת אורך n-1.

b - מחרוזת בעלת אורך m-1.

([s(a[i],b[j - פונקציית הזהות בין אותיות.

(W(i - עלות הכנסת מרווחים.

[H[i][j - הציון המקסימלי בהשוואת המחרוזת a מתחילתה עד האות ה-i ומחרוזת b מתחילתה עד האות ה-j.

**גרסה 0 (ללא מימוש):**

נממש את האלגוריתם ע"י שימוש ברקורסיה.

**הבעיה** - הפתרון הרקורסיבי הישיר אינו יעיל (זמן ריצה אקספוננציאלי) מאחר שהוא דורש את פתרון אותן תתי בעיות מספר רב של פעמים וכך אנחנו מבזבזים משאבים רבים (זמן ריצה וזיכרון).

**הפתרון** - נשתמש בתכנון דינאמי.

**גרסה 1:**

נשתמש בתכנון דינאמי על מנת לשמור תוצאות שכבר פתרנו עבור שימוש עתידי, את התוצאות נשמור במטריצה (מערך דו מימדי) ואנו נשתמש בהן בצורה הבאה, לכל תא במטריצה, ערכה נקבע כמקסימום בין התאים הבאים:

1. התא שבאלכסון משמאל.
2. המקסימום בין התאים שמשמאלו באותה השורה.
3. המקסימום בין התאים שמעליו באותה העמודה.

צריך לעבור על המטריצה ולכל תא שערכו נקבע לפי התאים שצוינו למעלה ולכן נקבל יעילויות כלהלן:

יעילות זמן ריצה (O(m\*n^2+m^2\*n.

יעילות זיכרון של (O(m\*n.

**הבעיה** - התכנון הדינאמי אולי משפר את היעילות מאקספוננציאלית לפולינומיאלית (n^3) אך עדיין נרצה להוריד את זמן הריצה הגבוהה.

**הפתרון** - נשתמש בזיכרון נוסף כדי לחסוך בזמן ריצה.

**גרסה 2:**

במקום שעבור כל תא נעבור על כל התאים שמעליו באותה העמודה ועל כל התאים שמשמאלו באותה שורה ונמצא מיהו המקסימום, נשתמש בשני מטריצות באופן הבא:

1. מטריצה ראשונה - תשמור בכל תא את הערך הגבוהה ביותר מבין כל התאים שמשמאלו באותה השורה.
2. מטריצה שנייה - תשמור בכל תא את הערך הגבוהה ביותר מבין כל התאים שמעליו באותה העמודה.

עכשיו לכל תא במטריצה המרכזית נקבע את ערכו לפי התאים הבאים:

1. התאים שבאלכסון משמאל.
2. התא המקביל אליו במטריצה הראשונה (מכיל את המקסימום בין התאים שמשמאלו באותה השורה).
3. התא המקביל אליו במטריצה השנייה (מכיל את המקסימום בין התאים שמעליו באותה העמודה).

צריך לעבור על המטריצה ולכן תא שערכו נקבע לפי התאים שצויינו למעלה ולכן נקבל יעילויות כלהלן:

יעילות זמן ריצה (O(m\*n.

יעילות זיכרון של (O(3\*m\*n.

**הבעיה** - הגרסה הנוכחית יעילה בזמן ריצה אך בזבזנית מאד בזיכרון, במידה ונרצה להריץ את הקוד במקביל עם זיכרון מוגבל יכול להיות שצריכת הזיכרון תמנע מאיתנו לנצל לחלוטין את מספר המעבדים בהם נשתמש בריצה המקבילית.

**הפתרון** - לא נשמור את כל טבלאות התכנון הדינאמי אלא רק את התאים שדרושים לי כדי לחשב את ערכה של השורה הבאה.

**גרסה 3:**

מכיוון שהערך בכל תא בשלושת המטריצות נקבע לפי השורה שמעליו, משמאלו ובאלכסון ממנו, במקום לשמור את כל הטבלאות של הנתונים אנו צריכים רק שורה אחת אחורה כדי לחשב את השורה החדשה.

נשמור שלושה מערכים של שתי שורות בלבד (עם אורך n) במקום כל אחד מהמערכים הגדולים שהיו מקודם, ננהל את הזיכרון כך שנמקסם את יעילות השימוש בזיכרון.

יעילות זמן הריצה לא תשתנה אך הזיכרון יקטן בהרבה, היעילויות שנקבל:

יעילות זמן ריצה (O(m\*n.

יעילות זיכרון של (O(6\*n.

**הבעיה** - הגרסה הנוכחית יעילה בזמן ריצה ובזיכרון, ננסה להקטין עוד יותר את צריכת הזיכרון בשביל שנוכל להגיע לניצול מרבי של מספר המעבדים בהם נשתמש בריצה המקבילית, שהרי הזיכרון מוגבל וככל שכל מעבד יצרוך פחות זיכרון כך יש לנו יותר פוטנציאל להפעלה של יותר מעבדים.

**הפתרון** -נצמצם עוד יותר את השימוש בזיכרון שכן ערכו של כל תא נקבע לפי התא שמעליו או באלכסון ולכן לא צריך את כל התאים שמשמאל לאלכסון.

**גרסה 4:**

מכיוון שהערך בכל תא במטריצה המרכזית וכן במטריצה ששומרת מקסימום בעמודה, נקבע לפי הערך בתא שמעליו והערך בתא שבאלכסון השמאלי שלו, במקום לשמור את שתי השורות של הנתונים אנו צריכים לשמור בנפרד רק את התא שמעליו ואת האלכסון.

מכיוון שהערך בכל תא במטריצה ששומרת מקסימום בשורה נקבעת רק לפי הערך בתא שמשמאל בשורה, נצטרך רק את התאים שבשורה הנוכחית.

נשמור שלושה מערכים באורך n במקום שלושה מערכים עם שני שורות שהיו בגרסה הקודמת, בשל מורכבות הקוד נתכנן בקפידה את ניהול הזיכרון.

יעילות זמן הריצה לא תשתנה אך הזיכרון יקטן בחצי, היעילויות שנקבל:

יעילות זמן ריצה (O(m\*n.

יעילות זיכרון של (O(3\*n.

**גרסה 5 (תיאורטי בשלב זה):**

מכיוון שהערך בכל תא במטריצה ששומרת מקסימום בשורה נקבעת רק לפי הערך בתא שמשמאל בשורה, נצטרך רק את התא שנמצא משמאל.

במידה ואכן נצליח לממש את הקוד החדש, היעילויות שנקבל:

יעילות זמן ריצה (O(m\*n.

יעילות זיכרון של (O(2\*n.

**בדיקות:**

**בדיקות נכונות:**

את הבדיקות עשיתי באמצעות מספר דוגמאות הרצה מהאינטרנט, את הדוגמאות ניתן למצוא בקישורים הבאים:

1. <http://en.wikipedia.org/wiki/Smith%E2%80%93Waterman_algorithm>
2. <http://amrita.vlab.co.in/?sub=3&brch=274&sim=1433&cnt=1>
3. <http://www.slideshare.net/avrilcoghlan/the-smith-waterman-algorithm>

עבור שלושת הדוגמאות הנ"ל הרצתי את אלגוריתם Smith-Waterman, הדפסתי את הערך המוחזר וכן את טבלאת התכנון הדינאמי כדי לבדוק שלא רק התוצאה הסופית נכונה אלא גם החישובים שנעשו בדרך.

הבדיקות נמצאות בקובץ CorrectnessTest.c כאשר התוצאות להן אנו מצפים נמצאות בקובץ  ExpectedOutput. עבור בדיקת הנכונות השתמשתי בגרסאות של האלגוריתם בעלות יכולת ההדפסה (הגרסאות שנמצאות בתקייה PrintableSW).

תוצאות: בזכות בדיקת הנכונות מצאתי שאכן היתה טעות בחישובי הביניים של גרסה 2, מה שגרם לטעות בכל גרסה שהתבססה עליה. הטעות היתה באלגוריתם המוצא שהתבסס על תאים לא נכונים בטבלאות העזר, התאים הנכונים אותרו והטעות תוקנה.

לאחר שהטעות תוקנה, בדיקות הנכונות עבור כל גרסה הניבו תוצאות סופיות נכונות וכן כל חישובי הביניים (טבלאות התכנון הדינמי) היו נכונים עבור כל אחת מהגרסאות.

**בדיקות יעילות זמן ריצה:**

כדי לבדוק את ההבדל ביעילות בין גרסאות הקוד השונות כתבתי תוכנית שתדמה את הפעולה של ריצת הקוד הנתון, בריצת הקוד הנ"ל יש מספר רב של מחרוזות מעל הא"ב {A,C,G,T} באורך 35 תווים וכן מחרוזת נוספת מעל הא"ב אך עם מספר רב של תווים. לכל אחת מהמחרוזות באורך 35 תווים מופעל אלגוריתם SW על המחרוזת הנתונה וכן על המחרוזת הארוכה מנקודת התחלה ועד נקודת סיום.

כדי לבדוק יעילות כתבתי תוכנית שתדמה את הפעולה של הקוד הנתון, התוכנית שכתבתי בונה באופן רנדומלי מחרוזות באורך 35 תווים מעל הא"ב {A,C,G,T}) ומפעילה עליהן את אלגוריתם ה-SW.

החלטתי שאת בניית המחרוזות אעשה תוך שימוש בזיכרון סטטי מוגדר מראש ואדאג להפרדה מוחלטת בין בניית המחרוזות לבין ריצת אלגוריתם ה-SW על המחרוזות השונות מכיוון שלא ארצה שבניית המחרוזות תיכלל במדידת זמן הריצה של הפעלת אלגוריתם ה-SW (למרות שאם כל גרסה תכליל זאת כחלק מזמן הריצה שלה הדבר לא אמור להשפיע על השאלה איזה גרסה יעילה יותר אלא רק על השאלה מהו זמן הריצה המדויק של הגרסה).

תוצאות צפויות: מכיוון שאנו בודקים יעילות בלבד ע"י הרצת אלגוריתם ה-SW מספר קבוע של פעמים היעילות של גרסאות 2,3 ו-4 אמורות להיות זהות (עלות של n^2 מכיוון ש m=n) לעומת גרסה 1 שאמורה להיות גדולה במידה ניכרת (עלות של n^3).

**בדיקת יעילות גרסה 1:**

יצרתי 340,000 מחרוזות באורך 35 תווים (מגבלת הזיכרון במחשב הפרטי שלי) ועבור כל זוג מחרוזות עוקבות מופעל אלגוריתם SW, כלומר הרצת האלגוריתם 340,000 פעמים.

**הבעיה** - בגרסה המדוברת יש חסם תחתון נמוך על מספר ההרצות שניתן לבצע, החסם הנמוך מונע ביצוע של הרצות רבות שידמו את פעולת הקוד האמיתי.

**הפתרון** - יצרנו את גרסה מספר 2 שבה יש שימוש במספר מחרוזות קטן יותר אך עבור כל זוג מחרוזות מריצים את אלגוריתם SW.

**בדיקת יעילות גרסה 2:**

יצרתי n מחרוזות באורך 35 תווים ועבור כל שתי מחרוזות שנוצרו מופעל אלגוריתם SW, כלומר הרצת האלגוריתם n^2 פעמים.

אם מספר המחרוזות הוא 1,000 (n=1000) אזי אלגוריתם SW ירוץ 1,000,000 פעמים עבור זוגות שונים של מחרוזות.

בגרסה זו נוכל להשתמש כדי לבדוק את זמני הריצה השונים שנקבל עבור מספר מחרוזות שונה וכן כדי לבדוק את זמני הריצה השונים שנקבל עבור אורכי המחרוזות.

**בדיקות יעילות זיכרון:**

את הזיכרון נבדוק על ידי ניתוח תיאורטי בלבד, בנוסף נספק הסבר על המניע שלנו להגיע ליעילות זיכרון גבוהה.

יש לשים לב ששתי המחרוזות הן בעלות אותו האורך (n=m), בהמשך הניתוח נתייחס לעובדה הזו.

בגרסה מספר 1 של אלגוריתם SW יש בזבוז של זיכרון עם יעילות נמוכה של n^2.

בגרסה מספר 2 של אלגוריתם  SW ייעלנו את זמן הריצה אך התשלום על כך היה שיצרנו את הגרסה עם הבזבוז הרב ביותר של זיכרון, פי שלוש יותר מאשר בגרסה 1.

בגרסה מספר 3 של אלגוריתם SW ייעלנו את הזיכרון ל- 6\*n שזה נמוך משמעותית מהגרסאות הקודמות ויתן לנו להשתמש ביותר מעבדים במידה והזיכרון יגביל את מספר המעבדים שניתן להשתמש בהם.

בגרסה מספר 4 של אלגוריתם SW ייעלנו את הזיכרון אפילו יותר והגענו ל- 3\*n. הייעול הזה לא נראה משמעותי כי זה כך הכל חצי מהגרסה הקודמת אך מכיוון שאם הזיכרון יגביל לנו את כמות המעבדים שיוכלו לרוץ, הקטנה של הזיכרון שכל מעבד דורש בחצי בלבד יתן לנו לעבוד עם פי 2 יותר מעבדים.