תרגיל בית רטוב 2 – חלק יבש

תיאור מבנה הנתונים

למימוש מבנה הנתונים השתמשנו במבני הנתונים הנלמדו בהרצאה: עץ AVL, Hash Table, Union Find.

עץ AVL - העץ מומש באמצעות template עם מחלקה אחת כפרמטר – KEY, אשר מייצגת את מפתחות העצים אשר קובעים את מיון המידע בעץ, שהוא תמיד מצביע לאובייקט קבוצה.

Hash Table- מומש באמצעות שיטת Chain Hashing שנלמדה בהרצאה, זהו מערך דינמי בגודל התחלתי 63=m, אשר משתנה דינמית ומכפיל את גודלו כאשר כמות האיברים מגיעה לכמות המלאה של מספר מקומות המערך (אך הוא לא חייב להיות מלא, יכולות להיות התנגשויות בהתאם לשיטה).

Union Find – מומש עץ הפוך של שחקנים שאיננו ממוין, שהוא עץ דרגות בעל 2 דרגות: partialSpiritDiff, gamesPlayedDiff.

בשונה מהמימוש הסטנדרטי הנלמד בשיעורים, במקום מערך רגיל אשר מצביע על צמתי העץ ההפוך, הHash Table שמימשנו, מצביע על צמתי העץ.

מימשנו את המחלקות הבאות:

- **Team** – מחלקה המתארת קבוצה במשחק, היא מחזיקה מצביע למפתח כללי (כפי שנפרט בהמשך), אובייקט פרמוטציה, ניקוד של הקבוצה, מצביע לשורש עץ השחקנים ההפוך של הקבוצה, משתנה בוליאני שקובע אם הקבוצה חוקית (אחד השחקנים בה הוא שוער), כמות השחקנים בקבוצה.
- **Player** - מחלקה המתארת שחקן, מכילה את ת"ז ואת מספר הכרטיסים שקיבל.
- **PlayerTable** - Hash Table, מומש באמצעות שיטת Chain Hashing, מצביע על מיקומי השחקנים בעצים ההפוכים.
- **TeamId** – מחלקה אשר שומרת את ת"ז של קבוצה ומשמשת למעשה כמפתח בעץ הקבוצות הממוין לפי ת"ז.
- **TeamStats** – מחלקה היורשת מTeamId אשר שומרת בנוסף את היכולת של הקבוצה, המשמש כמפתח בעץ הקבוצות הממוין לפי יכולת הקבוצה כפי שהוגדר בפונקציה get_ith_pointless_ability.
- **Chain**- מחלקת רשימה, אשר מצביעה על מיקומי השחקנים בעצים ההפוכים, משתמשים בה בPlayerTable .
- **PlayerNode** – צומת בעץ הפוך, אשר מצביע על שחקן ושומר 2 דרגות. חישוב הערך מהדרגות כולל מעבר על מסלול החיפוש עד לשורש העץ ו"סכימת" הדרגות המתאימות (סכימה עבור כמות המשחקים ששוחקו והרכבה עבור רוח חלקית), ולבסוף כיווץ מסלולים, כל זה מתבצע בסיבוכיות משוערכת עם שער פעולות ה-UnionFInd של O(log* n). בעת איחוד העצים, אנו יכולים לבצע עדכון מתאים בדרגות השורשים על מנת לשמר את ערכים אלה או להתאימם לסדר הכרונולוגי במקרה של הרוח החלקית, כפי שראינו **בשאלת הארגזים בתרגול** (מכיוון שפרמוטציות הן חבורה עם פעולת הרכבה). צירפנו הסבר מפורט לעדכון הדרגות בסוף המסמך.
- **TeamNode**- צומת בעץ הקבוצות, אשר מצביע על קבוצה ושומר בן שמאלי, בן ימני ודרגה שהיא כמות הצמתים בתת העץ.
- **TeamTree** – מחלקה אשר מנהלת עץ AVL של קבוצות.

מבנה הנתונים שמימשנו מורכב מ:

- **שני עצי AVL של קבוצות**, אחד ממויין לפי מספר מזהה והשני בדגש על יכולת הקבוצה.
- **Hash Table** ששומר את מיקומי השחקנים בעצים שלהם.
- **קבוצה מזוייפת** (בעלת מספר מזהה 0 שאינו חוקי) אשר משמשת כקבוצת כל השחקנים שקבוצתם הודחה מהטורניר.

סיבוכיות זמן

world_cup_t()

מאתחלת את מבנה הנתונים הנ"ל: אתחול 2 עצי AVL ריקים בסיבוכיות זמן של O(1), אתחול Hash Table בגודל התחלתי קבוע בסיבוכיות זמן של O(1), ובניית הקבוצה המזוייפת, כאשר אתחול אובייקט קבוצה הוא בסיבוכיות זמן של O(1) כפי שתואר קודם לכן. סה"כ בסיבוכיות זמן של O(1), כנדרש.

virtual ~world_cup_t()

מוחק את כל הקבוצות מעצי הקבוצות בסיבוכיות זמן של O(k), לאחר מכן, גם את 2 עצי הקבוצות גם בסיבוכיות זמן של O(k). לבסוף מוחק את הHash Table , על ידי מעבר לינארי על איברי המערך, כאשר כל אחד עלול להיות רשימה של צמתי השחקנים (במקרה הרע), ומוחק את אותם הצמתים עם השחקנים מהמערך ובאופן כללי מהעצים, מתבצע בסיבוכיות זמן של O(n). מאחר שגודל המערך לא יכול לעלות על 2n מאופן הגדלת המערך באופן דינמי. לסיכום, סיבוכיות זמן של O(k+n), כנדרש.

StatusType add_team(int teamId)

תחילה, הפונקציה בודקת את תקינות הקלט, שבמקרה הגרוע תגיע לבדיקה האם קיימת כבר קבוצה בעלת המזהה הנתון, כאשר בדיקה זו מחפשת האם קיימת כבר קבוצה זו בעץ הקבוצות. מאחר שעץ הקבוצות AVL סיבוכיות הזמן היא O(log(k)). לאחר בדיקת הקלט, בהנחה שהקבוצה לא קיימת במערכת, מוסיפה את הקבוצה לשני העצים בסיבוכיות זמן של O(log(k)). סה"כ סיבוכיות זמן של O(log(k)), כנדרש.

StatusType remove_team(int teamId)

שוב תחילה הפונקציה בודקת את תקינות הקלט, ובכל מקרה מוצאת את הקבוצה במערכת בעלת המזהה הנתון, לפי חיפוש בעץ AVL אשר מתבצע בסיבוכיות זמן של O(log(k)). כעת, הפונקציה לוקחת את שורש עץ השחקנים של הקבוצה (עץ הפוך) ומאחדת אותו עם עץ השחקנים של הקבוצה המזוייפת שלנו, לפי UnionFind, כאשר האיחוד שנלמד בהרצאה מתבצע בסיבוכיות של גובה העץ ההפוך מכיוון שעל מנת לאחד את העצים עלינו להגיע לשורשם, אך במקרה זה אנו כבר שומרים את השורש ועל כן לא חייבים לבצע את מסלול החיפוש ויכולים לאחד את שני העצים בסיבוכיות של O(1) ולשמור אותם בקבוצה המזוייפת (ולעדכנה בהתאם, גם בסיבוכיות זמן של O(1)). לבסוף מוחקת את הקבוצה הישנה, ובדומה ליצירת הקבוצה מחיקת הקבוצה (ללא עץ השחקנים שלה) מתבצע בסיבוכיות זמן של O(1). לסיכום, סיבוכיות זמן הריצה של הפונקציה היא O(log(k)), כנדרש.

StatusType add_player(int playerId, int teamId, const permutation_t &spirit, int gamesPlayed, int ability, int cards, bool goalKeeper)

תחילה, יש לבדוק אם השחקן לא נמצא כבר במערכת. לשם כך, נבדוק האם הוא נמצא ב Hash Table , מתבצע בסיבוכיות זמן משוערכת של O(1) בממוצע על הקלט, כפי שנלמד בהרצאה. בהנחה שהשחקן לא נמצא במערכת, הפונקציה מוצאת את הקבוצה אליה הוא אמור להתווסף השחקן בסיבוכיות זמן של O(log(k)), ובמקרה הגרוע כאשר הקבוצה קיימת ונמצאה, קודם כל מסירה אותה מעץ הקבוצות שממויין לפי יכולת, שכן היכולת הכללית של הקבוצה עלולה להשתנות בעת הוספת שחקן, כאשר ההסרה מתבצעת גם בסיבוכיות זמן של O(log(k)). לאחר מכן הפונקציה מוסיפה את השחקן אל הקבוצה בסיבוכיות זמן של O(1), בשימוש בUnionFind על מנת לאחד בין עץ השחקנים הנוכחי של הקבוצה לעץ שחקנים זמני הכולל רק את השחקן הנוכחי, שוב בסיבוכיות זמן של O(1) כפי שהסברנו קודם. כעת יש לנו את מיקומו של השחקן בעץ השחקנים ונותר להכניסו ל Hash Table, כאשר הכנסה זו מתבצעת בסיבוכיות זמן משוערכת של O(1) בממוצע על הקלט, כפי שנלמד בהרצאה. לבסוף,

נחזיר את הקבוצה לעץ השחקנים הממויין על פי יכולת הקבוצה, ומכיוון שהעץ הוא עץ AVL ההכנסה מתבצעת בסיבוכיות זמן של O(log(k)). לסיכום, מתבצעת בסיבוכיות זמן משוערכת של O(log(k)) בממוצע על הקלט, כנדרש.

output_t<int> play_match(int teamId1, int teamId2)

תחילה, במקרה הכי גרוע מוצאת את 2 הקבוצות בעץ הקבוצות הממוין לפי ת"ז בסיבוכיות זמן של O(log(k)). את $\sum_{\text{players}}$ player ability הקבוצה שומרת במשתנה, ומעדכנת אותו בעת הכנסת כל שחקן לקבוצה. לפיכך את החישוב

$$\text{points} + \sum_{\text{players}} \text{player ability}$$

לצורכי ההשוואה בין 2 הקבוצות, מקבלת את היכולת הכוללת של הקבוצה ואת כוחה הרוחני בסיבוכיות זמן של O(1), ולבסוף מעדכנת את ניקוד הקבוצות ומוסיפה משחק אחד לדרגת השורש של שני עצי השחקנים של עצי הקבוצות (מה שבעיקרון מוסיף משחק אחד לכל השחקנים בשני העצים לפי הגדרת חישוב המשחקים לפי הדרגות), כל זה בסיבוכיות של O(1). לסיכום, הפונקציה מתבצעת בסיבוכיות זמן של O(log(k)), כנדרש.

output_t<int> get_player_cards(int playerId)

במקרה הגרוע מחפשים את השחקן בHash Table, בסיבוכיות זמן של O(α) בממוצע על הקלט, כפי שנלמד בהרצאה על שיטת Chain Hashing, כאשר על ידי שינוי גודל המערך באופן דינמי אנו מבטיחים כי בכל רגע מתקיים $m = \theta(n)$, ועל כן $\alpha = \frac{n}{m} = \frac{n}{\Omega(n)} \leq \frac{n}{cn} = \frac{1}{c} = O(1)$, ועל כן מציאת השחקן מתבצעת בסיבוכיות זמן של O(1) בממוצע על הקלט. כעת נחזיר את כמות הכרטיסים שהשחקן שומר אצלו, בסיבוכיות זמן של O(1), ועל כן סיבוכיות זמן הריצה של כל הפונקציה היא O(1) בממוצע על הקלט, כנדרש.

output_t<int> get_team_points(int teamId)

במקרה הגרוע, מוצאים את הקבוצה בעץ הקבוצות הממוין לפי ת"ז, בסיבוכיות זמן של O(log(k)). אובייקט הקבוצה מכיל משתנה של ניקוד הקבוצה ולכן נחזיר מידע זה בסיבוכיות זמן של O(1). סה"כ בסיבוכיות זמן של O(log(k)), כנדרש.

output_t<int> get_ith_pointless_ability(int i)

מחפשים את הקבוצה בעץ הקבוצות הממוין לפי ability כפי שתואר בדרישות הפונקציה, על ידי חיפוש לפי דרגות כפי שנלמד בהרצאה על עצי דרגות לפי **אינדקס** (על מנת למצוא את הקבוצה שיש משמאלה לפי סדר InOrder בדיוק i קבוצות), בסיבוכיות זמן של O(log(k)). לבסוף נחזיר את ת"ז שלה, בסיבוכיות זמן של O(1), ועל כן סיבוכיות הזמן הכוללת של הפונקציה היא O(log(k)), כנדרש.

הסבר סיבוכיות משוערכת על 4 הפונקציות בעלות סיבוכיות משוערכת שאינן add_player

מכיוון שאנו משתמשים בפונקציות הבאות בUnionFind, כפי שנלמד בהרצאה הסיבוכיות המשוערכת של פעולות Find ו-Union היא O(log* n) כאשר אנו מבצעים בכל פעולת Find כיווץ מסלולים. על כן, מכיוון שהפונקציות הבאות משוערכות זו עם זו בכל פעם שנבצע Union או Find נתייחס לסיבוכיות המשוערכת של פעולות אלו כ O(log* n).

output_t<int> num_played_games_for_player(int playerId)

תחילה, נחפש את השחקן בHash Table בסיבוכיות זמן בממוצע על הקלט של O(1). בהנחה שמצאנו את מיקום השחקן, נגיע אל השחקן בעץ ההפוך. נעלה מהשחקן עד לשורש העץ, ונסכום את מספר המשחקים שלו, כפי שתיארנו במימוש המחלקה PlayerNode. אחרי שהגענו לשורש, נחזור שוב לשחקן בעץ ובעת מעבר על מסלול החיפוש (בסדר הפוך) נבצע כיווץ מסלולים, כפי שנלמד בהרצאה. סיבוכיות זמן הריצה המשוערכת של פעולה זו (Find) היא O(log* n), כפי שציינו קודם לכן. לסיכום הסיבוכיות המשוערכת של הפונקציה היא O(log* n) בממוצע על הקלט, כנדרש.

StatusType add_player_cards(int playerId, int cards)

תחילה, במקרה הגרוע ובו השחקן קיים, כאשר מציאתו מתבצעת ב(O(1 בממוצע על הקלט כפי שהסברנו בפונקציות הקודמות, עלינו לבדוק האם הוא משחק בקבוצה שלא הודחה. משמעות הדבר במימוש שלנו הוא לוודא ששורש העץ ההפוך בו הוא נמצא מצביע על קבוצה לא מזוייפת, ועל כן עלינו לבצע את מסלול החיפוש עד לשורש העץ על מנת לקבל את הקבוצה בה הוא משחק, ובדרך חזרה במסלול הקריאה נבצע כיווץ מסלולים. פעולה זו (Find) מתבצעת בסיבוכיות זמן משוערכת של O(log* n). לבסוף, אם הקבוצה אינה מזוייפת (בדיקה שמתבצעת ב-O(1) נגדיל את כמות הכרטיסים של אותו השחקן בסיבוכיות של O(1). לסיכום, הסיבוכיות המשוערכת של הפונקציה היא O(log* n) בממוצע על הקלט, כנדרש.

output_t<permutation_t> get_partial_spirit(int playerId)

תחילה, נחפש את השחקן בHash Table בסיבוכיות זמן בממוצע על הקלט של O(1). בהנחה שמצאנו את מיקום השחקן, נגיע אל השחקן בעץ ההפוך. קודם כל עלינו לוודא כי השחקן משחק בקבוצה פעילה (שאינה מזוייפת על פי ההגדרה שלנו), ועל כן עלינו לעלות במסלול החיפוש לשורש העץ ולבדוק את קבוצתו. במקרה הגרוע, הקבוצה משתתפת במשחק ועל כן עלינו להמשיך בחיפוש, אך קודם לכן בעת החזרה ממציאת הקבוצה נבצע כיווץ מסלולים, ועל כן סיבוכיות זמן הריצה המשוערכת של מציאת הקבוצה וכיווץ המסלולים (Find) היא O(log* n). לאחר מכן, נעלה מהשחקן לשורש העץ, ונרכיב את הרוח החלקית שלו משמאל לימין מלמעלה למטה, כפי שתיארנו במימוש המחלקה PlayerNode. נציין כי אנו שומרים על כרונולוגיות החישוב בעת מיזוג עצים כפי שלמדנו בשאלת הארגזים בתרגולים, עם תשומת לב לכיוון ההרכבה מכיוון שהרכבת פרמוטציות אינה פעולה קומוטטיבית בהכרח. כעת, במהלך מסלול החזרה לצומת של השחקן בעץ נבצע כיווץ מסלולים (שאמנם לא משנה שום דבר בפועל מכיוון שכבר כיווצנו מסלול זה, אך זה לא פוגע בסיבוכיות להתעלם מזה) מה שמתבצע בסיבוכיות זמן משוערכת של O(log* n). לסיכום, סיבוכיות הזמן המשוערכת של הפונקציה היא O(log* n) בממוצע על הקלט, כנדרש.

StatusType buy_team(int buyerId, int boughtId)

הפונקציה קודם כל מוצאת את שתי הקבוצות בעץ הקבוצות לפי ת"ז, בסיבוכיות זמן של O(log(k)). במקרה הגרוע, שתי הקבוצות קיימות ועל כן עלינו לאחד את שני עציהן ולשמור אותו בקבוצה הקונה. קודם לכן, עלינו להוציא את הקבוצה הנרכשת משני עצי הקבוצות שלנו, מה שמתבצע בסיבוכיות זמן של O(log(k)), ובנוסף להוציא את הקבוצה הרוכשת מעץ הקבוצות לפי יכולת קבוצתית בסיבוכיות זמן של O(log(k)), שכן ערך זה ישתנה לאחר האיחוד. כעת, נאחד את שני עצי השחקנים של שתי הקבוצות (שוב, לפי העיקרון של בעיית הארגזים שהוצגה בתרגול) כאשר שוב יש לנו כבר את שני שורשיהם, ועל כן ניתן לבצע את האיחוד בסיבוכיות זמן של O(1), כפי שציינו ב-add_player. לבסוף, נוסיף מחדש את הקבוצה המאוחדת לעץ הקבוצות הממויין על פי יכולת קבוצתית, בסיבוכיות זמן של O(log(k)). לסיכום, סיבוכיות זמן הריצה של הפונקציה היא O(log(k)), בשיפור מהנדרש.

סיבוכיות מקום

בכל רגע בזמן הריצה, צריכת המקום של מבנה הנתונים שלנו היא צריכת המקום של שני עצי הקבוצות, Hash Table עבור מיקומי השחקנים בעצים, עצי השחקנים עצמם, והקבוצה המזויפת.

צריכת המקום של שני עצי הקבוצות היא לינארית לכמות הצמתים בו, ששווה לכמות הקבוצות שבו, שהיא כמות כל הקבוצות במערכת שהיא k, על כן סיבוכיות המקום של שני העצים היא O(k).

צריכת המקום של Hash Table השחקנים היא צריכת המקום של מערך המצביעים לרשימות שמצביעות על מיקומי השחקנים בעץ, ובנוסף צריכת המקום של אותן הרשימות. צריכת המקום של המערך היא לינארית לאורך שלו, שהוא m, אשר לפי השינוי הדינמי שהגדרנו לו מקיים $m = O(n)$. צריכת המקום של הרשימות היא לינארית לכמות סך האיברים בהן, כאשר יש איבר אחד בדיוק לכל שחקן במערכת, ועל כן סיבוכיות המקום שלהן היא O(n) גם כן.

צריכת המקום של עצי השחקנים לינארית גם היא לסך כמויות הצמתים בהם, ששווה לכמות השחקנים n, ועל כן סיבוכיות המקום שלהם היא O(n).

צריכת המקום של הקבוצה המזוייפת אשר שומרת את עץ השחקנים של השחקנים שקבוצתם הודחה היא קבועה, ועל כן סיבוכיות המקום שלה היא O(1).

לסיכום, סיבוכיות המקום של כל מבנה הנתונים שלנו היא O(n + k), כנדרש.

<u>נקודה חשובה</u>: אף פונקציה שאנו מימשנו אינה מייצרת אובייקט אשר עובר על סיבוכיות מקום זו, מכיוון שאנו אך ורק מייצרים מפתח זמני לחיפוש בעץ הקבוצות (O(1)), קבוצה זמנית להוספת שחקן לעץ (O(1)), ובעת ביצוע rehash בHash Table אנו לא משנים את כמות האיברים ברשימות ומכפילים את גודל המערך פי שתיים, אך הדבר עדיין משאיר אותו בסיבוכיות מקום של O(n), ועל כן פעולה זו גם היא אינה משבשת את סיבוכיות המקום הנדרשת.

הסבר מפורט על שמירת שתי הדרגות בUnionFind

כפי שציינו קודם לכן, חישוב הערכים מן הדרגות מבצע סכימה שלהם מהשורש עד לצומת השחקן עבור כמות המשחקים, והרכבה עבור הרוח החלקית. עלינו לתקן את הדרגות במהלך שתי פעולות: כיווץ מסלולים שמתבצע לאחר Find, ובמהלך Union.

תיקון בעת כיווץ מסלולים – כיווץ המסלולים מתבצע באופן רקורסיבי, ועל כן לאחר שכיווצנו את המסלול של ההורה לשורש, לכל צומת שההורה שלו הוא אינו השורש, מתקיים שההורה של ההורה שלו הוא השורש, ועל כן הערכים הנוכחיים שנחשב מהדרגה שלו הם:

עבור כמות משחקים ששוחקו – $G(this) = g(root) + g(parent) + g_{old}(this)$

עבור רוח חלקית – $S(this) = s(root) \cdot s(parent) \cdot s_{old}(this)$

ולאחר כיווץ המסלולים אנחנו לא רוצים שערכים סופיים אלו ישתנו, אף בחישוב החדש נקבל:

$$G(this) = g(root) + g_{new}(this)$$

$$S(this) = s(root) \cdot s_{new}(this)$$

ועל כן, לאחר השוואת שני האגפים וסידורם נקבל:

$$g_{new}(this) = g(parent) + g_{old}(this)$$

$$s_{new}(this) = s(parent) \cdot s_{old}(this)$$

ועל כן, אם נבצע את העדכון הנ"ל נשמור על הערכים הנדרשים.

תיקון במהלך ביצוע Union – עבור קבוצה A בעלת שורש a מקורית שמאחדים אליה קבוצה B בעלת שורש b:

על מנת שכמות המשחקים של כל השחקנים בעץ השחקנים שמתאחד לקבוצה השנייה תישמר, מספיק לנו לשמר את הערך שמחושב עבור שורש העץ מכיוון שכל שאר הערכים מחושבים ביחס אליו. בשונה מכך, עלינו להרכיב משמאל על ערך הרוח החלקית המחושב עבור כל השחקנים בקבוצה המתווספת את ערך הרוח הקבוצתית של כל השחקנים שנכחו בקבוצה לפניהם (מכיוון שההרכבה היא כרונולוגית), שהוא הערך שנשמר כרגע בקבוצה עצמה, ומאותה הסיבה מספיק לנו להרכיב אותו משמאל על הערך המחושב עבור השורש. לפיכך נדרוש:

$$G_{new}(a) = G_{old}(a) \quad , \quad G_{new}(b) = G_{old}(b)$$

$$S_{new}(a) = S_{old}(a) \quad , \quad S_{new}(b) = S_{old}(A) \cdot S_{old}(b)$$

מכיוון שאנו מאחדים את העץ הקטן יותר אל תוך העץ הגדול יותר, יש לנו שני מקרים כתלות בגדלי העצים:

<u>מקרה 1</u> – מאחדים את *B* לתוך *A* – במקרה זה נגדיר את ההורה של *b* להיות *a*, ונקבל את הדרישות הבאות:

$$G_{new}(a) = g_{new}(a) = G_{old}(a) = g_{old}(a) \quad \rightarrow \quad g_{new}(a) = g_{old}(a)$$

$$G_{new}(b) = g_{new}(a) + g_{new}(b) = G_{old}(b) = g_{old}(b) \quad \rightarrow \quad g_{new}(b) = g_{old}(b) - g_{new}(a)$$

$$S_{new}(a) = s_{new}(a) = S_{old}(a) = s_{old}(a) \quad \rightarrow \quad s_{new}(a) = s_{old}(a)$$

$$S_{new}(b) = s_{new}(a) \cdot s_{new}(b) = S_{old}(A) \cdot S_{old}(b) = S_{old}(A) \cdot s_{old}(b) \quad \rightarrow$$

$$s_{new}(b) = \left(s_{new}(a)\right)^{-1} \cdot S_{old}(A) \cdot s_{old}(b)$$

ועל כן אם נבצע את השינויים הנ"ל בדרגות נקבל את הערכים הסופיים כנדרש.

<u>מקרה 2</u> – מאחדים את A לתוך B – במקרה זה נגדיר את ההורה של a להיות b, ונקבל את הדרישות הבאות:

$$G_{new}(a) = g_{new}(b) + g_{new}(a) = G_{old}(a) = g_{old}(a) \quad \rightarrow \quad g_{new}(a) = g_{old}(a) - g_{new}(b)$$

$$G_{new}(b) = g_{new}(b) = G_{old}(b) = g_{old}(b) \quad \rightarrow \quad g_{new}(b) = g_{old}(b)$$

$$S_{new}(a) = s_{new}(b) \cdot s_{new}(a) = S_{old}(a) = s_{old}(a) \quad \rightarrow \quad s_{new}(a) = \left(s_{new}(b)\right)^{-1} \cdot s_{old}(a)$$

$$S_{new}(b) = s_{new}(b) = S_{old}(A) \cdot S_{old}(b) = S_{old}(A) \cdot s_{old}(b) \quad \rightarrow \quad s_{new}(b) = S_{old}(A) \cdot s_{old}(b)$$

ועל כן אם נבצע את השינויים הנ"ל בדרגות נקבל את הערכים הסופיים כנדרש.