קבוצה 44
מגישים: ענבר קדם 325298438, עוז דניאל לוי 209999739, אילון חודניק 325130417.

שאלה 1

ב׳ 1. בתחילת הקוד אנחנו בודקים אם מערך הקלט יותר ארוך מאורך מתקפה בעלות של O(1).
לאחר מכן מתבצע Sort על רשימת הקלט לפי זמן קבלת הבקשות באמצעות פונקציית Sort המובנית בפייתון, שמתבצעת בסיבוכיות O(n*log(n)).
בשלב הבא באמצעות לולאת For הבקשות מתחלקות למילון לפי סוגי הבקשה בסיבוכיות של O(1) לכל קלט, כלומר O(n).
למרות שאנחנו משתמשים ב-if x in dict (פונקציית member כפי שהוגדרה בתרגול) בשביל לבדוק אם בקשה מסוימת כבר נמצאת במילון, חיפוש במילון נעשה ב-O(1).
בנוסף, למרות שבתוך לולאת ה-For יש לולאת While, כל בקשה יכולה להכנס או לצאת מהתור רק פעם אחת, לכן הסיבוכיות של כל לולאת ה-For כולל ה-While היא O(n).
כל פעולות ההכנסה וההוצאה מהתור, ופונקציות ההשוואה לאן מכן נעשות ב-O(1).
לכן הסיבוכיות הכוללת של הפונקציה היא O(n*log(n)

ג. האלגוריתם שממישנו לוקח את טבלת הקלט, הופך את עמודת הערכים שלה לערימת מינימום עם המחלקה שממישנו בסעיף א. לאחר מכן אנחנו מתאחלים שלושה משתני עזר שניים שישמרו את שני הערכים המינימלים בזמן הלולאה, ואחד שישמור את הסכום. אז מאתחלים לולאת while שתרוץ עד שגודל הערימה יהיה 1, שבתוך הלולאה מוציאים את שני הערכים המינימלים שאותם מחברים ומחזירים את הסכום שלהם לתוך הערימה. לאחר ההכנסה נבצע heapify. בסיום התהליך נחזיר את משתנה הסכום. האלגוריתם שלנו עובד מכיוון שאנחנו לוקחים תמיד סכום של שני המחוברים הכי קטנים על מנת "לחזור" בחיבור על הערכים הכי קטנים שאפשר למשל לפי הדוגמא עדיף "לחזור" על החיבור של A,B ואז C ואז D מאשר לעשות A+B וC+D.

ד. סיבוכיות האלגוריתם:

מהיבט של סיבוכיות של האלגוריתם אנחנו מתחילים מפעולת heapify על כל הערימה, וכפי שראינו בתרגול הסיבוכיות שלה היא O(N). לאחר מכן אנחנו עושים לולאת while שבה עושים פעמיים deletemin ופעם אחת insert שכפי שראינו בתרגול בעלות סיבוכיות של O(log(n)). בנוסף נשים לב כי הלולאת while תרוץ n-1 פעמים מכיוון שכל

פעם היא תאחד שני איברים עד שהיא תקבל איבר אחד סופי. לכן נקבל את המשוואה הכוללת לסיבוכיות:

$$O(N)+(n-1)*O(3\log(N))=O(N\log(N))$$

ה. מהו Huffman Coding ומהם השימושים שלו:

Huffman Coding היא שיטת קידוד סימנים או תווים ללא אובדן, אשר מבוססת על הקצאת קודים בינאריים משתני־אורך: סימנים בעלי שכיחות גבוהה יותר מקבלים קודים קצרים יותר, ובכך מנצלים בצורה מיטבית את מספר הביטים הדרוש לאחסון או העברה של הנתונים.

על מנת לבנות את הקודים, מתחילים עם רשימת הצמתים (עלים) של כל סימן וסטטיסטיקת התדירות שלו, בונים ערימת מינימום לפי תדירות ההופעה, ולאחר מכן בכל שלב שולפים מהערימה את שני הצמתים בעלי התדירות הנמוכה ביותר, מצליבים אותם לצומת־על חדש ששוקל את סכום התדירויות, ומחזירים אותו לערימה. כך ממשיכים עד שנשאר צומת־שורש יחיד. נתיב ההליכה מהשורש אל כל עלה בקצה העץ מייצג את הקוד הבינארי של אותו סימן, כאשר "0" מסמל הורדת שכבה לשמאל ו"1" הורדה לשכבה ימנית.

לדוגמה, נבחן את המילה "LOSSLESS" ונחשב את תדירות כל אות: האות "S" מופיעה ארבע פעמים, "L" פעמיים, "O" פעם אחת ו־"E" פעם אחת. לאחר הצבת הללו בעץ Huffman מתקבלים, למשל, הקודים "S → 0", "L → 10", "O → 110" ו־"E → 111", וזאת בשל העובדה שהסימנים הנפוצים יותר (כגון "S") מקבלים קוד בן ביט אחד בלבד, בעוד סימנים בעלי שכיחות נמוכה (כמו "O" או "E") יקבלו קודים ארוכים יותר.

שיטה זו נפוצה כחלק מתקן דחיסת תמונות JPEG, בתקן דחיסת קול MP3, וכן בפורמט PNG (חלק מ-DEFLATE), בגיבוי ZIP ו-GZIP, ובכלל במערכות תקשורת ופרוטוקולים רבים המשתמשים בדחיסה ללא אובדן באמצעות טכניקות כמו LZ77 ו-LZ78 בשילוב עם Huffman Coding להקטנת כמות הנתונים הנדרשת לאחסון או שידור.

שאלה 3

ה.

	חלוקה - שרשור	חלוקה - בדיקה ריבועית	חלוקה - גיבוב כפול	כפל - שרשור	כפל - בדיקה ריבועית	כפל - גיבוב כפול
Sheet 1	1.067	2.042	2.058	1.075	1.983	2.084
Sheet 2	1.0	1.0	1.0	1.0	1.243	1.789
Sheet 3	9.47	14.546	3.537	1.016	1.697	1.756

בסט הנתונים הראשון אנחנו רואים שקיים יחס כמעט 1:1 בין הפעולות לשורות כאשר משתמשים בשיטת שרשור לניהול התנגשויות. שאר השיטות מראות יחס פחות יעיל של בערך פי 2 פעולות משורות. מצב זה יכול להעיד על מצב בו אין הרבה התנגשויות לאחר פונקציית הגיבוב, אבל כאשר משתמשים במיעון פתוח נוצרות התנגשויות נוספות שמפחיתות את ה״יעילות״.

בסט הנתונים השני אנחנו רואים שלפי פונקציית גיבוב של חלוקה הנתונים כבר מגובבים באופן מושלם, מסיבה זאת כנראה אין הבדל בין שיטת ניהול ההתנגשויות.

עם זאת, כאשר משתמשים בפונקציית כפל ובמיעון פתוח לניהול התנגשויות נדרשות יותר פעולות ביחס לשורות הנתונים.

בסט הנתונים השלישי אנחנו רואים שפונקציית גיבוב של חלוקה דורשת הרבה יותר פעולות משורות וגורמת למדד יעילות גבוה לפחות פי 2 משל פונקציית כפל.

פונקציית חלוקה עם בדיקה ריבועית נותנת את היחס הכי קיצוני (ובכך הכי פחות ״יעיל״) בין פעולות לשורות. לעומת זאת, פונקציית כפל עם שרשור לניהול התנגשויות נותנת יחס של כמעט 1:1 בין הפעולות לשורות.

בשלושת הסטים אנחנו רואים שמיעון פתוח נוטה להגדיל את האפקט של התנגשויות על ״מדד היעילות״, כלומר הוא יכול לדרוש הרבה ניסיונות עד פתרון ההתנגשות.

ו.

עבור סט הנתונים הראשון והשלישי ההצעה שלנו היא להחליף את A ביחס הזהב ההופכי שהוא מספר אי רציונלי שקשה לקרב אותו בעזרת הרציונלים לכן הכפולות שלו בדרך כלל לא יתאמו לכפולות של מספרים רציונליים. תכונה זאת תגרום לחלוקת אינדקסים יותר אחידה ובכך לפחות התנגשויות. דבר זה יקרב את היחס שחשבנו בסעיף ה ל-1 עבור שני הסטים.

סט הנתונים השני מסודר בהתאם לפונקציית גיבוב חלוקה מראש, ולכן אין המון מקום לשיפור, אבל אנחנו מניחים ששינוי A לקבוע יחס הזהב ההופכי ישפר את הביצועים של המיעון הפתוח בפונקציית הכפל.

שימוש ביחס הזהב ההופכי מוכח מתמטית כיעיל ״באינסוף״, כלומר עבור כמות נתונים גדולה מאוד.
סטי הנתונים הם קטנים יחסית לכן לא מובטח שיפור במקרה הנוכחי.

שינוי נוסף שיכול לשפר את הביצועים הוא הגדלת m, לכאורה כמה שיותר.
הגדלת m תאפשר להשתמש בטבלה גדולה יותר וכמות יותר גדולה של אינדקסים פנויים.
לכן הנתונים יוכלו להיות פחות ״צפופים״ ויווצרו פחות התנגשויות.
שינוי זה יכול לשפר את ״מדד היעילות״ עבור כל פונקציות הגיבוב וכל סטי הנתונים.
עם זאת, טבלת גיבוב גדולה יותר דורשת יותר זיכרון ולכן נהוג להשתמש ב-m שהוא בין פי 1.5-2 מכמות הנתונים.
בנוסף, עבור גיבוב חלוקה כדאי להשתמש ב-m ראשוני בשביל גיבוב פחות תבניתי.