יעילות

מסמך זה מבוסס במידה רבה על פרק 6
"יעילותם של אלגוריתמים"
מספרו של פרופ' דוד הראל
"אלגוריתמיקה - יסודות מדעי המחשב"
שבהוצאת האוניברסיטה הפתוחה.
קטעים מסוימים אף נלקחו ככתבם וכלשונם
מהספר הזה, והם כתובים כאן בגופן "אריאל".

כאשר מתבקשים לבנות גשר, קל לבנות אותו "לא נכון". הגשר עלול להיות צר מלהכיל את מספר הנתיבים הרצוי, חלש מכדי לשאת את התנועה בשעות העומס, או שהוא עלול לא להגיע כלל אל העבר האחר! אולם, אפילו אם הגשר "נכון", במובן זה שהוא עונה לחלוטין על כל דרישות הביצוע, לא כל הצעת תכנון עבורו תוכל להתקבל. ייתכן שביצוע התכנון המוצע דורש כוח אדם רב מדי, או אולי חומרים או מרכיבים רבים מדי. במלים אחרות, למרות שהתוצאה עשויה להיות גשר טוב, התכנון עלול להיות יותר מדי יקר.

כך גם בכתיבת אלגוריתמים שמיושמים בסופו של דבר כתכניות מחשב. אנו לא נדון בכוח אדם ובעלויות אחרות הקשורות בו, אלא בקריטריונים אחרים שימדדו עד כמה האלגוריתמים או התכניות שלנו יעילים. בדרך כלל הקריטריונים המובילים בהערכת טיב תכנית נכונה הם זמן ומקום. במדעי המחשב קריטריונים אלו מכונים **מדדי סיבוכיות** (complexity measures). הזמן הוא **זמן** החישוב, כלומר משך הזמן שבו מתבצעת התכנית, והמקום הוא גודל **זיכרון** המחשב שיש להקצות לצורך ביצוע התכנית. כאן נתרכז בסיבוכיות הזמן, אך נציין שחקר סיבוכיות המקום עוסק בנושאים דומים לאלה שיועלו כאן.

כפי שראינו ביחידה 5, בתיאור של מבנה המחשב, המחשבים הנפוצים היום מסוגלים לבצע מיליוני פעולות בשנייה. כמו כן, כוח החישוב של המחשבים הולך ומשתפר במהירות גדולה כל-כך במשך השנים, עד כי נדמה שכל הדיון ביעילות של תכניות ואלגוריתמים הוא מיותר לחלוטין. כי הרי, מה זה משנה אם תכנית תבצע מיליון פעולות או שני מיליון פעולות, ממילא הדבר ייקח רק כמה שברירי שנייה. ובכן, הנחה זו מוטעית מיסודה.

זמן הוא גורם בעל חשיבות עליונה כמעט בכל שימוש של מחשבים. גם ביישומים יומיומיים, כגון תכניות לחיזוי מזג האוויר, מערכות ניהול מלאי ותכניות חיפוש בספרייה, יש מקום נרחב לשיפורים. זמן הוא כסף, וזמן מחשב אינו יוצא מכלל זה. יתרה מזאת, בכל הנוגע למחשבים, יכול למעשה הזמן להיות הגורם המכריע בהא הידיעה. מערכות ממוחשבות מסוגים מסוימים,

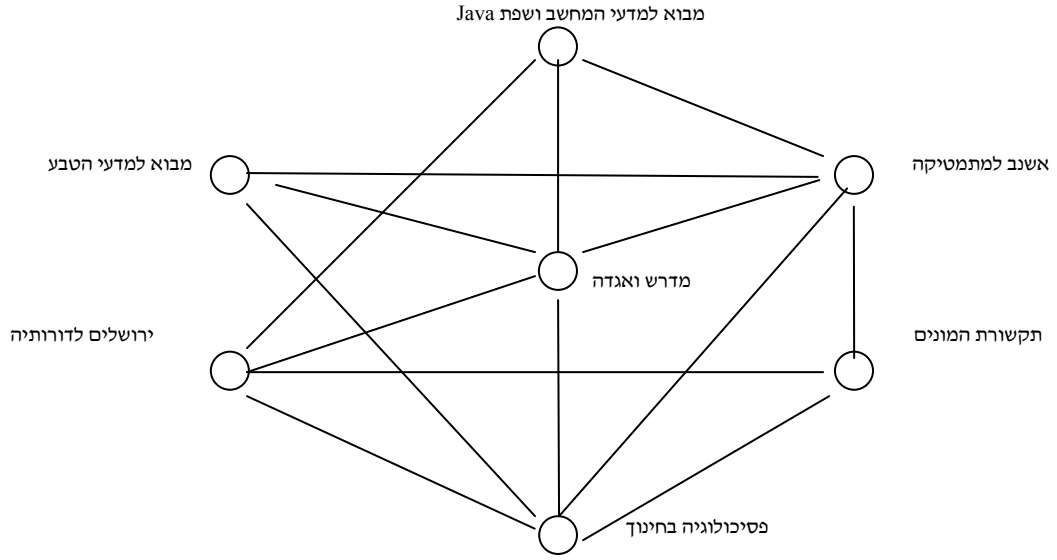
כגון בקרת טיסה, הנחיית טילים ותכניות ניווט, נקראות **מערכות זמן אמת** (real time systems). אלה חייבות להגיב לגירויים חיצוניים - כגון לחיצה על כפתורים - בזמן "אמיתי", כלומר כמעט מיד. אי יכולת לעמוד בכך עלולה להוביל לתוצאות קטלניות.

בעיות שזמן פתרונן אינו סביר

כדי להדגים עד כמה חשוב לעסוק בתחום היעילות של אלגוריתמים, וכמה הדיון בתחום אינו מיותר, נראה בעיה שפתרונה יארך זמן בלתי סביר, אפילו במחשבים המהירים ביותר הקיימים כיום, וגם באלו שייבנו בעתיד הנראה לעין.

האוניברסיטה הפתוחה רוצה לקבוע לוח בחינות לסיום הסמסטר. מתכנני הלוח מעוניינים לקבוע את מועדי הבחינות כך שלסטודנט הלומד שני קורסים (ואף יותר) בסמסטר, לא יקרה ששתי הבחינות בשני הקורסים יתקיימו באותו מועד, שכן אז יוכל לעשות רק בחינה אחת מהשתיים. כמובן שרצוי שמספר מועדי הבחינה יהיה קטן ככל האפשר, בגלל העלות הגבוהה של כל מועד בחינה (שכירת מרכזי הבחינה בכל הארץ, שכירת משגיחים וצוות לתפעול מרכזי הבחינה וכדו'). המצב הנתון כיום הוא שיש חמישה מועדי בחינה (למועד א). הבעיה היא לסווג את הקורסים השונים לחמשת מועדי הבחינה, כך שיקיימו את התנאי שלכל שני קורסים ששיכים לאותו מועד אין סטודנט משותף. למען האמת נדגיש כי באוניברסיטה המצב לא בדיוק כך, לכל סטודנט ישנם שני מועדי א ביניהם הוא יכול לבחור, אך אנחנו ניצמד לתיאור הבעיה שהצגנו.

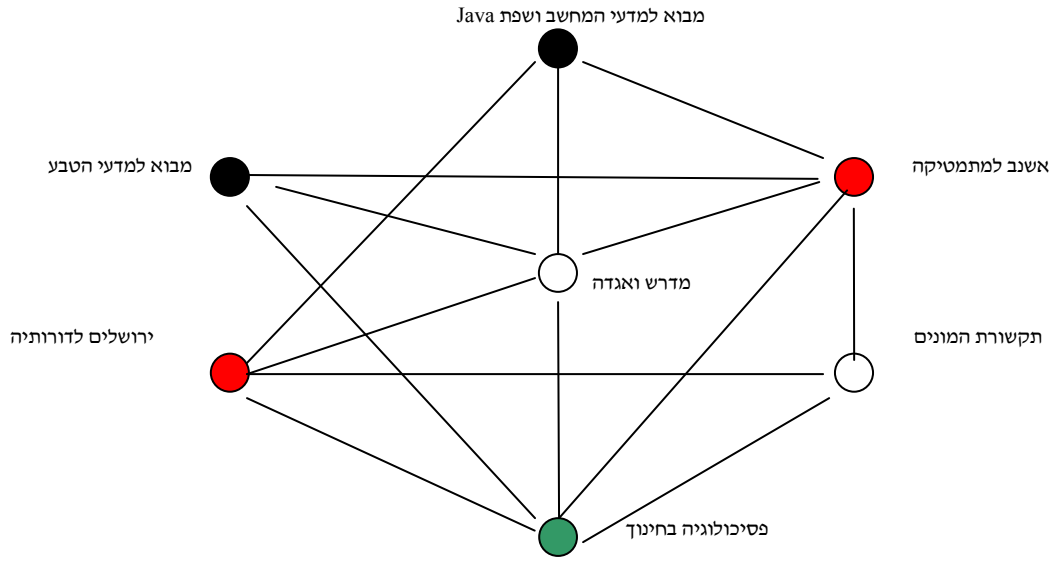
איך בכלל ניגשים לפתור את הבעיה? נתאר את הבעיה בעזרת המודל הבא: ניצור גרף בו כל קורס הוא צומת (עיגול באיור להלן), ואם יש סטודנט משותף לשני קורסים, אזי בין שני הצמתים של הקורסים מחברת קשת (קו באיור). כך לדוגמא, באיור שלהלן מצויר גרף המתאר את הבעיה עם שבעה קורסים: מבוא למדעי המחשב ושפת Java, אשנב למתמטיקה, מבוא למדעי הטבע, ירושלים לדורותיה, מדרש ואגדה, תקשורת המונים ופסיכולוגיה בחינוך. כל קו המחבר בין שני קורסים מייצג סטודנט הלומד את שני הקורסים האלה במקביל.



כמובן שככל שיש יותר קורסים, כך הגרף הולך וגדל.

אנו מעוניינים להצמיד לכל צומת מספר המציין באיזה מועד בחינה מתוך חמשת המועדים, תתקיים בחינה בקורס המיוצג על-ידי הצומת. אם במקום חמישה מספרים נבחר בחמישה צבעים שונים, הבעיה תהיה לצבוע את צומתי הגרף כך שבכל קשת (קו), שני קצותיה לא ייצבעו בשני צבעים שונים. לכן בעיה זו נקראת "בעיית צביעת גרף במינימום צבעים".

הנה פתרון אפשרי לגרף שהבאנו לעיל. כל צבע מייצג מועד בחינה אחר. (שימו לב שהסתפקנו בארבעה צבעים כדי לצבוע את הגרף. נסו לראות, האם אפשר לצבוע את הגרף בשלושה צבעים בלבד? בשניים?)



נחזור לבעיה המקורית עם חמישה צבעים (מועדים). השאלה היא, בהינתן גרף, האם אפשר לצבוע אותו בחמישה צבעים, כאשר אין קשת המחברת שני צמתים בעלי צבע זהה? הדרך הנאיבית ביותר לפתרון הבעיה היא למצוא את כל הדרכים האפשריות לצבוע את הגרף בחמישה צבעים, ולכל צביעה כזו לבדוק אם היא חוקית, כלומר לבדוק אם היא מקיימת את התנאי. הבעיה נראית לכאורה פשוטה, שהרי יש רק מספר סופי של דרכים לצבוע את הגרף. ובדיקת חוקיות הצביעה, גם היא נראית לא מסובכת; צריך לעבור על כל הקשתות בגרף ולבדוק אם כל אחת מהן צבועה בשני צבעים שונים.

לא ניכנס כאן לפרטי בדיקת החוקיות, אלא נספור כמה צביעות אפשריות יש. ובכן, זה נכון שמספרם סופי, אך מספר זה יכול להיות מאד גדול. את כל אחד מהצמתים אפשר לצבוע בחמישה צבעים שונים, ללא תלות בצבעים שצבענו את שאר הצמתים (כזכור, אנו לא בודקים כרגע את חוקיות הצביעה, אלא רק צובעים). אם בגרף יש n צמתים הרי שמספר הצביעות הוא 5^n. (למה?) האם זה מספר גדול?

ובכן, נניח שיש לנו מחשב המסוגל לבדוק חוקיות של מליון צביעות בשנייה. כדי לעבור על כל הצביעות האפשריות בגרף שלעיל עם שבעה קורסים ולבדוק את חוקיותם, המחשב יזדקק לכ- 8 מאיות השניה. כדי לבדוק גרף עם עשרה קורסים, המחשב יזדקק לכ- 9 שניות, זמן סביר לכל הדעות. אם נגדיל את מספר הקורסים במעט, ל- 15, המחשב יזדקק לכ- 8.5 שעות. כדי לפתור את

הבעיה ל- 20 קורסים, המחשב יזדקק ל- 3 **שנים**. עבור 25 קורסים - למעלה מ- 9,000 **שנה**, ועבור 30 קורסים בלבד, המחשב יזדקק לכמעט **30 מיליון שנים** כדי למצוא את הפתרון. נציין רק שבאוניברסיטה הפתוחה מוצעים היום למעלה מ- 400 קורסים! כדי לפתור בעיה זו המחשב יזדקק למספר בן 266 ספרות של שנים!

הטבלה שלהלן מציגה את מספר הצביעות האפשריות עבור מספרים שונים של קורסים ואת הזמנים הדרושים לחישובם, בהנחה של בדיקת מיליון צביעות לשנייה:

מספר הקורסים	מספר הצביעות האפשריות	זמן החישוב
7	78,125	78 מילישניות
10	9,765,625	כ- 9 שניות
15	30,517,578,125	כ- 8.5 שעות
20	95,367,431,640,630	כ- 3 שנים
25	$298,023,233,877e^{17}$	כ- 9450 שנים
30	9,313,225,746,155e20	29,532,045 שנים

מובן מאליו שהאלגוריתם הנאיבי בו בחרנו לפתור את הבעיה אינו יעיל בכלל, ועלינו לחפש אלגוריתם אחר יעיל יותר. האם יש כזה? ובכן, בעיה זו (צביעת גרף) נמצאת במוקד המחקר במדעי המחשב, וטרם נמצא אלגוריתם הפותר אותה בזמן סביר.

צריך לשים לב שכאשר אנו מדברים על סבירות (או על אי-סבירות), אנו מתייחסים ל**גידול** הסביר (או הלא-סביר) בזמן כאשר אורך הקלט גדל. גם אם זמן החישוב יהיה 30 מיליון שנה רק עבור 200 קורסים, עדיין זה יכול להיחשב כאלגוריתם לא סביר.

מתברר כי בעיית צביעת הגרף אינה היחידה, ויש אוסף גדול מאוד של בעיות שדינן דומה. לרבות מהבעיות יש שימושים בכלכלה ובתעשייה, והאלגוריתמים הטובים ביותר עבורן שנמצאו עד כה, אפילו אם ירוצו על מחשבים מהירים וחזקים בהרבה מאלו שיש בידנו, יארכו זמן רב מאוד אפילו על קלטים לא גדולים במיוחד.

נזכיר בקצרה שתי בעיות מפורסמות מאוד, שהאלגוריתמים הטובים ביותר עד כה לפתרונם אינם סבירים: בעיית הסוכן הנוסע ובעיית הפירוק לגורמים. בעיית הסוכן הנוסע ,מקבלת גרף ממושקל, כלומר, גרף שבו לכל קשת המחברת בין שני צמתים בו יש משקל, ומוצאת מהו המסלול הקצר ביותר העובר דרך **כל** הצמתים בגרף, ובכל צומת **פעם אחת** בדיוק. הבעיה נקראת על שמו של סוכן נוסע אלמוני, שניסה למצוא את המסלול הקצר ביותר העובר דרך כל לקוחותיו. בעיית הפירוק לגורמים, מקבלת מספר ומנסה לפרק אותו לגורמים (כלומר, למצוא את המספרים הראשוניים המחלקים אותו). בעיית הסוכן הנוסע חשובה מאוד למשל בשימוש ברשתות תקשורת, כדי לחסוך בעלויות. נכון לעכשיו, לא קיים אלגוריתם המסוגל למצוא מסלול כזה בפחות ממיליוני שנים של זמן חישוב! נשים לב שהאלגוריתם אותו אנו מחפשים צריך למצוא מסלול קצר ביותר עבור **כל**

קלט חוקי. כלומר, ייתכן שיש אלגוריתמים שמוצאים מסלולים כאלה על גרפים מסוימים, או כאשר הקלט קטן, אבל אנו מחפשים משהו כללי שימצא תמיד את המסלול הקצר ביותר בזמן סביר.

בעיית הפירוק לגורמים משמשת את העוסקים בהצפנת מידע במחשב, ויש לה שימושים רבים בתחום. אבל, לא קיים אלגוריתם המסוגל לפרק לגורמים מספרים שלמים גדולים, בעלי 300 ספרות למשל, בפחות ממיליוני שנים. מעמדן של בעיות אלה עשוי להשתנות, אולם עד כה לא ידועה דרך לפתור אותן בזמן סביר.

מכל האמור לעיל אפשר להבין כי אין לזלזל בחשיבות היעילות של האלגוריתמים השונים לפתרון בעיות. כאשר אנו כותבים אלגוריתם לפתרון בעיה כלשהי, מן הראוי שנקדיש חלק חשוב מזמננו כדי לבדוק את יעילותו, ואם הוא לא יעיל מספיק, לנסות למצוא אלגוריתם אחר, יעיל יותר. לא תמיד הדבר קל (כמו בבעיות שתיארנו לעיל), ולפעמים הוא אינו אפשרי, אך במקרים רבים הדבר בהחלט אפשרי.

כיצד נמדוד יעילות?

כאשר אנו רוצים להשוות בין שני אלגוריתמים הפותרים אותה הבעיה, איך נמדוד מי מביניהם יותר יעיל? דרך אחת היא לממש את שני האלגוריתמים כתכניות מחשב, ואז למדוד את זמני הריצה של שתי התכניות. אולם האם יספיק לנו לדעת שזמן הריצה של תכנית אחת יותר קצר מזמן הריצה של התכנית השנייה? לא בדיוק. קודם כל עלינו להבטיח כי שתי התכניות הורצו על אותו מחשב בדיוק, כך נוכל לדעת כי אין יתרון מהירות למחשב אחד על משנהו. חוץ מזה, אנו צריכים להריץ את שתי התכניות על אותו קלט בדיוק, שהרי ברור שתכנית לחיפוש שם ברשימה תהיה מהירה יותר כאשר הרשימה בת אלף מילים מאשר אם הרשימה בת מיליון מילים. "כמות העבודה" שעל האלגוריתם לבצע תלויה, אם כן, ב**אורך הקלט** שעליו פועל האלגוריתם. במילים אחרות נאמר כי נביע את זמן ריצת האלגוריתם כ**פונקציה** של אורך הקלט.

אורך הקלט של בעיה הוא מספר הסיביות הנדרשות כדי לייצג את הקלט. כך, לכל בעיה נייחס את אורך הקלט המתאים לה. למשל, אם אנו רוצים למיין רשימה של שמות, הרי שאורך הרשימה יכול להיות אורך הקלט. אם אנו מחפשים מילה בטקסט, אפשר למדוד את אורך הקלט לפי מספר האותיות או מספר המילים בטקסט. בבעיית צביעת הגרף שהבאנו לעיל, אורך הקלט יכול להיות מספר הצמתים בגרף (כלומר מספר הקורסים). בבעיה המנסה לפרק מספר לגורמיו הראשוניים, מספר הספרות של המספר מהווה את אורך הקלט.

אין זה מעשי להתייחס לזמן הריצה של המחשב המסוים שעליו הורצה התכנית. לכן אנו מחפשים מדד שלא יהיה תלוי במחשב מסוים וגם לא באורך הקלט לבעיה.

כדי למדוד את יעילותו של הפתרון המוצע לבעיה מסוימת, לא נתייחס למחשב עליו המימוש שלו רץ, אלא נעריך את האלגוריתם עצמו. נעשה זאת על-ידי ספירת הפעולות שהאלגוריתם מבצע. נבדוק זאת לגבי קלטים שונים באורכים שונים. הפעולות אותן אנו סופרים צריכות להיות פעולות בסיסיות, שמשך ביצוען אינו תלוי באורך הקלט. כך למשל, הפעולה "חפש את השם 'כהן' ברשימה" לא יכולה להיות פעולה בסיסית כיון שזמן ביצועה תלוי באורך הרשימה. לעומת זאת,

באלגוריתם המחפש שם ברשימה נוכל לספור פעולה כגון "בדוק האם אתה נמצא בסוף הרשימה", או "עבור לשם הבא ברשימה" וכדו'.

? שאלה 1

אילו מהפעולות הבאות יכולות לשמש כפעולות בסיסיות בניתוח אלגוריתם?

א. באלגוריתם המחפש מחרוזת בטקסט, הפעולה היא מציאת אורך הטקסט.

ב. באלגוריתם לחישוב מספרי, הפעולה היא פעולת חילוק בין שני מספרים.

ג. באלגוריתם המקבל את n כקלט, הפעולה היא חישוב $\sum_{i=1}^{n} i$

ד. באלגוריתם המבצע חיפוש מספר במערך, הפעולה היא מעבר לתא הבא במערך.

שיפורים לאחר-מעשה

קיימות דרכים סטנדרטיות רבות לשיפור זמן הריצה של אלגוריתם נתון. אחדות מהן משולבות בפעולותיהם של מהדרים, והופכות את אלה למהדרים המבצעים **אופטימיזציות**, והם מתקנים למעשה סוגים מסוימים של החלטות לקויות שקיבל המתכנת.

חלק מהאופטימיזציות המתבצעות על-ידי מהדר, ניתן לראותן כ**טרנספורמציות על תכניות**. אחד מסוגי הטרנספורמציות השכיחים ביותר הוא שינוי תכנית או אלגוריתם על-ידי העברת הוראות מתוך לולאות אל מחוצה להן. לפעמים זו העברה ישירה, כמו בדוגמה הבאה. נניח כי מורה, מתוך כוונה לשפר את ציוני הבחינה של תלמידי כיתה מסוימת, רוצה לנרמל את רשימת הציונים על-ידי מתן הציון 100 לתלמיד שציונו הוא הגבוה ביותר בבחינה ושינוי כל הציונים האחרים בהתאם. תיאור ברמה גבוהה של האלגוריתם יכול להיראות כך:

(1) חשב את הציון הגבוה ביותר ואחסן אותו ב- MAX;

(2) כפול כל ציון ב- 100 וחלק אותו ב- MAX.

(עלינו להניח שקיים ציון אחד חיובי לפחות כדי שפעולת החילוק תהיה מוגדרת היטב.)

אם רשימת הציונים נתונה בווקטור $L(1)$, ..., $L(N)$, ניתן לבצע את שני חלקי האלגוריתם באמצעות לולאות פשוטות העוברות על הווקטור. הראשונה מחפשת את הציון המקסימלי באחת הדרכים המקובלות, ואת השנייה ניתן לכתוב כך:

(2) עבור I מ- 1 עד N בצע את הפעולות הבאות:

$$L(I) \leftarrow L(I) \times 100/MAX \quad (2.1)$$

שים לב לכך שהאלגוריתם מבצע בתוך הלולאה כפל אחד וחילוק אחד עבור כל $L(I)$. אולם, הן 100 והן ערכו של MAX אינם משתנים למעשה בתוך הלולאה. לכן ניתן לחסוך כמחצית(!) מזמן הביצוע של הלולאה השנייה על-ידי חישוב היחס $100/MAX$ לפני הכניסה ללולאה. כל שיש לעשות הוא להוסיף פשוט את ההוראה

$$FACTOR \leftarrow 100/MAX$$

בין שלב (1) לשלב (2), ולכפול בתוך הלולאה את $L(I)$ ב- $FACTOR$. האלגוריתם המתקבל הוא:

(1) חשב את הציון הגבוה ביותר ואחסן אותו ב- MAX;

(2) $FACTOR \leftarrow 100/MAX$;

(3) עבור I מ- 1 עד N, בצע את הפעולות הבאות:

(3.1) $L(I) \leftarrow L(I) \times FACTOR$.

גוף הלולאה השנייה, שהכיל במקור שתי פעולות חשבון, מכיל עתה פעולת חשבון אחת בלבד, ומכאן השיפור בכמעט 50%. מובן שלא בכל המימושים של אלגוריתם כזה יישלט זמן הריצה על-ידי פעולות החשבון; ייתכן שעדכון ערכו של $L(I)$ גוזל זמן רב יותר מאשר חילוק מספרים. אולם אפילו במקרה זה ישנו שיפור משמעותי בזמן הריצה, וקל לראות שככל שהרשימה ארוכה יותר כך חוסכים זמן רב יותר כתוצאה מהשינוי.

כפי שנאמר קודם, שינויים כאלה הינם ישירים למדי, ומהדרים רבים יכולים לבצעם באופן אוטומטי. אולם, גם סילוק פשוט של הוראה דורש לפעמים שימוש בתחבולה ערמומית אחת או שתיים. הבה נתבונן אפוא בדוגמה נוספת.

חיפוש לינארי

נניח שאנו מחפשים איבר X ברשימה לא ממוינת (למשל, מספר טלפון בספר טלפונים "מבולגן"). האלגוריתם המקובל מכיל לולאה פשוטה שבתוכה מתבצעות שתי בדיקות:

(1) "האם מצאנו את X?" ו-(2) "האם הגענו לסוף הרשימה?". תשובה חיובית לאחת משתי השאלות הללו גורמת לאלגוריתם לעצור - בהצלחה במקרה הראשון וללא הצלחה בשני. שוב, אנו יכולים להניח שבדיקות אלה הן המרכיב הדומיננטי בביצועי הזמן של אלגוריתם החיפוש.

את הבדיקה השנייה ניתן להוציא אל מחוץ ללולאה בעזרת התחבולה הבאה: לפני תחילת החיפוש מוסיפים באופן מלאכותי את האיבר המבוקש X *לסופה* של הרשימה. אז מבצעים את לולאת החיפוש **ללא** בדיקה לגבי סוף הרשימה; בתוך הלולאה אנו בודקים רק אם X נמצא. בכך קיצרנו שוב את זמן הריצה של האלגוריתם כולו בערך ב- 50%. עתה, היות שהוספנו את X לסוף הרשימה, X יימצא *תמיד*, גם אם אינו מופיע ברשימה המקורית. אולם במקרה זה,

בפעם הראשונה שנפגוש את X נמצא את עצמנו בסופה של הרשימה החדשה, ואילו אם X אכן מופיע ברשימה המקורית, נימָצא בעת הפגישה עימו במקום כלשהו בתוכה. ואמנם, עלינו לבדוק *פעם אחת* בלבד האם הגענו לסוף הרשימה - כאשר X אכן נמצא, והאלגוריתם ידווח על הצלחה או על כישלון על פי תוצאת בדיקה זו. (דרך אגב, קל לשגות כאן ולשכוח לסלק מן הרשימה את ה- X הנוסף לפני העצירה.) הפעם היה עלינו להיות מעט יותר יצירתיים כדי לחסוך את 50 האחוזים מזמן הריצה.

שיפורים בסדרי גודל

קיצור זמן הריצה של אלגוריתם ב- 50% הוא מרשים למדי. אולם במקרים רבים אנו יכולים להשיג שיפורים טובים הרבה יותר. באמרנו "טובים יותר", כוונתנו לא לקיצור זמן הריצה בשיעור קבוע, של 50%, 60% או אפילו 90%, אלא לקיצורים ש**שיעורם** גדל והולך ככל שהקלט גדול יותר.

אחת מן הדוגמאות הידועות ביותר עוסקת בחיפוש איבר ברשימה מסודרת, או ממוינת (למשל, חיפוש מספר טלפון בספר טלפונים רגיל). ליתר דיוק, נניח שהקלט כולל שם Y ורשימה L של שמות ומספרי טלפון. אנו מניחים שהרשימה, שאורכה N, ממוינת בסדר אלפביתי על פי השמות.

אלגוריתם נאיבי לחיפוש מספר הטלפון של Y הוא אותו אלגוריתם שתואר עבור רשימה לא ממוינת: עוברים על הרשימה L, שם אחר שם, משווים בכל צעד את Y לשם הנוכחי ובודקים תוך כדי כך האם הגענו לסוף הרשימה, או שמשתמשים לצורך החיפוש באותה תחבולה שתוארה בסעיף הקודם ומבצעים את הבדיקה פעם אחת בלבד, כאשר Y אכן נמצא. גם אם נקצר באופן זה את זמן הריצה ב- 50%, בכל זאת ייתכנו מקרים שבהם תידרשנה N השוואות; זה עלול לקרות כאשר Y אינו מופיע כלל ברשימה, או כאשר הוא מופיע במקום האחרון. אנו אומרים שזמן הריצה של האלגוריתם, שנקרא לו A, הוא **לינארי** ב- N **במקרה הגרוע ביותר** (worst-case). או, אם להשתמש במונח מקביל, זמן הריצה של A הוא **מסדר גודל** של N. אפשר לומר בקיצור שבמקרה הגרוע ביותר רץ A בזמן $O(N)$ (קרי: O-גדול של N, ובקצרה O של N), כאשר O גדול משמעותו "מסדר גודל של" (on the order of).

הסימון $O(N)$ הוא מתוחכם למדי. שים לב שבאמרנו "A רץ בזמן $O(N)$", לא קבענו שאנו מתייחסים אך ורק למספר ההשוואות שמבצע A, על אף שהשוואות של Y מול שמות ב- L הן ההוראות היחידות שספרנו. את הסיבה לכך נסביר אחר כך. לעת עתה די לומר שכאשר אנו משתמשים ב**סימון O-גדול**, כפי שהוא נקרא לפעמים, לא משנה לנו אם האלגוריתם דורש N זמן (כלומר, מבצע N הוראות יסוד), $3N$ זמן (כלומר, מבצע פי שלושה הוראות יסוד), $10N$ או אפילו $100N$. יתר על כן, אפילו אם זמן ריצתו של האלגוריתם הוא רק חלק קבוע מ- N, $N/6$

למשל, בכל זאת נאמר שזמן הריצה שלו הוא $O(N)$. הדבר היחיד שמשנה הוא שזמן הריצה גדל **באופן לינארי** עם N. פירושו של דבר שקיים מספר קבוע כלשהו K, כך שזמן ריצתו של האלגוריתם במקרה הגרוע ביותר אינו גדול מ- $K\times N$. ואמנם, זמן ריצתה של הגרסה הבודקת בכל פעם האם הגענו לסוף הרשימה הוא $2N$ בקירוב, ואילו הגרסה המשופרת מקצרת זמן זה ל- N בקירוב. זמן הריצה בשני המקרים הוא אפוא ביחס ישר ל- N. לכן שני האלגוריתמים האלה הם לינאריים בזמן; זמן ריצתם במקרה הגרוע ביותר הוא $O(N)$.

המונח "במקרה הגרוע ביותר" פירושו שהאלגוריתם עשוי לרוץ הרבה פחות זמן על קלטים מסוימים, אולי אפילו על רוב הקלטים. כל שאנו טוענים הוא שזמן ריצתו של האלגוריתם אינו עולה לעולם על $K\times N$, ושקביעה זו נכונה לכל N ולכל קלט מאורך N, אפילו במקרים הגרועים ביותר. כמובן, אם ננסה לשפר את אלגוריתם החיפוש הזה, שזמן ריצתו לינארי, על-ידי כך שנתחיל את ההשוואות *מסוף* הרשימה, גם אז יהיו מקרים גרועים באותה מידה - Y אינו מופיע כלל או Y מופיע כשם ראשון ברשימה. למעשה, אם האלגוריתם מחייב חיפוש ממצה בכל הרשימה, לסדר השוואת השמות ל- Y אין כל חשיבות; זמן ריצתו של אלגוריתם כזה במקרה הגרוע ביותר יהיה גם אז $O(N)$.

מדוע אנו מנתחים את זמן הריצה של האלגוריתם דווקא על-פי המקרה הגרוע ביותר? מדוע לא להתייחס ל**מקרה הממוצע** (average-case)? הרי ברוב המקרים הקלט יהיה ממוצע ולא גרוע ביותר? ואולי בכלל כדאי להתייחס ל**מקרה הטוב ביותר** (best-case)? ובכן, יש לכך כמה סיבות:

ראשית - בניתוח המתמטי של זמן הריצה במקרה הגרוע ביותר, ברור כי זהו הזמן הארוך ביותר לריצת האלגוריתם על כל קלט חוקי, ואף קלט לא יגרום לאלגוריתם לרוץ בזמן ארוך יותר מזמן זה. בעוד שאם יהיה בידנו זמן הריצה הממוצע, ובהינתן לנו קלט מסוים, לא נוכל לדעת אם זמן הריצה של האלגוריתם על הקלט המסוים יהיה ארוך יותר, קצר יותר או זהה לזמן הריצה הממוצע. הדבר דומה לחישוב שכר של העובדים בחברה מסוימת. אם נחשב את השכר הממוצע בחברה, האם נוכל לומר משהו על שכרו של יעקב העובד בחברה? האם שכרו גבוה, נמוך או זהה לשכר הממוצע? אולם אם נחשב את שכרו של הפועל הזוטר ביותר, נוכל לומר כי שכרו של יעקב לא יותר נמוך מהשכר במקרה הגרוע ביותר.

שנית - מציאת זמן הריצה במקרה הממוצע בדרך-כלל קשה לאין ערוך ממציאת זמן הריצה במקרה הגרוע ביותר. צריך להביא בחשבון את כל הקלטים האפשריים באורך n, ואת ההסתברות שכל קלט כזה יופיע, ואחר-כך לחשב את זמני הריצה של כל הקלטים האלו, ולחשב את הממוצע המשוקלל. בכל זאת ישנם מקרים בהם ננתח גם את זמן הריצה הממוצע.

ומה דעתך? מדוע לא נשתמש בניתוח זמן הריצה של המקרה הטוב ביותר?

למרות כל האמור, אנו מסוגלים בכל זאת לקצר את החיפוש ברשימה ממוינת, לא רק במונחי הגורם הקבוע "החבוי בתוך" ה- O הגדול, אלא במונחי האומדן $O(N)$ עצמו. שיפור כזה נקרא

שיפור בסדר-גודל (order-of-magnitude improvement), ואנו נראה עתה כיצד אפשר להשיגו.

חיפוש בינרי

לשם המחשה, הבה נניח שספר הטלפונים מכיל מיליון שמות, כלומר N שווה 1000000, ונסמן את השמות X_1, X_2, ..., $X_{1000000}$.

ההשוואה הראשונה שנעשית על-ידי האלגוריתם החדש אינה בין Y לבין השם הראשון או האחרון ב- L, אלא בין Y לבין השם האמצעי (או, אם אורכה של הרשימה זוגי, השם הראשון במחצית השנייה של הרשימה), כלומר X_{500001}. בהנחה שהשמות שהושוו אינם זהים, כלומר שעדיין לא מצאנו את Y ברשימה, קיימות שתי אפשרויות: (1) Y קודם ל- X_{500001} בסדר אלפביתי, ו-(2) X_{500001} קודם ל- Y. היות שהרשימה ממוינת בסדר אלפביתי, הרי אנו יודעים שבמקרה (1), אם Y אכן מופיע ברשימה, הוא חייב להימצא במחציתה הראשונה, ובמקרה (2) הוא חייב להימצא במחציתה השנייה. לפיכך, יכולים אנו להגביל את המשך החיפוש למחצית המתאימה של הרשימה.

אי לכך, ההשוואה הבאה תהיה בין Y לבין האיבר האמצעי באותה מחצית: X_{250001} במקרה (1) ו- X_{750001} במקרה (2). ושוב, תוצאת ההשוואה הזאת תהיה צמצום האפשרויות למחצית מן הרשימה החדשה, הקצרה יותר; כלומר, לרשימה שאורכה הוא *רבע* מזה של הרשימה המקורית. תהליך זה נמשך כך שבכל שלב מתקצר לחצי אורכה של הרשימה, או במונחים כלליים יותר - מצטמצם לחצי **גודל הבעיה**, עד אשר קורה אחד מן השניים: Y נמצא, ובמקרה זה התהליך מסתיים ומדווח על הצלחה, או שמגיעים לרשימה הריקה הטריוויאלית, ובמקרה זה התהליך מסתיים ומדווח על כישלון.

תהליך זה נקרא **חיפוש בינרי** (binary search).

השיטה `binarySearch` הכתובה להלן, מממשת את האלגוריתם לחיפוש בינרי שתיארנו לעיל. השיטה מקבלת כפרמטרים מערך `a` של מספרים שלמים, ומספר שלם `x` ומחזירה `true` אם המספר `x` נמצא במערך `a`, ו- `false` אחרת. השיטה כתובה קצת אחרת מהשיטה שהוצגה בהרצאה. כאן אנו מחזירים משתנה בוליאני האומר אם המספר `x` נמצא במערך `a` ואילו בהרצאה הציג פרופ' רוזנשיין שיטה המחזירה את המיקום בו מצאנו את `x`, ואם `x` לא נמצא במערך, השיטה החזירה את הערך `-1`.

```
static int binarySearch (int [] a, int x)
{
   int low = 0, high = max-1, mid;

   while (low <= high)
   {
      mid = (low + high)/2;
      if (a[mid] == x)
          return true;
      if (a[mid] < x)
          low = mid + 1;
      else
          high = mid - 1;
   }
   return false;
}
```

מהי סיבוכיות הזמן של החיפוש הבינרי? כדי לענות על שאלה זו, הבה נמנה ראשית את ההשוואות. נוכל ללמוד משהו על התנהגות אלגוריתם החיפוש הבינרי על-ידי כך שנגסה לחשב כמה השוואות הוא ידרוש, במקרה הגרוע ביותר, עבור ספר הטלפונים שלנו המכיל מיליון שמות. זכור שהחיפוש הנאיבי ידרוש מיליון השוואות.

ובכן, במקרה הגרוע ביותר (מהי דוגמה של מקרה כזה? כמה מקרים גרועים ביותר ישנם?) ידרוש האלגוריתם 20 השוואות בלבד! יתרה מזאת, ככל ש- N גדול יותר כך השיפור מרשים יותר. ספר טלפונים בינלאומי, המכיל, למשל, מיליארד שמות, ידרוש לכל היותר 30 השוואות במקום מיליארד!

זכור שכל השוואה מקצצת את אורכה של רשימת הקלט L לחצי, ושהתהליך מסתיים לכל המאוחר כאשר מתקבלת רשימה ריקה. אי לכך, כדי למצוא את מספר ההשוואות הנדרש במקרה הגרוע ביותר עלינו לחשב כמה פעמים ניתן לחלק מספר N ב- 2 עד שהתוצאה היא 0. (חוקי המשחק מורים להתעלם משברים.) מספר זה נקרא **לוגריתם לפי בסיס 2**, (base-2 logarithm) והוא מסומן $\log_2 N$. למעשה אנו מעוניינים ב- $1+\log_2 N$. בכל מקרה, אפשר לומר בבטחה שהאלגוריתם דורש $O(\log_2 N)$ השוואות במקרה הגרוע ביותר.

נוכל לקבל תחושה על מידת השיפור שמשיג החיפוש הבינרי אם נתבונן בטבלה הבאה, המציגה כמה ערכים של N, ועבור כל N - את מספר ההשוואות הנדרש לחיפוש בינרי במקרה הגרוע ביותר:

$1 + \log_2 N$	N
4	10
7	100
10	1000
20	מיליון
30	מיליארד
60	מיליארד מיליארדים

ראוי לציין שאנו עצמנו משתמשים בוואריאציה של החיפוש הבינרי בעת שאנו מחפשים ידנית בספר הטלפונים. ההבדל הוא בכך שאיננו משווים בהכרח את השם שבידינו אל זה המופיע בדיוק באמצע הספר, ואז אל זה שבמיקום ה- 1/4 או ה- 3/4 וכן הלאה. במקום זאת, אנו משתמשים בידע נוסף שיש לנו, הנוגע להתפלגות הצפויה של שמות בספר טלפונים. אם מחפשים אנו למשל את "מרים סגל", נפתח את הספר בערך בנקודת השני-שליש. מובן שאנו עובדים באופן הרחוק מלהיות מדויק, ואנו נעזרים בחוקי אצבע אינטואיטיביים, גסים, הנקראים **היוריסטיקה** (heuristics). אף על פי כן, ישנם ניסוחים מדויקים של אלגוריתמי חיפוש המוכתבים על-ידי האופי וההתפלגות של האיברים. ככלל, למרות שאלגוריתמים אלה חסכוניים בממוצע במידה ניכרת, גם הם דורשים במקרה הגרוע ביותר סדר גודל של $O(\log_2 N)$ השוואות.

? שאלה 2

נתאר אלגוריתם נוסף לחיפוש איבר במערך ממוין בגודל n, הנקרא **חיפוש טרינרי**: במקום לחצות את המערך לשניים בכל שלב, נחלק את המערך לשלושה חלקים שווים ככל האפשר. נבצע שתי השוואות: עבור האיבר במקום ה- n/3 עבור האיבר במקום ה- 2n/3, ולפי השוואות אלו נדע באיזה שליש מערך להמשיך לחפש.

א. האם אלגוריתם זה יעיל יותר מהאלגוריתם לחיפוש בינרי?

ב. האם תשובתך תשתנה אם נחלק את המערך לארבעה או לחמישה חלקים שווים ככל האפשר?

חסינותו של הסימון O-גדול

כפי שהסברנו, O-ים גדולים מסתירים גורמים קבועים. לפיכך לא היה לנו צורך להיכנס לפרטיהן של ההוראות המסוימות המרכיבות את האלגוריתם תוך כדי ניתוח המקרה הגרוע

ביותר של החיפוש הבינרי. די היה לנתח את מבנה הלולאות של האלגוריתם, ולשכנע את עצמנו שבין כל שתי השוואות מבוצעות, במקרה הגרוע ביותר, רק מספר קבוע של הוראות.

משהו כאן לא נשמע לגמרי בסדר. הרי לא ייתכן שסיבוכיות הזמן של חיפוש בינרי אינה תלויה כלל בפעולות היסוד המותרות. אם היה מותר להשתמש בהוראה:

חפש את האיבר Y ברשימה L

היה האלגוריתם מכיל הוראה אחת ויחידה, וסיבוכיות הזמן שלו הייתה אפוא $O(1)$, כלומר מספר קבוע שאינו תלוי כלל ב- N. כיצד אפוא יכולים אנו לומר שפרטי ההוראות אינם חשובים?

ובכן, סיבוכיות הזמן של אלגוריתם הינה אכן מושג יחסי שמקבל משמעות אך ורק ביחד עם קבוצה מוסכמת של הוראות יסוד. מכל מקום, בדרך כלל אנו חושבים על סוגים סטנדרטיים של בעיות בהקשר של סוגים סטנדרטיים של הוראות יסוד. בבעיות חיפוש ומיון, למשל, משתמשים על פי רוב בהשוואות, בעדכוני אינדקסים ובבדיקות סוף-רשימה, כך שניתוח הסיבוכיות נערך ביחס להוראות אלה. יתר על כן, אם נקבעת שפת התכנות מראש, אזי נקבעת בכך גם קבוצה מסוימת של הוראות יסוד, לטוב או לרע. מה שהופך קביעות אלה למשמעותיות היא העובדה שעבור רוב הסוגים המקובלים של פעולות יסוד, אנו יכולים להרשות לעצמנו ערפול מסוים בתיאור הסיבוכיות במונחי סדר גודל. מובן שכתיבת אלגוריתם בשפת תכנות מסוימת, או שימוש במהדר מסוים, יכולים להשפיע על זמן הריצה הסופי. אולם, אם האלגוריתם משתמש בהוראות יסוד מקובלות, יסתכמו ההבדלים אך ורק בגורם קבוע לכל הוראת יסוד. פירושו של זה הוא שהסיבוכיות, במונחי O-גדול, אינה רגישה להבדלים מסוג זה במימוש האלגוריתם.

במלים אחרות, כל עוד הוסכם על הקבוצה הבסיסית של הוראות היסוד המותרות, וכל עוד כל קיצור המשמש בתיאורים ברמה גבוהה אינו מסתיר איטרציות לא-חסומות של הוראות כאלה אלא מייצג לא יותר מאשר אוסף סופי שלהן, הערכות O-גדול הינן *חסינות*.

לתועלת המתמצאים בלוגריתמים יש להוסיף שגם לבסיס הלוגריתם אין למעשה כל חשיבות. עבור כל מספר קבוע C, המספרים $\log_2 N$ ו- $\log_C N$ אינם נבדלים זה מזה אלא בגורם קבוע, ולכן ניתן להסתיר גם הפרש זה מתחת ל- O הגדול. מסיבה זו נציין בהמשך כל ביצוע בזמן לוגריתמי פשוט על-ידי $O(\log N)$, ולא $O(\log_2 N)$.

בחסינותו של הסימון O-גדול טמונים הן כוחו והן חולשתו. אם נתונים לנו שני אלגוריתמים, האחד A שזמן ביצועו לוגריתמי והאחר B שזמן ביצועו לינארי, ייתכן בהחלט שעל קלטים מסוימים רץ B מהר יותר מאשר A! הסיבה לכך נעוצה בגורמים הקבועים החבויים. הבה נאמר שחישבנו, בעבודת נמלים, את זמן ריצתו המדויק של האלגוריתם, תוך כדי שאנו

לוקחים בחשבון את המספר הקבוע של הוראות היסוד, את שפת התכנות שבה מקודד האלגוריתם, את המהדר המתרגם את הקוד לקוד ברמה נמוכה יותר, את הוראות המכונה הבסיסיות שבהן משתמש המחשב המריץ את קוד המכונה ואת מהירותו של המחשב עצמו. לאחר שעשינו כל זאת, אפשר שנגלה כי זמן ריצתו של האלגוריתם A חסום על-ידי $K\times\log_2 N$ מיקרו-שניות, וזה של B על-ידי $J\times N$ מיקרו-שניות, אולם K הוא 1000 ו- J הוא 10. פירושו של דבר הוא, כפי שהנך יכול לוודא, שעבור כל קלט שאורכו פחות מכ- 1000 (ליתר דיוק, פחות מ- 996), יש לאלגוריתם B עליונות על A. רק כאשר מגיעים לקלטים מגודל 1000 או יותר מתגלה ההפרש שבין N לבין $\log_2 N$ לעין, וכפי שרמזנו כבר קודם, כשמתחיל ההפרש לעבוד לטובתנו, הוא עושה זאת בנדיבות רבה: כאשר מגיעים לקלטים בגודל מיליון, הופך האלגוריתם A יעיל פי 500 מאלגוריתם B, ועבור קלטים מגודל מיליארד השיפור הוא ביותר מפי 330000!

וכך, על משתמש המעוניין רק בקלטים מאורך קטן מ- 1000 לאמץ ללא ספק את אלגוריתם B, למרות עליונותו של A במונחי סדר-גודל. אולם, ברוב המקרים אין הגורמים הקבועים רחוקים כל כך זה מזה כמו 10 ו- 1000, ומכאן שאומדני ה- O הגדול הם בדרך כלל מציאותיים הרבה יותר מאשר הדוגמה שהבאנו כאן.

מוסר ההשכל של הסיפור הוא שיש לחפש קודם כל אלגוריתם טוב ויעיל, תוך שימת דגש על ביצועים במונחי O-גדול, ואז לנסות ולשפר אותו בעזרת תחבולות מן הסוג שהשתמשנו בו קודם להקטנת הגורמים הקבועים. בכל מקרה, הואיל ויעילות O-גדול עלולה להטעות, יש לנסות ולהריץ אלגוריתמים מועמדים ולבדוק את ביצועי הזמן שלהם על סוגים טיפוסיים שונים של קלט.

כאשר אלגוריתם מתבצע בזמן קבוע של פעולות, ללא תלות באורך הקלט, אנו אומרים כי סדר גודל זמן הריצה של האלגוריתם הוא $O(1)$. דוגמא לאלגוריתם כזה הוא מציאת המינימום במערך ממוין. ברור שאם המערך ממוין, האיבר המינימלי שבו נמצא בתא הראשון במערך, ולא משנה מהו גודלו של המערך. לכן, כדי להדפיס את האיבר המינימלי צריך לפנות לתא הראשון, וזו פעולה אחת, בכל מקרה. כלומר, סיבוכיות זמן הריצה היא קבועה ונכתבת כ- $O(1)$.

? שאלה 3

לפניך רשימה של זמני ריצה עבור כמה אלגוריתמים. מיינו אותם מהטוב ביותר לגרוע ביותר, וציינו לצד כל זמן ריצה את סדר הגודל שלו:

1. $10n$
2. n^2
3. 2^n

4. $n^2 + n$

5. n^3

6. $n \cdot \log n + n$

7. $2^n + n \cdot \log n$

8. $n \cdot \log n + 10 \cdot \log n$

שאלה 4 ?

לפניך קטעי תכניות שונים. בכל תכנית אנו משתמשים בצעדים המורכבים מפעולות בסיסיות, *basic_step*. צעדים אלו אורכים זמן קבוע. לכל אחד מהקטעים, קבע מהו זמן הריצה ומהו סדר הגודל של זמן הריצה כפונקציה של n.

```
1. for (j=1; j<n; j++)
      basic_step;

2. for (i=n; i>1; i--)
      for (j=0; j<n; j++)
            basic_step;

3. for (i=1; i<10; i++)
      basic_step;

4. for (i=1; i<n/2; i++)
      basic_step_1;
   for (i=0; i<n; i++)
      for (j=0; j<i; j++)
            basic_step_2;

5. for (i=0; i<n/2; i++)
      for (j=0; j<15; j++)
            basic_step;

6. for (i=0; i<n; i++)    {
      basic_step_1;
      for (j=1; j<n; j++)
            basic_step_2;
   }
```

מיון מערך

בהרצאה ראינו את אחת הבעיות המפורסמות במדעי המחשב - בעיית המיון. בעיית המיון עוסקת במיון איברים במערך חד-ממדי. באומרנו "מיון אוסף ערכים" אנו מתכוונים לסידור הערכים בסדר עולה (כלומר מהקטן לגדול) או בסדר יורד (מהגדול לקטן). הצורך במיון יעיל עולה במגוון רחב ביותר של שימושים במחשבים. למשל, ראינו קודם אלגוריתמים שונים לחיפוש איבר ברשימה. אם הרשימה אינה ממוינת, איננו יכולים להשתמש באלגוריתם לחיפוש בינרי, ואנו נאלצים להשתמש בחיפוש הסדרתי, דבר המאריך מאוד את זמן פעולת החיפוש.

פרופ' רוזנשיין הציג את הבעיה וכמה אלגוריתמים לפתרונה. ביחידה 14 הוצגו כמה אלגוריתמים ריבועיים (מיון בחירה – Selection sort, מיון הכנסה – Insertion sort, מיון בועות - Bubble sort), וביחידה 16 הוצג אלגוריתם מיון-מהיר (Quick sort) שבמקרה הממוצע לוקח זמן של $O(n \log_2 n)$.

האם האלגוריתם מיון-מהיר הוא אמנם שיפור של ממש בהשוואה לאלגוריתם מיון-בועות, למשל? נסתכל על הטבלה להלן וניווכח בעצמנו, נראה שללא ספק, השיפור הוא גדול מאוד.

$n \times \log_2 n$	n^2	n
40	100	10
$50 \times 6 = 300$	2500	50
$100 \times 7 = 700$	10,000	100
$1,000 \times 10 = 10,000$	1,000,000	1,000
20 מיליון	אלף מיליארד	מיליון

סיבוכיות זמן ריצה מעריכית

נחזור לבעיה בה פתחנו את היחידה, צביעת גרף. בעיה זו ודומותיה, לפחות למיטב ידיעותינו הנוכחיות, נחשבות **בעיות בלתי סבירות** (intractable problems). באלגוריתמים הידועים לפתרון בעיות אלו, פונקציות זמן הריצה גדלות בקצב מהיר כל כך, שאי אפשר להשתמש בהם אפילו עבור קלטים קצרים למדי.

מתי זמן הריצה נחשב לבלתי סביר? כאשר הפונקציה המתארת אותו היא **פונקציה מעריכית**, או **אקספוננציאלית** (exponential function). בפונקציות אלה n, שהוא אורך הקלט, מופיע במעריך החזקה (והבסיס גדול מ- 1).

הפונקציות 2^n, 3^n ו- 5^n הן דוגמאות לפונקציות מעריכיות פשוטות, וקצב גידולן מהיר מאוד. קיימות פונקציות זמן ריצה שגדלות בקצב מהיר אף יותר; למשל: $n \cdot 2^n$, $n!$ ו- n^n. כולן נחשבות לבלתי סבירות מבחינת זמן הריצה שלהן.

ישנן בעיות עבורן אפשר להוכיח כי זמן הריצה הטוב ביותר של האלגוריתמים הפותרים אותן הוא אקספוננציאלי. למשל, בעיית מגדלי האנוי , אותה נלמד ביחידה העוסקת ברקורסיה. אפשר להוכיח כי $O(2^n)$ הוא זמן הריצה הטוב ביותר לבעיית מגדלי האנוי, כלומר אין אלגוריתם שיפתור את הבעיה בזמן קצר יותר. לעומת זאת, בבעיות האחרות שהבאנו: צביעת גרף, הסוכן הנוסע ופירוק מספר לגורמים, הדבר אינו פשוט כל-כך. האלגוריתמים הטובים ביותר הידועים עד כה לפתרון בעיות אלו הם אכן אקספוננציאליים, אך אף אחד עדיין לא הוכיח שאי אפשר לפתור את

הבעיות בזמן קצר יותר, סביר, או פולינומיאלי (polynomial). זהו אחד מהנושאים החשובים ביותר בתחום המחקר במדעי המחשב, וטובי המדענים בתחום בכל העולם שוקדים על פתרונו.

לסיום, נביא טבלה ובה הדגמה של קצב הגדילה היחסי בין כמה פונקציות (פולינומיאליות ואקספוננציאליות).

הפונקציה	**n = 10**	**n = 50**	**n = 100**	**n = 300**	**n = 1000**
$5n$	50	250	500	1500	5000
$n \times \log_2 n$	33	282	665	2469	9966
n^2	100	2500	10,000	90,000	1,000,000
n^3	1000	125,000	1,000,000	27 מיליון (7 ספרות)	מיליארד (10 ספרות)
2^n	1024	מספר בן 16 ספרות	מספר בן 31 ספרות	מספר בן 91 ספרות	מספר בן 302 ספרות
$n!$	3,600,000	מספר בן 65 ספרות	מספר בן 161 ספרות	מספר בן 623 ספרות	מספר גדול לאין שיעור
n^n	10 מיליארד (11 ספרות)	מספר בן 85 ספרות	מספר בן 201 ספרות	מספר בן 744 ספרות	מספר גדול לאין שיעור

לצורך ההשוואה: מספר הפרוטונים ביקום המוכר לנו הוא כבן 80 ספרות (כלומר 10^{80} בערך).
מספר המיקרו-שניות שחלפו מאז המפץ הגדול הוא כבן 24 ספרות.

תשובות לשאלות

תשובה 1

פעולות בסיסיות:

א. מציאת אורך הטקסט באלגוריתם המחפש מחרוזת בטקסט אינה יכולה להיות פעולה בסיסית כיוון שהיא תלויה באורך הקלט.

ב. פעולת חילוק בין שני מספרים ממומשת במחשבים על-ידי חמרה ולכן זמן ביצועה קבוע, כלומר היא פעולה בסיסית.

ג. חישוב $\sum_{i=1}^{n} i$ כאשר n הוא הקלט אינה פעולה בסיסית, כיוון שזמן החישוב של פעולה זו תלוי בגודלו של n. בעת החישוב אנו מבצעים n-1 פעולות חיבור.

ד. פעולת המעבר לתא הבא במערך אורכת זמן קבוע שאינו תלוי באורך הקלט, ולכן היא יכולה להיות פעולה בסיסית.

תשובה 2

א. נזכור שזמן הריצה של האלגוריתם לחיפוש בינרי הוא $O(\log_2 n)$. בדומה להוכחה זו אפשר להוכיח כי זמן הריצה של האלגוריתם לחיפוש טרינרי הוא $O(\log_3 n)$. שכן, בהנחה ש- n הוא חזרה של 3, מספר פעמים שצריך לחלק את n כדי לקבל 1 הוא $\log_3 n$. אמנם בכל צעד של חיפוש יש השוואה נוספת על זו שבחיפוש בינרי, אך זהו רק שינוי בקבוע.

אפשר להוכיח כי היחס בין שני זמני הריצה האלו הוא קבוע, ולכן הם שייכים לאותו סדר גודל.

$$\frac{\log_3 n}{\log_2 n} = \log_2 3$$

לכן, למרות שלכאורה נראה כי החיפוש הטרינרי יעיל יותר מהחיפוש הבינרי, בגלל הקטנת תחום החיפוש בכ- 2/3 בכל פעם, הרי שאנו רואים שסדר הגודל של זמן הריצה של שני החיפושים שווה.

ב. בצורה דומה אפשר לראות שגם אם נחלק את המערך לארבעה או חמישה חלקים, נישאר באותו סדר גודל. יתר על כן, כאשר מחלקים את התחום ליותר חלקים, צריך לבצע יותר השוואות. וכן, אנו רוצים לחלק את התחום לחלקים שווים ככל האפשר, אך זה אינו פשוט כל כך.

תשובה 3

זמני הריצה מסודרים בסדר עולה:

1. $10n$ - סדר גודל לינארי
2. $n \cdot \log n + 10 \cdot \log n$ - $O(n \cdot \log n)$
3. $n \cdot \log n + n$ - $O(n \cdot \log n)$
4. n^2 - סדר גודל ריבועי
5. $n^2 + n$ - סדר גודל ריבועי
6. n^3 - $O(n^3)$
7. 2^n - סדר גודל מעריכי
8. $2^n + n \cdot \log n$ - סדר גודל מעריכי

תשובה 4

קטע התכנית	זמן ריצה	סדר גודל
1. `for (j=1; j<n; j++)` `basic_step;`	$n-1$	$O(n)$

2. `for (i=n; i>1; i--)`
`for (j=0; j<n; j++)`
`basic_step;`
	$n\cdot(n-1) = n^2 - n$	$O(n^2)$

3. `for (i=1; i<10; i++)` `basic_step;`	10	$O(1)$

4. `for (i=1; i<n/2; i++)`
`basic_step_1;`
`for (i=0; i<n; i++)`
`for (j=0; j<i; j++)`
`basic_step_2;`
	$(n/2 - 1) + (1+2+...+ n-1)$
	$= (n/2 -1) + n\cdot(n-1)/2$
	$= n^2/2 - 1$
		$O(n^2)$

5. `for (i=0; i<n/2; i++)`
`for (j=0; j<15; j++)`
`basic_step;`
	$n/2 \cdot 15$	$O(n)$

6. `for (i=0; i<n; i++){`
`basic_step_1;`
`for (j=1; j<n; j++)`
`basic_step_2;`
`}`
	$n \cdot (1 + n-1) = n^2$	$O(n^2)$