

מבני נתונים – סיכום למבחן, יולי 2009

מאת: סשה גולדשטיין, sashag@cs

עדכון אחרון: 3.7.09, בשעה 17:50

הגבלת אחריות

הסיכום להלן הוא האינטרפרטציה שלי של החומר, שממש לא חייבת להיות נכונה או מייצגת את זו של הסגל. בכל זאת, אני מניח שהרשימות להלן – שמבוססות על סיכומי שיעורים ותרגולים, וכן על המצגות של התרגולים, הפתרונות של התרגילים ומקורות חיצוניים – יכולות להיות לעזר בהכנה למבחן. מה שאני מנסה להגיד כאן הוא: השימוש בסיכום הזה הוא על אחריותכם בלבד. אל תאשימו אף אחד אח"כ אם במבחן שאלו משהו שלא היה בסיכום, או שמופיע בסיכום עם טעות. דרך אגב, מסיבה זו בדיוק – אשמח מאוד אם תעבירו אליי הערות, תיקונים או כל דבר אחר בנוגע לרשימות האלה (sashag@cs).

פרק 1 - אלגוריתמי מיון ואנליזה אסימפטוטית

מיון בועות – Bubble Sort

נתונים איברים למיון בתוך מערך. נפעפע את האיבר הגדול ביותר לחלק של המערך שעוד לא מוין. לאחר שלב k יש לנו $n - k$ איברים שטרם מויינו, ו- k איברים בצידו הימני של המערך שכבר ממוינים (בסדר עולה).

BUBBLE SORT($A[1 \dots n]$)

for $i \leftarrow 1$ to $n - 1$

for $j \leftarrow 1$ to $n - i$

if $A[j] > A[j + 1]$ then swap($A[j], A[j + 1]$)

ניתוח זמן הריצה של האלגוריתם משמעותו חישוב מספר הצעדים הבסיסיים שהאלגוריתם מבצע עבור קלט בגודל n . אם אנו יודעים את המחיר (הקבוע) של הצעדים הבסיסיים, נוכל להסיק את התנהגות זמן הריצה כפונקציה של n , ונסמן כאן $T(n)$.

הגדרה: צעדים בסיסיים: השמה, השוואה, החלפה, קידום משתנה (כולם בעלות של 1).

נחשב את זמן הריצה בהנחה שפעולת ההחלפה (כמו כל שאר הפעולות) עולה יחידת זמן אחת:

$$\begin{aligned} T(n) &= \sum_{i=1}^{n-1} \left[1 + \sum_{j=1}^{n-i} (1 + 1 + 1) \right] = \sum_{i=1}^{n-1} 1 + \sum_{i=1}^{n-1} \sum_{j=1}^{n-i} 3 = n - 1 + 3 \sum_{i=1}^{n-1} (n - i) = n - 1 + \frac{3n(n-1)}{2} \\ &= \frac{3}{2}n^2 - \frac{1}{2}n - 1 \end{aligned}$$

לעומת זאת, אם למשל נבחר להתבונן בפעולת ההחלפה כפעולה שעולה 4 יחידות זמן, נקבל:

$$T(n) = \sum_{i=1}^{n-1} \left(2 + \sum_{j=1}^{n-i} 6 \right) = \dots = 3n^2 - n - 2$$

לכאורה קיבלנו שני ביטויים שונים לחלוטין, אך מה כאן העיקר ומה הטפל? האם יש משמעות לתמחור הצעדים הבסיסיים כאשר מעניינת אותנו בעיקר ההתנהגות האסימפטוטית של הקלט (כאשר $n \rightarrow \infty$)?

כדי לנתח אלגוריתמים ללא תלות במערכת שעליה הן רצות (שמשפיעה על תמחור הצעדים הבסיסיים), נרצה רק להבחין בין הפעולות הבסיסיות – שלוקחות זמן קבוע – לבין כל דבר אחר, כגון לולאות למשל, שזמן הביצוע שלו תלוי בגודל הקלט.

אנליזה אסימפטוטית

הגדרה: $f(n) = \mathcal{O}(g(n))$ אם $\exists c > 0, n_0 \in \mathbb{N} : \forall n > n_0, f(n) \leq cg(n)$

הגדרה: $f(n) = \Omega(g(n))$ אם $\exists c > 0, n_0 \in \mathbb{N} : \forall n > n_0, f(n) \geq cg(n)$

הגדרה: $f(n) = \Theta(g(n))$ אם $f(n) = \mathcal{O}(g(n))$ וגם $f(n) = \Omega(g(n))$

הגדרה: $f(n) = o(g(n))$ אם $\lim_{n \rightarrow \infty} \left| \frac{f(n)}{g(n)} \right| = 0$

טענה 1: הסימונים האסימפטוטיים הם טרנזיטיביים, למשל:

$$f(n) = \Omega(g(n)), g(n) = \Omega(h(n)) \Rightarrow f(n) = \Omega(h(n))$$

הוכחה: מיידיית מההגדרות.

הערה: בדרך כלל אנו עוסקים בפונקציות חיוביות אם הפונקציות אינן חיוביות, כמעט לא ניתן לומר דבר על היחסים ביניהן. למשל ייתכן שאם נסכום שתי פונקציות (שהאחת שלילית והאחרת חיובית) הן יבטלו זו את זו, ולעומת זאת בניתוח אסימפטוטי רגיל אנו לרוב מניחים שסכום של שתי פונקציות הוא לפחות Θ של כל אחת מהן בנפרד.

טענה 2: אם $f(n) = \Theta(g(n) + h(n)), g(n) = o(h(n))$ אזי $f(n) = \Theta(h(n))$

הוכחה: ר' פתרון תרגיל 1, מההגדרות.

דוגמה לניתוח סיבוכיות באמצעות אנליזה אסימפטוטית – נחזור למיון הבועות שראינו קודם. ראינו כבר ש- $T(n) = \frac{3}{2}n^2 - \frac{1}{2}n - 1$. אנו רוצים להשתמש בנוסחה אסימפטוטית.

טענה 3: סיבוכיות זמן הריצה של מיון בועות היא $\Theta(n^2)$.

הוכחה: נתבונן הפוך – נבחר קבוע $c = 1$ וכעת אנו רוצים להראות שקיים n_0 המקיים $T(n) \geq n^2 \forall n > n_0$. נציב באי שוויון:

$$\frac{3}{2}n^2 - \frac{1}{2}n - 1 \geq n^2 \Leftrightarrow n(n-1) \geq 2 \Leftrightarrow n_0 \geq 2$$

כלומר אכן עבור $n_0 = 2$ מתקיים $T(n) = \Omega(n^2)$ ובאופן סימטרי מראים $T(n) = \mathcal{O}(n^2)$ ולכן $T(n) = \Theta(n^2)$.

■

כללי אצבע לגבי התנהגות אסימפטוטית:

1. קבועים כפליים לא משנים, למשל $f(n) = \mathcal{O}(4n^2) \Leftrightarrow f(n) = \mathcal{O}(n^2)$ וזאת מאחר שמותר לנו לבחור קבוע c גדול ככל שנרצה, או קטן ככל שנרצה.

2. **טענה 2:** אם $f(n) = \Theta(h(n))$ אזי $f(n) = \Theta(g(n) + h(n))$, $g(n) = o(h(n))$

3. **טענה 4:** אם $f(n) = n^d$, $g(n) = c^n$ (כאשר $c > 1, d > 1, d \in \mathbb{N}$) אזי $f(n) = \mathcal{O}(g(n))$ ואפילו $f(n) = o(g(n))$

4. **טענה 5:** אם $f(n) = \log(n)^c$, $g(n) = n^d$ אזי $f(n) = \mathcal{O}(g(n))$ ואפילו $f(n) = o(g(n))$

הערה: בקורס זה אנו תמיד מתכוונים ללוגריתם בבסיס 2 (השקול עד כדי קבוע ללוגריתם בכל בסיס אחר).

הוכחת טענות 4-5: נראה שלא קיים $n_0 \in \mathbb{N}$ כך ש- $n^c > d \log(n)$. נתבונן בגבול (ונשתמש בכלל לופיטל):

$$(*) \lim_{n \rightarrow \infty} \frac{d \ln(n)}{n^c} = \lim_{n \rightarrow \infty} \frac{d \ln(n)'}{n^{c'}} = \lim_{n \rightarrow \infty} \frac{\frac{d}{n}}{c n^{c-1}} = 0$$

מהגדרת הגבול קיבלנו שלא קיים n_0 כנ"ל. כעת כדי להוכיח את טענה 4 נשים לב ש-

$$n^d = c^{\log_c(n)d} = c^{\left(\frac{d}{\log(c)}\right) \log(n)}$$

ועכשיו מ- (*) מקבלים (עבור $d \log(c) < 1$) $n^d < c^n$ כמו שרצינו. באופן דומה עבור טענה 5 מקבלים:

$$\log(n)^c < n^d \Leftrightarrow \log(n) < n^{\frac{d}{c}}$$

ויש לנו בדיוק את מה שרצינו. ■

ניתוח סיבוכיות של אלגוריתמים רקורסיביים

ראשית, נכתוב את מיון הבועות בתור אלגוריתם רקורסיבי (כאשר בכל פעם ממיינים מערך בגודל קטן יותר):

```
BUBBLE SORT(A[1 ... n])
if n > 1
  BUBBLE(A[1 ... n])
  BUBBLE SORT(A[1 ... n - 1])

BUBBLE(A[1 ... n])
for j ← 1 to n - 1
  if A[j] > A[j + 1] then swap(A[j], A[j + 1])
```

ניתן לראות בקלות ש- **BUBBLE** פועל ב- $\Theta(n)$, כי הלולאה מתבצעת $\Theta(n)$ פעמים, ובתוך הלולאה אנו עושים לכל היותר מספר קבוע של פעולות (השוואה, החלפה וכד'). ניתן לכתוב את משוואת הרקורסיה של האלגוריתם על ידי הבנת הקשר בין זמן הריצה של האלגוריתם עבור קלט בגודל n ועבור קלטים קטנים יותר (ברקורסיה). במקרה שלנו:

$$T(n) = T(n - 1) + \Theta(n)$$

$$T(1) = \Theta(1)$$

כיצד פותרים משוואות רקורסיביות מהסוג הזה? אחת הדרכים היא השיטה האיטרטיבית, שכרוכה בהצבה אחר הצבה עד ש"ברור" מה הביטוי שמתקבל. במקרה שלנו, מהגדרת $\Theta(n)$ מתקבל:

טענה 6א': סיבוכיות זמן הריצה של *BUBBLE SORT* הרקורסיבי היא $\Theta(n^2)$

הוכחה:

$$\begin{aligned} T(n) &\leq T(n-1) + cn \leq T(n-2) + c(n-1) + cn \leq T(n-3) + c(n-2) + c(n-1) + cn \leq \dots \\ &\leq T(1) + 2c + 3c + \dots + (n-1)c + cn = T(1) + \sum_{i=2}^n i = T(1) + c \left(\frac{n(n+1)}{2} - 1 \right) \end{aligned}$$

בביטוי שקיבלנו מופיע n^2 וכמובן כל הקבועים הכפליים והביטויים הזניחים יחסית ל- n^2 מושמטים, ולכן קיבלנו $T(n) = \mathcal{O}(n^2)$ ובאופן סימטרי לחלוטין נוכל להראות את הכיוון ההפוך – $T(n) = \Omega(n^2)$ ולהסיק ש- $T(n) = \Theta(n^2)$, כלומר שהגרסה הרקורסיבית של מיון הבועות מתנהגת באותו אופן (אסימפטוטית) כמו הגרסה האיטרטיבית. ■

הערה: אמנם בהגדרת $\Theta, \mathcal{O}, \Omega$ מופיע גם n_0 וגם c , אך קל לראות שאם בוחרים c כאוות נפשנו, ניתן להגיע למצב שבו $n_0 = 1$ ואז לא צריך להתחשב בו בכלל.

דרך אחרת לפתור משוואת רקורסיה מסוג זה היא שיטת הניחוש וההוכחה באינדוקציה. כאן פשוט ננחש ש- $T(n) = \mathcal{O}(n^2)$ וננסה להוכיח זאת באינדוקציה.

טענה 6ב': $T(n) \leq cn^2$ עבור c כלשהו.

הוכחה:

בסיס האינדוקציה: $n = 1, T(1) \leq c \Rightarrow T(1) = \mathcal{O}(1) \Rightarrow \exists c' : T(1) \leq c'$

הנחת האינדוקציה: עבור n , מתקיים $T(n) \leq cn^2$.

צעד האינדוקציה: רוצים להראות ש- $T(n+1) \leq c(n+1)^2$. נציב:

$$T(n+1) \leq T(n) + c(n+1) \leq cn^2 + c(n+1) = c(n^2 + n + 1) \leq c(n+1)^2$$

במעבר הראשון השתמשנו בהגדרת Θ במשוואת הרקורסיה, במעבר השני השתמשנו בהנחת האינדוקציה על $T(n)$ והמעבר השלישי ברור. לכן קיבלנו $T(n) = \mathcal{O}(n^2)$ ובאופן דומה מוכיחים את הצד השני של הטענה. ■

מיון מיזוג – Merge Sort

נתבונן באלגוריתם מיון אחר – מיון מיזוג – ונפעיל עליו את עקרונות הניתוח הרקורסיבי שראינו קודם, וכן עקרונות מתקדמים יותר. זהו אלגוריתם ממשפחת "הפרד ומשול" (Divide and Conquer), ובו מפרקים את הבעיה לחלקים קטנים, פותרים את הבעיות הקטנות ולבסוף מאחדים את התוצאות החלקיות יחד.

במקרה שלנו, מחלקים את המערך לחצאים עד שמגיעים למערך בגודל 1 שהוא מקרה הבסיס, ואז מאחדים את התוצאות החלקיות למערך ממוין.

MERGE SORT($A[1 \dots n]$)

if $n > 1$

MERGE SORT ($A[1 \dots \frac{n}{2}]$)

MERGE SORT ($A[\frac{n}{2} + 1 \dots n]$)

MERGE ($A[1 \dots \frac{n}{2}], A[\frac{n}{2} + 1 \dots n], A[1 \dots n]$)

MERGE($A, B, C[1 \dots n]$)

$l \leftarrow 1, i \leftarrow 1, j \leftarrow 1$

do while $i \leq \frac{n}{2}$ or $j \leq \frac{n}{2}$

$C[l] \leftarrow \min(A[i], B[j])$

 if minimum was $A[i]$, increment i , else increment j

 increment l

המיון עצמו מתבצע כאן במהלך המיזוג – המיזוג מבטיח שבמערך C , שהוא הפלט של האלגוריתם, האיברים נמצאים בסדר ממזין. ניתן להתבונן בעץ הרקורסיה על מנת להבין מה קורה במהלך המיון – בכל קודקוד בעץ (עד שמגיעים לעלים) יש התפצלות לשני תת-מערכים, ובכל קודקוד מתבצע **MERGE** של שני הקודקודים שמתחתיו.

כמו כן, ברור ש-**MERGE** היא בעלת סיבוכיות של $\Theta(n)$ מאחר שהאלגוריתם "נוגע" פעם אחת בכל איבר במערך הפלט. לכן נוכל לכתוב את משוואת הרקורסיה של **MERGE SORT**:

$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)$$

$$T(1) = \Theta(1)$$

כעת נרצה לבדוק מה זמן הריצה האמיתי של האלגוריתם.

טענה 7: סיבוכיות זמן הריצה של **MERGE SORT** היא $\Theta(n \log(n))$.

הוכחה: נתבונן בעץ הרקורסיה, בכל שורה בעץ יש k פעולות **MERGE** (כאשר k חזקה של 2), ובכל פעולת **MERGE** מתבצעות $\frac{cn}{k}$ פעולות בסיסיות. לכן בכל שורה יש לנו cn פעולות, וכן יש לנו $\log(n)$ שורות כיוון שבכל פעם שיוורדים בעץ, מחלקים את מספר האיברים במערך ב-2, ועוצרים כשהגענו לעלים שבהם מספר האיברים הוא 1 (הנחנו כאן ש- n חזקה של 2). לכן קיבלנו $T(n) = \mathcal{O}(n \log(n))$ ומשיקולים דומים גם $T(n) = \Omega(n \log(n))$ ולכן $T(n) = \Theta(n \log(n))$. ■

מה שראינו כאן הוא מקרה פרטי של משפט המאסטר – הכללה של משוואות הרקורסיה לכל אלגוריתם מסוג "הפרד ומשול". כעת נייצר את המשפט עבור כל המקרים השונים, ומשוואות הרקורסיה שאנחנו מתבוננים בהן הן מהצורה:

$$T(n) = aT\left(\frac{n}{b}\right) + \Theta(n^k)$$

במלים אחרות, בכל שלב מחלקים את הבעיה ל- a בעיות שכל אחת בגודל $\frac{n}{b}$. במקרה האחרון שהסתכלנו עליו היה לנו $a = b = 2, k = 1$ (עבור **MERGE SORT**).

אם נתבונן בעץ הרקורסיה, אזי ברור שבכל שלב העץ מתפצל ל- a חלקים שבכל אחד מהם הבעיה נהיית קטנה יותר פי b . סכום העבודה בשורה הראשונה הוא פשוט cn^k עבור קבוע כלשהו, בשורה השנייה סכום העבודה הוא $ac \left(\frac{n}{b}\right)^k$, בשורה השלישית סכום העבודה הוא $a^2 c \left(\frac{n}{b^2}\right)^k$ וכן הלאה. אם נסכום את כל העבודה נקבל:

$$cn^k + cn^k \left(\frac{a}{b^k}\right) + cn^k \left(\frac{a}{b^k}\right)^2 + \dots + cn^k \left(\frac{a}{b^k}\right)^l + \dots$$

כמובן שביטוי זה מסתיים לאחר שמגיעים לעלים, כלומר מעניין אותנו כמה שורות יש בעץ. נסמן $m = \log_b(n)$ ואז מספר השורות הוא כמובן $m + 1$. בשורה האחרונה של העץ יש לנו a^m עלים, והעבודה המתבצעת בשורה התחתונה היא לפיכך $\Theta(a^m) = \Theta(a^{\log_b(n)})$. כעת ננסה לחשב את הסכום, שהוא סכום סדרה הנדסית:

$$T(n) \leq cn^k \sum_{l=0}^m \left(\frac{a}{b^k}\right)^l$$

עכשיו נסמן $q = \frac{a}{b^k}$ ונבחין בין שלושה מקרים:

- אם $q < 1$ הסדרה היא סדרה יורדת, והשורה הראשונה תתרום הכי הרבה עבודה:

$$T(n) \leq cn^k \left(\frac{q^{m+1} - 1}{q - 1}\right) = cn^k \left(\frac{1 - q^{m+1}}{1 - q}\right) \leq cn^k \left(\frac{1}{1 - q}\right) = O(n^k)$$

יש לשים לב כאן שבכיוון השני (Ω) צריך עדינות כדי להראות ש- $\frac{1 - q^{m+1}}{1 - q}$ גדול מקבוע.

אם כך במקרה זה מקבלים $T(n) = \Theta(n^k)$ - כלומר העבודה שנעשית בשורה הראשונה של העץ היא הדומיננטית.

- אם $q = 1$ אזי יש לנו סכום של 1, כלומר:

$$T(n) \leq cn^k (m + 1) = cn^k (\log_b(n)) = \Theta(n^k \log(n))$$

כלומר במקרה זה קיבלנו $T(n) = \Theta(n^k \log(n))$, וזה אכן מתאים למה שגילינו על *MERGE SORT*.

- אם $q > 1$, אזי אנחנו מצפים שהגורם הדומיננטי יהיה בשורה האחרונה, כי הסדרה עולה:

$$T(n) \leq cn^k \left(\frac{q^{m+1} - 1}{q - 1}\right) = \Theta(n^k q^m) = \Theta\left(n^k \left(\frac{a}{b^k}\right)^m\right) = \Theta\left(n^k \left(\frac{a^m}{b^{mk}}\right)\right) = \Theta(a^m)$$

$$= \Theta(a^{\log_b(n)}) = \Theta(n^{\log_b(a)})$$

כלומר במקרה זה קיבלנו $T(n) = \Theta(n^{\log_b(a)})$

כך למעשה "חישבנו" את:

משפט 1: (משפט המאסטר) אם $T(n) = aT\left(\frac{n}{b}\right) + \Theta(n^k)$ אזי:

אם $\frac{a}{b^k} > 1$ אזי $T(n) = \Theta(n^{\log_b(a)})$

אם $\frac{a}{b^k} < 1$ אזי $T(n) = \Theta(n^k)$

אם $\frac{a}{b^k} = 1$ אזי $T(n) = \Theta(n^k \log(n))$

הוכחה: החישוב לעיל מהווה הוכחה. ■

מיון הכנסה – Insertion Sort

במיון זה, בכל שלב k יהיו לנו k איברים ממוינים בהתחלה של המערכ, ו- $n - k$ איברים לא ממוינים עדיין בסופו. בכל שלב, ניקח את האיבר ה- $k + 1$ ונכניס אותו למקום הנכון שלו בין $k + 1$ האיברים הראשונים.

INSERTION SORT($A[1 \dots n]$)

```
for  $i \leftarrow 1$  to  $n - 1$ 
     $val \leftarrow A[i]$ 
     $j \leftarrow i - 1$ 
    while  $j \geq 0$  and  $A[j] > val$ 
         $A[j + 1] \leftarrow A[j]$ 
        decrement  $j$ 
     $A[j + 1] \leftarrow val$ 
```

באופן דומה אפשר גם לממש את האלגוריתם בצורה רקורסיבית

INSERTION SORT($A, curr, n$)

```
if  $curr = n + 1$  then return
INSERT ( $A[curr], A[1 \dots curr - 1]$ )
INSERTION SORT( $A, curr + 1, n$ )
```

זמן הריצה של האלגוריתם במקרה שהמערכ ממוין (המקרה הטוב ביותר) הוא $O(n)$, כיוון שצריך רק להשוות את האיבר החדש עם הקצה של הטווח הממוין. במקרה הגרוע, למשל כאשר המערכ ממוין בסדר הפוך, בכל איטרציה יש לסרוק את כל החלק הממוין של המערכ, אז אם נסכום ונזרוק חצי מהאיברים נקבל $O(n^2)$ (באופן דומה להוכחות אחרות). מסתבר שגם המקרה הממוצע של INSERTION SORT הוא $O(n^2)$ ולכן הוא שימושי רק עבור מערכים מאוד קטנים (נחשב מאוד יעיל עבורם) או עבור קלט ממוין בהסתברות גבוהה.

מיון בזמן ליניארי

רוב האלגוריתמים למיון משתמשים בהשוואות על מנת למיין את האיברים במקרה זה (נראה בהמשך), קיים חסם תחתון על סיבוכיות זמן הריצה של האלגוריתם – והוא $\Omega(n \log(n))$ (טענה 13). עם זאת, במקרים מיוחדים אנו יכולים לבצע מיון גם בזמן ליניארי. למשל, האלגוריתם COUNTING SORT מניח שהקלט הוא n מספרים שכולם בטווח $\{1, \dots, k\}$.

COUNTING SORT($A[1 \dots n], B[1 \dots n], k$)

```
initialize  $C[k]$  an array
for  $i \leftarrow 1$  to  $n$ 
    increment  $C[A[i]]$ 
for  $j \leftarrow 2$  to  $k$ 
     $C[j] \leftarrow C[j] + C[j - 1]$ 
for  $j \leftarrow n$  downto 1
     $B[C[A[j]]] \leftarrow A[j]$ 
    decrement  $C[A[j]]$ 
```

הרעיון הוא לספור את מספר האיברים הקטנים או שווים לכל $j \in \{1, \dots, k\}$. עבור כל j , המספר הזה הוא האינדקס הגבוה ביותר של j במערכ. אנו יכולים להשתמש במידע הזה כדי לשחזר את המערכ הממוין כולו

סיבוכיות זמן הריצה של האלגוריתם היא כמובן $\Theta(n + k)$. מדוע האלגוריתם עובד? לאחר הלולאה הראשונה, ספרנו את מספר הפעמים שכל j מופיע במערך. לאחר הלולאה השנייה, ספרנו את מספר האיברים במערך הקטנים או שווים ל- j . לאחר מכן, רצים על המערך מהסוף להתחלה ובכל פעם האינדקס הגבוה ביותר של j שלא תפוס עדיין במערך נמצא ב- $C[j]$, שאותו מעדכנים לאחר כל הכנסה. כתוצאה מזה, כל איבר נכנס למקום שלו באופן מדויק. ■

הגדרה: אלגוריתם מיון נקרא יציב אם עבור כל $i < j : A[i] = A[j]$ במערך המקורי, המקומות i', j' בהתאמה במערך הממויין מקיימים $i' < j'$.

הערה: המיון *COUNTING SORT* הוא מיון יציב. קל לראות זאת כיוון שהאיברים מהמערך המקורי מוכנסים למערך החדש מהסוף להתחלה, בדיוק לפי הסדר שבו הם היו במערך המקורי.

מיון מהיר – Quick Sort

במיון זה בוחרים pivot – מקום כלשהו במערך – ומחלקים את המערך לחלק הקטן ממנו והחלק הגדול ממנו (תהליך זה נקרא partition). לאחר החלוקה מחזירים את המקום של ה-pivot במערך החדש, וממשיכים למיון ברקורסיה את שני החלקים (משמאל ל-pivot ומימין לו).

QUICK SORT($A[left, right]$)

if $right > left$

$m \leftarrow PARTITION(A[left, right])$

QUICK SORT($A[left, m - 1]$)

QUICK SORT($A[m + 1, right]$)

יש לנו הרבה אפשרויות לבחור את ה-pivot בתוך *PARTITION*, אז במקרה כן נבחר את ה-pivot להיות האיבר הימני ביותר במערך:

PARTITION($A[left, right]$)

$L \leftarrow left, R \leftarrow right, Pivot \leftarrow A[right]$

for $j \leftarrow left$ *to* $right$

if $A[j] < Pivot$ *then* $B[L] \leftarrow A[j]$, *increment* L

else $B[R] \leftarrow A[j]$, *decrement* R

$B[L] \leftarrow A[right]$

copy B *to* A

return L

כאן קיבלנו אלגוריתם אמנם פשוט אבל בזבזני כי אנחנו יוצרים מערך חדש לצורך התוצאות אין בכך צורך – ניתן לממש partition in-place.

כלומר, יהיו לנו שני מצביעים הרצים על המערך (משמאל לימין ומימין לשמאל) והם מתקדמים עד שהם פוגשים משהו "לא מתאים". במקום שבו המצביעים מתנגשים, מכניסים את ה-pivot.

PARTITION2($A[left, right]$)

$L \leftarrow left, R \leftarrow right, Pivot \leftarrow A[right]$

while **TRUE**

while $A[L] \leq Pivot$ *increment* L

while $A[R] \geq Pivot$ *decrement* R

```

if L > R then return R
swap(A[L], A[R])

```

הערה: זו רק אפשרות אחת לכתוב את *PARTITION*, ניתן גם לבצע זאת באמצעות מצביע אחד בלבד (מפתרון תרגיל 3):

```

PARTITION3(A[left, right], pivotIndex)
Pivot ← A[pivotIndex]
swap(A[pivotIndex], A[right])
storedIndex ← left
for i ← left to right - 1
    if A[i] ≤ Pivot then swap(A[i], A[storedIndex]) and increment storedIndex
swap(A[storedIndex], A[right])
return storedIndex

```

כעת נרצה לנתח את סיבוכיות זמן הריצה של האלגוריתם. קל לראות שכל הגרסאות של *PARTITION* פועלות ב- $\Theta(n)$, אפילו הגרסה הבזבזנית ביותר שבה מקצים מערך נוסף. אבל איך נתבונן באלגוריתם כולו? אנו יודעים לכתוב את משוואת הרקורסיה הבאה, כתלות ב- m שהוא מיקום ה-*pivot* לאחר ש-*PARTITION* חוזר:

$$T(n) = T(m - 1) + T(n - m) + \Theta(n)$$

אנחנו לא יודעים איך לנתח את זה כי זה מתנהג באופן שונה עבור כל m , וכמובן שהבחירה של ה- m משתנה בכל שלב ברקורסיה. אבל, אנחנו יכולים לבחון מקרי קצה.

- אם המערך ממורכז ואנחנו בוחרים את ה-*pivot* להיות המקום הימני ביותר, אזי $m = n$ בכל פעם ולכן מקבלים: $T(n) = T(n - 1) + \Theta(n) = \Theta(n^2)$ (אפשר גם להוכיח ישירות באמצעות עץ רקורסיה למשל).
- אם $m = \frac{n}{2}$ תמיד (הבחירה ה"מושלמת" של ה-*pivot*) אזי אנו מקבלים את עץ הרקורסיה של *MERGE SORT*, שהוא בעומק $\log(n)$ ובכל שלב יש n פעולות, ולכן $T(n) = \Theta(n \log(n))$ (ניתן גם היה לטעון ישירות ממשפט המאסטר).
- אם $m = \frac{9}{10}n$ (בחירה לא טובה כל כך של ה-*pivot*), אזי בעץ שלנו יש התפלגויות ל- $\frac{1}{10}n$ ו- $\frac{9}{10}n$ בכל שלב, אבל עומק הענף המקסימלי הוא עדיין רק $\log_{\frac{9}{10}}(n)$, וזה בסה"כ הבדל של קבוע מ- $\log(n)$. כיוון שהעבודה המקסימלית בכל ענף היא עדיין n (ויש ענפים שנגמרים באמצע), אנחנו עדיין מקבלים $\Theta(n \log(n))$.

למעשה, נראה כאילו עבור "כמעט" כל הקלטים (למעט $m = n$) יוצאות לנו תוצאות לא רעות בכלל. אבל איך נדע מה קורה במקרה ה"ממוצע", או "בדרך כלל"? נכניס הסתברות לכל הסיפור.

טענה 8א': עבור כל קלט, $T(n) = \mathcal{O}(n^2)$

הוכחה: נתבונן על הקלט הגרוע ביותר, ובו:

$$T(n) = \max_{1 \leq q \leq n-1} \{T(q) + T(n - q - 1)\} + \Theta(n)$$

ונשתמש בהגדרת Θ ונקבל:

$$T(n) \leq \max_{1 \leq q \leq n-1} \{T(q) + T(n-q-1)\} + dn$$

כעת נוכיח באינדוקציה ש- $T(n) = O(n^2)$. בסיס האינדוקציה: $T(1) < c$ מתקיים.

נניח כי $\forall k < n, T(k) \leq ck^2$ וכעת נרצה להראות ש- $T(n) \leq cn^2$.

$$T(n) \leq \max_{1 \leq q \leq n-1} \{cq^2 + c(n-q-1)^2\} + dn = c \max_{1 \leq q \leq n-1} \{q^2 + (n-q-1)^2\} + dn$$

הביטוי $q^2 + (n-q-1)^2$ מקבל מקסימום על הטווח $1 \leq q \leq n-1$ באחד הקצוות, כיוון שזוהי פרבולה "מחייכת". לכן יש לנו:

$$\max_{1 \leq q \leq n-1} \{cq^2 + c(n-q-1)^2\} = 1^2 + (n-2)^2 = n^2 - 2(n-2) + 5 < n^2 - n \quad \forall n \geq 9$$

מכאן אנחנו מקבלים ישירות:

$$T(n) \leq c(n^2 - n) + dn \leq cn^2 \quad \forall c > d$$

■

טענה 8ב': (ללא הוכחה) עבור כל קלט, $T(n) = O(n \log(n))$.

מהטענה נובע שמקרי הקצה שהסתכלנו עליהם קודם אכן היו מייצגים, עבור הקלט הטוב ביותר והקלט הגרוע ביותר של האלגוריתם.

הגדרה: לקלט שעבורו סיבוכיות זמן הריצה היא המקסימלית קוראים הקלט הגרוע ביותר (worst case).

הגדרה: לקלט שעבורו סיבוכיות זמן הריצה היא המינימלית קוראים הקלט הטוב ביותר (best case).

הגדרה: לקלט שעבורו סיבוכיות זמן הריצה היא הממוצעת על פני כל הקלטים קוראים הקלט הממוצע (average case).

הערה: ייתכן ששלושת המקרים הם אותו הקלט, או שיש יותר מקלט אחד עבור כל מקרה.

בסה"כ עבור בעיית המיון יש לנו $n!$ קלטים אפשריים (לא נאפשר כפילויות). לכן, מעניינת אותנו הסיבוכיות הממוצעת עבור כל הפרמוטציות (תמורות) האפשריות של הקלטים:

$$\langle T(n) \rangle = \frac{\sum_{\sigma} T(n) \text{ for input } \sigma}{n!}$$

טענה 9: (ללא הוכחה) $\langle T(n) \rangle = O(n \log(n))$

מהטענה ברור שמקרה הקצה הגרוע ביותר ($O(n^2)$) הוא במקרה זניח יחסית לאחרים, ובדרך כלל הקלט יתנהג כמו עבור הקלט הטוב ביותר ($O(n \log(n))$) עד כדי קבוע. בכל זאת, כדי לומר שהאלגוריתם עובד עבורנו ב- $O(n \log(n))$ הנחנו כאן הנחה גדולה – שהקלטים נבחרים באקראי. אולי מישהו משתמש ב- *QUICK SORT* כדי למיין רק מערכים ממוינים?

אנו רוצים לשנות את האלגוריתם כך שיהיה תמיד $O(n \log(n))$ ללא תלות בקלט. אפשרות אחת היא לעשות רנדומיזציה של הקלט בהתחלה, אך אפשרות טובה יותר היא פשוט לבחור את ה- *pivot* באקראי במקום לבחור את האיבר הימני ביותר במערך. כלומר, במקום להניח שהקלטים מתפלגים אקראית נניח שה- *pivot* נבחר באקראי ואז נוכל לנתח את הסיבוכיות של האלגוריתם באופן מדויק.

מונחים בסיסיים בהסתברות

הגדרה: אוסף המאורעות שיכולים להתרחש נקראים מרחב המדגם Ω .

למשל, הטלות מטבע יוצרות את מרחב המדגם $\Omega_1 = \{heads, tails\}$, והטלות קובייה יוצרות את מרחב המדגם $\Omega_2 = \{1, 2, 3, 4, 5, 6\}$. שתי הטלות מטבע יוצרות את מרחב המדגם $\Omega_3 = \{HH, HT, TH, TT\}$.

הגדרה: על מרחב המדגם נגדיר פונקציית הסתברות $P: \Omega \rightarrow \mathbb{R}$ המקיימת:

1. $\forall A \in \Omega, P(A) \geq 0$
2. $P(\Omega) = 1$
3. $\forall A, B \in \Omega: A \cap B = \emptyset, P(A \cup B) = P(A) + P(B)$

פונקציית ההסתברות מייצגת עד כמה סביר מאורע מסוים. למשל, במקרה של קובייה הוגנת פונקציית ההסתברות מחזירה $\frac{1}{6}$ לכל אפשרות הטלה. ייתכן כמובן גם מצב לא הוגן, למשל מטבע לא הוגן:
 $P(H) = p, P(T) = 1 - p$

הגדרה: משתנה מקרי היא פונקציה שהערך שלה הוא הסתברותי והיא מוגדרת על מרחב המדגם, אולם רק עבור המאורעות האלמנטריים במרחב המדגם.

למשל, סכום ההטלות של שתי קוביות זהו משתנה מקרי על מרחב המדגם של הטלות שתי קוביות (הקבוצה $\{1 \dots 6\} \times \{1 \dots 6\}$).

הגדרה: תוחלת של משתנה מקרי היא ממוצע משוקלל של הערכים השונים וההסתברויות שלהם, כלומר עבור משתנה מקרי X , נסמן את התוחלת שלו ב:

$$E(X) = \sum_{x \in \Omega} P(X = x) x$$

נחשב למשל את התוחלת של המשתנה המקרי X המוגדר להיות ערך ההטלה של קובייה. יש לנו:

$$E(X) = \sum_{x=1}^6 \frac{1}{6} x = 3.5$$

טענה 10: (אי-שוויון מרקוב) יהי X משתנה מקרי אי-שלילי ו- $a > 0$, אזי: $P(x \geq a) \leq \frac{E(X)}{a}$.

הוכחה:

$$\begin{aligned} E(X) &= \sum_{x=0}^{\infty} P(X = x) x = \sum_{x=0}^a P(X = x) x + \sum_{x=a}^{\infty} P(X = x) x \geq \sum_{x=a}^{\infty} P(X = x) x \geq \sum_{x=a}^{\infty} P(X = x) a \\ &= a \sum_{x=a}^{\infty} P(X = x) = a P(x \geq a) \Rightarrow P(x \geq a) \leq \frac{E(X)}{a} \end{aligned}$$

■

הערה: במקרים רבים, אי-שוויון מרקוב מספק חסם מאוד לא הדוק על ההסתברות, ולכן יש להשתמש בו בזהירות כדי לא לאבד אינפורמציה.

הגדרה: חיתוך המאורעות A, B הוא המאורע שבו שניהם מתרחשים, ומסמנים: $P(A \cap B)$.

למשל, המאורע שבו הטלת קוביה היא זוגית וגם לכל היותר 3 הוא בעל הסתברות $\frac{1}{6}$ (ומכיל את המאורע האלמנטרי 2).

הגדרה: איחוד המאורעות A, B הוא המאורע שבו אחד מהם או שניהם מתרחשים, ומתקיים:

$$P(A \cup B) = P(A) + P(B) - P(A \cap B)$$

הגדרה: הסתברות מותנית היא ההסתברות של מאורע מסוים בהינתן שמאורע אחר התרחש, ומתקיים:

$$P(B|A) = \frac{P(A \cap B)}{P(A)}$$

למשל, ההסתברות שהטלת הקובייה הייתה 3 בהינתן שהטלת הקובייה הייתה אי-זוגית היא $\frac{1}{3}$ לפי האינטואיציה,

$$\text{וגם לפי הנוסחה: } P(3|odd) = \frac{P(3 \cap odd)}{P(odd)} = \frac{P(3)}{P(odd)} = \frac{1}{3}$$

הגדרה: מאורעות בלתי תלויים A, B מקיימים $P(B|A) = P(B)$.

במקרה זה ניתן לקבל:

$$P(B) = P(B|A) = \frac{P(A \cap B)}{P(A)}$$
$$P(A \cap B) = P(A)P(B)$$

לדוגמה, שתי הטלות מטבע הן מאורעות בלתי תלויים.

הגדרה: המשלים של מאורע מסוים הוא אוסף כל המאורעות שבהם המאורע לא התרחש, ומתקיים:
 $P(A) + P(\text{not } A) = 1$

טענה 11: יהיו X, Y משתנים מקריים, אזי $E(X + Y) = E(X) + E(Y)$

הוכחה:

$$\begin{aligned} E(X + Y) &= \sum_i \sum_j (i + j) P(X = i \cap Y = j) = \sum_i \sum_j [i P(X = i \cap Y = j) + j P(X = i \cap Y = j)] \\ &= \sum_i \sum_j i P(X = i \cap Y = j) + \sum_i \sum_j j P(X = i \cap Y = j) \\ &= \sum_i i \sum_j P(X = i \cap Y = j) + \sum_j j \sum_i P(X = i \cap Y = j) \\ &= \sum_i i P(X = i) + \sum_j j P(Y = j) = E(X) + E(Y) \end{aligned}$$

■

Quick Sort רנדומי

בחזרה לבחירת ה-pivot בצורה אקראית: אנו רוצים להגדיר משתנה מקרי שהוא זמן הריצה של האלגוריתם, ולחשב את תוחלת זמן הריצה.

הערה: אפשר לחשב את הממוצע של זמני הריצה לכל הקלטות בהנחה שבוחרים קלט באקראי, או שאפשר לקבע קלט ולבחור את ה-pivot באקראי ולהתבונן בתוחלת זמני הריצה. שני הדברים אינם שקולים!

טענה 12: תוחלת זמן הריצה של RANDOMIZED QUICK SORT על קלט קבוע היא $\Theta(n \log(n))$.

הוכחה: נרצה לכתוב את נוסחת הרקורסיה עבור $E(T(n))$, תוחלת זמן הריצה:

$$E(T(n)) \neq E(T(m-1)) + E(T(n-m)) + \Theta(n)$$

זה לא נכון, כיוון שאנחנו לא יודעים מהו m - הוא נקבע באקראי. לכן אנחנו צריכים להיזהר יותר ולהסתכל על הממוצע של כל האפשרויות לבחור את m , שההסתברות לכל אחת מהן היא אחידה ושווה ל- $\frac{1}{n}$:

$$E(T(n)) = \sum_{m=1}^n \frac{1}{n} [E(T(m-1)) + E(T(n-m))] + \Theta(n)$$

ונשים לב שכאן $E(T(m-1))$ למשל הוא לא ממוצע על אותו ערך שחוזר על עצמו כמה פעמים, כיוון שבכל קריאה רקורסיבית תהיה לנו עבודה נוספת שתלווה בהגרלה חדשה של pivot. נפשט את הביטוי:

$$\begin{aligned} E(T(n)) &= \sum_{m=1}^n \frac{1}{n} [E(T(m-1)) + E(T(n-m))] + \Theta(n) \\ &= \sum_{m=0}^{n-1} E(T(m)) + \sum_{m=n-1}^0 (E(T(n-m)) + \Theta(n)) = \sum_{m=0}^{n-1} \frac{2}{n} E(T(m)) + \Theta(n) \end{aligned}$$

ועכשיו מהגדרת Θ מקבלים:

$$\begin{aligned} \sum_{m=0}^{n-1} \frac{2}{n} E(T(m)) + c_2 n &\leq E(T(n)) \leq \sum_{m=0}^{n-1} \frac{2}{n} E(T(m)) + c_1 n \\ E(T(0)) = \Theta(1) &\Rightarrow c_2 \leq E(T(0)) \leq c_1 \end{aligned}$$

מה שקיבלנו הוא אמנם פשוט בהרבה, אבל זו עדיין משוואת רקורסיה עם היסטוריה מלאה - התוחלת של $T(n)$ תלויה בכל ה- $T(m)$ ים הקטנים יותר, ולא רק באחד או שניים מהם.

למען הפשטות, נגדיר פונקציה חדשה $U_c(n)$ שתקיים את אי השוויון, כלומר:

$$\begin{aligned} U_c(n) &= \sum_{m=0}^{n-1} \frac{2}{n} U_c(m) + cn \\ U_c(0) &= c \end{aligned}$$

טענה 12א': $U_{c_1}(n) \leq E(T(n)) \leq U_{c_2}(n)$

הוכחה: באינדוקציה פשוטה, ר' למשל פתרון תרגיל 4

כעת נציב בביטוי שיש לנו n ו- $n+1$ ונקבל את שני הביטויים:

$$nU_c(n) = 2 \sum_{m=0}^{n-1} U_c(m) + cn^2$$

$$(n+1)U_c(n+1) = 2 \sum_{m=0}^n U_c(m) + c(n+1)^2$$

כעת נחסר את המשוואה הראשונה מהשנייה:

$$(n+1)U_c(n+1) - nU_c(n) = 2U_c(n) + c(n+1)^2 - cn^2$$

$$(n+1)U_c(n+1) = (n+2)U_c(n) + 2cn + c$$

$$U_c(n+1) = \frac{n+2}{n+1}U_c(n) + \frac{2cn+c}{n+1}$$

עכשיו קיבלנו משוואת רקורסיה עם היסטוריה של שלב אחד אחורה בלבד – ולא היסטוריה מלאה. כדי לפתור אותה, נשים לב לעובדה הפשוטה ש- $c \leq \frac{2cn+c}{n+1} \leq 2c$ וקיבלנו שני אי שוויונות בשני הכיוונים הרצויים. נתחיל מחסם מלמעלה $(\mathcal{O})$:

$$U_c(n+1) \leq \frac{n+2}{n+1}U_c(n) + 2c \leq \frac{n+2}{n+1} \left(\frac{n+1}{n}U_c(n-1) + 2c \right) + 2c$$

$$= \frac{n+2}{n+1} \left(\frac{n+1}{n} \right) U_c(n-1) + \frac{n+2}{n+1} 2c + \frac{n+2}{n+1} 2c$$

$$\leq \frac{n+2}{n+1} \left(\frac{n+1}{n} \right) \left(\frac{n}{n-1}U_c(n-2) + 2c \right) + \frac{n+2}{n+1} 2c + \frac{n+2}{n+1} 2c \leq \dots$$

$$\leq \frac{n+2}{n+2} 2c + \frac{n+2}{n+1} 2c + \dots + \frac{n+2}{1} 2c + \frac{n+2}{1} U_c(0)$$

עכשיו ניזכר ש- $U_c(0) \leq 2c$ ולכן גם אותו אפשר להחליף ב- $2c$ ולקבל לבסוף:

$$U_c(n+1) \leq (n+2)2c \left(\sum_{j=1}^{n+2} \frac{1}{j} + 1 \right)$$

אז בעצם קיבלנו את הטור ההרמוני: $H(n) = \sum_{j=1}^n \frac{1}{j}$ ועל מנת שנוכל לסיים אנחנו צריכים לחשב אותו.

טענה 12ב': $H(n) = \Theta(\log(n))$

הוכחה: ראשית נניח $n = 2^k$. אזי נוכל לחלק את הטור לבלוקים כאשר הגודל של בלוק k הוא 2^k איברים, ונסמן את סכומם S_k . אזי:

$$S_k = \frac{1}{2^k} + \frac{1}{2^k+1} + \dots + \frac{1}{2^{k+1}-1}$$

$$S_k \geq 2^k \left(\frac{1}{2^{k+1}-1} \right) \geq \frac{1}{2}$$

$$s_k \leq 2^k \left(\frac{1}{2^k} \right) \leq 1$$

מאחר שכל הטור הוא פשוט $H(n) = \sum_{j=0}^{k-1} s_j$ אז מקבלים:

$$\begin{aligned} H(n) &\geq \frac{k}{2} = \frac{\log(n+1)}{2} \\ H(n) &\leq k \leq \log(n+1) \end{aligned}$$

נבחר k כך ש- $2^{k-1} - 1 < n \leq 2^k - 1$. כעת נפתח:

$$\begin{aligned} 2^{k-1} &< n+1 \leq 2^k \\ k-1 &< \log(n+1) \leq k \end{aligned}$$

כעת ממונוטוניות $H(n)$ ברור ש- $H(2^{k-1} - 1) < H(n) \leq H(2^k - 1)$ ולכן $\frac{k-1}{2} \leq H(n) \leq k$. מהצבה בחסם שקיבלנו כרגע על k יש לנו:

$$\begin{aligned} \frac{\log(n+1) - 1}{2} &< H(n) \leq \log(n+1) + 1 \\ \frac{\log\left(\frac{n+1}{2}\right)}{2} &< H(n) \leq \log(2(n+1)) \end{aligned}$$

וכעת יש לנו בשני האגפים ביטויים שהם $\Theta(\log(n))$ ומייד ניתן להסיק $H(n) = \Theta(\log(n))$. הוכחנו את טענה 12. ■

קודם היה לנו ביטוי על $U_c(n+1)$ ונחזור אליו ע"י הצבת $H(n+2) = \Theta(\log(n+2))$:

$$U_c(n+1) \leq (n+2)2c(H(n+2) + 1) = (n+2)2c\Theta(\log(n+2)) + (n+2)2c = \mathcal{O}(n \log(n))$$

באופן סימטרי לחלוטין אפשר לקבל גם $U_c(n+1) = \Omega(n \log(n))$ ולכן לבסוף $U_c(n) = \Theta(n \log(n))$. ניזכר שבמקור חסמנו את $E(T(n))$ של *RANDOMIZED QUICK SORT* משני הצדדים באמצעות $U_{c_1}(n)$ ו- $U_{c_2}(n)$, ולכן בעצם יש לנו:

$$\begin{aligned} \Theta(n \log(n)) &\leq E(T(n)) \leq \Theta(n \log(n)) \\ E(T(n)) &= \Theta(n \log(n)) \end{aligned}$$

■

אם מעניין אותנו לדעת כמה סביר הוא המקרה שבו זמן הריצה יהיה משמעותית יותר גדול מהתוחלת אפשר למשל להשתמש באי-שוויון מרקוב (טענה 10). לדוגמה, נוכל לומר שב- 99% מהמקרים זמן הריצה חסום ע"י $100cn \log(n)$.

חסם תחתון על אלגוריתמי מיון במודל ההשוואות

כמעט בכל האלגוריתמים עד עכשיו (למעט *COUNTING SORT*) השתמשנו במיון במודל ההשוואות – מיון שלא מתעניין בתוכן של הקלט אלא רק ביחסים בין איברי הקלט, תוך ביצוע השוואה בין האיברים השונים. נרצה לטעון טענה לגבי כל אלגוריתם מיון באשר הוא – כל עוד הוא משתמש במודל ההשוואות – כדי לחסום את זמן הריצה

שלו. זה יעודד אותנו במקצת כי המשמעות היא שאין אלגוריתם מיון אסימפטוטית טוב יותר מ-
RANDOMIZED QUICK SORT למשל.

טענה 13: אלגוריתם מיון דטרמיניסטי במודל ההשוואות יבצע $\Omega(n \log(n))$ צעדים במקרה הגרוע ביותר.

הערה: אנו מוכיחים את הטענה לגבי אלגוריתם מיון דטרמיניסטי, אולם ניתן להרחיב אותו בלא קושי מיוחד גם לאלגוריתם המשתמש באקראיות.

הוכחה: נייחס לכל אלגוריתם מיון במודל ההשוואות מבנה הנקרא עץ הכרעה. האלגוריתם בשלב כלשהו שואל שאלה (השוואה), למשל האם $x_1 < x_2$, והתשובה היא כן או לא. בכל תשובה נוצר נתיב ריצה מסוים שנבחר, ולכן בעצם נוצר עץ (בינארי) שבכל קודקוד שלו יש שאלה ותוצאת המיון הסופית היא עלה כלשהו בעץ. לפיכך, סיבוכיות האלגוריתם היא פשוט עומק העץ – המסלול הארוך ביותר – ונסמנו D .

כדי שאלגוריתם המיון יהיה נכון, הוא חייב להיות נכון לכל קלט אפשרי. במקרה שלנו (נתעלם מכפילויות) יש לנו $n!$ קלטים אפשריים בגודל n , ולכן גם $n!$ עלים בעץ ההכרעה. לצורך ההוכחה נסתפק בכך שבעץ לפחות $n!$ עלים.

טענה 13א': לעץ הכרעה של אלגוריתם מיון במודל ההשוואות חייבים להיות בדיוק $n!$ עלים.

הוכחה: קיימות $n!$ פרמוטציות (תמורות) של קלט בגודל n (נזכור שאנו מניחים שכל איברי הקלט שונים זה מזה). האלגוריתם דטרמיניסטי ומשייך לכל קלט מסלול בעץ. בנוסף, כיוון שכל התמורות שונות זו מזו, גם העלים בעץ שונים זה מזה – כלומר אין שני קלטים המובילים לאותו העלה. לפיכך, ההתאמה בין קלט מסוים לעלה בעץ (כלומר המסלול) היא העתקה חח"ע ועל, ולכן יש בדיוק $n!$ עלים בעץ ההכרעה. ■

לעץ בינארי בגובה D יש לכל היותר 2^D עלים, אם הוא עץ מלא. לפיכך, נוכל לחסום את מספר העלים:

$$2^D \geq \#leaves \geq n!$$

נרצה לחלץ מכאן את D באמצעות הוצאת $\log$ ונקבל:

$$running\ time \geq D \geq \log(n!)$$

טענה 13ב': $\log(n!) = \Theta(n \log(n))$

הוכחה: ראשית, $\log(n!) = \log(n) + \log(n-1) + \dots + \log(2) + \log(1)$ וכאן כמובן כל אחד מהאיברים קטן או שווה ל- $\log(n)$ ולכן זה $\mathcal{O}(n \log(n))$ אבל אנחנו גם יודעים שיש לנו חצי מהאיברים שגדולים מ- $\log\left(\frac{n}{2}\right)$,

$$\log(n!) \geq \frac{n}{2} \log\left(\frac{n}{2}\right) = \frac{n}{2} \log(n) - \frac{n}{2} \log(2)$$

כלומר $\log(n!) = \Omega(n \log(n))$ וז"א $\log(n!) \geq \frac{n}{2} \log\left(\frac{n}{2}\right) = \frac{n}{2} \log(n) - \frac{n}{2} \log(2)$ ולכן גם $\log(n!) = \Theta(n \log(n))$. ■

מהטענה האחרונה נובע ש- $D = \Omega(n \log(n))$ ולכן קיבלנו את החסם התחתון על כל אלגוריתם מיון במודל ההשוואות כפי שרצינו. ■

הוכחה אלטרנטיבית של טענה 13: נשתמש בטענת עזר שתהפוך את החיים לקלים מאוד.

טענה 13ג': בכל עץ בעל m עלים מתקיים $\frac{1}{m} \sum_{l \in leaves(T)} d(l) \geq \log(m)$ כאשר $d(l)$ עומק העלה (מרחקו מהשורש).

הוכחה: אינדוקציה על מספר העלים. עבור $m = 1$ הטענה מתקיימת. נניח עבור כל $m \leq n$ ונוכיח עבור $m + 1$. ניקח עץ עם $m + 1$ עלים, ונניח שלשורש העץ יש שני בנים, תתי עצים T_1, T_2 , עם m_1, m_2 עלים בהתאמה. אזי לפי הנחת האינדוקציה:

$$\begin{aligned}\frac{1}{m_1} \sum_{l \in \text{leaves}(T_1)} d_{T_1}(l) &\geq \log(m_1) \\ \frac{1}{m_2} \sum_{l \in \text{leaves}(T_2)} d_{T_2}(l) &\geq \log(m_2)\end{aligned}$$

כמו כן כמובן ברור שעומק של עלה בכל אחד מתתי העצים קטן ב-1 מעומק העלה בעץ המקורי. לכן מתקיים:

$$\begin{aligned}\frac{1}{m} \sum_{l \in \text{leaves}(T)} d_T(l) &= \frac{1}{m} \left[\sum_{l \in \text{leaves}(T_1)} (d_{T_1}(l) + 1) + \sum_{l \in \text{leaves}(T_2)} (d_{T_2}(l) + 1) \right] \\ &= \frac{1}{m} \left[\sum_{l \in \text{leaves}(T_1)} d_{T_1}(l) + m_1 + \sum_{l \in \text{leaves}(T_2)} d_{T_2}(l) + m_2 \right] \\ &= 1 + \frac{1}{m} \left[\sum_{l \in \text{leaves}(T_1)} d_{T_1}(l) + \sum_{l \in \text{leaves}(T_2)} d_{T_2}(l) \right]\end{aligned}$$

וכעת מהנחת האינדוקציה ניתן לקבל:

$$\dots = 1 + \frac{1}{m} (m_1 \log(m_1) + m_2 \log(m_2)) \geq \log(m)$$

וכאן אי-השוויון האחרון מתקיים כיוון ש- $m_1 + m_2 = m$ ולכן אם נתבונן בפונקציה:

$$h(x) = 1 + \frac{1}{m} (x \log(x) + (m - x) \log(m - x)) - \log(m)$$

כאן המקסימום הוא ב- $x = \frac{m}{2}$ וכאן $h\left(\frac{m}{2}\right) = 0$. לכן סיימנו את האינדוקציה. ■

נחזור להוכחה האלטרנטיבית של טענה 13. כיוון שבעץ הכרעה של אלגוריתם מיון על קלט בגודל n יש בדיוק $n!$ עלים (טענה 13א'), סיבוכיות זמן הריצה במקרה הממוצע היא בדיוק העומק הממוצע של העלים, ולפי טענה 13ג' זה לפחות $\log(n!)$, כלומר $\Omega(n \log(n))$. ■

פרק 2 – עצים בינאריים ועצי חיפוש בינאריים

עץ בינארי

הגדרה: עץ הוא גרף קשיר ללא מעגלים.

הגדרה: עץ בינארי הוא עץ בעל שורש יחיד שבו לכל קודקוד בין 0 ל-2 ילדים (בנים).

הגדרה: שורש העץ הוא הקודקוד שאין לו הורה.

הגדרה: עלה הוא קודקוד ללא ילדים.

הגדרה: קודקוד פנימי הוא קודקוד שיש לו ילד אחד או שני ילדים

הגדרה: גובה העץ או עומק העץ – אורך המסלול הארוך ביותר מהשורש לעלה בעץ – לרוב יסומן ב- D .

הגדרה: עומק של קודקוד x הוא אורך המסלול מהשורש ל- x ומסומן $d(i)$.

הגדרה: גובה של קודקוד x הוא אורך המסלול הארוך ביותר מ- x לעלה כלשהו בעץ ומסומן $h(i)$.

הגדרה: עץ מלא הוא עץ שבו לכל קודקוד 0 או 2 בנים (אין קודקוד עם בן אחד).

הגדרה: עץ שלם הוא עץ בינארי שכל הרמות שלו מלאות וכל העלים שלו נמצאים באותו הגובה

בעץ בינארי שלם יש 2^D עלים, ו- $2^D - 1 = 2^0 + 2^1 + \dots + 2^{D-1}$ קודקודים פנימיים.

הגדרה: עץ בינארי כמעט שלם הוא עץ בינארי שלם, למעט הרמה התחתונה שבה חלק מהעלים נמחק. כל העלים משמאל עד לנקודה מסוימת קיימים (כמו בעץ שלם), ומאותה נקודה ימינה אין עלים ברמה התחתונה.

בעץ בינארי כמעט שלם מתקיים $\#leaves \leq 2^D$ (כאשר שוויון מתקיים אם העץ שלם).

טענה 14: בעץ בינארי כמעט שלם מתקיים $D \leq \log(2n)$ כאשר D עומק העץ, n מס' הקודקודים בעץ.

הוכחה: בעץ בינארי כמעט שלם בגובה h יש לכל היותר $2^{h+1} - 1$ איברים בסה"כ (אם הוא שלם – זה נובע מהאבחנות לגבי מס' העלים ומס' הקודקודים הפנימיים בעץ בינארי שלם), ולכל הפחות $2^h - 1 + 1 = 2^h$ איברים (אם ברמה התחתונה יש עלה אחד בלבד). לפיכך:

$$2^h \leq n \leq 2^{h+1} - 1 < 2^{h+1}$$

$$D \leq \log(n) < D + 1$$

■

תור קדימויות, ערימה

כעת נוכל להתחיל לדבר על מבני נתונים אמיתיים. כדי לעשות זאת, אנו משתמשים במושג של ADT = Abstract Data Type, המספק הפרדה בין הפעולות שאנו רוצים לעשות על מבני הנתונים לבין האופן שבו מבנה הנתונים ממומש (למעשה, זהו ההבדל בין interface ל- implementation). ראינו כבר מספר דוגמאות ל-ADT, למשל תור (queue) ומחסנית (stack), שאת שניהם אפשר לממש באמצעות רשימה מקושרת או באמצעות מערך.

תור קדימויות (Priority Queue) מספק את הפעולות הבאות עבור איברים בעלי קדימות (מספרית) כלשהי:

- $MAX(S)$ – מחזיר את האיבר עם הקדימות הגבוהה ביותר
- $EXTRACT MAX(S)$ – מוציא את האיבר עם הקדימות הגבוהה ביותר מהתור ומחזיר אותו
- $INSERT(S, x)$ – מכניס לתור את הרשומה x
- $INCREASE KEY(S, x, key)$ – משנה את הקדימות של הרשומה x להיות key

את תור הקדימויות ניתן לממש באמצעות מבנה נתונים שנקרא ערימה (Heap).

הגדרה: תכונת הערימה – אם A הוא תור הקדימויות, אזי $A[parent(x)] \geq A[x]$.

הגדרה: ערימה היא עץ בינארי כמעט שלם, ולכל קודקוד הקדימות שלו והערכים בקודקודי העץ מקיימים את תכונת הערימה.

הערה: כל ההגדרות לעיל מגדירות ערימת מקסימום (Max Heap), אך ניתן לדבר גם על ערימת מינימום עם ההגדרות המתאימות עבורה (Min Heap).

הערה: אין שום הבטחה לסדר בתוך הערימה למעט תכונת הערימה. למשל, לא מתקיימת תכונת עץ חיפוש בינארי (שנראה בהמשך).

כדי לממש עץ בינארי כמעט שלם, אין צורך במצביעים לילדים ולהורה – ניתן לממש אותו באמצעות מערך, כאשר כל רמה k מיוצגת על ידי רצף איברים בגודל 2^k בתוך המערך. אנחנו צריכים כמובן פעולות נביגציה בתוך המערך – ימינה, שמאלה ולכיוון ההורה:

$$\begin{aligned} \text{left}(x) &\leftarrow 2x \\ \text{right}(x) &\leftarrow 2x + 1 \\ \text{parent}(x) &\leftarrow \left\lfloor \frac{x}{2} \right\rfloor \end{aligned}$$

מספר האיברים בערימה כמובן משתנה גם, ונעקוב אחריו בכל עת:

$$\text{HEAPSIZE}(A) \leftarrow n$$

כעת נוכל לעבור למימוש של הפעולות השונות על הערימה. נתחיל ב- MAX - זה כמובן קל, כי מובטח לנו בגלל תכונת הערימה שהאיבר הראשון במערך הוא המקסימלי (שורש העץ):

MAX(S)
return $A[1]$

כעת נרצה לממש את EXTRACT MAX . נשים לב לרעיון שגוי – אם פשוט נוציא את האיבר הראשון ונזיז את כל שאר האיברים שמאלה, נקבל ערימה לא חוקית! דוגמה: אם הערימה הייתה $\{100, 2, 99\}$ אזי היא ערימת מקסימום חוקית, אבל לאחר ההזזה שמאלה מתקבלת הערימה $\{2, 99\}$ שהיא לא ערימה חוקית. לכן אנחנו צריכים משהו מתוחכם יותר.

נחליף את האיבר הראשון באיבר האחרון (כי זה כרוך בהכי פחות הזזות) ואז "נאזן" את הערימה – בכל פעם נחליף את השורש עם הבן הגדול ביותר שלו, עד שנגיע לסוף או שהוא יהיה גדול משני הילדים שלנו לצורך ביצוע האיזון הזה, נשתמש בשיטת עזר – MAX HEAPIFY :

MAX HEAPIFY(A, i)
 $l \leftarrow \text{left}(i), r \leftarrow \text{right}(i)$
if $l \leq \text{HEAPSIZE}(A)$
 if $r > \text{HEAPSIZE}(A)$ *then* $r \leftarrow \text{HEAPSIZE}(A)$
 if $A[l] > A[i]$ *then* $\text{largest} \leftarrow l$
 else $\text{largest} \leftarrow i$
 if $A[r] > A[\text{largest}]$ *then* $\text{largest} \leftarrow r$
 if $\text{largest} \neq i$
 $\text{swap}(A[i], A[\text{largest}])$
 $\text{MAX HEAPIFY}(A, \text{largest})$

הסיבוכיות של $MAX\ HEAPIFY$ היא $\mathcal{O}(\log(n))$ כי אנחנו בכל פעם יורדים ברמה אחת, וגובה העץ הוא לכל היותר $\log(2n)$ (טענה 14). כעת אפשר לכתוב את $EXTRACT\ MAX$:

$EXTRACT\ MAX(A)$

$max \leftarrow A[1]$

$A[1] \leftarrow A[HEAPSIZE(A)]$

$decrement\ HEAPSIZE(A)$

$MAX\ HEAPIFY(A, 1)$

$return\ max$

הסיבוכיות הכוללת של $EXTRACT\ MAX$ היא גם כן $\mathcal{O}(\log(n))$ כיוון שאנו מבצעים מספר קבוע של פעולות ואז קוראים ל- $MAX\ HEAPIFY$.

הערה: הוכחת הנכונות של $EXTRACT\ MAX$ תלויה בהוכחת הנכונות של $MAX\ HEAPIFY$. אם מניחים ש- $MAX\ HEAPIFY$ נכון (טענה 15) וכן מניחים את טענה 15ג', אזי ברור שמה שעושים ב- $EXTRACT\ MAX$ גורם למצב שבו שני הבנים של השורש הם עדיין ערימות מקסימום תקינות, וההפרה היחידה יכולה להיות בשורש. זה בדיוק התנאי של טענה 15ג', ולכן לאחר ביצוע $MAX\ HEAPIFY$ עדיין יש לנו ערימה חוקית.

כעת נרצה לממש את $INCREASE\ KEY$. יש לנו בעיה דומה – איבר בערימה שהמפתח שלו משתנה גורם למצב שהוא עשוי להפר את תכונת הערימה (למשל, כי הוא גדול יותר מההורה שלו). לכן, אנחנו צריכים להעלות אותו למעלה עד המקום הנכון שלו. נשים לב שבדרך אנחנו שומרים על מבנה הערימה, כי אם החלפנו הורה בבן שגדול יותר ממנו, אז הבן הזה כמובן גדול יותר מהבן השני של ההורה, בגלל תכונת הערימה. נכתוב את זה:

$INCREASE\ KEY(A, i, key)$

$if\ key < A[i]\ then\ raise\ error$

$A[i] \leftarrow key$

$while\ i > 1\ and\ A[parent(i)] < A[i]$

$swap(A[i], A[parent(i)])$

$i \leftarrow parent(i)$

גם כאן אנחנו מעלים את הקודקוד לכל היותר לכל גובה העץ, כלומר הסיבוכיות היא $\mathcal{O}(\log(n))$.

נותר לנו לטפל בהכנסה לערימה. כדי לעשות זאת, נכניס את האיבר לסוף המערך (מה שמקל עלינו לבצע את ההכנסה), אבל ניתן לו קדימות של $-\infty$ במקום הקדימות האמיתית שלו. לאחר מכן, נבצע $INCREASE\ KEY$ כדי להעלות אותו למקום הנכון:

$INSERT(A, key)$

$increment\ HEAPSIZE(A)$

$A[HEAPSIZE(A)] \leftarrow -\infty$

$INCREASE\ KEY(A, HEAPSIZE(A), key)$

גם כאן הסיבוכיות היא $\mathcal{O}(\log(n))$ כי למעשה אנחנו עושים מספר קבוע של פעולות ואז קוראים ל- $INCREASE\ KEY$.

נותר לנו רק לבנות את הערימה מרשימה קיימת של איברים. ניתן לעשות זאת במספר דרכים:

- להשתמש במיון – מערך ממיון (בסדר עולה) הוא ערימה חוקית, זמן ריצה: $\mathcal{O}(n \log(n))$.

- שימוש ב- $INSERT$ עבור כל איבר, n פעמים – סה"כ $O(n \log(n))$. לדוגמה, ההכנסה של $\{1, \dots, n\}$ גורמת לכך שבכל שלב אנו נאלצים לפעפע איבר חדש לשורש של הערימה, כי כל איבר שמוכנס גדול מכל האחרים. לכן, לאיבר ה- j נשלם $c \log(j)$ ובסה"כ אם ניקח למשל את ה- $\frac{n}{2}$ איברים האחרונים ברור שההכנסה היא $\Omega(n \log(n))$.
- מעבר על המערך הקיים מהסוף להתחלה ושימוש ב- $MAX\ HEAPIFY$ - על פניו גם זה $O(n \log(n))$ אך נוכיח שזה $O(n)$ (טענה 16).

BUILD HEAP(A)

$n \leftarrow HEAPSIZE(A) \leftarrow LENGTH(A)$

for $i \leftarrow n$ to 1

$MAX\ HEAPIFY(A, i)$

טענה 15: אלגוריתם $BUILD\ HEAP$ יוצר ערימת מקסימום תקינה.

הוכחה: ראשית ננסח כמה טענות עזר שנזדקק להן לצורך ההוכחה, ונוכיח אותן.

טענה 15א': כל קודקוד בעץ כמעט שלם הוא שורש של עץ כמעט שלם.

הוכחה: כל קודקוד הוא שורש של תת עץ. תת עץ עשוי להכיל חלק מהעלים ברמה התחתונה (או שהוא עצמו ברמה התחתונה, ואז הטענה נכונה באופן ריק). אם תת עץ זה לא מכיל אף איבר מהרמה התחתונה, אזי הוא נגמר ברמה שלפניו ואז לפי הגדרת עץ כמעט שלם, תת עץ זה הוא עץ שלם (שהוא גם בפרט עץ כמעט שלם). אם תת העץ מכיל חלק מהרמה התחתונה, אז בהכרח בחלק שהוא מכיל אין חורים (כי אחרת יש חורים בעץ המקורי). לכן זהו תת עץ כמעט שלם. ■

טענה 15ב': כל קודקוד בערימת מקסימום הוא שורש של ערימת מקסימום.

הוכחה: עבור כל קודקוד בערימה מתקיים ששני הבנים שלו קטנים ממנו (תכונת הערימה). לפי טענה 15א', כל קודקוד בעץ כמעט שלם הוא עץ כמעט שלם, ולכן כל קודקוד בערימה הוא שורש של עץ כמעט שלם שמקיים את תכונת הערימה, ולכן הוא שורש של ערימת מקסימום. ■

טענה 15ג': אם נפעיל את $MAX\ HEAPIFY$ על קודקוד x ששני בניו הם שורשים של ערימות מקסימום, אזי לאחר סיום ריצת האלגוריתם גם x יהיה שורש של ערימת מקסימום.

הוכחה: נוכיח באינדוקציה על גובה העץ h . עבור עץ בגובה 0, הטענה נכונה באופן ריק. נניח שהטענה מתקיימת עבור עצים בגובה h ונוכיח עבור $h + 1$. נתבונן בערימה בגובה $h + 1$ ונסמן את השורש שלה ב- r . לאחר הפעלת האלגוריתם, נבחין בין שני מקרים: אם r היה שורש תקין של ערימת מקסימום, אזי הוא המקסימלי מבין בניו ועצמו, ולכן לא עשינו שום דבר והוא עדיין שורש של ערימת מקסימום אם אחד הבנים של r היה המקסימלי (נסמנו c) אזי החלפנו בינו לבין r וביצענו קריאה רקורסיבית על r במקום החדש שלו. הבן השני של r לא השתנה, ולכן הוא עדיין שורש של ערימת מקסימום, הקריאה הרקורסיבית היא על תת עץ בגובה h ולפי הנחת האינדוקציה לאחר סיום ריצת האלגוריתם נקבל ערימת מקסימום תקינה. לכן כעת c הוא שורש של ערימת מקסימום חוקית כנדרש, וכמובן לא שינינו קודקודים שנמצאים מחוץ לתת העץ שהתחיל ב- r , וזאת מאחר שהאלגוריתם תמיד "יורד למטה". ■

כעת נוכיח את נכונות האלגוריתם. נטען טענה שקולה: בשלב ה- i של הריצה, הקודקודים $n, n - 1, \dots, n - i + 1$ הם שורשים של ערימות מקסימום. עבור $k = 0$ כל העלים הם ערימות מקסימום והטענה נכונה. נניח עבור k שהקודקודים $n, n - 1, \dots, n - k + 1$ הם שורשים של ערימות מקסימום. צעד האינדוקציה: אנו עובדים על

הקודקוד ב- $A[n-k]$, ונשים לב שהבנים שלו $right(n-k)$, $left(n-k)$ הם שורשים של ערימות מקסימום ע"פ הנחת האינדוקציה, וזאת מאחר ש- $right(n-k) > left(n-k) > n-k$. כלומר, לפי טענה 15ג', לאחר הפעלת $MAX\ HEAPIFY$ על קודקוד זה נקבל ערימת מקסימום תקינה, והשלמנו את האינדוקציה.

כעת נותר רק לוודא שלא קילקלנו אף קודקוד אחר במהלך הפעלת האלגוריתם. נבחין בין שני מקרים:

- אם קודקוד $n-k < j \leq n$ הוא בתת עץ של $n-k$ אזי לפי טענה 15ב' הוא שורש של ערימת מקסימום ולכן הכל נשאר תקין.
- אם הקודקוד כנ"ל הוא לא בתת עץ של $n-k$, אזי $MAX\ HEAPIFY$ כלל לא משפיע עליו (כיוון שהוא "יורד למטה" בלבד), ולכן הוא היה שורש של ערימת מקסימום (ע"פ הנחת האינדוקציה) ונשאר כזה.

■

טענה 16: זמן הריצה של $BUILD\ HEAP$ הוא $O(n)$.

הוכחה: בעץ בעל n קודקודים יש לכל היותר $\frac{n}{2^h}$ קודקודים המהווים שורש של תת עץ בגובה h . זאת מאחר שבתת עץ הנ"ל לפחות 2^h קודקודים, וכל n הקודקודים של העץ מחולקים על פני עצים אלה.

כעת אם נגדיר ch כמחיר התיקון של עץ בגובה h , יש לנו $\frac{n}{2^h}$ עצים כאלה, ויש $\log(n)$ גבהים שונים בעץ (טענה 14), נקבל:

$$\sum_{h=0}^{\log(n)} ch\left(\frac{n}{2^h}\right) \leq cn \sum_{h=0}^{\infty} \frac{h}{2^h} = 2cn = O(n)$$

כדי לטעון שזה נכון חסרה לנו רק הוודאות שאכן $\sum_{h=0}^{\infty} \frac{h}{2^h} = 2$. ניתן להראות זאת למשל כיוון שזו נגזרת של טור:

$$\sum_{h=0}^{\infty} hx^h = x \sum_{h=0}^{\infty} hx^{h-1} = x \sum_{h=0}^{\infty} (x^h)' = x \left(\sum_{h=0}^{\infty} x^h \right)' = x \left(\frac{1}{1-x} \right)' = \frac{x}{(1-x)^2} = 2$$

■

הערה: האלגוריתם עובר על כל קודקוד, ולכן הוא טריוויאלי $\Omega(n)$, כלומר מתקבל שהוא $\Theta(n)$.

הערה: ניתן לממש ערימה גם באמצעות עץ בינארי אמיתי (עם מצביעים), ללא מערך. הקושי הוא לדעת לאן להכניס איבר חדש למשל (או מאיפה לקחת איבר כשעושים $EXTRACT\ MAX$), שכן אנו לא יכולים לגשת ישירות לאיבר האחרון של העץ בקלות. ניתן להתגבר על הבעיה הזאת באמצעות שמירת מצביע לאיבר האחרון בעץ יחד עם מבנה הערימה, או (באופן חסכוני יותר) החזקת מספר האיברים ברמה התחתונה של העץ. מסתבר שהייצוג הבינארי של מספר זה משקף בדיוק את המסלול לעלה האחרון, כאשר 0 מייצג פנייה שמאלה ו-1 פנייה ימינה. נשים לב שכיוון שהעץ הוא עץ בינארי כמעט שלם (לא עץ חיפוש כמובן), הפעולות הן ב- $O(\log(n))$ כמו שהיו לנו קודם.

הערה: ניתן לממש ערימה גם באמצעות עץ חיפוש בינארי (BST), למשל עץ AVL (ר' בהמשך). במקרה זה, הערך המקסימלי בעץ נמצא בעומק $O(\log(n))$ ולכן נרצה לשמור ולתחזק אותו לאחר כל פעולה על הערימה, כלומר, לאחר שמבצעים למשל $EXTRACT\ MAX$, נזכור לחפש את $MAX(T)$ ולשמור אותו יחד עם הערימה (וכאשר מבצעים הכנסה, יש לבדוק האם האיבר שהוכנס גדול מהמקסימום הנוכחי). באופן כזה אנו שומרים על $O(1)$ בזמן הריצה של MAX על הערימה, וגם שאר הפעולות לא נפגעות (למשל $INCREASE\ KEY$ ניתן לממש

באמצעות מחיקה והוספה מחדש, ב- $O(\log(n))$. הבעיה המרכזית עם שני המודלים הנ"ל היא פעולת $BUILD\ HEAP$, שכן הדרך הטובה ביותר לבנות עץ ממערך לא ממורן היא לבצע רגנסה n פעמים, והעלות כאן היא $\Theta(n \log(n))$.

מיון באמצעות ערימה – Heap Sort

ניתן להשתמש בערימה על מנת לבצע מיון של מערך (in place). הרעיון הוא: נבנה ערימה מהמערך המקורי, וכעת נתחיל מהאיבר המקסימלי (השורש) ונחליף אותו עם האיבר שבסוף המערך. כעת נקטין את גודל הערימה ב-1, ונבצע $MAX\ HEAPIFY$ על השורש החדש. נחזור על התהליך עד שישאר לנו איבר אחד, והמערך יהיה ממורן.

```

HEAP SORT( $A, n$ )
  BUILD HEAP( $A, n$ )
  for  $i \leftarrow n$  downto 2
    swap( $A[1], A[i]$ )
    decrement HEAPSIZE( $A$ )
  MAX HEAPIFY( $A, 1$ )

```

כאן כמובן מתקבל זמן ריצה שהגורם הדומיננטי בו הוא ביצוע n פעמים $MAX\ HEAPIFY$, ולסיכום $O(n \log(n))$ במקרה הגרוע (כמו $MERGE\ SORT$).

עץ חיפוש בינארי

עץ חיפוש בינארי הוא מימוש של ADT המגדיר את הפעולות הבאות:

- $SEARCH(T, x)$ - מבצע חיפוש של איבר עם מפתח מסוים במבנה
- $MIN(T)$ - מחזיר את האיבר המינימלי במבנה
- $MAX(T)$ - מחזיר את האיבר המקסימלי במבנה
- $SUCCESSOR(T, x)$ - מחזיר את האיבר העוקב של איבר מסוים במבנה (מוגדר בתור האיבר המינימלי מבין האיברים הגדולים מ- x)
- $PREDECESSOR(T, x)$ - מחזיר את האיבר הקודם של איבר מסוים במבנה (מוגדר בתור האיבר המקסימלי מבין האיברים הקטנים מ- x)

אנו נממש עץ חיפוש בינארי (Binary Search Tree, או BST) באמצעות מצביעים, כאשר נשמור את שורש העץ – $root(T)$ ובכל קודקוד יהיו לנו המצביעים: $parent(x)$, $left(x)$, $right(x)$ וכן את פעולת הגישה למפתח – $key(x)$. היתרון במימוש של עץ באמצעות מצביעים על פני מערכים הוא שקל מאוד לעדכן מצביעים ואין צורך להקצות זיכרון מיותר – רק עבור הקודקודים שהם אכן חלק מהעץ אנו מקצים זיכרון.

הערה: בהמשך נניח לצורך הפשטות שהאיברים בעץ שונים זה מזה. עם זאת, ניתן להתמודד גם עם כפילויות ללא קושי.

הגדרה: עץ חיפוש בינארי הוא עץ בינארי שבו לכל קודקוד x מתקיימת תכונת עץ חיפוש בינארי:

- לכל קודקוד y בתת עץ השמאלי של x , מתקיים $key(y) < key(x)$
- לכל קודקוד y בתת עץ הימני של x , מתקיים $key(y) > key(x)$

מההגדרות בלבד, וללא שימוש בתוכן של האיברים בעץ, נוכל כבר בקלות לממש חלק מהפעולות

```

MIN( $T$ )
 $l \leftarrow \text{root}(T)$ 
while  $\text{left}(l) \neq \text{NIL}$ 
     $l \leftarrow \text{left}(l)$ 
return  $l$ 

```

```

MAX( $T$ )
 $r \leftarrow \text{root}(T)$ 
while  $\text{right}(r) \neq \text{NIL}$ 
     $r \leftarrow \text{right}(r)$ 
return  $r$ 

```

כלומר, בצורה הפשוטה ביותר, כדי למצוא את האיבר המינימלי הולכים בעץ שמאלה עד שנתקעים, וכדי למצוא את האיבר המקסימלי הולכים בעץ ימינה עד שנתקעים.

על מנת לממש חיפוש של איבר בעץ, נעבור על העץ מהשורש ובכל פעם נקבל החלטה האם אנחנו צריכים ללכת ימינה או שמאלה לפי השוואת הערך שמחפשים עם הערך בקודקוד הנוכחי. ניתן לממש זאת בקלות ברוסיה, אך גם הפתרון האיטרטיבי די פשוט.

```

SEARCH( $T, x$ )
if  $T = \text{NIL}$  or  $\text{key}(T) = x$  then return  $T$ 
if  $\text{key}(T) < x$  then return SEARCH( $\text{right}(T), x$ )
else return SEARCH( $\text{left}(T), x$ )

```

כל הפונקציות שתיארנו בינתיים פועלות ב- $\mathcal{O}(D)$ כאשר D גובה העץ. במקרה שבו העץ הוא למעשה "שרוך" (כלומר שרשרת שבה אין לאף קודקוד יותר מבן אחד), קיבלנו זמן חיפוש ליניארי - $\mathcal{O}(n)$. בהמשך נראה מבנה דומה, עץ AVL, שבו על ידי ביצוע איזון מכוון הבעיה הזו נמנעת.

כעת נתבונן במציאת איבר עוקב בעץ.

הגדרה: העוקב של x הוא האיבר המינימלי מבין האיברים הגדולים מ- x , $\text{succ}(x) \stackrel{\text{def}}{=} \min\{x' \in T : x' > x\}$.

נבחין בין שלושה מקרים למציאת העוקב.

1. אם $x = \max(T)$ אזי אין לו עוקב (NIL).
2. אם ל- x יש בן ימני, אזי העוקב של x הוא המינימום של תת העץ הימני הזה:
 $\text{succ}(x) = \min(\text{right}(y))$
3. אם אין ל- x בן ימני, אזי אנו צריכים לעלות בעץ ולחפש את הפעם הראשונה שהמקום ממנו עלינו הוא הבן השמאלי של המקום אליו הגענו (כלומר עולים בעץ עד שפונים ימינה).

```

SUCCESSOR( $T, x$ )
if  $\text{right}(x) \neq \text{NIL}$ 
    return  $\min(\text{right}(x))$ 
 $y \leftarrow \text{parent}(x)$ 
while  $y \neq \text{NIL}$  and  $x = \text{right}(y)$ 
     $x \leftarrow y$ 

```

```

y ← parent(x)
return y

```

טענה 17: האלגוריתם *SUCCESSOR* מחזיר תמיד את העוקב של x .

הוכחה: המקרה הראשון ברור, אם x הוא המקסימום אזי אין לו עוקב המקרה השני והשלישי פחות ברורים. במקרה השני, ל- x יש בן ימני. בקבוצת האיברים בתת העץ של x , כל האיברים הגדולים מ- x נמצאים ב- $right(x)$ והמינימום מביניהם הוא אכן העוקב של x בתוך תת העץ שלו. אבל האם ייתכן שהעוקב נמצא מחוץ לתת העץ של x ?

טענה 17א': (טענת הטווח) בעץ חיפוש בינארי, לכל תת עץ $T(x)$, כל האיברים שמחוץ ל- $T(x)$ נמצאים מחוץ לטווח הערכים $[MIN(T(x)), MAX(T(x))]$.

הוכחה: נבחר קודקוד מחוץ לתת העץ של x ונקרא לו z . נצייר את המסלולים מ- $MIN(T(x))$ ו- $MAX(T(x))$ עד השורש של העץ, ונסמן ב- w את הקודקוד שבו יש פיצול (אם קיים כזה). כעת נתבונן בשלושה מקרים:

- אם הפיצול ל- z הוא ימינה מהמסלול מהשורש ל- x , אזי $z > w > MAX(T(x))$ ולכן z מחוץ לטווח.
- אם הפיצול ל- z הוא שמאלה מהמסלול מהשורש ל- x , אזי $z < w < MIN(T(x))$ ולכן z מחוץ לטווח.
- אם z על המסלול מהשורש ל- x , אזי אם x נמצא בתת עץ השמאלי של z , מתקיים $z > MAX(T(x))$, ואם x נמצא בתת עץ הימני של z אזי $z < MIN(T(x))$ ובשני המקרים הוא מחוץ לטווח.

■

כעת מהטענה ברור שהעוקב נמצא בתת העץ הימני של x אם קיים כזה, כי אם הוא קיים, אזי בתת העץ הזה קיים איבר כלשהו הגדול מ- x , וכל האיברים מחוץ לתת העץ של x גדולים באיבר זה ולכן לא מקיימים את מינימליות העוקב.

במקרה השלישי, ל- x אין בן ימני, וכעת למעשה תהליך חיפוש העוקב הוא בדיוק ההפך מלחפש את המקסימום בתת העץ השמאלי של העוקב שנמצא (נעלה למעלה עד שנפנה ימינה). במקרה זה נסמן ב- y את הקודקוד שמצאנו ונתבונן בתת העץ שלו $T(y)$. בתוך תת עץ זה, ברור ש- y הוא העוקב של x שכן $x = MAX(left(y))$ (כדי להגיע מ- y ל- x פונים שמאלה ואז ימינה עד הסוף, שזה המקסימום). כלומר, מתקבל ש- x הוא המקסימום של כל האיברים הקטנים מ- y , ולכן y הוא המינימום של כל האיברים הקטנים מ- x . מחוץ לתת העץ $T(y)$ אין לנו בכלל טעם להסתכל כיוון שלא קיים ערך בין x לבין y הנמצא מחוץ ל- $T(y)$, על פי טענת הטווח. ■

סיבוכיות זמן הריצה של *SUCCESSOR* היא שוב $O(D)$ כיוון שבכל המקרים אנו מטפסים במסלול במעלה העץ או במורד העץ.

טענה 18: הפעלת *SUCCESSOR* ברצף החל מהאיבר המינימלי בעץ ועד האיבר המקסימלי בעץ הוא אלגוריתם בסיבוכיות $O(n)$.

TRAVERSE SUCCESSOR(T)

```

y ← MIN(T)
while y ≠ NIL do y ← SUCCESSOR(T, y)

```

הוכחה: האלגוריתם עובר פעמיים בדיוק על כל צלע בעץ, ולכן כמובן זמן הריצה שלו הוא $O(n)$. על מנת להוכיח זאת ניעזר בשתי טענות עזר:

טענה 18א': עבור כל תת עץ בגובה h , לאחר שהאלגוריתם נכנס אליו הוא עובר בדיוק פעמיים על כל צלע ויוצא מתת העץ.

הוכחה: אינדוקציה על h . הבסיס עבור $h = 1$ נכון. נניח עבור עצים בגובה h ונוכיח עבור עצים בגובה $h + 1$. כאשר האלגוריתם מגיע לשורש של עץ בגובה $h + 1$ בפעם הראשונה, הוא עושה זאת בדרך למטה. בין אם x הוא בן ימני או שמאלי, הכיוון הוא לחפש את המינימום בתת העץ של x , כלומר הולכים שמאלה ועוברים על הצלע $\{x, left(x)\}$ בפעם הראשונה. לאחר שנסיים בתת העץ השמאלי של x נחזור שוב על הצלע בכיוון מעלה, כלומר על $\{left(x), x\}$ ובאופן דומה עבור $\{x, right(x)\}$. לפי טענה 18ב' לא נחזור לעולם לתת העץ של x ולכן לא נעבור שוב על צלעות אלה. לכן על שתי צלעות אלה ביצענו בדיוק שני מעברים, ובכל צלע בתתי העצים השמאלי והימני של x בוודאי נעבור על כל צלע פעם אחת, מתוקף הנחת האינדוקציה. ■

טענה 18ב': לאחר שהאלגוריתם יצא מתת עץ בגובה h הוא לא יחזור אליו לעולם.

הוכחה: אינדוקציה על h בסדר הפוך. בסיס H גובה העץ כולו, נכון באופן ריק כי לא ניתן לצאת מחוץ לשורש של העץ. עבור קודקוד $x, h(x) = h$ האלגוריתם נכנס קודם כל לתת העץ השמאלי. בשלב מסוים האלגוריתם יוצא ממנו ומגיע ל- $x = \text{SUCCESSOR}(\text{MAX}(\text{right}(x)))$. לאחר פעולת SUCCESSOR נוספת האלגוריתם יבקר בתת העץ הימני של x . כשחוזרים מתת העץ הימני, לא נעצור ב- x אלא נצא מתת העץ שלו ונגיע להורה של x . מהנחת האינדוקציה $(h(\text{parent}(x)) > h(x) = h)$ לא נחזור יותר ל- x ובפרט לא נחזור לכל תת העץ שלו. ■

משתי הטענות יחד ברור שעוברים בדיוק פעמיים על כל צלע כפי שרצינו. ■

כעת נותרו לנו פעולות ההכנסה והמחיקה, שמבצעות שינויים בעץ. ההכנסה היא פעולה פשוטה יחסית – נחפש את המקום שבו האיבר החדש צריך היה להיות אילו היינו מחפשים אותו, ושם נכניס אותו. כמובן, כל איבר חדש שנכנס לעץ הוא עלה (לא ניתן להכניס איברים ב"אמצע" העץ).

INSERT(T, z)

$y \leftarrow \text{NIL}$

$x \leftarrow \text{root}(T)$

while $x \neq \text{NIL}$

$y \leftarrow x$

if $\text{key}(z) < \text{key}(x)$ then $x \leftarrow \text{left}(x)$

else $x \leftarrow \text{right}(x)$

if $y = \text{NIL}$ then $\text{root}(T) \leftarrow z$

else

if $\text{key}(z) < \text{key}(y)$ then $\text{left}(y) \leftarrow z$

else $\text{right}(y) \leftarrow z$

$\text{parent}(z) \leftarrow y$

גם כאן סיבוכיות זמן הריצה של האלגוריתם היא פשוט $\mathcal{O}(D)$ כיוון שאנחנו מתקדמים במסלול יחיד במורד העץ.

המחיקה היא פעולה עדינה יותר – כאן נצטרך להבחין בין שלושה מקרים, כדי שלא לפגוע בתכונת ה-BST:

1. אין לקודקוד בנים כלל. במקרה זה אפשר פשוט למחוק אותו ורק צריך לעדכן את ההורה שלו.
2. יש לקודקוד בן אחד. במקרה זה ניתן לחבר את ההורה של הקודקוד לבן שלו (עוקפים את הקודקוד).
3. יש לקודקוד שני בנים. במקרה זה נדרשת העדינות המירבית. ניעזר בטענה 18:

טענה 19: אם ל- z שני בנים, לעוקב של z אין בן שמאלי.

הוכחה: אם היה לעוקב של z שנשמנו s בן שמאלי שנשמנו s' , אזי מתקיים $s > s' > z$ בסתירה להגדרת העוקב. (ניתן גם להתבונן במקרים הפרטיים של מציאת עוקב, כאן אנחנו במקרה השני של מציאת עוקב ואין לעוקב בן שמאלי כי אילו היה כזה, הוא היה $\min(\text{right}(z))$ ולא העוקב עצמו.) ■

כעת במקרה 3, לפי טענה 19, אפשר פשוט למחוק את העוקב של z מעץ ולהעביר אותו למקום שבו היה z (המחיקה לא מורכבת כי לעוקב יש לכל היותר בן אחד), ועדיין נשמרת תכונת BST כי בין העוקב של איבר מסוים לבין האיבר הזה אין אף איבר אחר (לפי ההגדרה).

DELETE(T, z)

```
if left(z) = NIL or right(z) = NIL
  child ← left(z)
  if child = NIL then child ← right(z)
  if z = left(parent(z)) then left(parent(z)) ← child
  else right(parent(z)) ← child
  delete z
else
  y ← SUCCESSOR(T, z)
  value(z) ← value(y)
  DELETE(T, y)
```

וכאן במקרים 1, 2 סיבוכיות זמן הריצה היא פשוט $O(1)$ אבל במקרה 3 אנו צריכים למצוא את העוקב, ולכן סיבוכיות זמן הריצה היא $O(D)$ כמו ביתר הפעולות.

ניתן להציע גם אלגוריתם אלטרנטיבי למחיקה מעץ חיפוש בינארי. אם יש ל- z שני ילדים, נחבר את תת העץ השמאלי שלו בתור תת העץ השמאלי של העוקב שלו, ואז נמחק את z (שכעת יש לו בן אחד בלבד). האלגוריתם נראה כך:

DELETE2(T, z)

```
if left(z) = NIL or right(z) = NIL do same as above
else
  y ← SUCCESSOR(T, z)
  left(y) ← left(z)
  DELETE2(T, z)
```

דבר שימושי נוסף שאפשר לשאול על עץ הוא מה הקוטר שלו – מקסימום המרחקים בין כל שני קודקודים בעץ. אם ננסח את הבעיה אחרת, הפתרון יהיה ברור: הקוטר של העץ הוא למעשה המקסימום בין סכום הגבהים של תת העצים (ועוד 2 עבור הצלעות לשורש), לבין הקוטר של תת העץ השמאלי והקוטר של תת העץ הימני. מניסוח זה ברור הפתרון הרקורסיבי, בהינתן הגובה של כל קודקוד:

DIAMETER(x)

```
if x = NIL then return 0
ld ← DIAMETER(left(x))
rd ← DIAMETER(right(x))
```

```

 $lp \leftarrow height(left(x)) + height(right(x)) + 2$ 
return max{ld, rd, lp}

```

הערה: אם העץ מאוזן, התשובה יכולה להיות פשוטה יותר. למשל, בעץ AVL שבו מובטח שהפרש הגבהים בין תת העצים הוא לכל היותר 1, הקוטר הוא פשוט סכום הגבהים של תת העצים של השורש, פלוס 2.

Order Statistics

הגדרה: ה-order statistic ה- i בקבוצה $\{a_1, \dots, a_n\}$ הוא האיבר ה- i בקבוצה $\{a'_1, \dots, a'_n\}$ שבה מתקיים לכל $j < j'$ ש- $a_j < a'_{j'}$ (כלומר זהו האיבר ה- i הקטן ביותר).

נרצה לפתור בעיה קטנה – כיצד מוצאים את ה-order statistic ה- i במבנה מסוים? במערך ממוין זה לא קשה במיוחד, אם האיברים כולם שונים זה מזה אזי זו פעולה של $O(1)$. בעץ חיפוש בינארי, לדוגמה, זו בעיה קשה יותר. פתרון נאיבי לדוגמה הוא למצוא את המינימום ואז לקרוא ל-*SUCCESSOR* במשך i פעמים. זה אמנם $O(n)$ (לפי טענה 18) אבל ניתן לעשות משהו טוב יותר.

הצעה ראשונה: בכל קודקוד נחזיק את ה-order statistic שלו. מציאת ה-OS ה- i היא אפוא פעולה ב- $O(D)$ כאשר D גובה העץ. הבעיה המרכזית עם ההצעה הזאת היא התחזוקה – מחיקה או הכנסה לעץ הופכות להיות פעולות מאוד מורכבות.

הצעה שנייה: בכל קודקוד נחזיק את גודל תת העץ שלו (נקרא לו *size*). כאשר מבצעים חיפוש, ניתן להסתכל על הגודל של ה-*בן השמאלי* כדי להבין האם צריך לחפש שם או ב-*בן הימני*.

OS FIND(*root*, *i*)

```

lsize ← 0
if left(root) ≠ NIL then lsize ← size(left(root))
if lsize ≥ i then return OS FIND(left(root), i)
if lsize + 1 = i then return root
return OS FIND(right(root), i - lsize - 1)

```

נשים לב שעכשיו ההכנסה וההוצאה מהעץ הן פעולות יותר זולות, כי בכל פעולה כזו אנו לא מבצעים שינויים מחוץ למסלול לקודקוד שנמחק או הוכנס, ולכן קל לנו יחסית לשלוט על ה-*size* השמור בכל קודקוד. ניתן אף לשפר אותו ולשמור בכל קודקוד את הגודל של תת העץ השמאלי שלו – שזה מה שאנחנו באמת משתמשים בו באלגוריתם.

ניתן להשתמש בעץ שבו בכל קודקוד יש את גודל תת העץ שלו לצרכים נוספים למשל לצורך מימוש האלגוריתם *COUNT RANGE* שבדק כמה קודקודים בעץ נמצאים בטווח $[a, b]$ (כאשר הקצוות של הטווח לא בהכרח נמצאים בעץ).

COUNT RANGE(*T*, *a*, *b*)

```

return COUNT BIGGER(root(T), a) - COUNT BIGGER(root(T), b)

```

COUNT BIGGER(*x*, *a*)

```

if x = NIL then return 0
if a ≤ key(x) then return 1 + size(right(x)) + COUNT BIGGER(left(x), a)
else return COUNT BIGGER(right(x), a)

```

עצי AVL

נשים לב שכל הפעולות שהגדרנו תלויות בגובה העץ – אם גובה העץ ליניארי במספר האיברים, אזי קיבלנו מבנה מאוד לא יעיל המזכיר רשימה מקושרת. לחלופין, אם העץ הוא עץ שלם (או כמעט שלם), אזי הפעולות לוגריתמיות במספר האיברים. אם כך, המטרה שלנו צריכה להיות לאזן את העץ – להביא את העץ למצב שבו גובהו הוא $\mathcal{O}(\log(n))$ עבור n קודקודים.

קיימות הרבה דרכים לבנות עצים כאלה, ביניהן עצי 2-3-4, עצים אדומים-שחורים, וגם עצי AVL שיעסיקו אותנו. הגובה של עצים אלה הוא לוגריתמי במספר האיברים, מה שגורם לפעולות להיות מאוד מהירות. כמובן יש לזה מחיר – איזון העץ תוך כדי ביצוע פעולות המשנות אותו (הכנסה ומחיקה) היא משימה שלוקחת זמן.

הגדרה: עץ חיפוש בינארי הוא עץ AVL אם לכל קודקוד x מתקיים $|h(\text{left}(x)) - h(\text{right}(x))| \leq 1$, כלומר ההבדל בגבהים של תת העץ השמאלי והימני לכל קודקוד הוא לכל היותר 1.

הערה: חשוב לשים לב שתכונת AVL צריכה להתקיים בכל קודקוד, ולא רק בשורש.

על מנת להרוויח מהשימוש ב-AVL, עלינו להוכיח שגובה העץ הוא אכן לוגריתמי. נסמן n_k מספר הקודקודים המינימלי בעץ AVL בגובה k .

טענה 20: n_k עולה מונוטונית ב- k , כלומר $n_{k+1} > n_k$.

הוכחה: נתבונן בעץ AVL בגובה $k + 1$ שבו מספר הקודקודים הוא מינימלי. אחד מהבנים של השורש הוא עץ AVL בגובה k (אחרת גובה העץ לא היה $k + 1$). בתת עץ זה שגובהו k יש בדיוק n_k קודקודים (שכן אם יש בו יותר קודקודים, העץ כולו לא היה בעל מספר הקודקודים המינימלי בגובה $k + 1$, ואם יש בו פחות – זו סתירה להגדרת n_k). לכן ברור ש- $n_{k+1} \geq n_k + 1$ (בגלל השורש עצמו), אבל כמובן כיוון שגם קיים תת עץ נוסף בגובה k או $k - 1$ (בפועל – בהכרח בגובה $k - 1$ משיקולי מינימליות), העלייה המונוטונית היא אפילו ביותר מ-1. ■

כעת אנו יכולים לכתוב נוסחת רקורסיה על מספר הקודקודים המינימלי:

$$n_k = n_{k-1} + n_{k-2} + 1$$

הערה: משיקולי מינימליות, אחד מתת העצים של עץ AVL בעל מספר קודקודים מינימלי בגובה k הוא עץ AVL בעל מספר קודקודים מינימלי בגובה $k - 1$, והשני בגובה $k - 2$. לכן מתקבלת הנוסחה.

על ידי הוספת 1 לשני האגפים נוכל לקבל:

$$\begin{aligned}(n_k + 1) &= (n_{k-1} + 1) + (n_{k-2} + 1) \\ F_k &\stackrel{\text{def}}{=} n_k + 1 \\ F_k &= F_{k-1} + F_{k-2}\end{aligned}$$

והנוסחה האחרונה שקיבלנו היא בדיוק הנוסחה לאיברי סדרת פיבונאצ'י. נוכל לחסום את n_k בכמה דרכים.

טענה 21: $k \leq 2 \log(n) + \mathcal{O}(1)$

הוכחה: נשתמש במשוואת הרקורסיה המקורית ונוציא ממנה אי-שוויון משיקולי מונוטוניות (טענה 20), ואז נשתמש בשיטת האיטרציה:

$$n_k = n_{k-1} + n_{k-2} + 1$$

$$\begin{aligned}
 n_k &> 2n_{k-2} + 1 > 2(2n_{k-4} + 1) + 1 = 2^2 n_{k-4} + 2 + 1 > 2^2(2n_{k-6} + 1) + 2 + 1 \\
 &= 2^3 n_{k-6} + 2^2 + 2 + 1 > \dots > 2^{\frac{k}{2}} n_0 + 2^{\frac{k}{2}-1} + \dots + 2 + 1
 \end{aligned}$$

כאשר כאן כמובן $n_0 = 1$ (גודל עץ AVL המינימלי בגובה 0 – קודקוד אחד). לכן קיבלנו כאן סדרה הנדסית, וסכומה $1 - 2^{\frac{k}{2}+1}, \dots$, לכן יש לנו:

$$\begin{aligned}
 n_k &> 2^{\frac{k}{2}+1} \\
 n_k + 1 &> 2^{\frac{k}{2}+1} \\
 \log(n_k + 1) &> \frac{k}{2} + 1 \\
 2 \log(n_k + 1) - 2 &> k \\
 k < 2 \log(n_k + 1) - 2 &\leq 2 \log(2n_k) - 2 = 2 \log(n_k)
 \end{aligned}$$

כלומר יש לנו $k \leq 2 \log(n_k)$ כמו שרצינו. (במקרה האי-זוגי מתקבל גם הגורם של $\mathcal{O}(1)$ כפי שדרשנו בטענה). כיוון שאנו יודעים ש- n_k הוא מספר הקודקודים המינימלי בעץ בגובה k , ברור ש- $k \leq 2 \log(n_k) \leq 2 \log(n)$ שרצינו. ■

טענה 22: $k \leq 1.44 \log(n_k) + \mathcal{O}(1)$ (חסם הדוק יותר)

הוכחה: ראינו את הקשר בין סדרת פיבונאצ'י לבין n_k , בפרט $n_k = F_{k+3} - 1$. $\forall k$, נשתמש בידוע לנו על מספרי פיבונאצ'י (את השוויון הראשון קל להראות באינדוקציה או ישירות ע"י התבוננות בפתרונות המשוואה הריבועית $x^2 = x + 1$):

$$\begin{aligned}
 F_n &= \frac{1}{\sqrt{5}} \left[\left(\frac{1+\sqrt{5}}{2} \right)^n - \left(\frac{1-\sqrt{5}}{2} \right)^n \right] \\
 \left| F_n - \frac{1}{\sqrt{5}} \left(\frac{1+\sqrt{5}}{2} \right)^n \right| &= \left| -\frac{1}{\sqrt{5}} \left(\frac{1-\sqrt{5}}{2} \right)^n \right| < 1
 \end{aligned}$$

לכן נוכל פשוט להציב:

$$\begin{aligned}
 n_k &\geq \frac{1}{\sqrt{5}} \left(\frac{1+\sqrt{5}}{2} \right)^{k+3} \\
 \log(n_k) &\geq (k+3) \log\left(\frac{1+\sqrt{5}}{2}\right) - \log(\sqrt{5}) \\
 k &\leq 1.44 \log(n_k) + \mathcal{O}(1)
 \end{aligned}$$

■

אנחנו מאוד מרוצים מגובה עץ AVL, וכעת נרצה לראות כיצד ניתן לתחזק את הגובה הזה – כיצד ניתן לשמור על העץ מאוזן אבל עדיין לדאוג לכך שפעולות ההכנסה והמחיקה יישארו יעילות יחסית

הערה: כל הפעולות על עץ חיפוש בינארי שאינן משנות אותו (למשל חיפוש, מציאת עוקב וכד') נכונות עדיין בעץ AVL ובעלות אותה סיבוכיות, כאשר כעת אנו יודעים ש- $D = \mathcal{O}(\log(n))$.

על מנת לבצע את הבדיקה האם העץ עדיין מאוזן, ולעשות זאת בצורה יעילה, נשמור בכל קודקוד את הגובה שלו (גובה תת העץ שהוא השורש שלו), כאשר נתייחס לגובה של תת עץ ריק בתור 1-. נבצע את ההכנסה באופן רגיל, אך לאחר שהגענו למקום שבו ביצענו את ההכנסה, נטפס במעלה העץ ובכל קודקוד נעדכן את הגובה בהתאם. נשים לב שבמהלך ביצוע העדכונים הגבהים של תתי עצים שאינם נמצאים במסלול אל הקודקוד המוכנס בוודאי לא נפגעו, כלומר הפרת תכונת AVL (אחת או יותר) יכולה להתרחש רק בקודקודים בדרך מהשורש לקודקוד המוכנס.

כאשר נזהה הפרה של תכונת AVL (כלומר נמצא קודקוד $x : |h(\text{left}(x)) - h(\text{right}(x))| > 1$), נרצה לתקן אותה באמצעות רוטציות של העץ. כמובן נרצה גם שהרוטציות ישמרו את תכונת ה-AVL ולא יפגעו בתת עצים אחרים, וכן שהשדה המחזיק את הגובה של הקודקוד יהיה נכון ומעודכן נבחין בין ארבעה מקרים שונים באשר למוקד ההפרה.

במקרה LL, זיהינו את ההפרה בקודקוד B , והקודקוד שהוכנס נמצא תחת הבן השמאלי (שיסומן A_L) של תת הבן השמאלי של B (שיסומן A) – לכן המקרה נקרא LL, כי יש ללכת פעמיים שמאלה מ- B כדי להגיע לתת העץ שבו בוצעה ההכנסה. כמו כן נסמן גם את הבן הימני של B ושל A באמצעות B_R, A_R בהתאמה. הטענה שיש לנו הפרה של תכונת AVL בקודקוד B משמעותה ש- $|h(B_L) - h(B_R)| > 1$. הדבר המעניין הוא שאנחנו יכולים לקבוע באופן חד משמעי מה הגבהים היחסיים של העצים השונים שסימנו קודם

- נניח שלפני ההכנסה הגובה של A_L היה h .
- הגובה של A_R יכול להיות $h - 1, h, h + 1$. בפועל, אם הגובה היה $h + 1$ אזי לא הייתה לנו הפרה לאחר ההכנסה (כי לכל היותר A_L יכול להפוך להיות בגובה $h + 1$). באופן דומה, אם הגובה היה $h - 1$ אזי לאחר ההכנסה ב- A_L נקבל הפרה כבר ב- A (כי גובהו של A_L הופך להיות $h + 1$ - אחרת לא הייתה לנו הפרה בכלל), אבל אנו יודעים שההפרה היא רק ב- B . לכן בהכרח הגובה של A_R הוא h .
- לפני ההכנסה הגובה של A היה $h + 1$, ולכן הגובה של B_R יכול להיות $h + 2, h + 1, h$. רק המקרה h ייתכן כי בשני המקרים האחרים (לאחר שהגובה של A הופך ל- $h + 2$) לא מתרחשת הפרה. כלומר הגובה של B_R הוא h .
- לסיכום, לאחר ביצוע ההכנסה הגבהים הם:

$$h(A_L) = h + 1, h(A_R) = h, h(A) = h + 2, h(B_R) = h, h(B) = h + 3$$

על מנת לתקן את המצב, נבצע רוטציית LL – נתלה את העץ על הקודקוד A במקום על הקודקוד B , כך ש- B הופך להיות הבן הימני של A , הבן הימני של A הופך לבן השמאלי של B ו- A מחליף את B במקום שהוא היה:

$$\begin{aligned} &LL(B) \\ &A \leftarrow \text{left}(B), A_R \leftarrow \text{right}(A) \\ &\text{left}(B) \leftarrow A_R \\ &\text{right}(A) \leftarrow B \end{aligned}$$

השמטנו כאן את הקוד המטפל ב- $\text{parent}(x)$ עבור כל הקודקודים המעורבים, למען הקריאות. לאחר ביצוע הרוטציה, הפרת תכונת ה-AVL תוקנה – כעת הגבהים הם:

$$h(A) = h + 2, h(A_L) = h + 1, h(B) = h + 1, h(A_R) = h, h(A_L) = h$$

נשים לב גם שלא פגענו בתכונת ה-BST, כיוון שהרוטציה שומרת עליה. בפרט, כיוון ש- A היה בן שמאלי של B , אנחנו יכולים להפוך את B להיות הבן הימני של A (כל האברים בו גדולים מ- A), וכן כיוון שהבן הימני של A נמצא

בתת עץ השמאלי של B אנחנו יכולים להפוך אותו להיות הבן השמאלי של B (כל האיברים בו קטנים מ- B), ובכל אופן הוא נמצא בתת עץ הימני של A כמו קודם.

במקרה RR, ההפרה היא בקודקוד B כך שהקודקוד שהוכנס נמצא תחת הבן הימני (A_R) של תת הבן הימני (A) של B . מקרה זה זהה עד כדי סימטריה למקרה הקודם, אלא שאנו נדרשים לרוטציה הפוכה – רוטציית RR.

$RR(B)$

$A \leftarrow right(B), A_L \leftarrow left(A)$

$right(B) \leftarrow A_L$

$left(A) \leftarrow B$

במקרה LR, ההפרה היא בקודקוד C , והקודקוד שהוכנס נמצא תחת הבן הימני (B) של תת העץ השמאלי (A) של C . כאן הסיטואציה מורכבת יותר. נניח קודם את המקרה LRR, כלומר שההכנסה הייתה תחת הבן הימני של B שנשמנו B_R . במקרה זה שוב אנו יכולים לקבוע מראש את הגבהים של כל הקודקודים המעורבים

$$h(B_R) = h \Rightarrow h(B_L) = h \Rightarrow h(B) = h + 1 \Rightarrow h(A_L) = h + 1 \Rightarrow h(A) = h + 2 \Rightarrow h(C_R) = h + 1 \\ \Rightarrow h(C) = h + 3$$

לאחר ההפרה הגבהים משתנים כמובן:

$$h(B_R) = h + 1, h(B) = h + 2, h(A) = h + 3, h(C) = h + 4$$

אם נבצע רוטציה רגילה (LL) אזי נקבל חוסר איזון תחת C_R . במקום זאת, נעשה קודם רוטציית RR מסביב ל- A , ואז רוטציית LL מסביב ל- C – כלומר אנחנו צריכים שתי רוטציות!

$LRR(C, A)$

$RR(A)$

$LL(C)$

לאחר ביצוע הרוטציות נקבל את הגבהים המתוקנים הבאים:

$$h(B_R) = h + 1, h(C) = h + 2, h(A) = h + 2, h(B) = h + 3$$

כאשר כמובן שורש תת העץ בסביבת ההפרה של C (לאחר ביצוע הרוטציות) יהיה B . בנוסף, אין אפילו צורך לבדוק ששמרנו על תכונת BST כיוון שהרוטציות LL, RR שומרות בנפרד על תכונת BST ולכן גם אם מבצעים אותן זו אחר זו – תכונת BST עדיין נשמרת.

במקרה LRL, כלומר כאשר הקודקוד שהוכנס נמצא תחת B_L ולא תחת B_R , ניתן לבצע רוטציית LRR בדיוק כמו קודם, ולקבל את אותה תוצאה. לכן למעשה אפשר לדבר על מקרה LRR ומקרה LRL כמקרה אחד של LR.

במקרה RL, ההפרה היא בקודקוד C , והקודקוד שהוכנס נמצא תחת הבן השמאלי (B) של תת העץ הימני (A) של C . מקרה זה הוא סימטרי למקרה LR (וגם כאן ניתן להבחין בשני תת מקרים זהים RLL, RLR), אלא שכאן מבצעים קודם רוטציית LL סביב A ולאחר מכן רוטציית RR סביב C :

$RL(C, A)$

$LL(A)$

$RR(C)$

בכל המקרים, עלות כל רוטציה היא $O(1)$ ועלות ההכנסה ועדכון הגבהים היא במצטבר הדומיננטית, ומקבלים בכל זאת הכנסה ב- $O(\log(n))$ כמו שרצינו.

ב- http://en.wikipedia.org/wiki/AVL_tree ניתן למצוא תרשימים והסברים מפורטים המדגימים את אופן הפעולה של הרוטציות השונות.

הערה: על מנת לוודא שכל הדין הנ"ל מוגדר היטב, חשוב לציין שלא ייתכן שהמסלול בין הקודקוד שהוכנס לבין נקודת ההפרה קצר יותר משני צעדים. נניח למשל שהמרחק היה צעד אחד, ונזכור שהקודקוד שהוכנס הוא עלה. בין אם להורה של הקודקוד שהוכנס יש בן נוסף ובין אם לא, הפרש הגבהים המקסימלי שיכול להיווצר כתוצאה מכך הוא 1, ולכן לא תיתכן הפרה.

הערה: כאשר מתרחשת הפרה בהכנסה, הגובה של תתי העצים לא משתנה. כלומר, לאחר ביצוע הרוטציות, הגבהים של תתי העצים חוזרים לקדמותם לפני ההכנסה. לעומת זאת, דווקא כאשר לא מתרחשת הפרה, ייתכן מצב שבו גובה העץ כולו משתנה, למשל אם מכניסים קודקוד חדש לעץ AVL שהיה עץ בינארי שלם.

נותר לנו לטפל במחיקה מעץ AVL. נבחין כרגיל בין שלושת המקרים עבור קודקוד z שרוצים למחוק:

1. אין ל- z ילדים, מוחקים אותו ישירות
2. ל- z ילד אחד, מחברים את ההורה שלו לילד שלו
3. ל- z שני ילדים, ואז מחליפים אותו עם העוקב שלו ומוחקים את העוקב

למרות שהמקרה השלישי נראה המורכב ביותר, דווקא במקרה זה לא מתבצעת פגיעה בתכונת AVL כיוון שאנחנו לא באמת מבצעים את המחיקה. המקרים המדאיגים הם המקרה הראשון והשני, שבמקרה השלישי משתמש בהם. אם כך, במחיקה מעץ AVL נבצע את המחיקה ואז נתחיל לעלות לשורש עד שנגלה הפרה בנקודה שבה מתגלה ההפרה, מסתכלים על המסלול לקודקוד העמוק ביותר מנקודת ההפרה (בניגוד למצב ההכנסה, שם ידענו מי הקודקוד שהוכנס).

נניח שההפרה התגלתה בקודקוד A ומחקנו קודקוד מתת העץ השמאלי (A_L) של A . כיוון שמחקנו מתת העץ השמאלי, המסלול הארוך ביותר לעלה כלשהו בהכרח נמצא בתת העץ הימני של A , B . שוב אין לגבהי תת העצים ברירה, והם מוכרחים להיות (לפני המחיקה): $h(A_L) = h + 1, h(B) = h + 2$. לאחר המחיקה, מתקבל חוסר איזון ב- A כיוון ש- $h(A_L) = h$.

עם זאת, אין וודאות האם המסלול הארוך ביותר הוא ב- B_L (מה שמביא אותנו למצב RL) או ב- B_R (מה שמביא אותנו למצב RR). נניח שאנחנו במקרה RR, ואז נבצע רוטציית RR סביב A ולאחר ביצוע הרוטציה נקבל גבהים מעודכנים: $h(A) = h + 1, h(B) = h + 2, h(B_L) = h + 3$. כיוון שיתכן שהגובה של B (שקודם היה A) הפך מ- $h + 3$ ל- $h + 2$, יכול להיות שגרמנו להפרה נוספת במעלה העץ. לכן עלינו להמשיך ולטפס למעלה עד השורש ולתקן את ההפרות אם נתקלים בהן.

שאר המקרים דומים מאוד למקרים שכבר התבוננו בהם לעומק במקרה ההכנסה. במקרים RR ו-LL מספיקה רוטציה אחת, במקרים LR ו-RL יש צורך בשתי רוטציות. בכל המקרים על מנת לתקן את ההפרות מבצעים לכל היותר $O(\log(n))$ רוטציות, כל אחת במחיר קבוע של $O(1)$, וכן כמובן משלמים את מחיר המחיקה עצמו, שהוא $O(\log(n))$. לכן סיבוכיות המחיקה היא $O(\log(n))$ כפי שרצינו.

בעיה מעניינת הקשורה לעצי AVL היא בעיית המיזוג שלהם. בהינתן שני עצי AVL, כיצד ניתן למזג אותם לעץ AVL אחד? הרעיון הטריטוריאלי הוא לבנות עץ חדש באמצעות הכנסת כל הקודקודים משני העצים זה כמובן ייקח $O(n \log(n))$. לחלופין, אפשר לנסות להתייחס לכמה מקרים פרטיים ולהסיק מכך את הרעיון הכללי:

- אם גבהי העצים שווים, וכן ידוע לנו $MAX(T_1) < MIN(T_2)$, אזי נניח ש- T_2 הוא העץ הגדול יותר. נוציא מתוכו את האיבר המינימלי, שהוא בהכרח גדול מכל האיברים של T_1 , ונקבע אותו להיות השורש של העץ החדש. משמאלו נושיב את T_1 ומימינו את מה שנשאר מ- T_2 ונקבל עץ המקיים את תכונת BST. מדוע הוא עץ AVL? לפני ההוצאה לא היה הבדל בגבהי העצים כלל, ולכן לאחר ההוצאה ההבדל יכול להיות לכל היותר 1 – וזהו הבדל חוקי בעץ AVL. אם T_1 הוא העץ הגדול יותר, אזי נחזור על אותו הרעיון אלא שנקבע את השורש להיות האיבר המקסימלי ב- T_2 .
- אם גבהי העצים אינם שווים, אך בכל זאת ידוע $MAX(T_1) < MIN(T_2)$ אזי נלך למינימום של T_2 ונעלה למעלה עד שנגיע לקודקוד x , שהבנים שלו הם T_2', T_2'' והוא עצמו בגובה של T_1 . נוציא מתוך T_2' את המינימום שלו m , והוא יהיה השורש של T_1 משמאל ו- T_2' מימין. m גם יהיה הבן השמאלי החדש של x (הבן הימני שלו נשאר T_2''). כעת יש לנו הפרה אפשרית של תכונת AVL ב- x , וניתן לתקן אותה באמצעות רוטציות וסיימנו את המיזוג.

במקרה הכללי, כאשר לא ידוע לנו דבר על T_1, T_2 , נוכל להשתמש ברעיון הבא: נעבור על שני העצים במקביל באמצעות קריאות רצופות ל- $SUCCESSOR$ (או כל מעבר אחר שהוא in-order), ובכל איטרציה נבחר את המינימלי מבין שני המצביעים, ואותו נכניס לעץ החדש. הדבר היפה כאן הוא שאנחנו יכולים לבנות את עץ ה-AVL מראש, רק לא לשים בו ערכים, ולאכלס אותו רק במהלך המעבר על שני העצים שאותם ממצגים. אם נעשה זאת, אזי נצטרך גם להחזיק איטרטור שעובר על עץ ה-AVL (שמתחיל עם קודקודים ללא ערכים) באמצעות $SUCCESSOR$. למרות שנראה שאלגוריתם זה עובד ב- $O(n \log(n))$, הוא מפתיע לטובה בגלל טענה 18 – מעבר על כל העץ באמצעות $SUCCESSOR$ הוא פעולה של $O(n)$.

בעיה אחרת היא בעיית הבניה של עץ AVL ממערך ממוין. ניתן לעשות זאת באופן הבא, על ידי חלוקה של המערך לחצאים בכל פעם שצריך לבחור בן ימני ובן שמאלי לשורש הנוכחי; ברקורסיה:

BUILD AVL($A[1 \dots n]$, start, end)

$N \leftarrow \text{start} - \text{end} + 1$

if $N < 0$ then return NIL

$r \leftarrow \text{start} + \left\lfloor \frac{n-1}{2} \right\rfloor$

$\text{tree} \leftarrow \text{new Node}(A[r])$

if $N > 1$

$\text{left}(A[r]) \leftarrow \text{BUILD AVL}(A, \text{start}, r-1)$

$\text{right}(A[r]) \leftarrow \text{BUILD AVL}(A, r+1, \text{end})$

return tree

זמן ריצת האלגוריתם הוא כמובן $O(n)$ כי אנחנו עוברים על כל איבר במערך פעם אחת ומבצעים לגבי מספר קבוע של פעולות (בניית node, השמת left, right וכד'). נותר רק להראות שהוא עובד. נתחיל מלהראות תכונת BST, באינדוקציה על גודל העץ n . עבור $n = 1$ העץ חוקי. נניח עבור n ונרצה להראות שעבור $n + 1$ נוצר עץ חוקי. השורש נמצא באינדקס ה- r במערך, ולכן הוא גדול מכל האיברים שנמצאים בין $[start, r-1]$ וקטן מכל האיברים הנמצאים בין $[r+1, end]$ (כיוון שהמערך ממוין). לפי הנחת האינדוקציה שני תת העצים הנ"ל מקיימים את תכונת BST. כעת נרצה להראות שמתקיימת גם תכונת AVL, נעשה זאת באינדוקציה על n . שוב מקרה הבסיס

טרינארלי (עץ בגובה 0), אז נניח לכל $n < k$ ונוכיח עבור n . לשורש העץ שני בנים עם $r = \lfloor \frac{n-1}{2} \rfloor$, $l = \lfloor \frac{n-1}{2} \rfloor$ קודקודים. ממנוטוניות הגובה ברור ש- $H_n = H_l + 1$ ובאותו אופן אם נתבונן על עץ בגודל $n-1$, שנשמך את הגדול מבניו עם $l' = \lfloor \frac{n-2}{2} \rfloor$ קודקודים אז מתקיים $H_{n-1} = H_{l'} + 1$ וכמובן $l - l' \leq 1$ ולכן לפי הנחת האינדוקציה $H_l - H_{l'} \leq 1$. מהצבה במשוואות המקוריות מתקבל הדרוש, כלומר קיבלנו שגדלי תת העצים נבדלים לכל היותר ב-1 ומכאן גבהי תת העצים נבדלים לכל היותר ב-1, ולכן מתקיימת תכונת AVL בכל קודקוד ולכן זהו עץ AVL חוקי.

הערה: קיימים מבני נתונים אחרים שנועדו לאזן עצים על מנת לאפשר חיפוש, הכנסה ומחיקה בזמן לוגריתמי. ביניהם נמצא עצים אדומים-שחורים, המקיימים:

- השורש הוא שחור
- כל עלה הוא שחור
- לכל קודקוד אדום, שני הבנים שחורים
- לכל קודקוד, כל המסלולים ממנו לכל עלה מכילים את אותו מספר של קודקודים שחורים

באמצעות הרעיון הזה (שממומש ע"י רוטציות בעיקר), מגיעים לגובה $2 \log(n+1)$ לכל היותר עבור עץ עם n קודקודים פנימיים. זהו חסם פחות הדוק מאשר על עץ AVL, אולם מצד שני התחזוקה הכרוכה בניהול המבנה קטנה יותר.

כמו כן, מבנים מורכבים יותר (וכלליים יותר) הם B-Trees, שבהם בכל קודקוד יש יותר מ-2 בנים, ותת העצים הנפרשים מהם מתחלקים ליותר מ-2 טווחים. כמו כן, קובעים לעץ דרגה מינימלית $t \geq 2$ כך שלכל קודקוד לא פחות מ- t בנים (חוץ מהעלים והשורש), ולא יותר מ- $2t$ בנים. המקרה הפשוט ביותר הוא $t = 2$, ואלה הם עצי 2-3-4. החסם כאן על גובה העץ הוא: $h \leq \log_t \left(\frac{n+1}{2} \right)$.

פרק 3 – דחיסת מידע

קוד הופמן

הבעיה שאליה אנו מתייחסים כעת היא בעיית דחיסת מידע – רוצים לשלוח את המידע בצורה החסכונית ביותר האפשרית (מבחינת ביטים) כדי שעדיין ניתן יהיה לשחזר אותו. דוגמה – אם רוצים לקודד את כל האותיות באנגלית (רק lower case), אז ניתן להסתפק ב-5 ביטים (שכן יש 26 אותיות, ו- $2^5 = 32 > 26$). קידוד כזה הוא קל מאוד בשני הכיוונים, והוא דורש בסה"כ טבלה הממפה לכל אות רצף קבוע של 5 ביטים.

הבעיה עם קידוד מסוג זה היא שלא כל האותיות הן בעלות אותה שכיחות למשל, האות q שכיחה הרבה פחות מהאות e, אולם אורך הקידוד שלהן זהה. נרצה להגיע למצב שלאותיות נפוצות יהיה קידוד קצר יותר מאותיות נדירות, כלומר שאורך הקידוד של אות יהיה תלוי בשכיחות שלה למשל, נוכל לתת את הקידוד הבא:

$$a \rightarrow 0, b \rightarrow 101, c \rightarrow 100, d \rightarrow 111, e \rightarrow 1101, f \rightarrow 1100$$

כאשר השכיחות של האותיות כאן הן:

$$a: 45, b: 13, c: 12, d: 16, e: 9, f: 5$$

קידוד זה נראה תקין, אולם השאלה המרכזית היא: האם קידוד זה ניתן לפענוח בצורה יחידה? אנחנו דורשים שאף תחילית שלקידוד של אות מסוימת אינה זהה לקידוד של אות אחרת, שכן אם זה יהיה המצב, לא נוכל לפענח

באופן חד משמעי. לדוגמה, אם $a \rightarrow 1$ בקידוד הנ"ל, אזי כשניתקל בפענוח ברצף 1101 לא נוכל לדעת האם יש לפענח אותו בתור ab או בתור e .

הגדרה: קידוד חסר תחיליות הוא קידוד שבו אין אף מילת קוד שהיא תחילית של מילת קוד אחרת

ניתן לתאר קידודים כאלה באמצעות עצים, כאשר 0 מתאר פנייה שמאלה, 1 מתאר פנייה ימינה, ובעלים של העץ נמצאות האותיות עצמן שאותן אנו מקודדים. על ידי מעבר על עץ כזה נוכל בקלות לפענח הודעות בלי להכיר את הטבלה המקורית המשייכת לכל אות קידוד מסוים.

הגדרה: עבור א"ב C עם פונקציית שכיחות $f: C \rightarrow (0,1]$ ואורך קידוד $l(c) \forall c \in C$, נגדיר את עלות הקידוד (או אורך הקידוד) עבור עץ T כ- $L(T) = \sum_{c \in C} f(c)l(c)$.

לדוגמה, במקרה שראינו קודם, אורך הקידוד של הקידוד הנאיבי הוא כמובן 3, היא הקידוד של כל אות הוא באורך 3 וסכום השכיחויות של כל האותיות הוא 1. לכן הסכום המשוקלל הוא פשוט 3. לעומת זאת, עבור הקוד הדחוס יותר שתיארנו קודם, מתקבל אורך קידוד 2.24, שהוא משמעותית יותר טוב. בהמשך נראה שאורך קידוד זה הוא האופטימלי עבור קוד חסר תחיליות.

כיצד בונים עץ קידוד כזה? אנו נתבונן באלגוריתם של הופמן, שעל שמו נקרא קוד הופמן. הרעיון הוא לסדר את האותיות לפי סדר שכיחויות עולה, ולקחת בכל פעם את שתי האותיות בעלות השכיחויות המינימליות ולאחד אותן לקודקוד אחד בעץ שהשכיחות שלו היא סכום השכיחויות של שתי האותיות אם ממשיכים כך עד שמקבלים קודקוד יחיד, מתקבל עץ הקידוד המבוקש.

HUFFMAN(C)

$n \leftarrow |C|$

$Q \leftarrow \text{BUILD MIN HEAP}(\{c \in C\})$

for $i \leftarrow 1$ to $n - 1$

 allocate a new node z

$\text{left}(z) \leftarrow \text{EXTRACT MIN}(Q)$

$\text{right}(z) \leftarrow \text{EXTRACT MIN}(Q)$

$\text{key}(z) \leftarrow \text{key}(\text{left}(z)) + \text{key}(\text{right}(z))$

$\text{INSERT}(Q, z)$

return $\text{EXTRACT MIN}(Q)$

על ידי שימוש בערימת מינימום, אנו מגיעים למצב שבניית העץ דורשת $n - 1$ איטרציות שבכל אחת מבצעים שתי פעולות הוצאה ופעולת הכנסה אחת מהערימה, וכן עוד עבודה במחיר קבוע, כלומר בסה"כ $O(n \log(n))$. כמו כן, בניית הערימה היא $O(n)$ ולכן זניחה.

טענה 23: בעץ בינארי מלא מספר הקודקודים הפנימיים קטן באחד ממספר העלים. (תזכורת: בעץ בינארי מלא לכל קודקוד יש 0 או 2 בנים.)

הוכחה: אינדוקציה על מספר האיברים בעץ, n . עבור $n = 1$ יש לנו עלה יחיד ואפס קודקודים פנימיים, כלומר הטענה מתקיימת. נניח שהטענה נכונה עבור עץ בגודל n , ונתבונן בעץ בגודל $n + 1$. נסמן $l, r > 0$ את מספר העלים בשני תתי העצים של השורש (קיימים כאלה כיוון ש- $n > 1$ והעץ מלא). כמו כן ברור שבתתי העצים יש פחות מ- n קודקודים, ולכן מהנחת האינדוקציה מספר הקודקודים הפנימיים בתת העצים הוא $l - 1, r - 1$ בהתאמה. שורש העץ כולו הוא כמובן קודקוד פנימי גם כך, ולכן קיבלנו $l + r + 1 = (l - 1) + (r - 1) + 1$ קודקודים פנימיים כאשר מספר העלים הוא $l + r$, כפי שרצינו. ■

הערה: צריך כאן $n - 1$ איטרציות כי אנחנו בונים עץ בינארי מלא, ומספר הקודקודים בעץ בינארי מלא עם n עלים הוא $n - 1$ (טענה 23).

משפט 2: קוד הופמן הוא קוד ללא תחיליות אופטימלי, כלומר עבור א"ב C אם T הוא העץ המיוצר ע"י אלגוריתם הופמן, אזי לכל T' אחר מתקיים $L(T) \leq L(T')$.

הוכחה: על מנת להוכיח את המשפט נזדקק למספר טענות עזר.

טענה 24א': אם T קידוד ללא תחיליות אופטימלי, אזי $\forall a, b \in C : f(a) < f(b) \Rightarrow l(a) \geq l(b)$.

הוכחה: בשלילה, אחרת נחליף בין האותיות ונקבל עץ עם עלות יותר קטנה, שכן אם העץ המוחלף הוא T' אזי מתקבל:

$$L(T) = \left[\sum_{c \in C \setminus \{a,b\}} f(c)l(c) \right] + f(a)l(a) + f(b)l(b)$$

$$L(T') = \left[\sum_{c \in C \setminus \{a,b\}} f(c)l(c) \right] + f(a)l(b) + f(b)l(a)$$

וכעת נרצה להגיע לכך ש- $L(T') < L(T)$ בסתירה לאופטימליות של T , אז נבצע רדוקציה של אי שוויונות

$$L(T') < L(T)$$

$$f(a)l(b) + f(b)l(a) < f(a)l(a) + f(b)l(b)$$

$$f(a)l(b) + f(b)l(a) - f(a)l(a) - f(b)l(b) < 0$$

$$(f(a) - f(b))(l(b) - l(a)) < 0$$

אבל אנו יודעים $f(a) - f(b) < 0$ לפי ההנחה, ו- $l(b) - l(a) > 0$ לפי ההנחה בשלילה, ולכן הביטוי אכן שלילי כלומר קיבלנו סתירה לאופטימליות של T . ■

טענה 24ב': אם א"ב $C : |C| \geq 2$ אזי קיים עץ אופטימלי T שבו שתי האותיות עם השכיחויות הכי נמוכות הן עלים בעץ ובנים לאותו קודקוד פנימי, ברמה הכי תחתונה בעץ.

הוכחה: נשים לב שעץ אופטימלי הוא עץ מלא (אחרת יש לנו מסלול שבו יש לקודקוד מסוים בן אחד, וניתן להוריד את הקודקוד הזה ולשפר את הקידוד, בסתירה לאופטימליות). מכאן, ברמה התחתונה של העץ יש לפחות שני קודקודים אחים (בנים לאותו קודקוד פנימי), כי העץ מלא. שני הקודקודים עם השכיחויות הנמוכות ביותר חייבים גם הם להיות ברמה התחתונה ביותר, לפי טענה 24א' (משיקולי אורך הקידוד שלהם). נותר רק להראות מדוע הם אכן אחים – אם הם לא אחים, ניתן להחליף בין קודקודים אחרים ברמה התחתונה כך שהם יהיו אחים, מבלי לפגוע באופטימליות של העץ כיוון שלא משנים את אורך הקידוד של אף אות. ■

טענה 24ג': יהי $|C| > 2$ א"ב, ו- T עץ שמקודד אותו. נניח שב- T שתי האותיות עם השכיחות הכי נמוכה $\{x, y\}$ הן אחים בעץ (וכמובן עלים). נגדיר T' להיות העץ שבו מקצצים את שני הקודקודים $\{x, y\}$ וזהו עץ שמקודד את C' שבו החלפנו את $\{x, y\}$ ב- z שהשכיחות שלה היא $f(z) \stackrel{\text{def}}{=} f(x) + f(y)$. אזי $L(T) = L(T') + f(z)$.

הוכחה:

$$\begin{aligned}
L(T) &= \left[\sum_{c \in C \setminus \{x,y\}} f(c)l(c) \right] + f(x)l_T(x) + f(y)l_T(y) \\
&= \left[\sum_{c \in C \setminus \{x,y\}} f(c)l(c) \right] + f(x)(l_{T'}(z) + 1) + f(y)(l_{T'}(z) + 1) \\
&= \left[\sum_{c \in C \setminus \{x,y\}} f(c)l(c) \right] + f(x)l_{T'}(z) + f(y)l_{T'}(z) + f(x) + f(y) = L(T') + f(z)
\end{aligned}$$

והמעבר האחרון נכון כי לפי ההגדרה $f(z) = f(x) + f(y)$ וכן לכל $c \in C \setminus \{x, y\}$ מתקיים $l_T(c) = l_{T'}(c)$. ■

נחזור להוכחת המשפט, ונוכיח אותו באינדוקציה על גודל הא"ב $|C|$. בסיס עבור $|C| = 2$ ברור כיוון שהקוד היחיד שאפשרי בכלל הוא 0 עבור אחת האותיות ו-1 עבור האות השנייה. נניח שהאלגוריתם בונה קוד אופטימלי עבור $|C| < k$ ונוכיח עבור k . נניח בשלילה שעבור $|C| = k$ קיים עץ אופטימלי $W : L(W) < L(T)$ כאשר T הוא קוד הופמן של C . כיוון ש- W הוא קוד אופטימלי, לפי טענה 24' ניתן להניח ששתי האותיות בעלות השכיחות הכי נמוכה $\{x, y\}$ הן אחים ברמה התחתונה ביותר של W (אחרת נוכל להחליף את W בעץ אופטימלי שקול W_n שבו זה מתקיים). לפי הבנייה באלגוריתם הופמן, אנו יודעים שגם ב- T שתי האותיות בעלות השכיחות הכי נמוכה הן אחים ברמה התחתונה ביותר של העץ. לכן נוכל להחיל על שני העצים T, W את טענה 24' ולקבל:

$$\begin{aligned}
L(W) &= L(W') + f(z) \\
L(T) &= L(T') + f(z)
\end{aligned}$$

לפי ההנחה בשלילה מתקיים גם $L(W') < L(T')$ אבל T' הוא עץ הופמן על א"ב בגודל $k-1$, ולכן זו סתירה להנחת האינדוקציה שלפיה T' הוא אופטימלי. ■

הערה: קוד הופמן אינו הקוד האופטימלי ביותר באופן כללי, אלא רק עבור עצים המייצגים קידוד חסר תחיליות. עם זאת, אפילו הקוד האופטימלי ביותר לא מנצח את קוד הופמן ביותר מביט אחד לכל אות

הערה: יהי n חזקה של 2, וכן יהיו שכיחויות $\{f_1, \dots, f_n\}$ שעבורן מתקיים $\max\{f_j\} \leq 2\min\{f_j\}$, אזי אלגוריתם הופמן מייצר קידוד באורך קבוע $\log(n)$. ניתן להתמודד עם בעיה זו ע"י בחירת קבוצת המכפלה של האותיות – כלומר במקום לקודד $\{a, b\}$ נקודד $\{aa, ab, ba, bb\}$. באופן כזה ניתן להגדיל את המרווחים בין השכיחויות ולקבל קידוד טוב יותר מקבוע

פרק 4 – טבלאות גיבוב

פונקציות גיבוב

אנו רוצים מבנה נתונים שמאפשר להחזיק נתונים בצורה של מילון – חיפוש, מחיקה והכנסה של מפתחות וערכים. הדבר החשוב ביותר לנו הוא שפעולת החיפוש תהיה מהירה ככל האפשר - $\mathcal{O}(1)$. הפעולות שנרצה אם כן:

- $INSERT(H, k)$ - הכנסה של מפתח (וערך) למבנה הנתונים
- $LOOKUP(H, k)$ - חיפוש של מפתח (וערך) במבנה הנתונים
- $DELETE(H, k)$ - מחיקה של מפתח (וערך) ממבנה הנתונים

הגישה הישירה ביותר היא להחזיק מערך שהאינדקסים בו הם המפתחות למשל, אם יש לנו 300 תלמידים שמספרי תעודות הזהות שלהם הם בין 0 ל-600, אזי נוכל להחזיק מערך בגודל 600 ואפילו לא נבזבז כל כך הרבה מקום. כמובן שכאשר עולם המפתחות מאוד גדול, או שהרווחים ביניהם גדולים, גישה זו לא עובדת טוב כל כך (למשל, אם נרצה לאכסן ת"ז אמיתיות – בנות 8-9 ספרות – של 300 תלמידים, לא נוכל להקצות בכל פעם מערך בגודל 10^9). מה שנרצה לעשות הוא להגדיר פונקציה המתאימה למפתחות מספרים קטנים יותר, שיוכלו לשמש כאינדקסים לתוך המערך.

הגדרה: יהיה U עולם המפתחות האפשריים, $|U| \ll m$ גודל הטבלה, ונגדיר פונקציית גיבוב $h: U \rightarrow \{1, \dots, m\}$ במציאות נתעניין גם ב- $S \subset U$ שהיא תת קבוצה של המפתחות שבאמת נשתמש בהם, ונסמן $|S| = N$.

הערה: לא בהכרח $m > N$, כלומר לפי עיקרון שובך היונים בהכרח יכולות להיות התנגשויות (הפונקציה h אינה חח"ע, כמובן רצוי שתהיה על).

כעת, אם המפתח שקיבלנו הוא k אזי נרצה להכניס אותו לאינדקס $h(k)$ במערך.

הגדרה: פונקציית גיבוב בשיטת החלוקה היא $h(k) = k \pmod m$.

כאן ברור בקלות שיכולות להיות התנגשויות, אם שני מפתחות k, k' בעלי אותה שארית חלוקה ב- m . כיצד נוכל לטפל בהתנגשויות? ובכן, נוכל לשמור בכל מקום במערך רשימה מקושרת של המפתחות $\{k_1, \dots, k_r\}$ כך ש- $h(k_1) = \dots = h(k_r)$ (טכניקה זו נקראת chaining, שרשור).

כמובן שעלויות הפעולות על המבנה הזה תלויה בעומס על המקומות השונים במערך – אם יש רשימה מקושרת אחת שכל המפתחות הגיעו אליה (למשל כי פונקציית הגיבוב גרועה), אזי כל הפעולות יהיו בזמן ליניארי כי צריך לחפש בתוך הרשימה המקושרת את האיבר המתאים (כדי להכניס, למחוק, לחפש).

אם כך, ה- wish list שלנו מבחינת טבלת הגיבוב הוא:

- אורך הרשימות יהיה קצר (פונקציית הגיבוב תפזר את המפתחות בצורה "אחידה")
- פונקציית הגיבוב קלה לחישוב ($O(1)$)
- מתקיים $m = O(N)$, כלומר רוצים שהמקום יהיה ליניארי

אם משתמשים בפונקציית גיבוב מסוימת, השאלה האם היא עובדת טוב או לא טוב תלויה גם במפתחות למשל, אם כך הקלטים מתחלקים ב- m ללא שארית, אזי שיטת החלוקה לא עובדת טוב למעשה, נרצה שפונקציית הגיבוב תיראה מבחוח כפונקצייה פסאודו-אקראית, כלומר שהפונקציה תיראה אקראית אך למעשה כמובן תהיה ניתנת לשחזור.

הגדרה: פונקציית גיבוב בשיטת ההכפלה היא $h(k) = \lfloor m(Ak - \lfloor Ak \rfloor) \rfloor$ עבור קבוע כלשהו A .

מקובל לבחור את $A = \frac{1+\sqrt{5}}{2}$ (יחס הזהב), אבל יש גם אפשרויות אחרות שעובדות

טענה 25: לכל $h: U \rightarrow \{1, \dots, m\}$ כך ש- $|U| > Nm$ קיימת קבוצת מקורות בגודל לפחות N שנשלחת לתא אחד בטבלה, ונגדיר $S \subset U: |S| = N$ להיות קבוצת N המפתחות הראשונים בה.

הוכחה: נסתכל עבור כל תא במערך על המקורות שלו ב- U . לתא מסוים קיימת קבוצת מקורות בגודל N וזוהי S . מדוע זה נכון? כי אם לא קיימת קבוצת מקורות בגודל N (אלא כולן קטנות מ- N), אזי יש לנו לכל היותר $(N-1)m$ איברים במערך אבל זה סותר את ההנחה ש- $|U| > Nm$. ■

הערה: נניח כעת שפונקציית הגיבוב מתאימה מפתח לתא מסוים בהסתברות אחידה על פני כל התאים. כלומר המקום שאליו הפונקציה תתאים את האיבר הבא אינה תלויה בהיסטוריה, וגם אחידה על פני כל התאים. הנחה זו נקראת Simple Uniform Hashing. תחת הנחה זו, תוחלת אורך הרשימה הוא $\frac{N}{m}$. (ר' בהמשך טענה 26)

הגדרה: פקטור העומס של טבלת גיבוב הוא $\alpha = \frac{N}{m}$ (הממוצע של מספר האיברים בכל שרשרת).

הערה: בגיבוב דינאמי (Dynamic Hashing) מגדילים את הטבלה, למשל פי 2, בכל פעם ש- α עובר סף מסוים.

הערה: בטבלת גיבוב המשתמשת בשרשור, זמן החיפוש הממוצע הוא $\Theta(1 + \alpha)$, תחת ההנחה של Simple Uniform Hashing. זה טריוויאלי כי ההסתברות להגיע לכל תא היא אחידה, ולכן מספר האיברים שנבחן בכל פעולה הוא α בממוצע, וכיוון שייתכן שהאיבר כבר קיים בטבלה נקבל בדיוק $\Theta(1 + \alpha)$.

קיימות שיטות רבות אחרות להתמודד עם התנגשויות (ללא צורך בשרשור, כאשר הנתונים נשמרים רק במערך – Open Addressing), נזכיר כמה מהן:

- **Linear Probing** – נגדיר את פונקציית הגיבוב $h(k, i) = (h'(k) + i) \bmod m$, כאשר כאן $i = 0, \dots, m-1$ ו- $h': U \rightarrow \{0, \dots, m-1\}$ היא פונקציית הגיבוב שהכרנו קודם. הרעיון הוא שקודם כל נבדוק את התא $h(k, 0)$ עבור מפתח k . אם הוא תפוס, ננסה את $h(k, 1)$ וכן הלאה. למעשה, אנחנו עוברים על התאים בצורה ציקלית עד שמשלימים מעגל עד $h(k, 0)$ ועוצרים. הבעיה כאן היא Clustering – די במהירות נבנים רצפים של תאים מלאים, מה שמגדיל את זמן החיפוש, שהרי ההסתברות שתא ריק יתמלא תלויה במספר התאים המלאים j לפניו, והיא $\frac{j+1}{m}$.
- **Quadratic Probing** – נגדיר את פונקציית הגיבוב $h(k, i) = (h'(k) + c_1 i + c_2 i^2) \bmod m$ כאשר כאן הכל כמו קודם ו- $c_1, c_2 \neq 0$. כאן אמנם אין Clustering חמור כמו קודם, אבל עדיין קיימת בעיית Clustering. כמו כן, רצף הדגימה (probing) עדיין תלוי רק ב- $h'(k)$, כך ששני מפתחות בעלי אותו $h'(k)$ יגרמו לדגימה של אותו הרצף.
- הערה:** בשני הפתרונות הנ"ל, כמובן נדרוש שהרצף $h(k, 0), \dots, h(k, m-1)$ (לכל k) יהיה תמורה של $0, \dots, m-1$ כדי להבטיח שעוברים על כל תא בטבלה.
- **Double Hashing** – נשתמש בשתי פונקציות גיבוב: $h(k, i) = (h_1(k) + i h_2(k)) \bmod m$. כאן פונקציית הגיבוב השנייה – שגם היא תלויה במפתח – קובעת את האופן שבו "נקפץ" בטבלה.

טענה 25א': גם עבור Linear Probing, Quadratic Probing ו-Double Hashing ניתן למצוא קבוצה $S \subset U$ בגודל N הנשלחת לתא אחד בטבלה. עבור השניים הראשונים די למצוא n איברים שעבורם $h'(k)$ זהה, וההוכחה זהה לחלוטין לטענה 25. במקרה של Double Hashing, נחלק את העולם ל- m^2 קבוצות: $A_{ij} = \{k \in U : h_1(k) = i, h_2(k) = j\}$.

כעת אם נניח $|U| > Nm^2$ אזי קיימת קבוצה עם לפחות N איברים, וכמובן ערכי שתי הפונקציות זהים לכל האיברים בקבוצה ולכן ההכנסה והחיפוש לאחר מכן יהיו בזמן ליניארי. ■

משפחה אוניברסלית של פונקציות גיבוב

הגדרה: $\mathcal{H} = \{h: U \rightarrow \{1, \dots, m\}\}$ היא משפחה אוניברסלית של פונקציות גיבוב אם לכל $x \neq y \in U$ מתקיים $\Pr_{h \in \mathcal{H}}(h(x) = h(y)) \leq \frac{1}{m}$.

הערה: המשמעות של ההגדרה היא שבהינתן $x \neq y \in U$, אם נעבור על כל פונקציות הגיבוב $h \in \mathcal{H}$ במשפחה, ההסתברות עבור כל אחת מהן לגרום להתנגשות $h(x) = h(y)$ היא $\frac{1}{m}$ לכל היותר.

טענה 26: אם $h \in_R \mathcal{H}$ (נבחרת אקראית) אזי לכל $x \in U$ תוחלת אורך הרשימה שבה x נמצא (מספר ההתנגשויות) היא $1 + \frac{N}{m}$ לכל היותר.

הוכחה: לכל x, y נגדיר משתנה מקרי $C_{xy} = \begin{cases} 1 & h(x) = h(y) \\ 0 & h(x) \neq h(y) \end{cases}$. אורך הרשימה שבה x נמצא הוא פשוט:

$$\sum_{y \in U} C_{xy} = 1 + \sum_{y \in U \setminus \{x\}} C_{xy} = 1 + \sum_{y \in U \setminus \{x\}} E(C_{xy}) = 1 + \sum_{y \in U \setminus \{x\}} \Pr(C_{xy} = 1) \leq 1 + \frac{N-1}{m} \leq 1 + \frac{N}{m}$$

■

מכאן בעצם ניתן להסיק שאם $m = \mathcal{O}(N)$ אזי התוחלת להתנגשות היא $\mathcal{O}(1)$, שזה "ממש בסדר".

דרך אחת לבנות משפחה אוניברסלית של פונקציות גיבוב היא באמצעות מטריצות מעל $\mathbb{Z}_2$. נניח $m = 2^r$ וכן $|U| = 2^l$, כמובן $l > r$. ניתן להסתכל על פונקציית גיבוב כאן כעל פונקציה המעבירה l ביטים של אינפורמציה ל- r ביטים של אינפורמציה.

נבחר פונקציית גיבוב ע"י בחירת מטריצה $M \in_R \mathbb{M}_{r \times l}(\mathbb{Z}_2)$. כעת אם נכתוב את המפתח $v \in U$ בתור וקטור עמודה של l ביטים, אזי כפל משמאל ב- M יספק לנו את וקטור העמודה $Mv \in \mathbb{Z}_2^r$ (שהוא ב- $\{0, \dots, m-1\}$).

המשפחה האוניברסלית $\mathcal{H}$ כאן היא פשוט אוסף כל המטריצות $\mathcal{H} = \mathbb{M}_{r \times l}(\mathbb{Z}_2)$.

טענה 27: $\mathcal{H}$ הנ"ל היא משפחה אוניברסלית של פונקציות גיבוב

הוכחה: אם $x \neq y$ אזי:

$$\Pr_{h \in \mathcal{H}}(h(x) = h(y)) = \Pr_{h \in \mathcal{H}}(h(x - y) = \vec{0}) = \Pr_{h \in \mathcal{H}}(h(z \neq 0) = \vec{0})$$

נסמן את העמודות $M = ((h_1), \dots, (h_l))$ ונכת:

$$Mz = \sum_{z_j=1} h_j = h_{j_0} + \sum_{z_j=1, j \neq j_0} h_j$$

כאשר כאן z_j זו הקואורדינטה ה- j של z , ויש לנו לפחות אחת כזו שהיא לא 0 כי $z \neq 0$. כעת אנו מפרקים את הסכום לחלק הראשון, שהוא וקטור העמודה הראשון כך ש- $z_{j_0} = 1$, וחלק השני שנקרא לו u . אם קיבענו את u , אזי כדי ש- $Mz = 0$ אנחנו צריכים $h_{j_0} = u$ (כיוון שאנחנו מעל $\mathbb{Z}_2$). h_{j_0} מוגרל באקראי ויש בו r קואורדינטות, ולכן ההסתברות לצירוף מסוים שלהן היא בדיוק $\frac{1}{2^r}$. כלומר $\frac{1}{m}$. מכאן ההסתברות להתנגשות על $h \in \mathcal{H}$ היא בדיוק $\frac{1}{m}$, כלומר קיבלנו משפחה אוניברסלית של פונקציות גיבוב. ■

נתבונן בדוגמה נוספת של משפחה אוניברסלית של פונקציות גיבוב נניח $m \ll |U| \leq p$ (כלומר בחרנו p גדול מספיק, ראשוני). כעת תחת $\mathbb{Z}_p^* = \{1, \dots, p-1\}$, $\mathbb{Z}_p = \{0, \dots, p-1\}$ נגדיר:

$$\forall a \in \mathbb{Z}_p^*, b \in \mathbb{Z}_p \quad h_{a,b}(k) = ((ak + b) \bmod p) \bmod m$$

ואת המשפחה האוניברסלית נגדיר: $\mathcal{H}_{p,m} = \{h_{a,b} : a \in \mathbb{Z}_p^*, b \in \mathbb{Z}_p\}$.

טענה 28: הקבוצה $\mathcal{H}_{p,m}$ היא משפחה אוניברסלית של פונקציות גיבוב

הוכחה: יש לנו $p-1$ אפשרויות לבחור את a ו- p אפשרויות לבחור את b , כלומר בסה"כ $p(p-1)$ פונקציות במשפחה. נבחר שני מפתחות $k \neq l \in \mathbb{Z}_p$. עבור כל $h_{a,b} \in \mathcal{H}_{p,m}$ מתקיים:

$$r = (ak + b) \bmod p, s = (al + b) \bmod p \Rightarrow r \neq s$$

כלומר, לפני שביצעו $\bmod m$ יש לנו התאמה חח"ע (כי $a \neq 0$ וכי בחרנו $|U| \geq p$ - זה חשוב). נשים לב גם שכל בחירת a, b נותנת r, s שונים עבור $k \neq l$, ולמעשה:

$$al + b = s, ak + b = r \Rightarrow b = (r - ak) \bmod p, a = \frac{r - s}{k - l} \bmod p$$

כיוון שיש $p(p-1)$ זוגות a, b וגם $p(p-1)$ זוגות r, s יש התאמה חח"ע ועל בין $a, b \rightarrow r, s$. אם כך, כאשר נתונים $k \neq l$ ונבחר a, b באופן אקראי, אזי r, s נבחרים גם בצורה אחידה מכל זוגות הערכים השונים מודולו p . נבדוק מה ההסתברות להתנגשות. נקבע את r , וקעת מספר הערכים s כך ש- $s \equiv r \bmod m$ הוא לכל $s \neq r$. $\left\lfloor \frac{p}{m} \right\rfloor - 1$ וכמובן: $\left\lfloor \frac{p}{m} \right\rfloor - 1 \leq \frac{p+m-1}{m} - 1 = \frac{p-1}{m}$. יש לנו $p-1$ בחירות שונות של $s \neq r$ ולכן ההסתברות להתנגשות בסה"כ היא $\frac{1}{m}(p-1) = \frac{p-1}{m}$. לכן קיבלנו משפחה אוניברסלית של פונקציות גיבוב ■

מה שראינו עבור משפחות אוניברסליות של פונקציות גיבוב זו תוחלת על מספר ההתנגשויות אבל עדיין לכל משפחה כזו ניתן למצוא קלט גרוע במיוחד שבו המבנה יתחיל לעבוד בזמן ליניארי! נרצה לבנות טבלת גיבוב שבה גם במקרה הגרוע ביותר החיפוש יהיה ב- $\mathcal{O}(1)$, ונעשה זאת ע"י תשלום מחיר כבד יותר של בניית הטבלה. כלומר, נשלם מחיר יקר כי הבנייה תהיה תלויה בקלט, אבל לאחר הבנייה החיפוש יהיה ב- $\mathcal{O}(1)$.

הגדרה: קבוצה $\mathcal{H} = \{h: U \rightarrow \{1, \dots, m\}\}$ היא משפחה c -אוניברסלית של פונקציות גיבוב אם:

$$\forall x \neq y \in U, \Pr_{h \in \mathcal{H}}(h(x) = h(y)) \leq \frac{c}{m}$$

טענה 29: לכל משפחה c -אוניברסלית של פונקציות גיבוב $\mathcal{H}$ מתקיים ש- $1 - \frac{m}{|U|} \leq c$.

הוכחה: ראשית נראה טענת עזר שימושית.

טענה 29א': עבור a_i אי-שליליים נגדיר $A = \sum_i a_i$ ואז $\sum_i (a_i)^2 \geq n \left(\frac{A}{n}\right)^2$.

הוכחה: נגדיר $v_i = 1$ לכל i , אזי לפי אי שוויון קושי-שוורץ:

$$\sum_i a_i = \sum_i a_i b_i \leq \sqrt{\sum_i (b_i)^2} \sqrt{\sum_i (a_i)^2}$$

ומכאן $A^2 = n \sum_i (a_i)^2$ כלומר $A = \sum_i a_i \leq \sqrt{n} \sqrt{\sum_i (a_i)^2}$. ■ כנדרש.

כעת נסמן את מספר האיברים המתמפים לתא ה- j בתור a_j ואת $A = |U| = \sum_i a_i$. כמו קודם, נסמן משתנה מקרי C_{xy} המייצג התנגשות אם קיימת כזו, וכמובן $\sum_j (a_j)^2 = \sum_{x \in U} \sum_{y \in U} C_{xy}$. לכן לפי טענה 29א', נקבל

$$\sum_{x \in U} \sum_{y \in U} C_{xy} \geq m \left(\frac{|U|}{m} \right)^2$$

ונחלץ מכאן את המקרה ש- $y = x$:

$$\begin{aligned} \sum_{x \in U} \left(\sum_{y \in U \setminus \{x\}} C_{xy} + C_{xx} \right) &= \sum_{x \in U} \sum_{y \in U \setminus \{x\}} C_{xy} + |U| \geq m \left(\frac{|U|}{m} \right)^2 \\ \sum_{x \in U} \sum_{y \in U \setminus \{x\}} C_{xy} &\geq \frac{|U|^2}{m} - |U| = |U| \left(\frac{|U|}{m} - 1 \right) \end{aligned}$$

נחיל את אי-השוויון הזה על כל ה- $h \in \mathcal{H}$:

$$\begin{aligned} \sum_{h \in \mathcal{H}} \sum_{x \in U} \sum_{y \in U \setminus \{x\}} C_{xy} &\geq |\mathcal{H}| |U| \left(\frac{|U|}{m} - 1 \right) \\ \sum_{x \in U} \sum_{y \in U \setminus \{x\}} \frac{1}{|\mathcal{H}|} \sum_{h \in \mathcal{H}} C_{xy} &\geq |U| \left(\frac{|U|}{m} - 1 \right) \end{aligned}$$

כעת נשים לב שיש לנו סכום של $|U|(|U| - 1)$ איברים (בגלל $x \in U, y \in U \setminus \{x\}$) ולכן לפחות אחד מהם צריך להיות גדול יותר מ- $\frac{|U| \left(\frac{|U|}{m} - 1 \right)}{|U|(|U| - 1)}$. במילים אחרות, קיימים x', y' שעבורם:

$$\frac{1}{|\mathcal{H}|} \sum_{h \in \mathcal{H}} C_{x' y'} \geq \frac{|U| \left(\frac{|U|}{m} - 1 \right)}{|U|(|U| - 1)} = \frac{1}{|U| - 1} \left(\frac{|U|}{m} - 1 \right)$$

כעת נשים לב שמה שאנו רוצים להוכיח הוא שההסתברות להתנגשות (אגף שמאל) היא גדולה או שווה ל- $\left(1 - \frac{m}{|U|}\right) \left(\frac{1}{m}\right)$. באי השוויון למעלה יש לנו:

$$\frac{1}{|U| - 1} \left(\frac{|U|}{m} - 1 \right) = \frac{1}{|U| - 1} \left(\frac{|U| - m}{m} \right)$$

לעומת זאת מה שאנחנו מחפשים הוא $\left(1 - \frac{m}{|U|}\right) \left(\frac{1}{m}\right) = \frac{|U| - m}{|U|} \left(\frac{1}{m}\right)$. לאחר צמצום של $|U| - m$ ושל $\frac{1}{m}$ אנו נותרים עם אי-השוויון:

$$\frac{1}{|U| - 1} \geq \frac{1}{|U|}$$

שהוא בוודאי נכון, ולכן גם הטענה נכונה. ■

המשמעות של טענה 29 היא ששיפור הנטייה לא להתנגש יחסית ל- $\frac{1}{m}$ הוא קטן מאוד, והוא לכל היותר יהיה $\left(1 - \frac{m}{|U|}\right) \left(\frac{1}{m}\right)$. כלומר, זה לא מאוד מעניין להסתכל על משפחות כאלה.

גיבוב מושלם – Perfect Hashing

נתחיל מגיבוב מושלם במקום ריבועי, כלומר נאפשר $m = O(N^2)$. אנו מסתכלים על מצב סטטי שבו המפתחות נתונים ואנו רוצים למצוא את פונקציית הגיבוב המתאימה.

נבחר $2N^2 > m \geq N^2$ שיש עבורו משפחה אוניברסלית $\mathcal{H}$ של פונקציות גיבוב $U \rightarrow \{1, \dots, m\}$. נבחר באקראי $h \in_R \mathcal{H}$ ונבדוק האם יש התנגשות עבור אחד הזוגות $x \neq y$ ב- $S \subset U$ (קבוצת המפתחות האמיתיים שלנו). נגיד ש- h "טובה" אם לאף אחד מהזוגות אין התנגשות.

טענה 30: ההסתברות ש- h טובה גדולה מ- $\frac{1}{2}$.

הוכחה: בעולם המפתחות שלנו S יש לנו $\binom{N}{2}$ זוגות שונים, כלומר $\frac{N(N-1)}{2}$. עבור כל זוג כזה, לכל היותר $\frac{1}{m}$ מהפונקציות ב- $\mathcal{H}$ גורמות להתנגשות עבורו (כי $\mathcal{H}$ משפחה אוניברסלית של פונקציות גיבוב). אפילו אם עבור כל זוג יש $\frac{1}{m}$ מהפונקציות שגורמות להתנגשות, עדיין יש לנו לכל היותר $\frac{N(N-1)}{2} \left(\frac{1}{m}\right)$ פונקציות "רעות" עבור קלט כלשהו. בגלל ההנחות שלנו על m קיבלנו:

$$\Pr_{h \in \mathcal{H}} (\exists x \neq y: h(x) = h(y)) \leq \frac{N(N-1)}{2N^2} < \frac{1}{2}$$

■

כלומר, בהסתברות גדולה מ- $\frac{1}{2}$ בחרנו פונקציית גיבוב שאין בה התנגשות כלל עבור אוסף המפתחות הנתון (לאף זוג מפתחות). אם מצאנו כזו, סיימנו למעשה; אם לא, נמשיך לחפש פונקציה (בכל פעם בהסתברות $\frac{1}{2}$ נמצא אותה) שאין בה התנגשויות כלל. תוחלת מספר הניסיונות שלנו היא 2, שזה ממש לא הרבה, ולכן גם זמן הבנייה הוא בסה"כ $O(n)$ בממוצע, כיוון שהבדיקה האם יש התנגשויות או לא לוקחת $O(n)$.

מה שעשינו בינתיים היה תחת ההנחה $m = O(N^2)$. נרצה לבצע גיבוב מושלם במקום ליניארי – כלומר להתיר רק $m = O(N)$ כמו קודם. נבצע זאת בשני שלבים:

1. נגדיל באקראי $h \in_R \mathcal{H}$ כאשר $m = N$ לצורך הפשטות.
2. נפעיל את h על N האיברים בקלט, ולכל תא j נתאים את מספר האיברים $n_j = \#\{k: h(k) = j\}$. לכל תא j נתאים טבלת גיבוב משנית בגודל $(n_j)^2$ ובכל אחד מהתאים נשתמש בגיבוב מושלם במקום ריבועי (כמו קודם) עד שלא יהיו יותר התנגשויות.

ברור שבסופו של דבר מתקבל מבנה שבו החיפוש הוא ב- $O(1)$, אבל השאלה היא כמה מקום הוא תופס. נסמן $S(N)$ את המקום שהמבנה תופס, ויש לנו $N = \sum_j n_j$ ולכן:

$$S(N) = N + \sum_{j=1}^N n_j^2$$

טענה 31: $E(S(N)) = O(N)$.

הוכחה: נחשב את התוחלת של הביטוי השני ב- $S(N)$. נגדיר כמו קודם C_{xy} משתנה מקרי המקבל 1 אם יש התנגשות $h(x) = h(y)$ ו-0 אחרת. נתבונן ב- $E(\sum_x \sum_y C_{xy})$:

$$\sum_x \sum_y C_{xy} = \sum_{j=1}^N n_j^2$$

שהרי מספר הזוגות הסדורים שמתנגשים שווה לסכום מספרי הזוגות הסדורים שמתנגשים בתא מסוים מדוע אם כן זה n_j^2 ? כי עבור כל תא, יש לנו n_j איברים המתנגשים בתא זה. עבור כל איבר שמתנגש, מותר לנו לבחור את כל האיברים האחרים שמתנגשים ב- j כולל אותו עצמו, ויש לנו $n_j n_j = n_j^2$ זוגות כאלה.

עכשיו נותר לנו לחשב את התוחלת:

$$E\left(\sum_x \sum_y C_{xy}\right) = E\left(\sum_x C_{xx}\right) + \sum_x \sum_{y \neq x} C_{xy} \leq N + \sum_x \sum_{y \neq x} \frac{1}{m} = N + \frac{N(N-1)}{m} < 2N$$

כלומר קיבלנו שתוחלת המקום היא אכן ליניארית. ■

ניתן להשתמש באי-שוויון מרקוב (טענה 10) כדי לקבל ש- $\Pr(\sum_{j=1}^N n_j^2 > 4N) < \frac{1}{2}$ למשל. מכאן מגיעים לגרסה הסופית של גיבוב מושלם במקום ליניארי:

1. נבחר את ה- h הראשית ונחשב את $\sum_{j=1}^N n_j^2$. אם הוא קטן מ- $4N$ (וזוה קורה בהסתברות הגדולה מחצי) אזי נמשיך לשלב 2. אם לא, נבצע את הפעולה שוב (וכמו קודם, תוחלת מספר הניסיונות היא 2).
2. עבור כל טבלת גיבוב משנית נשתמש בשיטת הגיבוב המושלם במקום ריבועי, ושוב כאן ההסתברות גדולה מחצי שנצליח למצוא פונקציית גיבוב משנית שלא גורמת לאף התנגשות כך מוצאים את $h_1, \dots, h_m$ פונקציות הגיבוב המשניות וזמן הגישה לכל המפתחות הוא $\mathcal{O}(1)$ - ראשית מפעילים את $h(k) = j$ ואז את $h_j(k)$ כדי למצוא את המקום בטבלה המשנית.

טענה 32: תוחלת זמן הבנייה של טבלת גיבוב בשיטת הגיבוב המושלם במקום ליניארי היא ליניארית

הוכחה: נתבונן קודם בשלב הראשון של האלגוריתם. בשלב הראשון אנו מגרילים פונקציית גיבוב והיא מספקת לנו $\sum_{j=1}^N n_j^2$ הקטן מ- $4N$ בהסתברות גדולה מ- $\frac{1}{2}$. בכל ניסיון אנו מבצעים $\mathcal{O}(N)$ עבודה, ותוחלת מספר הניסיונות היא 2, ולכן בסה"כ בשלב הראשון תוחלת העבודה היא $\mathcal{O}(N)$.

גם בכל מערך משני יש לנו תהליך המצליח בהסתברות הגדולה מ- $\frac{1}{2}$, ולכן תוחלת מספר הניסיונות היא 2, ומליניאריות התוחלת, תוחלת העבודה בכל הטבלאות המשניות שווה לסכום התוחלות בכל אחת מהטבלאות ובסה"כ היא $\mathcal{O}(N)$.

כיוון שיש לנו שני תהליכים שהתוחלת שלהם היא ליניארית, גם ביצוע שניהם יחד נותן תוחלת זמן ליניארית. ■

פרק 5 – גרפים

הגדרות וייצוג

הגדרה: גרף $G = (V, E)$ הוא אוסף של קודקודים V וקשתות (או צלעות) E . נאמר על קשת $e = (v, w) \in E$ שהיא מחברת את הקודקודים v, w . לעתים נסמן $|V| = n, |E| = m$.

ראינו כבר גרפים, למשל עצים הם מקרה פרטי של גרפים. הקישורים מאתר לאתר באינטרנט הם דוגמה לגרף, ויש כמובן עוד הרבה דוגמאות.

בגרף מכוון לצלעות יש כיוון. בגרף לא מכוון לצלעות אין כיוון (כלומר, ניתן להסתכל על גרף לא מכוון כעל גרף מכוון שבו לכל צלע (v, w) יש גם צלע (w, v)).

הגדרה: מעגל בגרף (ביחס לקודקוד x) הוא מסלול $e_1, \dots, e_k \in E$ כך ש- $e_1 = (x, v_1)$ ו- $e_k = (v_k, x)$, או במילים אחרות $x \rightarrow v_1 \rightarrow \dots \rightarrow v_k \rightarrow x$.

ייצוג של גרף באמצעות מטריצת שכנויות, שהיא $Adj(G) \in \mathbb{M}_{n \times n}(\mathbb{Z}_2)$ הוא פשוט:

$$(Adj(G))_{ij} = \begin{cases} 1 & \exists e = (v_i, v_j) \in E \\ 0 & \text{otherwise} \end{cases}$$

הערה: אם הגרף לא מכוון, מטריצת השכנויות שלו היא מטריצה סימטרית ($Adj(G) = Adj^t(G)$).

המקום הנדרש עבור ייצוג זה הוא $\mathcal{O}(n^2)$, ועבור גרף דליל במיוחד (שבו הרבה קודקודים אבל מעט קשתות) זה ייצוג בזבזני למדי.

ייצוג של גרף באמצעות רשימת שכנויות הוא פשוט מערך בגודל n של רשימות מקושרות. בתא ה- j מופיעה רשימה מקושרת שמכילה את השכנים של הקודקוד v_j . במקרה זה המחיר מבחינת מקום הוא $\mathcal{O}(n + m)$.

כלל אצבע להחלטה על ייצוג:

- אם $m = \mathcal{O}(n)$ אזי הגרף דליל וכדאי להשתמש ברשימת שכנויות
- אם $m = \Omega(n^2)$ אזי הגרף צפוף וכדאי להשתמש במטריצת שכנויות.

הגדרה: רכיב קשירות (Connected Component) בגרף לא מכוון היא קבוצת קודקודים מקסימלית בגודלה שבה בין כל שני קודקודים קיים מסלול.

הגדרה: גרף קשיר הוא גרף בעל רכיב קשירות אחד.

הגדרה: רכיב קשירות חזק (Strongly Connected Component) בגרף מכוון היא קבוצת קודקודים מקסימלית בגודלה שבה בין כל שני קודקודים קיים מסלול (מכוון). כלומר לכל זוג קודקודים v, w קיים מסלול מ- v ל- w וגם מסלול מ- w ל- v .

טענה 33: מספר הצלעות המינימלי בגרף עם n קודקודים ו- k רכיבי קשירות הוא $n - k$.

הוכחה: בגרף עם n קודקודים וללא צלעות יש n רכיבי קשירות. בכל פעם שנוסיף צלע בין שני רכיבי קשירות, נצמצם את מספר רכיבי הקשירות ב- 1. לכן לאחר $n - k$ צעדים נוכל להגיע ל- k רכיבי קשירות. נשים לב שכאשר מוסיפים צלעות, אם מוסיפים צלע באותו רכיב קשירות אזי לא משנים את מספר רכיבי הקשירות ולכן צריך לפחות את $n - k$ הצלעות הנ"ל כדי להגיע ל- k רכיבי קשירות, וזה המספר המינימלי. ■

טענה 34: מספר הצלעות המקסימלי בגרף עם n קודקודים ו- k רכיבי קשירות הוא $\frac{(n-k+1)(n-k)}{2}$.

הוכחה: יש לנו k רכיבי קשירות, וכמובן בכל אחד מהם נרצה לשים את מספר הצלעות המקסימלי. ב- k רכיבי קשירות עם m קודקודים נוכל לשים $\frac{m(m-1)}{2}$ צלעות. נרצה לטעון שמספר הצלעות המקסימלי מתקבל כאשר יש

רכיב קשירות עם $n - k + 1$ קודקודים (שביניהם $\frac{(n-k+1)(n-k)}{2}$ צלעות) ועוד $k - 1$ רכיבי קשירות עם קודקוד אחד בלבד וללא צלעות.

נניח שנבחר סידור אחר, שבו יש שני רכיבי קשירות בעלי $a < b$ קודקודים כל אחד. אז אם ניקח קודקוד מהרכיב הקטן יותר ונעביר אותו לרכיב הגדול יותר, הורדנו מהרכיב הקטן יותר $a - 1$ צלעות אבל הרווחנו b צלעות חדשות, וכמובן $b > a - 1 \Rightarrow a < b$. מכאן מוטב לנו לעבוד עם רכיב קשירות אחד עם $n - k + 1$ קודקודים ועוד $k - 1$ רכיבי קשירות בעלי קודקוד אחד בלבד. ■

Depth First Search

השאלה המעניינת ביותר שאפשר לשאול על גרף היא האם קיים מסלול בין שתי נקודות בו (למשל – חיפוש יציאה ממבוך). נכתוב אלגוריתם המבקר בכל הקודקודים אליהם ניתן להגיע מקודקוד מסוים

EXPLORE(v)

$visited(v) \leftarrow 1$

$\forall w: (v, w) \in E$

if $visited(w) = 0$ then **EXPLORE**(w)

טענה 35: האלגוריתם **EXPLORE** מגיע לכל הקודקודים בגרף אליהם ניתן להגיע מהקודקוד v .

הוכחה: נניח שקיים מסלול $z \rightarrow v_k \rightarrow \dots \rightarrow v_1 \rightarrow v$. אם אכן **EXPLORE** לא הגיע לקודקוד z אז קיים קודקוד אחרון במסלול v_j כך שהגענו אליו אבל לא המשכנו ל- v_{j+1} . זה כמובן לא ייתכן, כי באלגוריתם אנו סורקים את כל השכנים של v_j וביניהם גם את v_{j+1} . ■

שיטת חיפוש זו נקראת Depth First Search (DFS) מאחר שסורקים את כל הקודקודים לעומק לפני שחוזרים וממשיכים לשכנים אחרים של קודקוד מסוים. כדי להרחיב את החיפוש בגרף לכל הקודקודים (ולא רק v שבחרנו כאן), נרצה פשוט לעשות לולאה על כל הקודקודים בגרף (שלא ביקרנו בהם עדיין):

DFS(G)

$\forall v \in V, visited(v) \leftarrow 0$

$\forall v \in V, \text{if } visited(v) = 0 \text{ then } \text{EXPLORE}(v)$

בכל ביצוע של **EXPLORE** נוצר עץ המתאר את הצלעות שעליהן עברנו. כמובן קיימות גם צלעות שלא עברנו עליהן – שהן חלק מהגרף אבל לא חלק מהעץ – למשל אם קיים בו מעגל. לצלעות שהאלגוריתם עבר עליהן נקרא $tree\ edges$ והצלעות האחרות הן "כל השאר".

כיוון שבביצוע **DFS** על גרף לא קשיר יש מספר עצים נפרדים של **EXPLORE**, ניתן לדבר על "חורשת ה-DFS" כעל אוסף העצים שנבנו במהלך ביצוע האלגוריתם.

סיבוכיות זמן הריצה של האלגוריתם היא $\mathcal{O}(|V| + |E|)$, וזאת מאחר שאנו צריכים להסתכל על כל הקודקודים (כל אחד פעם אחת), אך לכל אחד מהם אנו מסתכלים על כל הצלעות שיוצאות ממנו. כיוון שהגרף לא מכוון, מסתכלים על כל צלע פעמיים, אך בכל אופן הסיבוכיות ליניארית במספר הקודקודים והקשתות.

על מנת שיהיה לנו יותר מקום להרחבת האלגוריתם, נכניס ב-**EXPLORE** שינוי קל המאפשר לנו לבצע פעולות מסוימות ברגע ש"נכנסים" לקודקוד מסוים וברגע ש"יוצאים" ממנו (לאחר שביקרנו בכל השכנים שלו):

EXPLORE'(v)

$visited(v) \leftarrow 1$

$PREVISIT(v)$

$\forall w: (v, w) \in E \text{ if } visited(w) = 0 \text{ then } EXPLORE'(w)$

$POSTVISIT(v)$

מציאת רכיבי קשירות

כעת נרצה לכתוב אלגוריתם המוצא את ה- Connected Components של הגרף, כלומר מתאים לכל קודקוד את מספר רכיב הקשירות שהוא שייך אליו. היכולת הזאת חשובה, כי למשל יש מסלול בין שני קודקודים בגרף אם הם שניהם שייכים לאותו רכיב קשירות.

נחזיק מונה בשם $ccNum$ ונקדם אותו ב-1 בכל פעם שמתוך DFS קוראים ל- $EXPLORE$. ב- $PREVISIT$ נוכל לעדכן את מספר ה- CC של הקודקוד הנוכחי לערך הנוכחי של $ccNum$.

$CC(G)$

$ccNum \leftarrow 0$

$DFS(G)$

$PREVISIT(v)$

$cc(v) \leftarrow ccNum$

כדי לזהות רכיבי קשירות (חזקים) בגרף מכוון, אנחנו צריכים לעבוד הרבה יותר קשה – המבנה הקישורי של הגרף יותר מסובך. נעשה זאת בהמשך, אבל קודם:

הגדרה: מעגל בגרף הוא קבוצת קודקודים $\{v_1, \dots, v_k\}$ כך שקיימות הצלעות $\{(v_1, v_2), \dots, (v_{k-1}, v_k), (v_k, v_1)\}$.

הגדרה: גרף מכוון חסר מעגלים (Directed Acyclic Graph = DAG) הוא גרף מכוון שאין בו מעגלים.

הערה: אם נבנה את גרף רכיבי הקשירות החזקים בגרף מכוון, נקבל DAG. אחרת, אם בגרף רכיבי הקשירות החזקים יש מעגל, אזי יש שני רכיבים קשירות חזקים שבהם בעצם רכיב קשירות חזק אחד.

מיון טופולוגי

הגדרה: מיון טופולוגי של גרף מכוון $G = (V, E)$ הוא סידור של הקודקודים כך שלכל צלע (u, v) מתקיים ש- u נמצא לפני v בסידור.

הערה: מיון טופולוגי הוא דבר שימושי, למשל נניח שיש לנו תלויות בין משימות שמיוצגות כ-DAG, אזי אנו רוצים למצוא את הסדר שלפיו צריך לבצע את המשימות – וזהו המיון הטופולוגי.

טענה 36: בגרף מכוון שיש בו מעגל לא קיים מיון טופולוגי:

הוכחה: נניח שקיים מעגל $v_0 \rightarrow v_1 \rightarrow \dots \rightarrow v_k \rightarrow v_0$ בגרף, וקיים גם מיון טופולוגי. עבור v_0 , כל הקודקודים במעגל מופיעים אחריו במיון הטופולוגי כי יש צלעות מכוונות במסלול בפרט, גם v_k מופיע אחרי v_0 וזו סתירה לקיום הצלע (v_k, v_0) שכן אם קיים מיון טופולוגי אזי v_0 צריך להופיע אחרי v_k ולא לפניו. ■

כאשר מריצים DFS על גרף מכוון, נקבל ארבעה סוגי צלעות בכל עץ שנבנה במהלך ה- DFS :

- צלעות בעץ (tree edges), שעליהן האלגוריתם עבר
- צלעות קדימה (forward edges), שהאלגוריתם לא עבר עליהן אבל הן צלעות המחברות קודקוד במעלה העץ לקודקוד במורד העץ, כאשר הקודקוד במעלה העץ הוא אב קדמון של הקודקוד במורד העץ

- צלעות אחורה (backward edges), שהאלגוריתם לא עבר עליהן אבל הן צלעות המחברות קודקוד במורד העץ לקודקוד במעלה העץ, כאשר הקודקוד במעלה העץ הוא אב קדמון של הקודקוד במורד העץ
- צלעות חוצות (cross edges), שהאלגוריתם לא עבר עליהן אבל הן צלעות המחברות קודקודים שאף אחד מהם אינו אב קדמון של השני (למשל, שני קודקודים שהם בנים של שורש העץ)

טענה 37: בגרף מכוון יש מעגל אם"ם יש בו צלעות אחורה (backward).

הוכחה: אם (u, v) צלע אחורה, אזי יש מעגל המורכב מצלע זו יחד עם המסלול $u \rightarrow v$ בעץ החיפוש. אם בגרף יש מעגל $v_0 \rightarrow v_1 \rightarrow \dots \rightarrow v_k \rightarrow v_0$ אזי יהי v_i הקודקוד הראשון שהתגלה בחיפוש, כל שאר ה- $v_j: i \neq j$ כמובן יהיו צאצאים שלו בעץ החיפוש כיוון שאפשר להגיע ממנו אליהם. אבל, הצלע (v_{i-1}, v_i) (או (v_k, v_0) אם $i = 0$) היא צלע המובילה מצאצא לאב קדמון, ולכן היא צלע אחורה בעץ. ■

נשתמש ב"חותמות זמן" (timestamps) במהלך ביצוע המעבר על הגרף. בכל פעם שניכנס לקודקוד נרשום את הזמן הנוכחי, ובכל פעם שנצא מקודקוד נרשום את הזמן הנוכחי. כך עבור כל קודקוד יהיו לנו שתי נקודות "זמן" – מתי נכנסנו אליו ומתי יצאנו ממנו:

PREVISIT(v)

$pre(v) \leftarrow clock$
 $increment\ clock$

POSTVISIT(v)

$post(v) \leftarrow clock$
 $increment\ clock$

ניתן לקבוע את היחס בין שני קודקודים בעץ ה-DFS ע"פ התבוננות בערכי ה- $pre, post$ שלהם. כך למשל, אם בעץ ה-DFS הקודקוד v הוא אב קדמון של הקודקוד u אזי מתקיים $pre(v) < pre(u) < post(u) < post(v)$. באופן כללי עבור כל שני קודקודים u, v מתקיים אחד מהתנאים הבאים:

$$\begin{aligned} pre(v) < pre(u) < post(u) < post(v) \\ pre(u) < pre(v) < post(v) < post(u) \\ pre(u) < post(u), pre(v) < post(v), post(u) < pre(v) \\ pre(u) < post(u), pre(v) < post(v), post(v) < pre(u) \end{aligned}$$

במלים אחרות, האינטרבליים $[pre(u), post(u)]$, $[pre(v), post(v)]$ יכולים להיות מוכלים זה בזה (ממש) או זרים לחלוטין. לפי ערכי ה- $pre, post$ נוכל לקבוע את סוג הצלע בעץ ה-DFS. נניח ש- $(u, v) \in E$, אזי:

$$\begin{aligned} pre(u) < pre(v) < post(v) < post(u) &\Rightarrow (u, v) \text{ is tree or forward edge} \\ pre(v) < pre(u) < post(u) < post(v) &\Rightarrow (u, v) \text{ is backward edge} \\ [pre(v), post(v)] \cap [pre(u), post(u)] = \emptyset &\Rightarrow (u, v) \text{ is cross edge} \end{aligned}$$

טענה 38: אם G הוא DAG, סידור הקודקודים ע"פ סדר יורד של ערכי ה- $post$ שלהם מהווה מיון טופולוגי.

הוכחה: נניח שיש צלע (u, v) . אם הגענו ל- u קודם, אז הסיום ב- v יהיה לפני הסיום ב- u (כלומר $post(u) > post(v)$). אם הגענו ל- v לפני שהגענו ל- u , אזי כיוון שאנחנו ב-DAG אנחנו נסיים קודם את ה-EXPLORE של v עוד לפני שנתחיל את ה-EXPLORE של u , כלומר שוב $post(u) > post(v)$. ■

הערה: סיבוכיות זמן הריצה של אלגוריתם זה היא פשוט הסיבוכיות של DFS, כלומר $\mathcal{O}(|E| + |V|)$.

הגדרה: קודקוד אשר אין לו צלעות יוצאות נקרא כיור.

הגדרה: קודקוד אשר אין לו צלעות נכנסות נקרא מקור.

הגדרה: ב-DAG, הקודקוד בעל ערך ה- $post$ המינימלי (שהוא האחרון במיון הטופולוגי) הוא כיור.

הגדרה: ב-DAG, הקודקוד בעל ערך ה- $post$ המקסימלי (שהוא הראשון במיון הטופולוגי) הוא מקור.

הערה: בכל DAG, יש לפחות כיור אחד ולפחות מקור אחד.

אלגוריתם חלופי למיון טופולוגי: נמצא מקור בגרף, נדפיס אותו, ונמחק אותו מהגרף. נחזור על התהליך עד שהגרף מתרוקן. ניתן לעשות זאת אם נשמור עבור כל קודקוד את מספר הצלעות הנכנסות אליו (in degree), ובכל פעם שמוחקים קודקוד נעדכן את ה- $in\ degree$ של כל השכנים שלו. אם שומרים את הקודקודים בתור קדימויות (ערימה) ממיון לפי ה- $in\ degree$, אזי הסיבוכיות היא $O(|E| \log(|V|) + |V| \log(|V|))$.

מציאת רכיבי קשירות חזקים

אנו יודעים כבר שגרף רכיבי הקשירות החזקים (ה-SCC) הוא DAG. אנחנו רוצים להתחיל את ה-DFS ממקום שממנו אי אפשר "לזלוג" לרכיבי קשירות חזקים אחרים בגרף. ב-DAG רכיבי הקשירות, אנו רוצים להתחיל מהכיור! אם נתחיל מהכיור, נעבור על כולו (והיה לנו רכיב קשירות חזק אחד), ואז נעבור לכיור הבא וכו', מובטח שלא נזלוג לרכיבי קשירות חזקים אחרים ונקבל באופן מדויק את גרף רכיבי הקשירות

עכשיו הבעיה היא למצוא קודקוד שנמצא בכיור של DAG ה-SCC. נתחיל בלמצוא דווקא קודקוד במקור של DAG ה-SCC.

טענה 39: אם יש צלע מרכיב קשירות חזק C לרכיב קשירות חזק C' אזי ערך ה- $post$ המקסימלי ב- C יהיה גדול מערך ה- $post$ המקסימלי ב- C' .

הוכחה: ר' הוכחת טענה 38. אם הגענו ל- C קודם אז נגיע ל- C' לפני שנסיים עם C , ואם הגענו ל- C' קודם אז כיוון שגרף ה-SCC הוא DAG, נסיים עם C' לפני שנגיע ל- C . ■

אם כך, כדי למצוא קודקוד ברכיב קשירות שהוא המקור של גרף ה-SCC, פשוט נבחר את הקודקוד עם ה- $post$ המקסימלי (הוא בוודאי שייך ל-SCC שהוא המקור בגרף ה-SCC).

הערה: הקודקוד בעל ה- $post$ המינימלי אינו בהכרח קודקוד בכיור של הגרף.

עכשיו נבצע טריק: אם נהפוך את הצלעות של הגרף (כל צלע (u, v) הופכת לצלע $((v, u))$ אזי הקודקוד בעל ה- $post$ המקסימלי נמצא בדיוק בכיור של גרף ה-SCC!

$SCC(G)$

$DFS(G^R)$

$CC(G \text{ sorted by post descending})$

נעבור לשאלה אחרת לגמרי – רוצים לדעת מה המרחק הקצר ביותר בין שני קודקודים, למשל מי הקודקודים הקרובים ביותר לקודקוד מסוים. כאן DFS לא כל כך מתאים כיוון שהוא לא מגלה את המסלולים הקצרים ביותר.

אלגוריתם למציאת גרף רכיבי הקשירות החזקים G^{SCC} של גרף G :

1. נריץ את האלגוריתם SCC ונקבל כפלט מערך שנקרא scc ובו עבור כל קודקוד נשמר ה- SCC שלו. הערכים במערך הם בטווח $[1, |V|]$ מספרים שלמים.
 2. נמיין את המערך הנ"ל באמצעות $COUNTING SORT$ בזמן $O(|V|)$. נעבור על המערך הממוין ובכל פעם שנפגוש ערך השונה מהקודם לו, נוסיף אותו ל- V^{SCC} (רשימת הקודקודים של גרף G^{SCC}).
 3. נבנה את הקבוצה $S = \{(x, y) : \exists (u, v) \in E, x = scc[u], y = scc[v], x \neq y\}$, וזה לוקח $\Theta(|E|)$.
 4. נמיין את איברי S על פני כל רכיב קשירות נפרד באמצעות $COUNTING SORT$ בזמן $O(|V| + |E|)$.
 5. נעבור על הקבוצה הממויינת S , ובכל פעם שמוצאים איבר (x, y) (צלע) השונה מהקודם לו, נוסיף את הצלע (x, y) ל- E^{SCC} (אוסף הצלעות של G^{SCC}).
- זמן הריצה הכולל של האלגוריתם הוא $O(|V| + |E|)$.

Breadth First Search

נשתמש בתור במקום במחסנית, ונחזיק את המרחק של כל קודקוד מנקודת התחלה כלשהי:

```

BFS( $G, s$ )
 $\forall v \in V, dist(v) \leftarrow \infty$ 
 $dist(s) \leftarrow 0, prev(s) \leftarrow NIL$ 
 $Q \leftarrow BUILD\ QUEUE(\{s\})$ 
while not EMPTY( $Q$ )
     $u \leftarrow DEQUEUE(Q)$ 
     $\forall w: (u, w) \in E, if\ dist(w) = \infty\ then$ 
         $ENQUEUE(Q, w)$ 
         $prev(w) \leftarrow u$ 
         $dist(w) \leftarrow dist(u) + 1$ 

```

זמן הריצה של האלגוריתם הוא כמובן כמו של DFS , כי אנחנו עוברים על כל צלע פעמיים וכל קודקוד נבדק פעם אחת, כלומר זמן הריצה הוא $O(|V| + |E|)$.

טענה 40: (נכונות האלגוריתם) לאחר הוצאת k קודקודים מהתור, קיים $d \geq 0$ כך ש:

- כל הקודקודים שהוצאו כבר מהתור מקיימים שעבורם $dist(v) \leq d$ והוא מייצג את המרחק הנכון שלהם מהמקור.
- בתוך התור ישנם מספר קודקודים בהתחלה (או כולם) שעבורם $dist(v) = d$, ייתכנו עוד מספר קודקודים שעבורם $dist(v) = d + 1$ ועבור כל הקודקודים האלה הוא מייצג את המרחק הנכון שלהם מהמקור.
- כל הקודקודים שלא נכנסו עדיין לתור מקיימים $dist(v) = \infty$ וכן המרחק האמיתי שלהם מהמקור גדול או שווה ל- $d + 1$.

הוכחה: אינדוקציה על k .

עבור $k = 1$, הוצאנו איבר אחד מהתור וכאן $d = 1$. הקודקודים בתור הם כולם השכנים של s , והמרחק שלהם הוא אכן 1 (וזה גם מה שכתוב ב- $dist$ שלהם). כל שאר הקודקודים אינם שכנים ישירים של s , ה- $dist$ הנוכחי שלהם הוא ∞ והמרחק האמיתי שלהם הוא ≥ 2 .

נניח ל- k , ונוכיח עבור $k+1$. אם בשלב ה- k היה רק איבר אחד עם $dist = d$ וכל השאר עם $dist = d+1$ (בתוך התור), אזי בשלב $k+1$ נבחר $d' = d+1$. אם היה יותר מאיבר אחד עם $dist = d$ אזי בשלב $k+1$ נבחר $d' = d$. כעת נבדוק שחלקי הטענה מתקיימים:

- עבור ה- k הראשונים שהוצאנו מהתור ה- $dist$ היה נכון ונשאר נכון. הקודקוד ה- $k+1$ היה בעל $dist$ נכון לפי הנחת האינדוקציה (כיוון שהוא בהכרח היה בתור בשלב ה- k).
- עבור הקודקודים שנמצאים בתור: הוצאנו קודקוד עם מרחק d והכנסנו קודקודים עם מרחק $d+1$ (כיוון שהם שכנים של הקודקוד שהוצאנו). שאר הקודקודים בתור היו בו גם קודם, ולא עדכנו את המרחק שלהם, ולכן הוא עדיין נכון לפי הנחת האינדוקציה. מדוע המרחק נכון לגבי הקודקודים שהוספנו? אם היה עבור קודקוד מסוים w מסלול קצר יותר, הקטן מ- $d'+1$, אזי מהנחת האינדוקציה היינו מגלים את w קודם כשסרקנו את הקודקודים עם $dist \leq d'$.
- הקודקודים שלא נמצאים בתור כמובן בעלי $dist = \infty$ וכן המרחק שלהם גדול או שווה ל- $d'+1$ מאותם שיקולים כמו בסעיף הקודם.

■

כעת נרחיב את הדיון למצב שבו יש משקלות על הצלעות, כלומר לא כל הצלעות בעלות אותו מחיר מבחינת חיפוש המסלול והמרחק האופטימלי. נסמן $l_{(w,v)}$ את המשקל של הצלע (w,v) וכעת רוצים למצוא את המרחק הקצר ביותר בגרף מקודקוד s נתון, ונסמנו ב- $length(s,w)$ עבור כל קודקוד $w \in V$. ייתכן כמובן שהגענו לקודקוד מסוים ורשמנו לעצמנו את המרחק שלו מ- s , אולם בהמשך נגלה מסלול קצר יותר!

כעת נרצה להחזיק את הקודקודים לא לפי הסדר שלהם בגרף, אלא בתור קדימויות (מינימום) לפי המשקלות שלהם. אלגוריתם זה הוא האלגוריתם של Dijkstra:

DIJKSTRA(G,s)

```

 $\forall u \in V, dist(u) \leftarrow \infty, prev(u) \leftarrow NIL$ 
 $dist(s) \leftarrow 0$ 
 $Q \leftarrow BUILD\ MIN\ HEAP(V\ sorted\ by\ dist)$ 
while not EMPTY( $Q$ )
   $v \leftarrow EXTRACT\ MIN(Q)$ 
   $\forall w: (u,w) \in E\ if\ dist(w) > dist(u) + l_{(u,w)}$ 
     $dist(w) \leftarrow dist(u) + l_{(u,w)}$ 
     $prev(w) \leftarrow u$ 
  DECREASE KEY( $Q,w$ )

```

סיבוכיות זמן הריצה של האלגוריתם היא פשוט $\mathcal{O}((|V| + |E|) \log(|V|))$ כיוון שצריך לעבור על כל קודקוד וכל צלע ולבצע פעולות על הערימה.

טענה 41: (נכונות האלגוריתם) לאחר הוצאת k קודקודים מהערימה, נסמן ב- R את קבוצת הקודקודים שהוצאנו מתקיים:

- לכל $v \in R$, $dist(v)$ מכיל את המרחק הנכון שלו מ- s והוא לא השתנה מאז שהקודקוד נכנס ל- R .
- R מכילה את k הקודקודים הקרובים ביותר ל- s (אין קודקוד היותר קרוב ל- s מקודקודי R).
- אם קודקוד w אינו ב- R אך הוא שכן של R (כלומר קיימת צלע בינו לבין קודקוד אחר ב- R), אזי $dist(w)$ מכיל את המרחק הקצר ביותר מ- s ל- w .

הוכחה: אינדוקציה על k .

עבור $k = 1$, $R = \{s\}$ וכמובן $dist(s) = 0$, וכל השכנים של s קיבלו את המרחק שלהם מ- s והוא הקצר ביותר.

נניח עבור k ונניח עבור $k + 1$. נניח w הוא הקודקוד ה- $k + 1$, אזי לפי האלגוריתם היה לו את ערך ה- $dist$ המינימלי בערימה בסוף השלב ה- k . נניח בשלילה שמרחק זה אינו המרחק הקצר ביותר שלו מ- s , אזי קיים מסלול קצר יותר ונניח z הקודקוד הראשון מחוץ ל- R שנמצא על מסלול זה. כיוון ש- z על המסלול הקצר ביותר ל- w , אזי אם נעצור ב- z יש לנו את המסלול הקצר ביותר מ- s ל- z , ולפי הנחת האינדוקציה (חלק 3) מתקיים $dist(z) = length(s, z)$ (המרחק נכון) ולכן קיבלנו $dist(w) < length(s, w) < dist(z) = length(s, z) \leq length(s, w)$ וזה בסתירה לכך ש- w היה הקודקוד עם ה- $dist$ המינימלי בערימה לאחר השלב ה- k .

כעת נראה שאין קודקוד הקרוב יותר ל- R (מחוצה לה) מאשר w . נניח שקיים z שקרוב יותר ל- s מאשר w . אזי $length(s, z) < length(s, w) = dist(w)$. לכן ניקח את הקודקוד x שהוא הקודקוד הראשון מחוץ ל- R על המסלול הקצר $s \rightarrow z$. לפי הנחת האינדוקציה $dist(x) = length(s, x)$. לפיכך, $dist(x) = length(s, x) \leq length(s, z) < length(s, w) < dist(w)$ שוב בסתירה לכך ש- w הוא הקודקוד עם ה- $dist$ המינימלי בערימה לאחר השלב ה- k .

לבסוף, נניח w קודקוד מחוץ ל- R שהוא שכן של R , ונתבונן במסלול הקצר ביותר מ- s ל- w ויהי z הקודקוד האחרון על מסלול זה לפני w (קיימת צלע (z, w)). אנו רוצים להראות שמתקיים $dist(w) = length(s, z) \in R + l_{(z, w)}$. לפי הנחת האינדוקציה, כש- z נכנס ל- R , $dist(z) = length(s, z)$ והמסלול הקצר ביותר ל- z היה כמובן בתוך R . בנוסף, לפי האלגוריתם המרחק של כל שכני z עודכן, כך שמתקיים $dist(w) \leq length(s, z) \in R + l_{(z, w)}$. אם $dist(w) < length(s, z)$ קטן ממש מביטוי זה, אזי יש מסלול $s \rightarrow w$ שהוא קצר ממש מזה. המסלול הזה כולו ב- R למעט הצלע האחרונה, כיוון שהקודקודים ב- R והשכנים שלהם הם היחידים שהמרחק שלהם עודכן לפי האלגוריתם (אצל כל השאר הוא ∞). אבל זו סתירה לכך שבחרנו את המסלול הקצר ביותר ל- w בתוך R מלכתחילה. ■

הערה: ניתן לממש את האלגוריתם *DIJKSTRA* באמצעות ערימת פיבונאצ', שהיא מבנה נתונים מורכב יותר המבטיח amortized cost של $O(\log^*(n))$ (ר' בהמשך) לפעולת *DECREASE KEY*. במקרה זה, אנו מורידים את הסיבוכיות של האלגוריתם ל- $O(|V| \log(|V|) + |E| \log^*(|V|))$ - שיפור די משמעותי בעיקר כאשר יש הרבה צלעות בגרף.

אם מנסים להשתמש באלגוריתם הזה בגרף עם משקלות שליליים על הצלעות מהר מאוד נגיע למסקנה שזה לא עובד. בפרט, הבעיה היא שאנו מניחים שכאשר מוציאים קודקוד מהערימה, מצאנו את המרחק הקצר ביותר אליו ולא יהיה צורך לעדכן אותו יותר. אם נוצר הצורך לעדכן אותו, אז אנחנו אולי נעדכן אותו – אבל לא את השכנים שלו שכבר הוצאנו מהתור. וזו בדיוק הבעיה.

הערה: בגרף עם מעגלים שליליים המונח "המסלול הקצר ביותר" לא מוגדר היטב, כיוון שהמסלול הקצר ביותר בין שני קודקודים בתוך מעגל שהעלות שלו שלילית הוא מסלול אינסופי! לא נתעניין בגרפים כאלה, אך נעריך את היכולת של האלגוריתמים השונים לזהות אותם.

האלגוריתם של בלמן-פורד (Bellman-Ford) יודע למצוא את המרחק המינימלי גם בגרף עם משקלות שליליים (אבל כמובן כשאין בו מעגלים שליליים). בנוסף, הוא יכול לזהות ולהתריע כאשר מתגלה מעגל שלילי. הרעיון הכללי הוא שבכל זמן נתון יש לנו "ניחוש" של המרחק בקצר ביותר מ- s לכל קודקוד אחר. אנו עוברים על כל הצלעות (הרבה פעמים) ובודקים האם הן יכולות לשפר את הניחוש שלנו, כאשר בכל פעם שנתקלנו בצלע מסתכלים על הקודקוד שלתוכו היא נכנסת, ובודקים האם היא משפרת את המרחק.

BELLMAN – FORD(G, w, s)

$\forall u \in V, dist(u) \leftarrow \infty$

$dist(s) \leftarrow 0$

for $i \leftarrow 1$ to $|V| - 1$

$\forall (u, v) \in E$

$dist(v) = \min\{dist(v), dist(u) + w(u, v)\}$

כאשר כאן סימנו $w(u, v)$ את המשקל של הצלע (u, v) .

לאחר שביצענו $|V| - 1$ איטרציות, אנו יודעים בוודאות שמצאנו את המרחקים הקצרים ביותר (נראה מיד מדוע). אם כעת נבצע איטרציה נוספת, ועדיין יש קודקוד מסוים שה- $dist$ שלו עודכן, אזי יש בגרף מעגל שלילי אחד לפחות – וכך אנו יודעים לזהות את זה.

טענה 42א': עבור כל קודקוד v , אם $dist(v) < \infty$ אזי $dist(v)$ הוא אורך מסלול כלשהו $s \rightarrow v$.

הוכחה: **באינדוקציה**

טענה 42ב': לאחר k איטרציות הערך $dist(v)$ יכיל עבור כל קודקוד את אורך המסלול הקצר ביותר $s \rightarrow v$ המכיל לכל היותר k צלעות.

הוכחה: אינדוקציה על k . הבסיס $k = 0$ טריוויאלי. נניח עבור $k - 1$ ונוכיח עבור k . יהי v קודקוד עם מסלול קצר ביותר $s \rightarrow v$ המכיל בדיוק k צלעות. לפי הנחת האינדוקציה, באיטרציה הקודמת לקודקוד האחרון על המסלול u שנמצא לפני v המרחק $dist(u)$ מכיל את המרחק הנכון. אורך המסלול הקצר ביותר הוא לפי הגדרה $dist(u) + w(u, v)$ ולכן אחרי האיטרציה ה- k זהו לכל היותר הערך של $dist(v)$. אבל מטענה 42א' נובע שזהו בדיוק אורך המסלול הקצר ביותר, כנדרש. ■

סיבוכיות זמן הריצה של האלגוריתם היא $O(|V||E|)$ אם הגרף מיוצג ע"י רשימת שכנויות, או $O(|V|^3)$ אם הגרף מיוצג ע"י מטריצת שכנויות.

Union Find

אנו מחפשים מבנה נתונים לאחסון מידע על קבוצות זרות. הפעולות שאנו רוצים לבצע (ה-ADT) הן:

- $MAKESET(x)$ – יצירת קבוצה חדשה המכילה את x בלבד.
- $FIND(x)$ – מחזיר נציג של הקבוצה אליה x שייך.
- $UNION(x, y)$ – מאחדת (ממזגת) את הקבוצות אליהן x, y שייכים.

הרעיון הוא להחזיק כל קבוצה בתור עץ מכוון, כאשר לכל קודקוד יש רק מצביע להורה שלו (אין מצביעים לילדים), ולנציג הקבוצה (שורש העץ) יש מצביע לעצמו. כמו כן, לכל קודקוד נחזיק גם דרגה – את גובה תת העץ שמתחתיו.

MAKESET(x)

$\pi(x) \leftarrow x$

$rank(x) \leftarrow 0$

FIND(x)

while $x \neq \pi(x)$ do $x \leftarrow \pi(x)$

return x

```

UNION( $x, y$ )
 $r_x \leftarrow \text{FIND}(x)$ 
 $r_y \leftarrow \text{FIND}(y)$ 
if  $r_x = r_y$  then return
if  $\text{rank}(r_x) > \text{rank}(r_y)$  then  $\pi(r_y) \leftarrow r_x$ 
else
   $\pi(r_x) \leftarrow r_y$ 
  if  $\text{rank}(r_x) = \text{rank}(r_y)$  then  $\text{rank}(r_y) \leftarrow \text{rank}(r_y) + 1$ 

```

האיחוד הוא הפעולה העדינה – כמובן שתמיד אפשר לגרום לכך שנציג של קבוצה אחת יצביע לנציג של הקבוצה השנייה. עם זאת, אנחנו גם רוצים לצמצם את הגובה של העץ – אז האסטרטגיה צריכה להיות לחבר את שורש העץ הקצר לשורש העץ הארוך.

כעת ניתן לקבל מספר תכונות חשובות שיתנו לנו להבין איך המבנה מתנהג מבחינת זמן ריצה.

טענה 43א': לכל x , $\text{rank}(x) < \text{rank}(\pi(x))$.

הוכחה: ברור מהמשמעות של rank .

טענה 43ב': לכל קודקוד מדרגה k יש לפחות 2^k קודקודים בתת-עץ.

הוכחה: קודקוד מדרגה k נבנה ע"י מיזוג שני עצים מדרגה $k - 1$ שניהם, ובאינדוקציה מקבלים את הדרוש. ■

טענה 43ג': אם יש בסה"כ n קודקודים, יש לכל היותר $\frac{n}{2^k}$ קודקודים מדרגה k .

הוכחה: כל קודקוד פנימי היה פעם שורש, ומאז כמובן הדרגה שלו (וגם הקודקודים מתחתיו) לא השתנו. לשני קודקודים מדרגה k אין כמובן אף צאצא משותף, כיוון שלפי טענה 43א' לכל קודקוד יש אב קדמון אחד לכל היותר מדרגה k . לכן כל הקודקודים מתחלקים על פני העצים שנפרשים ע"י קודקודים בדרגה k , ולכן לכל היותר יש $\frac{n}{2^k}$ קודקודים בדרגה k . ■

מכאן מתקבל שהדרגה המקסימלית (הגובה המקסימלי) היא $\log(n)$, ומכאן מתקבל חסם של $\mathcal{O}(\log(n))$ על זמן הריצה של FIND ושל UNION .

מסתבר שניתן לשפר את הביצועים אפילו יותר אם נבצע עבודה נוספת במהלך FIND . כאשר במהלך FIND אנו עוקבים אחר מצביעים לשורש העץ (הנציג), נוכל לעדכן את כל המצביעים האלה כדי שיצביעו ישירות לנציג ברקורסיה (לטכניקה זו קוראים path compression).

```

FIND( $x$ )
if  $x \neq \pi(x)$  then  $\pi(x) \leftarrow \text{FIND}(\pi(x))$ 
return  $\pi(x)$ 

```

כעת אנו רוצים לנתח את הסיבוכיות מחדש, ולהבין האם הרווחנו משהו. זה מורכב כי אנחנו צריכים לנתח רצף של פעולות איחוד וחיפוש, ולא רק פעולה אחת בודדת בכל פעם.

אמנם כבר אי אפשר לקרוא לדרגה "הגובה של העץ" כי זה לא נכון, אבל התכונות שהוכחנו קודם (טענות 43א-ג') עדיין מתקיימות. לכן אם יש לנו n איברים, הדרגות שלהם יכולות להיות בין 0 ל- $\log(n)$ ע"פ טענה 43ג'. את כל הדרגות נחלק לקבוצות כאשר כל קבוצה היא מהצורה $\{k+1, \dots, 2^k\}$ כאשר k בעצמו חזקה של 2.

הגדרה: $x = \log^*(n)$ אם צריך לבצע $\log$ שוב ושוב x פעמים כדי להביא את הערך של n להיות פחות מ-1. לדוגמה, $\log^*(1000) = 4 \Rightarrow \log(\log(\log(\log(1000)))) < 1$.

טענה 44: העלות הכוללת של m פעולות $FIND$ היא $O(m \log^*(n) + n \log^*(n))$.

הוכחה: מספר הקבוצות שיש לנו הוא $\log^*(n)$, וזו פונקציה שגדלה כל כך לאט שאפשר להתייחס אליה כקבוע עבור כל צורך פרקטי אפשרי. כעת מה שאנו עושים הוא ניתוח Amortized Cost, ועל מנת לעשות זאת נשתמש בטכניקה הבאה: ניתן לכל קודקוד סכום כסף כך שסך הכסף שנוציא יתאים למה שאנחנו רוצים.

לכל קודקוד נעניק כסף ברגע שהוא מפסיק להיות שורש של עץ (ואז הדרגה שלו נקבעת לנצח). אם הדרגה שלו היא באינטרבל $\{k+1, \dots, 2^k\}$ אזי הוא יקבל 2^k דולרים. לפי טענה 43ג', מספר הקודקודים מגובה k או יותר חסום ע"י: $\frac{n}{2^{k+1}} + \frac{n}{2^{k+2}} + \dots \leq \frac{n}{2^k}$. לכן, סכום הכסף שמקבלים הקודקודים באינטרבל זה הוא לכל היותר n דולרים, וכיוון שיש $\log^*(n)$ אינטרבלים סך הכסף שנתנו לקודקודים הוא $n \log^*(n)$.

הזמן שלוקח לבצע $FIND$ הוא פשוט מספר המצביעים שאנחנו צריכים לעבור לפני שמגיעים לשורש. נתבונן בדרגות הקודקודים שאותם אנו מבקרים בדרך לשורש, המתחלקים לשתי קבוצות:

- הדרגה של $\pi(x)$ נמצאת באינטרבל גדול יותר מהדרגה של x , כלומר יש לנו: $[k+1, \dots, x], [\pi(x), \dots]$. יש לכל היותר $\log^*(n)$ קודקודים כאלה (לפי מס' האינטרבלים), ואנחנו סופגים את העלות שלהם - $O(\log^*(n))$. כיוון שאנו מבצעים m חיפושים, מכאן מגיעה העלות $m \log^*(n)$.
- הדרגה של $\pi(x)$ היא באותו אינטרבל כמו הדרגה של x , ואז נבקש מהקודקוד לשלם לנו דולר אחד.

אבחנה חשובה: בכל פעם שבמקרה השני קודקוד מסוים משלם לנו דולר, ההורה שלו עובר להיות מדרגה גבוהה יותר (כי האלגוריתם מקשר אותו לנציג הקבוצה, אלא אם כן הוא כבר היה מקושר אליו – אך לא חייבים להתייחס למקרה זה). לכן, אם $rank(x) \in \{k+1, \dots, 2^k\}$ אזי הוא ייאלץ לשלם לכל היותר 2^k דולרים לפני שההורה שלו מגיע לדרגה הגבוהה ביותר בקבוצה, ומאותו רגע הוא לא ישלם יותר לעולם (כי הוא יגיע למקרה הראשון, בו אנחנו סופגים את העלות). לכן הקודקודים קיבלו מלכתחילה מספיק כסף ($n \log^*(n)$) ולכן ניתוח הסיבוכיות שלנו עומד בעינו – אנו נדרשים ל- $O(m \log^*(n) + n \log^*(n))$ עבודה על מנת לבצע m פעולות $FIND$ על המבנה. ■

עץ פורש מינימלי – Minimal Spanning Tree

אנו רוצים לקחת גרף לא מכונן קשיר עם משקלות חיוביים על הצלעות ולייצר גרף קשיר אחר עם אותם הקודקודים שבו סכום המשקלות יהיה מינימלי. זהו עץ פורש מינימלי – MST. מבנה זה נקרא עץ, כי בתת קבוצה זו של צלעות לא ייתכן מעגל (אחרת ניתן להוציא אותו ולא פגענו בקישוריות, אבל בגלל חיוביות המשקלות – הורדנו את העלות). כמובן, בהרבה מקרים ה-MST אינו יחיד.

הגדרה: עץ הוא גרף קשיר ללא מעגלים.

הגדרה: יהי $G = (E, V)$ גרף לא מכונן, אזי עץ פורש של G הוא עץ $T = (E', V)$ כך ש- $E' \subset E$.

הגדרה: יהי $G = (E, V)$ גרף לא מכונן, ו- $w: E \rightarrow R^+$ פונקציית משקל על הצלעות. אזי MST של הגרף G הוא עץ פורש $T = (E', V)$ כך ש- $E' \subset E$ המינימלי, כלומר מתקיים $weight(T) = \min_T \sum_{e \in E'} w(e)$.

טענה 45א': לעץ על n קודקודים יש בדיוק $n - 1$ צלעות.

הוכחה: נסתכל על T עץ ונסמן את מספר הצלעות בו ב- t . נתחיל ב- n קודקודים ונוסיף צלעות אחת אחת התחלנו ב- n רכיבי קשירות, ובכל הוספת צלע אנחנו מורידים מספר זה ב-1 (אחרת נוצר מעגל, אם מוסיפים צלע לאותו רכיב קשירות). לכן כדי להגיע לרכיב קשירות אחד (העץ) אנו צריכים $t = n - 1$ צלעות. ■

טענה 45ב': אם בגרף קשיר על n קודקודים יש $n - 1$ צלעות, אזי זהו עץ.

הוכחה: נניח בשלילה שיש מעגל, אזי אפשר להוריד צלעות בלי לפגוע בקשירות ולקבל עץ. אבל בעץ הזה יש $n - 1$ צלעות לפי טענה 45א', ולכן לא הורדנו אף צלע, כלומר לא היו מעגלים מלכתחילה, כלומר זה היה עץ מלכתחילה. ■

האלגוריתם של קרוסקל למציאת MST:

1. בנה גרף המכיל רק את הקודקודים של G
2. סדר את הצלעות של G בסדר עולה לפי משקלן
3. בדוק האם הוספת הצלע e בעלת המשקל הנמוך ביותר יוצרת מעגל
 - a. אם לא, הוסף לגרף את e
 - b. אחרת, זרוק את e מרשימת הצלעות הממוינת
4. חזור ל-3 עד שאין יותר צלעות

הגדרה: חתך הוא חלוקה של גרף לשתי קבוצות של קודקודים (לא ריקות וזרות) V_1, V_2 , וצלעות בחתך הן צלעות שעוברות בין קודקודים $v_1 \in V_1, v_2 \in V_2$.

טענה 46: (תכונת החתך) נניח ש- $X \subset E$ כך שגם $X \subset T$ (כש- T הוא MST כלשהו), ונניח שיש חתך בגרף כך שאף צלע ב- X אינה בצלעות החתך. אזי אם e היא צלע עם משקל מינימלי בחתך, אזי $X \cup \{e\} \subset T'$ כאשר T' הוא גם כן MST (אך לא בהכרח $T = T'$).

הוכחה: אם $e \in T$ אזי נבחר $T' = T$ והטענה נכונה. אחרת, נוסיף את e ל- T וניצור את $T \cup \{e\}$. בגרף זה יש כמובן מעגל, שכן שני קודקודי e היו מחוברים ב- T בדרך אחרת שאינה e . המסלול ביניהם עובר מצד אחד של החתך לצד השני, ולכן יש צלע במסלול e' שנמצאת בחתך (ואינה e). אז נגדיר $T' = T \cup \{e\} \setminus \{e'\}$ ונותר לנו להראות שהוא MST. הוא קשיר, כי הורדנו צלע ממעגל. הוא עץ שכן יש בו $n - 1$ צלעות על n קודקודים (לקחנו את T , הוספנו צלע והורדנו צלע). מדוע הוא מינימלי? ובכן, e נבחרה להיות המינימלית בין צלעות החתך, כלומר $w(e) \leq w(e')$. אנו יודעים גם ש- $w(T') = w(T) + w(e) - w(e')$ ולכן ברור שאם $w(e) < w(e')$ אז T בכלל לא היה עץ פורש מינימלי, ולכן בהכרח $w(e) = w(e')$ ולכן $w(T) = w(T')$ ולכן גם T' הוא MST. ■

טענה 47: האלגוריתם של קרוסקל מוצא עץ פורש מינימלי.

הוכחה: כאן נחזיק X אוסף הצלעות בכל שלב של האלגוריתם. מוסיפים צלע מינימלית שלא יוצרת מעגל ל- X , ורוצים להראות ש- $X \cup \{e\}$ עדיין מוכל ב-MST כלשהו. יודעים ש- X הוא יער (יש בו עצים $T_1, \dots, T_k$) ונתבונן בצלע e . היא לא יכולה להיות באותו העץ (כי אחרת היה נוצר מעגל), אז היא בוודאי מחברת בין שני עצים שונים, בה"כ T_1, T_2 . לכן החתך שלנו יהיה – קבוצה ראשונה $\{T_1\}$ וקבוצה שנייה $\{T_2, \dots, T_k\}$ וכאן כמובן X לא מכיל אף צלע בחתך ואפשר להפעיל את תכונת החתך. מדוע e היא הצלע המינימלית בחתך? ובכן, היא הצלע המינימלית שלא יוצרת מעגלים ולכן היא בוודאי בחתך, ולכן משיקולי מינימליות היא המינימלית בצלעות החתך. ■

כיצד מממשים את האלגוריתם הזה? אפשר לעשות זאת באמצעות מבנה הנתונים UNION FIND שראינו קודם. במקרה שלנו, נרצה לשמור במבנה את הקבוצות הזרות $\{T_1, \dots, T_k\}$. נמיינ את הצלעות לפי משקלן. בכל פעם

שבוחרים צלע, בודקים האם שני הקודקודים בקצוות שלה נמצאים באותה קבוצה – $find(v) = find(w)$. אם לא, נוסיף את הצלע לקבוצה X ונאחד את שתי הקבוצות שבהן v, w נמצאים.

KRUSKAL(G, w)

$\forall v \in V, MAKESET(v)$

$X \leftarrow \emptyset$

$QUICK\ SORT(E\ by\ w)$

$\forall (v, u) \in E$

if $find(v) \neq find(u)$ then $X \leftarrow X \cup \{(v, u)\}, UNION(v, u)$

סיבוכיות זמן הריצה של האלגוריתם היא קודם כל $\mathcal{O}(|E| \log(|E|))$ על מנת למיין את הצלעות לפי המשקלות, ולאחר מכן בוחרים בכל פעם צלע ומבצעים פעולות $FIND, UNION$ שהן במקרה הגרוע $\mathcal{O}(\log(|V|))$ ולכן בסה"כ אנו מקבלים $\mathcal{O}(|E| \log(|E|)) + |E| \log(|V|)$ אבל כיוון שבכל מקרה $|E| \leq |V|^2$ אזי $\log(|E|) \leq 2 \log(|V|)$ ולכן אפשר פשוט לומר שהסיבוכיות היא $\mathcal{O}(|E| \log(|E|))$.

האלגוריתם של פריים (Prim) למציאת עץ פורש מינימלי עובד באופן שונה, ומאוד דומה ל- $DIJKSTRA$. להלן האלגוריתם:

PRIM(G, w, s)

$\forall u \in V, key(u) \leftarrow \infty, \pi(u) \leftarrow NIL$

$key(s) \leftarrow 0$

$Q \leftarrow BUILD\ HEAP(V)$

while not $EMPTY(Q)$

$u \leftarrow EXTRACT\ MIN(Q)$

$\forall v: (u, v) \in E$ if $v \in Q$ and $w(u, v) < key(v)$

$\pi(v) \leftarrow u, key(v) \leftarrow w(u, v)$

יש לשים לב שכאשר משנים את $key(v)$, מתבצע $DECREASE\ KEY$ ל- v מאחורי הקלעים. בסוף זמן הריצה, צלעות העץ הפורש הן $\{(v, \pi(v)): v \in V, \pi(v) \neq NIL\}$.

הערה: ניתן להראות שעבור כל קודקוד, הצלע המחוברת אליו בעלת המשקל המינימלי חייבת להיות ב-MST. זה די אינטואיטיבי, כיוון שאם נבחר ב-MST צלע אחרת, אז נוכל להוסיף את הצלע המינימלית, ליצור מעגל, ולהוריד את הצלע המקורית – ולקבל MST "יותר מינימלי".

טענה 48: בכל שלב באלגוריתם $PRIM$ קיים עץ פורש מינימלי המכיל את קבוצת הצלעות שנבחרה ע"י האלגוריתם.

הוכחה: נגדיר את החתך S שהוא רכיב הקשירות של קודקוד ההתחלה s וכל שאר הגרף. כעת נוכיח באינדוקציה על מספר הצלעות ב-MST. עבור קבוצה ריקה הטענה טריוויאלית. אם בשלב הקודם קבוצת הצלעות X שנבחרה ע"י האלגוריתם יכולה להתרחב ל-MST, אזי בשלב הזה לפי הגדרת החתך אין ל- X צלעות בחתך, והצלע החדשה e היא המינימלית בצלעות החתך, ולכן קיים MST המכיל את $X \cup \{e\}$ (לפי טענה 46) כנדרש. ■

טענה 49: אם כל המשקלות בגרף G שונים זה מזה, אזי קיים לו MST יחיד.

הוכחה: נניח בשלילה שיש $T' \neq T$ שניהם MST של G . נתבונן בצלע המינימלית במשקלה (והיחידה, בגלל ההנחה) בקבוצה $(T' \cup T) \setminus (T' \cap T)$, ונסמנה e . נניח $e \in T$ ונוסיף אותה ל- T' , וכעת נוצר לנו מעגל. לא ייתכן

שכל שאר הצלעות המעגל נמצאות ב- T כי $e \in T$ ואז היה לנו מעגל ב- T . לכן יש צלע $f \neq e$ במעגל שניתן להסיר אותה כדי לקבל עץ T'' . כמובן $w(T'') = w(T) + w(e) - w(f)$ אבל $w(e) < w(f)$ כי הייתה המינימלית במשקלה, ולכן $w(T'') < w(T)$ וזאת בסתירה למינימליות של T . לכן העץ יחיד. ■

בהצלחה במבחן!