

מבני נתונים - פרויקט מספר 2 – ערמות מתקדמות

חלק מעשי - תיעוד

בפרויקט זה כתבנו מחלקה בשם Heap המממשת סוגי ערימות שונות מעל מספרים שלמים חיוביים. המחלקה תומכת בגמישות מימושית ומאפשרת לבחור (בעת יצירת הערימה) האם להשתמש ב lazy melds והאם להשתמש ב lazy decrease keys. מטרת המחלקה היא לאחסן זוגות של Key-Value ולתמוך בפעולות ערימה יעילות.

תפקידו של כל חבר במחלקה

- `self.lazyMelds` – שדה בוליאני הקובע האם הערימה עובדת במצב של "מיזוג עצל". אם הערך `true`, פעולות מיזוג והכנסה הן מהירות והסידור נדחה. אם `false`, מתבצע סידור (successive linking) בכל פעולת הכנסה ומיזוג.
- `self.lazyDecreaseKeys` – שדה בוליאני הקובע את אופן הטיפול בהקטנת מפתח. אם הערך `true`, מתבצע שימוש במנגנון ה-Cascading Cuts של ערימת פיבונאצ'י. אם `false`, מתבצעת פעולת `Heapify-Up` סטנדרטית.
- `self.min` – שדה המחזיק מצביע לאובייקט בעל המפתח המינימלי הנוכחי בערימה.
- `self.head` – שדה המחזיק מצביע לאיבר הראשון ברשימת השורשים של הערימה.
- `self.last` – שדה המחזיק מצביע לאיבר האחרון ברשימת השורשים של הערימה.
- `self.size` – שדה המחזיק את מספר האיברים הכולל בערימה.
- `self.numTrees` – שדה המחזיק את מספר העצים הנוכחי ברשימת השורשים.
- `self.numMarkedNodes` – שדה המונה את כמות הצמתים המסומנים בערימה.
- `self.totalLinks` – מונה הסופר את סך פעולות הקישור (Link) שבוצעו בין שני עצים לאורך חיי הערימה.
- `self.totalCuts` – מונה הסופר את סך פעולות החיתוך (Cut) שבוצעו לאורך חיי הערימה.
- `self.totalHeapifyCosts` – מונה הסופר את סך פעולות ההחלפה (Swap) שבוצעו במסגרת תהליך `Heapify-Up`.

תיאור הפונקציות

- **insert** - הפונקציה מכניסה איבר חדש לערימה עם המפתח והמידע הנתונים. היא מחזירה את האובייקט `HeapItem` החדש שנוצר, המכיל מצביע לצומת בערימה. מבחינת אופן הפעולה, הפונקציה יוצרת תחילה צומת חדש ומאתחלת את מצביעי ה-`next` וה-`prev` שלו להצביע על עצמו. לאחר מכן, היא יוצרת ערימה זמנית חדשה (`heap2`) המכילה רק את האיבר החדש, ומבצעת קריאה לפונקציה `meld` כדי למזג את הערימה הזמנית לתוך הערימה הנוכחית.
ניתוח סיבוכיות: מן הריצה תלוי בערכו של השדה `lazyMelds`:
 - אם `lazyMelds` הוא `true`, פעולת ה-`meld` מבצעת שרשור של רשימות השורשים בלבד, ללא `successiveLinking`. פעולה זו כוללת מספר קבוע של עדכוני מצביעים. לכן, זמן הריצה הוא $O(1)$.
 - אם `lazyMelds` הוא `false`, פעולת ה-`meld` קוראת ל-`successiveLinking`. במקרה הגרוע ביותר (בו חיבור העץ החדש גורר רצף של מיזוגים), ייתכנו מיזוגים עד לגובה העץ המקסימלי. מאחר וגובה העץ חסום לוגריתמית, זמן הריצה במקרה הגרוע הוא $O(\log n)$.
- **findMin** - הפונקציה מחזירה את האובייקט בעל המפתח המינימלי בערימה. הפונקציה ניגשת ישירות לשדה `self.min`, אשר מחזיק מצביע לאיבר המינימלי הנוכחי. אם הערימה ריקה, מוחזר ערך `null`.

ניתוח סיבוכיות: המצביע למינימום מתוחזק באופן שוטף ולכן הגישה אליו קוראת בזמן קבוע של $O(1)$.

- **deleteMin** - הפונקציה מוחקת את האיבר המינימלי מהערימה. תחילה הפונקציה מטפלת במקרי קצה (ערימה ריקה או ערימה עם איבר יחיד). לאחר מכן, היא מוציאה את איבר המינימום מרשימת השורשים ומעדכנת את המצביעים ואת גודל הערימה בהתאם. אם לאיבר שנמחק היו ילדים, הפונקציה הופכת אותם לשורשים עצמאיים. לבסוף, הפונקציה קוראת ל-*successiveLinking* כדי לבצע פעולת *consolidate* ולסדר את הערימה מחדש, ואז סורקת את רשימת השורשים החדשה כדי למצוא ולעדכן את מצביע ה-*min* החדש.

ניתוח סיבוכיות: זמן הריצה תלוי בכמות העצים ברשימת השורשים ובדרגת האיבר המינימלי שנמחק.

- במקרה של *lazyMelds=true*, ייתכן שרשימת השורשים מכילה $O(n)$ עצים (עקב רצף הכנסות ללא סידור). פעולת ה-*successiveLinking* ועדכון ה-*min* יאלצו לסרוק את כל העצים הללו. לכן, זמן הריצה במקרה הגרוע הוא $O(n)$. עם זאת, נציין כי עלות ה-*amortize* במצב זה היא $O(\log n)$.
- במקרה של *lazyMelds=false*, רשימת השורשים תמיד מכילה לכל היותר $O(\log n)$ עצים. לכן, תהליך המחיקה, הקישור מחדש ועדכון המינימום חסומים כולם על ידי גובה העץ. זמן הריצה הוא $O(\log n)$ ב-*WC*.

- **decreaseKey** - הפונקציה מקטינה את המפתח של איבר נתון (x) בערך מסוים (*diff*) ושומרת על חוקיות הערימה. תחילה הפונקציה מעדכנת את המפתח של האיבר ובודקת אם הוא קטן מהמינימום הנוכחי (ומעדכנת את *min* בהתאם). לאחר מכן, אופן הפעולה תלוי בדגל *lazyDecreaseKeys*:

- אם הוא *true*, נבדק האם הופך כלל הערימה. במידה וכן, נקראת הפונקציה *cascadingCut* כדי לנתק את הצומת ולטפל בשרשרת הסימונים במעלה העץ.
- אם הוא *false*, נקראת הפונקציה *heapifyUp* כדי לבצע פעפוע של האיבר כלפי מעלה.

ניתוח סיבוכיות: נראה כי זמן הריצה ב-*WC* הוא $O(\log n)$.

- במקרה הראשון (*if*): הפונקציה מבצעת *cascadingCut*. במקרה הגרוע, בהתאם לערך של *lazyMelds*, עלות הריצה היא $O(\log^2 n)$ או $O(\log n)$.
- במקרה השני (*else*): הפונקציה מבצעת *heapifyUp* שעלותו חסומה בגובה העץ ולכן זמן הריצה יהיה $O(\log n)$.

- **heapifyUp** - פונקציית עזר המעלה צומת במעלה העץ עד שחוקיות הערימה משוחזרת. הפונקציה רצה בלולאה כל עוד הצומת אינו שורש ומפתחו קטן ממפתח אביו. בכל איטרציה מתבצעת החלפה (באמצעות קריאה למתודה *swapWithParent*) והצומת מתקדם לאביו.
- ניתוח סיבוכיות: במקרה הגרוע, הצומת יעלה מהעלה העמוק ביותר ועד לשורש. על כן, זמן הריצה ב-*WC* הוא $O(\log n)$.

- **swapWithParent** - פונקציית עזר המבצעת החלפה בין צומת לאביו. ההחלפה מתבצעת על ידי החלפת התוכן של ה-*HeapItem* בין שני הצמתים, ועדכון המצביעים המתאימים בתוך ה-*Item*. כמו כן, הפונקציה מעדכנת את המצביעים הגלובליים (*head, last, min*) במידת הצורך.
- ניתוח סיבוכיות: הפונקציה מבצעת פעולות בזמן קבוע בלבד, לכן זמן הריצה הוא $O(1)$.
- **cascadingCut** - פונקציה רקורסיבית המבצעת את מנגנון הניתוק המדורג. תחילה הפונקציה קוראת ל-*cut* כדי לנתק את הבן (x) מאביו (y). לאחר מכן היא בודקת את האב: אם האב אינו שורש והוא אינו מסומן (*marked*), נסמן אותו (*marked=true*) והתהליך נעצר. אם האב כבר מסומן, הסימון מוסר (*marked=false*), והפונקציה קוראת לעצמה רקורסיבית כדי לנתק גם את האב מאביו שלו.
- ניתוח סיבוכיות: במקרה הגרוע ביותר, שרשרת הניתוקים נמשכת עד לשורש העץ. מכיוון שגובה העץ חסום לוגריתמית, מספר הקריאות הרקורסיביות חסום על ידי גובה העץ. כאשר *lazyMelds=True*, זמן הריצה הוא $O(\log n)$, אחרת כל פעולת *cut* היא בזמן ריצה של $O(\log n)$ ולכן סה"כ $O(\log^2 n)$.

- **cut** - פונקציית עזר המנתקת קשר בין צומת (x) לאביו (y) והופכת את x לשורש חדש בערימה. הפונקציה מעדכנת את מצביעי הילדים של האב, מקטינה את דרגתו, ומאפסת את סימון ה-*marked*.

של הבר. לאחר מכן, הבר מתווסף לרשימת השורשים של הערימה באמצעות יצירת ערימה זמנית וקריאה ל-`meld`.

ניתוח סיבוכיות: הפונקציה מבצעת מספר קבוע של עדכוני מצביעים וקוראת ל-`meld`. בהנחה שהערימה מוגדרת כעצלה (`lazyMelds=true`), פעולת המיזוג היא $O(1)$ וכך גם זמן הריצה של המתודה. במידה ו-`lazyMelds=false`, הסיבוכיות תהיה $O(\log n)$.

- **delete** - הפונקציה מוחקת איבר כללי (x) מהערימה. המימוש מתבצע על ידי הקטנת המפתח של האיבר לערך המינימלי האפשרי כדי להבטיח שהוא יהיה הקטן ביותר בערימה. לאחר שינוי המפתח, הפונקציה קוראת ל-`deleteMin` כדי להוציא את האיבר מהערימה.
ניתוח סיבוכיות: הפונקציה מורכבת משני שלבים: `decreaseKey` שעלותו $O(\log n)$ במקרה הגרוע, ו-`deleteMin`. בהנחה שהערימה מוגדרת כלא-עצלה (`lazyMelds=false`), גם `deleteMin` חסומה ב- $O(\log n)$ אם `lazyDecreaseKeys=false` אחרת חסומה ב- $O(\log^2 n)$, אם `lazyMelds=True` חסומה ב- $O(n)$. בפרט, WC בזמן הריצה הוא $O(n)$.

- **meld** - הפונקציה ממזגת את הערימה הנוכחית עם ערימה נוספת (`heap2`). תחילה הפונקציה משרשרת את רשימת השורשים של הערימה השנייה לקצה רשימת השורשים של הערימה הנוכחית. לאחר מכן היא מעדכנת את השדות הסטטיסטיים (`size`, `numTrees`, `numMarkedNodes` וכו') ואת המינימום הגלובלי. לבסוף, נבדק את הערך של `lazyMelds`: אם הוא `false`, מתבצעת קריאה ל-`successiveLinking` כדי לסדר את הערימה המאוחדת.
ניתוח סיבוכיות: מתחלק לפי ערכו של השדה `lazyMelds`:

- אם `lazyMelds=true`: מתבצע שרשרת מצביעים בלבד ועדכון שדות, ולכן הזמן הוא $O(1)$.
- אם `lazyMelds=false`: מתבצעת קריאה ל-`successiveLinking` הסורקת את כל השורשים. במקרה הגרוע, מספר השורשים פרופורציונלי ל- n . לכן, הזמן הריצה הוא $O(n)$.

- **successiveLinking** - פונקציית עזר המבצעת את תהליך ה-`Consolidation`. הפונקציה אחראית על ארגון מחדש של רשימת השורשים כך שלא יישארו שני עצים בעלי אותה דרגה. היא יוצרת מערך עזר בגודל לוגריתמי, מאתחלת אותו ב-`null`, וקוראת לפונקציה `consolidate`. לאחר מכן היא מעדכנת את מצביעי ה-`head` וה-`last` בהתאם לרשימה החדשה שהתקבלה.

ניתוח סיבוכיות: הפונקציה תלויה ישירות בפונקציה `consolidate` הסורקת את כל שורשי הערימה. בפרט, במקרה הגרוע כמות השורשים לינארית בכמות האיברים ולכן זמן הריצה הוא $O(n)$.

- **link** - פונקציית עזר המחברת שני עצים בעלי דרגה זהה. הפונקציה קובעת מי מהשורשים הוא בעל המפתח הקטן יותר, והופכת את השורש השני לילד שלו. הפעולה כוללת עדכון מצביעי הורים, ילדים ואחים, והגדלת הדרגה של השורש בעל המפתח הגדול.
ניתוח סיבוכיות: הפונקציה מבצעת מספר קבוע של שינויי מצביעים ופעולות אריתמטיות. זמן הריצה הוא $O(1)$.

- **toBucket** - פונקציית עזר המעבירה את כל השורשים לתוך מערך ה"סלים" (`Buckets`) ומבצעת מיזוגים תוך כדי. הפונקציה עוברת על רשימת השורשים. עבור כל שורש, היא בודקת אם קיים כבר עץ באותו הדרגה ב"סל" המתאים. אם כן, היא מבצעת `link` ביניהם, מפנה את הסל הנוכחי, ומנסה להכניס את העץ המאוחד לסל בדרגה הבאה (תהליך החוזר עד שנמצא סל ריק).
ניתוח סיבוכיות: הפונקציה עוברת על כל השורשים ברשימה המקורית. במקרה הגרוע כאמור עבור ערימות עצלות, ייתכנו $O(n)$ שורשים ועל כן זמן הריצה יהיה $O(n)$, אחרת זמן הריצה יהיה $O(\log n)$.

- **fromBucket** - פונקציית עזר הבונה מחדש את רשימת השורשים מתוך מערך הסלים. הפונקציה סורקת את מערך הסלים, וכל עץ שנמצא בו מחובר לשרשרת חדשה של שורשים. במקביל, היא מעדכנת את מונה העצים (`numTrees`).
ניתוח סיבוכיות: הפונקציה עוברת על מערך הסלים שגודלו חסום לוגריתמית ומבצעת פעולות קבועות לכל תא. לכן, זמן הריצה הוא $O(\log n)$.

- **consolidate** - פונקציה המאחדת את תהליך הסידור: קריאה ל-`toBucket` לפיזור ומיזוג העצים, ולאחריה קריאה ל-`fromBucket` לאיסוף העצים לרשימה חדשה.
ניתוח סיבוכיות: הפונקציה מבצעת את שני השלבים לעיל. השלב הדומיננטי הוא `toBucket` ועל כן זמן הריצה הכולל ב- WC יהיה $O(n)$ עבור ערימות עצלות, אחרת $O(\log n)$.

- **getters** - הפונקציות הבאות מחזירות ערכים של שדות המחלקה. כל הערכים מתוחזקים באופן שוטף במהלך פעולות הערימה, ולכן השליפה היא מיידית בזמן ריצה של $O(1)$.
 - **size** – מחזירה את מספר האיברים בערימה.
 - **numTrees** – מחזירה את מספר העצים (השורשים) בערימה.
 - **numMarkedNodes** – מחזירה את מספר הצמתים המסומנים.
 - **totalLinks** – מחזירה את סך פעולות הקישור שבוצעו.
 - **totalCuts** – מחזירה את סך פעולות החיתוך שבוצעו.
 - **totalHeapifyCosts** – מחזירה את כמות הפעולות שבוצע תהליך ה-*heapify*.

חלק ניסויי/תיאורטי

1. נגדיר פונקציית פוטנציאל. בניגוד לערימת פיבונאצ'י רגילה, שבה פונקציית הפוטנציאל היא $T + 2m$, כאן מספר העצים תמיד חסום ע"י $O(\log n)$ בגלל שכל פעולה שקוראת ל *meld* מבצעת *successive linking*. כמו כן, מאותה סיבה כל ניתוק עולה $O(\log n)$. לכן נגדיר את פונקציית הפוטנציאל להיות $\Phi = T + m * \log n$.

insert – *actual cost* הוא $O(\log n)$ מאחר ומתבצעת הכנסה ($O(1)$) ולאחריה *successive linking* ($O(\log n)$). כמו כן, $\Delta\Phi = T_1 + m * \log n - (T_0 + m * \log n) = O(1)$. לכן סה"כ עלות בניתוח לשיעורין היא $O(\log n)$.

findMin – לאורך המימוש מתוחזק מצביע לאיבר המינימלי, לכן $O(1)$.

deleteMin – כפי שראינו בהרצאה, *actual cost* חסום ע"י $T_0 + \log n$. נקבל כי מתקיים במקרה זה $\Delta\Phi = T_1 + m_1 * \log n - (T_0 + m_0 * \log n)$. לכן $\Delta\Phi = T_1 + (m_1 - m_0) \log n + \log n$. מכיוון ש T_1 חסום ע"י $\log n$, ובנוסף מתקיים $m_1 - m_0 \leq 0$ (לא ייתכן שנוספו צמתים מסומנים, יתכן שירדו בכמות) אז נסיק כי $\text{amortize} = O(\log n)$.

decreaseKey – במקרה זה, *actual cost* הוא $O(c \log n)$ כאשר c הוא כמות החיתוכים שמבצעים בפעולה, זאת מאחר וכל חיתוך גורר הוספה לרשימה וביצוע *meld* נפרד (כאמור, *meld* היא פעולה בעלות של $O(\log n)$). מבחינת ניתוח השינוי בפוטנציאל, ראשית נשים לב כפי שראינו בהצגה כי השינוי בכמות הצמתים המסומנים הוא $\Delta m = 1 - (c - 1) = 2 - c$. לכן קיבלנו כי השינוי בפוטנציאל הוא $\Delta\Phi = \Delta T + \Delta m * \log n = c + (2 - c) \log n$. מכאן נקבל כי סה"כ ניתוח של זמן הריצה לשיעורין הוא $\text{amortize} = c \log n + c + 4 \log n - c \log n = 4 \log n + c = O(\log n)$.

delete – קוראת ל *decrease key* ולאחר מכן ל *delete min*. הראנו כי ניתוח לשיעורין של שתי הפעולות הוא $\text{amortize} = O(\log n)$.

לכן סך הכל $\text{amortize} = O(\log n)$.

2.

פעולה	ערימה בינומית	ערימה בינומית עצלה	ערימת פיבונאצ'י	ערימה בינומית עם ניתוקים
Insert	$O(1)$	$O(1)$	$O(1)$	$O(\log n)$
findMin	$O(1)$	$O(1)$	$O(1)$	$O(1)$
deleteMin	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
decreaseKey	$O(\log n)$	$O(\log n)$	$O(1)$	$O(\log n)$
delete	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$

3.

ערימה בינומית עם ניתוקים	ערימת פיבונאצ'י	ערימה בינומית עצלה	ערימה בינומית	
$O(n \log n)$	$O(n)$	$O(n)$	$O(n)$	ניסוי 1
$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	ניסוי 2
$O(n \log n)$	$O(n)$	$O(n \log n)$	$O(n \log n)$	ניסוי 3

4.

ניסוי 1:

ערימה בינומית עם ניתוקים	ערימת פיבונאצ'י	ערימה בינומית עצלה	ערימה בינומית	
116	18	19	123	זמן ריצה (מילישניות)
464645	464645	464645	464645	גודל הערימה בסיום
9	9	9	9	מספר העצים בסיום
464653	464653	464653	464653	מספר חיבורים
0	0	0	0	מספר חיתוכים
0	0	0	0	סה"כ עלויות של <i>heapify up</i>
18	464636	464636	18	עלות מקסימלית לפעולה

ניסוי 2:

ערימה בינומית עם ניתוקים	ערימת פיבונאצ'י	ערימה בינומית עצלה	ערימה בינומית	
505	209	888	1205	זמן ריצה (מילישניות)
46	46	46	46	גודל הערימה בסיום
2	2	4	4	מספר העצים בסיום
1405996	748761	7552691	7552819	מספר חיבורים
748756	748717	0	0	מספר חיתוכים
0	0	4148715	4149301	סה"כ עלויות של <i>heapify up</i>
39	464636	464636	35	עלות מקסימלית לפעולה

ניסוי 3:

ערימה בינומית	ערימה בינומית עזלה	ערימת פיבונאצ'י	ערימה בינומית עם ניתוקים	
115	23	24	121	זמן ריצה (מילישניות)
464644	464644	464644	464644	גודל הערימה בסיום
8	8	11	11	מספר העצים בסיום
464670	464653	509247	509262	מספר חיבורים
0	0	44614	44610	מספר חיתוכים
114559	114600	0	0	סה"כ עלויות של <i>heapify up</i>
18	464636	464636	18	עלות מקסימלית לפעולה

5.

א. אכן ניתן לראות כי קיימים הבדלים בביצועים של הערימות בניסויים השונים.

בניסוי הראשון, ניתן לראות כי הערימות אשר $\text{lazy meld} = \text{True}$, (פיבונאצ'י ועזלה) הפגינו ביצועים טובים יותר. זאת מכיוון ש lazy meld חוסכת זמן על ידי דחיית איחוד העצים לפעולת delete min . לעומת זאת, הערימות האחרות מבצעות את האיחודים בכל פעולת Insert ומכאן נובע הפער בזמני הריצה בין שני סוגי הערימות.

בניסוי השני, נשים לב כי הערימות אשר $\text{lazy decrease keys} = \text{True}$, (פיבונאצ'י ובינומית עם ניתוקים) הציגו ביצועים טובים יותר. זאת מכיוון שפעולת decrease key מתבצעת ב $O(1)$ לשיעורין לעומת הערימה הבינומית והערימה העזלה שבהן מבצעים את פעולת heapifyUp שלוקחת $O(\log n)$ לשיעורין. ניתן לראות בטבלה כי מספר החיתוכים שביצענו בערימות (בערך 750,000) קטן בהרבה ממספר פעולות ה heapifyUp שביצענו (בערך 4,150,000) ובנוסף לכך שפעולה זאת יקרה בהרבה גרם להבדל הגדול בביצועים.

בנוסף גם כאן ניתן לראות הבדלים בין הערימות אשר ביצעו lazy meld הנובעים מהחלק הראשון של הניסוי שזהה לניסוי 1.

בניסוי השלישי, באופן דומה לניסוי 1, הערימות עם הביצועים הטובים יותר היו פיבונאצ'י ועזלה. זאת מכיוון שמספר החיתוכים, heapify up זניח ביחס לכמות ה Linking כפי שניתן לראות בטבלה, לכן הערימות עם lazy meld הציגו ביצועים טובים יותר.

ב. תוצאות הניסוי בסעיף 4 מאששות את התוצאות בסעיף 3.

ראשית, נשים לב שלפי ההערות, מימוש הקוד לערימה בינומית מתנהג כמו ערימה בינומית אבל הביצועים שלו אינם זהים לערימה בינומית. לכן בניסויים שביצענו זמני הריצה של ערימה בינומית גבוהים מהמצופה.

בניסוי הראשון אכן ראינו כי ערימות פיבונאצ'י ועזלה היו מהירות יותר כפי שציינו בסעיף 3. כמו כן ראינו כי ערימה בינומית עם ניתוקים הגיעה לביצועים הנמוכים ביותר. לעומת זאת זמני הריצה הנמוכים של ערימה בינומית בניסוי נובעים מהמימוש בקוד ולכן אינם תואמים את סעיף 3.

בניסוי השני אכן הגענו לסדרי גודל דומים יותר בין הערימות ובניסוי השלישי כפי שצפינו ערימת פיבונאצי הגיעה לביצועים הטובים ביותר (עם הסתייגות על בינומית עצלה).

ג. ניתן לראות כי בכל הניסויים העלות המקסימלית לפעולה בפיבונאצי ועצלה היא $O(n)$ לעומת בינומית ובינומית עם ניתוקים שם העלות המקסימלית לפעולה קטנה משמעותית. זאת מכיוון שבערימות עם lazy meld = True פעולה המחיקה הראשונה הינה הפעולה הכבדה ביותר והיא משלמת על כל הפעולות הזולות שביצענו עד כה. למרות זאת, זמני הריצה שלהן קצרים יותר משל שתי האחרות, שכן העלות לשיעורין שלהן נמוכה יותר כפי שנלמד בהרצאה.