נושא חדש מסתורי וחדש

איזה קטע, הוא שוב פעם עשה את הבדיחה הגרועה הזו

בעיית מבני נתונים

- בהינתן מערך A בגודל n , נרצה מבנה נתונים התומך בפעולות הבאות: (מתנצלים מראש על הלעז, הרבה יותר קריא)
- $Init(A, n)$ – initialize and build the DS. $O(n)$ time.
- $Sum(i, j)$ - Returns the sum of $A_i + A_{i+1} + \cdots + A_j$ (may assume $i < j$). $O(1)$ time.

- דרישת המקום למבנה: $O(n)$. (כיאה לתכנות תחרותי, נרצה גם קבועים נמוכים ככל הניתן).

פתרון בעיית מבני נתונים

- המערך לא משתנה – כלומר ניתן לשמור את הסכום של כל רישא במערך בגודל $n+1$ המקיים:

$$P[0] = 0, \qquad P[i] = P[i-1] + A[i-1] \qquad \forall i \geq 1$$

ובזאת סיימנו את הבנייה.

בוודאי ש:

$$Sum(i, j) = P[j] - P[i-1]$$

טוויסט בעלילה

- שימו לב שהפתרון הקודם עובד כל כך טוב רק בגלל שהמערך לא משתנה. אם ננסה לשנות את המערך A במיקום i ניאלץ לעשות $n-i+1$ שינויים למערך P וזו אינה דרך חכמה לעשות זאת.

- אז שינוי לבעיה המקורית יגרור באולי סיבוכיויות שונות לשאר הפעולות.
- כיאה למדעי המחשב, מה אם נאפשר פקטור נוסף של $\log n$?

בעיית מבני נתונים 2

- בהינתן מערך A בגודל n , נרצה מבנה נתונים התומך בפעולות הבאות: (מתנצלים מראש על הלעז, הרבה יותר קריא)

- $Init(A,n)$ – initialize and build the DS. $O(n)$ time.
- $Sum(i,j)$ - Returns the sum of $A_i + A_{i+1} + \cdots + A_j$ (may assume $i < j$). $O(\log n)$ time.
- $Change(i,k)$ - Changes $A[i] \mathrel{+}= k$. $O(\log n)$ time.

- דרישת המקום למבנה: $O(n)$. (כיאה לתכנות תחרותי, נרצה גם קבועים נמוכים ככל הניתן).

אבחנה ראשונה

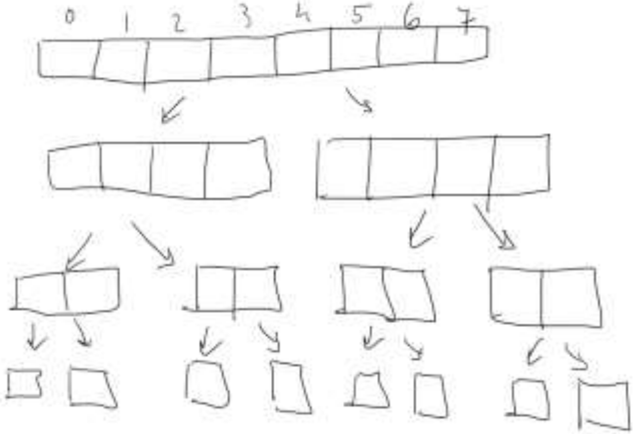
- <u>טענה:</u> לכל $i < j$ ניתן לפרק את $[i,j]$ ל- $O(\log n)$ מקטעים שכל אחד הוא בגודל חזקת 2.
- הסבר: נביט על $k = j - i + 1$, בפרט $(k)_2$ כלומר הייצוג הבינארי של k . ניתן לפרק את k לגדלים של חזקות 2 ע"פ אם הביט דולק או לא. (להוכחה פורמלית יותר – אינדוקציה חזקה).

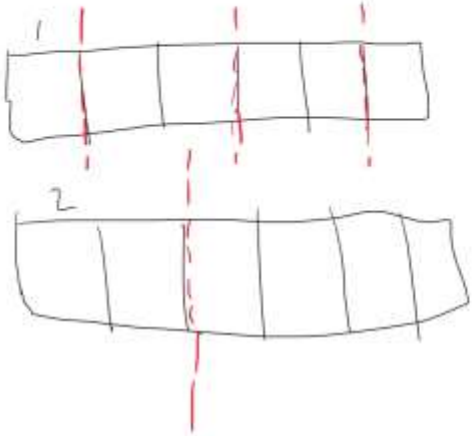
- $(14)_2 = 1110 \quad \Longrightarrow \quad 14 = 2^3 + 2^2 + 2^1 + 0 = 8 + 4 + 2 + 0$

אבחנה שנייה

- <u>טענה:</u> יתרה מכך אותו k יכול להתפרק ל - $O(\log n)$ חזקות 2 בסדר כך שכל בלוק בגודל B מתחיל באינדקס l המקיים $l \equiv 0 \ \ (mod\ B)$.
- <u>הוכחה:</u> נשים לב כי עבור i,j אם ניקח את b הגדול ביותר כך ש $i + 2^b \equiv 0\ \left(mod\ 2^b\right)$, בוודאי

$i \equiv 0\ \left(mod\ 2^b\right)$ וגם קיים פירוק ל $[i + 2^b, j]$ ע"פ אינדוקציה על הגודל.



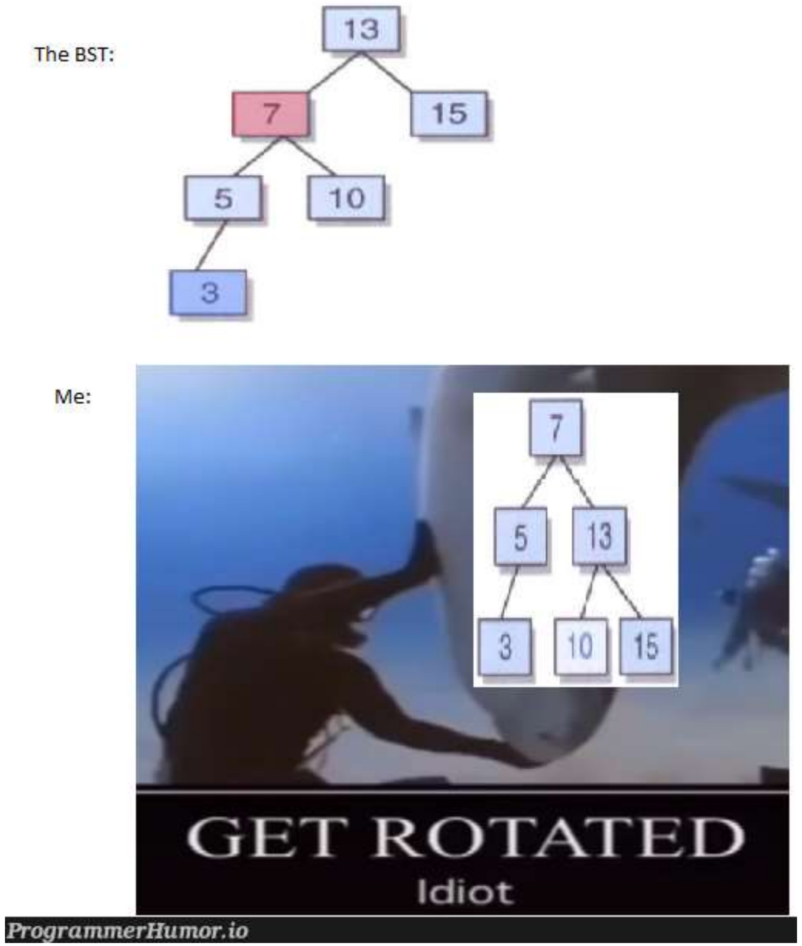


זה באשר לפירוק עצמו, אבל הוכחה זו אינה מוכיחה שכמות הקטעים היא $O(\log n)$ - תרגיל לכיתה (אונימודליות (שם מפואר ל-"גבעה").

עץ סגמנטים

Segment Tree

הרעיון

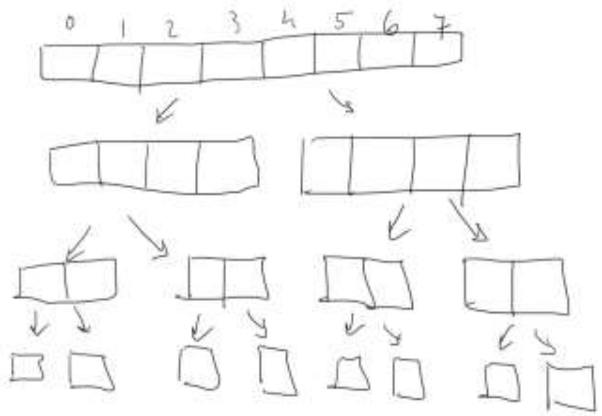


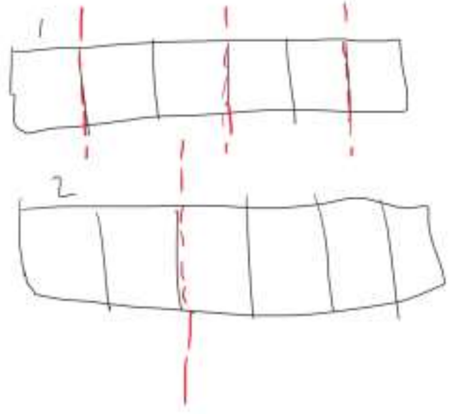
- נשתמש במבנה נתונים שהוא מייצג בדיוק את העץ מאבחנה 2.

כלומר משהו כזה:

הרעיון

- נשתמש במבנה נתונים שהוא מייצג בדיוק את העץ מאבחנה 2.
- סליחה, התכוונתי משהו כזה





- באופן זה, כל שינוי שאנחנו נעשה ישפיע

רק על $O(\log n)$ ערכים (שנמצאים במסלול לסינגלטון

של האינדקס), וכל שאילתא תהיה מורכבת רק מ - $O(\log n)$ איברים שאת כולם אנחנו חישבנו כבר.

מבחינת זמן בנייה ומקום – לאחסן את העץ ייקח $2n$ צמתים, ולבנות אותו ייקח $O(n)$ שכן עוברים על כל צומת פעמיים – לחישוב הערך שלו, ולחישוב הערך של אבא שלו

למעשה

- במהלך כל האבחנות שעשינו לא הזכרנו את השימוש בסכום אפילו לא פעם אחת – האם התבלבלנו?
- עצי סגמנטים הם כלי מאוד חזק – שעובד על כל פעולה אסוציאטיבית (כלומר כזו שאפשר להעלים לה את הסוגריים בלי לשנות את התוצאה $((a+b)+c=a+(b+c))$.
- הם כלי כל כך חזק שאפילו זה לא תנאי הכרחי, מספיק לדעת איך לחבר בזמן טוב את הפתרון לשני מקטעים סמוכים.

Code No.1

More general code might be added to the website shortly.

```cpp
vi a;
struct sum_segment_tree{
    int l, r, m;
    struct sum_segment_tree *left, *right;
    int value;

    sum_segment_tree(int L, int R) : l(L), r(R), m((l+r)/2), left(nullptr), right(nullptr){
        if(l < r) {
            left = new sum_segment_tree(L, R: m);
            right = new sum_segment_tree( L: m+1, R);
            value = left->value + right->value;
        }
        else if (l == r){
            value = a[l];
        }
    }
    void update(int i, int k){
        value += k;
        if(l == r) return;
        if(m < i) right->update(i, k);
        else left->update(i, k);
    }
    int query(int L, int R){
        if(R < l || r < L) return 0; //
        if(L <= l && r <= R) return value;
        return left->query(L, R) + right->query(L, R);
    }
};
```

הפסקה של 10 דקות

לפני ההפסקה דיברנו על

- עץ סגמנטים

- ספציפית עדכונים נקודתיים (במקום יחיד) .
- ושאילתות טווח (טווחיים).

בעיית מבני נתונים 3

- בהינתן מערך A בגודל n , נרצה מבנה נתונים התומך בפעולות הבאות: (מתנצלים מראש על הלעז, הרבה יותר קריא)

- $Init(A, n)$ – initialize and build the DS. $O(n)$ time.
- $Sum(i, j)$ - Returns the sum of $A_i + A_{i+1} + \cdots + A_j$ (may assume $i < j$). $O(\log n)$ time.
- $Change(i, j, k)$ - Changes $A[i, \ldots, j] \mathrel{+}= k$. $O(\log n)$ time.

- דרישת המקום למבנה: $O(n)$. (כיאה לתכנות תחרותי, נרצה גם קבועים נמוכים ככל הניתן).

פתרון בעיית מבני נתונים 3

- עכשיו נוכל שוב לפרק את התחום $[i,j]$ למקטעים שחישבנו את התוצאה, ונשים לב שבהינתן טווח שכולו נמצא בתוך טווח העדכון – אני יודע בוודאות שהסכום שלו גדל ב- $k \cdot (r - l + 1)$, ולכן זהו השינוי שנצטרך לבצע לקוד.

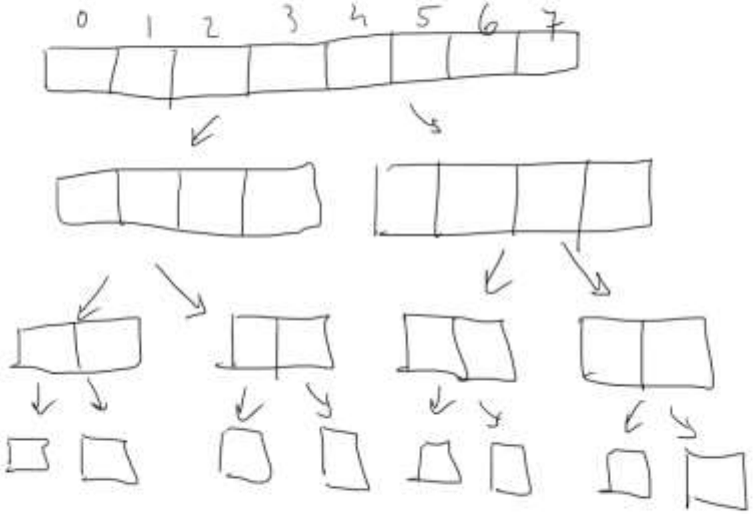
```
void update(int L, int R, int k){
    if(R < l || r < L) return;
    if(L <= l && r <= R) value += k * (r-l+1);
    else left->update(L, R, k), right->update(L, R, k);
}
```

- המבנה הזה מאוד נוח והוא גם מאוד קל ואינטואיטיבי לכתיבה!

מי הצליח להבין ששיקרנו?

- בהצבעה בבקשה

לא על הכל אבל



- אז תחשבו מה קורה אם עדכנו מקטע גדול, ואז היינו צריכים את הילדים שלו.
- כאן מגיעה הטעות – אבל גם השינוי במחשבה.

- מה אם נפרק את החישוב של כל ערך לכדי חישוב של סכום המסלול שלו?
- כך נוכל לתמוך בשאילתות על מספר אחד, אבל לאפשר נכונות בעדכונים של טווחים:

```
sum_segment_tree(int L, int R) : l(L), r(R), m((l+r)/2), left(nullptr), right(nullptr){
    if(l < r) {
        left = new sum_segment_tree(L, R: m);
        right = new sum_segment_tree( L: m+1, R);
        value = 0;
    }
    else if (l == r){
        value = a[l];
    }
}
```

```
void update(int L, int R, int k){
    if(R < l || r < L) return;
    if(L <= l && r <= R) value += k;
    else left->update(L, R, k), right->update(L, R, k);
}
int get(int i){
    if(l == r) return value;
    return value + (m < i ? right->get(i) : left->get(i));
}
};
```

אבל רגע...


- ביקשתם מאיתנו לעשות עדכונים על טווחים ושאילתות על טווחים, לא שאילתות על אינדקסים!
- נשאר כתרגיל לקורא

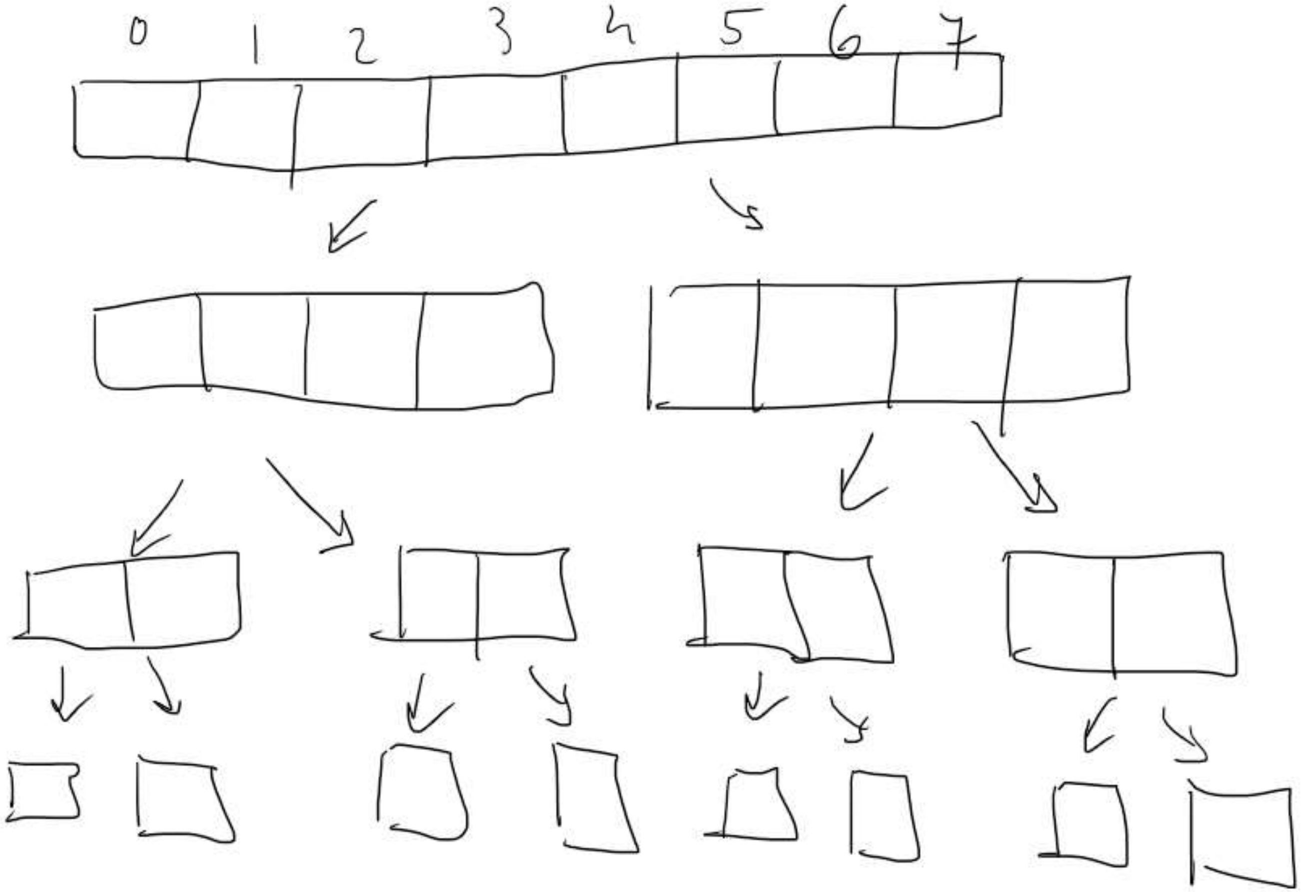
סתם

- הבעיה הזו באמת מורכבת משגרמנו לה להראות – וכל הכאב ראש הזה גורם לי לרצות פשוט להפסיק.
- אני עצלן.
- וזה הפתרון.

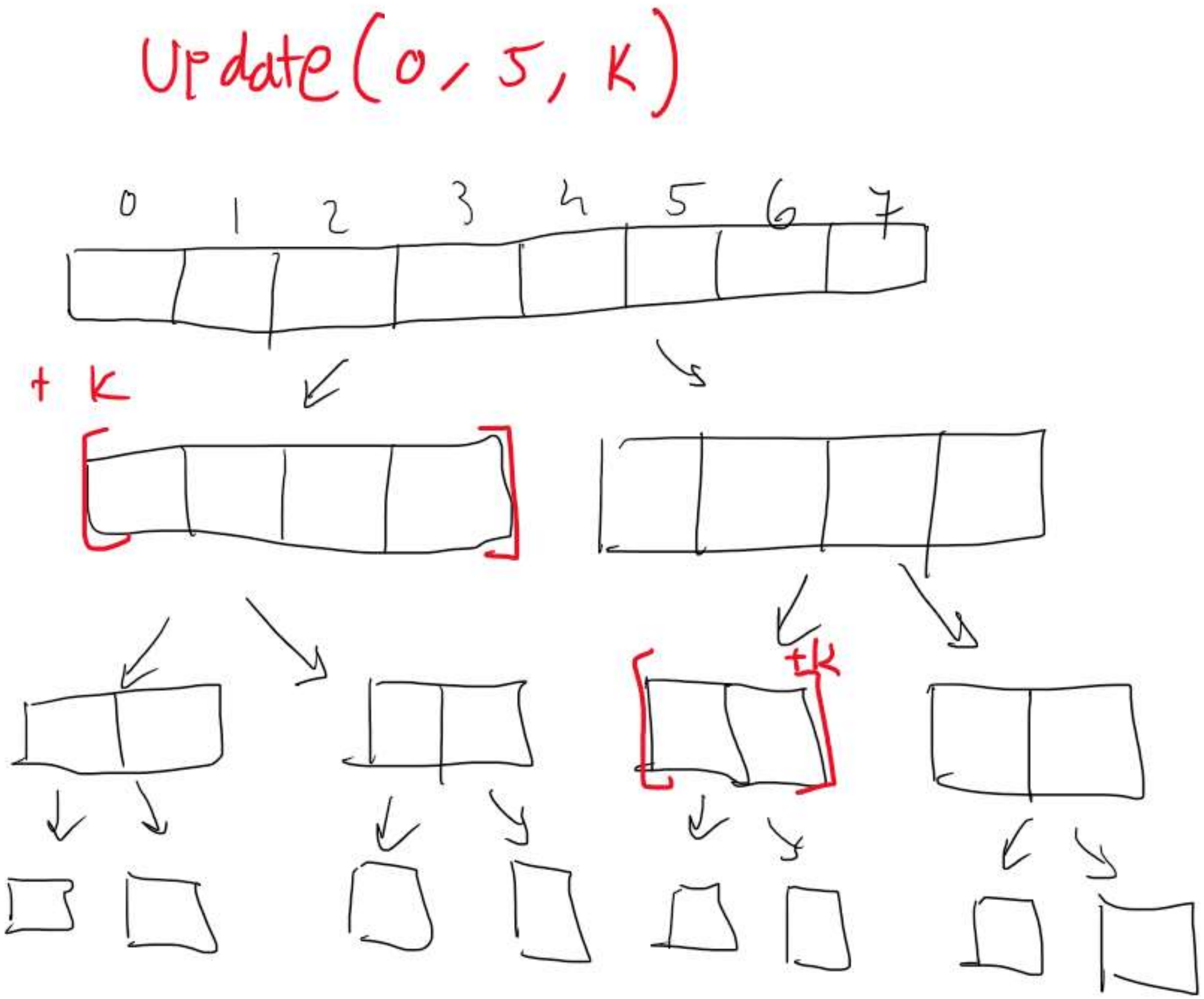
Lazy propagation

Segment Tree Kashe

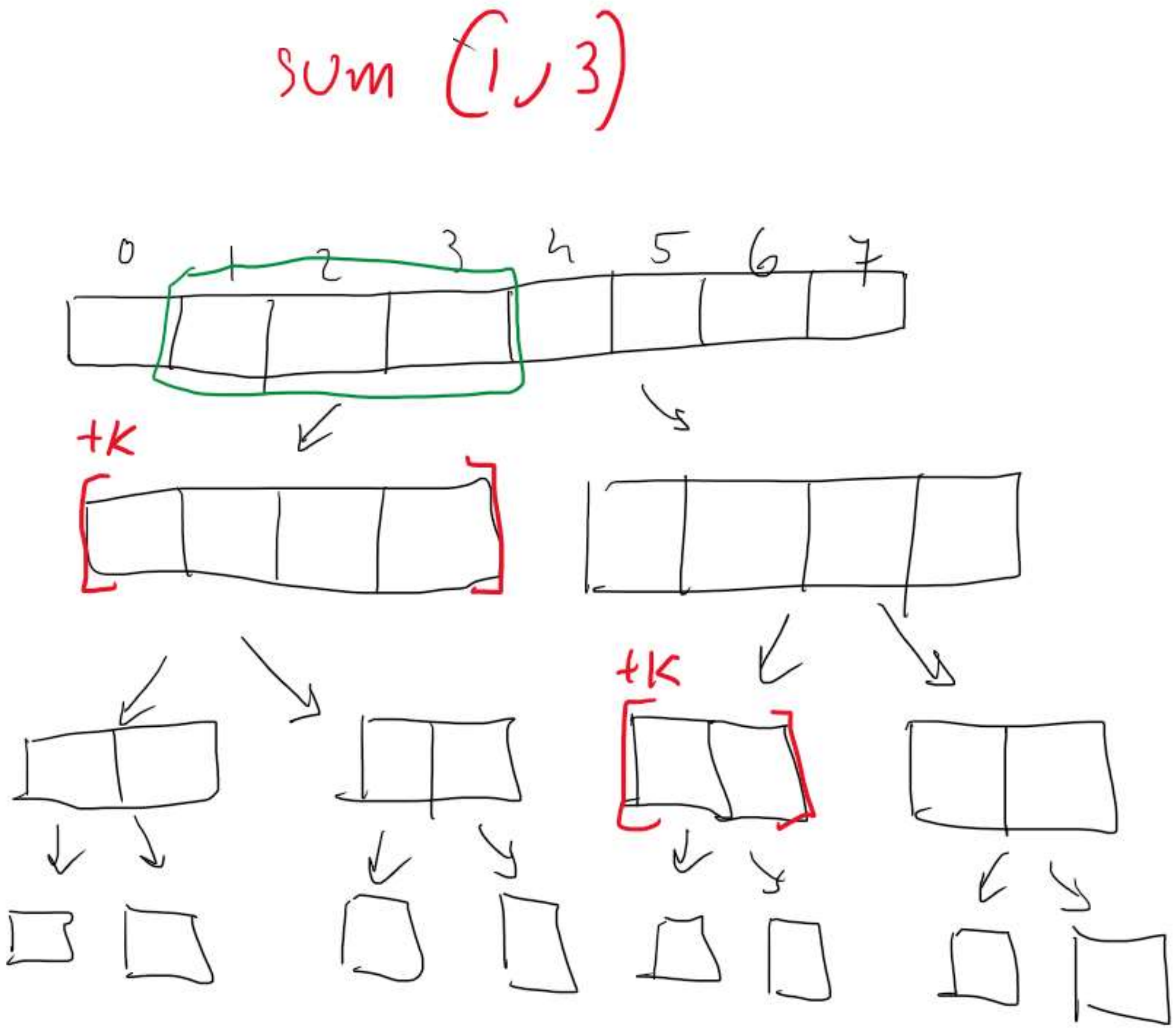
עצלנות לשמה



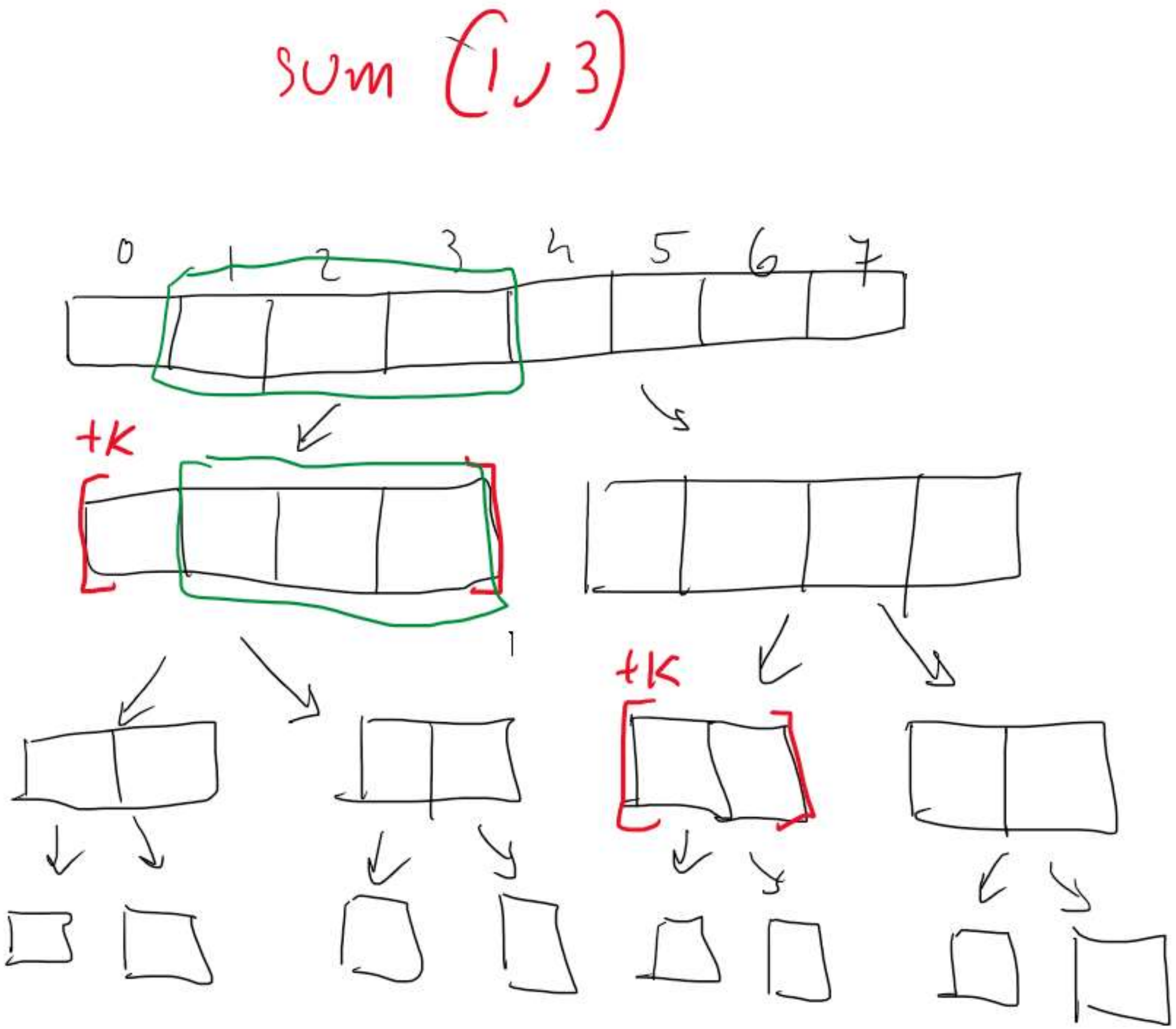
עצלנות לשמה



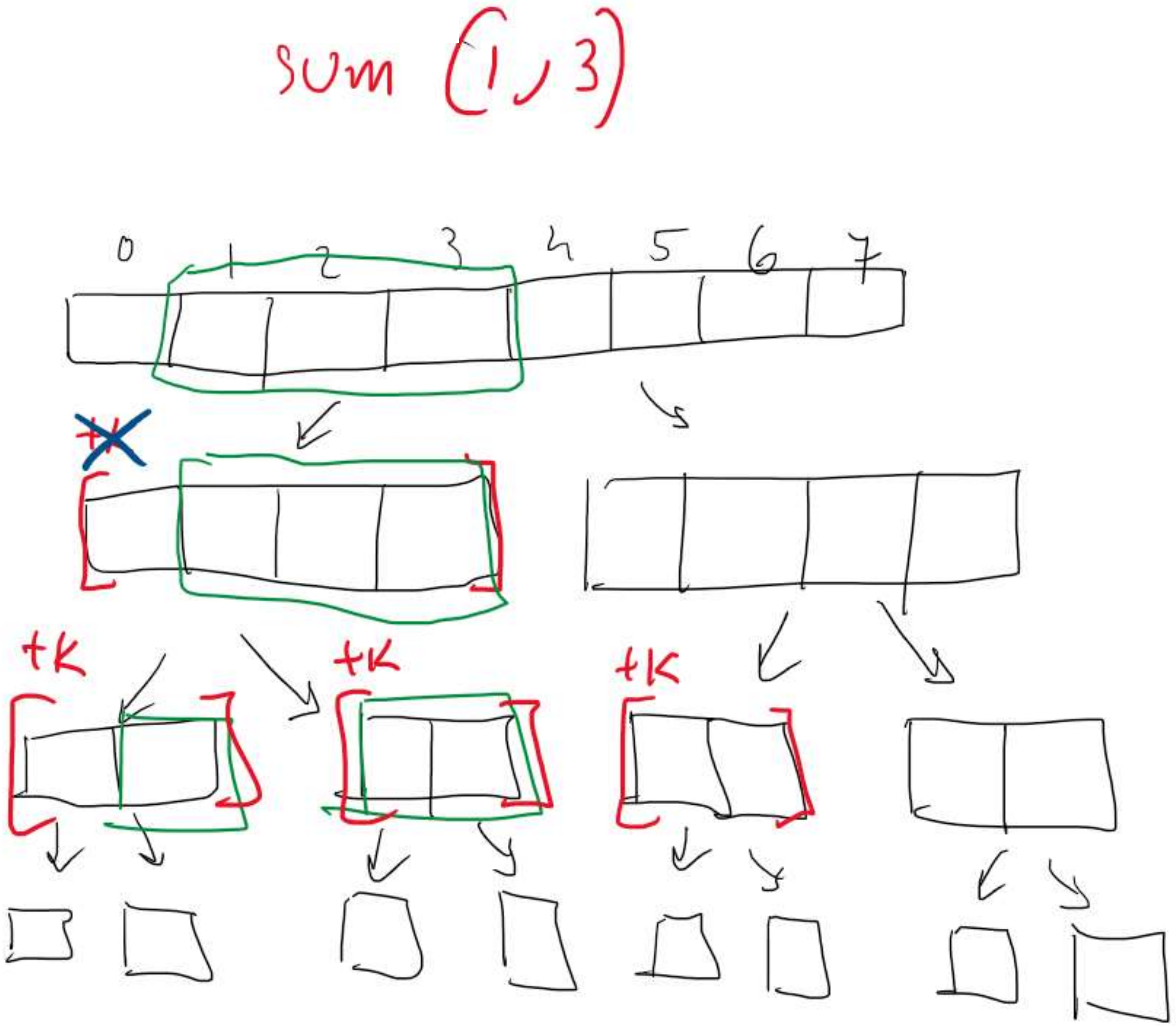
עצלנות לשמה



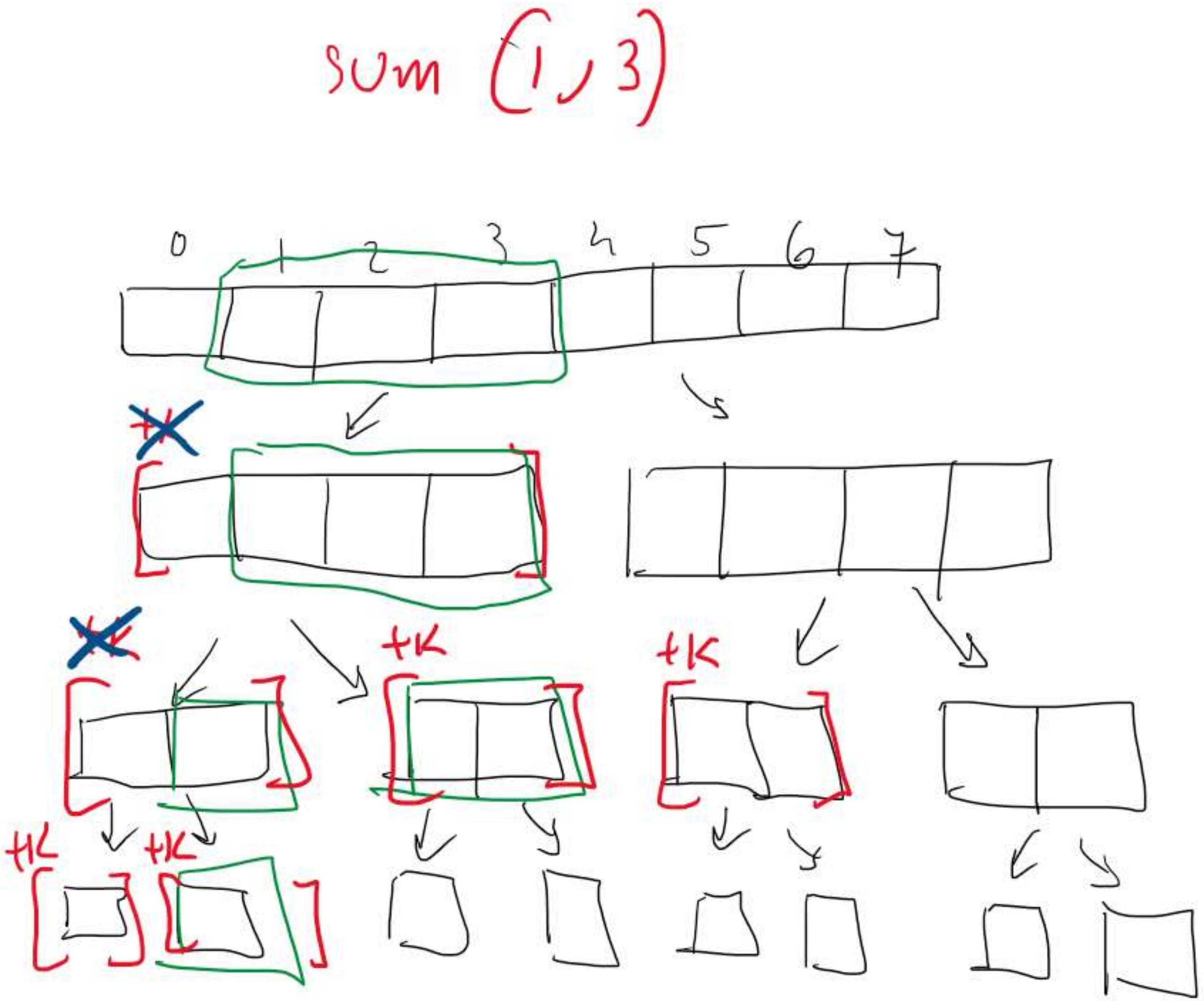
עצלנות לשמה



עצלנות לשמה



עצלנות לשמה



Push down

- על כל פעולה שנעשה, נרצה לדחוף את העדכון לבנים תוך עדכון הצומת הנוכחי.
- צריך לזכור לאפס את העדכון של האב, ולשמור ערך בשם lazy שיהיה "ערך השינוי של הטווח בצומת הזה".
- כך הפעולה של דחיפה למטה תתבצע ב $O(1)$ ע"י לעדכן את lazy של צד ימין ושל צד שמאל, ואתחול של הנוכחי.
- פעולה זו תיקח $O(1)$ וסה"כ עוברים בעץ ב - $O(\log n)$ צמתים

ולכן הסיבוכיות לפעולות נשארת $O(\log n)$.

- הקוד לכך:

```
void pushdown(){
    left->value += lazy;
    left->lazy += lazy;
    right->value += lazy;
    right->lazy += lazy;
    lazy = 0;
}
```

Pull Up

- בדרך חזרה ברקורסיה, חשוב לוודא שאנחנו משתמשים בvalue שהוא נכון ומעודכן – ולכן נמשוך למטה מהבנים שלו את הvalue שלהם ונחשב את של הצומת הנוכחי.
- בכניסה לרקורסיה נדחוף למטה, וביציאה ממנה נחשב את הערך של האב מהבנים שלו – במילים אחרות נדחוף למטה לפני שנרד למטה, נמשוך למעלה לפני שנעלה למעלה.

```
void pullup(){
    value = left->value + right->value;
}
```

השינויים הנוספים

- אז הוספנו שדה של lazy. בנוסף שינינו את update, sum באופן הבא:

```
void update(int L, int R, int k){
    if(R < l || r < L) return;
    if(L <= l && r <= R) {
        value += k;
        lazy += k;
        return;
    }
    pushdown();
    left->update(L, R, k);
    right->update(L, R, k);
    pullup();
}
```

```
int sum(int L, int R){
    if(R < l || r < L) return 0;
    if(L <= l && r <= R) return value;
    pushdown();
    int x = left->sum(L, R);
    int y = right->sum(L, R);
    // pullup();
    return x+y;
}
```

כאמור – זה מבנה נתונים מאוד חזק

- אפשר לתמוך הרבה פעולות, והזמן הוא worst-case , לא לשיעורין.
- אפשר לתמוך בהרבה פעולות, והרבה פעמים גם במקביל (כלומר גם עדכונים על טווחים כתוספת וגם כassignment.
- כדוגמה (תוכלו בוודאות להשתמש בשאילתא אחד ועדכון אחד ולתחזק אותם נכון (אולי גם יותר)):

- Sum
- Max
- Min
- GCD

- [l, r] = k
- [l, r] += k
- XOR
- AND

- OR
- MSB
- etc/.

בעיית מבני נתונים 2

- בהינתן מערך A בגודל n , נרצה מבנה נתונים התומך בפעולות הבאות: (מתנצלים מראש על הלעז, הרבה יותר קריא)

- $Init(A, n)$ – initialize and build the DS. $O(n)$ time.
- $Sum(i, j)$ - Returns the sum of $A_i + A_{i+1} + \cdots + A_j$ (may assume $i < j$). $O(\log n)$ time.
- $Change(i, k)$ - Changes $A[i] += k$. $O(\log n)$ time.

- דרישת המקום למבנה: $O(n)$. (כיאה לתכנות תחרותי, נרצה גם קבועים נמוכים ככל הניתן).

פתרון בעיית מבני נתונים 2 אבל אחרת

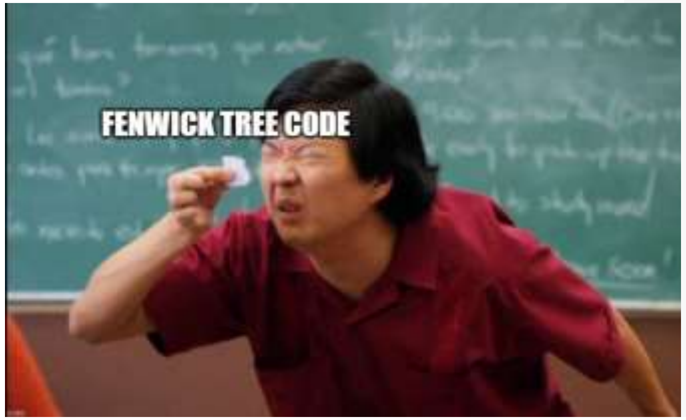
- אולי נאחסן את הסכומים בצורה שונה, כלומר:

$$T_i = \sum_{j=g(i)}^{i} a_j$$

דבר כזה יכול להראות משהו כמו:

$$a_0, a_0 + a_1, a_2, a_0 + a_1 + a_2 + a_3$$

ברור שמספיקים לנו n משתנים בלתי תלויים כדי להרכיב את הסכומים הנכונים באיזושהי דרך, השאלה היא איך בצורה נוחה ויעילה?



Fenwick Tree

BIT = Binary Indexed tree

הגדרת $g(i)$

- ניתן לבחור כל מיני $g(i)$ ועבור חלק מהן המבנה יעבוד בסדר. במגבלות הקורס נעסוק ב $g(i)$ ספציפית:
- $g(11) = g(1011_2) = 1000_2$
- $g(10) = g(1010_2) = 1010_2$
- $g(7) = g(111_2) = 000_2$

- מי רוצה לעלות על הרצף?

- $g(i) = \begin{cases} i & LSB(i) = 0 \\ i\ w/o\ leading\ 1s & else \end{cases}$

מתברר ש...

- ניתן לאחסן את הסכומים האלה ע"י משחקי ביטים חכמים: (לא באמת רלוונטי להיכנס לפרטים).

```cpp
struct FenwickTree {
    vector<int> bit;  // binary indexed tree
    int n;
    FenwickTree(int n) {
        this->n = n;
        bit.assign(n,  val: 0);
    }
    FenwickTree(vector<int> const &a) : FenwickTree( n: a.size()) {
        for (size_t i = 0; i < a.size(); i++)
            add( idx: i,  delta: a[i]);
    }
```

```cpp
    int sum(int r) {
        int ret = 0;
        for (; r >= 0; r = (r & (r + 1)) - 1)
            ret += bit[r];
        return ret;
    }
    int sum(int l, int r) {
        return sum(r) - sum( r: l - 1);
    }
    void add(int idx, int delta) {
        for (; idx < n; idx = idx | (idx + 1))
            bit[idx] += delta;
    }
};
```

אבל אבל   

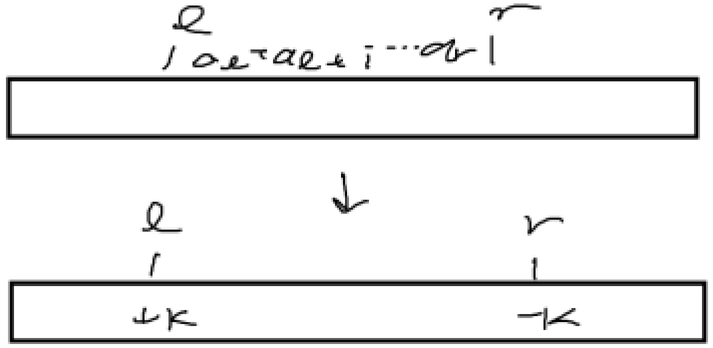
- למה שנשתמש בזה אם אנחנו כבר יודעים seg tree?
- קוד הרבה יותר קריא, ומכוון בדיוק למה שצריך. בנוסף העובדה שסוכמים רישא מאפשרת לעשות רעיונות מאוד יפים (ויותר קל להסביר עם מבנה חדש).

- אז recap קצר על הAPI (פונקציות של פנוויק):

- $Add(i,k) - A[i] \mathrel{+}= k$
- $Sum(j) - return\ sum\ of\ prefix\ j$

שאלות נקודה ועדכוני טווח

- אנחנו יודעים לבצע שאלות טווח ועדכוני נקודה. מה אם נשתמש בסכום בשביל הערך, ובערך בשביל הסכום?
- נאתחל את כל המערך ל0.
- כשנרצה לחשב ערך במיקום i נחשב אותו ע"י הסכום של $a_0 + \cdots + a_i$.
- כשנרצה לעדכן טווח ולהוסיף לו k , נעשה זאת ע"י להוסיף k לאינדקס של l ולהוריד k מהאינדקס של $r + 1$.



- ניתן לתחזק אפילו שני עצי פנוויק, אחד מאותחל ל-0 ויטפל בעדכונים, אחר מאותחל למערך ויטפל בבקשות הסכום. ניתן לתחזק את זה אבל לא במסגרת הקורס.

רעיונות נוספים למתעניינים

- Sqrt decomposition – mo’s algorithm.
- Offline queries.
- Array segment tree.

- There are more but we do not recommend to dive into them at this stage.

בואו נפתור שאלה בסיפי