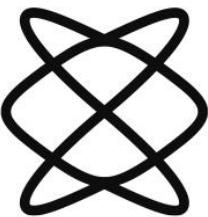


החולג למדעי המחשב (0368)  
מכוא מורהח למדמ"ח (1105)  
(גרסה ארוכה)

מרצה: דר' מיכל קליננברט, פרופ' אלחנן בורנשטיין  
מתרגל: נעם פרזנט'בסקי / עמרי פורת  
תשפ"ב, סמסטר ב' (2022)

מסכם: רועי מעין



The Raymond and  
Beverly Sackler Faculty  
of Exact Sciences  
Tel Aviv University

## A – יסודות פיתון

4.....	תכנות בסיסי
5.....	מודל היזכרון
7.....	דקדוקים פורמליים ותהליך הפירוש של פיתון
8.....	פונקציות למבדא ופונקציות סדר גובה
8.....	תכנות נכון וסוגי שגיאות
8.....	אקראיות

## B – ייצוג טיפוסי מידע

9.....	ייצוג טבעיים (int)
10.....	ייצוג ממשיים (float)
11.....	ייצוג תווים (Unicode ASCII)

## C – אלגוריתמים בסיסיים וסיבוכיות

12.....	חיפוש
12.....	מיון
13.....	מיוג
13.....	סיבוכיות ו-notation O notation

## D – רקורסיה

15 .....	מבוא
15 .....	חיפוש ומיון
17 .....	מגדלי האנו
18 .....	מומאייזציה
20 .....	Munch
22.....	פתרון בעיות רקורסיביות (תרגולים 6-8)

## E – נושאים בתורת המספרים

27 .....	העלאה בחזקה
30 .....	בדיקות ראשוניות
31 .....	Diffie-Hellman
32 .....	GCD

## F – חישוב נומי

32 .....	מציאת שורש
----------	------------

32 .....	נגזרות וaintegrals
32 .....	קירוב לפאי

## G – OOP ומבני נתונים

33 .....	מבוא
34 .....	רשימות הקשורות
36 .....	עצי חיפוש ביןאריים
38 .....	Hashtables
40 .....	גנרטורים

## H – טקסט

43 .....	אלגוריתם CYK
45 .....	דჩיסט האפמן
49 .....	דჩיסט למפל-זיו

## I – קודים לגילוי ולתיקון שגיאות

51 .....	דוגמאות בסיסיות
51 .....	מבחן האמיניג
53 .....	קוד חרצה
54 .....	בית זוגיות
55 .....	קוד האמיניג

## J – ייצוג ועיבוד תמונה

57 .....	ייצוג תמונה דיגיטלית
57 .....	ኒקי רעש

## 1 – פיתון

### A – יסודות פיתון

**תכנות בסיסי**

טיפוסים:

- מחזרות אפשר לכתוב עם " וגם עם ' אחד, זה אותו דבר.
- string multiplication – באשר מכפילים ב-0 נקבל מחזרות ריקה, כאשר מכפילים ב-float נקבל `TypeError`.
- יש אופרטור של `//` - חולקה שהיא לא שארית (עיגול לפני מטה), מחזרה זו בניגוד לחולקה וגילתה שמחזרה float הוא לא היכן מדויק שיש, זה קירוב כלשהו שנגמר בכמה ספרות אחרי הנקודה.
- בוליאנים – True מיוצג על ידי 1, בעוד False מיוצג על ידי 0.

אופרטורים:

טבלת סדר קידימות האופרטורים שראים בתרגול (מקדימות גובהה בראש הטבלה לנמוכה בתחתית הטבלה):

אופרטורים המופיעים באותה השורה הינם בעלי אותה קידימות וכן סדר השיעורם שלהם יהיה משmaal לימין

**	העלאה בחזקה
<code>*</code> , <code>/</code> , <code>//</code> , <code>%</code>	כפל, חילוק, שארית
<code>+</code> , <code>-</code>	חיבור, חיסור
<code>&lt;</code> , <code>&lt;=</code> , <code>&gt;</code> , <code>&gt;=</code> , <code>!=</code> , <code>==</code>	אופרטורים השוואתיים
<code>not x</code>	Boolean NOT
<code>and</code>	Boolean AND
<code>or</code>	Boolean OR

השוויה לקסיקוגרפיה בין מחזרות:

כאשר משווים בין מחזרות בשיטות ההשוואה היא לקסיקוגרפיה, הם החליטו להיות פורמליים ולרשום את זה בצורה הבאה:

- Given 2 strings over some totally ordered alphabet

$$S = s_0s_1 \cdots s_{n-1} \text{ and } T = t_0t_1 \cdots t_{m-1},$$

$S < T$  if and only if

there exists some  $i \geq 0$  such that

(1) for every  $0 \leq j < i$  we have  $s_j = t_j$  and

(2) either  $s_i < t_i$  or  $i = n < m$ .

15

משפט תנאי ולולאות:

- if, else, elif
- while, for
- break ו-continue ולא הסתייגו שזה לא כזה יפה להשתמש בה (פחות ככמה חונכתי – bad practice).

אפשר לבצע איטרציות לא רק על `range`, גם על `str` ועל `list`.

:range

– אפשר לשנות על ה-`diff` שבאמצעותם מתקדמים ב-`range` •

`range(n)` defines the sequence  $0, 1, 2, \dots, n - 1$

More generally, `range` can create **arithmetic progressions**: given three integers  $a, b, d$  with  $d > 0$ , `range(a, b, d)` contains all integers of the form  $a + i \cdot d$ , satisfying  $a \leq a + i \cdot d < b$  ( $i > 0$ ).

– So `range(a, b)` is a shorthand for `range(a, b, 1)`.

זכור כי בעבר היה נהוג להשתמש ב-`range` אבל מסתבר שהגדרה `range` היא פשוטה יותר והוא יודע ליצר כל פעם את האיבר הבא באוסף, הוא לא מייצר ישר את כמה האיברים שניתנו לו כפרמטר.

:list

נשים לב להבדל – עברו רשימה ריקה שהגדרכנו  $L = [ ]$ .

- הפעולה  $[i] = L[i]$  גורמת לכך שתווסף רשימה חדשה בזיכרון, ואז מתבצעת השמה למשתנה  $L$ .

- הפעולה  $[i] = L.append(i)$  וכך פשוט מתווספים איברים לרשימה.

slicing זה חשוב – יש אפשרות דוגמאות במצגת לכל מני ווריציות. אפשר לבצע slicing גם על רשימות וגם על מחרוזות.

```
num_list = [11, 12, 13, 14, 15, 16, 17, 18, 19, 20]
```

```
>>> num_list[1:5]      slicing
[12, 13, 14, 15]
```

```
>>> num_list[0:10:2]   slicing an arithmetic progression
[11, 13, 15, 17, 19]
```

```
>>> num_list[::-2]     shorthand for previous slicing
[11, 13, 15, 17, 19]
```

```
>>> num_list[::-1]     reversing the list
[20, 19, 18, 17, 16, 15, 14, 13, 12, 11]
```

פונקציות:

• פונקציות שלא מחזירות שום דבר יחזירו את הערך המקורי `.None`.

**מודל הזיכרון**

נבחן בין `equality` (שווין) ל-`identity` (זהות) עברו שני משתנים:

- `equality` (שווין) – המשתנים מצבעים לאובייקטים בעלי **אותו ערך**. שווין אנו בודקים באמצעות האופרטור `==`.

- `identity` (זהות) – המשתנים מצבעים לאותו האובייקט בזיכרון, ככלומר הם מפנים **אותה כתובות בזיכרון**. זהות אנו בודקים באמצעות האופרטור `is`, או באמצעות הפונקציה `id`.

הפונקציה `id` – מחזירה את הכתובת המשתנה מפנה אליה בזיכרון. נשים לב לשיקולות הבאה:

```
id(object1) == id(object2) if and only if object1 is object2
```

- באופן כללי, בשתי ריצות שונות, אותו האובייקט יכול לקבל בתוצאות שונות.

- רצוי לעבור עם בסיס 16 בתצוגה של פלט הפונקציה, כיון שגם כתובות בזיכרון.

- מסיבות של אופטימיזציה, עבור ערכים שהם בשימוש גבוה (מספרים שלמים קטנים, true/false, נפוצים) פיתוח מזכה להם **מקום קבוע בזיכרון** ברגע שה-shell עולה. יש להם כתובות קבועות. כך ניתן מנצח שבו כל פעם שנבצע פעולה עם המספר 1 למשל, תיווצר כתובות חדשה בזיכרון.

השנה:

- השמה למשתנה – כאשר אנו מבצעים השמה חדשה למשתנה, נוצר קודם כל האובייקט באגף ימין **במקום חדש בזיכרון**, ולאחר מכן המשתנה מצביע למקום שבו נוצר הערך הנ"ל.
- השמה בין משתנים – לא נוצר ערך חדש בזיכרון, והמשתנה הצד שמאל יפנה לאותו מקום בזיכרון שאליו מפנה המשתנה הצד ימין. זהה השמה בין מצביעים, אך לאחר ההשמה **שני המשתנים מצביעים לאותו המקום**.

ניתן לעקב אחרי התהילן בקהלות באמצעות [python tutor](#). בשורה התחתונה, **משתנה הוא רק שם זמני לכתובת בזיכרון**. כתובות מסויימת לא גוררת ערך מסוים, וערך מסוים לא גורר כתובות מסוימת (חו"ץ מהמרקם החריגים של ערכים בשימוש גבוה).

:mutable vs. immutable

נבחין בין שני מושגים עבור טיפוס בזיכרון:

- **mutable** – ניתן למוטציה, אפשר לשנות את הערך שלו.
- **immutable** – לא ניתן למוטציה, או אפשר לשנות את הערך שלו.

	Ordered (sequence)		unordered	
	type	example	type	Example
<b>Mutable</b>	list	[1, 2, 3]	set dict	{1, 2, 3} {1: "a", 2: "b", 3: "c"}
<b>Immutable</b>	str range Tuple	"123" range(1, 4) (1, 2, 3)		

דוגמאות ליצירת רשימות (תרגול 3):

פעולה	תיאור	דיברון
<code>num_lst = [0]*10</code>	מייצר את האובייקט 0, ורשימה באורך 10 שכל האיברים בה מצביעים על אותו האובייקט 0.	
<code>nested_lst = [[]]*10</code>	במקום ליצור את האובייקט 0 ויצרנו רשימה ריקה, ורשימה באורך 10 שכל האיברים בה מצביעים על הרשימה הריקה.	
<code>nested_lst2 = [[ ] for i in range(10)]</code>	כל איבר ברשימה מצביע לרשימה אחרת. ואז כל רשימה יכולה להשתנות בנפרד.	

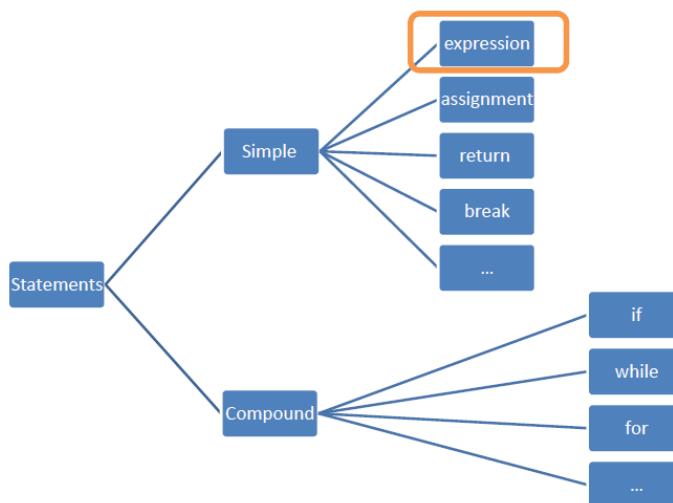
## דקדוקים פורמליים ותהליכי הפירוש של פיתון

מושגים בסיסיים:

- תחביר – מגדיר מהו מבנה של פקודה חוקית בשפה. התחביר מוגדר באמצעות דקדוק.
  - סמנטיקה – המשמעות של הibernית, מה התוצאה הצפיה בתוצאה מהריצה שלה. הגדרת סמנטיקה היא הרבה יותר מורכבת.
- דקדוק – מוגדר על ידי אלף-בית של השפה, משתנים (נדרש שייהי משתנה המחלה), וכלי דקדוק (חוקים). נאמר שמהירות נוצרת על ידי דקדוק מסוים, אם ניתן להפעיל את הכללים שוב ושוב עד שמתקבלת מחרוזת שמכילה רק תווים מהאלף-בית ולא מכילה משתנים בכלל. הקוד של פיתון עובר Parsing לפי כללי הדקדוק.

פקודות ב-Python מתחולקות לשני סוגים:

Simple statements	Compound statements
<ul style="list-style-type: none"> <li>• expression statements, e.g., <code>3+4**2</code></li> <li>• assignment statements, e.g., <code>res = 400</code></li> <li>• return statement <code>return res</code></li> <li>• break statement <code>break</code></li> </ul>	<ul style="list-style-type: none"> <li>• if statement <code>if a &gt; b:</code>     ... • while statement <code>while a &gt; b:</code>     ... • for statement <code>for a in lst:</code></li> </ul>



:expression

כל מה שיש לו ערך, יכול להיות בצד ימין של השמה: `res = <expression>`.  
ביטויים יכולים להיות משני סוגים:

- **anonymous** – ביטוי שעובר evaluation אבל לא נשמר בתוך משתנה. באן לא ניתן לבצע שימוש חוזר בערך.
- **named** – ביטוי ששמור בתוך משתנה. באן ניתן לבצע שימוש חוזר בערך.

## פונקציות למבدا ופונקציות סדר גובה

### פונקציות למבדא:

כל שמאפשר להגדיר פונקציות בצורה אונומית, באופן ישיר. **תחשיב למבדא** זהה לצורה שבה הגדרנו אותו בבודהה. לדוגמה:

*lambda x, y: x + y*

אפשר להפעיל את הפונקציה בצורה הבאה – זהה צורה אונומית, באופן ישיר. ערך החזקה לא נשמר למשתנה: (4)(2, 4).

### פונקציות סדר גובה:

פונקציית סדר גובה היא **פונקציה שמחזירה/מקבלת פונקציה**.

- **פונקציה שמחזירה פונקציה** – ראיינו בתרגול דוגמה לפונקציה `make` שמחזירה פונקציה שמחשבת את החזקה.
- **פונקציה שמקבלת פונקציה** – לפונקציה `sorted` ניתן להעביר פרמטר אופציונלי `key` שהוא פונקציה שלפיה יבוצע המילוי של הרשימה.

## תכון נכון וסוגי שגיאות

שלושת ה-C: נכונות (Correctness), יעילות (Cost), בהירות (Clarity).

### נכונות:

- **סוגי שגיאות – סינטקס** (קמפול), ריצה, סמנטיקה (האגאים שלא יודעים שיש לנו בתכנית).
- **דיבוג – בשיל שיטות:** `print`, `python tutor`, `UT`.

### בהירות:

1. **שמות שימושתיים** – קובננציות, שמות ברורים שמסבירים את עצם.
2. **שימוש בזיכרון** – צריכת זיכרון נכונה.
3. **מקרי קצה** – להימנע מבדיקות מקרים קצה שלא לצורך.
4. **מבנה איטרציה** – בחירת המבנה הפשוט והנכון ביותר למצב.
5. **שכפול קוד – להימנע!**
6. **פשטות – גם ויזואלית וגם לוגית.**
7. **מתחת למכתה המנוח –** בשימושים בכליים של פיתוח צריך להיזהר, בעיקר בשלא ברור מה מתבצע שם.
8. **тиיעוד – לא כדי לבסוט קוד גרווע.**

## אקרואיות

### מבוא:

תוצאה רנדומלית היא תוצאה שקשה לחזות אותה. הסתכלות על הסדרה עד מקום מסוים לא אומרת לנו דבר על המקום הבא. דוגמאות ליצירת ערכים רנדומיים:

- **TRNG** – ניסו להסתכל על תופעה הטבעייה רנדומלית – כמו פירוק דיאקטי.
- **PRNG** – אלגוריתם שמחולل מספרים שהם כמעט רנדומיים – קשה מאוד להזות שהם לא רנדומיים באמת. אולם, מתיישחו זה חוזר לאותו מספר ואז הסדרה חוזרת על עצמה.
- **Mersenne Twister** – מה שמשתמש בו. גם כאן משתמשים בערך התחלתי בלשחו (`seed`) רק שהוא מtabס על הזמן הנוכחי של המחשב.

### שימושים:

מושא שימושים בו הרבה במדעי המחשב. דיברנו על הדוגמה של הילוך מקרי (Random Walking).

## B – יצוג טריגרוני מיידע

### **יצוג טריגרוני (int)**

פעולות – עברו שני מספרים באורך  $n$  ביטים:

- חיבור מספרים בינאריים:
  - פעולות חיבור: במקרה הכى גרע  $2^n$
  - אורך מקסימלי של התוצאה:  $1 + n$
- כפל מספרים בינאריים:
  - פעולות (כפל וחיבור): במקרה הכى גרע  $2^n$  פעולות כפל וחיבור.
  - אורך מקסימלי של התוצאה:  $2n$ .

זהויות חשובות:

- $2^n$  מיוצג על ידי הרצף 1 ואחריו  $n$  אפסים. למשל  $2^3$  יהיה 1000.

יצוג בסיס  $d$ :

אפשר לייצג את המספר כפולינום שמקדמי קטנים מ- $d$ :

$$N = a_k \cdot b^k + a_{k-1} \cdot b^{k-1} + \dots + a_1 \cdot b + a_0$$

where for each  $i$ ,  $0 \leq i \leq k$ .

$$N_{\text{in base } b} = (a_k a_{k-1} \dots a_1 a_0)$$

**טענה:** המספר  $N$  מיוצג כפולינום מדרגה  $k$  בסיס  $b$  מקיים:  $b^k \leq N < b^{k+1}$ .

- באמצעות  $1 + k$  ביטים המספר הכى קטן שנוכל לייצג הוא  $2^k$  – הביט 1 ואחריו  $k$  אפסים.
- המספר הכى גדול שנוכל לייצג הוא  $1 - 2^{k+1}$  – הביט 1 ואחריו  $k$  אחדים. זה בדיקת המספר הבא ( $2^{k+1}$ ) פחות 1.

### **ערכים אפשריים של מספר בעל $n$ ביטים**

- נתון מספר טבעי  $N$  שביצוג הבינארי שלו יש בדיוק  $n$  ביטים. מהו הטווח האפשרי של  $N$  כפונקציה של  $n$ ?
- היכי קטן:  $0 \dots 10000$  (יצוג עשרוני:  $(2^{n-1})$ )
- היכי גדול:  $1 \dots 11111$  (יצוג עשרוני:  $(2^n - 1)$ , למה?)
- מסקנה:  $2^{n-1} \leq N \leq 2^n - 1$

**מסקנה:** מספר הספרות  $d$  שדרוש כדי לייצג את המספר  $N$  בסיס  $b$  הוא  $1 + \lceil \log_b N \rceil$ .

- למשל בשביל לייצג  $3147251$ , צריך 7 ספרות. כי  $7 = \lceil \log_3 10000000 \rceil < \lceil \log_{10} 3147251 \rceil < \lceil \log_3 10000000 \rceil$ .

## כמה ביטים יש ביצוג הבינארי של המספר N?

- מהו מספר הביטים  $a$  ביצוג הבינארי של  $N$ ?

ערך יחיד ולא טווח כמו קודם. למה?

נשתמש בסעיף הקודם:

$$a^x = b \Leftrightarrow \log_a b = x$$

$n - 1 \leq \log N \leq 2^{n-1}$  וכן

$\log N < 2^n$  וכן  $n < \lceil \log N \rceil$

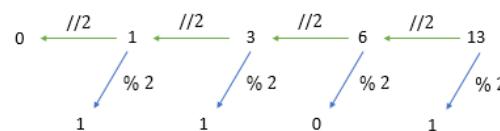
כלומר:  $\lceil \log N \rceil \leq n \leq \lceil \log N \rceil + 1$

- יש בדיקה ערך אחד שלם בטוח (למה?):  $1 + \lfloor N \log_2 10 \rfloor$

קייםיחס לינארי בין היצוג של מספרים בסיסים שונים. למשל, המספר שמיוצג עם  $d$  ספרות בסיס 10, יקח  $\lceil d \log_2 10 \rceil$  ספרות ביבנארי.

### המרה מדצימלי (בסיס 10) לבינארי (בסיס 2)

נמיר את המספר 13 הדצימלי לבינארי:



- The sign bit is 0 → sign = +1
  - exponent = 10000000001<sub>2</sub> = 1025<sub>10</sub>
  - The fraction bits are 01100...0 →

$$\begin{aligned} \text{fraction} &= \sum_{i=1}^{52} b_i \cdot \frac{1}{2^i} = 0 \cdot \frac{1}{2} + 1 \cdot \frac{1}{4} + 1 \cdot \frac{1}{8} + 0 \cdot \frac{1}{16} + \dots + 0 \cdot \frac{1}{2^{52}} \\ &= \frac{1}{4} + \frac{1}{8} = 0.375 \end{aligned}$$

$$\text{sign} \cdot 2^{\text{exponent}-1023} \cdot (1+\text{fraction}) =$$

$$(+1) \cdot 2^{1025-1023} \cdot (1 + 0.375) = 2^2 \cdot 1.375 = 5.5$$

### חסרונות:

- למשל הבעה הידועה עם 0.1.
  - או אפילו יותר.
  - או אפילו יותר.
  - או אפילו יותר.

אי הבדיקה של נקודה צפה (תרגול 4):

- נשים לב שבחלק של fraction יש 52 ביטים, ולכן יש  $2^{52}$  **מספרים שונים** (מ-0 ועד  $1 - 2^{-52}$ ). מידת הדיק בתחום מסויים, נניח  $(2^k, 2^{k+1}]$  תהיה אורך האינטראול חלקי במספרים השונים בו:  $2^{k-52} = \frac{2^{k+1}-2^k}{2^{52}}$ .
  - למספרים קטנים יש רמת דיק גבוהה וטעויות החישוב יהיו קטנות יחסית, נניח עבור התחום  $(8, 16]$  מידת הדיק היא  $2^{-49}$ .
  - מהכיוון השני, למספרים גדולים רמת הדיק היא נמוכה ויורר טעויות חישוב, נניח עבור התחום  $(2^{132}, 2^{133}]$  מידת הדיק היא  $2^{-80}$ . **בכל** שהמספר גבוה יותר רמת הדיק יורדת.

## יצוג תווים (Unicode, ASCII)

דברנו על הייצוג של תווים במחשב בשיטת ASCII ובשיטת Unicode. Unicode הייתה שיטה מאוד מוגבלת, רק עם 128 תווים ולכן הרחיבו ועboro ל-Unicode.

ASCII Code Chart																
0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI	
DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US	
!	"	#	\$	%	&	'	(	)	*	+	,	-	.	/		
0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?	
@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	
P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	-	
~	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	
p	q	r	s	t	u	v	w	x	y	z	{		}	~	DEL	

## 2 - סיבוכיות

### C – אלגוריתמים בסיסיים וסיבוכיות

#### חיפוש

פעולה מאוד בסיסית שאנו נדרשים לעשות. אנחנו נתמקד בחיפוש ברשימה.

##### נדיר את הבעיה:

- קלט – רשימה ממינית (מסודרת מהקטן לגדול), ומפתח לחיפוש.
- פלט – האינדקס של המקום ברשימה שבו נמצא האיבר שאנו מחפשים, ואם הוא לא נמצא להחזיר ערך כלשהו.

##### פתרונות אפשריים:

(1) **חיפוש לינארי (Sequential search):** עוברים על הרשימה לפי הסדר מההתחלת עד הסוף.

- במקרה הypi טוב – נבצע איטרציה 1.
- במקרה הypi גרע – נבצע ח איטרציות.

האלגוריתם רץ בסדר גודל לינארי בגודל הקלט. זמן הריצה הוא פונקציה של אורך הקלט, עד זמן הריצה ישתנה אם הקלט יהיה יותר גדול, מה המגמה של השינוי בזמן הריצה. אם ננסה אלטרנטיבה אחרת - נהפוך את הרשימה וכך בכך נverb מוסף להתחלה. גם כאן אין שינוי במקרה הypi גרע.

מה הבעיה כאן? **לא ניצלנו את העובדה שמדובר ברשימה ממינית.**

**הנחתת היסוד:** כל איטרציה חסומה על ידי קבוע כלשהו. הקבוע הזה הווה תליי במחשב וכן קשה להשתמש בו בתור מdad. המספר שאינו תלוי במחשב הוא **מספר האיטרציות**. לכן כרך נחשב את הייעילות, וזה הדבר **שמשתנה עם אורך הרשימה**.

(2) **חיפוש ביניארי (Binary search):** ניעזר בשלושה מצבים למקומות שונים ברשימה, left-mid-right. בכל איטרציה אנו בודקים את הערך של  $\text{lst}[mid]$  מול הערך הנוכחי ואנו נקבעו אם הוא נמצא או לא (right > mid). נמשיך בתהליך זה עד שנמצא את הערך.

- במקרה הypi טוב – נבצע איטרציה 1.
- במקרה הypi גרע – נחלק כל פעם ב-2 את אורך הרשימה ונconiח שלא נמצא (right < mid). נבצע בקיורוב  $1 + \lceil n / 2 \log n \rceil$  איטרציות.

##### מסקנות:

- אם נשווה בין זמן הריצה הלינארי לאלגוריתמי, נשים לב שהපער הוא ממשועוט. אם נכפיל ב-2 את אורך הקלט שלנו. נקבל שהלינארי **גדל פי 2**, והאלגוריתמי בסך הכל **גדל בתוספת של קבוע**:  $2 \log(n) = \log(2n)$ .
- בחיפוש הביניארי – כאשר הקלט גדול פי 2, ניתן לחשב על כרך בלבד יש לבצע חלוקה נוספת ב-2, ועוד חזרנו לקלט המקורי. זו **בדיקה נוספת של איטרציה אחת**.

#### מיזן

##### נדיר את הבעיה:

- קלט – רשימה.
- פלט – רשימה עם אותם איברים, ממינית לפי "גודל".

**מיזן בחירה (Selection sort):** הרעיון הוא לעבור על כל הרשימה ולמחפש את האיבר הypi קטן. ברגע שמצאנו אותו, נחליף את האיבר הypi קטן עם האיבר שהוא במקום הראשון ברשימה.

- כרך קיבלנו **במקום הראשון ברשימה את האיבר הypi קטן** (אייפה שהוא אמור להיות).
- בעת נתן我们必须經過某個元素，才能知道它是否是我們要找的最小值。

ננתה את היעילות:

- הרשימה באורך  $n$ .
- בחישוב של המינימום, רצה LOLAH פנימית ומבצעת  $i = 1 - n$  איטרציות לפחות?

$$(n-1) + (n-2) + \dots + 0 = \frac{n(n-1)}{2}$$

## מיזוג

נדיר את הבעה:

- קלט – שתי רשימות ממוכנות.
- פלט – מיזוג ממויין של שתי הרשימות.

הפתרון הנבחר יהיה עם 3 אידקסים רצים, כך שבכל איטרציה בוחרים את האיבר הקטן ביותר משתי הרשימות ומכניסים אותו לרשימה הממזגת. הפתרון זהה מתבסס על כך שהרשימות ממוכנות.

## סיבוכיות ו-notation O

fonction f(n) היא O(g(n)) אם קיים קבוע c ביחסו כך שהכל מ- $n$  מסויים:  $|f(n)| \leq c|g(n)|$ .

- ( $f$ ) היא הפונקציה שיחסבנו במספר הפעולות שמבצע האלגוריתם על קלט באורך  $n$ . **זהו ביטוי מדויק של מספר הפעולות.**
- את זה נרצה לחסום על ידי הפונקציה  $(n)g$  בדומה dazu שהfonction  $(n)g$  תהיה בהמה שיותר פשוטה.
- יכול להיות שעבור  $n$ -ים קטנים זה עדין לא מתקיים, אנחנו מתעניינים החל ממקום מסוים.
- ( $n$ )  $g$  היא סדר הגדול.

דוגמאות (סוף שיעור 8):

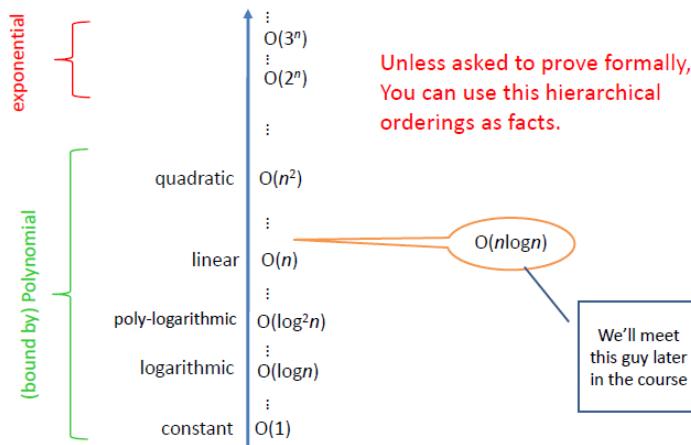
- $3n + 7 = O(n^2)$ . מתקיים:  $3n + 7 \leq n^2 + 5n^2 + 7n^2 = 13n^2 = n^2 + 5n + 7$  כיוון עבור  $13 = c$  זה מתקיים. אפשר גם להסתכל על  $n^2 + 5n + 7 \leq n^2 + 5n^2 + n^2 = 7n^2$  עבור  $3 \geq n$ . במקרה זה קיבלנו  $7 = c$ . **אנו מחפשים את החסם ההדק ביותר.**

Claim	Explanation
$3n + 7 = O(n)$	$3n + 7 \leq 3n + 7n = 10n$ $c = 10, n \geq 1$
$3n + 7 = O(n^2)$	$3n + 7 \leq 3n^2 + 7n^2 = 10n^2$ $c = 10, n \geq 1$
$3n + 7 \neq O(\sqrt{n})$	נניח בשילוליה שקיים $c > 0$ כך ש- $3n + 7 \leq c\sqrt{n}$ עבור כל $n > n_0$ . $3n \leq 3n + 7 \leq c\sqrt{n}$ $\frac{n}{\sqrt{n}} \leq \frac{c}{3}$ $\sqrt{n} \leq \frac{c}{3}$ שורש היא פונקציה מונוטונית עולה שאינה חסומה מלמעלה, ולכן קיבלנו סתירה. $c = \frac{3}{\sqrt{n}}$
$5n \cdot \log_2 n + 1 = O(n \log n)$ לא משנה איזה בסיס נבחר ל- $\log$ בצד ימין, ניתן לשנות את $c$ בהתאם, ועודין להגיע לאותה תוצאה.	$5n \log_2 n + 1 \leq 5n \log_2 n + \log_2 n \leq n$ $5n \log_2 n + n \log_2 n = 6n \log_2 n = \frac{6n \log_{10} n}{\log_{10} 2} =$ $c = \frac{6}{\log_{10} 2} \log_{10} n$
$3^n \neq O(2^n)$	נניח בשילוליה שקיים $c > 0$ כך ש- $3^n \leq c2^n$ עבור כל $n > n_0$ . נקבל: $\left(\frac{3}{2}\right)^n \leq c$ כיון שהfonction הזו מונוטונית ולא חסומה, קיבלנו סתירה.

הערות:

- $O(1)$  – האלגוריתם רץ בזמן קבוע **לא תלות בגודל הקלט**.
- הערה: כאשר שואלים מהי סיבוכיות הח�ן ב-BCT של sort, לא לענות: "באשר ממיניכם רשותה באורך אחד". סיבוכיות תמיד מודדים כפונקציה של אורך הקלט, שכן אין משמעות להגד הסיבוכיות קטנה כאשר הקלט קטן. זה בהינתן קלט אורך בכל שייה, מה עדין יכולים להיות המאפיינים שלו שיגרמו לה לרוץ יותר מהר או פחות מהר.

## Complexity Hierarchy


דוגמאות:

- All these results refer to **worst case** scenarios.
- Algorithms on sequences:
  - **Binary search** on a sorted list of length  $n$  takes  $O(\log n)$  iterations
  - **Selection Sort** on a list of length  $n$  takes  $O(n^2)$  iterations
  - **Merging** of 2 sorted lists of sizes  $n$  and  $m$  takes  $O(n + m)$  iterations
  - **Palindrome** checking on a string of length  $n$  takes  $O(n)$  iterations
- Algorithms on integers:
  - **Addition** of two  $n$ -bit integers takes  $O(n)$  iterations
  - **Multiplication** of two  $n$ -bit integers takes  $O(n^2)$  iterations

גודל הקלט – הבהרות:

- אנו מודדים את זמן הריצה כפונקציה של גודל הקלט.
- עברו מספרים – גודל הקלט הוא **מספר הביטים** ביצוג של המספר במחשב.
- עברו אוספים כמו רשותה, מחירות, מיליון – גודל הקלט הוא רוב **מספר האיברים** באוסף.

חסם הדוק -  $\Theta$ :

- אם נכתב  $(n) f(n) = \Theta(g(n))$  אז נאמר כי  $\Theta$  הוא חסם הדוק על  $f$ .
- עברו הפונקציה  $.logn + 1 = O(logn)$  אבל זה לא יהיה חסם הדוק. ( $O(logn + 1) = \Theta(n)$  ובמקרה זה מדובר בחסם הדוק ולכן יוכל לסמן  $.logn + 1 = \Theta(n)$ )
- צריך להתקיים  $f(n) = O(g(n)), g(n) = O(f(n))$

### 3 - רקורסיה

#### D - רקורסיה

##### מבוא

הגדרה – פונקציה ורקורסיבית היא פונקציה **שמכילה קראיה עצמה**.

ע策ת: דוגמה פשוטה לrekурсיה היא חישוב של עצרת. ההגדרה היא ורקורסיבית:  $!n = !n-1 \cdot n$  עבור  $n \geq 1$ .

```
def factorial(n):
    if n==0:
        return 1
    else:
        return n*factorial(n-1)
```

נשים לב שמקורה הבסיסי/תנאי העצירה שלנו בחישוב הוא  $n=0$ , שם נחזיר 1. הערך זהה יופיע בחזרה לכל הקריאה הקדומות ובכך יוחשב הערך בצורה ורקורסיבית.

מספרים פיבונאצ'י: גם ההגדרה של מספרי פיבונאצ'י נעשו באמצעות ורקורסיה (נוסחת נסיגה):  $F_n = F_{n-1} + F_{n-2}$  ונקבל:

```
def fibonacci(n):
    if n<=1:
        return 1
    else:
        return fibonacci(n-1) + fibonacci(n-2)
```

כדי לוודא שהפונקציה הרקורסיבית תהיה תקינה:

1. מdag שקיים מקרה בסיס – תנאי עצירה.
2. מdag שעל כל קלט שיגיע לפונקציה, הוא יLR ויתכנס לתנאי העצירה. הקריאה הרכסיםיות ילבו ויקדמו אותו לבירור תנאי העצירה.

עצי רקורסיה:

- חישוב סיבוכיות – ליד כל צומת בעץ, נרצה לכתוב כמה זמן השיקענו. סיבוכיות הזמן תהיה סך כל העבודה שעשינו בכל הצמתים של העץ.
- זיהוי חוסר עילוות – ניתן להזוט חוסר עילוות, למשל אם אנחנו מחשבים ערך מסוים שוב פעמיים ולא מנצלים את המידע פעם נוספת – למשל – `fibonacci(2)`.

פונקציית `enumerate` – **עוטפת את הקריאה הראשונה לפונקציה הרקורסיבית**, מסתירה מידע לא נחוץ ממי שקורא לפונקציה (בנcluding שהקריאה הראשונה היא עם פרמטרים של 0 ו-1-ה).

##### חיפוש ומילוי

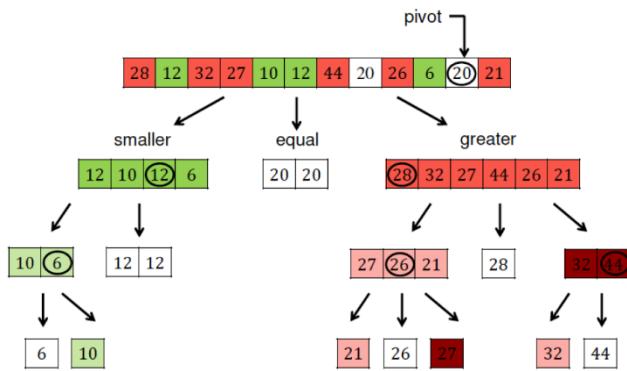
חיפוש בינארי (Binary search):

נרצה לכתוב מימוש ורקורסיבי לחיפוש בינארי:

- בימוש הקודם הלולאה עצרה כאשר  $right > right < left$ .
- תנאי העצירה יהיה  $right > left$ . כל עוד זה לא קורה, נחשב את  $mid$ , ונקרה לפונקציה שוב עם התווים המתאים, חיפוש בחצי הימני או השמאלי.
- אם במקומות להעבר `left`, `right`, הינו מעבירים את תת-הרשימה באוזור שבו נשאר לחפש – באמצעות `slicing`, כבר כמוות העבודה שהיינו עושים בקריאה הראשונה, הייתה ( $n$ ) 0 ולא הינו עובדים בסיבוכיות ( $\log n$ ).

### מיזן מהיר (Quicksort):

#### האלגוריתם:



1. בוחרים איבר אחד שיהיה **pivot** (באופן רנדומלי), ויצרים 3 רשימות חדשות – כל האיברים שקטנים ממנו, האיברים שגדולים ממנו, והאיברים שווים לו.

2. נשים לב שרישימת השווים לו מכילה לפחות איבר 1, והרשימות של **smaller**, **greater** קטנות ממש מאורך הרשימה המקורי.

3. באופן וקורסיבי ממיניכם את הרשימות **smaller**, **greater** בפניה עצמן.

4. תנאי העצירה – אם אורך הרשימה הוא 1 (איבר אחד בפנוי עצמו) הוא נחשב ממון) או 0 (ריקה).

נשים לב כי ברכורסיה קודם כל מחושב הבן שמאלית של השורש, ורק אז הבן הימני של השורש.

#### הערות:

- בקורס מבני נתונים – נחזור ללמידה quicksort ונראה גרסה שימושת ממיינת את הרשימה במקומ, לא יוצרת רשימה חדשה (אצלנו זה קורה בגל האופרטור +), אלא מבצעת swap של איברים.

- בימוש הפיתונו אנו משתמשים ב-list comprehension – ביצענו 3 פעמים מעבר על הרשימה, כלומר 3 במקומות לעבור על הרשימה פעם אחת שהיא לוקחת לנו זאת. **3 מול 1 – המחיר הוא זניח**.

התוכנסות – איך אנו יודעים שזה מתכנס? כיוון שתמיד יש איבר אחד ברשימה `equ`, תמיד הרשימות `smaller`, `greater` קטנות יותר ולבן כל פעם מתרבצת קרייה על קלט קטן יותר.

#### נכונות – לרוב נוכח באינדוקציה:

- בסיס – אם הרשימה ריקה או מכילה איבר אחד, לא מבצעים רקורסיה ופושוט מוחזרים את הרשימה עצמה (היא הרי ממוינת).
- צעד – אם `smaller`, `greater` ממוניות, אז אם נוסיף את `equ` ביןיהם נקבל רשימה ממונית.

#### פיתוח סיבוכיות:

בצע ניתוח סיבוכיות **בערך** רקורסיה. נניח לשם הפשטות כי כל האיברים שונים זה מזה.

עומק הרקורסיה	סיבוכיות הזמן	בערך רקורסיה	מקרה
$O(n)$	$O(n^2)$	$WCT(n) = cn + WCT(n - 1)$	<b>WCT</b> <b>worst</b> <b>(case)</b> בחרנו את pivot להיות המינימום או המקסימום
$O(\log n)$ כמה פעמים צריך לחולק את ח-2 עד שנגיע ל-1?	$O(n \log n)$ בכל רמה בערך הסכום הוא $c \cdot n$ . יש $\log n$ רמות.	$BCT(n) = cn + 2BCT\left(\frac{n}{2}\right)$	<b>BCT</b> <b>(best case)</b> בחרנו את pivot להיות החזין

נשים ❤️

- בחירת אינדקס אקראי בראשימה – לא תהיה יותר  $M(n)$ .
- סיבוכיות הזמן – סך כל העבודה בכל צמת העץ. בכל רמה בערך נראת מה הסכום.
- עומק רקורסיה (כותבים במונחי 0).

**Random quicksort**

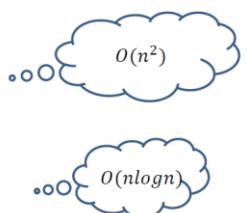
Ordered list  
 $n=200 \quad 0.10658217853053813$   
 $n=400 \quad 0.23033834734888864$   
 $n=800 \quad 0.48683051048205617$

Random list  
 $n=200 \quad 0.11578862173792887$   
 $n=400 \quad 0.24735086264755812$   
 $n=800 \quad 0.49966520589048125$


**Deterministic quicksort**

Ordered list  
 $n=200 \quad 0.5891511705949701$   
 $n=400 \quad 2.1920545619645466$   
 $n=800 \quad 8.422338821114138$

Random list  
 $n=200 \quad 0.08930474949662441$   
 $n=400 \quad 0.1763828023214497$   
 $n=800 \quad 0.36725255635832443$


**:deterministic quicksort**

נתבונן באלגוריתם deterministic quicksort – שבו אנו לוקחים את pivot כאיבר הראשון בראשימה. עברו קלט ממון – נקבע WCT אם נבחר/almento הראשון (המינימום) או/almento האחרון (המקסימום). לעומת זאת, אם נבחר באופן רנדומלי, עברו קלט ממון יש סיכוי נמוך יותר שנקבע WCT.

ב-deterministic quicksort עברו רשימה ממוקנת (שהיא קלט גרען) זמן הריצה הוא  $O(n^2)$ . לעומת זאת התנהגות היא גם  $O(n \log n)$ . למה הזמן של רשימה אקראית כאן קצר יותר טובים מה-random quicksort? כי אנחנו ניגשים ישר לאיבר מסויים [pivot = lst[0]].

**:Merge sort**

זה אלגוריתם רקורסיבי, דטרמיניסטי (לא שום אקראיות).

1. בכל איטרציה האלגוריתם מפצל את הרשימה ל-2 (באמצע).
2. מבצעים merge על 2 החצאים לאחר מיזן (באמצעות mergesort).

**סיבוכיות:**

בכל קדאה משתמשים ב-slicing (עליה בגודל  $\frac{n}{2}$ ,  $slice(n)$ ) וב-merge ( $O(n)$ ). בולמר בערך צומת בערך אחד משקיים  $c.n$ . אין הבדל בין WCT ל-BCT, התנהגות היא תמיד אותה התנהגות – עץ הרקורסיה כולל בנייתו והוא יראה אותו הדבר. גם כאן הסיבוכיות היא  $O(n \log n)$ .

**מגדלי האנו**

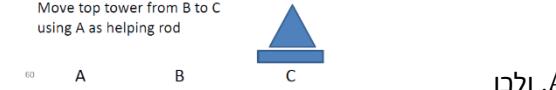
הבעיה הידועה והנוכרת שתפסה אותנו לא מוכנים במועד א' של בדידה – תעוגן.

הפלט שאנו רוצים ליצור בפונקציה ש牢记 את פתרון הבעיה, היא סט של פקודות: "להעביר את דסקית X ממוט A למוט B".

רקורסיבית נרצה להעביר מגדל של  $n$  דיסקיות ממוט A למוט B באמצעות מוט העזר C (רקורסיבית היה כאן סט של פקודות). ניקח את הדיסקית התחתונה מ-A ל-C, וניקח רקורסיבית את כל הדיסקיות ממוט B למוט C באמצעות מוט העזר A.

נכונות - בשום שלב באלגוריתם החוקים לא מופרים:

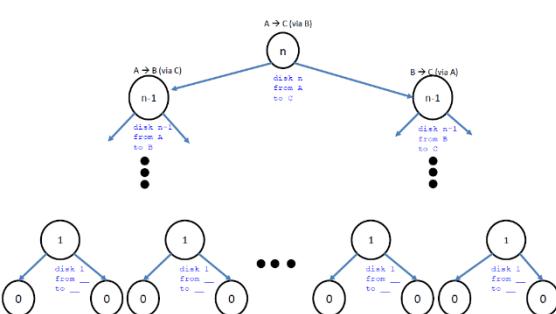
1. כל  $n$  –  $n$  הדיסקיות שלקחנו ממוט A, יותר קטנות מהדיסקית שנשארה במוט A, ולכן בתהליך הרקורסיבי נוכל להניח עליה כל דיסקית אחרת ולא נפר את התנאי של הנחת דיסקית קטנה מעתה.
2. אין אף דיסקית קטנה שנמצאת מתחתיה, כל  $n-1$  –  $n$  הקטנות יושבות על מוט אחר.
3. אותה טענה נכונות של השלב הראשון.

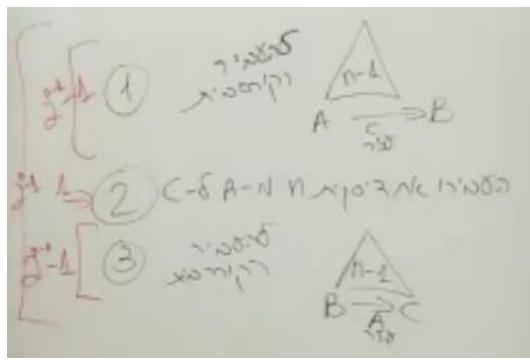

**סיבוכיות:**

- תנאי העזירה –  $0 = n$  ואז לא נעשה שום דבר.
- עומק הרקורסיה –  $(n)$  בכל שכבה אנו מקטינים ב-1, יש  $1 + n$  שכבות.
- סיבוכיות הזמן – בכל צומת העבודה היא  $c$ . כמה צמתים יש בעץ? יש  $2^n$ .

בולם הסיבוכיות היא  $= (2^{n+1} - 1)c = c(2^{n+1}) = c(2^n + 2^{n-1} + 2^{n-2} + \dots + 2^0)c = c(2^n)0$ .

נדרשים  $1 - 2^{n-1}$  צעדים (הdfsות) כדי לפתור את הבעיה בצורה האופטימלית ביותר.





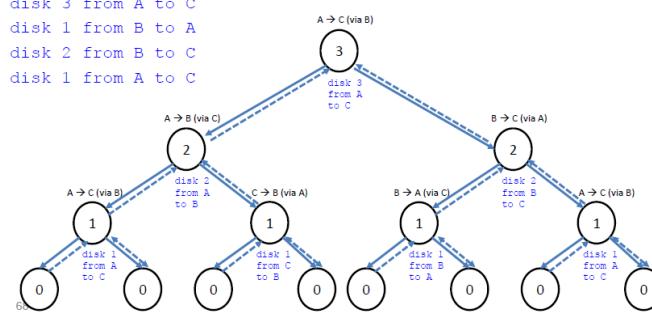
**מפלצת האנו:** עבור מגדל עם 200 דיסקיות, קשה לבנות את עץ הרקורסיה באוטומאציה. נרצה לדעת רק מהלך א' בפתרון. כיצד שאנחנו יודעים את מבנה הדרישות -  $1 - 2^{n-1}$  צעדים, נוכל להחליט האם צריך לחפש את הפוקודה הרלוונטיות באוצר של הקוריות הרקורסיביות 1 או 3, או בפקודה 2 עצמה:

- אם  $1 - 2^{n-1} \leq k \leq 1$  נחפש בפקודה 1.
- אם  $k = 2^{n-1}$  נחפש בפקודה 2.
- אם  $1 \leq k \leq 2^n - 2^{n-1} + 1$  נחפש בפקודה 3.

לדוגמה, אם נרצה לחפש את הפוקודה החמישית, אנו נחפש בתת-העץ הימני אבל נדרש לחסר את כמהות הפוקודות שיש מצד השני. לכן בקריאה רקורסיבית זו אנחנו נعبر  $.k - 2^{n-1}$ .

```
>>> HanoiTowers("A", "B", "C", 3)
```

```
disk 1 from A to C
disk 2 from A to B
disk 1 from C to B
disk 3 from A to C
disk 1 from B to A
disk 2 from B to C
disk 1 from A to C
```



כלומר אנו מבצעים סוג של "חיפוש ביןארי" על המיקום הנכון בעץ הרקורסיה שבו פוקודה k אמורה להיות. סיבוכיות זמן הריצה היא (n)0.

## ממויאיזציה

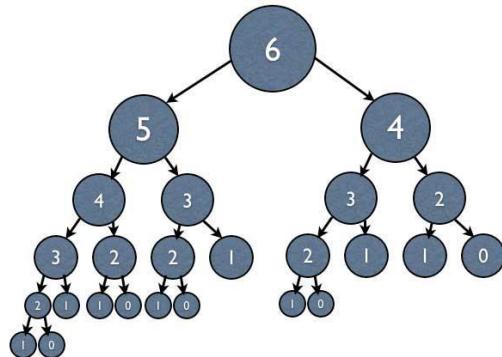
### סיבוכיות זיכרון:

- עד עבשו דיברנו על סיבוכיות זמן. באופן כללי סיבוכיות זיכרון, זו הדרך שלנו לתת מدد **לכמהות הזיכרון שהאלגוריתם משתמש בו ומקצה בהמשך הריצה שלו** על קלט מסוים בגודל ג'. נתעלם מהזיכרון שהקלט דורש – זה בסדר. מעניין אותנו מה האלגוריתם עושה מעבר לייצוג של הקלט בהמשך הריצה שלו.
- הבדל מהותי בין זיכרון לזמן** – בזיכרון צריך לבצע reuse, בזמן T מסויים צריך הרבה זיכרון אבל הזיכרון הזה יכול להיות זיכרון שהשתמשנו בו בזמן קודם. נניח שיש לנו לולה שבכל איטרציה מיקצה לעצמה משתנה ובכל איטרציה קצת מחדש – במקרה של כמהות הזיכרון, **לא נסכם את כל הזיכרון לאורך כל האיטרציה**. מבחןינו, לפני כל איטרציה עבר-h-GC וחרר את כל הזיכרון. נניחשה-h-GC שעבד כל הזמן, אין זבל, וכל זיכרון שסתם תפוס משוחרר כל הזמן. אנו מתעניינים **בכמהות הזיכרון המקסימלית לאורך כל זמן ריצת האלגוריתם עבור זמן T מסוים**. לעומת זאת בזמן – אנו סובבים את כמהות הזמן, את "השתח" שמתחetta לפונקציית הזמן.
- רקורסיה** – לרוב לא נבקש לנתח סיבוכיות זמן של פונקציות רקורסיביות. יש קשר בין עומק הרקורסיה **לזיכרון – עמוק** – הרקורסיה זה המספר המקסימלי של סביבות בזיכרון, קוריות סימולטניות שיכולה להיות פתוחות בזיכרון ולהיות לתשובה שהקריאה הת תמונה |תוחזר. בהאנו למשל יהיו **צ'ו** סיבובים בחזרות שמחכות. וכל סיבבה בזאת תופסת מקום קבוע בזיכרון. **עומק הרקורסיה נותן חסם מוחלט לכמהות הזיכרון שהפונקציה הרקורסיבית צריכה**.

### חרונות עיקריים של פונקציות רקורסיביות:

1. קוריאות לפונקציה הן דבר יקר! (מבחן זמן)
2. קוד רקורסיבי עלול לבצע חישובים חוזרים – היא עלולה לקרוא לעצמה על אותו קלט מספר פעמים (למשל בפיבונacci). את זה ננסה לשפר באמצעות **mmoiaizcha**.

כיתוח עץ רקורסיה של פיבונאצ'י:



- בכל צומת בעץ ב�ות העבודה היא קבועה. **עומק הרקורסיה היא (n).**
  - במסלול הימני יהיו  $\frac{n}{2}$  צמתים, במסלול השמאלי יהיו  $n$  צמתים. אפשר להשלים את העץ להיות כמו בהאנוי, لكن נוכל לומר שיש כאן פחות מ- $n^2$  צמתים (ב�ות הצמתים בעץ ה- $n$ 'ל קטנה מכמות הצמתים בעץ של האני).

לכן סיבוכיות הזמן היא  $O(n^2)$  – חסם מלמעלה לא הדוק!

• אם נסתכל על העץ עד  $\frac{n}{2}$ , מספר הצלטמים הוא  $2^{\frac{n}{2}} - 1$  ו-  $2^{\frac{n}{2}} - 1 = O(\sqrt{2}^n)$

ב-ח כלומר זה גרוע!

ממצאים:

נרצה לחסוך את הקריאהות החזירות. באמצעות זיכרון עדר, זהו מבנה אבסטרקטי שנקרא מילון (לא בהכרח מילון של פיותן), אפשר למשמש אותו בכל מיני דרכים. שיטה זו נקראת ממואיציה. זה מקרה פרטי של תכנון דינמי (dynamic programming).

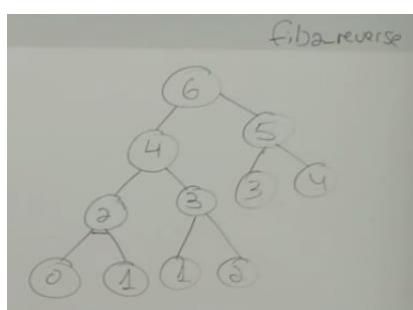
נשתמש בזיכרונו עזר כדי למפות קלטיים לתוכנת הפונקציה **עליהם**. אם בזיכרון העזר יש כבר את התוצאה, אין צורך לבצע את הקראיה הרקורסיבית וננסה לחסוך אותה. נשתמש במבנה שנקרא **dictionary**. איך נממש אותו?

1. על ידי dict - נשמר זוגות של מפתח וערך, המפתח יהיה הקלט, והערך יהיה הפלט של הפונקציה על הקלט. בחיפוש/הכנסה עולם (1) O במשמעותו.
  2. על ידי רשימה – הפלט המתאים לקלט ? ישמר בתא ה-? ברשימה. במקרה זה הקלטים צריכים להיות מספרים (שנוכל ליצג אותם באמצעות האינדקסים). גישה לתא ה-? (שליפה/הכנסה) לוחחת **(1) O**.

הערות:

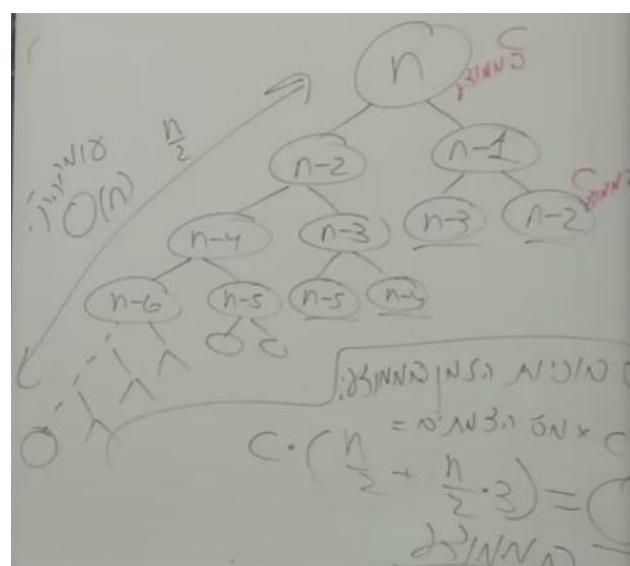
- הפונקציה הרקורסיבית המקורית תקבל גם את מילון העזר, הוא תמיד עבר בתור פרמטר נוסף (מצביע למקומות בזיכרון שבו ישוב המילון) וזהו אותו המילון לכל אורך הדרך. **מילון הוא mutable** וכל שינוי ישתקף אחר כך בכל גישה אליו.
  - לא תמיד נכון להשתמש בממואיציה, ורק אם יש חישובים חוזרים. בוגדי אני נגד אין חישוב שוחזר על עצמו, הסדר של הפרמטרים והתוצאות הן אחרות.

**ע"ז רקורסיה של פיבונאצ'י (עם ממואיזציה):** עומק הרקורסיה –  $n$ . סיבוכיות הזמן – ב奏מת סך העבודה לוקחת  $(1) \cdot 0$  במשמעות, אז זה  $c$  במשמעות. סה"ב ויצא  $(n - 1) + n = 0$  (א צמתים על הענף השמאלי, ועוד 1 – א קטנים שיוצאים ימינה). זה שיפור ממשמעותי.



מה יקרה אם נהפוך את סדר הקריאה? קודם נחשב את פיבונאצ'י על  $n-2$  ואז על  $1-n$ .  
עבור  $n=6$

## וְעַבְדָּה חַבְלִי:



אפשר גם למשתמש את פיבונאצ'י באופן איטרטיבי ולא רקורסיבי באמצעות רשימה. פתרון זה יוצא יותרiesel מהפתרון הרקורסיבי. גם כאן ניתן לעשות שיפור ולשמור שני משתנים במקומם לשינה שלמה. מפורט במצבה שיעור 11.

סיכום:

- אם משתמשים ברקורסיה יש חישובים חוזרים – כדאי להשתמש בממואיזציה. אבל לא לכל פונקציה רקורסיבית ממואיזציה תועיל, מדובר בגישה ליזיכון זהה יכול להאט. ב-quicksort ה בעיה היא שונה ושם ממואיזציה לא תעוזר לנו.
- אם אפשר להשתמש בלולאות ולא ברקורסיה, כדאי לבחור בלולאות – אם מצליחים לשמור על אותה סיבוכיות זمان.

רקורסיות שראינו עד כה:

דוגמא	פערות מעבר לרקורסיה	קריאות רקורסיביות	נוסחת נסיגה	סיבוכיות
max (מהתרגול), עצרת	1	N-1	$T(N)=1+T(N-1)$	$O(N)$
חיפוש בינארי	1	N/2	$T(N)=1+T(N/2)$	$O(\log N)$
Quicksort (worst case)	N	N-1	$T(N)=N+ T(N-1)$	$O(N^2)$
Mergesort Quicksort (best case)	N	N/2 , N/2	$T(N)=N+2T(N/2)$	$O(N \log N)$
max (מהתרגול)	1	N/2 , N/2	$T(N)=1+2T(N/2)$	$O(N)$
הanoi	1	N-1, N-1	$T(N)=1+2T(N-1)$	$O(2^N)$
פיבונאצ'י	1	N-1, N-2	$T(N)=1+T(N-1)+T(N-2)$	$O(2^N)$ (לא הדוק)

## Munch

תורת המשחקים הוא תחום שמנסה ללמידה מודלים מתמטיים של קונפליקטיבים ושיטות פעולה בין שחקנים. המשחק נפל תחת קטגוריה של "full information" – כל השחקנים רואים את הלוח וידועים מה קרה בדיק לכל אורך המשחק עד אותו הרגע. מדובר משחק "zero-sum" – אם מדובר בשני שחקנים, כאשר שחקן אחד מרוויח זה על חשבו שחקן אחר – מפסיד.

במקרה של משחק צזה (סכום אפס, דטרמיניסטי) טעונה מתורת המשחקים אומرت כי בהכרח לאחד השחקנים, יש אסטרטגיית ניצחון. לא משנה מה יעשה השחקן השני, קיימת צדו אסטרטגיה כך שהשחקן הראשון ינצח.

נסתכל על המשחק Munch מתוך המתורת המתחרה האלגוריתמית (בעולם המשחק מופיע תחת השם Chomp), אשר עומד בכל הקירטוניים – ולן לאחד מהשחקנים, יש אסטרטגיית ניצחון. הבעיה – להגיד מהי אסטרטגיית הניצחון.

█			

[3, 3, 2, 1]

חוקים – בכל שלב המשחק, כל שחקן צריך לאכול לפחות קובייה אחת. הקובייה מגדרה מלבן שהוא הפינה השמאלית התוחנה שלו – ואוכלום את כל מה שנמצא במלבן (מיינין, מלמעלה, באלבסון). תמיד "נאכל" מכיוון ימין ולמעלה. הקובייה השמאלית התוחנה מורעלת – וכי שאוכל אותה מפסיק במשחק.

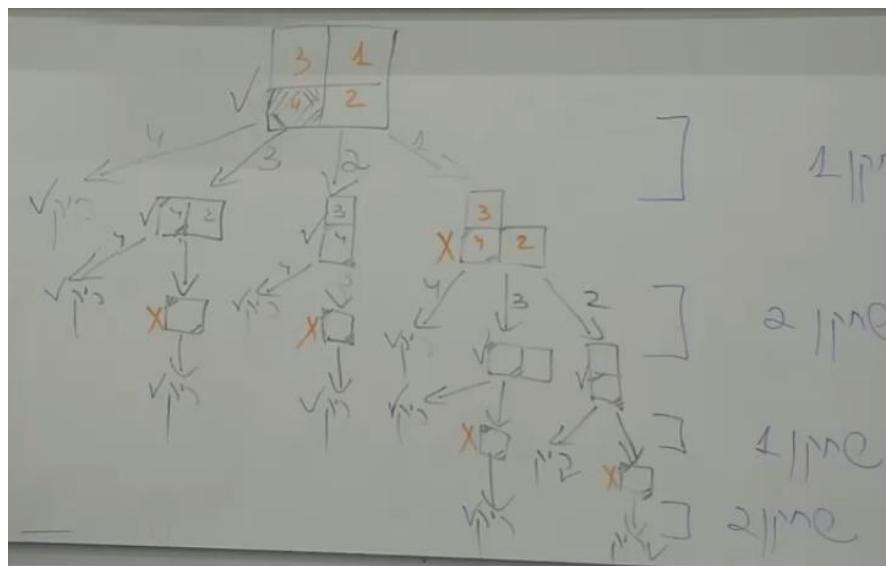
כדי ליאציג מצב של שוקולד ניעזר ברשימה, למשל [3,2,2,1]

בהתנן קונפיגורציה מסוימת C, נרצה להבין האם ניתן לנצח בה או להפסיד בה (בהנחה שני השחקנים משחקים באופן רצינאי):

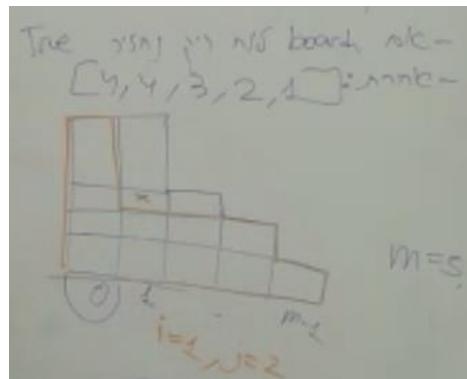
- C הוא **winnable** – אם קיימ איזה מהלך שהשחקן שעכשיו יוכל את הלוח הזה יכול לעשות, שיביא את הלוח הבא במצב מפסיד עבור היריב.
- C הוא **losing** – אם כל המהלךים שניתנו לעשות יובילו את הלוח למצב **winnable**. לא משנה איזה מהלך געשה, הלוח יהיה במצב מנצח עבור היריב.
- תנאי העצירה – לוח ריק הוא **winnable**.



ניתחנו את כל האפשרויות (ראינו כי ממוואיזציה יכולה לעזור את זמן הריצה). זה מזכיר את פיבונאצ'י ויש כאן אפילו יותר צמתים, لكن סיבוכיות הזמן היא אקספוננציאלית.



היצירה של `nimched_board` בקוד קצר קשה לעיכול (מודר לא? אמרנו שוקולד!):



נוכח שעבור לוח מלכני שמכיל לפחות 2 קוביות, לשחקן 2 אין אסטרטגיית ניצחון (בשילוב עם הטענה מתוך המשחקים המשקנה היא שלשחקן 1 קיימת אסטרטגיית ניצחון).  
נכיה בשילhouette שיש לשחקן 2 אסט' ניצחון.

- כמובן, כל מצב לוח שיתקבל לאחר צעד אחד של שחקן 1, יהיה `winnable` עבור שחקן 2 שקיבל אותו.
- בicut, נכיה ששחקן 1 בחר לאבול רק את הקוביה הימנית העליונה. לפי ההנחה שלנו הלוח הזה הוא `winnable`, לשחקן 2 יש תגובה טובה שתותר לו במצב `losing` עבור שחקן 1.
- "גניבת אסטרטגיה" – אבל, אם קיימת תגובה טובה שתביא לניצחון של שחקן 2, אז שחקן 1 יוכל היה לשחק אותה בצעד ראשון, ולהותיר את עbor 2 את הלוח במצב `losing`. בסתיו נראה לשחקן 2 יש אסט' ניצחון. כמובן, לשחקן 2 אין אסט' ניצחון ולכן לשחקן 1 יש.

הוכחנו קיום של אסטרטגיה אבל לא אמרנו מהי.

## פתרון בעיות רקורסיביות (תרגול 6):

שלבים בפתרון בעיה רקורסיבית:

1. תנאי עצירה: קלטים "פשוטים"uboּרומ ניתן לחשב תשובה ישירות - !
2. פירוק הבעיה לבעיות (אחד או יותר) קטנות יותר - !(1 – n)
3. חוזרת מפתרון הבעיות הקטנות לפתרון הבעיה המקורית - !(1 – n)n

עצי רקורסיה:

כדי לנתח זמן ריצה של פונקציה רקורסיבית, נשתמש בתבונה מאוד פשוטה – נדע מהן כל הקריאות הרקורסיביות שהfonקציה ביצעה,  
כמו זמן לוקחת כל קראיה, ואז נסכם את כל העבודה על פני כל הקריאות.

נתאר את ריצת הפונקציה שלנו כריצה של עצ. שורש העץ יהיה הקלט שהמשתמש סיפק, והבנים יהיו קראיות רקורסיביות שנפתחו על ידי אותו צומת.

- אפשר לשאול מהו המסלול הארוך ביותר משורש לצומת בלשו בעץ, ונקרו לאורך זה העומקה.
- נרשום לצד כל צומת את העבודה שמתבצעת בו, אם זה זמן קבוע נרשום (1). זמן הריצה יהיה גדול העץ במקרה זה, מספר הצמתים שיש בעץ.

בדוגמה של עצרת – כל קראיה מבצעת (1), ויש לנו א' קראיות רקורסיביות ולבן סה"ב זמן הריצה יהיה (n)0.

### 1 - בעיית המקסימום:

- קלט: רשימה L לא ריקה של מספרים
- פלט: איבר מקסימלי ב-L

תובנה: אם  $L_1 + L_2 \geq L$  אז  $\max(L) = \max(\max(L_1), \max(L_2))$

ב-idx\_max יש שורש, שני בניים, ארבעה נכדים, וכו' .. סדרה הנדסית  
באורך  $n$ :  $(n)0 = 2n - 1 = 2^{log n} = 1 + 2 + 4 + \dots + 2^{\log n}$

נרצה לפתור אותה רקורסיבית:

1. אם אורך הרשימה הוא 1, נחזיר אותו.
2. נחשב רקורסיבית מקסימום  $m_1$  ב- $L$  ואת המקסימום  $m_2$  ב- $L$ .
3. נחזיר את הגדל מביניהם.

בפתרון זה אנחנו מבצעים slicing, אז בכל איטרציה יהיה לנו זמן ריצה של (n)0, וזה יהיה לכל קראיה רקורסיבית שבוצע.  
לכן לא נחשב את ה-slice אלא משתמש בפרמטרים start, end בפורמטורים start, end על תח-רישמה.  
כדי להציג את טווח החיפוש "הפעיל" קיבל סיבוכיות של (n)0. **בכל צומת** בימוש עם האינדקסים נקבל סיבוכיות של (n)0. **בכל צומת** יש (1)0 עבודה וכן צריך רק להבין כמה צמתים יש בעץ.

### 2 - בעית Subset Sum:

- קלט: רשימת שלמים L ושלם s.
- פלט: האם יש תת-רישימה ב-L שסכוםה s.

תובנה: נסמן  $L[0] = x$ :

- או שיש לנו תת-רישימה בלי x שסכוםה s.
- או שיש לנו תת-רישימה עם x שסכוםה s.
- או שתתי האפשרויות לא מתקיימות ולבן אין תת-רישימה שסכוםה s.

פתרון וקורסיביות:

## 1. תנאי עצירה:

 a. אם  $0 = z$ : במקרה הריקה היא חילוק כל רשיימה וסכוםה הוא 0.

 b. אם  $[] = L \neq z$ : לא.

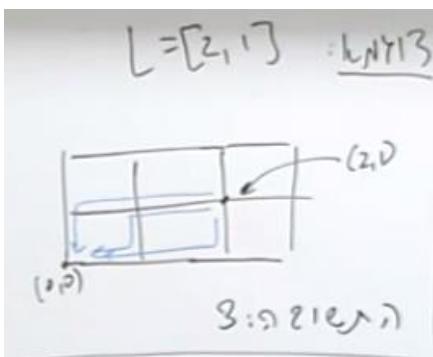
 2. נקרא לפונקציה על  $[1:L-s]$  או על  $[0:L-s]$ .

חישוב סיבוכיות:

נרצה להראות כי זמן הריצה חסום מלמטה על ידי  $2^n$ . לשם זה נראה כי בעץ יש בדיקת  $2^n$  עליים. אם נניח שזמן הריצה קבוע, גם אם נשאל רק כמה עליים יש בעץ, בברגע לזמן ריצה שהוא אקספוננציאלי  $2^n$ . כל עלה בעץ מייצג לנו תת-קובוצה אפשרית אחת, יש לכך  $2^n$  אפשרויות.

פתרון בעיות רקורסיביות – המשך (תרגול 7):

## 1 - בעית cnt\_paths



- קלט: רשימה  $L$  באורך  $d$  של שלמים אי-שליליים.

- פלט: מספר מסלולים מ- $L$  בראשית הצלרים שנעים רק על הצלרים וכיוון הראשית.

רקורסיה:

 1. תנאי עצירה: אם  $[0, \dots, 0] = L$  יש מסלול 1.

 2. לכל  $i$  בר- $0 < i < |L|$  (לא הגיענו עדין לנקודה אפס) נחשב רקורסיבית מסלולים אם צנו צעד אחד בקוורדינטה  $i$ .

3. נחזיר סכום.

במקרה הכללי הסקנו כי ב- $m$  הרמות הראשונות יש  $d$  קратיות רקורסיביות, ולכן זמן הריצה הוא לפחות  $2^{n \log d}$ . אם  $m = \text{cnt\_paths}(L)$  אז זמן הריצה הוא לפחות  $m$ .

למה זה נכון?

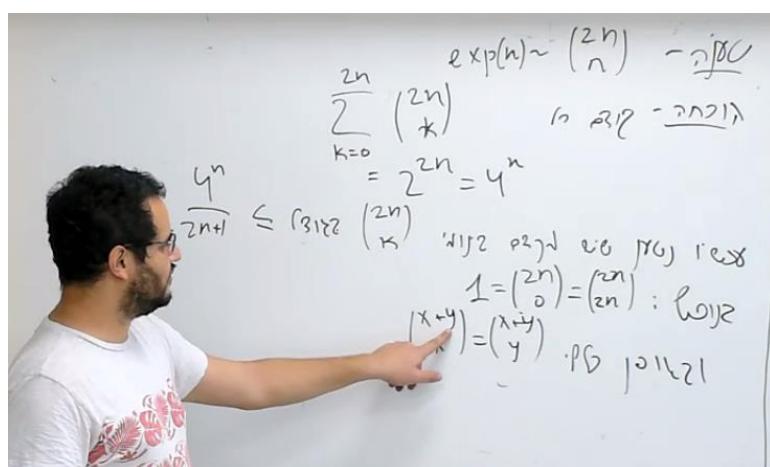
- כל פעם שmaguiim למקורה בסיס, מחליל 1 למעלה ואנו סובמים את כל ה-1ים האלה. רק שורה 3 בקוד (return 1) צריכה לkerot  $m$  פעמים, אז זמן הריצה הוא לפחות  $m$ .
- בעץ יש  $m$  עליים, וכל return כזה הוא עלה. מצאנו  $m$  צמתים בעץ ומשם גם נסיק כי זמן הריצה הוא לפחות  $m$ .**

## 2 - בינום:

זכורנו בהגדרת המקדם הבינומי  $\binom{n}{k}$ . לפי זהות פסקל ניתן להגיד רקורסיבית:  $\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$ . בהנחה  $s-k \geq n$  אנחנו תמיד נתבננס למקרי הבסיס  $\binom{n}{0} = 1$ .

עבור  $(4,2)$  נשים לב כי במות העלים בעץ (כל עלה אכן שב זה return) מיצגת את התוצאה הכללית – 6.

משהו שנען עשה שעמרי לא עשה ולא מופיע במצגת (25 דקוט לתרגול):



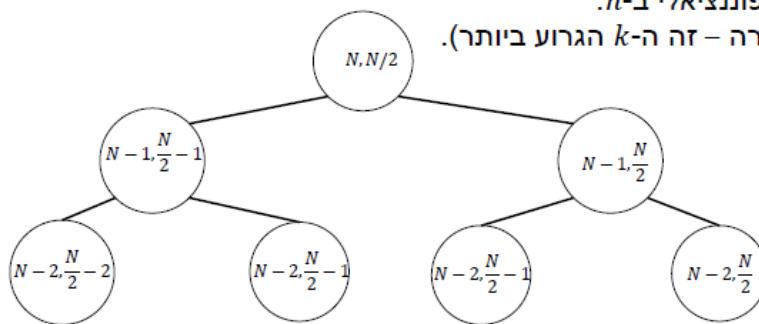
ניתוח סיבוכיות binom – ללא ממואיזציה:

בחרנו קלט ספציפי, ועבור המקרה הנוכחי זה רק יכול להיות יותר גראן. הקלט שבחרנו הוא  $n = N, k = \frac{N}{2}$  (ההפרש ביניהם, ובין  $k$  ל-0 הוא המקסימלי האפשרי, זה אכן ה- $k$  הגרוע ביותר).

- נראה שוב חסם תחתון אקספוננציאלי ב- $n$ .

• נקבע  $n = N, k = \frac{N}{2}$  הערת – זה ה- $k$  הגרוע ביותר.

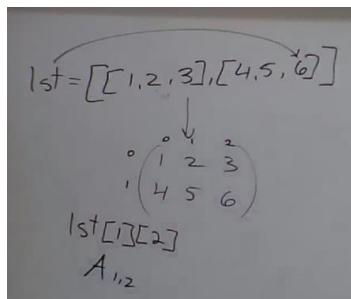
• עץ הרקורסיה:



נסתכל על  $\frac{N}{2} < j$  הרמות הראשונות – נראה שהן מלאות, ככלمر אנו לא מגיעים לתנאי העזירה:

$$\begin{aligned} n_j &= N - j > N - \frac{N}{2} = \frac{N}{2} & \bullet \\ \frac{N}{2} &\geq k_j \geq \frac{N}{2} - j > \frac{N}{2} - \frac{N}{2} = 0 & \bullet \end{aligned}$$

כלומר, קיבלנו כי  $k_j > j$  וגם  $0 > j$  ככלמר לאף אחד משני תנאי העזירה! גודל העץ הוא לפחות  $2^{N/2} = 2^{n/2} \cdot 2^{n/2}$ . לכן זמן הריצה הוא לפחות  $2^{n/2}$ .

ניתוח סיבוכיות fast binom – עם ממואיזציה:


יצורו ממבנה נתונים כדי לשמר את התוצאות שיחסבנו עד עכשיו, ולהימנע מקריאות וקורסיבות בבר

### התבצעו במהלך החישוב. לא חייבים להשתמש במילון, אפשר לייצג מטריצה (טבלה) באמצעות רשימות של רשימות:

**נזכיר: לגשת לאייר ברשימה זה (1) 0** – זה חישוב פשוט של אינדקסים, لأن ללכת בזיכרון. במילון גישה לאייר במרקחה הגרוע יכול להיות  $(n) 0$ .

**הממואיזציה חוסכת לנו קריאות וקורסיבות.**

ניתחנו באמצעות טבלה את זמן הריצה:

k\ n	0	1	2			k			n-k			n
0	*	*	*	*	*	*	*	*	*			
1	*	*	*	*	*	*	*	*	*			
2		*	*	*	*	*	*	*	*			
			*	*	*	*	*	*	*			
				*	*	*	*	*	*			
					*	*	*	*	*			
						*	*	*	*			
							*	*	*			
								*	*			
									*			
										*		
											*	
												*

זמן הריצה:

Num. of visited cells  $\times$  Num. of visits per cell  $\times$  time per cell visit (not including recursive calls)

עבדיו אנחנו רצים חסם עליון כדי להציג שהסיבוכיות כאן היא טובה. **באן צריך להבין מה ביקורים עושים בכל תא, בוגוד לעצם בכל צומת יש ביקור יחיד.** הוכחנו כי בכל תא אנחנו מבקרים מקסימום פעמיים (יש קריאה מ-2 תאים אחרים לפחות בכל היותר, מפורט במצגת של תרגול 7 ההוכחה המדוייקת). הגענו ל- $(k) 0$ .

### 3 – בעיתת העודף:

בכמה דרכים ניתן לפצל את  $\text{coins}$  באמצעות קומבינציות שונות של  $\text{coins}$ ? (מציר את  $\text{sum}$  subset אבל לא בדיק, כן יכולות להיות חזות – אפשר לבחור את אותו מטבע יותר מפעם אחת)

פתרוח וקורסיבי:

1. מקרי בסיס:
  - a. אם  $0 = amount$  אז יש דרך אחת להחזיר עודף (אף מטבע) – נחזיר 1.
  - b. אם  $0 > amount \neq 0$  אז אין דרך להחזיר עודף מדיוק – נחזיר 0.
2. תתי בעיות:
  - a. בלי להשתמש במטבע הראשון -  $change(coins[1:], amount)$
  - b. משתמשים במטבע הראשון (**לפחות פעם אחד!**) -  $change(coins, amount - coins[0])$
3. פתרון: נסכם את שתי הבעיות שלמו – מספר האפשרויות הכולל להחזיר את העודף  $amount$ .

הערה: במקרה ש- $amount$  יכול להיות שלילי, אנו מטפלים זהה בקורסיה (לא במקרה הבסיס), אם  $[0] \geq amount \geq coins$  בכל לא נכנים לקריאה השנייה (b) ולא מගיעים למצב שהוא שלילי.

פתרונות סיבוכיות: בחרנו קלט  $[n, \dots, 1, 2], amount = n^2$ , וזה מספיק בשבייל להראות שהסיבוכיות לא טובה. ניתוח דומה למה שעשינו קודם, נקבל כי מספר הצעדים הוא לפחות אקספוננציאלי.

**ממאיציה: גישה קצרה.** בטור הצומת עצמו, מבצעים את הבדיקה מול זיכרן העוזר לראות אם זה חשוב בבר. **בודקים על הקלט הנוכחי** האם הוא נמצא **במיון** (קודם לכן ביצענו את זה לפני הקריאה על קלט מסוים, האם הוא בבר במיון).

:choose sets – 4

הבעיה:

- קלט: רשימה ומספר  $k$
- פלט: כל תתי הקבוצות בגודל  $k$  של הרשימה.

פתרוח וקורסיבי:

1. מקרי בסיס:
  - a. אם  $0 = k$  יש תת-קבוצות אחת בגודל 0, הקבוצה הריקה – נחזיר [[]]
  - b. אם  $(len(lst) > k)$  – נחזיר []
2. תתי בעיות:
  - a. כל תתי הרשימהות שלא מכילות את האיבר הראשון –  $choose\_sets(lst[1:], k)$
  - b. כל תתי הרשימהות שכן מכילות את האיבר הראשון –  $choose\_sets(lst[1:], k - 1)$
  - c.
3. פתרון: נוסיף את  $[0]$  לבל רשימה בתת בעיה (b) – כי הן לא מכילות את האיבר הראשון.

**בעית N המלכות (תרגול 8):**

מלכה יכולה לזרז בצד אחד לביוון.இiom הדדי בין שתי מלכות יהיה באשר אחת מהן נמצאת בטוויח התנועה של המלכה השנייה בצד אחד – כלומר במשבצות שהמלכה השנייה מאויימת עליו (יכולת לזרז אליו).

נרצה לפתור את הבעיה עבר N מלבות באופן כללי. לכל  $4 \geq n$  יש לפחות פתרון אחד.

נעבור עם מטריצה (רשימה של רשימהות) בגודל  $N \times N$ . נסמן מלכה עם Q במטריצה. הקלט של הבעיה הוא לא-N שקיבלנו, אלא מטריצה בגודל  $N \times N$  עם  $N \leq k \leq 0$  מלבות. הפלט הוא בכמה דרכים ניתן להשלים את הלוח כך שיביל בדיק N מלבות בלבד.

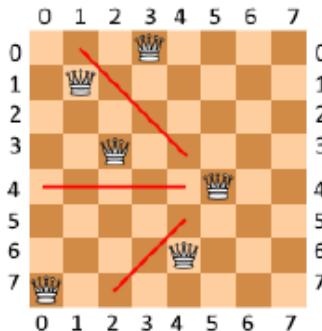
ניסוי ראשון:

- תנאי עצירה: אם מיקמנו N מלכות  $(N = k)$  אז נחזיר 1 – לא לגעת בלוח.
- יש לנו לוח עם  $N < k$  מלבות. צעד הרקורסיה: נאתחל מונה ל-0. לכל משבצת שבה אפשר למקם מלכה (בלי שתאים או תהיה מאויימת ע"י מלכה אחרת). נסיף מלכה, נקרא וקורסיבית ונוסיף את ערך החזרה למונה.

הפתרון הנוכחי בעיתוי. אפשר להחזיר את התשובה חלקית! N (כיו אנחנו סופרים פתרונות חוקיים יותר מפעם אחת בלבד, כל לוח עם k מלבות אנחנו סופרים!  $k$  פעמים). **זה מאוד לא יעיל.**

ניסוי שני:

בכל לוח תקין יש לבדוק מלכיה אחת בכל עמודה (ובאופן שקול בכל שורה) – נבנה את הלוח משמאלו לימין, וכך לא נפספס אף פתרון. ניצג כל לוח זמני עם **רשימה אחת באורך  $k$**  ולא עם מטריצה, כל תא ברשימה מייצג את השורה שבה יש מלכיה (כל תא מייצג אינדקס של עמודה משמאלו לימין). זה יהיה **הקלט החדש – הייצוג הקומפקטי של הלוח**.



- תנאי עצירה:  $N = \text{len}(\text{partial}) = 1$ , נחזיר 1.
- יש לנו רשימה באורך  $N < k$ . אנו שואלים איפה אפשר למקם מלכיה בעמודה  $1 + k$ . נאתחל מונה ל-0.
- לכל אינדקס  $N \leq i \leq 0$  אם אפשר לשים מלכיה בעמודה  $k$  שורה זו: נסיף, נקרא רקורסיבית, נסכם, נחזיר את המונה.
- עבור תא מסוים, לא צריך לבדוק את כל הצדדים – רק את מה שנמצא משמאלו **לא צריך לבדוק אלכסונים ימינה**. כי אנחנו בונים את הלוח משמאלו לימין, העמודה עצמה לא רלוונטית (אנחנו בונים אותה).

בהתעת  $\text{partial}$ , שורה  $i$  ועמודה  $k$  – איך בודקים אם המיקום  $i$  פנוי?

- איך בודקים את השורה  $i \in \text{partial}$  ?
- ההפרש בין העמודה והשורה נשאר קבוע בשרצים לאורך האלכסון העולה.
- הסכום בין העמודה והשורה נשאר קבוע בשרצים לאורך האלכסון היורד.

ניתוח זמן הריצעה:

ננתח את גודל העץ משני הרכיבים – גם חסם עליון וגם חסם תחתון.

- בرمה 0 של השורש יהיו לנו  $N$  קריאות וקורסיביות (יש  $N$  אופציות שצורך לבדוק). בرمה  $k$  יש לנו  $k$  מלכות שבכל אחת מהן "שורפת" את השורה שבה היא נמצאת וכן מספר הבנים המקסימלי יהיה  $k - N$ .
- במקורה הגרוע ביותר מלכיה שורפת 3 שורות שונות בעמודה אחת (עמודה שללה, אלכסון עולה ואלכסון יורדת). ואז בرمה 1 יהיו לנו מספר מינימלי של בנינים שהוא  $3 - N$ . ובאופן כללי בرمה  $-k$  יש לפחות  $3k - N$  בנינים.

## בעיית $N$ המלכות: סיבוכיות זמן

cutת נחסום את גודל העץ  $T_N$  מלמעלה ומלמטה:

### חסם עליון:

- בדרגה  $-k$  של העץ לכל צומת יש לכל היוטר  $k - N$  בנינים
  - לכן מספר הצמתים בעץ הוא לכל היוטר
- $$1 + N + N \cdot (N - 1) + \dots + N! = \sum_{i=0}^N \frac{N!}{i!} \leq N! \cdot \sum_{i=0}^{\infty} \frac{1}{i!} = N! \cdot e = O(N!)$$
- $$e \cdot N! = 2^{\theta(N \log N)}$$

### חסם תחתון:

- בדרגה  $-k$  עבור  $N/3 < k$  לכל צומת יש לפחות  $3k - N$  בנינים
- נסתכל על העמודה ה- $k$ . מספר הצמתים בה הוא לפחות:

$$N \cdot (N - 3) \cdot (N - 6) \cdots \left( N - \frac{N}{2} \right) \gg \left( \frac{N}{2} \right)^{\frac{N}{6}} = 2^{\theta(N \log N)}$$

- בסה"כ  $2^{\theta(N \log N)} \leq T_N \leq 2^{\theta(N \log N)}$

$$\left( \frac{N}{2} \right)^{\frac{N}{6}} = 2^{\log \left( \frac{N}{2}^{\frac{N}{6}} \right)} = 2^{\frac{N}{6} \log \left( \frac{N}{2} \right)} \geq 2^{\theta(N \log N)}$$

## 4 - מתמטיקה

### E – נושאים בתורת המספרים

#### העלאה בחזקה (Exponentiation of integers)

כלט: שני מספרים שלמים  $a, b$ , כאשר  $0 \leq b$ .  
פלט:  $a^b$ .

בניתוחי הסיבוכיות כאן נניח כי גודל הקלט הוא **כמויות הביטים** שנדריש כדי לייצג את המספרים.

#### 1 – Naïve algorithm

השיטה הנאיבית היא ביצוע לולאה פשוטה,  $\ell$  איטרציות, שבו נכפול את התוצאה שלמו- $b$ - $a$ .

- נשימט כאן את עלות פעולת הכפל. אם  $b = 0$  יש לנו א' ביטים אז  $a^0$  יהיה  $(a^2)^0$ . אבל אז מספר הביטים החדש הוא לפחות יותר  $2a$ .
- עלות פעולה הכפל גדול.** נספור רק כמה מכפלות מתבצעות. במקרה זה, בדיקה של כמה מכפלות מתבצעות תספיק לנו כדי להבין שהה קלט לא עילו.
- אנו מבצעים  $\ell$  פעולות בפל. **אם  $\ell$  הוא מספר בעל  $m$  ביטים, אז ערכו  $2^\ell < b \leq 2^{\ell-1} = 2^{n-1}$** , אם נגיד  $2^\ell = (n)g$  נקבל כי הוא כפולה של  $2^{\ell-1}$ , וזה סיבוכיות אקספוננציאלית. אפשר גם להגיד כי  $(\log b) = \Theta(n)$ .
- $\ell$  שומר בצורה בינארית ומקובל לנתח סיבוכיות בפעולות במספר הביטים – זה באמת גודל הקלט.

#### 2 – Iterated squaring algorithm

בשים לב כי:

- אם  $b$  אי-זוגי אז  $a^b = a^{b-1} \cdot a$
- אם  $b$  זוגי אז  $a^b = a^{\frac{b}{2}} \cdot a^{\frac{b}{2}} = \left(a^{\frac{b}{2}}\right)^2$

#### פיתוח סיבוכיות:

- אם  $b$  חזקה שלמה של 2 אז כל פעם שנחלק אותו ב-2 נמשך לקבל משהו זוגי, ואז נקבל בדיקון:  $O(n) = O(\log(2^n)) = O(\log b)$
- אם  $b$  אי-זוגי תהיה כאן עוד פעולה של חישוב  $1 - b$  ואולי נציג בין זוגי לא-זוגי.
- במקרה הגורע נציג בכה, אבל תמיד נחזור מאי-זוגי לזוגי. גם המקרה הגורע שעובר דרך האי-זוגיים – הוא יהיה גדול פי 2 לפחות, וגם שם זה יהיה  $O(n)$ .

שוב, אנו בודקים רק פעולות כפל. התעלמנו מפעולות אריתמטיות:

- פעולות חיסור –  $(a)O$ .
- לרוב לוקחת  $(n^2)O$  – floor division – לדוגמה –  $3 \cdot 3^2 \cdot 3^4 \cdot 3^8 \cdot 3^{16} \cdot 3^{32} \cdot 3^{64} \cdot 3^{128} \cdot 3^{256} \cdot 3^{512} \cdot 3^{1024} \cdot 3^{2048} \cdot 3^{4096} \cdot 3^{8192} \cdot 3^{16384} \cdot 3^{32768} \cdot 3^{65536} \cdot 3^{131072} \cdot 3^{262144} \cdot 3^{524288} \cdot 3^{1048576} \cdot 3^{2097152} \cdot 3^{4194304} \cdot 3^{8388608} \cdot 3^{16777216} \cdot 3^{33554432} \cdot 3^{67108864} \cdot 3^{134217728} \cdot 3^{268435456} \cdot 3^{536870912} \cdot 3^{1073741824} \cdot 3^{2147483648} \cdot 3^{4294967296} \cdot 3^{8589934592} \cdot 3^{17179869184} \cdot 3^{34359738368} \cdot 3^{68719476736} \cdot 3^{137438953472} \cdot 3^{274877906944} \cdot 3^{549755813888} \cdot 3^{1099511627776} \cdot 3^{219902325552} \cdot 3^{439804651104} \cdot 3^{879609302208} \cdot 3^{1759218604416} \cdot 3^{3518437208832} \cdot 3^{7036874417664} \cdot 3^{14073748835328} \cdot 3^{28147497670656} \cdot 3^{56294995341312} \cdot 3^{112589990682624} \cdot 3^{225179981365248} \cdot 3^{450359962730496} \cdot 3^{900719925460992} \cdot 3^{1801439850921984} \cdot 3^{3602879701843968} \cdot 3^{7205759403687936} \cdot 3^{14411518807375872} \cdot 3^{28823037614751744} \cdot 3^{57646075229503488} \cdot 3^{115292150459006976} \cdot 3^{230584300918013952} \cdot 3^{461168601836027904} \cdot 3^{922337203672055808} \cdot 3^{1844674407344111616} \cdot 3^{3689348814688223232} \cdot 3^{7378697629376446464} \cdot 3^{14757395258752892928} \cdot 3^{29514790517505785856} \cdot 3^{59029581035011571712} \cdot 3^{118059162070023143424} \cdot 3^{236118324140046286848} \cdot 3^{472236648280092573696} \cdot 3^{944473296560185147392} \cdot 3^{1888946593120370294784} \cdot 3^{3777893186240740589568} \cdot 3^{7555786372481481179136} \cdot 3^{15111572744962962358272} \cdot 3^{30223145489925924716544} \cdot 3^{60446290979851849432984} \cdot 3^{120892581959703698865968} \cdot 3^{241785163919407397731936} \cdot 3^{483570327838814795463872} \cdot 3^{967140655677629590927744} \cdot 3^{1934281311355259181855488} \cdot 3^{3868562622707518363710976} \cdot 3^{7737125245415036727421952} \cdot 3^{15474250490830073454843904} \cdot 3^{30948500981660146909687808} \cdot 3^{61897001963320293819375616} \cdot 3^{123794003926640587638751232} \cdot 3^{247588007853281175277502464} \cdot 3^{495176015706562350555004928} \cdot 3^{990352031413124701110009856} \cdot 3^{1980704062826249402220019712} \cdot 3^{3961408125652498804440039424} \cdot 3^{7922816251304997608880078848} \cdot 3^{15845632522609995217760157696} \cdot 3^{31691265045219985435520315392} \cdot 3^{63382530090439970870400630784} \cdot 3^{126765060180879941740801261568} \cdot 3^{253530120361759883481602523136} \cdot 3^{507060240723519766963205046272} \cdot 3^{1014120481447039533926410092544} \cdot 3^{2028240962894079067852820185088} \cdot 3^{4056481925788158135705640370176} \cdot 3^{8112963851576316271411280740352} \cdot 3^{16225927703152632542822561480704} \cdot 3^{32451855406305265085645122961408} \cdot 3^{64903710812610530171290245922816} \cdot 3^{129807421625221060342580491845632} \cdot 3^{259614843250442120685160983691264} \cdot 3^{519229686500884241370321967382528} \cdot 3^{1038459373001768482740643934765568} \cdot 3^{2076918746003536965481287869531136} \cdot 3^{4153837492007073930962575739062272} \cdot 3^{8307674984014147861925151478124544} \cdot 3^{16615349968028295723852302956249088} \cdot 3^{33230699936056591447704605912491776} \cdot 3^{6646139987211318289540921182493552} \cdot 3^{1329227997442263657908184236497104} \cdot 3^{2658455994884527315816368472994208} \cdot 3^{5316911989769054631632736945988416} \cdot 3^{1063382397953810926326547389197632} \cdot 3^{2126764795907621852653094778395264} \cdot 3^{4253529591815243705306189556790528} \cdot 3^{8507059183630487410612379113581056} \cdot 3^{17014118367260974821224782227162112} \cdot 3^{34028236734521949642449564454324224} \cdot 3^{68056473469043899284899128908648448} \cdot 3^{136112946938087798569792557817296896} \cdot 3^{272225893876175597139585115634593792} \cdot 3^{544451787752351194279170231268787584} \cdot 3^{1088903575504702388558340462537775168} \cdot 3^{2177807151009404777116680925075550336} \cdot 3^{4355614302018809554233361850151100672} \cdot 3^{8711228604037619108466723700302201344} \cdot 3^{17422457208075238216933447400604402688} \cdot 3^{34844914416150476433866894801208805376} \cdot 3^{69689828832300952867733789602417610752} \cdot 3^{139379657664601905735467579204835221504} \cdot 3^{278759315329203811470935158409670443008} \cdot 3^{557518630658407622941870316819340886016} \cdot 3^{111503726131681525888370633363868172032} \cdot 3^{223007452263363051776741266727736344064} \cdot 3^{446014904526726103553482533455472688128} \cdot 3^{892029809053452207106965066910945364256} \cdot 3^{1784059618106904414213930133821890728512} \cdot 3^{3568119236213808828427860267643781457024} \cdot 3^{7136238472427617656855720535287562914048} \cdot 3^{14272476944955235313711441070575325828096} \cdot 3^{28544953889910470627422882141150651656192} \cdot 3^{57089907779820941254845764282301303312384} \cdot 3^{114179815559641882509695328546602666247768} \cdot 3^{228359631119283765019390657093205332495536} \cdot 3^{456719262238567530038781314186410664981072} \cdot 3^{913438524477135060077562628372821329621544} \cdot 3^{1826877048954270120155125256745642659243088} \cdot 3^{3653754097908540240307550513491285318486176} \cdot 3^{7307508195817080480615101026982576636972352} \cdot 3^{1461501639163416096123020205396553327394464} \cdot 3^{2923003278326832192246040410793106654788928} \cdot 3^{5846006556653664384492080821586213109577856} \cdot 3^{11692013113307328768984161643172426219155712} \cdot 3^{23384026226614657537968323286344852438311424} \cdot 3^{46768052453229315075936646572689048866222848} \cdot 3^{93536104906458630151873293145378097732445696} \cdot 3^{187072209812917260303746586290756195464901392} \cdot 3^{374144419625834520607493172581512389899802784} \cdot 3^{748288839251669041214986345163026779799605568} \cdot 3^{1496577678503338082429772690326053559599211136} \cdot 3^{2993155357006676164859545380652107119198422272} \cdot 3^{5986310714013352329718890761304214238396844544} \cdot 3^{11972621428026704659437781522608428476793689088} \cdot 3^{23945242856053409318875563045216856953587378176} \cdot 3^{47890485712106818637751126090433733907175561552} \cdot 3^{95780971424213637275502252180867467814351123056} \cdot 3^{191561942848427274551004504361734935628702246112} \cdot 3^{383123885696854549102009008723469871257404492224} \cdot 3^{76624777139370909820401801744693974254980898448} \cdot 3^{15324955427874181964080360348938794850996178192} \cdot 3^{30649910855748363928160720697877589701992356384} \cdot 3^{61299821711496727856321441395755178423986713768} \cdot 3^{122599643422993455712642882791510356847973427536} \cdot 3^{245199286845986911425285765583020713695946855072} \cdot 3^{490398573691973822850571531166041427389893700144} \cdot 3^{980797147383947645701143062332082854779787400288} \cdot 3^{1961594294767895291402286124664165709559574800576} \cdot 3^{3923188589535790582804572249328321419119149601152} \cdot 3^{7846377179071581165609144498656642838238299202304} \cdot 3^{15692754358143162331218288997313285666476598404608} \cdot 3^{31385508716286324662436577994626571332953196809216} \cdot 3^{62771017432572649324873155989253142665863937618432} \cdot 3^{125542034865145298649746311978506285331727875236864} \cdot 3^{251084069730290597299492623957012570663455750473728} \cdot 3^{502168139460581194598985247914025413326911500947456} \cdot 3^{100433627892116238919797049582805822665822300189512} \cdot 3^{200867255784232477839594099165611645331644600378024} \cdot 3^{401734511568464955679188198331223290663289200756048} \cdot 3^{803469023136929911358376396662446581326578401512096} \cdot 3^{1606938046273859822716752793324893162653556803024192} \cdot 3^{3213876092547719645433505586649786325307113606048384} \cdot 3^{6427752185095439290867011173299572650614227212096768} \cdot 3^{12855504370188678581734022346599145201228544244193536} \cdot 3^{25711008740377357163468044693198290402457088488387072} \cdot 3^{51422017480754714326936089386396580804904176976744144} \cdot 3^{10284403496150942865387217877279316160980835395348828} \cdot 3^{20568806992301885730774435754558632321961670790697656} \cdot 3^{41137613984603771461548871509117264643833341581395312} \cdot 3^{82275227969207542923097743018234529287666683162785624} \cdot 3^{164550455938415085846195486036469558573333366325571248} \cdot 3^{329100911876830171692390972072939117146666732651142496} \cdot 3^{65820182375366034338478194414587823429333346530228492} \cdot 3^{13164036475073206867695638822917646685866673106045688} \cdot 3^{26328072950146413735391277645835293371733346212091376} \cdot 3^{52656145900292827470782555291670586742466681424182752} \cdot 3^{105312291800585654941565110583341734849333428448365504} \cdot 3^{21062458360117130988313022116668346889866685689673008} \cdot 3^{42124916720234261976626044233337693377733371379346016} \cdot 3^{84249833440468523953252088466675387555466742758689032} \cdot 3^{168499666880937047906504176933350775110933485517378064} \cdot 3^{336999333761874095813008353866701552221866911034756128} \cdot 3^{673998667523748191626016707733403104443733822069512256} \cdot 3^{1347997335047496383252033415466806208887467644139024512} \cdot 3^{2695994670094992766504066830933612417774935388278049024} \cdot 3^{5391989340189985533008133661867224835549357776556098048} \cdot 3^{10783978680379971066016267323734449671098715553112196096} \cdot 3^{21567957360759942132032534647468899342197431106224392192} \cdot 3^{43135914721519884264065069294937798684394862212448784384} \cdot 3^{86271829443039768528130138589875597368789324424897568768} \cdot 3^{172543658886079537056260277179751194737578648849795137536} \cdot 3^{345087317772159074112520554359502389475157297799590275072} \cdot 3^{690174635544318148225041108719047778950314595599180550144} \cdot 3^{1380349271088636296450082217438095557806289911198361100288} \cdot 3^{2760698542177272592900164434876191115612578822396722200576} \cdot 3^{5521397084354545185800328869752382231225157644793444401152} \cdot 3^{11042794168709090371600657739504764462453113295868888802304} \cdot 3^{22085588337418180743201315479009528924906226591737777604608} \cdot 3^{44171176674836361486402630958019057848812453083475555209216} \cdot 3^{88342353349672722972805261916038115686424906166951110418432} \cdot 3^{176684706699345445945610523832076231372849123333875550836864} \cdot 3^{353369413398690891891221047664152462745698246675551101673728} \cdot 3^{706738826797381783782442095328308925491396493371102203347456} \cdot 3^{141347765359476356756488419065661785098279298674220440669512} \cdot 3^{282695530718952713512976838131334170196558597348440881339024} \cdot 3^{565391061437905427025953676263343403803117194688881762678048} \cdot 3^{1130782122875810854051907342526686807606234389777763525356096} \cdot 3^{2261564245751621708103$

הערך של שמות הולאה הוא תמיד אותו הדבר בכל איטרציה. אמרנו כי  $a_0^{b_0} = result \cdot a^b$ , כאשר  $b_0, a_0$  הקלטים המקוריים של הפונקציה. ניעזר בהזאת כדי להוכיח את נכונות הפתרון. נוכיח באינדוקציה:

- בסיס – שמות הולאה נכונה לפני האיטרציה הראשונה.

Base: The first time we enter the loop

$$result = 1, a = a_0, \text{ and } b = b_0$$

so the condition is true.

- צעד – בנסיבות שהזאת נכונה עבור איטרציה  $i$ , זה נכון גם עבור  $i+1$ .

o אם  $b$  אי זוגי:

If  $b$  is odd, then

$$result' = result \cdot a$$

$$b' = (b - 1)$$

$$a' = a \quad \text{unchanged}$$

$$\text{So: } result' \cdot (a')^{b'} = result \cdot a \cdot a^{b-1} = result \cdot a^b = a_0^{b_0}$$



Substitute the values



Inductive assumption

o אם  $b$  זוגי:

If  $b$  is even, then

$$result' = result \quad \text{unchanged}$$

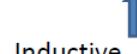
$$b' = b/2$$

$$a' = a^2$$

$$\text{So: } result' \cdot (a')^{b'} = result \cdot (a^2)^{b/2} = result \cdot (a)^b = a_0^{b_0}$$



Substitute the values



Inductive assumption

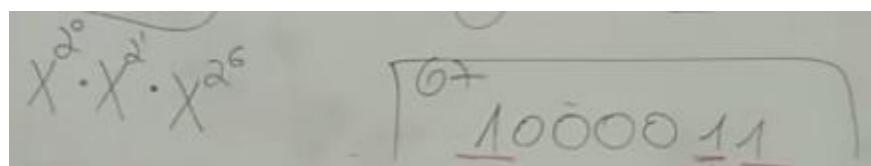
- סיום – נראה שהולאה אכן נכונה. הסיבה לכך היא שבכל איטרציה  $b$  קטן לפחות ב-1. בשיצאים מהולאה  $0 = b$ .

$$\text{So: } result \cdot a^b = result \cdot a^0 = result = a_0^{b_0}$$

Binary Interpretation: ראיינו דרך לפשט את הקוד, כאשר משתמשים על כרך ש- $b$  מספר בינארי, ומוכל המשיך ולבצע  $/b$ , כך נעיף את הביט הימני ביותר ונקבע את המספר פי 2 (בין אם יש שם 0 או 1 – זאת כיון שאחרי איטרציה שבת  $b$  אי זוגי, תמיד תגיעה איטרציה שבה הוא זוגי).

result	a	b	
1	X	67	1000011
1·X	X <sup>2</sup>	33	100001
1·X·X <sup>2</sup>	X <sup>4</sup>	16	10000
"	X <sup>8</sup>	8	1000
"	X <sup>16</sup>	4	100
"	X <sup>32</sup>	2	10
"	X <sup>64</sup>	1	1
1·X·X <sup>2</sup> ·X <sup>64</sup>	X <sup>128</sup>	0	-

חויבנו מכפלות במספר הביטים הדלוקים בייצוג הבינארי של  $a$ , זה **מספר הפעמים שהוספנו משווה ל-result**. מתוך כל החזקות הרלוונטיות לאורך הדריך, אנו **אוסףים את החזקות שנחוצות כדי לחשב את  $a^b$** , נחוצות רק  $2^0, 2^1, 2^2, \dots, 2^{n-1}$ .



עבור מספר כללי נקבל (הבית ה- $i$ -ן יכול להיות תמייד 0 או 1):

$$\text{Then } a^b = a^{\sum_{i=0}^{n-1} b_i \cdot 2^i} = \prod_{i=0}^{n-1} (a^{b_i \cdot 2^i}) = \prod_{b_i=1} (a^{2^i})$$

$\uparrow$   
 $a^{x+y} = a^x \cdot a^y$

אבחנות:

- הקוד עדין מחשב את החזקה  $a^{2^{128}}$  שבירר אין בה צורך (דוגמה מקודם).
- אם  $b$  היה רק אחדים, אז הוא  $1 - 2^n$  עבור  $n$  בלבד. כאשר נקוץ ממנו את הביט הימני ביותר, נקבל  $1 - 2^{n-1}$ , מספר שהוא עדין א-זוגי.

סיכום עד כה:

Given two integers  $a, b$ , where  $b \geq 0$  and the size of  $b$  is  $n$  bits, namely  $2^{n-1} \leq b < 2^n$ :

- The **Naïve** algorithm takes  $b$  multiplications, which is between  $2^{n-1}$  and  $2^n - 1$
- **Iterated squaring** takes between  $n - 1$  and  $2(n - 1)$  multiplications

$O(2^n)$   
multiplications

$O(n)$   
multiplications

So naïve is **exponentially slower** than iterated squaring!

### 3. העלאה בחזקה מודולרית:

במספרים גדולים מאוד אנחנו נתקלים בבעיה. במקרה זה נרצה לחשב לא את  $a^b \bmod c$ , אך שלא בעבר דרך מספר מאוד גדול שלא ניתן לייצג במחשב ( $a^b$  יכול להיות גדול מידי).

הקוד דומה, רק שbulk פעמיון מבצעים  $c \bmod$  ו- $a \bmod c$  יותר. בהסתמך על הנוסחה הבאה:

$$(a \cdot b) \bmod c = ((a \bmod c) \cdot (b \bmod c)) \bmod c$$

הערך המקסימלי תמיד יהיה  $(1 - c)^2$ . בכל איטרציה החישוב של result יהיה  $O(n^2)$ . אך גם החישוב של  $a$ . יש לנו  $(n)$  פעולות כפל, כל פעולה כפלה עולה  $O(n^2)$  ולכן **סיבוכיות הזמן היא  $O(n^3)$**  (כאשר  $c, b, a$  מספרים עם כל היotta  $n$  ביטים).

## בדיקות ראשוניות הסתברותיות (המשפט הקטן של פרמה)

מספר ראשוני הוא מספר שאין לו אף מחלק נוסף חוץ מעצמו ו-1. מספר שאיןו ראשוני נקרא מספר **פריק**.

ובכך כי קיימים אינסוף ראשוניים:

נניח בשלילה כי קיימים  $k$  ראשוניים (באשר  $k$  סופי) – כלומר  $p_1, \dots, p_k$ . נסתכל על המספר  $1 + p_1 \cdot \dots \cdot p_k = z$ . אף אחד מבין  $p_1, \dots, p_k$  אינו מחלק של  $z$ . קיימות שתי אפשרויות:

1.  $z$  הוא ראשוני בעצמו – זו סתרה.
2.  $z$  אינו ראשוני – לכן ניתן לפרק אותו לגורמים ראשוניים שאינם  $p_1, \dots, p_k$ .

טענה: מספר רנדומלי עם  $n$  ביטים הוא ראשוני עם הסתברות של  $(\frac{1}{n})^0$ .

### Trial Division לבדיקה ראשונית:

נרצה להיות מסוגלים לבדוק האם מספר הוא ראשוני.

הקלט הוא מספר  $N$  שמיוצג על ידי  $n$  ביטים – כלומר  $\Theta(2^n) = N$ .

- אם  $N$  הוא פריק אז אפשר לכתוב אותו כמכפלה  $KL = N$  כאשר  $N < L < K$ .
- לפחות אחד מבין  $L, K$  קטן שווה ל- $\sqrt{N}$ .
- לכן נעבור על כל המספרים החל מ-2 עד  $\sqrt{N}$  ונבדוק האם הם מחלקים את  $N$ .

האלגוריתם הנ"ל מאד לא עלייל:

- $2^{n-1} \leq N < 2^n$
- $\sqrt{2^{n-1}} = 2^{\frac{n-1}{2}} \leq \sqrt{N} < 2^{\frac{n}{2}} = \sqrt{2^n}$
- בולומר נקבל כי  $\sqrt{2^n} = \sqrt{N}$ . זה אקספוננציאלי ולכן גורע מאוד.
- מקרה גורע – כל האיטרציות.
- $N$  ראשוני.
- $N$  פריק אבל מהצורה  $a^2 = N$  כאשר  $a$  ראשוני.
- מקרה טוב – מספר זוגי, איטרציה אחת.

האמונה היא שאין אלגוריתם פולינומייאלי כדי לפתור את הבעיה זו. מה לגבי בעית ההכרעה? רק להגיד האם מספר הוא פריק/ראשוני או לא? בלי לתת את הגורמים. על שאלת זו אפשר לענות ביעילות (ambil לדעת שום דבר על המחלקים של  $N$ ).

### :Primality Testing

ה**העיוון הבסיסי** – כדי להראות ש- $N$  פריק, מספיק למצאו עדות לכך שהמספר לא מתנהג כמו ראשוני. עדות כזו לא כוללת שום מחלק של  $N$ . "עד" – הוכחה קונקרטית שמצויה טעונה מסוימת. בהינתן מספר  $N$ , מחפשים עדות לכך ש- $N$  הוא פריק. נראה 3 סוגי "עדים" לפרקות של מספרשלם  $N$  אי-שלילי.

**המשפט הקטן של פרמה** – אם  $k$  ראשוני, אז לכל  $1 - a \leq 1 \text{ מתקיים } 1 \equiv a^{p-1} \pmod{p}$  (מבחן פרמה).  
 מסקנה מהמשפט – נתון מספרשלם אי-שלילי כלשהו שנסמן ב- $N$ . אם קיימ  $1 - a \leq N - 1 \equiv a^{p-1} \pmod{p}$  – כלומר  $a^{p-1} \neq 1 \pmod{p}$  – כלומר מבחן פרמה נכשל, נסיק כי  $N$  אינו ראשוני (ולכן פריק).

סוג	הגדרה	איך מוצאים
$FACT_N - 1$	זהוי קבוצת המחלקים של $N$ .	Trial Division
$GCD_N - 2$	קבוצת הלא-זרים של $N$ . כל המספרים $a$ כך $\text{gcd}(N,a) > 1$ .	נורץ על הערכים ונבדוק את התנאי. חישוב $\text{gcd}$ נראה בהרצתה הבאה.
$FERM_N - 3$	קבוצת המספרים שעבורם מבחן פרמה נכשל.	עוברים על כל הערכים בתחום הרלוונטי, וביצעו את מבחן פרמה. <b>איטרציות - <math>n^2</math>, סיבוכיות זמן (העלאה בחזקה כבר) - <math>O(n^3)</math></b>

בין שלוש הקבוצות האלה מתקיים:  $FACT_N \subseteq GCD_N \subseteq FERM_N$  (הוכחנו זאת בתרגול 8), זו הסיבה שאנו בודקים באלגוריתם רק את מבחן פרמה ולא את האחרים.

נעבור עם אלגוריתם שמבצע **הגולות**. אלגוריתם רנדומרי לביקורת ראשונית של הקלט N:

- נבחר a אקראי בתחום  $1 \leq a \leq N$  (באופן בלתי תלוי, הבחירה הבאה אינה קשורה לבחירה הקודמת)
- אם a עובר את מבחן פרמה – a אינו עד נחזר True (prime)
- אם a לא עובר את מבחן פרמה – a הוא עד לכך ש-N אינו ראשוני, נחזר False (composite)

נשים לב:

- אם N אינו ראשוני והגלונו a שהוא עד – נחזר False וזה התוצאה הרצiosa.
- אם N אינו ראשוני והגלונו a שאינו עד – נחזר True וזה אינה התוצאה הרצiosa. זה בעייתו.
- אם N ראשוני אך כל a שנגזר לו נחזר True – וזה נכון.

כדי לבדוק יותר עבור N שאינו ראשוני, לבצע את זה 100 פעמים, ואם אחד או יותר מה-a הוא עד נאמר כי N הוא פריך, ואם לא מצאנו עד N ראשוני.

מספרים Carmichael הם מספרים פריקים, שבעורם מתקיים  $GCD_N = FERM_N$  – אין אף מספר שהוא עד FERM שהוא לא עד GCD. עבורם לא מתקיים שחזci מהערכיכים בתחום המעדים. עבור מספרים כאלה בסיסically גבוהה יותר מאשרו,

### פרוטוקול דיפי-הלמן (Diffie-Hellman)

Alice ו-Bob רוצחים לקיים תקשורת מוצפנת.

- E – אלגוריתם הצפנה. בלחומר  $E(msg, K_{AB}) = \text{encrypted msg}$
- D – אלגוריתם פענוח.  $D(\text{encrypted msg}, K_{AB}) = msg$
- $K_{AB}$  – מפתח סודי משותף.

איך ליצור מפתח סודי משותף שלא יהיה קל לפענה אותו חישובית? דיפי והלמן תיארו פרוטוקול למציאת מפתח סודי משותף. הפרוטוקול מסתמך על בעיה חישובית קשה.

- קיים p ראשוני גדול, ו- $g$  אקראי בתחום  $1 < g < p - 1$ .
- בהינתן ערך a נרצה לחשב את  $p \bmod g^a$  (זה העלה בחזקה מודולרית) זה בינוון קל לחישוב –  $O(n^3)$ .
- בהינתן התוצאה – כיצד נמצא את a? זהו בינוון קשה לחישוב. לא קיים אלגוריתם פולינומילי שפותר את זה. הכוון זהה נקרא בעיתת הלוג הדיסקרטי.

מהלך הפרוטוקול:

### Diffie and Hellman Key Exchange

- **Public parameters:** A large prime  $p$  (1024 bit long, say) and a random element  $g$  in the range  $1 < g < p - 1$ .
- Alice chooses at random an integer  $a$  from the interval  $[2..p - 2]$ . She sends  $x = g^a \pmod p$  to Bob (over the insecure channel).
- Bob chooses at random an integer  $b$  from the interval  $[2..p - 2]$ . He sends  $y = g^b \pmod p$  to Alice (over the insecure channel).
- Alice, holding  $a$ , computes  $y^a = (g^b)^a = g^{ba} \pmod p$ .
- Bob, holding  $b$ , computes  $x^b = (g^a)^b = g^{ba} \pmod p$ .
- Now both have the shared secret,  $g^{ba} \pmod p$ .
- An eavesdropper cannot infer the key,  $g^{ba} \pmod p$  after seeing "only"  $p$ ,  $g$ ,  $x = g^a \pmod p$  and  $y = g^b \pmod p$  (under the assumption that discrete log is intractable).
- We have just witnessed a **small miracle** !

הערות:

- בעית הלוג הדיסקרטי הוא חסם עליון על הקושי של בעית דיפי הלמן. דיפי הלמן קשה בכל היותר במו לוג דיסקרטי (אבל יכולה להיות קשה פחות, תלוי מה יוכיחו בעתיד).
- מנחים ש-eve פסיבית.

**מחלק משותף מקסימלי (GCD)**הבעיה:

- קלט: שני מספרים שלמים חיוביים –  $m$ ,  $k$ .
- פלט: המספר השלם הגדול ביותר  $g$  שהוא המחלק של שניהם.

שני מספרים שמקיימים  $1 = \text{gcd}(k, m)$  נקראים זרים-הבדית.

השיטה הנאיבית: בצע איטרציה ונבדוק את כל המספרים הרלוונטיים. ה-T WCT הוא  $\min(k, m)$ . אם המספר המינימלי מוצג על ידי  $n$  ביטים, יש לנו  $O(2^n)$  איטרציות שבהן איטרציה בצע 2 חלוקות –  $O(n^2)$ .

אלגוריתם אוקלידס: אוקלידס פיתח אלגוריתם איטרטיבי שבבסיסו על Invariant (תכמה שמתקיימת לפני ואחרי SMBZ) פעולה (בשלהי). האינוריאנט שהוא מצא הוא:  $\text{gcd}(m, k) = \text{gcd}(m, k \bmod m)$ . כלומר, שני המספרים קטנים יותר מחצי מהמספר אליו התחלנו. מ בנות הביטים במספר, אז נקבל סה"ב  $(n) = O(\log(2^n)) = O(n)$ .

## F – חישוב נומרי

אנו מחפשים פתרון מספרי לביעות כלשהן. ברוב המקרים נתעסק עם מספרים ממשיים, והיצוג במחשב הוא כזה שיש בו בעיות דיווק (floating point representation).

**מציאת שורש של פונקציה**

נניח שקיבלנו פונקציה כלשהי (לא פירוט)  $f : \mathbb{R} \rightarrow \mathbb{R}$ . אנו רוצים למצוא את השורש שלו, כלומר ערך שבו הפונקציה מתאפשרת. בזיהוי שהධוק מוגבל בדרך כלל לפחות למוצא מקום  $\epsilon < |f(a)|$  – במעט שורש. אם נדע ש- $f$  רציפה, יוכל להשתמש במשפט ערך הביניים. נזכיר פה מהו שדומה לחיפוש ביןاري, "שיטת החציה".

תנאי עצירה:

- קרוב מספיק  $\epsilon$ .
- אורך האינטראול קטן מידי בשבייל הדיווק של floating point.

איך נחשב את שורש  $f$ ? נגדיר פונקציה  $g(x) = f(x)^2 - 2$  ונשתמש באורה השיטה כדי לחשב את שורש  $g$ .

**קירוב לפאי**

פאי הוא מספר טרנסידנטלי – הוא לא שורש של אף פולינום עם מקדמים ממשיים. השתמש בלחוב 1 על 1, בלחוב מטרה לחיצים. נזכיר שהפגיעה בלוח תהיה רנדומלית לגמרי. שטח כל הלוח הוא 1, שטח רביע העיגול המסומן הוא  $\frac{\pi}{4}$ . היחס בין רביע העיגול לבין כל השטח הוא  $\frac{\pi}{4}$ . נספר את היחס בין כל החיצים שזרקנו בתוך העיגול לכל החיצים total.

**חישוב נגזרות וaintegלים**

נגזרת: אנו לא יכולים לחשב באמצעות את הגבול של  $\frac{h}{h}$  שואף לאפס, אך נגדיר  $\delta$  קטן מאוד. נשים לב כי אם נקטין את  $\delta$  יותר מידי, בגלל אי הדיווק של היצוג במחשב תוכל לקבל בעיות חלוקה ב-0.

## 5 – OOP ומבנה נתונים

### OOP – מבני נתונים

#### מבוא

OOP – גישה תכניתית. python מאפשרת לבתו OOP, אבל טהראנים של OOP יאמרו שהוא לא מדויק. נרחיב על גישה זאת בתוכנה 1 בשפת Java. גישה זו ישות ממודلات בתור עצמים (objects) שהם בעלי:

1. **שדות** (attributes/fields) – פרטיו מדע מייצגים את מצב העצם.
2. **מתודות** (methods) – פעולות שיכולה להיות מופעלת על העצם.

**מחלקה** (class) – טמפליט/תבנית ליצירת אובייקטים. עצם מיוצר בתור **מופע** של מחלקה.

ראינו דוגמה למחלקה שמייצגת סטודנט:

```
class Student:

    def __init__(self, name, surname, ID):
        self.name = name
        self.surname = surname
        self.id = ID
        self.grades = dict()

    def __repr__(self): #must return a string
        return "<" + self.name + ", " + str(self.id) + ">"

    def update_grade(self, course, grade):
        self.grades[course] = grade

    def avg(self):
        s = sum([self.grades[course] for course in self.grades])
        return s / len(self.grades)
```

`__init__` and `__repr__`  
are special standard methods,  
with pre-allocated names.  
More on this coming soon.

שmeno לב לדברים הבאים:

- `self` – מצביע על המופיע הנוכחי. כל פונקציה פנימית **שהאובייקט מפעיל** מקבלת את `self` בתור פרמטר.
- `init` – הבנאי (constructor) של המחלקה. נקרא אוטומטית באשר יוצרים את האובייקט, ומוסברים אליו הארגומנטים הרלוונטיים ליצירת האובייקט.
- `repr` – אם לא ממשים אותו, ההנהגות הדיפולית של פייתון היא להדפיס מידע כללי על האובייקט.
- **מתודות מיוחדות** – יש מתודות שנקבעות באשר אנו מבצעים פעולות מסוימות:

You Want...	So You Write...	And Python Calls...
to initialize an instance of class MyClass	<code>x = MyClass()</code>	<code>x.__init__()</code>
the “official” representation as a string	<code>print(x)</code>	<code>x.__repr__()</code>
addition	<code>x + y</code>	<code>x.__add__(y)</code>
subtraction	<code>x - y</code>	<code>x.__sub__(y)</code>
multiplication	<code>x * y</code>	<code>x.__mul__(y)</code>
equality	<code>x == y</code>	<code>x.__eq__(y)</code>
less than	<code>x &lt; y</code>	<code>x.__lt__(y)</code>
for collections: to know whether it contains a specific value	<code>k in x</code>	<code>x.__contains__(k)</code>
for collections: to know the size	<code>len(x)</code>	<code>x.__len__()</code>
... (many more)		

למשל, כל עוד לא מימשנו את `Eq`, פיתון משווה בין אובייקטים לפי הכתובות שלהם בזיכרון. כך `s1 == s2` יניב `False`. אם נממש את הפונקציה ונגידו כיצד להשוות בין שני האובייקטים זה יעבד. במשמעות וידאנו כי האובייקט השני שהוא אכן `Student` מקבלים על ידי הפונקציה `isinstance`.

### ביד לתוכן מחלוקת ב-OO:

1. לקבוע מה יהיו המתוודות, ה-API שהמחלקה מספקת.
2. לקבוע כיצד ניצג את `state` של העצמים מהמחלקה (על ידי השdots), כך שהמתוודות יוכל להתבצע באופן יעיל. לאחר מכן ניתן למש את הבנייה `__init__`.
3. למש את הקוד של שאר המתוודות.

### **רשימות מקשורות (Linked Lists)**

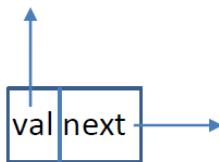
בשימוש המבונה של פיתון, מדובר על **מערך ב-C**, כלומר זהו מקטע זיכרון רציף של מצביעים. נציג אלטרנטיבה לרישימה זו: רישימה מקושרת, שגม היא מייצגת רישימה (ADT – מבנה נתונים אבסטרקט). נתבונן תחילתה בשימוש המבונה של רישימה:

#### יתרונות:

- כדי לגשת לאייר כלשהו אפשר לחשב את הכתובת שלו **ולגש אליו מידית ב-(1) O בגישה ישירה** (random access).
- ניתן לעשות `[1, "hello", [1,2] = L`, כיוון שמדובר במצביעים, אז אפשר להחזיק אותה רישימה איברים מסוגים שונים.

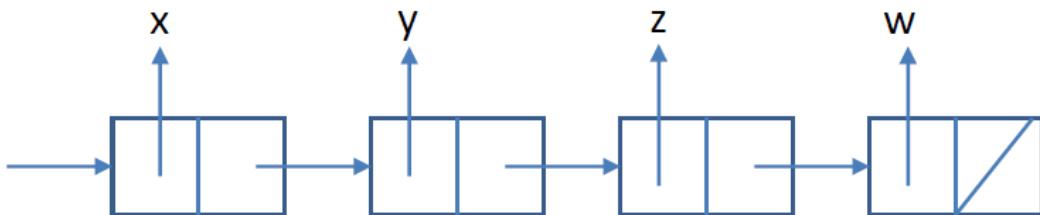
#### חסרונות:

- הוספה – פיתון לוקח "מקדם ביטחון" ומגדיל את גודל הרישימה לא-`1` אלא `2` למשל,ऋיך להזיז קדימה את כל האיברים – **זהו סיבוכיות של (n) O** במרקחה הגורע ביותר.
- זה קורה גם במחיקה –ऋיך להזיז את האיברים הרלוונטיים זהה גם `(n) O`.



רישימה מקושרת – נחזיק מצביעים ל-values אבל במקומות שונים בזיכרון. איך נדע אילן הם נמצאים? כל אייר מכל מצביע לאייר הבא, מצביע בשם `next`. כל אייר ברישימה יהיה מסוג `Node`.

- פתרון זה יקל על הוספה ומחיקה והן ייעשו ב-(1) **O**.
- אבל החסרון הוא שכרגע **random access** כבר לא קיים – צריך לטויל על כל השרשרת עד שמצאים אייר. אין דרך לкопץ ישר לאייר ה-1000.



התבוננו ב:

- מימוש להוספה במקום הראשון ברישימה (מעקב אחר מספר הרצאות – במצגת). סדר הפקודות חשוב, על מנת לשמור את המיקום בזיכרון של האיבר הקודם שהוא ראשון ברישימה. אם שומרים `my` במחלקה `LinkedList` אז נdag לעדן אותו (להגדיל את הערך שלו ב-1).
- המימוש של `_repr_` עם הולאה המפורשת `None != p while`, כך נבצע ריצה על השרשרת של החוליות עד שנגיעה לאחרונה. המבנה הזה חוזר על עצמו בפונקציה `print` שמחפש איבר ברישימה. **סיבוכיות מקרה גורע (n) O**.
- המימוש של `getitem` – אם נמשש את `indexing` – יכול להשתמש בסוגרים מוחבעים על האובייקט! ולבצע `[] lst_my` למשל. מאחרו הקלעים אנו עוברים על הרישימה עד המיקום המבוקש ומחזירים את הערך שהגענו אליו. גם כאן **mareka gorut (n) O**. כאשר אנו כתובים את הערך צריך למש את הפעולה `setitem`.

מה פחות **יעיל** לממש? **חיפוש ביןארי** – ברישימה רגילה אפשר לעשות חיפוש ביןארי כי אפשר לגשת באופן מיידי באמצעות הרישימה ואך הרבה וכו (`random access`). כאן אפשר למשש את זה, אבל הגישות שאנו עושים לחצי/רביע הרישימה לא יהיה ב-(1) **O** ואך לא נחשס שום דבר. חיפוש ביןארי יהיה פחות **יעיל** ברישימה מקושרת.

הכנסה במקום ספציפי ([אינדקס loc](#)) – כאשר נרצה להכניס איבר חדש במקום מסוים שאינו בהתחלה, צריך לזכור צעד אחד אחר זה. לכן נורץ בוללת את `for` עד האינדקס 1-`loc`, כדי לחבר את האיבר החדש.

סיכום רshima מקושרת מול רshima וגיליה:

Method	List	Linked list
init	$O(1)$	$O(1)$
init with k items	$O(k)$	$O(k)$
<b>for the rest of the methods we assume that the structure contains n items</b>		
add at start	$O(n)$	$O(1)$
add at end	$O(1)$	$O(n)$
<code>lst[k]</code> (get the k-th item)	$O(1)$	$O(k)$
delete <code>lst[k]</code>	$O(n-k)$	$O(k)$

### :Cycle Detection

- פתרון נאובי – לשמר ב-set את כל ה-ids של Nodes שבירנו בהם. אם ה-`id` כבר ב-set נჩזר ש褪. ה-`id` הוא מזהה ייחודי של האובייקט Node. הביעות היא סיבוכיות זמן ריצה ( $O(n^2)$ ) וסיבוכיות זיכרון ( $O(n)$ ).
- נפתר בצורה יותר פשוטה, בסיבוכיות ( $O(n)$ ) בשיטת "הצב והארנב". הארנב יתקדם ב-2 צעדים כל פעם, והצב בצעד אחד. ברגע שהצב נכנס למגעל, הארנב נמצא איפשהו בפנים. בשלב בלשחו, הארנב יגיע אל הצב, ואם הם נפגשים אז בזודאות יש מגעל בגרף. אפשר לשים לב שגם אין מגעל בgraf, הם יכולים לא יפגשו (הארנב תמיד מישיג את הצב). האבחנה היא שאם במבנה למגעל הארנב צריך עוד  $j$  צעדים כדי להגיע לצב, בכלל במקרים בהם הם עושים, בכל איטרציה  $j$  קטן ב-1 וכשהוא יגיע ל-0 הם יפגשו. **כלומר עד הפגיעה יהו בדיקת איטרציות.**

### תרגיל 9:

**מהבחן 2018:**

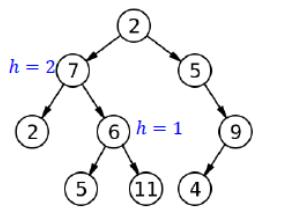
- אם הרשימות ממוזגות – ה-Node האחרון שלון זהה. כלומר כדי לבדוק אם שתי רשימות ממוזגות, לבדוק רק את ה-**Node האחרון שלון**.
- בעת נרצה לדעת מהו ה-Node שבו הרשימות מתמצאות. נתחל בהקלת שב האורך שלון זהה – נעבור עם שני מצביעים ונבדוקמתי הם שוויים. מה געשה במקרה שהרשימות לא באוטו אוור?
- אם יש אורכים אפשר להשתמש בהם, או בזמן לינארי למצאו אותם (לרחוץ על כל הרשימה). ואז נקוץ את הרשימה הארוכה יותר ומשם נשווה איבר איבר. אבל נסתכל על **פתרון יותר מגביב**.
- אחרי שהגענו לסוף של רשימה מסוימת, ניקח את המצביע ונעביר אותו **לתחילת הרשימה השנייה**. נמשיך כך עד שנגיע לקודקód משותף וזה הקודקód שבו הרשימות מתמצאות. עמרי הסביר על הלווח עם  $c, b, a$  צעדים. בדיק אחרי  $c + b + a$  צעדים שני המצביעים יגיעו לאותו הקודקód.

### מבחון 2022 – רshima מקושרת לוגריתמית:

- מימוש נוסף לרשימה, בה כל צומת מצביע לכל הצמתים שנמצאים  $i^2$  צמתים אחרים. כלומר נחזיק רshima של מצביעים, המצביע באינדקס 0 יצביע לצומת הבא, באינדקס 1 יצביע לצומת שנמצא 2 אחריו, באינדקס 2 יצביע לצומת שנמצא 4 אחריו וכן הלאה.
- כדי להוסיף Node בתחלת הרשימה, צריך לסדר רק את המצביעים שלו, כל מה שקרה אחר כך עדין עובד ואין צורך לגעת בהצהה. אבחנה חשובה כדי לעשות את זה בצורה ייעילה -  $[1 - i][i] = node[i] = node[1 - i]$  כי האיבר הראשון הוא פשוט `head` ועוד האיבר השני הוא איבר 1 ברשימה של איבר 1, האיבר השלישי הוא איבר 2 ברשימה של איבר 2 ...
- כדי להגיע לאייר ה- $n$ -nvבע קפיצות לפי הפירוק של האינדקס הבא בbinary, למספרים שסקומם נותנים. למשל 13 ניתן על ידי סכום של 1, 4, 8 וזה הדרך הכי קצרה להגיע. איך נבון מה הקפיצות?
- אם ביט דולק (1) נ Kapoor בהתאם את החזקה שמתאיימה לו. הקפיצה היא בעצם האינדקס הבא בתוך `list.next`, כי כל אינדקס כזה הוא גדול פי 2 מהאינדקס הקודם לו ב-`list.next`. לדוגמה  $list[0].next$  זה Kapoor ב- $2^0$ . לדוגמה  $list[2].next$  זה Kapoor ב- $2^2$ . זה החישוב המחייב של קפיצות שיביא אותנו ל- $n$  המבוקש.

## עץ חיפוש בינאריים (Binary Search Tree)

אנחנו נדבר על rooted binary tree – צומת אחד נח呼 לשורש העץ, ולבן צומת **יש עד שני בניים**. הגדרנו עץ זה באופן רקורסיבי. שמן לב לשני מושגים בהקשר לצומת מסוים:



depth 0  
depth 1  
depth 2  
depth 3

- עומק – מספר הקשתות ממנה עד לשורש.
- גובה – המספר המקסימלי של קשתות ממנה לאחד מצאצאי.
- גובה/עומק של עץ זה העומק של העלה הכى עמוק, כלומר הגובה של השורש.

עבור עץ בעל  $h$  צמתים ועומק  $d$  מתקיים  $1 - 2^{d+1} \leq n \leq 1 + d$ . הצד השמאלי מתקבל עבור שרשרת פשוטה, והצד ימני מתקיים עבור עץ מלא – לכל צומת יש שני בניים. אם הולכים לייצר עצים, שבם בכל צומת יהיה איבר אחד. לכן **במota מידע שאפשר לשמור בעץ זהה תהיה ת**.

**עץ חיפוש בינאריים** – בכל node נאחסן מידע מסוים, עם key ועם value. המפתחות מסודרים בעץ כך שבכל צומת, כל המפתחות **בתת העץ השמאלי** קטנים יותר מהפתחה הנוכחי, וכל המפתחות **בתת העץ ימני גדולים** יותר מהפתחה הנוכחי. דבר זה מאפשר לנו לחפש ולהכניס בהתאם על המין הפנימי שיש בעץ. **יש מקום ספציפי לכל מפתח**.

- **עצים בהם לכל צומת יש  $\geq 2$  בניים (left, right – אויל None)**

בנוספָה:

- כל צומת מחזיק גם **פתח יחודי (key)**, וערך (value)
- כל צומת  $v$  מקיים את האינוריאנטה הבאה:  
 $v.left$  גדול ממש מכל המפתחות בתת העץ שורשו  
וגם  
 $v.right$  קטן ממש מכל המפתחות בתת העץ שורשו

כלומר:  $keys(v.left) < v.key < keys(v.right)$

הערה:

- **סדר ההכנסה של המפתחות משפיע על מבנה העץ**. למשל, אם נכניס ערכים מהקטן לגדול, נקבל שרוך ימין (אין אופציה אחרת, לפי הגדירה של עץ חיפוש ביןארי).
- **שתי רשימות שונות יכולות לייצר את אותו העץ**, על אף שסדר האיברים ברשימה בראשיות שונה (לא תמיד).

**סיבוכיות חיפוש** – הסיבוכיות במקרה הגרוע ביותר של חיפוש, תלוי באיך שנראה העץ. אם העץ מאוזן ומלא, עומק העץ הוא לוגריתמי ולכן זה יהיה  $O(\log n)$ . אם קיבלנו שרוך והאיבר יהיה בקצתו של השורץ יהיה  $O(n)$ .

	best case	worst case for any tree	worst case for balanced trees
insert	$O(1)$	$O(n)$	$O(\log n)$
lookup	$O(1)$	$O(n)$	$O(\log n)$

**חיפוש מינימום – מתי he-best case? לא כשהעץ נורא קטן! אלא במקרה שלשורש אין תת-עץ שמאלי.** זה אומר בהגדרה שהוא הכי קטן ולכן נמצא אותו ב-(1). אם הוא שרוך שמאלה ואז יקח לנו  $O(n)$ .

**עומק העץ** – כדי לחשב את עומק העץ בבעלות רקורסיבית, וביצע תמיד קריאה פעמיים לפונקציה הרקורסיבית – רקורסיה כפולה (עבור תת-עץ הימני ועבור תת-עץ השמאלי), שכן בעצם אנחנו עוברים על כל הצמתים בעץ והסיבוכיות היא  $O(n)$ .

**תרגום 10:**

ראינו את המחלקה `BinarySearchTree` ואת המימוש הרקורסיבי `lookup`, שבו אנו מטילים על העץ באופן רקורסיבי וניגשים לתת העץ הימני או השמאלי בהתאם לערך שאנו מחפשים. ב-`insert` הקונספט דומה מאוד רק שצורך להכניס ערך.

– נרצה למשר פונקציה שמקבלת עץ בינארי, ומדפיסה את המפתחות בסדר עולה. ברקורסיה:

- תנאי עצירה: אם `Node = None` אל תעשה כלום.
- צעד רקורסיבי:
  - נדפיס רקורסיבית את תת העץ השמאלי.
  - נדפיס את `node.key` הנוכחי.
  - נדפיס רקורסיבית את תת העץ הימני.

**מבחר 2019:**

בהתנן שתי רשימות שונות, נרצה לדעת האם הן מייצרות את אותו העץ? ברקורסיה:

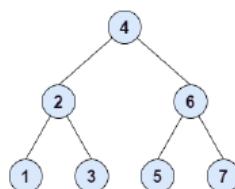
- תנאי עצירה:
  - אם הרשימות ריקות הן מייצרות את אותו העץ – `True`
  - אם האיבר הראשון ברשימה שונה, קיבל שורש שונה – `False`
  - אם אורכי הרשימות שונים – `False`
- צעד רקורסיבי:
  - נוצר 4 תתי רשימות שמייצגות את תת� העצים, נקרא רקורסיבית פעמים על שני זוגות תת� העצים.
  - נבדוק האם תתי הרשימות שקטנות מהשורש יוצרות אותו עץ, וגם תתי הרשימה שגדלות מהשורש.

בנייה עץ מאוזן:

עץ מאוזן הוא עץ חיפוש בינארי מלא – **לכל צומת יש שני בנים פרט לעליים**. השורש הוא החזון של המפתחות  $[1 - 2^n, 1]$ .

- תחילת נשים את `point` והוא יהיה השורש שלו – כאשר  $mid = \frac{first+last}{2}$ .
- נעשה קרייה רקורסיבית על תת העץ הימני, על הטווח  $[mid + 1, last]$ .
- נעשה קרייה רקורסיבית על תת העץ השמאלי, על הטווח  $[first, mid - 1]$ .

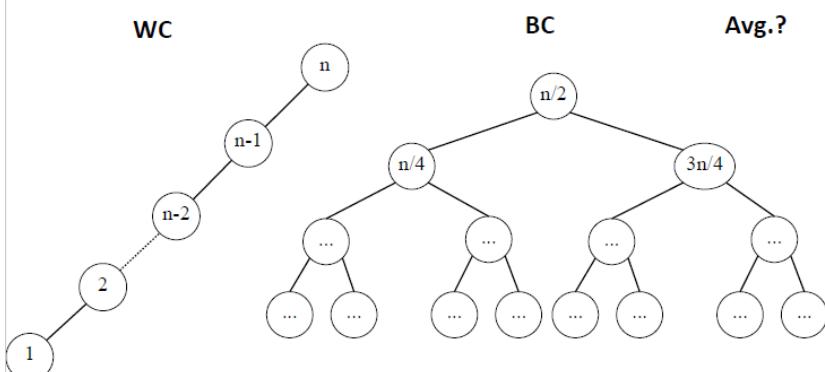
נשים לב – **לכל  $n$  קיים עץ יחיד** זה. למשל, עבור  $3 = n$ :



:Takeaways

עומק עץ חיפוש בינארי בעל  $n$  איברים ולמה זה מעניין

- הכנסה וחיפוש לוגיים ( $O(h)$ ) כאשר  $h$  הוא גובה העץ.
- כאשר עץ שמכיל  $n$  צמתים הוא מאוזן, אז  $O(\log n) = O(h)$ .
- הרבה פעולות במחלקה זו ממומשות באמצעות רקורסיה.



## Hash Tables

המטרה – לבנות מבנה נתונים שיחזק למשל סטודנטים. נרצה לשמר אותם על ידי מפתח מסויים (למשל ת.ז. שהיא עקבות), כך שמבנה הנתונים יאפשר לחפש/להכנס/למחוק **בזמן O(1)**. בכל המבנים שראינו עד כה לא הגיעו לסיבוכיות כזו בזמן ממוצע. טבלאות hash הן פתרון מתאים במקרה זה.

### פונקציות hash:

המשמעות של hash שRELONVENTIAL במקרה זה היא confuse/muddle, ובעברית פונקציית hash נקראת "פונקציית גיבוב". הקונה למפות מרחב מאוד גדול (אפילו אינסופי) למרחב הרבה יותר קטן (בטווח של 999-0):

- הפונקציה היא **דטרמיניסטיבית**, בכל ריצה שלה על קלט x נקבל את אותו הפלט (x).hash.
- עבור 12 ו-13 מהם קלייטים קרובים, הפליטים יכולים להיות מאוד רחוקים – נראה שמתבצע **פיזור אחד**.
- הפונקציה הזאת **לא חח"ע**, כי התחום שלו גדול (בהרבה) מהטווח.

### הערות:

- לפיתון יש פונקציה מובנת בשם **hash**.
- כל פעם שמתחלים את IDLE, הוא מייצר ערך seed רנדומלי, שימושים בו בחישוב של hash וכן נקבל ערכים שונים בritchוטות שונות של התוכנית (**הגנה מפני התקפה של המערכת**).

### טבלאות hash:

נכיה שיש לנו עולם גדול של אובייקטים – נסמן אותו ב- $n$ . אנו רוצים לאgor במערכת תות-קבוצה של המרחב הזה מגודל  $m$ , כאשר  $m$  קטן בהרבה מהגודל של  $n$ . נשמר את הערכים **במבנה T** שנקרא **hash table** שהגודל שלו יהיה  $m$  **שהוא בסדר גדול של מספר האיברים שאנו רוצים לשמר** –  $T$ . כל אחד מהאיברים בטבלה יחזק את האיבר.

איך נמפה את האיברים לתאים? עם **פונקציית hash**:  $\{1 - m, \dots, n\} \rightarrow n : h$ . הפונקציה מהירה מאוד לחישוב וכן ( $O(1)$ ). כדי למצוא איבר נחשב שוב את-hash (דטרמיניסטי לנוכח לוגיון המיקום).

איך נפתר מקרה של התנגשות שבו ערכים ממופים באותו תא בטבלה? אי אפשר למנוע collision לגמרי (בגלל עקרון שובר היוניים תמיד תהיה לנו התנגשות). הפונקציה הזאת בהגדירה היא לא חח"ע. כדי **למזרע את הסיכוי ל-collision** אפשר **להגדיל את הטבלה T** (הסיכוי ששננים יפלו באותה התא קטן) או **"לשפר" את הפונקציה**  $h$  שלנו (שהפיזור יהיה כמו שיוצר אחד, ואז זה מצמצם את ההסתברות להתנגשות).

### פתרונות Collisions באמצעות Chaining:

אם מגיעים כמה ערכים לאותו התא בטבלה, נתmodoּד עם זה בכר **שכל תא נחזיק רשיימה, וככניס את הערכים לרשיימה בתא**. והוסףנו רשיימה (ນצטרך לחפש בה, להוסיף אליה, למחוק ממנה), וב-worst case כל האיברים מתחמפים באותו תא וכן מוצאים זירה לרשיימה אחת ארוכה... יכול להיות שהנתונים לא נחמדים לפונקציית-hash. אנחנו לא יודעים יותר מידיע על הנתונים שכנים וקשה **למנוע מצב של worst case**.

**סיבוכיות** – ב-worst case אנו עוברים על אותו chain, ועוד קיבל ( $O(n)$ ). אבל בהינתן פיזור אחד הסיכוי ל蹶ה זה מאוד קטן. אם נניח ש- $h$  אכן מבצעת פיזור אחד, נמדד סיבוכיות לפי האורך הממוצע של chain.  $\frac{n}{m}$  יקרא **ה-load factor**. אם נבחר  $m$  כר שיטקיים  $(m) = n$  אז  $O(1) = \alpha$  והכל יתבצע **בסיבוכיות (1)** **בזמן O(1)**.

נמשח מחלוקת HashTable משל עצמנו:

- כאשר אנו מבצעים על מחלוקת שיצרנו בעצמנו כמו hash, ניאלץ למשם את Student, נציג שתהגיד לפיתון מה לעשות אם ביקשנו לעשות hash על אובייקט זה, אחרת זה יבצע hash על כתובות בזיכרון, ועל עצמים זרים יתקבלו ערכים שונים!
- נצטרך גם את eq כדי להשוות בין אלמנטים שהטיפוס Student.

**Open Addressing** – בכל תא בטבלה יש **לכל היותר ערך אחד**. לכן בהגדירה  $m$  לא יכול להיות גדול מ- $m$ , ואז ה-load factor שלנו קטן מ-1.

**תרגום 10:**

מה שראינו עד כה:

implementation	insert	search
Python list	$O(1)$ always at the end	$O(n)$
Python ordered list	$O(n)$	$O(\log n)$
Linked list	$O(1)$ always at the start	$O(n)$
Sorted linked list	$O(n)$	$O(n)$
Unbalanced Binary Search Tree	$O(n)$	$O(n)$
Balanced Binary Search Tree	$O(\log n)$	$O(\log n)$

- $m$  גודל הטללה
- $n$  מספר האיברים שנשמרו
- $k$  גודל מפתח (לעתים  $O(1)$ )  $O(k = \alpha \cdot m)$  פקטור העומס = המספר הממוצע של איברים בתא
- נניח ש  $h$  פועלת בסיבוכיות לינארית בוגודל הקולט שלו

נרצה לממש מאגר ת"ז של סטודנטים בת"א. רשיימה המכילה אפסים, בגודל של כל עולם האפשרויות ( $10^9$ ): אופטימלי בחיפוש והכנסה, אבל בזיכרון זיכירון ותחול יקר מאד. לעומת זאת BST חיפוש והכנסה יהיו ב- $O(\log n)$ : אופטימלי באתחול וזכירון, אבל חיפוש והכנסה "סבירים". נרצה להציג  $O(1)$  ממוצע, ולחסוך בזכירון ובאתחול ולהזדוף  $O(n)$ .

**חיפוש:**

מחפשים את המפתח  $q$  בטבלה באמצעות  $hash(q)$ . אם יש רשיימה של דברים, עוברים עליהם איבר-איבר.

- **סיבוכיות ממוצעת:** הפעלת  $h$  על מפתח בגודל  $k$  זה  $O(k)$ . בממוצע יהיו  $\alpha$  איברים בתא, אציר לעבר על כולם וכל השוואה בו היא  $O(ak)$  או שה"ב מדובר ב- $O(a + 1)$ . נקבל
- אם  $(1) O = k$  כמו במקרה של ת"ז, אז הסיבוכיות הממוצעת היא  $(a + 1)O$ .
- אם ב��וף  $cn = m$  עברו קבוע כלשהו, אז הסיבוכיות הממוצעת היא  $(1)O$ .
- **סיבוכיות במקורה הגרוע:** א' בבר לא משחק תפקיד. במקרה הזה כל האיברים הללו לאותו תא (ביש מזל). ואז כל  $t$  האיברים באותו תא ונקבל  $O(nk)$ .
- אם  $(1) O = k$  כמו במקרה של ת"ז, אז הסיבוכיות היא  $(n)O$ .

**Repeating Substring**

יש לנו מחרוזת  $st$  ואורך  $|l|$ . נרצה לבדוק האם  $st$  יש תת מחרוזות רצופה באורך  $|l|$ , שמויפה יותר מפעם אחת?

- **פתרון נאיבי** – נשווה כל תת מחרוזת באורך  $|l|$  לכל תת מחרוזות באורך  $|l|$  שמופיעות אחרת. אנו מבצעים  $l - n$  השוואות באיטרציה הראשונה. **כל השוואה עולה לנו באורך  $|l|$  (אורך תת המחרוזת).** באיטרציה השנייה נבצע  $l - l - n$  השוואות שועלות  $|l|$ . שה"ב נקבל  $(l - n)(l)O = O(l^2)$ .
- **פתרון עם hash** – נ עבור באותה הצורה, **וכל פעם שתת מחרוזת לא נמצאת בטבלה נסיף אותה.** בפעם הבאה שנגיע לאותה המחרוזת נבדוק בטבלה אם היא כבר קיימת (זה מוכיח ממאיציה). נבחר  $l + 1 - n = m$ , כמות המחרוזות שנרצה להכניס.

```
def repeat_hash1(st, l):
    m=len(st)-l+1
    htable = Hashtable(m)
    for i in range(len(st)-l+1):
        if htable.find(st[i:i+l]) == False:
            htable.insert(st[i:i+l])
        else:
            return True
    return False
```

- יש כרגע לפחות אחת, שעוברת על  $|l| - n$  מחרוזות וכן יש  $|l| - n$  איטרציות.
- **כל חיפוש והכנסה בטבלה עולה  $(l)O$ :** סלייסינג, פונקציית האש, השוואה **ב-chain הרלוונטי**.
- **בחרנו את  $h - a$  שלנו כמו שצרכו:**  $\frac{n-l+1}{m} = a$  וכן יש בממוצע מספר קבוע של השוואות (התנגשויות). אך בזיה כפול  $(1)O$ .
- סה"ב נקבל  $(n^2)O(l(n - l)) \approx O(l(n^2))$ .

## :Takeaways

- שימושות בהרבה אלגוריתמים, כולל במקומית.
- הכנסה וחיפוש המ(1) 0 בזמן O(1) בזמן WCT (כאשר  $W$  הוא מספר האלמנטים בטבלה).
- חשוב להבין את ניתוחי הסיבוכיות של hash tables.

כאשר בשאלת מצינים (a) 0 בזמן O(1) במשמעות hash table או set (במציאות משתמשים יותר ב-set).

## Generators

דיברנו על streams – אוסףים אינסופיים של מידע שזורם אליו. אלגוריתמים שמתפלים בשטפי מידע באלו נקראים online, זאת להבדיל מכל האלגוריתמים שראינו עד כה שקיבלו מידע סופי והחזירו מידע offline. כאמור, לא ניתן לשמר stream זה בזיכרון. כדי לטפל וליצג stream נועד generator.

פונקציית generator מכילה את המילה השמורה yield.

- כאשר נקרא לפונקציה () naturals מה שיוחזר הוא אובייקט מסוג generator.
- בעת שניתן לקרוא לפונקציה next(nat) כלומר על הגנרטור שיש בידינו, ואז מקבל את הערך הבא שהפונקציה עשויה לו yield.
- לאובייקט זה יש state שהוא נשמר בין קריאות (אחרי שקראננו לה פעמי אחת, אחרי ה-**yield** היא "קופאת"), ואז בשנקרא ל-next אבן מקבל את הערך הבא (הפונקציה חוזרת לרווח מאותו המקום עד ה-**yield** הבא).
- כל אובייקט generator עם state משלה מחייב בזיכרון לפעול להמשך הריצה (קריאה נוספת ל-next תמשיך את הריצה מאותו המקום, עד ל-**yield** הבא).
- אין בשום מקום בזיכרון את כל הטעבים – האובייקט יודע לייצר כל פעם את המספר הבא אבל הוא לא שומר אותו בזיכרון.

```
>>> nat = naturals()
>>> next(nat)
0
>>> next(nat)
1
>>> [next(nat) for i in range(10)]
[2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
```

```
def naturals():
    n = 0
    while True:
        yield n
        n+=1
```

- We see that nat has a state, which is retained, unchanged, between successive calls to next.
- We can have additional instances of the generator:

```
>>> nat2 = naturals()
>>> next(nat2)
0
>>> next(nat)
12
```

5

ראינו עוד דוגמאות: מספר פיבונacci, ממוצע זו של טמפרטורות.

מיזוג שני גנרטורים: עברו שני גנרטורים שונים, יצרכו גנרטור חדש (stream) שמחזיר את המיזוג הממויין של שניים. קידמנו את שני הגנרטורים תחילת עם next, ואז בדקנו בולאה מה הערך הקטן יותר, ואת הגנרטור שלו קידמנו עם next.

גנרטורים סופיים באים להחליף רשימות גדולות בזיכרון. אפשר ב-list comprehension () במקומות [] ואז במקומות לקבל את כל הרשימה, נקבל generator שמייצר את הערכים ברשימה.

## מגבילות על גנרטורים אינסופיים:

- לא ניתן ליצור גנרטור אחר שמחזיר את אותם האיברים בסדר הפוך (אי אפשר ללבת עד סוף הרשימה אם היא אינסופית).
- לא ניתן ליצור גנרטור אחר שמחזיר את כל האיברים שמוופיעים יותר מפעם אחת בGENERATOR המוקור.

איטטור (Iterator) – מאחריו הקלעים מאוד דומה לגנרטור. איטטור מספק גישה לאוסף נתון של איברים, שכןן לבקש אותם אחד.

**תרגול 11:**גנרטור סופי מול אינסופי:

- באשר גנרטור מגיע לסוף הריצה שלו או ייתר פקודות `yield` הוא מחייב שגיאת של `StopIteration`. זה קורה אר ורך עבור גנרטור סופי.
- בגנרטור אינסופי אנחנו נראה `True` while (למרות שלא בורא נרצה לראות בזה מימוש של לולא אינסופית), כי אנחנו רצים תהליך אינסופי. אמנם, אין בעיה כי כל פעם יוחזר ערך אחד, הסטטוס "МОוקפא".

**Primes Generator**

המשימה היא ליצר `stream` של כל המספרים הראשוניים.

רעיון: 2 הוא ראשוני. לאחר מכן נתחילה-3 ולכל מספר נבדוק אם הוא לא מתחלק בכל מספר ראשוני שמצאנו קודם לכן. ככלומר, כל פעם שאנו מוצאים ראשוני אחד מתחסנים אותו בראשימה. בכל פעם שאנו מחפשים ראשוני חדש, אנו עוברים על כל הראשיונים שמצאנו קודם, ויש לנו **תלות בקריאה הקודמת**.

אנו מעוניינים לממש גנרטור, זה שלא שומר את כל התוצאות של החישובים עד בה. לכן, נбурר מספר מספר וונעזר בפונקציה `prime_is` (לפי מבחן פרמה, זו בדיקה שלא תלויה במספרים שמצאו עד בה).

**שאלה מבחון:**

נרצה ליצר את כל הזוגות הסדריים של מספרים טבעיים, ללא חשבות לסדר, ולא חוזרת.

- בקוד שלעיל יש בעיה.
  - לא - קריאה `next()` בסה"כ ממחירה גנרטור.
  - לא – קריאה `next` לא תיכנס לולא אינסופית, זה רק מחייב ערך ועוצר.
  - כן – ה-`i` מאותחל ל-0, `j` מאותחל ל-0, וכל פעם `j` מוקדם, אבל **לא מוקדם!** ולכן זוג כמו (2,3) לא מיוצר אף פעם.
  - לא – לא מייצרים שום דבר בכלל.
  - לא – קריאה `next` אף פעם לא גורמת לולא אינסופית, תמיד אנו עושים `yield` ומחכים.
- ב. זוגות בהם  $i < j$ :

```
def SomePairs():
    i=0
    while True:
```

```
        for j in range(i):
            yield (i, j)
        i = i + 1
```

ג. מקבלת גנרטור ומיצרת את הזוגות הפוכים:

PairsGen is a generator function

```
def RevGen(PairsGen):
```

pgen is a generator!

```
    pgen = PairsGen()
    while True:
        pair = next(pgen)
        yield (pair[1], pair[0])
```

ד. איחוד של שני גנרטורים אינסופיים:

```
def UnionGenerators(gen1, gen2):
```

```
    while True:
        yield (next(gen1))
        yield (next(gen2))
```

```
gen =
UnionGenerators
(g1_2, g3)
```

```
{ g1_2 =
UnionGenerators(g1, g2) { g1 = EqPairs()
g2 = RevGen(SomePairs)
g3 = SomePairs()
```

```
>>> gen = AllPairs()
>>> for k in range(7):
print(next(gen))
(0,0)
(1,0)
(0,1)
(2,0)
(1,1)
(2,1)
(0,2)
```

פעם ראשונה שאנו עושים `next(gen)` הוא קודם הולך בראשון באיחוד שזה `g1_2` שהוא גם איחוד של גנרטורים אחד קודם כל שיש איבר מ-`g1` שזה מהאלבסון וכן (0,0).

הגנרטור הראשון `g1` הולך בעצםו ל-`g3` ומהזיר (1,0).

בפעם השלישייה שוב הולכים לגנרטור `g1_2` אבל בעצםו הוא הולך לגנרטור השני שלו, שהוא `g2` ואז קיבל את (0,1).

כיוון שה-`SomePairs` מאחד רק 2 גנרטורים, הינו צריכים לעשות איחוד על איחוד. אם `SomePairs` היה מאחד 3 גנרטורים הינו קוראים לו עם כל 3 הגנרטורים שהינו רוצים לאחד.

בונוס: ניתן להציג את כל הזוגות במכה אחת, על ידי מעבר על אלכסונים סופיים כ אלה. במקום ללכט על שורות, נלק על אלכסונים שנעיצרים בעמודה השמאלית.

i \ j	0	1	2	3	4	5	...
0	/						
1	/	/					
2	/		/				
3			/				
4				/			
5					/		
...							

```
def AllPairs_v2():
    sum = 0
    while True:
        for i in range(sum + 1):
            yield i, sum - i
        sum += 1
```

### :Takeaways

- פונקציית גנרטור היא פונקציה שמקילה את הפוקודה `yield` ומחזירה אובייקט גנרטור.
- גנרטורים יכולים לבצע איטרציה על סדרות אינסופיות, על ידי חישוב כל אלמנט באופן איטרטיבי, באמצעות `next`.
- כל פעם ש-`next` רץ על גנרטור, הוא מורייך את הקוד מהנקודה שבה הוא עצר קודם לכן, עד פקודת `yield`-ה-`state`.
- ה-`state` של הפונקציה נשמר בזיכרון כדי שזה יהיה אפשרי.
- לשמור מצבים `next` על גנרטור שישים לו, נקבל שגיאת `StopIteration`.
- בשימושם בגנרטור בוליאני – `for x in gen`, זה משתמש ב-`(next(gen))` עד שהגנרטור מסתיים.

## 6 – נושאים נוספים

### H - טקסט

#### אלגוריתם CYK

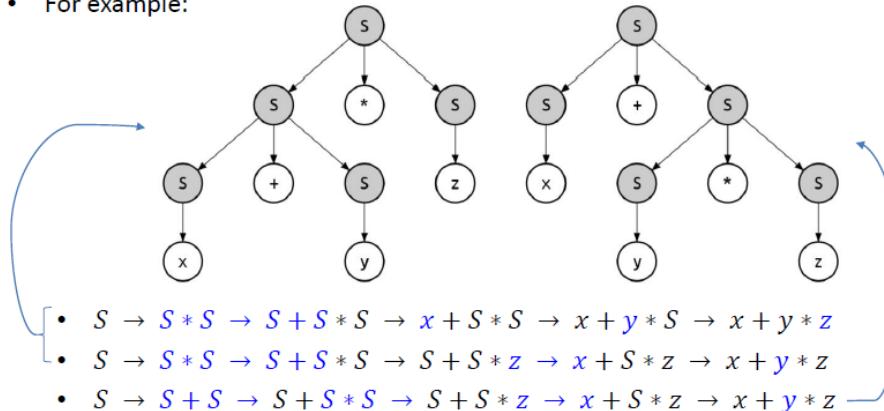
טוב אך עצי גזירה זה מוכר וכיפי (ת לחבר מתחילה/מתקדמים). נשים לב שבדקוק יש 4 סטם של איברים:

- $V$  – משתנים ( $S, NP$ )
- $\Sigma$  – טרמינלים/אלפבית ("א", "נ", "ו")
- $R$  – חוקי גזירה.
- $S$  – משתנה שמננו מתחילה.

**דקוק חסר הקשר (CFG):** בצד שמאל של החוק יש אך ורק **Variable** ייחיד. בלומר, ניתן להפעיל את החוק ללא חשיבות להקשר של המשתנה, מה נמצא בסביבה שלו, תמיד ניתן להחליף אותו על ידי הצד הימני של החוק (ה-output).

אם מחרוזת (מילה)  $w$  יכולה להיגדר על ידי הדקוק  $G$  אז נאמר שהמחרוזת שיבת לשפה שהדקוק מגדיר, ( $w \in L(G)$ ).  
- יכולה להיווצר דו-משמעות. אותה המחרוזת יכולה להיווצר על ידי עצי גזירה שונים. **Ambiguity**

- For example:



בניגוד למספר בעיות שלא ניתן לפתור (disjointness, inclusion, equality, ambiguity), הבעיה של בדיקה האם מחרוזת שיבת לשפה לפעמים היא כן פתרה. ניבור למבנה ספציפי של CFG.

**(Chomsky Normal Form) CNF:** בצורה זו, כל חוק הוא מהצורות הבאות:

- $A \rightarrow a$ : כל דירוייצה מושיפה (לכל היותר) טרמיניל אחד. כאשר  $\Sigma \in V$  ו-  $a \in \Sigma$
- $A \rightarrow BC$ : כאשר  $\{S\} \setminus V \in A$  ו-  $B, C \in V$
- $S \rightarrow \epsilon$ : מופיע רק פעם אחת, ו-  $\epsilon$  מופיע רק במילה הריקה.

**הערות:**

- מצד ימין יכול להופיע או טרמיניל אחד, או שני משתנים. **לכל צומת פנימי בעץ יש שני בניים, אלא אם כן הוא אבא של עלה.**
- מכל CFG ניתן לייצר דקוק מקביל-BNF.
- חסכנות – נדרש יותר משתנים וחוקים, ויכול להיות יותר מורכב ופחות אינטואיטיבי.
- תרונות – פשוט, מאפשר חישובים מתמטיים ואלגוריתמים של parsing כמו CYK.

בהתאם דקוק  $G$  ומחרוזת  $st$ :

- **בעיית הזיהוי** (בדיקה שיבות, Recognition) – האם  $st$  יכולה להתקבל על ידי  $G$ , האם ( $L(G)$ )
  - **בעיית הגזירה** (Parsing) – אם  $st$  מתקיים על ידי  $G$ , מציאת עץ גזירה (בלשונו) שמייצר את  $st$ .
- הבעיות אלו פתירות בזמן פולינומיי ב-CNF.

## אלגוריתם CYK

האלגוריתם המקורי (נלמד השנה שעברית בקורס) קצר מורכב יותר. הוא פועל בשיטה של "תכנות דינמי", הוא עובד בצורה up-bottom (וקורסיה מתחילה מלמעלה, ובסוף מתפרקת שוב מלמטה). אנחנו נלמד את הגישה הרקורסיבית שפועלות top-down והיא פחותות עילית. בתרגול נוסף ממוואיציה ונקבל את הסיבוכיות הפולינומיאלית.

הקלט שלנו הוא דקדוק  $G$  ומחרוזת  $st$ , כאשר  $t$  הוא אורך המחרוזת. הפלט שלנו הוא האם  $.st \in L(G)$

הרעישן: מחרוזת שיבת לדקדוק אם אפשר להתחיל מ- $S$ , להפעיל כל-לי גזירה ולקבל את  $st$ . ב-CNF יש חוקים מאד מסויימים, עבור  $st$  באורך  $k$ -1 מתקיים שניתן לגזר את  $st$   $\Leftrightarrow$  קיימת חלוקה של  $st$  לשני חלקים: הראשון באורך  $k$  והשני באורך  $t-k$ . החלק הראשון הוא  $[k:st]$  והשני הוא  $[st:k]$  כאשר קיימים חוק  $X \rightarrow S$  ויש דרך לגזר מ- $X$  את  $[k:st]$  ו- $[st:k]$ .

הסתכלנו על המילה "baaba" באורך 5. יש 4 אפשרויות לחלק את המילה לשני אורכים שונים, ובאופן כללי עבור  $t-n$  אפשרויות חלוקה. לכל אפשרות חלוקה יש לנו שני חוקים שאפשר להפעיל:  $AB \rightarrow BC \rightarrow \dots \rightarrow S$ . לכל חלק יש 2 קריאות רקורסיביות, עבור  $B$  ועבור  $C$ . סה"כ לכל אפשרויות חלוקה יש 4 כלים שימושיים. בולם בrama הראשונה של העץ יהיו  $(1-n)$  רמות.

### הקוד:

start\_var: משתנה ההתחלת.

- rule\_dict: את החוקים נשמר במלון, המפתח יהיה הצד השמאלי של החוק, הערך יהיה set של הצד ימני של החוק: כל איבר ב-set זהה הוא מחרוזת באורך 2 של תווים-uppercase, UPPER CASE, או מחרוזת באורך 1 ב-ascii\_lowercase שהוא טרמינל.
- נשים לב כי את הטרמינלים  $S$  ואת המשתנים  $V$  אנחנו לא מקבלים בפונקציה (הם מובלעים בתוך החוקים).
- מימוש פונקציית מעטפת שקופה לפונקציה רקורסיבית עם האינדקסים שלlicing שאמו רצים לבצע, בולם מعتبرים תחילת את 0 ואת  $\text{len}(st)$ .

```
if st == "": # In CNF, only the start_var can derive the empty string
    return "" in rule_dict[start_var]
return cyk_rec(rule_dict, start_var, st, 0, len(st))
```

.var  $\rightarrow st[i:j]$  :var: עבורתו בודד, האם יש כלל כלשהו שצל ימין שלו הוא אותו הבודד הצד שמאל שלו הוא אותו הבודד.

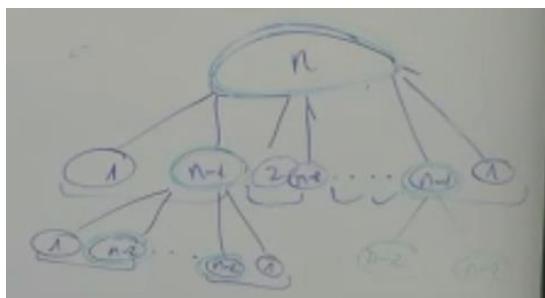
```
if i == j-1: # If st[i:j] is single char
    return st[i] in rule_dict[var] # check if there is a rule var -> st[i]
```

בollowה נבדוק כל חלוקה אפשרית של המחרוזת. עבור כל חלוקה, נבדוק על כל החוקים עם  $var$  בצד שמאל. אם מדובר בחוק שצד ימין שלו משתנים XY (חוק שבצד ימין שלו יש טרמינל לא מעניין אותנו, אנו רוצים להתקדם ברקורסיה), אנו נבדוק האם קיימים חוקים כך ש- $X$  גוזר את  $[k:st]$  ו- $Y$  גוזר את  $[j:k]$ .

```
for k in range(i+1, j): # any non-trivial split of st[i:j] into st[i:k], st[k:j]
    for var_rule in rule_dict[var]: # any rule with var on LHS (var -> ...)
        if len(var_rule) == 2: # rule of the form var -> XY
            X, Y = var_rule[0], var_rule[1]
            # If X derives st[i:k] and Y derives st[k:j] then var derives st[i:j]
            if cyk_rec(rule_dict, X, st, i, k) and \
                cyk_rec(rule_dict, Y, st, k, j):
                return True

# If all rules and splits fail, var cannot derive st[i:j]
return False
```

### ביטחון סיבוכיות:



- מתחלים מהשורש עבור הקלט  $t$ . ממנו יש קריאה רקורסיבית לכל חלוקה אפשרית, יש  $1-n$  (כפול 2, כי מכל חוק יוצאות שתי קריאות רקורסיביות, ואנחנו מניחים ברגע חוק אחד מכל משתנה).
- עבור  $1-n$  יש גם  $2-n$  קריאות כפול 2 (מכל חוק יוצאות שתי קריאות כאמור).
- קיבלו עץ עם  $n$  רמות, ולפחות 2 התפצלויות בכל רמה, לפחות  $2^n$ .
- נקבל  $\dots + (1-n) = 2T(n)$ , וזה מזכיר לנו את האנוי, שהוא אקספוננציאלי-ב- $n$ . הסתכלנו רק על חלק מהצמתים, יש הרבה יותר צמתים ולבן הסיבוכיות פה ברורית.

## תרגול 11

נשפר את זמן היריצה באמצעות ממואיזציה:

- או שימושיים את זה בתור תנאי התחליה – **זה מה שנעשה**.
- או שימושיים בתור בדיקה וגע לפניו שעושים קרייה וקורסיבית.

( $var, j, i$ ) מייצגים לנו קריאה וקורסיבית. בשורה לפניה, את ההשמה של אותו הערך (תוצאת החישוב שאנו רוצים לשומר) למיילן *mem*.

ניתוח סיבוכיות עם ממואיזציה:

$a$  – אורך המחרוזת,  $z$  – מספר חוקי הגזירה.

ניתוח ראשוני – נסכם את כל העבודה על פני השלשות ( $var, j, i$ ).

- בדיקות במילוניים וב-set-ים הם  $O(1)$  בפועל.
- הוללה החיצונית תבצע  $i \leq j$  איטרציות, והוללה הפנימית רצה על כמות החוקים שיצאים מ-*var*.
- מתקיים  $r \leq |rule_{dict[var]}|, n \leq i - j$ , ולכן לכל שלשה הסיבוכיות היא  $O(nr)$ .
- כמות השלשות:  $a$  אפשרויות ל- $i$ -ו- $j$ , אז סה"כ  $a^2$  וככפイル ב- $z$  עבור כל המשתנים (יש לפחות  $z$  חוקים)  $= O(n^3r^2)$ .

אבחנה: אם עבור משתנה אחד יש הרבה מאוד חוקים, ועבור אחר יש מעט חוקים, נקבע לחסום את כל העבודות בו-זאת? אנו יודעים כי  $rule_{dict[var]} = \sum_{var} |rule_{dict[var]}$ , אבל אין סיבה לחסום עבור כל המשתנים את כל העבודות שלו ב- $z$ .

ניתוח שני – נסכם את העבודה על פני כל הזוגות ( $j, i$ ). זה שונה ממה שראינו עד עכשיו. כי הזוג הזה אינו קלט של הפונקציה, זה בעצם קבוצה של קריאות וקורסיביות. יש קבוצה של קריאות וקורסיביות עם -( $j, i$ ) הספציפיים האלה.

עבדיו אנו מכפילים את  $(ar)O$  רק בכמות הזוגות שהוא  $a^2$ , ויצא לנו  $(r^3n)O$ .

## דחיסת האפס

### מבוא לדחיסת טקסט

אנו מדברים באופן כללי על תקשורת בין צדים, שהוצאים להחליף מידע. אתגרים שניתקל בהם בתחום תקשורת:

1. רעש - הקוו/הערץ לא אמין במאה אחד, יש רעש והפרעות (פתרון: **קודים לתיקון שגיאות**).
2. אבטחת מידע – בלי שימושו יכול לצלט ולgelות את תוכן המידע (פתרון: **криיפטוגרפיה**).
3. חסכון – תקשורת זה דבר יקר, ונרצה לצמצם את גודל המידע שאנחנו מעבירים (פתרון: **דחיסה**).

הערה: בעולם האמיתי, כל האתגרים האלה באים ביחד ומשפיעים זה על זה. אם מצפינים משהוו המידע הופך להיות הרבה יותר מפוזר, וזה מאד יקשה על הדחיסה שלו.

نبחוין בין:

- lossless compression – בהנחה שיש מנגנון דחיסה C ומנגנון פענוח D, שתיהן פונקציות ממחרוזת בינהן למחזרות ביןארית. נדרש  $len(x) < len(C(x))$  אחרת לא דחסנו כלום. כדי שנוכל לשחזר את המידע, נדרש שגם  $C(x) = y$   $x = C(y)$ . קידוד מסゴ' בזה, ונרצה בכל מה שקשרו לטקסט.
- lossy compression – אמונם, בכל מה שקשרו למדייה (אודיו, תמונה, וידאו) נרצה לעצמנו אובדן של מידע.
- universal lossless compression – אם תמיד  $len(x) < len(C(x))$  לכל מחזרות אפשרית. בכך נטען: **אי אפשר ליצור קידוד שהוא אוניברסלי**. لكن, בקידוד אנו נדחוס דברים רלוונטיים, ונשלם מחיר שעבור חלק מהדברים אנחנו לא נדחוס ואפיו אולי ננפח (כי לא התאמנו את הקידוד אליהם). זה בעצם אומר, **שיהיה WCT** שעבור הדחיסה לא תעבור וננפח את המידע.

תוכנות:

.prefix-free ,variable-length .prefix-free ,variable-length晦ות של ביטים. הוא מקיים שתי תכונות מאוד מרכזיות –

**fixed vs. variable length**: אנו רוצים לקודד כל תו למחרוזת בינארית, וכך נקרא **fixed-length code**. התווים ממופים למחרוזות בינאריות באורך (בmmo ASCII). לעומת זאת, **variable-length** ממפה לארכימטרים שונים (Unicode). כדי שהקידוד יהיה יעיל, אותיות נפוצות יקודדו למחרוזות קצרה, אותיות נדירות יקודדו למחרוזות ארוכות. הקידוד יבסס על מערכ שabicities של אותיות (מתוך קורפוס כלשהו), שאנו מוקים שמייצג את השפה.

**prefix-free**: אין שתי אותיות בשפה, שמתכוודות לשני קודים כך שאחד מהקודים הוא ה-**prefix** של הקוד השני. אין שני קודים בקידוד זהה, כך שהקצר יותר הוא התחילו של השני.

### Prefix Free Codes and Ambiguity

Code 1:	<table border="1" style="width: 100%; border-collapse: collapse;"> <tr> <td style="padding: 2px;">a</td><td style="padding: 2px;">b</td><td style="padding: 2px;">c</td><td style="padding: 2px;">d</td><td style="padding: 2px;">e</td><td style="padding: 2px;">f</td></tr> <tr> <td style="padding: 2px;">000</td><td style="padding: 2px;">001</td><td style="padding: 2px;">010</td><td style="padding: 2px;">011</td><td style="padding: 2px;">100</td><td style="padding: 2px;">101</td></tr> </table>	a	b	c	d	e	f	000	001	010	011	100	101
a	b	c	d	e	f								
000	001	010	011	100	101								
Code 2:	<table border="1" style="width: 100%; border-collapse: collapse;"> <tr> <td style="padding: 2px;">a</td><td style="padding: 2px;">b</td><td style="padding: 2px;">c</td><td style="padding: 2px;">d</td><td style="padding: 2px;">e</td><td style="padding: 2px;">f</td></tr> <tr> <td style="padding: 2px;">0</td><td style="padding: 2px;">10</td><td style="padding: 2px;">110</td><td style="padding: 2px;">1110</td><td style="padding: 2px;">11110</td><td style="padding: 2px;">111110</td></tr> </table>	a	b	c	d	e	f	0	10	110	1110	11110	111110
a	b	c	d	e	f								
0	10	110	1110	11110	111110								
Code 3:	<table border="1" style="width: 100%; border-collapse: collapse;"> <tr> <td style="padding: 2px;">a</td><td style="padding: 2px;">b</td><td style="padding: 2px;">c</td><td style="padding: 2px;">d</td><td style="padding: 2px;">e</td><td style="padding: 2px;">f</td></tr> <tr> <td style="padding: 2px;">0</td><td style="padding: 2px;">1</td><td style="padding: 2px;">00</td><td style="padding: 2px;">01</td><td style="padding: 2px;">10</td><td style="padding: 2px;">11</td></tr> </table>	a	b	c	d	e	f	0	1	00	01	10	11
a	b	c	d	e	f								
0	1	00	01	10	11								

Code 1 is a **fixed length code**, hence it is also a **prefix free code**.

Code 2 and 3 are **variable length codes**.

Code 2 is a **prefix free code**: No codeword is a prefix of another.

Code 3 is **not**. For example, 1 is a prefix of 11.

לסייעם: נctrar לדוחם חלק מהדברים, חלק עם קודים קצרים וחלק ארוכים – **לכן הוא variable length**. ואנחנו חיבים שהוא היה **prefix-free** אחרת לא יוכל לפענח את זה בחזרה.

דחסית האפמן:

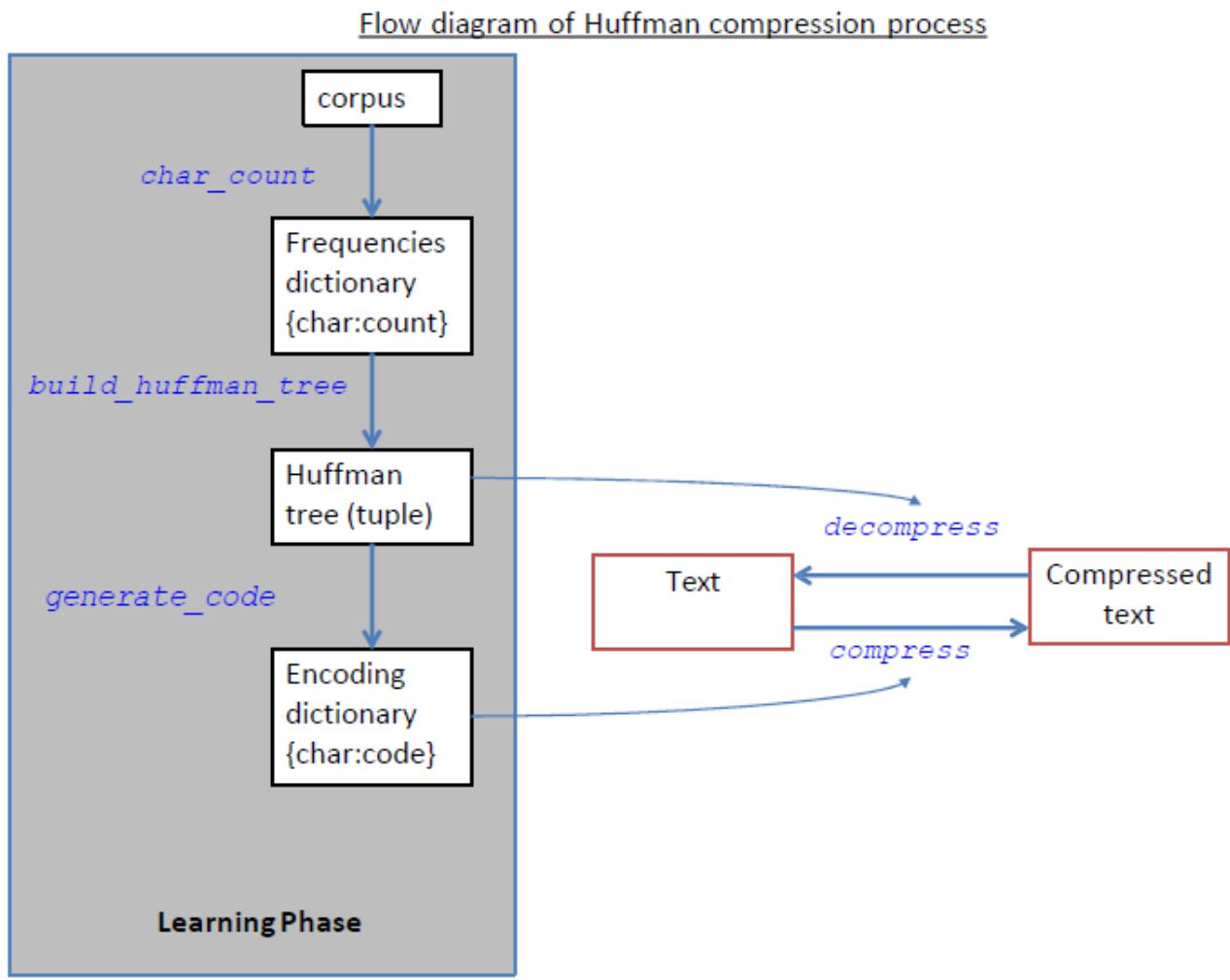
בהתעת קורפוס מסוים, נבנה מילון שכיחויות – כמה פעמים כל אות מופיעה. המטריה של האפמן היא להציג בכל תו למחרוזת ביטים, **שהיא variable-length, prefix-free**.

- נשים את כל האותיות עם השכיחויות בעקבות priority, מבנה נתונים שמאפשר לנו להוציא כל פעם את המינימלי (נשתמש ב-dict ברגע, מיומשים יותר טובים יהיו בקורס אלגוריתמים).
- נוציא את המינימום פעמיים.
- עבשו ניצור צומת אב לשני הבנים שלו, כאשר השכיחות שלו תהיה סכום השכיחויות של הבנים.
- נכניס את הצומת החדש (עם הבנים שלו) בחזרה ל-queue.

בוצע זאת בלולאה עד אשר התור יכול רק אלמנט אחד – והוא העץ המלא שהתקבל. בעוד, כל אות ניצג על ידי המסלול שmobiel אליה, כאשר התפצלות ימינה מסמן 1, ושמאליה מסמן 0. נשרר את כל הספרות שבדרך ונקלט למחרוזת בינארית.

סה"כ קיבלנו:

- בכל שאות הייתה יותר נדירה, הכנסנו אותה יותר מוקדם לעץ, ובוון שהבניה מתחילה מלמטה למעלה **נקבל את האותיות הבינדירות עמוק בעץ, והאותיות השכיחות יהיו גובה בעץ!**
- הוא **prefix-free**.



ניצג את העץ בתור tuples עם 2 איברים בצורה רקורסיבית, כלומר אלו 2 צמתים בעץ שהאיחוד שלהם הוא האבא, וכל אחד מהם יכול להכיל עוד tuples בצורה רקורסיבית.

יש לנו 2 דרגות חופש – כך שניתן לקבל עצים שונים וקידודים שונים:

- אם יש יותר מכמה ערכי מינימום – את מי להוציא?
- לא אמרנו כיצד לסדר אותם.

אופטימליות: זו הדרך הביא יעליה לדחוס את הקורפוס עצמו.

## תרגול 12:

שלב	דוגמה	_bh
1 – חישוב תדירות מקורפוס	<p>קוד (נ) גודל הקורפוס, <math>m</math> גודל הودעה, <math>n</math> כמות הביטים בהודעה המקודדת, כמות התווים באلفבית (<math> S  =  \Sigma </math>)</p> <p><u>:char count</u></p> <ul style="list-style-type: none"> <li>בשמדברים על מיליון, יש הבדל בין סיבוכיות ממוצעת לבין סיבוכיות WC. אנחנו ננתה לרוב סיבוכיות ממוצעת.</li> <li>בכל הכנסה למיילון עלה (<math>O(1)</math>). צריך להכניס למיילון <math>n</math> פעמים וסה"כ נקבל (<math>O(n)</math>).</li> </ul>	<p>corpus = 'aaaabbcdddeee'</p> <p>'a', 4    'b', 2    'c', 1    'd', 3    'e', 3</p>
2 – בניית בעורת התדריות	<p><u>:build huffman tree</u></p> <ul style="list-style-type: none"> <li>הולאה עשוה כמות איטרציות לאורק התווים שיש לנו, אנחנו שארוך זה הוא (<math>O(1)</math>) אך יש לנו (<math>O(1)</math>) איטרציות.</li> <li>חישוש מינימום במילון שהוא גם בגודל התווים הוא (<math>O(1)</math>).</li> <li>כל מה שקרה פה תלוי רק ב-<math>S</math> וכן מה"כ נקבל (<math>O(1)</math>).</li> </ul> <p>יצוג רקורסיבי לעצים:</p> <ul style="list-style-type: none"> <li>עליה מיוצג על ידי מחרוזת של התווים שאותו הוא מקודד.</li> <li>צומת שהוא לא עליה מיוצג על ידי (left, right) (left, right) הם עצים.</li> </ul> <p><math>\{(((\text{'c'}, \text{'b'}), \text{'d'}), (\text{'e'}, \text{'a'})) : 13\}</math></p>	<p>build forest from corpus while  forest  &gt; 1:   extract 2 min trees: t1, t2   union t' = t1 + t2   put t' in forest</p> <p>בכל שלב נמצג שני עצים ונתקבל עץ אחד פחות. כל פעם נסתכל על שני העצים בעל הערכיהם המינימליים. ערך העץ יהיה סכום הערכים של הבנים שלו.</p>
3 – בניית קוד האפמן מתוך העץ	<p><u>:generate_hcode</u></p> <ul style="list-style-type: none"> <li>גודל העץ הוא תלו依 בכמות העלים, והוא (<math>O( \Sigma )</math>).</li> <li>מספר הקראות הרקורסיביות הוא בדיקות כמות הצמתים בעץ. עוברים בכל צומת פעם אחת.</li> <li>נקבל גם כאן (<math>O(1)</math>).</li> </ul> <p>ברקורסיה אנו סוחבים את המשטנה <math>prefix</math>, ומשרשרים אליו את הקידוד עד שmaguiim לעלה, ושם מוסיפים את הערך של העלה (שהוא <math>t</math>, למשל 'c') ואת ה-<math>prefix</math> שיש לנו עד בה.</p>	<p>שמала זה 0, ימינה זה 1.</p> <p>color edges: left = 0, right = 1 <math>H(c) = path(c)</math></p> <ul style="list-style-type: none"> <li><math>H(\text{'c'}) = 000</math></li> <li><math>H(\text{'b'}) = 001</math></li> <li><math>H(\text{'d'}) = 01</math></li> <li><math>H(\text{'e'}) = 10</math></li> <li><math>H(\text{'a'}) = 11</math></li> </ul> <p>אבחנות:</p> <ul style="list-style-type: none"> <li>העלים של העץ הם בהכרח התווים.</li> <li>כל צומת בעץ זהה או שהוא עלה, או שיש לו שני בנים. אין צומת רק עם בן אחד. כל פעם אנחנו לוקחים 2 צמתים ומאתדים אותם, אז זה לא יכול לקרות.</li> <li>הקוד הוא <math>prefix-free</math>. כל מילת קוד היא לא <math>prefix</math> של מילת קוד אחרת. לא יכול להיותתו שהוא המשר שלתו אחר, כי <b>בולם עלים</b>.</li> </ul>
4 – דחיסה	<p><u>:compress</u></p> <ul style="list-style-type: none"> <li>כל <math>t</math> מmirים לקידוד המתאים.</li> <li> אנחנו עושים <math>m</math> איטרציות, על כל <math>t</math> בהודעה המקורית שלנו.</li> <li> קיבלים משחו באורך <math>d</math>. אנחנו טוענים כי (<math>b = \Theta(m)</math>).</li> <li> בזווית-<math>S</math>-סופיות זאת כל קידוד הוא גם באורך סופי.</li> <li> <math>m =  \Sigma b</math></li> <li> <math>O(m)</math> במשמעות.</li> </ul>	
5 - פענוח	<p><u>:decompress</u></p> <ul style="list-style-type: none"> <li> אנחנו ישר משתמשים בעץ, ומטיילים עליו עד שmaguiim לעלה, ואז מתחליםשוב מהשורש.</li> <li> הטויל מתבצע באמצעות גישה שמאלה עם אינדקס 0, ימינה עם אינדקס 1, לפי ה-<math>root</math> הנוכחי שאנו חנכו רצים עליון.</li> <li>זה עולה (<math>O(b)</math>).</li> </ul>	<p>עוברים ספירה-ספרה עד שמקבליםתו שנמצא במילון, בשנקבלתו נדע בוודאות שזה מקודד אותו ולא אףתו אחר (תוכנה של <math>prefix-free</math>).</p>



UD – Uniquely Decodable: כל הזדעה בינהarity זו שנוכל לקודד, יש סדרה אחת בדיקות של תווים שנוכל לקודד כדי לקבל אותה. יש למחזרות זו זו מוקור אחד של תווים. בכיה נוכל לפעננה. כל קוד PF הוא גם UD – כדי שנוכל לפעננה בדיקות בצורה אחת את ההזדעה. (הכיוון השני לא בהכרח מתקיים, לא כל קוד שהוא UD הוא גם PF).

### PF and UD

	PF	Not PF
UD	$a \rightarrow 011$ $b \rightarrow 10$	$a \rightarrow 0$ $b \rightarrow 01$
Not UD		$a \rightarrow 0$ $b \rightarrow 01$ $c \rightarrow 001$

How to decode?

msg = 001

ab? c?

קוד האפמן הוא האופטימלי ביותר!

### שאלה מבחן (2014) – Alternating Tress

- עיצים הם אלטרנטיביים אם הם מייצרים מאותו הקורפוס אבל לקידודים של האותיות יש אורכים שונים. זה נובע מהבחירה שיש לנו בשניים המינימליים מבחינת השכיחויות.
- טענה לא נכונה: "אם בקורס אין שני תווים עם אותה שכיחות, אין עיצים אלטרנטיביים". זה שיש ייחודיות בערכיכם ההתחלתיים לא אומר שבמבחן לא יהיו התנגשויות.

### דחיסת למפל-зи

פותח במקור במאמר משנת 1977. קידוד זה מאפשר לקחת טקסט, לדחוס אותו בדרך כלל, בצורה בו שהצד השני מקבל את הטקסט הדחוס ובליל לשלוח לו שום מפתח או עקרונות של הקידוד, יודע לפרש את זה בצורה אוטומטית.

- הקידוד הוא **אדפטיבי** – מתאים את עצמו לטקסט שהוא מקודד באותו רגע. לכן לא צריך לשלוח שום מפתח.
- לא מבוסס על שכיחויות, הוא מtabסס על תוכנה אחרת של שפה טבעית – יש בה **זיכרון**. זאת בניגוד להאפמן, שמtabסס על שכיחיות בהקשר לקרופוס מסוים, וצריך את המידע הזה בשביל לבצע אחר כך **decompress**.

הרישון: השתמש בעובדה שבטקסט יש חוזרת. נסתכל על החזרה השנייה של טקסט באורך  $k$ , וכך לקודד אותו, נעשה רפרנס אחרת בטקסט למקום הרישון שבו אותו הטקסט באורך  $k$  הוזכר. **צריך ל漉ת אחרת  $\Rightarrow$  צעדים ולהעתיק  $k$  תווים.**

```
>>> LZW_decompress([[3, 'a', 'b', 'abcabcabc',
```

למה זה עובד? איך אפשר להעתיק 6 תווים כשייש רק 3? זה איטרטיבי תוך כדי פירישה. בהתחלה יש  $abc$ , מעתיקים 6 תווים: תחילת מעתיקים את  $abc$  (הולכים 3 אחרת ומעתיקים 3 תווים), ואזשוב מעתיקים  $abc$  (הולכים שוב 3 אחרת ומעתיקים את 3 התווים הנוצרים).

### יצוג באמצעות ביטים:

באלגוריתם זה ההמלצת היא להסתכל על  $4095 = 1 = 2^{12} - 1 = 4096 - 1 = W$  התווים האחרונים:

- יתרון – אפשר לייצג את המספר באמצעות 12 ביטים. לא צריך 400 ביטים, לא נבדך הרבה מאוד על  $W$ . אנחנו יודעים לזהות ISR – שר חוזרת, 12 הביטים הראשונים מייצגים את  $W$ .
- חיסרון – חוזרת שהן לפני ה- $W$  תווים האחרונים לא יכנסו.

את  $k$  ניצג עם 5 ביטים, כלומר  $31 = 32 - 1 = 2^5 - 1 \leq k \leq L$ , ולכן סה"כ **נוצרן 17 ביטים**.

נחלק את ה-compression לשני שלבים:
**1. ייצוג ביןים: ניקח מחרוזת אותיות ונΚודד אותה ל-intermediate – אותיות מקוריות וחזרות.**

## LZ Compression - Solution

- Step 1: fix **length**
  - $char \rightarrow ascii(char)$  in exactly 7 bits
  - $[back, rep] \rightarrow 000 \dots bin(back) + 000 \dots bin(rep)$ 
    - Requires  $\log |\max back| + \log |\max rep|$  bits
- Step 2: add **indicator** to distinguish between char and repetition
  - $char \rightarrow 0 + ascii(char)$ 
    - Requires 8 bits
  - $[back, rep] \rightarrow 1 + 000 \dots bin(back) + 000 \dots bin(rep)$ 
    - Requires  $1 + \log |\max back| + \log |\max rep|$  bits
- If  $|\max back| = |\max rep| = n$ , each rep. takes  $O(\log n)$  bits
- Choose  $|\max back|, |\max rep| = O(1)$ . Why?

נלק לבכל offset אפשרי (בלערך של  $m$  עד שגיגע ל- $W = m$ ) ונבדוק חזרה בכל חלק, עד לגודל  $L$ . מספר ההשוואות הוא  $WL$ . אז זה יוצא ( $n$ )  $O$  פשוט עם מקדם לא נעים של  $2^L$ . כדי למצאו את החלקים שחזרות על עצםם באופן מקסימלי – ניעזר בפונקציה `maxmatch`.

**2. ייצוג בינארי: נתרגם את זה לביטים.**

כל אות נקודד ב-ASCII בלבד 7 ביטים, ובכל צמד של  $\{k, m\}$  נצטרך 17 ביטים. כדי לדעת איך לפרש, לפני כל ASCII נשים '0', ולפני כל צמד של  $\{k, m\}$  נשים '1'. אז **כל** צמד יהיה 18 ביטים, וכל ASCII יהיה 8 ביטים. لكن, חזרה בגודל 2 לא שווה לקודד, ומאריך 3 ומעלה נשתמש בדחיפסה.

טעות לוגית בשקף 24 – התנאי של  $2 \leq k$  רלוונטי לגדים של  $W$  ו- $L$  הדיפולטים. אם הם ישתנו, צריך לחשב  $\log W + \log L$

סיבוכיות: maximal match – במקרה הגראע ביותר, יהו  $WL$  חיפושים עבור כל תוו ותו בטקסט. בلومר ( $n$ )  $O(n^{2^L})$ . הקבוע הוא נורא ואiom. גם אם נקודד רק חצי מהתווים, קיבל פשוט  $O(2^{16})$ .

**תרגול 12:**

- נבחר את  $L$  ו- $W$  להיות קבועים שאינם תלויים ב- $m$  כדי שסיבוכיות זמן הריצה של הקוד לא תהיה בעייתית.
- **נשים לב** שתמיד **a** יהיה לפחות 3 – חזרה של תוו אחד היא לא שווה מבחינת הקידוד, תיקח 18 ביטים כאשר עם ASCII אפשר להסתפק ב8 ביטים בלבד. גם חזרה של 2 תווים לא שווה! נסתכל רק על חזרות באורך 3 לפחות.
- בשלב הראשון – maxmatch עובר על כל האפשרויות, ובמיה את ההתאמה הכי קרובה בעלת האורך הכי גדול. ההתאמה ב-4 תווים תהיה יותר טובה מההתאמה ב-3 תווים – זה יהיה **a** שיוחזר.  $m$  יסמן כמה צדדים צריכים לבלכת אחרת.
- בשלב השני – אנו מבצעים fillz כדי לרדף אפסים בהתאם לגודל שאנו חצינו שייהה, לתווים ב-ASCII נרדף באפסים בהתחלה כדי הגיעו ל-7 ביטים, ועבור  $k, m$  נעשה זאת לפי כמות הביטים שצריך ל- $L$  (עבור  $k$ ) ול- $W$  (עבור  $m$ ).

דוגמאות נוספת:

- “abcab”  $\rightarrow$  [‘a’, ‘b’, ‘c’, ‘a’, ‘b’]
- “abcabcdabc”  $\rightarrow$  [‘a’, ‘b’, ‘c’, [3,3], ‘d’, [4, 3]]
- “a”\*10  $\rightarrow$  [‘a’, [1,9]]
- “a”\*40  $\rightarrow$  [‘a’, [1, 31], [1, 8]]

## I – קודים לגילוי וلتיקון שגיאות

### דוגמאות בסיסיות

#### ספרת ביקורת:

ראינו כבר את הקוד לחישוב ספרת ביקורת בת.ז. ישראלית.

- ספרת הביקורת לא מוסיפה שום מידע חדש, הוא עוזרת לוודא ש-8 הספרות שמשופיעות לפניה, תקיןות.
- לא נוכל להזות אם יש שגיאה ב-2 ספרות למשל, כי נוכל ליצור סכום דומה של ספרות באמצעות שגיאה בספרה אחת, או בשגיאה בשתי ספרות.
- יכול להיות גם מצב שיש טעות בשתי ספרות והן מקוזחות אחת את השניה, וקיבלונו עוד ת.ז. חוקית.

#### "קסם קלפיים":

השורה והעמודה האחרונות במטריצה מתknות את מספר ה-1-ים בכל שורה ועמודה להיות זוגי.  
המידע האקראי הוא במטריצה  $1 - a \times 1 - a$ . בלביטים שעוטפים את המטריצה זו הם לא אקראים, והם שם כדי לתkn את הזוגיות.

- נוכל למצאו שני שיטות אחד בלבד. הצלבה של שורה ועמודה שבה מספר ה-1-ים לא זוגי.
- שני שיטות יוצרים הופך את המטריצה ללא תקינה. אך לא ניתן לדעת מה המטריצה המקורי.

### מרחב האמיניג

#### :The binary symmetric channel

לפי מודל Shanon-Weaver אנו מכנים תקשורת, שבה מטרתנו שתהיליך-decoding' יהיה כמו שיותר מוצלח.

**Goal:**  $\text{Prob}(\text{original message} = \text{decoded message}) \approx 1$

הנמען מקבל סיגנל כלשהו שאולי קרו בו שגיאות (ביטים התהפהכו). הנמען מפעיל אלגוריתם decoding':

- תיקון – אם הנמען יכול לשחזר את ההודעה המקורי גם אם קרו שגיאות בסיגנל.
- גילוי – הנמען יכול להצביע על בר שמשהו בסיגנל שגוי, בלי לדעת איך לתקן.

הערה: הנמען לעולם לא יודע אם ההודעה שהוא חושב שהיא תקינה אחרי decoding', היא אכן ההודעה המקורי שנשלחה אליו.

הנחות מוקלטות:

- ההודעה המקורי היא בגודל קבוע של ביטים.
- ביטים לא נעלמים.
- השגיאה היחידה האפשרית: ביט התהפהן.
- ערך-binary symmetric channel: הסיכוי שכל ביט יטהף, בלתי תלוי בסיכוי שביט אחר יטהף.

#### מרחב האמיניג:

מרחב האמיניג: עברו שתי מחרוזות באורך  $n$ , מרחב האמיניג הוא מספר הקואורדינטות בהן הן שונות זו מזו.

```
def hamming_distance(s1, s2):  
    assert len(s1) == len(s2)  
    return sum(s1[i] != s2[i] for i in range(len(s1)))
```

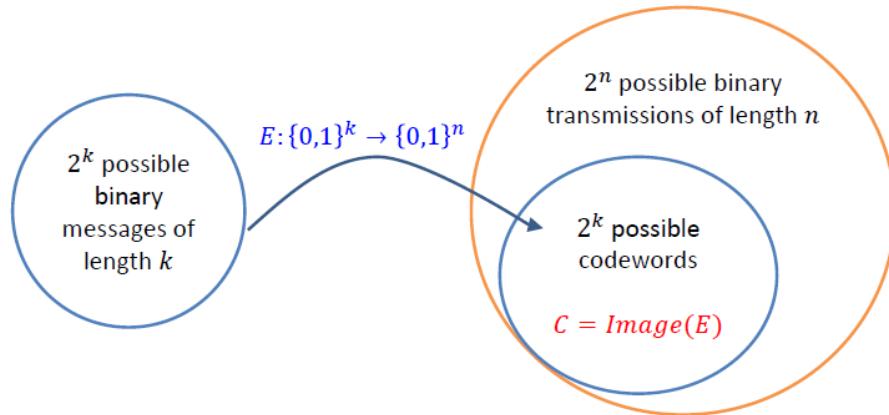
אנו משתמשים כאן בגנרטור, כי אנו בסה"כ רצים למספר כל פעם, ואין לנו צורך בזיכרון זהה לשימור את כל הערכים.



### :Closest codeword decoding

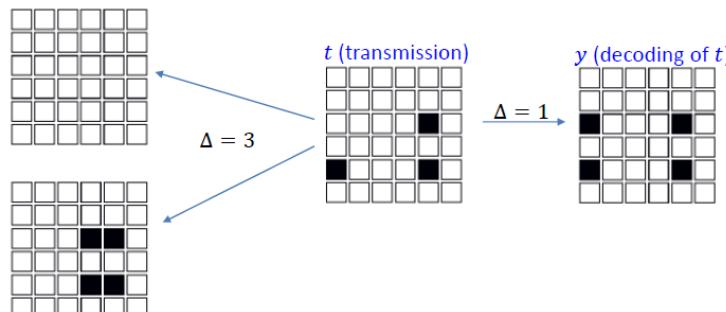
ההודהה המקורית באורך  $k$ , הקידוד הוא באורך  $n$ . אם קיבל מילה שאינה מילת קוד, נרצה להחזיר את הקרובות ביותר אליה מבחינת מרחק האמינה.

- ▶ An **encoding**,  $E$ , from  $k$  to  $n$  ( $k < n$ ) bits, is a one-to-one mapping  $E : \{0,1\}^k \mapsto \{0,1\}^n$ .
- ▶ The set of **codewords** is  $C = \{y \in \{0,1\}^n \mid \exists x \in \{0,1\}^k, E(x) = y\}$ .  
The set  $C$  is often called **the code**.

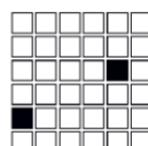


דוגמה על קסם הקלפיים:

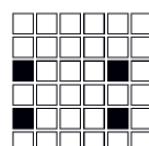
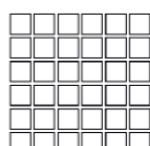
- יש רק מילה אחת למרחק 1.
- מספר ה-1ים הוא אי-זוגי, لكن שבוי של 2 ביטים יփוך את זה לא-זוגי ובמטריצה חוקית, לא יכול להיות מספר אי-זוגי של 1ים.
- לכן אין מילים למרחק 2 מהשדר הזה.
- מילים למרחק 3 קיימות.



בדוגמה הבאה, אין מילה למרחק 1 או 3 ממנו, כיון שיש מספר זוגי של 1ים. למרחק 2 יש את המטריצה הריקה שוכלה אפסים, וזאת שמשלימה אותו למלבן. לא נוכל להבדיע בין האופציות.



- There is no codeword at **distance 1** from this string (why?). There are **exactly two** codewords that are at **distance 2** from this string. They are shown below. In such situation, closest codeword decoding announces **an error**.



## מרחק האMING מינימלי

נסמן  $d$  מרחק של קוד  $C$  על ידי מרחק האMING מינימלי של הקוד  $C$ . זה מרחק האMING מינימלי בין כל זוג מילות קוד בקוד  $C$ .

$$\Delta(C) = \min_{y \neq z \in C} \{\Delta(y, z)\}.$$

זהות מילה שאינה מילת קוד - מילת הקוד הקרויה ביותר, היא במרחק לפחות  $d$  (זה המרחק המינימלי בין מילות קוד). לכן, אם יצאנו מילת קוד והתקדמנו במרחק שקטן מ- $d$ , **בוודאות לא הגיעו למילת קוד**.

### קודקסם קלפים:

- עבור קודקסם קלפים – נראה כי המרחק של הקוד  $4 = d$ .
- $\leq 4$ : מוכחים על ידי דוגמה.
- $\geq 4$ : נראה ש- $d$  הוא לא 1 או 3. לא ניתן שקו 2 שגיאות וקיבלנו מטריצה חוקית.

המרחק בין שתי מטריצות חוקיות הוא 4:

- מטריצה שהיא במרקח 1 מטריצה חוקית, היא בעצם מרחק של לפחות 3 מטריצה חוקיות אחרות. תמיד נניח שסביר שקו 3 מינימום שגיאות, שכן נרצה לתקן את השגיאה ה-1, ולא את 3 השגיאות. התיקון יהיה נכון בהנחה שקו 3 מינימום שגיאות.
- אם אנחנו במרקח 2 שגיאות מטריצה חוקית, לא נדע לתקן. יש שתי אפשרויות לבאן ולכאן.

### קוד תעוזת זהות:

- $d = 2$
- נוכל לזהות שגאה 1, אבל לא נוכל לתקן.
- אם קו 2 שגיאות לא בהכרח נוכל לזהות.

### The geometry of codes

נראה אינטראקטיבית גיאומטרית לסיפור זהה.

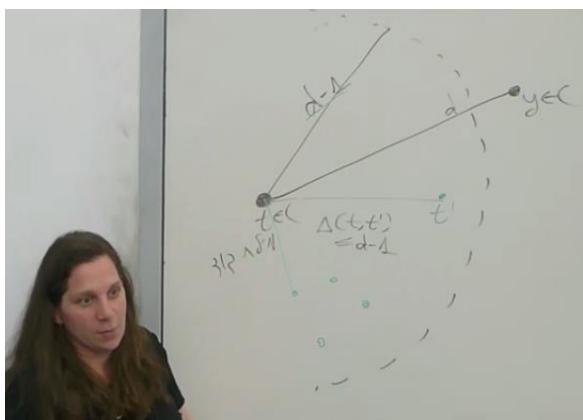
### טענה:

Suppose  $d = \Delta(C)$  is the minimum distance of a code,  $C$ . Then this code is capable of detecting up to  $d - 1$  errors, and correcting up to  $\lfloor (d - 1)/2 \rfloor$  errors.

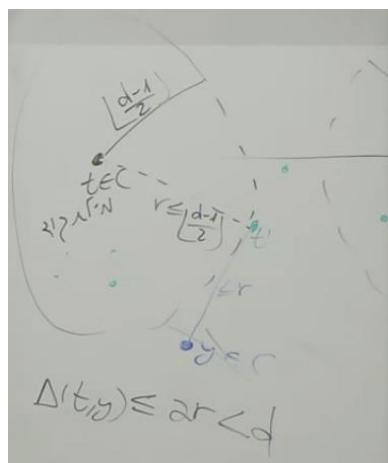
### הוכחה:

Let  $y \in \{0, 1\}^n$ . The sphere of radius  $r$  around  $y$  is the set  $B(y, r) = \{z \in \{0, 1\}^n \mid \Delta(y, z) \leq r\}$ .

- **זהות של עד  $d - 1$  שגיאות:**  $C \in \mathcal{C}$  מילת קוד. המילה חוויתה  $r$  שגיאות, עד שהפכה ל- $t'$ . כלומר  $\Delta(t, t') = r \leq d - 1$ . אין אנו יודעים כי אין אף מילת קוד שמרחקה מ- $t$  קטן ממש מ- $d$ . בהברה  $t'$  היא לא מילת קוד, ונדע לזהות את זה.
- אם נציג ספירה מסביב למילה, לא נקבל שום מילה תקינה אחרת פרט למילה זו.



We say that a code,  $C$ , is capable of detecting  $r$  errors if for every  $y \in C$ ,  $B(y, r) \cap C = \{y\}$ .

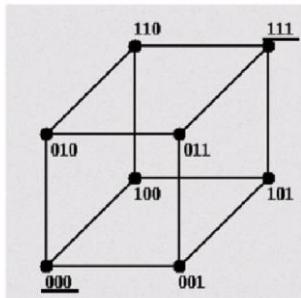


- **תיקון של עד  $\left\lfloor \frac{d-1}{2} \right\rfloor$  שגיאות:** זה היא מילת הקוד הקרויה ביותר ל- $d$ . לא יתכן שיש מילת קוד קרויה יותר ל- $d$ . נניח בשלילה שיש מילה אחרת  $z$  שההמරחক בינה לבין  $z$  הוא לפחות  $d$ . לפי איו שוויון המשולש:  $d < 2r \leq (t, y)$ . לא יתכן כי  $d$  הוא המרחק המינימלי של הקוד.
- לא נרצה להגיע במצב שאנו חנו לא יודעים لأن תקין, הבודרים סביב כל מילה הם זרים! אם מילה הייתה מופיעה בשתי ספורות של מילים שונות, לא היינו יודעים לאן תקין אותה.

We say that a code,  $C$ , is capable of **correcting**  $r$  errors if for every  $y \neq z \in C$ ,  $B(y, r) \cap B(z, r) = \emptyset$ .

נרצה ש- $d$  יהיה כמה שיותר גדול, אבל הפרמטר השני שעומד בסתרה אליו הוא  $t$  – כמה ביטים אנחנו מוסיפים על הודעה המקורית. את  $t$  נרצה שייהי כמה שיותר קטן. שתי המטרות האלה הן סותרות.

### קוד חזרה



Codewords are underlined.

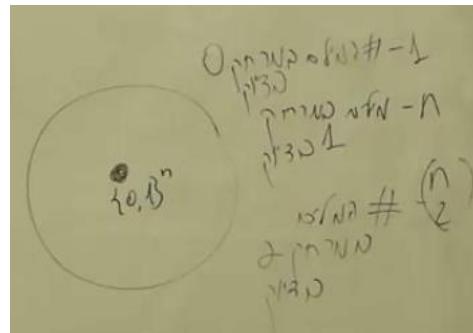
נניח שאליס שולחת הודעה לבוב וחזרה על כל בית 3 פעמים. כלומר, שני הביטים אחרי הביט הראשון צריכים להיות זהים לו.

Encoding: message → transmission (codeword)

0	000
1	111

למשל את 001 נתקן ל-000.

- $k=1, n=3, d=3$
  - יש רק 2 מילוטים קוד – תת קבוצה של כל האופציות באורך 3 מעל  $\{0,1\}$ .
- כמה מילים יש בכדור שהדיאט שלו הוא  $2 \binom{n}{2} + 1$ .

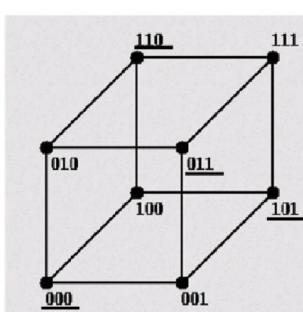


### ביט זוגיות

$k=2$  (length of message)  
 $n=3$  (length of codeword)  
 $d=2$  (in fact all HD are 2 here)

Can detect 1 error, correct 0

קוד זוגיות חד מימדי. באן הודעה המקורית היא באורך 2 ( $k=2$ ), אם מספר ה-1 זוגי הביט יהיה 0, אם מספר ה-1 אי-זוגי הביט יהיה 1. **הוא דואג שמספר ה-1ים יהיה זוגי.** באן  $d=2$ .



Codewords are underlined.

מהו היחס בין  $d, k, n$ ?

- בקוד חזרה  $-3k = n$  כמות הביטים שהוספנו הייתה **לינארית** בגודל ההודעה  $2k$ . קיבלנו  $3 = d$ .
- בקוד זוגיות  $-1 + k = n$  כי אנו מוסיפים בית אחד, כמות **קבוצה**. אבל  $2 = d$ , לא ידוע האם אפילו שגיאה אחת.

נרצה לשאוף ל- $d$  גדול, ול- $k$  קטן. נוכיח כי בהינתן  $k$  מסוים, מתקיים  $1 \leq n - k + 1 \leq d$ .

- נכתוב את כל  $2^k$  מילוטים הקוד שיש בקוד.
- אורך כל מילוט קוד הוא  $n$ , כל מחזורות אחרת באורך  $n$  היא לא מילוט קוד.
- כל זוג מילוטים נבדלות לפחות בפחות  $d$  ביטים – כי המרחק המינימלי בין כל זוג בקוד הוא  $d$ .
- נמחוק את  $1 - d$  הקודאות הראשונות, ונשארנו עם כל זוג מחזורות יהיה הבדל של לפחות ביט  $d$ .
- אורך כל מחזורות היה  $n$  ובכעת הוא  $(1 - d) - n$ . עדין אין מחזורות זרות באנו, כי בין כל זוג מחזרות יהיה הבדל של לפחות ביט  $d$ .
- אם בדיק מחקנו את בדיק  $1 - d$  הבדיקות, נשארנו עם הבדל של בית 1. יש לנו  $2^k$  מחזרות באורך  $(1 - d) - n$  שהן כולן שונות זו מזו.
- באופן כללי, יש לנו  $(1 - d) - n$  מחזרות מהאורק הזה, ובהכרח מתקיים  $1 \leq 2^{n-d+1} \leq 2^n$  ולכן  $1 \leq n - k + 1 \leq d$ .

קוד האמיןיג:

$(2^t - 1, 2^t - t - 1, 3)$  – Hamming(7,4,3) – קוד האמיןיג זהה בעצם אומר  $3 = d, k = 4, n = 7$ . עברו  $t$  במשהו אפשר לדבר על  $(1,3)$  או לא כי הם בערך  $2^t$ . אנחנו מוסיפים להודעה המקורית  $t$  ביטים וזה מספר לוגירטמי ביחס ל- $k$  או לא כי הם בערך  $t$ . לעומת זאת קוד חזרה וקוד זוגיות, עם תוספת לוגירטמית נקבל את אותו  $d$ .

ההודעות המקוריות הן באורך 4, וננסמן את מספרי הביטים כך:

$$(x_3, x_5, x_6, x_7) \in Z_2^4 = \{0, 1\}^4$$

ובצע הוספה של שלושה ביטים כך שכל בית הוא בית זוגיות של השלשה הרלוונטית:

$$(x_3, x_5, x_6, x_7) \longrightarrow (x_1, x_2, x_3, x_4, x_5, x_6, x_7) ,$$

where  $x_1, x_2, x_4$  are **parity check bits**, computed (**modulo 2**) as following:

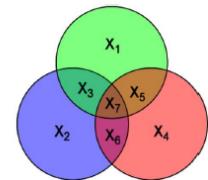
$$\begin{aligned} x_1 &= x_3 + x_5 + x_7 \\ x_2 &= x_3 + x_6 + x_7 \\ x_4 &= x_5 + x_6 + x_7 \end{aligned}$$

אם יש בעיה למשל ב- $x_2$ , אז האשם היחיד לכך הוא  $x_3$  והוא יתכן אותן.

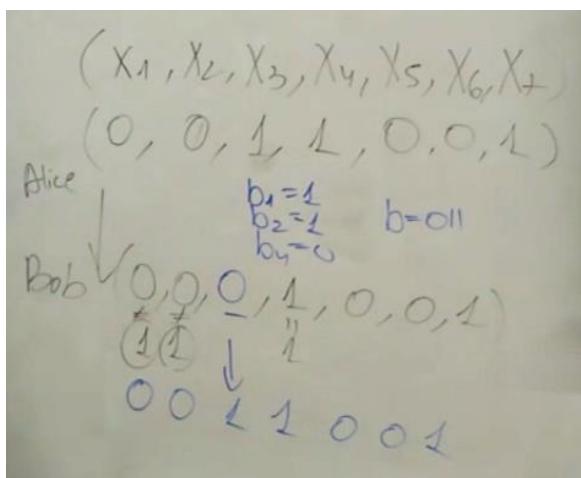
- Suppose that a **single error** has occurred

Flipped bit	Parity bits that look illegal
$x_1$	$x_1$
$x_2$	$x_2$
$x_3$	$x_1, x_2$
$x_4$	$x_4$
$x_5$	$x_1, x_4$
$x_6$	$x_2, x_4$
$x_7$	$x_1, x_2, x_4$

$$\begin{aligned} x_1 &= x_3 + x_5 + x_7 \\ x_2 &= x_3 + x_6 + x_7 \\ x_4 &= x_5 + x_6 + x_7 \end{aligned}$$



- So the decoder knows which bit was flipped and can fix the (**single**) error!
- In fact, the error is at location  $b = b_4 b_2 b_1$  where  $b_i = 1$  if  $x_i$  is illegal and  $b_i = 0$  otherwise!
- What if **more than one** error occurred?



**תרגול 13**

אם מצאנו שתי מילוט קוד עם מרחק  $M$ , אנו יודעים כי המרחק של הקוד הוא לפחות  $M$ , כלומר  $M \leq d$ . כי המרחק של הקוד הוא המרחק המינימלי בין שתי מילוט קוד, ולכן הוא לא יהיה גדול מהמרחק שמצאנו. הוא יכול להיות רק קטן יותר (או שווה).

ביש זוגיות – נוכיח כי  $2 = d$

כיוון ראשון -  $2 \geq d$ :

- בולם לכל  $x, y \in \{0,1\}^k$  מתקיים  $\Delta(C(x), C(y)) \geq 2$ . מתקיים:  $\Delta(x + par(x), y + par(y)) = \Delta(x, y) + \Delta(par(x), par(y))$
- אם  $2 \geq \Delta(x, y)$  סימנו.
- אם  $1 = \Delta(x, y)$  אז שינוי של בית אחד יגרור שינוי בזוגיות של כמהות ה-1-ים ולכן ביש זוגיות שונה. כך נקבל שהмарחק בין ביטי הזוגיות הוא 1, וקיים סה"כ 2.

כיוון שני -  $d \leq 2$ :

- נביא דוגמה לשתי מילוטים שהмарחק ביניהן הוא 2. למשל: 11 ... 0. כאן נקבל  $2 = d$ .

:Index Code – IC

## Index code (IC) construction

- Pick  $k = 2^\ell - 1$  for some  $\ell$
- Encode a message  $x = x_1 \dots x_k$  as follows:
  - For every  $i$  such that  $x_i = 1$ , take  $bin(i)$  (note:  $|bin(i)| = \ell$ ).
  - Compute bitwise xor ( $\oplus$ ) over all such  $i$ , call it  $EC(x)$
  - Transmit  $x \circ EC(x)$
- Example:  $x = 0110110$  (for  $\ell = 3$ )
  - $2 = \textcolor{red}{0} \textcolor{blue}{1} \textcolor{green}{0} \oplus$
  - $3 = \textcolor{red}{0} \textcolor{blue}{1} \textcolor{green}{1} \oplus$
  - $5 = \textcolor{red}{1} \textcolor{blue}{0} \textcolor{green}{1} \oplus$
  - $6 = \textcolor{red}{1} \textcolor{blue}{1} \textcolor{green}{0} \oplus$
  - $EC = \textcolor{red}{0} \textcolor{blue}{1} \textcolor{green}{0}$
- Transmit  $0110110\textcolor{red}{0} \textcolor{blue}{1} \textcolor{green}{0}$

Reminder:

- For bits:  $i \oplus j = 1$  if  $i \neq j$
- For words:  $x \oplus y = x_1 \oplus y_1 \dots x_n \oplus y_n$

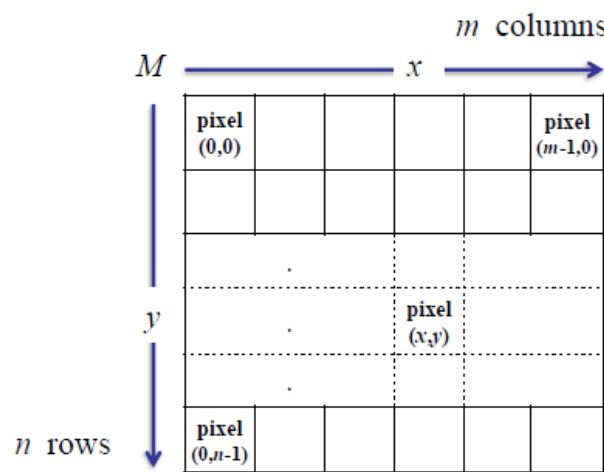
## $\Delta(IC) \geq 2$ , lower bound

- Let  $x, x'$  be two messages
- If  $\Delta(x, x') \geq 2$  we are done since:
  - $\Delta(x \circ EC(x), x' \circ EC(x')) = \Delta(x, x') + \Delta(EC(x), EC(x')) \geq \Delta(x, x') \geq 2$
- So we only need to show that if  $\Delta(x, x') = 1$ ,  $EC(x) \neq EC(x')$ 
  - Why is this enough?
- Let  $j \in \{1, \dots, k\}$  be an index such that (wlog)  $x_j = 1, x'_j = 0$ 
  - All other “on” indices in  $x, x'$  are the same:  $i_1, i_2, \dots, i_t$
- $EC(x) = i_1 \oplus i_2 \oplus \dots \oplus i_t \oplus j$
- $EC(x') = i_1 \oplus i_2 \oplus \dots \oplus i_t$
- $EC(x) \oplus EC(x') = j \neq 0 \leftrightarrow EC(x) \neq EC(x')$

## [ – ייצוג תמונה ]

### ייצוג תמונה דיגיטלית

תמונה דיגיטלית מיוצגת לרוב באמצעות מטריצה דו-מימדית. כל תא  $[x, y]$  הוא בעמודה  $x$  ובסורה  $y$ . כל פיקסל מסמן את הבrightness/צבע של המיקום זהה בתמונה.



דרך מקובלת להציג צבע הוא באמצעות פורמט שנkirא RGB, כאשר כל צבע ניתן לפרק לרכיבים שלו ב-RGB: אדום, ירוק, כחול. בכל פיקסל נשמר 3 ערכים בהתאם. כל ערך זהה הוא מספר בין 0 ל-255.

אם התמונה אינה צבעונית, בכל פיקסל נשמר ערך אחד שמייצג את הבrightness שלו מבחינת גוון אפור. נייצג גם כאן בין 0 ל-255 את גוון האפור. לכל פיקסל נדרש ל-8 ביטים עבור כך.

### :Bit Depth

באופן כללי, לא צריך לייצג כל פיקסל על ידי 8 ביטים. ככל שניציג על ידי יותר ביטים, יהיו יותר גווני אפור. לעין האנושית, לא צריך לעבוד עם יותר מ-8 ביטים. עבורנו bit depth יהיה 8.

### :process img

הfonקציה `process img` מעדכנת את `new_image` על ידי תוצאה הפעלת הfonקציה `do` על הפיקסל במטריצה המקורית.

### תיקוי רעשים

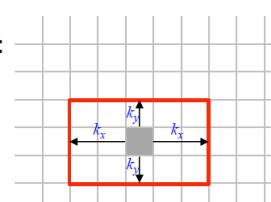
#### Neighborhood of a Pixel

- **Local denoising:** pixel at  $(x, y)$  will change as a function of its surrounding pixels (called **neighborhood**, or **environment**)
- **Neighborhood** of a pixel  $(x, y)$  is the set of all pixels **close** to it. For example, a  **$3 \times 3$  square** neighborhood:

$$N_{3 \times 3}(x, y) = \begin{bmatrix} x-1, y-1 & x, y-1 & x+1, y-1 \\ x-1, y & x, y & x+1, y \\ x-1, y+1 & x, y+1 & x+1, y+1 \end{bmatrix}$$

- More generally, a rectangular neighborhood of dimensions  $(k_x, k_y)$  is a  $(2k_x+1)$ -by- $(2k_y+1)$  rectangle:

When  $k_x = k_y = 1$  we get a  $3 \times 3$  square.



## Local Operator – Python Code

```

def local_op(img, op, kx=1, ky=1):
    w,h = img.size
    mat = img.load()
    new_img = img.copy()
    new_mat = new_img.load()

    for x in range(w):
        for y in range(h):
            # 4 corners, do not exceed image boundaries
            left  = max(x-kx, 0)
            up    = max(y-ky, 0)
            right = min(x+kx, w-1)
            down  = min(y+ky, h-1)

            # flatten 2D neighborhood into 1D list
            neighbors_list = [mat[xx,yy] for xx in range(left, right+1) \
                               for yy in range(up, down+1)]
            # apply op in list and assign result to pixel x,y
            new_mat[x,y] = op(neighbors_list)

    return new_img

```

Default: 3x3 square neighborhood

36

The operator is applied on the neighboring pixels

```

def local_means(img, kx=1, ky=1):
    mean = lambda lst: round(sum(lst)/len(lst))
    return local_op(img, mean, kx, ky)

```

```

def local_medians(img, kx=1, ky=1):
    median = lambda lst: sorted(lst)[len(lst)//2]
    return local_op(img, median, kx, ky)

```