

Extended Introduction to Computer Science

CS1001.py

Chapter C Basic Algorithms:

Lecture 7b Binary Search in Sorted Sequences

Amir Rubinstein, Michal Kleinbort

School of Computer Science
Tel-Aviv University
Fall Semester 2023-24
<http://tau-cs1001-py.wikidot.com>

* Slides based on a course designed by Prof. Benny Chor

מבנה ונושאי הקורס (באדום - חומר שירד בשל קיצור הסמסטר)

פרק	נושאים מתוכננים
A. יסודות פייתון	<ul style="list-style-type: none"> • תכנות בסיסי: טיפוסים ערכים, משתנים, משפטי תנאי, לולאות, פונקציות, מודל הזיכרון • נושאים נוספים: דקדוקים פורמליים ותהליך הפירוש של פייתון, פונקציות למבדא, ופונקציות סדר גבוה, אקראיות ושימושיה, סוגי שגיאות (תחביר, זמן ריצה), סגנון תכנות "נכון"
B. ייצוג טיפוסים מידע	<ul style="list-style-type: none"> • ייצוג שלמים בשיטה הבינארית • ייצוג מספרים עם נקודה עשרונית בשיטת floating point • ייצוג תווים (Unicode, ASCII)
C. אלגוריתמים בסיסיים וסיבוכיות	<ul style="list-style-type: none"> • חיפוש בינארי, מיון בחירה, מיזוג רשימות ממוינות • סיבוכיות O notation -I
D. חישוב נומרי	<ul style="list-style-type: none"> • מציאת שורש של פונקציה ממשית רציפה בשיטת החציה בעבר: שיטת ניוטון-רפסון, • חישוב נגזרות ואינטגרלים, קירוב ל π
E. רקורסיה	<ul style="list-style-type: none"> • עצרת, פיבונאצ'י, חיפוש בינארי, מיון מהיר, מיון מיזוג, ממואיזציה, דוגמאות נוספות
F. נושאים בתורת המספרים	<ul style="list-style-type: none"> • העלאה בחזקה טבעית בשיטת Iterated squaring • בדיקת ראשוניות הסתברותית (המשפט הקטן של פרמה) • פרוטוקול Diffie-Hellman להחלפת מפתח סודי • מחלק משותף מקסימלי (GCD)
G. תכנות מונחה עצמים (OOP) ומבני נתונים	<ul style="list-style-type: none"> • מחלקות, שדות ומתודות • רשימות מקושרות והשוואה לרשימות של פייתון • עצי חיפוש בינאריים • טבלאות hash • זרמים (streams) ופונקציות גנרטור
H. טקסט	<ul style="list-style-type: none"> • אלגוריתם CYK בעבר: אלגוריתם קארפ-רבין • דחיסת האפמן, דחיסת למפל זיו
I. ייצוג ועיבוד תמונה	<ul style="list-style-type: none"> • ייצוג תמונה דיגיטלית, ניקוי רעש (ממוצע וחציון מקומי), נושאים נוספים לפי הזמן
J. קודים לגילוי ולתיקון שגיאות	<ul style="list-style-type: none"> • ספרת ביקורת, קוד חזרה, ביט זוגיות, מרחק האמינג, קוד האמינג

You are here

Chapter and Current Lesson Plan

- Basic algorithms:
 - Searching in a sorted sequence using "Binary Search"
 - Sorting a sequence using selection sort (next time)
 - Merging sorted lists (next time)
- Time complexity Analysis (next time)

Search

- Search has always been a central computational task. In early days, search supposedly took **one quarter** of all computing time.
- The emergence and the popularization of the **world wide web** has literally created a **universe of data**, and with it the need to pinpoint information in this universe.
- Various **search engines** have emerged, to cope with this challenge. They constantly collect data on the web, organize it, index it, and store it in sophisticated data structures that support efficient (fast) access, resilience to failures, frequent updates, including deletions, etc., etc.
- In this class we will deal with two, much simpler search algorithms:
 - Sequential search
 - Binary search

Search in a Sorted Input

- The computational problem:
 - Input: a **sorted sequence** of elements, and some value called **key**
 - Output: an **index** of an element in the list with the given key (if exists)
- **Naïve** solution:

```
def sequential_search(lst, key):  
    for i in range(len(lst)):  
        if lst[i] == key :  
            return i  
  
    # we get here when key is not in list  
    return None
```

- Efficiency: how many iterations in the **worst** and **best** cases, as a function of the list size, ***n***?

Improved Sequential Search

- Note we did not make any use of the input list being **sorted**.
- Improved solution?

```
def sequential_search_improved(lst, key):  
    for i in range(len(lst)):  
        if lst[i] == key :  
            return i  
        if lst[i] > key :  
            break  
  
    # we get here when key is not in list  
    return None
```

- Alternatively, we could use a **while** loop with the condition
 $i < \text{len}(\text{lst})$ **and** $\text{lst}[i] \leq \text{key}$
- Does this have any effect on the efficiency?

Sequential vs. Binary Search

For unordered lists of length n , in the worst case, a search operation compares the key to **all list items**, namely n comparisons.

On the other hand, if the n element list is **sorted**, search can be performed **much faster**. We first compare input key to the key of the list's **middle element**, an element whose index is $\lfloor (n - 1)/2 \rfloor$.

- If the input key **equals** the middle element's key, we return the middle element and terminate.
- If the input key is **greater than** the middle element's key, we can restrict our search to the **top half** of the list (indices from $\lfloor (n - 1)/2 \rfloor + 1$ to $n - 1$).
- If the input key is **smaller than** the middle element's key, we can restrict our search to the **bottom half** of the list (indices from 0 to $\lfloor (n - 1)/2 \rfloor - 1$).

Each time we either **terminate** or **cut** the remaining search interval by **half**. This is why this process is termed **binary** search.

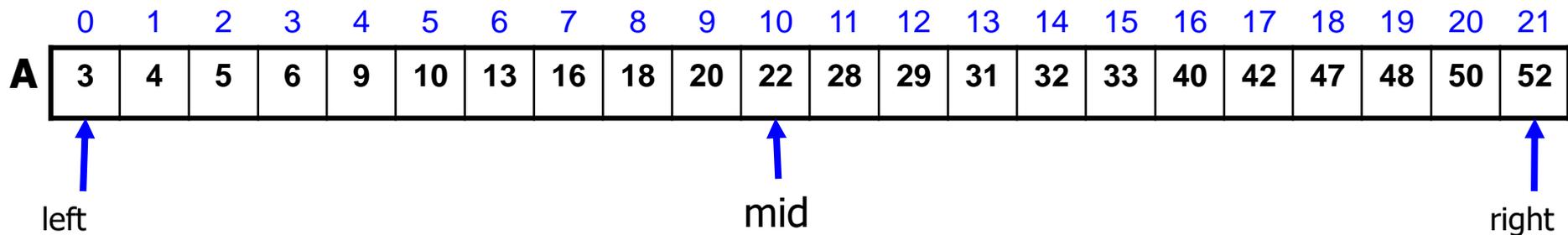
Animated Example - success

Searching for the **existing** Item, 18

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
A	3	4	5	6	9	10	13	16	18	20	22	28	29	31	32	33	40	42	47	48	50	52

Animated Example - success

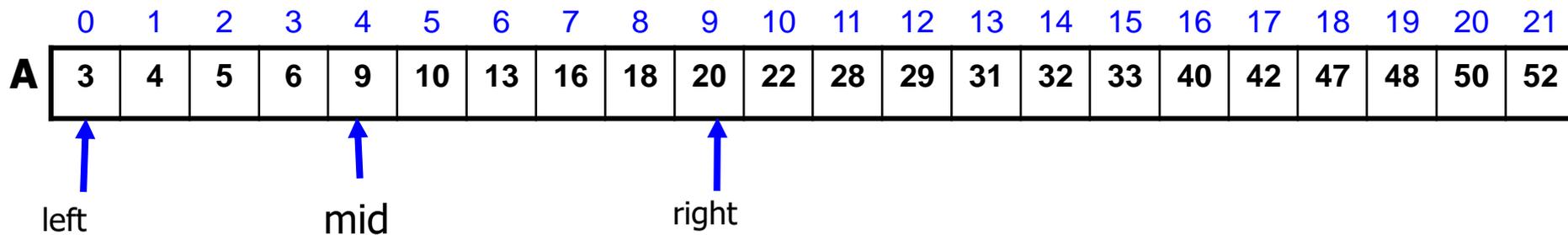
Searching for an **existing** key: 18



$A[\text{mid}] == 22 > 18$

Animated Example - success

Searching for an **existing** key: 18



$A[\text{mid}] == 9 < 18$

Animated Example - success

Searching for an **existing** key: 18

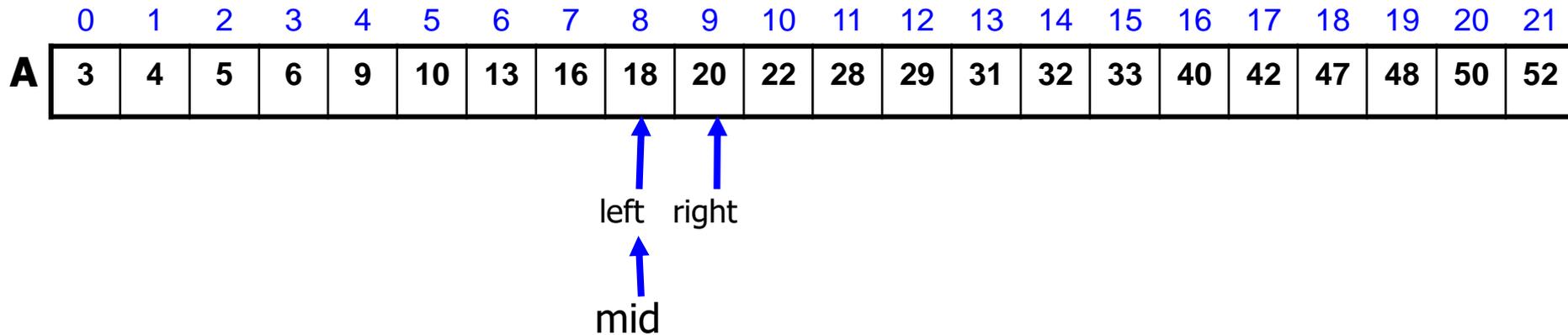
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
A	3	4	5	6	9	10	13	16	18	20	22	28	29	31	32	33	40	42	47	48	50	52

left mid right

$A[\text{mid}] == 16 < 18$

Animated Example - success

Searching for an **existing** key: 18



$A[\text{mid}] == 18$

Found!

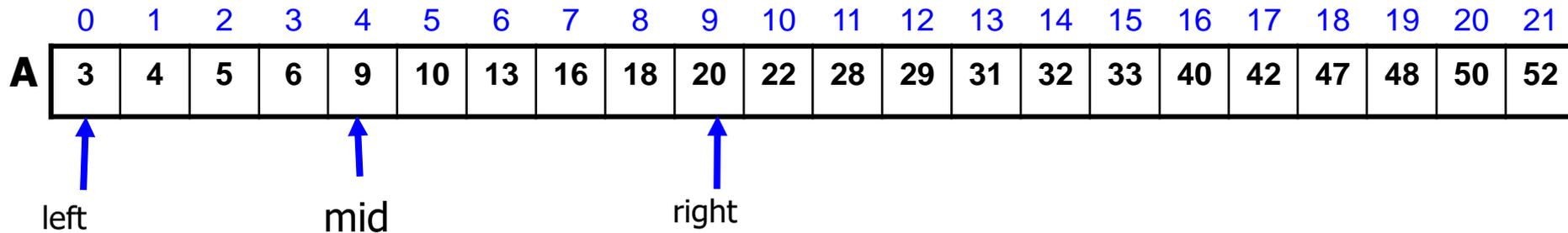
Animated Example - failure

Searching for an **missing** key: 17

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
A	3	4	5	6	9	10	13	16	18	20	22	28	29	31	32	33	40	42	47	48	50	52

Animated Example - failure

Searching for an **missing** key: 17



$$A[\text{mid}] == 9 < 17$$

Animated Example - failure

Searching for an **missing** key: 17

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
A	3	4	5	6	9	10	13	16	18	20	22	28	29	31	32	33	40	42	47	48	50	52

left mid right

$A[\text{mid}] == 16 < 17$

Animated Example - failure

Searching for an **missing** key: 17

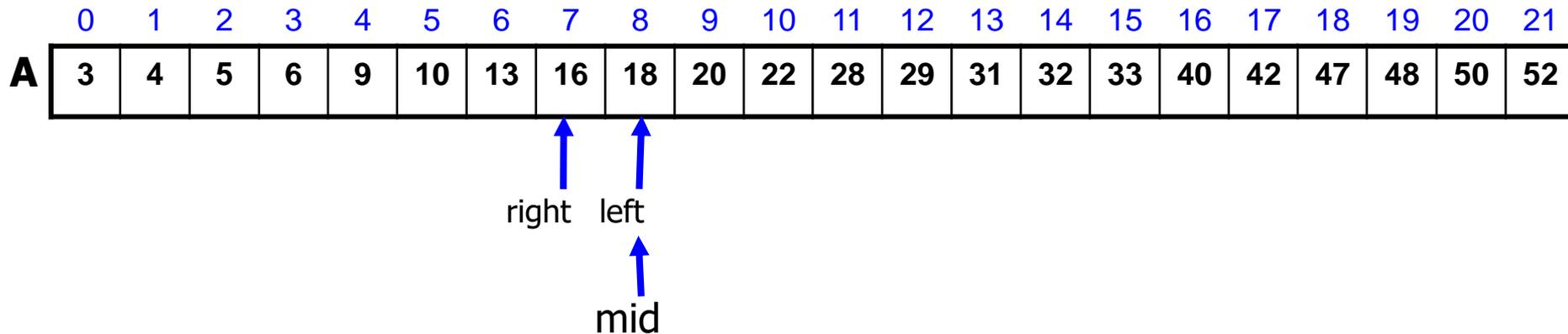
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
A	3	4	5	6	9	10	13	16	18	20	22	28	29	31	32	33	40	42	47	48	50	52

Diagram illustrating a search process on an array **A**. The array contains values: 3, 4, 5, 6, 9, 10, 13, 16, 18, 20, 22, 28, 29, 31, 32, 33, 40, 42, 47, 48, 50, 52. The search target is 17. The current search range is from index 8 (left) to index 9 (right), with the midpoint (mid) at index 8. The value at index 8 is 18, which is greater than the search target 17.

$A[\text{mid}] == 18 > 17$

Animated Example - failure

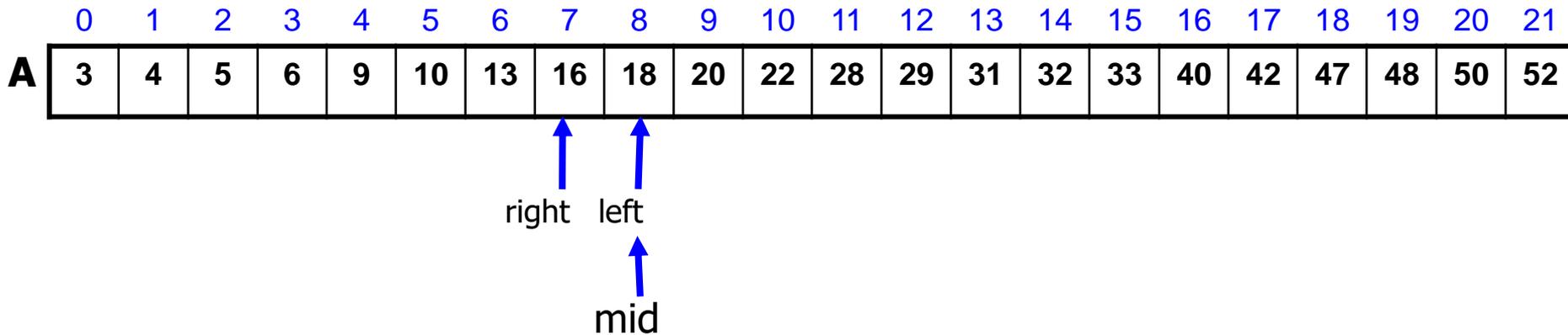
Searching for an **missing** key: 17



$A[\text{mid}] == 18 > 17$

Animated Example - failure

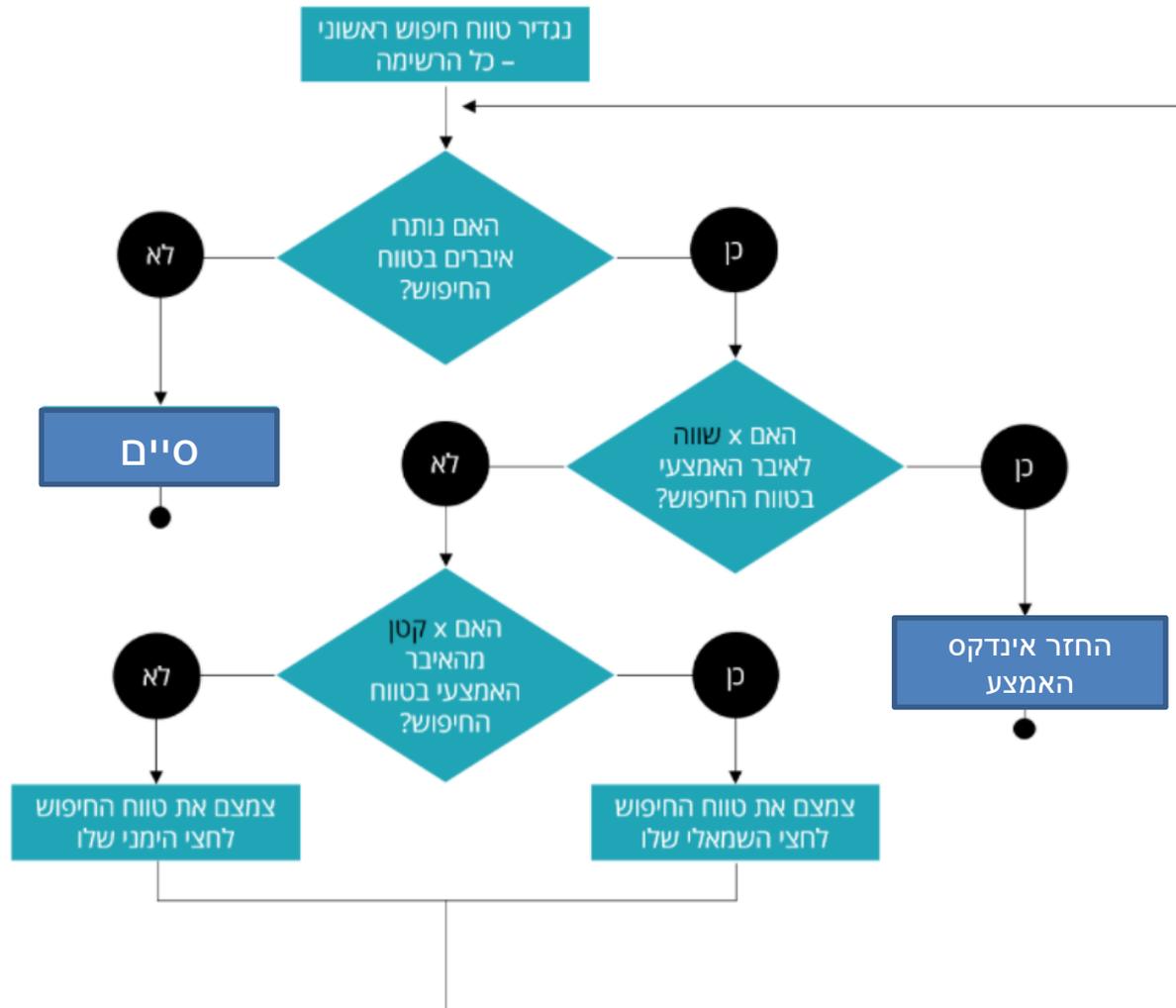
Searching for an **missing** key: 17



left > right

Not Found!

Binary Search – the Algorithm



Binary Search – Python Code

```
def binary_search(lst, key):
    """ lst better be sorted for binary search to work """
    n = len(lst)
    left = 0
    right = n-1

    while left <= right:
        mid = (left+right)//2      # middle rounded down
        if key == lst[mid]:       # item found
            return mid
        elif key < lst[mid]:      # item not in top half
            right = mid-1
        else:                     # item not in bottom half
            left = mid+1

    #print(key, "not found")
    return None
```

Time **Analysis** of Binary Search

- At each stage, we either
 - terminate (if the key is found) or
 - **cut** the size of the remaining list by \sim half.
- In the **worst case** (in terms of running time) the **key** is **not in the list**
- We start with an ordered list of size n
- If key not found, we continue with a **sub-list** of length **at most** $\left\lfloor \frac{n}{2} \right\rfloor$
- If key not found again, we have a **sub-list** of size at most $\left\lfloor \frac{\left\lfloor \frac{n}{2} \right\rfloor}{2} \right\rfloor = \left\lfloor \frac{n}{4} \right\rfloor$
- After i iterations sub-list size is **at most** $\left\lfloor \frac{n}{2^i} \right\rfloor$
- Termination when $left > right$, which means:
$$\left\lfloor \frac{n}{2^i} \right\rfloor = 0 \quad \rightarrow \quad \frac{n}{2^i} < 1 \quad \rightarrow \quad i > \log_2 n$$
- So the total number of iterations is the **smallest integer** $i > \log_2 n$, namely $\lfloor \log_2 n \rfloor + 1$.

$\lfloor \log_2 n \rfloor + 1$ Visits Again

- So we need $\lfloor \log_2 n \rfloor + 1$ iterations in the **worst case** for binary search in a list of size n .
- Does that expression remind you of anything?
- Can you explain this?

Binary Search –

Worst Case Actual Time Measurements

```
import time, random

REPEAT = 20 #repeat execution several times, for more significant results

for f in [sequential_search, binary_search]:
    print(f.__name__)

    for n in [10**6, 2*10**6, 4*10**6]:
        print("n=", n)

        L = [i for i in range(n)]
        key = random.randrange(-1, n) + 0.5

        t0 = time.perf_counter()
        for i in range(REPEAT):
            res = f(L, key)
        t1 = time.perf_counter()
        print("time for", REPEAT, "executions:", (t1-t0))
```



why?

Comic Relief*



* אני מזמין אתכם לשלוח לי הצעות לתמונות שיופיעו על שקפים אלו לאורך הסמסטר

Worst Case Actual Time Measurements

sequential_search

n= 1000000

time for 20 executions: 1.1663285

n= 2000000

time for 20 executions: 2.2578393000000005

n= 4000000

time for 20 executions: 4.5683432999999996

binary_search

n= 1000000

time for 20 executions: 9.840000000060911e-05

n= 2000000

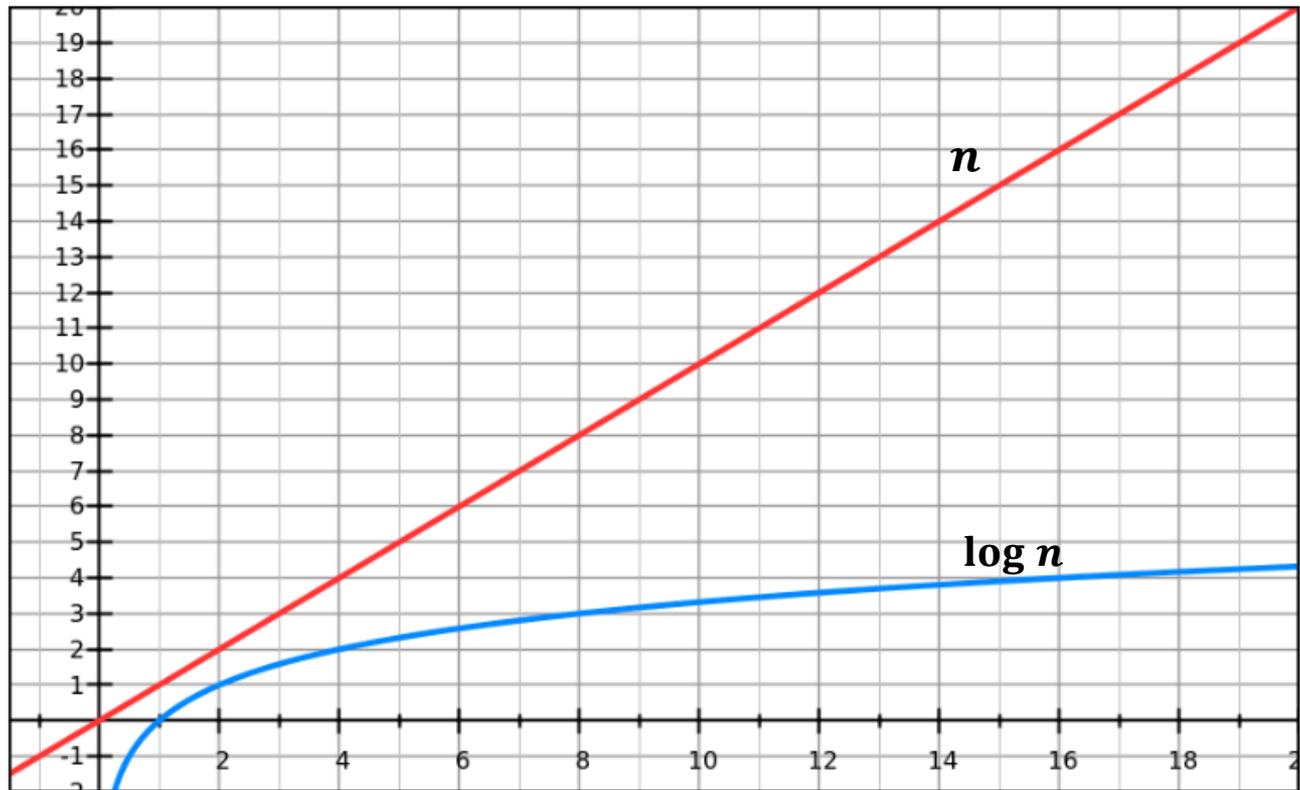
time for 20 executions: 0.00014649999999960528

n= 4000000

time for 20 executions: 0.00015880000000123573

- How would the results change if we searched an element that **exists** in the list? Does it depend on **where in the list** the element is found?

Logarithmic vs. Linear Time Algorithms



Created using <https://graphsketch.com/>

- **Logarithmic:** input $\times 2 \rightarrow$ time + constant
- **Linear:** input $\times 2 \rightarrow$ time $\times 2$ (approximately)