Unit 3 – Amortized Analysis

CS 201 - Data Structures II

Spring 2023

Habib University

Syeda Saleha Raza

$O(n)$ — ?

Asymptotic Analysis - Recap

Complexity Analysis

- When we are trying to find the complexity of the function/ procedure/ algorithm/ program, we are not interested in the exact number of operations that are being performed. Instead, we are interested in the relation of the number of operations to the problem size.

n →

O → order of growth

Example

$3K$ K K K

n n

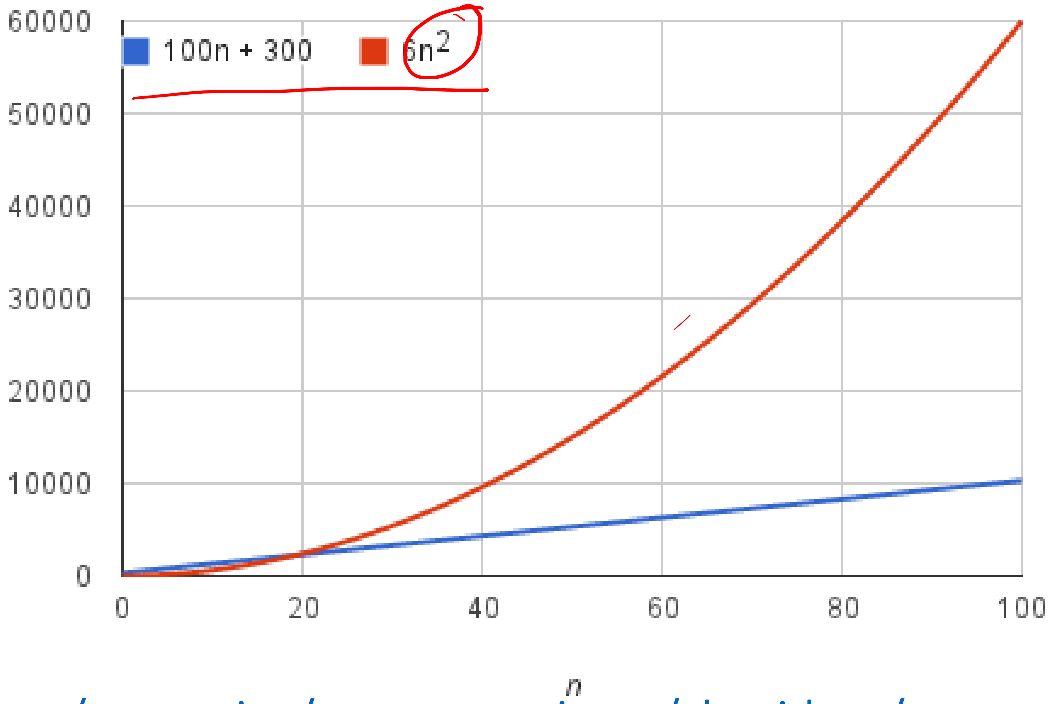
```
a=5
b=6
c=10
for i in range(n):
    for j in range(n):
        x = i * i
        y = j * j
        z = i * j
for k in range(n):
    w = a*k + 45
    v = b*b
d = 33
```

$3K + 3Kn^2 + 2Kn + 1$

$\rightarrow O(n^2)$

Asymptotic Analysis

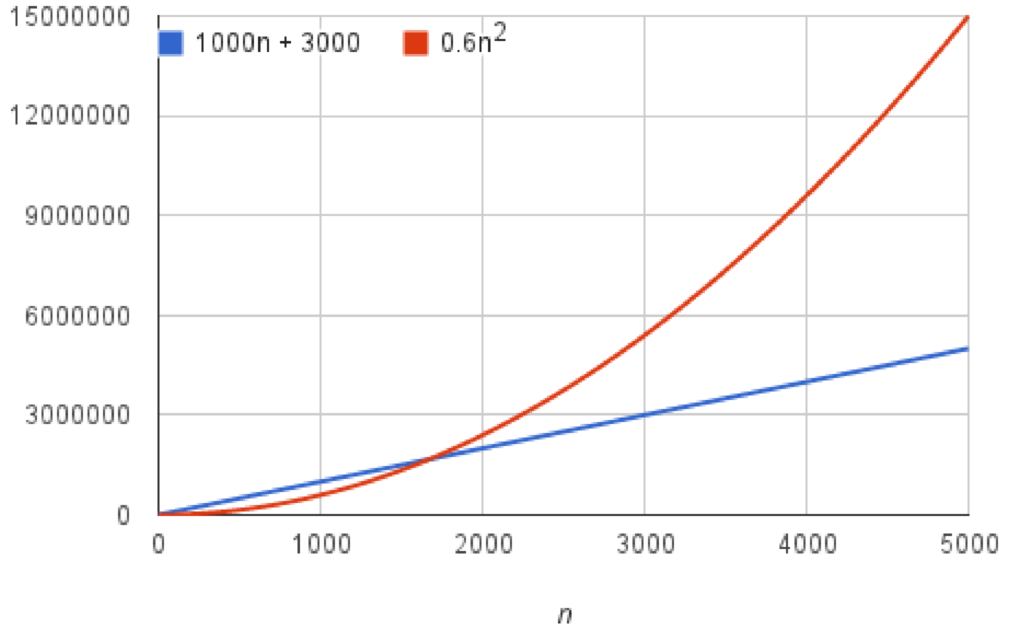
For example, suppose that an algorithm, running on an input of size n takes $6n^2 + 100n + 300$. $6n2+100n+300$
The 6n2 term becomes larger than the remaining terms, 100 n + 300 once n becomes large enough, 20 in this case.



- https://www.khanacademy.org/computing/computer-science/algorithms/asymptotic-notation/a/asymptotic-notation

Asymptotic Analysis

- It doesn't really matter what coefficients we use; as long as the running time is $an^2 + bn + c$ for some numbers a, b, and c, there will always be a value of n for which an^2 is greater than $bn + c$ and this difference increases as n increases.

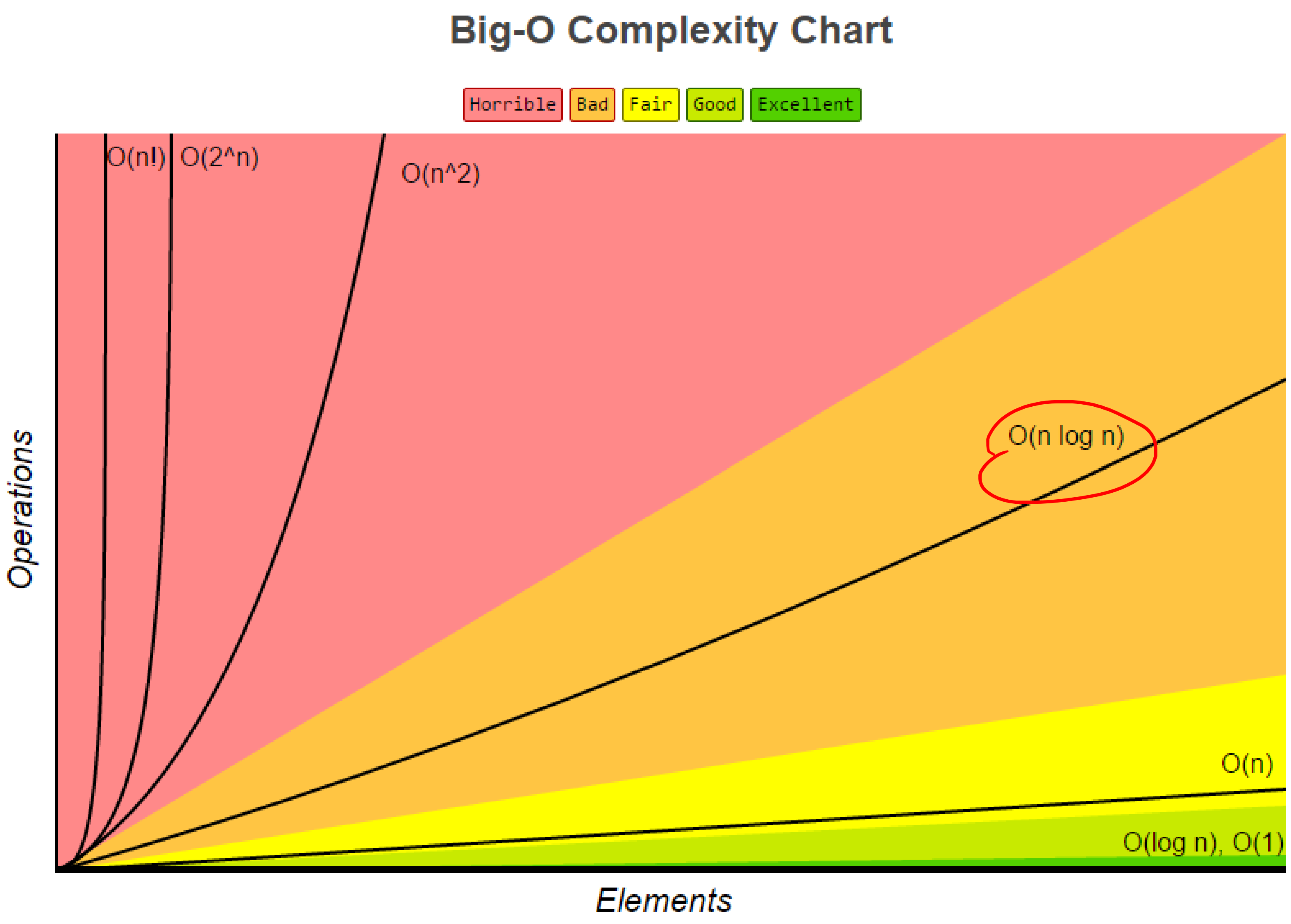


https://www.khanacademy.org/computing/computer-science/algorithms/asymptotic-notation/a/asymptotic-notation

Asymptotic Analysis

- By dropping the less significant terms and the constant coefficients, we can focus on the important part of an algorithm's running time, — its rate of growth. When we drop the constant coefficients and the less significant terms, we use **asymptotic notation**.

1. $\Theta(1)$
2. $\Theta(\lg n)$
3. $\Theta(n)$
4. $\Theta(n \lg n)$
5. $\Theta(n^2)$
6. $\Theta(n^2 \lg n)$
7. $\Theta(n^3)$
8. $\Theta(2^n)$



http://bigocheatsheet.com/

Amortized Analysis

Array → Resize

Amortized Analysis

- Amortized Analysis is used for algorithms where an occasional operation is very slow, but most of the other operations are faster.

$O(1)$

$O(n)$

Amortized time

1, 2, 3, 4, 5, 6, 7, 8, 9 → O(n)

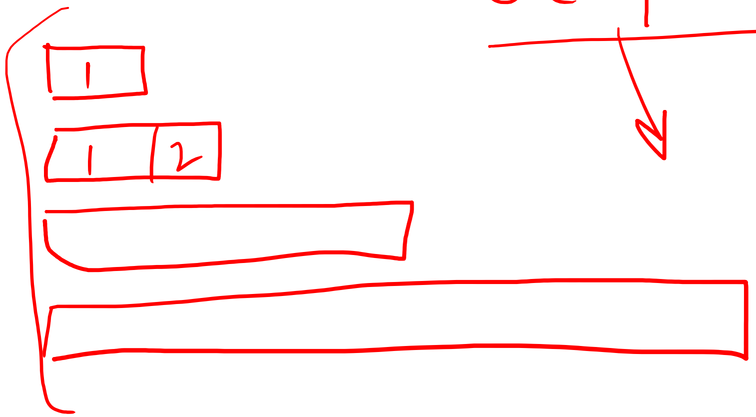
Dynamic → Resize O(1)

Proposition 5.1: *Let S be a sequence implemented by means of a dynamic array with initial capacity one, using the strategy of doubling the array size when full. The total time to perform a series of n append operations in S, starting from S being empty, is $O(n)$.*

3n

O(n)

O(n) → n

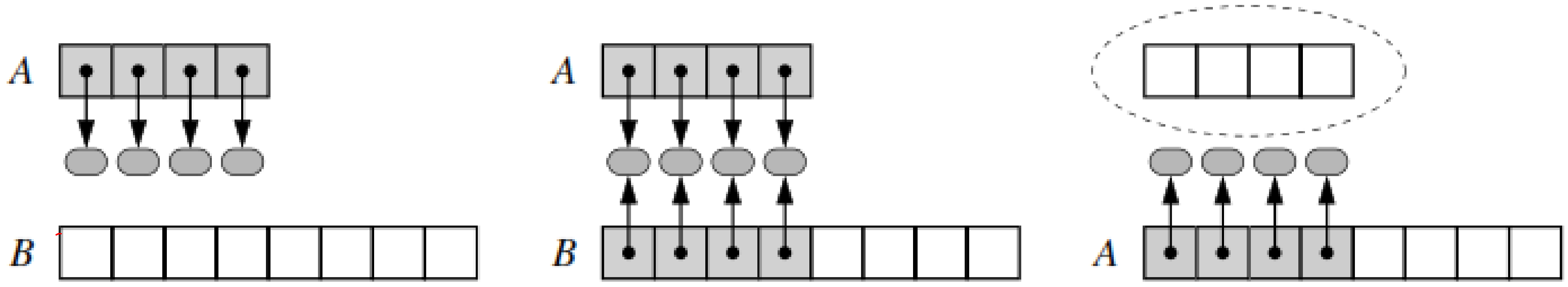


O(n)

Static → n → O(n)

Dynamic — n → O(n) — ?

Resizing an array



(a) (b) (c)

Aggregate Method

- Determine the worst-case cost of the entire sequence of operations, $T(n)$.
- Divide this cost by the number of operations in the sequence, n.

Dynamic $\rightarrow$ n $\rightarrow$ $T(n)/n$

$O(n)$

Aggregate Method

1
1

$O(n)$

'n'

$2^{\log_2 n} = ?$

n

1	2
1	1+1

$O(1)$ $O(n)$

$O(n)$

1	2	3	4
1	1+1	1+2	1

$2^{n-1} + 1$

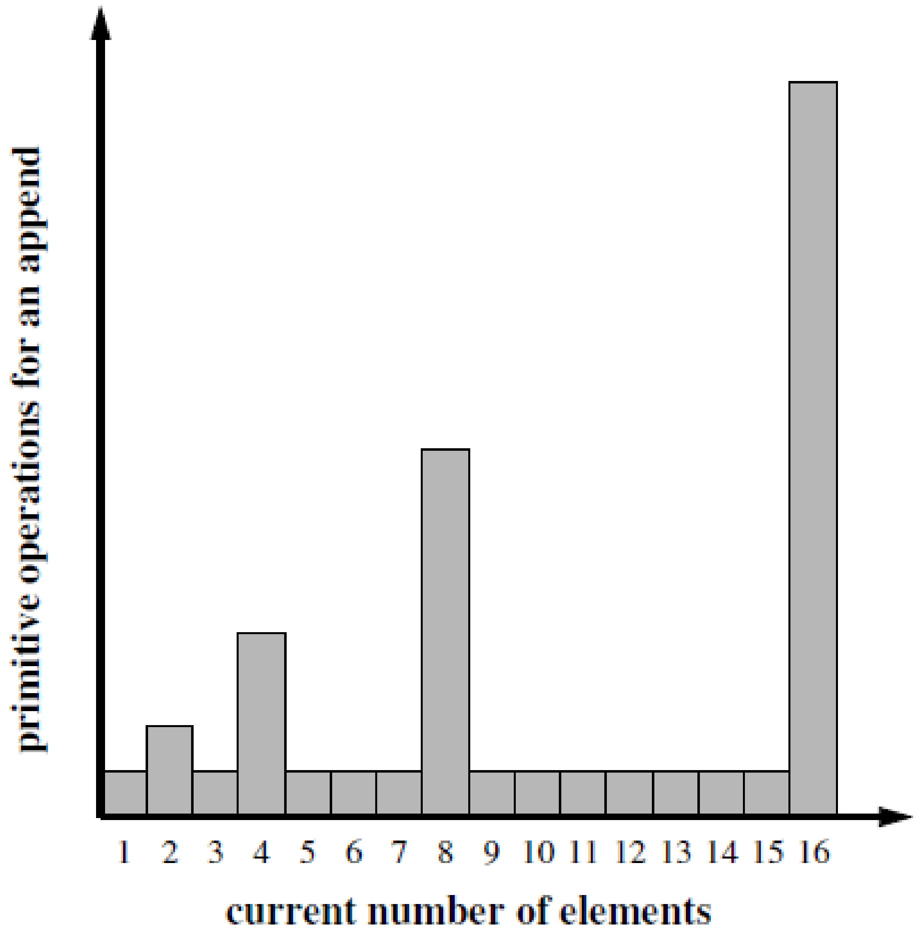
1	2	3	4	5	6	7	8	9	. - - - -
1	1+1	1+2	1	1+4	1	1	1	1+8	

1	2^0
1	
2	2^1
1	
4	2^2
1	
8	2^3
1	
16	2^4
1	
32	2^5

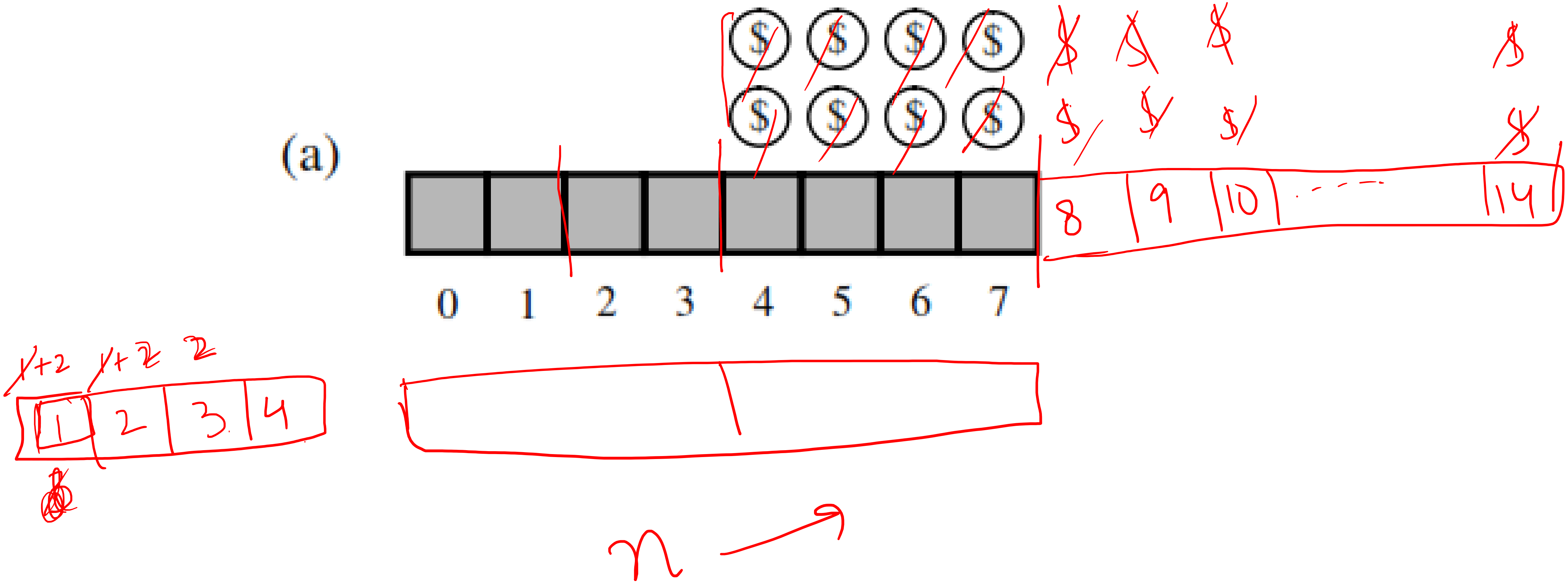
$n \to 2^{\log n}$

$$\underbrace{(1 + 1 + 1 + \cdots + 1 + \cdots)}_{n + 2^{\log n} - 1} + \underbrace{(2^0 + 2^1 + 2^2 + 2^3 + 2^4 + \cdots 2^{\log_2 n - 1})}_{n} = 2n \to O(n)$$

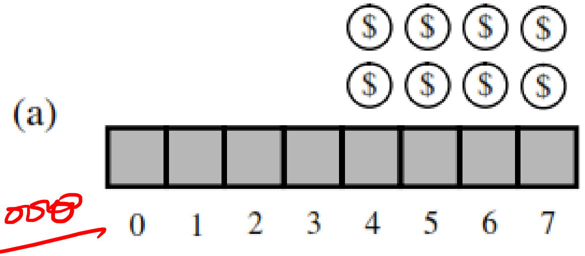
Doubling the array



Accounting Method



Accounting Method

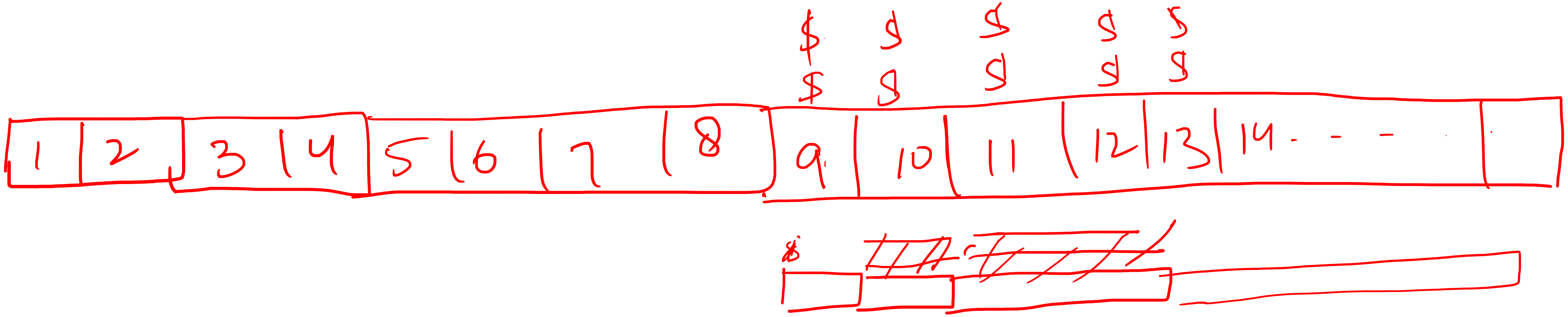


Jan 7000
Feb 7000
Mar 7000
June +36000
⋮
Dec 7000

7000
36000

Jan 10000 → 7000+3000
Feb 10000 → 7000+3000
Mar 10000 → 7000+3000
Apr 10000 → 7000+3000
⋮
Dec 10000 → 7000+3000

(10000)



append → 1 $O(1)$

→ 3

1+2

$O(1)$

n

$3n \to O(n)$

n

$3n$ → $O(n)$

Amortized time

Proposition 5.1: *Let S be a sequence implemented by means of a dynamic array with initial capacity one, using the strategy of doubling the array size when full. The total time to perform a series of n append operations in S, starting from S being empty, is $O(n)$.*

By a constt number:

By 1:

1	2	3	4	5	. . . . . .	n	n+1

1 2 3 4 5 . . . n

$n + (n-1) + (n-2) \ldots$

$= 1 + 2 + 3 + 4 + \ldots + n$

$= \frac{n(n+1)}{2} \longrightarrow O(n^2)$

$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$

K

By K:

$K + 2K + 3K + 4K + \ldots$

$K(1 + 2 + 3 + 4 + \ldots n) \longrightarrow O(n^2)$

for $1 - n$ $+5$

for $i - n$ $\cancel{*2}\ 3$

$1, 2^0, 2^1, 2^2 \ldots$

$O(n)$ $O(n)$

$O(\log n)$

Amortized Analysis

- Increasing the size by:
  - A constant factor
  - A constant number of cells

$K + 1.5K +$

$K + 2^0 K + 2^2 K \rightarrow 2^3 K \rightarrow$

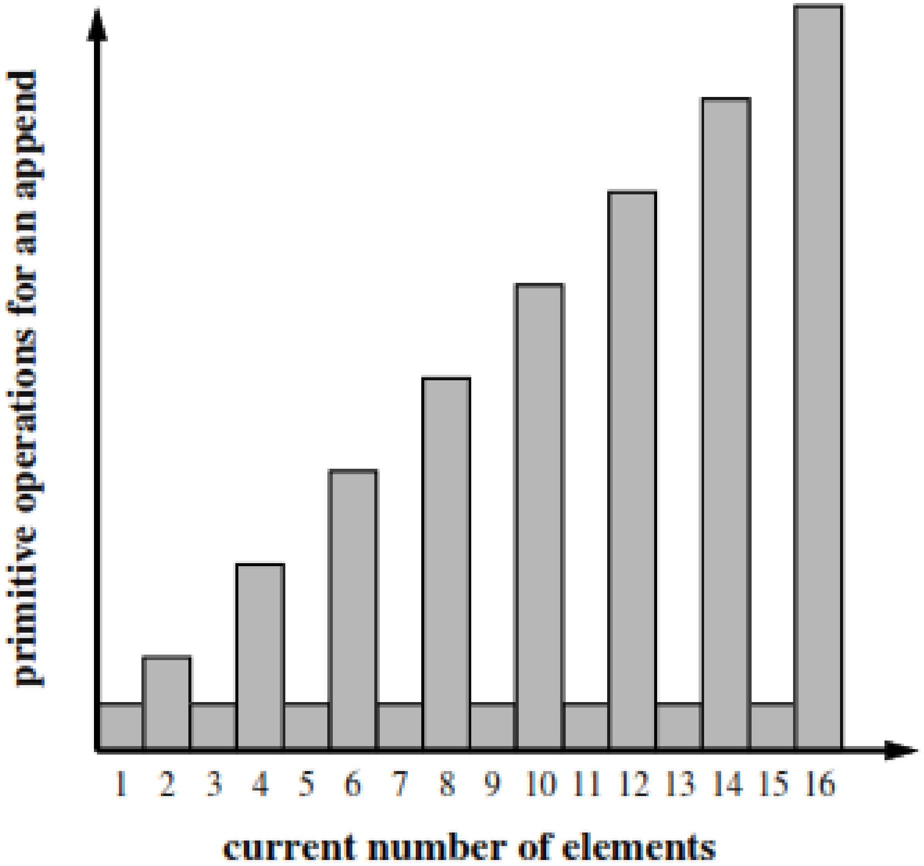
$O(n)$

$K + (K+5) + (K+10) + (K+15) + \cdots (\cdot)$

$10 \xrightarrow{\times 2} 20 \xrightarrow{\times 2} 40 \xrightarrow{\times 2} 80 \xrightarrow{\times 2} 160 \longrightarrow O(n)$

$10 \xrightarrow[5]{} 15 \xrightarrow[5]{} 20 \xrightarrow[5]{} 25 \xrightarrow[5]{} 30 \xrightarrow[5]{} \longrightarrow O(n^2)$

Increasing by a constant number



(a)

Increasing by a constant number

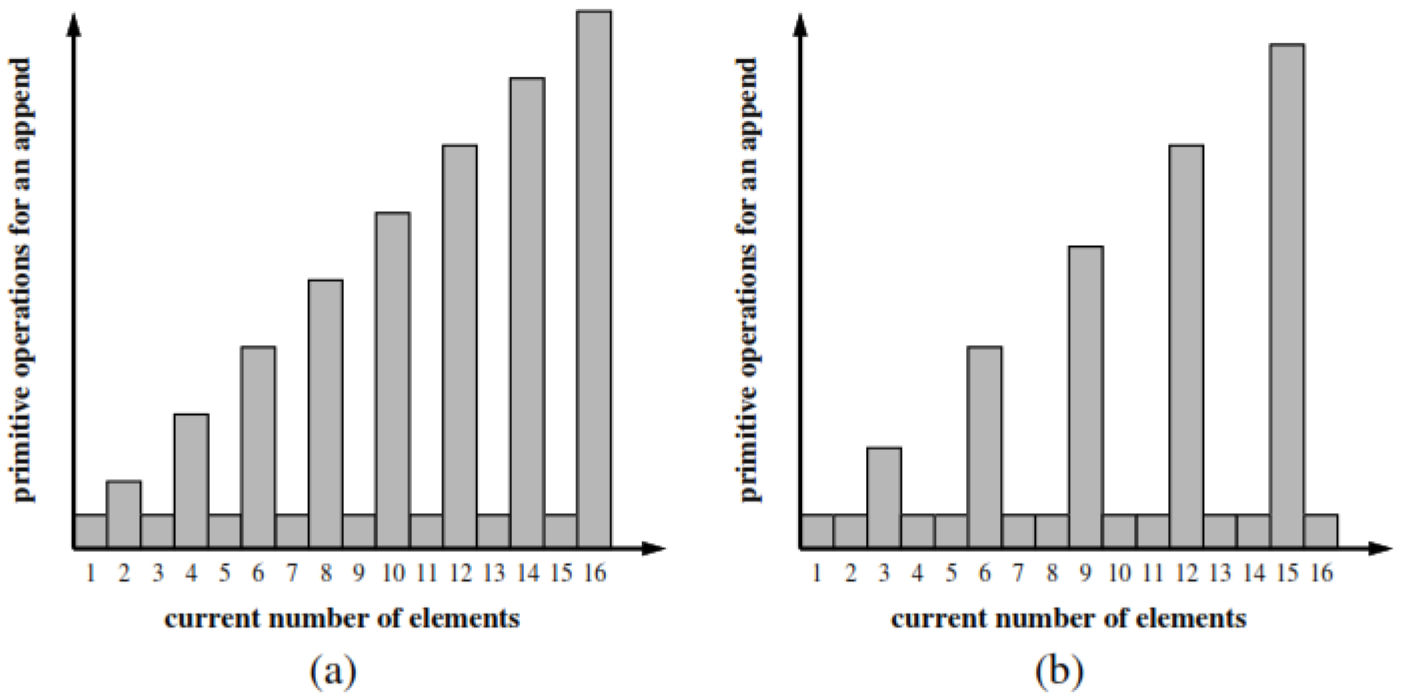


Figure 5.15: Running times of a series of append operations on a dynamic array using arithmetic progression of sizes. (a) Assumes increase of 2 in size of the array, while (b) assumes increase of 3.

Amortized time

Proposition 5.2: *Performing a series of n append operations on an initially empty dynamic array using a fixed increment with each resize takes $\Omega(n^2)$ time.*

Resources

- Open Data Structures (pseudocode edition), by Pat Morin. Available online at http://opendatastructures.org
- Data Structures and Algorithms in Python, by Michael T. Goodrich, Roberto Tamassia, and Michael H. Goldwasser. 2013. (1st. ed.). Wiley Publishing
- https://www.khanacademy.org/computing/computer-science/algorithms/asymptotic-notation/a/asymptotic-notation

Thanks