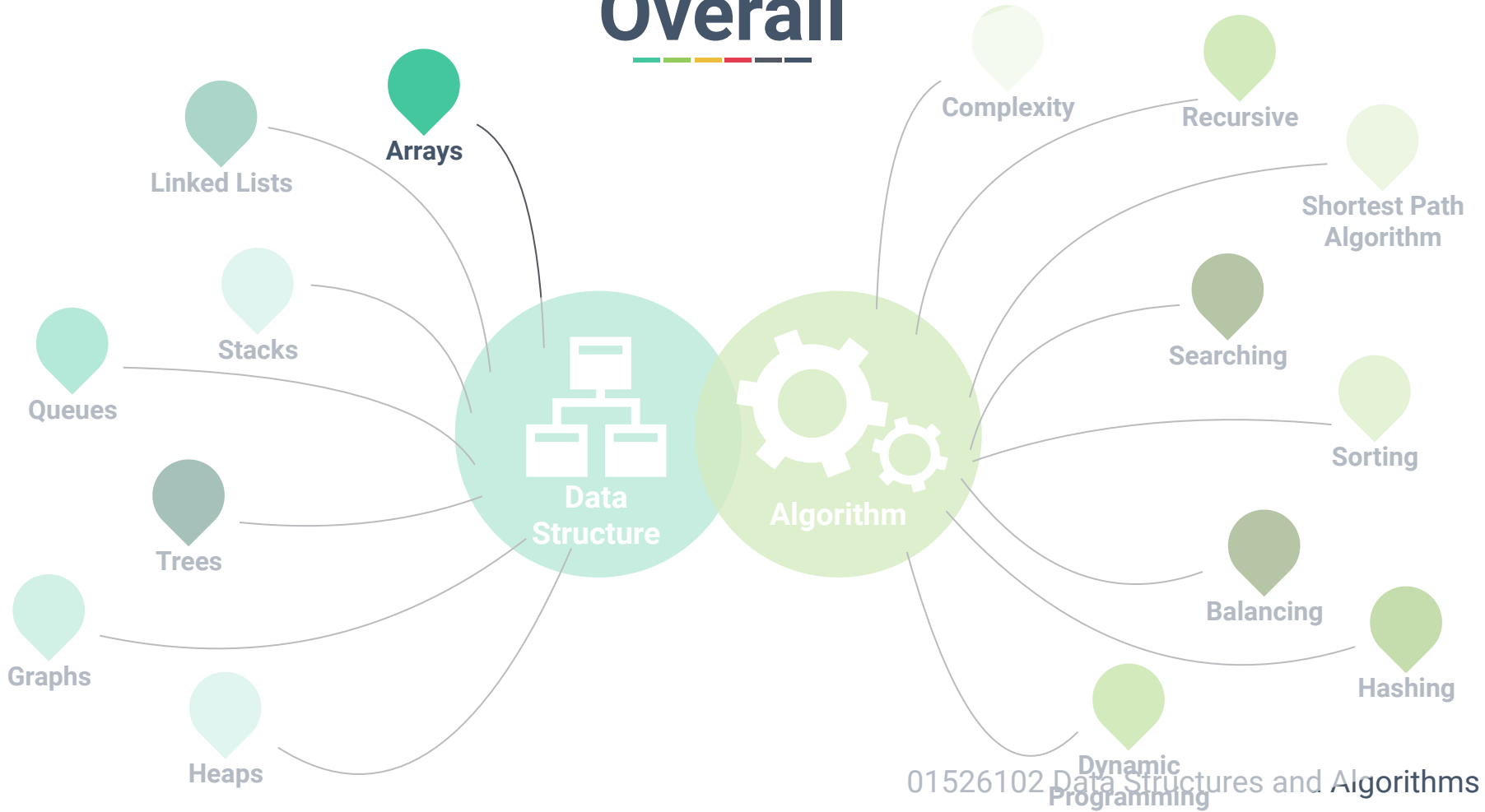


Chapter 2: Arrays



Dr. Sirasit Lochanachit

Overall





Today's Outline



1. What is an Array?
2. 2D Arrays
3. 3D Arrays
4. Array Operations and Asymptotic Performance



Python Sequence Types



- Built-in class
 - list
 - tuple
 - str



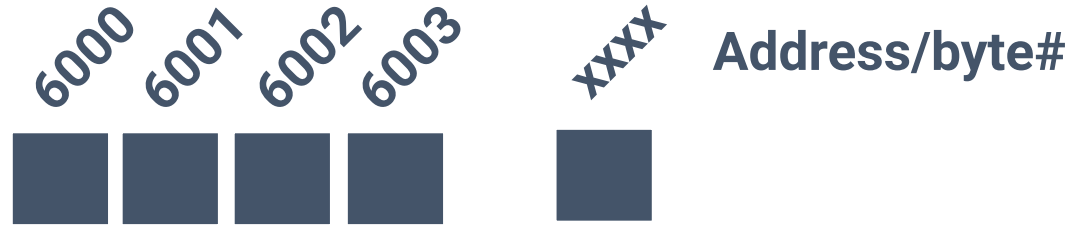
What is an Array?



- A computer will have a large number of bytes of memory.



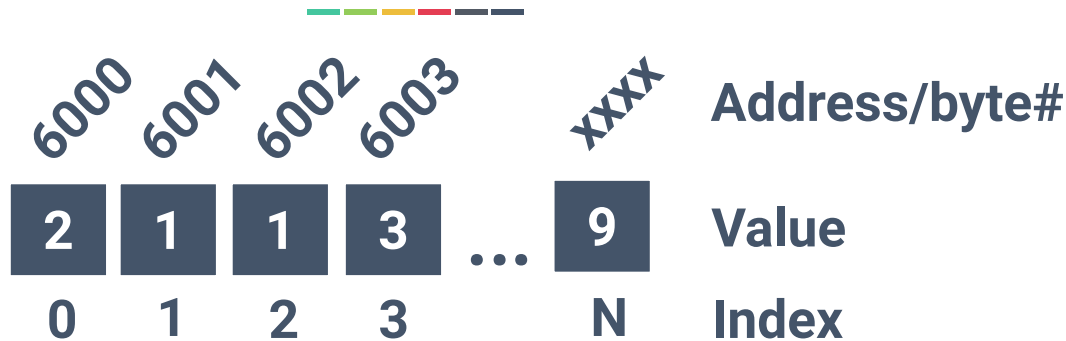
What is an Array?



- It has a memory address to keep track of where a data is stored.
 - Each byte has a unique number as its address.
- Although the number is sequential, any byte/element in a RAM can be accessed to read or write with a constant time $O(1)$.



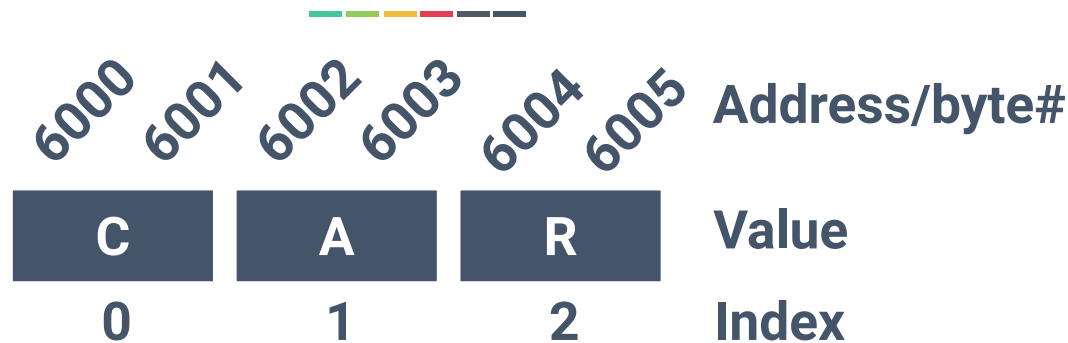
What is an Array?



- An array is a chunk of memory, consisting of equal-size elements.
- Each of those elements has an integer index, which uniquely refers to the value stored.
- The values are all of the same type (integer, character, etc.).



Array of Characters



- In Python, it represents a unicode character with 16 bits (i.e. 2 bytes).
- Since each cell has an equal-size bytes, any element can be accessed constantly with this formula:
 - $\text{start_address} + \text{cell_size} * \text{index}$



Exercise



Given an array:

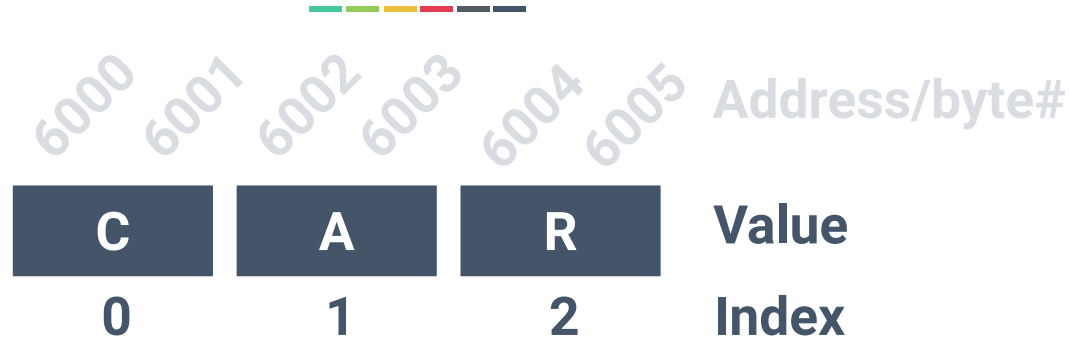
**Start Address is 6000,
cell size is 8,**

What is the address of the element at index 6?

- $\text{start_address} + \text{cell_size} * \text{index}$

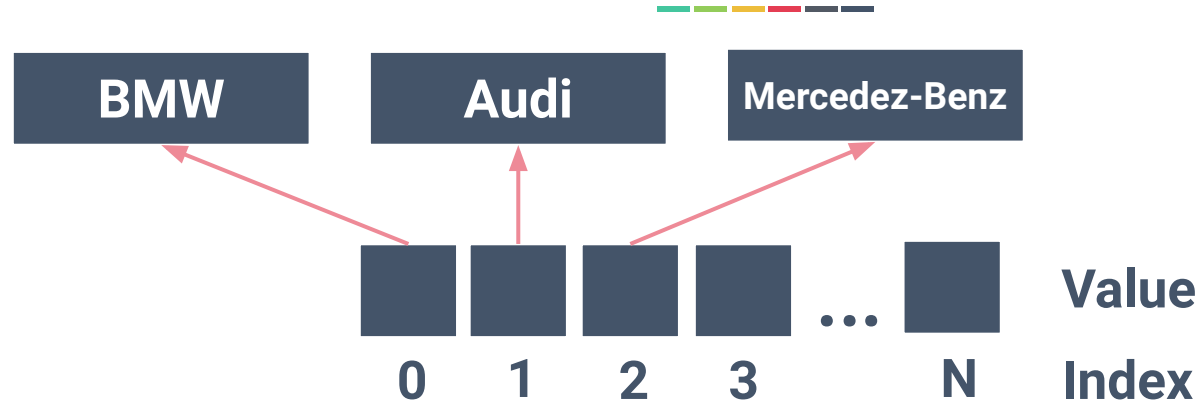


Array of Characters



- Luckily, a programming language calculate memory addresses of an array automatically.

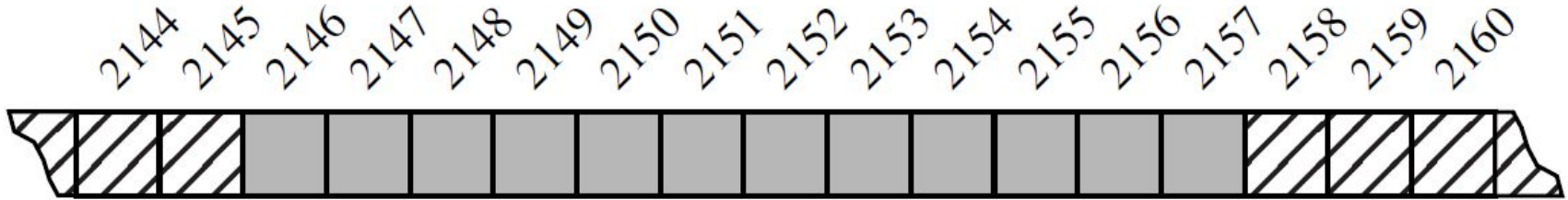
Python List



- Python list is a referential-type array that stores the memory addresses (references) of a value instead of the value itself.
- Strings can be in any length, but memory addresses are fixed-size.

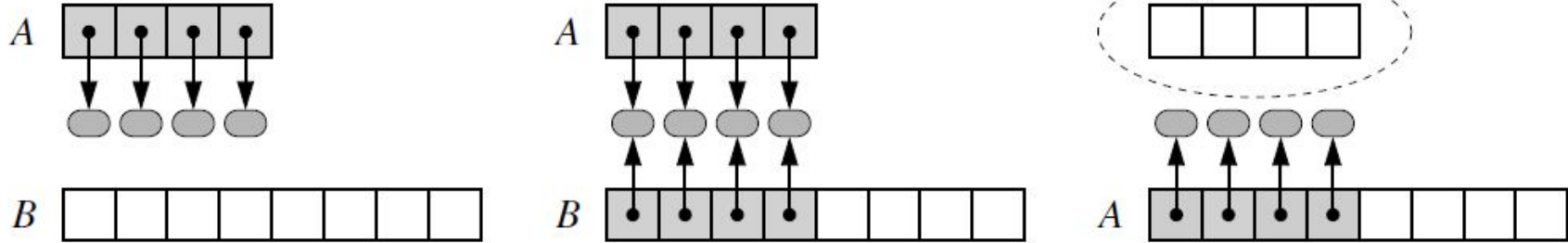


Static Array



- Once an array in C or Java has been created, it's size is fixed.
- Python tuple and str instances are immutable.
 - Unable to change size and value

Dynamic Array



- Python's list instance maintains an underlying array that often has greater capacity than the current length of the list.
- If a capacity is exhausted, the list class requests a new, larger array.



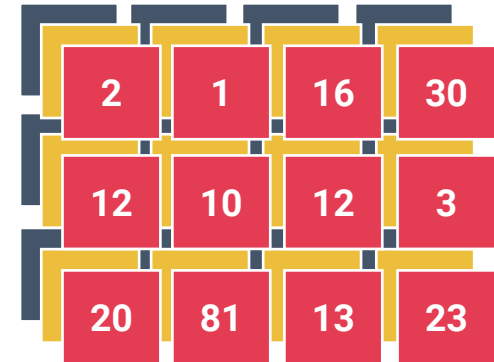
Arrays



(a) One dimension



(b) Two dimensions




(c) Three dimensions

2-Dimensional Arrays



(0,0)	(0,1)	(0,2)	(0,3)
(1,0)	(1,1)	(1,2)	(1,3)
(2,0)	(2,1)	(2,2)	(2,3)

2-Dimensional Arrays



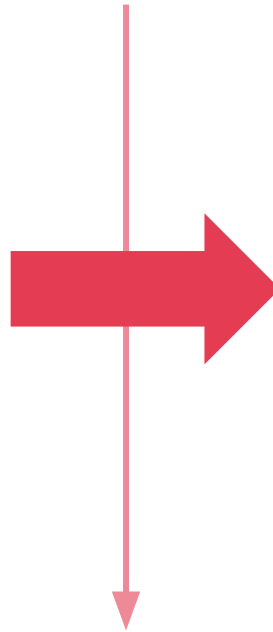
6000	6001	6002	6003
(0,0)	(0,1)	(0,2)	(0,3)
6004	6005	6006	6007
(1,0)	(1,1)	(1,2)	(1,3)
6008	6009	6010	6011
(2,0)	(2,1)	(2,2)	(2,3)

2-D Arrays



2	1	16	30
12	10	12	3
20	81	13	23

(a) Two dimensions



Rank

(0)

(1)

(2)

Size/length

(0) (1) (2) (3)

2	1	16	30
---	---	----	----

12	10	12	3
----	----	----	---

20	81	13	23
----	----	----	----

(b) Nested-one dimension

Example

Python Code: 2-D Arrays

```
c = np.array([[ 'h', 'e', 'l', 'l', 'o'], [ 'w', 'o', 'r', 'l', 'd']])
```

Rank	6000 6001	6002 6003	6004 6005	6006 6007	6008 6009
(0)	h	e	l	l	o
	(0,0)	(0,1)	(0,2)	(0,3)	(0,4)
	6010 6011	6012 6013	6014 6015	6016 6017	6018 6019
(1)	w	o	r	l	d
	(1,0)	(1,1)	(1,2)	(1,3)	(1,4)



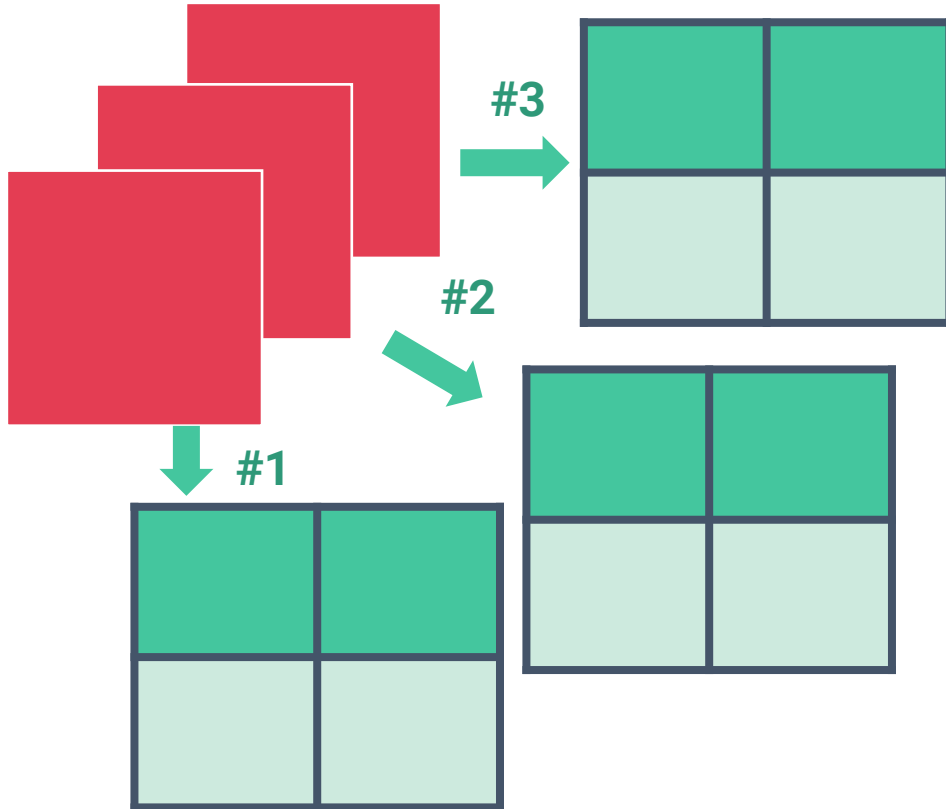
Exercise



Based on previous slide, suppose start address is 6000, find the address of index (1,4)

- $\text{start_address} + \text{cell_size} * \text{index}$
- Where $\text{index} = (\text{target_rank} * \text{array_length}) + \text{target_index}$

3D Arrays

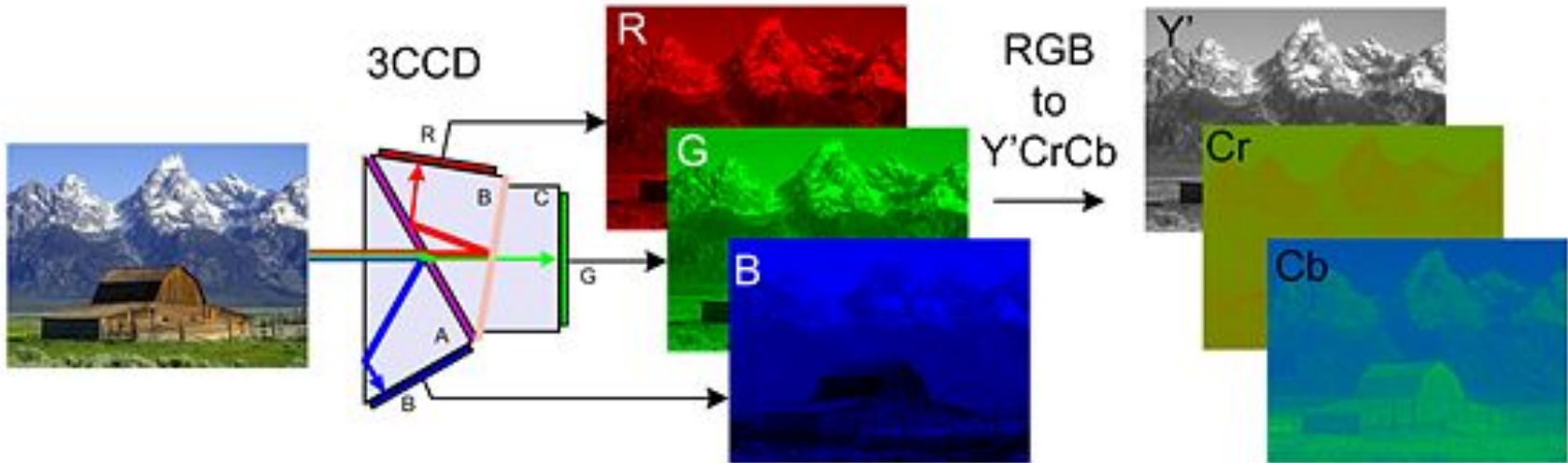


Python Code: 3D Arrays

```
e = np.array([  
    [[1,2,3],[4,5,6]],  
    [[7,8,9],[10,11,12]],  
    [[13,14,15],[16,17,18]]])
```

	Dimension
<code>[[1,2,3],[4,5,6]]</code>	#1
<code>[[7,8,9],[10,11,12]]</code>	#2
<code>[[13,14,15],[16,17,18]]</code>	#3

3D Arrays





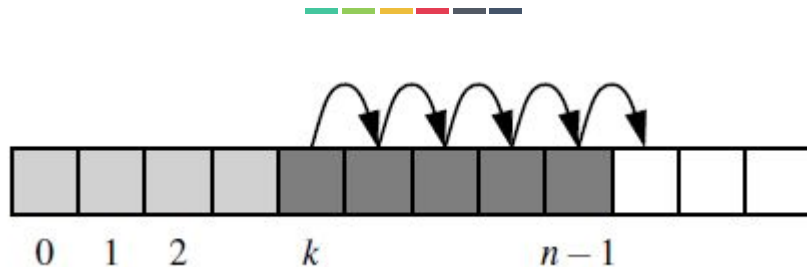
List Operations



Operation	Running Time
<code>len(data)</code>	$O(1)$
<code>data[i]</code>	$O(1)$
<code>data[i] = val</code>	$O(1)$
<code>c * data</code>	$O(n)$
<code>data.reverse()</code>	$O(n)$
<code>data.sort()</code>	$O(n \log n)$



Adding Elements to a List



- Operation **insert**(k , *value*) requires creating a room to insert a new element at index k of a dynamic array
 - Make room by shifting forward the $(n - 1) - k$ elements $data[k], \dots, data[n-1]$
 - $O(n - k + 1)$ for inserting at index k



Lab 3-1



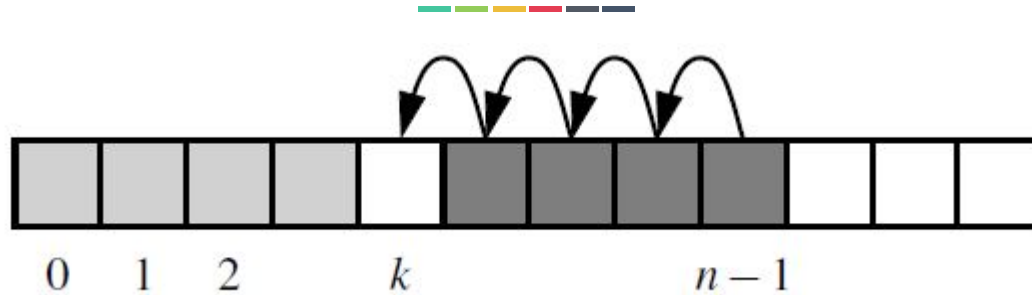
Record the average running time of `insert(k, 20)` in seconds with three different inserting patterns for each of the N calls:

1. Repeatedly insert at the beginning of a list
 2. Repeatedly insert near the middle of a list
 3. Repeatedly insert at the end of the list
- Each pattern starts from an empty list `[]` -> `data = []`

Lab 3-1

	N				
Patterns	10	20	50	100	200
First Case					
Second Case					
Third Case					

Removing Elements from a List



- Operation **pop**(k) removes the element that is at index $k < n$ of a list
 - Then shifting backward the $(n - 1) - (k + 1)$ elements $data[k+1], \dots, data[n-1]$
 - $O(n - k)$ for removing element at index k

Asymptotic Performance



	Add Operation	Running Time	Remove Operation	Running Time
Beginning				
End				
Between				

Data

10	5	8	7	1			
----	---	---	---	---	--	--	--

Asymptotic Performance



	Add Operation	Running Time	Remove Operation	Running Time
Beginning				
End	data.append(val)	$O(1)$		
Between				

Data

10	5	8	7	1	20		
----	---	---	---	---	----	--	--

Asymptotic Performance



	Add Operation	Running Time	Remove Operation	Running Time
Beginning				
End	data.append(val)	$O(1)$	data.pop()	$O(1)$
Between				

Data

10	5	8	7	1			
----	---	---	---	---	--	--	--

Asymptotic Performance



	Add Operation	Running Time	Remove Operation	Running Time
Beginning			data.pop(0) Del data[0]	$O(n)$
End	data.append(val)	$O(1)$	data.pop()	$O(1)$
Between				

Data

	5	8	7	1			
--	---	---	---	---	--	--	--

Asymptotic Performance



	Add Operation	Running Time	Remove Operation	Running Time
Beginning			data.pop(0) Del data[0]	$O(n)$
End	data.append(val)	$O(1)$	data.pop()	$O(1)$
Between				

Data

5	8	7	1				
---	---	---	---	--	--	--	--

Asymptotic Performance



	Add Operation	Running Time	Remove Operation	Running Time
Beginning	data.insert(0, val)	$O(n)$	data.pop(0) Del data[0]	$O(n)$
End	data.append(val)	$O(1)$	data.pop()	$O(1)$
Between				

Data

9	5	8	7	1			
---	---	---	---	---	--	--	--

Asymptotic Performance



	Add Operation	Running Time	Remove Operation	Running Time
Beginning	data.insert(0, val)	$O(n)$	data.pop(0) Del data[0]	$O(n)$
End	data.append(val)	$O(1)$	data.pop()	$O(1)$
Between			data.remove(val)	$O(n)$

Data

9	5		7	1			
---	---	--	---	---	--	--	--

Asymptotic Performance



	Add Operation	Running Time	Remove Operation	Running Time
Beginning	data.insert(0, val)	$O(n)$	data.pop(0) Del data[0]	$O(n)$
End	data.append(val)	$O(1)$	data.pop()	$O(1)$
Between			data.remove(val)	$O(n)$

Data

9	5	7	1				
---	---	---	---	--	--	--	--

Asymptotic Performance



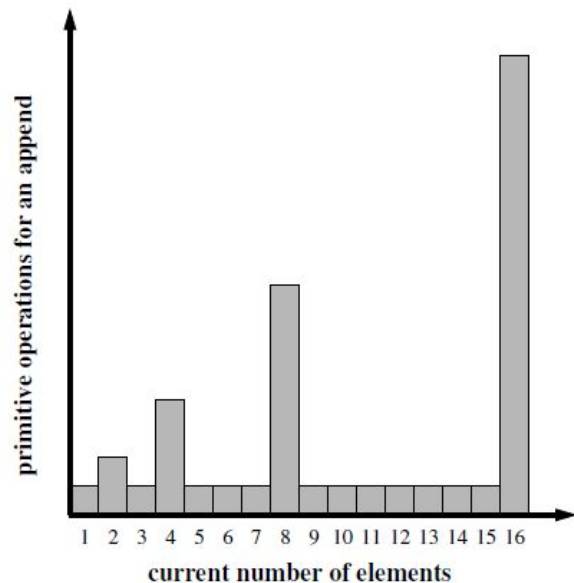
	Add Operation	Running Time	Remove Operation	Running Time
Beginning	data.insert(0, val)	$O(n)$	data.pop(0) Del data[0]	$O(n)$
End	data.append(val)	$O(1)$	data.pop()	$O(1)$
Between	data.insert(index, val)	$O(n)$	data.remove(val)	$O(n)$

9	0	5	7	1			
---	---	---	---	---	--	--	--



Amortization (Distribution) Analysis

- Operation **append**(*value*)
 - If an array is not full, copy the value into the array - $O(1)$
 - Otherwise, growing a dynamic array by copy old values to a new array - $O(n)$
 - Does this means **append**() has $O(n)$?





Amortization (Distribution) Analysis

- Let S be a dynamic array with initial capacity of size 1 and double the array when it hits a capacity (full)
 - An array is full when S has 2^i elements, for some integer $i \geq 0$
 - The size of the array S is 2^i
 - The cost for doubling the size is 2^i coins.
- For each append operation, 3 coins are charged as a cost.
 - One is for executing append operation
 - Two is for moving elements to a new array
- If an array is not full when append, save 2 coins

