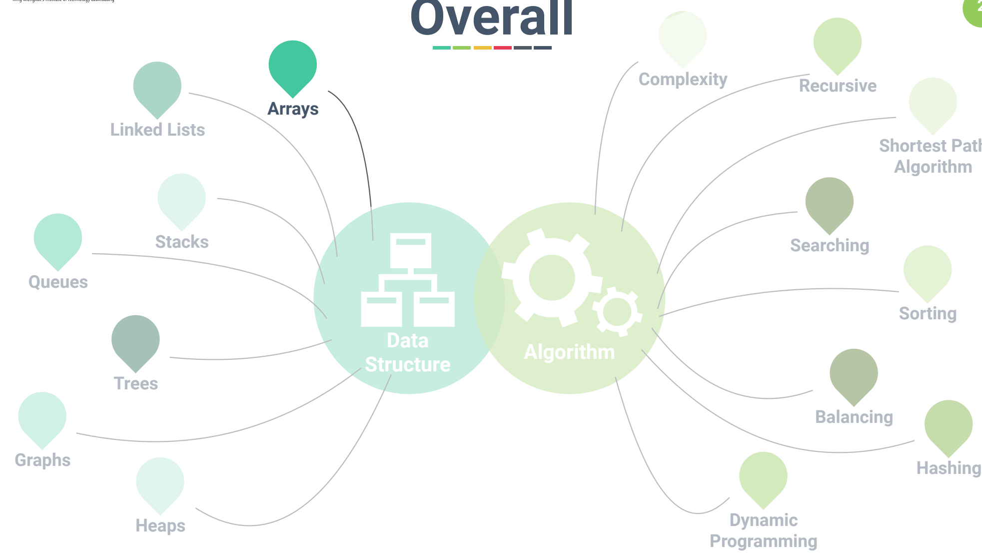
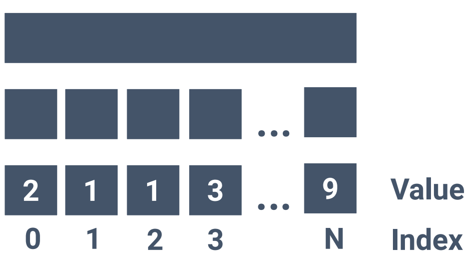
Chapter 2: Arrays

Dr. Sirasit Lochanachit

Overall

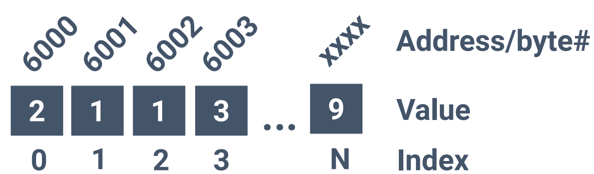


Arrays



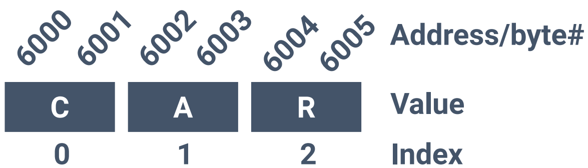
An array is a chunk of memory, consisting of equal-size elements.
Each of those elements have an integer index, which uniquely refers to the value stored. The values are all of the same type (integer, character, etc.).

Physical Level Arrays



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

Array of Characters



- In Python, it represents a unicode character with 16 bits (i.e. 2 bytes).
- Each element/cell in array is index with an integer starting with 0, 1, 2, and so on.
- Since each cell has an equal-size bytes, any element can be accessed constantly with this formula:
  - start_address + elem_size * index

Exercise

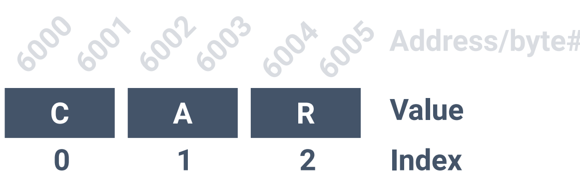
Given an array:

Start Address is 6000,
element size is 8,

What is the address of the element at index 6?

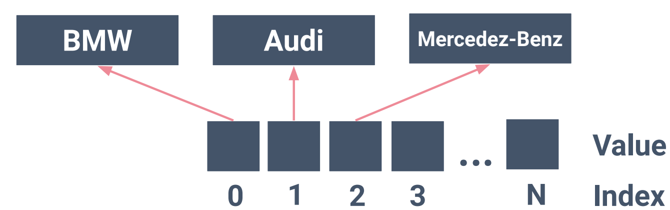
- start_address + elem_size * index

Array of Characters



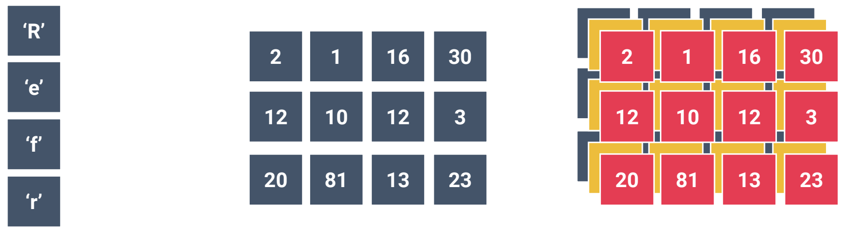
- Luckily, a programming language calculate memory addresses of an array automatically, so we can focus on values and indexes.

Python List



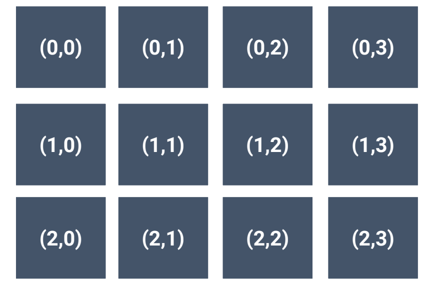
- Python list is a referential-type array that stores the memory addresses (references) of a value instead of the value itself.
- Strings can be in any length, but memory addresses are fixed-size.

Arrays

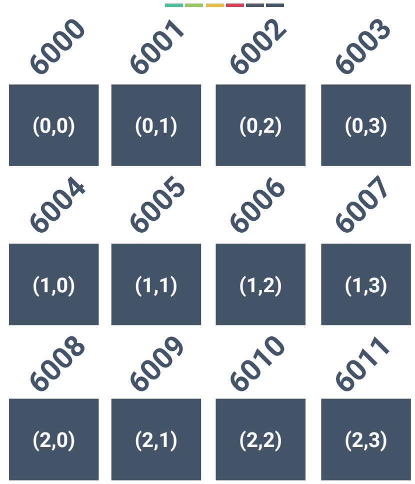


(a) One dimension (b) Two dimensions (c) Three dimensions

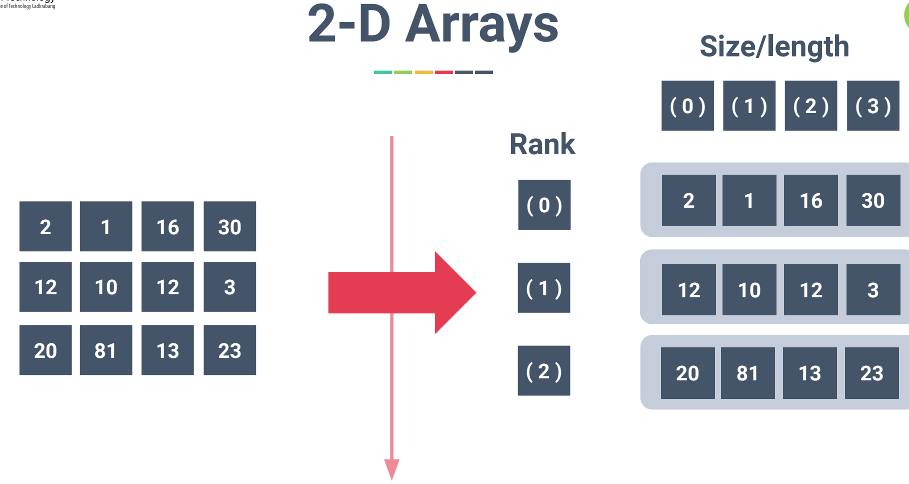
2-Dimensional Arrays



2-Dimensional Arrays



2-D Arrays



(a) Two dimensions (b) Nested-one dimension

Example

Python Code: 2-D Arrays

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

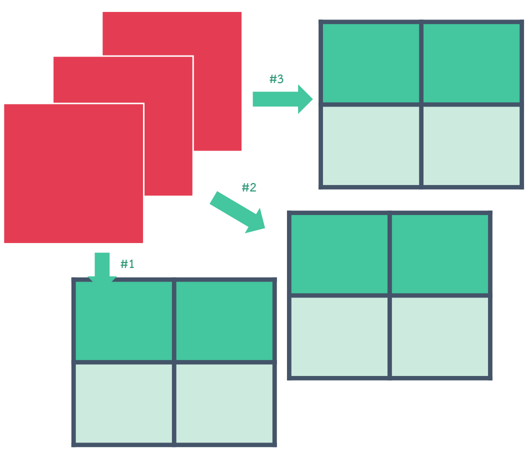
Rank	6000 6001	6002 6003	6004 6005	6006 6007	6008 6009
(0)	h	e	l	l	o
	(0,0)	(0,1)	(0,2)	(0,3)	(0,4)
	6010 6011	6012 6013	6014 6015	6016 6017	6018 6019
(1)	w	o	r	l	d
	(1,0)	(1,1)	(1,2)	(1,3)	(1,4)

Exercise

Suppose start address is 6000, find the address of index (1,4)

- start_address + elem_size * index
- Where index = (rank * array_length) + target_index

3-D Arrays

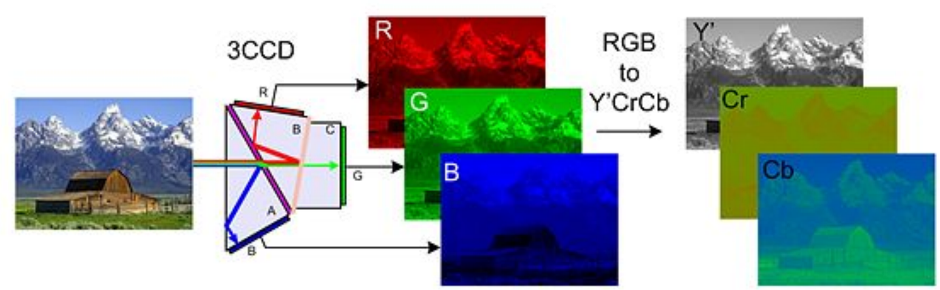


Python Code: 3-D Arrays

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

Dimension
#1
#2
#3

3-D Arrays



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)

Data	9	0	5	7	1			

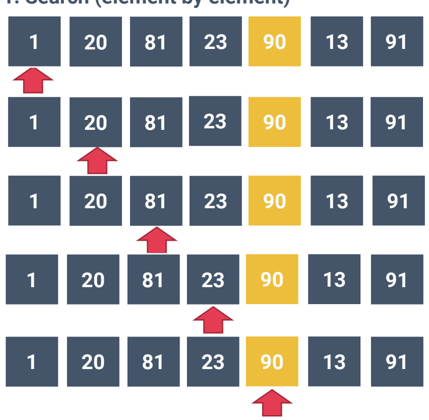
Asymptotic Performance

Operation	Running Time
len(data)	*O*(1)
data[i]	*O*(1)
Data[i] = val	*O*(1)
c * data	*O*(n)
data.reverse()	*O*(n)
data.sort()	*O*(n log n)

Searching in Array

Sequence

1. Search (element by element)



Binary

1. Sort

2. Search

Searching in Array

Sequence

1. Search (element by element)

Pseudocode: Sequential/linear search

```
linear_search (list, target_value)
  for each item in the list
    if item value == target_value
      return the item's location
    end if
  end for
  return 'no match'
END
```

Binary Search

- Binary Search
    - Locate a target value in a sequence of *n* elements that are sorted.
    - mid = (low + high) / 2
    - Initially, low = 0, high = n-1
- For instance, find number 5.

Data	1	5	7	9	10	11	20
Index	0	1	2	3	4	5	6

Binary Search

- Binary Search
    - If target value < data[mid], next interval is from low to mid-1.
    - If target value > data[mid], next interval is from mid + 1 to high.

	low			mid			high	
Data	1	5	7	9	10	11	20	mid = (0 + 6) / 2 = 3
Index	0	1	2	3	4	5	6	

	low	mid	high					
Data	1	5	7	9	10	11	20	mid = (0 + 2) / 2 = 1
Index	0	1	2	3	4	5	6	