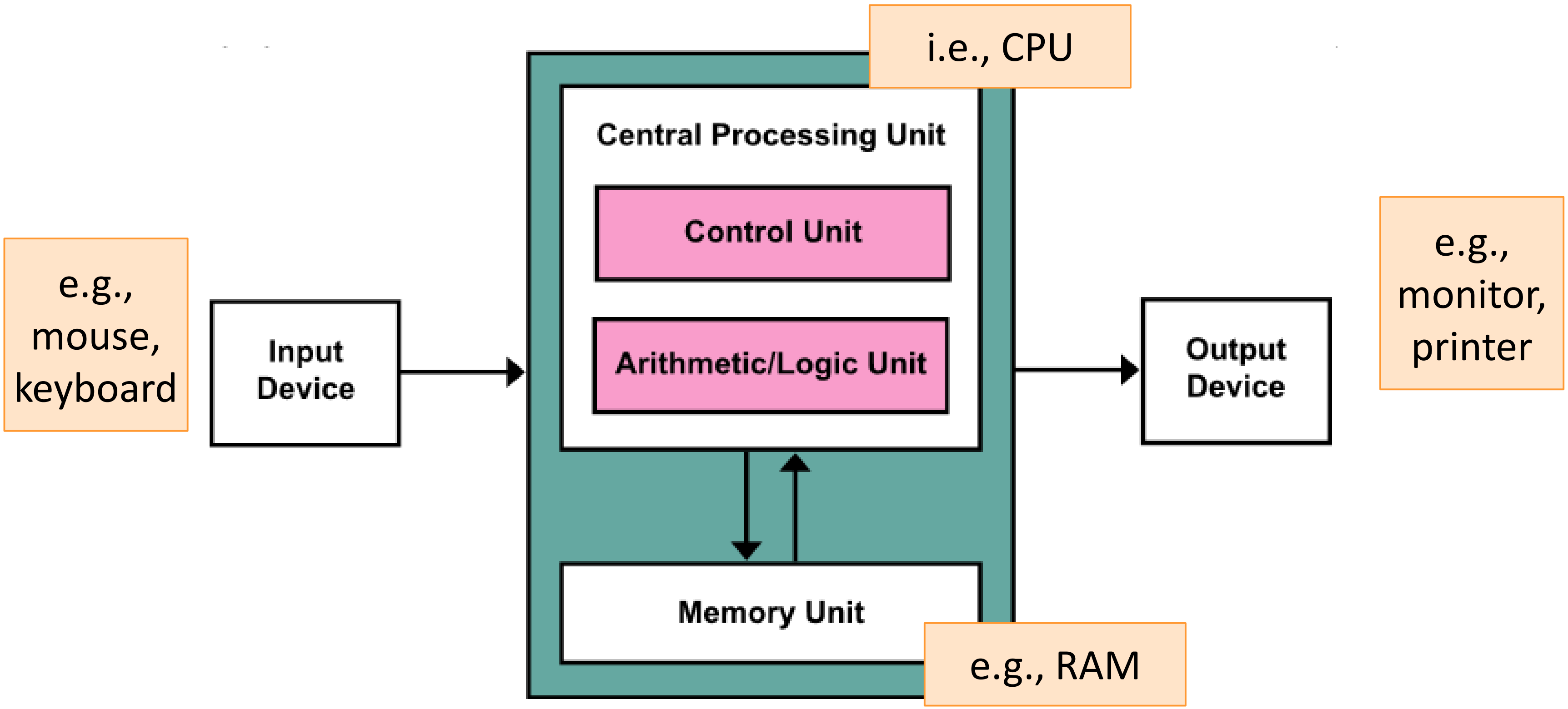
05
Array-based Sequences
Chapter 5

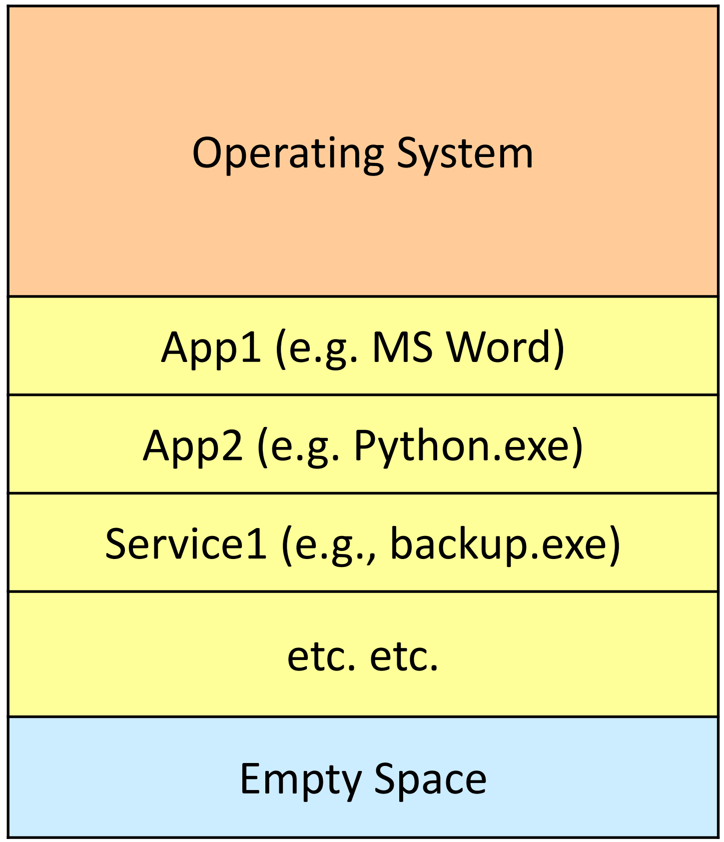
Low-level Arrays

von Neumann Architecture



Low-level Arrays

Inside the memory…



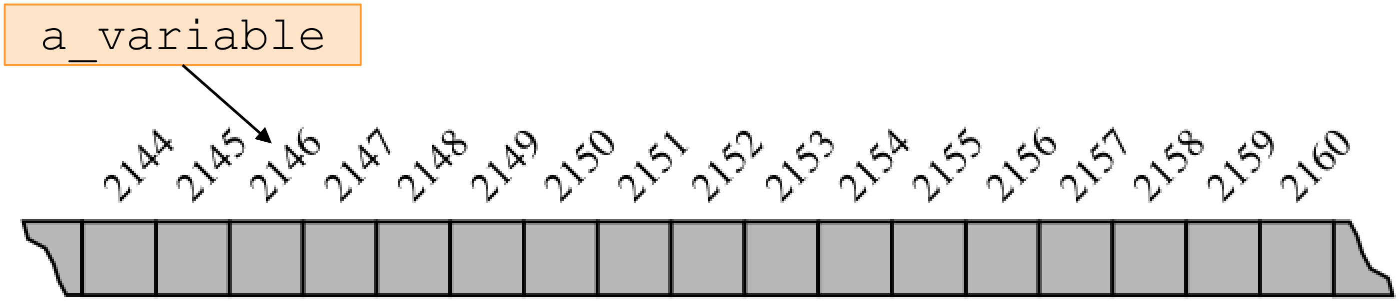
Low-level Arrays

Memory is a large contiguous array of **bytes** (4GB = 2^{32} bytes).

Each byte has a unique ID, a.k.a. memory address.

From theoretical point of view, any byte in the memory can be accessed within constant amount of time (O(1)).

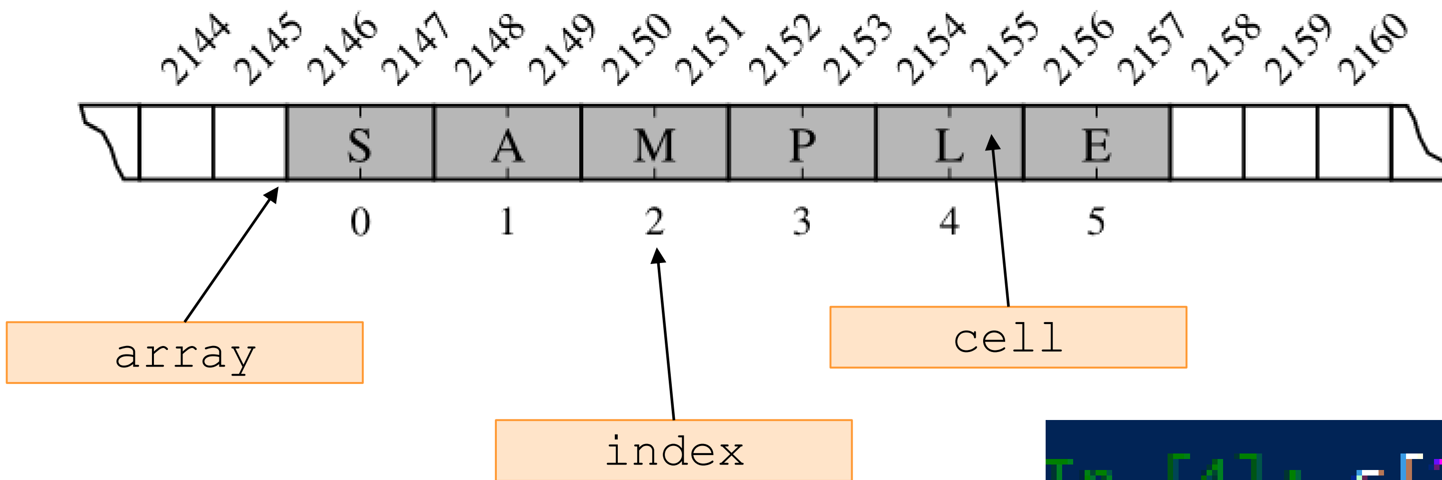
Low-level Arrays



Low-level Arrays

Array Concept

Representation of a series of values of the same type that contiguously reside in memory.



$\text{start} + \text{cellsize} * \text{index}$

Arrays in Python

Arrays exist in almost all programming languages.

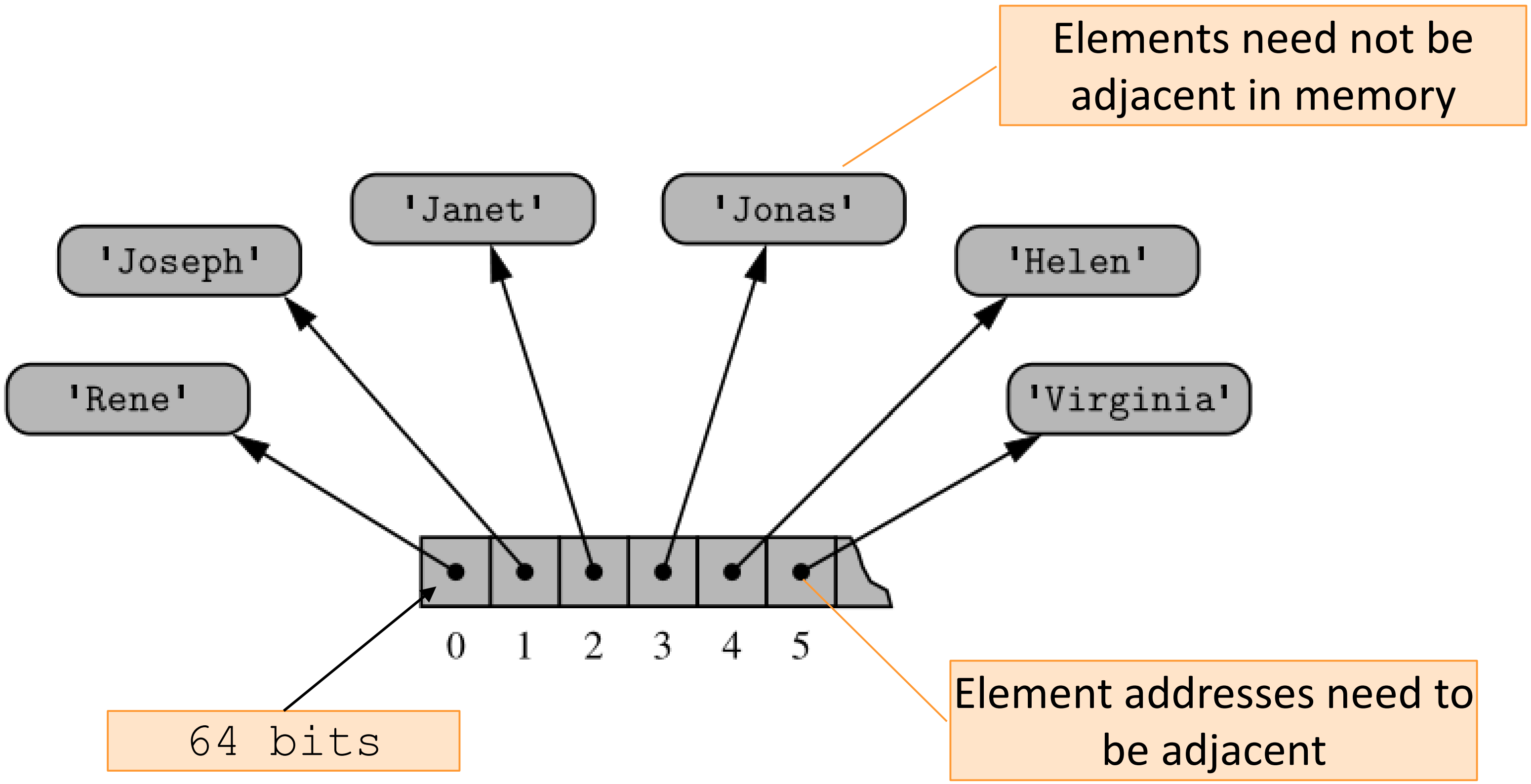
We will cover two implementation examples in Python:

- Referential Arrays
- Compact Arrays

Referential Arrays

In referential arrays, references to objects are stored.

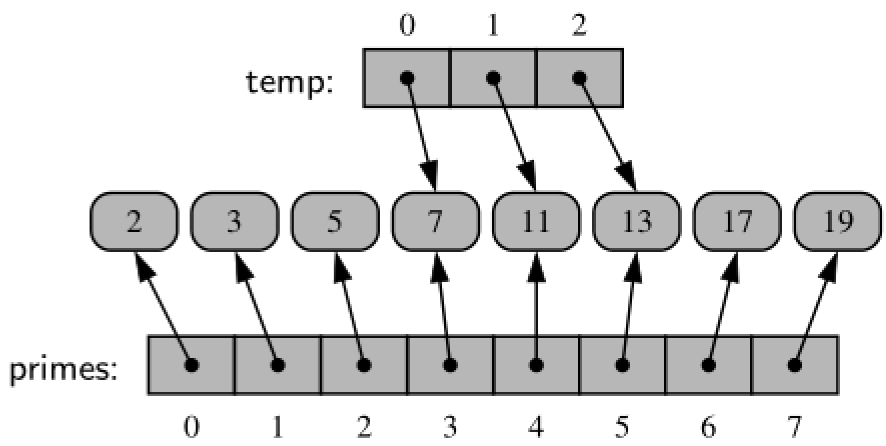
References can point to any type of object including `None`.



Referential Arrays

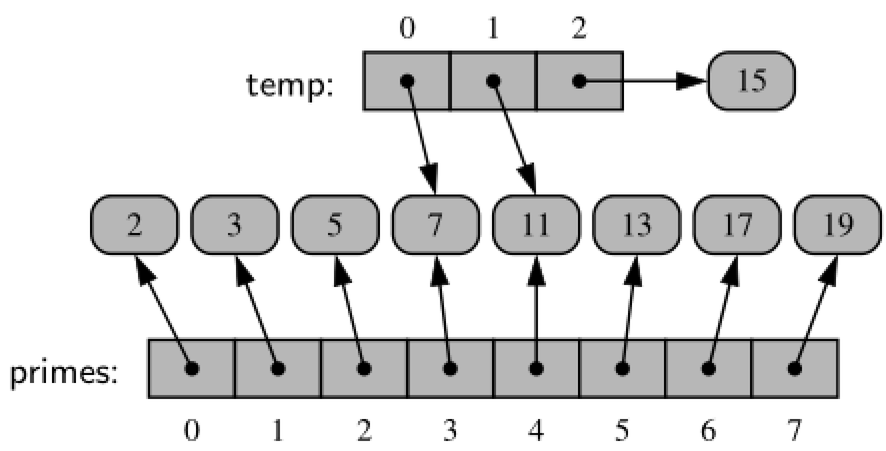
There may be multiple references to the same elements.

An element can belong to more than one array.




```
temp = primes[3:6]
```

Referential Arrays




```
temp[2] = 15
```

Referential Arrays

Shallow Copy

Will not be a problem when we are working with immutable types.

If array items are of mutable type, then we need to make a deep copy.

```
backup = list(primes)
```

creates a shallow copy

Referential Arrays

Shallow Copy

```
backup = list(primes)
```

creates a shallow copy

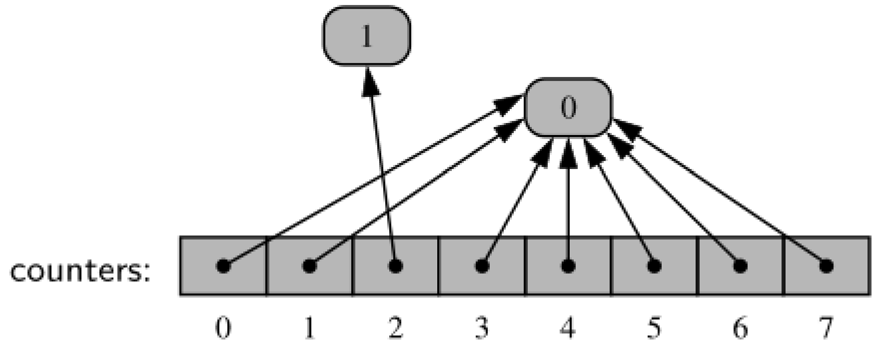
Immutable Case

```
In [32]: l1 = [1,2]

In [33]: l2 = list(l1)

In [34]: l1
Out[34]: [1, 2]

In [35]: l2
Out[35]: [1, 2]

In [36]: l2[0] = 999

In [37]: l1
Out[37]: [1, 2]

In [38]: l2
Out[38]: [999, 2]
```

Mutable Case

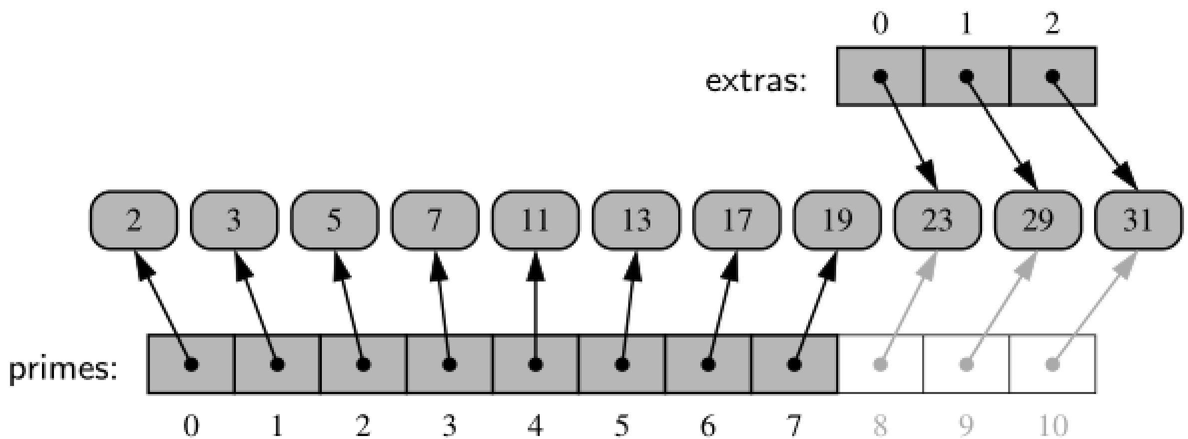
```
In [17]: l1 = [[1,2],[3,4]]

In [18]: l2 = list(l1)

In [19]: l1
Out[19]: [[1, 2], [3, 4]]

In [20]: l2
Out[20]: [[1, 2], [3, 4]]

In [21]: l1[0][0] = 99

In [22]: l1
Out[22]: [[99, 2], [3, 4]]

In [23]: l2
Out[23]: [[99, 2], [3, 4]]
```

Referential Arrays



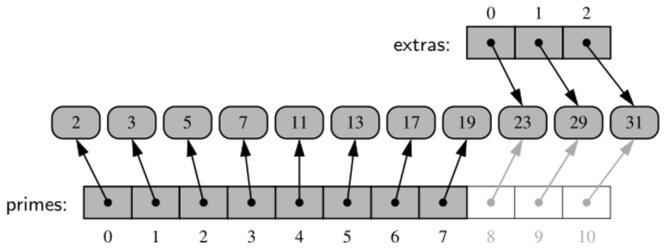

```
counters = [0] * 8
counters[2] += 1 # create a new int value and assign it to counter[2]
```

Referential Arrays




```
primes.extend(extras)
```

Referential Arrays

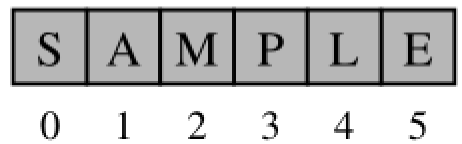



```
primes.extend(extras)
```

```
In [55]: l1 = [[1,2],[3,4]]

In [56]: l2 = [[5,6]]

In [57]: l1.extend(l2)

In [58]: l1
Out[58]: [[1, 2], [3, 4], [5, 6]]

In [59]: l2
Out[59]: [[5, 6]]

In [60]: l2[0][1] = 999

In [61]: l2
Out[61]: [[5, 999]]

In [62]: l1
Out[62]: [[1, 2], [3, 4], [5, 999]]
```

Compact Arrays

Arrays in which elements (rather than their references) are contiguously stored in memory.



Memory usage is much less compared to referential arrays (no need to use space for memory addresses).

```
In [14]: import sys

In [15]: r = [0,1,2,3,4]

In [16]: sys.getsizeof(r) + sys.getsizeof(0) + sys.getsizeof(1) + sys.getsizeof(2) + sys.getsizeof(3) + sys.getsizeof(4)
Out[16]: 232

In [17]: from array import array

In [18]: aa = array('i', [0,1,2,3,4])

In [19]: sys.getsizeof(aa)
Out[19]: 84
```

Compact Arrays

High-performance computing - data stored consecutively

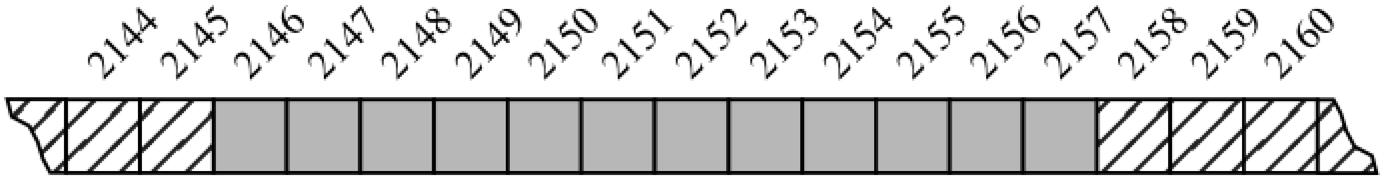
"Locality of reference"

Python's `array` module can be used to create compact arrays.

```
primes = array('i', [2, 3, 5, 7, 11, 13, 17, 19])
```

Code	C Data Type	Typical Number of Bytes
'b'	signed char	1
'B'	unsigned char	1
'u'	Unicode char	2 or 4
'h'	signed short int	2
'H'	unsigned short int	2
'i'	signed int	2 or 4
'I'	unsigned int	2 or 4
'l'	signed long int	4
'L'	unsigned long int	4
'f'	float	4
'd'	float	8

Dynamic Arrays



For immutable sequences (e.g., tuple, str) no expansion necessary.

For mutable sequences (e.g., list) arrays might need to be **re-**allocated.

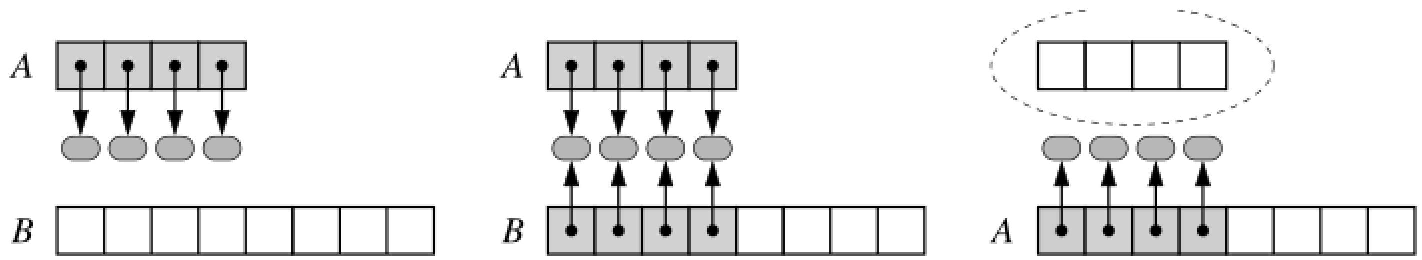
Python handles this automatically.

Example: list objects usually maintain a larger underlying array than needed. So, new add operations are instant.

```
In [37]: s = []

In [38]: sys.getsizeof(s)
Out[38]: 56

In [39]: s.append(1)

In [40]: sys.getsizeof(s)
Out[40]: 88

In [41]: s.append(1)

In [42]: sys.getsizeof(s)
Out[42]: 88

In [43]: s.append(1)

In [44]: sys.getsizeof(s)
Out[44]: 88
```

Nice reading: https://code.tutsplus.com/tutorials/understand-how-much-memory-your-python-objects-use--cms-25609

Dynamic Arrays



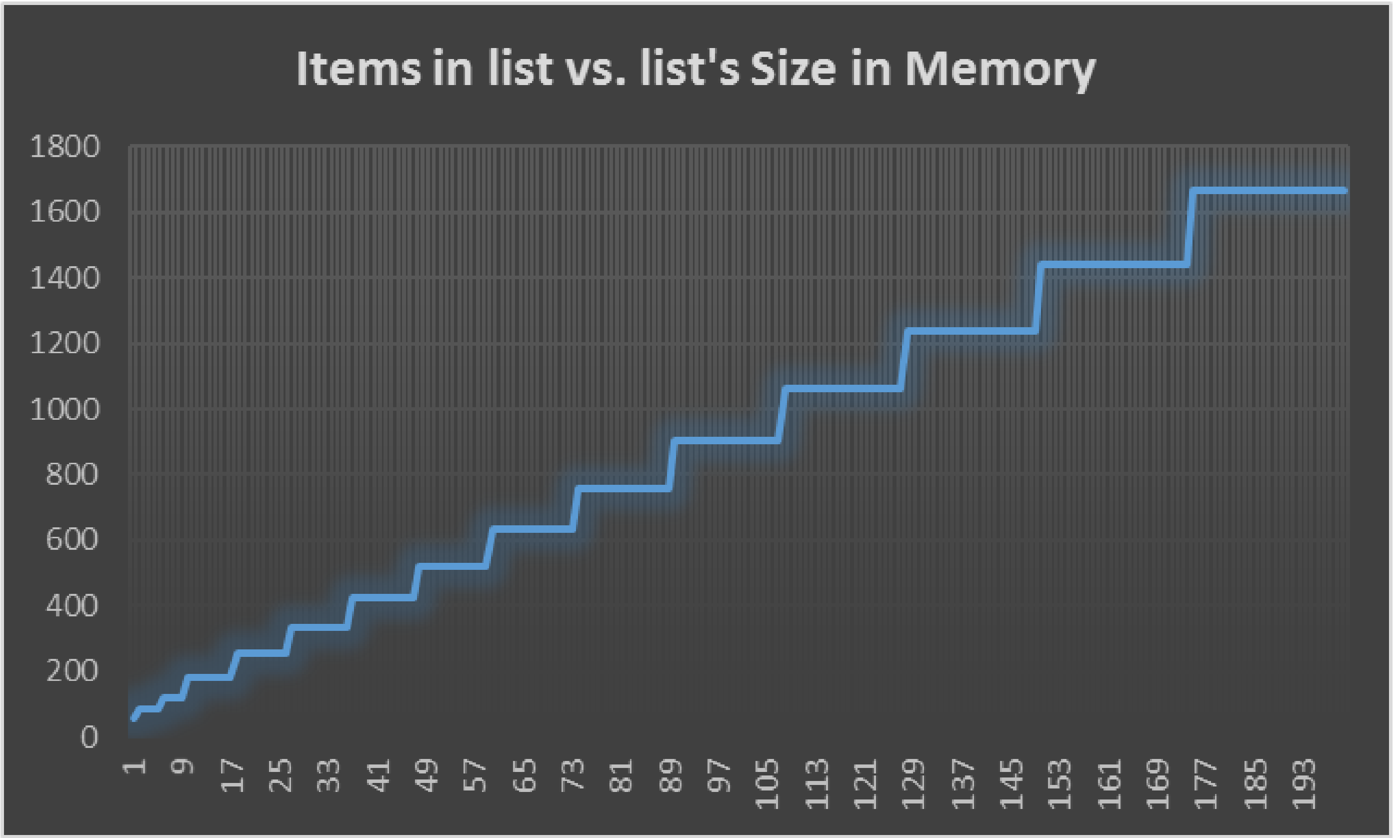
Expanding continues until capacity is exhausted.

list class makes a system call and requests a new larger array.

Items are carried from the original array to the newly created one.

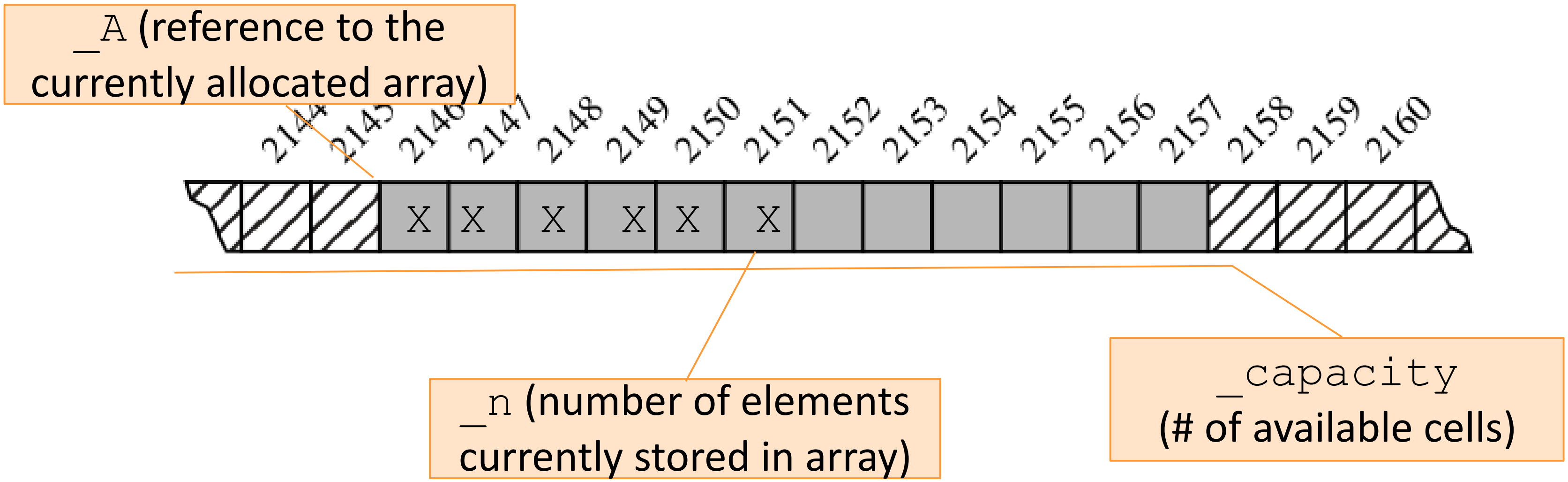
Original array's space is reclaimed by the system.

Dynamic Arrays



`empty list: 56 bytes, 1-4 elements 88 bytes, ...`

Dynamic Arrays



Dynamic Arrays

```
import ctypes                                        # provides low-level arrays

class DynamicArray:
  """A dynamic array class akin to a simplified Python list."""

  def __init__(self):
    """Create an empty array."""
    self._n = 0                                      # count actual elements
    self._capacity = 1                               # default array capacity
    self._A = self._make_array(self._capacity)       # low-level array

  def __len__(self):
    """Return number of elements stored in the array."""
    return self._n
```

~~return len(self._A)~~

Dynamic Arrays

```
def __getitem__(self, k):
  """Return element at index k."""
  if not 0 <= k < self._n:
    raise IndexError('invalid index')
  return self._A[k]                          # retrieve from array

def append(self, obj):
  """Add object to end of the array."""
  if self._n == self._capacity:              # not enough room
    self._resize(2 * self._capacity)         # so double capacity
  self._A[self._n] = obj
  self._n += 1
```

Dynamic Arrays

```
def _resize(self, c):                              # nonpublic utitity
  """Resize internal array to capacity c."""
  B = self._make_array(c)                          # new (bigger) array
  for k in range(self._n):                         # for each existing value
    B[k] = self._A[k]
  self._A = B                                      # use the bigger array
  self._capacity = c

def _make_array(self, c):                          # nonpublic utitity
   """Return new array with capacity c."""
   return (c * ctypes.py_object)( )                # see ctypes documentation
```

Dynamic Arrays

```
def append(self, obj):
  """Add object to end of the array."""
  if self._n == self._capacity:                 # not enough room
    self._resize(2 * self._capacity)            # so double capacity
  self._A[self._n] = obj
  self._n += 1
```

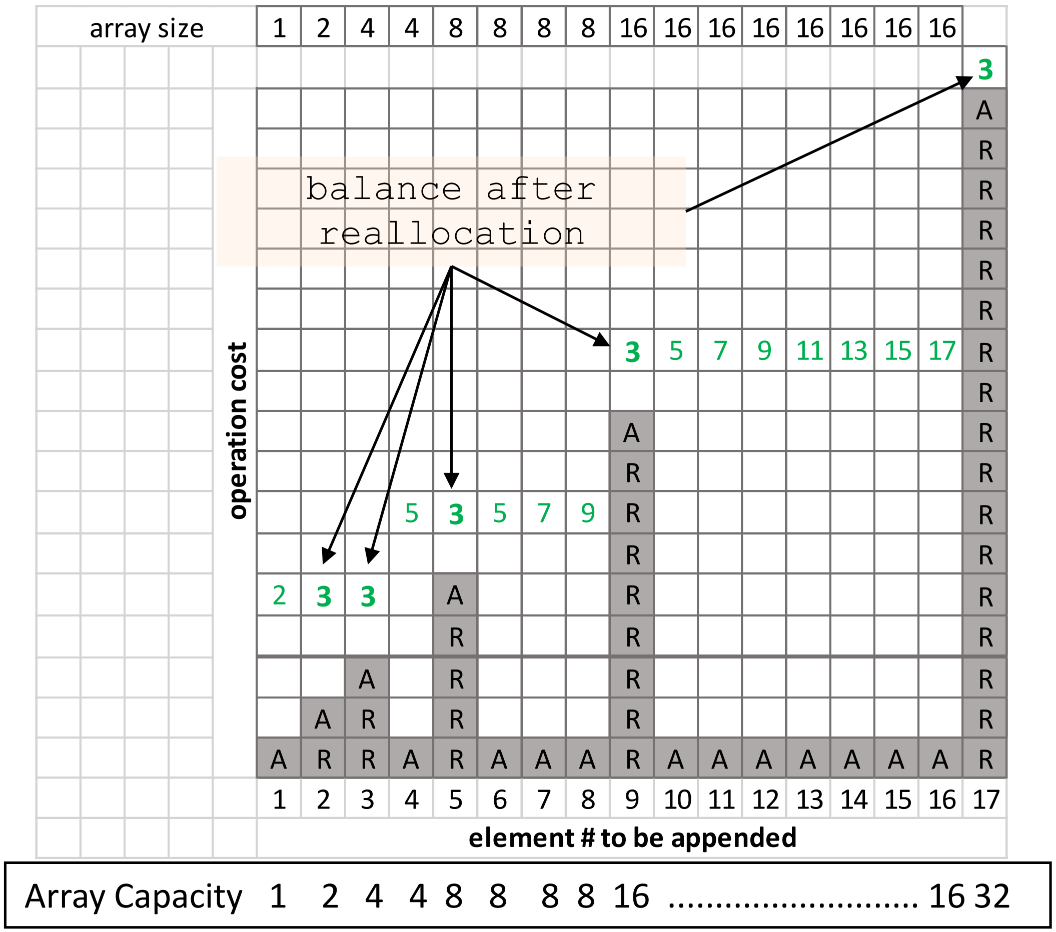
If no need to increase the capacity:

Append operation takes constant time

Else:

Append operation takes time that is proportional to the number of items (n) in the array.

Dynamic Arrays



A: Append Operation

R: Reallocation

We charge 3 coins for each append operation, regardless of the need for reallocation.

With constant charge, we can maintain this scheme:

3 items: 3*3 -3 coins

5 items: 5*3 -3 coins

9 items: 9*3 -3 coins

n items: n*3 -3 coins (O(n))

Efficiency of Sequence Types

The behaviors of list and tuples are two folds:

- Nonmutating behaviors (applies to both list and tuples)
- Mutating behaviors (applies to lists only)

Efficiency of Sequence Types

Nonmutating Behaviors

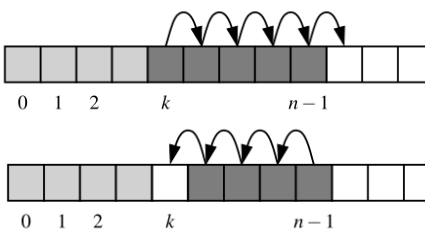
Let **n** be the size of the list/tuple, and **k** denote the index of the leftmost occurrence of a value in the list/tuple.

Operation	Running Time
len(data)	
data[j]	
data.count(value)	
data.index(value)	
value **in** data	
data1 == data2 (similarly !=, <, <=, >, >=)	
data[j:k]	
data1 + data2	
c * data	

Efficiency of Sequence Types

Mutating Behaviors

Let **n** be the size of the list, and **k** denote the index of the leftmost occurrence of a value in the list.



Operation	Running Time
data[j] = val	
data.append(value)	
data.insert(k, value)	
data.pop()	
data.pop(k) **del** data[k]	
data.remove(value)	
data1.extend(data2) data1 += data2	
data.reverse()	
data.sort()	

```
In [50]: data = [1,2,3,4]

In [51]: data.append(5)

In [52]: data
Out[52]: [1, 2, 3, 4, 5]

In [53]: data.insert(3, 3.5)

In [54]: data
Out[54]: [1, 2, 3, 3.5, 4, 5]

In [55]: data.pop()
Out[55]: 5

In [56]: data
Out[56]: [1, 2, 3, 3.5, 4]

In [57]: data.pop(3)
Out[57]: 3.5

In [58]: data
Out[58]: [1, 2, 3, 4]

In [59]: data += data

In [60]: data
Out[60]: [1, 2, 3, 4, 1, 2, 3, 4]

In [61]: data.reverse()

In [62]: data
Out[62]: [4, 3, 2, 1, 4, 3, 2, 1]
```

Efficiency of Sequence Types

String Functions

Considering a string **s** of length **n** and a pattern string **p** of length **m**, complexity of functions:

```
s.capitalize()
s.islower()
s.__contains__(p)
s.find(p)
s.count(p)
s.replace(p1,p2)
```

```
In [43]: s = 'It has TO wOrK'

In [44]: s.capitalize()
Out[44]: 'It has to work'

In [45]: s.islower()
Out[45]: False

In [46]: 'to' in s
Out[46]: False

In [47]: s.find('TO')
Out[47]: 7

In [48]: s.count('t')
Out[48]: 1

In [49]: s.replace('wOrK', 'play')
Out[49]: 'It has TO play'
```

Efficiency of Sequence Types

Notable Example on Strings

Let's suppose we need to process a large string such that we will create a new string that contains alphabetical characters only.

```
Input: '1 plus 2 makes 3'
Output: 'plusmakes'

letters = ''
for c in a_long_string:
    if c.isalpha():
        letters += c
```

```
letters = ''
for c in '1 plus 2 makes 3':
    if c.isalpha():
        letters += c
        # print(letters)
```

```
p  1
pl  2
plu  3
plus  .
plusm  .
plusma
plusmak
plusmake  .                    O(n^2)
plusmakes  n
(1+2+..+n)--->cost of n string recreation
```

What is the complexity?

How many string concatenations?

What's the complexity of concatenation?

Efficiency of Sequence Types

Notable Example on Strings

Let's suppose we need to process a large string such that we will create a new string that contains alphabetical characters only.

A solution with O(n) complexity: Store characters in a list (O(n)) and then merge them (O(n)).

```
temp = [ ]
for c in document:
    if c.isalpha():
        temp.append(c)

letters = ''.join(temp)
```

n * O(1) = O(n)

O(n)

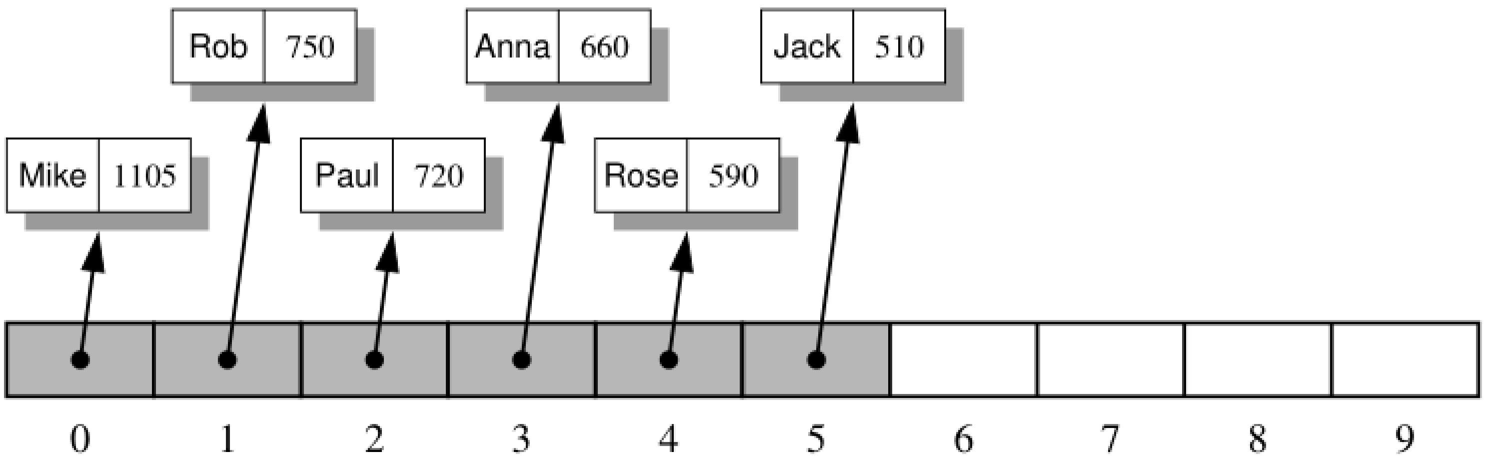
O(n) + O(n) = O(n)

Example-1 Scoreboard

Example-1 Scoreboard

```python
class GameEntry:
  """Represents one entry of a list of high scores."""

  def __init__(self, name, score):
    self._name = name
    self._score = score

  def get_name(self):
    return self._name

  def get_score(self):
    return self._score

  def __str__(self):
    return '({0}, {1})'.format(self._name, self._score) # e.g., '(Bob, 98)'
```

Example-1 Scoreboard

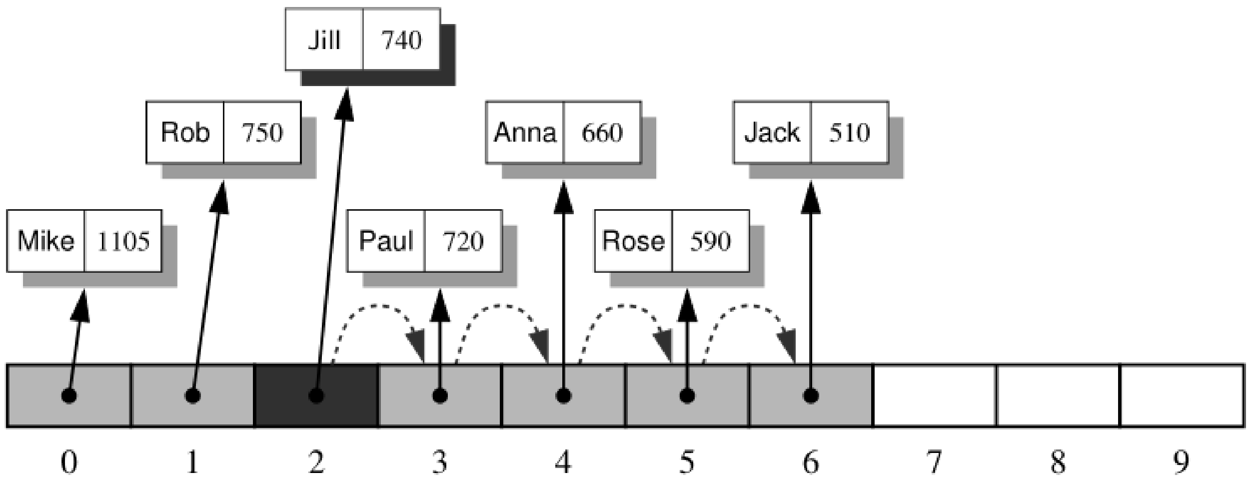



```
class Scoreboard:
  """Fixed-length sequence of high scores in nondecreasing order."""

  def __init__(self, capacity=10):
    """Initialize scoreboard with given maximum capacity.

    All entries are initially None.
    """
    self._board = [None] * capacity          # reserve space for future scores
    self._n = 0                              # number of actual entries
```

Example-1 Scoreboard

```
def __getitem__(self, k):
  """Return entry at index k."""
  return self._board[k]

def __str__(self):
  """Return string representation of the high score list."""
  return '\n'.join(str(self._board[j]) for j in range(self._n))
```

Example-1 Scoreboard



Example-1 Scoreboard

```
def add(self, entry):
  """Consider adding entry to high scores."""
  score = entry.get_score()

  # Does new entry qualify as a high score?
  # answer is yes if board not full or score is higher than last entry
  good = self._n < len(self._board) or score > self._board[-1].get_score()

  if good:
    if self._n < len(self._board):          # no score drops from list
      self._n += 1                          # so overall number increases

    # shift lower scores rightward to make room for new entry
    j = self._n - 1
    while j > 0 and self._board[j-1].get_score() < score:
      self._board[j] = self._board[j-1]     # shift entry from j-1 to j
      j -= 1                                # and decrement j
    self._board[j] = entry                  # when done, add new entry
```

similar to `insert` function of `list`

Example-2 Insertion Sort

There exists many sorting algorithms in CS literature.

One of the simplest algorithm is insertion sort.

Algorithm InsertionSort(A):

Input: An array A of n comparable elements

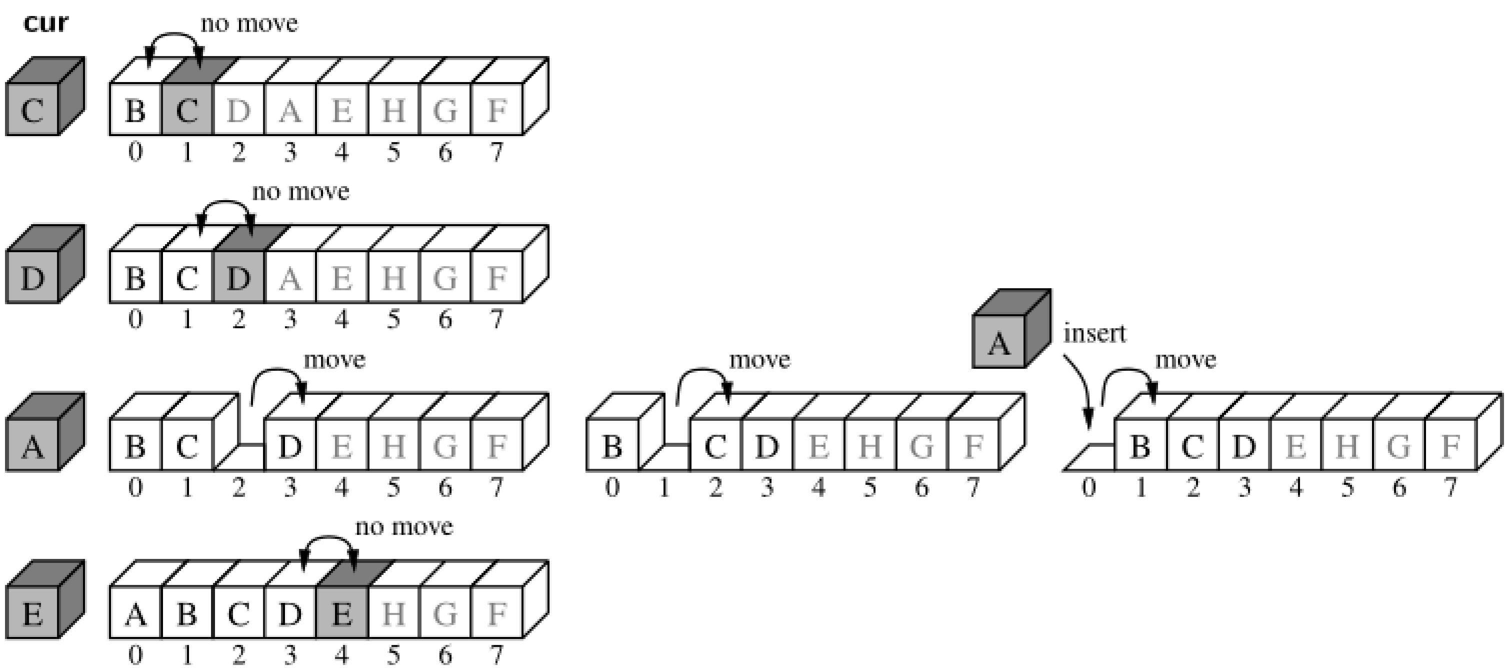
Output: The array A with elements rearranged in nondecreasing order

for k from 1 to n − 1 **do**

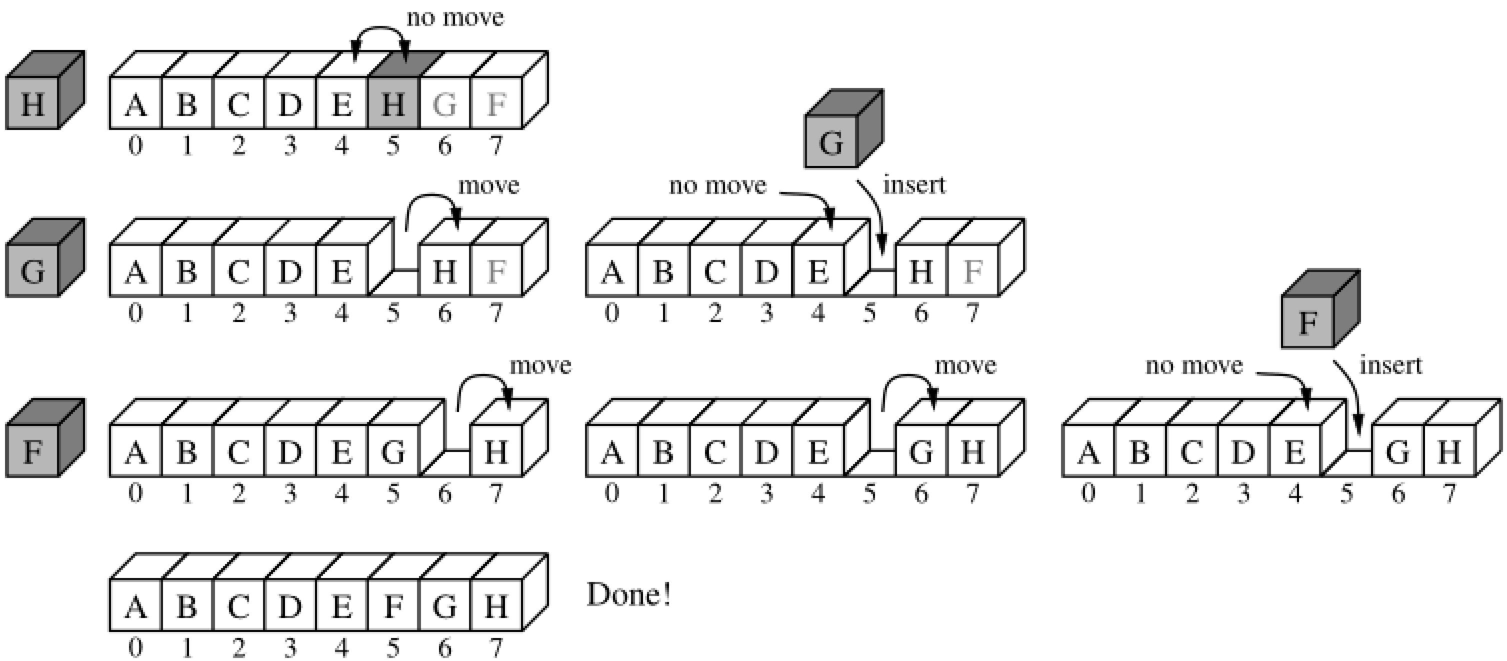
Insert A[k] at its proper location within A[0], A[1], ..., A[k].

Code Fragment 5.9: High-level description of the insertion-sort algorithm.

Example-2 Insertion Sort



Example-2 Insertion Sort



Example-2 Insertion Sort

start from the 2nd item

```
def insertion_sort(A):
  """Sort list of comparable elements into nondecreasing order."""
  for k in range(1, len(A)):              # from 1 to n-1
    cur = A[k]                            # current element to be inserted
    j = k                                 # find correct index j for current
    while j > 0 and A[j-1] > cur:         # element A[j-1] must be after current
      A[j] = A[j-1]
      j -= 1
    A[j] = cur                            # cur is now in the right place
```

move

insert

an index that will work backwards

Example-2 Insertion Sort

Worst-case complexity $O(n^2)$ (i.e., list is in reversed order)

Make 1+2+3+...+(n-1) comparisons. n(n+1)/2

Best-case O(n) (i.e., list is already ordered)

Make n-1 comparisons

Average case $O(n^2)$

Not recommended for large arrays.

For small arrays, it is strongly advised.

Example-2 Insertion Sort

Question for the next lecture:

Prove that insertion Sort's average performance is $O(n^2)$.

Hint: Number of operations ≈ Number of inversions

What is inversion?

A pair of items that are not in some proper order. For example, if we were to order a list of numbers from low to high, then any pair of numbers from that list is called an inversion when the number at the lower index is higher than the other number at a higher index. For example, in a list such as [1,3,2,5,4], pairs (3,2) and (5,4) are inversions.