Data structure and algorithm in Python

Array-Based Sequences

Seongjin Lee

Dept of Aerospace and Software Engineering,
Gyeongsang National University

Table of contents

Python's Sequence Types

Python's Sequence Types

In this chapter, we explore Python's various "sequence" classes, namely the built-in **list**, **tuple**, and **str** classes.

Python's Sequence Types

In this chapter, we explore Python's various "sequence" classes, namely the built-in **list**, **tuple**, and **str** classes.

Commonality

- supports indexing to access an individual element of a sequence, using a syntax such as `seq[k]`;
- uses a low-level concept known as an **array** to represent the sequence.

Python's Sequence Types

In this chapter, we explore Python's various "sequence" classes, namely the built-in **list**, **tuple**, and **str** classes.

Commonality

- supports indexing to access an individual element of a sequence, using a syntax such as `seq[k]`;
- uses a low-level concept known as an **array** to represent the sequence.

Differences

- the abstractions that these classes represent
- the way that instances of these classes are represented internally by Python

Python's Sequence Types

Because these classes are used so widely in Python programs, and because they will become building blocks upon which we will develop more complex data structures, it is imperative that we establish a clear understanding of both the public behavior and inner workings of these classes.

Low-Level's Arrays

Memory Address

Each byte of memory is associated with a unique number that serves as its address.

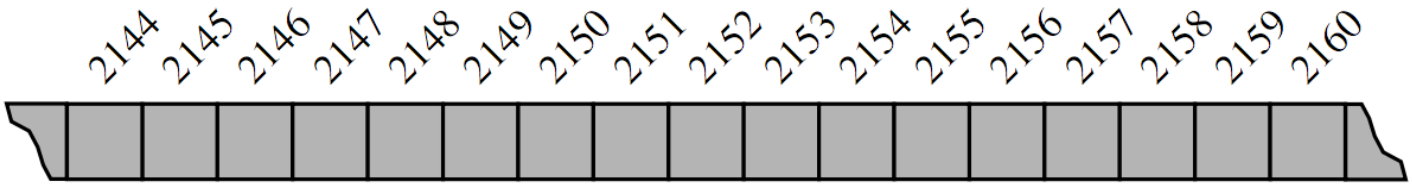


Figure 5.1: A representation of a portion of a computer's memory, with individual bytes labeled with consecutive memory addresses.

Memory addresses are typically coordinated with the physical layout of the memory system, and so we often portray the numbers in sequential fashion.

Low-Level's Arrays

Definition : Array

A group of related variables can be stored one after another in a contiguous portion of the computer's memory. Such a representation is denoted as **array**.

Low-Level's Arrays

Example

A text string is stored as an ordered sequence of individual characters. We describe this as an array of six characters, even though it requires 12 bytes of memory. We will refer to each location within an array as a **cell**, and will use an integer index to describe its location within the array, with cells numbered starting with 0, 1, 2, and so on.

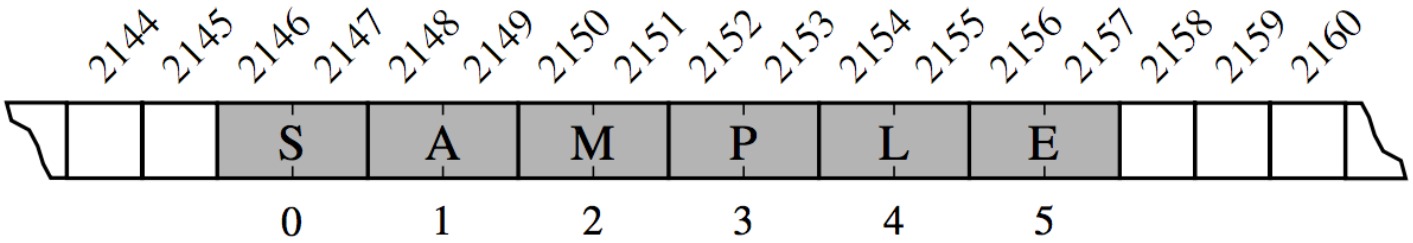


Figure 5.2: A Python string embedded as an array of characters in the computer's memory. We assume that each Unicode character of the string requires two bytes of memory. The numbers below the entries are indices into the string.

Low-Level's Arrays

Each cell of an array must use the same number of bytes. This requirement is what allows an arbitrary cell of the array to be accessed in constant time based on its index.

Example

Given the memory address at which an array starts, the number of bytes per element, and a disired index within the array, the appropriate memory address can be computed using start + cellsize * index.

Low-Level's Arrays

Referential Arrays

Referential Arrays

In Python, each cell of the array must use the same number of bytes.

Referential Arrays

In Python, each cell of the array must use the same number of bytes.

Given a list of names:

```
['Rene', 'Joseph', 'Janet', 'Jonas', 'Helen', '
Virginia']
```

How to represent such a list within an array?

Referential Arrays

In Python, each cell of the array must use the same number of bytes.

Given a list of names:

```
['Rene', 'Joseph', 'Janet', 'Jonas', 'Helen', '
Virginia']
```

How to represent such a list within an array?

Way 1: Reserve enough space for each cell to hold the maximum length string, but that would be wastfull.

Referential Arrays

Way 2: Python represents a list or tuple instance using an internal storage mechanism of an array of object **references**. At the lowest level, what is stored is a consecutive sequence of memory addresses at which the elements of the sequence reside.

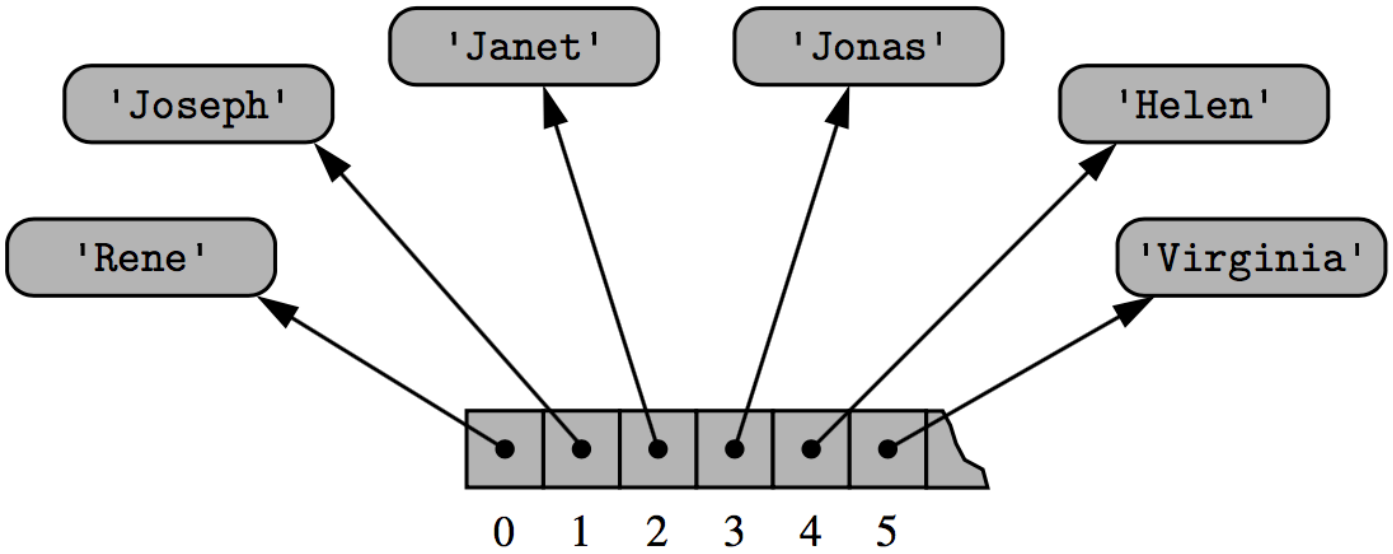


Figure 5.4: An array storing references to strings.

Referential Arrays

Although the relative size of the individual elements may vary, the number of bits used to store the memory address of each element is fixed. In this way, Python can support constant-time access to a list or tuple element based on its index.

Referential Arrays

A single list instance may include multiple references to the same object as elements of the list, and it is possible for a single object to be an element of two or more lists, as those lists simply store references back to that object.

Referential Arrays

Example

When you compute a slice of a list, the result is a new list instance, but that new list has references to the same elements that are in the original list.

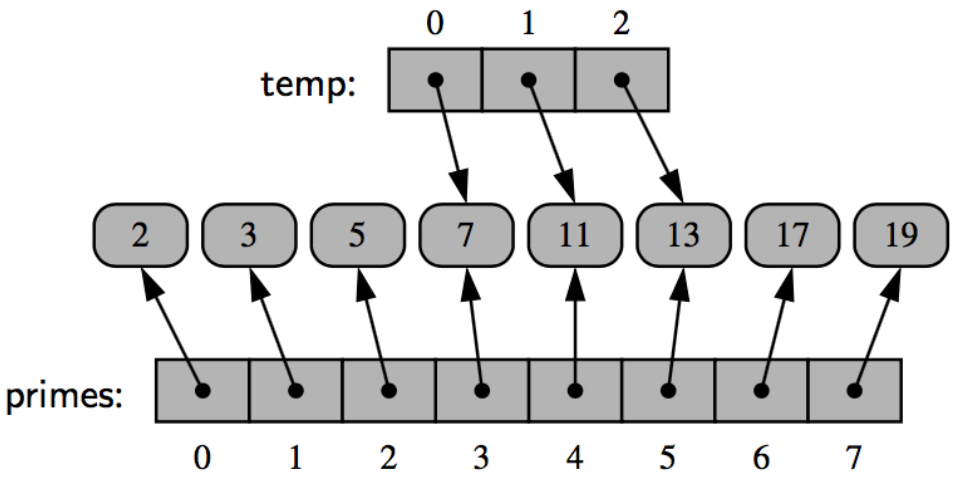


Figure 5.5: The result of the command temp = primes[3:6].

Referential Arrays

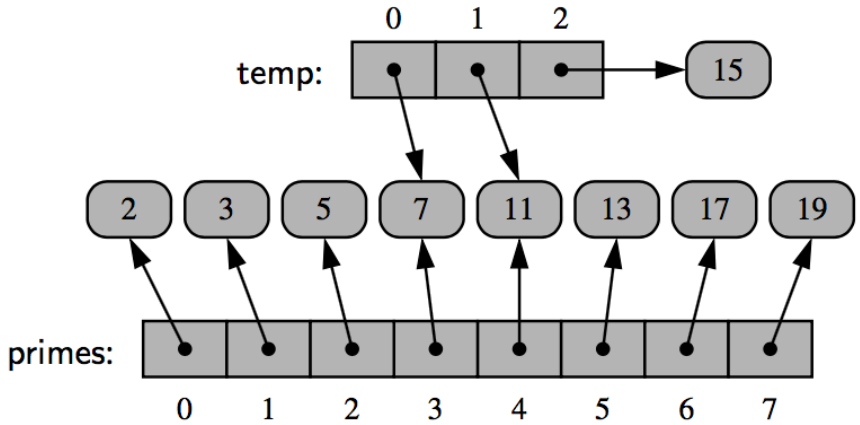


Figure 5.6: The result of the command temp[2] = 15 upon the configuration portrayed in Figure 5.5.

Referential Arrays

```
>>> data = [0] * 8
```

produces a list of length 8, with all 8 elements being the value 0.
Technically, all 8 cells of the list reference the same object.

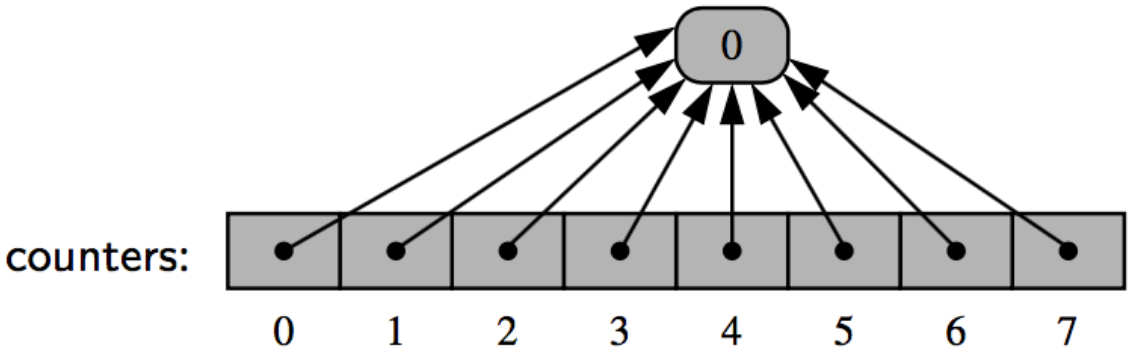


Figure 5.7: The result of the command data $=$ [0] $*$ 8.

Referential Arrays

```
>>> data[2] += 1
```

Due to the fact that the referenced integer is immutable, the above command does not change the value of the existing integer instance.

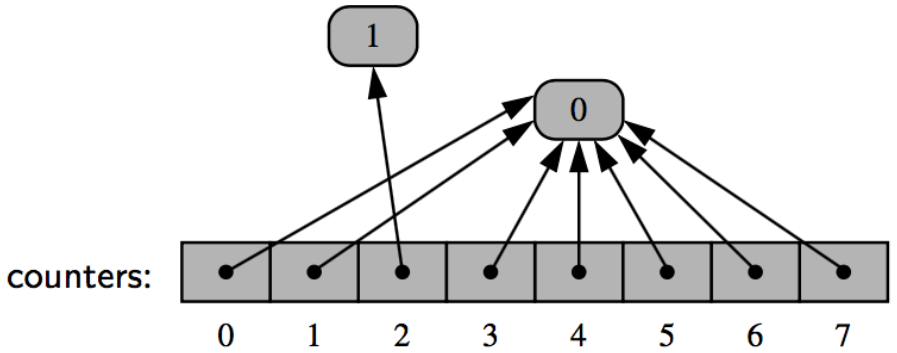


Figure 5.8: The result of command data[2] += 1 upon the list from Figure 5.7.

Referential Arrays

```
>>> primes = [2,3,5,7,11,13,17,19]
>>> extras = [23,29,31]
>>> primes.extend(extras)
```

The **extend** command is used to add all elements from one list to the end of another list.

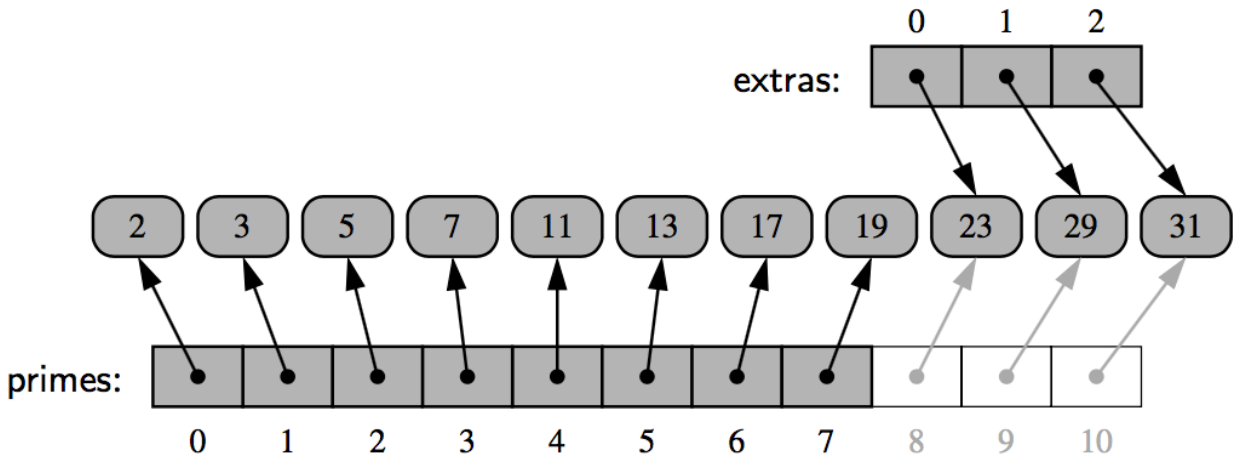


Figure 5.9: The effect of command primes.extend(extras), shown in light gray.

Referential Arrays

```
>>> primes = [2,3,5,7,11,13,17,19]
>>> extras = [23,29,31]
>>> primes.extend(extras)
```

The **extend** command is used to add all elements from one list to the end of another list.

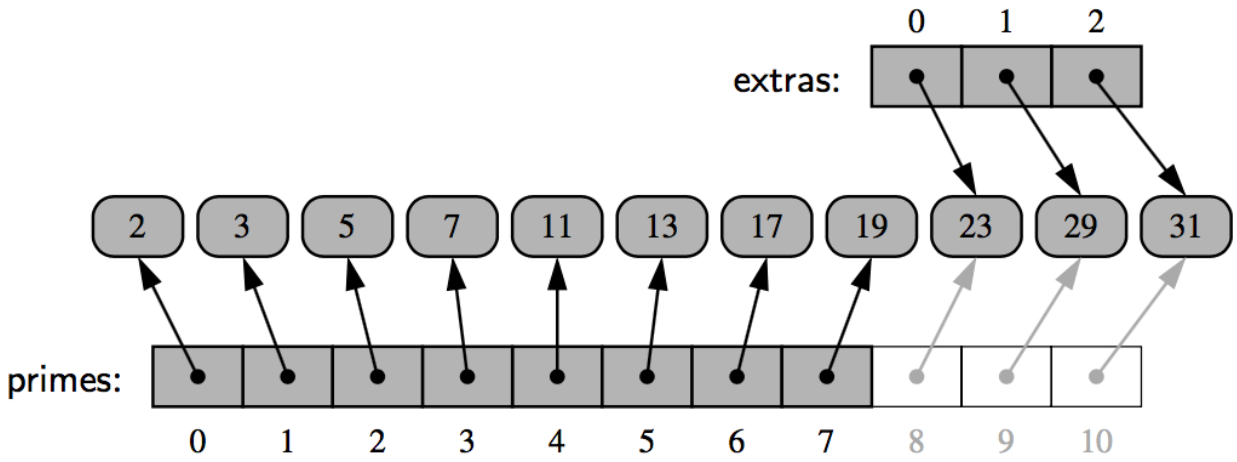


Figure 5.9: The effect of command primes.extend(extras), shown in light gray.

The extended list does not receive copies of those elements, it receives references to those elements.

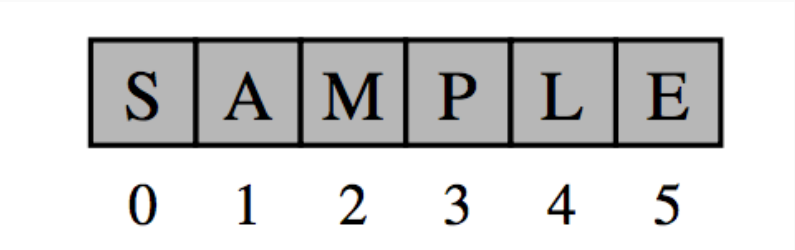
Low-Level's Arrays

Compact Arrays in Python

Compact Arrays in Python

Strings are represented using an array of characters (NOT an array of references).

We will refer to this more direct representation as a **compact array** because the array is storing the bits that represent the primary data (characters, in the case of strings).



Compact Arrays in Python

Compact arrays have several advantages over referential structures in terms of computing performance.

- overall memory usage will be much lower
- the primary data are stored consecutively in memory

Compact Arrays in Python

Primary support for compact arrays is in a module named **array**. That module defines a class, also named **array**, providing compact storage for arrays of primitive data types.

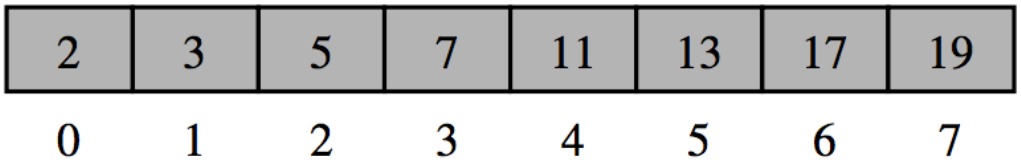


Figure 5.10: Integers stored compactly as elements of a Python array.

Compact Arrays in Python

```
>>> from array import array
>>> print(array('i', [1,2,3,4,5]))
array('i', [1, 2, 3, 4])
>>> print(array('f', [1,2,3,4,5]))
array('f', [1.0, 2.0, 3.0, 4.0])
>>> print(array('d', [1,2,3,4,5]))
array('d', [1.0, 2.0, 3.0, 4.0])
```

Compact Arrays in Python

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

Table 5.1: Type codes supported by the array module.

Dynamic Arrays

Dynamic Arrays

When creating a low-level array in a computer system, the precise size of that array must be explicitly declared in order for the system to properly allocate a consecutive piece of memory for its storage.

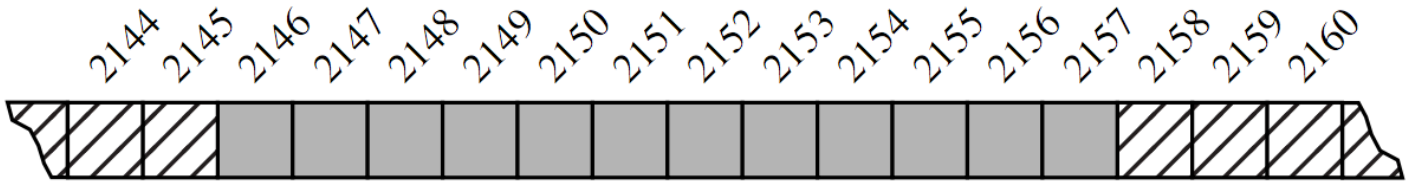


Figure 5.11: An array of 12 bytes allocated in memory locations 2146 through 2157.

Dynamic Arrays

Because the system might dedicate neighboring memory locations to store other data, the capacity of an array cannot trivially be increased by expanding into subsequent cells. In the context of representing a Python **tuple** or **str** instance, this constraint is no problem.

Dynamic Arrays

Python's list class presents a more interesting abstraction.

Although a list has a particular length when constructed, the class allows us to add elements to the list, with no apparent limit on the overall capacity of the list.

To provide this abstraction, Python relies on an algorithmic sleight of hand known as a **dynamic array**.

Dynamic Arrays

A list instance maintains an underlying array that often has greater capacity than the current length of the list.

This extra capacity makes it easy to append a new element to the list by using the next available cell of the array.

Dynamic Arrays

If a user continues to append elements to a list, any reserved capacity will eventually be exhausted.

- In that case, the class requests a new, larger array from the system, and initializes the new array so that its prefix matches that of the existing smaller array.
- At that point in time, the old array is no longer needed, so it is reclaimed by the system.

Dynamic Arrays

If a user continues to append elements to a list, any reserved capacity will eventually be exhausted.

- In that case, the class requests a new, larger array from the system, and initializes the new array so that its prefix matches that of the existing smaller array.
- At that point in time, the old array is no longer needed, so it is reclaimed by the system.

Like hermit crab

Dynamic Arrays

listsize.py

```
import sys
try:
    n = int(sys.argv[1])
except:
    n = 100
data = []
for k in range(n):   # NOTE: must fix choice of n
  a = len(data)      # number of elements
  b = sys.getsizeof(data) # actual size in bytes
  print('Length: {0:3d}; Size in bytes: {1:4d}'.
  format(a, b))
  data.append(None) # increase length by one
```

Dynamic Arrays

```
$ python listsize.py 6
Length:    0; Size in bytes:    64
Length:    1; Size in bytes:    96
Length:    2; Size in bytes:    96
Length:    3; Size in bytes:    96
Length:    4; Size in bytes:    96
Length:    5; Size in bytes:   128
```

Dynamic Arrays

Implementing a Dynamic Array

Implementing a Dynamic Array

Although the Python list class provides a highly optimized implementation of dynamic arrays, upon which we rely for the remainder of this book, it is instructive to see how such a class might be implemented.

Implementing a Dynamic Array

Although the Python list class provides a highly optimized implementation of dynamic arrays, upon which we rely for the remainder of this book, it is instructive to see how such a class might be implemented.

The key is to provide means to grow the array A that stores the elements of a list. Of course, we cannot actually grow that array, as its capacity is fixed.

Implementing a Dynamic Array

If an element is appended to a list at a time when the underlying array is full, we perform the following steps:

- Allocate a new array B with larger capacity.
- Set $B[i] = A[i]$, for $i = 0, ..., n1$, where n denotes current number of items.
- Set $A = B$, that is, we henceforth use B as the array supporting the list.
- Insert the new element in the new array.

Implementing a Dynamic Array

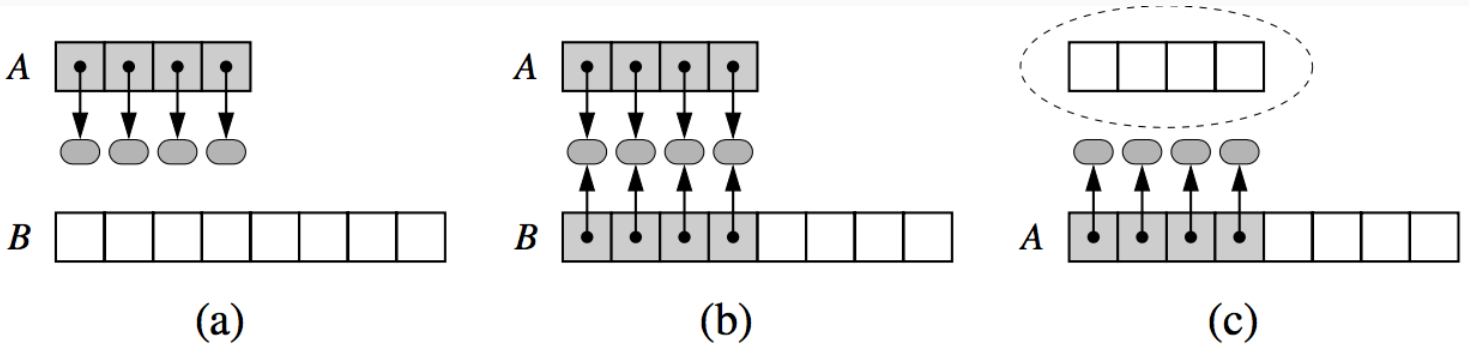


Figure 5.12: An illustration of the three steps for "growing" a dynamic array: (a) create new array B; (b) store elements of A in B; (c) reassign reference A to the new array. Not shown is the future garbage collection of the old array, or the insertion of the new element.

Implementing a Dynamic Array

The remaining issue to consider is how large of a new array to create. A commonly used rule is for the new array to have twice the capacity of the existing array that has been filled.

Implementing a Dynamic Array

```
import ctypes

class DynamicArray:
  """A dynamic array class akin to a simplified
  Python list."""
  def __init__(self):
    """Create an empty array."""
    self._n = 0
    self._capacity = 1
    self._A = self._make_array(self._capacity)

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

Implementing a Dynamic Array

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

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

Implementing a Dynamic Array

```
def _resize(self, c):
  """Resize internal array to capacity c."""
  B = self._make_array(c)
  for k in range(self._n):
    B[k] = self._A[k]
  self._A = B
  self._capacity = c
```

Implementing a Dynamic Array

```
def append(self, obj):
  """Add object to end of the array."""
  if self._n == self._capacity:
    self._resize(2 * self._capacity)
  self._A[self._n] = obj
  self._n += 1
```

Efficiency of Python's Sequence Types

Efficiency of Python's Sequence Types

Python's List and Tuple Classes

Efficiency of Python's Sequence Types

Operation	Running Time
len(data)	$O(1)$
data[j]	$O(1)$
data.count(value)	$O(n)$
data.index(value)	$O(k+1)$
value **in** data	$O(k+1)$
data1 == data2 (similarly !=, <, <=, >, >=)	$O(k+1)$
data[j:k]	$O(k-j+1)$
data1 + data2	$O(n_1+n_2)$
c * data	$O(cn)$

Table 5.3: Asymptotic performance of the nonmutating behaviors of the list and tuple classes. Identifiers data, data1, and data2 designate instances of the list or tuple class, and n, n_1, and n_2 their respective lengths. For the containment check and index method, k represents the index of the leftmost occurrence (with $k = n$ if there is no occurrence). For comparisons between two sequences, we let k denote the leftmost index at which they disagree or else $k = \min(n_1, n_2)$.

Constant Operation

- `len(data)`
- `data[j]`

Searching for Occurrences of a Value

- `count`

  the loop for computing the count must proceed through the entire sequence,

- `index`, `__contains__`

  the loops for checking containment of an element or determining the index of an element immediately exit once they find the leftmost occurrence of the desired value, if one exists.

Searching for Occurrences of a Value

Example

```
data = list(range(10000000))
```

- `5 in data`: Best
- `9999995 in data`: Middle
- `-5 in data`: Worst

Creating New Instances

The asymptotic behavior is proportional to the length of the result.

Example

- The slice `data[6000000:6000008]` can be constructed almost immediately because it has only eight elements;
- the slice `data[6000000:7000000]` has one million elements, and thus is more time-consuming to create

Mutating Behaviors

Operation	Running Time
data[j] = val	$O(1)$
data.append(value)	$O(1)^*$
data.insert(k, value)	$O(n-k+1)^*$
data.pop( )	$O(1)^*$
data.pop(k) **del** data[k]	$O(n-k)^*$
data.remove(value)	$O(n)^*$
data1.extend(data2) data1 += data2	$O(n_2)^*$
data.reverse( )	$O(n)$
data.sort( )	$O(n \log n)$

*amortized

Table 5.4: Asymptotic performance of the mutating behaviors of the list class. Identifiers data, data1, and data2 designate instances of the list class, and n, n_1, and n_2 their respective lengths.

Mutating Behaviors

The simplest of those behaviors has syntax `data[j] = val`, and is supported by the special `__setitem__` method. This operation has worst-case $O(1)$ running time because

- it simply replaces one element of a list with a new value;
- no other elements are affected and the size of the underlying array does not change.

Adding Elements to a List

- The append method

  requires $O(n)$ time because the underlying array is resized, but uses $O(1)$ time in the amortized sense.

Adding Elements to a List

- The insert method

  `insert(k, value)` inserts a given value into the list at index $0 \leq k \leq n$ while shifting all subsequent elements back one slot to make room.

Adding Elements to a List

```
def insert(self, k, value):
  """Insert value at index k, shifting
  subsequent values rightward."""
  # (for simplicity, we assume 0 <= k <= n in
  this verion)
  if self._n == self._capacity:
    self._resize(2 * self._capacity)
  for j in range(self._n, k, -1):
    self._A[j] = self._A[j-1]
  self._A[k] = value
  self._n += 1
```

Adding Elements to a List

	N				
	100	1,000	10,000	100,000	1,000,000
$k = 0$	0.482	0.765	4.014	36.643	351.590
$k = n // 2$	0.451	0.577	2.191	17.873	175.383
$k = n$	0.420	0.422	0.395	0.389	0.397

Table 5.5: Average running time of insert(k, val), measured in microseconds, as observed over a sequence of N calls, starting with an empty list. We let n denote the size of the current list (as opposed to the final list).

Adding Elements to a List

	N				
	100	1,000	10,000	100,000	1,000,000
$k = 0$	0.482	0.765	4.014	36.643	351.590
$k = n // 2$	0.451	0.577	2.191	17.873	175.383
$k = n$	0.420	0.422	0.395	0.389	0.397

Table 5.5: Average running time of insert(k, val), measured in microseconds, as observed over a sequence of N calls, starting with an empty list. We let n denote the size of the current list (as opposed to the final list).

- Inserting at the beginning of a list is most expensive, requiring linear time per operation;
- Inserting at the middle requires about half the time as inserting at the beginning, yet is still $O(n)$ time;
- Inserting at the end displays $O(1)$ behavior, akin to append.

Removing Elements from a List

- `pop()`: removes the last element from a list.

  This is most efficient, because all other elements remain in their original location. This is effectively an $O(1)$ operation, but the bound is amortized because Python will occasionally shrink the underlying dynamic array to conserve memory.

Removing Elements from a List

- `pop(k)`: removes the element that is at index $k < n$ of a list, shifting all subsequent elements leftward to fill the gap that results from the removal.

  The efficiency of this operation is $O(nk)$, as the amount of shifting depends upon the choice of index k.

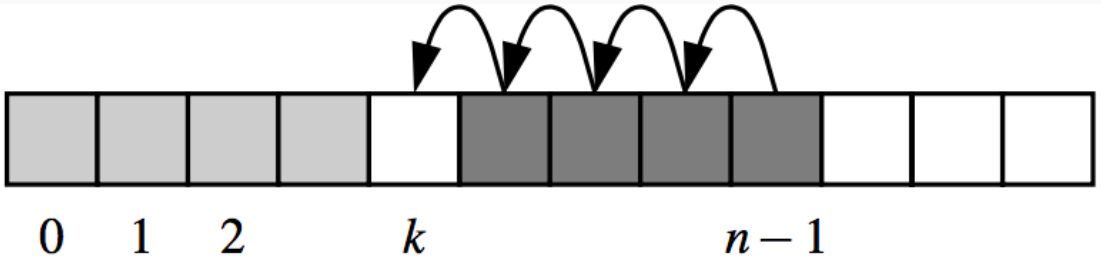


Figure 5.17: Removing an element at index k of a dynamic array.

Removing Elements from a List

- The `remove` method

  `remove(value)` allows the caller to specify the value that should be removed.

  The efficiency of this operation is $O(n)$. One part of the process searches from the beginning until finding the value at index k, while the rest iterates from k to the end in order to shift elements leftward.

Removing Elements from a List

```
def remove(self, value):
  """Remove first occurrence of value (or
  raise ValueError)."""
  # note: we do not consider shrinking the
  dynamic array in this version
  for k in range(self._n):
    if self._A[k] == value:
      for j in range(k, self._n - 1):
        self._A[j] = self._A[j+1]
      self._A[self._n - 1] = None
      self._n -= 1 # we have one less item
      return       # exit immediately
  raise ValueError('value not found')
```

Extending a List

- The `extend` method
- add all elements of one list to the end of a second list
- A call to `data.extend(other)` produces the same outcome as the code,

```
for element in other:
  data.append(element)
```

In either case, the running time is proportional to the length of the other list, and amortized because the underlying array for the first list may be resized to accommodate the additional elements.

Extending a List

In practice, the extend method is preferable to repeated calls to append because the constant factors hidden in the asymptotic analysis are significantly smaller.

Extending a List

The greater efficiency of extend is threefold.

1. There is always some advantage to using an appropriate Python method, because those methods are often implemented natively in a compiled language (rather than as interpreted Python code).
2. There is less overhead to a single function call that accomplishes all the work, versus many individual function calls.
3. Increased efficiency of extend comes from the fact that the resulting size of the updated list can be calculated in advance. If the second data set is quite large, there is some risk that the underlying dynamic array might be resized multiple times when using repeated calls to append. With a single call to extend, at most one resize operation will be performed.

Constructing New Lists

- List Comprehension

```
squares = [k*k for k in range(1, n+1)]
```

- Loop

```
squares = []
for k in range (1, n+1):
  squares.append(k*k)
```

The list comprehension syntax is significantly faster than building the list by repeatedly appending.

Constructing New Lists

Initialize a list of constant values using the multiplication operator, as in `[0] * n` to produce a list of length n with all values equal to zero. It is more efficient than building such a list incrementally.

Using Array-Based Sequences

Using Array-Based Sequences

Storing High Scores for a Game

Storing High Scores for a Game

```
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)'
```

Storing High Scores for a Game

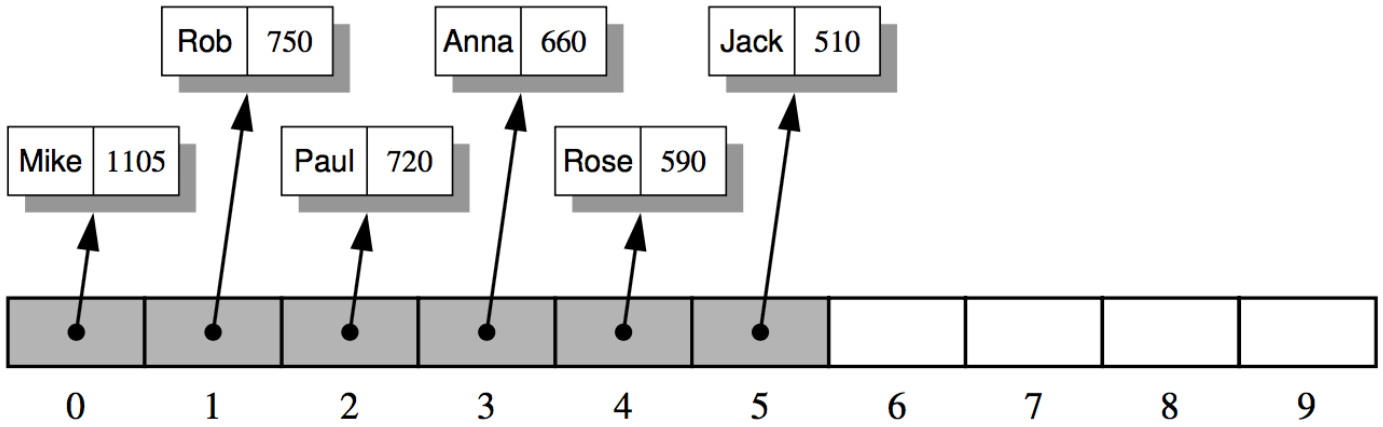


Figure 5.18: An illustration of an ordered list of length ten, storing references to six GameEntry objects in the cells from index 0 to 5, with the rest being None.

Storing High Scores for a Game

```
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
    self._n = 0

  def __getitem__(self, k):
    return self._board[k]

  def __str__(self):
    return '\n'.join(str(self._board[j]) for j
    in range(self._n))
```

Storing High Scores for a Game

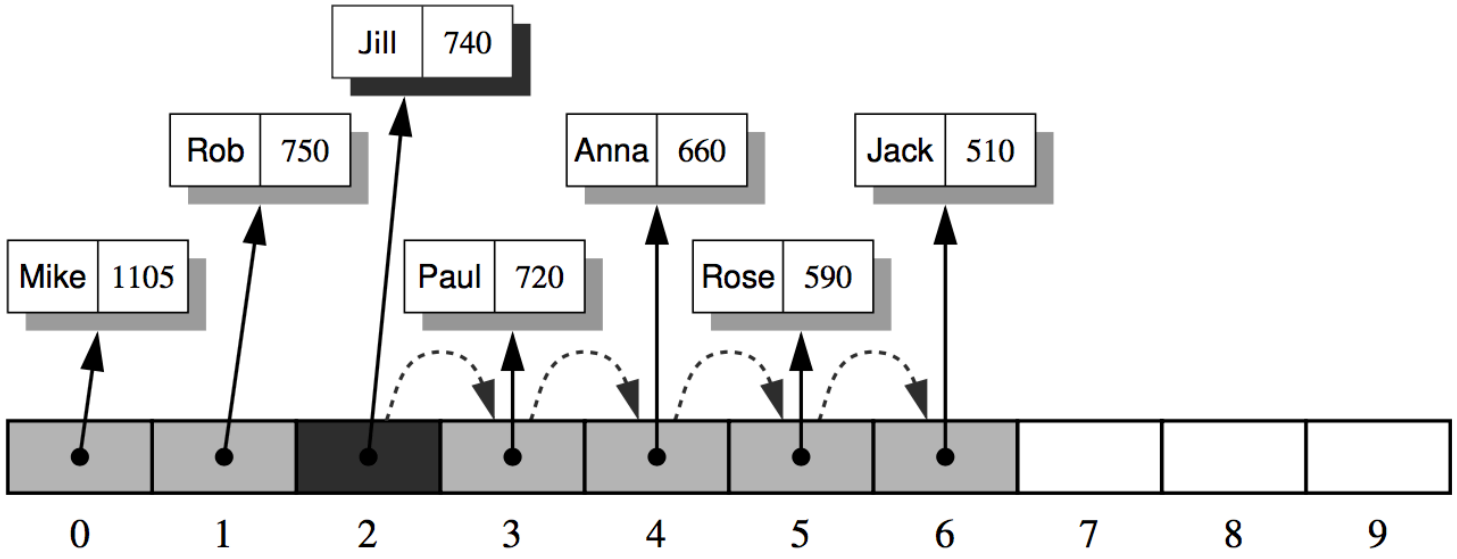


Figure 5.19: Adding a new GameEntry for Jill to the scoreboard. In order to make room for the new reference, we have to shift the references for game entries with smaller scores than the new one to the right by one cell. Then we can insert the new entry with index 2.

Storing High Scores for a Game

```
def add(self, entry):
  score = entry.get_score()
  good = self._n < len(self._board) or score >
   self._board[-1].get_score()

  if good:
    if self._n < len(self._board):
      self._n += 1
    j = self._n - 1
    while j > 0 and self._board[j-1].get_score
    () < score:
      self._board[j] = self._board[j-1]
      j -= 1
    self._board[j] = entry
```

Storing High Scores for a Game

```
if __name__ == '__main__':
  board = Scoreboard(5)
  for e in (
    ('Rob', 750), ('Mike',1105), ('Rose', 590),
    ('Jill', 740),('Jack', 510), ('Anna', 660),
    ('Paul', 720), ('Bob', 400),
  ):
    ge = GameEntry(e[0], e[1])
    board.add(ge)
    print('After considering {0}, scoreboard is:
    '.format(ge))
    print(board)
    print()
```

Using Array-Based Sequences

Sorting a Sequence

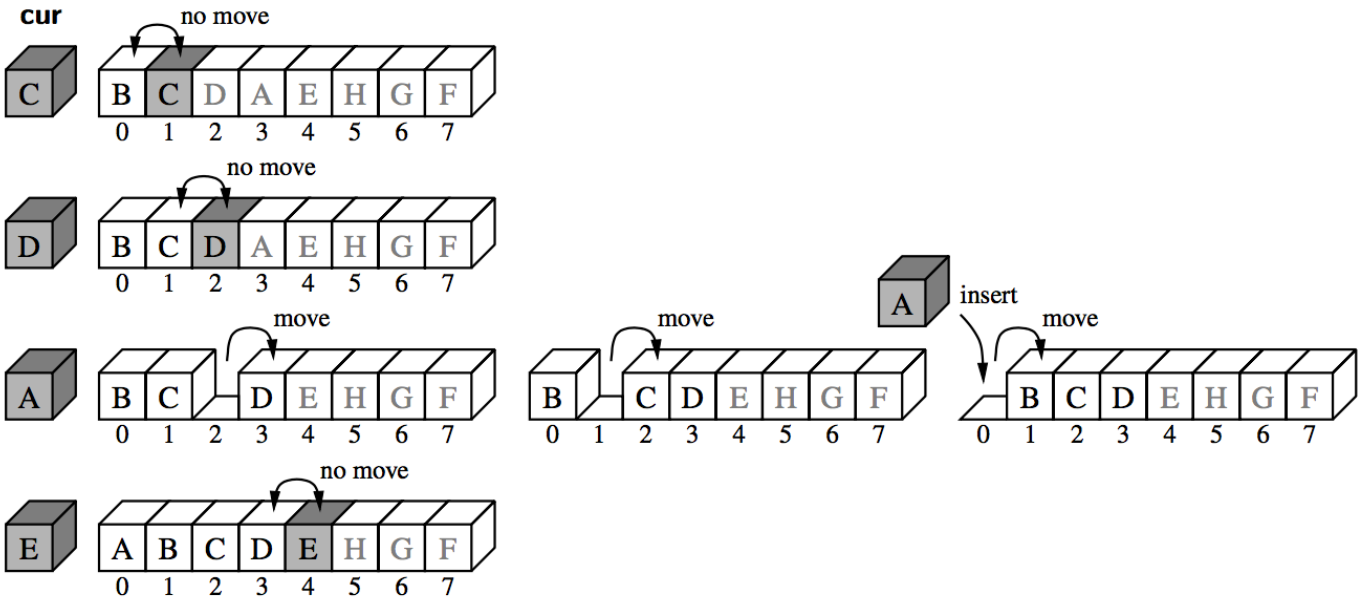
Sorting a Sequence

Starting with an unordered sequence of elements and rearranging them into nondecreasing order.

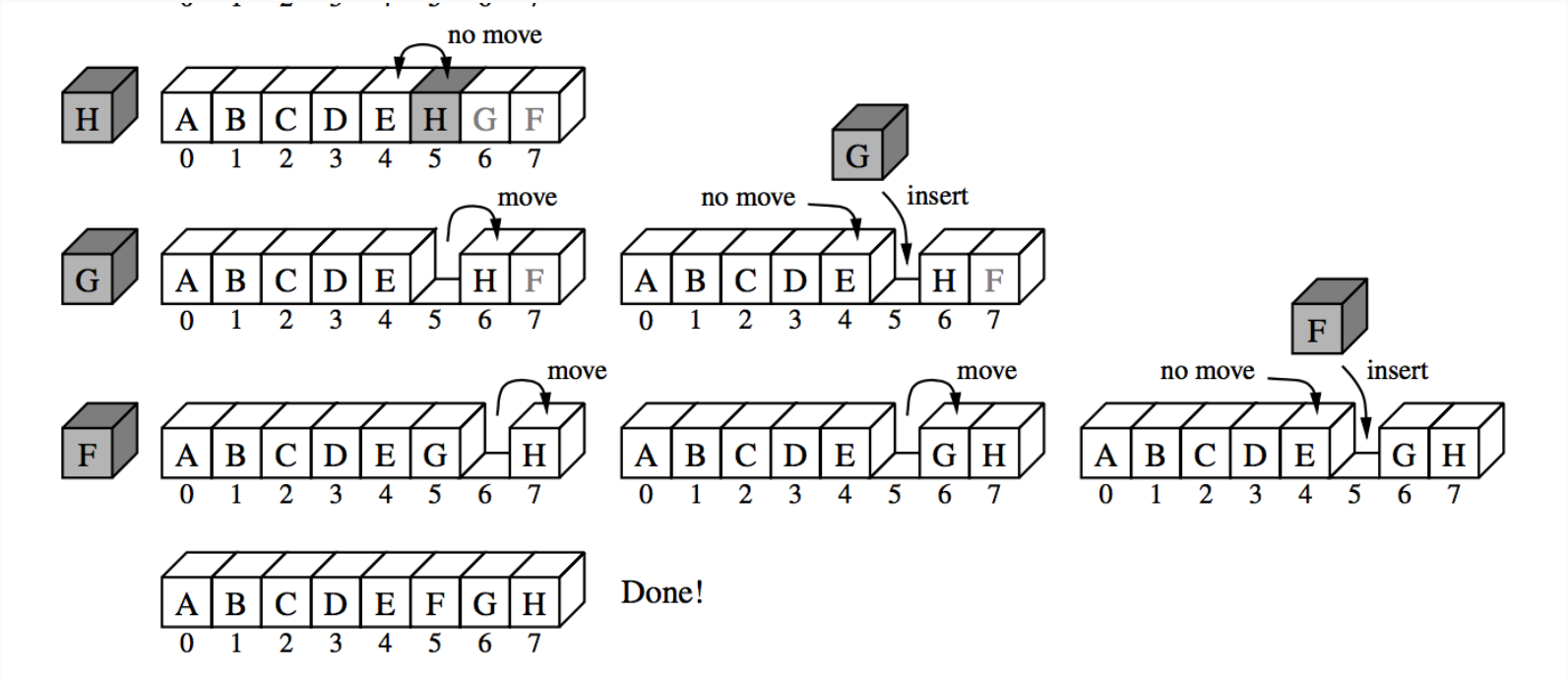
Insert-Sorting

- First start with the first element in the array. One element by itself is already sorted.
- Then consider the next element in the array. If it is smaller than the first, we swap them.
- Next consider the third element in the array. Swap it leftward until it is in its proper order with the first two elements.
- Then consider the fourth element, and swap it leftward until it is in the proper order with the first three.
- Continue in this manner with the fifth element, the sixth, and so on, until the whole array is sorted.

Insert-Sorting



Insert-Sorting



Insert-Sorting

```
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
```