5.2 Low-Level Arrays

To accurately describe the way in which Python represents the sequence types, we must first discuss aspects of the low-level computer architecture. The primary memory of a computer is composed of bits of information, and those bits are typically grouped into larger units that depend upon the precise system architecture. Such a typical unit is a ***byte***, which is equivalent to 8 bits.

A computer system will have a huge number of bytes of memory, and to keep track of what information is stored in what byte, the computer uses an abstraction known as a ***memory address***. In effect, each byte of memory is associated with a unique number that serves as its address (more formally, the binary representation of the number serves as the address). In this way, the computer system can refer to the data in “byte #2150” versus the data in “byte #2157,” for example. Memory addresses are typically coordinated with the physical layout of the memory system, and so we often portray the numbers in sequential fashion. Figure 5.1 provides such a diagram, with the designated memory address for each byte.

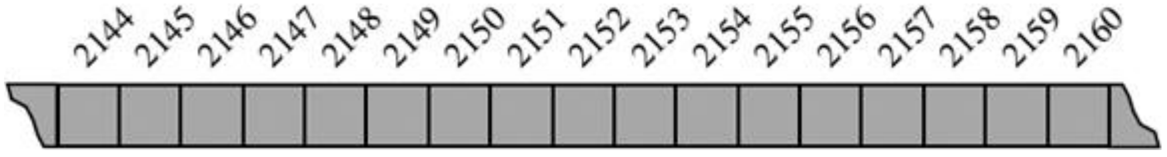


Figure 5.1: A representation of a portion of a computer's memory, with individual bytes labeled with consecutive memory addresses.

Despite the sequential nature of the numbering system, computer hardware is designed, in theory, so that any byte of the main memory can be efficiently accessed based upon its memory address. In this sense, we say that a computer's main memory performs as ***random access memory (RAM)***. That is, it is just as easy to retrieve byte #8675309 as it is to retrieve byte #309. (In practice, there are complicating factors including the use of caches and external memory; we address some of those issues in Chapter 15.) Using the notation for asymptotic analysis, we say that any individual byte of memory can be stored or retrieved in $O(1)$ time.

In general, a programming language keeps track of the association between an identifier and the memory address in which the associated value is stored. For example, identifier `x` might be associated with one value stored in memory, while `y` is associated with another value stored in memory. A common programming task is to keep track of a sequence of related objects. For example, we may want a video game to keep track of the top ten scores for that game. Rather than use ten different variables for this task, we would prefer to use a single name for the group and use index numbers to refer to the high scores in that group.

A group of related variables can be stored one after another in a contiguous portion of the computer's memory. We will denote such a representation as an ***array***. As a tangible example, a text string is stored as an ordered sequence of individual characters. In Python, each character is represented using the Unicode character set, and on most computing systems, Python internally represents each Unicode character with 16 bits (i.e., 2 bytes). Therefore, a six-character string, such as `'SAMPLE'`, would be stored in 12 consecutive bytes of memory, as diagrammed in Figure 5.2.

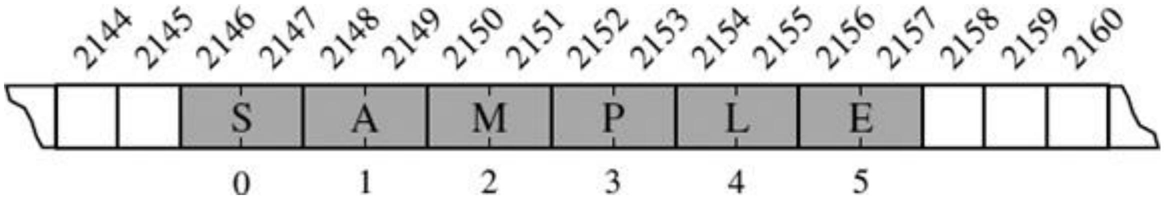


Figure 5.2: A Python string embedded as an array of characters in the computer's memory. We assume that each Unicode character of the string requires two bytes of memory. The numbers below the entries are indices into the string.

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. For example, in Figure 5.2, the cell of the array with index 4 has contents `L` and is stored in bytes 2154 and 2155 of memory.

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. In particular, if one knows the memory address at which an array starts (e.g., 2146 in Figure 5.2), the number of bytes per element (e.g., 2 for a Unicode character), and a desired index within the array, the appropriate memory address can be computed using the calculation, `start + cellsize * index`. By this formula, the cell at index 0 begins precisely at the start of the array, the cell at index 1 begins precisely `cellsize` bytes beyond the start of the array, and so on. As an example, cell 4 of Figure 5.2 begins at memory location 2146 + 2 · 4 = 2146 + 8 = 2154.

Of course, the arithmetic for calculating memory addresses within an array can be handled automatically. Therefore, a programmer can envision a more typical high-level abstraction of an array of characters as diagrammed in Figure 5.3.

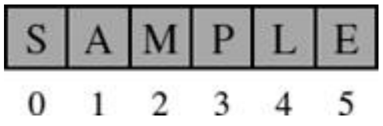


Figure 5.3: A higher-level abstraction for the string portrayed in Figure 5.2.

5.3 Dynamic Arrays and Amortization

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. For example, Figure 5.11 displays an array of 12 bytes that might be stored in memory locations 2146 through 2157.

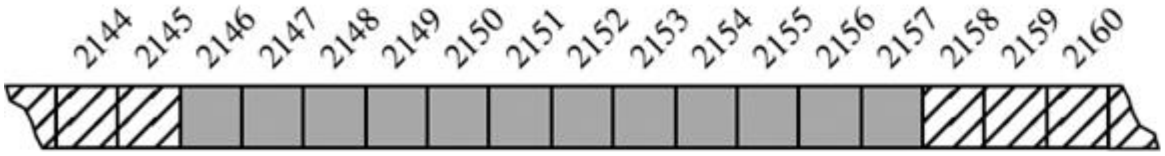


Figure 5.11: An array of 12 bytes allocated in memory locations 2146 through 2157.

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. Instances of those classes are immutable, so the correct size for an underlying array can be fixed when the object is instantiated.

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***.

The first key to providing the semantics of a dynamic array is that a list instance maintains an underlying array that often has greater capacity than the current length of the list. For example, while a user may have created a list with five elements, the system may have reserved an underlying array capable of storing eight object references (rather than only five). This extra capacity makes it easy to append a new element to the list by using the next available cell of the array.

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. Intuitively, this strategy is much like that of the hermit crab, which moves into a larger shell when it outgrows its previous one.

We give empirical evidence that Python's `list` class is based upon such a strategy. The source code for our experiment is displayed in Code Fragment 5.1, and a sample output of that program is given in Code Fragment 5.2. We rely on a function named `getsizeof` that is available from the `sys` module. This function reports the number of bytes that are being used to store an object in Python. For a list, it reports the number of bytes devoted to the array and other instance variables of the list, but *not* any space devoted to elements referenced by the list.

```
import sys                                          # provides getsizeof function
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
```

Code Fragment 5.1: An experiment to explore the relationship between a list's length and its underlying size in Python.

```
Length:   0; Size in bytes:   72
Length:   1; Size in bytes:  104
Length:   2; Size in bytes:  104
Length:   3; Size in bytes:  104
Length:   4; Size in bytes:  104
Length:   5; Size in bytes:  136
Length:   6; Size in bytes:  136
Length:   7; Size in bytes:  136
Length:   8; Size in bytes:  136
Length:   9; Size in bytes:  200
Length:  10; Size in bytes:  200
Length:  11; Size in bytes:  200
Length:  12; Size in bytes:  200
Length:  13; Size in bytes:  200
Length:  14; Size in bytes:  200
Length:  15; Size in bytes:  200
Length:  16; Size in bytes:  200
Length:  17; Size in bytes:  272
Length:  18; Size in bytes:  272
Length:  19; Size in bytes:  272
Length:  20; Size in bytes:  272
Length:  21; Size in bytes:  272
Length:  22; Size in bytes:  272
Length:  23; Size in bytes:  272
Length:  24; Size in bytes:  272
Length:  25; Size in bytes:  272
Length:  26; Size in bytes:  352
```

Code Fragment 5.2: Sample output from the experiment of Code Fragment 5.1.

In evaluating the results of the experiment, we draw attention to the first line of output from Code Fragment 5.2. We see that an empty list instance already requires a certain number of bytes of memory (72 on our system). In fact, each object in Python maintains some state, for example, a reference to denote the class to which it belongs. Although we cannot directly access private instance variables for a list, we can speculate that in some form it maintains state information akin to:

`_n`	The number of actual elements currently stored in the list.
`_capacity`	The maximum number of elements that could be stored in the currently allocated array.
`_A`	The reference to the currently allocated array (initially `None`).

As soon as the first element is inserted into the list, we detect a change in the underlying size of the structure. In particular, we see the number of bytes jump from 72 to 104, an increase of exactly 32 bytes. Our experiment was run on a 64-bit machine architecture, meaning that each memory address is a 64-bit number (i.e., 8 bytes). We speculate that the increase of 32 bytes reflects the allocation of an underlying array capable of storing four object references. This hypothesis is consistent with the fact that we do not see any underlying change in the memory usage after inserting the second, third, or fourth element into the list.

After the fifth element has been added to the list, we see the memory usage jump from 104 bytes to 136 bytes. If we assume the original base usage of 72 bytes for the list, the total of 136 suggests an additional $64 = 8 \times 8$ bytes that provide capacity for up to eight object references. Again, this is consistent with the experiment, as the memory usage does not increase again until the ninth insertion. At that point, the 200 bytes can be viewed as the original 72 plus an additional 128-byte array to store 16 object references. The 17^{th} insertion pushes the overall memory usage to $272 = 72 + 200 = 72 + 25 \times 8$, hence enough to store up to 25 element references.

Because a list is a referential structure, the result of `getsizeof` for a list instance only includes the size for representing its primary structure; it does not account for memory used by the *objects* that are elements of the list. In our experiment, we repeatedly append `None` to the list, because we do not care about the contents, but we could append any type of object without affecting the number of bytes reported by `getsizeof(data)`.

If we were to continue such an experiment for further iterations, we might try to discern the pattern for how large of an array Python creates each time the capacity of the previous array is exhausted (see Exercises R-5.2 and C-5.13). Before exploring the precise sequence of capacities used by Python, we continue in this section by describing a general approach for implementing dynamic arrays and for performing an asymptotic analysis of their performance.

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

1. Allocate a new array B with larger capacity.
2. Set $B[i] = A[i]$, for $i = 0,\ldots, n-1$, where n denotes current number of items.
3. Set $A = B$, that is, we henceforth use B as the array supporting the list.
4. Insert the new element in the new array.

An illustration of this process is shown in Figure 5.12.

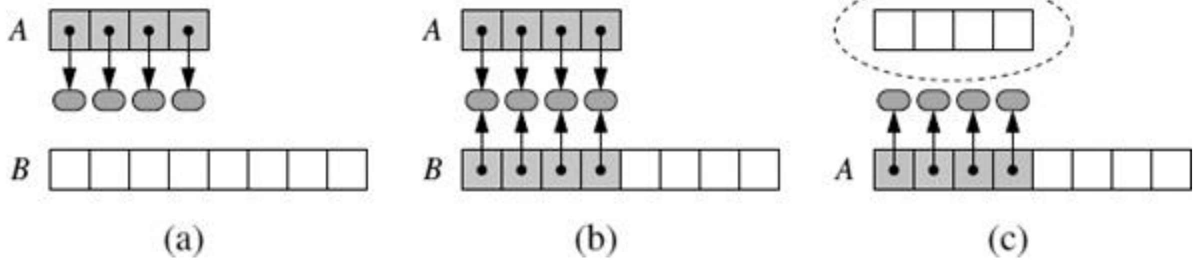


Figure 5.12: An illustration of the three steps for "growing" a dynamic array: (a) create new array *B*; (b) store elements of *A* in *B*; (c) reassign reference *A* to the new array. Not shown is the future garbage collection of the old array, or the insertion of the new element.

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. In Section 5.3.2, we will provide a mathematical analysis to justify such a choice.

In Code Fragment 5.3, we offer a concrete implementation of dynamic arrays in Python. Our `DynamicArray` class is designed using ideas described in this section. While consistent with the interface of a Python `list` class, we provide only limited functionality in the form of an `append` method, and accessors _ _len _ _ and _ _getitem _ _. Support for creating low-level arrays is provided by a module named `ctypes`. Because we will not typically use such a low-level structure in the remainder of this book, we omit a detailed explanation of the `ctypes` module. Instead, we wrap the necessary command for declaring the raw array within a private utility method `_make_array`. The hallmark expansion procedure is performed in our nonpublic `_resize` method.

```
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

  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

  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
```

Code Fragment 5.3: An implementation of a `DynamicArray` class, using a raw array from the ctypes module as storage.

5.4 Efficiency of Python's Sequence Types

In the previous section, we began to explore the underpinnings of Python's `list` class, in terms of implementation strategies and efficiency. We continue in this section by examining the performance of all of Python's sequence types.

5.4.1 PYTHON'S LIST AND TUPLE CLASSES

The *nonmutating* behaviors of the `list` class are precisely those that are supported by the `tuple` class. We note that tuples are typically more memory efficient than lists because they are immutable; therefore, there is no need for an underlying dynamic array with surplus capacity. We summarize the asymptotic efficiency of the nonmutating behaviors of the `list` and `tuple` classes in Table 5.3. An explanation of this analysis follows.

Operation	**Running Ti**
`len(data)`	$O(1)$
`data[j]`	$O(1)$
`data.count(value)`	$O(n)$
`data.index(value)`	$O(k + 1)$
`value` **in** `data`	$O(k + 1)$
`datal == data2` (similarly !=, <, <=, >, >=)	$O(k + 1)$
`data[j:k]`	$O\{k - j + 1)$
`datal + 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-Time Operations

The length of an instance is returned in constant time because an instance explicitly maintains such state information. The constant-time efficiency of syntax `data[j]` is assured by the underlying access into an array.

Searching for Occurrences of a Value

Each of the `count, index`, and `_ _contains_ _` methods proceed through iteration of the sequence from left to right. In fact, Code Fragment 2.14 of Section 2.4.3 demonstrates how those behaviors might be implemented. Notably, the loop for computing the count must proceed through the entire sequence, while 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. So while `count` always examines the n elements of the sequence, `index` and `_ _contains_ _` examine n elements in the worst case, but may be faster. Empirical evidence can be found by setting `data = list(range(10000000))` and then comparing the relative efficiency of the test, `5` **in** `data`, relative to the test, `9999995` **in** `data`, or even the failed test, `−5` **in** `data`.

Lexicographic Comparisons

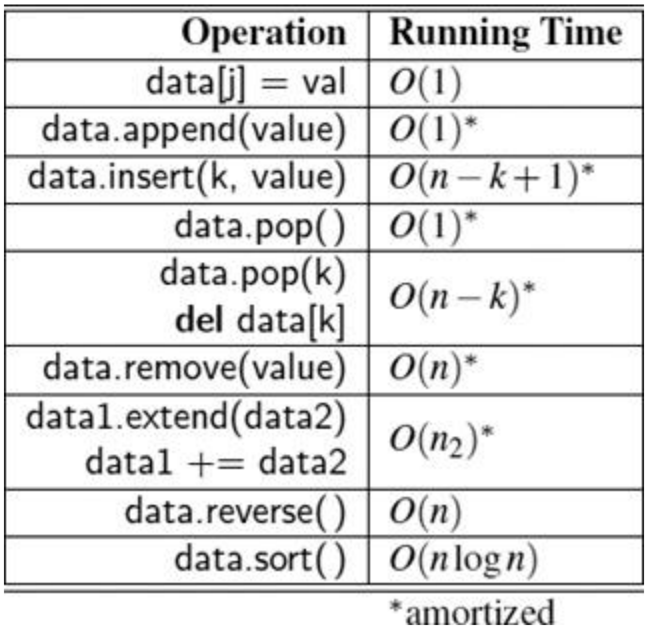
Comparisons between two sequences are defined lexicographically. In the worst case, evaluating such a condition requires an iteration taking time proportional to the length of the *shorter* of the two sequences (because when one sequence ends, the lexicographic result can be determined). However, in some cases the result of the test can be evaluated more efficiently. For example, if evaluating `[7, 3, ...] < [7, 5, ...]`, it is clear that the result is `True` without examining the remainders of those lists, because the second element of the left operand is strictly less than the second element of the right operand.

Creating New Instances

The final three behaviors in Table 5.3 are those that construct a new instance based on one or more existing instances. In all cases, the running time depends on the construction and initialization of the new result, and therefore the asymptotic behavior is proportional to the *length* of the result. Therefore, we find that slice `data[6000000:6000008]` can be constructed almost immediately because it has only eight elements, while slice `data[6000000:7000000]` has one million elements, and thus is more time-consuming to create.

Mutating Behaviors

The efficiency of the mutating behaviors of the `list` class are described in Table 5.3. 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. The more interesting behaviors to analyze are those that add or remove elements from the list.



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.

Adding Elements to a List

In Section 5.3 we fully explored the `append` method. In the worst case, it requires $\Omega(n)$ time because the underlying array is resized, but it uses $O(1)$ time in the amortized sense. Lists also support a method, with signature `insert(k, value)`, that 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. For the purpose of illustration, Code Fragment 5.5 provides an implementation of that method, in the context of our `DynamicArray` class introduced in Code Fragment 5.3. There are two complicating factors in analyzing the efficiency of such an operation. First, we note that the addition of one element may require a resizing of the dynamic array. That portion of the work requires $\Omega(n)$ worst-case time but only $O(1)$ amortized time, as per `append`. The other expense for `insert` is the shifting of elements to make room for the new item. The time for that process depends upon the index of the new element, and thus the number of other elements that must be shifted. That loop copies the reference that had been at index $n - 1$ to index n, then the reference that had been at index $n - 2$ to $n - 1$, continuing until copying the reference that had been at index k to $k + 1$, as illustrated in Figure 5.16. Overall this leads to an amortized $O(n - k + 1)$ performance for inserting at index k.

```
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:                # not enough room
    self._resize(2 * self._capacity)           # so double capacity
  for j in range(self._n, k, −1):              # shift rightmost first
    self._A[j] = self._A[j−1]
  self._A[k] = value                           # store newest element
  self._n += 1
```

Code Fragment 5.5: Implementation of `insert` for our `DynamicArray` class.

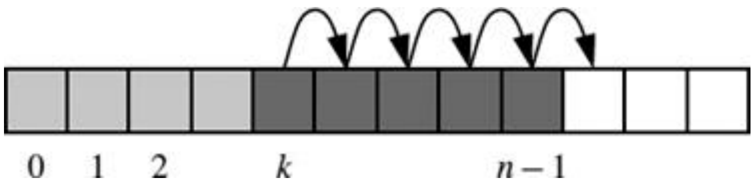


Figure 5.16: Creating room to insert a new element at index *k* of a dynamic array.

When exploring the efficiency of Python's `append` method in Section 5.3.3, we performed an experiment that measured the average cost of repeated calls on varying sizes of lists (see Code Fragment 5.4 and Table 5.2). We have repeated that experiment with the `insert` method, trying three different access patterns:

- In the first case, we repeatedly insert at the beginning of a list,
  - `for n in range(N):`
    `data.insert(0, None)`
- In a second case, we repeatedly insert near the middle of a list,
  - `for n in range(N):`
    `data.insert(n // 2, None)`
- In a third case, we repeatedly insert at the end of the list,
  - `for n in range(N):`
    `data.insert(n, None)`

The results of our experiment are given in Table 5.5, reporting the *average* time per operation (not the total time for the entire loop). As expected, we see that 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 $\Omega(n)$ time. Inserting at the end displays $O(1)$ behavior, akin to append.

	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).

Removing Elements from a List

Python's `list` class offers several ways to remove an element from a list. A call to `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.

The parameterized version, `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 $O(n-k)$, as the amount of shifting depends upon the choice of index *k*, as illustrated in Figure 5.17. Note well that this implies that `pop(0)` is the most expensive call, using $\Omega(n)$ time. (see experiments in Exercise R-5.8.)

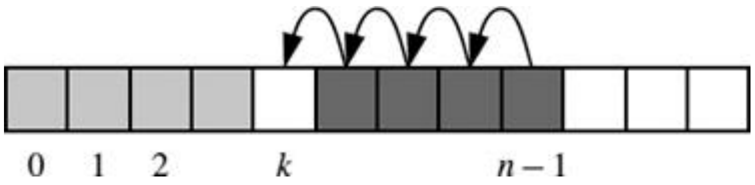


Figure 5.17: Removing an element at index *k* of a dynamic array.

The `list` class offers another method, named `remove`, that allows the caller to specify the *value* that should be removed (not the *index* at which it resides). Formally, it removes only the first occurrence of such a value from a list, or raises a `ValueError` if no such value is found. An implementation of such behavior is given in Code Fragment 5.6, again using our `DynamicArray` class for illustration.

Interestingly, there is no "efficient" case for `remove`; every call requires $\Omega(n)$ time. 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. This linear behavior can be observed experimentally (see Exercise C-5.24).

```
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:                  # found a match!
      for j in range(k, self._n − 1):        # shift others to fill gap
        self._A[j] = self._A[j+1]
      self._A[self._n − 1] = None            # help garbage collection
      self._n −= 1                           # we have one less item
      return                                 # exit immediately
  raise ValueError('value not found')        # only reached if no match
```

Code Fragment 5.6: Implementation of `remove` for our `DynamicArray` class.

Extending a List

Python provides a method named `extend` that is used to add all elements of one list to the end of a second list. In effect, 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.

In practice, the `extend` method is preferable to repeated calls to append because the constant factors hidden in the asymptotic analysis are significantly smaller. The greater efficiency of `extend` is threefold. First, 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). Second, there is less overhead to a single function call that accomplishes all the work, versus many individual function calls. Finally, 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. Exercise C-5.22 explores the relative efficiency of these two approaches experimentally.

Constructing New Lists

There are several syntaxes for constructing new lists. In almost all cases, the asymptotic efficiency of the behavior is linear in the length of the list that is created. However, as with the case in the preceding discussion of `extend`, there are significant differences in the practical efficiency.

Section 1.9.2 introduces the topic of ***list comprehension***, using an example suchas `squares = [ k*k` **for** `k` **in** `range(1, n+1) ]` as a shorthand for

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

Experiments should show that the list comprehension syntax is significantly faster than building the list by repeatedly appending (see Exercise C-5.23).

Similarly, it is a common Python idiom to 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. Not only is this succinct for the programmer; it is more efficient than building such a list incrementally.

5.4.2 PYTHON'S STRING CLASS

Strings are very important in Python. We introduced their use in Chapter 1, with a discussion of various operator syntaxes in Section 1.3. A comprehensive summary of the named methods of the class is given in Tables A.1 through A.4 of Appendix A. We will not formally analyze the efficiency of each of those behaviors in this section, but we do wish to comment on some notable issues. In general, we let n denote the length of a string. For operations that rely on a second string as a pattern, we let m denote the length of that pattern string.

The analysis for many behaviors is quite intuitive. For example, methods that produce a new string (e.g., `capitalize, center, strip`) require time that is linear in the length of the string that is produced. Many of the behaviors that test Boolean conditions of a string (e.g., `islower`) take $O(n)$ time, examining all n characters in the worst case, but short circuiting as soon as the answer becomes evident (e.g., `islower` can immediately return `False` if the first character is uppercased). The comparison operators (e.g., ==, <) fall into this category as well.

Pattern Matching

Some of the most interesting behaviors, from an algorithmic point of view, are those that in some way depend upon finding a string pattern within a larger string; this goal is at the heart of methods such as `_ _contains_ _, find, index, count, replace`, and `split`. String algorithms will be the topic of Chapter 13, and this particular problem known as ***pattern matching*** will be the focus of Section 13.2. A naive implementation runs in $O(mn)$ time case, because we consider the $n - m + 1$ possible starting indices for the pattern, and we spend $O(m)$

time at each starting position, checking if the pattern matches. However, in Section 13.2, we will develop an algorithm for finding a pattern of length *m* within a longer string of length *n* in *O*(*n*) time.

Composing Strings

Finally, we wish to comment on several approaches for composing large strings. As an academic exercise, assume that we have a large string named `document`, and our goal is to produce a new string, `letters`, that contains only the alphabetic characters of the original string (e.g., with spaces, numbers, and punctuation removed). It may be tempting to compose a result through repeated concatenation, as follows.

```
# WARNING: do not do this
letters = ' '
for c in document:          # start with empty string
 if c.isalpha():
    letters += c            # concatenate alphabetic character
```

While the preceding code fragment accomplishes the goal, it may be terribly inefficient. Because strings are immutable, the command, `letters += c`, would presumably compute the concatenation, `letters + c`, as a new string instance and then reassign the identifier, `letters`, to that result. Constructing that new string would require time proportional to its length. If the final result has *n* characters, the series of concatenations would take time proportional to the familiar sum 1 + 2 + 3 + ⋯ + *n*, and therefore $O(n^2)$ time.

Inefficient code of this type is widespread in Python, perhaps because of the somewhat natural appearance of the code, and mistaken presumptions about how the += operator is evaluated with strings. Some later implementations of the Python interpreter have developed an optimization to allow such code to complete in linear time, but this is not guaranteed for all Python implementations. The optimization is as follows. The reason that a command, `letters += c`, causes a new string instance to be created is that the original string must be left unchanged if another variable in a program refers to that string. On the other hand, if Python knew that there were no other references to the string in question, it could implement += more efficiently by directly mutating the string (as a dynamic array). As it happens, the Python interpreter already maintains what are known as ***reference counts*** for each object; this count is used in part to determine if an object can be garbage collected. (See Section 15.1.2.) But in this context, it provides a means to detect when no other references exist to a string, thereby allowing the optimization.

A more standard Python idiom to guarantee linear time composition of a string is to use a temporary list to store individual pieces, and then to rely on the `join` method of the `str` class to compose the final result. Using this technique with our previous example would appear as follows:

```
temp = [ ]                  # start with empty list
for c in document:
 if c.isalpha():
    temp.append(c)          # append alphabetic character
letters = ' ' .join(temp)  # compose overall result
```

This approach is guaranteed to run in *O*(*n*) time. First, we note that the series of up to *n* append calls will require a total of *O*(*n*) time, as per the definition of the amortized cost of that operation. The final call to `join` also guarantees that it takes time that is linear in the final length of the composed string.

As we discussed at the end of the previous section, we can further improve the practical execution time by using a list comprehension syntax to build up the temporary list, rather than by repeated calls to append. That solution appears as,

```
letters = '' .join([c for c in document if c.isalpha()])
```

Better yet, we can entirely avoid the temporary list with a generator comprehension:

```
letters = '' .join(c for c in document if c.isalpha())
```

5.5 Using Array-Based Sequences

5.5.1 STORING HIGH SCORES FOR A GAME

The first application we study is storing a sequence of high score entries for a video game. This is representative of many applications in which a sequence of objects must be stored. We could just as easily have chosen to store records for patients in a hospital or the names of players on a football team. Nevertheless, let us focus on storing high score entries, which is a simple application that is already rich enough to present some important data-structuring concepts.

To begin, we consider what information to include in an object representing a high score entry. Obviously, one component to include is an integer representing the score itself, which we identify as `_score`. Another useful thing to include is the name of the person earning this score, which we identify as `_name`. We could go on from here, adding fields representing the date the score was earned or game statistics that led to that score. However, we omit such details to keep our example simple. A Python class, `GameEntry`, representing a game entry, is given in Code Fragment 5.7.

```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)'
```

Code Fragment 5.7: Python code for a simple `GameEntry` class. We include methods for returning the name and score for a game entry object, as well as a method for returning a string representation of this entry.

A Class for High Scores

To maintain a sequence of high scores, we develop a class named `Scoreboard`. A scoreboard is limited to a certain number of high scores that can be saved; once that limit is reached, a new score only qualifies for the scoreboard if it is strictly higher than the lowest “high score” on the board. The length of the desired scoreboard may depend on the game, perhaps 10, 50, or 500. Since that limit may vary depending on the game, we allow it to be specified as a parameter to our `Scoreboard` constructor.

Internally, we will use a Python `list` named `_board` in order to manage the `GameEntry` instances that represent the high scores. Since we expect the scoreboard to eventually reach full capacity, we initialize the list to be large enough to hold the maximum number of scores, but we initially set all entries to `None`. By allocating the list with maximum capacity initially, it never needs to be resized. As entries are added, we will maintain them from highest to lowest score, starting at index 0 of the list. We illustrate a typical state of the data structure in Figure 5.18.

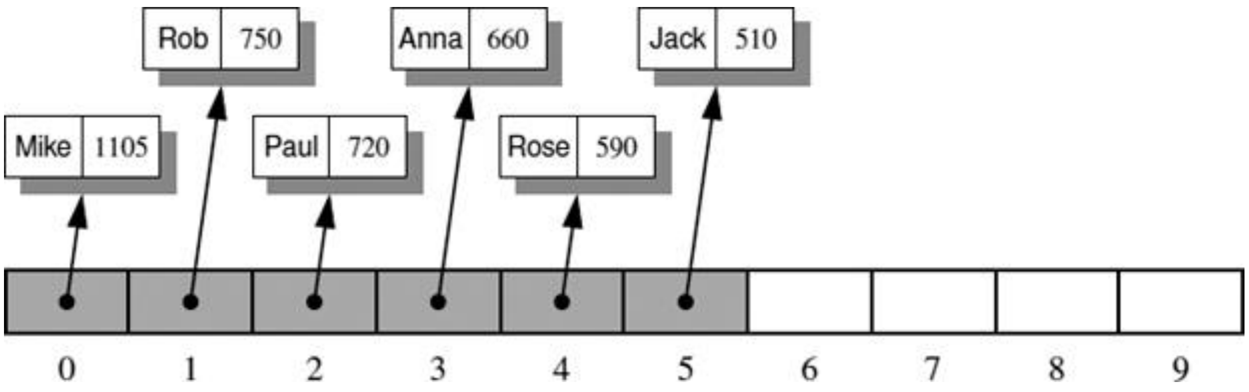


Figure 5.18: An illustration of an ordered list of length ten, storing references to six `GameEntry` objects in the cells from index 0 to 5, with the rest being `None`.

A complete Python implementation of the `Scoreboard` class is given in Code Fragment 5.8. The constructor is rather simple. The command

```
  self._board = [None] * capacity
```

creates a list with the desired length, yet all entries equal to `None`. We maintain an additional instance variable, `_n`, that represents the number of actual entries currently in our table. For convenience, our class supports the _ `_getitem _ _` method to retrieve an entry at a given index with a syntax `board[i]` (or `None` if no such entry exists), and we support a simple _ `_str_ _` method that returns a string representation of the entire scoreboard, with one entry per line.

```python
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

  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))

  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
```

Code Fragment 5.8: Python code for a `Scoreboard` class that maintains an ordered series of scores as `GameEntry` objects.

Adding an Entry

The most interesting method of the `Scoreboard` class is `add`, which is responsible for considering the addition of a new entry to the scoreboard. Keep in mind that every entry will not necessarily qualify as a high score. If the board is not yet full, any new entry will be retained. Once the board is full, a new entry is only retained if it is strictly better than one of the other scores, in particular, the last entry of the scoreboard, which is the lowest of the high scores.

When a new score is considered, we begin by determining whether it qualifies as a high score. If so, we increase the count of active scores, `_n`, unless the board is already at full capacity. In that case, adding a new high score causes some other entry to be dropped from the scoreboard, so the overall number of entries remains the same.

To correctly place a new entry within the list, the final task is to shift any inferior scores one spot lower (with the least score being dropped entirely when the scoreboard is full). This process is quite similar to the implementation of the `insert` method of the `list` class, as described on pages 204–205. In the context of our scoreboard, there is no need to shift any `None` references that remain near the end of the array, so the process can proceed as diagrammed in Figure 5.19.

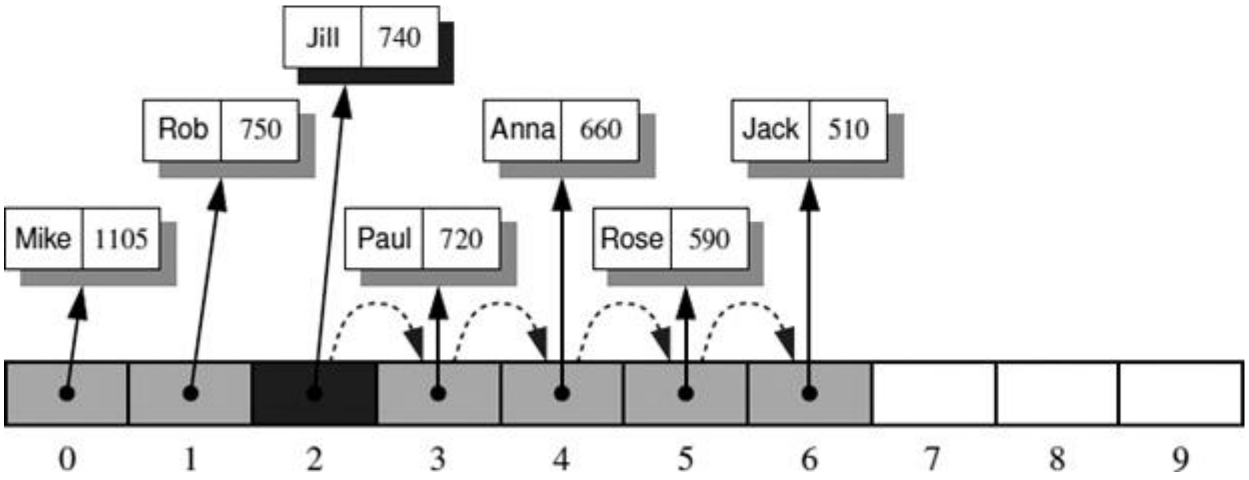


Figure 5.19: Adding a new `GameEntry` for Jill to the scoreboard. In order to make room for the new reference, we have to shift the references for game entries with smaller scores than the new one to the right by one cell. Then we can insert the new entry with index 2.

To implement the final stage, we begin by considering index `j = self._n − 1`, which is the index at which the last `GameEntry` instance will reside, after completing the operation. Either *j* is the correct index for the newest entry, or one or more immediately before it will have lesser scores. The while loop at line 34 checks the compound condition, shifting references rightward and decrementing *j*, as long as there is another entry at index $j - 1$ with a score less than the new score.

5.5.2 SORTING A SEQUENCE

In the previous subsection, we considered an application for which we added an object to a sequence at a given position while shifting other elements so as to keep the previous order intact. In this section, we use a similar technique to solve the ***sorting*** problem, that is, starting with an unordered sequence of elements and rearranging them into nondecreasing order.

The Insertion-Sort Algorithm

We study several sorting algorithms in this book, most of which are described in Chapter 12. As a warm-up, in this section we describe a nice, simple sorting algorithm known as ***insertion-sort***. The algorithm proceeds as follows for an array-based sequence. We start with the first element in the array. One element by itself is already sorted. Then we consider the next element in the array. If it is smaller than the first, we swap them. Next we consider the third element in the array. We swap it leftward until it is in its proper order with the first two elements. We then consider the fourth element, and swap it leftward until it is in the proper order with the first three. We continue in this manner with the fifth element, the sixth, and so on, until the whole array is sorted. We can express the insertion-sort algorithm in pseudo-code, as shown in Code Fragment 5.9.

```
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.

This is a simple, high-level description of insertion-sort. If we look back to Code Fragment 5.8 of Section 5.5.1, we see that the task of inserting a new entry into the list of high scores is almost identical to the task of inserting a newly considered element in insertion-sort (except that game scores were ordered from high to low). We provide a Python implementation of insertion-sort in Code Fragment 5.10, using an outer loop to consider each element in turn, and an inner loop that moves a newly considered element to its proper location relative to the (sorted) subarray of elements that are to its left. We illustrate an example run of the insertion-sort algorithm in Figure 5.20.

The nested loops of insertion-sort lead to an $O(n^2)$ running time in the worst case. The most work is done if the array is initially in reverse order. On the other hand, if the initial array is nearly sorted or perfectly sorted, insertion-sort runs in $O(n)$ time because there are few or no iterations of the inner loop.

```
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
```

Code Fragment 5.10: Python code for performing insertion-sort on a list.

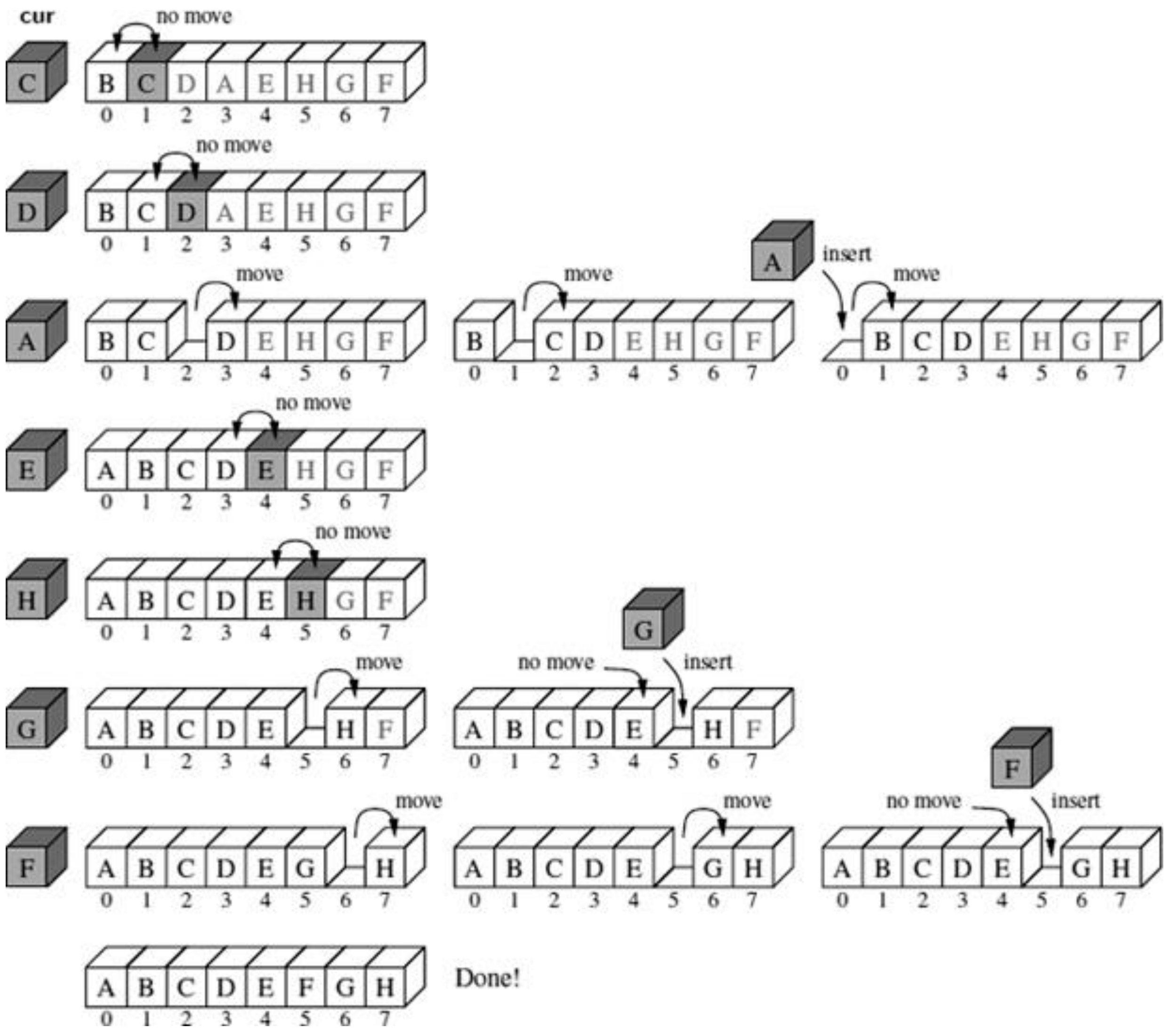


Figure 5.20: Execution of the insertion-sort algorithm on an array of eight characters. Each row corresponds to an iteration of the outer loop, and each copy of the sequence in a row corresponds to an iteration of the inner loop. The current element that is being inserted is highlighted in the array, and shown as the `cur` value.