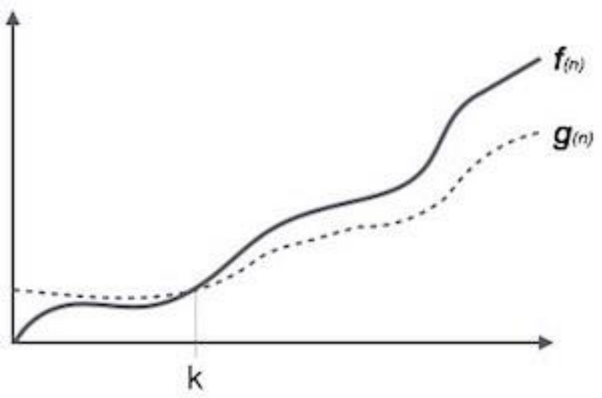
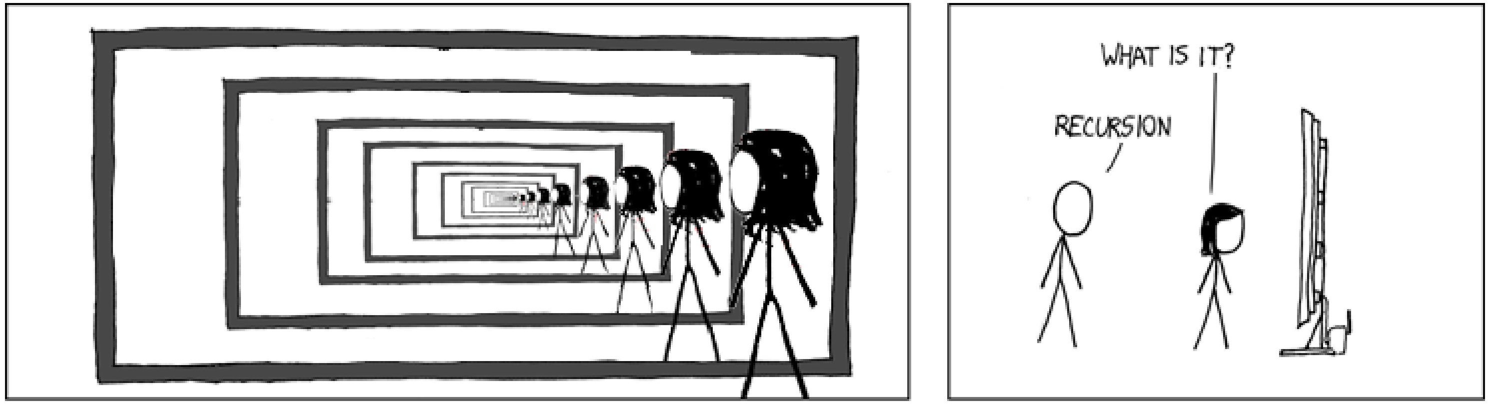
ARRAY BASED SEQUENCES

School of Artificial Intelligence

PREVIOUSLY ON DS&A

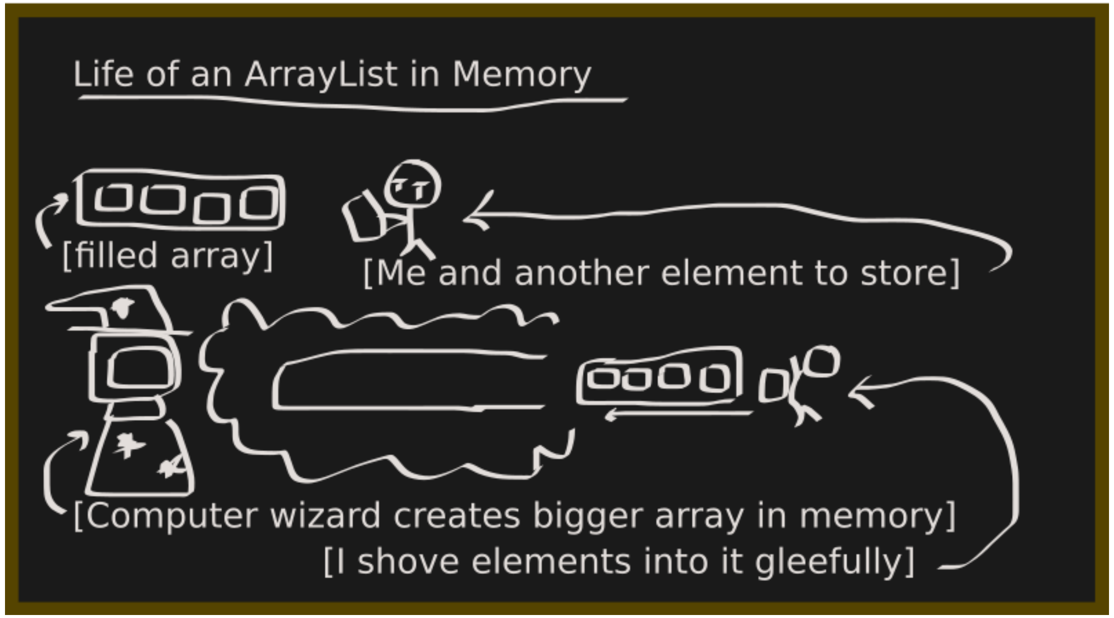
- Asymptotic Analysis
  - Big-O notation
  - Big-Theta notation
  - Big-Omega notation
- Recursion
  - Base case(s)
  - General Rule(s)
- Recursion analysis
  - # primitive operations
  - # of invocations
- Types of recursion
  - Linear
  - Binary
  - Multiple





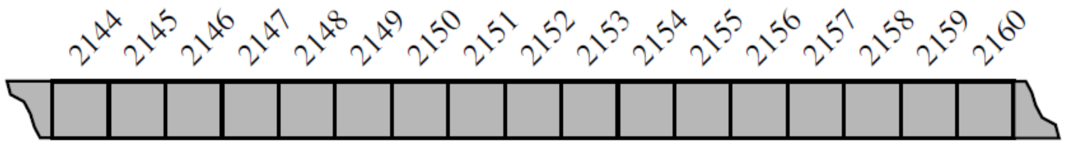
UP NEXT

- Array-Based Sequences
  - List
  - Tuple
  - String
- Public behaviours
- Implementation Details
- Asymptotic and Experimental Analyses

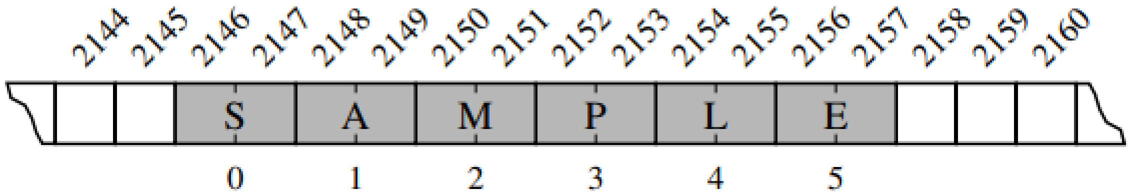


LOW-LEVEL ARRAYS

- Computer memory
  - Small ‘chunks’ of memory – bytes (8 bit)
  - Each chunk: a unique number – **memory address**
  - Access of a memory location: O(1)
    - Random Access Memory (RAM)



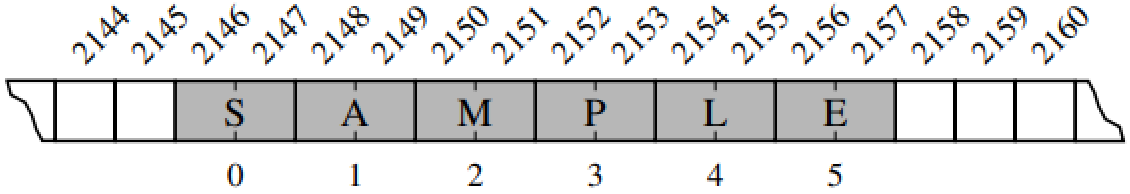
- Programming language
  - Identifier -> memory location
  - x -> byte #8086
  - y -> byte #80286
  - A sequence of values of the same type: array



LOW-LEVEL ARRAYS

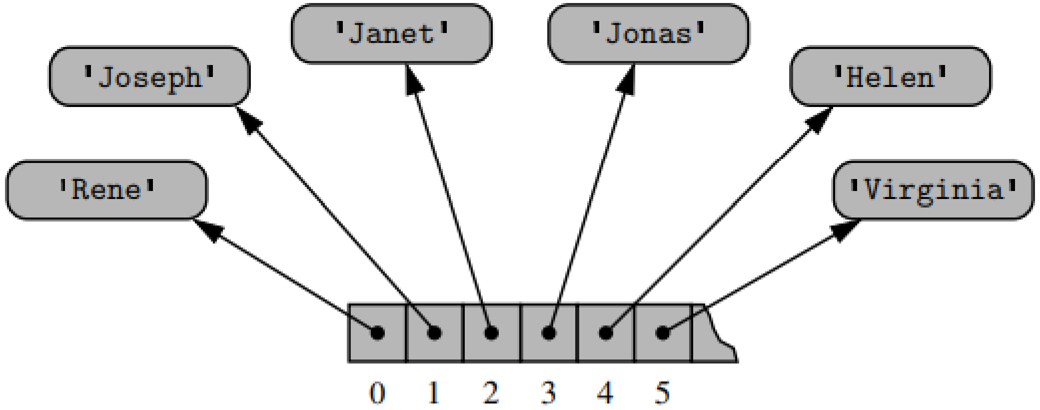
- Example:
- Array of 6 characters with Unicode encoding
- 12 bytes because Python uses 16 bits to represent Unicode
- Each location within an array: **cell**
- Integer to describe the location within the array: **index**
- Each cell must use the same number of bytes
  - Cells can be accessed in O(1)

  start + cellsize * index



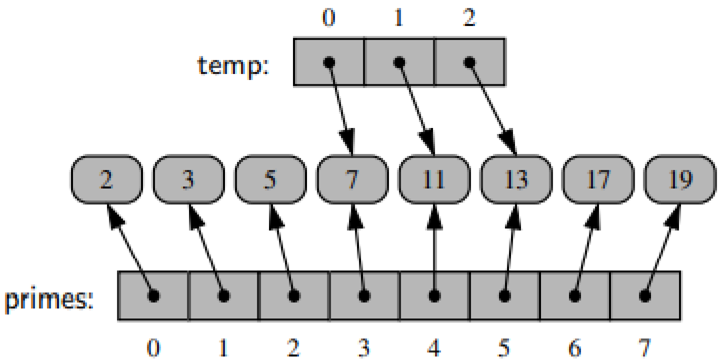
REFERENTIAL ARRAYS

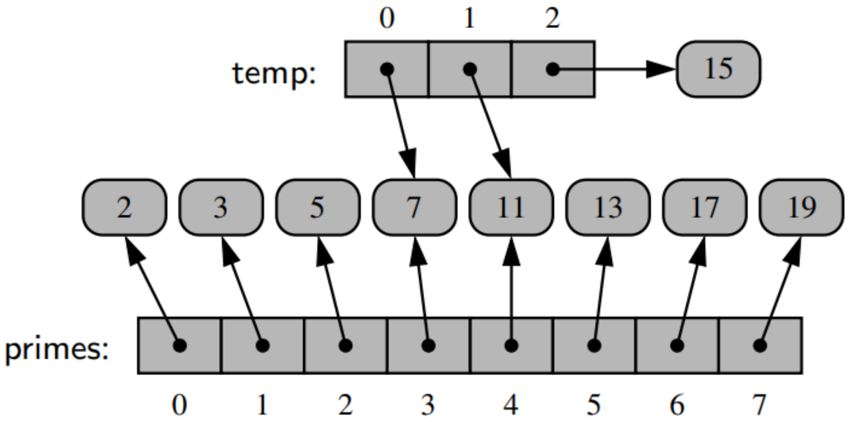
- Referential Arrays
  - A list of names ['Rene', 'Joseph', 'Janet', 'Jonas', 'Helen', 'Virginia', ...]
- Remember: each cell must use the same number of bytes
  - One approach: reserve enough space for each cell to hold a String with the maximum length – problem?
  - Python's approach: each cell stores a **reference** to an object
    - Sequence of memory addresses – benefits?
    - Size of each cell? (32/64 bits)
    - Reference to **None** represents an empty cell
  - What if we want to store more than names?



REFERENTIAL ARRAYS

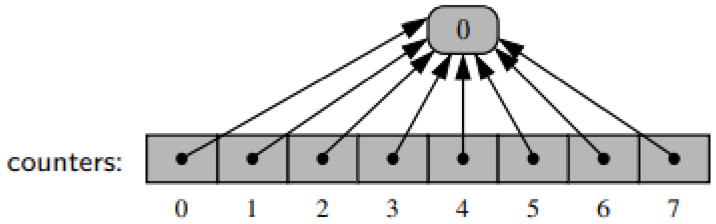
- Referential Arrays: significant to the semantics of sequence classes
  - A single list instance may include multiple references to the same object as elements of the list
  - A single object to be an element of multiple lists
  - Compute a slice of a list: new list instance, but points to the same elements
  - When elements are immutable objects
    - No problem

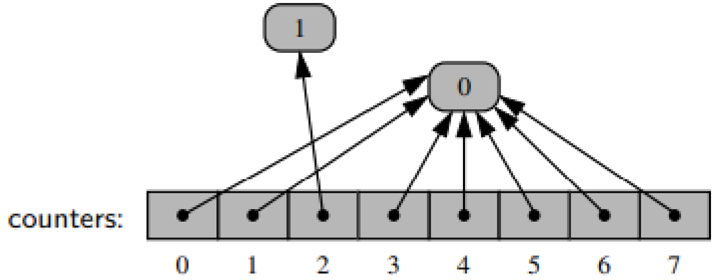




REFERENTIAL ARRAYS

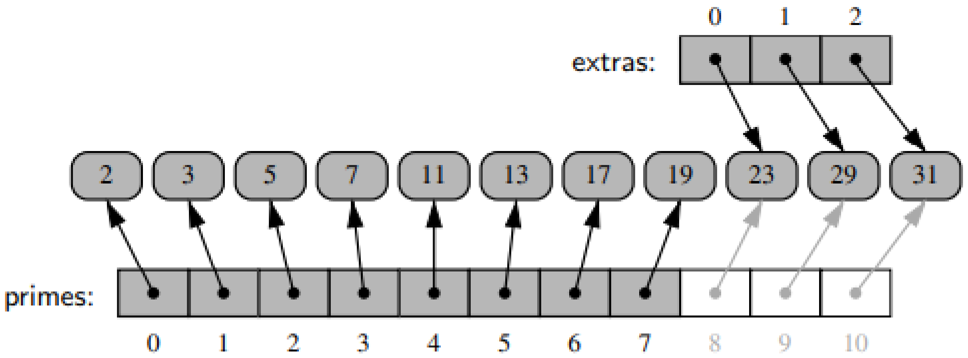
- Referential Arrays: significant to the semantics of sequence classes
  - List with immutable objects
    - Making a new list as a copy of an existing one
    - backup = list(primes)
    - **shallow copy**: it references the same elements as in the first list
  - List with mutable objects
    - backup = list(primes)
    - **deep copy**: a new list with new elements (different objects) is produced
- Initialisation of an array
  - counters = [0] * 8
  - Referenced integer is immutable
  - counters[2] += 1 does not alter the value but computes a new integer





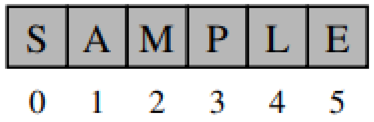
REFERENTIAL ARRAYS

- extend(): add all elements from one list to the end of another
  - primes.extend(extra)
  - Extended list receives references to elements added to it
  - More on complexity of extend() later
  - extend() is not generally supported in other programming languages
    - Especially in real-time/safety-critical systems



COMPACT ARRAYS

- Compact Arrays
  - String: an array of characters
  - **Compact arrays**
    - Stores the bits that represent the primary data
      - Characters in this case
  - Advantages in computing performance
    - Lower overall memory usage
      - Referential arrays: 64 bits to store memory address + memory used by the referenced objects
      - Compact arrays: 2 bytes (for characters)



COMPACT ARRAYS

- Advantage: lower overall memory usage
  - Consider: storing a sequence of one million, 64-bit integers with referential arrays
  - 64 million bits?
  - No, 4-5 times as much in Python
    - Each cell: 64-bit for memory address + int instance else where in memory
    - But Python uses 14 bytes for int objects (some other states take additional space)
- Advantage: primary data stored consecutively
  - Crucial for high performance computing
  - Intel’s ArBB library: data parallelism
  - Not case for referential structure

PYTHON SUPPORT FOR COMPACT ARRAYS

- Module **array**
- Constructor requires a **type code**

2	3	5	7	11	13	17	19
0	1	2	3	4	5	6	7

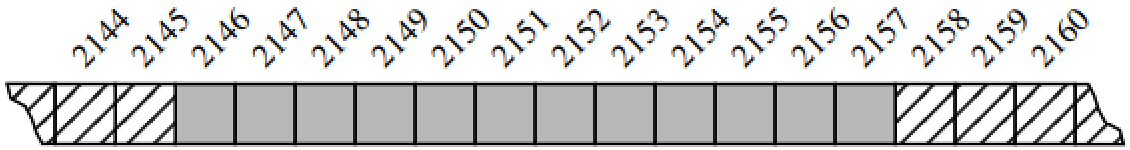
```
primes = array('i', [2, 3, 5, 7, 11, 13, 17, 19])
```

- Type code enables the interpreter to compute the size (# of bits) needed for each element in the array
- No support for user-defined data types
  - Unlike C/C++

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 AND AMORTISATION

- When creating (initialising) an array, the precise size of it must be declared for the system to allocate a consecutive piece of memory
    - E.g. array of 12 bytes



- However
    - 2158 may be allocated by the system
    - NOT trivial to 'grow' the array by simply extending to subsequence cells
    - Not a problem for Python **tuple** or **str**
        - Because they are immutable
- Python's list class
    - Allows us to add elements to the list, with no apparent limit on the overall capacity
    - **Dynamic array**

DYNA S

- Key idea
  - A list maintains an underlying array that often has greater
    - When create a list with 5 elements, the system may have re array of 8 elements
    - So that it is easy to append a new element by using the nex
  - When the reserved capacity is exhausted
    - Need a new, larger array from the system
    - Copy over the elements of the old array to the new one
    - Old array is reclaimed by the system
- We can write a program to test this

```
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
```

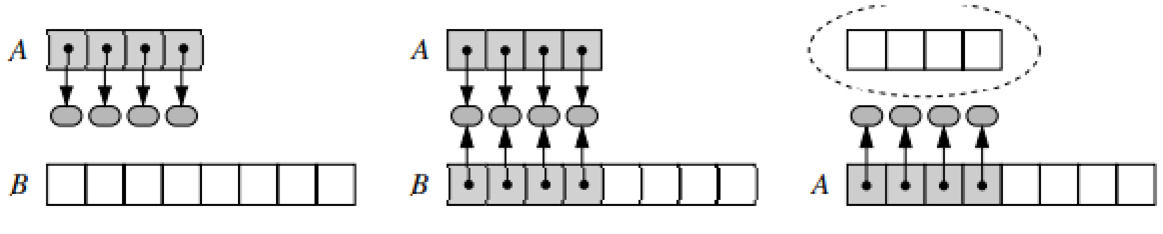
Length:	0;	Size in bytes:	72
Length:	1;	Size in bytes:	104
Length:	2;	Size in bytes:	104
Length:	3;	Size in bytes:	104
Length:	4;	Size in bytes:	104
Length:	5;	Size in bytes:	136
Length:	6;	Size in bytes:	136
Length:	7;	Size in bytes:	136
Length:	8;	Size in bytes:	136
Length:	9;	Size in bytes:	200
Length:	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

- Empty list already requires a certain number of bytes
  - Each object maintains some state
  - _n: # of actual elements currently stored
  - _capacity: maximum # of element that can be stored
  - _A: reference to the currently allocated array
- When 1st element is added
  - Change in the underlying size of the structure
    - 32 bytes (4x8 bytes)
    - Reserved for 4 elements
- When 5th element is added
  - 64 bytes increase: reserved for 8 elements
- 9th insertion
  - 128 bytes increase

```
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
```

IMPLEMENT A DYNAMIC ARRAY

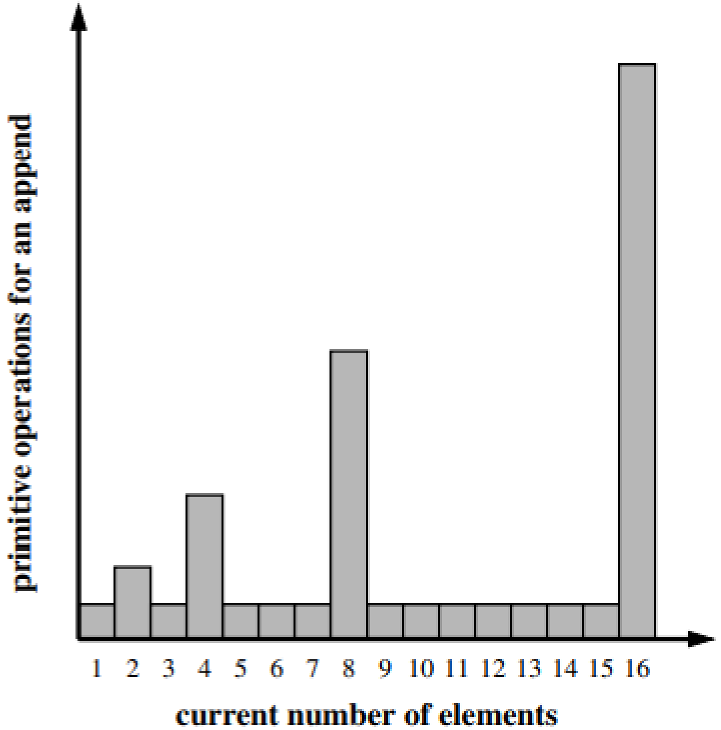
- Task: grow array A that stores the element of a list
  - Note: we are not really growing that array, why?
- When an element is appended to a list when A is full
  1. Allocate a new array B with larger capacity
  2. Set B[i] = A[i] for i = 0, …, n-1; where n is size of A
  3. Set A = B
  4. Insert new elements in A



(a) (b) (c)

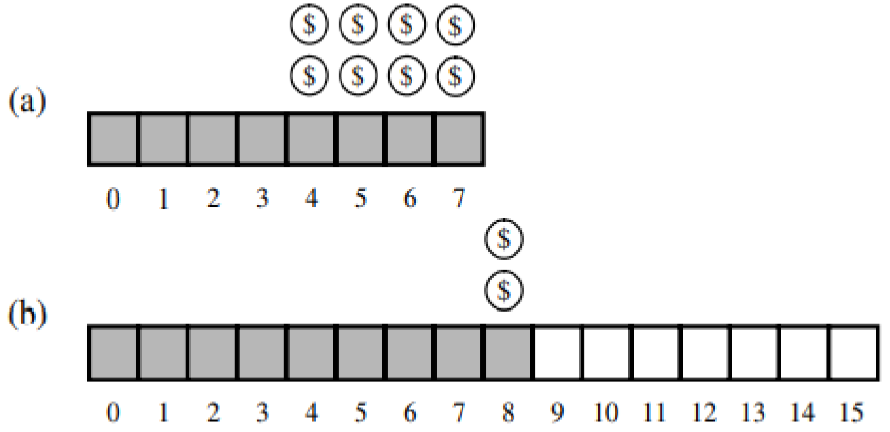
AMORTISATION (摊销)

- Question: how large should the new array be?
  - Twice the capacity?
- Complexity to 'grow' an array A to twice of its size?
  - Create a new array twice the size
  - Copying old elements into the new array
  - O(n)
- However
  - After 'growing'
  - Each append() takes O(1) time



AMORTISATION (摊销)

- Let S be a dynamic array with initial capacity 1, with table doubling, the total time to perform a series of n append() operation in S, is O(n)
- Justification: cyber-coin analogy
  - Computer as a coin-operated machine
    - 1 coin for append() excluding the time spent for growing the array
    - Growing the array from k to 2k requires k cyber coins
  - Strategy:
    - charge each append() 3 coins (we are overcharging) and store unspent coins
    - When 'overflow' occurs (S has 2^i elements), doubling requires 2^i coins, use our 'stored' coins from 2^{i-1} to $2^i - 1$
  - We can pay for the execution of n append() with 3n cyber coins
  - Running time for each append(): O(1)
  - Total running time: O(n)

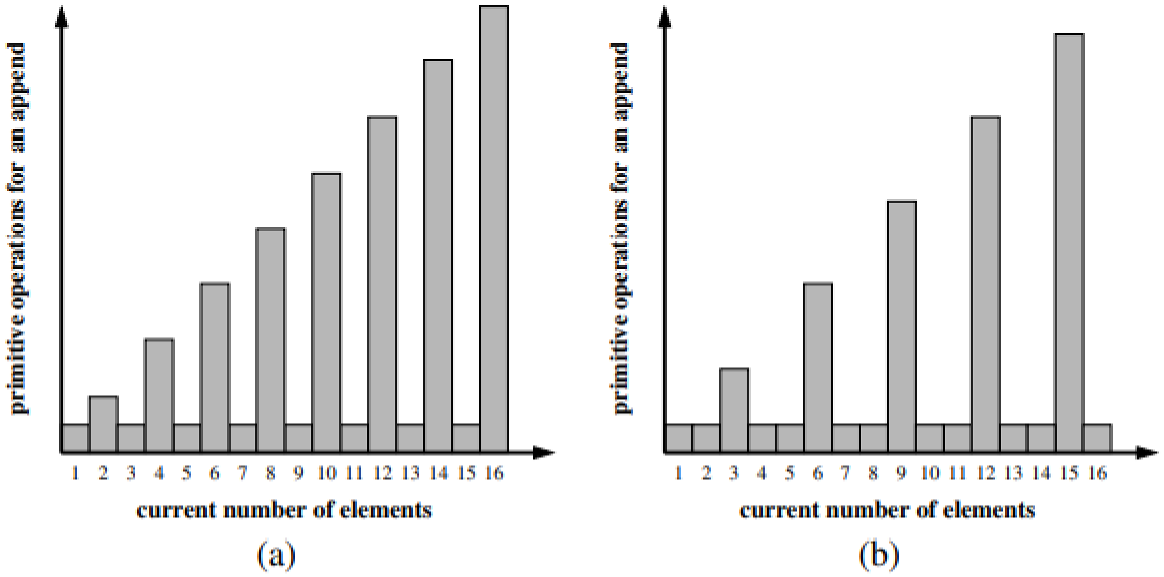


AMORTISATION (摊销)

- Amortisation (摊销)
- We used base of 2 (table doubling)
- What if we use base of 1.25 (grows by 25% each time we grow)?
  - OK but more intermediate resize events
  - Still possible to prove an O(1) armotised bound for append()
    - Homework: 如何以1.25为底（25% 增幅）证明append()函数执行复杂度是 O(1) amortised?

AMORTISATION (摊销)

- Arithmetic progression (等差数列)?
- Each time the array grows, only reserve a constant number of additional cells
- E.g. increase of 2 in size vs increase of 3 in size

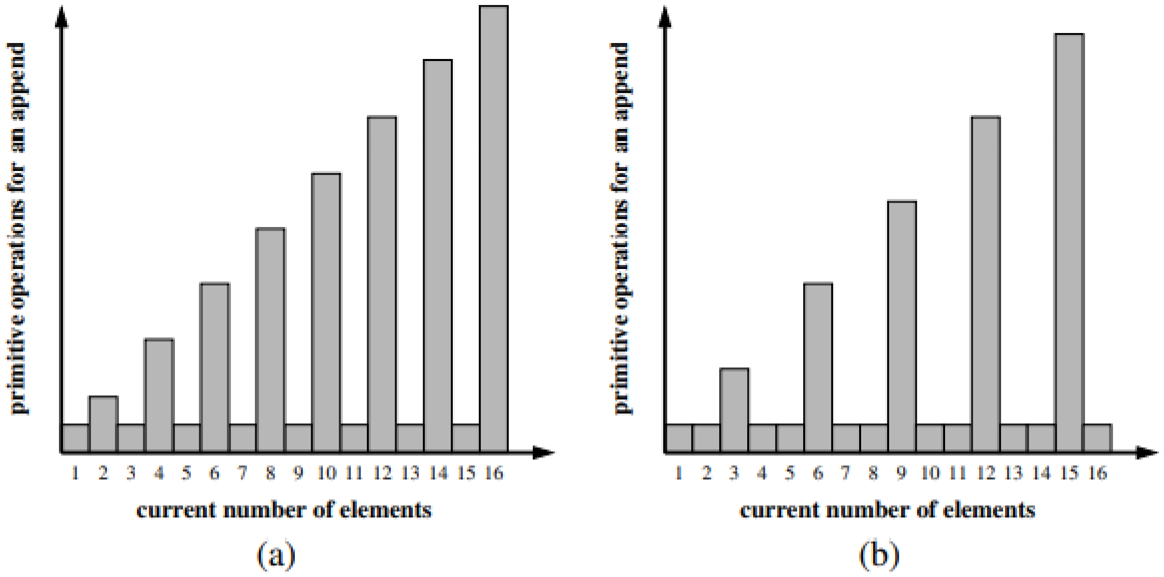


AMORTISATION (摊销)

- Proposition: performing a series of n append operations on an empty dynamic array using a fixed increment with each resize takes $\Omega(n^2)$ time
- Justification: c = fixed increment in capacity (c>0)
  - For n append operations, time will be spent when the array is of size c, 2c, 3c, …mc for m = ceiling(n/c), the overall time is:

$$\sum_{i=1}^{m} ci = c \cdot \sum_{i=1}^{m} i = c\frac{m(m+1)}{2} \geq c\frac{\frac{n}{c}(\frac{n}{c}+1)}{2} \geq \frac{n^2}{2c}.$$

  - Therefore n append() takes $\Omega(n^2)$



SHRINKING AN ARRAY

- When enough elements are deleted, e.g. using a pop()
- Pointless to keep the size of the array as big
- What’s the strategy of shrinking an array?
- Reduce by half?

PYTHON'S LIST CLASS

- Amortised O(1) for append()
- We can test this in Python

```
from time import time                     # import time function from time module
def compute_average(n):
  """Perform n appends to an empty list and return average time elapsed."""
  data = [ ]
  start = time( )                         # record the start time (in seconds)
  for k in range(n):
    data.append(None)
  end = time( )                           # record the end time (in seconds)
  return (end − start) / n                # compute average per operation
```

n	100	1,000	10,000	100,000	1,000,000	10,000,000	100,000,000
μs	0.219	0.158	0.164	0.151	0.147	0.147	0.149

EFFICIENCY OF PYTHON'S SEQUENCE TYPES

- Constant operations
    - len(data): O(1)
    - data[j]: O(1)
- Searching for occurrences of a value
    - data.count(value): O(n), n = size of array
    - data.index(value): O(k), k = leftmost occurrence
    - value **in** data: O(k), k = leftmost occurrence
- Comparisons
    - data1 == data2 O(k), k = min(n1, n2)
- Creating new instances
    - data[j:k]: O(k-j)
    - data1 + data2: O(n1 + n2)
    - c * data = O(cn)
- **Non-mutating behaviours**

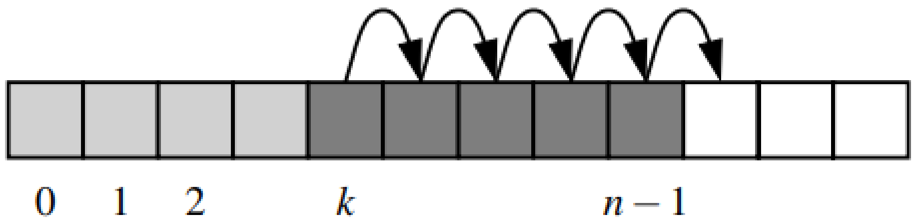
EFFICIENCY OF PYTHON'S SEQUENCE TYPES

- Changing values in the list
  - data[j] = val <u>O(1)</u>
  - data.append(value) <u>O(1) amortised</u>
  - data.insert(k, value) <u>O(n-k) amortised</u>
  - data.pop() <u>O(1) amortised</u>
  - data.pop(k) <u>O(n-k) amortised</u>
  - **del** data[k] <u>O(n-k) amortised</u>
  - data.remove(value) <u>O(n) amortised</u>
  - data1.extend(data2) <u>O(n) amortised</u>
  - data1 += data2 <u>O(n2) amortised</u>
  - data.reverse() <u>O(n)</u>
  - data.sort() <u>O(n log n)</u>

EFFICIENCY OF PYTHON'S SEQUENCE TYPES

- Adding elements to a list
    - Worst case Ω(n) time due to resize
    - O(1) amortised
- insert(k, value)
    - Insert value at k, where 0 <= k <= n
    - Shifting all elements to the right cell
    - Things that affect efficiency
        - Addition of one element may cause a resize
        - Shifting of elements to make room for the new item
    - Complexity?

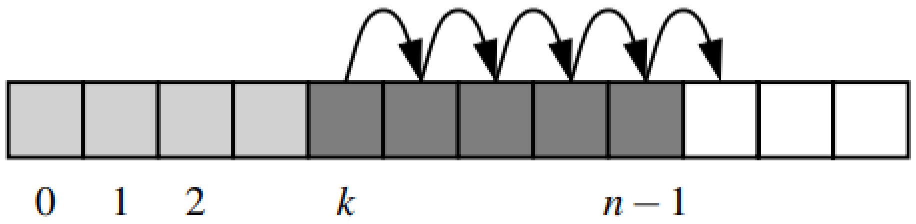
```
def insert(self, k, value):
  """Insert value at index k, shifting subsequent values rightward."""
  # (for simplicity, we assume 0 <= k <= n in this verion)
  if self._n == self._capacity:                  # not enough room
    self._resize(2 * self._capacity)             # so double capacity
  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
```



EFFICIENCY OF PYTHON'S SEQUENCE TYPES

- Adding elements to a list
  - Worst case Ω(n) time due to resize
  - O(1) amortised
- insert(k, value)
  - Insert value at k, where 0 <= k <= n
  - Shifting all elements to the right cell
  - Things that affect efficiency
    - Addition of one element may cause a resize
    - Shifting of elements to make room for the new item
  - Complexity?
    - O(n-k+1) amortised

```
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
```



EFFICIENCY OF PYTHON'S SEQUENCE TYPES

- Experiments with insert()
  - Case 1: insert at the beginning of the list

```
for n in range(N):
  data.insert(0, None)
```

  - Case 2: insert near the middle of the list

```
for n in range(N):
  data.insert(n // 2, None)
```

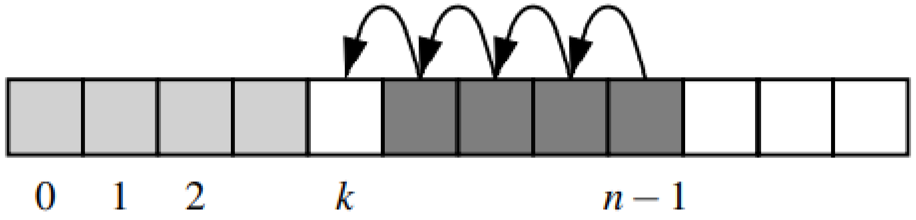
  - Case 3: insert at the end of the list

```
for n in range(N):
  data.insert(n, None)
```

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

EFFICIENCY OF PYTHON'S SEQUENCE TYPES

- Removing elements from a list
  - pop(): O(1)
    - Is it amortised?
  - pop(k)
    - Removes index at k, where k < n
    - Shifts all elements from k to n-1 to one cell to the left
    - O(n-k)
  - pop(0)?
    - Ω(n)
- remove(value)
  - Removes first occurrence only
  - Error when value is not in the list
  - Ω(n)




```
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
```

EFFICIENCY OF PYTHON'S SEQUENCE TYPES

- Extending a list
  - extend(): data.extend(other)
  - Equivalent to:
    ```
    for element in other:
      data.append(element)
    ```
  - O(k) amortised, k: # of elements in other
  - Preferable to data.append() repeatedly
    - Why?
- Constructing new lists
  - O(n): n # of elements to be created
  - squares = [k*k for k in range(1, n+1)] **vs.**
  - append() is significantly slower than [0]*n

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

QUIZ FOR THIS WEEK

- Problem setting
  - A frog
  - A stair case with 100 steps
  - A frog can jump 1 step or 2 steps max, at a time
- Question
  - # of ways that the frog can jump to 100?
  - Can you write a program to compute it?

QUIZ FOR THIS WEEK

- Can you use rand7() to implement rand10()
  - rand7() produces a random number from 0-7
  - rand10() produces a random number from 0-10
  - Pure randomness vs. pseudo randomness

THANKS

See you in the next session!