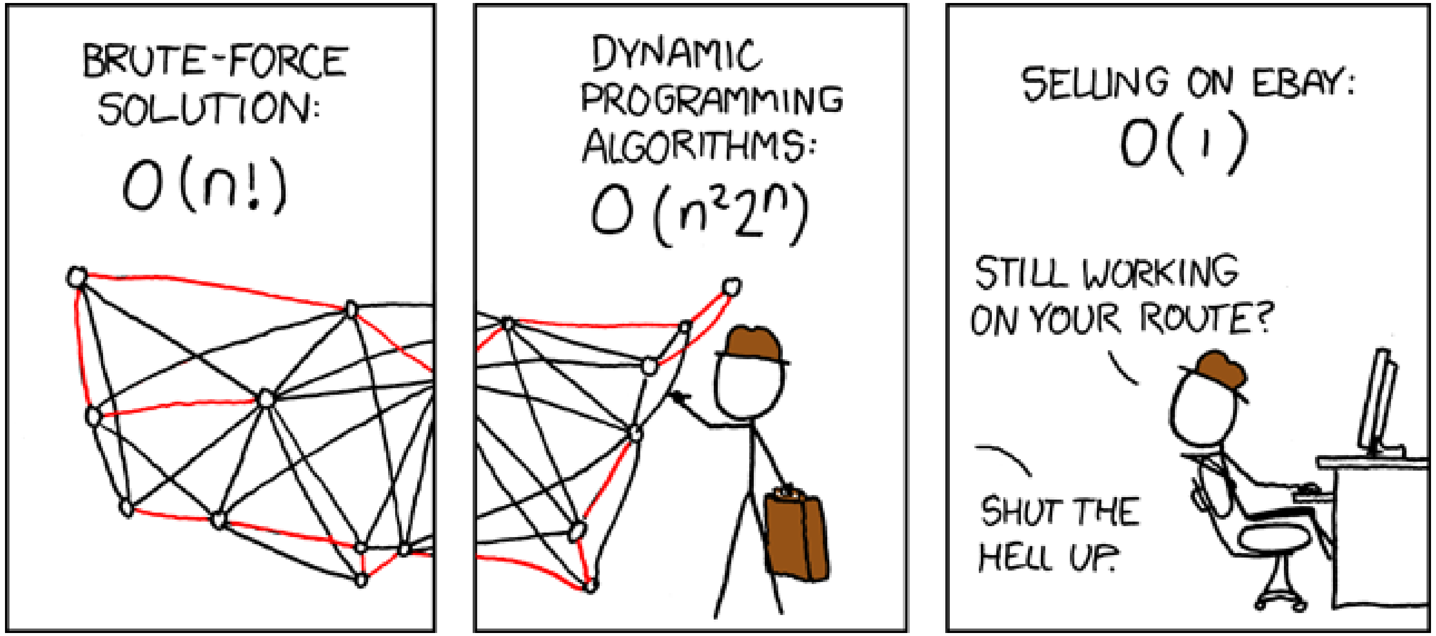
PERIODIC REVIEW

School of Artificial Intelligence

WHAT WE HAVE COVERED SO FAR

- Algorithmic analysis
  - Experimental Studies
  - Big-O Notation
- Recursion
  - Recursive algorithm
  - Recursion analysis
  - Types of Recursion
- Arrays
  - Referential Arrays
  - Compact Arrays
  - Dynamic Arrays
  - Armotisation
  - Sorting
- Stack, Queues and Deques



FIRST THINGS FIRST

- Pseudocode （伪代码） – 算法描述语言
  - 目的：为了是被描述的算法可以轻易的以任何一种变成语言实现
  - 目标：够清晰、代码简单、可读性强、类似自然语言
- 规则：
  - 算法中出现的数组、变量可以是以下类型：整数、实数、字符或指针。通常类型需要声明
  - 算法中的某些指令或子任务可以用文字来描述，如：“设max是A中的最大项”，这里A是数组；或者“将x入栈S中”，这里S是栈
  - 运算符：+，-，*，/，以及^。逻辑运算符：==, !=, <, >, <= 和 >=
  - 赋值： a <- b 或者 a = b
  - 变量呼唤需借助temp，使算法更易理解
  - 避免使用goto或者jump语句
  - 用{ }表示当前算法作用域

FIRST THINGS FIRST

- 规则：
  - While loop:
    ```
    While(condition) {
    }
    ```
  - For loop:
    ```
    for(int i = 0; i < n, i++) {
    }
    ```
  - Return
  - 注释
    - 单行：//
    - 多行：/* */

PRACTICE

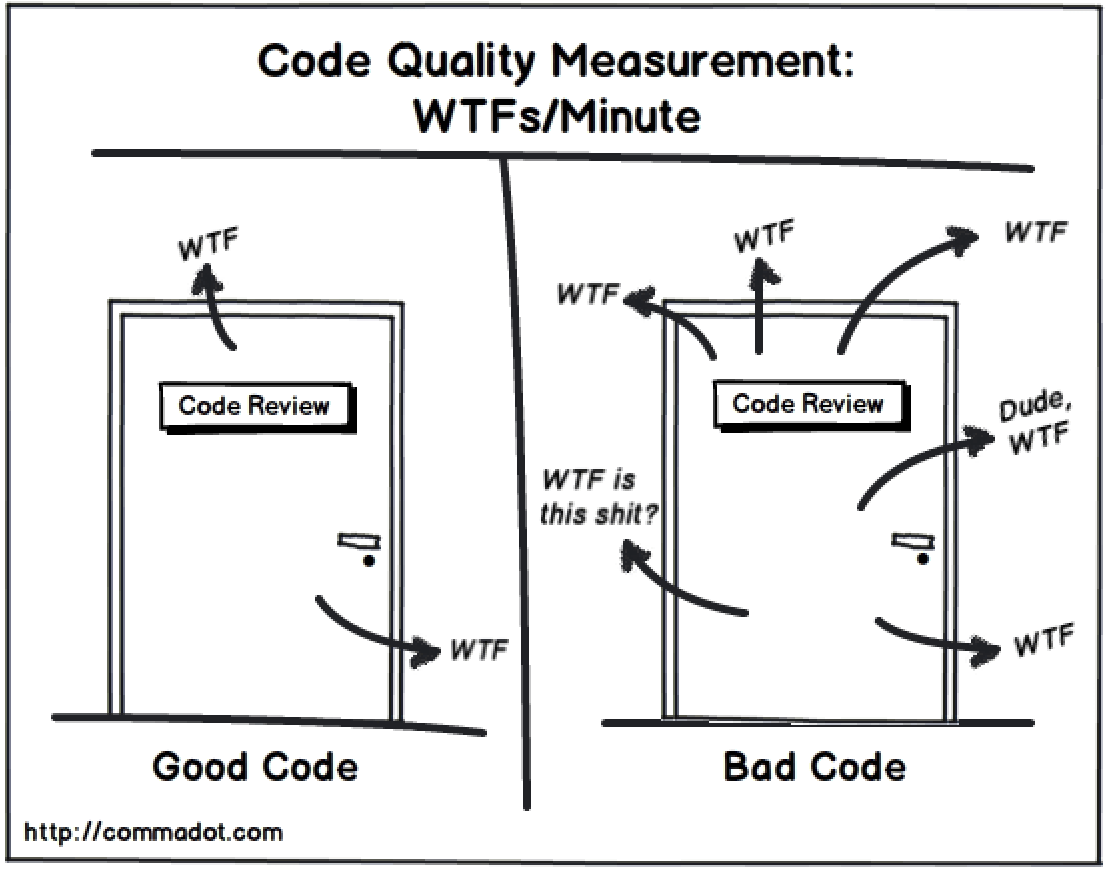
- 写出一个算法，找出一个数组中是否有重复的元素

```
Function unique(S) {
        for (j = 0; j < len(S), j++) {
                for(k = j+1; k < len(S), k++) {
                        if(S[j] == S[k]) {
                                return false;
                        }
                }
                return true;
        }
```

ALGORITHM ANALYSIS 算法分析

- 衡量算法质量的方法
  - Good vs. Poor？
- Experimental Studies （实验性分析）

```
from time import time
start_time = time( )                    # record the starting time
run algorithm
end_time = time( )                      # record the ending time
elapsed = end_time − start_time         # compute the elapsed time
```



ALGORITHM ANALYSIS 算法分析

- Experimental Studies （实验性分析）
- Limited set of test inputs
  - Input may not be the worst case
  - Algorithm may not scale
- Directly comparing two algorithms are difficult
  - Same hardware and software
  - Same CPU activities
- An algorithm must be fully implemented in order to execute

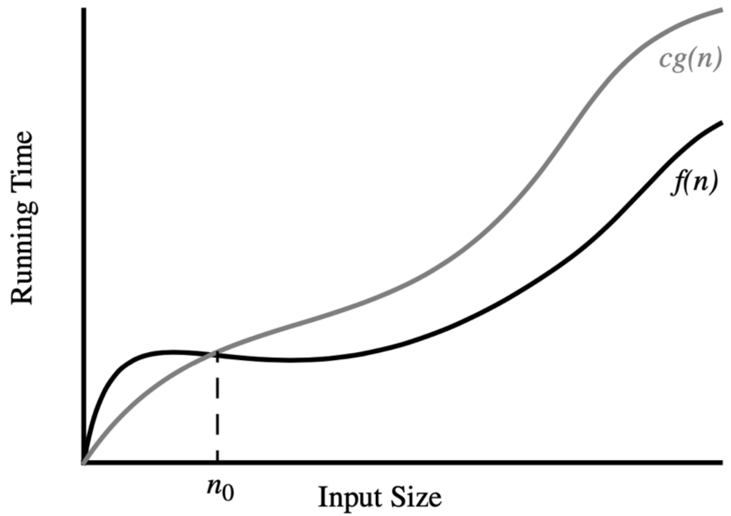
```
from time import time
start_time = time( )                      # record the starting time
run algorithm
end_time = time( )                        # record the ending time
elapsed = end_time − start_time           # compute the elapsed time
```

ALGORITHM ANALYSIS 算法分析

- Asymptotic analysis （渐进性分析）
- Counting primitive operations （原始操作）
    - Assignment （赋值）
    - Obtaining an object by an identifier （标识符）
    - Arithmetic operations （运算操作）: +, -, *, /, %, 等
    - Comparing two numbers
    - Accessing an element of an array by index
    - Calling a function （不包括函数内部执行时间）
    - Returning from a function

ALGORITHM ANALYSIS
算法分析

- Asymptotic analysis （渐进性分析）
- Best case vs. worst case?
- Big-O notation:
  - Let $f(n)$ and $g(n)$ be functions mapping positive integers to positive real numbers. We say that $f(n)$ is $O(g(n))$ if there is a real constant c>0 and an integer constant n_0 >= 1 such that



ALGORITHM ANALYSIS 算法分析

```
def find_max(data):
    """Return the maximum element from a nonempty Python list."""
    biggest = data[0]                    # The initial value to beat
    for val in data:                     # For each value:
        if val > biggest                 # if it is greater than the best so far,
            biggest = val                # we have found a new best (so far)
    return biggest                       # When loop ends, biggest is the max
```

- Asymptotic analysis （渐进性分析）
- Best case vs. worst case?
- Big-O notation:
  - Function 8n + 5 is O(n)
  - Find a constant c > 0 and an integer constant $n_0 \geq 1$
    - $8n+5 \leq cn$ for every $n \geq n_0$
    - c = 9 and $n_0 = 5$
  - Big-O: f(x) is “less than or equal to” another function g(n) up to a constant factor and in the asymptotic sense as n grows towards infinity
  - $f(n) = O(g(n))$
  - “$f(n)$ is $O(g(n))$ “

ALGORITHM ANALYSIS 算法分析

- Asymptotic analysis （渐进性分析）
- We can ignore constant factors and lower-order terms when talking about asymptotic complexity
- $5n^4 + 3n^3 + 2n^2 + 4n^1$ is $O(n^4)$
  - $5n^4 + 3n^3 + 2n^2 + 4n^1 \leq (5+3+2+4)\mathrm{n}^4 = \mathrm{cn}^4$
- Characterising functions in simplest terms
  - $f(n) = 4n^3 + 3n^2$ is $O(n^5)$ or $O(n^4)$
  - It is more accurate to say $f\,(\mathrm{n}) = O(n^3)$

BIG-OMEGA AND BIG-THETA

- Big-Omega
  - Big Omega is "greater than or equal to"
    - Normally referred to as the "upper bound"
  - $f(n)$ is $\Omega(g(n))$ if there is a real constant c>0 and an integer constant n_0 >= 1

$$f(n) \geq c\ g(n), \text{ for } n \geq n_0$$

- Big-Theta
  - Two function grow at the same rate
  - $f(n)$ is $\theta(g(n))$, if $f(n)$ is $O(g(n))$ and $f(n)$ is $\Omega(g(n))$.
  - If there are real constants c'>0 and c''>0, and an integer constant n_0 >= 1

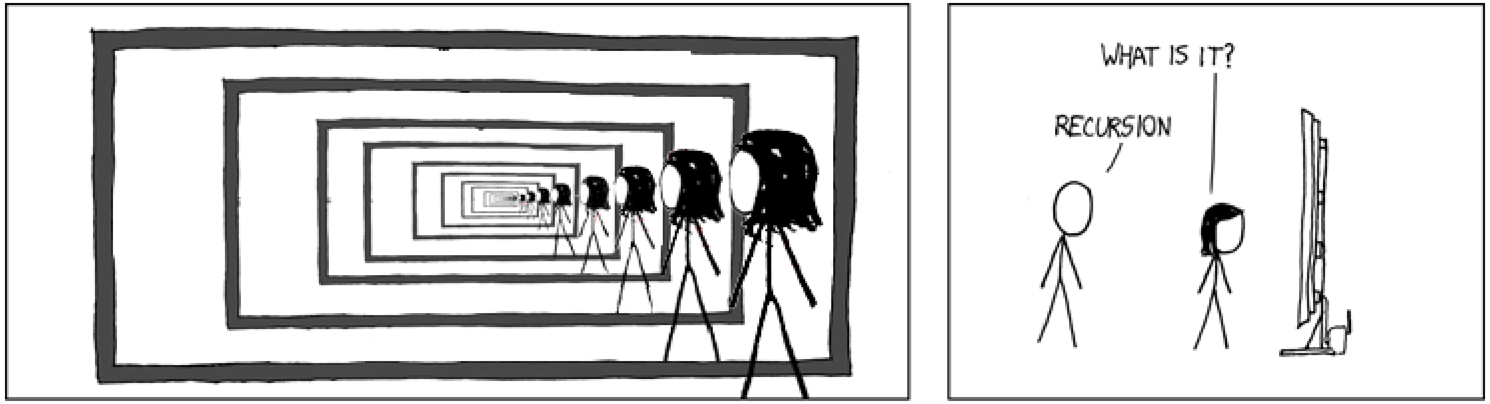
$$c'\, g(n) \leq f(n) \leq c''\ g(n), \text{ for } n \geq n_0$$

PRACTICE

- Order the following functions by asymptotic growth rate
  - 4n logn + 2n
  - 2^{10}
  - $2^{\log n}$
  - 3n + 100 log n
  - 4n
  - 2^n
  - $n^2 + 10n$
  - n^3
  - n log n

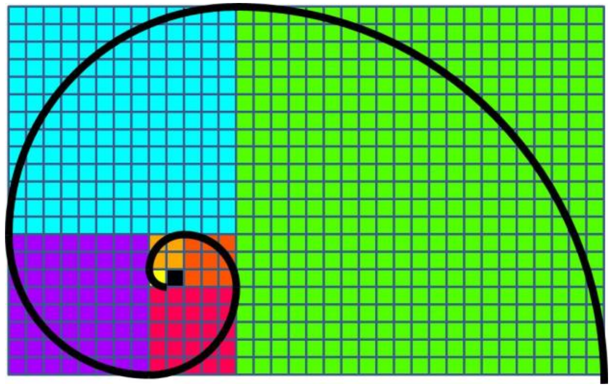
RECURSION
递归

- Theory on recursion
- Examples
- Analysis of recursive algorithm
- Bad recursions
- Types of recursions



RECURSION
递归

- Repetition within a computer program
  - Loops: while-loop, for-loop
  - Recursion
- Recursion: a function makes one or more calls to itself during execution
  - Earliest means for repetitive tasks
  - Some programming language do not support loops
    - Scheme, Smalltak
  - Not only in computer science
    - Fractal patterns
    - Russian Matryoshka dolls



THE FACTORIAL FUNCTION

- The factorial function
- The factorial of a positive integer n, often denoted as n!, is defined as the product of the inters from 1 to n
  - 5! = 5*4*3*2*1 = 120

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot (n-1) \cdot (n-2) \cdots 3 \cdot 2 \cdot 1 & \text{if } n \geq 1. \end{cases}$$

- This definition is typical of many recursive definitions
  - Base case: 1
  - General (recursive) case: n*(n-1)
- A recursive implementation of the factorial function

```
def factorial(n):
  if n == 0:
    return 1
  else:
    return n * factorial(n−1)
```

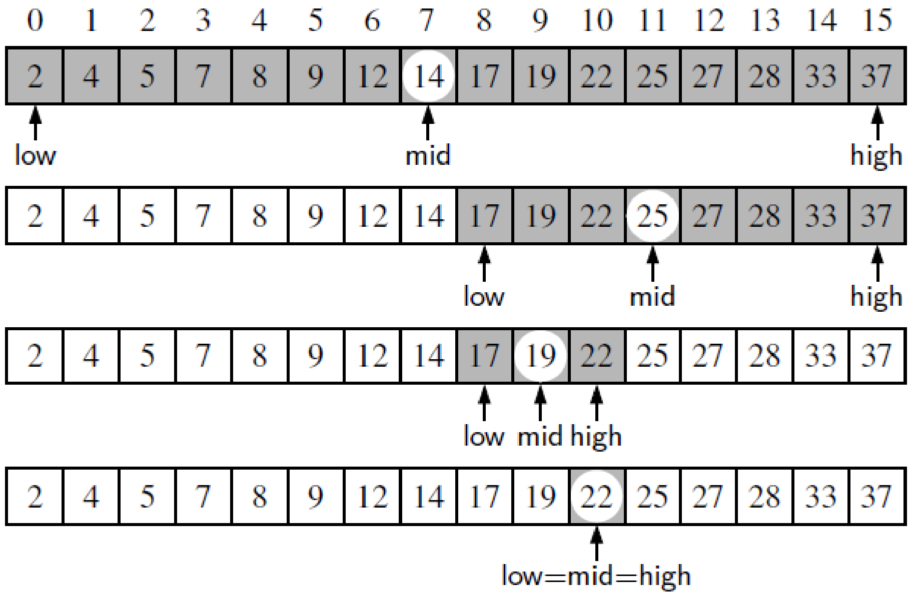
BINARY SEARCH

- Problem: locate a target value within a sequence
- Unsorted sequence: O(n)
  - Sequential search
- Sorted and indexed sequence: O(log n)
- Rationale
  - Values stored at indices 0, …, j-1
  - j+1 value is greater than or equal to j value
  - We can always start in the middle: mid = floor((low + high/2))
- Three cases:
  - If the target equals data[mid], we are done
  - If target < data[mid], look left
  - If target > data[mid], look right

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
2	4	5	7	8	9	12	14	17	19	22	25	27	28	33	37

BINARY SEARCH

- Python implementation
  - binary_search(data, target, low, high)
- O(log n) time
  - Takes (at most) only 30 steps to locate a number from 1,000,000,000 elements
- Find 22

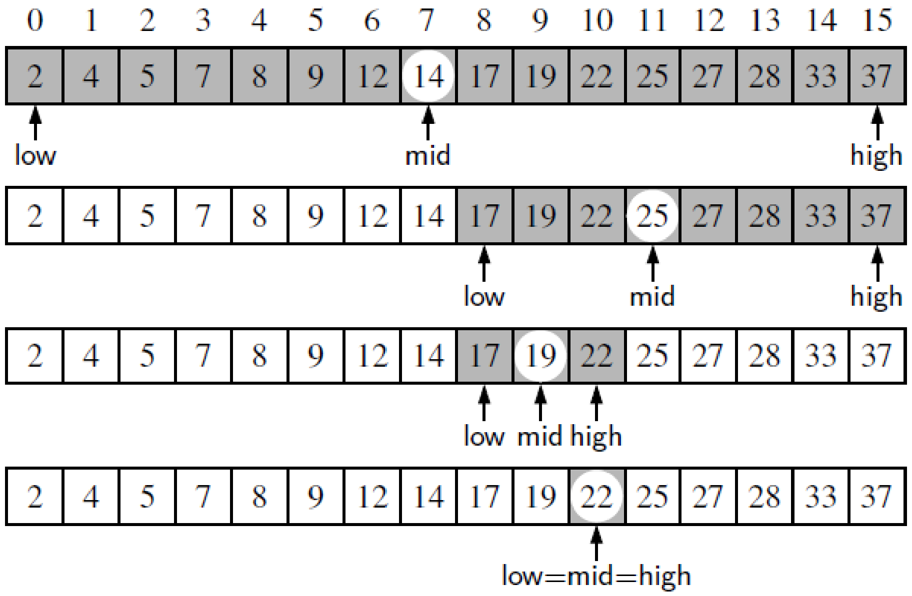


ANALYSING RECURSIVE ALGORITHMS

- Asymptotic analysis: estimating the number of primitive operations
  - Big-O, Big-Omega, Big-Theta
- Recursive algorithm
  - For each invocation (**activation**) of the function, account for:
    - Number of operations performed within the body of
  - Overall number of operations executed
  - Over all **activations**

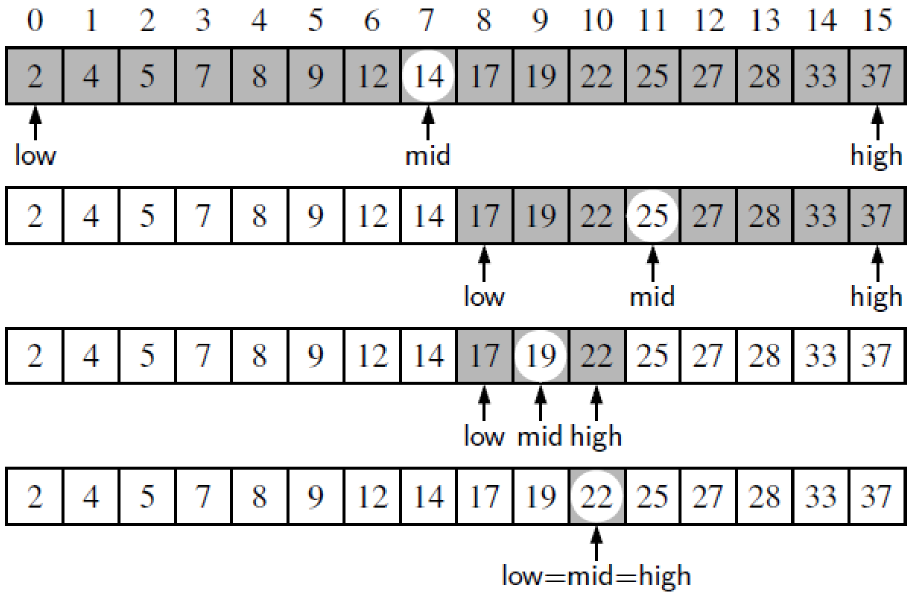
BINARY SEARCH

- Binary search
  - Constant # of primitive operations are executed at each recursive call
  - Running time: proportional to the number of recursive calls performed
- At most floor(log n) + 1 recursive calls are made during a binary search of a sequence with n elements



BINARY SEARCH

- Proposition: the binary search algorithms runs in O(log n) for a sorted sequence with n elements
- Proof:
- Each call: # of candidates to be searched:
  - High – low + 1
- # of remaining candidates reduced by at least one half with each recursive call:
- (mid-1)-low+1 = floor((low + high)/2) – low <= (high-low +1)/2
- Or
- high-(mid+1)+1 = high-floor((low+high)/2) <= (high-low+1)/2

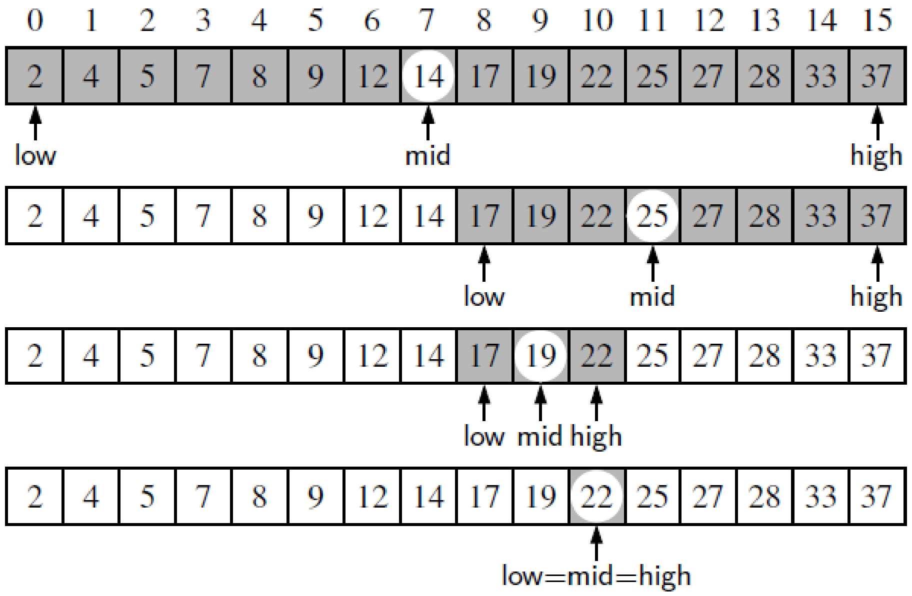


BINARY SEARCH

- Begin: # of candidates = n
- 1^{st} call: # of candidates = n/2
- 2^{nd} call: # of candidates = n/4
- j^{th} call: # of candidates $n/2^j$
- Worst case?
  - Stops when there are no more candidates
- Maximum number of recursive calls = smallest integer r such that

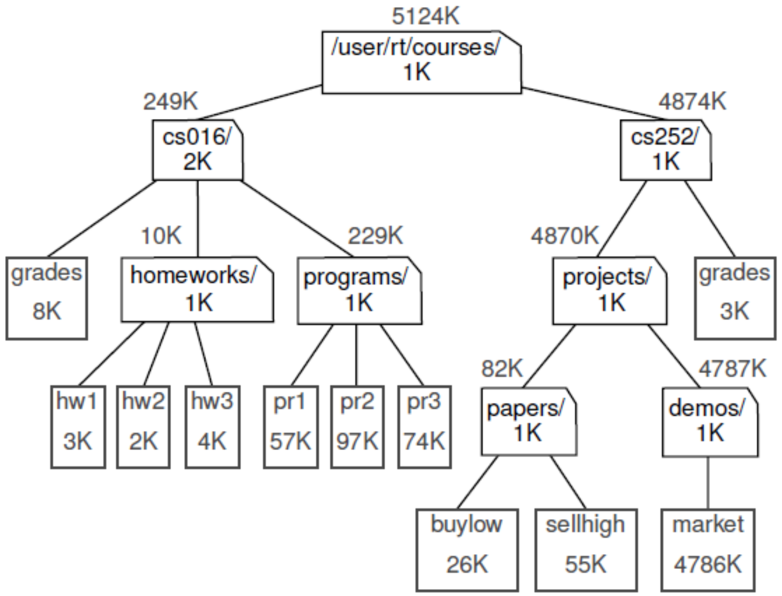
$$\frac{n}{2^r} < 1.$$

- r > log n
- r = floor(log n) + 1



FILE SYSTEM

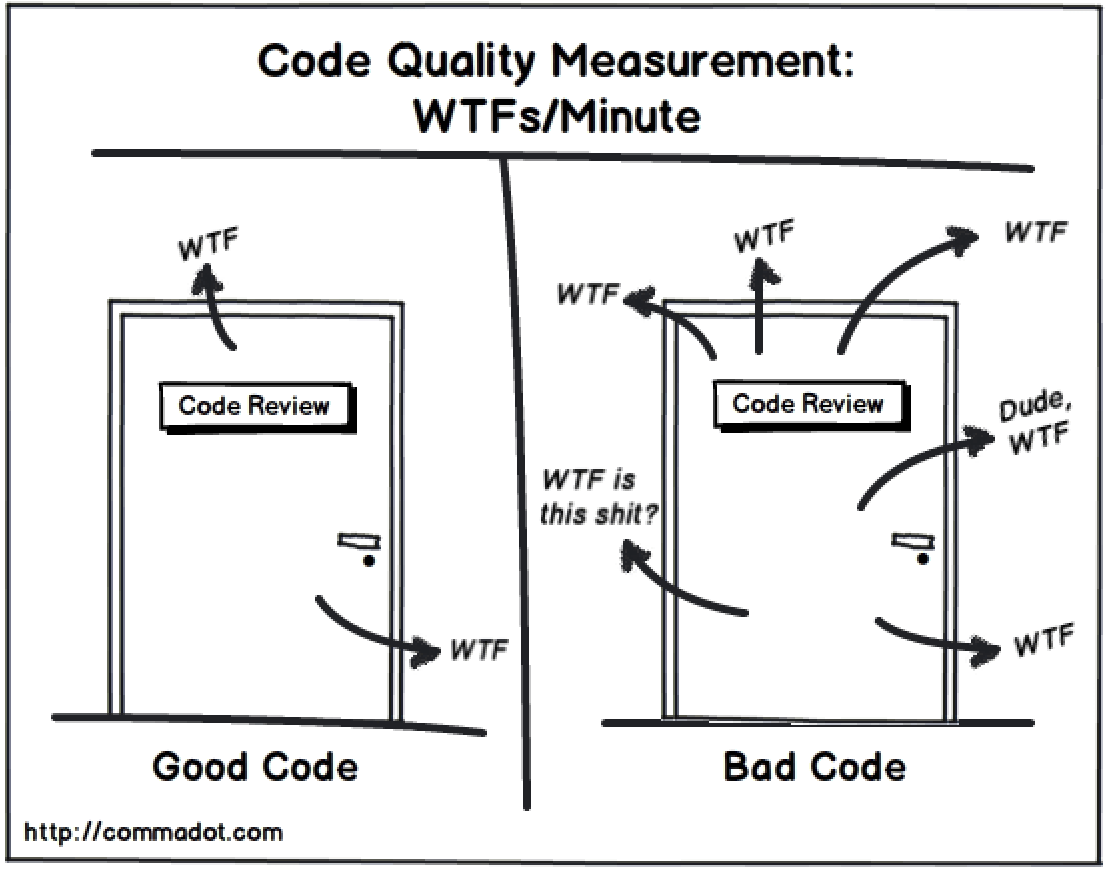
- By induction: there is exactly one recursive invocation of disk_usage for each entry at nesting level k
  - Base: k = 0, # calls = 1 (the initial one)
  - Inductive: for k > 0, # calls = 1 for each entry e
- Complexity
  - O(1) for function call?
    - os.path.getsize(): for loop iterates over all entries contained within the directory
    - Worst case n-1 entries
  - O(n) recursive calls, each runs in O(n) time
  - O(n^2)
- Is this a accurate estimation?



Algorithm DiskUsage(path):
Input: A string designating a path to a file-system entry
Output: The cumulative disk space used by that entry and any nested entries
total = size(path) {immediate disk space used by the entry}
if path represents a directory **then**
for each child entry stored within directory path **do**
total = total + DiskUsage(child) {recursive call}
return *total*

RECURSION

- What can go wrong?



FIBONACCI FUNCTION

- Previously on Fibonacci function
  - F(0) = 0
  - F(1) = 1
  - F(n) = F(n-1) + F(n-2)

```python
def bad_fibonacci(n):
  """Return the nth Fibonacci number."""
  if n <= 1:
    return n
  else:
    return bad_fibonacci(n-2) + bad_fibonacci(n-1)
```

FIBONACCI FUNCTION

- Previously on Fibonacci function
  - F(0) = 0
  - F(1) = 1
  - F(n) = F(n-1) + F(n-2)
- Efficiency?
  - C_n : # of calls to bad_fibonacci() performed
  - # of calls doubles for each two consecutive indeices
  - $O(2^n)$

```
def bad_fibonacci(n):
  """Return the nth Fibonacci number."""
  if n <= 1:
    return n
  else:
    return bad_fibonacci(n−2) + bad_fibonacci(n−1)
```

$$
\begin{aligned}
c_0 &= 1 \\
c_1 &= 1 \\
c_2 &= 1 + c_0 + c_1 = 1 + 1 + 1 = 3 \\
c_3 &= 1 + c_1 + c_2 = 1 + 1 + 3 = 5 \\
c_4 &= 1 + c_2 + c_3 = 1 + 3 + 5 = 9 \\
c_5 &= 1 + c_3 + c_4 = 1 + 5 + 9 = 15 \\
c_6 &= 1 + c_4 + c_5 = 1 + 9 + 15 = 25 \\
c_7 &= 1 + c_5 + c_6 = 1 + 15 + 25 = 41 \\
c_8 &= 1 + c_6 + c_7 = 1 + 25 + 41 = 67
\end{aligned}
$$

FIBONACCI FUNCTION

- What is the problem with the current Fibonacci function?
- In F(n-1), we compute F(n-3) and F(n-2)
- For F(n), we need to compute F(n-1) and F(n-2) again
- In the textbook: returning a tuple

```
def good_fibonacci(n):
  """Return pair of Fibonacci numbers, F(n) and F(n-1)."""
  if n <= 1:
    return (n,0)
  else:
    (a, b) = good_fibonacci(n-1)
    return (a+b, a)
```

FIBONACCI FUNCTION

- What is the problem with the current Fibonacci function?
- In F(n-1), we compute F(n-3) and F(n-2)
- For F(n), we need to compute F(n-1) and F(n-2) again
- But returning a tuple may not be what typical computer programs do
- Complexity?
- Dynamic Programming

```
memo = { }
fib(n):
        if n in memo: return memo[n]
        else: if n ≤ 2 : f = 1
              else: f = fib(n − 1) + fib(n − 2)
              memo[n] = f
              return f
```

FIBONACCI FUNCTION

- What is the problem with the current Fibonacci function?
- In F(n-1), we compute F(n-3) and F(n-2)
- For F(n), we need to compute F(n-1) and F(n-2) again
- In the textbook: returning a tuple

```
def good_fibonacci(n):
  """Return pair of Fibonacci numbers, F(n) and F(n-1)."""
  if n <= 1:
    return (n,0)
  else:
    (a, b) = good_fibonacci(n−1)
    return (a+b, a)
```

OTHER TOPICS IN RECURSION

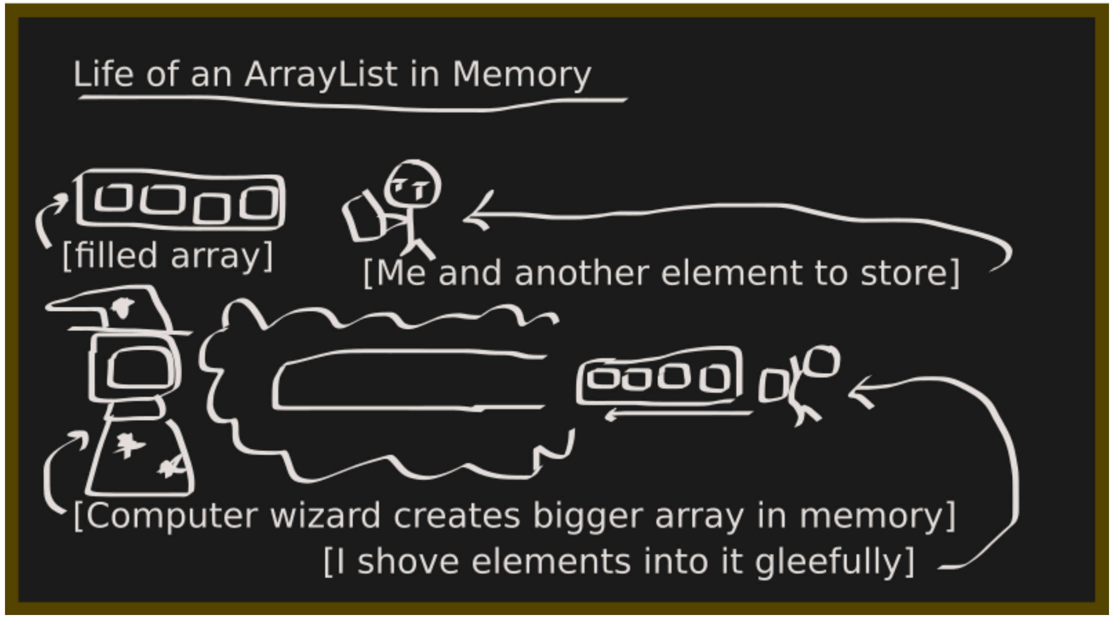
- Maximum Recursive Depth （最大递归深度）？
- Types of recursion
  - Linear recursion – 线性递归
  - Binary recursion – 二元递归
  - Multiple recursion – 多元递归

设计递归算法

- 递归算法包括：
  - Base case(s): at least one
    - Test input for base cases
    - Base case(s) should not use recursion
  - Recursive step(s):
    - Linear
    - Binary
    - Multiple
    - Progress towards the base case(s)
- Design of the function to facilitate recursion
  - binary_search(data, target)
  - binary_search(data, target, low, high)

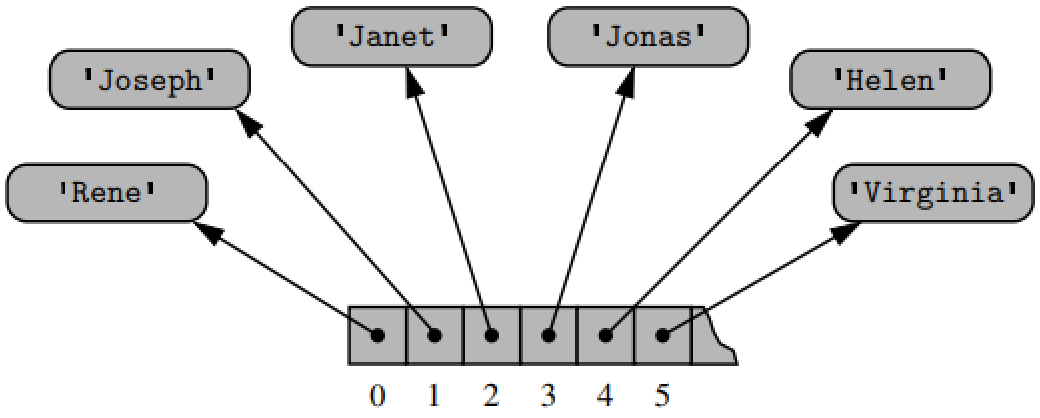
ARRAY BASED SEQUENCES
基于数组的序列

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



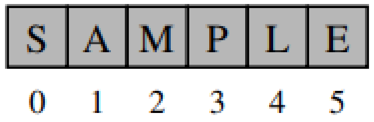
REFERENTIAL ARRAYS
引用数组

- 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?



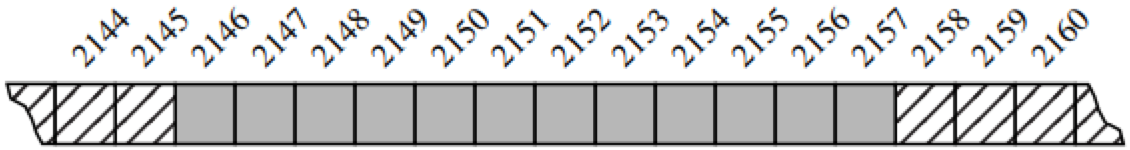
COMPACT ARRAYS

- 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)



DYNAMIC ARRAYS AND AMORTISATION

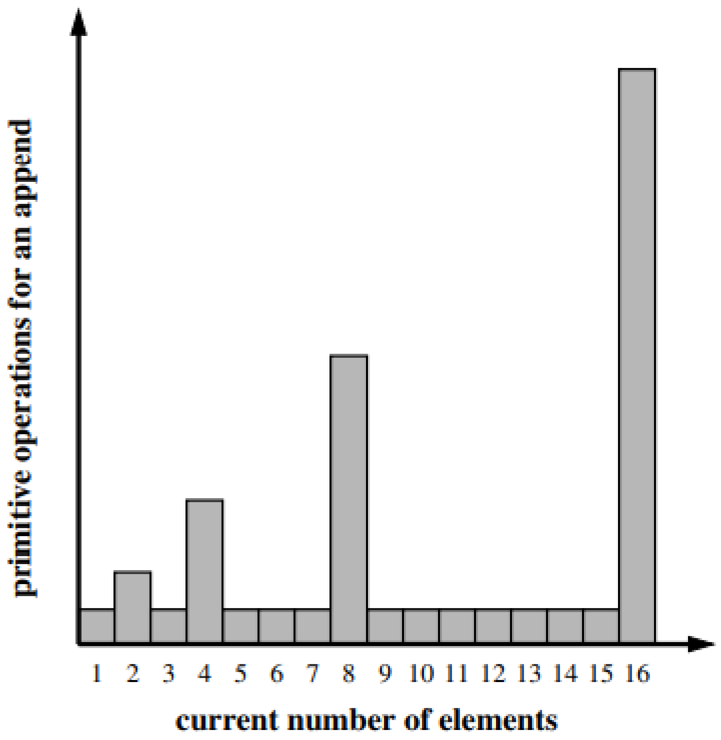
- When creating (initialising) an array, the precise size of it must be declared for the system to allocate a consecutive piece of memory
  - E.g. array of 12 bytes



- 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**

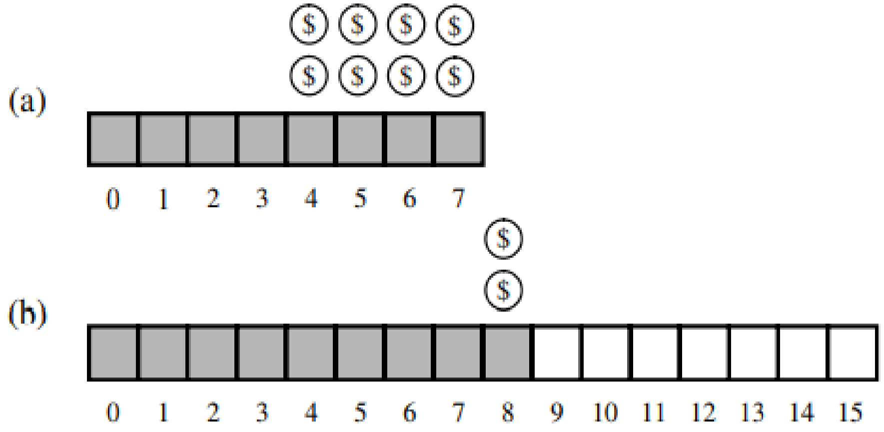
AMORTISATION (摊销)

- 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)



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?

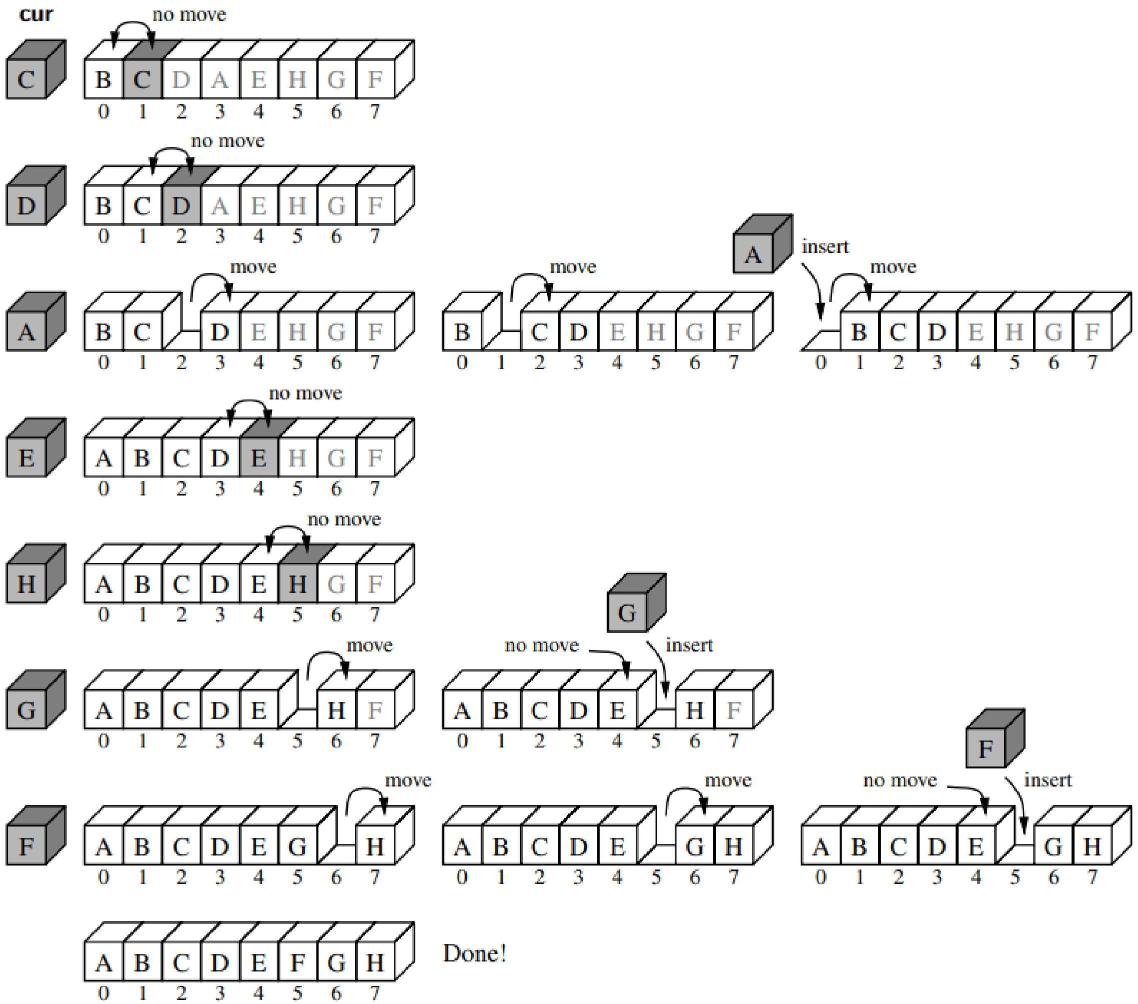
SORTING A SEQUENCE

- Insertion sort
- Like what you do when you play playing cards
  - Start with the first element (no comparison needed)
  - Move to the 2nd element, compare it with the 1st one
    - If smaller than the 1st element, swap with 1st
  - Move to the 3rd element, compare with the 2nd one
    - If smaller, then swap, compare with 1st, if smaller, then swap
  - Move to the nth element, compare with n-1 one
    - If smaller, swap with n-1, then repeat the compare-swap process
- Complexity?

INSERTION SORT

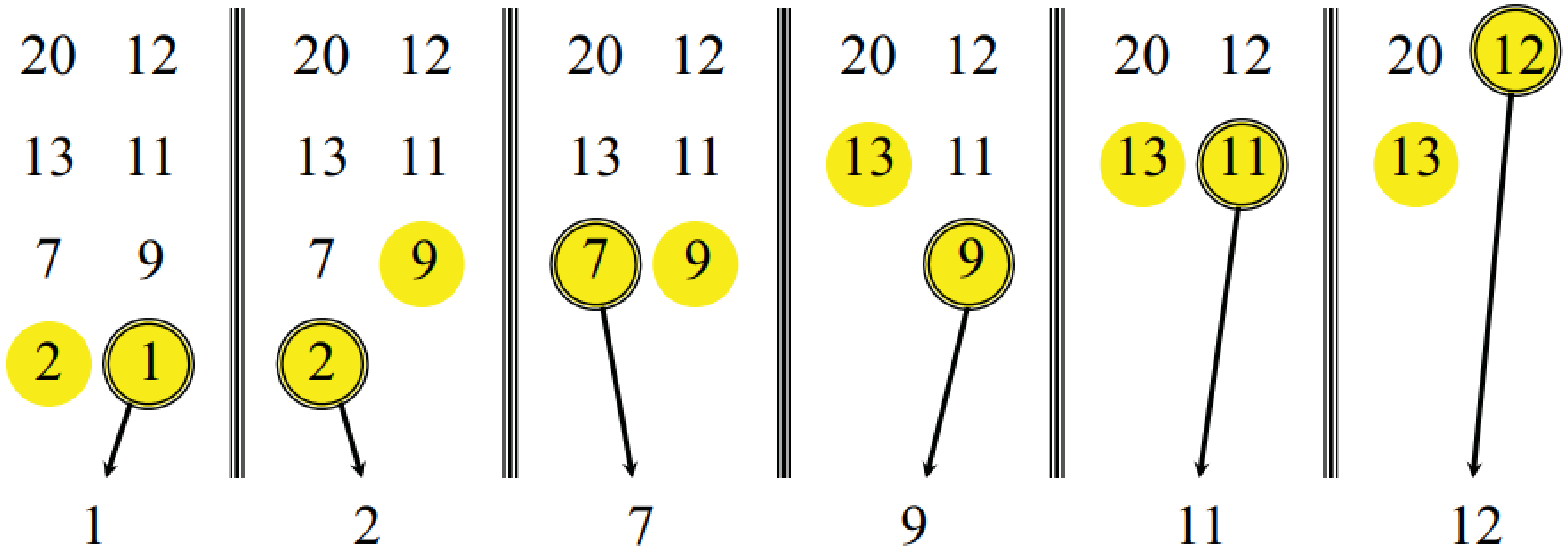
```
def insertion_sort(A):
  """Sort list of comparable elements i
  for k in range(1, len(A)):
    cur = A[k]
    j = k
    while j > 0 and A[j−1] > cur:
      A[j] = A[j−1]
      j −= 1
    A[j] = cur
```

- $O(n^2)$



QUESTION

- How do you merge 2 sorted list into 1 sorted list?

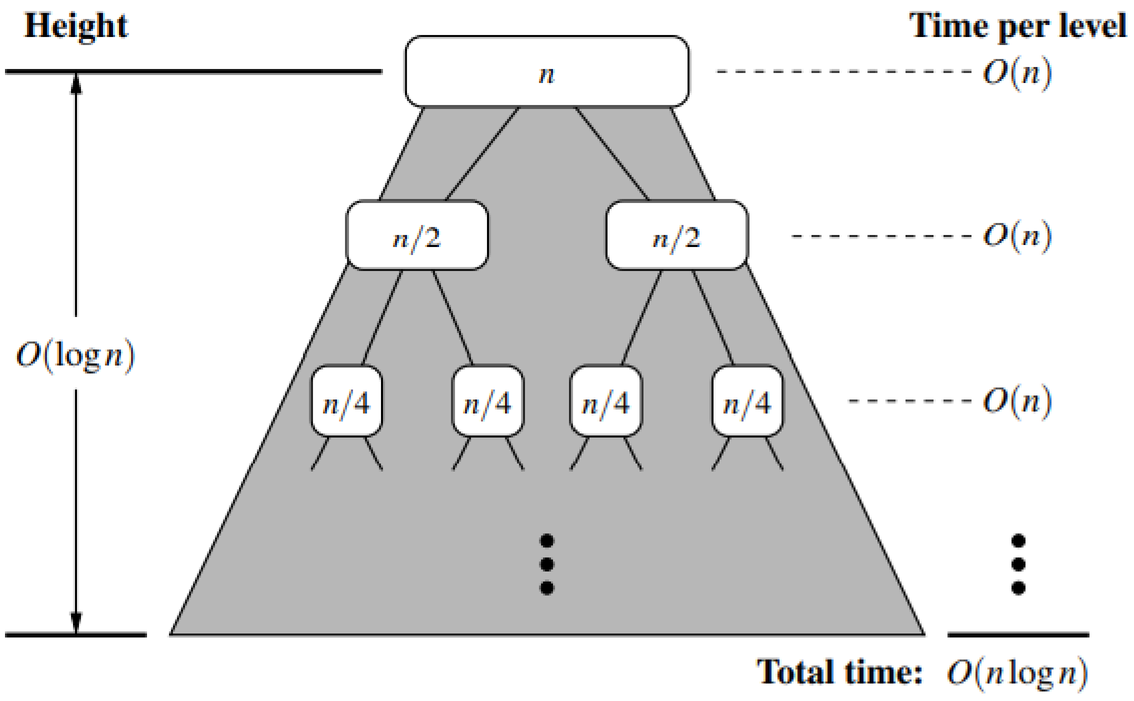


- Time to merge a total of n elements?

MERGE SORT

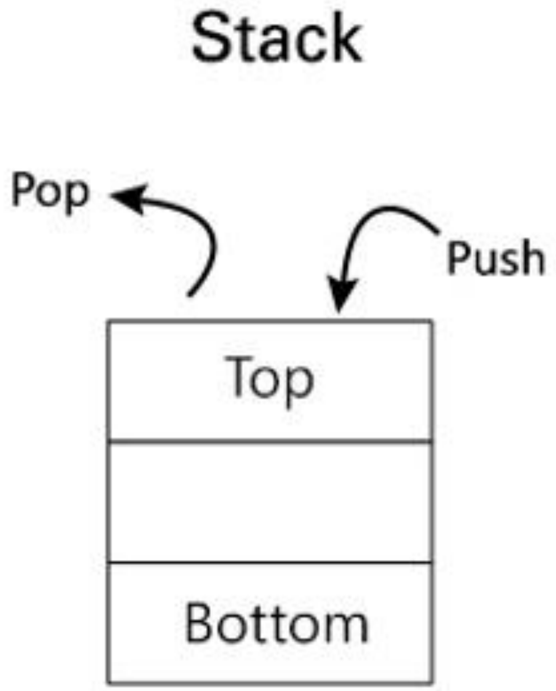
- merge_sort(A[1..n])

1. If n = 1, done
2. recursively sort A[1..n//2], A[n//2+1..n]
3. Merge the two sorted list



STACK （栈）

- Definition（定义）: collection of objects that are inserted and removed according to the LIFO principle
  - Last-in, First-out （后进先出）
- Insertion: can happy at any time
- Access/remove: most recently inserted object
- A stack of plates/cups
- Fundamental operations involved
  - push() – 进栈
  - pop() – 退栈
- Examples
  - Browsing history of your browser
  - Text editor: “undo”



STACK （栈）

- Abstract Data Type (抽象数据类 - ADT)
- S.push(e): adds e to the top of stack S
- S.pop(): removes and return the top element from stack S
  - Error when stack is empty
- S.top(): returns a reference to the top element of stack S (but not to remove it)
- S.is_empty(): returns True if S does not have any elements
- len(s): returns the number of elements in S

IMPLEMENTING A STACK WITH A PYTHON LIST

- How?

EFFICIENCY ANALYSIS

- S.push(e) : O(1) Amortised
- S.pop(): O(1) Amortised
- S.top(): O(1)
- S.is_empty(): O(1)
- len(S): O(1)

REVERSING DATA USING A STACK

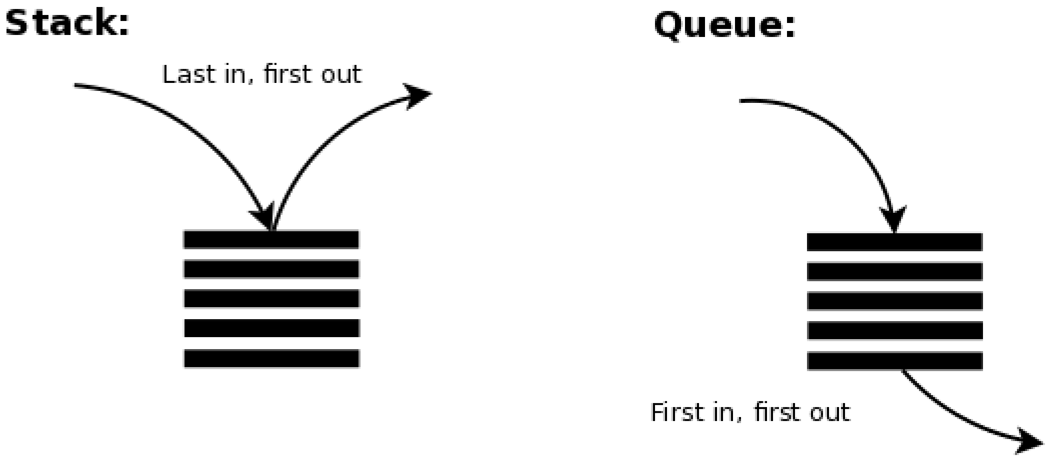
- LIFO is useful when we want to reverse a data sequence
- [1, 2, 3] can be easily reversed to [3, 2, 1]

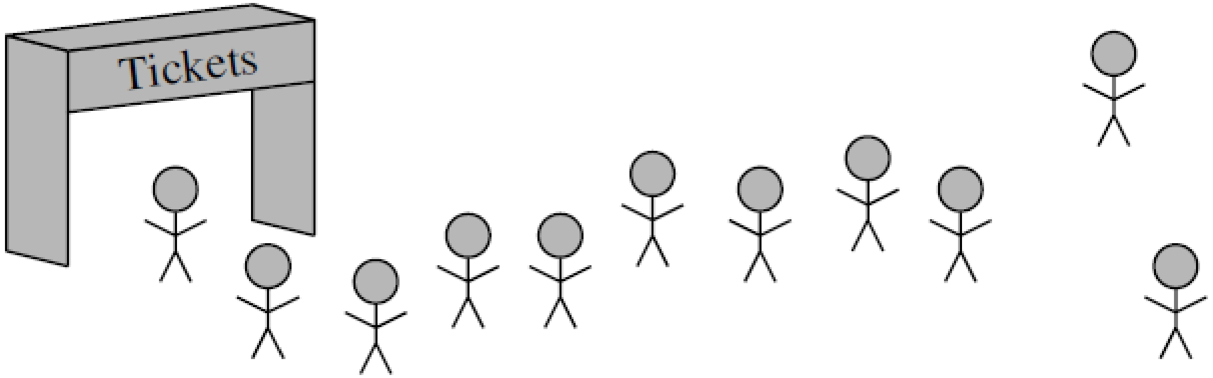
```
def reverse_file(filename):
  """Overwrite given file with its contents line-by-line reversed."""
  S = ArrayStack()
  original = open(filename)
  for line in original:
    S.push(line.rstrip('\n'))          # we will re-insert newlines when writing
  original.close()

  # now we overwrite with contents in LIFO order
  output = open(filename, 'w')      # reopening file overwrites original
  while not S.is_empty():
    output.write(S.pop() + '\n')  # re-insert newline characters
  output.close()
```

QUEUES （队列）

- FIFO principle
  - First-In, First-Out (先进先出)
- Elements can be inserted at any time
- Only the elements that has been in the queue the longest can be removed next
- Applications
  - Reservation centres
  - Process management
  - Web server





QUEUES

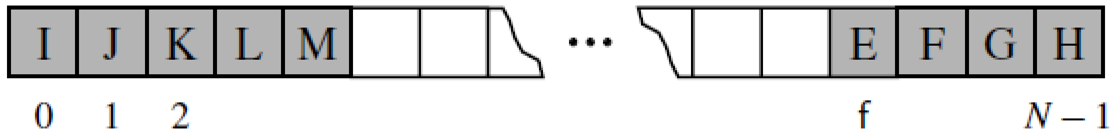
- Abstract Data Type (ADT)
- Q.enqueuer(e): add element e to the back of queue Q （入队）
- Q.dequeue(): remove and return the first element from queue Q （出队）
- Q.first(): returns a reference to the element at the front of queue Q
  - Error if Q is empty
- Q.is_empty(): returns True if queue Q does not contain any elements
- len(Q): returns the number of elements in Q

QUEUES

- Array-Based Implementation
- How?

QUEUES

- Using a circular array?
  - We allow the front of the queue to drift rightward
  - We allow the contents of the queue to “wrap around” the end of an array
  - Assumption: underlying array has fixed length N
    - N > actual number of elements in the queue



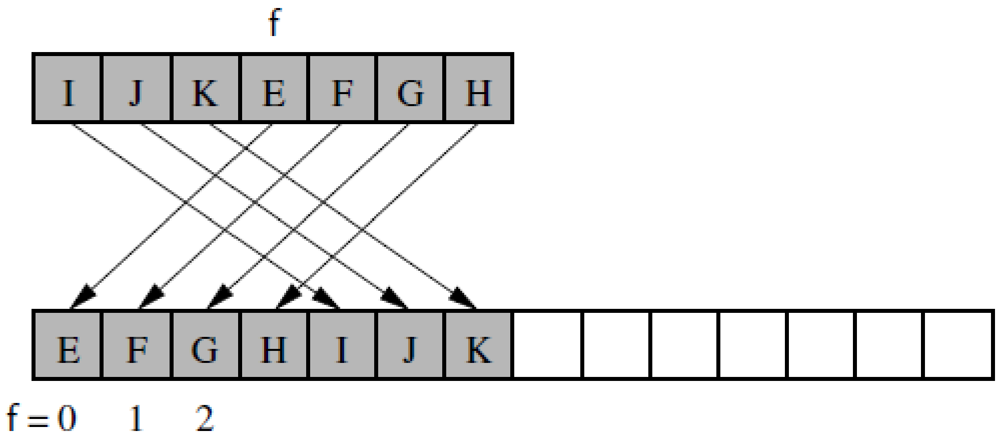
  - Variable f to keep track of the “front” of the queue
  - dequeue() causes f to shift right
    - f = (f+1)%N
  - Example: N = 10, f = 7
    - (7+1)%10 = 8, (8+1)%10 = 9, (9+1)%10 = 0

QUEUES

- Adding and removing elements :enqueuer() and dequeue()
- enqueue(): determine the proper index at which to place the new element
  - avail = (self._front + self._size) % len(self._data)
  - Capacity 10, Current size 3, First element 5
    - Three elements stored at 5, 6 and 7
      - New elements should be placed at index (front+size) = 8
    - In a case with wrap-around
      - We do (front+size)%capacity
- Dequeue()
  - If we return self._data[self._front], then the next operation
    - self._data[self._front] = None
    - Why?

QUEUES

- Resizing (变容) the Queue
- When the size of the queue equals the size of the list
  - Standard “doubling” of the array
  1. New array with greater capacity
  2. Copy over the elements
     - re-align the front of the queue with index 0



QUEUES

- Shrinking （缩小） the underlying array
- Space efficiency: $\Theta(n)$, n # of elements in the queue
- Do you remember the strategy of shrinking a dynamic array?
  - Shrink by half when the usage falls below one fourth

```
if 0 < self._size < len(self._data) // 4:
  self._resize(len(self._data) // 2)
```

QUEUES

- Efficiency of Array-Based Queue
- Q.enqueuer(e): O(1) amortised
- Q.dequeuer(): O(1) amortised
- Q.first(): O(1)
- Q.is_empty(): O(1)
- len(Q): O(1)

QUIZ FOR THIS WEEK

- You have 25 horses, and you want to pick the fastest 3 horses out of the 25.
- You do this by getting them to race
- In each race, only 5 horses can run at the same time
  - Because there are only 5 tracks
- What is the minimum number of races required to find the 3 fastest horses without using a stopwatch?

QUIZ FOR THIS WEEK

1. Divide the 25 into groups of 5, race the horses in each group
2. Take the winner from each group and race those 5 horses. The winner of this race is the fastest horse overall
3. Label 5 groups from step 1 as a, b, c, d, e to correspond to horses finishing 1st, 2nd, 3rd, 4th and 5th in step 2.
4. Do one more race with horses a2, a3, b1, b2, c1. The top 2 horses in this race are the 2nd and 3rd fastest horses overall.

THANKS

See you in the next session!