Introduction to Algorithms & Data Structures

Developing Efficient Code

George Heineman

October 9 2019

O'REILLY

GitHub

- https://github.com/heineman/IntroductionAlgorithmsDataStructures
  - Visit https://github.com/heineman and see it as one of the pinned repositories

Python IDLE

- Development environment used for course
  - You can use your own and just grab source
- Relevant Toolset
  - Python 3.7

Presentation Outline

- Log(n) Behavior
- Basic Data Structures
- Sorting Algorithms
- Graph Algorithms
- Data Structure Summary
- SkipList

Course Objectives

- Learn about existing Python libraries
  - Avoid reinventing wheel
  - Suggestion on when to use data structures

Question: I ask a number of questions throughout course, which appear at the bottom of a slide in a purple box.

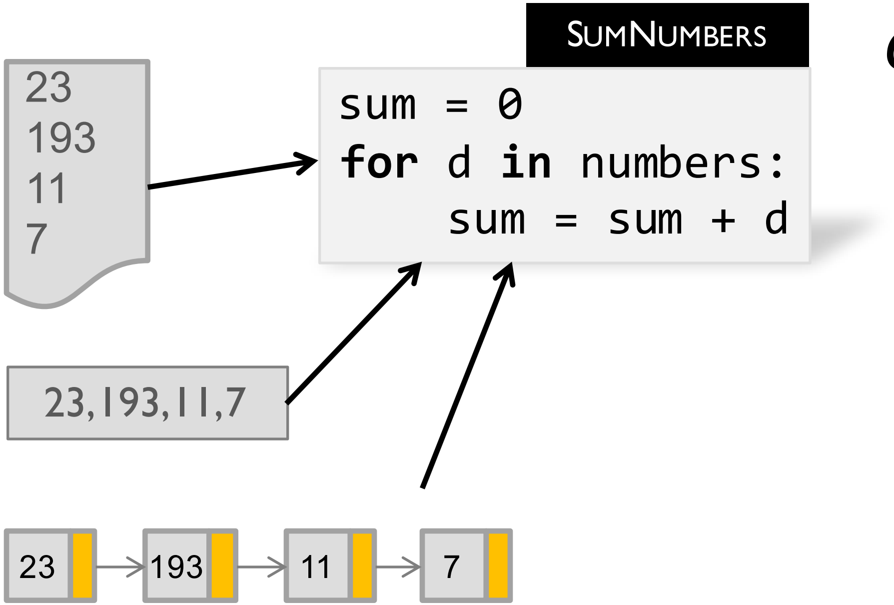
Algorithm Formalities

- Definition of an *algorithm*
  - An algorithm describes the **computational steps** to compute an exact answer for a single **problem instance** on a **sequential deterministic computer**
- How to compare two different algorithms that solve the same problem?

Asymptotic Analysis

- Characterize **time complexity**
  - Time for algorithm to complete
  - Calculate time as function $t(n)$ relating the number of steps to problem instance size, n
- Characterize **space complexity**
  - Amount of computer storage required
  - Determine required space $s(n)$ in similar fashion

Small Algorithm Example
Sum n Integers



SumNumbers

```
sum = 0
for d in numbers:
    sum = sum + d
```

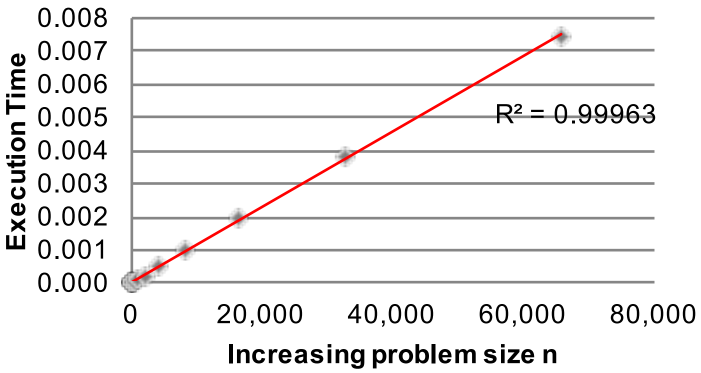
This algorithm uses ***n*** *addition operations regardless of how data is stored*

Time complexity: *t*(*n*) is directly proportional to *n*

Space complexity: *s*(*n*) is constant (only sum)

Asymptotic Growth

- Determine **order of growth** in worst case: O(*f*(*n*))
  - Evaluate *t*(*n*) as problem size *n* doubles
- Execution times of SUM show correlation between *n* and *t*(*n*)
  - SUM exhibits linear behavior
  - SUM is O(*n*)
  - Additive constants don't matter



Check If List Contains A Target Integer

- With unordered `aList`
  - Just use Python **in** operator
- If `aList` is sorted
  - Is `SortedContains` faster?
  - It stops at first instance greater than desired target
- To evaluate: use last value in `aList` – worst case

CONTAINS

```
return tgt in aList
```

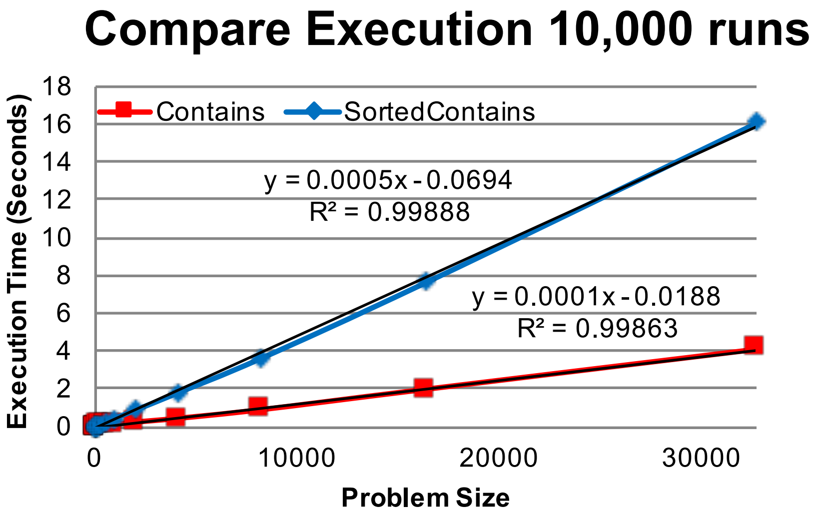
SORTEDCONTAINS

```
for v in aList:
❶  if v > tgt:
     return False
②  if v == tgt:
     return True
return False
```

O'REILLY

Empirical Evaluation

- Python **in** operator about 4x faster
- Both exhibit linear growth or O(*n*)
  - As problem size doubles, programs work twice as hard
  - Multiplicative constant (i.e., the slope of each line) does not change classification



O'REILLY

Observations on BinaryArraySearch

- A phone book with *n* entries is sorted by last name (and first name within last name)
  - Easy to locate a phone # for a given person
  - Hard to locate a person for a given phone #

Question: Is it twice as hard to search through a phone book with 400 pages than one with 200 pages?

BinaryArray Search

- With ordered `aList`
  - Cuts the problem size in half with each pass

lo			*mid*			*hi*
1	4	8	9	11	15	17

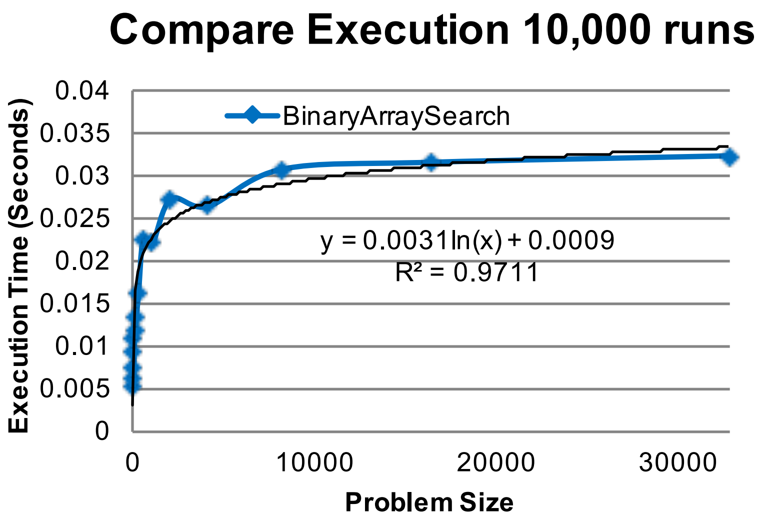
BinaryArraySearch

```
①  lo = 0
    hi = len(aList) - 1
②  while lo <= hi:
      mid = (lo + hi) // 2
      if tgt < aList[mid]:
③        hi = mid - 1
      elif tgt > aList[mid]:
④        lo = mid + 1
      else:
⑤        return True
⑥  return False
```

  - What performance should we expect?

Empirical Evaluation

- Different performance
  - As problem size doubles, time increased is constant
  - Only one more pass through the loop
- Time classification
  - O(log *n*)



Algorithm Classification Summary

- Space Complexity is storage above and beyond the input
- Time Complexity gives "ball park" classification of worst-case performance

	Space Complexity	Time Complexity
`SumNumbers`	O(1)	O(*n*)
`Contains`	--	O(*n*)
`SortedContains`	O(1)	O(*n*)
`BinaryArraySearch`	O(1)	O(log *n*)

O'REILLY

Amortized Analysis

- Performing an operation may have different profiles
  - Sometimes an operation requires constant time – O(1)
  - 1 out of *n* times, the same operation requires more – O(*n*)
- Amortized Average Case is O(1)
  - When you make *n* operations and *n-1* require constant time while just 1 requires O(*n*)

O'REILLY

Amortized Analysis

– Consider $\underbrace{c + c + \ldots + c}_{\substack{n-1 \text{ times} \\ \text{operation requires} \\ \text{constant time } c}} + \underbrace{cn}_{\substack{\text{One time,} \\ \text{operation requires} \\ \text{time } c*n}} = c(n-1) + cn = 2cn - c$

$$\text{Average} = \frac{2cn}{n} - \frac{c}{n} \approx 2c$$

And $2c$ is O(1)

O'REILLY

Problem Instances

- Best Case – Require least work
- Worst Case – Require most work
- Average Case – Hard to evaluate
- Example: **Use BINARYARRAYSEARCH**
  - **Best Case**: Target is midpoint, so found immediately: O(1)
  - **Worst Case**: Target is not contained in array: O(log *n*)
  - **Average Case**: Can prove it is O(log *n*)

Asymptotic Growth Defined By Family

	Logarithmic	Linear		Quadratic	Cubic		Exponential
n	**log(n)**	**n**	**n log(n)**	**n^2**	**n^3**	**n^4**	**2^n**
2	1	2	2	4	8	16	4
4	2	4	8	16	64	256	16
8	3	8	24	64	512	4096	256
16	4	16	64	256	4096	65536	65536
32	5	32	160	1024	32768	1048576	4.29E+09
64	6	64	384	4096	262144	16777216	1.84E+19
128	7	128	896	16384	2097152	2.68E+08	3.4E+38
256	8	256	2048	65536	16777216	4.29E+09	1.16E+77
512	9	512	4608	262144	1.34E+08	6.87E+10	1.3E+154
1024	10	1024	10240	1048576	1.07E+09	1.1E+12	∞
2048	11	2048	22528	4194304	8.59E+09	1.76E+13	∞
4096	12	4096	49152	16777216	6.87E+10	2.81E+14	∞
8192	13	8192	106496	67108864	5.5E+11	4.5E+15	∞
16384	14	16384	229376	2.68E+08	4.4E+12	7.21E+16	∞
32768	15	32768	491520	1.07E+09	3.52E+13	1.15E+18	∞

Performance Families

$O(1)$

$O(\log n)$

$O(n)$

$O(n \log(n))$

$O(n^2)$

$O(2^n)$

4096 = 1hr 8min

O'REILLY

Algorithms and Data Structures Solve Problems

- Consider a Word Ladder problem
  - Start with source 4-letter word and change one letter at a time to transform into a target 4-letter word
  - Each intermediate word must be a valid English word
  - Find a shortest path (may be others with same length)

`COLD` → `WARM`

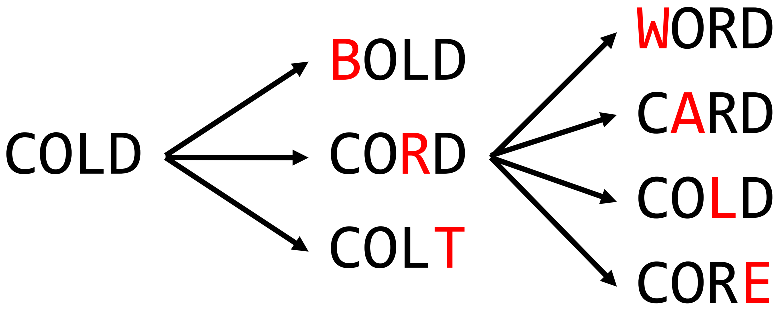
Algorithms and Data Structures Solve Problems

- Consider a Word Ladder problem
  - Start with source 4-letter word and change one letter at a time to transform into a target 4-letter word
  - Each intermediate word must be a valid English word
  - Find a shortest path (may be others with same length)

COLD → CORD → WORD → WARD → WARM

Essential Tasks

- Task 1: Check if four-letter word is English word
  - Is AOLD a word?
- Task 2: Keep track of progress
- Bonus: How to ensure we find a shortest ladder?



Task 1: Check If Word Is English word

- Load up Python `list` of four-letter words
  - n=5,875 in my dictionary
- Use **in** operation
  - O(*n*) in worst case
- But if list is sorted, then…
  - BINARYARRAYSEARCH is O(log *n*)
  - Can we do better? Yes!

LIST EXAMPLE

```
words = []
words.append('AAHS')
words.append('AALS')
words.append('AANI')
…
if 'COLD' in words:
  print ('Yes')
```

O'REILLY

Task 1: Use Python Dictionary

- A set of (key → value) pairs
  - Look up a key in dictionary and return associated value…
  - … or just check existence (i.e., value is not important)
  - Keys are immutable
  - Using hashing, optimal performance can be achieved

DICTIONARY EXAMPLE

```
words = {}
words['COLD'] = 1
if 'COLD' in words:
  print ('Yes')
```

O'REILLY

Task 1: Use Python Dictionary

- Check if word exists using lookup
  - Worst case behavior for dictionary is statistically unlikely
- Collection of four-letter words does not change

	Lookup Value		Insert or Delete	
	Average	Worst	Average	Worst
`Dictionary`	O(1)	O(*n*)	O(1)	O(*n*)
`Self-Balancing Binary Trees`	O(log *n*)	O(log *n*)	O(log *n*)	O(log *n*)
`Sorted List`	O(log *n*)	O(log *n*)	O(*n*)	O(*n*)
`List`	O(*n*)	O(*n*)	O(*n*)	O(*n*)

Task 2: Keep Track Of Progress

- `COLD` has 20 neighbors
  - If none of these are `WARM` what to do?
- Use a **Queue** to keep track of progress
  - Records a sequence of elements
  - Dequeue – Remove from the Left
  - Enqueue – Add to the Right
  - FIFO behavior

`COLD`

↓

`COLA`
`BOLD`
`COLE`
`....`

A D R

Task 2: Keep Track Of Progress

COLD

↓

COLA
BOLD
COLE
....

- COLD has 20 neighbors
  - If none of these are WARM what to do?
- Use a **Queue** to keep track of progress
  - Records a sequence of elements
  - **Dequeue** – Remove from the Left
  - Enqueue – Add to the Right
  - FIFO behavior

D R

O'REILLY

Task 2:
Keep Track Of Progress

COLD

↓

COLA

BOLD

COLE

....

- `COLD` has 20 neighbors
  - If none of these are `WARM` what to do?
- Use a **Queue** to keep track of progress
  - Records a sequence of elements
  - Dequeue – Remove from the Left
  - **Enqueue** – Add to the Right
  - FIFO behavior

D	R	E

Task 2: Keep Track Of Progress

- `COLD` has 20 neighbors
  - If none of these are `WARM` what to do?
- Use a **Queue** to keep track of progress
  - Records a sequence of elements
  - Dequeue – Remove from the Left
  - **Enqueue** – Add to the Right
  - FIFO behavior

COLD

↓

COLA

BOLD

COLE

....

D	R	E	F

Task 2:
Keep Track Of Progress

- `COLD` has 20 neighbors
  - If none of these are `WARM` what to do?
- Use a **Queue** to keep track of progress
  - Records a sequence of elements
  - **Dequeue** – Remove from the Left
  - Enqueue – Add to the Right
  - FIFO behavior

COLD
↓
COLA
BOLD
COLE
....

R E F

Task 2:
Keep Track Of Progress

- Introduce concept of `Stage` in Word Ladder
  - Each stage has word and knows its previous stage
  - Start by enqueuing starting stage (initial word)

Task 2: Keep Track Of Progress

- Introduce concept of `Stage` in Word Ladder
  - Each stage has word and knows its previous stage
  - Start by **enqueuing** starting stage (initial word)

Task 2:
Keep Track Of Progress

- Introduce concept of `Stage` in Word Ladder
  - Each stage has word and knows its previous stage
  - Dequeue a stage and use to enqueue new stages

COLD

↓

COLA

BOLD

COLE

....

Task 2: Keep Track Of Progress

- Introduce concept of `Stage` in Word Ladder
  - Each stage has word and knows its previous stage
  - **Dequeue** a stage and use to enqueue new stages

COLD

↓

COLA

BOLD

COLE

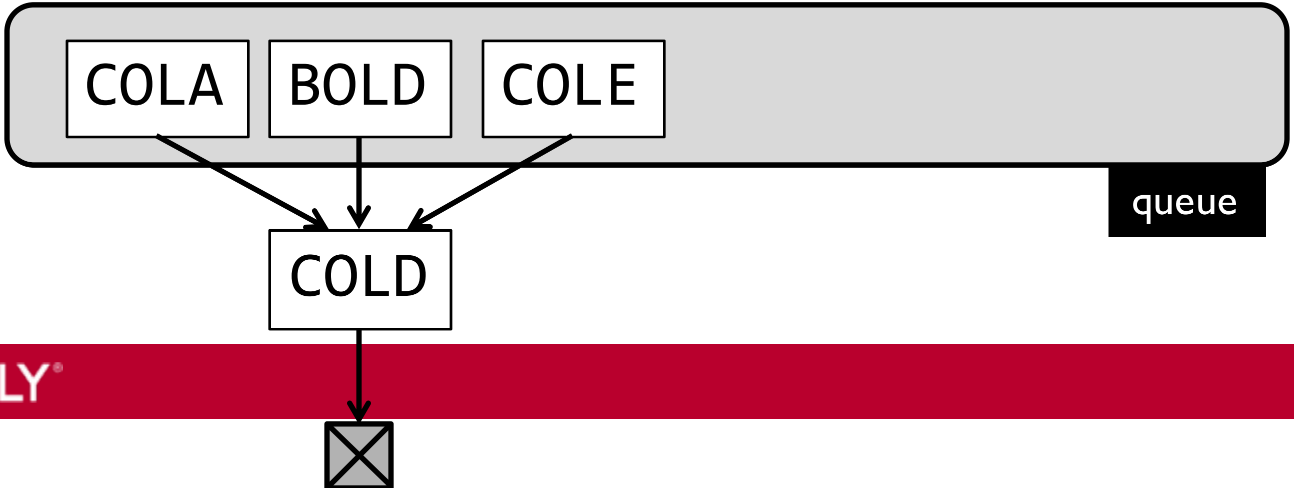
....

O'REILLY

Task 2: Keep Track Of Progress

- Introduce concept of `Stage` in Word Ladder
  - Each stage has word and knows its previous stage
  - Dequeue a stage and use to **enqueue** new stages

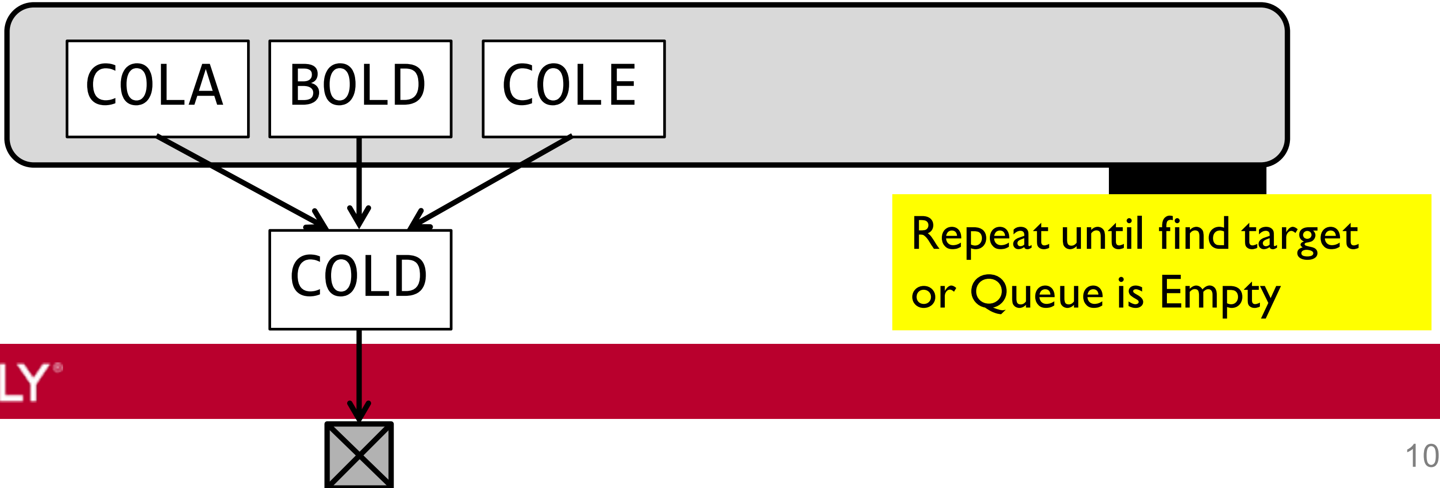
COLD
↓
COLA
BOLD
COLE
....



Task 2: Keep Track Of Progress

- Introduce concept of `Stage` in Word Ladder
  - Each stage has word and knows its previous stage
  - Dequeue a stage and use to enqueue new stages

COLD
↓
COLA
BOLD
COLE
....



Task 2:
Keep Track Of Progress

COLA

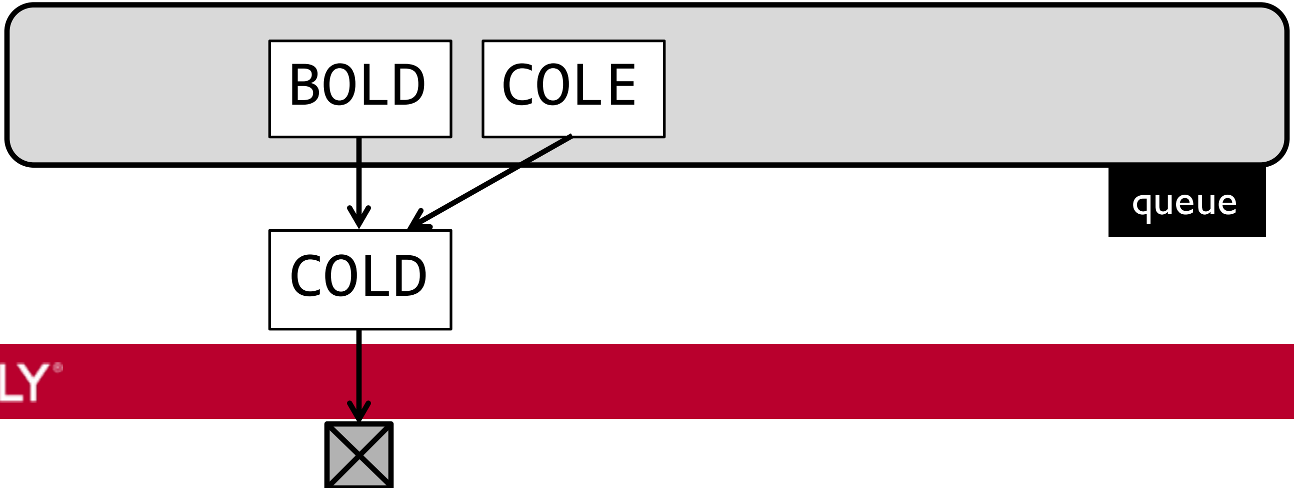
↓

BOLA

COCA

....

- Introduce concept of `Stage` in Word Ladder
  - Each stage has word and knows its previous stage
  - **Dequeue** a stage and use to enqueue new stages



Task 2:
Keep Track Of Progress

COLA

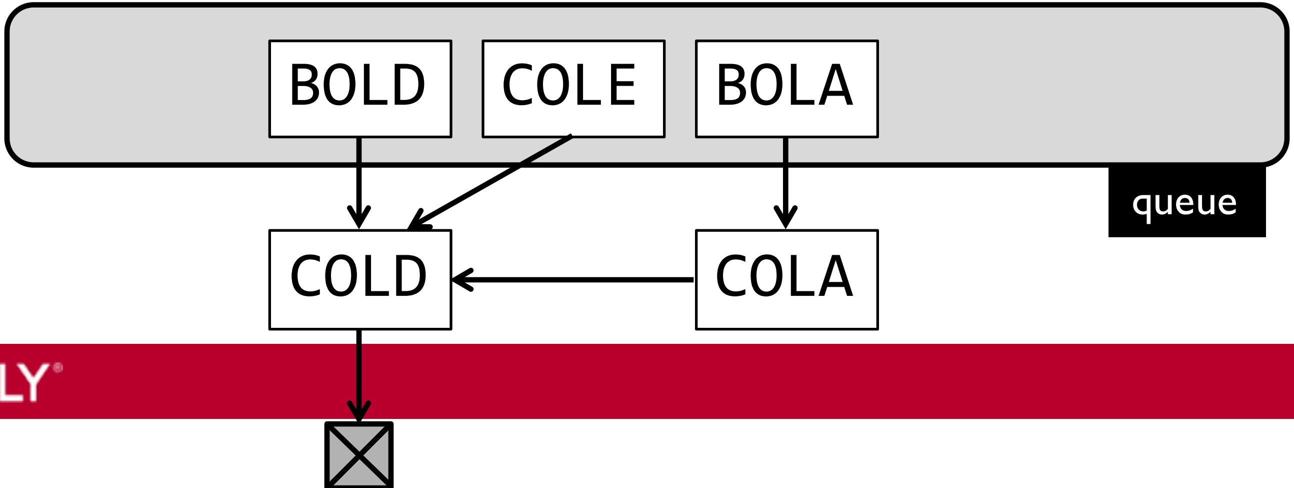
↓

BOLA

COCA

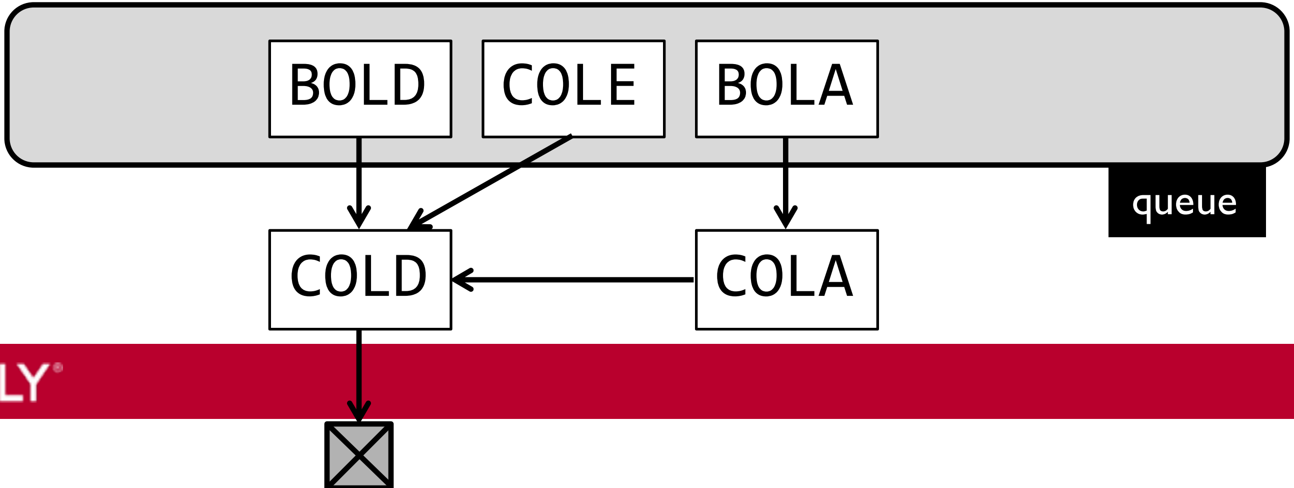
....

- Observe structure of queue
  - BOLD and COLE just one step away from COLD
  - BOLA and subsequent elements at least two steps



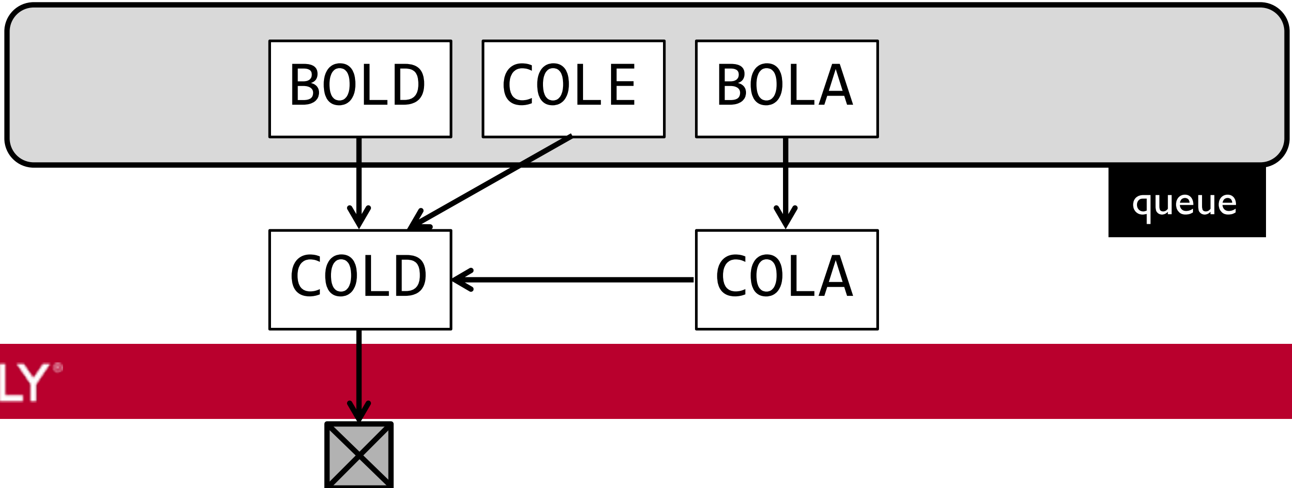
Task 2:
Keep Track Of Progress

- Stages removed from queue in increasing distance from starting stage
  - First all those that are one step away… then two steps…



Task 2:
Keep Track Of Progress

- When enqueing stage, check to see if its word is the target
  - This represents a shortest path from source to target



Summarize Progress

- Task 1: Use dictionary to validate words
- Task 2: Use queue to store progress
  - Process queue by dequeing stage…
  - …and enqueing all neighbor stages

Question: Can the same word appear multiple times in the queue?

Summarize Progress

- Structure of Queue ensures a shortest Word Ladder path will be found first
  - There may be multiple such paths with other words

Question: Can a shortest Word Ladder contain the same word more than once?

Show Code

- Use efficient queue implementation from **deque** package
- Returns target word when found
  - Then follow back pointers to generate Word Ladder

EXPLOREQUEUE

```
❶ active = deque()
  active.append(Stage(start, None))

❷ while active:
❸   st = active.popleft()

    for nxt in neighbors(st.word):
❹     link = Stage(nxt, st)
❺     if nxt == end:
        return link
❻     active.append(link)

❼ return []   # no chain
```

O'REILLY

Performance Comparison

- Timing of `COLD` → `WARM` in seconds
- Different options to check for English words
  - BinaryArraySearch is 24x faster than `list`
  - Dictionary is 5x faster than BinaryArraySearch
  - Your mileage may vary!

```
List        BASearch    Dictionary
190.89      7.81        1.43
```

O'REILLY

Word Ladder Summary

- Algorithm uses a queue data structure to solve the problem
  - Different implementations give wildly different performance results
- Inefficient at heart because it constantly has to compute `neighbors(word)`

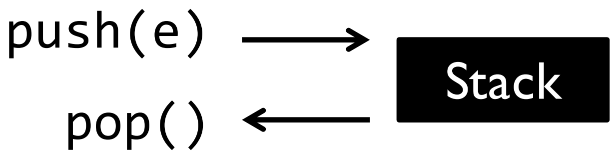
Basic Data Structures

- Fundamental data structures
  - Stack
  - Queue
  - Deque (double-ended queue)
- Collection of elements with specific behavior

Stack	Queue	Deque

Stack

- Last-in, first-out behavior
  - Push any number of elements onto a stack
  - Pop returns the most recently pushed element of stack



Queue

- First-in, first-out behavior
  - Enqueue any number of elements onto queue
  - Dequeue returns the oldest element in queue

`dequeue()` ← **Queue** ← `enqueue(e)`

Double-Ended Queue

- Complete Flexibility
  - Enqueue any number of elements to left or to right
  - Dequeue an element from left or right

Stack Choices In Python

- List

```
st = []
st.append(27)
v = st.pop()
```

- Deque

```
from collections import deque
st = deque()
st.append(27)
v = st.pop()
```

- LifoQueue

```
from queue import LifoQueue
st = LifoQueue()
st.put(95)
v = st.get()
```

Stack grows to the right here for efficiency

Queue Choices In Python

- List

```
st = []
st.append(27)
v = st.pop(0)
```

Remove 1st element in list

- Deque

```
from collections import deque
st = deque()
st.append(27)
v = st.popleft()
```

- Queue

```
from queue import Queue
st = Queue()
st.put(95)
v = st.get()
```

O'REILLY

Python Data Structures Packages Summary

- General-purpose `list` structure
  - Using a list as a queue is noticeably inefficient
- The `queue` module is thread-safe and supports multiple producers and consumers
  - Has additional overhead for locking
- `collections` is a high performance alternatives to Python built in `dict`, `list`, `set` and `tuple`

O'REILLY

Word Ladder Summary

- Compare times of three queue implementations
  - Queue is slowest because it is made for multi-threaded producer/consumer applications

Queue Implementation	List	BinaryArraySearch	Dictionary
Deque	178.98	8.19	1.43
list	187.83	8.43	1.73
Queue	185.84	9.11	2.41

O'REILLY

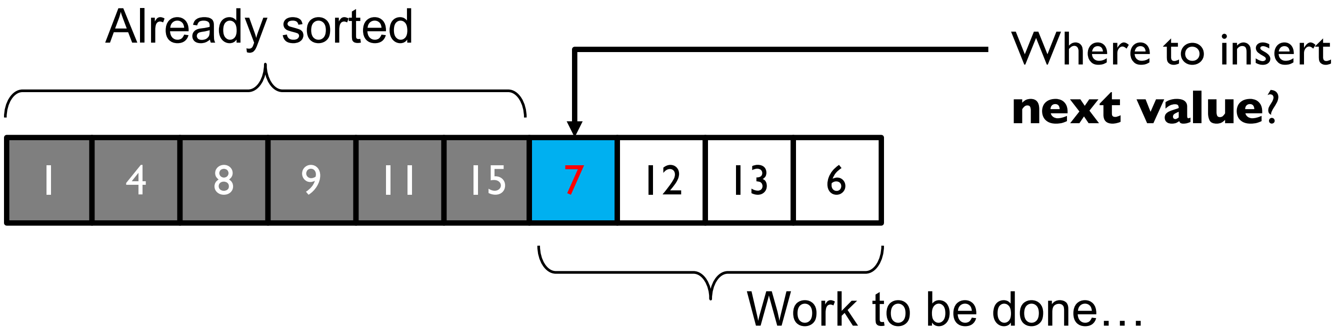
Sorting

- Fundamental problem in computer science
  - Practical application in most programs
- Input
  - A Python **list** of elements
  - Criteria for determining how to compare e_1 with e_2
- Output
  - **list** sorted <u>in place</u>

O'REILLY

INSERTION SORT

- Snapshot of partial progress of INSERTION SORT
  – Extend sorted sublist by inserting **next value** into proper place within this partially-sorted list



Insertion Sort

- Snapshot of partial progress of Insertion Sort
  - Extend sorted sublist by inserting **next value** into proper place within this partially-sorted list
  - Find its proper location by swapping from the right

1	4	8	9	11	15	7	12	13	6

Insertion Sort

- Snapshot of partial progress of Insertion Sort
  - Extend sorted sublist by inserting **next value** into proper place within this partially-sorted list
  - Find its proper location by swapping from the right

1	4	8	9	11	7	15	12	13	6

INSERTION SORT

- Snapshot of partial progress of INSERTION SORT
  – Extend sorted sublist by inserting **next value** into proper place within this partially-sorted list
  – Find its proper location by swapping from the right

1	4	8	9	7	11	15	12	13	6

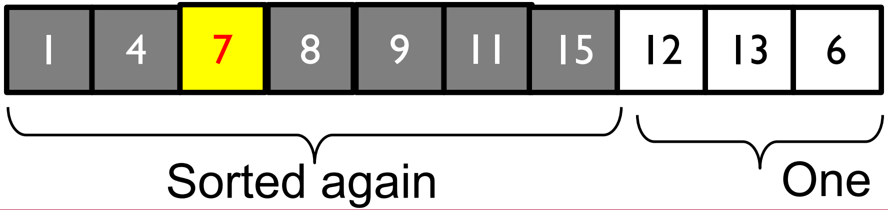
INSERTION SORT

- Snapshot of partial progress of INSERTION SORT
  - Extend sorted sublist by inserting **next value** into proper place within this partially-sorted list
  - Find its proper location by swapping from the right

1	4	8	7	9	11	15	12	13	6

INSERTION SORT

- Snapshot of partial progress of INSERTION SORT
  - Extend sorted sublist by inserting **next value** into proper place within this partially-sorted list
  - Find its proper location by swapping from the right



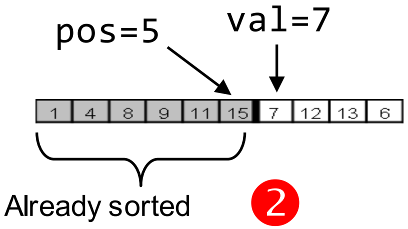
INSERTION SORT

- ❶ `A[0:i]` is sorted
- ❷ `val` at `A[i]` to be inserted into `A[0:i+1]`

❶ ❷

```
def insertionSort (A):
  for i in range(1, len(A)):
    pos = i-1
    val = A[i]
    while pos >= 0 and A[pos] > val:
      A[pos+1] = A[pos]
      pos -= 1
    A[pos+1] = val
```

At start
`A[0:1]` is sorted

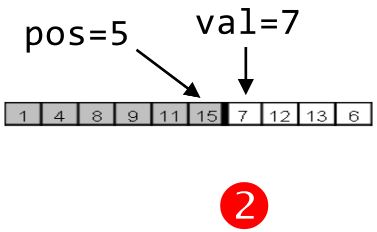


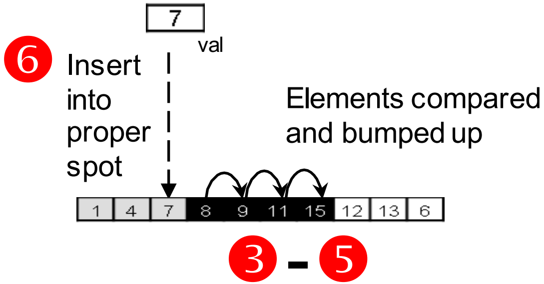
Important!
copy `val = A[i]`

INSERTION SORT

- ❶ `A[0:i]` is sorted
- ❷ `val` at `A[i]` to be inserted into `A[0:i+1]`
- ❸-❺ moves elements in A to the right to make room for `val`
- ❻ `val` in proper spot

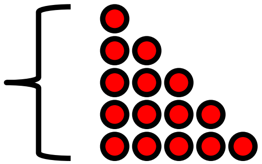
```
   def insertionSort (A):
     for i in range(1, len(A)):
❶      pos = i-1
❷      val = A[i]
❸      while pos >= 0 and A[pos] > val:
❹        A[pos+1] = A[pos]
❺        pos -= 1
❻      A[pos+1] = val
```





INSERTION SORT

$n-1$



$= 1 + 2 + 3 + \ldots + n\text{-}1$

$= (n\text{-}1)*(n)/2$

$= \frac{n^2 - n}{2}$

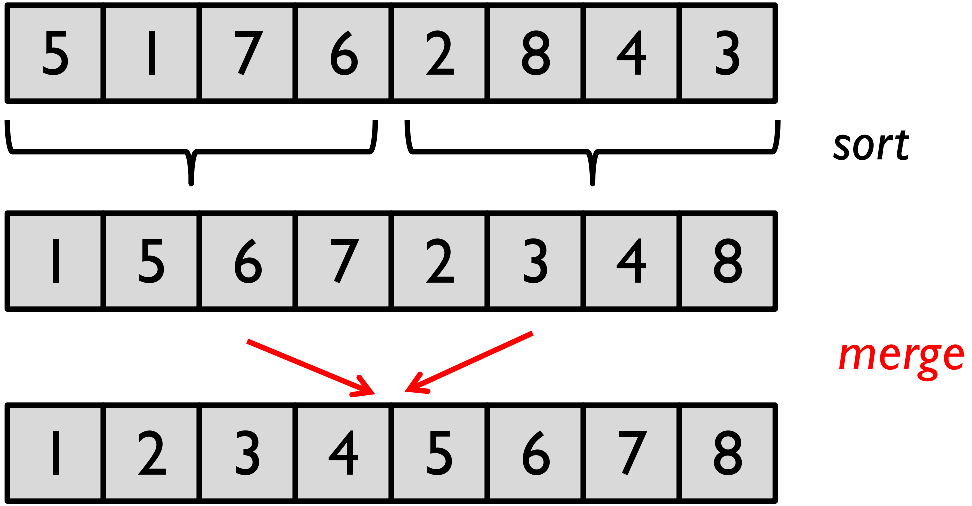
$= \frac{n^2}{2} - \frac{n}{2}$

$= O(n^2)$

- Iterates *n* times through the list
  - Reduces problem size by 1 with each pass
- Far too much swapping
  - Consider when initial list is in reverse order
  - N-1 loop iterations, swapping always
- Can we do better?

Divide And Conquer Algorithm Structure

- Problem subdivided into <u>two half-sized</u> problems



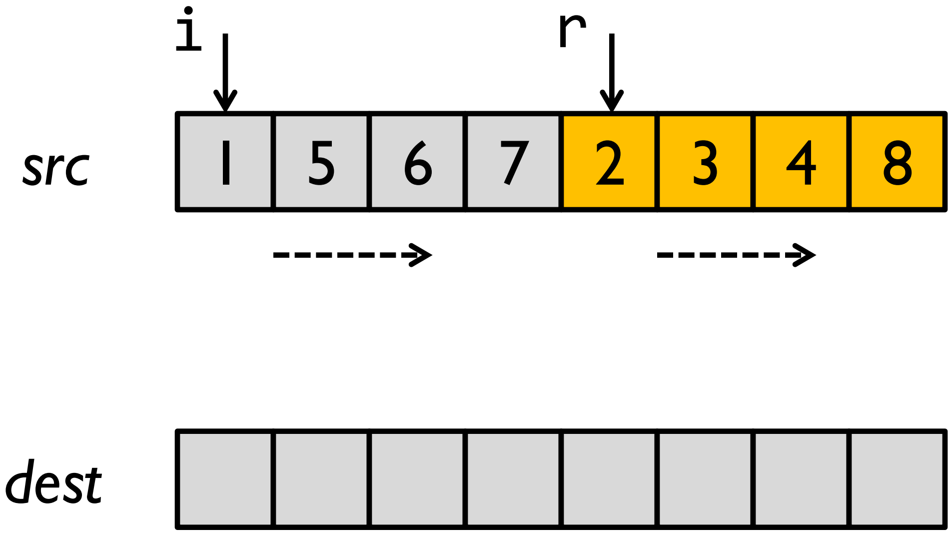
You need <u>additional space</u> equivalent to size of array being sorted

MERGESORT

- Recursively divide problem into two smaller problems
- Efficiently merge two sorted sub-lists with auxiliary storage

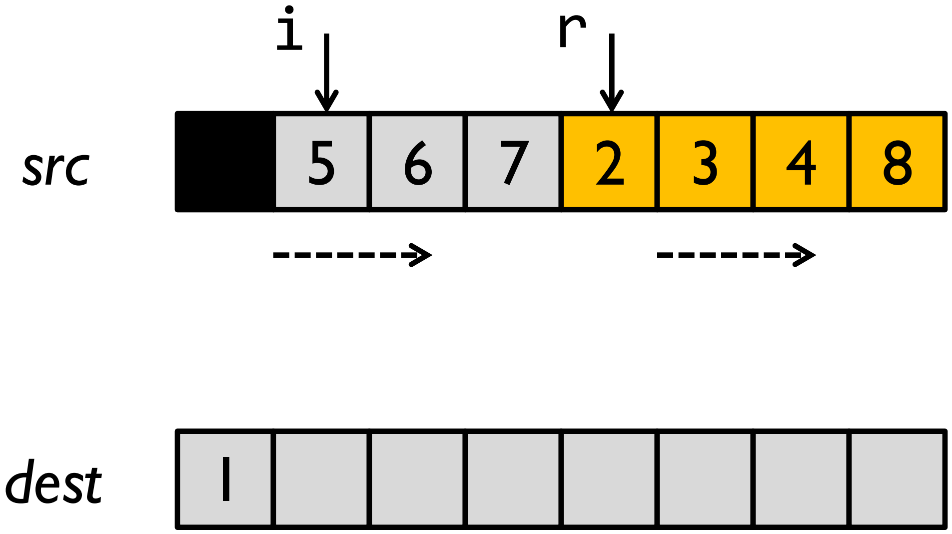
```
def mergeSort(A):
  msort(A, [None]*len(A), 0, len(A)-1)

def msort(A, aux, lo, hi)
  if hi > lo:
    mid = (lo + hi) // 2

    msort(A, aux, lo,     mid)
    msort(A, aux, mid+1, hi)

    merge(A, aux, lo, mid, hi)
```

Merge Two Sorted SubLists Into One

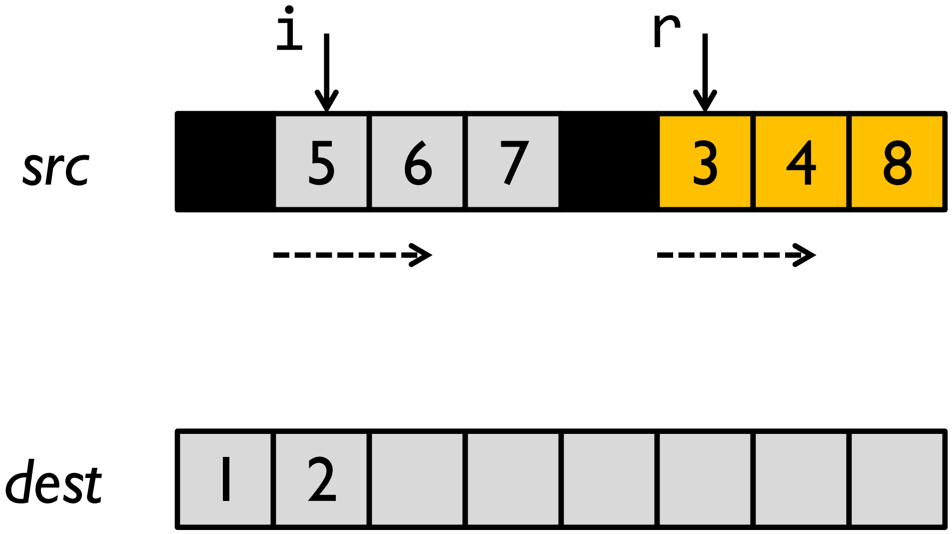


Need an auxiliary storage whose size is same as original

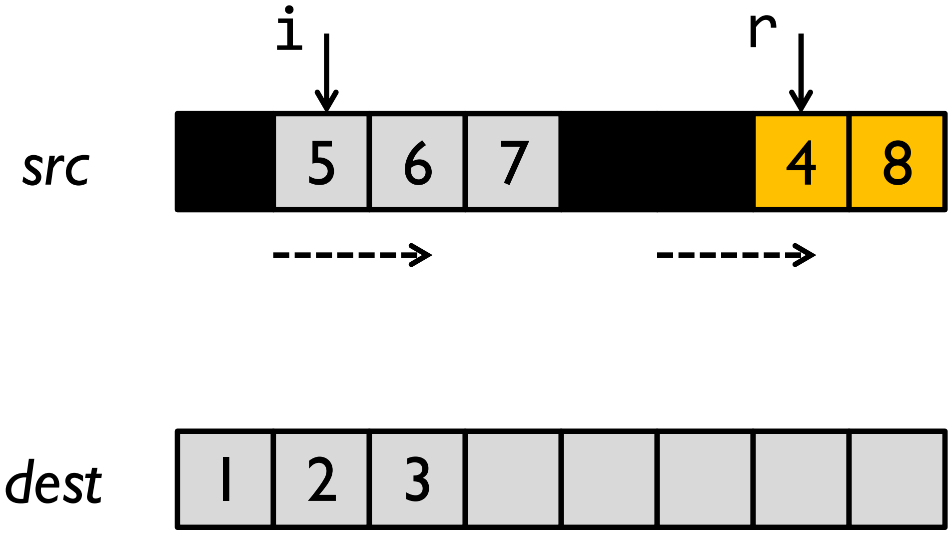
Merge Two Sorted SubLists Into One



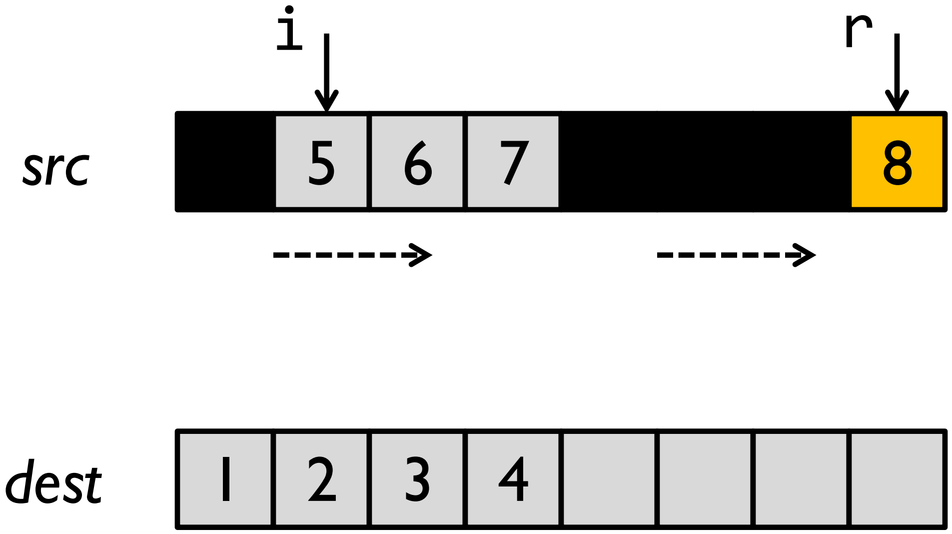
Merge Two Sorted SubLists Into One



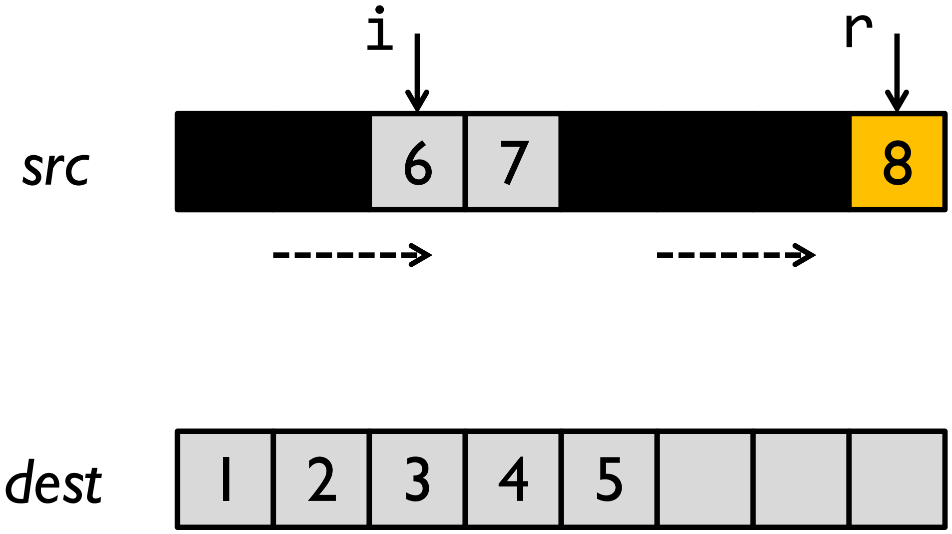
Merge Two Sorted SubLists Into One



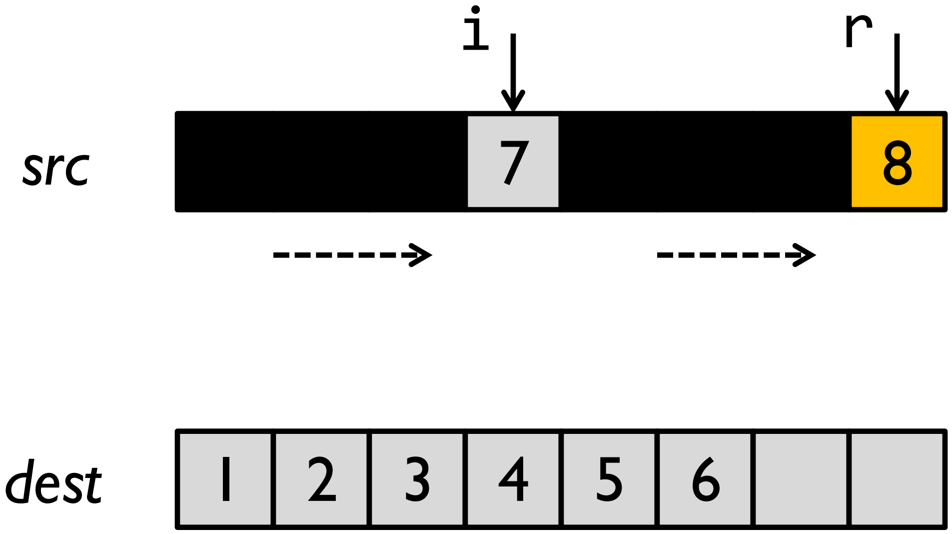
Merge Two Sorted SubLists Into One



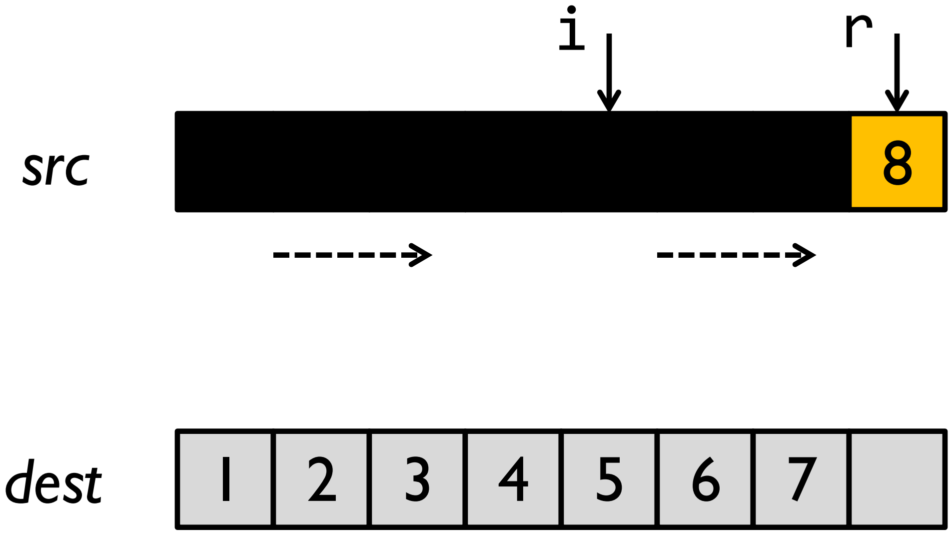
Merge Two Sorted SubLists Into One



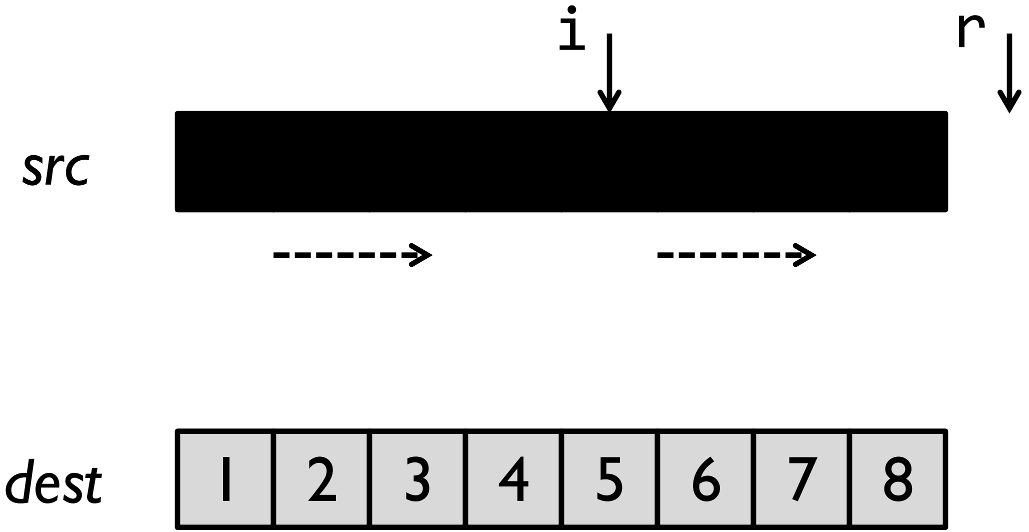
Merge Two Sorted SubLists Into One



Merge Two Sorted SubLists Into One



Merge Two Sorted SubLists Into One



Efficient Merge

- Copies `A[lo:hi+1]` into aux
  - Repeatedly computes `A[k]`
- Sweeps left to right through
  - `i > mid` exhausted left side
  - `r > hi` exhausted right side
  - $aux_r < aux_i$ take from right
  - `else` take from left

```
def merge(A, aux, lo, hi)
  for k in range(lo, hi+1):
    aux[k] = A[k]
  i = lo
  r = mid+1

  for k in range(lo, hi+1):
    if i > mid:
      A[k] = aux[r]
      r += 1
    elif r > hi:
      A[k] = aux[i]
      i += 1
    elif aux[r] < aux[i]:
      A[k] = aux[r]
      r += 1
    else:
      A[k] = aux[i]
      i += 1
```

O'REILLY

Timing Analysis

- Is MergeSort more efficient than InsertionSort?
  - InsertionSort reduces problem size by one with each pass, leading to $O(n^2)$ with only $O(1)$ extra storage
  - MergeSort reduces problem size in half with each recursive invocation, leading to $O(n \log n)$ using $O(n)$ extra storage

O'REILLY

Sorting Considerations

- Stable sort of A
  - If val_i = `A[i]` and val_j = `A[j]` are equal and `i` < `j` …
  - When A is sorted, final location of val_i in A is to left of val_j
  - MERGESORT and INSERTIONSORT are stable
- Comparing `A[i]` < `A[j]` may be expensive
  - How to minimize number of comparisons?

TimSort

- Implemented by Tim Peters in 2002 for Python
  - Finds subsequences that are already ordered
  - Uses that knowledge to sort remainder efficiently
- Full details
  - https://hg.python.org/cpython/file/tip/Objects/listsort.txt
  - Useful when adding data to sorted list
  - `newList = sorted(oldSortedList + newData)`

Comparison To Sorting Methods

- Real-world data has running sequences
  - Trials of Sorting (S ++ ND)
  - S is sorted list of N items
  - ND is N/4 random new items
- TIMSORT is astounding
  - 100x faster than MERGESORT
  - In this special case

N	Insertion	Merge	Tim
16	0.0016	0.0038	0.0001
32	0.0045	0.0085	0.0001
64	0.016	0.0165	0.0002
128	0.0583	0.0351	0.0004
256	0.2195	0.081	0.0006
512	0.915	0.1907	0.0012
1024	3.6916	0.3986	0.0027
2048	14.7147	0.8566	0.0063
4096	*	1.8115	0.0147
8192	*	3.8731	0.0332
16384	*	8.1721	0.0923
32768	*	17.0643	0.1968

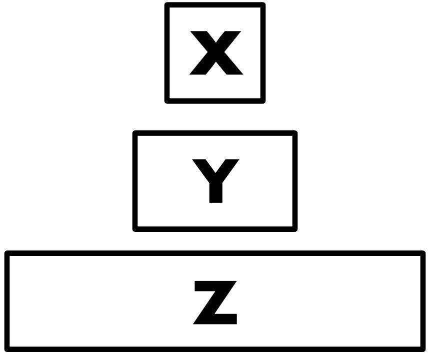
TimSort

5	1	7	6	2	8	4	3

- Scan list from left to right to identify *runs* of at least two elements
  - *Non-descending* – each subsequent element is ≥ last
  - *Strictly descending* – each subsequent element is < last
- Each identified run is pushed onto a task stack
  - Requires up to N/2 auxiliary storage
  - Why? Any two elements are either (N-D) or (S-D)

TimSort Merges Runs on Stack

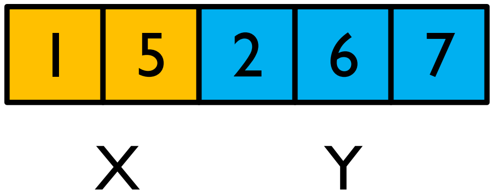
- Consider three runs **X**, **Y**, **Z** on top
  - |**Y**| > |**X**| and |**Z**| > |**Y**| + |**X**|
- If push of X violates invariants
  - Merge Y with smaller of X and Z
  - Repeat until invariants hold again
  - Continue forming runs until done with data
- Once done, repeatedly merge top two runs on stack



O'REILLY

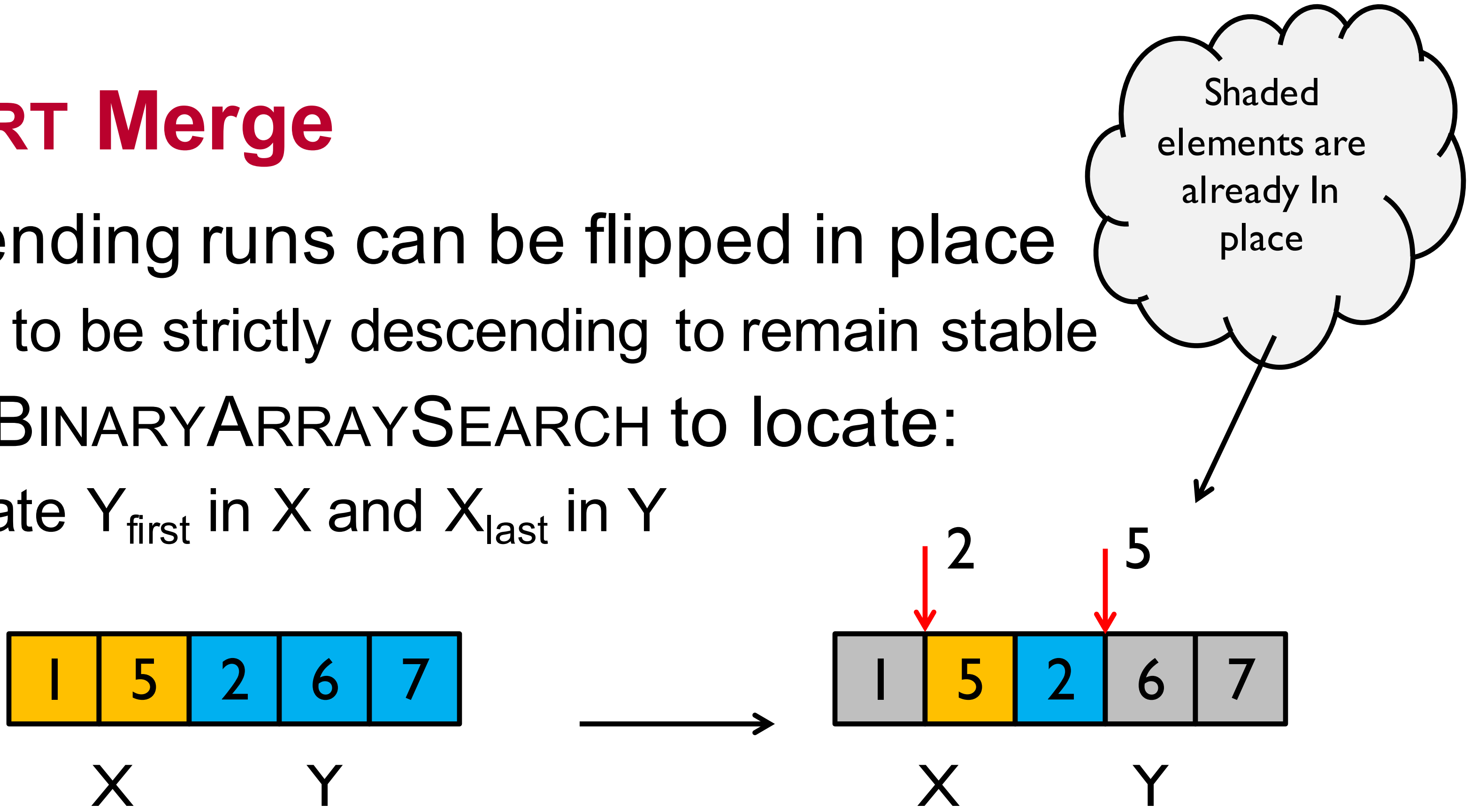
TimSort Merge of Runs X and Y

- Descending runs can be flipped in place
  - Has to be strictly descending to remain stable
- Uses BinaryArraySearch to locate:
  - Locate Y_{first} in X and X_{last} in Y



TIMSORT Merge

- Descending runs can be flipped in place
  – Has to be strictly descending to remain stable
- Uses BINARYARRAYSEARCH to locate:
  – Locate Y_{first} in X and X_{last} in Y

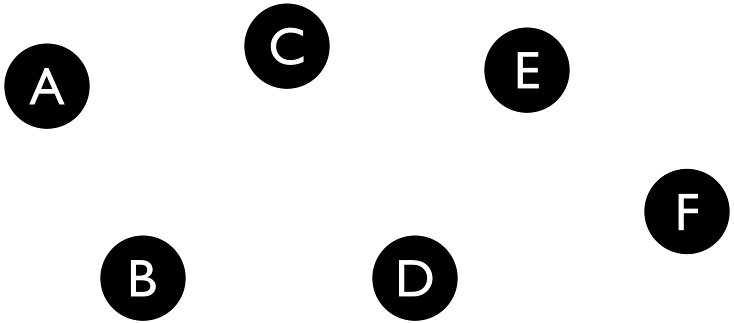


TimSort Summary

- TimSort optimizes use of InsertionSort and MergeSort
  - Smaller data sets [get size]
  - Highly ordered data (quite common) up to 25x faster
- Difficult to implement
  - In 2015 formal verification detected a defect in standard implementation

Graph Data Structure

- Introduce terms and concepts
- Nodes (also called Vertices)
  - Named
  - Unique in graph



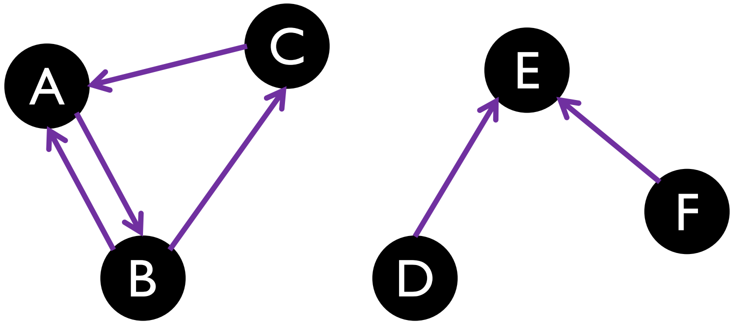
O'REILLY

Graph Data Structure

- Introduce terms and concepts
- Nodes (also called Vertices)
  - Named
  - Unique in graph
- Edges
  - Connect nodes

<u>Simple Graphs</u>

1. *Edges connect exactly two nodes*
2. *No self-loops*



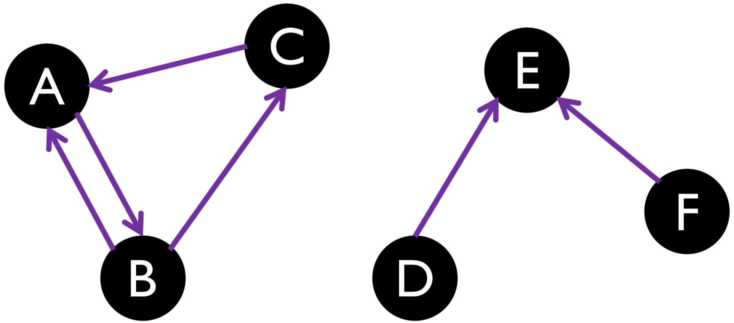
O'REILLY

Graph Data Structure

- Introduce terms and concepts
- Nodes (also called Vertices)
  - Named
  - Unique in graph
- Edges
  - Connect nodes

Common Graph Families

1. *Directed Graph consists of edge from u to v*
2. *Note arrow head on each edge*

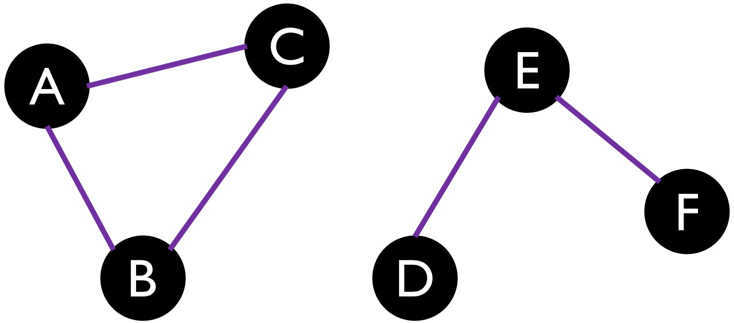


Graph Data Structure

- Introduce terms and concepts
- Nodes (also called Vertices)
  - Named
  - Unique in graph
- Edges
  - Connect nodes

<u>Common Graph Families</u>

1. *Undirected Graph consists of edge <u>between</u> u <u>and</u> v*
2. *Represents symmetric or bi-directional relationship*

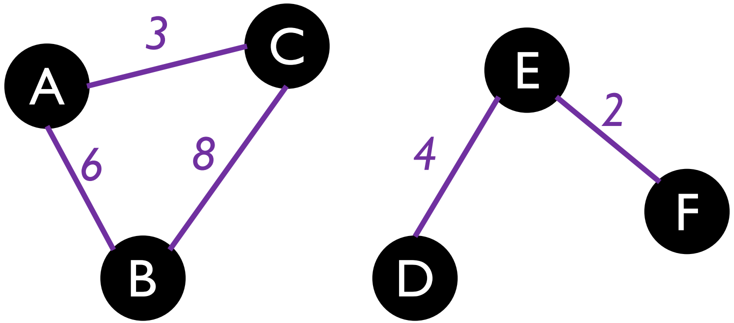


Graph Data Structure

- Introduce terms and concepts
- Nodes (also called Vertices)
  - Named
  - Unique in graph
- Edges
  - Connect nodes
  - Can have labels

Common Graph Families

1. *Undirected Weighted Graph consists of edge between u and v*
2. *Edge contains numeric weights*



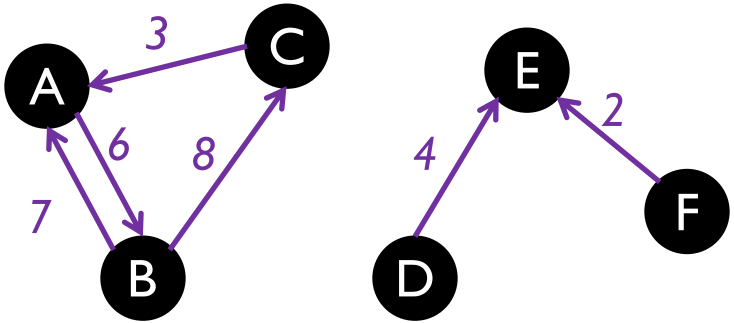
O'REILLY

Graph Data Structure

- Introduce terms and concepts
- Nodes (also called Vertices)
  - Named
  - Unique in graph
- Edges
  - Connect nodes

Common Graph Families

1. *Directed Weighted Graph consists of edge from u to v*
2. *Edge contains numeric weights*



Do Not Implement Graph Data Structure Use NetworkX Python Library

- `List` and even dictionary not effective
  - Hard to capture binary relationships between nodes
- NetworkX first released in 2005
  - Currently at version 2.3 and quite stable
- Use pip to install (takes just a few seconds)
  - `pip3 install networkx==2.3`

Manipulate Graphs in NetworkX

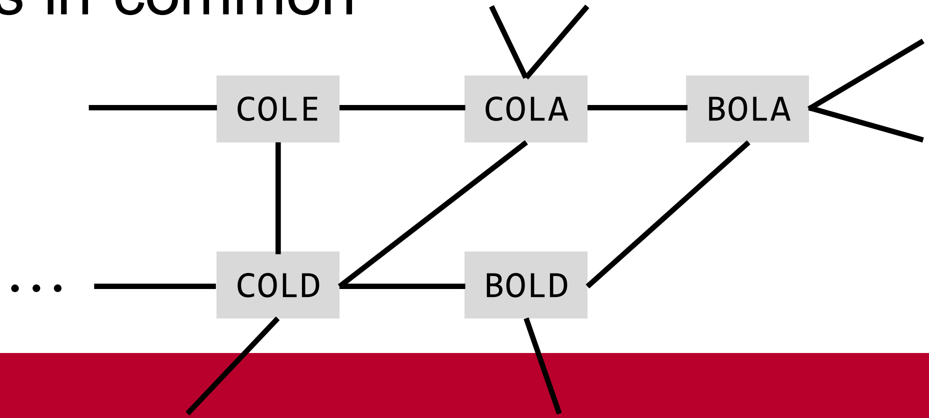
- Create graphs
  - `G = nx.Graph()`
  - `DG = nx.DiGraph()`
- Add nodes
  - `G.add_node(id)`
  - `G.add_nodes_from([id1, id2])`
- Add edges

```
import networkx as nx
G = nx.Graph()
G.add_edge(n1, n2, object=x)
G.add_edge(1, 2, weight=4.7)
```

O'REILLY

Word Ladder Exercise

- Revisit this as a graph problem
  - Each node represents a four-letter word
  - An undirected edge exists between two nodes that have three letters in common



Tasks To Solve #1

- Compute Word Ladder from w_1 to any w_2
  - Not just a single Word Ladder
  - Construct a graph that can be used to answer such requests from any two four-letter words
- Graph searching algorithms will be useful

Tasks To Solve #2

- Find words not involved in any Word Ladder
  - Once graph is constructed, find all nodes that have no edges
  - Do this by checking each of the nodes
  - Performance will be O(*n*) which is quite efficient

Question: What is time complexity if you only had list of n words?

O'REILLY®

Tasks To Solve #3

- Determine longest Word Ladder that exists
  - For any two four-letter words
- Different kind of problem
  - For all nodes (*u*, *v*) you want to compute word ladder
  - Then find the one that is longest

O'REILLY

Tasks To Solve #4

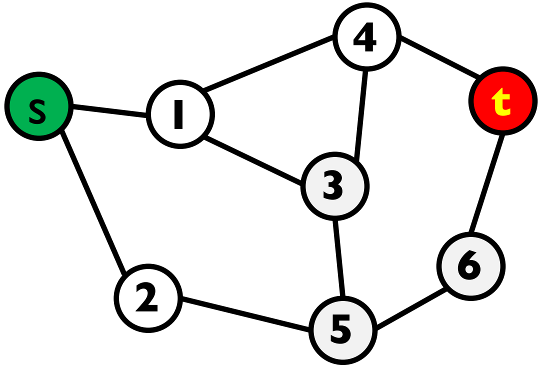
- Are there “islands” of non-connectable words
  - Can you form a Word Ladder from AAHS to any other four letter word?
  - Disjoint subsets of words $a_i \in A$ and $b_i \in B$ where Word Ladder exists between any a_i and a_k but not between a_i and b_j
  - Ignore the words not part of any Word Ladder (#2)

Graph Algorithms To Cover

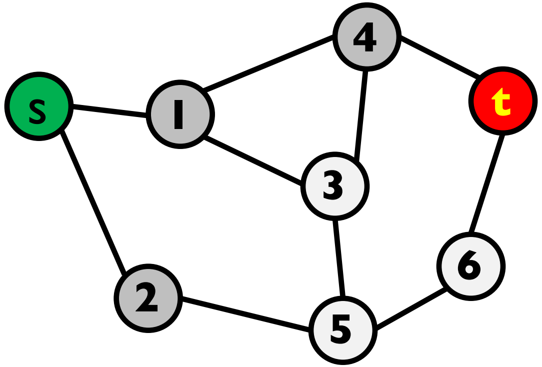
- Searching
  - DepthFirstSearch over Graph
  - BreadthFirstSearch over Graph
- Graph Processing
  - AllPairsShortestPath

Searching Through Graphs
Key Concepts

- Source node
  - Start of the search
- Target node
  - Destination of search, if known in advance

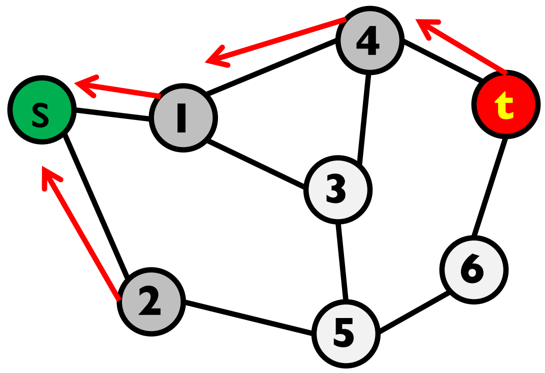


Searching Through Graphs
Key Concepts



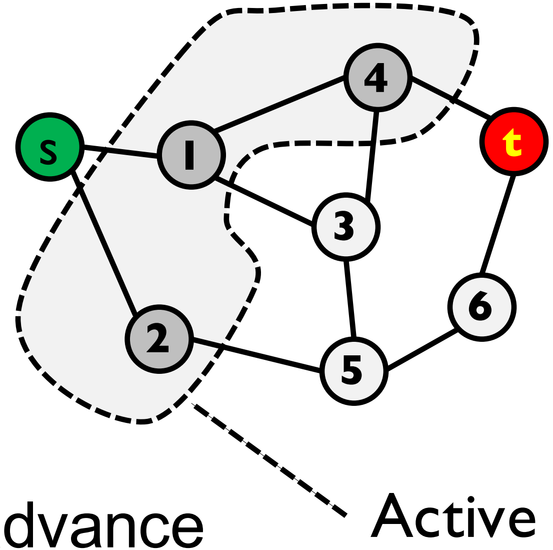
- Source node
  - Start of the search
- Target node
  - Destination of search, if known in advance
- Visited nodes
  - Use dictionary for efficient look-up

Searching Through Graphs
Key Concepts



- Source node
  - Start of the search
- Target node
  - Destination of search, if known in advance
- Visited nodes
  - Use dictionary for efficient look-up
- Predecessor link – to record path *in reverse*

Searching Through Graphs
Key Concepts



- Source node
  - Start of the search
- Target node
  - Destination of search, if known in advance
- Active state
  - The "search horizon" for algorithm
  - At each step, algorithm removes a state to explore further

O'REILLY

Searching Through Graphs Fundamental Strategies

- Find shortest path from src to target
  - BREADTHFIRSTSEARCH uses queue as we have seen
  - Methodical approach to searching
- Find whether a path exists from src to target
  - DEPTHFIRSTSEARCH uses stack
  - Solutions can become quite long

BreadthFirst Search

- Uses Queue
  - Words to be searched
  - In an order that ensures shortest path found
- Extra storage needed
  - Visited stores past
  - Pred records path

```
active = deque()
active.append(start)
visited[start] = True
pred[start] = None
while active:
  u = active.popleft()
  if u == end:
    return trail(pred, end)

  for n in G.neighbors(u):
    if not n in visited:
      visited[n] = True
      pred[n] = u
      active.append(n)
return None
```

DepthFirst Search

- Uses Stack
  - Words to be searched
  - Arbitrary order
- Same overall approach
  - Much longer solutions
  - 'COLD' to 'WARM' in 392 steps

DEPTHFIRSTSEARCH

```
active = deque()
active.append(start)
visited[start] = True
pred[start] = None
while active:
   u = active.pop() # Act as Stack
   for n in G.neighbors(u):
      if not n in visited:
         visited[n] = True
         pred[n] = u
         if n == end:
           return trail(pred, end)

         active.append(n)
return None
```

Solve Task #3 Longest Word Ladder

- How to find longest possible Word Ladder between any two words

$$\frac{n * (n-1)}{2}$$

  - N = 5,875 words
  - Do we really have to check each of the possible 17,254,875 Word Ladders?
  - ALLPAIRSSHORTESTPATH to the rescue

AllPairsShortestPath

- Computes what seems to be a harder problem
  - For any two nodes, this computes the distance of the shortest path between those two nodes

```
results = dict(nx.all_pairs_shortest_path(G))
```

  - `results[s][t]` is length of shortest path from `s` to `t`

All Pairs Shortest Path

- Dynamic Programming
  - Try each of the n^3 possible (u, k, v)
  - Update if dist(u,k) + dist(k,v) is less than dist(u,v)
  - Once done, locate largest value in `dist[ui][vi]`

```
def allPairsShortestPath (G)
  for ui in range(n):
    for vi in range(n):
      dist[ui][vi] = sys.maxsize

  dist[ui][ui] = 0
  u = allNodes[ui]
  for v in G.adj[u]:
    vi = index[v]
    dist[ui][vi] = 1     # Edge exists

  for ki in range(n):
    for ui in range(n):
      for vi in range(n):
        newLen = dist[ui][ki] + dist[ki][vi]
        if newLen < dist[ui][vi]:
          dist[ui][vi] = newLen
```

O'REILLY

Solve Task #4 Disjoint Subsets

- Use DepthFirstSearch without a known target
  - Explore until all nodes are visited
  - Repeat process on any unvisited nodes with edges

```
AAHS -> 5807 with sample of ['AAHS', 'HAHS', 'HEHS', 'PEHS']
EPPY -> 2 with sample of ['EPPY', 'ESPY']
ERYX -> 4 with sample of ['ERYX', 'ORYX', 'ONYX', 'ONYM']
GEGG -> 2 with sample of ['GEGG', 'YEGG']
```

O'REILLY

Graph Summary

- Lots of other graph algorithms to explore
- Weighted graphs offer different problems
  - Shortest path by accumulated edge weights
  - When edge weights can include negative numbers, other strategies necessary

O'REILLY

Data Structure Summary

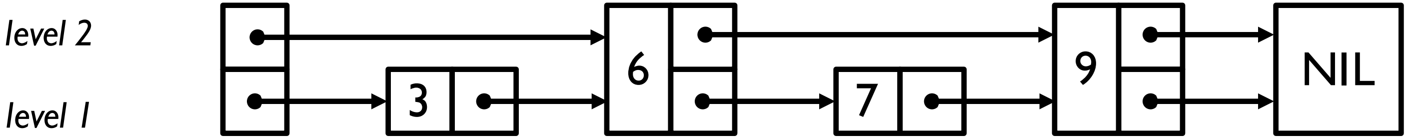
- List
- Stack
- Queue (and Deque)
- Graph (Directed and Undirected)
- Together with Python packages to use

SkipList Implementation

- Implementation
  - https://pypi.org/project/pyskiplist/
- A Probabilistic Alternative to Balanced Trees
  - ftp://ftp.cs.umd.edu/pub/skipLists/skiplists.pdf
- Install using pip
  - `pip install pyskiplist`

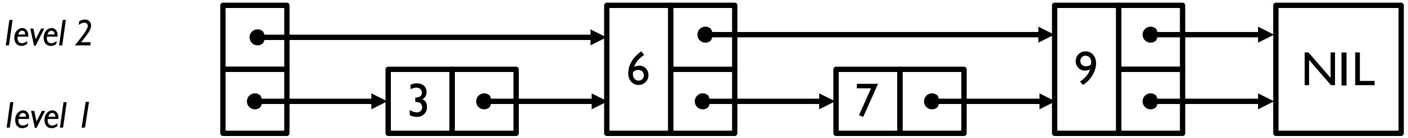
Novel Structure For Storing Lists

- Each element is represented by a node
  - Each node has a level *i*
  - Each node has *i* forward pointers



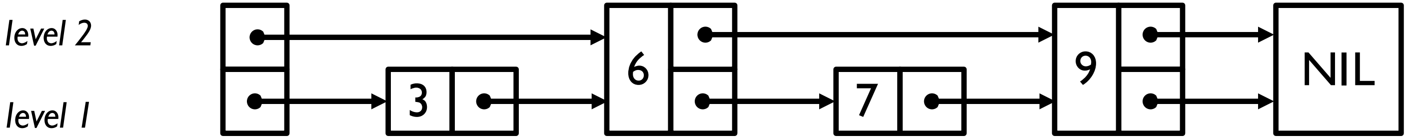
Searching Through SkipList (look for 7)

- Traverse forward pointers that do not overshoot
  - Start at top level (Level 2)
  - If it exists in list, drop down to lower level between <u>6</u> and <u>9</u>
  - If you cannot find it at lowest level, then not in list



Inserting Into SkipList (insert 8)

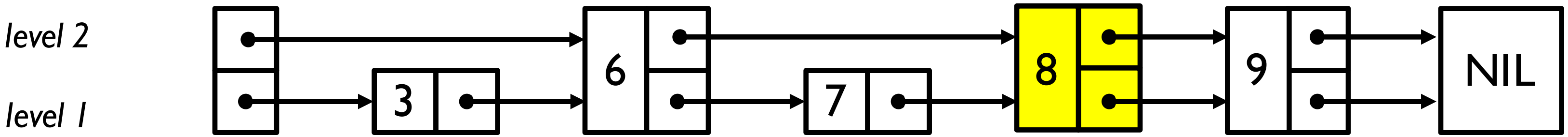
- Search until you…
  - Find node for element; or
  - Find node *n* on lowest level 1 that is smaller but *n.next* is greater
  - Randomly choose level into which to insert new value



O'REILLY

Inserting Into SkipList

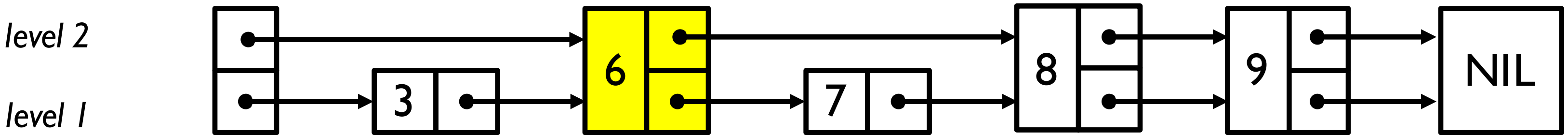
- Search until you…
  - Find node for element; or
  - Find node *n* on lowest level 1 that is smaller but *n.next* is greater
  - Randomly choose level into which to insert new value



O'REILLY

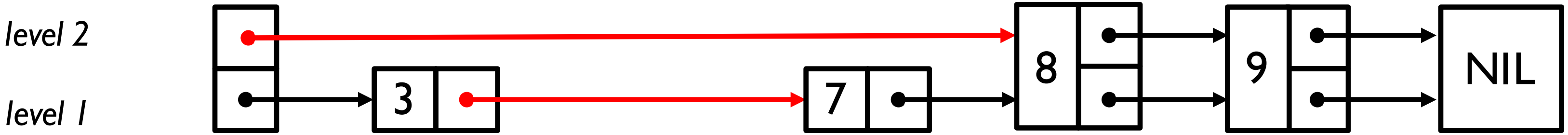
Deleting From SkipList (delete 6)

- Search until you find node with element
  - Splice out the node and reattach pointers
  - If top level becomes empty, reduce level by 1



Deleting From SkipList (delete 6)

- Search until you find node with element
  - Splice out the node and reattach pointers
  - If top level becomes empty, reduce level by 1



SkipList

- Review code
- Provides comparable performance to balanced binary trees with less programming effort
- Can easily be converted to store (key, value) pairs with each node
  - Becomes a dictionary structure

Conclusion

- Working with algorithms requires a solid understanding of fundamental data structures
- Do not reinvent the wheel
  - Use available high-quality code libraries
- Evaluate your code on sample problems of varying size
  - Identify time complexity empirically