PL Fundamentals

What are the three main strategies to translate source code to machine language?

- **Ahead-of-Time (AOT) Compilation**: Source code translated to machine code before run-time
  - Usually has better performance since more optimizations can be performed, and has already been translated.
- **Interpreted**: Code translated on the fly, line-by-line during execution.
  - Advantages: easier to implement (writing compilers is hard), no compilation stage. Only necessary, actually executed code is translated.
- **Just-In-Time (JIT) Compilation**: Compilation is during runtime, rather than before execution. Combines speedup of AOT compilation with flexibility interpretation. In particular, lines initially interpreted, but those which are commonly executed will be compiled. This trade-off is continually, dynamically analyzed by the JIT compiler. Unlike AOT, has access to dynamic runtime info.

Ideally, machine translation strategies should be abstracted from the language itself:

- Java supports both AOT and JIT, depending on the compiler used
- Python can both be interpreted or compiled (into C, via Cython)

What are the 3 ways that a programming language can be typed?

- **Static**: Types checked before run-time (for example, during compilation)
  - Faster, since types have already been checked
- **Dynamic**: Types checked on the fly (during run-time)
- **Duck**: The general idea is “we don’t care about its type, we care about what it can do”. There’s no checking for a common interface or specs of the object; the interpreter just tries to call objects’ methods and executes the statement. This makes type errors more prone to crashing at runtime, instead of failing at compilation, if there are issues (eg method not found). However, the advantage is that it makes things a lot simpler.

Describe the following programming paradigms: declarative, functional, imperative/procedural, and object oriented.

- **Declarative**: Expressing the logic of computation without describing control flow. Order doesn't matter. Examples are SQL and HTML.
- **Functional**: A style of programming which treats computation as mathematical functions and avoids changing-state and mutable data. This style is good for parallelism and recursion. Gets its origin from Church numerals, the lambda calculus, and mathematical logic. An example is Haskell.
- **Imperative/Procedural**: Uses statements that change a program's state and describing how a program's flow operates. Examples are C, C++, Java. Gets its origin from Turing’s finite state machines.
- **Object Oriented**: Based on object classes (factories) and object instances

State the (standard) compilation, typing, and paradigm of the following languages: C, Java, Python, Haskell.

C:

- AOT compiled. Imperative, with static typing.

Java:

- JIT compiled, but can also support AOT. Imperative, with static typing

Python:

- Interpreted. Supports functional, OO, and imperative paradigms. However, most python is coded in an imperative way. Dynamic and duck typing, with no advance type declaration.
- **These properties (dynamic typing, interpreted) make python useful for prototyping, but more difficult to be used in big projects**
    a. **Compiling & type checking finds a lot of bugs vs linting.**
- On “Functional Python”: For example, you can write functions which are “pure”, which don’t change the state (ie, doesn’t change the input or data that exists outside the functions scope). You can also use immutable data structures (like tuples), and use higher order functions which take a function as the argument. However, python is not the best for purely functional programming since side effects can occur if one is not careful.
- On “OO Python”: You have things like classes, polymorphism, etc in python. But, python doesn’t support encapsulation, which many believe is a primary requirement for OO.

Haskell:

- AOT compiled. Purely functional, with static typing.

<u>What are the four basic principles of OOP?</u>

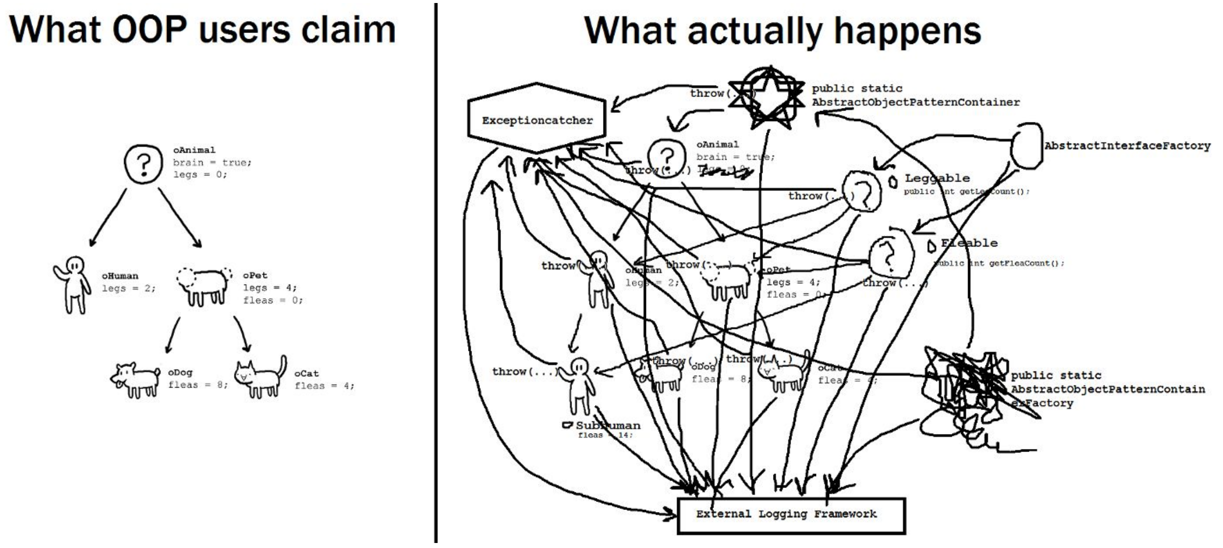
- **Encapsulation**: Private & public methods
- **Abstraction**: Don’t need to know inner details to use class
- **Inheritance**: To reuse code
- **Polymorphism**: Ability of one function to perform in different ways.

What are the SOLID principles in OOD?

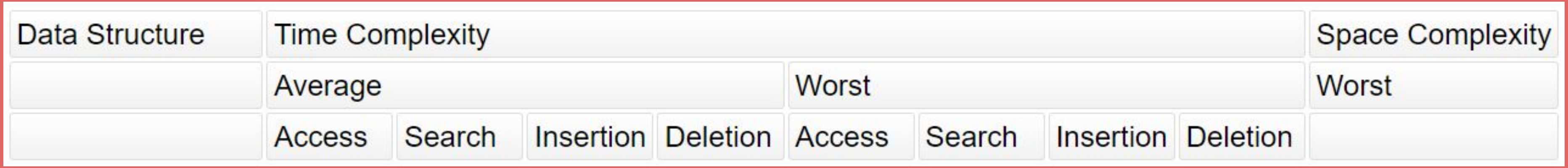
- **Single-responsibility principle**: very class should have only one responsibility
- **Open–closed principle**: Software entities should be open for extension, but closed for modification.
- **Liskov substitution principle:** "Functions that use pointers or references to base classes must be able to use objects of derived classes without knowing it (design by contract)
- **Interface segregation principle**: Many client-specific interfaces are better than one general-purpose interface
- **Dependency inversion principle**: Depend upon abstractions, concretions.

What are some issues with OOP?

- Banana gorilla jungle problem
  - The problem with object-oriented languages is they've got all this implicit environment that they carry around with them. You wanted a banana but what you got was a gorilla holding the banana and the entire jungle
  - Too much overhead, deep class hierarchies, hard to navigate/reason about
- Real world doesn't always break down into neat categories with well-defined properties
- Composition is sometimes better than inheritance
- In recent years, functional programming has had more "hype" than oop



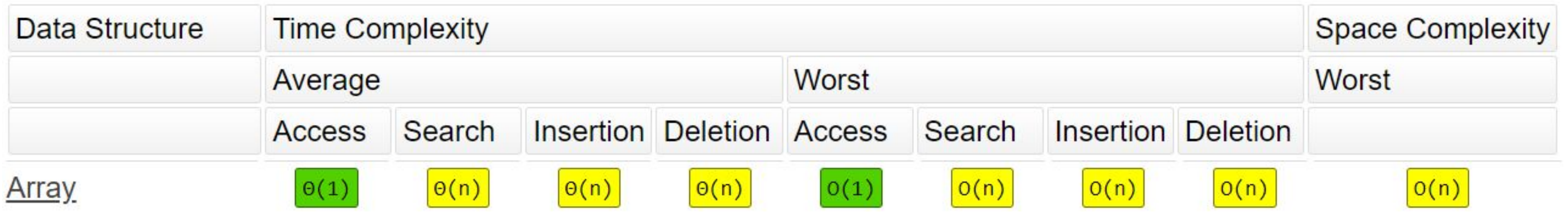
Data Structures
&
Algorithms



Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	

Define the complexity of lists. Is it a dynamic or static array (and, what’s the difference between them)?

Static arrays require you to state the size up front, and cannot change. **Dynamic arrays** (the case in python) have an “underlying capacity”, which can automatically grow as you add items to it. Both use **contiguous memory** to increase ease of, reduce overhead, and speed up, access. However, in general, contiguous memory can cause memory to be wasted due to “gaps”.



Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	Θ(1)	Θ(n)	Θ(n)	Θ(n)	O(1)	O(n)	O(n)	O(n)	O(n)

Notes:

- Largest costs come from growing beyond the current allocation, and inserting/deleting near the beginning (since everything after must move)
  - Eg, Append or pop **at end** is O(1) amoritized (but, O(n) if a resize is needed)

Define the behavior and complexity of stacks, and how to implement in python using lists and deques.

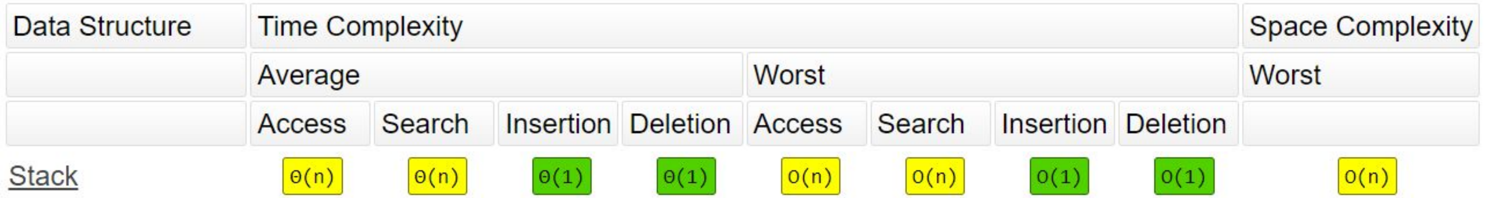
A stack is last-in, first out (LIFO) with space complexity O(n). It can be implemented using a list S, where:

- Push: S.append(e), O(1)
- Pop: e = S.pop(), O(1)
- Top: S[-1], O(1)

However, a O(n) operation is needed if the list capacity is exceeded.

Alternatively, one can use append() and pop() collections.deque, which is based of a linked-list and avoids this:

len(D)	Number of elements
D.appendleft(e)	Add to beginning
D.append(e)	Add to end
D.popleft()	Remove from beginning
D.pop()	Remove from end
D[i]	Arbitrary access
D.remove(e)	Find and remove element e
D.insert(i, e)	Insert element e at index i
del D[i]	Remove element at index i



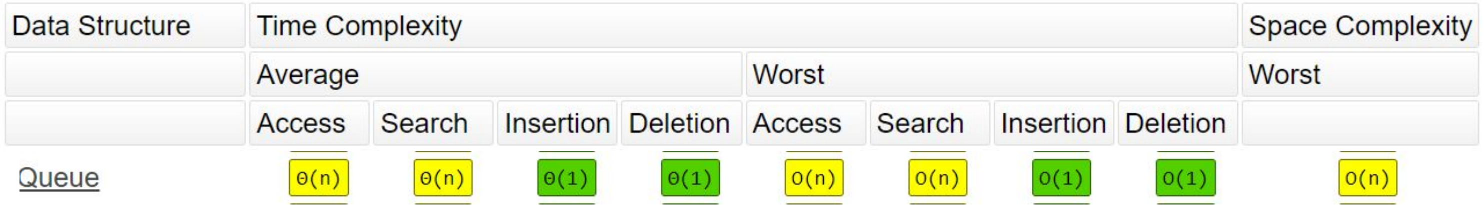
Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Stack	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)

Define the behavior and complexity of queues, and how to implement one in python using a list and deque.

A queue is first in, first out (FIFO) with space complexity O(n). Enqueue and dequeue have time complexity O(1). There are several ways to use it in python:

1. Using a list L, where you L.append(e) to enqueue and use L.pop(0) to dequeue.
    a. This is not very efficient, since pop(0) causes the list to be recreated which takes O(n)
2. Collections.deque

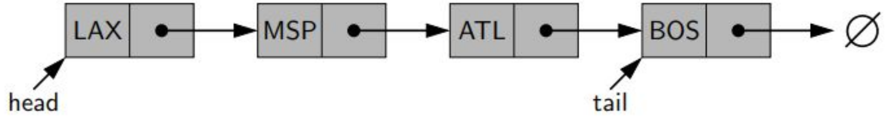
len(D)	Number of elements
D.appendleft(e)	Add to beginning
D.append(e)	Add to end
D.popleft()	Remove from beginning
D.pop()	Remove from end
D[i]	Arbitrary access
D.remove(e)	Find and remove element e
D.insert(i, e)	Insert element e at index i
del D[i]	Remove element at index i

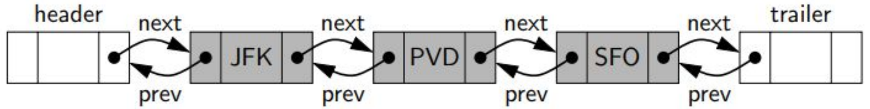


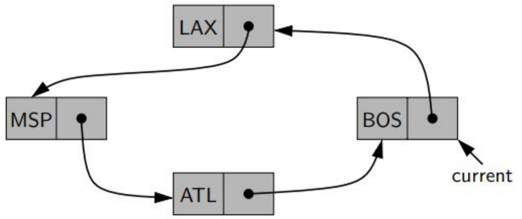
Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Queue	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)

Define the behavior and complexity of linked lists, and how to implement one in python. Compare singly, circular, and doubly linked lists.

- A **singly linked list** is a collection of nodes each of which store an element and a reference to the next node. The nodes are not contiguous in memory, so unlike lists, inserting at the beginning takes O(1). However, you cannot efficiently delete a node that is not the head.
- A **circularly linked list** can be useful if there is no notion of beginning or end.
- A **doubly linked list** has more symmetry, and you can efficiently delete any node in O(1) , if given a reference to it. Usually, a sentinel header/trailer is used to make implementation simpler.
- However, unlike arrays, you can't access the contents of an index in O(1); you need O(n).

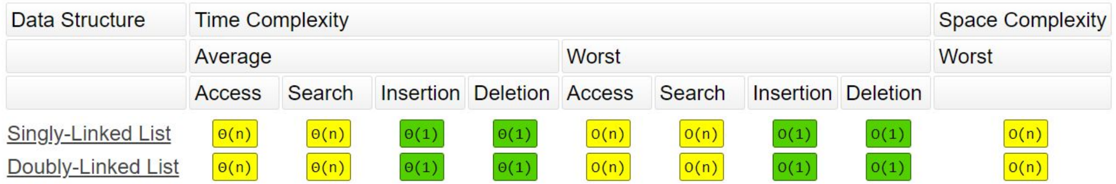






Linked lists can be implemented with the collections.deque class; this is an implementation of double ended queues based on linked lists.

len(D)	Number of elements
D.appendleft(e)	Add to beginning
D.append(e)	Add to end
D.popleft()	Remove from beginning
D.pop()	Remove from end
D[i]	Arbitrary access
D.remove(e)	Find and remove element e
D.insert(i, e)	Insert element e at index i
del D[i]	Remove element at index i



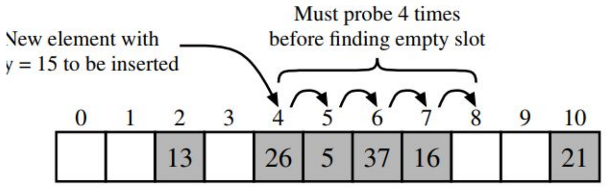
Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Singly-Linked List	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$
Doubly-Linked List	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$

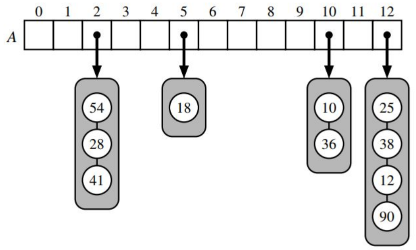
Define the behavior and complexity of hash tables, and how to implement one in python.

- A **hash table** contains a contiguous array of N memory locations. Its implemented as a dict in python.
- Given a pair of {immutable key k, value v}, we use a hash function hash(k) to map variable-length k into fixed locations in {0,...,N-1}
- In the case of collisions, there are two options:
    - **Separate chaining**, where each bucket has its own secondary container. Assuming a good hash function, the core operations take O(n/N), where n/N is the **load factor** (n = # of values).
    - **Open Addressing**, which requires that the load factor is always <=1 and items are stored directly in the array. This can take the form of **linear/quadratic probing**. However, these are subject to bad clustering patterns. So, python uses **random probing**, which utilizes an RNG giving a repeatable but somewhat arbitrary sequence of probes.
- Note that in all cases, both the key and value are stored so that the correct value can be retrieved using the key.
- Unlike hashes in cryptography, they need not be irreversible or secure, but does need to be efficient and disperse into n bins with uniform probability
    - In python, different objects (eg strings, floats, etc) have a different hash function (with magic method __hash__()).
- When the load factor is too high, python will double the size of the underlying array and rehash the current entries using the new size.

Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	Θ(1)	Θ(n)	Θ(n)	Θ(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Hash Table	N/A	Θ(1)	Θ(1)	Θ(1)	N/A	O(n)	O(n)	O(n)	O(n)

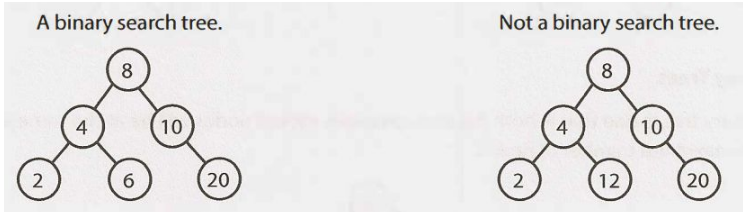
Operation	List	Hash Table	
		expected	worst case
__getitem__	$O(n)$	$O(1)$	$O(n)$
__setitem__	$O(n)$	$O(1)$	$O(n)$
__delitem__	$O(n)$	$O(1)$	$O(n)$

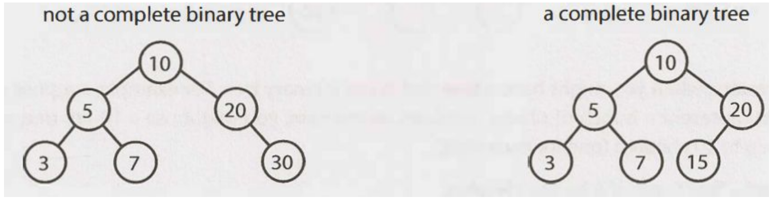


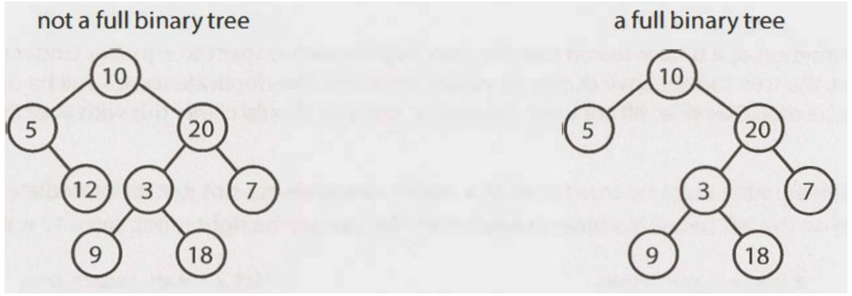


Define Tree, binary tree, binary search tree. What’s the difference between complete, full, and perfect binary trees?

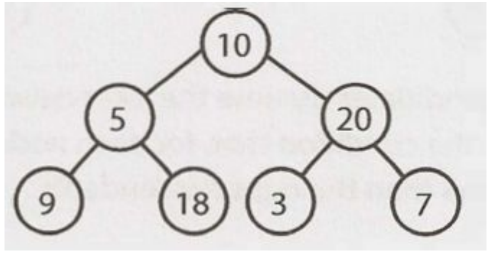
- A **tree** is a data structure which has a root node, and recursively, each child node has >=0 child nodes.
- A **binary tree** is a tree in which each node has <=2 children.
- A **binary search tree** is a binary tree in which for all nodes n, all left descendents <= n <= all right descendents.
- A **complete binary tree** has every level fully filled except for rightmost elements on last level.
- A **full binary tree** is a binary tree where each node has either 0 or 2 children.
- A **perfect binary tree** has every level fully filled.





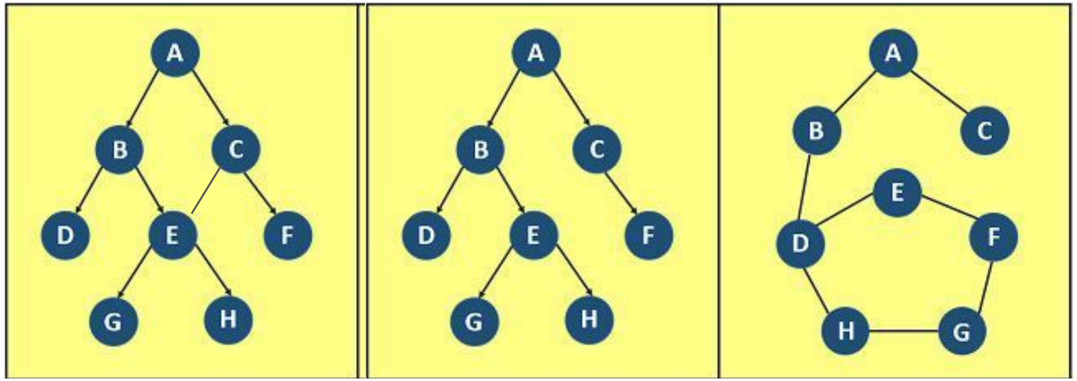


Perfect binary tree



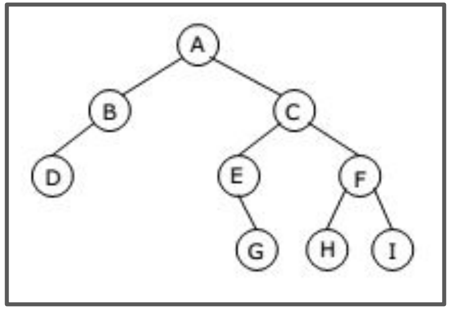
What is the difference between a graph and a tree?

- A Tree is just a restricted form of a Graph.
- Trees have direction (parent / child relationships) and don't contain cycles. They fit with in the category of Directed Acyclic Graphs (or a DAG).
- So Trees are DAGs with the restriction that a child can only have one parent.
- For the graph to be a valid tree, it must have exactly n - 1 edges. Any less, and it can't possibly be fully connected. Any more, and it has to contain cycles.
    - Additionally, if the graph is fully connected and contains exactly n - 1 edges, it can't possibly contain a cycle, and therefore must be a tree!



DAG vs Tree vs Graph

Compare/Contrast the ways to traverse a tree. How to make sure that if the tree is a BST, numbers are sorted ascending after traversal? What is the complexity of these traversals?



For graph traversals in general, complexity is O(|V| + |E|). However, since the max. number of edges in a tree is |V|-1, for trees it's O(|V|).

Depth First Searches (DFS) include **preorder**, **inorder**, and **postorder**.

```
void preOrderTraversal(TreeNode node) {
   if (node != null) {
      visit(node);
      preOrderTraversal(node.left);
      preOrderTraversal(node.right);
   }
}
```

```
void inOrderTraversal(TreeNode node) {
   if (node != null) {
      inOrderTraversal(node.left);
      visit(node);
      inOrderTraversal(node.right);
   }
}
```

```
void postOrderTraversal(TreeNode node) {
   if (node != null) {
      postOrderTraversal(node.left);
      postOrderTraversal(node.right);
      visit(node);
   }
}
```

Breadth First Search (BFS), AKA **level-order**:

```
void search(Node root) {
  Queue queue = new Queue();
  root.marked = true;
  queue.enqueue(root); // Add to the end of queue

  while (!queue.isEmpty()) {
    Node r = queue.dequeue(); // Remove from the front of the queue
    visit(r);
    foreach (Node n in r.adjacent) {
      if (n.marked == false) {
        n.marked = true;
        queue.enqueue(n);
      }
    }
  }
}
```

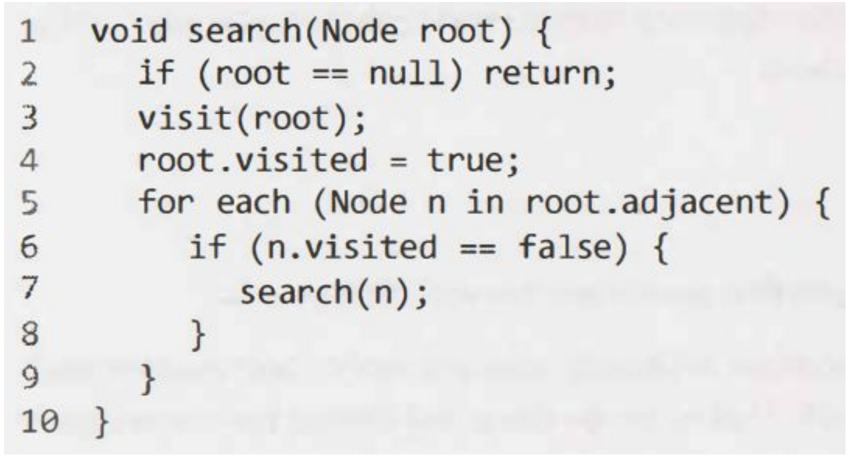
- Preorder traversal yields: A, B, D, C, E, G, F, H, I
- Inorder traversal yields: D. B. A. E. G. C. H. F. I
- Postorder traversal yields: D, B, G, E, H, I, F, C, A
- Level order traversal yields: A, B, C, D, E, F, G, H, I

IMPORTANT NOTE:
For a BST, inorder yields numbers sorted ascending

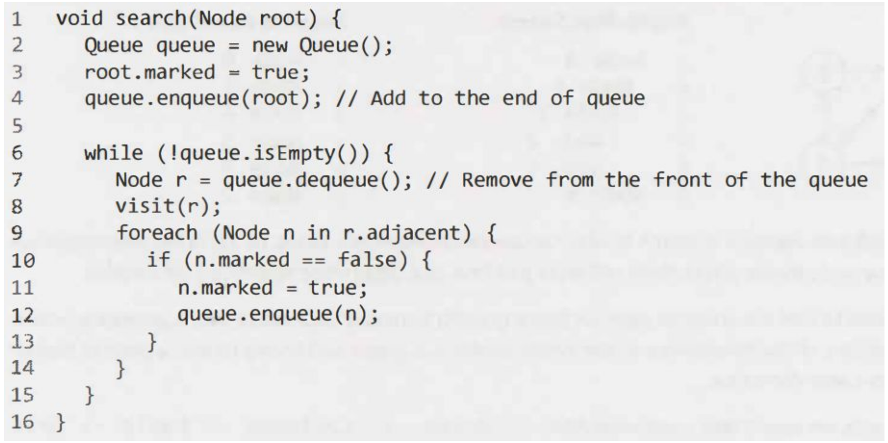
What are the ways to traverse a graph? Contrast them. How does it differ from traversing a tree?

DFS (left), BFS (right). DFS is a bit simpler, and used if you want to visit every node. However, to find short paths between nodes, BFS is generally better because you won't get stuck going very deep; you focus on the immediate neighbors. In both cases, complexity is O(|V| + |E|).

The main difference between tree and graph traversal is that for the latter, you need to mark nodes as visited, or there might be infinite loops.

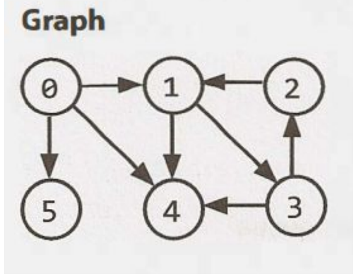


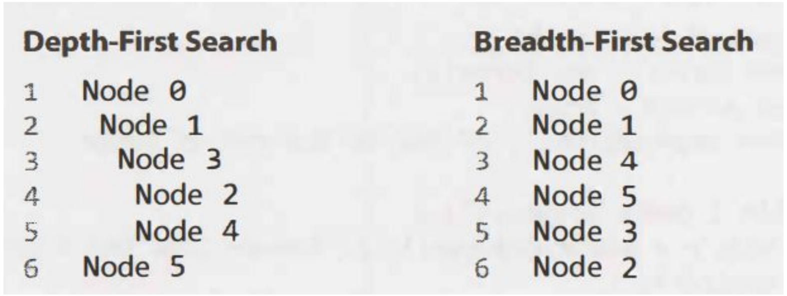
```
void search(Node root) {
   if (root == null) return;
   visit(root);
   root.visited = true;
   for each (Node n in root.adjacent) {
      if (n.visited == false) {
         search(n);
      }
   }
}
```



```
void search(Node root) {
   Queue queue = new Queue();
   root.marked = true;
   queue.enqueue(root); // Add to the end of queue

   while (!queue.isEmpty()) {
      Node r = queue.dequeue(); // Remove from the front of the queue
      visit(r);
      foreach (Node n in r.adjacent) {
         if (n.marked == false) {
            n.marked = true;
            queue.enqueue(n);
         }
      }
   }
}
```

Graph





Depth-First Search

```
Node 0
 Node 1
  Node 3
   Node 2
   Node 4
Node 5
```

Breadth-First Search

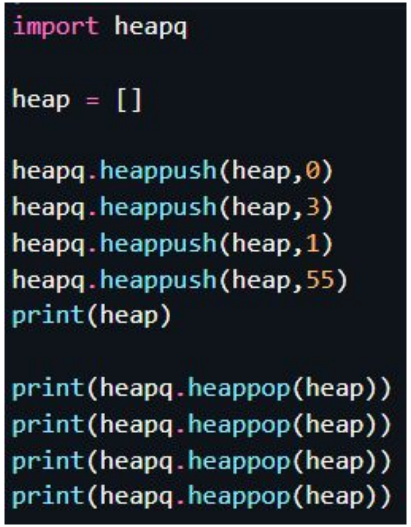
```
Node 0
Node 1
Node 4
Node 5
Node 3
Node 2
```

Define the behavior and complexity of heaps, and how to implement one in python.

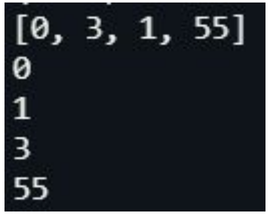
A min/max heap is a complete binary tree (so, filled except for rightmost elements on last level), where each node is smaller/larger than its children. So, the root is the smallest/largest element. The two key operations are:

- Insert, O(log n)
  - Add element to the bottom, preserving completeness. Then, "bubble up" by checking if the added element is smaller/larger than its parent.
- Extract_min/max, O(log n)
  - The min/max will always be at the top; then, we replace it with the last element in the heap (bottommost, rightmost element). Then, we bubble down the element, swapping it with its smaller/larger child as necessary.

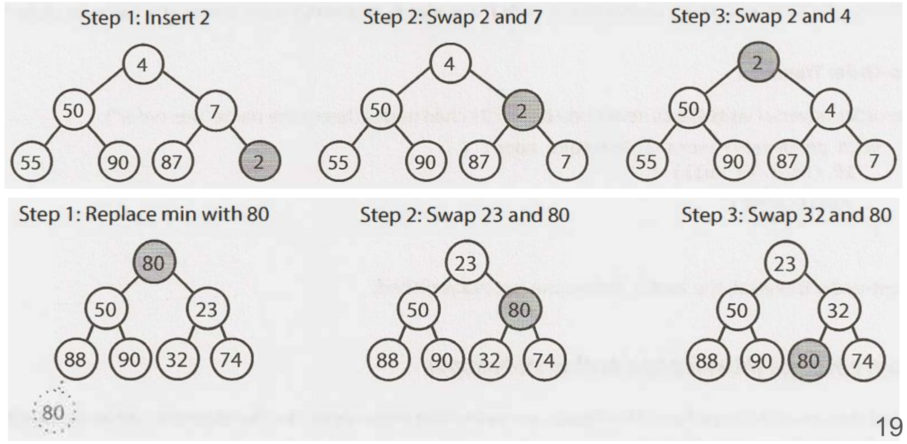
Heaps can be implemented with the heapq library, using heappush and heappop on a list. Note that it **defaults to a min heap**; for a max heap, make the values negative.



```python
import heapq

heap = []

heapq.heappush(heap,0)
heapq.heappush(heap,3)
heapq.heappush(heap,1)
heapq.heappush(heap,55)
print(heap)

print(heapq.heappop(heap))
print(heapq.heappop(heap))
print(heapq.heappop(heap))
print(heapq.heappop(heap))
```



```
[0, 3, 1, 55]
0
1
3
55
```



How do you keep track of the top-k smallest elements? How about the top-k largest?

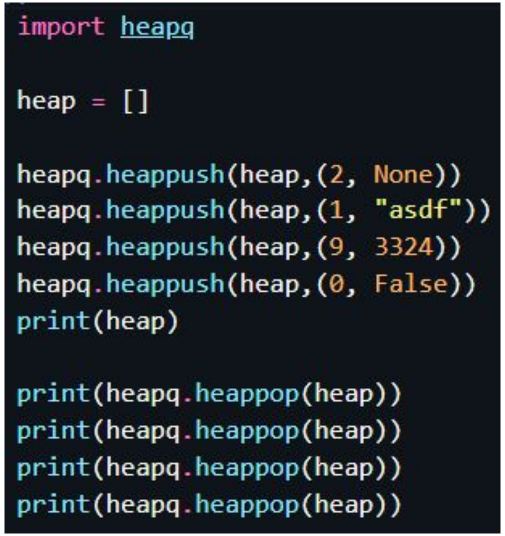
- Top k largest: use min heap to constantly remove the minimum elements, leaving only the largest
- Top k smallest: use max heap to constantly remove the largest elements, leaving only the smallest

What is a priority queue? State the complexities of implementing one with an unsorted linked list, sorted linked list, and heap.

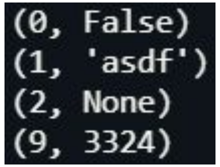
A **priority queue** takes in {key, value} pairs. The main operations are adding elements, and then removing the element with the minimum key. Three ways to implement one is as follows:

- Unsorted linked list. Here, adding elements takes O(1) but finding/removing the min takes O(n)
- Sorted linked list. Here, adding elements takes O(n) but finding/removing the min is trivial O(1)
- Min heap. Here, adding and finding/removing the min takes O(log n) on average; worst case is O(n).

An example of using heapq to implement a priority queue:



```python
import heapq

heap = []

heapq.heappush(heap,(2, None))
heapq.heappush(heap,(1, "asdf"))
heapq.heappush(heap,(9, 3324))
heapq.heappush(heap,(0, False))
print(heap)

print(heapq.heappop(heap))
print(heapq.heappop(heap))
print(heapq.heappop(heap))
print(heapq.heappop(heap))
```



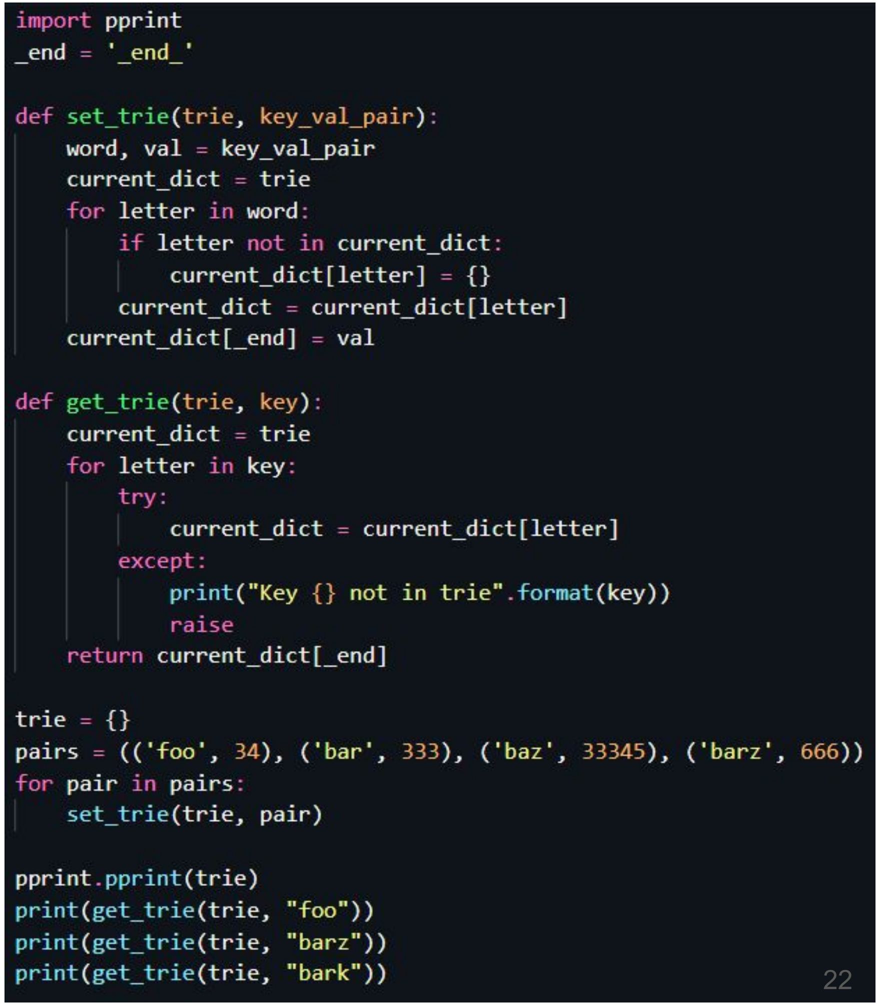
```
(0, False)
(1, 'asdf')
(2, None)
(9, 3324)
```

Note that this is possible since:

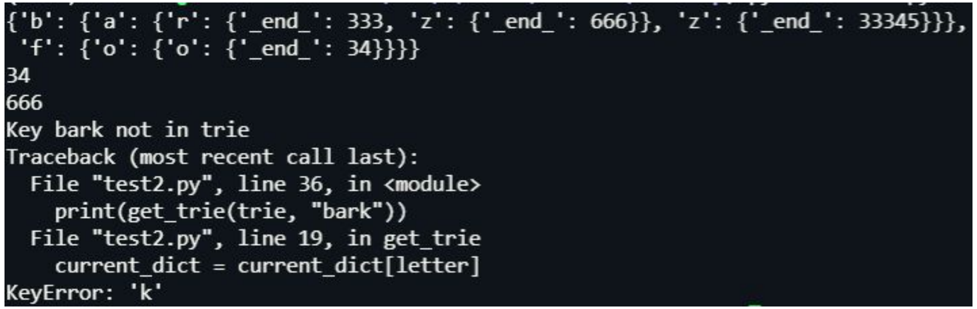
1. *python tuples evaluate “less than” one element at a time, from left to right in the tuple*
2. *python uses lazy evaluation*

What is a trie/prefix tree? What is its complexity, compare it to hash tables, and implement one.

- A **trie** is an n-ary tree where characters are stored at each node; each path down the tree may represent a word. **Null nodes** (*) are often used to indicate complete words.
- Commonly, a trie is used to store the entire english language, for quick prefix lookups.
- It can also be used to replace hash tables.\
  - Advantages of tries over hash tables:
    - Inserting/deleting is faster in the worst case, taking O(m) where m is the length of a search string. In hash tables, worst case is O(N)
    - There are no collisions of different keys in a trie
    - There is no need to “rehash” to a larger allocation as more keys are added
    - Unlike hash tables, prefixes can be easily found
  - Disadvantages of tries over hash tables:
    - On average, hash tables are usually O(1) with O(m) spent evaluating the hash. On average, hash table is faster since there is less time needed to access memory.
    - Some keys (eg numbers) can lead to long chains which are not particularly meaningful



```python
import pprint
_end = '_end_'

def set_trie(trie, key_val_pair):
    word, val = key_val_pair
    current_dict = trie
    for letter in word:
        if letter not in current_dict:
            current_dict[letter] = {}
        current_dict = current_dict[letter]
    current_dict[_end] = val

def get_trie(trie, key):
    current_dict = trie
    for letter in key:
        try:
            current_dict = current_dict[letter]
        except:
            print("Key {} not in trie".format(key))
            raise
    return current_dict[_end]

trie = {}
pairs = (('foo', 34), ('bar', 333), ('baz', 33345), ('barz', 666))
for pair in pairs:
    set_trie(trie, pair)

pprint.pprint(trie)
print(get_trie(trie, "foo"))
print(get_trie(trie, "barz"))
print(get_trie(trie, "bark"))
```



```
{'b': {'a': {'r': {'_end_': 333, 'z': {'_end_': 666}}, 'z': {'_end_': 33345}}},
 'f': {'o': {'o': {'_end_': 34}}}}
34
666
Key bark not in trie
Traceback (most recent call last):
  File "test2.py", line 36, in <module>
    print(get_trie(trie, "bark"))
  File "test2.py", line 19, in get_trie
    current_dict = current_dict[letter]
KeyError: 'k'
```

What is a python set? How is it different from a frozenset?

Sets are unordered, have unique elements, and mutable (but their elements within must be immutable). Some common methods are:

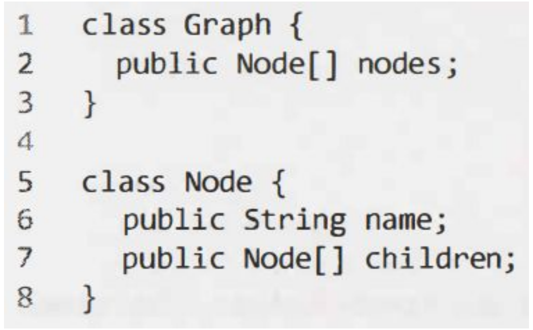
- **Creation**
  - X = set([1,2,3])
  - X = {1,2,3}
  - X = set()
- **Modifying**
  - X.add(4)
  - X.remove(4)
- **Union of sets X1, X2**: X1 | X2
- **Intersection of sets X1, X2**: X1 & X2
- **Difference of sets X1, X2**: X1 - X2
- **XOR of sets X1, X2**: X1 ^ X2
- **X1 is a subset of X2**: X1 <= X2
  - Similar with superset
  - To enforce strict subset, drop the =

A **frozenset** is similar to a set, but is immutable. So, a set can't contain sets, but can contain frozensets.

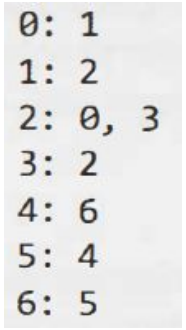
- To create, use x = frozenset([1,2,3]). The argument must be an iterable, so frozenset(1) or frozenset(1,2) won't work.
- All the set operations still work, eg frozenset([1,2]) | frozenset([2,3]) = frozenset([1,2,3]). However, because they're immutable, you can't add or remove.

What are 3 ways to describe a graph?

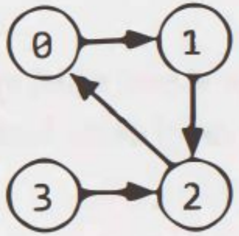
- Directly in the node class (usually, this is most efficient)
- As an adjacency list
- As an adjacency matrix, where a true value at matrix[i][j] represents an edge from node i to node j (ie, column node -> row node).

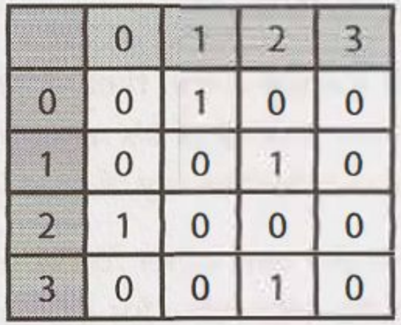


```
class Graph {
  public Node[] nodes;
}

class Node {
   public String name;
   public Node[] children;
}
```



```
0: 1
1: 2
2: 0, 3
3: 2
4: 6
5: 4
6: 5
```



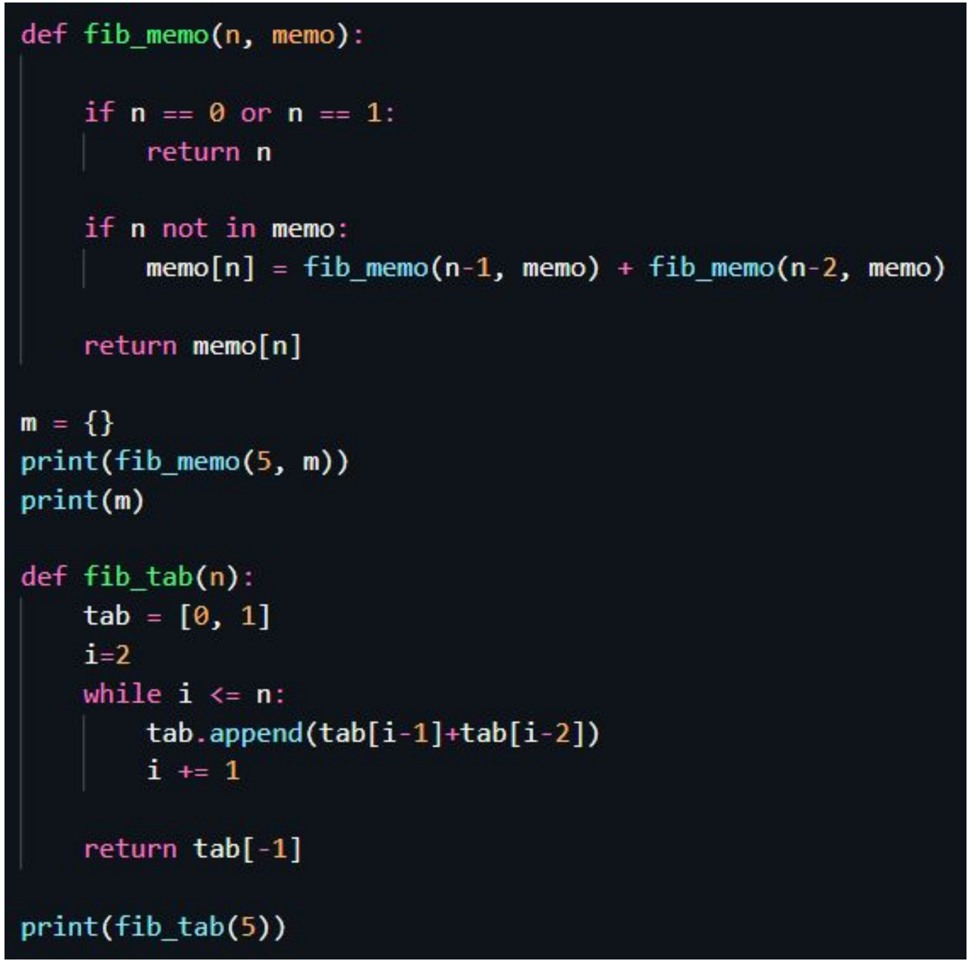


	0	1	2	3
0	0	1	0	0
1	0	0	1	0
2	1	0	0	0
3	0	0	1	0

Define the following terms: Optimal Substructure, Dynamic Programming, Bottom-up/Top-down recursion, Tabulation, and Memoization. How do you calculate the runtime for recursive algorithms?

- A problem is said to have **optimal substructure** if an optimal solution can be constructed from optimal solutions of its subproblems
- **Dynamic programming:** A way to optimize problems of a recursive nature, by caching the results of subproblems that might be encountered later on.
- **Bottom-Up Recursion:** Here, you start with the sample case, then build on that. For example, solving for one element, then two, then three, etc. You kind of "build up to the final solution from scratch"
  - For bottom-up, DP can be often done by using **tabulation**. Here, a table's values can be populated/cached sequentially.
  - When figuring out time complexity for recursion, multiply number of recursive calls with the time it takes for each result/call.
- **Top-Down Recursion**: Here, you start with the big solution, by trying to divide the problem into subproblems.
  - For top-down, DP can be often done by using **memoization**. The data might not be stored/cached sequentially, but as a byproduct of the recursive calls which can be used for later recursive calls.

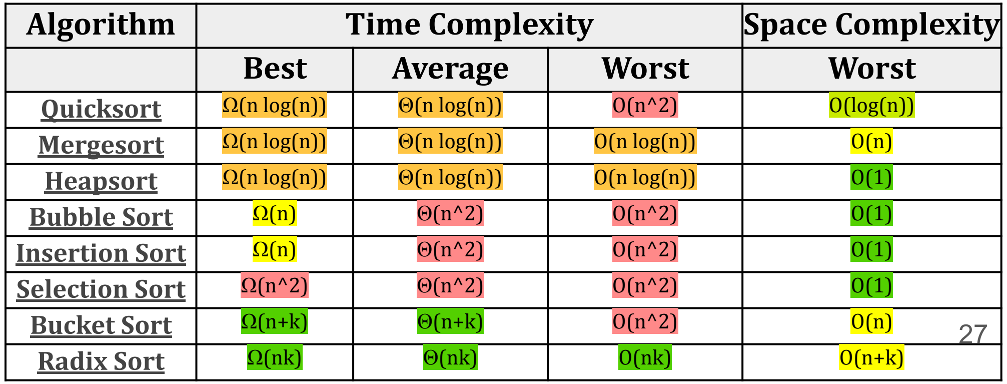
Implement fib(n) using 2 flavors of dynamic programming: memoization and tabulation..



```python
def fib_memo(n, memo):

    if n == 0 or n == 1:
        return n

    if n not in memo:
        memo[n] = fib_memo(n-1, memo) + fib_memo(n-2, memo)

    return memo[n]

m = {}
print(fib_memo(5, m))
print(m)

def fib_tab(n):
    tab = [0, 1]
    i=2
    while i <= n:
        tab.append(tab[i-1]+tab[i-2])
        i += 1

    return tab[-1]

print(fib_tab(5))
```

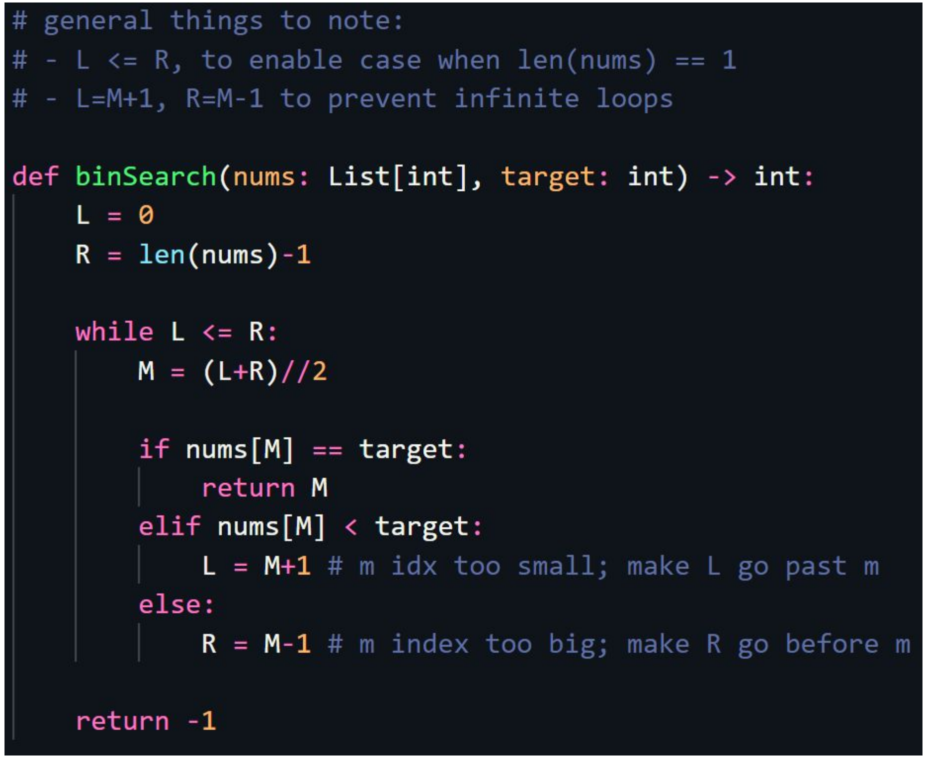
Describe & state complexity: bubble sort, insertion sort, selection sort, quick sort, merge sort, heap sort, bucket sort, radix sort.

- **Quick sort:** Centers around a partition subroutine, such that elements to the left of a chosen partition are less than it, and elements to its right are greater (ie, the **partition is in its final position**). This can be implemented efficiently using swapping, with a “low” pointer and “high” pointer. The partition step is recursively applied to the sublist to the left and right of the partition. Choosing a bad partition can lead to O(n^2) worst case.
- **Merge sort**: Repeatedly divides the array in half, sorts those halves, and then merges them back together. Eventually you just merge two single element arrays. The merge step does all the heavy lifting, and can be implemented using pointers for each sorted half.
- **Heap sort:** Place all elements in a min heap, and then pop everything. Recall that min heaps are complete binary trees where parents are always smaller than its children. Elements can be added/removed by bubbling down/up.
- **Bubble sort:** Start at beginning of array and traverse through each consecutive pair, swapping them if the first is greater than the second. Array needs to be traversed n times.
- **Insertion sort:** Here, we keep the head of the list sorted, and repeatedly consider the next element in the tail, finding the appropriate place to insert in the head. At each iteration, the tail shrinks and the head grows until sorted.
- **Selection sort**: Simply repeatedly find the smallest element in the “unsorted tail”, using a linear scan, and swap it to the “sorted head”.
- **Bucket sort:** Separate the array into k smaller buckets, sort them individually using a sorting algorithm or recursive call to itself, and then combine the result. Unlike merge sort, the combine step is trivial because the buckets are already sorted.
- **Radix sort:** Mostly works for **discretizable** data, like integers. At each iteration, we focus on ith digit position and place each number into one of 10 buckets, based on that digit. Then, we combine the buckets in order so that we have **sorted the numbers by the ith digit**. Each iteration takes only a single pass; in total, the sort takes O(k*n), assuming the largest number has k digits.

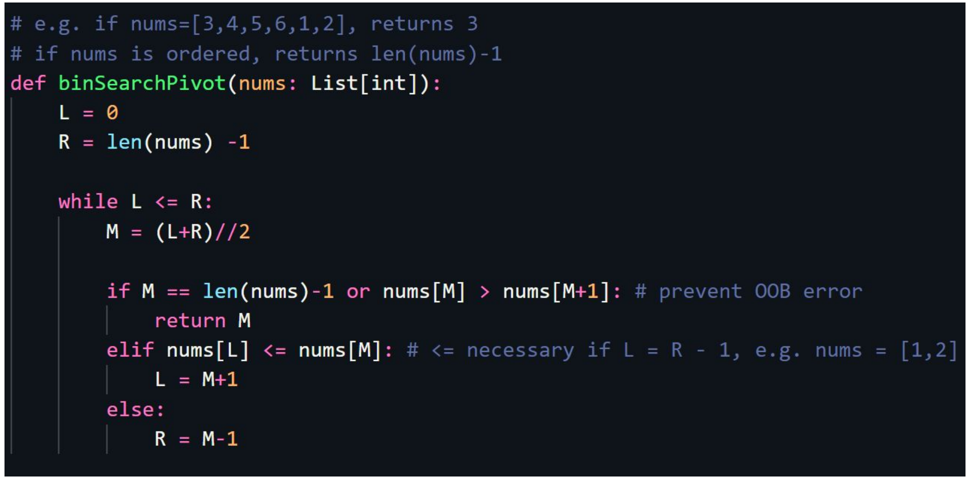


Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Quicksort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(log(n))
Mergesort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(n)
Heapsort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(1)
Bubble Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Insertion Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Selection Sort	Ω(n^2)	Θ(n^2)	O(n^2)	O(1)
Bucket Sort	Ω(n+k)	Θ(n+k)	O(n^2)	O(n)
Radix Sort	Ω(nk)	Θ(nk)	O(nk)	O(n+k)

Implement binary search, and binary search for a pivot.



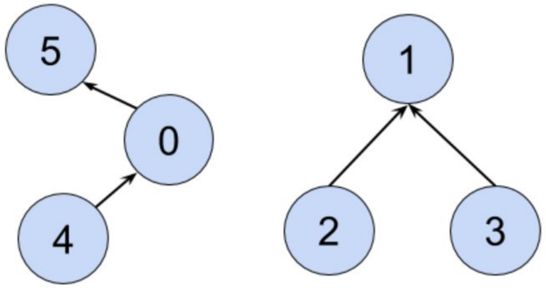
```python
# general things to note:
# - L <= R, to enable case when len(nums) == 1
# - L=M+1, R=M-1 to prevent infinite loops

def binSearch(nums: List[int], target: int) -> int:
    L = 0
    R = len(nums)-1

    while L <= R:
        M = (L+R)//2

        if nums[M] == target:
            return M
        elif nums[M] < target:
            L = M+1 # m idx too small; make L go past m
        else:
            R = M-1 # m index too big; make R go before m

    return -1
```



```python
# e.g. if nums=[3,4,5,6,1,2], returns 3
# if nums is ordered, returns len(nums)-1
def binSearchPivot(nums: List[int]):
    L = 0
    R = len(nums) -1

    while L <= R:
        M = (L+R)//2

        if M == len(nums)-1 or nums[M] > nums[M+1]: # prevent OOB error
            return M
        elif nums[L] <= nums[M]: # <= necessary if L = R - 1, e.g. nums = [1,2]
            L = M+1
        else:
            R = M-1
```

Describe What Union-Find is and how to implement it. What are some applications?

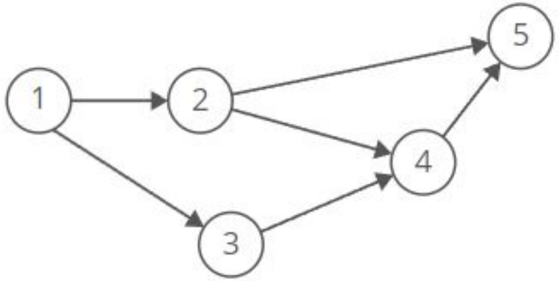
- Union-Find is a data structure that stores a collection of disjoint sets. It represents each of the sets as a directed tree, with the edges pointing towards the root node.
  - For example, {{0,4,5}, {1,2,3}} can be represented as below, or alternatively, [5,1,1,1,0,5], where each index node has the value of its parent.
- The two operations are:
  - find(A), which returns the top parent of A. An optimization is to reset all nodes traversed to the top parent.
  - union(A,B), which finds the top parents of A and B, then sets B's parent to be A (or vice versa).
- In practice, find and union are effectively O(1), though technically there are some other details.
- Applications:
  - Keeping track of the number of connected components & their contents of a graph (initialize empty set for each vertex, then repeatedly merge sets based on the edges with union)
  - Determining if two vertices belong to the same connected component
  - Checking for cycles
    - If an edge's merge did not result in any change, then there's a redundant path which is a loop (assuming there aren't any repeated edges or self loops)



```
class UnionFind:
   def __init__(self, n):
      self.nodes = [i for i in range(n)]

   def find(self, a):
      to_remap = []
      while self.nodes[a]!= a:
         to_remap.append(a)
         a = self.nodes[a]
      # optimization to remap, making things faster next time
      for n in to_remap:
         self.nodes[n] = a
      return a

   # returns true if a union happened, else returns false
   def union(self, a, b):
      a_parent = self.find(a)
      b_parent = self.find(b)
      if a_parent == b_parent:
         return False
      else:
         self.nodes[b_parent] = a_parent
         return True
```

What is a topological sort, and what are some applications?

- The topological sort algorithm takes a directed graph and returns an array of the nodes where each node appears before all the nodes it points to.
- There can be more than one; for example, in the graph below, [1, 2, 3, 4, 5] and [1, 3, 2, 4, 5] are both valid
- One easy way to generate a topological sort is to repeatedly pick the nodes with an indegree of 0.
- The topological sort is a way to find an ordering that resolves/respects dependencies in scheduling.
    - For example, compilation in makefiles



How do you find the shortest path between two nodes in an unweighted graph?

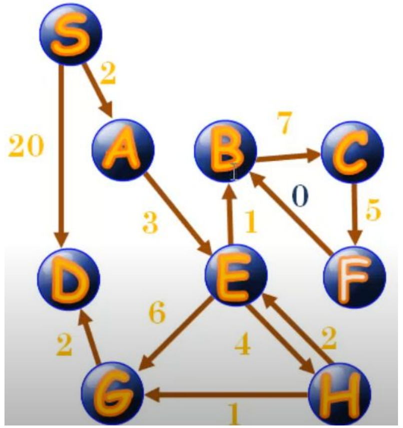
This can be done using BFS.

Imagine pouring water on the source vertex, and imagine all the edges are tubes that can spread the water to neighboring vertices in one unit of time. BFS comes down to simulating this process and asking at what time the water reaches each vertex: that time is the distance of the vertex from the source. (Dijkstra's algorithm is asking the same question when the tubes don't necessarily take unit time to spread the water.)

What is Dijkstra’s algorithm and the Bellman-Ford algorithm? How is Dijkstra’s implemented, and what’s its time/space complexity? Perform it for the following graph.

- **Dijkstra’s Algorithm**: Finds the shortest path between a start node to every node in a weighted directed graph, which may have cycles. All edges must have positive values. Time is O(|E| log|V|), space is O(|V|+|E|). Note that if the graph is unweighted, BFS=dijkstras.
- **Bellman-Ford Algorithm**: Finds the shortest paths from a single node in a weighted directed graph with positive and negative edges.

```python
# assumes nodes are represented by 0-indexed integers; n is number of nodes
# edges is a list of [source node num, target node num, weight]
def dijkstras_algorithm(edges: List[List[int]], n: int, start_node: int) -> int:
    # setup node distances, previous links, and visited/finalized set
    dist = [float("inf")]*n
    dist[start_node] = 0
    prev = [None]*n
    visited = set()
    # min_heap holds [weight, source_node]
    min_heap = [[0, start_node]]
    # convert edges into an adjacency list of form {source node: [[target node 1, weight 1],...]}
    adj_list = defaultdict(lambda:[])
    for a, b, weight in edges:
        adj_list[a].append([b, weight])

    # note: if you want to just search for a specific node, loop can be broken once that's found.
    while len(min_heap) > 0:
        curr_min_dist, curr_node = heapq.heappop(min_heap)
        visited.add(curr_node) # this node will no longer be touched; true min distance
        for neighbor, edge_weight in adj_list.get(curr_node, []):
            if neighbor not in visited and curr_min_dist+edge_weight < dist[neighbor]:
                dist[neighbor] = curr_min_dist+edge_weight
                prev[neighbor] = curr_node
                heapq.heappush(min_heap, [curr_min_dist+edge_weight, neighbor])

    return dist, prev
```

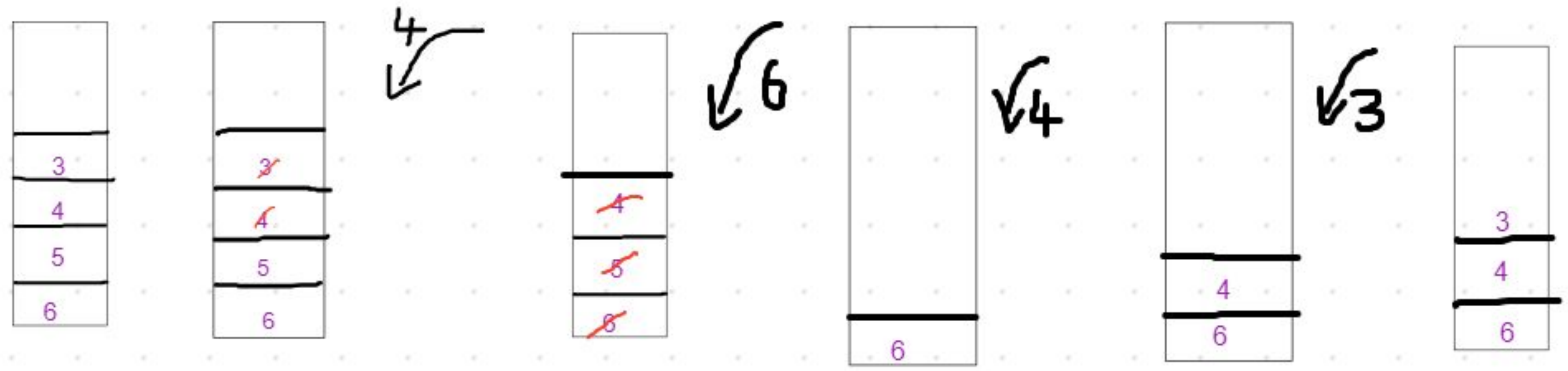


	S	A	B	C	D	E	F	G	H
d	0	2	6	13	12	5	18	10	9
π	-	S	E	B	G	A	C	H	E

What is a monotonic stack and what can it be used for?

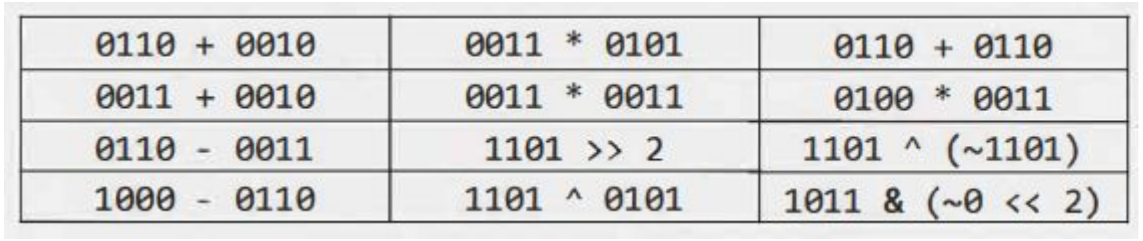
A monotonic stack is a stack whose elements are monotonically increasing or decreasing.

It’s useful for previous/next less/greater element.

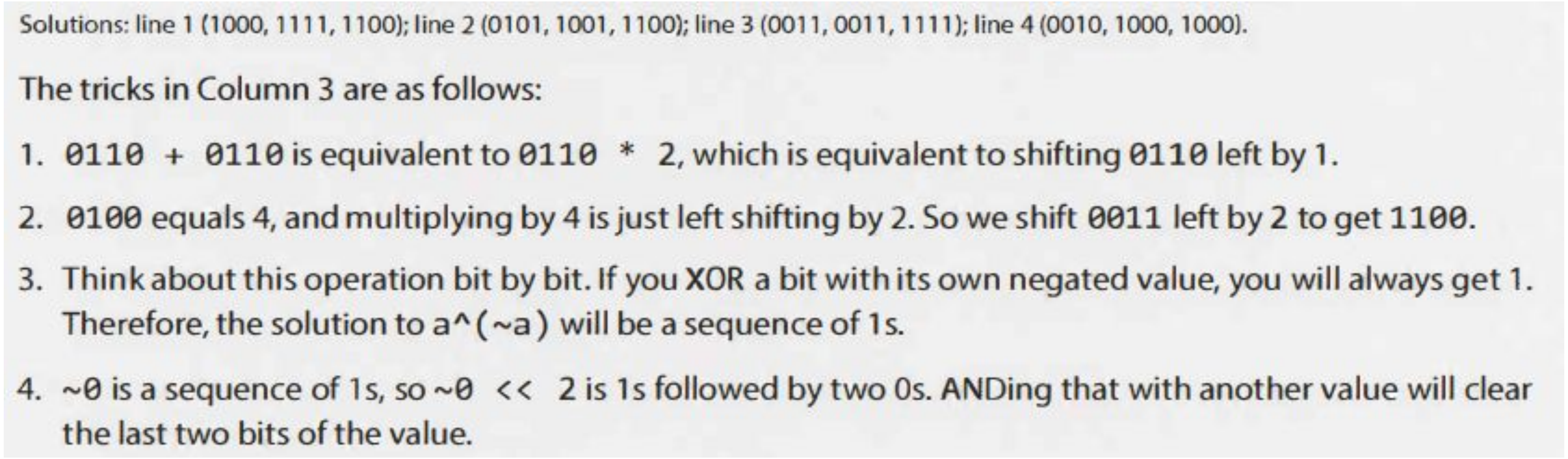


Bit Manipulations

Calculate the following, describing any possible tricks/shortcuts.



`0110 + 0010`	`0011 * 0101`	`0110 + 0110`
`0011 + 0010`	`0011 * 0011`	`0100 * 0011`
`0110 - 0011`	`1101 >> 2`	`1101 ^ (~1101)`
`1000 - 0110`	`1101 ^ 0101`	`1011 & (~0 << 2)`



Solutions: line 1 (1000, 1111, 1100); line 2 (0101, 1001, 1100); line 3 (0011, 0011, 1111); line 4 (0010, 1000, 1000).

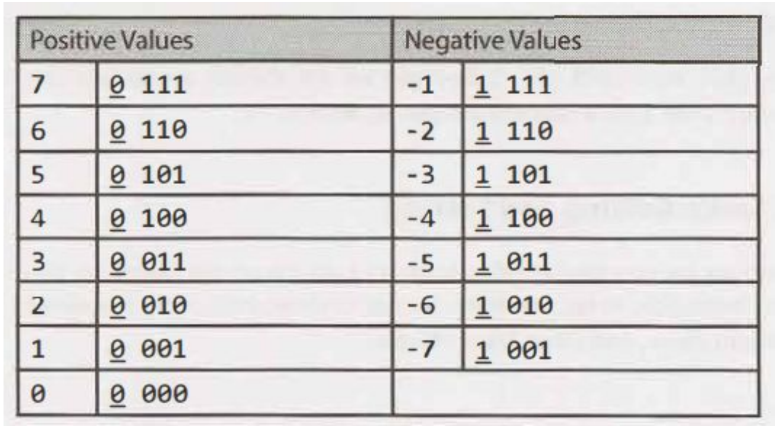
The tricks in Column 3 are as follows:

1. `0110 + 0110` is equivalent to `0110 * 2`, which is equivalent to shifting `0110` left by 1.
2. `0100` equals 4, and multiplying by 4 is just left shifting by 2. So we shift `0011` left by 2 to get `1100`.
3. Think about this operation bit by bit. If you XOR a bit with its own negated value, you will always get 1. Therefore, the solution to `a^(~a)` will be a sequence of 1s.
4. `~0` is a sequence of 1s, so `~0 << 2` is 1s followed by two 0s. ANDing that with another value will clear the last two bits of the value.

```
x ^ 0s =                 x & 0s =                 x | 0s =
x ^ 1s =                 x & 1s =                x | 1s =
 x ^ x =                  x & x =                  x | x =
```

```
x ^ 0s = x               x & 0s = 0               x | 0s = x
x ^ 1s = ~x              x & 1s = x              x | 1s = 1s
 x ^ x = 0                x & x = x                x | x = x
```

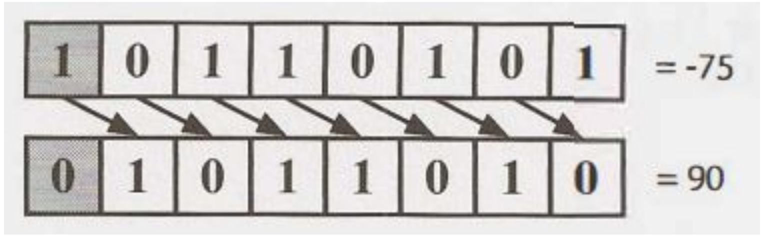
Explain two's complement integers, and represent -7 to 7 in bits. How do you convert between pos and neg? What is the difference between logical and arithmetic shifts?

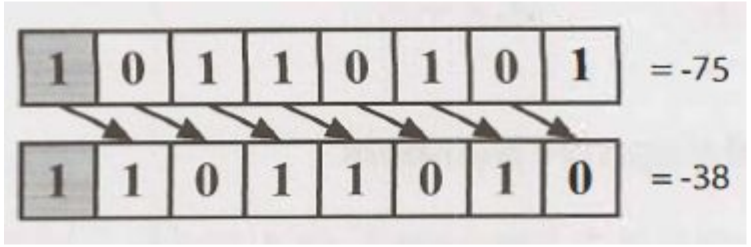


Positive Values		Negative Values	
7	0 111	-1	1 111
6	0 110	-2	1 110
5	0 101	-3	1 101
4	0 100	-4	1 100
3	0 011	-5	1 011
2	0 010	-6	1 010
1	0 001	-7	1 001
0	0 000		

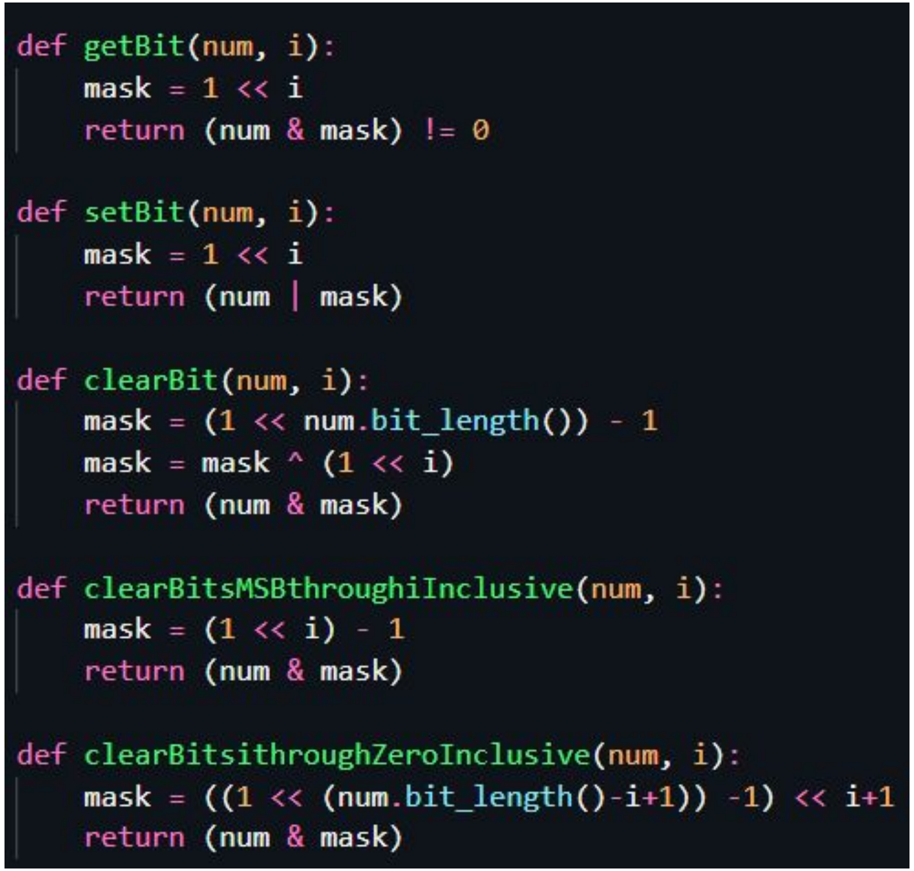
$$\begin{tikzcd}\textbf{Pos} \arrow[r, shift left, "\textit{flip, +1}"] & \textbf{Neg} \arrow[l, shift left, "\textit{-1, flip}"]\end{tikzcd}$$

In logical right shift (>>>, left), you shift bits and put 0 in the most significant bits. In arithmetic right shift (>>, right), you shift to the right but fill the new bits with the value of the sign bit. Note that left logical (<<<) and arithmetic shifts (<<) are the same.





Implement getBit(num, i), setBit(num, i), clearBit(num,i)


```
def getBit(num, i):
    mask = 1 << i
    return (num & mask) != 0

def setBit(num, i):
    mask = 1 << i
    return (num | mask)

def clearBit(num, i):
    mask = (1 << num.bit_length()) - 1
    mask = mask ^ (1 << i)
    return (num & mask)

def clearBitsMSBthroughiInclusive(num, i):
    mask = (1 << i) - 1
    return (num & mask)

def clearBitsithroughZeroInclusive(num, i):
    mask = ((1 << (num.bit_length()-i+1)) -1) << i+1
    return (num & mask)
```

Time/Space Complexity

What is Big O/Theta/Omega? How is it useful and what are its shortcomings?

This is a mathematical notation that describes the limiting behavior of a function.

- **Big O** describes an upper bound on the runtime.
- **Big Omega (Ω)** describes a lower bound
- **Big Theta (Θ)** describes a tight bound on the run time

Formally,

$$f(x) \in O(g(x)) \text{ iff } \exists M, x_0 \text{s.t. } f(x) < Mg(x) \forall x \geq x_0$$

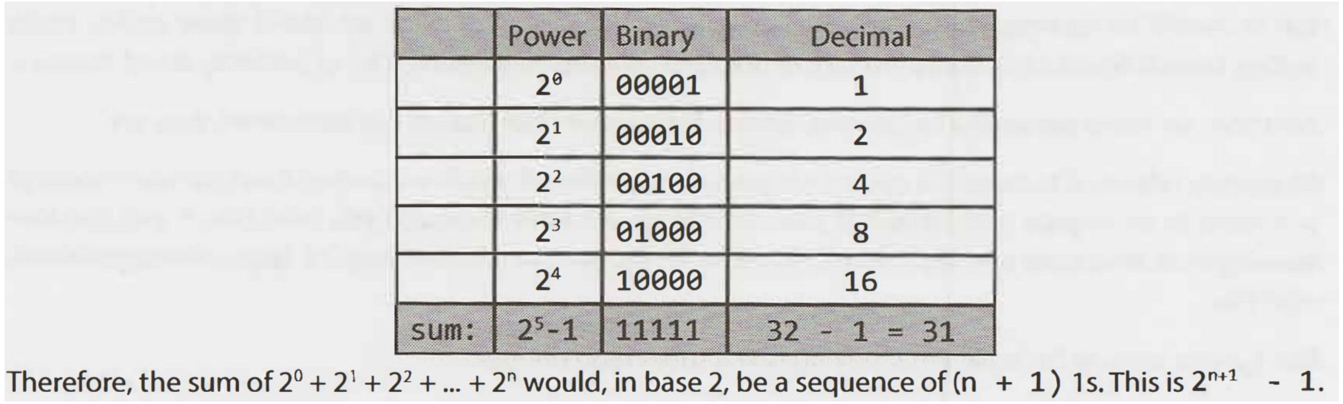
Some notes:

- In industry, people usually mean “big theta” when they say “big O”.
- Big O allows us to express how runtime scales.
- However, it is not sufficient to know the true runtime of a function, since the constant factor M or value of x_0 can be nontrivial. There are also factors, such as cache/compiler optimization, which are not captured by Big O.

What is the big-O for the following series:
1 + 2 + 3 + … + n = ?
1 + 2 + 4 + 8 + … + n = ?
2^0 + 2^1 + 2^2 + … + 2^n = ?

1+2+...+n = (n(n+1))/2 = O(n^2)

1 + 2 + 4 + 8 + … + n = O(2n) = O(n)



	Power	Binary	Decimal
	2^0	00001	1
	2^1	00010	2
	2^2	00100	4
	2^3	01000	8
	2^4	10000	16
sum:	2^5-1	11111	32 - 1 = 31

Therefore, the sum of $2^0 + 2^1 + 2^2 + ... + 2^n$ would, in base 2, be a sequence of (n + 1) 1s. This is 2^{n+1} - 1.

What is the difference between combinations and permutations, and how do you compute them? What's the intuition behind the equation?

Permutations: Order matters

- Represents multiplying n*(n-1)*...*(n-k-1)
- If n=k, becomes n!

$$P_k^n = \frac{n!}{(n-k)!}$$

Combinations: Order invariant

- Same as permutation, but divide out the duplicates with same elements but different ordering
- If n=k, becomes 1

$$C_k^n = \frac{n!}{k!(n-k)!}$$

What is the time complexity of nCk?

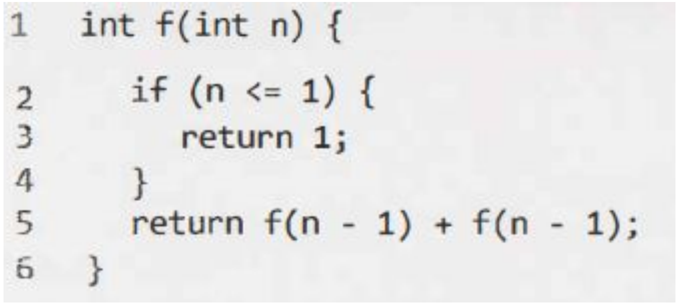
- In general, O(n^min{k,n-k})
- In most cases, k is very small, so O(n^k)

How does one calculate the amortized time of adding to an array?

Amortized time describes the average time it takes to perform an action; it’s useful if periodically, things take longer/shorter than usual.

For an array, after inserting N elements, 1 + 2 + 4 + 8 + … + N ≈ 2N copies will have been needed. Thus, N insertions can be performed in O(N) time; a single insertion takes O(1).

What’s the space and time complexity of this code?



```
int f(int n) {
    if (n <= 1) {
        return 1;
    }
    return f(n - 1) + f(n - 1);
}
```

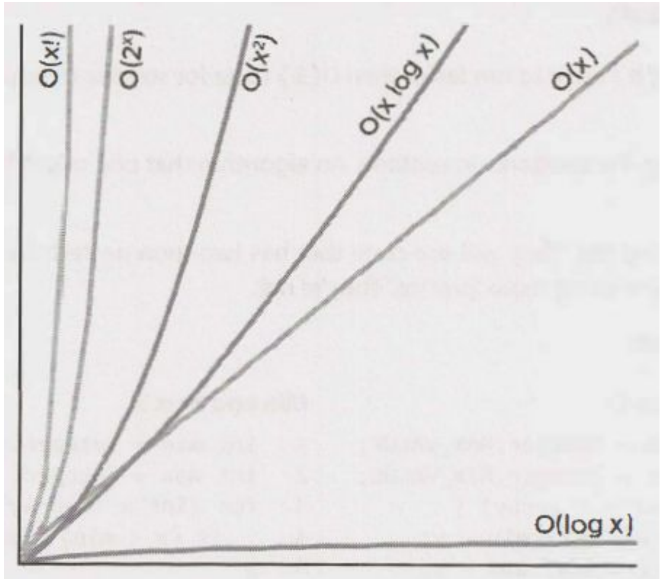
There are 1+2^1+2^2+...+2^n = 2^(n+1)-1 = O(2^n) recursive calls, each taking O(1) to complete. Thus, the time complexity is O(2^n).

At each point in time, the stack will only contain at most n function call frames; each frame only takes O(1) space. Thus, the space complexity is O(n).

How many ways can you partition a string of length N?

- 2^N (partitioning the ith str with the i+1th string is a binary decision)

Rank the rate of increase for common big O times.



What is the time complexity of hashing a string?

O(n)

How do you determine runtime for recursive+memoization problems?

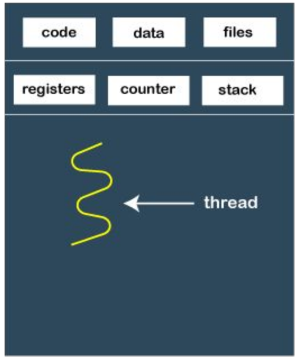
Since using the memo means you don’t calculate any state twice, the runtime is how many different combinations you can create from the “state variables”, times the time it takes for each combination individually to calculate.

Misc

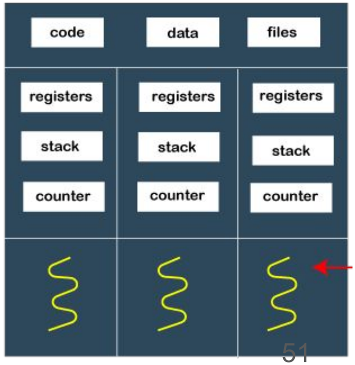
What are processes and threads?

- Both processes and threads are independent sequences of execution. Each process must start with least one thread, but can also later create multiple.
- Processes/threads are implemented on the OS level, and uses the underlying CPU's cores/processors.
    - On a multicore system, multiple threads can be executed in parallel, while on a uniprocessor system, scheduling/context switching is used for an illusion of concurrency.
- In Python, the Global Interpreter Lock (GIL) allows only one thread to hold control of the Python interpreter.
    - So, only one thread can be executing at any point in time.

Processes	Threads
Each has a separate memory space	Threads of a process run in a shared memory space
Harder to share objects between processes	Easier to share objects in the same memory, but need to be careful to avoid race conditions
Code is usually straightforward	Code usually harder to understand and get right
Larger memory footprint	Lightweight, low memory footprint
In python, takes advantages of multiple CPUs/cores	In python, threads cannot be run in parallel using multiple CPUs/cores, due to the GIL.
Use case in python: For when you want to really do more than one thing at a given time on the CPU.	**Use case in python:** enables applications to be responsive, when execution is I/O bound (eg from internet, database retrieval, etc). Things aren't being done in parallel on the CPU, but concurrently due to context switching.



Single-threaded process



Multi-threaded process

Explain what a race condition is, and how locks/semaphores can help. What’s a deadlock? Give examples.

A **race condition** occurs when a system’s final behavior is depending on the sequence/timing of other uncontrollable events, which can lead to bugs/undesirable, nondeterministic results.

For example, suppose two threads each increment the value of a global integer by 1. Ideally, left would happen, but right could potentially occur:

Thread 1	Thread 2		Integer value
			0
read value		←	0
increase value			0
write back		→	1
	read value	←	1
	increase value		1
	write back	→	2

Thread 1	Thread 2		Integer value
			0
read value		←	0
	read value	←	0
increase value			0
	increase value		0
write back		→	1
	write back	→	1

A **lock** (ie, **mutex**) can help with synchronization by enforcing limits on access to a resource when there are many threads/processes of execution. It’s like a single key that must be first acquired, used, and when done, passed on to the next thread/process.

```
# obtain lock for x
if x == 5: # the "check"
    y = x*2 # the "act"
    # using locks, no other thread can change the value of x
    # when a thread is in this if-statement "critical section"
    # y is guaranteed to be equal to 10.
# release lock for x
```

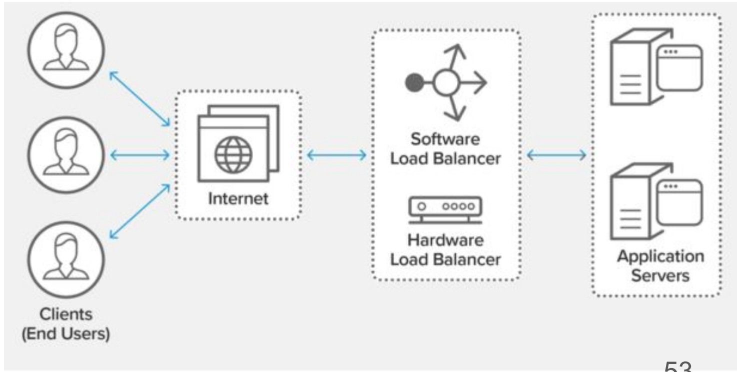
A **deadlock** can occur if a thread is waiting for an object lock that another thread holds, and that second thread is waiting for an object lock that the first thread holds. For example, if you have the following code for bank transfer, calling transfer(a,b) and transfer(b,a) will cause a deadlock where the second lock cannot be acquired.

```
void transfer(Account from, Account to, double amount){
     sync(from);
     sync(to);
     from.withdraw(amount);
     to.deposit(amount);
     release(to);
     release(from);
 }
```

A **semaphore** is a generalization that allows x number of threads to enter; for example, this can be used to limit the number of CPU, IO, or RAM intensive tasks running at the same time.

Define the following scalability concepts: Horizontal/Vertical scaling, load balancing, Database sharding/partitioning, caching.

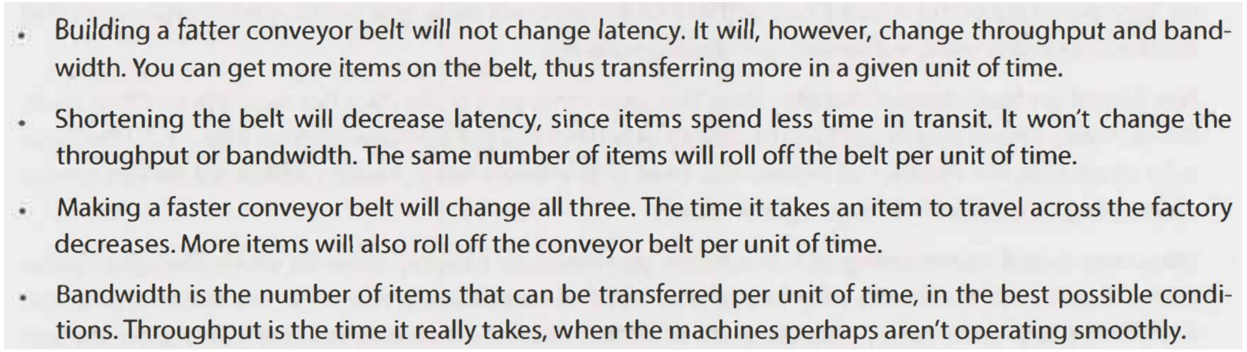
- **Vertical scaling** improves the hardware on a single machine/node. This is easier to do, but is limited.
- **Horizontal scaling** adds additional machines/node. This is more complicated and can require more overhead, but is more scalable.
- **Load balancing** is often done for frontend. This ensures that no one server is overworked, and if a server goes down, the other remaining servers can compensate.
  - There are different load balancing algorithms, such as **round robin, random, least connections, or IP-based hash**.
- **Database sharding/partitioning**: Split up the data onto multiple machines. There are several techniques, which can be combined:
  - **Vertical Partitioning**: Partitioning by feature (eg one data table for profiles, one for messages, one for videos). However, repartition may be needed in the future.
  - **Directory-based partitioning:** You maintain a lookup table. This makes it easy to add additional servers, but can add a point of failure and add overhead.
- **Caching**: Here, when an application requests data, it first tries the cache, and if it is not there, then it will look up the data.



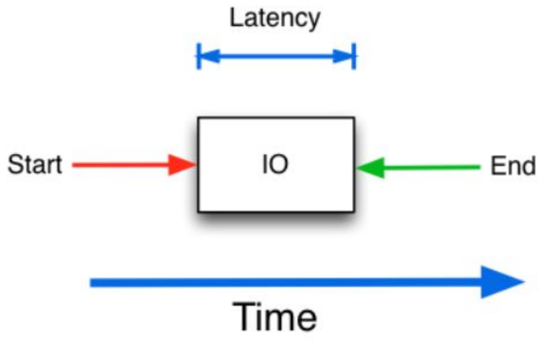
load balancing diagram

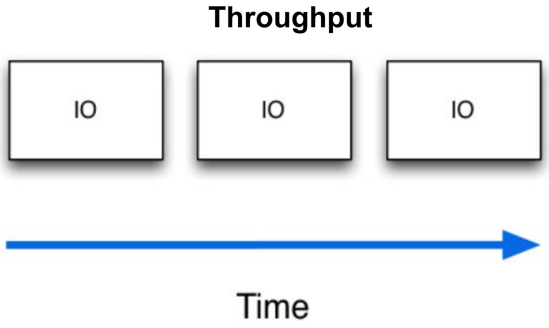
What’s the difference between bandwidth, throughput, and latency? Compare them in a conveyor belt context; how do they change as the belt is faster/slower, longer/shorter?

- **Bandwidth**: Maximum amount of data that can be transferred in a unit of time. For example, Mb/s.
- **Throughput**: Actual amount of data transferred per unit of time. While bandwidth is the upper bound, throughput is the actual rate.
- **Latency**: how long it takes for data to go from one end to the other, e.g. measured in seconds

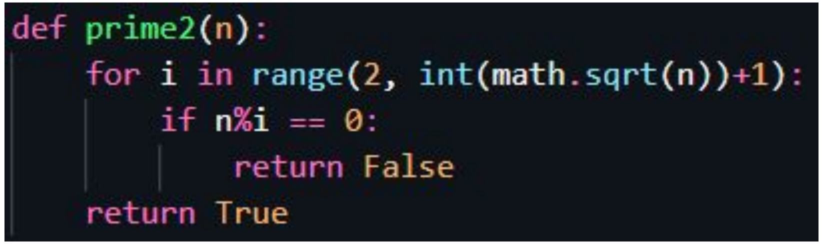


- Building a fatter conveyor belt will not change latency. It will, however, change throughput and bandwidth. You can get more items on the belt, thus transferring more in a given unit of time.
- Shortening the belt will decrease latency, since items spend less time in transit. It won’t change the throughput or bandwidth. The same number of items will roll off the belt per unit of time.
- Making a faster conveyor belt will change all three. The time it takes an item to travel across the factory decreases. More items will also roll off the conveyor belt per unit of time.
- Bandwidth is the number of items that can be transferred per unit of time, in the best possible conditions. Throughput is the time it really takes, when the machines perhaps aren’t operating smoothly.

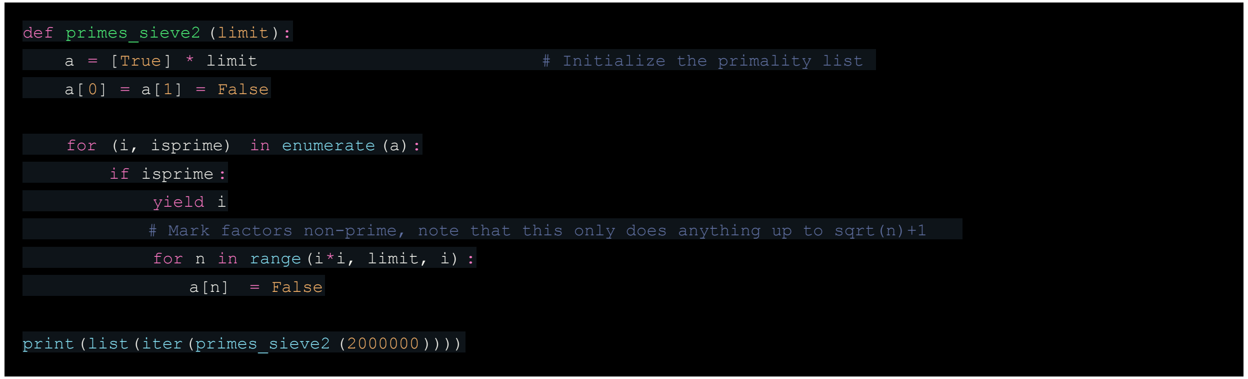




Write an efficient way to compute if a number is prime. Then, implement a method to find all the primes up to n.



```python
def prime2(n):
    for i in range(2, int(math.sqrt(n))+1):
        if n%i == 0:
            return False
    return True
```



```python
def primes_sieve2 (limit):
    a = [True] * limit                          # Initialize the primality list
    a[0] = a[1] = False

    for (i, isprime) in enumerate (a):
        if isprime:
            yield i
            # Mark factors non-prime, note that this only does anything up to sqrt(n)+1
            for n in range (i*i, limit, i) :
                a[n] = False

print(list(iter(primes_sieve2 (2000000))))
```

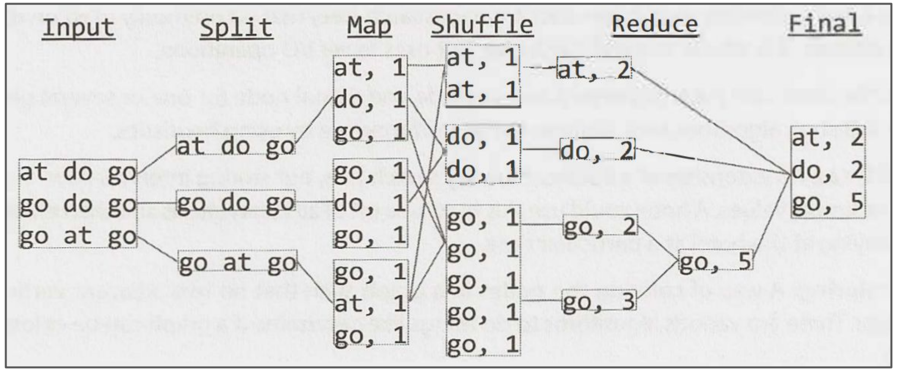
At a high level, what is MapReduce?

MapReduce is a programming model/framework for processing and generating big data sets with a parallel, distributed algorithm on a cluster. Its main advantage is that it is very simple, requiring only 2 functions as input to leverage many machines at scale:

- **Map** function takes in data and outputs a {key, value} pair.
- **Reduce** takes a key and a set of values, reducing the values into a single one. It then outputs a new {key, value} pair (which could be used for more reducing).

Then, the full pipeline is:

1. **Splitting** data to multiple machines
2. Each machine performs **map** independently, with no communication between each other
3. A **shuffle** operation reorganizes the data so that all {key,value} pairs from map go to the same reduce machine
4. **Reduce** is performed to consolidate results

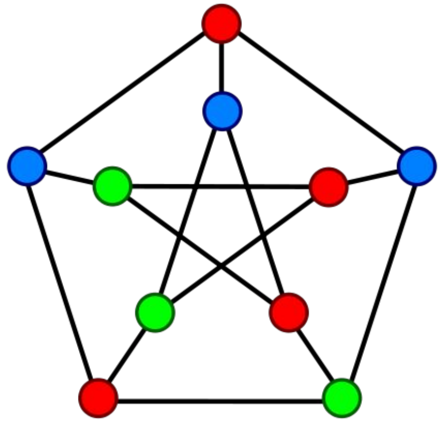


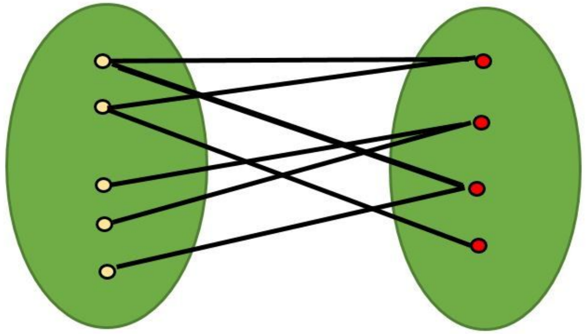
Note that while powerful, forcing this structured type of computation does limit some data processing tasks. There are improvements/generalizations of MapReduce that solve this.

What is graph coloring? What property do bipartite graphs have wrt coloring?

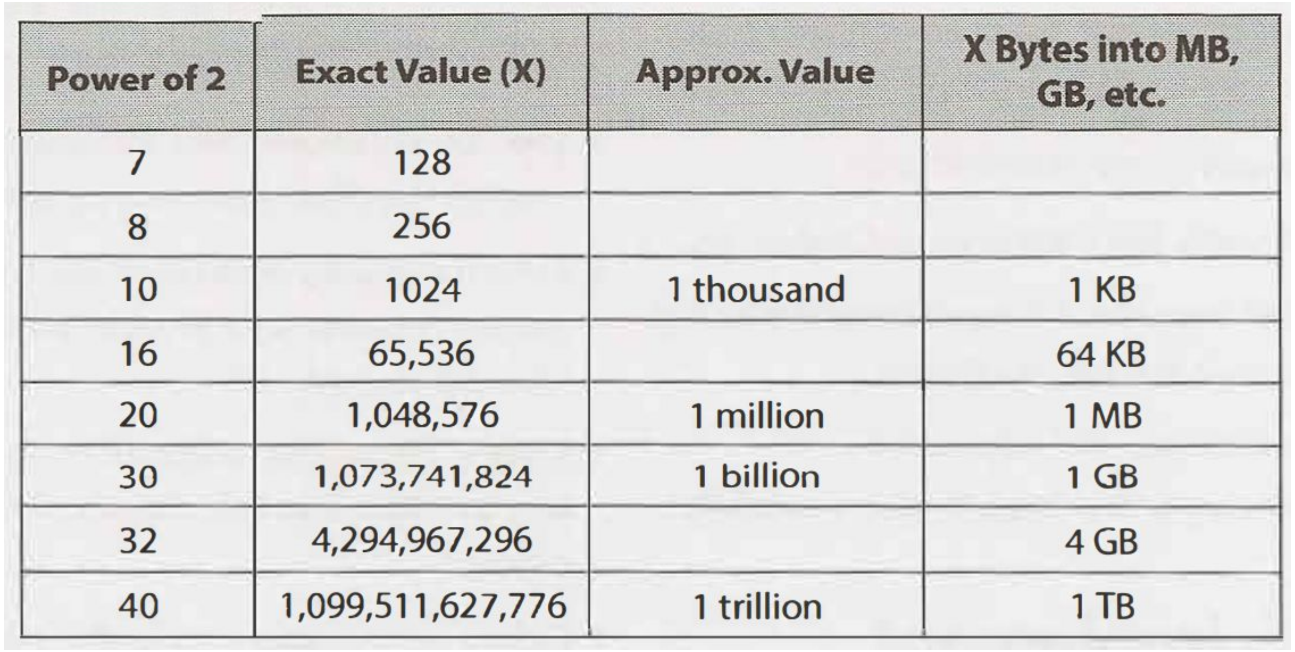
Graph coloring is a way of coloring the nodes in a graph such that no two adjacent vertices have the same color.

A **bipartite graph** is a graph where you can divide its nodes into 2 sets such that every edge stretches across the two sets. That is, there is never an edge between two nodes in the same set. There are algorithms to check if a graph is a bipartite graph. A graph is a bipartite graph if and only if it can be colored with 2 colors.



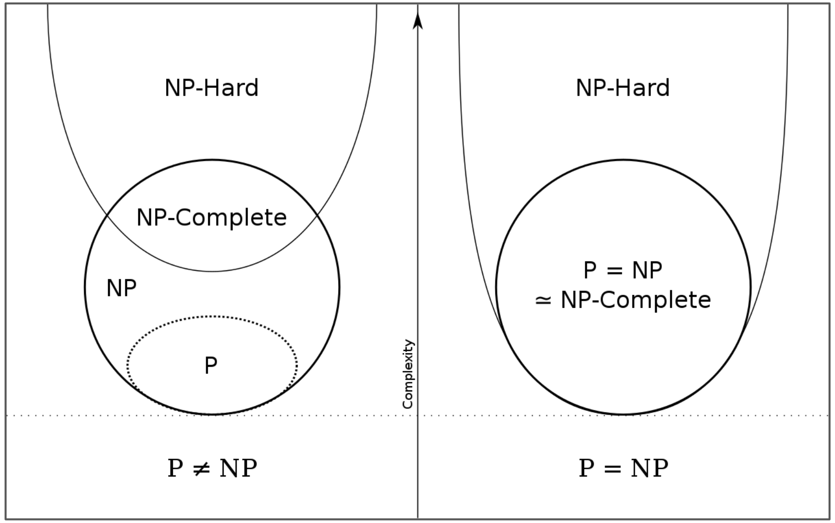


What’s the power of 2 table?



Power of 2	Exact Value (X)	Approx. Value	X Bytes into MB, GB, etc.
7	128		
8	256		
10	1024	1 thousand	1 KB
16	65,536		64 KB
20	1,048,576	1 million	1 MB
30	1,073,741,824	1 billion	1 GB
32	4,294,967,296		4 GB
40	1,099,511,627,776	1 trillion	1 TB

Define P, NP, NP-Complete, and NP-Hard. At a high level, what is the P vs NP problem?



- **P**: Decision problems that can be solved in polynomial time
- **NP**: Decision problems where given a solution, that solution's correctness can be verified in polynomial time
- **NP-Complete**: The set of "universal" problems X in NP where any NP problem Y can be reduced to X in polynomial time.
- **NP-Hard**: These problems are at least as hard as NP-Complete problems. If X is NP-Hard, there must be a NP-Complete problem Y which reduces to X in polynomial time.
  - Note: NP-Hard problems do not have to be decision problems.
- The **P vs NP problem** essentially asks "if a problem is easy to check for correctness, must it also be easy to solve?

Note: Decision problems are problems with a yes/no answer.

<u>Question goes here</u>

Answer goes here