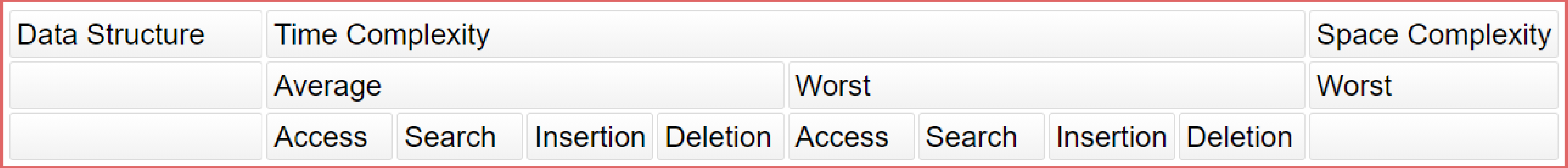
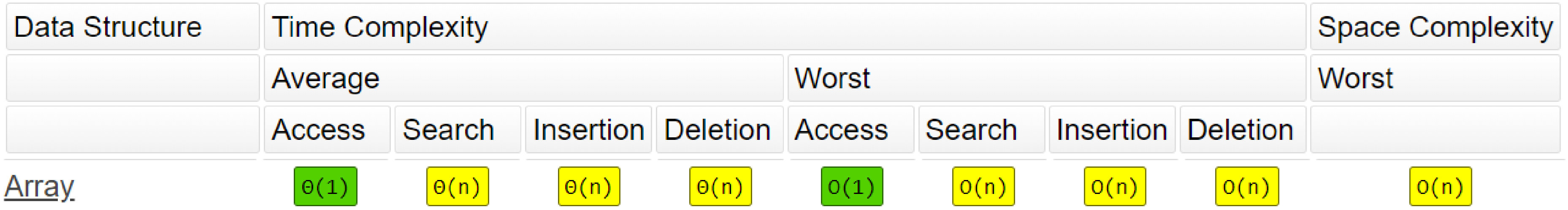
Data Structures
&
Algorithms



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 / fragmentation.



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). In python use collections.deque, which is linked-list based:

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.

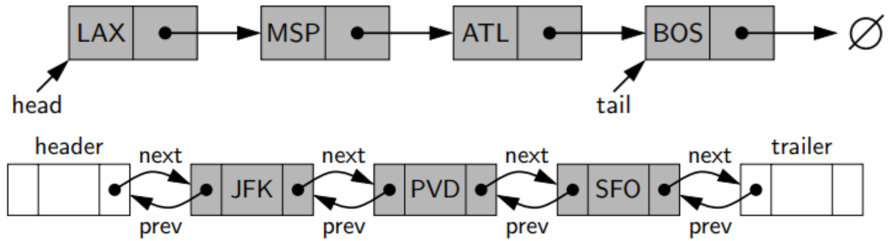
A queue is first in, first out (FIFO) with space complexity O(n). Enqueue and dequeue have time complexity O(1). Using collections.deque:

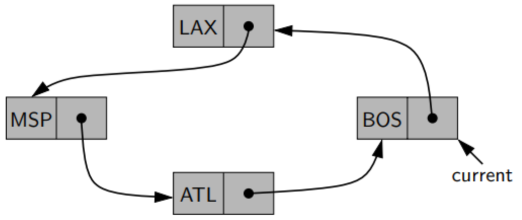
len(D)	Number of elements
D.appendleft(e)	Add to beginning
D.append(e)	Add to end
D.popleft()	Remove from beginning
D.pop()	Remove from end
D[i]	Arbitrary access
D.remove(e)	Find and remove element e
D.insert(i, e)	Insert element e at index i
del D[i]	Remove element at index i

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.

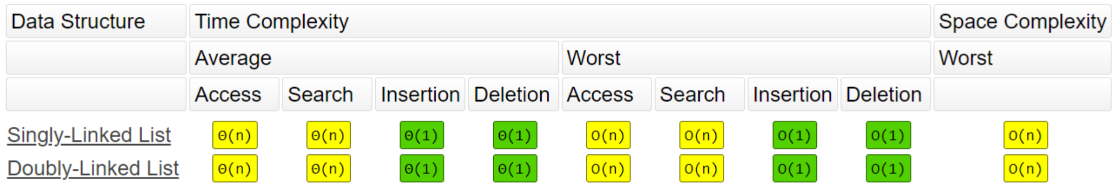
- A **singly linked list** is a collection of nodes each of which store an element and a reference to the next node. The nodes are not contiguous in memory, so unlike lists, inserting at the beginning takes O(1). However, you cannot efficiently delete a node that is not the head.
- A **circularly linked 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.
- ; 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.

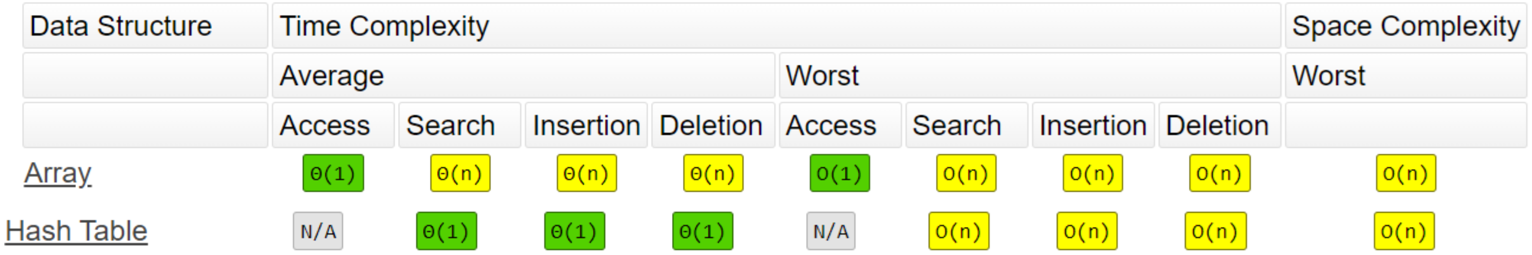
len(D)	Number of elements
D.appendleft(e)	Add to beginning
D.append(e)	Add to end
D.popleft()	Remove from beginning
D.pop()	Remove from end
D[i]	Arbitrary access
D.remove(e)	Find and remove element e
D.insert(i, e)	Insert element e at index i
del D[i]	Remove element at index i



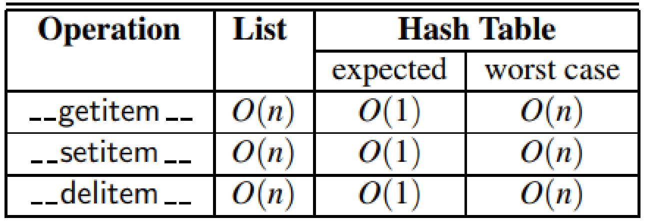
Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Singly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Doubly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)

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

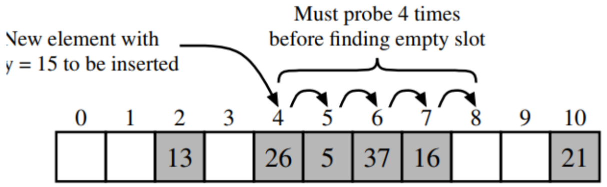
- A **hash table** contains a contiguous array of N memory locations. Its implemented as a dict in python.
- Given a pair of {immutable key k, value v}, we use a hash function hash(k) to map variable-length k into fixed locations in {0,...,N-1}
- In the case of collisions, there are two options:
  - **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.

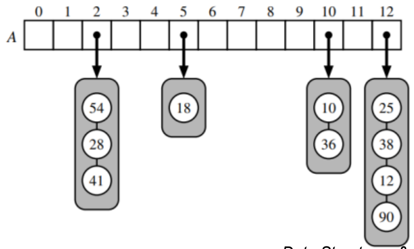


Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	Θ(1)	Θ(n)	Θ(n)	Θ(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Hash Table	N/A	Θ(1)	Θ(1)	Θ(1)	N/A	O(n)	O(n)	O(n)	O(n)



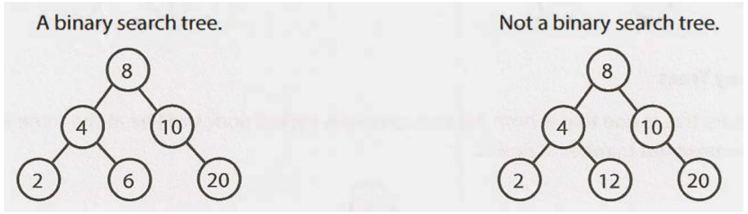
Operation	List	Hash Table	
		expected	worst case
__getitem__	$O(n)$	$O(1)$	$O(n)$
__setitem__	$O(n)$	$O(1)$	$O(n)$
__delitem__	$O(n)$	$O(1)$	$O(n)$

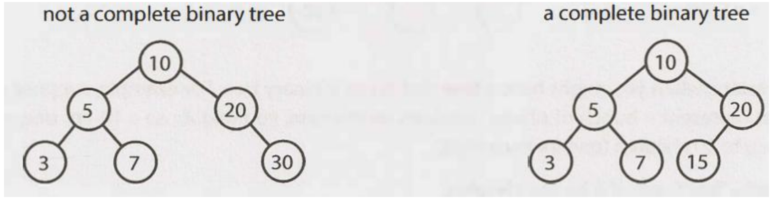


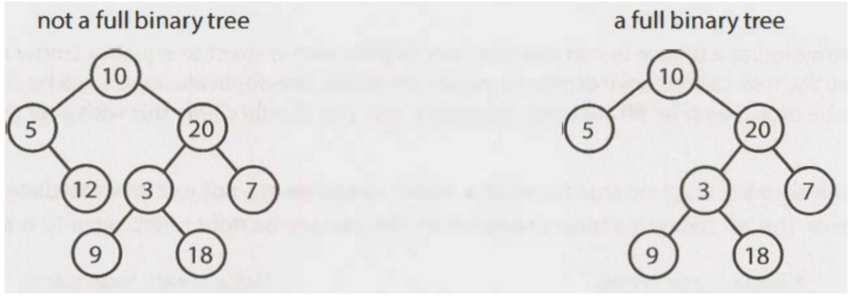


Define Tree, binary tree, binary search tree. What’s the difference between complete, full, and perfect binary trees?

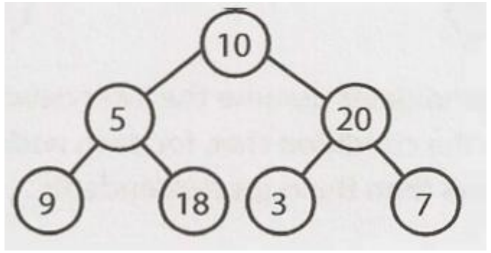
- A **tree** is a data structure which has a root node, and recursively, each child node has >=0 child nodes.
- A **binary tree** is a tree in which each node has <=2 children.
- A **binary search tree** is a binary tree in which for all nodes n, all left descendents <= n <= all right descendents.
- 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.





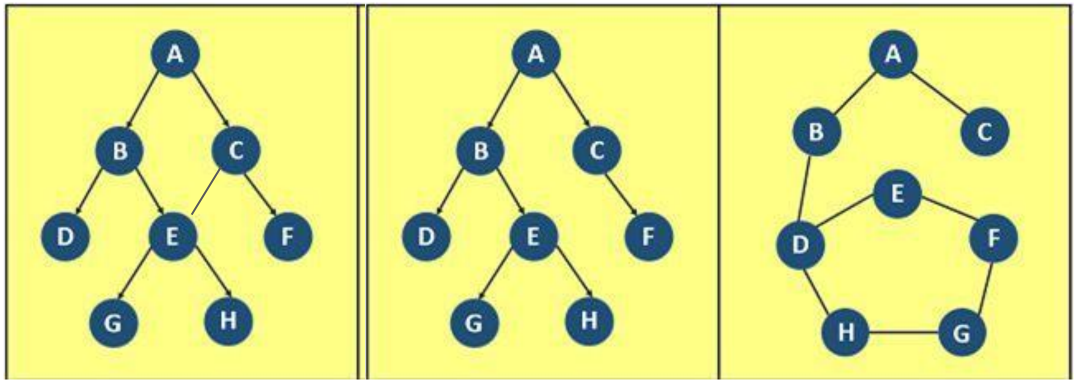


Perfect binary tree



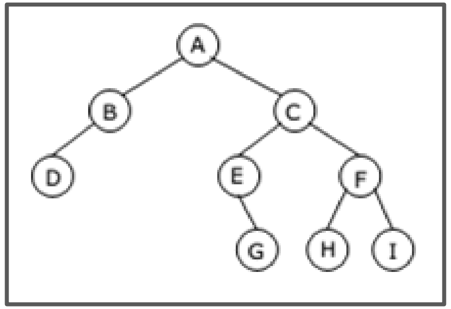
What is the difference between a graph and a tree?

- 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);
         }
      }
   }
}
```

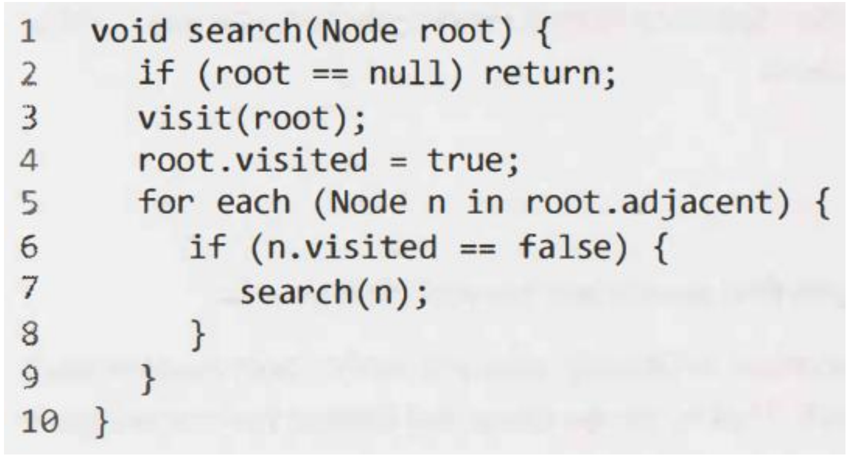
- Preorder traversal yields:
  A, B, D, C, E, G, F, H, I
- Inorder traversal yields:
  D. B. A. E. G. C. H. F. I
- Postorder traversal yields:
  D. B. G. E. H. I. F. C. A
- Level order traversal yields:
  A, B, C, D, E, F, G, H, I

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

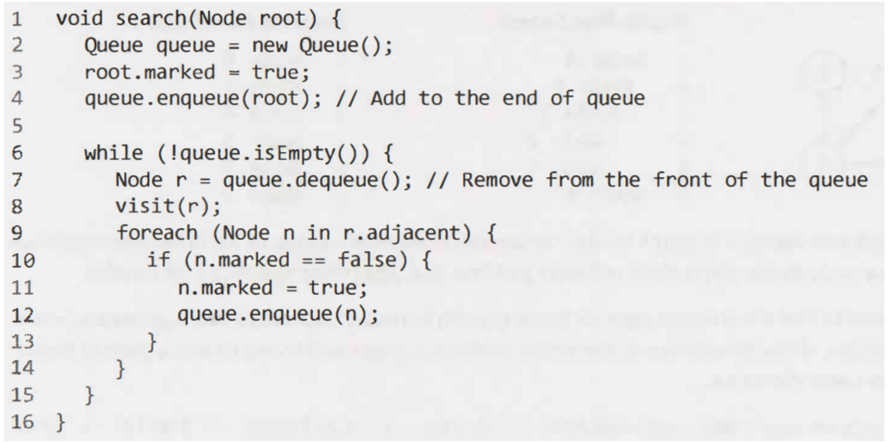
What are the ways to traverse a graph? Contrast them. How does it differ from traversing a tree?

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|).

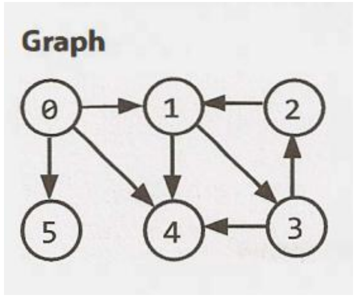
The main difference between tree and graph traversal is that for the latter, you need to mark nodes as visited, or there might be infinite loops.

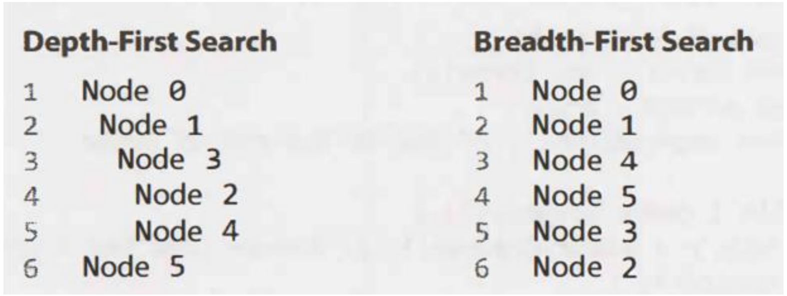


```
void search(Node root) {
   if (root == null) return;
   visit(root);
   root.visited = true;
   for each (Node n in root.adjacent) {
      if (n.visited == false) {
         search(n);
      }
   }
}
```



```
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);
      }
    }
  }
}
```





Depth-First Search

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

Breadth-First Search

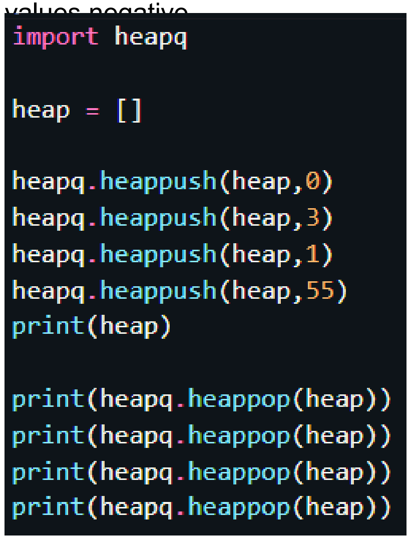
```
Node 0
Node 1
Node 4
Node 5
Node 3
Node 2
```

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

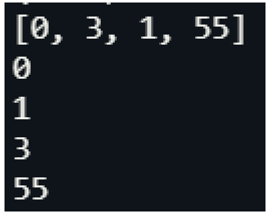
A min/max heap is a complete binary tree (so, filled except for rightmost elements on last level), where each node is smaller/larger than its children. So, the root is the smallest/largest element. The two key operations are:

- 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.

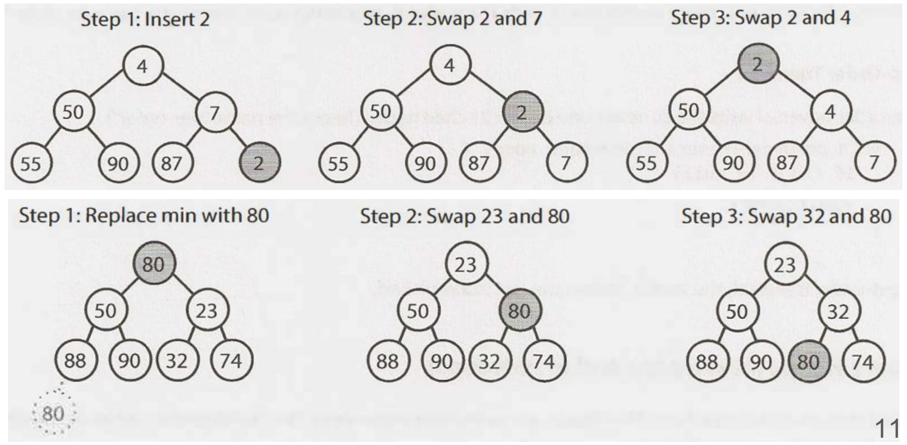
Heaps can be implemented with the heapq library, using heappush and heappop on a list. Note that it **defaults to a min heap**; for a max heap, make the values negative.



```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?

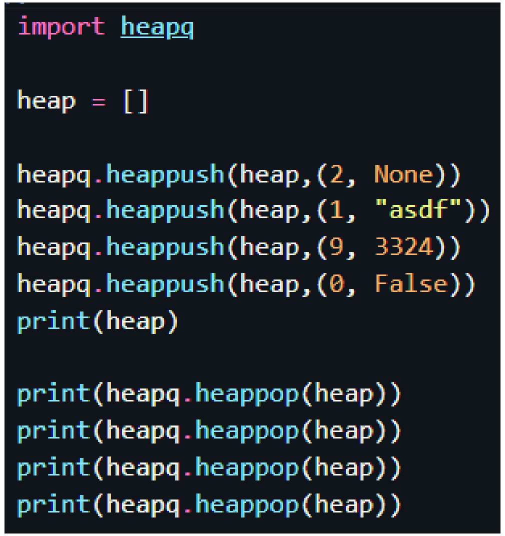
- Top k largest: use min heap to constantly remove the minimum elements, leaving only the largest
- Top k smallest: use max heap to constantly remove the largest elements, leaving only the smallest

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

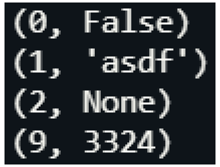
A **priority queue** takes in {key, value} pairs. The main operations are adding elements, and then removing the element with the minimum key. Three ways to implement one is as follows:

- 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))
```



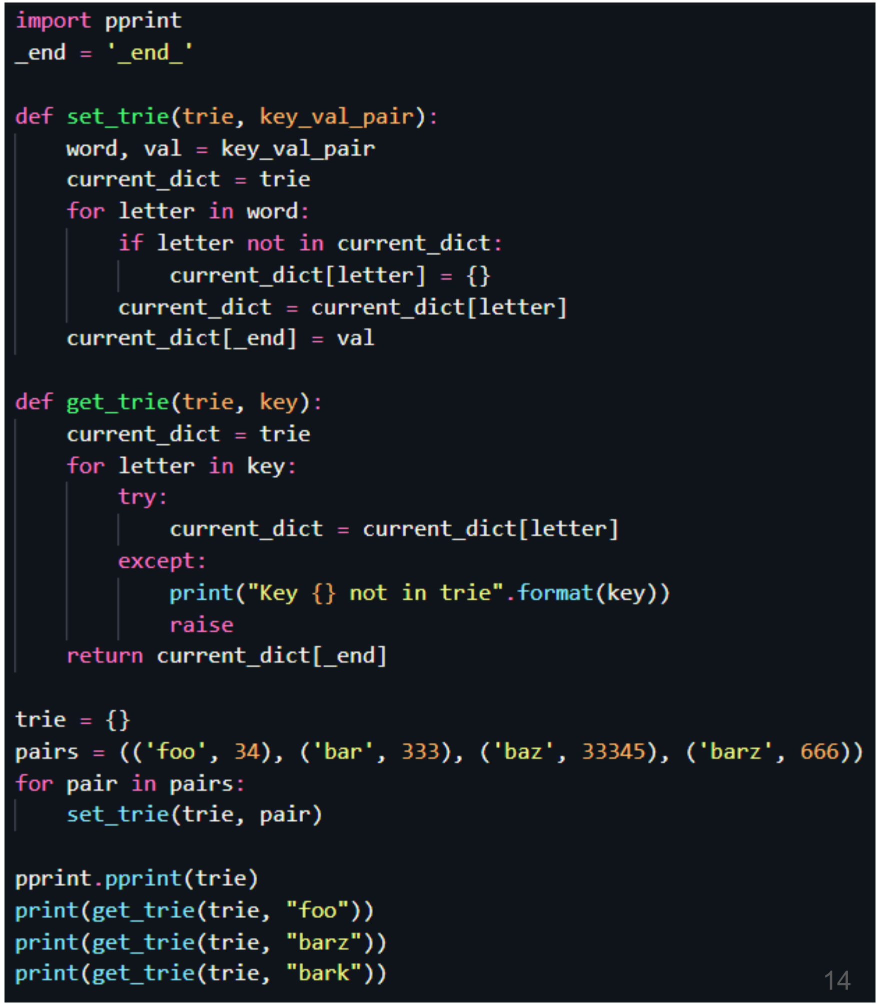
```
(0, False)
(1, 'asdf')
(2, None)
(9, 3324)
```

Note that this is possible since:

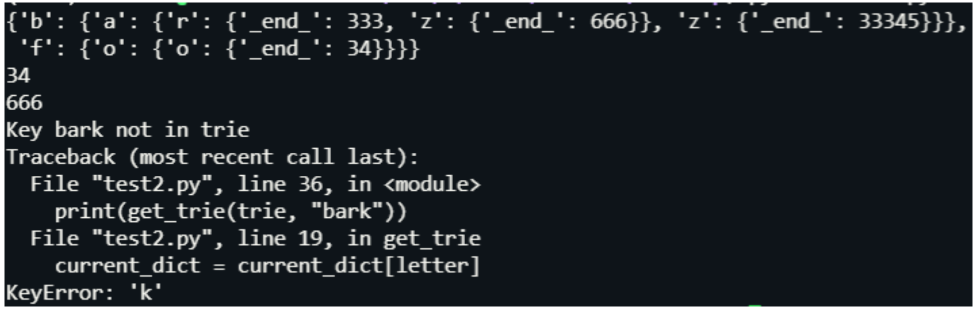
1. *python tuples evaluate “less than” one element at a time, from left to right in the tuple*
2. *python uses lazy evaluation*

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:

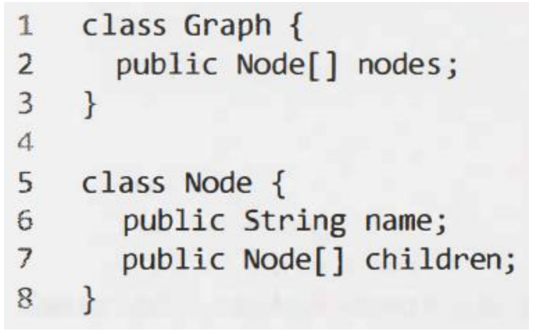
- **Creation**
    - X = set([1,2,3])
    - X = {1,2,3}
    - X = set()
- **Modifying**
    - X.add(4)
    - X.remove(4)
- **Union of sets X1, X2**: X1 | X2
- **Intersection of sets X1, X2**: X1 & X2
- **Difference of sets X1, X2**: X1 - X2
- **XOR of sets X1, X2**: X1 ^ X2
- **X1 is a subset of X2**: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

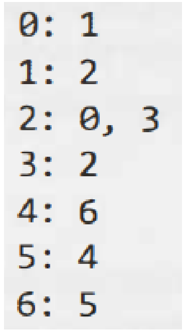
- To create, use x = frozenset([1,2,3]). The argument must be an iterable, so frozenset(1) or frozenset(1,2) won't work.
- All the set operations still work, eg frozenset([1,2]) | frozenset([2,3]) = frozenset([1,2,3]). However, because they're immutable, you can't add or remove.

What are 3 ways to describe a graph?

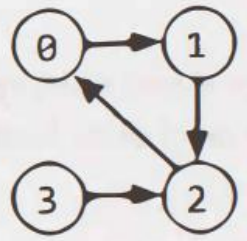
- Directly in the node class
- As an adjacency list
- As an adjacency matrix, where a true value at matrix[i][j] represents an edge from node i to node j (ie, column node -> row node).

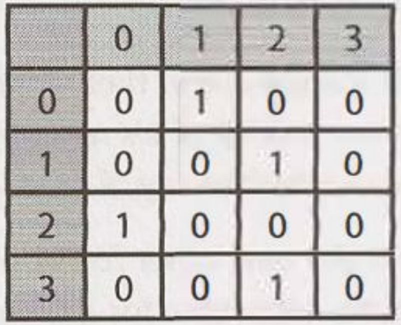


```
class Graph {
  public Node[] nodes;
}

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



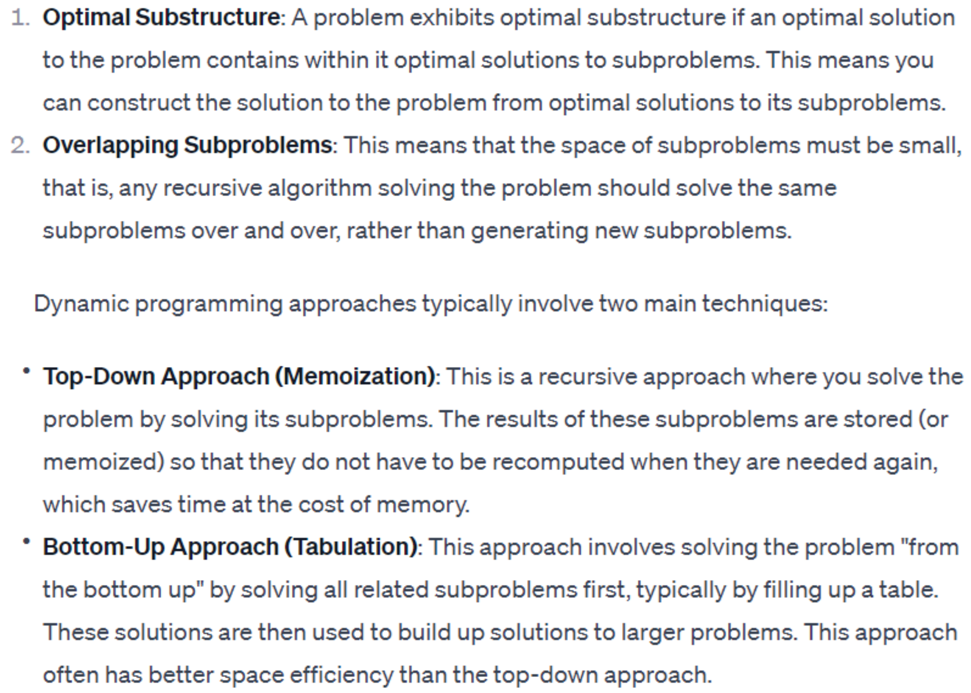
```
0: 1
1: 2
2: 0, 3
3: 2
4: 6
5: 4
6: 5
```





	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?



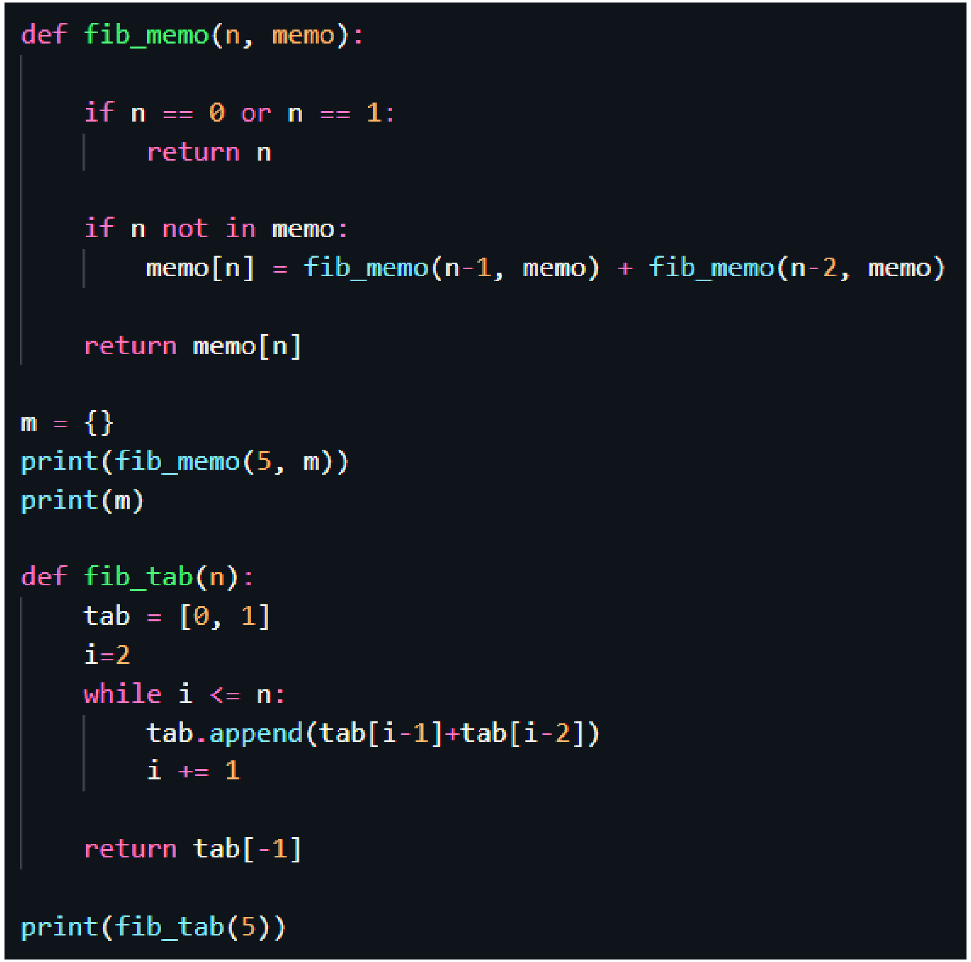
1. **Optimal Substructure**: A problem exhibits optimal substructure if an optimal solution to the problem contains within it optimal solutions to subproblems. This means you can construct the solution to the problem from optimal solutions to its subproblems.
2. **Overlapping Subproblems**: This means that the space of subproblems must be small, that is, any recursive algorithm solving the problem should solve the same subproblems over and over, rather than generating new subproblems.

Dynamic programming approaches typically involve two main techniques:

- **Top-Down Approach (Memoization)**: This is a recursive approach where you solve the problem by solving its subproblems. The results of these subproblems are stored (or memoized) so that they do not have to be recomputed when they are needed again, which saves time at the cost of memory.
- **Bottom-Up Approach (Tabulation)**: This approach involves solving the problem "from the bottom up" by solving all related subproblems first, typically by filling up a table. These solutions are then used to build up solutions to larger problems. This approach often has better space efficiency than the top-down approach.

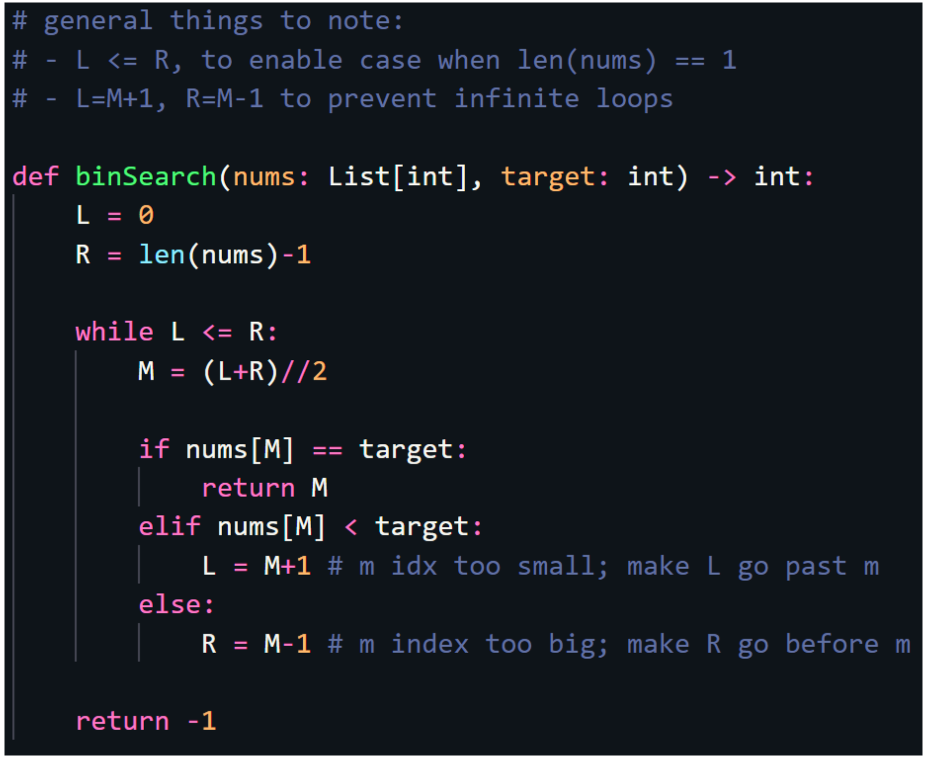
- When figuring out time complexity for recursion, multiply number of recursive calls with the time it takes for each result/call.

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))
```

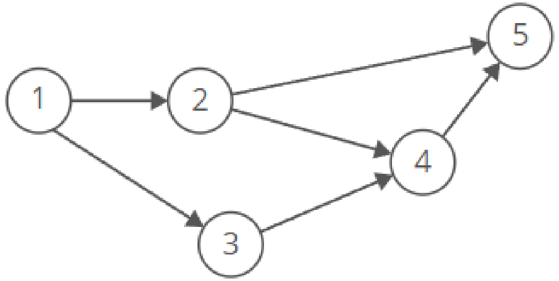
Implement binary search.



```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
```

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.

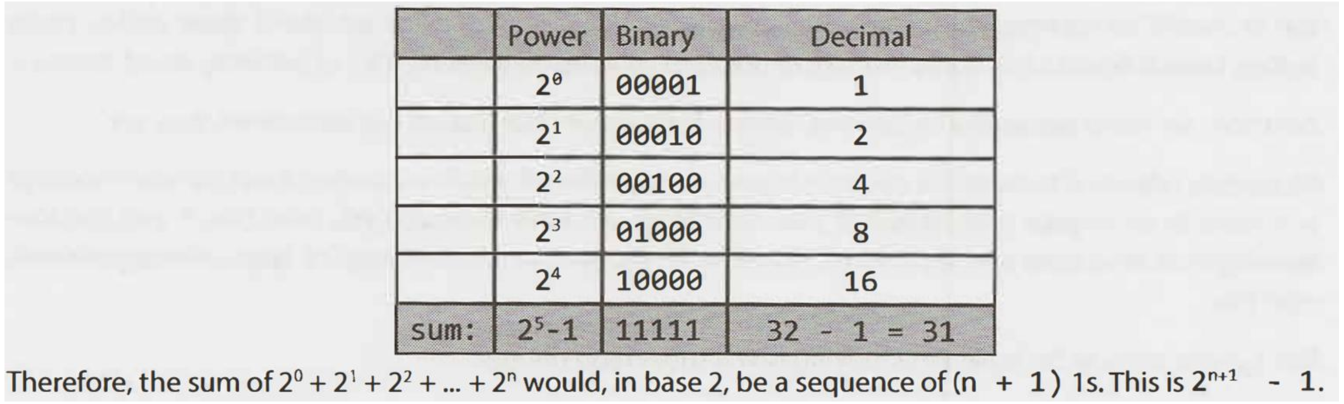
Imagine pouring water on the source vertex, and imagine all the edges are tubes that can spread the water to neighboring vertices in one unit of time. BFS comes down to simulating this process and asking at what time the water reaches each vertex: that time is the distance of the vertex from the source.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

Time/Space Complexity

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.

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

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?

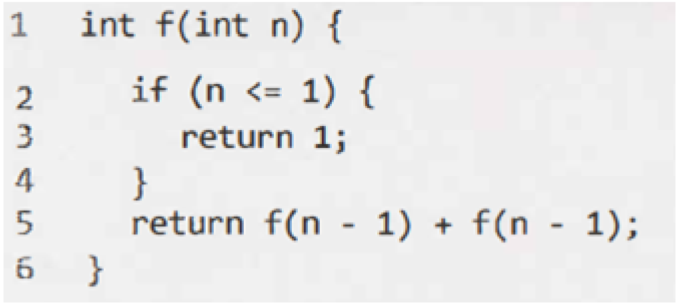
- In general, O(n^min{k,n-k})
- In most cases, k is very small, so O(n^k)

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, $1 + 2 + 4 + 8 + \ldots + N \approx 2N$ 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);
}
```

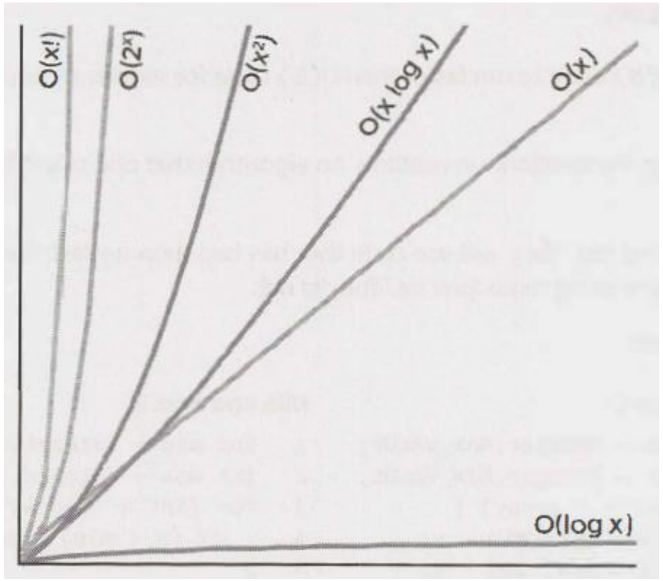
There are 1+2^1+2^2+...+2^n = 2^(n+1)-1 = O(2^n) recursive calls, each taking O(1) to complete. Thus, the time complexity is O(2^n).

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)

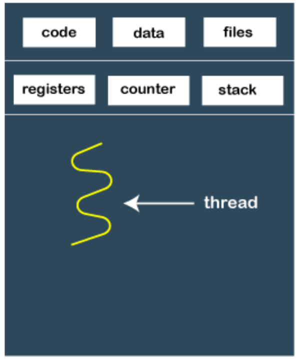
Rank the rate of increase for common big O times.



Misc

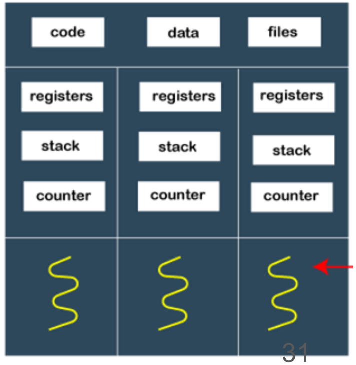
What are processes and threads?

- 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 CPython, 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. This was done for simplicity/compatability
  - However, other python implementations don't have a GIL



Single-threaded process

	Processes	**Threads**
Memory Space	Each has a separate memory space	Threads of a process run in a shared memory space
Communication	Harder to share objects between processes; more isolated from each other. Need inter-process communication (IPC)	Easier to share objects in the same memory, but need to be careful to avoid race conditions
Python Package	threading	multiprocessing
Overhead	Larger memory footprint	Lightweight, low memory footprint
True parallelism?	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	**Use case in python:** For when you want to really do more than one thing at a given time on the CPU (CPU-bound).	**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.



Multi-threaded process

misc

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
```

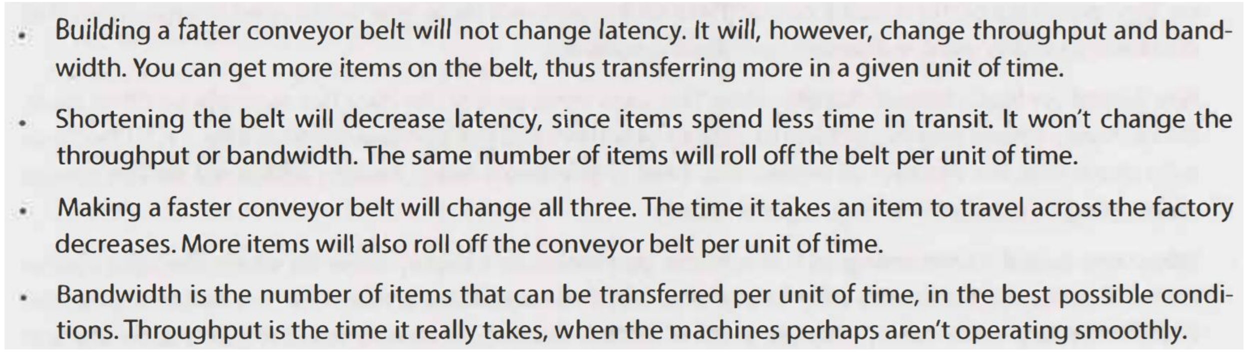
A **deadlock** can occur if a thread is waiting for an object lock that another thread holds, and that second thread is waiting for an object lock that the first thread holds. For example, if you have the following code for bank transfer, calling transfer(a,b) and transfer(b,a) will cause a deadlock 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);
 }
```

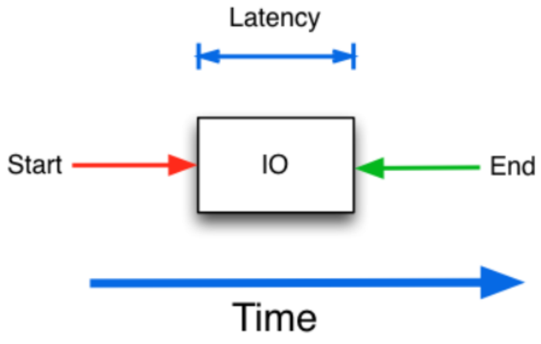
A **semaphore** is a generalization that allows x number of threads to enter; for example, this can be used to limit the number of CPU, IO, or RAM intensive tasks running at the same time.

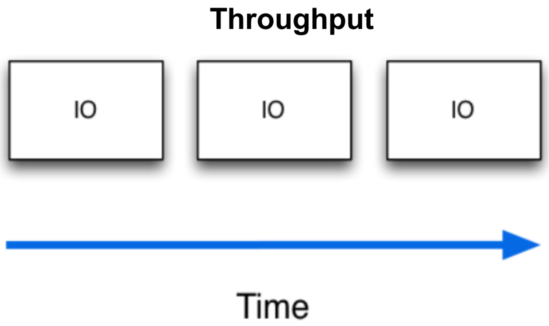
What's the difference between bandwidth, throughput, and latency? Compare them in a conveyor belt context; how do they change as the belt is faster/slower, longer/shorter?

- **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.





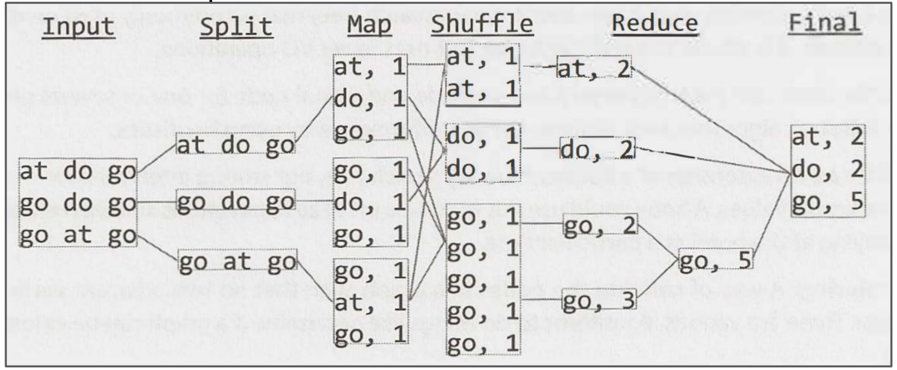
At a high level, what is MapReduce?

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.