Preparing for Coding Interviews

Jack Dunn and Daisy Zhuo

Why are we talking about coding interviews?

- Most jobs you apply for will probably have some portion of the interview that involves testing your programming ability
- Especially at tech companies, the interview is almost exclusively based around coding and not like a traditional interview at all
- In a coding interview, you will typically be given a challenging problem and asked to code up a solution over the course of the interview, typically in a text editor or on a whiteboard.

What makes a coding interview different to a normal interview?

- The biggest difference to a normal interview is that if you are not prepared, you will probably perform poorly
- Coding interviews are closer to an exam, they can and should be studied for in order to maximise your chance of success

How do you prepare for a coding interview?

There are three important aspects to your preparation:

1. Solid knowledge of the theory (data structures and algorithms)
2. Knowing how to approach and behave in the interview
3. Practice!

Overview of Today

- Big-O notation
- Arrays and strings
- Linked lists
- Stacks and queues

Break

- Recursion and dynamic programming
- Trees and graphs
- Sorting
- Overview of the coding interview
- Mock interviews and interview question practice

Big-O Notation

Which algorithm is fastest?

For a given problem, there might be multiple valid solution approaches.

How are we going to decide which is best for our use case?

We could try each method out and choose the fastest?

We need a way of determining the efficiency of an algorithm without needing to implement it

Big-O notation

Big-O notation is the typical way to measure the complexity of an algorithm

We measure how the algorithm performance scales with respect to the size of the input. It provides a **worst-case** analysis of the algorithm.

We typically measure two aspects of performance:

- time complexity (how long does it take to run?)
- space complexity (how much memory space does it require?)

Examples (all for time complexity and an input of size n):

- $O(n)$ means the time taken can be as much as a linear function of n
- $O(\log n)$ means the time taken can be as much as a linear function of $\log n$
- $O(1)$ means the time taken is independent of n

Some big-O examples

Finding the minimum element of an array:

In [ ]:
```
minval = float('inf')
for val in array:
    minval = min(minval, val)
```

- Time complexity is $O(n)$ (have to check each value once)
- Space complexity is $O(1)$

Calculating 1-norm of square matrix:

In [ ]:
```
total = 0
for i in xrange(n):
    for j in xrange(n):
        total += abs(X[i][j])
```

- Time complexity is $O(n^2)$
- Space complexity is $O(1)$

Simplifying big-O

When we use big-O, we always reduce the notation into its simplest form. We only care about measuring how the algorithm **scales** with the input size, not calculating the exact complexity as a function of input size.

Calculating min and max of array:

In [ ]:
```
minval = +float('inf')
maxval = -float('inf')
for val in array:
    minval = min(minval, val)
    maxval = max(maxval, val)
```

Even though there are $2n$ operations here, the time complexity is still $O(n)$ - we ignore the constant factors

Calculating 1-norm of square matrix and sum of diagonals:

In [ ]:

```
total = 0
diag = 0
for i in xrange(n):
    for j in xrange(n):
        total += abs(X[i, j])
    diag += X[i, i]
```

The number of operations is $O(n^2 + n)$ which we reduce to $O(n^2)$

Choosing an algorithm using big-O

The main purpose of big-O is to compare different approaches.

Suppose we want to find the position of a specified value in an array. Our simplest approach is simply to loop over the array until we find the value we are looking for:

In [ ]:
```
for (i, val) in enumerate(array):
    if val == searchval:
        break
```

Worst-case, we have to iterate over the entire array to find our value, so the time complexity is $O(n)$

Can we do better?

What if we knew the array was pre-sorted?

If the array is sorted, we can use a binary search

We examine the middle element of the array. If it matches, we are done. If it's less than our search value we search the upper half, otherwise we search the lower half. This proceeds recursively until we find our target.

What's the time complexity of this?

At each step, we divide the search space in half. This means with k iterations we can search 2^k elements. Therefore we need $2^k > n$, which gives $k > \log n$ so the time complexity is $O(\log n)$

This approach should be much faster, especially as n grows large!

Arrays and Strings

Arrays, strings, and related data structures

Arrays and strings are probably the most common data structures. Questions about these can typically be used interchangeably in interviews as a string is basically just an array of characters for practical purposes.

Hash tables (or dictionaries) are also very common, and are used to create a mapping from key to value pairs. They are very useful because they have $O(1)$ lookup to retrieve the value for a given key, whereas we saw that checking membership of an array will typically take $O(n)$ time. A variant of these is the **hash set (or set)** which only stores keys.

Array lists are resizable arrays (like list in python) that can grow as you add entries. The typical way these are implemented are as fixed-size arrays that are doubled in size when more space is needed (by creating another array of the new size and copying the old data over).

Example question 1

Implement an algorithm to determine if a string has all unique characters.

What's the simplest answer you can come up with?

Could a dictionary or set help?

As a simple solution, we can compare each character to every other character in the string, which would take $O(n^2)$ time.

We could also use a hash set to keep track of which characters in the string we have seen before and return false if we encounter a character that's already shown up.

In [ ]:

```python
def is_unique(s):
    d = set()
    for char in s:
        if char in d:
            return False
        else:
            d.add(char)
    return True
```

Example question 1

Implement an algorithm to determine if a string has all unique characters.

What if you cannot use additional data structures? Can we do better than $O(n^2)$?

How else can we figure out if there are multiples of a character?

If we can modify the input string, we can sort it at a cost of $O(n \log n)$. This would turn "abcbda" into "aabbcd", and then we can simply loop over the string and check for doubles in $O(n)$ time.

Example question

Write a function to check if a given string is a permutation of a palindrome.

Think about the defining characteristics of a palindrome

Could a dictionary help?

Let's think about it. For a string to be a palindrome, it needs to read the same forward and backward. This means that other a possible middle letter, every letter appears twice.

This means we just need to check if there is more than one letter that is used an odd number of times.

What's the easiest way to count? We can use a dictionary to keep track of the counts

In [ ]:

```
def is_palindrome(s):
    counts = dict()
    for char in s:
        if not char in counts:
            counts[char] = 0
        counts[char] += 1
    found_odd = False
    for char_count in counts.itervalues():
        if char_count % 2 == 1:
            if found_odd:
                return False
            else:
                found_odd = True
    return True
```

What's the complexity of this?

Example question 3

Write an algorithm that takes in an m x n matrix and for any element of the matrix that is zero, sets the entire row and column containing that element to 0

What's the simplest approach?

We can just loop over the matrix and start processing any zeros we find. Are there any issues with this?

What if we use a matrix to store the positions of any zeros before we modify?

What is the time complexity of this? Can we do better?

Can we do it without needing $O(mn)$ space?

We can just track the columns and rows that have any zero in them, and then finish by zeroing these out

In [ ]:

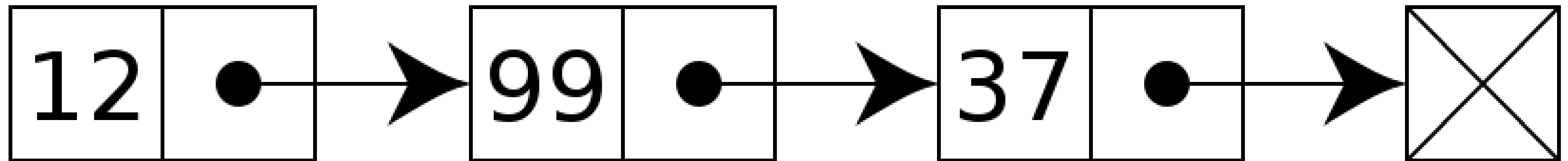
```
def set_zeros(mat):
    m = length(mat)
    n = length(mat[0])

    rows = [False for _ in xrange(m)]
    cols = [False for _ in xrange(n)]

    for i in xrange(m):
        for j in xrange(n):
            if mat[i][j] == 0:
                rows[i] = True
                cols[j] = True

    for i in xrange(m):
        if rows[i]:
            for j in xrange(n):
                mat[i][j] = 0
    for j in xrange(n):
        if cols[j]:
            for i in xrange(m):
                mat[i][j] = 0

    return mat
```

Linked Lists

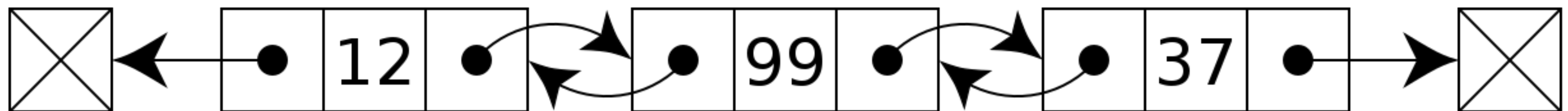
Intro to linked lists

Linked lists are a data structure used to represent a sequence of nodes

A **singly-linked** list has links from each node to the next in the chain



A **doubly-linked** list has links in both directions



Implementing a linked list

It's easy to implement a linked list ourselves. This is a singly-linked list:

In [1]:
```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

    def append(self, data):
        end = Node(data)
        current = self
        while current.next:
            current = current.next
        current.next = end
```

We might also want to delete a specified node:

In [3]:
```
def delete_node(head, data):
    current = head

    # First node is deleted
    if current.data == data:
        return current.next

    while current.next:
        if current.next.data == data:
            current.next = current.next.next
            return head
        current = current.next

    return head
```

Example question 1

Write a function that removes duplicate nodes from an unsorted linked list.

How can we track if we have seen a particular node before?

What if you use a dictionary or set to track which nodes have been visited?

In [ ]:

```python
def remove_dups(node):
    visited = set()
    previous = None

    while node:
        if node.data in visited:
            # We have seen this before, remove it
            previous.next = node.next
        else:
            visited.add(node.data)
            previous = node
        node = node.next
```

What's the complexity of this?

How about if we are not allowed to use any extra storage space? Can we still remove the duplicates?

We can loop over the list and at each node scan through the remainder of the list for any duplicates of the current value

In [ ]:

```
def remove_dups(node):
    while node:
        # Scan ahead for duplicates
        future = node
        while future.next:
            if future.next.data == current.data:
                future.next = future.next.next
            else:
                future = future.next
        # Move to next node
        node = node.next
```

What's the time and space complexity here?

Example question 2

Write a function that deletes a node in the middle (i.e. not the first or last node) of a singly-linked list, given only access to that node.

Example: If the list is `a->b->c->d->e->f` and you are given node `c`, nothing is returned, but the list should now look like `a->b->d->e->f`

If we are trying to delete `c` in the above example, we want it to look like `d`. How can we do this?

We can simply copy the data and next fields from `d` into `c` and we are done

In [7]:
```
def delete_middle(node):
    node.data = node.next.data
    node.next = node.next.next
```

Example question 3

Given two singly-linked lists, determine if the two lists intersect and if so return the intersecting node.

Would a dictionary or set help?

We could use a set to track which nodes appear in the first list and then traverse the second list and stop when we find the first match in the set.

We can do better though, we don't need any extra data structures.

Let's think about what intersecting and non-intersecting lists would look like

Let's first worry about whether there is an intersection, we can find it later. How can we tell if the lists intersect?

Observe that if the lists intersect, their tail is the same. How can we use this?

If the tail is the same, we can just traverse both lists and see if we end up at the same place.

Now we need to find the intersection point. Suppose the lists were the same length, how would we find the intersection?

If they were the same length, we could just traverse the lists at the same time until we found the first match.

How would this generalize to lists of different lengths?

If we know the difference between the lengths of the lists, can we use that?

We can just move ahead in the longer list by the length difference, and then advance each list together until we find the first match

In [ ]:

```
def length_and_tail(node):
    length = 1
    while node.next:
        length += 1
        node = node.next
    return length, node

def find_intersection(node1, node2)
    length1, tail1 = length_and_tail(node1)
    length2, tail2 = length_and_tail(node2)

    if tail1 != tail2:
        return None

    if length1 > length2:
        longer = node1
        shorter = node2
    else:
        longer = node2
        shorter = node1

    for _ in xrange(abs(length1 - length2)):
        longer = longer.next

    while longer != shorter:
        longer = longer.next
        shorter = shorter.next

    return longer
```

What's the time complexity here?

Stacks and Queues

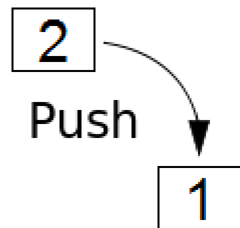
Intro to Stacks

A stack is a data structure that has the following operations.

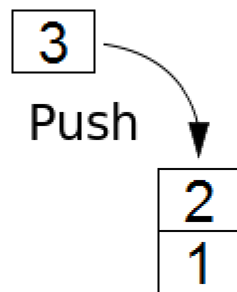
- isEmpty(): Returns true if the stack is empty, false otherwise.
- push(s): Inserts item s to the top of the stack.
- pop(): Removes and returns the item at the top of the stack.

All operations of a stack are assumed to take $O(1)$ time.

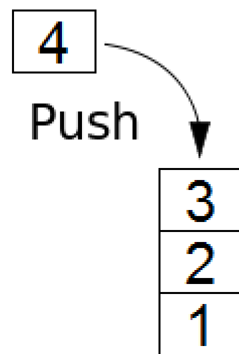
1



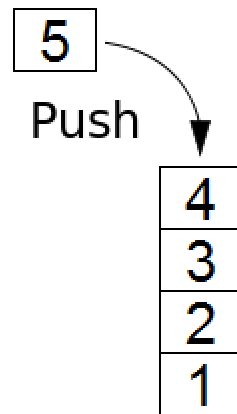
2



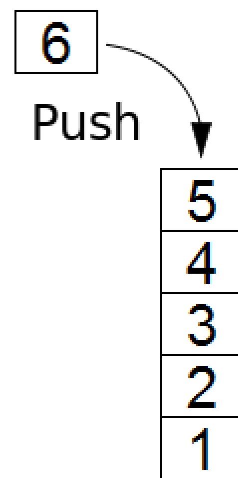
3



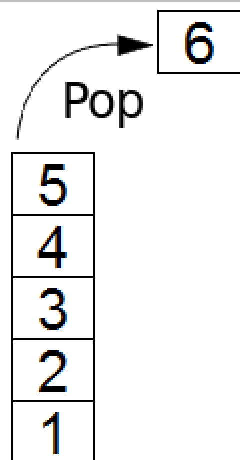
4



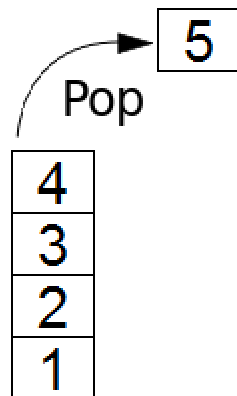
5



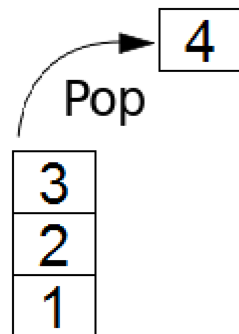
6



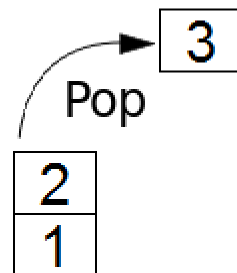
7



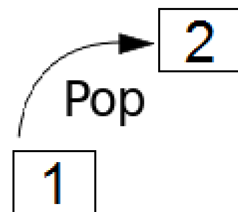
8



9



10



Implementing a Stack

A stack can be implemented using a singly-linked list.

The head of the linked list contains the top item of the stack. The next node in the linked list contains the second-to-top element, and so on.

For example, if we pushed 1, then 2, then 3, our linked list would be 3->2->1.

In [18]:

```
class Stack:
    def __init__(self):
        self.top = None

    def isEmpty(self):
        return not self.top

    def push(self, data):
        new_top = Node(data)
        new_top.next = self.top
        self.top = new_top

    def pop(self):
        if not self.isEmpty():
            data = self.top.data
            self.top = self.top.next
            return data
        return None
```

Example: Adding size to stack

Create a new stack data structure, called `NewStack`, which in addition to the normal functions of a stack has an additional method called `size()`, which returns the current number of elements in the stack. We can assume that we have access to a Stack data structure, which implements `isEmpty()`, `push(data)`, and `pop()` each in $O(1)$ time.

Each time `size()` is called, we could pop all the items out of the stack into a second stack, and count as we go. Then, since the second stack has the items in reverse, we would pop the items out of the second stack and push them back into the first stack. This would take `O(n)`.

Can this be done faster?

Better idea is to introduce a new variable `n`, which stores the number of elements in the stack.

Each time we push an element into the stack, we increase `n` by one. Each time we pop an element from the stack, we decrease `n` by one.

In [ ]:

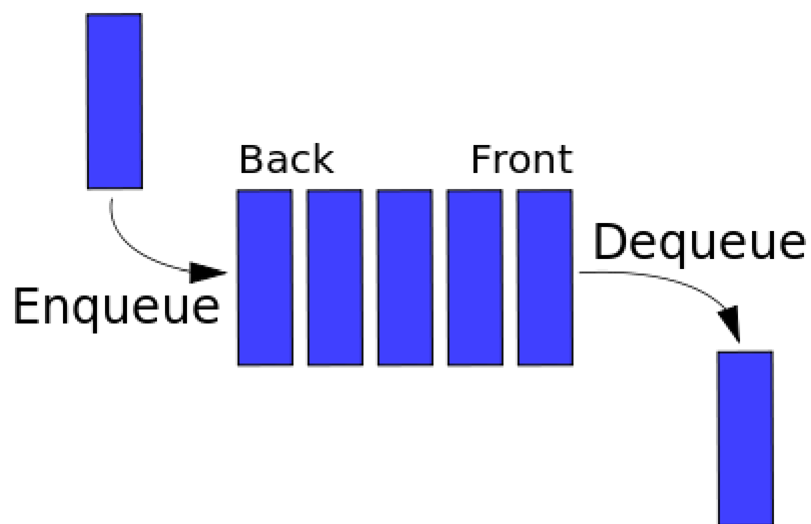
```
class NewStack:
    def __init__(self):
        self.stack = Stack()
        self.n = 0

    def isEmpty(self):
        return self.stack.isEmpty()

    def push(self, data):
        self.stack.push(data)
        n = n + 1

    def pop(self):
        self.stack.pop()
        n = n - 1
```

Intro to Queues

A queue is a data structure that mimics a line at a store. The order that the items are removed from the queue is the same as the as the order in which they were added.

- isEmpty(): Returns true if the queue is empty, false otherwise.
- enqueue(s): Inserts item s to the end of the queue.
- dequeue(): Removes and returns the item at the beginning of the queue.

All operations of a queue are assumed to take $O(1)$ time.



Implementing a Queue

A queue can also be implemented using a singly-linked list, with new items added to the tail of the linked list.

In [19]:
```
class Queue:
    def __init__(self):
        self.first = None
        self.last = None

    def isEmpty(self):
        return not self.first

    def enqueue(self, data):
        if not self.first:
            self.last = Node(data)
            self.first = self.last
        else:
            self.last.next = Node(data)
            self.last = self.last.next

    def dequeue(self):
        if not self.isEmpty():
            data = self.first.data
            self.first = self.first.next
            return data
        return None
```

Example question 1

Create an implementation of `Queue` which is implemented using two stacks.

We can use one stack to push in the incoming items. But, the stack will be in reversed order from that of a queue.

How can we make use of the second stack?

Let our stacks be called `s1` and `s2`.

Each time we perform `enqueue(data)`, we push the data into `s1`.

To perform `dequeue(data)`, we want the bottom element of `s1`. We can use the second stack to reverse the order of the elements by popping elements from `s1` and pushing into `s2`. After we see the bottom element, we return to `s1` by popping the elements from `s2` back into `s1`.

This implementation results in `enqueue(data)` taking $O(1)$ and `dequeue()` taking $O(n)$. This is slow if we repeatedly call `dequeue()`.

A better idea is to use a lazy approach, where we move the elements from `s1` into `s2` only when `s2` is empty. In this approach, `s1` has the newest elements on top and `s2` has the oldest elements on top.

When we dequeue an element, we want to remove the oldest element, so we remove the top element of `s2`.

If `s2` is empty, we move all of the elements from `s1` into `s2` to reverse the order.

In this approach, each element is moved from `s1` into `s2` at most once!

In [20]:

```
class Queue:
    def __init__(self):
        self.s1 = Stack()
        self.s2 = Stack()

    def isEmpty(self):
        return self.s1.isEmpty() and self.s2.isEmpty()

    def enqueue(self,data):
        # Push onto s1, which always has the newest elements
        # on top
        self.s1.push(data)

    def dequeue(self):
        if not self.isEmpty():
            # If s2 is empty, move the elements of s1 into
            # s2 ino order to reverse the order.
            if self.s2.isEmpty():
                while not self.s1.isEmpty():
                    self.s2.push(self.s1.pop())

            # Return the top element of s2, which is has
            # the oldest element on top
            return self.s2.pop()
        return None
```

Recursion

Introduction to Recursion

Terminology: In computer science, a distinction is made between *recursion* and *dynamic programming*.

A function is said to be *recursive* if its code consists of calling itself.

Example: Sorting

Given a list `l` of n numbers, a basic task is sorting the list.

- `[1,2,5.5,2] -> [1,2,3,5.5]`

We will see a fundamental recursive sorting algorithm called *merge sort*.

The idea of merge sort is the following: divide the list in half, sort each half separately, then merge the two sorted lists together.

- `[1,6,5,3,7,2,8,4] -> [1,6,5,3] [7,2,8,4] -> [1,3,5,6] [2,4,7,8]`

In [65]:

```
# merges two sorted lists
def merge(l_left,l_right):
    l = []
    i_left = 0
    i_right = 0

    # Repeatedly add the min of l_left[i_left] and l_right[i_right]
    # to l
    while i_left < len(l_left) and i_right < len(l_right):
        # Choose the minimum of l_left[i_left] and l_right[i_right]
        if l_left[i_left] <= l_right[i_right]:
            l.append(l_left[i_left])
            i_left += 1
        else:
            l.append(l_right[i_right])
            i_right += 1

    # Add remaining list to end of l
    while i_left < len(l_left):
        l.append(l_left[i_left])
        i_left += 1
    while i_right < len(l_right):
        l.append(l_right[i_right])
        i_right += 1
    return l
```

Now, use recursion to sort the left and right sides.

Running time of merge sort

The running time $T(n)$ of `mergesort` on a list of size `n` is found from the following:

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ 2T(n/2) + n & \text{if } n > 1 \end{cases}$$

The $2T(n/2)$ comes from sorting the left and right sides. The n amount of additional work comes from merging the left and right sides back together. To try to put $T(n)$ is closed form, let's try unrolling the recursion by replacing $T(n/2)$ with its recursive definition.

$$\begin{aligned} T(n) &= 2T(n/2) + n \\ &= 2(2T(n/4) + n/2) + n \\ &= 4T(n/4) + 2n \end{aligned}$$

Let's unroll one more time, by replacing $T(n/4)$.

$$\begin{aligned} T(n) &= 4(2T(n/8) + n/4) + 2n \\ &= 2^3 T(n/2^3) + 3n \end{aligned}$$

We will have to keep unrolling into we get to $T(1)$. This means we will need to unroll t times, where $n/2^t \approx 1$, which implies that $t \approx \log_2 n$. Thus,

$$
\begin{aligned}
T(n) &\approx 2^{\log_2 n} T(1) + \log_2(n) n \\
&= n + n \log_2 n \\
&= O(n \log n)
\end{aligned}
$$

In *general*, this is the best possible running time for a sorting algorithm.

Example: binary search

Implement a recursive version of binary search that takes a sorted list `l` of n numbers and a number `x` and returns True if `x` is in `l`. What is the running time of your algorithm?

In [46]:

```
# Option 1: Recursive approach
def binarySearchRecursive(l,x,low,high):
    if low > high:
        return False
    mid = (low + high) / 2
    if l[mid] < x:
        return binarySearchRecursive(l,x,mid+1,high)
    elif l[mid] > x:
        return binarySearchRecursive(l,x,low,mid-1)
    else:
        return True

# Option 2: Iterative approach
def binarySearchIterative(l,x):
    low = 0
    high = len(l) - 1
    while low <= high:
        mid = (low + high) / 2 # Rounds down to nearest integer
        if l[mid] < x:
            low = mid + 1
        elif l[mid] > x:
            high = mid - 1
        else:
            return True
    return False
```

Running time of binary search

Let $T(n)$ be the running time of binary search.

$$
\begin{aligned}
T(n) &= T(n/2) + 1 \\
&= T(n/2^2) + 2 \\
&= T(n/2^3) + 3 \\
&\vdots \\
&= O(\log_2(n))
\end{aligned}
$$

Example: Search in rotated list

Suppose we had a sorted list of unique numbers, but has been rotated an unknown amount.

- Example: `[1,3,4,6,8] -> [6,8,1,3,4]` Write an algorithm that finds an element `x` in the list. What is the running time of your algorithm?

We could check the left, right and midpoint values in the list.

- [2,3,4,0,1]
- [3,4,0,1,2]

It must be the case that the breakpoint occurs in either the left or right side. Thus, one side of the list must be sorted.

Can we determine which side is sorted just by looking at the left, middle, and right values?

If the middle value is strictly greater than the left side, then the breakpoint must occur in the right side.

If the middle value is strictly less than the left side, then the breakpoint must occur in the right side.

In [64]:

```
def search(l,x,left,right):
    mid = (left + right) / 2

    # Base cases
    if l[mid] == x:
        return True
    if right < left:
        return False

    # Recursive case
    if l[left] < l[mid]:
        # The breakpoint occurs on the right side.
        if x >= l[left] and x < l[mid]:
            return search(l,x,left,mid-1) # Search the left side
        else:
            return search(l,x,mid+1,right)
    else:
        # The breakpoint occurs on the left side
        if x > l[mid] and x <= l[right]:
            return search(l,x,mid+1,right)
        else:
            return search(l,x,left,mid-1)
```

By same reasoning as before, the running time is $O(\log n)$.

Dynamic programming

Introduction to Dynamic programming

Recall that the fibonacci sequence is defined as `f(1),f(2) = 1` and `f(n) = f(n-1) + f(n-2)` otherwise. Then, the fibonacci sequence `f` can be implemented as follows.

In [41]:

```
def f(n):
    if n == 1 or n == 2:
        return 1
    else:
        return f(n-1) + f(n-2)
```

The running time of this function can be shown to be $O(2^n)$.

We observe that the recursion recomputes the same subproblems! For example, the subproblems for $f(n)$ (which are $f(n-1)$ and $f(n-2)$) and the subproblems for $f(n-1)$ (which are $f(n-2)$ and $f(n-3)$) overlap at $f(n-2)$.

To improve the speed of our algorithm, we save the results of the overlapping subproblems, so we only have to compute each of them once. This is referred to as *dynamic programming*.

Specifically, we can create a list l of length n, where $l[i]$ stores the value of $f(i)$. We then start at $l[1]$ and fill in the list for increasing values of i. This process of storing the results of the subproblems in some data structure is called *memoization*!

In [67]:
```
def f(n):
    l = [0,1,1]
    for i in xrange(3,n+1):
        l.append(l[i-1]+l[i-2])
    return l[n]
```

The running time of the dynamic programming approach is the time required to populate the array, which is $O(n)$.

Example: Running up stairs

A child is running up a staircase containing n steps. The child can hop either 1 step, 2 steps, or 3 steps at a time. Write an algorithm `steps(n)` which returns the number of different ways the child can get up the staircase. What is its running time?

We could write `steps(n)` recursively.

$$\text{steps}(n) = \begin{cases} 1 & \text{if } n = 1 \\ 2 & \text{if } n = 2 \\ 4 & \text{if } n = 3, (1+1+1,1+2,2+1 \\ \text{steps}(n-1) + \text{steps}(n-2) + \text{steps}(n-3) & \text{if } n \geq 4 \end{cases}$$

How do we solve this with dynamic programming?

In [85]:
```
# Array of length n
def steps(n):
    l = [1,2,4]
    for i in xrange(3,n):
        l.append(l[i-1]+l[i-2]+l[i-3])
    return l[n-1]
```

Running time is the time required to populate a $O(n)$-length array, where each element

requires $O(1)$ time to compute. Thus, the running time is $O(n)$.

Example question

Implement a function `maxSubset(l)` that takes in a list `l` of real numbers (some of which may be negative) and finds the continguous subset with the greatest sum.

For example:

- `maxSubset([1,2,-1])` should return 3.
- `maxSubset([2,-1,2])` should return 3.

What is the running time of the algorithm?

Could compute the sum of every subset from index i to index j as follows:

$$\text{SubsetSum}(l, i, j) = \begin{cases} 0 & \text{if } j < i \\ l[i] + \text{SubsetSum}(i+1, j) & \text{otherwise} \end{cases}$$

The running time would be exponential without dynamic programming.

We could use dynamic programming as follows:

In [105]:
```
def maxSubset(l):
    SubsetSum = {}

    # Compute every subset of size 1
    for i in range(len(l)):
        SubsetSum[(i,i)] = l[i]

    # Compute every subset of size 2, 3, 4,...
    for k in xrange(1,len(l)):
        for i in xrange(0,len(l)-k):
            SubsetSum[(i,i+k)] = l[i] + SubsetSum[(i+1,i+k)]

    # Find the maximum subset over all the subsets
    max_subset = 0
    for i in xrange(0,len(l)):
        for j in xrange(i,len(l)):
            max_subset = max(max_subset, SubsetSum[(i,j)])
    return max_subset
```

Running time is $\sum_{i=1}^{n} \sum_{k=1}^{n-1} 1 = O(n^2)$.

Can we do better?

In order to do better than $O(n^2)$, we cannot look at each of the $\sum_{i=1}^{n} \sum_{j=i}^{n} 1 = O(n^2)$ subsets. We have to do something else!

What if we just considered the max subset that begins at each index.

Let `maxSubsetEndingAt(i)` denote the maximum subset that ends at `l[i]`.

$$\text{maxSubsetEndingAt}(i) = \begin{cases} \max\{l[0], 0\} & \text{if } i = 0 \\ \max\{l[i], \text{maxSubsetEndingAt}(i-1) + l[i]\} & \text{if } i \geq 1 \end{cases}$$

The maximum subset must end at some index i. So, we just need to check the value of `maxSubsetEndingAt(i)` for each `i` to solve the problem.

In [108]:
```
def maxSubset(l):
    maxSubsetEndingAt = [max(l[0],0)]
    for i in xrange(1,len(l)):
        maxSubsetEndingAt.append(max(l[i],maxSubsetEndingAt[i-1]+l[i]))

    max_subset = 0
    for i in range(len(l)):
        max_subset = max(max_subset, maxSubsetEndingAt[i])
    return max_subset
```

In [111]:
```
maxSubset([2,-8,3,-2,4,-10])
```

Out[111]:
```
5
```

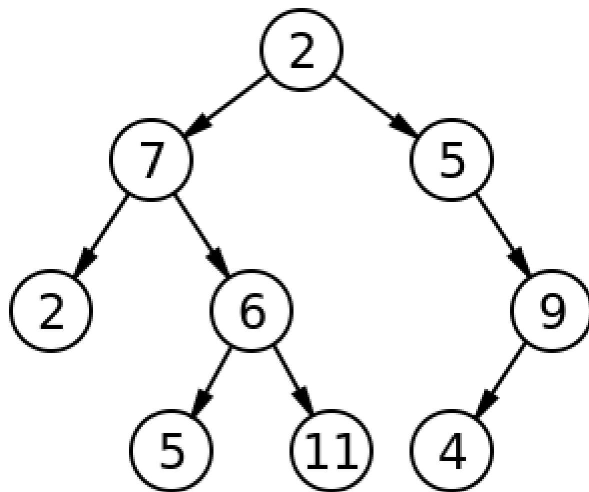
Binary trees

Introduction to binary trees

Questions involving the binary tree data structure are very popular in tech interviews, and can be challenging and varied!

A binary tree is a data structure consisting of a collection of nodes (starting at a root node), where each node consists of a value (`data`), together with a directed edges to at most two nodes (the "left child" and "right child"), with the additional conditions that no two edges point to the same node and no edge points to the root.

The following is a binary tree consisting of $n = 9$ nodes, where the root node has `data = 2`.



The tree with root 7 and the tree with root 5 are called the *left subtree* and *right subtree* of node 2.

The height of a tree is equal to the number of levels in the tree.

Implementing a binary tree

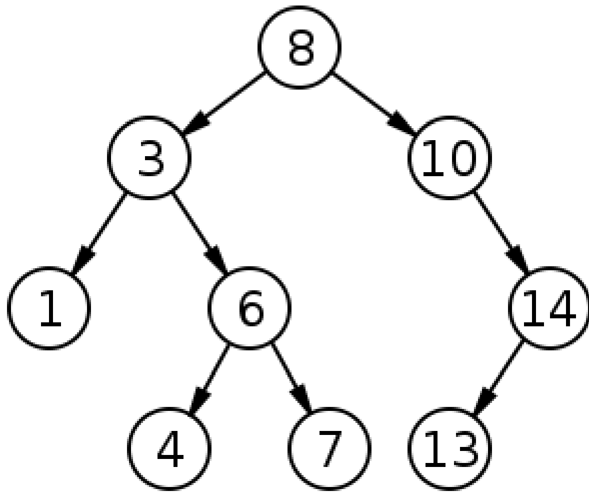
A binary tree is implemented similarly to a linked list. The main difference is that each node has two outgoing edges

In [42]:
```
class Node:
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None
```

Binary search tree

A binary tree is a binary search tree if it stores real numbers and the root satisfies the following properties:

- The root's value is larger than every node in the left subtree.
- The root's value is smaller than every node in the right subtree.
- The left and right subtrees are binary search trees



Example: finding closest data

Given the root of a binary search tree and a value `x`, implement an algorithm called `find_closest(root,x)`, which returns the closest point in the binary search tree to `x`. What is its running time?

In [3]:
```
def find_closest(root,x):
    if root == None:
        return None

    # The current closest to x is root.data
    closest = root.data

    # If x is larger than root.data, there might
    # be a closer node on the right subtree
    if x > closest and root.right != None:
        right_closest = find_closest(root.right)
        if abs(right_closest - x) < abs(closest - x):
            closest = right_closest

    # If x is larger than root.data, there might
    # be a closer node on the right subtree
    if x < closest and root.left != None:
        left_closest = find_closest(root.left)
        if abs(left_closest - x) < abs(closest - x):
            closest = left_closest

    return closest
```

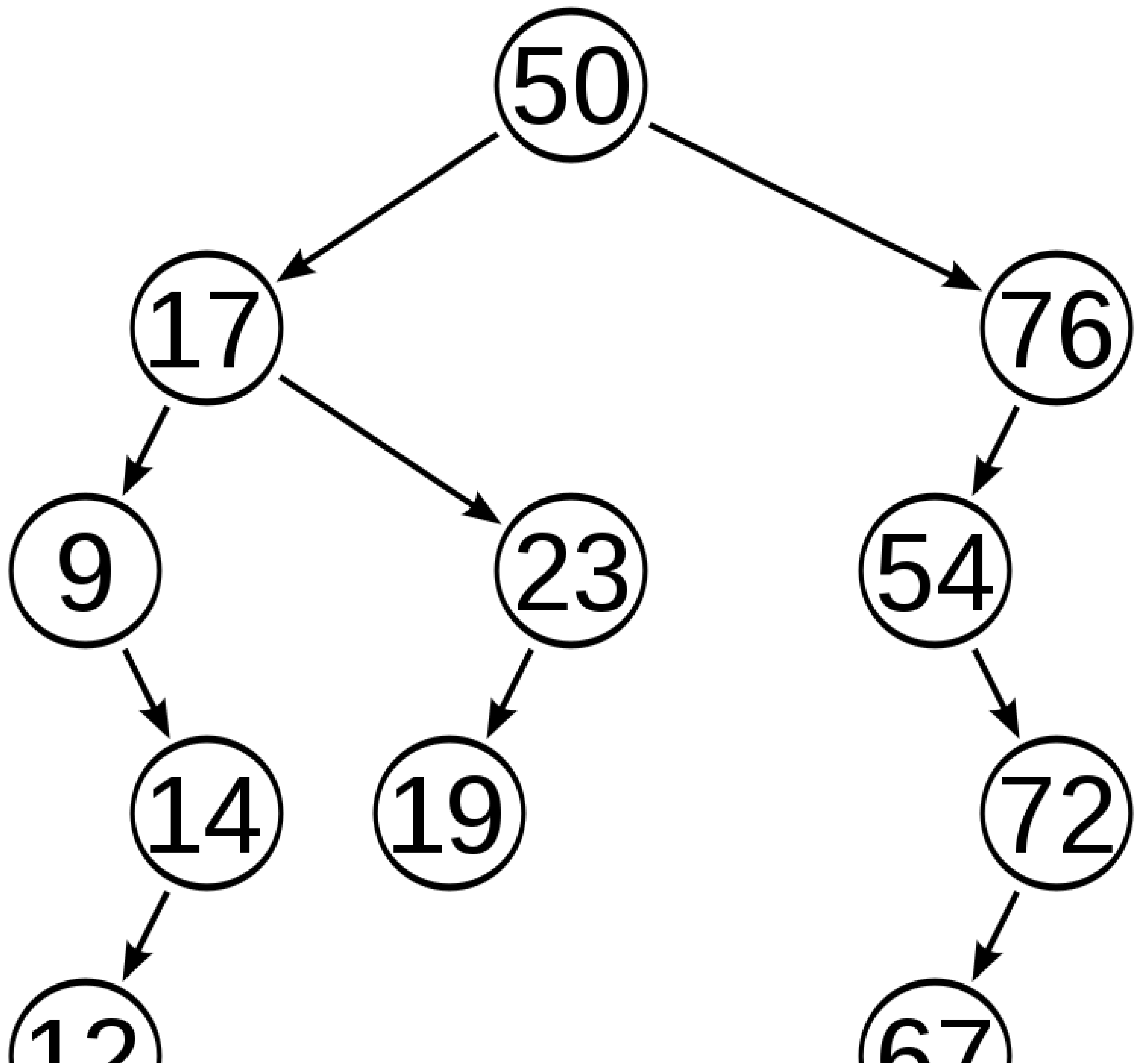
Suppose the tree has n nodes and $\ell \leq n$ levels.

In the worst case, `find_closest(root,x)` has to travel down ℓ levels.

Balanced binary search tree

A binary search tree is called *balanced* if, for each node in the tree, the heights of the left and right subtrees of that node differ by at most 1.

This is an unbalanced binary search tree.



50
17
76
9
23
54
14
19
72

A self-balanced binary search tree is a binary search tree that performs the following operations in a way that maintains the BST properties and keeps the tree balanced. Examples include AVL and Red-Black trees. It has the following operations.

- insert(root,data): Inserts a new node with value `data` into the binary search tree in time $O(\log n)$.
- remove(root,data): Remove a node with key `data` in time $O(\log n)$.
- search(root,data): Search for the key `data` in the binary search tree in time (?).

How long should we expect `search(root,data)` to take?

A balanced BST will keep the depth of the tree at $\ell = O(\log n)$. Thus, using the same searching algorithm as before, searching can be done in $O(\ell) = O(\log n)$.

Question: implementing a dictionary

Can a balanced binary search tree be used as a dictionary? If so, what are the pros and cons versus a hash table?

Yes, if the keys are numbers. Rather than the `root.data` being just a number, we can have `root.data = (key,value)`, where the `key` is what is used for the BST property, and the `value` is the information mapped to by the key.

Compared to hash tables, self-balanced binary search trees are slower, as insert, remove, and search require $O(\log n)$ time instead of $O(1)$ time.

However, binary search trees make it easy to iterate through all of the keys in the dictionary.

Also, given a `key`, we can find the *closest* entry in the dictionary if the dictionary is implemented as a BST, which we cannot do with a hash table.

Example: checking if a tree is a BST

Implement an algorithm `isBST(root)` that takes a node `root` and returns True if `root` is the root of a binary search tree (BST). What is the running time of the algorithm?

What information do we need in order to determine whether `root` is the root of a BST?

Need to know the following:

- Whether the left and right subtrees are BSTs.
- The maximum value in the left subtree.
- The minimum value in the right subtree.

If `root.data` is larger than the maximum value in the left subtree, then it is larger than every element in the left subtree.

If `root.data` is smaller than the minimum value in the right subtree, then it is larger than every element in the right subtree.

Can we formulate this information recursively?

$$\text{maxVal}(\text{root}) = \begin{cases} -\infty & \text{if root is} \\ \max\{\text{root. data}, \text{maxVal}(\text{root. left}), \text{maxVal}(\text{root. right}) & \text{otherwis} \end{cases}$$

$$\text{minVal}(\text{root}) = \begin{cases} +\infty & \text{if root is} \\ \min\{\text{root. data}, \text{maxVal}(\text{root. left}), \text{maxVal}(\text{root. right}) & \text{otherwis} \end{cases}$$

Then, `isBST(root) = True` if `root` is `None` or if:

- `isBST(root.left) = True`
- `isBST(root.right) = True`
- `root.data > maxVal(root.left)`
- `root.data < minVal(root.right)`

In [113]:

```
def maxVal(root):
    if root == None:
        return -1e9
    else:
        max_val = root.data
        max_val = max(max_val, maxVal(root.left))
        max_val = max(max_val, maxVal(root.right))
        return max_val

def minVal(root):
    if root == None:
        return 1e9
    else:
        min_val = root.data
        min_val = min(min_val, minVal(root.left))
        min_val = min(min_val, minVal(root.right))
        return min_val

def isBST(root):
    if root is None:
        return True
    if (isBST(root.left) and
        isBST(root.right) and
        maxVal(root.left) < root.data and
        minVal(root.right) > root.data):
        return True
    return False
```

What is the running time?

Assume that `root` is the root of a tree with n nodes. Then, the running times of `maxVal(root)` and `minVal(root)` are both $O(n)$, since we have to visit every node below the root.

Thus, the running time `T(root)` of `isBST(root)` is

$$T(\text{root}) = n + T(\text{root.left}) + T(\text{root.right})$$

Suppose all of the nodes are in the left subtree, meaning that the tree is basically a linked list. In that case,

$$\begin{aligned} T(\text{root}) &= n + T(\text{root.left}) \\ &= n + (n-1) + T(\text{root.left.left}) \\ &\vdots \\ &= \sum_{i=1}^{n} i \\ &= O(n^2) \end{aligned}$$

Use dynamic programming, by simply storing the values of `maxVal(root)`, `minVal(root)`, and `isBST(root)` in the node itself!

In [ ]:

```
def populateMaxVals(root):
    if root != None:
        populateMaxVals(root.left)
        populateMaxVals(root.right)
        root.max_val = root.data
        if root.left != None:
            root.max_val = max(root.max_val, root.left.max_val)
        if root.right != None:
            root.max_val = max(root.max_val, root.right.max_val)

def populateMinVals(root):
    if root != None:
        populateMaxVals(root.left)
        populateMaxVals(root.right)
        root.max_val = root.data
        if root.left != None:
            root.max_val = min(root.max_val, root.left.max_val)
        if root.right != None:
            root.max_val = min(root.max_val, root.right.max_val)

def populateIsBST(root):
    # Assume that the max and min values have already been populated
    if root != None:
        populateIsBST(root.left)
        populateIsBST(root.right)
        root.isBST = True
        if root.left != None:
            root.isBST = root.isBST and root.left.isBST
        if root.right != None:
            root.isBST = root.isBST and root.right.isBST

def isBST(root):
    if root is None:
        return True

    populateMaxVals(root)
    populateMinVals(root)
```

Example question

You are enjoying a walk in the fields and come across a leprechaun! The leprechaun has n pots in a line, where each pot contains some known number of gold coins. These numbers are stored in a list `pots`. For example, if `pots = [1,6,0,0,3,100,3]`, then the leftmost pot has `1` gold coin and the rightmost pot has `3` gold coins.

The leprechaun proposes the following game. You can select to take the leftmost or rightmost pot, and keep the gold coins inside of it. Afterwards, the leprechaun similarly chooses the leftmost or rightmost of the (remaining) pots. This repeats until there are no pots remaining.

You want to play optimally to maximize the sum of the gold pieces you obtain. We can assume that the leprechaun also wants to maximize their number of gold pieces. Given the pots `l`, write an algorithm `myMove(pots)` that returns whether we should select the left or right pot.

Example question 2

Suppose you are given an $m \times n$ array A, where the (i, j)-th element is obtained by `A[i][j]`. The array is binary, meaning that each `A[i][j]` either equals 0 or 1.

We are looking for the largest rectangle (by area) in the array formed (entirely) by 1's. For example, the largest rectangle in the following array has area 4.

$$\begin{matrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 \end{matrix}$$

The largest rectangle (by area) in the following array has area 3 (the rectangle must be filled in).

$$\begin{matrix} 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 0 \end{matrix}$$

Implement an algorithm `LargestArea(A,m,n)` that takes in an $m \times n$ binary array, and returns the largest rectangle inside of it.

Approaching the Interview

Strategies

- Simplest answer first
- What is the best conceiveable runtime?
- Simplify the problem
- See if any data structure might fit
- Can the problem be formulated recursively?
- Test cases to check correctness

Preparation

- Practice, practice, practice!
- Do problems from "Cracking the Coding Interview"
- Practice writing code on a whiteboard

The interviewer is more interested in seeing how you think, ask questions, and respond to hints that whether you get everything right

In the real world, they aren't going to know the answer either, so it's important to understand how you would behave in a team