```
template <class KeyType, class ValueType>
class Node { // variables could be public, or could add getters/setters
    int numberKeys;
    KeyType key[2]; // space for up to 2 keys
    ValueType value[2]; // space for the corresponding values
    Node subtree[3]; // pointers to the up to 3 subtrees
}
```

Inserting a new key:

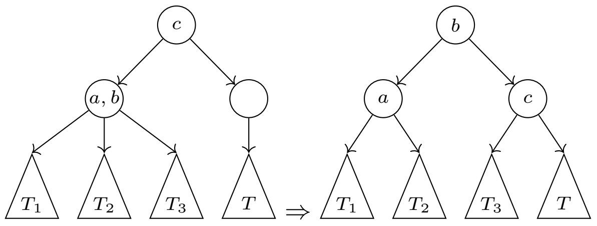
- Search for the key. If found, deal with duplicates. Otherwise, we've found the leaf.
- Insert the key into the leaf. If the leaf had only one key, we're done.
- Otherwise, fix the new leaf:
- To fix a node with 3 keys `[a b c]`
  1. If there is a parent, move `b` into it.
  2. If there is not a parent, make a new root for the tree, assign it height of `[a b c].height + 1`, move `b` into it.
  3. Create a node with just `a` as the key and make it the left child of the `b` node.
  4. Create a node with just `c` as the key and make it the left child of the `b` node.

Deleting a key:

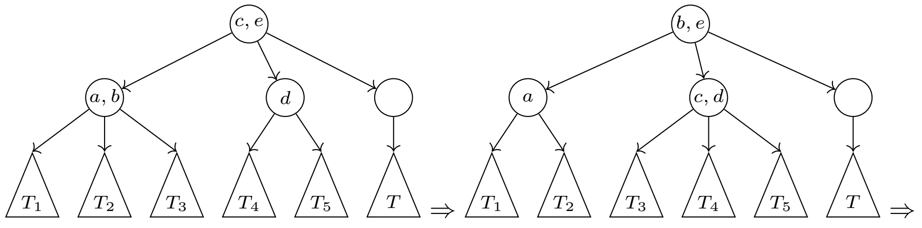
1. Search for the key (call this the `target`).
2. If the `target` is in a leaf, move to step 5.
3. If the `target` is not in a leaf, search for its `successor` in a leaf.
4. Swap the `target` key with its `successor` key. (Leave the heights in the original nodes).
5. Now that the `target` key is in a leaf, delete it.
6. Now that the `target` key has been deleted, call `fix_empty()` on the leaf.
7. If the leaf still has a key, `fix_empty()` will do nothing. Otherwise it will propagate the changes needed to fix the tree.

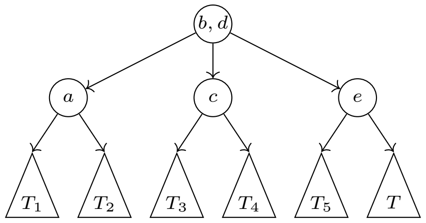
`fix_empty()`:

1. If the node still has a key, do nothing.
2. If the empty node has a sibling with two keys, perform a rotation in order to borrow a key from the sibling.

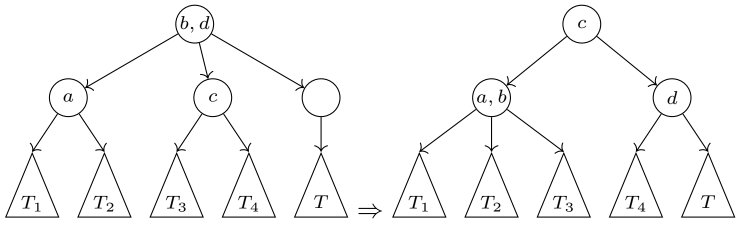


   In this instance don't adjust the heights.
3. If the empty node has two siblings and one has two keys, it's similar. Either the same thing can be done (if the sibling is adjacent) or, if it's two hops away:

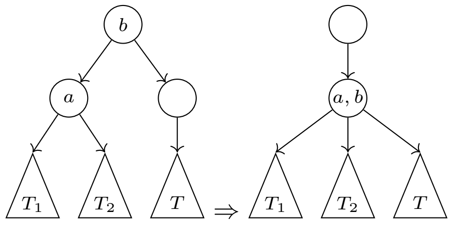




4. If no sibling of the empty node has two keys, then the next case is that the empty node has two siblings. In that case the empty node borrows from the parent:



5. Lastly, if the node has only one sibling with only one key, the parent is merged with the sibling into a node with two keys, and the empty is pushed up:



6. In each case, only swap key values. Keep heights the same.

0.1 Insertion Sort:

```
Insertion_Sort(A):
    for j = 2 to A.length:
        key = A[j]
        i = j - 1
        while i > 0 and A[i] > key:
            A[i + 1] = A[i]
            i = i - 1
        A[i + 1] = key
```

Loop Invariant: At the start of each iteration of the for loop of lines 1-8, the subarray `A[1..j-1]` consists of the elements originally in `A[1..j-1]`, but in sorted order.

Running Time: $O(n^2)$

0.2 MergeSort:

```
MergeSort(A):
    n = len(A)
    if n > 1:
        left = A[0..n/2]
        right = A[(n/2)+1..n-1]
        lsort = MergeSort(left)
        rsort = MergeSort(right)
        return Merge(lsort, rsort)
    return A

Merge(A,B):
    i = 0
    j = 0
    C = []
    A[len(A)] = infinity
    B[len(B)] = infinity
    while A[i] < infinity or B[j] < infinity:
        if B[j] < A[i]:
            C[i + j] = B[j]
            j += 1
        else:
            C[i + j] = A[i]
            i += 1
    return C
```

0.3 QuickSort:

```
QuickSort(A, p, r):
    if p < r:
        q = Partition(A, p, r)
        QuickSort(A, p, q - 1)
        Quicksort(A, q + 1, r)

Partition(A, p, r):
    x = A[r]
    i = p - 1
    for j = p to r - 1:
        if A[j] <= x:
            i += 1
            swap(A[i], A[j])
    swap(A[i + 1], A[r])
    return i + 1
```

0.4 Heaps:

```
Build_Max_Heap(A):
    A.heap-size = A.length
    for i = A.length/2 downto 1:
        Max_Heapify(A,i)

Max_Heapify(A, i)
    l = Left(i)
    r = Right(i)
    if l <= A.heap-size and A[l] > A[i]:
        largest = l
    else:
        largest = i
    if r <= A.heap-size and A[r] > A[largest]:
        largest = r
    if largest != i:
        swap(A[i], A[largest])
        Max_Heapify(A, largest)

Parent(i):
    return i/2

Left(i):
    return 2i

Right(i):
    return 2i + 1

Insert(A, k):
    A.heap-size += 1
    A[A.heap-size] = -infinity
    Heap_Increase_Key(A, A.heap-size, key)

Heap_Increase_Key(A, i, key):
    A[i] = key
    while i > 1 and A[Parent(i)] < A[i]:
        swap(A[i], A[Parent(i)])
        i = Parent(i)

Extract_Max(A):
    max = A[1]
    A[1] = A[A.heap-size]
    A.heap-size = A.heap-size -1
    Max-Heapify(A,1)
    return max
```

0.5 BFS:

```
BFS(G,s):
    Q = queue()
    dist = [infinity]
    pred = [null]
    color = [white]
    color[s] = gray
    dist[s] = 0
    Q.enqueue(s)
    while !Q.is_empty():
        u = Q.dequeue()
        color[u] = black
        for v in N(u):
            if color[v] == white:
                color[v] = gray
                pred[v] = u
                dist[v] = dist[u] + 1
                Q.enqueue(v)
```

BFS Theorem 1: When `BFS(s)` terminates, $\forall v \in V$ `dist[v]` is the shortest path distance from s to v. Furthermore: `v, pred[v], pred[pred[v]],...,s` is such a shortest path in reverse.

BFS Theorem 2: If vertices s and t are connected, when `BFS(s)` terminates, the color of t is black. If s and t are not connected, when `BFS(s)` terminates, the color of t is white.

BFS Theorem 3: (main loop invariant) At any stage of the algorithm `BFS(s)` at step 4, the queue Q contains vertices in at most two adjacent levels l and $l+1$.

1. All vertices in level $< l$ are black
2. All vertices in level l that are not gray are black
3. All vertices in level $l+1$ that are not gray are white
4. All vertices in level $> l+1$ are white
5. In Q, all vertices in level l are ahead of those in level $l+1$

BFS Theorem 4: When `BFS(s)` terminates, $\forall v$, `dist[v]` $= \delta_s(v)$.

0.6 Djikstra:

```
Djikstra(G,s):
    dist = [infinity]
    pred = [null]
    S = []
    dist[s] = 0
    Q = priority_queue(all vertices)
    while !Q.is_empty():
        u = Q.extract_min()
        S.append(u)
        for v in N(u):
            Relax(u,v)
            Q.decrease_key(v)

Relax(u,v):
    if dist[v] > dist[u] + w(u,v):
        dist[v] = dist[u] + w(u,v)
        pred[v] = u
```

Running Time: $O((m+n)\log n)$

Theorem: When Djikstra's algorithm terminates, $\forall v$, `dist[v]` $= \delta_s(v)$

0.7 Bellman-Ford:

```
Bellman-Ford(G, s):
    dist = [infinity]
    pred = [null]
    dist[s] = 0
    for u in G.V:
        for v in N(u):
            Relax(u,v)
```

Running time: $O(mn)$

0.8 DFS:

```
DFS(G):
    color = [white]
    pred = [null]
    disc = [null]
    finish = [null]
    time = 0
    for u in G.V:
        DFS_Visit(u)

DFS_Visit(u):
    color[u] = gray
    disc[u] = time
    time += 1
    for v in N(V):
        if color[v] == white:
            pred[v] = s
            DFS_Visit(v)
    color[u] = black
    finish[u] = time
    time += 1
```

Running Time: $O(m + n)$

Parentheses Theorem: In any DFS forest, for vertices u, v either:

1. $[d[u], f[u]]$ is disjoint from $[d[v], f[v]]$ and neither is the descendant of the other or
2. $[d[u], f[u]]$ contains $[d[v], f[v]]$ and v is a descendant of u

White Path Theorem: v is a descendant of u $\iff$ there is a white path from u to v when `DFS_Visit(u)` is called.

0.9 Topological Sort:

Theorem: Perform `DFS(G)`:

(1) G is acyclic $\iff$ there is no back edge

(2) If G is acyclic, then decreasing order of finish times is a topological order

1 Question Bank Stuff:

Question: Given an integer x and an array of numbers, find if two numbers in the array sum up to x. (Or, find the pair of numbers whose sum is closest to x.)

```
two_sum(A, x):
    QuickSort(A)
    i = 0
    j = len(A) - 1
    guess = A[i] + A[j]
    dist = abs(x - guess)
    while i < j and guess != x:
        if guess < x:
            i += 1
        else:
            j -= 1
        new_guess = A[i] + A[j]
        if dist > abs(x - new_guess)
        guess = new_guess
        dist = abs(x - guess)
    return guess
```

Question: Given two sorted arrays, find the median of the union of elements. - or - Given two sorted arrays, find the kth smallest element in the union of elements.

```
find_kth_smallest(A, B, k):
    n = len(A)
    m = len(B)
    if A[k] < B[0]:
        return A[k]
    if B[k] < A[0]:
        return B[k]
    if k < (n+m)/2:
        if A[n/2] < B[m/2]:
            return find_kth_smallest(A, B[0..m/2], k)
        else:
            return find_kth_smallest(A[0..n/2], B, k)
    else:
        if A[n/2] < B[m/2]:
            return find_kth_smallest(A[(n/2)+1..n-1], B, k-(n/2)])
        else:
            return find_kth_smallest(A, B[(m/2)+1..m-1], k-(m/2))
```

Question: You have an unsorted array of integers where every integer appears exactly twice, except one integer that appears once. Find it.

Answer: Option A is build a hash table, insert every number the first time, delete it the second time, at the end the only number remaining is the one that appears only once. Takes $O(n)$ space and time. Option two is sort the list and iterate through it. $O(1)$ space and $O(n \log n)$ time.

Question: Given an array of integers, move all the zeroes to the right end. The order of the others doesn't matter. Use $O(1)$ extra storage.

Answer: Just use `Partition()` and modify it to sort zeros to the right.

```
maintain_median():
    big = min_heap()
    small = max_heap()

    while stream exists:
        a = stream.get_int()
        if big.heap_size < 1:
            big.insert(a)
        else:
            if a < big.find_min():
                small.insert(a)
            else:
                big.insert(a)
            if big.heap_size + 1 > small.heap_size:
                b = big.extract_min()
                small.insert(b)

return_median():
    if big.heap_size > small.heap_size:
        return big.find_min()
    elif small.heap_size > big.heap_size:
        return small.find_max()
    else:
        return (big.find_min() + small.find_max())/2
```

Question: Given an array of integers containing all the elements from $1, 2, \ldots, n$ except one of them. Find which one. Try doing it using $O(1)$ memory. What if two elements were missing?

$$\sum_{0}^{n} i = \frac{n(n-1)}{2}$$

```
find_missing{A}:
    k = 0
    for num in A:
        k += num
    actual = (len(A) * (len(A) - 1))/2
    return actual - k
```

If two elements are missing just sort the array and iterate through it.

Question: Given array of integers, determine if it contains a Pythagorean triple (integers a, b, c such that $c^2 = a^2 + b^2$).

Answer: Sort the array. Put a pointer at each end. Those point to a and c so just binary search for b. Adjust pointers as necessary.

Question: The input is a sequence of ranges $[x1, y1], [x2, y2], \ldots$ (where each $x_i \leq y_i$. Given two ranges that intersect, merge then into a single range by taking the union of ranges. Keep doing this until you finally get disjoint ranges. Design an algorithm that determines this output.

Question: Given three arrays in non-decreasing order, find the elements that are common to all of them. Try $O(1)$ extra storage.

Answer: Use 3 pointers. They're in sorted order so just iterate all of them.

Question: Consider two sorted arrays that are identical, except one array has an extra element. Find that element and its index.

Answer: Stick a pointer on each array. Increment them both until they don't match. The smaller element is the missing one.

Question: Given a sorted array, find the closest element (in terms of absolute difference) to a given number.

Answer: Binary search.

Question: Given an array, a local maximum is an element $A[i]$ such that is greater than or equal to the neighboring elements in the array. (There can be two neighboring elements, or just one if $i = 0$ or i is the last index.) Find a local maximum.

Answer: Binary search. Go towards the higher number.

Question: Find if a singly linked list has a loop. Use $O(1)$ extra memory.

Answer: Use a tortoise and a hare. If they ever cross there's a loop.

Question: Going beyond the problem above, find the first node on the loop (assume there is a single loop). Use $O(1)$ extra memory.

Answer: Use the answer above. Then use an additional tortoise/hare pair. Alternate which pointer gets sent from where.

Question: Given a sorted array A of distinct integers, determine if there is an index i such that $A[i] = i$.

Answer: Iterate through the array and return true if $A[i] = i$ otherwise return false.

Question: There is an extremely large array A such that the first n entries are positive integers, and all remaining entries are 0. Find n.

Answer: Use binary search. If the mid is a 0 then go left, if the mid is a number go right.

Question: Consider an $n \chi n$ 2D array where all rows and columns are in sorted order. Equivalently, the following holds for any i, j: for $i_0 > i$, then $A[i_0][j] > A[i][j]$, and for $j_0 > j$, $A[i][j_0] > A[i][j]$. Given an x, find if x is in the array.

Answer: Binary search.

Question: Describe a stack based data structure that supports push, pop, and findmin in $O(1)$ operations.

Answer: Use two stacks. The first is just the stack. The second is a stack of pointers to the min elements of the first.

Question: Given the pre-order traversal of a binary search tree, reconstruct the tree.

Answer: Just insert in order.

Question: A pair of nodes x, y in a (supposed) binary search tree violate the BST property if x is an ancestor of y, and the corresponding values are "out of order". Given a BST, find the number of pairs that violate the BST property.

```
def find_pairs(node, path):
    count = 0
    for i in range(len(path)):
        ancestor = path[i][0]
        direction = path[i][1]
        if direction == 'right' and node.key < ancestor.key:
            count++
        if direction == 'left' and node.key > ancestor.key:
            count++
    return count + find_pairs(node.left , path.append([node, 'left'])) +
        find_pairs(node.right , path.append([node, 'right']))
```

Question: A cycle is a sequence of vertices $v_1, v_2, \ldots, v_k$ such that for each i, (v_i, v_{i+1}) is an edge, and (v_k, v_0) is an edge. Determine if G has a cycle containing edge e. Try using both BFS and DFS.

Answer: Run BFS but add a check such that when we are looking at v_0, we ignore v_0. When BFS exits, if $dist[v_0] \neq \infty$ then e is contained in a cycle.

Question: Determine if G contains any cycle. (Clearly, you can use the solution for the previous problem, but there's a more efficient method.)

Answer: Run DFS. If it ever hits a gray vertex, return true. If it finishes without hitting a gray node, return false.

Question: A directed graph is called a DAG if it contains no directed cycles. Determine if a graph is a DAG.

Answer: Run DFS on the whole graph. If it ever hits a gray vertex there is a a cycle so it is not a DAG.

Question: Prove that a graph G is bipartite iff it has no cycles of odd length. (Hint: follow the progress of BFS on G

Answer: Run BFS. The source is set to color A and its neighborhood is set to color B, then the colors switch. If BFS ever discovers a vertex u with an edge to a vertex v that has the same color then it has a cycle of odd length. If it does not, then the graph can be colored fully with only two colors, meaning it can be split into two subsets with no edges within the same subset, which is the definition of bipartite.

Question: A vertex v is a cut or articulation vertex if its removal disconnects two other vertices. Equivalently, all paths (in G, before removal) from u to u_0 pass through v. Determine if v is a cut vertex. If so, find some pair (u, u_0) that got disconnected.

Answer: Color v black. Run `DFS_Visit()` on any neighbor. When it finishes, if it has any white neighbors, it is a cut vertex by the white path theorem.

Question: The proof of Dijkstra's algorithm fails when there are negative edges. Why?

Answer: Consider the graph G containing the source vertex s and vertices u, v such that v is dequeued before u is. If there exists an edge u, v with negative weight such that the actual shortest path from s to v goes through this edge, then the shortest path to anything that goes through v will not be adjusted properly, because v's neighborhood is only relaxed when it is dequeued.

Question: We have three containers, whose sizes are 10 pints, 7 pints, and 4 pints. The 7-pint and 4-pint containers are full of water, while the 10-pint container is empty. We are only allowed one operation: pouring the contents on one container into another, stopping only when the source container is empty, or the destination container is empty. Determine if there is a sequence of pourings that leave exactly 2 pints in the 7- or 4- pint container. Model this as a graph problem. Trying solving the general version where there are three containers of integer capacities of a maximum of n. The starting configuration is provided, and we have some desired "end state". Determine if the "end state" can be achieved. Work out the running time.

Construct a list of `operations[]` for the set of possible operations (pouring from one bucket to another).

Construct a function `valid()` to return whether, given a bucket state s and an operation $op()$, $op(s)$ results in a valid state.

Build the graph as a hash table G.

Build the adjacency lists as hash tables.

Use a recursive function to explore the possible vertices of the graph. If it finds the desired end state, return true, otherwise return false.

```
Find_State(start, end, operations):
    G = hash_table()

    found = Pour(start, end, operations, G):
    return found

Pour(start, end, operations, G):
    for op in operations:
        x = op(start)
        found = false
        if valid(x):
            if x == end:
                found = true
            else:
                G.insert(x)
                G.find(start).insert(x)
                G.find(x).insert(start)
                found = Pour(x, end, operations, G)
            if found == true:
                return found
        return found
```

Question: Given a directed (unweighted) graph G, the reverse is obtained by simply reversing all edges. Assume G is represented as an adjacency list of out neighbors. Construct the reverse of G.

```
reverse(G):
    reversed = []

    for u in G.V:
        for v in N(u):
            reversed[v].push(u)

    return reversed
```

Question: Consider a game of snakes and ladders. Determine the minimum number of throws required to win the game.

```
snakes_and_ladders(G):
    squares = []
    for v in G.V:
        if v has a chute or a ladder:
            squares[v] = chute/ladder destination
        else:
            squares[v] = v
    for u in G.V:
        for v in range(u,min(u+6,n-1)):
            N(u).push(squares[v])

    BFS(G,0)

    min_throws=dist[n-1]
    return min_throws
```

Question: Biologists often construct a food network of species in an ecosystem. The vertices represent species, and a directed edge (u, v) means species u eats species v. An apex species is one that is not eaten by another species. Suppose you are give a list of all possible "eating" relationships (so a list of "u eats v"). Determine all apex species.

```
Find_Apex(G):
    color = [white]
    pred = [null]
    for s in G.V:
        DFS_Visit(s)

    apex = []
    for s in G.V:
        if pred[s] == null:
            apex.append(s)
    return apex

DFS_Visit(u):
    color[u] = gray
    for v in N(u):
        pred[v] = u
        if color[u] == white:
            DFS_Visit(v)
    color[u] = black
```

Question: In a food network (described above), determine the effect of the extinction of a species. If a species v goes extinct, and there is some other species u that only eats v, then u will become extinct. This can cascade through the food network. Given a food network and a species v, determine all other species that will become extinct if v goes extinct.

```
Extinction(G, v):
    color = [white]
    pred = [null]
    for u in G.V:
        DFS_Visit(u)
    extinct = Kill(v)

DFS_Visit(u):
    color[u] = gray
    for v in N(u):
        pred[v].insert(u)
        if color[v] == white:
            DFS_Visit(v)
    color[u] = black

Kill(v):
    extinct = [v]
    for u in pred[v]:
        G.adj[u].delete(v)
        if len(N(u)) < 1:
            extinct.append(Kill(u))
    return extinct
```

Question: Given a weighted, directed graph G, determine the length of the shortest (directed) cycle in G. (Hint: you can find this using a number of calls to Djikstra, but can you do only $O(n)$ calls?)

```
Shortest_Cycle(G):
    shortest = infinity
    for s in G.V:
        shortest = min(shortest, Djikstra(G, s))
    return shortest

Djikstra(G, s):
    dist = [infinity]
    pred = [null]
    done = []
    min_cycle = infinity
    Q = pq(all v in V)
    while !Q.is_empty():
        u = Q.extract_min()
        done.append(u)
        for v in N(u):
            if v == s:
                if dist[u] + w(u,v) < min_cycle:
                    min_cycle = dist[u] + w(u,v)
            else:
                Relax(u,v)
                Q.decrease_key(v)
    return min_cycle
```

Question: Given a simple graph G, determine a shortest cycle in G. (Hint: it's a little trickier than above if you want $O(n)$ BFS calls. You need to dig into BFS.)

```
find_cycle(G):
    cycles = [infinity]
    for s in G.V:
        cycles = BFS(G,s)
    return min(cycles)

BFS(G,s):
    min_cycle = infinity
    Q = queue()
    dist = [infinity]
    pred = [null]
    color = [white]
    color[s] = gray
    dist[s] = 0
```

```
Q.enqueue(s)
while !Q.is_empty():
    u = Q.dequeue()
    color[u] = black
    for v in N(u):
        if color[v] == white:
            color[v] = gray
            pred[v] = u
            dist[v] = dist[u] + 1
            Q.enqueue(v)
        if v == s and pred[u] != s and dist[u] + 1 < min_cycle:
            min_cycle = dist[u] + 1
return min_cycle
```

Question: Consider a road network, where each vertex in a location, and each edge (u, v) (directed, weighted) is the distance of getting from u to v. You have to send supplies from vertex s to t, obviously using the shortest path. That's easy, just use Djikstra. The catch is that your truck actually doesn't have the supplies and has to pick it up from a warehouse. There is a set of vertices W that correspond to the warehouses, and you truck can go to any (just one) of the warehouses. Find the shortest route for your truck, with this additional constraint. The input to your algorithms is G, s, t, and W.

Answer: Run Dijkstra's from the source and get the distance to the warehouses. Now reverse the graph and run Dijkstra's from the target to get the distance to the warehouses. Track the minimum distance. $O((m + n) \log n)$

Question: A directed graph G is semi-connected if for every pair u, v of vertices, there is either a path from u to v or from v to u. Determine if G is semi-connected.

Answer: The only thing I can think of is brute force using DFS.

Question: You are given the dependency structure of coursework at a university. Think of the courses are vertices, and you can given a (directed) edge (u, v) if you must finish course u before course v. Assuming a student can take as many courses as she wishes in a quarter, determine the minimum number of quarters required to finish all courses.

Answer: Run DFS. Modify `DFS_Visit()` to return `quarters` equal to the max of 1 or the largest value of the neighborhood. Do a linear scan through all vertices and return the largest value.

Question: At some point in the DFS algorithm, suppose all vertices become gray. What does that imply about the graph? Specifically, can you determine what the DFS forest will look like?

Answer: The forest is a linked list that may or may not have additional edges. There is one recursive stack containing all vertices.

Question: Determine arbitrage in a currency exchange network. Basically, for every pair of currencies (u, v), you have the exchange rate: given 1 unit of u, how many units of v you can buy with it. Arbitrage occurs when you start with one unit of u_1, convert that to u_2, convert that to $u_3, \ldots,$ convert that to u_k, and then back to u_1, and end up with ¿ 1 unit of u_1.

Answer: Bellman-Ford. The distance calculation is multiplicative. Check if the distance to the source ever goes over 1.

Other Stuff:

$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}, \sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}, \sum_{i=1}^{n} i^3 = \frac{n^2(n+1)^2}{4}$

$\sum_{i=0}^{n} c^i = \frac{c^{n+1}-1}{c-1}, c \neq 1; \sum_{i=0}^{\infty} c^i = \frac{1}{1-c}$

$y = \log_b x \rightarrow x = b^y$

$\log_b b = 1$

$\log_b b^x = x$

$b^{\log_b x} = x$

$\log_b(x^r) = r \log_b x$

$\log_b(xy) = \log_b x + \log_b y$

$\log_b \left(\frac{x}{y}\right) = \log_b x - \log_b y$

$a^n a^m = a^{n+m}$

$(a^n)^m = a^{nm}$

$\frac{d}{dx}(a^x) = \ln a$

$(fg)' = f'g + fg'$

$\left(\frac{f}{g}\right)' = \frac{f'g - fg'}{g^2}$

$\frac{d}{dx}(f(g(x))) = f'(g(x))g'(x)$

$\frac{d}{dx}(e^{g(x)}) = g'(x)e^{g(x)}$

$\frac{d}{dx}(\ln g(x)) = \frac{g'(x)}{g(x)}$