UC Berkeley – Computer Science
CS61B: Data Structures

Final, Spring 2018

This test has 12 questions worth a total of 400 points and is to be completed in 170 minutes. There is also an additional 30 point question that is part of midterm 2. The exam is closed book, except that you are allowed to use three double sided written cheat sheets (can use front and back on all 3 sheets). No calculators or other electronic devices are permitted. Give your answers and show your work in the space provided. **Write the statement out below in the blank provided and sign. You may do this before the exam begins.**

"I have neither given nor received any assistance in the taking of this exam."

These solutions are probably correct. Let us know if you spot any errors.

Video walkthrough: https://www.youtube.com/watch?v=XJuPjSbexWM

Signature: ___________________________

#	Points	#	Points
0	1	7	32
1	44	8	24
2	33	9	24
3	46	10	40
4	0	11	55
5	13	12	51
6	37	13 (mt2)	30
		TOTAL	400 + 30

Name: ___________________________

SID: ___________________________

Three-letter Login ID: __________

Login of Person to Left: ________

Login of Person to Right: _______

Exam Room: ______________________

Tips:

- **Hint: Use Math.max instead of if statements to save space on coding problems.**
- There may be partial credit for incomplete answers.
- There are a lot of problems on this exam. **Work through the ones with which you are comfortable first. Do not get overly captivated by interesting design issues or complex corner cases you're not sure about.**
- Not all information provided in a problem may be useful, and **you may not need all lines**.
- Unless otherwise stated, all given code on this exam should compile. All code has been compiled and executed before printing, but in the unlikely event that we do happen to catch any bugs in the exam, we'll announce a fix. **Unless we specifically give you the option, the correct answer is not 'does not compile.'**
- ○ indicates that only one circle should be filled in.
- □ indicates that more than one box may be filled in.
- **For answers which involve filling in a ○ or □, please fill in the shape completely.**

Optional. Mark along the line to show your feelings on the spectrum between ☹ and ☺.

Before exam: [☹______________☺].
After exam: [☹______________☺].

0. So it begins (1 point). Write your name and ID on the front page. Write the exam room. Write the IDs of your neighbors. Write the given statement and sign. Write your login in the corner of every page. Enjoy your free point ☺.

1. Graph basics. a) **(12 points).** For the graph below, give the DFS preorder, DFS postorder, and BFS order using 61A as the source. Assume that DFS and BFS **do not restart** when they run out of vertices to process. You may not need all seven blanks. Assume that ties are broken in alphanumeric order (i.e. the edge 61A → 61B would be considered before 61A → 61C).

DFS Preorder	61A 61B 169 170 188 61C
DFS Postorder	169 170 188 61B 61C 61A
BFS Order	61A 61B 61C 169 170 188

b) **(4 points).** Give a topological sort for **all vertices** in the graph from part a.

70 61A 61C 61B 188 170 169 (but many other solutions are valid)

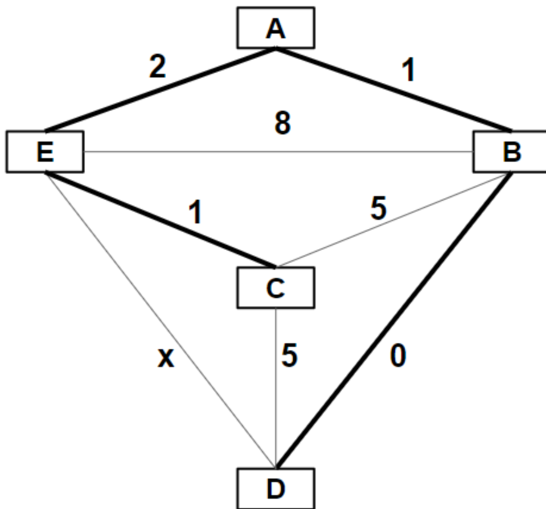
c) **(8 points).** For the graph below, give the MST in the order that the edges are added by Prim's algorithm starting at the top vertex (containing a circle), and the order added by Kruskal's algorithm. **Identify each edge with its weight**, e.g. 0 refers to the edge at the top right of the image. You may not need all blanks.

Edges in MST in order added by Prim's	0 -1000 1 2 3 4 5
Edges in MST in order added by Kruskal's	-1000 0 1 2 3 4 5

d) **(4 points).** For the graph from part c, do the edges of the MST **change** if we add 1000 to every edge weight? Do not worry about edge order.

○ Yes ● No

e) **(8 points).** For the graph below, let x be the unknown weight of edge DE. Can the marked edges (in bold) be a valid minimum spanning tree? If so, for what values of x? If not, why not?

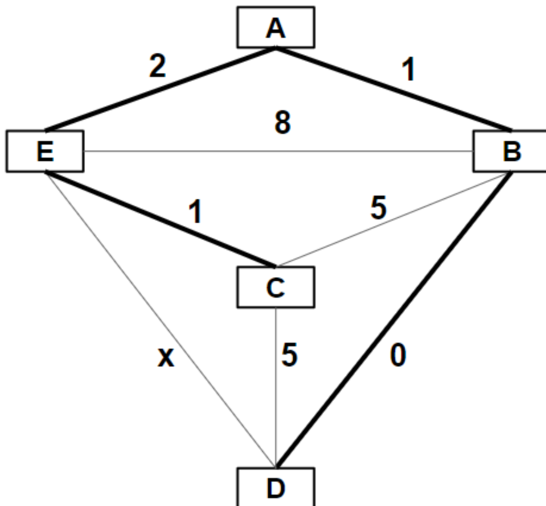


● Valid MST, range of valid x: x is ≥ 2

○ Never a valid MST, because: ____________________

____

f) **(8 points).** For the graph below (same as part e), can the marked edges (in bold) be a shortest paths tree rooted at B? If so, for what values of x? If not, why not?



● Valid SPT rooted in B, range of valid x: x is ≥ 3

○ Never a valid SPT rooted in B, because: ___________

____

2. **Sorting.**

a) **(8 points).** Suppose that we have an initial array of unsorted symbols: `[@, #, $, %, &, *]`, where **no two symbols are considered equal**. Suppose that we are in the middle of insertion sort and have reached the state where the array contains `[&, #, $, @, %, *]`.

Mark each of the following propositions as true, false, or not enough information (NEI).

○ T ○ F ● NEI	& is the smallest element of the array.
○ T ○ F ● NEI	* is the largest element of the array.
● T ○ F ○ NEI	& < #
○ T ● F ○ NEI	% < @

b) **(8 points).** Suppose we have the array [6, 1, 4, 7, 3, 2, 5, 8]. Suppose we use 6 as the pivot, and partition using Tony Hoare's partitioning scheme.

Give the array immediately after the first swap: 6 1 4 5 3 2 7 8

Give the array after the entire partition operation is complete: 2 1 4 5 3 6 7 8

c) **(6 points).** Given an array, suppose that x < y, and that x appears to the right of y. What happens to the **inversion count** if we swap x and y? Fill in completely all boxes that are possible.

■ It can decrease. ☐ It can increase. ☐ It can stay the same.

d) **(6 points).** What happens to the **inversion count** if we min-heapify an array using bottom up heapification? Fill in all that are possible.

■ It can decrease. ☐ It can increase. ■ It can stay the same.

This is just the same as part c, but the items can also be all duplicates in which case inversion count doesn't change.

e) **(5 points).** A 61B student studying for his final observes that the time for the `get` operation in a hash table is constant, so long as the items in a hash table are evenly distributed. Inspired, the student proposes a sort called `HashSort` that works very simply: the `HashSort` class contains a single instance variable `HashMap<int[], int[]>` that maps any integer array to a sorted version of that array. For example, `[8, 1, 21772, 8, 2]` would map to `[1, 2, 8, 8, 21772]`. When a user calls `HashSort(x)`, the method simply returns the result of calling `.get(x)` on its `HashMap` instance variable.

The student notes that this would take infinite memory to store all possible arrays, but claims that if you somehow had infinite memory, the algorithm would have constant runtime for an input array of length N. Is the student correct that `HashSort` is constant time? If yes, explain why the puppy, cat, dog bound does not apply. If not, explain why the student is wrong.

○ Yes / ● No. Explanation: Computing the hashcode takes time N.

3. Back to the First Half of the Semester.

a) **(16 points).** Given two `IntLists` defined as follows, fill in the append method below so that it **non-destructively** creates a new `IntList` with y appended to the end of x. For example, if x is 3 → 4 → 5, and y is 9 → 10 → 11, then the result should be 3 → 4 → 5 → 9 → 10 → 11. You may not need all lines.

```
public class IntList {
    public int first;
    public IntList rest;
    public IntList(int f, IntList r) { first = f; rest = r; }

    public static IntList append(IntList x, IntList y) {
        if (x == null) {
            return y;
        } else {
            return new IntList(x.first, append(x.rest, y));
            _____________________________________________________
        }
    }
}
```

b) **(16 points).** Suppose we modify our `IntList` class so that it can be iterated over. Fill in the code below so that iteration works correctly. You do not need to complete part a correctly to do this problem. You may not need all lines.

```
public class IntList implements Iterable<Integer> {
    … // same as in part a
    @Override
    public Iterator<Integer> iterator() {
        return new IntListIterator(this);
    }
    public static class IntListIterator implements Iterator<Integer> {
        private IntList p;
        public IntListIterator(IntList il) {
            p = il;
        }
        public boolean hasNext() {
            return p != null;
        }
        public Integer next() {
            int rval = p.first;
            p = p.rest;
            return rval;
        }
    }
}
```

c) **(6 points).** Suppose we have an `IntList` from part b called L. There are three ways to iterate over L, shown below. Assume that `print` is a method that simply calls `System.out.print`.

One.java	Two.java	Three.java

`for (int x : L) {`
`  print(x);`
`}`
	`Iterator<Integer> it`
	`  = L.iterator();`
	
	`while (it.hasNext()) {`
	`  print(it.next());`
	`}`
		`IntListIterator it =`
		`  (IntListIterator)`
		`  L.iterator();`
		
		`while (it.hasNext()) {`
		`  print(it.next());`
		`}`

Suppose we changed our `IntListIterator` **class** in `IntList` from **`public`** to **`private`**. For each of our four files, what happens? Fill in one bubble per line.

IntList.java: ○Fails to compile ● Compiles
One.java: ○Fails to compile ○ Compiles but works incorrectly ● Compiles and works correctly
Two.java: ○Fails to compile ○ Compiles but works incorrectly ● Compiles and works correctly
Three.java: ●Fails to compile ○ Compiles but works incorrectly ○ Compiles and works correctly

d) **(8 points).** Suppose we have a software system that makes a huge number of calls to a very large `ArrayList<Integer>`. We want to squeeze every last bit of performance out of the system. To accomplish this, an engineer suggests building and switching to our own special purpose `IntArrayList` class that is exactly the same as `ArrayList`, except that it will use an `int[]` to store the data. This is contrast to the `ArrayList<Integer>` class, which uses an `Integer[]` array.

In terms of practical runtime and space usage, would we expect our `IntArrayList` to perform better, almost exactly the same, or worse?

Runtime of `IntArrayList`: ● Better ○ No measurable difference ○ Worse

Brief justification for runtime: Autoboxing and autounboxing operations take a non-trivial amount of time.

Space usage of `IntArrayList`: ● Better ○ No measurable difference ○ Worse

Brief justification for space: Integer objects take much more space to store than ints.

4. PNH (0 points). When was the oldest surviving film recorded? Name something that happens in it.

1885. Someone spins around in a circle.

5. Hash Codes (13 points). Suppose we have the class defined below.

```
public class SneakySnake {
    public static final String UUIDString = "64de13c7fc79154d0570a12f268df";
    private int seed;
    public Random random;

    public SneakySnake(int s) { seed = s; random = new Random(seed); }

    @Override
    public boolean equals(Object other) {
        SneakySnake o = (SneakySnake) other;
        return this.UUIDString.equals(o.UUIDString) && this.seed == o.seed;
    }

    @Override
    public int hashCode() {
       ...
    }
}
```

For each of the `hashCode()` implementations below, fill in whether the implementation is valid, valid but slow, or invalid. A hashcode is valid if the methods of a hash table implementation (e.g. HashSet) always return the correct results. A hashcode is considered "valid but slow" if hash table operations are guaranteed to be correct, but are asymptotically much slower than otherwise possible with a better hash code. A hashcode is invalid if the code either doesn't compile, or some operations don't work as expected.

`hashCode()` implementation	**valid**	**valid but slow**	**invalid**
`return UUIDString;`	○	○	●
`return UUIDString.hashCode();`	○	●	○
`return seed;`	●	○	○
`return random.nextInt();`	○	○	●
`Random hashcodeRand = new Random(10);` `return hashcodeRand.nextInt();`	○	●	○

Small Area of Total Peace and Tranquility. Please, enjoy your stay.

6. More Graphs! As usual, throughout this problem, assume that our graphs have no edges that connect a node to itself, and that there is at most one edge between any two nodes.

a) **(25 points).** For each of the following propositions about graphs, mark Always True, Sometimes True, or Always False.

○ AT ● ST ○ AF	A* finds a correct shortest path (if it exists) from a source vertex to the goal vertex.
● AT ○ ST ○ AF	Suppose we have a goal node in mind. BFS finds a correct shortest path (if it exists) from a source vertex to that goal vertex in an unweighted graph.
○ AT ● ST ○ AF	The MST of a graph contains the largest edge in a graph.
● AT ○ ST ○ AF	Given a valid topological sort, the last vertex has no outgoing edges.
○ AT ● ST ○ AF	The middle vertex in a topological sort of 3 or more vertices has no edge going into it.
● AT ○ ST ○ AF	Given a graph with all positive edges except one or more negative edges leaving the start node, Dijkstra's algorithm finds a correct shortest paths tree.
○ AT ● ST ○ AF	Consider a variant of Dijkstra's algorithm DAV1 that tries to find a target node and stops as soon as the target node is <u>**enqueued**</u>. DAV1 finds a correct shortest path to the target on a graph with no negative edges.
● AT ○ ST ○ AF	Consider a variant of Dijkstra's algorithm DAV2 that tries to find a target node and stops as soon as the target node is <u>**dequeued**</u>. DAV2 finds a correct shortest path to the target on a graph with no negative edges.

b) **(12 points).** For this problem, consider only graphs with non-negative edge weights.

Naively, you'd assume that storing the shortest paths from a start vertex to every other vertex would require storing V - 1 lists, one for each target vertex, requiring $O(V^2)$ space. However, we proved in lecture that the shortest paths actually form a tree, allowing us to store a single "shortest paths tree" in O(V) space.

Suppose that we instead want to store the <u>second shortest paths</u> from a start vertex to every other vertex for which a second shortest path exists. Do the resulting edges form a tree? If yes, explain why. If not, give a counter-example.

○ Yes, there exists some sort of second-shortest-paths-tree because: ________________________

____

● No, because (include drawing with no more than 5 nodes):

No, consider A – B. The second shortest path from A → B is A→C→B, and second shortest path from
\ / A→C is A→B→C. Together, A→C→B and A→B→C do not form a tree.
C

7. Can you do that? For each of the following problems, select Yes and provide an explanation OR select No and provide a counterexample (for full credit) or an explanation (for partial credit).

a) **(10 points).** One way to implement insertion sort as described in class is to call `travel(0)`, `travel(1)`, `travel(2)`, …, `travel(N-1)`, in that order, where `travel(i)` is a helper function where the chosen item swaps itself with its left neighbor repeatedly as long as its left neighbor is greater than it. In other words, each item is the traveler exactly once, and traveling means heading as close to the front as possible. Suppose that we instead call travel in the reverse order, e.g. `travel(N-1)`, `travel(N-2)`, …, `travel(0)`. Does this also work? Assume that the travel operation is **exactly as above**. If yes, explain why. If no, give a counter-example.

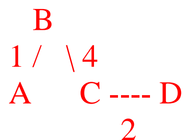
○ Yes, because: ____

● No, counter-example: 3 0 1 2 here, 2 gets stuck behind the 1

b. **(12 points).** Consider a new MST algorithm: Pruskal's Algorithm. For Pruskal's algorithm, we assume a connected weighted undirected graph. The algorithm is the following: Create a `Set<Edge>`. For each vertex in the graph, find the cheapest edge connected to it, and add this edge to the `Set`. Repeat until all vertices have been considered and return the Set. Does this algorithm correctly return the MST for all possible graphs? If so, explain why. If not, give a counter example. Assume edge weights are unique.

○ Yes, because: ____

● No, counter-example (no more than 5 nodes):



4 is not anybody's cheapest edge, but must be part of MST.

c. **(10 points).** Suppose we want to create a new class called `MutationSafeHeapMinPQ`.

```
public class MutationSafeHeapMinPQ<K extends Comparable<K>> {
    private K[] items;
    public void add(K item) { ... }
    public K min() { ... }
    public K removeMin() { ... }
}
```

The only difference between this and the `HeapMinPQ` from lecture is that our new class's reporting methods (`min` and `removeMin`) must always return the correct result, even if previously added items in the priority queue are modified. In other words, it handles mutable objects perfectly fine, even if they change while in the PQ.

Is it possible to implement such a class? If so, briefly describe how. If not, explain why not. Don't worry about runtime.

● Yes: Reheapify at the beginning of each call to min() and removeMin()

○ No, because: ____

8. **Weird Sorts.**

a) **(12 points).** Suppose we create a new sorting algorithm called `PartitionHybridSort(input, lo, hi, subsort)` where `input` is an input array of integers, `lo` is the lowest index in the array to be sorted, `hi` is the highest index in the array to be sorted, and `subsort` is the sorting routine to use in step 3:

1. If `hi - lo <= 1`, `return`.
2. Partition around `input[lo]` (i.e. the leftmost item of the current subproblem).
3. Use `subsort` to sort the left and right subproblems.

Give the **worst case** runtime for the function calls below. **Give your answer as a function of only N in big theta notation**. Each represents a different choice of `subsort`. For example, `InsertionSort` is a reference to an insertion sort implementation for integer arrays.

`PartitionHybridSort(input, 0, N, InsertionSort)`	$\Theta(N^2)$
`PartitionHybridSort(input, 0, N, Mergesort)`	$\Theta(N \log N)$
`PartitionHybridSort(input, 0, N, LSDSort)`	$\Theta(N)$
`PartitionHybridSort(input, 0, N, PartitionHybridSort)`	$\Theta(N^2)$

b) **(12 points).** In lecture, our implementation of LSD sort used counting sort as subroutine to sort by each digit. Suppose that we used Mergesort as a subroutine instead of counting sort. Let's call this new algorithm LSDMergesort.

The Mergesort used as a subroutine by LSDMergesort is exactly like regular Mergesort, except that its merge operation compares only one digit of an input to decide which is larger. For example, the merge operation would ordinarily consider 361 to be less than 410, but if we're sorting on the final digit, it will consider 361 to be larger than 430 (since 1 > 0).

Just like regular LSD sort, LSDMergesort would sort by the last digit, then second to last digit, and so forth. We can define LSDQuicksort in a similar way. Assume our LSDQuicksort uses shuffling, always picks the leftmost pivot, and uses Tony Hoare's partitioning scheme.

For each of these two new sorts, **give their worst case runtime in terms of N and W**, where W is the number of digits in each key. For simplicity, assume all keys have the same number of digits. Don't worry about the alphabet size R. **Also state whether or not they always return the right answer, and give an explanation for why**.

	Worst case runtime	Always works correctly?	Explanation for why it always works or not.
LSDMergesort	$\Theta(WN \log N)$	● Yes ○ No	Mergesort is stable.
LSDQuickSort	$\Theta(WN^2)$	○ Yes ● No	Quicksort is unstable.

9. Dijkstra's Runtime (24 points). In lecture, our cost model for Dijkstra's algorithm was to count the number of `add`, `changePriority`, and `removeMin` calls. We showed that we make V calls to `add`, O(E) calls to `changePriority`, and O(V) `removeMin` calls. Using a binary heap-based PQ that allows priority updates and can handle all three operations in O(log V) time, Dijkstra's algorithm therefore requires O(E log V + V log V) time.

For each of the priority queue implementations below, give a tight asymptotic runtime bound for each `add`, `changePriority` and `removeMin` operation, and also provide the total runtime for Dijkstra's algorithm. Assume that our graph has vertices numbered from 0 to V - 1.

The approaches to consider are:

1. `TrinaryMinHeapPQ`: Exactly the same as the PQ used in lecture (and described above), except that nodes can have up to 3 children instead of 2.
2. `UnorderedArrayPQ`: The PQ is a Double[] array. The ith entry in the array holds the priority value of the ith vertex stored as a Double. Any vertices that have been removed have a null value. Adding a new vertex increases the size of the array by 1.
3. `OrderedLinkedListPQ`: Each node in the linked list holds a vertex number and a priority value as a double. Items are stored in increasing order of priority.

Give your answers in terms of E and V. **Do not simplify expressions involving E and V**, e.g. don't simplify O(E log V + V log V) into O(E log V).

	`TrinaryMinHeapPQ`	`UnorderedArrayPQ`	`OrderedLinkedListPQ`
`add`	O(log V)	O(V)	O(V)
`changePriority`	O(log V)	O(1)	O(V)
`removeMin`	O(log V)	O(V)	O(1)
Total runtime	O(E log V + V log V)	O(V^2+E)	O(V^2 + EV)

10. By the Numbers (40 points).

Give the best and worst case number of `compareTo` calls needed for each task below. Give exact values! Do not include calls to `equals`, usages of primitive comparison operators, etc. The answer fomay be zero.

Best	Worst	
2	3	Solving puppy, cat, dog for N = 3.
4	10	Insertion sorting 5 numbers.
6	10	Inserting 5 numbers into an initially empty binary search tree.
0	0	Finding the successor (smallest item larger than) the root in a BST with 7 items.
20	49	Merging a sorted array of size 20 with a sorted array of size 30 to yield a sorted result.
6	8	Bottom up heapification of an array of 7 items.
1	3	Binary searching an ordered array of 7 numbers for a key.
0	0	Using counting sort to sort 52 cards by suit (heart, diamond, spade, club).

11. Sheep Herding. You are a shepherd and have found your way to the heart of Grigometh's maze. Your prize awaits: a line of golden sheep stands before you. You may take as many as you wish, as long as you follow the rules. Your goal is to take the sheep with the greatest total value. The rules are:

1. You must take the first sheep.
2. Walk forward between A and B spots, **inclusive**. If there are no sheep left, you are done. Otherwise, repeat step 1, where the sheep you just walked to is the new first sheep.

Four examples follow:

- `values = {1, 0, 1, -1, 1, 0, 10}, A = 2, B = 3, best = 13`
- `values = {0, 2}, A = 2, B = 3, best = 0`
- `values = {5, 2, -3, 2, 2, 10, 5, 3, 3, 4}, A = 3, B = 5, best = 19`
- `values = {2, 1, -3, 6, 8, -2, -1, 4}, A = 2, B = 3, best = 11`

Throughout this problem, assume that the correct answer is always small enough to fit into an integer, and that B ≥ A > 0. Note that A cannot be zero, otherwise we could keep picking the same sheep.

a) **(8 points).** To test your understanding, suppose we have `values = {5, 2, -3, 2, 2, 10}`, `A = 2`, and `B = 3`, compute `best`.

`best: 17`

b. **(20 points).** Fill in the naive recursive solution below.

```
public static int maxSheepNaive(int[] values, int A, int B) {
    return maxSheepNaiveHelper(values, A, B, 0);
}

/** Returns the max value if we take sheep with values[c] as well as the
  * subsequent sheep(s) with maximum total value according to the rules. */
public static int maxSheepNaiveHelper(int[] values, int A, int B, int c) {
    if (c >= values.length) {
        return 0;
    }
    int best = Integer.MIN_VALUE;
    for (int s = A;_s <= B; s += 1) {
        int val = values[c] + maxSheepHelperNaive(values, A, B, c + s);
        best = Math.max(best, val);
    }   // hint: For particularly concise code, use Math.max somewhere.
    return best;
}
```

c) **(20 points).** Unfortunately, the solution from part b can be exponential in runtime. Memoization is one way to speed things up considerably, but an even better approach is to use dynamic programming. The tricky part is deciding what subproblems you need to solve.

Your goal: Come up with a useful subproblem definition such that Q(N) is the solution to each subproblem, and the solution to the entire problem can be easily calculated given all Q(N).

Describe your subproblem briefly in English, and fill in your values for Q(N) in the table given below. Hint if you're stuck: Consider drawing a DAG, where an edge from i to j indicates that subproblem i is useful for solving subproblem j.

Description of Q(N) **(optional, ungraded)**: max total possible starting from N

```
values = {2, 1, -3, 6, 8, -2, -1, 4}, A = 2, B = 3, best = 11
```

values[N]	2	1	-3	6	8	-2	-1	4
Q(N)	11	13	9	8	12	2	-1	4

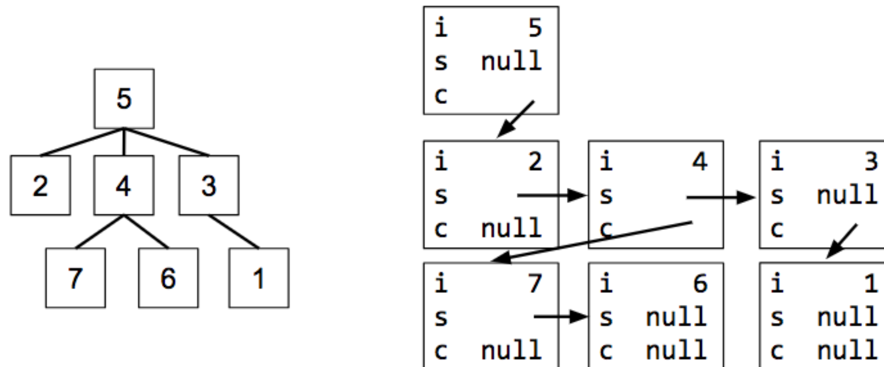
Note: Only your table values will be graded for part c!

DAG drawing **(optional, ungraded):**

d) **(7 points).** Give the runtime of the dynamic programming solution in big Theta notation in terms of N, A, and B. You may not need all variables.

$\Theta(N(B-A))$

12. IntSibTree. During lecture, we talked about an alternate way of implementing a tree known as a sibling tree. In a sibling tree, each node stores an item, a link to its first child (if any), and a link to its first sibling (if any). For example, the abstract tree given on the left would be represented as the structure on the right. Here `i` means `item`, `s` means `sibling`, and `c` means `child`. **<u>For this problem, assume that our tree contains only non-negative integers!</u>**



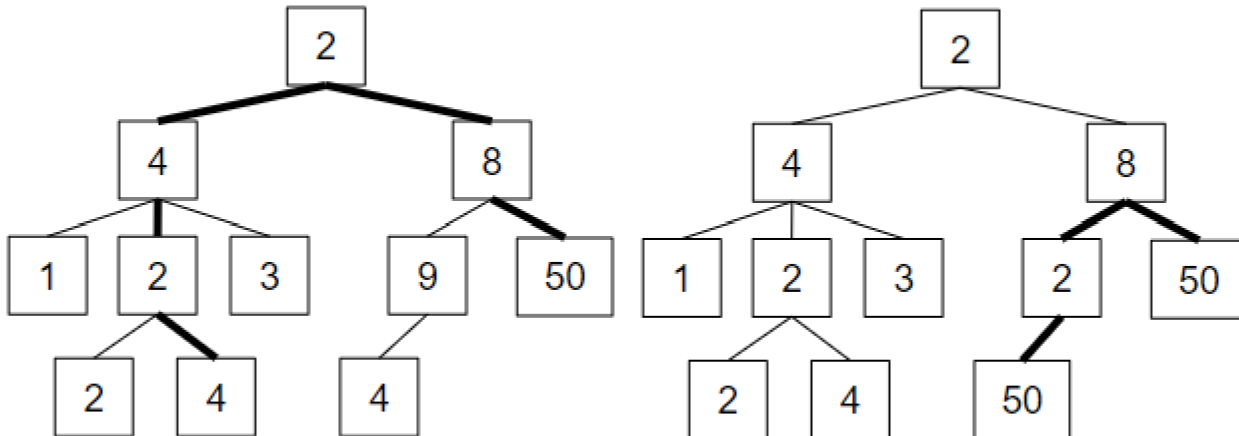
a) **(10 points).** Fill in the `largestSibling` method in the `IntSibTree` class below. For example, if called on the node containing 2, the method would return 4. You may not need all lines.

```
public class IntSibTree {
  public int item;
  public IntSibTree sibling;   // link to first sibling
  public IntSibTree child;     // link to first child
  /** Returns largest item from siblings of t (including t) or 0 if null. */
  public static int largestSibling(IntSibTree t) {
      if (t == null) { return 0; }
      ____________________________________________________
      ____________________________________________________
      return Math.max(t.item, largestSibling(t.sibling));
  }
  public static int secondLargestSibling(IntSibTree t) { // code not shown }
}
```

b) **(13 points).** Suppose we want to add a post order method to the `IntSibTree` class that performs a post order visit of the `IntSibTree` using the `Visitor` pattern from lecture. Fill in the method below. You may not need all lines. The `Visitor` interface is defined below the method.

```
public static void postOrder(IntSibTree t, Visitor v) {
    if (t == null) { return_____________ } // Note: postOrder is part of
    postOrder(t.child, v);______________    // the IntSibTree class.
    v.visit(t);_________________________    // For above graph, should visit
    postOrder(t.sibling, v);                // in order 2, 7, 6, 4, 1, 3, 5.
}
public interface Visitor {                  // The Visitor interface is not
    public void visit(IntSibTree t);        // part of the IntSibTree class.
}
```

c) **(28 points).** Let the "max path" be the path that has the maximum total weight in the entire tree, where the weight of a path is defined as the sum of the integers along the path. For example, the max path for the tree on the left has weight 70, and the tree on the right has max path weight 110 (and doesn't include the root). An empty tree has a max path weight of zero.



Fill in the `MaxPathFinder` class such that the code below prints the weight of the best path. **Your visitor may be destructive,** but is not required to be destructive. You may not need all lines.

<u>THIS PROBLEM IS HARD. DON'T SPEND TOO LONG ON IT!</u>

```
  Visitor v = MaxPathFinder();
  IntSibTree.postOrder(ist, v);  //ist is some IntSibTree
  System.out.println(v.result());//prints 70 for left example, 110 for right

public class MaxPathFinder implements Visitor {
    int bestPathSum;

    public MaxPathFinder() {
        bestPathSum = 0;
    }

    public void visit(IntSibTree t) {
        int largest = t.largestSibling(t.child);
        int secondLargest = t.secondLargestSibling(t.child);
        int subTreeTotal = largest + t.item;
        int thisBestPathSum = largest + secondLargest + t.item;
        t.item = subTreeTotal;
        bestPathSum = Math.max(bestPathSum, thisBestPathSum);
    }
    public int result() {
        return bestPathSum;
    }
} //  And now you're done! Unless you want another crack at Flight…
```

13. (30 points). Flight Planning. This problem will be scored under two special rules:
1. Your score for this problem does not count towards your final exam score, but will instead be added to your score on problem 9 of midterm 2. If that total exceeds 30 points, the overflow is gold points.
2. **If you fill in the bubble below**, we'll give you 10% credit instead of grading your answer.

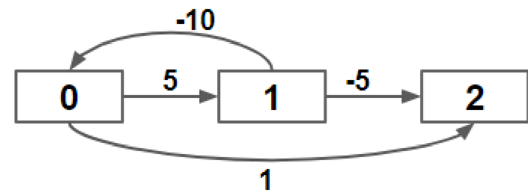
◯ I'll just take 10% credit, please don't grade my answer.

Consider the `SNRPLTE` (shortest no-repeats-path less than exists) method below. It returns true if there exists a path in the graph from s to t that:

1. Has total weight less than or equal to W.
2. Has no repeated vertices, i.e. each **vertex** appears at most once.

```
/** Returns true if there is a path from s to t in G with weight <= W. */
public static boolean SNRPLTE(Graph G, int s, int t, int W) { ... }
```

For example, `SNRPLTE` returns true on the graph to the right for `s = 0, t = 2,` and any value of `W` that is greater than or equal to 0. `SNRPLTE` handles undirected and directed edges.



Suppose that the runtime of SNRPLTE is Θ(F(V, E)), where F(V, E) is some unknown function that is Ω(V + E).

Given an undirected unweighted graph G, where each node is an airport, and each edge indicates that there exists a flight between the two airports, suppose we want to know if there exists a path that visits **every airport exactly once**, which we'll call a world tour. In other words, you want to write the method below:

```
/** Returns true if there exists a world tour of the graph G. */
public static boolean WTE(Graph G) { ... } // G is unweighted and undirected
```

Give a concise but thorough explanation of how you could solve `WTE` for a given graph using `SNRPLTE`. Your algorithm must have runtime Θ(F(V, E)), i.e. the same as `SNRPLTE`. **Give your algorithm as a numbered list of steps.** You may not modify the SNRPLTE method in any way.

1. Given G, create a new graph γ with the following properties:
a. Copies of every vertex and edge in G as well as two new vertices s and t.
b. New directed edges from s to every vertex that was copied from G.
c. New directed edges from every vertex that was copied from G to t.
d. Set all edge weights to -1.

2. Return the result of SNRPLTE(γ, s, t, -(V+1))

Why this works: Let V be the number of vertices in G. A shortest path with no repeats on γ of length (V – 1) + 2 would be the same as a world tour which must have V – 1 edges. Such a path γ would have total weight –(V+1).

Thus we simply need to return the result of SNRPLTE(γ, s, t, -(V+1))