UC Berkeley – Computer Science
CS61B: Data Structures

Final, Spring 2016

This test has 13 questions worth a total of 100 points. The exam is closed book, except that you are allowed to use three pages (both front and back, for 6 total sides) as a written cheat sheet. 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.** Any plagiarism, no matter how minor, will result in an F.

"I have neither given nor received any assistance in the taking of this exam."
Video walkthrough: Nonexistent, sorry :(

Signature: _____

Name: _____ **Your 3-Letter Login:** _____
SID: _____ **Name of person to left:** _____ **No ID:** _____
Exam Room: _____ **Name of person to right:** _____ **No ID:** _____
Primary TA: _____

Tips:

- There may be partial credit for incomplete answers. Write as much of the solution as you can, but bear in mind that we may deduct points if your answers are much more complicated than necessary.
- 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.
- 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 during the exam, we'll announce a fix. Unless we specifically give you the option, the correct answer is not 'does not compile.'
- The exam roughly increases in difficult as you approach the end.

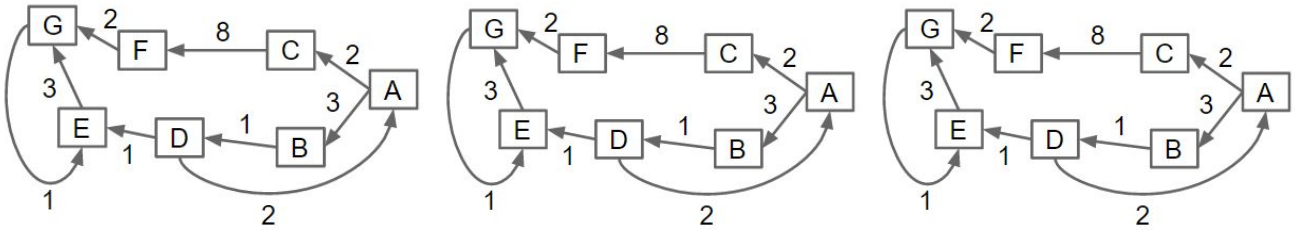
Problem	0	1	2	3	4	5	6	7	8	9	10	11	12	13
Points	0.5	8	8	6	12	7.5	4	5	0	10	6	10	13	10

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

Before exam: [:(_____☺].
After exam: [:(_____☺].

0. So It Begins III (0.5 points). Write your name and ID on the front page. Circle the exam room. Write the names of your neighbors. If a neighbor is missing their ID, make sure to mark No ID in the appropriate blank. Write and sign the given statement. Write your login in the corner of every page.

1. **Giraphage (8 points)**. For your convenience, we have provided 3 copies of the graph for parts a through c.



a. **(2 pts)** For the graph above, give the vertices in the order they'd be visited by depth first search starting from vertex A, assuming that we always visit alphabetically earlier vertices first if there are multiple valid choices. You may not need all blanks. The alphabet is ABCDEFG.

___A___ ___B____ ___D____ ___E____ ___G____ ___C____ ___F____ _____

b. **(2 pts)** For the graph above, give the vertices in the order they'd be visited by breadth first search starting from vertex D, assuming that we always visit alphabetically earlier vertices first if there are multiple valid choices. You may not need all blanks.

___D___ ___A____ ___E____ ___B____ ___C____ ___G____ ___F____ _____

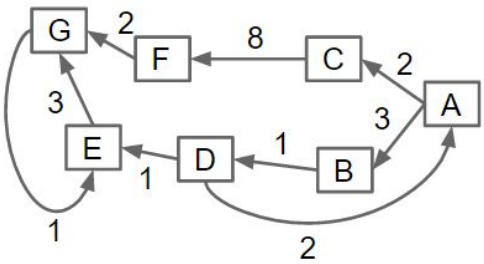
c. **(2 pts)** For the graph above, give the vertices in the order they'd be visited by Dijkstra's algorithm starting from vertex A, assuming that we always visit alphabetically earlier vertices first if there are multiple valid choices. Assume that "visiting a vertex v" means "relaxing all of the edges out of v". You may not need all blanks.

___A___ ___C____ ___B____ ___D____ ___E____ ___G____ ___F____ _____

Dijkstra's works by expanding the node that is the least distance away from A first.

d. Suppose we are trying to find the shortest path from A to G. Give an example of a heuristic function for which A* returns the **wrong shortest paths tree**. Specify your heuristic function by circling **<u>one number</u>** for `h(E)`.

```
h(A) =   0
h(B) =   3
h(C) =   2
h(D) =   2
h(E) =   -4   1   2   5   8   14
h(F) =   1
h(G) =   0
```



By selecting a large value for h(E), A* search will prioritize searching the bottom route lowly, and thus returns the non-optimal top solution when it finds G.

2. **Sorting (8 points).** a) **(6 pts)** Below, the leftmost column is an array of strings to be sorted. The column to the far right gives the strings in sorted order. Each of the remaining columns gives the contents of the array during some intermediate step of one of the algorithms listed below. Match each column with its corresponding algorithm. You will use each answer once. Write your answer in the blanks provided.

7777	1979	1979	7777	2001	1234	1234
2001	1234	2001	7777	8009	2001	1979
3015	1984	2015	4444	3015	3015	1981
2015	1981	2048	2015	2015	2015	1984
2048	2001	3015	7450	2016	2048	2001
8009	2015	4444	3015	2048	1981	2015
1979	2048	4500	2016	9150	1979	2016
7777	2016	7450	2001	1234	2016	2048
9150	3015	7777	1979	4444	1984	3015
4500	4500	7777	4500	7450	4500	4444
7450	4444	8009	2048	4500	7450	4500
4444	7777	9150	1981	7777	4444	7450
1234	7777	1234	1234	7777	7777	7777
1984	7450	1984	1984	1979	9150	7777
2016	8009	2016	8009	1981	7777	8009
1981	9150	1981	9150	1984	8009	9150
----	----	----	----	----	----	----
_1_	_6__	_2__	_4__	_5__	_3__	_7__

1: Unsorted, 2: Insertion, 3: Quick, 4: Heap, 5: LSD, 6: MSD, 7: Sorted

Column 2 is sorted by the thousandths place => MSD Sort
Column 3 is perfectly sorted except for the last four numbers, which are untouched => Insertion Sort
Column 4 is in heap form, with the last 2 max elements in its correct place => Heap
If you ignore the thousandths place of all the elements of column 5, it is in order => LSD
The second to last column immediately puts the first element in the correct place => strongly hints QuickSort

Notes:

- Quicksort is non-random and uses leftmost item as a pivot. The pivoting strategy is the Hoare two-pivot strategy, discussed in class.
- Insertion, Heapsort, LSD Radix Sort, and MSD Radix Sort as described in class.

b) **(1 pt)** One way to sort N items is to insert them randomly into a left leaning red black tree, and then traverse the LLRB. Which traversal should we use in order to print out the keys in sorted order? Circle one:

Preorder **Inorder** **Postorder** **Reverse-Preorder** **Reverse-Postorder**

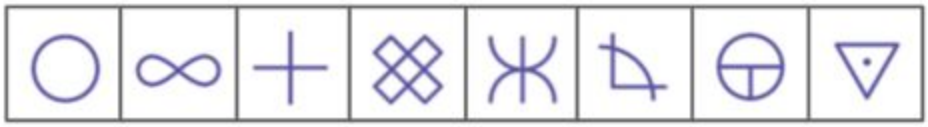
c) **(1 pt)** `Θ(N * log(N))` What is the worst case runtime of the sort described in part b? **Give your answer in Big Theta notation in terms of N**. Put your answer in the given blank.

3. Reverse Engineering (6 Points).

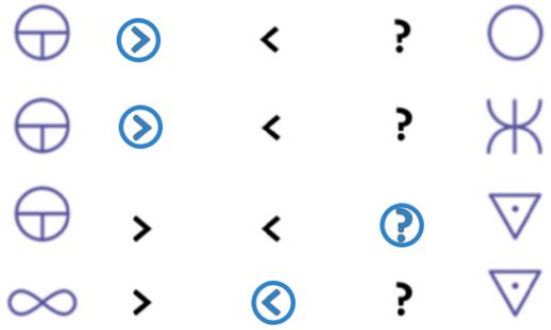
a. **(4 pts)** Consider the following unsorted array, and the array after an unknown number of iterations of selection sort as discussed in class (where we sort by identifying the minimum item and moving it to the front by swapping). Assume no two elements are equal.

Unsorted:

After ? Iterations of Selection Sort:

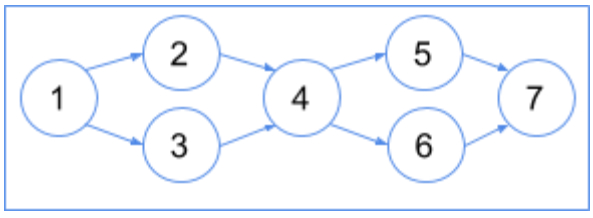


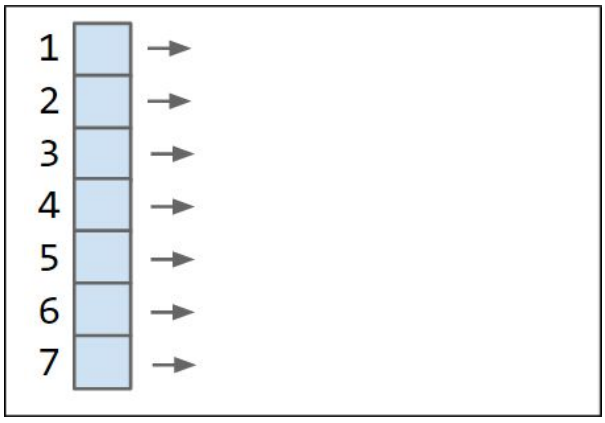
For each relation, **circle** <, >, or ? if there is insufficient information to determine the relation between the two objects. For example, if you believe that  is greater than  , you'd circle the > on the first line.



b. **(2 pts)** Suppose we have a graph G. All of G's topological sorts are listed below. **In the space to the right, fill in the <u>adjacency list</u> for G**. There may be more than one right answer. Don't worry about the exact formatting for your answer. As long as it is an adjacency list and it is easy to understand, we will accept your answer. **<u>Graph drawings will be given only partial credit -- please fill out the adjacency list!</u>** There are multiple answers. The minimal one is given. Note, topological sorts only exist for directed graphs!

1 2 3 4 5 6 7
1 2 3 4 6 5 7
1 3 2 4 5 6 7
1 3 2 4 6 5 7





4. **Facts (12 Points)**

a. **(7 pts) FSacginorstt.** You will be given an answer bank, each item of which **<u>may be used multiple times. You may not need to use every answer.</u>**

Word Bank

A. QuickSort (non-random, in-place using Hoare partitioning, and choose the leftmost item as the pivot)
B. MergeSort
C. Selection Sort
D. Insertion Sort
E. LSD Sort
F. MSD Sort
G. HeapSort
N. (None of the above)

Questions

List all letters that apply. List them in alphabetical order, or if the answer is none of them, use N. **<u>All answers refer to the entire sorting process, not a single step of the sorting process.</u>** For each incorrect letter (either additional or missing), you will lose half credit for that blank.

A, B, C bounded by Ω(N log N) lower bound. Note: Insertion and heapsort can be linear time.

B, G is a comparison sort and has worst case runtime that is asymptotically better than Quicksort's worst case runtime.

C in the worst case, performs Θ(N) pairwise swaps of elements.

A, B, D comparison based sort, and never compares the same two elements twice.

N runs in best case Θ(log N) time for certain inputs.

b) Tree Facts (5 pts). Answer 'True' or 'False' for each of the statements below.

Free Inserting a single item into a "bushy" (balanced) BST with N items takes Θ(log N) time in all cases. Note: We took either True or False for this one, since bushy could be interpreted to mean h = Θ(log(N))

False Inserting a single item into a heap with N items takes Θ(log N) time in all cases.

True The height of a BST with N items is O(N). (Note the Big O).

False Suppose X is a valid BST containing integers. If we square all values in X, the result is always a BST.

True All valid left leaning red black trees are valid BSTs.

False All valid weighted quick union trees are valid BSTs.

Free The parent of the second largest item in a max heap is always the root. Note: We took either true or false for this one, since it was unclear how to resolve duplicates.

False The parent of the parent of the third largest item in a max heap is always the root.

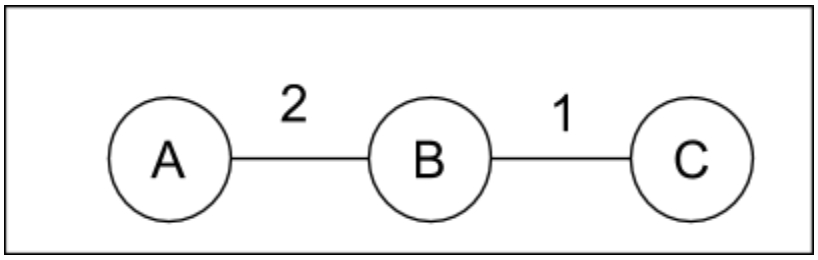
True The height of a perfectly balanced quadtree with N items is **<u>asymptotically</u>** the same as the height of a 2-3 tree with N items.

False Finding all matching tiles in getMapRaster using a quadtree takes Θ(log N) time in all cases, where N is the number of png files.

5. (7.5 pts) Graph Algorithms. For each statement below, circle either **TRUE** or **FALSE**. **If your answer is FALSE, draw a counterexample graph in the given box, and if applicable, provide a starting vertex**. Please use **unique** edge weights for any weighted graphs. If your answer is true, don't do anything in the box/blank. **Any counterexamples should have 5 vertices or less.** Keep in mind these are only worth 1.5 points each!

FALSE: The last edge added to the MST by Prim's algorithm is always the highest weight edge in the MST.

Starting vertex: ___A___

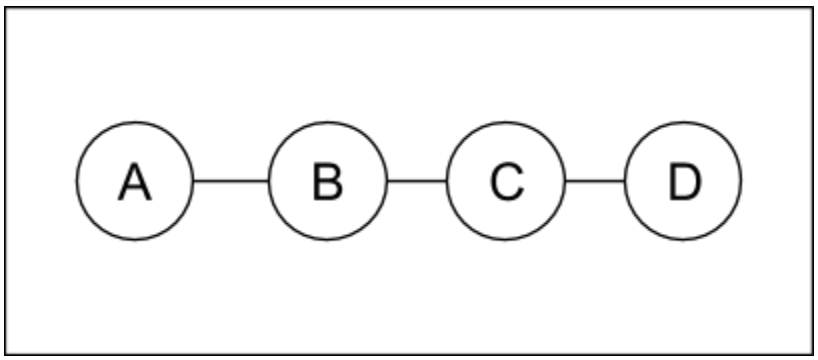


FALSE: The largest edge in a graph is never part of a SPT.

Starting vertex: ___Either___

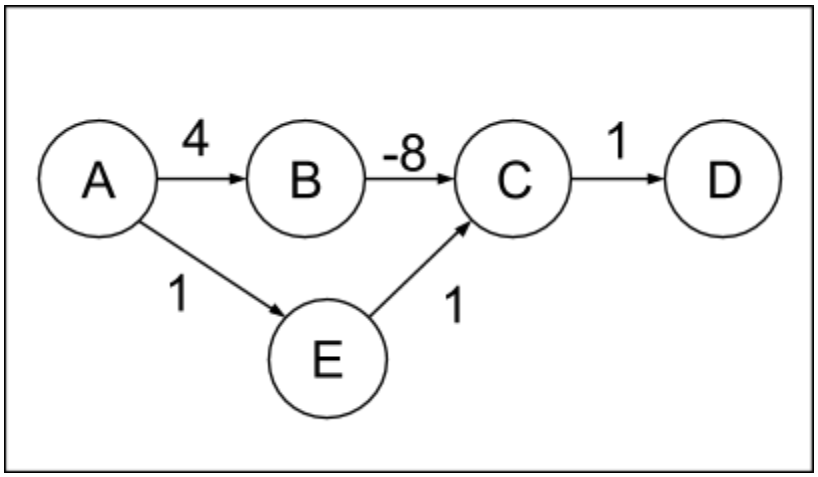
FALSE: On a graph of 4 or more nodes, DFS and BFS never visit vertices in the same order when run from the same start vertex.

Starting vertex: ___A or D___



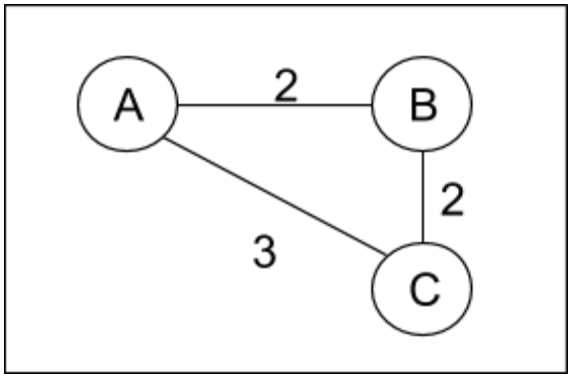
FALSE: Dijkstra's algorithm always finds the shortest path in a directed acyclic graph, even if there are negative edges:

Starting vertex: ____A____



FALSE: In any undirected graph, the shortests paths tree from any vertex always has total weight less than or equal to the weight of the MST for that graph.

Starting vertex: ____A____

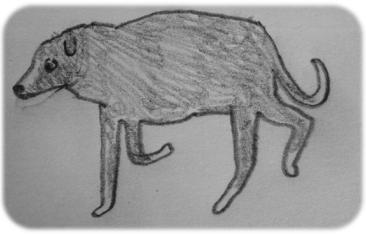


6. **Advanced Hash Party (4 points).**

(4 pts) There are other ways to resize a hash table than the way we discussed in lecture. For each scheme below, give the **amortized** best and worst case runtime for a single insertion operation in Θ notation in terms of N, the number of items in the hash table. If the operation given could result in an infinite runtime, write "infinity" or ∞ inside the big Theta. Assume the hashCode function takes constant time. By the "best case", we mean a set of items that are spread out nicely by their hashCode, and by the "worst case", we mean a set of items that have the worst possible collisions by their hashCode. Define L to be the average number of items in each bucket. Assume resizing takes linear time. Note: For all answers below, L is bounded above by a constant, so we do not include them as part of our answers.

Best Case	Worst Case	Scheme
Θ(1)	Θ(N)	Quadruple # of buckets when L > 1.
Θ(1)	Θ(N)	Double # of buckets when L > 1/100.
Θ(N)	Θ(N)	Increases # of buckets by 10 when L > 1
Θ(1)	Θ(∞)	Doubles # of buckets until no bucket has more than 5 elements in it. This may result in multiple doublings!

Enjoy this **space**.



7. Asymptotics (5 points). For parts a, b, and c, assume that f runs in $\Theta(N)$ time (in all cases), g runs in $\Theta(N^2)$ time (in all cases), and h runs in $O(N^2)$ time (in all cases). Assume that each function returns an array of the same length as its input. **Note, the runtime for h is given in O notation, not Θ notation.**

a) **(1 pt)** Give the runtime to complete the doStuff1 method in Θ notation if possible, or O notation otherwise. Your answer should be simple, with no unnecessary leading constants or unnecessary summations. Write your answer in this blank: _____$\Theta(N^2)$_____

```
public static int[] doStuff1(int[] x) {
    int N = x.length;
    int[] effedX = f(x);

    int[] newArray = new int[N];
    for (int i = 0; i < N; i += 1) {
        newArray[i] = effedX[i] * 2;
    }

    int[] result = g(newArray);
    return result;
}
```

The complexity $\Theta(N^2)$ comes from the g(newArray) call.

b) **(3 pts)** Give the runtimes for each line of code shown in Θ notation if possible, or O notation otherwise. Your answer should be simple, with no unnecessary leading constants or unnecessary summations.

```
public static void doStuff2(int[] x) {
    int N = x.length;
    int[] fx  = f(x);                    _____Θ(N)_____
    int[] gfx = g(f(x));                 _____Θ(N^2)_____
    int[] hgfx = h(g(f(x)));             _____Θ(N^2)_____
    int[] hfx = h(f(x));                 _____O(N^2)_____
    int[] ddx = f(f(x));                 _____Θ(N)_____
}
```

Running g(f(x)) is asymptotically the same as running g(x) then f(x), so the complexity of g(f(x)) is the *sum* of the runtimes of g and f. That's $\Theta(N^2) + \Theta(N) = \Theta(N^2 + N) = \Theta(N^2)$. Observer that when we write g(f(x)), we're basically just calling doStuff1() (but without the for loop).

Similarly, for h(g(f(x))), the correct answer is $O(N^2) + \Theta(N^2) + \Theta(N) = O(N^2) + \Theta(N^2)$, which simplifies to just $\Theta(N^2)$ because $\Theta(N^2)$ *implies* $O(N^2)$.

c) **(1 pt)** Suppose we have an input array x. Will f(x) always take fewer seconds to execute than g(x), assuming we run them on the same computer? Briefly explain why or why not in the blank below.

No, asymptotic runtime only applies for arbitrarily large inputs. If the input size is small, it is possible for g to execute faster than f (constant factors may become significant).

8. (0 points) Lloyd's of London predicted in 2013 that this 1859 event, if it occurred today, would cause as much as 2.6 trillion dollars of damage to the U.S. economy. What event were they referring to?

Carrington event (solar storm)

9. Heaps and JUnit (10 points).

Write a JUnit test to check a method `public void minheapify(int[] arr) {...}` that is supposed to perform bottom-up min-heap heapification. This means ensuring that the array is actually a heap, and also that the array still has all the same items. Our tests will verify only correctness, not runtime. Assume there will be no duplicates (which may make it easier to test that the array still contains the same inputs after heapification).

- **Hint: We're not leaving index 0 blank, so the left child of k is 2k + 1, and the right child is 2k + 2.**
- Hint 2: Feel free to use assertEquals(x, y), assertTrue(b), assertArrayEquals(x, y), etc.
- Hint 3: If you don't remember the exact syntax but your meaning is clear, penalties will be minimal.

```
@Test
public static void testHeapify() {
      int[] arr = generateRandomIntArray();
      int[] original = new int[arr.length]; // copy of array before being
heapified
      System.arraycopy(arr, 0, original, 0, arr.length);
      minheapify(arr);
      // Check the integrity of the result using one or two calls to helper
methods
      testIsAHeap(arr);
      testHaveSameItems(original, arr);
}
private static void testIsAHeap(int[] arr) {
    for (int i = 0; i < arr.length; i++) {
        if (2 * i + 1 < arr.length) {
            assertTrue(arr[i] < arr[2 * i + 1]);
        }
        if (2 * i + 2 < arr.length) {
            assertTrue(arr[i] < arr[2 * i + 2]);
        }
    }
} //You may not need all lines for these methods. Must include at least one
assert!
private static void testHaveSameItems(int[] original, int[] arr) {
    assertEquals(original.length, arr.length);
    for (int i = 0; i < arr.length; i++) {
        boolean exists = false;
        for (int j = 0; j < original.length; j++) {
            if (arr[i] == original[j]) {
                exists = true;
            }
        }
```

```
        assertTrue(exists);
    }
} // There is also a shorter solution that uses Arrays.sort
```

10. Fancy Asymptotics (6 points)

Ben Bitdiddle has created a generalized sorting algorithm called `BitdiddleSort`. The pseudocode is provided:

```
procedure BitdiddleSort(array):
    if the array has length 1:
        return the array
    else:
        divide the array into two equal halves, half1 and half2
        sort half1 using algorithm A
        sort half2 using algorithm B
        merge the two sorted halves in O(N) time and return the merged result
```

Let N be the length of the input array. Assume the array consists only of integers between 1 and 9.

a) Suppose algorithm A is a comparison sort and algorithm B is counting sort. In big-omega notation, give the tightest possible lower bound on the runtime of `BitdiddleSort` as a function of N.
Hint (that you might not actually need): log(ab) = log(a) + log(b).

Answer: Ω(N log N) was the intended answer because comparison sorting takes Ω(N log N) time in the worst case. But since the question didn't specify "in the worst case", Ω(N) is also acceptable. (For example, if the input is already sorted, then insertion sort, which is a comparison sort, runs in linear time.)

b) Suppose algorithm A is counting sort and algorithm B is quicksort. In big-O notation, give the tightest possible upper bound on the runtime of `BitdiddleSort` as a function of N.

Answer: O(N^2). Quicksort runs in O(N^2) time and counting sort runs in O(N) time.

c) Suppose algorithm A is quicksort and algorithm B is BitdiddleSort. In big-O notation, give the tightest possible upper bound on the runtime of `BitdiddleSort` as a function of N.

Answer: O(N^2). Quicksort runs in O(N^2) time. It's applied to N/2 items, then N/4 items, then N/8 items, and so on. So the runtime is O(N^2 * (1/4 + 1/16 + 1/64 + … + 1/N^2)) = O(N^2).

d) Suppose algorithm A and B are both `BitdiddleSort`. In big-theta notation, give the runtime of `BitdiddleSort` as a function of N.

Answer: Θ(N log N). This is just mergesort.

11. Dynamic Programming (10 points): Warning, the exam from here on out is pretty hard!

Letters in the alphabet that are next to each other are said to be **neighborly**. For example, `'c'` and `'d'` are neighborly, and so are `'b'` and `'a'`. Note that `'a'` and `'z'` are not neighborly. Characters are also not neighborly with respect to themselves: `'a'` and `'a'` are not neighborly.

Given a non-empty array of **lowercase** characters (`'a'` through `'z'`), find the length of the longest alphabetically neighborly subsequence (LANS) of the array. Remember that **subsequences are not necessarily contiguous**, and that **neighborly can be either increasing or decreasing**. Examples **(read carefully!)**:

* For input `['a', 'b', 'c']`, the answer is 3, since the entire array is the LANS.
* For input `['a', 'a', 'c', 'a', 'd']`, the answer is 2, since the LANS is the subsequence `['c', 'd']`.
* For input `['a', 'd', 'c', 'd']`, the answer is 3, since the LANS is the subsequence `['d', 'c', 'd']`.
* For input `['a', 'z', 'a', 'z']`, the answer is 1, since the LANS can be any character as a standalone subsequence, e.g. `['a']`.

Your algorithm must run in O(N) expected time, where N is the length of the input. Solutions that do not run in O(N) expected time will receive zero points. It is OK to assume constant time HashMap operations. **You may assume you have access to the following 3 methods**, which take constant, linear, and constant time:

```
/* returns hm.get(key) if hm.containsKey(key), defaultValue otherwise */
public static <K, V> V get(HashMap<K, V> hm, K key, V defaultValue)
/* returns the largest value in the HashMap hm (linear time) */
public static <K, V> V maxValue(HashMap<K, V> hm)
```

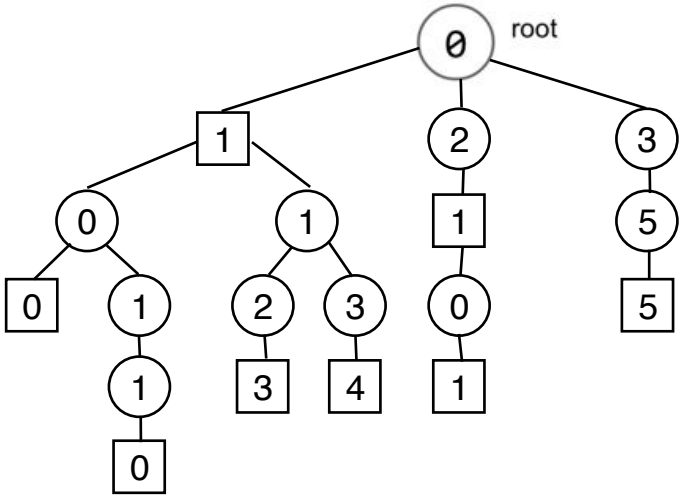
- Also, don't forget about `Math.max(int x, int y)` and `Math.min(int x, int y)`

Hint: You can use the subtraction operator - to find the distance between two char values without casting to int. For example: `'b' - 'a'` evaluates to 1. Similarly: `'c' - 1` evaluates to `'b'`.

```
public static int llans(char[] input) {
    HashMap<Character, Integer> cache = new HashMap<Character, Integer>();
    cache.put(input[0], 1);
    for (int i = 1; i < input.length; i += 1) {
        int prev = get(cache, input[i] - 1, 0);
        int next = get(cache, input[i] + 1, 0);
        cache.put(input[i], Math.max(prev, next) + 1);
    }
    return maxValue(cache);
}
```

12. Calling Collect: Tries and Recursive Tree Programming (13 points).

a. **(2 pts)** Suppose we implement a set of integers by using a Trie that stores positive base 10 numbers digit-by-digit (so R = 10). Draw the R-way Trie (not TST!) that results from inserting the numbers `1, 100, 10110, 1123, 1134, 2101, 355, 21`. Inside each node, draw the value of that node. We have drawn the root for you, which has a dummy value of 0. Do not draw null links. **Put a square box around all nodes corresponding to keys that exist.** If two nodes have the same parent, the one that is less should go to the left (e.g. if the root has a child node with a 1, and another with a 2, the node with a 1 should be to the left). Like strings, insert most significant digits first.



b. **(1 pt)** Suppose we have the `TrieIntegerSet` definition given below.

```
public class TrieIntegerSet {
   private Node root = new Node(0);        // root of trie

    // R-way trie node
    private class Node implements Comparable<Node> {
        private Set<Node> children = new TreeSet<Node>();
        private int dig;                // this digit
        private boolean exists;         // true if this item exists

        public Node(int x)             { dig = x; }
        public int compareTo(Node x)   { return this.dig - x.dig; }
    } ...
```

`collect(Node z, List<Integer> matches, int topDigits)` is a method which finds all keys in the subtrie rooted at `z`, appends `topDigits` to each key, and adds the result to `matches`. Example: `collect(z1, matches, 99)` would add `[991, 99100, 9910110, 991123, 991134]` to `matches`, assuming that `z1` is the 1 child of the root. There is no specific required ordering for the keys added by a call to `collect`.

Give the results of calling `collect(z10, matches, 1)`, assuming `z10` is the **0 child of the 1 child** of the root. Use exactly the two blanks provided below. Either order is fine. Hint: Unlike our example above with `topDigits = 99`, your answer should exactly match two keys in the trie from part a! This problem should be easy and is setting you up for part c.

____100____ ___10110___

c. (5 points) Fill in the private `collect` method so that it behaves as in part b. Note that if `topDigits` is 0 you should not prepend a zero (in fact, it's impossible), i.e. `collect(z10, matches, 0)` would add `[0, 110]`. Assume that `collect` is never called on the root, so you don't need to worry about any weird edge cases. **Note that this method is a method of `TrieIntegerSet`, not `Node`.**

```
/* Collects a list of all keys in the subtrie rooted at x, assuming that
 * they are all prefixed by topDigits. Assume never called on root. */
private void collect(Node x, List<Integer> matches, int topDigits) {
   if (x == null) { return; }
   if (x.exists == true) { matches.add(topDigits * 10 + x.dig); }
   for (Node child : x.children) {
      collect(child, matches, topDigits * 10 + x.dig);
      ____________________________________________________________;
   }
}    // You may not need all lines. This is a time consuming problem.
```

d. (5 points) Fill in the private method below such that `public findRepeaters` returns a list of all numbers in a `TrieIntegerSet` that have any consecutive repeated digits. For example, for the Trie from part a, this method would return `[100, 10110, 1123, 1134, 355]`. It would not return `2101` since the 1s are not consecutive. You may use `collect` from part c even if you didn't finish it or your answer is incorrect. You do not need to use the modulus operator % for this problem. Order doesn't matter. This method also belongs to **`TrieIntegerSet`.**

```
public List<Integer> findRepeaters() {
   List<Integer> matches = new ArrayList<Integer>();
   findRepeaters(root, matches, 0);
   return matches;
}

/** Finds all keys in the subtrie rooted in x that have at least one
   * pair of repeated digits, assuming they are all prefixed by topDigits. */
private void findRepeaters(Node x, List<Integer> matches, int topDigits) {
   if (x == null) { return; }
   for (Node child : x.children) {
      if (x.dig == child.dig) {
         collect(child, matches, topDigits * 10 + x.dig);
      } else {
         findRepeaters(child, matches, topDigits * 10 + x.dig);
      }
      ____________________________________________________________
      ____________________________________________________________
   }
}       // You may not need all lines. This is a time consuming problem.
```

13. **A (Fond?) Farewell to 61B (10 points).** In your life after 61B, you'll often need to use one data structure to implement another. In this problem, you'll build a FIFO (first-in first-out) queue of type `MagicStringQueue` which has two operations that are just like a regular queue, namely `enqueue` and `dequeue`. For example, if we made the following calls into an initially empty `MagicStringQueue` called `mq`:

- `mq.enqueue("giraffe"), mq.enqueue("zebra"), mq.enqueue("alf")`
- `System.out.println(mq.dequeue())`

Then the print statement would print **`"giraffe"`** since it was at the front of the queue. Instead of using an array or linked list to build the queue, you must use a `MagicBag<K>`, which has the following operations:

```
public void add(K key): adds an item of type K to the MagicBag, or replaces
                        it if there is already an item that .equals key
public K remove(K key): removes the item that .equals key (if it exists)
                        and returns that item in the bag, or null otherwise
```

Describe how you'd build a `MagicStringQueue` using only a `MagicBag` and **a constant amount of additional memory.** Solutions that use more memory will be given zero points.}

Notes: You may assume that enqueue() is called at most 1 million times. Your `MagicStringQueue` **may only use a constant amount of memory, except for a single `MagicBag` which may use linear memory.** It is OK to create a single helper class (see part b of this problem). Strings are immutable in Java.

a) List the instance variables of your `MagicStringQueue`.

```
class MagicStringQueue {
      MagicBag<Helper> bag;
      int addNumber, removeNumber;
}
```

b) Very briefly describe your helper class (if needed). List its instance variables as well as any methods. Briefly describe how any such methods work. Include any default methods that you @Override.

```
class Helper {
      String value;
      int position;
      public Helper(String value, int position);
      public boolean equals(Object obj);  // Dependent on position
      public int hashCode();              // Consistent with .equals
}
```

c) Describe how your `MagicStringQueue`'s enqueue and dequeue operations work in terms of its instance variables (including any calls to `MagicBag`'s methods). You may use pseudocode if you'd like. Do not write Java code. Don't worry about handling dequeueing from an empty queue.

Use two integer instance variables in MagicStringQueue to keep track of (1) the "queue position" of the first item in the queue (*removeNumber*), and (2) the queue position that should be used for the next item that's enqueued (*addNumber*). The queue position of the first item enqueued will be 0, the position of the next item will be 1, and so on. When an item is enqueued, it will be assigned a position equal to *addNumber*, and *addNumber* will be incremented by 1.

The MagicBag will store a helper class with two instance variables: a String representing the item in the queue, and an integer representing the queue position. To make it possible to implement the dequeue operation, the helper class should override the .equals() method to only compare the position. To dequeue something from the queue, the dequeue method can create a dummy instance of the helper class with position equal to *removeNumber* and then call the remove() method of MagicBag using that dummy object. Before returning, the dequeue method should increment *removeNumber*.

Or in pseudocode:

```
enqueue(string) {
      Helper h = new Helper(string, toAdd);
      bag.add(h);
      increment addNumber;
}

dequeue() {
      Helper h = new Helper("", toRemove);
      Increment removeNumber;
      return bag.remove(h).value;
}
```

Solution B (no credit): Create a Node helper class with reference to other nodes. Store a Node instance variable in MagicStringQueue. Use the nodes to store the items that need to be enqueued/dequeued, effectively reusing the idea from LinkedListDeque from project 1a. This approach was not allowed even if you put all the nodes in a MagicBag. In this solution, the MagicBag is useless, and you've just built a linked list, which is forbidden by the problem statement.

Solution C (no credit): Create an array with 1,000,000 items and keep two pointer indices to the front and the back. This approach is not acceptable because you are just using an array, and are basically redoing ArrayDeque from project 1a. In this solution, the MagicBag is useless, and you've built and array list, which is forbidden by the problem statement.

Also of note: Some of you made notes to the effect of "1,000,000 is constant". Technically yes, but by that logic ArrayLists use constant time and memory since no array can be larger than 2,000,000,000 items, and taking it

even a bit further, ALL data structures we've discussed in the course use constant time and memory since the memory of real computers is finite.

Solution D: Magic bag has two String instance variables: *first* and *last*. Class *Helper* has two String instance variables, *item* and *previous*, and overrides the .equals method to only compare *item*. To enqueue, create a new Helper with *item* set to the new String and *previous* set to *last*, and then set *last* equal to the new String (also set *first* equal to the new String, if the MSQ is empty). To dequeue, set a variable *temp* equal to *first*, set *first* equal to MagicBag.remove(new Helper(item = first, previous = "")).previous, and return *temp*. This solution was unexpected and a very clever way of essentially creating a linked list, but without actually using a linked list or nodes. Unlike solution B, the MagicBag is essential to the functioning of this implementation, rather than an afterthought to satisfy the problem statement. Unfortunately, It did not receive full credit because it does not handle duplicate Strings in the queue correctly (e.g., if "giraffe" is put in the queue multiple times at different locations).