Name: _____

To complete this homework, you do not need to use any special software. In fact, you are encouraged to write out your answers on paper (or you may use Word, GoogleDoc, etc.). You may wish to consult the readings and lecture notes (available online) related to this material (Lectures 9-13).

This assignment is **due on Wednesday, April 5th, 2017, at 11:59pm**. Please turn it in by uploading to Gradescope. Note: if you write out your answers by hand, you will need to scan or upload a photo of your work. Be sure to leave enough time for this.

Questions

1. (10 points) Let $f(n)$ represent the runtime of an algorithm in nano seconds as a function of the input size n. So, for example, $f(n) = 1000$ means the algorithm takes one microsecond regardless of the size of the input. As another example, $f(n) = 2n + 132$ implies the algorithm on an input of size $n = 1000$ would take $2 \times 1000 + 132 = 2.132$ microseconds and on an input of size $n = 1,000,000$ would take 2.000132 milliseconds.

   For each of the following, given the runtime $f(n)$, circle the *smallest* big Oh approximation that holds.

   (a) $f(n) = 10$

   A. $O(1)$ B. $O(\lg n)$ C. $O(n)$ D. $O(n \lg n)$ E. $O(n^2)$ F. $O(n^3)$

   (b) $f(n) = 1000^3 + \lg n$

   A. $O(1)$ B. $O(\lg n)$ C. $O(n)$ D. $O(n \lg n)$ E. $O(n^2)$ F. $O(n^3)$

   (c) $f(n) = 12n^2 + 43n + 18 \lg n + 101231$

   A. $O(1)$ B. $O(\lg n)$ C. $O(n)$ D. $O(n \lg n)$ E. $O(n^2)$ F. $O(n^3)$

   (d) $f(n) = \sum_{i=1}^{n}(2 \times i)$

   A. $O(1)$ B. $O(\lg n)$ C. $O(n)$ D. $O(n \lg n)$ E. $O(n^2)$ F. $O(n^3)$

   (e) $f(n) = \log_{100} n$ (Hint: you may want to use the following fact about logarithms: $\log_b x = \log_a x / \log_a b$)

   A. $O(1)$ B. $O(\lg n)$ C. $O(n)$ D. $O(n \lg n)$ E. $O(n^2)$ F. $O(n^3)$

2. (18 points) For each of the following algorithms, identify its best- and worst-case running time. If there is a difference between them, then briefly (1-2 sentences) explain why. For example, if the algorithm "typically" runs in $O(n)$ time but for some inputs runs a lot slower, say $O(n^2)$ time, then describe the inputs that lead to the slower runtime.

   (a) Best case runtime for BinarySearch

   A. $O(1)$ B. $O(\lg n)$ C. $O(n)$ D. $O(n \lg n)$ E. $O(n^2)$ F. $O(n^3)$

   (b) Worst case runtime for BinarySearch

   A. $O(1)$ B. $O(\lg n)$ C. $O(n)$ D. $O(n \lg n)$ E. $O(n^2)$ F. $O(n^3)$

   (c) If the previous two answers are different, briefly explain why.

   (d) Best case runtime for InsertionSort

   A. $O(1)$ B. $O(\lg n)$ C. $O(n)$ D. $O(n \lg n)$ E. $O(n^2)$ F. $O(n^3)$

   (e) Worst case runtime for InsertionSort

   A. $O(1)$ B. $O(\lg n)$ C. $O(n)$ D. $O(n \lg n)$ E. $O(n^2)$ F. $O(n^3)$

   (f) If the previous two answers are different, briefly explain why.

(g) Best case runtime for MergeSort

A. $O(1)$ B. $O(\lg n)$ C. $O(n)$ D. $O(n \lg n)$ E. $O(n^2)$ F. $O(n^3)$

(h) Worst case runtime for MergeSort

A. $O(1)$ B. $O(\lg n)$ C. $O(n)$ D. $O(n \lg n)$ E. $O(n^2)$ F. $O(n^3)$

(i) If the previous two answers are different, briefly explain why.

3. (16 points) Three search algorithms – LinearSearch, SentinelLinearSearch, and BinarySearch – were described in class, on handouts (available on the course website), and in the readings from *Algorithms Unlocked*.

In this question, I want you to simulate running each algorithm on a particular array. The question is the following: which cells of the array are examined by the algorithm? If the algorithm examined a cell, please indicate this by marking the cell with an 'X'.

For example, suppose I run LinearSearch$(A, 4, 3)$ meaning that we are looking for $x = 3$ in a length 4 array A. Suppose the array A is equal to:

12	7	3	9
1	2	3	4

LinearSearch examines every cell, so we could show that as:

~~12~~	~~7~~	~~3~~	~~9~~
1	2	3	4

For each of the following, mark out the examined cells.

(a) SentinelLinearSearch$(A, 8, 31)$ where A is

23	7	16	31	12	9	8	14
1	2	3	4	5	6	7	8

(b) SentinelLinearSearch$(A, 8, 17)$ where A is

23	7	16	31	12	9	8	14
1	2	3	4	5	6	7	8

(c) BinarySearch$(A, 8, 17)$ where A is

7	8	9	12	14	16	23	31
1	2	3	4	5	6	7	8

(d) BinarySearch$(A, 8, 9)$ where A is

7	8	9	12	14	16	23	31
1	2	3	4	5	6	7	8

4. (20 points) Suppose that while waiting for Joe Biden to speak, the audience members start to get a little boisterous. You realize that it's because there are Republicans in the audience and they are sprinkled throughout the mostly Democrat audience. To make things more peaceable you decide that you will rearrange people by political affiliation. However, to avoid creating even more chaos, you want to do this quickly and in a way that doesn't create a "free for all" with everyone standing up at once and moving about.

   Assume the seats are numbered 1 to n. All seats are occupied. You can move to any seat and inquire about the political affiliation of the person sitting there, which must be either D or R (independents stayed home). You can remove someone from a seat, creating an empty seat. You can also move someone from one seat to an empty seat. Finally, you can "swap" two people. Each of these operations has a "cost" of 1 unit. At any given time only a small number of people should be without a seat.

   (a) Describe the most efficient algorithm (i.e. lowest cost) for moving all of the Republicans to the seats in the back (meaning the highest numbered seats). Please write your algorithm in the same style as the algorithms described in *Algorithms Unlocked* and on the handouts.

   (b) Analyze the runtime of your algorithm. You can give a rough approximation using big Oh notation, such as $O(\lg n)$, or $O(n)$, etc.

This space is blank for you to describe your algorithm and analyze its runtime.

5. (16 points) Your task in this question is to compute the edit distance between two strings (arrays of characters). In addition, you must identify the sequence of edit commands that achieves the edit distance. Here is an example. Suppose we want to calculate the edit distance between $s =$`'ABC'`and $t =$`'AxCD'`. First, we calculate the edit distance matrix. The first cell in the matrix, the one corresponding to (_, _), is always equal to zero.

```
   _  A  x  C  D
_  0, 1, 2, 3, 4
A  1, 0, 1, 2, 3
B  2, 1, 1, 2, 3
C  3, 2, 2, 1, 2
```

The sequence of edit commands that achieves this edit distance:

```
COPY A -> A   (cost 0)
SUB  B -> x   (cost 1)
COPY C -> C   (cost 0)
INSERT D      (cost 1)
```

Complete the following problems using the style shown in the example above. In addition to lecture notes, you may wish to consult the Wikipedia page `https://en.wikipedia.org/wiki/Levenshtein_distance`. (Please cite any additional sources.)

(a) Compute the edit distance matrix between strings $s =$`'zAzB'`and $t =$`'AB'`.

(b) Compute the sequence of edit commands that achieves the above edit distance for strings $s =$`'zAzB'`and $t =$`'AB'`. (Please follow the template above.)

(c) Compute the edit distance matrix between strings $s =$`'aAB'`and $t =$`'ABzz'`.

(d) Compute the sequence of edit commands that achieves the above edit distance for strings $s =$`'aAB'`and $t =$`'ABzz'`. (Please follow the template above.)

Algorithm 1 An algorithm for seeking location of an entry in a *sorted* array

```
1: procedure SEEK(A, x) ▷ Find x in sorted array A of length n or return location where
   x would be inserted if it's not there.
2:     Set p to 1 and r to n.
3:     while p ≤ r do
4:         Set q to ⌊(p + r)/2⌋.
5:         If A[q] = x, then return q.
6:         Otherwise, if A[q] > x, then set r to q − 1.
7:         Otherwise, it must be that A[q] < x, so set p to q + 1.
8:     end while
9:     We know x is not there. Return p.
10: end procedure
```

6. (20 points) In this question, I'm going to ask you to analyze the runtime of an algorithm. But first, let me set up the problem.

   Recall from Lecture 12 and Ch. 2 of *Nine Algorithms* the problem of finding web pages that contain certain search terms. For this problem, we assume we have the following structure. Array W is a sorted array. An entry of the array W is a pair: a word and an array of document ids. The array of documents ids represents the ids of documents where that word occurs. Each array of document ids is also sorted.

   Here is an example that matches the example on p. 15 of *Nine Algorithms*.

index	word	document id arrays
1	a	[3]
2	cat	[1, 3]
3	dog	[2, 3]
4	mat	[1, 2]
5	on	[1, 2]
6	sat	[1, 3]
7	stood	[2, 3]
8	the	[1, 2, 3]
9	while	[3]

   Let's assume that binary search can be extended to handle arrays where the entries of the array are pairs. So if I do binary search on W for word "dog", I would get back 3. Then I could retrieve the document id array for "dog." Let's call this B_{dog}. The array B_{dog} has two entries 2 and 3. So, if I index into B_{dog} at position 2, as in $B_{dog}[2]$, I would get back 3.

   The SEEK algorithm shown below is identical to BINARYSEARCH except for one key difference: if x is not in the sorted array, then it returns the index where x could be inserted while still maintaining the array in sorted order. For example, SEEK(W, "colgate") would

return 3, indicating that "colgate" could be inserted after "cat" but before "dog."

The ZigZagJoin algorithm solves the following task: given two words such as "cat" and "dog" it returns an array that contains the document ids of those documents that contains both words. In this example, it would return an array containing a single entry, document 3.

The basic idea of ZigZagJoin is the following: since the document id arrays are in sorted order, we can repeatedly use binary search to find the next matching document id. This allows to potentially skip large numbers of non-matching documents.

Algorithm 2 An algorithm for finding documents that contain two given words.

```
1: procedure ZigZagJoin(W, v, w) ▷ W is an array as described above and v and w are two words.
2:     Initialize results to be a sufficiently large empty array. Initialize r to 1.
3:     Run BinarySearch on W to find the entry for v.
4:     If v is not in W return the empty results.
5:     Otherwise, let B_v be the array of document ids for v.
6:     Run BinarySearch on W to find the entry for w.
7:     If w is not in W return the empty results.
8:     Otherwise, let B_w be the array of document ids for w.
9:     Let n_1 be the number of entries in B_v; n_2 the number in B_w.
10:    Initialize i = 1 and j = 1.
11:    while i ≤ n_1 and j ≤ n_2 do          ▷ We're not yet at end of either array
12:        Let docid_v be the document id B_v[i].
13:        Let docid_w be the document id B_w[j].
14:        If docid_v = docid_w, then both words appear in this document.
15:        Put docid_v in results[r] and increment r.
16:        Otherwise, if docid_v < docid_w, then set i equal to the result of Seek(B_v, docid_w).
17:        Otherwise, we know docid_w < docid_v. Set j equal to Seek(B_w, docid_v).
18:    end while
19:    Return results.
20: end procedure
```

Analyze the runtime of ZigZagJoin using big Oh notation. Assume the following:

- There are m words in W.
- There n_1 documents that contain word v and n_2 documents that contain word w.
- There are k documents that contain both v and w.

Hint: the correct answer includes m, n_1, n_2, and k. For full credit, give the smallest big Oh expression possible. For example, while it's true that the runtime is $O(2^{m+n_1+n_2+k})$, the runtime is considerably smaller than the function in this big-Oh expression.

This space is blank for you to analyze the runtime of ZigZagJoin. *At the top of this page, give your big-Oh expression. Then include a brief justification for it.*