COS 226	Algorithms and Data Structures	Fall 2018

Final

This exam has 16 questions (including question 0) worth a total of 100 points. You have 180 minutes. This exam is preprocessed by a computer when grading, so please **write darkly** and **write your answers inside the designated spaces.**

Policies. The exam is closed book, except that you are allowed to use a one-page cheatsheet (8.5-by-11 paper, two sides, in your own handwriting). Electronic devices are prohibited.

Discussing this exam. Discussing the contents of this exam before solutions have been posted is a violation of the Honor Code.

This exam. Do not remove this exam from this room. In the space provided, write your name and NetID. Also, mark your exam room and the precept in which you are officially registered. Finally, write and sign the Honor Code pledge. You may fill in this information now.

Name:

NetID:

Course: COS 126 ○ COS 226 ●

Exam room: McCosh 10 ○ Other ○

Precept: P01 ○ P01A ○ P02 ○ P02A ○ P03 ○ P03A ○ P04 ○ P05 ○

"I pledge my honor that I will not violate the Honor Code during this examination."

Signature

0. **Initialization. (2 point)**

   In the space provided on the front of the exam, write your name and NetID; mark your exam room and the precept in which you are officially registered; write and sign the Honor Code pledge.

1. **Empirical running time. (5 points)**

   Suppose that you observe the following running times for a program on inputs of size n for varying values of n.

n	*time*
10,000	1.2 seconds
30,000	2.1 seconds
90,000	3.9 seconds
270,000	7.9 seconds
810,000	16.0 seconds

   (a) Estimate the running time of the program (in seconds) for an input of size n = 2,430,000.

   seconds

   (b) Estimate the *order of growth* of the running time of the program as a function of n.

2. **Mathematical running time. (5 points)**

Let `list` be a `LinkedList` object containing a sequence of `n` characters. For each code fragment at left, write the letter corresponding to the order of growth of the worst-case running time as a function of n.

Java's `LinkedList` data type represents a sequence of items using a *doubly linked list*, maintaining references to the first and last nodes. All operations are implemented in an efficient manner for the given representation.

☐
```
// convert the list to a string
String s = "";
for (char c : list)
    s += c;
```

☐
```
// Knuth shuffle
for (int i = 0; i < list.size(); i++) {
    int r = (int) (Math.random() * (i + 1));
    char c1 = list.get(r);  // get element r
    char c2 = list.get(i);  // get element i
    list.set(r, c2);        // replace element r
    list.set(i, c1);        // replace element i
 }
```

☐
```
// sort (using Timsort/mergesort)
Collections.sort(list);
```

☐
```
// palindrome?
boolean isPalindrome = true;
while (list.size() > 1) {
    char c1 = list.removeFirst();
    char c2 = list.removeLast();
    if (c1 != c2) isPalindrome = false;
}
```

☐
```
// create a reverse copy of the list
LinkedList<Character> copy = new LinkedList<Character>();
for (char c : list)
    copy.addFirst(c);
```

A. 1

B. $\log n$

C. n

D. $n \log n$

E. n^2

F. n^3

3. **String sorts. (5 points)**

The column on the left contains the original input of 24 strings to be sorted; the column on the right contains the strings in sorted order; the other 5 columns contain the contents at some intermediate step during one of the 3 radix-sorting algorithms listed below. Match each algorithm by writing its letter in the box under the corresponding column.

6862	1131	5091	1131	3906	5790	1131
7924	1216	1131	1188	9608	9880	1188
1131	1188	2294	1216	8814	7270	1216
8276	2786	5790	2786	1216	1131	2294
9299	2294	1216	2294	7924	7671	2786
5790	3906	5035	3906	8424	6551	3906
1216	5790	2786	5790	1131	5091	5035
7383	5035	3906	5035	5035	6862	5091
8424	5091	1188	5091	9545	7383	5790
3906	6862	6188	6862	6551	7924	6188
9545	6551	6862	6551	9757	8424	6551
7671	6188	6551	6188	6862	8814	6862
9880	7924	9880	7924	7270	2294	7270
6551	7383	7671	7383	7671	9545	7383
1188	7671	9545	7671	8276	5035	7671
2786	7270	9608	7270	9880	8276	7924
9608	8276	8424	8276	7383	1216	8276
5035	8424	9757	8424	2786	3906	8424
9757	8814	8814	8814	1188	2786	8814
8814	9299	7383	9299	6188	9757	9299
2294	9545	9299	9545	5790	1188	9545
6188	9880	8276	9880	5091	9608	9608
5091	9608	7270	9608	2294	6188	9757
7270	9757	7924	9757	9299	9299	9880
A						E

A. Original input

B. LSD radix sort

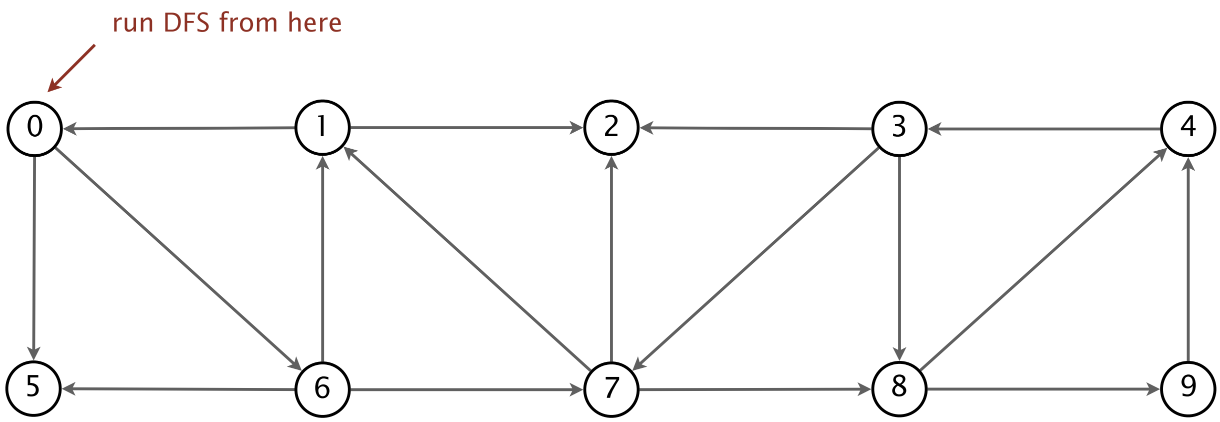
C. MSD radix sort

D. 3-way radix quicksort (*no shuffle*)

E. Sorted

4. **Depth-first search. (6 points)**

   Run depth-first search on the following digraph, starting from vertex 0. Assume the adjacency lists are in sorted order: for example, when iterating over the edges pointing from 6, consider the edge 6→1 before either 6→5 or 6→7.



   (a) List the 10 vertices in *preorder*.

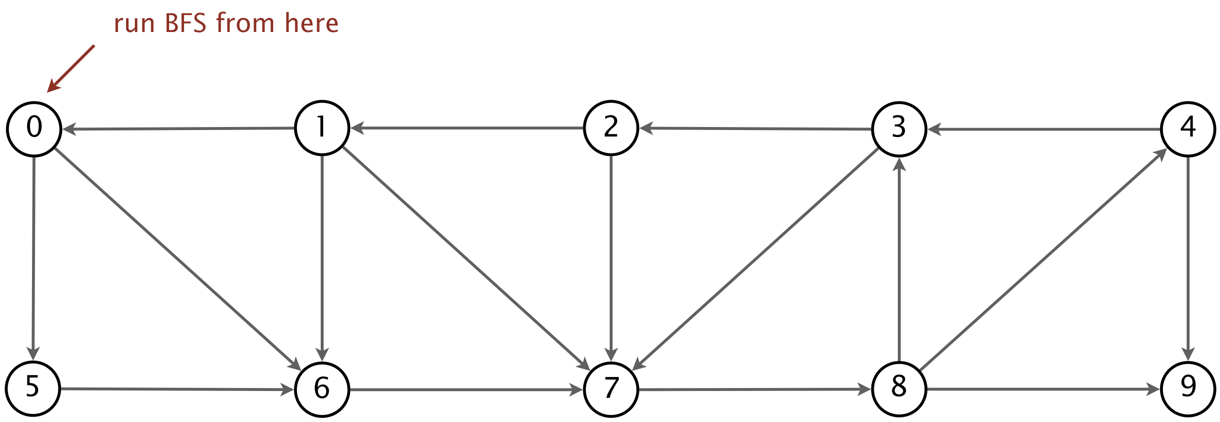
   0 ___ ___ ___ ___ ___ ___ ___ ___ ___ ___

   (b) List the 10 vertices in *postorder*.

   ___ ___ ___ ___ ___ ___ ___ ___ ___ 0

5. **Breadth-first search. (6 points)**

Run breadth-first search on the following digraph, starting from vertex 0. Assume the adjacency lists are in sorted order: for example, when iterating over the edges pointing from 8, consider the edge $8 \to 3$ before either $8 \to 4$ or $8 \to 9$.



(a) List the 10 vertices in the order in which they are added to the queue.

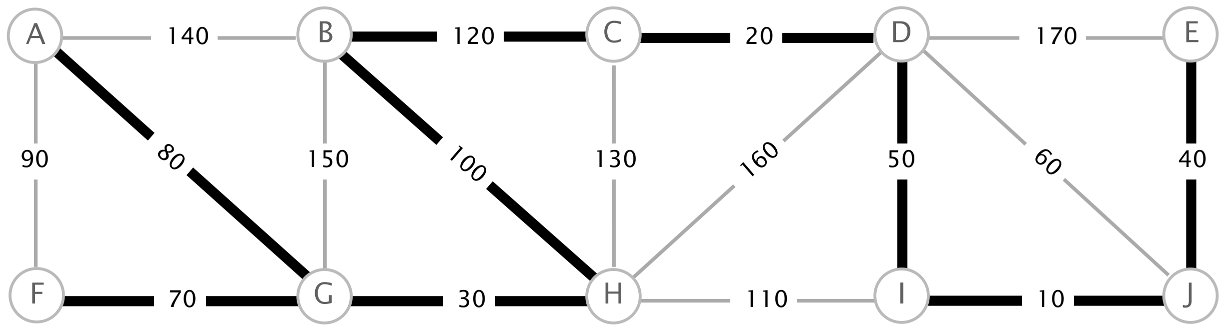
0 ___ ___ ___ ___ ___ ___ ___ ___ ___

(b) Give the entries in the `edgeTo[]` array upon termination of breadth-first search.

v	0	1	2	3	4	5	6	7	8	9
`edgeTo[v]`	–									

6. **Minimum spanning tree. (6 points)**

Consider the following edge-weighted graph G containing 10 vertices and 17 edges. The thick black edges T define a spanning tree of G but *not* a minimum spanning tree of G.



(a) Find a cut in G whose minimum weight crossing edge is *not* an edge in T.
Mark the vertices on the side of the cut containing vertex A.

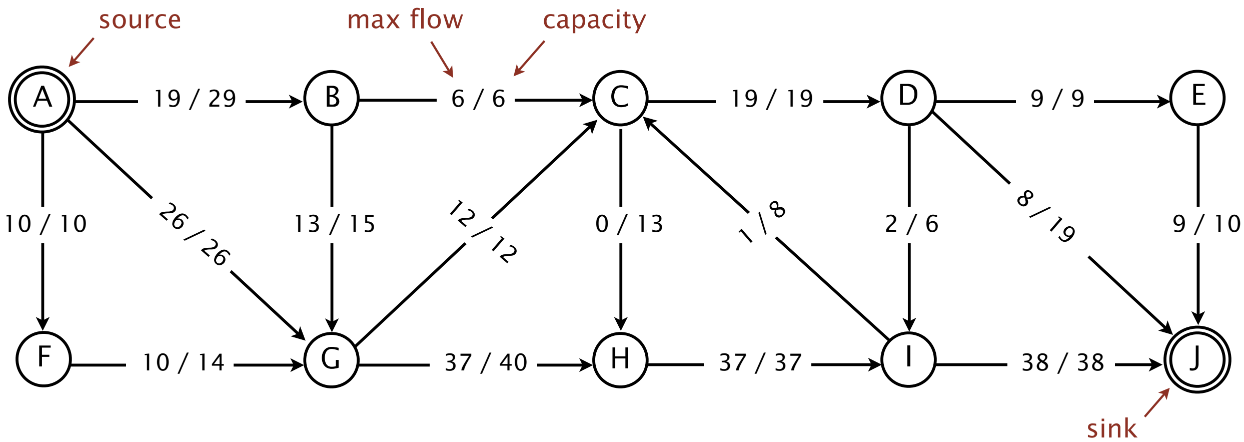
A	*B*	*C*	*D*	*E*	*F*	*G*	*H*	*I*	*J*
■	□	□	□	□	□	□	□	□	□

(b) Which of the following edges are in the MST of G? Mark all that apply.

A–B	*B–C*	*B–G*	*B–H*	*C–H*	*D–H*	*D–I*	*D–J*	*H–I*
□	□	□	□	□	□	□	□	□

7. Maximum flow. (8 points)

Consider the following flow network and *maximum flow* f^*.



(a) What is the *value* of the flow f^*?

(b) What is the *capacity* of the cut $\{A, B, C\}$?

(c) What is the *net flow* across the cut $\{A, B, C\}$?

(d) Which vertices are on the source side of the *minimum cut*? Mark all that apply.

A	B	C	D	E	F	G	H	I	J
■	□	□	□	□	□	□	□	□	□

(e) Mark each edge below if increasing its capacity by 1 would increase the value of the maximum flow by exactly 1.

$A \to F$	$A \to G$	$B \to C$	$I \to C$	$I \to J$	$H \to I$
□	□	□	□	□	□

8. **Huffman compression. (6 points)**

Consider running Huffman compression over an alphabet of 16 characters with a given *frequency distributions* of characters (i.e., entry i is how many times character i appears in the input). For each frequency distribution below, write the *length of the longest codeword*.

$\square$ $\{ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 \}$

(equal frequencies)

$\square$ $\{ 1, 2, 4, 8, 16, 32, 64, 128, \ldots, 2^{15} \}$

(powers of 2)

$\square$ $\{ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, \ldots, 16 \}$

(positive integers)

$\square$ $\{ 1, 1, 2, 3, 5, 8, 13, 21, 34, \ldots, 987 \}$

(Fibonacci numbers)

9. **LZW compression. (6 points)**

Compress the following string of length 15 using LZW compression.

`A A B C B C A B B B C B C A C`

As usual, assume that the original encoding table consists of all 7-bit ASCII characters and uses 8-bit codewords. Recall that codeword 80 is reserved to signify end of file.

(a) Give the resulting sequence of 11 two-digit hexadecimal integers in the space below.

41										80

(b) Which of the following strings are in the LZW dictionary upon termination of the algorithm? Mark all that apply.

AA	AB	ABB	ABBB	ABC	BB	BC	BCA	BCAC	BCB	BCBC	CB
☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2	SP	!	"	#	$	%	&	‘	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

For reference, this is the hexadecimal-to-ASCII conversion table from the textbook.

10. **Knuth–Morris–Pratt substring search. (6 points)**

Consider the Knuth–Morris–Pratt DFA for the string

`C C A C C A C B`

over the alphabet { `A`, `B`, `C` }.

(a) In which state is the DFA after consuming the following sequence of characters?

`A B C`

○ 0 ○ 1 ○ 2 ○ 3 ○ 4 ○ 5 ○ 6 ○ 7 ○ 8

(b) In which state is the DFA after consuming the following sequence of characters?

`C C B A C C A C C`

○ 0 ○ 1 ○ 2 ○ 3 ○ 4 ○ 5 ○ 6 ○ 7 ○ 8

(c) In which state is the DFA after consuming the following sequence of characters?

`A C C C C A C C A C C C A C C C A C C C A C C A C C A C C A C`

○ 0 ○ 1 ○ 2 ○ 3 ○ 4 ○ 5 ○ 6 ○ 7 ○ 8

11. **Properties of shortest paths. (6 points)**

For each statement at left, identify whether it is a property of *Dijkstra's algorithm* and/or the *Bellman–Ford algorithm* by writing the letter corresponding to the best-matching term at right.

Assume that the digraph has positive edge weights and that all vertices are reachable from the source vertex s. Recall that relaxing a vertex v means relaxing every edge pointing from v. As usual, E denotes the number of edges and V denotes the number of vertices.

☐ Each vertex is relaxed at most once.

☐ Throughout the algorithm, `distTo[v]` is either infinite or the length of some directed path from s to v.

☐ When relaxing edge $v \to w$, `distTo[w]` either remains unchanged or decreases.

☐ If the length of the shortest path from s to v is less than the length of the shortest path from s to w, then vertex v is not the last vertex relaxed.

☐ In the *worst case*, the order of growth of the running time is EV.

☐ In the *best case*, the order of growth of the running time is $E + V$.

A. *Dijkstra's algorithm* (using a binary heap for PQ)

B. *Bellman–Ford algorithm* (queue-based implementation)

C. *Both A* and *B*.

D. *Neither A* nor *B*.

12. **Why did we do that? (8 points)**

For each pair of algorithms or data structures, identify a critical reason why we prefer the first to the second. Mark the best answer.

☐ Use a *queue* instead of a *stack* to store the vertices to be processed during breadth-first search of a graph.

☐ Use *reverse postorder traversal* instead of *preorder traversal* to compute a topological order in a DAG.

☐ Process the edges in *ascending order* of weight in Kruskal's algorithm instead of *descending order*.

☐ Use *Knuth–Morris–Pratt* instead of *brute-force* for substring search.

☐ Use a *stable* sorting algorithm (key-indexed counting) instead of an *unstable* one to rearrange the strings as a subroutine of LSD radix sort.

☐ Form an array of `Suffix` objects instead of an array of `String` objects when suffix sorting a string.

☐ Use a *ternary-search trie* instead of a *256-way trie* for a string symbol table over the extended ASCII alphabet.

☐ Initialize `right[c]` in Boyer–Moore to contain the index of the *rightmost* occurrence of character `c` instead of the *leftmost* occurrence.

A. Guarantees correctness.

B. Improves worst-case running time.

C. Uses less memory.

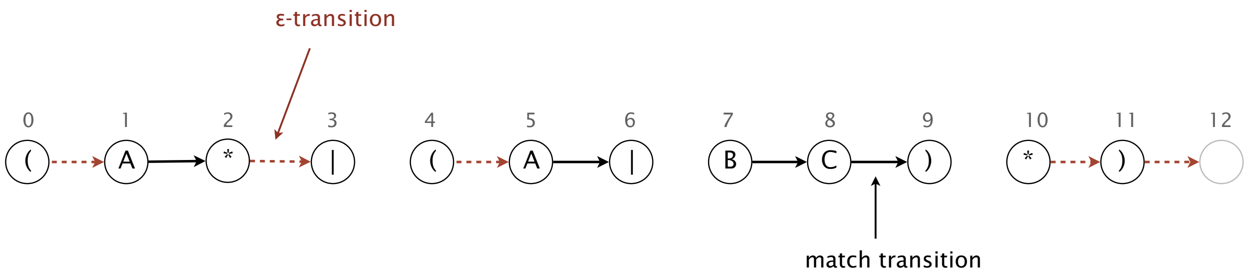
D. Simpler to code.

13. **Regular expressions. (6 points)**

Consider the NFA that results from applying the RE-to-NFA construction algorithm from lecture and the textbook to the regular expression

```
( A * | ( A | B C ) * )
```

The states and match transitions (solid lines) are shown below, but most of the ϵ-transitions (dotted lines) are suppressed.



(a) Which of the following are edges in the full ϵ-transition digraph? Mark all that apply.

0→1	0→3	0→4	0→9	1→2	1→4	2→1	3→0	3→11	4→7
■	□	□	□	□	□	□	□	□	□

4→9	4→10	6→7	6→9	7→6	9→4	9→10	9→12	10→2	10→4
□	□	□	□	□	□	□	□	□	□

(b) Suppose that you simulate the NFA with the following input:

```
A A A A A A A A
```

In which states could the NFA be after consuming the entire input? Mark all that apply.

0	1	2	3	4	5	6	7	8	9	10	11	12
□	□	□	□	□	□	□	□	□	□	□	□	□

14. **Prefix count data structure. (10 points)**

Design a data structure that supports *inserting* strings and *prefix-count queries.* A *prefix-count query* returns the number of strings inserted into the data structure (including duplicates) that start with a given prefix. To do so, describe how to implement this API:

public class PrefixCount		
	`PrefixCount()`	*create an empty data type*
`void`	`insert(String s)`	*add the string to the data structure*
`int`	`prefixCount(String prefix)`	*number of strings that start with prefix*

Here is an example:

```
PrefixCount pc = new PrefixCount();
pc.insert("ANNA");
pc.insert("BELLA");
pc.insert("ANNABELLA");
pc.insert("AN");
pc.prefixCount("ANNA");     //  2
pc.prefixCount("BELL");     //  1
pc.insert("ANNA");          // duplicate
pc.insert("ANNABEL");
pc.prefixCount("ANNA");     //  4
pc.prefixCount("BANANA");   //  0
```

Your answer will be graded for correctness, efficiency, and clarity (but not precise Java syntax). For full credit, the `PrefixCount` *constructor must take constant time;* `insert()` *must take time proportional to* RL *(or better); and* `prefixCount()` *must take time proportional to* L *(or better), where* L *is the length of the string argument and* R *is the alphabet size.*

(a) In the space below, declare the Java instance variables for your `PrefixCount` data type *using Java code*. You may define nested classes and/or use any of the data types that we have considered in this course (either `algs4.jar` or `java.util` versions).

```
public class PrefixCount {

}
```

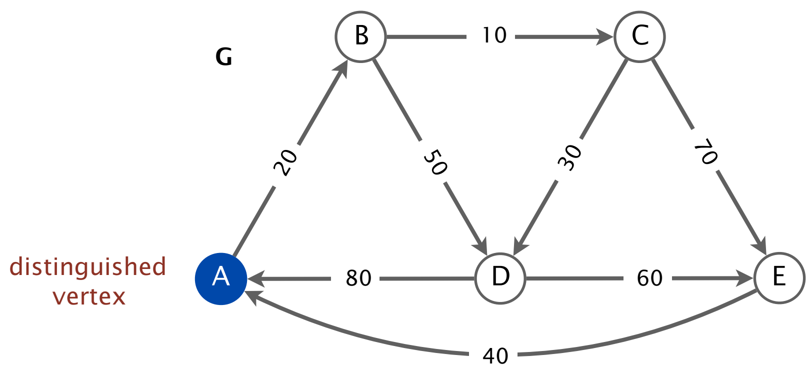
(b) Describe how to implement `insert()`, using either Java code or concise prose. If it is similar to an algorithm that we implemented in class, just say so and focus your answer on the part that is different.

(c) Describe how to implement `prefixCount()`, using either Java code or concise prose. If it is similar to an algorithm that we implemented in class, just say so and focus your answer on the part that is different.

15. **Shortest directed cycle containing a given vertex. (9 points)**

Given a digraph G with positive edge weights and a distinguished vertex s, design an algorithm to find a shortest directed cycle that contains s (or report that no such cycle exists). To do so, solve a source–sink shortest-paths problem on a related edge-weighted digraph.

For full credit, the order of growth of running time must be $E \log V$ (or better) in the worst case, where E is the number of edges and V is the number of vertices. For simplicity, assume no parallel edges or self loops.



The shortest directed cycle containing A is A–B–C–D–A and has weight 140 (20 + 10 + 30 + 80).

(a) Draw the source–sink shortest-paths problem that you would construct in order to find the shortest directed cycle containing A in the 5-vertex digraph shown above. Be sure to label the source and sink vertices and include the edge weights.

(b) Give a crisp and concise English description of your algorithm in the space below.

Your answer will be graded for correctness, efficiency, and clarity.