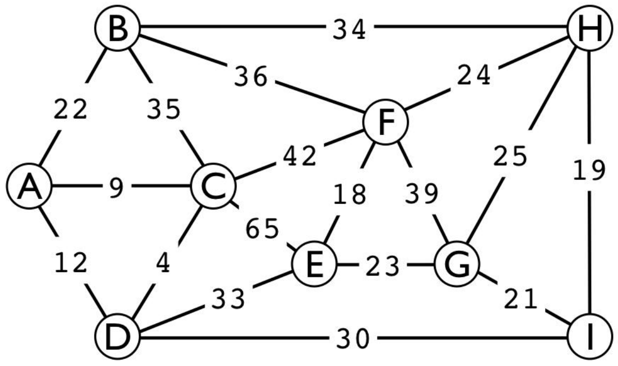
NAME: **login ID:**
precept #:

COS 226 Final Exam, Spring 2010

This test is 16 questions, weighted as indicated. The exam is closed book, except that you are allowed to use a one page cheatsheet. No calculators or other electronic devices are permitted. Give your answers and show your work in the space provided. *Put your name, login ID, and precept number on this page* (*now*), and write out and sign the Honor Code pledge before turning in the test. You have *three hours* to complete the test.
"I pledge my honor that I have not violated the Honor Code during this examination."

1.	MST	/4		
2.	KMP	/4		
3.	DFS trace	/6		
4.	acronyms	/7		
5.	LZW	/7		
			Subtotal	/28
6.	TST	/5		
7.	String ST	/6		
8.	RE I	/6		
9.	RE II	/4		
10.	String sort	/7		
			Subtotal	/28
11.	Graph algs	/8		
12.	Graph memory	/5		
13.	3-way	/8		
14.	Tree encoding	/5		
15.	TwitStream	/9		
16.	Hard problems	/9		
			Subtotal	/44
			TOTAL	/100

1. **MST** (4 points). By reversing the sense of the comparator, you can use Prim's algorithm and Kruskal's algorithm to find the *maximum* spanning tree of a weighted graph. Consider the following graph:



A. Give the list of edges in the maximum spanning tree in the order that *Kruskal's algorithm* inserts them. For reference, the 18 edge weights are listed here:

```
4 9 12 18 19 21 22 23 24 25 30 33 34 35 36 39 42 65
```

B. Give the list of edges in the maximum spanning tree in the order that *Prim's algorithm* inserts them, assuming that it starts at vertex A..

2. **KMP** (4 points). The following is a KMP state-transition table for a 9-character string.
A. Write the characters in the string in the blanks below the table.

	0	1	2	3	4	5	6	7	8
A			3						9
E		2		4		6		8	
T	1				5		7		

____ ____ ____ ____ ____ ____ ____ ____ ____

B. Fill in the blanks in the table.

3. **DFS trace** (6 points). Consider the following recursive depth-first search implementation for directed graphs. Assume that `Digraph G` is an instance variable of the class.

```
private void dfs(int v)
{
   marked[v] = true;
   for (int w : G.adj(v))
      if (!marked[w]) dfs(w);
}
```

At left is a trace of DFS for the call `dfs(0)` in a certain digraph (made by instrumenting the `if` statement to print `check w` for marked vertices and `dfs(w)` for unmarked vertices, and adding a statement to print `done v` as the last statement of `dfs()`, all with appropriate indenting). To the right of the trace, draw the digraph and give its adjacency lists. Then give a trace in the same style for the call `dfs(3)` in that digraph.

A (1 points). Digraph drawing **B** (2 points). Adjacency lists

```
dfs(0)
  dfs(1)
    dfs(2)
      dfs(3)
        check 0
        check 1
      3 done
      check 1
    2 done
    check 0
  1 done
  check 2
0 done
```

C (3 points). Trace of `dfs(3)`

```
dfs(3)
```

4. **Acronyms** (7 points). Each of the descriptions below fits a commonly-used three-letter acronym. Write the acronym corresponding to each description in the blank to its right.

   A. Abstract machine, basis for KMP algorithm. ______

   B. Data structure for implementing symbol tables. ______

   C. Substring-search algorithm. **_KMP_**

   D. Fundamental recursive method. ______

   E. For single-source shortest paths in unweighted graphs. ______

   F. Symbol table-based compression algorithm. ______

   G. Fundamental search problem, from logic. ______

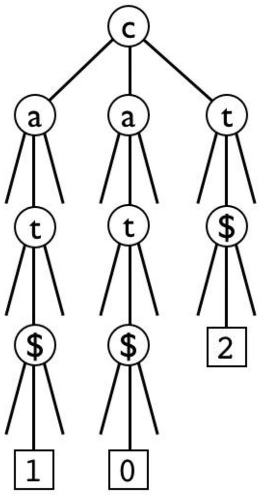
   H. Abstract machine, basis for grep ______

5. **LZW compression** (7 points). Complete the line labeled `out` and the following table for computing the LZW compression of the string `B A N D A N A B A N A N A`.

```
in:   B   A   N   D   A   N   A   B   A   N   A   N   A
out:  42  41  4E                                           80
```

key	value
B A	81
	82
	83
	84
	85
	86
	87
	88
	89

6. **Suffix TST** (5 points). The suffix TST corresponding to a string is constructed by building the TST corresponding to the string suffixes, including on each an end-of string character `$` that is larger than every string character. Thus, every path through the TST ends in a `$`, below which we put an external node corresponding to the starting point of the suffix. For example, the suffix TST corresponding to the string `cat` is drawn at left.



Draw the suffix TST corresponding to the string `babab` using the same order as the example above.

7. **String symbol table** (6 points). Match each of the given search data structures with one or more of the given characteristics, where N is the number of keys, L is the average number of characters in a key, and R is the size of the alphabet. Write as many letters (in alphabetical order) as apply in the blank to the left of the name of the data structure. Assume that the keys were inserted in random order. The average search time referred to in B, D, and E is for a random *successful* search (assuming that each key in the structure is equally likely to be the search key).

   A. tree/trie shape is dependent on the order in which keys are inserted

   B. average search time is ~*c*log N for some constant *c*

   C. space usage is ~*c*NR for some constant *c*

   D. average search time ~*c*L for some constant c

   E. asymptotic average search time cannot be determined from the information given

________	R-ary trie
________	BST
________	TST
________	red-black tree

8. **RE pattern matching I** (6 points). Draw an NFA (nondeterministic finite state automata) that recognizes the same language that the regular expression `((AB* | C )* | D* )` describes. Use the notation and construction given in lecture and the book (if you don't have a red pencil, mark epsilon transitions as dotted lines). Circle your final answer.

9. **RE pattern matching II** (4 points). Why do we not use a DFA (deterministic finite state automata) instead of an NFA (nondeterministic finite state automata) to implement RE pattern matching? Circle one of the following choices.

   A. NFAs are easier to simulate than DFAs.

   B. More than one DFA might correspond to each RE.

   C. We do not know how to compute a DFA corresponding to a given RE.

   D. We do know how to compute a DFA corresponding to a given RE, but not one guaranteed to have a reasonable number of states.

   E. NFAs lead to simpler code.

10. **7 sorting algorithms** (7 points). The leftmost column is the original input of strings to be sorted, and the rightmost column is the sorted result. The other columns are the contents at some intermediate step during one of the 7 sorting algorithms listed below. Match up each algorithm by writing its letter under the corresponding column. Use each letter exactly once.

swab	ably	abba	ably	ably	abba	abet	ably	abba
ably	babe	ably	abys	babe	baba	ably	babe	abed
babe	able	abut	abba	able	swab	babe	swab	abet
wabs	blab	abys	abut	blab	blab	able	wabs	able
blab	cabs	babe	abed	cabs	drab	blab	baby	ably
cabs	baby	baby	abet	baby	flab	cabs	blab	abut
baby	dabs	blab	able	dabs	slab	baby	cabs	abys
dabs	abys	cabs	babe	abys	grab	dabs	dabs	baba
abys	drab	dabs	blab	drab	crab	abys	abys	babe
drab	flab	drab	baby	flab	scab	drab	drab	baby
flab	abba	flab	baba	sabs	stab	flab	flab	blab
sabs	gabs	gabs	cabs	abba	abed	sabs	sabs	cabs
abba	abut	sabs	crab	gabs	babe	abba	abba	crab
gabs	baba	swab	dabs	abut	able	gabs	abut	dabs
abut	gaby	wabs	drab	baba	wabs	abut	baba	drab
baba	grab	baba	flab	gaby	cabs	baba	gabs	flab
gaby	jabs	gaby	gabs	slab	dabs	gaby	gaby	gabs
slab	labs	grab	gaby	grab	abys	slab	grab	gaby
grab	abed	jabs	grab	stab	sabs	grab	slab	grab
tabu	nabs	labs	jabs	jabs	gabs	stab	tabu	jabs
jabs	crab	slab	labs	labs	jabs	jabs	abed	labs
labs	abet	tabu	nabs	abed	labs	labs	jabs	nabs
abed	sabs	abed	swab	nabs	nabs	abed	labs	sabs
nabs	stab	abet	sabs	crab	tabs	nabs	nabs	scab
crab	swab	able	slab	scab	abut	crab	abet	slab
scab	scab	crab	scab	abet	abet	scab	crab	stab
abet	slab	nabs	stab	swab	tabu	swab	scab	swab
stab	tabs	scab	tabu	tabs	ably	tabu	stab	tabs
tabs	tabu	stab	tabs	tabu	baby	tabs	able	tabu
able	wabs	tabs	wabs	wabs	gaby	wabs	tabs	wabs
	____	____	____	____	____	____	____	

A. Mergesort
B. MSD string sort
C. LSD string sort
D. Quicksort (with no random shuffle)
E. Bottom-up mergesort
F. Quicksort with 3-way partitioning (with no random shuffle)
G. 3-way string quicksort

11. **Graph algorithms** (8 points). Match each of the ideas listed below to the named algorithms by writing a letter to the left on each algorithm name. Two letters get used twice; one blank gets two letters.

A. Relax edges until no relaxation is possible.

B. Sort edges by their weight, then check connectivity.

C. Keep vertices for which the DFS is not yet complete on a stack.

D. Use DFS of the reverse digraph to provide a vertex-checking order for a standard DFS.

E. Put vertices on a stack just before completing a recursive DFS.

F. Add vertices one by one to a growing tree.

______ Prim

______ Kosaraju

______ Bellman-Ford

______ Directed cycle detection

______ Topological sort

______ Kruskal

______ Dijkstra

12. **Graph space usage** (5 points). Analyze the following code in terms of its memory usage with respect to *V* (number of vertices) and *E* (number of edges). For partial credit, make sure to fill in the blanks to the right of each declaration with its associated memory usage. Assume that object overhead is 8 bytes , that `int` values and references use 4 bytes, and that an `Integer` is 12 bytes. Note that the graph is undirected. Make sure that your final answer is concise and clear.

```
public class Graph
{
   private final int V;                                            _________
   private final Bag<Integer>[] adj;                               _________
   public Graph(int V)
   {
      this.V = V;
      adj = (Bag<Integer>[]) new Bag[V];
      for (int v = 0; v < V; v++)
      adj[v] = new Bag<Integer>();
   }
   public void addEdge(int v, int w)
   {  adj[v].add(w);  adj[w].add(v); }
...
}
public class Bag<Item>
{
   private Node first = null;
   private class Node                                              _________
   {  Item item; Node next;  }
   public void add (Item item)
   {
      Node oldfirst = first;
      first = new Node();
      first.item = item;
      first.next = oldfirst;
   }
...
}
```

Total memory usage for graph with *V* vertices and *E* edges ________________________

13. **3-way partitioning** (8 points). Consider running the 3-way partitioning algorithm we studied in class on a (sub-)array of length N. Assume that elements less than, equal to, and greater than the pivot are *distributed uniformly* throughout the array (i.e., if there are k elements greater than the pivot, then $\sim k/2$ of them are in each half of the array, $\sim k/4$ are in each quarter, and so on). Give the expected number of compares and exchanges performed by the algorithm in each of the following scenarios. Use tilde notation, so that you can give answers like $\sim N/3$ and $\sim N$ instead of exact answers like $N - 1$ or $N/2 + 1$.

A. Half of the elements in the array are less than the pivot element, while half are greater (and none are equal).

Compares: _____~**N**______ Exchanges:_____________

B. Half of the elements in the array are equal to the pivot element, while a quarter are less and a quarter are greater.

Compares: _____________ Exchanges:_____________

C. All of the elements in the array are equal to the pivot element.

Compares: _____~**N**______ Exchanges:_____`0`______

Now consider an alternative partitioning algorithm that operates as follows: First, perform a 2-way partition on the array, in which the scans *do not* stop on elements equal to the pivot. This gives you two parts, with elements <= and >= the pivot. Then, perform a 2-way partition on just the first part of the array (elements classified as <= the pivot), modified so that elements equal to the pivot are moved to the end (of that part of the array), while elements strictly less than the pivot are moved to the beginning. Similarly, partition the second part of the array, moving the elements greater than the pivot to the end and elements equal to the pivot to the beginning. Just as with the algorithm from class, the result should be that elements less than the pivot are at the beginning, equal in the middle, and greater at the end. Give the expected number of compares and exchanges carried out by this algorithm under scenarios A, B, and C above.

D. (scenario A) Compares: ___________ Exchanges:_____________

E. (scenario B) Compares: ___________ Exchanges:_____________

F. (scenario C) Compares: ___________ Exchanges:______`0`______

14. **Tree encoding** (5 points). A *binary tree with leaves* is a binary tree where null links only appear in *leaves*, which are nodes with both links null. A *preorder encoding* of a binary tree with leaves is created by traversing the tree in preorder, writing 0 when an internal node is first encountered and 1 when a leaf is first encountered. This encoding is like the encoding we used in Huffman encoding, omitting the character codes.

    A. (2 points) Using circles for internal nodes and squares for leaves, draw the tree encoded by

    ```
    0 0 0 1 1 1 0 1 0 0 1 1 1
    ```

    B. (3 points) Give a simple rule for determining whether a binary string is a legal preorder encoding of a binary tree with leaves. Your answer should be *one or two sentences.*

15. **Burrows-Wheeler** (9 points). Your first task at a hot new startup is to make a newly acquired whole-Twitter-archive called the TwitStream easily searchable: for any given query string, you are to return the number of instances of that query string in the TwitStream. Since Twitter is highly repetitive and the TwitStream is extremely large, it is critical that clients can count the number of occurrences in *time proportional only to the length of the query string*, without any dependence on the number of occurrences of query string. Luckily for you, when you were doing your Burrows-Wheeler assignment in COS 226 you noticed a way that the BWT could be used to solve the problem, as long as you have the following data structures, which can be computed in a single preprocessing pass through the TwitStream requiring time and space proportional to its length.

```
s[i] = character i in the sorted list
       i = 0...N-1

t[i] = character i in the Burrows Wheeler Transform
       i = 0...N-1

sIndex[c] = first index of s that contains character c
            c = 0...R-1, with sIndex[R] = N

tCount[c,i] = count of char c from t[0] ... t[i]
              i= -1...N-1, c = 0...R-1
```

```
A B R A C A D A B R A !
                                          tCount[,]
 i  s   Sorted Suffixes (fyi)   t     ! A B C D R
--  -  -----------------------        -----------
-1                                    0 0 0 0 0 0
 0  !  ! A B R A C A D A B R A  A     0 1 0 0 0 0
 1  A  A ! A B R A C A D A B R  R     0 1 0 0 0 1
 2  A  A B R A ! A B R A C A D  D     0 1 0 0 1 1
 3  A  A B R A C A D A B R A !  !     1 1 0 0 1 1
 4  A  A C A D A B R A ! A B R  R     1 1 0 0 1 2
 5  A  A D A B R A ! A B R A C  C     1 1 0 1 1 2
 6  B  B R A ! A B R A C A D A  A     1 2 0 1 1 2
 7  B  B R A C A D A B R A ! A  A     1 3 0 1 1 2
 8  C  C A D A B R A ! A B R A  A     1 4 0 1 1 2
 9  D  D A B R A ! A B R A C A  A     1 5 0 1 1 2
10  R  R A ! A B R A C A D A B  B     1 5 1 1 1 2
11  R  R A C A D A B R A ! A B  B     1 5 2 1 1 2
                           sIndex[]   0 1 6 8 9 10 12
```

Your task is to figure out how to use these data structures to perform substring search queries in time proportional to the length of the query string, by answering the questions on the next page.

A. (1 point) The entry `tcount[c, N-1]` gives the number of occurrences of search string that consists of a single char `c`. Give another expression for that value. Answer with *one line of code.*

B. (2 points) All suffixes that start with the same string will sort to consecutive indices. Suppose that for a string `S`, you know that the first index that starts with that string is `first` and that the last index that starts with that string is `last`. How many occurrences are there of the string `cS`, which is the string `S` prefixed by the char `c`? Assume that `cS` appears in the TwitStream. Answer with *one line of code* involving `first, last, tCount,` and `c.` Be careful with the indices!

C. (2 points) Suppose that you know that the first suffix that starts with the string S has index `first`. If you add the single char `c` to the front of the string `S`, to make a new string `cS`, what is `first_cs`, the first index of the a suffix that starts with `cS`? Answer with *one line of code.*

D. (1 point) Similarly, if for a string `S` you know that the largest index of a suffix that starts with `S` is `last`, what is the last index, `last_cs`, of the string `cS`? Answer with *one line of code.*

E. (3 points) How can you calculate the number of occurrences of a string in time proportional to its length? Answer with *two or three sentences.*

16. **Hard problem identification** (9 points). This question is in regard to your new job working for a software technology company. Your boss (having been told by you on the basis on your 126 knowledge that the company had better not bet its future on developing an application that finds an optimal tour connecting a set of cities) is still looking for a challenging project for you. Your boss is willing to invest in trying to solve problems that might be difficult, but not problems that we know to be impossible to solve or that we believe to be intractable. On the basis of your 226 knowledge, which of the following ideas can you tell your boss to forget about? Circle all that apply.

A. A regular expression that describes strings with balanced parentheses

B. An algorithm that guarantees to compress any given file by at least one percent

C. A linear-time algorithm for finding the convex hull of a set of points in the plane that can only compare the distances between two points

D. A linear-time algorithm for finding the MST of a graph with positive edge weights

E. A linear-time algorithm for sorting an array of `double` values.

F. A fast poly-time algorithm for bipartite matching

G. A fast poly-time algorithm for finding a maximal set of vertices in a graph such that no two of them are connected by an edge.

H. A linearithmic algorithm for finding one triple `a`, `b`, `c` such that `a*b=c` in an array of `double` values.

I. A linear-time algorithm for the maxflow problem.