X) 10 points : Radix Sort

Perform a radix sort of this list of numbers, using a radix of 10, into ascending order:

`620   696   298   395   568   971   29    41    531   21`

Show the bin/bucket sort conducted in each pass of the radix sort (i.e., as a table).

Write and circle the order of the numbers after each pass of the radix sort.

0	1	2	3	4	5	6	7	8	9
620	971				395	696		298	29
	41							568	
	531								
	21								

620 971 41 531 21 395 696 298 568 29

		620	531	41		568	971		395
		21							696
		29							298

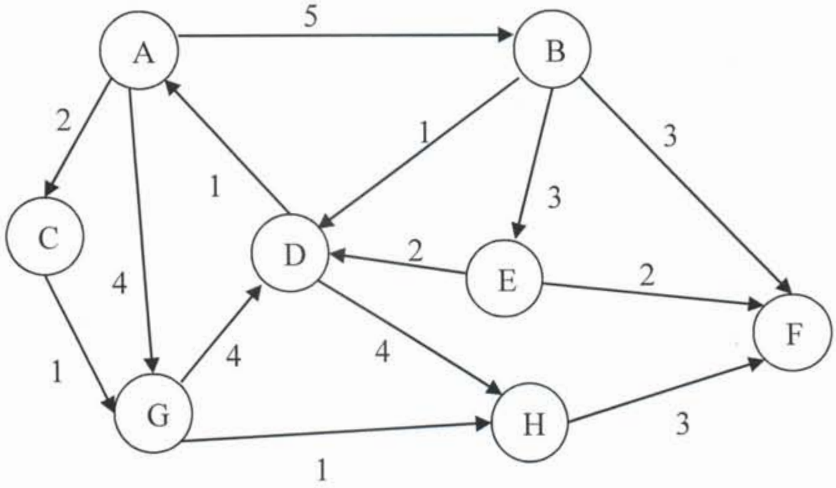
620 21 29 531 41 568 971 395 696 298

21		298	395		531	620			971
29					568	696			
41									

21 29 41 298 395 531 568 620 696 971

X) 10 points : Graph Representation

Consider the following directed, weighted graph:



a) Draw an adjacency matrix representation of the above graph.

	To:	A	B	C	D	E	F	G	H
	A		5	2				4	
	B				1	3	3		
	C							1	
From:	D	1							4
	E				2		2		
	F								
	G				4				1
	H						3		

b) Provide an appropriately tight O (Big-Oh) bound on the time for:

For a given vertex pair (v_1, v_2), testing whether there is an edge from v_1 to v_2: O(1)

Computing the in-degree of a given vertex: O(V)

Enumerating the vertices adjacent to a given vertex: O(V)

c) Draw an adjacency list representation of the above graph.

A → B | 5 → C | 2 → G | 4
B → D | 1 → E | 3 → F | 3
C → G | 1
D → A | 1 → H | 4
E → D | 2 → F | 2
F
G → D | 4 → H | 1
H → F | 3

d) Provide an appropriately tight O (Big-Oh) bound on the time for:

For a given vertex pair (v_1, v_2), testing whether there is an edge from v_1 to v_2: $O(d)$ → or $O(V)$

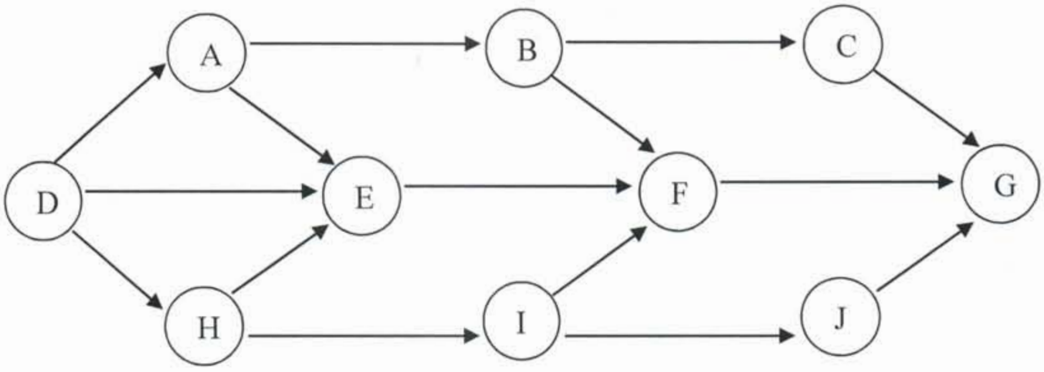
Computing the in-degree of a given vertex: $O(E)$

Enumerating the vertices adjacent to a given vertex: $O(d)$ → or $O(V)$

d = average out-degree

X) 10 Points : Topological Sort

Consider the following directed graph:



You will perform two topological sorts on this graph.

In each sort, maintain a "bag" of "pending" vertices. When the processing of a vertex creates more than one new "pending" vertex, add the new "pending" vertices to the "bag" in alphabetical order (e.g., `push(x), push(y), push(z)`).

For each topological sort, write and circle the order the graph's vertices are processed. Show your work to allow partial credit (e.g., show adding and removing from the "bag").

a) Perform a topological sort using a **queue** to maintain the set of "pending" vertices:

A	B	C	D	E	F	G	H	I	J
1	1	1	0	3	3	3	1	1	1
0	0	0		2	2	2	0	0	0
				1	1	1			
				0	0	0			

Q: D A H B E I C F J G

$\boxed{\text{D A H B E I C F J G}}$

b) Perform a topological sort using a **stack** to maintain the set of "pending" vertices:

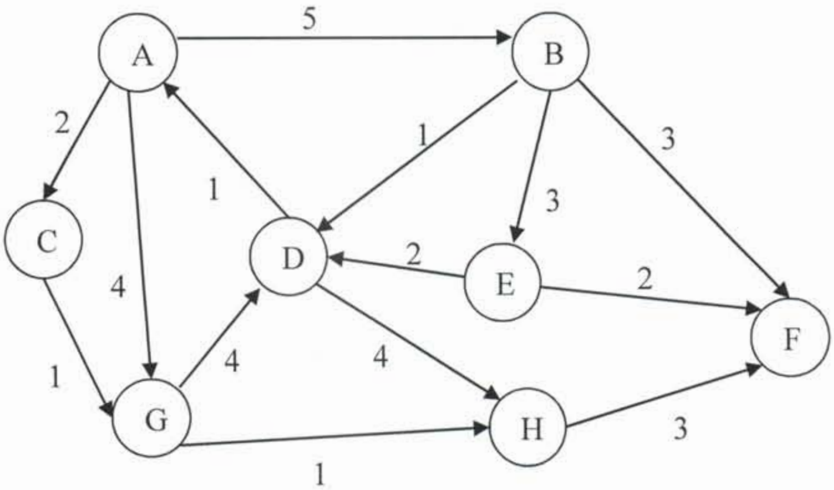
A	B	C	D	E	F	G	H	I	J
1	1	1	0	3	3	3	1	1	1
0	0	0		2	2	2	0	0	0
				1	1	1			
				0	0	0			

S: $\overset{1}{D}$ $\overset{5}{A}$ $\overset{2}{H}$ $\overset{3}{I}$ $\overset{4}{J}$ $\overset{7}{B}$ $\overset{6}{E}$ $\overset{9}{C}$ $\overset{8}{F}$ $\overset{10}{G}$

$\boxed{\text{D H I J A E B F C G}}$

X) 10 points : Single-Source Shortest Paths

Consider the following directed, weighted graph:



a) Step through Dijkstra's algorithm to calculate the single-source shortest paths from vertex *A* to every other vertex. Show your steps in the table below. Cross out old values and write in new ones, from left to right in each cell, as the algorithm proceeds. Also list the vertices in the order which Dijkstra's algorithm marks them known:

Order vertices marked as known: A C G H B D F E

6

Vertex	Known	Distance	Path
A	x	0	–
B	x	5	A
C	x	2	A
D	x	7 6	G B
E	x	8	B
F	x	7	H
G	x	4 3	A [illegible] C
H	x	4	G

b) What is the lowest-cost path from *A* to *F* in the graph, as computed above?

A → C → G → H → F

2

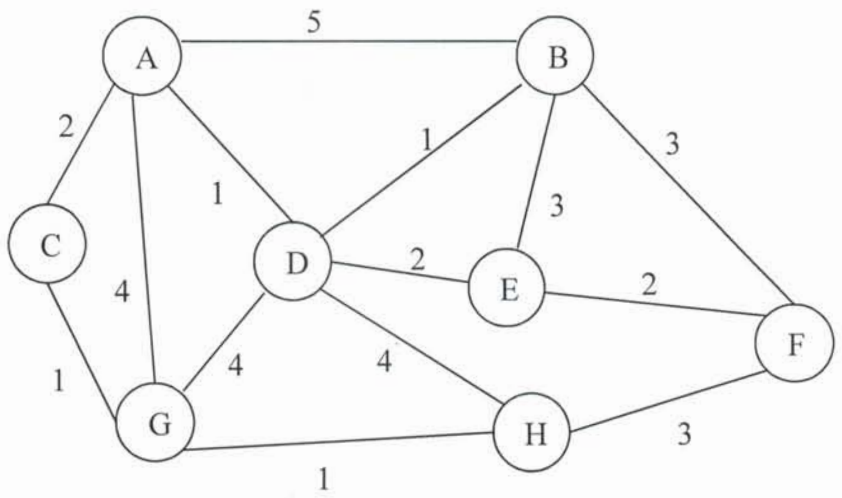
c) To guarantee correctness of Dijkstra's algorithm, all edge costs must be non-negative. Imagine the edge from A to B had cost -3. Why would this make it impossible for any algorithm to provide a correct answer for single-source shortest paths?

(2)

Negative cycle A → B → D → A would allow any path to be made shorter by going around the cycle one more time

X) 10 points : Minimum Spanning Tree

Consider the following undirected, weighted graph:



a) Step through Prim's algorithm to calculate a minimum spanning tree, starting from vertex *A*. Show your steps in the table below. Cross out old values and write in new ones, from left to right in each cell, as the algorithm proceeds. Also list the vertices in the order which Prim's algorithm marks them known:

Order vertices marked as known: A D B C G H E F

Vertex	Known	Distance	Path
A	x	0	–
B	x	5 1	A D
C	x	2	A
D	x	1	A
E	x	2	D
F	x	3 2	B E
G	x	4 1	A C
H	x	4 1	D G

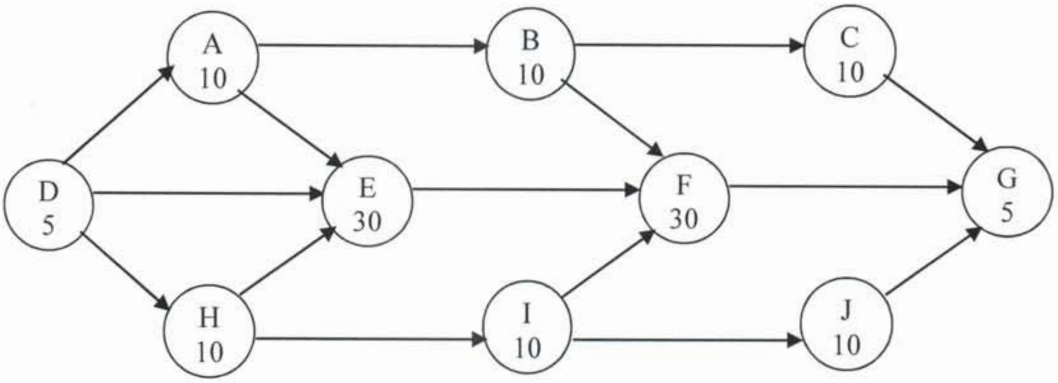
b) What are the edges in the minimum spanning tree, as computed above?

(B, D) (C, A) (D, A) (E, D) (F, E) (G, C) (H, G)

X) 10 Points : Work and Span

Consider the following directed acyclic graph representing the dependencies in a parallel computation implemented using a fork / join technique.

Each vertex is annotated with the cost of performing its work.



a) What is the work of this computation (i.e., a number)?

1 130

b) More generally, what is the work of a computation represented in this manner (i.e., described in terms of the graph and the cost of each vertex)?

1.5 sum of the work in all vertices

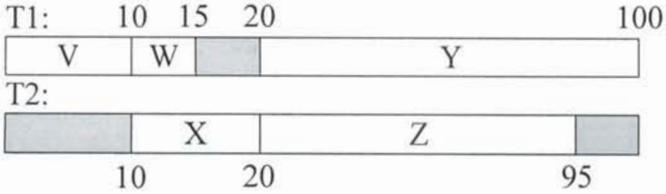
c) What is the span of this computation (i.e., a number)?

1 80

d) More generally, what is the span of a computation represented in this manner (i.e., described in terms of the graph and the cost of each vertex)?

1.5 sum of the work in the most expensive path in the graph

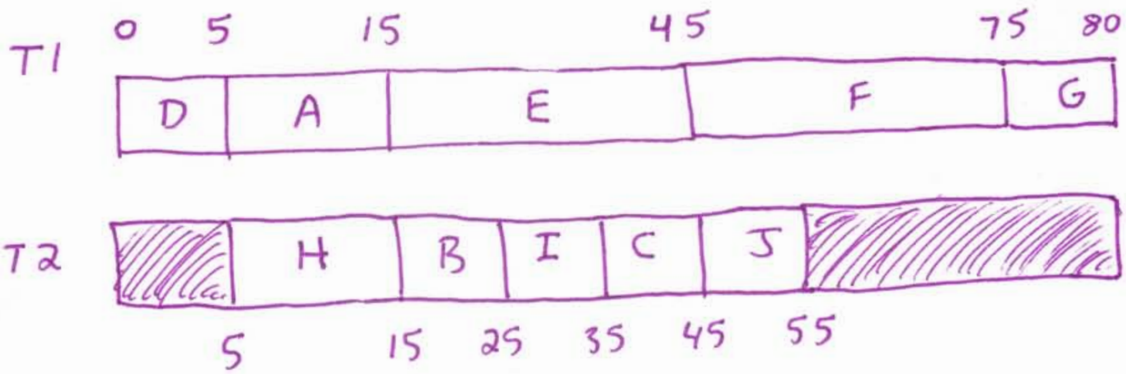
e) Assume **two threads** are executing a computation. We can illustrate the parallel work of multiple threads by drawing timelines of the work they execute. For example:



This pair of timelines illustrates a hypothetical computation in which:
T1: V for 10 units, W for 5 units, idle for 5 units, Y for 80 units
T2: idle for 10 units, X for 10 units, Z for 75 units, idle for 5 units

Draw a timeline using **two threads** to execute the above graph as quickly as possible.

Be sure your timeline illustrates the start and stop time of each task on each thread.



3

f) Would additional threads be able to perform the computation more quickly? Why?

2

No. This timeline completes in 80, which is the span.

7) 10 points : MoveToFrontList Concurrency

Consider this pseudocode for a MoveToFrontList, which is correct in a sequential context. It does not map keys to data items, but instead just tests whether a key is in the list.

```
01: class Node {
02:   Key  key;
03:   Node next;
04:
05:   Node(...) { // Constructor that stores these 2 fields }
06: }
07:
08: class MoveToFrontList {
09:   Node front = null;
10:
11:   void insert(Key key) {
12:     front = new Node(key, front);
13:   }
14:
15:   Boolean contains(Key find) {
16:     if(front == null) { return false; }
17:     if(front.key == find) { return true; }
18:
19:     Node prev = front;
20:     Node current = front.next;
21:     while(current != null) {
22:       if(current.key == find) {
23:         prev.next = current.next;
24:         current.next = front;
25:         front = current;
26:         return true;
27:       }
28:       current = current.next;
29:     }
30:     return false;
31:   }
32: }
```

We have numbered the lines of code so that you can reference them in your answers. Please do this, as it will be faster and will keep your answer more concise.

In describing an interleaving, you might write:
Thread 1 runs `contains(...)`, *stopping between lines 16 and 17.*

In describing a modification of the code, you might write:
Insert additional code after line 9:

```
09a:   Node middle = null;
09b:   Node back = null;
```

Replace line 28:

```
30:     return true;
```

Now consider using our MoveToFrontList in a multi-threaded context:

a) Describe an interleaving of `insert("a")` and `contains("b")` that results in the `insert("a")` being "missed" (i.e., `contains("a")` will return `false`).

T1 runs contains('b'), Finds 'b' in list,
pauses between lines 24 and 25

T2 runs entire insert('a')

T1 resumes and completes

Line 25 in T1 overwrites front with the
node containing the 'old' front
as its 'next' value, the 'new' front is lost

b) Describe an interleaving of `insert("a")` and `insert("b")` that results in the `insert("a")` being "missed" (i.e., `contains("a")` will return `false`).

T1 runs insert('b'),
pauses on line 12 after Node constructor
completes but before assignment to front

T2 runs entire insert('a')

T1 resumes and completes

Line 12 assignment in T1 overwrites front
with node constructed using 'old' front,
the 'new' front is lost

c) Describe an interleaving of `contains("a")` and `contains("a")` that results in them returning different values (i.e., one returns `true` and one returns `false`).

T1 runs contains('a'), finds 'a', or 24, 25
pauses between lines 23 and 24
T2 runs contains('a'), does not find 'a'
T1 resumes and completes

T1 takes 'a' out of the list in order to move it to the front, T2 sees this bad state not containing 'a'

d) Using any of the mutual exclusion mechanisms discussed in lecture (including those unique to Java), describe how to fix this class so that it is correct in concurrent usage. Your only concern is correctness (i.e., performance is not a concern).

synchronize boths insert and contains

11 : synchronized void insert (Key key) {

15 : synchronized Boolean contains (Key key) {

8) 10 points : Heap Concurrency

You and a partner are implementing an array-based binary heap. Your partner wants to use fine-grained locking to simultaneously allow multiple concurrent operations in the heap. They propose the following strategy for implementing the locking:

1) Guard each location in the array with a lock (i.e., guard each node in the heap with a lock). Always obtain the lock before reading from or writing to the array location (i.e., the node).
2) Implement `percolateUp` and `percolateDown` such that they lock nodes in the course of the percolation. Before comparing the keys of two nodes, for example, they will lock both nodes. To ensure nothing else is reading or manipulating the portion of the heap affected by percolation, they will hold locks they obtain until the percolation completes.

Your partner claims this is a good strategy and that you can work out the details in the course of the implementation. But you already see two major problems.

a) As described, this approach includes a data race. A critical variable for implementing the heap's `insert` and `deleteMin` operations is unguarded. What is that critical variable? Give an example of a bad behavior might result from it being unprotected.

The size of the heap is unguarded

Insert or Deletemin may fail due

b) Why will it be extremely difficult to guard the variable from (a) while also preserving your partner's desire to allow multiple concurrent operations in the heap?

Guarding size will be coarse-grained

c) In additional to potential race conditions, the proposed approach has another serious concurrency problem. What is the name for that problem? How could the problem occur with this strategy for implementing the locking?

Deadlock

a) Provide pseudocode for the class `InternTask`. We provide its member declarations and its constructor. You just need to implement the `compute()` method. Do not use a sequential cutoff: the base case should process a single `Customer`. Your implementation should perform the computation in *O*(n) work and *O*(log n) span.

```
class InternResult {
  Customer[] customers;
  Boolean[]  migrate;
  Letter[]   letters;
  int        numMigrate;

  InternResult(...) { // Constructor that stores these 4 fields }
}

class InternTask extends RecursiveTask<InternResult> {
  Customer[] customers;
  Boolean[]  migrate;
  Letter[]   letters;
  int        low;
  int        high;

  InternTask(...) { // Constructor that stores these 5 fields }

  InternResult compute() {
```

```
        // Base case
  +.5   if ( low == high) {
  +1        migrate [low] = InternProject. isProfitableToMigrate (customers[low])
            if (migrate [low]) {
                letters [low] = InternProject. generateLetter (customers [low])
  +1        }
     +.5    return new InternResult (
                customers, migrate, letters,
     +1         migrate [low] ? 1 : 0
            );
        }
```

```
        // Recursive Case
        InternTask leftTask = new InternTask(
+.75        customers, migrate, letters, low, (low +high) /2
        );
        InternTask rightTask = new InternTask(
+.75        customers, migrate, letters, (low + high) /2, high
        );

+1      leftTask.fork();

+1      InternResult rightResult = rightTask.compute();
+1      InternResult leftResult = leftTask.join();

+.5     return new InternResult(
                customers, migrate, letters,
                leftResult.numMigrate + rightResult.numMigrate
+1
          );




    }
}
```