Princeton University – Computer Science
COS226: Data Structures and Algorithms

Final Exam, Fall 2013

This test has 12 questions worth a total of 91 points. The exam is closed book, except that you are allowed to use a one page written cheat sheet. No calculators or other electronic devices are permitted. Give your answers and show your work in the space provided. Write and sign the Honor Code pledge before turning in the test.

"I pledge my honor that I have not violated the Honor Code during this examination."

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

Total	/92

Name:

Login ID:

P01	Guna	F 9	P03A	Debbie	F 11
P02	Guna	F 10	P04	Debbie	F 1230
P02A	Tengyu	F 10	P04A	Ruth	F 1230
P03	Bob	F 11			

Jan 22: 653e 10f3 8824

Tips:

- There may be partial credit for incomplete answers. Write as much of the solution as you can, but bear in mind that we will deduct points if your answers are 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.
- On all design problems, you may assume the uniform hashing assumption unless otherwise stated.
- Not all information provided in a problem may be useful.

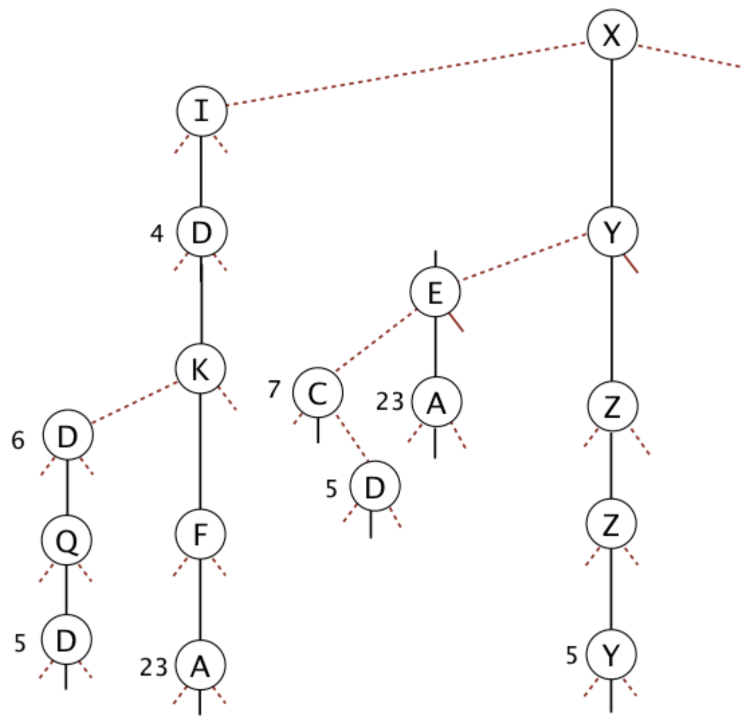
Optional. Mark along the line to show your feelings on the spectrum between ☹ and ☺.

Before exam: [☹__________________☺].
After exam: [☹__________________☺].

0. So it begins. (1 point). Write your name and Princeton NetID on the front page. Circle your precept. Enjoy your free point.

1. **TSTs. (6 points)**

Consider the TST below. Values associated with a node (if any) are shown to the left of the node.

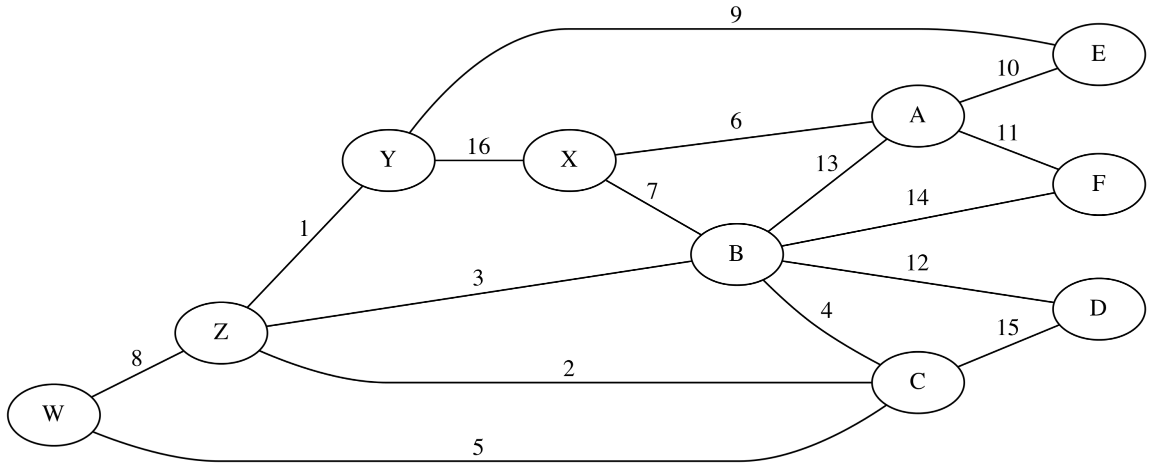


(a) List the 8 strings in the TST in alphabetical order.

(b) Give an example of a minimum length string that will increase the height of the TST. Recall that the height is the maximum number of links that must be traversed before reaching a leaf node.

2. **MSTs (6 points)**

Consider the following edge-weighted graph with 10 vertices and 16 edges. The edge weights are distinct integers between 1 and 16.



(a) Complete the sequence of edges in the MST in the order that Kruskal's algorithm includes them (by specifying their edge weights).

 1
---- ---- ---- ---- ---- ---- ---- ---- ----

(b) Suppose we add a single constant k to every edge weight (i.e. all edges are increased by the same amount). At most, how many of the nine edges in the MST can change?

(c) Consider two vertices in an undirected graph. Is the shortest path between two vertices always part of the MST? If so, explain why. If not, provide a counter-example.

3. **String sorting (10 points)**

(a) The column on the left is the original input of strings to be sorted; the column on the right is the strings in sorted order; the other columns are the contents at some intermediate step during one of the algorithms listed below. Match up each algorithm by writing its number under the corresponding column. Numbers may be used more than once.

```
HOWD   ACEF   FARM   TOHA   ACEF   ISAF   ACEF   YAND   ACEF
YAND   ANDD   ACKS   WELC   DINO   ETAS   ACKS   TAND   ACKS
WELC   ACKS   ETAS   HOWD   FARM   ANDD   ANDD   MANS   ANDD
OMET   DINO   ENJO   YAND   HOWD   ACEF   DINO   FARM   DINO
OTHE   ENJO   ANDD   ANDD   ISAF   OMET   ENJO   SAUR   ENJO
DINO   ETAS   DINO   TAND   OMET   TOHA   ETAS   ACEF   ETAS
SAUR   FARM   ACEF   OTHE   OTHE   OTHE   FARM   ACKS   FARM
FARM   HOWD   HOWD   ISAF   SAUR   ORHU   HOWD   WELC   HOWD
THIS   ISAF   THIS   ACEF   THIS   THIS   INOS   NGOU   INOS
ISAF   INOS   ISAF   UNPL   UNPL   ENJO   ISAF   THIS   ISAF
UNPL   MANS   UNPL   FARM   WELC   ACKS   MANS   DINO   MANS
ACEF   NGOU   SAUR   YSOM   YAND   WELC   NGOU   OMET   NGOU
ORHU   OMET   ORHU   TYSN   ANDD   YAND   OMET   ANDD   OMET
MANS   OTHE   MANS   DINO   INOS   TAND   ORHU   ENJO   ORHU
ANDD   ORHU   OTHE   ENJO   MANS   DINO   OTHE   INOS   OTHE
INOS   SAUR   INOS   SAUR   NGOU   MANS   SAUR   UNPL   SAUR
TOHA   THIS   TOHA   THIS   ORHU   YSOM   THIS   TOHA   TAND
NGOU   TOHA   NGOU   MANS   TOHA   INOS   TOHA   HOWD   THIS
TAND   TAND   TAND   INOS   ACKS   NGOU   TAND   ORHU   TOHA
ENJO   TYSN   OMET   ETAS   ENJO   UNPL   TYSN   ISAF   TYSN
YSOM   UNPL   YSOM   ACKS   ETAS   FARM   UNPL   YSOM   UNPL
ETAS   WELC   WELC   OMET   TAND   TYSN   WELC   ETAS   WELC
TYSN   YAND   TYSN   ORHU   TYSN   SAUR   YAND   OTHE   YAND
ACKS   YSOM   YAND   NGOU   YSOM   HOWD   YSOM   TYSN   YSOM
----   ----   ----   ----   ----   ----   ----   ----   ----
 0                                                       1
```

0: Original input
1: Sorted
2: Mergesort
3: LSD
4: MSD
5: Quicksort (standard no shuffle)

(b) As you may recall, a Point2D is an object consisting of an x coordinate and a y coordinate, each stored as doubles. The natural order is given by sorting first on the y coordinate, then the x coordinate.

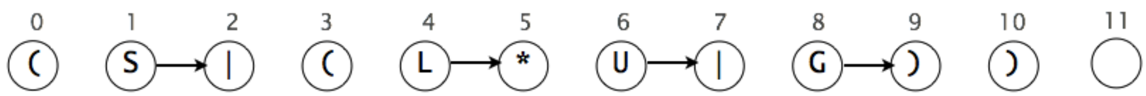
Sorting N Point2D objects using a divide and conquer based sort (e.g. mergesort) requires $O(N \log N)$ time. In this problem, we'll consider using LSD radix sort to avoid the logarithmic factor.

One approach would be to use 128 bit digits, effectively treating the entire Point2D object as one big digit. In this case $R = 2^{128}$, and $W = 1$. If we use LSD sort, sorting takes only $O(N)$ time. Concisely explain why this technique is not practically possible.

(c) At the opposite extreme, we could treat each Point2D as a sequence of 128 binary digits. In this case, $R = 2$ (the binary alphabet), and $W = 128$. As before, we can sort our Point2Ds in $O(N)$ time. Give one reason why this might not be a good idea.

4. **Regular Expressions (5 points)**

(a) Suppose we use the RE-to-NFA construction technique **from the book** on the regular expression `(S|(L*U|G))`. The match transition are shown below:

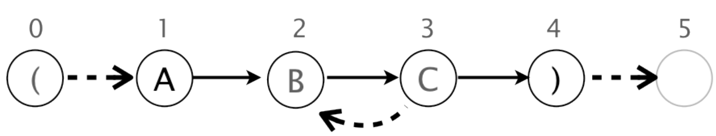


Circle all edges in the list below that appear in the ϵ-transition digraph. Not all ϵ-transitions appear in the list below (e.g. $10 \to 11$ is a correct ϵ-transition but it is not in the list below). One answer is already provided for you.

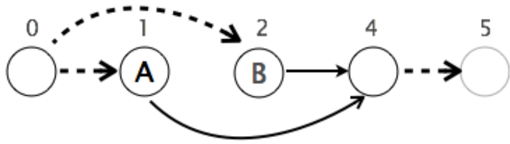
$0 \to 1$	$0 \to 2$	$0 \to 3$	$0 \to 4$	$0 \to 9$	$0 \to 10$
$2 \to 3$	$2 \to 4$	$2 \to 5$	$2 \to 6$	$2 \to 9$	$2 \to 10$
$3 \to 4$	$3 \to 6$	$3 \to 7$	$3 \to 8$	$3 \to 9$	$3 \to 10$
$4 \to 3$	$4 \to 4$	$4 \to 5$	$4 \to 6$		
$5 \to 4$	$5 \to 6$	$5 \to 7$	$5 \to 9$		
$7 \to 3$	$7 \to 5$	$7 \to 8$	$7 \to 9$	$7 \to 10$	

(b) Give a valid regular expression corresponding to the NFAs below. These NFAs were NOT generated using the procedure described in class. **States without a displayed character (i.e. empty circles) do not necessarily correspond to a regular expression meta-character.** Write your answer in the blank provided. You are permitted to use any standard regular expression meta-character. It is also ok if you stick to the basic metacharacters: `( ) * |`

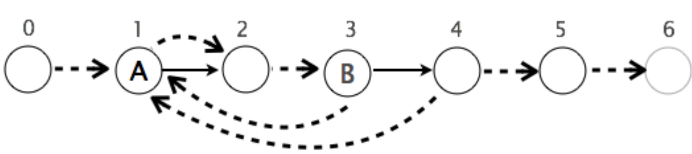
Epsilon transitions are shown with dotted lines. First answer is provided for you.



ABB*C



____



____

5. **Substring Search (8 points)**

(a) Construct the KMP DFA that matches the string BAABBAABB.

	0	1	2	3	4	5	6	7	8
A									
B	1								
s	B	A	A	B	B	A	A	B	B

(b) Suppose that you run Boyer-Moore as discussed in class on the text GULLSSSSTLOSTSLUGSUG to search for the pattern SLUGS. Give the trace of the algorithm in the grid below, **circling the characters in the pattern that get compared with the text**.

G	U	L	L	S	S	S	S	T	L	O	S	T	S	L	U	G	S	U	G
S	L	U	G	S															

(c) Suppose we try to modify our substring matching algorithms so that any permutation of the characters in a pattern will be accepted as a match. For example, suppose our pattern is "SLUGS". In this case, "SUGLS" and "GSLUS" will be considered a match, but "SSSSS" will not.

Concisely explain why the KMP-DFA technique is poorly suited to the job of finding permutations of a pattern.

(d) **Bonus problem (2 pts):** Suppose we modify Boyer-Moore to match any permutation. The new string comparison method will work as follows: Initialize an array of length 26, with one entry corresponding to each letter in English. For example, for “SLUGS”, we store the number 1 in position G, 1 in position U, 2 in position S, and 1 in position L, and 0 in every other position. When scanning the text, decrement the appropriate letter in the array. If any entry becomes negative we have a mismatch. For example, for the first substring compare in the table at the bottom of this page, S, S, U, and finally A would be decremented. The compare method would notice that A was negative, indicating a mismatch.

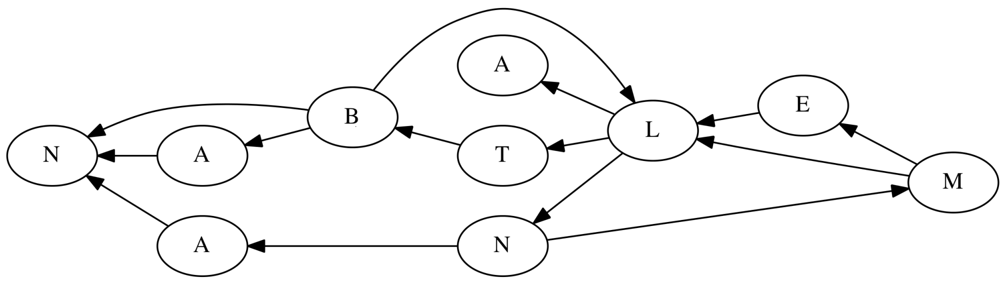
On a mismatch, the pattern is then moved X positions to the right, and for every **examined** text character skipped, the appropriate position in the array is incremented by 1. For example, assume that X=3 positions, then we’re throwing away G, A, and U. Since A and U were decremented, we have to undo these decrement operations, and add one back to both A and U. S would remain at 0 (since neither S was skipped).When the second search began (assuming X=3), our counters would be G=1, U=1, L=1, S=0, everything else 0. X=3 may not be the optimal answer and is provided only as an example.

Give an optimal rule for deciding how much to skip. If you have multiple cases, describe each. You are not required to use the chart on the bottom of this page to design your rule, but it might be helpful. Be as concise as possible.

G	A	U	S	S	L	I	S	T	L	O	S	T	S	L	U	G	S	U	G
S	L	U	G	S	For this first compare, S decreases by 2, U and A dec. by 1														

6. **Directed Graphs (9 points).**

(a) Give the reverse postorder of the graph below when using DFS. Start from vertex M. Assume the adjacency lists are in sorted lexicographic order. For example, follow the edge $L \rightarrow A$ before following $L \rightarrow N$.



____ ____ ____ ____ ____ ____ ____ ____ ____ ____

(b) Consider the following algorithm for finding the shortest ancestral path (from the WordNet assignment) of two vertices A and B: Use BFS, but maintain two queues of vertices instead of one. One queue starts with only vertex A, and one queue starts with only vertex B. Each iteration of the algorithm, dequeue a vertex v from the A queue, and mark v by A if it has not already been marked by A, and finally enqueue all of v's neighbors into the A queue. Then repeat the same thing but with the B queue. Repeat this process of alternating between the A queue and B queue, stopping as soon as a vertex is found that is marked with both an A and a B, and return that vertex as the shortest ancestor.

Is this algorithm correct? If so, provide an intuitive explanation for why (you do not need to provide a full proof). If not, give a small counter-example.

(c) Consider the DFS-based code below, intended to find the longest path from some starting vertex in an edge weighted digraph assuming edge weights are integers.

```
private void dfsLongestPath(EdgeWeightedDigraph G, int v, int distToV) {
    marked[v] = true;
    distTo[v] = distToV;
    edgeTo[v] = v;
    for (DirectedEdge e : G.adj(v)) {
        int to = e.to();
        if (!marked[to]) dfsLongestPath(G, to, distTo[v] + e.weight());
    }
}
```

Suppose we search from some vertex s using the call **`dfsLongestPath(G, s, 0).`** For each of the situations below, write "Yes" if this call is guaranteed to find the longest path to every vertex for any graph obeying the stated property, and write "No" if it is not guaranteed to do so.

----- The graph is a tree.

----- The graph contains no cycles.

----- There are no cycles containing the target vertex.

----- All edge weights are positive.

----- All graphs.

This area of this page is a designated FUN ZONE.

7. **Reductions (6 points)**

For each of the two reductions below, circle the appropriate order of growth and state whether or not the algorithm is correct.

(a) To find the longest path in a weighted directed acyclic graph with V vertices and E edges, negate every edge and use the DAG shortest paths algorithm. *You may not make any assumptions about the weights on the original graph*.

i. Is this algorithm correct for all inputs (circle one):

Yes No

ii. What is the worst case runtime to complete this algorithm, including the time to perform the reduction as a function of V and E (circle one):

Constant Logarithmic Linear Linearithmic Quadratic Polynomial[1] Exponential

(b) Given a sliding puzzle on an NxN grid (e.g. an 8 puzzle is where N=3), build a graph where each vertex represents exactly one state of the puzzle. Add an edge between any two reachable states. After constructing the graph, run BFS starting from the initial state. A shortest solution to the sliding-puzzle is found as soon as the goal state of the puzzle is reached.

i. Is this algorithm correct for all inputs (circle one):

Yes No

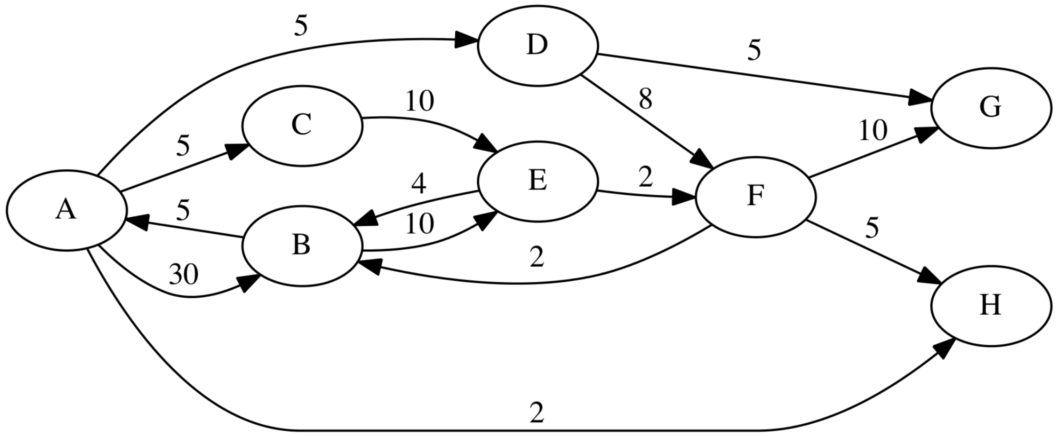
ii. What is the worst case runtime to complete this algorithm, including the time to perform the reduction as a function of the grid size (circle one):

Constant Logarithmic Linear Linearithmic Quadratic Polynomial[1] Exponential

[1] Here polynomial means the worst case runtime is $\Theta(N^k)$ for some k > 2 (worse than quadratic).

8. **Shortest paths (6 points)**

Consider the directed weighted graph below.



(a) Complete the table of edgeTo[] and distTo[] values immediately after the first five vertices (A, H, C, D, G) have been relaxed during the execution of Dijkstra's algorithm. Some values have already been provided for you.

v	edgeTo[]	distTo[]
A	null	0.0
B		
C	A	5.0
D	A	5.0
E		
F		
G	D	10.0
H	A	2.0

(b) What vertex will be relaxed next by Dijkstra's algorithm?

(c) Fill in the table of edgeTo[] and distTo[] values after the 6th vertex you listed in part b is relaxed – you are only required to list any values that have changed since part a (i.e. leave rows blank which did not change).

v	edgeTo[]	distTo[]
A		
B		
C		
D		
E		
F		
G		
H		

(d) A modified version of Dijkstra's algorithm with two additional lines of code is shown below (annotated in bold). Given a graph G for which Dijkstra's algorithm returns a correct result, will this version of Dijkstra's algorithm always return the correct result G? Give an intuitive reason for your answer (you do not need to provide a full proof).

```
public DijkstraSP(EdgeWeightedDigraph G, int s) {

    distTo = new double[G.V()];
    edgeTo = new DirectedEdge[G.V()];
    for (int v = 0; v < G.V(); v++)
        distTo[v] = Double.POSITIVE_INFINITY;
    distTo[s] = 0.0;

    // relax vertices in order of distance from s
    pq = new IndexMinPQ<Double>(G.V());
    pq.insert(s, distTo[s]);
    while (!pq.isEmpty()) {
        int v = pq.delMin();
        for (DirectedEdge e : G.adj(v))
            relax(e);
        for (DirectedEdge e : G.edges()) // G.edges() returns an Iterable of
            relax(e);                    // every DirectedEdge in G.
    }
}
```

9. LZW Compression **(5 points)**

For each of the compressed bitstreams below, remark on whether that bitstream could possibly have been generated by the LZW algorithm as discussed in class. If the bitstream is possible, provide the original input stream recovered after decompression. If it is not possible, use the blank to explain why.

Assume that we're using the same parameters as are used in the booksite code, i.e. 16 bit codewords (between `0000` and `FFFF`), with the first 256 codewords reserved for our ASCII characters (between `00` and `FF`). `0100` is used for the end of file character, and `0101` will be the first new codeword added to the codebook. The first one has been completed for you. Codebooks reset after each problem.

`0048 0045 004C 004C 004F 0100`	HELLO
`004C 004F 0101 0100`	
`005A 005A 005A 0100`	
`004C 004F 0101 0103 0100`	
`005A 0101 0102 0100`	
`005A 0102 0100`	

For reference, below is the hexademical-to-ASCII conversion table from the textbook:

	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

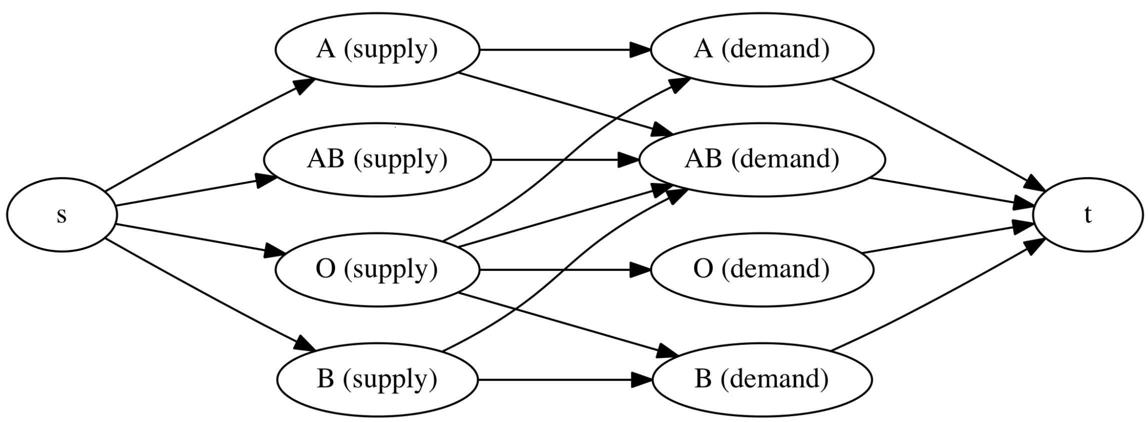
10. **MaxFlow and Reductions (11 points).**

Enthusiastic celebration of a sunny day at a prominent northeastern university has resulted in the arrival at the university's medical clinic of 169 students in need of emergency treatment. Each of the 169 students requires a transfusion of one unit of whole blood. The clinic has supplies of 170 units of whole blood. The number of units of blood available in each of the four major blood groups and the distribution of patients among the groups is summarized below.

Blood type	A	B	O	AB	SUM
Supply	46	34	45	45	170
Demand	39	38	42	50	169

Type A patients can only receive type A or O; type B patients can receive only type B or O; type O patients can receive only type O; and type AB patients can receive any of the four types.

Your job in this problem is to find a maxflow formulation that determines a distribution that satisfies the demands of a maximum number of patients.



(a) Provide edge capacities that will help solve the blood distribution problem to each edge in the problem above. If you use any infinite weight edges, it is OK to leave them unlabeled. Draw the edge capacities on the graph above.

(b) What is the value of the max flow for the graph?

(c) Which vertices are on the **t side** (target side!) of the min-cut? Note: These vertices should provide a concise description of why there is not enough blood for everyone (sry); you need not write such an explanation, just reminding you about this neat min-cut fact.

(d) Imagine a generalized version of the blood demand problem where we have N distinct blood types, up to N^2 possible supply/demand connections, and wish to determine whether or not there is sufficient blood for everybody. Let's call this problem BloodDemand. Suppose we are given the following information:

i. BloodDemand reduces to a MaxFlow problem with a number of edges E that grows with order N^2 using the technique you used in the problem above.
ii. Sleator and Tarjan have discovered an algorithm that can solve MaxFlow in time $E^2 \log E$, where E is the number of edges.
iii. Guna has found that $E \log\log\log E$ is a lower bound for solving MaxFlow, or equivalently that MaxFlow is $\Omega(E \log\log\log E)$.

Which of the following can you definitively infer from these three facts? Write "Yes" for those that you can infer, and "No" for those you cannot definitively infer.

----- BloodDemand linear time reduces to MaxFlow.

----- BloodDemand quadratic time reduces to MaxFlow.

----- MaxFlow provides an $N^4 \log N^2$ solution to BloodDemand.

----- There exists an $E \log\log\log E$ solution for MaxFlow.

----- An $N^2 \log\log\log N^2$ solution to BloodDemand would provide an order $E \log\log\log E$ solution to MaxFlow.

----- $N^2 \log\log\log N^2$ is a lower bound for BloodDemand.

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

The following questions are intended to assess your understanding of the ideas in the course by asking things in a slightly oblique way. They are not trick questions.

(a) Heaps and binary search trees are both types of binary trees. We stored heaps as an array to save space. Why don't we do the same for binary search trees?

(b) In our algorithm for building NFAs for regular expression matching, we push left parentheses and OR symbols onto a stack. In what two specific cases do we utilize the left parenthesis?

(c) Suppose we used a priority queue of vertices instead of a queue of vertices for BFS. Suppose we gave each vertex a priority equal to its distTo[] from the source. Would BFS still work? Would runtime performance be better or worse?

(d) In the Ford-Fulkerson algorithm, every time we find an augmenting path, the max flow is increased. If all edges have integer capacity, what is the minimum increase in flow that results from this augmenting path? How does this prove that Ford-Fulkerson always completes on graphs whose edges all have integer capacity?

(e) In the lazy version of Prim's algorithm for finding MSTs on an undirected weighted graph, we add the smallest edge pointing from any vertex in the MST unless both vertices connected by that edge are already marked (in order to avoid cycles in the MST). In Dijkstra's algorithm for directed graphs, we never explicitly check to see whether the to-vertices are marked, i.e. we always relax an edge, even if it would form a cycle. Explain why such a check is not necessary for Dijkstra's algorithm.

(f) In our implementation of the Huffman coding algorithm, we use a standard trie instead of a TST in order to map from a compressed bitstream back to the original data. Why don't we use a TST?

12. **Algorithm Design (10 points)**

(a) Given an edge weighted directed graph G and a source vertex s, and assuming that all edge weights are positive, provide an algorithm for finding the **second** shortest paths from s to every other vertex. In other words, you want to find something similar to the shortest paths tree (i.e. distTo and edgeTo), except that every path found must be the second shortest. Assume that E is greater than V. Your algorithm should complete in $O\big(VElog(V)\big)$ time. You may use any algorithm from class as a subroutine without writing out the details of that algorithm.

(b) Given an unweighted directed graph and a starting vertex u, give an algorithm for finding all vertices such that there is an odd-length path to those vertices. These paths may involve cycles. For full credit your algorithm should complete in E+V time. For partial credit, give an algorithm that completes in EV time.