ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Final Exam, Algorithms 2014-2015

- You are only allowed to have a handwritten A4 page written on both sides.
- Communication, calculators, cell phones, computers, etc... are not allowed.
- Your explanations should be clear enough and in sufficient detail so that a fellow student can understand it. In particular, do not only give pseudocode without explanations. A good guideline is that a description of an algorithm should be so that a fellow student can easily implement the algorithm following the description.
- **Do not touch until the start of the exam.**

Good luck!

Name: ______________________ **N° Sciper:** ______________

Problem 1 / 15 points	Problem 2 / 15 points	Problem 3 / 5 points	Problem 4 / 11 points	Problem 5 / 29 points	Problem 6 / 25 points

Total / 100

1 *(15 pts)* **Asymptotics and basic runtime analysis.**

1a *(5 pts)* Arrange the following functions in increasing order according to asymptotic growth.

$$2^n, (\log\log n)^{10}, n^{300}, \sqrt{n}, n/\log n, 2^{2^n}, n^{\sqrt{n}}, \log n$$

(In this problem, you only need to give the right order, i.e., you do not need to explain your answer.)

Solution:

$$(\log\log n)^{10} < \log n < \sqrt{n} < n/\log n < n^{300} < n^{\sqrt{n}} < 2^n < 2^{2^n}$$

1b *(10 pts)* Consider the following four recursive functions. We assume $f_i(0) = 0$ and they are defined as follows for $n \geq 1$:

$$f_1(n) = \max_{1 \leq i \leq n} (p_i + f_1(n-i)), \qquad f_2(n) = \max_{1 \leq i \leq \lfloor \log n \rfloor} (p_i + f_2(n-i))$$

$$f_3(n) = p_n + f_3(n-1), \qquad f_4(n) = \max_{\substack{1 \leq i \leq n \\ 1 \leq k \leq i}} (k^2 p_i + f_4(n-k)),$$

where $p_1, \ldots, p_n$ are positive integers.

Give tight asymptotic running times (using the $\Theta(\cdot)$ notation) of the bottom-up dynamic programming implementations for calculating $f_1(n), f_2(n), f_3(n)$, and $f_4(n)$. We assume that the implementation of the bottom-up does not use any special properties of the recursions, i.e., it fills in each cell in the table using exactly the definition of the recursion.

(In this problem, you only need to give the answer, i.e., no explanations are needed. In particular, you do not need to explain the bottom-up dynamic programs.)

Solution: Using the bottom-up dynamic programming technique (without using any special properties of the recursions)

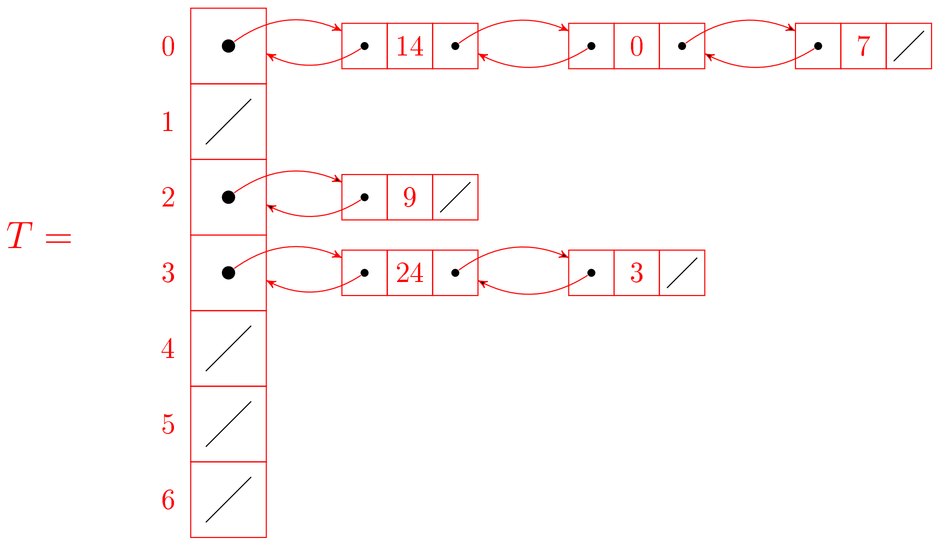
- the asymptotic running time for calculating $f_1(n)$ is $\Theta(n^2)$.
- the asymptotic running time for calculating $f_2(n)$ is $\Theta(n \log n)$.
- the asymptotic running time for calculating $f_3(n)$ is $\Theta(n)$.
- the asymptotic running time for calculating $f_4(n)$ is $\Theta(n^3)$.

2 *(15 pts)* **Hash tables.**

2a *(7 pts)* **Illustrate/draw the hash table** $T[0,\ldots,6]$ obtained after inserting the keys $3, 9, 7, 24, 0, 14$ in the given order using the hash function $h(k) = k \mod 7$. Collisions are resolved using chaining with double-linked lists.

(Note that you only need to draw *one* hash table: the one obtained after inserting *all* the above keys.)

Solution:



2b *(8 pts)* Suppose you have a hash table with $2n$ slots and suppose that n distinct keys are inserted into the table. Each key is equally likely to be hashed into each slot (*simple uniform hashing*).

Prove that the expected number of collisions is $(n-1)/4$. Recall that we say that two keys k_i and k_j with $i \neq j$ collide if they are hashed to the same slot. The number of collisions is the number of pairs of keys that collide.

Solution:

For $1 \leq i < j \leq n$, we let X_{ij} be the indicator variable that indicates whether key k_i is hashed to the same slot as k_j. That is $X_{ij} = I\{h(k_i) = h(k_j)\}$ where h is the hash-function. Let X be the total number of collisions. By the above definitions and the definition of collisions, we have $X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}$. Using linearity of expectation, we thus have that the expected number of collisions is

$$\mathbb{E}[X] = \mathbb{E}\left[\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}\right] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \mathbb{E}[X_{ij}].$$

Since we assume simple uniform hashing we have that the probability that two different keys hash to the same slot is $1/(\text{the number of slots}) = 1/(2n)$. Substituting in this equality, we have that the expected number of collisions is

$$\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{1}{2n} = \sum_{i=1}^{n-1} \frac{n-i}{2n} = \frac{1}{2n} + \frac{2}{2n} + \ldots + \frac{n-1}{2n} = \frac{n(n-1)}{4n} = \frac{(n-1)}{4},$$

as required.

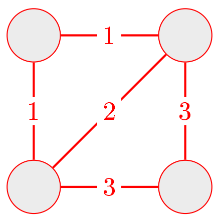
3 *(5 pts)* **Spanning trees.** Let $G = (V, E)$ be a connected undirected graph with edge weights $w : E \to \mathbb{R}$. Consider an arbitrary edge $e \in E$. Professor Homer Simpson thinks the following statement is always true:

"e is contained in a minimum spanning tree **or** e is contained in a maximum spanning tree."

Show that Homer is wrong by **giving a counterexample** to this statement.

Solution:

A counter example is as follows:



The above is a counterexample to the statement of Homer because the edge of weight 2 is in no minimum spanning tree and in no maximum spanning tree. Indeed a minimum spanning tree is formed by taking both edges of weight 1 and then one edge of weight 3; and a maximum spanning tree is formed by taking both edges of weight 3 and then one edge of weight 1.

Another way to see that Homer is wrong is as follows: Take a complete graph with $n \geq 5$ vertices where all edges have distinct weights. Since all edge-weights are unique, there exists a unique minimum spanning tree and a unique maximum spanning tree. Hence at most $2(n-1)$ edges can be in any of them but a complete graph has $n(n-1)/2$ edges.

4 *(11 pts)* **Basic algorithm design.** It is well-known that finding the minimum in an array of n numbers requires time $\Theta(n)$. However, if we are given more information about the structure of this array, we might be able to do it more efficiently. For instance, if the array is sorted, then one can find the minimum in constant time.

In this problem, we are given an array $A = [a_1, \ldots, a_n]$ of n numbers, with the promise that there exists an index $\ell \in \{1, 2, \ldots, n\}$ such that

$$a_1 > a_2 > \cdots > a_\ell \qquad \text{and} \qquad a_\ell < a_{\ell+1} < \cdots < a_n$$

Note that in this case, the minimum in this array is a_ℓ.

Design and analyze an algorithm that, given such an array A, returns the minimum element a_ℓ in time $O(\log n)$ (no points will be given for worse running times). You may assume that the array is always in the *correct* format, and the elements in the array are distinct, and hence the minimum is unique.

Solution:

We observe that our array has a very nice structure: whenever we check an element, we know it is the one we are looking for if it is smaller than its neighbors. Otherwise, we know in which direction to look for: if the element is smaller than its right neighbor and larger than its left one, we should look towards the left direction, otherwise we should look towards the right direction.

This prompts us to design a divide-and-conquer algorithm, based on the binary search algorithm. The essential difference is that, while in binary search the condition we check for is if the element we are looking at is the one we are searching for, here instead the condition will be whether the element is smaller than its neighbors. Other than that, the two algorithms will be identical.

So, here is the recursive procedure we will use:

```
SEARCH($A, l, r$)
1. if $A[l] < A[l+1]$
2.     return $A[l]$
3. if $A[r] < A[r-1]$
4.     return $A[r]$
5. $m \leftarrow \lfloor (l+r)/2 \rfloor$
6. if $A[m-1] > A[m] < A[m+1]$
7.     return $A[m]$
8. if $A[m-1] < A[m] < A[m+1]$
9.     SEARCH($A, l, m-1$)
10. if $A[m-1] > A[m] > A[m+1]$
11.     SEARCH($A, m+1, r$)
```

So, this procedure implements the ideas we discussed above: it picks the middle element of the subarray we are looking at, checks if it is the minimum element of the whole array, and if not, chooses the correct direction to continue looking at, always halving the size of the subarray at each recursive call. In order to solve the problem, we run SEARCH($A, 1, n$).

To verify the correctness of the algorithm, observe that at each recursive call, the minimum element will be at the subarray we will apply the recursive call on. Hence, all we have to guarantee is two things:

- that the conditions checked in lines 6, 8 and 10 are always well-defined; this is true because they would be ill-defined only if we tried $m = 1$ or $m = n$, in which case case the subarray we are looking at has size 2, and we would have reurned a result either in line 2 or 4.

- that the size of the subarray we are looking at decreases at each recursive call; this is true, because of lines 9 and 11, which imply that the size of the subarray always decreases by at least 1.

To analyze the running time, observe that due to the selection of m and due to lines 9 and 11, the size of the subarray we will be looking at every time we apply a recursive call will be at most half the size of the original subarray. Since every recursive call takes constant time to complete, we can write the following recursive formula for the running time:

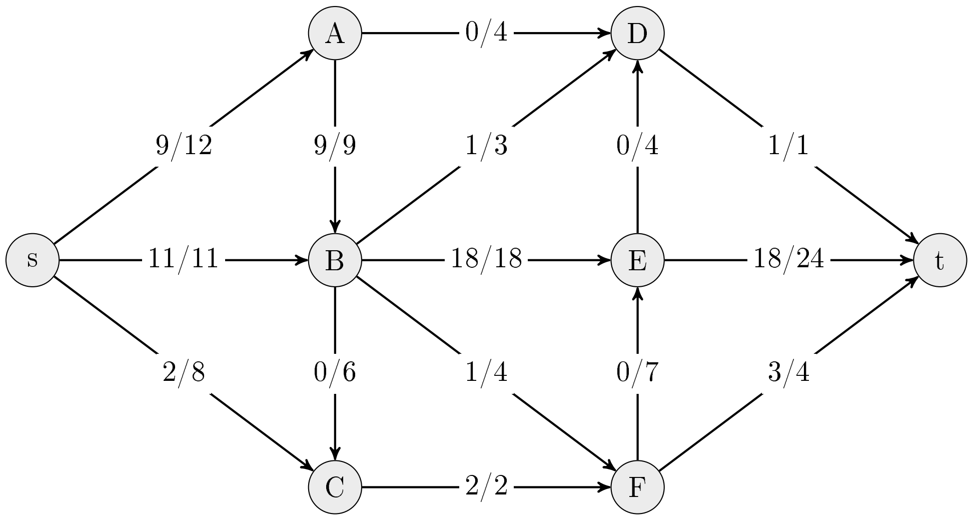
$$T(n) = T(n/2) + O(1)$$

with $T(1) = O(1)$. Applying the Master Method with $a = 1$ and $b = 2$, we get that the running time is

$$T(n) = O(\log n)$$

5 *(29 pts)* **Flows and Cuts.**

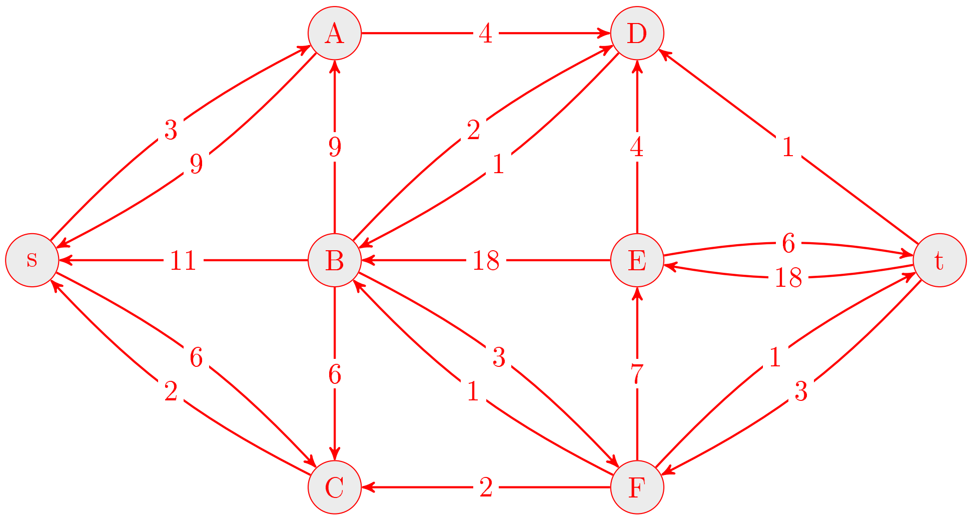
5a *(10 pts)* Consider the flow of value 22 in the following flow network (the numbers on an edge determine its current flow and its capacity).



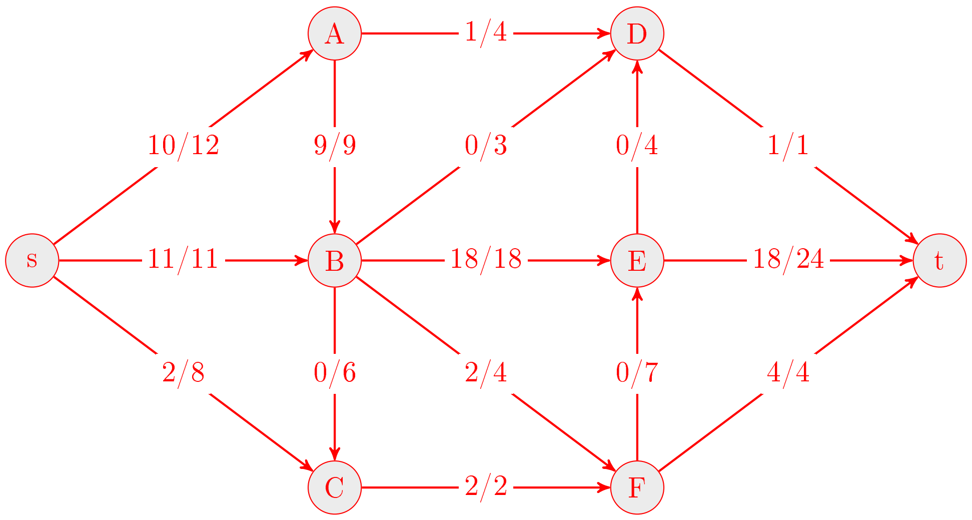
Starting with the above flow, **find a max flow by running the Ford-Fulkerson method**. In each iteration, draw the residual network, and if there exists an augmenting path, indicate which one you selected and explain how the flow is updated along this path.

Solution:

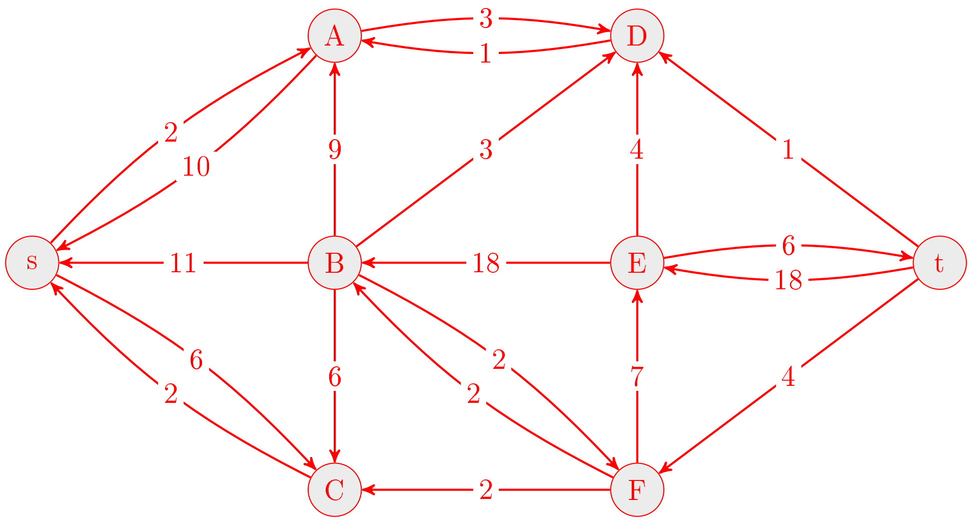
Iteration 1: The residual network is:



There exists an augmenting path: $s \to A \to D \to B \to F \to t$. The bottleneck of this path is 1 because of arc (D, B). Hence, we update the flow by sending one unit of flow along this path to obtain the new flow:



Iteration 2: The residual network is:



Note that the vertices reachable from the source are s, C, A, D and hence there is no augmenting path and the flow is a max flow of value 23. As a sanity check, the cut "corresponding" to this flow is $(\{s, C, A, D\}, \{B, E, F, t\})$ and it has indeed capacity 23.

5b *(6 pts)* Consider the same flow network as in Subproblem 5a and consider the min cut "corresponding" to the max flow that you found in that problem. Use the structure of this cut to **prove** (in a couple of sentences) that the max flow value must decrease if the capacity of arc (C, F) is decreased from 2 to 1.

Solution:

The cut corresponding to the max flow of 5a was $S = \{s, C, A, D\}, T = \{B, E, F, t\}$. As it is a minimum cut, it has capacity 23 which equals the max flow in the original flow network. Now since (C, F) is an arc of going from S to T in the cut we have that the capacity of the cut would decrease from 23 to 22 if we would decrease the capacity of (C, F) from 2 to 1. Hence, as the value of a max flow is at most the capacity of any cut, we have that the max flow value must decrease to be at most 22 if we decrease the capacity of (C, F) from 2 to 1.

5c *(13 pts)* Thanks to the many excellent students (and professors :)), EPFL has seen a rapid increase in size and quality. However, there is one worry: are there enough professors and PhD students to cover all courses? In order to ensure high quality teaching we have the following constraints in assigning teaching staff to courses. Professors and PhD students can be assigned to at most one course in their expertise. Moreover, each course needs a course-dependent number of staff members, one of which needs to be a professor.

Formally, we wish to solve the following teaching assignment problem:

Input: a set C of courses where each course $c \in C$ has a requirement $r(c) \geq 1$, a set T of teaching staff, partitioned into a set P of professors and a set S of PhD students, where each staff member $t \in T$ can be assigned to a subset $C_t \subseteq C$ of courses (corresponding to his/her expertise).

Output: If possible, an assignment of teaching staff to the courses so that

- Each staff member $t \in T$ is assigned to at most one course and, if assigned, the assignment must be to a course in C_t.
- Each course $c \in C$ is assigned at least $r(c)$ teaching staff members.
- At least one professor is assigned to each course (note that it is also fine if more than one professor is assigned to a course).

If no such assignment exists, simply output "We need to hire!".

Your task is to **design and analyze** a polynomial time algorithm for the above problem.

(Hint: formulate a flow problem so that an (integral) max flow corresponds to an assignment if one exists. However, do not forget to explain how to turn a solution to the flow problem into a solution to the original problem.)

Solution:

We shall model the problem as a flow problem with a source s and a sink t. The remaining nodes of our flow network will be arranged into two layers: the left-layer and the right-layer. The left-layer has a node L_p for each professor $p \in P$ and a node $s \in S$ for each PhD student. The right-layer has two nodes R_c and $R_{c'}$ for each course $c \in C$. (The reason that we have two nodes for each course is to ensure that each course will be assigned one professor as will be clear below.) Our construction has the following arcs:

- For each professor $p \in P$ there is an arc (s, L_p) of capacity 1. Similarly, for each PhD student $s \in S$, there is an arc (s, L_s) of capacity 1.

- For each course $c \in C$ there is an arc from $(R_{c'}, t)$ of capacity 1 and an arc (R_c, t) of capacity $r(c) - 1$ (which is non-negative since $r(c) \geq 0$).

- Finally, for each professor $p \in P$ and $c \in C_p$, there are arcs $(L_p, R_{c'})$ and (L_p, R_c) of capacity ∞; and for each PhD student $s \in S$ and $c \in C_s$, there are arcs (L_s, R_c) of capacity ∞.

This finishes the construction of the flow network, which is easy to do in linear time. See also Figure 1 for an illustration. We relate the max-flow of the flow network to a solution to the problem:

Claim 0.1 *The flow network has max flow of value $\sum_{c\in C} r(c)$ iff there is an assignment of teaching staff satisfying the required constraints. Moreover, given an (integral) max flow, we can in linear time transform it into an assignment.*

Proof. We first prove that the value of the max flow is $\sum_{c\in C} r(c)$ iff there is an assignment of teaching staff satisfying the required constraints. Suppose first that the there is an assignment satisfying the required constraints. Note that we may assume that each course is assigned exactly $r(c)$ staff members; otherwise, remove one of the assigned students (or professors if there are several professors). Then construct a flow as follows. For each professor $p \in P$ that is assigned to a course $c \in C_p$, send a unit flow along the path $s \to L_p \to R_{c'} \to t$ if no other professor has already sent flow through $R_{c'}$ otherwise send a unit of flow through the path $s \to L_p \to R_c \to t$. Similarly, for each PhD student $s \in S$ that is assigned a course $c \in C_s$ send a unit flow along the path $s \to L_s \to R_c \to t$. Clearly, flow conservation constraints are satisfied because we only send flow along paths from the source to the sink. Capacity constraints are satisfied since a professor/student is assigned at most one course. Finally, we have that exactly one professor sends flow through $R_{c'}$ for each $c \in C$ and that $r(c) - 1$ other teaching staff members send unit flows through R_c. Moreover, the flow has value equal to the total flow on incoming arcs to the sink which equals $\sum_{c\in C} r(c)$ as required.

We now prove that if there a flow of value $\sum_{c\in C} r(c)$ then there is a feasible assignment. First observe that we may assume that the flow is integral since all the capacities are integral and therefore the Ford-Fulkerson method will find an integral flow. We claim that the following is a feasible assignment: assign each professor $p \in P$ to course $c \in C_p$ if there is a unit flow from L_p to R_c or $R_{c'}$; assign each PhD student $s \in S$ to course $c \in C_s$ if there is a unit flow from L_s to R_c. Note that since the capacity of the arcs exiting the source is 1, we have that each professor/student is assigned to at most one course. Moreover, recall that the capacity of an arc (R_c, t) is $r(c) - 1$ and $(R_{c'}, t)$ is 1. Therefore, the incoming arcs have total capacity $\sum_{c\in C} r(c)$ which implies that a flow of that value has to saturate all those arcs. In other words, since only professors can send a unit flow to $R_{c'}$ we have that each course c is assigned one professor and $r(c)$ teaching staff members in total. This shows that the assignment satisfies the requirements. Furthermore, the transformation above can easily be done in linear time. □

To summarize, our argument works as follows:

1. Construct the flow network above
2. Solve it using the Ford-Fulkerson method.
3. Check if the max-flow has value $\sum_{c\in C} r(c)$. If not, output "We need to hire!".
4. Otherwise, output the assignment that we obtain from the flow as described in the claim.

As the construction of the flow network and the transformation of the flow solution into an assignment both can be implemented in linear time, the running time is dominated by the Ford-Fulkerson method. We have that the Ford-Fulkerson method can run for at most $|T|$ iterations because each iteration increases the flow by at least one unit and each teaching staff member $t \in T$ can send one unit of flow (and hence max-flow has value $\leq |T|$). Moreover, we can implement each iteration in linear time of the size of the flow network (using BFS) which has size at most $O(|T||C|)$ (if all teaching staff members can be assigned to all courses). Hence, the total running time required is at most $O(|T|^2|C|)$.

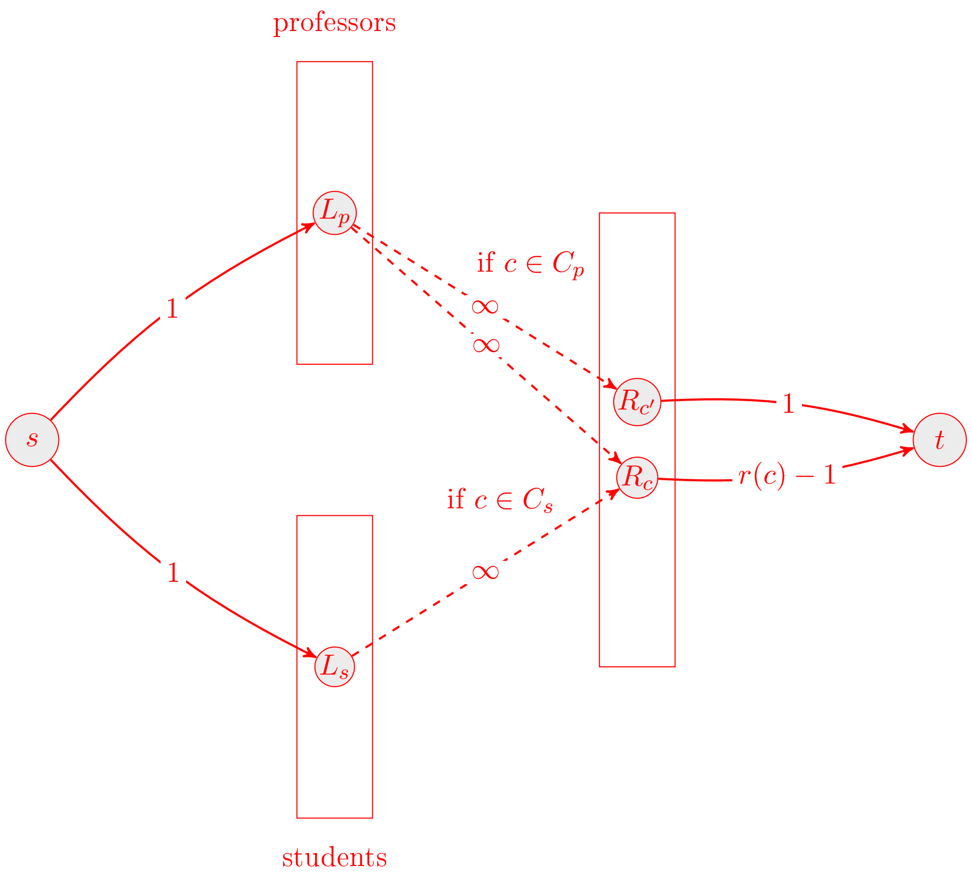


Figure 1. Illustration of the flow network.

6 *(25 pts)* **Shortest paths and dynamic programming.** Consider a directed graph $G = (V, E)$ with *positive* edge weights $w : E \to \mathbb{R}_{>0}$, i.e., $w(e) > 0$ for $e \in E$, and a vertex $s \in V$. We shall design and analyze an efficient algorithm that finds, for each $v \in V \setminus \{s\}$, the *number* of shortest paths from s to v. We divide this task into that of solving two subproblems.

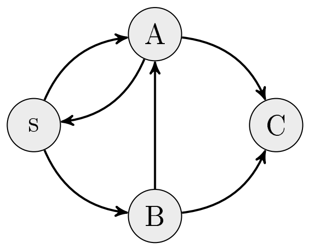
(Note that you can solve the second subproblem even if you did not manage to solve the first one.)

6a *(12 pts)* **Design and analyze** a polynomial time algorithm that finds a permutation $\pi : V \to \{1, 2, \ldots, n\}$ of the vertices so that

- $\pi(s) = 1$; and
- no shortest path from s to another vertex uses a "back-edge", that is, an edge $(u, v) \in E$ such that $\pi(u) > \pi(v)$.

For full score, you should argue why your algorithm returns a permutation that satisfies the above properties.

Example: Consider the following graph where all edges have weight 1:



A permutation satisfying the required properties would be $\pi(s) = 1, \pi(A) = 2, \pi(B) = 3, \pi(C) = 4$ because no shortest path from s uses the edges (B, A) and (A, s). Also note that the number of shortest paths from s to A is 1, from s to B is 1, and from s to C is 2.

Solution:

We find a permutation as follows:

1. Find the shortest path from s to all other vertices using Dijkstra's algorithm. Note that all weights are positive and we can therefore use Dijkstra's algorithm.
2. Let $d(v)$ be the length/weight of the shortest path from s to v. Now order the vertices in non-decreasing order according to $d(v)$'s, that is if v is in i possition of the d after sorting, then $\pi(v) = i$. One can see that the permutation π satisfies $\pi(u) < \pi(v) \rightarrow d(u) \leq d(v)$.

The running time of the above algorithm is clearly dominated by Dijkstra's algorithm which runs in time $O(|E| \log |V|)$ or $O(|E| + |V| \log |V|)$ with a more careful implementation. Indeed, the second step is a simple sorting which can be done in time $O(|V| \log |V|)$.

Let us now prove that the returned permutation satisfies the required properties. First, we have $\pi(s) = 1$ since $d(s) = 0$ and $d(v) > 0$ for all other $v \in V \setminus \{s\}$ because the edge-weights are strictly positive. Now, for the second property, assume toward contradiction that there is a back-edge (v_i, v_j) that is part of a shortest path $P = s, v_1, \ldots, v_i, \ldots, v_j, \ldots v_k$. First observe that the length of the path $s, v_1, \ldots, v_i, \ldots, v_j$ equals $d(v_j)$ since otherwise we could decrease the length of the path P. Similarly the length of path $s, v_1, \ldots, v_i$ equals $d(v_i)$. Therefore we must have that $w(v_i, v_j) = d(v_j) - d(v_i) > 0$. This is a contradiction if $\pi(v_i) > \pi(v_j) \Rightarrow d(v_i) \geq d(v_j)$ since all edge-weights are strictly positive.

It follows that we can efficiently find a permutation π as required.

6b *(13 pts)* Use the permutation π of the vertex set V from Subproblem 6a to **design and analyze** a polynomial time dynamic programming algorithm that fills in a table/array c indexed by the vertices in V so that $c[s] = 1$ and, for $v \in V \setminus \{s\}$, $c[v]$ equals the number of shortest paths from s to v.

(Hint: suppose that you calculated $c[u]$ for all u with $\pi(u) < \pi(v)$. How would you use this information to calculate $c[v]$?)

Solution:

Assume for simplicity that $V \setminus \{s\} = \{v_2, \ldots, v_n\}$ such that $\pi(v_i) = i$. We shall use the permutation to fill in the array c in a bottom-up fashion, starting from s we continue with v_2 and then v_3 and so on until we filled in the whole array. The advantage with this (also highlighted in the hint) is that when we fill in v_i we know $c[v_j]$ for all $j < i$ and moreover we know that no shortest-path uses a back-edge. Therefore the number of shortest paths that go from s to v_i is exactly $c[v_{j_1}] + c[v_{j_2}] + \ldots + c[v_{j_\ell}]$ where $\{v_{j_1}, \ldots, v_{j_\ell}\}$ are the vertices that can be visited just prior to v_i in a shortest path from s. That is $d(v_{j_1}) + w(v_{j_1}, v_i) = d(v_i)$, $\ldots$, $d(v_{j_1}) + w(v_{j_\ell}, v_i) = d(v_i)$ where $d(u)$ denotes the shortest path distance from s to u. This gives us the following pseudo-code for fill in c:

```
1. $c[s] \leftarrow 1$ and $c[v] \leftarrow 0$ for $v \in V \setminus \{s\}$
2. for $i = 2, 3, \ldots, n$
3.     for each $(u, v_i) \in E$ such that $d(u) + w(u, v_i) = d(v)$ (and therefore $\pi(u) < \pi(v)$)
4.         $c[v_i] \leftarrow c[v_i] + c[u]$
5. return $c$.
```

Note that we can find the incoming edges to each vertex similar to the way that we find the outgoing edges of each vertex. The time it takes is $O(n^2 + |E|) = O(n^2)$ as in each iteration of the for-loop of Step 2 we need to check all incoming arcs to v_i which are at most n many and hence the for-loop runs at most n iterations each time. Also finding all the incoming edges can be done in $O(|E|)$.