CS218 Design and analysis of algorithms **Jan-Apr 2025**

Endsem

Total Marks: 90 *Time: 3 hours*

Instructions.

- Please write your answers clearly and concisely.
- You can directly use any algorithm discussed in this or previous courses.

Question 1 [4 marks]. Consider the following algorithm for computing **minimum** weight spanning tree. Input is a connected graph $G(V, E)$ with distinct edge weights. Sort the edges in decreasing order of weights. Let the edges be $e_1, e_2, \ldots, e_m$ in decreasing order of weights. We will be going over edges in this order, i.e., highest weight first.

---

for ($j \leftarrow 1$ to m) {
  if _____ { $E \leftarrow E \setminus \{e_j\}$; } //delete e_j
}
return E;

What condition should we put in the blank, so that the output edge set forms the minimum weight spanning tree. Describe the condition in a sentence, a pseudocode is not expected. The condition should not use weights of the edges.

Answer 1. There are multiple ways to describe the correct answer. Any one of the following or something which implies the same is correct.

- Removing e_j keeps the graph connected.
- e_j is part of some cycle in the graph $G(V, E)$.
- e_j is not a "bridge" edge. That is, removing e_j should not disconnect the graph.
- There is path between the endpoints of e_j (which is not using e_j).

Any other equivalent statement gets full marks.

Question 2 [10 marks]. If someone proves P $\neq$ NP, which of the following can we conclude with certainty? For each point below, say yes (we can conclude) or no (we cannot conclude) and give 1-2 line justification for your answer. Marks only when justification is correct.

(a) There is no polynomial time algorithm for any problem in NP.

(b) We will have a polynomial time algorithm for the Boolean satisfiability problem (SAT).

(c) There is no $O(n)$ time algorithm for the Boolean satisfiability problem (SAT).

(d) To solve SAT, any algorithm must take exponential time ($\Omega(2^n)$).

(e) The s-t path problem (deciding whether there is a path between a pair of vertices in a graph) is not NP-complete.

Answer 2.

(a) No, we cannot conclude. Because any problem in P is also in NP, which certainly has a polynomial time algorithm.

(b) No. This is clearly false, because this would imply P = NP, which contradicts the assumption in question.

(c) Yes. We can conclude this because P $\neq$ NP implies there is no polynomial time algorithm for SAT, in particular there is no linear time algorithm.

(d) No, we cannot conclude this. Even if P $\neq$ NP, it is possible that there is an algorithm for SAT that takes, say quasi-polynomial $O(2^{\log^2 n})$ time or sub-exponential $O(2^{\sqrt{n}})$ time, which is not exponential.

(e) Yes. The path problem is known to be in P. So, it cannot be NP-complete, otherwise we will have P = NP which contradicts the assumption in the question.

If someone answers "cannot say for sure" instead of No, that is acceptable.

Question 3 [6 marks]. We have an array A with n integers sorted in increasing order and an array B with n integers sorted in decreasing order. Consider a game. You can choose any number i from $\{0, 1, \ldots, n-1\}$. Then you will be given either $A[i]$ or $B[i]$ points, whichever is minimum. Naturally, you would like to choose an i which gives you maximum points.

Example:
$A = [2, 5, 7, 10, 11, 12, 15, 18, 25, 45]$.
$B = [50, 40, 32, 25, 22, 20, 18, 16, 10, 5]$.
Here the maximum points you can get is 16 by choosing $i = 7$.
How will you find such an i in $O(\log n)$ time?

Answer 3. We will do binary search for an index i such that

- for all $j \leq i$, we have $A[j] \leq B[j]$
- and for all $j \geq i+1$, we have $A[j] \geq B[j]$.

Then either i or $i+1$ will be the desired index.

First check if $B[n-1] \geq A[n-1]$, if yes then return $n-1$. Then check if $B[0] \leq A[0]$ if yes then return 0. Even if we do not handle this corner case explicitly, the below code takes care of it anyway.

Initialize left $\leftarrow 0$ and right $\leftarrow n-1$.
//Invariant: $A[left] \leq B[left]$ and $A[right] > B[right]$.
While (right $-$ left > 1) {
    mid = (left+right)/2;
    if ($A[mid] \leq B[mid]$) then left $\leftarrow$ mid;
    otherwise right $\leftarrow$ mid;
}
if ($A[left] \leq B[right]$) return right;
otherwise return left;

It is fine if one assumes that in each array the numbers are all distinct. The above code should work even without this assumption.

It is fine even if the exact implementation is not shown. Crucial idea is when to go left and when to go right in the binary search. At the end, one has to compare two neighboring indices, if this is not checked then **-2 marks.**

Question 4 [2+2 marks]. For a degree 5 polynomial (6 coefficients), you are given its evaluations at 10 distinct points.

(a) What is the maximum number of evaluations one can **hide** so that the polynomial can still be determined uniquely from the remaining evaluations?

(b) What is the maximum number of evaluations one can **corrupt** arbitrarily so that the polynomial can still be determined uniquely?

Answer 4. (a) 4

Because the 6 coefficients can be recovered from any 6 evaluations (Explanation not expected.)

(b) 2

First let us see what if 3 evaluations are allowed to be corrupted. An adversary can choose a different polynomial of degree 5 which agrees with first 5 evaluations (this is always possible). Then from the remaining 5 evaluations, it can corrupt 3 of them so that those also agree with the chosen polynomial. This way, an incorrect polynomial agrees with 8 evaluations, while the correct polynomial may agree on only 7.

Now, let us see if only 2 evaluations are allowed to be corrupt. Then the correct polynomial agrees with 8 evaluations. We claim that there is no other polynomial of degree 5 which agrees with some subset of 8 evaluations. Because if it does then it will agree with at least 6 uncorrupted evaluations. And there is a unique solution from 6 evaluations.

(Explanation not expected.)

Question 5 [6 marks]. Express the following optimization problem as a linear program (you do not need to solve the linear program). We are given four real numbers a, b, c, d and we want to

$$\begin{gathered}\text{minimize } (\max\{x, y\} + |z|) \\ \text{subject to} \\ ax + by + cz = d. \\ x, y, z \in \mathbb{R}\end{gathered}$$

Answer 5. There are multiple ways of doing this. Here is one.

$$\begin{gathered}\text{minimize } (u + v) \\ \text{subject to} \\ u \geq x \\ u \geq y \\ v \geq z \\ v \geq -z \\ ax + by + cz = d. \\ x, y, z, u, v \in \mathbb{R}\end{gathered}$$

Question 6 [8 marks]. You have to interview n candidates, and you have fixed a set S of days to schedule interviews. Each candidate has given a subset of days (need not be an interval) on which they are available. Let $S_i \subseteq S$ be the subset given by candidate i. Then the candidate i can be interviewed on one of the days from S_i. You have to make a schedule so that each candidate gets assigned a day. You do not want to keep too many interviews on a single day. Give a polynomial time algorithm to make a schedule that minimizes the maximum number of interviews you have to do on any day.

Example $S = \{1, 2, 3, 4, 5, 6\}$, $S_1 = \{1, 3, 4\}$, $S_2 = \{4, 5\}$, $S_3 = \{4, 5\}$, $S_4 = \{5\}$. Here is an optimal schedule Day 1: Candidate 1, Day 4: Candidates 2 and 3, Day 5: Candidate 4. In this schedule, we have to interview maximum 2 candidates in a day.

Answer 6. Let us solve a related problem. Given a number k, is it possible to make a schedule so that on any day we have at most k interviews. If we can solve this, then we can find the minimum k by binary search or just linear search.

The mentioned problem can be solved using bipartite matching or max flow. Both reductions are similar. Let us see how the reduction to max flow works.

- Create a vertex for every day in $\cup_i S_i$ (the days outside this are not relevant).
- Create a vertex for every candidate.
- Create source and sink vertices.
- Add an edge from source to every day vertex with capacity k.
- Add an edge from every candidate vertex to sink with capacity 1.
- Add an edge from a day vertex to a candidate vertex with capacity 1, if the candidate is available on that day.

Compute the max flow on this network. If the max flow value is same as the number of candidates, then the flow gives us a schedule which assigns a day to every candidate (iff the corresponding day-candidate edge has 1 unit of flow). Because of the capacity constraint k on source-day edge, it is implied that a day can get at most k candidates assigned.

If the max flow value is less than the number of candidates then there is no schedule possible with at most k interviews on each day.

The above network can be reversed – source and sink interchanged and all edges reversed.

One can also give a solution via bipartite matching. Create k vertices for each day. Create one vertex for every candidate. Add an edge between a day vertex and a candidate vertex if and only if the candidate is available on that day. A schedule is possible if and only if the maximum matching size is the number of candidates.

Various greedy algorithms have been proposed. 1 mark has been given for greedy strategies. They do not work. Here is an example, where most greedy strategies should fail.

8 Candidates, 8 days. $S_1 = S_2 = S_3 = \{1, 2, 3\}, S_4 = \{1, 2, 3, 8\}, S_5 = S_6 = S_7 = \{4, 5, 6, 7\}, S_8 = \{7, 8\}$.

There was another algorithm via maximum matching. Compute maximum matching, delete matched candidates and continue like this till every candidate gets matched. This may not achieve minimum number of interviews on a day. Example. $S_1 = \{1\}, S_2 = \{1\}, S_3 = \{1, 2\}, S_4 = \{2\}$. If one chooses maximum matching as (candidate 3, day 1) and (candidate 4, day 2), then the remaining candidates will get day 1. That would mean 3 interviews on day 1, but optimal is 2.

Question 7 [2 + 6 marks]. Consider a variant of the above question, where you are given $S_1, S_2, \ldots, S_n$ and a threshold k. You are allowed to schedule at most k interviews on any day. Given a number t, we want to decide whether all interviews can be scheduled using at most t days.

Example $k = 4$. $S_1 = \{1, 3, 4\}$, $S_2 = \{4, 5\}$, $S_3 = \{4, 5\}$, $S_4 = \{5\}$. Suppose $t = 2$. Then the answer is yes. We can schedule all interviews using Day 3 and Day 5. Candidate 1 on Day 3, remaning three candidates on Day 5.

(a) Prove that this problem is in NP.

(b) Prove that this problem is NP-hard (i.e. using a subroutine for this problem you can solve a known NP-hard problem).

You can assume the following problems are NP-hard: 3-SAT, subset-sum, independent set, Hamiltonian cycle. If you want to use NP-hardness of some other problem, then you will have to prove it.

Hint: if needed, you may fix k to be a large number.

Answer 7. (a) If the answer is yes, then the desired schedule is the certificate. One can verify easily that the schedule uses at most t days and each day has at most k interviews scheduled. One should also verify that each candidate is assigned exactly one day and that they are available on that day.

(b) For NP-hardness, we will reduce independent set problem to the given problem.

Indepenent set: Given a graph $G(V, E)$ and a number ℓ, is there an independent set of size ℓ?

Reduction Given a graph G and a number ℓ, we will create an instance of the given problem as follows. For every edge e in the graph, create a candidate C_e. For every vertex v, create a day D_v. If the edge e has endpoints u, v, then candidate C_e is available on two days D_u and D_v. Set k to be the number of vertices, say n. Set t to be $n - \ell$.

Claim 1. *The graph G has an independent set of size ℓ if and only if there is a schedule which uses at most $t = n - \ell$ days.*

Proof. ($\implies$). Let I be an independent set of size ℓ. Consider the complement set $V \setminus I$. Note that by definition of an independent set, every edge has at least one endpoint in $V \setminus I$. For each candidate C_e, let us assign the day corresponding to one of the endpoints of e, which lies in $V \setminus I$ (if both endpoints lie then choose arbitrarily). Clearly, each candidate gets exactly one day. Moreover, on any day D_u, the maximum possible number of candidates assigned is the degree of D_u, which is at most $k = n$. Thus, this is a valid schedule. Clearly, it use at most $t = n - \ell$ days (some days from $V \setminus I$ may be unused).

($\impliedby$). Let there be a schedule, which assigns every candidate a day. Let T be the set of days used by the schedule. Note that a candidate C_e can only be assigned a day which corresponds to one of the endpoints of e. That means, every edge has at least one of its endpoints in set T. Equivalently, $V \setminus T$ is an independent set. Thus, we have an independent set of size $n - |T| \geq n - t = \ell$. □

3 marks for construction, 1.5 marks for each direction of the proof.

Question 8 [4+6 marks]. Consider the roommate allocation problem, where each room can have at most 2 students. There are n students and we are given a set E of pairs of students such that $(i, j) \in E$ if and only if student i and student j are fine with sharing a room. We want to minimize the number of rooms needed, with the constraint that two students i, j are allocated the same room only if $(i, j) \in E$. We run the following algorithm.

While (E is non-empty) {
  pick an arbitrary pair $(i, j) \in E$ and allocate one unassigned room to i and j.
  remove all pairs from E which involve i or j.
}
Each of the remaining students is put in a distinct unassigned room.

(a) Give an example, where the above algorithm does not give an optimal allocation. You should show the output of the algorithm and another allocation with smaller number of rooms.

(b) Prove that the above algorithm gives 3/2 approximation. That is, the number of rooms used by the algorithm is at most 3/2 of the optimal number of rooms. *Hint:* A lower bound on optimal number of rooms is the number of rooms which the above algorithm assigned only one student. If you want to use this then you need to argue why. There may be other lower bounds.

Answer 8. (a) Simple example. Four students. $E = \{(1, 2), (1, 3), (2, 4)\}$. If the algorithm first puts $(1, 2)$ in one room, then it has to assign 3 and 4 distinct rooms. So, in total 3 rooms. On the other hand, the optimal solution is use 2 rooms, one for $(1, 3)$ and other for $(2, 4)$.

(b) Suppose the algorithm uses x rooms for double occupancy and y rooms for single occupancy, that is, it uses $x + y$ rooms in total. And total number of students is $2x + y$.

First we claim that y is a lower bound on the number of optimal rooms. This is because, the algorithm assigns single occupancy only when in the remaining set of students, say Y, there is no edge, i.e., no one in set Y is fine with any one else in set Y. That means, each student in set Y must get a distinct room (with possibly some partner from outside Y). Thus, $|Y| = y$ is a lower bound on optimal number of rooms.

3 marks if this lower bound proved correctly.

Now, there is another trivial lower bound, which is half of total number of students. This is because a room can have at most 2 students. 1 mark.

The two lower bounds we got are as follows.

$$OPT \geq y.$$

$$OPT \geq (2x + y)/2.$$

Let us multiply $1/2$ in first inequality and add with second inequality. We get,

$$(3/2)OPT \geq x + y.$$

In other words, the number of rooms used by the algorithm $(x+y)$ is at most $3/2$ times the optimal number.

2 marks if this analysis done correctly.

Question 9 [4+4 marks]. Recall the definition of Fourier transform. Let ω be a primitive d-th root of unity. For a given length-d tuple $(a_0, a_1, \ldots, a_{d-1})$, we want to compute d values, i.e.,

$$a_0 + a_1\omega^i + a_2\omega^{2i} + \cdots + a_{d-1}\omega^{(d-1)i},$$

for each $0 \leq i \leq d-1$. We had discussed a divide and conquer approach for fast Fourier transform. It is possible to do an iterative implementation, which is memory efficient. For simplicity assume $d = 8$. Consider the following pseudocode.

(a) Find the values of $\mathbf{f(0)}, \mathbf{f(1)}, \mathbf{f(2)}, \mathbf{f(3)}$, which would be appropriate for line 5 in the pseudocode.

(b) For $d = 16$, how will line 2 and line 4 change?

---

1. Let A be a length 8 array (indexed from 0 to 7) initialized as $[a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7]$.
2. for $k = 2$ to 0 {
3. for $i = 0$ to $(2^k - 1)$ {
4. for $j = 0$ to $(\frac{4}{2^k} - 1)$ {
5. define $index_1 = i + \mathbf{f(j)}$ and $index_2 = index_1 + 2^k$.
6. Initialize two temporary variables $temp_1$ and $temp_2$.
7. $temp_1 \leftarrow A[index_1] + \omega^{j \times 2^k} \times A[index_2]$
8. $temp_2 \leftarrow A[index_1] - \omega^{j \times 2^k} \times A[index_2]$.
9. Copy $temp_1$ into $A[index_1]$ and $temp_2$ into $A[index_2]$.
10. }
11. }
12. }

At the end, array A would look like $[P(\omega^0), P(\omega^4), P(\omega^2), P(\omega^6), P(\omega), P(\omega^5), P(\omega^3), P(\omega^7)]$, where

$$P(x) = a_0 + a_1x + a_2x^2 + \cdots + a_7x^7.$$

Answer 9. (a) $\mathbf{f(0) = 0, f(1) = 4, f(2) = 2, f(3) = 6}$.

One mark for each correct answer.

(b)

Line 2. for $k = \mathbf{3}$ to 0 {

Line 4. for $j = 0$ to $(\frac{\mathbf{8}}{2^k} - 1)$ {

Two mark for each.

Question 10 [6 + 6 marks]. Consider the makespan problem with **three** processors. We are given n jobs with their processing times $p_1, p_2, \ldots, p_n$. We want to assign each job to a processor so that the makespan (maximum total load on any processor) is minimized.

(a) Design a pseudo-polynomial time algorithm find the minimum makespan, i.e., the time complexity should be polynomial in n and $\sum_{i=1}^{n} p_i$.

(b) Now, we want to use this idea to design a polynomial time algorithm, which gives an approximate solution. Suppose we want 1.1-approximation for makespan. We choose an appropriate number N and define new processing times $p'_i = \lfloor \frac{p_i}{N} \rfloor$. Then we run the above algorithm for $p'_1, p'_2, \ldots, p'_n$ and we output the same job assignment for the original problem. How should we choose N so that the time complexity is poly(n) and the makespan we get is at most 1.1 times the optimal makespan? You need to prove both the guarantees.

Answer 10. Let $P_j = \sum_{i=1}^{j} p_i$. Create a 3 dimensional Boolean table T with dimensions $n \times (P_n + 1) \times (P_n + 1)$. $T[i, x, y]$ will denote whether it is possible to assign first i jobs in way that total load on processor 1 is x and total load on machine 2 is y and total load on machine 3 is $P_i - x - y$.

For any i, x, y, the table entry will computed as follows.

$$T[i, x, y] = T[i-1, x-p_i, y] \text{ OR } T[i-1, x, y-p_i] \text{ OR } T[i-1, x, y].$$

This is just saying that the ith job will be assigned to one of three machines, and depending on which, we get the remaining loads on three machines. Clearly, the table T can be computed in increasing order of i, x, y.

Base case: $T[1, p_1, 0] = T[1, 0, p_1] = T[1, 0, 0] =$ TRUE. For any other entry $T[1, \cdot, \cdot] =$ FALSE.

To find the minimum makespan, we go over all the entries $T[n, x, y]$ of the table which are TRUE. For each entry, the makespan is $\max\{x, y, P_n - x - y\}$. We find the minimum of all these makespans. Formally,

$$\min\{\max\{x, y, P_n - x - y\} : T[n, x, y] = \text{TRUE.}\}$$

The running time of this algorithm will be $O(n \times P_n \times P_n)$. Thus, it is pseudo-polynomial.

(b) For time complexity to be poly(n), we must choose N such that $\sum_i p'_i$ is poly(n). That is, we should choose $N = (\sum_i p_i)/n^c$ for some constant c.

Let the optimal makespan for the given problem be M^* and let the corresponding job assignment be A^*. The same job assignment A^* will have makespan at most $\frac{M^*}{N}$, with respect to the new processing times. The algorithm will find optimal makespan M' for the new processing times, say with job assignment A'. Hence,

$$M' \leq \frac{M^*}{N} \implies NM' \leq M^*. \tag{1}$$

Now, let us analyze the assignment A' with respect to the original processing times. First observe that for any i, we know $p_i < Np'_i + N$.

Now, for processor j, let S_j be the set of jobs it gets in assignment A'. Thus, the load on this processor (with respect to original processing times)

$$\sum_{i \in S_j} p_i \leq \sum_{i \in S_j} (Np'_i + N) \leq NM' + N|S_j| \leq NM' + Nn \leq M^* + Nn.$$

The last inequality is from (1). Thus, we can say that makespan of assignment A' is at most $M^* + Nn$. Now, we want $M^* + Nn \leq 1.1M^*$. Thus, we should choose $N \leq M^*/(10n)$. It is sufficient to choose $N = (\sum_i p_i)/(3 \times 10n)$, because M^* is at least average load per processor.

For this choice of N, the time complexity will be $O(n \times (\sum_i p_i/N)^2) = O(n^3)$.

2 marks for arguing polynomial bound. 4 marks for proving approximation guarantee.

Question 11 [8 marks]. In this question, we will prove an approximate max flow min cut theorem. Consider the following variation of the max flow algorithm. We are given a directed graph $G(V, E)$ with integer capacities for edges and a source s and a sink t. Recall that for a path in the residual graph, its bottleneck is defined as the minimum residual capacity of any edge on the path.

MaxFlow(D, f) : Here D is a positive integer. And $f : E \to \mathbb{R}$ is an initial flow in the network (can be zero).

---

1. While (true) {
2. Try to find a path from s to t in the residual graph G_f with bottleneck at least D.
3. If there is no such path, then break the loop.
4. Otherwise let P be such a path.
5. Push bottleneck(P) amount of flow along path P. For each edge e, update $f(e)$ appropriately.
6. Update the residual graph G_f accordingly.
7. }
8. Return f.

Recall that the total flow going out of s ($f^{out}(s)$) is upper bounded by capacity (say C^*) of minimum s-t cut. Prove that at the end of the above algorithm

$$f^{out}(s) \geq C^* - |E| \cdot D.$$

Answer 11. The algorithm stops only when there is no s-t path in the residual graph G_f with bottleneck at least D. In other words, any s-t path has at least one edge with residual capacity less than D. If from the residual graph, we delete all edges with residual capacity less than D, then there will be no s-t path. Consider this modified residual graph G'_f. Let U be the set of vertices reachable from s in G'_f. This means, all the outgoing edges from U to outside U in G_f must have residual capacity less than D. Let us analyze the total outgoing flow from U to outside U.

For any edge $e \in E[U, \bar{U}]$, we can say the corresponding forward edge's residual capacity is $c_e - f(e)$, which is less than D from above. Thus, we have $f(e) > c_e - D$.

For any edge $e \in E[\bar{U}, U]$, we know the corresponding backward edge's residual capacity is $f(e)$. Thus, we have $f(e) < D$.

The net outgoing flow from U, say $f^{out}(U)$ will be

$$\begin{aligned}
f^{out}(U) &= \sum_{e \in E[U,\bar{U}]} f(e) - \sum_{e \in E[\bar{U},U]} f(e) \\
&> \sum_{e \in E[U,\bar{U}]} (c_e - D) - \sum_{e \in E[\bar{U},U]} D \\
&\geq \sum_{e \in E[U,\bar{U}]} c_e - \sum_{e \in E[U,\bar{U}] \cup E[\bar{U},U]} D \\
&\geq \sum_{e \in E[U,\bar{U}]} c_e - |E| \cdot D \\
&\geq cap(U) - |E| \cdot D. \\
&\geq C^* - |E| \cdot D.
\end{aligned}$$

Note that $f^{out}(U)$ is same as $f^{out}(s)$. Thus we have the desired inequality.

Many solutions had some intuition, but not completely formal.

Example 1. After the algorithm is finished, every path has at least one edge with capacity less than D. There are at most $|E|$ paths, hence, the difference from max flow is at most $D \cdot |E|$. This is not correct, because number of paths can be exponential. Some people meant disjoint paths, but then it is not clear which disjoint paths to consider and why only disjoint paths should be considered.

Example 2. Some other solutions mentioned that each edge can be used in only one augmenting path. Again this is not clear. Recall that the standard algorithm can have many augmenting path iterations, as large as the capacities.

Some have argued that flow can be increased by pushing more flow along edges whose capacity and flow differ by at most D. Note that flow can also increased by using backward edges, that is, by reducing the flow along some edge coming towards the source.

The correct way to argue is via a cut. Some have not specified which cut they are talking about. Some have argued via min cut. What we need is a cut where any edge going out has residual capacity less than D. It is not clear if min cut will have this property.

These kinds of incomplete solutions were given 3 or 4 marks.

Some have described the correct cut and the right argument, but missed some other detail from the solution, then they were given 6 or 7 marks.

Question 12 [4+2 marks]. Assume the inequality stated in the previous question even if you cannot prove it.

Recall that the max flow algorithm we saw in the class was pseudo-polynomial time. That is, the time complexity was proportional to the max flow value, which may be exponential in the input size. Here is an idea for designing a polynomial time algorithm.

---

1. Let the initial flow f in the network be zero.
2. Initially we set D as the largest power of 2 smaller than the maximum capacity of any edge.
3. While $D \geq 1$ {
4. $\quad f \leftarrow MaxFlow(D, f)$; // procedure from previous question
5. $\quad D \leftarrow D/2$.
6. $\quad$ }
6. Return $f^{out}(s)$.

(a) Prove that in any one call of the MaxFlow subroutine, there can be at most $O(|E|)$ augmenting iterations.

(b) Argue that the algorithm outputs the correct value of max flow in the network. You can directly use the fact that the algorithm seen in the class was correct.

Answer 12. (a) Consider a particular call $MaxFlow(D, f)$ in the algorithm. The previous call must have been $MaxFlow(2D, f)$. From Answer 11, we know that at the end of $MaxFlow(2D, f)$, we have

$$f^{out}(s) \geq C^* - |E| \cdot 2D.$$

That is, the outgoing flow from s is close to the maximum possible flow C^*. The further increase in the flow can be at most $|E| \cdot 2D$ units. Observe that in $MaxFlow(D, f)$, every augmenting iteration increases flow by at least D units. Hence, the number of iterations will be at most $2|E|$.

(b) The last call in the algorithm is $MaxFlow(D = 1, f)$. When $D = 1$, then the subroutine $MaxFlow(1, f)$ is nothing but the standard algorithm seen in the class. We know that that algorithm stops only when we have the maximum possible flow.