(1) Finding Primes. (10 points) In class on October 7 and 9, we discussed the Rabin-Miller prime testing algorithm that has the following guarantee for an input integer N

1. if N is prime, the algorithm is guaranteed to output "N is probably prime"
2. if N is composite (not prime), the algorithm has at least a 50% probability that it will output that "N is not prime"

So if the algorithm concludes N is not prime, this is definitely correct, but when the algorithm outputs "N is probably prime", this can happen if N is prime, or if N is not prime, but we are in the unlucky 50% of case 2. Recall that it runs in time polynomial on $\log N$ (more precisely at most $O(\log^3 N)$ time, using $2\log_2 N$ multiplications on numbers that are at most N).

We also agreed that if we run this algorithm k times, and output "N is not prime" if any one of the runs showed that N is not prime, we get the following improved guarantee (at the expense that the algorithm is now k times slower):

1. if N is prime, the algorithm is guaranteed to output "N is probably prime"
2. if N is composite (not prime), the algorithm has at least a $(1 - 2^{-k})$ chance that it will output that "N is not prime", but with the remaining 2^{-k} probability it may still output "N is probably prime".

We will call this algorithm *Miller-Rabin-k*. So of we run *Miller-Rabin-k* for a number N with $k = 7$, and find "N is probably prime", we can be $127/128 > 99\%$ sure that N is actually prime.

Now recall that an important application of this algorithm is finding large primes. To do this, we take advantage of *Prime number theorem* from number theory stating that a random integer in the range $[2, \ldots, P]$ has approximately a $\frac{1}{\log P}$ probability of being prime, where $\log P = \log_e P$ is the natural logarithm of P. Here is a proposed algorithm:

[Find Prime$[P, k]$**]**
While no "probably prime" found
  Select a random number $N \in [2, \ldots, P]$
  run *Miller-Rabin-k* with input N
    if output "N is probably prime" **Output** N
endWhile

The expected running time of this algorithm is $O(k \log^4 P)$ as the *Miller-Rabin-k* $O(k \log^3 P)$ time, and in expectation we need to repeat it at most $\log P$ times by the *Prime number theorem.*

Recall that the output number N can either be prime, or be a composite number on which the *Miller-Rabin-k* test failed.

(a) Show that for a given P and k, the probability that the resulting output N is prime is at least $\frac{2^k}{2^k + \log P}$

(b) How large should k be if we want to be at least 99% sure that the number we output is prime.

Solution 1: We want to use Bayes's theorem to get the probability that the output is prime. Recall Bayes' theorem considers two random events A and B, and asks what is the probability of A being true, conditioned on observing B, which is denoted by $Pr(A|B)$. So for us

- A="N is prime"
- B="The algorithm output 'N is probably prime"

Bayes' theorem says

$$Pr(A|B) = \frac{Pr(B|A)Pr(A)}{Prob(B)}$$

So $Pr(B|A) = 1$, as we know the algorithm always outputs "N is probably prime" if N is prime.

Further, from the prime number theorem above, we know that $Pr(A) = \frac{1}{\log P}$.

This leaves us having to compute $Pr(B)$. This can happen in one of two reasons: $Pr(B) = Pr(B|A)Pr(A) + Pr(B|\bar{A})Pr(\bar{A})$, where $\bar{A}$ stands for A is not true. The first term is $Pr(B|A)Pr(A) = \frac{1}{\log P}$ from above. The second term needs $Pr(B|\bar{A})Pr(\bar{A}) \leq 2^{-k}$. Putting this together we get

$$Pr(A|B) = \frac{Pr(B|A)Pr(A)}{Prob(B)} \geq \frac{\frac{1}{\log P}}{\frac{1}{\log P} + 2^{-k}} = 1 - \frac{2^k}{\log P} = \frac{2^k}{2^k + \log P}$$

To makes sure that he output is 99% surely prime, we need that

$$\frac{2^k}{2^k + \log P} \geq \frac{99}{100}$$

that is $100 \cdot 2^k \geq 99(2^k + \log P)$ which happens is $2^k \geq 99 \log P$, so we need $k \geq log_2 99 + \log_2 \log P \approx \log \log P + 7$

Alternate solution 2: Using events A and B as above. Since prime numbers do pass the prime number test for sure, we do know that $A \subset B$. Using this fact, it then directly follows from the definition of conditional probability that

$$P(A|B) = \frac{Pr(A)}{Pr(B)}$$

to use this fact, we need to compute $Pr(B)$ as was done above.

Alternate solution 3: The probability that one iteration doesn't output any number is at most

$$(1 - 1/\log N)(1 - 2^{-k})$$

this happens when we pick a non-prime (which has probability $(1-1/\log N)$), and have the test correctly observe that the number we picked is not a prime (which is at least $(1 - 2^{-k})$). The probability that an iteration outputs a non-prime is at most $(1 - 1/\log N)x$. Now the algorithm outputs a non-prime, if it goes on for some i iteration and on the $(i+1)$st iteration it outputs a non-prime. Summing over all i this probability is at most

$$\leq \sum_{i=0}^{+\infty} ((1 - 1/\log N)(1 - 2^{-k}))^i (1 - 1/\log N)2^{-k}$$

Using that $\sum_i a^i = 1/(1-a)$ we get that this is

$$= (1 - 1/\log N)2^{-k} \frac{1}{1 - (1 - 1/\log N)(1 - 2^{-k})} \quad (1)$$

$$= (1 - 1/\log N)2^{-k} \frac{2^k \log N}{2^k \log N - (\log N - 1)(2^k - 1)} \quad (2)$$

$$= (1 - 1/\log N) \frac{\log N}{\log N + 2^k - 1} \leq \frac{\log N}{\log N + 2^k} \quad (3)$$

The probability that the algorithm outputs a prime is at least 1 minus this number, so that is at least $\frac{2^k}{\log N + 2^k}$ as claimed.

Common Mistakes: The most common mistake is writing a chain of equations without any word of explanation. A few words are enough to explain what you are doing, but we took some points off for students with no explanations.

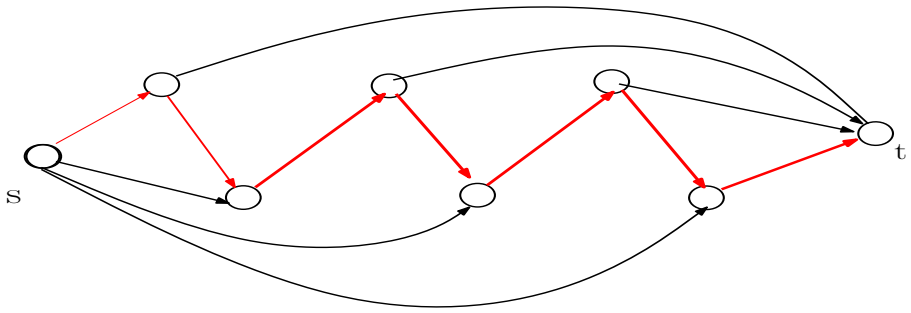
(2) "Max-value-forward-edge-only" flow algorithm. (10 points) Your friends have written a very fast piece of maximum flow code based on repeatedly finding augmenting paths as in Section 7.2. However, after you've looked at a bit of output from it, you realize that it's not always finding a flow of *maximum* value. The bug turns out to be pretty easy to find; your friends never figured out how backwards edges in a residual graph worked when writing the code, and so their implementation builds a variant of the residual graph that *only includes the forward edges*. On the other hand, the code is doing a great job selecting an augmenting path: at each iteration it is selecting the path that allows the maximum augmentation. In other words, it searches for s-t paths in a graph $\tilde{G}_f$ consisting only of edges e for which $f(e) < c_e$, find the path P such that $\delta = \min_{e \in P} c_e - f(e)$ is as high as possible, and it terminates when there is no augmenting path consisting entirely of such edges. We'll call this the "max-value-forward-edge-only" flow algorithm "with maximum augmentation path selection".

It's hard to convince your friends they need to re-implement the code; in addition to its blazing speed, they claim in fact that it never loses much compared to the maximum flow. Concretely, they claim that the resulting flow always has value at least half of the true maximum flow for the graph. Do you believe this? To be precise the proposed claim is as follows:

> *On every instance of the maximum flow problem, the max-value-forward-edge-only algorithm with maximum augmentation path selection is guaranteed to find a flow of value at least* $1/2$ *of the maximum flow value.*

Decide whether you think this statement is true or false. If true, prove the statement. If false, give a counter example and explain how this claim could be false for the counter example.

Solution: Not true. For example, consider the graph below, and assume capacities on the red edges is 1.01 while the black edges have capacity 1 (or also OK if all capacities are 1). If we first choose the red path, there will be no further augmentation available, so the flow value will be 1.01, while the max flow has value 4.



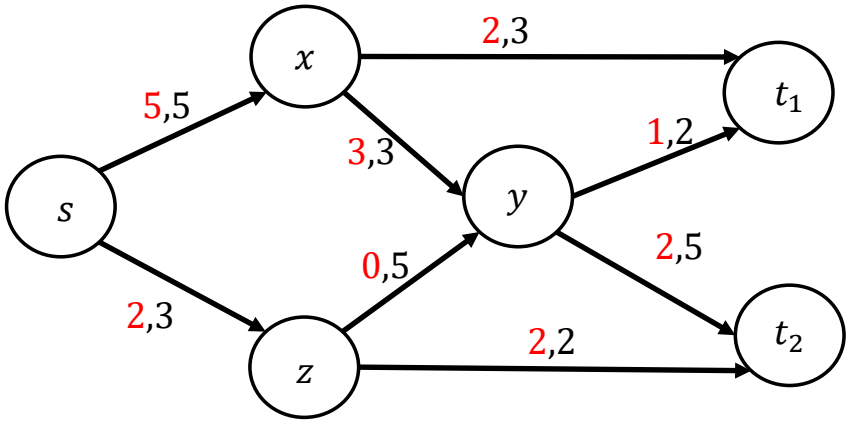
Common mistakes: Some students only offered a an example with the forward only flow is exactly $1/2$ of the maximum possible.

(3) Fair Max-flow. (10 points) Consider the following variant of the max-flow problem. We are given a directed graph $G = (V, E)$ with n nodes and $m \geq n$ edges, and integer capacities $c_e > 0$ on the edges, a source s and a two sinks t_1 and t_2 (assume neither t_1 not t_2 has edges leaving it). We will be interested in a maximum flow that sends from to both t_1 and t_2, and does so in a fair way. We say that a flow for this problem is any vector that satisfies the following constraints

$$0 \leq f(e) \leq c_e \text{ for all edges } e \in E. \text{ (capacity constraint)}$$

$$\sum_{e \text{ into } v} f(e) = \sum_{e \text{ out of } v} f(e) \text{ for all nodes } v \neq s, t_1, t_2. \text{ (flow conservation)}$$

with the value for t_1 and t_2 defined as $v_i(f) = \sum_{e \text{ into } t_i} f(e)$ for $i = 1, 2$. We say that this flow is *fair* if $|v_1(f) - v_2(f)| \leq 1$. The following figure is giving an example of a fair flow sending 3 units to t_1 and 4 units to t_2. It is not the maximum value fair flow as we can send another unit of flow along the s, z, y, t_1 path, making the total 4 units to both t_1 and t_2. Here the black number next to the edges is the capacity and the red number is the flow value.



Let $C_i = \sum_{e \text{ into } t_i} c_e$ for $i = 1, 2$. Give an $O(m(C_1+C_2))$ algorithm for finding a fair flow of maximum value.

Solution 1: Run the augmenting path algorithm alternating in finding an augmenting path s to t_1 and s to t_2, augmenting the flow always just with 1 unit (even if the path can take more). To be precise, on odd iterations we can accept a path to either t_1 or t_2, while on even iterations we insist that the path needs to go to the other terminal. While we always succeed in find an augmenting path our flow values either have $v_1(f) = v_2(f)$ (after an even number of augmentation), or $|v_1(f) - v_2(f)| = 1$ (after an odd number of augmentation). Terminate if no augmenting path is found. The running time is clearly bounded by $O(m(C_1+C_2))$ as each iteration takes $O(m)$ time and there are at most $O(C_1+C_2)$ iterations.

We claim that the resulting flow is the maximum fair flow. As usual consider the set of nodes A reachable from the source in the final residual graph $A = \{v : \text{ there is an } s \text{ to } v \text{ path in } G_f\}$. To prove this, we need to separate two cases

Case 1 $t_1, t_2 \notin A$. This must be the case if the algorithm terminates with $v_1(f) = v_2(f)$, but can happen even if the two values are not the same. In this case, there is no augmenting path to either t_1 or t_2. Adding a new terminal t with edges (t_1, t) and (t_2, t) of large (or infinite) capacity and extending the flow to t, we now found a maximum flow from s to t, so our flow is maximizing $v_1(f) + v_2(f)$

and has the two value each half of this maximum value (within possibly a +1 difference), so this is clearly a maximum value fair flow.

Case 2 when the algorithm terminates $|v_1(f) - v_2(f)| = 1$, say $v_1(f) = v_2(f) + 1$ (the other case is symmetric), and $t_2 \notin A$, but $t_1 \in A$. In this case we claim that the flow we found maximized $v_2(f)$. Note that the cut (A, B) we found is an (s, t_2)-cut and we claim that the capacity of this cut equal $v_2(f)$, which then shows that $v_2(f)$ is maximal, and hence our flow is a maximal fair flow, as $v_1(f)$ cannot be higher due to the fairness constraint.

To show that $c(A, B) = v_2(f)$ recall the proof from class:

$$\begin{aligned} v_2(f) &= \sum_{e \text{ into } t_2} f(e) \\ &= \sum_{e \text{ into } t_2} f(e) + \sum_{v \in B, v \neq t_2} \Big(\sum_{e \text{ into } v} f(e) - \sum_{e \text{ out of } v} f(e) \Big) \\ &= \sum_{e \text{ leaving } A} f(e) - \sum_{e \text{ entering } A} f(e) \\ &= \sum_{e \text{ leaving } A} c_e = c(A, B) \end{aligned}$$

where the first equation follows as all nodes $v \in B, v \neq t_2$ have flow conservation, the second is rearranging terms as was done in class, the third follows as edges leaving A are not in the residual graph by definition of the set A.

Common Mistakes with this solution:

- using Ford Fulkerson in its original form without limiting each augmentation to $\delta = 1$.
- alternating between the two sources, so if the flow is equal to the two sinks, and there is no path to t_1 stopping, rather than also checking if there is a path to t_2.
- proof is missing or very incomplete. Many students claim that not having an augmenting path implies that the flow is maximum without much further explanation. We only see how to argue this using cuts. For the case that $v_1(f) = v_2(f)$ the nodes reachable from the source, for a min-cut, and the proof from class can be adapted to argue that this is the max flow to the two sinks combined. But when one is higher value, say $v_1(f) = v_2(f) + 1$ we will still have paths to t_1, why is this max fair flow?

Solution 2: An alternate way to think about the above solution: add a new node t and edges (t_1, t) and (t_2, t) with the same capacity k on both edges. Start with $k = 1$, and while the algorithm finds a max flow of value $2k$ increase k by 1 and continue.

Running time: k can at the maximum go up to only $C/2$, so in at most $C/2$ iteration of the algorithm it will terminate. One iteration is at most 2 augmenting path, so takes $O(mC)$ time in total as required.

Correctness. Clearly the capacities on the two new edges ensure that the flow is fair throughout.

This version has a simple proof that this is maximum fair flow: If the maximum fair flow possible is at least $2k$ than the algorithm most find a flow with k on both edges, and continue to the next iteration. This shows that if the maximum fair flow is even (sending the same amount to both sinks), than the algorithm will find it. If the max fair flow value is odd (so the two sinks get different amounts, say k and $k + 1$, then the max flow with capacities $k + 1$ on both edges must have max flow value of $2k + 1$, and so will find the maximum fair flow.

Algorithms that don't work:

(A) Find max flow to t_1 and t_2 separately. Decide which of the two has smaller max flow. Say this is t_1. Now find a max flow f_1 to t_1, of value v_1, the max flow value. Revise the network to have flow f_1 go to t_1, and take the capacity remaining $c_e - f_1(e)$ as the capacity, and find a flow to t_2 in this

network of value v_1 or $v_1 + 1$. This "greedy" commitment to the flow to t_1 doesn't work, and also the bottleneck can be combined for the two sinks, so just because both terminals can take a flow of some value v, that doesn't mean we can send v to both of them together.

(B) Adding a super-sink the algorithm part with infinite capacity edges from t_1 and t_2, determining the max flow value v to the super-sink and then trying to send $\lfloor v/2 \rfloor$ and $\lceil v/2 \rceil$ to the two sinks with adding these as capacities on the edges to the super-sink. Doesn't work as there may not be this much flow to one of the sinks, but is a better start then the previous one.