Introduction to Algorithms	Problem Set 5 (3 problems)
CS 4820 Fall 2019	Due 11:59pm Thursday, October 24

There are three written questions on this homework.

Your homework submissions for written questions need to be typeset (hand-drawn figures are OK). See the course web page for suggestions on typing formulas. The solution to each written question needs to be uploaded to CMS as a separate pdf file. To help provide anonymity in your grading, do not write your name on the homework (CMS will know it's your submission).

Collaboration is encouraged while solving the problems, but

1. list the names of those with whom you collaborated with (as a separate file submitted on CMS);
2. you must write up the solutions in your own words;

Remember that when a problem asks you to design an algorithm, you must also prove the algorithm's correctness and analyze its running time.

(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.

(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.*

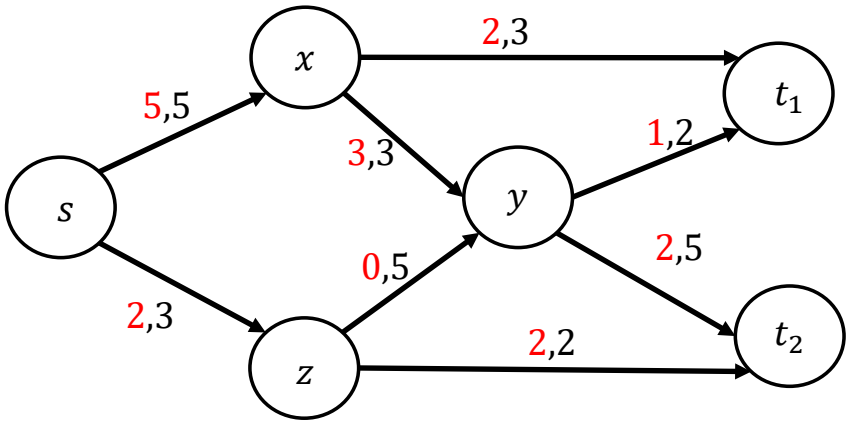
Decide whether you think this statement is true or false. If true, prove the statement. If false, give a counter example and explain how this claim could be false for the counter example.

(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.