task 1

by implementing the `push-relabel`-algorithm in odin, i was able to get a better understanding of the algorithm and could more tangibly reason about the following questions.

a) $n = m = C$

when the two towers have equally many nodes and the capacity of the middle nodes are non-limiting, we can attempt to derive a simple formula for the iteration count, given n.

the iteration count can be seen by running the algorithm on different configurations of the problem, i.e. for different n.

first, let's explore the impact of varying the parameters.

- by varying k, which is not specified in this task, we can see that increasing it by 1, increases the iteration count by 2. which makes sense, since we need to first relabel, then push, for each middle node. this happens regardless of what n is, as long as $n = m = C$.
- adding 1 to n adds 5 to the iteration count.

notice that $n = 3, k = 2$ yields 18 iterations.

using this, we can posit that the iteration count is given as

$$c(n, k) = 5n + 2k - 1$$

trying out this fancy formula, we can predict the outcome of using certain parameters:

- c(1, 0) = 4 ok.
- c(1, 1) = 6 ok.
- c(2, 1) = 11 ok.

- c(3, 3) = 20 ok.
- c(6, 6) = 41 ok.

seems to work fine.

b) $n = C, n \gg m$
in this case, the formula above won't work, because $n \neq m$.

so let's find the pattern again.

let $c(n, m, k)$ denote the iteration count again.

fix $k = 1$, then let row denote n and col m:

	1	2	3	4	5	6	7	8	9
1	6								
2	26	11							
3	44	47	16						
4	58	71	74	21					
5	85	88	104	107	26				
9	203	206	234	237	265	268	296	299	46

what's the pattern?

we can clearly see that along the diagonal, the previous formula holds. additionally, numbers along a row follow an arithmetic sequence with a difference of 3 – sometimes. there are some discrete jumps, and i don't know why. that becomes apparent in the last row, when $n = C = 9$.

it might be more interesting looking at when $m = k = 1$, and only varying $n = C$.

n	c	d
10	226	
11	280	54
12	306	26
13	369	63
14	398	29
...	...	...
20	746	
25	1155	409
30	1566	411
35	2140	574
...	...	...
50	4106	
75	9080	

the differences d are wild. it might be impossible to find an explicit formula for the iteration count in this case.

i suppose the general pattern is that the iteration count grows disproportionately to the number of nodes in the first tower, i.e. a lot faster, when $n \gg m$.

but let's think about this. when n grows and m is fixed, it just means the algorithm will have to relabel only a few nodes in the second tower, since there are only a few nodes there to begin with. thus, the major time spent in the algorithm will be in the first tower, relabeling and pushing flow through that network. and since $n = C$, the flow coming out of the first tower is not restricted in any way, but it poses a problem for the second tower, which then acts

as a bottle neck for the flow. a bottle neck would only mean that the algorithm terminates quicker, i.e. has a positive impact on the iteration count.

i'm sure you could argue about this more rigorously, but my current understanding is just that.

P.S: i found that the numbers got wild with $k = 0$, so i assume there's maybe a bug for that case, thus the fact that the formula in a) lined up for this might just be a coincidence (in the cases where i had $k = 0$).

c) $n \ll m, m = C$
so now the situation is opposite, where the flow out of the first tower is (possibly) restricted and the flow into the second tower is unrestricted, since $m = C$.

i think this should yield similar iteration counts as for b), but since we are limiting the flow earlier, i would think that the algorithm terminates quicker, yielding a lower iteration count. let's test it.

now let's fix $k = 1, n = 1$ and increase $m = C$ gradually to note the rate of change in the iteration count c.

m	c	d
5	6	
6	6	0
...	...	...
10	6	
20	6	0
...	...	...
100	6	

the pattern is immediately clear: it is not only a little faster than b), it is incredibly fast.

with $c(n, m, k)$ we get $c(1, 100, 1) = 6$ and $c(2, 100, 2) = 13$ and $c(2, 100, 1) = 11$.

a formula satisfying these is

$$c(n, m, k) = c(n, k) = 2k + 5n - 1$$

that looks familiar...

it seems the iteration count is independent of m, and it is the same as in a) for when $n = m = C$.

i suppose that makes sense. when the nodes are close to the ground, they dont have to be relabeled that much. but to be independent of m? that is kind of weird.

my algorithm implementation might be wrong, of course. however i don't feel like looking any more at this now. ciao!

Task 2

Next page is Readable

Let $G=(V,E)$ be a directed, weighted graph with non-negative weights

Let $V=U \cup W$ for some partition $\{U,W\}$. Furthermore,

$\sum_{u \in U, w \in W} w(u,w)$ is maximized.

In other words, we want to find the maximum Cut (see ex 2.)

A randomized approximation would be to assign a vertex $v \in U \cup W$ based on a coin flip, i.e. Heads $\rightarrow v \in U$, Tails $\rightarrow v \in W$.

This yields an approximation guarantee of $\frac{1}{4}$.

Let's see what $\Pr[v \in U] = \Pr[v \in W] = 50\%$ exactly is.

Take two vertices $u, v \in V$. Then

$$\Pr[u \in U \wedge v \in W] = \Pr[u \in U] \cdot \Pr[v \in W]$$
$$= \frac{1}{2} \cdot \frac{1}{2} = \frac{1}{4}.$$

This is the probability that these vertices are in the cut. Let

$$X_{uv} = \begin{cases} 1 & \text{if } u \in U \wedge v \in W \\ 0 & \text{otherwise} \end{cases}$$

Then

$$E[X_{uv}] = \Pr[X_{uv}] \cdot 1 + \Pr[\neg X_{uv}] \cdot 0 = \frac{1}{4}$$

And

$$\text{ALG} = E\left[\sum_{uv \in E} w_{uv} X_{uv}\right] = \sum_{uv \in E} w_{uv} E[X_{uv}] = \frac{1}{4} \sum_{uv \in E} w_{uv}$$

But

$$\text{OPT} \leq \sum_{uv \in E} w_{uv}$$

So

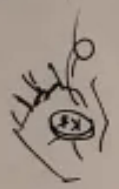
$$\text{ALG} = \frac{1}{4} \sum_{uv \in E} w_{uv}$$
$$\geq \frac{1}{4} \text{OPT}$$
$$\Rightarrow \frac{\text{ALG}}{\text{OPT}} \geq \frac{1}{4}$$

Comment:
I should specify the sums and fix the wrong indg. above. But good enough...

Task 2

Let $G=(V,E)$ be a directed, weighted graph with non-negative weights. Let $V = U \cup W$ for some partition $\{U, W\}$. Furthermore,

$$\sum_{\substack{u \in U \\ v \in W}} w(u,w)$$ is maximized.



In other words, we want to find the Maximum Cut (see ex 2.)

A randomized approximation would be to assign a vertex $v \in V$ to either U or W based on a coin flip, i.e. Heads $\rightarrow v \in U$, Tails $\rightarrow v \in W$.

This yields an approximation guarantee of $1/4$.

Let's see what $Pr[v \in U] = Pr[v \in W] = 50\%$ entails.

Take two vertices $u, v \in V$. Then

$$Pr[u \in U \wedge v \in W] = Pr[u \in U] \cdot Pr[v \in W]$$
$$= 1/2 \cdot 1/2 = 1/4.$$

This is the probability that these vertices are in the cut. Let

$$X_{uv} = \begin{cases} 1 & \text{if } u \in U \wedge v \in W \\ 0 & \text{otherwise} \end{cases}$$

Then

$$E[X_{uv}] = Pr[X_{uv}] \cdot 1 + Pr[\neg X_{uv}] \cdot 0 = 1/4$$

And

$$APX = E\left[\sum_{\substack{u \in U \\ v \in W}} w_{uv}\right] = E\left[\sum w_{uv} \cdot X_{uv}\right]$$

$$= \sum w_{uv} E[X_{uv}] = 1/4 \sum w_{uv}$$

But

$$OPT \geq \sum w_{uv}$$

so

$$APX = 1/4 \sum w_{uv} \leq 1/4\, OPT$$

$$\Rightarrow \frac{APX}{OPT} \leq 1/4$$

Comment:
I should specify the sums and fix the wrong ineq. above, but good enough...

task 3

the following article explains an algorithm that solves our problem: https://en.wikipedia.org/wiki/Reservoir_sampling#Simple:_Algorithm_R

the main idea is to keep a reservoir of k items, deciding randomly if the i-th item you revealed should be added to the reservoir (replacing any item already in there at that slot), otherwise discarding the item.

the random variable determining the inclusion of the item in the reservoir is given as a random integer between 1 and i; if it is less than or equal to k, it will fit in the reservoir and then occupies that drawn slot in the reservoir. if it is greater than k, it will be discarded.

this ensures a sampling with a uniform probability distribution of n

proof: since an item i is included in the reservoir with a probability of $\frac{k}{i}$, the next item $i+1$ has probability $\frac{k}{i+1}$ of being included in the reservoir. but since processing the $(i+1)$-th input affects the probability of the previous elements, the probability of i is now $\frac{k}{i} \times \left(1 - \frac{1}{i+1}\right) = \frac{k}{i+1}$. but then it isn't actually the case that the probability is different for the previous elements, i.e. the sampling was uniform and independent.

as the wiki mentions, the algorithm is needlessly slow, but plenty apt for a simple solution to our problem.