Midterm 1

1 Comparing asymptotics (4 points)

For each question, fill in all circles that apply.

f	g	$f = O(g)$?	$g = O(f)$?
$f(n) = 4^n$	$g(n) = 16^{\log_2(n)}$	○	●
$f(n) = (\sqrt{n} + n)(30\sqrt{n})$	$g(n) = n^2$	●	○
$f(n) = n^3$	$g(n) = n^{\log_3(26)}$	○	●
$f(n) = n^{0.001}$	$g(n) = \log_2 n$	○	●

- $g(n) = 16^{\log_2 n} = 2^{4 \log_2 n} = 2^{\log_2 n^4} = n^4$, so $g(n) = O(f(n))$ as exponential functions grow more than polynomial functions.
- $f(n) = O(n^{1.5})$, so $f(n) = O(g(n))$ by polynomial powers
- $\log_3(27) = 3 > \log_3(26)$, so $g = O(f)$ by polynomial powers
- Any polynomial grows more than any power of log

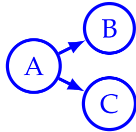
2 True or False? (5 points)

Mark your choice for each of the following. Fill in the bubble completely, as incomplete markings will not be given credit.
Grading: +1 for each correct answer, 0 for leaving blank, and −1 for each incorrect answer. Any negative score will affect your entire exam.

(a) (**1 point**) A DAG does not necessarily have a unique topological ordering.

● True ○ False

Consider the following DAG:



Two valid linearizations/topological orderings are A, B, C and A, C, B

(b) (**1 point**) On graphs with negative edge weights, Dijkstra does not work since it does not necessarily halt; otherwise, once it halts, it outputs the correct solution.

○ True ● False

Dijkstra will always halt on finite graphs. The reason why Dijkstra fails is it assumes that edge weights are positive.

(c) (**1 point**) If DFS on a directed graph $G = (V, E)$ produces exactly one back edge, then it is always possible to remove an edge e from the graph G such that $G' = (V, E - \{e\})$ is a DAG.

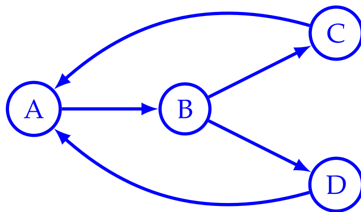
● True ○ False

DFS finds a back-edge iff the graph is cyclic. Therefore the graph with the removed back edge is acyclic, as its DFS does not have a back-edge.

(d) (**1 point**) If the directed graph G contains a cycle, and removing an edge from G can make it acyclic, then any DFS on G would produce exactly one back-edge.

○ True ● False

Consider the following graph:



The graph has two cycles, $A \to B \to C$ and $A \to B \to D$. Removing $A \to B$ makes the graph acyclic, but the DFS with lexiographic order on the vertices has two back-edges.

(e) (**1 point**) If DFS on a directed graph $G = (V, E)$ produces two back edges, there must be at least two strongly connected components which each have at least 2 vertices in the original graph

○ True ● False

The answer to the previous part is also a counterexample for this one. It is one strongly connected component.

3 Recurrences (8 points)

Write down the solutions to the following recurrence relations (only the final answer is needed). Write down the tightest bound that you can derive. You may use big-O notation.

(a) **(2 points)** $T(n) = 23T(\frac{n}{3}) + 2n^3$

$O(n^3)$

$\frac{a}{b^d} = \frac{23}{27} < 1$. Using the Master theorem, we get $O(n^3)$.

(b) **(2 points)** $T(n) = 3T\left(n^{\frac{1}{3}}\right) + 5n$, and $T(3) = 3$.

$O(n)$

Solution 1:

Consider the recursion tree for this recurrence relationship to find the total work done as $\sum_{i=0}^{\text{\# of layers}}$ work done at the i-th layer.

At the i-th layer, we have 3^i nodes, with each node doing $O(n^{\frac{1}{3^i}})$ work. Therefore, the work done at the i-th layer is on the order of $3^i n^{\frac{1}{3^i}}$.

We reach the base case after k layers when $n^{\frac{1}{3^k}} = 3$. Taking the log of both sides and rearranging the terms gives us $k = \log \log n$, which is the height of our tree.

Hence, the total work done is $\sum_{i=0}^{\log \log n} 3^i n^{\frac{1}{3^i}} = n + 3n^{\frac{1}{3}} + ... + 3^{\log \log n} n^{\frac{1}{3^{\log \log n}}}$. We can bound all terms but the first by $3^{\log \log n} n^{\frac{1}{3}}$ since $3^{\log \log n}$ is the largest value 3^i attains and $n^{\frac{1}{3}}$ is the largest value $n^{\frac{1}{3^i}}$ attains.

Therefore, the summation of all but the first term can be bounded by $3^{\log \log n} n^{\frac{1}{3}} \log \log n$, which is $O(n)$. Since the first term in our summation is n and the remaining terms can be bound by $O(n)$, the runtime is $O(n)$.

Solution 2:

We know that the recurrence $S(n) = 3S(\frac{n}{4}) + 5n$ is $O(n)$, so $T(n) = O(S(n)) = O(n)$.

(This is a tight bound as $T(n) = \Omega(n)$ since $T(n)$ does at least $5n^2$ work)

(c) **(2 points)** $T(n) = 8T(n - 3) + 1$, and $T(0) = T(1) = T(2) = 1$.

$O(2^n)$

We have 8^i nodes at layer i which do constant work and there are $n/3$ layers. The work done in the last layer is $8^{n/3-1}$. So in total we have $\sum_{j=1}^{n/3-2} 8^j = \frac{8^{n/3}-1}{8-1} = O(8^{n/3}) = O(2^n)$

(d) **(2 points)** $T(n) = T(n/5) + T(4n/5) + 3n^2$

$O(n^2)$

We proceed using a method we learned from *Median of Medians* from Homework 2, Question 4. We "guess" that $T(n) = O(n^2)$ from the $3n^2$ term; with this we say that $T(n) \leq c \times n^2$ for some $c > 0$. Plugging this into the recurrence relation:

$$T(n) \leq c\left(\frac{n}{5}\right)^2 + c\left(\frac{4n}{5}\right)^2 + 3n^2$$
$$\leq \left(\frac{17c}{25} + 3\right) n^2$$

Recall we want $T(n) \leq c \cdot n^2$, plugging in the above this means

$$\frac{17c}{25} + 3 \leq c$$

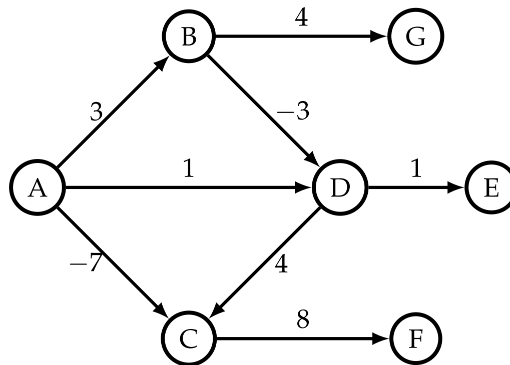
Which is valid for $c \geq \frac{75}{8}$. Thus we see that $T \leq \frac{75}{8}n^2$, and $T(n) = O(n^2)$. Notice we cannot get a tighter bound than this since the recurrence relation itself has a $3n^2$ term.

4 Dijkstra's (6 points)

Execute Dijkstra's algorithm on the following graph starting at vertex A and breaking ties alphabetically.
Here is the algorithm for reference. Assume that decreasekey does nothing if the vertex is not in the heap.

Algorithm 1 Dijkstras(G, l, s)

```
for all u ∈ V do
   dist(u) = ∞
   prev(u) = null
end for
dist(s) = 0
H = makequeue(V) (using dist-values as keys)
while H is not empty do
   u = deletemin(H)
   for all edges (u,v) ∈ E do
      if dist(v) > dist(u) + l(u,v) then
         dist(v) = dist(u) + l(u,v)
         prev(v) = u
         decreasekey(H, v)
      end if
   end for
end while
return dist
```



Fill in the following table with the shortest paths computed by Dijkstra's algorithm:

A	B	C	D	E	F	G
0	3	-7	0	2	1	7

5 Factorials (10 points)

In this question, assume you can multiply two d-bit integers in time $O(d^{1.59})$. Given an integer n, design an algorithm to compute $n!$ that runs in time $O(n^{1.59}\log^c n)$ for some constant $c > 0$. *Hint: divide and conquer.*

(a) Describe your algorithm **succinctly** below.

Solution. Let $P(k,k') = \prod_{i=k}^{k'} i$ be the product of integers $\{k, k+1, \cdots, k'\}$. Recursively compute $x_1 = P(1, \lceil n/2 \rceil)$ and $x_2 = P(\lceil n/2 \rceil + 1, n)$ and output $x_1 \cdot x_2$.

(b) Provide a rigorous analysis for the runtime of your algorithm. Recall you showed on homework that $\log(n!) = \Theta(n \log n)$.

Solution. Let $T(k)$ be the runtime of computing the product of k integers, each of at most $\log n$ bits. Computing $x_1 \cdot x_2$ takes $O\left((k\log n)^{1.59}\right)$ time, since x_1, x_2 are of at most $k \log n$ bits. Then the runtime of the algorithm is given by

$$T(k) = 2T(k/2) + (k\log n)^{1.59}.$$

This leads to the bound

$$T(k) = O\left(k^{1.59}\log k \cdot \log^{1.59} n\right).$$

It follows that $T(n) = O\left(n^{1.59}\log^{2.59} n\right)$. You can see this by looking at the recurrence tree. Notice that there are $\log k$ layers, and the i-th layer has $2^i\left(\frac{k}{2^i}\log n\right)^{1.59} \leq (k\log(n))^{1.59}$ work. There are $\log k$ layers, the total amount of work is at most

$$\log k \cdot (k \log n)^{1.59} = k^{1.59}\log k \log^{1.59} n.$$

One can improve the bound by a more careful calculation:

$$\begin{aligned} T(k) &= \sum_{i=1}^{\log k} 2^i \left(\frac{k}{2^i}\log n\right)^{1.59} \\ &\leq \left(k^{1.59}\log^{1.59} n\right) \cdot \sum_{i=1}^{\infty} 2^i \frac{1}{2^{1.59i}} \\ &\leq \left(k^{1.59}\log^{1.59} n\right) \cdot \sum_{i=1}^{\infty} \left(\frac{2}{3}\right)^i \\ &= O\left(k^{1.59}\log^{1.59} n\right). \end{aligned}$$

6 Fast Force Computation (10 points)

5 charged particles are placed on a line. Particle i is placed at point i on the x-axis with charge c_i. The force on particle j in the system is

$$f_j = \sum_{i<j} \frac{c_i}{(i-j)^2} - \sum_{i>j} \frac{c_i}{(i-j)^2}.$$

(a) Set up two polynomials $p(x)$ and $q(x)$ so that $f_1, \ldots, f_5$ appear as coefficients of $p(x) \cdot q(x)$. (It is okay if $p(x) \cdot q(x)$ have other *irrelevant* coefficients as well.) Your answer should depend on $c_1, \ldots, c_5$.

Solution.

$$p(x) = c_1 x^4 + c_2 x^3 + c_3 x^2 + c_4 x + c_5$$
$$q(x) = -\frac{1}{16}x^8 - \frac{1}{9}x^7 - \frac{1}{4}x^6 - x^5 + x^3 + \frac{1}{4}x^2 + \frac{1}{9}x + \frac{1}{16}$$

The idea of reversing one of the coefficients/taking successive dot products is very similar to HW3 Q6. One could start with the equation for the coefficient r_k, and manipulate it to get the sums that compute f_j.

$$f_j = \sum_{i<j} \frac{c_i}{(i-j)^2} - \sum_{i>j} \frac{c_i}{(i-j)^2}$$

This can be understood as the computation of successive dot products between the vectors

$$\begin{bmatrix} c_1 & c_2 & c_3 & c_4 & c_5 \end{bmatrix}, \begin{bmatrix} \frac{1}{16} & \frac{1}{9} & \frac{1}{4} & 1 & 0 & -1 & -\frac{1}{4} & -\frac{1}{9} & -\frac{1}{16} \end{bmatrix}$$

We can implement the successive dot product by polynomial multiplication and flipping the first vector. If $r = p \cdot q = r_0 + r_1 x + \ldots + r_{12} x^{12}$, then $r_4 = \frac{c_1}{16} + \frac{c_2}{9} + \frac{c_3}{4} + c_4 = f_5$, $r_3 = f_4$, etc.

(b) What is the index of the coefficient of $p(x) \cdot q(x)$ corresponding to f_3? (e.g. if it is the coefficient of x^2, write 2.)

Solution 1.

6

Points were only awarded if part (a) was correct.

7 Tree matches (20 points)

Let T be an unweighted and undirected tree on vertex set $V = \{1, \ldots, 2n\}$. $E(T)$ is the set of T's edges, and $V(T)$ is the set of its vertices. We call a collection of n pairs $P = \{(u_1, v_1), \ldots, (u_n, v_n)\}$ a *valid pairing* if for $i = 1, \ldots, n$, $u_i \neq v_i$ and each vertex $v \in V$ occurs in exactly one of the n pairs. We define the *cost* of a valid pairing P as

$$f_T(P) := \sum_{i=1}^{n} d_T(u_i, v_i)$$

where $d_T(a, b)$ is the length of the shortest path between a and b in T. In this question we will see an $O(n)$ time algorithm to compute the minimum cost of a valid pairing; in particular, to compute

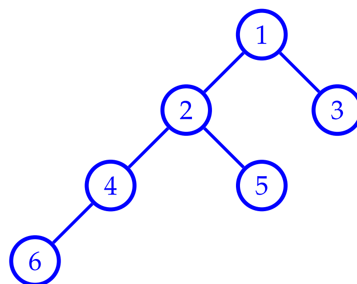
$$\mathsf{OPT}(T) := \min_{P \text{ valid pairing}} f_T(P).$$

We emphasize that $\mathsf{OPT}(T)$ is a *number* — namely the minimum value attained by f (We don't ask you to output the a pairing with minimum cost.)

(a) **(5 points)** Let $P^* = \{(u_1, v_1), \ldots, (u_n, v_n)\}$ be a valid pairing such that $f_T(P^*) = \mathsf{OPT}(T)$ and let p_t denote the unique path between vertices u_t and v_t in T. Prove that for any $i, j \in \{1, \ldots, n\}$ such that $i \neq j$, p_i and p_j are *edge-disjoint*. We say two paths p_i and p_j are *edge-disjoint* if the collection of edges used by p_i does not intersect the collection of edges used by p_j. You are encouraged to illustrate your proof idea in a diagram.
Hint: What if for some $i < j$, p_i and p_j shared an edge e? Can you make a small change to P^* to obtain a new pairing P' with $f(P') < f(P^*)$?

Solution:
Notice how $f_T(P)$ is the sum of the shortest paths of all pairs in P. The high level idea is ideally we don't want to 'double count' an edge, as we are trying to minimize the summation.

Let's consider the following diagram.



Let's consider the case where the pairs are: {(1, 3), (2, 6), (4, 5)}. In this pairing the edge (2, 4) is used by the shortest path from 2 to 6 and from 4 to 5. Instead it would be better to have the pairs, (2, 5) and (4, 6) (which are edge-disjoint), as the shared edges would no longer be used.

Formally said, suppose p_i and p_j intersect, then replacing p_i and p_j with the two paths induced by $p_i \Delta p_j$ strictly decreases the cost. Since P^* is optimal, its cost cannot be decreased

and hence all p_i, p_j pairs must be edge-disjoint.

(b) (**5 points**) Let P^* and p_t be defined as in part (a) and let $L \subseteq E(T)$ be a subset of edges defined as follows:

$$L := \{e : \exists p_t \text{ such that } e \in p_t\}.$$

Deleting an edge $e \in E(T)$ splits T into two trees $T_{e,1}$ and $T_{e,2}$. Prove that $e \in L$ *if and only if* $|V(T_{e,1})|$ and $|V(T_{e,2})|$ are both odd numbers.
Try to use the result of part (a) to show that if $|V(T_{e,1})|$ and $|V(T_{e,2})|$ are both even, then $e \notin L$. You will additionally need to argue that if $|V(T_{e,1})|$ and $|V(T_{e,2})|$ are both odd, then $e \in L$.

Solution.
The first thing to notice is that not all edges in E are used in L, and that all vertices appear exactly once in the n pairs. In addition, because T is a tree, there must be exactly one path between every pair of vertices that doesn't repeat vertices. The question asks to prove an if and only if, so we need to prove both directions.
Proof that oddness implies $e \in L$:
If $|V(T_{e,1})|$ and $|V(T_{e,2})|$ are both odd numbers, then any valid pairing must contain a pair (u, v) such that $u \in V(T_{e,1})$ and $v \in V(T_{e,2})$. Since any path between $T_{e,1}$ and $T_{e,2}$ must pass through e (as T is a tree), it must be in L.

Proof that $e \in L$ implies oddness:
We will prove the contrapositive, i.e., if $|V(T_{e,1})|$ and $|V(T_{e,2})|$ are both even, $e \notin L$. e must appear in every path between $T_{e,1}$ and $T_{e,2}$, and by part (a), it can appear at most in one path. On the other hand the number of paths between $T_{e,1}$ and $T_{e,2}$ must be an even number and hence e must appear in 0 paths, which means $e \notin L$.

(c) (**10 points**) Use the result of part (b) to devise an algorithm to compute $\mathsf{OPT}(T)$. Full points will be given for an $O(n)$-time algorithm.
Hint: Observe that $\mathsf{OPT}(T) = f(P^*) = |L|$ *where L is defined in part (b).*

Solution.
We want to construct an algorithm that finds all the odd sized subtrees, so that we can utilize part b, to find the edges that connect the odd sized subtrees. We also know the number of vertices is even (specifically $2n$), and so if a subtree has an odd number of vertices, we know that the set of all other vertices must also be odd because odd + odd must equal even.
The algorithm is as follows: Pick an arbitrary vertex r and run DFS starting at r augmented with a counter that for each visited vertex v, keeping track of the size of the subtree at v (mod 2). This counter will be used to determine whether the size is even or odd. For each edge e in T, one of its endpoints u can be identified as a 'child' and the other endpoint v as a 'parent' in the DFS tree. If the size of the subtree rooted at u, the child is 1 mod 2 (odd), deleting the edge splits the tree into two odd-sized components, and so e is in $|L|$. Otherwise, deleting the edge splits the tree into two even-sized components, and e is not in $|L|$ [a]

[a] Footnote: if you store the size of the subtree rather than the size mod 2, each arithmetic operation would take $O(\log n)$ time, and the algorithm would be $O(n \log n)$ time. Points were not docked for this solution.

8 Trickster negative cycles (22 points)

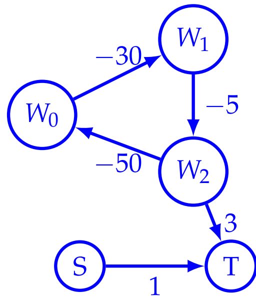
(a) (**4 points**) Draw a weighted directed graph G with the following properties:

(i) it contains a *negative-weight cycle*,

(ii) it has two vertices S and T such that the length of the shortest path between S and T is *positive*.

Make sure to explicitly write the weight of each directed edge in the graph you draw! The distance between S and T in your graph should be positive.

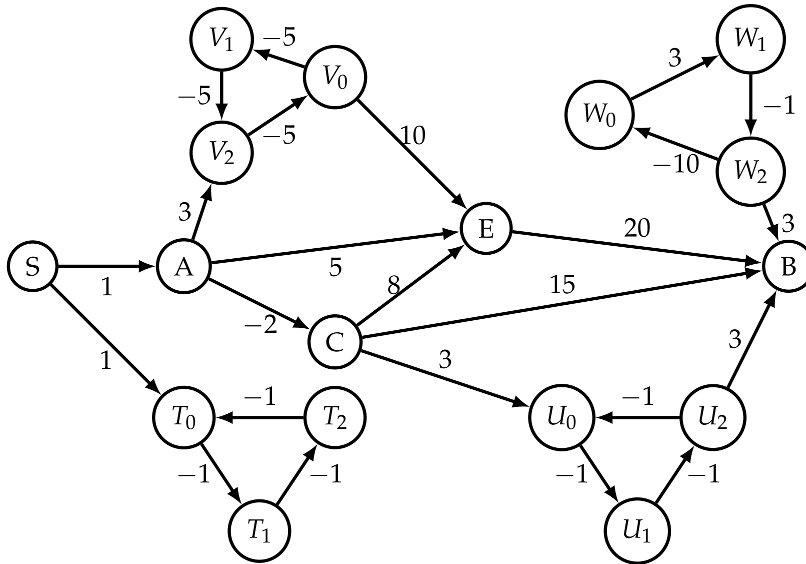
Solution.



The negative cycle in this picture does not affect the distance from S to T.

You could also just not connect the cycle to the graph, and this would accomplish the same.

(b) (**8 points**) Consider the following graph:



The graph has exactly four negative cycles: $T_0T_1T_2$, (which we will call cycle T), $U_0U_1U_2$ (which we will call cycle U), $V_0V_1V_2$ (which we will call cycle V), $W_0W_1W_2$ (which we will call cycle W).

We say that the distance path between nodes X and Y is *not well-defined* if for any path from X to Y, you can find a strictly shorter path (e.g. there's a path of length 10 from X to Y, one of length 5 from X to Y, one of length 0, one of length -5, etc)

Notice how if there is a negative cycle on the path from the X to the Y, then the distance path is **not** well defined.

(i) Consider what happens if all negative cycles except one are removed from the graph. You would like the distance from S to **B** to be well-defined.
Mark all of the negative cycles for which keeping only that negative cycle results in the distance from S to B being well-defined.

● T ○ U ○ V ● W

We want to choose all cycles which are **not** on a path from S to B. Notice how there is a path from S to B, going through V and a different one going through U, but no path going T or W. Thus we want T and W because there **is** a well defined path if these cycles are the only ones in the graph.

(ii) Consider what happens if all negative cycles except one are removed from the graph. You would like the distance from S to **E** to be well-defined.
Mark all of the negative cycles for which keeping only that negative cycle results in the distance from S to E being well-defined.

● T ● U ○ V ● W

We want to choose all cycles which are **not** on a path from S to E. Notice how there is a path from S to E, going through V, but no path going through T, U or W. Thus we want T, U and W because there **is** a well defined path if these cycles are the only ones in the graph.

(iii) Consider what happens if all negative cycles except one are removed from the graph. You are interested if Bellman-Ford starting from S then reports that distances are not well-defined.

Mark all of the negative cycles for which keeping only that negative cycle results in Bellman-Ford starting from S reporting that distances are not well-defined.

● T ● U ● V ○ W

We want to choose cycles are on the path from S to any other vertex that is remaining in the graph. Essentially what this ends up being is the cycles that are reachable from s. The only cycle **not** reachable from S is W. So, the answer is T, U, and V.

(iv) Now consider what happens if **all** negative cycles are removed from the graph, and you run Bellman-Ford starting from S. True or false: The shortest-paths tree produced by Bellman-Ford in this case is independent of the order in which it updates the edges.

● True ○ False

The shortest path tree is unique (considering all negative cycles are gone), and since Bellman Ford computes the shortest path tree when there are no negative cycles, it must return the same answer.

The main insight from the previous problem is the shortest path can still be well-defined between some pairs of vertices despite the existence of negative cycles.

(c) (**10 points**) Give an algorithm that takes in a weighted directed graph $G = (V, E)$ (potentially with negative weight cycles) along with two vertices s and t as input, and outputs the length of the shortest path between s and t if it is well-defined and the string "`no well-defined shortest path`" otherwise.

Any algorithm that correctly solves this problem and runs in time polynomial in $|V|$ and $|E|$ will receive full credit. You may freely use algorithms covered in lecture in a black-box fashion. *Hint: modify the graph and feed the modified graph to the Bellman–Ford algorithm.*

(i) Give a description of your algorithm.

Solution 1: Notice in this problem we specifically only care about the length of the path from s to t. We also know that Bellman Ford can detect negative cycles, but due to the fact mentioned in the previous statement, we only care about negative cycles that could potentially be on the path from s to t. Negative cycles can be in three places in relation to a path of s and t:

i. On the path (i.e. there exists a path between s through the cycle to t)
ii. The negative cycle is reachable from s but can't reach t
iii. The negative cycle can reach t but s can't reach it

We need to check for each case if this cycle is relevant to the length of the path from s to t.

i. If it is on the path, then we want to keep the cycle, because it causes the problem to have a not well-defined shortest path from s to t.
ii. If the negative cycle can't reach t but is reachable from s, then we don't want this cycle because it is not relevant to us. Specifically, Bellman Ford would assign vertices in this cycle a distance because s can reach it.
iii. If the negative cycle can reach t but s can't reach it, then Bellman Ford will never update the distances of the vertices in the cycle to be not ∞, as the distances computed in Bellman Ford is from s to the vertices, so these cycles don't affect the correctness of the algorithm.

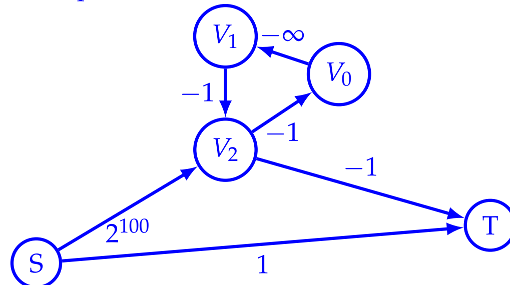
We want to remove vertices that cannot reach t. To do this, we reverse the graph and run explore starting on t and remove all vertices that are not visited by explore. Then, we can run Bellman Ford, and if there is a negative cycle in the graph, then there is no well-defined shortest path from s to t, and if there is not a negative cycle, then we return the length that Bellman Ford returns for the distance from s to t.

Solution 2: From the previous section "Common Misconceptions", notice how the main problem is that the negative cycle might take many iterations before Bellman Ford updates t's distance correctly. To fix this, the idea of this solution is we want to change all negative cycles to be extremely negative (in fact $-\infty$ big), so that we finish in poly $|V|$ and $|E|$, as Bellman Ford will definitely favor a path with a negative cycle if it can.

To do this, we first compute the SCCs of the graph using the SCC algorithm. Then, for every SCC, run Bellman Ford choosing an arbitrary vertex in the SCC to be s to determine if there is a negative cycle in the SCC. If there is, make one edge in the SCC $-\infty$ in a modified graph G' (which starts initially as a copy of the **original** graph).

Then run Bellman Ford on G'. If t becomes $-\infty$, then we know that it must have gone through a negative cycle, otherwise, return the length from s to t returned by this Bellman Ford.

Example:



Common Misconception

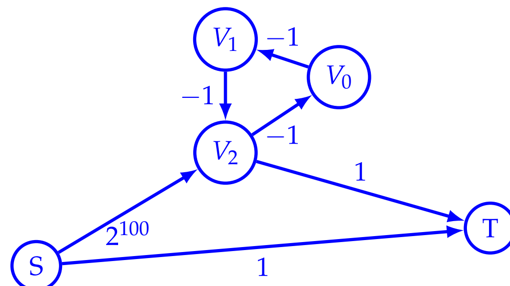
A incorrect solution was the following:

Let a be the distance from s to t after running Bellman-Ford for $|V|$ iterations.
Let b be the distance from s to t after running Bellman-Ford for $2|V|$ iterations.

If $a = b$, output as the distance. Otherwise, output "not well defined".

This solution would produce an incorrect output for graphs where the shortest path does not go through the negative cycle, so running Bellman Ford a second time would not result in a different distance for t. In order to guarantee correctness, you have to run this graph for the length of the maximum possible negative path which could be on the order of $\frac{\sum_{e \in \text{edges}} |e|}{|\text{weight of smallest negative cycle}|}$ iterations (longest possible path divided by length of smallest negative cycle). This is **not** poly in $|V|$ or $|E|$, so any solution that tried to do run Bellman Ford for multiple iterations without modifying the graph received no points.

Example:



Note that you need to run $\frac{2^{100}+1}{3}$ iterations of Bellman Ford in order for the distance to t to be updated.

(ii) Give a run-time analysis of your algorithm.

Solution 1:
Reversing the graph takes $O(|E|)$ time. explore takes $O(|V| + |E|)$ time. Creating the modified graph takes $O(|E|)$ time. Bellman-Ford on the modified graph takes $O(|V| \cdot |E|)$ time. The runtime of the entire algorithm is $O(|V| \cdot |E|)$.
Solution 2:
We run Bellman Ford on all SCCs. The runtime is v_i is the number of vertices in SCC i, and e_i is the number of edges in SCC i. The runtime of this is the $\sum_{i=1}^{\text{number of SCCs}} v_i e_i \leq \sum_{i=1}^{\text{number of SCCs}} v_i |E| \leq |E||V|$ Running Bellman Ford on the modified graph is $|V||E|$, so the overall runtime is $|V||E|$.