CS 161 **Problem Set 7**
Fall 2025 **Due:** Friday December 5, at 11:59pm on Gradescope.

Style guide and expectations: Please see the top of the "Homework" page on the course webpage for guidance on what we are looking for in homework solutions. We will grade according to these standards.

Make sure to look at the "**We are expecting**" blocks below each problem to see what we will be grading for in each problem!

Collaboration policy: You may do the HW in groups of size up to three. Please submit one HW for your whole group on Gradescope. (Note that there is an option to submit as a group). See the "Policies" section of the course website for more on the collaboration policy.

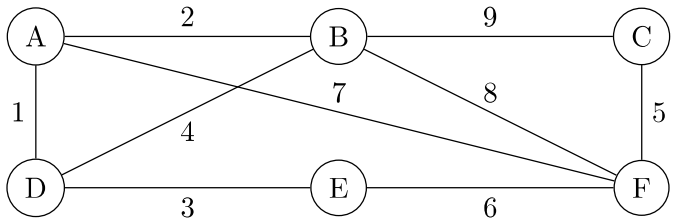
LLM policy: Check out the course webpage for best practices on how to productively use LLMs on homework, if you use them at all.

Exercises

We recommend you do the exercises on your own before collaborating with your group.

---

1. **(2 pt.)** Consider the graph G below.



   (a) **(1 pt.)** In what order does Prim's algorithm add edges to an MST when started from vertex C?

   (b) **(1 pt.)** In what order does Kruskal's algorithm add edges to an MST?

   [**We are expecting:** *For both, just a list of edges. You do not need to draw the MST, and no justification is required.*]

SOLUTION:

(a) Prim's algorithm adds edges in the order:
{C,F}, {F, E}, {E, D}, {A, D}, {A, B}

(b) Kruskal's algorithm returns the same tree, and adds edges in the order:
{A,D}, {A, B}, {D, E}, {F, C}, {E,F}

2. **(6 pt.)** In this exercise, we will walk through a greedy algorithm for the following problem. Suppose that a troupe of penguins is hiking along a scenic antarctic hiking trail of length T meters. There are n scenic vistas along the way, located at distances $x_1 < x_2 < \cdots < x_n$ meters. The troupe wants to stop at as many scenic vistas as possible, but wants to space them out so that there are at least k meters between any two stops.

   (a) **(3 pt.)** Design a greedy algorithm that achieves the goals outlined above. Your algorithm should take as input k and the values $x_1, \ldots, x_n$ (in sorted order). It should output a list of stops $i_1 < i_2 < \ldots < i_m$ so that m is as large as possible, subject to the constraint that $x_{i_j} - x_{i_{j-1}} \geq k$ for all $j = 1, \ldots, m$. Your algorithm should take time $O(n)$.

   [**We are expecting:** *A clear explanation of your greedy choices, and a clear English description of your algorithm. You don't need to justify the running time. (You'll prove correctness in part (b)).*]

   SOLUTION: We'll always take the next available stop. That is, we always take x_1. Then we take the next stop that's at least k meters away from x_1, and so on.

   Pseudocode isn't required, but here it is for clarity:

```
# X is an array with x_1, ..., x_n in it
def findScenicVistas(k, X):
     vistas = [ X[0] ]
     for x in X[1:]: # iterate through the remaining stops
           if x - vistas[-1] < k:
                continue
           else:
                vistas.append(x)
     return vistas
```

   (b) **(3 pt.)** In this part, you will prove that your algorithm from part (a) is correct. Following the examples that we saw in class, the structure of the proof is shown below: we have filled in the inductive hypothesis, base case, and conclusion. Fill in the inductive step to complete the proof.

   - **Inductive hypothesis:** After making the t'th greedy choice, there is an optimal way to pick vistas that extends the partial solution the algorithm has constructed so far.
   - **Base case:** Any optimal solution extends the empty solution, so the inductive hypothesis holds for $t = 0$.
   - **Inductive step:** *(you fill in)*
   - **Conclusion:** At the end of the algorithm, the algorithm returns a list S^* of scenic vistas that are all at least k meters apart. By the inductive hypothesis, we know there is an optimal solution extending S^*. Yet, there is no solution extending S^* other than S^* itself, since there are no more vistas that could be added to the end without making two vistas closer than k meters. This implies that S^* is optimal, and hence the algorithm returns an optimal solution.

[**We are expecting:** *A proof of the inductive step: assuming the inductive hypothesis holds for $t-1$, prove that it holds for t.*]

SOLUTION: Suppose that the inductive hypothesis holds for $t-1$, so there is some optimal solution S that extends the choices $x_{i_1}, \ldots, x_{i_{t-1}}$ that have been made so far. Suppose that the algorithm chooses i_t as its t'th choice. If $S[t] = x_{i_t}$, then we are done. So suppose that $S[t] = x_j$, for $j \neq i_t$. Then $j > i_t$, since if $j < i_t$ we would have $x_j - x_{i_{t-1}} < k$, violating the assumption that S was a valid solution. Let S^* be the same as S, except with i_t swapped for j.

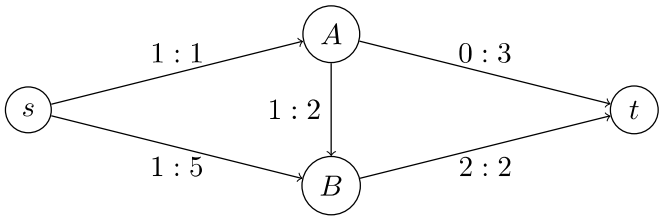
We claim that S^* is still a valid solution. Indeed, by our choice of i_t, we have $x_{i_t} - x_{i_{t-1}} \geq k$. Let $S[t+1] = x_\ell$ be the *next* stop that S would make. Then

$$x_\ell - x_{i_t} \geq x_\ell - x_j \geq k,$$

where the first inequality is because $j > i_t$ and hence $x_{i_t} < x_j$; and the second is because we are assuming that S is a valid solution, so we must have $S[t+1] - S[t] \geq k$. Thus, in the new solution S^*, $S^*[t] = x_{i_t}$ is at least k away from $S^*[t-1]$ and $S^*[t+1]$. Since all other entries of S^* are the same as S, it follows that S^* is a valid solution.

Moreover, S^* has the same number of stops in it as S, so if S was optimal, then S^* was optimal. So S^* is an optimal valid solution that extends the choices $x_{i_1}, \ldots, x_{i_t}$ that our algorithm has made after the t'th choice. This establishes the inductive hypothesis for t.

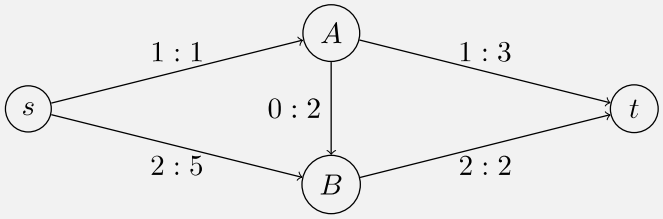
3. **(3 pt.)** *Note: this exercise covers material from Lecture 16, which we will not get to until after the break.* Consider the following graph. The notation $x : y$ means that the edge has flow x and capacity y.



Following the Ford-Fulkerson algorithm, find an augmenting path in this graph, and update the flow accordingly. What is the flow after your update?

[**We are expecting:** *A description of you augmenting path (of the form $\_\_ \to \_\_ \to \_\_ \to \_\_$), and either a picture or a list of the flows after the update. Also, we are expecting the value of the flow after the update.*]

SOLUTION: Our augmenting path is $s \to B \to A \to t$. The updated flows look like:



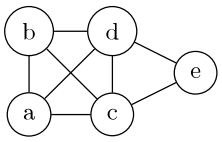
The new flow has value 3.

Problems

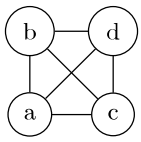
4. **(6 pt.)** [***k*-well-connected graphs.**] Let $G = (V, E)$ be an undirected, unweighted graph with n vertices and m edges. For a subset $S \subseteq V$, define the **subgraph induced by** S to be the graph $G' = (S, E')$, where $E' \subseteq E$, and an edge $\{u, v\} \in E$ is included in E' if and only if $u \in S$ and $v \in S$.

   For any $k < n$, say that a graph G is k-well-connected if every vertex has degree at least k.

   For example, in the graph G below, the subgraph G' induced by $S = \{a, b, c, d\}$ is shown on the right. G' is 3-well-connected, since every vertex in G' has degree at least 3. However, G is not 3-well-connected since vertex E has degree 2.



$G = (V, E)$



$G' = (S, E')$, for $S = \{a, b, c, d\}$

   Design a greedy algorithm to find a maximal set $S \subseteq V$ so that the subgraph $G' = (S, E')$ induced by S is k-well-connected. In the example above, if $k = 3$, your algorithm should return $\{a, b, c, d\}$, and if $k = 4$ your algorithm should return the empty set.

   You may assume that your representation of a graph supports the following operations:

   - `degree(v)`: return the degree of a vertex in time $O(1)$
   - `remove(v)`: remove a vertex and all edges connected to that vertex from the graph, in time $O(\text{degree}(v))$.
   - `vertices(G)`: return the list of non-removed vertices in time $O(n)$.

   Your algorithm should run in time $O(n^2)$.

   You do not need to prove that your algorithm works, but you should give an informal (few sentence) justification.

   [**Hint:** *Think about greedily* ***removing*** *vertices.*]

   [**We are expecting:**

   - *Pseudocode* ***AND*** *an English description of what your algorithm is doing.*
   - *An informal justification of the running time.*
   - *An informal justification that the algorithm is correct.*

]

SOLUTION:

Our algorithm will greedily remove any vertex with degree less than k, along with all the edges attached to it:

```
myGreedyAlg(G = (V,E), k):
     while True:
         for v in V:
             if G.degree(v) < k:
                 G.remove(v)
                 break
         if we did not break inside the inner loop:
             break
     return vertices(G)
```

The running time is at most $O(n^2)$ because the while loop executes at most n times (since each time it removes a vertex and there are only n vertices), and within each iteration of the while loop, we loop over all vertices $v \in V$ for another factor of n. Each time the inner for-loop runs, we additionally remove a single vertex v, which takes time $O(deg(v))$. Since the degree of v is at most k if it is getting removed, this takes time at most $O(k)$. Thus, the inner for-loop takes time $O(n + k) = O(n)$. Altogether the running time is the $O(n \times n) = O(n^2)$.

Intuitively, the algorithm works because we are only removing vertices that are "safe" to remove. That is, no optimal solution could have a vertex of degree less than k in it, so by removing a vertex of degree less than k, we haven't ruled out success.

5. **(6 pt.) [Badger Badger Badger (part 2)]** The tunnel-digging badger from HW4 is back, along with $n - 1$ of their friends! There are n badgers in total. You'd like to get a message to all n of the badgers (perhaps you'd like to tell them your solutions to HW4!). However, each of the badgers pops up out of the ground for only one interval of time. Say that badger i is above ground in the interval $[a_i, b_i]$. Your plan is to shout your message repeatedly, at certain times $t_1, \ldots, t_m$. Any badger who is above ground when you shout your message will hear it, while any badger who is below ground will not. Assume that shouting the message is instantaneous.

   You'd like to ensure that each badger hears your message at least once (it's okay if a badger hears it multiple times), while making m as small as possible (so that you don't have to shout too much).

   Design a greedy algorithm which takes as input the list of intervals $[a_i, b_i]$ each badger is above ground and outputs a list of times $t_1, \ldots, t_m$ so that every badger hears your message at least once and m is as small as possible. Your algorithm should run in time $O(n \log(n))$. If it helps, you may assume that all the a_i, b_j are distinct.

   You do not need to prove that your algorithm is correct. However, we strongly suggest that you at least prove it to yourself—or at least work out the intuition/outline of a proof—to

make sure that your algorithm is correct! (Note: The solutions will include a proof so you can check your answer later if you choose to write down a formal proof for more practice).

[**We are expecting:** *Pseudocode and an English description of the main idea of your algorithm, as well as a short justification of the running time.*]

SOLUTION:

The high-level idea of the algorithm is as follows: First, the algorithm sorts the intervals by end time. Then, it goes through the intervals in order: if b_i is the next end time, and Badger i has not heard the message, then we broadcast the message at time b_i.

Naively, the idea above would take time $O(n^2)$, since for each endpoint processed we'd have to look through the list of times output so far and see if the current badger has heard the message. However, we can make it run in time $O(n \log(n))$ (dominated by the time to sort the intervals) by not looking through the whole list of times, but by iterating through the sorted list of intervals and just keeping track of the last time we shouted as we go. Then the while loop takes time $O(n)$ as each interval only requires constant time operations. The pseudocode is as follows.

```
def badgerBadgers(awakeTimes):
    # awakeTimes is a list [ [a_0,b_0], ..., [a_{n-1}, b_{n-1}] ]
    sort awakeTimes based on the b_i's, with the smallest first.
    shoutTimes = []
    lastShout = -infinity
    for i in {0, ... ,n-1}:
        wake, sleep = awakeTimes[i]
        if wake > lastShout:
            shoutTimes.append(sleep)
            lastShout = sleep
    return shoutTimes
```

A proof of correctness is not required, but for completeness, here it is:

To prove that the algorithm is correct, formally we proceed by induction.

Inductive Hypothesis. When we add the t'th broadcast time, there is an optimal solution that extends the current solution.

Base Case. After we've added no broadcast times, there is an optimal solution which extends the empty solution.

Inductive step. Suppose that we've added t broadcast times $t_0, \ldots, t_{t-1}$, and assume by induction that this extends to an optimal solution. We would like to show that there is an optimal solution extending $t_0, \ldots, t_t$, where t_t is the time we'll choose next.

Let $T^* = [t_0, t_1, \ldots, t_{t-1}, s_t, s_{t+1}, \ldots, s_m]$ be the optimal solution extending $t_0, \ldots, t_{t-1}$. First, we claim that $s_t \leq t_t$. To see this, suppose toward a contradiction that $s_t > t_t$. But then the badger who returned underground at t_t would never hear the message (since we chose t_t to be the end time of a badger who hasn't yet heard the message), so T^* wouldn't have been a valid solution. Thus, $s_t \leq t_t$ as claimed.

Next we argue that t_t is just a good a choice as s_t, in the sense that the schedule

$$T^{**} = [t_0, \ldots, t_{t-1}, t_t, s_{t+1}, \ldots, s_m]$$

is still an optimal solution. Since T^* and T^{**} have the same length, we only need to show that T^{**} is a legitimate solution, meaning that all badgers hear the message. Suppose that there's some badger that does not hear the message in T^{**}. But then this badger must have (a) have not already heard the message during $t_0, \ldots, t_{t-1}$, and (b) heard the message at s_t and (c) not heard the message at t_t. In particular, this badger's interval (a_i, b_i) satisfies $a_i \leq s_t \leq b_i < t_t$. However, this contradicts the choice of t_t, which was the smallest time that an un-notified badger goes underground; if $s_t \leq b_i < t_t$, then we should have chosen b_i instead of t_t in the algorithm above. Thus, every badger hears the message in T^{**}, so T^{**} is indeed an optimal schedule.

This implies that there is an optimal schedule, T^{**}, which extends $t_0, \ldots, t_t$, which is what we wanted to show.

Conclusion. At the end of the algorithm, there is an optimal solution which extends the current one. Since the current one reaches all badgers, any solution that strictly extends it (adding more broadcast points) would be sub-optimal, so the current solution must be optimal.

6. **(5 pt.) [EthiCS: Building the "Best" Emergency Network]** Suppose a region wants to build a network of emergency supply stations for wildfire response. Each town gives a list of nearby towns they would be willing to share supplies with during an emergency. We model this using an undirected weighted graph $G = (V, E)$ where:

   - Each vertex is a town.
   - There is an edge $a, b \in E$ if towns a and b agree to share supplies.
   - The weight on each edge represents the estimated cost of building and maintaining a supply route between them.

   A straightforward way to minimize total cost is to apply a greedy minimum-spanning-tree algorithm (such as Kruskal's algorithm). This algorithm finds a minimum-cost network of supply routes. However, the wildfire response team expresses concerns that the resulting plan is not in the best interests of all towns.

   Using the language developed in our Embedded Ethics lectures (idealization, abstraction, and incommensurability), explain some of the real-world issues with using this greedy algorithm to design an emergency supply network.

[**We are expecting:** *2-4 sentences identifying real-world issues with this algorithm, explicitly mentioning and applying at least two of idealization, abstraction, incommensurability.*]

SOLUTION:

their answer must cover at least two of the three terms:
The MST model **abstracts** away important real-world differences between towns such as population size, fire risk, or vulnerability that matter for emergency preparedness but are not represented in the graph.

It also **idealizes** the situation by assuming that the "cost" of a route is the only value that should guide planning, even though real emergencies involve dynamic conditions that cannot be captured by fixed edge weights.

Finally, many relevant considerations like economic burden, environmental impact, and human safety are **incommensurable**, and forcing them into a single numerical weight can distort what "best" actually means for different communities.

7. **(5 pt.) [Uniqueness of MSTs]** Recall the following statement from class:

   Lemma 1. *Let G be a connected, weighted, undirected graph with distinct edge weights. Then G has a unique minimum spanning tree.*

   We will prove the lemma together.

   (a) **(2 pt.)** For sake of contradiction, suppose that T_1 and T_2 are two distinct minimum spanning trees in G, and let $e = \{u, v\}$ be the lowest weight edge that is in exactly one of T_1 and T_2. Without loss of generality, suppose that $e \in T_2$ and $e \notin T_1$.
   Now consider adding e to T_1. Prove that this must form a cycle in T_1, and further that this cycle must contain some other edge $f = \{x, y\}$ with weight strictly larger than e.
   [**We are expecting:** *A formal proof. Formal proofs can sometimes be short, but they always must fully explain why the statement is true.*]

   (b) **(2 pt.)** Let T_1' be the following modification of T_1: We start with T_1, add e, and we remove the edge f from the previous part. Prove that T_1' is a spanning tree.
   [**We are expecting:** *A formal proof. Formal proofs can sometimes be short, but they always must fully explain why the statement is true.*]

   (c) **(1 pt.)** Derive a contradiction and conclude that T_1 and T_2 couldn't have been distinct MSTs.
   [**We are expecting:** *A formal proof. Formal proofs can sometimes be short, but they always must fully explain why the statement is true.*]

SOLUTION:

(a) This forms a cycle in T_1, because there was already a path from u to v in T_1 (since it is a spanning tree), and we have completed that path into a cycle by adding $\{u, v\}$. We

must now show there will be an edge $\{x, y\}$ in the cycle with strictly larger weight than e. First, we observe that there is at least one edge $\{x, y\}$ in this cycle that is in T_1, but not T_2. This must hold, as otherwise T_2 would have contained the whole cycle, which is impossible for a tree. Since $\{u, v\}$ was the smallest weight edge that is in exactly one of T_1 and T_2, then the weight of $\{x, y\}$ must be at least as large, and since all edge weights are distinct, it must be strictly larger. This completes our proof.

(b) This follows from the logic on a reference slide in class. First, T_1' spans because it touches all the same vertices as T_1 did, and T_1 was a spanning tree. Next, to show T_1' is a tree, we will use the fact that a graph on n vertices is a tree if and only if it is connected and contains $n - 1$ edges. We note that T_1' is connected, since deleting $\{x, y\}$ formed two connected components, one with x and one with y; once we added back $\{u, v\}$, these components become connected again, because there is a path from x to y by taking the remaining part of the cycle excluding $\{x, y\}$. Finally, T_1' contains $n - 1$ edges, since T_1 did and T_1' has the same number of edges. So T_1' must be a tree. We already said that it spans, so it's a spanning tree.

(c) Recall that T_1' is a spanning tree. Moreover, T_1' was formed by starting with T_1, adding the edge $\{u, v\}$, and removing an edge with weight strictly larger than $\{u, v\}$. Hence, the weight of T_1' is smaller than T_1. This causes a contradiction because T_1 cannot have been an MST.

8. **(0 pt.)[OPTIONAL: A Band of Busy Beavers]** *Note: This problem relies on Lecture 16, which we will not get to until December 2, the same week HW is due. For that reason, we've made it optional (we realize you probably already have lots of stuff to do in week 10). But we think it's great practice and would encourage you to do it, either during Week 10 or when studying for the exam. This material is fair game for the exam.*

A band of n busy beavers has a n dams to build at various points along the river. They have n bundles of sticks at their disposal to build the dams with. However, there are a number of constraints:

- Each dam requires exactly one beaver and exactly one bundle of sticks to complete.
- Each beaver can only work on one dam (due to time constraints).
- Each bundle of sticks can only be used once.
- Each beaver can only work on particular dams (e.g., the ones near where they happen to be right now). For each beaver i and dam j, let $c_{i,j}$ be 1 if beaver i can work on dam j, and zero otherwise.
- Each bundle of sticks can only be used on particular dams (the ones that happen to be near where the sticks are right now). For bundle k and dam j, let $d_{k,j}$ be 1 if bundle k can be used on dam j, and zero otherwise.

Given these inputs (that is, $c_{i,j}$ for $i = 1, \ldots, n$ and $j = 1, \ldots, n$; and $d_{k,j}$ for $k = 1, \ldots, n$ and $j = 1, \ldots, n$), design an algorithm to assign beavers and stick-bundles to dams in a way that maximizes the number of dams that can be built.

Your algorithm should run in time $O(n^5)$.

[**We are expecting:** *Nothing, this problem is optional. But if we were to ask for something, it would be: A very clear English description of your algorithm, an informal justification of why it is correct, and an informal justification of the running time. Pseudocode is not required, but you may include it if you think it will make your answer clearer. You can (and hint, probably should), use any algorithm we have seen in class as a black box.*]

SOLUTION: We will set up a directed graph so that the maximum flow corresponds to the allocation we want; and then run Ford-Fulkerson.

Our graph will be as follows:

Vertices:

- Special vertices s and t
- For each $i = 1, \ldots, n$, a vertex b_i corresponding to the i'th beaver.
- For each $k = 1, \ldots, n$, a vertex a_k corresponding to the k'th bundle of sticks.
- For each $j = 1, \ldots, n$, *two* vertices d_j^{in} and d_j^{out} corresponding to the k'th dam.

Edges (all edges will have capacity 1):

- For each $i = 1, \ldots, n$, include the edge (s, b_i)
- For each $k = 1, \ldots, n$, include the edge (a_k, t)
- For each $j = 1, \ldots, n$, include the edge (d_j^{in}, d_j^{out})
- For each i, j so that $c_{i,j} = 1$, include the edge (b_i, d_j^{in})
- For each j, k so that $d_{k,j} = 1$, include the edge (d_j^{out}, a_k)

Now we run Ford-Fulkerson to get the max flow of this graph from s to t. The assignment is given by this max flow: as we said in class, since all the capacities are 1, each edge will end up with flow 0 or 1 going through it. If we send one unit of flow through (b_i, d_j^{in}), we assign beaver i to dam j. If we send one unit of flow through (d_j^{out}, a_k), we assign stick bundle k to dam j.

Explanation of correctness: First, we observe that all of our constraints are satisfied.

- The edges from s to the b_i each have capacity 1, so this enforces that no beaver works on more than one dam.
- The edges from t to the a_k each have capacity 1, so this enforces that no bundle of sticks is used more than once.
- The edges from d_k^{in} to d_k^{out} each have capacity 1, so this enforces that no dam can have more than one bundle of sticks or more than one beaver on it. Moreover, if a dam j has a beaver assigned to it, it must also have a bundle of sticks assigned to it and vice versa: otherwise either d_j^{in} or d_j^{out} would have flow 1 in and 0 out or vice versa.
- The edges (b_i, d_j^{in}) and (d_j^{out}, a_k) enforce the constraints that certain beavers/stick-bundles can only work on certain dams.

This means that any legitimate assignment corresponds to a flow in this graph. Moreover, the value of the flow is exactly the number of bundles of sticks used, which is exactly the number of dams built. So a maximum flow corresponds to an assignment that builds the maximum number of dams.

Explanation of the running time: The number of vertices in this graph is $|V| = 4n$. The number of edges is at most $3n + 2n^2 = O(n^2)$, where we have bounded the number of edges between the beaver layer and the dam layer by n^2 (the maximum possible such edges) and similarly between the dam layer and the stick layer. Ford-Fulkerson (or rather, the Edmonds-Karp version of it that uses BFS to find the augmenting paths) runs in time $O(|V||E|^2)$, which is $O(n^5)$ in this case.

In fact, one can do better: if we run FF with either BFS or DFS to find augmenting paths, the running time would be at most $O(|E|F)$, where F is the max flow. That's because the number of times we have to find an augmenting path is at most F, since the value of the flow increases each time and starts at zero. The running time to find an augmenting path with DFS (or BFS) is $O(|V| + |E|) = O(|E|)$ in this case since the graph is connected. So that gives total running time $O(|E|F) = O(n^3)$, since the max flow can't be more than n.