CSL-531: Dynamic Graph Algorithms

Mmukul Khedekar

January 28, 2026

Notes for the course *Dynamic Graph Algorithms* instructed by Dr. Shahbaz Khan at IIT Roorkee, Spring, 2026.

Contents

1 [Lecture] Jan 13, 2026

This course will be graded through the following components:

- **MTE:** 30%
- **CWS:** 30%
- **ETE:** 40%

Both the MTE and the ETE will be written examinations. The CWS component will be based on announced quizzes, tutorial session and a course project. The goal of the course is:

1. to be able to **Design** dynamic graph algorithms using standard design techniques and **prove** its correctness.
2. to be able to **Analyze** dynamic graph algorithms using standard analysis techniques and **prove** its tightness.
3. to be able to **Prove** the hardness of a given dynamic graph problem by **reducing** it to a standard hard problem.

1.1 Introduction to Dynamic Graph Algorithms

> **Definition 1.1.** Dynamic Graphs are graphs that *change* with time. A *change* in a graph refers to **graph updates**, that are *online* sequences of insertion or deletion of edges or vertices.

Therefore we can model a dynamic graph as a sequence of updates on a static graph. Formally, we can view this as

$$G_0 \xrightarrow{\Delta_1} G_1 \xrightarrow{\Delta_2} G_2 \xrightarrow{\Delta_3} \dots$$

where each Δ_i is an update to the graph. Suppose we are interested in computing some quantity $\mathcal{X}$ of the graph G for which there exists a known static graph algorithm $\mathbf{X}$. Upon each update Δ_i to G, we can trivially compute $\mathcal{X}$ for the new graph G' by updating G and running $\mathbf{X}$ on it. The goal of a **dynamic graph algorithm** is to reuse previous computations to update the quantity $\mathcal{X}$ *much faster* than a static graph algorithm.

Most graphs in real world are dynamic. The sizes of parameters for these graphs are significantly large and thus, recomputing solutions on such graphs from scratch is impractical. This motivates us to look for dynamic variants of static graph algorithms. However, we are only interested in dynamic graph algorithms that perform *better* than the best static graph algorithms that exist for the problem.

1.2 Classification of DGAs

Dynamic Graph Algorithms (DGAs) can be classified into

1. **Incremental** (only insertions)
2. **Decremental** (only deletions)
3. **Fully Dynamic** (both insertions and deletions)

It is worth mentioning that in case of weighted graphs, we might want to consider incrementing and decrementing the weights as graph updates too. However, any increment or decrement to the weight of an edge could be modelled as a deletion followed by an insertion. Therefore, we limit the definition of graph updates to

Definition 1.2. A dynamic graph $G(V, E)$ supports the following **graph updates**

- Insertion of edges
- Deletion of edges
- Insertion of vertices
- Deletion of vertices

To analyse the performance of algorithms, we need to equip ourselves with some theory on time complexity analysis.

1.3 Time Complexity Analysis

Some key notations on time complexities.

Definition 1.3. For functions f and g, we have

1. $f(n) = \mathcal{O}(g(n))$, if there exists positive real numbers M and n_0 such that,
$$|f(n)| \leq Mg(n), \qquad \forall n \geq n_0$$
2. $f(n) = o(g(n))$, if for every positive real number M, there exists n_0 such that,
$$|f(n)| \leq Mg(n), \qquad \forall n \geq n_0$$
3. $f(n) = \Omega(g(n))$, if there exists positive real numbers M and n_0 such that,
$$|f(n)| \geq Mg(n), \qquad \forall n \geq n_0$$
4. $f(n) = \omega(g(n))$, if for every positive real number M, there exists n_0 such that,
$$|f(n)| \geq Mg(n), \qquad \forall n \geq n_0$$
5. $f(n) = \Theta(g(n))$, if there exists positive real numbers M_1, M_2 and n_0 such that,
$$M_1 g(n) \leq |f(n)| \leq M_2 g(n), \qquad \forall n \geq n_0$$

A common way to establish a tight bound while analysing an algorithm is to find a worst-case example, which helps us establish a lower bound. In dynamic graphs, we commonly deal with the following.

1. **Preprocessing Time**: Initial time to preprocess the graph.
2. **Update Time**: Time taken after each update to reconstruct the data structure.
3. **Query Time**: Time taken to answer query on the updated structure.

1.4 Update-Query Tradeoff

There is often a trade-off between how fast updates are processed with how fast queries can be answered. Suppose a static algorithm has time complexity $\mathcal{O}(T)$. The extremes of the update-query tradeoff occur when we adopt the following approaches.

- **Eager Approach**
  - $\mathcal{O}(T)$ update time.
  - $\mathcal{O}(1)$ query time.
- **Lazy Approach**
  - $\mathcal{O}(1)$ update time.
  - $\mathcal{O}(T)$ query time.

The above approaches are feasible when queries are more frequent, and when updates are more frequent, respectively. Usually when we design an algorithm, we aim to find a sweet spot between these extremes depending on the constraints and requirements of the problem. At times, even when the worst-case bound is proven to be tight, improving the amortized bound is still considered an improvement.

Example 1.4

DSU (**Disjoin Set Union**) is a popular example of a dynamic graph algorithm. It supports two operations, **Join** and **Find**, both of which achieve an amortized time complexity of $\mathcal{O}(\alpha(n))$. Here, $\alpha(n)$ is the **inverse ackermann function**. Suppose we consider an incremental dynamic graph model that has m edge updates. One might think that the total time complexity of this algorithm would be $\mathcal{O}(m\alpha(n))$. However, we can only have $n-1$ edges in the final graph. Therefore, the total time complexity must be $\mathcal{O}(n\alpha(n))$ and the amortized time complexity achieved per update is

$$\mathcal{O}\left(\frac{n\alpha(n)}{m}\right)$$

1.5 Time Complexity Bounds

So far, we have given an informal discussion of the ideal performance of a dynamic algorithm. Now we consider various scenarios and analyse them in detail.

Consider a problem $\mathcal{P}$ having a best static algorithm $T_s(n)$. Now consider a dynamic algorithm $T_d(n)$ for the problem $\mathcal{P}$.

1.5.1 Upper Bound on $T_d(n)$

Since we cannot have $T_d(n)$ perform worse than $T_s(n)$, therefore

$$T_d(n) = \mathcal{O}(T_s(n))$$

1.5.2 Lower Bound on $T_d(n)$

1. If $T_d(n)$ is an incremental algorithm, then consider a graph with size $\mathcal{O}(n)$ constructed with m incremental updates. Applying $T_d(n)$ at every incremental update and summing up, we get

$$mT_d(n) = \Omega(T_s(n)) \implies T_d(n) = \Omega\left(\frac{T_s(n)}{m}\right)$$

2 [Lecture] Jan 14, 2026

In this lecture, we elaborate on our discussion of the lower bounds of $T_d(n)$. Here is the premise of our discussion. Given a problem $\mathcal{P}$ and the best known static algorithm to solve $\mathcal{P}$ called $T_s(n)$, we wish to analyse the time complexity of a dynamic algorithm $T_d(n)$ for solving $\mathcal{P}$.

Lemma 2.1 (Upper Bound on $T_d(n)$)
For any problem $\mathcal{P}$, it always holds that

$$T_d(n) = \mathcal{O}(T_s(n))$$

Proof. If the time complexity of the dynamic algorithm $T_d(n)$ exceeds that of the static algorithm $T_s(n)$, then there is no use in developing the dynamic algorithm. In such a case, one could simply use $T_s(n)$ to recompute the desired quantity after every update. □

2.1 $T_d(n)$ is Incremental

Suppose $T_d(n)$ is an incremental dynamic algorithm for $\mathcal{P}$, meaning that $T_d(n)$ only supports incremental updates. We consider the following two types of incremental updates.

2.1.1 Edge Updates

Let $G(n, m)$ be a graph on n vertices and m edges. We would like to analyse the bounds on $T_d(n)$ when considering **edge insertions** to G.

1. *Upper Bound.* By **lemma 2.1**, we immediately obtain an upper bound on $T_d(n)$ given by
$$\boxed{T_d(n) = \mathcal{O}(T_s(n))}$$
2. *Lower Bound.* Consider a graph with n vertices and no edges. Constructing this graph takes $\mathcal{O}(1)$ time, and we assume that computing $\mathcal{P}$ on this graph also takes $\mathcal{O}(1)$ time. To obtain the target graph G, we must perform m edge insertions. Applying the dynamic algorithm $T_d(n)$ after each edge update yields a valid *static* algorithm to compute $\mathcal{P}$. Since $T_s(n)$ is the best known algorithm for static computation of $\mathcal{P}$ on G, therefore we must have
$$\mathcal{O}(1) + mT_d(n) = \Omega(T_s(n))$$
which implies
$$\boxed{T_d(n) = \Omega\left(\frac{T_s(n)}{m}\right)}$$

2.1.2 Vertex Updates

Let $G(n, m)$ be a graph on n vertices and m edges. Now we would like to analyse the bounds on $T_d(n)$ when considering **vertex insertions** to G.

1. *Upper Bound.* By **lemma 2.1**, we have

$$\boxed{T_d(n) = \mathcal{O}(T_s(n))}$$

2. *Lower Bound.* If each update on G inserts a singleton vertex *without* any edges, then assuming that no nontrivial recomputation is required for $\mathcal{P}$, we obtain the lower bound

$$\boxed{T_d(n) = \Omega(1)}$$

   If instead, each update on G adds a singleton vertex *with* edges, then computing the lower bound for $T_d(n)$ in a similar fashion to that for **edge insertions**, we obtain

$$\mathcal{O}(1) + nT_d(n) = \Omega(T_s(n))$$

   which implies

$$\boxed{T_d(n) = \Omega\left(\frac{T_s(n)}{n}\right)}$$

2.2 $T_d(n)$ is Decremental

We now assume that $T_d(n)$ is a decremental algorithm for $\mathcal{P}$. We first want to solve the problem on a trivial case and then apply a sequence of graph updates to obtain the given graph.

For incremental algorithms, we started with an empty graph. Similarly, for decremental algorithms, we would require a graph from which we could perform a sequence of decremental graph updates and obtain the given graph. A **complete graph** is a suitable choice since, every graph on n vertices is a subgraph of the complete graph on n vertices.

Suppose that we can solve $\mathcal{P}$ on a complete graph trivially, in $\mathcal{O}(1)$ time. If the update operation consists of **edge deletions**, then constructing the complete graph requires $\mathcal{O}(n^2)$ time, and we would require $\mathcal{O}(n^2)$ edge deletions to transform it to G. This implies that

$$\mathcal{O}(n^2) + \mathcal{O}(n^2)T_d(n) = \Omega(T_s(n))$$

From here, we can derive lower bounds on $T_d(n)$ depending upon $T_s(n)$. For example,

$$T_s(n) = \mathcal{O}(n^2) \implies T_d(n) = \Omega(1)$$

and,

$$T_s(n) = \omega(n^2) \implies T_d(n) = \Omega\left(\frac{T_s(n)}{n^2}\right)$$

However, solving $\mathcal{P}$ on a complete graph is not always trivial. For example, the **Traveling Salesman Problem** is exponential on a complete graph. Hence, a general treatment of the time complexity of decremental algorithms is difficult, and we instead analyse specific problems to obtain more meaningful bounds.

2.2.1 Decremental MST (Minimum Spanning Tree)

Let $T_d(n)$ be a decremental algorithm for computing the **Minimum Spanning Tree** of a graph G on n vertices, and let $T_s(n)$ be the best known static algorithm to compute the MST.

> **Definition 2.2.** The **Minimum Spanning Tree** of a connected, weighted, undirected graph G is a subset of edges that connects all the vertices of G and has the minimum sum of weights.

Suppose we consider **edge deletions** for our algorithm. We could obtain a lower bound on $T_d(n)$ by the following method. Introduce a dummy vertex v' and connect it to every vertex $v \in G$ with edges of extremely small weights. Let the resulting graph be G'. To construct G' from G, we require $\mathcal{O}(n)$ time. The minimum spanning tree of G' is trivial because it contains all the edges connecting v'. Deleting these n edges recovers G, and we obtain

$$\mathcal{O}(n) + nT_d(n) = \Omega(T_s(n))$$

and hence

$$\boxed{T_d(n) = \Omega\left(\frac{T_s(n)}{n}\right)}$$

If instead we considered **vertex deletions** for our algorithm, then

$$\mathcal{O}(n) + T_d(n) = \Omega(T_s(n))$$

which implies

$$T_d(n) = \Omega(T_s(n))$$

Using **lemma 2.1**, we obtain

$$\boxed{T_d(n) = \Theta(T_s(n))}$$

This tells us that the we have a tight bound and there is no asymptotic benefit in designing a decremental algorithm for computing the MST that supports only vertex deletions.

2.2.2 Decremental BFS Tree

Let $T_d(n)$ be a decremental algorithm to compute the **BFS Tree** on a graph G with n vertices, and $T_s(n)$ be the best known static algorithm to compute the BFS Tree.

Definition 2.3. The **BFS Tree** of an unweighted graph G is the spanning tree produced by running the BFS algorithm on G.

Let's say we allow **edge deletions**. Observe that we can trivially compute the BFS Tree on a complete graph in $\mathcal{O}(n)$ time. Hence, we could add dummy edges to G to transform it to a complete graph on n vertices in $\mathcal{O}(n^2)$ time. Moreover, we require $\mathcal{O}\left(n^2\right)$ edge deletions to transform the complete graph back to G. Therefore

$$\mathcal{O}(n^2) + \mathcal{O}(n^2)T_d(n) = \Omega(T_s(n))$$

and hence

$$\boxed{T_d(n) = \Omega\left(\frac{T_s(n)}{n^2}\right)}$$

2.2.3 Decremental Traveling Salesman Problem

Let $T_d(n)$ be a decremental algorithm to compute the **Traveling Salesman Problem** on a graph G with n vertices, and $T_s(n)$ be the best known algorithm to compute this problem.

Definition 2.4. Given a complete, weighted, undirected graph G and a source vertex s, the **Traveling Salesman Problem** asks for a minimum weight cycle starting and ending on s that visits all the vertices in graph G, exactly once.

Suppose we allow **vertex deletions** for our algorithm. Add n dummy vertices to the graph G and for each dummy vertex v, connect all the vertices in G to v with extremely small weights. Let the resulting graph be G'. In this graph, computing the minimum weight cycle that visits all the vertices is trivial. It is just a cycle that alternates between each vertex of G and the dummy vertices. Constructing G' takes $\mathcal{O}(n^2)$ time, and computing the minimum weight cycle requires $\mathcal{O}(n)$ time. Further, we require n vertex deletions to transform G' to G. Hence

$$\mathcal{O}(n^2) + nT_d(n) = \Omega(T_s(n))$$

and therefore

$$\boxed{T_d(n) = \Omega\left(\frac{T_s(n)}{n}\right)}$$

If instead we allow **edge deletions** for our algorithm, then we would have to relax the constraint on the Traveling Salesman Problem for G to be a complete graph, so that every edge deletion results in valid subproblem. We construct graph G' again, but in this case we require $\mathcal{O}(n^2)$ edge deletions to transform G' to G. Therefore

$$\mathcal{O}(n^2) + \mathcal{O}(n^2)T_d(n) = \Omega(T_s(n))$$

and hence

$$\boxed{T_d(n) = \Omega\left(\frac{T_s(n)}{n^2}\right)}$$

2.3 $T_d(n)$ is Fully Dynamic

For fully dynamic algorithms, it is difficult to make general statements. However, since incremental and decremental algorithms are special cases of fully dynamic algorithms, any lower bound for these special cases also yields a (possibly weaker) lower bound for fully dynamic algorithms.

Remark 2.5 — There cannot exist a polynomial time dynamic algorithm for an **NP-hard** problem, because if such an algorithm existed then we could apply it in an incremental fashion to obtain a polynomial time static algorithm to solve the problem, contradicting NP-hardness.

3 [Lecture] Jan 16, 2026

We begin our discussion with some general comments on *hardness* of problems.

Lemma 3.1

The **fully dynamic version** of a problem is at least as hard as the **partially dynamic version**.

Proof. Suppose there exists a problem $\mathcal{P}$ for which the partially dynamic version version is *harder* than the fully dynamic version. Consider an algorithm A that solves the problem $\mathcal{P}$ in the fully dynamic version. Then the same algorithm can be used to solve the problem in the partially dynamic version with equal efficiency. This is because the fully dynamic version of $\mathcal{P}$ allows all types of updates. If we were to restrict the sequence of updates to only increments or decrements, we would have the partially dynamic version. This contradicts our assumption that the partially dynamic version is harder than the fully dynamic version. □

Lemma 3.2

A problem $\mathcal{P}$ on **directed graphs** is atleast as hard as if it were on **undirected graphs**.

Proof. An undirected graph can be modelled as a directed graph by replacing each of its undirected edge $e(u, v)$ by two directed edges $d(u, v)$ and $d(v, u)$. Therefore, an algorithm A that solves $\mathcal{P}$ on directed graphs can also be used to solve $\mathcal{P}$ on undirected graphs. Hence, the problem $\mathcal{P}$ on directed graphs is atleast as hard as it is on undirected graphs. □

Lemma 3.3

If T_1 and T_2 are the **amortized** and **worst-case** time complexities of a problem $\mathcal{P}$, then

$$T_1 \leq T_2$$

Proof. This follows from

$$T_1 = \frac{\text{Total Time}}{\#\text{Updates}} \leq \frac{T_2 \times \#\text{Updates}}{\#\text{Updates}} = T_2$$

□

We now discuss the classic **single source reachability problem**, which is defined on directed graphs and asks if there exists a path from node s to t in a directed graph G. The undirected version of this problem is called the **connectivity problem**. We

are interested in designing an incremental algorithm for the single source reachability problem.

3.1 Static Single Source Reachability

Consider the *static single source reachability problem*.

Problem 3.1 (Static Single Source Reachability)
Given a directed graph $G(V, E)$ and a source vertex $s \in V$, design an algorithm to determine which vertices are reachable from s.

This problem can be solved using either a **depth first search** (DFS) or a **breadth first search** (BFS) trasversal starting from s, which computes a **reachability tree** T rooted at s. We maintain an array R of size $n = |V|$, where

$$R[u] = \begin{cases} 1, & \text{if u is reachable from s} \\ 0, & \text{otherwise} \end{cases}$$

Initially, all entries of R are set to 0. After computing the reachability tree T, we set $R[u] = 1$ for every vertex $u \in T$. Subsequently, **reachability queries** can be answered in constant time by checking if $R[u] = 1$.

Algorithm 1 Static Single Source Reachability

Require: Graph $G(V, E)$, source vertex $s \in V$
Ensure: Array R where $R[u] = 1$ if and only if u is *reachable* from s
1: **for all** $u \in V$ **do**
2: $\quad R[u] \leftarrow 0$
3: **end for**
4: Run DFS or BFS from s, obtaining a reachability tree T
5: **for all** $u \in T$ **do**
6: $\quad R[u] \leftarrow 1$
7: **end for**
8: **return** R

Algorithm 2 Reachability Query

Require: Array R, vertex u
Ensure: **true** if u is *reachable* from s, otherwise **false**
1: **if** $R[u] = 1$ **then**
2: $\quad$ **return true**
3: **else**
4: $\quad$ **return false**
5: **end if**

Remark 3.4 — Since we use a DFS/BFS, it takes $\mathcal{O}(n+m)$ time for the algorithm to complete and each reachability query can be answered in $\mathcal{O}(1)$ time. The algorithm uses $\mathcal{O}(n)$ additional space for the array R.

3.2 Incremental Single Source Reachability

We now consider the *incremental* version of the problem.

Problem 3.2 (Incremental Single Source Reachability)
Given a directed graph $G(V, E)$ having n vertices and m edges with a source vertex $s \in V$, design an **incremental algorithm** that computes all vertices reachable from s under edge insertions to G.

We are interested in maintaining the set of vertices reachable from the source vertex s under edge insertions to the directed graph G. We begin with a few definitions before designing an algorithm for the problem.

3.2.1 Definitions

Definition 3.5. Let $G(V, E)$ be a directed graph. Then we use $\rightsquigarrow$ to denote that there exists a directed path from u to v in G and this is defined as the *reachability relation*. A vertex u is said to be *reachable* from s in G, if $s \rightsquigarrow u$. Similarly, a vertex u is said to be *unreachable* from s in G, if there does not exist a path from s to v.

Definition 3.6. Let $G(V, E)$ be a directed graph with source vertex s. A vertex w is said to be ***newly reachable*** from s, if after inserting $e(u, v)$ to G, the vertex w, which was previously unreachable from s, becomes reachable from s.

3.2.2 Algorithm

Consider the insertion of a directed edge $e(u, v)$ to G. This edge insertion can only increase the set of vertices reachable from s. It can never remove any reachable vertex.

If v is already reachable from s, then the insertion of e has no effect. Similarly, if u is not reachable from s, then adding an outgoing edge from u cannot introduce any new reachable vertices.

However, if u is reachable from s and v is not, then adding e makes v reachable from s. Moreover, every vertex reachable from v that was previously unreachable from s also becomes reachable from s. Therefore, we should **perform a graph traversal** starting from v and add all vertices that are reachable from v and previously unreachable from s to our data structure.

Algorithm 3 Incremental Single Source Reachability

Require: Graph $G(V, E)$, source vertex $s \in V$
Ensure: Array R where $R[u] = 1$ if and only if u is *reachable* from s

```
1: function UPDATE(u, v)
2:     if R[u] = 1 and R[v] = 0 then
3:         R[v] ← 1
4:         for all (v, w) ∈ E do
5:             UPDATE(v, w)
6:         end for
7:     end if
8: end function
```

3.2.3 Correctness

We will now prove the **correctness of the algorithm**. Essentially, we want to show that the algorithm correctly identifies all vertices that become newly reachable from s after an edge insertion.

Lemma 3.7

Let $G(V, E)$ be a directed graph with a source vertex s. Adding a directed edge $e(u, v)$ to G can never decrease the set of reachable vertices from s.

Proof. Define $G' = (V, E \cup \{e(u, v)\})$. Clearly, every edge $e \in G(E)$ also belongs to $G'(E)$. Therefore, any path p in G also belongs to G'. Suppose Q is the set of vertices reachable from s in G. For every vertex $v \in Q$, let p be the path from s to v. Since p also belongs to G', therefore v is reachable from s in G'. Therefore, every vertex reachable from s in G remains reachable from s in G'. Hence, adding a directed edge cannot decrease the set of reachable vertices from s. □

Lemma 3.8 (Reachability is Transitive)

Let $G(V, E)$ be a directed graph. Then the reachability relation $\rightsquigarrow$ is transitive. That is, $\forall\ u, v, w \in G(V)$

$$u \rightsquigarrow v \text{ and } v \rightsquigarrow w \implies u \rightsquigarrow w$$

Proof. Given that $u \rightsquigarrow v$ and $v \rightsquigarrow w$, we know that there exist directed paths p_1 and p_2 in G that connect u to v and v to w. Concatenating these paths to form a new path $p = p_1 \cup p_2$ implies that p is the directed path from u to w implying that $u \rightsquigarrow w$. □

Lemma 3.9

Let $G(V, E)$ be a directed graph with a source vertex s. Adding a directed edge $e(u, v)$ to G will increase the set of reachable vertices from s *if and only if* u is reachable from s and v is not.

Proof. If u is reachable from s and v is not, then adding a directed edge from u to v makes v reachable. Hence, the set of reachable vertices increases.

To prove the other direction, define $G' = (V, E \cup \{e(u, v)\})$. Suppose v is reachable from s in G. If adding the edge e made a vertex w newly reachable, then there exists a path p from s to w in G', but not in G. Since every edge $e \in G(E)$ also belongs to $G(E')$, hence p must contain $e(u, v)$. We can conclude from here that w must be reachable from v in G. Using **lemma 3.8**, and v being reachable from s and w from v in G implies that w is reachable from s, contradicting the fact that w is newly reachable. Hence, no such w exists.

Similarly, if u is not reachable from s and adding e to G makes a vertex w newly reachable, then there exists a path p from s to w in G', but not in G. Since every edge $e \in G(E)$ also belongs to $G(E')$, hence p must contain $e(u, v)$. However this implies that u is reachable from s, contradicting our assumption. Hence, no such w exists and the set of reachable vertices from s does not increase. □

Main Proof. We now show that **Algorithm 3** correctly computes the set of newly reachable vertices when a directed $e(u, v)$ is added to G. For the sake of contradiction, assume there exists a vertex w which is newly reachable vertex and is not computed by the algorithm.

By definition, w is reachable from s in G' but not in G. Hence there must exist a path p in $G' = (V, E \cup \{e(u, v)\})$ from s to w such that p does not exist in G. Since all the edges $e \in G(E)$ also belongs to $G'(E)$, therefore the path p must contain $e(u, v)$. Thus, w is reachable from v in G'.

Now consider the path from v to w. Consider any vertex x on this path, such that $x \neq v, w$. Clearly x is reachable from s in G'. Furthermore, we can say that x must be newly reachable too. If x was previously reachable then, w must have been previously reachable too because the path from x to w exists in G. Therefore, every vertex on this path is newly reachable.

The algorithm first identifies v as newly reachable and then performs a traversal that marks every previously unreachable vertex reachable from v. By induction on the length of the path, the algorithm must discover every vertex on the path from v to w, including w itself.

This contradicts our assumption that w is not computed by the algorithm. Hence, no newly reachable vertex is missed, and the algorithm is correct. □

3.2.4 Time Complexity Analysis

For **Algorithm 3**, we show that the

1. **Worst-case time complexity** is $\mathcal{O}(m+n)$.

   Proof. If a vertex u is newly reachable, then we perform $(1+\deg u)$ operations on it. Thus, the total running time of the algorithm is

   $$\sum_{u \in Q} (1 + \deg u) = \mathcal{O}(m+n)$$

   where Q is the set of newly reachable vertices. □

2. **Amortized time complexity** is $\mathcal{O}(1)$.

   Proof. Since the algorithm reduces edges $e(u,v)$ of the form

   $$(R[u]=1, R[v]=0) \mapsto (R[u]=1, R[v]=1)$$

   Hence, each edge could be reduced *atmost once*. Thus, the number of operations are bounded by m and the amortized time complexity turns out to be

   $$\frac{1}{m} \sum_{\substack{v \in V \\ R[v]=1}} \big(1 + \deg v\big) = \frac{\mathcal{O}(m+n)}{m} = \mathcal{O}(1)$$

   □

4 [Lecture] Jan 20, 2026

We discuss some methods to compute the amortized time of an algorithm.

4.1 Amortized Analysis

1. **Aggregate Method**. Compute the total cost of n updates and divide it by n.
2. **Accounting Method**. Assign an extra credit to each operation to pay for expensive operations later on.

$$\text{Amortized Cost} = \text{Assigned Cost} + \text{Extra Credit}$$

3. **Potential Method**. Assign a potential function and compute the credit in the form of a potential.

$$\text{Amortized Cost} = \text{Actual Cost} + \text{Change in Potential}$$

To see how these methods are used, we consider a sample problem and compute its amortized time using the above methods.

4.2 Bit Counter

Problem 4.1

Consider the binary representation of the numbers. Incrementing by one unit, several bits may change in the binary representation. Compute the average number of bits flipped per increment.

Define $T(n)$ as the number of bits flips after n increments. The amortized cost is

$$\leq \frac{T(n)}{n}$$

4.2.1 Aggregate Method

For a number n, the number of bits in its binary representation are exactly

$$1 + \lfloor \log_2 n \rfloor$$

Observe that each increment flips the lowest significant bit of n. The next significant bit is flipped atmost $\lfloor \frac{n}{2} \rfloor$ and so on. Using induction we can prove that for the ith significant bit of n, the number of times this bit is flipped in the n increments is

$$\left\lfloor \frac{n}{2^{i-1}} \right\rfloor$$

Therefore, the total cost of n increments can be computed as

$$\begin{aligned}\#(\text{Total Cost}) &= n + \left\lfloor \frac{n}{2} \right\rfloor + \left\lfloor \frac{n}{4} \right\rfloor + \ldots + \left\lfloor \frac{n}{2^{\lfloor \log_2 n \rfloor}} \right\rfloor \\ &< n \left(1 + \frac{1}{2} + \frac{1}{4} + \ldots\right) \\ &= 2n\end{aligned}$$

Finally, the amortized time complexity $T(n)$ is calculated as

$$\frac{\#(\text{Total Cost})}{n} = 2$$

4.2.2 Accounting Method

We observe that every operation flips the least significant bit. So let's assign the cost of each operation as 1. Since we will need to flip the second LSB once in two times, the third LSB once in four times and so on $\implies$ thus we could store an extra credit of

$$\frac{1}{2} + \frac{1}{4} + \ldots + \frac{1}{2^{\lfloor \log_2 n \rfloor}}$$

for every operation. Therefore, the amortized cost amounts to

$$1 + \frac{1}{2} + \frac{1}{4} + \ldots + \frac{1}{2^{\lfloor \log_2 n \rfloor}} < 1 + \frac{1}{2} + \frac{1}{4} + \ldots = 2$$

4.2.3 Potential Method

Define a potential function ϕ as the number of ones in the binary representation of the number.

Lemma 4.1

For a number n, suppose W is the cost of incrementing n. Then

$$W + \Delta\phi = W + \phi(n+1) - \phi(n) = 2$$

Proof. Suppose n has k consecutive least significant set bits. The cost of incrementing n is $(k+1)$. Since, $\phi(n) = T + k$ and $\phi(n+1) = T + 1$, where T is the remaining set bits. Hence,

$$W + \Delta\phi = (k+1) + (T + 1 - (T + k)) = 2$$

□

Suppose W_i corresponds to the actual cost of incrementing in the ith update. Then

$$\sum_{i=1}^{n} (W_i + \Delta\phi) = \phi(n) - \phi(0) + \sum_{i=1}^{n} W_i = 2n$$

which can be written as

$$\left(\frac{W_1 + W_2 + \ldots + W_n}{n}\right) + \left(\frac{\phi(n) - \phi(0)}{n}\right) \leq 2$$

Since by definition of ϕ, we have $\phi(n) > \phi(0)$. Hence

$$\text{Amortized Cost} = \left(\frac{W_1 + W_2 + \ldots + W_n}{n}\right) \leq 2$$

4.2.4 Properties of Potential Functions

To see why the potential method works, observe that

$$\begin{aligned} w_1 + \phi_1 - \phi_0 &\leq k \\ w_2 + \phi_2 - \phi_1 &\leq k \\ &\vdots \\ w_n + \phi_n - \phi_{n-1} &\leq k \end{aligned}$$

Summing up all the inequalities, we have

$$\begin{aligned} &(w_1 + w_2 + \ldots + w_n) + \phi_n - \phi_0 \leq nk \\ \iff &\left(\frac{w_1 + w_2 + \ldots + w_n}{n}\right) \leq k - \left(\frac{\phi_n - \phi_0}{n}\right) \end{aligned}$$

The choice for ϕ is made such that

1. $\phi_i \geq 0$
2. $\left(\frac{\phi_0}{n}\right)$ is negligible compared to k

Homework 4.2. Compute the average number of digit changes per increment for a **digit counter**. Generalise the result for a **B-ary counter**.

4.3 Multi-Pop Stack

Problem 4.2 (Multi-Pop Stack)

Consider a stack that supports the following operations.

1. PUSH(X): Push x to the top of the stack.
2. POP(K): Pop k elements from the stack.

Compute the amortized time complexity of the data structure.

4.3.1 Aggregate Method

Suppose T is the total cost of operations. Then

$$T = \#\,(\text{Push Cost}) + \#\,(\text{Pop Cost})$$

Since each element that is pushed can be either popped once or choose to stay in the stack, hence

$$\#\,(\text{Pop Cost}) \leq \#\,(\text{Push Cost})$$

Therefore, we can write the amortized time as

$$\begin{aligned}\text{Amortized Time} &= \frac{\#\,(\text{Push Cost}) + \#\,(\text{Pop Cost})}{n} \\ &\leq \frac{2}{n}\,(\#\,(\text{Push Cost})) \\ &\leq \frac{2n}{n} = 2\end{aligned}$$

4.3.2 Accounting Method

For a push operation, we assign a cost of one unit and an extra credit of one unit for a future pop for this element. The pop operation cost thus assigned is zero, because the operation cost is already accounted in the extra credit stored for the element when it was pushed. Thus,

$$\text{Amortized Time} = 1 + 1 = 2$$

4.3.3 Potential Method

Define the potential function ϕ as the number of elements in the stack. Then for a push operation

$$W + \Delta\phi = 1 + 1 \leq 2$$

and for a pop operation

$$W + \Delta\phi = k - k = 0 \leq 2$$

Hence, we can claim that

$$\text{Amortized Time} = 2$$

Homework 4.3. Implement a queue using two stacks. Design a formal data structure, state the algorithm and show that the amortized time of operation is $\mathcal{O}(1)$.

Remark 4.4 — **Amortized analysis** is used when expensive operations are rare. It's not possible to perform an amortized analysis of binary search, for example. Moreover, if we consider modifications of the multi-pop stack such as a **multi-push stack**, we still cannot hope to perform an amortized analysis of this data structure.

Remark 4.5 — **Amortized analysis** can be used to compute not only the average time complexity, but also the best case time and worst case time complexity. Consider

$$\text{Amortized Time} = W + \phi$$

The expression results in best case time complexity when ϕ is maximised and a worst case time complexity when ϕ is minimised.

4.4 Amortized Analysis of Incremental SSR

4.4.1 Aggregate Method

4.4.2 Accounting Method

4.4.3 Potential Method

5 [Homework] Jan 20, 2026

6 [Lecture] Jan 21, 2026

Today's discussion is very important from an analysis standpoint.

6.1 Analysis of Algorithms

When we talk about analysing the complexity, it could be of either three entities.

1. **Problems**. Consider a problem $\mathcal{P}$. There could be multiple algorithms to solve this problem:

   $$\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_n$$

   each with a time complexity

   $$T_1, T_2, \ldots, T_n$$

   > **Definition 6.1.** The **complexity of a problem** is defined as the lowest running time among all the existing algorithms, assuming no other extra constraint on the problem.

   Example 6.2

   For example, consider the problem of **sorting**. There are several well known algorithms for sorting that run in $\mathcal{O}(n^2)$ (such as the **Bubble Sort**) and even in $\mathcal{O}(n \log n)$ (such as the **Merge Sort**). Since there may be more efficient ways to solve a problem than the existing algorithms. Hence, we say that the existence of algorithms to a problem gives an upper bound on the complexity of the problem.
   However in case of **sorting**, we can show that there *cannot* exist a comparison-based sorting algorithm that runs in time complexity faster than $\Omega(n \log n)$. Thus, this establishes a tight bound on the **complexity of the sorting problem**

   $$\boxed{\Theta(n \log n)}$$

2. **Algorithms**. Consider an algorithm $\mathcal{A}$. An algorithm performs a sequence of instructions on an input and produces an output. Thus, the parameter that varies in this process is the *input* to the algorithm. So when we talk about the **complexity** of an algorithm, we are possibly talking about either the

   a) *Best case time complexity.*

   b) *Average time complexity.*

   c) *Worst case time complexity.*

The best case time complexity is the running time corresponding to the best case input. Similarly, the worst case time complexity is the running time corresponding to the worst case input. The average time complexity of an algorithm is the expected running time over all inputs. We could either assume each input occurs equiprobably or consider a probability distribution.

Example 6.3

For example, consider **Quick Sort**. It has a worst case time complexity of $\mathcal{O}(n^2)$ when the array is reverse sorted and a best case time complexity of $\mathcal{O}(n)$ when the array is sorted. The average time complexity of the algorithm is $\mathcal{O}(n \log n)$ that is resulted when the average is taken over all permutations of the input array, assuming that each permutation is equally likely.

3. **Functions**. A mathematical function has a lower bound and an upper bound and thus, we can define the complexity of a mathematical function too.

6.2 Estimating Algorithms

An algorithm performs differently for different inputs. Overall, we have the following

1. **Best Case** is the fastest performance for any input.
2. **Average Case** is the average performance for all inputs.
3. **Worst Case** is the slowest performance for any input.

To establish the **best case bound** for an algorithm, we need to find the **best case** example and show that for all other inputs, we cannot achieve a better running time. It's worth mentioning that the running time of a particular input yields an upper bound on the best case bound and to establish a lower bound on the same, we resort to analysis techniques.

Similarly, to establish the **worst case bound** for an algorithm, we need to find the **worst case** example and show that for all other inputs, we cannot achieve a worse running time. Again, the running time of a particular input yields a lower bound on the worst case bound and to establish an upper bound on the same, we resort to analysis techniques.

For example, consider the **Euclid's gcd Algorithm**. It is an algorithm that is used to compute the greatest common divisor of two natural numbers and it can be shown that the worst case time complexity of the algorithm is $\mathcal{O}(\log n)$. The worst case input is

$$(a, b) = (f_n, f_{n-1})$$

where f_i is the ith fibonacci number, for which the algorithm has the worst case running time.

Algorithm 4 Euclid's gcd Algorithm

```
1: function GCD(a, b)
2:     if a > b then
3:         return GCD(b, a)
4:     else if a = 0 then
5:         return b
6:     else
7:         return GCD(b mod a, a)
8:     end if
9: end function
```

6.3 Incremental All Pairs Reachability

7 [Tutorial] Jan 23, 2026

Some problems on *Amortized Analysis*.

7.1 Problem 1: Tall Buildings

Problem 7.1

Chicago has many tall buildings, but only some of them have a clear view of Lake Michigan. Suppose we are given an array $A[1 \dots n]$ that stores the height of n buildings on a city block, indexed from west to east. Building i has a good view of Lake Michigan if and only if every building to the east of i is shorter than i. Use a stack to design an algorithm that computes which buildings have a good view of Lake Michigan in $\mathcal{O}(n)$ total time.

We use a *stack* to solve this problem. The stack S would maintain the sequence of buildings that have a good view of Lake Michigan among the buildings processed. The outline of the algorithm is as follows: iterate on the array A from left to right, and insert the current building to the stack while removing those that have their view disturbed due to the current added building.

```
1: function INSERT(x)
2:     while top(S) ≤ x do
3:         pop(S)
4:     end while
5:     push(x, S)
6: end function
```

The property that we claim is invariant in the above algorithm is that stack S ensures a sequence of buildings sorted in descending order. Now we prove the correctness of the algorithm and show that it does not violate the invariant, thus satisfying it.

Correctness Proof.

Lemma 7.1

$x \notin S$ (or in other words, x is removed from stack S) $\iff \exists\ y$ to the right of x such that $x \leq y$.

Proof. Observe that every element is present in the stack at some point of time. If x is removed from the stack, then it must have been popped from the stack while processing a y which must have occurred after x, so y is on the right of x. If $\exists\ y$ to the right of x such that $x \leq y$ and $x \in S$, then while processing y the while loop will pop off x implying $x \notin S$. □

This implies that the stack S is sorted in decreasing order, because if there exists x and y in the stack such that $x \leq y$ and y is on the right of $x \implies x \notin S$. Therefore the algorithm correctly computes a sequence of buildings in decreasing order of height. If there exists a building x such that all buildings to its right are smaller than it, then by the above lemma it can never be popped and hence must exist in $S \implies$ algorithm does not leave out any building in the longest decreasing sequence.

Time Complexity Analysis.

The algorithm is similar to a multi-pop single-push stack which has an amortized time complexity of $\mathcal{O}(n)$, and hence so does our algorithm.

7.2 Problem 2: Queue using two Stacks

Problem 7.2

Queues and Stacks are two commonly used data structures, queues implement First In First Out policy and stacks implement Last In First Out Policy. Describe an implementation of the *push* and *pop* operations of a queue, using two stacks where the operations of stacks are used as a black box. Show that the queue takes $\mathcal{O}(1)$ amortized time per operation.

We use two stacks, label them as S_1 and S_2. S_1 is the stack used to contain elements that are pushed into the queue and S_2 is the stack used to contain elements that will answer the pop operations. Everytime we push x in the queue, we push x to S_1. If want to pop from the queue, we check if S_2 is non-empty. If it is non-empty, then the top most element of S_2 is popped off. If it empty, then we empty S_1 into S_2 in the reverse order and the top most element of S_2 is popped off.

```
1:  function QueuePush(x)
2:      Push(x, S_1)
3:  end function
4:  function QueuePop()
5:      if not S_2.empty() then
6:          return Pop(S_2)
7:      else
8:          while not S_1.empty() do
9:              x ← Pop(S_1)
10:             Push(x, S_2)
11:         end while
12:         return Pop(S_2)
13:     end if
14: end function
```

Proposition 7.2

We claim that the above algorithm maintains the following invariants.

1. Elements in S_1 are in descending order of their time of arrivals.
2. Elements in S_2 are in increasing order of their time of arrivals.
3. Earliest time in S_1 is $>$ Latest time in S_2.

We will first establish the correctness of the algorithm, which is that it correctly pops off the first element that was pushed to the queue, and show that the algorithm does not violate the stated invariant.

Correctness Proof.

Lemma 7.3

first element pushed to the queue $\iff$ popped first from the queue.

Proof. Suppose x and y are pushed to the queue such that x was pushed before y. If x and y are in S_1, then x occurs below y in the stack S_1 because y was pushed later. Otherwise, x occurs in S_2 when y was pushed and definitely x will be popped before y. If the pop operation is applied when x and y occur in S_1, then the contents of S_1 are moved to S_2 in reverse direction so x occurs above y in S_2 and thus x will get popped before y. This implies that the first element pushed to the queue is popped first from the queue.

Consider the element x which was popped from the queue and another element y in the queue. Then x and y must be in stack S_2 before x was popped and x must have been above y in the stack. So when x and y were moved to S_2 due to a pop operation, hence x lied below y in $S_1 \implies x$ was pushed to the queue before y was pushed, which proves the other direction too. □

This implies that the algorithm correctly computes the elements to be popped. We can further show that the algorithm maintains the previously stated invariant.

Time Complexity Analysis.

"Very rough writeup. Need to write it in a better way, as soon as i get some free time"

Accounting Method =¿ assign an extra credit of 3. Hence amortized time complexity is $\mathcal{O}(n)$.

Potential Method =¿ number of elements in the stack S_1.

7.3 Problem 3: Dynamic Arrays

Space Complexity =¿ $\mathcal{O}(n)$ due to the bound $n \leq k \leq 4n$.

Time Complexity =¿ Accounting Method: assign +2 to insertion, +1 to deletion Potential Method: use the potential function $\phi = \max(2n - k, \frac{k}{2} - n)$ Aggregate Method: $2^k \leq t \leq 2^{k+1}$. Insertion is $t + 2 \cdot \left(2^k - 1\right) = \mathcal{O}(t)$. Similarly, for deletions too.

8 [Lecture] Jan 28, 2026

In this lecture we discuss about the incremental versions of the connectivity problem. By definition, the connectivity problem is defined for undirected graphs.

Definition 8.1. In a graph $G = (V, E)$, we say that u is **connected** to v if there exists a path of edges from u to v for $u, v \in V$.

8.1 Incremental Single Source Connectivity

Problem 8.1 (Incremental Single Source Connectivity)
Given an undirected graph $G = (V, E)$ and a source vertex s, maintain the set of vertices connected to s under edge additions.

Since an undirected graph can be modelled as a directed graph, therefore algorithms that can compute the **Incremental Single Source Reachability** can also be used here. This leads to

$$\text{Worst case time} = \mathcal{O}(m), \text{ and Amortized time} = \mathcal{O}(1)$$

8.2 Incremental All Pairs Connectivity

Problem 8.2 (Incremental All Pairs Connectivity)
Given an undirected graph $G = (V, E)$, maintain the sets of vertices connected to each $x \in V$ under edge additions.

We maintain a two-dimensional array $R[n][n]$, where

$$R[u][v] = \begin{cases} 1, & \text{if v is connected to u} \\ 0, & \text{otherwise} \end{cases}$$

Following a similar algorithm to **Incremental Single Source Reachability** and updating the data structure for all vertices $u \in V$, we end up with

$$\text{Worst case time} = \mathcal{O}(mn), \text{ and Amortized time} = \mathcal{O}(n)$$

8.3 Disjoint Set Union

Another data structure that could be used to solve the **Incremental All Pairs Connectivity** problem is **Disjoint Set Union**. It is a data structure that supports

1. FIND: identify the component an element belongs to.
2. UNION: merge two components.

With this data structure, we can propose an algorithm to solve the **Incremental All Pairs Connectivity** problem.

```
1: function UPDATE(x, y)
2:     Add e(x, y) to E
3:     MERGE(x, y)
4: end function
5: function QUERY(x, y)
6:     return FIND(x) == FIND(y)
7: end function
```

So now we focus on improving the time complexity of this data structure.

8.3.1 DSU using Lists and Arrays

We use two data structures here.

1. List $L[i]$ that stores all elements in a component c_i.
2. Array $A[i]$ that stores the component index for each i.