Recap: Asymptotic Analysis

**Five Representative Problems**
- Algorithmic Techniques: Greedy, Dynamic Programming, Network Flow
- Computationally Intractable Problems: Unlikely that polynomial time algorithm exists.

**Formal Definition of Big O, \( \Omega \), \( \Theta \) notation**
- \( T(n) \in O(f(n)) \) ---- upper bound
- \( T(n) \in \Omega(f(n)) \) ---- lower bound
- \( T(n) \in \Theta(f(n)) \) ---- lower bound and upper bound
- \( \Theta(n) \) is also referred to as \( \Theta(n) \) time.

**Polynomial Time function** \( T(n) \in O(n^d) \) for some constant \( d \) (\( d \) is independent of the input size).

---

2.4 A Survey of Common Running Times

**Linear Time: \( O(n) \)**

- **Linear time.** Running time is proportional to input size.
- **Computing the maximum.** Compute maximum of \( n \) numbers \( a_1, \ldots, a_n \).

\[
\begin{align*}
\text{max} & = \text{a}_1, \\
\text{for } i = 2 \text{ to } n \{ & \text{if } a_i > \text{max} \\
& \text{max} \leftarrow a_i \\
& \}; \\
\end{align*}
\]

**Claim.** Merging two lists of size \( n \) takes \( O(n) \) time.

**Proof:** After each comparison, the length of output list increases by 1.

---

**Linear Time: \( O(n \log n) \)**

- **Arises in divide-and-conquer algorithms.**
- **Sorting.** Merge sort and heap sort are sorting algorithms that perform \( O(n \log n) \) comparisons.

**Largest empty interval.** Given \( n \) time-stamps \( x_1, \ldots, x_n \) on which copies of a file arrive at a server, what is largest interval of time when no copies of the file arrive?

**Also referred to as linearithmic time.**

\[
\begin{align*}
\text{Largest empty interval: } & \text{Sort the time-stamps. Scan the sorted list in order, identifying the maximum gap between successive time-stamps.} \\
\end{align*}
\]
Quadratic Time: $O(n^2)$

**Quadratic time.** Enumerate all pairs of elements.

**Closest pair of points.** Given a list of $n$ points in the plane $(x_1, y_1), \ldots, (x_n, y_n)$, find the pair that is closest.

**$O(n^2)$ solution.** Try all pairs of points.

```plaintext
min \epsilon: (x_i - x_j)^2 + (y_i - y_j)^2 
for i = 1 to n {
    for j = i+1 to n {
        d \epsilon (x_i - x_j)^2 + (y_i - y_j)^2
        if (d < min)
            min \epsilon d
    }
}
```

**Remark:** $O(n^2)$ seems inevitable, but this is just an illusion.

Cubic Time: $O(n^3)$

**Cubic time.** Enumerate all triples of elements.

**Set disjointness.** Given $n$ sets $S_1, \ldots, S_n$, each of which is a subset of $1, 2, \ldots, n$, is there some pair of these which are disjoint?

**$O(n^3)$ solution.** For each pair of sets, determine if they are disjoint.

```plaintext
foreach set $S_i$ {
    foreach other set $S_j$ {
        foreach element $p$ of $S_i$ {
            determine whether $p$ also belongs to $S_j$
        }
        if (no element of $S_i$ belongs to $S_j$)
            report that $S_i$ and $S_j$ are disjoint
    }
}
```

Polynomial Time: $O(n^k)$

**Independent set of size $k$.** Given a graph, are there $k$ nodes such that no two are joined by an edge?

**$O(n^k)$ solution.** Enumerate all subsets of $k$ nodes.

```plaintext
foreach subset $S$ of $k$ nodes {
    check whether $S$ is an independent set
    if ($S$ is an independent set)
        report $S$ is an independent set
}
```

**Polynomial time for $k=17$, but not practical**

```plaintext
Cubic Time: $O(n^3)$
```

**Set disjointness.** Given $n$ sets $S_1, \ldots, S_n$, each of which is a subset of $1, 2, \ldots, n$, is there some pair of these which are disjoint?

**$O(n^3)$ solution.** For each pair of sets, determine if they are disjoint.

```plaintext
foreach set $S_i$ {
    foreach other set $S_j$ {
        foreach element $p$ of $S_i$ {
            determine whether $p$ also belongs to $S_j$
        }
        if (no element of $S_i$ belongs to $S_j$)
            report that $S_i$ and $S_j$ are disjoint
    }
}
```

**Remark:** $O(n^2)$ seems inevitable, but this is just an illusion.

**Quadratic time.** Enumerate all pairs of elements.

**Closest pair of points.** Given a list of $n$ points in the plane $(x_1, y_1), \ldots, (x_n, y_n)$, find the pair that is closest.

**$O(n^2)$ solution.** Try all pairs of points.

```plaintext
min \epsilon: (x_i - x_j)^2 + (y_i - y_j)^2 
for i = 1 to n {
    for j = i+1 to n {
        d \epsilon (x_i - x_j)^2 + (y_i - y_j)^2
        if (d < min)
            min \epsilon d
    }
}
```

**Remark:** $O(n^2)$ seems inevitable, but this is just an illusion.

**Cubic Time: $O(n^3)$**

**Cubic time.** Enumerate all triples of elements.

**Set disjointness.** Given $n$ sets $S_1, \ldots, S_n$, each of which is a subset of $1, 2, \ldots, n$, is there some pair of these which are disjoint?

**$O(n^3)$ solution.** For each pair of sets, determine if they are disjoint.

```plaintext
foreach set $S_i$ {
    foreach other set $S_j$ {
        foreach element $p$ of $S_i$ {
            determine whether $p$ also belongs to $S_j$
        }
        if (no element of $S_i$ belongs to $S_j$)
            report that $S_i$ and $S_j$ are disjoint
    }
}
```

**Polynomial Time: $O(n^k)$**

**Independent set of size $k$.** Given a graph, are there $k$ nodes such that no two are joined by an edge?

**$O(n^k)$ solution.** Enumerate all subsets of $k$ nodes.

```plaintext
foreach subset $S$ of $k$ nodes {
    check whether $S$ is an independent set
    if ($S$ is an independent set)
        report $S$ is an independent set
}
```

**Polynomial time for $k=17$, but not practical**

**Exponential Time**

**Independent set.** Given a graph, what is maximum size of an independent set?

**$O(2^n)$ solution.** Enumerate all subsets.

```plaintext
2^n \leq \phi 
foreach subset $S$ of nodes {
    check whether $S$ is an independent set
    if ($S$ is largest independent set seen so far)
        update $S^* \leftarrow S$
}
```
Undirected Graphs

Undirected graph. \( G = (V, E) \)
- \( V \) = nodes.
- \( E \) = edges between pairs of nodes.
- Captures pairwise relationship between objects.
- Graph size parameters: \( n = |V|, m = |E| \).

\[ V = \{1, 2, 3, 4, 5, 6, 7, 8\} \]
\[ E = \{1-2, 1-3, 2-3, 2-4, 2-5, 3-5, 3-7, 3-8, 4-5, 5-6\} \]
\( n = 8 \)
\( m = 11 \)

Some Graph Applications

<table>
<thead>
<tr>
<th>Graph</th>
<th>Nodes</th>
<th>Edges</th>
</tr>
</thead>
<tbody>
<tr>
<td>transportation</td>
<td>street intersection, highway</td>
<td></td>
</tr>
<tr>
<td>communication</td>
<td>computer, fiber optic cables</td>
<td></td>
</tr>
<tr>
<td>World Wide Web</td>
<td>web page, hyperlink</td>
<td></td>
</tr>
<tr>
<td>social</td>
<td>people, relationship</td>
<td></td>
</tr>
<tr>
<td>food web</td>
<td>species, predator-prey</td>
<td></td>
</tr>
<tr>
<td>software systems</td>
<td>function, function calls</td>
<td></td>
</tr>
<tr>
<td>scheduling</td>
<td>topic, precedence constraint</td>
<td></td>
</tr>
<tr>
<td>circuits</td>
<td>gates, wires</td>
<td></td>
</tr>
</tbody>
</table>

World Wide Web

Web graph.
- Node: web page.
- Edge: hyperlink from one page to another.

Ecological Food Web

Food web graph.
- Node: species.
- Edge: from prey to predator.

9-11 Terrorist Network

Social network graph.
- Node: people.
- Edge: relationship between two people.

Graph Representation: Adjacency Matrix

Adjacency matrix. \( n \times n \) matrix with \( A_{uv} = 1 \) if \((u, v)\) is an edge.
- Two representations of each edge.
- Space proportional to \( n^2 \).
- Checking if \((u, v)\) is an edge takes \( \Theta(1) \) time.
- Identifying all edges takes \( \Theta(n^2) \) time.

\[
\begin{pmatrix}
1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\end{pmatrix}
\]
Graph Representation: Adjacency List

- **Adjacency list**: Node indexed array of lists.
  - Two representations of each edge.
  - Space proportional to m + n.
  - Checking if \((u, v)\) is an edge takes \(O(deg(u))\) time.
  - Identifying all edges takes \(\Theta(m + n)\) time.

Paths and Connectivity

- **Path**: In an undirected graph \(G = (V, E)\), a path \(P\) is a sequence \(v_1, v_2, \ldots, v_k\) of nodes with the property that each consecutive pair \(v_i, v_{i+1}\) is joined by an edge in \(E\).
- **Simple path**: A path is simple if all nodes are distinct.

Cycles

- **Cycle**: A cycle is a path \(v_1, v_2, \ldots, v_k-1, v_k\) in which \(v_1 = v_k\), \(k > 2\), and the first \(k-1\) nodes are all distinct.

Trees

- **Tree**: An undirected graph is a tree if it is connected and does not contain a cycle.

**Theorem**: Let \(G\) be an undirected graph on \(n\) nodes. Any two of the following statements imply the third:
  - \(G\) is connected.
  - \(G\) does not contain a cycle.
  - \(G\) has \(n-1\) edges.

Rooted Trees

- **Rooted tree**: Given a tree \(T\), choose a root node \(r\) and orient each edge away from \(r\).

**Importance**: Models hierarchical structure.

Phylogeny Trees

- **Phylogeny tree**: Describe evolutionary history of species.
**Binary Tree**

Def. A rooted tree in which each node has at most 2 children.

Def. Height of a tree is the number of edges in the longest path from root to leaf.

Then. Number of nodes in binary tree of height \( h \) is \( \leq 2^{h+1} - 1 \).

Balanced Binary Tree. Height \( h = \Theta(\log n) \).

**3.2 Graph Traversal**

**Breadth First Search**

BFS intuition: Explore outward from \( s \) in all possible directions, adding nodes one "layer" at a time.

### BFS algorithm:
- \( L_0 = \{ s \} \)
- \( L_1 = \) all neighbors of \( L_0 \)
- \( L_2 = \) all nodes that do not belong to \( L_0 \) or \( L_1 \) and that have an edge to a node in \( L_1 \)
- \( L_3 = \) all nodes that do not belong to an earlier layer, and that have an edge to a node in \( L_2 \)

Theorem. For each \( i \), \( L_i \) consists of all nodes at distance exactly \( i \) from \( s \). There is a path from \( s \) to \( t \) iff \( t \) appears in some layer.

**GUI Containment Hierarchy**

**GUI containment hierarchy.** Describes organization of GUI widgets.

**Connectivity**

**s-t connectivity problem.** Given two node \( s \) and \( t \), is there a path between \( s \) and \( t \)?

**s-t shortest path problem.** Given two node \( s \) and \( t \), what is the length of the shortest path between \( s \) and \( t \)?

Applications:
- Navigation (Google Maps)
- Maze traversal
- Kevin Bacon number
- Fewest number of hops in a communication network
Property. Let $T$ be a BFS tree of $G = (V, E)$, and let $(x, y)$ be an edge of $G$. Then the level of $x$ and $y$ differ by at most 1.

**Theorem.** The above implementation of BFS runs in $O(m + n)$ time if the graph is given by its adjacency representation.

**Proof.**
- Easy to prove $O(n^2)$ running time:
  - at most $n$ lists $L[i]$
  - each node occurs on at most one list; for loop runs $\leq n$ times
  - when we consider node $u$, there are $\leq n$ incident edges $(u, v)$, and we spend $O(1)$ processing each edge
- Actually runs in $O(m + n)$ time:
  - when we consider node $u$, there are $\deg(u)$ incident edges $(u, v)$
  - total time processing edges is $\sum_{u \in V} \deg(u) = 2m$

**Flood Fill.** Given a lime green pixel in an image, change color of entire blob of neighboring lime pixels to blue.
- Node: pixel.
- Edge: two neighboring lime pixels.
- Blob: connected component of lime pixels.

**Flood Fill**
- Given lime green pixel in an image, change color of entire blob of neighboring lime pixels to blue.
- Node: pixel.
- Edge: two neighboring lime pixels.
- Blob: connected component of lime pixels.

**Theorem.** Upon termination, $R$ is the connected component containing $s$.
- BFS = explore in order of distance from $s$.
- DFS = explore in a different way.
3.4 Testing Bipartiteness

Testing Bipartiteness

Given a graph G, is it bipartite?

- Many graph problems become:
  - easier if the underlying graph is bipartite (matching)
  - tractable if the underlying graph is bipartite (independent set)
- Before attempting to design an algorithm, we need to understand structure of bipartite graphs.

An Obstruction to Bipartiteness

Lemma. If a graph G is bipartite, it cannot contain an odd length cycle.

Proof. Not possible to 2-color the odd cycle, let alone G.

Bipartite Graphs

Def. An undirected graph G = (V, E) is bipartite if the nodes can be colored red or blue such that every edge has one red and one blue end.

Applications.
- Stable marriage: men = red, women = blue.
- Scheduling: machines = red, jobs = blue.

Bipartite Graphs

Lemma. Let G be a connected graph, and let L0, …, Lk be the layers produced by BFS starting at node s. Exactly one of the following holds.
(i) No edge of G joins two nodes of the same layer, and G is bipartite.
(ii) An edge of G joins two nodes of the same layer, and G contains an odd-length cycle (and hence is not bipartite).

Case (i)

Case (ii)
Bipartite Graphs

Lemma. Let G be a connected graph, and let L0, L1, ..., Lk be the layers produced by BFS starting at node s. Exactly one of the following holds.

(i) No edge of G joins two nodes of the same layer, and G is bipartite.
(ii) An edge of G joins two nodes of the same layer, and G contains an odd-length cycle (and hence is not bipartite).

Pf. (ii)

- Suppose (x, y) is an edge with x, y in same level Lj.
- Let z = lca(x, y) = lowest common ancestor.
- Let Lk be level containing z.
- Consider cycle that takes edge from x to y, then path from y to z, then path from z to x.
- Its length is 1 + (j-k) + (j-k), which is odd.

Obstruction to Bipartiteness

Corollary. A graph G is bipartite iff it contains no odd length cycle.

Directed Graphs

Directed graph. G = (V, E)

- Edge (u, v) goes from node u to node v.

Directed s-t shortest path problem. Given two node s and t, what is the length of the shortest path between s and t?

Graph search. BFS extends naturally to directed graphs.

Web crawler. Start from web page s. Find all web pages linked from s, either directly or indirectly.

Strong Connectivity

Def. Node u and v are mutually reachable if there is a path from u to v and also a path from v to u.

Def. A graph is strongly connected if every pair of nodes is mutually reachable.

Lemma. Let s be any node. G is strongly connected iff every node is reachable from s, and s is reachable from every node.

Pf. ⇒ Follows from definition.

Pf. ⇐ Path from u to v: concatenate u-s path with s-v path.

Directed reachability. Given a node s, find all nodes reachable from s.

Directed s-t shortest path problem. Given two node s and t, what is the length of the shortest path between s and t?
Strong Connectivity: Algorithm

**Theorem.** Can determine if \( G \) is strongly connected in \( O(m + n) \) time.

**Proof.**
- Pick any node \( s \).
- Run BFS from \( s \) in \( G \).
- Run BFS from \( s \) in \( G^{rev} \).
- Return true iff all nodes reached in both BFS executions.
- Correctness follows immediately from previous lemma.

Directed Acyclic Graphs

**Def.** A DAG is a directed graph that contains no directed cycles.

**Ex.** Precedence constraints: (\( v_i, v_j \)) means \( v_i \) must precede \( v_j \).

**Def.** A topological order of a directed graph \( G = (V, E) \) is an ordering of its nodes as \( v_1, v_2, \ldots, v_n \) so that for every edge (\( v_i, v_j \)) we have \( i < j \).

**Lemma.** If \( G \) has a topological order, then \( G \) is a DAG.

**Proof.** (by contradiction)
- Suppose that \( G \) has a topological order \( v_1, \ldots, v_n \) and that \( G \) also has a directed cycle \( C \). Let’s see what happens.
- Let \( v_i \) be the lowest-indexed node in \( C \), and let \( v_j \) be the node just before \( v_i \); thus \( (v_j, v_i) \) is an edge.
- By our choice of \( i \), we have \( i < j \).
- On the other hand, since \( (v_j, v_i) \) is an edge and \( v_1, \ldots, v_n \) is a topological order, we must have \( j < i \), a contradiction.

Directed Acyclic Graphs

**Lemma.** If \( G \) has a topological order, then \( G \) is a DAG.

**Q.** Does every DAG have a topological ordering?
**Q.** If so, how do we compute one?

3.6 DAGs and Topological Ordering

**Precedence Constraints**

**Def.** (\( v_i, v_j \)) means task \( v_i \) must occur before \( v_j \).

**Applications.**
- Course prerequisite graph: course \( v_i \) must be taken before \( v_j \).
- Compilation: module \( v_i \) must be compiled before \( v_j \). Pipeline of computing jobs: output of job \( v_i \) needed to determine input of job \( v_j \).
Directed Acyclic Graphs

**Lemma.** If $G$ is a DAG, then $G$ has a node with no incoming edges.

**Proof (by contradiction):**
- Suppose that $G$ is a DAG and every node has at least one incoming edge. Let's see what happens.
- Pick any node $v$, and begin following edges backward from $v$. Since $v$ has at least one incoming edge $(u, v)$, we can walk backward to $u$.
- Then, since $u$ has at least one incoming edge $(x, u)$, we can walk backward to $x$.
- Repeat until we visit a node, say $w$, twice.
- Let $C$ denote the sequence of nodes encountered between successive visits to $w$. $C$ is a cycle.

\[ \text{\includegraphics{dag}} \]

Directed Acyclic Graphs

**Lemma.** If $G$ is a DAG, then $G$ has a topological ordering.

**Proof (by induction on $n$):**
- Base case: true if $n = 1$.
- Given DAG on $n+1$ nodes, find a node $v$ with no incoming edges.
- $G - \{v\}$ is a DAG, since deleting $v$ cannot create cycles.
- By inductive hypothesis, $G - \{v\}$ has a topological ordering.
- Place $v$ first in topological ordering; then append nodes of $G - \{v\}$ in topological order. This is valid since $v$ has no incoming edges.

\[ \text{\includegraphics{topo}} \]

**Theorem.** Algorithm finds a topological order in $O(m + n)$ time.

**Proof:**
- Maintain the following information:
  - $	ext{count}(v)$: remaining number of incoming edges
  - $S$: set of remaining nodes with no incoming edges
- Initialization: $O(m + n)$ via single scan through graph.
- Update: to delete $v$
  - remove $v$ from $S$
  - decrement $	ext{count}(w)$ for all edges from $v$ to $w$, and add $w$ to $S$ if $	ext{count}(w)$ hits 0
  - this is $O(1)$ per edge.