算法设计与分析(2024年春季学期)
Final Review

授课老师: 栗师
南京大学计算机科学与技术系
Introduction
What is an Algorithm?

- **Donald Knuth**: An algorithm is a finite, definite effective procedure, with some input and some output.

- **Algorithm**: “abstract”, can be specified using computer program, English, pseudo-codes or flow charts.

- **Computer program**: “concrete”, implementation of algorithm, using a particular programming language
Def. \( f : \mathbb{N} \rightarrow \mathbb{R} \) is an asymptotically positive function if:
- \( \exists n_0 > 0 \) such that \( \forall n > n_0 \) we have \( f(n) > 0 \)

\( O \)-Notation For a function \( g(n) \),
\[
O(g(n)) = \{ \text{function } f : \exists c > 0, n_0 > 0 \text{ such that } f(n) \leq cg(n), \forall n \geq n_0 \}.
\]

- We use “ \( f(n) = O(g(n)) \)” to denote “ \( f(n) \in O(g(n)) \)”
**Ω-Notation**  For a function $g(n)$,

$$\Omega(g(n)) = \{\text{function } f : \exists c > 0, n_0 > 0 \text{ such that } f(n) \geq cg(n), \forall n \geq n_0\}.$$
**Θ-Notation** For a function $g(n)$,

$$\Theta(g(n)) = \{ \text{function } f : \exists c_2 \geq c_1 > 0, n_0 > 0 \text{ such that } c_1 g(n) \leq f(n) \leq c_2 g(n), \forall n \geq n_0 \}. $$
$o$-Notation  For a function $g(n)$,

$$o(g(n)) = \{ \text{function } f : \forall c > 0, \exists n_0 > 0 \text{ such that } f(n) \leq cg(n), \forall n \geq n_0 \}.$$ 

$\omega$-Notation  For a function $g(n)$,

$$\omega(g(n)) = \{ \text{function } f : \forall c > 0, \exists n_0 > 0 \text{ such that } f(n) \geq cg(n), \forall n \geq n_0 \}.$$
Running Times

- **Linear time**: compute the summation/max/min of an array of $n$ elements.
- $O(n \log n)$ Running Time: merge-sort, counting-inversion, D&C algorithms with recurrence $T(n) = 2T(n/2) + O(n)$.
- $O(n^2)$ (Quadratic) time: enumerating pairs of elements in an array of size $n$.
- $O(n^3)$ (Cubic) time: naive algorithm for computing matrix multiplication.
- Beyond polynomial time: $2^{O(n)}$ and $O(n!)$.
Outline

2. Graph Basics
   - Connectivity and Graph Traversal
   - Topological Ordering
   - Finding Bridges
**Adjacency matrix**
- $n \times n$ matrix, $A[u, v] = 1$ if $(u, v) \in E$ and $A[u, v] = 0$ otherwise
- $A$ is symmetric if graph is undirected

**Linked lists**
- For every vertex $v$, there is a linked list containing all neighbours of $v$. 

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Representation of Graphs

- **Adjacency matrix**
  - $n \times n$ matrix, $A[u, v] = 1$ if $(u, v) \in E$ and $A[u, v] = 0$ otherwise
  - $A$ is symmetric if graph is undirected

- **Linked lists**
  - For every vertex $v$, there is a linked list containing all neighbours of $v.$
Comparison of Two Representations

- Assuming we are dealing with undirected graphs
- \( n \): number of vertices
- \( m \): number of edges, assuming \( n - 1 \leq m \leq n(n - 1)/2 \)
- \( d_v \): number of neighbors of \( v \)

<table>
<thead>
<tr>
<th></th>
<th>Matrix</th>
<th>Linked Lists</th>
</tr>
</thead>
<tbody>
<tr>
<td>memory usage</td>
<td>( O(n^2) )</td>
<td>( O(m) )</td>
</tr>
<tr>
<td>time to check ((u, v) \in E)</td>
<td>( O(1) )</td>
<td>( O(d_u) )</td>
</tr>
<tr>
<td>time to list all neighbours of ( v )</td>
<td>( O(n) )</td>
<td>( O(d_v) )</td>
</tr>
</tbody>
</table>
Outline

2. Graph Basics
   - Connectivity and Graph Traversal
   - Topological Ordering
   - Finding Bridges
Connectivity Problem

**Input:** graph $G = (V, E)$, (using linked lists)
  two vertices $s, t \in V$

**Output:** whether there is a path connecting $s$ to $t$ in $G$

- Algorithm: starting from $s$, search for all vertices that are reachable from $s$ and check if the set contains $t$
  - Breadth-First Search (BFS)
  - Depth-First Search (DFS)
Implementing BFS using a Queue

**BFS(s)**

1. $\textit{head} \leftarrow 1$, $\textit{tail} \leftarrow 1$, $\text{queue}[1] \leftarrow s$
2. mark $s$ as “visited” and all other vertices as “unvisited”
3. $\textbf{while} \ \textit{head} \leq \textit{tail} \ \textbf{do}$
4. $\quad v \leftarrow \text{queue}[\textit{head}]$, $\textit{head} \leftarrow \textit{head} + 1$
5. $\quad \textbf{for} \ \text{all neighbours } u \ \text{of} \ v \ \textbf{do}$
6. $\quad \quad \textbf{if} \ u \ \text{is “unvisited”} \ \textbf{then}$
7. $\quad \quad \quad \textit{tail} \leftarrow \textit{tail} + 1$, $\text{queue}[\textit{tail}] = u$
8. $\quad \quad \text{mark } u \ \text{as “visited”}$

- Running time: $O(n + m)$. 
Depth-First Search (DFS)

- Starting from \( s \)
- Travel through the first edge leading out of the current vertex
- When reach an already-visited vertex ("dead-end"), go back
- Travel through the next edge
- If tried all edges leading out of the current vertex, go back
Depth-First Search (DFS)

- Starting from \( s \)
- Travel through the first edge leading out of the current vertex
- When reach an already-visited vertex ("dead-end"), go back
- Travel through the next edge
- If tried all edges leading out of the current vertex, go back
Depth-First Search (DFS)

- Starting from \( s \)
- Travel through the first edge leading out of the current vertex
- When reach an already-visited vertex (“dead-end”), go back
- Travel through the next edge
- If tried all edges leading out of the current vertex, go back
Depth-First Search (DFS)

- Starting from \( s \)
- Travel through the first edge leading out of the current vertex
- When reach an already-visited vertex ("dead-end"), go back
- Travel through the next edge
- If tried all edges leading out of the current vertex, go back
Depth-First Search (DFS)

- Starting from \( s \)
- Travel through the first edge leading out of the current vertex
- When reach an already-visited vertex ("dead-end"), go back
- Travel through the next edge
- If tried all edges leading out of the current vertex, go back
Depth-First Search (DFS)

- Starting from \( s \)
- Travel through the first edge leading out of the current vertex
- When reach an already-visited vertex ("dead-end"), go back
- Travel through the next edge
- If tried all edges leading out of the current vertex, go back
Depth-First Search (DFS)

- Starting from $s$
- Travel through the first edge leading out of the current vertex
- When reach an already-visited vertex ("dead-end"), go back
- Travel through the next edge
- If tried all edges leading out of the current vertex, go back
Depth-First Search (DFS)

- Starting from \( s \)
- Travel through the first edge leading out of the current vertex
- When reach an already-visited vertex ("dead-end"), go back
- Travel through the next edge
- If tried all edges leading out of the current vertex, go back
Depth-First Search (DFS)

- Starting from \( s \)
- Travel through the first edge leading out of the current vertex
- When reach an already-visited vertex ("dead-end"), go back
- Travel through the next edge
- If tried all edges leading out of the current vertex, go back
**Depth-First Search (DFS)**

- Starting from \( s \)
- Travel through the first edge leading out of the current vertex
- When reach an already-visited vertex ("dead-end"), go back
- Travel through the next edge
- If tried all edges leading out of the current vertex, go back
Depth-First Search (DFS)

- Starting from \( s \)
- Travel through the first edge leading out of the current vertex
- When reach an already-visited vertex ("dead-end"), go back
- Travel through the next edge
- If tried all edges leading out of the current vertex, go back
Implementing DFS using Recursion

**DFS(s)**

1. mark all vertices as “unvisited”
2. recursive-DFS(s)

**recursive-DFS(v)**

1. mark v as “visited”
2. for all neighbours u of v do
3. if u is unvisited then recursive-DFS(u)
Def. A graph $G = (V, E)$ is a bipartite graph if there is a partition of $V$ into two sets $L$ and $R$ such that for every edge $(u, v) \in E$, we have either $u \in L, v \in R$ or $v \in L, u \in R$. 
Test Bipartiteness
Test Bipartiteness
Test Bipartiteness
Test Bipartiteness
Test Bipartiteness
Test Bipartiteness
Test Bipartiteness
Test Bipartiteness
Test Bipartiteness
Test Bipartiteness
Test Bipartiteness

bad edges!
Testing Bipartiteness using BFS

\textbf{BFS}(s)

1: $\text{head} \leftarrow 1, \text{tail} \leftarrow 1, \text{queue}[1] \leftarrow s$
2: mark $s$ as “visited” and all other vertices as “unvisited”
3: $\text{color}[s] \leftarrow 0$
4: \textbf{while} $\text{head} \leq \text{tail}$ \textbf{do}
5: \hspace{1em} $v \leftarrow \text{queue}[\text{head}], \text{head} \leftarrow \text{head} + 1$
6: \hspace{1em} \textbf{for all neighbours} $u$ of $v$ \textbf{do}
7: \hspace{2em} \textbf{if} $u$ is “unvisited” \textbf{then}
8: \hspace{3em} $\text{tail} \leftarrow \text{tail} + 1, \text{queue}[\text{tail}] = u$
9: \hspace{3em} mark $u$ as “visited”
10: \hspace{1em} $\text{color}[u] \leftarrow 1 - \text{color}[v]$
11: \hspace{1em} \textbf{else if} $\text{color}[u] = \text{color}[v]$ \textbf{then}
12: \hspace{2em} print(“$G$ is not bipartite”) and exit
Testing Bipartiteness using BFS

1: mark all vertices as “unvisited”
2: for each vertex $v \in V$ do
3: if $v$ is “unvisited” then
4: test-bipartiteness($v$)
5: print(“$G$ is bipartite”)

Obs. Running time of algorithm = $O(n + m)$
Outline

2 Graph Basics
- Connectivity and Graph Traversal
- Topological Ordering
- Finding Bridges
Topological Ordering Problem

**Input:** a directed acyclic graph (DAG) \( G = (V, E) \)

**Output:** 1-to-1 function \( \pi : V \rightarrow \{1, 2, 3 \ldots , n\} \), so that

- if \((u, v) \in E\) then \(\pi(u) < \pi(v)\)
Topological Ordering Problem

**Input:** a directed acyclic graph (DAG) $G = (V, E)$

**Output:** 1-to-1 function $\pi : V \rightarrow \{1, 2, 3 \cdots, n\}$, so that

- if $(u, v) \in E$ then $\pi(u) < \pi(v)$
Topological Ordering

- Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.
Topological Ordering

- Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.
Topological Ordering

- Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.
Topological Ordering

- Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.
Topological Ordering

- Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.
Topological Ordering

- Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.
Topological Ordering

Q: How to make the algorithm as efficient as possible?

A:
- Use linked-lists of outgoing edges
- Maintain the in-degree $d_v$ of vertices
- Maintain a queue (or stack) of vertices $v$ with $d_v = 0$
topological-sort($G$)

1: let $d_v \leftarrow 0$ for every $v \in V$
2: for every $v \in V$ do
3: for every $u$ such that $(v, u) \in E$ do
4: $d_u \leftarrow d_u + 1$
5: $S \leftarrow \{v : d_v = 0\}, i \leftarrow 0$
6: while $S \neq \emptyset$ do
7: $v \leftarrow$ arbitrary vertex in $S$, $S \leftarrow S \setminus \{v\}$
8: $i \leftarrow i + 1$, $\pi(v) \leftarrow i$
9: for every $u$ such that $(v, u) \in E$ do
10: $d_u \leftarrow d_u - 1$
11: if $d_u = 0$ then add $u$ to $S$
12: if $i < n$ then output “not a DAG”

- $S$ can be represented using a queue or a stack
- Running time $= O(n + m)$
Outline

Graph Basics
- Connectivity and Graph Traversal
- Topological Ordering
- Finding Bridges
**Vertical and Horizontal Edges**

- $G = (V, E)$: connected graph
- $T = (V, E_T)$: rooted spanning tree of $G$
- $(u, v) \in E \setminus E_T$ is
  - **vertical** if one of $u$ and $v$ is an ancestor of the other in $T$,
  - **horizontal** otherwise.
- $G = (V, E)$: connected graph

$T$: a DFS tree for $G$

**Q:** Can there be a horizontal edges $(u, v)$ w.r.t $T$?

**A:** No!
$G = (V, E)$: connected graph

$T$: a DFS tree for $G$

$G$ contains only tree and vertical edges
\[ G = (V, E): \text{connected graph} \]

\[ T: \text{a DFS tree for } G \]

\[ G \text{ contains only tree and vertical edges} \]

\[ \text{vertical edges: not bridges} \]
- $G = (V, E)$: connected graph
- $T$: a DFS tree for $G$
- $G$ contains only tree and vertical edges
- vertical edges: not bridges

**Lemma**
- $(u, v) \in T$, $u$ is parent
- $(u, v)$ is not a bridge $\iff \exists$ vertical edge connecting an (inclusive) descendant of $v$ and an (inclusive) ancestor of $u$
- $G = (V, E)$: connected graph
- $T$: a DFS tree for $G$
- $G$ contains only tree and vertical edges
- vertical edges: not bridges

**Lemma**

- $(u, v) \in T$, $u$ is parent
- $(u, v)$ is not a bridge $\iff \exists$ vertical edge connecting an (inclusive) descendant of $v$ and an (inclusive) ancestor of $u$
Outline

3 Greedy Algorithms

- Interval Scheduling
- Scheduling to Minimize Lateness
- Weighted Completion Time Scheduling
- Offline Caching
- Data Compression and Huffman Code
Greedy Algorithm
- Build up the solutions in steps
- At each step, make an **irrevocable** decision using a “reasonable” strategy

A Common Way to Analyze Greedy Algorithms
- Prove that the reasonable strategy is “safe” (**key**)
- Show that the remaining task after applying the strategy is to solve a (many) smaller instance(s) of the same problem (**usually easy**)

**Def.** A strategy is safe: there is always an optimum solution that agrees with the decision made according to the strategy.
Generic Greedy Algorithm

1: while the instance is non-trivial do
2: make the choice using the greedy strategy
3: reduce the instance

Exchange argument: Proof of Safety of a Strategy

- let $S$ be an arbitrary optimum solution.
- if $S$ is consistent with the greedy choice, done.
- otherwise, show that it can be modified to another optimum solution $S'$ that is consistent with the choice.

- The procedure is not a part of the algorithm.
Greedy Algorithms

- Interval Scheduling
  - Scheduling to Minimize Lateness
  - Weighted Completion Time Scheduling
  - Offline Caching
  - Data Compression and Huffman Code
Interval Scheduling

**Input:** \( n \) jobs, job \( i \) with start time \( s_i \) and finish time \( f_i \)

\( i \) and \( j \) are **compatible** if \([s_i, f_i)\) and \([s_j, f_j)\) are disjoint

**Output:** A maximum-size subset of mutually compatible jobs
**Lemma** It is safe to schedule the job $j$ with the earliest finish time: There is an optimum solution where the job $j$ with the earliest finish time is scheduled.

**Proof.**
- Take an arbitrary optimum solution $S$
- If it contains $j$, done
- Otherwise, replace the first job in $S$ with $j$ to obtain another optimum schedule $S'$.
Greedy Algorithm for Interval Scheduling

**Lemma** It is safe to schedule the job \( j \) with the earliest finish time: There is an optimum solution where the job \( j \) with the earliest finish time is scheduled.

- What is the remaining task after we decided to schedule \( j \)?
- Is it another instance of interval scheduling problem? Yes!

![Diagram showing intervals and the greedy algorithm for scheduling jobs]
Outline

3 Greedy Algorithms
- Interval Scheduling
- Scheduling to Minimize Lateness
- Weighted Completion Time Scheduling
- Offline Caching
- Data Compression and Huffman Code
Scheduling to minimize lateness

**Input:** $n$ jobs, each job $j \in [n]$ with a processing time $p_j$ and deadline $d_j$

**Output:** schedule jobs on 1 machine, to minimize the max. lateness

$C_j$: completion time of $j$ 

$lateness l_j := \max\{C_j - d_j, 0\}$

### Example input:

<table>
<thead>
<tr>
<th>$j$</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p_j$</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>$d_j$</td>
<td>5</td>
<td>7</td>
<td>4</td>
<td>8</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

solution 1

- [a, b, c, d]

solution 2

- [c, a, b, d]

- solution 1: max lateness = $\max\{0, 3 - 5, 6 - 7, 8 - 4, 9 - 8\} = 4$
- solution 2: max lateness = $\max\{0, 2 - 4, 5 - 5, 8 - 7, 9 - 8\} = 1$
- solution 2 is better
Lemma  The ascending order of deadlines $d_j$ (the Earliest Deadline First order or the EDF order) is the optimum schedule.
• maximum lateness = \( \max \left\{ 0, \max_{j \in [n]} \{ C_j - d_j \} \right\} \).

\[
\begin{align*}
\text{before:} & \quad \max \{ t + p_j - d_j, t + p_j + p_{j'} - d_{j'} \} = t + p_j + p_{j'} - d_{j'} \\
\text{after:} & \quad \max \{ t + p_{j'} - d_{j'}, t + p_j + p_{j'} - d_j \} \\
& \quad p_{j'} - d_{j'} < p_j + p_{j'} - d_{j'} \quad \text{and} \quad p_j + p_{j'} - d_j < p_j + p_{j'} - d_{j'} \\
& \quad \max \{ t + p_{j'} - d_{j'}, t + p_j + p_{j'} - d_j \} < t + p_j + p_{j'} - d_{j'} \\
& \quad \text{after swapping, the maximum of the two terms strictly decreases}
\end{align*}
\]
Outline

3 Greedy Algorithms
- Interval Scheduling
- Scheduling to Minimize Lateness
- Weighted Completion Time Scheduling
- Offline Caching
- Data Compression and Huffman Code
Scheduling to Minimize Weighted Completion Time

**Input:** A set of \(n\) jobs \([n] := \{1, 2, 3, \cdots, n\}\)
each job \(j\) has a weight \(w_j\) and processing time \(p_j\)

**Output:** an ordering of jobs so as to minimize the total weighted completion time of jobs

\[
\begin{align*}
&pa = 1 \quad pb = 2 \quad pc = 3 \\
&\begin{array}{c}
a \\
w_1 = 2 \\
\end{array} \\
&\begin{array}{c}
b \\
w_2 = 5 \\
\end{array} \\
&\begin{array}{c}
c \\
w_3 = 7 \\
\end{array}
\end{align*}
\]

\[
\begin{align*}
wa & = 2 \quad wb & = 5 \quad wc & = 7 \\
\begin{array}{c}
a \\
0 \\
\end{array} & \\
\begin{array}{c}
b \\
1 \\
\end{array} & \\
\begin{array}{c}
c \\
2 \\
\end{array} & \\
0 & 1 & 2 & 3 & 4 & 5 & 6
\end{align*}
\]

\[
\begin{align*}
\text{cost} & = 2 \times 1 + 5 \times 3 + 7 \times 6 = 59 \\
\begin{array}{c}
b \\
0 \\
\end{array} & \\
\begin{array}{c}
c \\
2 \\
\end{array} & \\
\begin{array}{c}
a \\
5 \\
\end{array} & \\
0 & 1 & 2 & 3 & 4 & 5 & 6
\end{align*}
\]

\[
\begin{align*}
\text{cost} & = 5 \times 2 + 7 \times 5 + 2 \times 6 = 57
\end{align*}
\]
Def. The Smith ratio of a job is $w_j/p_j$.

Lemma The descending order of Smith ratios (the Smith rule) is optimum.
Outline

3 Greedy Algorithms
- Interval Scheduling
- Scheduling to Minimize Lateness
- Weighted Completion Time Scheduling
- Offline Caching
- Data Compression and Huffman Code
Offline Caching

- Cache that can store $k$ pages
- Sequence of page requests
- Cache miss happens if requested page not in cache. We need bring the page into cache, and evict some existing page if necessary.
- Cache hit happens if requested page already in cache.
- Goal: minimize the number of cache misses.

```
page sequence           cache
1                      ✔
5                      ❌  ✔  ❌
4                      ❌  ✔  ✔  ✔
2                      ❌  ✔  ✔  ✔
5                      ❌  ✔  ✔  ✔
3                      ❌  ✔  ✔  ✔
2                      ✔
1                      ✔
misses = 6
```
Optimum Offline Caching

**Furthest-in-Future (FF)**

- Algorithm: every time, evict the page that is not requested until furthest in the future, if we need to evict one.
- The algorithm is **not** an online algorithm, since the decision at a step depends on the request sequence in the future.
Q: How can we make the algorithm as fast as possible?

A:
- The running time can be made to be $O(n + T \log k)$.
- For each page $p$, use a linked list (or an array with dynamic size) to store the time steps in which $p$ is requested.
- We can find the next time a page is requested easily.
- Use a priority queue data structure to hold all the pages in cache, so that we can easily find the page that is requested furthest in the future.
Outline

3 Greedy Algorithms
- Interval Scheduling
- Scheduling to Minimize Lateness
- Weighted Completion Time Scheduling
- Offline Caching
- Data Compression and Huffman Code
Encoding Letters Using Bits

- 8 letters $a, b, c, d, e, f, g, h$ in a language
- need to encode a message using bits
- idea: use 3 bits per letter

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
<th>g</th>
<th>h</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>001</td>
<td>010</td>
<td>011</td>
<td>100</td>
<td>101</td>
<td>110</td>
<td>111</td>
</tr>
</tbody>
</table>

$deacfg \rightarrow 011100000010101110$

Q: Can we have a better encoding scheme?

- Seems unlikely: must use 3 bits per letter

Q: What if some letters appear more frequently than the others?
Q: If some letters appear more frequently than the others, can we have a better encoding scheme?

A: Using variable-length encoding scheme might be more efficient.

Idea
- using fewer bits for letters that are more frequently used, and more bits for letters that are less frequently used.
**Prefix Codes**

**Def.** A prefix code for a set $S$ of letters is a function $\gamma : S \to \{0, 1\}^*$ such that for two distinct $x, y \in S$, $\gamma(x)$ is not a prefix of $\gamma(y)$.

<table>
<thead>
<tr>
<th></th>
<th>$a$</th>
<th>$b$</th>
<th>$c$</th>
<th>$d$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\gamma(a)$</td>
<td>001</td>
<td>0000</td>
<td>0001</td>
<td>100</td>
</tr>
<tr>
<td>$\gamma(e)$</td>
<td>11</td>
<td>1010</td>
<td>1011</td>
<td>01</td>
</tr>
</tbody>
</table>

![Prefix Code Tree]

The table above and the diagram show an example of a prefix code for the set $S = \{a, b, c, d, e, f, g, h\}$.
### Example

<table>
<thead>
<tr>
<th>letters</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
</tr>
</thead>
<tbody>
<tr>
<td>frequencies</td>
<td>18</td>
<td>3</td>
<td>4</td>
<td>6</td>
<td>10</td>
</tr>
<tr>
<td>scheme 1 length</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>scheme 2 length</td>
<td>1</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>scheme 3 length</td>
<td>1</td>
<td>4</td>
<td>4</td>
<td>3</td>
<td>2</td>
</tr>
</tbody>
</table>

**Scheme 1**
```
(a) -- (d) -- (e)
  |   |   |
  b   c
```

**Scheme 2**
```
(a) -- (b) -- (c) -- (d) -- (e)
  |   |   |   |
  a   b   c   d   e
```

**Scheme 3**
```
(a) -- (e) -- (d) -- (b) -- (c)
  |   |   |   |   |
  a   e   d   b   c
```
Which Two Letters Can Be Safely Put Together As Brothers?

- Focus on the “structure” of the optimum encoding tree
- There are two deepest leaves that are brothers

**Lemma** It is safe to make the two least frequent letters brothers.
- $f_x$: the frequency of the letter $x$ in the support.
- $x_1$ and $x_2$: the two letters we decided to put together.
- $d_x$ the depth of letter $x$ in our output encoding tree.

\[ \sum_{x \in S} f_x d_x \]
\[ = \sum_{x \in S \setminus \{x_1, x_2\}} f_x d_x + f_{x_1} d_{x_1} + f_{x_2} d_{x_2} \]
\[ = \sum_{x \in S \setminus \{x_1, x_2\}} f_x d_x + (f_{x_1} + f_{x_2}) d_{x_1} \]
\[ = \sum_{x \in S \setminus \{x_1, x_2\}} f_x d_x + f' (d_x + 1) \]
\[ = \sum_{x \in S \setminus \{x_1, x_2\} \cup \{x'\}} f_x d_x + f_x' \]

Def: $f' = f_{x_1} + f_{x_2}$
In order to minimize
\[ \sum_{x \in S} f_x d_x, \]
we need to minimize
\[ \sum_{x \in S \setminus \{x_1, x_2\} \cup \{x'\}} f_x d_x, \]
subject to that \( d \) is the depth function for an encoding tree of \( S \setminus \{x_1, x_2\} \).

This is exactly the best prefix codes problem, with letters \( S \setminus \{x_1, x_2\} \cup \{x'\} \) and frequency vector \( f \)!
Outline

4 Divide-and-Conquer

- Counting Inversions
- Solving Recurrences
- Quicksort
- Lower Bound for Comparison-Based Sorting Algorithms
- Selection Problem
- Polynomial Multiplication
- Strassen’s Algorithm for Matrix Multiplication
- Finding Closest Pair of Points in 2D Euclidean Space
- Computing $n$-th Fibonacci Number
Divide-and-Conquer

- **Divide**: Divide instance into many smaller instances
- **Conquer**: Solve each of smaller instances recursively and separately
- **Combine**: Combine solutions to small instances to obtain a solution for the original big instance
Outline

4 Divide-and-Conquer
   • Counting Inversions
   • Solving Recurrences
   • Quicksort
   • Lower Bound for Comparison-Based Sorting Algorithms
   • Selection Problem
   • Polynomial Multiplication
   • Strassen’s Algorithm for Matrix Multiplication
   • Finding Closest Pair of Points in 2D Euclidean Space
   • Computing $n$-th Fibonacci Number
**Def.** Given an array $A$ of $n$ integers, an inversion in $A$ is a pair $(i, j)$ of indices such that $i < j$ and $A[i] > A[j]$.

**Counting Inversions**

**Input:** an sequence $A$ of $n$ numbers

**Output:** number of inversions in $A$
Count Inversions between $B$ and $C$

- Procedure that merges $B$ and $C$ and counts inversions between $B$ and $C$ at the same time

```plaintext
merge-and-count($B, C, n_1, n_2$)

1: count ← 0;
2: $A \leftarrow$ array of size $n_1 + n_2$; $i \leftarrow 1$; $j \leftarrow 1$
3: while $i \leq n_1$ or $j \leq n_2$ do
4: if $j > n_2$ or ($i \leq n_1$ and $B[i] \leq C[j]$) then
5: $A[i + j - 1] \leftarrow B[i]$; $i \leftarrow i + 1$
6: count ← count + ($j - 1$)
7: else
8: $A[i + j - 1] \leftarrow C[j]$; $j \leftarrow j + 1$
9: return ($A, count$)
```
A procedure that returns the sorted array of $A$ and counts the number of inversions in $A$:

**sort-and-count($A$, $n$)**

1. **if** $n = 1$ **then**
2. **return** ($A$, 0)
3. **else**
4. ($B$, $m_1$) $\leftarrow$ sort-and-count($A[1..\lfloor n/2 \rfloor], \lfloor n/2 \rfloor$)
5. ($C$, $m_2$) $\leftarrow$ sort-and-count($A[\lfloor n/2 \rfloor + 1..n], \lceil n/2 \rceil$)
6. ($A$, $m_3$) $\leftarrow$ merge-and-count($B$, $C$, $\lfloor n/2 \rfloor$, $\lceil n/2 \rceil$)
7. **return** ($A$, $m_1 + m_2 + m_3$)

- **Divide:** trivial
- **Conquer:** 4, 5
- **Combine:** 6, 7
sort-and-count\((A, n)\)

1: if \(n = 1\) then
2: return \((A, 0)\)
3: else
4: \((B, m_1) \leftarrow\) sort-and-count\((A[1..\lfloor n/2 \rfloor], \lfloor n/2 \rfloor)\)
5: \((C, m_2) \leftarrow\) sort-and-count\((A[\lceil n/2 \rceil + 1..n], \lceil n/2 \rceil)\)
6: \((A, m_3) \leftarrow\) merge-and-count\((B, C, \lfloor n/2 \rfloor, \lfloor n/2 \rfloor)\)
7: return \((A, m_1 + m_2 + m_3)\)

- Recurrence for the running time: \(T(n) = 2T(n/2) + O(n)\)
- Running time = \(O(n \log n)\)
Outline

4 Divide-and-Conquer
- Counting Inversions
- **Solving Recurrences**
- Quicksort
- Lower Bound for Comparison-Based Sorting Algorithms
- Selection Problem
- Polynomial Multiplication
- Strassen’s Algorithm for Matrix Multiplication
- Finding Closest Pair of Points in 2D Euclidean Space
- Computing $n$-th Fibonacci Number
Theorem $T(n) = aT(n/b) + O(n^c)$, where $a \geq 1$, $b > 1$, $c \geq 0$ are constants. Then,

$$T(n) = \begin{cases} 
O(n^{\log_b a}) & \text{if } c < \log_b a \\
O(n^c \log n) & \text{if } c = \log_b a \\
O(n^c) & \text{if } c > \log_b a
\end{cases}$$
Outline

4 Divide-and-Conquer
- Counting Inversions
- Solving Recurrences
- Quicksort
- Lower Bound for Comparison-Based Sorting Algorithms
- Selection Problem
- Polynomial Multiplication
- Strassen’s Algorithm for Matrix Multiplication
- Finding Closest Pair of Points in 2D Euclidean Space
- Computing $n$-th Fibonacci Number
## Quicksort vs Merge-Sort

<table>
<thead>
<tr>
<th>Divide</th>
<th>Merge Sort</th>
<th>Quicksort</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conquer</td>
<td>Trivial</td>
<td>Separate small and big numbers</td>
</tr>
<tr>
<td>Combine</td>
<td>Recurse</td>
<td>Recurse</td>
</tr>
<tr>
<td></td>
<td>Merge 2 sorted arrays</td>
<td>Trivial</td>
</tr>
</tbody>
</table>
def quicksort(A, n):
    1: if n ≤ 1 then return A
    2: x ← lower median of A
    3: AL ← array of elements in A that are less than x
    4: AR ← array of elements in A that are greater than x
    5: BL ← quicksort(AL, length of AL)
    6: BR ← quicksort(AR, length of AR)
    7: t ← number of times x appear A
    8: return concatenation of BL, t copies of x, and BR

- Recurrence \( T(n) \leq 2T(n/2) + O(n) \)
- Running time = \( O(n \log n) \)
**Assumption**  We can choose median of an array of size $n$ in $O(n)$ time.

**Q:** How to remove this assumption?

**A:**

1. There is an algorithm to find median in $O(n)$ time, using divide-and-conquer (we shall not talk about it; it is complicated and not practical)

2. Choose a **pivot randomly** and pretend it is the median (it is practical)
Quicksort Using A Random Pivot

**quicksort**\((A, n)\)

1. if \(n \leq 1\) then return \(A\)
2. \(x \leftarrow\) a random element of \(A\) (\(x\) is called a pivot)
3. \(A_L \leftarrow\) array of elements in \(A\) that are less than \(x\) \(\parallel\) Divide
4. \(A_R \leftarrow\) array of elements in \(A\) that are greater than \(x\) \(\parallel\) Divide
5. \(B_L \leftarrow\) quicksort\((A_L, \text{length of } A_L)\) \(\parallel\) Conquer
6. \(B_R \leftarrow\) quicksort\((A_R, \text{length of } A_R)\) \(\parallel\) Conquer
7. \(t \leftarrow\) number of times \(x\) appear \(A\)
8. return concatenation of \(B_L, t\) copies of \(x\), and \(B_R\)

**Lemma** The expected running time of the algorithm is \(O(n \log n)\).
Outline

4 Divide-and-Conquer
  - Counting Inversions
  - Solving Recurrences
  - Quicksort
  - Lower Bound for Comparison-Based Sorting Algorithms
  - Selection Problem
  - Polynomial Multiplication
  - Strassen’s Algorithm for Matrix Multiplication
  - Finding Closest Pair of Points in 2D Euclidean Space
  - Computing $n$-th Fibonacci Number
Comparison-Based Sorting Algorithms

Q: Can we do better than $O(n \log n)$ for sorting?

A: No, for comparison-based sorting algorithms.

Comparison-Based Sorting Algorithms

- To sort, we are only allowed to compare two elements
- We can not use “internal structures” of the elements

Number of comparisons is at least $\log_2 n! = \Theta(n \log n)$
Outline

4 Divide-and-Conquer
  • Counting Inversions
  • Solving Recurrences
  • Quicksort
  • Lower Bound for Comparison-Based Sorting Algorithms
  • Selection Problem
    • Polynomial Multiplication
    • Strassen’s Algorithm for Matrix Multiplication
    • Finding Closest Pair of Points in 2D Euclidean Space
    • Computing $n$-th Fibonacci Number
### Selection Problem

**Input:** a set $A$ of $n$ numbers, and $1 \leq i \leq n$

**Output:** the $i$-th smallest number in $A$

- Sorting solves the problem in time $O(n \log n)$.
- Our goal: $O(n)$ running time
Selection Algorithm with Median Finder

**selection**(*A*, *n*, *i*)

1. **if** *n* = 1 **then return** *A*
2. \(x \leftarrow\) lower median of *A*
3. \(A_L \leftarrow\) elements in *A* that are less than *x* ▷ **Divide**
4. \(A_R \leftarrow\) elements in *A* that are greater than *x* ▷ **Divide**
5. **if** *i* \(\leq\) \(A_L\).size **then**
6. return **selection**(*A_L*, \(A_L\).size, *i*) ▷ **Conquer**
7. **else if** *i* > *n* – \(A_R\).size **then**
8. return **selection**(*A_R*, \(A_R\).size, *i* – (*n* – \(A_R\).size)) ▷ **Conquer**
9. **else**
10. return *x*

- Recurrence for selection: \(T(n) = T(n/2) + O(n)\)
- Solving recurrence: \(T(n) = O(n)\)
Randomized Selection Algorithm

\[
\text{selection}(A, n, i)
\]

1. if \( n = 1 \) then return \( A \)
2. \( x \leftarrow \) random element of \( A \) (called pivot)
3. \( A_L \leftarrow \) elements in \( A \) that are less than \( x \) ▶ Divide
4. \( A_R \leftarrow \) elements in \( A \) that are greater than \( x \) ▶ Divide
5. if \( i \leq A_L.\text{size} \) then
6. \hspace{1em} return \( \text{selection}(A_L, A_L.\text{size}, i) \) ▶ Conquer
7. else if \( i > n - A_R.\text{size} \) then
8. \hspace{1em} return \( \text{selection}(A_R, A_R.\text{size}, i - (n - A_R.\text{size})) \) ▶ Conquer
9. else
10. \hspace{1em} return \( x \)

- expected running time = \( O(n) \)
Outline

**Divide-and-Conquer**
- Counting Inversions
- Solving Recurrences
- Quicksort
- Lower Bound for Comparison-Based Sorting Algorithms
- Selection Problem

**Polynomial Multiplication**
- Strassen’s Algorithm for Matrix Multiplication
- Finding Closest Pair of Points in 2D Euclidean Space
- Computing $n$-th Fibonacci Number
Polynomial Multiplication

**Input:** two polynomials of degree $n - 1$

**Output:** product of two polynomials

\[
pq = (p_H x^{n/2} + p_L)(q_H x^{n/2} + q_L) = p_H q_H x^n + (p_H q_L + p_L q_H) x^{n/2} + p_L q_L
\]

\[
\bullet \quad p_H q_L + p_L q_H = (p_H + p_L)(q_H + q_L) - p_H q_H - p_L q_L
\]
Divide-and-Conquer for Polynomial Multiplication

\[ r_H = \text{multiply}(p_H, q_H) \]
\[ r_L = \text{multiply}(p_L, q_L) \]

\[
\text{multiply}(p, q) = r_H \times x^n + (\text{multiply}(p_H + p_L, q_H + q_L) - r_H - r_L) \times x^{n/2} + r_L
\]

- Solving Recurrence: \[ T(n) = 3T(n/2) + O(n) \]
- \[ T(n) = O(n^{\log_2 3}) = O(n^{1.585}) \]
Outline

Divide-and-Conquer
- Counting Inversions
- Solving Recurrences
- Quicksort
- Lower Bound for Comparison-Based Sorting Algorithms
- Selection Problem
- Polynomial Multiplication
- Strassen’s Algorithm for Matrix Multiplication
- Finding Closest Pair of Points in 2D Euclidean Space
- Computing \( n \)-th Fibonacci Number
Matrix Multiplication

**Input:** two $n \times n$ matrices $A$ and $B$

**Output:** $C = AB$

\[
C = \begin{pmatrix}
A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\
A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22}
\end{pmatrix}
\]

- $M_1 \leftarrow (A_{11} + A_{22}) \times (B_{11} + B_{22})$
- $M_2 \leftarrow (A_{21} + A_{22}) \times B_{11}$
- $M_3 \leftarrow A_{11} \times (B_{12} - B_{22})$
- $M_4 \leftarrow A_{22} \times (B_{21} - B_{11})$
- $M_5 \leftarrow (A_{11} + A_{12}) \times B_{22}$
- $M_6 \leftarrow (A_{21} - A_{11}) \times (B_{11} + B_{12})$
- $M_7 \leftarrow (A_{12} - A_{22}) \times (B_{21} + B_{22})$
- $C_{11} \leftarrow M_1 + M_4 - M_5 + M_7$
- $C_{12} \leftarrow M_3 + M_5$
- $C_{21} \leftarrow M_2 + M_4$
- $C_{22} \leftarrow M_1 - M_2 + M_3 + M_6$
Outline

4 Divide-and-Conquer
- Counting Inversions
- Solving Recurrences
- Quicksort
- Lower Bound for Comparison-Based Sorting Algorithms
- Selection Problem
- Polynomial Multiplication
- Strassen’s Algorithm for Matrix Multiplication
- Finding Closest Pair of Points in 2D Euclidean Space
- Computing $n$-th Fibonacci Number
**Closest Pair**

**Input:** $n$ points in plane: $(x_1, y_1), (x_2, y_2), \cdots, (x_n, y_n)$

**Output:** the pair of points that are closest

- Trivial algorithm: $O(n^2)$ running time
Closest Pair

**Input:** $n$ points in plane: $(x_1, y_1), (x_2, y_2), \cdots, (x_n, y_n)$

**Output:** the pair of points that are closest

- Trivial algorithm: $O(n^2)$ running time
Divide-and-Conquer Algorithm for Closest Pair

- **Divide**: Divide the points into two halves via a vertical line
- **Conquer**: Solve two sub-instances recursively
- **Combine**: Check if there is a closer pair between left-half and right-half
Divide-and-Conquer Algorithm for Closest Pair

- **Divide**: Divide the points into two halves via a vertical line
- **Conquer**: Solve two sub-instances recursively
- **Combine**: Check if there is a closer pair between left-half and right-half
Divide-and-Conquer Algorithm for Closest Pair

- **Divide**: Divide the points into two halves via a vertical line
- **Conquer**: Solve two sub-instances recursively
- **Combine**: Check if there is a closer pair between left-half and right-half
Divide-and-Conquer Algorithm for Closest Pair

- Each box contains at most one pair
- For each point, only need to consider $O(1)$ boxes nearby
- Implementation: Sort points inside the stripe according to $y$-coordinates
- For every point, consider $O(1)$ points around it in the order
Outline

4 Divide-and-Conquer
   • Counting Inversions
   • Solving Recurrences
   • Quicksort
   • Lower Bound for Comparison-Based Sorting Algorithms
   • Selection Problem
   • Polynomial Multiplication
   • Strassen’s Algorithm for Matrix Multiplication
   • Finding Closest Pair of Points in 2D Euclidean Space
   • Computing $n$-th Fibonacci Number
Fibonacci Numbers

- $F_0 = 0, F_1 = 1$
- $F_n = F_{n-1} + F_{n-2}, \forall n \geq 2$
- Fibonacci sequence: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ⋯

$n$-th Fibonacci Number

**Input:** integer $n > 0$

**Output:** $F_n$
Computing $F_n$: Even Better Algorithm

\[
\begin{pmatrix}
F_n \\
F_{n-1}
\end{pmatrix} =
\begin{pmatrix}
1 & 1 \\
1 & 0
\end{pmatrix}
\begin{pmatrix}
F_{n-1} \\
F_{n-2}
\end{pmatrix}
\]

\[
\begin{pmatrix}
F_n \\
F_{n-1}
\end{pmatrix} =
\begin{pmatrix}
1 & 1 \\
1 & 0
\end{pmatrix}^2
\begin{pmatrix}
F_{n-2} \\
F_{n-3}
\end{pmatrix}
\]

\[\ldots\]

\[
\begin{pmatrix}
F_n \\
F_{n-1}
\end{pmatrix} =
\begin{pmatrix}
1 & 1 \\
1 & 0
\end{pmatrix}^{n-1}
\begin{pmatrix}
F_1 \\
F_0
\end{pmatrix}
\]
**power\( (n) \)**

1. if \( n = 0 \) then return \( \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \)
2. \( R \leftarrow \) power\( ([n/2]) \)
3. \( R \leftarrow R \times R \)
4. if \( n \) is odd then \( R \leftarrow R \times \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \)
5. return \( R \)

**Fib\( (n) \)**

1. if \( n = 0 \) then return 0
2. \( M \leftarrow \) power\( (n - 1) \)
3. return \( M[1][1] \)

- Recurrence for running time? \( T(n) = T(n/2) + O(1) \)
- \( T(n) = O(\log n) \)
5 Dynamic Programming
- Weighted Interval Scheduling
- Segmented Least Squares
- Subset Sum and Knapsack Problems
- Longest Common Subsequence
- Shortest Paths in Directed Acyclic Graphs
- Matrix Chain Multiplication
- Optimum Binary Search Tree
Outline

5 Dynamic Programming
- Weighted Interval Scheduling
- Segmented Least Squares
- Subset Sum and Knapsack Problems
- Longest Common Subsequence
- Shortest Paths in Directed Acyclic Graphs
- Matrix Chain Multiplication
- Optimum Binary Search Tree
Weighted Interval Scheduling

**Input:** \( n \) jobs, job \( i \) with start time \( s_i \) and finish time \( f_i \)

each job has a weight (or value) \( v_i > 0 \)

\( i \) and \( j \) are compatible if \([s_i, f_i)\) and \([s_j, f_j)\) are disjoint

**Output:** a maximum-weight subset of mutually compatible jobs

Optimum value = 220
Designing a Dynamic Programming Algorithm

- Sort jobs according to non-decreasing order of finish times
- $opt[i]$: optimal value for instance only containing jobs $\{1, 2, \cdots, i\}$

Recursion for $opt[i]$

$$opt[i] = \max \{opt[i - 1], v_i + opt[p_i]\}$$
Dynamic Programming

1: sort jobs by non-decreasing order of finishing times
2: compute $p_1, p_2, \cdots, p_n$
3: $\text{opt}[0] \leftarrow 0$
4: for $i \leftarrow 1$ to $n$ do
5: \hspace{1em} $\text{opt}[i] \leftarrow \max\{\text{opt}[i - 1], v_i + \text{opt}[p_i]\}$

- Running time sorting: $O(n \lg n)$
- Running time for computing $p$: $O(n \lg n)$ via binary search
- Running time for computing $\text{opt}[n]$: $O(n)$
Outline

5 Dynamic Programming
- Weighted Interval Scheduling
- Segmented Least Squares
- Subset Sum and Knapsack Problems
- Longest Common Subsequence
- Shortest Paths in Directed Acyclic Graphs
- Matrix Chain Multiplication
- Optimum Binary Search Tree
Linear Regression

- \( P = \{(x_1, y_1), (x_2, y_2), \cdots, (x_n, y_n)\} \), \( x_1 < x_2 < \cdots < x_n \)
- \( L : y = ax + b \)

\[
\text{Error}(L, P) = \sum_{i=1}^{n} (y_i - ax_i - b)^2
\]

- find \( L \), minimize \( \text{Error}(L, P) \)

\[
a := \frac{n \sum_i x_i y_i - (\sum_i x_i)(\sum_i y_i)}{n \sum_i x_i^2 - (\sum_i x_i)^2}
\]

\[
b := \frac{\sum_i y_i - a \sum_i x_i}{n}
\]
Segmented Least Squares

**Input:** \((x_1, y_1), (x_2, y_2), \cdots, (x_n, y_n), x_1 < x_2 < \cdots < x_n\)

penalty parameter \(C > 0\)

**Output:** partition into \(k \geq 1\) (\(k\) unknown) segments,

minimize cost := error + penalty

error: sum of squared error over all the \(k\) segments

penalty: \(kC\)

\[
\text{cost} = \text{error}(L_1, P_1) + \text{error}(L_2, P_2) + \text{error}(L_3, P_3) + 3C
\]
Dynamic Programming

- \( e_{ji}, 1 \leq j \leq i \leq n \): minimum error for \((x_j, y_j), \ldots, (x_i, y_i)\) using 1 line
- \( opt[i]\): minimum cost for the instance with first \( i \) points

\[
\begin{align*}
    opt[i] &= \begin{cases} 
    0 & \text{if } i = 0 \\
    \min_{j:1 \leq j \leq i} (opt[j-1] + e_{ji}) + C & \text{if } i \geq 1
    \end{cases}
\end{align*}
\]

- running time = \(O(n^2)\).
Dynamic Programming

- Weighted Interval Scheduling
- Segmented Least Squares
- Subset Sum and Knapsack Problems
- Longest Common Subsequence
- Shortest Paths in Directed Acyclic Graphs
- Matrix Chain Multiplication
- Optimum Binary Search Tree
Subset Sum Problem

**Input:** an integer bound $W > 0$

a set of $n$ items, each with an integer weight $w_i > 0$

**Output:** a subset $S$ of items that

maximizes $\sum_{i \in S} w_i$ s.t. $\sum_{i \in S} w_i \leq W$.

Consider the instance: $i, W', (w_1, w_2, \cdots, w_i)$;

- $opt[i, W']$: the optimum value of the instance

$$opt[i, W'] = \begin{cases} 0 & i = 0 \\ opt[i - 1, W'] & i > 0, w_i > W' \\ \max \left\{ \begin{array}{l} \text{opt}[i - 1, W'] \\ \text{opt}[i - 1, W' - w_i] + w_i \end{array} \right\} & i > 0, w_i \leq W' \end{cases}$$
Running Time of Algorithm

1. for $W' \leftarrow 0$ to $W$ do
2. \hspace{1em} $opt[0, W'] \leftarrow 0$
3. for $i \leftarrow 1$ to $n$ do
4. \hspace{1em} for $W' \leftarrow 0$ to $W$ do
5. \hspace{2em} $opt[i, W'] \leftarrow opt[i - 1, W']$
6. \hspace{2em} if $w_i \leq W'$ and $opt[i - 1, W' - w_i] + w_i \geq opt[i, W']$ then
7. \hspace{3em} $opt[i, W'] \leftarrow opt[i - 1, W' - w_i] + w_i$
8. return $opt[n, W]$

- Running time is $O(nW)$
- Running time is pseudo-polynomial because it depends on value of the input integers.
**Knapsack Problem**

**Input:** an integer bound $W > 0$

a set of $n$ items, each with an integer weight $w_i > 0$

a value $v_i > 0$ for each item $i$

**Output:** a subset $S$ of items that

$$\text{maximizes } \sum_{i \in S} v_i \quad \text{s.t. } \sum_{i \in S} w_i \leq W.$$ 

- $\text{opt}[i, W']$: the optimum value when budget is $W'$ and items are $\{1, 2, 3, \ldots, i\}$.

$$\text{opt}[i, W'] = \begin{cases} 
0 & i = 0 \\
\text{opt}[i - 1, W'] & i > 0, w_i > W' \\
\max \left\{ \begin{array}{l}
\text{opt}[i - 1, W'] \\
\text{opt}[i - 1, W' - w_i] + v_i
\end{array} \right\} & i > 0, w_i \leq W'
\end{cases}$$
Outline

5 Dynamic Programming
- Weighted Interval Scheduling
- Segmented Least Squares
- Subset Sum and Knapsack Problems
- Longest Common Subsequence
- Shortest Paths in Directed Acyclic Graphs
- Matrix Chain Multiplication
- Optimum Binary Search Tree
A = $bacdca$  $C = adca$

C is a subsequence of A

**Def.** Given two sequences $A[1 .. n]$ and $C[1 .. t]$ of letters, $C$ is called a **subsequence** of $A$ if there exists integers $1 \leq i_1 < i_2 < i_3 < \ldots < i_t \leq n$ such that $A[i_j] = C[j]$ for every $j = 1, 2, 3, \ldots, t$.

**Longest Common Subsequence**

**Input:** $A[1 .. n]$ and $B[1 .. m]$

**Output:** the longest common subsequence of $A$ and $B$

**Example:**

- $A = 'bacdca'$
- $B = 'adbcda'$
- $\text{LCS}(A, B) = 'adca'$
Dynamic Programming for LCS

- \( opt[i, j], 0 \leq i \leq n, 0 \leq j \leq m \): length of longest common sub-sequence of \( A[1 \ldots i] \) and \( B[1 \ldots j] \).
- if \( i = 0 \) or \( j = 0 \), then \( opt[i, j] = 0 \).
- if \( i > 0, j > 0 \), then

\[
opt[i, j] = \begin{cases} 
\text{opt}[i - 1, j - 1] + 1 & \text{if } A[i] = B[j] \\
\max \{ \text{opt}[i - 1, j], \text{opt}[i, j - 1] \} & \text{if } A[i] \neq B[j]
\end{cases}
\]
Dynamic Programming

- Weighted Interval Scheduling
- Segmented Least Squares
- Subset Sum and Knapsack Problems
- Longest Common Subsequence
- Shortest Paths in Directed Acyclic Graphs
- Matrix Chain Multiplication
- Optimum Binary Search Tree
Directed Acyclic Graphs

**Def.** A directed acyclic graph (DAG) is a directed graph without (directed) cycles.

**Lemma** A directed graph is a DAG if and only if its vertices can be topologically sorted.
Shortest Paths in DAG

**Input:** directed acyclic graph $G = (V, E)$ and $w : E \rightarrow \mathbb{R}$.

Assume $V = \{1, 2, 3 \cdots, n\}$ is topologically sorted: if $(i, j) \in E$, then $i < j$

**Output:** the shortest path from 1 to $i$, for every $i \in V$
Shortest Paths in DAG

- $f[i]$: length of the shortest path from 1 to $i$

$$f[i] = \begin{cases} 
0 & \text{i} = 1 \\
\min_{j: (j,i) \in E} \{ f(j) + w(j,i) \} & \text{i} = 2, 3, \cdots, n
\end{cases}$$
Dynamic Programming

- Weighted Interval Scheduling
- Segmented Least Squares
- Subset Sum and Knapsack Problems
- Longest Common Subsequence
- Shortest Paths in Directed Acyclic Graphs
- Matrix Chain Multiplication
- Optimum Binary Search Tree
Matrix Chain Multiplication

**Input:** \( n \) matrices \( A_1, A_2, \ldots, A_n \) of sizes \( r_1 \times c_1, r_2 \times c_2, \ldots, r_n \times c_n \), such that \( c_i = r_{i+1} \) for every \( i = 1, 2, \ldots, n - 1 \).

**Output:** the order of computing \( A_1 A_2 \cdots A_n \) with the minimum number of multiplications.

**Fact**  Multiplying two matrices of size \( r \times k \) and \( k \times c \) takes \( r \times k \times c \) multiplications.

- \( \text{opt}[i, j] \): the minimum cost of computing \( A_i A_{i+1} \cdots A_j \)

\[
\text{opt}[i, j] = \begin{cases} 
0 & i = j \\
\min_{k : i \leq k < j} \left( \text{opt}[i, k] + \text{opt}[k + 1, j] + r_i c_k c_j \right) & i < j
\end{cases}
\]
Dynamic Programming
- Weighted Interval Scheduling
- Segmented Least Squares
- Subset Sum and Knapsack Problems
- Longest Common Subsequence
- Shortest Paths in Directed Acyclic Graphs
- Matrix Chain Multiplication
- Optimum Binary Search Tree
Optimum Binary Search Tree

- $n$ elements $e_1 < e_2 < e_3 < \cdots < e_n$
- $e_i$ has frequency $f_i$
- goal: build a binary search tree for $\{e_1, e_2, \cdots, e_n\}$ with the minimum accessing cost:

$$
\sum_{i=1}^{n} f_i \times \text{(depth of } e_i \text{ in the tree)}
$$

- $opt[i, j]$: the optimum cost for the instance $(f_i, f_i+1, \cdots, f_j)$
- In general, $opt[i, j] =$

$$
\begin{cases}
0 & \text{if } i = j + 1 \\
\min_{k:i \leq k \leq j} \left( opt[i, k - 1] + opt[k + 1, j] \right) + \sum_{\ell=i}^{j} f_\ell & \text{if } i \leq j
\end{cases}
$$
Outline

6  Graph Algorithms
   • Minimum Spanning Tree, Kruskal’s Algorithm, Prim’s Algorithm
   • Shortest Path Algorithms
   • Minimum Cost Arborescence
Outline

6 Graph Algorithms
- Minimum Spanning Tree, Kruskal’s Algorithm, Prim’s Algorithm
- Shortest Path Algorithms
- Minimum Cost Arborescence
Minimum Spanning Tree (MST) Problem

**Input:** Graph $G = (V, E)$ and edge weights $w : E \to \mathbb{R}$

**Output:** the spanning tree $T$ of $G$ with the minimum total weight

Two Classic Greedy Algorithms for MST

- Kruskal’s Algorithm
- Prim’s Algorithm
Kruskal’s Algorithm

Lemma It is safe to include the lightest edge in the MST.
Residual Problem by Contraction

\[ g^* \]
Kruskal’s Algorithm: Example

Sets: \{a\}, \{b\}, \{c\}, \{d\}, \{e\}, \{f\}, \{g\}, \{h\}, \{i\}
Kruskal’s Algorithm: Example

Sets: \{a\}, \{b\}, \{c\}, \{d\}, \{e\}, \{f\}, \{g\}, \{h\}, \{i\}
Kruskal’s Algorithm: Example

Sets: \{a\}, \{b\}, \{c\}, \{d\}, \{e\}, \{f\}, \{g, h\}, \{i\}
Kruskal’s Algorithm: Example

Sets: \{a\}, \{b\}, \{c\}, \{d\}, \{e\}, \{f\}, \{g, h\}, \{i\}
Kruskal’s Algorithm: Example

Sets: \{a\}, \{b\}, \{c, i\}, \{d\}, \{e\}, \{f\}, \{g, h\}
Kruskal’s Algorithm: Example

Sets: \{a\}, \{b\}, \{c, i\}, \{d\}, \{e\}, \{f\}, \{g, h\}
Sets: \{a\}, \{b\}, \{c, i\}, \{d\}, \{e\}, \{f, g, h\}
Kruskal’s Algorithm: Example

Sets: \{a\}, \{b\}, \{c, i\}, \{d\}, \{e\}, \{f, g, h\}
Kruskal’s Algorithm: Example

Sets: $\{a\}, \{b\}, \{c, i, f, g, h\}, \{d\}, \{e\}$
Kruskal’s Algorithm: Example

Sets: \{a\}, \{b\}, \{c, i, f, g, h\}, \{d\}, \{e\}
Kruskal’s Algorithm: Example

Sets: \{a, b\}, \{c, i, f, g, h\}, \{d\}, \{e\}
Kruskal’s Algorithm: Example

Sets: \{a, b\}, \{c, i, f, g, h\}, \{d\}, \{e\}
Kruskal’s Algorithm: Example

Sets: \{a, b, c, i, f, g, h\}, \{d\}, \{e\}
Kruskal’s Algorithm: Example

Sets: \{a, b, c, i, f, g, h\}, \{d\}, \{e\}
Sets: \{a, b, c, i, f, g, h\}, \{d, e\}
Kruskal’s Algorithm: Example

Sets: \{a, b, c, i, f, g, h\}, \{d, e\}
Sets: \{a, b, c, i, f, g, h, d, e\}
Running Time of Kruskal’s Algorithm

MST-Kruskal\((G, w)\)

1: \(F \leftarrow \emptyset\)
2: \(S \leftarrow \{\{v\} : v \in V\}\)
3: sort the edges of \(E\) in non-decreasing order of weights \(w\)
4: for each edge \((u, v) \in E\) in the order do
5: \(S_u \leftarrow\) the set in \(S\) containing \(u\)
6: \(S_v \leftarrow\) the set in \(S\) containing \(v\)
7: if \(S_u \neq S_v\) then
8: \(F \leftarrow F \cup \{(u, v)\}\)
9: \(S \leftarrow S \setminus \{S_u\} \setminus \{S_v\} \cup \{S_u \cup S_v\}\)
10: return \((V, F)\)

Use union-find data structure to support \(2, 5, 6, 7, 9\).
Prim’s Algorithm

Greedy strategy for Prim’s algorithm: choose the lightest edge incident to $a$. 
Prim’s Algorithm: Example
Prim’s Algorithm: Example
Prim’s Algorithm: Example
Prim’s Algorithm: Example
Prim’s Algorithm: Example
Prim’s Algorithm: Example
Prim’s Algorithm: Example
Prim’s Algorithm: Example
Prim’s Algorithm: Example
Prim’s Algorithm: Example
Prim’s Algorithm: Example
Prim’s Algorithm: Example
Prim’s Algorithm: Example
Prim’s Algorithm: Example
Prim’s Algorithm: Example
Prim’s Algorithm: Example
Prim’s Algorithm: Example
Prim’s Algorithm: Example
Prim’s Algorithm

MST-Prim\((G, w)\)

1: \(s \leftarrow \text{arbitrary vertex in } G\)
2: \(S \leftarrow \emptyset, d(s) \leftarrow 0 \text{ and } d[v] \leftarrow \infty \text{ for every } v \in V \setminus \{s\}\)
3: \(\text{while } S \neq V \text{ do}\)
4: \(u \leftarrow \text{vertex in } V \setminus S \text{ with the minimum } d[u]\)
5: \(S \leftarrow S \cup \{u\}\)
6: \(\text{for each } v \in V \setminus S \text{ such that } (u, v) \in E \text{ do}\)
7: \(\text{if } w(u, v) < d[v] \text{ then}\)
8: \(d[v] \leftarrow w(u, v)\)
9: \(\pi[v] \leftarrow u\)
10: \(\text{return } \{(u, \pi[u]) | u \in V \setminus \{s\}\}\)
Running Time of Prim’s Algorithm Using Priority Queue

\[ O(n) \times \text{(time for extract\_min)} + O(m) \times \text{(time for decrease\_key)} \]

<table>
<thead>
<tr>
<th>concrete DS</th>
<th>extract_min</th>
<th>decrease_key</th>
<th>overall time</th>
</tr>
</thead>
<tbody>
<tr>
<td>heap</td>
<td>( O(\log n) )</td>
<td>( O(\log n) )</td>
<td>( O(m \log n) )</td>
</tr>
<tr>
<td>Fibonacci heap</td>
<td>( O(\log n) )</td>
<td>( O(1) )</td>
<td>( O(n \log n + m) )</td>
</tr>
</tbody>
</table>
“Evidence” for $e \in \text{MST}$ or $e \notin \text{MST}$

Assumption  Assume all edge weights are different.

- $e \in \text{MST} \iff$ there is a cut in which $e$ is the lightest edge
- $e \notin \text{MST} \iff$ there is a cycle in which $e$ is the heaviest edge

Exactly one of the following is true:
- There is a cut in which $e$ is the lightest edge
- There is a cycle in which $e$ is the heaviest edge
Outline

6 Graph Algorithms
- Minimum Spanning Tree, Kruskal’s Algorithm, Prim’s Algorithm
- Shortest Path Algorithms
- Minimum Cost Arborescence
<table>
<thead>
<tr>
<th>algorithm</th>
<th>graph</th>
<th>weights</th>
<th>SS?</th>
<th>running time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simple DP</td>
<td>DAG</td>
<td>( \mathbb{R} )</td>
<td>SS</td>
<td>( O(n + m) )</td>
</tr>
<tr>
<td>Dijkstra</td>
<td>U/D</td>
<td>( \mathbb{R}_{\geq 0} )</td>
<td>SS</td>
<td>( O(n \log n + m) )</td>
</tr>
<tr>
<td>Bellman-Ford</td>
<td>U/D</td>
<td>( \mathbb{R} )</td>
<td>SS</td>
<td>( O(nm) )</td>
</tr>
<tr>
<td>Floyd-Warshall</td>
<td>U/D</td>
<td>( \mathbb{R} )</td>
<td>AP</td>
<td>( O(n^3) )</td>
</tr>
</tbody>
</table>

- DAG = directed acyclic graph
- U = undirected
- D = directed
- SS = single source
- AP = all pairs
**s-t Shortest Paths**

**Input:** directed graph $G = (V, E)$, $s, t \in V$

$w : E \rightarrow \mathbb{R}_{\geq 0}$

**Output:** shortest path from $s$ to $t$
Single Source Shortest Paths

**Input:** directed graph $G = (V, E)$, $s \in V$

$w : E \rightarrow \mathbb{R}_{\geq 0}$

**Output:** $\pi[v], v \in V \setminus s$: the parent of $v$ in shortest path tree

$d[v], v \in V \setminus s$: the length of shortest path from $s$ to $v$
Improved Running Time using Priority Queue

Dijkstra\((G, w, s)\)

1: \(s \leftarrow \text{arbitrary vertex in } G\)
2: \(S \leftarrow \emptyset, d(s) \leftarrow 0\) \(\text{and } d[v] \leftarrow \infty\) \(\text{for every } v \in V \setminus \{s\}\)
3: \(Q \leftarrow \text{empty queue, for each } v \in V: Q.\text{insert}(v, d[v])\)
4: \(\textbf{while } S \neq V \textbf{ do}\)
5: \(u \leftarrow Q.\text{extract\_min}()\)
6: \(S \leftarrow S \cup \{u\}\)
7: \(\textbf{for each } v \in V \setminus S \text{ such that } (u, v) \in E \textbf{ do}\)
8: \(\text{if } d[u] + w(u, v) < d[v] \textbf{ then}\)
9: \(d[v] \leftarrow d[u] + w(u, v), Q.\text{decrease\_key}(v, d[v])\)
10: \(\pi[v] \leftarrow u\)
11: \(\textbf{return } (\pi, d)\)
Running time:
\[ O(n) \times \text{(time for } \text{extract\_min}) + O(m) \times \text{(time for } \text{decrease\_key}) \]

<table>
<thead>
<tr>
<th>Priority-Queue</th>
<th>extract_min</th>
<th>decrease_key</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Heap</td>
<td>( O(\log n) )</td>
<td>( O(\log n) )</td>
<td>( O(m \log n) )</td>
</tr>
<tr>
<td>Fibonacci Heap</td>
<td>( O(\log n) )</td>
<td>( O(1) )</td>
<td>( O(n \log n + m) )</td>
</tr>
</tbody>
</table>
Single Source Shortest Paths, Weights May be Negative

**Input:** directed graph $G = (V, E)$, $s \in V$
assume all vertices are reachable from $s$

$w : E \rightarrow \mathbb{R}$

**Output:** shortest paths from $s$ to all other vertices $v \in V$

Unfortunately, computing the shortest simple path between two vertices is an **NP-hard** problem.

Cases: no negative cycles, need to detect negative cycles
\[ f^\ell[v], \ell \in \{0, 1, 2, 3 \cdots , n - 1\}, v \in V : \text{length of shortest path from } s \text{ to } v \text{ that uses at most } \ell \text{ edges} \]

- \( f^2[a] = 6 \)
- \( f^3[a] = 2 \)

\[
f^\ell[v] = \begin{cases} 
0 & \ell = 0, v = s \\
\infty & \ell = 0, v \neq s \\
\min \left\{ \min_{u: (u,v) \in E} (f^{\ell-1}[u] + w(u, v)) \right\} & \ell > 0
\end{cases}
\]
Bellman-Ford Algorithm

Bellman-Ford($G, w, s$)

1: $f[s] \leftarrow 0$ and $f[v] \leftarrow \infty$ for any $v \in V \setminus \{s\}$
2: for $\ell \leftarrow 1$ to $n - 1$ do
3:    updated $\leftarrow$ false
4:    for each $(u, v) \in E$ do
5:        if $f[u] + w(u, v) < f[v]$ then
6:            $f[v] \leftarrow f[u] + w(u, v)$
7:            updated $\leftarrow$ true
8:    if updated $= \text{false}$ then break
9: return $f$
All-Pair Shortest Paths

**Input:** directed graph $G = (V, E)$,

$w : E \rightarrow \mathbb{R}$ (can be negative)

**Output:** shortest path from $u$ to $v$ for every $u, v \in V$

1: for every starting point $s \in V$ do
2: run Bellman-Ford($G, w, s$)

- Running time $= O(n^2m)$
Design a Dynamic Programming Algorithm

- It is convenient to assume \( V = \{1, 2, 3, \ldots, n\} \)
- For simplicity, extend the \( w \) values to non-edges:

\[
w(i, j) = \begin{cases} 
0 & i = j \\
\text{weight of edge } (i, j) & i \neq j, (i, j) \in E \\
\infty & i \neq j, (i, j) \notin E
\end{cases}
\]
Floyd-Warshall \((G, w)\)

1: \(f \leftarrow w\)

2: for \(k \leftarrow 1\) to \(n\) do

3: \hspace{1em} for \(i \leftarrow 1\) to \(n\) do

4: \hspace{2em} for \(j \leftarrow 1\) to \(n\) do

5: \hspace{3em} if \(f[i, k] + f[k, j] < f[i, j]\) then

6: \hspace{3em} \(f[i, j] \leftarrow f[i, k] + f[k, j]\)

Lemma  Assume there are no negative cycles in \(G\). After iteration \(k\), for \(i, j \in V\), \(f[i, j]\) is exactly the length of shortest path from \(i\) to \(j\) that only uses vertices in \(\{1, 2, 3, \cdots, k\}\) as intermediate vertices.

- Running time = \(O(n^3)\).
Graph Algorithms
- Minimum Spanning Tree, Kruskal’s Algorithm, Prim’s Algorithm
- Shortest Path Algorithms
- Minimum Cost Arborescence
**Def.** An arborescence is a directed rooted tree, where all edges are directed away from the root.

**Minimum Cost Arborescence Problem**

**Input:** a directed graph $G = (V, E)$, edge weights $w : E \rightarrow \mathbb{R}_{\geq 0}$, root $r \in V$

**Output:** a minimum-cost sub-graph $T = (V, E')$ of $G$ that is an arborescence with root $r$
Outline

7 Network Flow
- Ford-Fulkerson Method
- Running Time of Ford-Fulkerson-Type Algorithm
- Bipartite Matching Problem
- $s-t$ Edge-Disjoint Paths Problem
- More Applications
Flow Network

- Abstraction of fluid flowing through edges
- Digraph $G = (V, E)$ with source $s \in V$ and sink $t \in V$
  - No edges enter $s$
  - No edges leave $t$
- Edge capacity $c_e \in \mathbb{R}_{>0}$ for every $e \in E$
Def. An \emph{s-t flow} is a function \( f : E \rightarrow \mathbb{R} \) such that

- for every \( e \in E \): \( 0 \leq f(e) \leq c_e \) \hspace{1cm} (capacity conditions)
- for every \( v \in V \setminus \{s, t\} \):
  \[
  \sum_{e \in \delta_{\text{in}}(v)} f(e) = \sum_{e \in \delta_{\text{out}}(v)} f(e). \hspace{1cm} \text{(conservation conditions)}
  \]

The \textbf{value} of a flow \( f \) is

\[
\text{val}(f) := \sum_{e \in \delta_{\text{out}}(s)} f(e).
\]

\textbf{Maximum Flow Problem}

\textbf{Input:} directed network \( G = (V, E) \), capacity function \( c : E \rightarrow \mathbb{R}_{>0} \), source \( s \in V \) and sink \( t \in V \)

\textbf{Output:} an \emph{s-t flow} \( f \) in \( G \) with the maximum \( \text{val}(f) \)
Outline

7 Network Flow
- Ford-Fulkerson Method
- Running Time of Ford-Fulkerson-Type Algorithm
- Bipartite Matching Problem
- s-t Edge-Disjoint Paths Problem
- More Applications
**Assumption** \((u, v)\) and \((v, u)\) are not both in \(E\)

**Def.** For a \(s-t\) flow \(f\), the residual graph \(G_f\) of \(G = (V, E)\) w.r.t. \(f\) contains:
- the vertex set \(V\),
- for every \(e = (u, v) \in E\) with \(f(e) < c_e\), a forward edge \(e = (u, v)\), with residual capacity \(c_f(e) = c_e - f(e)\),
- for every \(e = (u, v) \in E\) with \(f(e) > 0\), a backward edge \(e' = (v, u)\), with residual capacity \(c_f(e') = f(e)\).
Augmenting Path

Augmenting the flow along a path $P$ from $s$ to $t$ in $G_f$

**Augment($P$)**

1. $b \leftarrow \min_{e \in P} c_f(e)$
2. **for** every $(u, v) \in P$ **do**
3. **if** $(u, v)$ is a forward edge **then**
4. \[ f(u, v) \leftarrow f(u, v) + b \]
5. **else** $(u, v)$ is a backward edge
6. \[ f(v, u) \leftarrow f(v, u) - b \]
7. **return** $f$
Ford-Fulkerson’s Method

**Ford-Fulkerson** \((G, s, t, c)\)

1. let \(f(e) \leftarrow 0\) for every \(e\) in \(G\)
2. while there is a path from \(s\) to \(t\) in \(G_f\) do
3. let \(P\) be any simple path from \(s\) to \(t\) in \(G_f\)
4. \(f \leftarrow\) augment \((f, P)\)
5. return \(f\)
The procedure \( \text{augment}(f, P) \) maintains the two conditions:

1. for every \( e \in E \): \( 0 \leq f(e) \leq c_e \) (capacity conditions)
2. for every \( v \in V \setminus \{s, t\} \):

\[
\sum_{e \in \delta_{in}(v)} f(e) = \sum_{e \in \delta_{out}(v)} f(e). \quad \text{(conservation conditions)}
\]

When Ford-Fulkerson’s Method terminates, \( \text{val}(f) \) is maximized

Ford-Fulkerson’s Method will terminate
Coro.\[ \text{val}(f) \leq \min_{s-t \text{ cut } (S,T)} c(S, T) \text{ for every } s-t \text{ flow } f. \]

**Main Lemma** The flow \( f \) found by the Ford-Fulkerson’s Method satisfies
\[ \text{val}(f) = c(S, T) \text{ for some } s-t \text{ cut } (S, T). \]

Corollary and Main Lemma implies

**Maximum Flow Minimum Cut Theorem**
\[ \sup_{s-t \text{ flow } f} \text{val}(f) = \min_{s-t \text{ cut } (S,T)} c(S, T). \]
Maximum Flow Minimum Cut Theorem

\[
\sup_{s-t \text{ flow } f} \ \text{val}(f) = \min_{s-t \text{ cut } (S,T)} \ c(S,T).
\]
Outline

7 Network Flow

- Ford-Fulkerson Method
- Running Time of Ford-Fulkerson-Type Algorithm
- Bipartite Matching Problem
- $s$-$t$ Edge-Disjoint Paths Problem
- More Applications
Shortest Augmenting Path

\[
\text{shortest-augmenting-path}(G, s, t, c)
\]

1: let \( f(e) \leftarrow 0 \) for every \( e \) in \( G \)
2: while there is a path from \( s \) to \( t \) in \( G_f \) do
3: \( P \leftarrow \text{breadth-first-search}(G_f, s, t) \)
4: \( f \leftarrow \text{augment}(f, P) \)
5: return \( f \)

Due to [Dinitz 1970] and [Edmonds-Karp, 1970]
Capacity-Scaling Algorithm

- Idea: find the augment path from $s$ to $t$ with a sufficiently large bottleneck capacity
- Assumption: Capacities are integers between 1 and $C$

**capacity-scaling**($G, s, t, c$)

1. let $f(e) \leftarrow 0$ for every $e$ in $G$
2. $\Delta \leftarrow$ largest power of 2 which is at most $C$
3. while $\Delta \geq 1$ do do
   4. while there exists an augmenting path $P$ with bottleneck capacity at least $\Delta$ do
   5. $f \leftarrow$ augment($f, P$)
   6. $\Delta \leftarrow \Delta/2$
7. return $f$
Outline

7 Network Flow
- Ford-Fulkerson Method
- Running Time of Ford-Fulkerson-Type Algorithm
- Bipartite Matching Problem
- $s$-$t$ Edge-Disjoint Paths Problem
- More Applications
Def. Given a bipartite graph $G = (L \cup R, E)$, a matching in $G$ is a set $M \subseteq E$ of edges such that every vertex in $V$ is an endpoint of at most one edge in $M$.

**Maximum Bipartite Matching Problem**

**Input:** bipartite graph $G = (L \cup R, E)$

**Output:** a matching $M$ in $G$ of the maximum size
Reduce Maximum Bipartite Matching to Maximum Flow Problem
Perfect Matching

**Def.** Given a bipartite graph $G = (L \cup R, E)$ with $|L| = |R|$, a perfect matching $M$ of $G$ is a matching such that every vertex $v \in L \cup R$ participates in exactly one edge in $M$.

**Hall’s Theorem** Let $G = (L \cup R, E)$ be a bipartite graph with $|L| = |R|$. Then $G$ has a perfect matching if and only if $|N(X)| \geq |X|$ for every $X \subseteq L$. 
7 Network Flow

- Ford-Fulkerson Method
- Running Time of Ford-Fulkerson-Type Algorithm
- Bipartite Matching Problem
- $s$-$t$ Edge-Disjoint Paths Problem
- More Applications
**s-t Edge Disjoint Paths**

**Input:** a directed (or undirected) graph $G = (V, E)$ and $s, t \in V$

**Output:** the maximum number of edge-disjoint paths from $s$ to $t$ in $G$
Menger’s Theorem

In an undirected graph, the maximum number of edge-disjoint paths between $s$ to $t$ is equal to the minimum number of edges whose removal disconnects $s$ and $t$.

$s$-to-$t$ connectivity measures how well $s$ and $t$ are connected.
Global Min-Cut Problem

**Input:** a connected graph $G = (V, E)$

**Output:** the minimum number of edges whose removal will disconnect $G$
Outline

7 Network Flow
- Ford-Fulkerson Method
- Running Time of Ford-Fulkerson-Type Algorithm
- Bipartite Matching Problem
- $s-t$ Edge-Disjoint Paths Problem
- More Applications
Extension of Network Flow: Circulation Problem

**Input:** A digraph $G = (V, E)$
- capacities $c \in \mathbb{Z}_E^{\geq 0}$
- supply vector $d \in \mathbb{Z}^V$ with $\sum_{v \in V} d_v = 0$

**Output:** whether there exists $f : E \to \mathbb{Z}_{\geq 0}$ s.t.

$$
\sum_{e \in \delta^{\text{out}}(v)} f(e) - \sum_{e \in \delta^{\text{in}}(v)} f(e) = d_v \quad \forall v \in V
$$

$$
0 \leq f(e) \leq c_e \quad \forall e \in E
$$

- $d_v$ denotes the net supply of a good
- $d_v > 0$: there is a supply of $d_v$ at $v$
- $d_v < 0$: there is a demand of $-d_v$ at $v$

problem: whether we can match the supplies and demands without violating capacity constraints
Lemma  The instance is feasible if and only if for every $S \subseteq V$, 
$d(S) \leq c(S, V \setminus S)$. 

Example

Reduction
Circulation Problem with Capacity Lower Bounds

**Input:** A digraph $G = (V, E)$
- capacities $c \in \mathbb{Z}_E^E \geq 0$
- capacity lower bounds $l \in \mathbb{Z}_E^E$, $0 \leq l_e \leq c_e$
- supply vector $d \in \mathbb{Z}_V^V$ with $\sum_{v \in V} d_v = 0$

**Output:** whether there exists $f : E \rightarrow \mathbb{Z}_{\geq 0}$ s.t.

$$\sum_{e \in \delta^{\text{out}}(v)} f(e) - \sum_{e \in \delta^{\text{in}}(v)} f(e) = d_v \quad \forall v \in V$$

$$l_e \leq f(e) \leq c_e \quad \forall e \in E$$
Removing Capacity Lower Bounds

Handling $e = (u, v)$ with $l_e > 0$

- $d'_u \leftarrow d_u - l_e$
- $d'_v \leftarrow d_v + l_e$
- $c'_e \leftarrow c_e - l_e$
- $l'_e \leftarrow 0$

- In old instance: flow is $f(e) \in [l_e, c_e] \implies f(e) - l_e \in [0, c_e - l_e]$
- In new instance: flow is $f(e) - l_e \in [0, c'_e]$
Survey Design

**Input:** integers $n, k \geq 1$ and $E \subseteq [n] \times [k]$
integers $0 \leq c_i \leq c_i', \forall i \in [n]$
integers $0 \leq p_j \leq p_j', \forall j \in [k]$

**Output:** $E' \subseteq E$ s.t.
\[
    c_i \leq |\{j \in [k] : (i, j) \in E'\}| \leq c_i', \quad \forall i \in [n]
\]
\[
    p_j \leq |\{i \in [m] : (i, j) \in E'\}| \leq p_j', \quad \forall j \in [k]
\]
Reduction to Circulation

- vertices $\{s, t\} \cup [n] \cup [k]$,
- $(i, j) \in E$: $(i, j)$ with bounds $[0, 1]$
- $\forall i$: $(s, i)$ with bounds $[c_i, c'_i]$
- $\forall j$: $(j, t)$ with bounds $[p_j, p'_j]$
- $(t, s)$ with bounds $[0, \infty]$
Airline Scheduling

**Input:** a DAG $G = (V, E)$

**Output:** the minimum number of disjoint paths in $G$ to cover all vertices
Image Segmentation

**Input:** A graph $G = (V, E)$, with edge costs $c \in \mathbb{Z}_E^{\geq 0}$

two reward vectors $a, b \in \mathbb{Z}_V^{\geq 0}$

**Output:** a cut $(A, B)$ of $G$ so as to maximize

$$
\sum_{v \in A} a_v + \sum_{v \in B} b_v - \sum_{(u,v) \in E : |\{u,v\}\cap A|=1} c(u,v)
$$
The cut value of \((S = \{s\} \cup A, \{t\} \cup B)\) is

\[
\sum_{v \in V} (a_v + b_v) - \left( \sum_{v \in A} a_v + \sum_{v \in B} b_v - \sum_{(u,v) \in E: |\{u,v\} \cap A| = 1} c(u,v) \right)
\]

\[
= \sum_{v \in V} (a_v + b_v) - \text{(objective of } (A, B))
\]

So, maximizing the objective of \((A, B)\) is equivalent to minimizing the cut value.
Project Selection

**Input:** A DAG \( G = (V, E) \)

- revenue on vertices: \( p \in \mathbb{Z}^V \); \( p_v \)'s could be negative.

**Output:** A set \( B \subseteq V \) satisfying the precedence constraints:

\[ v \in B \implies u \in B, \quad \forall (u, v) \in E \]
• min-cut $\left(S = \{s\} \cup A, T = \{t\} \cup B\right)$

• no $\infty$-capacity edges from $A$ to $B$

• cut value is

$$
\sum_{v \in B \cap L} (-p_v) + \sum_{v \in A \cap R} p_v = - \sum_{v \in B \cap L} p_v - \sum_{v \in B \cap R} p_v + \sum_{v \in R} p_v
$$

$$
= \sum_{v \in R} p_v - \sum_{v \in B} p_v
$$
More Applications

- Graph orientation
- maximum independent set (and minimum vertex cover) in a bipartite graph
- ...
Outline

8 Linear Programming
  • Linear Programming Duality
  • Integral Polytopes: Exact Algorithms Using LP
Standard Form of Linear Programming

Let \( x = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix}, \) \( c = \begin{pmatrix} c_1 \\ c_2 \\ \vdots \\ c_n \end{pmatrix}, \) \( A = \begin{pmatrix} A_{1,1} & A_{1,2} & \cdots & A_{1,n} \\ A_{2,1} & A_{2,2} & \cdots & A_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ A_{m,1} & A_{m,2} & \cdots & A_{m,n} \end{pmatrix}, \) \( b = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{pmatrix}. \)

Then, LP becomes

\[
\min \quad c^T x \quad \text{s.t.} \quad Ax \geq b \\
\quad \quad x \geq 0
\]
Standard Form of Linear Programming

\[ \min c^T x \quad \text{s.t.} \]
\[ Ax \geq b \]
\[ x \geq 0 \]

- Linear programmings can be solved in polynomial time

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Theory</th>
<th>Practice</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simplex Method</td>
<td>Exponential Time</td>
<td>Works Well</td>
</tr>
<tr>
<td>Ellipsoid Method</td>
<td>Polynomial Time</td>
<td>Slow</td>
</tr>
<tr>
<td>Internal Point Methods</td>
<td>Polynomial Time</td>
<td>Works Well</td>
</tr>
</tbody>
</table>
Outline

8 Linear Programming
  • Linear Programming Duality
  • Integral Polytopes: Exact Algorithms Using LP
### Primal LP

Minimize \( c^T x \)

Subject to:
- \( Ax \geq b \)
- \( x \geq 0 \)

### Dual LP

Maximize \( b^T y \)

Subject to:
- \( A^T y \leq c \)
- \( y \geq 0 \)

- \( P \) = value of primal LP
- \( D \) = value of dual LP

**Theorem (weak duality theorem)** \( D \leq P \).

**Theorem (strong duality theorem)** \( D = P \).

- Can always prove the optimality of the primal solution, by adding up primal constraints.
Outline

8. Linear Programming
   • Linear Programming Duality
   • Integral Polytopes: Exact Algorithms Using LP
Def. A polytope $P \subseteq \mathbb{R}^n$ is said to be integral, if all vertices of $P$ are in $\mathbb{Z}^n$.

**Maximum Weight Bipartite Matching**

**Input:** bipartite graph $G = (L \cup R, E)$

edge weights $w \in \mathbb{Z}_E^E > 0$

**Output:** a matching $M \subseteq E$ so as to maximize $\sum_{e \in M} w_e$

**LP Relaxation**

$max \sum_{e \in E} w_e x_e$

$\sum_{e \in \delta(v)} x_e \leq 1 \quad \forall v \in L \cup R$

$x_e \geq 0 \quad \forall e \in E$

**Theorem** The LP polytope is integral: It is the convex hull of $\{\chi^M : M \text{ is a matching}\}$. 
LP for Maximum Flow

\[
\begin{align*}
\text{max} & \quad \sum_{e \in \delta_{\text{in}}(t)} x_e \\
\text{subject to} & \quad x_e \leq c_e \quad \forall e \in E \\
& \quad \sum_{e \in \delta_{\text{out}}(v)} x_e - \sum_{e \in \delta_{\text{in}}(v)} x_e = 0 \quad \forall v \in V \setminus \{s, t\} \\
& \quad x_e \geq 0 \quad \forall e \in E
\end{align*}
\]

Theorem  The LP polytope is integral.
Weighted Interval Scheduling Problem

**Input:** $n$ activities, activity $i$ starts at time $s_i$, finishes at time $f_i$, and has weight $w_i > 0$

$i$ and $j$ can be scheduled together iff $[s_i, f_i)$ and $[s_j, f_j)$ are disjoint

**Output:** maximum weight subset of jobs that can be scheduled

---

The diagram illustrates the activities with their start times, end times, and weights. The optimum value is 220.
Weighted Interval Scheduling Problem

**Linear Program**

\[
\begin{align*}
\text{max} & \quad \sum_{j \in [n]} x_j w_j \\
\sum_{j \in [n]: t \in [s_j, f_j]} x_j & \leq 1 \quad \forall t \in [T] \\
x_j & \geq 0 \quad \forall j \in [n]
\end{align*}
\]

**Theorem**  The LP polytope is integral.

**Def.** A matrix \( A \in \mathbb{R}^{m \times n} \) is said to be totally unimodular (TUM), if every sub-square of \( A \) has determinant in \( \{-1, 0, 1\} \).

**Theorem** If a polytope \( P \) is defined by \( Ax \geq b, x \geq 0 \) with a totally unimodular matrix \( A \) and integral \( b \), then \( P \) is integral.

**Lemma** A matrix \( A \in \{0, 1\}^{m \times n} \) where the 1’s on every column form an interval is TUM.
**Lemma** Let $A' \in \{0, \pm 1\}^{n \times n}$ such that every row of $A'$ contains at most one 1 and one $-1$. Then $\det(A') \in \{0, \pm 1\}$.

**Lemma** Let $A \in \{0, \pm 1\}^{m \times n}$ such that every row of $A$ contains at most one 1 and one $-1$. Then $A$ is TUM.

**Coro.** The matrix for $s$-$t$ flow polytope is TUM; thus, the polytope is integral.

**Lemma** The edge-vertex incidence matrix $A$ of a bipartite graph is totally-unimodular.
Outline

NP-Completeness Theory
- P, NP and Co-NP
- Polynomial Time Reductions and NP-Completeness
- NP-Complete Problems
- Dealing with NP-Hard Problems
Outline

NP-Completeness Theory
- P, NP and Co-NP
  - Polynomial Time Reductions and NP-Completeness
  - NP-Complete Problems
  - Dealing with NP-Hard Problems
Complexity Class P

**Def.** The complexity class $P$ is the set of decision problems $X$ that can be solved in polynomial time.

- The decision versions of interval scheduling, shortest path and minimum spanning tree all in $P$. 
The Complexity Class NP

**Def.**  \( B \) is an **efficient certifier** for a problem \( X \) if

- \( B \) is a polynomial-time algorithm that takes two input strings \( s \) and \( t \), and outputs 0 or 1.
- there is a polynomial function \( p \) such that, \( X(s) = 1 \) if and only if there is string \( t \) such that \(|t| \leq p(|s|)\) and \( B(s, t) = 1 \).

The string \( t \) such that \( B(s, t) = 1 \) is called a **certificate**.

**Def.** The complexity class NP is the set of all problems for which there exists an efficient certifier.
The Complexity Class Co-NP

**Def.** For a problem $X$, the problem $\overline{X}$ is the problem such that $\overline{X}(s) = 1$ if and only if $X(s) = 0$.

**Def.** Co-NP is the set of decision problems $X$ such that $\overline{X} \in \text{NP}$.
Let $X \in P$ and $X(s) = 1$

The certificate is an empty string

Thus, $X \in NP$ and $P \subseteq NP$

Similarly, $P \subseteq Co-NP$, thus $P \subseteq NP \cap Co-NP$

$P = NP$? A famous, big, and fundamental open problem in computer science
Outline

NP-Completeness Theory
- P, NP and Co-NP
- Polynomial Time Reductions and NP-Completeness
- NP-Complete Problems
- Dealing with NP-Hard Problems
**Def.** Given a black box algorithm $A$ that solves a problem $X$, if any instance of a problem $Y$ can be solved using a polynomial number of standard computational steps, plus a polynomial number of calls to $A$, then we say $Y$ is polynomial-time reducible to $X$, denoted as $Y \leq_P X$.

**Def.** A problem $X$ is called NP-complete if
1. $X \in \text{NP}$, and
2. $Y \leq_P X$ for every $Y \in \text{NP}$.

**Theorem** If $X$ is NP-complete and $X \in \text{P}$, then $\text{P} = \text{NP}$.
Outline

9. NP-Completeness Theory
   - P, NP and Co-NP
   - Polynomial Time Reductions and NP-Completeness
   - NP-Complete Problems
   - Dealing with NP-Hard Problems
The First NP-Complete Problem: Circuit-Sat

Circuit Satisfiability (Circuit-Sat)

**Input:** a circuit

**Output:** whether the circuit is satisfiable
Circuit-Sat is NP-Complete

- key fact: algorithms can be converted to circuits

- Any problem $Y \in \text{NP}$ can be reduced to Circuit-Sat.
\[ Y \leq_p \text{Circuit-Sat, For Every } Y \in \text{NP} \]

- check-\(Y(s, t)\) returns 1 if \(t\) is a valid certificate for \(s\).
- \(s\) is a yes-instance if and only if there is a \(t\) such that check-\(Y(s, t)\) returns 1.

Construct a circuit \(C'\) for the algorithm check-\(Y\)

- hard-wire the instance \(s\) to the circuit \(C'\) to obtain the circuit \(C\)
- \(s\) is a yes-instance if and only if \(C\) is satisfiable.

**Theorem**  Circuit-Sat is NP-complete.
Reducions of NP-Complete Problems

- Circuit-Sat
- 3-Sat
- Ind-Set
- Clique
- Vertex-Cover
- Set-Cover
- 3D-Matching
- HC
- TSP
- Subset-Sum
- 3-Coloring
- Knapsack
3-Sat

3-CNF (conjunctive normal form) is a special case of formula:

- **Boolean variables:** $x_1, x_2, \cdots, x_n$
- **Literals:** $x_i$ or $\neg x_i$
- **Clause:** disjunction ("or") of at most 3 literals: $x_3 \lor \neg x_4$, $x_1 \lor x_8 \lor \neg x_9$, $\neg x_2 \lor \neg x_5 \lor x_7$
- **3-CNF formula:** conjunction ("and") of clauses: $(x_1 \lor \neg x_2 \lor \neg x_3) \land (x_2 \lor x_3 \lor x_4) \land (\neg x_1 \lor \neg x_3 \lor \neg x_4)$

**3-Sat**

**Input:** a 3-CNF formula

**Output:** whether the 3-CNF is satisfiable
Circuit-Sat $\leq_P$ 3-Sat

- Associate every wire with a new variable
- The circuit is equivalent to the following formula:

$$(x_4 = \neg x_3) \land (x_5 = x_1 \lor x_2) \land (x_6 = \neg x_4)$$
$$\land (x_7 = x_1 \land x_2 \land x_4) \land (x_8 = x_5 \lor x_6)$$
$$\land (x_9 = x_6 \lor x_7) \land (x_{10} = x_8 \land x_9 \land x_7) \land x_{10}$$
Circuit-Sat $\leq_P$ 3-Sat

$$(x_4 = \neg x_3) \land (x_5 = x_1 \lor x_2) \land (x_6 = \neg x_4)$$
$$\land (x_7 = x_1 \land x_2 \land x_4) \land (x_8 = x_5 \lor x_6)$$
$$\land (x_9 = x_6 \lor x_7) \land (x_{10} = x_8 \land x_9 \land x_7) \land x_{10}$$

Convert each clause to a 3-CNF

$$x_5 = x_1 \lor x_2 \iff$$
$$\left( x_1 \lor x_2 \lor \neg x_5 \right) \land$$
$$\left( x_1 \lor \neg x_2 \lor x_5 \right) \land$$
$$\left( \neg x_1 \lor x_2 \lor x_5 \right) \land$$
$$\left( \neg x_1 \lor \neg x_2 \lor x_5 \right)$$

Circuit $\iff$ Formula $\iff$ 3-CNF
Reductions of NP-Complete Problems

Circuit-Sat

3-Sat

3-D-Matching

3-Coloring

Subsets-Sum

Knapsack

Vertex-Cover

Set-Cover

Ind-Set

HC

TSP

Clique
Recall: Independent Set Problem

**Def.** An independent set of $G = (V, E)$ is a subset $I \subseteq V$ such that no two vertices in $I$ are adjacent in $G$.

**Independent Set (Ind-Set) Problem**

**Input:** $G = (V, E), k$

**Output:** whether there is an independent set of size $k$ in $G$
3-Sat \leq_P \text{Ind-Set}

- \((x_1 \lor \neg x_2 \lor \neg x_3) \land (x_2 \lor x_3 \lor x_4) \land (\neg x_1 \lor \neg x_3 \lor x_4)\)

- A clause \(\Rightarrow\) a group of 3 vertices, one for each literal
- An edge between every pair of vertices in same group
- An edge between every pair of contradicting literals
- Problem: whether there is an IS of size \(k = \#\text{clauses}\)

3-Sat instance is yes-instance \(\Leftrightarrow\) Ind-Set instance is yes-instance:
- satisfying assignment \(\iff\) independent set of size \(k\)
Reductions of NP-Complete Problems

Circuit-Sat

3-Sat

HC

3D-Matching

3-Coloring

Clique

Ind-Set

Vertex-Cover

Set-Cover

TSP

Subset-Sum

Knapsack
**Clique** $=^p$ **Ind-Set**

**Def.** A clique in an undirected graph $G = (V, E)$ is a subset $S \subseteq V$ such that $\forall u, v \in S$ we have $(u, v) \in E$.

**Clique Problem**

**Input:** $G = (V, E)$ and integer $k > 0$,

**Output:** whether there exists a clique of size $k$ in $G$.

**Obs.** $S$ is an independent set in $G$ if and only if $S$ is a clique in $\overline{G}$. 
Reductions of NP-Complete Problems

- 3D-Matching
- Circuit-Sat
  - 3-Sat
    - Ind-Set
      - Vertex-Cover
        - Set-Cover
    - HC
    - 3D-Matching
      - Subset-Sum
        - Knapsack
  - TSP
  - Knapsack
    - 3-Coloring
  - Clique
    - Ind-Set
      - Vertex-Cover
        - Set-Cover

Def. Given a graph $G = (V, E)$, a vertex cover of $G$ is a subset $S \subseteq V$ such that for every $(u, v) \in E$ then $u \in S$ or $v \in S$.

Vertex-Cover Problem

**Input:** $G = (V, E)$ and integer $k$

**Output:** whether there is a vertex cover of $G$ of size at most $k$

Vertex-Cover $\equiv_P$ Ind-Set

$S$ is a vertex-cover of $G = (V, E)$ if and only if $V \setminus S$ is an independent set of $G$. 
Reductions of NP-Complete Problems

- Circuit-Sat
- 3-Sat
- Ind-Set
- Vertex-Cover
- HC
- 3D-Matching
- Subset-Sum
- TSP
- Knapsack
- 3-Coloring
- Clique
- Ind-Set
- Vertex-Cover
- Set-Cover
### Set Cover

**Input:** $S_1, S_2, \cdots, S_M \subseteq [N]$ with $\bigcup_{i \in [m]} S_i = [N]$

**Output:** The smallest set $I \subseteq [M]$ satisfying $\bigcup_{i \in I} S_i = [N]$

- Decision version: given $t$, does there exist a solution $I$ with $|I| \leq t$?

### Vertex Cover $\leq_P$ Set Cover

- $m$ edges $\iff N$ elements
- $n$ vertices $\iff M$ sets
- vertex is incident to edge $e$ $\iff$ set contains element

- Vertex cover is the special case of set cover where each element appears in exactly two sets.
Reductions of NP-Complete Problems

Circuit-Sat

3-Sat

3-Coloring

Clique

Ind-Set

Vertex-Cover

3D-Matching

HC

TSP

Subset-Sum

Set-Cover

Knapsack
**Def.** A $k$-coloring of $G = (V, E)$ is a function $f : V \rightarrow \{1, 2, 3, \cdots, k\}$ so that for every edge $(u, v) \in E$, we have $f(u) \neq f(v)$. $G$ is $k$-colorable if there is a $k$-coloring of $G$.

**$k$-coloring problem**

**Input:** a graph $G = (V, E)$

**Output:** whether $G$ is $k$-colorable or not
3-SAT $\leq_P$ 3-Coloring

- Construct the base graph
- Construct a gadget from each clause: gadget is 3-colorable if and only if the clause is satisfied.

**Base Graph**

$$x_1 \lor \neg x_2 \lor x_3$$
Reductions of NP-Complete Problems

Diagram:
- Circuit-Sat
  - 3-Sat
    - Ind-Set
    - HC
      - Vertex-Cover
        - Clique
      - 3D-Matching
        - Subset-Sum
          - TSP
            - Knapsack
    - 3-Coloring
  - Set-Cover
- Subset-Sum

Recall: Hamiltonian Cycle (HC) Problem

**Input:** graph $G = (V, E)$

**Output:** whether $G$ contains a Hamiltonian cycle
3-Sat $\leq_P$ Directed-HC

- Vertices $s, t$
- A long enough double-path $P_i$ for each variable $x_i$
- Edges from $s$ to $P_1$
- Edges from $P_n$ to $t$
- Edges from $P_i$ to $P_{i+1}$
- $x_i = 1 \iff$ traverse $P_i$ from left to right
- e.g., $x_1 = 1, x_2 = 1, x_3 = 0, x_4 = 0$
3-Sat $\leq_P$ Directed-HC

- There are exactly $2^n$ different Hamiltonian cycles, each correspondent to one assignment of variables.
- Add a vertex for each clause, so that the vertex can be visited only if one of the literals is satisfied.
3-Sat $\leq_P$ Directed-HC

- $c_1 = x_1 \lor \overline{x}_2 \lor x_3$

There are exactly $2^n$ different Hamiltonian cycles, each correspondent to one assignment of variables.

- Add a vertex for each clause, so that the vertex can be visited only if one of the literals is satisfied.
A Path Should Be Long Enough

\[ \leq 3k + 1 \text{ vertices} \]

- \( k \): number of clauses
Reductions of NP-Complete Problems

- Circuit-Sat
  - 3-Sat
    - Ind-Set
    - Vertex-Cover
      - Clique
      - Set-Cover
    - HC
    - 3D-Matching
      - Subset-Sum
      - Knapsack
    - TSP
  - 3-Coloring
Traveling Salesman Problem

- A salesman needs to visit $n$ cities $1, 2, 3, \cdots, n$
- He needs to start from and return to city 1
- Goal: find a tour with the minimum cost

**Travelling Salesman Problem (TSP)**

**Input:** a graph $G = (V, E)$, weights $w : E \rightarrow \mathbb{R}_{\geq 0}$, and $L > 0$

**Output:** whether there is a tour of length at most $D$
**Obs.** There is a Hamilton cycle in $G$ if and only if there is a tour for the salesman of length $n = |V|$. 
A Strategy of Polynomial Reduction

Given an instance $s_Y$ of problem $Y$, show how to construct in polynomial time an instance $s_X$ of problem such that:

1. $s_Y$ is a yes-instance of $Y$ $\Rightarrow$ $s_X$ is a yes-instance of $X$
2. $s_X$ is a yes-instance of $X$ $\Rightarrow$ $s_Y$ is a yes-instance of $Y$
Outline

NP-Completeness Theory
- P, NP and Co-NP
- Polynomial Time Reductions and NP-Completeness
- NP-Complete Problems
- Dealing with NP-Hard Problems
Dealing with NP-Hard Problems

- Faster exponential time algorithms
- Solving the problem for special cases
- Fixed parameter tractability
- Approximation algorithms
Faster Exponential Time Algorithms

3-SAT:
- Brute-force: $O(2^n \cdot \text{poly}(n))$
- $2^n \rightarrow 1.844^n \rightarrow 1.3334^n$
- Practical SAT Solver: solves real-world sat instances with more than 10,000 variables

Travelling Salesman Problem:
- Brute-force: $O(n! \cdot \text{poly}(n))$
- Better algorithm: $O(2^n \cdot \text{poly}(n))$
- In practice: TSP Solver can solve Euclidean TSP instances with more than 100,000 vertices
Solving the problem for special cases

Maximum independent set problem is NP-hard on general graphs, but easy on

- trees
- bounded tree-width graphs
- interval graphs
- ...
Problem: whether there is a vertex cover of size $k$, for a small $k$ (number of nodes is $n$, number of edges is $\Theta(n)$.

- Brute-force algorithm: $O(kn^{k+1})$
- Better running time: $O(2^k \cdot kn)$
- Running time is $f(k)n^c$ for some $c$ independent of $k$
- Vertex-Cover is fixed-parameter tractable.
Approximation Algorithms

- For optimization problems, approximation algorithms will find sub-optimal solutions in **polynomial time**
- **Approximation ratio** is the ratio between the quality of the solution output by the algorithm and the quality of the optimal solution
- We want to make the approximation ratio as small as possible, while maintaining the property that the algorithm runs in polynomial time
- There is an 2-approximation for the vertex cover problem: we can efficiently find a vertex cover whose size is at most 2 times that of the optimal vertex cover
Advanced Topics

- Randomized Algorithms
- Extending the Limits of Tractability
- Solving NP-Hard Problems on Bounded-Tree-Width Graphs
- Approximation Algorithms using Greedy
- Arbitrarily Good Approximation Using Rounding Data
- Approximation Using LP Rounding and Primal Dual
Advanced Topics

- Randomized Algorithms
- Extending the Limits of Tractability
- Solving NP-Hard Problems on Bounded-Tree-Width Graphs
- Approximation Algorithms using Greedy
- Arbitrarily Good Approximation Using Rounding Data
- Approximation Using LP Rounding and Primal Dual
Matrix Multiplication Verification

**Input:** 3 matrices $A, B, C \in \mathbb{Z}^{n \times n}$

**Output:** whether if $C = AB$

---

**Freivald’s Matrix Verification Algorithm: one experiment**

1. randomly choose a vector $r \in \{0, 1\}^n$
2. return $ABr = Cr$

- to compute $A(Br)$, need $O(n^2)$ time

<table>
<thead>
<tr>
<th></th>
<th>true</th>
<th>false</th>
</tr>
</thead>
<tbody>
<tr>
<td>$AB = C$</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>$AB \neq C$</td>
<td>$\leq 1/2$</td>
<td>$\geq 1/2$</td>
</tr>
</tbody>
</table>

- probabilities with $k$ experiments:

<table>
<thead>
<tr>
<th></th>
<th>true</th>
<th>false</th>
</tr>
</thead>
<tbody>
<tr>
<td>$AB = C$</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>$AB \neq C$</td>
<td>$\leq 1/2^k$</td>
<td>$\geq 1 - 1/2^k$</td>
</tr>
</tbody>
</table>
Friewald’s algorithm is a Monta Carlo algorithm.

**Def.** A Monta Carlo algorithm is a randomized algorithm whose output may be incorrect with some probability.
Analysis of Randomized Quicksort Algorithm

- $T(n)$: an upper bound on the expected running time of the randomized quicksort algorithm on $n$ elements

\[
T(n) = \frac{1}{n} \sum_{i=1}^{n} (T(i-1) + T(n-i)) + O(n)
\]

\[
= \frac{2}{n} \sum_{i=0}^{n-1} T(i) + O(n)
\]

Can prove $T(n) \leq c(n \log n)$ for some constant $c$ by reduction
Indirect Analysis Using Number of Comparisons

- Running time $= O(\text{number of comparisons})$
- $\forall 1 \leq i < j \leq n$, $D_{i,j}$ indicates if we compared the $i$-th smallest element with the $j$-th smallest element
- number of comparisons $= \sum_{1 \leq i < j \leq n} D_{i,j}$

**Lemma**
$$\mathbb{E}[D_{i,j}] = \frac{2}{j-i+1} \implies \mathbb{E}[\text{number of comparisons}] = O(n \log n).$$

**Def.** A Las-Vegas algorithm is a randomized algorithm that always outputs a correct solution but has randomized running time.
Randomized Selection Algorithm

`selection(A, n, i)`

1. if $n = 1$ then return $A$
2. $x \leftarrow$ random element of $A$ (called pivot)
3. $A_L \leftarrow$ elements in $A$ that are less than $x$ ▷ Divide
4. $A_R \leftarrow$ elements in $A$ that are greater than $x$ ▷ Divide
5. if $i \leq A_L$.size then
6. return $\text{selection}(A_L, A_L$.size, $i$) ▷ Conquer
7. else if $i > n - A_R$.size then
8. return $\text{selection}(A_R, A_R$.size, $i - (n - A_R$.size)) ▷ Conquer
9. else
10. return $x$

- expected running time $= O(n)$
Global Min-Cut Problem

**Input:** a connected graph $G = (V, E)$

**Output:** the minimum number of edges whose removal will disconnect $G$
Karger’s Randomized Algorithm for Min-Cut

1: $G' = (V', E') \leftarrow G$
2: while $|V'| > 2$ do
3: pick $uv \in E'$ uniformly at random
4: contract $uv$ in $G'$, keeping parallel edges, but not self-loops
5: return the cut in $G$ correspondent to $E'$
The probability that the algorithm succeeds is at least

\[
\left(1 - \frac{2}{n}\right) \left(1 - \frac{2}{n-1}\right) \left(1 - \frac{2}{n-2}\right) \cdots \left(1 - \frac{2}{3}\right)
\]

\[= \frac{n-2}{n} \times \frac{n-3}{n-1} \times \frac{n-4}{n-2} \times \cdots \times \frac{1}{3} = \frac{2}{n(n-1)}
\]

**Coro.** Any graph \( G \) has at most \( \frac{n(n-1)}{2} \) distinct minimum cuts.

- \( A := \frac{n(n-1)}{2} \): algorithm succeeds with probability at least \( \frac{1}{A} \)
- Running the algorithm for \( Ak \) times will increase the probability to

\[
1 - \left(1 - \frac{1}{A}\right)Ak \geq 1 - e^{-k}.
\]

- To get a success probability of \( 1 - \delta \), run the algorithm for \( O(n^2 \log \frac{1}{\delta}) \) times.
Karger-Stein: A Faster Algorithm

**Karger-Stein**($G = (V, E)$)

1. **if** $|V| \leq 6$ **then return** min cut of $G$ directly
2. **repeat** twice and return the smaller cut:
3. run Karger($G'$) down to $\left\lfloor n/\sqrt{2} \right\rfloor$ vertices, to obtain $G'$
4. consider the candidate cut returned by Karger-Stein($G'$)

**Running time:**

$$T(n) = 2T\left(\frac{n}{\sqrt{2}}\right) + O(n^2)$$

$$T(n) = O(n^2 \log n)$$
Max 3-SAT

**Input:** $n$ boolean variables $x_1, x_2, \cdots, x_n$

$m$ clauses, each clause is a disjunction of 3 literals from 3 distinct variables

**Output:** an assignment so as to satisfy as many clauses as possible

**Example:**
- clauses: $x_2 \lor \neg x_3 \lor \neg x_4, \quad x_2 \lor x_3 \lor \neg x_4,$
  $\neg x_1 \lor x_2 \lor x_4, \quad x_1 \lor \neg x_2 \lor x_3, \quad \neg x_1 \lor \neg x_2 \lor \neg x_4$
- We can satisfy all the 5 clauses: $x = (1, 1, 1, 0, 1)$
Outline

Advanced Topics

- Randomized Algorithms
- Extending the Limits of Tractability
- Solving NP-Hard Problems on Bounded-Tree-Width Graphs
- Approximation Algorithms using Greedy
- Arbitrarily Good Approximation Using Rounding Data
- Approximation Using LP Rounding and Primal Dual
Finding Small Vertex Covers: Fixed Parameterized Tractability

**Vertex-Cover Problem**

**Input:** \( G = (V, E) \)

**Output:** a vertex cover \( C \) with minimum \(|C|\)

**Lemma** There is an algorithm with running time \( O(2^k \cdot kn) \) to check if \( G \) contains a vertex cover of size at most \( k \) or not.

**Vertex-Cover** \((G' = (V', E'), k)\)

1. if \(|E'| = \emptyset\) then return true
2. if \( k = 0 \) then return false
3. pick any edge \((u, v) \in E'\)
4. return Vertex-Cover\((G' \setminus u, k - 1)\) or Vertex-Cover\((G' \setminus v, k - 1)\)
Outline

Advanced Topics

- Randomized Algorithms
- Extending the Limits of Tractability
- Solving NP-Hard Problems on Bounded-Tree-Width Graphs
- Approximation Algorithms using Greedy
- Arbitrarily Good Approximation Using Rounding Data
- Approximation Using LP Rounding and Primal Dual
Bounded-Tree-Width Graphs

**Def.** A tree decomposition of a graph $G = (V, E)$ consists of
- a tree $T$ with node set $U$, and
- a subset $V_t \subseteq V$ for every $t \in U$, which we call the bag for $t$,
satisfying the following properties:
  - (Vertex Coverage) Every $v \in V$ appears in at least one bag.
  - (Edge Coverage) For every $(u, v) \in E$, some bag contains both $u$ and $v$.
  - (Coherence) For every $u \in V$, the nodes $t \in U : u \in V_t$ induce a connected sub-graph of $T$. 
**Def.** The tree-width of the tree-decomposition \((T, (V_t)_{t \in U})\) is defined as \(\max_{t \in U} |V_t| - 1\).

**Def.** The tree-width of a graph \(G = (V, E)\), denoted as \(tw(G)\), is the minimum tree-width of a tree decomposition \((T, (V_t)_{t \in U})\) of \(G\).

- The graph on the top right has tree-width 2.
Many problems on graphs with small tree-width can be solved using dynamic programming.

Typically, the running time will be exponential in $\text{tw}(G)$.

**Example: Maximum Weight Independent Set**

- given $G = (V, E)$, a tree-decomposition $(T, (V_t)_{t \in U})$ of $G$ with tree-width $\text{tw}$.
- vertex weights $w \in \mathbb{R}^V_{>0}$.
- find an independent set $S$ of $G$ with the maximum total weight.

The running time of the dynamic programming: $O(2^{\text{tw}} \cdot \text{tw} \cdot n)$.

It is efficient when $\text{tw}$ is $O(\log n)$. 
Outline

10 Advanced Topics
- Randomized Algorithms
- Extending the Limits of Tractability
- Solving NP-Hard Problems on Bounded-Tree-Width Graphs
- Approximation Algorithms using Greedy
- Arbitrarily Good Approximation Using Rounding Data
- Approximation Using LP Rounding and Primal Dual
2-Approximation Algorithm for Vertex Cover

1: \( E' \leftarrow E, C \leftarrow \emptyset \)
2: \textbf{while} \( E' \neq \emptyset \) \textbf{do}
3: \hspace{1em} let \((u, v)\) be any edge in \( E' \)
4: \hspace{1em} \( C \leftarrow C \cup \{u, v\} \)
5: \hspace{1em} remove all edges incident to \( u \) and \( v \) from \( E' \)
6: \textbf{return} \( C \)

- Can be extended to an \( f \)-approximation for set-cover problem with frequency \( f \).
Set Cover

**Input:** $U, |U| = n$: ground set
$S_1, S_2, \ldots, S_m \subseteq U$

**Output:** minimum size set $C \subseteq [m]$ such that $\bigcup_{i \in C} S_i = U$

Greedy Algorithm for Set Cover

1. $C \leftarrow \emptyset, U' \leftarrow U$
2. **while** $U' \neq \emptyset$ **do**
3. choose the $i$ that maximizes $|U' \cap S_i|$
4. $C \leftarrow C \cup \{i\}, U' \leftarrow U' \setminus S_i$
5. **return** $C$
\( g \): minimum number of sets needed to cover \( U \)

**Lemma** Let \( u_t, t \in \mathbb{Z}_{\geq 0} \) be the number of uncovered elements after \( t \) steps. Then for every \( t \geq 1 \), we have

\[
  u_t \leq \left( 1 - \frac{1}{g} \right) \cdot u_{t-1}.
\]

In at most \( \lceil g \ln(n + 1) \rceil \) iterations, all elements will be covered.
Maximum Coverage

**Input:** \( U, |U| = n \): ground set,
\( S_1, S_2, \ldots, S_m \subseteq U, \quad k \in [m] \)

**Output:** \( C \subseteq [m], |C| = k \) with the maximum \( \bigcup_{i \in C} S_i \)

Greedy Algorithm for Maximum Coverage

1. \( C \leftarrow \emptyset, U' \leftarrow U \)
2. for \( t \leftarrow 1 \) to \( k \) do
3. choose the \( i \) that maximizes \( |U' \cap S_i| \)
4. \( C \leftarrow C \cup \{i\}, U' \leftarrow U' \setminus S_i \)
5. return \( C \)

Theorem  Greedy algorithm gives \((1 - \frac{1}{e})\)-approximation for maximum coverage.
Outline

10 Advanced Topics
- Randomized Algorithms
- Extending the Limits of Tractability
- Solving NP-Hard Problems on Bounded-Tree-Width Graphs
- Approximation Algorithms using Greedy
- Arbitrarily Good Approximation Using Rounding Data
- Approximation Using LP Rounding and Primal Dual
Knapsack Problem

**Input:**
- an integer bound $W > 0$
- a set of $n$ items, each with an integer weight $w_i > 0$
- a value $v_i > 0$ for each item $i$

**Output:**
- a subset $S$ of items that

$$\text{maximizes } \sum_{i \in S} v_i \text{ s.t. } \sum_{i \in S} w_i \leq W.$$
Let $A$ be some integer to be defined later

$v'_i := \left\lfloor \frac{v_i}{A} \right\rfloor$ be the scaled value of item $i$

Definition of DP cells: $f[i, V'] = \min_{S \subseteq [i]: v'(S) \geq V'} w(S)$

$$f[i, V'] = \begin{cases} 
0 & V' \leq 0 \\
\infty & i = 0, V' > 0 \\
\min \left\{ f[i - 1, V'], f[i - 1, V' - v'_i] + w_i \right\} & i > 0, V' > 0 
\end{cases}$$

Output $A$ times the largest $V'$ such that $f[n, V'] \leq W$. 
Makespan Minimization on Identical Machines

**Input:** $n$ jobs index as $[n]$

Each job $j \in [n]$ has a processing time $p_j \in \mathbb{Z}_{>0}$

$m$ machines

**Output:** schedule of jobs on machines with minimum makespan

$\sigma : [n] \rightarrow [m]$ with minimum $\max_{i \in [m]} \sum_{j \in \sigma^{-1}(i)} p_j$

![Diagram showing job scheduling and makespan]
Dynamic Programming for Big jobs:
- Round job sizes so that there are only $k = O\left(\frac{1}{\epsilon} \log \frac{1}{\epsilon}\right)$ distinct sizes
- Running time exponential in $k$
- Greedily add small jobs.

$1 + \epsilon$-Approximation
Outline

10 Advanced Topics

- Randomized Algorithms
- Extending the Limits of Tractability
- Solving NP-Hard Problems on Bounded-Tree-Width Graphs
- Approximation Algorithms using Greedy
- Arbitrarily Good Approximation Using Rounding Data
- Approximation Using LP Rounding and Primal Dual
Weighted Vertex-Cover Problem

**Input:** \( G = (V, E) \) with vertex weights \( \{w_v\}_{v \in V} \)

**Output:** a vertex cover \( S \) with minimum \( \sum_{v \in S} w_v \)

Linear programming relaxation for WVC:

\[
\begin{align*}
\text{(LP}_{\text{WVC}}) \quad \min & \quad \sum_{v \in V} w_v x_v \\
\text{s.t.} & \quad x_u + x_v \geq 1 \quad \forall (u, v) \in E \\
& \quad x_v \in [0, 1] \quad \forall v \in V
\end{align*}
\]
Algorithm for Weighted Vertex Cover

1: Solving (LP\textsubscript{WVC}) to obtain a solution \( \{x_u^*\}_{u \in V} \)
2: Thus, LP = \( \sum_{u \in V} w_u x_u^* \leq IP \)
3: Let \( S = \{ u \in V : x_u \geq 1/2 \} \) and output \( S \)

Lemma \( S \) is a vertex cover of \( G \).

Lemma \( \text{cost}(S) := \sum_{u \in S} w_u \leq 2 \cdot \text{LP} \).
Algorithm for Weighted Vertex Cover

1: Solving \((LP_{WVC})\) to obtain a solution \(\{x_u^*\}_{u \in V}\)
2: Thus, \(LP = \sum_{u \in V} w_u x_u^* \leq IP\)
3: Let \(S = \{u \in V : x_u^* \geq 1/2\}\) and output \(S\)

**Lemma** \(S\) is a vertex cover of \(G\).

**Lemma** \(\text{cost}(S) := \sum_{u \in S} w_u \leq 2 \cdot LP\).
Algorithm constructs a maximal dual solution $y$, and returns the set of all vertices whose dual constraints are tight.