# CPSC 420 Review Session

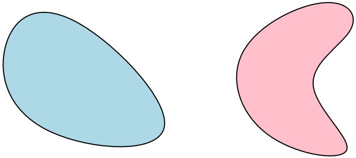
- Final Exam: Tue Apr 18, 2023 08:30am LSK 200.
  One 2-sided page of notes

## Today's Plan (Yikes)

- Convex Hull & its Algorithms
- Voronoi diagrams
- Linear Programming
- Network Flow
- Ford-Fulkerson algorithm
- Maximum matching in bipartite graphs
- Linear programming duality
- Compression
- Huffman Coding & Lempel-Ziv Compression
- P, NP, and NP-hardness
- Karp's 21 Problems
- Approximation algorithms
- Hardness of approximation
- Online Algorithms
- Cuckoo Hashing
- RSA cryptosystem
- Quantum Computing
- Zero-knowledge Proofs

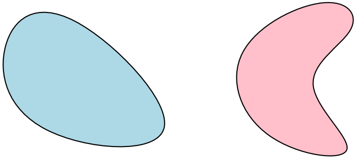
# Convex Hull

A set $S$ is **convex** if for all $a, b \in S$ the segment $\overline{ab}$ is in $S$.



# Convex Hull

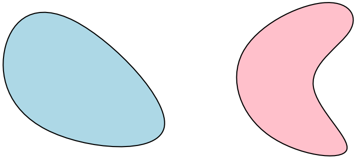
A set $S$ is **convex** if for all $a, b \in S$ the segment $\overline{ab}$ is in $S$.



The **convex hull** of a set $P$ of points is the intersection of all convex sets that contain $P$, or equivalently the set of all convex combinations of $P$ $(\sum \lambda_i p_i, \sum \lambda_i = 1)$.

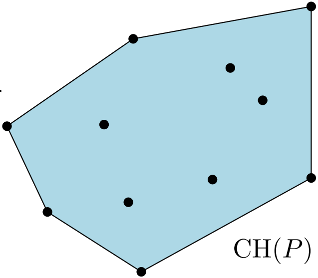
# Convex Hull

A set $S$ is **convex** if for all $a, b \in S$ the segment $\overline{ab}$ is in $S$.

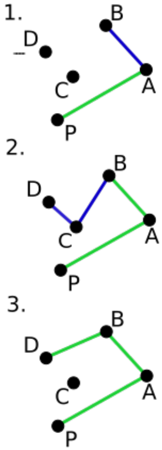


The **convex hull** of a set $P$ of points is the intersection of all convex sets that contain $P$, or equivalently the set of all convex combinations of $P$ $(\sum \lambda_i p_i, \sum \lambda_i = 1)$.

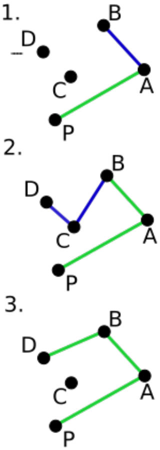
A point $p \in P$ is on the boundary of $\mathrm{CH}(P)$ iff there exists a line $\ell$ through $p$ with all $P$ on one side of $\ell$.

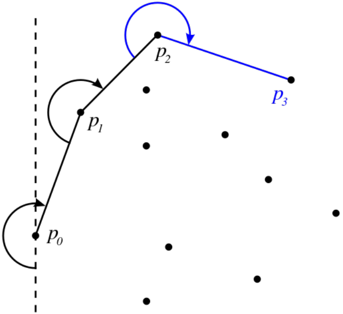


# Convex Hull Algorithms

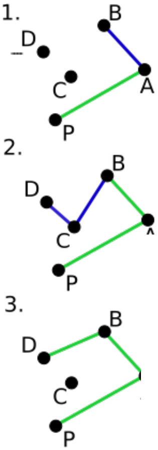


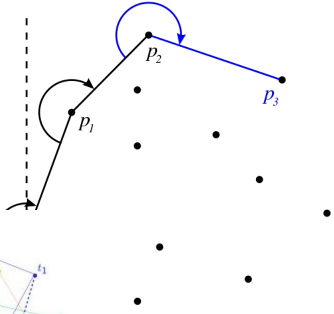
# Convex Hull Algorithms

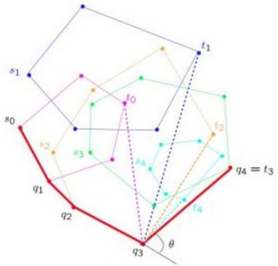


# Convex Hull Algorithms



# Reductions (the fast kind)

As a reminder: the concept of reductions is not only for NP! Here's a tiny refresher.

# Reductions (the fast kind)

As a reminder: the concept of reductions is not only for NP! Here's a tiny refresher.

I can use convex hull to sort using the mapping $i \mapsto (i, i^2)$ in time equal to $O(Convex + n)$, so convex hull can't be faster than comparison sorting, and I can use a decision tree to show that's $\Omega(\lg n)$.

# Reductions (the fast kind)

As a reminder: the concept of reductions is not only for NP! Here's a tiny refresher.

I can use convex hull to sort using the mapping $i \mapsto (i, i^2)$ in time equal to $O(Convex + n)$, so convex hull can't be faster than comparison sorting, and I can use a decision tree to show that's $\Omega(\lg n)$.

**(Reminder)** in essence the tree has $n!$ leaves, which requires a depth of $\Omega(n \lg n)$.

# Voronoi diagrams

### Definition

*A* **Voronoi diagram** *of a set of n sites (points)* $s_1, \ldots, s_n$ *is a set of regions* $R_1, \ldots, R_n$ *where* $R_i$ *is the set of points* $x$ *such that* $d(x, s_i) \leq d(x, s_j)$ *for all* $j$*.*

# Voronoi diagrams

### Definition

*A* **Voronoi diagram** *of a set of n sites (points)* $s_1, \ldots, s_n$ *is a set of regions* $R_1, \ldots, R_n$ *where* $R_i$ *is the set of points* $x$ *such that* $d(x, s_i) \leq d(x, s_j)$ *for all* $j$.

*A* **Voronoi edge** *is the border between two regions:* $\{x | x \in R_i \text{ and } x \in R_j \text{ and } i \neq j\}$.

# Voronoi diagrams

### Definition

*A* **Voronoi diagram** *of a set of n sites (points)* $s_1, \ldots, s_n$ *is a set of regions* $R_1, \ldots, R_n$ *where* $R_i$ *is the set of points* $x$ *such that* $d(x, s_i) \leq d(x, s_j)$ *for all* $j$.

*A* **Voronoi edge** *is the border between two regions:* $\{x | x \in R_i \text{ and } x \in R_j \text{ and } i \neq j\}$.

*A* **Voronoi vertex** *is the intersection of Voronoi edges:* $\{x | x \text{ in more than 2 Vor. Regions}\}$.

# Voronoi diagrams

Definition

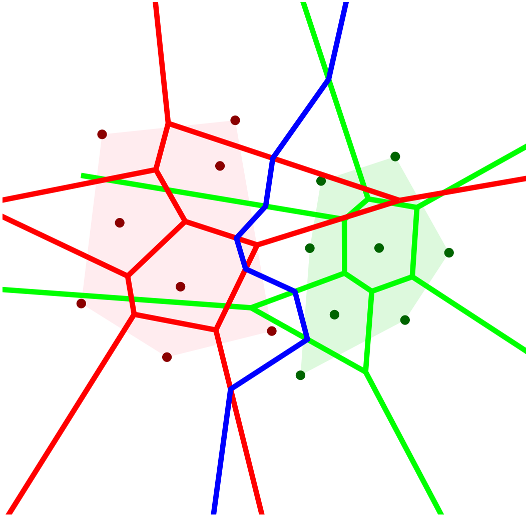
*A **Voronoi diagram** of a set of $n$ sites (points) $s_1, \ldots, s_n$ is a set of regions $R_1, \ldots, R_n$ where $R_i$ is the set of points $x$ such that $d(x, s_i) \leq d(x, s_j)$ for all $j$.*

*A **Voronoi edge** is the border between two regions: $\{x | x \in R_i$ and $x \in R_j$ and $i \neq j\}$.*

*A **Voronoi vertex** is the intersection of Voronoi edges: $\{x | x$ in more than 2 Vor. Regions$\}$.*

Problem Given $S = s_1, s_2, \ldots, s_n$, find Voronoi vertices and edges

# Algorithm from Computational Geometry by Preparata & Shamos



# Exercises

1. Of Graham and Jarvis, which is faster on a circle of points?

# Exercises

1. Of Graham and Jarvis, which is faster on a circle of points?
   **Graham**
2. Which is faster on a line?

# Exercises

1. Of Graham and Jarvis, which is faster on a circle of points? **Graham**
2. Which is faster on a line? **Jarvis**
3. How can you tell a vertex is on the convex hull using a Voronoi diagram?

# Exercises

1. Of Graham and Jarvis, which is faster on a circle of points? **Graham**
2. Which is faster on a line? **Jarvis**
3. How can you tell a vertex is on the convex hull using a Voronoi diagram? **It has infinite area**
4. Say I offered you a $O(h \log(\log h)^4)$ algorithm for Convex Hull. Is that cool?

# Exercises

1. Of Graham and Jarvis, which is faster on a circle of points? **Graham**
2. Which is faster on a line? **Jarvis**
3. How can you tell a vertex is on the convex hull using a Voronoi diagram? **It has infinite area**
4. Say I offered you a $O(h \log(\log h)^4)$ algorithm for Convex Hull. Is that cool? **No**

# Linear Programming

## Definition

*A* **Linear Program** *is a problem that can be arranged into the form*

$$\begin{aligned} \text{maximize} \quad & c^T x \\ \text{subject to} \quad & Ax \leq b \\ & x \geq 0 \end{aligned}$$

# Linear Programming

## Definition

*A* **Linear Program** *is a problem that can be arranged into the form*

$$\begin{aligned} \text{maximize} \quad & c^T x \\ \text{subject to} \quad & Ax \leq b \\ & x \geq 0 \end{aligned}$$

A popular algorithm for solving LPs is Danzig's **Simplex Algorithm**:

1. Start at a vertex $v$ of the feasible set
2. While there is a neighbor $v'$ of $v$ with better objective value
3. $\quad v = v'$

# Network Flows

A **flow network** is a directed graph $G = (V, E)$ in which each edge $(u, v) \in E$ has a positive **capacity** $c(u, v)$ (non-edges have capacity 0).

$G$ contains a **source** vertex $s$ and a **sink** vertex $t$.

# Network Flows

A **flow network** is a directed graph $G = (V, E)$ in which each edge $(u, v) \in E$ has a positive **capacity** $c(u, v)$ (non-edges have capacity 0).

$G$ contains a **source** vertex $s$ and a **sink** vertex $t$.

A **flow** is an assignment $f$ of real numbers to edges of $G$:

1. **Non-negativity:** $0 \leq f_e$
2. **Capacity** $f_e \leq c_e$
3. **Conservation** $v \neq s, t : \sum f_{\delta^-(v)} = \sum f_{\delta^+(v)}$

# Network Flows

A **flow network** is a directed graph $G = (V, E)$ in which each edge $(u, v) \in E$ has a positive **capacity** $c(u, v)$ (non-edges have capacity 0).

$G$ contains a **source** vertex $s$ and a **sink** vertex $t$.

A **flow** is an assignment $f$ of real numbers to edges of $G$:

1. **Non-negativity:** $0 \leq f_e$
2. **Capacity** $f_e \leq c_e$
3. **Conservation** $v \neq s, t : \sum f_{\delta^-(v)} = \sum f_{\delta^+(v)}$

The **size** (or **value**) of a flow is: $\text{size}(f) = \sum_v f(s, v) - f(v, s)$

# Network Flows

A **flow network** is a directed graph $G = (V, E)$ in which each edge $(u, v) \in E$ has a positive **capacity** $c(u, v)$ (non-edges have capacity 0).

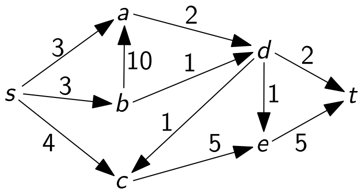
$G$ contains a **source** vertex $s$ and a **sink** vertex $t$.

A **flow** is an assignment $f$ of real numbers to edges of $G$:

1. **Non-negativity:** $0 \leq f_e$
2. **Capacity** $f_e \leq c_e$
3. **Conservation** $v \neq s, t : \sum f_{\delta^-(v)} = \sum f_{\delta^+(v)}$

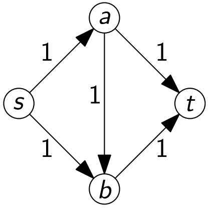
The **size** (or **value**) of a flow is: $\text{size}(f) = \sum_v f(s, v) - f(v, s)$

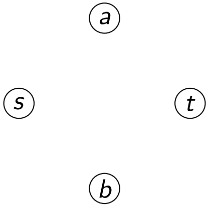
**Goal:** Find flow with maximum size.



# Max Flow via Path Augmentation [Ford & Fulkerson 1962]

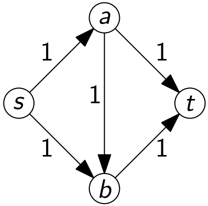
1. Start with zero flow (a feasible solution)
2. Repeat until impossible
   - Choose an **augmenting path** from $s$ to $t$
   - Increase flow on this path as much as possible



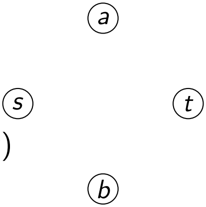
# Max Flow via Path Augmentation [Ford & Fulkerson 1962]

1. Start with zero flow (a feasible solution)
2. Repeat until impossible
   - Choose an **augmenting path** from $s$ to $t$
   - Increase flow on this path as much as possible



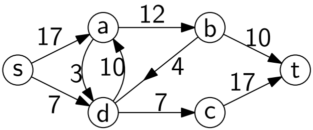

The **residual network** of flow network $G = (V, E)$ with flow $f$ is $G^f = (V, E^f)$ where
$E^f = \{(u, v) | f(u, v) < c(u, v) \text{ or } f(v, u) > 0\}$
The **residual capacity** of an edge $(u, v) \in E^f$ is

$$c^f(u, v) = \begin{cases} c(u, v) - f(u, v) & \text{if } f(u, v) < c(u, v) \\ f(v, u) & \text{if } f(v, u) > 0 \end{cases}$$

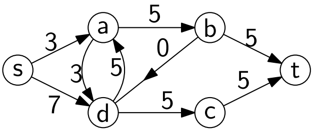



An **augmenting path** in $G$ is an $s \rightsquigarrow t$ path in $G^f$

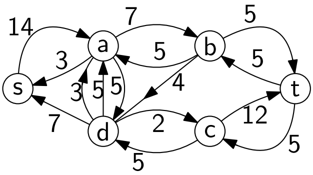
# Residual Network Example



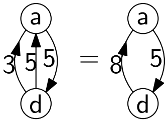

Flow network




Flow




Residual network



# Ford & Fulkerson, cont.

### More Terminology

A **cut** is a partition $(S, T)$ of $V$ such that $s \in S$ and $t \in T$. (Cut separates $s$ from $t$.)

The **capacity** of cut $(S, T)$ is $c(S, T) = \sum_{u \in S, v \in T} c(u, v)$

The **flow** across cut $(S, T)$ is $f(S, T) = \sum_{u \in S, v \in T} f(u, v) - f(v, u)$

# Ford & Fulkerson, cont.

### More Terminology

A **cut** is a partition $(S, T)$ of $V$ such that $s \in S$ and $t \in T$. (Cut separates $s$ from $t$.)

The **capacity** of cut $(S, T)$ is $c(S, T) = \sum_{u \in S, v \in T} c(u, v)$

The **flow** across cut $(S, T)$ is $f(S, T) = \sum_{u \in S, v \in T} f(u, v) - f(v, u)$

### The Gem: Max-Flow Min-Cut

$$|f^*| = c(S^*, T^*)$$

### Runtime

$O(m|f^*|)$ (pseudo-polynomial). **Solution?**

# Ford & Fulkerson, cont.

## More Terminology

A **cut** is a partition $(S, T)$ of $V$ such that $s \in S$ and $t \in T$. (Cut separates $s$ from $t$.)

The **capacity** of cut $(S, T)$ is $c(S, T) = \sum_{u \in S, v \in T} c(u, v)$

The **flow** across cut $(S, T)$ is $f(S, T) = \sum_{u \in S, v \in T} f(u, v) - f(v, u)$

## The Gem: Max-Flow Min-Cut

$$|f^*| = c(S^*, T^*)$$

## Runtime

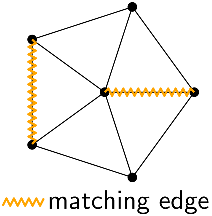
$O(m|f^*|)$ (pseudo-polynomial). **Solution?** *Edmonds-Karp* is $O(mn^2)$ by using shortest path. There is even faster though!

# Maximum Matching in Bipartite Graphs

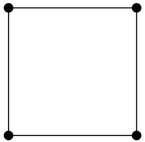
A **matching** in a graph $G$ is a subset $M$ of its edges with no vertex the endpoint of more than one edge in $M$.

A **maximum matching** is a matching with the maximum number of edges.

A **maximal matching** is a matching to which another edge cannot be added to form a new matching.




A **bipartite graph** is a graph $G = (V, E)$ where $V$ can be partitioned into $A$ and $B$ such that $\forall(u, v) \in E$, either $u \in A$ and $v \in B$ or $u \in B$ and $v \in A$.

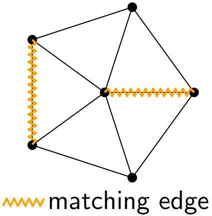


# Maximum Matching in Bipartite Graphs

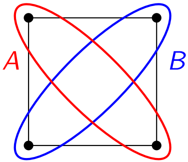
A **matching** in a graph $G$ is a subset $M$ of its edges with no vertex the endpoint of more than one edge in $M$.

A **maximum matching** is a matching with the maximum number of edges.

A **maximal matching** is a matching to which another edge cannot be added to form a new matching.




A **bipartite graph** is a graph $G = (V, E)$ where $V$ can be partitioned into $A$ and $B$ such that $\forall (u, v) \in E$, either $u \in A$ and $v \in B$ or $u \in B$ and $v \in A$.




<u>Given</u> bipartite graph $G = (V, E)$ with partitions $A$ and $B$
<u>find</u> maximum matching in $G$.

# Exercises

## Projects with Dependencies

We have $n$ projects, each with value $p_i$. $p_i > 0$ is a profit, and $p_i < 0$ is a cost. Additionally, we have an acyclic dependency graph $G$ where the edge $(i, j)$ means $i$ depends on $j$.
**Goal:** find projects to do so as to maximize profit.

# LP Duality

The dual program is constructed as follows:

- Variable $\longleftrightarrow$ Constraint
- Maximum $\longleftrightarrow$ Minimum

$$\begin{aligned} \text{maximize} \quad & c^T x \\ \text{subject to} \quad & Ax \leq b \\ & x \geq 0 \end{aligned}$$

$$\begin{aligned} \text{minimize} \quad & b^T y \\ \text{subject to} \quad & A^T y \geq c \\ & y \geq 0 \end{aligned}$$

# LP Duality

The dual program is constructed as follows:

- Variable $\longleftrightarrow$ Constraint
- Maximum $\longleftrightarrow$ Minimum

$$\begin{array}{rl} \text{maximize} & c^T x \\ \text{subject to} & Ax \leq b \\ & x \geq 0 \end{array} \qquad \begin{array}{rl} \text{minimize} & b^T y \\ \text{subject to} & A^T y \geq c \\ & y \geq 0 \end{array}$$

**Weak Duality** gives us says the dual solution is always an upper bound on the primal solution. **Strong Duality** says that they have the same optima if it exists.

# Dynamic Programming

*Dynamic Programming* is an oft-overloaded term. The two key things you need to call what you're doing *Dynamic Programming* are

# Dynamic Programming

*Dynamic Programming* is an oft-overloaded term. The two key things you need to call what you're doing *Dynamic Programming* are

1. The solution is a combination of sub-problem solutions.

# Dynamic Programming

*Dynamic Programming* is an oft-overloaded term. The two key things you need to call what you're doing *Dynamic Programming* are

1. The solution is a combination of sub-problem solutions.
2. There is a nice number of total unique subproblems, so we solve them once

# Dynamic Programming

*Dynamic Programming* is an oft-overloaded term. The two key things you need to call what you're doing *Dynamic Programming* are

1. The solution is a combination of sub-problem solutions.
2. There is a nice number of total unique subproblems, so we solve them once

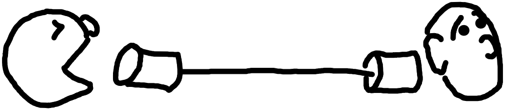
Within this, there are 2 key ways to approach the solution:

- *Recursive* or *Top-Down* solutions solve the top level by calling & memoizing sub-problems.
- *Iterative* or *Bottom-Up* solutions start with the sub-problems with no dependencies, then build up until they reach the top level.

# Exercises: 0-1 Knapsack

We have $n$ items, each with a profit $p_i$ and a weight $w_i$. Given a total profit $P$ and total weight $W$, does there exists a set $S \subseteq [n]$ with $\sum p_S \geq P, \sum p_S \leq W$?

# Information Theory [Shannon 1948]



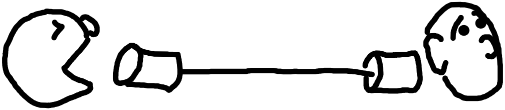
The **information content / surprisal** contained in a message $i$ that has probability $p_i$ of being sent is

$$\log_2 \frac{1}{p_i}$$

**Entropy** is the average information content of a message $X$:

$$H(X) = \sum_i p_i \log_2 \frac{1}{p_i} = -\sum_i p_i \log_2 p_i$$

# Information Theory [Shannon 1948]



The **information content / surprisal** contained in a message $i$ that has probability $p_i$ of being sent is

$$\log_2 \frac{1}{p_i}$$

**Entropy** is the average information content of a message $X$:

$$H(X) = \sum_i p_i \log_2 \frac{1}{p_i} = -\sum_i p_i \log_2 p_i$$

Source Coding Theorem $m$ i.i.d. random variables each with entropy $H(X)$ can be compressed into more than $mH(X)$ bits with negligible risk of information loss but using less than $mH(X)$ bits results almost certainly in information loss. [Wikipedia-ish]

# Huffman Coding and LZ78

## Huffman Coding

Given a set of characters $a_1, a_2, \ldots, a_\alpha$ and a probability $p_i$ for each $a_i$ ($\sum_i p_i = 1$), construct an encoding $c_i$ for each character $a_i$ so that the expected length of an encoded message is minimized, then it's just lookup.

# Huffman Coding and LZ78

## Huffman Coding

Given a set of characters $a_1, a_2, \ldots, a_\alpha$ and a probability $p_i$ for each $a_i$ ($\sum_i p_i = 1$), construct an encoding $c_i$ for each character $a_i$ so that the expected length of an encoded message is minimized, then it's just lookup.

## LZ78

Parse input into distinct phrases reading from left to right.
Each phrase is the shortest string not already a phrase. The 0th phrase is $\emptyset$.
Output $ic$ for each phrase $w$, where $c$ is the last character of $w$ and $i$ is the index of phrase $u$ where $w = u \circ c$
Length is $c(n)(\lg c(n) + \lg \alpha)$ bits, as good as any finite state compressor.

# Exercises

1. Imagine I promise you 2 compression algorithms: the first compresses to 70% on average, and at worst it compresses to 110%. The second compresses to 25% on average, and at worst it compresses to 97%. Why would you ask for the first?

# Exercises

1. Imagine I promise you 2 compression algorithms: the first compresses to 70% on average, and at worst it compresses to 110%. The second compresses to 25% on average, and at worst it compresses to 97%. Why would you ask for the first? **The second is impossible, so I am either wrong or lying.**
2. T/F: A low probability message has high information

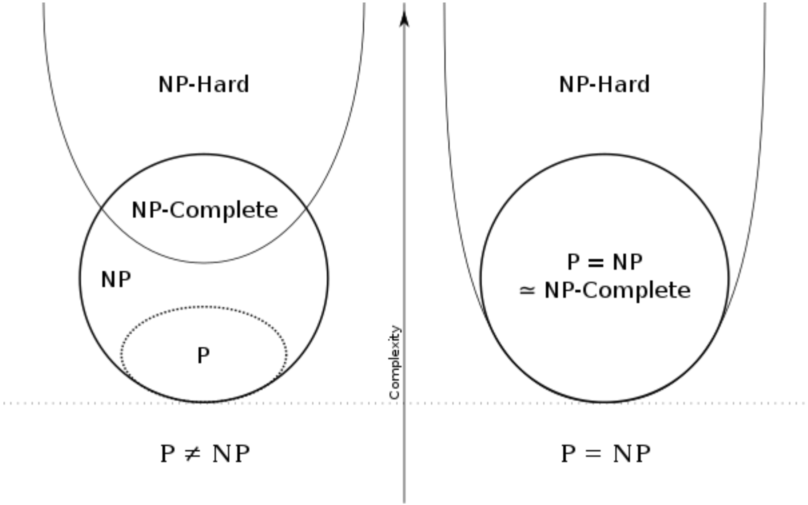
# Exercises

1. Imagine I promise you 2 compression algorithms: the first compresses to 70% on average, and at worst it compresses to 110%. The second compresses to 25% on average, and at worst it compresses to 97%. Why would you ask for the first? **The second is impossible, so I am either wrong or lying.**
2. T/F: A low probability message has high information **True**
3. Lempel-Ziv Compress the string AABAABBAA$.

# Exercises

1. Imagine I promise you 2 compression algorithms: the first compresses to 70% on average, and at worst it compresses to 110%. The second compresses to 25% on average, and at worst it compresses to 97%. Why would you ask for the first? **The second is impossible, so I am either wrong or lying.**
2. T/F: A low probability message has high information **True**
3. Lempel-Ziv Compress the string AABAABBAA$. **Dict:**$\{_{,} 0A, 1B, 1A, 0B, 4A, 1\$\}$**, String: 0A1B1A0B4A1$**

# Complexity Classes



# What is a Reduction?

There are loads of ways to conceptualize a hardness reduction from $A \rightarrow B$:

- Prove that if I can solve $B$, I can solve $A$
- I can turn an input to $A$ into an input for $B$, then transform the output back to the answer for $A$.
- If you hand me a $B$-machine, I can write a polyime algorithm for $A$

# Reduction Structure & Strategy

## Rules

The strategy of reduction we typically use is often called a *Karp Reduction* (guess why?). In this strategy, **you cannot mess with the oracle**. **All** you get to do is transform an input, then transform an output.

## Proof Structure

With that in mind, a nice Completeness/Hardness answer for problem $A$ usually follows the following structure (as do our rubrics):

# Reduction Structure & Strategy

## Rules

The strategy of reduction we typically use is often called a *Karp Reduction* (guess why?). In this strategy, **you cannot mess with the oracle**. **All** you get to do is transform an input, then transform an output.

## Proof Structure

With that in mind, a nice Completeness/Hardness answer for problem $A$ usually follows the following structure (as do our rubrics):

1. Prove $A$ in NP **if** we want Completeness

# Reduction Structure & Strategy

## Rules

The strategy of reduction we typically use is often called a *Karp Reduction* (guess why?). In this strategy, **you cannot mess with the oracle**. **All** you get to do is transform an input, then transform an output.

## Proof Structure

With that in mind, a nice Completeness/Hardness answer for problem $A$ usually follows the following structure (as do our rubrics):

1. Prove $A$ in NP **if** we want Completeness
2. State your input mapping and output mapping

# Reduction Structure & Strategy

## Rules

The strategy of reduction we typically use is often called a *Karp Reduction* (guess why?). In this strategy, **you cannot mess with the oracle**. **All** you get to do is transform an input, then transform an output.

## Proof Structure

With that in mind, a nice Completeness/Hardness answer for problem $A$ usually follows the following structure (as do our rubrics):

1. Prove $A$ in NP **if** we want Completeness
2. State your input mapping and output mapping
3. Prove $A = T \implies B = T$ OR $B = F \implies A = F$

# Reduction Structure & Strategy

## Rules

The strategy of reduction we typically use is often called a *Karp Reduction* (guess why?). In this strategy, **you cannot mess with the oracle**. **All** you get to do is transform an input, then transform an output.

## Proof Structure

With that in mind, a nice Completeness/Hardness answer for problem $A$ usually follows the following structure (as do our rubrics):

1. Prove $A$ in NP **if** we want Completeness
2. State your input mapping and output mapping
3. Prove $A = T \implies B = T$ OR $B = F \implies A = F$
4. Prove $A = F \implies B = F$ OR $B = T \implies A = T$

If it seems really intuitive **and** you think it's still rigourous enough, you can combine the last two, but beware!

# Karp's 21 Problems

### A quick bit of history

: In 1971 The Cook-Levin Theorem proves that SAT is NP-Hard from scratch, then in 1972 Karp uses reductions to prove the same for 21 other problems (a year after Edmonds-Karp). They receive a Turing Award each, and get everyone excited.

# Karp's 21 Problems (Exercises)

Clique / Independent Set

- Set packing
- Vertex cover
  - Set covering
  - Feedback node set
  - Feedback arc set
  - Directed Hamilton cycle
    - Undirected Hamilton cycle

Oh, and 0–1 integer programming (A sat-only variation)

3-SAT

- Chromatic number / Coloring
  - Clique cover
  - Exact cover
    - Hitting set
    - Steiner tree
    - 3-D matching
    - Knapsack / Subset Sum
    - - Job sequencing
    - - Partition
    -   - Max Cut

# Approximation Algorithms

Approximation Algorithms are useful for **optimization problems**, not decision problems.

An $f(n)$-approximation algorithm $A$ for a minimization problem has

$$\frac{A}{OPT} \leq f(n)$$

An $f(n)$-approximation algorithm $A$ for a maximization problem has

$$\frac{OPT}{A} \leq f(n)$$

Sometimes, you get problems that are even hard to approximate!

# Online Algorithms

Online Algorithms work on streams of input instead of fixed inputs.

An $f(n)$-competitive algorithm $A$ for a minimization problem has

$$\frac{\mathbb{E}(A)}{OPT} \leq f(n)$$

An $f(n)$-competitive algorithm $A$ for a maximization problem has

$$\frac{\mathbb{E}(A)}{OPT} \geq f(n)$$

$OPT$ here is **not** an online algorithm. It gets the whole string up front!

*Wait, those equations look different.*

# Online Algorithms

Online Algorithms work on streams of input instead of fixed inputs.
An $f(n)$-competitive algorithm $A$ for a minimization problem has

$$\frac{\mathbb{E}(A)}{OPT} \leq f(n)$$

An $f(n)$-competitive algorithm $A$ for a maximization problem has

$$\frac{\mathbb{E}(A)}{OPT} \geq f(n)$$

$OPT$ here is **not** an online algorithm. It gets the whole string up front!

*Wait, those equations look different. How does this make me feel?*

# Online Algorithms

Online Algorithms work on streams of input instead of fixed inputs.
An $f(n)$-competitive algorithm $A$ for a minimization problem has

$$\frac{\mathbb{E}(A)}{OPT} \leq f(n)$$

An $f(n)$-competitive algorithm $A$ for a maximization problem has

$$\frac{\mathbb{E}(A)}{OPT} \geq f(n)$$

$OPT$ here is **not** an online algorithm. It gets the whole string up front!

*Wait, those equations look different. How does this make me feel? Confused.*

# Exercise

### Knapsack 2-Approximation

We have $n$ items, each with a profit $p_i$ and a weight $w_i$. Given a total weight $W$, maximize $\sum p_S \geq P$ by picking $S \subseteq [n]$ with $\sum p_S \leq W$.

# Randomized Marking Mouse

If Mouse follows a deterministic strategy, there is a sequence $S$ of Cat probes that causes

$$\text{MouseCost}(S) \geq (m-1)\text{OPT}(S)$$

$$\text{Paging}\ \boxed{\begin{aligned} m-1 &= \text{cache size} \\ m &= \ \text{different pages} \\ \text{Mouse} &= \text{page not in cache} \\ \text{Cat probes} &= \text{page requests} \\ \text{Must move} &= \text{page fault} \end{aligned}}$$

## Randomized Marking Mouse (RMM)

- Start at random spot
- If Cat probes a spot, mark it
- If Cat probes Mouse's spot,
  Mouse moves to random unmarked spot
- If Mouse is at last unmarked spot, clear marks [phase ends]

# Randomized Marking Mouse performance

Claim: $E[\text{RMMCost}(S)] \le O(\log m)\text{OPT}(S)$

Proof: Initially, RMM is equally likely to be at any of the $m$ spots.

1st probe finds Mouse with probability $1/m$.

Whether Mouse is found or not, Mouse is at each of the $m-1$ unmarked spots with prob. $1/(m-1)$.

2nd probe (to unmarked spot) finds Mouse with prob $1/(m-1)$.

Mouse is at each of the $m-2$ unmarked spots with prob. $1/(m-2)$. Etc.

Let $X_i = \begin{cases} 1 & \text{if Mouse found on } i\text{th probe to unmarked spot} \\ 0 & \text{otherwise} \end{cases}$

$$\begin{aligned} E[\#\text{times found per phase}] &= E[X_1 + X_2 + \cdots + X_m] \\ &\le \frac{1}{m} + \frac{1}{m-1} + \cdots + \frac{1}{1} = O(\log m) \end{aligned}$$

OPT moves once per phase. $\square$

# Is Totally Random Mouse (TRM) better?

TRM runs to a random spot if found.

Consider the Methodical Cat (MC):

- Probe spots 1, 2, 3, ... until Mouse found
- Repeat

What does the OPT mouse do?

# Is Totally Random Mouse (TRM) better?

TRM runs to a random spot if found.

Consider the Methodical Cat (MC):

- Probe spots 1, 2, 3, ... until Mouse found
- Repeat

What does the OPT mouse do? Hide in spot $m$

$E[\#\text{times RM found before MC probes } m] =$
$E[\#\text{rolls of } m\text{-sided dice before } m] = m$

$\Rightarrow$ RM is $m$-competitive.

# Random Marking Mouse is best

Claim: Any Mouse $A$ has $E[A(S)] \in \Omega(\log m)\text{OPT}(S)$

Proof:
Idea: Show that a Cat exists that will cause $E[A(S)] \in \Omega(\log m)$ regardless of the Mouse.

Random Cat (RC) probes a random spot with each probe. RC finds Mouse with prob. $\frac{1}{m}$ no matter what Mouse does.
$\Rightarrow E[A(S)]$ after $t$ probes is $\frac{t}{m}$.

How many RC probes until RC examines every spot?

Coupon Collector Problem $\Rightarrow \Theta(m \log m)$

So OPT Mouse (that knows RC's probes) moves once in sequence $S$ of $\Omega(m \log m)$ probes, while Mouse $A$ moves $E[A(S)] \in \frac{\Omega(m \log m)}{m} = \Omega(\log m)$ times. $\square$

# Universal Families of Hash Functions

A family of hash functions $H$ (that map $U \to \{0, 1, \ldots, m-1\}$) is **universal** if for all distinct keys $x, y \in U$

$$\Pr_{h \in H}[h(x) = h(y)] \leq \frac{1}{m}.$$

## Example

Let $h_{a,b}(x) = ((ax + b) \bmod p) \bmod m$ where $p$ is a prime bigger than any key.

$$H = \left\{ h_{a,b} | a \in \{1, 2, \ldots, p-1\}, b \in \{0, 1, \ldots, p-1\} \right\}$$

# Cuckoo Hashing

## Time per operation

| | |
|---|---|
| Find | $O(1)$ time worst case |
| Delete | $O(1)$ time worst case |
| Insert | $O(1)$ expected, amortized time |

- Use two hash functions $h_1$ and $h_2$.
- Item $x$ will be stored in slot $h_1(x)$ or $h_2(x)$ of hash table $T$.
- Each slot in the hash table can contain at most one item.
- $n =$ maximum number of items stored at any time
- $m =$ size of hash table $T$ $(m > n)$

On an **insert**$(x)$ collision, item $x$ kicks the resident item $y$ out. Item $y$ then goes to its alternate slot (kicking whoever's there out). Etc. Etc.

# RSA public/private key cryptosystem [Rivest,Shamir,Adleman '77]

Bob has two functions: secret $S_B()$ and public $P_B()$

Properties:

1. $S_B(P_B(M)) = M$ and $P_B(S_B(M)) = M$
2. Hard to find $M$ given $P_B(M)$ without $S_B()$

   Alice sends $P_B(M)$ to Bob.
   Bob decrypts: $S_B(P_B(M)) = M$

Good: Use again and again
Bad: No one knows if it's secure.

factoring easy $\Rightarrow$ RSA breakable.
factoring hard $\Rightarrow$ RSA secure? (unknown)

Digital Signatures: Alice sends $(M, \sigma = S_A(M)$ to Bob
Bob can check that $P_A(\sigma) = M$.

# Quantum Computing

## Qubits

A *qubit* is a combination $|0\rangle$ and $|1\rangle$:

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$$

$\alpha$ and $\beta$ are both complex numbers here, but since it must be that $\alpha^2 + \beta^2 = 1$, we actually only have 3 degrees of freedom.

## Gates

Quantum gates, at the end of the day, are unitary matrices. For example, Hadamard is $\frac{1}{\sqrt{2}}\begin{pmatrix}1 & 1\\ 1 & -1\end{pmatrix}$.

# Zero Knowledge

The key idea here, is prove your identity without giving away any of your secrets.
If there's time, a quick example: Colour-Blindness [Credits to Wikipedia]

# Closing Remarks

Any Questions?