Graph Coloring

• Instance: A graph $G = (V, E)$ and an integer $k$.
• Question: Can $G$ be "properly colored" with at most $k$ colors?

• Proper Coloring: one of the $k$ colors is assigned to each vertex so that adjacent vertices have different colors.

• Suppose we have two instances of this problem (1) is True (Yes) and the other (2) is False (No).

• AND you know (1) is Yes and (2) is No. (Maybe you have a secret program that has analyzed the two instance.)
Checking a “Yes” Answer

- Without showing how your program works (you may not even know), how can you convince someone else that instance (1) is, in fact, a Yes instance?
- We can assume the output of the program was an actual coloring of $G$. Just give that to a doubter who can easily check that no adjacent vertices are colored the same, and that no more than $k$ colors were used.

- How about the No instance?

- What could the program have given that allows us to quickly "verify" (2) is a No instance?
  - No One Knows!!

- For all problems seem to be harder than there exists ones in many contexts
- Think re versus co-re
Checking a “No” Answer

• The only thing anyone has thought of is to have it test all possible ways to \( k \)-color the graph – all of which fail, of course, if “No” is the correct answer.
• There are an exponential number of things (colorings) to check.
• For some problems, there seems to be a big difference between verifying Yes and No instances.
• To solve a problem efficiently, we must be able to solve both Yes and No instances efficiently and so it would seem we should be able to verify both quickly.
Hard and Easy

• **True Conjecture:** If a problem is easy to solve, then it is easy to verify (just solve it and compare).

• **Contrapositive:** If a problem is hard to verify, then it is (probably) hard to solve.

• There is nothing magical about Yes and No instances – sometimes the Yes instances are hard to verify and No instances are easy to verify.

• And, of course, sometimes both are hard to verify.
Easy Verification

• Are there problems in which both Yes and No instances are easy to verify?

• Yes. For example: Search a list $L$ of $n$ values for a key $x$.

• Question: Is $x$ in the list $L$?

• Yes and No instances are both easy to verify.

• In fact, the entire problem is easy to solve!!

• Analogy: recursive means re and co-re
Verify vs Solve

- **Conjecture**: If both Yes and No instances are easy to verify, then the problem is easy to solve.
- **Analogy (Failure?)**: Does re/co-re imply recursive have analogy here?

- No one has yet proven this claim, but most researchers believe it to be true.
- Note: It is usually relatively easy to prove something is easy – just write an algorithm for it and prove it is correct and that it is fast (usually, we mean polynomial).
- But it is usually very difficult to prove something is hard – we may not be clever enough yet. So, you will often see "appears to be hard."
A CS Grand Challenge

Does $P=NP$?

There are many equivalent ways to describe $P$ and $NP$. For now, we will use the following.

$P$ is the set of decision problems (those whose instances have “Yes”/“No” answers) that can be solved in polynomial time on a deterministic computer (no concurrency or guesses allowed).

$NP$ is the set of decision problems that can be solved in polynomial time on a non-deterministic computer (equivalently one that can spawn an unbounded number of parallel threads; equivalently one that can be verified in polynomial time on a deterministic computer).

Again, as “Does $P=NP$?” has just one question, it is solvable, we just don’t yet know which solution, “Yes” or “No”, is the correct one.
Decision vs Optimization

Two types of problems are of particular interest:

Decision Problems ("Yes/No" answers)

Optimization problems ("best" answers)

(there are other types)
Vertex Cover (VC)

• Suppose we are in charge of a large network (a graph where edges are links between pairs of cities (vertices). Periodically, a line fails. To mend the line, we must call in a repair crew that goes over the line to fix it. To minimize down time, we station a repair crew at one end of every line. How many crews must you have and where should they be stationed?

• This is called the Vertex Cover Problem. (Yes, it sounds like it should be called the Edge Cover problem – something else already had that name.)

• An interesting problem – it seems to a hard problems but may not be.
VC Decision vs Optimization

• As a Decision Problem:
  – Instances: A graph $G$ and an integer $k$.
  – Question: Does $G$ possess a vertex Cover with at most $k$ vertices?

• As an Optimization Problem:
  – Instances: A graph $G$.
  – Question: What is the smallest $k$ for which $G$ possesses a vertex cover?
Relation of VC Problems

• If we can (easily) solve either one of these problems, we can (easily) solve the other. To solve the optimization version, just solve the decision version with several different values of $k$. Use a binary search on $k$ between 1 and $n$. That is $\log(n)$ solutions of the decision problem solves the optimization problem. It's simple to solve the decision version if we can solve the optimization version.

• We say their time complexity differs by no more than a multiple of $\log(n)$.
• If one is polynomial, then so is the other.
• If one is exponential, then so is the other.

• We say they are equally difficult (both poly. or both exponential).
A Word about Time

An algorithm for a problem is said to be polynomial if there exists integers $k$ and $N$ such that $t(n)$, the maximum number of steps required on any instance of size $n$, is at most $n^k$, for all $n \geq N$.

Otherwise, we say the algorithm is exponential. Usually, this is interpreted to mean $t(n) \geq c^n$ for an infinite set of size $n$ instances, and some constant $c > 1$ (often, we simply use $c = 2$).
Three Classes of Problems

Problems proven to be in these three groups (classes) are, respectively,

Undecidable, Exponential, and Polynomial.

Theoretically, all problems belong to exactly one of these three classes.
Unknown Complexity

Practically, there are a lot of problems (maybe, most) that have not been proven to be in any of the classes (Yet, maybe never will be).

Most currently "lie between" polynomial and exponential – we know of exponential algorithms but have been unable to prove that exponential algorithms are necessary.

Some may have polynomial algorithms, but we have not yet been clever enough to discover them.
Subset Sum

• Problem – Subset Sum

• Instances: A list \( L \) of \( n \) integer values and an integer \( B \).
• Question: Does \( L \) have a subset which sums exactly to \( B \)?

• No one knows of a polynomial (deterministic) solution to this problem.

• On the other hand, there is a very simple (dynamic programming) algorithm that runs in \( O(nB) \) time.

• Why isn't this "polynomial"?
  – Because the "length" of an instance is \( n \log(B) \) and
  – \( nB > (n \log(B))^k \) for any fixed \( k \).
Research Territory

Decidable – vs – Undecidable
(area of Computability Theory)

Exponential – vs – polynomial
(area of Computational Complexity)

Algorithms for any of these
(area of Algorithm Design/Analysis)
NonDeterminism
Non-Determinism

Since we can't seem to find a model of computation that is more powerful than a TM, can we find one that is 'faster'?

In particular, we want one that takes us from exponential time to polynomial time.

Our candidate will be the Non-Deterministic Turing Machine (NDTM).
As NDTMs are automata, we only care about what they accept and generally have just one accepting state. We want to determine how long it takes to get to the accept state - that's our only motive!!

So, what is a NDTM doing? In a NDTM, like in a NFA or NPDA, what we are doing is making, and testing, all of those choices at once by 'spawning' a different NDTM for each of them. Those that don't work out, simply die (or something).

This is the ultimate in parallel programming.
NDTM and Oracles

Another interpretation of non-determinism:
From the basic definition, we notice that out of every state having a non-deterministic choice, at least one choice is valid and all the rest sort of die off. That is, they really have no reason for being spawned for this instance. So, we station at each such state, an 'oracle' (an all-knowing being) who only allows the correct NDTM to be spawned.

An 'Oracle Machine.'
All we care about is the time associated with the right choices. In fact, we can have a single oracle that makes all the right choices in advance, if they exist.
Checking an Oracle/Witness

For example, in the SubsetSum problem, we ask the oracle to write down the subset of objects whose sum is B (the desired sum). Then we ask "Can we write a deterministic polynomial algorithm to test the given witness."

The answer for SubsetSum is Yes, we can, i.e., the witness is verifiable in deterministic polynomial time.
NDTMs - Witnesses

Just what can we ask and expect of a "witness"?

The witness must be something that

(1) we can verify to be accurate (for the given problem and instance) and

(2) we must be able to "finish off" the solution.

All in polynomial time.
Witness and Graph Coloring

The information provided must be something we could have come up with ourselves, but probably at an exponential cost. And, it has to be enough so that we can conclude the final answer Yes from it.

Consider a witness for the graph coloring problem:

Given: A graph $G = (V, E)$ and an integer $k$.
Question: Can the vertices of $G$ be assigned colors so that adjacent vertices have different colors and use at most $k$ colors?
Good and Bad Witnesses

The witness could just say Yes or No.
   
   But that's not good enough - we don't know of a polynomial algorithm for graph coloring.

It could be "vertex 10 is colored Red."

   That's not good enough either. Any single vertex can be colored any color we want.

It could be a color assigned to each vertex.

   That would work, because we can verify its validity in polynomial time, and we can conclude the correct answer of Yes.
More on Witnesses

What if it was a color for all vertices but one?

That also is enough. We can verify the correctness of the \( n-1 \) given to us, then we can verify that the one uncolored vertex can be colored with a color not on any neighbor, and that the total is not more than \( k \).

What if all but 2, 3, or 20 vertices are colored

All are valid witnesses. (Potentially exponential but for a constant, not a variable \( n \))

What if half the vertices are colored?

Usually, No. There's not enough information. Sure, we can check that what is given to us is properly colored, but we don't know how to "finish it off" except perhaps in \( k^{n/2} \) time.
Deterministic Exponential Time

A major question remains: Do we have, in NDTMs, a model of computation that solves all deterministic exponential (DE) problems in polynomial time (nondeterministic polynomial time)??

It definitely solves some problems we think are DE in nondeterministic polynomial time.
DEXP Versus NEXP

But, so far, all problems that have been proven to require deterministic exponential time also require nondeterministic exponential time.

So, the jury is still out. In the meantime, NDTMs are still valuable, because they might identify a larger class of problems than does a deterministic TM - the set of decision problems for which Yes instances can be verified in polynomial time.
Problem Classes

We now begin to discuss several different classes of problems. The first two will be:

- **NP**: 'Nondeterministic' Polynomial
  - The 'easiest' problems in NP
- **P**: 'Deterministic' Polynomial,
  - The 'easiest' problems in NP

Their definitions are rooted in the depths of Computability Theory as just described, but it is worth repeating some of it in the next few slides.
Class – NP

First Significant Complexity
Class of Problems
Consider two seemingly closely related statements (versions) of a single problem. We show they are actually very different. Let $G = (V, E)$ be a graph.

**Definition:** $X \subseteq V(G)$ is a *vertex cover* if every edge in $G$ has at least one endpoint in $X$. 
Variants of VC

**Version 1.** Given a graph $G$ and an integer $k$. Does $G$ contain a vertex cover with at most $k$ vertices?

**Version 2.** Given a graph $G$ and an integer $k$. Does the smallest vertex cover of $G$ have exactly $k$ vertices?

Suppose, for either version, the answer is "yes," and someone also gives us a set $X$ of vertices and claims "$X$ satisfies the conditions."
Version 1 of VC

In Version 1, we can easily check that the claim is correct – in polynomial time.

That is, in polynomial time, we can check that $X$ has $k$ vertices, and that $X$ is a vertex cover.
Version 2 of VC

In **Version 2**, we can also easily check that $X$ has exactly $k$ vertices and that $X$ is a vertex cover.

But we don't know how to easily check that there is not a smaller vertex cover!!

That seems to require exponential time.

These are very similar looking "decision" problems (Yes/No answers), yet they are VERY different in this one important respect.
**Version 1 in NP**

**Version 1** problems make up the class called **NP**

**Definition:** The *Class NP* is the set of all decision problems for which answers to Yes instances can be verified in polynomial time.

For historical reasons, **NP** means "**Nondeterministic Polynomial.**"

*(Specifically, it does not mean "not polynomial").*
Version 3 of VC

Version 2 of the Vertex Cover problem is not unique. There are other versions that exhibit this same property. For example,

Version 3: Given: A graph \( G = (V, E) \) and an integer \( k \).

Question: Do all vertex covers of \( G \) have more than \( k \) vertices?

What would/could a 'witness' for a Yes instance be?

Version 3 is the Complement Problem of Version 1. It is a “for all not” versus a “there exist.”
Characteristics of NP

All problems in NP are **decidable**.

That means there is an algorithm.

And the algorithm is no worse than $O(2^n)$. 
Non-NP

Version 2 and 3 problems are apparently not in NP.

So, where are they??

We need more structure! {Again, later.}

First we look inward, within NP.
Class – P
Second Significant Complexity
Class of Problems
Some decision problems in \textbf{NP} can be solved (without knowing the answer in advance) - in polynomial time. That is, not only can we verify a correct answer in polynomial time, but we can actually \textit{compute} the correct answer in polynomial time - from "scratch."

These are the problems that make up the class \textbf{P}.

\textbf{P} is a subset of \textbf{NP}. 
A Witness Analogy for $\mathbf{P}$

Problems in $\mathbf{P}$ can also have a witness – we just don't need one. This line of thought leads to an interesting observation.

**Given:** A list $L$ of $n$ values and a key $X$.

**Question:** Is $X$ in $L$?

An oracle can provide a "witness" for a Yes instance by writing the index $k$ for $X$.

We can verify the correctness with one simple comparison Is $L[k] = X$?
Complement of P

Now, consider the complement (Version 3) of this problem:

Given: A list $L$ of $n$ values and a key $X$.

Question: Is $X$ not in $L$?

Here, for any Yes instance, no 'witness' seems to exist, but if the oracle simply writes down "Yes" we can verify the correctness in polynomial time by comparing $X$ with each of the $n$ values and report "Yes, $X$ is not in the list" if that is so.
P and Co-P

Therefore, both problems can be verified in polynomial time and, hence, both are in \( \text{NP} \).

This is a characteristic of any problem in \( \text{P} \) - both it and its complement can be verified in polynomial time (of course, they can both be 'solved' in polynomial time, too.)

Therefore, we can again conclude \( \text{P} \subseteq \text{NP} \).
There is a popular conjecture that if any problem and its complement are both in $\text{NP}$, then both are also in $\text{P}$. This has been the case for several problems that for many years were not known to be in $\text{P}$, but both the problem and its complement were known to be in $\text{NP}$.

For example, Linear Programming (proven to be in $\text{P}$ in the 1980’s).

A notable 'holdout' to date is Graph Isomorphism.
Is $P=NP$?

At the moment, no one knows.

Some believe all problems in $NP$ have polynomial algorithms. Many do not (believe that).

The fundamental question in theoretical computer science is:
Does $P = NP$?

There is an award of one million dollars for a proof.
– Either way, True or False.
The "Key" to Complexity Theory

'Reductions,'
'Reductions,'
'Reductions,'
Abstract Solution

For any problem $X$, let $X(I_X, \text{Answer}_X)$ represents an algorithm for problem $X$ – even if none is known to exist.

$I_X$ is an arbitrary instance given to the algorithm and $\text{Answer}_X$ is the returned answer determined by the algorithm.
Polynomial Time Reductions

**Definition:** For problems $A$ and $B$, a *(Polynomial) Turing Reduction* is an algorithm $A(I_A, \text{Answer}_A)$ for solving all instances of problem $A$ and satisfies the following:

1. Constructs zero or more instances of problem $B$ and invokes algorithm $B(I_B, \text{Answer}_B)$, on each.
2. Computes the result, $\text{Answer}_A$, for $I_A$.
3. Except for the time required to execute algorithm $B$, the execution time of algorithm $A$ must be polynomial with respect to the size of $I_A$.

We say $A \leq_P B$
Best Algorithm for B

We may assume a 'best' algorithm for problem B without actually knowing it.

If \( A(I_A, \text{Answer}_A) \) can be written without algorithm B, then problem A is simply a polynomial problem.
PolyTime Reductions

**Theorem.** If $A \leq_p B$ and problem $B$ is polynomial, then problem $A$ is polynomial.

**Corollary.** If $A \leq_p B$ and problem $A$ is exponential, then problem $B$ is exponential.
**PT Reduction Theorem**

**Theorem.** If $A \leq_p B$, then problem $A$ is "no harder than" problem $B$.

**Proof:** Let $f_X(n)$ be the inherent complexity of problem $X$. Let $t_A(n)$ and $t_B(n)$ be the maximum times for some algorithms to solve $A$ and $B$. Thus, $f_A(n) \leq t_A(n)$. Further, since we assume the best algorithm for $B$, $t_B(n) = f_B(n)$. Since $A \leq_p B$, there is a constant $k$ such that $t_A(n) \leq n^k t_B(n)$. Therefore, $f_A(n) \leq t_A(n) \leq n^k t_B(n) = n^k f_B(n)$. That is, $A$ is no harder than $B$ within a polynomial factor.
PT Reductions Properties

Theorem. (transitivity)
If $A \leq_p B$ and $B \leq_p C$ then $A \leq_p C$.

Definition.
If $A \leq_p B$ and $B \leq_p A$, then $A$ and $B$ are polynomially equivalent. $A \equiv_p B$

Note reflexive as $A \leq_p A$
NP–Complete

Third Significant Complexity Class of Problems
NP–Complete

• Polynomial Transformations enforce an equivalence relationship on all decision problems, particularly, those in the Class NP. Class P is one of those classes and is the "easiest" class of problems in NP.

• Is there a class in NP that is the hardest class in NP?

• A problem B in NP such that \( A \leq_p B \) for every A in NP is called NP-Complete (Analogy to re-complete)
NP–Hard

• A problem $B$ such that $A \leq_p B$ for every $A$ in NP-Complete is called NP-Hard (Second Analogy to re-hard)
Propositional Logic

How Hard Can That Be?
Propositional Calculus

- Mathematical of unquantified logical expressions
- Essentially Boolean algebra
- Goal is to reason about propositions
- Often interested in determining
  - Is a well-formed formula (wff) a tautology?
  - Is a wff refutable (unsatisfiable)?
  - Is a wff satisfiable? (will show this is the canonical NP-complete problem)
Tautology and Satisfiability

- The classic approaches are:
  - Truth Table
  - Axiomatic System (axioms and inferences)

- Truth Table
  - Clearly exponential in number of variables

- Axiomatic Systems Rules of Inference
  - Substitution and Modus Ponens
  - Resolution / Unification (1st order only)
Proving Consequences

• Start with a set of axioms (all tautologies)
• Using substitution and MP
  \((P, P \implies Q \implies Q)\)
  derive consequences of axioms (also
tautologies, but just a fragment of all
unless axioms are “complete”)
• Can create complete sets of axioms
• Need 3 variables for associativity, e.g.,
  \((p1 \lor p2) \lor p3 \implies p1 \lor (p2 \lor p3)\)
Some Undecidables

• Given a set of axioms,
  – Is this set complete?
  – Given a tautology $T$, is $T$ a consequent?
• The above are even undecidable with one axiom and with only 2 variables. I will show the latter result shortly.
Refutation

• If we wish to prove that some wff, F, is a tautology, we could negate it and try to prove that the new formula is refutable (cannot be satisfied; contains a logical contradiction).

• This is often done using resolution.
Resolution

• Put formula in Conjunctive Normal Form (CNF)

• If have terms of conjunction
  \((P \lor Q), (R \lor \sim Q)\)
  then can determine that \((P \lor R)\)

• If we ever get a null conclusion, we have refuted the proposition

• Resolution is not complete for derivation, but it is for refutation
Axioms

- Must be tautologies
- Can be incomplete
- Might have limitations on them and on WFFs, e.g.,
  - Just implication
  - Only n variables
  - Single axiom
Simulating Machines

- Linear representations require associativity, unless all operations can be performed on prefix only (or suffix only).
- Prefix and suffix-based operations are single stacks and limit us to CFLs.
- Can simulate Post normal Forms with just 3 variables. A PNF has rules $\alpha P \rightarrow P\beta$. 
Diadic PIPC

• Diadic limits us to two variables
• PIPC means Partial Implicational Propositional Calculus, and limits us to implication as only connective
• Partial just means we get a fragment
• Problems
  – Is fragment complete?
  – Can F be derived by substitution and MP?
Living without Associativity

• Consider a two-stack model of a TM
• Could somehow use one variable for left stack and other for right
• Must find a way to encode a sequence as a composition of forms – that’s the key to this simulation
Composition Encoding

• Consider \((p \supset p), \ (p \supset (p \supset p)), \ (p \supset (p \supset (p \supset p))), \ldots\)
  – No form is a substitution instance of any of the other, so they can’t be confused
  – All are tautologies

• Consider \(((X \supset Y) \supset Y)\)
  – This is just \(X \lor Y\)
Encoding

- Use \((p \supset p)\) as form of bottom of stack
- Use \((p \supset (p \supset p))\) as form for letter 0
- Use \((p \supset (p \supset (p \supset p)))\) as form for 1
- Etc.
- String 01 (reading top to bottom of stack) is
  
  \[
  \neg ( ( (p \supset p) \supset ( (p \supset p) \supset ( (p \supset p) \supset (p \supset p) ) ) ) ) \supset \\
  ( ( (p \supset p) \supset ( (p \supset p) \supset ( (p \supset p) \supset (p \supset p) ) ) ) ) \supset \\
  ( (p \supset p) \supset ( (p \supset p) \supset ( (p \supset p) \supset (p \supset p) ) ) ) )
  \]
Encoding

$I(p)$ abbreviates $[p \supset p]$

$\Phi_0(p)$ is $[p \supset I(p)]$ which is $[p \supset [p \supset p]]$

$\Phi_1(p)$ is $[p \supset \Phi_0(p)]$

$\xi_1(p)$ is $[p \supset \Phi_1(p)]$

$\xi_2(p)$ is $[p \supset \xi_1 (p)]$

$\xi_3(p)$ is $[p \supset \xi_2 (p)]$

$\psi_1(p)$ is $[p \supset \xi_3 (p)]$

$\psi_2(p)$ is $[p \supset \psi_1 (p)]$

$\ldots$

$\psi_m(p)$ is $[p \supset \psi_{m-1} (p)]$
Creating Terminal IDs

1. \([\xi_1 I(p_1) \lor I(p_1)]\).
2. \([\xi_1 I(p_1) \lor I(p_1)] \Rightarrow [\xi_1 I(p_1) \lor \Phi_1 I(p_1)]\).
3. \([\xi_1 I(p_1) \lor \Phi_i(p_2)] \Rightarrow [\xi_1 I(p_1) \lor \Phi_j \Phi_i(p_2)], \forall i, j \in \{0, 1\}\).
4. \([\xi_1 I(p_1) \lor p_2] \Rightarrow [\xi_2 \Phi_1 I(p_1) \lor p_2]\).
5. \([\xi_1 I(p_1) \lor p_2] \Rightarrow [\xi_3 \Phi_i I(p_1) \lor p_2], \forall i \in \{0, 1\}\).
6. \([\xi_2 \Phi_i(p_1) \lor p_2] \Rightarrow [\xi_2 \Phi_j \Phi_i(p_1) \lor p_2], \forall i, j \in \{0, 1\}\).
7. \([\xi_2 \Phi_i(p_1) \lor p_2] \Rightarrow [\xi_3 \Phi_j \Phi_i(p_1) \lor p_2], \forall i, j \in \{0, 1\}\).
8. \([\xi_3 \Phi_i(p_1) \lor p_2] \Rightarrow [\Psi_k \Phi_i(p_1) \lor p_2], \text{ whenever } q_k a_i \text{ is a terminal discriminant of } M\).
9. \([\Psi_k \Phi_i(p_1) \lor p_2] \Rightarrow [\Psi_h \Phi_j(p_1) \lor p_2]\), whenever \(q_h a_j a_i q_k \in T\).

10a. \([\Psi_k \Phi_0 I(p_1) \lor I(p_1)] \Rightarrow [\Psi_h \Phi_0 I(p_1) \lor I(p_1)]\),

b. \([\Psi_k \Phi_1 I(p_1) \lor I(p_1)] \Rightarrow [\Psi_h \Phi_0 I(p_1) \lor \Phi_1(p_1)]\),

c. \([\Psi_k \Phi_i I(p_1) \lor \Phi_j(p_2)] \Rightarrow [\Psi_h \Phi_0 I(p_1) \lor \Phi_i \Phi_j(p_2)]\),

d. \([\Psi_k \Phi_0 \Phi_0 \Phi_i(p_1) \lor I(p_2)] \Rightarrow [\Psi_h \Phi_0 \Phi_i(p_1) \lor I(p_2)]\),

e. \([\Psi_k \Phi_1 \Phi_0 \Phi_i(p_1) \lor I(p_2)] \Rightarrow [\Psi_h \Phi_0 \Phi_i(p_1) \lor \Phi_1 I(p_2)]\),

f. \([\Psi_k \Phi_i \Phi_0 \Phi_j(p_1) \lor \Phi_m(p_2)] \Rightarrow [\Psi_h \Phi_0 \Phi_j(p_1) \lor \Phi_i \Phi_m(p_2)]\),
\forall i, j, m \in \{0, 1\} whenever \(q_h 0Lq_k \in T\).

11a. \([\Psi_k \Phi_0 \Phi_1(p_1) \lor I(p_2)] \Rightarrow [\Psi_h \Phi_1(p_1) \lor I(p_2)]\),

b. \([\Psi_k \Phi_1 \Phi_1(p_1) \lor I(p_2)] \Rightarrow [\Psi_h \Phi_1(p_1) \lor \Phi_1 I(p_2)]\),

c. \([\Psi_k \Phi_i \Phi_1(p_1) \lor \Phi_j(p_2)] \Rightarrow [\Psi_h \Phi_1(p_1) \lor \Phi_i \Phi_j(p_2)]\),
\forall i, j \in \{0, 1\} whenever \(q_k 1Lq_k \in T\).
Reversing Right

12a. $[\Psi^t \Phi_0 I(p_1) \lor I(p_1)] \supset [\Psi^h \Phi_0 I(p_1) \lor I(p_1)],$

b. $[\Psi^t \Phi_0 I(p_1) \lor \Phi_0 \Phi_i(p_2)] \supset [\Psi^h \Phi_0 I(p_1) \lor \Phi_i(p_2)],$

c. $[\Psi^t \Phi_1(p_1) \lor I(p_2)] \supset [\Psi^h \Phi_0 \Phi_1(p_1) \lor I(p_2)],$

d. $[\Psi^t \Phi_0 \Phi_i(p_1) \lor I(p_2)] \supset [\Psi^h \Phi_0 \Phi_0 \Phi_i(p_1) \lor I(p_2)],$

e. $[\Psi^t \Phi_0 \Phi_i(p_1) \lor \Phi_0 \Phi_j(p_2)] \supset [\Psi^h \Phi_0 \Phi_0 \Phi_i(p_1) \lor \Phi_j(p_2)],$

f. $[\Psi^t \Phi_1(p_1) \lor \Phi_0 \Phi_i(p_2)] \supset [\Psi^h \Phi_0 \Phi_1(p_1) \lor \Phi_i(p_2)],$

\[ \forall i, j \in \{0, 1\} \text{ whenever } q_h 0 R q_k \in T. \]

13a. $[\Psi^t \Phi_0 I(p_1) \lor \Phi_1(p_2)] \supset [\Psi^h \Phi_1 I(p_1) \lor p_2],$

b. $[\Psi^t \Phi_1(p_1) \lor \Phi_1(p_2)] \supset [\Psi^h \Phi_1 I(p_1) \lor p_2],$

c. $[\Psi^t \Phi_0 \Phi_i(p_1) \lor \Phi_1(p_2)] \supset [\Psi^h \Phi_1 \Phi_0 \Phi_i(p_1) \lor p_2]$

\[ \forall i \in \{0, 1\} \text{ whenever } q_h 1 R q_k \in T. \]
Satisfiability

How Hard Can That Be?
Conjunctive Normal Form

U = \{u_1, u_2, \ldots, u_n\}, Boolean variables.

C = \{c_1, c_2, \ldots, c_m\}, "OR clauses"

For example:

\[ c_i = (u_4 \lor u_{35} \lor \neg u_{18} \lor u_3 \ldots \lor \neg u_6) \]
Satisfiability Challenge

Can we assign Boolean values to the variables in $U$ so that every clause is $\text{TRUE}$?

There is no known polynomial time algorithm!!
Cook’s Theorem

_Cooks Theorem:_

1) SAT is in NP
2) For every problem A in NP, $A \leq_p SAT$

Thus, SAT is as hard as every problem in NP.
SAT as NP–Complete

Since SAT is itself in NP, that means SAT is a hardest problem in NP (there can be more than one.).

A hardest problem in a class is called the "completion" of that class.

Therefore, SAT is NP–Complete.
Today, there are 1,000’s of problems that have been proven to be NP–Complete. (See Garey and Johnson, Computers and Intractability: A Guide to the Theory of NP–Completeness, for a list of over 300 as of the early 1980's).
What about $P = NP$?

If $P = NP$ then all problems in NP are polynomial problems.

If $P \neq NP$ then all NP–C problems are at least super-polynomial and perhaps exponential. That is, NP-C problems could require sub-exponential super-polynomial time. (Example of super-polynomial, sub-exponential is $o(2^{o(n)})$, e.g., $2^{\sqrt[3]{n}}$)
Evidence for $P = NP$?

Why should $P$ equal $NP$?

- There seems to be a huge "gap" between the known problems in $P$ and Exponential. That is, almost all known polynomial problems are no worse than $n^3$ or $n^4$.
- Where are the $O(n^{50})$ problems?? $O(n^{100})$? Maybe they are the ones in $NP$–Complete?
- It's awfully hard to envision a problem that would require $n^{100}$, but surely they exist?
- Some of the problems in $NP$–C just look like we should be able to find a polynomial solution (looks can be deceiving, though).
Why should P not equal NP?

• P = NP would mean, for any problem in NP, that it is just as easy to solve an instance form "scratch," as it is to verify the answer if someone gives it to you. That seems a bit hard to believe.

• There simply are a lot of awfully hard looking problems in NP–Complete (and Co–NP-Complete) and some just don't seem to be solvable in polynomial time.

• Many smart people have tried for a long time to find polynomial algorithms for some of the problems in NP-Complete - with no luck.
A decision problem, \( C \), is NP-complete if:

- \( C \) is in NP and
- \( C \) is NP-hard. That is, every problem in NP is polynomially reducible to \( C \).

\( D \) polynomially reduces to \( C \) means that there is a deterministic polynomial-time many-one algorithm, \( f \), that transforms each instance \( x \) of \( D \) into an instance \( f(x) \) of \( C \), such that the answer to \( f(x) \) is YES if and only if the answer to \( x \) is YES.

To prove that an NP problem \( A \) is NP-complete, it is sufficient to show that an already known NP-complete problem polynomially reduces to \( A \). By transitivity, this shows that \( A \) is NP-hard.

A consequence of this definition is that if we had a polynomial time algorithm for any NP-complete problem \( C \), we could solve all problems in NP in polynomial time. That is, \( P = NP \).

Note that NP-hard does not necessarily mean NP-complete, as a given NP-hard problem could be outside NP. Analogy to re-complete...
Returning to SAT

• SAT is the problem to decide of an arbitrary Boolean formula (wff in the propositional calculus) whether or not this formula is satisfiable (has a set of variable assignments that evaluate the expression to true).

• SAT clearly can be solved in time $k2^n$, where $k$ is the length of the formula and $n$ is the number of variables in the formula.

• What we now show is that SAT is NP-complete, providing us our first concrete example of an NP-complete decision problem.
Simulating NDTM

- Given a NDTM, $M$, and an input $w$, we need to create a formula, $\varphi_{M,w}$, containing a polynomial number of terms that is satisfiable just in case $M$ accepts $w$ in polynomial time.

- The formula must encode within its terms a trace of configurations that includes
  - A term for the starting configuration of the TM
  - Terms for all accepting configurations of the TM
  - Terms that ensure the consistency of each configuration
  - Terms that ensure that each configuration after the first follows from the prior configuration by a single move
A **tableau** is an array of tape alphabet symbols.

It represents a configuration history of **one branch** of our NDTM’s nondeterminism.

If the NDTM runs in $n^k$ time, the tableau is an $(n^k \times n^k)$ tableau.

- It’s big enough downward because, well, the TM runs in $n^k$.
- …and rightward because the TM can only count to $n^k$.

We assume that every configuration starts and ends with a # symbol.

We think of our tableau as looking like this in the “beginning”: the starting configuration across the top, and the other configurations blank.

(We quote “beginning” because SAT isn’t really a stateful algorithm, but just go with it for now.)

But we’ve assumed that we can “represent” alphabet symbols. How do we do that, in SAT?

<table>
<thead>
<tr>
<th></th>
<th>$q_0$</th>
<th>$w_1$</th>
<th>$w_2$</th>
<th>…</th>
<th>$w_n$</th>
<th>□</th>
<th>…</th>
<th>□</th>
<th>#</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>
</tr>
<tr>
<td>#</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>…</td>
<td>#</td>
<td>…</td>
<td>□</td>
<td>#</td>
<td>#</td>
</tr>
<tr>
<td>#</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>…</td>
<td>#</td>
<td>…</td>
<td>□</td>
<td>#</td>
<td>#</td>
</tr>
<tr>
<td>#</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>…</td>
<td>#</td>
<td>…</td>
<td>□</td>
<td>#</td>
<td>#</td>
</tr>
<tr>
<td>#</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>…</td>
<td>#</td>
<td>…</td>
<td>□</td>
<td>#</td>
<td>#</td>
</tr>
<tr>
<td>#</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>…</td>
<td>#</td>
<td>…</td>
<td>□</td>
<td>#</td>
<td>#</td>
</tr>
<tr>
<td>#</td>
<td>#</td>
<td>#</td>
<td>#</td>
<td>…</td>
<td>#</td>
<td>…</td>
<td>□</td>
<td>#</td>
<td>#</td>
</tr>
</tbody>
</table>

↑$n^k$↓

← $n^k$ →
Encoding the Tableau: Basics

Consider a set comprised of:
- The tape alphabet
- The state set
- The separator character

\[ C = \Gamma \cup Q \cup \{ \# \} \]

Consider a cell variable:

\[ x_{i,j,c} \]

*Turning this variable on* corresponds to *setting cell* \((i, j) = c\), for some \( c \in C \).
Encoding the Tableau: Cells

Consider our tableau alphabet:

\[ C = \Gamma \cup Q \cup \{ \# \} \]

Consider a cell and corresponding variable:

\[ x_{i,j,c} \]

Now we need to make sure the tableau is consistently encoded.

Create a clause for each cell \((i, j)\).

\[
\phi_{\text{encode}}(i, j) = \left( \bigvee_{c \in C} x_{i,j,c} \right) \land \left( \bigwedge_{c,d \in C} \left( \overline{x_{i,j,c}} \lor \overline{x_{i,j,d}} \right) \right)
\]

The left demands \( x_{i,j,c} \) be true for some \( c \).
The right demands \( x_{i,j,c} \) be true for only one \( c \).
**Encoding the Tableau: The Tableau**

Tableau alphabet: \( C = \Gamma \cup Q \cup \{ \# \} \)

Cell variable: \( x_{i,j,c} \)

Create an encoding clause for each cell \((i, j)\).

\[
\phi_{\text{encode}}(i, j) = \left( \bigvee_{c \in C} x_{i,j,c} \right) \land \left( \bigwedge_{c,d \in C, c \neq d} (x_{i,j,c} \lor \overline{x_{i,j,d}}) \right)
\]

Now repeat the clause across the tableau.

\[
\phi_{\text{cells}} = \bigwedge_{1 \leq i, j \leq n^k} \phi_{\text{encode}}(i, j)
\]

This is our *cell formula*. It ensures that each cell in the tableau is assigned a single symbol.
Encoding the Tableau: Complexity

We can create the single-cell encoding formula in polynomial time with a $|C|^2$ iteration.

$$\phi_{\text{encode}}(i, j) = \left[ \left( \bigvee_{c \in C} x_{i,j,c} \right) \land \left( \bigwedge_{c,d \in C, c \neq d} (\overline{x}_{i,j,c} \lor x_{i,j,d}) \right) \right]$$

We can create the entire cell formula in polynomial time with an $n^{2k}$ iteration around that.

$$\phi_{\text{cells}} = \bigwedge_{1 \leq i,j \leq n^k} \phi_{\text{encode}}(i, j)$$

So we can say that $\phi_{\text{cells}}$ is satisfied by, and only by, a properly encoded tableau, and is created in polynomial time.
Starting and accepting are (comparatively) easy. To start, take the start configuration padded to $n^k$ length with blanks…

$S = \#q_0w_1w_2\ldots w_n\square\ldots\square\#$ so that $|S| = n^k$

…and require the first row be equal to the start configuration:

$$\phi_{\text{start}} = \bigwedge_{1 \leq j \leq n^k} [x_1, j, s_j]$$

Then to accept, just require an accept state somewhere in the tableau.

$$\phi_{\text{accept}} = \bigvee_{1 \leq i, j \leq n^k} [x_i, j, q_A]$$
Starting and Accepting

\[ \phi_{\text{start}} = \bigwedge_{1 \leq j \leq n^k} [x_{1,j,s_j}] \]

\[ \phi_{\text{accept}} = \bigvee_{1 \leq i,j \leq n^k} [x_{i,j,q_A}] \]

We can generate the start and accept formulas in \( n^k \) and \( (n^k)^2 \) time, both polynomial.

So now we can say that:

\( \phi_{\text{start}} \) is satisfied by, and only by, a tableau with the starting configuration of \( M \) on \( w \) encoded as its first row, and is created in polynomial time.

...and...

\( \phi_{\text{accept}} \) is satisfied by, and only by, a tableau encoding an accepting configuration as one of its rows, and is created in polynomial time.
Now, for transitions. Recall the discussions we had about ID changes being limited to three characters or six, when looking at transitions.

A given 2x3 window is **legal** if it does not violate our machine’s transition function.

Given the linear sets of states and tape symbols, and the finite size of 2x3 windows, we can make a **polynomial-sized set of all legal windows**.

Let a sequence $A = (a_1, \ldots, a_6)$ be a 2x3 window, with $a_1$ the top left cell, $a_2$ the top middle, etc.

We say that $A$ is **legal** if it represents a legal window. Here we have $q_0 \rightarrow R q_1$
Transitions

A given 2x3 window is legal if it does not violate our machine’s transition function. We have a polynomial-sized set of all legal windows.

Let a sequence \( A = (a_1, \ldots, a_6) \) be a 2x3 window. \( A \) is legal if it represents a legal window.

Now we can come up with a formula to say that the window top-centered at cell \((i, j)\) is legal.

\[
\phi_{\text{legal}}(i, j) = \bigvee_{A=(a_1,\ldots,a_6)\text{ is legal}} \left[ x_{i,j-1,a_1} \land x_{i,j,a_2} \land x_{i,j+1,a_3} \land x_{i+1,j-1,a_4} \land x_{i+1,j,a_5} \land x_{i+1,j+1,a_6} \right]
\]

Don’t be intimidated by this formula!

It’s just counting off the six cells of the window and demanding that each be equal to the corresponding cell in some legal window.
Transitions

A given 2x3 window is legal if it does not violate our machine’s transition function.

We have a polynomial-sized set of all legal windows.

Let a sequence \( A = (a_1, \ldots, a_6) \) be a 2x3 window. \( A \) is legal if it represents a legal window.

\[
\phi_{\text{legal}}(i, j) = \bigvee_{A = (a_1, \ldots, a_6) \text{ is legal}} \left[ x_{i,j-1,a_1} \land x_{i,j,a_2} \land x_{i,j+1,a_3} \land x_{i+1,j-1,a_4} \land x_{i+1,j,a_5} \land x_{i+1,j+1,a_6} \right]
\]

Since we have a polynomial number of legal windows, this formula is also polynomial. So we can say:

\( \phi_{\text{legal}}(i, j) \) is satisfied by, and only by, a tableau whose window top-centered at \((i, j)\) is legal; and is created in polynomial time.
Windows and Configurations

Consider any upper and lower configuration in the tableau, so that the lower configuration is the one immediately below – that is, following – the upper.

If all the windows top-centered on cells in the upper configuration are legal, then:

- The legality of the windows that don’t involve the state symbol easily ensures the legality of the configuration below them.
- The window top-centered on the state symbol in the upper configuration is sufficient to ensure that the state symbol in the lower configuration makes a legal move.

The upper configuration yields the lower one if and only if all the windows top-centered on cells in the upper configuration are legal – and that holds all the way down the tableau.
Windows and Configurations

\[
\phi_{\text{legal}}(i, j) = \bigvee_{A=(a_1, \ldots, a_6)} \left[ x_{i, j-1, a_1} \land x_{i, j, a_2} \land x_{i, j+1, a_3} \land x_{i+1, j-1, a_4} \land x_{i+1, j, a_5} \land x_{i+1, j+1, a_6} \right]
\]

\(\phi_{\text{legal}}(i, j)\) is satisfied by, and only by, a tableau whose window top-centered at \((i, j)\) is legal; and is created in polynomial time.

An upper configuration yields a lower one iff all the windows top-centered within the upper are legal.

This holds all the way down the tableau.

Then we have:

\[
\phi_{\text{move}} = \bigwedge_{1 \leq i < n^k, \ 1 < j < n^k} \phi_{\text{legal}}(i, j)
\]

And can say \(\phi_{\text{move}}\) is satisfied by, and only by, a tableau that does not violate the machine’s transition function; and is created in polynomial time.
Pulling It Together

We have:

\( \phi_{\text{cells}} \) is satisfied by, and only by, a properly encoded tableau.

\( \phi_{\text{start}} \) is satisfied by, and only by, a tableau with the starting configuration of \( M \) on \( w \) encoded as its first row.

\( \phi_{\text{accept}} \) is satisfied by, and only by, a tableau encoding an accepting configuration as one of its rows.

\( \phi_{\text{move}} \) is satisfied by, and only by, a tableau that does not violate the machine’s transition function.

All are created in polynomial time.

Then \( \phi_{\text{NDTM}} \) is satisfied by, and only by, a tableau encoding an accepting computation history of \( M \) on \( w \), and is created in polynomial time.
SAT is NP-Complete

\[ \phi_{\text{NDTM}} = (\phi_{\text{cells}} \wedge \phi_{\text{start}} \wedge \phi_{\text{accept}} \wedge \phi_{\text{move}}) \]

\( \phi_{\text{NDTM}} \) created from NDTM \( M \) and input \( w \) is satisfied by, and only by, a tableau encoding an accepting computation history of \( M \) on \( w \), and is created in polynomial time.

This means that:

SAT accepts \( \phi_{\text{NDTM}} \) if and only if such a tableau exists…

…if and only if the NDTM we are encoding into \( \phi_{\text{NDTM}} \) accepts \( w \).

We’ve just polynomially reduced every possible NP language to SAT.

Let’s convince ourselves of that a bit more.

By definition, any NP language has an NDTM \( M \) that decides it in polynomial time.

We can decide any NP language with a result from SAT using the following algorithm:

On input \( \langle M, w \rangle \):

Create \( \phi_{\text{NDTM}} \) from \( M \) and \( w \).
Run the decider for SAT on \( \phi_{\text{NDTM}} \).
Accept if SAT accepts, reject if it rejects.

SAT is NP-complete.
Cook’s Theorem

• $\varphi_{M,w} = \varphi_{\text{cells}} \land \varphi_{\text{start}} \land \varphi_{\text{accept}} \land \varphi_{\text{move}}$

• See the following for another detailed description and discussion of the four terms that make up this formula.

NP–Complete

Within a year, Richard Karp added 22 problems to this special class.

We will focus on:

3-SAT
Integer Linear Programming
SubsetSum
Partition
Vertex Cover
Independent Set
K-Color
Multiprocessor Scheduling
Co-NP

• A problem is in co-NP if its complement is in NP
  – This is like co-RE, with respect to RE problems.

• An example is the problem to determine if a Boolean expression is a tautology.
  – If the answer to the problem "is B in TAUT ?" is NO, then ¬A is in SAT.

• A more direct example of a co-NP problem is to determine if a Boolean expression is self-contradictory.
  – This is the complement of the notion of satisfiability but not of an instance of satisfiability

• Both of the above are co-NP Complete
SAT to 3SAT

- 3-SAT means that each clause has exactly three terms
- If one term, e.g., (p), expand to (p ∨ p ∨ p)
- If two terms, e.g., (p ∨ q), expand to (p ∨ q ∨ p)
- Any clause with three terms is fine
- If n > three terms, can reduce to two clauses, one with three terms and one with n-1 terms, e.g., (p₁ ∨ p₂ ∨ ... ∨ pₙ) to (p₁ ∨ p₂ ∨ z) & (p₃ ∨ ... ∨ pₙ ∨ ¬z), where z is a new variable. If n=4, we are done, else apply this approach again with the clause having n-1 terms
3SAT Growth is Linear

- Let’s make sure we have not grown the length of the expression going from an instance E of SAT to 3SAT by more than a polynomial amount.
- Let N be the number of clauses in E and let n be number of variables in E. Worst case length of E is then nN literals.
- If a clause in E has k≥3 literals, we want to define G(k)=number of literals in the 3SAT version, E’.
  - G(3)=3; G(4)=G(3)+G(3)=6; G(5)=G(4)+G(3)=9; G(6)=G(5)+G(3)=12; …; G(k)=3(k-2), k≥3.
- The worst case is bounded above by 3nN literals in E’ which is polynomial (linear) growth.
Linear Programming (LP)

• Linear Programming (LP) is like solving a set of linear equations but allows just equality (=) but also inequality (> , < , ≥ , ≤)

• In actual fact, LP usually also includes an optimization function, but we are limiting ourselves to decision problems

• Example:
  x + y > 7
  x - y ≥ 4
  Has many solutions, some of which are integral, e.g.,
  x=7, y=1
Integer LP (ILP)

- Integer Linear Programming (ILP) just constrains the solutions to an LP problem to be integral values.
- This constraint, on the surface, may seem to make the problem easier but, in fact, makes it harder.
- This is even true when we view this as a decision problem where we just ask “Is there a solution to this instance of ILP?”
- We will see this complexity play out in the next few slides.
0-1 ILP

- 0-1 ILP says constrains the solution space to variable values of 1 and 1
- Start with an instance of SAT (or 3SAT), assuming variables v1,…, vn and clauses c1,…, cm
- For each variable vi, have the constraint that 0 ≤ vi ≤ 1
- For each clause we provide a constraint that it must be satisfied (evaluate to at least 1). For example, if clause cj is v2 ∨ ~v3 ∨ v5 ∨ v6 then add the constraint v2 + (1-v3) + v5 + v6 ≥ 1
- A solution to this set of integer linear constraints implies a solution to the instance of SAT and vice versa
- Note this works for any SAT instance not just 3SAT
0-1 ILP is NP-Complete

- Previous page just show 0-1 ILP is NP-Hard
- Must show it is in NP
- Can do by trying all $2^k$ 0-1 assignments to $k$ variables
- Or can show that verifying a solution is in P – it’s really just linear
0-1 ILP Example

• Original SAT: \( E = (a+b+\neg c+d+e)(\neg b)(\neg a+\neg d)(b+c+\neg e) \)
• \( 0 \leq a \leq 1; \ 0 \leq b \leq 1; \ 0 \leq c \leq 1; \ 0 \leq d \leq 1; \ 0 \leq e \leq 1 \)
• \( a+b+(1-c)+d+e \geq 1; \) alternatively, \( a+b-c+d+e \geq 0 \)
• \( 1-b \geq 1; \) alternatively, \( b = 0 \)
• \( (1-a)+(1-d) \geq 1; \) alternatively, \( a+d \leq 1 \)
• \( b+c+(1-e) \geq 1; \) alternatively, \( b+c-e \geq 0 \)
What about ILP?

• As we said, ILP just constrains the solution to integers not to binary values
• Clearly ILP is NP-Hard as the constrained version of 0-1 ILP is NP-Hard
• Showing ILP is in NP is easy using a verifier; you give me a proposed solution and I can check it in linear time
What about Linear Programming (LP) #1?

- Linear programming just requires that solution be real number values
- The only constraints are in the simultaneous inequalities (and equalities)
- If you limit LP to equalities, then it has a well-known complexity of $O(N^3)$ using Gaussian Elimination or one of its variants
What about Linear Programming (LP) #2?

• The problem of solving LP appeared to be exponential for a long time and was, and still is, generally attacked using the Simplex Method which involves adding slack variables, e.g.,
\[ x + y < 7 \text{ iff } x + y + e = 7 \text{ for some } e > 0 \text{ and } x + y \leq 7 \text{ iff } x + y + f = 7 \text{ for some } f \geq 0 \]

• One can show cases where the Simplex Method takes exponential time, but its average case is \( O(N^{\sqrt{d}}) \) time where \( N \) is the number of variables and \( d \) is bounded above by the size of the input in bits
What about Linear Programming (LP) #3?

• In 1984, LP was shown to be of polynomial complexity.

• Complexity is $O(N^{3.5} L \lg L \ lglg L)$ where $N$ is the number of variables and $L$ is the size of the input in bits.

• Simplex is still used much as QuickSort is used for sorting even though its worst case is $O(N^2)$ as its expected performance is $O(N \ lg N)$ and it often converges in $O(N)$. 
SubsetSum

$S = \{s_1, s_2, \ldots, s_n\}$

set of positive integers and an integer $G$.

**Question:** Does $S$ have a subset whose values sum to the goal $G$?

**Note:** This is really a Bag (Multiset) not a Set. No one knows of a polynomial algorithm. {No one has proven there isn’t one, either!!}
SubsetSum is in NP

- You give me a “solution” to \([ \{s_1, s_2, \ldots, s_n\}, G]\)
- Your solution is just a subset of the set of integers \{1 \ldots n\}
- I make sure that each number you give me is unique and in the correct range (that takes me \(n\) units of time). If not, I reject your “solution”
- I then add the selected numbers together. That takes the sum of the log based 2 of the numbers you selected. I then check that the sum equals \(G\). If so, I verify; if not, I reject.
- Note that the original representation is of length the sum of the log based 2 of the \(s_i\)’s and \(G\) so my growth of time is linear
- Thus, I can verify in polynomial time
Example SubsetSum

- Instance
  $[(15, 17, 27, 11, 4, 12, 33, 5, 6, 21, 2), 57]$  
- A solution is 15, 17, 11, 12, 2 (or with indices just 1, 2, 4, 6, 11)
- Note that an item can only be chosen once
- Note that we can try a heuristic like sorting low to high
  $[(2, 4, 5, 6, 11, 12, 15, 17, 21, 27, 33), 57]$  
- But, an attack with that might have us choose
  2, 4, 5, 6, 11, 12, 15 and we are stuck
- In above one can backtrack to remove 15 and replace by 17 works, but backtracking is in general exponential
- Try high to low $[(33, 27, 21, 17, 15, 12, 11, 6, 5, 4, 2), 57]$
  33, 27 (Fail), 33, 21, 17 (Fail), 33, 21, 15 (Fail), etc.
- Clearly all these are potentially exponential approaches
SAT $\leq_p$ 3SAT $\leq_p$ SubsetSum $\equiv_p$ Partition

Theorem. SAT $\leq_p$ ILP $\leq_p$ LP
Theorem. SAT $\leq_p$ 3SAT
Theorem. 3SAT $\leq_p$ SubsetSum
Theorem. SubsetSum $\leq_p$ Partition
Theorem. Partition $\leq_p$ SubsetSum

Therefore, not only is SAT in NP–Complete, but so are ILP, LP, 3SAT, Partition, and SubsetSum.
3SAT $\leq_p$ SubsetSum

Assuming a 3SAT expression $(a + \neg b + c) (\neg a + b + \neg c)$

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>a+$\neg$b+c</th>
<th>$\neg$a+b+$\neg$c</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>$\neg$a</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>b</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>$\neg$b</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>c</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>$\neg$c</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>C1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>C1’</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>C2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>C2’</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>
SubsetSum Matrix

• One column per variable and one column per clause
• Two rows per variable (true/false so only one can be chosen per variable) and two rows per clause (optional pads to get to 3’s in clause columns, provided we are already at least a 1).
• Each row is a number and summing them never results in carry to next column, so each column is independent of other and only influenced by rows we select.
• Goal of 1 … 1 3… 3 forces one choice (true or false per variable) and satisfiability for every clause (must have a 1 in at least one variable row of each clause column)
How it works

• Satisfying \((a + \neg b + c) (\neg a + b + \neg c)\)
• Make a, b and c true (satisfies)
  Rows a, b and c get us 11121
  Padding with C1, C2 and C2’ gets 11133
• Make a, \(\neg b\) and c true (does not satisfy)
  Rows a, \(\neg b\) and c get us 11130
  No amount of padding can get the last column to be 3 (2 is max)
Partition

• Given a Multiset \( S = \{s_1, s_2, \ldots, s_n\} \), where each \( s_i \) is a positive integer, can we partition it into two sub-bags \( P_1, P_2 \) such that \( P_1 \cup P_2 = S \) and \( P_1 \cap P_2 = \emptyset \)?

• Note: If \( S \) contains multiple copies of some integer, each is considered distinct and thus does not unduly influence the intersection and union operators above.
SubsetSum $\equiv_p$ Partition Details

- Partition is polynomial equivalent to SubsetSum
  - Let $i_1, i_2, \ldots, i_n, G$ be an instance of SubsetSum. This instance has answer “yes” iff
    $i_1, i_2, \ldots, i_n, 2\times \text{Sum}(i_1, i_2, \ldots, i_n) - G, \text{Sum}(i_1, i_2, \ldots, i_n) + G$ has answer “yes” in Partition. Here we assume that $G \leq \text{Sum}(i_1, i_2, \ldots, i_n)$, for, if not, the answer is “no.”
  - Let $i_1, i_2, \ldots, i_n$ be an instance of Partition. This instance has answer “yes” iff
    $i_1, i_2, \ldots, i_n, \text{Sum}(i_1, i_2, \ldots, i_n)/2$ has answer “yes” in SubsetSum
SubsetSum $\equiv_p$ Partition

- $[(15, 17, 27, 11, 4, 12, 33, 5, 6, 21, 2), 57]$
- A solution is 15, 17, 11, 12, 2
- Sum of all is 153
- Mapping to Partition is
  - $(15, 17, 27, 11, 4, 12, 33, 5, 6, 21, 2, 306-57, 153+57)$
  - $(15, 17, 27, 11, 4, 12, 33, 5, 6, 21, 2, 249, 210)$
  - $(15+17+11+12+2+249) = 306$
  - $(27+4+33+5+6+21+210) = 306$
- Going other direction map above to
  - $[(15, 17, 27, 11, 4, 12, 33, 5, 6, 21, 2, 249, 210), 306]$
VERTEX COVERING (VC) DECISION PROBLEM IS NP-HARD
3SAT to Vertex Cover

- Vertex cover seeks a set of vertices that cover every edge in some graph.
- Let $I_{3\text{-SAT}}$ be an arbitrary instance of 3-SAT. For integers $n$ and $m$, $U = \{u_1, u_2, \ldots, u_n\}$ and $C_i = \{z_{i1}, z_{i2}, z_{i3}\}$ for $1 \leq i \leq m$, where each $z_{ij}$ is either $u_k$ or $u_k'$ for some $k$.
- Construct an instance of VC as follows.
  - For each $i$, $1 \leq i \leq n$, construct two vertices, $u_i$ and $u_i'$ with an edge between them.
  - For each clause $C_i = \{z_{i1}, z_{i2}, z_{i3}\}$, $1 \leq i \leq m$, construct three vertices $z_{i1}$, $z_{i2}$, and $z_{i3}$ and form a "triangle on them. Each $z_{ij}$ is one of the Boolean variables $u_k$ or its complement $u_k'$. Draw an edge between $z_{ij}$ and the Boolean variable (whichever it is). Each $z_{ij}$ has degree 3. Finally, set $k = n+2m$.
- Theorem. The given instance of 3-SAT is satisfiable if and only if the constructed instance of VC has a vertex cover with at most $k$ vertices.
VC Variable Gadget

To cover the edge in between $x$ and $\neg x$, at least one of these must be chosen
To cover the edges here, at least two of three vertices must be chosen.

\[ a + b + \neg c \]
Goal is to cover all variables (really edges) with $v + 2c$ nodes (minimum needed); $v=#variables; c=#clauses; for above that is $2 + 2*3 = 8$: Light blue are choices
Why VC Gadgets Work

• For each variable gadget, we must either choose the variable or its complement to cover the edge connecting them; Choosing both is wasteful.

• For each clause gadget, we must cover its internal edges. This requires 2 per clause, but also need to cover all edges entering from variable gadgets, if not already covered by the selection of the corresponding variable gadget.

• If we can cover all edges, with just \( v + 2c \) nodes then we have attained the minimum possible and guaranteed that each clause has at least one of its literals true. This allows the corresponding variable selection (true/false) to cover the incoming edge allowing the three internal edges to be covered by the other two variable nodes in the clause. If more than one literal is covered, you have a choice of which covered internal node to not choose for the clause gadget.
Solution chooses \(~x_1, x_2\); By choosing those variable gadget nodes we cover both variable gadgets. We know we must cover all clause gadgets and the edges entering them from the variable gadgets. For the first clause, we do not need to cover the edge entering from variable gadget \(x_2\), but we must cover the other two external edges and the three internal edges. Choosing the two \(x_1\) nodes in clause 1 covers all external and internal edges. Had we chosen \(~x_1\) in the \(x_1\) variable gadget and \(~x_2\) in the \(x_2\) variable gadget we would have had to chose \(x_2\) in the first clause gadget thereby exceeding our quota of \(v + 2c\). I leave it to you to see why the others work and to understand why we had no actual constraints in the third clause gadget (any two would have worked).
Independent Set

• Independent Set
  – Given Graph $G = (V, E)$, a subset $S$ of the vertices is independent if there are no edges between vertices in $S$
  – The $k$-IS problem is to determine for a $k > 0$ and a graph $G$, whether or not $G$ has an independent set of $k$ nodes

• Note there is a related NP-Hard optimization problem to find a Maximum Independent Set. It is even hard to approximate a solution to the Maximum Independent Set Problem.
IS (VC) Clause Gadget

$a + b + \sim c$
3SAT to IS

\[(a + \sim b + c) (\sim a + b + \sim c)(a + b + c), \ k=3\]

\( (k=\text{number of clauses, not variables}) \)
Explaining the example

In each clause gadget we can only chose one node, else we would be choosing two nodes with a shared edge. Assume we select a in clause 1, we cannot choose ~a in any other clause as they have a shared edge. After choosing a in clause 1, we could choose either b or ~c in clause 2. Assume we select ~c in clause 2, we cannot choose c in any other clause as they have a shared edge. To reach three (the number of clauses, we need a choice left for clause 3. Fortunately we have two choices, either a or c.
K-COLOR (KC) DECISION PROBLEM IS NP-HARD
K-Coloring

Given:
A graph \( G = (V, E) \) and an integer \( k \).

Question:
Can the vertices of \( G \) be assigned colors from a palette of size \( k \), so that adjacent vertices have different colors and use at most \( k \) colors?

3Coloring (3C) uses \( k=3 \)

3/18/21 © UCF CS 135
3C Super + Variables Gadget

T \rightarrow F

Base

\{v_1, \bar{v}_1, v_2, \bar{v}_2, \bar{v}_n, v_n\}
What if $a$, $b$?
What if ~a, ~b?
What if $a$, $\neg b$?
What if \( \sim a, b \)?
3C Clause Gadget
Consider \( \sim a, \sim b, \sim c \)
Consider $a \parallel b, \neg c$
Consider \( \sim a, \sim b, c \)
Consider one of $a \lor b, c$
Consider $a, b, c$
KC Gadgets Combined

\[(u + \neg v + w) (v + x + \neg y)\]
Register Allocation

- Liveness: A variable is live if its current assignment may be used at some future point in a program’s flow
- Optimizers often try to keep live variables in registers
- If two variables are simultaneously live, they need to be kept in separate registers
- Consider the K-coloring problem (can the nodes of a graph be colored with at most K colors under the constraint that adjacent nodes must have different colors?)
- Register Allocation reduces to K-coloring by mapping each variable to a node and inserting an edge between variables that are simultaneously live
- K-coloring reduces to Register Allocation by interpreting nodes as variables and edges as indicating concurrent liveness
- This is a simple mapping because it’s an isomorphism
Live Variable Analysis

Code

\[
a = 1; \\
b = 2; \\
c = a \times b; \\
d = a + 3; \\
e = a + b;
\]

No Optimization

\[
T1 = 1 \\
T2 = 2 \\
T3 = T1 \times T2 \\
T4 = T1 + 3 \\
T5 = T1 + T2
\]

Minimum colors are 3 so need 3 registers to spilling to memory and reloading.
PROCESSOR SCHEDULING IS NP-HARD
Processor Scheduling

- A Process Scheduling Problem can be described by
  - m processors $P_1, P_2, \ldots, P_m$,
  - processor timing functions $S_1, S_2, \ldots, S_m$, each describing how the corresponding processor responds to an execution profile,
  - additional resources $R_1, R_2, \ldots, R_k$, e.g., memory
  - transmission cost matrix $C_{ij} (1 \leq i, j \leq m)$, based on proc. data sharing,
  - tasks to be executed $T_1, T_2, \ldots, T_n$,
  - task execution profiles $A_1, A_2, \ldots, A_n$,
  - a partial order defined on the tasks such that $T_i < T_j$ means that $T_i$ must complete before $T_j$ can start execution,
  - communication matrix $D_{ij} (1 \leq i, j \leq n)$; $D_{ij}$ can be non-zero only if $T_i < T_j$,
  - weights $W_1, W_2, \ldots, W_n$ -- cost of deferring execution of task.
Complexity Overview

- The intent of a scheduling algorithm is to minimize the sum of the weighted completion times of all tasks, while obeying the constraints of the task system. Weights can be made large to impose deadlines.
- The general scheduling problem is quite complex, but even simpler instances, where the processors are uniform, there are no additional resources, there is no data transmission, the execution profile is just processor time and the weights are uniform, are very hard.
- In fact, if we just specify the time to complete each task and we have no partial ordering, then finding an optimal schedule on two processors is an NP-complete problem. It is essentially the optimization version of the Partition or equally can be viewed as a SubsetSum problem.
The problem of optimally scheduling $n$ tasks $T_1, T_2, \ldots, T_n$ onto 2 processors with an empty partial order $<$ is the same as that of dividing a set of positive whole numbers into two subsets, such that the numbers are as close to evenly divided. So, for example, given the numbers $3, 2, 4, 1$ we could try a “greedy” approach as follows:

- put 3 in set 1
- put 2 in set 2
- put 4 in set 2 (total is now 6)
- put 1 in set 1 (total is now 4)

This is not the best solution. A better option is to put 3 and 2 in one set and 4 and 1 in the other. Such a solution would have been attained if we did a greedy solution on a sorted version of the original numbers. In general, however, sorting doesn’t work.
2 Processor Nastiness

Try the unsorted list \((2-1/m)\)
7, 7, 6, 6, 5, 4, 4, 4, 4

Greedy (Always in one that is least used)
7, 6, 5, 5 = 23
7, 6, 4, 4, 4 = 25

Optimal
7, 6, 6, 5 = 24
7, 4, 4, 4, 5 = 24

Sort it (non-increasing) \((4/3-1/3m)\)
7, 7, 6, 6, 5, 5, 4, 4, 4
7, 6, 5, 4, 4 = 26
7, 6, 5, 4 = 22

Sort it (non-decreasing) \((2-1/m)\)
4, 4, 4, 5, 5, 6, 6, 7, 7
4, 4, 5, 6, 7 = 26
4, 5, 6, 7 = 22

Both sorts are even worse than greedy unsorted !! (not a general result)
Consider the simple scheduling problem where we have a set of independent tasks running on a fixed number of processors, and we wish to minimize finishing time.

How would a list (first fit, no preemption) strategy schedule tasks with the following IDs and execution times onto four processors? Answer using Gantt chart.

(T1,4) (T2,1) (T3,3) (T4,6) (T5,2) (T6,1) (T7,4) (T8,5) (T9,7) (T10,3) (T11,4) (2-1/m)

Now show what would happen if the times were sorted non-decreasing. (2-1/m)

Now show what would happen if the times were sorted non-increasing. (4/3-1/3m)
2 Processor with partial order

List Schedule with $L = \{T_1, T_2, T_3, T_4, T_5, T_6\}$

Non-Preemptive, Delays Allowed

Preemptive
Anomalies everywhere

Diagram showing a list schedule with tasks T1, T2, T3, T4, T5, T6, T7, T8, T9 and their execution times.
More anomalies

Original List Schedule but with All Times Reduced

Original List Schedule but with T5 and T6 Independent
A UET is a Unit Execution Tree. Our Tree is funny. It has a single leaf by standard graph definitions.

1. Assign $L(T) = 1$, for the leaf task $T$
2. Let labels 1, ..., $k-1$ be assigned. If $T$ is a task with lowest numbered immediate successor then define $L(T) = k$ (non-deterministic)

This is an order n labeling algorithm that can easily be implemented using a breadth first search.

Note: This can be used for a forest as well as a tree. Just add a new leaf. Connect all the old leafs to be immediate parents of the new one. Use the above to get priorities, starting at 0, rather than 1. Then delete the new node completely.

Note: This whole thing can also be used for anti-trees. Make a schedule, read it backwards. You cannot just reverse priorities.
Theorem: Level Strategy is optimal for unit execution, m arbitrary, forest precedence.
Level – DAG with unit time

1. Assign $L(T) = 1$, for an arbitrary leaf task $T$
2. Let labels 1, ..., $k-1$ be assigned. For each task $T$ such that
   
   \{ $L(T')$ is defined for all $T'$ in $\text{Successor}(T)$ \}

   Let $N(T)$ be decreasing sequence of set members in
   \{ $S(T') \mid T'$ is in $S(T)$ \}

   Choose $T^*$ with least $N(T^*)$.
   Define $L(T^*) = K$.
   This is an order $n^2$ labeling algorithm. Scheduling with it involves $n$ union/find style operations. Such operations have been shown to be implementable in nearly constant time using an “amortization” algorithm.

**Theorem**: Level Strategy is optimal for unit execution, $m=2$, dag precedence.
Looking back at the UET example, consider adding two additional tasks numbered 15 and 16 that are siblings of 13 and 14. These four tasks must be completed before 12 is started.

a) Show the Gantt chart that reflects the new schedule associated with this enhanced tree

b) Show the Gantt chart that is associated with the corresponding anti-tree, in which all arcs are turned in the opposite direction. Use the technique of reversing the schedule from (a)

c) Show the Gantt chart associated with the anti-tree of b), where we now use the priorities obtained by treating lower numbered tasks as higher priority ones

d) Comment on the results seen in (b) versus (c), providing insight as to why they are different and why one is better than the other.
UNIVERSE OF SETS

NP-Complete

NP

P

Co-NP
HAMiLTONiAN CiRCuiT (HC)
DECiSiON PROBLEM IS NP-HARD
Hamiltonian Path/Circuit

- A **Hamiltonian Path** is a path through a graph from one node ‘start’ to another node ‘end’ that visits every node in the graph just once.
- A **Hamiltonian Circuit** is a Hamiltonian Path whose end node is adjacent to its start node. It can also be viewed that the start and end nodes are the same and that the only repeated node is the start node with its only repetition being at the end of the path.
HC Variable Gadget

This has many Hamiltonian Circuits
HC Gadgets Combined

We will set convention on $x_i$ true to be left to right and $x_i$ false to be right to left (can fix for opposite)

This has a Hamiltonian Circuit iff all clauses are satisfied with consistent assignments to each variable. Note left to right assigns $X_i$ as true; right to left assigns $\neg X_i$ as true. There are filler nodes on left and right and between clauses.
Hamiltonian Path

• Note we can split an arbitrary node, v, into two (v’,v’’’) – one, v’, has in-edges of v, other, v’’, has out-edges. Path (not cycle) must start at v’’ and end at v’ and goal is still K (the number of vertices).
Travelling Salesman

- **Travelling Salesman Problem:**
  Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city and returns to the origin city?

- This is a Hamiltonian Cycle with weights on edges and we seek *minimum* weight for cycle.

- The decision problem version involves setting a goal weight, $L$, and asking is we can achieve it.
Travelling Salesman and HC

• Start with HC = (V,E), K=|V|
• Set edges from HC instance to 1
• Add edges between pairs that lack such edges and make those weights 2 (often people make these K+1); this means that the reverse of unidirectional links also get weight 2
• Goal weight is K for cycle
Knapsack 0-1 Problem

- The goal is to maximize the value of a knapsack that can hold at most $W$ units (i.e. lbs or kg) worth of goods from a list of items $I_0, I_1, \ldots, I_{n-1}$.
  - Each item has 2 attributes:
    1) Value – let this be $v_i$ for item $I_i$
    2) Weight – let this be $w_i$ for item $I_i$

Thanks to Arup Guha
Knapsack 0-1 Problem

- The difference between this problem and the fractional knapsack one is that you CANNOT take a fraction of an item.
  - You can either take it or leave it.
  - Hence the name Knapsack 0-1 problem.
Knapsack Optimize vs Decide

• The optimization problem is to have the sum of the chosen values, $v_i$, to be as large as possible with the constraint that the sum of the corresponding weights, $w_i$, cannot exceed $W$.

• We can restate as decision problem to determine if there exists a set of items, each with equal weights and values < $W$, that reaches some fixed goal value, $W$. 
Knapsack and SubsetSum

• Let \( v_i = w_i \) for each item \( I_i \).
• By doing so, the value is maximized when the Knapsack is filled as close to capacity.
• The related decision problem is to determine if we can attain capacity (W).
• Clearly then, given an instance of the SubsetSum problem, we can create an instance of the Knapsack decision problem problem, such that we reach the goal sum, G, iff we can attain a Knapsack value of G.
Knapsack Decision Problem

• The reduction from SubsetSum shows that the Knapsack decision problem is at least as hard as SubsetSum, so it is NP-Complete if it is in NP.

• Think about whether or not it is in NP.

• Now, think about the optimization problem.
Related Bin Packing

- Have a bin capacity of B.
- Have item set $S = \{s_1, s_2, \ldots, s_n\}$
- Use all items in $S$, minimizing the number of bins, while adhering to the constraint that any such subset must sum to $B$ or less.
- This is similar to the processor scheduling problem without constraints, except we optimize on number of processors, not finishing time for all tasks. It is NP-Hard (WHY?)
Knapsack 0-1 Problem

• Brute Force
  – The naïve way to solve the 0-1 Knapsack problem is to cycle through all $2^n$ subsets of the n items and pick the subset with a legal weight that maximizes the value of the knapsack.
  – We can come up with a dynamic programming algorithm that is USUALLY faster than this brute force technique.
Knapsack 0-1 Problem

- We are going to solve the problem in terms of sub-problems and memoization (dynamic programming).
- Our first attempt might be to characterize a sub-problem as follows:
  - Let $S_k$ be the optimal subset of elements from \{I_0, I_1, ..., I_k\}.
  - What we find is that the optimal subset from the elements \{I_0, I_1, ..., I_{k+1}\} may not correspond to the optimal subset of elements from \{I_0, I_1, ..., I_k\} in any regular pattern.
  - Basically, the solution to the optimization problem for $S_{k+1}$ might NOT contain the optimal solution from problem $S_k$.  

3/18/21 © UCF CS
Knapsack 0-1 Problem

- Let’s illustrate that point with an example:

<table>
<thead>
<tr>
<th>Item</th>
<th>Weight</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>I₀</td>
<td>3</td>
<td>10</td>
</tr>
<tr>
<td>I₁</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>I₂</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>I₃</td>
<td>8</td>
<td>11</td>
</tr>
</tbody>
</table>

- The maximum weight the knapsack can hold is 20.

- The best set of items from \{I₀, I₁, I₂\} is \{I₀, I₁, I₂\}
- BUT the best set of items from \{I₀, I₁, I₂, I₃\}  is \{I₀, I₂, I₃\}.
  - In this example, note that this optimal solution, \{I₀, I₂, I₃\}, does NOT build upon the previous optimal solution, \{I₀, I₁, I₂\}.
    - (Instead it builds upon the solution, \{I₀, I₂\}, which is really the optimal subset of \{I₀, I₁, I₂\} with weight 12 or less.)
Knapsack 0-1 problem

• So now we must re-work the way we build upon previous sub-problems…
  ◦ Let $B[k, w]$ represent the maximum total value of a subset $S_k$ with weight $w$.
  ◦ Our goal is to find $B[n, W]$, where $n$ is the total number of items and $W$ is the maximal weight the knapsack can carry.

• So our recursive formula for subproblems:

$$B[k, w] = B[k - 1, w], \text{ if } w_k > w$$

$$= \max \{ B[k - 1, w], B[k - 1, w - w_k] + v_k \}, \text{ otherwise}$$

• In English, this means that the best subset of $S_k$ that has total weight $w$ is:
  1) The best subset of $S_{k-1}$ that has total weight $w$, or
  2) The best subset of $S_{k-1}$ that has total weight $w-w_k$ plus the item $k$
Knapsack 0-1 Problem – Recursive Formula

\[ B[k, w] = \begin{cases} 
B[k - 1, w], & \text{if } w_k > w \\
\max \{ B[k - 1, w], B[k - 1, w - w_k] + v_k \}, & \text{otherwise}
\end{cases} \]

- The best subset of \( S_k \) that has the total weight \( w \), either contains item \( k \) or not.

- **First case:** \( w_k > w \)
  - Item \( k \) can’t be part of the solution! If it was the total weight would be \( > w \), which is unacceptable.

- **Second case:** \( w_k \leq w \)
  - Then the item \( k \) can be in the solution, and we choose the case with greater value.
Knapsack 0-1 Algorithm

for w = 0 to W  // Initialize 1st row to 0’s
    B[0,w] = 0
for i = 1 to n  // Initialize 1st column to 0’s
    B[i,0] = 0
for i = 1 to n
    for w = 1 to W
        if w_i <= w  //item i can be in the solution
            if v_i + B[i-1,w-w_i] > B[i-1,w]
                B[i,w] = v_i + B[i-1,w- w_i]
            else
                B[i,w] = B[i-1,w]
        else B[i,w] = B[i-1,w]  // w_i > w
    }
Knapsack 0-1 Problem

Let's run our algorithm on the following data:

- $n = 4$ (# of elements)
- $W = 5$ (max weight)
- Elements (weight, value):
  - $(2,3)$, $(3,4)$, $(4,5)$, $(5,6)$
// Initialize the base cases
for w = 0 to W
  B[0,w] = 0

for i = 1 to n
  B[i,0] = 0
Knapsack 0-1 Example

<table>
<thead>
<tr>
<th>i / w</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

i = 1
v_i = 3
w_i = 2
w = 1
w-w_i = -1

if w_i <= w  //item i can be in the solution
    if v_i + B[i-1,w-w_i] > B[i-1,w]
        B[i,w] = v_i + B[i-1,w-w_i]
    else
        B[i,w] = B[i-1,w]

else B[i,w] = B[i-1,w]  // w_i > w
Knapsack 0-1 Example

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

\[i = 1\]
\[v_i = 3\]
\[w_i = 2\]
\[w = 2\]
\[w - w_i = 0\]

if \(w_i \leq w\) //item i can be in the solution

if \(v_i + B[i-1, w-w_i] > B[i-1, w]\)

\[B[i, w] = v_i + B[i-1, w-w_i]\]

else

\[B[i, w] = B[i-1, w]\]

else \(B[i, w] = B[i-1, w] // w_i > w\)
Knapsack 0-1 Example

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

i = 1

$v_i = 3$

$w_i = 2$

$w = 3$

$w - w_i = 1$

if $w_i \leq w$  // item i can be in the solution

if $v_i + B[i-1, w-w_i] > B[i-1, w]$

$B[i, w] = v_i + B[i-1, w-w_i]$

else

$B[i, w] = B[i-1, w]$

else $B[i, w] = B[i-1, w]$  // $w_i > w$
## Knapsack 0-1 Example

<table>
<thead>
<tr>
<th>i</th>
<th>w</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- i = 1
- $v_i = 3$
- $w_i = 2$
- $w = 4$
- $w - w_i = 2$

if $w_i \leq w$  //item i can be in the solution

if $v_i + B[i-1, w-w_i] > B[i-1, w]$  

$$B[i,w] = v_i + B[i-1, w-w_i]$$

else

$$B[i,w] = B[i-1,w]$$

else $B[i,w] = B[i-1,w]$  // $w_i > w$
# Knapsack 0-1 Example

<table>
<thead>
<tr>
<th>$i$</th>
<th>$w$</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

- $i = 1$
- $v_i = 3$
- $w_i = 2$
- $w = 5$
- $w - w_i = 3$

If $w_i \leq w$ //item i can be in the solution

- if $v_i + B[i-1, w-w_i] > B[i-1, w]$
  
  \[ B[i, w] = v_i + B[i-1, w-w_i] \]

- else
  
  \[ B[i, w] = B[i-1, w] \]

else $B[i, w] = B[i-1, w]$ // $w_i > w$
Knapsack 0-1 Example

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

<table>
<thead>
<tr>
<th>i / w</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

if \( w_i \leq w \)  //item i can be in the solution
  if \( v_i + B[i-1,w-w_i] > B[i-1,w] \)
    \( B[i,w] = v_i + B[i-1,w- w_i] \)
  else
    \( B[i,w] = B[i-1,w] \)
else \( B[i,w] = B[i-1,w] \)  // \( w_i > w \)
Knapsack 0-1 Example

<table>
<thead>
<tr>
<th>i / w</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

i = 2
v_i = 4
w_i = 3
w = 2
w - w_i = -1

if w_i <= w //item i can be in the solution
  if v_i + B[i-1,w-w_i] > B[i-1,w]
    B[i,w] = v_i + B[i-1,w-w_i]
  else
    B[i,w] = B[i-1,w]
else B[i,w] = B[i-1,w] // w_i > w
Knapsack 0-1 Example

Items:
1: (2,3)  
2: (3,4)  
3: (4,5)  
4: (5,6)

<table>
<thead>
<tr>
<th>i / w</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

if \( w_i \leq w \)  //item i can be in the solution

if \( v_i + B[i-1, w-w_i] > B[i-1, w] \)

\[ B[i,w] = v_i + B[i-1, w-w_i] \]

else

\[ B[i,w] = B[i-1, w] \]

else \( B[i,w] = B[i-1, w] \)  // \( w_i > w \)
## Knapsack 0-1 Example

<table>
<thead>
<tr>
<th>i / w</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Items:
- 1: (2,3)
- 2: (3,4)
- 3: (4,5)
- 4: (5,6)

- i = 2
- $v_i = 4$
- $w_i = 3$
- $w = 4$
- $w - w_i = 1$

if $w_i \leqslant w$  // item i can be in the solution
  - if $v_i + B[i-1, w-w_i] > B[i-1, w]$  
    \[ B[i, w] = v_i + B[i-1, w-w_i] \]
  - else  
    \[ B[i, w] = B[i-1, w] \]
else  
  \[ B[i, w] = B[i-1, w] \]  // $w_i > w$
Knapsack 0-1 Example

<table>
<thead>
<tr>
<th>i / w</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

\[
\begin{align*}
\text{if } w_i & \leq w \quad \text{//item i can be in the solution} \\
\text{if } v_i + B[i-1, w-w_i] & > B[i-1, w] \\
B[i, w] & = v_i + B[i-1, w-w_i] \\
\text{else} \\
B[i, w] & = B[i-1, w] \\
\text{else } B[i, w] & = B[i-1, w] \quad // w_i > w
\end{align*}
\]

i = 2
v_i = 4
w_i = 3
w = 5
w-w_i = 2
Knapsack 0-1 Example

<table>
<thead>
<tr>
<th>i / w</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Items:</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: (2,3)</td>
</tr>
<tr>
<td>2: (3,4)</td>
</tr>
<tr>
<td>3: (4,5)</td>
</tr>
<tr>
<td>4: (5,6)</td>
</tr>
</tbody>
</table>

if \( w_i \leq w \)  // item i can be in the solution

if \( v_i + B[i-1,w-w_i] > B[i-1,w] \)

\[ B[i,w] = v_i + B[i-1,w-w_i] \]

else

\[ B[i,w] = B[i-1,w] \]

else \( B[i,w] = B[i-1,w] \)  // \( w_i > w \)
**Knapsack 0-1 Example**

<table>
<thead>
<tr>
<th>i / w</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

if $w_i \leq w$  //item i can be in the solution
if $v_i + B[i-1, w-w_i] > B[i-1, w]$

\[ B[i, w] = v_i + B[i-1, w-w_i] \]

else

\[ B[i, w] = B[i-1, w] \]

else $B[i, w] = B[i-1, w]$  // $w_i > w$
**Knapsack 0-1 Example**

<table>
<thead>
<tr>
<th>i / w</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td><strong>→7</strong></td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Item 1: (2,3)
- Item 2: (3,4)
- Item 3: (4,5)
- Item 4: (5,6)

\[
i = 3 \\
v_i = 5 \\
w_i = 4 \\
w = 5 \\
w-w_i = 1
\]

if \( w_i \leq w \)  //item i can be in the solution

\[
\text{if } v_i + B[i-1,w-w_i] > B[i-1,w] \\
B[i,w] = v_i + B[i-1,w-w_i]
\]

else

\[
B[i,w] = B[i-1,w]
\]

else \( B[i,w] = B[i-1,w] \)  // \( w_i > w \)
Knapsack 0-1 Example

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

<table>
<thead>
<tr>
<th>i / w</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>7</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>

\(i = 4\)
\(v_i = 6\)
\(w_i = 5\)
\(w = 1..4\)
\(w-w_i = -4..-1\)

if \(w_i \leq w\)  //item i can be in the solution
  if \(v_i + B[i-1,w-w_i] > B[i-1,w]\)
    \(B[i,w] = v_i + B[i-1,w-w_i]\)
  else
    \(B[i,w] = B[i-1,w]\)
else \(B[i,w] = B[i-1,w]\)  // \(w_i > w\)
### Knapsack 0-1 Example

#### Items:
- 1: (2, 3)
- 2: (3, 4)
- 3: (4, 5)
- 4: (5, 6)

<table>
<thead>
<tr>
<th>i / w</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>7</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>7</td>
</tr>
</tbody>
</table>

- \(i = 4\)
- \(v_i = 6\)
- \(w_i = 5\)
- \(w = 5\)
- \(w - w_i = 0\)

#### Algorithm:

if \(w_i \leq w\)  // item \(i\) can be in the solution

if \(v_i + B[i-1, w-w_i] > B[i-1, w]\)

\[B[i, w] = v_i + B[i-1, w - w_i]\]

else

\[B[i, w] = B[i-1, w]\]

else \(B[i, w] = B[i-1, w]\)  // \(w_i > w\)
### Knapsack 0-1 Example

**Items:**
- 1: (2,3)
- 2: (3,4)
- 3: (4,5)
- 4: (5,6)

<table>
<thead>
<tr>
<th>i / w</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>7</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>7</td>
</tr>
</tbody>
</table>

We’re DONE!!

The max possible value that can be carried in this knapsack is $7$
Knapsack 0-1 Problem – Run Time

for \( w = 0 \) to \( W \)
\[
B[0,w] = 0
\]
\( O(W) \)

for \( i = 1 \) to \( n \)
\[
B[i,0] = 0
\]
\( O(n) \)

for \( i = 1 \) to \( n \)
Repeat \( n \) times
for \( w = 0 \) to \( W \)
\(< \text{the rest of the code} >\) \( O(W) \)

What is the running time of this algorithm?
\( O(n*W) – \text{of course, } W \text{ can be mighty big} \)

What is an analogy in world of sorting?

Remember that the brute-force algorithm takes: \( O(2^n) \)
Tiling Undecidable and NP-Complete Variants
Basic Idea of Tiling

A single tile has colors on all four sides. Tiles are often called dominoes as assembling them follows the rules of placing dominoes. That is, the color (or number) of a side must match that of its adjacent tile, e.g., tile, t2, to right of a tile, t1, must have same color on its left as is on the right side of t1. This constraint applies to top and bottom as well as sides. Boundary tiles do not have constraints on their sides that touch the boundaries.
Instance of Tiling Problem

- A finite set of tile types (a type is determined by the colors of its edges)
- Some 2d area (finite or infinite) on which the tiles are to be laid out
- An optional starting set of tiles in fixed positions
- The goal is to tile the plane following the adjacency constraints and whatever constraints are indicated by the starting configuration.
A Valid 3 by 3 Tiling of Tile Types from Previous Slide
Some Variations

• Infinite 2d plane (impossible, co-re-non-rec) in general
  – Our two tile types can easily tile the 2d plane

• Finite 2d plane (hard in general)
  – Our two tile types can easily tile any finite 2d plane
  – This is called the Bounded Tiling Problem

• One dimensional space
  – This is actually related to circuits in a directed graph
    (Why so? If tiles are vertices, what are edges?)
Tiling the Plane

- We will start with a Post Machine, \( M = (Q, \Sigma, \delta, q_0) \), with tape alphabet \( \Sigma = \{B, 1\} \) where \( B \) is blank and \( \delta \) maps pairs from \( Q \times \Sigma \) to \( Q \times (\Sigma \cup \{R,L\}) \). \( M \) starts in state \( q_0 \)
  - (Turing Machine with each action being L, R or Print)
- We will consider the case of \( M \) starting with a blank tape
- We will constrain our machine to never go to the left of its starting position (semi unbounded tape)
- We will mimic the computation steps of \( M \)
- Termination occurs if in state \( q \) reading \( b \) and \( \delta(q,b) \) is not defined
- We will use the fact that halting when starting at the left end of a semi unbounded tape in its initial state with a blank tape is undecidable; we will actually look at complement of this
The Tiling Decision Problem

• Given a finite set of tile types and a starting tile in lower left corner of 2d plane, can we tile all places in the plane?
• A place is defined by its coordinates \((x,y)\), \(x \geq 0, y \geq 0\)
• The fixed starting tile is at \((0,0)\)
Colors

- Given M, define our tile colors as
- \{X, Y, *, B, 1, YB, Y1\} \cup Q \times \{B,1\} \cup Q \times \{YB,Y1\} \cup Q \times \{R,L\}
- X appears only on bottom of any and all tiles that are resting on the X-axis
- Y appears only on left of any and all tiles that are adjacent to the Y-axis
- Y is part of the label on top of any tile with its left side adjacent to the Y-axis
Simple Tiles

- Simplest tile (represents Blank on X axis)

- Start tile (state q0; scanned symbol blank)
Tiles for Copying Tape Cell

Copy cells not on left boundary and not scanned

Copy cells on left boundary and not scanned
Right Move $\delta(q,a) = (p,R)$

where $b \in \Sigma = \{B, 1\}$
Left Move $\delta(q,a) = (p,L)$

where $b \in \Sigma = \{B, 1\}$
Print $\delta(q,a) = (p,c)$
Corner Tile and Bottom Row

Zero-ed Row is forced to be

\[
\begin{array}{c}
q_0, YB \\
Y \\
B \\
X \\
\end{array}
\]

\[
\begin{array}{c}
q_0, YB \\
Y \\
B \\
X \\
\end{array}
\]

\[
\begin{array}{c}
B \\
B \\
B \\
X \\
\end{array}
\]

\[
\begin{array}{c}
B \\
B \\
X \\
B \\
\end{array}
\]

\[
\begin{array}{c}
B \\
B \\
X \\
B \\
\end{array}
\]
As we cannot move left of leftmost character first action is either right or print. Assume for now that $\delta(q_0, B) = (p, a)$.
As we cannot move left of leftmost character first action is either right or print. Assume for now that $\delta(q_0, B) = (p, R)$.
The Rest of the Story Part 1

- Inductively we can show that, if the i-th row represents an infinite transcription of the Turing configuration after step i then the (i+1)-st represents such a transcription after step i+1. Since we have shown the base case, we have a successful simulation.
The Rest of the Story Part 2

• Consider the case where M eventually halts when started on a blank tape in state $q_0$. In this case we will reach a point where no actions fill the slots above the one representing the current state. That means that we cannot tile the plane.

• If M never halts, then we can tile the plane (in the limit).
The Rest of the Story Part 3

- The consequences of Parts 1 and 2 are that Tiling the plane is as hard as the complement of the Halting problem ($\forall t [\neg \text{STP}(M, x, t)]$) or of accepts 0 ($\forall t [\neg \text{STP}(M, 0, t)]$) which are both co-RE Complete.

- This is not surprising as this problem involve a universal quantification over all coordinates $(x,y)$ in the plane.
Constraints on M

- The starting blank tape is not a real constraint as we can create $M$ so its first actions are to write arguments on its tape.
- The semi unbounded tape is not new. If you look back at Standard Turing Computing (STC), we assumed there that we never moved left of the blank preceding our first argument.
- If you prefer to consider all computation based on the STC model then we can add to $M$ the simple prologue $(R1)^{x_1}R(R1)^{x_2}R\cdots(R1)^{x_k}R$ so the actual computation starts with a vector of $x_1 \ldots x_k$ on the tape and with the scanned square as the blank to right of this vector. The rest of the tape is blank.
- Think about how, in the preceding pages, you could actually start the tiling in this configuration.
Bounded Tiling Problem #1

• Consider a slight change to our machine M. First, it is non-deterministic, so our transition function maps to sets.
• Second, we add two auxiliary states \{q_a, q_r\}, where \(q_a\) is our only accept state and \(q_r\) is our only reject state.
• We make it so the reject state has no successor states, but the accept state always transitions back to itself rewriting the scanned square unchanged.
• We also assume our machine accepts or rejects in at most \(n^k\) steps, where n is the length of its starting input which is written immediately to the right of the initial scanned square.
We limit our rows and column to be of size $n^k + 1$. We change our initial condition of the tape to start with the input to $M$. Thus, it looks like

![Diagram of tile patterns](image)

Note that there are $n^k - n$ of these blank representations at the end. We really only need the first as the tiling constraint will force all the others to be of the same form.
The finitely bounded Tiling Problem we just described mimics the operation of any given polynomially-bound non-deterministic Turing machine.

This machine can tile the finite plane of size \((n^k+1) \times (n^k+1)\) just in case the initial string is accepted in \(n^k\) or fewer steps on some path (really a trace of at most \(n^k\)).

If the string is not accepted, then we will hit a reject state on all paths and never complete tiling.

This shows that the bounded tiling problem is NP-Hard

Is it in NP? Yes. How? Well, we can be shown a tiling (posed solution takes space polynomial in \(n\)) and check it for completeness and consistency (this takes linear time in terms of proposed solution). Thus, we can verify the solution in time polynomial in \(n\).
A Final Comment on Tiling

• If you look back at the unbounded version, you can see that we could have simulated a non-deterministic Turing machine there, but it would have had the problem that the plane would be tiled if any of the non-deterministic choices diverged and that is not what we desired.

• However, we need to use a non-deterministic machine for the finite case as we made this so it tiled iff some path led to acceptance. If all lead to rejection, we get stalled out on all paths as the reject state can go nowhere.
Tiling Example

- Turing Machine Recognizes strings of at least two 1’s in succession.
- $q_0 \ 0 \ 0 \ q_2$
- $q_0 \ 1 \ \text{R} \ q_1$
- $q_1 \ 0 \ \text{L} \ q_2$
- $q_1 \ 1 \ 1 \ q_3$
- $q_2 \ 0 \ 0 \ q_2$
- $q_2 \ 1 \ 1 \ q_2$
- No $q_3$ rules so entering here stops tiling
Tile Replication

0 * * 0

1 * * 1

Y0 * Y0

Y1 * Y1
q0 0 0 q2

- q2,0
  *     *
- q0,0

- Y
- q0,Y0

q0 1 R q1

- 1
  *    q1,R
- q0,1

- Y
- q0,Y1

- Y1
  q1,R
  q0,Y1

- q1,1
  q1,R
  1

230
Sample Starting Rows

```
q_0, Y_1
Y     0
X

0      0
0      X

...............           0
0      0
0      X

q_0, Y_1
Y     0
X

1      0
0      0
0      X

0      0
0      0
0      X
```

………...
Case 1; Two More Rows

Diagram:

- q2,Y1
- Y
- q2,L
- Y1

- q1,0
- 0
- q2,L
- *

- q1,0
- 0
- q1,R
- *

- q1,R
- 0
- q0,Y1
- *

- q0,Y1
- Y
- 0
- X

- 0
- 0
- 0
- X

- 0
- 0
- 0
- X

...
Case 1; Row 3 repeated

q2,Y1
Y
q2,Y1

0
*
*
0

0
*
*
0

0
*
*
0

...
Case 2; Only Two More Rows

Y1
q1,R
0

q1,1
q1,R
1

q0,Y1

Y
q0,Y1

X

Y1

Y1

q3,1

* 

* 

* 

* 

* 

* 

* 

*
More on Variations

• One dimensional space (I asked you to think about that on an earlier slide)

• Infinite 3d space (really impossible in general)
  – This become there an exists, for all, problem – Does there exist an initial tape for which M never halts
  – In fact, one can mimic acceptance on no inputs here, meaning M is not an algorithm iff we can tile the 3d space
PCP Revisited

Bounded Post Correspondence
Bounded Variation

• Limit correspondence to a length that is polynomial in \( n \), where \( n \) is length of initial input string.

• Outline of proof we can get for almost free
  – Convert halting problem for a Non-deterministic Turing machine to word problem for a Semi-Thue System
    Note: we originally did for deterministic machines, but the construction works for non-determinism and maps nicely to Semi-Thue systems which are non-deterministic by definition.
  – Recast as an instance of PCP
  – Limit the length of word to \((n+2)^k\), where original TM accepts or rejects in \( n^k \) steps.
Another Approach

• There is a tighter bound on Bounded PCP.

• Given sequences \((x_1, x_2, \ldots, x_n)\) and \((y_1, y_2, \ldots, y_n)\), and a positive integer \(K \leq p(\max(|x_1|+\ldots+|x_n|, |y_1|+\ldots+|y_n|))\), where \(p\) is some polynomial, is there a solution to this instance involving indices \(i_1, \ldots, i_k\), \(k \leq K\) (not necessarily distinct), of integers between 1 and \(n\), such that the corresponding \(x\) and \(y\) strings are identical.

Co-NP

Fourth Significant Class of Problems
For any decision problem A in NP, there is a ‘complement’ problem Co–A defined on the same instances as A, but with a question whose answer is the negation of the answer in A. That is, an instance is a "yes" instance for A if and only if it is a "no" instance in Co–A.

Notice that the complement of the complement of a problem is the original problem.
Co–NP

Co–NP is the set of all decision problems whose complements are members of NP.

For example: consider Graph Color GC

Given: A graph G and an integer k.
Question: Can G be properly colored with k colors?
Co–NP

The complement problem of GC

Co–GC

Given: A graph G and an integer k.
Question: Do all proper colorings of G require more than k colors?
Co–NP

Notice that Co–GC is a problem that does not appear to be in the set NP. That is, we know of no way to check in polynomial time the answer to a "Yes" instance of Co–GC.

What is the "answer" to a Yes instance that can be verified in polynomial time?
Not all problems in NP behave this way. For example, if X is a problem in class P, then both "yes" and "no" instances can be solved in polynomial time.

That is, both "yes" and "no" instances can be verified in polynomial time and hence, X and Co–X are both in NP, in fact, both are in P. This implies $P = \text{Co–P}$ and, further, $P = \text{Co–P} \subseteq NP \cap \text{Co–NP}$. 
Co–NP

This gives rise to a second fundamental question:

\[ NP = \text{Co–NP?} \]

If \( P = \text{NP} \), then \( \text{NP} = \text{Co–NP} \).

This is not "if and only if."

It is possible that \( \text{NP} = \text{Co–NP} \) and, yet, \( P \neq \text{NP} \).
If $A \leq_p B$ and both are in NP, then the same polynomial transformation will reduce Co-A to Co-B. That is, Co-A $\leq_p$ Co-B. Therefore, Co-SAT is 'complete' in Co-NP.

In fact, corresponding to NP–Complete is the complement set Co–NP–Complete, the set of hardest problems in Co–NP.
Now, return to Turing Reductions.

Recall that Turing reductions include polynomial transformations as a special case. So, we should expect they will be more powerful.
Turing Reductions

(1) Problems A and B can, but need not, be decision problems.

(2) No restriction placed upon the number of instances of B that are constructed.

(3) Nor, how the result, $\text{Answer}_A$, is computed.

In effect, we use an Oracle for B.
Turing Reductions

Technically, Turing Reductions include Polynomial Transformations, but it is useful to distinguish them.

Polynomial transformations are often the easiest to apply.
NP–Hard

Fifth Significant Class of Problems
NP–Hard

To date, we have concerned ourselves with decision problems. We are now in a position to include additional problems. In particular, optimization problems.

We require one additional tool – the second type of transformation discussed above – Turing reductions.
NP–Hard

Definition: Problem B is NP–Hard if there is a polynomial time Turing reduction $A \leq_{PT} B$ for some problem $A$ in NP–Complete.

This implies NP–Hard problems are at least as hard as NP–Complete problems. Therefore, they cannot be solved in polynomial time unless $P = NP$ (and maybe not then).

This use of an oracle, allows us to reduce co-NP-Complete problems to NP-Complete ones and vice versa.
QSAT

• QSAT is the problem to determine if an arbitrary fully quantified Boolean expression is true. Note: SAT only uses existential.
• QSAT is NP-Hard but may not be in NP.
• QSAT can be solved in polynomial space (PSPACE).
NP–Hard

Polynomial transformations are Turing reductions.

Thus, NP–Complete is a subset of NP–Hard. Co–NP–Complete also is a subset of NP–Hard. NP–Hard contains many other interesting problems.
NP-Easy

- **NP-Easy** is the set of function problems that are solvable in polynomial time by a deterministic Turing machine with an oracle for some decision problem in NP.
- That is, given an Oracle for some NP problem Y, if X is Turing reducible to Y in polynomial time then X is NP-Easy.
NP–Easy

Problem X need not be, but often is, NP–Complete.

In fact, X can be any problem in NP or Co–NP.

More to the point, an NP-Easy problem does not even need to be a decision problem – it can be an optimization problem or some other problem seeking a numerical rather than binary (yes/no answer).
Problem B in NP–Hard is **NP–Equivalent** when B reduces to some problem X in NP. That is, $B \leq_{PT} X$. This is, when B is also NP–Easy.

Since B is in NP–Hard, we already know there is a problem A in NP–Complete that reduces to B. That is, $A \leq_{PT} B$.

Since X is in NP, $X \leq_{PT} A$. Therefore, $X \leq_{PT} A \leq_{PT} B \leq_{PT} X$.

Thus, X, A, and B are all polynomially equivalent, and we can say

**Theorem.** Problems in NP–Equivalent are polynomial if and only if $P = NP$.

**Example:** Optimization version of Subset-Sum is NP-Equivalent.
NP-Easy and Equivalent

• NP-Easy -- these are problems that are polynomial when using an NP oracle (≤pt)
• NP-Equivalent is the class of NP-Easy and NP-Hard problems (assuming Turing rather than many-one reductions)
  – In essence this is the functional equivalent of NP-Complete but also of Co-NP-Complete since we can negate answers
SubsetSum Optimization

NP-Equivalence
SubsetSum Optimization (SSO)

\[ S = \{s_1, s_2, \ldots, s_n\} \]

set of positive integers and an integer \( B \).

Optimization: Find a subset of \( S \) whose values sum to the largest attainable value \( \leq B \)?

Strategy: Use Oracle for SubsetSum Decision Problem but only use it a polynomial number of times – Great care must be taken here as \( B \) takes only \( \lg B \) bits to represent
SSO is NP-Hard

• We can show $SS \leq_{PT} SSO$
• Let $[(s_1, s_2, \ldots, s_n), B]$ be an instance of SubsetSum (we’ll call it $SS$)
• We can ask the oracle for $SSO$ for the largest value $G \leq B$ such that some subsequence of $(s_1, s_2, \ldots, s_n)$ equals $G$. If its answer is $B$ we say “YES”; else we say “NO”
SSO is NP-Easy

- We can show $\text{SSO} \leq_{PT} \text{SubsetSum}$
- Let $[(s_1, s_2, \ldots, s_n), B]$ be an instance of SSO
- Again, our goal is to find the largest value $G \leq B$ such that some subsequence of $(s_1, s_2, \ldots, s_n)$ equals $G$
- The challenge is to do this in a number of steps that is polynomial in the size of the question. As any integer $k$ can be represented in $\log_2 k$ bits, we need to make sure we don’t ask more than $\log_2 S$ questions of our oracle, where $S$ is the length of the representation of $[(s_1, s_2, \ldots, s_n), B]$.
- Read the next slide very carefully
A Subtle Failure

• Let \([(s_1, s_2, \ldots, s_n), B]\) be an instance of SSO. Below sequence A is \((s_1, s_2, \ldots, s_n)\)

```
SUBSET–SUM–OPTIMIZATION(sequence A, int B) {
    for i=B downto 1
        if ( SubsetSum(A, i) ) then return i;
    return 0;
}
```

• This calls the oracle SS up to B times

• As B is \(2^{\log_2(B)}\), we might ask an exponential number of questions relative to the representation of our input parameter B

• As B can be as large as the sum of the sequence \((s_1, s_2, \ldots, s_n)\), the value B can be exponential in the size of the representation of our input and so our reduction is not polynomially bounded.
Using SubsetSum Oracle

SUBSET–SUM–OPTIMIZATION(sequence A, int B) {
    int best = B;
    for i = floor(log₂B) downto 0 do
        A = A + { 2^i };  // add to set; will succeed now
    for i = floor(log₂B) downto 0 do {
        A = A - { 2^i };  // remove from set
        if !SUBSET–SUM(A, best) then // 2^i was essential
            best = best - 2^i;  // reduce best
    }
    return best;
}
Example of SubsetSum Opt

• **Initial Values:**
  
  • \( A = \{1, 4, 5, 7\}, \text{ best } = b = 15 \)
  
  • \( A = \{1, 4, 5, 7, 8, 4, 2, 1\}, \text{ best } = 15 \)
  
  • \( A = \{1, 4, 5, 7, 4, 2, 1\}, \text{ best } = 15 \)
  
  • \( A = \{1, 4, 5, 7, 2, 1\}, \text{ best } = 15 \)
  
  • \( A = \{1, 4, 5, 7, 1\}, \text{ best } = 15-2 = 13 \)
  
  • \( A = \{1, 4, 5, 7\}, \text{ best } = 13 \)
Another Example

• Initial Values:
  • \( A = \{1, 4, 5, 7\}, \) best = \( b = 20 \)
  • \( A = \{1, 4, 5, 7, 16, 8, 4, 2, 1\}, \) best = 20
  • \( A = \{1, 4, 5, 7, 8, 4, 2, 1\}, \) best = 20
  • \( A = \{1, 4, 5, 7, 4, 2, 1\}, \) best = 20
  • \( A = \{1, 4, 5, 7, 2, 1\}, \) best = 20
  • \( A = \{1, 4, 5, 7, 1\}, \) best = 20
  • \( A = \{1, 4, 5, 7, 1\}, \) best = \( \frac{20-2}{1} = 18 \)
  • \( A = \{1, 4, 5, 7\}, \) best = \( \frac{18-1}{1} = 17 \)
Analysis

• Each loop has $O(\log_2 b)$ iterations, which is linear with respect to the size of $b$.
• Note that if we tried all values less that $b$, we would have $O(b)$ tries and that is exponential in $\log_2 b$, the size of $b$.
• The correct solution takes advantage of the NP-complete power of the oracle.
Minimum Colors for a Graph

- We know K-Color (KC) is NP Complete
- We can reduce KC to MinColor problem just by seeing if MinColor is \( \leq K \). Thus, MinColor is NP-Hard
- How do we reduce MinColor to KC asking only a log number of questions of the oracle for KC?
- Consider, if N nodes, then can easily N-Color
- Can we N/2-Color?
  - If so, then try N/4
  - If not, then try 3N/4
- This is a simple binary search for optimal value
More Complexity Topics
PSPACE

- PSPACE is set of problems solvable in polynomial space with unlimited time
  \[ \text{PSPACE} = \bigcup \text{SPACE}(n^k) \]
- PSPACE = co-PSPACE = NPSPACE
- PSPACE is a strict superset of CSLs
- PSPACE-Complete Problem is, given a regular expression \( e \) over \( \Sigma \), does \( e \) denote all strings in \( \Sigma^* \)?
- The above, while solvable, is potentially hard
- Another PSPACE-Complete problem is QSAT
- PSPACE is suspected to outside the P/NP hierarchy
EXPTIME and EXPSPACE

• EXPTIME is the set of problems solvable in $2^{p(n)}$ where $p$ is some polynomial.
• NEXPTIME is the set of problems solvable in $2^{p(n)}$ on a non-deterministic TM.
• EXPSPACE is the set of problems solvable in $2^{p(n)}$ space and unbounded time.
Elementary Functions

\[
\text{ELEMENTARY} = \bigcup_{k \in \mathbb{N}} \text{k-EXP} \\
= \text{DTIME}(2^n) \cup \text{DTIME}(2^{2^n}) \cup \text{DTIME}(2^{2^{2^n}}) \cup \cdots
\]
Alternating TM (ATM)

• ATM adds to NDTM notation the notion where, for each state q, q has one of the following properties: (accept, reject, ∨, ∧)
  – ∨ means mean accept the string if any final state reached after q is accepting
  – ∧ means mean accept the string if all final states reached after q are accepting

• AP = PSPACE where AP is class of problems solvable in polynomial time on an ATM
QSAT, Petri Net, Presburger

• QSAT is solvable by an alternating TM in polynomial time and polynomial space
• As noted, before, QSAT is PSPACE-Complete
• Petri net reachability is EXPSPACE-hard and requires 2-EXPTIME
• Presburger arithmetic is at least in 2-EXPTIME, at most in 3-EXPTIME, and can be solved by an ATM with n alternating quantifiers in doubly exponential time
Complexity Hierarchy

- $P \subseteq NP \subseteq \text{PSPACE} = \text{NPSPACE} \subseteq \text{EXPTIME} \subseteq \text{NEXPTIME} \subseteq \text{EXPSPACE}$
- $\not\subseteq 2\text{-EXPTIME} \not\subseteq 3\text{-EXPTIME} \not\subseteq \ldots \not\subseteq \text{ELEMENTARY} \not\subseteq \text{PRF} \not\subseteq \text{REC}$

- What if $P \neq \text{EXPTIME}$; At least one of these is true
  - $P \not\subseteq NP$
  - $NP \not\subseteq \text{PSPACE}$
  - $\text{PSPACE} \not\subseteq \text{EXPTIME}$

- If $NP \neq \text{NEXPTIME}$; At least of these is true
  - $NP \not\subseteq \text{PSPACE}$
  - $\text{PSPACE} \not\subseteq \text{EXPTIME}$
  - $\text{EXPTIME} \not\subseteq \text{NEXPTIME}$
    - Note that $\text{EXPTIME} = \text{NEXPTIME}$ iff $P=NP$
    - Note that $k\text{-EXPTIME} \not\subseteq (k+1)\text{-EXPTIME}$, $k>0$

- What If $\text{PSPACE} \neq \text{EXPSPACE}$; At least one of these is true
  - $\text{PSPACE} \not\subseteq \text{EXPTIME}$
  - $\text{EXPTIME} \not\subseteq \text{EXPSPACE}$

© UCF CS 277
FP and FNP

• FP is functional equivalent to P; $R(x,y)$ in FP if can provide value $y$ for input $x$ via a deterministic polynomial time algorithm

• FNP is functional equivalent to NP; $R(x,y)$ in FNP if can verify any pair $(x,y)$ via a deterministic polynomial time algorithm
TFNP

- TFNP is the subset of FNP where a solution always exists, i.e., there is a y for each x such that R(x,y).
  - Task of a TFNP algorithm is to find a y, given x, such that R(x,y)
  - Unlike FNP, the search for a y is always successful

- FNP properly contains TFNP contains FP (we don't know if proper)
Prime Factoring

• Prime factoring is defined as, given n and k, does n have a prime factor < k?
• Factoring is in NP and co-NP
  • Given candidate factor can check its primality in poly time and then see if it divides n
  • Given candidate set of factors can check their primalities, and see if product equals n; if so, and no candidate < k, then answer is no
Prime Factoring and TFNP

• Prime Factoring as a functional problem is in TFNP, but is it in FP?
• If TFNP in FP then TFNP = FP since FP contained in TFNP
• If that is so, then carrying out Prime Factoring is in FP and its decision problem is in P
  – If this is so, we must fear for encryption, most of which depends on difficulty of finding factors of a large number
More TFNP

- There is no known recursive enumeration of TFNP but there is of FNP
  - This is similar to total versus partially recursive functions (analogies are everywhere)
- It appears that TFNP does not have any complete problems!!!
  - But there are subclasses of TFNP that do have complete problems!!
Another Possible Analogy

• Is $P = (NP \cap Co-NP)$?
• Recall that $REC = (RE \cap co-RE)$
• The analogous result may not hold here
Turing vs m-1 Reductions

• In effect, our normal polynomial reduction (≤p) is a many-one polynomial time reduction as it just asks and then accepts its oracle’s answer

• In contrast, NP-Easy and NP-Equivalent employ a Turing machine polynomial time reduction (≤pt) that uses rather than mimics answers from its oracle
More Examples of NP Complete Problems
TipOver
Rules of Game

Numbers are height of crate stack; If could get 4 high out of way we can attain goal
Problematic OR Gadget

Can go out where did not enter
Single stack is two high; tipped over stack is one high, two long; red square is location of person travelling the towers
One directional Or gadget
AND Gadget

How AND Works
Variable Select Gadget

Tip A left to set x true; right to set x false
Can build bridge to go back but never to change choice
\(((x \lor \neg x \lor y) \land (\neg y \lor z \lor w) \land \neg w)\)
Win Strategy is NP-Complete

- TipOver win strategy is NP-Complete
- Minesweeper consistency is NP-Complete
- Phutball single move win is NP-Complete
  - Do not know complexity of winning strategy
- Checkers is really interesting
  - Single move to King is in P
  - Winning strategy is PSpace-Complete
Finding Triangle Strips

Adapted from presentation by
Ajit Hakke Patil
Spring 2010
Graphics Subsystem

• The graphics subsystem (GS) receives graphics commands from the application running on CPU/GPU over a bus, builds the image specified by the commands, and outputs the resulting image to display hardware

• Graphics Libraries:
  – OpenGL, DirectX.
Surface Visualization

- As Triangle Mesh
- Generated by triangulating the geometry
Triangle List vs Triangle Strip

• Triangle List: *Arbitrary ordering of triangles.*
• Triangle Strip: *A triangle strip is a sequential ordering of triangles.* i.e consecutive triangles share an edge
• In case of triangle lists we draw each triangle separately.
• So for drawing N triangles you need to call/send 3N vertex drawing commands/data.
• However, using a Triangle Strip reduces this requirement from 3N to N + 2, provided a single strip is sufficient.
Triangle List vs Triangle Strip

- four separate triangles: ABC, CBD, CDE, and EDF
- But if we know that it is a triangle strip or if we rearrange the triangles such that it becomes a triangle strip, then we can store it as a sequence of vertices ABCDEF
- This sequence would be decoded as a set of triangles ABC, BCD, CDE and DEF
- Storage requirement:
  - 3N => N + 2
Tri-strips example

• Single tri-strip that describes triangles is:
  1,2,3,4,1,5,6,7,8,9,6,10,1,2
K-Stripability

- Given some positive integer $K$ (less than the number of triangles).
- Can we create $K$ tri-strips for some given triangulation — no repeated triangles.
Triangle List vs Triangle Strip

// Draw Triangle Strip
glBegin(GL_TRIANGLES);
For each Vertex
{
    glVertex3f(x,y,z); //vertex
}
glEnd();

// Draw Triangle List
glBegin(GL_TRIANGLES);
For each Triangle
{
    glVertex3f(x1,y1,z1); // vertex 1
    glVertex3f(x2,y2,z2); // vertex 2
    glVertex3f(x3,y3,z3); // vertex 3
}
glEnd();
Problem Definition

- Given a triangulation $T = \{t_1, t_2, t_3, \ldots, t_n\}$. Find the triangle strip (sequential ordering) for it?
- Converting this to a decision problem.
- Formal Definition:
  Given a triangulation $T = \{t_1, t_2, t_3, \ldots, t_N\}$. Does there exists a triangle strip?
NP Proof

• Provided a witness of a ‘Yes’ instance of the problem, we can verify it in polynomial time by checking if the sequential triangles are connected.

• Cost of checking if the consecutive triangles are connected
  – For i to N -1
    • Check of $i_{th}$ and $(i+1)_{th}$ triangle are adjacent (have a common edge)
    • Three edge comparisons or six vertex comparisons
  – $\sim 6N$

• Hence it is in NP.
Dual Graph

- The *dual graph* of a triangulation is obtained by defining a vertex for each triangle and drawing an edge between two vertices if their corresponding triangles share an edge.
- This gives the triangulations *edge-adjacency* in terms of a graph.
- Cost of building a Dual Graph
  - $O(N^2)$
- e.g. $G'$ is a dual graph of G.
NP-Completeness

• To prove it’s NP-Complete we reduce a known NP-Complete problem to this one; the Hamiltonian Path Problem.

• Hamiltonian Path Problem:
  – Given: A Graph G = (V, E). Does G contains a path that visits every vertex exactly once?
NP-Completeness proof by restriction

• Accept an Instance of Hamiltonian Path, $G = (V, E)$, we restrict this graph to have max. degree = 3. The problem is still NP-Complete.
• Construct an Instance of HasTriangleStrip
  - $G' = G$
    • $V' = V$
    • $E' = E$
  - Let this be the dual graph $G' = (V', E')$ of the triangulation $T = \{t_1, t_2, t_3, .. t_N\}$.
    • $V' \sim$ Vertex $v_i$ represents triangle $t_i, i = 1 \text{ to } N$
    • $E' \sim$ An edge represents that two triangles are edge-adjacent (share an edge)
• Return HasTriangleStrip($T$)
NP-Completeness

- G will have a Hamiltonian Path iff G’ has one (they are the same).
- G’ has a Hamiltonian Path iff T has a triangle strip of length $N - 1$.
- T will have a triangle strip of length $N - 1$ iff G (G’) has a Hamiltonian Path.
- ‘Yes’ instance maps to ‘Yes’ instance. ‘No’ maps to ‘No.’
HP \leq_p \text{HasTriangleStrip}

- The ‘Yes/No’ instance maps to ‘Yes/No’ instance respectively and the transformation runs in polynomial time.
- Polynomial Transformation
- Hence finding Triangle Strip in a given triangulation is a NP-Complete Problem