Algorithms Week 10

Ljubomir Perković, DePaul University

We end the course by considering the following question:

We end the course by considering the following question:

How hard is it to solve a particular problem T?

We end the course by considering the following question:

How hard is it to solve a particular problem T?

Examples of problems we have been interested in are:

We end the course by considering the following question:

How hard is it to solve a particular problem T?

Examples of problems we have been interested in are:

Sorting

We end the course by considering the following question:

How hard is it to solve a particular problem T?

Examples of problems we have been interested in are:

Sorting, multiplication,

We end the course by considering the following question:

How hard is it to solve a particular problem T?

Examples of problems we have been interested in are:

Sorting, multiplication, closest pair of points.

We end the course by considering the following question:

How hard is it to solve a particular problem T?

Examples of problems we have been interested in are:

Sorting, multiplication, closest pair of points, longest increasing subsequence.

We end the course by considering the following question:

How hard is it to solve a particular problem T?

Examples of problems we have been interested in are:

Sorting, multiplication, closest pair of points, longest increasing subsequence, optimal binary search trees,

We end the course by considering the following question:

How hard is it to solve a particular problem T?

Examples of problems we have been interested in are:

Sorting, multiplication, closest pair of points, longest increasing subsequence, optimal binary search trees, text segmentation,

We end the course by considering the following question:

How hard is it to solve a particular problem T?

Examples of problems we have been interested in are:

Sorting, multiplication, closest pair of points, longest increasing subsequence, optimal binary search trees, text segmentation, activity selection,

We end the course by considering the following question:

How hard is it to solve a particular problem T?

Examples of problems we have been interested in are:

Sorting, multiplication, closest pair of points, longest increasing subsequence, optimal binary search trees, text segmentation, activity selection, subset-sum,

We end the course by considering the following question:

How hard is it to solve a particular problem T?

Examples of problems we have been interested in are:

Sorting, multiplication, closest pair of points, longest increasing subsequence, optimal binary search trees, text segmentation, activity selection, subset-sum, maximum sum subvector,

We end the course by considering the following question:

How hard is it to solve a particular problem T?

Examples of problems we have been interested in are:

Sorting, multiplication, closest pair of points, longest increasing subsequence, optimal binary search trees, text segmentation, activity selection, subset-sum, maximum sum subvector, edit distance,

We end the course by considering the following question:

How hard is it to solve a particular problem T?

Examples of problems we have been interested in are:

Sorting, multiplication, closest pair of points, longest increasing subsequence, optimal binary search trees, text segmentation, activity selection, subset-sum, maximum sum subvector, edit distance, shortest path,

We end the course by considering the following question:

How hard is it to solve a particular problem T?

Examples of problems we have been interested in are:

Sorting, multiplication, closest pair of points, longest increasing subsequence, optimal binary search trees, text segmentation, activity selection, subset-sum, maximum sum subvector, edit distance, shortest path, minimum spanning tree,

We end the course by considering the following question:

How hard is it to solve a particular problem T?

Examples of problems we have been interested in are:

Sorting, multiplication, closest pair of points, longest increasing subsequence, optimal binary search trees, text segmentation, activity selection, subset-sum, maximum sum subvector, edit distance, shortest path, minimum spanning tree, strongly connected components,

We end the course by considering the following question:

How hard is it to solve a particular problem T?

Examples of problems we have been interested in are:

Sorting, multiplication, closest pair of points, longest increasing subsequence, optimal binary search trees, text segmentation, activity selection, subset-sum, maximum sum subvector, edit distance, shortest path, minimum spanning tree, strongly connected components, DFS, . . .

We end the course by considering the following question:

How hard is it to solve a particular problem T?

Examples of problems we have been interested in are:

Sorting, multiplication, closest pair of points, longest increasing subsequence, optimal binary search trees, text segmentation, activity selection, subset-sum, maximum sum subvector, edit distance, shortest path, minimum spanning tree, strongly connected components, DFS, BFS.

We end the course by considering the following question:

How hard is it to solve a particular problem T?

Examples of problems we have been interested in are:

Sorting, multiplication, closest pair of points, longest increasing subsequence, optimal binary search trees, text segmentation, activity selection, subset-sum, maximum sum subvector, edit distance, shortest path, minimum spanning tree, strongly connected components, DFS, BFS, topological sorting, ...

We end the course by considering the following question:

How hard is it to solve a particular problem Π?

Examples of problems we have been interested in are:

Sorting, multiplication, closest pair of points, longest increasing subsequence, optimal binary search trees, text segmentation, activity selection, subset-sum, maximum sum subvector, edit distance, shortest path, minimum spanning tree, strongly connected components, DFS, BFS, topological sorting, ...

There are two ways to approach the question:

1. Algorithmic (or upper-bound analysis)

We end the course by considering the following question:

How hard is it to solve a particular problem T?

Examples of problems we have been interested in are:

Sorting, multiplication, closest pair of points, longest increasing subsequence, optimal binary search trees, text segmentation, activity selection, subset-sum, maximum sum subvector, edit distance, shortest path, minimum spanning tree, strongly connected components, DFS, BFS, topological sorting, ...

There are two ways to approach the question:

1. Algorithmic (or upper-bound analysis)
2. Computational complexity (or lower-bound analysis)

One way to answer our fundamental question is to find an algorithm to solve problem $\mathcal{T}$.

Algorithmic approach

One way to answer our fundamental question is to find an algorithm to solve problem T. This typically involves the following steps:

One way to answer our fundamental question is to find an algorithm to solve problem $\mathcal{T}$. This typically involves the following steps:

1. Develop the algorithm.

One way to answer our fundamental question is to find an algorithm to solve problem T. This typically involves the following steps:

1. Develop the algorithm.
2. Prove its correctness.

One way to answer our fundamental question is to find an algorithm to solve problem $\mathcal{T}$. This typically involves the following steps:

1. Develop the algorithm.
2. Prove its correctness.
3. Analyze its running time to determine that it solves a problem of size n in time $\Theta(f(n))$ (or $O(f(n))$) for some function f.

One way to answer our fundamental question is to find an algorithm to solve problem T. This typically involves the following steps:

1. Develop the algorithm.
2. Prove its correctness.
3. Analyze its running time to determine that it solves a problem of size n in time $\Theta(f(n))$ (or $O(f(n))$) for some function f.
4. Analyze whether the algorithm can be improved.

Algorithmic approach

One way to answer our fundamental question is to find an algorithm to solve problem T. This typically involves the following steps:

1. Develop the algorithm.
2. Prove its correctness.
3. Analyze its running time to determine that it solves a problem of size n in time $\Theta(f(n))$ (or $O(f(n))$) for some function f.
4. Analyze whether the algorithm can be improved.

This process gives you an upper bound on the running time of the best algorithm solving the problem.

Another way to answer the question of how hard problem T is

Another way to answer the question of how hard problem T is is to prove a claim of the following form:

Another way to answer the question of how hard problem T is is to prove a claim of the following form:

Claim

Every algorithm that solves problem T will have a running time of at least $\Omega(g(n))$ steps on an input of size n.

Another way to answer the question of how hard problem T is is to prove a claim of the following form:

Claim

Every algorithm that solves problem T will have a running time of at least $\Omega(g(n))$ steps on an input of size n.

It gives us a lower bound on the running time of any algorithm solving the problem.

Consider the following search problem:

Input: An array $A[1..N]$ of integers in sorted order and an integer x.

Output: Yes, if x is in A; No, otherwise.

Consider the following search problem:

Input: An array $A[1..N]$ of integers in sorted order and an integer x.

Output: Yes, if x is in A; No, otherwise.

The algorithmic approach would be to propose binary search as an algorithm that solves the problem.

Consider the following search problem:

Input: An array $A[1..N]$ of integers in sorted order and an integer x.

Output: Yes, if x is in A; No, otherwise.

The algorithmic approach would be to propose binary search as an algorithm that solves the problem.

Running time: $O(\log n)$.

The complexity (lower-bound) approach would be to prove the following claim

Claim

$\log n$ *queries are necessary in the worst-case to search a sorted array of size* n.

The complexity (lower-bound) approach would be to prove the following claim

Claim

$\log n$ queries are necessary in the worst-case to search a sorted array of size n.

Proof.

We can do no better than reduce the search space by a half, in the worst case, with each query. Therefore, $\log n$ questions are required to reduce the search space to size 1. □

Consider the problem of sorting n numbers using only comparisons:

- Input: An array $A[1 \ldots N]$ of integers
- Output: Array A in sorted order (say, non-decreasing).

Consider the problem of sorting n numbers using only comparisons:

- Input: An array $A[1..N]$ of integers

- Output: Array A in sorted order (say, non-decreasing).

The algorithmic approach would be to propose, say, Mergesort whose running time is $\Theta(n \log n)$.

Consider the problem of sorting n numbers using only comparisons:

- Input: An array $A[1..N]$ of integers
- Output: Array A in sorted order (say, non-decreasing).

The algorithmic approach would be to propose, say, Mergesort whose running time is $\Theta(n \log n)$. The lower-bound approach would be to prove:

Claim

Sorting n numbers requires $\Omega(n \log n)$ steps.

Consider the problem of sorting n numbers using only comparisons:

- Input: An array $A[1..N]$ of integers
- Output: Array A in sorted order (say, non-decreasing).

The algorithmic approach would be to propose, say, Mergesort whose running time is $\Theta(n \log n)$. The lower-bound approach would be to prove:

Claim

Sorting n numbers requires $\Omega(n \log n)$ steps.

Proof.

There are $n!$ possible orderings of n numbers.

Consider the problem of sorting n numbers using only comparisons:

- **Input:** An array $A[1..N]$ of integers
- **Output:** Array A in sorted order (say, non-decreasing).

The algorithmic approach would be to propose, say, Mergesort whose running time is $\Theta(n \log n)$. The lower-bound approach would be to prove:

Claim

Sorting n numbers requires $\Omega(n \log n)$ steps.

Proof.

There are $n!$ possible orderings of n numbers. Each comparison can only rule out on half of the possibilities,

Example: sorting

Consider the problem of sorting n numbers using only comparisons:

- Input: An array $A[1..N]$ of integers

- Output: Array A in sorted order (say, non-decreasing).

The algorithmic approach would be to propose, say, Mergesort whose running time is $\Theta(n \log n)$. The lower-bound approach would be to prove:

Claim

Sorting n numbers requires $\Omega(n \log n)$ steps.

Proof.

There are $n!$ possible orderings of n numbers. Each comparison can only rule out on half of the possibilities, so we need at least $\log(n!) = \Theta(n \log n)$ comparisons.

Example: sorting

Consider the problem of sorting n numbers using only comparisons:

- Input: An array $A[1..N]$ of integers

- Output: Array A in sorted order (say, non-decreasing).

The algorithmic approach would be to propose, say, Mergesort whose running time is $\Theta(n \log n)$. The lower-bound approach would be to prove:

Claim

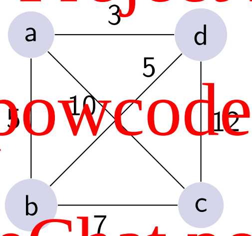
Sorting n numbers requires $\Omega(n \log n)$ steps.

Proof.

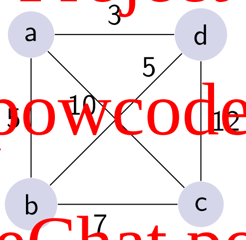
There are $n!$ possible orderings of n numbers. Each comparison can only rule out on half of the possibilities, so we need at least $\log(n!) = \Theta(n \log n)$ comparisons. So any algorithm requires $\Omega(n \log n)$ comparisons. $\square$

Example: Travelling Salesman Problem

Given a *complete* weighted graph $G = (V, E)$, find a cycle in G that visits every vertex once such that the sum of the weights of the edges on the cycle is minimized.



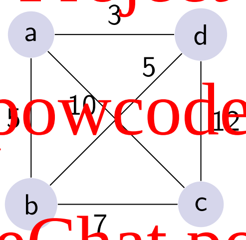
Given a *complete* weighted graph $G = (V, E)$, find a cycle in G that visits every vertex once such that the sum of the weights of the edges on the cycle is minimized.



Obvious backtracking algorithm: check every possible permutation of the vertices to find the minimum cost cycle.

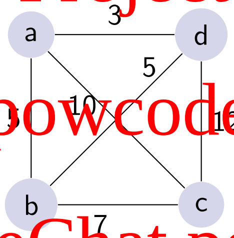
Example: Travelling Salesman Problem

Given a *complete* weighted graph $G = (V, E)$, find a cycle in G that visits every vertex once such that the sum of the weights of the edges on the cycle is minimized.



Obvious backtracking algorithm: check every possible permutation of the vertices to find the minimum cost cycle.
Running time: $\Theta(n!) = \Theta(2^{n \log n})$.

Given a *complete* weighted graph $G = (V, E)$, find a cycle in G that visits every vertex once such that the sum of the weights of the edges on the cycle is minimized.



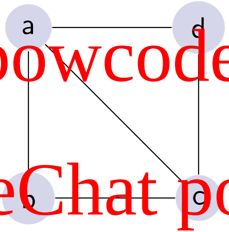
The obvious lower bound for the running time of any algorithm solving the TSP is $\Theta(n)$ (because it takes that much time to list the vertices of the cycle).

Given a graph $G = (V, E)$ and a value k, the problem is to find a valid coloring of the vertices of G using at most k colors.

Example: Graph coloring

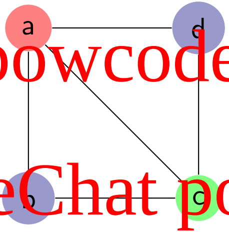
Given a graph $G = (V, E)$ and a value k, the problem is to find a valid coloring of the vertices of G using at most k colors.

A coloring is valid if no two adjacent vertices are colored the same color.



Given a graph $G = (V, E)$ and a value k, the problem is to find a valid coloring of the vertices of G using at most k colors.

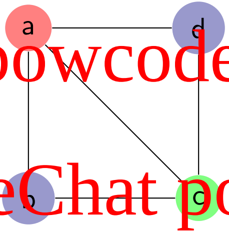
A coloring is valid if no two adjacent vertices are colored the same color.



Example: Graph coloring

Given a graph $G = (V, E)$ and a value k, the problem is to find a valid coloring of the vertices of G using at most k colors.

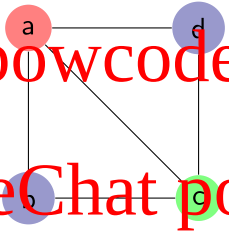
A coloring is valid if no two adjacent vertices are colored the same color.



Obvious backtracking algorithm: try every possible k-coloring of the vertices, and check whether any one of is valid.
Running time: $O(k^n n^2)$.

Given a graph $G = (V, E)$ and a value k, the problem is to find a valid coloring of the vertices of G using at most k colors.

A coloring is valid if no two adjacent vertices are colored the same color.



A lower bound for the running time of any algorithm solving the graph coloring problem is $\Theta(n)$ (it takes that much time to assign a color to each vertex).

The graph coloring problem and TSP have been extensively studied.

The graph coloring problem and TSP have been extensively studied.

However, there is no known algorithm that solves either problem in time that is a polynomial function of the input.

The graph coloring problem and TSP have been extensively studied.

However, there is no known algorithm that solves either problem in time that is a polynomial function of the input.

Conversely, there is no known lower bound on the running time of an algorithm that solves either problem that is asymptotically greater than a polynomial function of the input size.

The graph coloring problem and TSP have been extensively studied.

However, there is no known algorithm that solves either problem in time that is a polynomial function of the input.

Conversely, there is no known lower bound on the running time of an algorithm that solves either problem that is asymptotically greater than a polynomial function of the input size.

Unlike search in a sorted array and sorting, there is a gap between the known lower bound and upper bound on solving these problems.

What do you do if you cannot find a fast algorithm for your problem?

What do you do if you cannot find a fast algorithm for your problem? You can give several answers:

What do you do if you cannot find a fast algorithm for your problem? You can give several answers:

1. "I can't do it!" This argument can only covince your boss that you are not smart enough and need to be fired.

What do you do if you cannot find a fast algorithm for your problem? You can give several answers:

1. "I can't do it!" This argument can only covince your boss that you are not smart enough and need to be fired.
2. "I can prove it can't be done!" This would gain you respect in the eyes of your boss. Unfortunately, this is also very hard.

What do you do if you cannot find a fast algorithm for your problem? You can give several answers:

1. "I can't do it!" This argument can only covince your boss that you are not smart enough and need to be fired.
2. "I can prove it can't be done!" This would gain you respect in the eyes of your boss. Unfortunately, this is also very hard.
3. "I couldn't do it, but many others have tried and failed too". While not the most satisfactory response, at least you won't be fired.

What do you do if you cannot find a fast algorithm for your problem? You can give several answers:

1. "I can't do it!" This argument can only covince your boss that you are not smart enough and need to be fired.
2. "I can prove it can't be done!" This would gain you respect in the eyes of your boss. Unfortunately, this is also very hard.
3. "I couldn't do it, but many others have tried and failed too". While not the most satisfactory response, at least you won't be fired.

So, what we want to do is this: if we are unable to solve a hard problem, we would like to recognize that our problem is known to be equivalent to another problem known to be hard.

What do you do if you cannot find a fast algorithm for your problem? You can give several answers:

1. "I can't do it!" This argument can only covince your boss that you are not smart enough and need to be fired.
2. "I can prove it can't be done!" This would gain you respect in the eyes of your boss. Unfortunately, this is also very hard.
3. "I couldn't do it, but many others have tried and failed too". While not the most satisfactory response, at least you won't be fired.

So, what we want to do is this: if we are unable to solve a hard problem, we would like to recognize that our problem is known to be equivalent to another problem known to be hard. How can we make this idea more formal?

Definition

An instance of a problem is a list of the input parameters for the problem.

Graph coloring is specified by:

Input: A graph G, and a value k.

Output: A valid k-coloring, if one exists.

Definition

An instance of a problem is a list of the input parameters for the problem.

Graph coloring is specified by:

Input: A graph G, and a value k.

Output: A valid k-coloring, if one exists.

$< G, k >$ is an instance of the graph coloring problem.

Definition

An instance of a problem is a list of the input parameters for the problem.

Graph coloring is specified by:

Input: A graph G, and a value k.

Output: A valid k-coloring, if one exists.

$< G, k >$ is an instance of the graph coloring problem.

The TSP problem is specified by:

Input: A complete graph G, and weights $w(e)$ on the edges of G.

Output: A cycle in G that visits every vertex once such that the sum of the weights of the edges on the cycle is minimized.

Definition

An instance of a problem is a list of the input parameters for the problem.

Graph coloring is specified by:

- Input: A graph G, and a value k.
- Output: A valid k-coloring, if one exists.

$< G, k >$ is an instance of the graph coloring problem.

The TSP problem is specified by:

- Input: A complete graph G, and weights $w(e)$ on the edges of G.
- Output: A cycle in G that visits every vertex once such that the sum of the weights of the edges on the cycle is minimized.

$< G, w >$ is an instance of the TSP problem.

We can formulate each problem as a decision problem: a decision problem is a problem whose answer is either "YES" or "NO".

We can formulate each problem as a decision problem: a decision problem is a problem whose answer is either "YES" or "NO".

Then, the set of instances of a problem T for which the answer is "YES" forms a set we will call T.

We can formulate each problem as a decision problem: a decision problem is a problem whose answer is either "YES" or "NO".

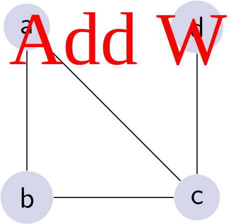
Then, the set of instances of a problem T for which the answer is "YES" forms a set we will call T.

For example, if $T = \{< G, k >: G$ has a valid k-coloring$\}$ be the set of instances of k colorable graphs then

We can formulate each problem as a decision problem: a decision problem is a problem whose answer is either "YES" or "NO".

Then, the set of instances of a problem T for which the answer is "YES" forms a set we will call T.

For example, if $T = \{< G, k >: G \text{ has a valid } k\text{-coloring}\}$ be the set of instances of k colorable graphs then

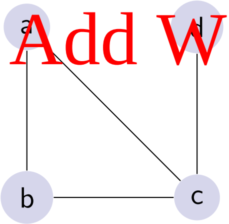


Formalism: decision problem

We can formulate each problem as a decision problem: a decision problem is a problem whose answer is either "YES" or "NO".

Then, the set of instances of a problem T for which the answer is "YES" forms a set we will call T.

For example, if $T = \{< G, k >: G \text{ has a valid } k\text{-coloring}\}$ be the set of instances of k colorable graphs then

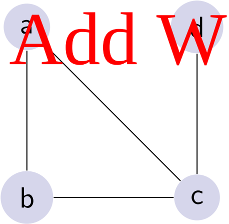


- $< G, 2 > \notin T$.

We can formulate each problem as a decision problem: a decision problem is a problem whose answer is either "YES" or "NO".

Then, the set of instances of a problem T for which the answer is "YES" forms a set we will call T.

For example, if $T = \{< G, k >: G \text{ has a valid } k\text{-coloring}\}$ be the set of instances of k colorable graphs then

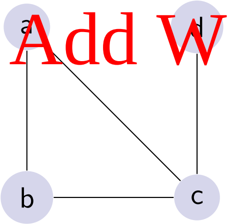


- $< G, 2 > \notin T$.
- $< G, 3 > \in T$.

We can formulate each problem as a decision problem: a decision problem is a problem whose answer is either "YES" or "NO".

Then, the set of instances of a problem T for which the answer is "YES" forms a set we will call T.

For example, if $T = \{< G, k >: G$ has a valid k-coloring$\}$ be the set of instances of k colorable graphs then



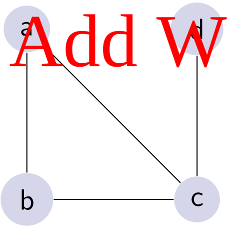
- $< G, 2 > \notin T$.
- $< G, 3 > \in T$.
- $< G, 4 > \in T$.

Formalism: decision problem

We can formulate each problem as a decision problem: a decision problem is a problem whose answer is either "YES" or "NO".

Then, the set of instances of a problem T for which the answer is "YES" forms a set we will call T.

For example, if $T = \{< G, k >: G \text{ has a valid } k\text{-coloring}\}$ be the set of instances of k colorable graphs then



- $< G, 2 > \notin T$.
- $< G, 3 > \in T$.
- $< G, 4 > \in T$.
- $< G, k > \in T$, where $k \geq 3$.

Formalism: decision problem

Every problem can be turned into a decision problem.

Every problem can be turned into a decision problem. For example, the decision version of the graph coloring problem is:

Input: Graph G and integer k.

Output: "YES" if G is k-colorable, "NO" otherwise.

Every problem can be turned into a decision problem. For example, the decision version of the graph coloring problem is:

Input: Graph G and integer k.

Output: "YES" if G is k-colorable, "NO" otherwise.

In the theory of problem complexity we will develop shortly, we will only consider decision problems.

Every problem can be turned into a decision problem. For example, the decision version of the graph coloring problem is:

Input:	Graph G and integer k.
Output:	"YES" if G is k-colorable, "NO" otherwise.

In the theory of problem complexity we will develop shortly, we will only consider decision problems.

We need to do this to develop the complexity theory but ...

Every problem can be turned into a decision problem. For example, the decision version of the graph coloring problem is:

Input: Graph G and integer k.

Output: "YES" if G is k-colorable, "NO" otherwise.

In the theory of problem complexity we will develop shortly, we will only consider decision problems.

We need to do this to develop the complexity theory but ... this is OK because every optimization problem can be rephrased as a decision problem.

The shortest path problem is specified by:

- Input: A graph $G = (V, E)$; positive weights $w(e)$ for every $e \in E$ and specified nodes $u, v \in V$.

  Output: The length of the shortest path from u to v in G.

Shortest path problem as a decision problem

The shortest path problem is specified by:

- Input: A graph $G = (V, E)$; positive weights $w(e)$ for every $e \in E$ and specified nodes $u, v \in V$.
- Output: The length of the shortest path from u to v in G.

Note that an instance of this problem is $< G, w, u, v >$.

The shortest path problem is specified by:

Input: A graph $G = (V, E)$; positive weights $w(e)$ for every $e \in E$ and specified nodes $u, v \in V$.

Output: The length of the shortest path from u to v in G.

Note that an instance of this problem is $< G, w, u, v >$.

The decision version of the shortest path problem is:

Input: A weighted graph $G = (V, E)$, positive weights $w(e)$ for every $e \in E$, specified nodes $u, v \in V$ and an integer k.

Output: "YES", if the length of the shortest path from u to v in G is k.

The shortest path problem is specified by:

- **Input:** A graph $G = (V, E)$, positive weights $w(e)$ for every $e \in E$ and specified nodes $u, v \in V$.

  Output: The length of the shortest path from u to v in G.

Note that an instance of this problem is $< G, w, u, v >$.

The decision version of the shortest path problem is:

Input: A weighted graph $G = (V, E)$, positive weights $w(e)$ for every $e \in E$, specified nodes $u, v \in V$ and an integer k.

Output: "YES", if the length of the shortest path from u to v in G is k.

An instance of the decision version of the shortest path problem is $< G, w, u, v, k >$.

Shortest path problem as a decision problem

We don't lose anything by only considering decision problems...

Shortest path problem as a decision problem

We don't lose anything by only considering decision problems...
because if we have an algorithm B that solves the decision version of a problem, then we can construct an algorithm A that solves the optimization version of the problem, and vice versa

Shortest path problem as a decision problem

We don't lose anything by only considering decision problems... because if we have an algorithm B that solves the decision version of a problem, then we can construct an algorithm A that solves the optimization version of the problem, and vice versa.

For example, suppose that we have an algorithm B that solves the optimization version of the shortest path problem.

We don't lose anything by only considering decision problems... because if we have an algorithm B that solves the decision version of a problem, then we can construct an algorithm A that solves the optimization version of the problem, and vice versa.

For example, suppose that we have an algorithm B that solves the optimization version of the shortest path problem. We want to use B to construct an algorithm A that solves the decision version of the problem.

Shortest path problem as a decision problem

We don't lose anything by only considering decision problems... because if we have an algorithm B that solves the decision version of a problem, then we can construct an algorithm A that solves the optimization version of the problem, and vice versa.

For example, suppose that we have an algorithm B that solves the optimization version of the shortest path problem. We want to use B to construct an algorithm A that solves the decision version of the problem. Here is how we do it:


```
Algorithm A(G,w,u,v,k)
  // Using solution to optimization problem to obtain
  // a solution to the decision problem
  if B(G,w,u,v) = k then
    output "YES"
  else
    output "NO"
```

We don't lose anything by only considering decision problems... because if we have an algorithm B that solves the decision version of a problem, then we can construct an algorithm A that solves the optimization version of the problem, and vice versa

Now suppose that we have an algorithm B that solves the decision version of the shortest path problem.

Shortest path problem as a decision problem

We don't lose anything by only considering decision problems... because if we have an algorithm B that solves the decision version of a problem, then we can construct an algorithm A that solves the optimization version of the problem, and vice versa

Now suppose that we have an algorithm B that solves the decision version of the shortest path problem. We use B to construct an algorithm A for the optimization version of the problem as follows


```
Algorithm B(G,w,u,v)
  // Using solution to decision problem to obtain
  // a solution to the optimization problem
  for k ← 1 to (|V| - 1)M // M ← max edge weight
    if A(G,w,u,v,k) ← "YES" then
      output k
      return
  output "INFINITY"
```

A complexity class is intuitively a set of problems of same difficulty.

A complexity class is intuitively a set of problems of same difficulty.

We introduce our first complexity class:

Definition

P is the set of all decision problems that can be solved in polynomial time by a deterministic algorithm.

A complexity class is intuitively a set of problems of same difficulty.

We introduce our first complexity class:

Definition

P is the set of all decision problems that can be solved in polynomial time by a deterministic algorithm.

Here polynomial time means time that is a polynomial function of the input, i.e. equal to $\Theta(n^k)$ for some constant $k > 0$, where n is the size of the input.

Among the problems we discussed in class, the following are in P:

1. Polynomial evaluation
2. Searching a sorted array
3. Sorting
4. Maximum sum subvector
5. Longest increasing subsequence
6. Activity selection
7. Huffman code
8. ...

The following problems are not known to be in P (and most experts believe they are not in):

1. Graph coloring
2. TSP
3. Subset-sum

These problems happen to have a property in common:

The following problems are not known to be in P (and most experts believe they are not in):

1. Graph coloring
2. TSP
3. Subset-sum

These problems happen to have a property in common: a solution for each of them can be verified quickly (in polynomial time).

Consider the graph $G = (V, E)$ where $V = \{A, B, C, D, E, F, G\}$ and $E = \{(A, D), (A, B), (B, D), (B, E), (D, E), (B, G), (E, G), (B, C), (E, F), (C, E)\}$.

Consider the graph $G = (V, E)$ where $V = \{A, B, C, D, E, F, G\}$ and $E =$
$\{(A, D), (A, B), (B, D), (B, E), (D, E), (B, G), (E, G), (B, C), (E, F), (C, F)\}$.

Is G 3-colorable?

Consider the graph $G = (V, E)$ where $V = \{A, B, C, D, E, F, G\}$ and $E =$
$\{(A, D), (A, B), (B, D), (B, F), (D, E), (B, G), (E, G), (B, C), (E, F), (C, F)\}$.

Is G 3-colorable?

If we have to devise the coloring, this is not easy in general...

Consider the graph $G = (V, E)$ where $V = \{A, B, C, D, E, F, G\}$ and $E =$

$\{(A, D), (A, B), (B, D), (B, E), (D, E), (B, G), (E, G), (B, C), (E, F), (C, F)\}$

Is G 3-colorable?

If we have to devise the coloring, this is not easy in general... But if someone claims that it is true, they can quickly convince us by giving a 3-coloring of G:

A	B	C	D	E	F	G
blue	green	blue	red	blue	green	red

Example: verifying a solution in polynomial time

Consider the graph $G = (V, E)$ where $V = \{A, B, C, D, E, F, G\}$ and $E =$
$\{(A, D), (A, B), (B, D), (B, E), (D, E), (B, G), (E, G), (B, C), (E, F), (C, F)\}$.

Is G 3-colorable?

If we have to devise the coloring, this is not easy in general... But if someone claims that it is true, they can quickly convince us by giving a 3-coloring of G:

A	B	C	D	E	F	G
blue	green	blue	red	blue	green	red

Verifying that a coloring is valid takes time $\Theta(n^2)$ for a graph with n nodes.

Formalism: verification

An algorithm A verifies a decision problem T if for every instance $x \in T$, there is some witness y such that $A(x, y) =$ "YES".

An algorithm A verifies a decision problem T if for every instance $x \in T$, there is some witness y such that $A(x, y) =$ "YES".

The witness y is a piece of information that "proves" that $x \in T$.

An algorithm A verifies a decision problem T if for every instance $x \in T$, there is some witness y such that $A(x, y) =$ "YES".

The witness y is a piece of information that "proves" that $x \in T$. We require that $|y|$ be polynomial in $|x|$.

An algorithm A verifies a decision problem T if for every instance $x \in T$, there is some witness y such that $A(x, y) =$ "YES".

The witness y is a piece of information that "proves" that $x \in T$. We require that $|y|$ be polynomial in $|x|$.

Let us clarify the relationship between solving and verifying:

Solving: Given instance x of problem T of size $|x| = n$ we want decide whether $x \in T$ or $x \notin T$.

Verifying: Given instance x of size $|x| = n$ and a witness y of size $|y| \leq c|x|$ for some constant $c > 0$, we want to verify that the witness y proves that $x \in T$.

Given a TSP problem, a witness y could be the list of nodes along a cycle...

Given a TSP problem, a witness y could be the list of nodes along a cycle... and the verification algorithm would be:

Given a TSP problem, a witness y could be the list of nodes along a cycle. . . and the verification algorithm would be:

1. Check that each node in G, except the first (and last) node in the cycle, appears exactly once. (If each node appears once in the cycle, then because G is a complete graph the edges in the path must be valid).

Given a TSP problem, a witness y could be the list of nodes along a cycle... and the verification algorithm would be:

1. Check that each node in G, except the first (and last) node in the cycle, appears exactly once. (If each node appears once in the cycle, then because G is a complete graph the edges in the path must be valid).

2. Compute the sum the weights of the edges and check that it is at most k.

Given a TSP problem, a witness y could be the list of nodes along a cycle. . . and the verification algorithm would be:

1. Check that each node in G, except the first (and last) node in the cycle, appears exactly once. (If each node appears once in the cycle, then because G is a complete graph the edges in the path must be valid).

2. Compute the sum the weights of the edges and check that it is at most k.

The running time of this verification algorithm is $\Theta(n)$ (n steps to verify that the nodes appear the proper number of times, and n steps to sum the edges in the cycle).

Definition

NP is the set of all decision problems that can be verified in polynomial time by a deterministic algorithm.

Definition

NP is the set of all decision problems that can be verified in polynomial time by a deterministic algorithm.

The following problems are contained in *NP*:

1. Graph coloring
2. TSP
3. Subset-sum

Theorem

If a problem can be solved, then it can be verified. So $P \subseteq NP$.

Theorem

If a problem can be solved, then it can be verified. So $P \subseteq NP$.

Proof.

Let $Q \in P$ be a problem. Since $Q \in P$, there must be an algorithm A that solves problem Q in polynomial time. Let x be an instance of Q. Then,

- if $x \in Q$ then $A(x) =$ "*YES*",
- if $x \notin Q$ then $A(x) =$ "*no*".

We need to produce an algorithm B that given x and a witness y verifies that $x \in Q$. Here is is:


```
Algorithm B(x, y)
if A(x) ← "YES" then
  return "YES"
else
  return "NO"
```

The algorithm B simply ignores the witness and solves the problem using the algorithm A. So for any x and for any witness y, $B(x, y)$ returns "YES" if and only if $x \in Q$.

Note that if A runs in polynomial time, then B will run in polynomial time as well. So, $Q \in NP$ and we can conclude that $P \subseteq NP$.

We know that $P \subseteq NP$.

We know that $P \subseteq NP$. This means that the set of problems in P are all contained in the set NP.

We know that $P \subseteq NP$. This means that the set of problems in P are all contained in the set NP. But what about the other direction?

We know that $P \subseteq NP$. This means that the set of problems in P are all contained in the set NP. But what about the other direction?

Is $NP = P$?

We know that $P \subseteq NP$. This means that the set of problems in P are all contained in the set NP. But what about the other direction?

Is $NP = P$?

Or is there some problem in $NP - P$?

We know that $P \subseteq NP$. This means that the set of problems in P are all contained in the set NP. But what about the other direction?

Is $NP = P$?

Or is there some problem in $NP - P$?

The question is very much open.

One way to approach the resolution of the P versus NP question is to try to understand the structure of the class NP.

One way to approach the resolution of the P versus NP question is to try to understand the structure of the class NP.

The NP-complete problems are the "hardest" problems in NP.

One way to approach the resolution of the P versus NP question is to try to understand the structure of the class NP.

The NP-complete problems are the "hardest" problems in NP.

What we mean by "hardest" is that if we could solve any of the NP-complete problems in polynomial time, then we could solve every problem in NP in polynomial time.

One way to approach the resolution of the *P* versus *NP* question is to try to understand the structure of the class NP.

The *NP*-complete problems are the "hardest" problems in NP.

What we mean by "hardest" is that if we could solve any of the NP-complete problems in polynomial time, then we could solve every problem in *NP* in polynomial time.

In order to develop the theory of *NP*-completeness, we define the notion of reduction.

Consider the problem of scheduling exams:

Input: List of lists $L_1, L_2, \ldots, L_n$ of students in classes $1, 2, \ldots, n$, respectively.

Output: The smallest number of exam periods so that all exams can be scheduled with no conflicts.

Consider the problem of scheduling exams:

Input: List of lists $L_1, L_2, \ldots, L_n$ of students in classes $1, 2, \ldots, n$, respectively.

Output: The smallest number of exam periods so that all exams can be scheduled with no conflicts.

Note that this is an optimization problem.

Consider the problem of scheduling exams:

Input: List of lists $L_1, L_2, ..., L_n$ of students in classes $1, 2, ..., n$, respectively.

Output: The smallest number of exam periods so that all exams can be scheduled with no conflicts.

Note that this is an optimization problem. Let us rephrase the problem as a decision problem:

Input: A list of students for each class, and an integer k.

Output: Output "YES" if k time periods are sufficient to schedule the exams, and "NO" otherwise.

The exam scheduling problem can be solved by translating it (reducing it) to the graph coloring problem and then solving the graph coloring problem.

The exam scheduling problem can be solved by translating it (reducing it) to the graph coloring problem and then solving the graph coloring problem.

For example, let $L_1 = \{A, B, D\}$, $L_2 = \{B, D, E\}$, $L_3 = \{A, C, F\}$ and suppose that $k = 2$.

The exam scheduling problem can be solved by translating it (reducing it) to the graph coloring problem and then solving the graph coloring problem.

For example, let $L_1 = \{A, B, D\}$, $L_2 = \{B, D, E\}$, $L_3 = \{A, C, F\}$ and suppose that $k = 2$. We need to construct an instance of the graph coloring problem $< G, k >$.

The exam scheduling problem can be solved by translating it (reducing it) to the graph coloring problem and then solving the graph coloring problem.

For example, let $L_1 = \{A, B, D\}$, $L_2 = \{B, D, E\}$, $L_3 = \{A, C, F\}$ and suppose that $k = 2$. We need to construct an instance of the graph coloring problem $< G, k >$.

Let each list be represented by a node of G, and let two nodes be adjacent if they share a student.

The exam scheduling problem can be solved by translating it (reducing it) to the graph coloring problem and then solving the graph coloring problem.

For example, let $L_1 = \{A, B, D\}$, $L_2 = \{B, D, E\}$, $L_3 = \{A, C, F\}$ and suppose that $k = 2$. We need to construct an instance of the graph coloring problem $< G, k >$.

Let each list be represented by a node of G, and let two nodes be adjacent if they share a student. So the graph $G = (V, E)$ representing the lists above is given by $V = \{1, 2, 3\}, E = \{(1, 2), (1, 3)\}$).

The exam scheduling problem can be solved by translating it (reducing it) to the graph coloring problem and then solving the graph coloring problem.

For example, let $L_1 = \{A, B, D\}$, $L_2 = \{B, D, E\}$, $L_3 = \{A, C, F\}$ and suppose that $k = 2$. We need to construct an instance of the graph coloring problem $< G, k >$.

Let each list be represented by a node of G, and let two nodes be adjacent if they share a student. So the graph $G = (V, E)$ representing the lists above is given by $V = \{1, 2, 3\}, E = \{(1, 2), (1, 3)\})$.

The crucial point is that k exam periods are sufficient if and only if the graph G we constructed is k-colorable.

Example: reduction

The exam scheduling problem can be solved by translating it (reducing it) to the graph coloring problem and then solving the graph coloring problem.

For example, let $L_1 = \{A, B, D\}$, $L_2 = \{B, D, E\}$, $L_3 = \{A, C, F\}$ and suppose that $k = 2$. We need to construct an instance of the graph coloring problem $< G, k >$.

Let each list be represented by a node of G, and let two nodes be adjacent if they share a student. So the graph $G = (V, E)$ representing the lists above is given by $V = \{1, 2, 3\}, E = \{(1, 2), (1, 3)\})$.

The crucial point is that k exam periods are sufficient if and only if the graph G we constructed is k-colorable. Since G is two colorable, two exam periods suffice.

The "translation" between problems is called a reduction.

- The "translation" between problems is called a reduction.
- Let T_1 and T_2 be decision problems.

The "translation" between problems is called a reduction.

Let T_1 and T_2 be decision problems.

T_1 is polynomial-time reducible to T_2, denoted $T_1 \leq_P T_2$

The "translation" between problems is called a reduction.

Let T_1 and T_2 be decision problems.

T_1 is polynomial-time reducible to T_2, denoted $T_1 \leq_P T_2$ if there is a polynomial-time algorithm A mapping instances of T_1 to instances of T_2 where $x \in T_1$ if and only if $A(x) \in T_2$.

The "translation" between problems is called a reduction.

Let T_1 and T_2 be decision problems.

T_1 is polynomial-time reducible to T_2, denoted $T_1 \leq_P T_2$ if there is a polynomial-time algorithm A mapping instances of T_1 to instances of T_2 where $x \in T_1$ if and only if $A(x) \in T_2$.

Note that instead of solving the problem T_1 on an instance x, we can, solve the problem T_2 on instance $A(x)$ and return that answer.

The "translation" between problems is called a reduction.

Let T_1 and T_2 be decision problems.

T_1 is polynomial-time reducible to T_2, denoted $T_1 \leq_P T_2$ if there is a polynomial-time algorithm A mapping instances of T_1 to instances of T_2 where $x \in T_1$ if and only if $A(x) \in T_2$.

Note that instead of solving the problem T_1 on an instance x, we can, solve the problem T_2 on instance $A(x)$ and return that answer.

Furthermore, if problem T_2 accepts a polynomial time algorithm, i.e. $T_2 \in P$, then $T_1 \in P$ as well.

Supppose that the following is true:

1. $T_1 \leq_P T$ using mapping algorithm A_1.

Supppose that the following is true:

1. $T_1 \leq_P T$ using mapping algorithm A_1.
2. $T_2 \leq_P T$ using mapping algorithm A_2.

Supppose that the following is true:

1. $T_1 \leq_P T$ using mapping algorithm A_1.
2. $T_2 \leq_P T$ using mapping algorithm A_2.
3. $T_3 \leq_P T$ using mapping algorithm A_3.

Supppose that the following is true:

1. $T_1 \leq_P T$ using mapping algorithm A_1.
2. $T_2 \leq_P T$ using mapping algorithm A_2.
3. $T_3 \leq_P T$ using mapping algorithm A_3.
4. $T_4 \leq_P T$ using mapping algorithm A_4.

Supppose that the following is true:

1. $T_1 \leq_P T$ using mapping algorithm A_1.
2. $T_2 \leq_P T$ using mapping algorithm A_2.
3. $T_3 \leq_P T$ using mapping algorithm A_3.
4. $T_4 \leq_P T$ using mapping algorithm A_4.

Then if we have an algorithm B for T, we can solve each one of $T_1, \ldots, T_4$ by translating their instances using the algorithms $A_1, \ldots, A_4$ and returning the answer that B gives on the translated instance.

Supppose that the following is true:

1. $T_1 \leq_P T$ using mapping algorithm A_1.
2. $T_2 \leq_P T$ using mapping algorithm A_2.
3. $T_3 \leq_P T$ using mapping algorithm A_3.
4. $T_4 \leq_P T$ using mapping algorithm A_4.

Then if we have an algorithm B for T, we can solve each one of $T_1, \ldots, T_4$ by translating their instances using the algorithms $A_1, \ldots, A_4$ and returning the answer that B gives on the translated instance.

Therefore, T is harder than T_1, T_2, T_3, T_4.

Supppose that the following is true:

1. $T_1 \leq_P T$ using mapping algorithm A_1.
2. $T_2 \leq_P T$ using mapping algorithm A_2.
3. $T_3 \leq_P T$ using mapping algorithm A_3.
4. $T_4 \leq_P T$ using mapping algorithm A_4.

Then if we have an algorithm B for T, we can solve each one of $T_1, \ldots, T_4$ by translating their instances using the algorithms $A_1, \ldots, A_4$ and returning the answer that B gives on the translated instance.

Therefore, T is harder than T_1, T_2, T_3, T_4.

We use this idea above to define NP-completeness.

Definition

A decision problem T is *NP*-complete if:

Definition

A decision problem T is NP-complete if:

1. $T \in NP$.

Definition

A decision problem T is NP-complete if:

1. $T \in NP$.
2. For all $T' \in NP$, $T' \leq_P T$.

- We are about to define our first NP-complete problem.

We are about to define our first NP-complete problem.

In order to do this, we define a literal to be a boolean variable or its negation, and a clause to be a literal or a disjunction ("OR") of literals. For example, $x, y, \neg x$ are literals, $(x \vee y), (\neg x \vee y \vee x)$ are both clauses, but $(x \wedge \neg y \vee z)$ is not.

We are about to define our first NP-complete problem.

In order to do this, we define a literal to be a boolean variable or its negation, and a clause to be a literal or a disjunction ("OR") of literals. For example, $x, y, \neg x$ are literals, $(x \vee y), (\neg x \vee y \vee x)$ are both clauses, but $(x \wedge \neg y \vee z)$ is not.

A formula in conjunctive normal form (CNF) is the conjunction ("AND") of a set of clauses.

We are about to define our first NP-complete problem.

In order to do this, we define a literal to be a boolean variable or its negation, and a clause to be a literal or a disjunction ("OR") of literals. For example, x, y, $\neg x$ are literals. $(x \vee y)$, $(\neg x \vee y \vee x)$ are both clauses, but $(x \wedge \neg y \vee z)$ is not.

A formula in conjunctive normal form (CNF) is the conjunction ("AND") of a set of clauses. For example,

$$(x \vee y) \wedge (\neg x \vee y \vee x) \wedge (\neg y \vee z)$$

is in CNF.

Let *CNF.SAT* be the problem defined by

$\{< f >: f$ is a formula in CNF with some satisfying assignment$\}$

Theorem (Cook, 1971)

CNF.SAT is NP-complete.

The proof is beyond the scope of this course.

Let us now consider the general satisfiability problem defined by:

$SAT = \{< f >: f$ is a Boolean formula with a satisfying assignment$\}$

Let us now consider the general satisfiability problem defined by:

$SAT = \{< f >: f$ is a Boolean formula with a satisfying assignment$\}$

Suppose that we want to show that SAT is NP-complete.

Let us now consider the general satisfiability problem defined by:

$SAT = \{< f >: f$ is a Boolean formula with a satisfying assignment$\}$

Suppose that we want to show that SAT is NP-complete. We need to show:

- $SAT \in NP$.

Let us now consider the general satisfiability problem defined by:

$SAT = \{< f >: f$ is a Boolean formula with a satisfying assignment$\}$

Suppose that we want to show that SAT is NP-complete. We need to show:

- $SAT \in NP$.
- For all $A \in NP$, $A \leq_P SAT$.

Proof.

The obvious witness is the a list y of truth values to assign to the variables of f. The verification algotithm A takes f and y as inputs and evaluates f at y. It clearly runs in time that is bounded by a polynomial function of the size of f. $\square$

To complete the proof that SAT is NP-complete, we must show that every problem A in NP is reducible to SAT.

To complete the proof that SAT is NP-complete, we must show that every problem A in NP is reducible to SAT. But there are thousands of known problems in NP, and many more unknown ones!

Proof: For all $A \in NP$, $A \leq_P SAT$

To complete the proof that SAT is NP-complete, we must show that every problem A in NP is reducible to SAT. But there are thousands of known problems in NP, and many more unknown ones! It would be a gigantic task to show that each one reduces to SAT.

To complete the proof that SAT is NP-complete, we must show that every problem A in NP is reducible to SAT. But there are thousands of known problems in NP, and many more unknown ones! It would be a gigantic task to show that each one reduces to SAT.

An easier way to show that every problem in NP reduces to SAT is to show that some NP-complete problem B reduces to SAT.

Proof: For all $A \in NP$, $A \leq_P SAT$

To complete the proof that SAT is NP-complete, we must show that every problem A in NP is reducible to SAT. But there are thousands of known problems in NP, and many more unknown ones! It would be a gigantic task to show that each one reduces to SAT.

An easier way to show that every problem in NP reduces to SAT is to show that some NP-complete problem B reduces to SAT. Then since every A in NP reduces to B, and since the relation $\leq_P$ is transitive, every A also reduces to SAT:

$$A \rightarrow^{T_1} B \rightarrow^{T_2} SAT$$

$$x \in A \iff T_2(T_1(x)) \in SAT$$

To complete the proof that SAT is NP-complete, we must show that every problem A in NP is reducible to SAT. But there are thousands of known problems in NP, and many more unknown ones! It would be a gigantic task to show that each one reduces to SAT.

An easier way to show that every problem in NP reduces to SAT is to show that some NP-complete problem B reduces to SAT. Then since every A in NP reduces to B, and since the relation $\leq_P$ is transitive, every A also reduces to SAT:

$$A \to^{T_1} B \to^{T_2} SAT$$

$$x \in A \iff T_2(T_1(x)) \in SAT$$

We will require that the reductions must be polynomial time.

Proof: For all $A \in NP$, $A \leq_P SAT$

To complete the proof that SAT is NP-complete, we must show that every problem A in NP is reducible to SAT. But there are thousands of known problems in NP, and many more unknown ones! It would be a gigantic task to show that each one reduces to SAT.

An easier way to show that every problem in NP reduces to SAT is to show that some NP-complete problem B reduces to SAT. Then since every A in NP reduces to B, and since the relation $\leq_P$ is transitive, every A also reduces to SAT:

$$A \to^{T_1} B \to^{T_2} SAT$$

$$x \in A \iff T_2(T_1(x)) \in SAT$$

We will require that the reductions must be polynomial time. We will show that $CNF.SAT \leq_P SAT$

Proof: For all $A \in NP$, $A \leq_P SAT$

To complete the proof that SAT is NP-complete, we must show that every problem A in NP is reducible to SAT. But there are thousands of known problems in NP, and many more unknown ones! It would be a gigantic task to show that each one reduces to SAT.

An easier way to show that every problem in NP reduces to SAT is to show that some NP-complete problem B reduces to SAT. Then since every A in NP reduces to B, and since the relation $\leq_P$ is transitive, every A also reduces to SAT:

$$A \to^{T_1} B \to^{T_2} SAT$$

$$x \in A \iff T_2(T_1(x)) \in SAT$$

We will require that the reductions must be polynomial time. We will show that $CNF.SAT \leq_P SAT$ and then,

To complete the proof that SAT is NP-complete, we must show that every problem A in NP is reducible to SAT. But there are thousands of known problems in NP, and many more unknown ones! It would be a gigantic task to show that each one reduces to SAT.

An easier way to show that every problem in NP reduces to SAT is to show that some NP-complete problem B reduces to SAT. Then since every A in NP reduces to B, and since the relation $\leq_P$ is transitive, every A also reduces to SAT:

$$A \rightarrow^{T_1} B \rightarrow^{T_2} SAT$$

$$x \in A \iff T_2(T_1(x)) \in SAT$$

We will require that the reductions must be polynomial time. We will show that $CNF.SAT \leq_P SAT$ and then, since every $A \in NP$ has the property that $A \leq_P CNF.SAT$,

To complete the proof that SAT is NP-complete, we must show that every problem A in NP is reducible to SAT. But there are thousands of known problems in NP, and many more unknown ones! It would be a gigantic task to show that each one reduces to SAT.

An easier way to show that every problem in NP reduces to SAT is to show that some NP-complete problem B reduces to SAT. Then since every A in NP reduces to B, and since the relation $\leq_P$ is transitive, every A also reduces to SAT:

$$A \rightarrow^{T_1} B \rightarrow^{T_2} SAT$$

$$x \in A \iff T_2(T_1(x)) \in SAT$$

We will require that the reductions must be polynomial time. We will show that $CNF.SAT \leq_P SAT$ and then, since every $A \in NP$ has the property that $A \leq_P CNF.SAT$, it must be the case that $A \leq_P SAT$.

Proof.

We start with the following observation: every formula f in CNF is a Boolean formula.

Proof.

We start with the following observation: every formula f in CNF is a Boolean formula.

So if we have an algorithm S for SAT then the following algorithm will solve $CNF.SAT$:

```
CNT.SAT(f)
  return S(f)
```

Proof: $CNF.SAT \leq_P SAT$

Proof.

We start with the following observation: every formula f in CNF is a Boolean formula.

So if we have an algorithm S for SAT then the following algorithm will solve $CNF.SAT$:

```
CNT.SAT(f)
  return S(f)
```


Thus the trivial transformation $T(f) = f$ works! It clearly runs in polynomial time. $\square$

To prove that a problem B is NP-complete, we must do the following:

1. Show that $B \in NP$.

To prove that a problem B is NP-complete, we must do the following:

1. Show that $B \in NP$. To do this we must:
    1. identify the witness y,

To prove that a problem B is NP-complete, we must do the following:

1. Show that $B \in NP$. To do this we must:
   1. identify the witness y,
   2. describe the verification algorithm

To prove that a problem B is NP-complete, we must do the following:

1. Show that $B \in NP$. To do this we must:
    1. identify the witness y,
    2. describe the verification algorithm
    3. argue that the verification algorithm runs in polynomial time.

To prove that a problem B is NP-complete, we must do the following:

1. Show that $B \in NP$. To do this we must:
   1. identify the witness y,
   2. describe the verification algorithm
   3. argue that the verification algorithm runs in polynomial time.
2. Reduce some NP-complete problem A to B.

To prove that a problem B is NP-complete, we must do the following:

1. Show that $B \in NP$. To do this we must:
   1. identify the witness y,
   2. describe the verification algorithm,
   3. argue that the verification algorithm runs in polynomial time.
2. Reduce some NP-complete problem A to B. To do this we must:
   1. find a suitable NP-complete problem A,

To prove that a problem B is NP-complete, we must do the following:

1. Show that $B \in NP$. To do this we must:
   1. identify the witness y,
   2. describe the verification algorithm
   3. argue that the verification algorithm runs in polynomial time.
2. Reduce some NP-complete problem A to B. To do this we must:
   1. find a suitable NP-complete problem A,
   2. describe the algorithm T that reduces A to B,

To prove that a problem B is NP-complete, we must do the following:

1. Show that $B \in NP$. To do this we must:
   1. identify the witness y,
   2. describe the verification algorithm,
   3. argue that the verification algorithm runs in polynomial time.
2. Reduce some NP-complete problem A to B. To do this we must:
   1. find a suitable NP-complete problem A,
   2. describe the algorithm T that reduces A to B,
   3. show that $x \in A$ iff $T(x) \in B$.

To prove that a problem B is NP-complete, we must do the following:

1. Show that $B \in NP$. To do this we must:
   1. identify the witness y,
   2. describe the verification algorithm,
   3. argue that the verification algorithm runs in polynomial time.
2. Reduce some NP-complete problem A to B. To do this we must:
   1. find a suitable NP-complete problem A,
   2. describe the algorithm T that reduces A to B,
   3. show that $x \in A$ iff $T(x) \in B$.
   4. argue that the reduction algorithm runs in polynomial time.

The End!