Time Complexity Classes

Chapter 28
The Language Class $P$

$L \in P$ iff

- there exists some deterministic Turing machine $M$ that decides $L$, and
- $\text{timereq}(M) \in \mathcal{O}(n^k)$ for some $k$.

We’ll say that $L$ is \textit{tractable} iff it is in $P$. 
Closure under Complement

Theorem: The class $P$ is closed under complement.

Proof:
Closure under Complement

**Theorem:** The class P is closed under complement.

**Proof:** If $M$ accepts $L$ in polynomial time, swap accepting and non accepting states to accept $\neg L$ in polynomial time.
Defining Complement

- CONNECTED = {<G> : G is an undirected graph and G is connected} is in P.

- NOTCONNECTED = {<G> : G is an undirected graph and G is not connected}.

- \(\neg\)CONNECTED = NOTCONNECTED \(\cup\) {strings that are not syntactically legal descriptions of undirected graphs}.

\(\neg\)CONNECTED is in P by the closure theorem. What about NOTCONNECTED?

If we can check for legal syntax in polynomial time, then we can consider the universe of strings whose syntax is legal. Then we can conclude that NOTCONNECTED is in P if CONNECTED is.
Languages That Are in P

- Every regular language.
- Every context-free language since there exist context-free parsing algorithms that run in $\mathcal{O}(n^3)$ time.
- Others:
  - $A^nB^nC^n$
  - Nim
To Show That a Language Is In P

- Describe a one-tape, deterministic Turing machine.
- It may use multiple tapes. Price:
- State an algorithm that runs on a conventional computer. Price:

How long does it take to compare two strings?

```
| a a a a ; a a a a ...
```

Bottom line: If ignoring polynomial factors, then just describe a deterministic algorithm.
Regular Languages

**Theorem:** Every regular language can be decided in linear time. So every regular language is in P.

**Proof:** If $L$ is regular, there exists some DFSM $M$ that decides it. Construct a deterministic TM $M'$ that simulates $M$, moving its read/write head one square to the right at each step. When $M'$ reads a $\square$, it halts. If it is in an accepting state, it accepts; otherwise it rejects.

On any input of length $n$, $M'$ will execute $n + 2$ steps.

So $\text{timereq}(M') \in O(n)$. 
Context-Free Languages

**Theorem:** Every context-free language can be decided in $O(n^{18})$ time. So every context-free language is in P.

**Proof:** The Cocke-Kasami-Younger (CKY) algorithm can parse any context-free language in time that is $O(n^3)$ if we count operations on a conventional computer. That algorithm can be simulated on a standard, one-tape Turing machine in $O(n^{18})$ steps.
Graph Languages

Represent a graph $G = (V, E)$ as a list of edges:

$101/1/11/11/10/10/100/100/101/11/101$
Graph Languages

CONNECTED =
{\langle G \rangle : G \text{ is an undirected graph and } G \text{ is connected}}.

Is CONNECTED in P?
**CONNECTED** is in P

\[ \text{connected}(<G = (V, E)>) = \]

1. Set all vertices to be unmarked.
3. Initialize \( L \) to \{1\}.
4. Initialize \( \text{marked-vertices-counter} \) to 1.
5. Until \( L \) is empty do:
   5.1. Remove the first element from \( L \). Call it \textit{current-vertex}.
   5.2. For each edge \( e \) that has \textit{current-vertex} as an endpoint do:
       Call the other endpoint of \( e \) \textit{next-vertex}. If \textit{next-vertex} is not already marked then do:
       Mark \textit{next-vertex}.
       Add \textit{next-vertex} to \( L \).
       Increment \( \text{marked-vertices-counter} \) by 1.
6. If \( \text{marked-vertices-counter} = |V| \) accept. Else reject.
Analyzing \textit{connected}

- Step 1 takes time that is $\mathcal{O}(|V|)$.
- Steps 2, 3, and 4 each take constant time.
- The loop of step 5 can be executed at most $|V|$ times.
  - Step 5.1 takes constant time.
  - Step 5.2 can be executed at most $|E|$ times. Each time, it requires at most $\mathcal{O}(|V|)$ time.
- Step 6 takes constant time.

So $\text{time}_{req}(\text{connected})$ is:

$$|V| \cdot \mathcal{O}(|E|) \cdot \mathcal{O}(|V|) = \mathcal{O}(|V|^2|E|).$$

But $|E| \leq |V|^2$. So $\text{time}_{req}(\text{connected})$ is:

$$\mathcal{O}(|V|^4).$$
Eulerian Paths and Circuits

Seven Bridges of Königsberg:
Eulerian Paths and Circuits

Seven Bridges of Königsberg:

As a graph:
EULERIAN-CIRCUIT

- An *Eulerian path* through a graph $G$ is a path that traverses each edge in $G$ exactly once.

- An *Eulerian circuit* through a graph $G$ is a path that starts at some vertex $s$, ends back in $s$, and traverses each edge in $G$ exactly once.

Bridge inspectors, road cleaners, and network analysts can minimize their effort if they traverse their systems by following an Eulerian path.
EULERIAN-CIRCUIT

Difference between:
- Eulerian circuit, and
- Hamiltonian circuit

EULERIAN-CIRCUIT = \{<G> : G is an undirected graph, and G contains an Eulerian circuit\}

Is EULERIAN-CIRCUIT in P?
Define the degree of a vertex to be the number of edges with it as an endpoint.
Define the **degree** of a vertex to be the number of edges with it as an endpoint.

Euler observed that:

- A connected graph possesses an Eulerian path that is not a circuit iff it contains exactly two vertices of odd degree. Those two vertices will serve as the first and last vertices of the path.
- A connected graph possess an Eulerian circuit iff all its vertices have even degree. Because each vertex has even degree, any path that enters it can also leave it without reusing an edge.
EULERIAN-CIRCUIT

Eulerian(<G: =(V, E)>) =
1. If connected(G) rejects, reject. Else:
   2. For each vertex v in G do:
      2.1. Count the number of edges that have v as one endpoint but not both.
      2.2. If the count is odd, exit the loop and reject.
   3. If all counts are even, accept.

- We have shown that connected runs in time that is polynomial in |<V>|.
- The loop in step 2 is executed at most |V| times. Each time through, it requires time that is $O(|E|)$.
- Step 3 takes constant time.

So the total time for steps 2 - 3 of Eulerian is $|V| \cdot O(|E|)$. But $|E| \leq |V|^2$. So the for steps 2-3 of Eulerian is $O(|V|^3)$. 
Spanning Trees

A **spanning tree** $T$ of a graph $G$ is a subset of the edges of $G$ such that:

- $T$ contains no cycles and
- Every vertex in $G$ is connected to every other vertex using just the edges in $T$.

An unconnected graph has no spanning trees.

A connected graph $G$ will have at least one spanning tree; it may have many.
Minimum Spanning Trees

A **weighted graph** is a graph that has a weight associated with each edge.

An **unweighted graph** is a graph that does not associate weights with its edges.

If $G$ is a weighted graph, the **cost** of a tree is the sum of the costs (weights) of its edges.

A tree $T$ is a **minimum spanning tree** of $G$ iff:
- it is a spanning tree and
- there is no other spanning tree whose cost is lower than that of $T$. 
Can we find an MST without searching all the possible trees?

The cheapest way to lay cable that connects a set of points is along a minimum spanning tree that connects those points.
Kruskal’s Algorithm

\textbf{Kruskal}(G : = (V, E) and connected) =

1. Sort the edges in \( E \) in ascending order by their cost. Break ties arbitrarily.
2. Initialize \( T \) to a forest with an empty set of edges.
3. Until all edges in \( E \) have been considered do:
   3.1. Select \( e \), the next edge in \( E \). If the endpoints of \( e \) are not connected in \( T \) then add \( e \) to \( T \).
4. Return \( T \).

Why does this work?
MST

MST = \{<G, cost> : G is an undirected graph with a positive cost attached to each of its edges and there exists a minimum spanning tree of G with total cost less than cost\}.

Is MST in P?
MST is in P

\[ \text{Kruskal}(G : = (V, E) \text{ and connected}) = \]

1. Sort the edges in \( E \) in ascending order by their cost. Break ties arbitrarily.
2. Initialize \( T \) to a forest with an empty set of edges.
3. Until all edges in \( E \) have been considered do:
   3.1. Select \( e \), the next edge in \( E \). If the endpoints of \( e \) are not connected in \( T \) then add \( e \) to \( T \).
4. Return \( T \).

- Step 1, takes \( |E| \cdot \log |E| \) comparisons and each comparison takes constant time.
- Step 2 takes constant time.
- The loop in step 3 takes \( O(|E| \cdot |V|) \) time.

So \( \text{timereq}(\text{Kruskal}) \in O(|E| \cdot |V|) \) or \( O(|<G>|^2) \).

With a more efficient implementation of step 3, it is possible to show that it is also \( O(|E| \cdot \log |V|) \).
Primality Testing

RELATIVELY-PRIME =
\{<n, m> : n and m are integers that are relatively prime\}.

PRIMES =
\{w : w is the binary encoding of a prime number\}

COMPOSITES =
\{w : w is the binary encoding of a nonprime number\}
But Finding Factors Remains Hard

http://xkcd.com/247/
Returning to TSP

TSP-DECIDE = \{<G, cost> : <G> encodes an undirected graph with a positive distance attached to each of its edges and G contains a Hamiltonian circuit whose total cost is less than <cost>\}.

An NDTM to decide TSP-DECIDE:
Returning to TSP

An NDTM to decide TSP-DECIDE:

1. For $i = 1$ to $|V|$ do:
   
   Choose a vertex that hasn’t yet been chosen.

2. Check that the path defined by the chosen sequence of vertices is a Hamiltonian circuit through $G$ with distance less than or equal to $cost$. 
TSP and Other Problems Like It

TSP-DECIDE, and other problems like it, share three properties:

1. The problem can be solved by searching through a space of partial solutions (such as routes). The size of this space grows exponentially with the size of the problem.

2. No better (i.e., not based on search) technique for finding an exact solution is known.

3. But, if a proposed solution were suddenly to appear, it could be checked for correctness very efficiently.
The Language Class NP

Nondeterministic deciding:

Let \( L \in \text{NP} \) iff:

- there is some NDTM \( M \) that decides \( L \), and
- \( \text{timereq}(M) \in O(n^k) \) for some \( k \).

NDTM deciders:

\[
\begin{array}{c}
s, \square \text{abab} \\
q_2, \# \text{abab} \\
q_1, \square \text{abab} \\
q_1, \square \text{abab} \\
q_3, \square \text{bbab}
\end{array}
\]
TSP Again

TSP-DECIDE = \{<G, cost> : <G> encodes an undirected graph with a positive distance attached to each of its edges and G contains a Hamiltonian circuit whose total cost is less than <cost>\}.

Suppose some Oracle presented a candidate path c:

\(<G, cost, v_1, v_7, v_4, v_3, v_8, v_5, v_2, v_6, v_1>\)

How long would it take to verify that c proves that:

\(<G, cost> is in TSP-DECIDE?\)
Deterministic Verifying

A Turing machine $V$ is a **verifier** for a language $L$ iff:

$$w \in L \text{ iff } \exists c \ (\langle w, c \rangle \in L(V)).$$

We’ll call $c$ a **certificate**.
An alternative definition for the class NP:

$L \in \text{NP}$ iff there exists a deterministic TM $V$ such that:

- $V$ is a verifier for $L$, and
- $\text{timereq}(V) \in O(n^k)$ for some $k$. 
ND Deciding and D Verifying

**Theorem:** These two definitions are equivalent:

1. \( L \in \text{NP} \) iff there exists a nondeterministic, polynomial-time TM that decides it.
2. \( L \in \text{NP} \) iff there exists a deterministic, polynomial-time verifier for it.

**Proof:** We must prove both directions of this claim:

1. Let \( L \) be in NP by (1):

2. Let \( L \) be in NP by (2):
Proving That a Language is in NP

- Exhibit an NDTM to decide it.
- Exhibit a DTM to verify it.
Example

- SAT = \{w : w is a Boolean wff and w is satisfiable\} is in NP.

\[
F_1 = P \land Q \land \neg R \\
F_2 = P \land Q \land R \\
F_3 = P \land \neg P \\
F_4 = P \land (Q \lor \neg R) \land \neg Q
\]

\[
SAT\text{-}decide(F_4) =
\]

\[
SAT\text{-}verify (<F_4, (P = True, Q = False, R = False)>) =
\]
3-SAT

- A literal is either a variable or a variable preceded by a single negation symbol.

- A clause is either a single literal or the disjunction of two or more literals.

- A wff is in conjunctive normal form (or CNF) iff it is either a single clause or the conjunction of two or more clauses.

- A wff is in 3-conjunctive normal form (or 3-CNF) iff it is in conjunctive normal form and each clause contains exactly three literals.
Every wff can be converted to an equivalent wff in CNF.

- 3-SAT = \{ \ w : w \ is \ a \ wff \ in \ Boolean \ logic, \\
   w \ is \ in \ 3\text{-}conjunctive \ normal \ form, \ and \\
   w \ is \ satisfiable \}. \\

Is 3-SAT in NP?
Other Languages That Are in NP

Graph-based languages:

- TSP-DECIDE.

- HAMILTONIAN-PATH = \{<G> : G is an undirected graph and G contains a Hamiltonian path}\}.

- HAMILTONIAN-CIRCUIT = \{<G, s> : G is an undirected graph and G contains a Hamiltonian circuit}\}.
CLIQUE

- CLIQUE = \{ <G, k> : G is an undirected graph with vertices \( V \) and edges \( E \), \( k \) is an integer, \( 1 \leq k \leq |V| \), and \( G \) contains a \( k \)-clique \}. 

A **clique** in \( G \) is a subset of \( V \) where every pair of vertices in the clique is connected by some edge in \( E \).

A **\( k \)-clique** is a clique that contains exactly \( k \) vertices.
CLIQUE

Course_1

Course_9

Course_8

Course_10

Student_2

Course_6

Course_7

Course_3

Course_4

Course_5
SUBGRAPH-ISOMORPHISM

• SUBGRAPH-ISOMORPHISM = \{<G_1, G_2> : G_1 \text{ is isomorphic to some subgraph of } G_2\}.

Two graphs $G$ and $H$ are *isomorphic* to each other iff there exists a way to rename the vertices of $G$ so that the result is equal to $H$. Another way to think about isomorphism is that two graphs are isomorphic iff their drawings are identical except for the labels on the vertices.
SUBGRAPH-ISOMORPHISM

PROPNALAL

PROPNAL
SHORTEST-SUPERSTRING

- SHORTEST-SUPERSTRING = \{<S, k> : S is a set of strings and there exists some superstring \( T \) such that every element of \( S \) is a substring of \( T \) and \( T \) has length less than or equal to \( k \}\).
SHORTEST-SUPERSTRING

Fig 2: Short fragments of DNA sequence are ordered by overlapping data to recreate the whole genome sequence

Source: Wiley: Interactive Concepts in Biology
SUBSET-SUM

- SUBSET-SUM = \{ <S, k> : S is a multiset of integers, k is an integer, and there exists some subset of S whose elements sum to k }.

- \(<\{1256, 45, 1256, 59, 34687, 8946, 17664\}, 35988\> \in SUBSET-SUM.
- \(<\{101, 789, 5783, 6666, 45789, 996\}, 29876\> \notin SUBSET-SUM.

To store password files: start with 1000 base integers. If each password can be converted to a multiset of base integers, a password checker need not store passwords. It can simply store the sum of the base integers that the password generates. When a user enters a password, it is converted to base integers and the sum is computed and checked against the stored sum. But if hackers steal an encoded password, they cannot use it unless they can solve SUBSET-SUM.
SET-PARTITION

- $\text{SET-PARTITION} = \{<S> : S \text{ is a multiset (i.e., duplicates are allowed) of objects each of which has an associated cost and there exists a way to divide } S \text{ into two subsets, } A \text{ and } S - A, \text{ such that the sum of the costs of the elements in } A \text{ equals the sum of the costs of the elements in } S - A\}.$

**SET-PARTITION** arises in many sorts of resource allocation contexts. For example, suppose that there are two production lines and a set of objects that need to be manufactured as quickly as possible. Let the objects’ costs be the time required to make them. Then the optimum schedule divides the work evenly across the two machines.
BIN-PACKING

• BIN-PACKING = \{<S, c, k> : S is a set of objects each of which has an associated size and it is possible to divide the objects so that they fit into k bins, each of which has size c}\.
BIN-PACKING

In two dimensions:

Source: mainlinemedia.com
BIN-PACKING

In three dimensions:
KNAPSACK

KNAPSACK = {<S, v, c> : S is a set of objects each of which has an associated cost and an associated value, v and c are integers, and there exists some way of choosing elements of S (duplicates allowed) such that the total cost of the chosen objects is at most c and their total value is at least v}. Notice that, if the cost of each item equals its value, then the KNAPSACK problem becomes the SUBSET-SUM problem.

How to pack a knapsack with limited capacity in such as way as to maximize the utility of the contents:
- A thief.
- A backpacker.
- Choosing ads for a campaign.
- What products should a company make?
MY HOBBY:
EMBEDDING NP-COMPLETE PROBLEMS IN RESTAURANT ORDERS

<table>
<thead>
<tr>
<th>APPETIZERS</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Mixed Fruit</td>
<td>2.15</td>
</tr>
<tr>
<td>French Fries</td>
<td>2.75</td>
</tr>
<tr>
<td>Side Salad</td>
<td>3.35</td>
</tr>
<tr>
<td>Hot Wings</td>
<td>3.55</td>
</tr>
<tr>
<td>Mozzarella Sticks</td>
<td>4.20</td>
</tr>
<tr>
<td>Sampler Plate</td>
<td>5.80</td>
</tr>
</tbody>
</table>

Sandwiches

Barbecue    6.55

We'd like exactly $15.05 worth of appetizers, please.

...exactly? Uhh...

Here, these papers on the Knapsack problem might help you out.

Listen, I have six other tables to get to—

As fast as possible, of course. Want something on Traveling Salesman?

http://xkcd.com/287/
The Relationship Between P and NP

Is $P = NP$?

Here are some things we know:

$P \subseteq NP \subseteq PSPACE \subseteq EXPTIME$

$P \neq EXPTIME$

*The Millenium Prize*
Using Reduction in Complexity Proofs

A *mapping reduction* $R$ from $L_1$ to $L_2$ is a

- Turing machine that
- implements some computable function $f$ with the property that:

$$\forall x \ (x \in L_1 \iff f(x) \in L_2).$$

If $L_1 \leq L_2$ and $M$ decides $L_2$, then:

$$C(x) = M(R(x)) \text{ will decide } L_1.$$
Using Reduction in Complexity Proofs

If $R$ is deterministic polynomial then:

$$L_1 \leq_P L_2.$$ 

And, whenever such an $R$ exists:

- $L_1$ must be in $P$ if $L_2$ is: if $L_2$ is in $P$ then there exists some deterministic, polynomial-time Turing machine $M$ that decides it. So $M(R(x))$ is also a deterministic, polynomial-time Turing machine and it decides $L_1$.

- $L_1$ must be in $NP$ if $L_2$ is: if $L_2$ is in $NP$ then there exists some nondeterministic, polynomial-time Turing machine $M$ that decides it. So $M(R(x))$ is also a nondeterministic, polynomial-time Turing machine and it decides $L_1$. 
Why Use Reduction?

Given $L_1 \leq_p L_2$, we can use reduction to:

- Prove that $L_1$ is in P or in NP because we *already know* that $L_2$ is.

- Prove that $L_1$ would be in P or in NP if we *could somehow show* that $L_2$ is. When we do this, we cluster languages of similar complexity (even if we’re not yet sure what that complexity is). In other words, $L_1$ is no harder than $L_2$ is.
INDEPENDENT-SET

- INDEPENDENT-SET = \{<G, k> : G is an undirected graph and G contains an independent set of at least k vertices\}.

An **independent set** is a set of vertices no two of which are adjacent (i.e., connected by a single edge). So, in the following graph, the circled vertices form an independent set:

In a scheduling program the vertices represent tasks and are connected by an edge if their corresponding tasks conflict. We can find the largest number of tasks that can be scheduled at the same time by finding the largest independent set in the task graph.
3-SAT and INDEPENDENT-SET

3-SAT $\leq_p$ INDEPENDENT-SET.

Strings in 3-SAT describe formulas that contain literals and clauses.

$$(P \lor Q \lor \neg R) \land (R \lor \neg S \lor Q)$$

Strings in INDEPENDENT-SET describe graphs that contain vertices and edges.

101/1/11/11/10/10/100/100/101/11/101
Gadgets

A **gadget** is a structure in the target language that mimics the role of a corresponding structure in the source language.

Example: $3$-SAT $\leq_p$ INDEPENDENT-SET.

\[(P \lor Q \lor \neg R) \land (R \lor \neg S \lor Q)\]

(approximately)

101/1/11/11/10/10/100/100/101/11/101

So we need:
- a gadget that looks like a graph but that mimics a literal, and
- a gadget that looks like a graph but that mimics a clause.
3-SAT ≤ₚ INDEPENDENT-SET

\[ R(<f: \text{Boolean formula with } k \text{ clauses}>) = \]
1. Build a graph \( G \) by doing the following:
   1.1. Create one vertex for each instance of each literal in \( f \).
   1.2. Create an edge between each pair of vertices for symbols in the same clause.
   1.3. Create an edge between each pair of vertices for complementary literals.
2. Return \(<G, k>\).

\((P \lor \neg Q \lor W) \land (\neg P \lor S \lor T)\):
R is Correct

Show: \( f \in 3\text{-SAT} \) iff \( R(\langle f \rangle) \in \text{INDEPENDENT-SET} \) by showing:

- \( f \in 3\text{-SAT} \rightarrow R(\langle f \rangle) \in \text{INDEPENDENT-SET} \)
- \( R(\langle f \rangle) \in \text{INDEPENDENT-SET} \rightarrow f \in 3\text{-SAT} \)
One Direction

\[ f \in 3\text{-SAT} \rightarrow R(<f>) \in \text{INDEPENDENT-SET}: \]
There is a satisfying assignment \( A \) to the symbols in \( f \).
So \( G \) contains an independent set \( S \) of size \( k \), built by:
1. From each clause gadget choose one literal that is made positive by \( A \).
2. Add the vertex corresponding to that literal to \( S \).

\( S \) will contain exactly \( k \) vertices and is an independent set:
- No two vertices come from the same clause so step 1.2 could not have created an edge between them.
- No two vertices correspond to complimentary literals so step 1.3 could not have created an edge between them.
The Other Direction

- \( R(<f>) \in \text{INDEPENDENT-SET} \).
- So the graph \( G \) that \( R \) builds contains an independent set \( S \) of size \( k \).
- We prove that there is some satisfying assignment \( A \) for \( f \).

No two vertices in \( S \) come from the same clause gadget. Since \( S \) contains at least \( k \) vertices, no two are from the same clause, and \( f \) contains \( k \) clauses, \( S \) must contain one vertex from each clause.

Build \( A \) as follows:
1. Assign \( True \) to each literal that corresponds to a vertex in \( S \).
2. Assign arbitrary values to all other literals.

Since each clause will contain at least one literal whose value is \( True \), the value of \( f \) will be \( True \).
Why Do Reduction?

Would we ever choose to solve 3-SAT by reducing it to INDEPENDENT-SET?