This version `Tue Sep 13 15:17:01 PDT 2022`

# Reductions

Throughout, if we say "machine" we mean deterministic machine, unless we specifically say "non-deterministic."

Let $L_1$, $L_2$ be languages over the alphabets $\Sigma_1$ and $\Sigma_2$, respectively. A function $R : \Sigma_1^* \to \Sigma_2^*$ is a *reduction* of $L_1$ to $L_2$ if, for all $w \in \Sigma_1^*$, $w \in L_1$ if and only if $R(w) \in L_2$.
A reduction $R$ is a function, which may not be recursive (computable). However, any reduction that we might use is recursive. Any reduction that we use in the theory of $\mathcal{NP}$-complete languages is not only recursive, but is also computable in polynomial time.

**Instances.** We frequently discuss the complexity of a problem, which we can interpret as a language. But we usually have a concept of "instances" of a problem. For example, for the Boolean satisfiability problem (SAT), an instance is a Boolean expression, rather than an arbitrary string of symbols. $L_{\text{bool}} \subseteq \Sigma^*$ is the language of all Boolean expressions, where $\Sigma$ is an appropriate alphabet. Since $L_{\text{bool}}$ is context-free, it is $\mathcal{P}$. Later, we give a reduction of SAT to 3-SAT. The reduction we give is not defined on all of $\Sigma^*$, but only on $L_{\text{bool}}$. We justify this by observing that it is easy to recognize whether a string is a member of $L_{\text{bool}}$.
If $R : L_{\text{bool}} \to \Sigma^*$ has the property that $e \in$ SAT if and only if $R(e) \in$ 3SAT, we extend $R$ by defining it to be some default string, such as the empty string, for all members of $\Sigma^*$ which are not in $L_{\text{bool}}$. Then $R$ is a reduction in the original sense.
The reductions used in most practical applications, including those in this document, satisfy an additional condition, namely that they are functions of instances of one problem to instances of another. For example, $L_{\text{3cnf}}$ is the set of all Boolean expressions in 3-CNF form, while $L_{\text{3sat}} \subseteq L_{\text{3cnf}}$ is the language of all satisfiable Boolean expressions in 3-CNF form. To reduce SAT to 3SAT, we simply give a function $R : L_{\text{bool}} \to L_{\text{3cnf}}$ such that $R(e) \in L_{\text{3sat}}$ if and only if $e \in L_{\text{sat}}$.
Henceforth, a reduction of a problem will only be defined on instances of the problem. By the logic given above, that is sufficient if the set of instances of the problem is in $\mathcal{P}$, which is usually true in practice.
A language $L$ is in the class $\mathcal{P}$–TIME (or simply $\mathcal{P}$) if there is some constant $k$ and some machine $M$ (A machine of any class: Turing machine, RAM, virtual C++ machine, or whatever.) which decides whether a given string $w$ is a member of $L$, in at most $n^k$ steps, where $n = |w|$.
A language $L$ is in the class $\mathcal{NP}$–TIME (or simply $\mathcal{NP}$) if there is some constant $k$ and some non-deterministic machine $M$ such that, given any string $w \in L$, there is a computation of $M$, with input $w$, which halts in at most $n^k$ steps, where $n = |w|$, and such that, for any string $w \notin L$, there is no computation of $M$ with input $w$ which halts. Note that, since $M$ is non-deterministic, there could be lots of computations of $M$ with input $w$ that do not halt, even if $w \in L$.

### Verifier Definition of $\mathcal{NP}$

There is an equivalent definition of $\mathcal{NP}$ which is much easier to work with, the *verifier* definition, given below.
Let $L \subseteq \Sigma^*$ be a language. Then $L$ is $\mathcal{NP}$ if there is an integer $k$ and a program $V$ (called the verifier) such that

1. The input of $V$ is an ordered pair of strings $(w, c)$, and the output is the single symbol "0" or "1."

2. Let $w \in \Sigma^*$, and let $n = |w|$. If $w \in L$, there is a string $c$ called a *certificate* of $w$ such that $V$ halts with input $(w, c)$ and outputs "1" within $n^k$ steps.

3. For any string $w \notin L$, the output of $V$ with input $(w, c)$ is "0." regardless of $c$.

**Definition 1** We say that a language $L_1$ is $\mathcal{NP}$*-complete* if the following two conditions are satisfied:

1. $L_1 \in \mathcal{NP}$

2. For any $L_2 \in \mathcal{NP}$, there exists a polynomial time reduction of $L_2$ to $L_1$.

**Theorem 1** *If $L_1$ is $\mathcal{NP}$-complete, $L_2 \in \mathcal{NP}$, and there is a polynomial time reduction of $L_1$ to $L_2$, then $L_2$ is $\mathcal{NP}$-complete.*

*Proof:* Condition 1 of the definition is given. To prove Condition 2, let $L_3 \in \mathcal{NP}$; We need to show that there is a polynomial time reduction $R$ of $L_3$ to $L_2$. Since $L_1$ is $\mathcal{NP}$-complete, there is a polynomial time reduction of $L_3$ to $L_1$, and we are given a polynomial time reduction of $L_1$ to $L_2$. Let $R$ be the composition of those two reductions. $\square$

**Theorem 2** *If $L_1 \in \mathcal{P}$ and $L_1$ is $\mathcal{NP}$-complete, then $\mathcal{P} = \mathcal{NP}$.*

*Proof:* Trivially, $\mathcal{P} \subseteq \mathcal{NP}$. We need only show that any $L_2 \in \mathcal{NP}$ is in $\mathcal{P}$.
Let $L_2 \in \mathcal{NP}$. Since $L_1 \in \mathcal{P}$, there is a machine $M_1$ that decides $L$ in polynomial time. Since $L_1$ is $\mathcal{NP}$-complete, there is a polynomial time reduction of $L_2$ to $L_1$, computed by some machine $M_2$. Connecting the input of $M_1$ with the output of $M_2$, we obtain a machine that decides $L_2$ in polynomial time. Thus, $L_2 \in \mathcal{P}$. $\square$

## Our List of $\mathcal{NP}$-Complete Problems

We use Theorem 1 to "bootstrap" the list of $\mathcal{NP}$-complete problems we discuss in this course.

- We have to start somewhere. SAT is proved to be $\mathcal{NP}$-complete from first principles, without the use of Theorem 1. You can find the proof on various websites, but I won't give it in class. From then on, we use the Theorem.

- There is a reduction of SAT to 3SAT. If $E$ is a Boolean expression, construct a parse tree for $E$ using a context-free grammar for $L_{\text{bool}}$. Then replace each internal node of that parse tree by a Boolean expression in 3CNF form according to certain rules.[1] The conjunction of those expressions is a member of $L_{cnft}$ which is satifiable if and only if $E$ is satisfiable.
- We give a reduction of 3SAT to IND, the independent set problem, proving that IND is $\mathcal{NP}$-complete.
- We give a reduction of IND to the subset sum problem, proving that the subset sum problem is $\mathcal{NP}$-complete.
- We give a reduction of the subset sum problem to the partition problem, proving that the partition problem is $\mathcal{NP}$-complete.

You can find massive lists of $\mathcal{NP}$-complete problems on the internet. New ones are found frequently.

## Boolean Satisfiability

Generally, we define a *Boolean expression* to be an expression involving variables and operators, where all variables have Boolean type and all operators have Boolean type.[2] A *satisfying assignment* of a Boolean expression is an assignment of truth values (there are only two truth values, *true* and *false*) to each variable so that the value of the expression is *true*. If a Boolean expression has a satisfying assignment, we say it is *satisfiable*; otherwise, we say it is a *contradiction.* The *Boolean satisfiability problem* is whether a given boolean expression is satisfiable. This problem is $\mathcal{NP}$-complete, and in fact is our "base" $\mathcal{NP}$-complete problem, the one we shall use to determine that other problems are $\mathcal{NP}$-complete.

### Formal Definition of Boolean Satisfiability

We first define a context-free language $L_{\text{bool}}$ to consist of all boolean expressions using the following rules:

1. All variables are written in alphanumeric style, *i.e.*, are strings containing letters and digits and starting with a letter.
2. The only operators permitted are
   (a) "+" denoting *or.*
   (b) "·" denoting *and.*
   (c) "!" denoting *not.*

[1]Not given here.

[2]This definition is much more restrictive than the definition of a Boolean expression in a programming language, which could contain other things, such as the C++ expression "`n == 5`" where `n` is a variable of integer type.

(d) "=" denoting *if and only if.*

(e) "⇒" denoting *implies.*

We apply the following precedence rules:

(a) "!" has precedence over all other operators.

(b) "·" has precedence over all operators except "!"

(c) "+" has precedence over all operators except "!" and "·"

(d) "⇒" has precedence over "="

(e) All binary operators are left-associative.

It is an easy exercise to design a context-free grammar for $L_{\text{bool}}$, using the alphabet consisting of the operator symbols listed above, letters and digits, and right and left parentheses.

**Certificates for $L_{\text{sat}}$** A certificate for any $w \in L_{\text{sat}}$ is a satisfying assignment. For example, $x = 1, y = 0$ is a certificate for the boolean expression $(x + y) \cdot (!x+!y)$, while the boolean expression $(x{\cdot}!x)$ has no certificate. Since a certificate for any $w \in L$ is no longer than $w$ itself, and since it can be verified in linear time, we have an $\mathcal{NP}$ certificate system for $L_{\text{sat}}$.

**Conjunctive normal form.** We say that $w \in L_{\text{bool}}$ is in *conjunctive normal form* (abbreviated CNF) if it is the conjunction of clauses, where each clause is the disjunction of terms, and each term is either a variable or the negation of a variable.

**Definition 2**

1. *$L_{\text{sat}}$ is the subset of $L_{\text{bool}}$ consisting of all satisfiable boolean expressions.*

2. *$L_{\text{3cnf}}$ is the subset of $L_{\text{bool}}$ consisting of all boolean expressions in conjunctive normal form where each clause has exactly three terms.*

3. *$L_{\text{3sat}} = L_{\text{3cnf}} \cap L_{\text{sat}}$.*

We give context-free grammars for both $L_{\text{bool}}$ and $L_{\text{3cnf}}$.

**Unambiguous Context-free grammar for $L_{\text{bool}}$**

The start symbol is $S$.
$S \to E = E \mid E \Rightarrow E \mid E$
$E \to E + T \mid T$
$T \to T \cdot F \mid F$
$F \to\ !F \mid V \mid (S)$
$V \to AP$
$P \to AP \mid NP \mid \epsilon$
$A \to a \mid b \mid c \mid d \mid e \mid f \mid g \mid h \mid i \mid j \mid k \mid l \mid m \mid n \mid o \mid p \mid q \mid r \mid s \mid t \mid u \mid v \mid w \mid x \mid y \mid z$
$N \to 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

**Ambiguous Context-free grammar for $L_{\text{bool}}$**

It is more practical to give an ambiguous grammar for $L_{\text{bool}}$, since parse trees are simpler. The start symbol of this grammar is $E$, for "expression." For simplicity, we use the symbol **id** to stand for any identifier.
$E \to E + E$
$E \to E \cdot E$
$E \to !E$
$E \to E \Rightarrow E$
$E \to E = E$
$E \to (E)$
$E \to \mathbf{id}$

**Unambiguous Context-free grammar for $L_{\text{3cnf}}$**

The start symbol is $E$.
$E \to E \cdot C \mid C$
$C \to (T + T + T)$
$T \to !V \mid V$
$V \to AP$
$P \to AP \mid NP \mid \epsilon$
$A \to a \mid b \mid c \mid d \mid e \mid f \mid g \mid h \mid i \mid j \mid k \mid l \mid m \mid n \mid o \mid p \mid q \mid r \mid s \mid t \mid u \mid v \mid w \mid x \mid y \mid z$
$N \to 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

**Theorem 3** *$L_{\text{sat}}$ is $\mathcal{NP}$-complete.*

The proof of Theorem 3 is long, but straightforward. I do not believe it to be a good idea to present it to the class, but you can read it in a number of places.

**Theorem 4** *$L_{\text{3sat}}$ is $\mathcal{NP}$-complete.*

Theorem 4 is proven by giving a polynomial time reduction of $L_{\text{sat}}$ to $L_{\text{3sat}}$. We will not give the details of this reduction, but we discuss some of the issues.
Two Boolean expressions are usually said to be equal if they have the same variables and every assignment of those variables satisfies either both expressions or neither expression. For example, the expression $x \cdot (y + z)$ is equal to the expression $x \cdot y + x \cdot z$. It is impossible to find a reduction $R$ of $L_{\text{sat}}$ to $L_{\text{3sat}}$ which maps every boolean expression of $L_{\text{bool}}$ to an equal Boolean expression in $L_{\text{3cnf}}$. Our reduction must make use of new variables.
Clauses are not independent. For example, let $E$ be the expression in 2-CNF form:

$$(x + y) \cdot (x + !y) \cdot (!x + y) \cdot (!x + !y)$$

Each of the four clauses is satisfiable. In fact, the conjunction of any three of those clauses is satisfiable. But $E$, the conjunction of all four clauses, is not satisfiable.
Let $E_1$ be the Boolean expression $(x + y + z + w) \cdot F$, where $F$ is satisfiable. Clearly $x + y + z + w$ is satisfiable, but $E_1$ may not be. For example, $x$, $y$, $z$, and $w$ could all appear somewhere in $F$, and it could be that the every satisfying assignment of $F$ assigns *false* to

all four of those variables. We must find a Boolean expression $E_2$ which is equivalent to $E_1$ in a weaker way, namely that $E_2$ is satisfiable if and only if $E_1$ is satisfiable.
We use the a new variable which must have a name that does not appear in $F$. Let $u$ be that new variable. Now let $E_2$ be the Boolean expression $(x + y + u) \cdot (!u + z + w) \cdot F$. If $F$ is 3CNF, then $E_2$ is 3-CNF.
We see that $E_1$ and $E_2$ are not equal, because $E_2$ has more variables that $E_1$, but they are equivalent in the weaker sense defined above: suppose $I_1$ is satisfiable. Then there is an assignment of the variables of $E_1$ such that at least one of the four variables $x$, $y$, $z$, and $w$ is assigned *true*. The same assignment, augmented by an assignment of $u$, satisfies $E_2$. To prove this, we consider two cases.
Case I: Either $x$ or $y$ is assigned true. Then assign $u$ the value *false*. The clause $(x+y+u)$ is true because $x$ is true, while the clause $(!u + z + w)$ is true because $u$ is false. The expression $F$ is still true, since the assignments of its variables have not changed.
Case II: Either $z$ or $w$ is assigned true. Then assign $u$ true, making the first clause true. The second clause is also true, and $F$ is still true as before.
On the other hand, suppose $E_1$ is a contradiction. Then $x$, $y$, $z$, and $w$ must all be assigned false. Whatever the assignment of $u$, either the first or the second clause must be false, hence $E_2$ is a contradiction.

## Other $\mathcal{NP}$-Complete Problems

The most common method of proving that a given problem (*i.e.*, language) is $\mathcal{NP}$-complete is to use Theorem 1, where $L_1$ is taken to be a problem (*i.e.*, language) already known to be $\mathcal{NP}$-complete. The problem $L_{3\text{sat}}$ is one of the more popular choices used for this purpose.[3]

### The Independent Set Problem

Given a graph $G$ and a number $k$, an *independent set* of $G$ is defined to be a set $\mathcal{I}$ of vertices of $G$ such that no two members of $\mathcal{I}$ are connected by an edge of $G$. The *order* of $\mathcal{I}$ is defined to be its size, *i.e..*, simply how many vertices it contains.
An instance of the *independent set problem* is $\langle G\rangle\langle k\rangle$, where $G$ is a graph and $k$ is an integer. The question is, "Does $G$ have an independent set of order $k$?"

**The language $L_{\text{ind}}$.** We define $L_{\text{instance-ind}} = \{\langle G\rangle k\}$ to be the instances of the independent set problem, and $L_{\text{ind}} \subseteq L_{\text{instance-ind}}$ the set of instances for which the graph $G$ has an independent set $k$ vertices.

**Theorem 5** *$L_{\text{ind}}$ is $\mathcal{NP}$ complete.*

*Proof:* For any $w \in L_{3\text{cnf}}$, we construct $R(w) \in L_{\text{instance-ind}}$ such that $w \in L_{3\text{sat}}$ if and only if $R(w) \in L_{\text{ind}}$. Thus $R$ is a reduction of 3SAT to IND.
Let $e \in L_{3\text{cnf}}$. Then $e = C_1 \cdot C_2 \cdot \cdots \cdot C_k$, where $C_i = (t_{i,1} + t_{i,2} + t_{i,3})$, where each $t_{i,j}$ is either $x$ or $!x$, where $x$ is a string which represents a variable.

[3] The precise definition of the problem described in this handout as $L_{3\text{sat}}$ differs from book to book, but they are all equivalent.

We now define a graph $G[e] = (V, E)$, where $V = \{v_{i,j} : 1 \leq i \leq k, 1 \leq j \leq 3\}$ is the set of vertices of $G[e]$, and $E$ the set of edges of $G[e]$, as follows:

1. For each $1 \leq i \leq k$, there is an edge from $v_{i,j}$ to $v_{i,j'}$ for all $1 \leq j < j' \leq 3$. Call these *short* edges.
2. If $t_{i,j} = x$ and $t_{i',j'} = !x$ for some variable $x$, there is an edge from $v_{i,j}$ to $v_{i',j'}$. Call these *long* edges.
3. There are no other edges.

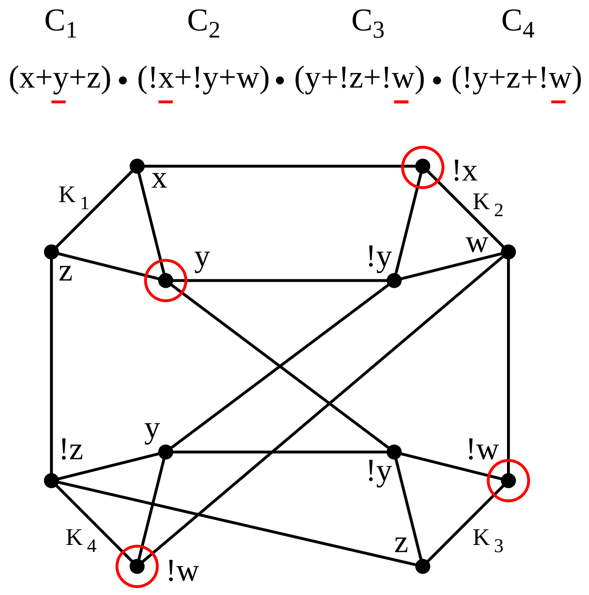
Let $R(e) = G[e], k)$ We now show that $R(e) \in L_{\text{ind}}$ if and only if $e$ is satisfiable. For each $i$, let $K_i$ be the subgraph of $G[e]$ consisting of the three vertices $v_{i,1}$, $v_{i,2}$, $v_{i,3}$, and the edges connecting them. We call this a *3-clique.*
Suppose $G[e], k \in L_{\text{ind}}$. Let $I \subset V$ be an independent set of of size $k$. Since $K_i$ is a 3-clique, and the number of such cliques is equal to $k$, exactly one member of $I$ must lie in each $K_i$. We define an assignment of $e$. If $v_{i,j} \in I$ and $t_{i,j} = x$ for some variable $x$, assign the value *true* to $x$, while if $t_{i,j} = !x$, assign *false* to $x$. Assign all remaining variables arbitrary Boolean values. This assignment is well-defined, for if $v_{i,j}, v_{i',j'} \in I$ for $i \neq i'$, there can be no edge between those two vertices, which implies that $t_{i,j}$ does not contradict $t_{i',j'}$. Furthermore, each clause has one term which is assigned true, hence each clause is assigned true, and we thus the assignment is satisfying.
Conversely, suppose that there is a satisfying assignment of $e$. That means each clause $C_i$ must contain one term, say $t_{i,j[i]}$ which is true under the assignment. Let $I = \left\{v_{i,j[i]}\right\} \subseteq V$. No two elements of $I$ are in the same clique $K_i$, hence there is no short edge connecting them, and there can be no long edge connecting them because $v_{i,j[i]}$ and $v_{i',j[i']}$ are both assigned true and hence cannot contradict each other. Thus $I$ is an independent set. $\square$

**Example**

A non-trivial example would have at least eight clauses, but I'll keep it simple. Let $E$ be the 3CNF expression

$$(x + y + z) \cdot (!x + !y + w) \cdot (y + !z + !w) \cdot (!y + z + !w)$$

Then $k = 4$. The following diagram illustrates $G[E]$. The vertices of $I$ are circled in red. The satisfying assignment shown is $x =$ false, $y =$ true, $w =$ false, while $z$ can be assigned either true or false.



## The Subset Sum Problem

Informally, the subset sum problem is whether there is a subset of a given set of items whose total is a given number. Formally, an instance of the subset sum problem is a finite sequence $x_1, \ldots x_k$ of non-negative numbers and a single number $B$. This instance is a member of the language $L_{\text{SS}}$ if there is some subsequence of $\{x_i\}$ whose sum is $B$.

**Theorem 6** *The subset sum problem is $\mathcal{NP}$-complete.*

*Proof:* Trivially, the subset sum problem satisfies the verifiability definition of $\mathcal{NP}$.
We reduce the independent set problem to the subset sum problem. Suppose $\langle G\rangle\langle k\rangle$ is an instance of the independent set problem, where $G$ has $n$ vertices and $m$ edges. Let $e_0, \ldots e_{m-1}$ be the edges of $G$ and $v_m \ldots v_{n+m-1}$ the vertices of $G$. Define $R(\langle G\rangle\langle k\rangle)$ to be the instance of the subset sum problem $w = (x_0, x_1, \ldots x_{n+m-1}, B)$ where

- $x_i = 4^i$ for $0 \le i < m$
- For $m \le i < n+m$, let $J_i = \{j : v_i \text{ is an endpoint of } e_j\}$, then $x_i = 4^m + \sum_{j \in J_i} 4^j$.
- $B = k4^m + \sum_{0 \le j < m} 4^j$

For $0 \le j < m$, $x_j$ corresponds to the edge $e_j$, while for $m \le i < m+n$, $x_i$ corresponds to the vertex $v_i$.
We need to prove $R$ is a reduction of IND to Subset Sum. Suppose $I \subseteq V$ is an independent set of $k$ vertices of $G$. Let

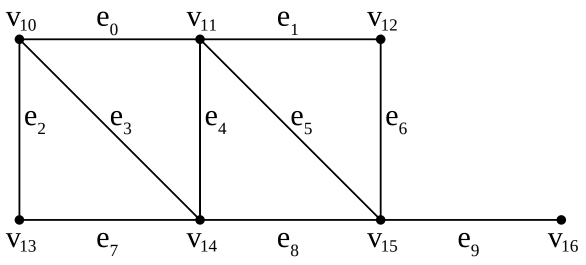
$$S = \{x_j : 0 \le j < m \text{ and no endpoint of } e_j \text{ is in } I\} \cup \{x_i : m \le i < n+m \text{ and } v_i \in I\}$$

We need to show that $\sum S$, the sum of the members of $S$, equals $B$. Since $|I| = k$, the coefficient of $4^m$ is $k$. An $e_j$ is adjacent to either just one member of $I$ or none. If $e_j$ is

adjacent to vertex $v_i \in I$, then the coefficient of $4^j$ in $x_i$ is 1, matching that of $B$. Otherwise, $S$ contains $x_j$, which is simply $4^j$, hence $\sum S = B$.
Conversely, suppose there is a subset $S$ of the sequence whose sum is $B$. Let $I = \{v_i : x_i \in S\}$. Since the coefficient of $4^m$ in $B$ is $k$, $I$ must contain exactly $k$ vertices. Since the coefficient of every $x_j$ in $B$ is 1, no two members of the $I$ can be adjacent, hence $I$ is a solution to the subset sum problem. $\square$

## Example

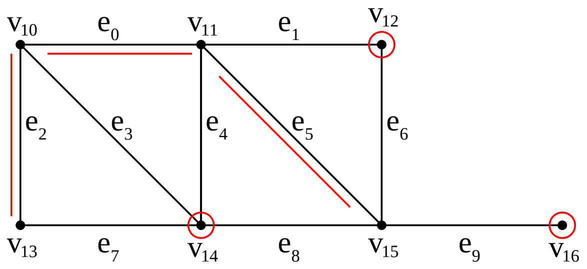
Let $G$ be the graph shown below, and let $k = 3$.



We show the reduced instance of the subset sum problem, where the $x_i$ are written in base 4.

$$\begin{aligned}
x_0 &= 1\\
x_1 &= 10\\
x_2 &= 100\\
x_3 &= 1000\\
x_4 &= 10000\\
x_5 &= 100000\\
x_6 &= 1000000\\
x_7 &= 10000000\\
x_8 &= 100000000\\
x_9 &= 1000000000\\
x_{10} &= 10000001101\\
x_{11} &= 10000110011\\
x_{12} &= 10001000010\\
x_{13} &= 10010000100\\
x_{14} &= 10110011000\\
x_{15} &= 11101100000\\
x_{16} &= 11000000000\\
B &= 31111111111
\end{aligned}$$

The graph below shows an independent set $I$ of vertices of order 3, together with the set of edges which are not adjacent to members of $I$.



Finally, we show the members of the subsequence corresponding to the chosen vertices and edges.

$$\begin{array}{rcr}
x_0 & = & 1 \\
x_2 & = & 100 \\
x_5 & = & 100000 \\
x_{12} & = & 10001000010 \\
x_{14} & = & 10110011000 \\
x_{16} & = & 11000000000 \\
\hline
B & = & 31111111111
\end{array}$$

## Subset Sum and Partition

The subset sum problem can be shown to be $\mathcal{NP}$-complete by reducing IND to subset sum. We can then show that the partition problem is $\mathcal{NP}$-complete by reducing Subset Sum to Partition.
Recall that an instance of the Subset Sum problem is a number followed by a sequence of positive numbers, followed by one number, $K$, that is, $(x_1, x_2, \dots x_m, K)$, and that instance is in $L_{\text{subs}}$ if there is some subsequence of $x_1, \dots x_m$ whose total is $K$. Let $L_{\text{subs}}$ be the language of all instances of the problem which have a solution.
Similarly, an instance of the Partition problem is a sequence of positive numbers, namely $\langle y_1, \dots y_\ell \rangle$, and there is a solution to that instance if and only if there is some subsequence of $y_1, \dots y_\ell$ whose sum is half the total, *i.e.* $\frac{1}{2}\sum_{j=1}^{\ell} y_j$. Let $L_{\text{part}}$ be the language of all instances of the problem which have a solution.
The partition problem is trivially in the class $\mathcal{NP}$, since the solution, if any, can be verified in polynomial time.

### Reduction

Given an instance $I_K = (x_1, x_2, \dots x_m, K)$ of Subset Sum, let $A = \sum_{i=1}^{m} x_i$. Let $R(I_K) = I_P$ be the following instance of Partition:

$$I_P = (x_1, \dots x_m, K + 1, A - K + 1)$$

That is, $I_P = (y_1, \dots y_\ell)$ where $\ell = m + 2$, $y_i = x_i$ for $i \le m$, $y_{m+1} = K + 1$, and $y_{m+2} = A - K + 1$.
We need to prove that this reduction works, that is, that $I_P \in L_{\text{part}}$ if and only if $I_K \in L_{\text{subs}}$. There are thus two directions to the proof.

Note that the sum of the items of $I_P$ is $2A+2$, and half of that is $A+1$.
Suppose that $I_K \in L_{\text{subs}}$. Then there is a subsequence of $\{x_i\}$ whose total is $K$. The items of this subsequence, together with $A-K+1$, total $A+1$, and thus $I_P \in L_{\text{part}}$.
Conversely, suppose some subsequence $S$ of $K+1, x_1, \ldots x_m, L+1$ has total $A+1$. That subsequence cannot contain both $K+1$ and $A-K+1$, since their total exceeds A+1. Similarly, and by symmetry, $S$ must contain either $K+1$ or $A-K+1$. Without loss of generality, $S$ contains $A-K+1$. The remaining members of $S$ constitute a subsequence of $x_1, \ldots x_m$ whose total is $K$, and we are done.
Conversely, suppose there is a subset $B$ of $\{x_i\}$ whose total is $K$. Then $B \cup \{A-K+1\}$ is a subset of the sequence $\{y_j\}$ whose total is $A+1$, and we are done.