CSE 201
Analysis of Algorithms
NP Completeness

A problem is generally considered *tractable* if there exists an algorithm to solve it whose runtime is in the class $O(n^k)$ for some constant *k*. Such problems are said to be solvable in *polynomial time*. This has been the case for all of the problems we've discussed so far, and the class of polynomial time solvable problems is referred to as *P*. Some problems are not solvable, no matter how much time is expended on them. Among these is the *Halting Problem*, (see the handout of the same name) discovered by Alan Turing. Problems that are solvable, but only in super-polynomial time (like $n^{\log(n)}$, 2^n, $n!$, n^n, … etc.) are usually called *intractable*. Another class of problems are those that can be *verified* in polynomial time. That is, given a problem instance, and a purported solution to that instance, a polynomial time algorithm exists that verifies the purported solution *is* a solution. These problems constitute the class *NP*.

Another interesting class of problems are the so-called *NP-Complete* problems. These problems belong to *NP* but are, in a sense that we will make precise, as hard or harder than any other problem in *NP*. The status of this class is unknown at this time. No polynomial time algorithm is known for any of them, and yet neither has it been proved that polynomial time algorithms solving them do not exist. This is the so called *P* vs. *NP* problem which has attracted much attention in theoretical computer science since it was first formulated in the late 1960's and early 70's. We begin with two examples.

SHORTEST-PATH (SP)
Instance: a graph *G* and a pair of vertices $u, v \in V(G)$
Solution: the length of a shortest *u*-*v* path in *G*

HAMILTONIAN-CYCLE (HC)
Instance: a graph *G*
Solution: determine whether *G* contains a Hamiltonian cycle (a cycle *C* that includes all vertices of *G*.)

Several well-known polynomial time algorithms exist for SP, such as BFS and Dyjkstra's algorithm, and therefore SP is in the class *P*. No polynomial time algorithm is known that solves HC. However, HC is easily seen to be polynomial time verifiable, as follows. Given a graph *G* and a purported Hamiltonian cycle $C = (x_0, x_1, x_2, \ldots, x_n)$, where $x_0 = x_n$, one need only check that (1) $\{x_1, x_2, \ldots, x_n\} = V(G)$, and that (2) $\{x_{i-1}, x_i\} \in E(G)$ for $1 \leq i \leq n$. These checks can obviously be performed in polynomial time.

Many problems of interest are *Optimization Problems*, which seek an optimal solution from among a set of feasible solutions satisfying some fixed constraints. SP defined above is an optimization problem. The theory of NP-completeness applies only to *Decision Problems* however. These are problems, such as HC above, whose solution is a 2-valued answer like Yes or No. In a more abstract and formal setting, these decision problems are encoded as sets of bit strings, which are called *Languages*. We let $\{0,1\}^*$ denote the set of all bit strings, so a language *L* is simply a subset $L \subseteq \{0,1\}^*$. Thus we will discuss the following relationships.

$$\text{Optimization Problems} \underset{\text{conversion}}{\longleftrightarrow} \text{Decision Problems} \underset{\text{encoding}}{\longleftrightarrow} \text{Languages}$$

Every optimization problem has a corresponding decision problem, and conversely. For instance the optimization problem SP defined above corresponds to the following decision problem.

PATH
Instance: a graph G, a pair of vertices $u, v \in V(G)$ and a bound $k \in \mathbb{Z}^+$
Solution: determine whether G contains a u-v path of length at most k

Observe that any algorithm that solves SP can be converted to one that solves PATH, as follows. Given an instance (G, u, v, k) of PATH, use the algorithm for SP to compute the length l of a shortest u-v path in G. If $l \leq k$, then return Yes for PATH, otherwise return No. Furthermore, if the algorithm for SP runs in polynomial time, then so does the one for PATH, since the only additional step is the comparison $l \leq k$, a constant cost operation. This argument shows that the decision problem PATH can be no harder than the optimization problem SP, a situation we denote by writing: PATH $\leq$ SP.

This example can be generalized to any optimization problem (OP). Assume for definiteness that OP is a minimization problem. (Reverse the inequality below if it is a maximization problem.) We define a decision problem (DP) by taking any instance α of OP and adding a bound $b \in \mathbb{R}$. An instance of DP is then the pair (α, b). The question to be answered is: does there exist a solution to α with value $v(\alpha)$ at most b? (Here $v(\alpha)$ is the objective function for the optimization problem.) Any algorithm for OP can then be converted to one for DP just as above. Given an instance (α, b) of DP, run the algorithm for OP on α to obtain an optimum value $v(\alpha)$. Return Yes for (α, b) if $v(\alpha) \leq b$, and No otherwise. Observe that if the algorithm for OP runs in polynomial time, then so does the one for DP since the only additional work is the comparison $v(\alpha) \leq b$. Thus, just as PATH $\leq$ SP, we have DP $\leq$ OP, which says DP is no harder than OP. We can also reason contrapositively: if no polynomial time algorithm for DP exists, then none exists for OP.

It is possible to reverse this process and convert an algorithm for DP into one for OP as follows. Given an instance α of OP, run the algorithm for DP successively on the instances $(\alpha, 1), (\alpha, 2), (\alpha, 3)$, ... until the answer Yes is returned. If (α, m) is the first instance of DP for which Yes is returned, then return m as $v(\alpha)$, the minimum value of a feasible solution to OP. (This is assuming again that OP is a minimization problem.) Typically, the size n of α (whatever "size" may mean) will provide an upper bound on the number of iterations necessary to get a Yes. In that case, if the algorithm for DP runs in time $O(n^k)$, then the algorithm for OP runs in time $n \cdot O(n^k) = O(n^{k+1})$. We can do better still by replacing the "linear" search for m described above, by a binary search, the details of which are left to the reader. Thus we can also say that OP $\leq$ DP. If a polynomial time algorithm exists for DP, one exists for OP, and equivalently, if no polynomial time algorithm exists for OP, then none exists for DP.

Exercise
The above conversion of an algorithm for DP to one for OP is presented in the abstract. Carry out the details in a concrete case by writing pseudo-code for an algorithm that solves SP, by repeatedly calling a subroutine that solves PATH. Either do a linear search for the length of a shortest u-v path, or do a binary search. Let $T(n)$ denote the run time of the subroutine for PATH. Write the run time of your algorithm for SP in terms of $T(n)$.

The above discussion establishes that, provided we consider two problems to be equivalent when each is convertible to the other in polynomial time, we lose nothing by concentrating on decision problems. From now on, we do exactly that.

This same comparison technique can be used on decision problems. Suppose A and B are two decision problems, and suppose we have an algorithm that transforms any instance α of A into an instance β of B in such a way that the answer to α is YES if and only if the answer to β is YES. In other words, the

algorithm transforms YES instances of A to YES instances of B, and NO instances of A to NO instances of B. We call this a *reduction* from A to B. Given such a reduction, we can always convert an algorithm for B into one for A as follows.

(1) Use the reduction algorithm to transform an instance α of A into an instance β of B.
(2) Run the algorithm for B on the instance β, yielding an answer YES or NO.
(3) Return that answer (YES or NO) as the answer for the instance α of A.

We depict this process schematically below.

Decision Problem:	A		B
Reduction:	instance α	$\longrightarrow$	instance β
Conversion:	Algorithm for A	$\longleftarrow$	Algorithm for B

Observe that if the reduction algorithm runs in time $O(f(n))$, and the algorithm for B runs in time $O(g(n))$, then the resulting algorithm for A runs in time $O(f(n) + g(n))$. In particular, if the reduction runs in polynomial time, then any polynomial time algorithm for B can be converted into a polynomial time algorithm for A. When such a polynomial time reduction exists, we write $A \leq_P B$.

Example
Let the role of A be played by Hamiltonian-Cycle (HC), defined above, and let B be the following decision problem.

TRAVELING-SALESMAN-PROBLEM (TSP)
Instance: an integer n, a weight function on the edges of K_n ($w: E(K_n) \to \mathbb{R}^+$) and a bound $b > 0$.
Solution: determine whether there exists a sequence of vertices $(x_1, x_2, \ldots, x_n)$ in K_n such that

$$\sum_{i=1}^{n-1} w(x_i, x_{i+1}) + w(x_n, x_1) \leq b,$$

i.e. does there exist a Hamiltonian Cycle C in K_n whose total weight is no more than b: $w(C) \leq b$?

(Note the name *Traveling Salesman Problem* is also commonly used to refer to the optimization problem corresponding to this decision problem, namely determine a minimum weight Hamiltonian Cycle in K_n, given a weight function on its edges. As we've seen, these two versions of TSP are in fact polynomial time equivalent.)

We define a reduction from HC to TSP as follows. Given an instance $G = (V, E)$ of HC, let $n = |V|$ and identify $V = V(G)$ with $V(K_n)$. Define $w: E(K_n) \to \mathbb{R}^+$ by

$$w(u, v) = \begin{cases} 1 & \text{if } \{u, v\} \in E(G) \\ 2 & \text{if } \{u, v\} \notin E(G) \end{cases}$$

and let $b = n$. First, observe that the instance (n, w, b) of TSP can be computed from the instance G of HC in polynomial time. (For each of the $\binom{n}{2} = \frac{1}{2}n(n-1)$ edges of K_n, check to see if it belongs to G.) To establish that this transformation is a reduction, first assume that G is a YES instance for HC. Then G contains a Hamiltonian Cycle C. Since all n edges of C belong to both $E(G)$ and $E(K_n)$, they all have weight 1, and hence $w(C) = n = b$. If follows that (n, w, b) is a YES instance of TSP. Finally, assume that (n, w, b) is a YES instance of TSP. Then there exists a Hamiltonian Cycle C in K_n with total weight

$w(C) \leq b = n$. The only way to add up n terms, all of which are 1 or 2, and get a sum $\leq n$, is if all terms are equal to 1 (in which case the sum equals n). Thus the weights of all edges in C are 1, hence they belong to $E(G)$, and C is a Hamiltonian Cycle in G. Therefore G is a YES instance of HC. By Contraposition, this shows that NO instances of HC are mapped to NO instances of TSP.

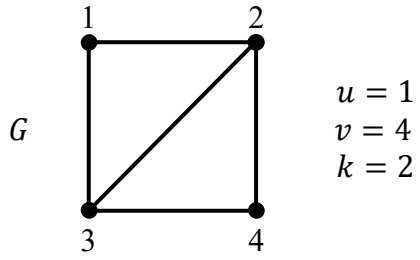
We have established that HC $\leq_P$ TSP, i.e. HC is no more difficult than TSP. Interestingly, no polynomial time algorithm is known for TSP. We could settle the matter for TSP by proving that no polynomial time algorithm exists for HC, perhaps by some adversary argument. Unfortunately, no one has been able to do that either. It is simply not known whether either of these decision problems are solvable in polynomial time, and it is strongly suspected that they are not.

Encodings

We must still formalize our notion of a decision problem. An *encoding* of a decision problem Q is a mapping e that transforms each instance α of Q into a bit string. We let $\{0,1\}^*$ denote the set of all bit strings, of any length. Thus e: {instances of Q} $\rightarrow \{0,1\}^*$.

Example

A common data structure used to encode a graph is the adjacency list representation, which is an array of lists, each storing the neighbors of a single vertex. The YES instance $(G, 1, 4, 2)$ of PATH



can be converted to text using the adjacency list representation

1: 2 3
2: 1 3 4
3: 1 2 4
4: 2 3
$u = 1$
$v = 4$
$k = 2$

then further encoded as a bit string using a suitable character set, such as ASCII, as below.

$e(G, 1, 4, 2) =$

00110001 00111010 00100000 00110010 00100000 00110011 00001010 00110010
00111010 00100000 00110001 00100000 00110011 00100000 00110100 00001010
00110011 00111010 00100000 00110001 00100000 00110010 00100000 00110011
00001010 00110100 00111010 00100000 00110010 00100000 00110011 00001010
01110101 00100000 00111101 00100000 00110001 00001010 01110110 00100000
00111101 00100000 00110100 00001010 01101011 00100000 00111101 00100000
00110010 00001010

Many other encodings are possible. Generally speaking, if one encoding can be transformed into another in polynomial time, then they are considered equivalent for our purposes. However, we must be careful not to use an inefficient encoding. For instance, an algorithm that operates on an integer n will appear to have a *better* run time when using an inefficient encoding.

$$\text{Unary:} \quad e_1(n) = \underset{\leftarrow}{111\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots} \underset{n}{} \underset{\rightarrow}{11}$$

$$\text{Binary:} \quad e_2(n) = \underset{\leftarrow \quad \lceil \lg(n+1) \rceil \quad \rightarrow}{1011\cdots\cdots\cdots\cdots\cdots 10}$$

Consider a simple operation like incrementing n by 1. In unary we would concatenate 1 to the end of the unary string $e_1(n)$, at cost $\Theta(1)$, which makes the problem look artificially easy. In a more efficient encoding, like binary, we perform approximately $\lg(n)$ bit operations on the bit string $e_2(n)$, at cost $\Theta(\log(n))$. The lesson of this example is that when analyzing any numerical algorithm, i.e. one that takes a number n as input, we should never consider the "size" of the input to be the number n itself, since we would then be measuring its size in unary. Instead define the *size* of n to be the number of bits needed to store it, namely $\lceil \lg(n+1) \rceil = \Theta(\log(n))$.

For instance, our Dynamic Programming solution to the general Coin Changing problem, gave us a run time in $\Theta(n \cdot N)$, where n was the size of the coin system and N was the amount to be paid. This run time appears to be polynomial, but not when thought of as a function of input size: $\Theta(nN) = \Theta\left(n \cdot 2^{\text{size}(N)}\right)$, which is actually exponential.

Definition
We say that two encodings e_1 and e_2 of a decision problem Q are *polynomially related* if and only if there exist functions $f_{12}: \{0,1\}^* \to \{0,1\}^*$ and $f_{21}: \{0,1\}^* \to \{0,1\}^*$, both computable in polynomial time, such that $f_{12}\big(e_1(\alpha)\big) = e_2(\alpha)$ and $f_{21}\big(e_2(\alpha)\big) = e_1(\alpha)$ for all instances α of Q.

We assume now that every decision problem comes with a class of polynomial-time equivalent "good" encodings. For instance, PATH can be encoded using the adjacency list representation, or using the adjacency matrix representation (see graph theory handout for definitions). Both are good encodings, and one representation can be converted to the other in polynomial time. We leave the verification of this as an exercise for the reader.

Definition
When a decision problem Q is transformed, via some encoding, into a collection of bit strings, each string representing one instance of Q, we call it a concrete problem. More precisely, a *concrete problem* is the image of the mapping $e: \{\text{instances of Q}\} \to \{0,1\}^*$, a set of bit strings.

Using such an encoding, we can define precisely what is meant by the size n of an instance α of Q.

$$n = |e(\alpha)| = \text{number of bits in } e(\alpha)$$

We agree to measure the run time of an algorithm for Q by a function

$$T(n) = \text{number of basic operations performed on input of size } n \text{ (as defined above)}$$

A concrete problem is called *polynomial time solvable* if and only if

$$T(n) = O(n^k)$$

for some constant k (not necessarily an integer.) We can now (informally) define the complexity class P to be the set of decision problems that are polynomial time solvable. Note that each concrete problem is a set of bit strings, so P is a set of sets of bit strings.

Lemma
Let Q be a decision problem with two polynomially related encodings e_1 and e_2. Write $e_1(Q)$ and $e_2(Q)$ for the corresponding concrete problems. Then $e_1(Q) \in P$ if and only if $e_2(Q) \in P$.

The proof can be found on page 1056 of CLRS (3rd edition.) This lemma allows us to speak of a problem being polynomial-time solvable, without reference to any particular encoding.

Definitions
A *language L* is simply a subset of the set of all bit strings

$$L \subseteq \{0,1\}^* = \{\epsilon, 0, 1, 00, 10, 10, 11, 000, 001, 010, \ldots \ldots \ldots\}.$$

Here ϵ denotes the empty string. Let A be an algorithm whose input is any $x \in \{0,1\}^*$, and whose output $A(x)$ is one of three things.

$$A(x) = \begin{cases} 0 & \text{NO} \\ 1 & \text{YES} \\ \text{no output} & \text{algorithm does not halt} \end{cases}$$

We say that A *accepts* x if $A(x) = 1$, and that it *rejects* x if $A(x) = 0$.

Definition
The *language accepted by A* is the set $L = \{x \in \{0,1\}^* \mid A(x) = 1\}$. If we wish to emphasize that the bit strings x are encoded instances of some decision problem Q, with respect to a particular encoding e, we will write $L(Q, e) = \{e(\alpha) \mid \alpha \text{ is a YES instance of problem } Q\}$.

Note that $x \in \{0,1\}^* - L$ does not imply $A(x) = 0$, since it is possible that A does not halt on input x. Sometimes we encounter the happy situation in which an algorithm always halts.

Definition
We say that a language L is *decided* by A if and only if

$$A(x) = 1 \text{ whenever } x \in L$$

and

$$A(x) = 0 \text{ whenever } x \in \{0,1\}^* - L$$

We see that if A decides L, then it also accepts L. The converse may not be true however, and therefore accepting a language is a weaker notion than that of deciding a language. Our natural impulse is to restrict A to only those $x \in \{0,1\}^*$ for which $A(x)$ halts. Then every algorithm would decide a language $L \subseteq \text{domain}(A) \subseteq \{0,1\}^*$. The problem with this idea is that $\text{domain}(A)$ is itself non-computable, by

the unsolvability of the halting problem. Thus algorithms do not determine functions in the mathematical sense. Instead, they determine *partial functions*, mappings whose proper domain may not be known. We must therefore use the weaker notion of an algorithim accepting, rather than deciding, a language.

Definition
We say a language $L \subseteq \{0,1\}^*$ is *accepted in polynomial time* by an algorithm A if and only if there exists a constant k such that $x \in L \Leftrightarrow A(x)$ returns 1 in time $O(n^k)$.

We say $L \subseteq \{0,1\}^*$ is *decided in polynomial time* by A if and only if both $x \in L \Leftrightarrow A(x)$ returns 1 in time $O(n^k)$, and $x \notin L \Leftrightarrow A(x)$ returns 0 in time $O(n^k)$. Here of course $n = |x|$.

We now give a more formal definition of the complexity class P.

Definition

$$P = \{ L \subseteq \{0,1\}^* \mid \text{There exists an algorithm } A \text{ that } \textit{decides } L \text{ in polynomial time}\}$$

Theorem

$$P = \{ L \subseteq \{0,1\}^* \mid \text{There exists an algorithm } A \text{ that } \textit{accepts } L \text{ in polynomial time}\}$$

The proof of this theorem can be found in section 34.2 of CLRS, p. 1059. Thus for the class P, there is no distinction between accepting in polynomial time and deciding in polynomial time.

Since languages are nothing but subsets of $\{0,1\}^*$, we can define set operations on languages, along with others built up from string concatenation.

- Union: $L_1 \cup L_2$
- Intersection: $L_1 \cap L_2$
- Concatenation: $L_1 L_2 = \{ xy \mid x = L_1 \text{ and } y \in L_2\}$
- Complement: $\bar{L} = \{0,1\}^* - L$
- $L^k = LL \cdots L = \{x_1 x_2 \cdots x_k \mid x_i \in L \text{ for } i = 1, \ldots, k \}$
- $L^* = \bigcup_{i=0}^{\infty} L^i$, where $L^0 = \{\epsilon\}$ and ϵ is the empty string

….Continue in Notes….