We now turn our attention to resource-bounded computations. Resources that are of interest to us are time and space/memory. First let us consider few languages and analyze their time complexity. We concentrate on following languages

\[ SAT = \{ \phi(x_1 \cdots x_n) \mid \phi \text{ is satisfiable} \}. \]

Here \( \phi \) is a Boolean/propositional formula.

\[ HAMILTONIAN = \{ G \mid G \text{ has a Hamiltonian cycle} \}. \]

Here \( G \) is a graph, it could be directed or undirected.

\[ REACHABILITY = \{ \langle G, s, t \rangle \mid \text{There is a path from } s \text{ to } t \text{ in the graph } G \}. \]

Again \( G \) could be directed or undirected.

Consider the following algorithm for \( SAT \).

1. Input \( \phi(x_1 \cdots x_n) \).
2. For every \( x \in \{0, 1\}^n \)
   Check if \( \phi(x) = 1 \). If so, ACCEPT.
3. REJECT

Clearly, above algorithm accepts \( SAT \), and this program runs in \( O(2^n) \) time. Can you write a program that accepts \( HAMILTONIAN \)? What is the time taken by your program? Consider \( REACHABILITY \), doing a BFS or DFS tells us whether there is a path from \( s \) to \( t \). Time taken for this is \( O(n + e) \) where \( n \) number of vertices and \( e \) is the number of edges in the graph.

Our algorithm for \( SAT \) takes time exponential in input size, where the algorithm for \( REACHABILITY \) take time polynomial in input size.
**Space**

We now consider the problem of parenthesis matching. Can you write a program that takes a sequence of parentheses and tells whether they are properly matched or not? For example “())(())” is properly matched, whereas “(()(()” is not properly matched.

Consider the following algorithm. Let $x$ be a sequence of parentheses. Set a variable $c$ to zero. Start scanning each symbol of $x$. If the scanned symbol if “(” increment $c$ by one and if the scanned symbol is “)”, then decrement $c$ by one. If $c$ becomes negative at any time, then reject. After scanning the entire sequence check the value of $c$. Accepts if and only if $c$ is zero.

Clearly this algorithm runs in $O(n)$ time, where $n$ is the size of the input. How much memory this algorithm requires? This algorithm has one variable namely $c$. The maximum value $c$ can have is $n$. Recall that a number $n$ can be stored using $\log n$ bits. Thus the memory needed by the above algorithm is $\log n$.

How much memory is needed to solve REACHABILITY? Observe that the DFS/BFS algorithm keeps a list all nodes visited. Since the number of vertices is $n$, the size of this list is at most $n$, and each member of this list is a vertex. So store each vertex we need $\log n$ bits. So the memory needed is $O(n \log n)$.

Now consider the REACHABILITY in graphs with out-degree most one. That is we are given a graph $G$ and two vertices $s$ and $t$. We are guaranteed that $G$ has out degree at most one. Consider the following algorithm: Let $next$ be a variable. Initially we set $next$ to $s$. Repeat the following steps: If $next$ is $t$, then accept. Else set $next$ to its outgoing neighbor. If $next$ does not have an outgoing neighbor, then reject.

This algorithm has one variable $next$, and it stores a vertex. Thus the memory is $O(\log n)$. Observe that if there is no path from $s$ to $t$, then the above algorithm may run forever. Can we come with a different algorithm that always stops, and yet uses $O(\log n)$ memory?

Observe that if there is a path from $s$ to $t$ in $G$, then the length of this path must be at most $n$. Thus if we start from $s$ and have not reached $t$ in $n$ steps, then we must have been traversing a cycle in the graph. So there is no path from $s$ to $t$. Now consider the following algorithm: Set $next$ to $s$ and set $c$ to zero. If $next = t$, then accept. Repeat following steps: If $next = t$, then accept. Else set $next$ to its outgoing neighbor, and increment $c$. If $next$ does not have an outgoing neighbor or $c > n$, then reject. Now this algorithm always halts. This algorithm has two variables $c$ and $next$. The maximum value of next is $n$. Thus the memory is $O(\log n)$.

While measuring space, we do not count the size of the input as part of space.

**Deterministic Complexity classes**

Let $t$ and $s$ are functions from natural numbers to natural numbers. A Turing machine $M$ is $t(n)$-time bounded if for every $n$, for input $x$ of length $n$, $M$ on input $x$ halts within $t(n)$ steps. An off-line Turing machine has a read-only input tape and additional work tapes. An off-line Turing machine $M$ is $s(n)$-space bounded if for every $n$, for every $x$ of length $n$, $M$ on input $x$ visits at most $s(n)$ cells on its work tape.

$\text{DTIME}(t(n))$ is the class of languages accepted by a $t(n)$-time bounded Turing machine, and
$DSPACE(s(n))$ is the class of languages accepted by a $s(n)$-space bounded Turing machine.

$$
P = \cup_{k>0} DTIME(n^k)
$$

$$
E = \cup_{c>0} DTIME(2^c)
$$

$$
EXP = \cup_{k>0} DTIME(2^{nk})
$$

$$
LogSpace = \cup_{c>0} DSPACE(c \log n)
$$

$$
PSPACE = \cup_{k>0} DSPACE(n^k)
$$

Here are some trivial inclusion relations: $DTIME(t(n)) \subseteq DSPACE(t(n))$ as a machine can visit at most $t(n)$-cells in $t(n)$ time. Thus $P \subseteq PSPACE$. Let $t$ and $t'$ be two functions with $t(n) > t'(n)$ for every $n$. Then $DTIME(t'(n)) \subseteq DTIME(t(n))$, and $DSPACE(t'(n)) \subseteq DSPACE(t(n))$. Thus $P \subseteq E \subseteq EXP$, and $L \subseteq PSPACE$.

If a language $L$ belongs to $DSPACE(s(n))$ then there is a $s(n)$-space bounded program $P$ that accepts $L$. Thus if a string $x$ in $L$, then $P$ on input $x$ halts and accepts. However, if $x \notin L$, then $P$ may not halt. This raises the following question: If $L$ is in $DSPACE(s(n))$, then is there a $s(n)$-space bounded program that always halts?

Let $P$ be a program $x$ be a string. Assume that $P$ is $s(n)$-space bounded. Suppose we run $P$ on $x$. At any instant, the configuration of $P$ on $x$ a string that describes the following information: Contents of memory at that time and Instruction number (of $P$) that is to be executed next.

Since $P$ is $s(n)$ space bounded, the contents of memory can be described by a string of length $s(|x|)$. There are $2^{s(|x|)}$ strings of length $s(|x|)$. Thus the contents of the memory must be one of these strings. Let $|P|$ denote the number of instructions in $P$. Thus the total number of configurations of $P$ on $x$ is at most $2^{s(|x|)} \cdot |P|$.

Run $P$ on $x$. At each time step observe the configuration of $P$ on $x$. Let us say at time step $t$, it is in configuration $v$. This means that the string $v$ describes the configuration at time $t$. What happens at next time step? The contents of memory and the instruction to be executed completely determines what happens next. Note that $v$ precisely contains this information. Thus the configuration at time $t + 1$ is completely determined by $v$. Suppose at time $t$ we are in configuration $u$. Suppose at a later time $l$ we are again in configuration $v$. What is the configuration at time $t + 1$? This is again $u$.

When we run $P$ on $x$ at any time it must be in one of the $2^{s(|x|)} \cdot |P|$ configurations. So if $P$ on $x$ runs for more than $2^{s(|x|)} \cdot |P|$ time, the one of the configurations is repeated. Thus if $P$ does not halt within $2^{s(|x|)} \cdot |P|$ steps, then it will never halt and does not accept $x$.

Now consider the following program $Q$ that accepts $L$. This program has a counter $c$ whose initial value is zero. On input $x$, $Q$ run $P$ on $x$. Whenever $P$ performs a step, $Q$ increments its counter. If $P$ accepts $x$, then $Q$ accepts and stops. If $P$ rejects $x$, then $Q$ also rejects $x$ and stops. If $c$ is bigger than $2^{s(|x|)} \cdot |P|$, then $Q$ stops $P$ and rejects $x$. Now $Q$ accepts $L$ and always halts. The space needed by $Q$ is the space needed by $P$ plus the space needed to store contents of the variable $c$. Thus the total space needed by $Q$ is at most $2s(n) \log |P|$. Observe that $Q$ always halts in time $O(2^{s(n)} \cdot |P|)$.

Thus if $L$ is accepted by a $s(n)$-space bounded program $P$, then there is program $Q$ that always halts and accepts $L$. Moreover $Q$ uses $O(s(n))$ space and runs in $O(2^{s(n)})$-time.

Thus we have the following relationships: $DSPACE(s(n)) \subseteq \cup_{c>0} DTIME(c2^{s(n)})$. Thus $LogSpace \subseteq P$ and $PSPACE \subseteq EXP$. 

3
Nondeterministic Algorithms

A nondeterministic algorithm has the following additional instruction:

Guess \( x \) from \( \{0, 1\} \)

Here \( x \) is a variable. This instruction is a *nondeterministic instruction*. When a program encounters such instruction, there are two possible “paths” in which computation can proceed. In one path the variable \( x \) gets value 1 and in the other path the variable \( x \) gets value 0. Sometimes, it helps to think in terms of sub-processes. It is as if two subprocess’s are created. In one sub-process the variable \( x \) gets a value 1 and in the other sub process the variable \( x \) gets value 0.

We often use nondeterministic instruction of the following form:

Guess \( x \) from \( S \)

Here \( S \) is a finite set. The time taken for this instruction is \( \log_2 |S| \)-steps. This instruction creates \( |S| \) paths. In each path the variable \( x \) is assigned to a member of \( S \).

Let \( P \) be a nondeterministic program and \( x \) be a string. We say that \( P \) accepts \( x \) if at least one computation path of \( P \) on \( x \) accepts. We say \( P \) does not accept \( x \) if every computation path of \( P \) on \( x \) either rejects or runs forever. We say that a nondeterministic program \( P \) accepts a language \( L \) if \( P \) accepts all string from \( L \) and does not accept any string from \( T \).

Let \( t : \mathbb{N} \to \mathbb{N} \) be a function. We say that a nondeterministic program \( P \) is \( t(n) \)-time bounded, if for every \( n \in \mathbb{N} \), for every \( x \in \Sigma^n \), every path of \( P \) on \( x \) stops within \( t(n) \) steps.

We will now consider non deterministic algorithms for \( SAT \) and Hamiltonian.

1. Input \( \phi(x_1, \ldots, x_n) \).
2. For \( i = 1 \) to \( n \) “Guess \( a_i \) from \( \{0, 1\} \)”
3. If \( \phi(a_1 \cdots a_n) = 1 \) accept, else reject.

If the formula \( \phi \) is really satisfiable, then there is an assignment \( b_1 \cdots b_n \) such that \( \phi(b_1 \cdots b_n) = 1 \). There is computation path in which \( a_1 \cdots a_n = b_1 \cdots b_n \), and this path accepts \( \phi \). If the formula \( \phi \) is not satisfiable, then for every assignment \( b_1 \cdots b_n \), \( \phi(b_1 \cdots b_n) = 0 \). So every path of the above program rejects \( \phi \). Thus the above program accepts \( SAT \). And the running time of the above program is \( O(n) \).

Now consider the following language

\[ HAMILTONIAN = \{ G \mid G \text{ has a Hamiltonian cycle} \} \]

Here is a nondeterministic program for \( HAMILTONIAN \).

1. Input \( G \). Say the vertex set of \( G \) is \( \{v_1, \cdots, v_n\} \).
2. \( s_1 = v_1 \).
3. For \( i = 2 \) to \( n \) “Guess \( s_i \) from \( \{v_1, \cdots, v_n\} \)”
4. If \( s_1 s_2 \cdots s_n s_1 \) is Hamiltonian cycle in \( G \) the accept, else reject.

Check that the above program accepts \( HAMILTONIAN \). Step 2 has \( n - 1 \) nondeterministic instructions, and each instruction takes \( \log n \) steps. The last step needs \( O(n) \) time. Thus the running time of the above program is \( O(n \log n) \).
Non-deterministic Complexity Classes

Let \( t : \mathbb{N} \rightarrow \mathbb{N} \) be a function. We say that a non-deterministic program \( P \) is \( t(n) \)-time bounded, if for every \( n \in \mathbb{N} \), for every \( x \in \Sigma^n \), every path of \( P \) on \( x \) stops within \( t(n) \) steps.

Similarly a non-deterministic program \( P \) is \( t(n) \)-space bounded, if for every \( n \in \mathbb{N} \), for every \( x \in \Sigma^n \), every path of \( P \) on \( x \) uses at most \( s(n) \) amount of memory.

Now we have the following non-deterministic complexity classes.

The class \( NTIME(t(N)) \) is the collection of all languages that are accepted by a \( t(n) \)-time bounded non-deterministic program. Similarly, \( NSPACE(s(N)) \) is the collection of all languages that are accepted by a \( s(n) \)-space bounded non-deterministic program.

Now \( NP = \bigcup_{k>0} NTIME(n^k) \), \( NL = \bigcup_{c>0} NSPACE(c \log n) \), and \( NPSPACE = \bigcup_{k>0} NSPACE(n^k) \).