PARTITION

Partition: Given a set of $n$ nonnegative integers $S = \{a_1, ..., a_n\}$ represented in binary, is there a subset $P \subseteq \{1, ..., n\}$ such that

$$\sum_{i \in P} a_i = \frac{1}{2} \sum_{i=1}^{n} a_i$$

- Let $H$ be the sum of all integers in $S$ divided by two (not necessarily an integer)
- For each $i, 0 \leq i \leq n$, define a set of numbers:
  $$B(i) = \{ b \leq H \mid b \text{ is the sum of some subset of the numbers } \{a_1, ..., a_i\} \}$$
- If we knew $B(n)$, we could solve the partition problem easily by testing whether $H \in B(n)$
- $B(n)$ can be computed by the following algorithm

$$B(0) \leftarrow \{0\}$$

for $i = 1, 2, ..., n$ do

$$B(i) \leftarrow B(i-1),$$

for $j = a_i, a_i+1, a_i+2, ..., H$ do

if $j - a_i \in B(i-1)$ then add $j$ to $B(i)$

An example:  

$S = \{38, 17, 52, 61, 21, 88, 25\},$

$H = 38 + 52 + 61 = 17 + 21 + 88 + 25 = 151$

$B(0) = \{0\}$

$B(1) = \{0, 38\}$

$B(2) = \{0, 17, 38, 55\}$

$B(3) = \{0, 17, 38, 52, 55, 69, 90, 107\}$

...  

$B(7) = \{0, 17, 21, 25, 38, 42, 46, 52, 55, 59, 61, 63, ..., 151\}$

So the answer is "yes"
Induction on $i$ shows that this algorithm correctly computes $B(i)$ for $i = 0, \ldots, n$, and does so in $O(nH)$ time.

So is PARTITION in P?

- Despite all, the bound $O(nH)$ is not polynomial in the length of the input.
- The integers $a_1, \ldots, a_n$ are given in binary and, thus, their sum will in general be exponentially large when compared to the length of the input.
- E.g., if all integers in an instance of PARTITION are about $2^n$, then $H$ is approximately $(n/2)2^n$.
- The length of the input is only $O(n^2)$.

**KNAPSACK-1**

**Knapsack-1**: Given a set $S = \{a_1, \ldots, a_n\}$ of nonnegative integers, and an integer $k$, all represented in binary, is there a subset $P \subseteq S$ such that

$$\sum_{i \in P} a_i = k$$

- **KNAPSACK-1** is obviously closely related to PARTITION.
- To reduce an instance of **KNAPSACK-1** with integers $a_1, \ldots, a_n$ and $K$ to an equivalent instance of PARTITION when it happens that $K = H$, all that we have to do is erase $K$ from input.
- In general, $K$ is of course not equal to $H$, but that is easy to fix:
  - Add to $S$ two large new integers
    - $a_{n+1} = 2H + 2K$
    - $a_{n+2} = 4H$
The reduction can obviously be carried out in polynomial time.

We need to show that an instance of \textsc{Partition} has a solution if and only if the original instance of \textsc{Knapsack-1} had one.

If the new set of integers can be partitioned into two sets with equal sums, then the two new integers must be on opposite sides.

Let be \( P \) the set of integers among the original ones \( a_p, \ldots, a_n \) that are on the same side as \( a_{n+2} = 4H \).

We have
\[
4H + \sum_{a_i \in P} a_i = 2H + 2K + \sum_{a_i \notin P} a_i
\]

Adding \( \sum_{a_i \notin P} a_i \) to both sides yields
\[
4H + 2 \sum_{a_i \notin P} a_i = 4H + 2K \iff \sum_{a_i \notin P} a_i = K
\]

Hence, if the resulting instance of \textsc{Partition} has a solution, then the original instance of \textsc{Knapsack-1} has one. And conversely, if we have a solution of \textsc{Knapsack-1}, then adding \( a_{n+2} \) to it yields a solution of the instance of \textsc{Partition}.

\[
\text{SAT / CSAT} \quad \text{3SAT} \quad \text{Subset-sum} \quad \text{VC} \quad \text{Hamiltonian path} \\
\text{Knapsack-1} \quad \text{IS} \quad \text{TSP} \quad \text{Partition} \quad \text{CLIQUE}
\]
NP-intermediate problems

- If $P \neq NP$ then there exist problems in $NP$ that are neither in $P$ nor $NP$-complete
- Such problems are called **NP-intermediate problems**
- The graph isomorphism problem, the discrete logarithm problem and the integer factorization problem are examples of problems believed to be $NP$-intermediate
- The graph isomorphism problem is the computational problem of determining whether two finite graphs are isomorphic.
- Whether the graph isomorphism problem is in $P$, $NP$-complete, or $NP$-intermediate is unsolved
- It is believed that the problem is at least not $NP$-complete
- The best algorithm for this problem has run time $2^{O((\log n))}$ for graphs with $n$ vertices.

The integer factorization problem is the computational problem of determining the prime factorization of a given integer

- Phrased as a decision problem, it is the problem of deciding whether the input has a factor less than $k$
- No efficient integer factorization algorithm is known, and this fact forms the basis of several modern cryptographic systems, such as the RSA algorithm
- The best known algorithm for integer factorization is the general number field sieve
- However, the best known quantum algorithm for this problem, Shor's algorithm, does run in polynomial time
- Unfortunately, this fact doesn't say much about where the problem lies with respect to non-quantum complexity classes
8. Space Complexity

- The space complexity of a standard Turing machine \( M = (Q, \Sigma, \Gamma, \delta, q_0, \text{accept}, \text{reject}) \) on input \( w \) is

\[
\text{space}_M(w) = \max\{ |uav| : q_0 \xrightarrow{u} q \text{ accepts } q \in Q, u, a, v \in \Gamma^* \}
\]

- The space complexity of a nondeterministic Turing machine \( N = (Q, \Sigma, \Gamma, \delta, q_0, \text{accept}, \text{reject}) \) on input \( w \) is

- \( \text{space}_N(w) \) = the maximum number of tape cells scanned by the computation \( q_0 \xrightarrow{...} \text{requiring the most space} \)

Let \( s : \mathbb{N} \to \mathbb{R}^+ \) be an arbitrary function

- The deterministic space complexity class is:

\[
\text{DSPACE}(s(n)) = \{ L \mid L \text{ is a language decided by an } O(s(n)) \text{ space deterministic TM} \}
\]

- Respectively, the nondeterministic space complexity class is:

\[
\text{NSPACE}(s(n)) = \{ L \mid L \text{ is a language decided by an } O(s(n)) \text{ space nondeterministic TM} \}
\]

- Space can be reused, while time cannot
- E.g., deciding an instance of SAT only requires linear space, while it most probably is not polynomial-time decidable, since it is \( \text{NP-complete} \)
• The composite space complexity classes are:

\[ \text{PSPACE} = U \{ \text{DSPACE}(s) \mid s \text{ is a polynomial} \} \]
\[ = U_{k \geq 0} \text{DSPACE}(n^k) \]

\[ \text{EXPSPACE} = U_{k \geq 0} \text{DSPACE}(2^{nk}) \]

\[ \text{NPSPACE} = U_{k \geq 0} \text{NSPACE}(n^k) \]

\[ \text{NEXPSPACE} = U_{k \geq 0} \text{NSPACE}(2^{nk}) \]

\[ \text{Lemma} \] For all \( t(n), s(n) \geq n \):

1. \( \text{DTIME}(t(n)) \subseteq \text{DSPACE}(t(n)) \), and

2. \( \text{DSPACE}(s(n)) \subseteq U_{k \geq 0} \text{DTIME}(k^{s(n)}) \)

\[ \text{Proof.} \]

1. In \( t(n) \) steps a Turing machine can write at most \( t(n) \) symbols to the tape.

2. A TM requiring space \( s(n) \) has at most \( k^{s(n)} \) \( (k \) constant) distinct configurations on an input of length \( n \). By letting the computation continue for at most \( k^{s(n)} \) steps we can make the TM always use time at most \( k^{s(n)} \). \( \Box \)

\[ \text{Consequence:} \quad P \subseteq \text{PSPACE} \subseteq \text{EXPTIME} \subseteq \text{EXPSPACE} \]
8.1 Savitch’s Theorem

**Theorem 8.5** Let \( f: \mathbb{N} \to \mathbb{R}^+ \) be any function, where \( f(n) \geq n \).
Then
\[
\text{NSPACE}(f(n)) \subseteq \text{DSPACE}(f^2(n)).
\]

**Proof.**

- Simulate a nondeterministic TM \( N \) with a deterministic one \( M \).
- Naïve approach: try all the branches of the computation tree of \( N \) one by one.
- A branch using \( f(n) \) space may run for \( 2^{O(f(n))} \) steps and each step may be a nondeterministic choice.
- Reach the next branch \( \Rightarrow \) record all the choices of the branch.
- Thus, in the worst case the naïve approach may use \( 2^{O(f(n))} \) space
- Hence, the naïve approach is not the one we are looking for.

Instead, let us examine the **yieldability problem**: can a TM \( N \) get from configuration \( c_1 \) to configuration \( c_2 \) within \( t \) steps.

By deterministically solving the yieldability problem, where
- \( c_1 \) is the start configuration of \( N \) on some \( w \) and
- \( c_2 \) is the accept configuration,
without using too much space implies the claim.

We search an intermediate configuration \( c_m \) of \( N \) such that
- \( c_1 \) can get to \( c_m \) within \( t/2 \) steps, and
- \( c_m \) can get to \( c_2 \) within \( t/2 \) steps.
We can reuse the space of the recursive calls.
CANYIELD(c₁, c₂, t): \% t is a power of 2

1. If \( t = 1 \), then test directly whether \( c₁ = c₂ \) or whether \( c₁ \) yields \( c₂ \) in one step according to the rules of \( N \).
   - Accept if either test succeeds; reject if both fail.
2. If \( t > 1 \), then for each configuration \( cₘ \) of \( N \) on \( w \) using space \( f(n) \):
   3. Run CANYIELD(c₁, cₘ, t/2).
   4. Run CANYIELD(cₘ, c₂, t/2).
   5. If steps 3 and 4 both accept, then accept.
   6. If haven’t yet accepted, reject.

\( M = \) “On input \( w \):

1. Output the result of CANYIELD(c\text{start}, c\text{accept}, 2^{d(n)}):
2. The constant \( d \) is selected so that \( N \) has no more than \( 2^{d(n)} \)
   configurations using \( f(n) \) tape, where \( n = |w| \)
3. The algorithm needs space for storing the recursion stack.
4. Each level of the recursion uses \( O(f(n)) \) space to store a configuration.
   - Just store the current step number of the algorithm and values \( c₁, c₂ \), and \( t \) on a stack
   - Because each level of recursion divides the size of \( t \) in half, the depth of the recursion is logarithmic in the maximum time that \( N \) may use on any branch; i.e., \( \log(2^{O(f(n))}) = O(f(n)) \).
5. Hence, the deterministic simulation uses \( O(f(n)) \) space.
Corollary
1. NPSPACE = PSPACE
2. NEXPSpace = EXPSPACE

- Now our knowledge of the relationship among time and space complexity classes extends to the linear series of containments:

\[
\begin{align*}
P & \subseteq NP \\
\{PSPACE = \} & \subseteq \text{EXPTIME} \\
\{\text{EXPSPACE} = \} & \subseteq \text{NEXPSpace}
\end{align*}
\]

Theorem
1. P ≠ EXPTIME
2. PSPACE ≠ EXPSPACE

- Complexity classes that have "exponential distance" from each other are known to be distinct:

\[
P \neq \text{EXPTIME}, NP \neq \text{NEXPTIME} \text{ and } PSPACE \neq \text{EXPSPACE}
\]

- However, one does not know whether the following hold:

\[
P \neq NP, NP \neq PSPACE, PSPACE \neq \text{EXPTIME}, ...
\]

- Most researchers believe that all these inequalities hold, but cannot prove the results.

- There does not exist the "most difficult" decidable language.
Theorem

1. If \( A \) is decidable, then there exists a computable function \( t(n) \), for which \( A \in \text{DTIME}(t(n)) \).

2. For every computable function \( t(n) \) there exists a decidable language \( A \) not belonging to \( \text{DTIME}(t(n)) \).

Function \( f(n) = f(n, n) \) is computable
- \( f(0, n) = 2^{2^2} \times n \) times
- \( f(m+1, n) = 2^{2^{f(m, n)}} \times f(m, n) \) times
- \( f(0) = 1, f(1) = 4, f(2) = 2^{2^{2^{16}}} \times 65536 \) times

By the above theorem: \( A \in \text{DEC} \setminus \text{DTIME}(f(n)) \neq \emptyset \)
In other words there are “decidable” problems, whose time complexity on input of length \( n \) is not bounded by function \( f(n) \).

8.3 PSPACE-Completeness

- A language \( B \) is PSPACE-complete, if
  1. \( B \in \text{PSPACE} \) and
  2. \( A \preceq_p B \) for all \( A \in \text{PSPACE} \)

- If \( B \) merely satisfies condition 2, we say that it is PSPACE-hard
- In a fully quantified Boolean formula each variable appears within the scope of some quantifier; e.g.
  \[ \varphi = \forall x \exists y [(x \lor y) \land \neg x \lor \neg y] \]
- TQBF is the language of true fully quantified Boolean formulas

Theorem 8.9 TQBF is PSPACE-complete.
A polynomial space algorithm deciding TQBF

\( T = \) "On input (\( \phi \)), a fully quantified Boolean formula:

1. If \( \phi \) contains no quantifiers, then it is an expression with only constants, so evaluate \( \phi \) and accept if it is true; otherwise, reject.
2. If \( \phi \) equals \( \exists x \psi \), recursively call \( T \) on \( \psi \), first with 0 substituted for \( x \) and then with 1 substituted for \( x \). If either result is accept, then accept; otherwise, reject.
3. If \( \phi \) equals \( \forall x \psi \), recursively call \( T \) on \( \psi \), first with 0 substituted for \( x \) and then with 1 substituted for \( x \). If both results are accept, then accept; otherwise, reject.

We need to reduce any \( A \in \text{PSPACE} \) polynomially to TQBF starting from the polynomial-space bounded TM \( M \) for \( A \)

- Imitating the proof of Cook-Levin theorem is out of the question because \( M \) can in any case run for exponential time
- Instead, one has to resort to a proof technique resembling the one used in the proof of Savitch’s theorem
- The reduction maps string \( w \) to a quantified Boolean formula \( \varphi \) that is true if and only if \( M \) accepts \( w \)
- Collections of variables denoted \( c_1 \) and \( c_2 \) represent two configurations
We construct a formula \( \varphi(c_1, c_2, t) \), which is true if and only if \( M \) can go from \( c_1 \) to \( c_2 \) in at most \( t > 0 \) steps.

This yields a reduction when we construct the formula \( \varphi(c_{\text{start}}, c_{\text{accept}}, h) \), where \( h = 2^{df(n)} \).

The constant \( d \) is chosen so that \( M \) has no more than \( 2^{df(n)} \) possible configurations on an input of length \( n \).

The formula encodes the contents of tape cells.

Each cell has several indicator variables associated with it,

- one for each tape symbol and one for each state of \( M \)

Each configuration has \( n^k \) cells and so is encoded by \( O(n^k) \) variables.

If \( t = 1 \), then

1. either \( c_1 = c_2 \), or
2. \( c_2 \) follows from \( c_1 \) in a single step of \( M \)

Now constructing formula \( \varphi(c_1, c_2, t) \) is easy.

1. Each of the variables representing \( c_1 \) has the same Boolean value as the corresponding variable representing \( c_2 \).
2. (As in the proof of Cook-Levin theorem) writing expressions stating that the contents of each triple of \( c_1 \)'s cells correctly yields the contents of the corresponding triple of \( c_2 \)'s cells.
• If $t > 1$, we construct $\varphi(c_1, c_2, t)$ recursively.

• A straightforward approach would be to define

$$\varphi(c_1, c_2, t) = \exists m_1 \{ \varphi(c_1, m_1, t/2) \land \varphi(m_1, c_2, t/2) \},$$

where $m_1$ represents a configuration of $M$.

• Writing $\exists m_1$ is a shorthand for $\exists x_1, ..., x_l$, where $l = O(n^k)$ and $x_1, ..., x_l$ are the variables that encode $m_1$.

• $\varphi(c_1, c_2, t)$ has the correct value, its value is true whenever $M$ can go from $c_1$ to $c_2$ within $t$ steps.

• However, the formula becomes too big.

• Every level of recursion cuts $t$ in half, but roughly doubles the size of the formula $\Rightarrow$ we end up with a formula of size $\approx t$.

• Initially $t = 2^{\text{df}(n)} \Rightarrow$ exponentially large formula.

• To shorten formula $\varphi(c_1, c_2, t)$ we change it to form

$$\exists m_1, \forall (c_3, c_4) \in \{ (c_2, m_1), (m_1, c_4) \}: [ \varphi(c_3, c_4, t/2) ].$$

• This formula preserves the original meaning, but folds the two recursive subformulas into a single one.

• To obtain a syntactically correct formula

$$\forall x \in \{ y, z \}: [...]$$

may be replaced with the equivalent

$$\forall x \{ (x = y \lor x = z) \Rightarrow [...] \}.$$

• Each level of recursion adds a portion of the formula that is linear in the size the configurations, i.e., $O(f(n))$.

• The number of levels of recursion is $\log(2^{\text{df}(n)}) = O(f(n))$.

• Hence the size of the resulting formula is $O(f^2(n))$.
The course book presents the PSPACE-completeness of two "artificial" games. For them no polynomial time algorithm exists unless $P = PSPACE$.

- Standard chess on a $8 \times 8$ board does not directly fall under the same category.
- There is only a finite number of different game positions.
- In principle, all these positions may be placed in a table, along with the best move for each position.
- Thus, the control of a Turing machine (or finite automaton) can store the same information and it can be used to play optimally in linear time using table lookup.
- Unfortunately the table would be too large to fit inside our galaxy.

Current complexity analysis methods are asymptotic and apply only to the rate of growth of the complexity as the problem size increases. The complexity of problems of any fixed size cannot be handled using these techniques.

- Generalizing games to an $n \times n$ board gives some evidence for the difficulty of computing optimal play.
- For example, generalizations of chess and GO have been shown to be at least PSPACE-hard.
8.4 The Classes L and NL

- In time complexity there is really no point in considering sublinear complexity classes, because they are not sufficient for reading the entire input.
- On the other hand, one does not necessarily need to store the entire input and, therefore, it makes sense to consider sublinear space complexity classes.
- To make things meaningful, we must modify our model of computation.
- Let a Turing machine have in addition to the read/write work tape a read only input tape.
- Only the cells scanned on the work tape contribute to the space complexity of the TM.

$L$ is the class of languages that are decidable in logarithmic space on a deterministic Turing machine:

$$L = \text{DSPACE}(\log n)$$

$NL$ is the respective class for nondeterministic TMs.

- Earlier we examined deciding language $A = \{ 0^k 1^k \mid k \geq 0 \}$ with a Turing machine requiring linear space.
- We can decide whether a given string belongs to the language using only a logarithmic space.
- It suffices to count the numbers of zeros and ones, separately (and make the necessary checks). The binary representations of the counters uses only logarithmic space.
- Therefore, $A \in L$. 
Earlier we showed that the language \textit{PATH: }"$G$ has a directed path from $s$ to $t$." is in $P$

The algorithm given uses linear space

We do not know whether \textit{PATH} can be solved deterministically in logarithmic space

A nondeterministic logarithmic space algorithm, on the other hand, is easy to find

Starting at node $s$ repeat at most $m$ steps

\begin{itemize}
  \item If current node is $t$, then \textit{accept}
  \item Record the position of the current node on the work tape (logarithmic space)
  \item Nondeterministically select one of the followers of the current node
\end{itemize}

For very small space bounds the earlier claim that any $f(n)$ space bounded TM also runs in time $2^{O(f(n))}$ is no longer true

For example, a Turing machine using constant $O(1)$ space may run for a linear $O(n)$ number of steps

For small space bounds the time complexity has the asymptotic upper bound $n2^{O(f(n))}$

If $f(n) \geq \log n$, then $n2^{O(f(n))} = 2^{O(f(n))}$

Also Savitch's theorem holds as such when $f(n) \geq \log n$

As mentioned, \textit{PATH} is known to be in $\textit{NL}$ but probably not in $\textit{L}$

In fact, we don’t know of any problem in $\textit{NL}$ that can be proved to be outside $\textit{L}$
8.5 NL-Completeness

- $L =?= \text{NL}$ is an analogous question to $P =?= \text{NP}$
- We can define $\text{NL}$-complete languages as the most difficult languages in $\text{NL}$
  
  - Polynomial time reducibility, however, cannot be used in the definition, because all problems in $\text{NL}$ are solvable in polynomial time
  - Therefore, every two problems in $\text{NL}$ except $\emptyset$ and $\Sigma^*$ are polynomial time reducible to one another
  
  - Instead we use logarithmic space reducibility $\leq_L$
  - A function is log space computable if the Turing machine computing it uses only $O(\log n)$ space from its work tape

Theorem 8.23 If $A \leq_L B$ and $B \in L$, then $A \in L$. □

Corollary 8.24 If any $\text{NL}$-complete language is in $L$, then $L = \text{NL}$. □

Theorem 8.25 $\text{PATH}$ is $\text{NL}$-complete. □

Corollary 8.26 $\text{NL} \subseteq P$.

Proof. Theorem 8.25 shows that for any language $A$ in $\text{NL}$ it holds that $A \leq_L \text{PATH}$. A Turing machine that uses space $f(n)$ runs in time $n^{2^{O(f(n))}}$, so a reducer that runs in log space also runs in polynomial time.

Because $A \leq_{m} \text{PATH}$ and $\text{PATH} \in P$, by Theorem 7.14, also $A \in P$. □
8.6 NL = coNL

- For Example, NP and coNP are generally believed to be different
- At first glance, the same would appear to hold for NL and coNL
- There are, however, gaps in our understanding of computation:

**Theorem 8.27** $\text{NL} = \text{coNL}$

**Proof.** The complement of $\text{PATH}$ — “The input graph $G$ does not contain a path from $s$ to $t$.” — is in NL. Because $\text{PATH}$ is NL-complete, every problem in coNL is also in NL. □

- Our knowledge of the relationship among complexity classes $L \subseteq \text{NL} \subseteq \text{coNL} \subseteq \text{P} \subseteq \text{PSPACE}$
- Whether any of these containments are proper is unknown, but all are believed to be