Recitation 08: P and NP

This recitation covers some basic definitions and tools of complexity theory, and uses dynamic programming to show that a language is in P.

In the first half of the semester, we learned about computability theory, where we placed languages in classes (regular, context-free, decidable, T-recognizable) based on whether they could be solved using certain models of computation. We now move on to complexity theory, where we restrict our attention to *decidable* languages and first focus on determining how much *time* is required to decide them.

Time complexity for deterministic models

To determine the time complexity of a language, we must first define what it means for a Turing machine to run in $t(n)$ time, where $t : \mathbb{N} \to \mathbb{N}$.

Definition 1. A single-tape deterministic TM M runs in time $t(n)$ if M halts in at most $t(n)$ steps on all inputs of length n.

Definition 2. $TIME(t(n)) = \{B \mid$ some single-tape deterministic TM M runs in $O(t(n))$ time and $L(M) = B\}$.

A TM runs in $O(t(n))$ time if it runs in $\leq ct(n)$ time for some constant $c > 0$ independent of n.

We can now define our first time complexity class.

Definition 3. $P = \bigcup_{k \in \mathbb{N}} TIME(n^k)$.

In words, P is the set of languages that can be decided in cn^k time for constants c, k. These are the languages that can be decided in *polynomial time*. To show that a language is in P, we typically must construct a TM that decides that language, then argue that the TM halts in polynomial time.

Model independence

While Definitions 1 and 2 require a single-tape deterministic TM, Definition 3 is *model independent*. This means that for all reasonable deterministic models of computation, P defines the same class of languages.

This is useful because we are not restricted to any specific deterministic model when proving languages are in P.

Time complexity for nondeterministic models

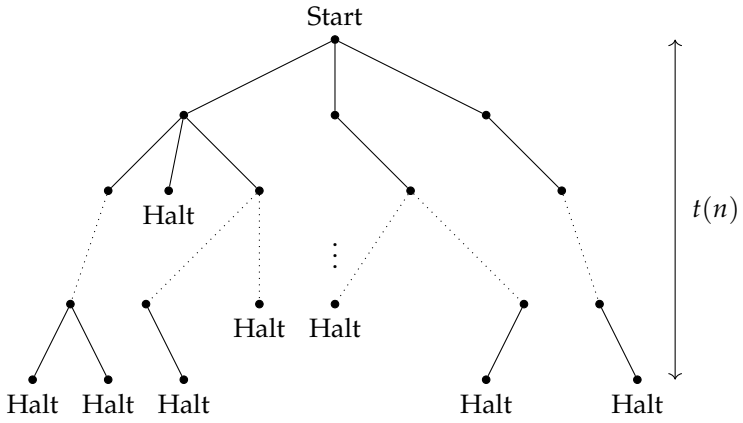
We define the analogous terms for nondeterministic TMs.

Definition 4. A nondeterministic TM N runs in time $t(n)$ if all of the threads of N halt in at most $t(n)$ steps on all inputs of length n.

Note the difference between acceptance and runtime: for an NTM to accept an input, it is enough for one thread to accept. For an NTM to run in time $t(n)$, *all* threads need to halt in that time.

Definition 5. $NTIME(t(n)) = \{B \mid \text{some NTM } N \text{ runs in } O(t(n)) \text{ time and } L(N) = B\}$.

Intuitively, this means that the tree consisting of all the branches of N's computation can have *height* at most $t(n)$, as shown below. Note however that these definitions do not limit the *width* of the tree, and there could be a non-polynomial number of branches, as long as each branch has polynomial length.



Definition 6. $NP = \bigcup_{k \in \mathbb{N}} NTIME(n^k)$.

In words, NP is the set of languages decided by some NTM in polynomial time. This definition is again model independent for all reasonable nondeterministic models of computation.

Certificates for NP

Intuitively, NP consists of languages L for which we can *verify membership quickly*. For an input $x \in L$, there is some short "certificate" c such that if given c, it is easy to confirm that x is in L. Here are some examples of certificates for instances of languages in NP:

Here, "short," "quickly," and "easy" all mean polynomial in $|x|$.

- If $\langle G, s, t \rangle \in HAMPATH$, the certificate would be the sequence of nodes corresponding to a Hamiltonian path from s to t in G. It is certainly polynomial time to check that each node in the sequence

is in G, the sequence starts at s and ends at t, all nodes in G appear in the sequence exactly once, and all pairs of adjacent nodes in the sequence are connected by an edge in G.

- For the *SUBSET-SUM* language, defined as $\{\langle S, t\rangle \mid S = \{x_1, ..., x_k\}$ and $\exists\{y_1, ..., y_l\} \subseteq S$ such that $\sum_{i=1}^{l} y_i = t\}$, the certificate would be $c = \{y_1, ..., y_l\}$. Given c, it is easy to check that $c \subseteq S$ and that the sum of the elements of c is t.

This concept can be formalized with the following theorem.

Theorem 7. *$L \in NP \iff$ there exists verifier TM V such that $L(V) \in P$ and ($x \in L \iff$ there exists c such that $|c| = O(poly(|x|))$ and $\langle x, c\rangle \in V$).*

Proof. Informally, we want to show that $L \in NP \iff$ strings in L have short and quickly checkable certificates, and strings not in L don't.

($\Longrightarrow$) Let $L \in NP$. Then, there exists NTM N that decides L in polynomial time. We will show that strings in L have short and quickly checkable certificates. Construct TM V which takes in $\langle x, c\rangle$ and accepts iff c is an accepting computation history of N on x. From our previous algorithms for checking that computation histories are valid and accepting, we know that V runs in polynomial time. Then, for all $x \in L$, let c be an accepting computation history of N on x, which exists since N accepts x. We know that c is short since N runs in polynomial time. Conversely, if $x \notin L$, then N has no accepting computation history on x, so V accepts no $\langle x, c\rangle$.

Intuitively, think of the computation history as all the nondeterministic choices that N made on an input. These choices specify a thread, and V just needs to check whether this thread leads to an accept state.

($\Longleftarrow$) Let language L have short and quickly checkable certificates for strings in L. We will show that $L \in NP$. Let V be the verifier for L; since $L(V) \in P$, we may assume that V halts in polynomial time on every input. Let p be a polynomial such that every $x \in L$ of length n has a certificate c with $|c| \leq p(n)$. We can thus build NTM N: on input x of length n, nondeterministically guess certificate c of length at most $p(n)$. Run V on $\langle x, c\rangle$ and accept if and only if V accepts. N will accept x if and only if there exists a c with $|c| \leq p(n)$ such that V accepts $\langle x, c\rangle$, so $L(N) = L$. All threads will halt in polynomial time since $|c| \leq p(n)$ is polynomial, so $|\langle x, c\rangle|$ is polynomial in n, and V runs in polynomial time. □

A small detail on guessing: since a TM has a fixed number of states, we can't guess the entire c all at once (since the number of possibilities may depend on n). Instead we can guess c bit-by-bit.

Given Theorem 7, it is now easy to show that a language $L \in NP$. To construct an NTM running in polynomial time that decides L, we can

1. Think of a (short, easy to check) certificate for strings in L.
2. Build the following NTM. On input x:
   (a) Guess the certificate c.

(b) Check whether or not c is a valid certificate for x. Accept if so. Else, reject.

Example 8. *$HAMPATH = \{\langle G, s, t\rangle : G$ is a directed graph with a Hamiltonian path from s to $t\} \in NP$.*

Proof. Let n be the number of vertices of G. A Hamiltonian path from s to t is a sequence of vertices $s = v_1, \ldots, v_n = t$ in which every vertex of G appears exactly once and (v_i, v_{i+1}) is a (directed) edge for every $1 \leq i < n$. Following the recipe above, we use the Hamiltonian path itself as the certificate. Construct NTM N = "On input $\langle G, s, t\rangle$:

1. Nondeterministically guess a sequence $v_1, \ldots, v_n$ of n vertices of G.
2. Accept if $v_1 = s$, $v_n = t$, no vertex appears twice in the sequence, and $(v_i, v_{i+1}) \in E(G)$ for every $1 \leq i < n$. Else, reject."

Since the sequence has exactly n entries and no repeats, it contains every vertex exactly once. So a thread accepts iff its guess is a Hamiltonian path from s to t, and N accepts iff $\langle G, s, t\rangle \in HAMPATH$.

Each vertex has an $O(\log n)$ bit representation, so stage 1 guesses $O(n \log n)$ bits. Stage 2 compares $O(n^2)$ pairs of vertices and scans the description of G once for each of the $n - 1$ consecutive pairs. All of this is polynomial in n, and hence in the input length (which is at least n), so every thread halts in polynomial time. □

Dynamic programming

We now use *dynamic programming* to show that a language is in P. Consider the variant of *SUBSET-SUM* where all numbers are written in *unary*:

$$UNARY\text{-}SUBSET\text{-}SUM = \{\langle S, t\rangle \mid \exists I \subseteq \{1, \ldots, k\} \text{ with } \textstyle\sum_{i \in I} x_i = t\},$$

where $S = \{x_1, \ldots, x_k\}$, and $x_1, \ldots, x_k$ and t are natural numbers written in unary, i.e., a number m is written as 1^m, a string of m ones. For example, 5 is written as 11111 (rather than 101 in binary). The key observation is that since t is written in unary, the input length n is at least t. Similarly, $n \geq k$, since the input must contain at least one symbol (such as a separator) for each of the k numbers.

We sum over a set of *indices* I rather than a subset of S so that repeated numbers in S are handled correctly. The empty set $I = \emptyset$ is allowed and has sum 0.

Example 9. *$UNARY\text{-}SUBSET\text{-}SUM \in P$.*

Proof. We break the problem into subproblems $SS(i, j)$ for $0 \leq i \leq k$ and $0 \leq j \leq t$, where $SS(i, j) =$ TRUE if some subset of $x_1, \ldots, x_i$ sums to j, and FALSE otherwise. Then $\langle S, t\rangle \in UNARY\text{-}SUBSET\text{-}SUM$ iff $SS(k, t) =$ TRUE. We have $SS(0, j) =$ TRUE iff $j = 0$, and for $i \geq 1$,

$$SS(i, j) = SS(i-1, j) \ \vee \ \big(x_i \leq j \ \wedge \ SS(i-1, j - x_i)\big),$$

since a subset of $x_1, \ldots, x_i$ summing to j either omits x_i (so it is a subset of $x_1, \ldots, x_{i-1}$ summing to j), or contains x_i (so the rest is a subset of $x_1, \ldots, x_{i-1}$ summing to $j - x_i$, which forces $x_i \leq j$). There are $(k+1)(t+1) \leq (n+1)^2$ subproblems, and dynamic programming computes each of them only once. There are two standard ways to do this.

Approach 1: Bottom-up (tabulation)
Construct TM M_1 = "On input $\langle S, t \rangle$:

1. Create a table T indexed by $0 \leq i \leq k$ and $0 \leq j \leq t$. Set $T[0,0] =$ TRUE and $T[0,j] =$ FALSE for $1 \leq j \leq t$.
2. For $i = 1, \ldots, k$ and $j = 0, \ldots, t$: set $T[i,j] = T[i-1,j] \vee \big(x_i \leq j \wedge T[i-1, j-x_i]\big)$.
3. Accept if $T[k,t] =$ TRUE. Else, reject."

Row $i-1$ is filled in before row i, so by induction on i, $T[i,j] = SS(i,j)$ for every entry. Each of the at most $(n+1)^2$ entries takes one comparison and at most two lookups in the previous row, which a TM can do in polynomial time by scanning its tape. So M_1 decides *UNARY-SUBSET-SUM* in polynomial time.

Approach 2: Top-down (recursion with memoization)
Construct TM M_2 = "On input $\langle S, t \rangle$:

1. Create a table T indexed by $0 \leq i \leq k$ and $0 \leq j \leq t$, with all entries blank.
2. Compute $SS(k,t)$ recursively, where each $SS(i,j)$ is computed as follows:
   (a) If $T[i,j]$ is not blank, return $T[i,j]$.
   (b) Otherwise, compute $SS(i,j)$ by the recurrence: if $i = 0$, it is TRUE iff $j = 0$; if $i \geq 1$, recursively compute $SS(i-1,j)$ and, if $x_i \leq j$, $SS(i-1, j-x_i)$.
   (c) Store the result in $T[i,j]$ and return it.
3. Accept if $SS(k,t) =$ TRUE. Else, reject."

Each recursive call decreases i by one and keeps j between 0 and t, so the recursion only involves the $(k+1)(t+1)$ subproblems above, and since it evaluates the recurrence, it computes $SS(k,t)$ correctly. Because of memoization, stages 2(b) and 2(c) run at most once for each pair (i,j), and each such run makes at most two recursive calls, so there are at most $2(k+1)(t+1)+1 \leq 2(n+1)^2+1$ calls in total. Each call does polynomial work outside its recursive calls, and the recursion depth is at most $k+1$. So M_2 also decides *UNARY-SUBSET-SUM* in polynomial time. □

Recursion can be implemented on a TM by keeping a stack of pending calls on its tape. By model independence, we do not need to worry about these details.

Memoization is essential. Without stage 2(a), the recursion can make exponentially many calls. For example, if $x_1 = \cdots = x_k = 2$ and $t = 2k+1$, every call with $i \geq 1$ has $j \geq 2i+1 > x_i$, so it makes both recursive calls, for a total of $2^{k+1} - 1$ calls, while $n = \Theta(k)$.

The two approaches compute the same table, but organize the work differently. The bottom-up approach fills in all $(k+1)(t+1)$ entries in a fixed order, so we must choose an order in which every entry's dependencies come first (here, row by row). The top-down approach follows the recurrence directly, and the recursion determines the order automatically. It also only computes the subproblems that are actually reachable from $SS(k,t)$, which may be far fewer than all of them, at the cost of keeping a recursion stack. Either way, at most $(k+1)(t+1)$ subproblems are solved, which gives the same polynomial bound.

Note that both algorithms run in time polynomial in k and t, which is polynomial in the input length only because t is written in unary. If the numbers are written in *binary*, as in *SUBSET-SUM*, then t can be as large as $2^{\Omega(n)}$, and the table can have exponentially many entries. So these algorithms do *not* show that $SUBSET\text{-}SUM \in P$.

In fact, *SUBSET-SUM* is NP-complete, so a polynomial time algorithm for it would show that $P = NP$.