Recitation 08: P, NP, Dynamic Programming

This recitation covers some basic definitions and tools of complexity theory.

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 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

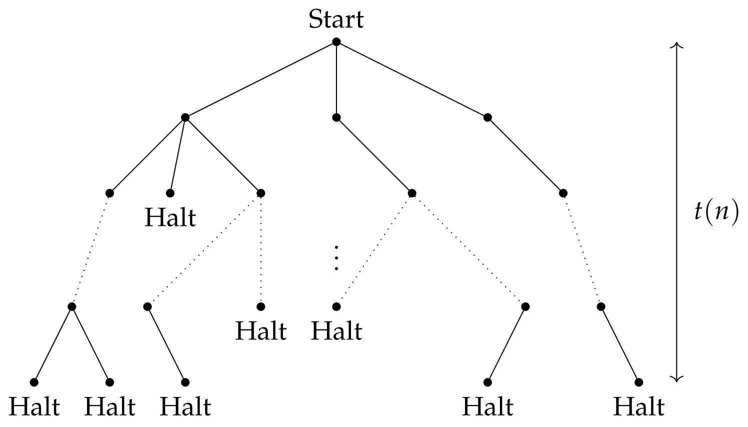
We define the analogous terms for nondeterministic TMs.

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 a graph $G \in HAMPATH$, the certificate would be the sequence of nodes corresponding to the Hamiltonian path in G. It is certainly polynomial time to check that each node in the sequence is in G, 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 $SUBSET\text{-}SUM$ 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 the computation history of N on x. We know that c is short since N runs in polynomial time.

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 that runs in time n^k for inputs of length n, for some constant k. For an input x of length n, since V accepts $\langle x,c\rangle$ for some certificate c in time n^k, we know that $|c|$ won't exceed n^k. We can thus build NTM N: on input x, nondeterministically guess certificate c of length at most n^k. 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 such that V accepts $\langle x,c\rangle$, so $L(N) = L$. All threads will halt in polynomial time since $|c|$ is polynomial, and V runs in polynomial time. □

A small detail on guessing: since a TM 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.

Dynamic programming

Dynamic programming is a tool to show that languages are in P. Sometimes, the brute-force algorithm for a problem is non-polynomial, but using dynamic programming, we can solve it in polynomial time. Dynamic programming is applicable when the problem has the following two properties:

- **Recursive:** The answer for a problem can be computed from answers of smaller subproblems that can be solved in the same way.
- **Memory:** The total number of different subproblems is polynomial.

The first property guarantees that dynamic programming works, and the second property guarantees that the runtime is polynomial. Dynamic programming is essentially a recursive algorithm that *memoizes* (remembers) the answers of solved subproblems, so each subproblem is solved only once. The recursive approach is referred to as the *top-down approach*.

A simple application of dynamic programming is in computing the n-th term of the Fibonacci sequence $(0, 1, 1, 2, 3, 5, ...)$. The n-th term $f(n)$ is equal to $f(n-1) + f(n-2)$, with $f(0) = 0, f(1) = 1$. How many steps would be required if the algorithm were purely recursive? How about if we use dynamic programming?

An often cleaner way to design dynamic programming solutions is the *bottom-up approach*. In this approach, subproblems are sorted in order of size, and each is computed one-by-one. Each time a subproblem is encountered, the answer can be directly computed, since the answers to the smaller subproblems have already been computed and stored.

The following example illustrates the bottom-up approach.

The top-down approach has the advantage that it doesn't compute the answers to subproblems that are not used, but its runtime is often harder to reason about. The bottom-up approach may do redundant work since it solves *all* subproblems.

Proposition 8. *The class P is closed under $*$. More precisely, $A \in P \implies A^* \in P$.*

Recall that $A^* = \{w \mid w = x_1 x_2 \cdots x_k, x_i \in A, k \geq 0\}$. The brute-force solution would be to try all possible splits of w into $x_1 x_2 \cdots x_k$, for all $k = 1, ...n$. However, this is not solvable in polynomial time, since there are exponentially many possible ways to split w. Instead, we use dynamic programming.

Proof idea. Let A be recognized by TM M in polynomial time. We want to construct TM N recognizing A^* in polynomial time.

On an input w of length n, checking whether $w \in A^*$ can be done recursively. More precisely, $w \in A^*$ if and only if at least one of the following two conditions holds:

- $w \in A$ or $w = \epsilon$
- There exists i, $1 \leq i < n$, such that $w_{[1...i]} \in A^*$ and $w_{[i+1...n]} \in A^*$

Let $w = w_1 w_2 \cdots w_n$. Here, we are using $w_{[l...r]}$ to denote substring $w_l w_{l+1} \cdots w_r$.

With the properties above, we can determine whether $w \in A^*$ from the answers to the subproblems for $w_{[1...i]}$ and $w_{[i+1...n]}$. Thus, the problem has the recursive property.

For the memory property, we want to bound the total number of different subproblems. Each subproblem will correspond to a substring

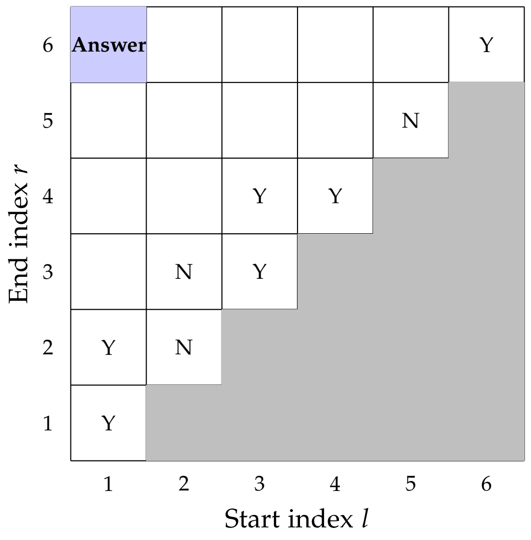
$w_{[l...r]}$. There are $O(n^2)$ ways to choose l and r, so there are polynomially many subproblems. To solve each subproblem, we try all possible values of i, for $1 \le i < n$, so we use $2(n-1) = O(n)$ subproblems to compute the answer. This shows that the total runtime $(O(n^2) \cdot O(n))$ is polynomial.

Proof. We will use the bottom-up approach of dynamic programming.
Build TM N recognizing A^* as follows:

- On input w, accept if $x = \epsilon$. Otherwise, consider the subproblems "is $w_{[l...r]} \in A^*$?" for all $1 \le l \le r < n$. Sort the $O(n^2)$ subproblems by length (i.e., $r - l + 1$).
- To solve the problem for $w_{l \cdots r}$, we first check if $w_{[l...r]} \in A$ and answer Y if so. Otherwise, we answer Y to $w_{l \cdots r}$ if $w_{[l...i]} \in A^*$ and $w_{[i+1...r]} \in A^*$ for any i, where $i = l, l+1, ..., r-1$. If not, answer N.
- If the answer for $w_{1 \dots n}$ (the largest and final subproblem) is Y, accept. Otherwise, reject.

See the proof idea above for the analysis of why N runs in polynomial time. We thus have $A^* \in P$. ☐

Consider a simple example, where $A = \{a, ab\}$, and $w = abaaba$. We have that $w \in A^*$ using the following split: $ab|a|ab|a$. The visual depiction below shows a snapshot of N's memory while following the dynamic programming solution detailed above.



Answer
Y
N
Y
Y
N
Y
Y
N
Y
End index r
Start index l
1
2
3
4
5
6