Recitation 08: Dynamic Programming, PSPACE

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.

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?

The first property guarantees that dynamic programming works, and the second property guarantees that the runtime is polynomial. Dynamic programming is essentially a recursion 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*.

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 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.

When presenting a dynamic programming solution it is important to specify:

- The definition of the sub-problem you are solving.
- The base case - what is the smallest subproblem that you can solve without relying on other subproblems to start?
- How to get the solution for a larger subproblem from the solution to the smaller subproblems.
- The subproblem that gives you the answer to the original problem.

The following example illustrates a dynamic programming solution with these components.

We define $UNARY - SSUM$ to be the language $SUBSET - SUM$ with the inputs given in unary, i.e.

$$UNARY - SSUM = \{\langle S, t\rangle | \text{ There exists some subset of } S \text{ that sums to } t \text{ where all inputs } S \text{ and } t \text{ are given in unary.}\}$$

Theorem 1. $UNARY - SSUM \in P$.

This theorem might come as a surprise, since we showed last recitation that $SUBSET - SUM$ without unary encoding is NP-Complete. It's a helpful exercise to revisit the proof from last recitation and figure out exactly what goes wrong with the unary encoding. In this case, the size of t as constructed by the reduction from last recitation will be $O(n+k)$ where there are n variables and k clauses. However, when we convert the input to be in unary, the size of t is equal to its value, which is exponential in terms of the input size, i.e. the size of t will be $O(10^{n+k})$ if we treat the numbers as being in base 10. Therefore, the reduction can no longer run in polynomial time, so it fails to show that $UNARY - SSUM$ is NP-Complete.

Proof idea. We want to construct a dynamic programming algorithm that will decide $UNARY - SSUM$ in polynomial time.

Let $S = \{x_1, x_2, \ldots, x_n\}$. Note that the brute-force approach here of trying all possible subsets will take non-polynomial time as there are $O(2^n)$ possible subsets of S to check.

Instead, we can split the problem into smaller subproblems by observing that for each element $x_i \in S$, we have a choice of whether or not to include x_i in our sum. If there is a subset of elements of S that sums to t, then there will either be a subset of elements of $S \backslash \{x_i\}$ that sum to $t - x_i$ or sum to t, where we include x_i in our sum in the former case and do not include it in the latter one. This establishes the recursive nature of the problem, since we reduce the problem into one with a smaller set and/or smaller target sum.

We can remember possible sums that we can create with elements in the subset $\{x_1, \ldots x_{i-1}\}$ for each i $1 \leq i \leq n$ and use these values to compute the possible sums involving x_i. Therefore, we can define our subproblem to be if there is a subset of elements $\{x_1, \ldots, x_i\}$ that sum to j for some $0 \leq j \leq t$. There will be a total of $O(nt)$ subproblems, which is polynomial in terms of the input size since t is expressed in unary.

Our final answer will be the subproblem corresponding to the subset $\{x_1, \ldots, x_n\}$ with target sum t. We start by building our table with small i and j and build from there using the recursive property we found.

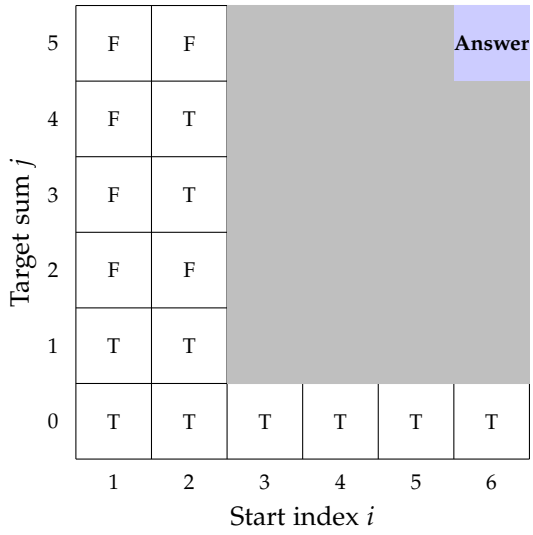
Proof. We will use the bottom-up approach of dynamic programming. Build TM M deciding $UNARY - SSUM$ as follows:

M: on input $\langle S, t \rangle$

1. Define the subproblem $\langle S_i, j \rangle$ to be the question of whether there is a subset of S_i that sums to j where $S_i = \{x_1, \ldots, x_i\}$ for all $1 \leq i \leq n$ and $0 \leq j \leq t$.
2. **Base Case**. For all subproblems $\langle S_i, 0 \rangle$, set the value equal to true (T), since we can always produce a sum of 0 from a subset of any length by taking the empty subset.
3. For each sub-problem $\langle S_i, j \rangle$, set its value equal to T if at least one of $\langle S_{i-1}, j$ and if $\langle S_{i-1}, j - x_i \rangle$ is true (where we only consider the second subproblem if $j - x_i \geq 0$). Otherwise, set it equal to false.
4. After solving all the subproblems, if $\langle S_n, t \rangle$ is true, accept. Otherwise, reject.

Time Complexity Analysis
We see that M runs in polynomial time because it builds a table of size $O(nt)$ to compute all the subproblems. Since t is expressed in unary, $O(nt) \in P$, so we only have a polynomial number of subproblems. For each subproblem, we only do polynomial work to look up two previous values and decide the current subproblem's answer based on those. Therefore, M will decide $UNARY - SSUM$ in polynomial time, so $UNARY - SSUM \in P$. □

Consider a simple example, where $S = \{1, 3, 4, 8, 11, 14\}$ and $t = 5$. We know the subset $\{1, 4\}$ will sum to 5, so we should accept. The visual depiction below shows a snapshot of M's memory while following the dynamic programming solution detailed above.



Space Complexity

So far in this class, we've focused on constructing Turing machines that run within a specific *time* bound. The dynamic programming approach we just saw is one technique where we traded space for time. Now we're adjusting our perspective and instead considering *space* bounds!

Define $SPACE(s(n))$ to be the set of languages decided by an $O(s(n))$ space *deterministic* Turing machine. Likewise, define $NSPACE(s(n))$ to be the set of languages decided by an $O(s(n))$ space *nondeterministic* Turing machine.

Definition 2.

$$PSPACE = \bigcup_k SPACE(n^k)$$
$$NPSPACE = \bigcup_k NSPACE(n^k)$$

We will look at a couple of examples of languages that are in $PSPACE$ whose time complexity does not necessarily fit neatly into P or NP.

MIN-FORMULA

We say that two Booolean formulas are *equivalent* if they have the same set of variables and evaluate to the same boolean for all possible sets of assignments to those variables. A Boolean formula is *minimal* if no shorter Boolean formula (i.e. a formula with fewer symbols) is equivalent to it. Using these defintions, we can define the language $MIN-FORMULA$ to be a collection of minimal boolean formulas.

Definition 3 (MIN-FORMULA)**.**

$$MIN - FORMULA = \{\langle \phi \rangle | \phi \text{ is a minimal boolean formula}\}.$$

For example, $x_1 \vee x_2$ is a minimal boolean formula since it cannot be simplified any further. However, $x_1 \vee (x_1 \wedge x_2)$ is not a minimal boolean formula since it can be simplified to x_1.

Next, we will show that $MIN - FORMULA$ is in *PSPACE*.

Theorem 4. *$MIN - FORMULA \in PSPACE$.*

Proof idea. The main idea is to iteratively try every single smaller formula and see if it is equivalent to ϕ. We test equivalence by looping through every single variable assignment and testing if the outputs are the same. If every single assignment produces the same result as ϕ, then we know that the smaller boolean formula is equivalent to ϕ, so ϕ is not minimal. If any one of these smaller formulas is equivalent to ϕ, then we know ϕ is not minimal.

Proof. We construct a TM M that decides MIN-FORMULA and uses polynomial space.

> M: On input $\langle \phi \rangle$:
> For every smaller formula ϕ':
>
> 1. For every possible assignment S of variables in ϕ:
>   i. Plug in S into ϕ and simplify until you get bool b (b will either be True or False).
>   ii. Plug in S into ϕ' and simplify until you get bool b' (b' will either be True or False).
>   iii. If $b = b'$ for every possible assignment S, then reject.

Space Complexity Analysis
On each iteration of the inner loop of trying each assignment, we simply make an assignment to each variable and plug into the formula. This substitution only takes polynomial space. After each iteration, we clear the tape so it can be reused. Therefore, we only ever use polynomial space, so $MIN - FORMULA \in PSPACE$. □

Let's see how $MIN - FORMULA$ relates to the P and NP time complexity class.

Theorem 5. *If $P = NP$, then MIN-FORMULA $\in P$.*

Proof. Note that if $P = NP$, then $coP = coNP$, but $coP = P$ (P is closed under complement), so the statement implies $P = NP = coNP$.

We define a language

$$EQUIV - FORMULA = \{\langle \phi, \phi' \rangle | \phi \text{ and } \phi' \text{ are equivalent formulas}\}.$$

We see that $\overline{EQUIV - FORMULA}$ is in NP, as we can take certificate to be a variable assignment in which the two formulas do not match, and this can be verified by plugging in the variables to each formula and comparing the resulting booleans to confirm they differ, which runs in polynomial time.

Hence, $EQUIV - FORMULA \in coNP$, so by our assumption that $P = NP$, $EQUIV - FORMULA \in P$. Let its polynomial time decider be TM M.

Then, we can define NTM N that decides $\overline{MIN - FORMULA}$, i.e. the language of non-minimal boolean formulas, in polynomial time to be the following:

> N: On input $\langle \phi \rangle$:
>
> 1. Nondeterministically guess a smaller formula ϕ'.
> 2. Run polynomial time decider for $EQUIV - FORMULA$ M on input $\langle \phi, \phi' \rangle$. If M accepts, then accept.

N will essentially "guess" a smaller formula and accept if it is equivalent to ϕ. If any one of the branches of the non-determinism accepts, then we will have found a smaller equivalent formula to ϕ, so it is not minimal, meaning $\langle \phi \rangle \in \overline{MIN - FORMULA}$.

Therefore, we have shown that $MIN - FORMULA \in coNP$. Since we have assumed $P = NP$ and shown this implies $P = NP = coNP$, this gives us that $MIN - FORMULA \in P$, as desired.

□

Note, while we are able to show that $MIN - FORMULA \in P$ if $P = NP$, it is actually not known if $MIN - FORMULA \in NP$ or $MIN - FORMULA \in coNP$

Why does our proof of Theorem 5 not imply that $MIN - FORMULA \in coNP$?

TQBF

A *fully quantified boolean formula* is a Boolean formula that bounds each variable with a quantifier (i.e. $\exists, \forall$). For example,

$$\forall x \exists y[(x \vee y)]$$

is a true quantified boolean formula, since for every value chosen for x, there exists the assignment for y such that the formula will be satisfied (namely $y = 1$).

What happens if the formula is $\forall x \exists y[(x \wedge y)]$ instead?

Definition 6. We define TQBF to be the langauge of true quantified boolean formulas, i.e.

$$TQBF = \{\langle \phi \rangle | \phi \text{ is a true fully quantified boolean formula}\}.$$

Theorem 7. *$TQBF \in PSPACE$.*

Proof. We can construct a recursive algorithm that runs in *PSPACE* that decides TQBF by trying all assignments. We construct TM M deciding TQBF that does the following:

> M: On input $\langle \phi \rangle$
>
> 1. **Base case:** If there is only one variable x_1, plug in both 0 and 1 and determine if the quantified boolean formula can be satisfied. Accept if it can be, and otherwise reject.
> 2. Otherwise, take the first variable x_1. Plug in $x_1 = 0$ and $x_1 = 1$ and simplify to get ϕ_0 and ϕ_1 respectively.
> 3. Run M on $\langle \phi_0 \rangle$ and $\langle \phi_1 \rangle$ and let the results be booleans b_0 and b_1 respectively.
> 4. If the quantifier for x_1 is $\forall$, return $b_0 \wedge b_1$, as we want the formula to be satisfied regardless of the value of x_1. If it is $\exists$, return $b_0 \vee b_1$, since only one of the two formulas is required to be true in the exists condition.

Space Complexity Analysis
We see that M uses polynomial space since for each recursive call, it only needs to plug in values for one variable, which can be done in polynomial space in terms of the input size. Then, after each recursive call, we clear the extra tape used to plug in the variable value so that it may be reused. Therefore, overall M decides TQBF in *PSPACE*, so $TQBF \in PSPACE$, as desired. □

Next, we look at how $TQBF$ is related to NP and $coNP$.

Theorem 8. *$TQBF$ is NP-Hard.*

A language B is NP-Hard if $A \leq_p B$ for every language in $A \in NP$. This is different from the notion of NP-completeness, because it does *not* imply that $B \in NP$. It is unknown if $TQBF \in NP$. We will see later that if $TQBF \in NP$, this would actually imply that $PSPACE = NP$.

Proof. We show that $3SAT \leq_p TQBF$. Note that asking if there is a satisfying assignment for ϕ is equivalent to the quantified boolean formula for ϕ with existence quantifiers for each variable. Therefore, we map ϕ to a quantified boolean formula ϕ' which has the same boolean formula as ϕ with existence quantifiers for each variable.

More formally, we define our mapping to be

$$f(\phi) = \exists x_1, \ldots, \exists x_n[\phi].$$

We show this mapping reduction is correct by showing $\phi \in 3SAT \Leftrightarrow f(\phi) \in TQBF$.

($\rightarrow$) If ϕ is satisfiable, then there is some assignment of $x_1, \ldots, x_n$ that makes ϕ true. Therefore, the quantified boolean formula $\exists x_1, \ldots, \exists x_n[\phi]$ is true, so $f(\phi) \in TQBF$.

($\leftarrow$) If $\exists x_1, \ldots, \exists x_n[\phi]$, there is some assigment of variables $x_1, \ldots, x_n$ that makes ϕ true, so it has a satisfying assignment. Therefore, $\phi \in 3SAT$.

Thus, the mapping reduction is correct. Furthermore, the reduction can be computed in polynomial time, since it just requires a linear scan of the formula and determining all the variables to add a there exists quantifier before them. □

Theorem 9. *TQBF is coNP-Hard.*

Analogously, a language B is coNP-Hard if $A \leq_p B$ for every language in $A \in coNP$. Or equivalently, $A \leq_p B$ for every language A such that $\overline{A} \in NP$. Again, B does not necessarily have to be in $coNP$, and similarly it is unknown if $TQBF \in coNP$ (and we will see later that this would similarly imply that $PSPACE = coNP$).

Proof. We show that $\overline{3SAT} \leq_p TQBF$. This is sufficient to show that $TQBF$ is $coNP$-Hard since $3SAT \in NP$ implies $\overline{3SAT} \in coNP$. Furthermore, since $3SAT$ is NP-complete, $A \leq_p 3SAT \Rightarrow \overline{A} \leq_p \overline{3SAT}$ for all $A \in NP$, which implies $\overline{3SAT}$ coNP-complete.

A formula does not have a satisfying assignment if and only if for all possible values of the variables the formula ϕ evaluates to false. This is equivalent to saying for all possible variable assignments, $\overline{\phi}$ is true. Therefore, we can map ϕ, an instance of $\overline{3SAT}$, to $\overline{\phi}$ with for all quantifiers for every the variables.

More formally, we define our mapping to be

$$f(\phi) = \forall x_1, \ldots, \forall x_n[\overline{\phi}].$$

We show this mapping reduction is correct.

($\rightarrow$) If ϕ is unsatisfiable, then for all $x_1, \ldots, x_n$, ϕ is false, so $\overline{\phi}$ is true for any assignment. Therefore, the quantified boolean formula $\forall x_1, \ldots, \forall x_n[\overline{\phi}]$ will be true, so $f(\phi) \in TQBF$.

($\leftarrow$) If $\forall x_1, \ldots, \forall x_n[\overline{\phi}]$ is true, then for every possible assignment of variables, $\overline{\phi}$ is true. This means that every possible assignment of variables results in ϕ being false, so there is no satisfying assignment and $\phi \in \overline{3SAT}$.

Therefore, the mapping reduction is valid. Furthermore, it can be computed by doing a linear scan of the input, as it requires scanning the formula and taking the complement and adding a quantifier for each variable. □