CS 38/138 (Introduction to Algorithms) **Spring 2015**

Recitation Notes - Dynamic Programming

William Hoza and Nicholas Schiefer Week 2 – April 8, 2015

1 A motivating example: longest subsequences

Problem 1 (Longest non-decreasing subsequence)**.** Given a sequence of n integers $(s_1, s_2, \ldots, x_n)$, find the longest *non-decreasing subsequence*. (i.e. find the longest sequence of indices $i_1 < \cdots < i_k$ such that $x_{i_1} \leq \cdots \leq x_{i_k}$.)

At first glance, this problem looks quite hard; there are 2^n subsequences to check, so a trivial "brute-force" algorithm would take exponential time (not good!). However, we can use a table to store some intermediate results and do much, much better.

Algorithm 1 Efficient algorithm for longest non-decreasing subsequence problem

```
ℓ[0] = 0, p[0] = ∅
for i = 1 to n do
    ℓ[i] = 1, p[i] = ∅
    for j = 1 to i − 1 do
        if s_i ≥ s_j then
            ℓ[i] = max(ℓ[i], ℓ[j] + 1)
            p[i] = j if ℓ[i] just changed
Find the index k that maximizes ℓ[k].
return The subsequence we get by following the pointers from p[k] backwards.
```

Proposition 1 (Correctness of algorithm 1)**.** After it is initialized, $\ell[i]$ is the length of the longest non-decreasing subsequence that ends with s_i, and $p[i]$ points to the "rest" of the longest non-decreasing subsequence.

Proof. The result trivially holds for $i = 0$. Observe that the longest non-decreasing subsequence that ends at position i is either:

1. The longest non-decreasing subsequence that ends at some position $j < i$, with s_i tacked on at the end. This works if $s_j \leq s_i$.
2. A new subsequence that starts at s_i.

Our algorithm fills in $\ell[i]$ by finding either the longest non-decreasing subsequence where we can "tack on" s_i or starts a new subsequence. It uses $p[i]$ to store the pointer to the "rest" of the list so it can be reconstructed later. Therefore, algorithm 1 is correct by induction. □

Proposition 2 (Runtime of algorithm 1)**.** algorithm 1 runs in $\Theta(n^2)$ time.

Proof. Each iteration of the inner loop takes constant time (assuming constant time comparisons/arithmetic, which is fine). The inner loop runs at most n times for each iteration of the outer loop, and the outer loop runs at most n times, so the total runtime is clearly $\mathcal{O}(n^2)$. (Why does it extend to $\Theta(n^2)$ time?) □

This technique, where we store the results of "smaller" computations to build up an answer to a larger computation is called **dynamic programming**.

2 An overview of dynamic programming

Unlike some techniques we will look at in this class, "dynamic programming" does not have a precise definition; it is a general class of techniques used for designing algorithms. Typically, dynamic programming is used for solving *optimization problems* where we are trying to find the "best" of a very large number of possible choices (like the longest non-decreasing subsequence problem above). However, **not all problems should/can be solved with dynamic programming**; it only works when certain properties are met:

Property 1 (Optimal substructure)**.** Dynamic programming requires *optimal substructure*: somehow, finding the optimal solution for a problem of size n is easy (or easier) once we have optimal solutions to problems of smaller sizes.

We often draw the "sub-problem tree" of a problem, showing which subproblems are directly required for each problem. If a problem exhibits optimal substructure, the problem has a naïve recursive algorithm that just recurses down the tree directly. Often, this leads to so-called *divide and conquer* algorithms, which are not the same as dynamic programming. Usually, we only call a solution "dynamic programming" when the problem has another property:

Property 2 (Overlapping subproblems)**.** A problem has *overlapping subproblems* if each sub-problem shares many (sub^n-problems) with other subproblems. Visually, this means that each sub^n-problem appears many times in the sub-problem tree.

In a sub-problem tree with many overlapping subproblems, our naïve recursive strategy is very wasteful: we compute solutions again and again and again. Even worse, these overlapping subproblems often have overlapping sub-subproblems and so on. What if we could just compute these once?

> In dynamic programming, we turn the sub-problem tree "upside down" and build-up a table of solutions to subproblems, starting with the smallest ones and using those to compute the solutions to the bigger ones.

This is usually called a "bottom up" approach, instead of the "top-down" recursive method. Dynamic programming usually works best when there are not too many *unique* subproblems, but many overlapping ones, so that we can save a lot of time by pre-computing our table. Typically, the runtime of a dynamic programming algorithm is:

$$\text{runtime} \approx (\text{amount of time to solve a subproblem}) \times (\#\text{ of subproblems})$$

Occasionally, we want to do a more precise analysis, but something like this is usually good enough.

2.1 Memoization: being lazy with class!

We sometimes call dynamic programming an *eager strategy*, since we anticipate the problems that we will need to solve later and solve them ahead of time (how very responsible!). However, we can also use another technique—called memoization—whenever dynamic programming works.

When we use memoizatin, we run our normal "top-down" recursive strategy, but we build-up a table of solutions to subproblems. Before solving a subproblem recursively, we check if we've already solved it; if we have, we use that result instead! Memoization is a *lazy strategy*, since we do everything just-in-time before we need it. However, it's still just as efficient, since we still don't repeat work that's already been done.

Memoization is nice because it is very, very easy to code up and can be faster if our bottom-up approach would have solved subproblems that never get used. On the other hand, its runtime is much harder to analyze (bad in theory!) and recursion incurs extra overhead on real computers (bad in practice!).

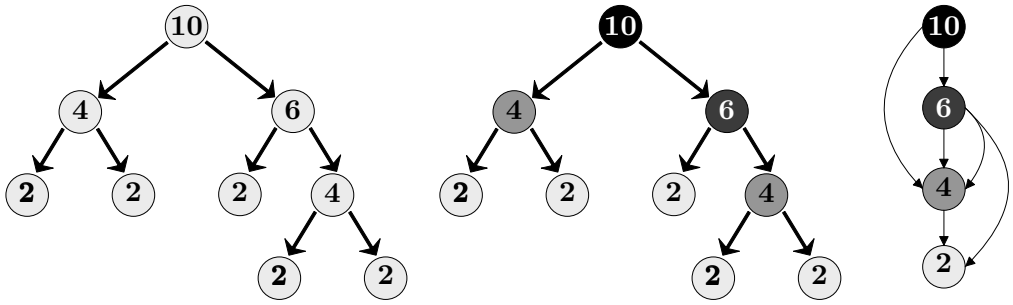

Figure 1: (a) Illustration of a subproblem tree. (b) The subproblem tree has many overlapping subproblems, which are highlighted in colours. (c) The collapsed subproblem graph. Notice that by computing "bottom-up," we can avoid recomputing values many times.

3 Strategy for solving problems with dynamic programming

Every problem is different, so there's no "magic formula" that you can use to solve all of your problems. However, there are some common strategies that are usually applicable when solving problems with dynamic programming:

1. **Identify optimal substructure:** When you look at a problem, try to see if you can compute "bigger" instances use the answers to "smaller" instances. For example, the longest non-decreasing subsequence that ends in position i was given in proposition 1 in terms of the longest non-decreasing subsequences that ended in positions $1, \ldots, i-1$.

2. **Identify overlapping subproblems:** Look at the sub-problem tree and look for repeated subproblems. If you problem is well-suited for dynamic programming, it will usually have many repeated subproblems and not too many unique subproblems. It's often helpful to collapse the sub-problem tree into a (directed, acyclic) *subproblem graph* to visualize this structure. In the longest non-decreasing subsequence problem, we need to know $\ell[3]$ to compute $\ell[4], \ell[5], \ldots, \ell[n]$, for example.

3. **Turn the recursive strategy "upside-down" to get a "bottom-up" strategy:** Your collapsed sub-problem graph should have a node at the bottom that can be solved on its own. Starting there, try to reverse the recursive strategy to build up a solution from the bottom. It is often helpful to store the results in a table. For example, in algorithm 1, we started with a really easy problem (what's the longest subsequence starting at the very beginning?) and worked our way up. We used ℓ and p as tables to store the previous results.

4. **Build up the solution:** Often, dynamic programming does not directly give us the answer we want; we need to build it up from many partial solutions. For example, in algorithm 1, our main computation was just the *length* of the longest non-decreasing subsequence, and not the sequence itself. We stored the pointer table p to reconstruct the solution at the very end.

4 Dynamic programming on graphs

Although dynamic programming solutions often use straightforward (and maybe boring) rectangular tables to store the intermediate results, many interesting problem can be solved using dynamic programming in non-tabular contexts. Here's a straightforward example:

Problem 2 (Reachability on directed graphs)**.** Given a directed graph $G = (V, E)$ and vertices s, t, determine whether there is a (directed) path from s to t.

This problem is quite basic; you might be able to come up with a solution without thinking about fancy techniques like "dynamic programming." In fact, it might not look like dynamic programming is applicable at all, since there's no clear "tabular" solution. However, the framework we developed can help us find a solution in a more systematic way:

1. **Identify optimal substructure:** Consider a vertex t' that is reachable from s. If (t', t) is an edge, then there is a path from s to t. Thus, we can solve our larger problem by looking at a smaller subproblem. More generally, we have a path from s to v if and only if there is some in-neighbour w of v such that there is a path from s to w.

2. **Identify overlapping subproblems:** Consider two vertices a and b that share a common in-neighbour w. Solving the reachability problem for a and b involves solving the common sub-problem: "is w reachable from s?" In fact, our reduced subproblem graph is just the original graph G but with all edges reversed.

3. **Turn the recursive strategy "upside-down" to get a "bottom-up" strategy:** Our strategy is to start at s, and "propogate" the reachability outwards. Instaed of maintainng a table, we maintain a property ("reachable") at each vertex, that is iniitally FALSE and can get updated to be TRUE whenever we find a path to the vertex. Specifically, we maintain a set of vertices S under consideration; initially $S = \{s\}$. Repeat the following until either t is marked or S is empty: Mark all the vertices in S. Update S to be the set of unmarked out-neighbors of vertices in S. At the end, the set of marked vertices is precisely the set of vertices reachable from s.

4. **Build up the solution:** This is a decision problem, so we don't need to build up a solution except to check whether t is marked.

5 Solving NP-complete problems: partition

Although we usually use dynamic programming to make problems tractable (in the sense of solving them in polynomial time, instead of exponential time), we can also use dynamic programming to solve problems that are not believed to have polynomial time solutions. This is really important, since many NP-complete problems (which do not have polynomial-time solutions unless $\mathbf{P} = \mathbf{NP}$) are actually very useful.

Problem 3. (PARTITION) Given an array of positive integers $x_1, \ldots, x_n$, determine whether the array can be split into two subarrays (not necessarily contiguous) with equal sums. That is, is there an $I \subseteq \{1, 2, \ldots, n\}$ such that $\sum i \in I x_i = \sum_{i \notin I} x_i$.

In CS21, you leaned that PARTITION is **NP**-complete. A brute force solution (checking all subsets) involves $\Omega(2^n)$ arithmetic operations, and unless $\mathbf{P} = \mathbf{NP}$, there is no polynomial-time algorithm for solving PARTITION. However, there is a clever pseudo-polynomial time algorithm that works well if all of the x_i are small (even if there are a lot of them); naturally, this algorithm will use dynamic programming, much like the algorithm for KNAPSACK in Lecture 1.

For convenience, we define $N = \sum_i x_i$; we want to know if there is a subsequence of the x_i that sum to $N/2$. If N is odd, the answer is obviously no, but we still need to solve the even case. Let's try our usual strategy:

1. **Identify optimal substructure:** Define $P(S, i)$ to be a boolean function that answers the question: "is there a subset $I \subseteq \{1, \ldots, i\}$ such that $\sum_{i \in I} = S$. The PARTITION problem is equivalent to the question: "what is $P(N/2, n)$?"

   Observe that $P(S, i) =$ TRUE if and only if there is some largest element of I. If this element is x_i, then $P(S - x_i, i - 1)$ must be true. If this element is not x_i, then $P(S, i - 1) =$ TRUE. Thus, $P(S, i - 1)$ if and only if $P(S, i - 1)$ or $P(S - x_i, i - 1)$ is true. This gives us optimal substructure.

2. **Identify overlapping subproblems:** The overlapping substructure is quite clear, if a bit difficult to state precisely. Clearly, we will end up using the values $P(S, i-1)$ again and again.

3. **Turn the recursive strategy "upside-down" to get a "bottom-up" strategy:** Since we want to compute $P(N/2, n)$, we can create a table Q that stores values of P. Specifically, Q is a $(N/2+1)\times(n+1)$ 2D table of booleans. Our algorithm is then quite clear:

Algorithm 2 A pseudo-polynomial time solution to PARTITION.

initialize $Q(0, x) =$ TRUE for all $0 \leq x \leq n+1$.
initialize $Q(x, 0) =$ FALSE for all $1 \leq x \leq N/2$.
for $S = 0$ to $N/2$ **do**
  for $i = 0$ to n **do**
    $Q[S, i]] = Q[S, i-1] \vee Q[S - x_i, i]$
return $Q[N/2, n]$

4. **Build up the solution:** Our solution is just a boolean value, so we don't need to build anything up at the end.

Proposition 3 (Correctness of algorithm 2)**.** algorithm 2 returns TRUE if and only if there is a valid partition of the input.

Proof sketch. Our algorithm is correct because it directly implements the optimal substructure we obtain, with correct initial conditions. □

Proposition 4 (Runtime of algorithm 2)**.** algorithm 2 runs in $\mathcal{O}(nN)$ time.

Proof. The initialization steps take $\mathcal{O}(n)$ and $\mathcal{O}(N)$ time, respectively. The operation inside the loops takes constant time (it's just an array access) and it runs for a total of $\mathcal{O}(nN)$ times. Therefore, the total runtime is $\mathcal{O}(nN)$. □

Notice that $\mathcal{O}(nN)$ is not polynomial in the input size, since N can be encoded in binary using only $\mathcal{O}(\log N)$ symbols. For example, if $n = 1$, then N is exponentially large as a function of the input size. We call this kind of runtime "pseudo-polynomial:" when all of the x_i are small, this algorithm actually runs quite quickly.

The PARTITION problem is fascinating because it is very hard (**NP**-complete, in fact), but is nonetheless practical to solve in many cases. For that reason, many have called PARTITION "the easiest hard problem."