Lecture Notes: Dynamic programming

We will discuss the subset sum problem, and introduce the main idea of dynamic programming. We illustrate it further via the problem of finding short paths in directed graphs, as well as a variant of the so-called *knapsack* problem.

Subset sum problem and dynamic programming

We start by recalling the problem defined in the previous lecture:

Subset Sum: Given a set of non-negative integers $A[0], A[1], ..., A[n-1]$ and an integer S, find if there is a subset of $A[i]$'s whose sum is S.

As we discussed, the trivial algorithm is to consider all possible subsets of $A[0], A[1], ..., A[n-1]$ and see if any of the subsets sum to S. This algorithm clearly takes $O(2^n)$ since there are 2^n possible subsets.

How about divide and conquer? One idea is to break up the array $A[]$ into two halves, obtain one "part" of the sum S from the first half, and the rest from the second half. Formally, let $\mathcal{A} = \{A[0], A[1], ..., A[n-1]\}$, and let us denote the first half by $\mathcal{A}_1$, and the second half by $\mathcal{A}_2$. For each t in $[0, S]$, we could check if there is a subset of $\mathcal{A}_1$ that sums up to t AND a subset of $\mathcal{A}_2$ that sums to $S - t$. If this is true for some t, then there is a subset of $\mathcal{A}$ that sums to S (and vice versa, because $A[i]$ are all non-negative integers). How long does this procedure take?

As there are $2(S+1)$ sub-problems we are solving of size $n/2$, we have the recurrence:

$$T(n) = 2(S+1)T(n/2) \implies T(n) = (2(S+1))^{\log_2 n} = n^{\log_2 2(S+1)} = n^{1+\log_2(S+1)} \simeq n^{1+\log_2 S}.$$

This is much better than 2^n if S is relatively small (to be precise, anything smaller than $\exp(n/\log n)$).

Now lets try another recursive formulation. Suppose there is a set, $\{i_1, ...i_k\}$, which $S = A[i_1] + ... + A[i_k]$. If $A[0]$ is used in this sum, we only need to check if we can make a sum of $S - A[0]$ using $A[1], A[2], ...A[n-1]$, otherwise we would need to write S using $A[1], A[2], ...A[n-1]$.

Of course, if we write this as a recursive procedure, a little thought reveals that the tree produced contains all the possible subsets (it's a complete binary tree, with depth n — such trees are called *decision trees*, as each step involves deciding if an $A[j]$ is used or not).

However, the main observation is that when S isn't too large, we have many recursive calls that are *identical*. To illustrate, consider the example in Figure 1. The branches obtained by (a) including 1 and 2 and excluding 3 and (b) excluding 1 and 2 and including 3, both result in a recursive call in which we have the set $\{5, 7, \dots, \}$, and we try to make a sum of 17.

Indeed, we *must* have many repetitions in this process, because every instance basically has a suffix of the original array, and we wish to make a sum that lies in the interval $[0, 20]$. So the total number of difference instances we can have is 8×21, and this is $< 2^8$, which is the number of nodes in the tree.

The main idea in dynamic programming is to ask, can we avoid recomputing the answer to the same instance by *storing the answers*? This way, whenever we want to make a recursive call, we can check if the answer to that call has already been obtained, and proceed with the call only if it has not. The question is thus (a) how do we characterize the "sub-instances", and (b) how much storage do we need?

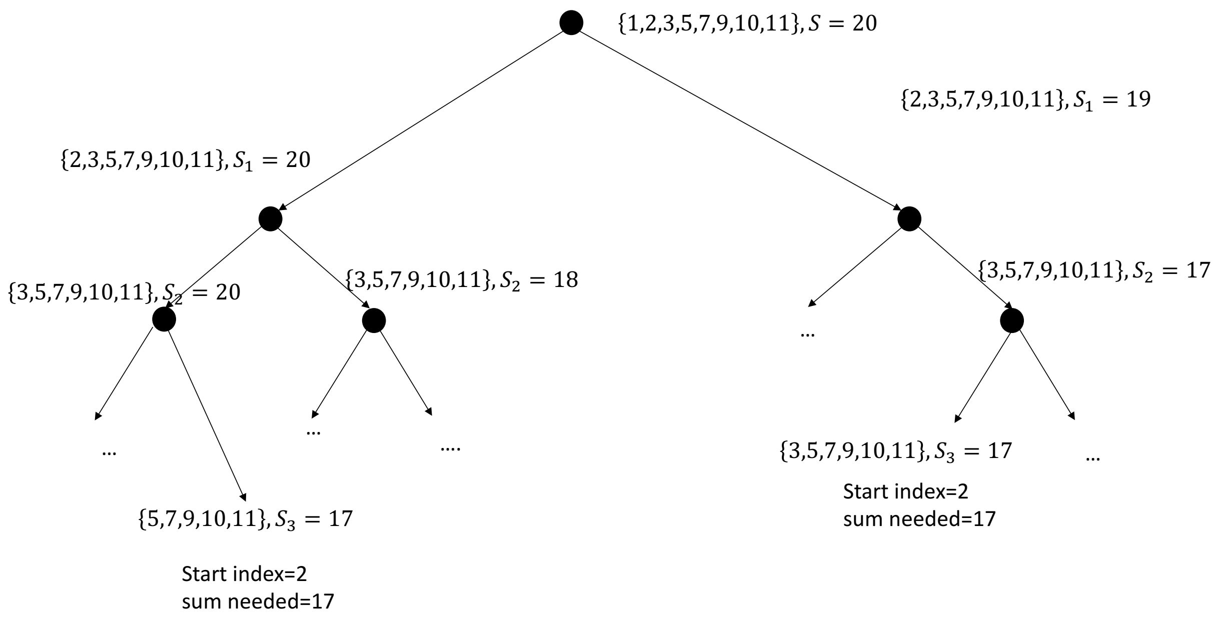
{1,2,3,5,7,9,10,11}, $S = 20$
{2,3,5,7,9,10,11}, $S_1 = 20$
{2,3,5,7,9,10,11}, $S_1 = 19$
{3,5,7,9,10,11}, $S_2 = 20$
{3,5,7,9,10,11}, $S_2 = 18$
{3,5,7,9,10,11}, $S_2 = 17$
...
....
{5,7,9,10,11}, $S_3 = 17$
Start index=2
sum needed=17
{3,5,7,9,10,11}, $S_3 = 17$
Start index=2
sum needed=17

Figure 1: Example of using second recursive call on the subset sum problem, as you can see, different branches can have the same instance, i.e., same problem

We have answered both the questions in our example. The sub-instances essentially have a suffix of the original array, and a target sum that lies in the interval $[0, S]$. We can thus store the answers based on two parameters: the starting index in the array, and the value of the sum. The starting index can range between 0 and $n-1$, and the sum has $(S+1)$ different values. Thus the total number of instances is $n(S+1)$.

Note that depending on n and S, this can be significantly smaller than 2^n. In summary, we have the modified algorithm, whose pseudocode we show below.

```
//  Procedure takes as input the starting index j for the suffix of A, and the target sum T <= S

Initialize array answer[n][S+1]
Initialize array isComputed[n][S+1] to false

procedure SubsetSum (j, T) {
   if (isComputed[j][T]) return answer[j][T];
   else {
      if (SubsetSum(j+1, T) || SubsetSum(j+1, T-A[j]))
         answer[j][T] = true;
      else answer[j][T] = false;
      isComputed[j][T] = true;
      return answer[j][T];
   }
}
```

[We have omitted two extreme cases in the above procedure. First, one should have the "base case" of $j = n-1$. Second, one should check if $T - A[j] \geq 0$ when making the recursive call.] The analysis above gives a running time and space complexity of $O(nS)$.

Is this polynomial time? The answer is obviously polynomial in n and S, but typically, when we talk of polynomial time, we wish to have the running time be polynomial in the *input size* of the problem. This is defined to be the number of *bits* needed to write down the input. In this case, the number of bits is

$$\lceil \log_2 A[0] \rceil + \lceil \log_2 A[1] \rceil + ... + \lceil \log_2 A[n-1] \rceil + \lceil \log_2 S \rceil.$$

It turns out that the quantity nS is not necessarily polynomial in this quantity. To see an explicit example, consider an instance where $A[i] \in [2^{n-1}, 2^n]$, and $S \in [2^n, n \cdot 2^n]$. The input size in this case (i.e. the expression above) would be approximately n^2, while the nS value would be $\geq n \cdot 2^n$

Key idea: sub-problems. To summarize, the main "trick" in the procedure above was realizing that the exact same sub-problems were being solved in many nodes of the tree. Indeed, at any level of the tree, there are at most S distinct sub-problems. Thus, in the rth level, if $S \ll 2^r$, we get a significant saving.

Shortest paths on directed graphs

Another one of the initial applications of dynamic programming is the so-called "Bellman-Ford" algorithm[1] for the problem of computing shortest paths in graphs.

Let G be a directed graph with vertex set V (of size n) and edge set E (of size m, which consists of ordered pairs (i, j)). We are also given a length $\ell(e)$ for every edge e. The problem is now the following: given vertices u, v and an integer L satisfying $1 \leq L \leq n$, find a path in G that contains precisely L "hops" and has the minimum total length. The number of hops is the total number of distinct edges in the path. [For example, $(u, w), (w, v)$ is a two-hop path.]

Trivial solution. A solution that enumerates all paths, of course, does not work in polynomial time, as there could be exponentially many paths between two given vertices.

A simple recursive procedure also suggests itself:

```
procedure ShortestPath (u, v, L) {
  if (L = 0):  // base case
    if (u = v) return 0; else return INFTY;
  for all w such that (w, v) is an edge, compute:
    val = ShortestPath(u, w, L-1) + length(w, v);
  return the least 'val' encountered in the loop above;
}
```

As such, the algorithm basically enumerates all the paths from u to v. However, the key observation now is that all the recursive calls are of the form (u, w, L') for some parameter $L' \in [0, L-1]$. Indeed, the only variants in the different calls are the vertex w and the intended length L'. There are n choices for w, and L choices for L'. Thus the total number of distinct recursive calls is $\leq n \cdot L$.

We can thus store all the answers in an array of size $n \times L$. Each time, before making a call, we look-up the corresponding value in the array. If it has not yet been computed, we go ahead with the recursive call (and at the end of the call, store the value in the array). If it has been computed, we simply look up.

Correctness. The proof, as with all recursive algorithms, proceeds by induction. However, we additionally observe that each time the answer to a recursive call is *written* to the array, the answer is correct. This

[1]While the algorithm is popularly known by this name, earlier works had already discovered the same algorithm, while not calling it dynamic programming.

ensures that for the purpose of analysis, the algorithm is precisely the path-enumeration procedure above!

Running time. We note that all computations happen within some recursive call (or the initial function call ShortestPath(u, v, L)). The call (u, w, L') takes time equal to the in-degree(w) plus the cost of recursive calls. These will either be constant (in case the value was already computed), or will be accounted for when we consider the time for the corresponding recursive call. Thus, the overall running time can be written as

$$\sum_{\text{recursive calls}(u,w,L')} O(\text{in-degree}(w)) \le \sum_{0 \le L' \le L} \sum_{w \in V} O(\text{in-degree}(w)) = (L+1) \sum_{w \in V} O(\text{in-degree}(w)) = O(Lm).$$

In the last equality, we used the fact that the indegrees sum up to m, the total number of edges in the graph.

Since we said that $L \le n$, the run time is $O(nm)$. Note that the space complexity is $O(Ln)$. This can be as bad as n^2. Space is a general concern in dynamic programming, and we will discuss this briefly later.

Coin change

Our next example is a simple case of what is called the *knapsack problem*. [We did not cover this in class. You may treat this as bonus material.]

Coin change problem. Suppose we are given a set of coins of denominations $d_0, d_1, ...d_{n-1}$ cents. Given a target S, make change for S cents using the fewest number of coins.

Of course, we assume that d_i are all integers, and suppose that they are in increasing order. As an example, suppose that the denominations are $1c, 5c, 10c, 20c, 25c, 50c$ and S is equal to \$1.90. One heuristic method would be to always choose the largest coin that you can use. If so, the selected coins would be $50c, 50c, 50c, 25c, 10c, 5c$ although the optimal solution would be $50c, 50c, 50c, 20c, 20c$. Thus this *greedy* heuristic (greedy in the sense of making as much progress as possible — we will see a better definition in the next lecture) does not always produce the right answer.

A recursive algorithm

Our first attempt will be to formulate a recursive algorithm, that reduces the number of denominations by 1 in each recursive call. Let us consider the denomination d_0. In the optimal way of making change for S cents, it could be used either 0 times, or 1 time, up to $\lfloor S/d_0 \rfloor$ times. We can thus write the following recursive formulation. Let $OPT(i, S)$ be the optimal number of coins to make change for S, using only the coins $\{d_i, d_{i+1}, \dots, d_{n-1}\}$. We define $OPT(i, S) = \infty$ if there is no way of making change for S using the given coins. We then have the recursive formulation:

$$OPT(i, S) = \min_{0 \le j \le \lfloor S/d_i \rfloor} j + OPT(i+1, S - jd_i).$$

To see why this is true, assume that we used j coins of denomination d_i in the optimal way of making change. Then the number of coins used is j plus the optimal number of coins used in making change for $S - jd_i$ using only the coins $\{d_{i+1}, \dots, d_{n-1}\}$. Since we want the minimum number of coins, we look a the minimum over all j. The "base case" here is $i = n-1$ (in which case we simply see if S is a multiple of d_{n-1} and if so, the OPT() value is S/d_{n-1} and if not, the OPT() value is ∞).

Now, we can ask the same question as we did for the subset sum problem. How many different recursive calls are these? As in subset sum, there are at most $n \cdot S$ different sub-problems possible. The slight difference is that the call $OPT(i, S)$ now calls multiple recursive calls (not just two) and takes the minimum. But we can always bound the number of recursive calls by $\lfloor S/d_0 \rfloor$ (as d_0 is the smallest denomination).

This gives an upper bound of $O\left(nS \cdot \frac{S}{d_0}\right)$ on the running time.

Improving the space and time complexities slightly

We note that for the coin change problem, one can formulate a slightly different recursive algorithm that is slightly better than the above.

Suppose we denote by $opt(S)$ the minimum number of coins needed to make change for S (here there is no restriction on the coins used). Again, we define the quantity to be ∞ if there is no way to make change for S. Then, if we are to come up with a recursive definition for $opt(S)$, we can observe that

$$opt(S) = 1 + \min_{i\,:\,d_i \le S} opt(S - d_i).$$

The base case for this will be $\text{OPT}(0) = 0$. (Also, if $S \neq 0$ and $S < \min_i d_i$, we have $opt(S) = \infty$.) This is a simpler recurrence. It holds because any valid way of making change for S –when $S \neq 0$– must use some coin whose denomination is $\le S$. We are trying all the possible coins.

The analysis of this procedure is simpler than the above. We have a total of $S+1$ possible recursive calls (corresponding to values of $\le S$). Thus storing all the values uses space $O(S)$. The time complexity is also better than the earlier bound. For computing the value of $opt(S)$, we will in the worst case check all i, and check $opt(S - d_i)$. Thus the time can be bounded by $O(n)$. This gives an overall run time of $O(Sn)$.

Bottom-up vs Top-down in Dynamic programming

The way I have described the procedures above is Top-down. Think of the tree in the figure above. We have a recursive procedure for computing a value at the root (top) node. Each time we make the recursive call, we check if the answer to that sub-tree has been computed already, and we decide to go further down the tree only if the answer has not been computed so far.

Now, a different way to solve the problem is to start populating the values of the different sub-problems "from below". In the subset sum example, this would mean that we first fill out the values of `answer`$[n-1][T]$ for all values of $T \in [0, S]$. Then we compute the values of `answer`$[n-2][T]$ for all values of $T \in [0, S]$, and so on.

In the case of the coin change problem, this would lead to a solution in which we first fill in the entry $opt(0)$, then we compute $opt(1)$ (using the description above), then we find $opt(2)$, and so on.

Positives of bottom-up. Filling in the values "from below" has one main advantage – we don't need an array that tells us if the value has been computed before or not! We know that we always computed the value. So for instance, it leads to the following simple algorithm for the coin change problem.

```
initialize array answer[S+1];
set answer[0] = 0;
for s = 1, ..., S do:
  if (s < min_i d_i) answer[s] = infinity;
  else answer[s] = 1 + min_i answer[s - d_i] // minimum over i s.t. d_i <= s
```

This algorithm is arguably simpler than checking if we have computed the answers. However, it could be more inefficient.

Positives of top-down. The first advantage of the top-down approach is that it is conceptually how dynamic programming was invented (observing that common solutions are being recomputed in a recursive procedure and storing them). The second is that quite often, this may be much more efficient. Note that a whole bunch of sub-problems may never actually come up in the tree shown in the figure above. As a trivial example, in the subset sum problem, if all the $A[i]$ are even and the S is even, no odd value of S will ever show up as a candidate, while the bottom-up approach will try to compute those values as well.

Overall, it is a matter of taste. I prefer the top-down approach because of the first advantage I described above.