Lecture 24 — Dynamic Programming I

Parallel and Sequential Data Structures and Algorithms, 15-210 (Fall 2012)

Lectured by Guy Blelloch — 15 November 2012

Today:

- Introduction to Dynamic Programming
- The subset sum problem
- The minimum edit distance problem

1 Dynamic Programming

> "An interesting question is, 'Where did the name, dynamic programming, come from?' The 1950s were not good years for mathematical research. We had a very interesting gentleman in Washington named Wilson. He was Secretary of Defense, and he actually had a pathological fear and hatred of the word, research. I'm not using the term lightly; I'm using it precisely. His face would suffuse, he would turn red, and he would get violent if people used the term, research, in his presence. You can imagine how he felt, then, about the term, mathematical. The RAND Corporation was employed by the Air Force, and the Air Force had Wilson as its boss, essentially. Hence, I felt I had to do something to shield Wilson and the Air Force from the fact that I was really doing mathematics inside the RAND Corporation. What title, what name, could I choose? In the first place I was interested in planning, in decision making, in thinking. But planning, is not a good word for various reasons. I decided therefore to use the word, 'programming.' I wanted to get across the idea that this was dynamic, this was multistage, this was time-varying—I thought, let's kill two birds with one stone. Let's take a word that has an absolutely precise meaning, namely dynamic, in the classical physical sense. It also has a very interesting property as an adjective, and that is it's impossible to use the word, dynamic, in a pejorative sense. Try thinking of some combination that will possibly give it a pejorative meaning. It's impossible. This, I thought dynamic programming was a good name. It was something not even a Congressman could object to. So I used it as an umbrella for my activities".
> Richard Bellman ("Eye of the Hurricane: An autobiography", World Scientific, 1984)

The Bellman-Ford shortest path algorithm we have covered is named after Richard Bellman and Lester Ford. In fact that algorithm can be viewed as a dynamic program. Although the quote is an interesting bit of history it does not tell us much about dynamic programming. But perhaps the quote will make you feel better about the fact that the term has little intuitive meaning.

In this course, as commonly used in computer science, we will use the term dynamic programming to mean an algorithmic technique in which (1) one constructs the solution of a larger problem instance

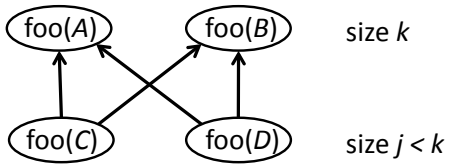
†Lecture notes by Guy E Blelloch, Margaret Reid-Miller, and Kanat Tangwongsan.

by composing solutions to smaller instances, and (2) the solution to each smaller instance can be used in multiple larger instances. For example, in the Bellman-Ford algorithm, to find the shortest path length from the source to vertex v that uses at most i vertices, depends on finding the shortest path lengths to the in-neighbors of v that use at most $i-1$ vertices. As vertices may share in-neighbors, these smaller instances maybe used more then once. Dynamic programming is thus one of the inductive algorithmic techniques we are covering in this course.

Recall from Lecture 1 that in all the inductive techniques an algorithm relies on putting together smaller parts to create a larger solution. The correctness then follows by induction on problem size. The beauty of such techniques is that the proof of correctness parallels the algorithmic structure.

So far the inductive approaches we have covered are divide-and-conquer, the greedy method, and contraction. In the greedy method and contraction each instance makes use of only a single smaller instance. In the case of greedy algorithms the single instance was one smaller—e.g. Dijkstra's algorithm that removes the vertex closest to the set of vertices with known shortest paths and adds it to this set. In the case of contraction it is typically a constant fraction smaller—e.g. solving the scan problem by solving an instance of half the size, or graph connectivity by contracting the graph by a constant fraction.

In the case of divide-and-conquer, as with dynamic programming, we made use of multiple smaller instances to solve a single larger instance. However in divide-and-conquer we have always assumed the solutions are solved independently and hence we have simply added up the work of each of the recursive calls to get the total work. But what if two instances of size k, for example, both need the solution to the same instance of size $j < k$?



Although sharing the results in this simple example will only make at most a factor of two difference in work, in general sharing the results of subproblems can make an exponential difference in the work performed. The simplest, albeit not particularly useful, example is in calculating the Fibonacci numbers. As you have likely seen, one can easily write the recursive algorithm for Fibonacci:

```
1  fun fib(n) =
2    if (n ≤ 1) then 1
3    else fib(n − 1) + fib(n − 2)
```

But this recursive version will take exponential work in n. If the results from the instances are somehow shared, however, then the algorithm only requires linear work, as illustrated in Figure 1. It turns out there are many quite practical problems where sharing results of subinstances is useful and can make a significant differences in the work used to solve a problem. We will go through several of these examples.

With divide-and-conquer the composition of a problem instance in terms of smaller instances is typically described as a tree, and in particular the so called recursion tree. With dynamic programming

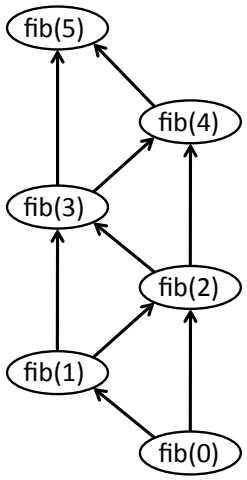


Figure 1: The DAG for calculating the Fibonacci numbers.

the composition can instead be viewed as a Directed Acyclic Graph (DAG). Each vertex in the DAG corresponds to a problem instance and each edge goes from an instance of size j to one of size $k > j$—i.e. we direct the edges (arcs) from smaller instances to the larger ones that use them. We direct them this way since the edges can be viewed as representing dependences between the source and destination (i.e. the source has to be calculated before the destination can be). The leaves of this DAG (i.e. vertices with no in-edges) are the base cases of our induction (instances that can be solved directly), and the root of the DAG (the vertex with no out-edges) is the instance we are trying to solve. More generally we might actually have multiple roots if we want to solve multiple instances.

Abstractly dynamic programming can therefore be best viewed as evaluating a DAG by propagating values from the leaves to the root and performing some calculation at each vertex based on the values of its in-neighbors. Based on this view calculating the work and span of a dynamic program is relatively straightforward. We can associate with each vertex a work and span required for that vertex. The overall work is then simply the sum of the work across the vertices. The overall span is the longest path in the DAG where the path length is the sum of the spans of the vertices along that path. For example consider the DAG shown in Figure 2. This DAG does $5 + 11 + 3 + 2 + 4 + 1 = 26$ units of work and has a span of $1+2+3+1 = 7$. Many dynamic programs have significant parallelism although some do not.

The challenging part of developing an algorithm for a problem based on dynamic programming is figuring out what DAG to use. The best way to do this, of course, is to think inductively—how can I solve an instance of a problem by composing the solutions to smaller instances? Once an inductive solution is formulated you can think about whether the solutions can be shared and how much savings can be achieved by sharing. As with all algorithmic techniques, being able to come up with solutions takes practice.

It turns out that most problems that can be tackled with dynamic programming solutions are optimization or decision problems. An *optimization problem* is one in which we are trying to find a solution that optimizes some criteria (e.g. finding a shortest path, or finding the longest contiguous subsequence sum). Sometimes we want to enumerate (list) all optimal solutions, or count the number

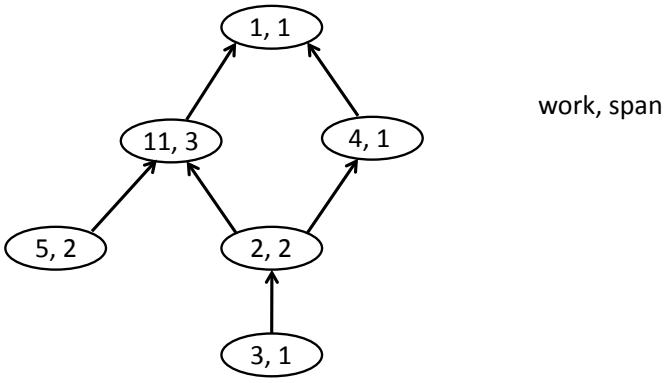


Figure 2: Work and span for a dynamic programming example. The work for each vertex is on the left and the span on the right. The total work is 26 and the span is 7.

of such solutions. A *decision problem* is one in which we are trying to find if a solution to a problem exists. Again we might want to count or enumerate the valid solutions. Therefore when you see an optimization or enumeration problem you should think about possible dynamic programming solutions.

Although dynamic programming can be viewed abstractly as a DAG, in practice we need to implement (code) the dynamic program. There are two common ways to do this, which are referred to as the top-down and bottom-up approaches. The *top-down* approach starts at the root and uses recursion, as in divide-and-conquer, but remembers solutions to subproblems so that when the algorithm needs to solve the same instance many times only the first call does the work and the remaining calls just look up the solution. Storing solutions for reuse is called *memoization*. The *bottom-up* approach starts at the leaves of the DAG and typically processes the DAG in some form of level order traversal—for example, by processing all problems of size 1 and then 2 and then 3, and so on. Each approach has its advantages and disadvantages. Using the top-down approach (recursion with memoization) can be quite elegant and can be more efficient in certain situations by evaluating only those instances actually needed. Using the bottom up approach (level order traversal of the DAG) assumes it will need every instance whether or not is it use in the overall solution, but can be easier to parallelize and can be more space efficient. *It is important, however, to remember to first formulate the problem abstractly in terms of the inductive structure, then think about it in terms of how substructure is shared in a DAG, and only then worry about coding strategies.*

1.1 Subset Sums

The first problem we will cover in this lecture is a decision problem, the subset sum problem:

Definition 1.1. The *subset sum* (SS) problem is, given a multiset[1] of positive integers S and a positive integer value k, determine if there is any $X \subseteq S$ such that $\sum_{x \in X} x = k$.

For example, consider the multiset $S = \{1, 4, 2, 9\}$. There is no subset that sums to 8, where as if the target sum is $k = 7$, the subset $\{1, 4, 2\}$ is a solution.

[1] A *multiset* is like a set, but may include duplicate elements.

In the general case when k is unconstrained this problem is a classic NP-hard problem. However, our goal here are more modest. We are going to consider the case where we include k in the work bounds. We show that as long as k is polynomial in $|S|$ then the work is also polynomial in $|S|$. Solutions of this form are often called *pseudo-polynomial* work (time) solutions.

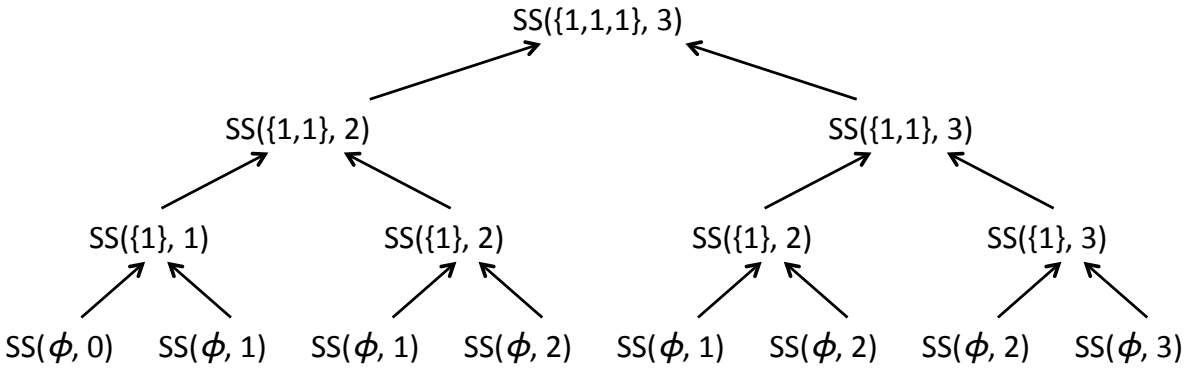
This problem can be solved by brute force simply considering all possible subsets. This takes exponential work since there are an exponential number of subsets. For a more efficient solution, one should consider an inductive solution to the problem. As greedy algorithms tend to be efficient, you should first consider some form of greedy method that greedily takes elements from S. Unfortunately greedy does not work.

We therefore consider a divide-and-conquer solution. Naively, this will also lead to exponential work, but by reusing subproblems we can show that it results in an algorithm that requires only $O(|S|k)$ work. The idea is to consider one element a out of S (any will do) and consider the two possibilities: either a is included in X or not. For each of these two possibilities we make a recursive call on the subset $S \setminus \{a\}$, and in one case we subtract a from k ($a \in X$) and in the other case we leave k as is ($a \notin X$). Here is an algorithm based on this idea. It assumes the input is given as a list (the order of the elements of S in the list does not matter):

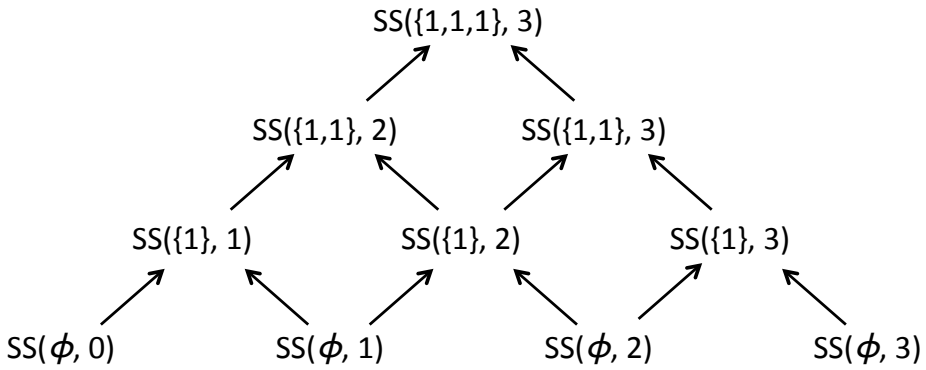
```
1  fun SS(S, k) =
2    case (showl(S), k) of
3      (_, 0) ⇒ true
4    | (NIL, _) ⇒ false
5    | (CONS(a, R), _) ⇒
6        if (a > k) then SS(R, k)
7        else (SS(R, k − a) orelse SS(R, k))
```

Lines 3 and 4 are the base cases. In particular if $k = 0$ then the result is true since the empty set adds to zero. If $k \neq 0$ and S is empty, then the result is false since there is no way to get k from an empty set. If S is not empty but its first element a is greater than k, then we clearly can not add a to X, and we need only make one recursive call. The last line is the main inductive case where we either include a or not. In both cases we remove a from S in the recursive call R. In the left case we are including a in the set so we have to subtract its value from k. In the right case we are not, so k remains the same.

What is the work of this recursive algorithm? Well, it leads to a binary recursion tree that might be n deep. This would imply something like 2^n work. This is not good. The key observation, however, is that there is a huge overlap in the subproblems. For example here is the recursion tree for the instance $\mathtt{SS}(\{1,1,1\},3)$:



As you should notice there are many common calls. In the bottom row, for example there are three calls each to $\texttt{SS}(\emptyset, 1)$ and $\texttt{SS}(\emptyset, 2)$. If we coalesce the common calls we get the following DAG where the leaves at the bottom are the base cases and the root at the top is the instance we are solving.



The question is how do we calculate the number of distinct instances of SS, which is also the number of vertices in the DAG?

For an initial instance $\texttt{SS}(S, k)$ there are are only $|S|$ distinct lists that are ever used (each suffix of S). Furthermore, the value of second argument in the recursive calls only decreases and never goes below 0, so it can take on at most $k+1$ values. Therefore the total number of possible instances of SS (vertices in the DAG) is $|S|(k+1) = O(k|S|)$. Each instance only does constant work to compose its recursive calls. Therefore the total work is $O(k|S|)$. Furthermore it should be clear that the longest path in the DAG is $O(|S|)$ so the total span is $O(|S|)$ and the algorithm has $O(k)$ parallelism.

Why do we say the algorithm is pseudo-polynomial? The size of the subset sum problem is defined to be the number of bits needed to represent the input. Therefore, the input size of k is $\log k$. But the work is $O(2^{\log k}|S|)$, which is exponential in the input size. That is, the complexity of the algorithm is measured with respect to the length of the input (in terms of bits) and not on the numeric value of the input. If the value of k, however, is constrained to be a polynomial in $|S|$ (i.e., $k \leq |S|^c$ for some constant c) then the work is $O(k|S|) = O(|S|^{c+1})$ on input of size $c \log |S| + |S|$, and the algorithm is polynomial in the length of the input.

At this point we have not fully specified the algorithm since we have not explained how to take advantage of the sharing—certainly the recursive code we wrote would not. We will get back to this

after a couple more examples. Again we want to emphasize that the first two orders of business are to figure out the inductive structure and figure out what instances can be shared.

1.2 Minimum Edit Distance

The second problem we consider is a optimization problem, the minimum edit distance problem.

Definition 1.2. The minimum edit distance (MED) problem is, given a character set Σ and two sequences of characters $S = \Sigma^*$ and $T = \Sigma^*$, determine the minimum number of insertions and deletions of single characters required to transform S to T.

For example if we started with the sequence

$$S = \langle A, B, C, A, D, A \rangle$$

we could convert it to

$$T = \langle A, B, A, D, C \rangle$$

with 3 edits (delete the C, delete the last A, and insert a C). This is the best that can be done so the fewest edits needed is 3.

The MED problem is an important problem that has many applications. For example in version control systems such as `git` or `svn` when you update a file and commit it, the system does not store the new version but instead only stores the "differences" from the previous version[2]. Storing the differences can be quite space efficient since often the user is only making small changes and it would be wasteful to store the whole file. Variants of the minimum edit distance problem are use to find this difference. Edit distance can also be used to reduce communication costs by only communicating the differences from a previous version. It turns out that edit-distance is also closely related to approximate matching of genome sequences.

The first observation is that inserting a character into S is equivalent to deleting a character in T. The solution in the above example could be stated as delete C in S, delete the last A in S, delete C in T. Using this formulation, a brute force solution would be to compute all subsequences of S and for any of them that are a subsequences of T return the longest one, clearly an exponential solution. This formulation of the problem is often known as the longest common subsequence problem.

Another possibility would be to consider a greedy method that scans the sequences finding the first difference, fixing it and then moving on. Unfortunately no simple greedy method is known to work. The problem is that there can be multiple ways to fix the error—we can either delete the offending character, or insert a new one. In the example above when we get to the C in S we could either delete C or insert an A. If we greedily pick the wrong way to fix it, we might not end up with an optimal solution. (Recall that in greedy algorithms, once you make a choice, you cannot go back and try an alternative.) Again in the example, if you inserted an A, then more than two more edits will be required.

The first key step in dynamic programming is to recognize the inductive structure of the problem. This step requires precisely defining the subproblems that we will consider. The goal is to keep the number of subproblems small. For example, one possibility would be to find the fewest edits needed

[2] Alternatively it might store the new version, but use the differences to encode the old version.

between any contiguous subsequences of S and T. Since the number of contiguous subsequences of a sequence of length n is $\binom{n+1}{2}$, there are $O(|S|^2|T|^2)$ possible pairs of subsequences to consider. Although this number is much less the exponential, it is still large. A better choice is to consider all suffixes (or prefixes) of S and T. Now there are at most $O(|S||T|)$ pairs of suffixes to consider.

Next, how do we find the $\texttt{MED}(S,T)$ in terms of the smaller problems? The greedy solution gives a good hint how. Suppose $S = s :: S'$ and $T = t :: T'$. If the first characters of S and T match, then no insertion or deleted is needed, and the we only need to consider edits to the suffixes, S' and T'. But what if the first two characters do not match? In particular when we get to the C in our example there were exactly two possible ways to fix it—deleting C or inserting A. As with the subset sum problem, why not consider both ways. This leads to the following algorithm.

```
1  fun MED(S, T) =
2    case (showl(S), showl(T)) of
3      (_, NIL) ⇒ |S|
4    | (NIL, _) ⇒ |T|
5    | (CONS(s, S′), CONS(t, T′)) ⇒
6        if (s = t) then MED(S′, T′)
7        else 1 + min(MED(S, T′), MED(S′, T))
```

In the first base case where T is empty we need to delete all of S to generate an empty string requiring $|S|$ insertions. In the second base case where S is empty we need to insert all of T, requiring $|T|$ insertions. If neither is empty we compare the first character. If they are equal we can just skip them and make a recursive call on the rest of the sequences. If they are different then we need to consider the two cases. The first case ($\texttt{MED}(S,T')$) corresponds to inserting the value t. We pay one edit for the insertion and then need to match up S (which all remains) with the tail of T (we have already matched up the head t with the character we inserted). The second case ($\texttt{MED}(S',T)$) corresponds to deleting the value s. We pay one edit for the deletion and then need to match up the tail of S (the head has been deleted) with all of T.

If we ran the code recursively we would end up with an algorithm that takes exponential work. In particular the recursion tree is binary and has a depth that is linear in the size of S and T. However, as with subset sums, there is significant sharing going on. Again we view the computation as a DAG in which each vertex corresponds to call to MED with distinct arguments. An edge is placed from u to v if the call v uses u. For Figure 3 shows an example of the DAG for $\texttt{MED}(\langle A,B,C\rangle, \langle D,B,C\rangle)$.

We can now place an upper bound on the number of vertices in our DAG by bounding the number of distinct arguments. There can be at most $|S|+1$ possible values of the first argument since in recursive calls we only use suffixes of the original S and there are only $|S|+1$ such suffixes (including the empty and complete suffixes). Similarly there can be at most $|T|+1$ possible values for the second argument. Therefore the total number of possible distinct arguments to MED on original strings S and T is $(|T|+1)(|S|+1) = O(|S||T|)$. Furthermore the depth of the DAG (longest path) is $O(|S|+|T|)$ since each recursive call either removes an element from S or T so after $|S|+|T|$ calls there cannot be any element left. Finally we note that assuming we have constant work operations for removing the head of a sequence (e.g. using a list) then each vertex of the DAG takes constant work and span.

All together this gives us

$$W(\texttt{MED}(S,T)) = O(|S||T|)$$

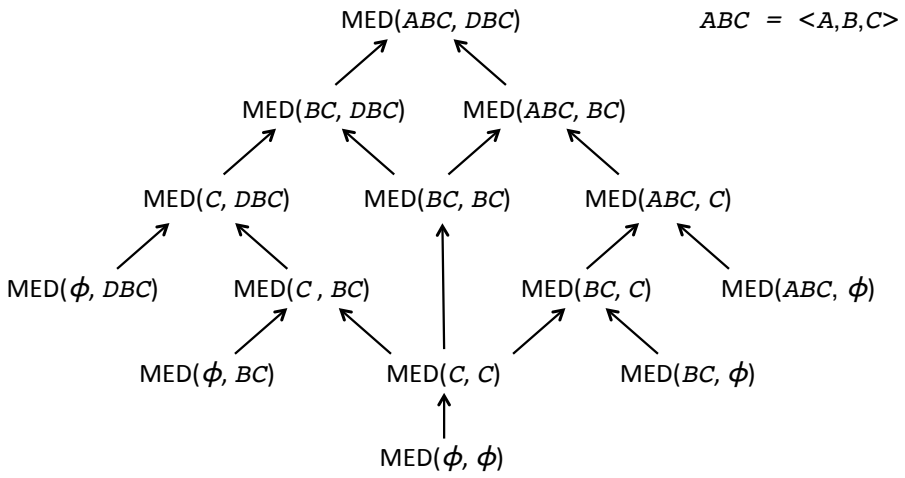


Figure 3: The dynamic programming DAG for an instance of the minimum edit distance (MED) problem.

and

$$S(\texttt{MED}(S,T)) = O(|S| + |T|).$$

1.3 Problems with Efficient Dynamic Programming Solutions

There are many problems with efficient dynamic programming solutions. Here we list just some of them to give a sense of what these problems are.

1. Fibonacci numbers
2. Using only addition compute (n choose k) in $O(nk)$ work
3. Edit distance between two strings
4. Edit distance between multiple strings
5. Longest common subsequence
6. Maximum weight common subsequence
7. Can two strings S1 and S2 be interleaved into S3
8. Longest palindrome
9. longest increasing subsequence
10. Sequence alignment for genome or protein sequences
11. subset sum
12. knapsack problem (with and without repetitions)

13. weighted interval scheduling
14. line breaking in paragraphs
15. break text into words when all the spaces have been removed
16. chain matrix product
17. maximum value for parenthesizing x1/x2/x3.../xn for positive rational numbers
18. cutting a string at given locations to minimize cost (costs n to make cut)
19. all shortest paths
20. find maximum independent set in trees
21. smallest vertex cover on a tree
22. optimal BST
23. probability of generating exactly k heads with n biased coin tosses
24. triangulate a convex polygon while minimizing the length of the added edges
25. cutting squares of given sizes out of a grid
26. change making
27. box stacking
28. segmented least squares problem
29. counting boolean parenthesization – true, false, or, and, xor, count how many parenthesization return true
30. balanced partition – given a set of integers up to k, determine most balanced two way partition
31. Largest common subtree