Programming Team Lecture: Dynamic Programming

Standard Algorithms to Know

Computing Binomial Coefficients (Brassard 8.1)
World Series Problem (Brassard 8.1)
Making Change (Brassard 8.2)
Knapsack (Brassard 8.4 Goodrich 5.3)
Subset Sum (special instance of knapsack where weights=values)
Floyd-Warshall's (Brassard 8.5 Cormen 26.2)
Chained Matrix Multiplication (Brassard 8.6, Cormen 16.1 Goodrich 5.3)
Longest Common Subsequence (Cormen 16.3)
Edit Distance (Skiena 11.2)
Polygon Triangulation (Cormen 16.4)

Example #1: Testing the Catcher

If you read this carefully, this problem is really a longest non-increasing sequence problem.

The recursive solution is as follows:

For the first value in the list, we have two options: (a) take it, (b) don't take it

Work out which of these two options is better and return the maximum of the two strategies.

Our recursive function takes in four parameters:

1) The whole list
2) The size of the list
3) The index of the current value we are considering
4) The height of the last missile taken previous to the current.

One thing to note is that sometimes, the current missile can not be intercepted. This is precisely when its height is greater than the height of the previous missle intercepted.

Initially, the last parameter is set to a value greater than the height of any missile.

In the code, if it's possible to intercept the current missile, then take the maximum of:

1) 1 + maximum number of missiles that can be intercepted from the rest of the list such that the maximum height is that of the current missile.

2) the maximum number of missiles that can be intercepted from the rest of the list such that the maximum height is that of the previous missile taken.

Now, the question is, how can we turn this into DP?

For each sublist starting from the beginning, perhaps we could store the maximum number of missiles that can be intercepted from that sublist. Thus, for the list:

0	1	2	3	4	5	6	7
80	70	60	50	65	45	60	61

We could store (in an auxiliary array) the values:

0	1	2	3	4	5	6	7
1	2	3	4	4	5	5	5

The key to DP is being able to construct this table just using previously filled in values to the table. The problem here is just because we know that the longest list using the first 7 elements (in indexes 0 through 5) is 5, doesn't mean we can decide whether or not we can do better using the last element, 35. In fact, the information we need is what was the height of the last missile in the list of four intercepted missiles. If this is greater than or equal to 61, then we can add 61 to the list, otherwise we can't.

So, we have two options:

1) Store the last missile height the corresponds to each of the longest sequences
2) Stipulate that the entry in the array corresponds to the longest sequence of missiles that can be intercepted with the LAST missile being intercepted.

It turns out that the characterization for choice 2 works quite well. Here is the adjusted auxiliary array if we choose to store this information:

0	1	2	3	4	5	6	7
1	2	3	4	3	5	4	3

To finish up the problem, we simply find the maximum value stored in this auxiliary array.

Now, how do we construct this auxiliary array?

We will fill in each value one by one, in order.

When we fill in the kth element, we must ask ourselves the following question:

Assuming that the last missile intercepted was any of the previous, which of these previous interceptions leads to the maximum number of missile interceptions that end with this current missile?

Here's an example:

Consider filling out the last element in the auxiliary array for the example above.

For each previous missile that is at a height greater than or equal to 61, we must find the one that gives us the maximum number of intercepted missiles.

0	1	2	3	4	5	6	7
80	70	60	50	65	45	60	61

0	1	2	3	4	5	6	7
1	2	3	4	3	5	4	3

We will loop through the array of missile heights, checking to see if those missiles are at height 61 or higher. There are only two. For these two missiles, we check the corresponding entry in the auxiliary array. We see that if 80 is the last missile we take, then by taking 61, we have taken 2 missiles. But then we see that if we take 70 as our last missile, by taking 61 afterwards, we have taken 3 missiles. This is better than 2, so we store 3 in the auxiliary array.

<u>**Example #2: Edit Distance**</u>

The problem of finding an edit distance between two strings is as follows:

Given an initial string s, and a target string t, what is the minimum number of chances that have to be applied to s to turn it into t. The list of valid changes are:

1) Inserting a character
2) Deleting a character
3) Changing a character to another character.

In initially looking for a recursive solution, you may think that there are simply too many recursive cases. We could insert a character in quite a few locations! (If the string is length n, then we can insert a character in n+1 locations.) However, the key observation that leads to a recursive solution to the problem is that ultimately, the last characters will have to match. So, when matching one word to another, on consider the last characters of strings s and t. If we are lucky enough that they ALREADY match, then we can simply "cancel" and recursively find the edit distance between the two strings left when we delete this character from both strings. Otherwise, we MUST make one of three changes:

1) delete the last character of string s
2) delete the last character of string t
3) change the last character of string s to the last character of string t.

Also, in our recursive solution, we must note that the edit distance between the empty string and another string is the length of the second string. (This corresponds to having to insert each letter for the transformation.)

So, an outline of our recursive solution is as follows:

1) If either string is empty, return the length of the other string.
2) If the last characters of both strings match, recursively find the edit distance between each of the strings without that last character.
3) If they don't match then return 1 + minimum value of the following three choices:

   a) Recursive call with the string s w/o its last character and the string t
   b) Recursive call with the string s and the string t w/o its last character
   c) Recursive call with the string s w/o its last character and the string t w/o its last character.

Now, how do we use this to create a DP solution? We simply need to store the answers to all the possible recursive calls. In particular, all the possible recursive calls we are interested in are determining the edit distance between prefixes of s and t.

Consider the following example with s="hello" and t="keep". To deal with empty strings, an extra row and column have been added to the chart below:

		h	**e**	**l**	**l**	**o**
	0	1	2	3	4	5
k	1	1	2	3	4	5
e	2	2	1	2	3	4
e	3	3	2	2	3	4
p	4	4	3	3	3	4

An entry in this table simply holds the edit distance between two prefixes of the two strings. For example, the highlighted square indicates that the edit distance between the strings "he" and "keep" is 3. In order to fill in all the values in this table we will do the following:

1) Initialize values corresponding to the base case in the recursive solution. These are all the values dealing with edit distances with the empty string. (They are the first row and first column inside the table.)

2) Loop through the table from the top left to the bottom right. In doing so, simply follow the recursive solution.

If the characters you are looking at match, store the number in the square diagonally up and left from the square you are filling in. This square holds the edit distance between the two strings w/o their last character.

If the characters don't match, look that the square to your left, above you, and the square diagonally up and left of the one you are filling in. Take the minimum of these and add 1. This corresponds exactly to the recursive call, except that instead of making it, you just look it up in the table.

Finally we can reconstruct the path as follows:

		h	**e**	**l**	**l**	**o**
	0	1	2	3	4	5
k	1	1	2	3	4	5
e	2	2	1	2	3	4
e	3	3	2	2	3	4
p	4	4	3	3	3	4

Start at the bottom right corner of the auxiliary array. Compare the corresponding characters. If they match, automatically go up the diagonal and do not edit anything. If they don't match, as in the case of 'o' and 'p', the look at the three squares we mentioned before: up, left, and up&left. If any of these is one less than the current square value, go to that square. (I chose one of the two possible choices.) Then make the edit that corresponds to this choice. For the choice above, that means deleting the o:

hello -> hell

From here on, the edit path dictated is hello->hell->help->heep->keep.

Here is pseudocode for the algorithm:

```
int strmatchdyn(String s, String t) {

  int table[s.length()+1][t.length()+1], i, j;

  // Initial edit distances (base cases).
  for (i=0; i< s.length()+1; i++) table[i][0] = i;
  for (i=0; i< t.length()++1; i++) table[0][i] = i;

  // Go through whole table.
  for (i=1; i<s.length()+1; i++)
    for (j=1; j<t.lengeth()+1; j++)

      // Check if the current characters match.
      if (s[i-1]==t[j-1])
        table[i][j] = table[i-1][j-1];

      // Otherwise take the minimum of the 3 cases.
      else
        table[i][j] = 1+min(table[i-1][j-1],
                            table[i-1][j]  ,table[i][j-1]);

  return table[slen][tlen];
}
```

<u>Example #3: 0-1 Knapsack Problem</u>

The problem is as follows: your goal is to maximize the value of a knapsack that can hold at most W units worth of goods from a list of items I_0, I_1, ... I_{n-1}. Each item has two attributes:

1) Value - let this be v_i for item I_i.
2) Weight - let this be w_i for item I_i.

Now, instead of being able to take a certain weight of an item, you can only either take the item or not take the item.

The naive way to solve this problem is to cycle through all 2^n subsets of the n items and pick the subset with a legal weight that maximizes the value of the knapsack. But, we can find a dynamic programming algorithm that will USUALLY do better than this brute force technique.

Our first attempt might be to characterize a sub-problem as follows:

Let S_k be the optimal subset of elements from $\{I_0, I_1,... I_k\}$. But what we find is that the optimal subset from the elements $\{I_0, I_1,... I_{k+1}\}$ may not correspond to the optimal subset of elements from $\{I_0, I_1,... I_k\}$ in any regular pattern. Basically, the solution to the optimization problem for S_{k+1} might NOT contain the optimal solution from problem S_k.

To illustrate this, consider the following example:

Item	Weight	Value
I_0	3	10
I_1	8	4
I_2	9	9
I_3	8	11

The maximum weight the knapsack can hold is 20.

The best set of items from $\{I_0, I_1, I_2\}$ is $\{I_0, I_1, I_2\}$ but the best set of items from $\{I_0, I_1, I_2, I_3\}$ is $\{I_0, I_2, I_3\}$. In this example, note that this optimal solution, $\{I_0, I_2, I_3\}$, does NOT build upon the previous optimal solution, $\{I_0, I_1, I_2\}$. (Instead it build's upon the solution, $\{I_0, I_2\}$, which is really the optimal subset of $\{I_0, I_1, I_2\}$ with weight 12 or less.)

So, now, we must rework our example. In particular, after trial and error we may come up with the following idea:

Let B[k, w] represent the maximum total value of a subset S_k with weight w. Our goal is to find B[n, W], where n is the total number of items and W is the maximal weight the knapsack can carry.

Using this definition, we have $B[0, w] = v_0$, if $w >= w_0$.
$= 0$, otherwise

Now, we can derive the following relationship that $B[k, w]$ obeys:

$B[k, w] = B[k - 1,w]$, if $w_k > w$
$= \max \{ B[k - 1,w], B[k - 1,w - w_k] + v_k\}$

In English, here is what this is saying:

1) The maximum value of a knapsack with a subset of items from $\{I_0, I_1, \ldots I_k\}$ with weight w is the same as the maximum value of a knapsack with a subset of items from $\{I_0, I_1, \ldots I_{k-1}\}$ with weight w, if item k weighs greater than w.

Basically, you can NOT increase the value of your knapsack with weight w if the new item you are considering weighs more than w – because it WON'T fit!!!

2) The maximum value of a knapsack with a subset of items from $\{I_0, I_1, \ldots I_k\}$ with weight w could be the same as the maximum value of a knapsack with a subset of items from $\{I_1, I_2, \ldots I_{k-1}\}$ with weight w, if item k should not be added into the knapsack.

OR

3) The maximum value of a knapsack with a subset of items from $\{I_0, I_1, \ldots I_k\}$ with weight w could be the same as the maximum value of a knapsack with a subset of items from $\{I_0, I_1, \ldots I_{k-1}\}$ with weight $w-w_k$, plus item k.

You need to compare the values of knapsacks in both case 2 and 3 and take the maximal one.

Recursively, we will STILL have an $O(2^n)$ algorithm. But, using dynamic programming, we simply have to do a double loop - one loop running n times and the other loop running W times.

Question: In which cases would a running time of $O(nW)$ be worse than a running time of $O(2^n)$?
Here is a dynamic programming algorithm to solve the 0-1 Knapsack problem:

Input: S, a set of n items as described earlier, W the total weight of the knapsack. (Assume that the weights and values are stored in separate arrays named w and v, respectively.)

Output: The maximal value of items in a valid knapsack.

```
int w, k;
for (w=0; w <= W; w++)
     B[w] = 0

for (k=0; k<n; k++)
     for (w = W; w>= w[k]; w--)

          if (B[w - w[k]] + v[k] > B[w])
              B[w] = B[w - w[k]] + v[k]
```

Note on run time: Clearly the run time of this algorithm is O(nW), based on the nested loop structure and the simple operation inside of both loops. When comparing this with the previous $O(2^n)$, we find that depending on W, either the dynamic programming algorithm is more efficient or the brute force algorithm could be more efficient. (For example, for n=5, W=100000, brute force is preferable, but for n=30 and W=1000, the dynamic programming solution is preferable.)

Once again, note that the inner loop MUST run backwards, otherwise we allow ourselves to take more than one copy of an item.

Let's run through an example:

i	Item	w_i	v_i
0	I_0	4	6
1	I_1	2	4
2	I_2	3	5
3	I_3	1	3
4	I_4	6	9
5	I_5	4	7

W = 10

Item	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	6	6	6	6	6	6	6
1	0	0	4	4	6	6	10	10	10	10	10
2	0	0	4	5	6	9	10	11	11	15	15
3	0	3	4	7	8	9	12	13	14	15	18
4	0	3	4	7	8	9	12	13	14	16	18
5	0	3	4	7	8	10	12	14	15	16	19

If the problem changes and we DO allow as many copies of a particular item, then all we need to do to edit the code is run the inner loop forwards, which then allows larger knapsacks to be built from ones that already have copies of the item being considered.

Example #4: The Matrix Chain Multiplication Problem

Given a chain of matrices to multiply, determine the how the matrices should be parenthesized to minimize the number of single element multiplications involved.

First off, it should be noted that matrix multiplication is associative, but not commutative. But since it is associative, we always have:

((AB)(CD)) = (A(B(CD)))

or equality for any such grouping as long as the matrices in the product appear in the same order.

It may appear on the surface that the amount of work done won't change if you change the parenthesization of the expression, but we can prove that is not the case with the following example:

Let A be a 2x10 matrix
Let B be a 10x50 matrix
Let C be a 50x20 matrix

Note that any matrix multiplication between a matrix with dimensions ixj and another with dimensions jxk will perform ixjxk element multiplications creating an answer that is a matrix with dimensions ixk. Also note that the second dimension in the first matrix and the first dimension in the second matrix must be equal in order to allow matrix multiplication.

Consider computing A(BC):

# multiplications for (BC) = 10x50x20 = 10000, creating a 10x20 answer matrix

# multiplications for A(BC) = 2x10x20 = 400,

Total multiplications = 10000 + 400 = 10400.

Consider computing (AB)C:

# multiplications for (AB) = 2x10x50 = 1000, creating a 2x50 answer matrix

# multiplications for (AB)C = 2x50x20 = 2000,

Total multiplications = 1000 + 2000 = 3000, a significant difference.

Thus, the goal of the problem is given a chain of matrices to multiply, determine the fewest number of multiplications necessary to compute the product. We will formally define the problem below:

Let $A = A_0 \bullet A_1 \bullet \ldots A_{n-1}$

Let $N_{i,j}$ denote the minimal number of multiplications necessary to find the product $A_i \bullet A_{i+1} \bullet \ldots A_j$. And let $d_i x d_{i+1}$ denote the dimensions of matrix A_i.

We must attempt to determine the minimal number of multiplications necessary($N_{0,n-1}$) to find A, assuming that we simply do each single matrix multiplication in the standard method.

The key to solving this problem is noticing the sub-problem optimality condition:

If a particular parenthesization of the whole product is optimal, then any sub-parenthesization in that product is optimal as well. Consider the following illustration:

Assume that we are calculating ABCDEF and that the following parenthesization is optimal:

(A (B ((CD) (EF))))

Then it is necessarily the case that

(B ((CD) (EF)))

is the optimal parenthesization of BCDEF.

Why is this?

Because if it wasn't, and say (((BC) (DE)) F) was better, then it would also follow that

(A (((BC) (DE)) F)) was better than

(A (B ((CD) (EF)))), contradicting its optimality.

This line of reasoning is nearly identical to the reasoning we used when deriving Floyd-Warshall's algorithm.

Now, we must make one more KEY observation before we design our algorithm:

Our final multiplication will ALWAYS be of the form

$(A_0 \bullet A_1 \bullet \ldots A_k) \bullet (A_{k+1} \bullet A_{k+2} \bullet \ldots A_{n-1})$

In essence, there is exactly one value of k for which we should "split" our work into two separate cases so that we get an optimal result. Here is a list of the cases to choose from:

$(A_0) \bullet (A_1 \bullet A_{k+2} \bullet \ldots A_{n-1})$
$(A_0 \bullet A_1) \bullet (A_2 \bullet A_{k+2} \bullet \ldots A_{n-1})$
$(A_0 \bullet A_1 \bullet A_2) \bullet (A_3 \bullet A_{k+2} \bullet \ldots A_{n-1})$
...
$(A_0 \bullet A_1 \bullet \ldots A_{n-3}) \bullet (A_{n-2} \bullet A_{n-1})$
$(A_0 \bullet A_1 \bullet \ldots A_{n-2}) \bullet (A_{n-1})$

Basically, count the number of multiplications in each of these choices and pick the minimum. One other point to notice is that you have to account for the minimum number of multiplications in each of the two products.

Consider the case multiplying these 4 matrices:

A: 2x4
B: 4x2
C: 2x3
D: 3x1

1. (A)(BCD) - This is a 2x4 multiplied by a 4x1,
so 2x4x1 = 8 multiplications, plus whatever
work it will take to multiply (BCD).

2. (AB)(CD) - This is a 2x2 multiplied by a 2x1,
so 2x2x1 = 4 multiplications, plus whatever
work it will take to multiply (AB) and (CD).

3. (ABC)(D) - This is a 2x3 multiplied by a 3x1,
so 2x3x1 = 6 multiplications, plus whatever
work it will take to multiply (ABC).

Thus, we can state the following recursive formula:

$N_{i,j}$ = min value of $N_{i,k} + N_{k+1,j} + d_i d_{k+1} d_{j+1}$, over all
valid values of k.

One way we can think about turning this recursive formula into a dynamic programming solution is by deciding which sub-problems are necessary to solve first. Clearly it's necessary to solve the smaller problems before the larger ones. In particular, we need to know $N_{i,i+1}$, the number of multiplications to multiply any adjacent pair of matrices before we move onto larger tasks. Similarly, the next task we want to solve is finding all the values of the form $N_{i,i+2}$, then $N_{i,i+3}$, etc. So here is our algorithm:

```
1) Initialize N[i][i] = 0, and all other entries in N to ∞.
2) for i=1 to n-1 do the following
        2i) for j=0 to n-1-i do
                2ii) for k=j to j+i-1
                        2iii) if (N[j][j+i-1] >
                                N[j][k]+N[k+1][j+i-1]+$d_j d_{k+1} d_{i+j}$)

                            N[j][j+i-1]=
                                N[j][k]+N[k+1][j+i-1]+$d_j d_{k+1} d_{i+j}$
```

Here is the example we worked through in class:

Matrix	Dimensions
A	2x4
B	4x2
C	2x3
D	3x1
E	1x4

	A	B	C	D	E
A	0	16	28	22	30
B		0	24	14	30
C			0	6	14
D				0	12
E					0

First we determine the number of multiplications necessary for 2 matrices:

AxB uses 2x4x2 = 16 multiplications
BxC uses 4x2x3 = 24 multiplications
CxD uses 2x3x1 = 6 multiplications
DxE uses 3x1x4 = 12 multiplications

Now, let's determine the number of multiplications necessary for 3 matrices

(AxB)xC uses 16 + 0 + 2x2x3 = 28 multiplications
Ax(BxC) uses 0 + 24 + 2x4x3 = 48 multiplications, so 28 is min.

(BxC)xD uses 24 + 0 + 4x3x1 = 36 multiplications
Bx(CxD) uses 0 + 6 + 4x2x1 = 14 multiplications, is 14 is min.

(CxD)xE uses 6 + 0 + 2x1x4 = 14 multiplications
Cx(DxE) uses 0 + 12 + 2x3x4 = 36, so 14 is min.

Four matrices next:

Ax(BxCxD) uses 0 + 14 + 2x4x1 = 22 multiplications
(AxB)x(CxD) uses 16 + 6 + 2x2x1 = 26 multiplications
(AxBxC)xD uses 28 + 0 + 2x3x1 = 34 multiplications, 22 is min.

Bx(CxDxE) uses 0 + 14 + 4x2x4 = 46 multiplications
(BxC)x(DxE) uses 24 + 12 + 4x3x4 = 84 multiplications
(BxCxD)xE uses 14 + 0 + 4x1x4 = 30 multiplications, 30 is min.

For the answer:

Ax(BxCxDxE) uses 0 + 30 + 2x4x4 = 62 multiplications
(AxB)x(CxDxE) uses 16 + 14 + 2x2x4 = 46 multiplications
(AxBxC)x(DxE) uses 28 + 12 + 2x3x4 = 64 multiplications
(AxBxCxD)xE uses 22 + 0 + 2x1x4 = 30 multiplications

Answer = 30 multiplications

Homework Problems from acm.uva.es site
10131, 10069, 10154, 116, 10003, 10261, 10271, 10201

The team should attempt to complete these collectively. I will email one more problem to the three team members that all of them should do.

References

Brassard, Gilles & Bratley, Paul. Fundamentals of Algorithmics (text for COT5405)
Prentice Hall, New Jersey 1996 ISBN 0-13-335068-1

Cormen, Tom, Leiserson, Charles, & Rivest, Ronald. Introduction to Algorithms
The MIT Press,Cambridge, MA 1992 ISBN 0-262-03141-8 (a newer version exists)

Goodrich, Michael & Tamassia, Roberto. Algorithm Design (text for COP3530)
John Wiley & Sons, New York 2002 ISBN 0-471-38365-1

Skiena, Steven & Revilla, Miguel. Programming Challenges
Springer-Verlag, New York 2003 ISBN 0-387-00163-8