Dynamic Programming Introduction
Chapter 16,26 – Material augments Ch 16

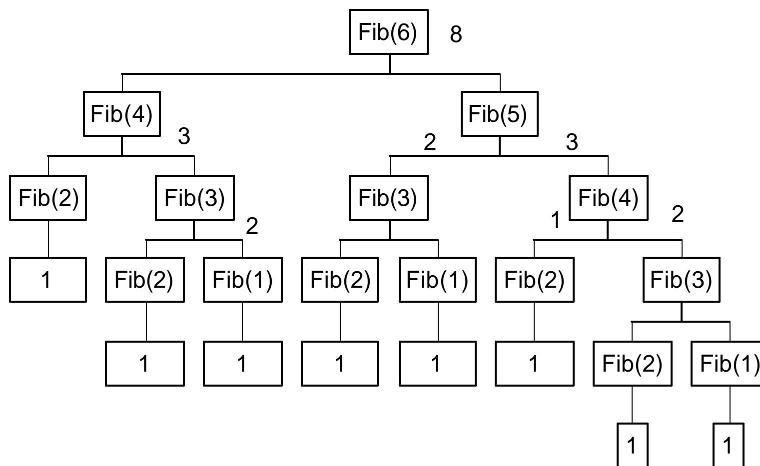
Any recursive formula can be directly translated into recursive algorithms. However, sometimes the compiler will not implement the recursive algorithm very efficiently. When this is the case, we must do something to help the compiler by rewriting the program to systematically record the answers to subproblems in a table. This is the basic approach behind dynamic programming – all problems must have "optimal substructure."

Example: Consider the Fibonacci sequence.
Fib(1)=1
Fib(2)=1
Fib(n)=Fib(n-1)+Fib(n-2)
Get Sequence: 1,1,2,3,5,8,12,20,32 …

Implementation:

```
Recursive-Fib(n)
    if n=1 or n=2 then return 1
    else return Recursive-Fib(n-1)+Recursive-Fib(n-2)
```

Recursion Tree for Fib(6):



We end up recomputing Fib(3) three times, and Fib(4) twice!
A lot of redundant work recomputed.

Runtime T(n)=T(n-2)+T(n-1) ; grows at same rate as Fib sequence, exponential.

A better solution is to remember the previous answers to Fib(x), perhaps in a table, and when needed again just pull the answer straight out of the table. Eliminates much recomputation. In Fib, we really only need to store the last two values, Fib(n-2) and Fib(n-1) instead of smaller Fib numbers.

```
Faster-Fib(n)
        if n=1 or n=2 then return 1
        else
                last← 1; next_to_last← 1;
                for I← 1 to n do
                        answer← last+next_to_last
                        next_to_last← last
                        last← answer
        return answer
```

Faster-Fib runs in O(n) time. Works in a straightforward manner by remembering the previous values. In this case we only had to remember the last two, but in general we may need a table to store previously computed values.

All-Pairs Shortest Path

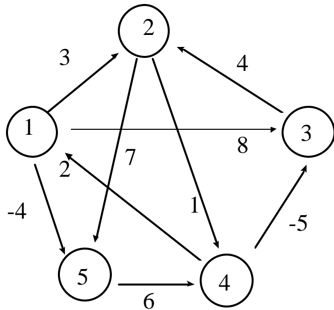
Say we want to compute the shortest distance between every single pair of vertices. As described earlier, we could just run Dijkstra's algorithm on every vertex, resulting in $O(V^3)$ runtime.

Recall the relaxation property, where we set d[v]=min(d[v],d[u]+w(u,v)). This formula indicates that the best distance to v is either the previously known distance to v, or the result of going from s to u and then directly from u to v.

The dynamic programming algorithm is based upon Dijkstra's observations.

Set $D_{k,i,j}$ to be the weight of the shortest path from vertex i to vertex j using only k nodes as intermediaries.
$D_{0,i,j}$ = w[i,j] by definition.

Example:



D_0 can be represented as a matrix: shortest path between nodes using 0 intermediates (direct links)

D(0)	**1**	**2**	**3**	**4**	**5**
1	0	3	8	∞	-4
2	∞	0	0	1	7
3	∞	4	0	∞	∞
4	2	∞	-5	0	∞
5	∞	∞	∞	6	0

Observation: The shortest path from vertex i to vertex j that uses only k intermediate nodes is the shortest path that either does not use vertex k at all, or consists of the merging of the two paths vertex i to vertex k and vertex k to vertex j. This leads to the formula:

$D_{k,i,j} = \min \{ \quad D_{k-1,i,j} \quad \text{or} \quad D_{k-1,i,k} + D_{k-1,k,j} \}$

Previous best — Previous best to vertex k, then best from k to j

Putting this together into an algorithm, named Floyd-Warshall:

```
Floyd-Warshall-All-Pairs-Shortest(G,w)
    Initialize d[i,j] ← w(i,j), ∞ if no such link
    Initialize path[i,j] ← ∞
    for k← 1 to |V|
        for i← 1 to |V|
            for j← 1 to |V|
                if d[i,k]+d[k,j]<d[i,j] then          ; update min
                    d[i,j] ← d[i,k]+d[k,j]
                    path[i,j] ← k          ; store to get path
```

Here we use one matrix and overwrite it for each iteration of k.

Example on above graph:

D(k=0)	j=1	2	3	4	5
I=1	0	3	8	∞	-4
2	∞	0	0	1	7
3	∞	4	0	∞	∞
4	2	∞	-5	0	∞
5	∞	∞	∞	6	0

Path	j=1	2	3	4	5
I=1	∞	∞	∞	∞	∞
2	∞	∞	∞	∞	∞
3	∞	∞	∞	∞	∞
4	∞	∞	∞	∞	∞
5	∞	∞	∞	∞	∞

Going through shortest path to vertex 1:

1,1: d(1,1)+d(1,1)<d(1,1)? 0+0<0? No

1,2: d(1,1)+d(1,2)<d(1,2? 0+3<3? No

…

2,1: d(2,1)+d(1,1)<d(2,1)? ∞+3<∞? No

2,2: d(2,1)+d(1,2)<d(2,2)? ∞+3<0? No

…

4,2: d(4,1)+d(1,2)<d(4,2)? 2+3<∞? YES! Update d(4,2) to 5

4,5: d(4,1)+d(1,5)<d(4,5)? 2+-4<∞? YES! Update d(4,5) to –2

D(k=1)	j=1	2	3	4	5
I=1	0	3	8	∞	-4
2	∞	0	0	1	7
3	∞	4	0	∞	∞
4	2	5	-5	0	-2
5	∞	∞	∞	6	0

Path	j=1	2	3	4	5
I=1	∞	∞	∞	∞	∞
2	∞	∞	∞	∞	∞
3	∞	∞	∞	∞	∞
4	∞	1	∞	∞	1
5	∞	∞	∞	∞	∞

Going through shortest path to vertex 2 (includes going through vertex 1)

1,4: d(1,2)+d(2,4)<d(1,4)? 3+1<∞? YES , update d(1,4) to 4

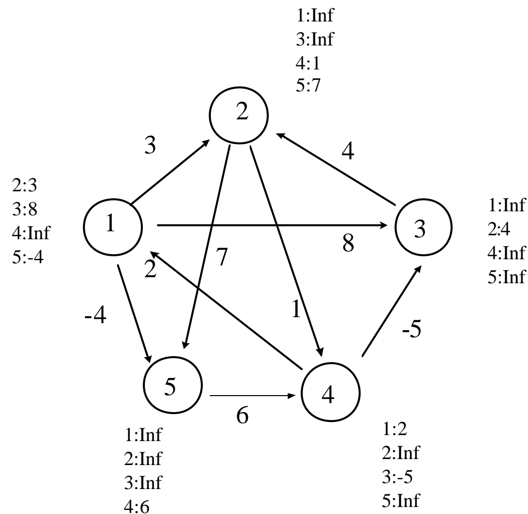
Keep going, when k=5:

D(k=5)	**j=1**	**2**	**3**	**4**	**5**
I=1	0	1	-3	2	-4
2	3	0	-4	1	-1
3	7	4	0	5	3
4	2	-1	-5	0	-2
5	8	5	1	6	0

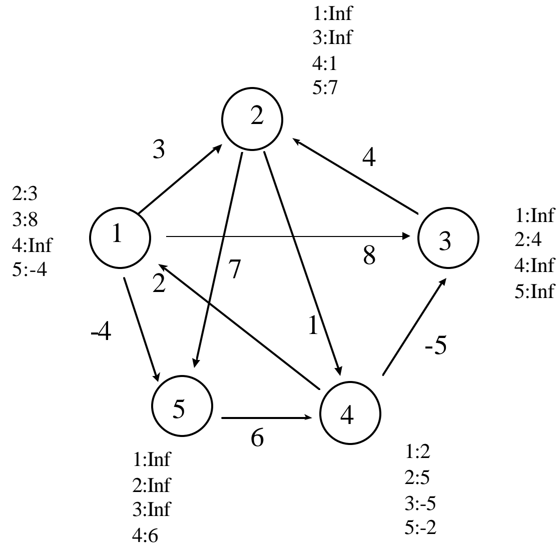
Path	**j=1**	**2**	**3**	**4**	**5**
I=1	∞	3	4	5	1
2	4	∞	4	2	1
3	4	3	∞	2	1
4	4	3	4	∞	1
5	4	3	4	5	∞

Distance matrix gives the distance of the shortest path from i to j. By following the node in the path matrix, we can find the path to get the shortest path from i to j.

Hard to follow table? Easier to visualize on graph.
After k=0:

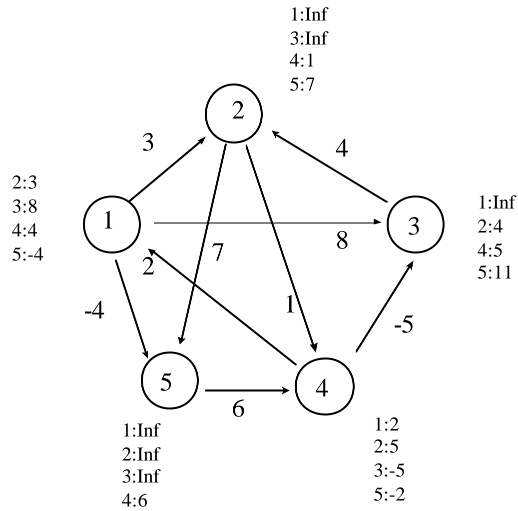


k=1: Update shortest paths going via shortest path to vertex 1.
This is 4→2, 4→5. Note 4→3 is larger going through 1 than direct edge.



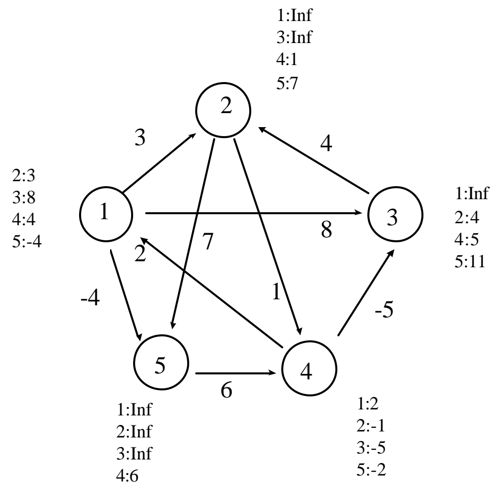
k=2: Update shortest paths via shortest path to vertex 2.

$3 \rightarrow 4$, $3 \rightarrow 5$, $1 \rightarrow 4$

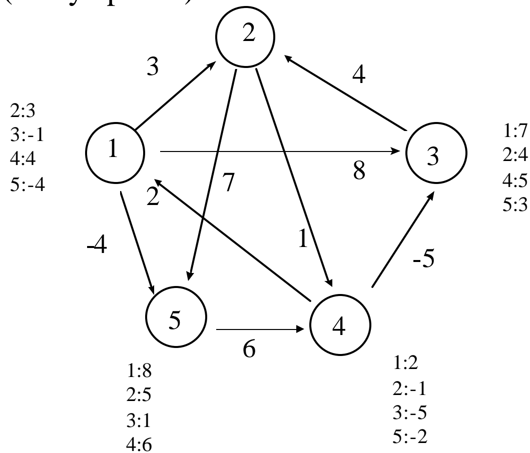


k=3: Update shortest paths via shortest path to vertex 3.

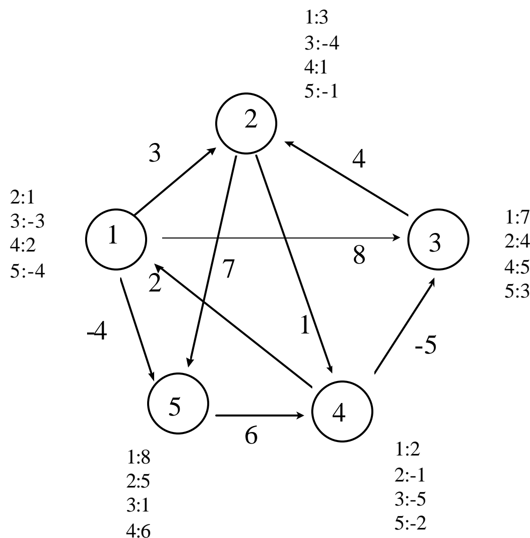
$4 \rightarrow 2$



k=4: Update shortest paths via shortest path to vertex 4.
(Many updates)



k=5: Update shortest paths via shortest path to vertex 5.
$1 \rightarrow 2$, $1 \rightarrow 3$,$1 \rightarrow 4$



Done! Have the shortest path between all nodes. If we maintain pointers to backnodes, we can get the actual path as was done with the matrix.

Approximate String Matching

Later on, we will examine some algorithms to efficiently determine if a pattern string P exists within text string T. If you have ever used the “Find” feature in a word processor to look for a word, then you have just performed string matching.

If the pattern P is m characters long, and the text T is n characters long (we assume n>>m) then the naïve string matching algorithm that scans through T, comparing each character to one in P, takes O(mn) in the worst case, O(n) on average.

But what about approximate matches? When if the pattern is only off by one character? What if the pattern is only missing one character? The naïve algorithm will say that the string does not match at all. The approximate string matching algorithm has many uses in text processing, language recognition, checking for misspelled words, and speech recognition.

Let k be a nonnegative integer. A k-approximate match is a match of P in T that has at most k differences. The differences may be one of the following three types:

1. The corresponding characters in P and T are different
2. P is missing a character that appears in T
3. T is missing a character that appears in P

Example:

P= a n c h o r g e
T= a c h o r a e (no space there, just added for illustrating mismatch)

Two mismatches, one at the missing n, another at different g/a. Naïve algorithm would say that every character mismatches except the a!

We can solve this with dynamic programming.

Let D[i,j] be the minimum number of differences between $P_1..P_i$ and a segment of T between $T_1..T_j$.

There will be a k-approximate match ending at t_j for any j such that D[m,j]<=k. Thus if we want to find the first k-approximate match, we can stop as soon as we find an entry less than or equal to k in the last row of D. The rules for computing entries of D consider each of the possible types of differences outlined above.

D[i,j] is the minimum of :

1. If $P_i = T_j$ then D[i-1,j-1] else D[i-1,j-1]+1 — If match, then previous best mismatch otherwise mismatch, so add 1 to difference
2. D[i-1,j]+1 — P_i missing from T
3. D[i,j-1]+1 — T_j missing from P

If building up values of D in a table, very simple lookup with rows to n, length of text, and columns m, length of the pattern. Can fill in top row with zero's since an empty pattern string has no differences in any target text. The first column is filled with the value of m, since a pattern of length m will have m mismatches with an empty text.

Computing D[i,j] can be done only looking at neighbors!

D	0	1	2	3	4	…	n
	0	0	0	0	0	0	0
1	1						
2	2		**D[i-1,j-1]**	**D[i-1,j]**			
…	…		**D[i,j-1]**	D[i,j]			
m	m						

Example:

		H	a	v	e		a		h	s	p	p	y
	0	0	0	0	0	0	0	0	0	0	0	0	0
h	1	1	1	1	1	1	1	1	0	1	1	1	1
a	2	2	1	2	2	2	1	2	1	1	2	2	2
p	3	3	2	2	3	3	2	2	2	2	1	2	3
p	4	4	3	3	3	4	3	3	3	3	2	1	2
y	5	5	4	4	4	4	4	4	4	4	3	2	1

If we were happy with a 3 element mismatch, we could stop once we find a 3 in the last row. The best we can do is a 1 element mismatch.

Runtime? O(mn) to fill the table.
Space? The table D is very large since n will typically be large.
BUT we don't have to store the whole table. What parts of the table do we need?

Implementation: Easy to do, left as an exercise.

Knapsack Problem

Given n items, each with a value v[I] and weight w[I] and knapsack capable of supporting weight W, find the largest-valued subset possible to carry in the knapsack.

This solution first solves the problem for knapsacks capable of supporting weight 1, weight 2, weight 3, and so on, up until weight W. Solutions for small knapsacks can be used to find solutions for larger knapsacks.

To further divide the problem, the steps outlined above are executed for different combinations of items. First, we find the optimal solution only using item 1. Then we find the optimal solution if we allow the use of item 1 and 2, then item 1,2, and 3, and so on up until we reach item n.

Let's define *best[i,k]* to be the maximum value that a knapsack capable of supporting weight k can hold, using items 1 through i. If we sort the items by weight (item 1 is the lightest, item n is the heaviest, then we have the following:

best[i,k] = 0 if i=0 or k=0 (no items or knapsack with a hole in it)

best[i-1,k] if $k<w_i$

(item is too heavy for knapsack of size k, so the best we can do is the previous best)

$\max(best[i-1,k], best[i-1,k-w_i]+v_i)$ if $k \geq w_i$

(if item fits, the best we can do is either the previous best, or it's the best value with the knapsack of weight just capable of supporting the item but not including the item ($k-w_i$), plus the value of the item, whichever is bigger)

We can now construct a table looking something like the following. We can fill in zeros for k=0 and I=0. The rest must be calculated from the recursive formula. For example, if W=4 and n=4, and v[1]=3, w[1]=2; v[2]=5, w[2]=2; v[3]=6, w[3]=2; v[4]=10, w[4]=5 we get the numbers shown below:

best[i,k]	k=0	k=1	k=2	k=3	…	k=W
I=0	0	0	0	0	0	0
I=1	0	0	3	3	3	
I=2	0	0	5	5	8	
I=3	0	0	6	6	11	
…	0	0	6	6	11	
I=n	0					

Once the table has been filled in, our answer for the largest weight possible is simply the number in the lower right hand corner of the matrix. To actually find the set of items, we can store the last item put into the knapsack into the array as a tuple (best[I,k],last item) whenever a cell is updated. The last item will either be I-1 or I, depending on which course of action is taken in the recursive formula. Our table becomes something like:

best[i,k],last	k=0	k=1	k=2	k=3	…	k=W
I=0	(0,0)	(0,0)	(0,0)	(0,0)	(0,0)	(0,0)
I=1	(0,0)	(0,0)	(3, item 1)	(3, item 1)	(3, item 1)	
I=2	(0,0)	(0,0)	(5, item 2)	(5, item 2)	(8, item 2)	
I=3	(0,0)	(0,0)	(6, item 3)	(6, item 3)	(6, item 3)	
…	(0,0)	(0,0)	(6, item 3)	(6, item 3)	(6, item 3)	
I=n	(0,0)					

To find the optimum set of items S, we start in the lower right hand corner and put that item into the knapsack. Then, we work our way to the left by subtracting off the weight of the item and move up until we get to an item different from the one we just removed, repeating the process until we end up 0. In the example above, we'd first put item 3 into the knapsack, then move left two spaces since the weight of item 3 is 2, which brings us to the column where k=2. Now we move up to column 2, at which point the item is different from item 3. The item stored is item 2, so we put an item2 into the knapsack, subtract 2, and quit since k is now zero.

The time required to build this matrix is clearly O(Wn) and would be very straightforward to implement in two FOR loops. The space required in the straightforward implementation is O(Wn) space, but we really only need to keep track of the previous row. If we just overwrite old values in the current row, the algorithm can be implemented in O(W) space.