Dynamic Programming (Ch. 15)

Dynamic programming solutions rely on the **optimal substructure property**. Usually the recursive solutions to these problems takes exponential time with many redundant calculations. The Floyd-Warshall algorithm used dynamic programming techniques to compute the APSP problem from the bottom up.

Two more dynamic programming examples from Chapter 15 that we will cover in this lecture:

- 0/1 Knapsack
- Longest-Common-Subsequence

0-1 Knapsack Problem

0-1 Knapsack Problem:
Given items T_1, T_2, T_3, ..., T_n, with associated weights w_1, w_2, w_3, ..., w_n and benefit values b_1, b_2, b_3, ..., b_n, how can we maximize the total benefit subject to an absolute weight limit W?

$$S = \{\text{maximize} \sum_{i \in T} b_i \quad \text{subject to} \sum_{i \in T} w_i \leq W.\}$$

A brute-force solution to this problem is to enumerate all possible subsets of T and select the one with the highest total benefit from among all those whose weight is ≤ W

The running time of this brute-force approach is $\theta(2^n)$.

0-1 Knapsack Problem

Suppose we use an approach like that used in the Floyd-Warshall APSPs algorithm:
Define subproblems by using a parameter k so that subproblem k is the best way to fill the knapsack using only items from the set $T_1 \ldots T_k$

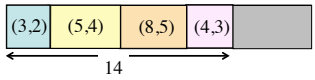
Derive an equation that takes the best solution using only items from T_{k-1} and considers how to add the item k to that

Unfortunately, we can find a counter-example to this approach that shows the global solution obtained in this way may actually contain a suboptimal subproblem solution

0-1 Knapsack solution counterexample

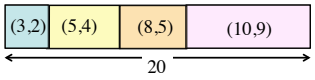
Given the following (benefit, weight) pairs
T_1 T_2 T_3 T_4 T_5
(3,2), (5,4), (8,5), (4,3) and (10,9) and W = 20

Best solution with the first 4 items: $\{T_1, T_2, T_3, T_4\}$



overall benefit = 20 (wt=14)

Best solution with the first 5 items excluding T_4:



overall benefit = 26 (wt=20)

0-1 Knapsack Problem

A better approach is to formulate each sub-problem as that of computing B[k,w], which is defined as the maximum total value of a subset of T_k from among all those having total weight* exactly *w.

$$B[k,w] = \begin{cases} B[k-1, w] & \text{if } w_k > w \\ \max\{B[k-1,w], B[k-1, w - w_k] + b_k\} & \text{otherwise} \end{cases}$$

The best subset of T_k that has total weight w is either the best subset of T_{k-1} that has total weight w or the best subset of T_{k-1} that has total weight $w - w_k$ plus the benefit of item k.

This solution is simple (only 2 parameters, k and w) and it satisfies the sub-problem optimization condition. The problem B[k, w] is built from B[k-1, w] or B[k-1, $w - w_k$].

Algorithm 0-1 Knapsack

Input: Set T of n items, such that item i has positive benefit b_i and positive integer weight w_i; positive integer for maximum total weight W.

Output: For w = 0, ...,W, maximum benefit B[w] of a subset of T with total weight w. B is an array indexed from 0 to W.

```
0-1Knapsack (T, W)
1. for w = 0 to W do
2.     B[w] = 0
3. for k = 1 to n  do
4.     for w = W downto w_k do
5.         if B[w – w_k] + b_k > B[w] then
6.             B[w] = B[w – w_k] + b_k
```

0-1 Knapsack Algorithm Execution

Given the following (benefit, weight) pairs

T_1 T_2 T_3 T_4

(12, 2), (10,1), (20, 3), (15, 2) and W = 5

0-1Knapsack (T, W)
1. for w = 0 to W do
2. B[w] = 0
3. for k = 1 to n do
4. for w = W downto w_k do
5. if B[w – w_k] + b_k > B[w] then
6. B[w] = B[w – w_k] + b_k

$$B[k,w] = \begin{cases} B[k-1, w] & \text{if } w_k > w \\ \max\{B[k-1,w], B[k-1, w-w_k] + b_k\} & \text{ow} \end{cases}$$

After adding T_1: B[k-1,w-w_k + b_k] is bigger than B[k-1, w] for all weight limits down to 2.

i	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2						
3						
4						

0-1 Knapsack Algorithm Execution

Given the following (benefit, weight) pairs

T_1 T_2 T_3 T_4

(12, 2), (10,1), (20, 3), (15, 2) and W = 5

0-1Knapsack (T, W)
1. for w = 0 to W do
2. B[w] = 0
3. for k = 1 to n do
4. for w = W downto w_k do
5. if B[w – w_k] + b_k > B[w] then
6. B[w] = B[w – w_k] + b_k

$$B[k,w] = \begin{cases} B[k-1, w] & \text{if } w_k > w \\ \max\{B[k-1,w], B[k-1, w-w_k] + b_k\} & \text{ow} \end{cases}$$

After adding T_2: B[k-1,w-w_k + b_k] is bigger than B[k-1, w] for all weight limits down to 1.

i	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0	10	12	22	22	22
3						
4						

0-1 Knapsack Algorithm Execution

Given the following (benefit, weight) pairs

T_1 T_2 T_3 T_4

(12, 2), (10,1), (20, 3), (15, 2) and W = 5

0-1Knapsack (T, W)
1. for w = 0 to W do
2. B[w] = 0
3. for k = 1 to n do
4. for w = W downto w_k do
5. if B[w – w_k] + b_k > B[w] then
6. B[w] = B[w – w_k] + b_k

$$B[k,w] = \begin{cases} B[k-1, w] & \text{if } w_k > w \\ \max\{B[k-1,w], B[k-1, w-w_k] + b_k\} & \text{ow} \end{cases}$$

After adding T_3: B[k-1,w-w_k + b_k] is bigger than B[k-1, w] for all weight limits down to 3.

i	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0	10	12	22	22	22
3	0	10	12	22	30	32
4						

0-1 Knapsack Algorithm Execution

Given the following (benefit, weight) pairs

T_1 T_2 T_3 T_4

(12, 2), (10,1), (20, 3), (15, 2) and W = 5

0-1Knapsack (T, W)
1. for w = 0 to W do
2. B[w] = 0
3. for k = 1 to n do
4. for w = W downto w_k do
5. if B[w – w_k] + b_k > B[w] then
6. B[w] = B[w – w_k] + b_k

$$B[k,w] = \begin{cases} B[k-1, w] & \text{if } w_k > w \\ \max\{B[k-1,w], B[k-1, w-w_k] + b_k\} & \text{ow} \end{cases}$$

After adding T_4: B[k-1,w-w_k + b_k] is bigger than B[k-1, w] for weight limit 5.

i	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0	10	12	22	22	22
3	0	10	12	22	30	32
4	0	10	15	25	30	37

0-1 Knapsack Algorithm Execution

Given the following (benefit, weight) pairs

T_1 T_2 T_3 T_4

(12, 2), (10,1), (20, 3), (15, 2) and W = 5

0-1Knapsack (T, W)
1. for w = 0 to W do
2. B[w] = 0
3. for k = 1 to n do
4. for w = W downto w_k do
5. if B[w – w_k] + b_k > B[w] then
6. B[w] = B[w – w_k] + b_k

$$B[k,w] = \begin{cases} B[k-1, w] & \text{if } w_k > w \\ \max\{B[k-1,w], B[k-1, w-w_k] + b_k\} & \text{ow} \end{cases}$$

The maximum benefit for a weight limit of 5 is the benefit of pairs $\{T_1, T_2, T_4\}$

i	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0	10	12	22	22	22
3	0	10	12	22	30	32
4	0	10	15	25	30	37

0-1 Knapsack Algorithm Execution

Given the following (benefit, weight) pairs

T_1 T_2 T_3 T_4 T_5

(3,2), (5,4), (8,5), (4,3) and (10,9) and W = 20

0-1Knapsack (T, W)
1. for w = 0 to W do
2. B[w] = 0
3. for k = 1 to n do
4. for w = W downto w_k do
5. if B[w – w_k] + b_k > B[w] then
6. B[w] = B[w – w_k] + b_k

$$B[k,w] = \begin{cases} B[k-1, w] & \text{if } w_k > w \\ \max\{B[k-1,w], B[k-1, w-w_k] + b_k\} & \text{ow} \end{cases}$$

1st iteration: $b_k = 3, w_k = 2$

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
0	0	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3

2nd iteration: $b_k = 5, w_k = 4$

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
0	0	3	3	5	5	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8

0-1 Knapsack Algorithm Execution

0-1Knapsack (T, W)
1. for w = 0 to W do
2. B[w] = 0
3. for k = 1 to n do
4. for w = W downto w_k do
5. if B[w – w_k] + b_k > B[w] then
6. B[w] = B[w – w_k] + b_k

Given the following (benefit, weight) pairs
T_1 T_2 T_3 T_4 T_5
(3,2), (5,4), (8,5), (4,3) and (10,9) and W = 20

3rd iteration: $b_k = 8, w_k = 5$

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
0	0	3	3	5	8	8	11	11	13	13	16	16	16	16	16	16	16	16	16	16

4th iteration: $b_k = 4, w_k = 3$

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
0	0	3	4	5	8	8	11	12	13	15	16	17	17	20	20	20	20	20	20	20

5th iteration: $b_k = 10, w_k = 9$

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
0	0	3	4	5	8	8	11	12	13	15	16	17	17	20	20	21	22	23	25	26

0-1 Knapsack Algorithm Execution

0-1Knapsack (T, W)
1. for w = 0 to W do
2. B[w] = 0
3. for k = 1 to n do
4. for w = W downto w_k do
5. if B[w – w_k] + b_k > B[w] then
6. B[w] = B[w – w_k] + b_k

Given the following (benefit, weight) pairs
T_1 T_2 T_3 T_4 T_5
(3,2), (5,4), (8,5), (4,3) and (10,9) and W = 20

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
0	0	3	4	5	8	8	11	12	13	15	16	17	17	20	20	21	22	23	25	26

So this gives us the best possible benefit of a set chosen from T with a total weight of 20, but which subset of T do we use?

0-1 Knapsack Algorithm Execution

0-1Knapsack (T, W)
1. for w = 0 to W do
2. B[w] = 0
3. for k = 1 to n do
4. for w = W downto w_k do
5. if B[w – w_k] + b_k > B[w] then
6. B[w] = B[w – w_k] + b_k
7. **list[w] = list[w – w_k] + k**

Given the following (benefit, weight) pairs
T_1 T_2 T_3 T_4 T_5
(3,2), (5,4), (8,5), (4,3) and (10,9) and W = 20

1st iteration: $b_k = 3, w_k = 2$

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
0	0	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
{1}									…				…							{1}

2nd iteration: $b_k = 5, w_k = 4$

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
0	0	3	3	5	5	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8
{1}	{1}	{2}	{2}	{1,2}				…				…								{1,2}

0-1 Knapsack Algorithm Execution

0-1Knapsack (T, W)
1. for w = 0 to W do
2. B[w] = 0
3. for k = 1 to n do
4. for w = W downto w_k do
5. if B[w – w_k] + b_k > B[w] then
6. B[w] = B[w – w_k] + b_k
7. **list[w] = list[w – w_k] + k**

Given the following (benefit, weight) pairs
T_1 T_2 T_3 T_4 T_5
(3,2), (5,4), (8,5), (4,3) and (10,9) and W = 20

3rd iteration: $b_k = 8, w_k = 5$

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
0	0	3	3	5	8	8	11	11	13	13	16	16	16	16	16	16	16	16	16	16
{1}	{1}	{2}	{3}	{1,2}	{1,3}				{2,3}	{2,3}	{1,2,3}		…							{1,2,3}

4th iteration: $b_k = 4, w_k = 3$

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
0	0	3	4	5	8	8	11	12	13	15	16	17	17	20	20	20	20	20	20	20
{1}	{4}	{2}	{3}	{1,2}	{1,3}	{3,4}	{2,3}	{1,3,4}		{2,3,4}		{1,2,3,4}		…						{1,2,3,4}

0-1 Knapsack Algorithm Execution

0-1Knapsack (T, W)
1. for w = 0 to W do
2. B[w] = 0
3. for k = 1 to n do
4. for w = W downto w_k do
5. if B[w – w_k] + b_k > B[w] then
6. B[w] = B[w – w_k] + b_k
7. **list[w] = list[w – w_k] + k**

Given the following (benefit, weight) pairs
T_1 T_2 T_3 T_4 T_5
(3,2), (5,4), (8,5), (4,3) and (10,9) and W = 20

5th iteration: $b_k = 10, w_k = 9$

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
0	0	3	4	5	8	8	11	12	13	15	16	17	17	20	20	21	22	23	25	26
											{1,2,3}									{1,2,3,5}

This verifies what we can confirm through visual inspection of this small set of (benefit, weight) pairs – that the best value for a set that weighs exactly 20 lbs. is the set {1,2,3,5}.

0-1 Knapsack Algorithm Execution

Given the following (benefit, weight) pairs
T_1 T_2 T_3 T_4 T_5
(3,2), (5,4), (8,5), (4,3) and (10,9) and W = 20

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
0	0	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
0	0	3	3	5	5	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8
0	0	3	3	5	8	8	11	11	13	13	16	16	16	16	16	16	16	16	16	16
0	0	3	4	5	8	8	11	12	13	15	16	17	17	20	20	20	20	20	20	20
0	0	3	4	5	8	8	11	12	13	15	16	17	17	20	20	21	22	23	25	26

0-1 Knapsack Algorithm Execution

```
0-1Knapsack (T, W)
1.  for w = 0 to W do
2.      B[w] = 0
3.  for k = 1 to n  do
4.      for w = W downto w_k do
5.          if B[w – w_k] + b_k > B[w] then
6.              B[w] = B[w – w_k] + b_k
7.              list[w] = list[w – w_k] + k
```

Complexity of 0-1 Knapsack Solution

Running time is dominated by 2 nested for-loops, where the outer loop iterates n times and the inner one iterates at most W times.

The running time of this algorithm is $O(nW)$

The running time of the 0-1Knapsack algorithm depends on a parameter W that is not proportional to the size of the input.

An algorithm whose running time depends on the magnitude of a number given in the input, not the size of the input set, is called a *pseudo-polynomial* time algorithm.

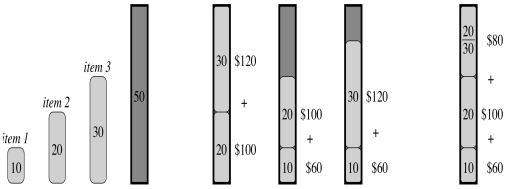
Fractional Knapsack Problem (S. 16.2)

Fractional Knapsack Problem:
Given items $T_1, T_2, T_3, ..., T_n$, with associated weights $w_1, w_2, w_3, ..., w_n$ and benefit values $b_1, b_2, b_3, ..., b_n$, how can we maximize the total benefit subject to an absolute weight limitation W?

We can take an amount x_i of each item i such that $0 \le x_i \le w_i$ for each $i \in T$ and $\sum_{i \in T} x_i \le W$

The total benefit is determined by computing the value per unit weight of each item and sorting by that value

Note that in this problem, unlike the 0-1Knapsack problem, we are allowed to use arbitrary fractions of an item.



Any optimal solution contains items 2 and 3, not 1. So method like Floyd-Warshall would not work here

For fractional knapsack problem, add the items in order of greatest value per pound. Top off with a fraction of lower value item.

Fractional Knapsack Algorithm

Input: Set T of items (such that each item has a positive benefit and a positive weight) and a positive maximum weight value W.
Output: Amount x_i of each item that maximizes the total benefit while not exceeding the maximum total weight W

```
FractionalKnapsack (T, W)
1.  for each item i in T do
2.      x_i = 0
3.      v_i = b_i/w_i    {value index of item}
4.  w = 0
5.  while w < W do
6.      remove from S item i with highest v_i
7.      a = min{w_i, W - w}
8.      x_i = a
9.      w = w + a
```

Fractional Knapsack Algorithm

The running time of the Fractional Knapsack algorithm is $O(n \lg n)$. Why?

This algorithm uses a greedy approach, not a dynamic programming technique, to find the optimal solution. Which other algorithms did we study that use a greedy approach to find an optimal solution?

```
FractionalKnapsack (T, W)
1.  for each item i in T do
2.      x_i = 0
3.      v_i = b_i/w_i    {value index of item}
4.  w = 0
5.  while w < W do
6.      remove from S item i with highest v_i
7.      a = min{w_i, W-w}
8.      x_i = a
9.      w = w + a
```

01 versus Fractional Knapsack Algorithm

Although these problems are similar, the fractional knapsack problem is solvable in polynomial time using a greedy strategy.

If we use the value per pound greedy strategy to make choices in the 0-1 knapsack problem, we end up with a sub-optimal solution.

Longest Common Subsequence Problem (s. 15.4)

Problem: Given $X = < x_1, x_2, ..., x_m >$ and $Y = <y_1, y_2,..., y_n>$, find the longest common subsequence (LCS) of X and Y.

Example:

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

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

$LCS_{XY} = \langle B, C, B, A \rangle$ (or also $LCS_{XY} = \langle B, D, A, B \rangle$)

Brute-Force solution:

1. Enumerate all subsequences of X and check to see if they appear in Y.
2. Each subsequence of X corresponds to a subset of the indices $\{1,2,...,m\}$ of the elements of X – so there are 2^m subsequences of X
3. Clearly, this is not a good approach...time to try dynamic programming!

Recursive Solution to LCS Problem

The recursive LCS Formulation

- Let C[i,j] = length of the LCS of X_i and Y_j, where
- $X_i = \langle x_1, x_2,..., x_i \rangle$ and $Y_j = \langle y_1, y_2,..., y_j \rangle$
- Our goal: C[m,n] (consider entire X and Y)
- Basis: C[0,j] = 0 and C[i,0] = 0
- C[i,j] is calculated as shown below (two cases):

Case 1: $x_i = y_j \ (i, j > 0)$
In this case, we can increase the size of the LCS of X_{i-1} and Y_{j-1} by one by appending $x_i = y_j$ to the LCS of X_{i-1} and Y_{j-1}, i.e.,
$C[i, j] = C[i-1, j-1] + 1$

Case 2: $x_i \neq y_j \ (i, j > 0)$
In this case, we take the LCS to be the longer of the LCS of X_{i-1} and Y_j, and the LCS of X_i and Y_{j-1}, i.e.,
$C[i, j] = \max(C[i, j-1], C[i-1, j])$

Top-Down DP Solution to LCS Problem

- initialize C[i, 0] = C[0 ,j] = 0 for i = 0...m and j = 1...n
- initialize C[i, j] = NIL for i = 1...m and j = 1...n

```
LCS(i, j)
1. if C[i, j] = NIL
2.     if x_i = y_j then
3.         C[i, j] = LCS(i-1, j-1) + 1
4.     else
5.         C[i, j] = max( LCS (i, j-1), LCS (i-1, j))
6. return C[i, j]
```

C is a two-dimensional array holding the solutions to subproblems.

Bottom-Up DP Solution to LCS Problem

We now want to figure out the "right" order to solve the subproblems.

To compute C[i, j], we need the solutions to:
C[i-1, j-1] (when $x_i = y_j$)
C[i-1, j] and C[i, j-1] (when $x_i \neq y_j$)
If we fill in the C array in row major order, these dependencies will be satisfied.

```
LCS(X, Y)
1.  m = length[X]
2.  n = length[Y]
3.  for i = 0 to m do  C[i, 0] = 0
4.  for j = 0 to n  do  C[0, j]=0
5.  for i = 1 to m do
6.      for j = 1 to n do
7.          if x_i = y_j then C[i, j] = C[i-1, j-1] + 1
8.          else C[i, j] = max (C[i, j-1], C[i-1, j])
9.  return C[m, n]
```

Bottom-Up LCS DP

Running time = O(mn) (constant time for each entry in C[ ])

This algorithm finds the value of the LCS, but how can we keep track of the characters in the LCS?

We need to keep track of which neighboring table entry gave the optimal solution to a sub-problem (break ties arbitrarily).
if $x_i = y_j$ the answer came from the upper left (diagonal)
if $x_i \neq y_j$ the answer came from above or to the left, whichever value is larger (if equal, default to above).

Bottom-Up DP Solution to LCS Problem

Idea: Save a pointer to find the path representing the longest common subsequence. Use a 2-dimensional array B[ ] to store the pointers (initially this array will be all NIL).

```
LCS(X, Y)
 1. m = length[X]
 2. n = length[Y]
 3. for i = 0 to m do  C[i, 0] = 0
 4. for j = 0 to n do  C[0, j]=0
 5. for i = 1 to m do
 6.    for j = 1 to n do
 7.        if x_i = y_j then C[i, j] = C[i-1, j-1] + 1
 8.            B[i, j] = "↖  "
 9.        else
10.            if C[i - 1, j]  >= C[i, j - 1] then
11.                C[i, j] = C[i - 1, j]
12.                B[i, j] = " ↑ "
13.            else C[i , j] = C[i, j - 1]
14.                B[i, j] = " ← "
```

Bottom-Up LCS DP

i ⇓ \ j ⇒		0	1 b	2 a	3 b
0		0	0	0	0
1	a	0	↑ 0	↖ 1	← 1
2	b	0	↖ 1	↑ 1	↖ 2
3	b	0	↖ 1	↑ 1	↖ 2
4	a	0	↑ 1	↖ 2	↑ 2

Complexity of LCS Algorithm

The running time of the LCS algorithm is O(mn), since each table entry takes O(1) time to compute.

The running time of the Print-LCS algorithm is O(m + n), since one of m or n is decremented in each stage of the recursion.