Dynamic Programming Introduction

Dynamic programming takes advantage of overlapping subproblems.

Dynamic programming takes advantage of overlapping subproblems.

When a subproblem is solved an algorithm can store the solution.

Dynamic programming takes advantage of overlapping subproblems.

When a subproblem is solved an algorithm can store the solution.

Dynamic programming takes advantage of overlapping subproblems.

When a subproblem is solved an algorithm can store the solution.

Common Examples,

Fibonacci

Dynamic programming takes advantage of overlapping subproblems.

When a subproblem is solved an algorithm can store the solution.

- Fibonacci
- Bellman-Ford

Dynamic programming takes advantage of overlapping subproblems.

When a subproblem is solved an algorithm can store the solution.

- Fibonacci
- Bellman-Ford
- Dijkstra's

Dynamic programming takes advantage of overlapping subproblems.

When a subproblem is solved an algorithm can store the solution.

- Fibonacci
- Bellman-Ford
- Dijkstra's
- Knapsack (and variants)

Dynamic programming takes advantage of overlapping subproblems.

When a subproblem is solved an algorithm can store the solution.

- Fibonacci
- Bellman-Ford
- Dijkstra's
- Knapsack (and variants)
- Floyd-Warshall

Dynamic programming takes advantage of overlapping subproblems.

When a subproblem is solved an algorithm can store the solution.

- Fibonacci
- Bellman-Ford
- Dijkstra's
- Knapsack (and variants)
- Floyd-Warshall
- Sequence Alignment

Dynamic programming takes advantage of overlapping subproblems.

When a subproblem is solved an algorithm can store the solution.

- Fibonacci
- Bellman-Ford
- Dijkstra's
- Knapsack (and variants)
- Floyd-Warshall
- Sequence Alignment
- Counting

Dynamic programming takes advantage of overlapping subproblems.

When a subproblem is solved an algorithm can store the solution.

- Fibonacci
- Bellman-Ford
- Dijkstra's
- Knapsack (and variants)
- Floyd-Warshall
- Sequence Alignment
- Counting
- Matrix Chain Multiplication

$$f_1 = 1$$

$$f_1 = 1; f_2 = 2$$

$$f_1 = 1$$
; $f_2 = 2$; $f_n = f_{n-1} + f_{n-2}$ for $n > 2$

Number sequence

$$f_1 = 1$$
; $f_2 = 2$; $f_n = f_{n-1} + f_{n-2}$ for $n > 2$

Naïve solution,

```
f_1 = 1; f_2 = 2; f_n = f_{n-1} + f_{n-2} for n > 2
Naïve solution,
Function F(N)
    If N < 3
         Return N
    End If
    Return F(N-1) + F(N-2)
End Function
```

Number sequence

$$f_1 = 1$$
; $f_2 = 2$; $f_n = f_{n-1} + f_{n-2}$ for $n > 2$

Naïve solution,

```
Function F(N)
   If N < 3
        Return N
   End If
   Return F(N-1) + F(N-2)
End Function</pre>
```

What is the runtime for computing F(N)?

Why are we recomputing F(3), F(4), ...

Why are we recomputing F(3), F(4), ...

Why are we recomputing F(3), F(4), ...

Main technique (Memoization)

After we determine F(n) store the value.

Why are we recomputing F(3), F(4), ...

- After we determine F(n) store the value.
- If an answer for a value is stored, use it.

Why are we recomputing F(3), F(4), ...

- After we determine F(n) store the value.
- If an answer for a value is stored, use it. (extra base case)

Why are we recomputing F(3), F(4), ...

- After we determine F(n) store the value.
- If an answer for a value is stored, use it. (extra base case)

Why are we recomputing F(3), F(4), ...

- After we determine F(n) store the value.
- If an answer for a value is stored, use it. (extra base case)

```
Initialize answer with INVALID

Function F(N)
    If N < 3
        Return N

End If
    If answer[N] is not INVALID
        Return answer[N]
    End If
    Return answer[N] = F(N-1) + F(N-2)</pre>

End Function
```

Why are we recomputing F(3), F(4), ...

- After we determine F(n) store the value.
- If an answer for a value is stored, use it. (extra base case)

Why are we recomputing F(3), F(4), ...

- After we determine F(n) store the value.
- If an answer for a value is stored, use it. (extra base case)

```
Initialize answer with INVALID

Function F(N)
    If N < 3
        Return N
    End If
    If answer[N] is not INVALID
        Return answer[N]
    End If
    Return answer[N]
    End If
    Return answer[N] = F(N-1) + F(N-2)

End Function

Can we improve the memory usage?</pre>
```

Memoization and recursion is a common dynamic programming approach.

Memoization and recursion is a common dynamic programming approach.

The amount of memory can be large at times.

Memoization and recursion is a common dynamic programming approach.

The amount of memory can be large at times.

Memoization and recursion is a common dynamic programming approach.

The amount of memory can be large at times.

Dynamic programming approaches can be turned into an iterative approach by

Figuring out the sub problem solutions prior to needing them.

Memoization and recursion is a common dynamic programming approach.

The amount of memory can be large at times.

- Figuring out the sub problem solutions prior to needing them.
- Moving in the reverse direction of function returns.

Memoization and recursion is a common dynamic programming approach.

The amount of memory can be large at times.

- Figuring out the sub problem solutions prior to needing them.
- Moving in the reverse direction of function returns.

```
answer[1] = 1; answer[2] = 2;
For i = 3 to N
        answer[i] = answer[i-1] + answer[i-2]
End For
Return answer[N]
```

Memoization and recursion is a common dynamic programming approach.

The amount of memory can be large at times.

- Figuring out the sub problem solutions prior to needing them.
- Moving in the reverse direction of function returns.

```
answer[1] = 1; answer[2] = 2;
For i = 3 to N
         answer[i] = answer[i-1] + answer[i-2]

End For
Return answer[N]

The memory footprint is still O(N)
```

Memoization and recursion is a common dynamic programming approach.

The amount of memory can be large at times.

- Figuring out the sub problem solutions prior to needing them.
- Moving in the reverse direction of function returns.

```
Previous = 1; Current = 2;

For i = 3 to N

    Next = Previous + Current // Compute
    Previous = Current // Update answer[N]
    Current = Next // Update answer[N-1]

End For
Return Current
```

Iterative and Space Saving

Memoization and recursion is a common dynamic programming approach.

The amount of memory can be large at times.

Dynamic programming approaches can be turned into an iterative approach by

- Figuring out the sub problem solutions prior to needing them.
- Moving in the reverse direction of function returns.

```
Previous = 1; Current = 2;

For i = 3 to N

    Next = Previous + Current // Compute
    Previous = Current // Update answer[N]
    Current = Next // Update answer[N-1]

End For
Return Current
```

0-1 Knapsack

We are going on a trip. We have items we can take. We would like to take everything, but we have a limited carrying capacity. Each item has some weight and value AND can only be take once. We want to know what is the largest value sum we can take.

0-1 Knapsack

We are going on a trip. We have items we can take. We would like to take everything, but we have a limited carrying capacity. Each item has some weight and value AND can only be take once. We want to know what is the largest value

sum we can take.

Carry capacity 12

Item	Weight	Value	
Banana	2	3	
Trail Mix	3	6	
Toilet Paper	10	15	
Sleeping Bag	9	11	
Pillow	5	6	
Butane Lighter	1	1	

0-1 Knapsack

We are going on a trip. We have items we can take. We would like to take everything, but we have a limited carrying capacity. Each item has some weight and value AND can only be take once. We want to know what is the largest value

sum we can take.

Carry capacity 12

Item	Weight	Value	
Banana	2	3	
Trail Mix	3	6	
Toilet Paper	10	15	
Sleeping Bag	9	11	
Pillow	5	6	
Butane Lighter	1	1	

What subproblem would be useful to solving this problem?

Subproblem:

Subproblem:

What is the best value sum using a knapsack with capacities up to the maximum?

Subproblem:

What is the best value sum using a knapsack with capacities up to the maximum?

Subproblem:

What is the best value sum using a knapsack with capacities up to the maximum?

In other words,

What is the best knapsack of cap 0?

Subproblem:

What is the best value sum using a knapsack with capacities up to the maximum?

- What is the best knapsack of cap 0?
- What is the best knapsack of cap 1?

Subproblem:

What is the best value sum using a knapsack with capacities up to the maximum?

- What is the best knapsack of cap 0?
- What is the best knapsack of cap 1?
- What is the best knapsack of cap 2?
- What is the best knapsack of cap 3?

Subproblem:

What is the best value sum using a knapsack with capacities up to the maximum?

- What is the best knapsack of cap 0?
- What is the best knapsack of cap 1?
- What is the best knapsack of cap 2?
- What is the best knapsack of cap 3?
- ...
- What is the best knapsack of cap <max>?

Subproblem:

What is the best value sum using a knapsack with capacities up to the maximum?

In other words,

- What is the best knapsack of cap 0?
- What is the best knapsack of cap 1?
- What is the best knapsack of cap 2?
- What is the best knapsack of cap 3?
- ...
- What is the best knapsack of cap <max>?

We also need to know what items we have left to select...

We need to know capacity (either left or used) AND the last item taken.

We need to know capacity (either left or used) AND the last item taken.

The method below is not efficient enough for my tastes...

We need to know capacity (either left or used) AND the last item taken.

The method below is not efficient enough for my tastes...

We need to know capacity (either left or used) AND the last item taken.

The method below is not efficient enough for my tastes...

```
Function K(c, i) // cap left and last item taken
  If seen state c and i // Check if we know the solution
     Return memo table at c and i // Return the known solution
  End If
  Initialize memo table at c and i with 0
  For j = i + 1 to Last item // Try all untaken items
     If can take item j with capacity c // Check the capacity is large enough
        Update memo table at c and i with K(c - weight of j, j) + value of j
     End If
  End For
  Return memo table at c and i
                                         What is the worst runtime?
End Function
                                         The Trick.
```

The 0-1 knapsack problem can be solved faster using better transition times.

The 0-1 knapsack problem can be solved faster using better transition times.

The current runtime for the transition is O(# items).

The 0-1 knapsack problem can be solved faster using better transition times.

The current runtime for the transition is O(# items).

The 0-1 knapsack can transition in O(1).

The 0-1 knapsack problem can be solved faster using better transition times.

The current runtime for the transition is O(# items).

The 0-1 knapsack can transition in O(1). How?

The 0-1 knapsack problem can be solved faster using better transition times.

The current runtime for the transition is O(# items).

The 0-1 knapsack can transition in O(1). How?

Take it or don't,

The 0-1 knapsack problem can be solved faster using better transition times.

The current runtime for the transition is O(# items).

The 0-1 knapsack can transition in O(1). How?

Take it or don't,

Take the current item (item moves to next position; update capacity/value)

The 0-1 knapsack problem can be solved faster using better transition times.

The current runtime for the transition is O(# items).

The 0-1 knapsack can transition in O(1). How?

Take it or don't,

- Take the current item (item moves to next position; update capacity/value)
- Don't take the current item (item moves to next position)

Updated Algorithm

```
Function K(c, i) // cap left and last item taken
  If i is at the end // Check if we cannot add more items
     Return 0 // Return no extra value
  End If
  If seen state c and i // Check if we know the solution
     Return memo table at c and i // Return the known solution
  End If
  Initialize memo table at c and i with K(c, i + 1) // Don't take the item
  Update memo table at c and i with K(c - weight of i, i) + value of i // Take the item
  Return memo table at c and i // Return the found solution
End Function
```

We loop over each item and...

Update an array that represents the best value for each knapsack capacity.

We loop over each item and...

Update an array that represents the best value for each knapsack capacity.

We update by looping over the array of different knapsack capacities and...

Updating the spots using other knapsacks with a lesser capacity.

We loop over each item and...

Update an array that represents the best value for each knapsack capacity.

We update by looping over the array of different knapsack capacities and...

Updating the spots using other knapsacks with a lesser capacity.

Suppose we have the following knapsack answers,

Сар	0	1	2	3	4	5	6
Value	0	1	4	5	5	7	8

We loop over each item and...

Update an array that represents the best value for each knapsack capacity.

We update by looping over the array of different knapsack capacities and...

Updating the spots using other knapsacks with a lesser capacity.

Suppose we have the following knapsack answers, Let's add a weight 2, value 3 to the given list.

Сар	0	1	2	3	4	5	6
Value	0	1	4	5	5	7	8

We loop over each item and...

Update an array that represents the best value for each knapsack capacity.

We update by looping over the array of different knapsack capacities and...

Updating the spots using other knapsacks with a lesser capacity.

Suppose we have the following knapsack answers, Let's add a weight 2, value 3 to the given list.

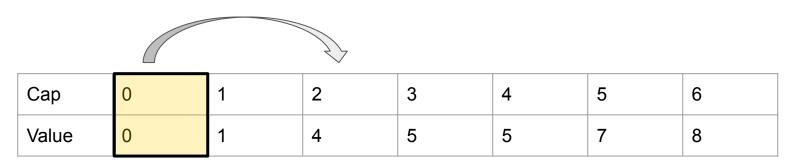
Сар	0	1	2	3	4	5	6
Value	0	1	4	5	5	7	8

Updating in the forward direction...

Moving forward updates... Weight 2 value 3.

Сар	0	1	2	3	4	5	6
Value	0	1	4	5	5	7	8

Moving forward updates... Weight 2 value 3.



Moving forward updates... Weight 2 value 3.

			3				
Сар	0	1	2	3	4	5	6
Value	0	1	4	5	5	7	8

0 + 3 no good

Moving forward updates... Weight 2 value 3.

Сар	0	1	2	3	4	5	6
Value	0	1	4	5	5	7	8

Moving forward updates... Weight 2 value 3.

Сар	0	1	2	3	4	5	6		
Value	0	1	4	5	5	7	8		

1 + 3 no good

Moving forward updates... Weight 2 value 3.

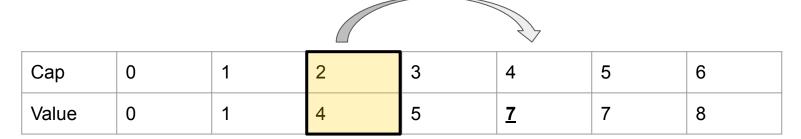
					3		
Сар	0	1	2	3	4	5	6
Value	0	1	4	5	5	7	8

Moving forward updates... Weight 2 value 3.

					\$\frac{1}{2}		
Сар	0	1	2	3	4	5	6
Value	0	1	4	5	5	7	8

4 + 3 good!

Moving forward updates... Weight 2 value 3.



Moving forward updates... Weight 2 value 3.

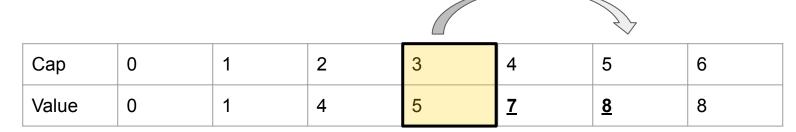
						D	
Сар	0	1	2	3	4	5	6
Value	0	1	4	5	<u>7</u>	7	8

5 + 3

Moving forward updates... Weight 2 value 3.



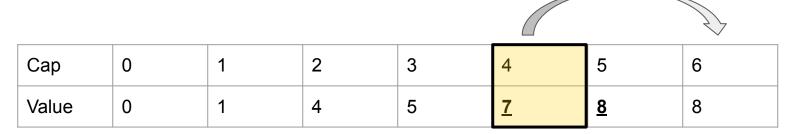
Moving forward updates... Weight 2 value 3.



Moving forward updates... Weight 2 value 3.

							3
Сар	0	1	2	3	4	5	6
Value	0	1	4	5	<u>7</u>	<u>8</u>	8

Moving forward updates... Weight 2 value 3.



7 + 3 crap!

Moving forward updates... Weight 2 value 3.

We actually want to move in the reverse to prevent using an item twice!

Сар	0	1	2	3	4	5	6
Value	0	1	4	5	5	7	8

Moving forward updates... Weight 2 value 3.

We actually want to move in the reverse to prevent using an item twice!

							D
Сар	0	1	2	3	4	5	6
Value	0	1	4	5	5	7	8

Moving forward updates... Weight 2 value 3.

We actually want to move in the reverse to prevent using an item twice!

Сар	0	1	2	3	4	5	6
Value	0	1	4	5	5	7	8

5 + 3 better-ish!

Moving forward updates... Weight 2 value 3.

We actually want to move in the reverse to prevent using an item twice!

						3	
Сар	0	1	2	3	4	5	6
Value	0	1	4	5	5	<u>8</u>	8

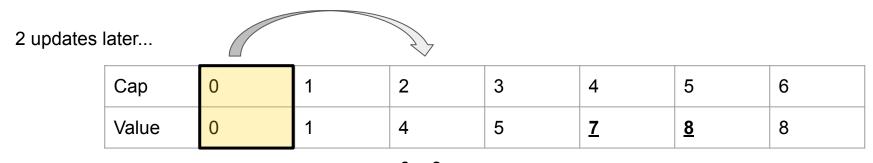
Moving forward updates... Weight 2 value 3.

We actually want to move in the reverse to prevent using an item twice!

Сар	0	1	2	3	4	5	6	
Value	0	1	4	5	<u>7</u>	<u>8</u>	8	

Moving forward updates... Weight 2 value 3.

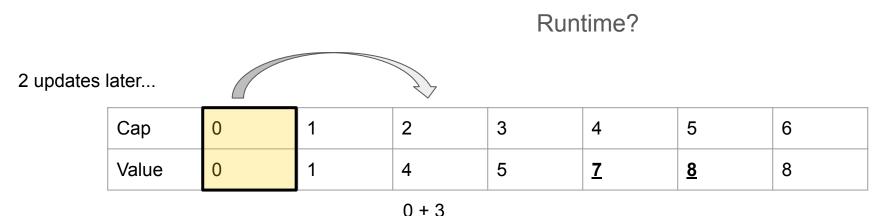
We actually want to move in the reverse to prevent using an item twice!



0 + 3 done!

Moving forward updates... Weight 2 value 3.

We actually want to move in the reverse to prevent using an item twice!



done!

0-∞ knapsack

0-∞ knapsack

Knapsack with items selected any number of times.

0-∞ knapsack

Knapsack with items selected any number of times.

Do the forward updating iterative method!

0-∞ knapsack

Knapsack with items selected any number of times.

Do the forward updating iterative method!

0-k knapsack

0-∞ knapsack

Knapsack with items selected any number of times.

Do the forward updating iterative method!

0-k knapsack

Knapsack with items selected any up to k times.

0-∞ knapsack

Knapsack with items selected any number of times.

Do the forward updating iterative method!

0-k knapsack

Knapsack with items selected any up to k times.

Do the reverse updating iterative method k times for each item!