CSE 421
Introduction to Algorithms

Lecture 12: Dynamic Programming
LIS, Knapsack

Ryan Williams
MIT
- Distinguished Lecture
in this room Thursday
3:30 pm
SAT ⟺ poly time algs
Surprising relationship

Dynamic Programming for Optimization

1. Formulate the *(optimum) value* as a recurrence relation or recursive algorithm

2. Figure out the possible values of parameters in the recursive calls.
   - This should be "small", i.e., bounded by a low-degree polynomial
   - Can use memoization to store a cache of previously computing values

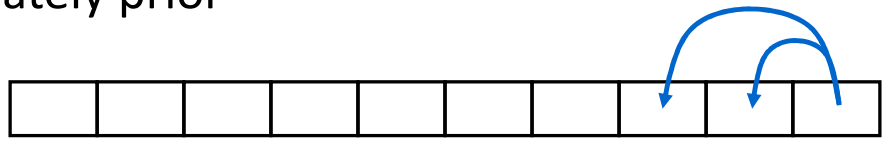
3. Specify an order of evaluation for the recurrence so that you already have the partial results stored in memory when you need them.
   - Produces iterative code
   - Store extra information to be able to reconstruct *optimal solution* and add reconstruction code

Once you have an iterative DP solution: see if you can save space.

Dynamic Programming Patterns

Fibonacci pattern:

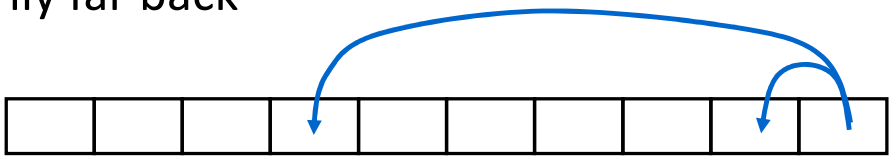
- 1-dimensional, $O(1)$ values immediately prior



- Space saving possible

Weighted interval scheduling pattern:

- 1-dimensional, $O(1)$ values arbitrarily far back

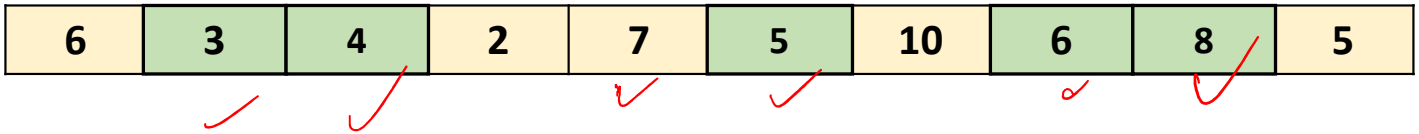


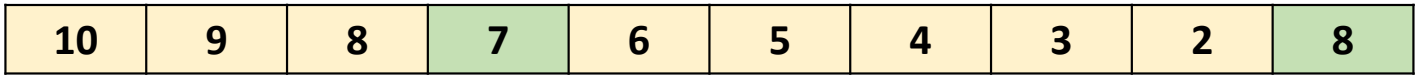
- No space saving possible

Longest Increasing Subsequence (LIS)

Given: An array A of n integers.

Find: A longest possible sequence $i_1 < i_2 < \cdots < i_k$ such that $A[i_1] < A[i_2] < \cdots < A[i_k]$.





Towards Dynamic Programming: Recursive Algorithm

We now want

- a recursive solution that makes calls to smaller problems and
- convenient indices for those smaller problems,

so we first focus on the options for the *last* index n.

Longest Increasing Subsequence: Subproblem structure

Suppose that the longest increasing subsequence ends at n.

- Many possibilities for the next previous element of the sequence
  - Could be any $j < n$.
  - Would be longest increasing subsequence ending at j.
  - Need to check that $A[j] < A[n]$.

Same type of subproblem

If longest increasing subsequence ends earlier, just check above value for ending at smaller j.

Defn: $\mathbf{OPT}(j)$= length of longest increasing subsequence in $A[1..j]$ ending at j

$$\mathbf{OPT}(j) = \begin{cases} 1 & \text{if } j = 1 \\ 1 + \max\{\mathbf{OPT}(i) : 1 \le i < j \text{ and } A[i] < A[j]\} & \text{if } j > 1 \end{cases}$$

$$\mathbf{LIS} = \max\{\mathbf{OPT}(j) : j = 1, \ldots, n\}$$

Bottom-up evaluation order: Increasing value of j

Longest Increasing Subsequence: Algorithm

```
INPUT: n, array A[1..n]

ComputeLISvalue() {
   OPT[1] = 1
   for j = 2 to n {
      OPT[j] = 1
      for i = 1 to j-1 {
         if A[i] < A[j]
            OPT[j] = max(OPT[j], 1+OPT[i])
      }
   }
   LISvalue = 0
   for j = 1 to n
      LIS = max(LISValue,OPT[j])

   return LIS
}
```

Nested loops $O(n^2)$

Longest Increasing Subsequence: Computing the Sequence

```
INPUT: n, array A[1..n]

ComputeLISvalue() {
   OPT[1] = 1
   pred[1] = 0
   for j = 2 to n {
      OPT[j] = 1
      pred[j]= 0
      for i = 1 to j-1 {
         if A[i] < A[j]
             if 1+OPT[i]> OPT[j]{
                OPT[j]=1+OPT[i]
                pred[j]=i}}}
   LISvalue = 0 ; end = 0
   for j = 1 to n
      if OPT[j] > LISvalue {
         LISvalue = OPT[j]
         end = j }

return LISvalue
}
```

Record the preceding position in the sequence that gave the optimum value.

Compute ending index of sequence

Follow pointers backwards to build the sequence.

```
INPUT: n, array A[1..n]

ComputeLIS() {
   j = end
   LIS = A[end]
   while pred[j] > 0 {
         j = pred[j]
         LIS = [A[j]; LIS]
   }

return LIS
}
```

Longest Increasing Subsequence (LIS)

Given: An array A of n integers.

Find: A longest possible sequence $i_1 < i_2 < \cdots < i_k$ such that $A[i_1] < A[i_2] < \cdots < A[i_k]$.

6	3	4	2	7	5	10	6	8	5

OPT[j]	1									
pred[j]	0									
j	1	2	3	4	5	6	7	8	9	10

Longest Increasing Subsequence (LIS)

Given: An array A of n integers.

Find: A longest possible sequence $i_1 < i_2 < \cdots < i_k$ such that $A[i_1] < A[i_2] < \cdots < A[i_k]$.

6	3	4	2	7	5	10	6	8	5

OPT[j]	1	1	2	1						
pred[j]	0	0	2	0						
j	1	2	3	4	5	6	7	8	9	10

Longest Increasing Subsequence (LIS)

Given: An array A of n integers.

Find: A longest possible sequence $i_1 < i_2 < \cdots < i_k$ such that $A[i_1] < A[i_2] < \cdots < A[i_k]$.

6	3	4	2	7	5	10	6	8	5

OPT[j]	1	1	2							
pred[j]	0	0	2							
j	1	2	3	4	5	6	7	8	9	10

Longest Increasing Subsequence (LIS)

Given: An array A of n integers.

Find: A longest possible sequence $i_1 < i_2 < \cdots < i_k$ such that $A[i_1] < A[i_2] < \cdots < A[i_k]$.

6	3	4	2	7	5	10	6	8	5

OPT[j]	1	1	2	1	3	3	4			
pred[j]	0	0	2	0	3	3	5			
j	1	2	3	4	5	6	7	8	9	10

Longest Increasing Subsequence (LIS)

Given: An array A of n integers.

Find: A longest possible sequence $i_1 < i_2 < \cdots < i_k$ such that $A[i_1] < A[i_2] < \cdots < A[i_k]$.

6	3	4	2	7	5	10	6	8	5

OPT[j]	1	1	2	1	3					
pred[j]	0	0	2	0	3					
j	1	2	3	4	5	6	7	8	9	10

Longest Increasing Subsequence (LIS)

Given: An array A of n integers.

Find: A longest possible sequence $i_1 < i_2 < \cdots < i_k$ such that $A[i_1] < A[i_2] < \cdots < A[i_k]$.

6	3	4	2	7	5	10	6	8	5

OPT[j]	1	1	2	1	3	3				
pred[j]	0	0	2	0	3	3				
j	1	2	3	4	5	6	7	8	9	10

Longest Increasing Subsequence (LIS)

Given: An array A of n integers.

Find: A longest possible sequence $i_1 < i_2 < \cdots < i_k$ such that $A[i_1] < A[i_2] < \cdots < A[i_k]$.

6	3	4	2	7	5	10	6	8	5

OPT[j]	1	1	2	1	3	3	4	4		
pred[j]	0	0	2	0	3	3	~~5~~	6		
j	1	2	3	4	5	6	7	8	9	10

Longest Increasing Subsequence (LIS)

Given: An array A of n integers.

Find: A longest possible sequence $i_1 < i_2 < \cdots < i_k$ such that $A[i_1] < A[i_2] < \cdots < A[i_k]$.

6	3	4	2	7	5	10	6	8	5

OPT[j]	1	1	2	1	3	3	4	4	5	3
pred[j]	0	0	2	0	3	3	5	6	8	3
j	1	2	3	4	5	6	7	8	9	10

Longest Increasing Subsequence (LIS)

Given: An array A of n integers.

Find: A longest possible sequence $i_1 < i_2 < \cdots < i_k$ such that $A[i_1] < A[i_2] < \cdots < A[i_k]$.

6	3	4	2	7	5	10	6	8	5

OPT[j]	1	1	2	1	3	3	4	4	5	
pred[j]	0	0	2	0	3	3	5	6	8	
j	1	2	3	4	5	6	7	8	9	10

Longest Increasing Subsequence (LIS)

Given: An array A of n integers.

Find: A longest possible sequence $i_1 < i_2 < \cdots < i_k$ such that $A[i_1] < A[i_2] < \cdots < A[i_k]$.

6	3	4	2	7	5	10	6	8	5

OPT[j]	1	1	2	1	3	3	4	4	5	3
pred[j]	0	0	2	0	3	3	5	6	8	3
j	1	2	3	4	5	6	7	8	9	10

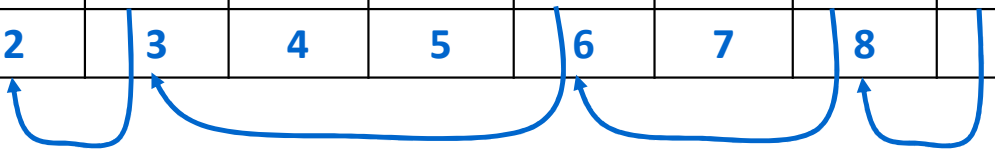
Longest Increasing Subsequence (LIS)

Given: An array A of n integers.

Find: A longest possible sequence $i_1 < i_2 < \cdots < i_k$ such that $A[i_1] < A[i_2] < \cdots < A[i_k]$.

6	3	4	2	7	5	10	6	8	5

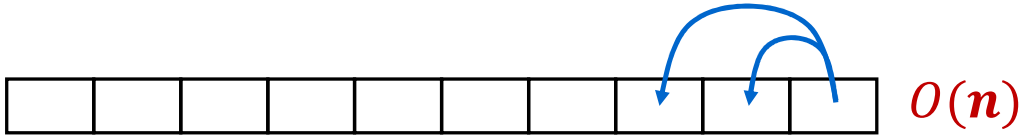
OPT[j]	1	1	2	1	3	3	4	4	5	3
pred[j]	0	0	2	0	3	3	5	6	8	3
j	1	2	3	4	5	6	7	8	9	10



Dynamic Programming Patterns

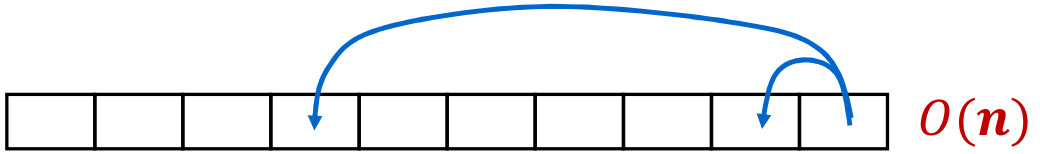
Fibonacci pattern:

- 1-D, $O(1)$ immediately prior
- $O(1)$ space



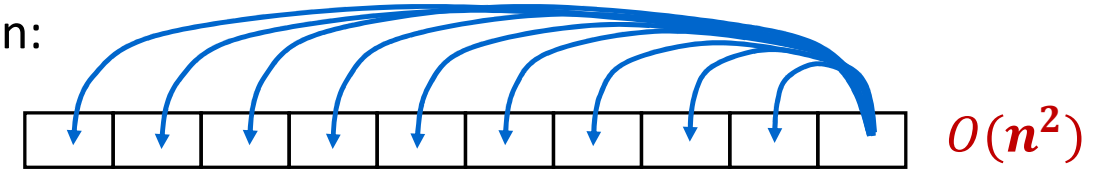
Weighted interval scheduling pattern:

- 1-D, $O(1)$ arbitrary prior
- $O(n)$ space



Longest increasing subsequence pattern:

- 1-D, all $n - 1$ prior
- $O(n)$ space



Knapsack Problem

Knapsack problem:

Given: n objects and a "knapsack"

- Item i weighs $w_i > 0$ kilograms and has value $v_i > 0$.
- Knapsack has capacity of W kilograms.

Goal: Fill knapsack so as to maximize total value.

Example: $\{3, 4\}$ has value 40.

W = 11

#	value	weight	ratio
1	1	1	1
2	6	2	3
3	18	5	3.6
4	22	6	3.666
5	28	7	4

Greedy: Repeatedly add item with maximum ratio v_i / w_i

- Applying greedy adds $\{5, 2, 1\}$ for $28+6+1=35$ total

$\Rightarrow$ greedy is not optimal.

Knapsack Dynamic Programming: False Start

Defn: $\mathbf{OPT}(i)$ = maximum value subset of items $1, \ldots, i$.

Case 1: **OPT** does not select item i.

- **OPT** selects best of $\{1, 2, \ldots, i-1\}$

OPT(i-1)

Case 2: **OPT** selects item i.

- accepting item i does *not* immediately imply that **OPT** will have to reject other items. Items $\{1, 2, \ldots, i-1\}$ might all still be OK.
- without knowing what other items were selected before i, we don't even know if we have enough room for item i!

Conclusion: We need more sub-problems!

Knapsack Dynamic Programming: Adding a New Variable

Defn: $\mathbf{OPT}(i, w)$ = maximum value subset of items $1, \ldots, i$ with weight limit w

Case 1: OPT does not select item i.

- OPT selects best of $\{1, 2, \ldots, i-1\}$ with weight limit w

Case 2: OPT selects item i.

- That gets value v_i but uses up weight w_i out of the limit w
- OPT selects best of $\{1, 2, \ldots, i-1\}$ with weight limit $w - w_i$

$$\mathbf{OPT}(i, w) = \begin{cases} 0 & \text{if } i = 0 \\ \mathbf{OPT}(i-1, w) & \text{if } i > 0 \text{ and } w_i > w \\ \max(\mathbf{OPT}(i-1, w), v_i + \mathbf{OPT}(i-1, w - w_i)) & \text{if } i > 0 \text{ and } w_i \leq w \end{cases}$$

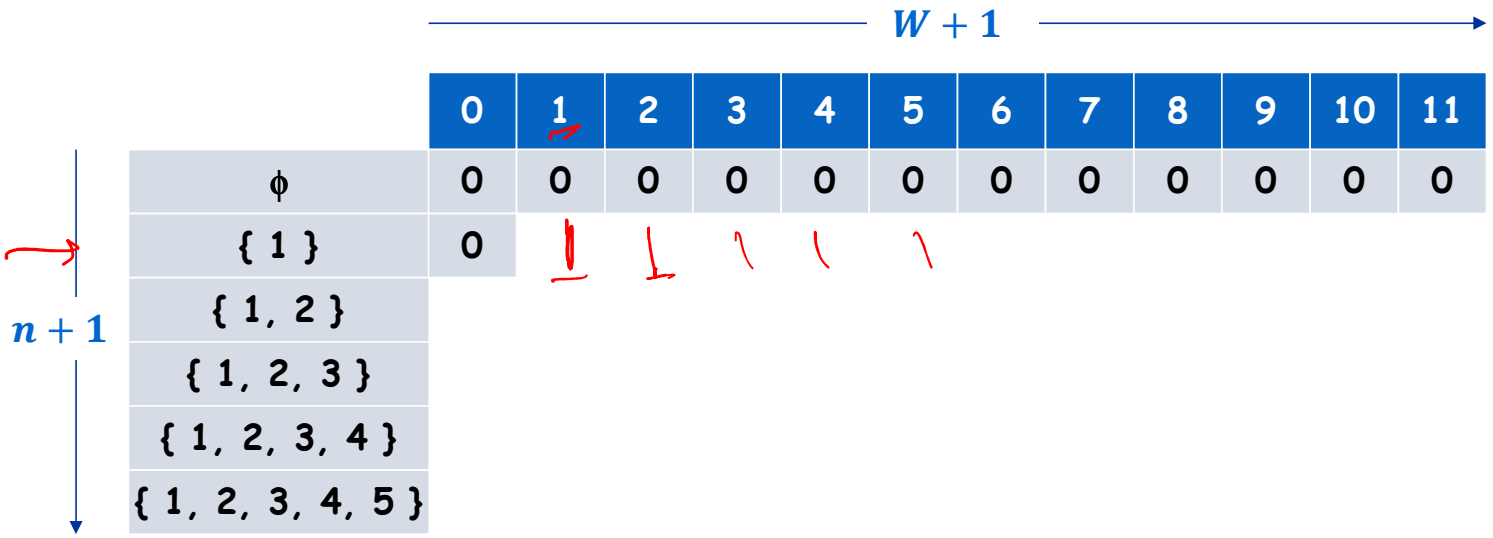
Knapsack Algorithm: Dynamic Programming

Fill up an $n \times W$ array:

```
Input: n, W, w_1,…,w_N, v_1,…,v_N

for w = 0 to W
   OPT[0, w] = 0

for i = 1 to n
   for w = 1 to W
      if (w_i > w)
         OPT[i, w] = OPT[i-1, w]
      else
         OPT[i, w] = max {OPT[i-1, w], v_i + OPT[i-1, w-w_i]}

return OPT[n, W]
```

$O(nW)$ time

Knapsack Algorithm




```
if (w_i > w)
  OPT[i,w] = OPT[i-1,w]
else
  OPT[i,w] = max {OPT[i-1,w], v_i + OPT[i-1,w-w_i]}
```

W = 11

#	value	weight
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

Knapsack Algorithm

$W + 1$

	0	1	2	3	4	5	6	7	8	9	10	11
ϕ	0	0	0	0	0	0	0	0	0	0	0	0
{ 1 }	0	1	1	1	1	1	1	1	1	1	1	1
{ 1, 2 }	0	1	6	7	7							
{ 1, 2, 3 }												
{ 1, 2, 3, 4 }												
{ 1, 2, 3, 4, 5 }												

$n + 1$

```
if (w_i > w)
  OPT[i,w] = OPT[i-1,w]
else
  OPT[i,w] = max {OPT[i-1,w], v_i + OPT[i-1,w-w_i]}
```

W = 11

#	value	weight
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

Knapsack Algorithm

$W + 1$ →

$n + 1$ ↓

	0	1	2	3	4	5	6	7	8	9	10	11
ϕ	0	0	0	0	0	0	0	0	0	0	0	0
{ 1 }	0	1	1	1	1	1	1	1	1	1	1	1
{ 1, 2 }	0	1	6	7	7	7	7	7	7	7	7	7
{ 1, 2, 3 }	0	1	6	7	7	18	19	24	25			
{ 1, 2, 3, 4 }												
{ 1, 2, 3, 4, 5 }												

```
if (w_i > w)
  OPT[i,w] = OPT[i-1,w]
else
  OPT[i,w] = max {OPT[i-1,w], v_i + OPT[i-1,w-w_i]}
```

W = 11

#	value	weight
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

Knapsack Algorithm

$W + 1$ →

$n + 1$ ↓

	0	1	2	3	4	5	6	7	8	9	10	11
ϕ	0	0	0	0	0	0	0	0	0	0	0	0
{ 1 }	0	1	1	1	1	1	1	1	1	1	1	1
{ 1, 2 }	0	1	6	7	7	7	7	7	7	7	7	7
{ 1, 2, 3 }	0	1	6	7	7							
{ 1, 2, 3, 4 }												
{ 1, 2, 3, 4, 5 }												

```
if (w_i > w)
  OPT[i,w] = OPT[i-1,w]
else
  OPT[i,w] = max {OPT[i-1,w], v_i + OPT[i-1,w-w_i]}
```

W = 11

#	value	weight
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

Knapsack Algorithm

$W + 1$ →

$n + 1$ ↓

	0	1	2	3	4	5	6	7	8	9	10	11
ϕ	0	0	0	0	0	0	0	0	0	0	0	0
{ 1 }	0	1	1	1	1	1	1	1	1	1	1	1
{ 1, 2 }	0	1	6	7	7	7	7	7	7	7	7	7
{ 1, 2, 3 }	0	1	6	7	7	18						
{ 1, 2, 3, 4 }												
{ 1, 2, 3, 4, 5 }												

```
if (w_i > w)
  OPT[i,w] = OPT[i-1,w]
else
  OPT[i,w] = max {OPT[i-1,w], v_i + OPT[i-1,w-w_i]}
```

W = 11

#	value	weight
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

Knapsack Algorithm

$W + 1$ →

$n + 1$ ↓

	0	1	2	3	4	5	6	7	8	9	10	11
ϕ	0	0	0	0	0	0	0	0	0	0	0	0
{ 1 }	0	1	1	1	1	1	1	1	1	1	1	1
{ 1, 2 }	0	1	6	7	7	7	7	7	7	7	7	7
{ 1, 2, 3 }	0	1	6	7	7	18						
{ 1, 2, 3, 4 }												
{ 1, 2, 3, 4, 5 }												

```
if (w_i > w)
  OPT[i,w] = OPT[i-1,w]
else
  OPT[i,w] = max {OPT[i-1,w], v_i + OPT[i-1,w-w_i]}
```

W = 11

#	value	weight
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

Knapsack Algorithm

$W + 1$ →

$n + 1$ ↓

	0	1	2	3	4	5	6	7	8	9	10	11
ϕ	0	0	0	0	0	0	0	0	0	0	0	0
{ 1 }	0	1	1	1	1	1	1	1	1	1	1	1
{ 1, 2 }	0	1	6	7	7	7	7	7	7	7	7	7
{ 1, 2, 3 }	0	1	6	7	7	18	19					
{ 1, 2, 3, 4 }												
{ 1, 2, 3, 4, 5 }												

```
if (w_i > w)
  OPT[i,w] = OPT[i-1,w]
else
  OPT[i,w] = max {OPT[i-1,w], v_i + OPT[i-1,w-w_i]}
```

W = 11

#	value	weight
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

Knapsack Algorithm

$W + 1$ →

$n + 1$ ↓

	0	1	2	3	4	5	6	7	8	9	10	11
ϕ	0	0	0	0	0	0	0	0	0	0	0	0
{ 1 }	0	1	1	1	1	1	1	1	1	1	1	1
{ 1, 2 }	0	1	6	7	7	7	7	7	7	7	7	7
{ 1, 2, 3 }	0	1	6	7	7	18	19	24	25	25	25	25
{ 1, 2, 3, 4 }	0	1	6	7	7	18	22	24				
{ 1, 2, 3, 4, 5 }												

```
if (w_i > w)
  OPT[i,w] = OPT[i-1,w]
else
  OPT[i,w] = max {OPT[i-1,w], v_i + OPT[i-1,w-w_i]}
```

W = 11

#	value	weight
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

Knapsack Algorithm

$W + 1$ →

$n + 1$ ↓

	0	1	2	3	4	5	6	7	8	9	10	11
ϕ	0	0	0	0	0	0	0	0	0	0	0	0
{ 1 }	0	1	1	1	1	1	1	1	1	1	1	1
{ 1, 2 }	0	1	6	7	7	7	7	7	7	7	7	7
{ 1, 2, 3 }	0	1	6	7	7	18	19	24	25	25	25	25
{ 1, 2, 3, 4 }	0	1	6	7	7	18	22	24	28			
{ 1, 2, 3, 4, 5 }												

```
if (w_i > w)
  OPT[i,w] = OPT[i-1,w]
else
  OPT[i,w] = max {OPT[i-1,w], v_i + OPT[i-1,w-w_i]}
```

W = 11

#	value	weight
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

Knapsack Algorithm

$W + 1$ →

$n + 1$ ↓

	0	1	2	3	4	5	6	7	8	9	10	11
ϕ	0	0	0	0	0	0	0	0	0	0	0	0
{ 1 }	0	1	1	1	1	1	1	1	1	1	1	1
{ 1, 2 }	0	1	6	7	7	7	7	7	7	7	7	7
{ 1, 2, 3 }	0	1	6	7	7	18	19	24	25	25	25	25
{ 1, 2, 3, 4 }	0	1	6	7	7	18	22	24	28			
{ 1, 2, 3, 4, 5 }												

```
if (w_i > w)
  OPT[i,w] = OPT[i-1,w]
else
  OPT[i,w] = max {OPT[i-1,w], v_i + OPT[i-1,w-w_i]}
```

W = 11

#	value	weight
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

Knapsack Algorithm

$W + 1$ →

$n + 1$ ↓

	0	1	2	3	4	5	6	7	8	9	10	11
ϕ	0	0	0	0	0	0	0	0	0	0	0	0
{ 1 }	0	1	1	1	1	1	1	1	1	1	1	1
{ 1, 2 }	0	1	6	7	7	7	7	7	7	7	7	7
{ 1, 2, 3 }	0	1	6	7	7	18	19	24	25	25	25	25
{ 1, 2, 3, 4 }	0	1	6	7	7	18	22	24	28	29		
{ 1, 2, 3, 4, 5 }												

```
if (w_i > w)
  OPT[i,w] = OPT[i-1,w]
else
  OPT[i,w] = max {OPT[i-1,w], v_i + OPT[i-1,w-w_i]}
```

used(i) = 1

used(i) = 0

W = 11

#	value	weight
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

Knapsack Algorithm

$W + 1$ →

$n + 1$ ↓

	0	1	2	3	4	5	6	7	8	9	10	11
ϕ	0	0	0	0	0	0	0	0	0	0	0	0
{ 1 }	0	1	1	1	1	1	1	1	1	1	1	1
{ 1, 2 }	0	1	6	7	7	7	7	7	7	7	7	7
{ 1, 2, 3 }	0	1	6	7	7	18	19	24	25	25	25	25
{ 1, 2, 3, 4 }	0	1	6	7	7	18	22	24	28	29	29	40
{ 1, 2, 3, 4, 5 }	0	1	6	7	7	18	22	28	29	34	34	40

OPTvalue = 40

```
if (w_i > w)
  OPT[i,w] = OPT[i-1,w]
else
  OPT[i,w] = max {OPT[i-1,w], v_i + OPT[i-1,w-w_i]}
```

W = 11

#	value	weight
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

Knapsack Algorithm – Finding the Optimal Set

1. Keep track of a second $n \times W$ array **Used** such that **Used**$[i, w] = 1$ iff solution contributing to value **OPT**$[i, w]$ includes item i.

   - Walk backwards from $[n, W]$ entry:

     $i \leftarrow n$; $w \leftarrow W$; $K \leftarrow \emptyset$

     while $i > 0$ {

       if **Used**$[i, w] = 1$

         $K \leftarrow K \cup \{i\}$

         $w \leftarrow w - w_i$

         $i \leftarrow i - 1$

     }

2. Instead of extra **Used** array: i is used iff **OPT**$[i, w] >$ **OPT**$[i - 1, w]$.

Knapsack Algorithm

$W + 1$ →

$n + 1$ ↓

	0	1	2	3	4	5	6	7	8	9	10	11
ϕ	0	0	0	0	0	0	0	0	0	0	0	0
{ 1 }	0	1	1	1	1	1	1	1	1	1	1	1
{ 1, 2 }	0	1	6	7	7	7	7	7	7	7	7	7
{ 1, 2, 3 }	0	1	6	7	7	18	19	24	25	25	25	25
{ 1, 2, 3, 4 }	0	1	6	7	7	18	22	24	28	29	29	40
{ 1, 2, 3, 4, 5 }	0	1	6	7	7	18	22	28	29	34	34	40

OPT: {4,3} **OPTvalue** = 22 + 18 = 40

```
if (w_i > w)
  OPT[i,w] = OPT[i-1,w]
else
  OPT[i,w] = max {OPT[i-1,w], v_i + OPT[i-1,w-w_i]}
```

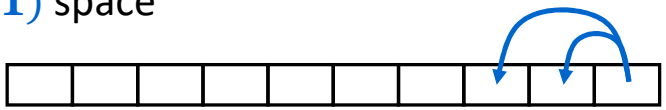
W = 11

#	value	weight
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

Dynamic Programming Patterns

Fibonacci pattern:

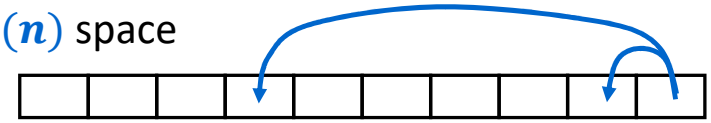
- 1-D, $O(1)$ immediately prior
- $O(1)$ space



$O(n)$

Weighted interval scheduling pattern:

- 1-D, $O(1)$ arbitrary prior
- $O(n)$ space

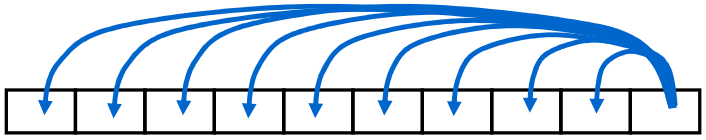


$O(n)$

Longest increasing subsequence pattern:

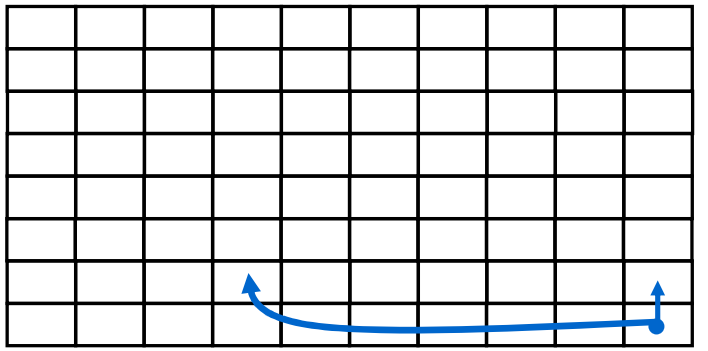
- 1-D, all $n-1$ prior
- $O(n)$ space

$O(n^2)$



Knapsack pattern:

- 2-D, O(1) in previous row, above and arbitrary prior
- $O(nW)$ space



$O(nW)$

- $O(W)$ space if only optimum value needed
  - Maintain current and previous rows

Knapsack Problem: Running Time

Running time is $O(nW)$

- Input size: W and the n weights are representable with only $\log_2 W$ bits
- Not polynomial in input size!
- "Pseudo-polynomial" – polynomial in the # of numbers and largest number
- Decision version of Knapsack is NP-complete.

Knapsack approximation algorithm:

- There exists a polynomial-time algorithm that produces a feasible solution with value within 0.01% of optimum
- Approximation algorithm uses the ideas from the dynamic programming algorithm.