CS 170
Efficient Algorithms and Intractable Problems

Lecture 13
Dynamic Programming II (updated)

Classroom annotations were not saved, so these annotations are added afterwards to help with the study material.

Nika Haghtalab and John Wright

EECS, UC Berkeley

Announcements

Midterm 1 exam:
→ We are still grading. Expect grades by the end of the week!

Homeworks:
→ HW6 is due Wednesday (tomorrow) night
→HW7 will be released later today and due as usual on Monday.

Recap of the Last Lecture

Dynamic Programming!

The recipe!

Step 1. Identify subproblems (aka optimal substructure)
Step 2. Find a recursive formulation for the subproblems
Step 3. Design the Dynamic Programming Algorithm
→ Memo-ize computation starting from smallest subproblems and building up.

We saw a lot of examples already
→ Fibonacci
→ Shortest Paths (in DAGs, Bellman-Ford, and All-Pair)

This lecture

Even more examples!

→ Longest increasing subsequence

→ Edit distance

→ Knapsack

Best way to learn dynamic programming is by doing a lot of examples!

By doing more examples today, we will also develop intuition about how to choose subproblems (Recipe's step 1).

Longest Increasing Subsequences (LIS)

To be consistent with the book, we aren't using 0-indexing for the input.

- Input: An array of n integers $a = [a_1, \dots, a_n]$
- Output: The length of the longest increasing subsequence of the input.

"Subsequences" can be non-contiguous by definition.

$$a = 5 \quad 2 \quad 8 \quad 6 \quad 3 \quad 6 \quad 9 \quad 7$$

Increasing subsequence: 2 3 6 7

Not an increasing subsequence: 2 8 6

The recipe!

Step 1. Identify subproblems (aka optimal substructure)
Step 2. Find a recursive formulation for the subproblems
Step 3. Design the Dynamic Programming Algorithm
→ Memo-ize computation starting from smallest subproblems and building up.

Longest Increasing Subsequence

Why care about this problem?

- An important algorithmic preprocessing step.
- Useful for understanding random processes.

→Shuffle cards and play the game of Solitaire (aka Patience Sorting), how many piles you need?

→Computations over random graphs, networks, social media.

Step 1: Subproblems of LIS

- Input: An array of n integers $a = [a_1, \dots, a_n]$
- Output: The length of the longest increasing subsequence (LIS) of the input.

Discuss

Which of these two subproblems is more appropriate for designing a dynamic programming algorithm?

1. $L[j] = len. of\ LIS\ in\ array\ [a_1, \dots, a_j]$, for $j = 1, \dots, n$
2. $L[j] = len. of\ LIS\ in\ array\ [a_1, \dots, a_j]$ $\boxed{\text{that ends in } a_j}$, for $j = 1, \dots, n$

What makes for good subproblems?

- Not too many of them (the more subproblems the slower the DP algorithm)
- Must have enough information in it to compute subproblems recursively (needed for step 2).

Step 1: Subproblems of LIS

- Input: An array of n integers $a = [a_1, \ldots, a_n]$
- Output: The length of the longest increasing subsequence (LIS) of the input.

Subproblems: $L[j] = len.\,of\ LIS\ in\ array\ [a_1, \ldots, a_j]$ that ends in a_j, for $j = 1, \ldots, n$

→ Because, if we don't keep track of the last (largest) element of the LIS we don't know whether we can add a new element to the subsequence, recursively.

→ Think of the subproblem's stored info as the only thing you observe about smaller instances!

Knowing only Len of LIS, we wouldn't know if we can add 7

len. of LIS = 4

$$a = \boxed{5\quad 2\quad 8\quad 6\quad 3\quad 6\quad 9}\ \ 7$$

Step 1: Subproblems of LIS

- <u>Input</u>: An array of n integers $a = [a_1, \dots, a_n]$
- <u>Output</u>: The length of the longest increasing subsequence (LIS) of the input.

Subproblems: $L[j] = len.\, of\ LIS\ in\ array\ [a_1, \dots, a_j]$ that ends in a_j, for $j = 1, \dots, n$

→ Because, if we don't keep track of the last (largest) element of the LIS we don't know whether we can add a new element to the subsequence, recursively.

→ Think of the subproblem's stored info as the only thing you observe about smaller instances!

Knowing only Len of LIS, we wouldn't know if we can add 7

len. of LIS = 4

$a = \boxed{5\ \ 2\ \ 8\ \ 6\ \ 3\ \ 6\ \ 9}\ \ 7$

len. of LIS ending in 9 = 4

We know for sure, we can't add 7

Step 2: Recurrence of LIS subproblems

- <u>Input</u>: An array of n integers $a = [a_1, \dots, a_n]$
- <u>Output</u>: The length of the longest increasing subsequence (LIS) of the input.

Step 1: $L[j] = len.\,of\ LIS\ in\ array\ [a_1, \dots, a_j]$ that ends in a_j

Step 2: Compute the recurrence: $L[j]$ in terms of $L[i]$ for $i < j$.

Case 1: $a[j] \le a[i]$

$L[i] \qquad a_i \qquad a_j$

$$a = \boxed{\boxed{\dots\ 6\ \ 3\ \ 6\ \ 9}\ \ \dots\ \ 7}\ \ 2\ \ 5$$

Can't add a_j to lengthen $L[i]$

Original class notes had a typo in this case.

Case 2: $a[j] > a[i]$

$L[i] \qquad a_i \qquad a_j$

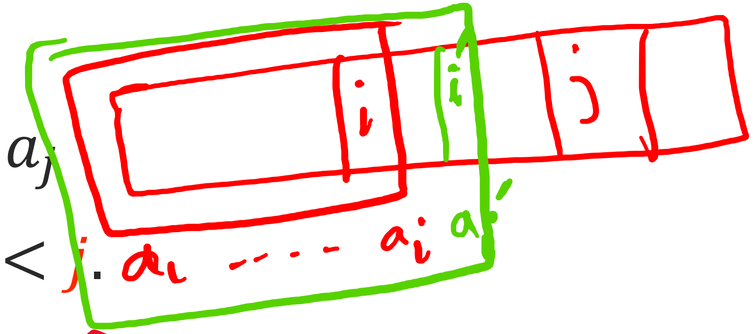
$$a = \boxed{\boxed{\dots\ 6\ \ 3\ \ 6}\ \ \dots\ \ 7}\ \ 2\ \ 5$$

$$L[j] = L[i] + 1$$

Step 2: Recurrence of LIS subproblems

To be consistent with the book, we aren't using 0-indexing for the input.

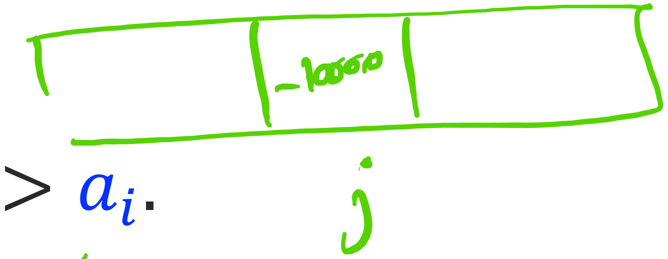
- Input: An array of n integers $a = [a_1, \dots, a_n]$
- Output: The length of the longest increasing subsequence (LIS) of the input.



Step 1: $L[j] = len. of\ LIS\ in\ array\ [a_1, \dots, a_j]$ that ends in a_j

Step 2: Compute the recurrence: $L[j]$ in terms of $L[i]$ for $i < j$.

$$L[j] = \boxed{\max_{i<j}} \{L[i]: a_j > a_i\} + 1$$



$$L[j] = 1, \text{ if } \underbrace{\text{no } i < j \text{ exists with } a_j > a_i}_{}.$$

e.g. a_j might be the min element. Can still serve as a start of a seq.

Step 3: Design the DP Algorithm

To be consistent with the book, we aren't using 0-indexing for the input.

- Input: An array of n integers $a = [a_1, \dots, a_n]$
- Output: The length of the longest increasing subsequence (LIS) of the input.

Runtime:
$O(n)$ subproblems

For each subproblem, we look at at most n smaller subproblems.
→ $O(n)$ time per subproblem.

Total: $O(n^2)$ runtime.

LIS($a_1, \dots a_n$)
  array L of length n
  for $j = 1, \dots, n$
    If exists $i < j$, s.t., $a_i < a_j$
      $L[j] \leftarrow 1 + \max_{i<j} \{ L[i] \mid a_i < a_j \}$
    Else $L[j] \leftarrow 1$
  return $\max_j L[j]$

Edit Distance

Computing the Edit Distance

Input: Two strings $S[1 \dots m]$ and $T[1 \dots n]$

Output: Compute the smallest number of edits to turn S into T.

Edits allowed:

1. Insert a character into S
2. Delete character from S
3. Change one character to another character.

Example:
What's the edit distance between
$S =$ "$SNOWY$" and $T =$ "$SUNNY$"?

S N O W Y

Add U

S U N O W Y

Change O to N

S U N N W Y

Delete W

S U N N Y

Applications of Edit Distance

- Auto correct!
- Word suggestions in search engines
- DNA analysis of similarities.

Edit Distance and Cost of Alignment

Input: Two strings $S[1 \dots m]$ and $T[1 \ \dots \ n]$

Output: Compute the smallest number of edits to turn S into T.

Edit Distance is the minimal cost of alignment between two strings.

→ An alignment: line up two words. Cost of an alignment = # columns that don't match

An alignment with cost 3

S	-	N	O	W	Y
S	U	N	N	-	Y

Add U

Delete W

Change O to N

An alignment with cost 4

S	N	-	O	W	Y
S	U	N	N	-	Y

Step 1: Subproblems of Edit Distance

Input: Two strings $S[1 \dots m]$ and $T[1 \ \dots \ n]$

Output: Compute the smallest number of edits to turn S into T.

What makes for good subproblems?
- Not too many of them (the more subproblems the slower the DP algorithm)
- Must have enough information in it to compute subproblems recursively (needed for step 2).

# of subproblems
$(m+1)(n+1) = O(mn)$

Subproblems: for all $0 \leq i \leq m$ and $0 \leq j \leq n$

$$E(i,j) = EditDist(S[1 \dots i], T[1 \dots j])$$

Cost of optimal alignment between $S[1 \dots i], T[1 \dots j]$

Step 2: Recurrence Relation of Edit Distance

Indicator function $\begin{cases} 1 & S_i \neq T_j \\ 0 & \text{o/w} \end{cases}$

Input: Two strings $S[1 \dots m]$ and $T[1 \dots n]$

Output: Compute the smallest number of edits to turn S into T.

Step 1: $E(i, j) = EditDist(S[1 \dots i], T[1 \dots j])$, for all $0 \leq i \leq m$ and $0 \leq j \leq n$

Discuss

There are three ways of aligning $S[1 \dots i]$ and $T[1 \dots j]$. What are their costs recursively?

Case 1

$S[1 \cdots i-1]$	$S[i]$
$T[1 \cdots j]$	$-$

Case 2

$S[1 \cdots i]$	$-$
$T[1 \cdots j-1]$	$T[j]$

Case 3

$S[1 \cdots i-1]$	$S[i]$
$T[1 \cdots j-1]$	$T[j]$

$$E(i, j) = \min \left\{ E(i-1, j) + 1, \quad E(i, j-1) + 1, \quad E(i-1, j-1) + \mathbb{1}(S_i \neq T_j) \right\}$$

Step 2: Recurrence Relation of Edit Distance

Input: Two strings $S[1 \dots m]$ and $T[1 \ \dots \ n]$

Output: Compute the smallest number of edits to turn S into T.

Step 1: $E(i,j) = EditDist(S[1 \dots i], T[1 \dots j])$, for all $0 \le i \le m$ and $0 \le j \le n$

Step 2: The recurrence relation

$$E(i,j) = \min\{E(i-1,j)+1, E(i,j-1)+1, E(i-1,j-1)+1(S[i] \neq T[j])\}$$

Base case: $E(i,0) = i$ and $E(0,j) = j$

Base Cases one string is empty

Step 3: Design the Algorithm

Input: Two strings $S[1 \ldots m]$ and $T[1 \ \ldots \ n]$

Output: Compute the smallest number of edits to turn S into T.

How do we memo-ize the subproblems in this recurrence relation?

$$E(i,j) = \min\{E(i-1,j)+1, E(i,j-1)+1, E(i-1,j-1)+1(S[i] \neq T[j])\}$$

Base case: $E(i,0) = i$ and $E(0,j) = j$

	0	$\cdots$	$j-1$	j	$\cdots$
0					
$\vdots$					
$i-1$			$E(i-1,j-1)$	$E(i-1,j)$	
i			$E(i,j-1)$	$E(i,j)$	
$\vdots$					

Implicit DAG of the DP indicating the order of computation for subproblems.

Step 3: Design the Algorithm

Input: Two strings $S[1 \dots m]$ and $T[1 \; \dots \; n]$

Output: Compute the smallest number of edits to turn S into T.

How do we memo-ize the subproblems in this recurrence relation?

$$E(i,j) = \min\{E(i-1,j)+1, E(i,j-1)+1, E(i-1,j-1)+1(S[i] \neq T[j])\}$$

Base case: $E(i,0) = i$ and $E(0,j) = j$

	0	$\cdots$	$j-1$	j	$\cdots$
0	0	0 1	j−1	j – – – –	
$\vdots$	1 2				
$i-1$	i−1		$E(i-1,j-1)$	$E(i-1,j)$	
i	i		$E(i,j-1)$	$E(i,j)$	
$\vdots$					

Step 3: Design the Algorithm

<u>Input</u>: Two strings $S[1 \ldots m]$ and $T[1 \ \ldots \ n]$

<u>Output</u>: Compute the smallest number of edits to turn S into T.

How do we memo-ize the subproblems in this recurrence relation?

$$E(i,j) = \min\{E(i-1,j)+1, E(i,j-1)+1, E(i-1,j-1)+1(S[i] \neq T[j])\}$$

Base case: $E(i,0) = i$ and $E(0,j) = j$

	0	$\cdots$	$j-1$	j	$\cdots$
0	0	$1 \quad 2$	$\ldots \quad j-1$	$j \quad \ldots$	$\ldots$
$\vdots$	1				
$i-1$	2 $\vdots$ $i-1$		$E(i-1,j-1)$	$E(i-1,j)$	
i	$\vdots$ i		$E(i,j-1)$	$E(i,j)$	
$\vdots$					

Step 3: Design the Algorithm

Input: Two strings $S[1 \dots m]$ and $T[1 \ \dots \ n]$

Output: Compute the smallest number of edits to turn S into T.

How do we memo-ize the subproblems in this recurrence relation?

$$E(i,j) = \min\{E(i-1,j)+1, E(i,j-1)+1, E(i-1,j-1)+1(S[i] \neq T[j])\}$$

Base case: $E(i,0) = i$ and $E(0,j) = j$

	0	$\cdots$	$j-1$	j	$\cdots$
0					
$\vdots$					
$i-1$			$E(i-1,j-1)$	$E(i-1,j)$	
i			$E(i,j-1)$	$E(i,j)$	
$\vdots$					

Runtime of this algorithm

Input: Two strings $S[1 \ldots m]$ and $T[1 \ \ldots \ n]$

Output: Compute the smallest number of edits to turn S into T.

$O(mn)$ number of subproblems.

For each subproblem, we take minimum of 3 values.
→ Work per subproblem $O(1)$

Total runtime: $O(mn)$.

Edit-Distance($S[1 \ldots m], T[1 \cdots n]$)

$(m+1)\times(n+1)$ array E

For $i = 0, 1, \ldots, m$, $E[i, 0] = i$

For $j = 0, 1, \ldots, n$, $E[0, j] = j$

For $i = 1, \ldots, m$

For $j = 1, \ldots, n$

$$\mathrm{E}(i,j) \leftarrow \min \begin{Bmatrix} E(i-1,j)+1, \\ E(i,j-1)+1\ , \\ E(i-1,j-1)+1(S[i] \neq T[j]) \end{Bmatrix}$$

return $\mathrm{E}(m, n)$

3 Minute Break

(Please close the auditorium door)

Knapsack
(with repetition)

Knapsack

All integers!

Input: A weight capacity W, and n items with (weights, values), $(w_1, v_1), \cdots, (w_n, v_n)$.

Output: Most valuable combination of items, whose total weight is at most W.

Two variants:

1. With repetition (aka unbounded supply, aka with replacement)

→ For each item i, we can take as many copies of it as we want

2. Without repetition (0-1 knapsack, aka without replacement)

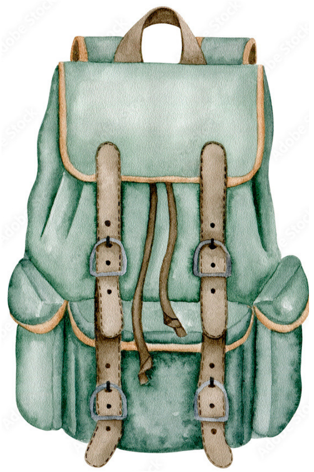
→ For each item, either we take 1 copy or 0 copy of it.

Knapsack

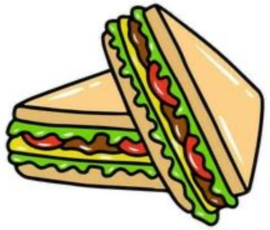
All integers!

Input: A weight capacity W, and n items with (weights, values), $(w_1, v_1), \cdots, (w_n, v_n)$.

Output: Most valuable combination of items, whose total weight is at most W.



W = 10

Item				
Weight:	6	3	4	2
Value:	30	14	16	9

With repetition:
1 tent + 2 sandwiches = **48 value**
Weight =10

Without repetition:
1 tent + 1 stove = **46 value**
Weight =10

Step 1: Subproblems of Knapsack (with repetition)

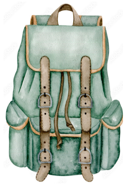
<u>Input</u>: A weight capacity W, and n items $(w_1, v_1), \cdots, (w_n, v_n)$. <u>All integers.</u>

<u>Output</u>: Most valuable combination of items (<u>with repetition</u>), whose total weight is ≤W.

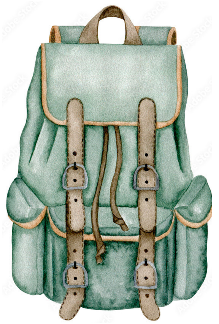
What makes for good subproblems?
- Not too many of them (the more subproblems the slower the DP algorithm)
- Must have enough information in it to compute subproblems recursively (needed for step 2).

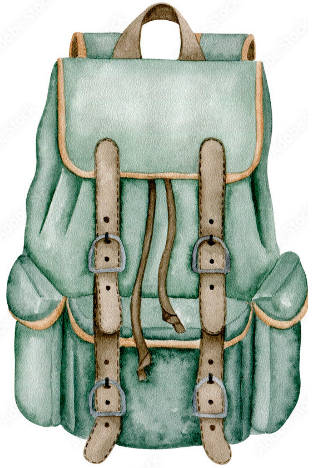
Subproblems: For all $c \leq W$, $K(c) =$ best value achievable for knapsack of capacity c.

First solve the problem
for small knapsacks



Then larger knapsacks





Step 2: Recurrence in Knapsack (with repetition)

<u>Input</u>: A weight capacity W, and n items $(w_1, v_1), \cdots, (w_n, v_n)$. <u>All integers.</u>

<u>Output</u>: Most valuable combination of items (<u>with repetition</u>), whose total weight is ≤W.

Step 1: Subproblems $K(c) =$ best value achievable for knapsack of capacity c, for c $\leq W$.

Step 2:

Let's say we commit to putting a copy of item i for which $w_i \leq c$ in the knapsack

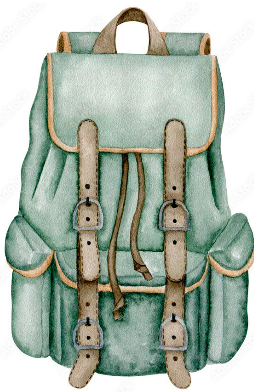
→ Then only $c - w_i$ capacity remains to be optimally packed.

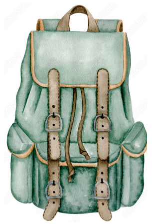
→ The recurrence relationship

$$K(c) = \max_{i: w_i \leq c} \{v_i + K(c - w_i)\}$$

can't be negative

b/c





Step 3: Design the Algorithm

Input: A weight capacity W, and n items $(w_1, v_1), \cdots, (w_n, v_n)$. All integers.

Output: Most valuable combination of items (with repetition), whose total weight is ≤W.

How do we memo-ize the subproblems in this recurrence relation?

$$K(c) = \max_{i: w_i \le c} \{v_i + K(c - w_i)\}$$

Runtime of this algorithm?

Number of subproblems: $O(W)$

Per subproblem, max over $O(n)$ cases
→ $O(n)$ time per subproblem.

Total runtime: $O(nW)$

Knapsack-with-repetition$(W, (w_1, v_1), \dots, (w_n, v_n))$

An array K of size $W + 1$.

$K[0] = 0$

For $c = 1, \dots, W$,

$K[c] = \max_{i: w_i \le c} \{v_i + K(c - w_i)\}$

return $K[W]$

Polynomial time?

We quantify runtimes as functions of input size.

What is the input size of Knapsack?

- Weight capacity W → Needs $O(\log(W))$ bits
- n items with weights at most W (remove any larger item immediately)

→ Each needs at most $O(\log(W))$ bits

- Total input size of knapsack: $O(n\log(W))$

Does the dynamic programming for knapsack run efficiently?

→ Not polynomial time exactly! Runtime $O(nW)$ but input size $O(n\log(W))$

→ Called a pseudo-polynomial time algorithm

→ A runtime that's polynomial in the numerical value of the input (like W) but not in the size of the input.

Preview of Next Lecture

What if we wanted to do knapsack without repetition?

Can we still use the same subproblems $K(c) =$ best value achievable for knapsack of capacity c, for $\mathrm{c} \leq W$?

Challenge: We are only allowed one copy of an item, so the subproblem need to “know” what items we have used and what we haven’t.

We need a different way of tracking subproblems!

Wrap up

More examples of dynamic programming.

- Longest increasing subsequence
- Edit distance
- Knapsack (with repetition)

→ Also got more experience on how to choose subproblems.

Next time: More examples of DP

Knapsack without repetition

Some graph problems