

Dynamic Programming and Shortest Paths

A beginner's textbook for COMP 550 Midterm 2 · Chapters 4 and 5
· UNC Fall 2026

Preface

This book covers the material of COMP 550 Midterm 2: Chapter 4, dynamic programming, and Chapter 5, shortest paths. It is written for a reader who has not been following the lectures and who may be seeing dynamic programming for the first time. Nothing is assumed beyond the ability to read pseudocode and the Midterm 1 material (arrays, breadth-first search, depth-first search, topological ordering, Big-O notation); wherever a Midterm 1 idea is needed it is restated in a sentence or two at the point of use. Every algorithm is presented the same way the lectures present it, in four numbered steps, and every algorithm is run by hand on a concrete input so that you can see what the table of answers looks like. Each chapter ends with exercises whose solutions are hidden until you click, Chapters 6 and 7 are two full practice exams in the format of Midterm 1, and Chapter 8 is a one-page summary for the night before.

The source material is the instructor's handwritten lecture notes for Chapter 4 (five lectures, September 24 to October 13) and Chapter 5 (two lectures, October 20 and 22), together with Midterm 1 and its solutions, which fix the format and the grading conventions. The notes for Chapter 4 Day 2 were damaged in the export and only their first five pages could be recovered; those pages contain the complete treatment of the three topics of that day. Definitions and recurrences are quoted from the notes; explanations that go beyond the notes are marked as such.

About the exam

WHEN. Tuesday, November 3, 2026, in class. Midterm 1 lasted 65 minutes; expect the same.

FORMAT. On paper, closed-everything (no notes, no electronics), assigned seating, bring your One Card. Midterm 1 had five problems worth five points each; expect the same.

SCOPE. Chapters 4 and 5: dynamic programming on arrays, strings, and trees; shortest paths by DAG dynamic programming, Bellman-Ford, and Dijkstra.

Everything known about the format comes from Midterm 1 and from the announcements written in the margins of the lecture notes. On Midterm 1, Problem 1 consisted of five one-point questions in which you run an algorithm by hand on a small input and write only the output, with no partial credit. Problems 2 through 5 each asked you to describe an algorithm with a stated running time, explain why it is correct, and analyze its running time, and at least two of the four were copied word for word from the problem sets. For Midterm 2 the hand-simulations will presumably be things like filling a row of a dynamic-programming table, running Bellman-Ford for a given number of rounds, or listing the order in which Dijkstra's algorithm finalizes vertices, and the design problems will be dynamic programs and shortest-path reductions. Problem Set 5 was due Thursday, October 29.

CONVENTIONS IN FORCE ON THE EXAM AND THROUGHOUT THIS BOOK. Array indices start at 1, and $A[i:j]$ denotes the elements $A[i]$ through $A[j]$. Algorithms are written in pseudocode in the style of the lectures; hashing is not allowed. Graphs are given as adjacency lists with n vertices and m edges; a directed edge from u to v is written (u, v) , and in Chapter 5 it has a length $\ell(u, v)$. On Midterm 1 the following algorithms could be called by name and assumed correct: Max-in-Array, Two-Sum, Binary-Search, Selection-Sort, Merge-Sort, BFS, DFS, Explore, Find-Cycle, Topological-Sort, Prim, Kruskal, Reverse-Delete, Select-Intervals, and Fractional-Knapsack. Expect the Midterm 2 list to add the algorithms of Chapters 4 and 5 by the names used here: LIS, Edit-Distance, Knapsack, MIS-Tree, DAG-DP (shortest paths in a DAG), Bellman-Ford, and Dijkstra.

WHAT THIS BOOK DOES NOT CONTAIN. Problem Sets 4 and 5, Homeworks 4 and 5, the lecture of Tuesday, October 27 (the fourth section of Chapter 5), and the official practice midterm were not in the course export. Because Midterm 1 reused problem-set problems verbatim, you should redo Problem Sets 4 and 5 against the posted solutions in addition to reading this book, and you should work the official practice midterm when it is posted. Section 5.7 says what the fourth section of Chapter 5 is most likely to be.

How solutions are written

Every design problem in this course is posed as an input, a goal, and a target running time, and every solution has three parts: the *algorithm*, a *correctness* argument, and a *running-time* analysis. Chapter 4 adds a fixed way of writing the algorithm part for a dynamic program, which the lectures number 1 through 4: define the *subproblems* (what $\text{OPT}[i]$ means, and which subproblem is the answer), give the *recurrence* with its base cases, write the *pseudocode* that fills a table d in the right order, and state the *running time*, which is almost always the number of subproblems times the time per subproblem. The correctness argument for a dynamic program is an explanation of why the recurrence is right: the optimal solution to a subproblem must fall into one of a few cases, and the recurrence takes the best case. Section 1.7 spells this out, and every worked solution in the book follows it.

What is in scope

Lecture	Date	Core ideas	In this book
Ch 4 Day 1	Thu Sep 24	What dynamic programming is; the four steps; Max-in-Array and Sum-of-Array as DPs; longest increasing subsequence and the "LIS trick"; longest contiguous increasing subsequence	Ch 1, Ch 2
Ch 4 Day 2	Tue Sep 29	Recovering the LIS with parent pointers; longest palindromic subsequence (subproblems on intervals $A[i:j]$); 0/1 knapsack and unbounded knapsack	Ch 2
Ch 4 Day 3	Thu Oct 1	Minimum-cost jumps; bottom-up versus top-down; Fibonacci, pure recursion, memoization; edit distance	Ch 1, Ch 2, Ch 3
Ch 4 Day 4	Thu Oct 8	The six subproblem patterns; edit distance recap; maximum-weight independent set in a tree	Ch 2, Ch 3, Ch 4
Ch 4 Day 5	Tue Oct 13	Saving space; best time to buy and sell a stock; the Alice-and-Bob removal game; maximum-weight matching in a tree	Ch 1, Ch 2, Ch 4
Ch 5 Day 1	Tue Oct 20	Single-source shortest paths; shortest paths in a DAG by DP; longest and widest paths in a DAG; Bellman-Ford	Ch 5
Ch 5 Day 2	Thu Oct 22	Bellman-Ford, space-saving version and edge relaxation; Dijkstra's algorithm; heaps; why longest paths break with cycles	Ch 5
Ch 5 Day 3	Tue Oct 27	Not in the export; see Section 5.7	—

How to read this book

Read Chapter 1 first, slowly; it is short and everything else depends on it. Then read Chapter 2 with pencil and paper, filling in each table yourself before reading the filled-in version. Chapters 3 and 4 extend the method to two strings and to trees and can be read in either order. Chapter 5 is shortest paths; its first section is a dynamic program, so it will feel familiar, and the other two algorithms are small variations. Then take the practice exams in Chapters 6 and 7 under time pressure, and finish with Chapter 8.

Blue boxes contain definitions and recurrences to know by heart. Orange boxes describe mistakes that are easy to make and cost points. Green boxes connect a topic to how it is tested. Grey text marked "extra context" goes beyond the lecture notes.

CH 4 · DAYS 1, 3, 5 1. Dynamic programming: the method

Dynamic programming, or DP, is the main subject of Midterm 2. The name sounds intimidating, but the idea is modest: **to solve a big problem, first write down the answers to all of its smaller versions, in order from smallest to largest, and build each new answer out of the ones already written down.** Everything in Chapter 4, and most of Chapter 5, is this one idea applied to different problems. This chapter teaches the idea on problems so easy that you already know how to solve them, which lets you concentrate on the method itself. Later chapters apply the same method to problems that are genuinely hard without it.

1.1 Boiling versus baking

The lecture opened with an analogy. A greedy algorithm (Chapter 3 of the Midterm 1 material) is like *boiling*: throw things in, pick whatever looks best at each moment, never look back. It is fast and it sometimes works. Dynamic programming is like *baking*: you follow a recipe exactly, you prepare every ingredient before you combine them, and you cannot skip steps, but the result is reliable. In a greedy algorithm you make one choice per step and commit to it. In a dynamic program you refuse to commit: you compute the best answer for *every* smaller version of the problem, and only then read off the answer to the whole thing.

Three words will be used constantly, so here is what they mean.

Optimization problem.

A problem whose answer is the best (largest, smallest, cheapest, longest) of many possible candidates. "Return the maximum element of A" and "return the length of the longest increasing subsequence of A" are optimization problems.

OPT.

The optimal value, that is, the correct answer. In this course OPT is written with square brackets, for example $\text{OPT}[i]$, to mean "the correct answer to subproblem number i." OPT is a mathematical quantity, not a variable in a program.

Subproblem.

A smaller version of the original problem. If the original input is an array A of length n, a typical subproblem is "the same question, but asked only about the first i elements $A[1:i]$." The original problem is the subproblem with $i = n$.

NOTATION. Array indices start at 1, as on every exam in this course. $A[1:i]$ means the elements $A[1], A[2], \dots, A[i]$, and $A[i:n]$ means $A[i], \dots, A[n]$. The phrase " $\forall i: 1 \rightarrow n$ " in the lecture notes means "for every i from 1 up to n." The array that the program fills in is called d (for "dynamic programming table"); the i-th entry $d[i]$ is meant to end up equal to $\text{OPT}[i]$.

1.2 The four-step recipe

Every dynamic programming solution in this course is written in the same four steps, and the exam expects to see them. Memorize the steps now; the rest of the book is practice in carrying them out.

THE FOUR STEPS OF A DP SOLUTION.

1. **Subproblems.** Say, in one English sentence, what $\text{OPT}[i]$ means. Example: "for every i from 1 to n , let $\text{OPT}[i]$ be the maximum element of $A[1:i]$." Also say which subproblem holds the final answer, for example "we will return $\text{OPT}[n]$."
2. **Recurrence.** Give a formula that computes $\text{OPT}[i]$ from the answers to *smaller* subproblems, together with the *base case(s)*, the smallest subproblems whose answers you can state directly without the formula.
3. **Pseudocode.** Write a loop that fills an array d in an order that guarantees that whenever the formula for $d[i]$ is evaluated, the entries it refers to have already been filled in. The lecture summarizes this step as "basically the recurrence, but $\text{OPT} \rightarrow d$."
4. **Running time.** Almost always this is $(\text{number of subproblems}) \times (\text{time to evaluate the recurrence for one subproblem})$.

Step 2 is where all the thinking happens. Steps 1 and 3 are mostly bookkeeping once step 2 is right, and step 4 is usually one sentence. When you are stuck on a DP problem, the question to ask is always the same: *if I already knew the answers to all the smaller subproblems, how would I get the answer to this one?*

1.3 First example: the maximum of an array

You already know how to find the maximum of an array: scan it, remembering the largest value seen so far. The lecture rewrote this familiar algorithm in the four-step format so that the format itself is the only new thing.

Problem. The input is an array A of n numbers. Return the maximum element of A , that is, the maximum of $A[1:n]$.

Step 1, subproblems. For every i from 1 to n , let $\text{OPT}[i]$ be the maximum element of $A[1:i]$. The original problem is $\text{OPT}[n]$, so we will return $\text{OPT}[n]$.

Step 2, recurrence. The base case is $\text{OPT}[1] = A[1]$, because the maximum of a one-element array is that element. For i from 2 to n , the maximum of $A[1:i]$ is either the maximum of $A[1:i-1]$ or the new element $A[i]$, whichever is larger:

RECURRENCE. $\text{OPT}[1] = A[1]$, and for every i from 2 to n , $\text{OPT}[i] = \max(\text{OPT}[i-1], A[i])$.

Why is the recurrence right? The largest element of $A[1:i]$ is somewhere in $A[1:i]$. Either it is $A[i]$ itself, or it is among $A[1], \dots, A[i-1]$, in which case it is the largest of those, which is $\text{OPT}[i-1]$ by definition. Taking the max of the two candidates therefore gives the correct answer. This style of justification, "the answer must come from one of these cases, and the recurrence takes the best case," is what the exam means by *explaining why your algorithm is correct* for a DP problem.

Step 3, pseudocode. Replace OPT by an array d and fill it from left to right, so that $d[i-1]$ is ready when $d[i]$ is computed.

```

ALG(A):                                # DP for Max in Array
    d = [A[1]] * n                      # an array of length n, every entry A[1]
    for i = 2, ..., n:
        d[i] = max(d[i-1], A[i])
    return d[n]

```

Step 4, running time. There are n subproblems and each takes $O(1)$ time (one comparison), so the total is $n \cdot O(1) = O(n)$.

Run it on $A = [3, 1, 4, 1, 5]$: $d[1] = 3$, then $d[2] = \max(3, 1) = 3$, $d[3] = \max(3, 4) = 4$, $d[4] = \max(4, 1) = 4$, $d[5] = \max(4, 5) = 5$, and the algorithm returns $d[5] = 5$. Notice that the array $d = [3, 3, 4, 4, 5]$ records the running maximum; $d[i]$ really is the maximum of $A[1:i]$ for every i , which is exactly what step 1 promised.

1.4 Second example: the sum of an array

The lecture asked the class to repeat the exercise for the sum. **Problem.** Given an array A of n numbers, return the sum of A .

1. **Subproblems.** For every i from 1 to n , $\text{OPT}[i]$ = the sum of $A[1:i]$. We will return $\text{OPT}[n]$.
2. **Recurrence.** $\text{OPT}[1] = A[1]$, and for i from 2 to n , $\text{OPT}[i] = \text{OPT}[i-1] + A[i]$. The sum of the first i elements is the sum of the first $i - 1$ elements plus the i -th element.
3. **Pseudocode.**

```

ALG(A):
    d = [A[1]] * n
    for i = 2, ..., n:
        d[i] = d[i-1] + A[i]
    return d[n]

```

4. **Running time.** $n - 1$ iterations, each $O(1)$, so $O(n)$.

The array d here is the *prefix-sum* array: $d[i]$ is the sum of the first i elements. It will reappear in Chapter 2 as a tool (the Alice-and-Bob game uses suffix sums, the same idea read from the right).

PATTERN 1. Both examples use the same kind of subproblem, which the lecture called Pattern 1: the input is an array A , and for every i from 1 to n , $\text{OPT}[i]$ is the answer to the question asked about the prefix $A[1:i]$. Chapter 2 catalogs five more patterns. When you face a new DP problem, the first thing to try is Pattern 1; when it fails, as it does for the longest increasing subsequence, the way in which it fails tells you which pattern to try next.

1.5 Fibonacci numbers, and three ways to run a recurrence

The Fibonacci sequence is 1, 1, 2, 3, 5, 8, 13, 21, ...: each number is the sum of the two before it. Formally $F(1) = F(2) = 1$ and $F(n) = F(n-1) + F(n-2)$ for $n \geq 3$. The problem is: given a positive integer n , return $F(n)$. Here the recurrence is handed to you by the definition, so this problem isolates a different question: *once you have a recurrence, how should you evaluate it?* The lecture compared three ways.

Bottom-up (the recommended way)

Subproblems: $OPT[i] = F(i)$ for i from 1 to n ; return $OPT[n]$. Base cases $OPT[1] = OPT[2] = 1$; recurrence $OPT[i] = OPT[i-1] + OPT[i-2]$ for $i \geq 3$. Fill d from left to right:

```
ALG(n):                                # Bottom-up
    d = [1] * n
    for i = 3, ..., n:
        d[i] = d[i-1] + d[i-2]
    return d[n]
```

Running time $O(n)$: n subproblems, $O(1)$ each. "Bottom-up" means starting from the smallest subproblems and working up to the big one, which is what every pseudocode in this book does. The lecture recommends writing all exam solutions bottom-up.

Pure recursion (do not do this)

```
ALG(n):                                # Pure recursion: correct but slow
    if n <= 2: return 1
    return ALG(n-1) + ALG(n-2)
```

This computes the right answer, but it is exponentially slow, roughly $O(2^n)$ (more precisely $O(\phi^n)$ with $\phi \approx 1.618$). The reason is repeated work. Drawing the tree of calls for $ALG(6)$: it calls $ALG(5)$ and $ALG(4)$; $ALG(5)$ calls $ALG(4)$ and $ALG(3)$; so $ALG(4)$ is computed twice in full, $ALG(3)$ three times, $ALG(2)$ five times, and so on. Nothing is remembered between calls, so the same subproblems are solved over and over.

Top-down with memoization

Memoization means storing each intermediate result in a global array the first time it is computed, and looking it up instead of recomputing it every later time. Recursion plus memoization is called top-down DP:

```
d = [0] * n                            # global; d[i] = 0 means "not computed yet"
ALG(n):                                # Top-down DP (recursion + memoize)
    if n <= 2: return 1
    if d[n] != 0: return d[n]           # already known
    d[n] = ALG(n-1) + ALG(n-2)
    return d[n]
```

Each $d[n]$ is computed once and then served from the array, so the running time is $O(n)$, the same as bottom-up. Top-down and bottom-up compute exactly the same table; they differ only in the order in which entries get filled. The exam will accept either, but bottom-up pseudocode is shorter, has no recursion to get wrong, and makes the running-time analysis obvious (count the loop iterations), which is why the lecture recommends it.

Extra context beyond the notes: the lecture's tangent on the golden rectangle is the fact that the ratio $F(n+1)/F(n)$ approaches $\phi = (1 + \sqrt{5})/2 \approx 1.618$, the positive solution of $x^2 = x + 1$. It explains where the $O(\phi^n)$ running time of pure recursion comes from, and it is not examinable.

1.6 Saving space

Day 5 of the chapter observed that the Max-in-Array DP stores an entire array d but only ever looks at $d[i-1]$ when computing $d[i]$. So instead of an array, keep a single number:

```
Max(A):                                # space-saving version of the DP
    d = A[1]
    for i = 2, ..., n:
        d = max(d, A[i])
    return d
```

This is the usual Max-in-Array algorithm; the DP version with a full array and the one-variable version do exactly the same comparisons. The general rule is:

WHEN SPACE CAN BE SAVED. The trick applies whenever $d[i]$ does not depend on all of $d[1:i-1]$ but only on a few recent entries (the last one, the last two, or in a two-dimensional table only the previous row). Then only those entries need to be kept. It does not apply when $d[i]$ may depend on every earlier entry, as in the longest increasing subsequence of Chapter 2, whose recurrence takes a max over all $j < i$.

Saving space has two drawbacks, both named in lecture: (1) the table values are harder to interpret, and (2) it is harder to reconstruct the solution itself, not just its value, because there are no stored entries to trace back through, "no parent pointers." For the exam this is a reason to write the full-array version unless a problem specifically asks about space. Chapter 2 shows the same trick on the two-dimensional knapsack table, where it reduces the space from $O(nB)$ to $O(B)$, and Chapter 5 shows it on Bellman-Ford.

1.7 How a DP answer is written on the exam

Design problems on Midterm 1 asked for three things: an algorithm, an explanation of why it is correct, and a running-time analysis. For a DP problem the four steps of Section 1.2 supply all three.

What the problem asks for	What you write
---------------------------------	----------------

Algorithm	Step 1 (the sentence defining $\text{OPT}[i]$ and what is returned), step 2 (recurrence with base cases), step 3 (pseudocode filling d).
Correctness	One short paragraph explaining why the recurrence is right: list the cases the optimal solution to subproblem i could fall into, say why the list is complete, and say why each case is scored correctly by the formula. Mention why the base cases are right and why the final answer is the subproblem you return.
Running time	Number of subproblems $\times$ time per subproblem, plus any precomputation or sorting.

MISTAKES THAT COST POINTS.

- Defining $\text{OPT}[i]$ vaguely (" $\text{OPT}[i]$ is the best answer so far"). The definition must say exactly which smaller input the subproblem is about and what is being optimized.
- Forgetting the base cases, or giving base cases that the recurrence then overwrites.
- Filling d in an order where $d[i]$ refers to an entry that is not yet computed. For Pattern 1 the order is $i = 1, 2, \dots, n$; for suffix subproblems it is $i = n, n - 1, \dots, 1$; for trees it is children before parents; for DAGs it is topological order.
- Returning the wrong entry. With Pattern 1 the answer is $d[n]$; with the "LIS trick" of Chapter 2 it is $\max(d)$, because the best solution need not use the last element.
- Writing the recurrence in terms of OPT but then writing pseudocode that computes something else. The pseudocode should be a literal transcription of the recurrence.

Exercises

1.1. State the four steps of a dynamic programming solution, in order.

(1) Subproblems: define $\text{OPT}[i]$ in words and say which subproblem is returned. (2) Recurrence: a formula for $\text{OPT}[i]$ in terms of smaller subproblems, plus base cases. (3) Pseudocode: fill an array d in an order that respects the recurrence. (4) Running time: number of subproblems $\times$ time per subproblem.

1.2. Give a DP, in the four-step format, that returns the minimum element of A .

Subproblems: $\text{OPT}[i] =$ the minimum of $A[1:i]$, for i from 1 to n ; return $\text{OPT}[n]$. Recurrence: $\text{OPT}[1] = A[1]$; for $i \geq 2$, $\text{OPT}[i] = \min(\text{OPT}[i-1], A[i])$. Pseudocode: $d = [A[1]] * n$; for $i = 2..n$: $d[i] = \min(d[i-1], A[i])$; return $d[n]$. Running time: n subproblems, $O(1)$ each, $O(n)$.

1.3. Give a DP that returns the number of elements of A that are equal to 0.

$\text{OPT}[i] =$ the number of zeros in $A[1:i]$; return $\text{OPT}[n]$. Base case $\text{OPT}[1] = 1$ if $A[1] = 0$ and 0 otherwise. For $i \geq 2$, $\text{OPT}[i] = \text{OPT}[i-1] + 1$ if $A[i] = 0$, and $\text{OPT}[i-1]$ otherwise. Fill d left to right; $O(n)$.

1.4. Run the bottom-up Fibonacci algorithm with $n = 7$ and list the array d .

$d = [1, 1, 2, 3, 5, 8, 13]$; it returns 13.

1.5. In the pure-recursion Fibonacci algorithm, how many times is $\text{ALG}(2)$ called when computing $\text{ALG}(6)$? How many times is $d[2]$ computed in the memoized version?

Counting through the call tree of $\text{ALG}(6)$: $\text{ALG}(5)$ is called once, $\text{ALG}(4)$ twice, $\text{ALG}(3)$ three times, $\text{ALG}(2)$ five times, and $\text{ALG}(1)$ three times. In the memoized version every value is computed at most once: $\text{ALG}(2)$ is a base case that is answered directly, and $d[3]$, $d[4]$, $d[5]$, $d[6]$ are each computed exactly once.

1.6. Why does the space-saving trick apply to the prefix-sum DP of Section 1.4 but not to a recurrence of the form $\text{OPT}[i] = \max \text{ over all } j < i \text{ of something involving } \text{OPT}[j]$?

The prefix-sum recurrence reads only $d[i-1]$, so one stored number suffices. A max over all $j < i$ reads every earlier entry, so all of them must be kept.

1.7. A student defines " $\text{OPT}[i] = \text{the best value}$ " and writes a correct recurrence. What will the grader object to?

The subproblem is not defined: it does not say which smaller input ($A[1:i]$? $A[i:n]$? must $A[i]$ be used?) or which quantity is optimized. Without that sentence the recurrence cannot be checked, and the correctness explanation has nothing to refer to.

CH 4 · DAYS 1–5 2. Dynamic programming on arrays

This chapter contains the array problems from Chapter 4, in the order the lectures introduced them, and ends with the catalog of six subproblem patterns that the lecture of October 8 drew up. The first problem, the longest increasing subsequence, is the most important: it introduces the "LIS trick," a way of choosing subproblems that is reused by half the problems in the chapter.

2.1 Longest increasing subsequence

Subsequence.

A sequence obtained from A by deleting any elements you like (possibly none) and keeping the survivors in their original order. The elements need not be adjacent in A.

Increasing.

Each element is strictly larger than the one before it.

Longest increasing subsequence (LIS).

An increasing subsequence of A of maximum length. There may be several; the problem asks only for the length.

The lecture's example is $A = [5, 1, 2, 7, 4, 6, 3]$. The sequences $[5]$, $[4]$, $[1, 2]$, and $[1, 2, 4]$ are all increasing subsequences. $[3, 6]$ is not a subsequence, because 3 comes after 6 in A. $[7, 4]$ is a subsequence but is not increasing. The longest increasing subsequence is $[1, 2, 4, 6]$, of length 4, so the algorithm should return 4. The target running time is $O(n^2)$.

Why Pattern 1 fails

Try the subproblems of Chapter 1: $\text{OPT}[i]$ = length of the LIS of $A[1:i]$. For the example this gives $\text{OPT} = [1, 1, 2, 3, ?]$. At $i = 5$ the new element is 4. We know that $A[1:4]$ has an increasing subsequence of length 3, namely $[1, 2, 7]$, but we cannot tell from the number 3 whether 4 extends it. It does not extend $[1, 2, 7]$, but it does extend $[1, 2]$, which is shorter. The subproblem answer has thrown away the information we need: *what value the subsequence ends on*.

The LIS trick: Pattern 3

STEP 1, SUBPROBLEMS. For every i from 1 to n , let $\text{OPT}[i]$ be the length of the longest increasing subsequence of $A[1:i]$ *that ends on $A[i]$* , that is, that is required to use $A[i]$ as its last element. We will return $\max(\text{OPT}[1], \dots, \text{OPT}[n])$, because the best subsequence can end anywhere.

The lecture calls the extra requirement the "LIS trick": define the subproblem for $A[1:i]$ but insist that the solution somehow involves $A[i]$. For the example this gives $\text{OPT} = [1, 1, 2, 3, 3, 4, 3]$: the best subsequence ending on 7 is $[1, 2, 7]$ (length 3), the best ending on 4 is $[1, 2, 4]$ (length 3), the best ending on 6 is $[1, 2, 4, 6]$ (length 4), the best ending on the final 3 is $[1, 2, 3]$ (length 3). The answer is the maximum entry, 4, not the last entry.

STEP 2, RECURRENCE. $\text{OPT}[1] = 1$. For i from 2 to n ,

$$\text{OPT}[i] = 1 + \max\{ \text{OPT}[j] : 1 \leq j \leq i - 1 \text{ and } A[j] < A[i] \},$$
where the maximum of an empty set is 0 (if no earlier element is smaller than $A[i]$, then $\text{OPT}[i] = 1$).

Why. Look at the longest increasing subsequence ending on $A[i]$. Either it is just $[A[i]]$, of length 1, or it has an element immediately before $A[i]$; call that element $A[j]$. Then $j < i$, $A[j] < A[i]$ (increasing), and the part of the subsequence up to $A[j]$ is an increasing subsequence ending on $A[j]$, so it has length at most $\text{OPT}[j]$. Conversely, any increasing subsequence ending on $A[j]$ with $A[j] < A[i]$ can be extended by $A[i]$. So the best choice is to pick the j with the largest $\text{OPT}[j]$ among those with $A[j] < A[i]$, and add 1 for $A[i]$ itself.

A second lecture example, with every step shown. $A = [5, 2, 5, 6, 3, 7, 4, 6]$.

i	$A[i]$	candidates j with $A[j] < A[i]$	$\text{OPT}[i]$	a subsequence achieving it
1	5	none	1	[5]
2	2	none	1	[2]
3	5	$j = 2$ (OPT 1)	2	[2, 5]
4	6	$j = 1, 2, 3$ (OPT 1, 1, 2)	3	[2, 5, 6]
5	3	$j = 2$ (OPT 1)	2	[2, 3]
6	7	$j = 1, 2, 3, 4, 5$ (OPT 1, 1, 2, 3, 2)	4	[2, 5, 6, 7]
7	4	$j = 2, 5$ (OPT 1, 2)	3	[2, 3, 4]
8	6	$j = 1, 2, 3, 5, 7$ (OPT 1, 1, 2, 2, 3)	4	[2, 3, 4, 6]

So $\text{OPT} = [1, 1, 2, 3, 2, 4, 3, 4]$ and the algorithm returns 4.

Step 3, pseudocode. Fill d from left to right; for each i , scan all earlier j .

```

LIS(A):
    d = [1] * n
    for i = 2, ..., n:
        for j = 1, ..., i-1:
            if A[j] < A[i] and d[i] < 1 + d[j]:
                d[i] = 1 + d[j]
    return max(d)

```

Step 4, running time. There are n subproblems, and subproblem i scans $i - 1$ earlier entries, so each takes $O(n)$ time; the total is $n \cdot O(n) = O(n^2)$. (Summing $1 + 2 + \dots + (n - 1)$ gives the same answer exactly.)

RETURN MAX(D), NOT D[N]. With the LIS trick, $d[n]$ is the length of the best subsequence that ends on the *last* element, which is usually not the best overall. For $A = [5, 1, 2, 7, 4, 6, 3]$, $d[n] = 3$ but the answer is 4. Any time your subproblem is forced to use $A[i]$, the final answer is a maximum (or minimum) over all i .

Recovering the subsequence itself: parent pointers

Day 2 asked how to output an actual longest increasing subsequence rather than its length. The answer is the same device BFS uses to recover a shortest path: **parent pointers**. Whenever $d[i]$ is set to $1 + d[j]$, also record $p[i] = j$, the index that $d[i]$ was built from. At the end, let i^* be an index with $d[i^*] = \max(d)$; then $i^*, p[i^*], p[p[i^*]], \dots$ lists the chosen indices from last to first. Reverse it to get the subsequence. The lecture's example is $A = [3, 1, 5, 2, 4, 6, 3]$, which gives $d = [1, 1, 2, 2, 3, 4, 3]$ with $p[3] = 1, p[4] = 2, p[5] = 4, p[6] = 5, p[7] = 4$; following the pointers from index 6 gives $6 \rightarrow 5 \rightarrow 4 \rightarrow 2$, that is, the subsequence $[1, 2, 4, 6]$.

2.2 Longest contiguous increasing subsequence

Now require the subsequence to be *contiguous*, a run of adjacent elements. For $A = [3, 1, 2, 4, 2, 5]$ the longest increasing run is $[1, 2, 4]$, so the answer is 3. The LIS trick applies again, and the recurrence becomes simpler because a run ending at $A[i]$ can only be extended from the run ending at $A[i-1]$.

SUBPROBLEMS AND RECURRENCE. $\text{OPT}[i]$ = length of the longest increasing run of $A[1:i]$ that ends on $A[i]$; return $\max(\text{OPT})$. $\text{OPT}[1] = 1$, and for $i \geq 2$, $\text{OPT}[i] = \text{OPT}[i-1] + 1$ if $A[i] > A[i-1]$, and 1 otherwise.

For the example, $\text{OPT} = [1, 1, 2, 3, 1, 2]$. Pseudocode: $d = [1]*n$; for $i = 2..n$: if $A[i] > A[i-1]$: $d[i] = d[i-1] + 1$; return $\max(d)$. Each subproblem now takes $O(1)$, so the running time is $O(n)$. The lesson is that the pattern (Pattern 3) stays the same while the recurrence, and hence the running time, depends on the problem.

2.3 Minimum-cost jumps

Problem (Day 3). The input is an array A of n positive integers, with $n \geq 2$. You start at position 0, just before the array, and want to reach position $n + 1$, just after it. Each jump moves 1 or 2 positions to the right, and landing on position i costs $A[i]$. Return the minimum total cost. For $A = [1, 10, 1]$: jump to position 1 (cost 1), jump two to position 3 (cost 1), jump out; total 2. Target $O(n)$.

Subproblems with the LIS trick: for i from 1 to n , $\text{OPT}[i]$ = the minimum cost of a sequence of jumps that lands on and stops at position i . The lecture asked, as an exercise, why " $\text{OPT}[i] = \min \text{cost given } A[1:i]$ " would not work: to extend a sequence of jumps you need to know where it currently stands, and a subproblem about $A[1:i]$ that does not say where it stops cannot be extended.

RECURRENCE. $\text{OPT}[1] = A[1]$ and $\text{OPT}[2] = A[2]$ (both positions are reachable directly from 0). For i from 3 to n , $\text{OPT}[i] = A[i] + \min(\text{OPT}[i-2], \text{OPT}[i-1])$. The answer is $\min(\text{OPT}[n-1], \text{OPT}[n])$, because the last landing before position $n + 1$ is on n or $n - 1$.

Why. The last jump onto position i came from position $i - 1$ or $i - 2$, and before that jump the path had landed on and stopped at that position with cost at least OPT of it; the cheaper of the two, plus $A[i]$, is optimal.

```
ALG(A):
    d = [0] * n
    d[1] = A[1]; d[2] = A[2]
    for i = 3, ..., n:
        d[i] = A[i] + min(d[i-2], d[i-1])
    return min(d[n-1], d[n])
```

n subproblems, $O(1)$ each: $O(n)$. For $A = [1, 10, 1]$, $d = [1, 10, 2]$ and the answer is $\min(10, 2) = 2$. Since $d[i]$ uses only the two previous entries, the space-saving trick of Section 1.6 applies.

2.4 Best time to buy and sell a stock

Problem (Day 5). $A[i]$ is the price of a stock on day i . You buy on some day i and sell on a later day $j > i$; return the maximum possible profit $A[j] - A[i]$. For $A = [3, 5, 1, 2, 7, 4]$ the answer is $7 - 1 = 6$. Trying all pairs costs $O(n^2)$; the target is $O(n)$.

Subproblems with the LIS trick: for j from 2 to n , $\text{OPT}[j]$ = the maximum profit given $A[1:j]$ *if we must sell on day j* . Then $\text{OPT}[j] = A[j] - \min(A[1:j-1])$: sell at today's price, having bought at the cheapest earlier price. We return $\max(\text{OPT})$. Computing $\min(A[1:j-1])$ from scratch for every j costs $O(j)$ each and $O(n^2)$ in total, which is the trap. The fix is to carry the running minimum along, which is itself a tiny prefix DP (Exercise 1.2).

```
ALG(A):
    min_A = A[1]
    d = [0] * n
    for j = 2, ..., n:
        d[j] = A[j] - min_A
        min_A = min(min_A, A[j])
    return max(d[2], ..., d[n])
```

For the example the running minimum before days 2, ..., 6 is 3, 3, 1, 1, 1, so $d = [-, 2, -2, 1, 6, 3]$ and the answer is 6. Running time $O(n)$. The space-saving version keeps only $\text{ans} = \max(\text{ans}, A[j] - \text{min_A})$ and returns ans .

2.5 Longest palindromic subsequence: Pattern 4

A **palindrome** reads the same forwards and backwards: a, aa, aba, racecar are palindromes; ab is not.

Problem (Day 2). The input is a string A of length n ; return the length of the longest subsequence of A that is a palindrome. For $A = \text{acbcabda}$ the answer is 5, achieved by acbca (positions 1, 2, 3, 4, 7) and also by abcba (positions 1, 3, 4, 5, 7). Target $O(n^2)$.

Prefix subproblems do not work here, because a palindrome is built from both ends inward: deciding whether the first and last characters can be matched requires a subproblem that has a first *and* a last character. The lecture's Pattern 4 uses subproblems indexed by an interval.

STEP 1. For every pair $1 \leq i \leq j \leq n$, let $\text{OPT}[i][j]$ be the length of the longest palindromic subsequence of $A[i:j]$. We will return $\text{OPT}[1][n]$. By convention $\text{OPT}[i][j] = 0$ when $i > j$ (the empty string).

STEP 2. Base case: $\text{OPT}[i][i] = 1$ (a single character is a palindrome). For $i < j$:

- if $A[i] = A[j]$: $\text{OPT}[i][j] = 2 + \text{OPT}[i+1][j-1]$;
- otherwise: $\text{OPT}[i][j] = \max(\text{OPT}[i+1][j], \text{OPT}[i][j-1])$.

Why. Consider the longest palindromic subsequence of $A[i:j]$. If it uses both $A[i]$ and $A[j]$, those two characters are its first and last and must be equal, and what lies between them is a palindromic subsequence of $A[i+1:j-1]$, of length at most $\text{OPT}[i+1][j-1]$; conversely any palindromic subsequence of $A[i+1:j-1]$ can be wrapped in the matching pair, gaining 2. If it does not use $A[i]$, it is a palindromic subsequence of $A[i+1:j]$; if it does not use $A[j]$, it is one of $A[i][j-1]$. When $A[i] = A[j]$ the first option is at least as good as the others, which is why the recurrence takes it directly. (Edge case: when $j = i + 1$ and the two characters match, $2 + \text{OPT}[i+1][i] = 2 + 0 = 2$, using the convention for an empty interval.)

The table and the order of filling it

Rows are i , columns are j , and only cells with $i \leq j$ are used (the upper triangle). The cell (i, j) depends on $(i+1, j-1)$, the cell diagonally below-left, $(i+1, j)$, the cell below, and $(i, j-1)$, the cell to the left. So the rows must be filled *from the bottom row upward*, each row from left to right. Here is the table for $A = \text{acbcabda}$, with the diagonal of 1s as the base case:

$i \setminus j$	1: a	2: c	3: b	4: c	5: b	6: d	7: a
1: a	1	1	1	3	3	3	5
2: c		1	1	3	3	3	3
3: b			1	1	3	3	3
4: c				1	1	1	1
5: b					1	1	1
6: d						1	1

Some cells worked out: (2, 4) is cbc; $c = c$, so $2 + \text{OPT}[3][3] = 3$. (1, 4) is acbc; $a \neq c$, so $\max(\text{OPT}[2][4], \text{OPT}[1][3]) = \max(3, 1) = 3$. (2, 6) is cbc bd; $c \neq d$, so $\max(\text{OPT}[3][6], \text{OPT}[2][5]) = \max(3, 3) = 3$. (1, 7), the answer: $a = a$, so $2 + \text{OPT}[2][6] = 5$. The lecture's second example, $A = \text{abaca}$, gives first row $[1, 1, 3, 3, 3]$ and answer 3 (aba or aca).

```
LPS(A):
    d = n x n table, all 0
    for i = 1, ..., n: d[i][i] = 1
    for i = n-1, ..., 1:                # bottom row first
        for j = i+1, ..., n:          # left to right
            if A[i] = A[j]: d[i][j] = 2 + d[i+1][j-1]
            else: d[i][j] = max(d[i+1][j], d[i][j-1])
    return d[1][n]
```

Running time. About $n^2/2$ subproblems, $O(1)$ each: $O(n^2)$.

INTERVAL SUBPROBLEMS. Whenever the input is a sequence and the natural way to shrink it is to peel off an element from *either* end, use Pattern 4: $\text{OPT}[i][j]$ for $A[i:j]$, filled by increasing length of the interval (equivalently, bottom row up). The lecture left the longest palindromic *substring* (contiguous) as an exercise; it is Exercise 2.8.

2.6 The 0/1 knapsack problem: Pattern 5

Problem (Day 2). There are n items; item i has value $v[i]$ and weight $w[i]$, all positive integers, and the knapsack has capacity B , a positive integer. Choose a set of items of total weight at most B so that the total value is as large as possible, and return that value. Each item is either taken whole or not at all, hence "0/1." For $v = [4, 2, 3, 5]$, $w = [3, 1, 2, 2]$, $B = 5$ the answer is 10, from items 2, 3, 4 (weight $1 + 2 + 2 = 5$, value $2 + 3 + 5$).

This is the problem that the greedy "best value per unit weight" rule of Midterm 1 solves only when items may be taken fractionally. For whole items greedy fails (Midterm 1 had the counterexample $v = [4, 4, 7]$, $w = [5, 5, 6]$, $B = 10$). The dynamic program succeeds because its subproblems record not only which items have been considered but also how much capacity is left.

STEP 1. For every i from 1 to n and every j from 0 to B , let $\text{OPT}[i][j]$ be the maximum value obtainable using only items 1 through i with a knapsack of capacity j . We will return $\text{OPT}[n][B]$.

STEP 2. Base cases: $\text{OPT}[i][0] = 0$ for every i (no capacity), and $\text{OPT}[1][j] = v[1]$ if $w[1] \leq j$ and 0 otherwise. For i from 2 to n and j from 1 to B :

- if $w[i] > j$: $\text{OPT}[i][j] = \text{OPT}[i-1][j]$ (item i does not fit);
- otherwise: $\text{OPT}[i][j] = \max(\text{OPT}[i-1][j], v[i] + \text{OPT}[i-1][j - w[i]])$.

Why. An optimal solution for items $1..i$ and capacity j either does not use item i , in which case it is an optimal solution for items $1..i-1$ and capacity j , or uses item i , in which case the remaining items form an optimal solution for items $1..i-1$ with the capacity $j - w[i]$ that is left over, and the value gains $v[i]$. The second option is only available when $w[i] \leq j$. The recurrence takes the better of the two.

Rows are items, columns are capacities. Each cell looks at the cell directly above it and at the cell in the row above, $w[i]$ columns to the left.

$i \ (v, w) \setminus j$	0	1	2	3	4	5
1 (4, 3)	0	0	0	4	4	4
2 (2, 1)	0	2	2	4	6	6
3 (3, 2)	0	2	3	5	6	7
4 (5, 2)	0	2	5	7	8	10

For instance $\text{OPT}[2][4] = \max(\text{OPT}[1][4], 2 + \text{OPT}[1][3]) = \max(4, 6) = 6$, and $\text{OPT}[4][5] = \max(\text{OPT}[3][5], 5 + \text{OPT}[3][3]) = \max(7, 10) = 10$.

```
Knapsack(v, w, B):
    d = n x (B+1) table, all 0          # columns 0..B
    for j = 1, ..., B: if w[1] <= j: d[1][j] = v[1]
    for i = 2, ..., n:
        for j = 1, ..., B:
            d[i][j] = d[i-1][j]
            if w[i] <= j: d[i][j] = max(d[i][j], v[i] + d[i-1][j - w[i]])
    return d[n][B]
```

Running time. $n(B + 1)$ subproblems, $O(1)$ each: $O(nB)$. Extra context beyond the notes: $O(nB)$ depends on the numerical value of B , not only on the number of items, so it is fast when the capacity is small and slow when it is huge; this is sometimes called pseudo-polynomial. The exam only needs $O(nB)$.

Unbounded knapsack

Day 2's second problem: each item may be taken any number of times. Two changes. The base row becomes $\text{OPT}[1][j] = k \cdot v[1]$ where k is the largest integer with $k \cdot w[1] \leq j$ (take as many copies of item 1 as fit). And in the "take item i " option, item i stays available afterwards, so $\text{OPT}[i-1][j - w[i]]$ becomes $\text{OPT}[i][j - w[i]]$: $\text{OPT}[i][j] = \max(\text{OPT}[i-1][j], v[i] + \text{OPT}[i][j - w[i]])$. Still $O(nB)$.

Saving space

Day 5 pointed out that row i of the 0/1 table depends only on row $i - 1$, so two arrays of length $B + 1$ suffice in place of the $n \times (B + 1)$ table, reducing space from $O(nB)$ to $O(B)$. Even one array works, provided that within each row j is processed *from B down to 0*: $d[j]$ needs the old value of $d[j - w[i]]$, which has not been overwritten yet when moving right to left. Processing left to right would instead use the already-updated $d[j - w[i]]$, which allows item i to be taken again, and computes the unbounded knapsack. This is the lecture's one-line summary: right to left solves 0/1; left to right solves unbounded.

2.7 The Alice-and-Bob game: Pattern 2

Problem (Day 5). The input is an array A of $n \geq 3$ positive integers. Alice and Bob alternate turns, Alice first. On a turn the player removes 1, 2, or 3 numbers from the *left* end of A and adds them to their score. Both play optimally, each maximizing their own total. Return Alice's maximum possible score. For $A = [4, 1, 5, 2, 3, 10, 1, 2]$ the answer is 18. Target $O(n)$.

The game is always played on a *suffix* $A[i:n]$ of the original array, which suggests Pattern 2: subproblems indexed by suffixes.

STEP 1. For every i from 1 to n , let $OPT[i]$ be the maximum score that the player whose turn it is can obtain from the remaining array $A[i:n]$. We will return $OPT[1]$.

Let $S[i] = A[i] + A[i+1] + \dots + A[n]$ be the suffix sum, with $S[n+1] = 0$. If the current player takes k numbers ($k = 1, 2, 3$), they score $A[i] + \dots + A[i+k-1] = S[i] - S[i+k]$, and then the opponent moves on $A[i+k:n]$ and, playing optimally, gets $OPT[i+k]$ out of the $S[i+k]$ that remains; everything the opponent does not take eventually comes back to the current player, who therefore also gets $S[i+k] - OPT[i+k]$.

STEP 2. Base cases: for $i = n - 2, n - 1, n$, $OPT[i] = S[i]$ (take everything that is left). For $i \leq n - 3$,

$$OPT[i] = \max(A[i] + S[i+1] - OPT[i+1], A[i] + A[i+1] + S[i+2] - OPT[i+2], A[i] + A[i+1] + A[i+2] + S[i+3] - OPT[i+3]).$$

Since $A[i] + S[i+1] = S[i]$ and so on, every option equals $S[i] - OPT[i+k]$, so $OPT[i] = S[i] - \min(OPT[i+1], OPT[i+2], OPT[i+3])$.

The table is filled from $i = n$ down to 1, because each subproblem needs larger indices. The suffix sums S are precomputed in $O(n)$ by $S[n] = A[n]$, $S[i] = A[i] + S[i+1]$; computing each sum from scratch inside the recurrence would make the algorithm $O(n^2)$, which the lecture called the naive version.

i	1	2	3	4	5	6	7	8
A[i]	4	1	5	2	3	10	1	2
S[i]	28	24	23	18	16	13	3	2
OPT[i]	18	14	10	15	14	13	3	2

For example $OPT[5] = 16 - \min(OPT[6], OPT[7], OPT[8]) = 16 - 2 = 14$ (take 3, 10, 1 and leave Bob the 2), and $OPT[1] = 28 - \min(14, 10, 15) = 18$ (Alice takes 4 and 1, leaving Bob the position worth only 10 of the remaining 23).

```

ALG(A):
  S = [0] * (n+1); S[n] = A[n]
  for i = n-1, ..., 1: S[i] = A[i] + S[i+1]
  d = [0] * n
  d[n] = S[n]; d[n-1] = S[n-1]; d[n-2] = S[n-2]
  for i = n-3, ..., 1:
    d[i] = S[i] - min(d[i+1], d[i+2], d[i+3])
  return d[1]

```

Running time. $O(n)$ to compute S , then n subproblems at $O(1)$ each: $O(n)$.

2.8 The six subproblem patterns

On October 8 the lecture collected the kinds of subproblems seen so far. The three starred ones were singled out as the most important.

Pattern	Input	Subproblem	Used by	Fill order	Answer
1	A	$\text{OPT}[i]$ = answer for the prefix $A[1:i]$	Max-in-Array, Sum, Fibonacci	$i = 1 \dots n$	$\text{OPT}[n]$
2	A	$\text{OPT}[i]$ = answer for the suffix $A[i:n]$	Alice-and-Bob game	$i = n \dots 1$	$\text{OPT}[1]$
3	A	$\text{OPT}[i]$ = answer for $A[1:i]$ that must involve $A[i]$ (the LIS trick)	LIS, LCIS, jumps, stock	$i = 1 \dots n$	max or min over i
4	A	$\text{OPT}[i][j]$ = answer for the interval $A[i:j]$	Longest palindromic subsequence	bottom row up, or by interval length	$\text{OPT}[1][n]$
5	(A, k)	$\text{OPT}[i][j]$ = answer for $(A[1:i], j)$, j a budget from 0 to k	0/1 knapsack, unbounded knapsack	row by row	$\text{OPT}[n][k]$
6	(A, B)	$\text{OPT}[i][j]$ = answer for the pair of prefixes $(A[1:i], B[1:j])$	Edit distance (Chapter 3)	row by row	$\text{OPT}[m][n]$

HOW TO CHOOSE A PATTERN ON THE EXAM. Start with Pattern 1. If the prefix answer cannot be extended because you do not know how the solution ends, add the LIS trick (Pattern 3) and remember to return the max or min over all i . If the problem is naturally played from the right end, use suffixes (Pattern 2). If a solution is built from both ends of a sequence, use intervals (Pattern 4). If there is a second numerical resource such as a capacity, add it as a second index (Pattern 5). If there are two sequences, index by a prefix of each (Pattern 6). Trees have their own pattern, Chapter 4.

Exercises

2.1. Run LIS on $A = [2, 1, 3, 5, 4, 7, 3]$ and give the array d and the returned value.

$d = [1, 1, 2, 3, 3, 4, 2]$; the algorithm returns 4 (for example $[1, 3, 5, 7]$ or $[1, 3, 4, 7]$). This is the lecture's own Day 2 example.

2.2. Explain in two sentences why the subproblem " $\text{OPT}[i] = \text{length of the LIS of } A[1:i]$ " does not lead to a recurrence.

To decide whether $A[i]$ extends the best subsequence of $A[1:i-1]$ you need to know the last value of that subsequence, and a shorter subsequence with a smaller last value might be the one that $A[i]$ extends. The number $\text{OPT}[i-1]$ does not contain that information, so $\text{OPT}[i]$ cannot be computed from it.

2.3. Give an $O(n^2)$ DP for the length of the longest *decreasing* subsequence, and an $O(n)$ DP for the longest contiguous run of *equal* elements.

Decreasing: same as LIS with the condition $A[j] > A[i]$; $d = [1]*n$; for i : for $j < i$: if $A[j] > A[i]$: $d[i] = \max(d[i], 1 + d[j])$; return $\max(d)$; $O(n^2)$. Equal run: $d[i] = d[i-1] + 1$ if $A[i] = A[i-1]$ else 1; return $\max(d)$; $O(n)$.

2.4. Run the minimum-cost-jump algorithm on $A = [2, 7, 3, 8, 1, 4]$ and give d and the answer.

$d[1] = 2$, $d[2] = 7$, $d[3] = 3 + \min(2, 7) = 5$, $d[4] = 8 + \min(7, 5) = 13$, $d[5] = 1 + \min(5, 13) = 6$, $d[6] = 4 + \min(13, 6) = 10$. Answer $\min(d[5], d[6]) = 6$: land on 2, 3, 1, then jump out.

2.5. (Maximum-sum contiguous subarray.) The input is an array A of n integers, possibly negative. Describe an $O(n)$ -time algorithm that returns the largest possible sum of a contiguous nonempty subarray $A[i:j]$, explain why it is correct, and analyze its running time. Example: $A = [2, -5, 3, -1, 4] \rightarrow 6$.

LIS trick. Subproblems: $\text{OPT}[i]$ = the maximum sum of a contiguous subarray that ends at position i ; return $\max(\text{OPT})$. Base $\text{OPT}[1] = A[1]$. Recurrence: $\text{OPT}[i] = A[i] + \max(\text{OPT}[i-1], 0)$: the subarray ending at i either is just $[A[i]]$ or extends the best subarray ending at $i-1$, and extending is worth doing exactly when that best sum is positive. Pseudocode: $d = [0]*n$; $d[1] = A[1]$; for $i = 2..n$: $d[i] = A[i] + \max(d[i-1], 0)$; return $\max(d)$. Running time n subproblems $\times O(1) = O(n)$. For the example $d = [2, -3, 3, 2, 6]$, answer 6.

2.6. Fill the LPS table for $A = \text{abaca}$ and give the answer.

Rows $i = 1..5$ (a, b, a, c, a), upper triangle: row 1: $[1, 1, 3, 3, 3]$; row 2: $[1, 1, 1, 3]$; row 3: $[1, 1, 3]$; row 4: $[1, 1]$; row 5: $[1]$. $\text{OPT}[1][5] = 3$ (aba or aca).

2.7. In the LPS recurrence, why is it safe to use both $A[i]$ and $A[j]$ whenever they are equal, rather than also comparing with the two "skip" options?

Take any palindromic subsequence of $A[i:j]$; if it uses at most one of the two endpoints, drop that endpoint (if used) and wrap the result with $A[i]$ and $A[j]$, which are equal; the result is a palindrome of length at least as large. So some optimal solution uses both endpoints, and $2 + \text{OPT}[i+1][j-1]$ is at least as large as the two skip options.

2.8. (Lecture exercise.) Give an $O(n^2)$ algorithm for the length of the longest palindromic substring (contiguous).

Subproblems: for $i \leq j$, $\text{OPT}[i][j] = \text{true}$ if $A[i:j]$ is a palindrome, else false. Base: $\text{OPT}[i][i] = \text{true}$; $\text{OPT}[i][i+1] = (A[i] = A[i+1])$. Recurrence for $j \geq i + 2$: $\text{OPT}[i][j] = (A[i] = A[j])$ and $\text{OPT}[i+1][j-1]$. Fill bottom row up, left to right, and keep the largest $j - i + 1$ with $\text{OPT}[i][j]$ true. $O(n^2)$ subproblems, $O(1)$ each.

2.9. Fill the 0/1 knapsack table for $v = [3, 4, 5]$, $w = [2, 3, 4]$, $B = 5$ and give the answer.

Row 1 (3, 2): [0, 0, 3, 3, 3, 3]. Row 2 (4, 3): [0, 0, 3, 4, 4, 7]. Row 3 (5, 4): [0, 0, 3, 4, 5, 7]. Answer $\text{OPT}[3][5] = 7$ (items 1 and 2, weight 5).

2.10. For the same instance, what does the unbounded knapsack return? Give the base row and the final row.

Base row $\text{OPT}[1][j] = \lfloor j/2 \rfloor \cdot 3$: [0, 0, 3, 3, 6, 6]. Row 2: $\text{OPT}[2][j] = \max(\text{OPT}[1][j], 4 + \text{OPT}[2][j-3])$: [0, 0, 3, 4, 6, 7]. Row 3: $\max(\text{OPT}[2][j], 5 + \text{OPT}[3][j-4])$: [0, 0, 3, 4, 6, 7]. Answer 7 (one of item 1 and one of item 2; two copies of item 1 give only 6).

2.11. Why must the one-array version of 0/1 knapsack process j from B down to 0, and what does the left-to-right order compute instead?

$d[j]$ must be computed from the *previous row's* $d[j - w[i]]$. Going right to left, $d[j - w[i]]$ has not been overwritten yet. Going left to right it has, so item i may be counted again, which is the unbounded knapsack recurrence.

2.12. In the Alice-and-Bob game with $A = [1, 2, 3, 4, 5]$, compute S and OPT and give Alice's score.

$S = [15, 14, 12, 9, 5]$, $S[6] = 0$. $\text{OPT}[5] = 5$, $\text{OPT}[4] = 9$, $\text{OPT}[3] = 12$. $\text{OPT}[2] = 14 - \min(12, 9, 5) = 9$. $\text{OPT}[1] = 15 - \min(9, 12, 9) = 6$. Alice's best score is 6: for example she takes 1, 2, 3 and Bob takes 4 and 5; or she takes only 1, after which the best Bob can do from $[2, 3, 4, 5]$ is $\text{OPT}[2] = 9$, leaving Alice $1 + (14 - 9) = 6$.

2.13. (Coin change.) Coins have positive integer denominations $c[1..k]$, unlimited supply; return the minimum number of coins whose values add up to exactly B , or report impossible. Give an $O(kB)$ algorithm.

Subproblems indexed by amount: for j from 0 to B , $OPT[j]$ = minimum number of coins summing to j (∞ if impossible); return $OPT[B]$. Base $OPT[0] = 0$. Recurrence $OPT[j] = 1 + \min$ over coins $c[i] \leq j$ of $OPT[j - c[i]]$ (the last coin used has some denomination $c[i]$, and the rest sum to $j - c[i]$ optimally). Fill $j = 1..B$; each takes $O(k)$; total $O(kB)$.

2.14. Name the pattern: (a) " $OPT[i][j]$ = best value for $A[1:i]$ with j moves left"; (b) " $OPT[i]$ = best value for $A[i:n]$ "; (c) " $OPT[i]$ = longest valid block ending at i ".

(a) Pattern 5 (a second budget index). (b) Pattern 2 (suffix). (c) Pattern 3 (the LIS trick), so the answer is a max over i .

CH 4 · DAYS 3–4 3. Dynamic programming on two strings: edit distance

The previous chapter used one-dimensional tables: one entry per prefix of a single array. This chapter introduces the two-dimensional table, which is needed when the input consists of *two* sequences and a subproblem must say how much of each has been used. The example is edit distance, which the lecture treated in detail on two separate days and which is the most important single algorithm of Chapter 4 for exam purposes: it is the model for every "two strings" or "two arrays" DP.

3.1 The problem

The story, from lecture, is spell-checking and genetics: how far apart are two words, or two DNA sequences? The input is two strings, A of length m and B of length n. You may change A using three kinds of *moves*:

1. **Insert** any character anywhere in A.
2. **Delete** any character from A.
3. **Replace** any character of A with any character.

The **edit distance** $ED(A, B)$ is the minimum number of moves needed to turn A into B. The goal is to return $ED(A, B)$. The lecture's examples: $ED(\text{cat}, \text{bat}) = 1$ (replace c with b), $ED(\text{cat}, \text{cats}) = 1$ (insert s), $ED(\text{cat}, \text{bar}) = 2$ (replace c, replace t), $ED(\text{cat}, \text{dog}) = 3$ (replace every letter). The running-time target is $O(mn)$.

WHAT "SUBSEQUENCE" AND "PREFIX" MEAN HERE. $A[1:i]$ is the prefix of A consisting of its first i characters, and $A[1:0]$ is the empty string, written "" in the table below. Index 0 is allowed in this chapter precisely so that the empty prefix is a legal subproblem.

3.2 Subproblems: Pattern 6

A subproblem must record how much of A and how much of B have been accounted for, so it needs two indices.

STEP 1. For every i from 0 to m and every j from 0 to n, let $OPT[i][j] = ED(A[1:i], B[1:j])$, the edit distance between the first i characters of A and the first j characters of B. We will return $OPT[m][n]$.

This is Pattern 6 in the lecture's catalog: the input is a pair (A, B) and $OPT[i][j]$ is the answer for the pair of prefixes $(A[1:i], B[1:j])$. There are $(m + 1)(n + 1)$ subproblems, arranged in a table with $m + 1$ rows (one per prefix of A, including the empty one) and $n + 1$ columns.

Base cases. $OPT[0][j] = j$, because turning the empty string into $B[1:j]$ requires inserting all j characters and nothing less works. Symmetrically $OPT[i][0] = i$: the only way to turn $A[1:i]$ into the empty string is to delete all i characters.

3.3 The recurrence: three ways to handle the last character

The lecture's running example is $A = \text{star}$, $B = \text{water}$. Fix a subproblem with $i, j \geq 1$, say turning "star" into "wate" ($i = 4, j = 4$). Think about what the *last* thing is that any sequence of moves does to produce the final character "e" of the target. There are exactly three possibilities:

1. Turn star into wat (cost $\text{OPT}[4][3]$) and then **insert** e at the end. Total $\text{OPT}[i][j-1] + 1$.
2. Turn sta into wate (cost $\text{OPT}[3][4]$) and then **delete** the leftover r. Total $\text{OPT}[i-1][j] + 1$.
3. Turn sta into wat (cost $\text{OPT}[3][3]$) and then deal with the last characters r and e directly: **replace** r with e, costing 1. Total $\text{OPT}[i-1][j-1] + 1$. If the last characters had been equal, as with $\text{star} \rightarrow \text{water}$ where both end in r, this step would cost nothing: $\text{star} \rightarrow \text{water}$ is the same as $\text{sta} \rightarrow \text{wate}$.

STEP 2, THE EDIT-DISTANCE RECURRENCE. For $i, j \geq 1$,

$$\text{OPT}[i][j] = \min(\text{OPT}[i][j-1] + 1, \text{OPT}[i-1][j] + 1, \text{OPT}[i-1][j-1] + \delta_{ij}),$$

where $\delta_{ij} = 0$ if $A[i] = B[j]$ and $\delta_{ij} = 1$ otherwise. Base cases $\text{OPT}[0][j] = j$ and $\text{OPT}[i][0] = i$.

In the table, the three terms are the cell to the left, the cell above, and the cell diagonally up-left. So each cell is the minimum of its three neighbors plus 1, except that the diagonal neighbor contributes +0 when the row's character equals the column's character. The lecture also noted a shortcut: **if $A[i] = B[j]$, then $\text{OPT}[i][j] = \text{OPT}[i-1][j-1]$** ; when the last characters already agree, you never need to touch them, so you may copy the diagonal entry without taking a min.

Why the recurrence is correct. Take any optimal way of turning $A[1:i]$ into $B[1:j]$ and look at what happens to the last character of the target $B[j]$. Either it was inserted (case 1), or it came from $A[i]$ (unchanged or replaced, case 3), or $A[i]$ was deleted and $B[j]$ came from an earlier character (case 2). These cases cover every possibility, each case costs exactly what the corresponding term says plus an optimal solution of the smaller subproblem, and the minimum of the three is therefore the optimum. The base cases are correct because inserting or deleting every character is forced when one string is empty.

3.4 Filling the table

Rows are prefixes of $A = \text{star}$, columns are prefixes of $B = \text{water}$. The first row and first column are the base cases 0, 1, 2, ...; every other cell is the minimum of (left + 1, up + 1, diagonal + δ). Fill row by row, left to right, so that all three neighbors are ready.

	"	w	a	t	e	r
"	0	1	2	3	4	5
s	1	1	2	3	4	5
t	2	2	2	2	3	4
a	3	3	2	3	3	4
r	4	4	3	3	4	3

Some cells, worked out loud. Cell (s, w): left is 1, up is 1, diagonal is 0 and $s \neq w$ so $\delta = 1$; the minimum of 2, 2, 1 is 1 (replace s with w). Cell (t, t), shown in bold: the characters match, so copy the diagonal cell (s, a) = 2; indeed "st" $\rightarrow$ "wat" costs 2 (replace s with w, insert a). Cell (a, a): match, copy the diagonal (t, w) = 2. Cell (r, a): left 4 + 1, up 2 + 1, diagonal 3 + 1 ($r \neq a$); the minimum is 3. Cell (r, r), the answer: match, copy the diagonal (a, e) = 3. So $ED(\text{star}, \text{water}) = 3$; one optimal sequence of moves is $\text{star} \rightarrow \text{wtar} \rightarrow \text{watar} \rightarrow \text{water}$ (replace s with w, insert a, replace the second a with e).

HOW THIS IS TESTED. A Problem 1 question can give two short strings and ask for $ED(A, B)$, or ask for one specific row of the table. Draw the whole table; with the base-case row and column written first it takes a minute, and the three-neighbors rule leaves nothing to think about. Write the strings along the edges with the empty prefix first, and check your answer by exhibiting an actual sequence of moves of that length.

3.5 Pseudocode and running time

```

Edit-Distance(A, B):           # |A| = m, |B| = n
    d = (m+1) x (n+1) table
    for i = 0, ..., m: d[i][0] = i
    for j = 0, ..., n: d[0][j] = j
    for i = 1, ..., m:
        for j = 1, ..., n:
            delta = 0 if A[i] = B[j] else 1
            d[i][j] = min(d[i][j-1] + 1, d[i-1][j] + 1, d[i-1][j-1] + delta)
    return d[m][n]

```

Running time. There are $(m + 1)(n + 1) = O(mn)$ subproblems and each takes $O(1)$ time (a comparison and a minimum of three numbers), so the total is $O(mn)$. The lecture's slogan applies exactly: number of subproblems times time per subproblem.

Order of evaluation. The double loop fills the table row by row. Any order works as long as the left, upper, and diagonal neighbors of a cell are filled before the cell itself; column by column is also fine. This is the two-dimensional version of "fill d from left to right."

3.6 Recovering the moves, not just the number

The table stores only costs. To recover an actual sequence of moves, start at the bottom-right cell and walk backward: at cell (i, j), find which of the three neighbors achieved the minimum, record the corresponding move (insert $B[j]$, delete $A[i]$, or replace/keep), and step to that neighbor; stop at (0, 0). Each step moves at least one index down, so this takes $O(m + n)$ time after the table is built. The lecture called such stored choices *parent pointers*; the space-saving version of a DP (Section 1.6) throws away exactly the information this walk needs, which is why saving space makes reconstruction hard.

3.7 Variants built on the same table

Exam problems like to change the rules slightly and ask for the new recurrence. The table and the running time do not change; only the three terms do.

- **Different move costs.** If an insertion costs c_{ins} , a deletion c_{del} , and a replacement c_{rep} , the recurrence becomes $\min(\text{OPT}[i][j-1] + c_{\text{ins}}, \text{OPT}[i-1][j] + c_{\text{del}}, \text{OPT}[i-1][j-1] + c_{\text{rep}} \cdot \delta_{ij})$, with base cases $\text{OPT}[0][j] = j \cdot c_{\text{ins}}$ and $\text{OPT}[i][0] = i \cdot c_{\text{del}}$.
- **No replacements allowed.** Drop the diagonal term unless $A[i] = B[j]$: $\text{OPT}[i][j] = \min(\text{OPT}[i][j-1] + 1, \text{OPT}[i-1][j] + 1)$ when the characters differ, and $\text{OPT}[i-1][j-1]$ when they agree.
- **Longest common subsequence (LCS).** Extra context beyond the notes, included because it is the standard companion to edit distance. A subsequence is obtained by deleting characters (not necessarily adjacent ones) without reordering. To find the length of the longest string that is a subsequence of both A and B, use the same subproblems $\text{OPT}[i][j] = \text{LCS length of } A[1:i] \text{ and } B[1:j]$, with base cases $\text{OPT}[0][j] = \text{OPT}[i][0] = 0$, and recurrence $\text{OPT}[i][j] = \text{OPT}[i-1][j-1] + 1$ if $A[i] = B[j]$, and $\max(\text{OPT}[i-1][j], \text{OPT}[i][j-1])$ otherwise. Running time $O(mn)$.

Exercises

3.1. Define the subproblems for edit distance, state the base cases, and say which entry is returned.

For $0 \leq i \leq m$ and $0 \leq j \leq n$, $\text{OPT}[i][j] = \text{ED}(A[1:i], B[1:j])$. Base cases $\text{OPT}[0][j] = j$ and $\text{OPT}[i][0] = i$. Return $\text{OPT}[m][n]$.

3.2. Fill the edit-distance table for $A = \text{sun}$ and $B = \text{snow}$ and report $\text{ED}(\text{sun}, \text{snow})$.

	" "	s	n	o	w
" "	0	1	2	3	4
s	1	0	1	2	3
u	2	1	1	2	3
n	3	2	1	2	3

$\text{ED}(\text{sun}, \text{snow}) = 3$, for example $\text{sun} \rightarrow \text{snn} \rightarrow \text{sno} \rightarrow \text{snow}$ (replace u, replace n, insert w).

3.3. Fill the table for $A = \text{ab}$ and $B = \text{ba}$. What is $\text{ED}(\text{ab}, \text{ba})$, and why is it not 1?

Rows " ", a, b; columns " ", b, a: $[0,1,2], [1,1,1], [2,1,2]$. $\text{ED} = 2$. A single move cannot work: one insertion or deletion changes the length, and one replacement changes one character, but ab and ba differ in both positions.

3.4. In the star/water table, explain in one sentence each why cell (t, w) equals 2 and why cell (a, t) equals 3.

(t, w): "st" $\rightarrow$ "w" needs two moves (delete one letter, replace the other); the recurrence gives $\min(2+1, 1+1, 1+1) = 2$. (a, t): "sta" $\rightarrow$ "wat": left (a, a) = 2 plus 1, up (t, t) = 2 plus 1, diagonal (t, a) = 2 plus $\delta = 1$ since $a \neq t$; the minimum is 3.

3.5. Write the recurrence for edit distance when a replacement costs 2 but insertions and deletions cost 1. Does the shortcut "if $A[i] = B[j]$ then copy the diagonal" still hold?

$\text{OPT}[i][j] = \min(\text{OPT}[i][j-1] + 1, \text{OPT}[i-1][j] + 1, \text{OPT}[i-1][j-1] + 2 \cdot \delta_{ij})$, base cases unchanged.
Yes: when $A[i] = B[j]$ the diagonal term costs $\text{OPT}[i-1][j-1] + 0$, which is never worse than the other two options, because any way of handling the two prefixes can be extended by leaving the matching characters alone. (With replacement cost 2, a replacement is never better than a deletion followed by an insertion, so the table equals the one for insert/delete only.)

3.6. Describe an $O(mn)$ -time algorithm that returns the length of the longest common subsequence of A and B. Give the four steps.

Subproblems: $\text{OPT}[i][j]$ = LCS length of $A[1:i]$ and $B[1:j]$ for $0 \leq i \leq m, 0 \leq j \leq n$; return $\text{OPT}[m][n]$.
Base: $\text{OPT}[0][j] = \text{OPT}[i][0] = 0$. Recurrence: if $A[i] = B[j]$, $\text{OPT}[i][j] = \text{OPT}[i-1][j-1] + 1$ (an optimal common subsequence can be assumed to use the matching last characters); else $\text{OPT}[i][j] = \max(\text{OPT}[i-1][j], \text{OPT}[i][j-1])$ (a common subsequence cannot use both $A[i]$ and $B[j]$, so one of them is unused). Pseudocode: fill the table row by row. Running time: $O(mn)$ subproblems, $O(1)$ each.

3.7. After computing the edit-distance table, how do you output an actual optimal sequence of moves, and how long does that take?

Walk backward from (m, n): at each cell move to whichever neighbor (left, up, diagonal) attains the minimum, recording insert, delete, or replace/keep respectively; stop at (0, 0). At most $m + n$ steps, so $O(m + n)$ extra time.

CH 4 · DAYS 4–5 4. Dynamic programming on trees

The last two lectures of Chapter 4 moved from arrays to trees. The method is unchanged, but the subproblems are now subtrees instead of prefixes, and the order in which the table is filled is "children before parents" instead of "left to right." Two problems were treated: the maximum-weight independent set, which is the model problem, and the maximum-weight matching, which was set as a worksheet.

4.1 Trees as input

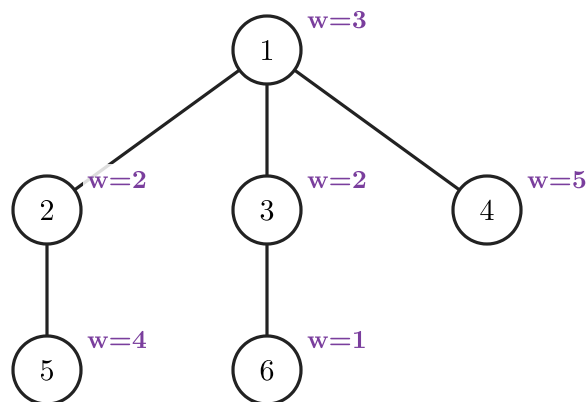
In this chapter a **tree** is a connected graph with no cycles, **rooted** at vertex 1. Rooting the tree gives every edge a direction, away from the root, and gives every vertex other than the root a unique **parent**; the vertices just below u are its **children**. A vertex with no children is a **leaf**. The tree is given as an array T where $T[u]$ is the list of u 's children (an adjacency list that only records edges going downward). The **subtree rooted at u** , written $T(u)$, consists of u together with all of its descendants. A tree with n vertices has exactly $n - 1$ edges, so anything that costs $O(m + n)$ on a tree costs $O(n)$.

Trees are friendly to dynamic programming because of one fact: *the subtrees of the different children of u do not touch each other*. There is no edge between a vertex in $T(v_1)$ and a vertex in $T(v_2)$ when v_1 and v_2 are distinct children of u . So a decision made inside one child's subtree never affects another child's subtree, and the subproblems "answer for $T(v)$ " can be combined independently at u .

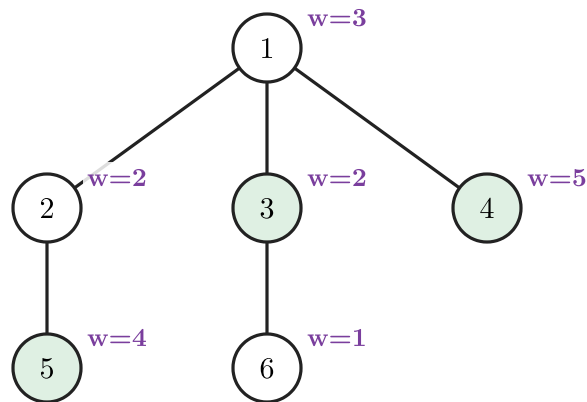
THE TREE PATTERN. Subproblems are indexed by vertices: $\text{OPT}[u]$ (or a small fixed number of values per vertex, such as $\text{OPT}_{\text{in}}[u]$ and $\text{OPT}_{\text{out}}[u]$) is the answer to the question restricted to the subtree $T(u)$. The recurrence combines the values of u 's children. The table is filled with every child computed before its parent, and the answer is read off at the root, vertex 1.

4.2 Maximum-weight independent set

An **independent set** in a graph is a set S of vertices such that no edge has both of its endpoints in S : no two chosen vertices are adjacent. **Problem (Day 4).** The input is a tree T rooted at 1 in which every vertex u has a weight $w[u] > 0$. Return the maximum possible total weight $\sum_{u \in S} w[u]$ of an independent set S . (The lecture abbreviates "maximum independent set" as MIS.)



In the lecture's first example above, the weights are 3 at the root, 2, 2, 5 at its children, 4 below the first child and 1 below the second. Taking the root (3) forbids all three of its children, leaving $3 + 4 + 1 = 8$. Better is to skip the root: take the vertex of weight 5, the vertex of weight 4 (which forbids its parent of weight 2), and the other child of weight 2 (which forbids the leaf of weight 1), for $5 + 4 + 2 = 11$, which is optimal.



Why one number per vertex is not enough

The first attempt in lecture was the natural subproblem $\text{OPT}[u] = \text{weight of the maximum independent set in } T(u)$. It was crossed out. The trouble appears when combining at u : if u is put into S , then no child of u may be in S , but the number $\text{OPT}[v]$ for a child v does not tell us whether the set it describes uses v . We need to know the best set in $T(v)$ that avoids v , and the best overall. So the lecture kept *two* numbers per vertex.

STEP 1, SUBPROBLEMS. For every vertex u ,

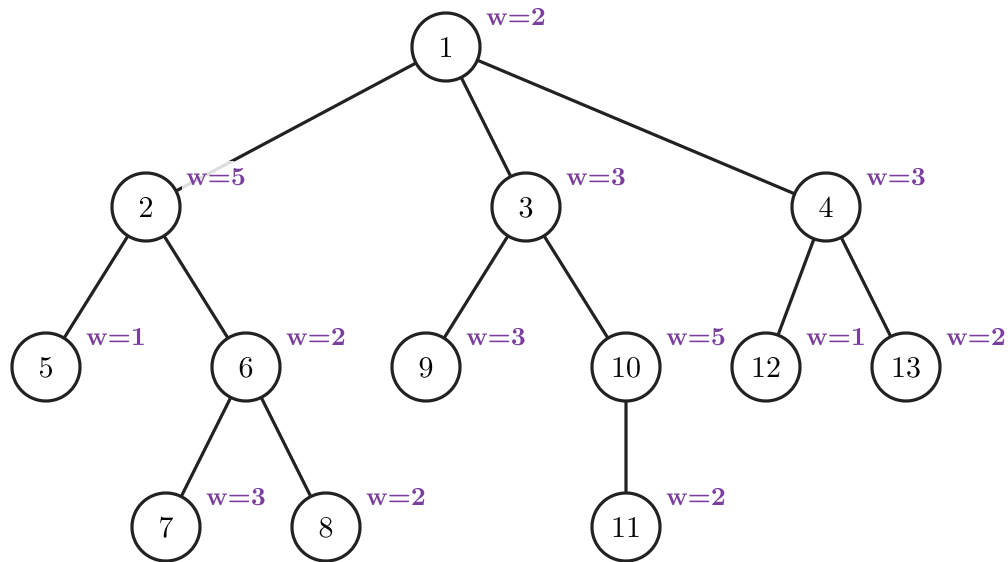
- $\text{OPT}_{\text{in}}[u]$ = the weight of the maximum independent set of $T(u)$ that *includes* u ;
- $\text{OPT}_{\text{out}}[u]$ = the weight of the maximum independent set of $T(u)$ that *excludes* u .

We will return $\max(\text{OPT}_{\text{in}}[1], \text{OPT}_{\text{out}}[1])$.

STEP 2, RECURRENCE. If u is a leaf: $\text{OPT}_{\text{in}}[u] = w[u]$ and $\text{OPT}_{\text{out}}[u] = 0$. Otherwise,
 $\text{OPT}_{\text{in}}[u] = w[u] + \sum_{v \in T[u]} \text{OPT}_{\text{out}}[v]$, $\text{OPT}_{\text{out}}[u] = \sum_{v \in T[u]} \max(\text{OPT}_{\text{in}}[v], \text{OPT}_{\text{out}}[v])$.

Why. If u is in S , none of its children can be, so within each child's subtree the best we can do is the best set that excludes the child, and the subtrees are independent of each other, so the totals add. If u is not in S , each child's subtree may do whatever is best for it, with or without the child, and again the totals add. The leaf cases are immediate. Every independent set of $T(u)$ either contains u or does not, so the two values cover every possibility, and the answer for the whole tree is the better of the two at the root.

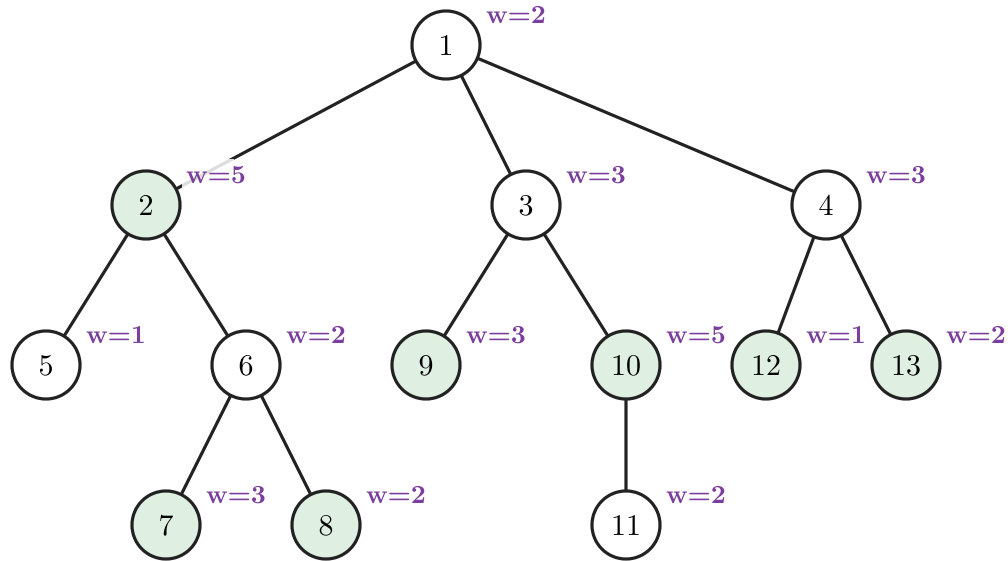
A worked example



This is the lecture's larger example. Each vertex is labelled with its number, and the weight is written beside it. Working upward from the leaves, the pair (OPT_{in}, OPT_{out}) at each vertex is:

Vertex	w	Children	OPT_{in}	OPT_{out}
5, 7, 8, 9, 11, 12, 13 (leaves)	1, 3, 2, 3, 2, 1, 2	none	w	0
6	2	7, 8	$2 + 0 + 0 = 2$	$\max(3,0) + \max(2,0) = 5$
10	5	11	$5 + 0 = 5$	$\max(2,0) = 2$
2	5	5, 6	$5 + 0 + 5 = 10$	$\max(1,0) + \max(2,5) = 6$
3	3	9, 10	$3 + 0 + 2 = 5$	$\max(3,0) + \max(5,2) = 8$
4	3	12, 13	$3 + 0 + 0 = 3$	$\max(1,0) + \max(2,0) = 3$
1 (root)	2	2, 3, 4	$2 + 6 + 8 + 3 = 19$	$\max(10,6) + \max(5,8) + \max(3,3) = 21$

The answer is $\max(19, 21) = 21$. To see which vertices achieve it, walk back down: the root is out; child 2 is in ($10 > 6$), so 5 and 6 are out and then 7 and 8 are in; child 3 is out ($8 > 5$), so 9 and 10 are free and both are in, forcing 11 out; child 4 is a tie, and taking it out gives 12 and 13. The set $\{2, 7, 8, 9, 10, 12, 13\}$ has weight $5 + 3 + 2 + 3 + 5 + 1 + 2 = 21$.



4.3 The order of computation and the running time

Each recurrence uses the values of u 's children, so every child must be computed before its parent. The lecture's phrase for this is "sort V in reverse topological order": if you think of each edge as pointing from parent to child, a topological ordering lists every parent before its children, and the reverse of it lists every child before its parent. In practice any of the following produces such an order in $O(n)$ time: run DFS from the root and take the vertices in order of their post values (a vertex finishes after all of its descendants); or run BFS from the root, record the order in which vertices are discovered, and reverse it (a child is always discovered after its parent).

```
MIS-Tree( $T, w$ ):
     $L$  = the vertices of  $T$  in an order with every child before its parent    # e.g. reverse BFS
    order
     $d\_in = [0] * n$ ;  $d\_out = [0] * n$ 
    for  $u$  in  $L$ :
         $d\_in[u] = w[u]$ 
        for  $v$  in  $T[u]$ :
             $d\_in[u] = d\_in[u] + d\_out[v]$ 
             $d\_out[u] = d\_out[u] + \max(d\_in[v], d\_out[v])$ 
    return  $\max(d\_in[1], d\_out[1])$ 
```

For a leaf the inner loop does nothing, so $d_in[u] = w[u]$ and $d_out[u] = 0$, which are the base cases.

Running time. Computing the order is a BFS or DFS on a graph with $n - 1$ edges, $O(n)$. The loop over children costs $O(1)$ per child, and every vertex except the root is the child of exactly one vertex, so the loops cost $O(n)$ in total. The whole algorithm is $O(n)$, which the lecture summarized as " $\approx$ BFS."

PROCESSING IN THE WRONG ORDER. If the vertices are processed in the order 1, 2, ..., n, or in BFS order, then $d_in[u]$ and $d_out[u]$ are read before they are computed and come out as 0. Always say in your pseudocode what order is used and why it works.

4.4 A single array, using grandchildren

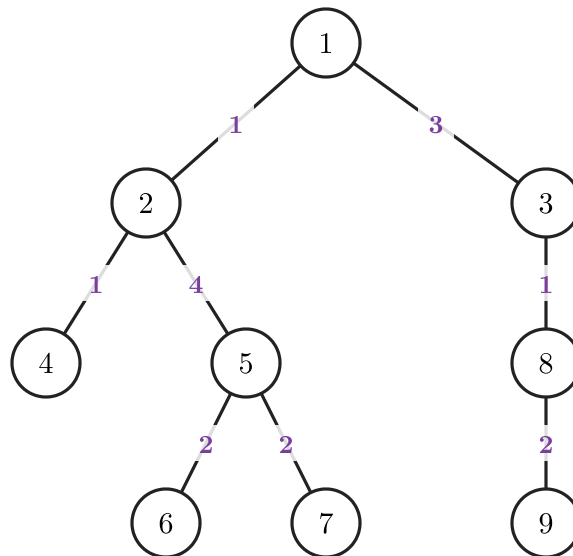
Day 5 returned to the crossed-out subproblem and showed it can be made to work after all, by looking two levels down. Let $OPT[u]$ be the weight of the maximum independent set of $T(u)$. Either u is out, and each child's subtree is free, or u is in, and then each child is out but each *grandchild*'s subtree is free:

SINGLE-ARRAY RECURRENCE. $OPT[u] = \max(\sum_{v \in T[u]} OPT[v], w[u] + \sum_{v \in T[u]} \sum_{x \in T[v]} OPT[x])$.

Running time is still $O(n)$: the triple loop "for u ; for v in $T[u]$; for x in $T[v]$ " visits every vertex once as a child and once as a grandchild, so about $2n$ steps in total. Both formulations are acceptable; the two-array version is easier to explain and extends more easily to other problems, which is why the exam solutions in this book use it.

4.5 Maximum-weight matching in a tree

A **matching** is a set F of edges such that no two edges in F share an endpoint. **Problem (Day 5 worksheet).** The input is a tree rooted at 1 in which every edge e has a weight $w(e) > 0$; return the maximum total weight of a matching.



In the figure, edges are labelled with their weights. The best matching takes the edges of weight 3 (root to its right child), 4 (vertex 2 to vertex 5), and 2 (vertex 8 to vertex 9), for a total of 9; no two of these share a vertex.

SUBPROBLEMS. For every vertex u , $\text{OPT}[u]$ = the weight of a maximum matching inside $T(u)$, and $\text{OPT}_{\text{out}}[u]$ = the weight of a maximum matching inside $T(u)$ in which u is *not* matched to any of its children. Both are 0 when u is a leaf. We will return $\text{OPT}[1]$.

RECURRENCE. $\text{OPT}_{\text{out}}[u] = \sum_{v \in T[u]} \text{OPT}[v]$, and
 $\text{OPT}[u] = \max(\text{OPT}_{\text{out}}[u], \max_{v \in T[u]} [w(u, v) + \text{OPT}_{\text{out}}[v] + \sum_{x \in T[u], x \neq v} \text{OPT}[x]])$.

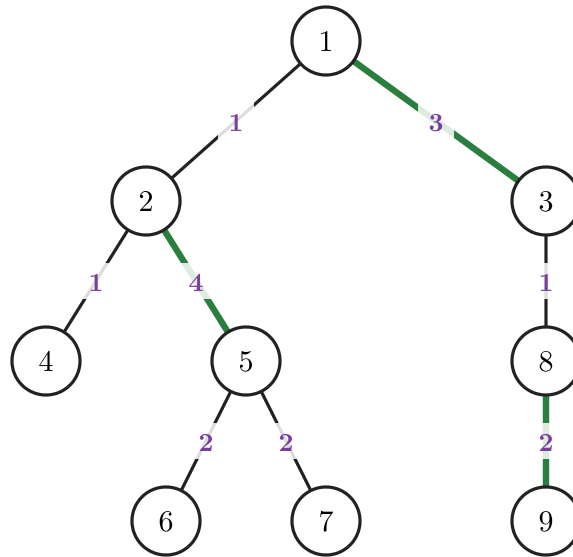
Why. In a matching of $T(u)$, vertex u is matched to at most one child. If it is matched to none, every child's subtree is free ($\text{OPT}_{\text{out}}[u]$). If it is matched to child v by the edge (u, v) , that edge contributes $w(u, v)$; v is now used, so inside $T(v)$ only matchings that leave v unmatched are allowed ($\text{OPT}_{\text{out}}[v]$); and every other child x of u has a free subtree ($\text{OPT}[x]$). Taking the best over the choice of v , and comparing with matching nobody, covers every case.

Running time, and the trick that makes it $O(n)$. Evaluated literally, the sum $\sum_{x \neq v} \text{OPT}[x]$ costs $O(\text{number of children})$ for each candidate v , so a vertex with k children costs $O(k^2)$, which is $O(n^2)$ for a star-shaped tree. The lecture's fix is to notice that $\sum_{x \neq v} \text{OPT}[x] = \text{OPT}_{\text{out}}[u] - \text{OPT}[v]$, which is $O(1)$ once $\text{OPT}_{\text{out}}[u]$ is known. So compute $\text{OPT}_{\text{out}}[u]$ first, then $\text{OPT}[u] = \max(\text{OPT}_{\text{out}}[u], \max_v [w(u, v) + \text{OPT}_{\text{out}}[v] + \text{OPT}_{\text{out}}[u] - \text{OPT}[v]])$, which costs $O(k)$ at a vertex with k children and $O(n)$ in total.

Worked example

Vertex	Children (edge weight)	OPT_{out}	OPT
4, 6, 7, 9	none	0	0
5	6 (2), 7 (2)	$0 + 0 = 0$	$\max(0, 2 + 0 + 0 - 0, 2 + 0 + 0 - 0) = 2$
8	9 (2)	0	$\max(0, 2 + 0 + 0 - 0) = 2$
2	4 (1), 5 (4)	$0 + 2 = 2$	$\max(2, 1 + 0 + 2 - 0, 4 + 0 + 2 - 2) = 4$
3	8 (1)	2	$\max(2, 1 + 0 + 2 - 2) = 2$
1	2 (1), 3 (3)	$4 + 2 = 6$	$\max(6, 1 + 2 + 6 - 4, 3 + 2 + 6 - 2) = 9$

$\text{OPT}[1] = 9$. At the root the winning option matches 1 to 3 (weight 3), leaves 3 unmatched below ($\text{OPT}_{\text{out}}[3] = 2$, the edge 8-9), and takes the best matching of $T(2)$, which is the edge 2-5 of weight 4.



4.6 A remark on general graphs

The lecture ended with a tangent: for an independent set in a *general* graph, not a tree, no polynomial-time algorithm is known, and the question of whether one exists is the P versus NP problem. The tree structure is what makes the dynamic program possible, because it is what makes the children's subproblems independent. On the exam, if a problem gives you a tree, expect a DP of this shape; if it gives you an arbitrary graph and asks for an independent set, something else is going on.

Exercises

4.1. State the recurrences for OPT_{in} and OPT_{out} in the maximum-weight independent set problem, including the base cases, and say what is returned.

Leaf: $\text{OPT}_{\text{in}}[u] = w[u]$, $\text{OPT}_{\text{out}}[u] = 0$. Otherwise $\text{OPT}_{\text{in}}[u] = w[u] + \sum_{\text{children } v} \text{OPT}_{\text{out}}[v]$ and $\text{OPT}_{\text{out}}[u] = \sum_{\text{children } v} \max(\text{OPT}_{\text{in}}[v], \text{OPT}_{\text{out}}[v])$. Return $\max(\text{OPT}_{\text{in}}[1], \text{OPT}_{\text{out}}[1])$.

4.2. A tree has root 1 with children 2 and 3; vertex 2 has children 4 and 5; vertex 3 has child 6. The weights are $w = [4, 6, 2, 3, 5, 7]$ for vertices 1 through 6. Compute $(\text{OPT}_{\text{in}}, \text{OPT}_{\text{out}})$ for every vertex and the answer.

Leaves 4, 5, 6: $(3, 0)$, $(5, 0)$, $(7, 0)$. Vertex 2: $\text{in} = 6 + 0 + 0 = 6$, $\text{out} = 3 + 5 = 8$. Vertex 3: $\text{in} = 2 + 0 = 2$, $\text{out} = 7$. Root: $\text{in} = 4 + 8 + 7 = 19$, $\text{out} = \max(6, 8) + \max(2, 7) = 15$. Answer 19, the set $\{1, 4, 5, 6\}$.

4.3. Give a specific small tree on which "OPT[u] = weight of the maximum independent set of T(u), computed from the children's OPT values only" cannot be made to work, and explain what information is missing.

Root r of weight 5 with one child v of weight 4, which has one child x of weight 4. $\text{OPT}[v] = 4$ is achieved both by $\{v\}$ and by $\{x\}$. To decide whether r can be added we must know whether the set below uses v : with $\{x\}$ we get $\{r, x\}$ of weight 9, with $\{v\}$ we cannot add r . The single number 4 does not say which; the pair $\text{OPT}_{\text{in}}[v] = 4$, $\text{OPT}_{\text{out}}[v] = 4$ does, and gives $\text{OPT}_{\text{in}}[r] = 5 + \text{OPT}_{\text{out}}[v] = 9$.

4.4. Describe two different $O(n)$ ways to produce an order of the vertices in which every child comes before its parent.

(1) Run DFS from the root and list vertices in increasing order of post value. (2) Run BFS from the root, record the order of discovery, and reverse it. Either is $O(n)$ since a tree has $n - 1$ edges.

4.5. Why is the running time of MIS-Tree $O(n)$ and not $O(n^2)$, even though there is a loop over children inside a loop over vertices?

The inner loop at u runs once per child of u . Each vertex other than the root is the child of exactly one vertex, so the inner loops run $n - 1$ times in total, not n per vertex.

4.6. Compute OPT_{out} and OPT for the maximum-weight matching in the tree with root 1, children 2 (edge weight 5) and 3 (edge weight 4); vertex 2 has children 4 (1) and 5 (2); vertex 3 has children 6 (3) and 7 (3).

Leaves 4, 5, 6, 7: $(0, 0)$. Vertex 2: $\text{out} = 0$; $\text{OPT} = \max(0, 1, 2) = 2$. Vertex 3: $\text{out} = 0$; $\text{OPT} = \max(0, 3, 3) = 3$. Root: $\text{out} = 2 + 3 = 5$; $\text{OPT} = \max(5, 5 + 0 + 5 - 2 = 8, 4 + 0 + 5 - 3 = 6) = 8$. Answer 8: edges 1-2 and 3-6 (or 3-7).

4.7. In the matching recurrence, explain the identity $\sum_{x \neq v} \text{OPT}[x] = \text{OPT}_{\text{out}}[u] - \text{OPT}[v]$ and why it matters for the running time.

$\text{OPT}_{\text{out}}[u]$ is the sum of OPT over all children; subtracting the term for v leaves the sum over the others. Without it each candidate v costs $O(k)$ at a vertex with k children, $O(k^2)$ per vertex and $O(n^2)$ overall; with it each candidate costs $O(1)$ and the total is $O(n)$.

4.8. (Vertex cover.) A vertex cover is a set C of vertices such that every edge has at least one endpoint in C . Describe an $O(n)$ algorithm for the minimum-weight vertex cover of a weighted tree.

Subproblems: $OPT_{in}[u] = \min$ weight of a vertex cover of $T(u)$ containing u ; $OPT_{out}[u] = \min$ weight of one not containing u . Leaf: $(w[u], 0)$. Otherwise $OPT_{in}[u] = w[u] + \sum_v \min(OPT_{in}[v], OPT_{out}[v])$ (edge (u, v) is covered by u , so v is free), and $OPT_{out}[u] = \sum_v OPT_{in}[v]$ (each edge (u, v) must be covered by v). Return $\min(OPT_{in}[1], OPT_{out}[1])$. Children before parents; $O(n)$.

4.9. (Warm-up.) Describe an $O(n)$ algorithm that returns an array s with $s[u] =$ the total weight of the subtree $T(u)$, for every u .

$OPT[u] = w[u] + \sum_{v \in T[u]} OPT[v]$; leaves have $OPT[u] = w[u]$. Process children before parents; each vertex is summed into its parent once, so $O(n)$. Return the whole array.

CH 5 · DAYS 1–2 5. Shortest paths

Chapter 5 of the course is about finding shortest paths in a directed graph whose edges have lengths. The lecture introduced it as "four sections, three of which are dynamic programming," and the first two algorithms in this chapter are indeed dynamic programs in exactly the sense of Chapter 1, with the vertices as subproblems. The third, Dijkstra's algorithm, is a greedy algorithm in the style of Prim's. Which algorithm applies depends on what the edge lengths are allowed to be, so keep track of the assumptions as you read.

5.1 The single-source shortest path problem

Edge length.

Every edge $e = (u, v)$ has a number $\ell(e) = \ell(u, v)$, its length (also called its weight). Unless a section says otherwise, lengths may be positive, zero, or negative.

Length of a path.

The sum of the lengths of its edges. A path with no edges has length 0.

Distance.

The distance from s to v is the smallest length of any path from s to v ; it is ∞ if v cannot be reached from s .

Walk.

Like a path, but vertices and edges may repeat. Bellman-Ford reasons about walks.

SINGLE-SOURCE SHORTEST PATHS (SSSP). Input: a directed graph G with edge lengths, and a start vertex s . Goal: return an array d such that for every vertex v , $d[v]$ is the distance from s to v .

From Midterm 1 you know one algorithm for this problem already: breadth-first search computes $d[v]$ = the number of edges on a shortest path, which is the distance when every edge has the same positive length. The lecture lists it as section 2.1 and notes that it "works when all $\ell(e)$ are the same and > 0 ." The three new algorithms handle more general lengths.

Algorithm	Requirement on the graph	Running time	Kind
BFS	all lengths equal and positive	$O(m + n)$	search
DAG DP (5.2)	no cycles; any lengths, negative allowed	$O(m + n)$	dynamic programming
Bellman-Ford (5.3)	any graph with no negative cycle	$O(mn)$	dynamic programming
Dijkstra (5.4)	all lengths ≥ 0	$O(n^2)$, or $O(m \log n)$ with a heap	greedy

5.2 Shortest paths in a DAG by dynamic programming

A DAG is a directed graph with no cycles. Recall from Midterm 1 that a DAG has a **topological ordering**, a listing of the vertices in which every edge goes from an earlier vertex to a later one, and that Topological-Sort finds one in $O(m + n)$ time. The lecture's section 5.1 is the shortest-path problem on a DAG, with lengths that "can be anything," including negative.

STEPS 1 AND 2. For every vertex v , $\text{OPT}[v]$ = the distance from s to v , with $\text{OPT}[s] = 0$ as the base case. For $v \neq s$,

$$\text{OPT}[v] = \min_{u: (u, v) \in E} (\text{OPT}[u] + \ell(u, v)),$$

the minimum over the in-neighbors u of v (the vertices with an edge into v); if v has no in-neighbor, $\text{OPT}[v] = \infty$.

Why. A shortest path from s to v , if $v \neq s$, has a last edge (u, v) for some in-neighbor u . The part of the path before that edge is a path from s to u , whose length is at least the distance $\text{OPT}[u]$; so the shortest path to v has length at least $\min_u (\text{OPT}[u] + \ell(u, v))$. Conversely, for each in-neighbor u , a shortest path to u followed by the edge (u, v) is a path to v of length $\text{OPT}[u] + \ell(u, v)$. So the minimum is exactly the distance.

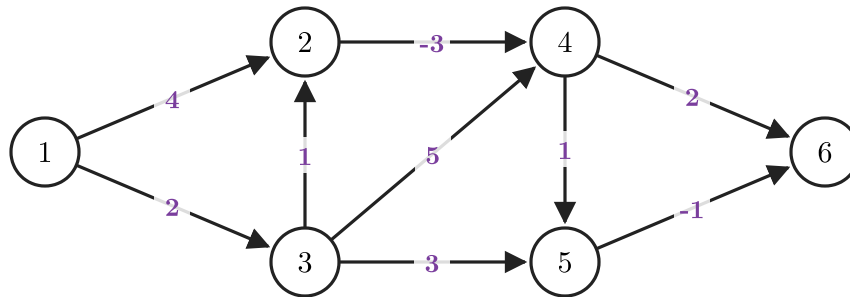
Why a DAG makes it work. The recurrence for $\text{OPT}[v]$ needs OPT of the in-neighbors of v . In a DAG, every in-neighbor of v comes before v in a topological ordering, so if we process vertices in that order, all the values needed for v are ready when v is reached. There is no circularity. (In a graph with a cycle, $\text{OPT}[v]$ might depend on $\text{OPT}[u]$, which depends on $\text{OPT}[v]$; that is the problem Bellman-Ford solves.)

Step 3. To enumerate the in-neighbors of v we need the **reverse graph** G^R , in which every edge is flipped: $G^R[v]$ lists the u with $(u, v) \in E$. It is built in $O(m + n)$ time by scanning every list once: for each u , for each v in $G[u]$, append u to $G^R[v]$.

```
DAG-DP( $G, s$ ):                                # shortest paths in a DAG
     $d = [\text{infinity}] * n$ ;  $d[s] = 0$ 
    topologically sort  $V$ 
     $G\_R = \text{reverse of } G$ 
    for  $v$  in  $V$ , in topological order:
        for  $u$  in  $G\_R[v]$ :
             $d[v] = \min(d[v], d[u] + \ell(u, v))$ 
    return  $d$ 
```

Step 4. Topological sorting is $O(m + n)$, building G^R is $O(m + n)$, and the double loop scans each reversed adjacency list once, $O(m + n)$. Total $O(m + n)$, the same as BFS. Note that vertices placed before s in the topological order, and vertices unreachable from s , keep $d = \infty$, which is correct.

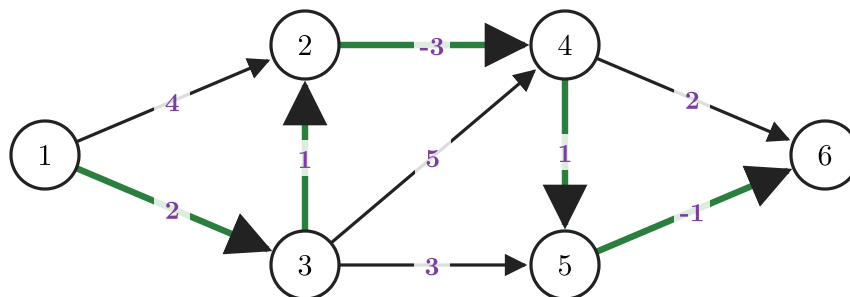
A worked example



Let $s = 1$. A topological ordering is 1, 3, 2, 4, 5, 6 (vertex 2 must come after 3 because of the edge (3, 2)). Processing in that order:

v	in-neighbors	candidates $d[u] + \ell(u, v)$	$d[v]$
1	—	—	0
3	1	$0 + 2$	2
2	1, 3	$0 + 4, 2 + 1$	3
4	2, 3	$3 - 3, 2 + 5$	0
5	3, 4	$2 + 3, 0 + 1$	1
6	4, 5	$0 + 2, 1 - 1$	0

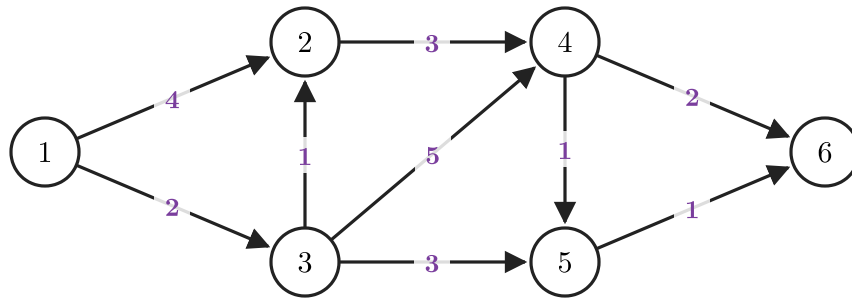
So $d = [0, 3, 2, 0, 1, 0]$. The shortest path to 6 is $1 \rightarrow 3 \rightarrow 2 \rightarrow 4 \rightarrow 5 \rightarrow 6$, of length $2 + 1 - 3 + 1 - 1 = 0$.



Two variations from lecture

Longest paths in a DAG. Does the same logic find the *longest* path from s to each vertex? Yes: $\text{OPT}[v] = \max_u(\text{OPT}[u] + \ell(u, v))$, with d initialized to $-\infty$ and $d[s] = 0$, in the same topological order, in $O(m + n)$ time. The argument is identical with "longest" in place of "shortest." This works only because a DAG has no cycles; see Section 5.6.

Widest paths. Suppose every edge has a positive weight $w(e)$, and define the *width* of a path to be the smallest weight of any edge on it (the "bottleneck"). We want, for every v , the maximum width of any $s \rightarrow v$ path. Subproblems $\text{OPT}[v]$ = that maximum width, with $\text{OPT}[s] = \infty$ (no edge to be a bottleneck); recurrence $\text{OPT}[v] = \max_u \min(\text{OPT}[u], w(u, v))$, because a path to v through its last edge (u, v) has width $\min(\text{width of the part to } u, w(u, v))$, and the part to u should be as wide as possible. Pseudocode: $d = [0]^*n$, $d[s] = \infty$, and in topological order, for u in $G^R[v]$: $d[v] = \max(d[v], \min(d[u], w(u, v)))$. $O(m + n)$.

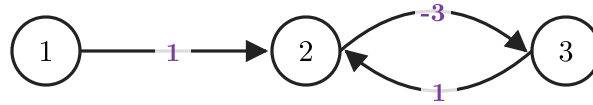


On the DAG above (same shape, all weights positive), the widest-path values from $s = 1$ in the order 1, 3, 2, 4, 5, 6 are $\infty, 2, 4, 3, 2, 2$: for instance $d[2] = \max(\min(\infty, 4), \min(2, 1)) = 4$ and $d[4] = \max(\min(4, 3), \min(2, 5)) = 3$. The longest-path values are 0, 2, 4, 7, 8, 9.

DAG DP AS A DESIGN PATTERN. Any time a problem can be rephrased as "best path in a DAG," the solution is this section: define $\text{OPT}[v]$ for each vertex, write a recurrence over in-neighbors with min or max and + or min as appropriate, process in topological order, $O(m + n)$. Problems on grids ("move only right or down"), on schedules with prerequisites, and on sequences ("jump forward") are DAG problems in disguise.

5.3 Bellman-Ford

Now let G be any directed graph, with cycles allowed and negative lengths allowed, subject to one condition: **G has no negative cycle**, that is, no cycle whose total length is negative. If a negative cycle were reachable from s , walking around it again and again would make walks of arbitrarily negative length, and "distance" would make no sense.



Above, the cycle $2 \rightarrow 3 \rightarrow 2$ has length $-3 + 1 = -2$, so there is no shortest walk from 1 to 2 or 3. Bellman-Ford assumes nothing of the kind exists.

Why the DAG recurrence breaks

The recurrence $\text{OPT}[v] = \min_u (\text{OPT}[u] + \ell(u, v))$ is still a true statement about distances when there is no negative cycle, but it can no longer be *evaluated*, because on a cycle the subproblems depend on each other in a circle (the lecture's example was a cycle $5 \rightarrow 7 \rightarrow 3 \rightarrow 5$): there is no order in which every in-neighbor is done first. The lecture crossed the recurrence out "because of cycles." The repair is the standard one for a circular recurrence: add a second index that always decreases.

STEP 1, SUBPROBLEMS. For every vertex v and every j from 0 to $n - 1$, let $\text{OPT}[v][j]$ be the length of the shortest walk from s to v that uses at most j edges (∞ if there is none). We will return the column $j = n - 1$, that is, $d[v] = \text{OPT}[v][n - 1]$ for every v .

Why $n - 1$ edges suffice. Since there are no negative cycles, removing a cycle from a walk never makes it shorter, so some shortest walk to v is a path with no repeated vertices, which has at most $n - 1$ edges. Hence $\text{OPT}[v][n - 1]$ is the true distance.

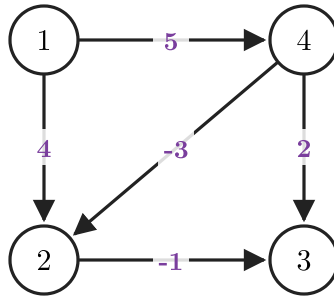
STEP 2, RECURRENCE. Base cases $\text{OPT}[s][0] = 0$ and $\text{OPT}[v][0] = \infty$ for $v \neq s$. For $j \geq 1$,

$$\text{OPT}[v][j] = \min(\text{OPT}[v][j - 1], \min_{u: (u, v) \in E} (\text{OPT}[u][j - 1] + \ell(u, v))).$$

Why. The shortest walk to v with at most j edges either uses at most $j - 1$ edges (first term), or uses exactly j edges, in which case it ends with some edge (u, v) and the walk before it reaches u with at most $j - 1$ edges, so it costs at least $\text{OPT}[u][j - 1] + \ell(u, v)$; conversely each such combination is a walk to v with at most j edges. The two indices make the dependence acyclic: column j depends only on column $j - 1$, so the columns are filled left to right.

The table

Rows are vertices, columns are $j = 0, 1, \dots, n - 1$. Column 0 is 0 at s and ∞ elsewhere; each later column is computed from the previous one; the last column is the answer. The lecture's Day 1 example, with $s = 1$, is this graph:



$v \setminus j$	0	1	2	3
1	0	0	0	0
2	∞	4	2	2
3	∞	∞	3	1
4	∞	5	5	5

Column 1: 2 is reached by the edge (1, 2) of length 4, 4 by (1, 4) of length 5, and 3 is not reachable with one edge. Column 2: $\text{OPT}[2][2] = \min(4, \text{OPT}[4][1] - 3) = 2$ using $1 \rightarrow 4 \rightarrow 2$; $\text{OPT}[3][2] = \min(\infty, \text{OPT}[2][1] - 1, \text{OPT}[4][1] + 2) = 3$. Column 3: $\text{OPT}[3][3] = \min(3, \text{OPT}[2][2] - 1, \text{OPT}[4][2] + 2) = 1$ using $1 \rightarrow 4 \rightarrow 2 \rightarrow 3$. Column 3 is returned: $d = [0, 2, 1, 5]$.

```

Bellman-Ford(G, s):
    d = n x n table, all infinity; d[s][0] = 0          # columns 0 .. n-1
    G_R = reverse of G
    for j = 1, ..., n-1:
        for v in V:
            d[v][j] = d[v][j-1]
            for u in G_R[v]:
                d[v][j] = min(d[v][j], d[u][j-1] + l(u, v))
    return column n-1 of d

```

Step 4, running time. There are $n - 1$ columns, and filling one column scans every reversed adjacency list once, $O(m + n)$. Assuming the graph is connected (in the undirected sense), $m \geq n - 1$, so $O(m + n) = O(m)$ and the total is $O(mn)$. The lecture wrote this as "n columns, each takes $O(m + n) = O(m)$."

The space-saving version and edge relaxation

Each column depends only on the previous one, so by Section 1.6 the table can be replaced by a single array that is updated in place. Day 2 gave this version, which is the form of Bellman-Ford found in most textbooks:

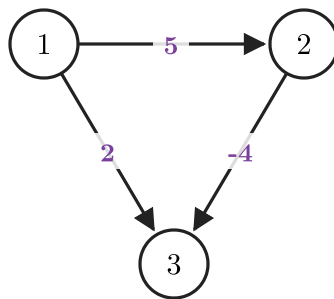
```

Bellman-Ford-2(G, s):                                # space-saving version
    d = [infinity] * n; d[s] = 0
    for j = 1, ..., n-1:
        for each edge (u, v) in E:                    # i.e. for u in V: for v in G[u]
            d[v] = min(d[v], d[u] + l(u, v))          # "relax (u, v)"
    return d

```

The update $d[v] = \min(d[v], d[u] + \ell(u, v))$ is called **relaxing** the edge (u, v) ; the intuition, in the lecture's words, is "check if we can improve $d[v]$ through $d[u]$." The space-saving version needs no reverse graph: it just visits every edge once per round by scanning all the adjacency lists. It is still $O(mn)$: $n - 1$ rounds of $O(m + n)$.

THE TWO VERSIONS AGREE ONLY AT THE END. The lecture posed a true/false question: after round j of Bellman-Ford-2, is d equal to column j of the table? For $j = 0$ and $j = n - 1$, yes. For other j , not necessarily. The reason is that within a single round, an edge (u, v) relaxed after $d[u]$ has already been improved in the same round passes the improvement on immediately, so d can run *ahead* of the table. In the lecture's example with edges $(1, 2)$ of length 5, $(1, 3)$ of length 2, and $(2, 3)$ of length -4 , the table's column 1 is $[0, 5, 2]$, but if the edges are relaxed in the order $(1, 2)$, $(1, 3)$, $(2, 3)$, then after round 1 the array is already $[0, 5, 1]$. Running ahead is harmless: every value of d is always the length of some actual walk, so d never drops below the true distance, and after $n - 1$ rounds it is at most the table's final column, hence equal to the distance. But if an exam question asks for "column j " or "the array after round j ," you must say which version you are running, and for the table version you must compute column j only from column $j - 1$.



Extra context beyond the notes: if a negative cycle is reachable from s , then even after $n - 1$ rounds some edge can still be relaxed. Running one extra round and checking whether any $d[v]$ changes therefore detects reachable negative cycles in $O(m)$ additional time.

5.4 Dijkstra's algorithm

Dijkstra's algorithm assumes **all lengths are nonnegative**, $\ell(e) \geq 0$, and in exchange is faster than Bellman-Ford. The lecture introduced it as " $\approx$ Prim's": like Prim's algorithm for minimum spanning trees, it grows a set S of finished vertices outward from the start, one vertex per step, always choosing

the vertex that is currently closest.

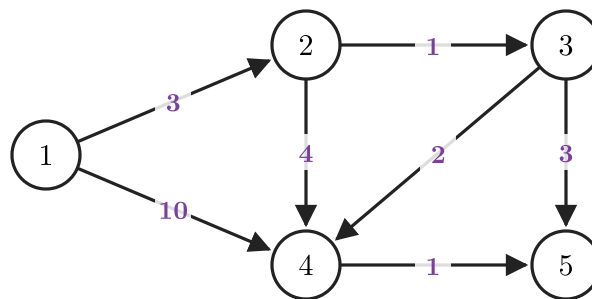
```

Dijkstra(G, s):
  d = [infinity] * n; d[s] = 0
  S = empty set
  while |S| < n:
    u = the vertex in V \ S with the smallest d[u]
    add u to S
    for v in G[u]:
      d[v] = min(d[v], d[u] + l(u, v))    # relax every edge leaving u
  return d

```

So each step finalizes one vertex u (its $d[u]$ never changes again) and relaxes the edges leaving it. Vertices never reached keep $d = \infty$ and are added to S last, which changes nothing.

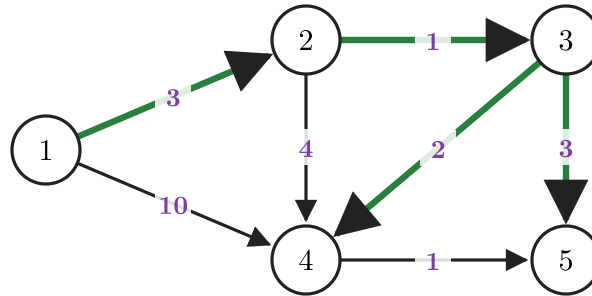
A worked example



This is the lecture's example with $s = 1$. The table shows d after each step; the chosen vertex is in bold.

Step	Chosen u ($d[u]$)	Relaxations	$d = [1, 2, 3, 4, 5]$
0	—	—	$[0, \infty, \infty, \infty, \infty]$
1	1 (0)	$d[2] = 3, d[4] = 10$	$[0, 3, \infty, 10, \infty]$
2	2 (3)	$d[3] = 4, d[4] = \min(10, 7) = 7$	$[0, 3, 4, 7, \infty]$
3	3 (4)	$d[4] = \min(7, 6) = 6, d[5] = 7$	$[0, 3, 4, 6, 7]$
4	4 (6)	$d[5] = \min(7, 7) = 7$	$[0, 3, 4, 6, 7]$
5	5 (7)	—	$[0, 3, 4, 6, 7]$

The vertices are finalized in the order 1, 2, 3, 4, 5, and $d = [0, 3, 4, 6, 7]$. Notice $d[4]$ was improved twice, from 10 to 7 to 6, before vertex 4 was chosen; once chosen, it is final.

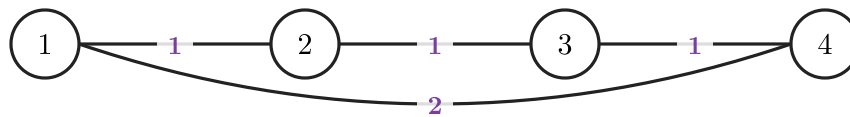


TRACING DIJKSTRA ON THE EXAM. Keep a row of d -values and a list S . At each step pick the smallest d outside S (break ties by the smaller vertex number unless told otherwise), mark it, and relax only the edges leaving it. A Problem 1 question can ask for the final d array, or for the order in which vertices enter S , or for the value of d after a given number of steps. Nonnegative lengths guarantee that the chosen value is final; that is the whole point.

Why it is correct

Extra context beyond the notes, which stated the algorithm without a proof: when u is chosen, $d[u]$ is the length of some $s \rightarrow u$ path that stays inside S until its last edge. Any other $s \rightarrow u$ path must leave S at some first edge (x, y) with x in S and y outside; its length is at least $d[y]$ (because (x, y) was relaxed when x was added) which is at least $d[u]$ (u had the smallest d outside S), and the rest of the path only adds nonnegative lengths. So no path to u is shorter than $d[u]$. This argument fails if an edge can be negative, which is why Dijkstra requires $\ell(e) \geq 0$: with edges $(1, 2)$ of length 1, $(1, 3)$ of length 3, and $(3, 2)$ of length -5 , Dijkstra finalizes $d[2] = 1$ at step 2, but the distance is $3 - 5 = -2$.

Why not just use Prim's algorithm?



Both algorithms grow a tree from vertex 1 by repeatedly adding one vertex, so the lecture asked whether the minimum spanning tree is a tree of shortest paths. It is not. In the graph above (undirected, so each edge may be used in either direction), the minimum spanning tree is the path 1–2–3–4 of total weight 3, but the shortest path from 1 to 4 is the direct edge of length 2, whereas the tree path has length 3. Prim's rule picks the lightest edge leaving S ; Dijkstra's rule picks the vertex with the smallest *total distance* $d[u] = d[x] + \ell(x, u)$, which is a different quantity.

Running time

With arrays. The set S can be stored as a 0/1 array ("add u to S " means $S[u] = 1$), and the minimum can be found by scanning the d array. There are n iterations, each scanning n entries, $O(n^2)$ in total, plus one relaxation per edge over the whole run, $O(m)$. Since $m \leq n^2$, the total is $O(n^2)$.

With a heap. Instead of scanning d for the minimum in every iteration, keep the unfinished vertices in a **priority queue** implemented as a **heap**, a binary tree in which every node's key (its d -value) is at most its children's keys, so the minimum sits at the top. In a heap, finding and removing the minimum takes $O(\log n)$ instead of $O(n)$, and changing a vertex's d -value takes $O(\log n)$ instead of $O(1)$. Dijkstra finds the minimum n times and changes a d -value at most m times (once per relaxation), for a total of $n \cdot O(\log n) + m \cdot O(\log n) = O((n + m) \log n)$, which is $O(m \log n)$ for a connected graph. The lecture mentioned, as a tangent from the textbook, that a Fibonacci heap improves this to $O(m + n \log n)$; that is not examinable.

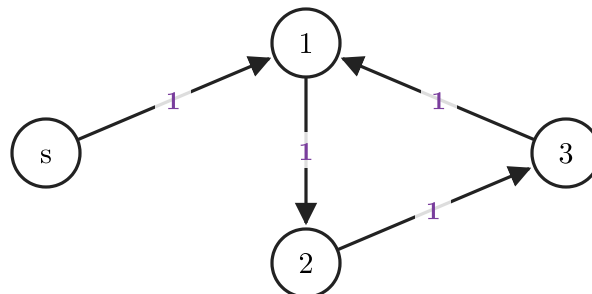
5.5 Choosing the right algorithm

A design problem on the exam will often be "model this situation as a shortest-path problem and call the right algorithm." Check the assumptions in this order.

1. Are all edge lengths the same? Use **BFS**, $O(m + n)$.
2. Is the graph a DAG? Use **DAG DP**, $O(m + n)$, negative lengths allowed, and remember it also does longest and widest paths.
3. Are all lengths nonnegative? Use **Dijkstra**, $O(n^2)$ or $O(m \log n)$.
4. Otherwise, if there is no negative cycle, use **Bellman-Ford**, $O(mn)$.

Undirected graphs with nonnegative lengths are handled by replacing each edge $\{u, v\}$ with the two directed edges (u, v) and (v, u) . A problem that limits the number of edges ("at most k edges," "at most k stops") is asking for column k of the Bellman-Ford table. A problem with a few extra states ("you may skip one toll") is usually a DAG DP or Dijkstra on a graph with a copy of each vertex per state.

5.6 Longest paths and cycles



The last question of Day 2 was whether the Bellman-Ford logic can compute longest paths in a general graph. It cannot. In the graph above, with every edge of length 1, the walk $s \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow \dots$ gets longer every time it goes around the cycle, so "the longest walk" does not exist and the

recurrence keeps increasing with each column, the lecture's "problem!" Longest paths behave well only in DAGs (Section 5.2). Extra context: finding the longest simple path, with no repeated vertices, in a general graph is a problem with no known polynomial-time algorithm, in the same family as the independent set problem of Section 4.6.

5.7 The fourth section of Chapter 5

The export ends with the lecture of October 22, which covered sections 5.1 through 5.3. The lecture of October 27 presumably covered the fourth section, and the Day 1 notes say that three of the four sections are dynamic programs, so the fourth is a DP as well. The standard fourth topic in a shortest-paths chapter, and the only widely taught shortest-path DP not already covered, is the **Floyd-Warshall** algorithm for *all-pairs* shortest paths. What follows is extra context, not taken from the course notes; check it against your own notes from October 27.

Problem: a directed graph with lengths and no negative cycle; return the distance between *every* pair of vertices. Subproblems: for every pair (u, v) and every k from 0 to n , $\text{OPT}[k][u][v]$ = the length of the shortest $u \rightarrow v$ path whose intermediate vertices (all vertices except the two endpoints) are among $1, \dots, k$. Base: $\text{OPT}[0][u][v] = \ell(u, v)$ if (u, v) is an edge, 0 if $u = v$, and ∞ otherwise. Recurrence: $\text{OPT}[k][u][v] = \min(\text{OPT}[k-1][u][v], \text{OPT}[k-1][u][k] + \text{OPT}[k-1][k][v])$, because the best path either avoids vertex k or passes through it once, going from u to k and then from k to v using only smaller intermediate vertices. Return $\text{OPT}[n][u][v]$. There are $n \cdot n^2$ subproblems at $O(1)$ each, so $O(n^3)$; and since layer k depends only on layer $k - 1$, the space-saving trick of Section 1.6 reduces the table to $n \times n$. For comparison, running Bellman-Ford from every vertex costs $O(mn^2)$ and running Dijkstra from every vertex costs $O(n^3)$ with arrays but requires nonnegative lengths.

Exercises

5.1. State the DAG shortest-path recurrence, say in what order the vertices are processed and why that order makes every needed value available, and give the running time.

$\text{OPT}[s] = 0$; $\text{OPT}[v] = \min$ over in-neighbors u of $(\text{OPT}[u] + \ell(u, v))$. Process vertices in topological order: every in-neighbor of v precedes v in that order, so $\text{OPT}[u]$ is computed before $\text{OPT}[v]$ is needed. Topological sort, building G^R , and the double loop are each $O(m + n)$.

5.2. Run DAG-DP from $s = 1$ on the DAG with edges $(1, 2)$ of length 3, $(1, 3)$ of length 1, $(3, 2)$ of length -2 , $(2, 4)$ of length 4, $(3, 4)$ of length 7, $(2, 5)$ of length 2, $(4, 5)$ of length -5 . Give a topological order and the array d .

Order 1, 3, 2, 4, 5 (2 must follow 3). $d[1] = 0$; $d[3] = 1$; $d[2] = \min(0 + 3, 1 - 2) = -1$; $d[4] = \min(-1 + 4, 1 + 7) = 3$; $d[5] = \min(-1 + 2, 3 - 5) = -2$. $d = [0, -1, 1, 3, -2]$.

5.3. For the same DAG, what are the longest-path values from 1, and what changes in the pseudocode?

Replace min by max and initialize d to $-\infty$ with $d[1] = 0$. $d[3] = 1$, $d[2] = \max(3, -1) = 3$, $d[4] = \max(3 + 4, 1 + 7) = 8$, $d[5] = \max(3 + 2, 8 - 5) = 5$. $d = [0, 3, 1, 8, 5]$.

5.4. Why can the DAG recurrence not be evaluated on a graph with a cycle, even when the graph has no negative cycle, and how does Bellman-Ford fix this?

On a cycle, $\text{OPT}[v]$ depends on $\text{OPT}[u]$ which (directly or indirectly) depends on $\text{OPT}[v]$, so there is no order in which all in-neighbors are finished first. Bellman-Ford adds the index j (number of edges allowed), so $\text{OPT}[v][j]$ depends only on entries in column $j - 1$, and the columns are filled in order.

5.5. Fill the Bellman-Ford table for $s = 1$ on the graph with edges (1, 2) of length 6, (1, 3) of length 7, (3, 2) of length -3 , (2, 4) of length -1 , (3, 4) of length 2, and give d.

$n = 4$, columns 0..3. Column 0: $[0, \infty, \infty, \infty]$. Column 1: $[0, 6, 7, \infty]$. Column 2: $d[2] = \min(6, 7 - 3) = 4$, $d[3] = 7$, $d[4] = \min(\infty, 6 - 1, 7 + 2) = 5$: $[0, 4, 7, 5]$. Column 3: $d[4] = \min(5, 4 - 1, 9) = 3$: $[0, 4, 7, 3]$. $d = [0, 4, 7, 3]$.

5.6. Explain why column $n - 1$ of the Bellman-Ford table equals the true distances when there is no negative cycle.

With no negative cycle, removing a cycle from a walk never makes it longer, so a shortest walk can be taken to be a simple path, which has at most $n - 1$ edges. Column $n - 1$ is the shortest walk with at most $n - 1$ edges, hence the distance.

5.7. On the graph with edges (1, 2) of length 5, (1, 3) of length 2, (2, 3) of length -4 and $s = 1$, give column 1 of the table and the array of Bellman-Ford-2 after round 1 when edges are relaxed in the order (1, 2), (1, 3), (2, 3). Explain the difference.

Column 1 is $[0, 5, 2]$ (one edge). Bellman-Ford-2 after round 1: relaxing (1, 2) sets $d[2] = 5$, (1, 3) sets $d[3] = 2$, and (2, 3) uses the new $d[2]$ to set $d[3] = 1$, giving $[0, 5, 1]$. The in-place version can propagate an improvement within the same round, so it may run ahead of the table; both give $[0, 5, 1]$ at the end.

5.8. Run Dijkstra from $s = 1$ on the graph with edges (1, 2) of length 4, (1, 3) of length 1, (3, 2) of length 2, (2, 4) of length 5, (3, 4) of length 8, (2, 5) of length 1. Give the order in which vertices enter S and the final d (break ties by smaller vertex number).

Start $[0, \infty, \infty, \infty, \infty]$. Choose 1: $d[2] = 4$, $d[3] = 1$. Choose 3 (1): $d[2] = \min(4, 3) = 3$, $d[4] = 9$. Choose 2 (3): $d[4] = \min(9, 8) = 8$, $d[5] = 4$. Choose 5 (4): no outgoing edges. Choose 4 (8). Order 1, 3, 2, 5, 4; $d = [0, 3, 1, 8, 4]$.

5.9. Give a graph with a negative edge on which Dijkstra returns a wrong distance, and say which step goes wrong.

Edges (1, 2) of length 1, (1, 3) of length 3, (3, 2) of length -5 , $s = 1$. Dijkstra finalizes vertex 2 with $d[2] = 1$ before looking at vertex 3, but the path $1 \rightarrow 3 \rightarrow 2$ has length -2 . The step "the chosen vertex's value is final" relies on every edge being nonnegative.

5.10. Compare the running times of Dijkstra with an array and with a heap, and say for which graphs each is better.

Array: n scans of $O(n)$ plus $O(m)$ relaxations, $O(n^2)$. Heap: n extractions and at most m key changes at $O(\log n)$ each, $O((n + m) \log n) = O(m \log n)$ when connected. The heap wins on sparse graphs (m much smaller than $n^2/\log n$); the array is as good or better on dense graphs (m near n^2).

5.11. (Design.) The input is a directed graph with nonnegative lengths, vertices s and t , and an integer k . Describe an $O(mk)$ -time algorithm that returns the length of the shortest $s \rightarrow t$ walk that uses at most k edges.

Run the table version of Bellman-Ford for k columns instead of $n - 1$ and return $\text{OPT}[t][k]$. Subproblems $\text{OPT}[v][j]$ = shortest walk to v with at most j edges; recurrence as in Section 5.3; correctness is the same case analysis on the last edge; k columns at $O(m + n) = O(m)$ each gives $O(mk)$.

5.12. (Design.) An $n \times n$ grid has a positive cost $c[i][j]$ in each cell. Starting at the top-left cell you may move only right or down, paying the cost of every cell you enter, including the first. Describe an $O(n^2)$ algorithm for the minimum cost of reaching the bottom-right cell.

The grid is a DAG (edges go right and down). Subproblems $\text{OPT}[i][j]$ = minimum cost of reaching cell (i, j) . Base $\text{OPT}[1][1] = c[1][1]$; first row $\text{OPT}[1][j] = \text{OPT}[1][j-1] + c[1][j]$; first column similarly. Otherwise $\text{OPT}[i][j] = c[i][j] + \min(\text{OPT}[i-1][j], \text{OPT}[i][j-1])$ (the last move came from above or from the left). Fill row by row; n^2 subproblems at $O(1)$ each; return $\text{OPT}[n][n]$. This is DAG-DP with the topological order "row by row."

5.13. (Design.) The input is a directed graph with nonnegative lengths and vertices s, t . You may make one edge free (treat its length as 0). Describe an $O(m \log n)$ -time algorithm for the length of the cheapest $s \rightarrow t$ path under this rule.

Build a graph with two copies of every vertex: $(v, 0)$ meaning "free edge not yet used" and $(v, 1)$ meaning "used." For each edge (u, v) add $(u, 0) \rightarrow (v, 0)$ and $(u, 1) \rightarrow (v, 1)$ of length $\ell(u, v)$, and $(u, 0) \rightarrow (v, 1)$ of length 0. All lengths are nonnegative, so Dijkstra from $(s, 0)$ on this graph of $2n$ vertices and $3m$ edges takes $O(m \log n)$; return $\min(d[(t, 0)], d[(t, 1)])$. Correct because every walk in the new graph corresponds to a walk in the old one with at most one edge made free, and vice versa.

6. Practice Midterm 2, version A

This is a full practice exam in the format of Midterm 1: 25 points, 65 minutes, closed-everything. Problem 1 has five one-point parts answered without explanation; Problems 2 through 5 each ask for an algorithm, a correctness argument, and a running-time analysis. Set a timer, write your answers on paper, and only then open the solutions. The official practice midterm, when it is posted, should be worked the same way.

Problem 1 (five parts, one point each, no explanation)

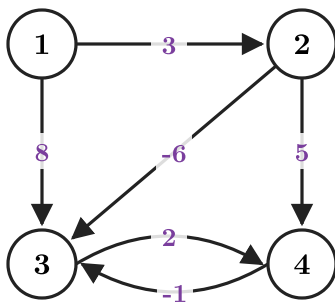
(a) Run the LIS algorithm of Chapter 2 on $A = [3, 8, 2, 5, 6, 1, 7]$. What are the values in the array d ?

$d = [1, 2, 1, 2, 3, 1, 4]$. (The algorithm returns $\max(d) = 4$, for example $[3, 5, 6, 7]$.)

(b) Run Edit-Distance on $A = \text{cat}$ and $B = \text{act}$. What is the last row of the table, the row for the prefix "cat"? (List the entries for the column prefixes "", a, ac, act in that order.)

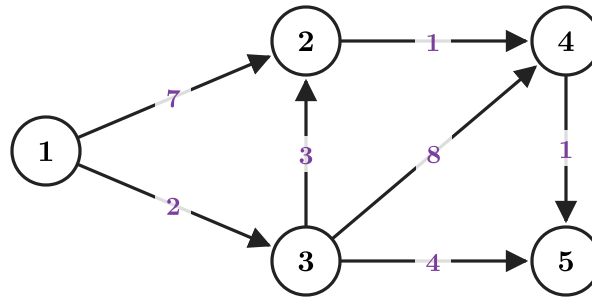
$[3, 2, 2, 2]$. The full table, rows "", c, a, t: $[0, 1, 2, 3], [1, 1, 1, 2], [2, 1, 2, 2], [3, 2, 2, 2]$. $\text{ED}(\text{cat}, \text{act}) = 2$ (delete c, insert c after a).

(c) Run the table version of Bellman-Ford from $s = 1$ on the graph below. What is column $j = 2$ of the table? (List the entries for vertices 1, 2, 3, 4.)



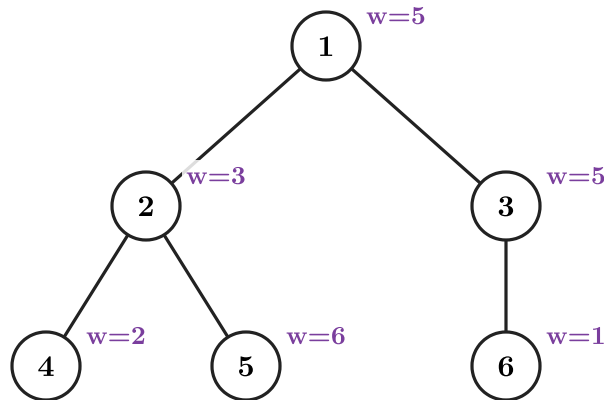
$[0, 3, -3, 8]$. Column 1 is $[0, 3, 8, \infty]$. In column 2, $d[3] = \min(8, d[2]_1 - 6, d[4]_1 - 1) = -3$ and $d[4] = \min(\infty, d[2]_1 + 5, d[3]_1 + 2) = 8$. (Column 3, the answer, is $[0, 3, -3, -1]$.)

(d) Run Dijkstra from $s = 1$ on the graph below, breaking ties by the smaller vertex number. In what order are the vertices added to S ?



1, 3, 2, 4, 5. After adding 1: $d = [0, 7, 2, \infty, \infty]$. Add 3 (2): $d[2] = 5$, $d[4] = 10$, $d[5] = 6$. Add 2 (5): $d[4] = 6$. Add 4 (6, tie with 5): $d[5]$ stays 6. Add 5. Final $d = [0, 5, 2, 6, 6]$.

(e) Run MIS-Tree on the tree below, rooted at 1, with the weights shown. What are $\text{OPT}_{\text{in}}[1]$ and $\text{OPT}_{\text{out}}[1]$?



$\text{OPT}_{\text{in}}[1] = 14$ and $\text{OPT}_{\text{out}}[1] = 13$. Leaves: $4 \rightarrow (2, 0)$, $5 \rightarrow (6, 0)$, $6 \rightarrow (1, 0)$. Vertex 2: $\text{in} = 3 + 0 + 0 = 3$, $\text{out} = 2 + 6 = 8$. Vertex 3: $\text{in} = 5$, $\text{out} = 1$. Root: $\text{in} = 5 + 8 + 1 = 14$, $\text{out} = \max(3, 8) + \max(5, 1) = 13$. The answer to the problem would be 14, the set $\{1, 4, 5, 6\}$.

Problem 2. Non-adjacent elements

The input is an array A of n positive integers. Our goal is to return the maximum possible sum of a set of elements of A , no two of which are adjacent (that is, you may not take both $A[i]$ and $A[i+1]$ for any i). Describe an $O(n)$ -time algorithm for this problem, explain why it is correct, and analyze its running time. Example: if $A = [5, 1, 1, 5]$, the algorithm should return 10.

Subproblems. For i from 1 to n , $\text{OPT}[i]$ = the maximum sum of a set of non-adjacent elements of $A[1:i]$. We return $\text{OPT}[n]$.

Recurrence. $\text{OPT}[1] = A[1]$, $\text{OPT}[2] = \max(A[1], A[2])$, and for $i \geq 3$, $\text{OPT}[i] = \max(\text{OPT}[i-1], A[i] + \text{OPT}[i-2])$.

```
ALG(A):  
    d = [0] * n  
    d[1] = A[1]; d[2] = max(A[1], A[2])  
    for i = 3, ..., n:  
        d[i] = max(d[i-1], A[i] + d[i-2])  
    return d[n]
```

Correctness. Consider an optimal set for $A[1:i]$. If it does not contain $A[i]$, it is a set of non-adjacent elements of $A[1:i-1]$, so its sum is at most $\text{OPT}[i-1]$, and $\text{OPT}[i-1]$ is achievable. If it contains $A[i]$, it cannot contain $A[i-1]$, so the rest is a non-adjacent set in $A[1:i-2]$, with sum at most $\text{OPT}[i-2]$; and $A[i]$ plus an optimal set for $A[1:i-2]$ is feasible. The recurrence takes the better case. The base cases are the best sets of one and two elements. (For the example, $d = [5, 5, 6, 10]$.)

Running time. n subproblems, $O(1)$ each, so $O(n)$.

Problem 3. Longest common substring

The input is two strings A and B of lengths m and n . A *substring* is a contiguous block of characters. Our goal is to return the length of the longest string that is a substring of both A and B . Describe an $O(mn)$ -time algorithm, explain why it is correct, and analyze its running time. Example: for $A = \text{xabcy}$ and $B = \text{zabca}$ the algorithm should return 3 (abc).

Subproblems (Pattern 6 with the LIS trick). For $0 \leq i \leq m$ and $0 \leq j \leq n$, $\text{OPT}[i][j]$ = the length of the longest string that is simultaneously a suffix of $A[1:i]$ and a suffix of $B[1:j]$, that is, the longest common block that *ends* at $A[i]$ and at $B[j]$. We return the maximum of $\text{OPT}[i][j]$ over all i, j .

Recurrence. $\text{OPT}[0][j] = \text{OPT}[i][0] = 0$. For $i, j \geq 1$: $\text{OPT}[i][j] = \text{OPT}[i-1][j-1] + 1$ if $A[i] = B[j]$, and 0 otherwise.

```
ALG(A, B):  
    d = (m+1) x (n+1) table, all 0  
    best = 0  
    for i = 1, ..., m:  
        for j = 1, ..., n:  
            if A[i] = B[j]:  
                d[i][j] = d[i-1][j-1] + 1  
                best = max(best, d[i][j])  
    return best
```

Correctness. A common block ending at $A[i]$ and $B[j]$ must have $A[i] = B[j]$ as its last character, and removing that character leaves a common block ending at $A[i-1]$ and $B[j-1]$ (possibly empty), of length at most $\text{OPT}[i-1][j-1]$; conversely any such block extends by one. If $A[i] \neq B[j]$ no common block ends there. Every common substring ends at some pair (i, j) , so the maximum over the table is the answer.

Running time. $(m + 1)(n + 1)$ subproblems at $O(1)$ each: $O(mn)$.

Problem 4. Heaviest downward path in a tree

The input is a tree T rooted at vertex 1 ($T[u]$ is the list of children of u) in which each vertex u has a weight $w[u]$, which may be positive or negative. A *downward path* starts at some vertex and repeatedly moves to a child; it may consist of a single vertex. Our goal is to return the maximum total weight of a downward path. Describe an $O(n)$ -time algorithm, explain why it is correct, and analyze its running time.

Subproblems (tree pattern with the LIS trick). For every vertex u , $\text{OPT}[u]$ = the maximum weight of a downward path that *starts at u* . We return $\max$ over all u of $\text{OPT}[u]$, since the best path can start anywhere.

Recurrence. If u is a leaf, $\text{OPT}[u] = w[u]$. Otherwise $\text{OPT}[u] = w[u] + \max(0, \max_{v \in T[u]} \text{OPT}[v])$.

```
ALG(T, w):
    L = vertices in an order with every child before its parent    # reverse BFS order from
    1
    d = [0] * n
    for u in L:
        best = 0
        for v in T[u]: best = max(best, d[v])
        d[u] = w[u] + best
    return max(d)
```

Correctness. A downward path starting at u either stops at u , with weight $w[u]$, or continues into some child v , and what follows is a downward path starting at v , of weight at most $\text{OPT}[v]$; conversely u followed by an optimal path from v is a downward path. So the best is $w[u]$ plus the better of 0 (stop) and the best child value. Children are processed before parents, so every $d[v]$ is final when $d[u]$ is computed. Every downward path starts somewhere, so the maximum over u is the answer.

Running time. Computing the order is a BFS on a tree, $O(n)$. Each vertex is examined once as a child, so the loops take $O(n)$. Total $O(n)$.

Problem 5. Counting paths in a DAG

The input is a DAG G and a vertex s . Our goal is to return an array c such that for every vertex v , $c[v]$ is the number of distinct paths from s to v (a path is a sequence of distinct vertices joined by edges; $c[s] = 1$ for the empty path). Describe an $O(m + n)$ -time algorithm, explain why it is correct, and analyze its running time. Example: if $G = [[2, 3], [3], []]$ then $c = [1, 1, 2]$.

Subproblems. For every vertex v , $\text{OPT}[v]$ = the number of paths from s to v . We return the whole array.

Recurrence. $\text{OPT}[s] = 1$. For $v \neq s$, $\text{OPT}[v] = \sum$ over in-neighbors u of v of $\text{OPT}[u]$ (0 if v has no in-neighbor).

```
ALG( $G, s$ ):  
   $c = [0] * n$ ;  $c[s] = 1$   
  topologically sort  $V$   
   $G\_R = \text{reverse of } G$   
  for  $v$  in  $V$ , in topological order, skipping  $s$ :  
    for  $u$  in  $G\_R[v]$ :  
       $c[v] = c[v] + c[u]$   
  return  $c$ 
```

Correctness. Every path from s to $v \neq s$ has a last edge (u, v) , and removing it gives a path from s to u ; different paths to v give different $(u, \text{path-to-}u)$ pairs and vice versa. So the number of paths to v is the sum over in-neighbors u of the number of paths to u . In a DAG every in-neighbor of v precedes v in topological order, so $c[u]$ is final when it is used. Vertices that cannot be reached from s keep $c = 0$ and contribute nothing to their out-neighbors, which is correct.

Running time. Topological sort and building G^R are $O(m + n)$, and the loop scans each reversed list once, $O(m + n)$. Total $O(m + n)$.

WHAT TO NOTICE. Problems 2 and 4 are the LIS trick in one dimension and on a tree; Problem 3 is the LIS trick in two dimensions; Problem 5 is the DAG recurrence with a sum instead of a minimum. The four-step layout is the same every time, and the correctness paragraph is always "the optimal (or every) solution ends in one of these ways, and the recurrence accounts for each."

7. Practice Midterm 2, version B

A second full exam in the same format, with a different mix of topics, followed by a set of short questions of the kind that are easy to lose points on. Take it after you have reviewed your mistakes from version A.

Problem 1 (five parts, one point each, no explanation)

(a) Run the minimum-cost-jump algorithm of Section 2.3 on $A = [5, 2, 9, 1, 3, 4]$. What are the values in the array d , and what is returned?

$d = [5, 2, 11, 3, 6, 7]$; returned value $\min(d[5], d[6]) = 6$ (land on 2, 1, 3, then jump out).

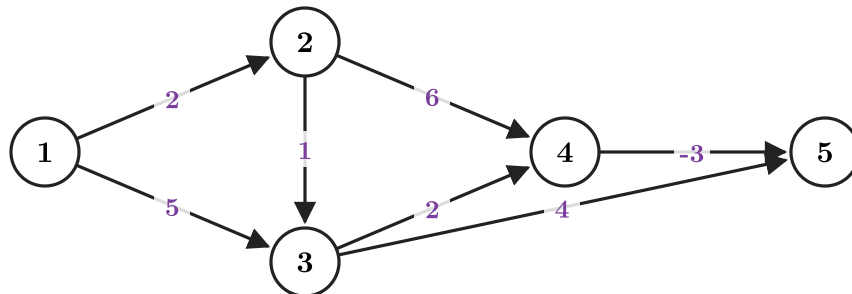
(b) Run LPS on $A = \text{abcab}$. What is the first row of the table, $\text{OPT}[1][1]$ through $\text{OPT}[1][5]$?

$[1, 1, 1, 3, 3]$. The answer is $\text{OPT}[1][5] = 3$ (aba, bab, or aca).

(c) Run Knapsack on $v = [6, 10, 12]$, $w = [1, 2, 3]$, $B = 5$. What is the last row of the table, $\text{OPT}[3][0]$ through $\text{OPT}[3][5]$?

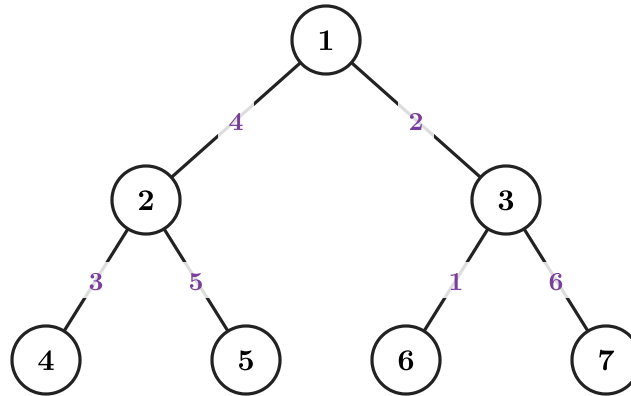
$[0, 6, 10, 16, 18, 22]$. Rows: item 1: $[0, 6, 6, 6, 6, 6]$; item 2: $[0, 6, 10, 16, 16, 16]$; item 3: $[0, 6, 10, 16, 18, 22]$. The answer 22 takes items 2 and 3.

(d) Run DAG-DP (shortest paths) from $s = 1$ on the DAG below. What are the values in the array d ?



$d = [0, 2, 3, 5, 2]$. Topological order 1, 2, 3, 4, 5: $d[3] = \min(5, 2 + 1) = 3$, $d[4] = \min(2 + 6, 3 + 2) = 5$, $d[5] = \min(3 + 4, 5 - 3) = 2$.

(e) Run the maximum-weight matching algorithm of Section 4.5 on the tree below, rooted at 1, with the edge weights shown. What is $\text{OPT}[1]$?



11. Vertex 2: $\text{OPT}_{\text{out}} = 0$, $\text{OPT} = \max(0, 3, 5) = 5$. Vertex 3: $\text{OPT}_{\text{out}} = 0$, $\text{OPT} = \max(0, 1, 6) = 6$. Root: $\text{OPT}_{\text{out}} = 5 + 6 = 11$; matching the root to 2 gives $4 + 0 + (11 - 5) = 10$, to 3 gives $2 + 0 + (11 - 6) = 7$; $\text{OPT}[1] = 11$ (edges 2-5 and 3-7).

Problem 2. Subset sum

The input is an array A of n positive integers and a positive integer k . Our goal is to return true if some subset of the elements of A has sum exactly k , and false otherwise. Describe an $O(nk)$ -time algorithm, explain why it is correct, and analyze its running time. Example: for $A = [3, 34, 4, 12, 5]$ and $k = 9$ the answer is true ($4 + 5$).

Subproblems (Pattern 5). For $0 \leq i \leq n$ and $0 \leq j \leq k$, $\text{OPT}[i][j] = \text{true}$ if some subset of $A[1:i]$ has sum exactly j , and false otherwise. We return $\text{OPT}[n][k]$.

Recurrence. $\text{OPT}[0][0] = \text{true}$ and $\text{OPT}[0][j] = \text{false}$ for $j \geq 1$ (the empty set sums to 0 only). For $i \geq 1$: $\text{OPT}[i][j] = \text{OPT}[i-1][j]$ if $A[i] > j$; otherwise $\text{OPT}[i][j] = \text{OPT}[i-1][j]$ or $\text{OPT}[i-1][j - A[i]]$.

```
ALG(A, k):
    d = (n+1) x (k+1) table, all false; d[0][0] = true
    for i = 1, ..., n:
        for j = 0, ..., k:
            d[i][j] = d[i-1][j]
            if A[i] <= j and d[i-1][j - A[i]]: d[i][j] = true
    return d[n][k]
```

Correctness. A subset of $A[1:i]$ with sum j either omits $A[i]$, in which case it is a subset of $A[1:i-1]$ with sum j , or includes $A[i]$, in which case the rest is a subset of $A[1:i-1]$ with sum $j - A[i]$ (which requires $A[i] \leq j$). Both directions hold, so $\text{OPT}[i][j]$ is true exactly when one of the two smaller statements is true. Row i uses only row $i - 1$, so filling row by row works.

Running time. $(n + 1)(k + 1)$ subproblems at $O(1)$ each: $O(nk)$. (This is the 0/1 knapsack table with "true/false" in place of "maximum value.")

Problem 3. Deleting characters to make two strings equal

The input is two strings A and B of lengths m and n . In one move you may delete one character from either string. Our goal is to return the minimum number of moves needed to make the two strings identical. Describe an $O(mn)$ -time algorithm, explain why it is correct, and analyze its running time. Example: for $A = \text{sea}$ and $B = \text{eat}$ the answer is 2 (delete s from A and t from B , leaving ea).

Idea. After all deletions the two strings are the same string C , which is a subsequence of both A and B . The number of moves is $(m - |C|) + (n - |C|)$, so we want C as long as possible: a longest common subsequence.

Subproblems (Pattern 6). For $0 \leq i \leq m$ and $0 \leq j \leq n$, $\text{OPT}[i][j]$ = the length of the longest common subsequence of $A[1:i]$ and $B[1:j]$. We return $m + n - 2 \cdot \text{OPT}[m][n]$.

Recurrence. $\text{OPT}[0][j] = \text{OPT}[i][0] = 0$. For $i, j \geq 1$: if $A[i] = B[j]$, $\text{OPT}[i][j] = \text{OPT}[i-1][j-1] + 1$; otherwise $\text{OPT}[i][j] = \max(\text{OPT}[i-1][j], \text{OPT}[i][j-1])$.

```
ALG(A, B):
    d = (m+1) x (n+1) table, all 0
    for i = 1, ..., m:
        for j = 1, ..., n:
            if A[i] = B[j]: d[i][j] = d[i-1][j-1] + 1
            else:           d[i][j] = max(d[i-1][j], d[i][j-1])
    return m + n - 2 * d[m][n]
```

Correctness. Any sequence of moves leaves a common subsequence C of length L and uses exactly $m + n - 2L$ moves, and conversely any common subsequence can be reached by deleting everything else, so the minimum number of moves is $m + n - 2 \cdot (\text{LCS length})$. For the recurrence: if $A[i] = B[j]$, some longest common subsequence of the two prefixes ends with that matched pair (if it does not use both, replace its last character with the pair without shortening it), so it is one longer than an LCS of $A[1:i-1]$, $B[1:j-1]$; if $A[i] \neq B[j]$, an LCS cannot use both $A[i]$ and $B[j]$, so it is an LCS of $A[1:i-1]$, $B[1:j]$ or of $A[1:i]$, $B[1:j-1]$, and the maximum covers both.

Running time. $O(mn)$ subproblems at $O(1)$ each, plus $O(1)$ to compute the final expression: $O(mn)$. (For the example the LCS is ea , so the answer is $3 + 3 - 4 = 2$.)

Problem 4. Heaviest connected piece containing the root

The input is a tree T rooted at vertex 1 in which every vertex u has a weight $w[u]$, positive or negative. Our goal is to return the maximum total weight of a set S of vertices such that S contains the root and S is connected (any two vertices of S are joined by a path that stays inside S). Describe an $O(n)$ -time algorithm, explain why it is correct, and analyze its running time.

Subproblems. For every vertex u , $\text{OPT}[u]$ = the maximum weight of a connected set inside $T(u)$ that contains u . We return $\text{OPT}[1]$.

Recurrence. If u is a leaf, $\text{OPT}[u] = w[u]$. Otherwise $\text{OPT}[u] = w[u] + \sum_{v \in T[u]} \max(0, \text{OPT}[v])$.

```
ALG(T, w):  
  L = vertices in an order with every child before its parent  
  d = [0] * n  
  for u in L:  
    d[u] = w[u]  
    for v in T[u]: d[u] = d[u] + max(0, d[v])  
  return d[1]
```

Correctness. A connected set S containing u meets each child subtree $T(v)$ either not at all, or in a connected set that contains v (if S contains any vertex of $T(v)$, the path from it to u passes through v and lies in S). The child subtrees do not touch each other, so the choices for different children are independent, and for each child the best is the larger of 0 (take nothing) and $\text{OPT}[v]$. Conversely, u together with any such choices is connected and contains u . Children are processed before parents, so each $d[v]$ is final when used.

Running time. The order is computed by a BFS from the root in $O(n)$, and each vertex is scanned once as a child, $O(n)$. Total $O(n)$.

Problem 5. Shortest path through a required vertex

The input is a directed graph G with nonnegative edge lengths and three vertices s, x, t . Our goal is to return the length of the shortest walk from s to t that visits x . Describe an $O(m \log n)$ -time algorithm, explain why it is correct, and analyze its running time.

Algorithm. Run Dijkstra twice: $d_1 = \text{Dijkstra}(G, s)$ and $d_2 = \text{Dijkstra}(G, x)$. Return $d_1[x] + d_2[t]$ (which is ∞ if either part is ∞).

```
ALG(G, s, x, t):  
    d1 = Dijkstra(G, s)  
    d2 = Dijkstra(G, x)  
    return d1[x] + d2[t]
```

Correctness. Any walk from s to t that visits x splits, at its first visit to x , into a walk from s to x and a walk from x to t , of lengths at least $d_1[x]$ and $d_2[t]$ respectively, so no such walk is shorter than $d_1[x] + d_2[t]$. Conversely, a shortest path from s to x followed by a shortest path from x to t is a walk from s to t through x of exactly that length. Dijkstra computes exact distances because all lengths are nonnegative.

Running time. Two runs of Dijkstra with a heap, $O(m \log n)$ each, plus $O(1)$: $O(m \log n)$.

Part 6. Short questions

6.1. A student's recurrence for the longest increasing subsequence is $\text{OPT}[i] = \max(\text{OPT}[i-1], 1 + \text{OPT}[i-1] \text{ if } A[i] > A[i-1])$. Give an input on which it is wrong.

$A = [1, 5, 2, 3]$: the LIS is $[1, 2, 3]$ of length 3, but the recurrence only compares adjacent elements and gives 2. It mistakes the contiguous problem for the general one.

6.2. Which entry is returned, and why: for LIS, for the minimum-cost jump, for edit distance, for LPS, for knapsack, for the Alice-and-Bob game?

LIS: $\max(d)$, the best subsequence can end anywhere. Jumps: $\min(d[n-1], d[n])$, the last landing is on n or $n - 1$. Edit distance: $d[m][n]$, the whole strings. LPS: $d[1][n]$, the whole string. Knapsack: $d[n][B]$, all items and full capacity. Game: $d[1]$, the whole array is the starting position.

6.3. In which order is the table filled for: Pattern 2 (suffixes), Pattern 4 (intervals), tree problems, DAG shortest paths, Bellman-Ford?

Suffixes: $i = n$ down to 1. Intervals: bottom row up (i decreasing), left to right within a row, equivalently by increasing interval length. Trees: children before parents (reverse BFS order or DFS post order). DAG: topological order. Bellman-Ford: column by column, $j = 1$ to $n - 1$.

6.4. State one input condition for each of BFS, DAG-DP, Dijkstra, and Bellman-Ford, and the running time of each.

BFS: all lengths equal and positive, $O(m + n)$. DAG-DP: no cycles, $O(m + n)$. Dijkstra: all lengths ≥ 0 , $O(n^2)$ or $O(m \log n)$. Bellman-Ford: no negative cycle, $O(mn)$.

6.5. True or false: after round j of the space-saving Bellman-Ford, $d[v]$ equals the length of the shortest walk to v with at most j edges. Explain.

False in general (true for $j = 0$ and $j = n - 1$). Within a round, an edge relaxed after its tail was improved passes the improvement on at once, so $d[v]$ can be smaller than the table entry; it is always the length of some walk, so it is never smaller than the true distance.

6.6. Why does the one-array 0/1 knapsack process capacities from B down to 0?

So that $d[j - w[i]]$ still holds the previous row's value when $d[j]$ is computed; going upward would reuse item i and compute the unbounded knapsack.

6.7. The 0/1 knapsack algorithm runs in $O(nB)$. Is that $O(n^2)$ when $B = n^2$? When $B = 2^n$?

With $B = n^2$ it is $O(n^3)$, still polynomial in n . With $B = 2^n$ it is $O(n \cdot 2^n)$, exponential; the running time depends on the size of the number B , which is why the algorithm is only efficient for moderate capacities.

6.8. What does "relax the edge (u, v) " mean, and which two algorithms of Chapter 5 consist entirely of relaxations in some order?

Set $d[v] = \min(d[v], d[u] + \ell(u, v))$. Bellman-Ford-2 relaxes every edge $n - 1$ times in rounds; Dijkstra relaxes the edges leaving each vertex once, in order of increasing d ; DAG-DP relaxes the edges into each vertex in topological order.

6.9. Give the recurrence for the number of ways to climb n stairs taking 1 or 2 steps at a time, and the running time.

$\text{OPT}[i]$ = number of ways to reach stair i : $\text{OPT}[0] = 1$, $\text{OPT}[1] = 1$, $\text{OPT}[i] = \text{OPT}[i-1] + \text{OPT}[i-2]$. The last step was of size 1 or 2. $O(n)$. (These are Fibonacci numbers.)

6.10. Top-down and bottom-up DP compute the same table. Name one advantage of each.

Bottom-up: no recursion, simpler running-time analysis (count loop iterations), easy to save space. Top-down: only the subproblems actually needed are computed, and the code follows the recurrence literally.

8. Summary

This chapter collects the recipes, recurrences, and facts of the book in the form of a single reference page. Read it the night before, and use it to check that every line means something to you.

WRITING A DP ANSWER. (1) Subproblems: "for every i , $\text{OPT}[i] = \dots$ " and "we return $\dots$ ". (2) Recurrence with base cases. (3) Pseudocode: array d filled in an order where everything needed is ready. (4) Running time = number of subproblems $\times$ time per subproblem (+ precomputation). Correctness = list the cases the optimal solution falls into, say the list is complete, say the formula scores each case correctly.

THE SIX PATTERNS. 1 prefix $A[1:i] \rightarrow$ return $\text{OPT}[n]$. 2 suffix $A[i:n] \rightarrow$ fill right to left, return $\text{OPT}[1]$. 3 prefix that must use $A[i]$ ("LIS trick") $\rightarrow$ return max or min over i . 4 interval $A[i:j] \rightarrow$ bottom row up, return $\text{OPT}[1][n]$. 5 ($A[1:i]$, budget j) $\rightarrow$ return $\text{OPT}[n][B]$. 6 ($A[1:i]$, $B[1:j]$) $\rightarrow$ return $\text{OPT}[m][n]$. Trees: children before parents, return at the root. DAGs: topological order.

ARRAYS, ONE INDEX. Max: $\text{OPT}[i] = \max(\text{OPT}[i-1], A[i])$. LIS: $\text{OPT}[i] = 1 + \max\{\text{OPT}[j] : j < i, A[j] < A[i]\}$, $O(n^2)$, return $\max(d)$, parent pointers recover it. LCIS: $\text{OPT}[i] = \text{OPT}[i-1] + 1$ if $A[i] > A[i-1]$ else 1, $O(n)$. Jumps: $\text{OPT}[i] = A[i] + \min(\text{OPT}[i-1], \text{OPT}[i-2])$, return $\min(d[n-1], d[n])$. Stock: $d[j] = A[j] - \min(A[1:j-1])$ with a running minimum, $O(n)$. Max-sum subarray: $\text{OPT}[i] = A[i] + \max(0, \text{OPT}[i-1])$. Non-adjacent sum: $\text{OPT}[i] = \max(\text{OPT}[i-1], A[i] + \text{OPT}[i-2])$.

FIBONACCI AND EVALUATION ORDER. $F(1) = F(2) = 1$, $F(i) = F(i-1) + F(i-2)$. Bottom-up loop $O(n)$ (recommended). Pure recursion $O(\phi^n)$ because subproblems repeat. Top-down = recursion + memoization (store in a global array, return if already computed), $O(n)$. Space saving: keep only the entries the recurrence reads (one number for Max, two for Fibonacci, one row for knapsack); not possible for LIS; costs interpretability and parent pointers.

INTERVALS: LPS. $\text{OPT}[i][j]$ = longest palindromic subsequence of $A[i:j]$; $\text{OPT}[i][i] = 1$, $\text{OPT}[i][j] = 0$ for $i > j$; if $A[i] = A[j]$: $2 + \text{OPT}[i+1][j-1]$, else $\max(\text{OPT}[i+1][j], \text{OPT}[i][j-1])$. Fill $i = n \dots 1$, $j = i \dots n$. $O(n^2)$. $acbcabda \rightarrow 5$; $abaca \rightarrow 3$. Palindromic substring: boolean table, $A[i] = A[j]$ and $\text{OPT}[i+1][j-1]$.

KNAPSACK. $\text{OPT}[i][j]$ = best value from items $1..i$ with capacity j ; $\text{OPT}[i][0] = 0$; $\text{OPT}[1][j] = v[1]$ if $w[1] \leq j$ else 0; if $w[i] > j$: $\text{OPT}[i-1][j]$, else $\max(\text{OPT}[i-1][j], v[i] + \text{OPT}[i-1][j-w[i]])$. $O(nB)$. Unbounded: $v[i] + \text{OPT}[i][j-w[i]]$, base $\lfloor j/w[1] \rfloor \cdot v[1]$. One array: j from B down to 0 (upward = unbounded). Subset sum: same table with true/false. $v = [4, 2, 3, 5]$, $w = [3, 1, 2, 2]$, $B = 5 \rightarrow 10$.

ALICE AND BOB. Suffix sums $S[i] = A[i] + S[i+1]$, $S[n+1] = 0$. $\text{OPT}[i]$ = best score of the player to move on $A[i:n] = S[i] - \min(\text{OPT}[i+1], \text{OPT}[i+2], \text{OPT}[i+3])$; $\text{OPT}[i] = S[i]$ for $i \geq n-2$. Fill $i = n \dots 1$, return $\text{OPT}[1]$. $O(n)$ with S precomputed. $[4, 1, 5, 2, 3, 10, 1, 2] \rightarrow 18$.

EDIT DISTANCE. $\text{OPT}[i][j] = \text{ED}(A[1:i], B[1:j])$; $\text{OPT}[0][j] = j$, $\text{OPT}[i][0] = i$; $\min(\text{left} + 1 \text{ insert, up} + 1 \text{ delete, diagonal} + \delta \text{ replace, } \delta = 0 \text{ if } A[i] = B[j])$. If characters match, copy the diagonal. Fill row by row, $O(mn)$, return $\text{OPT}[m][n]$. $\text{ED}(\text{star, water}) = 3$. LCS: match $\rightarrow$ diagonal + 1, else $\max(\text{up, left})$, base 0. Longest common substring: match $\rightarrow$ diagonal + 1, else 0, return max.

TREES. Input $T[u]$ = children, root 1, $n - 1$ edges. Order: children before parents (reverse BFS order, or DFS post order). MIS: $\text{in}[u] = w[u] + \sum \text{out}[v]$; $\text{out}[u] = \sum \max(\text{in}[v], \text{out}[v])$; leaf $(w[u], 0)$; return $\max(\text{in}[1], \text{out}[1])$; $O(n)$. One array: $\max(\sum \text{OPT}[\text{children}], w[u] + \sum \text{OPT}[\text{grandchildren}])$. Matching: $\text{out}[u] = \sum \text{OPT}[v]$; $\text{OPT}[u] = \max(\text{out}[u], \max \text{ over children } v \text{ of } w(u,v) + \text{out}[v] + \text{out}[u] - \text{OPT}[v])$; leaf 0; $O(n)$ thanks to the subtraction. Vertex cover: $\text{in} = w + \sum \min(\text{in}, \text{out})$, $\text{out} = \sum \text{in}$.

DAG SHORTEST PATHS. $\text{OPT}[s] = 0$; $\text{OPT}[v] = \min \text{ over in-neighbors } u \text{ of } \text{OPT}[u] + \ell(u,v)$. Topological sort, build G^R , process in order, scan $G^R[v]$. $O(m + n)$. Negative lengths fine. Longest: max, start $-\infty$. Widest: max of $\min(\text{OPT}[u], w(u,v))$, start $d[s] = \infty$, others 0. Counting paths: sum, $\text{OPT}[s] = 1$. Grids and "jump forward" problems are DAGs.

BELLMAN-FORD. Any lengths, no negative cycle. $\text{OPT}[v][j] = \text{shortest walk with } \leq j \text{ edges}$; column 0 = 0 at s , ∞ elsewhere; $\text{OPT}[v][j] = \min(\text{OPT}[v][j-1], \min \text{ over } u \text{ of } \text{OPT}[u][j-1] + \ell(u,v))$; return column $n - 1$ (a shortest walk is a path, $\leq n - 1$ edges). $O(mn)$. Space-saving: $d = [\infty]$, $d[s] = 0$; $n - 1$ rounds of relaxing every edge $d[v] = \min(d[v], d[u] + \ell(u,v))$. Intermediate rounds may run ahead of the table; final answers agree. Extra round still improving $\Rightarrow$ negative cycle.

DIJKSTRA. All lengths ≥ 0 . $d[s] = 0$; repeat n times: pick the unfinished vertex with smallest d , add to S , relax its outgoing edges. Chosen values are final. Array: $O(n^2)$. Heap: n extract-mins and $\leq m$ decrease-keys at $O(\log n) \rightarrow O(m \log n)$. Not Prim: MST $\neq$ shortest-path tree (path 1-2-3-4 of 1s versus edge 1-4 of 2). Negative edge breaks it $((1,2):1, (1,3):3, (3,2):-5)$. Fibonacci heap $O(m + n \log n)$, not examined.

WHICH SHORTEST-PATH ALGORITHM. Equal lengths $\rightarrow$ BFS $O(m + n)$. DAG $\rightarrow$ DAG-DP $O(m + n)$. Lengths $\geq 0 \rightarrow$ Dijkstra. Otherwise, no negative cycle $\rightarrow$ Bellman-Ford $O(mn)$. "At most k edges" $\rightarrow k$ columns of Bellman-Ford, $O(mk)$. "Through x " $\rightarrow$ two Dijkstras. "One free edge" $\rightarrow$ two copies of each vertex. Longest walk with cycles is unbounded; longest path only in DAGs.

RUNNING TIMES. Max, Sum, LCIS, jumps, stock, max subarray, non-adjacent, Fibonacci: $O(n)$. LIS, LPS, palindromic substring: $O(n^2)$. Knapsack, subset sum: $O(nB)$. Edit distance, LCS, common substring: $O(mn)$. MIS-Tree, matching in trees, any tree DP: $O(n)$. DAG-DP, BFS: $O(m + n)$. Bellman-Ford: $O(mn)$. Dijkstra: $O(n^2)$ array, $O(m \log n)$ heap. Floyd-Warshall (if covered): $O(n^3)$.

PRACTICE EXAM ANSWERS. A: 1(a) $[1,2,1,2,3,1,4]$; 1(b) $[3,2,2,2]$; 1(c) $[0,3,-3,8]$; 1(d) 1,3,2,4,5; 1(e) 14 and 13. 2 non-adjacent: $\max(\text{OPT}[i-1], A[i] + \text{OPT}[i-2])$. 3 common substring: diagonal + 1 on a match, return max. 4 downward path: $w[u] + \max(0, \text{best child})$, return max. 5 count paths: sum over in-neighbors in topological order. B: 1(a) $[5,2,11,3,6,7] \rightarrow 6$; 1(b) $[1,1,1,3,3]$; 1(c) $[0,6,10,16,18,22]$; 1(d) $[0,2,3,5,2]$; 1(e) 11. 2 subset sum: or of skip/take. 3 deletions: $m + n - 2 \cdot \text{LCS}$. 4 connected piece: $w[u] + \sum \max(0, \text{OPT}[v])$. 5 through x : Dijkstra from s plus Dijkstra from x .

THE EXAMINATION. Tuesday, November 3, in class; Chapters 4-5; expect 65 minutes, closed-everything, five problems of five points, assigned seats, One Card. Problem 1: five hand-simulations with no partial credit. Problems 2-5: algorithm, correctness, running time; at least two likely verbatim from Problem Sets 4 and 5, which this book does not contain. Write only inside the boxes.

Built October 27, 2026 from the COMP 550 lecture notes for Chapters 4–5 (September 24 – October 22) and Midterm 1 with its solutions (UNC Fall 2026). Personal study material.

© 2026 Deven Jadhav. Not affiliated with or endorsed by UNC-Chapel Hill; course material © its respective owners. Willful large-scale copyright infringement may be a federal crime (17 U.S.C. §506).