Algorithms Illuminated

Part 3: Greedy Algorithms and Dynamic Programming

Tim Roughgarden

Second Printing, 2021

First Edition

Cover image: *Untitled*, by Johanna Dickson

ISBN: 978-0-9992829-4-6 (Paperback)
ISBN: 978-0-9992829-5-3 (ebook)

Library of Congress Control Number: 2017914282

Soundlikeyourself Publishing, LLC
New York, NY
soundlikeyourselfpublishing@gmail.com
www.algorithmsilluminated.org

Contents

Preface

This book is the third in a series based on my online algorithms courses that have been running regularly since 2012, which in turn are based on an undergraduate course that I taught many times at Stanford University. The first two parts of the series are not strict prerequisites for this one, though portions of this book do assume at least a vague recollection of big-O notation (covered in Chapter 2 of *Part 1* or Appendix C of *Part 2*), divide-and-conquer algorithms (Chapter 3 of *Part 1*), and graphs (Chapter 7 of *Part 2*).

What We'll Cover

Algorithms Illuminated, Part 3 provides an introduction to and numerous case studies of two fundamental algorithm design paradigms.

Greedy algorithms and applications. Greedy algorithms solve problems by making a sequence of myopic and irrevocable decisions. For many problems, they are easy to devise and often blazingly fast. Most greedy algorithms are not guaranteed to be correct, but we'll cover several killer applications that are exceptions to this rule. Examples include scheduling problems, optimal compression, and minimum spanning trees of graphs.

Dynamic programming and applications. Few benefits of a serious study of algorithms rival the empowerment that comes from mastering dynamic programming. This design paradigm takes a lot of practice to perfect, but it has countless applications to problems that appear unsolvable using any simpler method. Our dynamic programming boot camp will double as a tour of some of the paradigm's killer applications, including the knapsack problem, the Needleman-Wunsch genome sequence alignment algorithm, Knuth's algorithm for opti-

mal binary search trees, and the Bellman-Ford and Floyd-Warshall shortest-path algorithms.

For a more detailed look into the book's contents, check out the "Upshot" sections that conclude each chapter and highlight the most important points. The "Field Guide to Algorithm Design" on page 201 provides a bird's-eye view of how greedy algorithms and dynamic programming fit into the bigger algorithmic picture.

The starred sections of the book are the most advanced ones. The time-constrained reader can skip these sections on a first reading without any loss of continuity.

Topics covered in the other parts. *Algorithms Illuminated, Part 1* covers asymptotic notation (big-O notation and its close cousins), divide-and-conquer algorithms and the master method, randomized QuickSort and its analysis, and linear-time selection algorithms. *Part 2* covers data structures (heaps, balanced search trees, hash tables, bloom filters), graph primitives (breadth- and depth-first search, connectivity, shortest paths), and their applications (ranging from deduplication to social network analysis). *Part 4* is all about NP-hard problems: how to recognize them in the wild (using reductions); ways to compromise on your algorithmic ambitions for them (through approximation or an exponential worst-case running time); and algorithmic tools to realize those revised ambitions (greedy heuristic algorithms, local search, advanced dynamic programming, MIP and SAT solvers).

Skills You'll Learn From This Book Series

Mastering algorithms takes time and effort. Why bother?

Become a better programmer. You'll learn several blazingly fast subroutines for processing data as well as several useful data structures for organizing data that you can deploy directly in your own programs. Implementing and using these algorithms will stretch and improve your programming skills. You'll also learn general algorithm design paradigms that are relevant to many different problems across different domains, as well as tools for predicting the performance of such algorithms. These "algorithmic design patterns" can help you come up with new algorithms for problems that arise in your own work.

Sharpen your analytical skills. You'll get lots of practice describing and reasoning about algorithms. Through mathematical analysis, you'll gain a deep understanding of the specific algorithms and data structures that these books cover. You'll acquire facility with several mathematical techniques that are broadly useful for analyzing algorithms.

Think algorithmically. After you learn about algorithms, you'll start seeing them everywhere, whether you're riding an elevator, watching a flock of birds, managing your investment portfolio, or even watching an infant learn. Algorithmic thinking is increasingly useful and prevalent in disciplines outside of computer science, including biology, statistics, and economics.

Literacy with computer science's greatest hits. Studying algorithms can feel like watching a highlight reel of many of the greatest hits from the last sixty years of computer science. No longer will you feel excluded at that computer science cocktail party when someone cracks a joke about Dijkstra's algorithm. After reading these books, you'll know exactly what they mean.

Ace your technical interviews. Over the years, countless students have regaled me with stories about how mastering the concepts in these books enabled them to ace every technical interview question they were ever asked.

How These Books Are Different

This series of books has only one goal: *to teach the basics of algorithms in the most accessible way possible.* Think of them as a transcript of what an expert algorithms tutor would say to you over a series of one-on-one lessons.

There are a number of excellent more traditional textbooks about algorithms, any of which usefully complement this book series with additional problems and topics. I encourage you to explore and find your own favorites. There are also several books that, unlike these books, cater to programmers looking for ready-made algorithm implementations in a specific programming language. Many such implementations are freely available on the Web as well.

Who Are You?

The whole point of these books and the online courses upon which they are based is to be as widely and easily accessible as possible. People of all ages, backgrounds, and walks of life are well represented in my online courses, and there are large numbers of students (high-school, college, etc.), software engineers (both current and aspiring), scientists, and professionals hailing from all corners of the world.

This book is not an introduction to programming. Ideally, you've already acquired basic programming skills, such as the use of arrays and recursion, in some standard programming language (be it Java, Python, C, Scala, Haskell, etc.). If you need to beef up your programming skills, there are several outstanding free online courses that teach basic programming.

We also use mathematical analysis as needed to understand how and why algorithms really work. The freely available book *Mathematics for Computer Science*, by Eric Lehman, F. Thomson Leighton, and Albert R. Meyer, is an excellent and entertaining refresher on mathematical notation (like $\sum$ and $\forall$), the basics of proofs (induction, contradiction, etc.), discrete probability, and much more.

Additional Resources

These books are based on online courses that are currently running on the Coursera and edX platforms. I've made several resources available to help you replicate as much of the online course experience as you like.

Videos. If you're more in the mood to watch and listen than to read, check out the YouTube video playlists available from `www.algorithmsilluminated.org`. These videos cover all the topics in this book series, as well as additional advanced topics. I hope they exude a contagious enthusiasm for algorithms that, alas, is impossible to replicate fully on the printed page.

Quizzes. How can you know if you're truly absorbing the concepts in this book? Quizzes with solutions and explanations are scattered throughout the text; when you encounter one, I encourage you to pause and think about the answer before reading on.

End-of-chapter problems. At the end of each chapter you'll find several relatively straightforward questions for testing your understanding, followed by harder and more open-ended challenge problems. Hints or solutions to all of these problems (as indicated by an "*(H)*" or "*(S)*," respectively) are included at the end of the book. Readers can interact with me and each other about the end-of-chapter problems through the book's discussion forum (see below).

Programming problems. Each of the chapters concludes with a suggested programming project whose goal is to help you develop a detailed understanding of an algorithm by creating your own working implementation of it. Data sets, along with test cases and their solutions, can be found at `www.algorithmsilluminated.org`.

Discussion forums. A big reason for the success of online courses is the opportunities they provide for participants to help each other understand the course material and debug programs through discussion forums. Readers of these books have the same opportunity, via the forums available at `www.algorithmsilluminated.org`.

Changes Since the First Printing

The second printing of this book (July 2021) features several minor improvements throughout the text, a handful of new end-of-chapter problems (all with hints or solutions), and a more streamlined version of Chapter 13.

Acknowledgments

These books would not exist without the passion and hunger supplied by the hundreds of thousands of participants in my algorithms courses over the years. I am particularly grateful to those who supplied detailed feedback on an earlier draft of this book: Tonya Blust, Yuan Cao, Carlos Guia, Jim Humelsine, Vladimir Kokshenev, Bayram Kuliyev, and Daniel Zingaro.

I always appreciate suggestions and corrections from readers. These are best communicated through the discussion forums mentioned above.

Tim Roughgarden
New York, NY
April 2019

Chapter 16

Introduction to Dynamic Programming

There's no silver bullet in algorithm design, and the two algorithm design paradigms we've studied so far (divide-and-conquer and greedy algorithms) do not cover all the computational problems you will encounter. The second half of this book will teach you a third design paradigm: the *dynamic programming* paradigm. Dynamic programming is a particularly empowering technique to acquire, as it often leads to efficient solutions beyond the reach of anyone other than serious students of algorithms.

In my experience, most people initially find dynamic programming difficult and counterintuitive. Even more than with other design paradigms, dynamic programming takes practice to perfect. But dynamic programming is relatively formulaic—certainly more so than greedy algorithms—and can be mastered with sufficient practice. This chapter and the next two provide this practice through a half-dozen detailed case studies, including several algorithms belonging to the greatest hits compilation. You'll learn how these famous algorithms work, but even better, you'll add to your programmer toolbox a general and flexible algorithm design technique that you can apply to problems that come up in your own projects. Through these case studies, the power and flexibility of dynamic programming will become clear—it's a technique you simply have to know.

Pep Talk

It is totally normal to feel confused the first time you see dynamic programming. Confusion should not discourage you. It does not represent an intellectual failure on your part, only an opportunity to get even smarter.

16.1 The Weighted Independent Set Problem

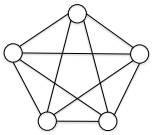
I'm not going to tell you what dynamic programming is just yet. Instead, we'll devise from scratch an algorithm for a tricky and concrete computational problem, which will force us to develop a number of new ideas. After we've solved the problem, we'll zoom out and identify the ingredients of our solution that exemplify the general principles of dynamic programming. Then, armed with a template for developing dynamic programming algorithms and an example instantiation, we'll tackle increasingly challenging applications of the paradigm.

16.1.1 Problem Definition

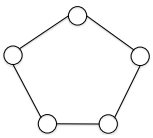
To describe the problem, let $G = (V, E)$ be an undirected graph. An *independent set* of G is a subset $S \subseteq V$ of mutually non-adjacent vertices: for every $v, w \in S$, $(v, w) \notin E$. Equivalently, an independent set does not contain both endpoints of any edge of G. For example, if vertices represent people and edges pairs of people who dislike each other, the independent sets correspond to groups of people who all get along. Or, if the vertices represent classes you're thinking about taking and there is an edge between each pair of conflicting classes, the independent sets correspond to feasible course schedules (assuming you can't be in two places at once).

Quiz 16.1

How many different independent sets does a complete graph with 5 vertices have?



How about a cycle with 5 vertices?



a) 1 and 2 (respectively)

b) 5 and 10

c) 6 and 11

d) 6 and 16

(See Section 16.1.4 for the solution and discussion.)

We can now state the *weighted independent set (WIS)* problem:

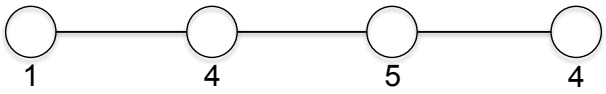
Problem: Weighted Independent Set (WIS)

Input: An undirected graph $G = (V, E)$ and a nonnegative weight w_v for each vertex $v \in V$.

Output: An independent set $S \subseteq V$ of G with the maximum-possible sum $\sum_{v \in S} w_v$ of vertex weights.

An optimal solution to the WIS problem is called a *maximum-weight independent set (MWIS)*. For example, if vertices represent courses, vertex weights represent units, and edges represent conflicts between courses, the MWIS corresponds to the feasible course schedule with the heaviest load (in units).

The WIS problem is challenging even in the super-simple case of *path graphs*. For example, an input to the problem might look like this (with vertices labeled by their weights):



This graph has 8 independent sets: the empty set, the four singleton sets, the first and third vertices, the first and fourth vertices, and the second and fourth vertices. The last of these has the largest total weight of 8. The number of independent sets of a path graph grows exponentially with the number of vertices (do you see why?), so there is no hope of solving the problem via exhaustive search, except in the tiniest of instances.

16.1.2 The Natural Greedy Algorithm Fails

For many computational problems, greedy algorithms are a great place to start brainstorming. Such algorithms are usually easy to come up with, and even when one fails to solve the problem (as is often the case), the manner in which it fails can help you better understand the intricacies of the problem.

For the WIS problem, perhaps the most natural greedy algorithm is an analog of Kruskal's algorithm: Perform a single pass over the vertices, from best (highest weight) to worst (lowest weight), adding a vertex to the solution-so-far as long as it doesn't conflict with a previously chosen vertex. Given an input graph $G = (V, E)$ with vertex weights, the pseudocode is:

WIS: A Greedy Approach

```
S := ∅
sort vertices of V by weight
for each v ∈ V, in nonincreasing order of weight do
    if S ∪ {v} is an independent set of G then
        S := S ∪ {v}
return S
```

Simple enough. But does it work?

Quiz 16.2

What is the total weight of the output of the greedy algorithm when the input graph is the four-vertex path on page 104? Is this the maximum possible?

a) 6; no

b) 6; yes

c) 8; no

d) 8; yes

(See Section 16.1.4 for the solution and discussion.)

Chapters 13–15 spoiled us with a plethora of cherry-picked correct greedy algorithms, but don't forget the warning back on page 3: Greedy algorithms are usually *not* correct.

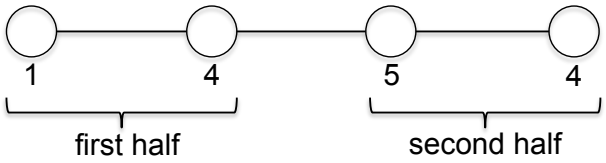
16.1.3 A Divide-and-Conquer Approach?

The divide-and-conquer algorithm design paradigm (Section 13.1.1) is always worth a shot for problems in which there's a sensible way to break the input into smaller subproblems. For the WIS problem with an input path graph $G = (V, E)$, the natural high-level approach is (ignoring the base case):

WIS: A Divide-and-Conquer Approach

G_1 := first half of G
G_2 := second half of G
S_1 := recursively solve the WIS problem on G_1
S_2 := recursively solve the WIS problem on G_2
combine S_1, S_2 into a solution S for G
return S

The devil is in the details of the combine step. Returning to our running example:



the first and second recursive calls return the second and third vertices as the optimal solutions to their respective subproblems. The union of their solutions is *not* an independent set due to the conflict at the boundary between the two solutions. It's easy to see how to defuse such a border conflict when the input graph has only four vertices; when it has hundreds or thousands of vertices, not so much.[1]

Can we do better than a greedy or divide-and-conquer algorithm?

[1] The problem can be solved in $O(n^2)$ time by a divide-and-conquer algorithm that makes *four* recursive calls rather than two, where n is the number of vertices. (Do you see how to do this?) Our dynamic programming algorithm for the problem will run in $O(n)$ time.

16.1.4 Solutions to Quizzes 16.1–16.2

Solution to Quiz 16.1

Correct answer: (c). The complete graph has no non-adjacent vertices, so every independent set has at most one vertex. Thus, there are six independent sets: the empty set and the five singleton sets. The cycle has the same six independent sets that the complete graph does, plus some independent sets of size 2. (Every subset of three or more vertices has a pair of adjacent vertices.) It has five size-2 independent sets (as you should verify), for a total of eleven.

Solution to Quiz 16.2

Correct answer: (a). The first iteration of the greedy algorithm commits to the maximum-weight vertex, which is the third vertex (with weight 5). This eliminates the adjacent vertices (the second and fourth ones, both with weight 4) from further consideration. The algorithm is then stuck selecting the first vertex and it outputs an independent set with total weight 6. This is not optimal, as the second and fourth vertices constitute an independent set with total weight 8.

16.2 A Linear-Time Algorithm for WIS in Paths

16.2.1 Optimal Substructure and Recurrence

To quickly solve the WIS problem in path graphs, we'll need to up our game. Key to our approach is the following thought experiment: Suppose someone handed us an optimal solution on a silver platter. What must it look like? Ideally, this thought experiment would show that an optimal solution must be constructed in a prescribed way from optimal solutions to smaller subproblems, thereby narrowing down the field of candidates to a manageable number.[2]

More concretely, let $G = (V, E)$ denote the n-vertex path graph with edges (v_1, v_2), (v_2, v_3), $\ldots$, (v_{n-2}, v_{n-1}), (v_{n-1}, v_n), and suppose that each vertex $v_i \in V$ has a nonnegative weight w_i. Assume that $n \geq 2$; otherwise, the answer is obvious. Suppose we magically knew

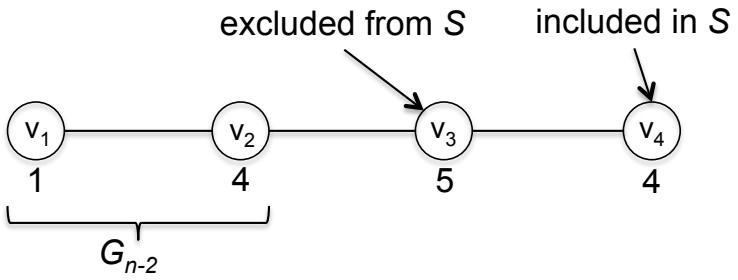
[2]There's no circularity in performing a thought experiment about the very object we're trying to compute. As we'll see, such thought experiments can light up a trail that leads directly to an efficient algorithm.

an MWIS $S \subseteq V$ with total weight W. What can we say about it? Here's a tautology: S either contains the final vertex v_n, or it doesn't. Let's examine these cases in reverse order.

Case 1: $v_n \notin S$**.** Suppose the optimal solution S happens to exclude v_n. Obtain the $(n-1)$-vertex path graph G_{n-1} from G by plucking off the last vertex v_n and the last edge (v_{n-1}, v_n). Because S does not include the last vertex of G, it contains only vertices of G_{n-1} and can be regarded as an independent set of G_{n-1} (still with total weight W). And S is not just any old independent set of G_{n-1}—it's a *maximum-weight* such set. For if S^* were an independent set of G_{n-1} with total weight $W^* > W$, then S^* would also constitute an independent set of total weight W^* in the larger graph G. This would contradict the supposed optimality of S.

In other words, once you know that an MWIS excludes the last vertex, you know exactly what it looks like: It's an MWIS of the smaller graph G_{n-1}.

Case 2: $v_n \in S$**.** Suppose S includes the last vertex v_n. As an independent set, S cannot include two consecutive vertices from the path, so it excludes the penultimate vertex: $v_{n-1} \notin S$. Obtain the $(n-2)$-vertex path graph G_{n-2} from G by plucking off the last two vertices and edges:[3]



Because S contains v_n and G_{n-2} does not, we can't regard S as an independent set of G_{n-2}. But after removing the last vertex from S, we can: $S - \{v_n\}$ contains neither v_{n-1} nor v_n and hence can be regarded as an independent set of the smaller graph G_{n-2} (with total weight $W - w_n$). Moreover, $S - \{v_n\}$ must be an MWIS

[3]When $n = 2$, we interpret G_0 as the empty graph (with no vertices or edges). The only independent set of G_0 is the empty set, which has total weight 0.

of G_{n-2}. For suppose S^* were an independent set of G_{n-2} with total weight $W^* > W - w_n$. Because G_{n-2} (and hence S^*) excludes the penultimate vertex v_{n-1}, blithely adding the last vertex v_n to S^* would not create any conflicts, and so $S^* \cup \{v_n\}$ would be an independent set of G with total weight $W^* + w_n > (W - w_n) + w_n = W$. This would contradict the supposed optimality of S.

In other words, once you know that an MWIS includes the last vertex, you know exactly what it looks like: It's an MWIS of the smaller graph G_{n-2}, supplemented with the final vertex v_n. Summarizing, two and only two candidates are vying to be an MWIS:

Lemma 16.1 (WIS Optimal Substructure) *Let S be an MWIS of a path graph G with $n \geq 2$ vertices. Let G_i denote the subgraph of G comprising its first i vertices and $i-1$ edges. Then, S is either:*

(i) an MWIS of G_{n-1}; or

(ii) an MWIS of G_{n-2}, supplemented with G's final vertex v_n.

Lemma 16.1 singles out the only two possibilities for an MWIS, so whichever option has larger total weight is an optimal solution. We therefore have a recursive formula—a *recurrence*—for the total weight of an MWIS:

Corollary 16.2 (WIS Recurrence) *With the assumptions and notation of Lemma 16.1, let W_i denote the total weight of an MWIS of G_i. (When $i = 0$, interpret W_i as 0.) Then*

$$W_n = \max\{\underbrace{W_{n-1}}_{\textit{Case 1}}, \underbrace{W_{n-2} + w_n}_{\textit{Case 2}}\}.$$

More generally, for every $i = 2, 3, \ldots, n$,

$$W_i = \max\{W_{i-1}, W_{i-2} + w_i\}.$$

The more general statement in Corollary 16.2 follows by invoking the first statement, for each $i = 2, 3, \ldots, n$, with G_i playing the role of the input graph G.

16.2.2 A Naive Recursive Approach

Lemma 16.1 is good news—we've narrowed down the field to just two candidates for the optimal solution! So, why not try both options and return the better of the two? This leads to the following pseudocode, in which the graphs G_{n-1} and G_{n-2} are defined as before:

A Recursive Algorithm for WIS

Input: a path graph G with vertex set $\{v_1, v_2, \ldots, v_n\}$ and a nonnegative weight w_i for each vertex v_i.
Output: a maximum-weight independent set of G.

```
if n = 0 then                                   // base case #1
    return the empty set
if n = 1 then                                   // base case #2
    return {v_1}
// recursion when n ≥ 2
S_1 := recursively compute an MWIS of G_{n-1}
S_2 := recursively compute an MWIS of G_{n-2}
return S_1 or S_2 ∪ {v_n}, whichever has higher weight
```

A straightforward proof by induction shows that this algorithm is guaranteed to compute a maximum-weight independent set.[4] What about the running time?

Quiz 16.3

What is the asymptotic running time of the recursive WIS algorithm, as a function of the number n of vertices? (Choose the strongest correct statement.)

a) $O(n)$

b) $O(n \log n)$

[4]The proof proceeds by induction on the number n of vertices. The base cases ($n = 0, 1$) are clearly correct. For the inductive step ($n \geq 2$), the inductive hypothesis guarantees that S_1 and S_2 are indeed MWISs of G_{n-1} and G_{n-2}, respectively. Lemma 16.1 implies that the better of S_1 and $S_2 \cup \{v_n\}$ is an MWIS of G, and this is the output of the algorithm.

c) $O(n^2)$

d) none of the above

(See Section 16.2.5 for the solution and discussion.)

16.2.3 Recursion with a Cache

Quiz 16.3 shows that our recursive WIS algorithm is no better than exhaustive search. The next quiz contains the key to unlocking a radical running time improvement. Think about it carefully before reading the solution.

Quiz 16.4

Each of the (exponentially many) recursive calls of the recursive WIS algorithm is responsible for computing an MWIS of a specified input graph. Ranging over all of the calls, how many *distinct* input graphs are ever considered?

a) $\Theta(1)$[5]

b) $\Theta(n)$

c) $\Theta(n^2)$

d) $2^{\Theta(n)}$

(See Section 16.2.5 for the solution and discussion.)

Quiz 16.4 implies that the exponential running time of our recursive WIS algorithm stems solely from its absurd redundancy, solving the same subproblems from scratch over, and over, and over, and over again. Here's an idea: The first time we solve a subproblem, why not save the result in a cache once and for all? Then, if we encounter the

[5] If big-O notation is analogous to "less than or equal to," then big-theta notation is analogous to "equal to." Formally, a function $f(n)$ is $\Theta(g(n))$ if there are constants c_1 and c_2 such that $f(n)$ is wedged between $c_1 \cdot g(n)$ and $c_2 \cdot g(n)$ for all sufficiently large n.

same subproblem later, we can look up its solution in the cache in constant time.[6]

Blending caching into the pseudocode on page 110 is easy. The results of past computations are stored in a globally visible length-$(n+1)$ array A, with $A[i]$ storing an MWIS of G_i, where G_i comprises the first i vertices and the first $i-1$ edges of the original input graph (and G_0 is the empty graph). In line 6, the algorithm now first checks whether the array A already contains the relevant solution S_1; if not, it computes S_1 recursively as before and caches the result in A. Similarly, the new version of line 7 either looks up or recursively computes and caches S_2, as needed.

Each of the $n+1$ subproblems is now solved from scratch only once. Caching surely speeds up the algorithm, but by how much? Properly implemented, the running time drops from exponential to *linear*. This dramatic speedup will be easier to see after we reformulate our top-down recursive algorithm as a bottom-up iterative one—and the latter is usually what you want to implement in practice, anyway.

16.2.4 An Iterative Bottom-Up Implementation

As part of figuring out how to incorporate caching into our recursive WIS algorithm, we realized that there are exactly $n+1$ relevant subproblems, corresponding to all possible prefixes of the input graph (Quiz 16.4).

WIS in Path Graphs: Subproblems

Compute W_i, the total weight of an MWIS of the prefix graph G_i.

(For each $i = 0, 1, 2, \ldots, n$.)

For now, we focus on computing the total weight of an MWIS for a subproblem. Section 16.3 shows how to also identify the vertices of an MWIS.

Now that we know which subproblems are the important ones, why not cut to the chase and systematically solve them one by one?

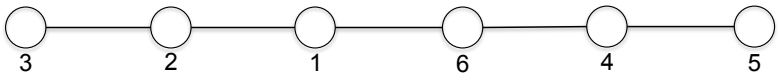
[6] This technique of caching the result of a computation to avoid redoing it later is sometimes called *memoization*.

The solution to a subproblem depends on the solutions to two smaller subproblems. To ensure that these two solutions are readily available, it makes sense to work bottom-up, starting with the base cases and building up to ever-larger subproblems.

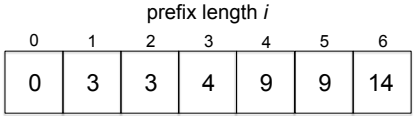
```
                          WIS

Input: a path graph G with vertex set {v_1, v_2, ..., v_n}
  and a nonnegative weight w_i for each vertex v_i.
Output: the total weight of a maximum-weight
  independent set of G.

-------------------------------------------------------------
A := length-(n + 1) array   // subproblem solutions
A[0] := 0                                  // base case #1
A[1] := w_1                                // base case #2
for i := 2 to n do
    // use recurrence from Corollary 16.2
    A[i] := max{A[i - 1], A[i - 2] + w_i}
                 Case 1        Case 2
return A[n] // solution to largest subproblem
```

The length-$(n+1)$ array A is indexed from 0 to n. By the time an iteration of the main loop must compute the subproblem solution $A[i]$, the values $A[i-1]$ and $A[i-2]$ of the two relevant smaller subproblems have already been computed in previous iterations (or in the base cases). Thus, each loop iteration takes $O(1)$ time, for a blazingly fast running time of $O(n)$.

For example, for the input graph



you should check that the final array values are:



At the conclusion of the `WIS` algorithm, each array entry $A[i]$ stores the total weight of an MWIS of the graph G_i that comprises

the first i vertices and $i-1$ edges of the input graph. This follows from an inductive argument similar to the one in footnote 4. The base cases $A[0]$ and $A[1]$ are clearly correct. When computing $A[i]$ with $i \geq 2$, by induction, the values $A[i-1]$ and $A[i-2]$ are indeed the total weights of MWISs of G_{i-1} and G_{i-2}, respectively. Corollary 16.2 then implies that $A[i]$ is computed correctly, as well. In the example above, the total weight of an MWIS in the original input graph is the value in the final array entry (14), corresponding to the independent set consisting of the first, fourth, and sixth vertices.

Theorem 16.3 (Properties of `WIS`) *For every path graph and non-negative vertex weights, the `WIS` algorithm runs in linear time and returns the total weight of a maximum-weight independent set.*

16.2.5 Solutions to Quizzes 16.3–16.4

Solution to Quiz 16.3

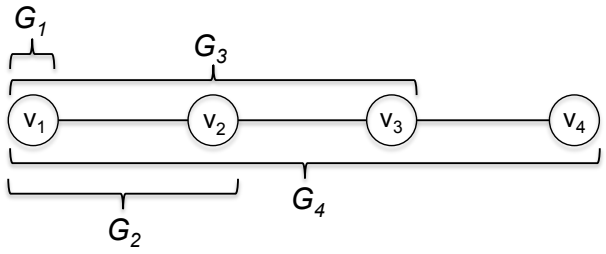
Correct answer: (d). Superficially, the recursion pattern looks similar to that of $O(n \log n)$-time divide-and-conquer algorithms like MergeSort, with two recursive calls followed by an easy combine step. But there's a big difference: The MergeSort algorithm throws away half the input before recursing, while our recursive WIS algorithm throws away only one or two vertices (perhaps out of thousands or millions). Both algorithms have recursion trees with branching factor 2.[7] The former has roughly $\log_2 n$ levels and, hence, only a linear number of leaves. The latter has no leaves until levels $n/2$ and later, which implies that it has at least $2^{n/2}$ leaves. (See Problem 16.4 for a sharper lower bound.) We conclude that the running time of the recursive algorithm grows exponentially with n.

Solution to Quiz 16.4

Correct answer: (b). How does the input graph change upon passage to a recursive call? Either one or two vertices and edges are

[7] Every recursive algorithm can be associated with a recursion tree, in which the nodes of the tree correspond to all the algorithm's recursive calls. The root of the tree corresponds to the initial call to the algorithm (with the original input), with one child at the next level for each of its recursive calls. The leaves at the bottom of the tree correspond to the recursive calls that trigger a base case and make no further recursive calls.

plucked off the end of the graph. Thus, an invariant throughout the recursion is that every recursive call is given some *prefix* G_i as its input graph, where G_i denotes the first i vertices and $i-1$ edges of the original input graph (and G_0 denotes the empty graph):



There are only $n+1$ such graphs $(G_0, G_1, G_2, \ldots, G_n)$, where n is the number of vertices in the input graph. Therefore, only $n+1$ distinct subproblems ever get solved across the exponential number of different recursive calls.

16.3 A Reconstruction Algorithm

The `WIS` algorithm in Section 16.2.4 computes only the *weight* possessed by an MWIS of a path graph, not an MWIS itself. A simple hack is to modify the `WIS` algorithm so that each array entry $A[i]$ records both the total weight of an MWIS of the ith subproblem G_i *and* the vertices of an MWIS of G_i that realizes this value.

A better approach, which saves both time and space, is to use a postprocessing step to reconstruct an MWIS from the tracks in the mud left by the `WIS` algorithm in its subproblem array A. For starters, how do we know whether the last vertex v_n of the input graph G belongs to an MWIS? The key is again Lemma 16.1, which states that two and only two candidates are vying to be an MWIS of G: an MWIS of the graph G_{n-1}, and an MWIS of the graph G_{n-2}, supplemented with v_n. Which one is it? The one with larger total weight. How do we know which one that is? Just look at the clues left in the array A! The final values of $A[n-1]$ and $A[n-2]$ record the total weights of MWISs of G_{n-1} and G_{n-2}, respectively. So:

1. If $A[n-1] \geq A[n-2] + w_n$, an MWIS of G_{n-1} is also an MWIS of G_n.

2. If $A[n-2] + w_n \geq A[n-1]$, supplementing an MWIS of G_{n-2} with v_n yields an MWIS of G_n.

In the first case, we know to exclude v_n from our solution and can continue the reconstruction process from v_{n-1}. In the second case, we know to include v_n in our solution, which forces us to exclude v_{n-1}. The reconstruction process then resumes from v_{n-2}.[8]

`WIS Reconstruction`

Input: the array A computed by the `WIS` algorithm for a path graph G with vertex set $\{v_1, v_2, \ldots, v_n\}$ and a nonnegative weight w_i for each vertex v_i.
Output: a maximum-weight independent set of G.

```
S := ∅                                  // vertices in an MWIS
i := n
while i ≥ 2 do
    if A[i − 1] ≥ A[i − 2] + w_i then   // Case 1 wins
        i := i − 1                      // exclude v_i
    else                                // Case 2 wins
        S := S ∪ {v_i}                  // include v_i
        i := i − 2                      // exclude v_{i−1}
if i = 1 then                           // base case #2
    S := S ∪ {v_1}
return S
```

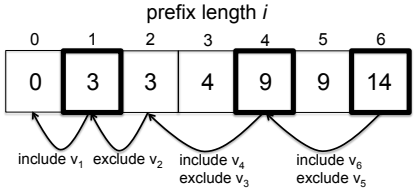
`WIS Reconstruction` does a single backward pass over the array A and spends $O(1)$ time per loop iteration, so it runs in $O(n)$ time. The inductive proof of correctness is similar to that for the `WIS` algorithm (Theorem 16.3).[9]

For example, for the input graph

[8] If there is a tie ($A[n-2] + w_n = A[n-1]$), both options lead to an optimal solution.

[9] The keen reader might complain that it's wasteful to recompute comparisons of the form $A[i-1]$ vs. $A[i-2] + w_i$, which have already been made by the `WIS` algorithm. If that algorithm is modified to cache the comparison results (in effect, remembering which case of the recurrence was used to fill in each array entry), these results can be looked up rather than recomputed in the `WIS Reconstruction` algorithm. This idea will be particularly important for some of the harder problems studied in Chapters 17 and 18.

the `WIS Reconstruction` algorithm includes v_6 (forcing v_5's exclusion), includes v_4 (forcing v_3's exclusion), excludes v_2, and includes v_1:



16.4 The Principles of Dynamic Programming

16.4.1 A Three-Step Recipe

Guess what? With `WIS`, we just designed our first dynamic programming algorithm! The general dynamic programming paradigm can be summarized by a three-step recipe. It is best understood through examples; we have only one so far, so I encourage you to revisit this section after we finish a few more case studies.

The Dynamic Programming Paradigm

1. Identify a relatively small collection of subproblems.
2. Show how to quickly and correctly solve "larger" subproblems given the solutions to "smaller" ones.
3. Show how to quickly and correctly infer the final solution from the solutions to all of the subproblems.

After these three steps are implemented, the corresponding dynamic programming algorithm writes itself: Systematically solve all the subproblems one by one, working from "smallest" to "largest," and extract the final solution from those of the subproblems.

In our solution to the WIS problem in n-vertex path graphs, we implemented the first step by identifying a collection of $n + 1$ subproblems. For $i = 0, 1, 2, \ldots, n$, the ith subproblem is to compute the total weight of an MWIS of the graph G_i consisting of the first i

vertices and $i-1$ edges of the input graph (where G_0 denotes the empty graph). There is an obvious way to order the subproblems from "smallest" to "largest," namely $G_0, G_1, G_2, \ldots, G_n$. The recurrence in Corollary 16.2 is a formula that implements the second step by showing how to compute the solution to the ith subproblem in $O(1)$ time from the solutions to the $(i-2)$th and $(i-1)$th subproblems. The third step is easy: Return the solution to the largest subproblem, which is the same as the original problem.

16.4.2 Desirable Subproblem Properties

The key that unleashes the potential of dynamic programming for solving a problem is the identification of the right collection of subproblems. What properties do we want them to satisfy? Assuming we perform at least a constant amount of work solving each subproblem, the number of subproblems is a lower bound on the running time of our algorithm. Thus, we'd like the number of subproblems to be as low as possible—our WIS solution used only a linear number of subproblems, which is usually the best-case scenario. Similarly, the time required to solve a subproblem (given solutions to smaller subproblems) and to infer the final solution will factor into the algorithm's overall running time.

For example, suppose an algorithm solves at most $f(n)$ different subproblems (working systematically from "smallest" to "largest"), using at most $g(n)$ time for each, and performs at most $h(n)$ postprocessing work to extract the final solution (where n denotes the input size). The algorithm's running time is then at most

$$\underbrace{f(n)}_{\#\text{ subproblems}} \times \underbrace{g(n)}_{\substack{\text{time per subproblem} \\ \text{(given previous solutions)}}} + \underbrace{h(n)}_{\text{postprocessing}}. \tag{16.1}$$

The three steps of the recipe call for keeping $f(n)$, $g(n)$, and $h(n)$, respectively, as small as possible. In the basic `WIS` algorithm, without the `WIS Reconstruction` postprocessing step, we have $f(n) = O(n)$, $g(n) = O(1)$, and $h(n) = O(1)$, for an overall running time of $O(n)$. If we include the reconstruction step, the $h(n)$ term jumps to $O(n)$, but the overall running time $O(n) \times O(1) + O(n) = O(n)$ remains linear.

16.4.3 A Repeatable Thought Process

When devising your own dynamic programming algorithms, the heart of the matter is figuring out the magical collection of subproblems. After that, everything else falls into place in a fairly formulaic way. But how would you ever come up with them? If you have a black belt in dynamic programming, you might be able to just stare at a problem and intuitively know what the subproblems should be. White belts, however, still have a lot of training to do. In our case studies, rather than plucking subproblems from the sky, we'll carry out a thought process that naturally leads to a collection of subproblems (as we did for the WIS problem). This process is repeatable and you can mimic it when you apply the dynamic programming paradigm to problems that arise in your own projects.

The main idea is to reason about the structure of an optimal solution, identifying the different ways it might be constructed from optimal solutions to smaller subproblems. This thought experiment can lead to both the identification of the relevant subproblems and a recurrence (analogous to Corollary 16.2) that expresses the solution of a subproblem as a function of the solutions of smaller subproblems. A dynamic programming algorithm can then fill in an array with subproblem solutions, proceeding from smaller to larger subproblems and using the recurrence to compute each array entry.

16.4.4 Dynamic Programming vs. Divide-and-Conquer

Readers familiar with the divide-and-conquer algorithm design paradigm (Section 13.1.1) might recognize some similarities to dynamic programming, especially the latter's top-down recursive formulation (Sections 16.2.2–16.2.3). Both paradigms recursively solve smaller subproblems and combine the results into a solution to the original problem. Here are six differences between typical uses of the two paradigms:

1. Each recursive call of a typical divide-and-conquer algorithm commits to a single way of dividing the input into smaller subproblems.[10] Each recursive call of a dynamic programming

[10]For example, in the MergeSort algorithm, every recursive call divides its input array into its left and right halves. The QuickSort algorithm invokes a

algorithm keeps its options open, considering multiple ways of defining smaller subproblems and choosing the best of them.[11]

2. Because each recursive call of a dynamic programming algorithm tries out multiple choices of smaller subproblems, subproblems generally recur across different recursive calls; caching subproblem solutions is then a no-brainer optimization. In most divide-and-conquer algorithms, all the subproblems are distinct and there's no point in caching their solutions.[12]

3. Most of the canonical applications of the divide-and-conquer paradigm replace a straightforward polynomial-time algorithm for a task with a faster divide-and-conquer version.[13] The killer applications of dynamic programming are polynomial-time algorithms for optimization problems for which straightforward solutions (like exhaustive search) require an exponential amount of time.

4. In a divide-and-conquer algorithm, subproblems are chosen primarily to optimize the running time; correctness often takes care of itself.[14] In dynamic programming, subproblems are usually chosen with correctness in mind, come what may with the running time.[15]

5. Relatedly, a divide-and-conquer algorithm generally recurses on subproblems with size at most a constant fraction (like 50%) of the input. Dynamic programming has no qualms about

partitioning subroutine to choose how to split the input array in two, and then commits to this division for the remainder of its execution.

[11] For example, in the `WIS` algorithm, each recursive call chooses between a subproblem with one fewer vertex and one with two fewer vertices.

[12] For example, in the MergeSort and QuickSort algorithms, every subproblem corresponds to a different subarray of the input array.

[13] For example, the MergeSort algorithm brings the running time of sorting a length-n array down from the straightforward bound of $O(n^2)$ to $O(n \log n)$. Other examples include Karatsuba's algorithm (which improves the running time of multiplying two n-digit numbers from $O(n^2)$ to $O(n^{1.59})$) and Strassen's algorithm (for multiplying two $n \times n$ matrices in $O(n^{2.81})$ rather than $O(n^3)$ time).

[14] For example, the QuickSort algorithm always correctly sorts the input array, no matter how good or bad its chosen pivot elements are.

[15] Our dynamic programming algorithm for the knapsack problem in Section 16.5 is a good example.

recursing on subproblems that are barely smaller than the input (like in the `WIS` algorithm), if necessary for correctness.

6. The divide-and-conquer paradigm can be viewed as a special case of dynamic programming, in which each recursive call chooses a fixed collection of subproblems to solve recursively. As the more sophisticated paradigm, dynamic programming applies to a wider range of problems than divide-and-conquer, but it is also more technically demanding to apply (at least until you've had sufficient practice).

Confronted with a new problem, which paradigm should you use? If you see a divide-and-conquer solution, by all means use it. If all your divide-and-conquer attempts fail—and especially if they fail because the combine step always seems to require redoing a lot of computation from scratch—it's time to try dynamic programming.

16.4.5 Why "Dynamic Programming"?

You might be wondering where the weird moniker "dynamic programming" came from; the answer is no clearer now that we know how the paradigm works than it was before.

The first point of confusion is the anachronistic use of the word "programming." In modern times it refers to coding, but back in the 1950s "programming" usually meant "planning." (For example, it has this meaning in the phrase "television programming.") What about "dynamic"? For the full story, I refer you to the father of dynamic programming himself, Richard E. Bellman, writing about his time working at the RAND Corporation:

> The 1950's were not good years for mathematical research. We had a very interesting gentleman in Washington named Wilson. He was Secretary of Defense, and he actually had a pathological fear and hatred of the word, research. I'm not using the term lightly; I'm using it precisely. His face with suffuse, he would turn red, and he would get violent if people used the term, research, in his presence. You can imagine how he felt, then, about the term, mathematical. The RAND Corporation was employed by the Air Force, and the Air Force had Wilson as its boss, essentially.

> Hence, I felt I had to do something to shield Wilson and the Air Force from the fact that I was really doing mathematics inside the RAND Corporation. What title, what name, could I choose? In the first place I was interested in planning, in decision making, in thinking. But planning, is not a good word for various reasons. I decided therefore to use the word, "programming." ... ["Dynamic"] has a very interesting property as an adjective, and that is it's impossible to use the word, dynamic, in the pejorative sense. Try thinking of some combination that will possibly give it a pejorative meaning. It's impossible. Thus, I thought dynamic programming was a good name. It was something not even a Congressman could object to. So I used it as an umbrella for my activities.[16]

16.5 The Knapsack Problem

Our second case study concerns the well-known *knapsack problem*. Following the same thought process we used to develop the `WIS` algorithm in Section 16.2, we'll arrive at the famous dynamic programming solution to the problem.

16.5.1 Problem Definition

An instance of the knapsack problem is specified by $2n + 1$ positive integers, where n is the number of "items" (which are labeled arbitrarily from 1 to n): a value v_i and a size s_i for each item i, and a knapsack capacity C.[17] The responsibility of an algorithm is to select a subset of the items. The total value of the items should be as large as possible while still fitting in the knapsack, meaning their total size should be at most C.

[16] Richard E. Bellman, *Eye of the Hurricane: An Autobiography*, World Scientific, 1984, page 159.

[17] It's actually not important that the item values are integers (as opposed to arbitrary positive real numbers). It *is* important that the item sizes are integers, as we'll see in due time.

Problem: Knapsack

Input: Item values $v_1, v_2, \ldots, v_n$, item sizes $s_1, s_2, \ldots, s_n$, and a knapsack capacity C. (All positive integers.)

Output: A subset $S \subseteq \{1, 2, \ldots, n\}$ of items with the maximum-possible sum $\sum_{i \in S} v_i$ of values, subject to having total size $\sum_{i \in S} s_i$ at most C.

Quiz 16.5

Consider an instance of the knapsack problem with knapsack capacity $C = 6$ and four items:

Item	Value	Size
1	3	4
2	2	3
3	4	2
4	4	3

What is the total value of an optimal solution?

a) 6

b) 7

c) 8

d) 10

(See Section 16.5.7 for the solution and discussion.)

I could tell you a cheesy story about a knapsack-wielding burglar who breaks into a house and wants to make off quickly with the best pile of loot possible, but this would do a disservice to the problem, which is actually quite fundamental. Whenever you have a scarce resource that you want to use in the smartest way possible, you're talking about a knapsack problem. On which goods and services should you spend your paycheck to get the most value? Given an operating budget and a set of job candidates with differing productivities and requested salaries, whom should you hire? These are examples of knapsack problems.

16.5.2 Optimal Substructure and Recurrence

To apply the dynamic programming paradigm to the knapsack problem, we must figure out the right collection of subproblems. As with the WIS problem, we'll arrive at them by reasoning about the structure of optimal solutions and identifying the different ways they can be constructed from optimal solutions to smaller subproblems. Another deliverable of this exercise will be a recurrence for quickly computing the solution to a subproblem from those of two smaller subproblems.

Consider an instance of the knapsack problem with item values $v_1, v_2, \ldots, v_n$, item sizes $s_1, s_2, \ldots, s_n$, and knapsack capacity C, and suppose someone handed us on a silver platter an optimal solution $S \subseteq \{1, 2, \ldots, n\}$ with total value $V = \sum_{i \in S} v_i$. What must it look like? As with the WIS problem, we start with a tautology: S either contains the last item (item n) or it doesn't.[18]

Case 1: $n \notin S$**.** Because the optimal solution S excludes the last item, it can be regarded as a feasible solution (still with total value V and total size at most C) to the smaller problem consisting of only the first $n-1$ items (and knapsack capacity C). Moreover, S must be an optimal solution to the smaller subproblem: If there were a solution $S^* \subseteq \{1, 2, \ldots, n-1\}$ with total size at most C and total value greater than V, it would also constitute such a solution in the original instance. This would contradict the supposed optimality of S.

Case 2: $n \in S$**.** The trickier case is when the optimal solution S makes use of the last item n. This case can occur only when $s_n \leq C$. We can't regard S as a feasible solution to a smaller problem with only the first $n-1$ items, but we can after removing item n. Is $S - \{n\}$ an optimal solution to a smaller subproblem?

[18] The WIS problem in path graphs is inherently sequential, with the vertices ordered along the path. This naturally led to subproblems that correspond to prefixes of the input. The items in the knapsack problem are not inherently ordered, but to identify the right collection of subproblems, it's helpful to mimic our previous approach and *pretend* they're ordered in some arbitrary way. A "prefix" of the items then corresponds to the first i items in our arbitrary ordering (for some $i \in \{0, 1, 2, \ldots, n\}$). Many other dynamic programming algorithms use this same trick.

Quiz 16.6

Which of the following statements hold for the set $S - \{n\}$? (Choose all that apply.)

a) It is an optimal solution to the subproblem consisting of the first $n-1$ items and knapsack capacity C.

b) It is an optimal solution to the subproblem consisting of the first $n-1$ items and knapsack capacity $C - v_n$.

c) It is an optimal solution to the subproblem consisting of the first $n-1$ items and knapsack capacity $C - s_n$.

d) It might not be feasible if the knapsack capacity is only $C - s_n$.

(See Section 16.5.7 for the solution and discussion.)

This case analysis shows that two and only two candidates are vying to be an optimal knapsack solution:

Lemma 16.4 (Knapsack Optimal Substructure) *Let S be an optimal solution to a knapsack problem with $n \geq 1$ items, item values $v_1, v_2, \ldots, v_n$, item sizes $s_1, s_2, \ldots, s_n$, and knapsack capacity C. Then, S is either:*

(i) an optimal solution for the first $n-1$ items with knapsack capacity C; or

(ii) an optimal solution for the first $n-1$ items with knapsack capacity $C - s_n$, supplemented with the last item n.

The solution in (i) is always an option for the optimal solution. The solution in (ii) is an option if and only if $s_n \leq C$; in this case, s_n units of capacity are effectively reserved in advance for item n.[19] The option with the larger total value is an optimal solution, leading to the following recurrence:

[19]This is analogous to, for the WIS problem in path graphs, excluding the penultimate vertex of the graph to reserve space for the final vertex.

Corollary 16.5 (Knapsack Recurrence) *With the assumptions and notation of Lemma 16.4, let $V_{i,c}$ denote the maximum total value of a subset of the first i items with total size at most c. (When $i = 0$, interpret $V_{i,c}$ as 0.) For every $i = 1, 2, \ldots, n$ and $c = 0, 1, 2, \ldots, C$,*

$$V_{i,c} = \begin{cases} \underbrace{V_{i-1,c}}_{Case\ 1} & \text{if } s_i > c \\ \max\{\underbrace{V_{i-1,c}}_{Case\ 1}, \underbrace{V_{i-1,c-s_i} + v_i}_{Case\ 2}\} & \text{if } s_i \leq c. \end{cases}$$

Because both c and items' sizes are integers, the residual capacity $c - s_i$ in the second case is also an integer.

16.5.3 The Subproblems

The next step is to define the collection of relevant subproblems and solve them systematically using the recurrence identified in Corollary 16.5. For now, we focus on computing the total value of an optimal solution for each subproblem. As for the WIS problem in path graphs, we'll be able to reconstruct the items in an optimal solution to the original problem from this information.

Back in the WIS problem in path graphs, we used only one parameter i to index subproblems, where i was the length of the prefix of the input graph. For the knapsack problem, we can see from Lemma 16.4 and Corollary 16.5 that subproblems should be parameterized by *two* indices: the length i of the prefix of available items and the available knapsack capacity c.[20] Ranging over all relevant values of the two parameters, we obtain our subproblems:

Knapsack: Subproblems

Compute $V_{i,c}$, the total value of an optimal knapsack solution with the first i items and knapsack capacity c.

(For each $i = 0, 1, 2, \ldots, n$ and $c = 0, 1, 2, \ldots, C$.)

[20] In the WIS problem in path graphs, there's only one dimension in which a subproblem can get smaller (by having fewer vertices). In the knapsack problem, there are two (by having fewer items, or less knapsack capacity).

The largest subproblem (with $i = n$ and $c = C$) is exactly the same as the original problem. Because all item sizes and the knapsack capacity C are positive integers, and because capacity is always reduced by the size of some item (to reserve space for it), the only residual capacities that can ever come up are the integers between 0 and C.[21]

16.5.4 A Dynamic Programming Algorithm

Given the subproblems and recurrence, a dynamic programming algorithm for the knapsack problem practically writes itself.

```
Knapsack

Input: item values v_1, ..., v_n, item sizes s_1, ..., s_n, and
  a knapsack capacity C (all positive integers).
Output: the maximum total value of a subset
  S ⊆ {1, 2, ..., n} with Σ_{i∈S} s_i ≤ C.

-----------------------------------------------------------------
// subproblem solutions (indexed from 0)
A := (n + 1) × (C + 1) two-dimensional array
// base case (i = 0)
for c := 0 to C do
    A[0][c] = 0
// systematically solve all subproblems
for i := 1 to n do
    for c := 0 to C do
        // use recurrence from Corollary 16.5
        if s_i > c then
            A[i][c] := A[i − 1][c]
        else
            A[i][c] :=
              max{A[i − 1][c], A[i − 1][c − s_i] + v_i}
                  Case 1          Case 2
return A[n][C] // solution to largest subproblem
```

[21] Or, thinking recursively, each recursive call removes the last item and an integer number of units of capacity. The only subproblems that can arise in this way involve some prefix of the items and some integer residual capacity.

The array A is now two-dimensional to reflect the two indices i and c used to parameterize the subproblems. By the time an iteration of the double for loop must compute the subproblem solution $A[i][c]$, the values $A[i-1][c]$ and $A[i-1][c-s_i]$ of the two relevant smaller subproblems have already been computed in the previous iteration of the outer loop (or in the base case). We conclude that the algorithm spends $O(1)$ time solving each of the $(n+1)(C+1) = O(nC)$ subproblems, for an overall running time of $O(nC)$.[22,23]

Finally, as with `WIS`, the correctness of `Knapsack` follows by induction on the number of items, with the recurrence in Corollary 16.5 used to justify the inductive step.

Theorem 16.6 (Properties of `Knapsack`) *For every instance of the knapsack problem, the `Knapsack` algorithm returns the total value of an optimal solution and runs in $O(nC)$ time, where n is the number of items and C is the knapsack capacity.*

16.5.5 Example

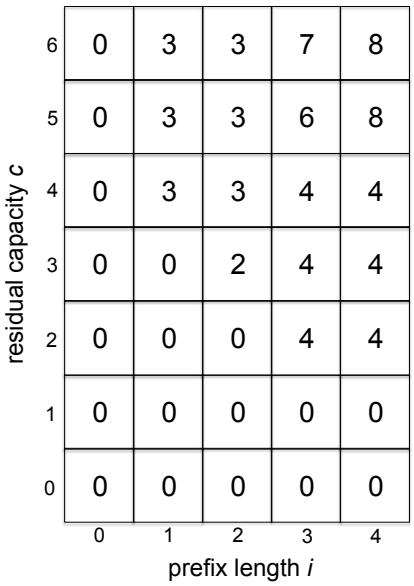
Recall the four-item example from Quiz 16.5, with $C = 6$:

Item	Value	Size
1	3	4
2	2	3
3	4	2
4	4	3

Because $n = 4$ and $C = 6$, the array A in the `Knapsack` algorithm can be visualized as a table with 5 columns (corresponding to $i = 0, 1, \ldots, 4$) and 7 rows (corresponding to $c = 0, 1, \ldots, 6$). The final array values are:

[22] In the notation of (16.1), $f(n, C) = O(nC)$, $g(n, C) = O(1)$, and $h(n, C) = O(1)$.

[23] The running time bound of $O(nC)$ is impressive only if C is small, for example, if $C = O(n)$ or ideally even smaller. In *Part 4* we'll see the reason for the not-so-blazingly fast running time—there is a precise sense in which the knapsack problem is a difficult problem.



`Knapsack` computes these entries column by column (working left to right), and within a column from bottom to top. To fill in an entry of the ith column, the algorithm compares the entry immediately to the left (corresponding to case 1) to v_i plus the entry one column to the left and s_i rows down (case 2). For example, for $A[2][5]$ the better option is to skip the second item and inherit the "3" immediately to the left, while for $A[3][5]$ the better option is to include the third item and achieve 4 (for v_3) plus the 2 in the entry $A[2][3]$.

16.5.6 Reconstruction

The `Knapsack` algorithm computes only the total value of an optimal solution, not the optimal solution itself. As with the `WIS` algorithm, we can reconstruct an optimal solution by tracing back through the filled-in array A. Starting from the largest subproblem in the upper-right corner, the reconstruction algorithm checks which case of the recurrence was used to compute $A[n][C]$. If it was case 1, the algorithm omits item n and resumes reconstruction from the entry $A[n-1][C]$. If it was case 2, the algorithm includes item n in its solution and resumes reconstruction from the entry $A[n-1][C-s_n]$.

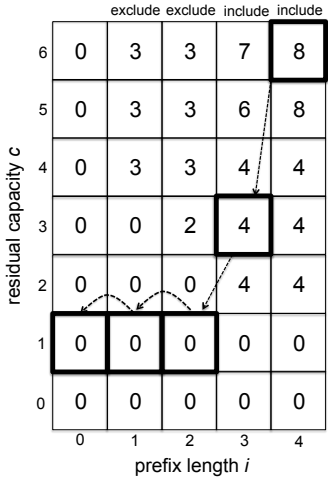
`Knapsack Reconstruction`

Input: the array A computed by the `Knapsack` algorithm with item values $v_1, v_2, \ldots, v_n$, item sizes $s_1, s_2, \ldots, s_n$, and knapsack capacity C.
Output: an optimal knapsack solution.

```
S := ∅                    // items in an optimal solution
c := C                              // remaining capacity
for i := n downto 1 do
    if s_i ≤ c and A[i − 1][c − s_i] + v_i ≥ A[i − 1][c] then
        S := S ∪ {i}          // Case 2 wins, include i
        c := c − s_i            // reserve space for it
    // else skip i, capacity stays the same
return S
```

The `Knapsack Reconstruction` postprocessing step runs in $O(n)$ time (with $O(1)$ work per iteration of the main loop), which is much faster than the $O(nC)$ time used to fill in the array in the `Knapsack` algorithm.[24]

For instance, tracing back through the array from the example on page 129 yields the optimal solution $\{3, 4\}$:



[24] In the notation of (16.1), postprocessing with the `Knapsack Reconstruction` algorithm increases the $h(n, C)$ term to $O(n)$. The overall running time $O(nC) \times O(1) + O(n) = O(nC)$ remains the same.

16.5.7 Solutions to Quizzes 16.5–16.6

Solution to Quiz 16.5

Correct answer: (c). Because the knapsack capacity is 6, there is no room to choose more than two items. The most valuable pair of items is the third and fourth ones (with total value 8), and these fit in the knapsack (with total size 5).

Solution to Quiz 16.6

Correct answer: (c). The most obviously false statement is (b), which doesn't even typecheck (C is in units of size, v_n in units of value). For example, v_n could be bigger than C, in which case $C - v_n$ is negative and meaningless. For (d), because S is feasible for the original problem, its total size is at most C; after n is removed from S, the total size drops to at most $C - s_n$ and, hence, $S - \{n\}$ is feasible for the reduced capacity. Answer (a) is a natural guess but is also incorrect.[25]

In (c), we are effectively reserving s_n units of capacity for item n's inclusion, which leaves a residual capacity of $C - s_n$. $S - \{n\}$ is a feasible solution to the smaller subproblem (with knapsack capacity $C - s_n$) with total value $V - v_n$. If there were a better solution $S^* \subseteq \{1, 2, \ldots, n-1\}$, with total value $V^* > V - v_n$ and total size at most $C - s_n$, then $S^* \cup \{n\}$ would have total size at most C and total value $V^* + v_n > (V - v_n) + v_n = V$. This would contradict the supposed optimality of S for the original problem.

The Upshot

☆ Dynamic programming follows a three-step recipe: (i) identify a relatively small collection of subproblems; (ii) show how to quickly solve "larger" subproblems given the solutions to "smaller" ones; and (iii) show how to quickly infer the final solution from the solutions to all

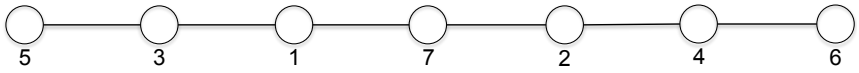
[25] For example, suppose $C = 2$ and consider two items, with $v_1 = s_1 = 1$ and $v_2 = s_2 = 2$. The optimal solution S is $\{2\}$. $S - \{2\}$ is the empty set, but the only optimal solution to the subproblem consisting of the first item and knapsack capacity 2 is $\{1\}$.

the subproblems.

- ☆ A dynamic programming algorithm that solves at most $f(n)$ different subproblems, using at most $g(n)$ time for each, and performs at most $h(n)$ postprocessing work to extract the final solution runs in $O(f(n) \cdot g(n) + h(n))$ time, where n denotes the input size.
- ☆ The right collection of subproblems and a recurrence for systematically solving them can be identified by reasoning about the structure of an optimal solution and the different ways it might be constructed from optimal solutions to smaller subproblems.
- ☆ Typical dynamic programming algorithms fill in an array with the values of subproblems' solutions, and then trace back through the filled-in array to reconstruct the solution itself.
- ☆ An independent set of an undirected graph is a subset of mutually non-adjacent vertices.
- ☆ In n-vertex path graphs, a maximum-weight independent set can be computed using dynamic programming in $O(n)$ time.
- ☆ In the knapsack problem, given n items with values and sizes and a knapsack capacity C (all positive integers), the goal is to select the maximum-value subset of items with total size at most C.
- ☆ The knapsack problem can be solved using dynamic programming in $O(nC)$ time.

Test Your Understanding

Problem 16.1 *(S)* Consider the input graph



where vertices are labeled with their weights. What are the final array entries of the `WIS` algorithm from Section 16.2, and which vertices belong to the MWIS?

Problem 16.2 *(S)* Consider an instance of the knapsack problem with five items:

Item	Value	Size
1	1	1
2	2	3
3	3	2
4	4	5
5	5	4

and knapsack capacity $C = 9$. What are the final array entries of the `Knapsack` algorithm from Section 16.5, and which items belong to the optimal solution?

Problem 16.3 *(H)* Which of the following statements hold? (Choose all that apply.)

a) The `WIS` and `WIS Reconstruction` algorithms of Sections 16.2 and 16.3 always return a solution that includes a maximum-weight vertex.

b) When vertices' weights are distinct, the `WIS` and `WIS Reconstruction` algorithms never return a solution that includes a minimum-weight vertex.

c) If a vertex v does not belong to an MWIS of the prefix G_i comprising the first i vertices and $i-1$ edges of the input graph, it does not belong to any MWIS of $G_{i+1}, G_{i+2}, \ldots, G_n$ either.

d) If a vertex v does not belong to an MWIS of G_{i-1} or G_i, it does not belong to any MWIS of $G_{i+1}, G_{i+2}, \ldots, G_n$ either.

Problem 16.4 *(H)* For the naive recursive algorithm for the WIS problem in path graphs (Section 16.2.2), prove that the number of leaves of its recursion tree (see footnote 7) is at least the nth Fibonacci number, where n is the number of vertices in the input graph.[26]

Problem 16.5 *(H)* Consider the following variation of the knapsack problem:

Problem: Double-Knapsack

Input: Item values $v_1, v_2, \ldots, v_n$, item sizes s_1, s_2, $\ldots$, s_n, and capacities C_1 and C_2 of *two* knapsacks. (All positive integers.)

Output: Two disjoint subsets $S_1, S_2 \subseteq \{1, 2, \ldots, n\}$ of items with the maximum-possible total value $\sum_{i \in S_1 \cup S_2} v_i$, subject to $\sum_{i \in S_1} s_i \leq C_1$ and $\sum_{i \in S_2} s_i \leq C_2$.

Here are two possible algorithmic approaches:

(1) Use the `Knapsack` algorithm from Section 16.5 to pick a maximum-value solution S_1 that fits in the first knapsack, and then use it again on the remaining items to pick a maximum-value solution S_2 that fits in the second knapsack.

(2) Use the `Knapsack` algorithm to pick a maximum-value solution S that would fit in a knapsack with capacity $C_1 + C_2$, then partition S arbitrarily into two sets S_1 and S_2 with total sizes at most C_1 and C_2, respectively.

Which of the following statements are true? (Choose all that apply.)

a) Algorithm (1) is guaranteed to produce an optimal solution to the double-knapsack problem but algorithm (2) is not.

[26] The *Fibonacci numbers* are the numbers in the sequence $1, 1, 2, 3, 5, 8, \ldots$, with each successive number defined as the sum of the previous two. The nth Fibonacci number is very closely approximated by $(\phi^n)/\sqrt{5}$, where $\phi = 1.618\ldots$ is the golden ratio. Thus, the lower bound proved in this problem is considerably larger than the bound of $2^{n/2} = (\sqrt{2})^n = (1.414\ldots)^n$ from the solution to Quiz 16.3.

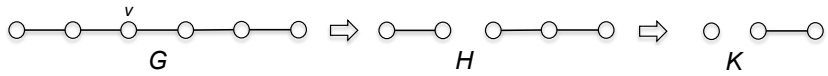
b) Algorithm (2) is guaranteed to produce an optimal solution to the double-knapsack problem but algorithm (1) is not.

c) Algorithm (1) is guaranteed to produce an optimal solution to the double-knapsack problem when $C_1 = C_2$.

d) Neither algorithm is guaranteed to produce an optimal solution to the double-knapsack problem.

Challenge Problems

Problem 16.6 *(H)* This problem outlines an approach to solving the WIS problem in graphs more complicated than paths. Consider an arbitrary undirected graph $G = (V, E)$ with nonnegative vertex weights, and an arbitrary vertex $v \in V$ with weight w_v. Obtain H from G by removing v and its incident edges. Obtain K from H by removing v's neighbors and their incident edges:



Let W_G, W_H, and W_K denote the total weight of an MWIS in G, H, and K, respectively, and consider the formula

$$W_G = \max\{W_H, W_K + w_v\}.$$

Which of the following statements are true? (Choose all that apply.)

a) The formula is not always correct in path graphs.

b) The formula is always correct in path graphs but not always correct in trees (that is, in connected acyclic graphs).

c) The formula is always correct in trees but not always correct in arbitrary graphs.

d) The formula is always correct in arbitrary graphs.

e) The formula leads to a linear-time dynamic programming algorithm for the WIS problem in trees.

f) The formula leads to a linear-time dynamic programming algorithm for the WIS problem in arbitrary graphs.

Problem 16.7 *(H)* This problem describes four generalizations of the knapsack problem. In each, the input consists of item values $v_1, v_2, \ldots, v_n$, item sizes $s_1, s_2, \ldots, s_n$, and additional problem-specific data (all positive integers). Which of these generalizations can be solved by dynamic programming in time polynomial in the number n of items and the largest number M that appears in the input? (Choose all that apply.)

a) Given a positive integer capacity C, compute a subset of items with the maximum-possible total value subject to having total size *exactly* C. (If no such set exists, the algorithm should correctly detect that fact.)

b) Given a positive integer capacity C and an item budget $k \in \{1, 2, \ldots, n\}$, compute a subset of items with the maximum-possible total value subject to having total size at most C *and at most* k *items*.

c) The double-knapsack problem from Problem 16.5: Given capacities C_1 and C_2 of two knapsacks, compute disjoint subsets S_1, S_2 of items with the maximum-possible total value $\sum_{i \in S_1} v_i + \sum_{i \in S_2} v_i$, subject to the knapsack capacities: $\sum_{i \in S_1} s_i \leq C_1$ and $\sum_{i \in S_2} s_i \leq C_2$.

d) Given capacities $C_1, C_2, \ldots, C_m$ of m knapsacks, where m could be as large as n, compute disjoint subsets $S_1, S_2, \ldots, S_m$ of items with the maximum-possible total value $\sum_{i \in S_1} v_i + \sum_{i \in S_2} v_i + \cdots + \sum_{i \in S_m} v_i$, subject to the knapsack capacities: $\sum_{i \in S_1} s_i \leq C_1$, $\sum_{i \in S_2} s_i \leq C_2$, $\ldots$, and $\sum_{i \in S_m} s_i \leq C_m$.

Programming Problems

Problem 16.8 Implement in your favorite programming language the `WIS` and `WIS Reconstruction` algorithms. (See `www.algorithmsilluminated.org` for test cases and challenge data sets.)

Problem 16.9 Implement in your favorite programming language the `Knapsack` and `Knapsack Reconstruction` algorithms. (See `www.algorithmsilluminated.org` for test cases and challenge data sets.)