Knapsack and Proofs of Correctness

CS/MCS 401: Computer Algorithms I
Lecture 21
Professor: Jonathan Liu

Icebreaker: If you could become good enough at something to write a book about it, what would it be?

Announcements

- HW 4 has been graded
- HW 5 has been released

Today's Overview

Today we'll wrap up this unit by learning about the Knapsack Problem, one of the most well-known algorithms problems.

We'll use it as a case study for proving the correctness of DP algorithms and analyzing their runtime.

The Knapsack Problem

The Knapsack Problem

$200



Size = 4

$110



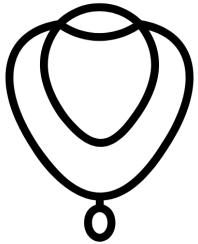
Size = 2

$70



Size = 3

$300



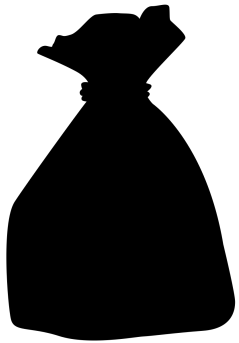
Size = 7

$240



Size = 5

Quick! You've broken into the jewelry shop and want to get away with as much value as you can. However, you've packed a really tiny knapsack to carry the jewelry. Which pieces should you grab to maximize your value?



Knapsack
Size = 10



The Knapsack Problem

$200



Size = 4

$110



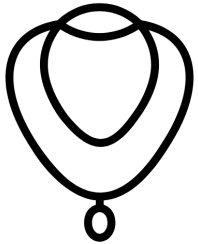
Size = 2

$70



Size = 3

$300



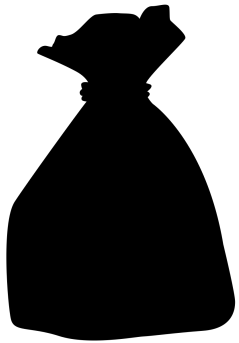
Size = 7

$240



Size = 5

Quick! You've broken into the jewelry shop and want to get away with as much value as you can. However, you've packed a really tiny knapsack to carry the jewelry. Which pieces should you grab to maximize your value?



Knapsack
Size = 10



Knapsack Problem Applications

Knapsack is a very human task that we do all the time:

- Picking what to do in your limited time per day
- Deciding what to put in your suitcase or leave out
- Deciding which exam questions to attempt in the time limit
- Picking people to invite (and not invite) to your wedding
- etc

Automated tasks do it all the time too:

- Picking which ads to place on a page,
- Picking which videos/songs to recommend to a user
- Choosing which sentences on a page are most important when summarizing a document
- Choosing how to allocate compute time or bandwidth for various tasks
- etc

Knapsack: Greedy Approaches

Heuristics:

A. Take the item with highest value.
B. Take the item with minimum weight.
C. Take the item with best value-to-weight ratio.

There are n items in total. Each has a *value* v_i and a *weight* w_i. You have a knapsack of size W. Pick the set of items that maximizes the *value*, where the total *weight* is no larger than W.

Pick the one you think works, and try it at https://tinyurl.com/knapsack-practice.

None work!

Knapsack: Dynamic Programming

Subproblem:

A. OPT[w] = The best you can do with a bag of size w.
B. OPT[i] = The best you can do with items 1...i.
C. OPT[w][i] = The best you can do with a bag of size w and items 1...i.

Try writing a recurrence!

There are n items in total. Each has a *value* v_i and a *weight* w_i. You have a knapsack of size W. Pick the set of items that maximizes the *value*, where the total *weight* is no larger than W.

Knapsack: Dynamic Programming

Subproblem:

A. OPT[w] = The best you can do with a bag of size w.

B. OPT[i] = The best you can do with items 1…i.

C. OPT[w][i] = The best you can do with a bag of size w and items 1…i.

Try writing a recurrence!

There are n items in total. Each has a *value* v_i and a *weight* w_i. You have a knapsack of size W. Pick the set of items that maximizes the *value*, where the total *weight* is no larger than W.

A and C both work! For today, we're going to focus on C.

Knapsack: Dynamic Programming

Recurrence: We want to know OPT[w][i].

Either you take item i or you don't. If you take item i, the best you can do with the remaining items is...

There are n items in total. Each has a *value* v_i and a *weight* w_i. You have a knapsack of size W. Pick the set of items that maximizes the *value*, where the total *weight* is no larger than W.

OPT[w][i] = The best you can do with a bag of size w and items 1...i.

Knapsack: Dynamic Programming

Recurrence: We want to know OPT[w][i].

Either you take item i or you don't. If you take item i, the best you can do with the remaining items is OPT[w-w_i][i-1].

Otherwise, if you don't take item i, the best you can do is …

There are n items in total. Each has a *value* v_i and a *weight* w_i. You have a knapsack of size W. Pick the set of items that maximizes the *value*, where the total *weight* is no larger than W.

OPT[w][i] = The best you can do with a bag of size w and items 1…i.

Knapsack: Dynamic Programming

Recurrence: We want to know OPT[w][i].

Either you take item i or you don't. If you take item i, the best you can do with the remaining items is OPT[w-w_i][i-1].

Otherwise, if you don't take item i, the best you can do is OPT[w][i-1].

So, our final recurrence is

...

There are n items in total. Each has a *value* v_i and a *weight* w_i. You have a knapsack of size W. Pick the set of items that maximizes the *value*, where the total *weight* is no larger than W.

OPT[w][i] = The best you can do with a bag of size w and items 1...i.

Knapsack: Dynamic Programming

Recurrence: We want to know OPT[w][i].

Either you take item i or you don't. If you take item i, the best you can do with the remaining items is OPT[w-w_i][i-1].

Otherwise, if you don't take item i, the best you can do is OPT[w][i-1].

So, our final recurrence is

$$OPT[w][i] = \max(OPT[w-w_i][i-1] + v_i, OPT[w][i-1])$$

There are n items in total. Each has a *value* v_i and a *weight* w_i. You have a knapsack of size W. Pick the set of items that maximizes the *value*, where the total *weight* is no larger than W.

OPT[w][i] = The best you can do with a bag of size w and items 1...i.

Knapsack: Dynamic Programming

Subproblem: OPT[w][i] = The best you can do with a bag of size w and items 1...i.

Recurrence:

$$OPT[w][i] = \max(OPT[w-w_i][i-1] + v_i, OPT[w][i-1])$$

Iteration Order: By row or by column!

Base Case: First row = 0.
First column = v_i when $w \geq w_i$.

Return Value: OPT[W][n]



	i=1	i=2	...	i=n
w=0				
w=1				
w=2				
...				
w=W				

Variants of Knapsack

Variant Name	Example
0-1 Knapsack	What we just did! You take 0 or 1 of each item.
(Unbounded) Knapsack	There are infinite copies of each item.
Bounded Knapsack	There are finite (but more than 1) copies of items.
Fractional Knapsack	You can take fractional pieces of items.
Multiple Knapsack	There are multiple knapsacks to fill.
Multidimensional Knapsack	You are constrained by, e.g., size **and** weight of an item.

Try designing an algorithm for unbounded knapsack!

Unbounded Knapsack

0-1 Knapsack	Unbounded Knapsack
Subproblem: OPT[w][i] = The best you can do with a bag of size w and items 1...i.	**Subproblem:**
Recurrence: OPT[w][i] = max(OPT[w][i-1], OPT[w-w_i][i-1] + v_i)	**Recurrence:**
Iteration Order: By row or by column!	**Iteration Order:**
Base Case: First row = 0. First column = v_i when w ≥ w_i.	**Base Case:**
Return Value: OPT[W][n]	**Return Value:**

Unbounded Knapsack

0-1 Knapsack	Unbounded Knapsack
Subproblem: OPT[w][i] = The best you can do with a bag of size w and items 1...i.	**Subproblem:** OPT[w][i] = The best you can do with a bag of size w and items 1...i.
Recurrence: OPT[w][i] = max(OPT[w][i-1], OPT[w-w_i][i-1] + v_i)	**Recurrence:**
Iteration Order: By row or by column!	**Iteration Order:**
Base Case: First row = 0. First column = v_i when w ≥ w_i.	**Base Case:**
Return Value: OPT[W][n]	**Return Value:**

Unbounded Knapsack

0-1 Knapsack	Unbounded Knapsack
Subproblem: OPT[w][i] = The best you can do with a bag of size w and items 1...i.	**Subproblem:** OPT[w][i] = The best you can do with a bag of size w and items 1...i.
Recurrence: OPT[w][i] = max(OPT[w][i-1], $OPT[w-w_i][i-1] + v_i$)	**Recurrence:** OPT[w][i] = max(OPT[w][i-1], **$OPT[w-w_i][i] + v_i$**)
Iteration Order: By row or by column!	**Iteration Order:**
Base Case: First row = 0. First column = v_i when $w \geq w_i$.	**Base Case:**
Return Value: OPT[W][n]	**Return Value:**

Unbounded Knapsack

0-1 Knapsack	Unbounded Knapsack
Subproblem: OPT[w][i] = The best you can do with a bag of size w and items 1...i.	**Subproblem:** OPT[w][i] = The best you can do with a bag of size w and items 1...i.
Recurrence: OPT[w][i] = max(OPT[w][i-1], OPT[w-w_i][i-1] + v_i)	**Recurrence:** OPT[w][i] = max(OPT[w][i-1], **OPT[w-w_i][i] + v_i**)
Iteration Order: By row or by column!	**Iteration Order:** By row or by column!
Base Case: First row = 0. First column = v_i when $w \geq w_i$.	**Base Case:**
Return Value: OPT[W][n]	**Return Value:**

Unbounded Knapsack

0-1 Knapsack	Unbounded Knapsack
Subproblem: OPT[w][i] = The best you can do with a bag of size w and items 1...i.	**Subproblem:** OPT[w][i] = The best you can do with a bag of size w and items 1...i.
Recurrence: OPT[w][i] = max(OPT[w][i-1], OPT[w-w_i][i-1] + v_i)	**Recurrence:** OPT[w][i] = max(OPT[w][i-1], **OPT[w-w_i][i] + v_i**)
Iteration Order: By row or by column!	**Iteration Order:** By row or by column!
Base Case: First row = 0. First column = v_i when w ≥ w_i.	**Base Case:** First row = 0. First column **= ?**
Return Value: OPT[W][n]	**Return Value:**

Unbounded Knapsack

0-1 Knapsack	Unbounded Knapsack
Subproblem: OPT[w][i] = The best you can do with a bag of size w and items 1...i.	**Subproblem:** OPT[w][i] = The best you can do with a bag of size w and items 1...i.
Recurrence: OPT[w][i] = max(OPT[w][i-1], $OPT[w-w_i][i-1] + v_i$)	**Recurrence:** OPT[w][i] = max(OPT[w][i-1], **$OPT[w-w_i][i] + v_i$**)
Iteration Order: By row or by column!	**Iteration Order:** By row or by column!
Base Case: First row = 0. First column = v_i when $w \geq w_i$.	**Base Case:** First row = 0. **First column = $(w // w_i)\ v_i$**
Return Value: OPT[W][n]	**Return Value:**

Unbounded Knapsack

0-1 Knapsack	Unbounded Knapsack
Subproblem: OPT[w][i] = The best you can do with a bag of size w and items 1...i.	**Subproblem:** OPT[w][i] = The best you can do with a bag of size w and items 1...i.
Recurrence: OPT[w][i] = max(OPT[w][i-1], OPT[w-w_i][i-1] + v_i)	**Recurrence:** OPT[w][i] = max(OPT[w][i-1], **OPT[w-w_i][i] + v_i**)
Iteration Order: By row or by column!	**Iteration Order:** By row or by column!
Base Case: First row = 0. First column = v_i when w ≥ w_i.	**Base Case:** First row = 0. **First column = (w // w_i) v**
Return Value: OPT[W][n]	**Return Value:** OPT[W][n]

Proofs of Correctness

Proving Correctness

0-1 Knapsack

Subproblem: OPT[w][i] = The best you can do with a bag of size w and items 1...i.

Recurrence: OPT[w][i] = max(OPT[w][i-1], OPT[w-w_i][i-1] + v_i)

Iteration Order: By row or by column!

Base Case: First row = 0.
First column = v_i when w ≥ w_i.

Return Value: OPT[W][n]

Need to prove: This algorithm returns the optimal set of items to take.

To do so, we must prove:

- OPT[W][n] is the optimal set of items to take.

To do that, we must prove:

- Every entry in the array is computed correctly.

To do that, we'll use induction!

- How do we prove by induction on multiple dimensions?
  - Just pick one dimension!

A Complete Proof of Correctness

0-1 Knapsack

Subproblem: OPT[w][i] = The best you can do with a bag of size w and items 1…i.

Recurrence: OPT[w][i] = max(OPT[w-w_i][i-1] + v_i, OPT[w][i-1])

Iteration Order: By row or by column!

Base Case: First row = 0.
First column = v_i when w ≥ w_i.

Return Value: OPT[W][n]

Need to prove: OPT[w][i] = The best you can do with a bag of size w and items 1…i.

First, we will prove that the entries in the array are correct. We will prove that, for any w and any i, OPT[w][i] contains the optimal value with a bag of size w and using items 1 to i. We will proceed by induction on i.
Base Case: When i = 1, there's only one item, so we should take it if we can. So v_i when w >= w_i.
Inductive Hypothesis: For j=1 … i-1, OPT[w][j] is correct for any w.
Inductive Step: We need to show that OPT[w][i] is correct. Any configuration either takes item i or it doesn't. If item i is taken, the best we can do is v_i + OPT[w-w_i][i-1] by the IH. If item i is not taken, the best we can do is OPT[w][i-1], which is also correct by the IH. Thus, the best we can do is the max of these two values, so our algorithm computes the correct thing for OPT[w][i].
We have shown that OPT[w][i] is computed correctly for all w and all i. Therefore, OPT[W][n] contains the optimal value with a bag of size W and items 1…n. That is what the original problem was asking for, so we return the correct value.

Proofs of Runtime

Proving Runtime

0-1 Knapsack

Subproblem: OPT[w][i] = The best you can do with a bag of size w and items 1...i.

Recurrence: OPT[w][i] = max(OPT[w-w_i][i-1] + v_i, OPT[w][i-1])

Iteration Order: By row or by column!

Base Case: First row = 0.
First column = v_i when w ≥ w_i.

Return Value: OPT[W][n]

What is the runtime of this algorithm?

We have an array of size W x n. For each entry, we're doing one comparison and some addition to compute its value. Thus, the runtime is O(nW).

That's polynomial in n and W, so it's efficient!

Proving Runtime

0-1 Knapsack

Subproblem: OPT[w][i] = The best you can do with a bag of size w and items 1...i.

Recurrence: OPT[w][i] = max(OPT[w-w_i][i-1] + v_i, OPT[w][i-1])

Iteration Order: By row or by column!

Base Case: First row = 0.
First column = v_i when w ≥ w_i.

Return Value: OPT[W][n]

What is the runtime of this algorithm?

We have an array of size W x n. For each entry, we're doing one comparison and some addition to compute its value. Thus, the runtime is O(nW).

~~That's polynomial in n and W, so it's efficient!~~

What is the **size** of W?

W is an input of size O(log W).

So, a runtime of O(nW) is an **exponential function of its input size. It is not a poly-time algorithm.**

Pseudo-polynomial Time Algorithms

Knapsack takes as input two things: an array A, and a target weight W.

- The **size** of the array is O(n). It takes n blocks of data to represent an array of size n.
- **The size of the knapsack weight is O(log W).** It only takes log W bits to specify the weight.

Runtime is a function of input size. When we say that a function runs in $O(n^2)$ time, what we are saying is, the runtime grows proportionally to the input size squared.

The input of the weight has size O(log W), but the runtime is O(W). **Therefore, the runtime is $O(2^{\text{input size}})$.**

This is known as a pseudo-polynomial algorithm: it is polynomial in terms of the value of an integer input, not in terms of the **size** of that input.

Lecture Recap

Today we talked about the knapsack algorithm, one of the most important problems in computer science.

Using that algorithm, we then talked about how to write proofs of correctness for dynamic programming algorithms.

Finally, we saw that Knapsack is, surprisingly, not a poly-time algorithm. All the other algorithms don't depend on how large the input numbers are, but this one does!