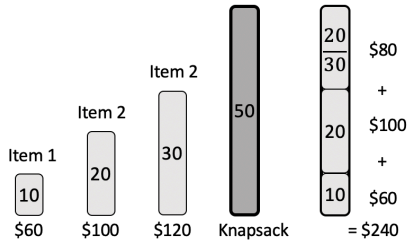


Greedy vs dynamic algorithms
Lecture 22

COSC 242 – Algorithms and Data Structures

UNIVERSITY of OTAGO
Te Whare Wānanga o Otāgo
NEW ZEALAND
SAPERE AUDE

Today’s outline

1. Greedy overview
2. Knapsack problems
3. Fractional knapsack
4. 0-1 knapsack problem
5. Visualising the problem space
6. Recursive top-down implementation
7. Top-down with memoisation
8. Bottom-up method

Today’s outline

1. Greedy overview
2. Knapsack problems
3. Fractional knapsack
4. 0-1 knapsack problem
5. Visualising the problem space
6. Recursive top-down implementation
7. Top-down with memoisation
8. Bottom-up method

Greedy algorithms

Dijkstra's algorithm and Prim's algorithm are both examples of **greedy algorithms**.

A greedy algorithm tries to solve an optimisation problem by making a sequence of choices.

At each decision point, it chooses the locally optimal choice in the hope that it will lead to a globally optimal solution. This is such a simple approach that it is what one usually tries first.

Greedy algorithms do not always yield optimal solutions, but for many problems they do.

Greedy vs dynamic programming

The choices made by greedy algorithms may depend on choices already made, but it cannot depend on the outcome of future unmade choices.

This contrasts with dynamic programming, which we will see in L23-24, which solves subproblems before making the first choice. In contrast, a greedy algorithm makes a choice before solving any subproblems.

Thus, dynamic programming can be seen as *bottom-up*, making a choice after assembling smaller solutions, whereas greedy programming is *top-down*, making one greedy choice after another, reducing each given problem instance to a smaller one.

Greedy algorithms

In both Dijkstra's and Prim's algorithms, a priority queue is used to extract the next node to visit.

Priority queues are essential data structures for many greedy algorithms.

Today’s outline

1. Greedy overview
2. Knapsack problems
3. Fractional knapsack
4. 0-1 knapsack problem
5. Visualising the problem space
6. Recursive top-down implementation
7. Top-down with memoisation
8. Bottom-up method

Knapsack problem

Consider now a totally different kind of optimisation problem.

A thief robbing a store has n items to choose from. These items belong to a set $S = \{s_1, s_2, \ldots, s_n\}$.

The i th item s_i is worth v_i dollars, and has a weight w_i, where v_i and w_i are integers.

The thief wants to take as valuable load as possible. But the thief can only carry a maximum weight of w_{max} .

Which items should the thief take that will maximise the load value?

0-1 and Fractional Knapsack problems

In the **0-1 knapsack problem**, the thief can either take an item, or leave the item. It is a binary choice, and hence the name 0-1: the thief cannot take a fractional amount of an item.

In the **Fractional knapsack problem**, the scenario is the same, but the thief can take fractions of items, rather than making a binary (0-1) choice.

If it helps, you can think of the 0-1 problems as involving gold bars. Whereas in the fractional problem, it involves bags of gold dust.

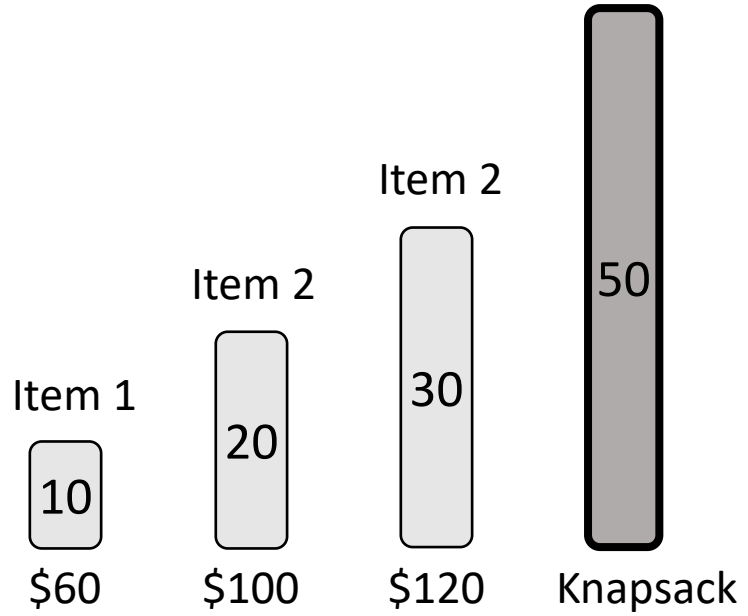
You can’t take a fraction of a gold bar, but you can take a fraction of a bag of gold dust.

Knapsack setup

The thief's max carry weight w_{max} = 50 kg. The thief can choose from the following items:

1. v_1 = \$60, w_1 = 10 kg.
2. v_2 = \$100, w_1 = 20 kg.
3. v_3 = \$120, w_1 = 30 kg.

Notice that the total weight of all three items is 60 kg, exceeding w_{max}.



Today’s outline

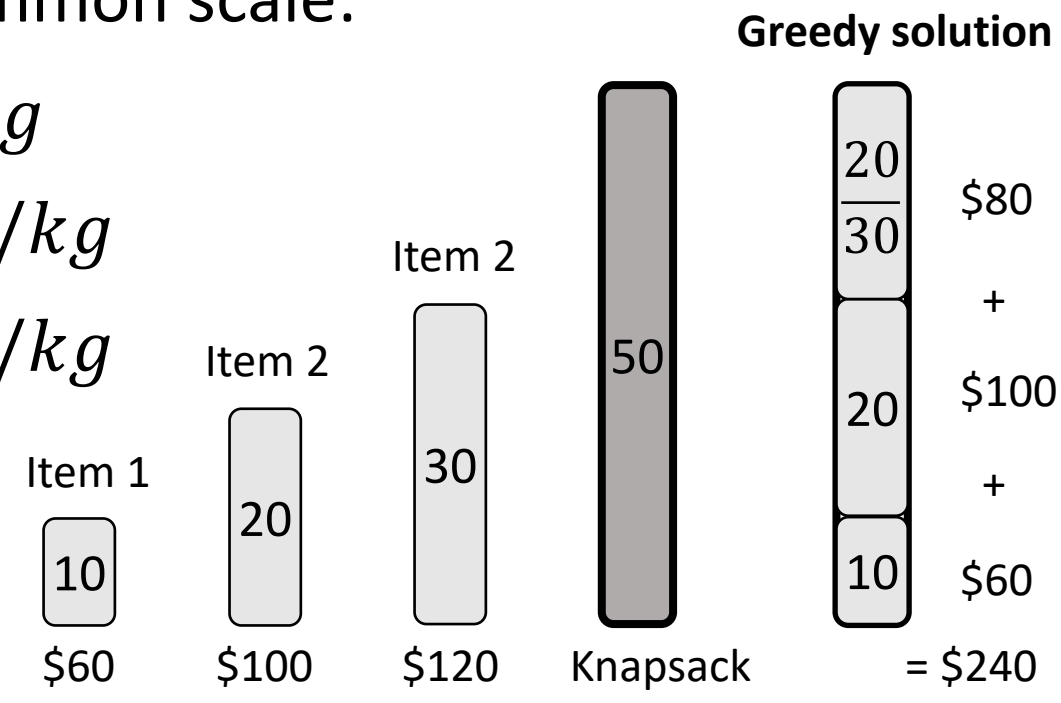
1. Greedy overview
2. Knapsack problems
3. Fractional knapsack
4. 0-1 knapsack problem
5. Visualising the problem space
6. Recursive top-down implementation
7. Top-down with memoisation
8. Bottom-up method

Fractional knapsack greedy approach

We will attempt a greedy approach for the fractional problem. What does that mean? We will prioritise items by their value per kg.

We'll call this an item's priority (or profit), p_i. This is a normalised value and with it we can compare items on a common scale:

1. v_1 = \$60, w_1 = 10 kg, $p_1 = {}^{60}/_{10} = \$6/kg$
2. v_2 = \$100, w_1 = 20 kg, $p_2 = {}^{100}/_{20} = \$5/kg$
3. v_3 = \$120, w_1 = 30 kg, $p_3 = {}^{120}/_{30} = \$4/kg$



Fractional knapsack greedy algorithm

```
procedure FracKnap(S, V, W, w_max)          // Items, Values, Weights, max weight
1:    Initialise priority queue Q           // Initialise empty
2:    for each s_i ∈ S
3:        p_i = v_i / w_i                   // pi is value to prioritise
4:        Q.Enqueue(s_i) using pi as priority // Max-priority queue
5:    current_weight = 0                    // Thief's current weight
6:    knapsack = ∅
7:    while current_weight < w_max          // Keep taking while we can carry more
8:        s_k = Q.Dequeue()                 // Get item with max profit
9:        x_k = min(w_k, w_max - current_weight) // How much of item's weight can we take?
10:       current_weight = current_weight + x_k
11:       knapsack = knapsack U {x_k / w_k * s_k}   // Add weight fraction of item s_k
end procedure
```

Today's outline

1. Greedy overview
2. Knapsack problems
3. Fractional knapsack
4. 0-1 knapsack problem
5. Visualising the problem space
6. Recursive top-down implementation
7. Top-down with memoisation
8. Bottom-up method

0-1 knapsack greedy approach

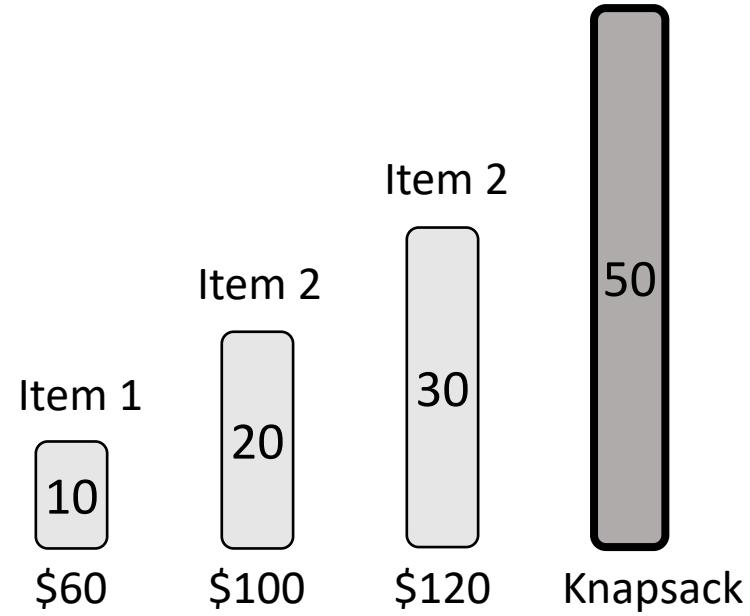
We will now attempt to solve the 0-1 problem. We will again use the greedy approach. Recall that in this version we can either take (1) or leave (0) an item; it is a binary choice.

What solution is given by the greedy approach?

What is the optimal solution?

Items

1. v_1 = \$60, w_1 = 10 kg.
2. v_2 = \$100, w_1 = 20 kg.
3. v_3 = \$120, w_1 = 30 kg.

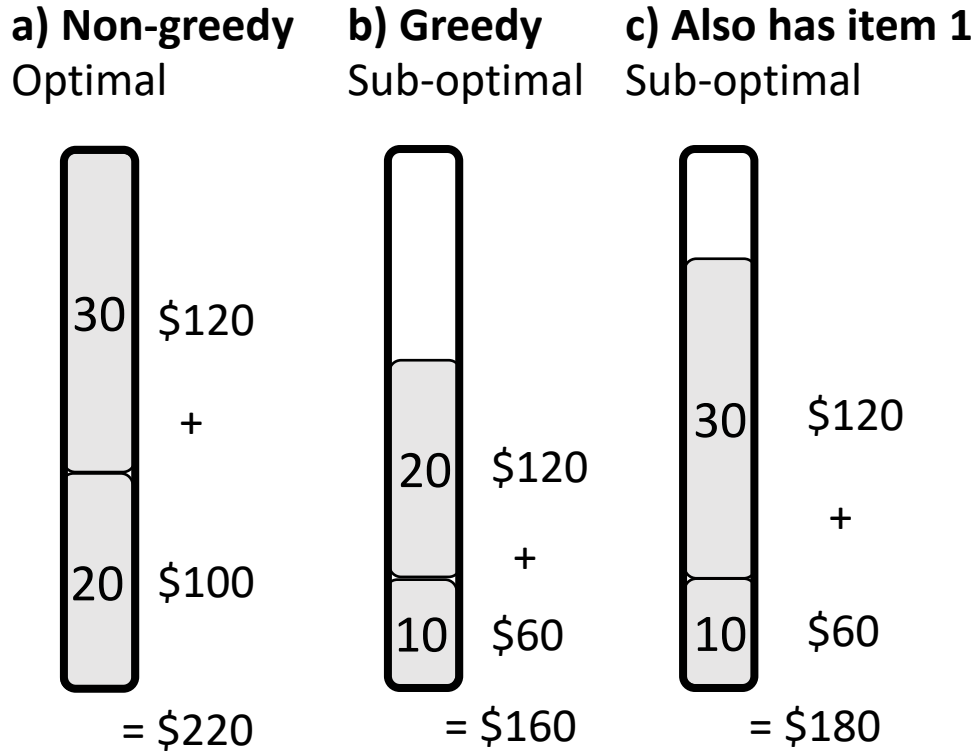


0-1 knapsack solution

a) The optimal subset includes items 2 and 3

b) The greedy approach. We can see that this approach yields a suboptimal solution.

c) In fact, any solution that involves taking item 1 is suboptimal.



Class challenge

What about this scenario?

W_{max} = 50 kg

Metric	Items									
Weight	5	3	10	20	22	17	30	25	15	31
Value	40	23	60	100	102	72	120	45	20	31
Profit	8	7.7	6	5	4.6	4.2	4	1.8	1.3	1.0

Solving the 0-1 knapsack problem

Given: a Set $S = \{s_1, s_2, ..., s_n\}$, where each item s_i has a positive benefit (or value) v_i and has a weight (or cost) w_i. Take v_i and w_i to be integers. A maximum total weight of w_{max}.

Required: to choose a subset of S such that the total weight does not exceed w_{max} and the sum of the values v_i is maximal.

Solving the 0-1 knapsack problem

Let V be a 2D array storing the maximum value possible considering the first k items in S (call those k items, S_k), and maximum total weight w.

We'll come back to this array in a few slides…

If $k \in S_k$ and we remove k from S_k (call it S_{k-1}), then the resulting set must be the optimum for the problem with a maximum total weight of $w - w_k$. Why?

Optimal substructure

A requirement of dynamic programming and greedy approaches is that the problem have the property known as **optimal substructure**.

A problem exhibits optimal substructure if an optimal solution contains within it optimal solutions to subproblems.

Recall: Dijkstra's algorithm relied on this property. From L21-S10:

"*A shortest path algorithm relies on the property that a shortest path between two vertices contains other shortest paths within it*."

That is, the shortest paths problem exhibited optimal substructure.

Optimal substructure in Knapsack

Both knapsack problems exhibit optimal substructure. Therefore, we can consider greedy and dynamic approaches to these problems.

The fractional knapsack is best solved by a *greedy approach*.

The 0-1 approach is best solved by a *dynamic programming approach*.

For the 0-1 problem, consider the most valuable load that weights at most W kg. If we remove item j, the remaining load must be the most valuable load weighting at most $W - w_j$ that the thief can take from the $n - 1$ original items excluding item j.

Recursive non-greedy solution

We are now left with the following observations:

1) If there are no items in our set S_0, then the maximum value is 0.
2) If there is no space in our knapsack, then the maximum value is 0
3) If the k^{th} item can't fit in the knapsack, then the maximum is the same as the maximum for $k - 1$ items.
4) Otherwise, the maximum is either:
   - the maximum without the k^{th} item in the optimal set, in which case we have a new problem with $k - 1$ items and maximum weight w.
   - the maximum with the k^{th} item in the optimal set, in which case we have a new problem with $k - 1$ items and maximum weight $w - w_k$.

Recursive non-greedy solution

So we can define our optimum $V[k, w]$ recursively as:

$$V[0, w] = 0$$
$$V[k, 0] = 0$$
$$V[k, w] = V[k-1, w] \; if \; w_k > w$$
$$V[k, w] = \max(V[k-1, w], v_k + V[k-1, w-w_k])$$

1) No items
2) No space
3) k^{th} item can't fit
4) Max of:
   a) Without k^{th}
   b) With k^{th}

Today’s outline

1. Greedy overview
2. Knapsack problems
3. Fractional knapsack
4. 0-1 knapsack problem
5. Visualising the problem space
6. Recursive top-down implementation
7. Top-down with memoisation
8. Bottom-up method

Example: Visualising our item array

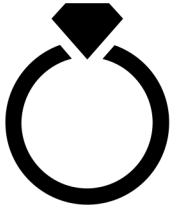
Before we implement the pseudocode, lets return to the 2D array. Every dynamic programming algorithm involves such an array.

Lets choose a simple 0-1 knapsack example to visualise in our array.

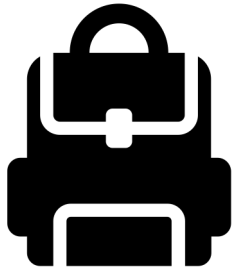
A thief can choose from three items:

Television
$300
4 kg

Laptop
$200
3 kg



Jewellery
$150
1 kg



w_{max} = 4kg

Example: Visualising our item array

Here's our 2D array for this problem. Each row represents the current best guess for max value in the knapsack. For each row, you can only consider the current item, or ones prior to it.

w Knapsack max weight from 1 to 4 kg

One row for each item for thief to choose from

k

	1	2	3	4
Jewellery				
Television				
Laptop				

Example: Visualising our item array

Jewellery
$150
1 kg

TV
$300
4 kg

Laptop
$200
3 kg

w_{max} = 4kg

Lets start with the jewellery row. For each cell we make our binary choice: take the item (1) or leave it (0). We must adhere to the weight limit set by the column.

	1	2	3	4
Jewellery				
Television				
Laptop				

Remember: Each row represents the current best guess for max value in the knapsack.

Example: Visualising our item array

Jewellery
$150
1 kg

TV
$300
4 kg

Laptop
$200
3 kg

w_{max} = 4kg

Lets start with the jewellery row. For each cell we make our binary choice: take the item (1) or leave it (0). We must adhere to the weight limit set by the column.

After the jewellery row, the thief's best guess to steal: jewellery for $150.

	1	2	3	4
Jewellery	$150 (J)	$150 (J)	$150 (J)	$150 (J)
Television				
Laptop				

Example: Visualising our item array

Jewellery
$150
1 kg

TV
$300
4 kg

Laptop
$200
3 kg

w_{max} = 4kg

Lets do the television row next. Now, in [2,1] we have two items we can take: Jewellery or Television. But we're still weight limited to 1kg.

Current max for 1kg knapsack

	1	2	3	4
Jewellery	$150 (J)	$150 (J)	$150 (J)	$150 (J)
Television				
Laptop				

New max for 1kg knapsack

Example: Visualising our item array

Jewellery
$150
1 kg

TV
$300
4 kg

Laptop
$200
3 kg

w_{max} = 4kg

Since the television weights 4 kg, our best option is to still steal the jewellery. In fact, that remains our choice until [2,4], at which point it's optimal to steal the TV.

We've now updated our estimate. With a 4kg knapsack, the thief can steal at least $300.

	1	2	3	4
Jewellery	$150 (J)	$150 (J)	$150 (J)	$150 (J)
Television	$150 (J)	$150 (J)	$150 (J)	$300 (TV)
Laptop				

Example: Visualising our item array

Jewellery
$150
1 kg

TV
$300
4 kg

Laptop
$200
3 kg

w_{max} = 4kg

Lets finish with the laptop row. We'll do the same thing, until the final cell in [3,4], the important part.

Our current best guess for 4kg is $300. We could take the laptop instead, but that's only worth $200.

But notice, if we take the laptop, we still have 1kg of space…

	1	2	3	4
Jewellery	$150 (J)	$150 (J)	$150 (J)	$150 (J)
Television	$150 (J)	$150 (J)	$150 (J)	$300 (TV)
Laptop	$150 (J)	$150 (J)	$200 (L)	

Example: Visualising our item array

Jewellery
$150
1 kg

TV
$300
4 kg

Laptop
$200
3 kg

w_{max} = 4kg

So our real choice is:

($300) vs ($200 + 1 kg)

TV Laptop ?

	1	2	3	4
Jewellery	$150 (J)	$150 (J)	$150 (J)	$150 (J)
Television	$150 (J)	$150 (J)	$150 (J)	$300 (TV)
Laptop	$150 (J)	$150 (J)	$200 (L)	

Example: Visualising our item array

Jewellery
$150
1 kg

TV
$300
4 kg

Laptop
$200
3 kg

w_{max} = 4kg

What was our previous best guess for 1kg? The jewellery!

($300) vs ($200 + $150)

TV Laptop Jewellery

	1	2	3	4
Jewellery	$150 (J)	$150 (J)	$150 (J)	$150 (J)
Television	$150 (J)	$150 (J)	$150 (J)	$300 (TV)
Laptop	$150 (J)	$150 (J)	$200 (L)	$350 (J & L)

Example: Visualising our item array

Here's the formula for calculating the value at each cell, including our all important final cell:

$$V[k,w] = \max(V[k-1,w], v_k + V[k-1, w-w_k])$$

($300) vs ($200 + $150)

TV Laptop Jewellery

	1	2	3	4
Jewellery	$150 (J)	$150 (J)	$150 (J)	$150 (J)
Television	$150 (J)	$150 (J)	$150 (J)	$300 (TV)
Laptop $200	$150 (J)	$150 (J)	$200 (L)	$350 (J & L)

Jewellery
$150
1 kg

TV
$300
4 kg

Laptop
$200
3 kg

w_{max} = 4kg

Example: Visualising our item array

Jewellery
$150
1 kg

TV
$300
4 kg

Laptop
$200
3 kg

w_{max} = 4kg

Put another way, our formula means:

Previous max

$$cell[i,j] = V[k,w] = \max \begin{cases} 1. \text{ Previous max value at V[k-1, w]} \\ 2. \text{ Value of current item + V[k-1, w-}w_k\text{]} \end{cases}$$

(value remaining space)

	1	2	3	4
Jewellery	$150 (J)	$150 (J)	$150 (J)	$150 (J)
Television	$150 (J)	$150 (J)	$150 (J)	$300 (TV)
Laptop $200	$150 (J)	$150 (J)	$200 (L)	$350 (J & L)

Today's outline

1. Greedy overview
2. Knapsack problems
3. Fractional knapsack
4. 0-1 knapsack problem
5. Visualising the problem space
6. Recursive top-down implementation
7. Top-down with memoisation
8. Bottom-up method

Refresh: Recursive non-greedy solution

So we can define our optimum $V[k, w]$ recursively as:

$$V[0, w] = 0$$
$$V[k, 0] = 0$$
$$V[k, w] = V[k-1, w] \; if \; w_k > w$$
$$V[k, w] = \max(V[k-1, w], v_k + V[k-1, w - w_k])$$

1) No items
2) No space
3) k^{th} item can't fit
4) Max of:
   a) Without k^{th}
   b) With k^{th}

Recursive top-down implementation (Brute force)


```
procedure RecursiveKnapsack(k, W, V, wmax) // Knap item, Weights, Values, max weight
1:    if k==0 or wmax ≤ 0 return 0, Ø        // No items OR no space (1 & 2)
2:    if W[k]>wmax                           // Can't fit k into knapsack (3)
3:        return RecursiveKnapsack(k-1,W,V,wmax)
4:                    // Check the maximum value (v1) without item k (4a)
5:    v1, items_not = RecursiveKnapsack(k-1,W,V,wmax)
6:                    // Check the maximum value (v2) with item k (4b)
7:    v2, items_do = RecursiveKnapsack(k-1,W,V,wmax-W[k])
8:    v2 = v2 + V[k]                         // Add value of current item
9:    items_do.add(k)                        // Add item k to list of take items
10:   if v2 > v1
11:       return v2, items_do                // Do use item k
11:   else
12:       return v1, items_not               // Don't use item k
end procedure
```

Running our example

Jewellery
$150
1 kg

TV
$300
4 kg

Laptop
$200
3 kg

w_{max} = 4kg

With our example, we would call our procedure as follow:

```
k = 3
W = [1, 4, 3]
V = [150, 300, 200]
```

w_{max} = 4

RecursiveKnapsack(k=3, W=[1,4,3], V=[150,300,200], w_{max}=4)

Running our example

Jewellery
$150
1 kg

TV
$300
4 kg

Laptop
$200
3 kg

w_{max} = 4kg

Lets think about our recursive code for a moment. When drawing out recursive code, it's easy to make an error, so it's helpful to start out knowing what we should get back with an easy example.

In our first outer call to RecursiveKnapsack, Line 5 is going to return the maximum value and set of items **without** item k = 3, our laptop. Naturally, if we can't take the laptop, this must mean the maximal set has to be the TV. So we should get: v1=$300, not = {2}.

Then Lines 7-9 is going to return the maximum value **with** item k = 3, our laptop. On Line 7 we're looking for max value with only 1 kg of space, v2 = $150, do = {1}.

With our remaining 3 kg of space, on Lines 8-9 will add k=3, our laptop. So the solution will return: v2 = 350, do = {1, 3}.

You'll get to practise this in Tutorial Week 11-12.

Today's outline

1. Greedy overview
2. Knapsack problems
3. Fractional knapsack
4. 0-1 knapsack problem
5. Visualising the problem space
6. Recursive top-down implementation
7. Top-down with memoisation
8. Bottom-up method

Recursive knapsack

What's the time complexity of `RecursiveKnapsack`? It turns out to be exponential: $O(2^n)$. Showing this is left as a tutorial exercise.

In `RecursiveKnapsack`, W and V don't change, so the only things that change in different recursive calls are k and w_{max}. k can be any integer from 1 to n and w_{max} can be any integer from 1 to w_{max}. So we should be able to produce a solution in $O(n*w_{max})$.

Time-memory trade-off

The reason why `RecursiveKnapsack` is so expensive is because it recomputes the same values over and over again.

If we could store those values when they're computed and then retrieve them when needed, we could save ourselves a lot of computation.

Dynamic programming uses additional memory to save computation time, by saving earlier subproblem results.

Time-memory trade-off

Doing this can have a dramatic effect, turning an exponential problem into a polynomial one.

There are two general approaches: the first is top-down with **memoisation**. In this approach we retain a recursive structure, but save the result of each subproblem in an array or table.

The other is a bottom-up method, that uses iteration rather than recursion. It again stores already-solved problems in an array.

Top-down memoisation vs bottom-up

Both approaches yield algorithms with the same asymptotic running time.

However the bottom-up approach has much better constant factors, since there is less overhead for procedure calls.

As we know, each time we call a recursive function new values have to get added to the stack, as it requires allocation of a new stack frame.

Memoised top-down implementation


```
1:   initialise global memo[n, wmax] as 2D array, all cells set to -1
procedure KnapMemo(k, W, V, w_max)                  // item, Weights, Values, max weight
2:   if k==0 or wmax ≤ 0 return 0, ∅                // No items OR no space (1 & 2)
3:   if memo[k,wmax] != -1 return memo[k,wmax] // Shortcut using memo
4:   if W[k]> wmax
5:       memo[k,wmax] = KnapMemo(k-1,W,V,wmax) // Same as RK L2-3, but cache it
6:   else
7:       v1, items_not = KnapMemo(k-1,W,V,wmax) // Same as RK L5
8:       v2, items_do = KnapMemo(k-1,W,V,wmax-W[k]) // Same as RK L7
9:       v2 = v2 + V[k]
10:      items_do.add(k)
11:      if v2 > v1
12:          memo[k,wmax] = v2, items_do              // Do use item k
13:      else
14:          memo[k,wmax] = v1, items_not             // Don't use item k
15:  return memo[k,wmax]
end procedure
```

Today’s outline

1. Greedy overview
2. Knapsack problems
3. Fractional knapsack
4. 0-1 knapsack problem
5. Visualising the problem space
6. Recursive top-down implementation
7. Top-down with memoisation
8. Bottom-up method

Bottom up method


```
1:  function KnapIter(n, W, V, wmax)
2:      initiallise bestVal[n,wmax] as 2D array, set all to 0
3:      initiallise bestSet[n,wmax] as 2D array, set all to φ
4:      for k← 1 to n do
5:          for w← 1 to wmax do
6:              if W[k]>w then
7:                  bestVal[k,w] ← bestVal[k-1,w]; bestSet[k,w] ← bestSet[k-1,w]
8:              else
9:                  withKVal ← V[k] + bestVal[k-1,max(w-W[k],0)]
10:                 if bestVal[k-1,w] > withKVal then
11:                     bestVal[k,w] ← bestVal[k-1,w]; bestSet[k,w] ← bestSet[k-1,w]
12:                 else
13:                     bestVal[k,w] ← withKVal
14:                     bestSet[k,w] ← bestSet[k-1,max(w-W[k],0)]+k
15:     return bestVal, bestSet
```

Suggested reading

Greedy algorithms are discussed in Section 16. The introduction and Section 16.2 are particularly relevant.

The knapsack problem isn't really discussed in the textbook, though there is a bit at the end of Section 16.2

Dynamic programming is discussed in Section 15. The introduction and Section 15.3 are relevant.

Optimal substructure is discussed in Sections 15.3 and 16.2.

Grokking Algorithms, 2016, A. Y. Bhargava has a great illustration of dynamic programming, which inspired today's example.