# Cosc 201
# Algorithms and Data Structures
# Lecture 21 (12/5/2025)
# Dynamic Programming

Brendan McCane
brendan.mccane@otago.ac.nz

UNIVERSITY OF OTAGO
Te Whare Wānanga o Otāgo
NEW ZEALAND

# 0-1 Knapsack problem

Let's try the greedy algorithm on the 0-1 knapsack problem.


- In this version we have to put either all of an item, or none of an item in the knapsack.
- What is the greedy solution for the above problem? What is the optimal solution?
- What about this problem:

$$W = [20, 30, 10, 5, 15, 25, 3, 17, 22, 31]$$
$$V = [100, 120, 60, 40, 20, 45, 23, 72, 102, 31]$$
$$W_{\max} = 50.$$

# 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, $w_{\max}$.
**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.
Let $V$ be a function returning the maximum value possible considering the first $k$ items in $S$ (call those $k$ items, $S_k$), and available weight $w$.
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?

# Recursive non-greedy solution

We are left with the following observations:

- If there are no items in our set ($S_0$), then the maximum value is 0.
- If there is no space in our knapsack, then the maximum value is 0
- If the $k^{\text{th}}$ item can't fit in the knapsack, then the maximum is the same as the maximum for $k-1$ items.
- Otherwise, the maximum is either:
  - the maximum without the $k^{\text{th}}$ 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 $k^{\text{th}}$ 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:

$$
\begin{aligned}
V(0, w) &= 0 \\
V(k, 0) &= 0 \\
V(k, w) &= V(k-1, w) \text{ if } w_k > w \\
V(k, w) &= \max(V(k-1, w), v_k + V(k-1, w-w_k))
\end{aligned}
$$

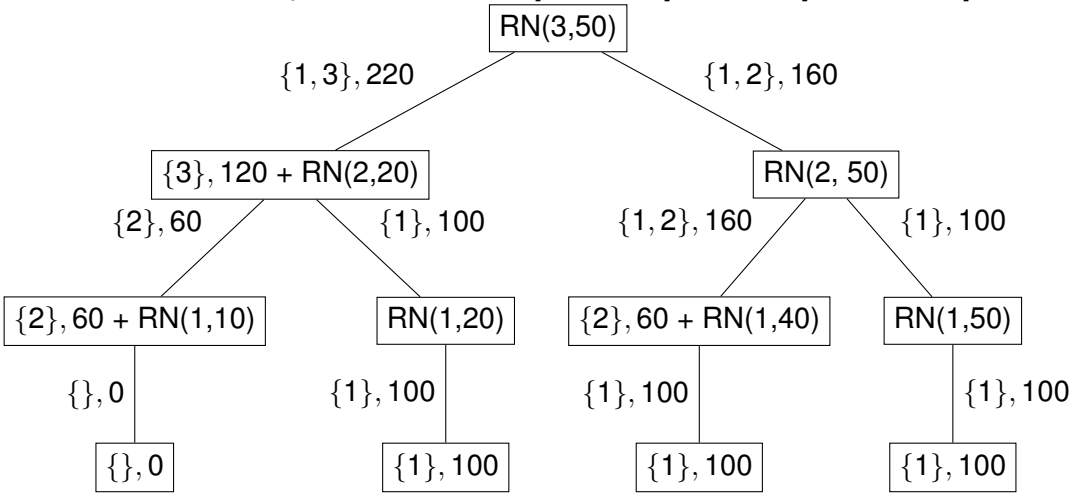
# The Recursive Algorithm

```
1: function RecursiveKnapsack(k, W, V, wmax)
2:     if k==0 or wmax ≤ 0 then return 0, φ
3:     end if
4:     if W[k]>wmax then                                   ▷ Can't fit k into knapsack
5:         return RecursiveKnapsack(k-1,W,V,wmax)
6:     end if
7:     v1, items_do ← RecursiveKnapsack(k-1,W,V,wmax-W[k])   ▷ Check the maximum value with k
8:     v1 ← v1 + V[k]                                      ▷ add the value of item k
9:     items_do.add(k)                                     ▷ add item k to the list
10:    v2, items_not ← RecursiveKnapsack(k-1,W,V,wmax)     ▷ Check the maximum value without k
11:    if v1>v2 then return v1, items_do                   ▷ do use k
12:    else return v2, items_not                           ▷ don't use k
13:    end if
14: end function
```

What is the complexity of this function?

# RecursiveKnapsack Call Tree

Let's consider our RN problem with $W = [20, 10, 30]$ and $V = [100, 60, 120]$:



# Efficiency of RecursiveKnapsack?

- Each call to RK, potentially spawns two more calls:
  - In the first call (line 7), $k$ is reduced by 1, and the weight left is reduced by the weight of item k.
  - In the second call (line 10), only $k$ is reduced by 1.
- Either way - each time we have a call that reduces only $k$ by 1, so about half of the call tree branches will be of length $n$.
- This means in the worst case, the number of calls to RK is exponential in $n$ (i.e. $2^n$). This is because the number of nodes in a tree of height $n$ is $2^{n+1} - 1$, and about half of our branches are height $n$.
- These are *very rough* approximations.
- The other branches are more difficult to predict because they depend on the weight of item $k$.

# Can we do better?

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

# Memoised Version

```
1: initiallise global memo[n, w_max] as 2D array, set all to -1
2: function KnapMemo(k, W, V, wmax)
3:     if k==0 or wmax ≤ 0 then return 0, φ
4:     end if
5:     if memo[k,wmax] ≠ −1 then return memo[k,wmax]
6:     end if
7:     if W[k]>wmax then memo[k,wmax] ← KnapMemo(k-1,W,V,wmax)
8:     else
9:         v1, items_do ← KnapMemo(k-1,W,V,wmax-W[k])
10:        v1 ← v1 + V[k]; items_do.add(k)          ▷ add item k to the list
11:        v2, items_not ← KnapMemo(k-1,W,V,wmax)
12:        if v1>v2 then memo[k,wmax] ← v2, items_do          ▷ do use k
13:        else memo[k,wmax] ← v1, items_not          ▷ don't use k
14:        end if
15:    end if
16:    return memo[k,wmax]
17: end function
```

# Top-down or Bottom-up?

- You might notice that in the KnapMemo algorithm, pretty much all we do is fill up the memo array which is of size $(n, w_{\max})$.
- Because of the way recursion works, we are starting the call at the largest values of $k$ and wmax, but filling the memo array at the smallest values first (basically depth-first on the call tree).
- We could instead fill up the array iteratively starting at the smallest values.
- But we still follow the same basic structure - the array is filled up by inspecting the entries with smaller $k$ and/or wmax.

# The full Dynamic Programming Solution

```
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:                end if
16:            end if
17:        end for
18:    end for
```

# Filling the array

Back to our problem: $W = [20, 10, 30]$ and $V = [100, 60, 120]$

bestVal

| $k \downarrow, w \rightarrow$ | 0 | 10 | 20 | 30 | 40 | 50 |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 100 | 100 | 100 | 100 |
| 2 | 0 | 60 | 100 | 160 | 160 | 160 |
| 3 | 0 | 60 | 100 | 160 | 180 | 220 |

I've only included the columns that are not copies of other columns. Since all the weights are multiples of 10, we only need to consider the column headings that are multiples of 10.

# Efficiency comparison

- Depending on the problem, either the memoised or iterative version can be the most efficient.
- For 0-1 knapsack, the memoised version has a worst case of $O(nw_{max})$, whereas the iterative version is $\Theta(nw_{max})$.
- If the weight distribution is sparse as in the 3-item case, then the memoised version will be more efficient because it doesn't fill up the whole array.
- If the weight distribution is dense, the memoised version does have to fill up the whole array, then the memoised version loses out because of the overhead of the recursive calls.
- For the 3-item case, the naive recursive version has 13 recursive calls, the memoised version has 13 calls, and the iterative version fills up 150 elements in the array.
- For the 10-item case, naive recursive has 397 calls, memoised has 254, and the iterative version fills up 1000 elements of the array.
- For a 1000-item case with maximum weight of 1000, memoised has 2M calls, and the iterative version fills up 1M elements. I stopped the naive version after 2 hours (more than 4B calls).

# Next time ...

More DP examples, plus a general classification of problems amenable to DP.