Dynamic programming

- memoization
- decorator memoized / functools.cache
- systematic subproblem computation

Binomial coefficient

```
--(7, 5)
  |--(6, 5)
  |   |--(5, 5)
  |    --(5, 4)
  |       |--(4, 4)
  |        --(4, 3)
  |           |--(3, 3)
  |            --(3, 2)
  |               |--(2, 2)
  |                --(2, 1)
  |                   |--(1, 1)
  |                    --(1, 0)
   --(6, 4)
      |--(5, 4)
      |   |--(4, 4)
      |    --(4, 3)
      |       |--(3, 3)
      |        --(3, 2)
      |           |--(2, 2)
      |            --(2, 1)
      |               |--(1, 1)
      |                --(1, 0)
       --(5, 3)
          |--(4, 3)
          |   |--(3, 3)
          |    --(3, 2)
          |       |--(2, 2)
          |        --(2, 1)
          |           |--(1, 1)
          |            --(1, 0)
           --(4, 2)
              |--(3, 2)
              |   |--(2, 2)
              |    --(2, 1)
              |       |--(1, 1)
              |        --(1, 0)
               --(3, 1)
                  |--(2, 1)
                  |   |--(1, 1)
                  |    --(1, 0)
                   --(2, 0)
```

identical computations

$$\binom{n}{k} = \begin{cases} 1 & \text{if } k = 0 \text{ or } k = n \\ \binom{n-1}{k} + \binom{n-1}{k-1} & \text{otherwise} \end{cases}$$

binomial_recursive.py

```python
def binomial(n, k):
    if k == 0 or k == n:
        return 1
    return binomial(n - 1, k) + binomial(n - 1, k - 1)
```

recursion tree for `binomial(7,5)`

Dynamic Programming

≡

Remember solutions already found

(memoization)

- Technique sometimes applicable when running time otherwise becomes exponential
- Only applicable if stuff to be remembered is manageable

Binomial Coefficient
Dynamic programming using a dictionary

recursion tree for
`binomial(7,5)`

```
--(7, 5)
   |--(6, 5)
   |   |--(5, 5)
   |    --(5, 4)
   |       |--(4, 4)
   |        --(4, 3)
   |           |--(3, 3)
   |            --(3, 2)
   |               |--(2, 2)
   |                --(2, 1)
   |                   |--(1, 1)
   |                    --(1, 0)
    --(6, 4)
       |--(5, 4)
        --(5, 3)
           |--(4, 3)
            --(4, 2)
               |--(3, 2)
                --(3, 1)
                   |--(2, 1)
                    --(2, 0)
```

binomial_dictionary.py

```python
answers = {}  # answers[(n, k)] = binomial(n, k)

def binomial(n, k):
    if (n, k) not in answers:
        if k == 0 or k == n:
            answer = 1
        else:
            answer = binomial(n - 1, k) + binomial(n - 1, k - 1)
        answers[(n, k)] = answer
    return answers[(n, k)]
```

Python shell

```
> binomial(6, 3)
| 20
> answers
| {(3, 3): 1, (2, 2): 1, (1, 1): 1, (1, 0): 1, (2, 1): 2, (3, 2):
  3, (4, 3): 4, (2, 0): 1, (3, 1): 3, (4, 2): 6, (5, 3): 10, (3,
  0): 1, (4, 1): 4, (5, 2): 10, (6, 3): 20}
```

- Use a dictionary `answers` to store already computed values

reuse value stored in dictionary `answers`

Question – What is the order of the size of the dictionary **answers** after calling **binomial(n, k)** ?

binomial_dictionary.py

```
answers = {}  # answers[(n, k)] = binomial(n, k)

def binomial(n, k):
    if (n, k) not in answers:
        if k == 0 or k == n:
            answer = 1
        else:
            answer = binomial(n - 1, k) + binomial(n - 1, k - 1)
        answers[(n, k)] = answer
    return answers[(n, k)]
```

a) max(n, k)
b) n + k
c) n * k
d) n^k
e) k^n
f) Don't know

Binomial Coefficient
Dynamic programming using decorator

- Use a decorator (@memoize) that implements the functionality of remembering the results of previous function calls

binomial_decorator.py

```
def memoize(f):
    # answers[args] = f(*args)
    answers = {}

    def wrapper(*args):
        if args not in answers:
            answers[args] = f(*args)
        return answers[args]

    return wrapper
```

```
@memoize
def binomial(n, k):
    if k == 0 or k == n:
        return 1
    else:
        return binomial(n - 1, k) + binomial(n - 1, k - 1)
```

www.python-course.eu/python3_memoization.php

binomial_decorator_trace.py	Python shell (without @memoize)	Python shell (with @memoize)

```python
def trace(f):  # decorator to trace recursive calls
    indent = 0

    def wrapper(*args):
        nonlocal indent

        spaces = '|  ' * indent
        arg_str = ', '.join(map(repr, args))
        print(spaces + f'{f.__name__}({arg_str})')
        indent += 1
        result = f(*args)
        indent -= 1
        print(spaces + f'> {result}')
        return result

    return wrapper

def memoize(f):
    answers = {}

    def wrapper(*args):
        if args not in answers:
            answers[args] = f(*args)
        return answers[args]

    wrapper.__name__ = f.__name__ + '_memoize'

    return wrapper

@trace
@memoize
def binomial(n, k):
    if k == 0 or k == n:
        return 1
    return binomial(n - 1, k) + binomial(n-1, k-1)

print(binomial(5, 2))
```

Python shell (without @memoize)

```
binomial(5, 2)
|  binomial(4, 2)
|  |  binomial(3, 2)
|  |  |  binomial(2, 2)
|  |  |  > 1
|  |  |  binomial(2, 1)
|  |  |  |  binomial(1, 1)
|  |  |  |  > 1
|  |  |  |  binomial(1, 0)
|  |  |  |  > 1
|  |  |  > 2
|  |  > 3
|  |  binomial(3, 1)
|  |  |  binomial(2, 1)
|  |  |  |  binomial(1, 1)
|  |  |  |  > 1
|  |  |  |  binomial(1, 0)
|  |  |  |  > 1
|  |  |  > 2
|  |  |  binomial(2, 0)
|  |  |  > 1
|  |  > 3
|  > 6
|  binomial(4, 1)
|  |  binomial(3, 1)
|  |  |  binomial(2, 1)
|  |  |  |  binomial(1, 1)
|  |  |  |  > 1
|  |  |  |  binomial(1, 0)
|  |  |  |  > 1
|  |  |  > 2
|  |  |  binomial(2, 0)
|  |  |  > 1
|  |  > 3
|  |  binomial(3, 0)
|  |  > 1
|  > 4
> 10
10
```

Python shell (with @memoize)

```
binomial_memoize(5, 2)
|  binomial_memoize(4, 2)
|  |  binomial_memoize(3, 2)
|  |  |  binomial_memoize(2, 2)
|  |  |  > 1
|  |  |  binomial_memoize(2, 1)
|  |  |  |  binomial_memoize(1, 1)
|  |  |  |  > 1
|  |  |  |  binomial_memoize(1, 0)
|  |  |  |  > 1
|  |  |  > 2
|  |  > 3
|  |  binomial_memoize(3, 1)
|  |  |  binomial_memoize(2, 1)
|  |  |  > 2
|  |  |  binomial_memoize(2, 0)
|  |  |  > 1
|  |  > 3
|  > 6
|  binomial_memoize(4, 1)
|  |  binomial_memoize(3, 1)
|  |  > 3
|  |  binomial_memoize(3, 0)
|  |  > 1
|  > 4
> 10
10
```

without assigning `wrapper.__name__` the name shown would be `wrapper`

saved recursive calls when using memoization

Dynamic programming using cache decorator

bionomial_cache.py

```
from functools import cache

@cache
def binomial(n, k):
    if k == 0 or k == n:
        return 1
    else:
        return binomial(n - 1, k) + binomial(n - 1, k - 1)
```

- The decorators `@cache` (since Python 3.9) and `@lru_cache(maxsize=None)` in the standard library `functools` supports the same as the decorator `@memoize`
- By default `@lru_cache` at most remembers (caches) 128 previous function calls, always evicting **L**east **R**ecently **U**sed entries from its dictionary
- `functools.cache` has problems when using `sys.setrecursionlimit` (e.g. with Python 3.13 on Windows)

docs.python.org/3/library/functools.html#functools.lru_cache

Subset sum using dynamic programming

- In the subset sum problem (Exercise 13.4) we are given a number `x` and a list of numbers `L`, and want to determine if a subset of `L` has sum `x`

```
L = [3, 7, 2, 11, 13, 4, 8]     x = 22 = 7 + 11 + 4
```

- Let S(v, k) denote if it is possible to achieve value v with a subset of `L`[:k], i.e. S(v, k) = True if and only if a subset of the first k values in `L` has sum v
- S(v, k) can be computed from the recurrence

$$S(v, k) = \begin{cases} \text{True} & \text{if } k = 0 \text{ and } v = 0 \\ \text{False} & \text{if } k = 0 \text{ and } v \neq 0 \\ S(v, k-1) \text{ or } S(v - L[k-1], k-1) & \text{otherwise} \end{cases}$$

Subset sum using dynamic programming

subset_sum_dp.py

```python
def subset_sum(x, L):
    @memoize
    def solve(value, k):
        if k == 0:
            return value == 0
        return solve(value, k - 1) or solve(value - L[k - 1], k - 1)
    return solve(x, len(L))
```

Python shell

```
> subset_sum(11, [2, 3, 8, 11, -1])
| True
> subset_sum(6, [2, 3, 8, 11, -1])
| False
```

Question – What is a bound on the size order of the memoization table if all values are possitive integers?

subset_sum_dp.py

```
def subset_sum(x, L):
    @memoize
    def solve(value, k):
        if k == 0:
            return value == 0
        return solve(value, k-1) or solve(value - L[k-1], k-1)

    return solve(x, len(L))
```

Python shell

```
> subset_sum(11, [2, 3, 8, 11, -1])
| True
> subset_sum(6, [2, 3, 8, 11, -1])
| False
```

a) `len(L)`
b) `sum(L)`
c) `x`
d) $2^{len(L)}$
e) `len(L)`
f) `len(L)*sum(L)`
g) Don't know

Subset sum using dynamic programming

subset_sum_dp.py

```python
def subset_sum_solution(x, L):
    @memoize
    def solve(value, k):
        if k == 0:
            if value == 0:
                return []
            else:
                return None
        solution = solve(value, k - 1)
        if solution != None:
            return solution
        solution = solve(value - L[k - 1], k - 1)
        if solution != None:
            return solution + [L[k - 1]]
        return None
    return solve(x, len(L))
```

Python shell

```
> subset_sum_solution(11, [2, 3, 8, 11, -1])
| [3, 8]
> subset_sum_solution(6, [2, 3, 8, 11, -1])
| None
```

Knapsack problem

	Volume	Value
0	3	6
1	3	7
2	2	8
3	5	9

- Given a **knapsack** with volume **capacity C**, and set of **objects** with different **volumes and value**
- **Objective**: Find a subset of the objects that fits in the knapsack (sum of volume ≤ capacity) and has maximal value
- **Example**: If C = 5 and the volume and weights are given by the table, then the maximal value 15 can be achieved by the 2nd and 3rd object
- Let V(c, k) denote the maximum value achievable by a subset of the first k objects within capacity c

$$V(c, k) = \begin{cases} 0 & \text{if } k = 0 \\ V(c, k-1) & \text{volume}[k-1] > c \\ \max\{V(c, k-1), \text{value}[k-1] + V(c - \text{volume}[k-1], k-1)\} & \text{otherwise} \end{cases}$$

Knapsack – maximum value

knapsack.py

```
def knapsack_value(volume, value, capacity):
    @memoize
    def solve(c, k):  # solve with capacity c and objects 0..k-1
        if k == 0:  # no objects to put in knapsack
            return 0
        v = solve(c, k - 1)  # try without object k-1
        if volume[k - 1] <= c:  # try also with object k-1 if space
            v = max(v, value[k - 1] + solve(c - volume[k - 1], k - 1))
        return v

    return solve(capacity, len(volume))
```

Python shell

```
> volumes = [3, 3, 2, 5]
> values = [6, 7, 8, 9]
> knapsack_value(volumes, values, 5)
| 15
```

Knapsack – maximum value and objects

knapsack.py

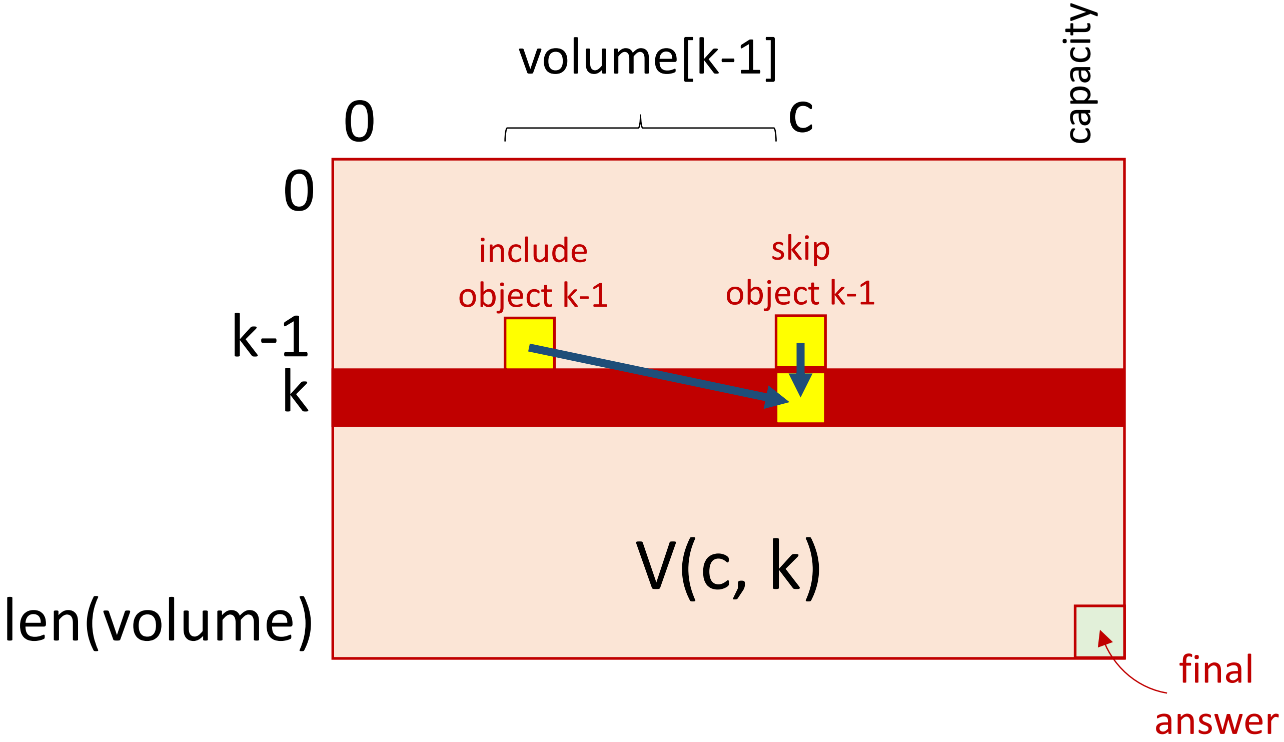
```python
def knapsack(volume, value, capacity):
    @memoize
    def solve(c, k):  # solve with capacity c and objects 0..k-1
        if k == 0:  # no objects to put in knapsack
            return 0, []
        v, solution = solve(c, k - 1)  # try without object k-1
        if volume[k - 1] <= c:  # try also with object k-1 if space
            v2, sol2 = solve(c - volume[k - 1], k - 1)
            v2 = v2 + value[k - 1]
            if v2 > v:
                v = v2
                solution = sol2 + [k - 1]
        return v, solution

    return solve(capacity, len(volume))
```

Python shell

```
> volumes = [3, 3, 2, 5]
> values = [6, 7, 8, 9]
> knapsack(volumes, values, 5)
| (15, [1, 2])
```

Knapsack - Table

$$V(c, k) = \begin{cases} 0 & \text{if } k = 0 \\ V(c, k-1) & \text{value}[k-1] > c \\ \max\{V(c, k-1), \text{value}[k-1] + V(c - \text{volume}[k-1], k-1)\} & \text{otherwise} \end{cases}$$



- systematic fill out table
- only need to remember two rows

Knapsack – Systematic table fill out

knapsack_systematic.py

```
def knapsack(volume, value, capacity):
  ① solutions = [(0, [])] * (capacity + 1)
  ② for obj in range(len(volume)): ⑤
         for c in reversed(range(volume[obj], capacity + 1)): ④
             prev_v, prev_solution = solutions[c - volume[obj]]
             v = value[obj] + prev_v
             if solutions[c][0] < v:
               ③ solutions[c] = v, prev_solution + [obj]

    return solutions[capacity]
```

Python shell

```
> volumes = [3, 3, 2, 5]
> values = [6, 7, 8, 9]
> knapsack(volumes, values, 5)
| (15, [1, 2])
```

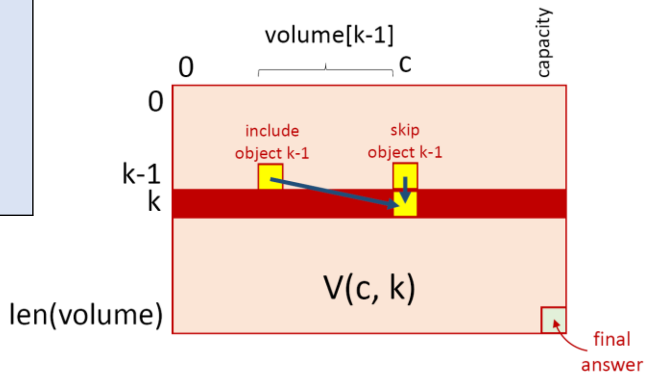
① base case k = 0

② consider each object

③ `solutions[c:]` current row
`solutions[:c]` previous row

④ compute next row right-to-left

⑤ `solutions[:volume[obj]]` unchanged from previous row



Summary

- Dynamic programming is a general approach for recursive problems where one tries to avoid recomputing the same expresions repeatedly

- **Solution 1: Memoization**
  - add dictionary to function to remember previous results
  - decorate with a @memoize decorator
- **Solution 2: Systematic table fill out**
  - can need to compute more values than when using memoization
  - can discard results not needed any longer (reduced memory usage)

Coding competitions and online judges

If you like to practice your coding skills, there are many online "judges" with numerous exercises and where you can upload and test your solutions.

- Project Euler
- Kattis
- CodeForces
- Topcoder

See cs.au.dk/~gerth/code/