Algorithms and Data Structures

Design and complexity (Ch. 8)

Morgan Ericsson

Today

- Algorithm design
- Exponential time
- Computability
- P and NP

Algorithm design

The knapsack problem

- Assume you have some items that each have a value and a weight
- And a bag (knapsack) that can hold at most a certain weight
- Which items should you pick to maximize the value
  - (How much of each)

The fraction version

- Assume that each item can be split into units of weight 1
- In this case, we can use a greedy strategy
- Pick as much as you can of the most valuable item
- If room left, pick from the second most valuable item, …

Implementation

```python
class Item:
  def __init__(self, val:float, weight:int) -> None:
    self.value = val
    self.weight = weight

  def __lt__(self, other) -> bool:
    return self.value / self.weight < other.value / other.

  def __repr__(self) -> str:
    return f'Item({self.value}, {self.weight})'
```

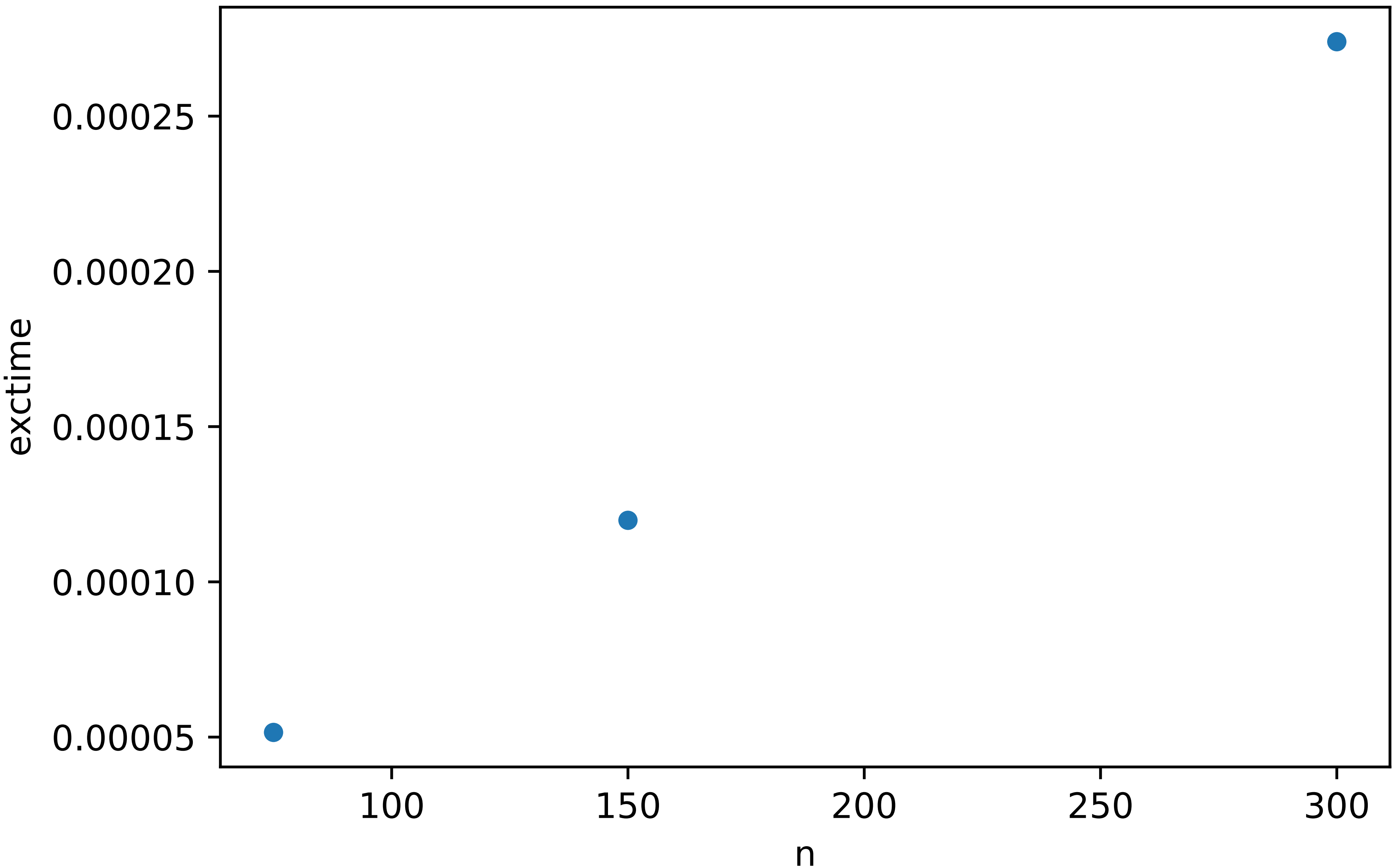
Implementation

```python
class FKnapsack:
  def __init__(self, itms, cap) -> None:
    self.itm = sorted(itms)
    self.cap = cap

  def fill(self) -> float:
    total = 0.0

    while self.cap > 0:
      i = self.itm.pop()
      mn = min(self.cap, i.weight)
      total += i.value / i.weight * mn
      self.cap -= mn

    return total
```

Testing it

```
il = [Item(5, 4), Item(10, 8), Item(3, 3), \
      Item(2, 5), Item(3, 2)]

ks = FKnapsack(il, 11)
assert ks.fill() == 14.25
```

Testing it



0.00025
0.00020
0.00015
0.00010
0.00005
exctime
100
150
200
250
300
n

Greedy algorithms

- A way to deal with optimization problems, where we need to make choices at each step
  - E.g., which order should tasks be scheduled in
- Make the optimal choice at each step
  - Can lead to the optimal solution
  - But not required
- Never re-evaluate previous choices

When does it work?

- Optimal substructure
  - An optimal solution can be constructed from optimal solutions to subproblems
- Greedy choice property
  - There is an optimal solution that is consistent with the greedy choices

Greedy choice property?

- Write any number, n, as $a \cdot 10 + b \cdot 5 + c \cdot 1$
  - $43 = 4 \cdot 10 + 0 \cdot 5 + 3 \cdot 1$
  - $99 = 9 \cdot 10 + 1 \cdot 5 + 4 \cdot 1$
- As long as $n > 10$, pick 10 and set $n \leftarrow n - 10$
- As long as $10 > n > 5$, pick 5 and set $n \leftarrow n - 5$
- As long as $5 > n > 0$, pick 1 and set $n \leftarrow n - 1$

Greedy choice property?

- Write any number, n, as $a \cdot 6 + b \cdot 5 + c \cdot 1$
  - $11 = 1 \cdot 6 + 1 \cdot 5 + 0 \cdot 1$
  - $10 = 1 \cdot 6 + 0 \cdot 5 + 4 \cdot 1$
- Greedy choice provides non-optimal solution
- (but correct)

Greedy choice property?

- Write any number, n, as $a \cdot 6 + b \cdot 5 + c \cdot 2$
  - $11 = 1 \cdot 6 + 1 \cdot 5 + 0 \cdot 2$
  - $7 \neq 1 \cdot 6 + 0 \cdot 5 + 0 \cdot 2$
- Greedy solution is not correct

Greedy algorithms

- There are problems that can be solved with greedy algorithms
  - often with low complexity
  - fraction Knapsack $O(n \log n)$ (from sorting)
- But it is easy to fool yourself!

The binary version

- What if we cannot split the items?
- We cannot use a greedy approach
  - Why? No greedy choice property!
- Highest value or value per weight does not guarantee an optimal solution
- Try all combinations?

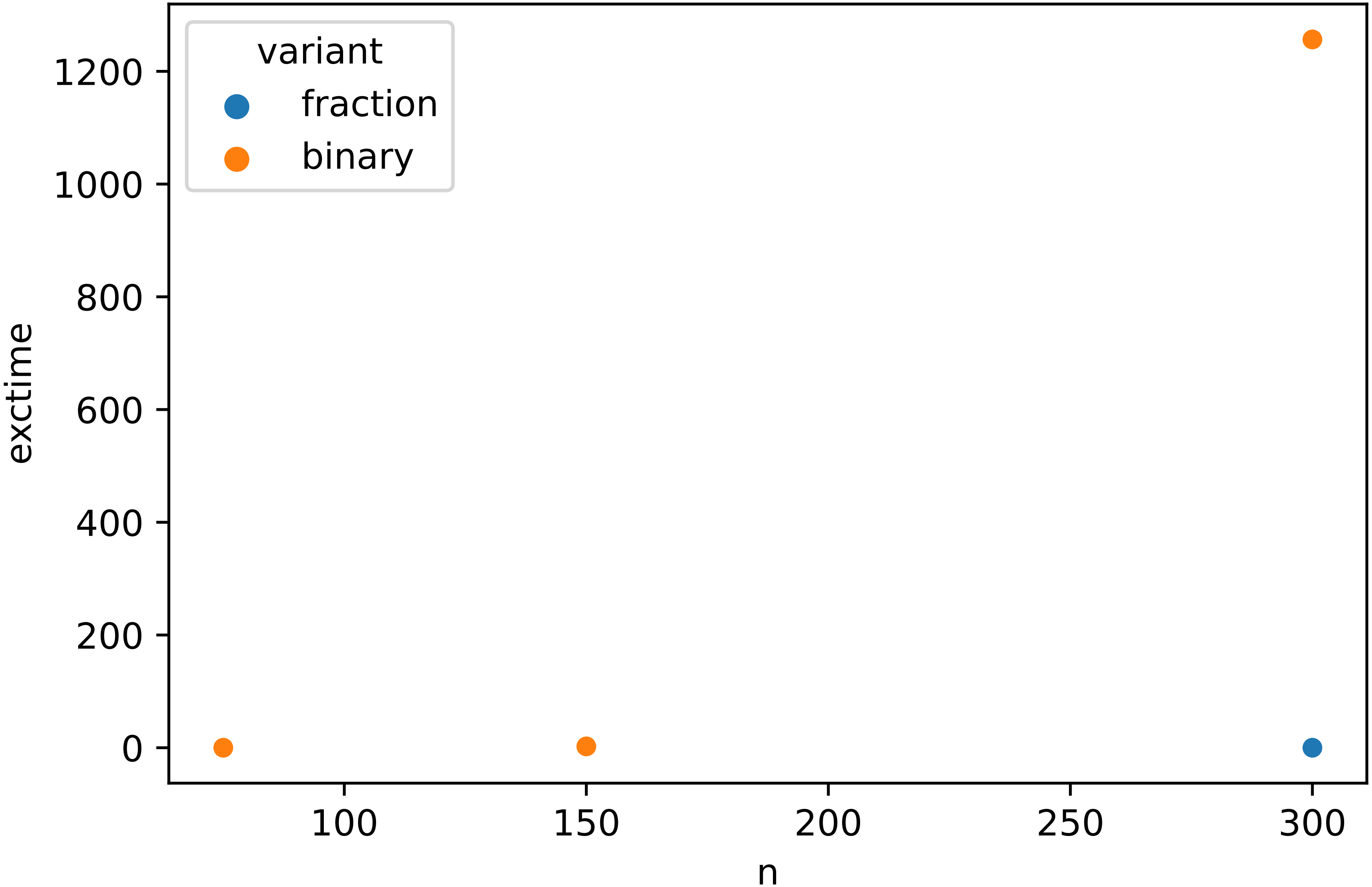
Implementation

```python
class BKnapsack:
  def __init__(self, itms, cap) -> None:
    self.itm = itms
    self.cap = cap

  def fill(self) -> float:
    return self._fill(self.cap, len(self.itm))

  def _fill(self, W, n) -> float:
    if n == 0 or W == 0.0:
      return 0.0

    if self.itm[n-1].weight > W:
      return self._fill(W, n-1)
    else:
      return max(self.itm[n-1].value + \
          self._fill(W - self.itm[n - 1].weight, n - 1), \
          self._fill(W, n - 1))
```

Testing it

```
il = [Item(5, 4), Item(10, 8), Item(3, 3), \
      Item(2, 5), Item(3, 2)]

ks = BKnapsack(il, 11)
assert ks.fill() == 13.0
```

Any difference?



Any difference?

variant	n	exctime
fraction	75	0.00005
fraction	150	0.00012
fraction	300	0.00027
binary	75	0.01532
binary	150	2.38345
binary	300	1256.56600

How can we fix it?

- We evaluate all combinations
- An item can either be picked or not picked
- If two items,
  - $\underbrace{\emptyset, \{i_1\}, \{i_2\}, \{i_1, i_2\}}_{2^2}$
- So, for n items, $O(2^n)$

How can we fix it?

- The brute force approach evaluates the same problem several times
  - Overlapping subproblems
- What if we store the results and reuse them?

Fibonacci

```python
def fib(n):
  print(f'fib({n}) ', end='')
  if n <= 1:
    return n
  return fib(n - 2) + fib(n - 1)
```

Fibonacci

```
assert fib(5) == 5
```

```
fib(5) fib(3) fib(1) fib(2) fib(0) fib(1) fib(4) fib(2)
fib(0) fib(1) fib(3) fib(1) fib(2) fib(0) fib(1)
```

Caching Fibonacci (memoization)

```python
class TDFibonacci:
  def __init__(self, n:int):
    self.c = {0:0, 1:1}
    self._fib(n)

  def _fib(self, n:int):
    if n in self.c:
      return self.c[n]
    print(f'fib({n}) ', end='')
    self.c[n] = self._fib(n - 2) + self._fib(n - 1)
    return self.c[n]
```

Fibonacci

```
f = TDFibonacci(5)
assert f.c[5] == 5
```

`fib(5) fib(3) fib(2) fib(4)`

Caching Fibonacci (tabulation)

```python
class BUFibonacci:
  def __init__(self, n:int):
    self.c = np.zeros(n+1, dtype=int)
    self.c[0:2] = [0, 1]
    self._fib(5)

  def _fib(self, n:int):
    for i in range(2, n + 1):
      print(f'fib({i}) ', end='')
      self.c[i] = self.c[i - 2] + self.c[i - 1]
```

Fibonacci

```
f = BUFibonacci(5)
assert f.c[5] == 5
```

`fib(2) fib(3) fib(4) fib(5)`

Dynamic programming

- If a problem has optimal substructure and overlapping subproblems
- We can use dynamic programming
  - Memoization (Top-down, recursion)
  - Tabulation (Botton-up, iterative)

Divide and conquer?

- We have seen divide and conquer multiple times
  - Quicksort, mergesort, ...
- Optimal substructure
- Non-overlapping subproblems

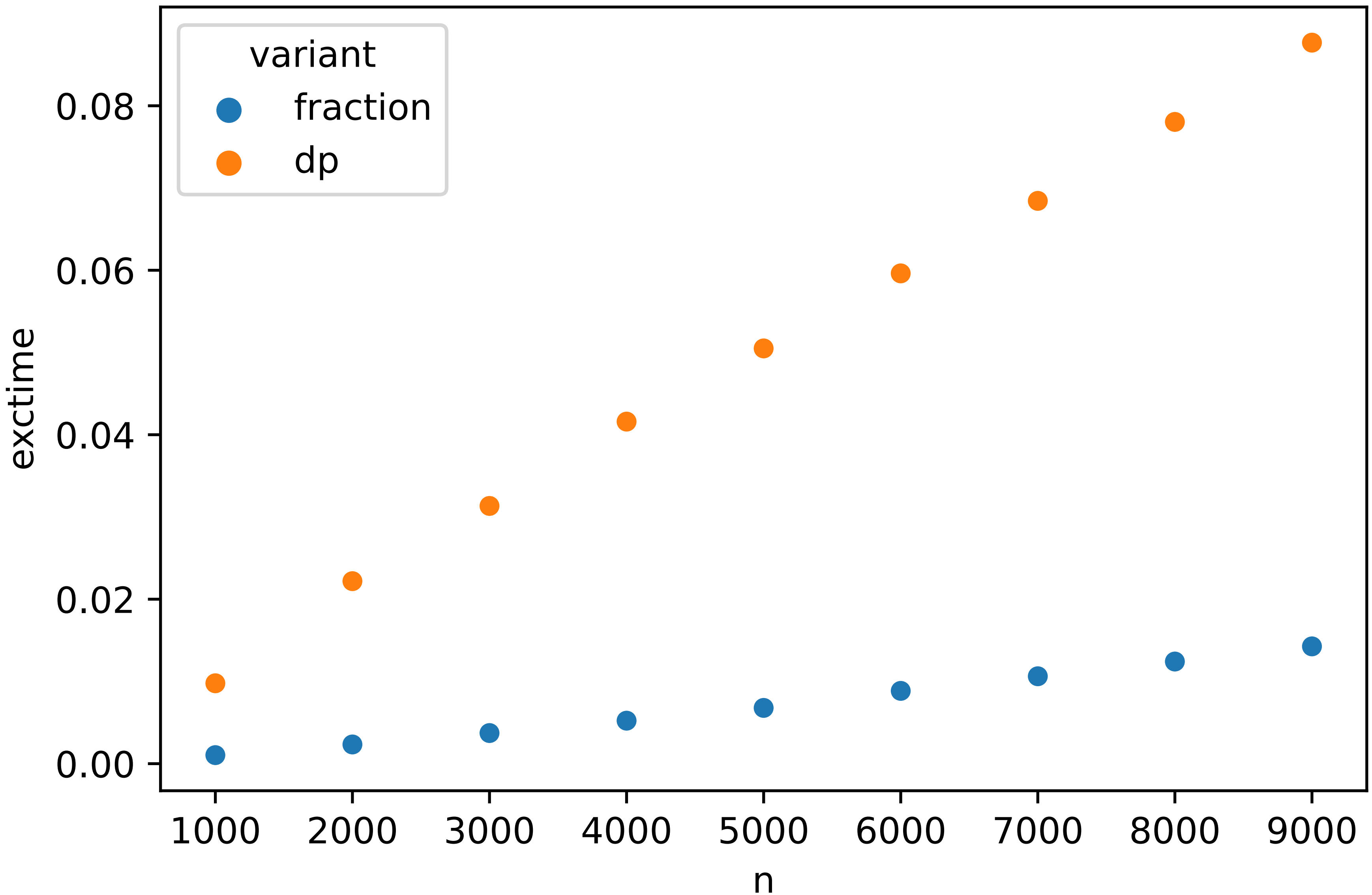
Implementation

```python
import numpy as np

class DPBKnapsack:
  def __init__(self, itms, cap) -> None:
    self.itm = itms
    self.cap = cap
    self.dp = np.zeros(self.cap + 1, dtype=float)

  def fill(self) -> float:
    for i in range(1, len(self.itm) + 1):
      for w in range(self.cap, 0, -1):
        if self.itm[i-1].weight <= w:
          self.dp[w] = max(self.dp[w], \
                           self.dp[w - self.itm[i - 1].weight] + \
                           self.itm[i - 1].value)

    return self.dp[self.cap]
```

Testing it

```
il = [Item(5, 4), Item(10, 8), Item(3, 3), \
        Item(2, 5), Item(3, 2)]

ks = DPBKnapsack(il, 11)
assert ks.fill() == 13.0
```

Better?


variant
fraction
dp
exctime
0.08
0.06
0.04
0.02
0.00
1000
2000
3000
4000
5000
6000
7000
8000
9000
n

P and NP

Disclaimer

- Some of this will be simplified to make it easier to understand
- The formal definitions can be difficult to understand without a course in languages and automata theory

P

- An algorithm is of polynomial time if its running time is bounded by a polynomial expression
- $T(n) = O(n^k)$, where k is some positive constant
  - $O(n^2)$
  - Most of the algorithms we have studied, including $O(\log n)$
- Problems with polynomial-time algorithms belong to the complexity class **P**

The class P

- Decision problems that can be solved on a deterministic Turing machine in polynomial time
- A decision problem is a problem that can be answered by yes or no
  - Given x and y, does x evenly divide y?
- Problems in P are efficiently solvable or tractable
  - Rule of thumb, not entirely true (e.g., $O(n^{50})$)

Turing machine

- A mathematical model of computation
- Abstract machine that manipulates symbols on a tape
- Capable of implementing any algorithm (Church-Turing thesis)

Turing machine

- Formally $M = \langle Q, \Gamma, b, \Sigma, \delta, q_0, F \rangle$
  - Γ is a set of symbols (alphabet)
  - b is the blank symbol
  - $\Sigma \subseteq \Gamma - \{b\}$
  - Q is a set of states
  - $q_0 \in Q$ is the initial state
  - $F \subseteq Q$ is the accepting states
  - $\delta : (Q - F) \times \Gamma \rightharpoonup Q \times \Gamma \times \{L, R\}$

Example

- A busy beaver
  - $Q = \{A, B, C, HALT\}$
  - $\Gamma = \{0, 1\}$
  - $b = 0$
  - $\Sigma = \{1\}$
  - $q_0 = A$
  - $F = \{HALT\}$

Example

	A	B	C
0	1RB	0RC	1LC
1	1RH	1RB	1LA

» E.g., $\delta = A \times 0 \rightharpoonup B \times 1 \times R, \ldots$

Example

```
0 0 0 0 0 0      1 1 1 1 0 0      1 1 1 1 1 1
  A                B                    C
0 1 0 0 0 0      1 1 1 1 0 0      1 1 1 1 1 1
    B                B                A
0 1 0 0 0 0      1 1 1 1 0 0      1 1 1 1 1 1
      C                B                H
0 1 0 1 0 0      1 1 1 1 0 0
    C                    B
0 1 1 1 0 0      1 1 1 1 0 0
  C                        C
0 1 1 1 0 0      1 1 1 1 0 1
A                        C
```

Universal Turing machine (UTM)

- A Universal Turing machine can simulat any turing machine on any input
- Reads the machine to be simulated and the input from tape
- Birth of the stored program concept

Turing completeness

- A system of data-manipulation rules is Turing complete if it can be used to simulate any Turing machine
- Any general purpose programming language is Turing complete
- And some other things, such as Cities: Skylines and chemical reaction networks
- Regular expressions are not

The class P

- Decision problems that can be solved on a deterministic Turing machine in polynomial time
- A decision problem is a problem that can be answered by yes or no
  - Given x and y, does x evenly divide y?
- Problems in P are efficiently solvable or tractable
  - Rule of thumb, not entirely true (e.g., $O(n^{50})$)

The class NP

- Decision problems that can be solved on a nondeterministic Turing machine in polynomial time
- If yes, the proof can be verified in polynomial time
- Problems in **NP** are intractable
  - Again, also rule of thumb
- Remember the N in NP is for nondeterministic turing machine

???

- Consider the travelling salesperson problem (TSP)
- Given a list of cities and distances between each pair, is there a route that visits each city and returns to the origin city of lenght k or less?
- Similarities with binary knapsack
- But, it is easy to see that a possible solution is verifiable in polynomial time, simply compute the length of the path and check if less than or equal to k

???

- The trick is the nondeterminism
- Think of this as a “forking” Turing machine
- At each step, all possible guesses are tried
- Explores the exponential tree in parallel
- A single Turing machine that always guesses correctly

Problems in NP

- Binary knapsack
- TSP
- Hamiltonian paths
- ...

Not that easy...

» NP

» NP hard

» NP complete

Reductions

- A reduction is an algorithm for transforming one problem into another
- Can be used to show that one problem is at least as difficult as another
- Consider finding the median of a list of integers
  - Can be reduced to sorting and picking the middle

NP complete

- A problem, C is NP complete if
  - C is in NP
  - Every problem in NP can be reduced to C in polynomial time
- Contains the hardest problems in NP
- Knapsack, Hamiltonian paths, TSP, SAT, ... are NP complete

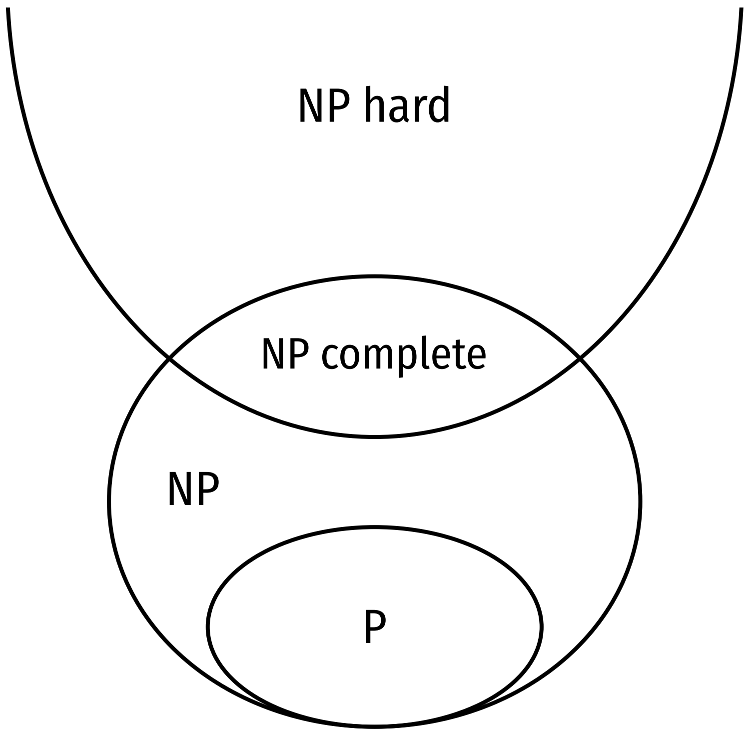
NP hard

» Problems that are at least as hard as problems in NP

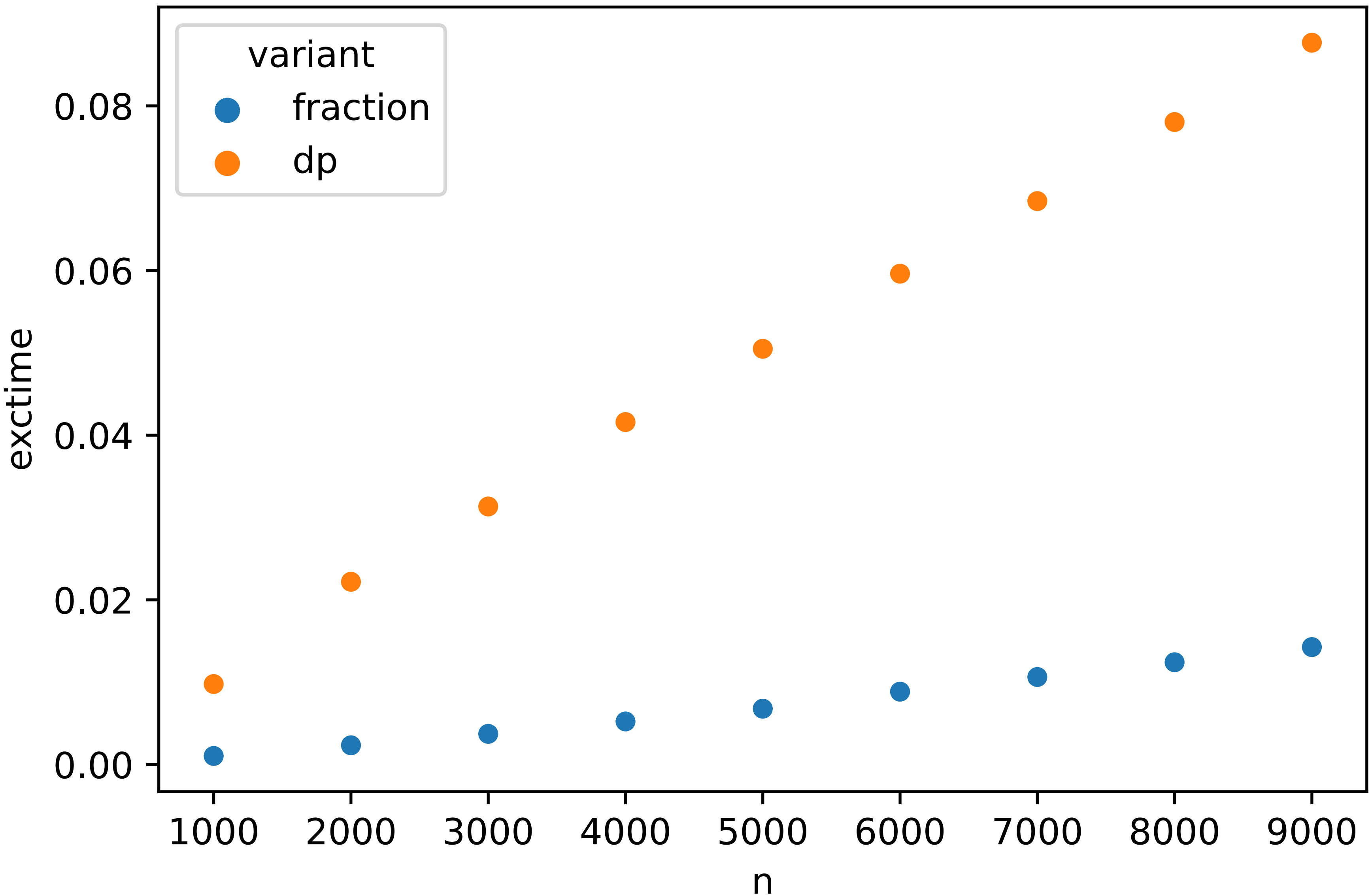
» Does not have to be in NP

» But problems in NP should be reducible to the NP hard problem

The classes (assuming P!=NP)



But?


variant
fraction
dp
0.08
0.06
0.04
0.02
0.00
exctime
1000
2000
3000
4000
5000
6000
7000
8000
9000
n

Not P (but almost...)