85301 – Algorithms and Data Structures in Biology

Lab 4 – Enumerating Sets, and Too Big Inputs

Enrico Malizia and Riccardo Treglia

DISI
University of Bologna, Italy

2nd semester, 2022/23

(parts of these slides are based on material by Prof. Ugo Dal Lago)

Enumerating Sets in Python

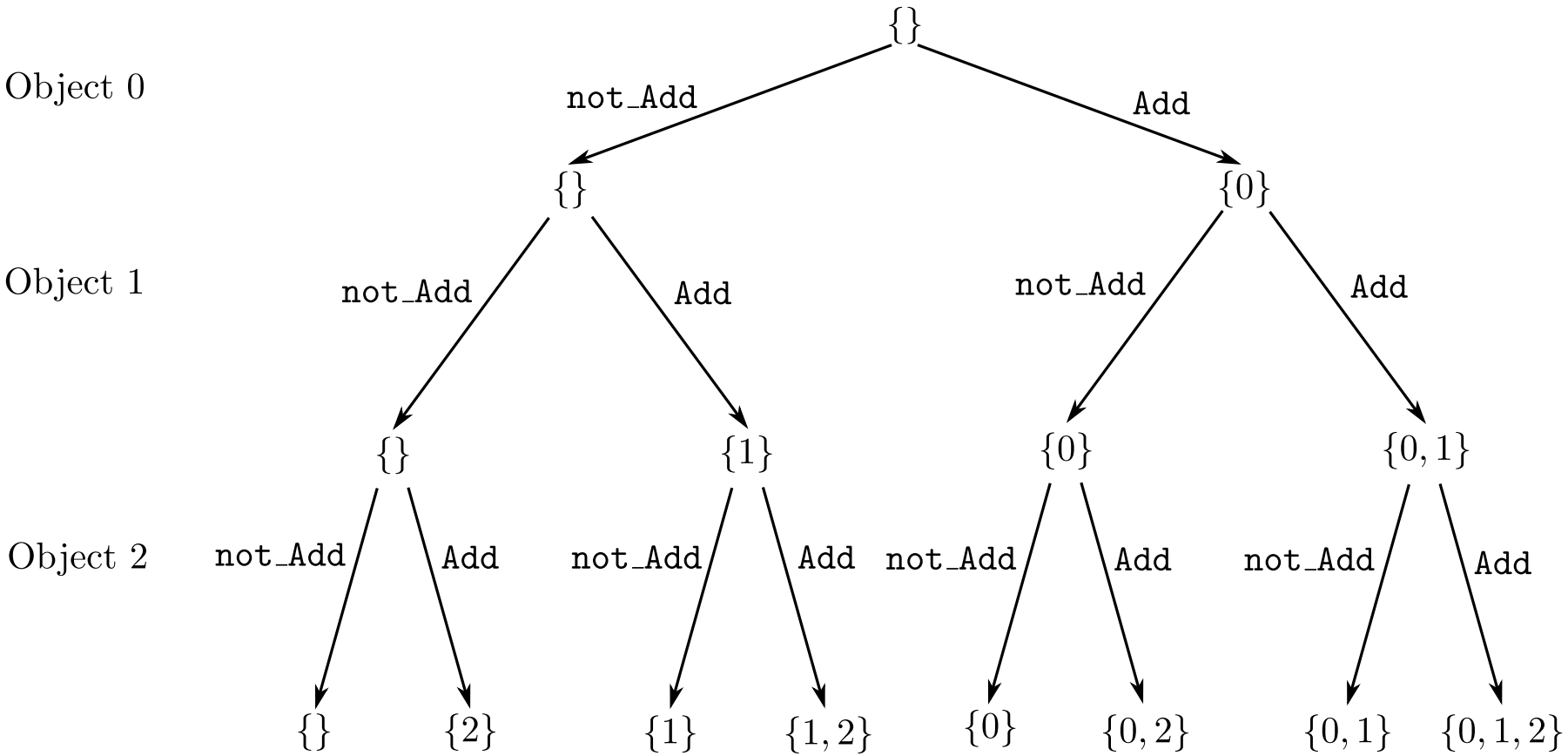
- It is often the case, particularly when designing exhaustive search algorithms, that our pseudocode contains a statement like

  $$\texttt{foreach } x \texttt{ such that } P$$

  where P typically is a property of some form, which is satisfied by *finitely many* xs
- Sometimes, P is such that the aforementioned loop can directly be implemented in `Python`:
  - *Example*: $1 \leq x \leq n$, $x \in \texttt{L}$, ...
- In many other cases, there is no direct way to implement P, unless you go via `itertools`:
  - *Example*: x is a list of Booleans of length n; x is a list of length n of natural numbers smaller or equal to m; ...
  - In these cases, one possibility consists in generating the list of all possible value of x, and then iterate over it with a `for` loop, as usual

Enumerating Subsets

- Suppose that we want to generate all the subsets of a set of n objects
- We can look at a tree structure to see how to accomplish the task
  - In particular, a binary tree of depth n



Enumerating Subsets: Iterative Version

```
def subsetsIter(n):
  result = [set()]     # A list to build layer by layer the final result
                       # We start with the empty set
  for i in range(n):   # We cycle through all the layers of the tree
    temp = []          # A temporary list where to store the results
                       # for the current layer
    for c in result:   # For each combination 'c' of the previous layer
      c1 = c.copy()    # We clone and do not add anything
      temp.append(c1)
      c2 = c.copy()    # We clone and ...
      c2.add(i)        # add element i
      temp.append(c2)
    result = temp      # 'result' is updated with the results for
                       # the current layer
  return result        # We return the final result
```

Enumerating Subsets: Recursive Version

```
def subsetsRecur(n):
  if n == 0:
    return [set()]      # As base of recursion we return the empty set
  else:
    result = []         # A list where to build the final result
    for c in subsetsRecur(n-1):  # We recursively generate the
                                 # subsets for n-1 elements
                                 # and we cycle trough them
      c1 = c.copy()              # We clone and do not add anything
      result.append(c1)
      c2 = c.copy()              # We clone and ...
      c2.add(n-1)                # add the last element (remember that
                                 # the first element is 0)
      result.append(c2)
    return result                # We return the final result
```

Enumerating Subsets

- A disadvantage of the previous methods is that they generate the entire list of subsets, which occupies exponential space
- If there is no need to have access to all the subsets at a given time, there are ways to avoid to occupy all this space
  - One might use the concept of `Python generators`; or
  - One might use a different representation: for example, a list of Booleans telling whether an element is present (`True`) or not (`False`); all the possible lists of Booleans can be "seen" by using the metaphor of the odometer (as we saw in the lectures)

Exponential Blowups

- Exhaustive search algorithms work in *exponential time* in the worst case, although this can sometime be mitigated
- As soon as n is bigger than a few dozens, 2^n is a very high number
  - Even if each instruction is executed in a millionth of a second, the total time it takes to solve the corresponding problem instance is *huge*
- It is thus normal that the time it takes to execute an exhaustive search algorithm becomes impractically large even when n is relatively small
  - If this happens to you in your assignment, simply *rescale the values* of n that we have provided, justifying this in your report