On February 16, 2022 (14:06:33), we had submissions for the following questions: 1, 2 — Details (/agns/CSE102/TD02/2021/uploads/:my/)

CSE 102 – Tutorial 2 – Recursion & generators

Discussion with colleagues in tutorials is encouraged and *not* considered to be plagiarism. The purpose is to learn, rather than to be assessed, and discussing with colleagues is a great way to learn. However, you must still produce and submit your own work and avoid plain copying, as this does not aid learning in any way and will leave you unprepared for the assessed assignments.

Sections tagged with a green marginal line are **mandatory**.

Sections tagged with an orange marginal line are more advanced, and are therefore considered *optional*. They will help you deepen your understanding of the material in the course. You may skip these sections on a first pass through the tutorial, and come back to them after finishing the mandatory exercises.

When submitting a solution, the system will do some sanity checks on your files and run some automated tests on your solutions. A green tick tells you that your file compiles and has passed all of the automated tests. Passing all the tests provides good evidence that your solutions are correct, although it might not guarantee it (typically we would need to run infinitely many tests!), and so you should always strive to understand *why* your code works and not just rely on the automated tests.

A red tick tells you either that your file is badly broken (e.g., a syntax error) or that your solutions fail one or more of the automated tests. You can see the result of the tests by clicking on "See my uploads" in the upper-left menu, and then clicking on the right arrow button next to each file you submitted.

Recursion

We expect you to write your solution to these problems in a file named `recursion.py` and to upload it using the form below. You can resubmit your solution as many times as you like.

Last submission: February 15, 2022 (11:56:35)

Choose | Choose one or more files... | Submit

☑ Reuse files from previous submissions

0.00 / 0

Submit on behalf of

Type to search...

Force date

Expected files: recursion.py

Subsets and permutations

1. The *binomial coefficient* $\binom{n}{k}$ counts the number of ways of choosing a k-element subset from an n-element set. Write a recursive function `binom` that given two non-negative integers n and k, computes $\binom{n}{k}$ based on the recurrence relation $\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}$ with boundary conditions $\binom{n}{0} = 1$ and $\binom{n}{k} = 0$ for $k > n$.

   Here is a test you can try with the expected output (generating OEIS sequence A000984 (https://oeis.org/A000984)):

   ```
   >>> print([binom(2*n, n) for n in range(10)])
   [1, 2, 6, 20, 70, 252, 924, 3432, 12870, 48620]
   ```

2. Python has built-in set types (https://docs.python.org/3/library/stdtypes.html#set-types-set-frozenset), which allow you to build sets using "curly braces notation" and satisfy equations like `{1,3,5} == {3,5,1} == {1,1,3,3,5,5}`. But sets can also be represented in other ways in Python, just like in other programming languages. For example, one general way of representing a set is as a list of elements that is sorted and does not contain duplicates, e.g., `[1,3,5]` is a valid representation of the set of odd numbers less than 6, but `[3,5,1]` is not (since it is unsorted) and neither is `[1,1,3,3,5,5]` (since it contains duplicates).

   Write a recursive function `choose` that takes as input a set S and a non-negative integer k, and returns a list of all the k-element subsets of S. For this question, you should *not* use built-in Python sets, rather you should assume that the input set S is represented as a duplicate-free sorted list and maintain this invariant for the output, i.e., each of the k-element subsets in the output list should be represented as a duplicate-free sorted list. On the other hand, you are allowed to return the subsets in an arbitrary order.

   Hint: The recursive structure of `choose` should exactly mirror the recursive structure of `binom`! You can start by copying in your code for `binom` and then modifying the cases appropriately...

   For example, `choose([1,3,5,7], 3)` should return the list of lists

   ```
   [[1, 3, 5], [1, 3, 7], [1, 5, 7], [3, 5, 7]]
   ```

   or some rearrangement thereof.

   As an additional test, we suggest to compute

   ```
   >>> [sum(len(choose(list(range(n)),k)) for k in range(n+1)) for n in range(10)]
   ```

   and compare it with what it should be. By the way, what should it be?

3. In the lecture, you saw a recursive implementation of the factorial function

   ```
   def factorial(n):
       if n == 0:
           return 1
       return n * factorial(n-1)
   ```

0.00 / 0

which counts the number of ways ($n!$) of rearranging a list of n distinct elements. Now write a recursive function `permutations` that takes as input a list L and returns a list of all of the permutations of L. You can assume that all of the values in L are distinct, and you can return the different permutations of L in an arbitrary order.

For example, `permutations([1,2,3])` should return the list of lists

```
[[3, 2, 1], [2, 3, 1], [2, 1, 3], [3, 1, 2], [1, 3, 2], [1, 2, 3]]
```

or some rearrangement thereof.

Hint: If you have computed all permutations of n elements, then the permutations of $n + 1$ elements can be obtained by inserting the $n + 1$-st element in all possible positions.

4. Implement `choose` and `permutations` without using recursion.

5. A multiset is a set in which elements may appear with some multiplicity. We can represent multisets as sorted lists of values possibly containing duplicates. For example, the multiset containing two copies of 1, three of 4, and one of 5 may be represented as `[1, 1, 4, 4, 4, 5]`. Write a recursive function `multichoose` that takes as input a multiset S represented as a sorted list of values and a non-negative integer k and returns the list of all its k-element submultisets.

   For example, `multichoose([1, 1, 4, 4, 4, 5], 2)` should return the list of lists

```
[[1, 1], [1, 4], [1, 5], [4, 4], [4, 5]]
```

   or some rearrangement thereof.

Anger management and freelancing

6. Every day, Olivier joins the Zoom session at his office late or on time, but his boss Emmanuel becomes angry if Olivier joins late two days in a row. Help Olivier by writing a recursive function `not_angry` that takes a non-negative integer n as input and computes in how many ways Olivier can join the meetings over n days so that Emmanuel will not get angry. For example, `not_angry(3) = 5` and `not_angry(4) = 8`.

   Hint: To compute `not_angry(n)`, analyze the possibilities after what happens on the first day (Olivier either joins late or is on time).

7. Emmanuel has become more relaxed, and now can tolerate Olivier joining late up to k days in a row, for some $k > 0$. Write a recursive function `not_so_angry(k, n)` to compute the number of ways of keeping Emmanuel calm during n days. For example, `not_so_angry(3, 4) = 15`.

   Hint: To compute `not_so_angry(k, n)`, analyze the possibilities for the initial days of the n-day sequence, considering for how many days in a row Olivier may join late.

8. Finally Olivier showed up late one too many times in a row and was fired. He decided to join a startup whose business model is based on matching freelancers with tasks to be done, and again he is asking you for help.

   You are given an $n \times n$ array array `prices`, where `prices[i][j]` is the amount of money the i-th freelancer wants for the j-th task. You need to assign each freelancer a task so that the total amount of money spent is as small as possible. Write a function `find_assignment` that takes the prices array as an input and returns a pair `(price, assign)` such that `assign[i]` is the number of the task assigned to the i-th freelancer in an optimal assignment, and `price` is the smallest amount of money to pay.

   Here are some tests:

0.00 / 0

```
>>> find_assignment([[1, 4], [5, 9]])
(9, [1, 0])
>>> find_assignment([[1, 1], [1, 2]])
(2, [1, 0])
```

We (and Olivier) expect you to implement a recursive solution that will decrease the number of freelancers by one each time. If you are curious about more efficient ways to solve this problem, you might want to take a look at the very elegant Hungarian algorithm (https://en.wikipedia.org/wiki/Hungarian_algorithm).

Generators

We expect you to write your solution to these problems in a file named `generators.py` and to upload it using the form below. You can resubmit your solution as many times as you like.

Last submission: February 15, 2022 (11:34:07)

Choose | Choose one or more files... | Submit

☑ Reuse files from previous submissions

Submit on behalf of

Type to search...

Force date

Expected files: generators.py

Basic generators

9. Write a function `fibs()` that produces a generator for the Fibonacci sequence $F_0, F_1, \ldots$ defined by:

$$F_0 = 0 \qquad F_1 = 1 \qquad F_{n+2} = F_n + F_{n+1} \text{ for } n \in \mathbb{N}.$$

Make sure that the number of operations between successive yields is constant, i.e., each individual item of the sequence is yielded in constant time.

You can test that the generator produces correct values for the first ten numbers in the sequence by running the following code:

```
>>> g = fibs()
>>> [next(g) for _ in range(10)]
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

10. Write a function `prefix_sums(k)` that produces a generator for the infinite sequence of sums k, $k + (k + 1)$, $k + (k + 1) + (k + 2)$, etc. That is, the nth element of the sequence is:

$$P_n = \sum_{i=0}^{n} (k + i).$$

```
>>> g = prefix_sums(10)
>>> [next(g) for _ in range(10)]
[10, 21, 33, 46, 60, 75, 91, 108, 126, 145]
```

0.00 / 0

11. Write a function `interleave(g1, g2)` that produces a generator that yields values alternately from the generators `g1` and `g2`. You may assume that both `g1` and `g2` yield values forever, i.e., that they generate infinite sequences.

```
>>> g = interleave(prefix_sums(10), fibs())
>>> [next(g) for _ in range(10)]
[10, 0, 21, 1, 33, 1, 46, 2, 60, 3]
```

More subsets and permutations

12. Write a function `choose_gen(S, k)` that produces a generator that yields all the k-element subsets of a set S (represented as a sorted list of values without duplicates) in some arbitrary order. For small sets, you can test the result against the `choose` function you wrote above. However, the generator should produce the subsets one-by-one and not store all of them simultaneously. For example, you should be able to use `choose_gen(list(range(100)), 50)` to generate 50-element subsets of a 100-element set, without having to store all the $\binom{100}{50} \approx 10^{29}$ possibilities.

```
>>> [s for s in choose_gen([1,3,5,7], 3)]
[[1, 3, 5], [1, 3, 7], [1, 5, 7], [3, 5, 7]]
>>> g = choose_gen(list(range(100)), 50)
>>> xs = [next(g) for _ in range(1000)]
>>> next(g)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22,
        23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41,
        42, 43, 44, 45, 46, 47, 73, 99]
```

13. Write a function `subsets_gen(S)` that produces a generator that yields all subsets of S, in increasing order of size.

```
>>> [s for s in subsets_gen([1,2,3])]
[[], [1], [2], [3], [1, 2], [1, 3], [2, 3], [1, 2, 3]]
>>> g = subsets_gen(list(range(100)))
>>> xs = [next(g) for _ in range(1000)]
>>> next(g)
[9, 54]
```

14. The same thing but for permutations, i.e., write a function `permutations_gen` that given as input a list L of distinct values, returns a generator yielding all permutations of L. Test your generator on small lists against the `permutations` function you implemented above, and also test it on larger lists. Again, you should be able to easily generate permutations of a list of length 100, without having to store all $100! \approx 10^{158}$ possibilities.

Sieve of Eratosthenes

We are here interested in the sieve of Eratosthenes (https://en.wikipedia.org/wiki/Sieve_of_Eratosthenes) algorithm for enumerating primes. The following series of problems will guide you towards implementing the sieve using generators.

15. Write a generator `repeat(x)` which repeatedly yields the value `x`.

16. Write a function `cross(n, g)` which crosses out every n-th element of the sequence generated by `g` and replaces it with the value `False`.

```
>>> g = cross(2, repeat(True))
>>> [next(g) for _ in range(10)]
[True, False, True, False, True, False, True, False, True, False]
```

0.00 / 0

17. Write a generator `sieve()` generating an infinite stream of booleans $S_0, S_1, S_2, \ldots$, such that S_n is

`True` iff n is prime. You should use the sieve of Eratosthenes algorithm, making repeated calls to `cross` to cross out the multiples of prime numbers.

```
>>> g = sieve()
>>> [next(g) for _ in range(10)]
[False, False, True, True, False, True, False, True, False, False]
```

18. Write a function `prime_pi(n)` which uses your generator above to compute $\pi(n)$, i.e., the number of primes $\leq n$. You can test that your implementation is correct by comparing the values to OEIS sequence A000720 (https://oeis.org/A000720/b000720.txt). How high can you go? In terms of efficiency, how does this version of $\pi(n)$ compare to the versions you implemented for CSE-103, and why might that be the case?

0.00 / 0