CCPS 721 Prolog Problems

Ilkka Kokkarinen

Chang School of Continuing Education
Toronto Metropolitan University

Version of January 18, 2024

This document contains the specifications for the Prolog programming labs for the course **CCPS 721 Artificial Intelligence I**, as prepared and taught by Ilkka Kokkarinen for Chang School of Continuing Education at Toronto Metropolitan University, Canada.

This document contains the specifications for the Prolog programming problems in the course CCPS 721 Artificial Intelligence I, as prepared and taught by Ilkka Kokkarinen for Chang School of Continuing Education, Ryerson University, Toronto. This document contains **twenty-four** different problems for students to freely choose which **up to ten problems** they wish to solve and submit for grading in this course. Each of these problems is worth the same two points in the Prolog labs component of the grading scheme of CCPS 721.

You can use these example cases to quickly verify that the rules for your predicates are at least on the right track. See the page "Notation of Predicate Descriptions" about the conventions used in this specification for mode indicators.

Your Prolog rules may freely use all of the built-in predicates of SWI-Prolog and its CLP(FD) extension for constraint logic programming in finite integer domains. For predicates that must succeed with multiple solutions one at the time, these automated tests are designed to accept your solutions regardless of the order in which your logic generates them.

These problems are adapted from the instructor's own compilation of Python problems for CCPS 109 to illustrate various techniques to solve useful and interesting computational problems, instead of the old cliches of "Aunt is a female sibling of the parent..." Students who have previously taken this author's CCPS 109 and solved these same programming exercises will surely enjoy comparing their respective solutions to understand the difference in spirit in programming in these very different languages. The same goes for double for those problems that were also included in the collection of Java problems for this author's second course CCPS 209!

The "upside-down" nature of thinking and coding in Prolog compared to imperative programming languages can be difficult to adapt to, even for experienced Python programmers. One way to not get stuck with some problem before you even started solving it would be to first solve the problem recursively in Python (this intermediate code doesn't have to be fancy or optimally efficient, as long as it works correctly and uses only some simple operations on top of recursion), and translate that solution into equivalent tail recursive Prolog formulas using the conversion techniques illustrated in the lecture example `tailrecdemo.pl`.

Write your predicates into the file `tester721.pl` as indicated by the comments in that file. For each predicate that you complete, add the call to the corresponding tester predicate in the body of the rule of the predicate `all/0`. This allows the instructor to immediately see how many predicates you claim to have successfully completed, and verify this claim by running the tests by calling `all`.

only_odd_digits(+N)

Succeeds if the base ten representation of the positive integer `N` contains only odd digits. To organize the checking activity inside this recursion, note how the integer arithmetic operators `div` and `mod` in Prolog can be used to extract the last digit from an integer.

```
?- only_odd_digits(0).
false.

?- only_odd_digits(135797531).
true.

?- only_odd_digits(7755334119955).
false.

?- findall(N, (between(1, 1000, N), only_odd_digits(N)), _L),
length(_L, L).
L = 155.

?- findall(N, (between(1, 100000, N), only_odd_digits(N)), _L),
length(_L, L).
L = 3905.
```

domino_cycle(+C)

An individual domino tile is represented as term `(A, B)` where `A` and `B` are its counts of **pips** that the tile starts and ends with when it has been placed on the board. These pips must be integers between 1 and 6, inclusive which is probably best enforced with the predicate `between/3`. A **domino cycle** is a list of domino tiles whose every tile ends with the pips that its successor tile starts with. To complete the cycle, the last tile must end with the pips that the first tile starts with.

```
?- domino_cycle([(2, X)]).
X = 2.

?- domino_cycle([(4, 1), (1, 7), (7, 2)]).
false.

?- domino_cycle([(4, 5), (5, 2), (2, 3), (3, X)]).
X = 4.

?- domino_cycle([(A, 1), (1, 2), (2, A)]).
A = 1;
A = 2;
A = 3;
A = 4;
A = 5;
A = 6.

?- findall(C, (length(C, 5), domino_cycle(C)), _L), length(_L, L).
L = 7776.
```

first_missing_positive(+Items, -Result)

Unify `Result` with the smallest positive integer that is not a member of `Items`. The argument `Items` is guaranteed to be a list, but it can contain any bound Prolog terms as its elements, not merely integers. However, this predicate does not need to recursively look into the elements that are lists, but treat such elements as non-integers.

```
?- first_missing_positive([6, 8, 2, 999, 1, 4, 7], N).
N = 3.

?- first_missing_positive([42, 99, 123456, -3, 777], N).
N = 1.

?- first_missing_positive([bob, jack, foo(bar, baz, qux)], N).
N = 1.

?- first_missing_positive([2, 3, 4, [1, 1, 1, 1]], N).
N = 1.
```

three_summers(+Items, +Goal, -A, -B, -C)

Given a list of positive integer `Items` whose elements are guaranteed to be in sorted ascending order, and a positive integer `Goal`, unify `A`, `B`, and `C` with three elements taken from `Items` that together add up to `Goal`. The elements `A`, `B`, and `C` must occur inside the Items list in that order.

```
?- three_summers([3, 7, 9, 10, 12, 14], 30, A, B, C).
A = 7, B = 9, C = 14.

?- three_summers([1, 2, 3, 4, 5, 6], 12, A, B, C).
A = 1, B = 5, C = 6;
A = 2, B = 4, C = 6;
A = 3, B = 4, C = 5.

?- findall(X, between(1, 20, X), _L), findall((A, B, C),
three_summers(_L, 40, A, B, C), _LL), length(_LL, L).
L = 33.
```

It might be a good idea to define a helper predicate `two_summers(Items, Goal, A, B)`.

riffle(-Left, -Right, -Result, -Mode)

Unify `Result` with the list created by taking elements from `Left` and `Right` lists in alternating fashion. Both `Left` and `Right` lists must have the same length, and `Mode` must be either `left` or `right` to indicate which list the first element in `Result` comes from.

```
?- riffle([bob, 42, foo(bar)], [99, hello, world], L, left).
L = [bob, 99, 42, hello, foo(bar), world].

?- riffle(L1, L2, [odd, number, of, elements, cannot, succeed, here],
M).
false.

?- riffle([42, bob, 99], [55, jack, tom], [55|_], Mode).
Mode = right.

?- riffle(L1, L2, [A, B, C, D, E, F], M).
L1 = [A, C, E],
L2 = [B, D, F],
M = left ;
L1 = [B, D, F],
L2 = [A, C, E],
M = right ;
```

group_and_skip(+N, +Out, +In, -Leftovers)

A pile of `N` identical coins lies on the table. Each individual move arranges these `N` coins into groups with exactly `out` coins in each group, where `out` is guaranteed to be a positive integer greater than one. The `mod(N, Out)` leftover coins that do not fit into any group are put aside and recorded. From each complete group of `out` coins, exactly `ins` coins are added back in the pile, and the rest of the coins of that group are discarded.

For example, if `N = 100`, `Out = 8` and `In = 5`, these hundred coins form twelve complete groups of eight coins, with four leftover coins put aside and recorded. Putting back five coins for each group gives the new `N = 60`. The same operation then creates seven complete groups of eight coins, with four leftover coins again put aside and recorded, and putting back five coins for each group given the new `N = 35`. This should continue until `N = 0` so that the entire pile becomes empty. This must eventually happen whenever `out > ins`. Unify `Leftovers` with a list of how many coins were taken aside in each step, starting from the most recent step.

```
?- group_and_skip(12345, 10, 1, L).
L = [1, 2, 3, 4, 5].

?- group_and_skip(255, 2, 1, L).
L = [1, 1, 1, 1, 1, 1, 1, 1].

?- group_and_skip(10^9, 13, 3, L).
L = [3, 8, 5, 10, 8, 6, 11, 8, 9, 7, 0, 2, 1, 12].

?- group_and_skip(9876543210, 1000, 1, L).
L = [9, 876, 543, 210].
```

This algorithm produces the list of digits of positive integer `N` expressed in arbitrary base `Out/In` that can be a rational number. When `In = 1`, this reduces to the special case of integer conversion between familiar integer bases of `Out` such as binary, octal or hexadecimal.

`taxi_zum_zum(+Moves, -Pos)`

A taxicab driving around the integer grid (in style of the idealized Manhattan street grid spanning throughout the entire $\mathbb{Z}^2$ integer lattice) starts at origin `(0, 0)` and executes a series of `Moves` given as a string consisting of characters `'f'` (move one step forward to the current direction), `'l'` (turn left in place without moving) and `'r'` (turn right in place without moving). The initial direction vector of this cab is `(0, 1)` with the cab heading north. This predicate should unify `Pos` with the position on the grid that the taxicab ends at after executing these `Moves`.

In this problem, same as the next two problems for extracting the list of increasing integers from the given digit string and performing a pancake scramble, the SWI-Prolog built-in predicates `string_codes/2` and `string_chars/2` will surely come handy. As a more general guideline, Prolog programs ought to immediately convert any strings given to them into lists, trees and other structures of proper Prolog terms so that they get to deal with the data encoded in these strings with recursive Prolog unification rules, instead of performing these string computations with position indexing and substring slicing as in imperative languages.

```
?- taxi_zum_zum("rfrfrfrf", Pos).
Pos = (0, 0).

?- taxi_zum_zum("llflflrlfr", Pos).
Pos = (1, 0).

?- taxi_zum_zum("fflllfrlflrfrlrrl", Pos).
Pos = (3, 2).
```

extract_increasing(+Digits, -Nums)

Unify `Nums` with the list of strictly increasing numbers extracted from the string of `Digits` by reading its digits from left to right. The leftover digits that do not form a sufficiently large integer larger than the previous number are discarded.

```
?- extract_increasing("1234567890987654321", Nums).
Nums = [1, 2, 3, 4, 5, 6, 7, 8, 9, 98, 765, 4321].

?- extract_increasing("122333444455555666666", Nums).
Nums = [1, 2, 23, 33, 44, 445, 555, 566, 666].

?- extract_increasing("3141592653589793238462643383279502884", Nums).
Nums = [3, 14, 15, 92, 653, 5897, 9323, 84626, 433832, 795028]
```

`pancake_scramble(+Text, -Result)`

Unify `Result` with the result of performing the full **pancake scramble** to the original `Text`, as defined in the "Pancake Scramble" problem at the excellent Wolfram Challenges site where the author originally found and shamelessly stole this little problem.

```
?- pancake_scramble("Z", Result).
Result = "Z".

?- pancake_scramble("pancake", Result).
Result = "eanpack".

?- pancake_scramble("Artificial Intelligence", Result).
Result = "englen acftAriiilItliec".
```

You might want to first implement a helper predicate `pancake_flip(L, N, Flipped)` that performs a single pancake flip to the first `N` elements of the list `L`. Harness the power of the seemingly simple predicates `length/2`, `append/2` and `reverse/2` instead of trying to perform the operation yourself.

```
?- string_chars("Prolog is the best", _L), pancake_flip(_L, 6, _F),
string_chars(Result, _F).
Result = "golorP is the best".
```

`tukeys_ninther(+Items, -M)`

Hailing from a simpler time back when computers had a lot less memory than we take for granted today not just for our cell phones but nearly any embedded device that contains some kind of microprocessor inside it, Tukey's ninther is a simple but surprisingly robust approximation algorithm to quickly find an element that is, if not the true **median** of those numbers, reasonably close to it with high probability.

Tukey's algorithm conceptually splits the list into triplets and constructs the list of the separate medians of each of these triplets. The search for the approximate median continues using this list of medians-of-three. This way the length of the list is always divided by three until it contains only one element that is then the final answer. For simplicity, this predicate may assume that length of `Items` is always some power of three. In the following table of example queries, the list used in each query is always the list of medians-of-three for the list used in the query after it.

Tukey's algorithm does not perform any integer arithmetic but depends only on the order comparisons of these elements, which makes it completely insensitive to the scale and the statistical distribution of these elements. This predicate should unify `M` with the element that Tukey's ninther algorithm would return for the given `Items`. Note that this is not necessarily the actual median of the list, since that would be a tad bit more expensive to find.

```
?- tukeys_ninther([15], M).
M = 15.

?- tukeys_ninther([42, 7, 15], M).
M = 15.

?- tukeys_ninther([99, 42, 17, 7, 1, 9, 12, 77, 15], M).
M = 15.

?- tukeys_ninther([55, 99, 131, 42, 88, 11, 17, 16, 104, 2, 8, 7, 0,
1, 69, 8, 93, 9, 12, 11, 16, 1, 77, 90, 15, 4, 123], M).
M = 15.
```

`bulgarian_solitaire(+L, +K, -Moves)`

Given a list `L` of positive integers, each move in the mathematical patience game of Bulgarian solitaire subtracts one from each element of the list. It then eliminates all elements that have become zeros, and adds a new element equal to the original length of `L` to the beginning of the list.

For example, `[6, 3, 5, 1]` turns into `[4, 5, 2, 4]`, which turns into `[4, 3, 4, 1, 3]`, which turns into `[5, 3, 2, 3, 2]`. You might want to first implement this operation to perform one move as a helper predicate `bs_next(Curr, Next)`, to simplify the actual predicate.

When the sum of elements of `L` equals `K*(K+1)/2`, repeating this simple move is guaranteed to eventually reach the goal state where every positive integer from 1 to `K` appears exactly once. (You might also want to write a separate predicate to test for that.) This predicate should unify `Moves` with the number of moves required to reach this goal starting from the list `L`.

```
?- bulgarian_solitaire([3, 5, 2, 1, 4], 5, Moves).
Moves = 0.

?- bulgarian_solitaire([5, 2, 2, 1], 4, Moves).
Moves = 11.

?- bulgarian_solitaire([10, 10, 10, 10, 10, 5], 10, Moves).
Moves = 74.

?- bulgarian_solitaire([1250, 1250, 2550], 100, Moves).
Moves = 8185.
```

`josephus(+Men, +K, -Last)`

In the classic Josephus problem, a list of `Men` has been arranged in a grim circle. Counting from the beginning, every `k`:th man is swiftly eliminated, removing him from this circle. This continues until only one man remains. This predicate should unify `Last` with the last man left standing.

```
?- josephus([joe, moe, bob, rob, josephus], 2, Last).
Last = bob.

?- josephus([joe, moe, bob, rob, josephus], 99, Last).
Last = josephus.

?- findall(N, between(1, 30, N), _L), josephus(_L, 4, Last).
Last = 6.

?- findall(N, between(1, 1234, N), _L), josephus(_L, 42, Last).
Last = 426.
```

sz(+N, -SZ)

Positive integers whose base-10 representation consists of some series of the digit seven followed by some series of zero digits (for example, 77777000, 7000 or 77777777) turn out to have a curious property that for any positive integer N, there exists at least one number SZ of such highly constrained form so that SZ is divisible by N. This predicate should unify SZ with the smallest such integer for the given N.

```
?- sz(42, SZ).
SZ = 7770.

?- sz(400000, SZ).
SZ = 70000000.

?- sz(123, SZ).
SZ = 777777777777777.

?- sz(721, SZ).
SZ = 77777777777777777777777777777777.

?- sz(1234, SZ).
SZ =
77777777777777777777777777777777777777777777777777777777777777777777777777
77777777777777770.
```

In absence of more clever mathematics of integers, this predicate can just systematically try out all integers of this seven-zero form in ascending order until it finds one that is divisible by N. You might therefore want to first define the predicate seven_zero(N, S, Z) to unify N with the integer whose base-10 representation consists of S consecutive copies of the digit seven, followed by Z consecutive copies of the digit zero.

```
?- seven_zero(N, 5, 8).
N = 7777700000000.

?- seven_zero(N, 20, 0).
N = 77777777777777777777.

?- seven_zero(N, 0, 10000000000).
N = 0.
```

`crag(-A, -B, -C, -Score)`

Unify `Score` with the maximum number of points you can get for rolling three six-sided dice `A`, `B`, and `C` according to the scoring table of Crag, an old dice game similar in style and spirit to the better known dice game of Yahtzee, but with much simpler rules (and combinatorics of only three dice instead of five) that make this game less tedious for us to encode in logic. The roll that consists of these three dice is considered in isolation from the rest of the game so that all scoring columns are still available for use.

This predicate must be fully general and reversible in that each one of its four arguments can be an unbound variable. Especially the most general query `crag(A, B, C, Score)` with its all four arguments unbound must produce each one of the $6^3 = 216$ possible solutions exactly once. (The system predicate `between/3` will probably come pretty handy in this task.)

```
?- crag(4, 5, 4, S).
S = 50.

?- crag(2, 3, 3, S).
S = 6.

?- crag(2, 6, 4, S).
S = 20.

?- findall((A, B), crag(A, B, 5, 10), _L), length(_L, L).
L = 8.

?- findall((A, B, C), crag(A, B, C, 6), _L), length(_L, L).
L = 57.

?- findall((A, B, C, S), crag(A, B, C, S), _L), length(_L, L).
L = 216.
```

This problem might also be nicely amenable to the finite domain constraint logic programming mechanism of the SWI-Prolog clpfd library.

count_dominators(+Items, -Result)

Define a **dominator element** inside a list to be an element that is **strictly greater** than **all** of the elements that follow it in the list. (The last element of any list is therefore automatically a dominator, by definition.) Unify `Result` with the count of how many dominator elements exist inside the list of integer `Items`.

```
?- count_dominators([42, 99, 17, 3, 9], D).
D = 3.

?- count_dominators([4, 3, 2, 1], D).
D = 4.

?- count_dominators([1, 2, 3, 4], D).
D = 1.

?- count_dominators([], D).
D = 0.
```

duplicate_digit_bonus(+N, -B)

Some people find it pleasant when large integers contain "dubs", "trips" and even longer blocks of consecutive repeated digits. Let us assume that such fellows assign a bonus for each positive integer so that every maximal consecutive block of `C` identical digits, with `C` at least two, earns a bonus of `10^(C-2)` points for that block. That is, these bonuses grow exponentially with respect to the block length so that length two earns one point, length three earns ten points, length four earns hundred points, and so on. Furthermore, a special rule says that any block of digits located at the lowest end of the integer earns twice the normal bonus. This predicate should unify `B` with the total bonus of the given positive integer `N`.

```
?- duplicate_digit_bonus(2223, B).
B = 10.

?- duplicate_digit_bonus(3222, B).
B = 20.

?- duplicate_digit_bonus(211111174711111777700, B).
B = 12002.

?- duplicate_digit_bonus(9999997777774444488872222, B).
B = 21210.

?- _N is 10^20, duplicate_digit_bonus(_N, B).
B = 2000000000000000000.

?- _N is 10^20 + 7, duplicate_digit_bonus(_N, B).
B = 1000000000000000000.

?- _N is 1234^5678, duplicate_digit_bonus(_N, B).
B = 15418.
```

give_change(+Amount, +Coins, -Change)

As a cashier, your job is to give back the correct `Change` for the `Amount` that you owe the customer, using only coin denominations listed in `Coins`, given in sorted descending order. To simplify your mental effort standing on your two feet all day in the front lines making sure that people are fed even the pandemic, you compute the change that you give back with the **greedy** method that always uses the largest available coin denomination.

Your till is assumed to have a sufficient number of every coin denomination available to give out. However, this predicate must recognize the situation where the greedy method cannot give back the exact change, and silently fail for all such queries.

```
?- give_change(120, [50, 40, 5], Change).
Change = [50, 50, 5, 5, 5, 5].

?- give_change(100, [42, 17, 11, 6, 1], Change).
Change = [42, 42, 11, 1, 1, 1, 1, 1].

?- give_change(34, [20, 9, 6], L).
false.
```

running_median(+Items, -Medians)

Unify `Medians` with a list of integers acquired by replacing each element with the **median** of that element and the two immediately preceding elements in `Items`. The first two elements of `Medians` should equal the first two elements of `Items`.

```
?- running_median([1, 2, 3, 4, 5], M).
M = [1, 2, 2, 3, 4].

?- running_median([99, 42, 17, 55, -4, 18, 77], M).
M = [99, 42, 42, 42, 17, 18, 18].

?- running_median([42, 42, 99, 42, 42], M).
M = [42, 42, 42, 42, 42].
```

safe_squares_rooks(+Rooks, +N, -S)

Unify `S` with the integer count of squares in a generalized `N`-by-`N` chessboard that are **safe** from the list of `Rooks` positioned on that board. A square is safe if no rooks on the board lie in the same row or column as that square. Positions of the individual rooks are expressed as terms of the form `(Row, Col)` where both coordinates `Row` and `Col` are integers between `1` and `N`, inclusive.

```
?- safe_squares_rooks([(2, 2), (3, 1), (5, 5), (2, 5)], 5, S).
S = 4.

?- safe_squares_rooks([(1, 1), (2, 2), (3, 3), (4, 4), (5, 5)], 5,
S).
S = 0.

?- safe_squares_rooks([], 100, S).
S = 10000.
```

trick_winner(+Cards, -Winner)

Unify `Winner` with the playing card that wins that trick in a trick-taking card game such as bridge, when the list of `Cards` has been played into that trick. The trick is won by the **highest ranking card in the suit of the first card played to the trick**. Cards from other suits, even if they are aces, can never win the trick. Each playing card is represented as a term `(Rank, Suit)`, where `Rank` is from `[ace, king, queen, jack, ten, nine, eight, seven, six, five, four, three, two]`, and `Suit` is from `[clubs, diamonds, hearts, spades]`.

```
?- trick_winner([(four, spades), (deuce, hearts), (nine, spades),
(nine, clubs)], C).
C = (nine, spades) ;
false.

?- trick_winner([(six, spades), (deuce, hearts), (X, spades), (nine,
clubs)], (six, spades)).
X = five ;
X = four ;
X = three ;
X = two ;
false.
```

You might first want to implement an auxiliary predicate `suit(S)` to generate all possible suits, and another predicate `higher_rank(R1, R2)` that generates all ranks `R1` that are higher than the second rank `R2`.

```
?- suit(S).
S = clubs ;
S = diamonds ;
S = hearts ;
S = spades.

?- higher_rank(seven, R).
R = six ;
R = five ;
R = four ;
R = three ;
R = two ;
false.
```

sum_of_two_squares(+N, -A, -B)

Unify `A` and `B` with positive integers whose squares add up to `N`. If there exist multiple ways to express `N` as a sum of two squares, this predicate should succeed only once, for the way that uses the largest possible working `A`. So use those cuts in their right and proper places to ensure that only one solution is generated.

It might be a good idea to first define a helper predicate to find the largest integer `A` whose square is less than `N`, to give you starting point where to look for these values of `A` and `B`. Then, depending on whether `A^2 + B^2` for your current values of `A` and `B` is greater than, less than or equal to the goal value `N`, your tail-recursive rules either decrease `A`, increase `B`, or terminate with success.

```
?- sum_of_two_squares(2, A, B).
A = B,
B = 1.

?- sum_of_two_squares(50, A, B).
A = 7,
B = 1.

?- sum_of_two_squares(100, A, B).
A = 8,
B = 6.

?- _N is 333^2 + 444^2, sum_of_two_squares(_N, A, B).
A = 528,
B = 171.

?- findall(N, (between(1, 100, N), sum_of_two_squares(N, _, _)), _L),
length(_L, L).
L = 35.
```

hitting_integer_powers(+A, +B, +T, -Pa, -Pb)

This predicate should unify `Pa` and `Pb` with the smallest possible positive integers so that the integer powers `A^Pa` and `B^Pb` are "close enough", which in this problem means that their absolute difference multiplied by the expected **tolerance level** given by `T` is at most equal to the smaller of these two powers.

Start with the first powers of `A` and `B`. While these two powers are not close enough, multiply the power that is currently smaller by the original base, and increment the corresponding power by one. Once the powers are close enough in the base case of the tail recursion produced by this simulated while-loop, unify the current exponents with the unbound parameters `Pa` and `Pb`.

```
?- hitting_integer_powers(3, 6, 100, Pa, Pb).
Pa = 137,
Pb = 84.

?- hitting_integer_powers(6, 10, 1000, Pa, Pb).
Pa = 595,
Pb = 463.

?- hitting_integer_powers(2, 10, 10000, Pa, Pb).
Pa = 13301,
Pb = 4004.
```

sum_of_distinct_cubes(+N, -L)

Unify `L` with the list of distinct positive integers, listed in descending order, whose cubes add up to the positive integer `N`.

Whenever there are multiple ways to break down the integer `N` into a sum of distinct cubes, this predicate must deterministically produce the **lexicographically highest** such breakdown, that is, whose first element is as large as possible, and continuing this same principle for the elements in the remaining positions. This is not necessarily the solution that uses the fewest number of cubes that are added together. (See the last example query in the table below for illustration of precisely such a situation.) However, this requirement of deterministically finding the lexicographically largest solution turns out to be easy to follow by ordering your recursive calls appropriately.

```
?- sum_of_distinct_cubes(100, L).
L = [4, 3, 2, 1].

?- sum_of_distinct_cubes(721, L).
false.

?- sum_of_distinct_cubes(12345, L).
L = [20, 12, 10, 9, 8, 6, 5, 3, 2].

?- sum_of_distinct_cubes(999999999999, L).
L = [9999, 669, 81, 27, 7, 6, 2].

?- X is 123^3 + 456^3 + 789^3, sum_of_distinct_cubes(X, L).
X = 587848752,
L = [837, 113, 30, 13, 6, 5, 4].
```

`fibonacci_sum(+N, -L)`

Unify `L` with the unique list of non-consecutive **Fibonacci numbers** that add up to the positive integer `N`. Since $F_{n+2} = F_n + F_{n+1}$, you can always replace two consecutive Fibonacci numbers with the next higher number of this series without affecting the overall total, and this way iteratively get rid of any consecutive Fibonacci numbers. The rest of the proof for Zeckendorf's theorem that the representation of `N` as a sum of distinct non-consecutive Fibonacci numbers is unique can be found in basically any introductory textbook on discrete math and combinatorics.

However, we don't need to know the details of this proof to be able to trust its predictions: same as with all machines both physical and virtual, this abstract proof machine goes brrrrr regardless of our state of knowledge. We just need to know that the **greedy algorithm** of each time using the largest Fibonacci number `F` that is less than or equal to `N` will always produce the correct answer, leaving the difference `N-F` to be recursively converted into Fibonacci numbers. So, please do that.

```
?- fibonacci_sum(30, L).
L = [21, 8, 1].

?- fibonacci_sum(1000000, L).
L = [832040, 121393, 46368, 144, 55].

?- X is 10^20, fibonacci_sum(X, L).
X = 100000000000000000000,
L = [836211434898484229777, 1220016041512187673 8, 288006719437081612 0,
110008777836610193 1, 160500643816367088, 37889062373143906,
117669030460994, 27777890035288, 4052739537881|...].
```

To simplify the logic of your predicate, you might define a helper predicate `fibs_upto(N, L)` that unifies `L` with the list of Fibonacci numbers up to the given integer `N`.

```
?- fibs_upto(34, L).
L = [34, 21, 13, 8, 5, 3, 2, 1, 1].

?- fibs_upto(77, L).
L = [55, 34, 21, 13, 8, 5, 3, 2, 1, 1].

?- N is 10^10, fibs_upto(N, L).
L = [7778742049, 4807526976, 2971215073, 1836311903, 1134903170,
701408733, 433494437, 267914296, 165580141|...].
```