1. **Написать предикат, который печатает все нечётные числа из диапазона в порядке убывания. Границы диапазона вводятся с клавиатуры в процессе работы предиката.**

функ(N, N) :-

N mod 2 =:= 1,

write(N), nl.

функ(N, N) :-

N mod 2 =:= 0,

write(''), nl.

функ(N, M) :-

M mod 2 =:= 1,

write(M), write(' '),

M1 is M - 1,

функ(N, M1).

функ(N, M) :-

M mod 2 =:= 0,

M1 is M - 1,

функ(N, M1).

нету\_иф(N, M) :-

N =< M,

функ(N, M).

нету\_иф(N, M) :-

N > M,

функ(M, N).

давай :-

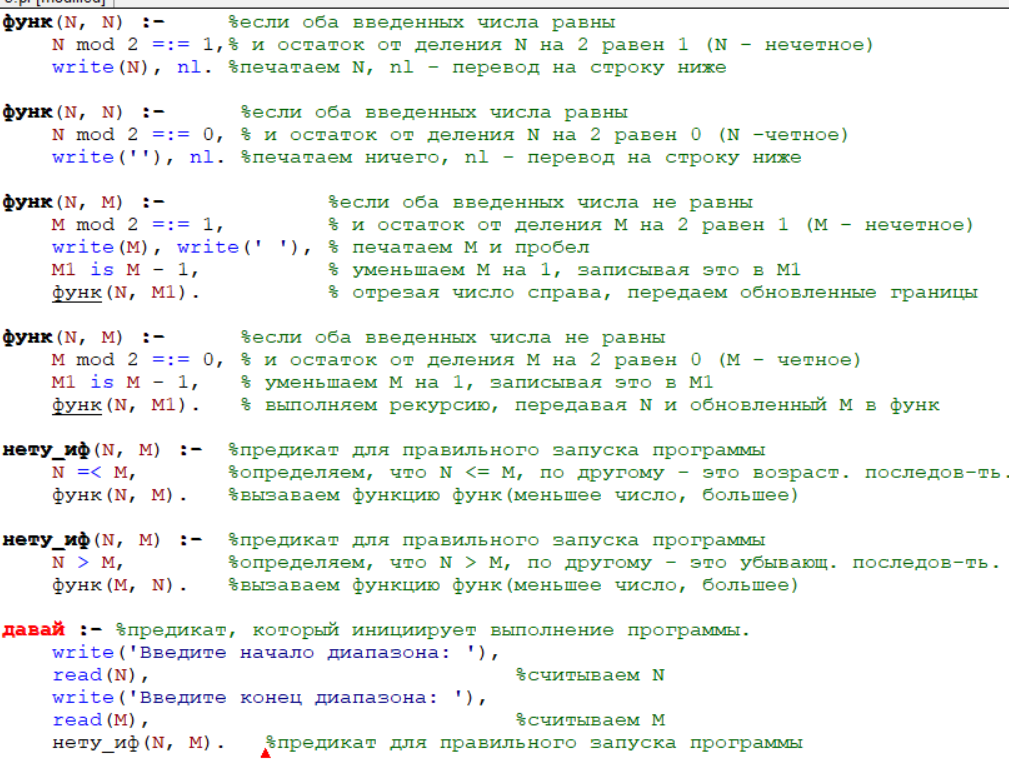
write('Введите начало диапазона: '),

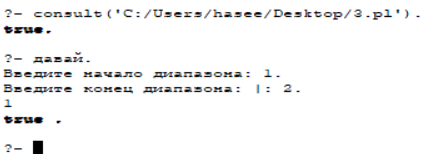
read(N),

write('Введите конец диапазона: '),

read(M),

нету\_иф(N, M).





1. **Написать предикат, который находит числа Фибоначчи по их номерам, которые в цикле вводятся с клавиатуры. Запрос номера и нахождение соответствующего числа Фибоначчи должно осуществляться до тех пор, пока не будет введено отрицательное число. Циклический ввод организовать с помощью предиката repeat. Числа Фибоначчи определяются по следующим формулам: F(0)=1, F(1)=1, F(i)=F(i-2)+F(i-1) (i=2, 3, 4, ...).**

найти\_фибоначчи :-

repeat,

write('Введите номер числа Фибоначчи (отрицательное число для выхода): '),

read(N),

(N < 0 ->

! ; % Выход из цикла, если введено отрицательное число

fib(N, Result),

write('Число Фибоначчи с номером '), write(N), write(' равно '), write(Result), nl,

fail % Повторить цикл

).

fib(0, 1). % Базовый случай

fib(1, 1). % Базовый случай

fib(N, Result) :-

N > 1,

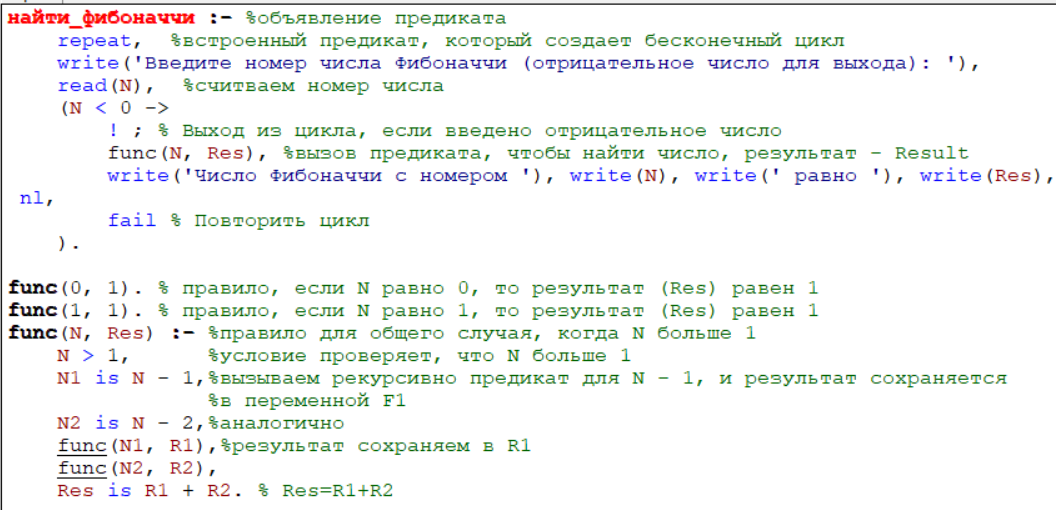
N1 is N - 1,

N2 is N - 2,

fib(N1, F1),

fib(N2, F2),

Result is F1 + F2.



Вызов:

**найти\_фибоначчи.**

После чего вводишь число и точку после него: 4.

1. **Написать предикат, который разбивает числовой список по двум числам, вводимым с клавиатуры на три списка: меньше меньшего введенного числа, от меньшего введенного числа до большего введенного числа, больше большего введенного числа. Список и два числа вводятся с клавиатуры в процессе работы предиката. Например: [3,7,1,-3,5,8,0,9,2], 8, 3 –> [1,-3,0,2], [3,7,5,8], [9]**

% Предикат для разбиения списка на три подсписка

split\_list(List, Low, High, Less, Between, More) :-

% Вызываем предикаты для разделения списка

split\_list(List, Low, High, [], [], [], Less, Between, More).

% Базовый случай: если список пустой, то возвращаем подсписки

split\_list([], \_, \_, Less, Between, More, Less, Between, More).

% Разбиваем список на три подсписка в зависимости от значения границ

split\_list([X | Rest], Low, High, LTL, BLAH, MTH, Less, Between, More) :-

X < Low, % Если элемент меньше Low

append(LTL, [X], NewLTL), % Добавляем его к списку "меньше Low"

split\_list(Rest, Low, High, NewLTL, BLAH, MTH, Less, Between, More).

split\_list([X | Rest], Low, High, LTL, BLAH, MTH, Less, Between, More) :-

X >= Low, X =< High, % Если элемент в пределах от Low до High включительно

append(BLAH, [X], NewBLAH), % Добавляем его к списку "между Low и High"

split\_list(Rest, Low, High, LTL, NewBLAH, MTH, Less, Between, More).

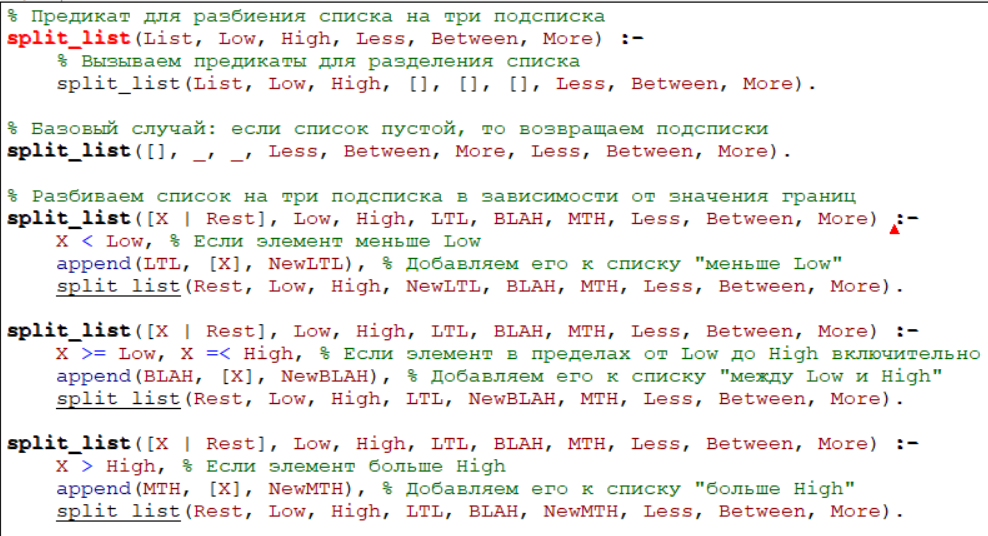
split\_list([X | Rest], Low, High, LTL, BLAH, MTH, Less, Between, More) :-

X > High, % Если элемент больше High

append(MTH, [X], NewMTH), % Добавляем его к списку "больше High"

split\_list(Rest, Low, High, LTL, BLAH, NewMTH, Less, Between, More).

split\_list([1, 2, 3, 4, 5, 6], 3, 5, Less, Between, More).



1. **split\_list(List, Low, High, Less, Between, More) :-**
   * Это определение предиката **split\_list** с аргументами **List**, **Low**, **High**, **Less**, **Between**, и **More**.
2. **% Вызываем предикаты для разделения списка**
   * Комментарий описывает, что в этом предикате будет использоваться рекурсивное разделение списка.
3. **split\_list([], \_, \_, Less, Between, More, Less, Between, More).**
   * Базовый случай: если **List** пустой, то **Less**, **Between** и **More** устанавливаются равными тому, что было передано как аргументы.
4. **split\_list([X | Rest], Low, High, LTL, BLAH, MTH, Less, Between, More) :-**
   * Это начало первого правила разделения, где **X** - это головной элемент списка, **Rest** - остаток списка, **Low** и **High** - нижняя и верхняя границы для разделения, а **LTL**, **BLAH** и **MTH** - текущие подсписки для "меньше Low", "между Low и High", и "больше High".
5. **X < Low,**
   * Это условие: если **X** меньше чем **Low**.
6. **append(LTL, [X], NewLTL),**
   * Создание нового списка **NewLTL**, который является результатом добавления элемента **X** в список **LTL**.
7. **split\_list(Rest, Low, High, NewLTL, BLAH, MTH, Less, Between, More).**
   * Рекурсивный вызов **split\_list** для остатка списка **Rest** с обновленным **NewLTL** и остальными подсписками.
8. Похожим образом, следующие два блока кода обрабатывают случаи, когда **X** находится в диапазонах "между Low и High" или "больше High". Они также выполняют добавление элементов в соответствующие подсписки и рекурсивные вызовы **split\_list**.

**split\_list([], \_, \_, Less, Between, More, Less, Between, More).**

Эта строка кода - это базовый случай для предиката **split\_list**. Здесь, когда список **List** становится пустым, не важно, какие были значения **Low**, **High** и подсписки **Less**, **Between**, **More**, эти значения должны оставаться неизменными. Поэтому они дублируются для ясности и согласно синтаксису Prolog.

Другими словами, эта строка кода означает следующее: "Когда список пустой, вернуть значения **Less**, **Between** и **More**, которые равны тому, что было передано в предикат в качестве аргументов."

1. Первоначально, **LTL**, **BLAH**, и **MTH** - это пустые списки.
2. При каждом рекурсивном вызове предиката **split\_list**, когда элементы добавляются в **LTL**, **BLAH**, или **MTH**, создаются новые списки (например, **NewLTL**, **NewBLAH**, и **NewMTH**), в которых добавлены соответствующие элементы.
3. В базовом случае (когда список **List** становится пустым), значения **Less**, **Between**, и **More** устанавливаются равными значениям **LTL**, **BLAH**, и **MTH** соответственно. Таким образом, в этот момент значения списков **LTL**, **BLAH**, и **MTH** "копируются" в **Less**, **Between**, и **More**.
4. **Написать предикат, который формирует список из наиболее часто встречающихся элементов списка. Список вводится с клавиатуры в процессе работы предиката. Встроенные предикаты поиска максимума и сортировки не использовать! Например: [0,3,5,7,1,5,3,0,3,3,5,7,0,5,0] –> [0,3,5].**

most\_frequent\_elements(List, Result) :-

% Подсчитываем частоту появления каждого элемента в списке

count\_elements(List, ElementCounts),

% Находим максимальную частоту

max\_frequency(ElementCounts, MaxFrequency),

% Извлекаем элементы с максимальной частотой

find\_most\_frequent(ElementCounts, MaxFrequency, Result).

% Подсчет частоты элементов в списке

count\_elements([], []).

count\_elements([H|T], Result) :-

count\_elements(T, TmpResult),

update\_element\_count(H, TmpResult, Result).

% Если элемент уже есть в списке частот, увеличиваем его счетчик

update\_element\_count(Element, [], [element(Element, 1)]).

update\_element\_count(Element, [element(Element, Count)|T], [element(Element, NewCount)|T]) :-

NewCount is Count + 1.

update\_element\_count(Element, [X|T], [X|NewT]) :-

Element \= X,

update\_element\_count(Element, T, NewT).

% Нахождение максимальной частоты в списке частот

max\_frequency([element(\_, Count)], Count).

max\_frequency([element(\_, Count)|Rest], MaxCount) :-

max\_frequency(Rest, TmpMax),

MaxCount is max(Count, TmpMax).

% Извлечение элементов с максимальной частотой

find\_most\_frequent([], \_, []).

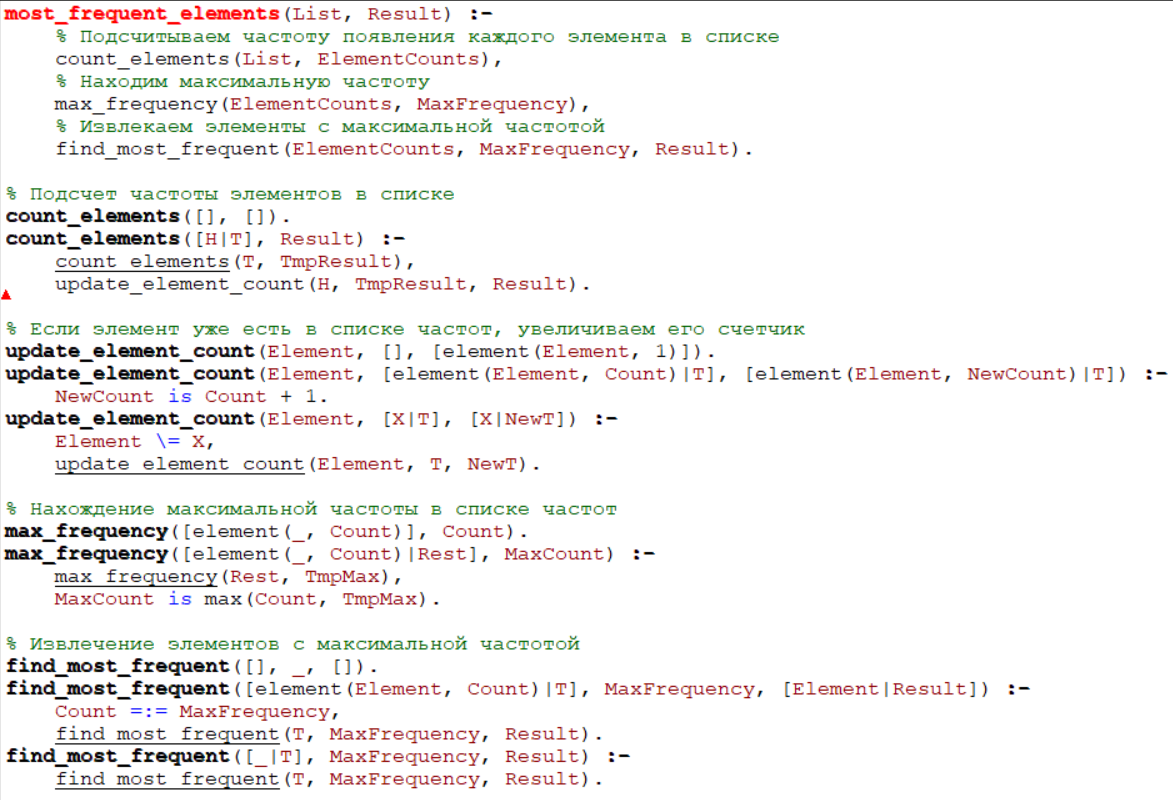
find\_most\_frequent([element(Element, Count)|T], MaxFrequency, [Element|Result]) :-

Count =:= MaxFrequency,

find\_most\_frequent(T, MaxFrequency, Result).

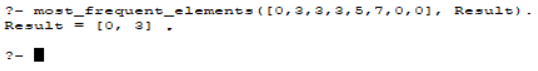
find\_most\_frequent([\_|T], MaxFrequency, Result) :-

find\_most\_frequent(T, MaxFrequency, Result).



Вызов:

most\_frequent\_elements([0,3,3,3,5,7,0,0], Result).



1. **most\_frequent\_elements(List, Result) :-**: Это определение предиката **most\_frequent\_elements**, который принимает два аргумента: **List** (список, в котором вы хотите найти наиболее часто встречающиеся элементы) и **Result** (результат - список наиболее часто встречающихся элементов).
2. **count\_elements(List, ElementCounts),**: Здесь вызывается предикат **count\_elements**, который подсчитывает частоту появления каждого элемента в **List** и возвращает результат в **ElementCounts**.
3. **max\_frequency(ElementCounts, MaxFrequency),**: Вызывается предикат **max\_frequency**, который находит максимальную частоту в списке **ElementCounts** и возвращает её в **MaxFrequency**.
4. **find\_most\_frequent(ElementCounts, MaxFrequency, Result).**: Здесь вызывается предикат **find\_most\_frequent**, который извлекает элементы из списка **ElementCounts** с максимальной частотой **MaxFrequency** и сохраняет результат в **Result**.
5. **count\_elements([], []).**: Это базовый случай предиката **count\_elements**. Если входной список (**List**) пуст, то результатом будет пустой список **[]**. Это базовое правило указывает на окончание рекурсии.
6. **count\_elements([H|T], Result) :-**: Это правило предиката **count\_elements**. Оно говорит о том, как выполнить подсчет частоты элементов в непустом списке. **H** - это голова списка, **T** - хвост списка, **Result** - результат подсчета частот.
7. **count\_elements(T, TmpResult), update\_element\_count(H, TmpResult, Result).**: В этой строке происходит рекурсивный вызов предиката **count\_elements**. Сначала мы рекурсивно вызываем **count\_elements** для хвоста списка **T** и сохраняем результат во временной переменной **TmpResult**. Затем мы вызываем предикат **update\_element\_count** с текущей головой **H**, временным результатом **TmpResult** и результатом подсчета **Result**. Это обновляет **Result**, учитывая текущий элемент **H** и уже подсчитанные частоты в **TmpResult**.

Таким образом, предикат **count\_elements** рекурсивно обходит весь список и создает список **Result**, который содержит частоту появления каждого элемента во входном списке.

1. **update\_element\_count(Element, [], [element(Element, 1)]).**: Это базовый случай предиката **update\_element\_count**. Если список **[]** (пустой), то создается новый элемент с частотой 1 для элемента **Element**. Таким образом, это начальное состояние, когда мы встречаем элемент впервые.
2. **update\_element\_count(Element, [element(Element, Count)|T], [element(Element, NewCount)|T]) :-**: Это правило предиката **update\_element\_count**. Если элемент **Element** уже существует в списке частот с частотой **Count**, то мы обновляем эту частоту, увеличивая её на 1. Таким образом, мы учитываем новое вхождение элемента.
3. **update\_element\_count(Element, [X|T], [X|NewT]) :-**: Это правило предиката **update\_element\_count**. Если текущий элемент **X** в списке не соответствует элементу **Element**, мы просто копируем его в новый список **NewT** без изменений, и рекурсивно вызываем **update\_element\_count** для хвоста списка **T**. Это обработка случая, когда мы ещё не дошли до элемента **Element** в списке частот.

Итак, **update\_element\_count** обновляет список частот появления элементов, добавляя новые элементы с частотой 1, увеличивая частоту существующих элементов или просто копируя элементы, которые не соответствуют **Element**.

**[element(Element, 1)]** - это список с одним элементом, который представляет информацию о частоте появления элемента **Element**. Элемент в данном контексте представлен как **element(Element, 1)**, где **Element** - это конкретный элемент, а **1** - это его частота появления.

Каждый элемент этого списка содержит информацию о частоте появления одного элемента в исходном списке. Например, если у нас есть список **[1, 2, 2, 3]**, то соответствующий список **element/2** может выглядеть так:

* **[element(1, 1), element(2, 2), element(3, 1)]**.

Это форма представления данных, которую использует предикат **update\_element\_count** для хранения частоты появления элементов.

**[element(Element, NewCount)|T]** представляет собой список, в котором первый элемент имеет структуру **element(Element, NewCount)**, а хвост списка обозначается как **T**.

В данном контексте:

* **element(Element, NewCount)** представляет информацию о частоте появления элемента **Element** в списке, где **Element** - это конкретный элемент, а **NewCount** - новая частота появления этого элемента.
* **T** представляет хвост списка, который содержит оставшиеся элементы списка частот. То есть, это оставшаяся часть списка частот после обработки текущего элемента.

Когда **update\_element\_count** используется, она обновляет список частот таким образом, что первый элемент в списке (**element(Element, NewCount)**) представляет информацию о частоте появления элемента **Element**, а **T** содержит остальные элементы списка частот после обработки текущего элемента.

1. **MaxFrequency([element(\_, Count)], Count).**: Это базовый случай предиката **max\_frequency**. Если в списке частот остался только один элемент (**[element(\_, Count)]**), то это значение **Count** (частота) считается максимальным значением. То есть, если у нас больше нечего сравнивать, мы просто возвращаем **Count** в качестве максимальной частоты.
2. **max\_frequency([element(\_, Count)|Rest], MaxCount) :-**: Это правило предиката **max\_frequency**. Оно применяется, когда в списке частот есть более одного элемента. **element(\_, Count)** представляет текущий элемент в списке, где **Count** - это частота этого элемента, а **Rest** - оставшаяся часть списка частот.
3. **max\_frequency(Rest, TmpMax), MaxCount is max(Count, TmpMax).**: В этой строке рекурсивно вызывается **max\_frequency** для оставшейся части списка частот (**Rest**) и результат сохраняется во временной переменной **TmpMax**. Затем мы сравниваем **Count** (частоту текущего элемента) с **TmpMax** с помощью **max/2**, чтобы найти максимальное значение, и сохраняем его в переменной **MaxCount**. Это позволяет нам найти максимальную частоту среди всех элементов списка частот.

Таким образом, предикат **max\_frequency** рекурсивно обходит список частот и находит максимальное значение (максимальную частоту) среди них.

1. **find\_most\_frequent([], \_, []).**: Это базовый случай предиката **find\_most\_frequent**. Если список частот пуст (**[]**), то результатом будет пустой список (третий аргумент).
2. **find\_most\_frequent([element(Element, Count)|T], MaxFrequency, [Element|Result]) :-**: Это первое правило предиката **find\_most\_frequent**. Оно применяется, когда у нас есть элемент **element(Element, Count)** в списке частот. Мы проверяем, равна ли частота **Count** максимальной частоте **MaxFrequency**. Если да, то мы добавляем **Element** в список **Result**, который содержит элементы с максимальной частотой, и рекурсивно вызываем **find\_most\_frequent** для оставшейся части списка **T**.
3. **find\_most\_frequent([\_|T], MaxFrequency, Result) :-**: Это второе правило предиката **find\_most\_frequent**. Оно применяется, когда текущий элемент в списке частот не соответствует максимальной частоте. В этом случае мы просто игнорируем этот элемент и переходим к следующему элементу списка, рекурсивно вызывая **find\_most\_frequent** для оставшейся части списка **T**.

Таким образом, предикат **find\_most\_frequent** фильтрует список частот и возвращает список элементов, чьи частоты соответствуют максимальной частоте **MaxFrequency**. Это позволяет найти наиболее часто встречающиеся элементы в списке.

Начало формы