Lista de Linguagens de Programação – 23

Nome: _____ Matrícula: _____

1. Escreva um predicado `max(X, Y, Z)`, que receba dois números, `X` e `Y` e unifique `Z` com o maior deles.

2. Escreva um predicado `maxList(L, M)`, que receba uma lista `L` de números e unifique `M` com o maior número nesta lista. O predicado deve falhar se a lista estiver vazia.

3. Escreva um predicado `ordered(L)`, que seja verdadeiro se a lista `L` estiver em ordem crescente.

4. Escreva um predicado `mergesort(In, Out)`, que receba duas listas, `In` e `Out`, e seja verdadeiro se `Out` for uma versão da lista `In`, ordenada em ordem crescente. Seu predicado deve usar o algoritmo de ordenação *mergesort*. Note que SWI-PL já possui um predicado *merge*, que obviamente você **não** vai usar. Use um outro nome para seu predicado, ou haverá conflito com SWI-PL.

5. Escreva um predicado `nqueens(N, X)`, que receba um inteiro `N` e encontre uma solução para o problema das N rainhas. O problema das N rainhas é parecido com o problema das oito rainhas, que é explicado nas transparências do livro usado no curso. Só que agora você precisa colocar N rainhas em um tabuleiro de tamanho N $\times$ N. Você pode começar a partir do código que o autor do livro disponibilizou. Há um *link* para este código na página do curso.

6. Escreva um predicado `multiknap(Pantry, Capacity, Knapsack)`, que funcione como o predicado `knapsackOptimization`, visto nas transparências do livro, mas que resolva o problema permitindo múltiplas ocorrências do mesmo item. Isto quer dizer que nesta versão do problema você pode pegar qualquer quantidade de itens da geladeira. Assim, se `Pantry` for:

   ```
   [food(break, 4, 9200),
    food(pasta, 2, 4600),
    food(peanutButter, 1, 6700),
    food(babyFood, 3, 6900)].
   ```

   Então sua mochila irá conter zero ou mais cópias de `food(bread, 4, 9200)`, zero ou mais cópias de `food(pasta, 2, 4600)`, etc.

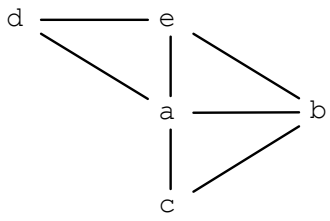
7. Um outro problema que é notoriamente difícil é o problema da cobertura de conjuntos. Você recebe duas listas: `Set` e `Subsets`. A primeira lista, `Subsets` é uma lista de listas, cada uma com uma sendo uma subsequência de `Set`. O problema é então encontrar o conjunto `Cover`: uma cobertura mínima, isto é, uma subsequência de `Subsets` com a propriedade que todo elemento de `Set` seja um elemento de alguma lista de `Cover`. Por exemplo, assuma que `Set` é a lista `[1, 2, 3, 4, 5]` e `Subsets` é a lista `[[1, 2], [2, 4], [3, 5], [1, 3], [3, 4, 5]]`. Então a cobertura mínima seria a lista `[[1, 2], [3, 4, 5]]`.

   (a) Escreva um predicado `CoverDecision(Set, Subsets, Goal, Cover)`, que receba uma lista `Set`, uma lista `Subsets` contendo subsequências de `Set`, e um inteiro positivo `Goal`. Este predicado deve unificar `Cover` com a subsequência de `Subsets` que cubra `Set` e tenha tamanho menor ou igual `Goal`. O predicado falha se não houver nenhuma cobertura possível. A sua solução deverá ser capaz de produzir todas as coberturas que satisfazem `Goal`.

   (b) Escreva um predicado `coverOptimization(Set, Subsets, Cover)` que receba uma lista `Set` e uma lista `Subsets`, como anteriormente. Este predicado deve unificar `Cover` com a subsequência da lista `Subsets` que cubra `Set`, e que tenha tamanho mínimo. O predicado falhará se não houver alguma sequência assim. A sua solução deverá ser capaz de gerar todas as sequências de tamanho mínimo.

8. Uma *clique* é um grafo completo. O problema de decidir se um grafo G possui uma clique de tamanho N é um problema NP-completo bem conhecido. Esse problema, inclusive, é parte da lista de 21 problemas proposta por Richard Karp em 1972. Embora não conheçamos qualquer algoritmo eficiente para encontrar cliques em grafos, é muito fácil resolver esse problema por força bruta em Prolog. Nesse caso, podemos representar um grafo como um conjunto de arestas, conforme feito na figura abaixo:

```
edge(a, b).
edge(a, c).
edge(b, c).
edge(a, d).
edge(b, e).
edge(d, e).
edge(a, e).
```



Cliques:

```
a, b, c

a, b, e

a, e, d
```

O predicado `cliqueN`, definido abaixo, é verdade quando `G` é uma lista de vértices de um grafo, `N` é um número inteiro, e `L` é uma sublista de `G` que forma uma clique:

```
cliqueN(N, G, L) :- sublist(G, L), length(L, N), clique(L).
```

Por exemplo:

```
?- consult(clique).
% clique compiled 0.00 sec, 64 bytes
true.

?- cliqueN(3, [a, b, c, d, e], L).
L = [a, b, c] ;
L = [a, b, e] ;
L = [a, d, e] ;
false.
```

(a) Defina o predicado `sublist(G, L)`, que seja verdade quando `L` for uma sublista de `G`.

(b) Defina o predicado `clique(L)`, que seja verdade quando `L` for uma lista de vértices que forme um grafo completo.