CAPÍTULO 8

Classificação e ordenação

Este capítulo complementa o capítulo sobre processamento de listas. São apresentados diversos exemplos de algoritmos de ordenação fazendo uso das operações definidas para listas. Em geral, estes algoritmos não têm um valor prático, em termos de eficiência, para serem usados com o propósito de ordenar um grande conjunto de dados. Um deles, o método por intercalação, é normalmente programado como o predicado de uso geral que vem embutido em sistemas Prolog (chamado de `sort/2`).

O estudo destes algoritmos visa ao exercício de técnicas de programação em problemas práticos. São apresentados sete métodos de ordenação: ordenação por permutação, por inserção direta, por seleção, por troca, por intercalação (2 métodos) e por partição e troca.

Nestes métodos são revistos vários algoritmos sobre listas e apresentados alguns novos, tais como: permutação, testar se uma lista está ordenada, inserir um elemento numa lista ordenada mantendo a ordem, selecionar o maior elemento de uma lista, trocar dois elementos contíguos; particionar uma lista em duas listas de mesmo tamanho, intercalar duas listas já ordenadas em uma lista ordenada, etc.

No final do capítulo, dedicamos uma seção ao teste de performance de predicados, no qual são comparadas as performances de alguns algoritmos de ordenação, de acordo com o número de inferências e o tempo em segundos. Uma inferência corresponde a uma operação da máquina abstrata do Prolog (equivale grosseiramente a uma operação de unificação).

8.1 Ordenação por permutação

A ordenação por permutação é um método declarativo de ordenação. Ele usa a estratégia de solução de problemas chamada *geração e teste*. O objetivo deste método é verificar, por meio de testes, se cada permutação gerada está ordenada. Do conjunto de todas as permutações sabemos que pelo menos uma delas está ordenada, i.e., é a solução.

```
1  permOrd(L,S):-permutation(L,S),isOrdered(S).
2
3  isOrdered([]).
4  isOrdered([_]).
5  isOrdered([X,Y|XYs]):-X=<Y,isOrdered([Y|XYs]).
6
7  select(X,[X|Xs],Xs).
8  select(X,[Y|Xs],[Y|Zs]):-select(X,Xs,Zs).
9
10 permutation(Xs,[Z|Zs]):-select(Z,Xs,Ys),permutation(Ys,Zs).
11 permutation([],[]).
```

O predicado `permOrd/2` ordena a lista L em S. Os predicados de manipulação de listas foram definidos no capítulo sobre listas: `permutation/2` gera todas as permutações de uma lista, enquanto `isOrdered/1` testa se a lista está ordenada.

```
?- isOrdered([2,3,7,7]).
   Yes
?- isOrdered([3,2,7]).
   No
?- permutation([5,2,4],L).
   L= 5,2,4;
   L= 5,4,2;
   L= 2,5,4;
   ...
?- permOrd([5,2,4],L).
  L=[2,4,5]
```

8.2 Inserção direta

O método de classificação por inserção direta sistematicamente insere os elementos da lista de entrada, um a um, numa nova lista ordenada.

Há dois procedimentos para programá-lo: com acumulador e sem acumulador. Ambos fazem uso de um predicado `insOrd/3`, que insere um elemento em uma lista mantendo-a ordenada. O predicado `insOrd/3` é definido por três regras:

- `insOrd` de um elemento `X` numa lista vazia é retornar `[X]`;
- `insOrd` de um elemento `X` numa lista com cabeça `Y`, se `X=<Y` resulta em inserir `X` antes de `Y`;
- caso contrário, se `X>Y`, chama-se recursivamente `insOrd/2` para inserir `X` na posição correta na cauda da lista.

```
1 insOrd(X,[Y|L],[X,Y|L]) :- X =< Y,!.
2 insOrd(X,[Y|L],[Y |Io]) :- X >  Y,!, insOrd(X,L,Io).
3 insOrd(X,[],[X]).
```

```
?- insOrd(4, [2,3,5,7],L).
   L=[2,3,4,5,7]
```

Neste programa, a cláusula base `insOrd(X,[],[X])` é a última do predicado `insOrd` e só é executada uma vez, quando o processo termina. Se ela for a primeira regra do predicado, será testada a cada iteração, apesar de ser executada apenas quando a lista estiver vazia. Nesta posição, ela é testada e executada uma única vez, trazendo eficiência ao predicado.

Com acumulador

Dada uma lista de entrada L, toma-se o elemento da cabeça de L para ser inserido em uma lista ordenada, representada no acumulador AC. Quando este processo termina, o acumulador ACC é devolvido como a solução. O processo codificado no predicado `insDirAcc/2` é exemplificado abaixo.

L-entrada	ACC	S-saída
7 4 8 3 8	[]	
4 8 3 8	7	
8 3 8	4 7	
3 8	4 7 8	
8	2 4 7 8	
[]	3 4 7 8 8	3 4 7 8 8

Sem acumulador

Pega-se a cabeça da lista de entrada e chama-se o algoritmo para ordenar a cauda, então insere-se a cabeça na posição correta, na cauda que já está ordenada. Este método é codificado em `insDir/2`.

```
1 insDirAcc(L,S):- insDirAcc(L,[],S).
2 insDirAcc([C|Ls],ACC,   S):-insOrd(C,ACC,ACC1),insDirAcc(Ls,ACC1,S).
3 insDirAcc([],    ACC, ACC).
4 %
5 insDir([C|Ls],So) :- insDir(Ls,Si),insOrd(C,Si,So).
6 insDir([],[]).
```

```
?-insDir([7,4,8,3,8],S)
   S=[3,4,7,8,8]
?-insDirAcc([7,4,8,3,8],S)
   S=[3,4,7,8,8]
```

8.3 Ordenação por seleção

Na ordenação por seleção, o maior elemento de uma lista de entrada é selecionado e inserido como cabeça de uma lista onde a solução é acumulada. Este método é similar ao da inserção direta, porém, neste o processamento ocorre na seleção do elemento a ser removido, enquanto que naquele o processamento ocorre na inserção do elemento.

L-entrada	S-saída
7 4 8 3 8	[]
7 4 3 8	8
7 4 3	8 8
4 3	7 8 8
3	4 7 8 8
[]	3 4 7 8 8

```
selecaoOrd(L,[M|S]):-remMax(M,L,Lo), selectOrd(Lo,S).
selecaoOrd([],[]).
%
maxL([X],X).
maxL([X|Xs],M):-max(Xs,M1),(X>M1,M=X; X=<M1,M=M1).
%
remMax(M,L,Lo):-maxL(M,L),select(M,L,Lo).
```

Os predicados maxL/2 e select/3 foram apresentados no capítulo sobre listas.

8.4 Ordenação por troca (bublle sort)

O método de ordenação por troca examina sistematicamente dois elementos contíguos na lista de entrada e se o elemento da direita for menor que o elemento da esquerda, eles são trocados de posição. O processo termina quando não há mais trocas a serem feitas.

```
trocaOrd(L,S):-
              append(ORD,[A,B|Ls],L), B<A,!,    %% tira A,B
              append(ORD,[B,A|Ls],Li),          %% troca B,A
              trocaOrd(Li,S).
trocaOrd(L,L).
```

```
?- L=[1,2,6,4,7], append(ORD,[A,B|Ls],L),B<A.
   ORD=[1,2], A=6, B=4, Ls=[7]
?- trocaOrd([1,2,6,4,7],L).
   L=[1,2,4,6,7]
```

Aqui usamos o append/3 para extrair dois elementos contíguos de uma lista. Sistematicamente são destacados dois elementos da lista, A e B, e, quando B<A, eles são trocados de posição usando append/3., como:

```
?- append(_,[A,B|_],[2,7,4,5]).
A = 2, B = 7 ;
A = 7, B = 4 ;
A = 4, B = 5 ; No
```

8.5 Ordenação por intercalação

A estratégia é particionar uma lista em duas, ordenando isoladamente cada uma delas e intercalando os resultados. No particionamento, se a lista de entrada tem um número par de elementos, as duas listas de saída terão o mesmo número de elementos. Se for impar, uma das listas terá um elemento a mais. Particionar consiste em tomar dois elementos da cabeça da lista e colocar cada um em uma das partições.

```
particiona([],[],[]).
particiona([X],[X],[]).
particiona([X,Y|XYs], [X|Xs],[Y|Ys]):-particiona(XYs,Xs,Ys).
%%
merge([A,B|Ls],S):-
    particiona([A,B|Ls],La,Lb),
    merge(La,Las),merge(Lb,Lbs),
    intercalar(Las,Lbs,S).
merge([X],[X]).
merge([],[]).
%
intercalar([A|As],[B|Bs], [A|ABs]):-A=<B, intercalar(As,[B|Bs],ABs).
intercalar([A|As],[B|Bs], [B|ABs]):-A >B, intercalar([A|As],Bs,ABs).
intercalar([],Bs,Bs).
intercalar(As,[],As).
```

Intercalar duas listas, já ordenadas, consiste em escolher a lista que tem o menor elemento na cabeça, assumir este elemento como a cabeça do resultado e chamar recursivamente o processo para obter a cauda do resultado.

```
?- particiona( [6,2,1,4],L, M).
   L = [6,1], M = [2,4]
?- intercalar( [1,2], [4], L).
   L = [1,2,4]
?- merge([1,2,6,4,7],L).
   L = [1,2,4,6,7]
```

Exercício 8.1 *Examine o particiona/3 e responda. Quando temos um número ímpar de elementos na lista de entrada, qual das duas partições terá um elemento a mais? Justifique.*

Exercício 8.2 *Reescreva* `intercalar` *usando um comando* `if-then`.

Solução:

```
intercalar([A|As],[B|Bs],R):- (( A=<B -> R=[A|ABs],intercalar(As,[B|Bs],ABs);
                                 A >B -> R=[B|ABs],intercalar([A|As],Bs,ABs)   )).
```

8.6 Ordenação por intercalação (otimizada)

O Prolog possui um algoritmo de ordenação embutido, implementado como um algoritmo de ordenação por intercalação. Porém, no Prolog, o trabalho de particionar explicitamente uma lista

exige muito tempo e recursos de memória. Por isso, é possível simular o particionamento da lista a partir da informação do número de elementos.

O Prolog oferece um algoritmo para ordenação em que cada elemento de uma lista é associado a uma chave de ordenação. Assim, os valores a serem ordenados são organizados como nos testes apresentados abaixo. A chave é o primeiro elemento do par (uma chave pode ser qualquer constante do Prolog, incluindo termos compostos). O segundo elemento do par também pode ser qualquer termo. É possível, por exemplo, ordenar uma lista de registros de pessoas usando como chaves o nome delas.

```
?- sortKey([1-f,5-b,1-c,6-d,15-e,3-d],S).
  S = [1 - f,1 - c,3 - d,5 - b,6 - d,15 - e]
%
?- sortKey([f-1,b-5,c-1,d-6,e-15,d-3],S).
  S = [b - 5,c - 1,d - 6,d - 3,e - 15,f - 1]
```

Segue o código para o algoritmo de ordenação:

```
1  sortKey(List, Sorted) :-
2      length(List, Length),
3      sortKey(Length, List/ _, Result),
4      Sorted = Result.
5  sortKey(2, [X1, X2|L]/L, R) :- !,
6      X1 = K1-_,
7      X2 = K2-_,
8      (( K1 @=< K2 ->  R = [X1, X2]
9                   ;   R = [X2, X1]   )).
10 sortKey(1, [X|L]/L, [X]) :- !.
11 sortKey(0, L/L, []) :- !.
12
13
14 sortKey(N, L1/L3, R) :-
15     N1 is N // 2, N2 is N - N1,
16     sortKey(N1, L1/L2, R1),
17     sortKey(N2, L2/L3, R2),
18     mergeKey(R1, R2, R).
19 mergeKey([], R, R) :- !.
20 mergeKey(R, [], R) :- !.
21 mergeKey(R1, R2, [X|R]) :-
22     R1 = [X1|R1a], X1 = K1-_,
23     R2 = [X2|R2a], X2 = K2-_,
24  ((  K1 @> K2 ->  X = X2, mergeKey(R1, R2a, R)
25                  ;   X = X1, mergeKey(R1a, R2, R)    )).
```

Neste algoritmo, a maior dificuldade está em ter que percorrer toda a lista para calcular o seu comprimento. A partir do comprimento, faz-se a partição implícita pela divisão do valor N por dois, em `N1 is N//2` e `N2 is N-N1`. Assim, as duas partições terão comprimento N1 e N2. Enquanto as partições são maiores que dois, o algoritmo é recursivamente chamado. Quando uma partição é menor ou igual a dois, os valores são efetivamente removidos da lista de entrada. Cada subchamada `sortKey(N, Li/Lo, S)` pega N elementos da lista Li e deixa os demais elementos em Lo. No final da subchamada, os N elementos estão ordenados em S.

Exercício 8.3 *Refaça o algoritmo acima, chame-o de* `xsortKey/2`*, para trabalhar com uma lista simples de elementos sem os pares (chave - termo).*

8.7 Ordenação por partição e troca (Quick sort)

Outro algoritmo eficiente para ordenação de vetores de elementos é conhecido como quick sort. Dada uma lista de entrada, seleciona-se um elemento como pivô (neste exemplo é assumido o primeiro elemento da lista). Tomando-se o pivô, particiona-se a lista em duas, uma com todos os elementos menores que o pivô e outra com os maiores ou iguais. Este processo serve para ordenar ambas as listas, a dos menores e a dos maiores. Sabe-se que o elemento pivô deve ficar no meio dos dois resultados: menores, pivô, maiores.

`partic(Lista, Pivo, Lm(enores), LM(aiores))` seleciona todos os menores do elemento pivô para `Lm` e os demais para `LM`. Por exemplo, se a entrada for [7,4,8,3,8], o elemento pivô é o 7 e as listas são Lm=[3,4] e LM =[8,8].

```
1 partic([X|L],Pivo,[X|Lm],LM):- X< Pivo,!,partic(L,Pivo,Lm,LM).
2 partic([X|L],Pivo,Lm,[X|LM]):- X>=Pivo,!,partic(L,Pivo,Lm,LM).
3 partic([],_,[],[]).
```

```
?- partic([4,8,3,8],7,Lm,LM).
   Lm = [4, 3], LM = [8, 8]
```

Para obter o resultado, ordena-se Lm e LM em Sm=[3,4] e SM=[8,8], junta-se Sm, pivo e SM, nesta ordem, resultando em [3,4,7,8,8].

Para o quick sort são apresentadas três versões:

- a primeira, `qsort/2`, usa o append/3 para concatenar as listas;
- a segunda, `qs_acc`, usa um *acumulador*, de forma que o append não é necessário;
- a terceira, `qs_dl`, usa *diferença de listas* para implicitamente executar a concatenação de listas.

A técnica de uso de acumulares foi vista em vários programas e a da diferença de listas foi apresentada no capítulo sobre estruturas de dados.

Na versão `qsort` podemos ler o corpo principal como: particiona-se a lista de entrada `[X|Xs]` em duas, Lm (menores) e LM (maiores), do elemento pivô `X`. Chama-se o `qsort` de cada uma delas e, com o append, unem-se os resultados, com o cuidado de colocar o elemento pivô no meio do resultado `Lm+X+LM`.

Na versão com acumulador, `qs_acc`, o predicado auxiliar `qs_acc1` faz todo o processamento. Os valores são acumulados dos menores em direção aos maiores, de forma que a lista resultante fique em ordem crescente.

Inicialmente, a lista é particionada em duas. A seguir, chama-se o predicado `qs_acc1` para cada uma das partições. Depois, chamando o `qs_acc1`, ordenamos cada uma das listas, iniciando pela lista dos maiores. Um acumulador é usado para juntar os dois resultados parciais. O elemento-pivô é colocado no acumulador antes da lista ordenada dos maiores.

A versão com diferença de listas é bem próxima à versão com acumulador. Porém, a estratégia da solução é um pouco diferente, talvez mais fácil. O `qs_dl1` de uma lista é a diferença de listas `Si/So` se o `qs_dl1` da lista dos menores é `Si/[X|SM]` e o `qs_dl1` da lista dos maiores é `SM/So`.

Aqui a cauda do resultado dos menores é ligada ao resultado dos maiores. O elemento-pivô fica no meio.

As soluções com acumulador e diferença de listas são similares, mas cada uma usa a sua estratégia particular: com o acumulador guardam-se resultados intermediários, dos maiores para os menores, e sempre o resultado é incluído na cabeça do acumulador; com a diferença de listas, vemos o resultado como listas parciais, uma ligada na cauda da outra.

```
qsort([X|Xs],S):-
    partic(Xs,X,Lm,LM),qsort(Lm,Sm),qsort(LM,SM),append(Sm,[X|SM],S).
qsort([],[]).
%%
qs_acc(L,S) :- qs_acc1(L, []/S).
qs_acc1([X|L],Acc/S) :-
     partic(L,X,Lm,LM),qs_acc1(LM,Acc/SM),qs_acc1(Lm,[X|SM]/S).
qs_acc1([],S/S).
%%
qs_dl(L,S) :- qs_dl1(L, S/[]).
qs_dl1([X|L],Si/So) :-
     partic(L,X,Lm,LM),qs_dl1(Lm,Si/[X|SM]),qs_dl1(LM,SM/So).
qs_dl1([],S/S).
```

Seguem exemplos de execução:

```
?- qsort([1,6,2,7],S).
   S = [1, 2, 6, 7]
?- qs_acc([1,2,6,1,7,1,0,11,-1],K).
   K = [-1,0,1,1,1,2,6,7,11]
?- qs_dl([1,2,6,1,7,1,0,11,-1],K).
   K = [-1,0,1,1,1,2,6,7,11]
```

8.8 Medindo a performance de programas

Neste capítulo vimos diversos algoritmos de ordenação. A seguir, vamos avaliar a performance destes algoritmos. No SWI-Prolog temos um predicado `time/1` que retorna o tempo gasto para execução de um predicado, bem como o número de inferências lógicas que foram executadas pela máquina abstrata do Prolog (todo sistema Prolog tem um predicado similar a este). Esta segunda medida é bem significativa para avaliar a performance de um programa Prolog.

Inicialmente definimos três listas como três fatos com, respectivamente, 25, 50 e 100 elementos. Podemos também usar o predicado `lista_rand/3` que gera uma lista de valores randômicos (foi estudado no capítulo sobre programação aritmética).

Precisamos também de um comando de repetição `repeatN/2` para executarmos o algoritmo várias vezes, pois para apenas 100 elementos o tempo será sempre 0.0 segundos.

```
lista_rand(   [],V,N):-N<1,!.
lista_rand([R|L],V,N):-N1 is N-1, R is random(V), lista_rand(L,V,N1).
%%
repeatN(0,_):-!.
repeatN(N,Call):-call(Call),N1 is N-1,repeatN(N1,Call).
```

```
%%
lista100r(L):-lista_rand(L,50,100).
%%
lista25([1,2,6,1,7,1,0,11,-11,2,6,1,-11,2,6,1,7,1,0,11,-11,2,6,1,7]).
lista50([1,2,6,1,7,1,0,11,-11,2,6,1,-11,2,6,1,7,1,0,11,-11,2,6,1,7,
         1,2,6,1,7,1,0,11,-11,2,6,1,-11,2,6,1,7,1,0,11,-11,2,6,1,7]).
lista100([11,-11,2,6,1,7,1,0,11,-11,2,6,1,7,1,0,11,-11,2,6,1,7,1,0,11,-1,
1,2,6,1,7,1,0,11,... ]).
```

Seguem alguns predicados de testes para a lista de 25 e 50 elementos. Os predicados `t01`, `t02` e `t03` apenas verificam se os algoritmos estão funcionando.

```
t01 :- lista25(L),write('qsort:'), qsort(L,S),write(S).
t02 :- lista25(L),write('qs_dl:'), qs_dl(L,S),write(S).
t03 :- lista25(L),write('qs_acc:'), qs_acc(L,S),write(S).
%
tm1 :- lista25(L),write('qsort:'), time(qsort(L,S)).
tm2 :- lista25(L),write('qs_dl:'), time(qs_dl(L,S)).
tm3 :- lista25(L),write('trocaOrd'), time(trocaOrd(L,S)).
%%
tm15 :- lista50(L),write('qsort:'), time(qsort(L,S)).
tm25 :- lista50(L),write('qs_dl:'), time(qs_dl(L,S)).
tm35 :- lista50(L),write('trocaOrd'), time(trocaOrd(L,S)).
```

Seguem os números para os testes com listas de 25 e 50 elementos (`tm?` e `tm?5`). O número de inferências para os algoritmos de quicksort cresce ligeiramente quando duplicamos o tamanho da lista. Porém, para o algoritmo de ordenação por troca `trocaOrd`, o valor é 8 vezes maior, saltou de 5,595 para 46,646 inferências.

```
?- t01.
    qsort:[-11, -11, -11, 0, 0, 1, 1, 1, 1, 1,...]
?- t02.
    qs_dl:[-11, -11, -11, 0, 0, 1, 1, 1, 1, 1,...]
?- t03.
    qs_acc:[-11, -11, -11, 0, 0, 1, 1, 1, 1, 1,...]
%%
?-tm1.
  % 447 inferences in 0.00 seconds (Infinite Lips)
   qsort:
?- tm2.
  % 401 inferences in 0.00 seconds (Infinite Lips)
  qs_dl:
?- tm3.
  % 5,595 inferences in 0.00 seconds (Infinite Lips)
   trocaOrd
%%
?-tm15.
    % 1,333 inferences in 0.00 seconds (Infinite Lips)
    qsort:
?- tm25.
```

```
   % 1,236 inferences in 0.00 seconds (Infinite Lips)
    qs_dl:
?- tm35.
    % 46,646 inferences in 0.11 seconds (424055 Lips)
    trocaOrd
```

Seguem os testes para alguns dos algoritmos de classificação, para listas de 100 elementos, primeiro com a lista codificada como fato e depois para a lista gerada com valores randômicos. O objetivo é saber qual o mais rápido e se a geração dos valores randômicos afeta o resultado.

```
1  tm10 :- lista100(L),write('qsort:'), time(repeatN(100,qsort(L,S))).
2  tm20 :- lista100(L),write('qs_dl:'), time(repeatN(100,qs_dl(L,S))).
3  tm30 :- lista100(L),write('qs_acc:'), time(repeatN(100,qs_acc(L,S))).
4  tm40 :- lista100(L),write('ins_dir:'), time(repeatN(100,insDir(L,S))).
5  tm50 :- lista100(L),write('sort:'), time(repeatN(100,sort(L,S))).
6  tm60 :- lista100(L),write('xsortKey'), time(repeatN(100,xsortKey(L,S))).
7  %%
8  tm10r :- lista100r(L),write('qsort:'), time(repeatN(100,qsort(L,S))).
9  tm20r :- lista100r(L),write('qs_dl:'), time(repeatN(100,qs_dl(L,S))).
10 tm30r :- lista100r(L),write('qs_acc:'), time(repeatN(100,qs_acc(L,S))).
11 tm40r :- lista100r(L),write('ins_dir:'), time(repeatN(100,insDir(L,S))).
12 tm50r :- lista100r(L),write('sort:'), time(repeatN(100,sort(L,S))).
13 tm60r :- lista100r(L),write('xsortKey'), time(repeatN(100,xsortKey(L,S))).
```

Abaixo, temos alguns resultados dos testes. O teste `tm10` mostra que o quick sort com uma lista randômica é mais eficiente. O algoritmo de inserção direta `tm40` se estende três vezes mais que o quicksort. O algoritmo embutido no Prolog `sort/2` é o mais rápido, pois, provavelmente está compilado em assembler. Os demais são executados na máquina abstrata do Prolog. O valor em Lips (Logic Inferences Per Second) deveria estar mais ou menos estável para todos os testes, pois define a performance de um sistema Prolog numa determinada máquina, o que aconteceu com exceção para o algoritmo `sort` embutido no Prolog.

```
?-tm10.
% 434,601 inferences in 0.66 seconds (658486 Lips)
?- tm10r.
% 284,801 inferences in 0.44 seconds (647275 Lips)
qsort:

?- tm40.
% 874,175 inferences in 1.49 seconds (586695 Lips)
?- tm40r.
% 960,231 inferences in 1.59 seconds (603919 Lips)
ins_dir:

?- tm50.
% 401 inferences in 0.06 seconds (6683 Lips)

?- tm50r.
% 401 inferences in 0.06 seconds (6683 Lips)
sort:
```

```
?- tm60.
% 272,701 inferences in 0.50 seconds (545402 Lips)
xsortKey
?- tm60r.
% 272,201 inferences in 0.49 seconds (555512 Lips)
xsortKey
```

Este último predicado `tm60` testa o algoritmo `xsortKey`, uma versão simples do `sortKey`, solicitado como exercício.