Sort

Quick Sort

- 구현

  퀵정렬 알고리즘은 리스트 L을 다음과 같이 정렬한다 :

  1. 리스트 L에서 특정 원소 N(called the pivot)을 선택한다.
  2. N보다 작은 모든 원소들의 리스트를 재귀적으로 정렬한다.
  3. N보다 큰 모든 원소들의 리스트를 재귀적으로 정렬한다.
  4. 다음의 리스트들을 연결(concatenate)한다:
     - 2단계에서 얻은 리스트
     - 리스트 L에서 N과 같은 모든 원소들의 리스트
     - 3단계에서 얻은 리스트

  피벗은 리스트 내 임의의 원소가 될 수 있다. 참고문헌 [52]에서는 “예를 들어, 피벗 N을 마지막 원소로 두자”라고 설명한다. 아래의 Maude 정의는 교재의 설명보다 더 일반적인 규격(specification)을 제공하는데, 피벗 N을 “예를 들어 마지막 원소”처럼 강제하지 않고 비결정적(nondeterministic)으로 선택하기 때문이다.

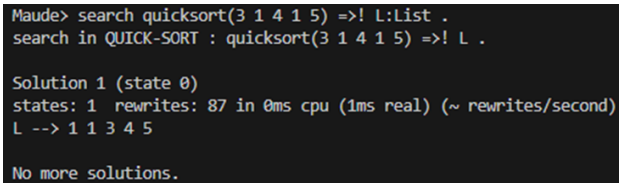
```
fmod QUICK-SORT is protecting LIST-INT .
   op quicksort : List → List .

   vars L L' : List .    vars M N : Int .

   eq quicksort(nil) = nil .
   eq quicksort(L N L') =
   quicksort(smallerElements(L L', N))
   equalElements(L N L', N)
   quicksort(greaterElements(L L', N)) .
```

  여기서 `smallerElements(l, n)` 은 리스트 l에서 n보다 작은 원소들을 포함한다.

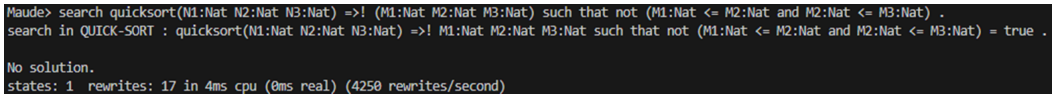
```
  ops smallerElements greaterElements equalElements : List Int → List .

  eq smallerElements(nil, N) = nil .
  eq smallerElements(N L, M) = if N < M then
                                 (N smallerElements(L, M))
                               else smallerElements(L, M) fi .

  eq equalElements(nil, N) = nil .
  eq equalElements(N L, M) = if N == M then (N equalElements (L, M))
                               else equalElements(L, M) fi .

  eq greaterElements(nil, N) = nil .
  eq greaterElements(N L, M) = if N > M then
                                 (N greaterElements(L, M))
```

```
                    else greaterElements(L, M) fi .
endfm
```

- 검증
  - 비결정적 피벗 선택 시에도 결과가 유일함을 확인
    - 피벗 N은 리스트 어디든 올 수 있어서 비결정적임
    - search로 “quicksort 결과가 한 가지 뿐인지 '' 를 확인해 봄



    - =>! : 더 이상 규칙이 적용되지 않은 정규형까지 탐색
    - 결과가 해 한 개만 나왔으므로, 피벗을 어디로 잡든지 결과 정렬 리스트는 유일한 것을 알 수 있음
  - 길이 3짜리 리스트에 대해 “항상 정렬된 결과가 나온다”는 성질 확인



    - no solution이므로 임의의 세 정수에 대해 quicksort 결과는 항상 M1 <= M2 <= M3를 만족함

Merge Sort

- 구현

  리스트 L을 정렬하는 병합정렬 알고리즘은 다음과 같다 :

  1. 리스트 L에 최소 두 개 이상의 원소가 있다면(그렇지 않다면 할 일이 없다), 리스트 L을 대략 같은 크기의 두 “부분 리스트” L_1과 L_2로 나눈다.
  2. 부분 리스트 L_1과 L_2를 재귀적으로 정렬한다.
  3. 2단계에서 얻은 두 리스트를 병합(merge)한다.

  병합정렬은 아래와 같이 Maude로 명세할 수 있다 :

```
fmod MERGE-SORT is protecting LIST-INT .
   op mergeSort : List → List .
   op merge : List List → List [comm] .

   vars L L' : List .
   vars NEL NEL' : NeList .
   vars I J : Int .

   eq mergeSort(nil) = nil .
```

```
    eq mergeSort(I) = I .
    ceq mergeSort(NEL NEL') = merge(mergeSort(NEL), mergeSort(NEL'))
      if length(NEL) == length(NEL') or length(NEL) == s length(NEL') .

    eq merge(nil, L) = L .
    ceq merge(I L, J L') = I merge(L, J L') if I <= J .
endfm
```

병합정렬의 존재 이유는 실행 시간이 (O(n log n))이기 때문이다.

위 명세(specification)는 리스트를 두 절반으로 나누기 위해 패턴 매칭을 사용하기 때문에 실제 구현보다 덜 효율적일 수 있다.

그러나 이러한 종류의 명세는 복잡한 알고리즘을 **정확하게 서술**할 수 있다는 장점이 있으며, 동시에 알고리즘을 **빠르게 테스트하고** 추가로 분석하기 위한 **프로토타입** 역할을 한다는 점에서 유용하다.

즉, 전체적으로 효율적인 구현을 완성하기 전에, 초기 단계에서 알고리즘을 검증하고 분석하기 위한 초안으로 사용할 수 있다.

- 검증
  - 결과 유일성 확인

```
Maude> search mergeSort(3 1 4 1 5) =>! L:List .
search in MERGE-SORT : mergeSort(3 1 4 1 5) =>! L .

Solution 1 (state 0)
states: 1  rewrites: 11 in 0ms cpu (0ms real) (~ rewrites/second)
L --> mergeSort(3 1 4 1 5)

No more solutions.
states: 1  rewrites: 11 in 0ms cpu (0ms real) (~ rewrites/second)
```

    - Solution 1 하나만 나오고 "No more solutions."이므로, 같은 입력에 대해 정규형이 하나뿐이라는 것을 볼 수 있음
  - 정렬이 깨진 경우가 있는지 확인

```
Maude> search mergeSort(Lpre) =>! (M1 M2 Lpost) such that M1 > M2 .
search in MERGE-SORT : mergeSort(Lpre) =>! M1 M2 Lpost such that M1 > M2 = true .

No solution.
states: 1  rewrites: 0 in 0ms cpu (0ms real) (~ rewrites/second)
```

    - No solution이므로 merge sort 결과 리스트에는 인접한 두 원소가 M1 > M2인 경우(내림차순)가 한 번도 없다는 것을 알 수 있음

Insertion Sort

Exercise 37 *Specify the* insertion sort *algorithm in Maude. Insertion sort works as when you get some cards and have to sort them: you take the (unsorted) cards one by one, and put them into the right place in your hand, which always remains sorted.*

삽입 정렬 알고리즘을 Maude 로 명세하시오. 삽입 정렬은 카드 정렬에 비유할 수 있다. 정렬되지 않은 카드를 한 장씩 집어서, 항상 정렬된 상태를 유지하고 있는 자신의 손패에서 알맞은 위치에 끼워 넣는 방식으로 동작한다.

- 구현
  - `insertSort : List → List`
    - 리스트의 첫 원소를 하나 꺼내서
    - 나머지 부분을 재귀적으로 정렬한 뒤
    - 보조 함수 `insert`를 이용해 **이미 정렬된 리스트에 올바른 위치로 끼워 넣기**.
  - `insert : Int List → List`
    - 정렬된 리스트를 왼쪽부터 보면서
      - 새 원소 `I`가 현재 원소 `J`보다 작거나 같으면 그 위치에 끼워 넣고
      - 그렇지 않으면 `J`는 그대로 두고 리스트의 나머지에 대해 재귀 호출.

```
fmod INSERTION-SORT is
    protecting LIST-INT .

    --- 정렬 함수
    op insertionSort : List → List .

    --- 정렬된 리스트에 한 원소를 끼워 넣는 함수
    op insert : Int List → List .

    vars I J : Int .
    var L : List .

    --- 빈 리스트는 이미 정렬되어 있음
    eq insertSort(nil) = nil .

    --- 첫 원소 I를 떼어 내고, 나머지 L을 재귀적으로 정렬한 뒤
    --- 정렬된 리스트에 I를 끼워 넣는다.
    eq insertionSort(I L) = insert(I, insertSort(L)) .

    --- 빈 리스트에 I를 끼워 넣으면, 리스트는 I 한 개뿐
    eq insert(I, nil) = I .

    --- I <= J 인 경우, I를 앞에 놓으면 정렬 조건을 유지
    ceq insert(I, J L) = I J L if I <= J .

    --- I > J 인 경우, J를 유지하고 나머지 L에 I를 삽입
    ceq insert(I, J L) = J insert(I, L) if I > J .
endfm
```

- 검증

- 결과 유일성 확인

```
Maude> search insertionSort(3 1 4 1 5) =>! L:List .
search in INSERTION-SORT : insertionSort(3 1 4 1 5) =>! L .

Solution 1 (state 0)
states: 1  rewrites: 4 in 0ms cpu (0ms real) (~ rewrites/second)
L --> insert(3, insert(1, insert(4, insert(1, insertionSort(5)))))

No more solutions.
states: 1  rewrites: 4 in 0ms cpu (0ms real) (~ rewrites/second)
```

  - Solution 1 하나만 나오고 “No more solutions.”이므로, 같은 입력에 대해 정규형이 하나뿐이라는 것을 볼 수 있음

- 정렬이 깨진 경우가 있는지 확인

```
Maude> search insertionSort(Lpre) =>! (M1 M2 Lpost) such that M1 > M2 .
search in INSERTION-SORT : insertionSort(Lpre) =>! M1 M2 Lpost such that M1 > M2 = true .

No solution.
states: 1  rewrites: 0 in 0ms cpu (0ms real) (~ rewrites/second)
```

  - No solution이므로 insertion sort 결과 리스트에는 인접한 두 원소가 M1 > M2인 경우(내림차순)가 한번도 없다는 것을 알 수 있음