

프로그래밍 언어

hw5

컴퓨터공학과

B711187

지승연

1. 하노이의 탑

알고리즘
1. 원반이 한 개면 그냥 옮긴다. (종료 조건) 2. 원반이 n 개면 1) 1번 기둥에 있는 n 개 원반 중 $(n-1)$ 개를 3번 기둥으로 옮긴다. (2번 기둥을 보조 기둥으로 사용) 2) 1번 기둥에 남아있는 가장 큰 원반을 2번 기둥으로 옮긴다. 3) 3번 기둥에 있는 $(n-1)$ 개 원반을 다시 2번 기둥으로 옮긴다. (1번 기둥을 보조 기둥으로 사용)

출처: <https://brunch.co.kr/@younggiseo/139>

prolog 코드	
hanoi(N) :- recursive(N, 1, 3, 2).	재귀함수에 기둥 정보가 들어가 있어야 하기 때문에 따로 함수를 만든다. recursive(N, From, Tmp, To) N: 총 원반 개수 From: 시작 기둥 Tmp: 보조 기둥 To: 최종적으로 원반이 옮겨갈 기둥
recursive(N, From, Tmp, To) :- N:=1 -> write("1->["), write(From), write(','), write(To), write(']'), nl;	알고리즘1. 원반 개수N이 1일 경우 1번 원반이 From기둥에서 To기둥으로 옮겨간다고 출력한 뒤 끝난다.
N2 is N-1, recursive(N2, From, To, Tmp),	알고리즘2-1). N2라는 변수에 N-1를 저장하고 N2개의 원반을 보조 기둥으로 옮긴다.
write(N), write("->["), write(From), write(','), write(To), write(']'), nl,	알고리즘2-2). 남아있는 원반을 To기둥에 옮겨간다고 출력한다.
recursive(N2, Tmp, From, To).	알고리즘2-3). 보조 기둥에 있던 원반들을 다시 To기둥으로 옮긴다.

전체 코드
<pre> hanoi(N) :- recursive(N, 1, 3, 2). recursive(N, From, Tmp, To) :- N:=1 -> write("1->["), write(From), write(','), write(To), write(']'), nl; N2 is N-1, recursive(N2, From, To, Tmp), write(N), write("->["), write(From), write(','), write(To), write(']'), nl, recursive(N2, Tmp, From, To). </pre>

예시 결과		
	<pre> ?- ['hanoi.pl']. true. ?- hanoi(3). 1->[1,2] 2->[1,3] 1->[2,3] 3->[1,2] 1->[3,1] 2->[3,2] 1->[1,2] true. </pre>	

trace - hanoi(2)		
[trace] ?- hanoi(2).		
Call: (7) hanoi(2) ? creep Call: (8) recursive(2, 1, 3, 2) ? creep Call: (9) 2:=1 ? creep Fail: (9) 2:=1 ? creep Redo: (8) recursive(2, 1, 3, 2) ? creep Call: (9) _G1229 is 2+ -1 ? creep Exit: (9) 1 is 2+ -1 ? creep Call: (9) recursive(1, 1, 2, 3) ? creep	<pre> graph LR A((recursive (2, 1, 3, 2) N != 1)) --> B((recursive (1, 1, 2, 3))) </pre>	
Call: (10) 1:=1 ? creep Exit: (10) 1:=1 ? creep Call: (10) write("1->") ? creep 1-> Exit: (10) write("1->") ? creep Call: (10) write([1, 3]) ? creep [1,3] Exit: (10) write([1, 3]) ? creep Call: (10) nl ? creep Exit: (10) nl ? creep Exit: (9) recursive(1, 1, 2, 3) ? creep	<pre> graph LR A((recursive (2, 1, 3, 2))) --> B((recursive (1, 1, 2, 3) N = 1)) B --> C[1->[1,3]] </pre>	
Call: (9) write(2) ? creep 2 Exit: (9) write(2) ? creep Call: (9) write("->") ? creep -> Exit: (9) write("->") ? creep Call: (9) write([1, 2]) ? creep [1,2] Exit: (9) write([1, 2]) ? creep Call: (9) nl ? creep Exit: (9) nl ? creep	<pre> graph LR A((recursive (2, 1, 3, 2))) --> B[2->[1,2]] </pre>	

<p>Call: (9) recursive(1, 3, 1, 2) ? creep</p> <p>Call: (10) 1:=1 ? creep</p> <p>Exit: (10) 1:=1 ? creep</p> <p>Call: (10) write("1->") ? creep</p> <p>1-></p> <p>Exit: (10) write("1->") ? creep</p> <p>Call: (10) write([3, 2]) ? creep</p> <p>[3,2]</p> <p>Exit: (10) write([3, 2]) ? creep</p> <p>Call: (10) nl ? creep</p> <p>Exit: (10) nl ? creep</p> <p>Exit: (9) recursive(1, 3, 1, 2) ? creep</p>	<pre>graph LR; A((recursive (2, 1, 3, 2))) --> B((recursive (1, 3, 1, 2) N = 1)); B --> C[1->[3,2]]</pre>
<p>Exit: (8) recursive(2, 1, 3, 2) ? creep</p> <p>Exit: (7) hanoi(2) ? creep</p> <p>true.</p>	

2. Quick Sort

알고리즘
1. partition: 맨 왼쪽 숫자를 pivot으로 잡고 pivot보다 작은 수는 왼쪽으로, 큰 수는 오른쪽으로 옮긴다. 2. pivot을 기준으로 왼쪽 혹은 오른쪽 숫자들로 1번을 반복한다. 3. merge: 나뉜 list들을 다시 하나로 병합한다.

prolog 코드	
<pre>partition(PIVOT,[],[],[]) :- !. partition(PIVOT,[A B],[A LEFT],RIGHT) :- A<PIVOT -> partition(PIVOT,B,LEFT,RIGHT). partition(PIVOT,[A B],LEFT,[A RIGHT]) :- A>PIVOT -> partition(PIVOT,B,LEFT,RIGHT).</pre>	<pre>partition(): 빈 list일 경우 아무것도 하지 않는다. (종료 조건) 정렬할 list의 가장 앞 숫자를 A라 할 때, A가 pivot보다 작은 수일 경우 왼쪽 list 파라미터를 [A LEFT]로 받아서 리턴할 경우 A가 LEFT에 포함 되도록 한다. 반대로 A가 pivot보다 클 경우는 오 른쪽 list 파라미터를 [A RIGHT]로 받아 A가 RIGHT에 포함되도록 한다.</pre>
<pre>printDivide(PIVOT,[],[]) :- !. printDivide(PIVOT,LEFT,RIGHT) :- write("divide="), write(PIVOT), write(' '), write(LEFT), write(RIGHT), nl.</pre>	<pre>printDivide(): LEFT와 RIGHT가 어떻게 나뉘었는지 출력하기 위 한 함수. 두 list가 모두 비어 있을 경우 출력하지 않고, 아 니면 출력형식에 맞게 출력한다.</pre>
<pre>printMerge(PIVOT,[],[]) :- write("merge:"), write([PIVOT]), nl. printMerge(PIVOT,LEFT,RIGHT) :- write("merge:"), write(LEFT), write([PIVOT]), write(RIGHT), nl.</pre>	<pre>printMerge(): LEFT,pivot,RIGHT순으로 병합할 list를 출력하기 위한 함수. LEFT와 RIGHT가 모두 비었을 경우 pivot만 list형 식으로 출력한다.</pre>
<pre>quickSort([],[]) :- !.</pre>	<pre>quickSort(): 빈 list일 경우 아무것도 하지 않는다.</pre>
<pre>quickSort([PIVOT B]) :- quickSort([PIVOT B],RESULT), write(RESULT), nl.</pre>	<pre>결과 list를 저장할 RESULT가 포함된 quickSort함 수를 호출하고 최종 결과인 RESULT를 출력한다.</pre>
<pre>quickSort([PIVOT B],RESULT) :- partition(PIVOT,B,LEFT,RIGHT), printDivide(PIVOT,LEFT,RIGHT), quickSort(LEFT,TMPL), quickSort(RIGHT,TMPR), printMerge(PIVOT,TMPL,TMPR), append(TMPL,[PIVOT TMPR],RESULT).</pre>	<pre>list의 제일 앞 숫자를 PIVOT으로 받아 partition함 수를 호출해 정렬한 후 재귀함수를 이용해 LEFT와 RIGHT를 각각 또 정렬한다. 마지막으로 prolog의 기존 함수인 append()를 이용해 정렬된 LEFT와 RIGHT의 가장 앞 부분에 pivot을 넣어 RESULT에 저장하고 리턴한다.</pre>

전체 코드
<pre>partition(PIVOT,[],[],[]) :- !. partition(PIVOT,[A B],[A LEFT],RIGHT) :- A<PIVOT -> partition(PIVOT,B,LEFT,RIGHT). partition(PIVOT,[A B],LEFT,[A RIGHT]) :- A>PIVOT -> partition(PIVOT,B,LEFT,RIGHT).</pre>

```

printDivide(PIVOT,[],[]) :- !.
printDivide(PIVOT,LEFT,RIGHT) :-
    write("divide="), write(PIVOT),
    write(' | '), write(LEFT),
    write(RIGHT), nl.

printMerge(PIVOT,[],[]) :-
    write("merge:"),
    write([PIVOT]), nl.
printMerge(PIVOT,LEFT,RIGHT) :-
    write("merge:"), write(LEFT),
    write([PIVOT]), write(RIGHT), nl.

quickSort([],[]) :- !.
quickSort([PIVOT|B]) :-
    quickSort([PIVOT|B],RESULT),
    write(RESULT), nl.
quickSort([PIVOT|B],RESULT) :-
    partition(PIVOT,B,LEFT,RIGHT),
    printDivide(PIVOT,LEFT,RIGHT),
    quickSort(LEFT,TMPL),
    quickSort(RIGHT,TMPR),
    printMerge(PIVOT,TMPL,TMPR),
    append(TMPL,[PIVOT|TMPR],RESULT).

```

예시 결과

```

?- quickSort([7,3,1,2,9,5,4,8]).
divide=7| [3,1,2,5,4] [9,8]
divide=3| [1,2] [5,4]
divide=1| [] [2]
merge: [2]
merge: [] [1] [2]
divide=5| [4] []
merge: [4]
merge: [4] [5] []
merge: [1,2] [3] [4,5]
divide=9| [8] []
merge: [8]
merge: [8] [9] []
merge: [1,2,3,4,5] [7] [8,9]
[1,2,3,4,5,7,8,9]
true .

```

trace - quickSort([3,1,2])

```

[trace] ?- quickSort([3,1,2]).
Call: (7) quickSort([3, 1, 2]) ? creep
Call: (8) quickSort([3, 1, 2], _G1248) ? creep
Call: (9) partition(3, [1, 2], _G1249, _G1250) ?
creep

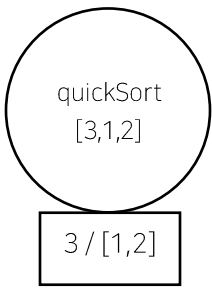
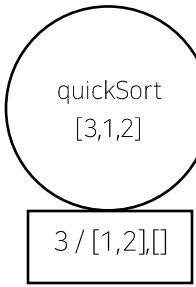
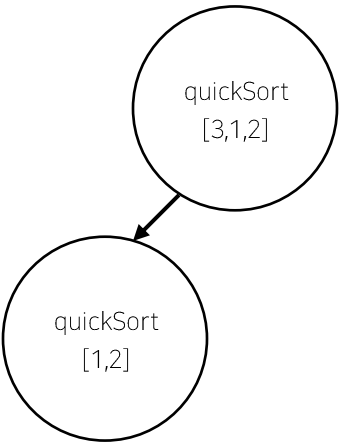
```

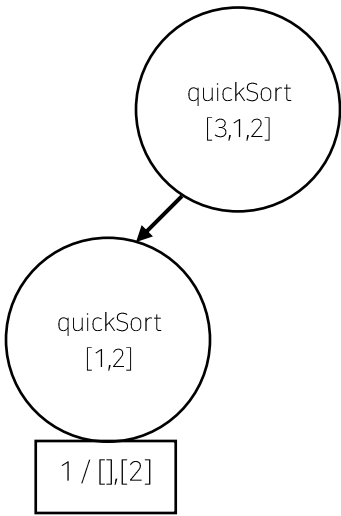
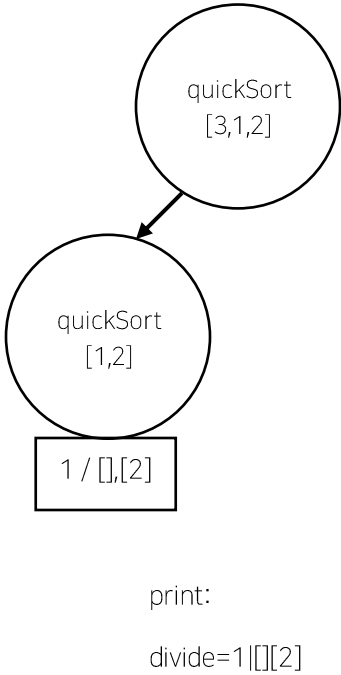
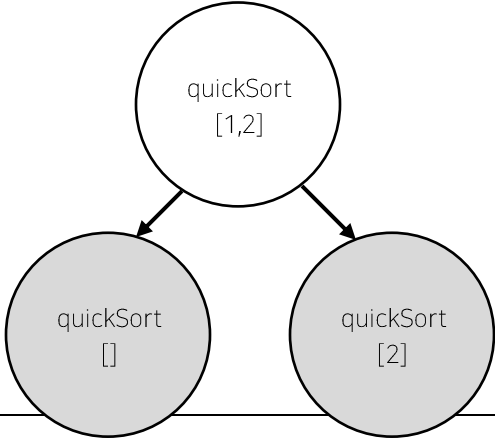
quickSort
[3,1,2]

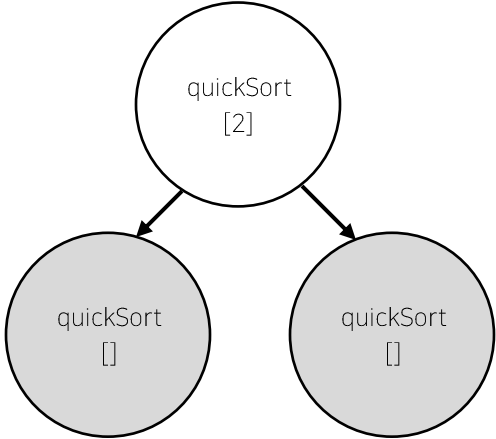
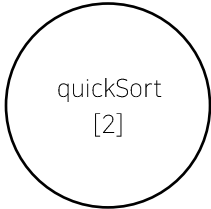
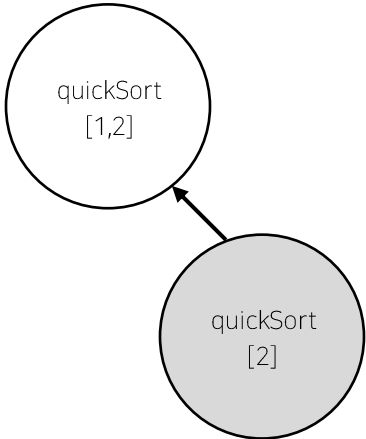
```

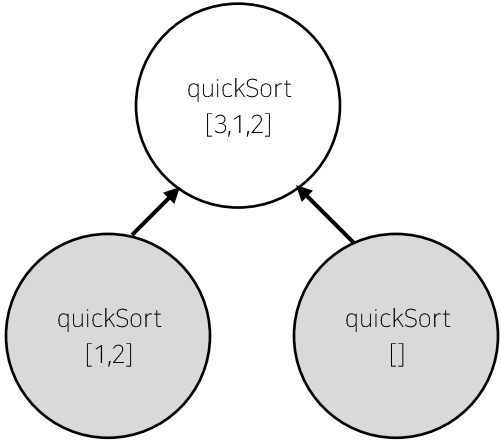
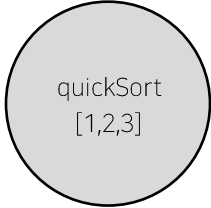
Call: (10) 1<3 ? creep
Exit: (10) 1<3 ? creep
Call: (10) partition(3, [2], _G1241, _G1253) ?

```

<p>creep Call: (11) 2<3 ? creep Exit: (11) 2<3 ? creep Call: (11) partition(3, [], _G1244, _G1256) ? creep Exit: (11) partition(3, [], [], []) ? creep Exit: (10) partition(3, [2], [2], []) ? creep Exit: (9) partition(3, [1, 2], [1, 2], []) ? creep</p>	
<p>Call: (9) printDivide(3, [1, 2], []) ? creep Call: (10) write("divide=") ? creep divide= Exit: (10) write("divide=") ? creep Call: (10) write(3) ? creep 3 Exit: (10) write(3) ? creep Call: (10) write(' ') ? creep Exit: (10) write(' ') ? creep Call: (10) write([1, 2]) ? creep [1,2] Exit: (10) write([1, 2]) ? creep Call: (10) write([]) ? creep [] Exit: (10) write([]) ? creep Call: (10) nl ? creep Exit: (10) nl ? creep Exit: (9) printDivide(3, [1, 2], []) ? creep</p>	 <p>print: divide=3 [1,2][]</p>
<p>Call: (9) quickSort([1, 2], _G1258) ? creep</p>	
<p>Call: (10) partition(1, [2], _G1259, _G1260) ? creep Call: (11) 2<1 ? creep Fail: (11) 2<1 ? creep Redo: (10) partition(1, [2], _G1259, _G1260) ? creep</p>	

<p>Call: (11) 2>1 ? creep Exit: (11) 2>1 ? creep Call: (11) partition(1, [], _G1262, _G1251) ? creep Exit: (11) partition(1, [], [], []) ? creep Exit: (10) partition(1, [2], [], [2]) ? creep</p>	 <pre> graph TD A((quickSort [3,1,2])) --> B((quickSort [1,2])) B --- C[1 / [], [2]] </pre>
<p>Call: (10) printDivide(1, [], [2]) ? creep Call: (11) write("divide=") ? creep divide= Exit: (11) write("divide=") ? creep Call: (11) write(1) ? creep 1 Exit: (11) write(1) ? creep Call: (11) write(' ') ? creep Exit: (11) write(' ') ? creep Call: (11) write([]) ? creep [] Exit: (11) write([]) ? creep Call: (11) write([2]) ? creep [2] Exit: (11) write([2]) ? creep Call: (11) nl ? creep Exit: (11) nl ? creep Exit: (10) printDivide(1, [], [2]) ? creep</p>	 <pre> graph TD A((quickSort [3,1,2])) --> B((quickSort [1,2])) B --- C[1 / [], [2]] C --- D[print: divide=1 [] [2]] </pre>
<p>Call: (10) quickSort([], _G1265) ? creep Exit: (10) quickSort([], []) ? creep Call: (10) quickSort([2], _G1265) ? creep Call: (11) partition(2, [], _G1266, _G1267) ? creep Exit: (11) partition(2, [], [], []) ? creep Call: (11) printDivide(2, [], []) ? creep Exit: (11) printDivide(2, [], []) ? creep</p>	 <pre> graph TD A((quickSort [1,2])) --> B((quickSort [])) A --> C((quickSort [2])) style B fill:#ccc style C fill:#ccc </pre>

<p>Call: (11) quickSort([], _G1265) ? creep Exit: (11) quickSort([], []) ? creep Call: (11) quickSort([], _G1265) ? creep Exit: (11) quickSort([], []) ? creep</p>	 <pre> graph TD A((quickSort [2])) --> B((quickSort [])) A --> C((quickSort [])) </pre>
<p>Call: (11) printMerge(2, [], []) ? creep Call: (12) write("merge:") ? creep merge: Exit: (12) write("merge:") ? creep Call: (12) write([2]) ? creep [2] Exit: (12) write([2]) ? creep Call: (12) nl ? creep Exit: (12) nl ? creep Exit: (11) printMerge(2, [], []) ? creep Call: (11) lists:append([], [2], _G1275) ? creep Exit: (11) lists:append([], [2], [2]) ? creep</p>	 <pre> graph TD A((quickSort [2])) </pre> <p>print: merge:[2]</p>
<p>Exit: (10) quickSort([2], [2]) ? creep Call: (10) printMerge(1, [], [2]) ? creep Call: (11) write("merge:") ? creep merge: Exit: (11) write("merge:") ? creep Call: (11) write([]) ? creep [] Exit: (11) write([]) ? creep Call: (11) write([1]) ? creep [1] Exit: (11) write([1]) ? creep Call: (11) write([2]) ? creep [2] Exit: (11) write([2]) ? creep Call: (11) nl ? creep Exit: (11) nl ? creep Exit: (10) printMerge(1, [], [2]) ? creep Call: (10) lists:append([], [1, 2], _G1284) ? creep creep</p>	 <pre> graph TD A((quickSort [1,2])) --> B((quickSort [2])) </pre> <p>print: merge:[][1][2]</p>

<p>Exit: (10) lists:append([], [1, 2], [1, 2]) ? creep</p> <p>Exit: (9) quickSort([1, 2], [1, 2]) ? creep</p> <p>Call: (9) quickSort([], _G1283) ? creep</p> <p>Exit: (9) quickSort([], []) ? creep</p> <p>Call: (9) printMerge(3, [1, 2], []) ? creep</p> <p>Call: (10) write("merge:") ? creep</p> <p>merge:</p> <p>Exit: (10) write("merge:") ? creep</p> <p>Call: (10) write([1, 2]) ? creep</p> <p>[1,2]</p> <p>Exit: (10) write([1, 2]) ? creep</p> <p>Call: (10) write([3]) ? creep</p> <p>[3]</p> <p>Exit: (10) write([3]) ? creep</p> <p>Call: (10) write([]) ? creep</p> <p>[]</p> <p>Exit: (10) write([]) ? creep</p> <p>Call: (10) nl ? creep</p> <p>Exit: (10) nl ? creep</p> <p>Exit: (9) printMerge(3, [1, 2], []) ? creep</p> <p>Call: (9) lists:append([1, 2], [3], _G1293) ?</p> <p>creep</p> <p>Exit: (9) lists:append([1, 2], [3], [1, 2, 3]) ?</p> <p>creep</p>	 <p>print:</p> <p>merge:[1,2][3][]</p>
<p>Exit: (8) quickSort([3, 1, 2], [1, 2, 3]) ? creep</p> <p>Call: (8) write([1, 2, 3]) ? creep</p> <p>[1,2,3]</p> <p>Exit: (8) write([1, 2, 3]) ? creep</p> <p>Call: (8) nl ? creep</p> <p>Exit: (8) nl ? creep</p> <p>Exit: (7) quickSort([3, 1, 2]) ? creep</p> <p>true .</p>	 <p>print:</p> <p>[1,2,3]</p>

3. nQueen

알고리즘
<ol style="list-style-type: none"> 가장 윗줄의 첫번째 칸부터 퀸을 놓는다. 그 다음 줄의 첫번째 칸부터 퀸을 하나 더 놓는다. <ol style="list-style-type: none"> 윗줄과 퀸이 겹치면 한 칸 오른쪽으로 옮긴다. 모든 칸이 겹치면 바로 윗줄로 돌아가 퀸을 한 칸 오른쪽으로 옮긴다. 모든 경우의 수를 출력한다.

prolog 코드	
<pre>listing(N,N,[N]) :- !. listing(A,N,[A B]) :- A<N -> A1 is A+1, listing(A1,N,B).</pre>	<pre>listing(): A부터 N까지의 숫자 list를 만드는 함수. A에 1을 더해가며 계속해서 재귀호출을 하 다가 A가 N과 같아지면 반대로 list에 하나 씩 넣으면서 리턴한다.</pre>
<pre>promising(_,_,_) :- !. promising(Q,[Q1 REST],DISTANCE) :- Q=W=Q1 -> Tmp is abs(Q1-Q), Tmp =W= DISTANCE -> DISTANCE1 is DISTANCE+1, promising(Q,REST,DISTANCE1).</pre>	<pre>promising(): 현재 퀸 위치가 다른 윗줄의 퀸들과 겹치는 지 확인해주는 함수. Q는 현재 퀸의 위치, [Q1 REST]는 윗줄 퀸 의 위치, DISTANCE는 Q1퀸이 있는 줄과 현재 줄과의 거리이다. Q와 Q1이 겹치거나 Q와 Q1의 차이가 DISTANCE랑 같으면 불가능하므로 그대로 리턴한다. 둘 다 겹치지 않으면 그 다음 윗줄을 확인 하기 위해 DISTANCE에 +1을 하고 Q1을 뺀 REST를 파라미터로 넣어 재귀호출한다.</pre>
<pre>queen([],_) :- !. queen([Q REST],N) :- queen(REST,N), listing(1,N,LIST), member(Q,LIST), promising(Q,REST,1).</pre>	<pre>queen(): 재귀호출을 이용해 퀸이 위치할 수 있는 경 우의 수를 계산하는 함수. 바로 윗줄의 퀸 위치까지 구한 뒤 listing함 수를 이용해 1부터 N까지 list를 만들고 1 부터 차례대로 퀸을 위치한다고 가정한 뒤 promising함수를 호출해 겹치는 지 확인한 다. 겹치지 않는 위치 Q를 REST앞에 넣어 리 턴한다.</pre>
<pre>nQueen(N) :- write("n="), write(N), nl, nQueen(N,RESULT), length(RESULT,NUM), write("# of answer="), write(NUM), nl, write(RESULT), nl. nQueen(N,RESULT) :- length(SOLUTION,N), findall(SOLUTION,queen(SOLUTION,N),RESULT).</pre>	<pre>nQueen(): findall함수를 이용해 queen을 만족하는 모 든 list들을 최종 결과인 RESULT에 저장한 다음 출력형태와 맞게 출력한다.</pre>

전체 코드	
<pre> nQueen(N) :- write("n="), write(N), nl, nQueen(N,RESULT), length(RESULT,NUM), write("# of answer="), write(NUM), nl, write(RESULT), nl. nQueen(N,RESULT) :- length(SOLUTION,N), findall(SOLUTION,queen(SOLUTION,N),RESULT). listing(N,N,[N]) :- !. listing(A,N,[A B]) :- A<N -> A1 is A+1, listing(A1,N,B). queen([],_) :- !. queen([Q REST],N) :- queen(REST,N), listing(1,N,LIST), member(Q,LIST), promising(Q,REST,1). promising(_,[],_) :- !. promising(Q,[Q1 REST],DISTANCE) :- Q=\=Q1 -> Tmp is abs(Q1-Q), Tmp =\= DISTANCE -> DISTANCE1 is DISTANCE+1, promising(Q,REST,DISTANCE1). </pre>	

예시 결과	
<pre> ?- nQueen(1) . n=1 # of answer=1 [[1]] true. ?- nQueen(4) . n=4 # of answer=2 [[3,1,4,2],[2,4,1,3]] true. </pre>	

trace - nQueen(4)	
<pre> [trace] ?- nQueen(4). Call: (7) nQueen(4) ? creep Call: (8) write("n=") ? creep n= Exit: (8) write("n=") ? creep </pre>	<p>"n=4"를 출력하고 queen(SOLUTION,4)를 호출한다.</p>

<p>Call: (8) write(4) ? creep</p> <p>4</p> <p>Exit: (8) write(4) ? creep</p> <p>Call: (8) nl ? creep</p> <p>Exit: (8) nl ? creep</p> <p>Call: (8) nQueen(4, _G1230) ? creep</p> <p>Call: (9) length(_G1229, 4) ? creep</p> <p>Exit: (9) length([_G1222, _G1225, _G1228, _G1231], 4) ? creep</p> <p>Call: (9) findall([_G1222, _G1225, _G1228, _G1231], queen([_G1222, _G1225, _G1228, _G1231], 4), _G1246) ? creep</p>	
<p>Call: (14) queen([_G1222, _G1225, _G1228, _G1231], 4) ? creep</p> <p>Call: (15) queen([_G1225, _G1228, _G1231], 4) ? creep</p> <p>Call: (16) queen([_G1228, _G1231], 4) ? creep</p> <p>Call: (17) queen([_G1231], 4) ? creep</p> <p>Call: (18) queen([], 4) ? creep</p> <p>Exit: (18) queen([], 4) ? creep</p> <p>Call: (18) listing(1, 4, _G1263) ? creep</p> <p>Call: (19) 1<4 ? creep</p> <p>Exit: (19) 1<4 ? creep</p> <p>Call: (19) _G1267 is 1+1 ? creep</p> <p>Exit: (19) 2 is 1+1 ? creep</p> <p>Call: (19) listing(2, 4, _G1255) ? creep</p> <p>Call: (20) 2<4 ? creep</p> <p>Exit: (20) 2<4 ? creep</p> <p>Call: (20) _G1273 is 2+1 ? creep</p> <p>Exit: (20) 3 is 2+1 ? creep</p> <p>Call: (20) listing(3, 4, _G1261) ? creep</p> <p>Call: (21) 3<4 ? creep</p> <p>Exit: (21) 3<4 ? creep</p> <p>Call: (21) _G1279 is 3+1 ? creep</p> <p>Exit: (21) 4 is 3+1 ? creep</p> <p>Call: (21) listing(4, 4, _G1267) ? creep</p> <p>Exit: (21) listing(4, 4, [4]) ? creep</p> <p>Exit: (20) listing(3, 4, [3, 4]) ? creep</p> <p>Exit: (19) listing(2, 4, [2, 3, 4]) ? creep</p> <p>Exit: (18) listing(1, 4, [1, 2, 3, 4]) ? creep</p> <p>Call: (18) lists:member(_G1231, [1, 2, 3, 4]) ? creep</p> <p>Exit: (18) lists:member(1, [1, 2, 3, 4]) ? creep</p> <p>Call: (18) promising(1, [], 1) ? creep</p>	<p>SOLUTION의 가장 윗줄까지 채귀로 들어간 뒤 [1,2,3,4]list를 만들어 1부터 차례로 promising한 지 확인한다.</p>
<p>Exit: (18) promising(1, [], 1) ? creep</p> <p>Exit: (17) queen([1], 4) ? creep</p>	<p>가장 윗줄이기 때문에 확인할 게 없으므로 바로 리턴 후 두번째 줄 퀸 위치를 결정하기 위해 다시</p>

<p>Call: (17) listing(1, 4, _G1284) ? creep Call: (18) 1<4 ? creep Exit: (18) 1<4 ? creep Call: (18) _G1288 is 1+1 ? creep Exit: (18) 2 is 1+1 ? creep Call: (18) listing(2, 4, _G1276) ? creep Call: (19) 2<4 ? creep Exit: (19) 2<4 ? creep Call: (19) _G1294 is 2+1 ? creep Exit: (19) 3 is 2+1 ? creep Call: (19) listing(3, 4, _G1282) ? creep Call: (20) 3<4 ? creep Exit: (20) 3<4 ? creep Call: (20) _G1300 is 3+1 ? creep Exit: (20) 4 is 3+1 ? creep Call: (20) listing(4, 4, _G1288) ? creep Exit: (20) listing(4, 4, [4]) ? creep Exit: (19) listing(3, 4, [3, 4]) ? creep Exit: (18) listing(2, 4, [2, 3, 4]) ? creep Exit: (17) listing(1, 4, [1, 2, 3, 4]) ? creep Call: (17) lists:member(_G1228, [1, 2, 3, 4]) ? creep Exit: (17) lists:member(1, [1, 2, 3, 4]) ? creep Call: (17) promising(1, [1], 1) ? creep</p>	<p>[1,2,3,4]list를 만들어 1부터 promising한 지 확인한다.</p>
<p>Call: (18) 1=W=1 ? creep Fail: (18) 1=W=1 ? creep Fail: (17) promising(1, [1], 1) ? creep Redo: (17) lists:member(_G1228, [1, 2, 3, 4]) ? creep Exit: (17) lists:member(2, [1, 2, 3, 4]) ? creep Call: (17) promising(2, [1], 1) ? creep</p>	<p>1번 위치는 바로 윗줄과 겹치므로 다시 돌아가 list의 두번째 원소인 2를 넣어 확인한다.</p>
<p>// 종락 //</p> <p>Exit: (9) findall([_G1222, _G1225, _G1228, _G1231], user:queen([_G1222, _G1225, _G1228, _G1231], 4, [[3, 1, 4, 2], [2, 4, 1, 3]]) ? creep Exit: (8) nQueen(4, [[3, 1, 4, 2], [2, 4, 1, 3]]) ? creep Call: (8) length([[3, 1, 4, 2], [2, 4, 1, 3]], _G1292) ? creep Exit: (8) length([[3, 1, 4, 2], [2, 4, 1, 3]], 2) ? creep Call: (8) write("# of answer=") ? creep # of answer= Exit: (8) write("# of answer=") ? creep Call: (8) write(2) ? creep 2 Exit: (8) write(2) ? creep</p>	<p>같은 방식으로 4개의 위치를 모두 찾은 뒤에 RESULT에 저장된 모든 list들을 출력한다.</p>

<p>Call: (8) nl ? creep</p> <p>Exit: (8) nl ? creep</p> <p>Call: (8) write([[3, 1, 4, 2], [2, 4, 1, 3]]) ? creep</p> <p>[[3,1,4,2],[2,4,1,3]]</p> <p>Exit: (8) write([[3, 1, 4, 2], [2, 4, 1, 3]]) ? creep</p> <p>Call: (8) nl ? creep</p> <p>Exit: (8) nl ? creep</p> <p>Exit: (7) nQueen(4) ? creep</p> <p>true.</p>	
--	--

4. Shortest Path

알고리즘
1. Backtracking 1-1). 출발지부터 시작해서 연결된 노드를 방문한다. 1-2). 도착지에 도착하면 다시 출발지로 돌아가면서 경로와 총 길이를 저장한다. 2. 모든 경우를 다 찾은 뒤에 가장 짧은 경로를 출력한다.

prolog 코드	
near(1,2,6). near(1,3,3). near(2,1,6). near(2,3,2). near(2,4,5). near(3,1,3). near(3,2,2). near(3,4,3). near(3,5,4). near(4,2,5). near(4,3,3). near(4,5,2). near(4,6,3). near(5,3,4). near(5,4,2). near(5,6,5). near(6,4,3). near(6,5,5).	near(A,B,C): A와 B가 연결되어 있으며 weight가 C라는 정보를 저장해둔다.
sp(START,ARRIVE) :- findall(X,sp(START,ARRIVE,PATH,X,[]),[A L]), findall(X,sp(START,ARRIVE,X,LENGTH,[]),P), minLength([A L],A,MINRES), minPath([A L],P,MINRES,RPATH,RLENGTH), write(RPATH), nl, write(RLENGTH), nl.	출발지부터 도착지까지 나올 수 있는 모든 경로와 그 경로의 총 길이를 list에 저장하고 그 중 가장 짧은 길이를 구한 뒤 그 길이에 대응되는 경로를 구해서 출력한다.
sp(START,START,[START],0,VISIT) :- !. sp(START,ARRIVE,[START PATH],LENGTH,VISIT) :- W+member(START,VISIT) -> near(START,START1,WEIGHT), sp(START1,ARRIVE,PATH,LENGTH1,[START VISIT]), LENGTH is LENGTH1+WEIGHT.	sp(): VISIT에는 이미 방문한 노드를 저장하는 list로 처음에는 빈 list로 시작한다. START 노드를 방문하지 않았으면 그 노드로 옮겨 가서 미리 저장해둔 near() 중 한 곳으로 경로를 설정한다. 만약 START 노드를 이미 방문했다면 아무것도 하지 않고 돌아간다. 만약 도착지에 도착해서 START와 ARRIVE 변수가 같아지면 경로에 도착지를 추가하고 총 길이를 0으로 시작해 재귀를 풀어가면서 경로를 추가하고 총 길이도 더해나간다.
minLength([],MIN,MIN) :- !. minLength([A1 L],MIN,RESULT) :- A1<MIN -> MIN1 is A1, minLength(L,MIN1,RESULT); minLength(L,MIN,RESULT).	minLength(): 길이의 최소값을 구하는 함수. 모든 list를 탐색해 가장 작은 값을 RESULT 변수에 저장해 리턴한다.
minPath([MINRES L],[A2 P],MINRES,A2,MINRES) :- !. minPath([A1 L],[A2 P],MINRES,RPATH,RLENGTH) :- minPath(L,P,MINRES,RPATH,RLENGTH).	minPath(): 가장 짧은 경로를 구하는 함수. list를 탐색하다가 최소 길이와 같은 경로가 나오면 최종 경로와 길이를 각각 RPATH와 RLENGTH에 저장해 리턴한다.

전체 코드	
<pre> near(1,2,6). near(1,3,3). near(2,1,6). near(2,3,2). near(2,4,5). near(3,1,3). near(3,2,2). near(3,4,3). near(3,5,4). near(4,2,5). near(4,3,3). near(4,5,2). near(4,6,3). near(5,3,4). near(5,4,2). near(5,6,5). near(6,4,3). near(6,5,5). sp(START,ARRIVE) :- findall(X, sp(START,ARRIVE,PATH,X,[]), [A L]), findall(X, sp(START,ARRIVE,X,LENGTH,[]),P), minLength([A L],A,MINRES), minPath([A L],P,MINRES,RPATH,RLENGTH), write(RPATH), nl, write(RLENGTH), nl. sp(START,START,[START],0,VISIT) :- !. sp(START,ARRIVE,[START PATH],LENGTH,VISIT) :- \+member(START,VISIT) -> near(START,START1,WEIGHT), sp(START1,ARRIVE,PATH,LENGTH1,[START VISIT]), LENGTH is LENGTH1+WEIGHT. minLength([],MIN,MIN) :- !. minLength([A1 L],MIN,RESULT) :- A1<MIN -> MIN1 is A1, minLength(L,MIN1,RESULT); minLength(L,MIN,RESULT). minPath([MINRES L],[A2 P],MINRES,A2,MINRES) :- !. minPath([A1 L],[A2 P],MINRES,RPATH,RLENGTH) :- minPath(L,P,MINRES,RPATH,RLENGTH). </pre>	

예시 결과	
	<pre> ?- sp(1,3). [1,3] 3 true. ?- sp(1,6). [1,3,4,6] 9 true. ?- sp(6,1). [6,4,3,1] 9 true. </pre>

trace - sp(1,3)	
<pre> [trace] ?- sp(1,3). Call: (7) sp(1,3) ? creep </pre>	모든 경로의 길이를 구하기 위해 findall()함수를 호출한다.

^ Call: (8) findall(_G1224, sp(1, 3, _G1223, _G1224, []), [_G1227 _G1228]) ? creep Call: (13) sp(1, 3, _G1223, _G1224, []) ? creep	
Call: (14) lists:member(1, []) ? creep Fail: (14) lists:member(1, []) ? creep Redo: (13) sp(1, 3, [1 _G1248], _G1224, []) ? creep Call: (14) near(1, _G1258, _G1259) ? creep Exit: (14) near(1, 2, 6) ? creep	VISIT에 1이 없기 때문에 1번 노드와 인접한 노드 중 하나인 2번 노드를 찾는다.
Call: (14) sp(2, 3, _G1248, _G1263, [1]) ? creep Call: (15) lists:member(2, [1]) ? creep Fail: (15) lists:member(2, [1]) ? creep Redo: (14) sp(2, 3, [2 _G1254], _G1266, [1]) ? creep Call: (15) near(2, _G1264, _G1265) ? creep Exit: (15) near(2, 1, 6) ? creep	2번 노드 역시 VISIT에 없기 때문에 2번 노드와 인접한 노드 중 하나인 1번 노드를 찾는다.
Call: (15) sp(1, 3, _G1254, _G1269, [2, 1]) ? creep Call: (16) lists:member(1, [2, 1]) ? creep Exit: (16) lists:member(1, [2, 1]) ? creep Fail: (15) sp(1, 3, _G1254, _G1269, [2, 1]) ? creep Redo: (15) near(2, _G1264, _G1265) ? creep Exit: (15) near(2, 3, 2) ? creep	1번 노드는 이미 VISIT에 있기 때문에 다시 돌아가 2번 노드와 다른 인접한 노드인 3번 노드를 찾는다.
Call: (15) sp(3, 3, _G1254, _G1269, [2, 1]) ? creep Exit: (15) sp(3, 3, [3], 0, [2, 1]) ? creep	3번 노드가 도착지이기 때문에 경로에 3번 노드를 넣고 길이를 0으로 초기화한다.
Call: (15) _G1272 is 0+2 ? creep Exit: (15) 2 is 0+2 ? creep Exit: (14) sp(2, 3, [2, 3], 2, [1]) ? creep	바로 전에 방문한 2번 노드로 돌아가 경로에 2번을 추가하고 LENGTH에 2번과 3번 사이의 길이인 2를 더한다.
Call: (14) _G1224 is 2+6 ? creep Exit: (14) 8 is 2+6 ? creep Exit: (13) sp(1, 3, [1, 2, 3], 8, []) ? creep	위 과정을 반복해 최종 경로와 길이를 리턴한다.
// 중략 // ^ Exit: (8) findall(_G1224, user:sp(1, 3, _G1223, _G1224, []), [8, 14, 17, 23, 3]) ? creep	마찬가지로 다른 경로들도 모두 찾은 뒤 구한 길이들을 list에 저장한다.
^ Call: (8) findall(_G1224, sp(1, 3, _G1224, _G1265, []), [_G1277]) ? creep // 중략 // ^ Exit: (8) findall(_G1224, user:sp(1, 3, _G1224, _G1265, []), [[1, 2, 3], [1, 2, 4, 3], [1, 2, 4, 5, 3], [1, 2, 4, 6 ...], [1, 3]]) ? creep	findall()함수를 다시 호출해 경로찾기를 한 번 더 반복한 뒤 이번에는 list에 경로를 저장한다.
Call: (8) minLength([8, 14, 17, 23, 3], 8, _G1372) ? creep Call: (9) 8<8 ? creep Fail: (9) 8<8 ? creep	길이의 최소값을 찾기 위해 minLength()함수를 호출했고 제일 앞에 있는 값인 8을 기준으로 그보다 더 작은 수를 찾아내려간다. 앞의 두 원소인 8, 14는 둘 다 8보다 작지 않으므로 최소값을 바꾸지

<p>Redo: (8) minLength([8, 14, 17, 23, 3], 8, _G1372) ? creep</p> <p>Call: (9) minLength([14, 17, 23, 3], 8, _G1372) ? creep</p> <p>Call: (10) 14<8 ? creep</p> <p>Fail: (10) 14<8 ? creep</p> <p>Redo: (9) minLength([14, 17, 23, 3], 8, _G1372) ? creep</p> <p>Call: (10) minLength([17, 23, 3], 8, _G1372) ? creep</p>	<p>않고 재귀만 호출한다.</p>
<p>Call: (11) 17<8 ? creep</p> <p>Fail: (11) 17<8 ? creep</p> <p>Redo: (10) minLength([17, 23, 3], 8, _G1372) ? creep</p> <p>Call: (11) minLength([23, 3], 8, _G1372) ? creep</p> <p>Call: (12) 23<8 ? creep</p> <p>Fail: (12) 23<8 ? creep</p> <p>Redo: (11) minLength([23, 3], 8, _G1372) ? creep</p> <p>Call: (12) minLength([3], 8, _G1372) ? creep</p> <p>Call: (13) 3<8 ? creep</p> <p>Exit: (13) 3<8 ? creep</p> <p>Call: (13) _G1370 is 3 ? creep</p> <p>Exit: (13) 3 is 3 ? creep</p>	<p>반복해서 list 끝까지 탐색하다가 8보다 작은 3을 찾고 리턴값을 3으로 바꾼다.</p>
<p>Call: (13) minLength([], 3, _G1372) ? creep</p> <p>Exit: (13) minLength([], 3, 3) ? creep</p> <p>Exit: (12) minLength([3], 8, 3) ? creep</p> <p>Exit: (11) minLength([23, 3], 8, 3) ? creep</p> <p>Exit: (10) minLength([17, 23, 3], 8, 3) ? creep</p> <p>Exit: (9) minLength([14, 17, 23, 3], 8, 3) ? creep</p> <p>Exit: (8) minLength([8, 14, 17, 23, 3], 8, 3) ? creep</p> <p>Call: (8) minPath([8, 14, 17, 23, 3], [[1, 2, 3], [1, 2, 4, 3], [1, 2, 4, 5, 3], [1, 2, 4, 6 ...], [1, 3]], 3, _G1376, _G1377) ? creep</p>	<p>위에서 찾은 최소값을 minPath()함수의 MIN 파라미터에 넣어 호출한다.</p>
<p>Call: (9) minPath([14, 17, 23, 3], [[1, 2, 4, 3], [1, 2, 4, 5, 3], [1, 2, 4, 6, 5 ...], [1, 3]], 3, _G1376, _G1377) ? creep</p> <p>Call: (10) minPath([17, 23, 3], [[1, 2, 4, 5, 3], [1, 2, 4, 6, 5, 3], [1, 3]], 3, _G1376, _G1377) ? creep</p> <p>Call: (11) minPath([23, 3], [[1, 2, 4, 6, 5, 3], [1, 3]], 3, _G1376, _G1377) ? creep</p> <p>Call: (12) minPath([3], [[1, 3]], 3, _G1376, _G1377) ? creep</p>	<p>list 원소를 차례대로 탐색하다 최소 길이와 같은 값을 찾으면 그때의 경로와 길이를 최종 리턴값에 저장한다.</p>

Exit: (12) minPath([3], [[1, 3]], 3, [1, 3], 3) ? creep	
Exit: (11) minPath([23, 3], [[1, 2, 4, 6, 5, 3], [1, 3]], 3, [1, 3], 3) ? creep Exit: (10) minPath([17, 23, 3], [[1, 2, 4, 5, 3], [1, 2, 4, 6, 5, 3], [1, 3]], 3, [1, 3], 3) ? creep Exit: (9) minPath([14, 17, 23, 3], [[1, 2, 4, 3], [1, 2, 4, 5, 3], [1, 2, 4, 6, 5 ...], [1, 3]], 3, [1, 3], 3) ? creep Exit: (8) minPath([8, 14, 17, 23, 3], [[1, 2, 3], [1, 2, 4, 3], [1, 2, 4, 5, 3], [1, 2, 4, 6 ...], [1, 3]], 3, [1, 3], 3) ? creep Call: (8) write([1, 3]) ? creep [1,3] Exit: (8) write([1, 3]) ? creep Call: (8) nl ? creep Exit: (8) nl ? creep Call: (8) write(3) ? creep 3 Exit: (8) write(3) ? creep Call: (8) nl ? creep Exit: (8) nl ? creep Exit: (7) sp(1, 3) ? creep true.	찾은 최종 경로와 최종 길이를 출력한다.