COS 397 week 5

BACKTRACKING AND SEARCHING

Whether a graph is an explicit or implicit structure in describing a problem, it is often the case that *searching* the graph structure may be necessary. Thus it is required to have methods which

- Can *mark* nodes in a graph which have already been *examined*.
- Determine which node should be examined next.
- Ensure that every node in the graph **can** (but not necessarily **will**) be visited.

These requirements must be realized subject to the constraint that the search process respects the structure of the graph.

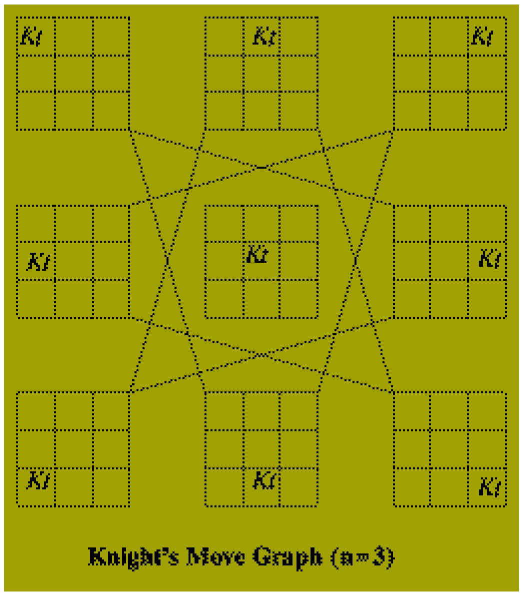
Any new node examined must be adjacent to some node that has previously been visited.
So, *search methods* implicitly describe an **ordering** of the nodes in a given graph.

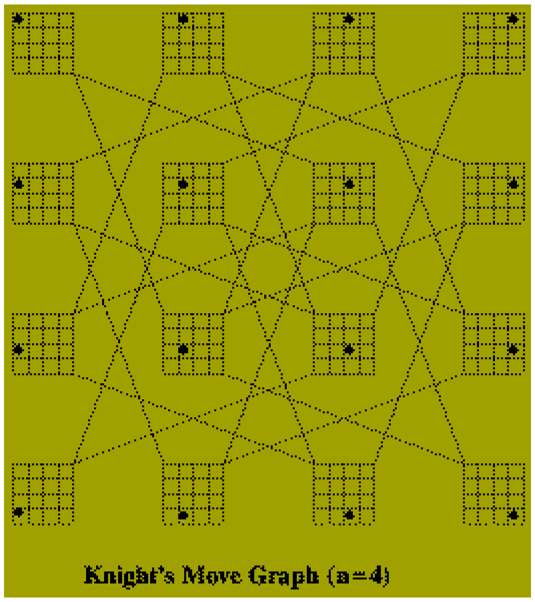
Suppose a problem may be expressed in terms of detecting a particular class of subgraph in a graph. Then the *backtracking* approach to solving such a problem would be:
Scan each node of the graph, *following a specific order*, until

- A subgraph constituting a solution has been found **or**
- It is discovered that the subgraph built so far cannot be extended to be a solution.

If (2) occurs then the search process is `*backed-up*' until a node is reached from which a solution might still be found.

Simple Example: Knight's Tour
Given a natural number, *n*, describe how a Knight should be moved on an *n times n* chessboard so that it visits every square exactly once and ends on its starting square. The *implicit graph* in this problem has n^2 nodes corresponding to each position the Knight must occupy. There is an edge between two of these nodes if the corresponding positions are *one move apart*. The sub-graph that defines a solution is a cycle which contains each node of the implicit graph.





Depth-First Search
The Knight's Tour algorithm organizes the search of the implicit graph using a **depth-first** approach. Depth-first search is one method of constructing a **search tree** for *explicit graphs*.
Let *G(V,E)* be a connected graph. A **search tree** of *G(V,E)* is a **spanning tree**, *T(V, F)* of *G(V,E)* in which the nodes of *T* are labelled with unique values *k* (*1 < = k < =* |*V*|) which satisfy:

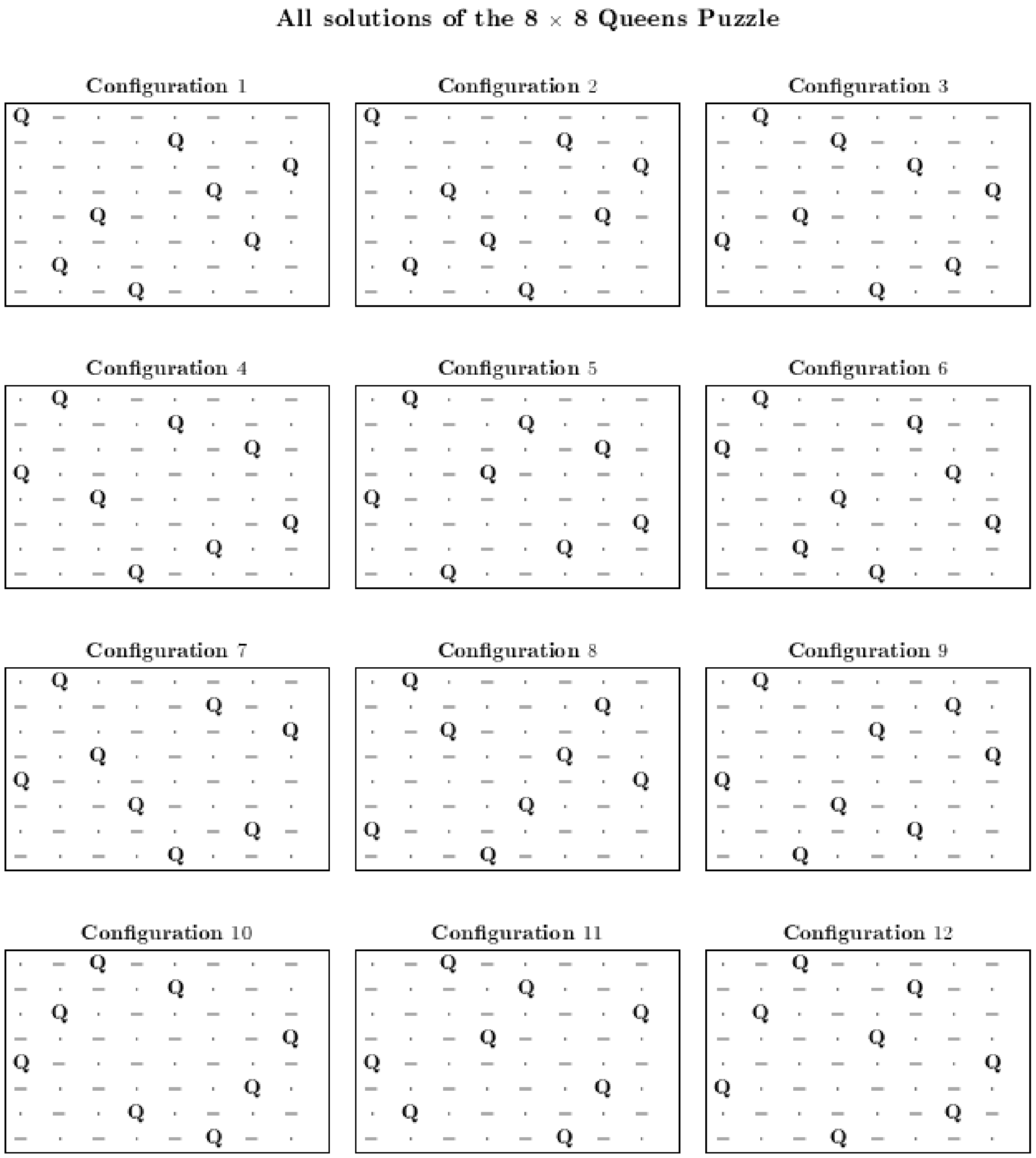
- A distinguished node called the **root** is labeled *1*.
- If *(p,q)* is an edge of *T* then the label assigned to *p* is less than the label assigned to *q*.

The labeling of a search tree prescribes the *order* in which the nodes of *G* are to be scanned.
Given an undirected graph *G(V,E)*, the depth-first search method constructs a search tree using the following recursive algorithm:

```
procedure depth_first_search (G(V,E) : graph; v : node;
                             lambda : integer; T : in outsearch_tree)
begin
   label(v) := lambda; lambda := lambda+1;
   for each w such that {v,w} in E do
     if label(w) = 0 then
       Add edge {v,w} to T;
       depth_first_search(G(V,E),w,lambda,T);
     end if;
   end loop;
end
begin
   for w in V do label(w) = 0;
   lambda := 1;
   depthfirstsearch ( G(V,E), v, lambda, T);
end;
```

The running time of the algorithm is *O*(|*E*|) since each edge of the graph is examined only once.

The *N* by *N* Queens Problem. In chess, a queen can move as far as she pleases, horizontally, vertically, or diagonally. A chess board has 8 rows and 8 columns. The standard 8 by 8 Queen's problem asks how to place 8 queens on an ordinary chess board so that none of them can hit any other in one move.
Two solutions are not essentially distinct if you can obtain one from another by rotating your chess board, or placing in in front of a mirror, or combining these two operations.
An obvious modification of the 8 by 8 problem is to consider an *N* by *N* "chess board" and ask if one can place *N* queens on such a board. This is impossible if *N* is 2 or 3, and it is reasonably straightforward to find solutions when *N* is 4, 5, 6, or 7. The problem begins to become difficult for manual solution precisely when *N* is 8. Probably the fact that this number coincidentally equals the dimensions of an ordinary chess board has contributed to the popularity of the problem.



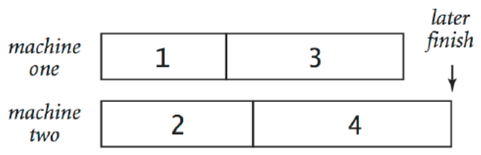
Searching Algorithms Overview

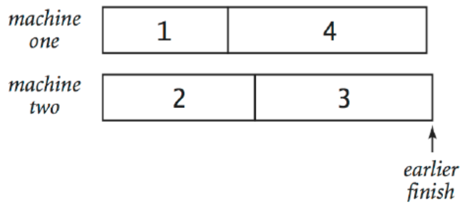
- **Exhaustive search**. Iterate through all elements of a search space.
- **Backtracking.** Systematic method for generating all solutions to a problem, by successively augmenting partial solutions.
- **Applicability.** Huge range of problems (include NP-hard ones).
- **Caveat.** Search space is typically exponential in size! Effectiveness is limited to relatively small instances.

Enumerating Subsets

Scheduling (set partitioning). Given *n* jobs of varying length, divide among two machines to minimize the time (or, equivalently, difference between finish times) the last job finishes.

job	length
1	1.41
2	1.73
3	2.00
4	2.23





Enumerating subsets. Given *n* items, enumerate all 2^n subsets.

- Count in binary from 0 to $2^n - 1$.
- Look at binary representation.

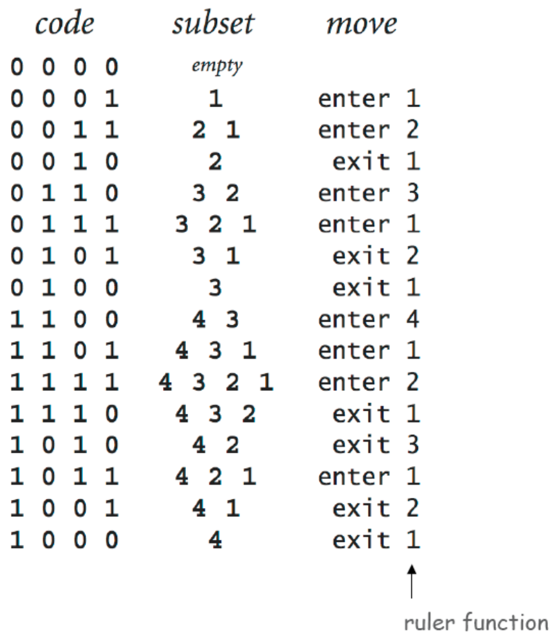
integer	*binary code*	*machine one*	*machine two*
0	0 0 0 0	empty	4 3 2 1
1	0 0 0 1	1	4 3 2
2	0 0 1 0	2	4 3 1
3	0 0 1 1	2 1	4 3
4	0 1 0 0	3	4 2 1
5	0 1 0 1	3 1	4 2
6	0 1 1 0	3 2	4 1
7	0 1 1 1	3 2 1	4
8	1 0 0 0	4	3 2 1
9	1 0 0 1	4 1	3 2
10	1 0 1 0	4 2	3 1
11	1 0 1 1	4 2 1	3
12	1 1 0 0	4 3	2 1
13	1 1 0 1	4 3 1	2
14	1 1 1 0	4 3 2	1
15	1 1 1 1	4 3 2 1	empty

```
int N = 1 << n;
for (int i = 0; i < N; i++)
 {
  int b = i;
  for(int j=0; j<n; j++)
   {cout << b%2; b /=2;}
  cout << endl;
 }
```

Output for *n* = 4

```
0000
1000
0100
1100
0010
1010
0110
1110
0001
1001
0101
1101
0011
1011
0111
1111
```

"Samuel Beckett"
Quad. Starting with empty stage, 4 characters enter and exit one at a time, such that each subset of actors appears exactly once.



code	subset	move
0 0 0 0	empty	
0 0 0 1	1	enter 1
0 0 1 1	2 1	enter 2
0 0 1 0	2	exit 1
0 1 1 0	3 2	enter 3
0 1 1 1	3 2 1	enter 1
0 1 0 1	3 1	exit 2
0 1 0 0	3	exit 1
1 1 0 0	4 3	enter 4
1 1 0 1	4 3 1	enter 1
1 1 1 1	4 3 2 1	enter 2
1 1 1 0	4 3 2	exit 1
1 0 1 0	4 2	exit 3
1 0 1 1	4 2 1	enter 1
1 0 0 1	4 1	exit 2
1 0 0 0	4	exit 1

ruler function

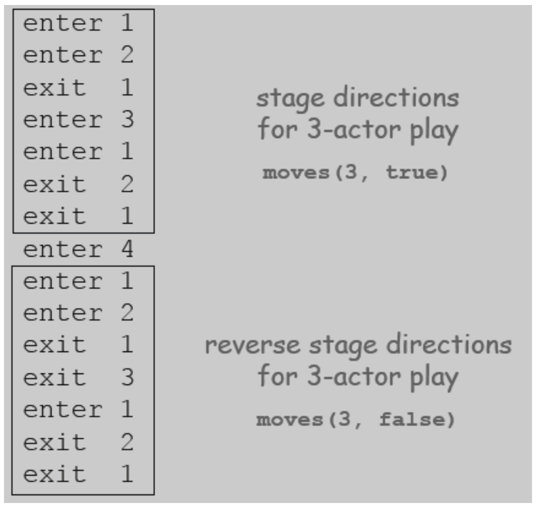
```
void rule(int L, int R, int h)
{ int m=(L+R)/2;
  if(h>0)
   {p[m]=h;
    rule(L,m,h-1);
    rule(m,R,h-1);
   }
 }
```

For example, the call `rule(0,16,4)` yields the following contents of `p[]`:
`0121312141213121`

```
void moves(int n, bool enter)
{
 if (n == 0) return;
 moves(n-1, true);
 if (enter) cout << "enter " << n << endl;
 else cout << "exit " << n << endl;
 moves(n-1, false);
}
```

For example, the call `moves(4, true)` yields:

```
enter 1
enter 2
exit 1
enter 3
enter 1
exit 2
exit 1
enter 4
enter 1
enter 2
exit 1
exit 3
enter 1
exit 2
exit 1
```



```
enter 1
enter 2
exit  1
enter 3
enter 1
exit  2
exit  1
enter 4
enter 1
enter 2
exit  1
exit  3
enter 1
exit  2
exit  1
```

stage directions for 3-actor play

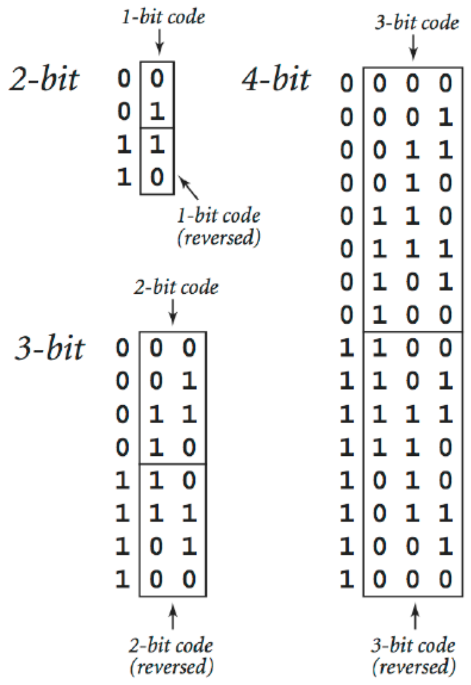
`moves(3, true)`

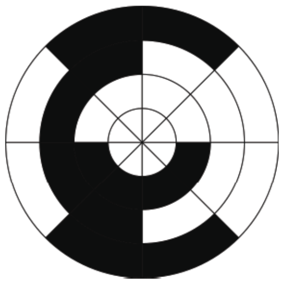
reverse stage directions for 3-actor play

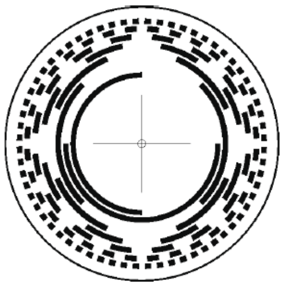
`moves(3, false)`

Enumerating Subsets: Binary Reflected Gray Code
The *n*-bit code is:

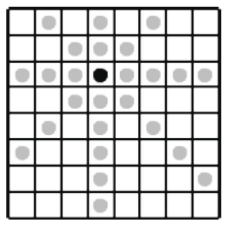
- the (n – 1) bit code with a 0 prepended to each word, followed by
- the (n – 1) bit code in reverse order, with a 1 prepended to each word.

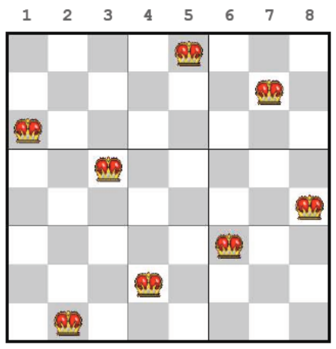



3-bit rotary encoder


8-bit rotary encoder

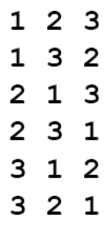
Enumerating Permutations.
8-queens problem. Place 8 queens on a chessboard so that no queen can attack any other queen.





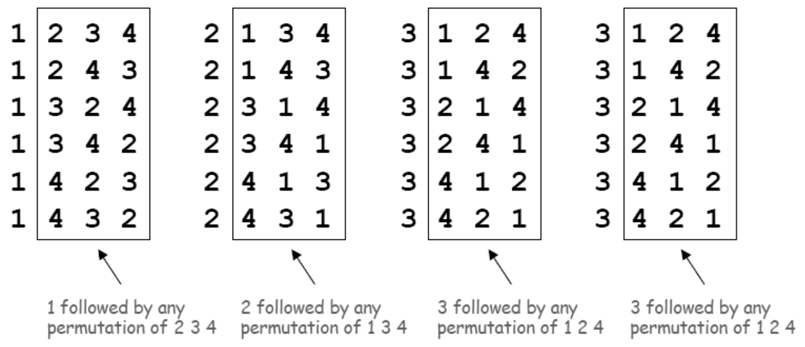
Represent the solution as a permutation: q[i] = column of a queen in row i.
Queens **i** and **j** can attack each other if |**q[i] + i**| = |**q[j] + j**|
Enumerating Permutations. Given *n* items, enumerate all *n*! permutations (in each presentation, order matters)

3-element permutations



1	2	3
1	3	2
2	1	3
2	3	1
3	1	2
3	2	1

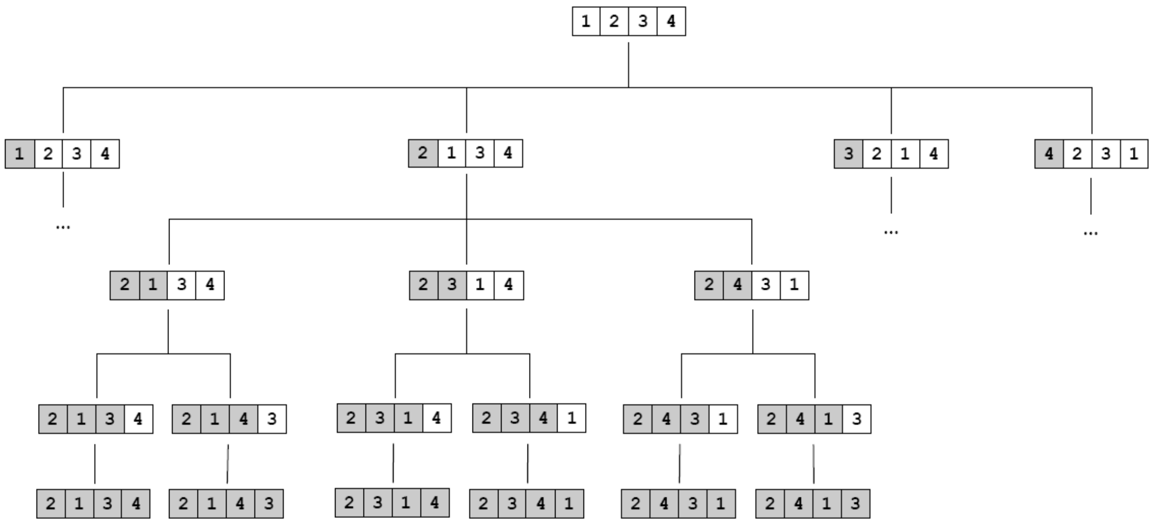
4-element permutations



1	2 3 4	2	1 3 4	3	1 2 4	3	1 2 4
1	2 4 3	2	1 4 3	3	1 4 2	3	1 4 2
1	3 2 4	2	3 1 4	3	2 1 4	3	2 1 4
1	3 4 2	2	3 4 1	3	2 4 1	3	2 4 1
1	4 2 3	2	4 1 3	3	4 1 2	3	4 1 2
1	4 3 2	2	4 3 1	3	4 2 1	3	4 2 1

1 followed by any permutation of 2 3 4
2 followed by any permutation of 1 3 4
3 followed by any permutation of 1 2 4
3 followed by any permutation of 1 2 4

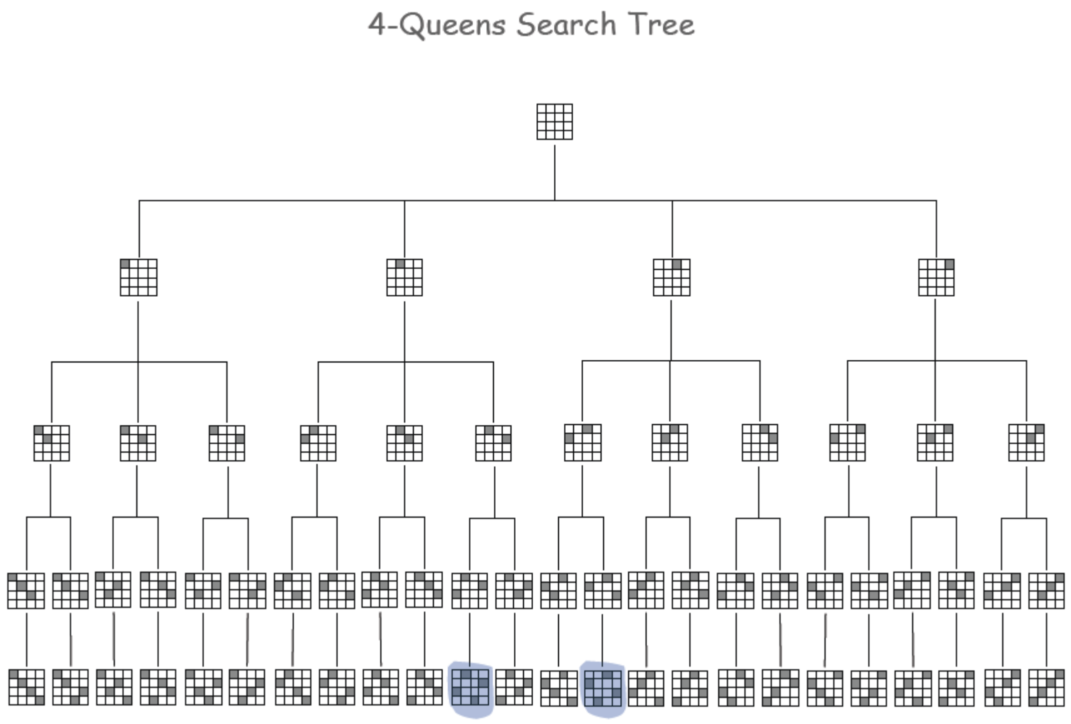
To enumerate all permutations of a set of *n* elements:
For each element a_i
– put a_i first, then append
– a permutation of the remaining elements ($a_0, \ldots, a_{i-1}, a_{i+1}, \ldots, a_{n-1}$)

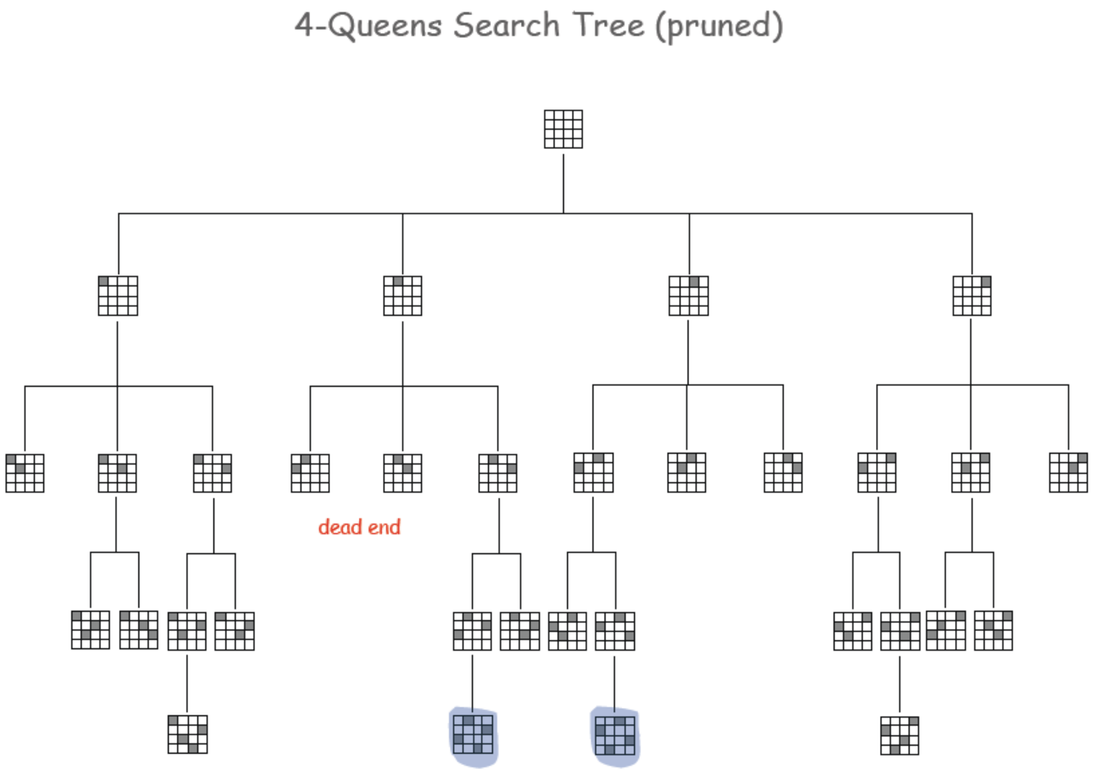


```
#include<iostream>
using namespace std;

int N = 4;
int a[] = {0, 1, 2, 3, 4};

void printPermutations(int a[])
{
 for(int i=1; i<=N; i++)
  cout << a[i];
 cout << endl;
}

void enumerate(int a[], int n)
{
 if (n == N) printPermutations(a);
 for (int i = n; i <= N; i++)
 {
  swap(a[i], a[n]);
  enumerate(a, n+1);
  swap(a[n], a[i]);
 }
}

int main()
{
 enumerate(a, 1);
}
```

```
1234
1243
1324
1342
1432
1423
2134
2143
2314
2341
2431
2413
3214
3241
3124
3142
3412
3421
4231
4213
4321
4312
4132
4123
```

Pruning

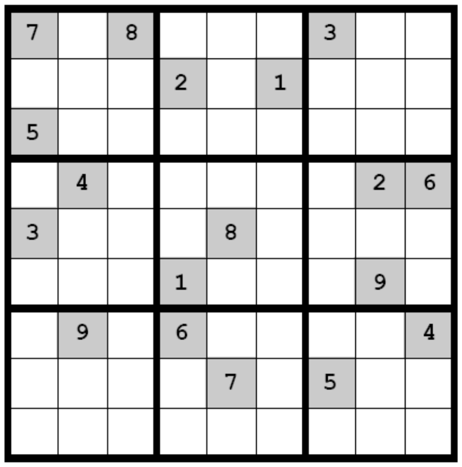


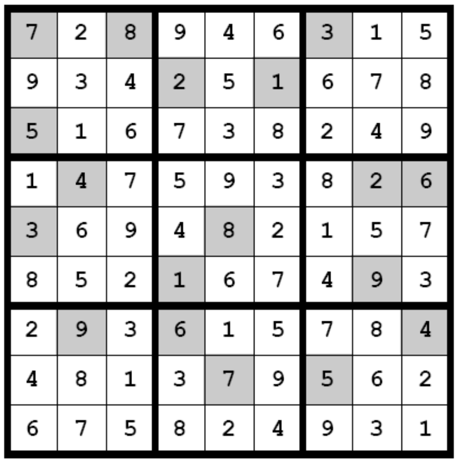


N-Queens: Backtracking Solution
Stop enumerating if adding the *n*-th queen leads to a violation.

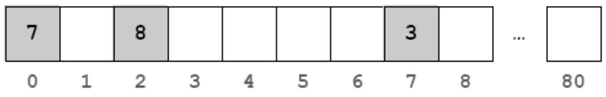
```
void enumerate(int q[], int n)
{
  if (n == N) printQueens(q);
  for (int i = n; i <= N; i++)
   {
    swap(q[i], q[n]);
    if (isConsistent(q, n)) enumerate(q, n+1);
    swap(q[n], q[i]);
   }
}
```

Sudoku: Fill 9-by-9 grid so that every row, column, and box contains the digits 1 through 9.





Solution: Linearize. Treat 9-by-9 array as an array of length 81.



Enumerate all assignments. Count from 0 to $9^{81} - 1$ in base 9 (using digits 1 to 9).

Sudoku: Backtracking Solution
Iterate through elements of search space.

- For each empty cell, there are 9 possible choices.
- Make one choice and recur.
- If you reach a contradiction, go back to previous choice, and make next available choice.

Pruning. Stop as soon as you reach a contradiction.
Improvements: Choose most constrained cell to examine next.