# Combinatorial Search

- permutations
- backtracking
- counting
- subsets
- paths in a graph

# Overview

Exhaustive search. Iterate through all elements of a search space.

Applicability. Huge range of problems (include intractable ones).


Caveat. Search space is typically exponential in size $\Rightarrow$ effectiveness may be limited to relatively small instances.

Backtracking. Systematic method for examining feasible solutions to a problem, by systematically pruning infeasible solutions.

## Warmup: enumerate N-bit strings

**Goal.** Process all $2^N$ bit strings of length N.

- Maintain `a[i]` where `a[i]` represents bit `i`.
- Simple recursive method does the job.

```
// enumerate bits in a[k] to a[N-1]
private void enumerate(int k)
{
   if (k == N)
   {  process(); return;   }
   enumerate(k+1);
   a[k] = 1;
   enumerate(k+1);
   a[k] = 0;
}
```

clean up

N = 3

| | | |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 0 | 1 |
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 0 | 1 | 1 |
| 0 | 1 | 0 |
| 0 | 0 | 0 |
| 1 | 0 | 0 |
| 1 | 0 | 1 |
| 1 | 0 | 0 |
| 1 | 1 | 0 |
| 1 | 1 | 1 |
| 1 | 1 | 0 |
| 1 | 0 | 0 |
| 0 | 0 | 0 |

N = 4

| | | | |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | 0 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 0 | 1 |
| 0 | 1 | 1 | 0 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 0 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 0 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 0 |
| 1 | 1 | 1 | 1 |

`a[0]`    `a[N-1]`

**Remark.** Equivalent to counting in binary from 0 to $2^N$ - 1.

# Warmup: enumerate N-bit strings

```
public class BinaryCounter
{
   private int N;   // number of bits
   private int[] a; // a[i] = ith bit

   public BinaryCounter(int N)
   {
      this.N = N;
      this.a = new int[N];
      enumerate(0);
   }

   private void process()
   {
      for (int i = 0; i < N; i++)
         StdOut.print(a[i]) + " ";
      StdOut.println();
   }

   private void enumerate(int k)
   {
     if (k == N)
     {  process(); return;  }
     enumerate(k+1);
     a[k] = 1;
     enumerate(k+1);
     a[k] = 0;
   }
}
```

```
public static void main(String[] args)
{
   int N = Integer.parseInt(args[0]);
   new BinaryCounter(N);
}
```

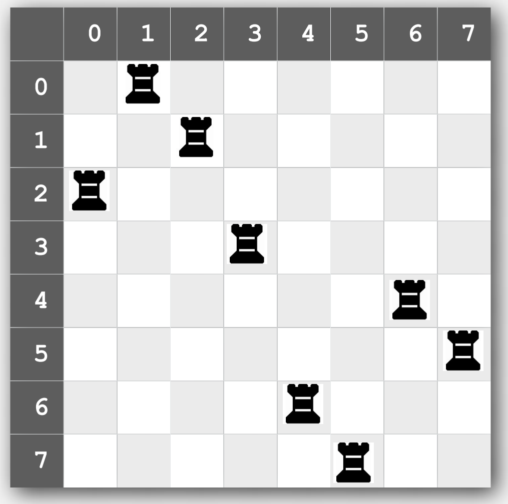
```
% java BinaryCounter 4
0 0 0 0
0 0 0 1
0 0 1 0
0 0 1 1
0 1 0 0
0 1 0 1
0 1 1 0
0 1 1 1
1 0 0 0
1 0 0 1
1 0 1 0
1 0 1 1
1 1 0 0
1 1 0 1
1 1 1 0
1 1 1 1
```

all programs in this lecture are variations on this theme

- permutations
- backtracking
- counting
- subsets
- paths in a graph

# N-rooks problem

Q. How many ways are there to place N rooks on an N-by-N board so that no rook can attack any other?




```
int[] a = { 2, 0, 1, 3, 6, 7, 4, 5 };
```

Representation. No two rooks in the same row or column $\Rightarrow$ permutation.

Challenge. Enumerate all N! permutations of 0 to N-1.

# Enumerating permutations

## Recursive algorithm to enumerate all N! permutations of size N.

- Start with permutation `a[0]` to `a[N-1]`.
- For each value of `i`:
  - swap `a[i]` into position `0`
  - enumerate all (N-1)! permutations of `a[1]` to `a[N-1]`
  - clean up (swap `a[i]` back to original position)

**`N = 2`**

| | |
|---|---|
| 0 | 1 |
| 1 | 0 |
| 0 | 1 |

**`N = 3`**

| | | |
|---|---|---|
| 0 | 1 | 2 |
| 0 | 2 | 1 |
| 0 | 1 | 2 |
| 1 | 0 | 2 |
| 1 | 2 | 0 |
| 1 | 0 | 2 |
| 0 | 1 | 2 |
| 2 | 1 | 0 |
| 2 | 0 | 1 |
| 2 | 1 | 0 |
| 0 | 1 | 2 |

cleanup swaps that bring perm back to original

0 followed by perms of 1 2 3

| | | | |
|---|---|---|---|
| 0 | 1 | 2 | 3 |
| 0 | 1 | 3 | 2 |
| 0 | 2 | 1 | 3 |
| 0 | 2 | 3 | 1 |
| 0 | 3 | 2 | 1 |
| 0 | 3 | 1 | 2 |

1 followed by perms of 0 2 3

| | | | |
|---|---|---|---|
| 1 | 0 | 2 | 3 |
| 1 | 0 | 3 | 2 |
| 1 | 2 | 0 | 3 |
| 1 | 2 | 3 | 0 |
| 1 | 3 | 2 | 0 |
| 1 | 3 | 0 | 2 |

2 followed by perms of 1 0 3

| | | | |
|---|---|---|---|
| 2 | 1 | 0 | 3 |
| 2 | 1 | 3 | 0 |
| 2 | 0 | 1 | 3 |
| 2 | 0 | 3 | 1 |
| 2 | 3 | 0 | 1 |
| 2 | 3 | 1 | 0 |

3 followed by perms of 1 2 0

| | | | |
|---|---|---|---|
| 3 | 1 | 2 | 0 |
| 3 | 1 | 0 | 2 |
| 3 | 2 | 1 | 0 |
| 3 | 2 | 0 | 1 |
| 3 | 0 | 2 | 1 |
| 3 | 0 | 1 | 2 |

`a[0]` `a[N-1]`

## Enumerating permutations

### Recursive algorithm to enumerate all N! permutations of size N.

- Start with permutation `a[0]` to `a[N-1]`.
- For each value of `i`:
  - swap `a[i]` into position `0`
  - enumerate all (N-1)! permutations of `a[1]` to `a[N-1]`
  - clean up (swap `a[i]` back to original position)

```
// place N-k rooks in a[k] to a[N-1]
private void enumerate(int k)
{
   if (k == N)
   {  process(); return;  }

   for (int i = k; i < N; i++)
   {
      exch(k, i);
      enumerate(k+1);
      exch(i, k);
   }
}
```

clean up

```
% java Rooks 4
0 1 2 3
0 1 3 2
0 2 1 3
0 2 3 1
0 3 2 1
0 3 1 2
1 0 2 3
1 0 3 2
1 2 0 3
1 2 3 0
1 3 2 0
1 3 0 2
2 1 0 3
2 1 3 0
2 0 1 3
2 0 3 1
2 3 0 1
2 3 1 0
3 1 2 0
3 1 0 2
3 2 1 0
3 2 0 1
3 0 2 1
3 0 1 2
```

0 followed by perms of 1 2 3

1 followed by perms of 0 2 3

2 followed by perms of 1 0 3

3 followed by perms of 1 2 0

`a[0]` `a[N-1]`

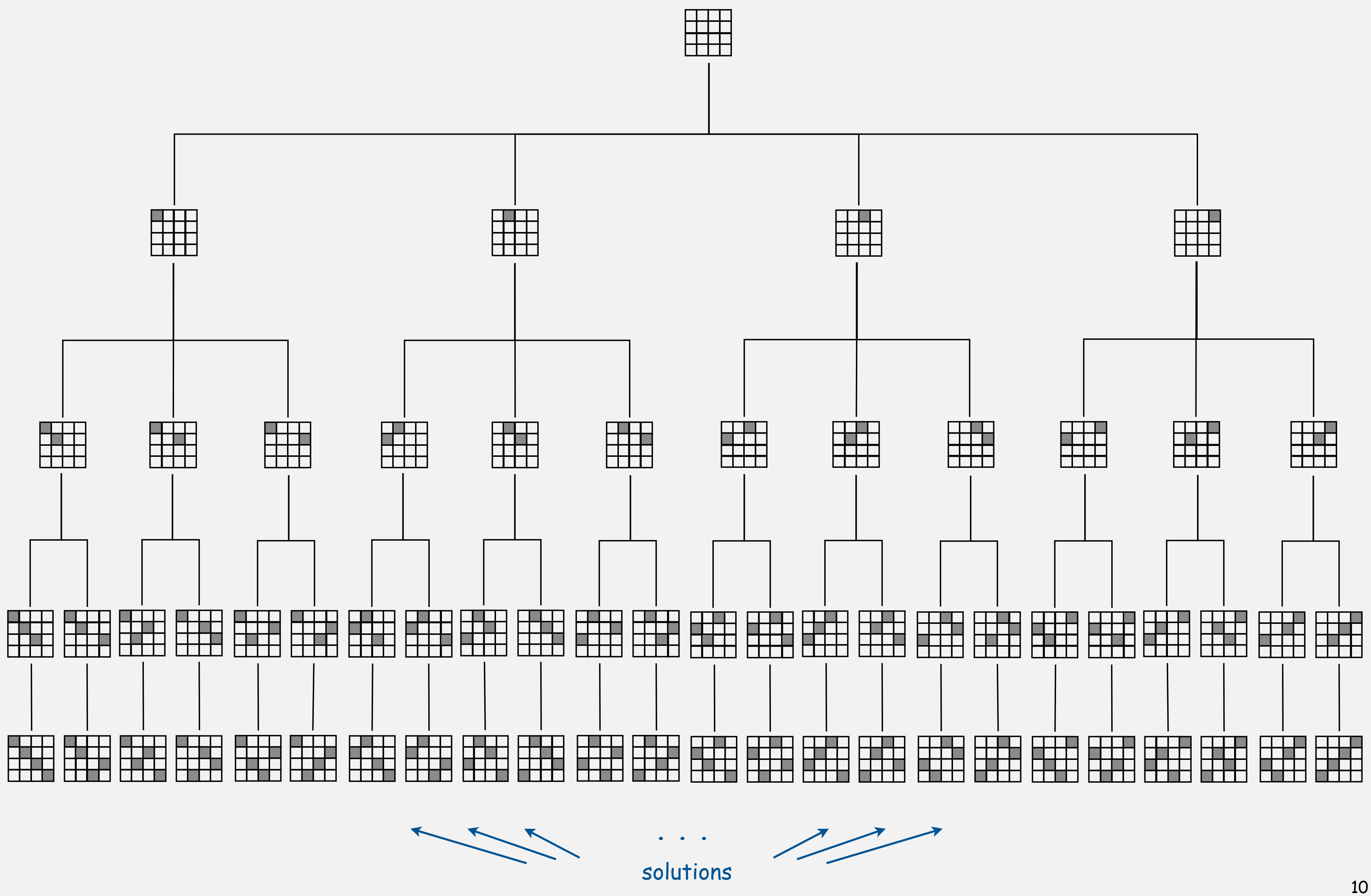
# Enumerating permutations

```
public class Rooks
{
   private int N;
   private int[] a; // bits (0 or 1)

   public Rooks(int N)
   {
      this.N = N;
      a = new int[N];
      for (int i = 0; i < N; i++)
         a[i] = i;
      enumerate(0);
   }

   private void enumerate(int k)
   { /* see previous slide */  }

   private void exch(int i, int j)
   {  int t = a[i]; a[i] = a[j]; a[j] = t;  }

   public static void main(String[] args)
   {
      int N = Integer.parseInt(args[0]);
      new Rooks(N);
   }
}
```

initial permutation

```
% java Rooks 2
0 1
1 0

% java Rooks 3
0 1 2
0 2 1
1 0 2
1 2 0
2 1 0
2 0 1
```

# 4-rooks search tree



# N-rooks problem: back-of-envelope running time estimate

Slow way to compute N!.

```
% java Rooks 7 | wc -l
5040

% java Rooks 8 | wc -l
40320

% java Rooks 9 | wc -l
362880

% java Rooks 10 | wc -l
3628800

% java Rooks 25 | wc -l
...
```

← instant

← 1.6 seconds

← 15 seconds

← 170 seconds

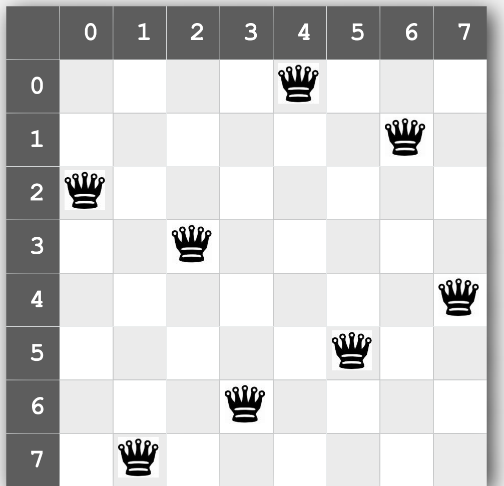
← forever

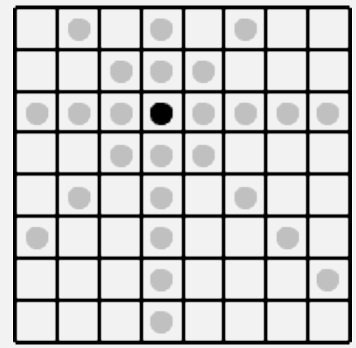
**Hypothesis.** Running time is about 2(N! / 8!) seconds.

- permutations
- **backtracking**
- counting
- subsets
- paths in a graph

# N-queens problem

Q. How many ways are there to place N queens on an N-by-N board so that no queen can attack any other?



```
int[] a = { 2, 7, 3, 6, 0, 5, 1, 4 };
```

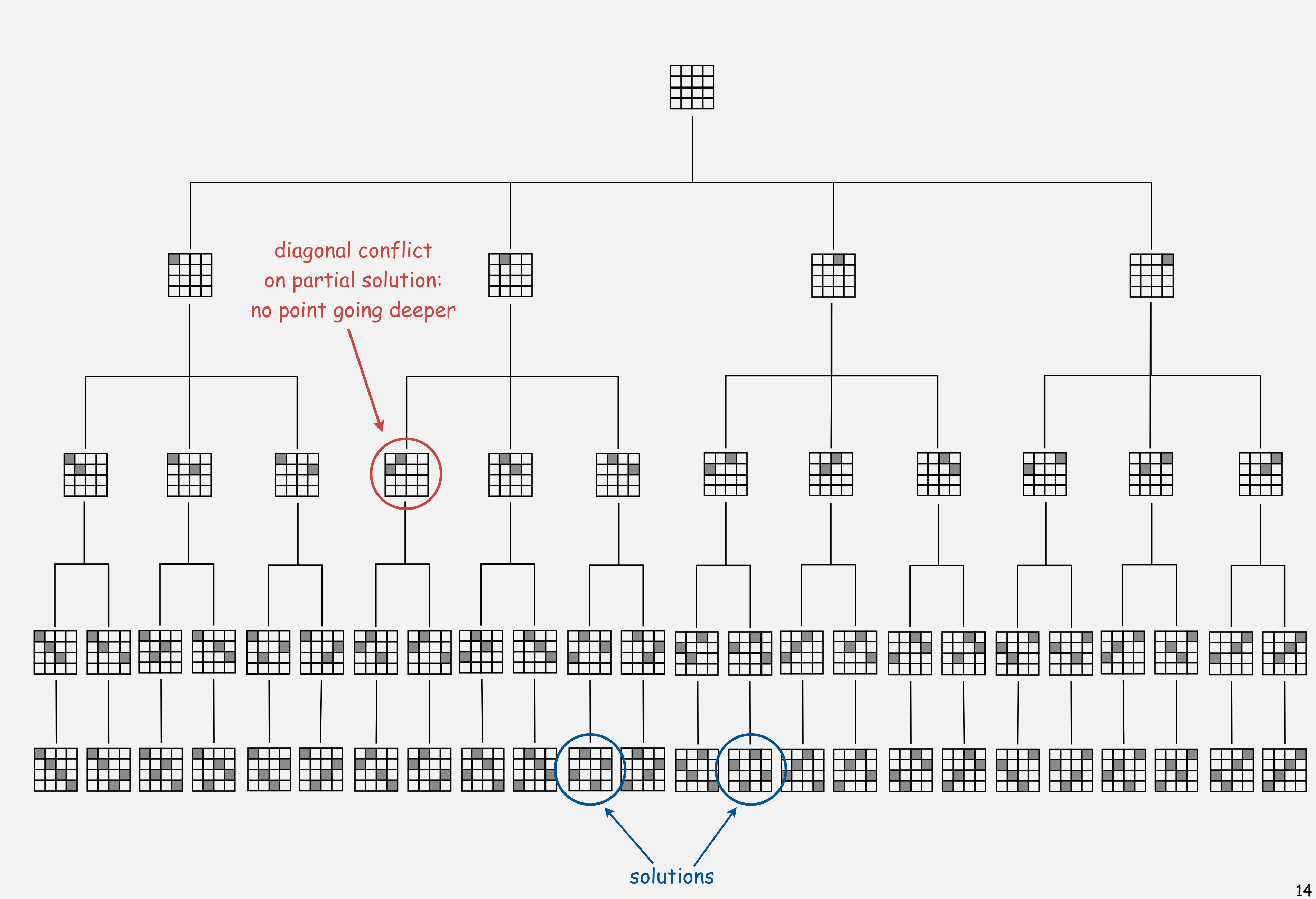
Representation. No two queens in the same row or column $\Rightarrow$ permutation.

Additional constraint. No diagonal attack is possible.

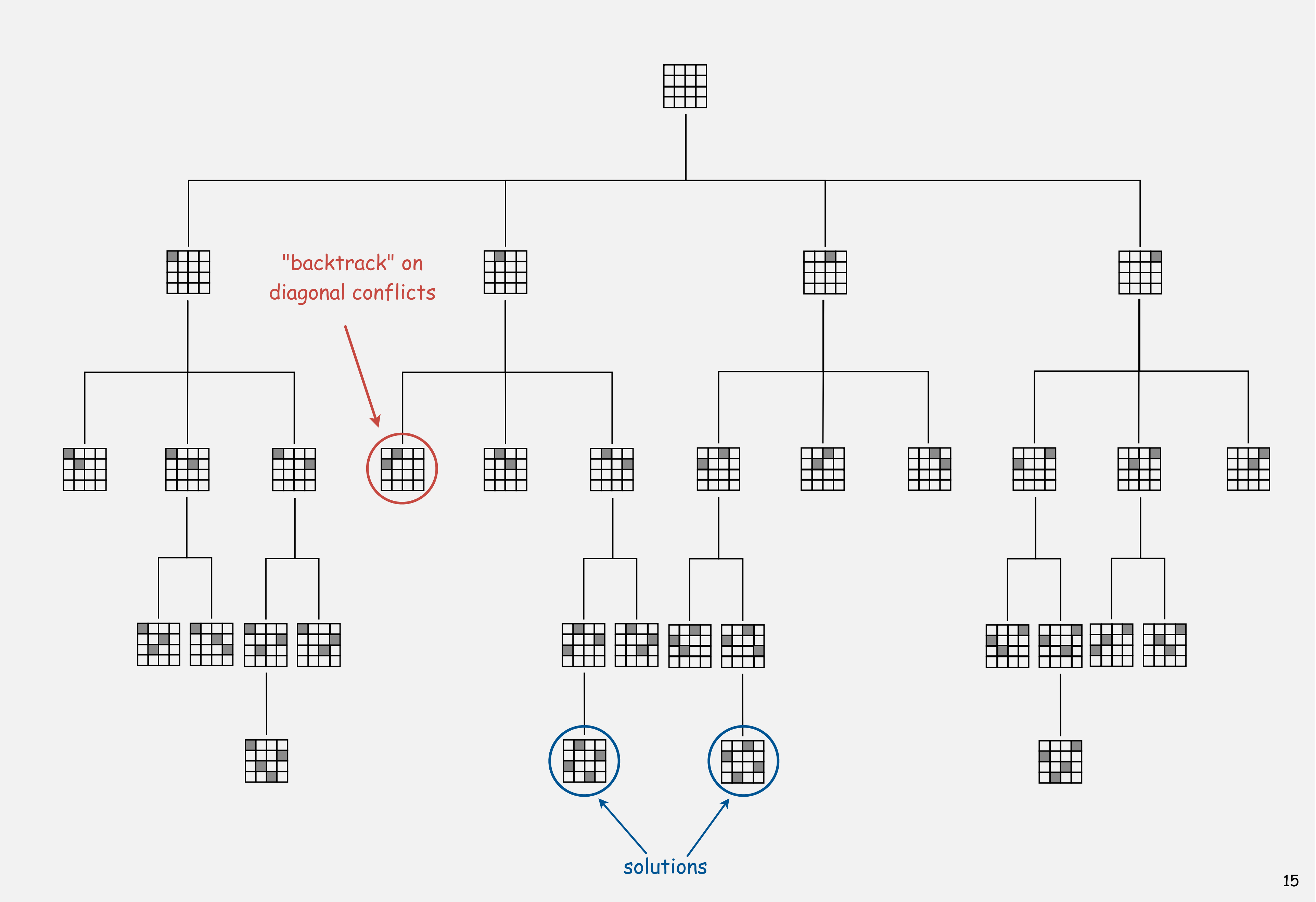
Challenge. Enumerate (or even count) the solutions. ←

unlike N-rooks problem, nobody knows answer for $N > 30$

# 4-queens search tree



## 4-queens search tree (pruned)



## N-queens problem: backtracking solution

Backtracking paradigm. Iterate through elements of search space.

- When there are several possible choices, make one choice and recur.
- If the choice is a dead end, backtrack to previous choice, and make next available choice.

Benefit. Identifying dead ends allows us to prune the search tree.

Ex. [backtracking for N-queens problem]

- Dead end: a diagonal conflict.
- Pruning: backtrack and try next column when diagonal conflict found.

# N-queens problem: backtracking solution

```
private boolean backtrack(int k)
{
   for (int i = 0; i < k; i++)
   {
      if ((a[i] - a[k]) == (k - i)) return true;
      if ((a[k] - a[i]) == (k - i)) return true;
   }
   return false;
}

// place N-k queens in a[k] to a[N-1]
private void enumerate(int k)
{
   if (k == N)
   {  process(); return;  }

   for (int i = k; i < N; i++)
   {
      exch(k, i);
      if (!backtrack(k)) enumerate(k+1);
      exch(i, k);
   }
}
```

stop enumerating if adding queen k leads to a diagonal violation

```
% java Queens 4
1 3 0 2
2 0 3 1

% java Queens 5
0 2 4 1 3
0 3 1 4 2
1 3 0 2 4
1 4 2 0 3
2 0 3 1 4
2 4 1 3 0
3 1 4 2 0
3 0 2 4 1
4 1 3 0 2
4 2 0 3 1

% java Queens 6
1 3 5 0 2 4
2 5 1 4 0 3
3 0 4 1 5 2
4 2 0 5 3 1
```

a[0]    a[N-1]

# N-queens problem: effectiveness of backtracking

Pruning the search tree leads to enormous time savings.

| $N$ | $Q(N)$ | $N!$ |
|---|---|---|
| 2 | 0 | 2 |
| 3 | 0 | 6 |
| 4 | 2 | 24 |
| 5 | 10 | 120 |
| 6 | 4 | 720 |
| 7 | 40 | 5,040 |
| 8 | 92 | 40,320 |
| 9 | 352 | 362,880 |
| 10 | 724 | 3,628,800 |
| 11 | 2,680 | 39,916,800 |
| 12 | 14,200 | 479,001,600 |
| 13 | 73,712 | 6,227,020,800 |
| 14 | 365,596 | 87,178,291,200 |

## N-queens problem: How many solutions?

```
% java Queens 13 | wc -l
73712

% java Queens 14 | wc -l
365596

% java Queens 15 | wc -l
2279184

% java Queens 16 | wc -l
14772512

% java Queens 17 | wc -l
...
```

← 1.1 seconds

← 5.4 seconds

← 29 seconds

← 210 seconds

← 1352 seconds

**Hypothesis.** Running time is about $(N! / 2.5^N) / 43{,}000$ seconds.

**Conjecture.** $Q(N)$ is $\sim N! / c^N$, where $c$ is about 2.54.

- permutations
- backtracking
- **counting**
- subsets
- paths in a graph

## Counting: Java implementation

**Goal.** Enumerate all N-digit base-R numbers.

**Solution.** Generalize binary counter in lecture warmup.

```
// enumerate base-R numbers in a[k] to a[N-1]
private static void enumerate(int k)
{
   if (k == N)
   {  process(); return;  }

   for (int r = 0; r < R; r++)
   {
      a[k] = r;
      enumerate(k+1);
   }
   a[k] = 0;
}
```

cleanup not needed; why?

```
% java Counter 2 4
0 0
0 1
0 2
0 3
1 0
1 1
1 2
1 3
2 0
2 1
2 2
2 3
3 0
3 1
3 2
3 3

% java Counter 3 2
0 0 0
0 0 1
0 1 0
0 1 1
1 0 0
1 0 1
1 1 0
1 1 1
```

a[0] a[N-1]

## Counting application: Sudoku

**Goal.** Fill 9-by-9 grid so that every row, column, and box contains each of the digits 1 through 9.

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 7 | | 8 | | | | 3 | | |
| | | | 2 | | 1 | | | |
| 5 | | | | | | | | |
| | 4 | | | | | | 2 | 6 |
| 3 | | | | 8 | | | | |
| | | | 1 | | | | 9 | |
| | 9 | | 6 | | | | | 4 |
| | | | | 7 | | 5 | | |
| | | | | | | | | |

**Remark.** Natural generalization is NP-complete.

# Counting application: Sudoku

Goal. Fill 9-by-9 grid so that every row, column, and box contains each of the digits 1 through 9.

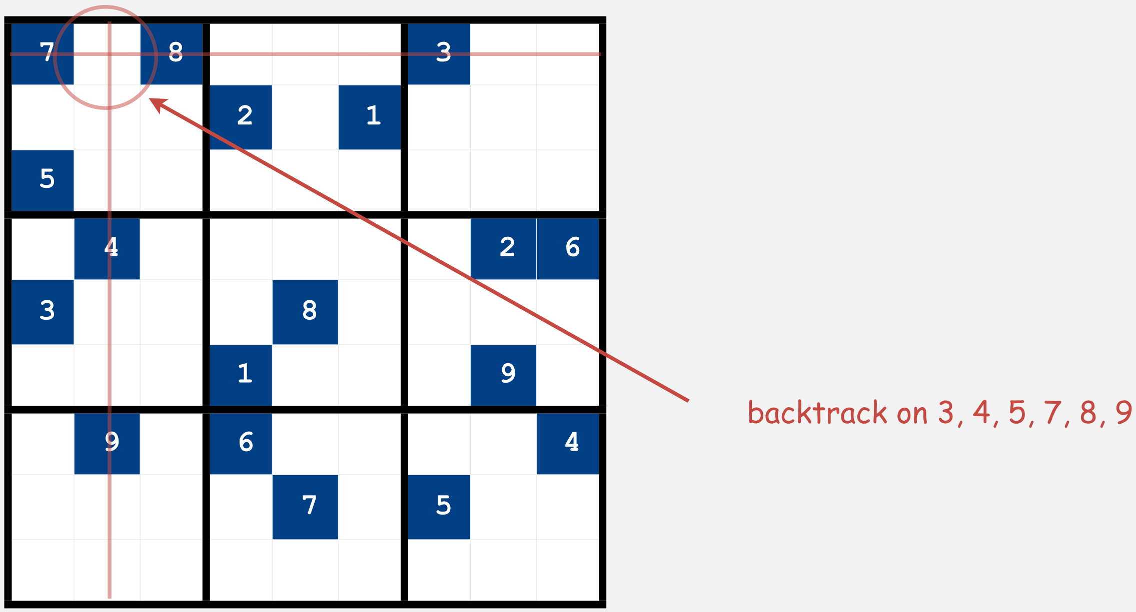
| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 7 | 2 | 8 | 9 | 4 | 6 | 3 | 1 | 5 |
| 9 | 3 | 4 | 2 | 5 | 1 | 6 | 7 | 8 |
| 5 | 1 | 6 | 7 | 3 | 8 | 2 | 4 | 9 |
| 1 | 4 | 7 | 5 | 9 | 3 | 8 | 2 | 6 |
| 3 | 6 | 9 | 4 | 8 | 2 | 1 | 5 | 7 |
| 8 | 5 | 2 | 1 | 6 | 7 | 4 | 9 | 3 |
| 2 | 9 | 3 | 6 | 1 | 5 | 7 | 8 | 4 |
| 4 | 8 | 1 | 3 | 7 | 9 | 5 | 6 | 2 |
| 6 | 7 | 5 | 8 | 2 | 4 | 9 | 3 | 1 |

Solution. Enumerate all 81-digit base-9 numbers (with backtracking).

## 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 find a conflict in row, column, or box, then backtrack.



# Sudoku: Java implementation

```
private void enumerate(int k)
{

   if (k == 81)
   {  process(); return;  }

   if (a[k] != 0)
   {  enumerate(k+1); return;  }

   for (int r = 1; r <= 9; r++)
   {
         a[k] = r;
         if (!backtrack(k))
            enumerate(k+1);
   }

   a[k] = 0;
}
```

found a solution

cell k initially filled in; recur on next cell

try 9 possible digits for cell k

unless it violates a Sudoku constraint (see booksite for code)

clean up

```
% more board.txt
7 0 8 0 0 0 3 0 0
0 0 0 2 0 1 0 0 0
5 0 0 0 0 0 0 0 0
0 4 0 0 0 0 0 2 6
3 0 0 0 8 0 0 0 0
0 0 0 1 0 0 0 9 0
0 9 0 6 0 0 0 0 4
0 0 0 0 7 0 5 0 0
0 0 0 0 0 0 0 0 0

% java Sudoku < board.txt
7 2 8 9 4 6 3 1 5
9 3 4 2 5 1 6 7 8
5 1 6 7 3 8 2 4 9
1 4 7 5 9 3 8 2 6
3 6 9 4 8 2 1 5 7
8 5 2 1 6 7 4 9 3
2 9 3 6 1 5 7 8 4
4 8 1 3 7 9 5 6 2
6 7 5 8 2 4 9 3 1
```

- permutations
- backtracking
- counting
- **subsets**
- paths in a graph

## Enumerating subsets: natural binary encoding

Given N items, enumerate all $2^N$ subsets.

- Count in binary from 0 to $2^N - 1$.
- Bit i represents item i.
- If 0, in subset; if 1, not in subset.

| *i* | *binary* | *subset* | *complement* |
|---|---|---|---|
| 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 |

# Enumerating subsets: natural binary encoding

Given N items, enumerate all $2^N$ subsets.

- Count in binary from 0 to $2^N - 1$.
- Maintain `a[i]` where `a[i]` represents item `i`.
- If `0`, `a[i]` in subset; if `1`, `a[i]` not in subset.

Binary counter from warmup does the job.

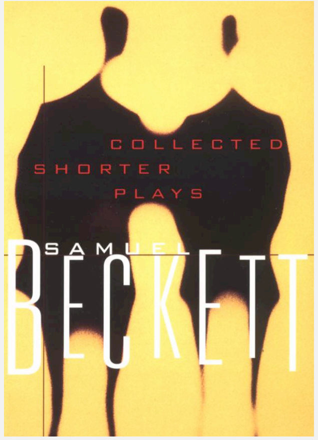
```
private void enumerate(int k)
{
  if (k == N)
  {  process(); return;  }
  enumerate(k+1);
  a[k] = 1;
  enumerate(k+1);
  a[n] = 0;
}
```

## Digression: Samuel Beckett play

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

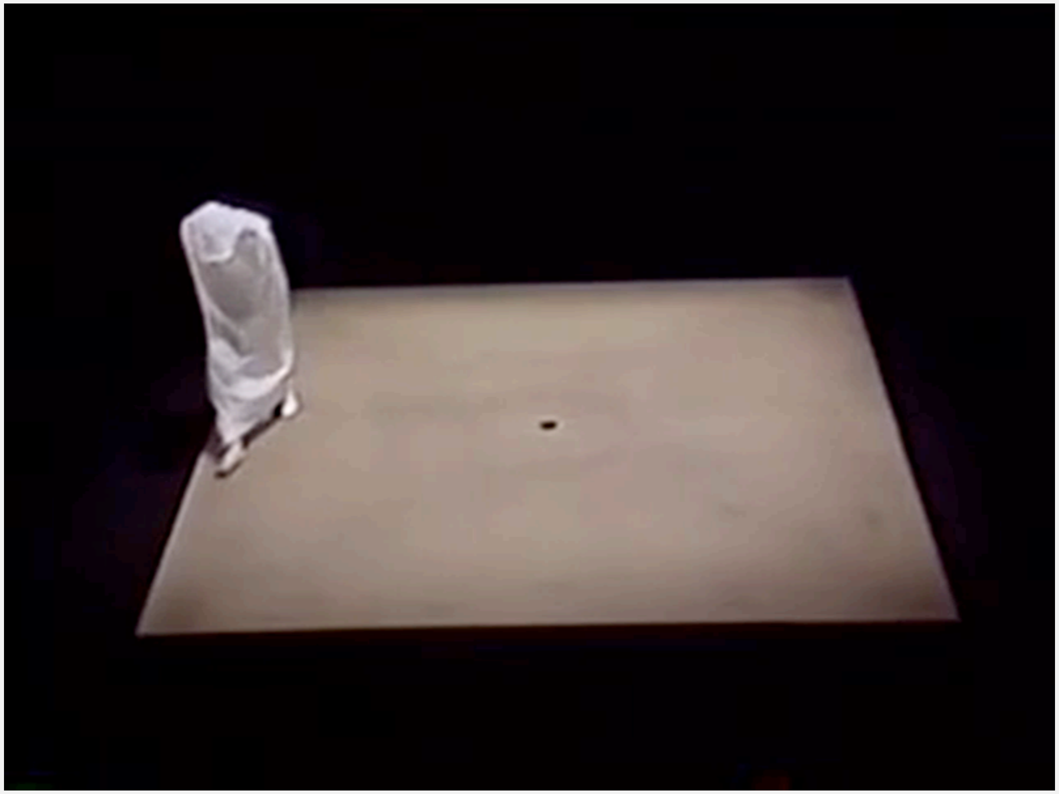
| code | subset | move |
|---|---|---|
| 0 0 0 0 | *empty* | |
| 0 0 0 1 | 1 | enter 1 |
| 0 0 1 1 | 2 1 | enter 2 |
| 0 0 1 0 | 2 | exit 1 |
| 0 1 1 0 | 3 2 | enter 3 |
| 0 1 1 1 | 3 2 1 | enter 1 |
| 0 1 0 1 | 3 1 | exit 2 |
| 0 1 0 0 | 3 | exit 1 |
| 1 1 0 0 | 4 3 | enter 4 |
| 1 1 0 1 | 4 3 1 | enter 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



## Digression: Samuel Beckett play

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

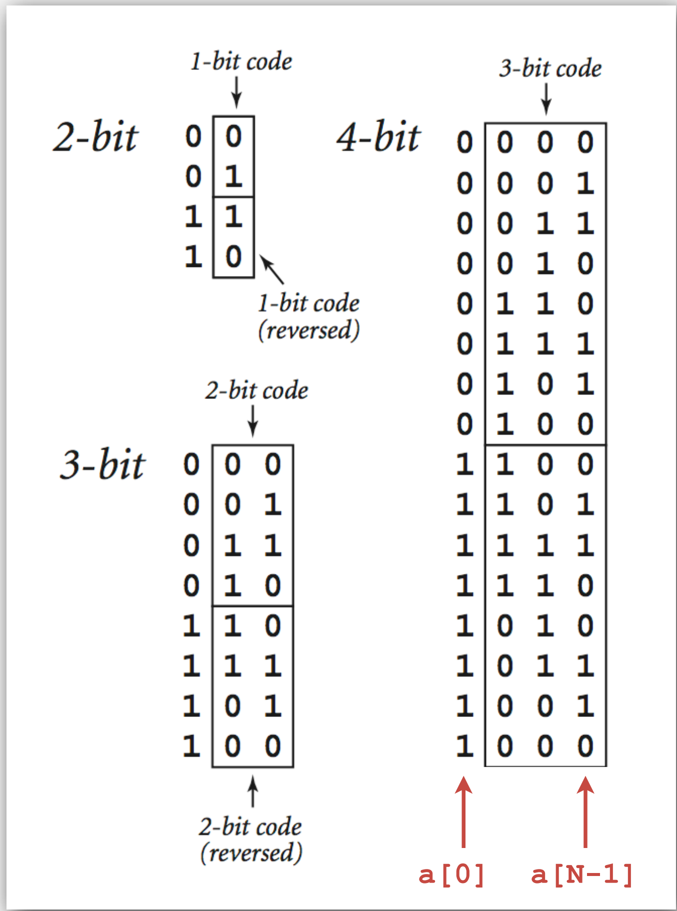


*"faceless, emotionless one of the far future, a world where people are born, go through prescribed movements, fear non-being even though their lives are meaningless, and then they disappear or die."* — Sidney Homan

# Binary reflected gray code

**Def.** The k-bit binary reflected Gray code is:

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



# Enumerating subsets using Gray code

Two simple changes to binary counter from warmup:

- Flip `a[k]` instead of setting it to 1.
- Eliminate cleanup.

*Gray code binary counter*

```
// all bit strings in a[k] to a[N-1]
private void enumerate(int k)
{
   if (k == N)
   { process(); return;  }
   enumerate(k+1);
   a[k] = 1 - a[k];
   enumerate(k+1);
}
```

| | | |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 0 | 1 |
| 0 | 1 | 1 |
| 0 | 1 | 0 |
| 1 | 1 | 0 |
| 1 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 0 | 0 |

same values
since no cleanup

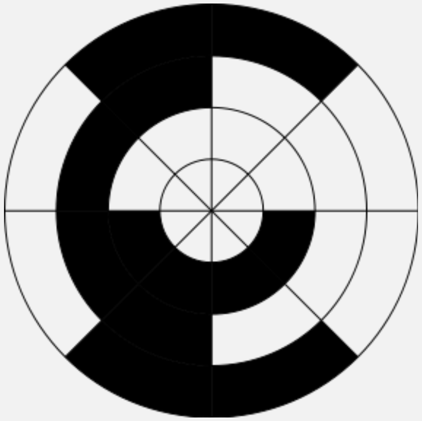
*standard binary counter (from warmup)*

```
// all bit strings in a[k] to a[N-1]
private void enumerate(int k)
{
   if (k == N)
   { process(); return;  }
   enumerate(k+1);
   a[k] = 1;
   enumerate(k+1);
   a[k] = 0;
}
```

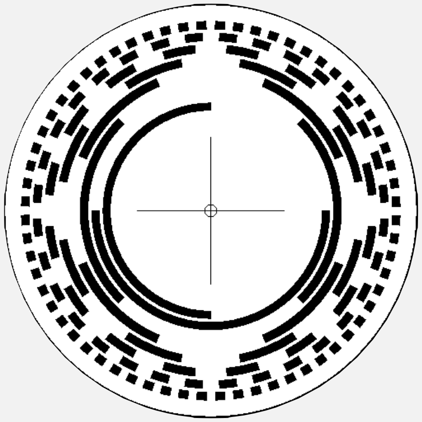
| a[0] | | a[N-1] |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 0 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |
| 1 | 1 | 1 |

Advantage. Only one item in subset changes at a time.

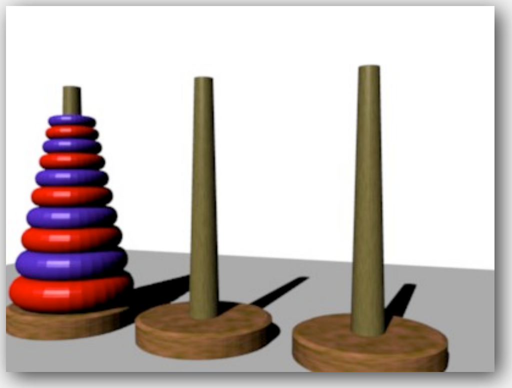
# More applications of Gray codes



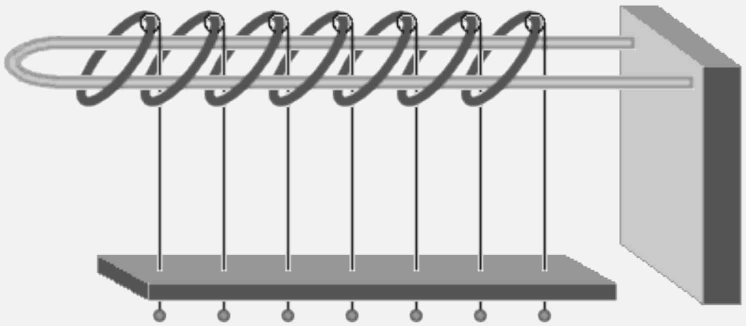
*3-bit rotary encoder*



*8-bit rotary encoder*



*Towers of Hanoi*



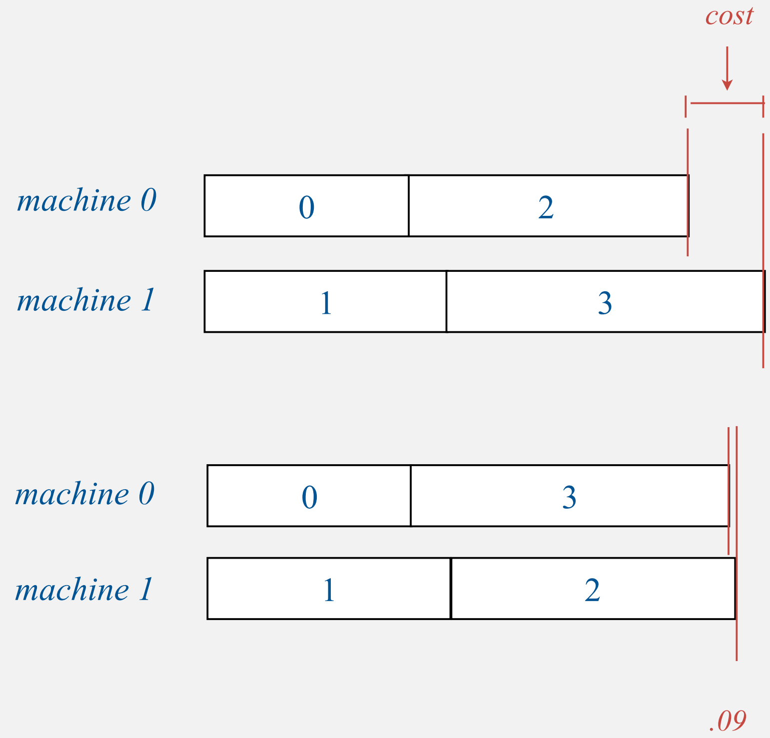
*Chinese ring puzzle*

# Scheduling

**Scheduling (set partitioning).** Given n jobs of varying length, divide among two machines to minimize the makespan (time the last job finishes).

or, equivalently, difference between finish times

| job | length |
|---|---|
| 0 | 1.41 |
| 1 | 1.73 |
| 2 | 2.00 |
| 3 | 2.23 |




**Remark.** This scheduling problem is NP-complete.

# Scheduling (full implementation)

```
public class Scheduler
{
   private int N;            // Number of jobs.
   private int[] a;          // Subset assignments.
   private int[] b;          // Best assignment.
   private double[] jobs;    // Job lengths.

   public Scheduler(double[] jobs)
   {
      this.N = jobs.length;
      this.jobs = jobs;
      a = new int[N];
      b = new int[N];
      enumerate(N);
   }

   public int[] best()
   {  return b;  }

   private void enumerate(int k)
   {  /* Gray code enumeration. */   }

   private void process()
   {
     if (cost(a) < cost(b))
       for (int i = 0; i < N; i++)
         b[i] = a[i];
   }

   public static void main(String[] args)
   {  /* create Scheduler, print results */  }
}
```

trace of

`% java Scheduler 4 < jobs.txt`

| a[] | | | | finish times | | cost |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 7.38 | 0.00 | 7.38 |
| 0 | 0 | 0 | 1 | 5.15 | 2.24 | 2.91 |
| 0 | 0 | 1 | 1 | 3.15 | 4.24 | 1.09 |
| 0 | 0 | 1 | 0 | 5.38 | 2.00 | |
| 0 | 1 | 1 | 0 | 3.65 | 3.73 | 0.08 |
| 0 | 1 | 1 | 1 | 1.41 | 5.97 | |
| 0 | 1 | 0 | 1 | 3.41 | 3.97 | |
| 0 | 1 | 0 | 0 | 5.65 | 1.73 | |
| 1 | 1 | 0 | 0 | 4.24 | 3.15 | |
| 1 | 1 | 0 | 1 | 2.00 | 5.38 | |
| 1 | 1 | 1 | 1 | 0.00 | 7.38 | |
| 1 | 1 | 1 | 0 | 2.24 | 5.15 | |
| 1 | 0 | 1 | 0 | 3.97 | 3.41 | |
| 1 | 0 | 1 | 1 | 1.73 | 5.65 | |
| 1 | 0 | 0 | 1 | 3.73 | 3.65 | |
| 1 | 0 | 0 | 0 | 5.97 | 1.41 | |

```
  MACHINE 0     MACHINE 1
 1.4142135624
               1.7320508076
               2.0000000000
 2.2360679775
-----------------------------
 3.6502815399  3.7320508076
```

# Scheduling (larger example)

Observation. Large number of subsets leads to remarkably low cost.

```
% java Scheduler < jobs.txt
 MACHINE 0       MACHINE 1
1.4142135624
1.7320508076
               2.0000000000
2.2360679775
2.4494897428
               2.6457513111
               2.8284271247
               3.0000000000
3.1622776602
               3.3166247904
               3.4641016151
               3.6055512755
               3.7416573868
3.8729833462
               4.0000000000
4.1231056256
               4.2426406871
4.3588989435
               4.4721359550
4.5825756950
4.6904157598
4.7958315233
4.8989794856
               5.0000000000
------------------------------
42.3168901295 42.3168901457
```

cost < $10^{-8}$

# Scheduling: improvements

Many opportunities (details omitted).

- Fix last job to be on machine 0 (quick factor-of-two improvement).
- Maintain difference in finish times (instead of recomputing from scratch).
- Backtrack when partial schedule cannot beat best known.
  (check total against goal: half of total job times)

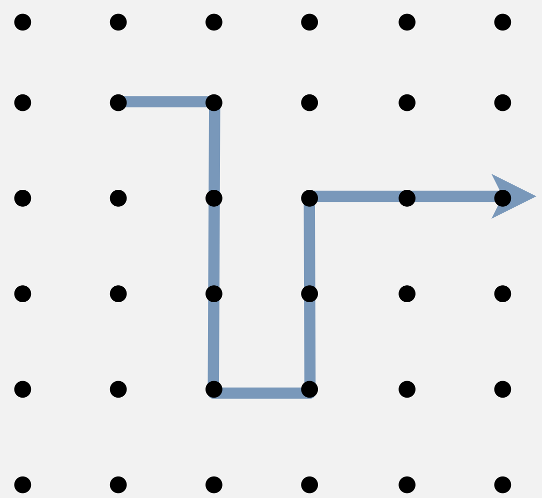
```
private void enumerate(int k)
{
   if (k == N-1)
   {  process(); return;  }
   if (backtrack(k)) return;
   enumerate(k+1);
   a[k] = 1 - a[k];
   enumerate(k+1);
}
```

- Process all $2^k$ subsets of last k jobs, keep results in memory,
  (reduces time to $2^{N-k}$ when $2^k$ memory available).

- permutations
- backtracking
- counting
- subsets
- **paths in a graph**

# Enumerating all paths on a grid

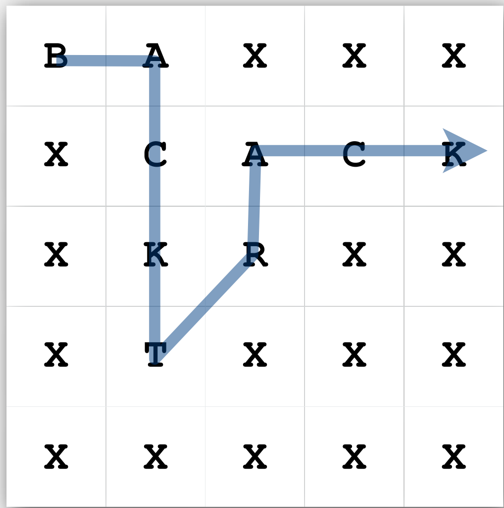
Goal. Enumerate all simple paths on a grid of adjacent sites.



no two atoms can occupy same position at same time

Application. Self-avoiding lattice walk to model polymer chains.

**Boggle.** Find all words that can be formed by tracing a simple path of adjacent cubes (left, right, up, down, diagonal).





**Pruning.** Stop as soon as no word in dictionary contains string of letters on current path as a prefix ⇒ use a trie.

```
B
BA
BAX
```

# Boggle: Java implementation

string of letters on current path to (i, j)

```
private void dfs(String prefix, int i, int j)
{
   if ((i < 0 || i >= N) ||
       (j < 0 || j >= N) ||
       (visited[i][j])   ||
       !dictionary.containsAsPrefix(prefix))
      return;

   visited[i][j] = true;
   prefix = prefix + board[i][j];

   if (dictionary.contains(prefix))
      found.add(prefix);

   for (int ii = -1; ii <= 1; ii++)
      for (int jj = -1; jj <= 1; jj++)
         dfs(prefix, i + ii, j + jj);

   visited[i][j] = false;
}
```

backtrack

add current character

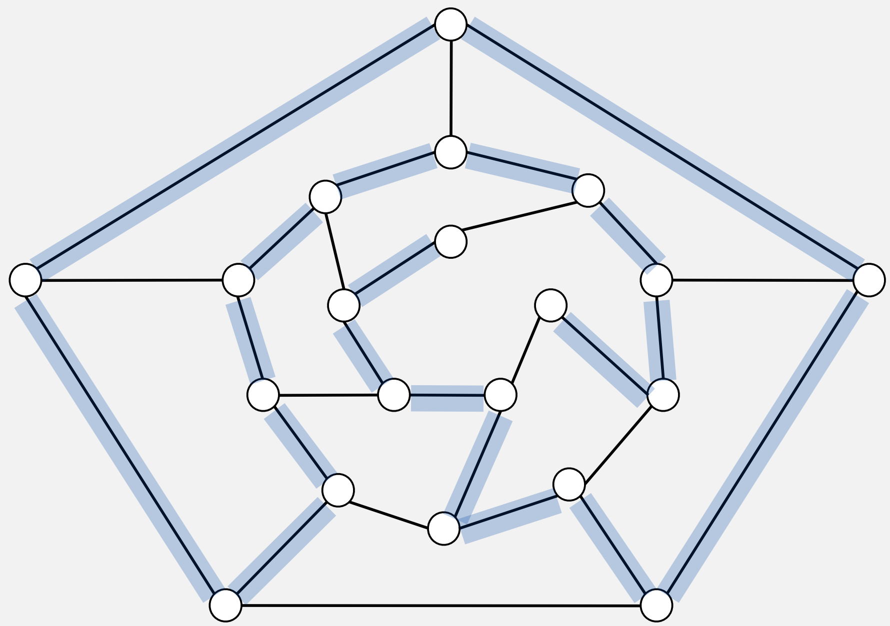
add to set of found words

try all possibilities

clean up

# Hamilton path

**Goal.** Find a simple path that visits every vertex exactly once.

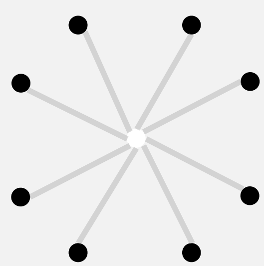


visit every edge exactly once

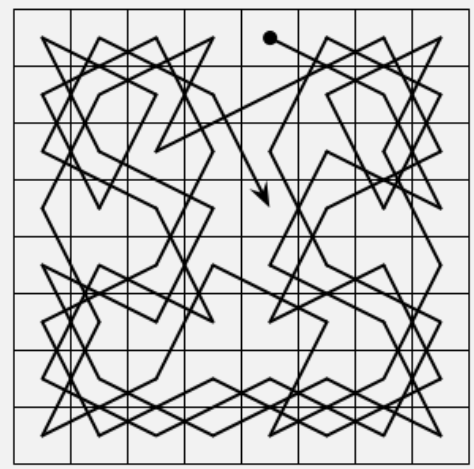
**Remark.** Euler path easy, but Hamilton path is NP-complete.

## Knight's tour

Goal. Find a sequence of moves for a knight so that (starting from any desired square) it visits every square on a chessboard exactly once.



*legal knight moves*



*a knight's tour*

Solution. Find a Hamilton path in knight's graph.

# Hamilton path: backtracking solution

**Backtracking solution.** To find Hamilton path starting at `v`:

- Add `v` to current path.
- For each vertex `w` adjacent to `v`
  - find a simple path starting at `w` using all remaining vertices
- Clean up: remove `v` from current path.

**Q.** How to implement?

**A.** Add cleanup to DFS (!!)

# Hamilton path: Java implementation

```
public class HamiltonPath
{
   private boolean[] marked;    // vertices on current path
   private int count = 0;       // number of Hamiltonian paths

   public HamiltonPath(Graph G)
   {
      marked = new boolean[G.V()];
      for (int v = 0; v < G.V(); v++)
         dfs(G, v, 1);
   }

   private void dfs(Graph G, int v, int depth)
   {
      marked[v] = true;
      if (depth == G.V()) count++;

      for (int w : G.adj(v))
         if (!marked[w]) dfs(G, w, depth+1);

      marked[v] = false;
   }
}
```

found one

length of current path (depth of recursion)

backtrack if w is already part of path

clean up

# Exhaustive search: summary

| problem | enumeration | backtracking |
|---|---|---|
| N-rooks | permutations | no |
| N-queens | permutations | yes |
| Sudoku | base-9 numbers | yes |
| scheduling | subsets | yes |
| Boggle | paths in a grid | yes |
| Hamilton path | paths in a graph | yes |

# The longest path

*Woh-oh-oh-oh, find the longest path!*
*Woh-oh-oh-oh, find the longest path!*

*If you said P is NP tonight,*
*There would still be papers left to write,*
*I have a weakness,*
*I'm addicted to completeness,*
*And I keep searching for the longest path.*

*The algorithm I would like to see*
*Is of polynomial degree,*
*But it's elusive:*
*Nobody has found conclusive*
*Evidence that we can find a longest path.*

*I have been hard working for so long.*
*I swear it's right, and he marks it wrong.*
*Some how I'll feel sorry when it's done: GPA 2.1*
*Is more than I hope for.*

*Garey, Johnson, Karp and other men (and women)*
*Tried to make it order N log N.*
*Am I a mad fool*
*If I spend my life in grad school,*
*Forever following the longest path?*

*Woh-oh-oh-oh, find the longest path!*
*Woh-oh-oh-oh, find the longest path!*
*Woh-oh-oh-oh, find the longest path.*

*Recorded by Dan Barrett in 1988*
*while a student at Johns Hopkins*
*during a difficult algorithms final*

# That's all, folks: Keep searching!

The world's longest path (Chile): 8500 km