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 ⇒ effectiveness may be limited to relatively small instances.

Backtracking. Systematic method for examining feasible solutions to a problem, by systematically pruning infeasible ones.
Warmup: enumerate N-bit strings

**Goal.** Process all $2^N$ bit strings of length $N$.
- Maintain array $a[]$ where $a[i]$ represents bit $i$.
- Simple recursive method does the job.

```java
// 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;
}
```

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

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

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

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

Recursive algorithm to enumerate all $N!$ permutations of $N$ elements.

- 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)

### Examples

- $N = 2$
- $N = 3$

![Permutation Examples](image-url)
Enumerating permutations

Recursive algorithm to enumerate all $N!$ permutations of $N$ elements.

- 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)

```java
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);
   }
}
```

// place N-k rooks in a[k] to a[N-1]
```java
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);
   }
}
```
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);
    }
}
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
...
```

---

**Hypothesis.** Running time is about $2 \left( \frac{N!}{8!} \right)$ 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?

![Chessboard diagram showing the placement of queens with coordinates]

```
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 the N-rooks problem, nobody knows the answer for $N > 30$. 

4-queens search tree

diagonal conflict on partial solution: no point going deeper

solutions
4-queens search tree (pruned)

"backtrack" on diagonal conflicts

solutions
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

```java
private boolean canBacktrack(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 (!canBacktrack(k)) enumerate(k+1);
        exch(i, k);
    }
}
```

**javab 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
```
N-queens problem: effectiveness of backtracking

Pruning the search tree leads to enormous time savings.

<table>
<thead>
<tr>
<th>N</th>
<th>Q(N)</th>
<th>N!</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>24</td>
</tr>
<tr>
<td>5</td>
<td>10</td>
<td>120</td>
</tr>
<tr>
<td>6</td>
<td>4</td>
<td>720</td>
</tr>
<tr>
<td>7</td>
<td>40</td>
<td>5,040</td>
</tr>
<tr>
<td>8</td>
<td>92</td>
<td>40,320</td>
</tr>
<tr>
<td>9</td>
<td>352</td>
<td>362,880</td>
</tr>
<tr>
<td>10</td>
<td>724</td>
<td>3,628,800</td>
</tr>
<tr>
<td>11</td>
<td>2,680</td>
<td>39,916,800</td>
</tr>
<tr>
<td>12</td>
<td>14,200</td>
<td>479,001,600</td>
</tr>
<tr>
<td>13</td>
<td>73,712</td>
<td>6,227,020,800</td>
</tr>
<tr>
<td>14</td>
<td>365,596</td>
<td>87,178,291,200</td>
</tr>
</tbody>
</table>
N-queens problem: How many solutions?

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

Conjecture. \( Q(N) \sim \frac{N!}{c^N} \), 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.

```java
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?
}
```
Counting application: Sudoku

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

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.

![Sudoku grid]

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

<table>
<thead>
<tr>
<th>a[]</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>2</td>
</tr>
</tbody>
</table>

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

```
    7 8 3
5 2 1
4 8 6
3 9 4
2 6 7
9 5 4
```
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 (!canBacktrack(k))
           enumerate(k+1);
    }
    a[k] = 0;
}
› permutations
› backtracking
› counting
› subsets
› paths in a graph
Given $N$ elements, enumerate all $2^N$ subsets.

- Count in binary from 0 to $2^N - 1$.
- Bit $i$ represents element $i$.
- If 1, in subset; if 0, not in subset.
Enumerating subsets: natural binary encoding

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

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

Binary counter from warmup does the job.

```java
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.

<table>
<thead>
<tr>
<th>code</th>
<th>subset</th>
<th>move</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0</td>
<td>empty</td>
<td></td>
</tr>
<tr>
<td>0 0 0 1</td>
<td>1</td>
<td>enter 1</td>
</tr>
<tr>
<td>0 0 1 1</td>
<td>2 1</td>
<td>enter 2</td>
</tr>
<tr>
<td>0 0 1 0</td>
<td>2</td>
<td>exit 1</td>
</tr>
<tr>
<td>0 1 1 0</td>
<td>3 2</td>
<td>enter 3</td>
</tr>
<tr>
<td>0 1 1 1</td>
<td>3 2 1</td>
<td>enter 1</td>
</tr>
<tr>
<td>0 1 0 1</td>
<td>3 1</td>
<td>exit 2</td>
</tr>
<tr>
<td>0 1 0 0</td>
<td>3</td>
<td>exit 1</td>
</tr>
<tr>
<td>1 1 0 0</td>
<td>4 3</td>
<td>enter 4</td>
</tr>
<tr>
<td>1 1 0 1</td>
<td>4 3 1</td>
<td>enter 1</td>
</tr>
<tr>
<td>1 1 1 1</td>
<td>4 3 2 1</td>
<td>enter 2</td>
</tr>
<tr>
<td>1 1 1 0</td>
<td>4 3 2</td>
<td>exit 1</td>
</tr>
<tr>
<td>1 0 1 0</td>
<td>4 2</td>
<td>exit 3</td>
</tr>
<tr>
<td>1 0 1 1</td>
<td>4 2 1</td>
<td>enter 1</td>
</tr>
<tr>
<td>1 0 0 1</td>
<td>4 1</td>
<td>exit 2</td>
</tr>
<tr>
<td>1 0 0 0</td>
<td>4</td>
<td>exit 1</td>
</tr>
</tbody>
</table>

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.

```
<table>
<thead>
<tr>
<th>2-bit</th>
<th>4-bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>3-bit</th>
<th>1-bit code (reversed)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0 0 0</td>
</tr>
<tr>
<td>0</td>
<td>0 0 1</td>
</tr>
<tr>
<td>0</td>
<td>0 1 1</td>
</tr>
<tr>
<td>0</td>
<td>0 1 0</td>
</tr>
<tr>
<td>1</td>
<td>1 1 0</td>
</tr>
<tr>
<td>1</td>
<td>1 1 1</td>
</tr>
<tr>
<td>1</td>
<td>1 0 1</td>
</tr>
<tr>
<td>1</td>
<td>1 0 0</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>2-bit code (reversed)</th>
<th>a[0]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 0 0</td>
<td></td>
</tr>
<tr>
<td>1 1 0 1</td>
<td></td>
</tr>
<tr>
<td>1 1 1 1</td>
<td></td>
</tr>
<tr>
<td>1 1 1 0</td>
<td></td>
</tr>
<tr>
<td>1 0 1 0</td>
<td></td>
</tr>
<tr>
<td>1 0 1 1</td>
<td></td>
</tr>
<tr>
<td>1 0 0 1</td>
<td></td>
</tr>
<tr>
<td>1 0 0 0</td>
<td></td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>3-bit code</th>
<th>a[N-1]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0</td>
<td></td>
</tr>
<tr>
<td>0 0 0 1</td>
<td></td>
</tr>
<tr>
<td>0 0 1 1</td>
<td></td>
</tr>
<tr>
<td>0 0 1 0</td>
<td></td>
</tr>
<tr>
<td>0 1 1 0</td>
<td></td>
</tr>
<tr>
<td>0 1 1 1</td>
<td></td>
</tr>
<tr>
<td>0 1 0 1</td>
<td></td>
</tr>
<tr>
<td>0 1 0 0</td>
<td></td>
</tr>
<tr>
<td>1 1 0 0</td>
<td></td>
</tr>
<tr>
<td>1 1 0 1</td>
<td></td>
</tr>
<tr>
<td>1 1 1 1</td>
<td></td>
</tr>
<tr>
<td>1 1 1 0</td>
<td></td>
</tr>
<tr>
<td>1 0 1 0</td>
<td></td>
</tr>
<tr>
<td>1 0 1 1</td>
<td></td>
</tr>
<tr>
<td>1 0 0 1</td>
<td></td>
</tr>
<tr>
<td>1 0 0 0</td>
<td></td>
</tr>
<tr>
<td>1 1 0 0</td>
<td></td>
</tr>
<tr>
<td>1 1 0 1</td>
<td></td>
</tr>
<tr>
<td>1 1 1 1</td>
<td></td>
</tr>
<tr>
<td>1 1 1 0</td>
<td></td>
</tr>
<tr>
<td>1 0 1 0</td>
<td></td>
</tr>
<tr>
<td>1 0 1 1</td>
<td></td>
</tr>
<tr>
<td>1 0 0 1</td>
<td></td>
</tr>
<tr>
<td>1 0 0 0</td>
<td></td>
</tr>
</tbody>
</table>
```
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**

```java
// 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);
}
```

**Standard binary counter (from warmup)**

```java
// 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;
}
```

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
(move ith smallest disk when bit i changes in Gray code)

Chinese ring puzzle (Baguenaudier)
(move ith ring from right when bit i changes in Gray code)
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

<table>
<thead>
<tr>
<th>job</th>
<th>length</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1.41</td>
</tr>
<tr>
<td>1</td>
<td>1.73</td>
</tr>
<tr>
<td>2</td>
<td>2.00</td>
</tr>
<tr>
<td>3</td>
<td>2.23</td>
</tr>
</tbody>
</table>

Remark. This scheduling problem is \( \text{NP-complete} \).
Scheduling (full implementation)

```java
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 */ }
}
```

<table>
<thead>
<tr>
<th>a[]</th>
<th>finish times</th>
<th>cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0</td>
<td>7.38</td>
<td>0.00</td>
</tr>
<tr>
<td>0 0 0 1</td>
<td>5.15</td>
<td>2.24</td>
</tr>
<tr>
<td>0 0 1 1</td>
<td>3.15</td>
<td>4.24</td>
</tr>
<tr>
<td>0 0 1 0</td>
<td>5.38</td>
<td>2.00</td>
</tr>
<tr>
<td>0 1 1 0</td>
<td>3.65</td>
<td>3.73</td>
</tr>
<tr>
<td>0 1 1 1</td>
<td>1.41</td>
<td>5.97</td>
</tr>
<tr>
<td>0 1 0 1</td>
<td>3.41</td>
<td>3.97</td>
</tr>
<tr>
<td>0 1 0 0</td>
<td>5.65</td>
<td>1.73</td>
</tr>
<tr>
<td>1 1 0 0</td>
<td>4.24</td>
<td>3.15</td>
</tr>
<tr>
<td>1 1 0 1</td>
<td>2.00</td>
<td>5.38</td>
</tr>
<tr>
<td>1 1 1 1</td>
<td>0.00</td>
<td>7.38</td>
</tr>
<tr>
<td>1 1 1 0</td>
<td>2.24</td>
<td>5.15</td>
</tr>
<tr>
<td>1 0 1 0</td>
<td>3.97</td>
<td>3.41</td>
</tr>
<tr>
<td>1 0 1 1</td>
<td>1.73</td>
<td>5.65</td>
</tr>
<tr>
<td>1 0 0 1</td>
<td>3.73</td>
<td>3.65</td>
</tr>
<tr>
<td>1 0 0 0</td>
<td>5.97</td>
<td>1.41</td>
</tr>
</tbody>
</table>

MACHINE 0 | MACHINE 1
-----------|-----------
1.41       | 1.73      |
1.73       | 2.00      |
2.24       | 3.65      |
3.65       | 3.73      |
```
Scheduling: improvements

Many opportunities to improve.

- Fix last job to be on machine 0.
- Maintain difference in finish times.
  
  (and avoid recomputing cost from scratch)
- Backtrack when partial schedule cannot beat best known.

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

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

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

no two atoms can occupy same position at same time
Enumerating all paths on a grid: Boggle

**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
```
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;
}

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

backtrack
add current character
add to set of found words
try all possibilities
clean up

Boggle: Java implementation
Hamilton path

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

**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 (!!)
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;
    }
}
## Exhaustive search: summary

<table>
<thead>
<tr>
<th>problem</th>
<th>enumeration</th>
<th>backtracking</th>
</tr>
</thead>
<tbody>
<tr>
<td>N-rooks</td>
<td>permutations</td>
<td>no</td>
</tr>
<tr>
<td>N-queens</td>
<td>permutations</td>
<td>yes</td>
</tr>
<tr>
<td>Sudoku</td>
<td>base-9 numbers</td>
<td>yes</td>
</tr>
<tr>
<td>scheduling</td>
<td>subsets</td>
<td>yes</td>
</tr>
<tr>
<td>Boggle</td>
<td>paths in a grid</td>
<td>yes</td>
</tr>
<tr>
<td>Hamilton path</td>
<td>paths in a graph</td>
<td>yes</td>
</tr>
</tbody>
</table>
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 (Sendero de Chile): 9,700 km.
(originally scheduled for completion in 2010; now delayed until 2038)