Combinatorial Search

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

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

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

**Applicability.** Huge range of problems (include NP-hard ones).

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

**Caveat to the caveat.** Backtracking may prune search space to reasonable size, even for relatively large instances.
Warmup: enumerate N-bit strings

Problem: process all $2^N$ N-bit strings (stay tuned for applications).

Equivalent to counting in binary from 0 to $2^N - 1$.
- maintain $a[i]$ where $a[i]$ represents bit $i$
- initialize all bits to 0
- simple recursive method does the job
  (call enumerate(0))

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

Invariant (prove by induction);
Enumerates all $(N-k)$-bit strings and cleans up after itself.
Warmup: enumerate N-bit strings (full implementation)

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

```java
public class Counter {
    private int N;   // number of bits
    private int[] a; // bits (0 or 1)

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

    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]);
        Counter c = new Counter(N);
    }
}
```

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

all the programs in this lecture are variations on this theme

```java
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]);
    Counter c = new Counter(N);
}
```

% java Counter 4
0000
0001
0010
0011
0100
0101
0110
0111
1000
1001
1010
1011
1100
1101
1110
1111
permutations
backtracking
counting
subsets
paths in a graph
N-rooks Problem

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

No two in the same row, so represent solution with an array
\[ a[i] = \text{column of rook in row } i. \]

No two in the same column, so array entries are all different
\[ a[] \text{ is a permutation (rearrangement of 0, 1, ... N-1)} \]

**Answer:** There are \( N! \) non mutually-attacking placements.

**Challenge:** Enumerate them all.

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

Recursive algorithm to enumerate all N! permutations of size N:

- Start with \(0 \ 1 \ 2 \ ... \ N-1\).
- For each value of \(i\),
  - swap \(i\) into position 0
  - enumerate all (N-1)! arrangements of \(a[1..N-1]\)
  - clean up (swap \(i\) and 0 back into position)

\[
\begin{array}{c|c|c|c|c}
0 & 1 & 2 & 3 & 4 \\
1 & 2 & 3 & 4 & 0 \\
2 & 3 & 4 & 2 & 1 \\
3 & 4 & 0 & 1 & 2 \\
4 & 0 & 1 & 2 & 3 \\
\end{array}
\]

Example showing cleanup swaps that bring perm back to original
public class Rooks
{
    private int N;
    private int[] a;

    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 next slide. */
    }

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

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

    public static void main(String[] args)
    {
        int N = Integer.parseInt(args[0]);
        Rooks t = new Rooks(N);
        t.enumerate(0);
    }
}
N-rooks problem (enumerating all permutations): recursive enumeration

Recursive algorithm to enumerate all $N!$ permutations of size $N$:

- Start with $0 \ 1 \ 2 \ldots \ N-1$.
- For each value of $i$
  - swap $i$ into position $0$
  - enumerate all $(N-1)!$ arrangements of $a[1..N-1]$
  - clean up (swap $i$ and $0$ back into position)

```java
private void enumerate(int k)
{
    if (k == N)
    {
        process();
        return;
    }
    for (int i = k; i < N; i++)
    {
        exch(a, k, i);
        enumerate(k+1);
        exch(a, k, i);
    }
}
```
4-Rooks search tree
N-rooks problem: back-of-envelope running time estimate

[ Studying slow way to compute N! but good warmup for calculations.]

```
% java Rooks 10
3628800 solutions

% java Rooks 11
39916800 solutions

% java Rooks 12
479001600 solutions
```

Hypothesis: Running time is about $2(N! / 11!)$ seconds.

```
% java Rooks 25

```

millions of centuries

Google: $2 \times \frac{25!}{11!}$ seconds = 246,277,800 centuries

More about calculator.

Search for documents containing the terms $2(25/11)!$ seconds in centuries.
- permutations
- backtracking
- counting
- subsets
- paths in a graph
How many ways are there to place 
N queens on an N-by-N board so that no queen can attack any other?

Representation. Same as for rooks: 
represent solution as a permutation: \(a[i] = \text{column of queen in row } i\).

Additional constraint: no diagonal attack is possible

Challenge: Enumerate (or even count) the solutions
4-Queens search tree

diagonal conflict on partial solution: no point going deeper

solutions
Iterate through elements of search space.
- when there are $N$ possible choices, make one choice and recur.
- if the choice is a dead end, backtrack to previous choice, and make next available choice.

Identifying dead ends allows us to prune the search tree

For $N$ queens:
- dead end: a diagonal conflict
- pruning: backtrack and try next row when diagonal conflict found

In general, improvements are possible:
- try to make an “intelligent” choice
- try to reduce cost of choosing/backtracking
4-Queens Search Tree (pruned)

Backtrack on diagonal conflicts

solutions
N-Queens: 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;
}

private void enumerate(int k)
{
    if (k == N)
    {
        process();
        return;
    }
    for (int i = k; i < N; i++)
    {
        exch(a, k, i);
        if (!backtrack(k)) enumerate(k+1);
        exch(a, k, i);
    }
}
**N-Queens: 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>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>
<tr>
<td>15</td>
<td>2,279,184</td>
<td>1,307,674,368,000</td>
</tr>
<tr>
<td>16</td>
<td>14,772,512</td>
<td>20,922,789,888,000</td>
</tr>
</tbody>
</table>

savings: factor of more than 1-million
N-Queens: How many solutions?

Answer to original question easy to obtain:
- add an instance variable to count solutions (initialized to 0)
- change `process()` to increment the counter
- add a method to return its value

<table>
<thead>
<tr>
<th>N</th>
<th>Q(N)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>10</td>
</tr>
<tr>
<td>5</td>
<td>40</td>
</tr>
<tr>
<td>6</td>
<td>92</td>
</tr>
<tr>
<td>7</td>
<td>352</td>
</tr>
<tr>
<td>8</td>
<td>2,680</td>
</tr>
<tr>
<td>9</td>
<td>14,200</td>
</tr>
<tr>
<td>10</td>
<td>73,712</td>
</tr>
<tr>
<td>11</td>
<td>365,596</td>
</tr>
<tr>
<td>12</td>
<td>2,279,184</td>
</tr>
<tr>
<td>13</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>14,772,512</td>
</tr>
<tr>
<td>17</td>
<td>95,815,104</td>
</tr>
<tr>
<td>18</td>
<td>666,090,624</td>
</tr>
<tr>
<td>19</td>
<td>4,968,057,848</td>
</tr>
<tr>
<td>20</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td></td>
</tr>
</tbody>
</table>

Source: On-line encyclopedia of integer sequences, N. J. Sloane [ sequence A000170 ]

% java Queens 4
2 solutions

% java Queens 8
92 solutions

% java Queens 16
14772512 solutions

took 53 years of CPU time (2005)
N-queens problem: back-of-envelope running time estimate

Hypothesis ??

% java Queens 13
73712 solutions
about a second

% java Queens 14
365596 solutions
about 7 seconds

% java Queens 15
2279184 solutions
about 49 seconds

% java Queens 16
14772512 solutions
about 360 seconds

Hypothesis: Running time is about \((N/2)!\) seconds.

% java Queens 25
about 54 years

ratio
6.32
6.73
7.38
- permutations
- backtracking
- counting
- subsets
- paths in a graph
Problem: enumerate all N-digit base-R numbers
Solution: generalize binary counter in lecture warmup

enumerate N-digit base-R numbers

```java
class Example {
    private static void enumerate(int k) {
        if (k == N) {
            process(); return;
        }
        for (int n = 0; n < R; n++) {
            a[k] = n;
            enumerate(k + 1);
        }
        a[k] = 0;
    }
}
```

enumerate binary numbers (from warmup)

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

clean up not needed: Why?

element showing cleanups that zero out digits

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

0 0 0
0 2 0
0 2 0
0 2 0

0 0 0
0 2 0
0 2 0
0 2 0
Counting application: Sudoku

Problem:
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-hard.
Counting application: Sudoku

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

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.

Improvements are possible.
• try to make an "intelligent" choice
• try to reduce cost of choosing/backtracking
Sudoku: Java implementation

```java
private static void solve(int cell) {
    if (cell == 81) {
        show(board); return;
    }
    if (board[cell] != 0) {
        solve(cell + 1); return;
    }
    for (int n = 1; n <= 9; n++) {
        if (!backtrack(cell, n)) {
            board[cell] = n;
            solve(cell + 1);
        }
    }
    board[cell] = 0;
}
```

Works remarkably well (plenty of constraints). Try it!
- 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

<table>
<thead>
<tr>
<th>i</th>
<th>binary</th>
<th>subset</th>
<th>complement</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0 0 0 0</td>
<td>empty</td>
<td>4 3 2 1</td>
</tr>
<tr>
<td>1</td>
<td>0 0 0 1</td>
<td>1</td>
<td>4 3 2</td>
</tr>
<tr>
<td>2</td>
<td>0 0 1 0</td>
<td>2</td>
<td>4 3 1</td>
</tr>
<tr>
<td>3</td>
<td>0 0 1 1</td>
<td>2 1</td>
<td>4 3</td>
</tr>
<tr>
<td>4</td>
<td>0 1 0 0</td>
<td>3</td>
<td>4 2 1</td>
</tr>
<tr>
<td>5</td>
<td>0 1 0 1</td>
<td>3 1</td>
<td>4 2</td>
</tr>
<tr>
<td>6</td>
<td>0 1 1 0</td>
<td>3 2</td>
<td>4 1</td>
</tr>
<tr>
<td>7</td>
<td>0 1 1 1</td>
<td>3 2 1</td>
<td>4</td>
</tr>
<tr>
<td>8</td>
<td>1 0 0 0</td>
<td>4</td>
<td>3 2 1</td>
</tr>
<tr>
<td>9</td>
<td>1 0 0 1</td>
<td>4 1</td>
<td>3 2</td>
</tr>
<tr>
<td>10</td>
<td>1 0 1 0</td>
<td>4 2</td>
<td>3 1</td>
</tr>
<tr>
<td>11</td>
<td>1 0 1 1</td>
<td>4 2 1</td>
<td>3</td>
</tr>
<tr>
<td>12</td>
<td>1 1 0 0</td>
<td>4 3</td>
<td>2 1</td>
</tr>
<tr>
<td>13</td>
<td>1 1 0 1</td>
<td>4 3 1</td>
<td>2</td>
</tr>
<tr>
<td>14</td>
<td>1 1 1 0</td>
<td>4 3 2</td>
<td>1</td>
</tr>
<tr>
<td>15</td>
<td>1 1 1 1</td>
<td>4 3 2 1</td>
<td>empty</td>
</tr>
</tbody>
</table>
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

```java
private void enumerate(int k)
{
    if (k == N)
    {
        process(); return;
    }
    enumerate(k+1);
    a[k] = 1;
    enumerate(k+1);
    a[k] = 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>
Binary reflected gray code

The n-bit binary reflected Gray 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.
public static void moves(int n, boolean enter) {
    if (n == 0) return;
    moves(n-1, true);
    if (enter) System.out.println("enter " + n);
    else System.out.println("exit " + n);
    moves(n-1, false);
}
More Applications of Gray Codes

3-bit rotary encoder

8-bit rotary encoder

Towers of Hanoi

Chinese ring puzzle
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 enumeration

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

standard binary (from warmup)

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

Advantage (same as Beckett): only one item changes subsets
**Scheduling**

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

<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: NP-hard.

```java
class Scheduling {
    public double[] finish(int[] a) {
        double[] time = new double[2];
        time[0] = 0.0; time[1] = 0.0;
        for (int i = 0; i < N; i++)
            time[a[i]] += jobs[i];
        return time;
    }

    private double cost(int[] a) {
        double[] time = finish(a);
        return Math.abs(time[0] - time[1]);
    }
}
```

<table>
<thead>
<tr>
<th>i</th>
<th>a[]</th>
<th>time[0]</th>
<th>time[1]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0 1 1 0</td>
<td>1.41</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0 1 1 0</td>
<td>1.41</td>
<td>1.73</td>
</tr>
<tr>
<td>2</td>
<td>0 1 1 0</td>
<td>1.41</td>
<td>3.73</td>
</tr>
<tr>
<td>3</td>
<td>0 1 1 0</td>
<td>3.64</td>
<td>3.73</td>
</tr>
</tbody>
</table>

Cost: .09
public class Scheduler
{
    int N;       // Number of jobs.
    int[] a;     // Subset assignments.
    int[] b;     // Best assignment.
    double[] jobs; // Job lengths.
    public Scheduler(double[] jobs)
    {
        this.N = jobs.length;
        this.jobs = jobs;
        a = new int[N];
        b = new int[N];
        for (int i = 0; i < N; i++)
            a[i] = 0;
        for (int i = 0; i < N; i++)
            b[i] = a[i];
        enumerate(0);
    }
    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 result. */
    }
}
Large number of subsets leads to remarkably low cost
Scheduling: improvements

Many opportunities (details omitted)
• fix last job on machine 0 (quick factor-of-two improvement)
• backtrack when partial schedule cannot beat best known
  (check total against goal: half of total job times)

```java
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).
Backtracking summary

- **N-Queens**: permutations with backtracking
- **Sudoku**: counting with backtracking
- **Scheduling**: subsets with backtracking
permutations
backtracking
counting
subsets
paths in a graph
Hamilton Path

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

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

Visit every edge exactly once.
Knight's Tour

Knight's tour. Find a sequence of moves for a knight so that, starting from any 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
- Remove $v$ from current path.

How to implement?
Add cleanup to DFS (!!)
public class HamiltonPath
{
    private boolean[] marked;
    private int count;

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

    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;
    }

    also need code to count solutions (path length = V)
}

clean up

Easy exercise: Modify this code to find and print the longest path
The Longest Path

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

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.