## 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.

## 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></td>
<td>4</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</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 \( 2^n - 1 \).
- bit \( i \) represents item \( i \).
- if 0, in subset; if 1, not in subset.

Note: bitflicking simpler in assembly language.

```java
long N = 1 << n;
for (long i = 0; i < N; i++)
{
   for (int bit = 0; bit < n; bit++)
   {
      if (((i >> bit) & 1) == 1)
         System.out.print(bit + " ");
   }
   System.out.println();
}
```

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.

<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 0</td>
<td>4 3 2</td>
<td>enter 2</td>
</tr>
<tr>
<td>1 1 1 1</td>
<td>4 3 2 1</td>
<td>enter 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>

Beckett: Java implementation

```java
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);
}
```

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.

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

% java Beckett 4
enter 1
enter 2
exit 1
enter 3
enter 1
exit 2
exit 1
enter 4
enter 1
exit 2
exit 1
exit 3
exit 1
exit 2
exit 1
stage directions
for 3-actor play
moves(3, true)
reverse stage directions
for 3-actor play
moves(3, false)
More Applications of Gray Codes

- 3-bit rotary encoder
- 8-bit rotary encoder
- Chinese ring puzzle
- Towers of Hanoi

Scheduling

**Scheduling (set partitioning).** Given \( n \) jobs of varying length, divide among two machines to minimize the 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>1</td>
<td>1.41</td>
</tr>
<tr>
<td>2</td>
<td>1.73</td>
</tr>
<tr>
<td>3</td>
<td>2.00</td>
</tr>
<tr>
<td>4</td>
<td>2.23</td>
</tr>
</tbody>
</table>

**Remark.** NP-hard.

Scheduling (using Gray Code)

Beckett's stage directions

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>7.37</td>
<td>1.41</td>
<td>1.73</td>
<td>2.00</td>
<td>2.23</td>
</tr>
<tr>
<td>2</td>
<td>4.55</td>
<td>-1.41</td>
<td>1.73</td>
<td>2.00</td>
<td>2.23</td>
</tr>
<tr>
<td>3</td>
<td>1.09</td>
<td>-1.41</td>
<td>1.73</td>
<td>2.00</td>
<td>2.23</td>
</tr>
<tr>
<td>1</td>
<td>3.91</td>
<td>1.41</td>
<td>1.73</td>
<td>2.00</td>
<td>2.23</td>
</tr>
<tr>
<td>3</td>
<td>-0.09</td>
<td>1.41</td>
<td>1.73</td>
<td>-2.00</td>
<td>2.23</td>
</tr>
<tr>
<td>1</td>
<td>-2.91</td>
<td>-1.41</td>
<td>1.73</td>
<td>-2.00</td>
<td>2.23</td>
</tr>
<tr>
<td>2</td>
<td>0.55</td>
<td>-1.41</td>
<td>1.73</td>
<td>-2.00</td>
<td>2.23</td>
</tr>
<tr>
<td>1</td>
<td>3.38</td>
<td>1.41</td>
<td>1.73</td>
<td>-2.00</td>
<td>2.23</td>
</tr>
<tr>
<td>4</td>
<td>-1.08</td>
<td>1.41</td>
<td>1.73</td>
<td>-2.00</td>
<td>-2.23</td>
</tr>
<tr>
<td>1</td>
<td>-3.91</td>
<td>-1.41</td>
<td>1.73</td>
<td>-2.00</td>
<td>-2.23</td>
</tr>
<tr>
<td>2</td>
<td>-7.37</td>
<td>-1.41</td>
<td>1.73</td>
<td>-2.00</td>
<td>-2.23</td>
</tr>
<tr>
<td>3</td>
<td>-4.55</td>
<td>-1.41</td>
<td>1.73</td>
<td>-2.00</td>
<td>-2.23</td>
</tr>
<tr>
<td>3</td>
<td>-0.55</td>
<td>-1.41</td>
<td>1.73</td>
<td>-2.00</td>
<td>-2.23</td>
</tr>
<tr>
<td>1</td>
<td>-3.38</td>
<td>-1.41</td>
<td>1.73</td>
<td>2.00</td>
<td>-2.23</td>
</tr>
<tr>
<td>2</td>
<td>0.09</td>
<td>-1.41</td>
<td>1.73</td>
<td>2.00</td>
<td>-2.23</td>
</tr>
<tr>
<td>1</td>
<td>2.91</td>
<td>1.41</td>
<td>1.73</td>
<td>2.00</td>
<td>-2.23</td>
</tr>
</tbody>
</table>

Scheduling: Java implementation

```java
public static void moves(int n, double[] a, double[] b) {
    if (n == 0) return;
    moves(n-1, a, b);
    a[n] = -a[n];
    a[0] += 2*a[n];
    if (Math.abs(a[0]) < Math.abs(b[0])) {
        for (int i = 0; i < a.length; i++)
            b[i] = a[i];
        moves(n-1, a, b);
    }
}
```

```java
int[] a = { 7.37, 1.41, 1.73, 2.00, 2.23 };
int[] b = { 7.37, 1.41, 1.73, 2.00, 2.23 };
```
Exploiting Symmetry

- Exploit symmetry.
  - Half of schedules are redundant.
  - Fix job n on machine one $\Rightarrow$ twice as fast.

Space-Time Tradeoff

- Space-time tradeoff.
  - Enumerate all subsets of first n/2 jobs; sort by gap.
  - Enumerate all subsets of last n/2 jobs; for each subset, binary search to find for best matching subset among first n/2 jobs.

<table>
<thead>
<tr>
<th>gap (subset)</th>
<th>-5.14 (empty)</th>
<th>-2.32 (1)</th>
<th>-1.68 (2)</th>
<th>-1.14 (3)</th>
<th>1.14 (1 2)</th>
<th>1.68 (1 3)</th>
<th>2.32 (2 3)</th>
<th>5.14 (1 2 3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>best match</td>
<td>-5.58 (empty)</td>
<td>-1.12 (4)</td>
<td>0.42 (5)</td>
<td>-4.08 (4)</td>
<td>4.48 (4 5)</td>
<td>-0.42 (4 6)</td>
<td>1.12 (1 6)</td>
<td>5.58 (1 5 6)</td>
</tr>
<tr>
<td>gap (subset)</td>
<td>-5.24 (1 2 3)</td>
<td>0.00 (2 4)</td>
<td>-0.72 (1 2 4)</td>
<td>0.26 (1 2 3 4)</td>
<td>-0.26 (2 4 6)</td>
<td>0.72 (1 5 6)</td>
<td>0.00 (1 2 3)</td>
<td>-0.44 (1 2 4)</td>
</tr>
</tbody>
</table>

Reduces running time from $2^n$ to $2^{n/2}$ log n by consuming $2^{n/2}$ memory.

8-Queens Problem

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

<table>
<thead>
<tr>
<th>Job</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.41</td>
</tr>
<tr>
<td>2</td>
<td>1.73</td>
</tr>
<tr>
<td>3</td>
<td>2.00</td>
</tr>
<tr>
<td>4</td>
<td>2.23</td>
</tr>
<tr>
<td>5</td>
<td>3.00</td>
</tr>
<tr>
<td>6</td>
<td>0.35</td>
</tr>
</tbody>
</table>

Representation. Can represent solution as a permutation: $q[i] =$ column of queen in row i.

```java
int[] q = { 5, 7, 1, 3, 8, 6, 4, 2 };
```

Queens 1 and 3 can attack each other if $|q[1] + 1| = |q[3] + 3|$.
Enumerating Permutations

**Permutations.** Given n items, enumerate all n! permutations.

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

3 followed by any permutation of 1 2 4
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

Enumerating all Permutations

To enumerate all permutations of a set of n elements:
- For each element aᵢ:
  - put aᵢ first, then append
    - a permutation of the remaining elements (a₀, ..., aᵢ₋₁, aᵢ₊₁, ..., aₙ₋₁)

4-Queens Search Tree
N Queens: Backtracking solution

**Backtracking.** Iterate through elements of search space.
- for each row, there are N possible choices.
- make one choice and recur.
- if the choice does not work, backtrack to previous choice, and make next available choice.

Backtracking amounts to **pruning** the search space.

For N queens: if you find a diagonal conflict, no need to continue

**Improvements.**
- try to make an “intelligent” choice
- try to reduce cost of choosing/backtracking

N-Queens: Backtracking solution

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

int N = 4;
int[] q = { 1, 2, 3, 4 };
enumerate(q, N);

4-Queens Search Tree (pruned)

subsets
permutations
counting
paths in a lattice
paths in a graph
Counting: Java Implementation

Enumerate all $M$-digit base-$R$ numbers.

```
private static void count(int[] number, int digit)
{
    if (digit == M)
    {  show(number); return;  }
    for (int n = 0; n < R; n++)
    {  count(number, digit + 1);  }
    number[digit] = 0;
}
```

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

```
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  
```

Remark. Natural generalization is NP-hard.

Sudoku

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.

```
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 6 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  
```

Remark. Natural generalization is NP-hard.
**Sudoku: Backtracking solution**

**Backtracking.** 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, backtrack to previous choice, and make next available choice.

Backtracking amounts to **pruning** the search space.

For Sudoku:
if you find a conflict in row, column or box, no need to continue

**Improvements.**
- try to make an "intelligent" choice
- try to reduce cost of choosing/backtracking

**Sudoku: Java implementation**

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

```java
int[] board = { 7, 0, 8, 0, 0, 0, 3, ... };
solve(board, 0);
```

**Subsets, permutations, counting, paths in a lattice, paths in a graph**

**All Paths on a Grid**

**All paths.** Enumerate all simple paths on a grid of adjacent sites.

Application. Self-avoiding lattice walk to model polymer chains.
no atoms can occupy same position at same time
**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 A X X X
X C A C K
X K A X X
X T X X X
XXX XXX
```

**Boggle: Java Implementation**

```java
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;
}
```

**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
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 (!)

Hamilton Path: Java implementation

```java
public class HamiltonPath {
    private boolean[] marked;
    private int[] pred;

    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()) System.out.println("Path found!");
        for (int w : G.adj(v))
            if (!marked[w]) {
                pred[w] = v;
                dfs(G, w, depth + 1);
            }
        marked[v] = false;
    }
}
```

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.

Gary, 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.

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.

![legal knight moves](image1)

![a knight's tour](image2)

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

Hamilton's Tour

**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 (!)
Course evaluation info

Course: COS 226
Term: Spring '07.
Lecturer: Robert Sedgewick
Precept instructor: Jimin Song (01)
or David Walker (01A or 02)
or Mohammad Ghidary (03)

Please use a #2 pencil (provided).

Final exam info

Saturday, May 19 at 7:30 PM.
Review sessions:
- Prepare and e-mail questions in advance.
- All questions answered.
- No questions? No session.
- Any student may attend any or all sessions.

mohammad: 1PM Wed 16 May
dave: 1PM Thu 17 May
jimin: 1PM Fri 18 May