http://algs4.cs.princeton.edu

COMBINATORIAL SEARCH

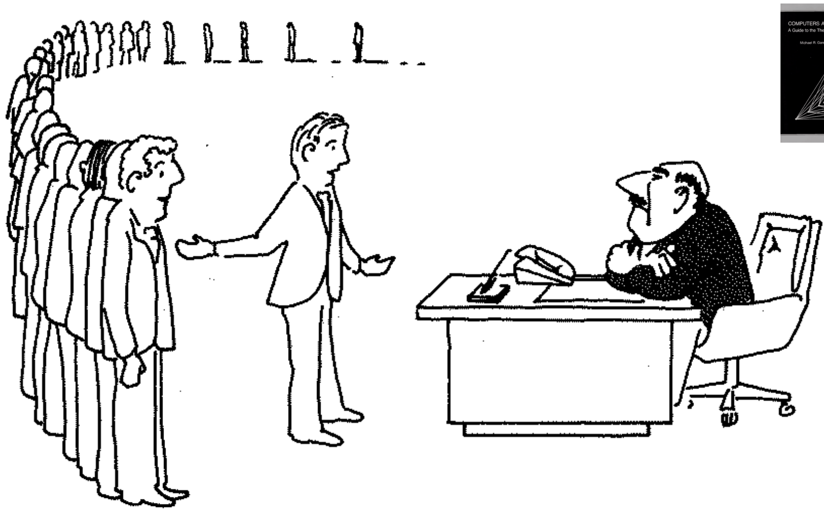
- *introduction*
- *permutations*
- *backtracking*
- *counting*
- *subsets*
- *paths in a graph*

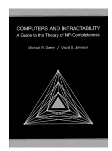
http://algs4.cs.princeton.edu

COMBINATORIAL SEARCH

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

Implications of NP-completeness





"I can't find an efficient algorithm, but neither can all these famous people."

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 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.

```
// 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;
   }
}
```

all programs in this lecture are variations on this theme

```
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
```


Algorithms

Robert Sedgewick | Kevin Wayne

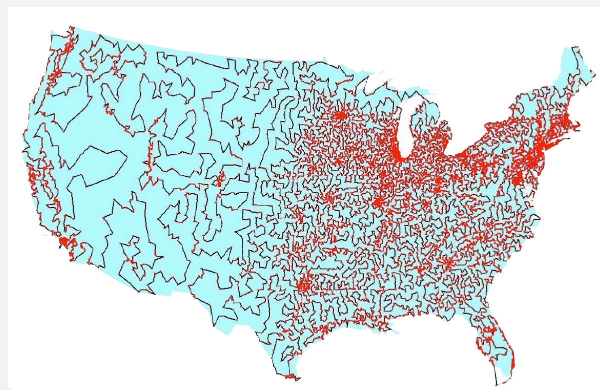
http://algs4.cs.princeton.edu

Combinatorial Search

Traveling salesperson problem

Euclidean TSP. Given N points in the plane, find the shortest tour.

Proposition. Euclidean TSP is NP-hard.

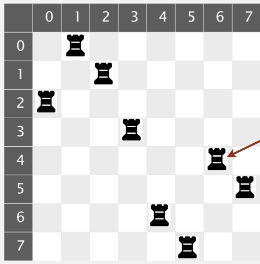


13509 cities in the USA and an optimal tour

Brute force. Design an algorithm that checks all tours.

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?



a[4] = 6 means the rook from row 4 is in column 6

`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 N integers 0 to $N-1$.

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)

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 permutation back to original

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
0 1 2 3	1 0 2 3	2 1 0 3	3 1 2 0
0 1 3 2	1 0 3 2	2 1 3 0	3 1 0 2
0 2 1 3	1 2 0 3	2 0 1 3	3 2 1 0
0 2 3 1	1 2 3 0	2 0 3 1	3 2 0 1
0 3 2 1	1 3 2 0	2 3 0 1	3 0 2 1
0 3 1 2	1 3 0 2	2 3 1 0	3 0 1 2

a[0] a[N-1]

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)

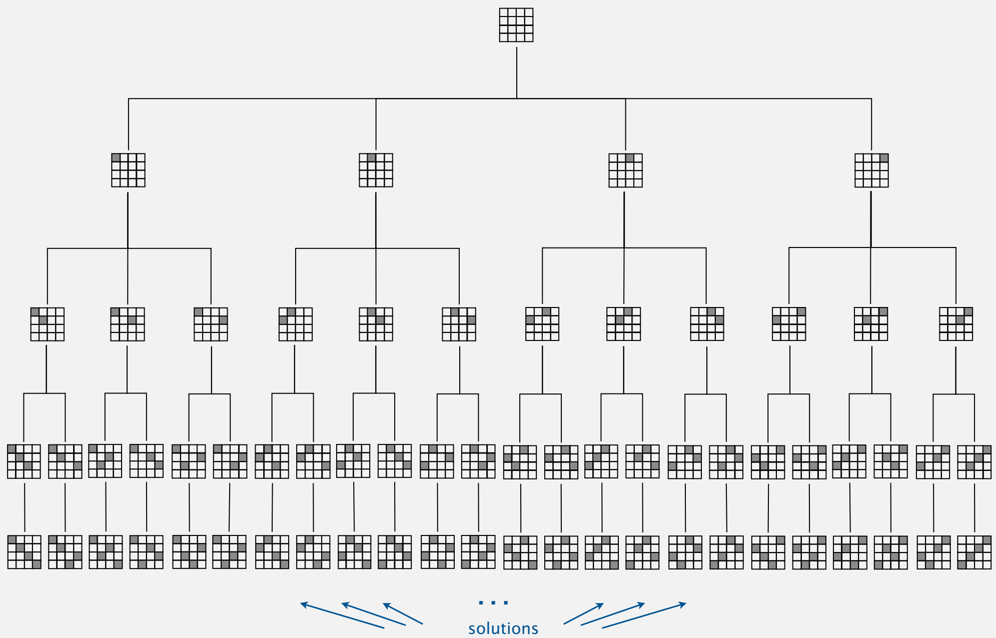
```
// place N-k rooks in a[k] to a[N-1]
private void enumerate(int k)
{
   if (k == N)
   {  process(); return;  }

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

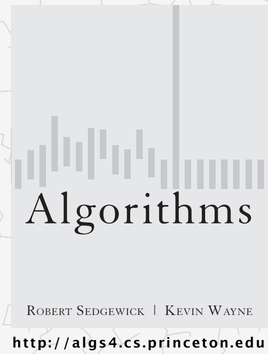

Enumerating permutations

```
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;            ← initial permutation
      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);
   }
}
```

```
% 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

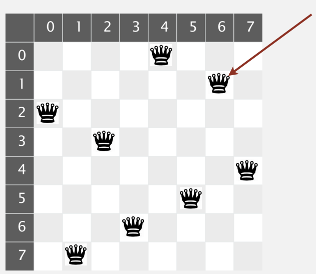


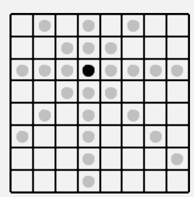


Combinatorial Search

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?



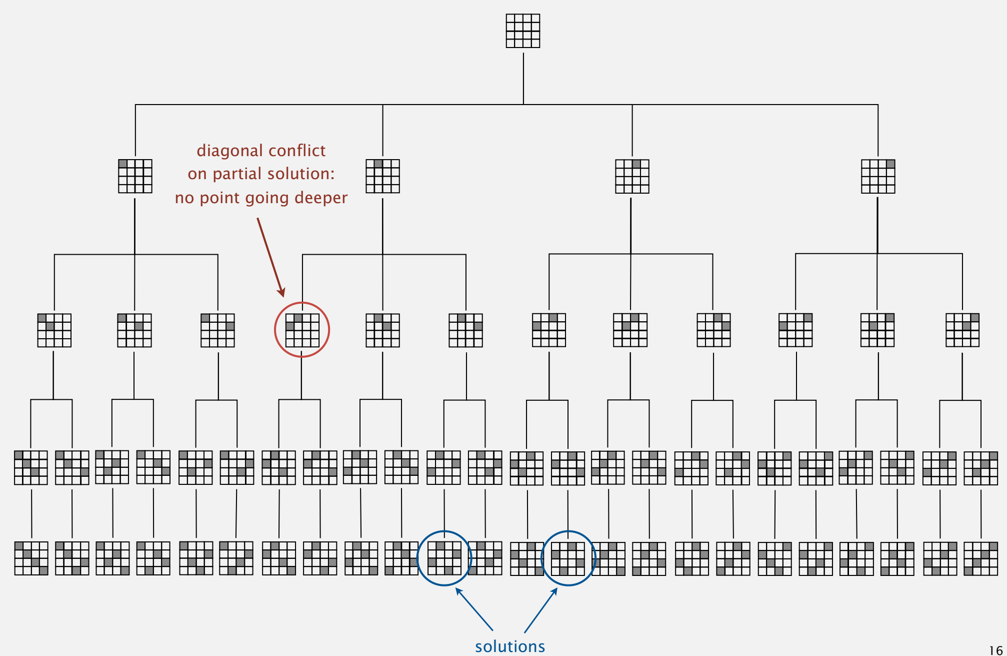


```
int[] a = { 2, 7, 3, 6, 0, 5, 1, 4 };
```

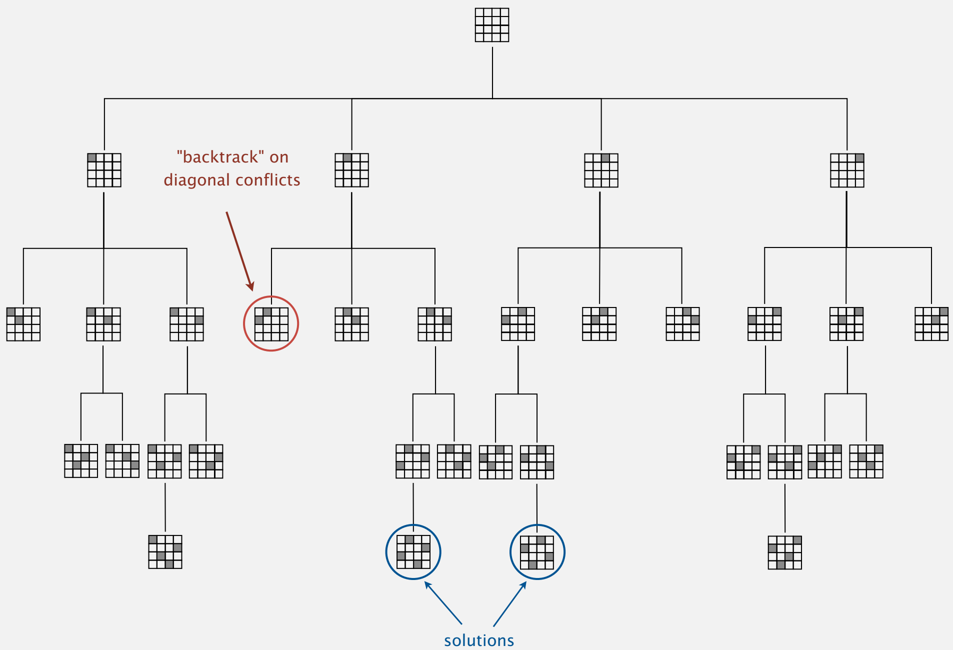
Representation. No 2 queens in the same row or column $\Rightarrow$ permutation.
Additional constraint. No diagonal attack is possible.

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)



Backtracking

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.

Applications. Puzzles, combinatorial optimization, parsing, ...

N-queens problem: backtracking solution

```
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);
   }
}
```

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 !	time (sec)
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	1.1
13	73,712	6,227,020,800	5.4
14	365,596	87,178,291,200	29
15	2,279,184	1,307,674,368,000	210
16	14,772,512	20,922,789,888,000	1352

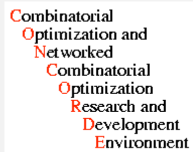
Conjecture. $Q(N) \sim N! \;/\; c^N$, where c is about 2.54.

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

Some backtracking success stories

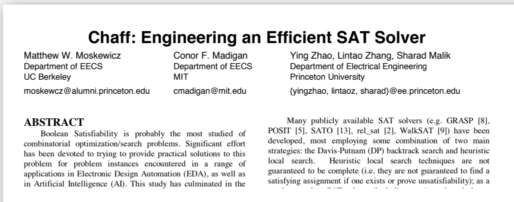
TSP. Concorde solves real-world TSP instances with ~ 85K points.

- Branch-and-cut.
- Linear programming.
- ...



SAT. Chaff solves real-world instances with ~ 10K variable.

- Davis-Putnam backtracking.
- Boolean constraint propagation.
- ...



Chaff: Engineering an Efficient SAT Solver

Matthew W. Moskewicz, Department of EECS, UC Berkeley, moskewcz@alumni.princeton.edu
Conor F. Madigan, Department of EECS, MIT, cmadigan@mit.edu
Ying Zhao, Lintao Zhang, Sharad Malik, Department of Electrical Engineering, Princeton University, {yingzhao, lintaoz, sharad}@ee.princeton.edu

ABSTRACT

Boolean Satisfiability is probably the most studied of combinatorial optimization/search problems. Significant effort has been devoted to trying to provide practical solutions to this problem for problem instances encountered in a range of applications in Electronic Design Automation (EDA), as well as in Artificial Intelligence (AI). This study has culminated in the

Many publicly available SAT solvers (e.g. GRASP [8], POSIT [5], SATO [13], rel_sat [2], WalkSAT [9]) have been developed, most employing some combination of two main strategies: the Davis-Putnam (DP) backtrack search and heuristic local search. Heuristic local search techniques are not guaranteed to be complete (i.e. they are not guaranteed to find a satisfying assignment if one exists or prove unsatisfiability); as a

http://algs4.cs.princeton.edu

Combinatorial Search

- introduction
- 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
```

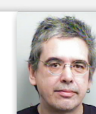
a[0] a[N-1]

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		

" Sudoku is a denial of service attack on human intellect. "
— Ben Laurie (founding director of Apache Software Foundation)



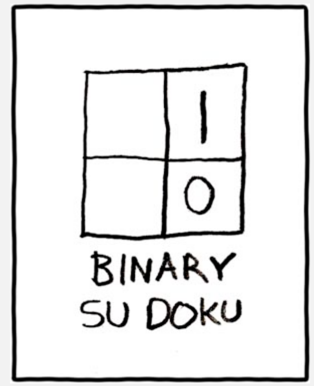
Sudoku

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

Sudoku is (probably) intractable

Remark. Natural generalization of Sudoku is NP-complete.



http://xkcd.com/74

Sudoku: brute-force solution

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

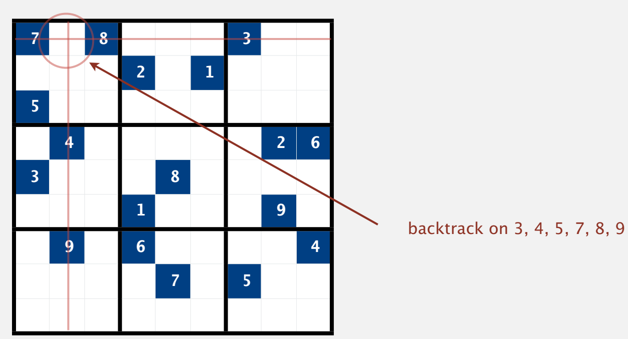
7		8				3		
			2		1			
5								
	4						2	6
3				8				
			1				9	
	9		6					4
				7		5		

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 (!canBacktrack(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
```


http://algs4.cs.princeton.edu

Combinatorial Search

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.

i	binary	subset
0	0 0 0 0	empty
1	0 0 0 1	0
2	0 0 1 0	1
3	0 0 1 1	1 0
4	0 1 0 0	2
5	0 1 0 1	2 0
6	0 1 1 0	2 1
7	0 1 1 1	2 1 0
8	1 0 0 0	3
9	1 0 0 1	3 0
10	1 0 1 0	3 1
11	1 0 1 1	3 1 0
12	1 1 0 0	3 2
13	1 1 0 1	3 2 0
14	1 1 1 0	3 2 1
15	1 1 1 1	3 2 1 0

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.

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

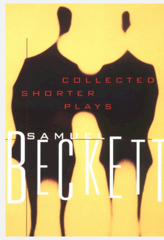
Digression: Samuel Beckett play

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

binary	subset	move
0 0 0 0	empty	-
0 0 0 1	0	enter 0
0 0 1 1	1 0	enter 1
0 0 1 0	1	exit 1
0 1 1 0	2 1	enter 2
0 1 1 1	2 1 0	enter 0
0 1 0 1	2 0	exit 1
0 1 0 0	2	exit 0
1 1 0 0	3 2	enter 3
1 1 0 1	3 2 0	enter 0
1 1 1 1	3 2 1 0	enter 1
1 1 1 0	3 2 1	exit 0
1 0 1 0	3 1	exit 2
1 0 1 1	3 1 0	enter 0
1 0 0 1	3 0	exit 1
1 0 0 0	3	exit 0

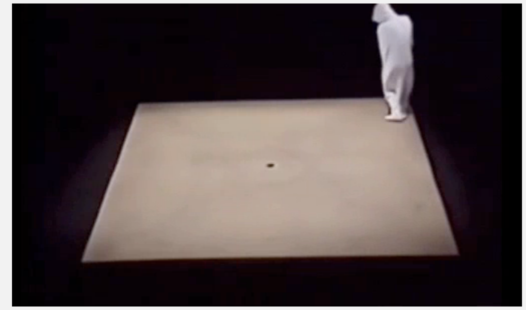
binary reflected Gray code

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.

2-bit (1-bit code; 1-bit code (reversed))

0	0
0	1
1	1
1	0

3-bit (2-bit code; 2-bit code (reversed))

0	0 0	
0	0 1	
0	1 1	
0	1 0	
1	1 0	
1	1 1	
1	0 1	
1	0 0	

4-bit (3-bit code)

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

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);
}
```

a[0]		a[N-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

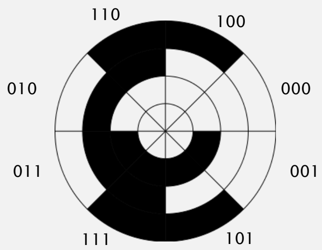
standard binary counter (from warmup)

```
// 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;
}
```

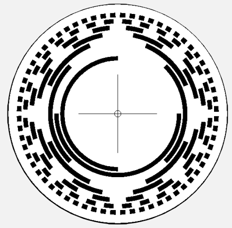
a[0]		a[N-1]
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

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

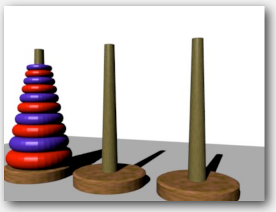
More applications of Gray codes



3-bit rotary encoder

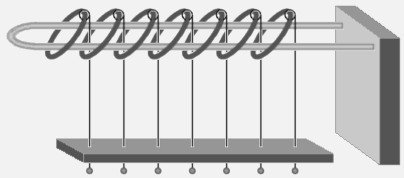


8-bit rotary encoder



Towers of Hanoi

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



Chinese ring puzzle (Baguenaudier)

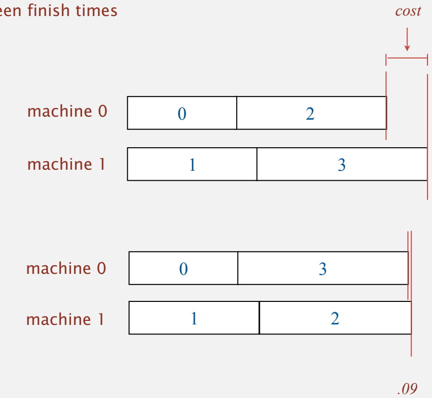
(move ith ring from right when bit i changes in Gray code)

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

job	length
0	1.41
1	1.73
2	2.00
3	2.23



Remark. This scheduling problem is NP-complete.

Scheduling: improvements

Brute force. Enumerate 2^N subsets; compute makespan; return best.

Many opportunities to improve.

- Fix first job to be on machine 0. — factor of 2 speedup
- Maintain difference in finish times. (and avoid recomputing cost from scratch) — factor of N speedup (using Gray code order)
- Backtrack when partial schedule cannot beat best known. — huge opportunities for improvement on typical inputs
- Preprocess all 2^k subsets of last k jobs; cache results in memory. — reduces time to 2^{N-k} at cost of 2^k memory

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

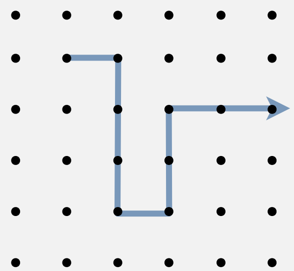

http://algs4.cs.princeton.edu

Combinatorial Search

- *introduction*
- *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.

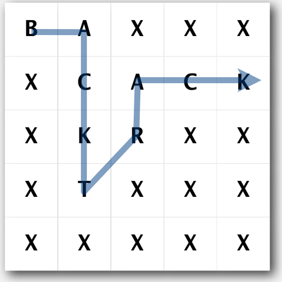


no two atoms can occupy same position at same time

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

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).



Backtracking. 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

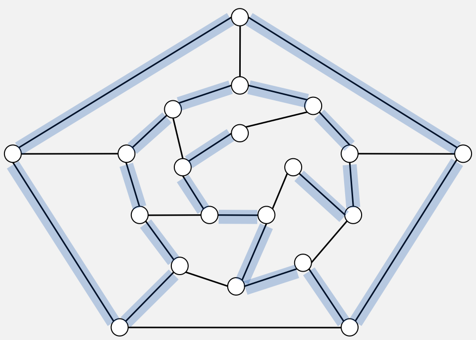
add to set of found words

try all possibilities

clean up

Hamilton path

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

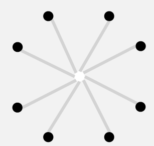


visit every edge exactly once

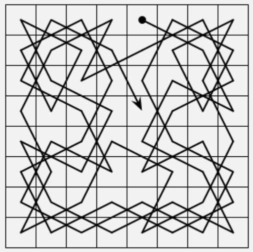
Remark. Euler path easy, but Hamilton path is NP-complete.

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. Depth-first search + cleanup (!)

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 → `if (depth == G.V()) count++;`

length of current path (depth of recursion) → `int depth`

backtrack if w is already part of path → `if (!marked[w]) dfs(G, w, depth+1);`

clean up → `marked[v] = false;`

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

The world's longest path (Sendero de Chile): 9,700 km.
(originally scheduled for completion in 2010; now delayed until 2038)

That's all, folks: keep searching!

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.

Written by Dan Barrett in 1988 while a student
at Johns Hopkins during a difficult algorithms take-home final