Complete Search

Ruben Becker June 4, 2026

RAVEN – Research on Algorithms Venice

Complete Search

Complete Search — Overview

What is it?

- also known as **brute force** or **(recursive) backtracking**
- traverse the *entire* (or part of the) search space to find the solution
- may **prune** parts of the space that cannot contain a valid solution
- always returns the correct answer

Complete Search — Overview

What is it?

- also known as **brute force** or **(recursive) backtracking**
- traverse the *entire* (or part of the) search space to find the solution
- may **prune** parts of the space that cannot contain a valid solution
- always returns the correct answer

When to use it?

- if no better algorithm exists (e.g., enumerate all $N!$ permutations)
- better algorithms exist, but input size is small enough
- consider it *first* — easy to code, easy to debug (KISS)
- small instances can reveal **patterns** leading to a faster algorithm

Two Flavours

Iterative (§3.2.1)

- nested loops, permutations, bitmask enumeration
- *filters:* generate all candidates, keep valid ones
- easier to code
- harder to prune deeply

Recursive / Backtracking (§3.2.2)

- build solution incrementally; prune invalid partial sol. *early*
- *generators:* never build an invalid partial solution
- usually faster in practice

Complete Search

1. Two Nested Loops — UVa 00725 *Division*

Problem: find all pairs of 5-digit numbers using digits 0–9 exactly once such that `abcde`/`fghij` $= N$.

1. Two Nested Loops — UVa 00725 *Division*

Problem: find all pairs of 5-digit numbers using digits 0–9 exactly once such that abcde/fghij $= N$.

Approach

- iterate fghij from 1234 to $98765/N$ — at most $\approx 50\,000$ values
- compute abcde = fghij * N; check 10 digits distinct via bitmask

```
for (int fghij = 1234; fghij <= 98765/N; ++fghij) {
    int abcde = fghij * N;
    int tmp, used = (fghij < 10000); // flag leading 0
    tmp = abcde; while (tmp) { used |= 1<<(tmp%10); tmp /= 10; }
    tmp = fghij; while (tmp) { used |= 1<<(tmp%10); tmp /= 10; }
    if (used == (1<<10)-1)          // all 10 digits used?
        printf("%05d / %05d = %d\n", abcde, fghij, N);
}
```

2. Many Nested Loops — UVa 00441 *Lotto*

Problem: given $6 < k < 13$ sorted integers, enumerate all size-6 subsets in sorted order.

2. Many Nested Loops — UVa 00441 *Lotto*

Problem: given $6 < k < 13$ sorted integers, enumerate all size-6 subsets in sorted order.
Six nested loops — at most $\binom{12}{6} = 924$ output lines.

```
for (int i = 0; i < k; ++i) scanf("%d", &S[i]);
for (int a = 0 ; a < k-5; ++a)
  for (int b = a+1; b < k-4; ++b)
    for (int c = b+1; c < k-3; ++c)
      for (int d = c+1; d < k-2; ++d)
        for (int e = d+1; e < k-1; ++e)
          for (int f = e+1; f < k  ; ++f)
            printf("%d␣%d␣%d␣%d␣%d␣%d\n", S[a],S[b],S[c],S[d],S[e],S
                [f]);
```

2. Many Nested Loops — UVa 00441 *Lotto*

Problem: given $6 < k < 13$ sorted integers, enumerate all size-6 subsets in sorted order.
Six nested loops — at most $\binom{12}{6} = 924$ output lines.

```
for (int i = 0; i < k; ++i) scanf("%d", &S[i]);
for (int a = 0 ; a < k-5; ++a)
  for (int b = a+1; b < k-4; ++b)
    for (int c = b+1; c < k-3; ++c)
      for (int d = c+1; d < k-2; ++d)
        for (int e = d+1; e < k-1; ++e)
          for (int f = e+1; f < k  ; ++f)
            printf("%d␣%d␣%d␣%d␣%d␣%d\n", S[a],S[b],S[c],S[d],S[e],S
                [f]);
```

Takeaway

Fixed-size enumeration with small bounds $\Rightarrow$ hard-coded nested loops work fine.

3. Loops + Pruning — UVa 11565 *Simple Equations*

Problem: find distinct integers x, y, z with $x + y + z = A$, $x \cdot y \cdot z = B$, $x^2 + y^2 + z^2 = C$, where $1 \leq A, B, C \leq 10000$

3. Loops + Pruning — UVa 11565 *Simple Equations*

Problem: find distinct integers x, y, z with $x + y + z = A$, $x \cdot y \cdot z = B$, $x^2 + y^2 + z^2 = C$, where $1 \leq A, B, C \leq 10000$

Bounding the search

- From $x^2 + y^2 + z^2 = C \leq 10000$: each variable in $[-100, 100]$
- Better: assuming x smallest, $x^3 \leq B$ so $x \in [-22, 22]$

```
bool sol = false; int x, y, z;
for (x = -22; (x <= 22) && !sol; ++x)
  if (x*x <= C)
    for (y = -100; (y <= 100) && !sol; ++y)
      if ((y != x) && (x*x + y*y <= C))
        for (z = -100; (z <= 100) && !sol; ++z)
          if ((z != x) && (z != y) &&
              (x+y+z == A) && (x*y*z == B) &&
              (x*x + y*y + z*z == C)) {
              printf("%d␣%d␣%d\n", x, y, z); sol = true;
          }
```

4. Permutations — UVa 11742 *Social Constraints*

Problem: $n \leq 8$ moviegoers, $m \leq 20$ seating constraints of the form a and b must be at least c seats apart. Count valid arrangements.

4. Permutations — UVa 11742 *Social Constraints*

Problem: $n \leq 8$ moviegoers, $m \leq 20$ seating constraints of the form a and b must be at least c seats apart. Count valid arrangements.

Approach

- just try all $n!$ permutations; check all m constraints for each
- $8! \times 20 = 806\,400$ operations — perfectly fine

C++:

```
#include <bits/stdc++.h> //
    next_permutation is inside C
    ++ STL <algorithm>
int p[8] = {0,1,2,3,4,5,6,7};
do {
    // test permutation p in O(m)
} while (next_permutation(p, p+n)
    );
```

Python:

```
import itertools
for p in itertools.permutations(
    range(n)):
    # test permutation p
```

5. Bitmask Subsets — UVa 12455 *Bars*

Problem: given $n \leq 20$ integers, does any subset sum to X?

5. Bitmask Subsets — UVa 12455 *Bars*

Problem: given $n \leq 20$ integers, does any subset sum to X?

- enumerate all 2^n subsets using bitmask $i \in [0, 2^n)$
- bit $j = 1 \Rightarrow$ element j included

```
for (int i = 0; i < (1<<n); ++i) {
    int sum = 0;
    for (int j = 0; j < n; ++j)
        if (i & (1<<j))
            sum += l[j];
    if (sum == X) break;
}
```

5. Bitmask Subsets — UVa 12455 *Bars*

Problem: given $n \leq 20$ integers, does any subset sum to X?

- enumerate all 2^n subsets using bitmask $i \in [0, 2^n)$
- bit $j = 1 \Rightarrow$ element j included

```
for (int i = 0; i < (1<<n); ++i) {
    int sum = 0;
    for (int j = 0; j < n; ++j)
        if (i & (1<<j))
            sum += l[j];
    if (sum == X) break;
}
```

Note

- largest case: $20 \times 2^{20} \approx 21\,\text{M}$ — viable
- this is the NP-hard *Subset-Sum* problem

Complete Search

General Backtracking Approach

backtracking algorithm builds solution **one decision at a time**:

1. make one choice consistent with all *past* decisions
2. recurse — let the algorithm handle the rest – **trust in recursion**!
3. if stuck (no valid choice), **backtrack** to the previous decision

General Backtracking Approach

backtracking algorithm builds solution **one decision at a time**:

1. make one choice consistent with all *past* decisions
2. recurse — let the algorithm handle the rest – **trust in recursion**!
3. if stuck (no valid choice), **backtrack** to the previous decision

What each recursive call needs

- **remaining input** not yet processed
- **compact summary** of past decisions
  (only what future choices depend on)

8-Queens Problem — UVa 00750

Problem: place 8 queens on a 8×8 board so no two attack each other

8-Queens Problem — UVa 00750

Problem: place 8 queens on a 8×8 board so no two attack each other

Four levels of refinement

- **Naïve** $\binom{64}{8} \approx 4\,\mathrm{B}$ — all 8-cell combos – way too slow

8-Queens Problem — UVa 00750

Problem: place 8 queens on a 8×8 board so no two attack each other

Four levels of refinement

- **Naïve** $\binom{64}{8} \approx 4\,\mathrm{B}$ — all 8-cell combos – way too slow
- **Still naïve** $8^8 \approx 17\,\mathrm{M}$ — one queen per column – borderline

8-Queens Problem — UVa 00750

Problem: place 8 queens on a 8×8 board so no two attack each other

Four levels of refinement

- **Naïve** $\binom{64}{8} \approx 4\,\mathrm{B}$ — all 8-cell combos – way too slow
- **Still naïve** $8^8 \approx 17\,\mathrm{M}$ — one queen per column – borderline
- **Faster** $8! \approx 40\,\mathrm{K}$ — one queen per column *and* row – probably ok

8-Queens Problem — UVa 00750

Problem: place 8 queens on a 8×8 board so no two attack each other

Four levels of refinement

- **Naïve** $\binom{64}{8} \approx 4\,\mathrm{B}$ — all 8-cell combos – way too slow
- **Still naïve** $8^8 \approx 17\,\mathrm{M}$ — one queen per column – borderline
- **Faster** $8! \approx 40\,\mathrm{K}$ — one queen per column *and* row – probably ok
- **Still Better** $\ll 8!$ — add diagonal constraints, prune *during* search

8-Queens Problem — UVa 00750

Problem: place 8 queens on a 8×8 board so no two attack each other

Four levels of refinement

- **Naïve** $\binom{64}{8} \approx 4\,\mathrm{B}$ — all 8-cell combos – way too slow
- **Still naïve** $8^8 \approx 17\,\mathrm{M}$ — one queen per column – borderline
- **Faster** $8! \approx 40\,\mathrm{K}$ — one queen per column *and* row – probably ok
- **Still Better** $\ll 8!$ — add diagonal constraints, prune *during* search

8-Queens Problem — UVa 00750

Problem: place 8 queens on a 8×8 board so no two attack each other

Four levels of refinement

- **Naïve** $\binom{64}{8} \approx 4\,\mathrm{B}$ — all 8-cell combos – way too slow
- **Still naïve** $8^8 \approx 17\,\mathrm{M}$ — one queen per column – borderline
- **Faster** $8! \approx 40\,\mathrm{K}$ — one queen per column *and* row – probably ok
- **Still Better** $\ll 8!$ — add diagonal constraints, prune *during* search

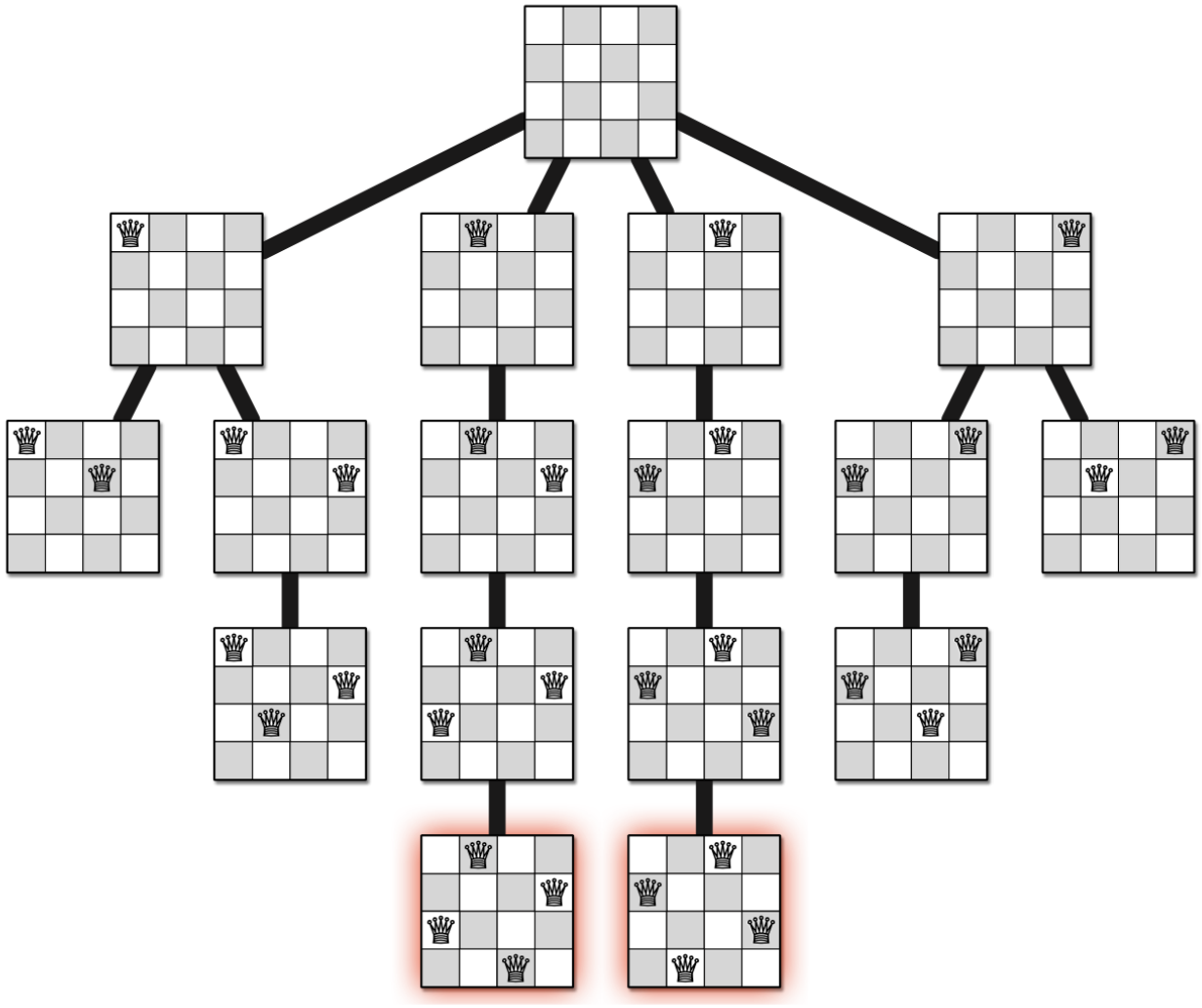
Backtracking approach: place queens one row at a time – at each row, try all columns; skip positions attacked by earlier queens

Gauss & Laquière's Algorithm

- $Q[i]$ = column of the queen in row i
- call `PlaceQueens(Q,1)` for all solutions

```
1  PlaceQueens(Q[1..n], r):
2    if r = n+1
3        print Q[1..n]                  // complete solution found!
4    else
5        for j <- 1 to n                // try each column in row r
6            legal <- true
7            for i <- 1 to r-1        // check all previous queens
8                if (Q[i]=j) or (Q[i]=j+r-i) or (Q[i]=j-r+i)
9                    legal <- false
10           if legal
11               Q[r] <- j
12               PlaceQueens(Q[1..n], r+1)     // recurse!
```

Recursion Tree for 4 Queens



Complete Search

Summary

Complete Search (§3.2)

- always correct — the only risk is `TLE`, not `WA`
- analyse complexity *before* coding

Summary

Complete Search (§3.2)

- always correct — the only risk is `TLE`, not `WA`
- analyse complexity *before* coding

Iterative CS (§3.2.1)

- nested loops, `next_permutation`, bitmask subsets
- prune with early `break`/ `continue`, short-circuit conditions

Summary

Complete Search (§3.2)

- always correct — the only risk is `TLE`, not `WA`
- analyse complexity *before* coding

Iterative CS (§3.2.1)

- nested loops, `next_permutation`, bitmask subsets
- prune with early `break`/ `continue`, short-circuit conditions

Recursive CS (§3.2.2)

- backtracking = DFS on implicit search tree
- 8-Queens: $4\,\text{B} \rightarrow 17\,\text{M} \rightarrow 40\,\text{K} \rightarrow\!\ll 40\,\text{K}$

Summary

Complete Search (§3.2)

- always correct — the only risk is `TLE`, not `WA`
- analyse complexity *before* coding

Iterative CS (§3.2.1)

- nested loops, `next_permutation`, bitmask subsets
- prune with early `break`/ `continue`, short-circuit conditions

Recursive CS (§3.2.2)

- backtracking = DFS on implicit search tree
- 8-Queens: $4\,\mathrm{B} \to 17\,\mathrm{M} \to 40\,\mathrm{K} \to\!\ll 40\,\mathrm{K}$

Practical Tips

- **Prune early** — the sooner detect infeasibility, the larger cut subtree
- **Bound tightly** — use problem constraints to narrow loop ranges

Complete Search

Your Task

- `https://open.kattis.com/contests/fnstj4/problems/natjecanje`

References

- Sections 3.2.1 & 3.2.2 in Halim, Halim, and Effendy. Competitive Programming 4. Book 1.
- `https://courses.grainger.illinois.edu/CS473/fa2025/notes/02-backtracking-new.pdf`