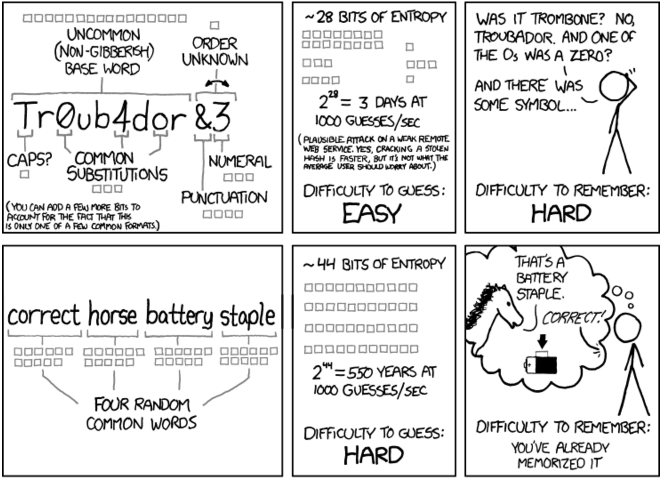
Algorithms for Programming Contests - Week 06

Prof. Dr. Javier Esparza
Pranav Ashok, A. R. Balasubramanian,
Tobias Meggendorfer, Philipp Meyer,
Mikhail Raskin,
conpra@in.tum.de

26. Mai 2020

Relevant XKCD



THROUGH 20 YEARS OF EFFORT, WE'VE SUCCESSFULLY TRAINED EVERYONE TO USE PASSWORDS THAT ARE HARD FOR HUMANS TO REMEMBER, BUT EASY FOR COMPUTERS TO GUESS.

Brute Force

Definition (Brute Force)

Systematically enumerate **all** solution candidates and test whether each candidate satisfies solution requirements.

a.k.a **Exhaustive Search** or **Generate and Test**.

Brute Force: Example

Task:

Given a combination lock of 4 decimal digits, find the right key for the lock.

Brute Force Solution:

Test all combinations from 0000 to 9999 until the right one is found.

Pros and Cons

Pros

- Simple
- Sound and complete - will find (optimum) solution if there exists one
- Used in safety critical applications because of its simplicity
- Serves as a benchmark for faster/more error-prone methods

Cons

- Inefficient
- Not feasible for large input sizes (combinatorial explosion)

Is Brute Force Feasible?

Estimate the number of operations used to see whether an implementation makes sense.

```
int main() {
    int i;
    for (i = 0; i < 1000000000; i++) { nop(); } // 10^9
    return 0;
}
```

How long would the above program take?

Combinatorial Explosion

Brute-forcing passwords (without any fancy business!)

Allowed	**Length**	**Search Space**	**Time**
0-9	5	10^5	$<$1s
	10	10^{10}	20s
	15	10^{15}	23 days

Combinatorial Explosion

Brute-forcing passwords (without any fancy business!)

Allowed	Length	Search Space	Time
0-9	5	10^5	$<$1s
	10	10^{10}	20s
	15	10^{15}	23 days
a-zA-Z	5	52^5	$\sim$1s
	10	52^{10}	9.1 years
	15	52^{15}	3.5 billion years

Combinatorial Explosion

Brute-forcing passwords (without any fancy business!)

Allowed	Length	Search Space	Time
0-9	5	10^5	$<$1s
	10	10^{10}	20s
	15	10^{15}	23 days
a-zA-Z	5	52^5	$\sim$1s
	10	52^{10}	9.1 years
	15	52^{15}	3.5 billion years
Printable `ASCII`	5	95^5	15s
	10	95^{10}	3795 years
	15	95^{15}	2100 x **age of universe**

Combinatorial Explosion

Brute-forcing passwords (without any fancy business!)

Allowed	Length	Search Space	Time
0-9	5	10^5	$<$1s
	10	10^{10}	20s
	15	10^{15}	23 days
a-zA-Z	5	52^5	$\sim$1s
	10	52^{10}	9.1 years
	15	52^{15}	3.5 billion years
Printable `ASCII`	5	95^5	15s
	10	95^{10}	3795 years
	15	95^{15}	2100 x **age of universe**

You can see that it quickly grows out of hand!

Workaround: Search Order Matters!

- Reorder state space: when only one solution is needed, **start with the most promising ones!**
  e.g. it makes sense for a password cracker to search for passwords like 1234 or *password* first.

Workaround: Exploit the constraints

- Problem might be complicated and intractable in general
- But easy to brute-force because the input is small
  eg. task to search through strings of length 5
- Exploit domain knowledge when possible

Example: Finding Divisors

Find divisors of n

Brute force approach: Enumerate all numbers $i \in [1, n]$ and check if i divides n.

Example: Finding Divisors

Find divisors of n

Brute force approach: Enumerate all numbers $i \in [1, n]$ and check if i divides n.

But we can use our domain knowledge and search only upto $\lceil \sqrt{n} \rceil$

- 32-bit number has up to 10 decimal digits.
  $\implies$ square root has about 5.
  $\implies$ 10^5 tests instead of 10^{10} tests

Example: Finding Divisors

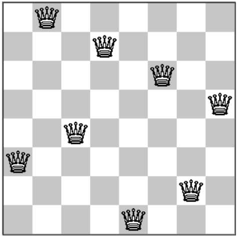
Find divisors of n

Brute force approach: Enumerate all numbers $i \in [1, n]$ and check if i divides n.

But we can use our domain knowledge and search only upto $\lceil \sqrt{n} \rceil$

- 32-bit number has up to 10 decimal digits.
  $\implies$ square root has about 5.
  $\implies$ 10^5 tests instead of 10^{10} tests
- 64-bit number has up to 20 decimal digits.
  $\implies$ square root has about 10.
  $\implies$ 10^{10} tests instead of 10^{20}.

Example: Queens Problem

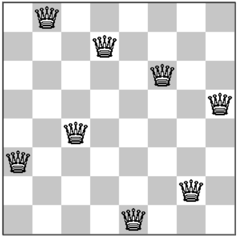


Queens Problem

Place 8 queens on a chess board such that they cannot threaten each other.

- Very naive approach: 9^{64} configurations (one of the 8 or none)

Example: Queens Problem

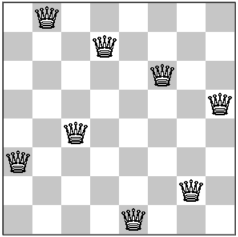


Queens Problem

Place 8 queens on a chess board such that they cannot threaten each other.

- Very naive approach: 9^{64} configurations (one of the 8 or none)
- Disregarding order: $\frac{64!}{56!}$ configurations.

Example: Queens Problem



Queens Problem

Place 8 queens on a chess board such that they cannot threaten each other.

- Very naive approach: 9^{64} configurations (one of the 8 or none)
- Disregarding order: $\frac{64!}{56!}$ configurations.
- All queens identical: $\binom{64}{8} \approx 4.4 \cdot 10^9$ configurations.

Generating Permutations

Algorithm to generate all permutations

Generates the lexicographical successor of a given permutation a.

1. Find largest index k: $a[k] < a[k+1]$
   If k does not exist then this is the last permutation
2. Find largest index l: $a[k] < a[l]$
3. Swap values $a[k]$ and $a[l]$
4. Reverse sequence from $a[k+1]$ up to the final element $a[n]$.

Generating Permutations

Algorithm to generate all permutations

Generates the lexicographical successor of a given permutation a.

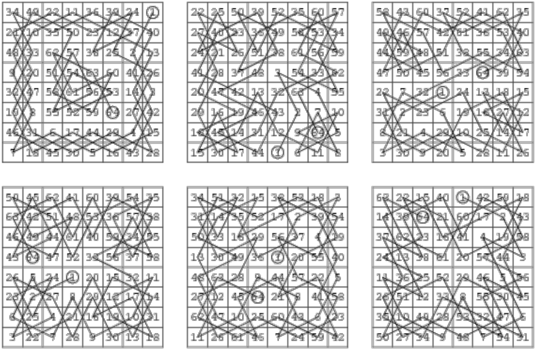
1. Find largest index k: $a[k] < a[k+1]$
   If k does not exist then this is the last permutation
2. Find largest index l: $a[k] < a[l]$
3. Swap values $a[k]$ and $a[l]$
4. Reverse sequence from $a[k+1]$ up to the final element $a[n]$.

Complexity

Since all steps are linear, we obtain a run time that is linear in n
$\Longrightarrow$ enumerating all permutations takes time $n! \cdot \mathcal{O}(n)$.

For a faster algorithm, see Steinhaus-Johnson-Trotter algorithm.

Knight's Tour



Solving Knight's Tour

Naive Solution

Generate all tours (permutations of [1..64]) and check whether the Knight can travel along such a path.

Solving Knight's Tour

Naive Solution

Generate all tours (permutations of [1..64]) and check whether the Knight can travel along such a path.

$64! \approx 10^{89}$ — impossible!

What other methods can you think of?

Backtracking

- Applicable when there exists
  - *Partial candidate solutions*
  - *Fast way of checking if the partial candidate can be completed*
- Consider search space as a tree
  Internal nodes represent partial solutions
- Dismiss subtree – **prune/backtrack** – if partial solution can't be completed

Example: CNF SAT

Given a boolean formula $\varphi(x_1, \ldots, x_n)$, is there a variable assignment such that φ is satisfied?
We may represent the space of all variable assignments as a tree.

Backtracking Pseudocode

Given a problem which admits partial solutions:

- *valid(s)*: Is partial solution s worth completing?
- *completed(c)*: Is c a complete solution?
- *next(c)*: Set of extensions of c by one step.

Backtracking Pseudocode

Given a problem which admits partial solutions:

- *valid(s)*: Is partial solution s worth completing?
- *completed(c)*: Is c a complete solution?
- *next(c)*: Set of extensions of c by one step.

```
function BACKTRACK(c)
    if !valid(c) then
        return false
    if completed(c) then
        output(c)
        return true
    for all c' in next(c) do
        if BACKTRACK(c') then
            return true
```

Constraint Satisfaction Problem

Constraint Satisfaction Problem: Find assignment to variables $\mathbb{X}$ such that some constraints $\mathbb{C}$ are satisfied.

Many discrete optimization/search problems can be specified as CSPs.

- Puzzles (Crossword, Sudoku)
- Graph Coloring
- Combinatorial Optimization (e.g. Knapsack)

CSP: Sudoku

Goal: Find integers $\mathbb{X} = (x_1, x_2, x_3 \ldots x_{81})$ in $[1 \ldots 9]^{81}$ satisfying $\mathbb{F} =$ sudoku constraints.

In order to use backtracking, we need $valid(c), completed(c)$ and $next(c)$ where c is a partial solution.

CSP: Sudoku

Goal: Find integers $\mathbb{X} = (x_1, x_2, x_3 \dots x_{81})$ in $[1 \dots 9]^{81}$ satisfying $\mathbb{F} =$ sudoku constraints.

In order to use backtracking, we need $\mathit{valid}(c), \mathit{completed}(c)$ and $\mathit{next}(c)$ where c is a partial solution.

Given partial solution, $c = (y_1, y_2, \dots y_k), k \leq n$

$\mathit{next}(c) = \{(y_1, y_2, \dots y_k, 1), (y_1, y_2, \dots y_k, 2), \dots (y_1, y_2, \dots y_k, 9)\}$

$\mathit{valid}(c) =$ effective way to check that c doesn't violate $\mathbb{F}$

Backtracking: Tips

- The order in which you complete your solution candidates matters.
- The better the order, the more branches of the tree can be cut off.
- Example: CNF SAT

This week's assignment

- There are 7 problems of which two problems cannot be solved within the given time bound.
  Do not upload solutions to infeasible problems. Upload submissions to five problems (no less), even if you didn't manage to solve all feasible problems.
- Scoreboard is reshuffled so that you cannot infer which problems are feasible.