

Subset Sum

Subset Sum

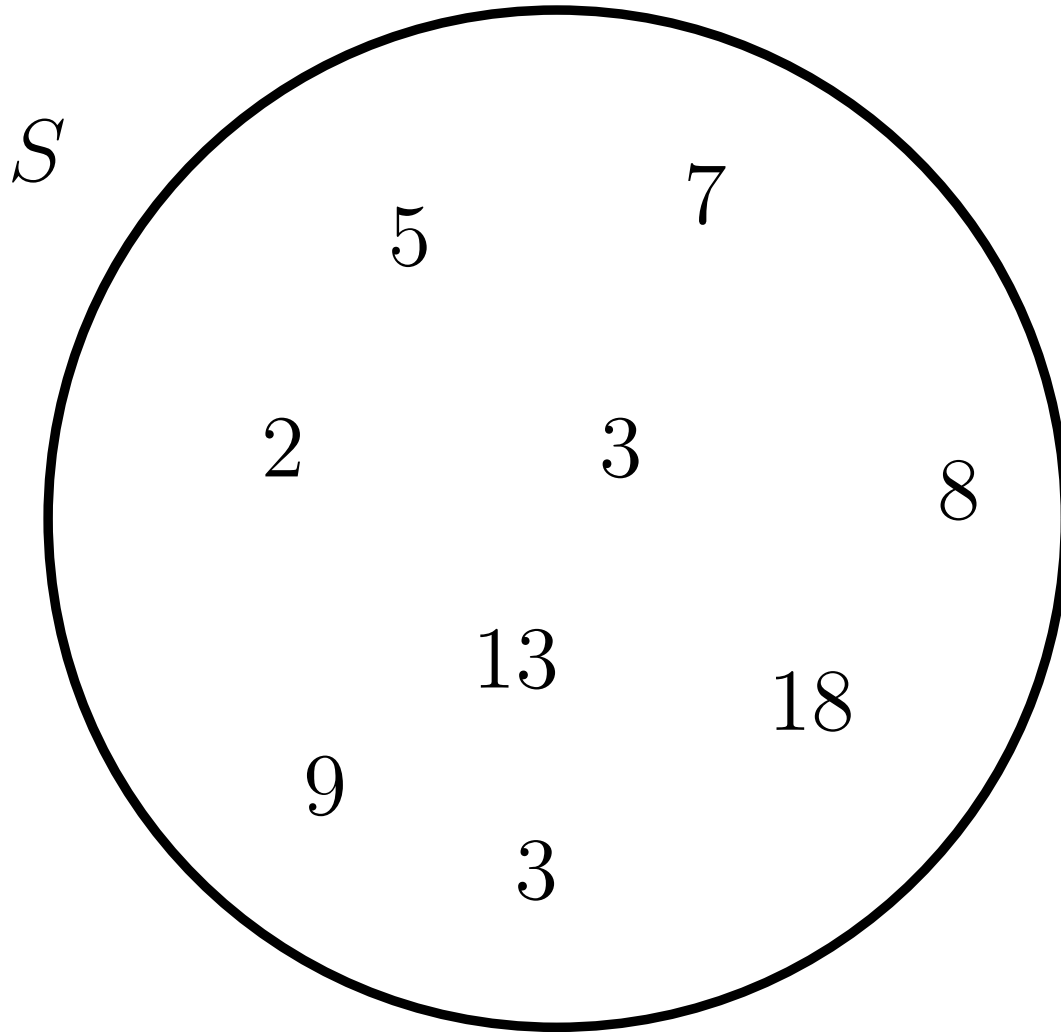
Input

- A (multi)-set $S \subseteq \mathbb{N}^+$ of n positive integers s_1, \dots, s_n .
- A *target value* $T \in \mathbb{N}^+$.

Question

Is there a subset $S' \subseteq S$ such that $\sum_{x \in S'} x = T$?

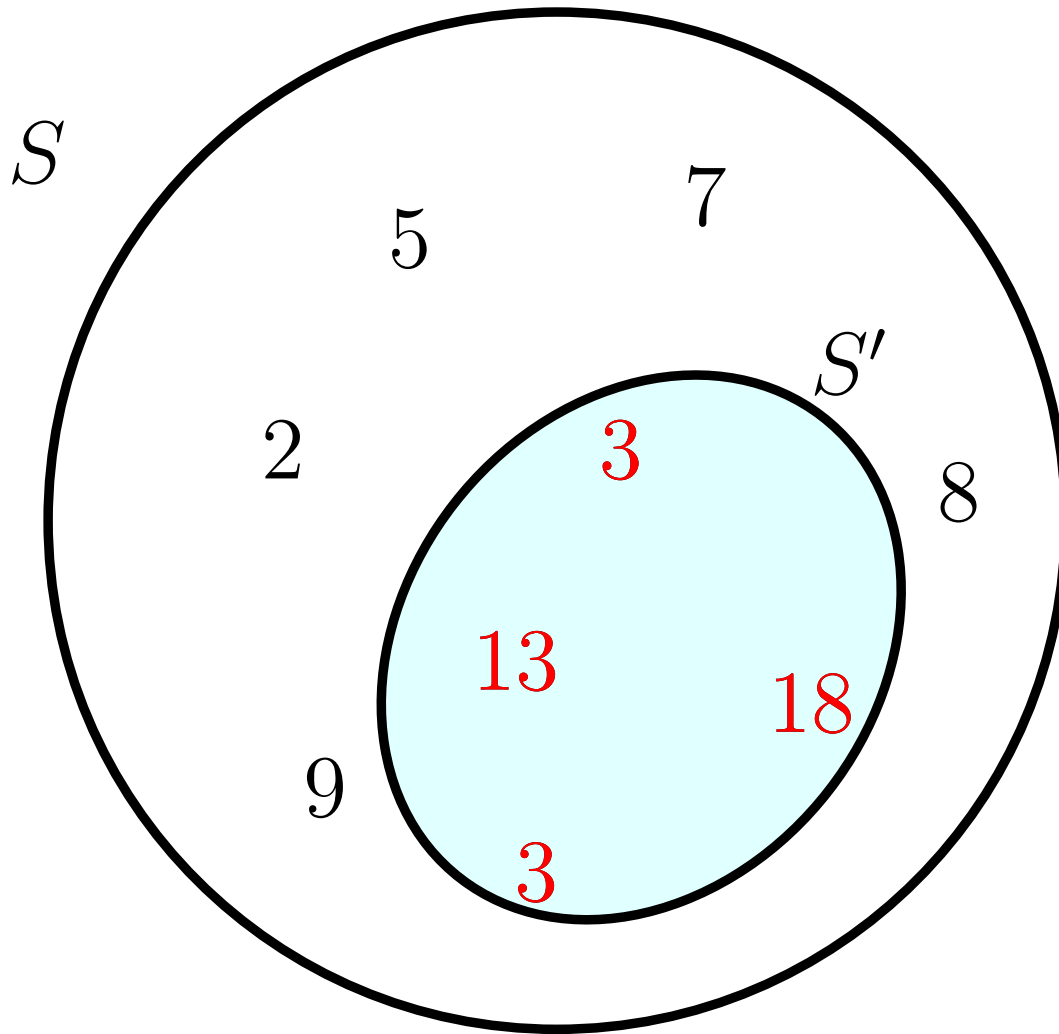
Example



S

$T = 37$

Example



$$T = 37$$

Answer: YES!

A Dynamic Programming Algorithm

Subproblem definition:

$$OPT[i, t] = \text{true iff } \exists S'' \subseteq \{s_1, \dots, s_i\} \text{ such that } \sum_{x \in S''} x = t.$$

Base cases:

$$OPT[0, 0] = \text{true}.$$

$$OPT[0, t] = \text{false, for } t > 0.$$

Recursive formula:

A Dynamic Programming Algorithm

Subproblem definition:

$OPT[i, t] = \text{true}$ iff $\exists S'' \subseteq \{s_1, \dots, s_i\}$ such that $\sum_{x \in S''} x = t$.

Base cases:

$OPT[0, 0] = \text{true}$.

$OPT[0, t] = \text{false}$, for $t > 0$.

Recursive formula:

- Either we ignore s_i

$$OPT[i, t] = OPT[i - 1, t]$$

- Or we include s_i in S'' ...

$$OPT[i, t] = OPT[i - 1, t - s_i]$$

(as long as $t \geq s_i$)

A Dynamic Programming Algorithm

Subproblem definition:

$$OPT[i, t] = \text{true iff } \exists S'' \subseteq \{s_1, \dots, s_i\} \text{ such that } \sum_{x \in S''} x = t.$$

Base cases:

$$OPT[0, 0] = \text{true.}$$

$$OPT[0, t] = \text{false, for } t > 0.$$

Recursive formula:

$$OPT[i, t] = \begin{cases} OPT[i - 1, t] & \text{if } t < s_i \\ OPT[i - 1, t] \vee OPT[i - 1, t - s_i] & \text{if } t \geq s_i \end{cases}$$

Time Complexity

- $\Theta(n \cdot T)$ subproblems
- Each problem can be solved in constant time
- **Overall time:** $\Theta(n \cdot T)$

Is this a polynomial-time algorithm?

Time Complexity

- $\Theta(n \cdot T)$ subproblems
- Each problem can be solved in constant time
- **Overall time:** $\Theta(n \cdot T)$

Is this a polynomial-time algorithm?


NO!

The input size is $O(n \log T)$ (Under reasonable assumptions)


Choose, e.g., $T = 2^n$.

This is called a *pseudo*-polynomial-time algorithm.

Can we do better?

- Subset Sum is a well-known NP-complete problem.
- A polynomial-time algorithm for Subset Sum would imply $P=NP$. 
- Let's give up on polynomial-time algorithms and look at exponential algorithms.
- **Easy exercise:** come up with an algorithm with time complexity $O^*(2^n)$. (OK for $n \approx 25$)
- Can the exponent be improved?

Can we do better?

- Subset Sum is a well-known NP-complete problem.
- A polynomial-time algorithm for Subset Sum would imply $P=NP$. 
- Let's give up on polynomial-time algorithms and look at exponential algorithms.
- **Easy exercise:** come up with an algorithm with time complexity $O^*(2^n)$. (OK for $n \approx 25$)
- $O^*(2^n)$ is a shorthand for $O(2^n \cdot \text{poly}(n))$.

Split & List

Split & List

Partition S into S_1 and S_2 .

Observation: The following two statements are equivalent:

- $\exists S' \subseteq S$ such that $\sum_{x \in S'} x = T$; and
- $\exists S'_1 \subseteq S_1, S'_2 \subseteq S_2$ such that $\sum_{x \in S'_1} x + \sum_{x \in S'_2} x = T$.

Idea: Check whether the second statement hold.

How does this help?

The Algorithm

- Partition S into S_1 and S_2 .
- $T_1 \leftarrow$ Set of the sums of *all possible* subsets of S_1 .
- $T_2 \leftarrow$ Set of the sums of *all possible* subsets of S_2 .
- $T_2 \leftarrow$ Sort T_2 .
- For each $t \in T_1$
 - Check whether $T - t \in T_2$

The Algorithm

- Partition S into S_1 and S_2 . $O(n)$
- $T_1 \leftarrow$ Set of the sums of *all possible* subsets of S_1 . $O(2^{|S_1|})$
- $T_2 \leftarrow$ Set of the sums of *all possible* subsets of S_2 . $O(2^{|S_2|})$
- $T_2 \leftarrow$ Sort T_2 . $O(|S_2| \cdot 2^{|S_2|})$
- For each $t \in T_1$ $|T_1| = O(2^{|S_1|})$
 - Check whether $T - t \in T_2$ $O(\log |T_2|) = O(|S_2|)$

$$O\left(|S_2| \cdot 2^{|S_1|} + |S_2| \cdot 2^{|S_2|}\right) = O^*\left(2^{|S_1|} + 2^{|S_2|}\right)$$

The Algorithm

- Partition S into S_1 and S_2 . $O(n)$
- $T_1 \leftarrow$ Set of the sums of *all possible* subsets of S_1 . $O(2^{|S_1|})$
- $T_2 \leftarrow$ Set of the sums of *all possible* subsets of S_2 . $O(2^{|S_2|})$
- $T_2 \leftarrow$ Sort T_2 . $O(|S_2| \cdot 2^{|S_2|})$
- For each $t \in T_1$ $|T_1| = O(2^{|S_1|})$
 - Check whether $T - t \in T_2$ $O(\log |T_2|) = O(|S_2|)$

Choosing $|S_1| = \lfloor \frac{n}{2} \rfloor$ and $|S_2| = \lceil \frac{n}{2} \rceil$:

$$O^* \left(2^{|S_1|} + 2^{|S_2|} \right) = O^* \left(2^{n/2} + 2^{n/2} \right) = O^* \left(2^{\frac{n}{2}} \right)$$

Intermission: Generating All Subsets

- Let S be a set of n elements, where n is *small*.
- **Option 1:** use integers to encode the characteristic vectors of all subsets $S' \subseteq S$

$$S = \{ 2, 5, 3, 13, 7, 8, 9, 18, 3 \}$$
$$x = 0b \quad 0 \quad 1 \quad 1 \quad 0 \quad 0 \quad 1 \quad 0 \quad 1 \quad 0$$
$$S' = \{ \quad 5, 3, \quad 8, \quad 18 \quad \}$$

```
uint64_t nsums = static_cast<uint64_t>(1)<<S.size(); //2^n
std::vector<int> sums(nsums, 0);
for(uint64_t x=0; x<nsums; x++)
    for(unsigned int i=0; i<S.size(); i++)
        sums[x] += ( (x>>i) & 1u )?S[i]:0;
```

Time: $O(n \cdot 2^n)$

$\approx 2s$ for $n = 25$

Intermission: Generating All Subsets

- Let S be a set of n elements, where n is *small*.
- **Option 1:** use integers to encode the characteristic vectors of all subsets $S' \subseteq S$

$$S = \{ 2, 5, 3, 13, 7, 8, 9, 18, 3 \}$$
$$x = 0b \quad 0 \quad 1 \quad 1 \quad 0 \quad 0 \quad 1 \quad 0 \quad 1 \quad 0$$
$$S' = \{ \quad 5, 3, \quad 8, \quad 18 \quad \}$$

```
uint64_t nsums = static_cast<uint64_t>(1)<<S.size(); //2^n
std::vector<int> sums(nsums, 0);
for(uint64_t x=0; x<nsums; x++)
    for(unsigned int i=0; i<S.size(); i++)
        sums[x] += ( (x>>i) & 1u ) * S[i];
```

Time: $O(n \cdot 2^n)$

$\approx 0.75s$ for $n = 25$

Intermission: Generating All Subsets

- **Option 2:** explicitly maintain the characteristic vector.
- *Update* the previous sum when the characteristic vector changes.

$$S = \{ 2 , 5 , 3 , 13 , 7 , 8 , 9 , 18 , 3 \}$$
$$0 \quad 1 \quad 1 \quad 0 \quad 0 \quad 1 \quad 0 \quad 1 \quad 0$$

$$\text{sum} = 34$$

Intermission: Generating All Subsets

- **Option 2:** explicitly maintain the characteristic vector.
- *Update* the previous sum when the characteristic vector changes.

$$S = \{ 2, 5, 3, 13, 7, 8, 9, 18, 3 \}$$
$$0 \quad 1 \quad 1 \quad 0 \quad 0 \quad 1 \quad 0 \quad 1 \quad 1$$

$$\text{sum} = 37$$

Intermission: Generating All Subsets

- **Option 2:** explicitly maintain the characteristic vector.
- *Update* the previous sum when the characteristic vector changes.

$$S = \{ 2, 5, 3, 13, 7, 8, 9, 18, 3 \}$$

0	1	1	0	0	1	1	0	0
---	---	---	---	---	---	---	---	---

$$\text{sum} = 25$$

Intermission: Generating All Subsets

- **Option 2:** explicitly maintain the characteristic vector.
- *Update* the previous sum when the characteristic vector changes.

$$S = \{ 2, 5, 3, 13, 7, 8, 9, 18, 3 \}$$

0	1	1	0	0	1	1	0	0
---	---	---	---	---	---	---	---	---

$$\text{sum} = 25$$

Time complexity?

Intermission: Generating All Subsets

b_{n-1}			\dots			b_2	b_1	b_0
0	1	1	0	0	1	0	1	0

- b_0 flips at every iteration
- b_1 flips every 2 iterations
- b_2 flips every 4 iterations
- \dots
- b_i flips every 2^i iterations

Intermission: Generating All Subsets

b_{n-1}				\dots			b_2	b_1	b_0
0	1	1	0	0	1	0	0	1	0

- b_0 flips at every iteration
- b_1 flips every 2 iterations
- b_2 flips every 4 iterations
- \dots
- b_i flips every 2^i iterations

Total # of operations (including updates to sum) \propto # bit flips

$$\sum_{i=0}^{n-1} \frac{2^n}{2^i} = \sum_{i=1}^n 2^i = 2^{n+1} - 2 = \Theta(2^n).$$

Generating All Subsets

```
uint64_t nsums = static_cast<uint64_t>(1)<<S.size(); //2^n
std::vector<int> sums(nsums);
std::vector<bool> bits(S.size());

for(uint64_t i=1; i<nsums; i++)
{
    sums[i] = sums[i-1];

    int j=0;
    while(bits[j])
    {
        bits[j] = 0;
        sums[i] -= S[j];
        j++;
    }

    bits[j]=1;
    sums[i] += S[j];
}
```

≈ 0.2s for $n = 25$

Back to Subset Sum: Which Algorithm?

Dynamic Programming

$$O(n \cdot T)$$

$$T \leq 2^{\frac{n}{2}}$$

OK for “small” T

Split and List

$$O(n \cdot 2^{\frac{n}{2}})$$

$$T \geq 2^{\frac{n}{2}}$$

OK for $n \leq 50$, regardless of T

Split & List

- Split input into two sets S_1, S_2
- Explicitly compute all possible (partial) solutions w.r.t. S_1 and S_2

Brute force!



- Combine the solutions of S_1 with those of S_2

Quicker than brute force



Can we split into 3 sets?

Binary Knapsack

Binary Knapsack

Input

- You are given a collection \mathcal{I} of n items indexed from 1 to n .
- Item i has a weight $w_i : \mathbb{N}^+$ and a value $v_i \in \mathbb{N}^+$.
- You can carry an overall weight of at most $W \in \mathbb{N}$.

Goal

Find a subset of $S \subset \mathcal{I}$ such that:

- Its overall weight $w(S) = \sum_{i \in \mathcal{I}} w_i$ is at most W ; and
- Its overall value $v(S) = \sum_{i \in \mathcal{I}} v_i$ is maximized.

Two Algorithms

- **Dynamic programming:** parameterize weights, store values.

$$O(nW)$$

Good for $W = O(nV)$

- **Dynamic programming:** parameterize values, store weights.

$$O(nV^*) = O(n^2V)$$

Good for $W = \Omega(nV)$

Neither algorithm runs in polynomial-time!

What if V and W are large but there are few items (n is small)?

A Split & List Algorithm

- **Split:** let $S_1 = \{1, \dots, \lceil n/2 \rceil\}$ and $S_2 = S \setminus S_1$.

A Split & List Algorithm

- **Split:** let $S_1 = \{1, \dots, \lceil n/2 \rceil\}$ and $S_2 = S \setminus S_1$.
- **List:** Let L_1 (resp. L_2) be the list of all the pairs $(w(X), v(X))$ for each $X \subseteq S_1$ (resp. $X \subseteq S_2$).

A Split & List Algorithm

- **Split:** let $S_1 = \{1, \dots, \lceil n/2 \rceil\}$ and $S_2 = S \setminus S_1$.
- **List:** Let L_1 (resp. L_2) be the list of all the pairs $(w(X), v(X))$ for each $X \subseteq S_1$ (resp. $X \subseteq S_2$).
- Sort L_2 in lexicographic order.
- For each $(w, \cdot) \in L_2$, add a pair (w, v) in a list L'_2 , where $v = \max\{v' : (w', v') \in L_2, w' \leq w\}$

A Split & List Algorithm

- **Split:** let $S_1 = \{1, \dots, \lceil n/2 \rceil\}$ and $S_2 = S \setminus S_1$.
- **List:** Let L_1 (resp. L_2) be the list of all the pairs $(w(X), v(X))$ for each $X \subseteq S_1$ (resp. $X \subseteq S_2$).
- Sort L_2 in lexicographic order.
- For each $(w, \cdot) \in L_2$, add a pair (w, v) in a list L'_2 , where $v = \max\{v' : (w', v') \in L_2, w' \leq w\}$
- For each pair $(w, v) \in L_1$ such that $w \leq W$:
 - Binary search for the last pair $(w', v') \in L'_2$ for which $w' \leq W - w$, if any.
 - If w' exists, $v + v'$ is the value of a candidate solution.

A Split & List Algorithm

- **Split:** let $S_1 = \{1, \dots, \lceil n/2 \rceil\}$ and $S_2 = S \setminus S_1$.
- **List:** Let L_1 (resp. L_2) be the list of all the pairs $(w(X), v(X))$ for each $X \subseteq S_1$ (resp. $X \subseteq S_2$).
- Sort L_2 in lexicographic order.
- For each $(w, \cdot) \in L_2$, add a pair (w, v) in a list L'_2 , where $v = \max\{v' : (w', v') \in L_2, w' \leq w\}$
- For each pair $(w, v) \in L_1$ such that $w \leq W$:
 - Binary search for the last pair $(w', v') \in L'_2$ for which $w' \leq W - w$, if any.
 - If w' exists, $v + v'$ is the value of a candidate solution.
- **Return:** Best candidate solution, if any.

A Split & List Algorithm

- **Split:** let $S_1 = \{1, \dots, \lceil n/2 \rceil\}$ and $S_2 = S \setminus S_1$. $O(n)$
- **List:** Let L_1 (resp. L_2) be the list of all the pairs $(w(X), v(X))$ for each $X \subseteq S_1$ (resp. $X \subseteq S_2$). $O(2^{\frac{n}{2}})$
- Sort L_2 in lexicographic order. $O(n \cdot 2^{\frac{n}{2}})$
- For each $(w, \cdot) \in L_2$, add a pair (w, v) in a list L'_2 , where $v = \max\{v' : (w', v') \in L_2, w' \leq w\}$ $O(2^{\frac{n}{2}})$
- For each pair $(w, v) \in L_1$ such that $w \leq W$: $O(2^{\frac{n}{2}})$
 - Binary search for the last pair $(w', v') \in L'_2$ for which $w' \leq W - w$, if any. $O(n)$
 - If w' exists, $v + v'$ is the value of a candidate solution.
- **Return:** Best candidate solution, if any.

Three Algorithms

- **Dynamic programming:** parameterize weights, store values.

$$O(nW)$$

- **Dynamic programming:** parameterize values, store weights.

$$O(nV^*) = O(n^2V)$$

- **Split & List:**

$$O(n2^{\frac{n}{2}})$$

1-in-3 positive SAT

1-in-3 positive SAT

Input: A formula ϕ consisting of

- A set of n boolean variables x_1, \dots, x_n
- A collection of m clauses C_1, \dots, C_m , i.e., triples of variables $C_j = (c_j^{(1)}, c_j^{(2)}, c_j^{(3)}) \in \{x_1, \dots, x_n\}^3$

A truth assignment is a function $\tau : \{x_1, \dots, x_n\} \rightarrow \{\top, \perp\}$

- A clause $C_j = (c_j^{(1)}, c_j^{(2)}, c_j^{(3)})$ is *satisfied* by τ iff *exactly* one of $\tau(c_j^{(1)})$, $\tau(c_j^{(2)})$, and $\tau(c_j^{(3)})$ is \top .
- ϕ is satisfied iff all m clauses C_1, \dots, C_m are satisfied.

Question: Is there a truth assignment that satisfies ϕ ?

Example

Formula

$$\phi = (x_1, x_2, x_4) \wedge (x_2, x_4, x_5) \wedge (x_1, x_3, x_5) \wedge (x_2, x_3, x_1)$$

Example

Formula

$$\phi = (x_1, x_2, x_4) \wedge (x_2, x_4, x_5) \wedge (x_1, x_3, x_5) \wedge (x_2, x_3, x_1)$$

Satisfying assignment:

$$x_1 = \perp \quad x_2 = \perp \quad x_3 = \top \quad x_4 = \top \quad x_5 = \perp$$

Example

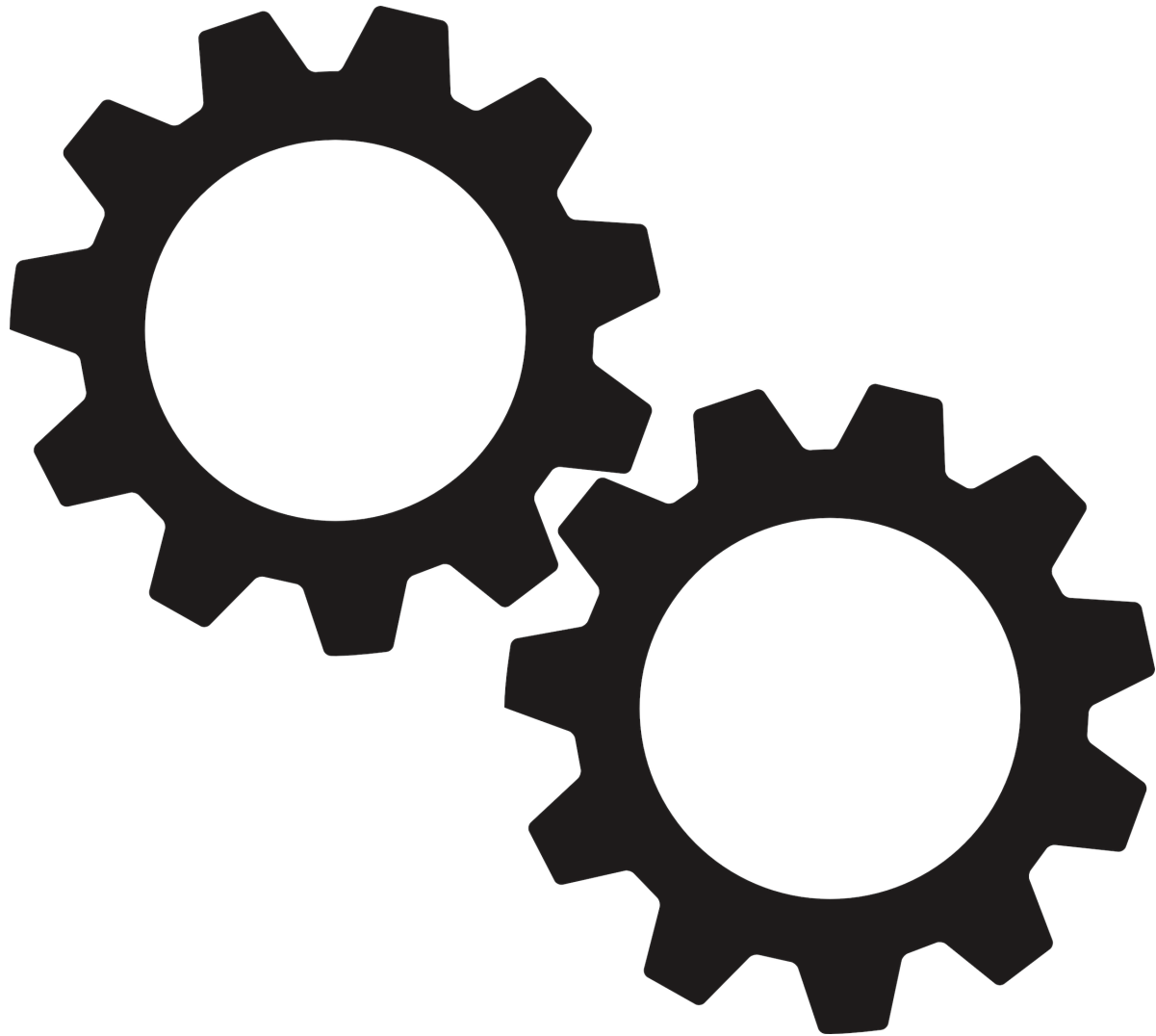
Formula

$$\phi = (x_1, x_2, x_4) \wedge (x_2, x_4, x_5) \wedge (x_1, x_3, x_5) \wedge (x_2, x_3, x_1)$$

Satisfying assignment:

$$x_1 = \perp \quad x_2 = \perp \quad x_3 = \top \quad x_4 = \top \quad x_5 = \perp$$

Trivial solution $O^*(2^n)$



An Algorithm Based on Split & List

- Split the n boolean variables into two sets S_1, S_2 of size $\approx \frac{n}{2}$
- For each possible truth assignment τ_1 of the variables in S_1
 - If τ_1 sets ≥ 2 variables in the same clause to \top , discard it.
 - Otherwise, store in a list L_1 the characteristic vector $\chi(\tau_1) = (\chi_1, \chi_2, \dots, \chi_m) \in \{\top, \perp\}^m$ of the satisfied clauses, where $\chi_j = \top$ iff τ_1 satisfies C_j .
- Build the list L_2 from S_2 in a similar way.
- Sort L_2 (e.g., w.r.t. the lexicographic order)
- For each vector $\chi = (\chi_1, \dots, \chi_m) \in L_1$
 - $\bar{\chi} \leftarrow (\bar{\chi}_1, \dots, \bar{\chi}_m)$
 - Binary search for $\bar{\chi}$ in L_2