

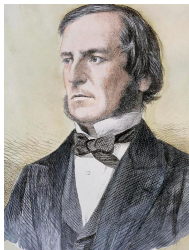
Subset Sum, and the
secret language of numbers

Boolean logic $x, y \in \{\text{TRUE}, \text{FALSE}\} = \{1, 0\}$

"x and y" $x \wedge y$

"x or y" $x \vee y$

"not x" $\bar{x}$ (or $\neg x$)



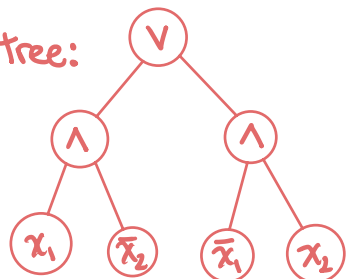
George Boole,
1815-1864

XOR ("exclusive-or")

(TRUE iff $x_1 \neq x_2$)

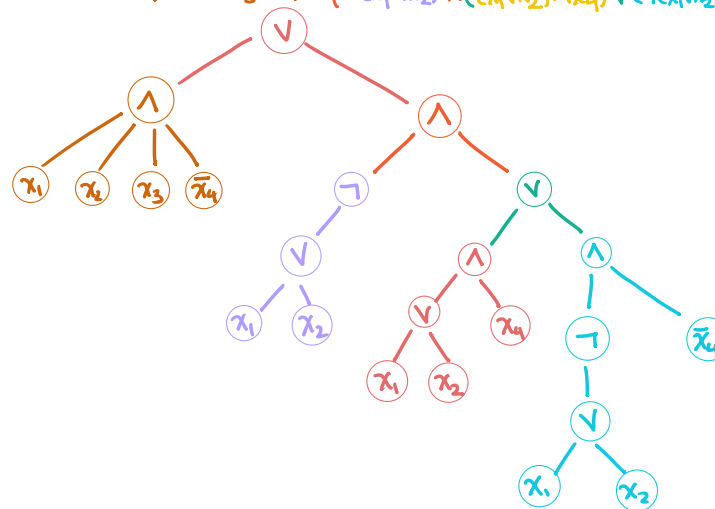
$$f(x_1, x_2) = (x_1 \wedge \bar{x}_2) \vee (\bar{x}_1 \wedge x_2)$$

as a tree:



Adding two bits $f(x_1, x_2, x_3, x_4) = \begin{cases} 1 & \text{if } x_1 + x_2 = x_3 x_4 \\ 0 & \text{otherwise} \end{cases}$

$$f(x_1, x_2, x_3, x_4) = (x_1 \wedge x_2 \wedge x_3 \wedge \bar{x}_4) \vee (\neg(x_1 \wedge x_2) \wedge ((x_1 \vee x_2) \wedge x_4) \vee (\neg(x_1 \vee x_2) \wedge \bar{x}_4))$$



"SAT Programming Language"

"x and y" $x \wedge y$

"x or y" $x \vee y$

"not x" $\bar{x}$ (or $\neg x$)

★ "z = x"

"both true or both false"
 $(z \wedge x) \vee (\bar{z} \wedge \bar{x})$

★ "if x then y else z"

"(x=1, y=1) or (x=0, y=0)"
 $(x \wedge y) \vee (\bar{x} \wedge \bar{y})$

So Boolean algebra is very expressive

Boolean Satisfiability

Input: Boolean formula $\varphi(x_1, x_2, \dots, x_n)$

• n variables • "size" m

e.g. $\varphi(0, 1, 0, 0, \neg, 1, 1) = 0$

Goal: decide if φ is "satisfiable";

i.e., if there exists $x_1, x_2, \dots, x_n \in \{0, 1\}$
s.t. $\varphi(x_1, x_2, \dots, x_n) = 1$

"search problem": we can recognize an answer (e.g. satisfying assignment x) when we see it (by evaluating $f(x)$).

- at the very least: we can try all x .
- harder "search problem" than "over/under"

Boolean Satisfiability

Input: Boolean formula $\varphi(x_1, x_2, \dots, x_n)$

- n variables
- "size" m

e.g. $\varphi(\overset{\text{(false)}}{0}, \overset{\text{(true)}}{1}, 0, 0, \neg, \overset{\text{(false)}}{1}, 1) = 0$

Boolean logic $x, y \in \{\text{TRUE}, \text{FALSE}\} = \{1, 0\}$

"x and y" $x \wedge y$

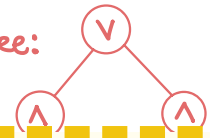
"x or y" $x \vee y$

"not x" $\bar{x}$ (or $\neg x$)

XOR ("exclusive-or") (TRUE iff $x_1 \neq x_2$)

$$f(x_1, x_2) = (x_1 \wedge \bar{x}_2) \vee (\bar{x}_1 \wedge x_2)$$

as a tree:



Goal: The Big Question

Is there a polynomial time algorithm for Boolean satisfiability?

x_2

$x_2 = x_3 x_4$

wise

$x_3 \wedge \bar{x}_4$

x_2

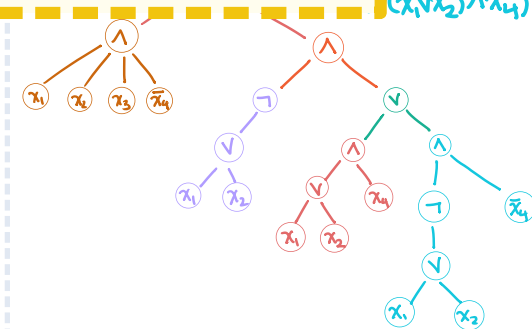
$x_2 \wedge x_4$

$(x_1 \vee x_2) \wedge \bar{x}_4$

answer (e.g. satisfying assignment x) when we see it (by evaluating $f(x)$).

- at the very least: we can try all x .

- harder "search problem" than "over/under"



"SAT Programming Language"

★ "z = x"

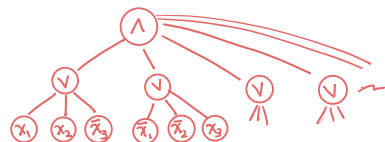
"both true or both false"
 $(z \wedge x) \vee (\bar{z} \wedge \bar{x})$

★ "if x then y else z"

"(x=1, y=1) or (x=0, z=1)"
 $(x \wedge y) \vee (\bar{x} \wedge z)$

CNF (Conjunctive Normal Form)

"and-of-or's"



$$f(x_1, \dots, x_n) =$$

"clause"

$$(\bar{x}_1 \vee x_2 \vee \bar{x}_3) \wedge (\bar{x}_1 \vee \bar{x}_2 \vee x_3) \wedge (m \vee n) \wedge m$$

(CNF-) SAT problem

Given boolean formula ϕ in CNF, decide if ϕ is satisfiable

More modest Question

Is there a polynomial time algorithm for CNF-satisfiability?

"CNF-SAT Programming Language"

"x and y"

$x \wedge y$

"x or y"

$x \vee y$

"not x"

$\overline{x}$ (or $\neg x$)

★ "Z=X"

"both true or both false"

$$(z \wedge x) \vee (\bar{z} \wedge \bar{x})$$

$$= (z \vee (\bar{z} \wedge \bar{x})) \wedge (x \vee (\bar{z} \wedge \bar{x}))$$

$$= (z \vee \bar{z}) \wedge (z \vee \bar{x}) \wedge (x \vee \bar{z}) \wedge (x \vee \bar{x})$$

$$= (z \vee \bar{x}) \wedge (x \vee \bar{y})$$

★ "if x then y else z"

$$"(x=1, y=1) \text{ or } (x=0, y=1)"$$

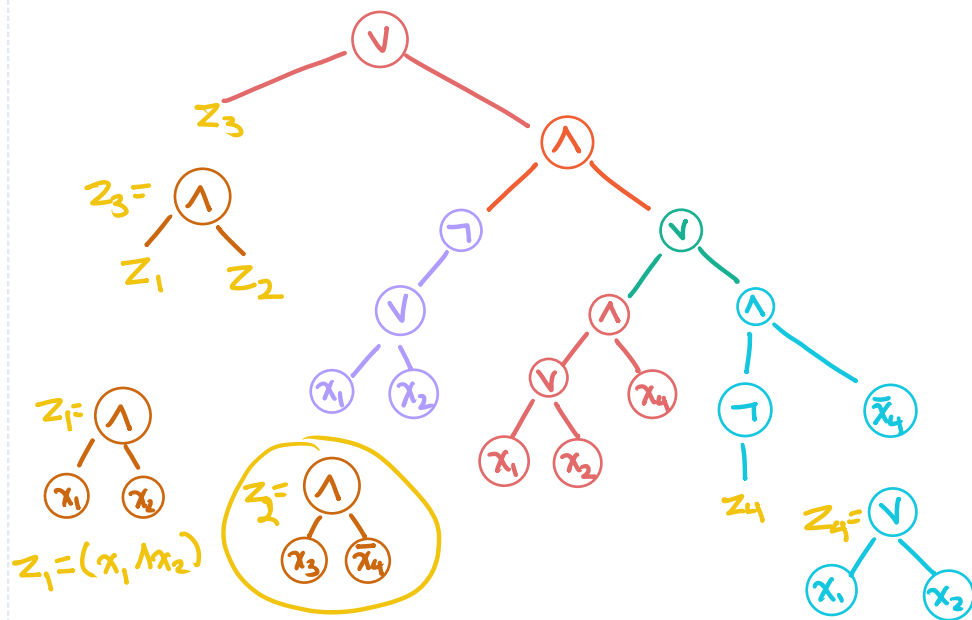
$$(x \wedge y) \vee (\bar{x} \wedge z)$$

$$= (x \vee \bar{x}) \wedge (y \vee \bar{x}) \wedge (x \vee z) \wedge (x \vee \bar{z})$$

$$= (y \vee \bar{x}) \wedge (x \vee z) \wedge (x \vee \bar{z})$$

So CNF is also very expressive

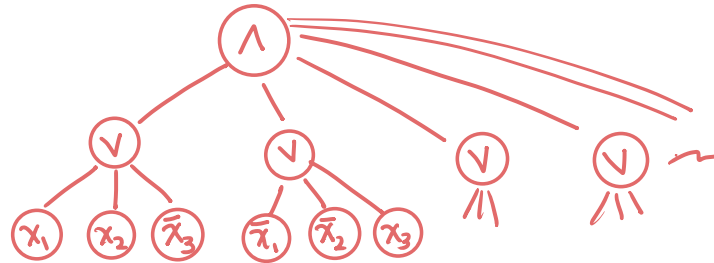
Theorem: there is a poly-time algo for Boolean SAT iff there is a poly-time algo for CNF-SAT.



more special case:

3-SAT

like CNF-SAT, except w/ exactly 3 variables per clause.



$$f(x_1, \dots, x_n) =$$

$$(\bar{x}_1 \vee x_2 \vee \bar{x}_3) \wedge (\bar{x}_1 \vee \bar{x}_2 \vee x_3) \wedge (\sim \vee \sim \vee \sim) \wedge \sim$$

"clause"

Is there a polytime algo for 3SAT?

Theorem: There is a poly-time algo for Boolean SAT iff there is a poly-time algo for 3-SAT.

Boolean logic $x, y \in \{\text{TRUE}, \text{FALSE}\} = \{1, 0\}$

"x and y" $x \wedge y$

"x or y" $x \vee y$

"not x" $\bar{x}$ (or $\neg x$)

CNF (Conjunctive Normal Form)

"and-of-or's"



$$f(x_1, \dots, x_n) =$$

$$(\bar{x}_1 \vee x_2 \vee \bar{x}_3) \wedge (\bar{x}_1 \vee \bar{x}_2 \vee x_3) \wedge (\sim \vee \sim \vee \sim) \wedge \sim$$

"clause"

(CNF $\rightarrow$) SAT problem

Given boolean formula f in CNF, decide if f is satisfiable

"CNF-SAT Programming Language"

★ "z = x"

"both true or both false"

$$(z \wedge x) \vee (\bar{z} \wedge \bar{x})$$

$$= (z \vee \bar{x}) \wedge (x \vee \bar{z})$$

★ "if x then y else z"

"(x=1, y=1) or (x=0, y=0)"

$$(x \wedge y) \vee (\bar{x} \wedge \bar{y})$$

$$= (y \vee \bar{x}) \wedge (x \vee \bar{y})$$

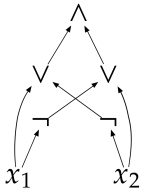
Theorem: There is a poly-time algo for Boolean SAT iff there is a poly-time algo for CNF-SAT.

Lemma. There is a polytime algo that, given a boolean formula $f(x_1, \dots, x_n)$ of size m , computes a CNF $g(x_1, \dots, x_p)$, w/ $p = O(m \log n)$ variables and size $O(m)$, s.t.

$$f \text{ is satisfiable} \Leftrightarrow g \text{ is satisfiable}$$

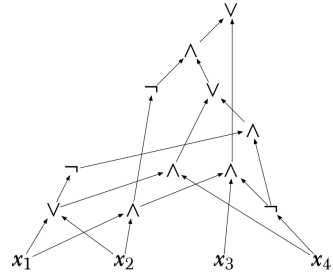
e.g. circuits

XOR ("exclusive-or")
(TRUE iff $x_1 \neq x_2$)

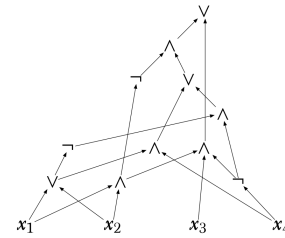
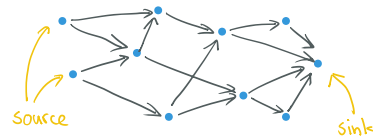


$$f(x_1, x_2) = (x_1 \wedge \bar{x}_2) \vee (\bar{x}_1 \wedge x_2)$$

Addition



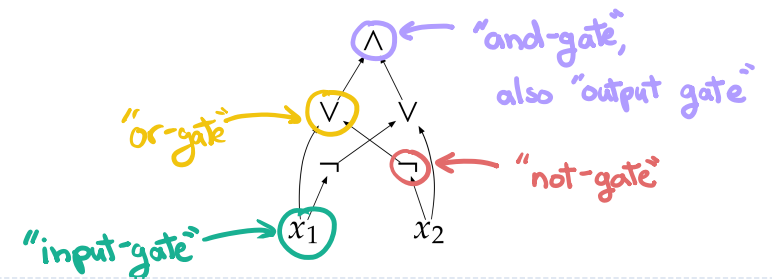
$$f(x_1, x_2, x_3, x_4) = \begin{cases} 1 & \text{if } x_1 + x_2 = x_3 x_4 \\ 0 & \text{otherwise} \end{cases}$$



(definition)

logical gates arranged as vertices
in a directed acyclic graph

"gates": take some inputs bits, output 1 bit



Circuit SAT

given a circuit, decide if it
is satisfiable

Theorem: there is a poly-time algo for
Boolean SAT iff there is a poly-time algo
for Circuit SAT

Conclusion

We want, but don't have, polynomial time algos for

Boolean SAT, CNF SAT, 3-SAT, or Circuit SAT

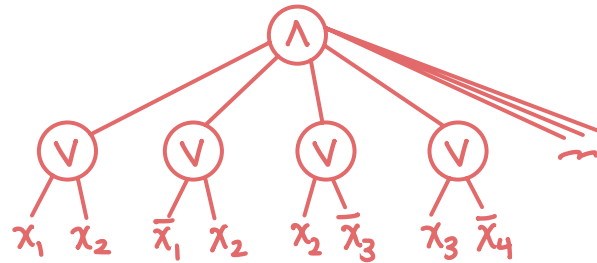
But we do know they are equivalent up to polynomial time!

2-SAT

like CNF-SAT, except w/ exactly 2 variables per clause.

$$f(x_1, \dots, x_n) =$$

$$(x_1 \vee x_2) \wedge (\bar{x}_1 \vee x_2) \wedge \underbrace{(x_2 \vee \bar{x}_3)}_{\text{clause}} \wedge (x_3 \vee \bar{x}_4) \wedge \dots$$



Max 2-SAT

like 2-SAT, except goal is to satisfy maximum number of clauses.
(not necessarily all)

Is there a polytime algo for Max 2-SAT?

Boolean logic $x, y \in \{\text{TRUE}, \text{FALSE}\} = \{1, 0\}$

"x and y" $x \wedge y$

"x or y" $x \vee y$

"not x" $\bar{x}$ (or $\neg x$)

CNF (Conjunctive Normal Form)

"and-of-or's"



$$f(x_1, \dots, x_n) =$$

$$(\bar{x}_1 \vee x_2 \vee \bar{x}_3) \wedge (\bar{x}_1 \vee \bar{x}_2 \vee x_3) \wedge (\dots) \wedge \dots$$

(CNF $\rightarrow$) SAT problem

Given boolean formula f in CNF, decide if f is satisfiable

"CNF-SAT Programming Language"

★ "z = x"

"both true or both false"

$$(z \wedge x) \vee (\bar{z} \wedge \bar{x})$$

$$= (z \vee \bar{x}) \wedge (x \vee \bar{z})$$

★ "if x then y else z"

"(x=1, y=1) or (x=0, y=0)"

$$(x \wedge y) \vee (\bar{x} \wedge \bar{y})$$

$$= (y \vee \bar{x}) \wedge (x \vee \bar{y})$$

Theorem: there is a poly-time algo for Boolean SAT iff there is a poly-time algo for CNF-SAT.

Lemma. There is a polytime algo that, given a boolean formula $f(x_1, \dots, x_n)$ of size m , computes a CNF $g(x_1, \dots, x_p)$, w/ $p = O(m \log n)$ variables and size $O(m)$, s.t.

f is satisfiable $\Leftrightarrow g$ is satisfiable

Theorem: there is a poly-time algo for Boolean SAT iff there is a poly-time algo for 3-SAT.

	Polytime Algo	Hard as ^(CNF-) SAT	Abstain
2SAT	5 8	10 4	
Max 2-SAT	10 0	5 12	0

let C be a sink SCC.

Complement $\bar{C}$ is a source SCC (why?)



Assign (all literals in) $C = \text{true}$, $\bar{C} = \text{false}$

Remove C and $\bar{C}$ and recurse on remaining SCC's

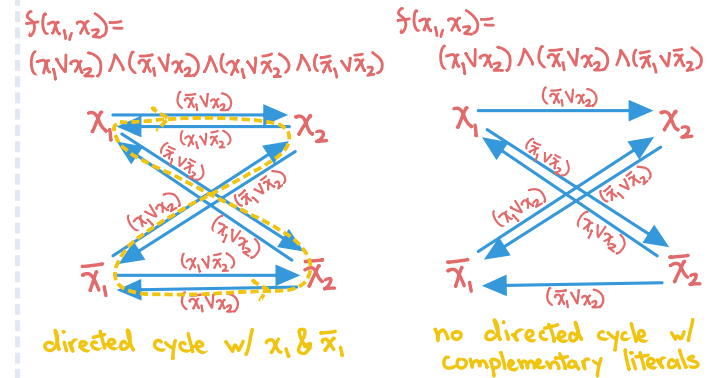
Key idea: assigning sinks to be true,
sources to be false
does not violate any edges

Implication graph

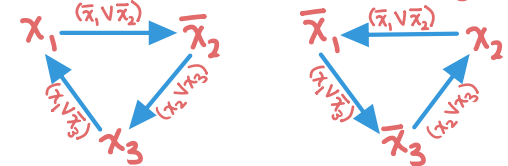
- vertex for each $x_i, \bar{x}_i$
- for clause $(a \vee b)$,

$$\bar{a} \xrightarrow{(a \vee b)} b \quad \bar{b} \xrightarrow{(a \vee b)} a$$

" $\bar{a} = \text{TRUE} \Rightarrow b = \text{TRUE}$ " " $\bar{b} = \text{TRUE} \Rightarrow a = \text{TRUE}$ "



all literals in SCC have same assignment



each SCC C has complement SCC $\bar{C}$
w/ negated literals & reversed edges

Fix a clause $(a \vee b \vee c)$

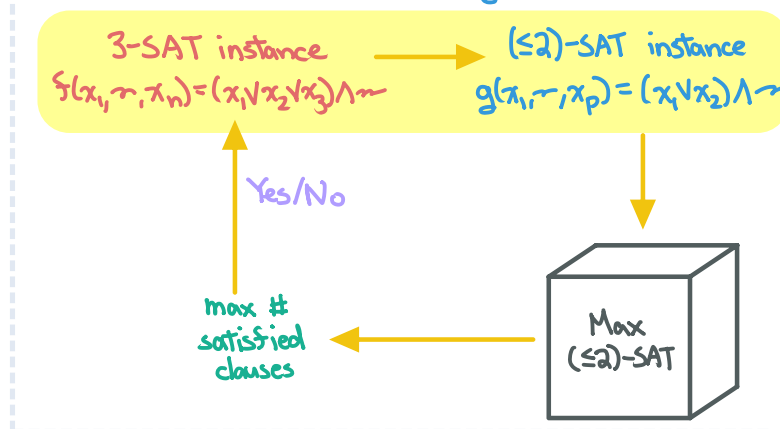
introduce new variable y

consider the 10 clauses

$(a), (b), (c), (\bar{a} \vee \bar{b}), (\bar{b} \vee \bar{c}), (\bar{c} \vee \bar{a}),$

$(y), (a \vee \bar{y}), (b \vee \bar{y}), (c \vee \bar{y})$

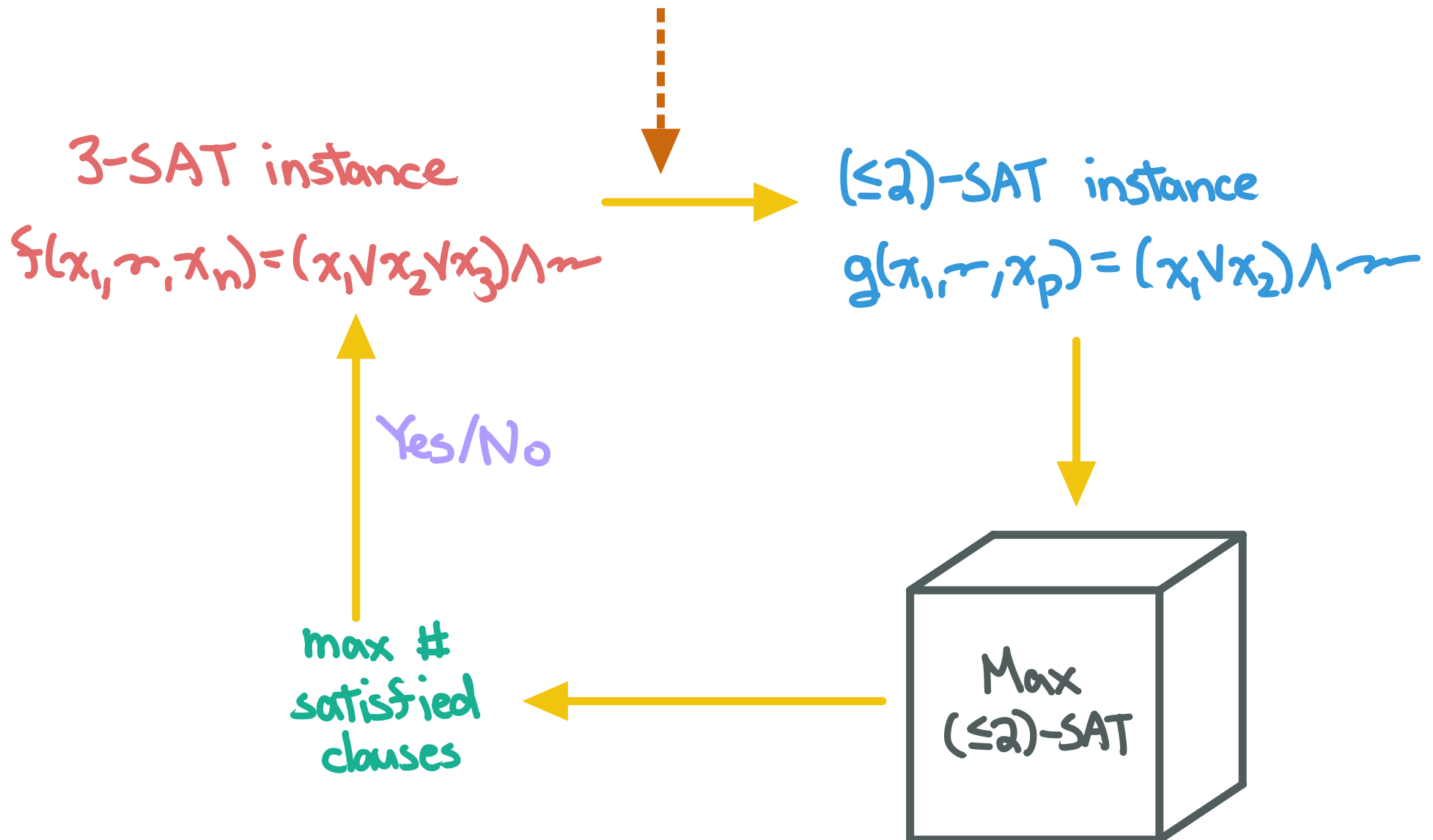
Theorem: A polytime algorithm for max (≤ 2) -SAT implies a polynomial time algorithm for SAT.



$(a, b, c) =$	(f, f, f)	(t, f, f)	(t, t, f)	(t, t, t)
$(a), (b), (c)$	0	1	2	3
$(\bar{a} \vee \bar{b}), (\bar{b} \vee \bar{c}), (\bar{c} \vee \bar{a})$	3	3	2	0
$\max_{y \in \{t, f\}} (y), (a \vee \bar{y}), (b \vee \bar{y}), (c \vee \bar{y})$	3	3	3	4
total	6	7	7	7

Lemma 10.3. Given a disjunction of 3 literals, $C = (a \vee b \vee c)$, one can compute a family $\mathcal{F}_C$ of 10 disjunctive clauses over a, b, c and an auxiliary variable y , each with at most 2 literals, such that for any assignment to a, b, c :

- (a) At most 7 clauses in $\mathcal{F}_C$ can be satisfied.
- (b) If $(a \vee b \vee c) = \text{true}$, then there is an assignment $y \in \{\text{true}, \text{false}\}$ that satisfies 7 of the clauses in $\mathcal{F}_C$.
- (c) If $(a \vee b \vee c) = \text{false}$, then any choice of $y \in \{\text{true}, \text{false}\}$ satisfies at most 6 of the clauses in $\mathcal{F}_C$.



Theorem: A polytime algorithm for max 2-SAT implies a polynomial time algorithm for SAT.

$$(a_i \vee y_i) \wedge (a_i \vee \bar{y}_i)$$

(≤ 2) -SAT instance

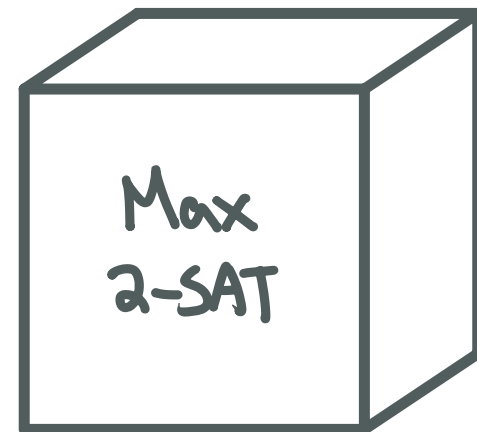
$$f(x_1, \dots, x_n) = (x_1 \vee x_2 \vee x_3) \wedge \dots$$

2-SAT instance

$$g(x_1, \dots, x_p) = (x_1 \vee x_2) \wedge \dots$$

max #
satisfied
clauses in g

$-k$



e.g. Making change (\$)

93 cents = 3 quarters,

1 dimes,

1 nickels,

3 pennies

Subset Sum

Input: $x_1, x_2, \dots, x_n \in \mathbb{N}$, $T \in \mathbb{N}$

Goal: Find a subset of x_i 's that sum to T
(decision version: decide if such a subset sum exists)

e.g. 14, 13, 17, 7, 18, 18, 10, 4, 11, 10

$T=52$

Another "search" problem

	Polytime Algo	Hard as ^(CNF-) SAT	Abstain
Subset Sum	5	8	0

Subset Sum

Input: $x_1, x_2, \dots, x_n \in \mathbb{N}$

Goal: Find a subset of x_i 's that sum to T
(decision version: decide if such a subset sum exists)

e.g. 14, 13, 17, 7, 18, 18, 10, 4, 11, 10

$T=52$

Ideas for algorithms?

DP

Recursive Spec 2^n

$SS(S, T)$: Given ~~subset~~ of numbers

S , return true if a subset of S adds to T

$SS(T)$ return true if a subset sums to T

$SS(k, T)$: return true if a subset of
 $x_1, \dots, x_k$ sums to T

Subset Sum

Input: $x_1, x_2, \dots, x_n \in \mathbb{N}$

Goal: Find a subset of x_i 's that sum to T
(decision version: decide if such a subset sum exists)

eg. 14, 13, 17, 7, 18, 18, 10, 4, 11, 10
 $T=52$

Recursive Spec

(Subset-Sum)

n

$SS((x_1, \dots, x_n), T) =$

TRUE if there is a subset of $x_1, \dots, x_n$ summing to T ,
and FALSE otherwise

Subset Sum

Input: $x_1, x_2, \dots, x_n \in \mathbb{N}$

Goal: Find a subset of x_i 's that sum to T
(decision version: decide if such a subset sum exists)

e.g. 14, 13, 17, 7, 18, 18, 10, 4, 11, 10
 $T=52$

Implementation

if $T < 0$ then False
if $T = 0$ then True

if $n = 0$ then False

$SS(n-1, T) \vee SS(n-1, T-x_n)$

Recursive Spec

(Subset-Sum)

$SS((x_1, \dots, x_n), T) =$

TRUE if there is a subset of $x_1, \dots, x_n$ summing to T ,
and FALSE otherwise

Subset Sum

Input: $x_1, x_2, \dots, x_n \in \mathbb{N}$

Goal: Find a subset of x_i 's that sum to T
(decision version: decide if such a subset sum exists)

e.g. 14, 13, 17, 7, 18, 18, 10, 4, 11, 10
 $T=52$

Implementation

$SS((x_1, \dots, x_n), T)$

1. If $T < 0$ then return FALSE

2. If $T = 0$ then return TRUE

3. If $n = 0$ then return FALSE

4. Return $SS((x_2, \dots, x_n), T) \vee SS((x_2, \dots, x_n), T - x_1)$

Running time:

Recursive Spec

(Subset-Sum)

$SS((x_1, \dots, x_n), T) =$

TRUE if there is a subset of $x_1, \dots, x_n$ summing to T ,
and FALSE otherwise

Subset Sum

Input: $x_1, x_2, \dots, x_n \in \mathbb{N}$

Goal: Find a subset of x_i 's that sum to T
(decision version: decide if such a subset sum exists)

e.g. 14, 13, 17, 7, 18, 18, 10, 4, 11, 10
 $T=52$

Implementation

$SS((x_1, \dots, x_n), T)$

1. If $T < 0$ then return FALSE

2. If $T = 0$ then return TRUE

3. If $n = 0$ then return FALSE

4. Return $SS((x_2, \dots, x_n), T) \vee SS((x_2, \dots, x_n), T - x_1)$

Running time: $T(n) = T(n-1) + O(1) = O(2^n)$

Recursive Spec

(Subset-Sum)

$SS(i, S) = \text{TRUE}$ if there is a subset of $x_1, \dots, x_n$ summing to S , and FALSE otherwise

Subset Sum

Input: $x_1, x_2, \dots, x_n \in \mathbb{N}$

Goal: Find a subset of x_i 's that sum to T
(decision version: decide if such a subset sum exists)

e.g. 14, 13, 17, 7, 18, 18, 10, 4, 11, 10
 $T=52$

Implementation

$SS(i, S)$

1. If $S < 0$ then return FALSE
2. If $S = 0$ then return TRUE
3. If $i > n$ then return FALSE
4. Return $SS(i+1, S) \vee SS(i+1, S - x_i)$

Running time:

$O(nT)$ subproblems

$O(1)$ time per subproblem

$O(nT)$ time overall

Subset Sum

Input: $x_1, x_2, \dots, x_n \in \mathbb{N}, T \in \mathbb{N}$

Goal: Find a subset of x_i 's that sum to T
(decision version: decide if such a subset sum exists)

e.g. 14, 13, 17, 7, 18, 18, 10, 4, 11, 10

$T=52$

	Polytime Algo	Hard as ^(CNF-) SAT	Abstain
Subset Sum			

Running time:

$O(nT)$ subproblems

$O(1)$ time per subproblem

$O(nT)$ time overall

"T" takes $\log(T)$ bits to write down

what if $T \approx 2^n$?

Sh*t. Any other ideas?

Subset Sum

Input: $x_1, x_2, \dots, x_n \in \mathbb{N}$

Goal: Find a subset of x_i 's that sum to T
(decision version: decide if such a subset sum exists)

e.g. 14, 13, 17, 7, 18, 18, 10, 4, 11, 10
 $T=52$

Recursive Spec

(Subset-Sum)
 $SS(i, S) = \text{TRUE}$ if there is a subset of $x_1, \dots, x_n$ summing to S , FALSE otherwise

Implementation

$SS(i, S)$

1. If $S < 0$ then return FALSE
2. If $S = 0$ then return TRUE
3. If $n = 0$ then return FALSE
4. Return $SS(i+1, S) \vee SS(i+1, S - x_i)$

Running time:

$O(nT)$ subproblems

$O(1)$ time per subproblem

$O(nT)$ time overall

Can we instead show that
Subset-Sum is hard as SAT?

Unlike previous problems, SubsetSum
looks nothing like SAT ~

Idea?
Primes? Number theory black magic

Theorem: A polytime algo for subset-sum
implies a polytime algo for SAT

Subset Sum

Input: $x_1, x_2, \dots, x_n \in \mathbb{N}$

Goal: Find a subset of x_i 's that sum to T
(decision version: decide if such a subset sum exists)

e.g. 14, 13, 17, 7, 18, 18, 10, 4, 11, 10
 $T=52$

Recursive Spec

(Subset-Sum)
 $SS(i, S) = \text{TRUE}$ if there is a subset
of $x_1, \dots, x_n$ summing to S , FALSE otherwise

Implementation

$SS(i, S)$

1. If $S < 0$ then return FALSE
2. If $S = 0$ then return TRUE
3. If $n = 0$ then return FALSE
4. Return $SS(i+1, S) \vee SS(i+1, S - x_i)$

Running time:

$O(nT)$ subproblems

$O(1)$ time per subproblem

$O(nT)$ time overall

"T" takes $\log(T)$ bits to write down

Theorem: A polytime algo for subset-sum implies a polytime algo for SAT

Subset Sum

Input: $x_1, x_2, \dots, x_n \in \mathbb{N}$

Goal: Find a subset of x_i 's that sum to T
(decision version: decide if such a subset sum exists)

e.g. 14, 13, 17, 7, 18, 18, 10, 4, 11, 10
 $T=52$

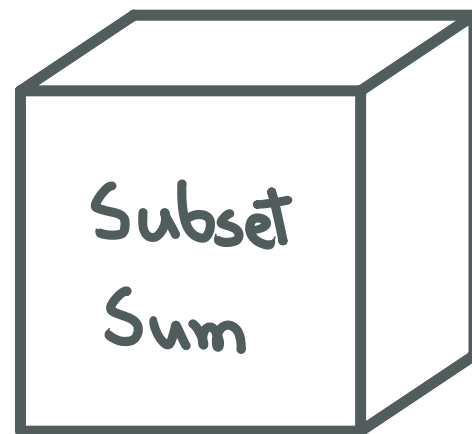
Proof setup.

3-SAT instance

$$\mathcal{F}(x_1, \dots, x_n) = (x_1 \vee x_2 \vee x_3) \wedge \dots$$

$x_1, x_2, \dots, x_p \in \mathbb{N}$
 $T \in \mathbb{N}$

Yes/No for
 $x_1, x_2, \dots, x_p \in \mathbb{N}$
 $T \in \mathbb{N}$



Key idea: numbers are bit-strings too!

— use many digits

— each digit encodes 1 constraint

Key idea: numbers are bit-strings too!

• use many digits • each digit encodes 1 constraint

1. Recast 3SAT as "subset" problem

$X = \text{literals } \{x_1, \bar{x}_1, x_2, \bar{x}_2, \dots, x_n, \bar{x}_n\}$

clause $(x_1 \vee \bar{x}_2 \vee x_3) = \text{set } \{x_1, \bar{x}_2, x_3\} \subset X$

Goal: find $S \subset X$ s.t.

Ⓐ for each clause C_i , $C_i \cap S = \emptyset$

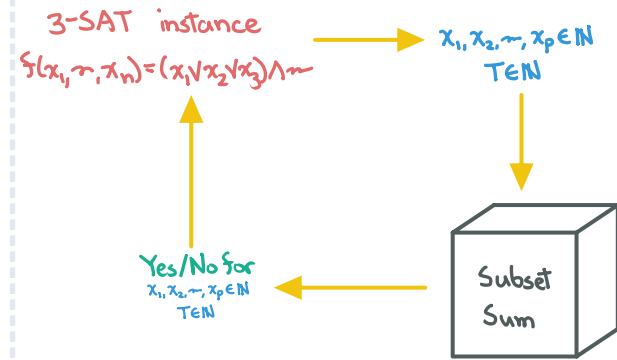
Ⓑ for each x_j , S contains (exactly) 1 of $\{x_j, \bar{x}_j\}$

Subset Sum

Input: $x_1, x_2, \dots, x_n \in \mathbb{N}$

Goal: find a subset of x_i 's that sum to T
(decision version: decide if such a subset sum exists)

e.g. 14, 13, 17, 7, 18, 18, 10, 4, 11, 10
 $T=52$



Key idea: numbers are bit-strings too!

• use many digits • each digit encodes 1 constraint

1. Recast 3SAT as "subset" problem

$X = \text{literals } \{x_1, \bar{x}_1, x_2, \bar{x}_2, \dots, x_n, \bar{x}_n\}$

clause $(x_1 \vee \bar{x}_2 \vee x_3) = \text{set } \{x_1, \bar{x}_2, x_3\} \subset X$

Goal: find $S \subset X$ s.t.

Ⓐ for each clause C_i , $C_i \cap S = \emptyset$

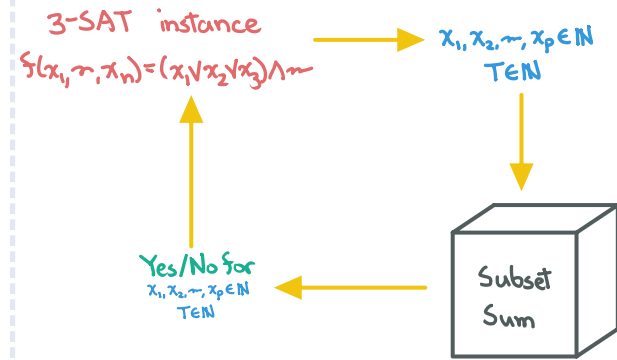
Ⓑ for each x_j , S contains (exactly) 1 of $\{x_j, \bar{x}_j\}$

Subset Sum

Input: $x_1, x_2, \dots, x_n \in \mathbb{N}$

Goal: find a subset of x_i 's that sum to T
(decision version: decide if such a subset sum exists)

e.g. 14, 13, 17, 7, 18, 18, 10, 4, 11, 10
 $T=52$



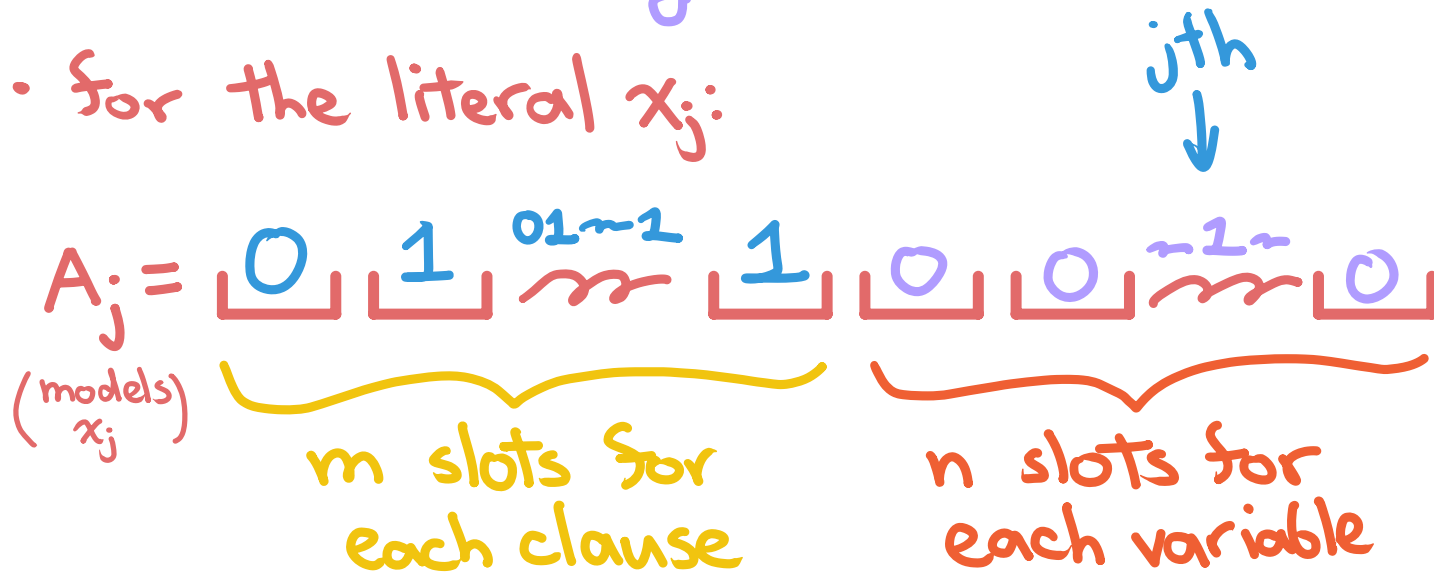
Key idea: numbers are bit-strings too!

- use many digits • each digit encodes 1 constraint

2. Start making #'s:

- we use mtn digits base 4

- for the literal x_j :



- for each clause C_i , set C_i th digit to 1 if $x_j \in C_i$, else 0

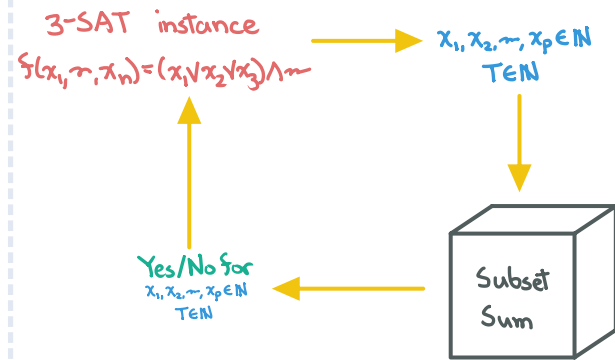
- set " x_j th digit" to 1, else 0

Subset Sum

Input: $x_1, x_2, \dots, x_n \in \mathbb{N}$

Goal: Find a subset of x_i 's that sum to T
(decision version: decide if such a subset sum exists)

e.g. 14, 13, 17, 7, 18, 18, 10, 4, 11, 10
 $T=52$



1. Recast 3SAT as "subset" problem

$X = \text{literals } \{x_1, \bar{x}_1, x_2, \bar{x}_2, \dots, x_n, \bar{x}_n\}$

clause $(x_1 \vee \bar{x}_2 \vee x_3) = \text{set } \{x_1, \bar{x}_2, x_3\} \subset X$

Goal: Find $S \subset X$ s.t.

Ⓐ for each clause C_i , $C_i \cap S = \emptyset$

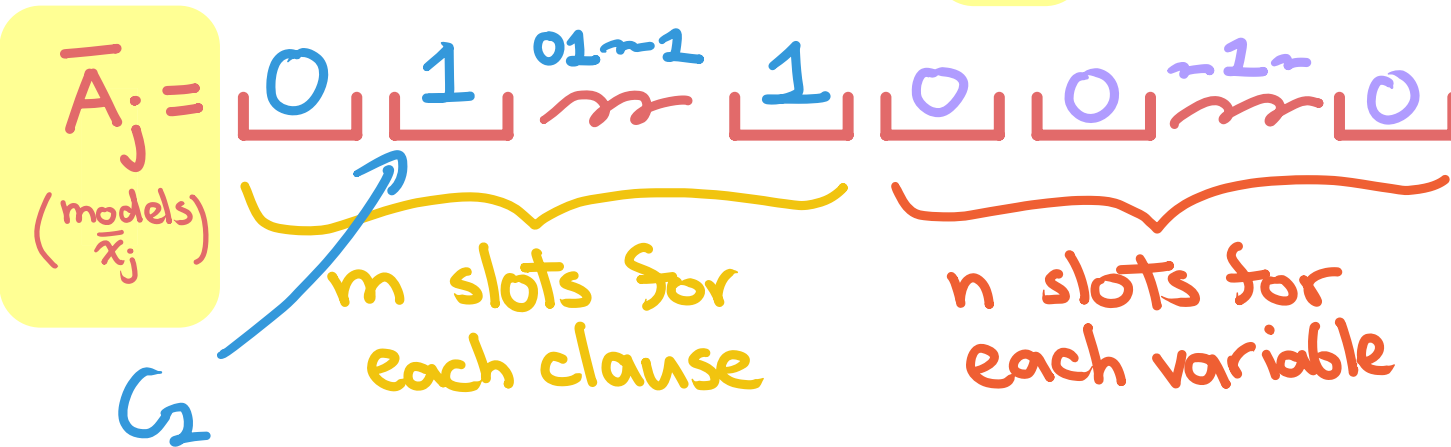
Ⓑ for each x_j , S contains ^(exactly) 1 of $\{x_j, \bar{x}_j\}$

Key idea: numbers are bit-strings too!

- use many digits • each digit encodes 1 constraint

2. Start making #'s:

- we use mtn digits base 4
- Similarly, for the literal $\bar{x}_j$:



- for each clause C_i , set C_i th digit to 1 if $\bar{x}_j \in C_i$, else 0

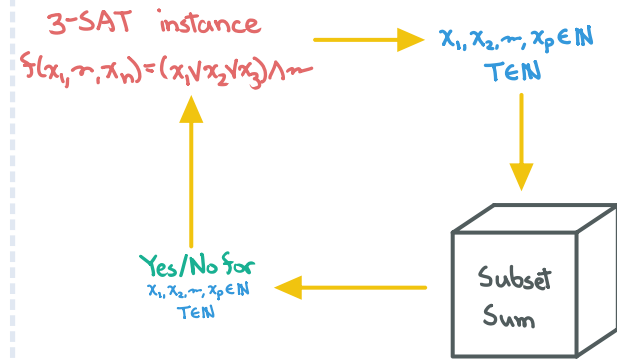
- set " x_j th digit" to 1, else 0

Subset Sum

Input: $x_1, x_2, \dots, x_n \in \mathbb{N}$

Goal: Find a subset of x_i 's that sum to T
(decision version: decide if such a subset sum exists)

e.g. 14, 13, 17, 7, 18, 18, 10, 4, 11, 10
 $T=52$



1. Recast 3SAT as "subset" problem

$X = \text{literals } \{x_1, \bar{x}_1, x_2, \bar{x}_2, \dots, x_n, \bar{x}_n\}$

clause $(x_1 \vee \bar{x}_2 \vee x_3) = \text{set } \{x_1, \bar{x}_2, x_3\} \subset X$

Goal: Find $S \subset X$ s.t.

(a) for each clause C_i , $C_i \cap S = \emptyset$

(b) for each x_j , S contains ^(exactly) 1 of $\{x_j, \bar{x}_j\}$

Key idea: numbers are bit-strings too!

• use many digits • each digit encodes 1 constraint

Literals

Long #'s

$$X = \{x_1, \bar{x}_1, \dots, x_n, \bar{x}_n\} \leftrightarrow \{A_1, \bar{A}_1, \dots, A_n, \bar{A}_n\}$$

Set SCX

$$D = \sum_{x_j \in S} A_j + \sum_{\bar{x}_j \in S} \bar{A}_j$$

$$\forall \text{ constraint } C_i \leftrightarrow \forall \text{ constraint } C_i, C_i \cap S \neq \emptyset$$
$$C_i \text{th digit of } D \geq 1$$

$$\forall \text{ variables } x_j \leftrightarrow \forall \text{ variables } x_j,$$

$$|\{x_j, \bar{x}_j\} \cap S| = 1$$

$$x_j \text{th digit of } D = 1$$

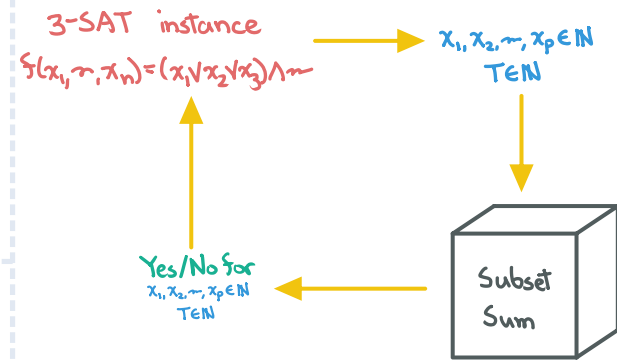
still one key diff. from subset sum...

Subset Sum

Input: $x_1, x_2, \dots, x_n \in \mathbb{N}$

Goal: Find a subset of x_i 's that sum to T
(decision version: decide if such a subset sum exists)

e.g. 14, 13, 17, 7, 18, 18, 10, 4, 11, 10
 $T=52$



1. Recast 3SAT as "subset" problem
 $X = \text{literals } \{x_1, \bar{x}_1, x_2, \bar{x}_2, \dots, x_n, \bar{x}_n\}$

clause $(x_1 \vee \bar{x}_2 \vee x_3) = \text{set } \{x_1, \bar{x}_2, x_3\} \subset X$

Goal: Find SCX st.

Ⓐ for each clause C_i , $C_i \cap S \neq \emptyset$

Ⓑ for each x_j , S contains ^(exactly) 1 of $\{x_j, \bar{x}_j\}$

2. Start making #'s:

• we use mtn digits base 4

• for the literal x_j :

$A_j = \underbrace{0 \ 1 \ 0 \ 1}_{m \text{ slots for each clause}} \underbrace{0 \ 0 \ 1 \ 0}_{n \text{ slots for each variable}}$

(models x_j)

$01 \sim 1$ (not $\bar{x}_j$)

$\sim 1 \sim 0$

• for each clause C_i , set C_i th digit to 1 if $x_j \in C_i$, else 0

• set " x_j th digit" to 1, else 0

Similar for the literal $\bar{x}_j$

Key idea: numbers are bit-strings too!

• use many digits • each digit encodes 1 constraint

Literals

Long #'s

$$X = \{x_1, \bar{x}_1, \dots, x_n, \bar{x}_n\} \leftrightarrow \{A_1, \bar{A}_1, \dots, A_n, \bar{A}_n\}$$

Set SCX

$$D = \sum_{x_j \in S} A_j + \sum_{\bar{x}_j \in S} \bar{A}_j$$

$$\forall \text{ constraint } C_i, C_i \cap S \neq \emptyset \leftrightarrow \forall \text{ constraint } C_i, C_i \text{th digit of } D \geq 1$$

$$\forall \text{ variables } x_j, |\{x_j, \bar{x}_j\} \cap S| = 1 \leftrightarrow \forall \text{ variables } x_j, x_j \text{th digit of } D = 1$$

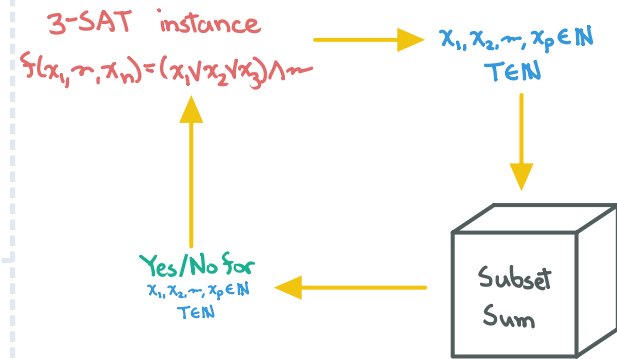
want to replace "≥" w/ "="

Subset Sum

Input: $x_1, x_2, \dots, x_n \in \mathbb{N}$

Goal: Find a subset of x_i 's that sum to T
(decision version: decide if such a subset sum exists)

e.g. 14, 13, 17, 7, 18, 18, 10, 4, 11, 10
T=52



1. Recast 3SAT as "subset" problem
 $X = \text{literals } \{x_1, \bar{x}_1, x_2, \bar{x}_2, \dots, x_n, \bar{x}_n\}$

clause $(x_1 \vee \bar{x}_2 \vee x_3) = \text{set } \{x_1, \bar{x}_2, x_3\} \subset X$

Goal: Find SCX st.

(a) for each clause C_i , $C_i \cap S \neq \emptyset$

(b) for each x_j , S contains ^(exactly) 1 of $\{x_j, \bar{x}_j\}$

2. Start making #'s:

• we use mtn digits base 4

• for the literal x_j :

$$A_j = \underbrace{0 \ 1 \ 0 \ 1}_{m \text{ slots for each clause}} \underbrace{0 \ 0 \ 1 \ 0}_{n \text{ slots for each variable}}$$

(models x_j)

• for each clause C_i , set C_i th digit to 1 if $x_j \in C_i$ (not $\bar{x}_j$), else 0

• set " x_j th digit" to 1, else 0

Similar for the literal $\bar{x}_j$

$\forall \text{ constraint } C_i, \leftrightarrow C_i \cap S \neq \emptyset$

$\forall \text{ constraint } C_i, C_i \text{th digit of } D \geq 1$

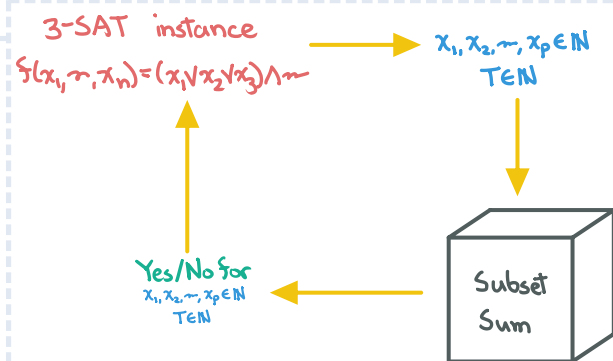
How to fix?

Subset Sum

Input: $x_1, x_2, \dots, x_n \in \mathbb{N}$

Goal: Find a subset of x_i 's that sum to T
(decision version: decide if such a subset sum exists)

e.g. 14, 13, 17, 7, 18, 18, 10, 4, 11, 10
 $T=52$



1. Recast 3SAT as "subset" problem

$X = \text{literals } \{x_1, \bar{x}_1, x_2, \bar{x}_2, \dots, x_n, \bar{x}_n\}$

clause $(x_1 \vee \bar{x}_2 \vee x_3) = \text{set } \{x_1, \bar{x}_2, x_3\} \subset X$

Goal: Find $S \subset X$ st.

(a) for each clause $C_i, C_i \cap S \neq \emptyset$

(b) for each x_j, S contains ^(exactly) 1 of $\{x_j, \bar{x}_j\}$

2. Start making #'s:

- we use mtrn digits base 4
- for the literal x_j :

$A_j = \underbrace{0 \ 1 \ 0 \ 1}_{\text{m slots for each clause}} \underbrace{0 \ 1 \ 0 \ 1}_{\text{n slots for each variable}}$
(models x_j)

- for each clause C_i , set C_i th digit to 1 if $x_j \in C_i$, else 0

- set " x_j th digit" to 1, else 0

Similar for the literal $\bar{x}_j$

$\forall \text{ constraint } C_i, \leftrightarrow \forall \text{ constraint } C_i, C_i \cap S \neq \emptyset$

Easy fix: "free padding"

for constraint C_i , create two copies of:

$B_i = 00 \sim 0100 \sim 0$
 $\uparrow$
 $C_i \text{th digit}$

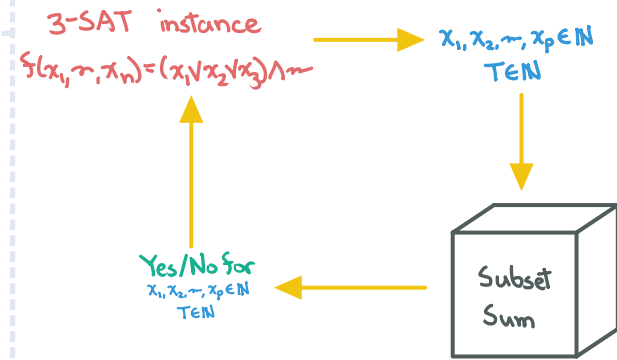
lets you increase $C_i \text{th digit}$ by 1 or 2

Subset Sum

Input: $x_1, x_2, \dots, x_n \in \mathbb{N}$

Goal: Find a subset of x_i 's that sum to T
 (decision version: decide if such a subset sum exists)

e.g. 14, 13, 17, 7, 18, 18, 10, 4, 11, 10
 $T=52$



1. Recast 3SAT as "subset" problem

$X = \text{literals } \{x_1, \bar{x}_1, x_2, \bar{x}_2, \dots, x_n, \bar{x}_n\}$

clause $(x_1 \vee \bar{x}_2 \vee x_3) = \text{set } \{x_1, \bar{x}_2, x_3\} \subset X$

Goal: Find $S \subset X$ st.

(a) for each clause C_i , $C_i \cap S \neq \emptyset$

(b) for each x_j , S contains ^(exactly) 1 of $\{x_j, \bar{x}_j\}$

2. Start making #'s:

• we use mtn digits base 4

• for the literal x_j :

$A_j = \boxed{0} \boxed{1} \overline{01 \sim 2} \boxed{1} \boxed{0} \boxed{0} \overline{\sim 1 \sim 2} \boxed{0}$
 (models) x_j $m \text{ slots for each clause}$ $n \text{ slots for each variable}$

• for each clause C_i , set $C_i \text{th digit}$ to 1 if $x_j \in C_i$ (not $\bar{x}_j$) else 0

• set " $x_j \text{th digit}$ " to 1, else 0

Similar for the literal $\bar{x}_j$

Key idea: numbers are bit-strings too!

• use many digits • each digit encodes 1 constraint

Literals

Long #'s

$X = \{x_1, \bar{x}_1, \dots, x_n, \bar{x}_n\} \leftrightarrow \{A_1, \bar{A}_1, \dots, A_n, \bar{A}_n\}$
and $\{B_1, B_1, \dots, B_m, B_m\}$

let $T = \underbrace{33\dots3}_m \text{ clauses } \underbrace{11\dots1}_n \text{ variables}$

for subset of A_i 's, $\bar{A}_i$'s, B_i 's summing to D :

• C_i th digit of $D = 3$ only if D include $A_j/\bar{A}_j$ for some $x_j/\bar{x}_j$ in C_i

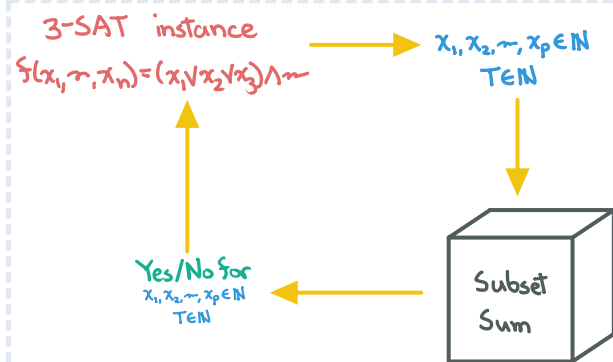
• x_j th digit of $D = 1 \Leftrightarrow$
includes (exactly) one of $A_j/\bar{A}_j$

Subset Sum

Input: $x_1, x_2, \dots, x_n \in \mathbb{N}$

Goal: find a subset of x_i 's that sum to T
(decision version: decide if such a subset sum exists)

e.g. 14, 13, 17, 7, 18, 18, 10, 4, 11, 10
 $T = 52$



1. Recast 3SAT as "subset" problem
 $X = \text{literals } \{x_1, \bar{x}_1, x_2, \bar{x}_2, \dots, x_n, \bar{x}_n\}$
clause $(x_1 \vee \bar{x}_2 \vee x_3) = \text{set } \{x_1, \bar{x}_2, x_3\} \subset X$

Goal: find $S \subset X$ st.

Ⓐ for each clause C_i , $C_i \cap S = \emptyset$

Ⓑ for each x_j , S contains ^(exactly) 1 of $\{x_j, \bar{x}_j\}$

2. Start making #'s:

• we use mtn digits base 4

• for the literal x_j :

$A_j = \underbrace{01\dots1}_{m \text{ slots for each clause}} \underbrace{0\dots0}_{n \text{ slots for each variable}}$
(models x_j)

• for each clause C_i , set C_i th digit to 1 if $x_j \in C_i$, else 0

• set " x_j th digit" to 1, else 0

Similar for the literal $\bar{x}_j$

Theorem: A polytime algo for subset-sum implies a polytime algo for SAT

Subset Sum
 Input: $x_1, x_2, \dots, x_n \in \mathbb{N}$
 Goal: Find a subset of x_i 's that sum to T
 (decision version: decide if such a subset sum exists)
 e.g. 14, 13, 17, 7, 18, 18, 10, 4, 11, 10
 $T=52$

Literals $X = \{x_1, \bar{x}_1, \dots, x_n, \bar{x}_n\} \leftrightarrow$ Long #'s $\{A_1, \bar{A}_1, \dots, A_n, \bar{A}_n\}$
 and $\{B_1, \bar{B}_1, \dots, B_m, \bar{B}_m\}$

3-SAT instance

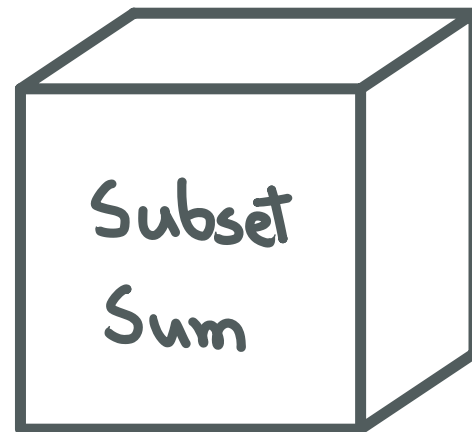
$$\mathcal{F}(x_1, \dots, x_n) = (x_1 \vee x_2 \vee x_3) \wedge \dots$$

$T = \underbrace{33 \sim 3}_{m \text{ clauses}} \underbrace{11 \sim 1}_{n \text{ variables}}$

$x_1, x_2, \dots, x_p \in \mathbb{N}$

$T \in \mathbb{N}$

Yes/No for
 $x_1, x_2, \dots, x_p \in \mathbb{N}$
 $T \in \mathbb{N}$



Key idea: numbers are bit-strings too!
 — use many digits
 — each digit encodes 1 constraint

Exercise 11.8. Let $x_1, \dots, x_n \in \mathbb{N}$. For each of the following problems, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that a polynomial time algorithm would imply a polynomial time algorithm for SAT.²

1. The *partition problem* asks if one can partition $x_1, \dots, x_n$ into two parts such that the sums of each part are equal.
2. The *3-partition problem* asks if one can partition $x_1, \dots, x_n$ into 3 parts such that the sums of each part are all equal.
3. The *any-k-partition problem* asks one can partition $x_1, \dots, x_n$ into k parts, for any integer $k \geq 2$, such that the sums of each part are all equal.
4. The *almost-partition* problem asks if one can partition $x_1, \dots, x_n$ into two parts such that the two sums of each part differ by at most 1.

²You can use the solution of one subproblem to solve another, as long as there's no circular dependencies overall.

5. ³Let n be even. The *perfect partition problem* asks if one can partition $x_1, \dots, x_n$ into two parts such that
 - (a) Each part has the same sum.
 - (b) Each part contains the same number of x_i 's.

Chapter 11

Subset-Sum, and the secret language of numbers

Consider the following problem, called *subset sum*. You are given as input n natural numbers $x_1, \dots, x_n \in \mathbb{N}$, and a target value $T \in \mathbb{N}$. The goal is to identify a subset of the numbers that sum to T :

$$x_{i_1} + x_{i_2} + \dots + x_{i_k} = T,$$

for distinct indices $i_1, i_2, \dots, i_k \in [n]$. We can also pose this as a decision problem where the goal is decide if there exists a subset that sums to T .

Subset sum is not an uncommon problem. We encounter it when we pay for groceries in cash and make exact change in quarters, dimes, nickels and pennies. It is also another example of a search problem. Given a candidate solution consisting of some numbers, we can easily verify that they are distinct numbers from the input, and that they add up to T .

11.1 Algorithms for subset sum

Let us design and analyze some algorithms for subset sum. We will focus on the decision version for simplicity but it will be easy to see how to modify our algorithms to return a feasible solution, if one exists.

A recursive algorithm. We will design a recursive algorithm. As always, we start with a recursive spec.

subset-sum($\{x_1, \dots, x_n\}, T$) = **true** if there is a subset of $\{x_1, \dots, x_n\}$ that sums to T , and otherwise **false**.

We implement the recursive spec as follows.

```

subset-sum( $\{x_1, \dots, x_n \in \mathbb{N}\}, T$ )
1. If  $T < 0$  then return false.
2. If  $T = 0$  then return true.           // The empty set sums to 0
3. If  $n = 0$  then return false.         // The empty set sums to 0
/* Otherwise try either excluding or including  $x_1$ . */
4. Return subset-sum( $x_2, \dots, x_n, T$ )  $\vee$  subset-sum( $x_2, \dots, x_n, T - x_1$ )
    
```

It is easy to see the algorithm is correct by induction on n ¹.

As per the running time, let $T(n)$ be the running time on a problem instance of n numbers. We have

$$T(0) = O(1)$$

$$T(n) = 2T(n-1) + O(1)$$

The recursion tree is a binary tree of height n , with 2^n leaves and $2^{n+1} - 1$ total nodes / subproblems. Each node / subproblem contributes $O(1)$ work, so we have

$$T(n) = O(2^n).$$

Dynamic programming. We can try to apply one of our favorite tricks: caching. In preparation for caching, let us index the arguments to our recursive function more compactly as follows. Fix an instance of subset sum, defined by $x_1, \dots, x_n \in \mathbb{N}$ and $T \in \mathbb{N}$. For $i, S \in \mathbb{N}$, we define

subset-sum(i, S) = **true** if there is a subset of $\{x_i, \dots, x_n\}$ that sums to S , and **false** otherwise.

The following pseudocode implements **subset-sum** based on the new definition above, and is just a more compact version of our previous pseudocode.

```

subset-sum( $i, S$ )
1. if  $S = 0$  then return true
2. if  $S < 0$  then return false
3. if  $i > n$  then return false
4. return subset-sum( $i + 1, S$ )  $\vee$  subset-sum( $i + 1, S - x_1$ ).
    
```

¹More precisely, it is easy to see that the pseudocode implements the specification for a given problem *assuming the specification holds inductively* for all subproblems.

To solve our initial problem, we call `subset-sum(1, T)`.

After adding caching / dynamic programming to the pseudocode, each problem takes only constant time (excluding recursive calls). Thus the total running time becomes

$$(\# \text{ subproblems}) \cdot \left(\begin{array}{c} \text{time per subproblem} \\ \text{excl. recursive calls} \end{array} \right) = nT \cdot O(1) = O(nT).$$

This running time looks much better than the previous, exponential running time of $O(2^n)$. (More on this in a moment.)

Constructing a subset sum. Our algorithm above only decides if a subset sum solution exists. Of course we may also be interested in the numbers that make up the sum. One approach to extracting the numbers is to use the decision algorithm as a black box; see exercise 11.2. (SAT was similarly *self-reducible*). A more efficient approach is as follows.

In the algorithm above, each subproblem `subset-sum(i, S)` that returns `true` came to that conclusion based on some immediate choice - namely, whether or not to include x_i , which leads to one of two subproblems that also returns `true`. Let us also record, in an additional table, the choice of subproblem that leads to `true` (when that is the case). Then, after solving the decision problem and building up this table, we have a trail of of subproblems, starting from `subset-sum(1, T)` and ending at a base case that returns `true`, that lists the actual set of numbers that add up to T .

A polynomial running time? Recall that our first goal for any problem is try to establish a running time that is a polynomial function of the input size. Do we now have a polynomial time algorithm for subset sum? $O(nT)$ might look like a polynomial, but in fact it is not a polynomial in the input size.

The catch hearkens back to our very first discussion about bits: the *bit complexity* of T is only $\log(T)$. That is, we only need $\log(T)$ bits to express T . Thus $O(nT) = O(n2^{\log(T)})$ is actually *exponential* in the input size of T . For example, if $T = 2^n$, then our $O(nT)$ running time is still exponential in n , while the input size of T is only $O(n)$. So for all the hoopla about dynamic programming, and the practical improvements for small T , we have not actually succeeded in obtaining a polynomial time algorithm.

11.2 Packing in the complexity

Subset sum doesn't look complicated, but it has resisted our attempts to obtain a (truly) efficient algorithm. Let us instead see if we can find good evidence that subset sum is actually *hard*; that is, unlikely to have a polynomial time algorithm.

How can I convince you that subset sum is hard? After all, it is a very simple looking problem. Quite probably all instances you've encountered in real life (namely, making change for cash) were very easy. And so what if we're stumped. We have overcome problems before. Surely, all we need is a good attitude, a hot cup of coffee, and copious scratch paper!

Recall the SAT problem, where we are given a 3-SAT formula and we want to find a satisfying assignment. Now *there's* a problem. Since we can reduce any general Boolean formula to a 3-SAT formula of roughly the same size, solving 3-SAT is tantamount to solving *anything* that can be expressed in Boolean logic. There is no shame in admitting that 3-SAT is too hard to expect a polynomial time algorithm. So what if I told you that

any algorithm for subset sum can be efficiently repurposed to solve 3-SAT.

That is, a polynomial time algorithm for subset-sum *would imply a polynomial time algorithm for 3-SAT*. That would be a satisfying explanation for our inability to find an efficient algorithm for subset sum: the problem is actually much more important than we realized.

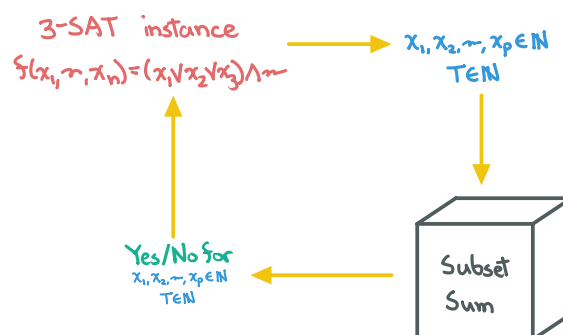
The main theorem of this section asserts precisely this, as follows.

Theorem 11.1. *If there is a polynomial time algorithm for subset sum, then there is a polynomial time algorithm for SAT.*

We have already seen that SAT and 3-SAT are polynomial time equivalent. To prove the theorem, we will describe a *polynomial time reduction* from 3-SAT to subset sum.

Given a 3-SAT formula $f(x_1, \dots, x_n)$, we will construct a subset sum problem that is feasible iff f is satisfiable. This might seem ridiculous at first. Subset sum uses only basic arithmetic, and arithmetic seems crude compared to Boolean logic.

The (simple yet profound) trick is to realize that *integers are bit strings too*: two numbers are equal iff their bit strings are equal. At a high level, we will create a subset sum problem where each number has a large number of digits. Each digit encode a different logical constraint. A feasible subset sum implies that every digit matches, which means that all logical constraints are met.



Consider an instance of 3-SAT given by a formula $f(x_1, \dots, x_n)$ with n variables and m disjunctive clauses, where each clause has three distinct variables, possibly negated. We let $C_1, \dots, C_m$ denote the m clauses, and treat them as subsets of the set of symbols $X = \{x_1, \bar{x}_1, \dots, x_n, \bar{x}_n\}$. For example, a clause $(x_1 \vee \bar{x}_2 \vee x_3)$ corresponds to the set $\{x_1, \bar{x}_2, x_3\}$. In this way, we can recast 3-SAT problem as a “subset” problem as follows:

Find a set $S \subset X$ that satisfies the following constraints.

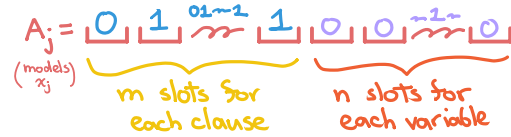
1. For each constraint $C_i \subset X$, $X \cap S \neq \emptyset$.
2. For each Boolean variable x_j , S contains exactly one of x_j or $\bar{x}_j$.

Above, the set $S \subset X$ indicates whether we assign **true** or **false** to each variable x_j . By item 2, for each j , S contains either x_j or $\bar{x}_j$. We interpret $x_j \in S$ as assigning $x_j = \text{true}$ and $\bar{x}_j \in S$ as assigning $x_j = \text{false}$. With this setup, item 1 is really saying that (the assignment corresponding to) S has to satisfy every clause.

As mentioned above, we will use numbers with a large number of digits and use each digit to encode some logic. In particular, we will create one digit for each constraint on S listed above. We will have 1 digit for each constraint C_i (corresponding to the constraints in item 1) and 1 digit for each variable x_j (corresponding to the constraints in item 2). Thus $m+n$ digits in all. We will also work with numbers in base 4 (without loss of generality). One can alternatively think of each $(m+n)$ -digit, base-4 number as an $(m+n)$ -length string composed of one of 4 letters; i.e., in $\{0, 1, 2, 3\}^{m+n}$.

We will create $2n + 2m$ numbers – one number for each x_j and each $\bar{x}_j$ in X , as well as $2m$ additional variables (two for each clause). For clarity, we will denote the numbers we create by capital letters $A, B, C, \dots$

For each x_j , we will create a number A_j as follows. Recall that we have reserved a digit for each constraint and a digit for each variable. We describe A_j in terms of the value of each digit (out of 4).



1. For each constraint C_i , if $x_j \in C_i$, we set the C_i th digit (i.e., the digit corresponding to C_i) of A_j to 1. Otherwise we set that digit to 0.
2. We set the digit corresponding to x_j to 1.
3. We set the remaining digits corresponding to other variables x_k ($k \neq j$) to 0.

For each negated symbol $\bar{x}_j \in X$ we create a number $\bar{A}_j$ similarly and with the obvious adjustments:

1. For each constraint C_i , if $x_j \in C_i$, we set the C_i th digit (i.e., the digit corresponding to C_i) of $\bar{A}_j$ to 1. Otherwise we set that digit to 0.
2. We set the digit corresponding to x_j to 1.
3. We set the remaining digits corresponding to other variables x_k ($k \neq j$) to 0.

This creates the first $2n$ numbers and we have promised to create $2m$ more. To motivate these remaining variables let us review what we have already created. We claim that

A set $S \subset X$ is a satisfying assignment iff, letting

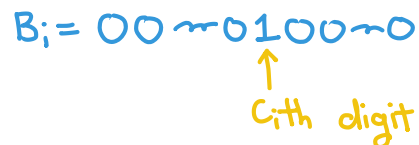
$$D = \sum_{x_j \in S} A_j + \sum_{\bar{x}_j \in S} \bar{A}_j$$

denote the sum of numbers corresponding to symbols in S , we have the following:

1. *For every variable x_j , the x_j th digit of D is exactly 1.*
2. *For every constraint C_i , the C_i th digit of D is at least 1.*

This is not quite a subset sum problem, but I think it is fair to say we are getting closer. We have two sets of constraints on D . The first family, corresponding to the variables x_j , is naturally suitable for subset sum, as it specifies *exactly* what the numbers have to add up to for each of each digits. However, the second set of constraints, for the digits corresponding to D , are based on inequalities rather than equalities. The issue is that, in 3-SAT, we could satisfy a clause C_i by taking one, two, or all three of the participating variables. Thus, for the corresponding digit, the values $\{1, 2, 3\}$ should all be allowed. How do we express the fact that we allow for a *range* of digits, in the native language of subset sum?

The following adjustment effectively implements *padding*. For each constraint C_i , we create two copies of the same number, called B_i . B_i has value 1 in the C_i th digit and value 0 in all the other digits. The role of B_i is simply to allow us to artificially increase the C_i th digit of D . In particular, with two copies of B_i , we can add up to two units to the C_i th digit of D . Values of 1 or 2 can easily be converted to 3. Crucially, a value of 0 cannot be increased to 3.

$B_i = 00 \sim 0100 \sim 0$


We now select our target value T . For each clause C_i , we set the C_i th digit of T to 3. For each variable x_j , we set the x_j th digit of T to 1.

$T = 33 \sim 3 \quad 11 \sim 1$
 $\underbrace{\hspace{1.5cm}}_{m \text{ clauses}} \quad \underbrace{\hspace{1.5cm}}_{n \text{ variables}}$

Observe that the size of our subset sum problem, $O((m+n)^2)$, is polynomial in the size of the 3SAT formula. It is also easy to see that our construction takes polynomial time (and maybe even linear time, with care).

In conclusion, our construction satisfies the following claim, which establishes theorem 11.1.

The Boolean formula f is satisfiable iff there is a subset of the A_j 's, $\bar{A}_j$'s, and (2 copies each of the) B_i 's (as defined above) that adds up to T .

11.3 Perfect is the enemy of good

Let us pause and reflect on our developments. We started with a simple and common problem: subset sum. We have some brute force algorithms, which when combined with caching (dynamic programming), leads to a reasonable running time for small target values. But this running time was not strongly polynomial; for large numbers, the algorithm can be very slow. The basic reason our algorithm did not qualify as polynomial time is that a very large integer x requires only $\lceil \log x \rceil$ many bits to represent it, and ultimately we seek running times that are polynomial in the input size. A $O(x)$ running time is *exponential* in the input size of x .

Our reduction from 3SAT to subset sum suggests that subset sum is much harder to solve than one might have expected. Moreover, we see explicitly in the reduction how the many bits of large numbers can be used to encode the rich complexity of a 3SAT problem.

Let us also point out that the difficulty of subset sum lies in our insistence on an *exact* answer to the problem. By contrast, suppose we had an algorithm that satisfies the following approximation guarantee:

If the subset sum is feasible, then return a subset $x_{i_1}, \dots, x_{i_k}$ whose sum lies in the range $[.99T, 1.01T]$. If the subset sum problem is infeasible, then either return a subset sum in the range $[.99T, 1.01T]$, or output that it is infeasible.

For many real applications, where the sums represent a measurable quantity, being off by a little is not the end of the world. On the other hand, since an approximate sum in the above sense does not imply an exact solution, it does not risk deciding a SAT problem in disguise. This gives hope for a polynomial time approximation algorithm.

Indeed, there *are* polynomial time approximation algorithms that satisfy the above guarantee, even if we replace $[\.99T, 1.01T]$ with $[(1 - \epsilon)T, (1 + \epsilon)T]$ for any fixed error parameter $\epsilon \in (0, 1)$. (See exercise 11.5.) We may discuss approximation algorithms later, but for the moment, it is good to be aware of their existence and motivation.

The other side of the coin - where the hardness of subset sum is *useful* - is cryptography. Loosely speaking, we can interpret the solution to a subset sum problem as a password - it is hard to guess (as far as we know), but easy to verify. Many other problems have this characteristic (as we will see) but subset sum is particularly simple, which has some computational advantages. The Merkle-Hellman system, one of the first public-key cryptographic systems, is based on subset sum, and in particular relies on the assumption that subset sum is hard. Lattice based cryptography, a more modern cryptographic technique, is (at a high level) based on the difficulty of multi-dimensional extensions of subset sum. See (e.g.) the notes [Pei13] for more details.

11.4 Additional notes and references

This lecture overlaps with [Eri19, §3.8] and [KT06, §6.4 and §8.8].

Spring 2024 lecture notes. Click on the links below for the following files:

- [Handwritten notes annotated during the presentation.](#)
- [Recorded video lecture.](#)

Spring 2023 lecture materials. Click on the links below for the following files:

- [Handwritten notes prepared before the lecture.](#)
- [Handwritten notes annotated during the presentation.](#)
- [Recorded video lecture.](#)

Spring 2022 lecture materials. Click on the links below for the following files:

- [Handwritten notes prepared before the lecture.](#)
- [Handwritten notes annotated during the presentation.](#)
- [Recorded video lecture.](#)

11.5 Exercises

Exercise 11.1. Recall the *1-in-3-SAT* problem from exercise 9.11. Suppose instead that we had reduced from 1-in-3-SAT to subset sum. How might this have simplified our construction in section 11.2?

Exercise 11.2. At the end of section 11.1, we discussed how to use an additional table to also extract the subset sum solution. Here we develop an alternative approach

that was briefly alluded to in section 11.1. It is not as efficient as building a secondary table, but the technique is more general and of independent interest.

Suppose one had a black box algorithm for deciding subset sum problems. That is, given an instance of subset sum described by n integers $x_1, \dots, x_n \in \mathbb{N}$ and an additional target value T , the algorithm returns **true** if there exists a feasible solution, and otherwise **false**. Show that one can use this decision algorithm as a subroutine to obtain an efficient constructive algorithm for subset sum.

Exercise 11.3. Consider the following variation of subset sum which allows for an additive error of 1. We'll call it *subset-sum ± 1* . The input consists of n numbers $x_1, \dots, x_n \in \mathbb{N}$ and a target $T \in \mathbb{N}$, like subset sum. The goal is to decide if there is a subset of $x_1, \dots, x_n$ that sums to either $T - 1$, T , or $T + 1$. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that a polynomial time algorithm would imply a polynomial time algorithm for SAT.

Exercise 11.4. Consider the following variation of subset-sum called *Kobe subset sum*. The input is similar to normal subset sum, with n numbers $x_1, \dots, x_n \in \mathbb{N}$ and $T \in \mathbb{N}$, except we are promised that each number x_i lies in the range $\{8, \dots, 24\}$. For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that a polynomial time algorithm would imply a polynomial time algorithm for SAT.

Exercise 11.5. (Approximating subset sum.) Let $\epsilon \in (0, 1)$ be fixed. Here we treat ϵ as a fixed constant (like $\epsilon = .1$, for 10% error); in particular, running times of the form $O(n^{O(1/\epsilon)})$ count as a polynomial.

A $(1 \pm \epsilon)$ -approximation algorithm for subset sum is one that (correctly) either:

1. Returns a subset whose sum lies in the range $[(1 - \epsilon)T, (1 + \epsilon)T]$.
2. Declares that there is no subset that sums to (exactly) T .

Note that such an algorithm does not solve the (exact) subset sum problem.

1. Suppose every input number x_i was “small”, in the sense that $x_i \leq \epsilon T$. Give a polynomial time $(1 \pm \epsilon)$ -approximation algorithm for this setting.
2. Suppose every input number x_i was “big”, in the sense that $x_i > \epsilon T$. Give a polynomial time $(1 \pm \epsilon)$ -approximation algorithm for this setting.

3. Now give a polynomial time $(1 \pm \epsilon)$ -approximation algorithm for subset sum in the general setting (with both big and small inputs).

Exercise 11.6. Let $A[1..m]$ be an array of integers, and let T be an additional number. We say that a sequence of numbers

$$y_1, y_2, \dots, y_k$$

is *weakly increasing* if $y_1 \leq y_2 \leq \dots \leq y_k$. The problem is to find the length of the longest increasing subsequence of A where the numbers sum to T . For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that a polynomial time algorithm would imply a polynomial time algorithm for SAT.

Exercise 11.7. Consider the following variation of subset sum. The input consists of n (positive) integers $x_1, \dots, x_n \in \mathbb{N}$, an integer $T \in \mathbb{N}$, and an integer $k \in \{1, \dots, n\}$. The problem is to decide if there is a subset of $x_1, \dots, x_n$ of cardinality k that sums to T . For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that a polynomial time algorithm would imply a polynomial time algorithm for SAT.

Exercise 11.8. Let $x_1, \dots, x_n \in \mathbb{N}$. For each of the following problems, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that a polynomial time algorithm would imply a polynomial time algorithm for SAT.²

1. The *partition problem* asks if one can partition $x_1, \dots, x_n$ into two parts such that the sums of each part are equal.
2. The *3-partition problem* asks if one can partition $x_1, \dots, x_n$ into 3 parts such that the sums of each part are all equal.
3. The *any- k -partition problem* asks one can partition $x_1, \dots, x_n$ into k parts, for any integer $k \geq 2$, such that the sums of each part are all equal.
4. The *almost-partition* problem asks if one can partition $x_1, \dots, x_n$ into two parts such that the two sums of each part differ by at most 1.

²You can use the solution of one subproblem to solve another, as long as there's no circular dependencies overall.

5. ³Let n be even. The *perfect partition problem* asks if one can partition $x_1, \dots, x_n$ into two parts such that
- (a) Each part has the same sum.
 - (b) Each part contains the same number of x_i 's.

Exercise 11.9. Here we consider a vast generalization of edit distance, as follows. Suppose the cost of an edit sequence is given by a nonnegative real-valued function $f(a, b, c) \geq 0$, where a is the number of insertions, b is the number of deletions, and c is the number of substitutions. We assume that the function is monotonically increasing in each of its arguments. That is, increasing a , b , or c increases the value of $f(a, b, c)$.

Assume that f is provided as a subroutine that takes $O(1)$ time to evaluate. Consider the problem the minimum cost edit sequence w/r/t the cost function f . For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that a polynomial time algorithm would imply a polynomial time algorithm for SAT.

Exercise 11.10. According to a report from the World Shipping Council (WSC), 661 shipping containers (a.k.a. crates) were lost overboard in the year 2022. This is actually an improvement, as an average of 1,566 containers were lost at sea each year in the fifteen year period from 2008 to 2022. (Cf. <https://www.worldshipping.org/news/world-shipping-council-releases-containers-lost-at-sea-report-2023-update>)

³IMO, this one is the trickiest.

Now suppose you are running a crate shipping company. To try to minimize the number of crates that fall off the boat, you have decided to impose a strict weight limit $L \in \mathbb{N}$ on the total weight of shipping crates on any boat. Given a collection of n crates of varying weight $W_1, \dots, W_n \in \mathbb{N}$, your goal is to ship all n crates with the minimum number of boats.

More formally, given the weight limit L and the crate weights $W_1, \dots, W_n$, the goal is to assign each crate to one of $k \in \mathbb{N}$ boats, for k as small as possible, so that the total weight of containers assigned to any boat is at most L . (To be clear, k is not part of the input; your job includes finding the minimum possible number of boats k . You can also assume that $L \geq W_i$ for all i .) For this problem, either (a) design and analyze a polynomial time algorithm (the faster the better), or (b) prove that a polynomial time algorithm would imply a polynomial time algorithm for SAT.



Sample solution.

Problem. Recall the subset sum problem, where the input consists of n positive integers $x_1, \dots, x_n \in \mathbb{N}$, and a target value $T \in \mathbb{N}$. The goal is to decide if there is a subset of $x_1, \dots, x_n$ that sums to T . Design and analyze a $O(nT)$ -time algorithm for the subset-sum problem.

Solution.

1. *Recursive spec / induction hypothesis.* For all integers $S \leq T$ and $i \in \mathbb{Z}_{\geq 0}$, we define

$\text{SS}(i, S) = \text{true}$ if there is a subset of $x_i, \dots, x_n$ that sums to S , and **false** otherwise.

2. *Solving the original problem.* The solution to the original problem is given by $\text{SS}(1, T)$.

3. *Recursive implementation.* $\text{SS}(i, S)$

1. If $S = 0$ then return **true**.

2. If $S < 0$ then return **false**.

3. If $i \leq 0$ then return **false**.

/ Otherwise $S > 0$ and $i \leq n$.*

**/*

4. Return the $\vee$ (or) of the following:

A. $\text{SS}(i + 1, S)$

B. $\text{SS}(i + 1, S - x_i)$.

4. *Running time with caching / dynamic programming.* After caching the (non-base case) solutions for the recursive implementation (i.e., applying dynamic programming), the running time is as follows. We have $O(nT)$ subproblems broken down as $O(n)$ choices for i and $O(T)$ choices for S , and each takes subproblem takes constant time (excluding recursive sub-calls). Overall the running time is $O(nT)$.

5. *Proof.* The base cases are pretty clear: if $S = 0$, then the empty set sum to S ; otherwise, if $S < 0$ then no set of (positive) integers can sum to S ; otherwise if $S > 0$ but $i > n$ then there are no numbers left to sum to S .

Consider $\text{SS}(i, S)$ where $i \leq n$ and $S > 0$. If there is a feasible solution, then it either includes x_i or it doesn't. If it doesn't, then by induction on i , $\text{SS}(i + 1, S)$ will return **true**. If it does, then by induction on i , $\text{SS}(i + 1, S - x_i)$ will return **false**.

If there is no feasible solution, then there is no feasible solution to the subset sum problems described by $\text{SS}(i + 1, S)$ or $\text{SS}(i + 1, S - x_i)$. By induction on i both sub-calls will return **false**, hence so will $\text{SS}(i, S)$, as desired.