アルゴリズム特論（第２回）

北海道大学　大学院　情報科学研究科
アルゴリズム研究室　　湊　真一

アルゴリズム特論 2006年度講義予定

- 前半7回(担当:湊助教授)
  - 論理関数と組合せ最適化
  - 大規模論理データ処理アルゴリズム
- 後半7回(担当:トーマス教授)
  - 計算論的学習アルゴリズムの基礎
  - 言語学習とそのアルゴリズム

4/11	M-1	
4/18	M-2	
4/25	M-3	
5/02	M-4	
5/09		T-1
5/16	M-5	
5/23	M-6	
5/30	M-7	
6/06	(休)	
6/13		T-2
6/20		T-3
6/27		T-4
7/04		T-5
7/11		T-6
7/18		T-7
7/25	(休)	

前半の講義内容

- 第１回(4/11)　論理関数の処理とデータ表現
- 第２回(4/18)　積和形論理式
- 第３回(4/25)　積和形論理式の簡単化
- 第４回(5/02)　二分決定グラフ(BDD)
- 第５回(5/16)　BDD処理系の実装技術
- 第６回(5/23)　BDDの変数順序づけ
- 第７回(5/30)　BDDの応用

講義ノートURL：
http://www-alg.ist.hokudai.ac.jp/~minato/class-j.html

前回（4月11日）の内容

論理関数の処理とデータ表現

- 論理関数とは
  - ２値と多値、論理式と論理関数、命題論理と述語論理
- 論理関数の処理とは
  - 論理式/回路からの論理関数データ生成、各種論理演算
  - 恒真/恒偽判定、等価性判定、包含性判定
  - 解探索、最小化
- 論理関数の表現法
  - 求められる性質
  - 関数の総数・情報理論的ビット量・完全ランダム関数
  - カルノー図、真理値表、積和形、BDD

今週の内容：　積和形論理式

- 積和形論理式の記述例
  - 実例
  - テーブル表現、論理式表現、CNF/DNF
  - 真理値表との比較（非固定的表現）
  - 論理関数に対する非一意性
- 計算機上での実装
  - 配列表現、リスト表現
- 論理演算アルゴリズム
  - OR, AND, NOT, EXOR
  - 冗長項除去、等価性判定、恒真性判定
- 積和形論理式のデータ量
  - コンパクトな例、不利な例
  - 現実的に扱える範囲

計算機上での論理関数表現の要求条件

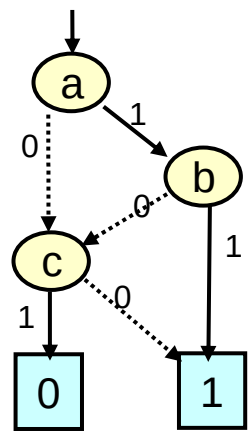
- **表現がコンパクトであること**
  - n入力の論理関数は2^{2^n}通り存在。
    → 固定長データで識別するには
    少なくとも2^nビットは必要。(例:真理値表)
  - 現実には全ての論理関数が均等に現れる訳ではない。
    → よく現れる関数がコンパクトに表現できる
    可変長データが使われる。
- **論理演算処理が高速に行えること**
  - 2つの論理関数データの等価性判定が高速に行えるか
  - AND, OR, NOT等の論理演算が効率よく実行できるか
- 人間が見やすいことはあまり重要でない。

代表的な論理関数データ表現法

- 真理値表
  - 論理関数によらず固定サイズ
  - 完全一様ランダム関数であれば理論上最小の表現
- 積和形論理式
  - 解集合を列挙する形式
  - 人間の直感に合い、比較的コンパクト
- BDD（二分決定グラフ）
  - Yes/Noの決定グラフ
  - 表現の一意性、コンパクト性、高速処理を併せ持つ

$x_1x_2x_3 \ldots x_n$	F_1	F_2	F_3
000 … 0	1	0	1
100 … 0	0	1	0
010 … 0	1	1	0
110 … 0	0	0	0
001 … 0	1	1	0
101 … 0	0	1	0
011 … 0	1	1	0
⋮	⋮	⋮	⋮
111 … 1	1	0	0

F = a b + c + ~a c ~d + ~b + …



一般的な論理式

- 論理変数（または論理定数）と論理演算子の組合せ

（例） F = (a + b) (~c + d) + b (c + ~d) + ~ (a c)

論理式は論理関数データの表現方法の1つであるが
あまりに自由すぎて、データ処理がしにくい

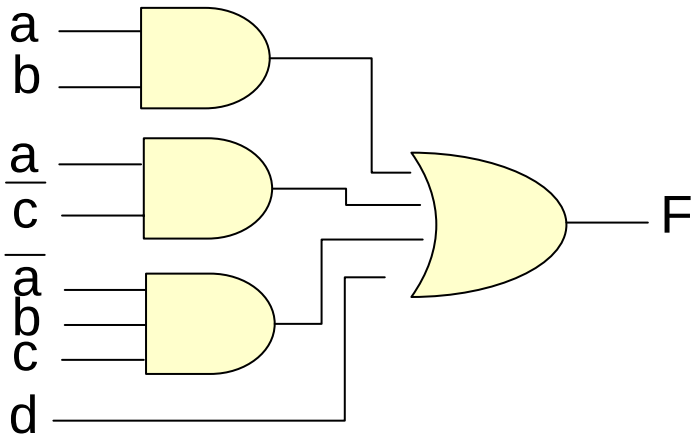
恒真/恒偽判定、等価性判定、包含性判定
充足解探索（任意解探索/最適化探索）

論理式の形に制約を加えて、処理しやすくした論理式表現
（の1つ）が**積和形論理式**(SOP: Sum-Of-Products Form)

（例） F = a b + a ~c + a d + b ~c + b d

積和形論理式

$ab + a\bar{c} + \bar{a}bc + d$



a	b	c	d	F
1	1	–	–	1
1	–	0	–	1
0	1	1	–	1
–	–	–	1	1

- 肯定･否定の2種類の入力変数を使用。正/負のリテラル(Literal)とも呼ぶ。
- 1個以上の異なるリテラルの組合せ(AND結合)で1つの積項(product termまたはCube)を構成。
- 1個以上の異なる積項の組合せ(OR結合)で1つの積和形論理式(Sum-of-ProductsまたはCube set)を構成。
- 否定演算は変数にだけ使える。

積和形と和積形

- 積和形と和積形は双対な表現形式
  - 積和形 (SOP: Sum-Of-Products)
    DNF: Disjunctive Normal Form
  - 和積形 (POS: Product-Of-Sums)
    CNF: Conjunctive Normal Form

（例） F = a b + a ~c + a d + b ~c + b d

~F = ~(a b + a ~c + a d + b ~c + b d)

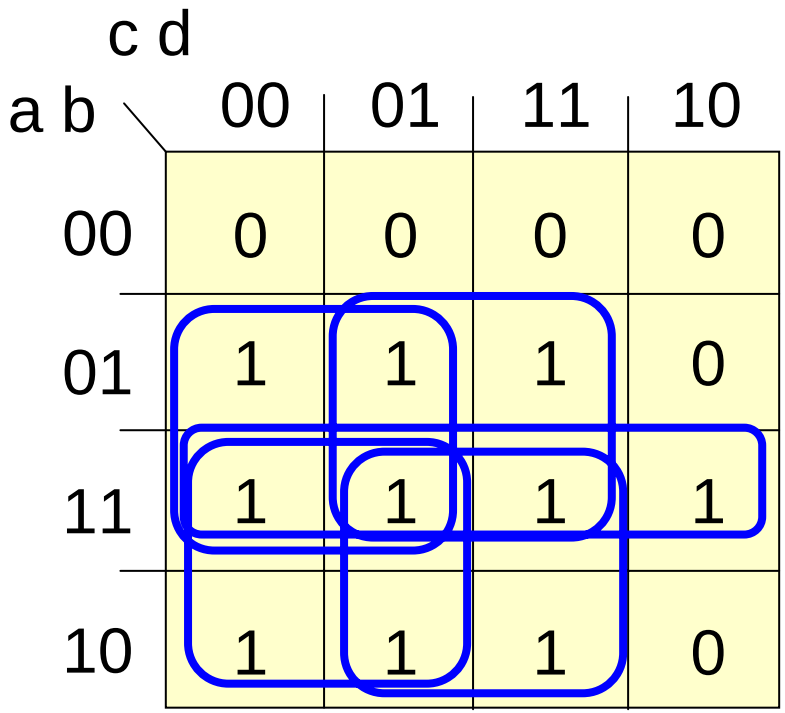
= ~(a b) ~(a ~c) ~(a d) ~(b ~c) ~(b d)

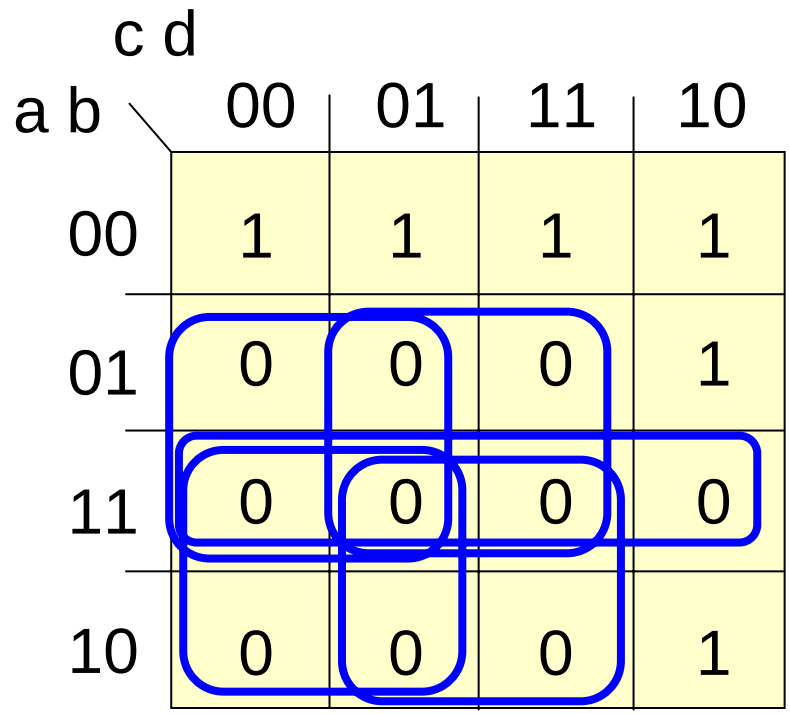
=(~a+~b)(~a+c)(~a+~d)(~b+c)(~b+~d)

積和形と和積形のどちらか一方を効率よく処理できれば
もう一方は0,1の意味を反転させれば同様にできる。

→ 本講義では主に積和形表現を用いて説明する。

積和形と和積形





F = a b + a ~c + a d + b ~c + b d

~F = ~(a b + a ~c + a d + b ~c + b d)
= ~(a b) ~(a ~c) ~(a d) ~(b ~c) ~(b d)
=(~a+~b)(~a+c)(~a+~d)(~b+c)(~b+~d)

真理値表表現との比較

- 真理値表では、論理関数がどんなに単純でも
  変数の個数に対して指数サイズの記憶量となるが、
  積和形では、単純な論理関数は比較的コンパクトに表現できる

（例） 変数が a,b,c,…z の26個あった場合：

	真理値表	積和形
“1”（恒真）、“0”（恒偽）:	2^{26}(=67108864)	定数サイズ
F = a + b + c + … + z :	2^{26}	リニア
G = a b c … z :	2^{26}	リニア

積和形の非一意性

- 積和形論理式は一般の論理式よりも制約が加えられているが、それでもまだ自由度がある。

(例) F1 = x ~y + x z + ~x y + ~x ~z
F2 = x ~y + ~x y + y z + ~y ~z
F3 = x ~y + ~x ~z + y z

同じ論理関数が異なる積和形で表現できる場合がある

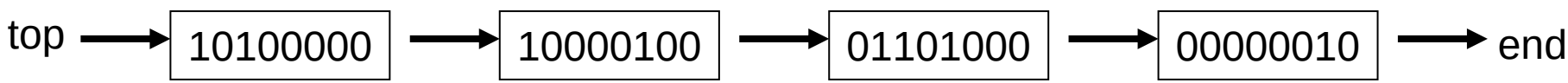
→ 論理の等価性判定が必要となる。
（残念ながら、あまり簡単ではない。
（最悪の場合、指数時間かかる）

積和形データの計算機上での表現方法

- 文字列(テキスト)表現
  - 人間が読み書きしやすいが、計算機での処理は面倒
  - 計算結果をファイルに保存するときにはよく用いられる
- テーブル表現
  - 横軸にリテラル、縦軸に積項の二次元配列で表現
  - 最大積項数があらかじめわかっていないと使いにくい

a	b	c	d	F
1	1	−	−	1
1	−	0	−	1
0	1	1	−	1
−	−	−	1	1

- 積項のリスト表現
  - 1つの積項をビット列で表し、積項の直列リストで表現
  - 使用する変数(リテラル)の数はあらかじめ宣言
    積項数は演算結果に応じて動的に確保

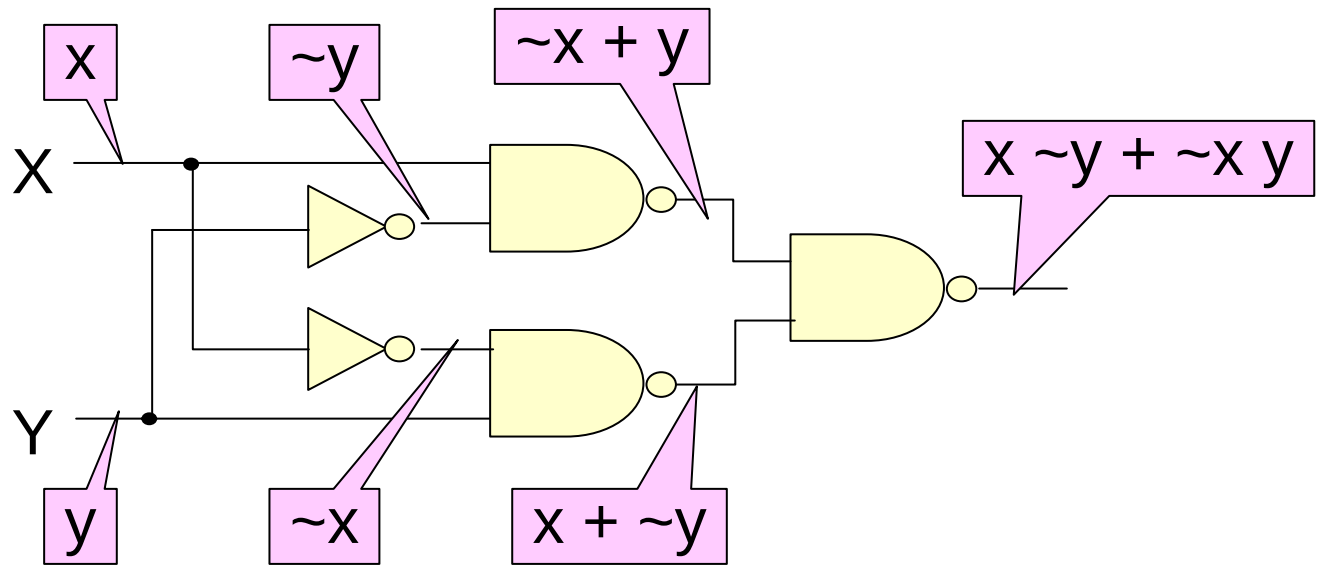


積項集合としての表現の一意性

- 積和形論理式は積項（リテラルの組合せ）の集合
  - 積項の並べ方の規則を決めておけば、同じ集合データは一通りに表現できる
  - 並び方を決めておくと何かと都合が良い
  - このような表現を一般的に標準形(canonical form)と呼ぶ
- 積和形論理式の場合：
  - 変数名に全順序をつける。（変数名とその順位の対応表を1つ作る）
  - 積項内のビット列は変数順位に従う
  - 積和形論理式に含まれる積項は辞書順にソート
  - 負リテラルは濁点のように扱う （a, ~a, b, ~b, c, ~c, …)
- 集合表現としての一意性と、論理関数としての一意性は意味が違うので注意
  - FとGが集合表現として等価なら、論理関数としても等価（逆は必ずしも成り立たない）

積和形データの論理演算処理

- 2つの積和形論理式 F, G から
  論理演算結果 H (= F [op] G)の積和形を作る処理



論理和(OR)演算

- 2つの積項集合の和　(H = F + G)
  - 基本的には2つの集合を併合する処理
  - リスト表現の場合、FとGを保存しておく必要がなければ、ポインタを1ヶ所張り替えるだけでHが得られる（標準形を保つには、積項をソートする必要がある）
  - F,Gに同じ積項が含まれていれば、重複を削除
  - F,Gがともにソートされているならば、順序を保ったままで併合する方法が効率的。計算時間はFとGの長さの和。

（例）　F = a b c + ~a ~b + ~c d
G = a ~c + ~b + ~c d
H = a b c + a ~c + ~a ~b + ~b + ~c d

- 論理和を計算した結果、集合としては標準形だが、論理としては冗長な表現になる場合がある
  - 最終的には論理式の簡単化・最小化処理が必要

（例）　F = a + b , G = ~a
H = a + ~a + b
= 1

論理積(AND)演算

- 2つの積項集合の直積　(H = F × G)
  - まず、Fが単純リテラル式 x(または~x)であるとき:
    Gの各積項のうち、x(~x)を含む項はそのまま残す。
    ~x(x)を含む項は削除。x,~xを含まない項はx(~x)を付けて残す。
  - Fが単項式 x1 x2 … xn であるとき:
    H = x1(x2(… (xn G)…))
    のように単純リテラル式の処理を繰り返し適用する。
  - Fが一般の多項式 (T1+T2+…+Tn)であるとき:
    H = T1 G + T2 G + … + Tn G
    のように単項式の処理と論理和処理を繰り返す。
- データ量は最悪の場合、FとGのデータ量の積になる。
  計算時間もデータ量に比例。

（例） F = a b c + ~a ~b + ~c d
G = a ~c + ~b + ~c d
H = a b c G + ~a ~b G + ~c d G
= 0 + (~a ~b + ~a ~b ~c d) + (a ~c d + ~b ~c d + ~c d)

論理和の場合と同様に、演算結果が冗長な論理になる場合がある

論理否定(NOT)演算

- 積項集合の論理否定　(H = ~ F)
  - 集合の否定演算（補集合）とはまったく違う処理
  - Fの各積項の否定を作り、それら全ての積を取る

（例）F = a b c + ~a ~b + ~c d

H = ~(a b c + ~a ~b + ~c d)

= ~(a b c) ~(~a ~b) ~(~c d)

= (~a + ~b + ~c) (a + b) (c + ~d)

= (~a + ~b + ~c) (a c + a ~d + b c + b ~d)

= ~a b c + ~a b ~d + a ~b c + a ~b ~d + a ~c ~d + b ~c ~d

- 最悪、Fの長さの指数のデータ量と計算時間になる。（→　指数爆発）

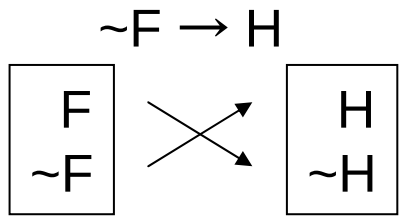
（例）アキレス腱関数

F = x1 x2 x3 + x4 x5 x6 + x7 x8 x9 + … + x_{3n} x_{3n+1} x_{3n+2}

は、~Fの積和形が指数関数的に増大する

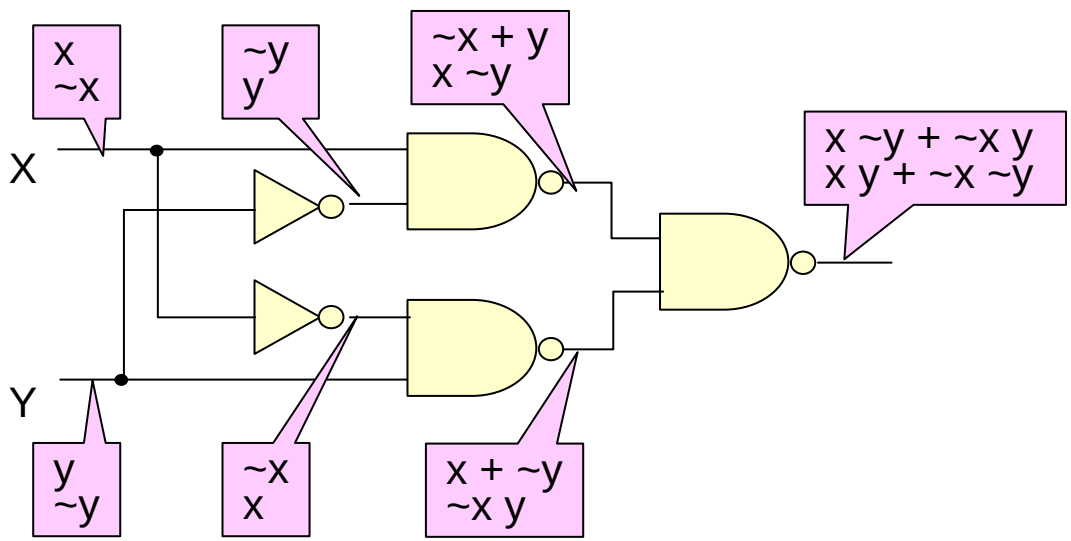
否定演算の高速化の工夫

- 積和形の処理では否定演算が最も手間がかかる
- 最初からFと~Fを両方用意しておけば、否定演算はFと~Fを交換するだけで実現できる。
- ANDとOR演算はそれぞれ双対な演算を適用する。
  - 否定演算処理を回避できる。（そのかわりデータ量が増える）
  - 最悪の場合のデータ量・計算時間は変わらない



H = F + G
~H = ~F × ~G

H = F × G
~H = ~F + ~G



排他的論理和(EXOR)演算

- 積和形同士の排他的論理和(EXOR)
  を効率よく直接実行するアルゴリズムは知られていない。
  - H = F ~G + ~F G　のように、
    AND, OR, NOT演算に分解して計算する。
- データ量は最悪の場合、FとGのデータ量の積になる。
  計算時間もデータ量に比例。

冗長項の除去アルゴリズム

- 積和形論理式の処理では、演算の途中で冗長項が発生しやすい。一旦発生するとますます増える。
  - ときどき、冗長項を除去する必要がある。
  - 完全に除去するには積和形の最小化・簡単化アルゴリズム（次回の講義）を適用するが、非常に時間がかかる。
    - 式の長さの指数またはそれ以上の処理時間
  - 簡単に見つかる冗長項だけを除去する方法はいくつかある。
    - 簡単化規則を繰り返し適用する
    - 式の長さの2乗程度の処理時間
    - 必ずしも最小形にならない

（簡単化規則の例）

$$F + F = F$$
$$F + \sim F = 1$$
$$x\,x = x$$
$$x\,\sim x = 0$$
$$x + \sim x\,F = x + F$$
$$x\,F + \sim x = F + \sim x$$

恒真・恒偽判定

- 恒偽判定は簡単。
  - 意味のある積項が1つでもあるかどうか調べればよい
- 恒真判定は非常に難しい。co-NP完全問題
  - 式の長さの指数の人数で手分けして解いたときに、多項式の時間がかかる問題
    - 手分けせずに1人で解いた場合には、最悪の場合、指数の時間がかかる。（証明されていないが信じられている）
  - 基本的には、変数に順に0,1を割り当てて、すべての組合せを試していくアルゴリズムしか知られていない。
    - 運がよい場合には、比較的高速に判定できる場合もある。（0になる例が1つでもみつかれば恒真でないことがわかる）
- 積和形（または和積形）の恒真（恒偽）判定アルゴリズムだけで、1つの研究分野となっている
  - 国際会議が毎年開かれている

等価性・包含性判定

- 等価性判定
  $(F \equiv G) \Leftrightarrow (F\,G + \sim F\,\sim G \equiv 1)$
  論理演算を実行して恒真判定
- 包含性判定　$(F \Rightarrow G) \Leftrightarrow (\sim F + G \equiv 1)$
  論理演算を実行して恒真判定

- いずれも最悪の場合は、式の長さの指数関数的なデータ量と処理時間がかかる。

充足解探索

- 積和形論理式の各積項は論理を充足する解の組合せを表している。
  - ただし、積項に出てくる組合せ以外にも充足解はあるかも知れない
- コスト最小の充足解の探索は簡単ではない。
  - 積和形が簡約化・最小化されていれば、冗長なリテラルは除去されているので、コストの小さい組合せを見つけやすい
  - 厳密にコスト最小の充足解を得るには、
    各変数に0,1の組合せを入れて調べる必要がある
    （式の長さの指数の処理時間）

積和形論理式のデータ量と処理時間

- 処理時間はデータ量に依存
  - 演算の種類によって、データ量の和または積に比例
  - データ量の積に比例する演算を繰り返すと、最悪の場合、指数関数的にデータ量が増大する
- 変数の個数が非常に多くても、１つの論理式に同時に出現する変数が少なければ、効率よく表現できる。
  - 真理値表の場合は常に最悪ケース
- 否定演算・排他的論理和(EXOR)演算が苦手
  - ２進数の算術演算（加算・乗算等）は苦手

現実的に扱える規模

- 総リテラル数（各積項のリテラル数の総和）が、処理時間の尺度になる。
  - 総リテラル数は処理中に動的に増えたり減ったりする。
  - 最終結果は小さくても、計算の途中で爆発的に増大している場合もある。
- 以下は経験的な目安。（例外もある）
  - 総リテラル数が数10～数100であれば、近頃のPCを使えば、たいていのデータ処理は待たずに終了する。
  - 総リテラル数が1,000を超えると、処理内容によっては、数時間～数週間以上かかる場合がある。
  - 総リテラル数が100,000を超えると、多くの場合、実用的な時間では処理が終わらないことが多い。
    （問題をもっと単純化するなど、アルゴリズムを根本的に考え直した方がよい）