Julius-Maximilians-
UNIVERSITÄT
WÜRZBURG

Lehrstuhl für
INFORMATIK I
Algorithmen & Komplexität

Institut für Informatik

Exact Algorithms

Sommer Term 2020

Lecture 2. Branching Algorithms and Satisfiability

Based on: [Exact Exponential Algorithms: §2]
Further discussions: [Parameterized Algorithms: §3; specifically §3.2]

(slides by J. Spoerhase, Th. van Dijk, S. Chaplick, and A. Wolff)

Alexander Wolff
Lehrstuhl für Informatik I

Problems in $\mathcal{NP}$

- Brute force?

Problems in $\mathcal{NP}$

- Brute force?
- $\forall$ $\textsc{Yes}$-instances : $\exists$ certificate

Problems in $\mathcal{NP}$

- Brute force?
- $\forall$ $\textsc{Yes}$-instances : $\exists$ certificate

Problems in $\mathcal{NP}$

- Brute force?
- $\forall$ Yes-instances : $\exists$ certificate
  - polynomial size

Problems in $\mathcal{NP}$

- Brute force?
- $\forall$ Yes-instances : $\exists$ certificate
  - polynomial size
  - polynomial-time verifiable

Problems in $\mathcal{NP}$

- Brute force?
- $\forall$ Yes-instances : $\exists$ certificate
  - polynomial size
  - polynomial-time verifiable
- Runtime?

Problem Types

Subset Problems

- Solutions: subsets of a given ground set

Problem Types

Subset Problems

- Solutions: subsets of a given ground set
- e.g., independent set in a graph, satisfying assignment to a Boolean formula

Problem Types

Subset Problems

- Solutions: subsets of a given ground set
- e.g., independent set in a graph, satisfying assignment to a Boolean formula
- Brute-force in $O^*(2^n)$ time

Problem Types

Subset Problems

- Solutions: subsets of a given ground set
- e.g., independent set in a graph, satisfying assignment to a Boolean formula
- Brute-force in $O^*(2^n)$ time

Permutation Problems

- Solutions: permutations of a given ground set

Problem Types

Subset Problems

- Solutions: subsets of a given ground set
- e.g., independent set in a graph, satisfying assignment to a Boolean formula
- Brute-force in $O^*(2^n)$ time

Permutation Problems

- Solutions: permutations of a given ground set
- e.g., Hamilton path of a graph, or tour as in TSP

Problem Types

Subset Problems

- Solutions: subsets of a given ground set
- e.g., independent set in a graph, satisfying assignment to a Boolean formula
- Brute-force in $O^*(2^n)$ time

Permutation Problems

- Solutions: permutations of a given ground set
- e.g., Hamilton path of a graph, or tour as in TSP
- Brute-force in $O^*(n!) = 2^{O(n \log n)}$ time

Problem Types

Subset Problems

- Solutions: subsets of a given ground set
- e.g., independent set in a graph, satisfying assignment to a Boolean formula
- Brute-force in $O^*(2^n)$ time

Permutation Problems

- Solutions: permutations of a given ground set
- e.g., Hamilton path of a graph, or tour as in TSP
- Brute-force in $O^*(n!) = 2^{O(n \log n)}$ time

Partitioning Problems

- Solutions: partitionings of a given ground set

Problem Types

Subset Problems

- Solutions: subsets of a given ground set
- e.g., independent set in a graph, satisfying assignment to a Boolean formula
- Brute-force in $O^*(2^n)$ time

Permutation Problems

- Solutions: permutations of a given ground set
- e.g., Hamilton path of a graph, or tour as in TSP
- Brute-force in $O^*(n!) = 2^{O(n \log n)}$ time

Partitioning Problems

- Solutions: partitionings of a given ground set
- e.g., graph coloring: partition vertices of a graph into independent sets

Problem Types

Subset Problems

- Solutions: subsets of a given ground set
- e.g., independent set in a graph, satisfying assignment to a Boolean formula
- Brute-force in $O^*(2^n)$ time

Permutation Problems

- Solutions: permutations of a given ground set
- e.g., Hamilton path of a graph, or tour as in TSP
- Brute-force in $O^*(n!) = 2^{O(n \log n)}$ time

Partitioning Problems

- Solutions: partitionings of a given ground set
- e.g., graph coloring: partition vertices of a graph into independent sets
- Brute-force in $O^*(n^n) = 2^{O(n \log n)}$ time

Branching Algorithms

- Standard technique

Branching Algorithms

- Standard technique

- Typical properties
  - polynomial space
  - in practice, often faster than its worst-case runtime
  - simple speedups (reduction rules, branch-and-bound)

Branching Algorithms

- Standard technique

- Typical properties
  - polynomial space
  - in practice, often faster than its worst-case runtime
  - simple speedups (reduction rules, branch-and-bound)

- Other names:
  Backtracking, search-tree algorithms, . . .

General Form

- **Branching Rules:**
  Choose a small part of the solution and solve a corresponding subproblem for each choice
  $\Rightarrow$ obtain a solution of the original problem based on the solutions of the subproblems

General Form

- **Branching Rules:**
  Choose a small part of the solution and solve a corresponding subproblem for each choice
  $\Rightarrow$ obtain a solution of the original problem based on the solutions of the subproblems

- **Reduction Rules:**
  Reduce the problem size (also: exit/rejection conditions)

General Form

- **Branching Rules:**
  Choose a small part of the solution and solve a corresponding subproblem for each choice
  $\Rightarrow$ obtain a solution of the original problem based on the solutions of the subproblems

- **Reduction Rules:**
  Reduce the problem size (also: exit/rejection conditions)

- **Correctness:**
  Often immediate/obvious

General Form

- **Branching Rules:**
  Choose a small part of the solution and solve a corresponding subproblem for each choice
  $\Rightarrow$ obtain a solution of the original problem based on the solutions of the subproblems

- **Reduction Rules:**
  Reduce the problem size (also: exit/rejection conditions)

- **Correctness:**
  Often immediate/obvious

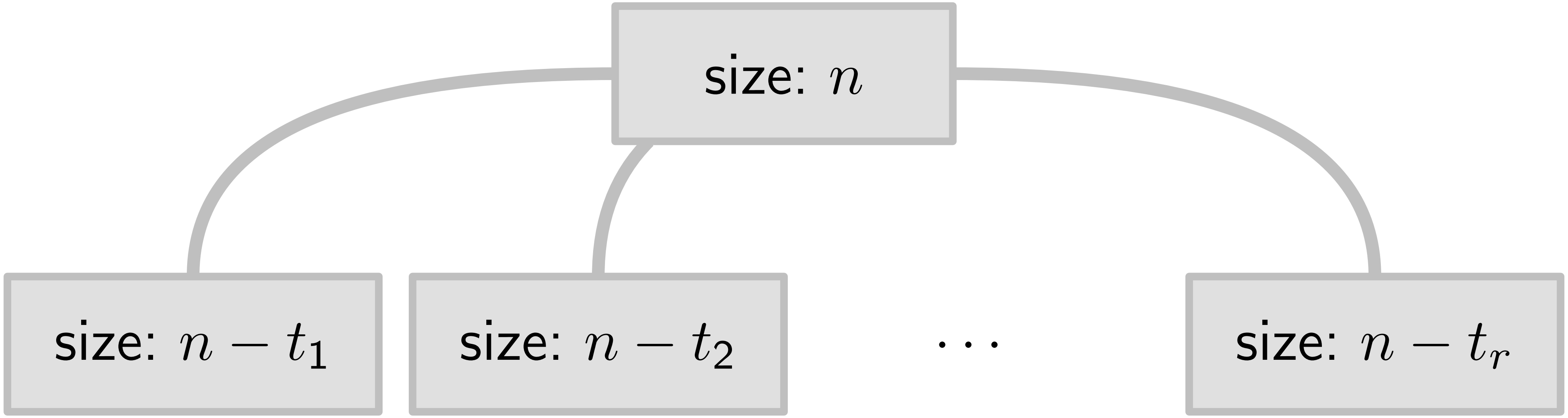
- **Specification:**
  Often only calculates the optimal value.

Branching Vectors

- Apply a branching rule b to an instance I of size n.

Branching Vectors

- Apply a branching rule b to an instance I of size n.
- Suppose that b decomposes I into $r \geq 2$ sub-instances of sizes $n-t_1,\ n-t_2,\ \ldots,\ n-t_r$, where $t_i > 0$ for each i.


size: n

size: $n-t_1$

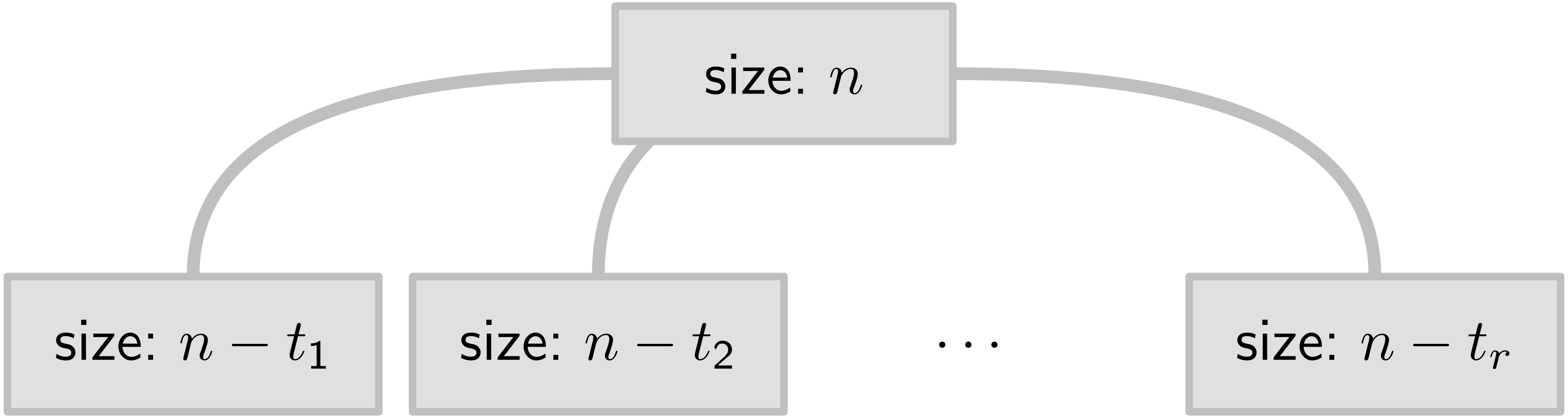
size: $n-t_2$

$\cdots$

size: $n-t_r$

Branching Vectors

- Apply a branching rule b to an instance I of size n.
- Suppose that b decomposes I into $r \geq 2$ sub-instances of sizes $n - t_1,\ n - t_2,\ \ldots,\ n - t_r$, where $t_i > 0$ for each i.
- We call $(t_1, t_2, \ldots, t_r)$ *branching vector of* b.



size: n

size: $n - t_1$

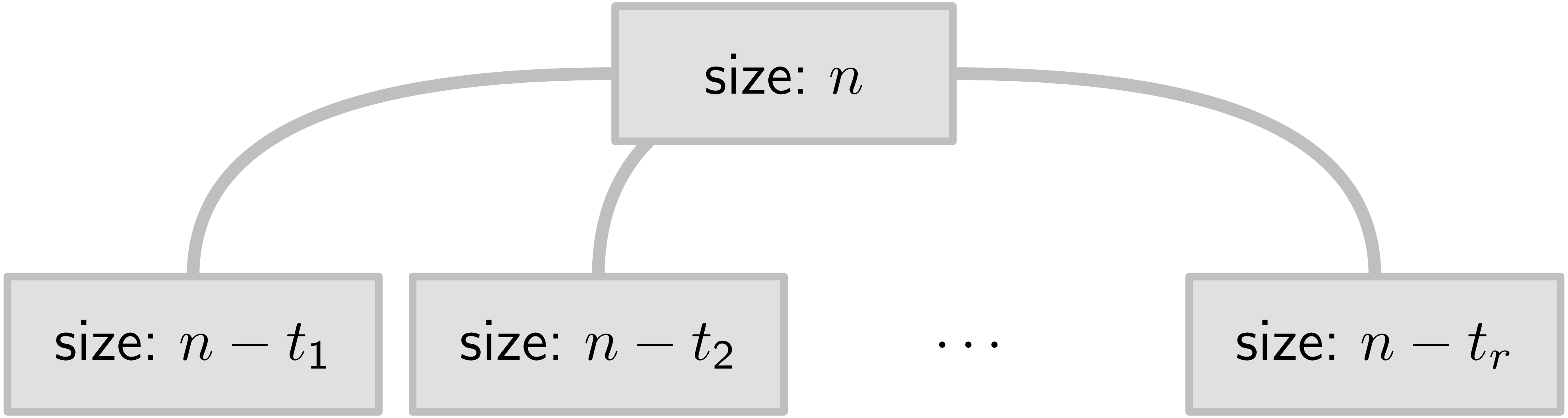
size: $n - t_2$

$\cdots$

size: $n - t_r$

Branching Vectors

- Apply a branching rule b to an instance I of size n.
- Suppose that b decomposes I into $r \geq 2$ sub-instances of sizes $n - t_1,\ n - t_2,\ \ldots,\ n - t_r$, where $t_i > 0$ for each i.
- We call $(t_1, t_2, \ldots, t_r)$ *branching vector of b*.



size: n

size: $n - t_1$

size: $n - t_2$

$\cdots$

size: $n - t_r$

Recurrence: $T(n) \ \leq \ T(n - t_1) + T(n - t_2) + \ldots + T(n - t_r)$

Theorem on Branching Vectors

Thm. Let b be a branching rule with vector $(t_1, t_2, \ldots, t_r)$. Then the runtime of an algorithm executing b is $O^*(\alpha^n)$, where α is the unique positive root of

Theorem on Branching Vectors

Thm. Let b be a branching rule with vector $(t_1, t_2, \ldots, t_r)$. Then the runtime of an algorithm executing b is $O^*(\alpha^n)$, where α is the unique positive root of

$$\alpha^n \;=\; \alpha^{n-t_1} + \alpha^{n-t_2} + \ldots + \alpha^{n-t_r}$$

Theorem on Branching Vectors

Thm. Let b be a branching rule with vector $(t_1, t_2, \ldots, t_r)$. Then the runtime of an algorithm executing b is $O^*(\alpha^n)$, where α is the unique positive root of

$$\alpha^n = \alpha^{n-t_1} + \alpha^{n-t_2} + \ldots + \alpha^{n-t_r}$$

$$\alpha^n - \alpha^{n-t_1} - \alpha^{n-t_2} - \ldots - \alpha^{n-t_r} = 0$$

Theorem on Branching Vectors

Thm. Let b be a branching rule with vector $(t_1, t_2, \ldots, t_r)$. Then the runtime of an algorithm executing b is $O^*(\alpha^n)$, where α is the unique positive root of

$$\begin{aligned} \alpha^n &= \alpha^{n-t_1} + \alpha^{n-t_2} + \ldots + \alpha^{n-t_r} \\ \alpha^n - \alpha^{n-t_1} - \alpha^{n-t_2} - \ldots - \alpha^{n-t_r} &= 0 \\ 1 - \alpha^{-t_1} - \alpha^{-t_2} - \ldots - \alpha^{-t_r} &= 0 \end{aligned}$$

Theorem on Branching Vectors

Thm. Let b be a branching rule with vector $(t_1, t_2, \ldots, t_r)$. Then the runtime of an algorithm executing b is $O^*(\alpha^n)$, where α is the unique positive root of

$$\begin{aligned} \alpha^n &= \alpha^{n-t_1} + \alpha^{n-t_2} + \ldots + \alpha^{n-t_r} \\ \alpha^n - \alpha^{n-t_1} - \alpha^{n-t_2} - \ldots - \alpha^{n-t_r} &= 0 \\ 1 - \alpha^{-t_1} - \alpha^{-t_2} - \ldots - \alpha^{-t_r} &= 0 \\ \alpha^m - \alpha^{m-t_1} - \alpha^{m-t_2} - \ldots - \alpha^{m-t_r} &= 0 \end{aligned}$$

where $m := \max_{1 \le i \le r} t_i$.

Theorem on Branching Vectors

Thm. Let b be a branching rule with vector $(t_1, t_2, \ldots, t_r)$. Then the runtime of an algorithm executing b is $O^*(\alpha^n)$, where α is the unique positive root of

$$\begin{aligned} \alpha^n &= \alpha^{n-t_1} + \alpha^{n-t_2} + \ldots + \alpha^{n-t_r} \\ \alpha^n - \alpha^{n-t_1} - \alpha^{n-t_2} - \ldots - \alpha^{n-t_r} &= 0 \\ 1 - \alpha^{-t_1} - \alpha^{-t_2} - \ldots - \alpha^{-t_r} &= 0 \\ \alpha^m - \alpha^{m-t_1} - \alpha^{m-t_2} - \ldots - \alpha^{m-t_r} &= 0 \end{aligned}$$

where $m := \max_{1 \le i \le r} t_i$.

- Denote solution by $\tau(t_1, t_2, \ldots, t_r)$.

Theorem on Branching Vectors

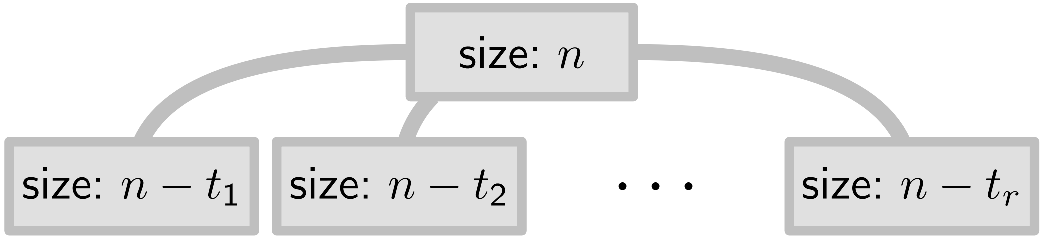
Thm. Let b be a branching rule with vector $(t_1, t_2, \ldots, t_r)$. Then the runtime of an algorithm executing b is $O^*(\alpha^n)$, where α is the unique positive root of

$$\begin{aligned} \alpha^n &= \alpha^{n-t_1} + \alpha^{n-t_2} + \ldots + \alpha^{n-t_r} \\ \alpha^n - \alpha^{n-t_1} - \alpha^{n-t_2} - \ldots - \alpha^{n-t_r} &= 0 \\ 1 - \alpha^{-t_1} - \alpha^{-t_2} - \ldots - \alpha^{-t_r} &= 0 \\ \alpha^m - \alpha^{m-t_1} - \alpha^{m-t_2} - \ldots - \alpha^{m-t_r} &= 0 \end{aligned}$$

where $m := \max_{1 \leq i \leq r} t_i$.

- Denote solution by $\tau(t_1, t_2, \ldots, t_r)$.
- Often irrational.

Properties of $\tau(\cdot)$



size: n

size: $n - t_1$

size: $n - t_2$

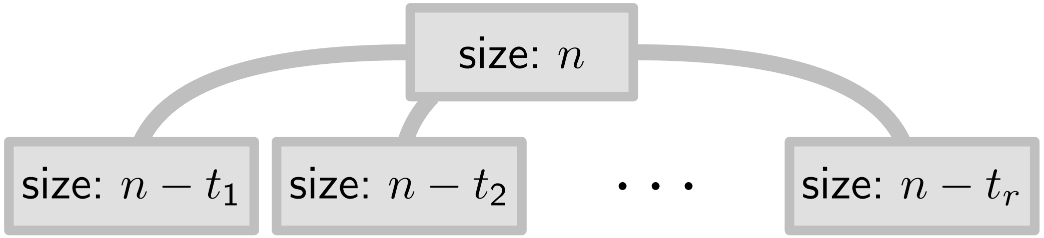
$\cdots$

size: $n - t_r$

Thm. Let $r \geq 2$ and, for $i = 1, \ldots, r$, let $t_i > 0$. Then:

(i) $\tau(t_1, \ldots, t_r) > 1$,

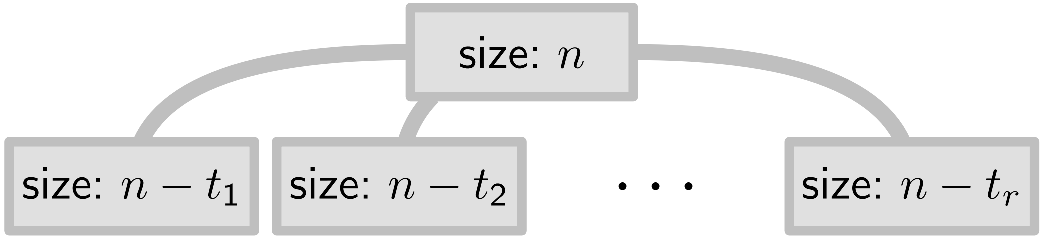
Properties of $\tau(\cdot)$



Thm. Let $r \geq 2$ and, for $i = 1, \ldots, r$, let $t_i > 0$. Then:

(i) $\tau(t_1, \ldots, t_r) > 1$,
(ii) $\tau(t_1, \ldots, t_r) = \tau(t_{\pi(1)}, \ldots, t_{\pi(r)})$
for every permutation π.

Properties of $\tau(\cdot)$



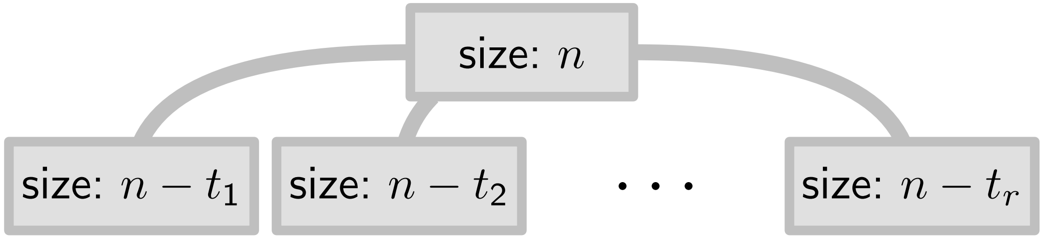
Thm. Let $r \geq 2$ and, for $i = 1, \ldots, r$, let $t_i > 0$. Then:

(i) $\tau(t_1, \ldots, t_r) > 1$,

(ii) $\tau(t_1, \ldots, t_r) = \tau(t_{\pi(1)}, \ldots, t_{\pi(r)})$ for every permutation π,

(iii) $\tau(t_1, t_2 \ldots, t_r) < \tau(t_1', t_2, \ldots, t_r)$ if $t_1 > t_1'$.

Properties of $\tau(\cdot)$



size: n

size: $n - t_1$ size: $n - t_2$ $\cdots$ size: $n - t_r$

Thm. Let $r \geq 2$ and, for $i = 1, \ldots, r$, let $t_i > 0$. Then:

(i) $\tau(t_1, \ldots, t_r) > 1$,

(ii) $\tau(t_1, \ldots, t_r) = \tau(t_{\pi(1)}, \ldots, t_{\pi(r)})$
for every permutation π,

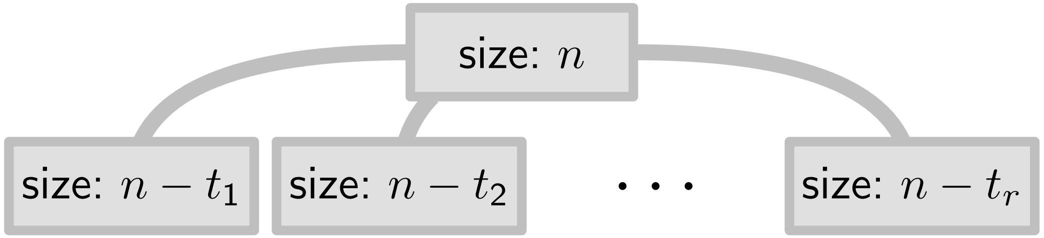
(iii) $\tau(t_1, t_2 \ldots, t_r) < \tau(t'_1, t_2, \ldots, t_r)$ if $t_1 > t'_1$.

Lemma. For $i, j, k > 0$, the following balancing properties apply:

(i) $\tau(k, k) \leq \tau(i, j)$ if $i + j = 2k$,

(ii) $\tau(i, j) > \tau(i + \varepsilon, j - \varepsilon)$
if $0 < i < j$ and $0 < \varepsilon < \frac{j-i}{2}$.

Properties of $\tau(\cdot)$



size: n

size: $n - t_1$

size: $n - t_2$

$\cdots$

size: $n - t_r$

Thm. Let $r \geq 2$ and, for $i = 1, \ldots, r$, let $t_i > 0$. Then:

(i) $\tau(t_1, \ldots, t_r) > 1$,

(ii) $\tau(t_1, \ldots, t_r) = \tau(t_{\pi(1)}, \ldots, t_{\pi(r)})$ for every permutation π,

(iii) $\tau(t_1, t_2 \ldots, t_r) < \tau(t'_1, t_2, \ldots, t_r)$ if $t_1 > t'_1$.

Lemma. For $i, j, k > 0$, the following balancing properties apply:

(i) $\tau(k, k) \leq \tau(i, j)$ if $i + j = 2k$,

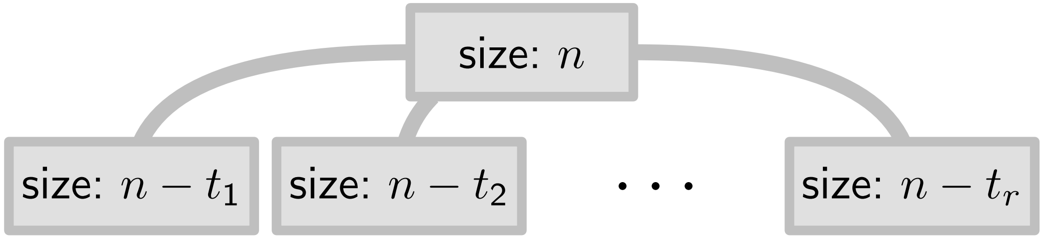
(ii) $\tau(i, j) > \tau(i + \varepsilon, j - \varepsilon)$ if $0 < i < j$ and $0 < \varepsilon < \frac{j-i}{2}$.

E.g.:

$$\tau(1, 1) = 2$$

$$\tau(1, 2) = \frac{1+\sqrt{5}}{2} < 1.62$$

Properties of $\tau(\cdot)$



size: n

size: $n-t_1$

size: $n-t_2$

$\cdots$

size: $n-t_r$

Thm. Let $r \geq 2$ and, for $i = 1, \dots, r$, let $t_i > 0$. Then:

(i) $\tau(t_1, \dots, t_r) > 1$,
(ii) $\tau(t_1, \dots, t_r) = \tau(t_{\pi(1)}, \dots, t_{\pi(r)})$
for every permutation π,
(iii) $\tau(t_1, t_2 \dots, t_r) < \tau(t'_1, t_2, \dots, t_r)$ if $t_1 > t'_1$.

Lemma. For $i, j, k > 0$, the following balancing properties apply:

(i) $\tau(k, k) \leq \tau(i, j)$ if $i + j = 2k$,
(ii) $\tau(i, j) > \tau(i + \varepsilon, j - \varepsilon)$
if $0 < i < j$ and $0 < \varepsilon < \frac{j-i}{2}$.

E.g.:

$\tau(1,1) = 2$

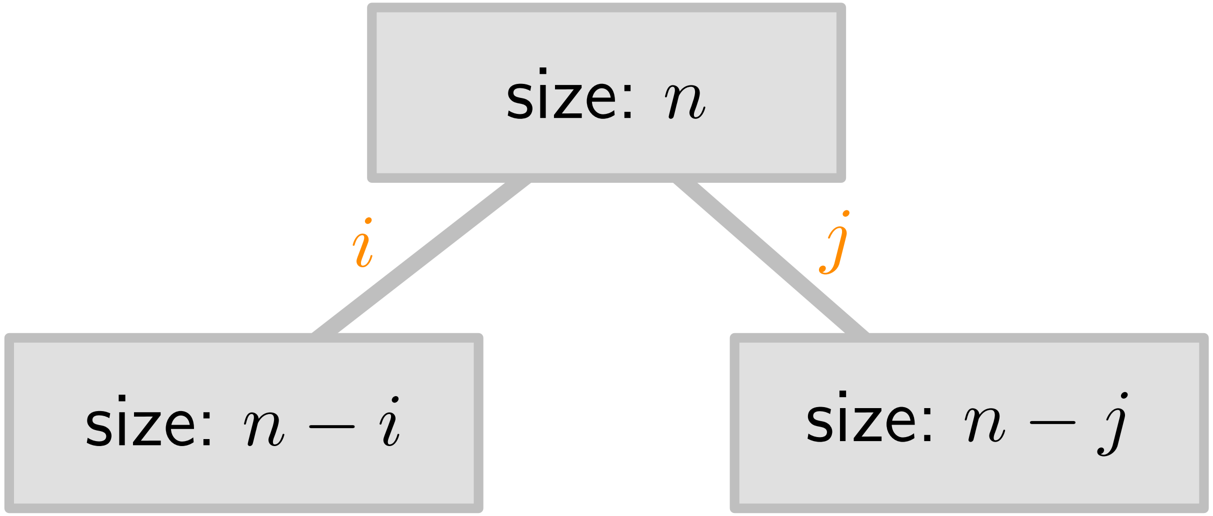
$\tau(1,2) = \frac{1+\sqrt{5}}{2} < 1.62$

$\tau(3,3) = \sqrt[3]{2} < 1.26$

$\tau(2,4) < 1.272$

$\tau(1,5) < 1.3248$

“Addition” of Branching Vectors



size: n

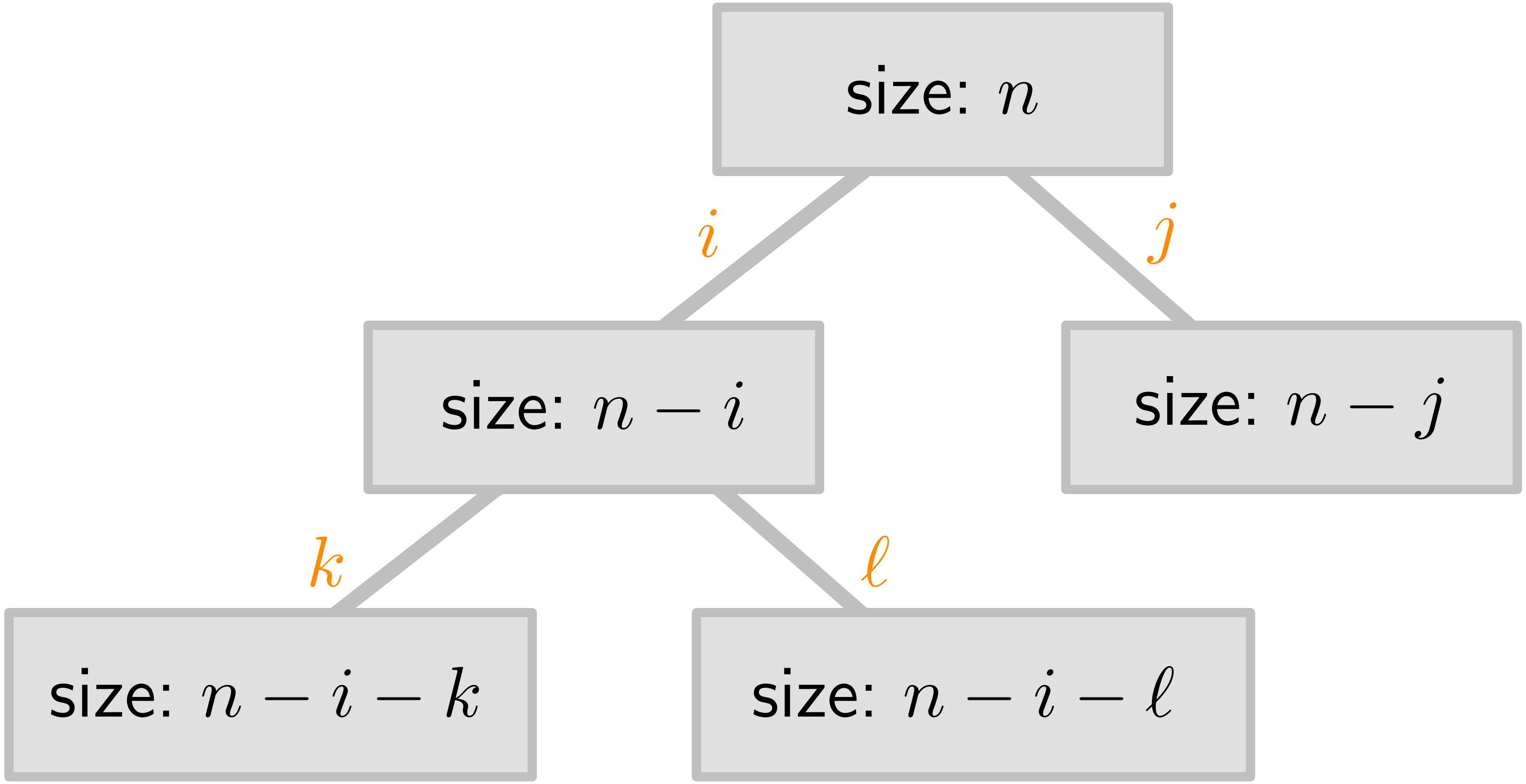
i

j

size: $n-i$

size: $n-j$

“Addition” of Branching Vectors



size: n

i

j

size: $n-i$

size: $n-j$

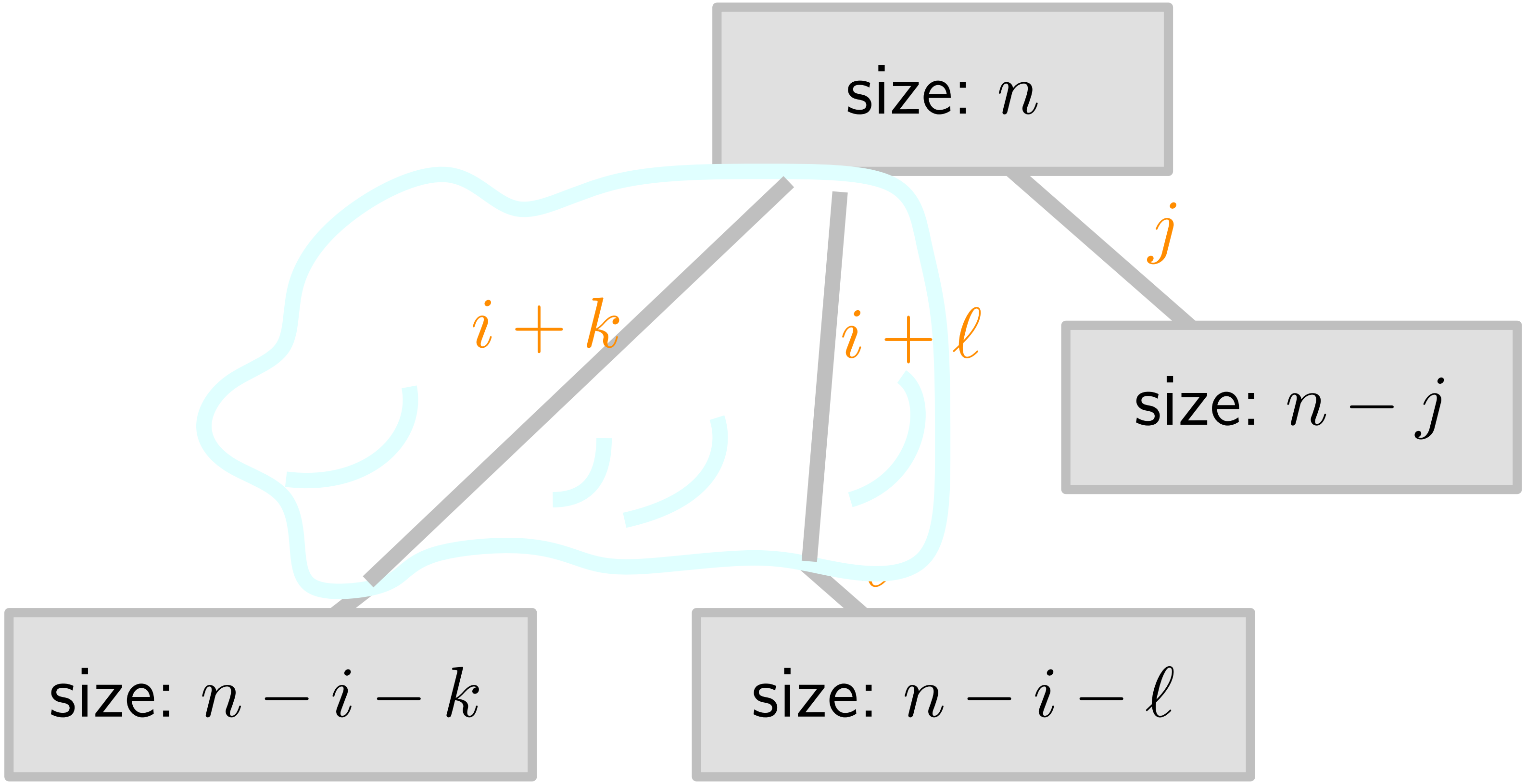
k

ℓ

size: $n-i-k$

size: $n-i-\ell$

"Addition" of Branching Vectors



Branching-Vector: $(i+k,\ i+\ell,\ j)$

SATISFIABILITY (SAT)

Input: propositional logic formula F in conjunctive NF

Question: $\exists$ satisfying assignment for F?

SATISFIABILITY (SAT)

Input: propositional logic formula F in conjunctive NF

Question: $\exists$ satisfying assignment for F?

E.g., $(x_1 \lor \bar{x}_2 \lor x_3) \land \underbrace{(\bar{x}_1 \lor x_4 \lor x_5)}_{\text{clause}}$

SATISFIABILITY (SAT)

Input: propositional logic formula F in conjunctive NF

Question: $\exists$ satisfying assignment for F?

E.g., $(\underset{\substack{\nearrow \\ \text{variable}}}{x_1} \vee \bar{x}_2 \vee x_3) \wedge \underbrace{(\bar{x}_1 \vee x_4 \vee x_5)}_{\text{clause}}$

SATISFIABILITY (SAT)

Input: propositional logic formula F in conjunctive NF

Question: $\exists$ satisfying assignment for F?

E.g., $(x_1 \vee \bar{x}_2 \vee x_3) \wedge \underbrace{(\bar{x}_1 \vee x_4 \vee x_5)}_{\text{clause}}$

variable literal

SATISFIABILITY (SAT)

Input: propositional logic formula F in conjunctive NF

Question: $\exists$ satisfying assignment for F?

E.g., $(x_1 \vee \bar{x}_2 \vee x_3) \wedge \underbrace{(\bar{x}_1 \vee x_4 \vee x_5)}_{\text{clause}}$

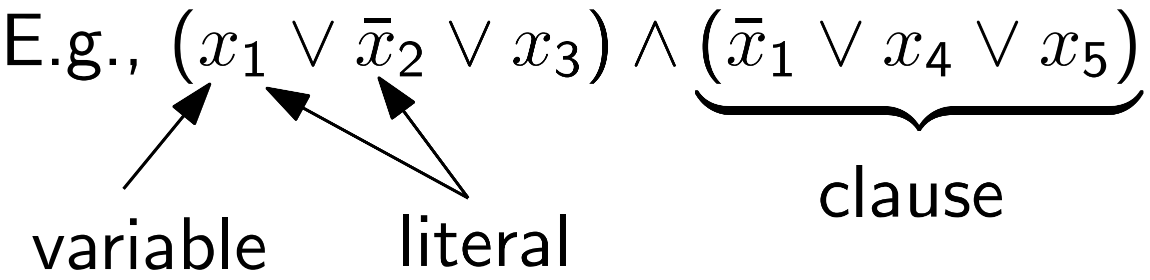
variable literal

Brute-Force:

SATISFIABILITY (SAT)

Input: propositional logic formula F in conjunctive NF

Question: $\exists$ satisfying assignment for F?



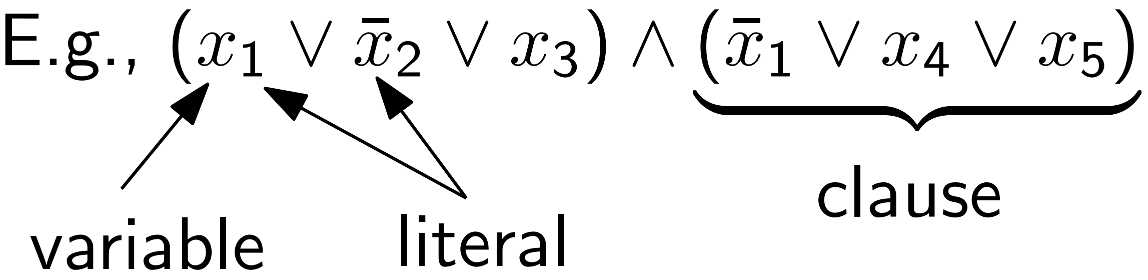
E.g., $(x_1 \vee \bar{x}_2 \vee x_3) \wedge \underbrace{(\bar{x}_1 \vee x_4 \vee x_5)}_{\text{clause}}$

Brute-Force: Try all 2^n variable assignments.

SATISFIABILITY (SAT)

Input: propositional logic formula F in conjunctive NF

Question: $\exists$ satisfying assignment for F?



E.g., $(x_1 \vee \bar{x}_2 \vee x_3) \wedge \underbrace{(\bar{x}_1 \vee x_4 \vee x_5)}_{\text{clause}}$

variable

literal

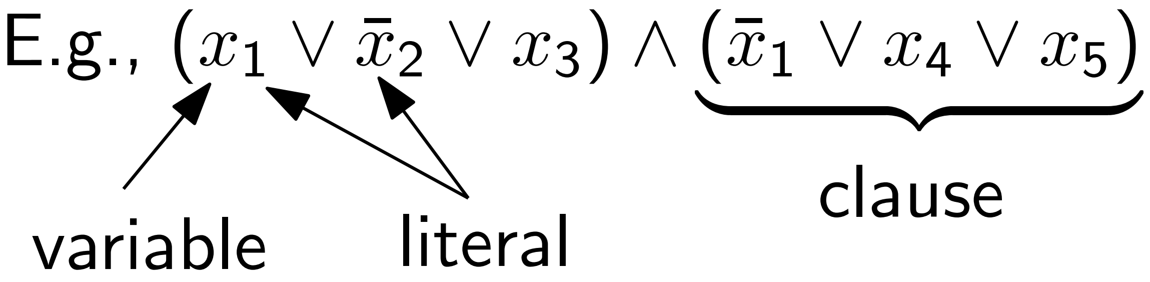
Brute-Force: Try all 2^n variable assignments.

Runtime:

SATISFIABILITY (SAT)

Input: propositional logic formula F in conjunctive NF

Question: $\exists$ satisfying assignment for F?



E.g., $(x_1 \vee \bar{x}_2 \vee x_3) \wedge \underbrace{(\bar{x}_1 \vee x_4 \vee x_5)}_{\text{clause}}$

variable literal

Brute-Force: Try all 2^n variable assignments.

Runtime: $O(2^n \cdot n \cdot m)$, where $n = \#$variables, $m = \#$clauses.

SATISFIABILITY (SAT)

Input: propositional logic formula F in conjunctive NF

Question: $\exists$ satisfying assignment for F?

E.g., $(x_1 \vee \bar{x}_2 \vee x_3) \wedge \underbrace{(\bar{x}_1 \vee x_4 \vee x_5)}_{\text{clause}}$

variable literal

Brute-Force: Try all 2^n variable assignments.

Runtime: $O(2^n \cdot n \cdot m)$, where $n = \#$variables, $m = \#$clauses.

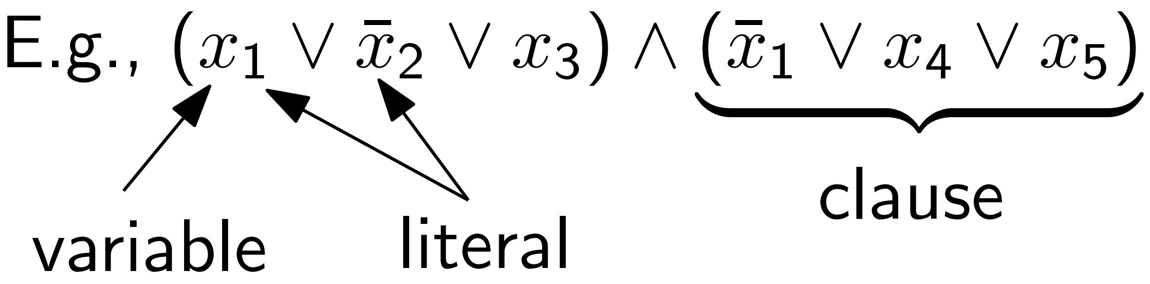
Strong Exponential Time Hypothesis (SETH) implies that:

$\nexists$ algorithm for SAT in $o(2^n)$ time, i.e., in $O^*((2-\varepsilon)^n), \varepsilon > 0$.

SATISFIABILITY (SAT)

Input: propositional logic formula F in conjunctive NF

Question: $\exists$ satisfying assignment for F?



E.g., $(x_1 \vee \bar{x}_2 \vee x_3) \wedge \underbrace{(\bar{x}_1 \vee x_4 \vee x_5)}_{\text{clause}}$

Brute-Force: Try all 2^n variable assignments.

Runtime: $O(2^n \cdot n \cdot m)$, where $n = \#$variables, $m = \#$clauses.

Strong Exponential Time Hypothesis (SETH) implies that:
$\nexists$ algorithm for SAT in $o(2^n)$ time, i.e., in $O^*((2-\varepsilon)^n), \varepsilon > 0$.

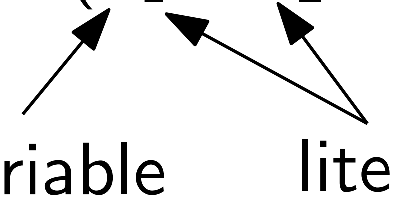
For more on (S)ETH, see [Parameterized Algorithms 14.1].

Satisfiability (SAT)

Input: propositional logic formula F in conjunctive NF

Question: $\exists$ satisfying assignment for F?

E.g., $(x_1 \vee \bar{x}_2 \vee x_3) \wedge \underbrace{(\bar{x}_1 \vee x_4 \vee x_5)}_{\text{clause}}$



variable literal

Brute-Force: Try all 2^n variable assignments.

Runtime: $O(2^n \cdot n \cdot m)$, where $n = \#$variables, $m = \#$clauses.

Strong Exponential Time Hypothesis (SETH) implies that:

$\nexists$ algorithm for SAT in $o(2^n)$ time, i.e., in $O^*((2-\varepsilon)^n), \varepsilon > 0$.

For more on (S)ETH, see [Parameterized Algorithms 14.1].

k-SAT: Each clause has $\leq k$ literals.

A Better Algorithm for k-SAT

- Goal:
  Solve k-SAT in time $O^*({\alpha_k}^n)$, where $\alpha_k < 2$ for every k.

A Better Algorithm for k-SAT

- Goal:
  Solve k-SAT in time $O^*({\alpha_k}^n)$, where $\alpha_k < 2$ for every k.

- Idea: Branch on variables.

A Better Algorithm for k-SAT

- Goal:
  Solve k-SAT in time $O^*({\alpha_k}^n)$, where $\alpha_k < 2$ for every k.

- Idea: Branch on variables.

- For a partial assignment t,
  let $F[t]$ be the *reduced* formula that we get after
  – removing false literals from clauses and
  – removing clauses with true literals.

A Better Algorithm for k-SAT

- Goal:
  Solve k-SAT in time $O^*({\alpha_k}^n)$, where $\alpha_k < 2$ for every k.

- Idea: Branch on variables.

- For a partial assignment t,
  let $F[t]$ be the *reduced* formula that we get after
  – removing false literals from clauses and
  – removing clauses with true literals.

  Example:

  - $F = (x_1 \vee \bar{x}_2 \vee x_3) \wedge (\bar{x}_1 \vee x_4 \vee x_5)$

A Better Algorithm for k-SAT

- Goal:
  Solve k-SAT in time $O^*({\alpha_k}^n)$, where $\alpha_k < 2$ for every k.

- Idea: Branch on variables.

- For a partial assignment t,
  let $F[t]$ be the *reduced* formula that we get after
  – removing false literals from clauses and
  – removing clauses with true literals.

  Example:

  - $F = (x_1 \vee \bar{x}_2 \vee x_3) \wedge (\bar{x}_1 \vee x_4 \vee x_5)$
  - t: $x_1 =$ false

A Better Algorithm for k-SAT

- Goal:
  Solve k-SAT in time $O^*({\alpha_k}^n)$, where $\alpha_k < 2$ for every k.

- Idea: Branch on variables.

- For a partial assignment t,
  let $F[t]$ be the *reduced* formula that we get after
  – removing false literals from clauses and
  – removing clauses with true literals.

  Example:
  - $F = (x_1 \vee \bar{x}_2 \vee x_3) \wedge (\bar{x}_1 \vee x_4 \vee x_5)$
  - t: $x_1 =$ false
  - $F[t] =$

A Better Algorithm for k-SAT

- Goal:
  Solve k-SAT in time $O^*({\alpha_k}^n)$, where $\alpha_k < 2$ for every k.

- Idea: Branch on variables.

- For a partial assignment t,
  let $F[t]$ be the *reduced* formula that we get after
  – removing false literals from clauses and
  – removing clauses with true literals.

  Example:

  - $F = (x_1 \vee \bar{x}_2 \vee x_3) \wedge (\bar{x}_1 \vee x_4 \vee x_5)$
  - t: $x_1 =$ false
  - $F[t] = (\bar{x}_2 \vee x_3)$

Properties of Reduced Formulas

- $F[t]$ satisfiable $\Rightarrow$ F satisfiable

- $F = (x_1 \vee \bar{x}_2 \vee x_3) \wedge (\bar{x}_1 \vee x_4 \vee x_5)$
- $F[x_1 = \mathsf{f}] =$

Properties of Reduced Formulas

- $F[t]$ satisfiable $\Rightarrow$ F satisfiable

- $F = (x_1 \vee \bar{x}_2 \vee x_3) \wedge (\bar{x}_1 \vee x_4 \vee x_5)$
- $F[x_1 = \mathsf{f}] = (\bar{x}_2 \vee x_3)$ satisfiable

Properties of Reduced Formulas

- $F[t]$ satisfiable $\Rightarrow F$ satisfiable

- $F = (x_1 \vee \bar{x}_2 \vee x_3) \wedge (\bar{x}_1 \vee x_4 \vee x_5)$
- $F[x_1 = \mathsf{f}] = (\bar{x}_2 \vee x_3)$ satisfiable
- $F[x_2 = \mathsf{t}, x_3 = x_4 = x_5 = \mathsf{f}] =$

Properties of Reduced Formulas

- $F[t]$ satisfiable $\Rightarrow$ F satisfiable

- $F = (x_1 \vee \bar{x}_2 \vee x_3) \wedge (\bar{x}_1 \vee x_4 \vee x_5)$
- $F[x_1 = \mathsf{f}] = (\bar{x}_2 \vee x_3)$ satisfiable
- $F[x_2 = \mathsf{t}, x_3 = x_4 = x_5 = \mathsf{f}] = x_1 \wedge \bar{x}_1$ not satisfiable

Properties of Reduced Formulas

- $F[t]$ satisfiable $\Rightarrow$ F satisfiable

- $F = (x_1 \vee \bar{x}_2 \vee x_3) \wedge (\bar{x}_1 \vee x_4 \vee x_5)$
- $F[x_1 = \mathsf{f}] = (\bar{x}_2 \vee x_3)$ satisfiable
- $F[x_2 = \mathsf{t}, x_3 = x_4 = x_5 = \mathsf{f}] = x_1 \wedge \bar{x}_1$ not satisfiable
- $F[x_1 = x_3 = \mathsf{f}, x_2 = \mathsf{t}] =$

Properties of Reduced Formulas

- $F[t]$ satisfiable $\Rightarrow$ F satisfiable

- $F = (x_1 \vee \bar{x}_2 \vee x_3) \wedge (\bar{x}_1 \vee x_4 \vee x_5)$
- $F[x_1 = \mathsf{f}] = (\bar{x}_2 \vee x_3)$ satisfiable
- $F[x_2 = \mathsf{t}, x_3 = x_4 = x_5 = \mathsf{f}] = x_1 \wedge \bar{x}_1$ not satisfiable
- $F[x_1 = x_3 = \mathsf{f}, x_2 = \mathsf{t}] = ()$

Properties of Reduced Formulas

- $F[t]$ satisfiable $\Rightarrow$ F satisfiable
- $F[t]$ contains an empty clause $\Rightarrow$ $F[t]$ not satisfiable

---

- $F = (x_1 \vee \bar{x}_2 \vee x_3) \wedge (\bar{x}_1 \vee x_4 \vee x_5)$
- $F[x_1 = \mathsf{f}] = (\bar{x}_2 \vee x_3)$ satisfiable
- $F[x_2 = \mathsf{t}, x_3 = x_4 = x_5 = \mathsf{f}] = x_1 \wedge \bar{x}_1$ not satisfiable
- $F[x_1 = x_3 = \mathsf{f}, x_2 = \mathsf{t}] = ()$

Properties of Reduced Formulas

- $F[t]$ satisfiable $\Rightarrow$ F satisfiable
- $F[t]$ contains an empty clause $\Rightarrow$ $F[t]$ not satisfiable

---

- $F = (x_1 \vee \bar{x}_2 \vee x_3) \wedge (\bar{x}_1 \vee x_4 \vee x_5)$
- $F[x_1 = \mathsf{f}] = (\bar{x}_2 \vee x_3)$ satisfiable
- $F[x_2 = \mathsf{t}, x_3 = x_4 = x_5 = \mathsf{f}] = x_1 \wedge \bar{x}_1$ not satisfiable
- $F[x_1 = x_3 = \mathsf{f}, x_2 = \mathsf{t}] = ()$
- $F[x_3 = x_4 = \mathsf{t}] =$

Properties of Reduced Formulas

- $F[t]$ satisfiable $\Rightarrow F$ satisfiable
- $F[t]$ contains an empty clause $\Rightarrow F[t]$ not satisfiable

---

- $F = (x_1 \vee \bar{x}_2 \vee x_3) \wedge (\bar{x}_1 \vee x_4 \vee x_5)$
- $F[x_1 = \mathsf{f}] = (\bar{x}_2 \vee x_3)$ satisfiable
- $F[x_2 = \mathsf{t}, x_3 = x_4 = x_5 = \mathsf{f}] = x_1 \wedge \bar{x}_1$ not satisfiable
- $F[x_1 = x_3 = \mathsf{f}, x_2 = \mathsf{t}] = ()$
- $F[x_3 = x_4 = \mathsf{t}] =$ empty formula

Properties of Reduced Formulas

- $F[t]$ satisfiable $\Rightarrow$ F satisfiable
- $F[t]$ contains an empty clause $\Rightarrow$ $F[t]$ not satisfiable
- $F[t]$ empty $\Rightarrow$ $F[t]$ satisfiable

- $F = (x_1 \vee \bar{x}_2 \vee x_3) \wedge (\bar{x}_1 \vee x_4 \vee x_5)$
- $F[x_1 = \mathsf{f}] = (\bar{x}_2 \vee x_3)$ satisfiable
- $F[x_2 = \mathsf{t}, x_3 = x_4 = x_5 = \mathsf{f}] = x_1 \wedge \bar{x}_1$ not satisfiable
- $F[x_1 = x_3 = \mathsf{f}, x_2 = \mathsf{t}] = ()$
- $F[x_3 = x_4 = \mathsf{t}] =$ empty formula

A First Algorithm for k-SAT

```
Algorithm $k$-SAT-v1($F$)
  if $F$ is empty then
    return true
  if $F$ contains an empty clause then
    return false
  pick clause $c = (\ell_1 \vee \ell_2 \vee \ldots \vee \ell_q)$ from $F$, where $q \leq k$
  $t_1$: $\ell_1 =$ true
  $t_2$: $\ell_1 =$ false, $\ell_2 =$ true
  $t_3$: $\ell_1 =$ false, $\ell_2 =$ false, $\ell_3 =$ true
  $\vdots$
  $t_q$: $\ell_1 =$ false, $\ell_2 =$ false, $\ldots, \ell_{q-1} =$ false, $\ell_q =$ true
  return $\bigvee_{i=1}^{q}$ $k$-SAT-v1($F[t_i]$)
```

A First Algorithm for k-SAT

```
Algorithm $k$-SAT-v1($F$)
  if $F$ is empty then
    return true
  if $F$ contains an empty clause then
    return false
  pick clause $c = (\ell_1 \vee \ell_2 \vee \ldots \vee \ell_q)$ from $F$, where $q \leq k$
  $t_1$: $\ell_1 =$ true
  $t_2$: $\ell_1 =$ false, $\ell_2 =$ true
  $t_3$: $\ell_1 =$ false, $\ell_2 =$ false, $\ell_3 =$ true
  $\vdots$
  $t_q$: $\ell_1 =$ false, $\ell_2 =$ false, $\ldots, \ell_{q-1} =$ false, $\ell_q =$ true
  return $\bigvee_{i=1}^{q}$ $k$-SAT-v1($F[t_i]$)
```

- t_i: first true literal in c is ℓ_i

Runtime

- $F[t_i]$ has $n - i$ variables

Runtime

- $F[t_i]$ has $n - i$ variables

  $\Rightarrow$ branching vector $(1, 2, \ldots, k)$

Runtime

- $F[t_i]$ has $n - i$ variables
  $\Rightarrow$ branching vector $(1, 2, \ldots, k)$
- Runtime: $O^*({\beta_k}^n)$, where $\beta_k = \tau(1, 2, \ldots, k)$.

Runtime

- $F[t_i]$ has $n - i$ variables

  $\Rightarrow$ branching vector $(1, 2, \ldots, k)$

- Runtime: $O^*({\beta_k}^n)$, where $\beta_k = \tau(1, 2, \ldots, k)$.

  Note: $\tau(1, 2, \ldots, k) \ \Rightarrow \ {\beta_k}^k = \sum_{i=0}^{k-1} {\beta_k}^i$. (*)

Runtime

- $F[t_i]$ has $n-i$ variables

  $\Rightarrow$ branching vector $(1, 2, \ldots, k)$

- Runtime: $O^*({\beta_k}^n)$, where $\beta_k = \tau(1, 2, \ldots, k)$.

  Note: $\tau(1, 2, \ldots, k) \;\Rightarrow\; {\beta_k}^k = \sum_{i=0}^{k-1} {\beta_k}^i.$ $\quad(*)$

  So, $(*)\cdot\beta_k - (*) \;\Rightarrow$

Runtime

- $F[t_i]$ has $n - i$ variables

  $\Rightarrow$ branching vector $(1, 2, \ldots, k)$

- Runtime: $O^*({\beta_k}^n)$, where $\beta_k = \tau(1, 2, \ldots, k)$.

  Note: $\tau(1, 2, \ldots, k) \ \Rightarrow \ {\beta_k}^k = \sum_{i=0}^{k-1} {\beta_k}^i$. $\quad (*)$

  So, $(*) \cdot \beta_k - (*) \ \Rightarrow \ {\beta_k}^{k+1} - 2{\beta_k}^k + 1 = 0$.

Runtime

- $F[t_i]$ has $n - i$ variables

  $\Rightarrow$ branching vector $(1, 2, \ldots, k)$

- Runtime: $O^*({\beta_k}^n)$, where $\beta_k = \tau(1, 2, \ldots, k)$.

  Note: $\tau(1, 2, \ldots, k) \;\Rightarrow\; {\beta_k}^k = \sum_{i=0}^{k-1} {\beta_k}^i$.   (*)

  So, $(*) \cdot \beta_k - (*) \;\Rightarrow\; {\beta_k}^{k+1} - 2{\beta_k}^k + 1 = 0$.

- $\beta_1 = 1, \beta_2 < 1.6181, \beta_3 < 1.8393, \beta_4 < 1.9276, \beta_5 < 1.9660$

Runtime

- $F[t_i]$ has $n - i$ variables

  $\Rightarrow$ branching vector $(1, 2, \ldots, k)$

- Runtime: $O^*({\beta_k}^n)$, where $\beta_k = \tau(1, 2, \ldots, k)$.

  Note: $\tau(1, 2, \ldots, k) \;\Rightarrow\; {\beta_k}^k = \sum_{i=0}^{k-1} {\beta_k}^i$. $\quad(*)$

  So, $(*)\cdot\beta_k - (*) \;\Rightarrow\; {\beta_k}^{k+1} - 2{\beta_k}^k + 1 = 0$.

- $\beta_1 = 1$, $\beta_2 < 1.6181$, $\beta_3 < 1.8393$, $\beta_4 < 1.9276$, $\beta_5 < 1.9660$

Speeding Up the Algorithm

- “Branch-on-shortest” rule

Speeding Up the Algorithm

- “Branch-on-shortest” rule
- Hope: $\exists$ a clause of length $\leq k - 1$

Speeding Up the Algorithm

- "Branch-on-shortest" rule
- Hope: $\exists$ a clause of length $\leq k-1$

Def. A partial assignment t is an *autark* if every clause with a literal assigned by t also contains a literal assigned true by t.

Speeding Up the Algorithm

- "Branch-on-shortest" rule
- Hope: $\exists$ a clause of length $\leq k-1$

Def. A partial assignment t is an *autark* if every clause with a literal assigned by t also contains a literal assigned true by t.

Expl. $F = (x_1 \vee \bar{x}_2 \vee x_3) \wedge (\bar{x}_1 \vee x_4 \vee x_5) \wedge (\bar{x}_3 \vee x_5)$

t: $x_1 = x_4 =$ true

t': $x_1 =$ true, $x_4 =$ false

Speeding Up the Algorithm

- "Branch-on-shortest" rule
- Hope: $\exists$ a clause of length $\leq k-1$

Def. A partial assignment t is an *autark*
if every clause with a literal assigned by t
also contains a literal assigned true by t.

Expl. $F = (x_1 \vee \bar{x}_2 \vee x_3) \wedge (\bar{x}_1 \vee x_4 \vee x_5) \wedge (\bar{x}_3 \vee x_5)$

t: $x_1 = x_4 =$ true *is* an autark

t': $x_1 =$ true, $x_4 =$ false

Speeding Up the Algorithm

- "Branch-on-shortest" rule
- Hope: $\exists$ a clause of length $\leq k-1$

Def. A partial assignment t is an *autark* if every clause with a literal assigned by t also contains a literal assigned true by t.

Expl. $F = (x_1 \vee \bar{x}_2 \vee x_3) \wedge (\bar{x}_1 \vee x_4 \vee x_5) \wedge (\bar{x}_3 \vee x_5)$

t: $x_1 = x_4 =$ true *is* an autark

t': $x_1 =$ true, $x_4 =$ false is *not* an autark

Speeding Up the Algorithm

- "Branch-on-shortest" rule
- Hope: $\exists$ a clause of length $\leq k - 1$

Def. A partial assignment t is an *autark*
if every clause with a literal assigned by t
also contains a literal assigned true by t.

Expl. $F = (x_1 \vee \bar{x}_2 \vee x_3) \wedge (\bar{x}_1 \vee x_4 \vee x_5) \wedge (\bar{x}_3 \vee x_5)$

t: $x_1 = x_4 =$ true *is* an autark

t': $x_1 =$ true, $x_4 =$ false is *not* an autark

Obs.
- If t is an autark, then: F satisfiable $\Leftrightarrow$ $F[t]$ satisfiable.

Speeding Up the Algorithm

- "Branch-on-shortest" rule
- Hope: $\exists$ a clause of length $\leq k-1$

Def. A partial assignment t is an *autark*
if every clause with a literal assigned by t
also contains a literal assigned true by t.

Expl. $F = (x_1 \vee \bar{x}_2 \vee x_3) \wedge (\bar{x}_1 \vee x_4 \vee x_5) \wedge (\bar{x}_3 \vee x_5)$

t: $x_1 = x_4 =$ true *is* an autark

t': $x_1 =$ true, $x_4 =$ false is *not* an autark

Obs.
- If t is an autark, then: F satisfiable $\Leftrightarrow$ $F[t]$ satisfiable.
- If t is *not* an autark,
  then $F[t]$ contains a clause of length $\leq k-1$.

An Improved k-SAT Algorithm

```
Algorithm $k$-SAT-v2($F$)
    if $F$ is empty then
        return true
    if $F$ contains an empty clause then
        return false
    pick a smallest clause $c = (\ell_1 \vee \ell_2 \vee \ldots \vee \ell_q)$ from $F$, where $q \leq k$
    $t_1$: $\ell_1 =$true
    $t_2$: $\ell_1 = \text{false}, \ell_2 = \text{true}$
    $\vdots$
    $t_q$: $\ell_1 = \text{false}, \ell_2 = \text{false}, \ldots, \ell_{q-1} = \text{false}, \ell_q = \text{true}$
    if $t_i$ is an autark for some $i = 1, \ldots, q$ then
        return $k$-SAT-v2($F[t_i]$)
    else
        return $\bigvee_{i=1}^{q}$ $k$-SAT-v2($F[t_i]$)
```

An Improved k-SAT Algorithm

```
Algorithm $k$-SAT-v2($F$)
  if $F$ is empty then
    return true
  if $F$ contains an empty clause then
    return false
  pick a smallest clause $c = (\ell_1 \vee \ell_2 \vee \ldots \vee \ell_q)$ from $F$, where $q \leq k$
  $t_1$: $\ell_1 =$true
  $t_2$: $\ell_1 = \text{false}, \ell_2 = \text{true}$
  $\vdots$
  $t_q$: $\ell_1 = \text{false}, \ell_2 = \text{false}, \ldots, \ell_{q-1} = \text{false}, \ell_q = \text{true}$
  if $t_i$ is an autark for some $i = 1, \ldots, q$ then
    return $k$-SAT-v2($F[t_i]$)          $\leftarrow$ "Reduce"
  else
    return $\bigvee_{i=1}^{q} k$-SAT-v2($F[t_i]$)   $\leftarrow$ "Branch"
```

Runtime Analysis

- Claim: The runtime is $O^*(\alpha_k^n)$, where $\alpha_k = \beta_{k-1}$.

Runtime Analysis

- Claim: The runtime is $O^*(\alpha_k^n)$, where $\alpha_k = \beta_{k-1}$.
- Consider a node v in the search tree (not the root).

Runtime Analysis

- Claim: The runtime is $O^*(\alpha_k^n)$, where $\alpha_k = \beta_{k-1}$.
- Consider a node v in the search tree (not the root).

  If v is a k-branching, then v's parent is a "Reduce"-node

Runtime Analysis

- Claim: The runtime is $O^*(\alpha_k^n)$, where $\alpha_k = \beta_{k-1}$.
- Consider a node v in the search tree (not the root).

  If v is a k-branching, then v's parent is a "Reduce"-node because in a "Branch"-node, for each branch, the formula $F[t_i]$ contains a clause of length $\leq k-1$.

Runtime Analysis

- Claim: The runtime is $O^*(\alpha_k^n)$, where $\alpha_k = \beta_{k-1}$.

- Consider a node v in the search tree (not the root).

  If v is a k-branching, then v's parent is a "Reduce"-node because in a "Branch"-node, for each branch, the formula $F[t_i]$ contains a clause of length $\leq k-1$.

  Now:
  $\Rightarrow$ branching vector

  Before:
  $(1, 2, \ldots, k-1, k)$

Runtime Analysis

- Claim: The runtime is $O^*(\alpha_k^n)$, where $\alpha_k = \beta_{k-1}$.
- Consider a node v in the search tree (not the root).

  If v is a k-branching, then v's parent is a "Reduce"-node because in a "Branch"-node, for each branch, the formula $F[t_i]$ contains a clause of length $\leq k - 1$.

  Now:
  $\Rightarrow$ branching vector $(1, 2, \ldots, k - 1)$

  Before:
  $(1, 2, \ldots, k - 1, k)$

Runtime Analysis

- Claim: The runtime is $O^*(\alpha_k^n)$, where $\alpha_k = \beta_{k-1}$.
- Consider a node v in the search tree (not the root).

  If v is a k-branching, then v's parent is a "Reduce"-node because in a "Branch"-node, for each branch, the formula $F[t_i]$ contains a clause of length $\leq k-1$.

  Now:
  $\Rightarrow$ branching vector $(1, 2, \ldots, k-1)$

  Before:
  $(1, 2, \ldots, k-1, k)$
  $\beta_k^{k+1} - 2\beta_k^k + 1 = 0$

Runtime Analysis

- Claim: The runtime is $O^*(\alpha_k^n)$, where $\alpha_k = \beta_{k-1}$.
- Consider a node v in the search tree (not the root).

  If v is a k-branching, then v's parent is a "Reduce"-node because in a "Branch"-node, for each branch, the formula $F[t_i]$ contains a clause of length $\leq k-1$.

  Now:
  $\Rightarrow$ branching vector $(1, 2, \ldots, k-1)$
  $\Rightarrow \alpha_k^k - 2\alpha_k^{k-1} + 1 = 0$

  Before:
  $(1, 2, \ldots, k-1, k)$
  $\beta_k^{k+1} - 2\beta_k^k + 1 = 0$

Runtime Analysis

- Claim: The runtime is $O^*(\alpha_k^n)$, where $\alpha_k = \beta_{k-1}$.
- Consider a node v in the search tree (not the root).

  If v is a k-branching, then v's parent is a "Reduce"-node because in a "Branch"-node, for each branch, the formula $F[t_i]$ contains a clause of length $\leq k-1$.

Now:	Before:
$\Rightarrow$ branching vector $(1, 2, \ldots, k-1)$	$(1, 2, \ldots, k-1, k)$
$\Rightarrow \alpha_k^k - 2\alpha_k^{k-1} + 1 = 0$	$\beta_k^{k+1} - 2\beta_k^k + 1 = 0$

Runtime Analysis

- Claim: The runtime is $O^*(\alpha_k^n)$, where $\alpha_k = \beta_{k-1}$.
- Consider a node v in the search tree (not the root).

  If v is a k-branching, then v's parent is a "Reduce"-node because in a "Branch"-node, for each branch, the formula $F[t_i]$ contains a clause of length $\leq k-1$.

  Now:
  $\Rightarrow$ branching vector $(1, 2, \ldots, k-1)$

  Before:
  $(1, 2, \ldots, k-1, k)$

$$\underbrace{\Rightarrow \alpha_k^{k} - 2\alpha_k^{k-1} + 1 = 0 \qquad \beta_k^{k+1} - 2\beta_k^{k} + 1 = 0}_{\Rightarrow\ \alpha_k = \beta_{k-1}}$$

Runtime Analysis

- Claim: The runtime is $O^*(\alpha_k^n)$, where $\alpha_k = \beta_{k-1}$.
- Consider a node v in the search tree (not the root).

  If v is a k-branching, then v's parent is a "Reduce"-node
  because in a "Branch"-node, for each branch,
  the formula $F[t_i]$ contains a clause of length $\le k-1$.

  Now:
  $\Rightarrow$ branching vector $(1, 2, \ldots, k-1)$

  Before:
  $(1, 2, \ldots, k-1, k)$

  $$\underbrace{\Rightarrow \alpha_k^k - 2\alpha_k^{k-1} + 1 = 0 \qquad\qquad \beta_k^{k+1} - 2\beta_k^k + 1 = 0}$$

  $$\Rightarrow \alpha_k = \beta_{k-1}$$

- $\beta_1 = 1, \beta_2 < 1.6181, \beta_3 < 1.8393, \beta_4 < 1.9276, \beta_5 < 1.9660$

Runtime Analysis

- Claim: The runtime is $O^*(\alpha_k^n)$, where $\alpha_k = \beta_{k-1}$.

- Consider a node v in the search tree (not the root).

  If v is a k-branching, then v's parent is a "Reduce"-node because in a "Branch"-node, for each branch, the formula $F[t_i]$ contains a clause of length $\leq k-1$.

  Now:
  $\Rightarrow$ branching vector $(1, 2, \ldots, k-1)$

  Before:
  $(1, 2, \ldots, k-1, k)$

  $$\Rightarrow \underbrace{\alpha_k^{k} - 2\alpha_k^{k-1} + 1 = 0 \qquad \beta_k^{k+1} - 2\beta_k^{k} + 1 = 0}_{\Rightarrow \ \alpha_k = \beta_{k-1}}$$

- $\beta_1 = 1, \beta_2 < 1.6181, \beta_3 < 1.8393, \beta_4 < 1.9276, \beta_5 < 1.9660$

$\Rightarrow \alpha_2 = 1, \alpha_3 < 1.6181, \alpha_4 < 1.8393, \alpha_5 < 1.9276, \alpha_6 < 1.9660$

$\square$