Julius-Maximilians-UNIVERSITÄT WÜRZBURG
Lehrstuhl für INFORMATIK I Algorithmen & Komplexität
Institut für Informatik

# Exact Algorithms

Sommer Term 2020

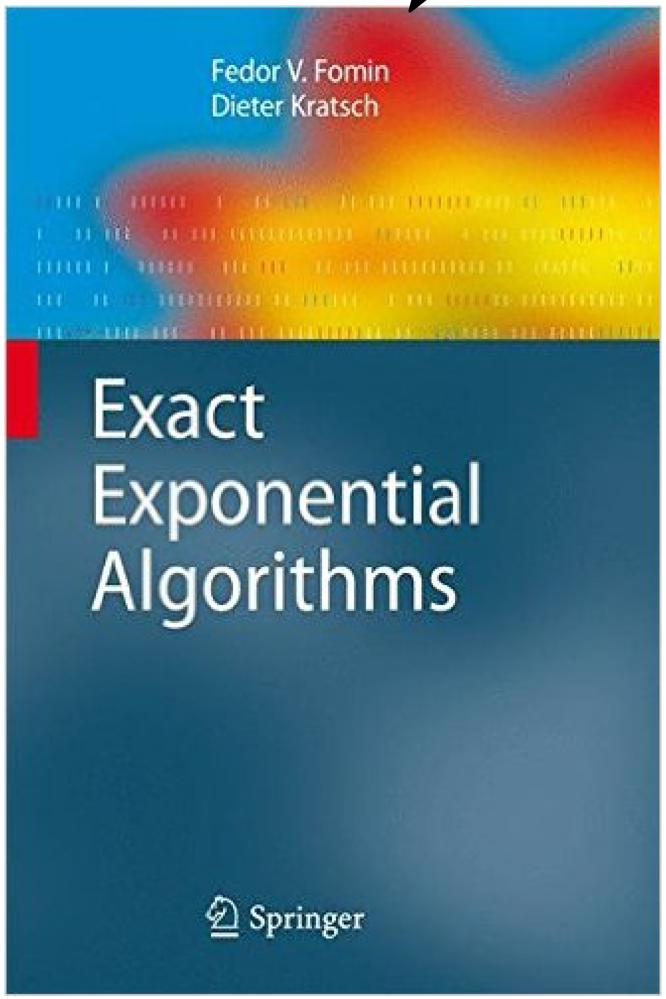
## Lecture 1. Introduction & Two Examples

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

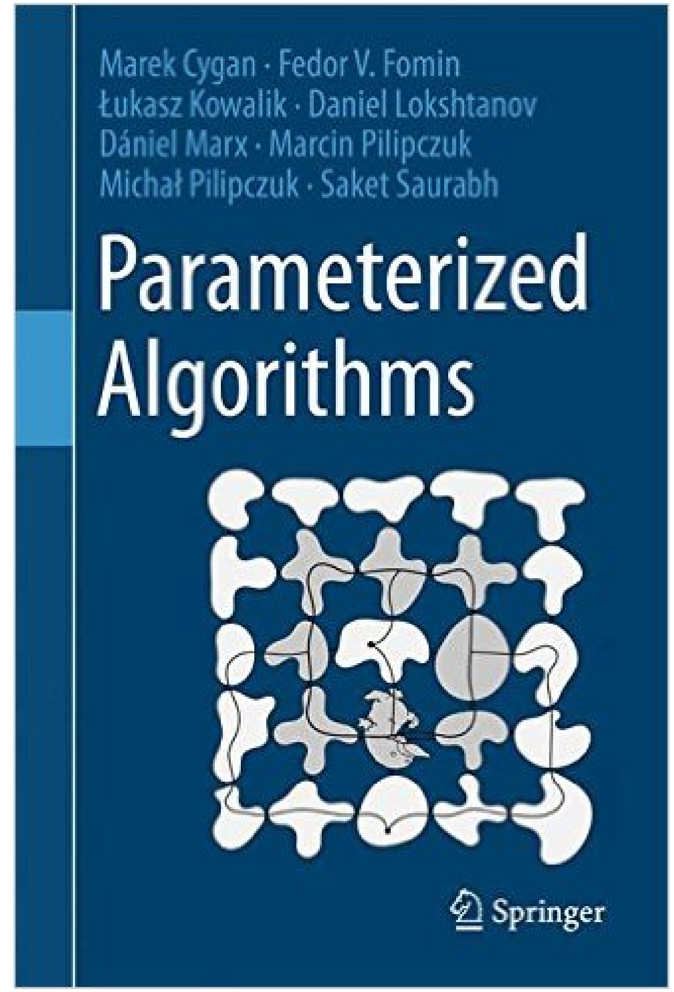
Alexander Wolff
Lehrstuhl für Informatik I

# Textbooks

This Lecture: Chapter 1




Fedor Fomin & Dieter Kratsch:
Exact Exponential Algorithms
Springer 2010

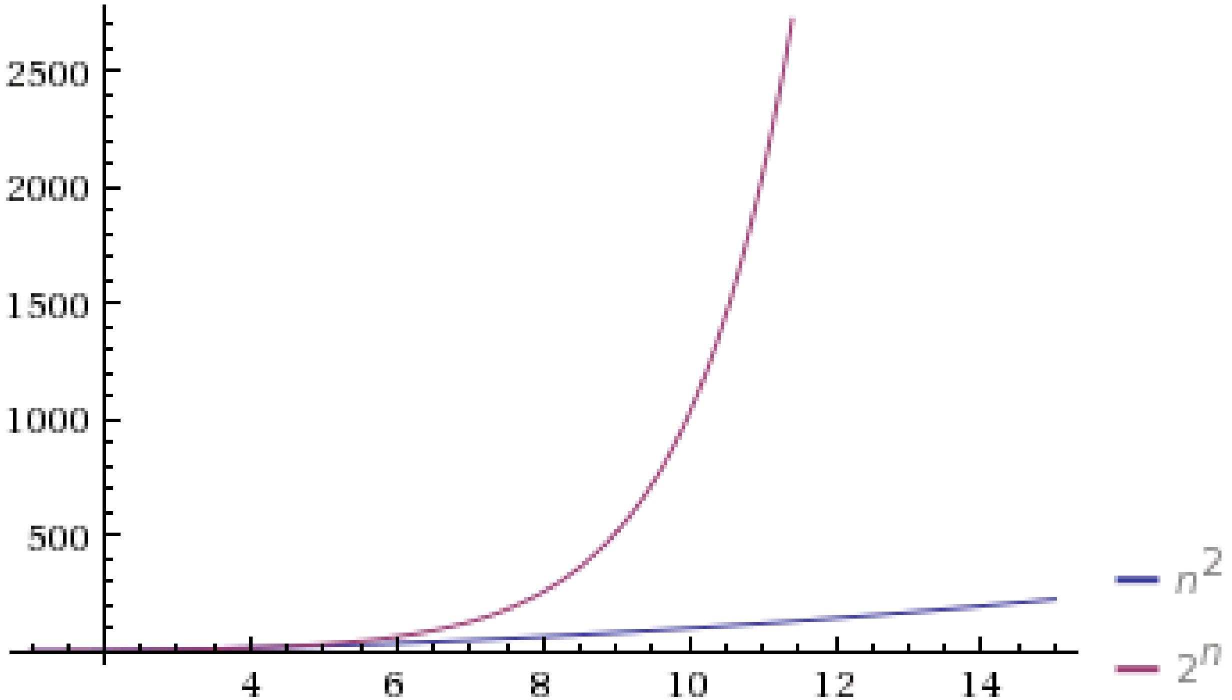



Marek Cygan et al.:
Parameterized Algorithms
Springer 2015

# Motivation

Efficient vs. inefficient algorithms

$\rightsquigarrow$ polynomial vs. super-polynomial algorithms

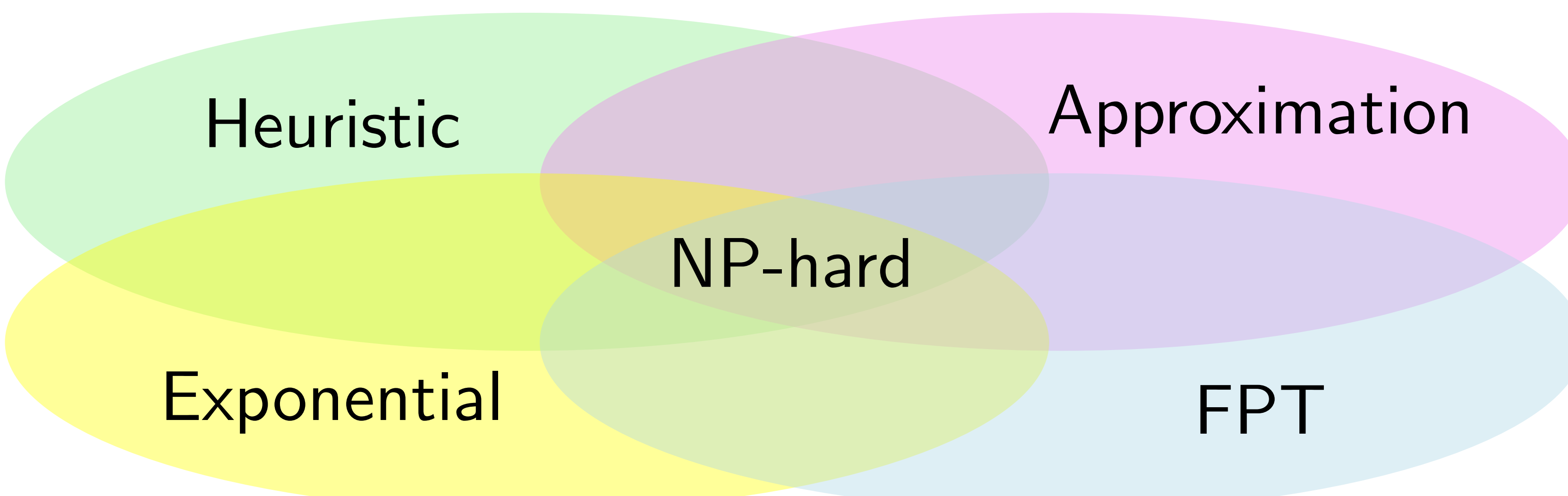


# Why Consider Exponential-Time Algorithms?

Many important (practical) problems are NP-hard!

How to deal with NP-hard problems?

- Sacrifice optimality for speed
  - heuristics (simulated annealing, tabu search)
  - approximation algorithms (Christofides' algorithm)
- Optimal Solutions — This Course!
  - exact exponential-time algorithms
  - fine-grained analysis (parameterized) algorithms



# Motivation: Exact Exponential Algorithms

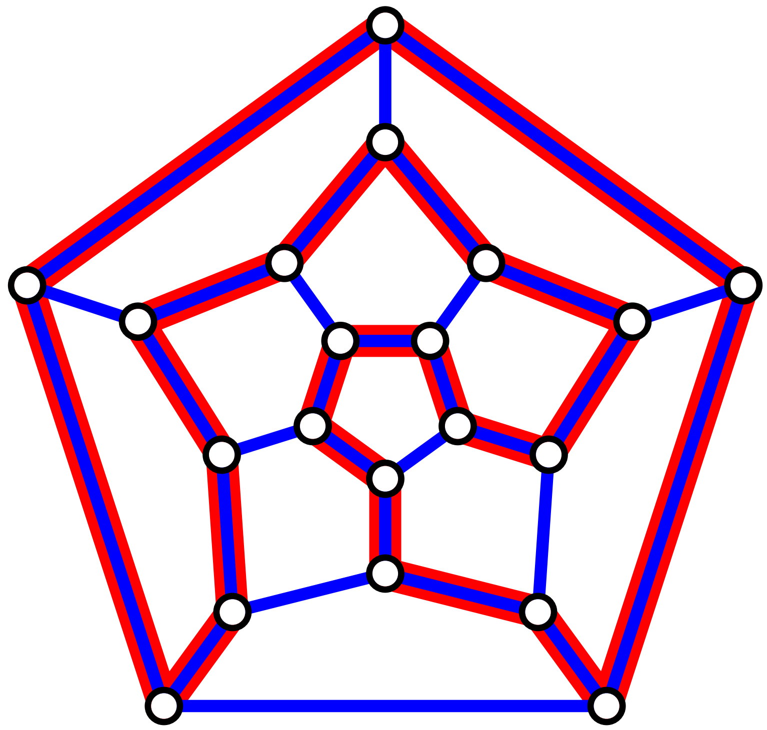
- Can be "fast" for **medium-sized** instances:

  $\rightsquigarrow$ e.g.: $n^4 > 1.2^n$ for $n \leq 100$

  $\rightsquigarrow$ e.g.: TSP solvable exactly for $n \leq 2000$ and specialized instances with $n \leq 85900$

  $\rightsquigarrow$ "hidden" constants in polynomial time algorithms: $2^{100} \cdot n > 2^n$ for $n \leq 100$

- Theoretical interest!



# Typical Results

- Idea (simplified): find exact algorithms that are faster than *brute-force* (trivial) approaches.

- Typical results for a (hypothetical) NP-hard problem:

| Approach | Runtime in $O$-Notation | $O^*$-Notation |
|---|---|---|
| Brute-Force | $O(2^n)$ | $O^*(2^n)$ |
| Algorithm A | $O(1.5^n \cdot n)$ | $O^*(1.5^n)$ |
| Algorithm B | $O(1.4^n \cdot n^2)$ | $O^*(1.4^n)$ |

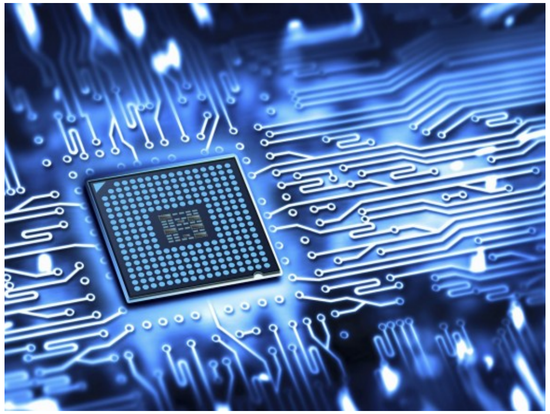
$$O(1.4^n \cdot n^2) \subsetneq O(1.5^n \cdot n) \subsetneq O(2^n)$$

- Neglect polynomial factors (exponential part dominates)!

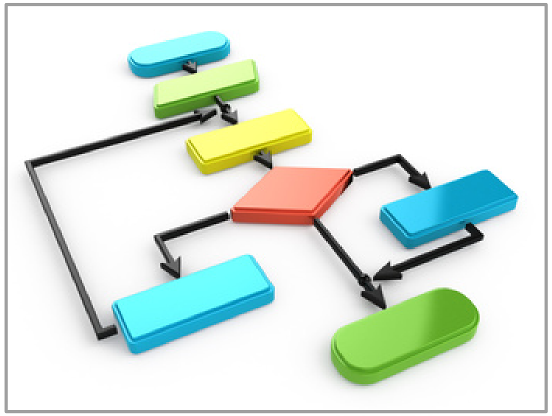
$$f \in O^*(g) \;\Leftrightarrow\; \exists \text{ polynomial } p\text{: } f \in O(g \cdot p)$$

# Faster Hardware vs. Better Algorithms

Suppose an algorithm uses $a^n$ steps, and we have a fixed amount of time to run it.



- Improving hardware by a constant factor $c$ only **adds a constant** (relative to $c$) to the maximum size $n_0$ of solvable instances.
- In contrast, reducing the base of the runtime to $b < a$ results in a **multiplicative increase** of $n_0$!



Why?

Hardware speedup: $a^{n_0'} = c \cdot a^{n_0} \Rightarrow n_0' = n_0 + \log_a c$

Base reduction: $b^{n_0'} = a^{n_0} \Rightarrow n_0' = n_0 \cdot \log_b a$

# Traveling Salesperson Problem (TSP)

**Input:** Complete directed graph $G = (V, E)$ with $n$ vertices and edge weights $c\colon E \to \mathbb{Q}_{\geq 0}$

**Output:** A Hamiltonian cycle $C = (v_1, \ldots, v_n, v_{n+1} = v_1)$ of $G$, of minimum weight $\sum_{i=1}^{n} c(v_i, v_{i+1})$.

Brute-Force?

- Each tour is a permutation of the vertices.
- Pick a permutation with the smallest weight.

Runtime: $\Theta(n! \cdot n) = n \cdot 2^{\Theta(n \log n)}$

# Bellman–Held–Karp Algorithm

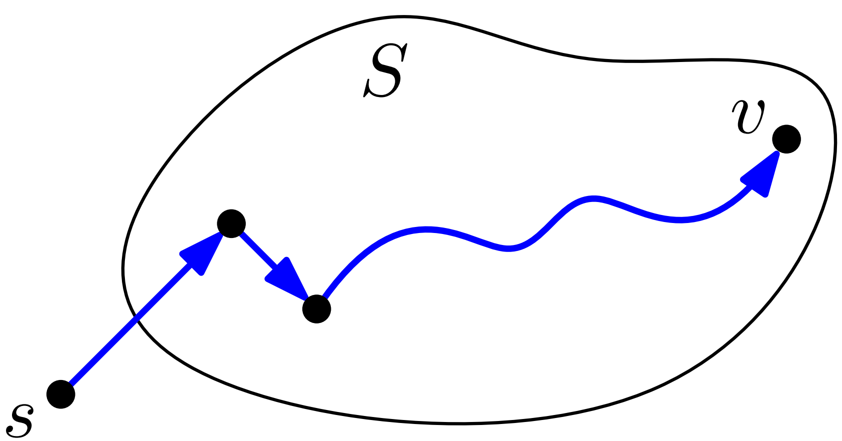
Technique: Dynamic Programming!

*Reuse optimal substructures!*

Select any starting vertex $s \in V$.

For each $S \subseteq V - s := V \setminus \{s\}$ and $v \in S$:

$\mathrm{OPT}[S, v] :=$ length of the shortest $s$–$v$ path that visits precisely the vertices of $S \cup \{s\}$.




Richard M. Karp

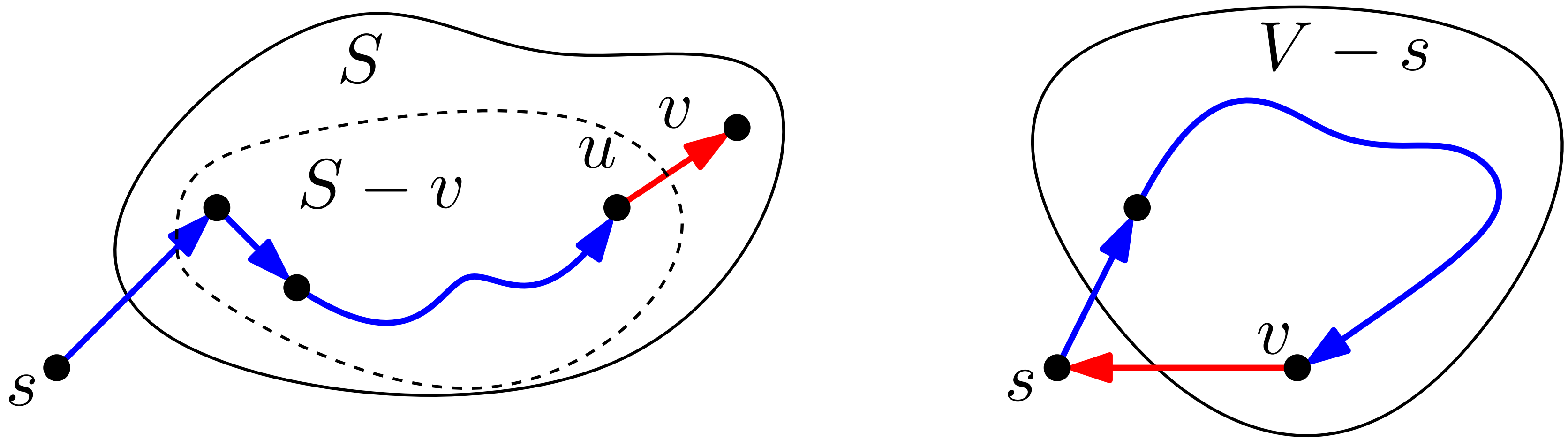
Richard E. Bellman

# Bellman–Held–Karp Algorithm

The base case, $S = \{v\}$, is easy: $\text{OPT}[S, v] = c(s, v)$.

When $|S| \geq 2$, we compute $\text{OPT}[S, v]$ recursively:

$$\text{OPT}[S, v] = \min\{\, \text{OPT}[S - v, u] + c(u, v) \mid u \in S - v \,\}$$




After computing $\text{OPT}[S, v]$ for each $S \subseteq V - s$,
the optimal solution is easily obtained as follows:

$$\text{OPT} = \min\{\, \text{OPT}[V - s, v] + c(v, s) \mid v \in V - s \,\} \qquad \square$$

# Pseudocode for the Dynamic Program

```
Algorithm Bellmann–Held–Karp(G, c)
  foreach v ∈ V − s do
    OPT[{v}, v] = c(s, v)
  for j = 2 to n − 1 do
    foreach S ⊆ V − s with |S| = j do
      foreach v ∈ S do
        OPT[S, v] = min{ OPT[S − v, u] + c(u, v) | u ∈ S − v }
  return min{ OPT[V − s, v] + c(v, s) | v ∈ V − s }
```

**Runtime:** The innermost loop has $O(2^n \cdot n)$ iterations, each taking $O(n)$ time.
In total: $O(2^n \cdot n^2) = O^*(2^n)$.

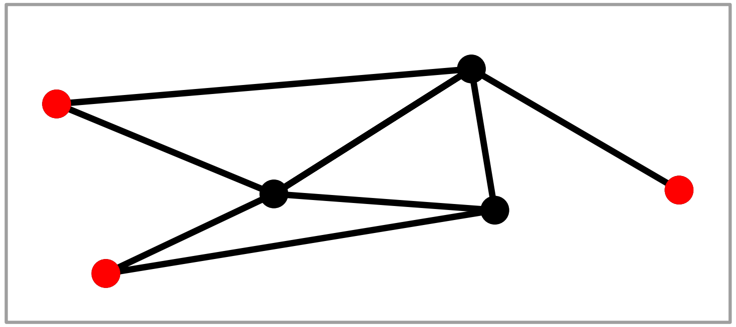
**Space usage:** $\Theta(2^n \cdot n) = \Theta^*(2^n)$

A shortest tour can be produced by backtracking the DP table (as usual). Compare: $O^*(2^n)$ with $2^{O(n \log n)}$ for Brute-Force!

# Maximum Independent Set (MIS)

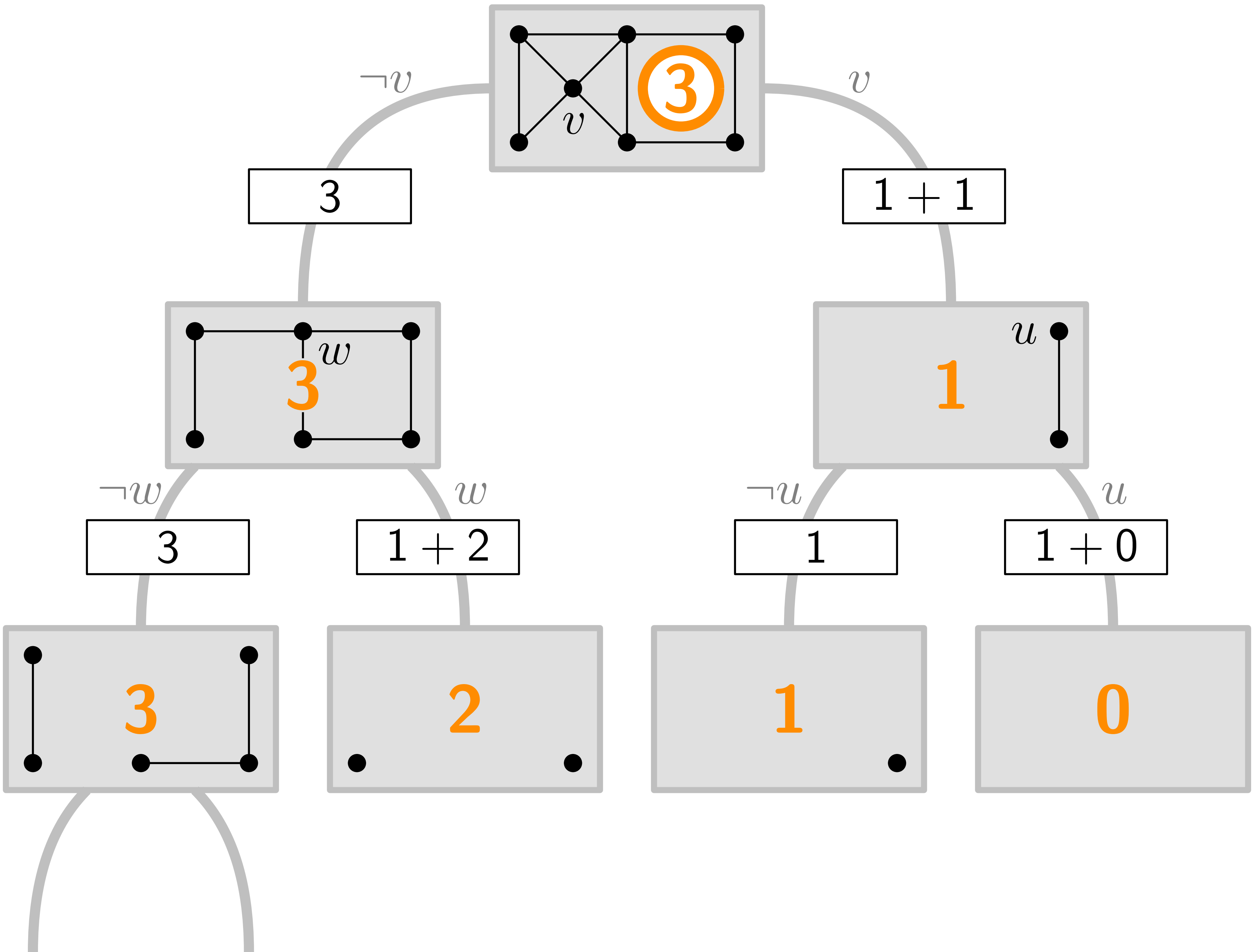
**Input:** Graph $G = (V, E)$ with $n$ vertices.

**Output:** Maximum size *independent* set, i.e., a largest set $U \subseteq V$, such that no pair of vertices in $U$ are adjacent in $G$.



**Brute Force?** Try all subsets of $V \Rightarrow$ runtime $O(2^n \cdot n)$.

Algorithm NaiveMIS(graph $G = (V, E)$)
  **if** $V = \emptyset$ **then**
    **return** 0
  $v \leftarrow$ arbitrary vertex in $V(G)$
  **return** max$\{1 +$ NaiveMIS$(G - N(v) - \{v\})$, NaiveMIS$(G - \{v\})\}$

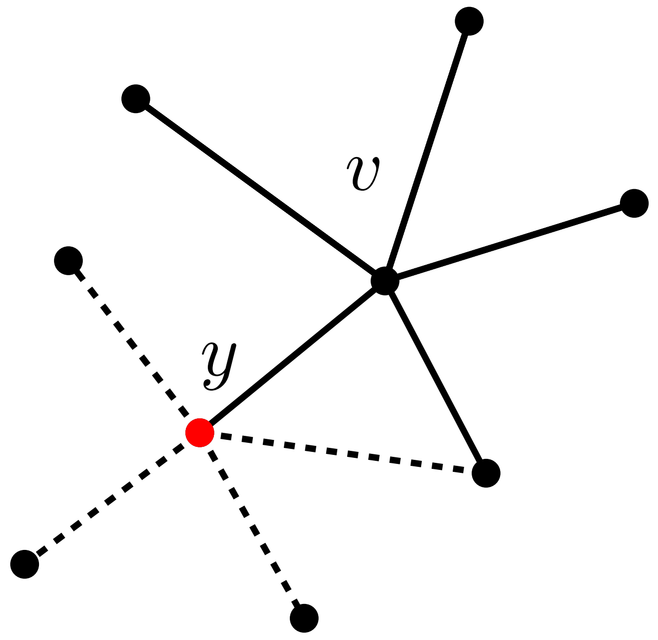

¬v
v
3
v
1 + 1
3
w
u
1
¬w
w
¬u
u
3
1 + 2
1
1 + 0
3
2
1
0

# Observation

**Lemma.** Let $U$ be a *maximum* independent set in $G$.
Then, for each vertex $v \in V$:

(i) $v \in U \Rightarrow N(v) \cap U = \emptyset$
(ii) $v \notin U \Rightarrow |N(v) \cap U| \geq 1$

Thus, $N[v] := N(v) \cup \{v\}$ contains some $y \in U$,
and no other vertex of $N[y]$ is in $U$.



# Smarter Branching Algorithm

```
Algorithm MIS(G)
  if V = ∅ then
  └ return 0
  v ← vertex of minimum degree in V(G)
  return 1 + max{MIS(G − N[y]) | y ∈ N[v]}
```

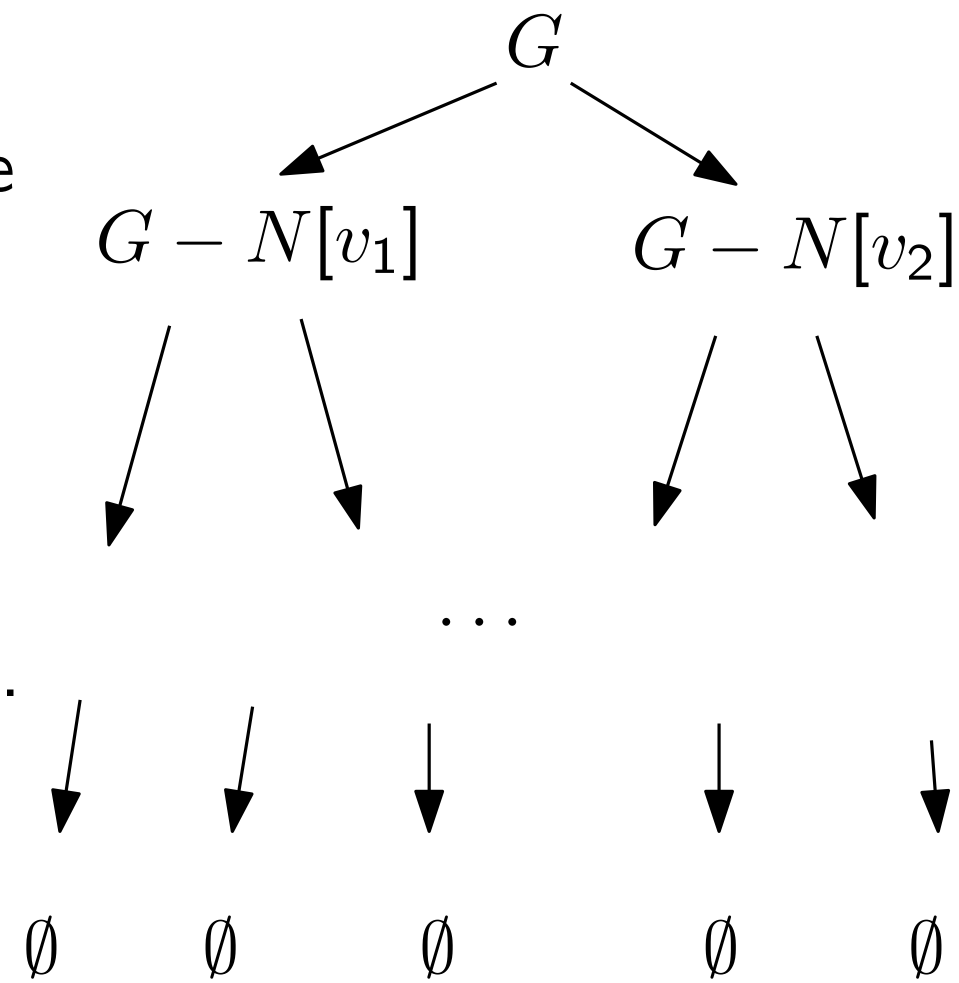
**Correctness:** follows from the previous lemma.

We will now prove a runtime of $O^*(3^{n/3}) = O^*(1.4423^n)$

# Runtime

Execution corresponds to a *search tree* whose nodes are labeled with the input of the respective recursive call.

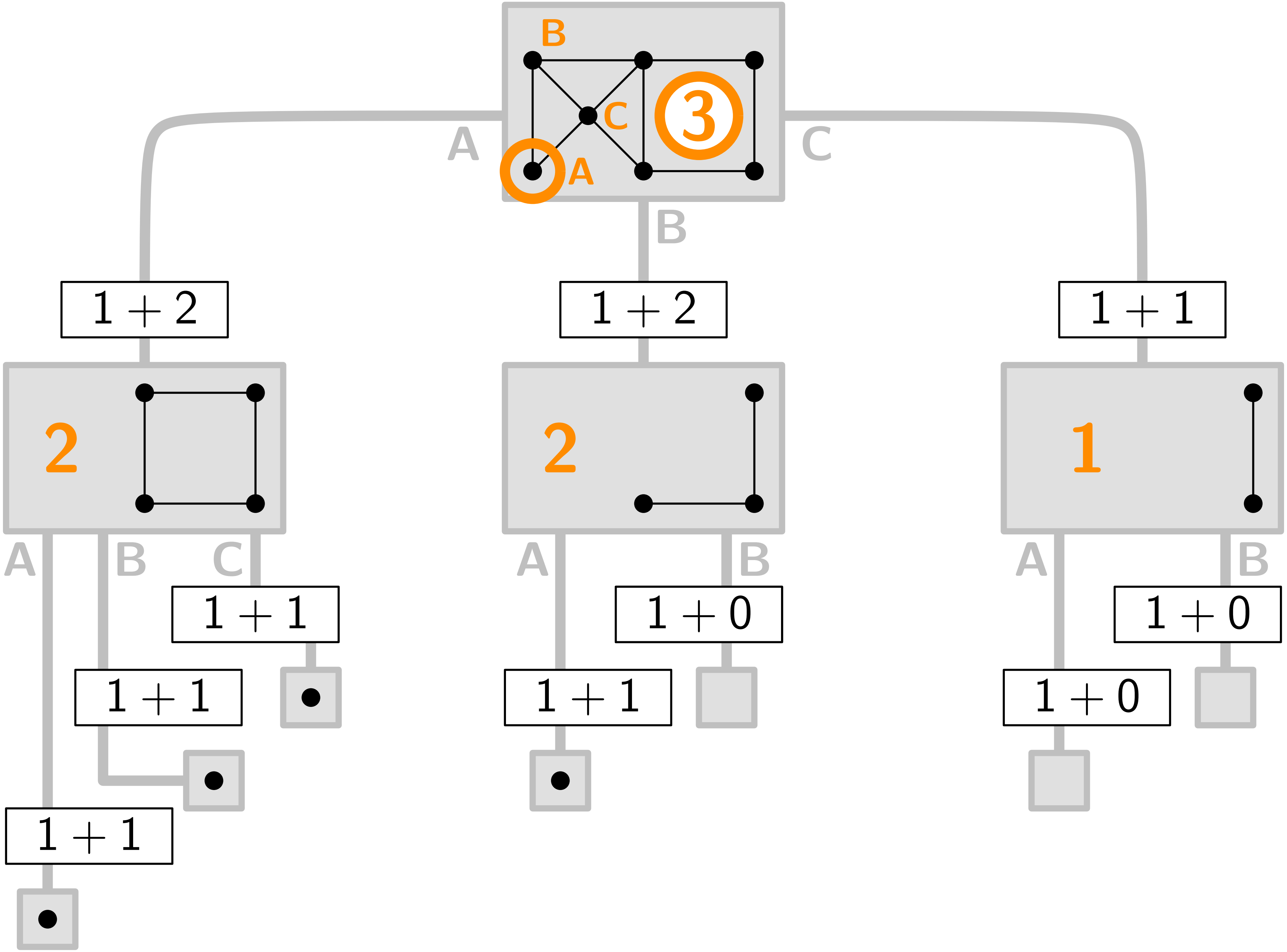
Let $B(n)$ be the maximum number of leaves of a search tree for a graph with $n$ vertices.




The search tree has height $\leq n$.

$\Rightarrow$ Algorithm runs in time $T(n) \in O^*(nB(n)) = O^*(B(n))$.

Let's consider an example run.


B
C
A
3
A
B
C
1 + 2
1 + 2
1 + 1
2
2
1
A
B
C
A
B
A
B
1 + 1
1 + 0
1 + 0
1 + 1
1 + 1
1 + 0
1 + 1

# Runtime Analysis

For a worst-case $n$-vertex graph $G$ $(n \geq 1)$:

$$B(n) \leq \sum_{y \in N[v]} B(n - (\deg(y) + 1))$$
$$\leq (\deg(v) + 1) \cdot B(\ n - (\deg(v) + 1)\ )\,,$$

where $v$ is a minimum-degree vertex of $G$.

For the second inequality, we still need to argue that $B$ is *monotone*, that is, $B(n') \leq B(n)$ for any $n' \leq n$.

This is not difficult: Let $G'$ be a graph with $n'$ vertices and a search tree with the maximum number of leaves, $B(n')$.

Add to $G'$ $n - n'$ independent vertices.

This yields an $n$-vertex graph witnessing that $B(n) \geq B(n')$.

# Runtime Analysis (cont'd)

Recall: $B(n) \leq (\deg(v)+1) \cdot B(\ n - (\deg(v)+1)\ )$

We proceed by induction to show that $B(n) \leq 3^{n/3}$.

Base case: $B(0) = 1 \leq 3^{0/3}$

Hypothesis: for $n \geq 1$, set $s = \deg(v) + 1$ in

Thus,

$$B(n) \leq s \cdot B(n-s) \leq s \cdot 3^{(n-s)/3} = \frac{s}{3^{s/3}} \cdot 3^{n/3} \leq 3^{n/3} \quad ✓$$

$$B(n) \in O^*(\sqrt[3]{3}^n) \subset O^*(1.44225^n)$$