Coping with NP-completeness

Q. Suppose I need to solve an **NP**-complete problem. What should I do?
A. Theory says you're unlikely to find poly-time algorithm.

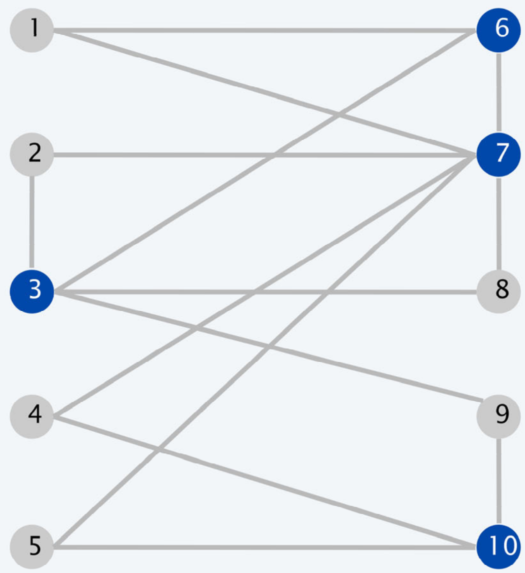
Must sacrifice one of three desired features.

- Solve problem to optimality.
- Solve problem in polynomial time.
- Solve arbitrary instances of the problem.

This lecture. Solve some special cases of **NP**-complete problems.

Vertex cover

Given a graph $G = (V, E)$ and an integer k, is there a subset of vertices $S \subseteq V$ such that $|S| \leq k$, and for each edge (u, v) either $u \in S$ or $v \in S$ or both?



S = { 3, 6, 7, 10 } is a vertex cover of size k = 4

Finding small vertex covers

Q. VERTEX-COVER is **NP**-complete. But what if k is small?

$O(n^k \cdot kn)$

n nodes

k-size subsets

$\binom{n}{k} = \#\ k$-subsets

Brute force. $O(k\, n^{k+1})$.

- Try all $C(n, k) = O(n^k)$ subsets of size k.
- Takes $O(k\, n)$ time to check whether a subset is a vertex cover.

??

n^K

Goal. Limit exponential dependency on k, say to $O(2^k\, k\, n)$.

Ex. $n = 1{,}000$, $k = 10$.
Brute. $k\, n^{k+1} = 10^{34} \Rightarrow$ infeasible.
Better. $2^k\, k\, n = 10^7 \Rightarrow$ feasible.

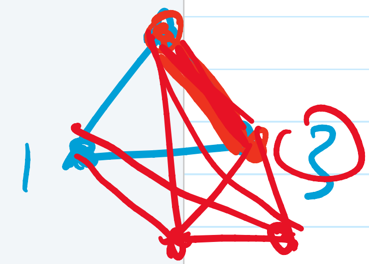
Remark. If k is a constant, then the algorithm is poly-time;
if k is a small constant, then it's also practical.

Finding small vertex covers

Claim. Let (u, v) be an edge of G. G has a vertex cover of size $\leq k$ iff at least one of $G - \{ u \}$ and $G - \{ v \}$ has a vertex cover of size $\leq k - 1$.

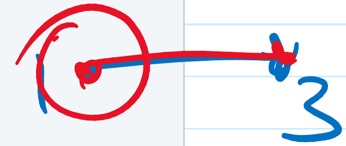
delete v and all incident edges

2 = 4 $\checkmark$ k=2



Pf. $\Rightarrow$

- Suppose G has a vertex cover S of size $\leq k$.
- S contains either u or v (or both). Assume it contains u.
- $S - \{ u \}$ is a vertex cover of $G - \{ u \}$.



.3

Pf. $\Leftarrow$

- Suppose S is a vertex cover of $G - \{ u \}$ of size $\leq k - 1$.
- Then $S \cup \{ u \}$ is a vertex cover of G. ▪

n − 1

Claim. If G has a vertex cover of size k, it has $\leq k\,(n - 1)$ edges.

Pf. Each vertex covers at most $n - 1$ edges. ▪

k such vertices
≤ k·(n−1) edges

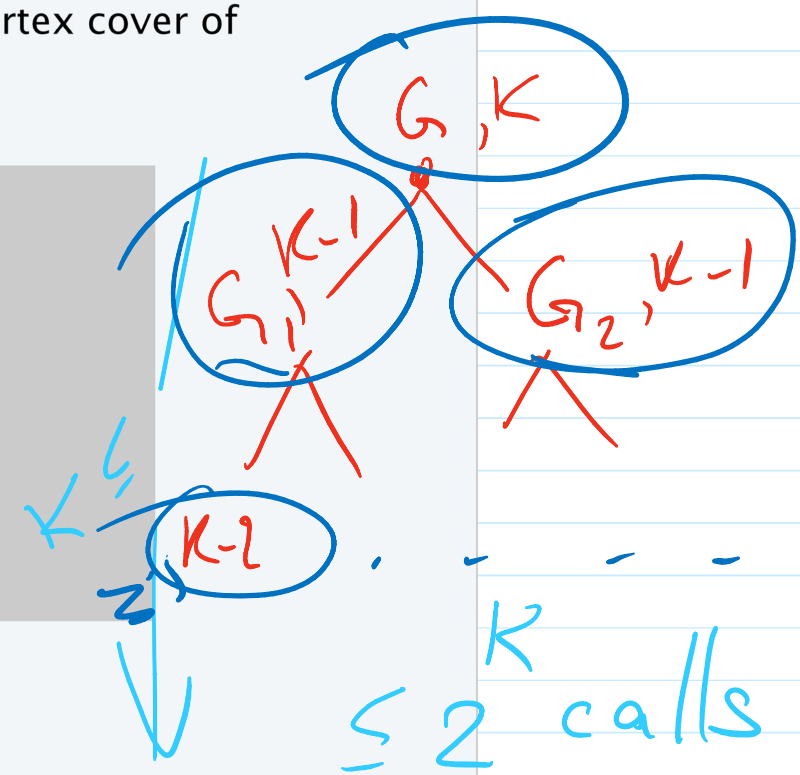
Finding small vertex covers: algorithm

Claim. The following algorithm determines if G has a vertex cover of

Claim. The following algorithm determines if G has a vertex cover of size $\leq k$ in $O(2^k\, kn)$ time.

```
Vertex-Cover(G, k) {
   if (G contains no edges)    return true
   if (G contains ≥ kn edges) return false

   let (u, v) be any edge of G
   a = Vertex-Cover(G - {u}, k-1)
   b = Vertex-Cover(G - {v}, k-1)
   return a or b
}
```

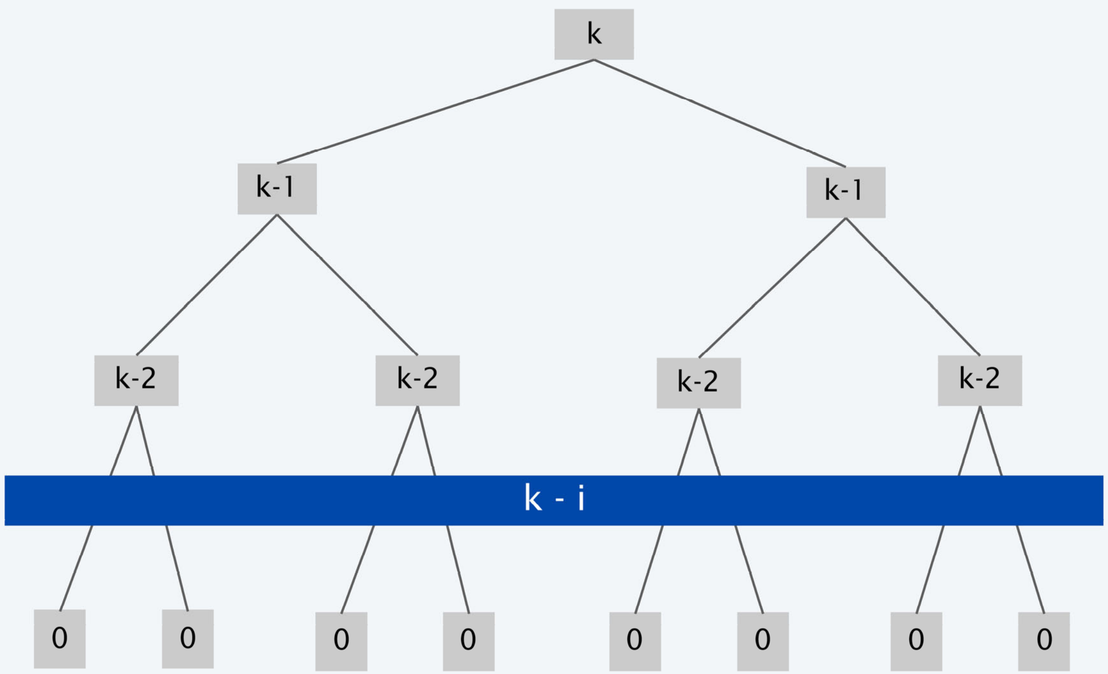


Pf.

- Correctness follows from previous two claims.
- There are $\leq 2^{k+1}$ nodes in the recursion tree; each invocation takes $O(kn)$ time. ▪

Finding small vertex covers: recursion tree

$$T(n,k) \le \begin{cases} c & \text{if } k = 0 \\ cn & \text{if } k = 1 \\ 2T(n,k-1) + ckn & \text{if } k > 1 \end{cases} \quad \Rightarrow \quad T(n,k) \le 2^k c\, k\, n$$



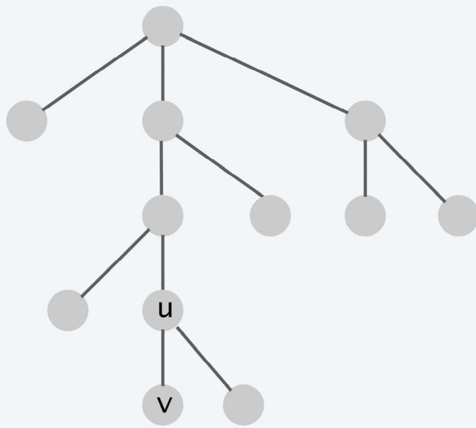
k

Independent set on trees

Independent set on trees. Given a tree, find a maximum cardinality subset of nodes such that no two share an edge.

Fact. A tree on at least two nodes has at least two leaf nodes.

degree = 1



Key observation. If v is a leaf, there exists a maximum size independent set containing v.

Pf. (exchange argument)

- Consider a max cardinality independent set S.
- If $v \in S$, we're done.
- If $u \notin S$ and $v \notin S$, then $S \cup \{ v \}$ is independent $\Rightarrow S$ not maximum.
- If $u \in S$ and $v \notin S$, then $S \cup \{ v \} - \{ u \}$ is independent. ▪

Independent set on trees: greedy algorithm

Theorem. The following greedy algorithm finds a maximum cardinality independent set in forests (and hence trees).

```
Independent-Set-In-A-Forest(F) {
   S ← ϕ
   while (F has at least one edge) {
      Let e = (u, v) be an edge such that v is a leaf
      Add v to S
      Delete from F nodes u and v, and all edges
      incident to them.
   }
   return S ∪ { all remaining nodes }
}
```

Pf. Correctness follows from the previous key observation. ▪

Remark. Can implement in $O(n)$ time by considering nodes in postorder.

Weighted independent set on trees

Before: $w_v = 1$ (special)

Weighted independent set on trees. Given a tree and node weights $w_v > 0$, find an independent set S that maximizes $\Sigma_{v \in S}\, w_v$.

Observation. If (u, v) is an edge such that v is a leaf node, then either OPT includes u or OPT includes all leaf nodes incident to u.

u = leaf

$Opt_{in}(u) = w_u$

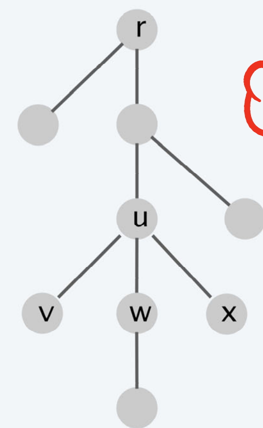
$Opt_{out}(u) = 0$

Dynamic programming solution. Root tree at some node, say r.

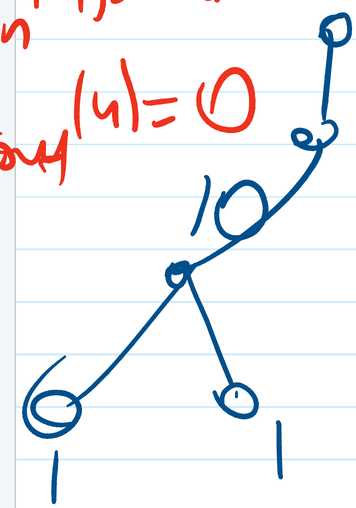
- $OPT_{in}(u)$ = max weight independent set of subtree rooted at u, containing u.
- $OPT_{out}(u)$ = max weight independent set of subtree rooted at u, not containing u.

$$OPT_{in}(u) = w_u + \sum_{v \in \text{children}(u)} OPT_{out}(v)$$

$$OPT_{out}(u) = \sum_{v \in \text{children}(u)} \max\left\{OPT_{in}(v),\ OPT_{out}(v)\right\}$$



children(u) = { v, w, x }



Weighted independent set on trees: dynamic programming algorithm

Theorem. The dynamic programming algorithm finds a maximum weighted independent set in a tree in $O(n)$ time.

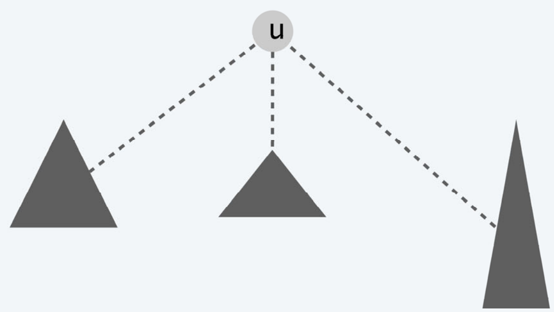
can also find independent set itself (not just value)

```
Weighted-Independent-Set-In-A-Tree(T) {
   Root the tree at a node r
   foreach (node u of T in postorder) {
      if (u is a leaf) {
         M_in [u] = w_u
         M_out[u] = 0
      }
      else {
         M_in [u] = w_u + Σ_{v∈children(u)} M_out[v]
         M_out[u] = Σ_{v∈children(u)} max(M_in[v], M_out[v])
      }
   }
   return max(M_in[r], M_out[r])
}
```

ensures a node is visited after all its children

Context

Independent set on trees. This structured special case is tractable because we can find a node that breaks the communication among the subproblems in different subtrees.



"tree-like graphs"

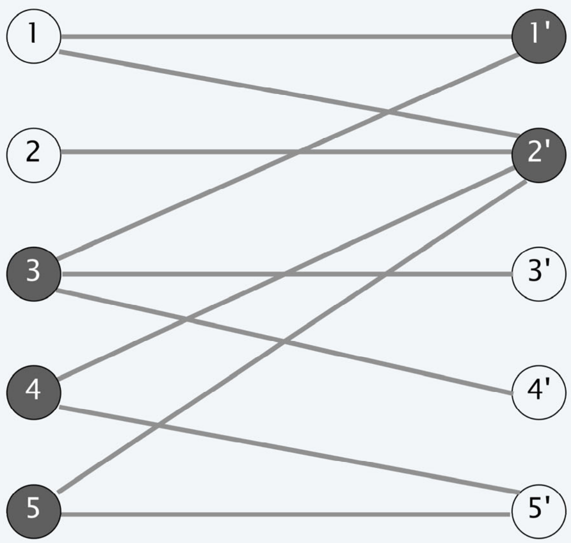
see Chapter 10.4
(but proceed with caution)

Graphs of bounded tree width. Elegant generalization of trees that:

- Captures a rich class of graphs that arise in practice.
- Enables decomposition into independent pieces.

Vertex cover

Given a graph $G = (V, E)$ and an integer k, is there a subset of vertices $S \subseteq V$ such that $|S| \leq k$, and for each edge (u, v) either $u \in S$ or $v \in S$ or both?

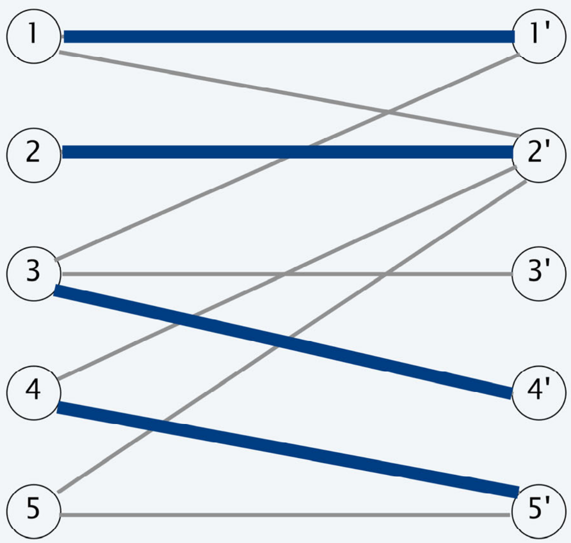


vertex cover S = { 3, 4, 5, 1', 2' }

Vertex cover and matching

Weak duality. Let M be a matching, and let S be a vertex cover.
Then, $|M| \leq |S|$.

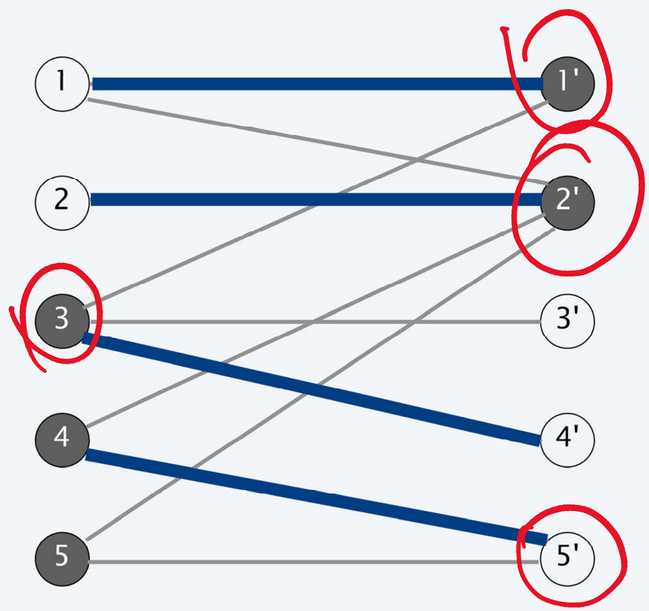
Pf. Each vertex can cover at most one edge in any matching.



matching M: 1-1', 2-2', 3-4', 4-5'

Vertex cover in bipartite graphs: König-Egerváry Theorem

Theorem. [König-Egerváry] In a bipartite graph, the max cardinality of a matching is equal to the min cardinality of a vertex cover.



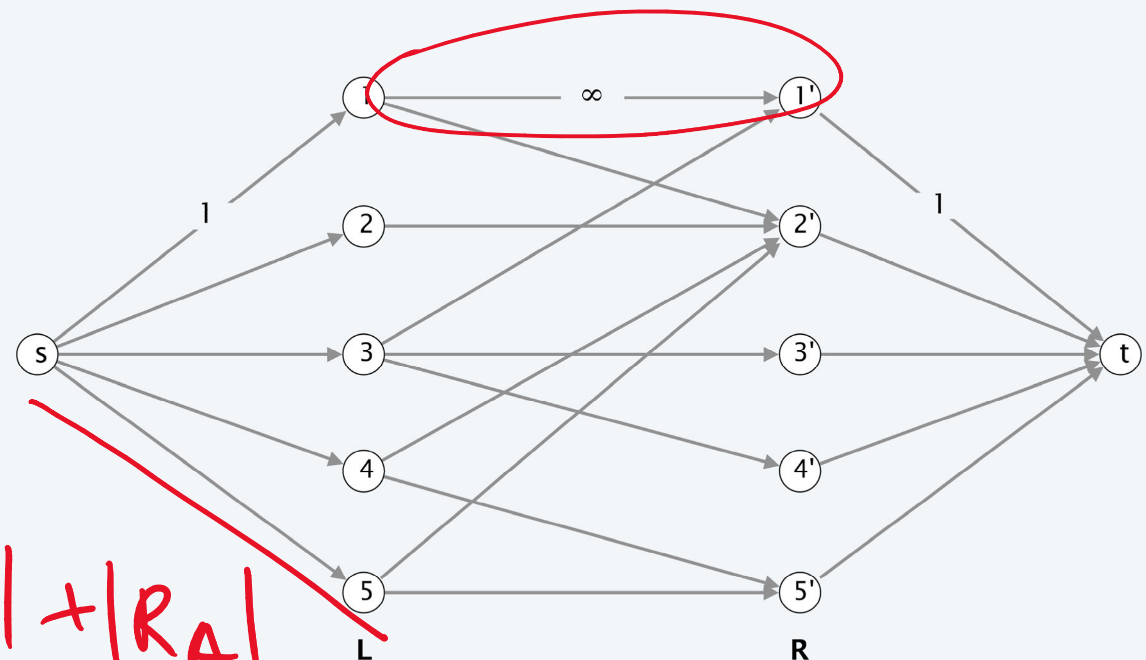
matching M: 1–1', 2–2', 3–4', 4–5'

vertex cover S = { 3, 4, 5, 1', 2' }

Proof of König-Egerváry theorem

Theorem. [König-Egerváry] In a bipartite graph, the max cardinality of a matching is equal to the min cardinality of a vertex cover.

- Suffices to find matching M and cover S such that $|M| = |S|$.
- Formulate max flow problem as for bipartite matching.
- Let M be max cardinality matching and let (A, B) be min cut.



$|L_B| + |R_A|$

Proof of König-Egerváry theorem

Theorem. [König-Egerváry] In a bipartite graph, the max cardinality of a matching is equal to the min cardinality of a vertex cover.

- Suffices to find matching M and cover S such that $|M| = |S|$.
- Formulate max flow problem as for bipartite matching.
- Let M be max cardinality matching and let (A, B) be min cut.
- Define $L_A = L \cap A$, $L_B = L \cap B$, $R_A = R \cap A$, $R_B = R \cap B$.

- Claim 1. $S = L_B \cup R_A$ is a vertex cover.
  - consider $(u, v) \in E$
  - $u \in L_A, v \in R_B$ impossible since infinite capacity
  - thus, either $u \in L_B$ or $v \in R_A$ or both

- Claim 2. $|M| = |S|$.
  - max-flow min-cut theorem $\Rightarrow |M| = cap(A, B)$
  - only edges of form (s, u) or (v, t) contribute to $cap(A, B)$
  - $|M| = cap(A, B) = |L_B| + |R_A| = |S|$. ▪

Review

(randomized, quantum) efficient algorithms (exact, approximation)

Remarks:

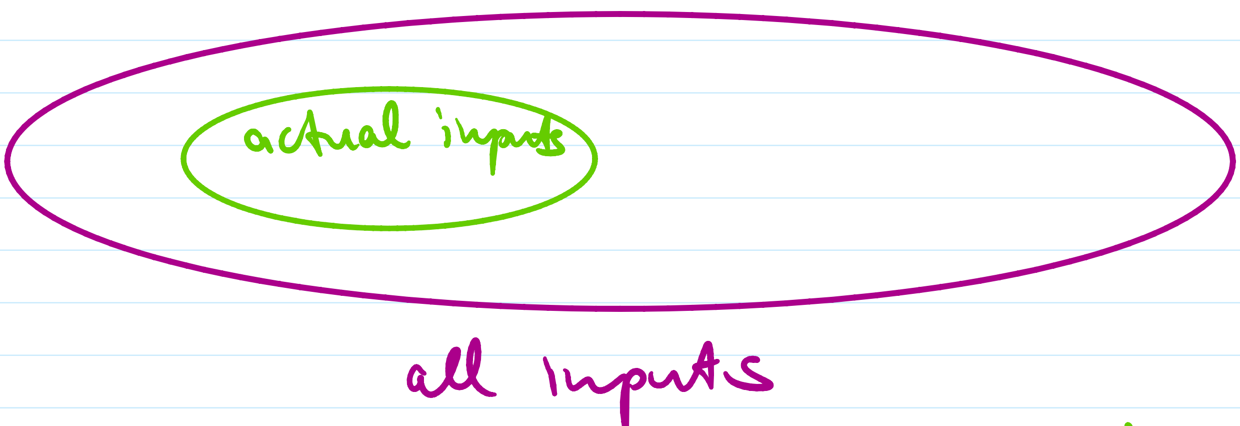
- The classification of algorithms into Greedy, DP, etc. is just for convenience. It's OK to create hybrid algorithms.

- **NP-complete** problems provide a source of many natural problems we **don't know how to solve** with **efficient** algorithms. A common belief is that no such algorithms exist (i.e., that P is not equal to NP), but we don't really know!

- **Randomized** (and quantum) polytime algorithms extend our notion of efficient algorithms (from the usual deterministic polytime algorithms).

- Our **ideal algorithm** for a given problem is
  - fast (polytime), and
  - correct on all inputs of that problem.

In practice:



it would suffice to have algos: fast & correct on only "actual inputs"

Challenge: (1) Formalize "actual inputs"

Challenge:

(1) Formalize "actual inputs"

(2) Design algos & prove they work on "actual inputs".

(analyze Heuristics used in practice)