6,8:15

MA/CSSE 473
Day 37

Student Questions

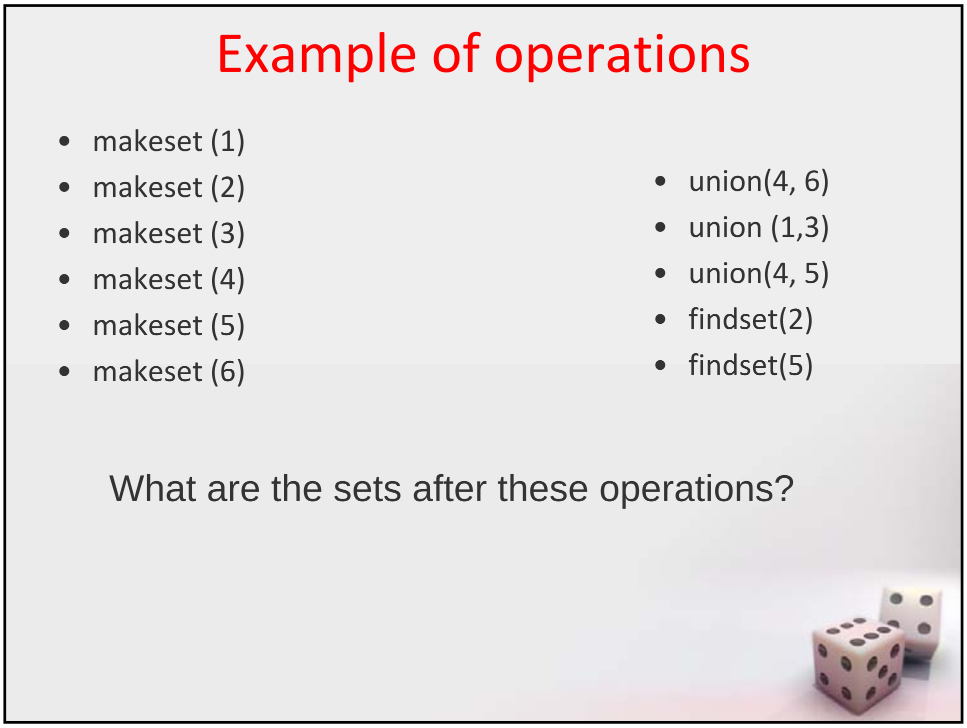
Kruskal data structures

Disjoint Set ADT

Complexity intro

Data Structures for Kruskal

- A sorted list of edges (edge list, not adjacency list)
  - Edge e has fields `e.v` and `e.w` (#s of its end vertices)
- Disjoint subsets of vertices, representing the connected components at each stage.
  - Start with n subsets, each containing one vertex.
  - End with one subset containing all vertices.
- Disjoint Set ADT has 3 operations:
  - **makeset(i)**: creates a singleton set containing vertex i.
  - **findset(i)**: returns the "canonical" member of its subset.
    - I.e., if i and j are elements of the same subset, **findset(i) == findset(j)**
  - **union(i, j)**: merges the subsets containing i and j into a single subset.

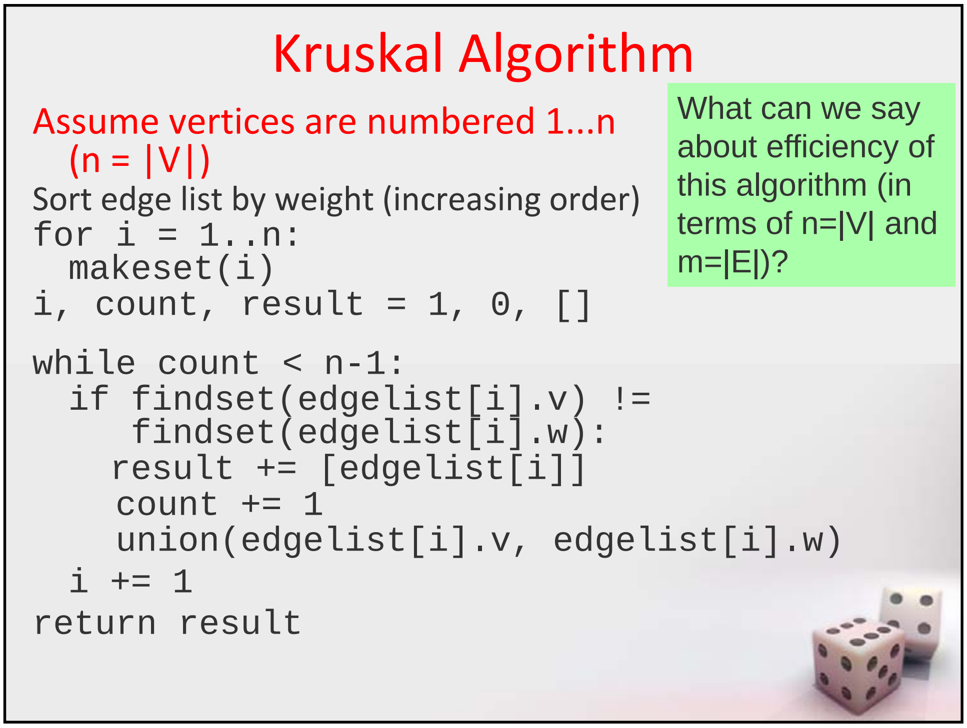


Example of operations

- makeset (1)
- makeset (2)
- makeset (3)
- makeset (4)
- makeset (5)
- makeset (6)

- union(4, 6)
- union (1,3)
- union(4, 5)
- findset(2)
- findset(5)

What are the sets after these operations?

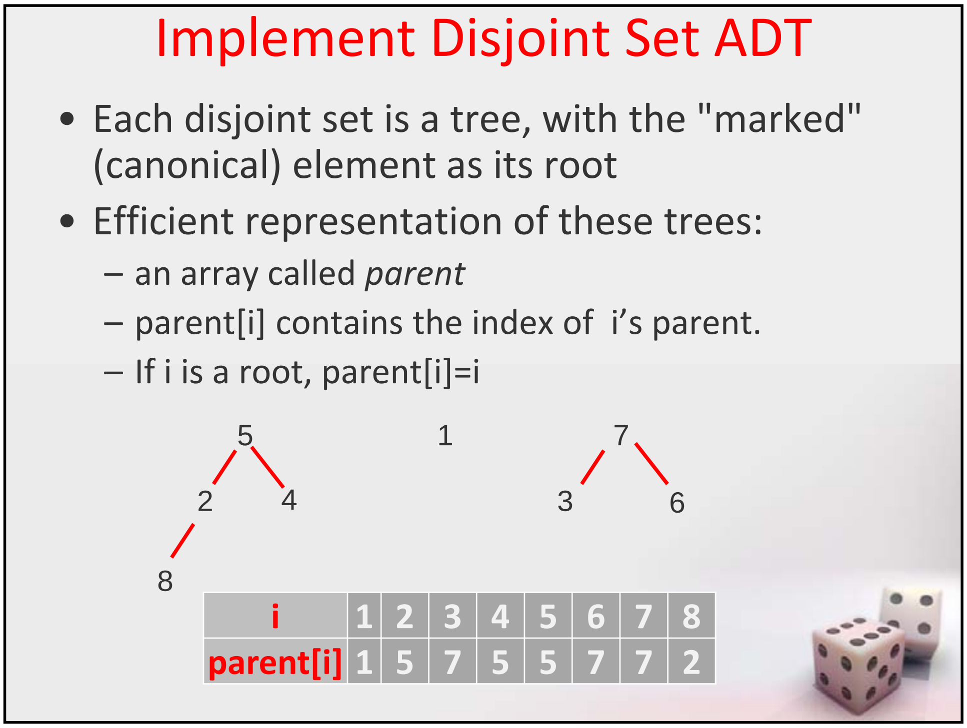


Kruskal Algorithm

What can we say about efficiency of this algorithm (in terms of n=|V| and m=|E|)?

Assume vertices are numbered 1…n (n = |V|)

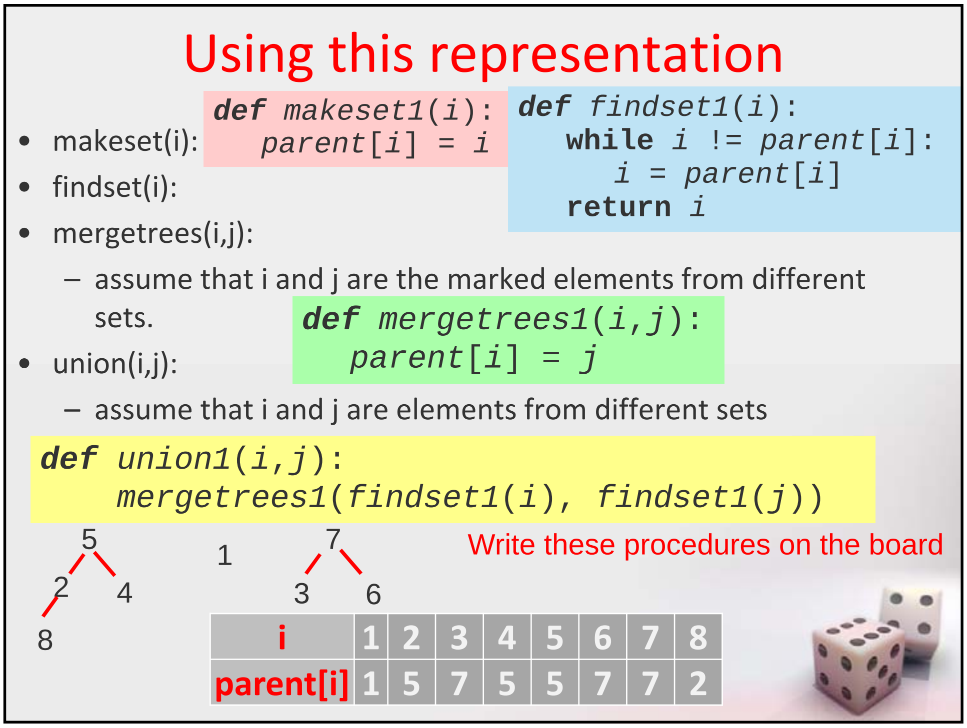
```
Sort edge list by weight (increasing order)
for i = 1..n:
  makeset(i)
i, count, result = 1, 0, []

while count < n-1:
  if findset(edgelist[i].v) !=
     findset(edgelist[i].w):
    result += [edgelist[i]]
    count += 1
    union(edgelist[i].v, edgelist[i].w)
  i += 1
return result
```



Implement Disjoint Set ADT

- Each disjoint set is a tree, with the "marked" (canonical) element as its root
- Efficient representation of these trees:
  - an array called *parent*
  - parent[i] contains the index of i's parent.
  - If i is a root, parent[i]=i

i	1	2	3	4	5	6	7	8
parent[i]	1	5	7	5	5	7	7	2



Using this representation

- makeset(i):

```
def makeset1(i):
    parent[i] = i
```

- findset(i):

```
def findset1(i):
    while i != parent[i]:
        i = parent[i]
    return i
```

- mergetrees(i,j):
  - assume that i and j are the marked elements from different sets.

```
def mergetrees1(i,j):
    parent[i] = j
```

- union(i,j):
  - assume that i and j are elements from different sets

```
def union1(i,j):
    mergetrees1(findset1(i), findset1(j))
```

Write these procedures on the board

i	1	2	3	4	5	6	7	8
parent[i]	1	5	7	5	5	7	7	2

Analysis

- Assume that we are going to do n makeset operations followed by m union/find operations
- time for makeset?
- worst case time for findset?
- worst case time for union?
- Worst case for all m union/find operations?
- worst case for total?
- What if m < n?
- Write the formula to use min

Can we keep the trees from growing so fast?

- Make the shorter tree the child of the taller one
- What do we need to add to the representation?
- rewrite makeset, mergetrees.

```
def makeset2(i):
   parent[i] = i
   height[i] = 0
```

```
def mergetrees2(i,j):
   if height[i] < height[j]):
          parent[i] = j
   elif height[i] > height[j]:
      parent[j] = i
   else:
      parent[i] = j
      height[j] = height[j] + 1
```

- `findset` & `union` are unchanged.
- What can we say about the maximum height of a k-node tree?

Theorem: max height of a k-node tree T produced by these algorithms is $\lfloor \lg k \rfloor$

- Base case…
- Induction step:
  - Let T be a k-node tree ($k > 1$)
  - Induction hypothesis…
  - T is the union of two trees:

    T_1 with k_1 nodes and height h_1

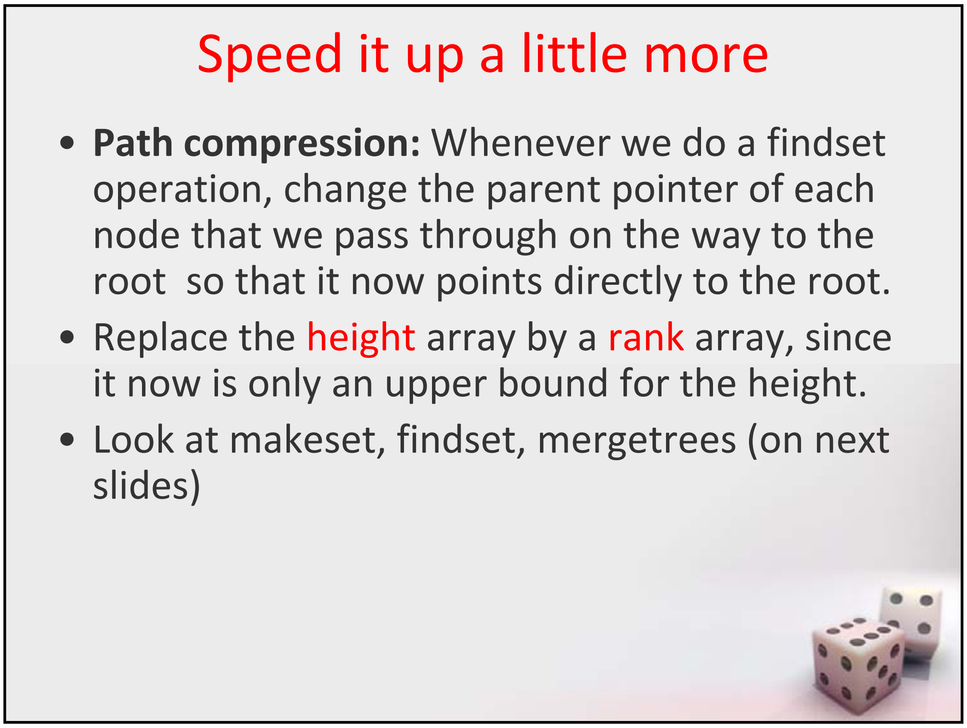
    T_2 with k_2 nodes and height h_2
  - What can we say about the heights of these two trees?
  - Case 1: $h_1 \neq h_2$. Height of T is
  - Case 2: $h_1 = h_2$. WLOG Assume $k_1 \geq k_2$. Then $k_2 \leq k/2$.

    Height of tree is

    $$1 + h2 <= 1 + \lfloor \lg k_2 \rfloor <= 1 + \lfloor \lg k/2 \rfloor$$
    $$= 1 + \lfloor \lg k - 1 \rfloor = \lfloor \lg k \rfloor$$

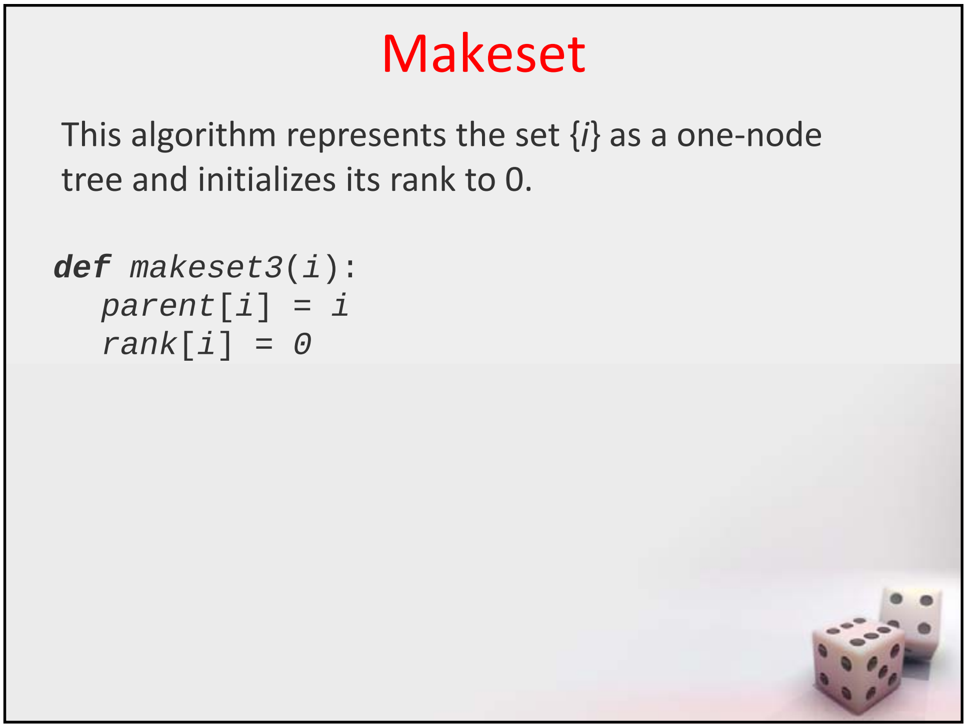
Worst-case running time

- Again, assume n makeset operations, followed by m union/find operations.
- If $m > n$
- If $m < n$



Speed it up a little more

- **Path compression:** Whenever we do a findset operation, change the parent pointer of each node that we pass through on the way to the root so that it now points directly to the root.
- Replace the height array by a rank array, since it now is only an upper bound for the height.
- Look at makeset, findset, mergetrees (on next slides)



Makeset

This algorithm represents the set {*i*} as a one-node tree and initializes its rank to 0.

```
def makeset3(i):
   parent[i] = i
   rank[i] = 0
```

Findset

- This algorithm returns the root of the tree to which i belongs and makes every node on the path from i to the root (except the root itself) a child of the root.

```
def findset(i):
   root = i
   while root != parent[root]:
      root = parent[root]
   j = parent[i]
   while j != root:
      parent[i] = root
      i = j
      j = parent[i]
   return root
```

Mergetrees

This algorithm receives as input the roots of two distinct trees and combines them by making the root of the tree of smaller rank a child of the other root. If the trees have the same rank, we arbitrarily make the root of the first tree a child of the other root.

```
def mergetrees(i,j) :
   if rank[i] < rank[j]:
      parent[i] = j
   elif rank[i] > rank[j]:
      parent[j] = i
   else:
      parent[i] = j
      rank[j] = rank[j] + 1
```

Analysis

- It's complicated!
- R.E. Tarjan proved (1975)*:
  - Let $t = m + n$
  - Worst case running time is $\Theta(t\,\alpha(t, n))$, where α is a function with an *extremely* slow growth rate.
  - Tarjan's α:
  - $\alpha(t, n) \leq 4$ for all $n \leq 10^{19728}$
- Thus the amortized time for each operation is essentially constant time.

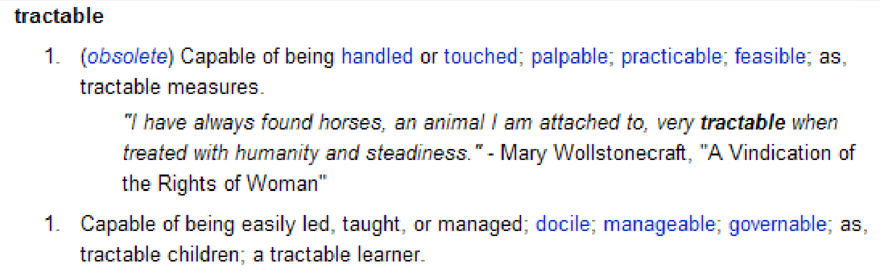
* According to *Algorithms* by R. Johnsonbaugh and M. Schaefer, 2004, Prentice-Hall, pages 160-161

Polynomial-time algorithms

INTRO TO COMPUTATIONAL COMPLEXITY

The Law of the Algorithm Jungle

- Polynomial good, exponential bad!
- The latter is obvious, the former may need some explanation
- We say that polynomial-time problems are **tractable,** exponential problems are **intractable**



tractable

1. (*obsolete*) Capable of being handled or touched; palpable; practicable; feasible; as, tractable measures.

   "I have always found horses, an animal I am attached to, very ***tractable*** *when treated with humanity and steadiness."* - Mary Wollstonecraft, "A Vindication of the Rights of Woman"

1. Capable of being easily led, taught, or managed; docile; manageable; governable; as, tractable children; a tractable learner.

Polynomial time *vs* exponential time

- What's so good about polynomial time?
  - It's not exponential!
    - We can't say that every polynomial time algorithm has an acceptable running time,
    - but it is certain that if it ***doesn't*** run in polynomial time, it only works for small inputs.
  - Polynomial time is closed under standard operations.
    - If f(t) and g(t) are polynomials, so is f(g(t)).
    - also closed under sum, difference, product
- Almost all of the algorithms we have studied run in polynomial time.
  - Except those (like permutation and subset generation) whose output is exponential.

Decision problems

- When we define the class P, of "polynomial-time problems", we will restrict ourselves to *decision problems.*
- *Almost any problem can be rephrased as a decision problem.*
- Basically, a decision problem is a question that has two possible answers, yes and no.
- The question is about some input.
- A *problem instance* is a combination of the problem and a specific input.

Decision problem definition

- The statement of a decision problem has two parts:
  - The ***instance description*** part defines the information expected in the input
  - The ***question* part** states the actual yes-or-no question; the question refers to variables that are defined in the instance description

Decision problem examples

- **Definition:** In a graph G=(V,E), a ***clique*** E is a subset of V such that for all u and v in E, the edge (u,v) is in E.
- **Clique Decision problem**
  - Instance: an undirected graph G=(V,E) and an integer k.
  - Question: Does G contain a clique of k vertices?
- **k-Clique Decision problem**
  - Instance: an undirected graph G=(V,E). **Note that k is some constant, independent of the problem.**
  - Question: Does G contain a clique of k vertices?

Decision problem example

- **Definition:** The ***chromatic number*** of a graph G=(V,E) is the smallest number of colors needed to color G. so that no two adjacent vertices have the same color
- **Graph Coloring Optimization Problem**
  - Instance: an undirected graph G=(V,E).
  - Problem: Find G's chromatic number and a coloring that realizes it
- **Graph Coloring Decision Problem**
  - Instance: an undirected graph G=(V,E) and an integer k>0.
  - Question: Is there a coloring of G that uses no more than k colors?
- Almost every optimization problem can be expressed in decision problem form

Decision problem example

- **Definition:** Suppose we have an unlimited number of bins, each with capacity 1.0, and n objects with sizes $s_1, \ldots, s_n$, where $0 < s_i \leq 1$ (all s_i rational)
- Bin Packing Optimization Problem
  - Instance: $s_1, \ldots, s_n$ as described above.
  - Problem: Find the smallest number of bins into which the n objects can be packed
- Bin Packing Decision Problem
  - Instance: $s_1, \ldots, s_n$ as described above, and an integer k.
  - Question: Can the n objects be packed into k bins?

Reduction

- Suppose we want to solve problem **p**, and there is another problem **q**.
- Suppose that we also have a function T that
  - takes an input x for **p**, and
  - produces T(x), an input for **q** such that the correct answer for **p** with input x is *yes* if and only if the correct answer for **q** with input T(X) is *yes*.
- We then say that **p** is *reducible* to **q** and we write **p≤q**.
- If there is an algorithm for **q**, then we can compose T with that algorithm to get an algorithm for **p.**
- If T is a function with polynomially bounded running time, we say that **p** is *polynomially reducible* to **q** and we write $\mathbf{p} \leq_P \mathbf{q}$.
- From now on, *reducible* means *polynomially reducible.*

Classic 473 reduction

- Moldy Chocolate is reducible to 4-pile Nim

Definition of the class *P*

- **Definition:** An algorithm is ***polynomially bounded*** if its worst-case complexity is big-O of a polynomial function of the input size n.
  - i.e. if there is a single polynomial p such that for each input of size n, the algorithm terminates after at most p(n) steps.
- **Definition:** A problem is polynomially bounded if there is a polynomially bounded algorithm that solves it
- **The class P**
  - P is the class of decision problems that are polynomially bounded
  - Informally (with slight abuse of notation), we also say that polynomially bounded optimization problems are in P

Example of a problem in *P*

- Shortest Path
  - Input: A weighted graph G=(V,E) with n vertices (each edge e is labeled with a non-negative weight w(e)), two vertices v and w and a number k.
  - Question: Is there a path in G from v to w whose total weight is ≤ k?
- How do we know it's in *P*?

Example: Clique problems

- It is known that we can determine whether a graph with n vertices has a k-clique in time $O(k^2n^k)$.
- **Clique Decision problem 1**
  - Instance: an undirected graph G=(V,E) and an integer k.
  - Question: Does G contain a clique of k vertices?
- **Clique Decision problem 2**
  - Instance: an undirected graph G=(V,E). **Note that k is some constant, independent of the problem.**
  - Question: Does G contain a clique of k vertices?
- Are either of these decision problems in *P*?

The problem class *NP*

- *NP* stands for Nondeterministic Polynomial time.
- The first stage assumes a “guess” of a possible solution.
- Can we **verify** whether the proposed solution really is a solution in polynomial time?

More details

- Example: Graph coloring. Given a graph G with N vertices, can it be colored with k colors?
- A solution is an actual k-coloring.
- A “proposed solution” is simply something that is in the right form for a solution.
  - For example, a coloring that may or may not have only k colors, and may or may not have distinct colors for adjacent nodes.
- The problem is in *NP* iff there is a polynomial-time (in N) algorithm that can check a proposed solution to see if it really is a solution.

Still more details

- A nondeterministic algorithm has two phases and an output step.
- The nondeterministic "guessing" phase, in which the proposed solution is produced. It will be a solution if there is one.
- The deterministic verifying phase, in which the proposed solution is checked to see if it is indeed a solution.
- Output "yes" or "no".