# Welcome to
# Applied Algorithms

IDC HERZLIYA
Efi Arazi School of Computer Science

Lecturer: Prof. Tami Tamir
(tami@idc.ac.il)

Peking University

Summer 2019

# Course Goals

- Algorithms course:
  - A set of basic algorithms for classical problems.
  - Mostly graph algorithms.
  - An initial toolbox of techniques.
- Here: We will focus on real-world applications.
  - A better sense of how to model problems you encounter as algorithmic problems.
  - Trade-offs and conflicting objectives in resource utilization.
  - System's costs v.s clients' quality of service.
  - Challenges in algorithm design for selfish users.
- Both: Fun! Beauty!

# About Me

- Tami Tamir
  - I'm from Israel
  - Prof. of computer science at the Interdisciplinary center, Herzliya.
  - Graduated with a Ph.D from the Technion in 2000.
  - Worked several years for Intel.
  - Spent several years as a post-doc and lecturer in univ. of Washington, Seattle, U.S.A.
  - My research interests are exactly the topics of this course.

# Course Topics

- Scheduling Theory
- Facility Location
- Packing Problems
- Algorithmic Game Theory


Along the way:

- Graph Algorithms
- Complexity classes
- Coping with NP-completeness
- Approximation Algorithms
- Heuristics
- Design for a centralized system vs. selfish agents
- Resource allocation principles

# Applied Algorithm Scenario

Real world problem

↓

Abstractly model the problem

↓

Find abstract algorithm

↓

Adapt to original problem

# Modeling

- What kind of algorithm is needed?
- Can I find an algorithm or do I have to invent one?
- Can I 'tune' an existing algorithm? Does it remind me a familiar problem?

# Algorithm Design Goals

- Correctness
- Efficiency
- Simple, if possible.

# Evaluating an algorithm

Mike: My algorithm can sort $10^6$ numbers in 3 seconds.

Bill: My algorithm can sort $10^6$ numbers in 5 seconds.

Mike: I've just tested it on my new Intel core duo.

Bill: I remember my result from my undergraduate studies (2005).

Mike: My input is a random permutation of $1,\dots,10^6$.

Bill: My input is the sorted output, so I only need to verify that it is sorted.

# Types of complexity

- We should analyze separately 'good' and 'bad' inputs.
- Processing time is surely a bad measure!!!
- We need a 'stable' measure, independent of the implementation.

* A complexity function is a function $T: N \rightarrow N$.

  $T(n)$ is the number of operations the algorithm does on an input of size $n$.

* We can measure three different things.

- Worst-case complexity
- Best-case complexity
- Average-case complexity

# The RAM Model of Computation

- Each simple operation takes 1 time step.
- Loops and subroutines are not simple operations.
- Each memory access takes one time step, and there is no shortage of memory.

For a given problem instance:

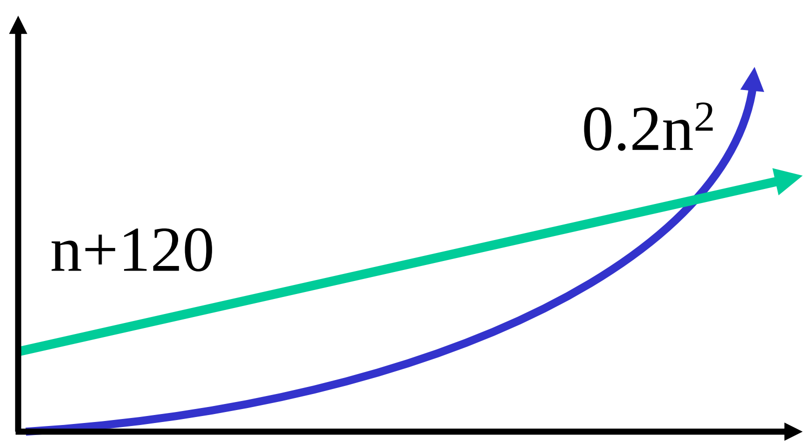
- Running time of an algorithm = # RAM steps.
- Space used by an algorithm = # RAM memory cells

useful abstraction $\Rightarrow$ allows us to analyze algorithms in a machine independent fashion.

# Big O Notation

- Goal :
  - A stable measurement independent of the machine.
- Way:
  - ignore constant factors.
- $f(n) = O(g(n))$ if $c \cdot g(n)$ is upper bound on $f(n)$

$\Leftrightarrow$ There exist $c$, $N$, s.t. for any $n \geq N$, $f(n) \leq c \cdot g(n)$




For all $n \geq 30$, $n+120 \leq 0.2n^2$

For all $n \geq 60$, $n+120 \leq 2n$

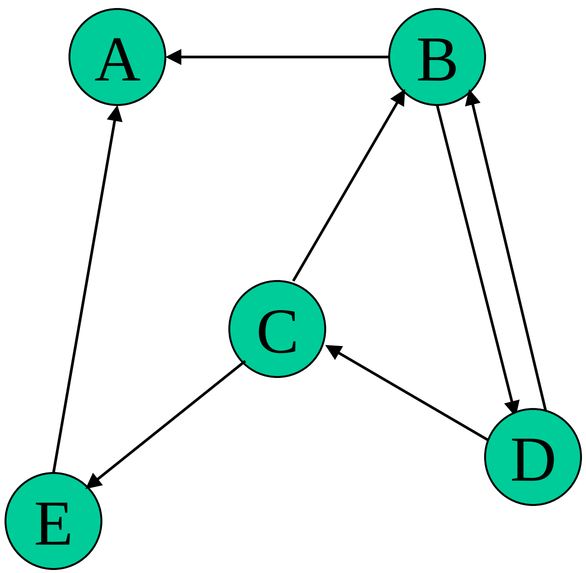
$\Rightarrow n+120 = O(n)$.

while $0.2n^2 = O(n^2)$.

# Growth Rates

- Even by ignoring constant factors, we can get an excellent idea of whether a given algorithm will be able to run in a reasonable amount of time on a problem of a given size.
- The "big O" notation and worst-case analysis are tools that greatly simplify our ability to compare the efficiency of algorithms.

# Review : Graphs

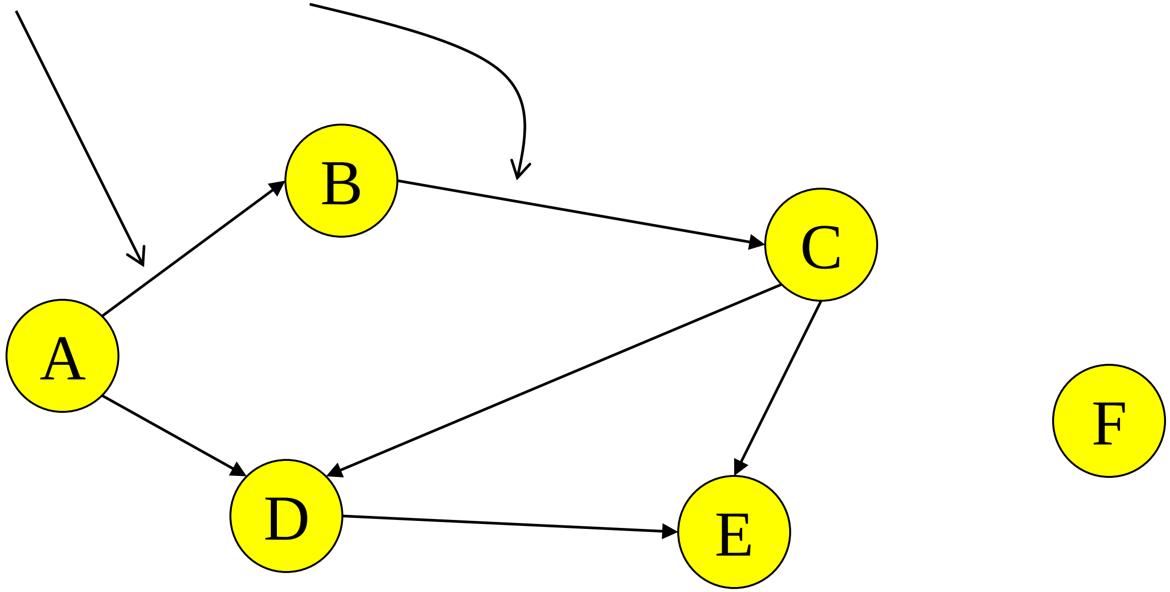



- $G = (V,E)$
- V : A set of nodes (vertices)
- E : A set of edges.
- $|V|=n$, $|E|=m$
- Directed/undirected
- Weighted/unweighted
- In/out-degree

# Graph Example

- The figure presents a directed graph $G = (V, E)$
  - Each edge is a pair $(v_1, v_2)$, where $v_1, v_2$ are vertices in $V$.
  - $V$ = {A, B, C, D, E, F}

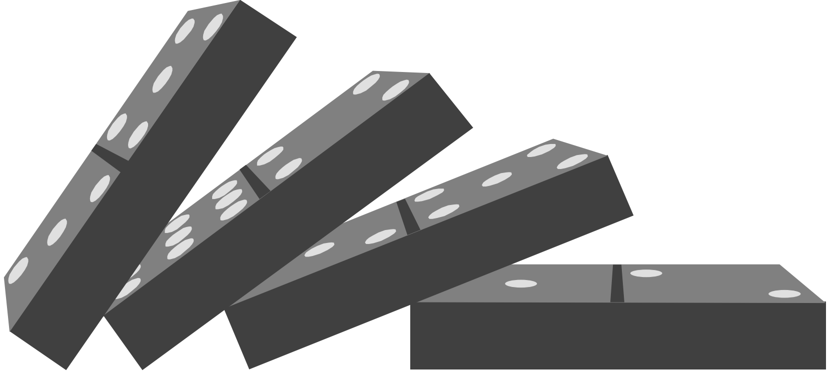
  $E$ = {(A,B), (A,D), (B,C), (C,D), (C,E), (D,E)}



# Examples (in class)

- *A path*: $P_n$
- *A cycle*: $C_n$
- *A complete graph* (clique): $K_n$
- *A star*: $S_n$

- *A bipartite* graph: $G=(V_1,V_2,E)$. For every edge $e=(u,v)$, $u \in V_1$, $v \in V_2$.
- *A complete bipartite*, $K_{a,b}$ : A bipartite in which $E= V_1 \times V_2$. $|E|=ab$.
- Note: A star $S_n$ is a complete bipartite $K_{1,n}$
- A path is a bipartite.
- What about a cycle?



# Induction has many appearances.

- Formal Arguments
- Loop Invariants
- Recursion
- Algorithm Design
- Dynamic Programming

# Review: Induction

- Let $P: N \rightarrow \{true, false\}$, be a predicate on $N$.
- Suppose
  1. $P(k)$ is true for a fixed constant $k$

     Often $k = 0$.
  2. $P(n) \rightarrow P(n+1)$ for all $n \geq k$
- Then $P(n)$ is true for all $n \geq k$

# Proof By Induction

- Claim: $P(n)$ is true for all $n \geq k$
- Base:
  - Show $P(n)$ is true for $n = k$
- Inductive hypothesis:
  - Assume $P(n)$ is true for an arbitrary $n$
- Step:
  - Show that $P(n)$ is then true for $n+1$

# Induction Example: Sum of Geometric sequence

- Prove by induction on n: for all $a \neq 1$

$$\sum_{i=0}^{n} a^i = \frac{a^{n+1}-1}{a-1}.$$

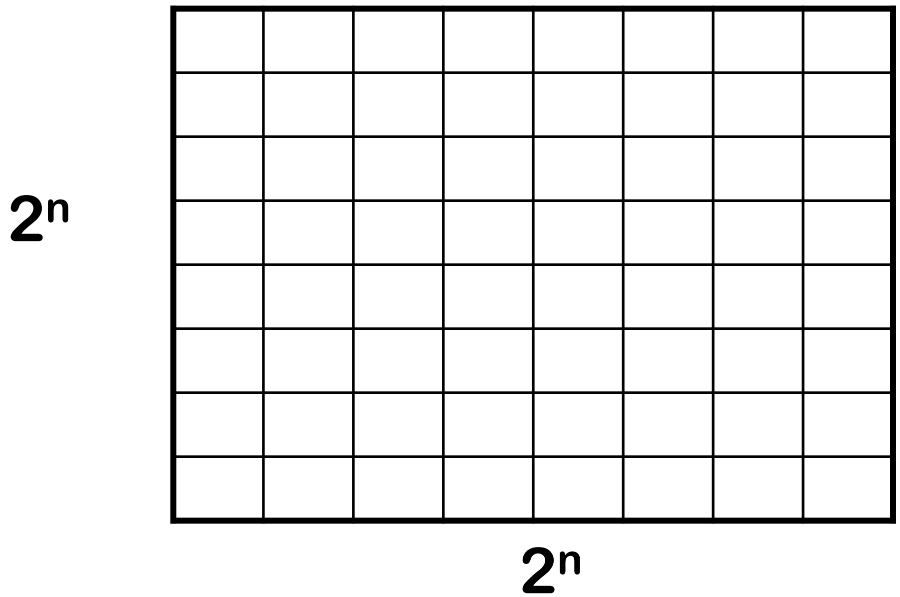
- Base: n=0. $a^0 = \frac{a^{0+1}-1}{a-1}$.

  Indeed, $a^0 = 1 = \frac{a-1}{a-1} = \frac{a^{0+1}-1}{a-1}$.

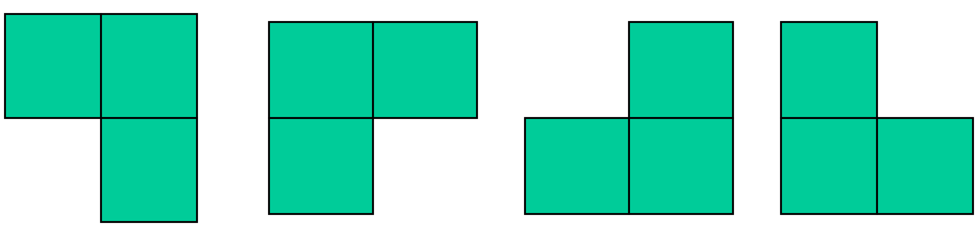
- Inductive hypothesis:
  - Assume $\sum_{i=0}^{n} a^i = \frac{a^{n+1}-1}{a-1}$
- Step (show true for n+1): In Class.

# Design by Induction. Example: Tiling

Goal: tile a room of $2^n$ x $2^n$ squares.



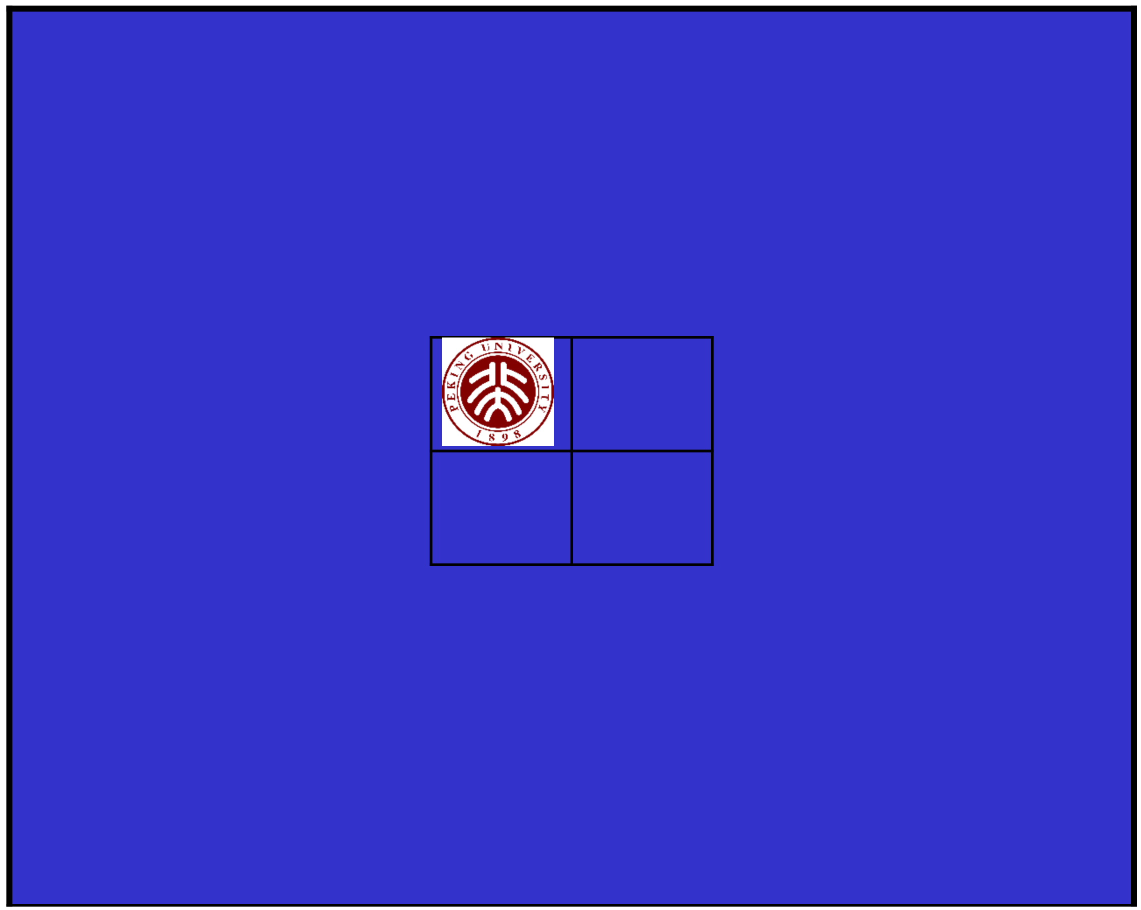

Architect allows only L-shaped tiles covering 3 squares



# Tiling

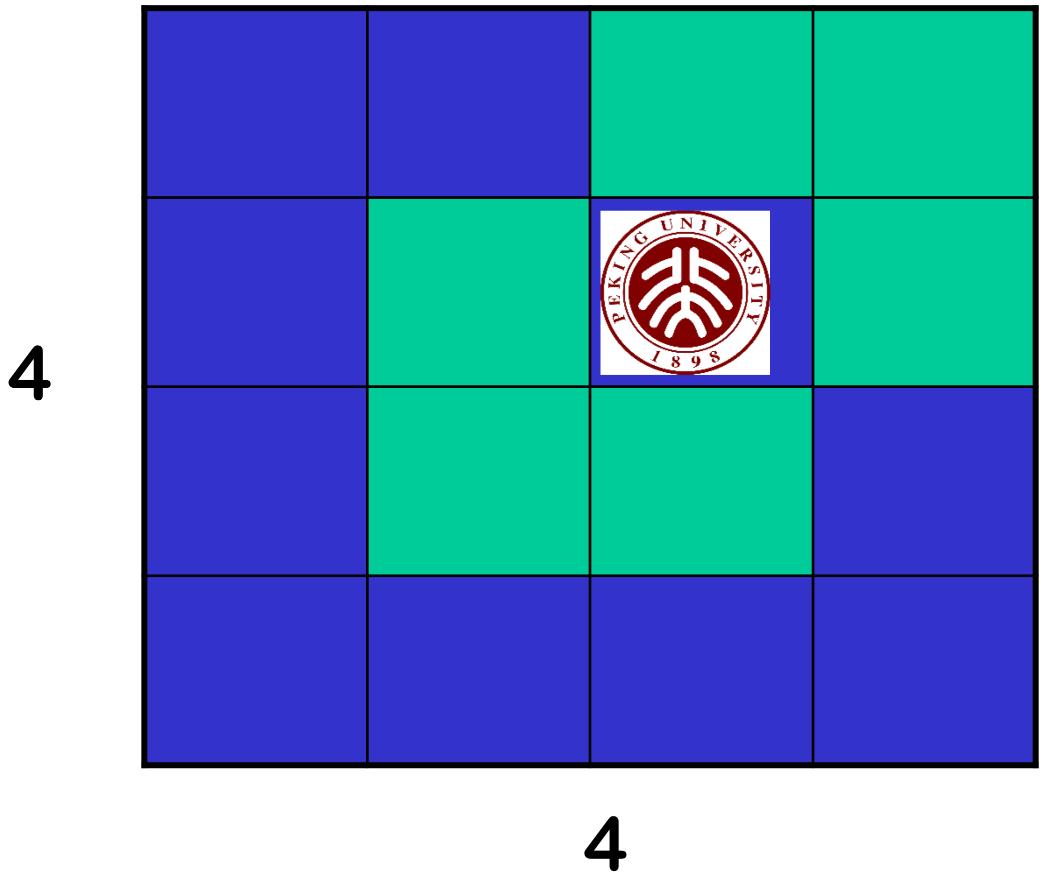
A tile in the middle is reserved to PKU logo.

Middle = one of the four middle squares.



# Tiling

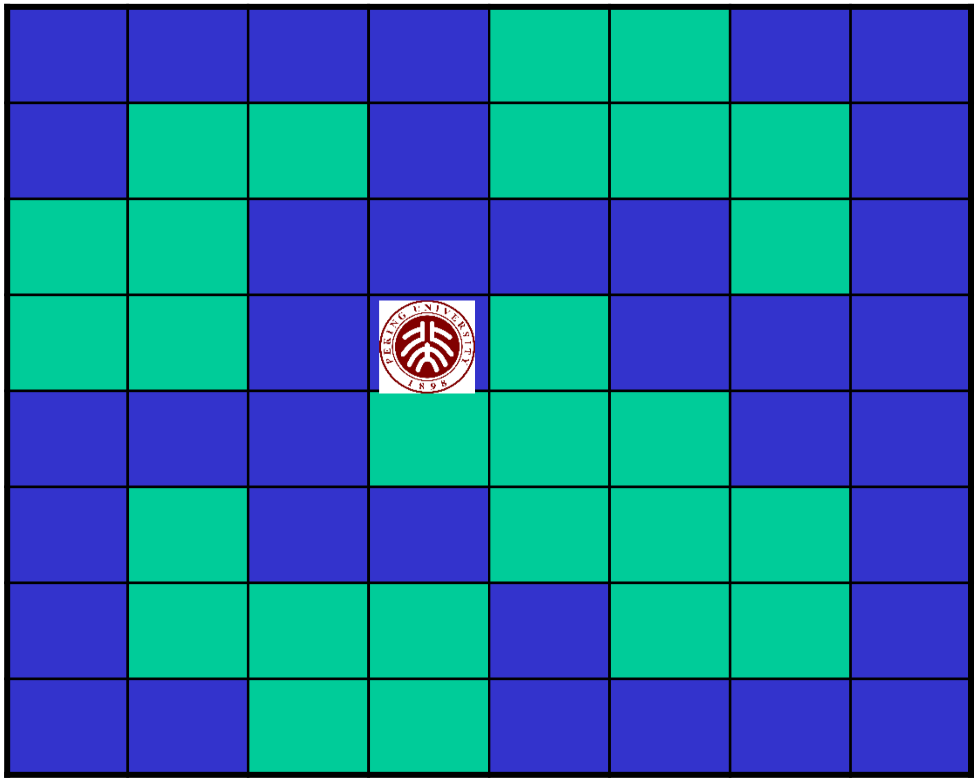
For n=2, the 4x4 square can be tiled as follows:



# Tiling

For n=3, the 8x8 square can be tiled as follows:

8



8

# Tiling

Theorem: For all $n \in N$ we can tile $2^n$ x $2^n$ square according to the conditions.

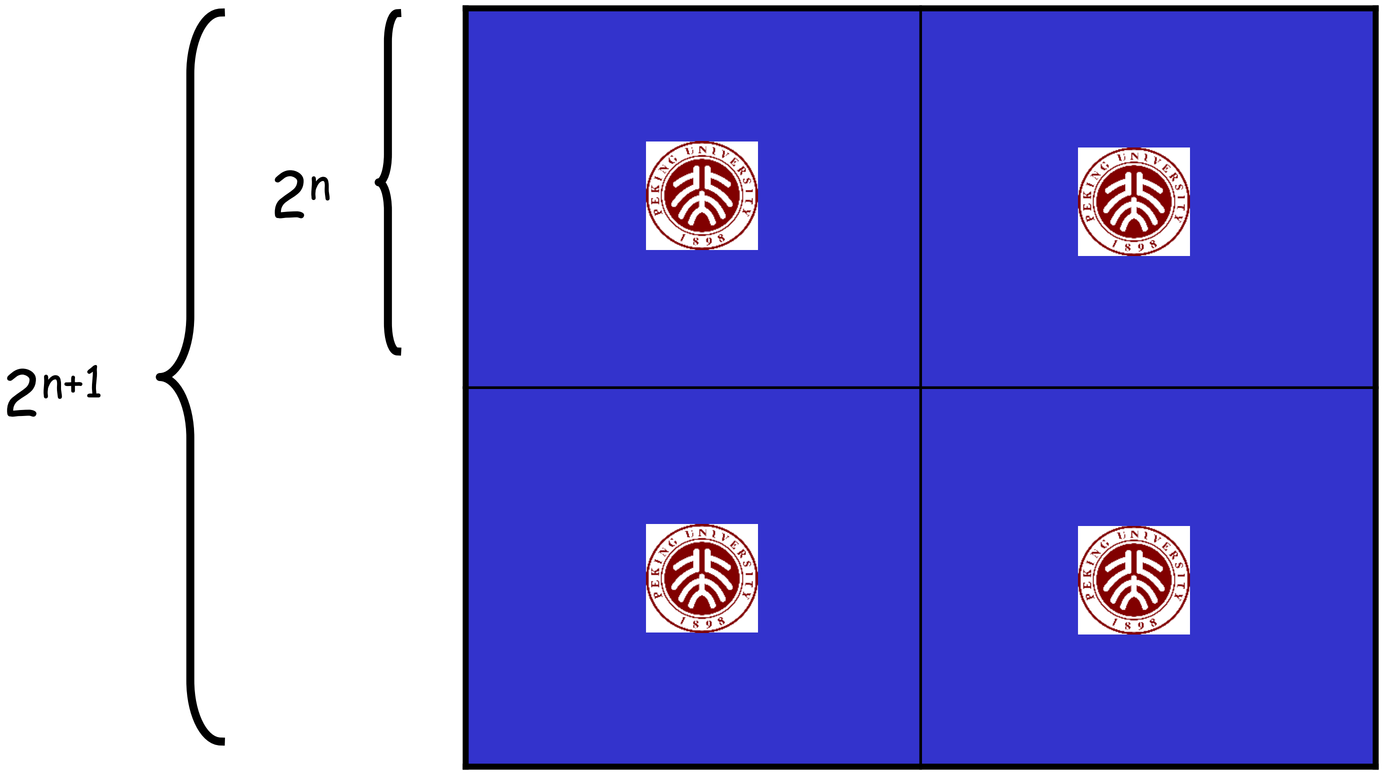
Proof: By induction on n.

Let P(n) := [can tile properly a $2^n$ x $2^n$ square with PKU logo in middle]

Base case: True for n = 0 (no tiles are needed).

# Tiling

<u>Induction step</u>: assume can tile $2^n$ x $2^n$ square, prove that can tile $2^{n+1}$ x $2^{n+1}$ square.




Now what??

# Tiling - stronger hypothesis

The idea: Use a stronger induction hypothesis.

1. Implies the original theorem.
2. Makes proving $P(n) \Rightarrow P(n+1)$ easier!

Proof (second attempt): By induction on n. Let

$P'(n)$:= [can tile properly $2^n \times 2^n$ square with PKU logo in any selected location]

Note: this implies

$P(n)$ := [can tile $2^n \times 2^n$ square with PKU logo in the middle]

# Tiling

P'(n):= [can tile properly $2^n \times 2^n$ square with PKU logo in any selected location]

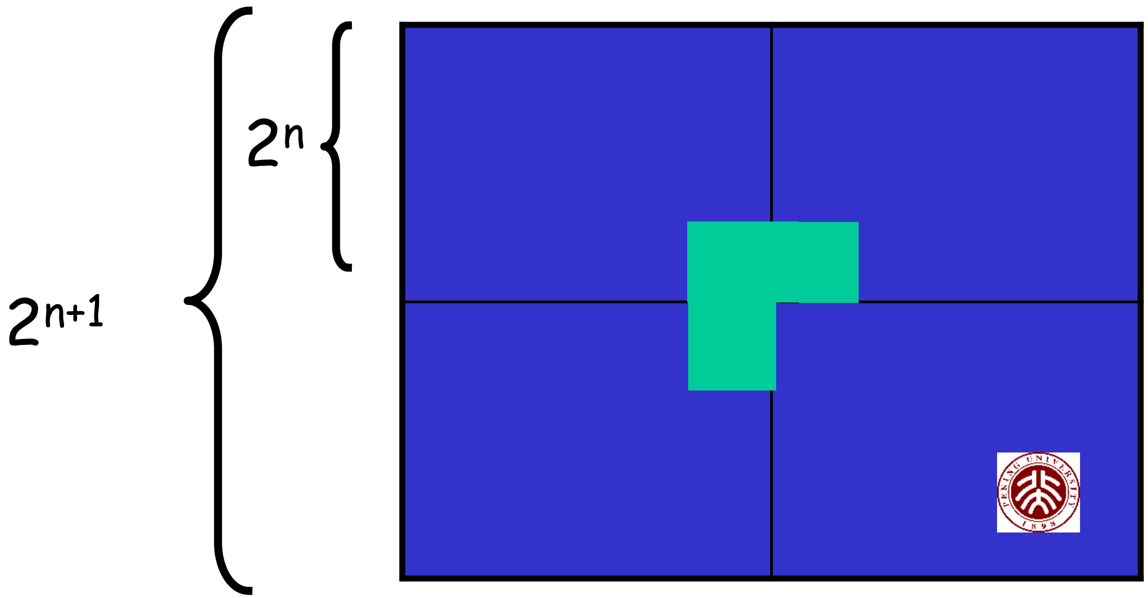
Base case: Still true for n = 0 (no tiles are needed).

Induction step: Assume can tile $2^n \times 2^n$ square with PKU logo in any location.

# Tiling

Given a $2^{n+1} \times 2^{n+1}$ square:

1. Ask the client to select PKU logo's location.
2. Locate the first tile in the middle, such that one block is missing from every quarter.
3. By the induction hypothesis the quarters can be legally tiled.



# What are the Lessons?

Proof by induction can be constructive:

1. Sometimes yields an efficient procedure/algorithm.
2. Our proof implicitly defined a recursive tiling algorithm.

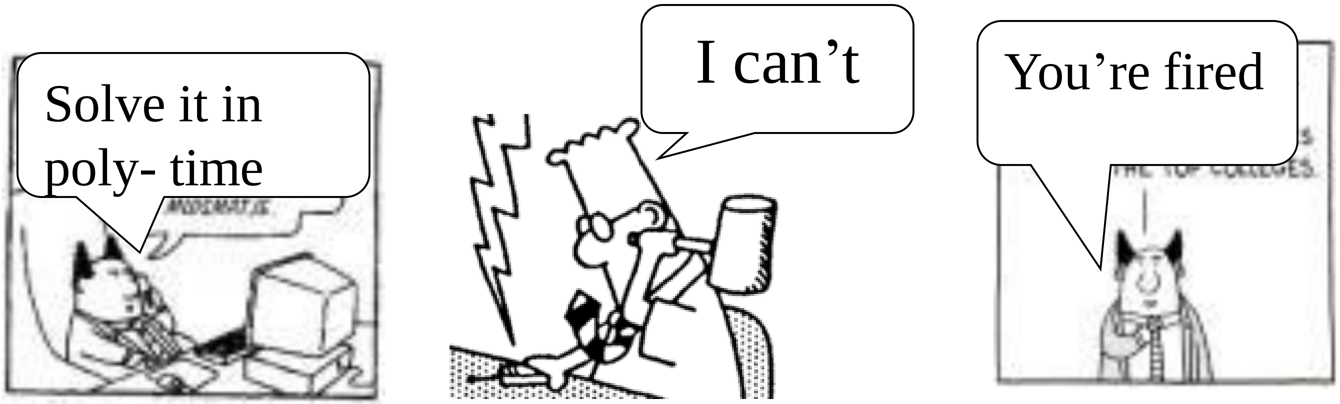
demo

Choice of the induction hypothesis is crucial:

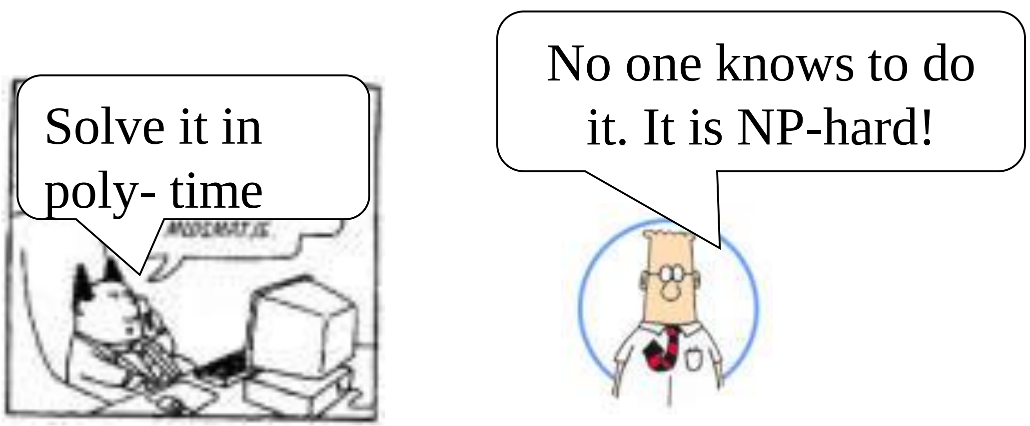
1. Assuming stronger hypothesis may make proof easier!
2. But need to ensure that $P(n) \Rightarrow P(n+1)$ is indeed true.

# NP-Completeness Theory

I.




II.



# NP-Completeness Theory

- Explains why some problems are hard and probably not solvable in polynomial time.
- Invented by Cook in 1971.
- Talks about the problems, independent of the implementation, the machine, or the algorithm.

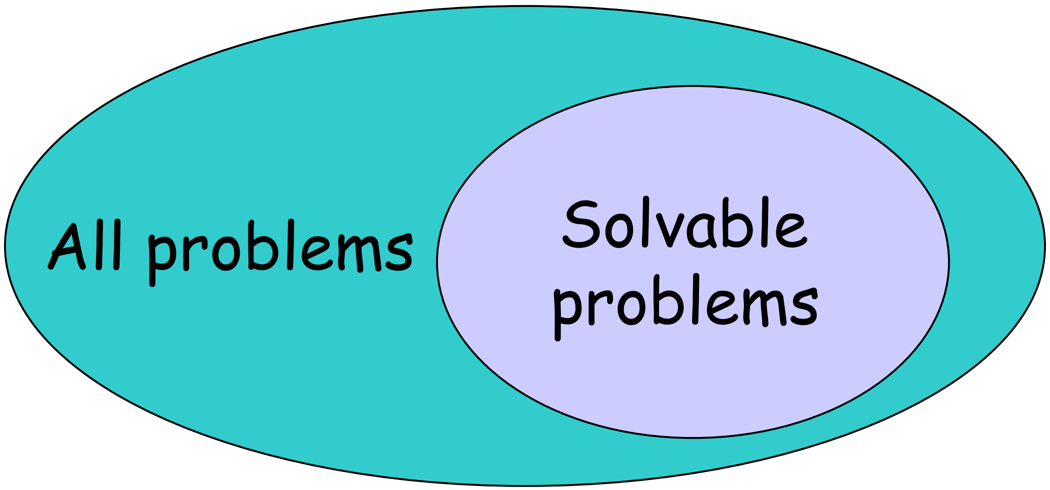
# Polynomial-Time Algorithms

- Some problems are intractable: as they grow large, we are unable to solve them in reasonable time.
- What constitutes reasonable time? Standard working definition: polynomial time
  - On an input of size $n$ the worst-case running time is $O(n^k)$ for some constant $k$
  - Polynomial time: $O(n^2)$, $O(n^3)$, $O(1)$, $O(n \log n)$
  - Not in polynomial time: $O(2^n)$, $O(n^n)$, $O(n!)$

# Polynomial-Time Algorithms

- Are some problems solvable in polynomial time?
  - Of course: most of the algorithms we've studied so far provide polynomial-time solutions to some problems.
  - We define **P** to be the class of problems solvable in polynomial time.
- Are all problems solvable in polynomial time?
  - No: Turing's "Halting Problem" is not solvable by any computer, no matter how much time is given
  - Such problems are clearly intractable, not in **P**

# So some problems cannot be solved at all



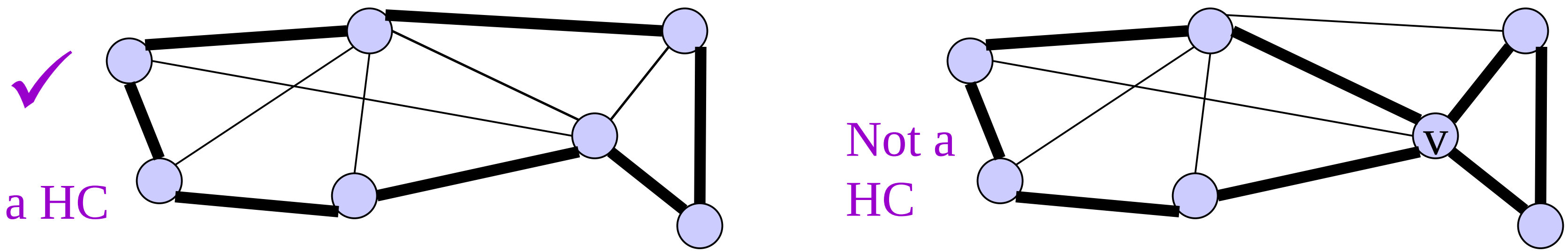

We will explore the 'solvable area', and will distinguish between problems that can be solved efficiently and those that cannot be solved efficiently.

# NP-Complete Problems

- The *NP-Complete* problems are an interesting class of solvable problems whose status is unknown
  - No polynomial-time algorithm has been discovered for an NP-Complete problem.
  - No above-polynomial lower bound has been proved for any NP-Complete problem, either.
- We call this *the P = NP question*
  - The biggest open problem in CS.

# An NP-Complete Problem: Hamiltonian Cycles

- An example of an NP-Complete problem:
  - A *hamiltonian cycle* of an undirected graph is a simple cycle that contains every vertex.




  - The hamiltonian-cycle problem: given a graph G, does it have a hamiltonian cycle?
  - A naive algorithm for solving the hamiltonian-cycle problem: check all paths.
  - Running time? Exponential in size of G.

# P and NP

- **P** = problems that can be solved in polynomial time
- **NP** = problems for which a solution can be verified in polynomial time = problems that can be solved in polynomial time by a non-deterministic machine.
- Unknown whether **P = NP** (most suspect not)

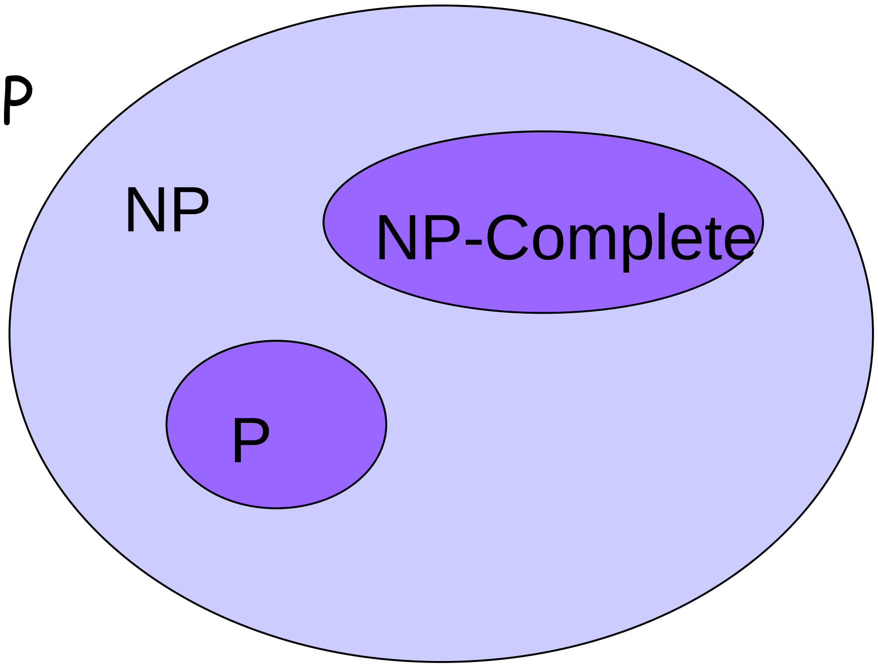
• Hamiltonian-cycle problem is in **NP**:
  - Cannot solve in polynomial time.
  - Easy to verify solution in polynomial time.

# NP-Complete Problems

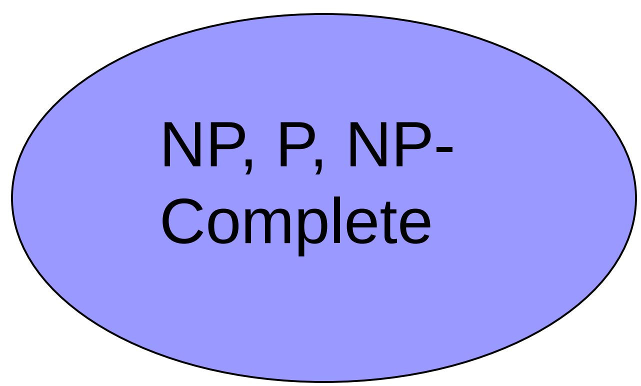
- NP-Complete problems are the "hardest" problems in NP:
  - If any *one* NP-Complete problem can be solved in polynomial time…
  - …then *every* NP-Complete problem can be solved in polynomial time…
  - …and in fact *every* problem in **NP** can be solved in polynomial time (which would show **P** = **NP**)
  - Thus: solve hamiltonian-cycle in $O(n^{100})$ time, you've proved that **P** = **NP**. Retire rich & famous.

# NP Problems

For sure P $\subseteq$NP

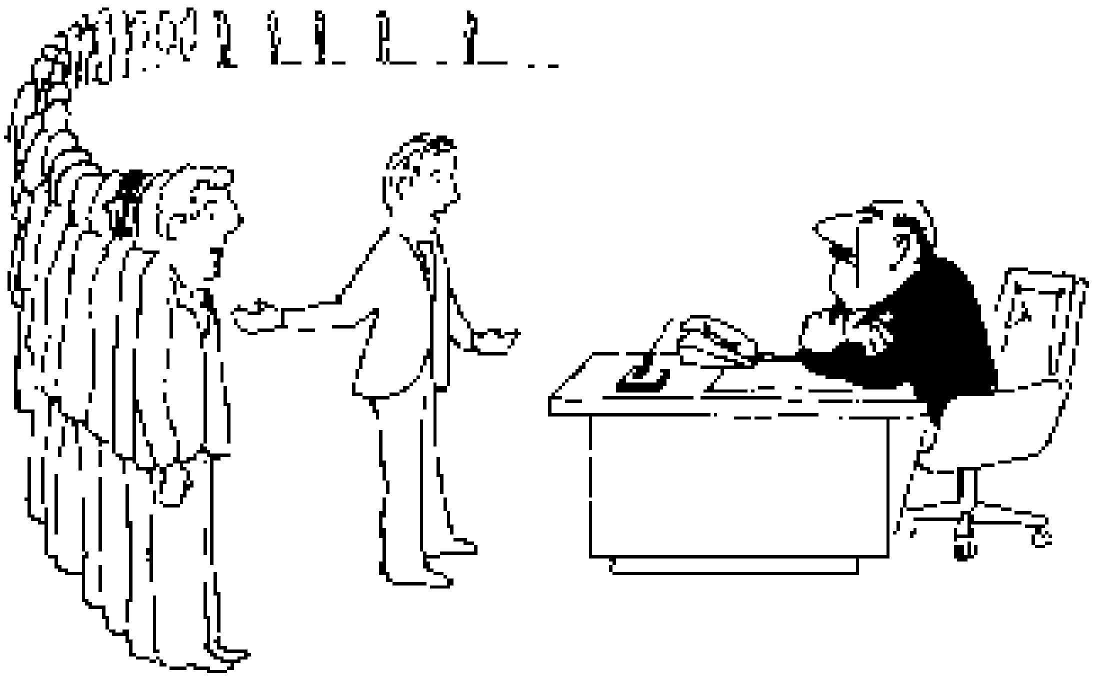



But maybe P=NP ??



# Why Prove NP-completeness?

—



I can't find an efficient algorithm, but neither can all these famous people.

# Why Prove NP-completeness?

- Though nobody has proven that **P** != **NP**, if you prove a problem is NP-Complete, most people accept that it is probably intractable.
- Therefore it can be important to prove that a problem is NP-Complete
  - Don't need to come up with an efficient algorithm.
  - Can instead work on *approximation algorithms*.

# NP-Hard and NP-Complete Problems

- If P is *polynomial-time reducible* to Q, we denote this $P \leq_p Q$ - it means that if we have a black box that solves Q in polynomial time then it is possible to use this black box to solve P in polynomial time.
- Definition of NP-complete:
  - P is NP-complete if $P \in NP$ and P is NP-hard.
- Definition of NP-Hard:
  - P is NP-hard if all problems R of NP are reducible to P. Formally: $R \leq_p P, \forall R \in NP$
- If $P \leq_p Q$ and P is NP-hard, Q is also NP-hard.

# Using Reductions

- Given one NP-Complete problem, we can prove that many interesting problems NP-Complete. This includes:
  - Graph coloring
  - Hamiltonian path/cycle
  - Knapsack problem
  - Traveling salesman
  - Job scheduling
  - Many, many, many more (see the compendium)

# Proving NP-Completeness

- How do we prove a problem P is NP-Complete?
  - Pick a known NP-Complete problem Q
  - Reduce Q to P (show $Q \leq_p P$, use P to solve Q)
    - Describe a transformation that maps instances of Q to instances of P, s.t. "yes" for P = "yes" for Q
    - Prove the transformation works
    - Prove it runs in polynomial time
  - and yeah, prove $P \in$ **NP**
- We need at least one problem for which NP-hardness is known. Once we have one, we can start reducing it to many problem.

# The SAT Problem

- The first problems to be proved NP-Complete was *satisfiability* (SAT):
  - Given a Boolean expression on $n$ variables, can we assign values such that the expression is TRUE?
  - Ex: $((x_1 \wedge x_2) \vee \neg((\neg x_1 \wedge x_3) \vee x_4)) \wedge \neg x_2$
  - **Cook's Theorem:** The satisfiability problem is NP-Complete
    - Note: Argue from first principles, not reduction (any computation can be described using SAT expressions)
    - Proof: not here

# The Traveling Salesman Problem:

- A well-known optimization problem:
  - Optimization variant: a salesman must travel to $n$ cities, visiting each city exactly once and finishing where he begins. How to minimize travel time?
  - Model as complete graph with cost $c(i,j)$ to go from city $i$ to city $j$
- How would we turn this into a decision problem?
  - Answer: ask if there exists a path with cost $< k$

We will prove that TSP is NP-hard by a reduction from Hamiltonian cycle to TSP

# Hamiltonian Cycle $\leq_p$ TSP

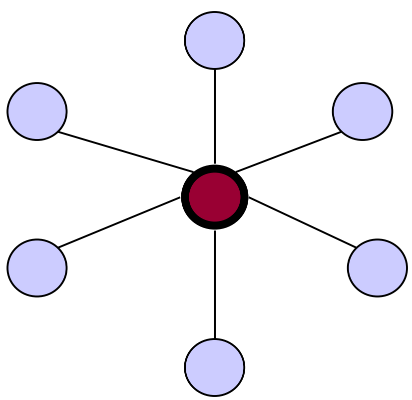
- The hamiltonian-cycle problem: given a graph G, is there a simple cycle that contains every vertex?
- Given a graph G = (V,E) and the question whether it includes a Hamiltonian cycle, we build a graph G' = (V,E'), input for TSP.
- G' is a complete graph
- Edges in E' that are also in E have cost 0
- All other edges in E' have cost 1
- Claim: G' has a TS cycle whose cost is 0 if and only if G has a ham. Cycle.

# Additional NP-Complete Problems

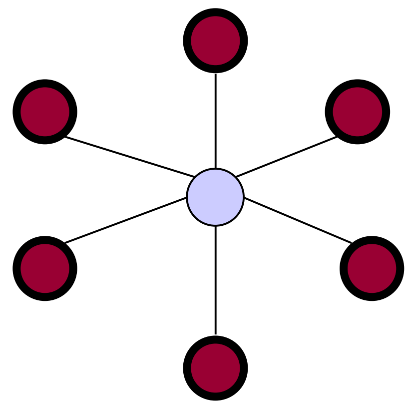
- Partition: Given a set of integers, whose total sum is 2S, can we partition them into two sets, each adds up to S?
- Subset-sum: Given a set of integers, does there exist a subset that adds up to some target *T*?

# Independent Set

- Input: A graph $G=(V,E)$, $k$
- Problem: Is there a subset $S$ of $V$ of size at least $k$ such that no pair of vertices in $S$ has an edge between them.
- Maximum independent set problem: find a maximum size independent set of vertices.



Maxima**l** independent set



Maxim**um** independent set

# Coping with NP-hardness

- O.K, I know that a problem is NP-hard. What should I do next?
- First, stop looking for an efficient algorithm.
- Next, you might insist on finding an optimal solution (knowing that this may takes a lot of time), or you can look for approximate solutions with guaranteed performance.

# Techniques for Dealing with NP-complete Problems

- Exactly
  - backtracking, branch and bound, dynamic programming.
- Approximately
  - approximation algorithms with performance guarantees.
  - heuristics with good average results.
- Change/relax the problem (if possible…), or consider only specific classes of instances.

# Approximation Algorithms

- The fact that a problem is NP-complete doesn't mean that we cannot find an approximate solution efficiently.
- We would like to have some guarantee on the performance – how far are we from the optimal?
- Many real-life resource-allocation problems are NP-hard. We will study several approximation algorithms in this course.