Welcome to
Applied Algorithms

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

Peking University Summer 2017

Course Goals

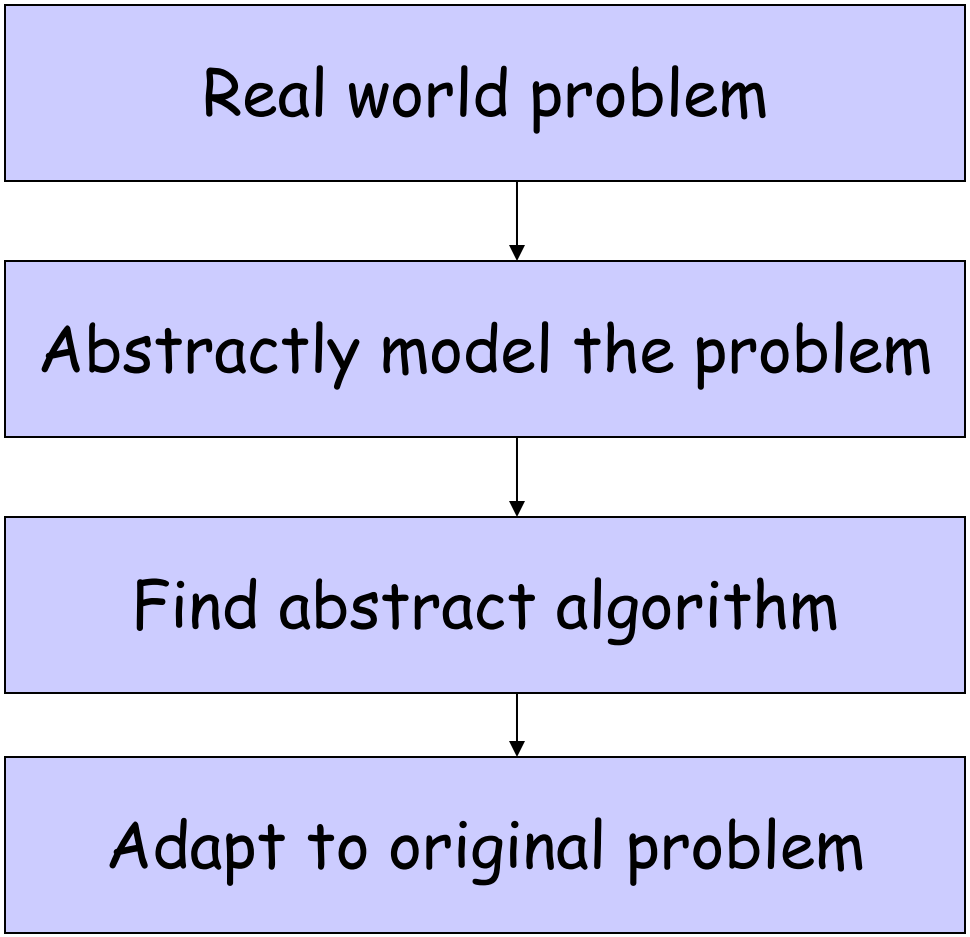
- An appreciation for applications of algorithmic techniques in the real world.
- A large toolbox.
- A better sense of how to model problems you encounter as well-known algorithmic problems.
- A deeper understanding of the issues and tradeoffs involved in algorithm design.
- Fun! Beauty!

Course Topics

- Review: Graphs, Complexity classes
- Stable Matching
- Scheduling Theory
- Facility Location
- Packing Problems
- Algorithmic Game Theory

- Along the way:
  - Coping with NP-completeness
  - Approximation Algorithms
  - Heuristics
  - Design for a centralized system vs. selfish agents
  - Resource allocation principles

Applied Algorithm Scenario



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 (1995).

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

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:

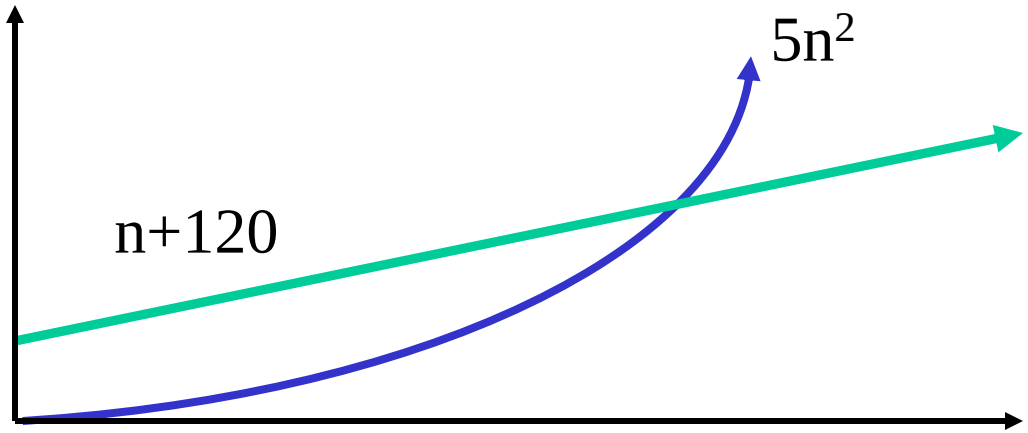
- Running time of an algorithm = # RAM steps.
- Space used by an algorithm = # RAM memory cells

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 $n \geq 5$
$n+120 \leq 5n^2$

$\Rightarrow n+120 = O(n^2)$.

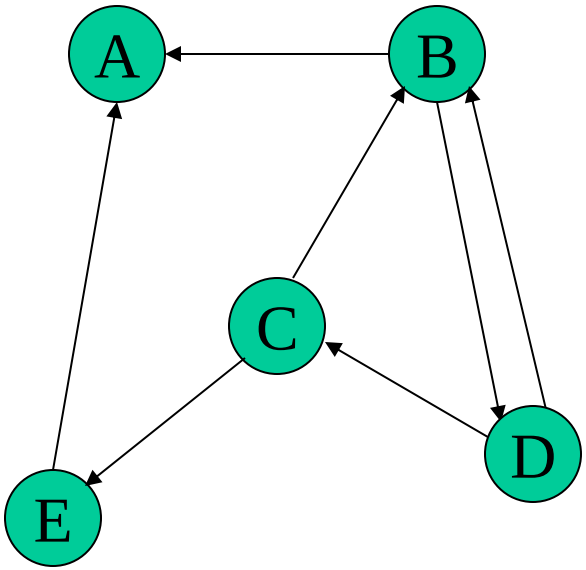
Also, for all $n \geq 60$
$n+120 \leq 2n$

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

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

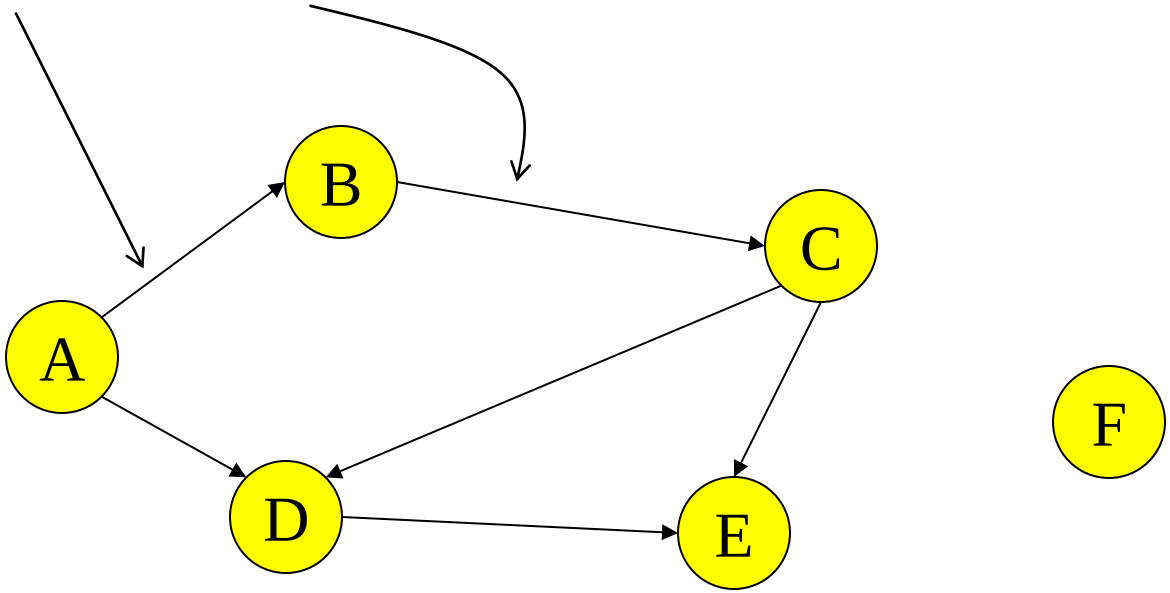


- G = (V,E)
- V : A set of nodes (vertices)
- E : A set of edges.
- |V|=n, |E|=m
- Directed/undirected
- Weighted/unweighted
- In/out-degree

Graph Example

- Here is 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?

Graph Traversals

Input: Graph G, vertex s

Output: All vertices connected from s.

Two main approaches:

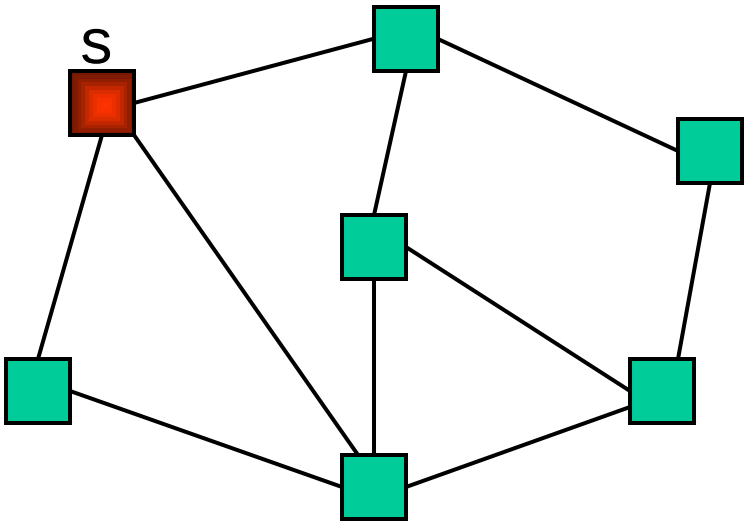
1. Breadth First Search
2. Depth First Search

Graph Traversals

```
F = {s}
Repeat
   u ← node from F
   F = F- {u}
   for all v neighbor of u
      if v not marked
         mark v
         (*) F = F + {v}
Until F is empty
```

BFS: use a queue to keep F

DFS: use a stack to keep F
(a recursive call at (*))



Shortest-path Algorithms

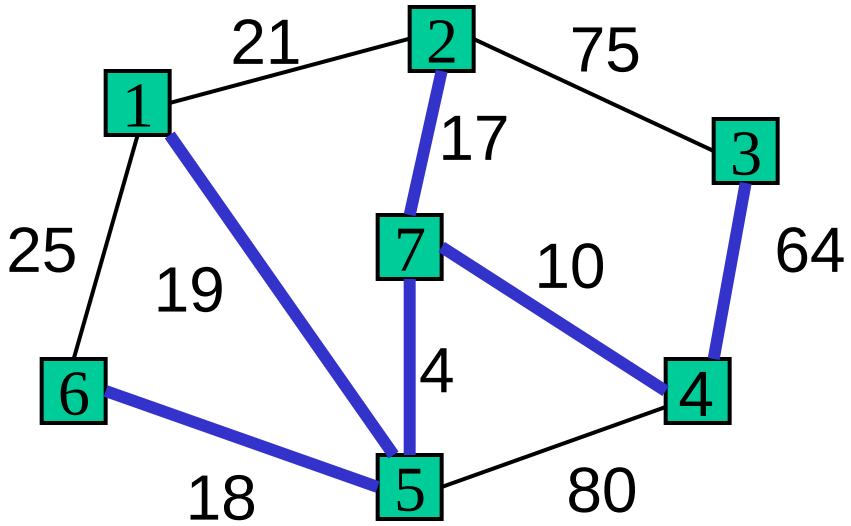
- Single source: given a vertex s, find the shortest path from s to any other vertex of G.
- Variants:
  - Different edges have different lengths (delay, cost, etc.)
  - Nonnegative/ any weight. Negative cycles.
- All-pair shortest path problem: no specific source.

Shortest-path algorithms – Summary of classical algorithms

- Single source, no weights:
  BFS - $O(m)$
- Single source, non-negative weights:
  Dijkstra $O((n + m) \log n))$ or $O(n^2)$
- Single source, arbitrary weights:
  Bellman-Ford: $O(nm)$
- All-pair shortest path, arbitrary weights:
  Floyd: $O(n^3)$

Minimum Spanning Tree

- Each edge has a cost.
- Find a minimal-cost subset of edges that will keep the graph connected. (must be a ST).



Price of this tree = 18+19+4+10+17+64

Minimum Spanning Tree Problem

- Input: Undirected connected graph $G = (V,E)$ and a cost function C from E to the reals. $C(e)$ is the cost of edge e.
- Output: A spanning tree T with minimum total cost. That is: T that minimizes

$$C(T) = \sum_{e \in T} C(e)$$

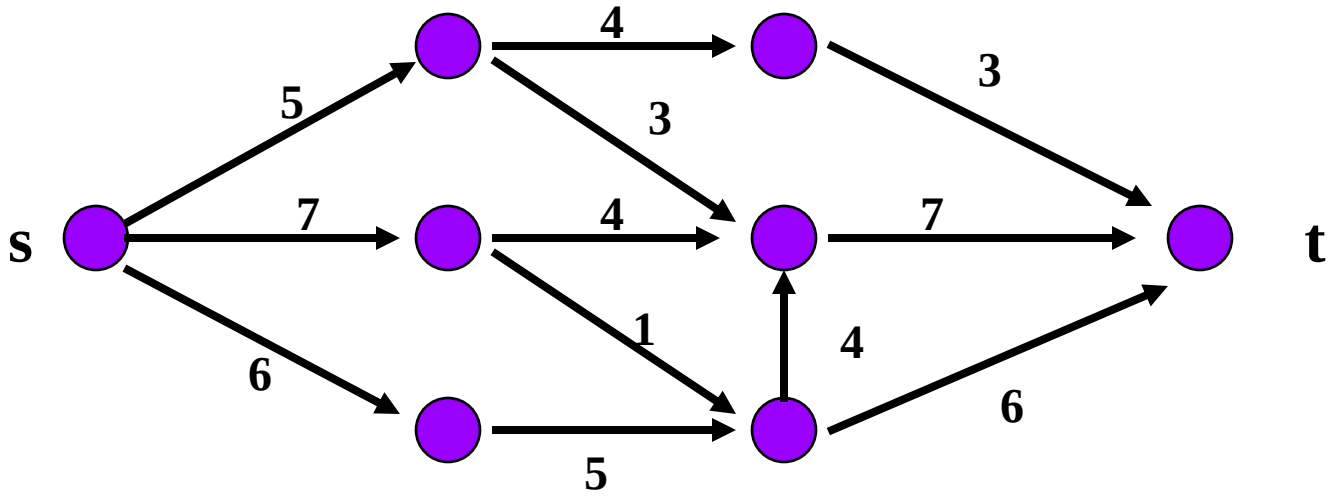
- Another formulation: Remove from G edges with maximal total cost, but keep G connected.

Two Popular algorithms:

- Kruskal
  - Simple (Greedy)
  - Good with sparse graphs - O(m log m)
- Prim
  - More complicated
  - A bit better with dense graphs - O(m log n)

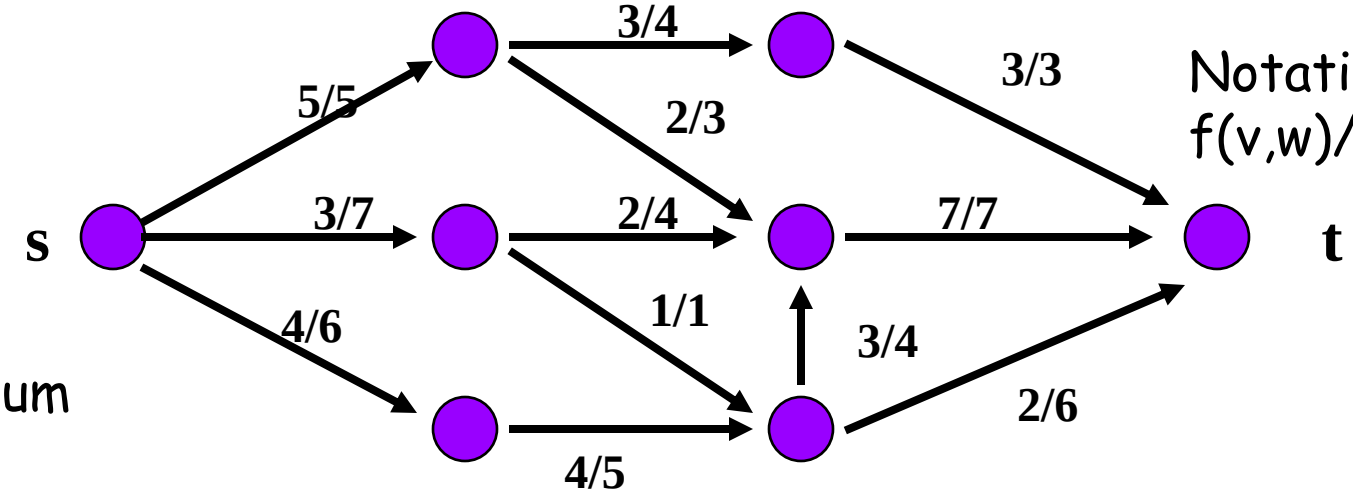
Maximum Flow

- Input: a directed graph (network) G
  - each edge (v,w) has associated capacity c(v,w)
  - a specified source node s and target node t
- Problem: What is the maximum flow you can route from s to t while respecting the capacity constraint of each edge?



Properties of Flow: f(v,w) - flow on edge (v,w)

- **Edge condition:** $0 \le f(v,w) \le c(v,w)$: the flow through an edge cannot exceed the capacity of an edge.
- **Vertex condition:** for all v except s,t : $\Sigma_u f(u,v) = \Sigma_w f(v,w)$: the total flow entering a vertex is equal to total flow exiting this vertex.
- total flow leaving s = total flow entering t.

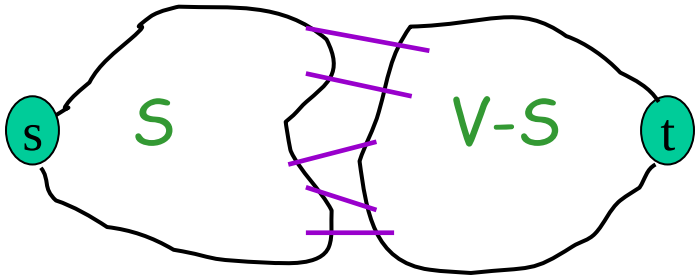


Not a maximum flow!

Cut

Cut - a set of edges that separates s from t.

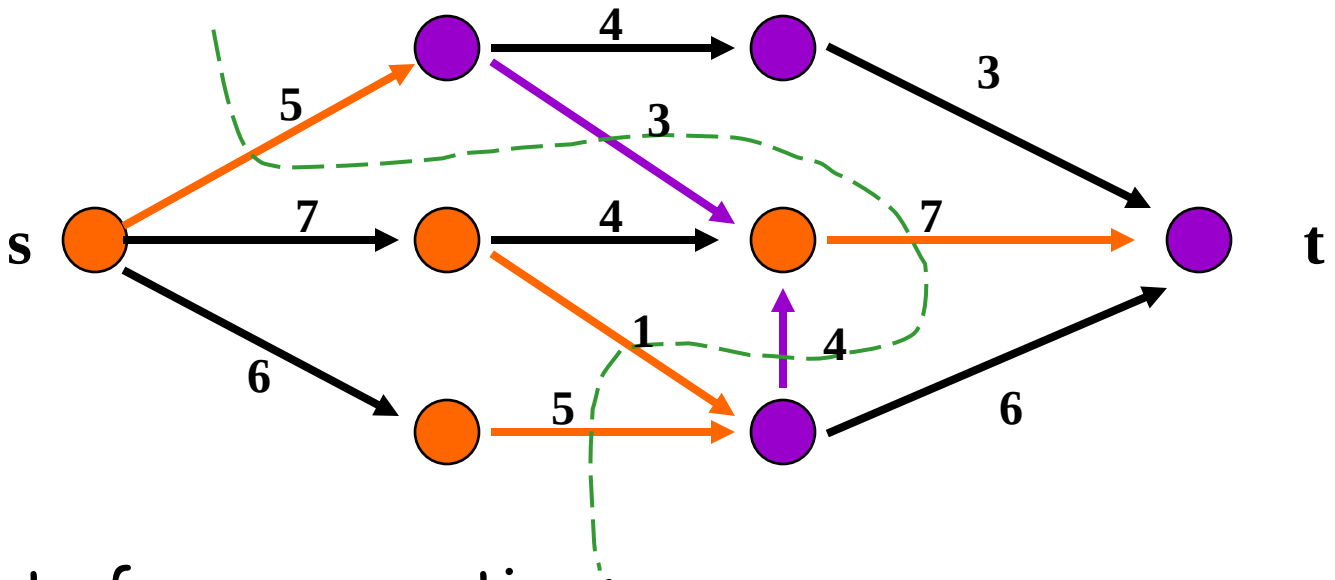
- A cut is defined by a set of vertices, S. This set includes s and maybe additional vertices reachable from s. The sink t is not in S.
- The cut is the set of edges (u,v) such that $u \in S$ and $v \notin S$, or $v \in S$ and $u \notin S$.



out(S) – edges in the cut directed from S to V-S

in(S) – edges in the cut directed from V-S to S

Cut - example



S – set of orange vertices.

out(S) – orange edges

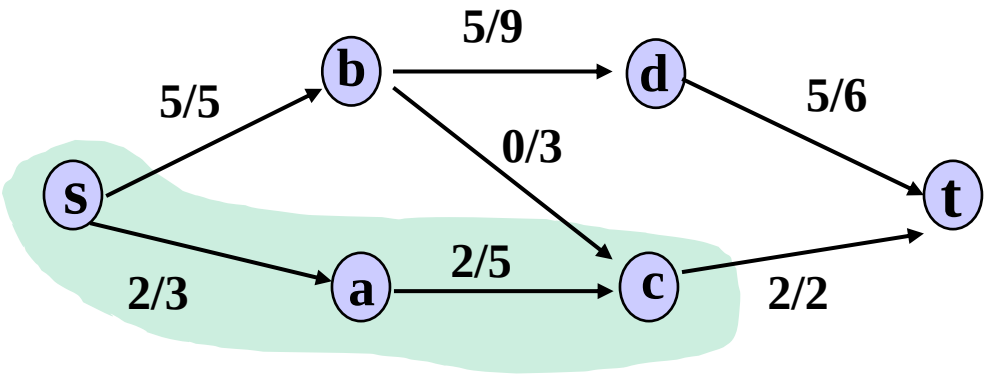
in(S) – purple edges

The capacity of this cut = 18

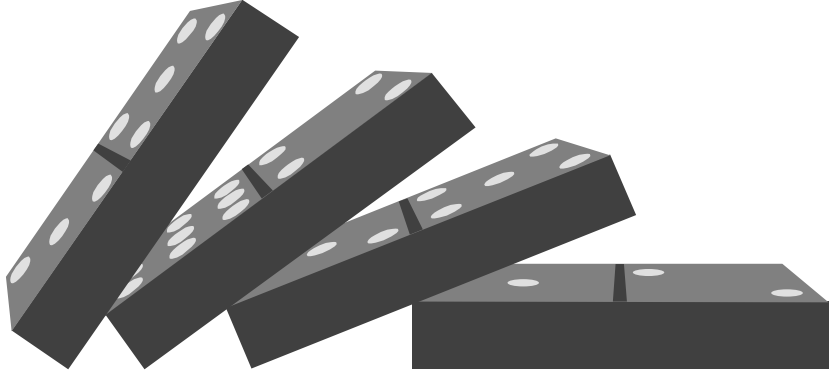
For a cut S, the *capacity of S* is $c(S) = \sum_{e \in out(S)} c(e)$.

Max-flow Min-Cut Theorem

The value of a maximum flow in a network is equal to the minimum capacity of a cut.



Example: Flow value = max-cut capacity = 7



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 $a \neq 1$

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

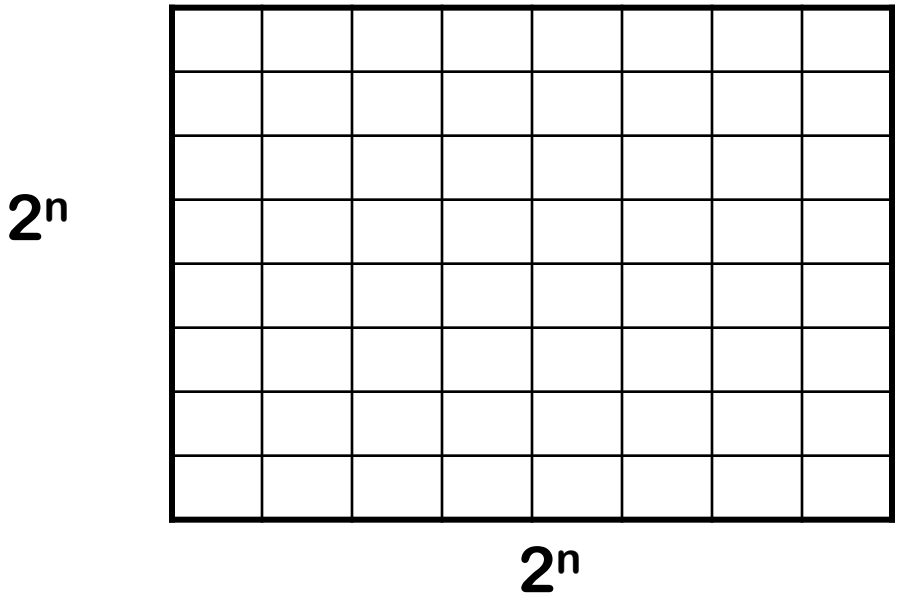
- Base: n=0. $a^0 = \frac{a^{0+1}-1}{a-1}$.

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

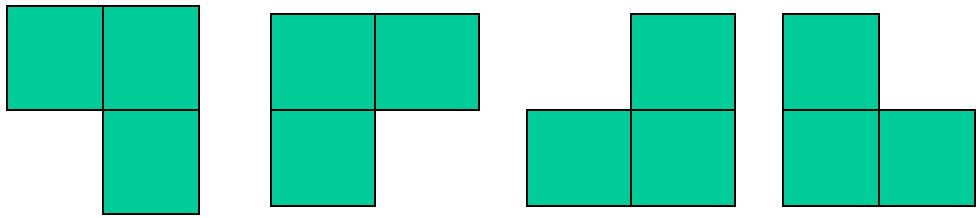
- Inductive hypothesis:
  - Assume $\sum_{i=0}^{n} a^i = \frac{a^{n+1}-1}{a-1}$
- Step (show true for n+1): In Class.

Design by Induction. Example: Tiling

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



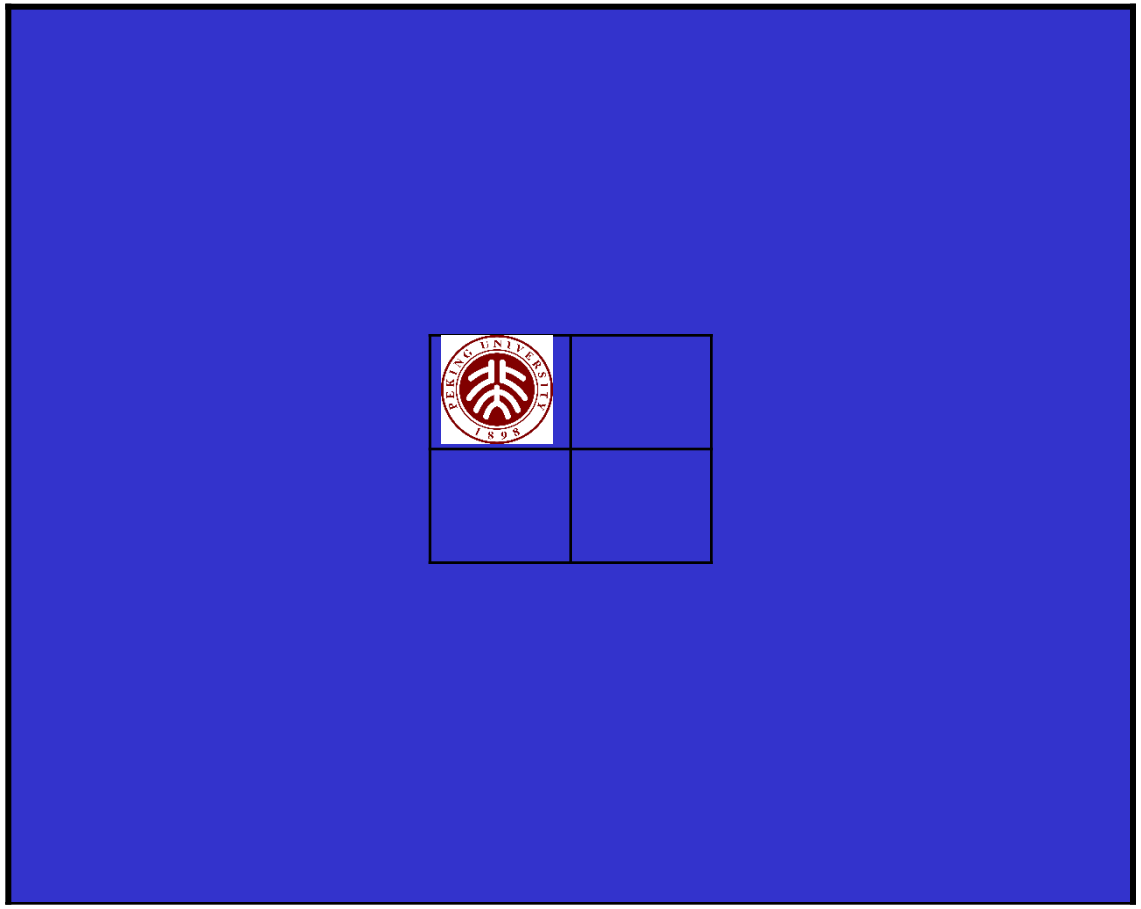
Architect allows only L-shaped tiles covering 3 squares



Tiling

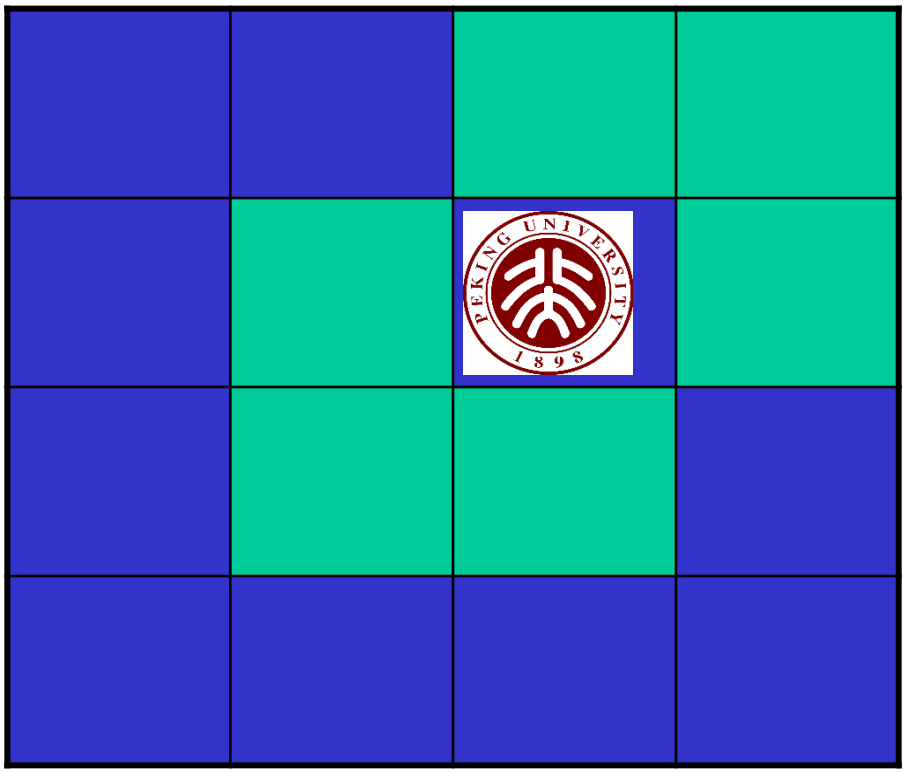
A tile in the middle is reserved to PKU logo.

Middle = one of the four middle squares.



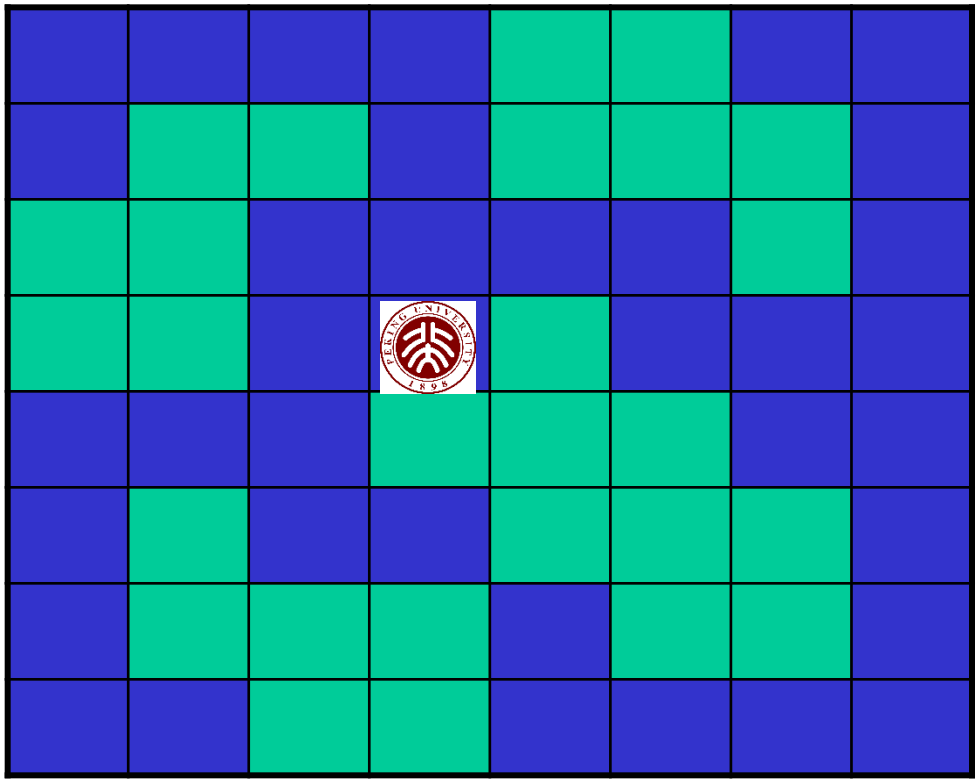
Tiling

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



Tiling

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



Tiling

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

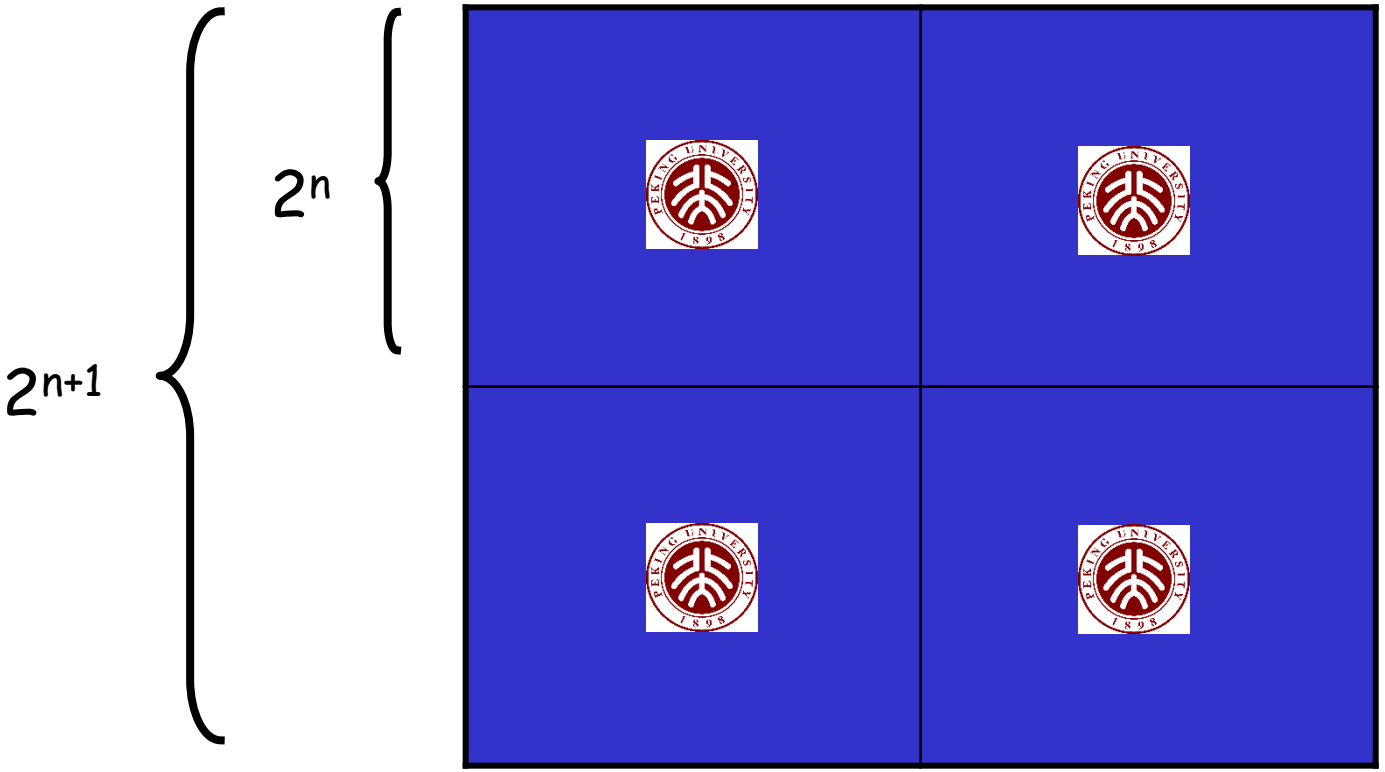
Proof: By induction on n.

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

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

Tiling

Induction step: assume can tile $2^n \times 2^n$ square, prove that can tile $2^{n+1} \times 2^{n+1}$ square.



Now what??

Tiling

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 x 2^n square with PKU logo in any selected location]

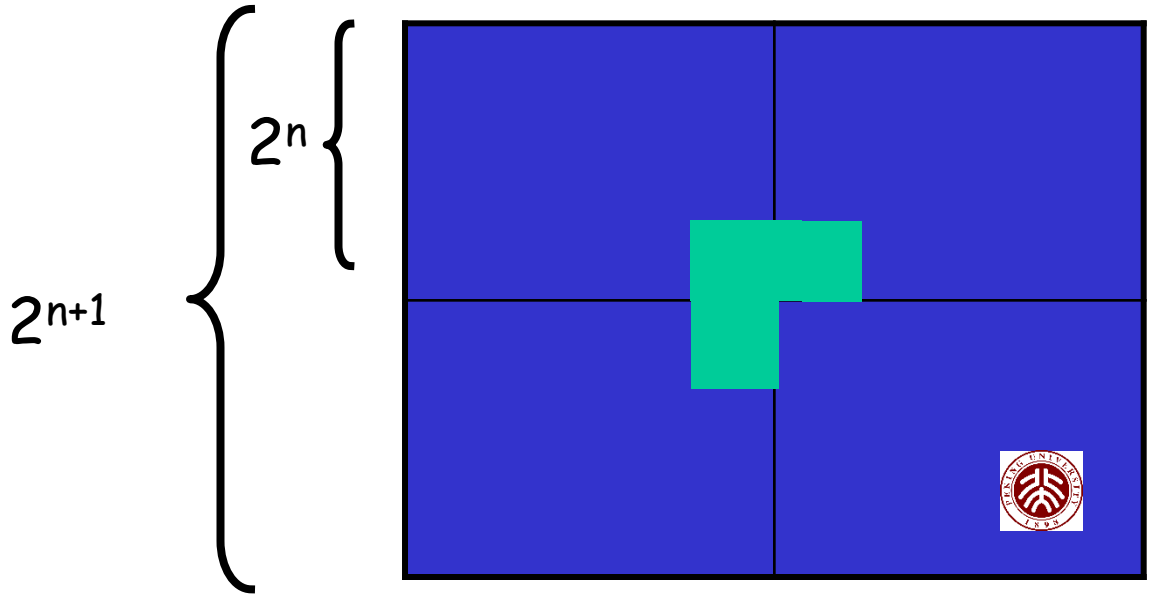
Base case: Still true for n = 0 (no tiles are needed).

Induction step: Assume can tile 2^n x 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:

demo

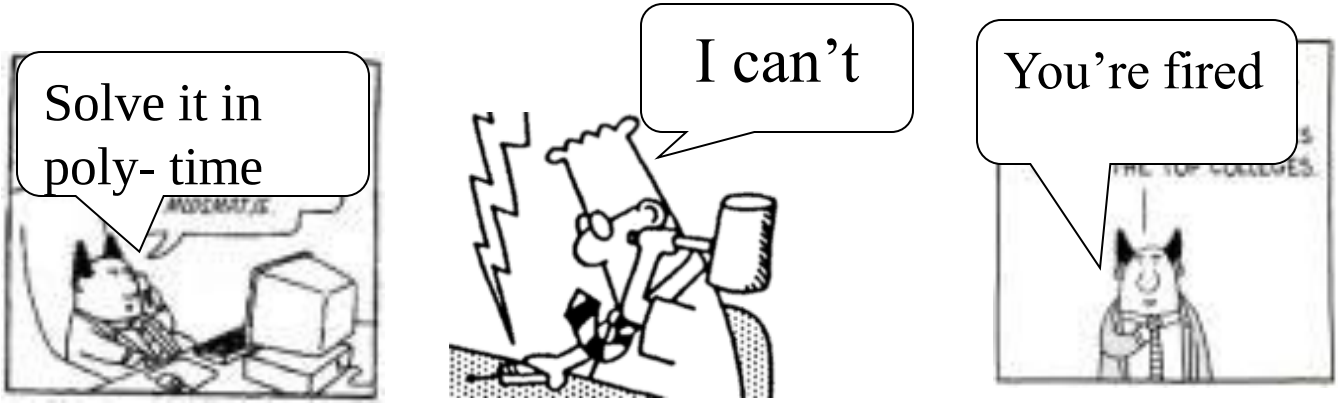
1. Sometimes yields an efficient procedure/algorithm.
2. Our proof implicitly defined a recursive tiling algorithm.

Choice of the induction hypothesis is crucial:

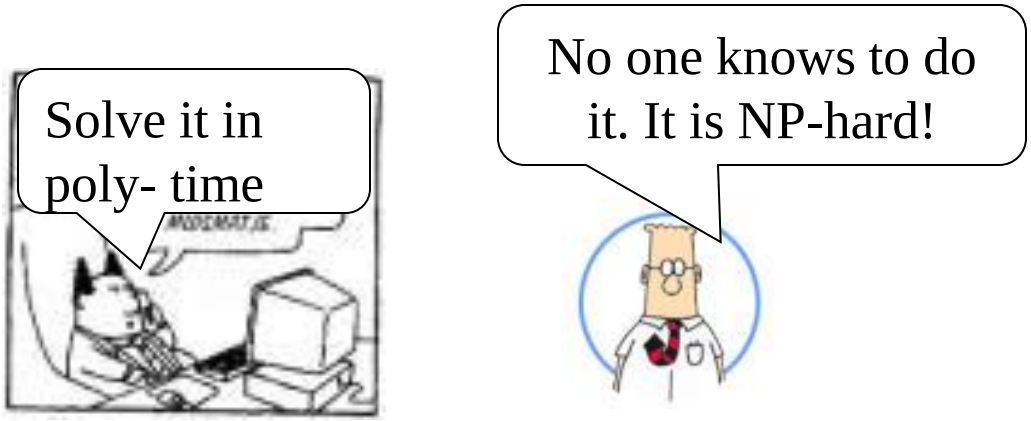
1. Assuming stronger hypothesis may make proof easier!
2. But need to ensure that $P(n) \Rightarrow P(n+1)$ is indeed true.

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.

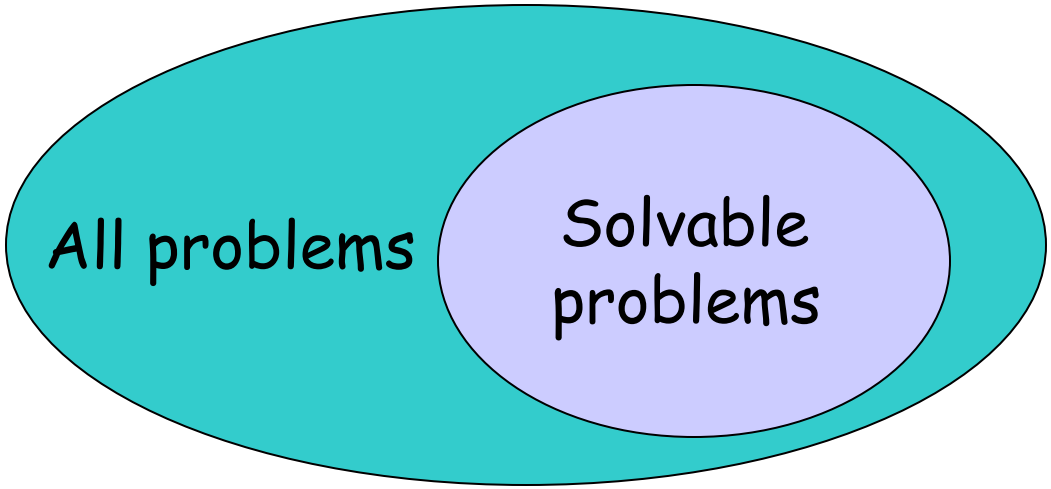
Polynomial-Time Algorithms

- 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



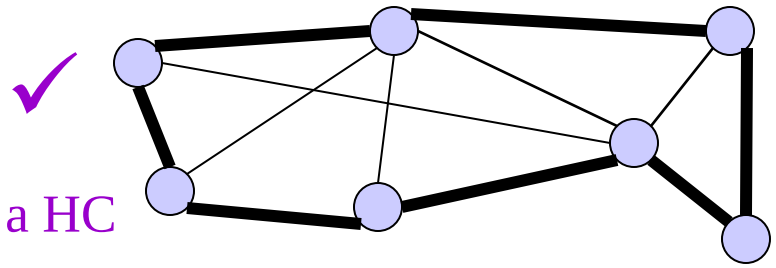
We will explore the 'solvable area', and will distinguish between problems that can be solved efficiently and those that cannot be solved efficiently.

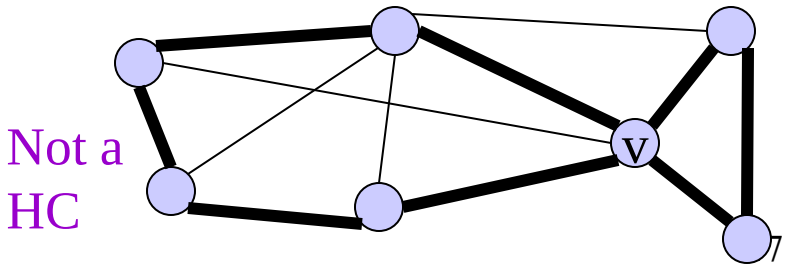
NP-Complete Problems

- 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)

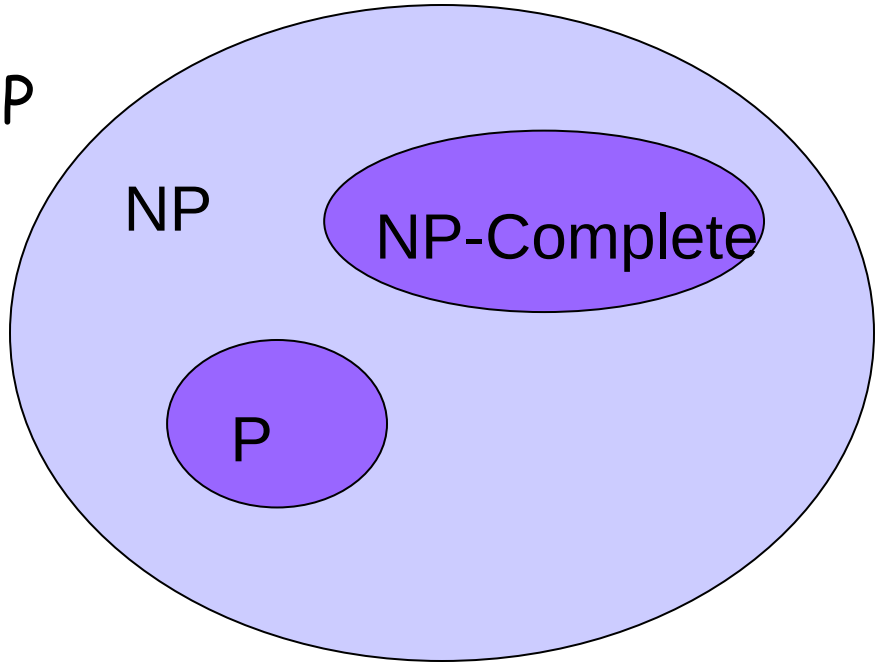
- Hamiltonian-cycle problem is in **NP**:
  - Cannot solve in polynomial time.
  - Easy to verify solution in polynomial time.

NP-Complete Problems

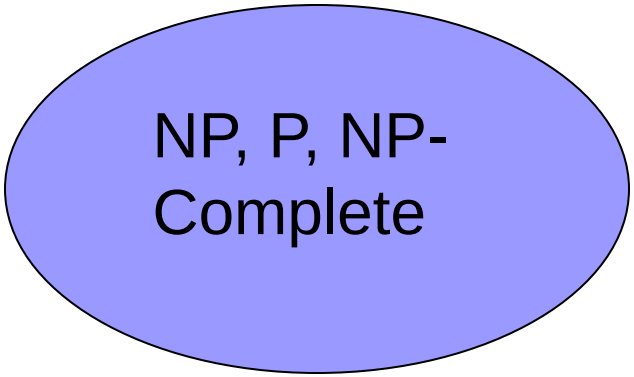
- NP-Complete problems are the "hardest" problems in NP:
  - If any *one* NP-Complete problem can be solved in polynomial time…
  - …then *every* NP-Complete problem can be solved in polynomial time…
  - …and in fact *every* problem in **NP** can be solved in polynomial time (which would show **P = NP**)
  - 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?

- 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 $P \in \mathbf{NP}$
- 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

Conjunctive Normal Form

- Even if the form of the Boolean expression is simplified, the problem may be NP-Complete
  - *Literal*: an occurrence of a Boolean or its negation
  - A Boolean formula is in *conjunctive normal form*, or *CNF*, if it is an AND of clauses, each of which is an OR of literals
    - Ex: $(x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_3 \vee x_4) \wedge (\neg x_5)$

  3-CNF: each clause has exactly 3 distinct literals
  - Ex: $(x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_3 \vee x_4) \wedge (\neg x_5 \vee x_3 \vee x_4)$
  - Note: true if at least one literal in each clause is true

The 3-CNF Problem

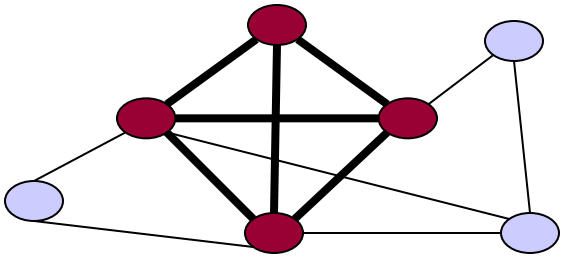
- Theorem: Satisfiability of Boolean formulas in 3-CNF form (the *3-CNF Problem*) is NP-Complete
  - Proof: not here
- The reason we care about the 3-CNF problem is that it is relatively easy to reduce to others.
  - Thus, knowing that 3-CNF is NP-Complete we can prove many seemingly unrelated problems are NP-Complete.

The k-clique Problem

- *A clique in a graph G is* a subset of vertices fully connected to each other, i.e. a complete subgraph of G.
- The *clique problem*: how large is the maximum-size clique in a graph?
- *Can we turn this into a decision problem?*
- A: Yes, we call this the *k-clique problem*
- *Is the k-clique problem within* ***NP****?*

  Yes: Nondeterministic algorithm: guess k vertices then check that there is an edge between each pair of them.

4-clique:



3-CNF $\leq_p$ Clique

- How can we prove that k-clique is NP-hard?
- We need to show that if we can solve k-clique then we can solve a problem which is known to be NP-hard.
- We will do it for 3-CNF:
- Given a 3-CNF formula, we will transform it to an instance of k-clique (a graph and a number k), for which a k-clique exists iff the 3-CNF formula is satisfiable.

3-CNF $\leq_p$ Clique

- The reduction:
  - Let $F = C_1 \wedge C_2 \wedge \ldots \wedge C_k$ be a 3-CNF formula with k clauses, each of which has 3 distinct literals.
  - For each clause, put three vertices in the graph, one for each literal.
  - Put an edge between two vertices if they are in different triples and their literals are consistent, meaning not each other's negation.

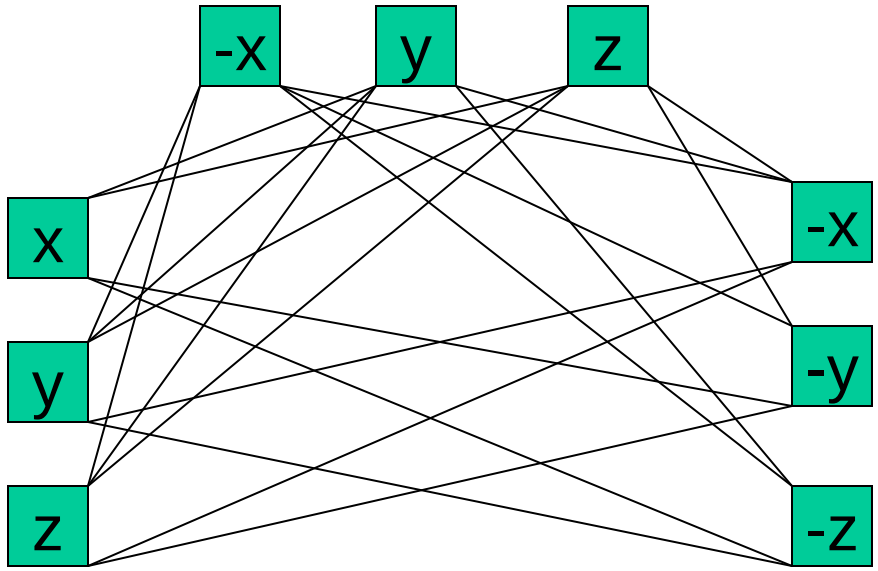
Construction by Example

$$F = (x \vee y \vee z) \wedge (\neg x \vee y \vee z) \wedge (\neg x \vee \neg y \vee \neg z)$$

literal

clause

G



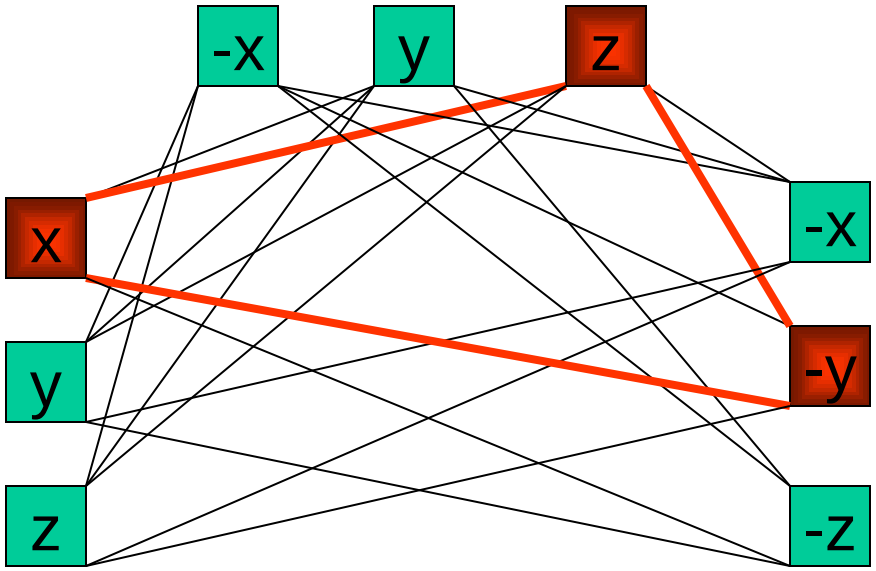
An edge means 'these two literals do not contradict each other'.

Construction by Example

$$F = (x \vee y \vee z) \wedge (\neg x \vee y \vee z) \wedge (\neg x \vee \neg y \vee \neg z)$$

$$x = 1, y = 0, z = 1$$

G



Any clique of size k must include exactly one literal from each clause.

General Construction

$$F = \bigcap_{i=1}^{k} \bigcup_{j=1}^{3} a_{ij} \quad \text{where } a_{ij} \in \{x_1, \neg x_1, \ldots, x_n, \neg x_n\}$$

literals

$G = (V, E)$ where

$V = \{a_{ij} : 1 \le i \le k, 1 \le j \le 3\}$

$E = \{(a_{ij}, a_{i',j'}) : i \neq i' \text{ and } a_{ij} \neq \neg a_{i'j'}\}$

k is the number of clauses

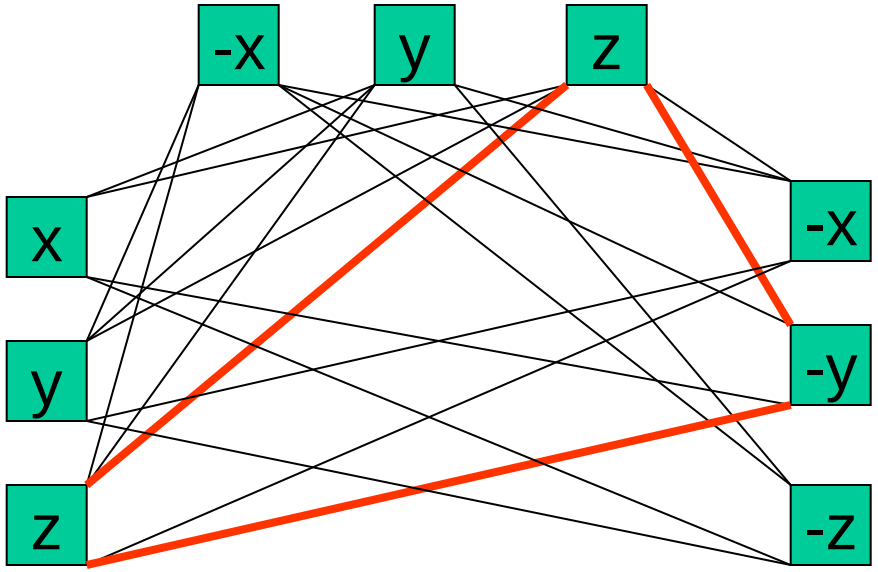
The Reduction Argument

- We need to show
  - F satisfiable implies G has a clique of size k.
    - Given a satisfying assignment for F, for each clause pick a literal that is satisfied. Those literals in the graph G form a k-clique.
  - G has a clique of size k implies F is satisfiable.
    - Given a k-clique in G, assign TRUE to each literal in the clique. This yields a satisfying assignment to F (why?).

Clique to Assignment

$$F = (x \vee y \vee z) \wedge (\neg x \vee y \vee z) \wedge (\neg x \vee \neg y \vee \neg z)$$

G



$$y = 0,\ z = 1$$

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

Hamiltonian Cycle $\leq_p$ TSP

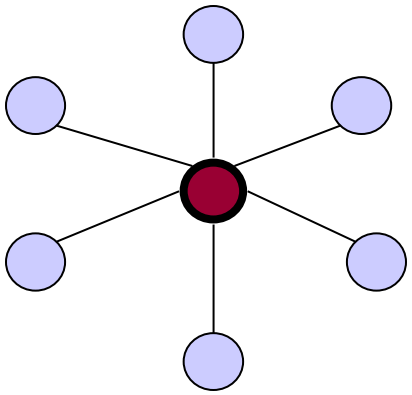
- The hamiltonian-cycle problem: given a graph G, is there a simple cycle that contains every vertex?
- To transform ham. cycle problem on graph $G = (V,E)$ to TSP, create graph $G' = (V,E')$:
- G' is a complete graph
- Edges in E' also in E have cost 0
- All other edges in E' have cost 1
- TSP: is there a TS cycle on G' with cost 0?
  - If G has a ham. cycle, G' has a TS cycle with cost 0
  - If G' has TS cycle with cost 0, every edge of that cycle has cost 0 and is thus in G. Thus, G has a ham. cycle.

Other NP-Complete Problems

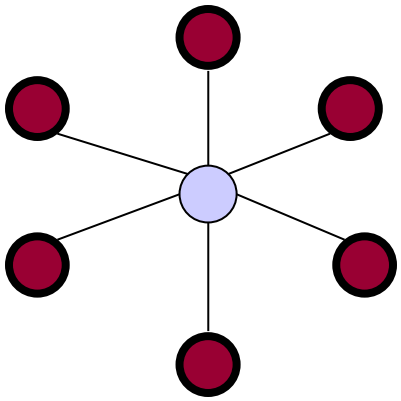
- Partition: Given a set of integers, whose total sum is 2S, can we partition them into two sets, each adds up to S?
- Subset-sum: Given a set of integers, does there exist a subset that adds up to some target *T*?

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.



Maximal independent set



Maximum independent set

Steiner Tree

- Input: A graph G=(V,E), a subset T of the vertices V, and a bound B
- Problem: Is there a tree connecting all the vertices of T of total weight at most B?

- Application: Network design and wiring layout.
- The case T=V is polynomially solvable (this is the Minimum Spanning Tree problem).

3-Partition

- Input: A set of numbers $A = \{a_1, a_2, \ldots, a_{3m}\}$ and a number B such that $B/4 < a_i < B/2$ and

$$\sum_{i=1}^{3m} a_i = mB.$$

- Output: Determine if A can be partitioned into $S_1, S_2, \ldots, S_m$ such that for all i

$$\sum_{j \in S_i} a_j = B.$$

Note: each S_i must contain exactly 3 elements.

Example of 3-Partition

- $A = \{26, 29, 33, 33, 33, 34, 35, 36, 41\}$
- $B = 100$, $m = 3$
- 3-Partition:
  - 26, 33, 41
  - 29, 36, 35
  - 33, 33, 34

Bin Packing

- Input: A set of numbers $A = \{a_1, a_2, \ldots, a_m\}$ and numbers B (capacity) and K (number of bins).
- Output: Determine if A can be partitioned into $S_1, S_2, \ldots, S_K$ such that for all i

$$\sum_{j \in S_i} a_j \leq B.$$

Bin Packing Example

- A = {2, 2, 3, 3, 3, 4, 4, 4, 5, 5, 5}
- B = 10, K = 4
- Bin Packing:
  - 3, 3, 4
  - 2, 3, 5
  - 5, 5
  - 2, 4, 4

Perfect fit!

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 the problem (if possible…)

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.