EC504 ALGORITHMS AND DATA STRUCTURES FALL 2020 MONDAY & WEDNESDAY 2:30 PM - 4:15 PM

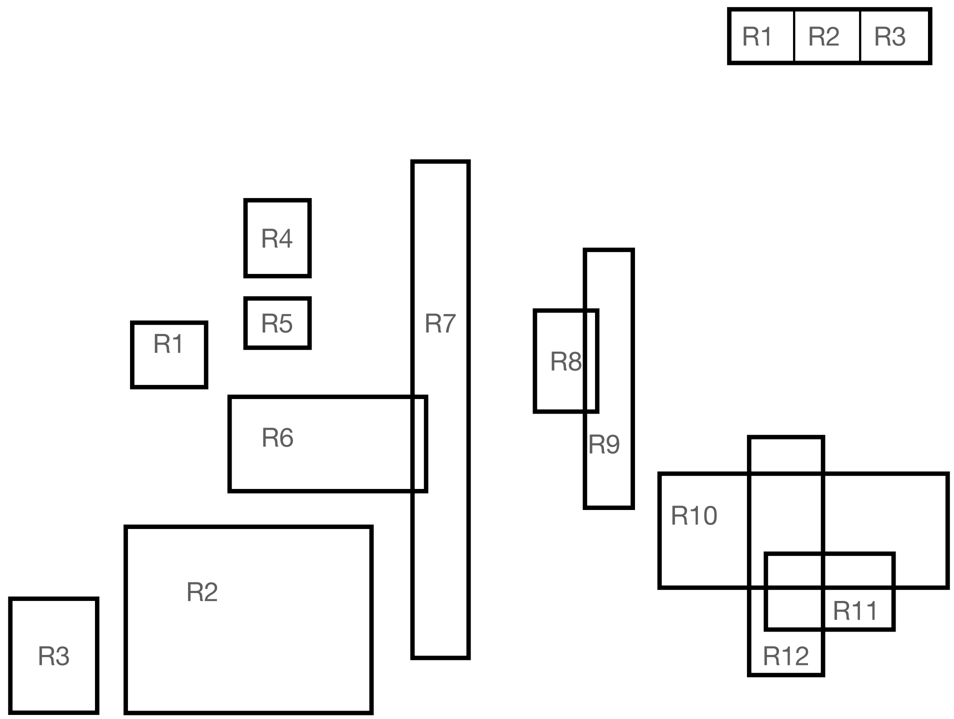
g	6	6	t

Prof: David Castañón, dac@bu.edu

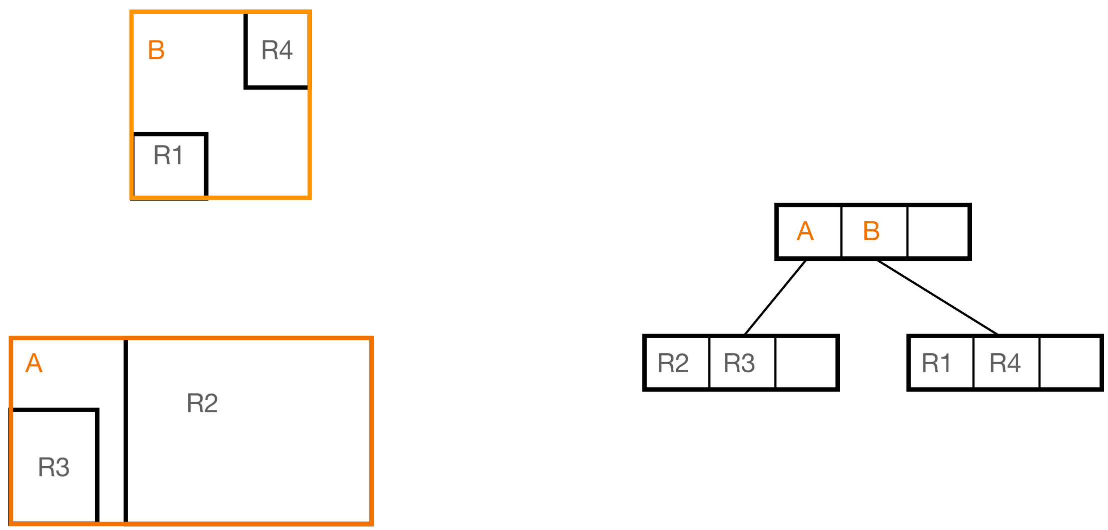
GTF: Mert Toslali, toslali@bu.edu

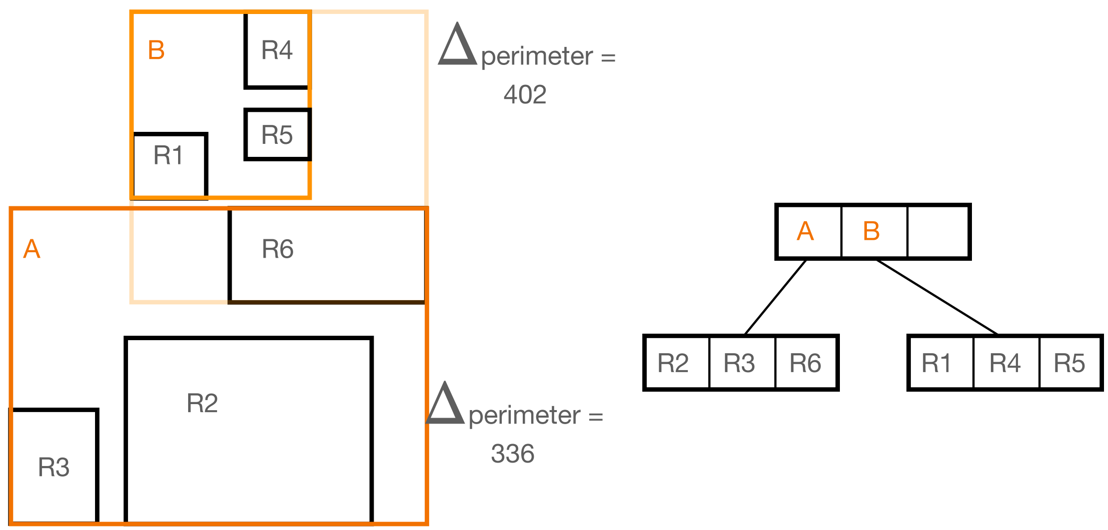
Haoyang Wang: haoyangw@bu.edu

Christopher Liao: cliao25@bu.edu

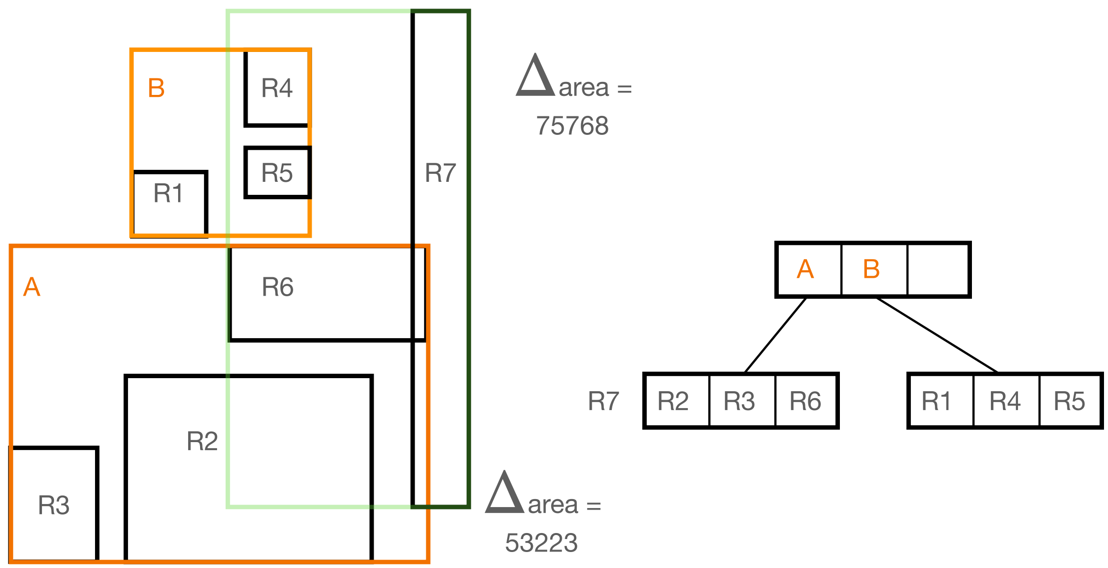

R1
R2
R3
R4
R5
R1
R7
R6
R8
R9
R10
R2
R3
R11
R12

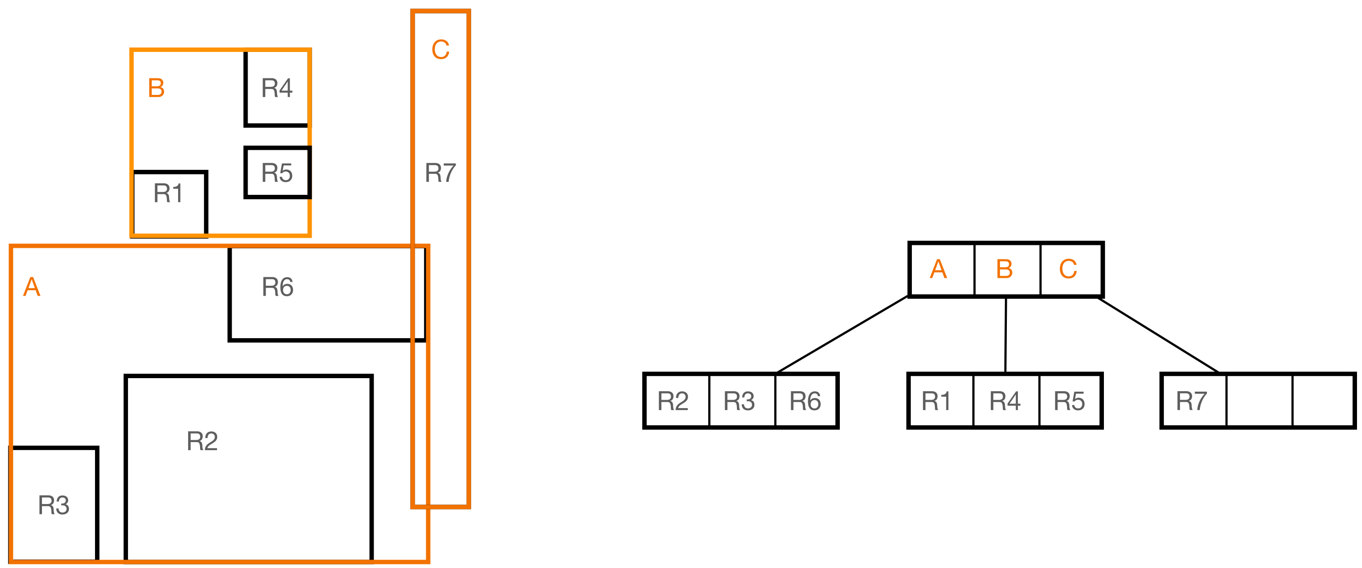
Seeds R3, R4
for split



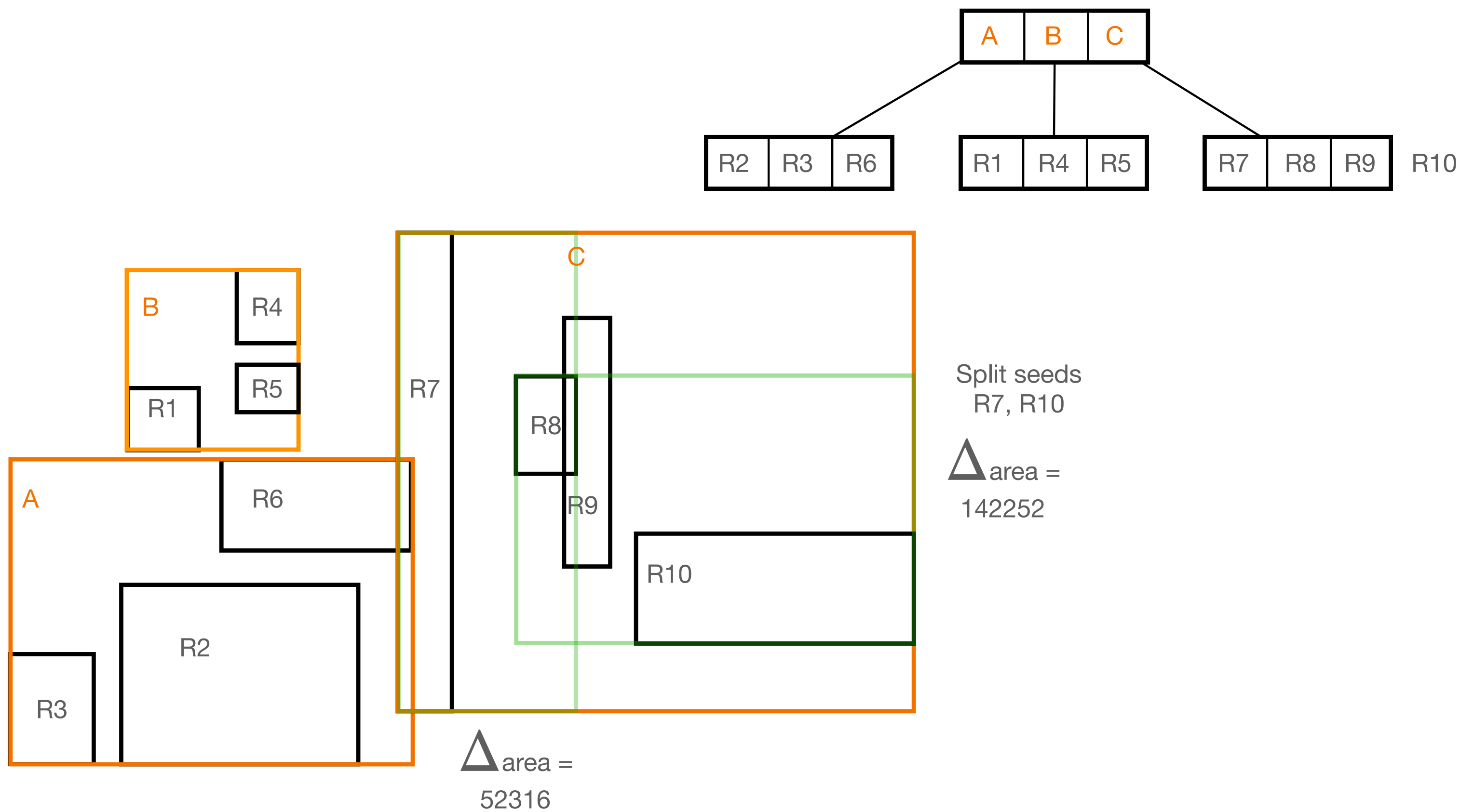


B
R4
R5
R1
A
R6
R2
R3
Δperimeter = 402
Δperimeter = 336
A
B
R2
R3
R6
R1
R4
R5

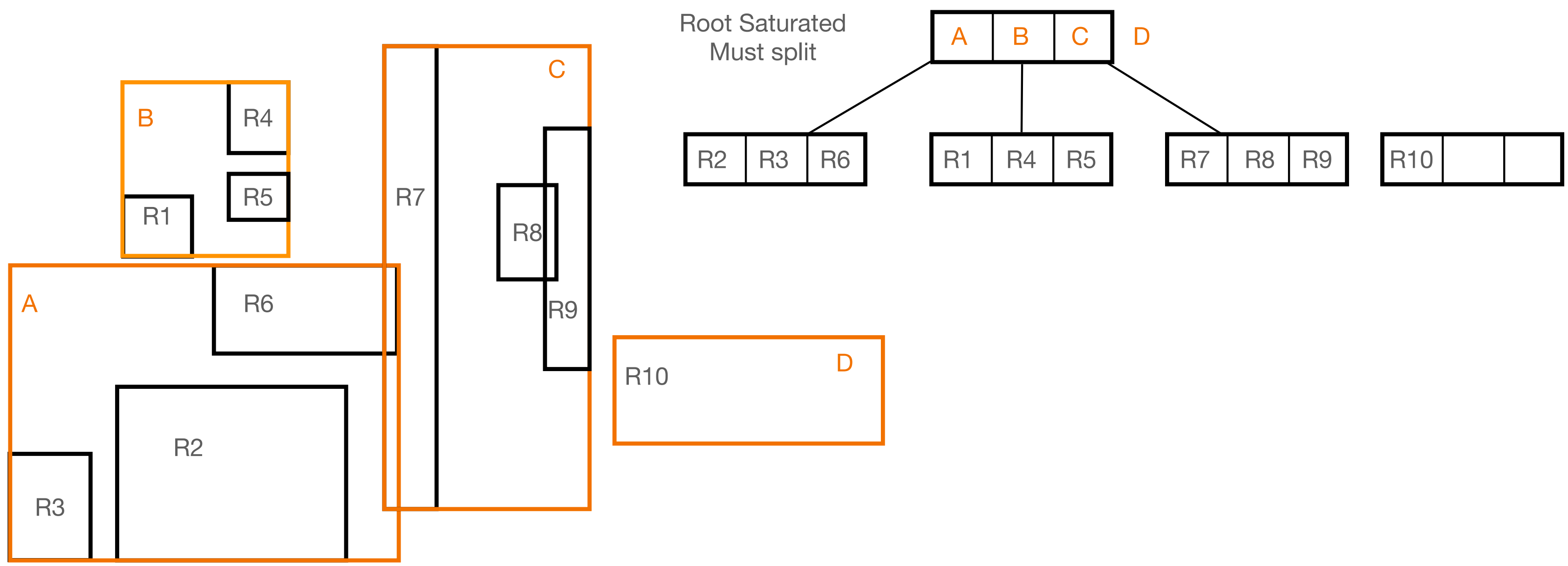

B
R4
R5
R1
R7
A
R6
R2
R3
Δarea =
75768
Δarea =
53223
A
B
R7
R2
R3
R6
R1
R4
R5

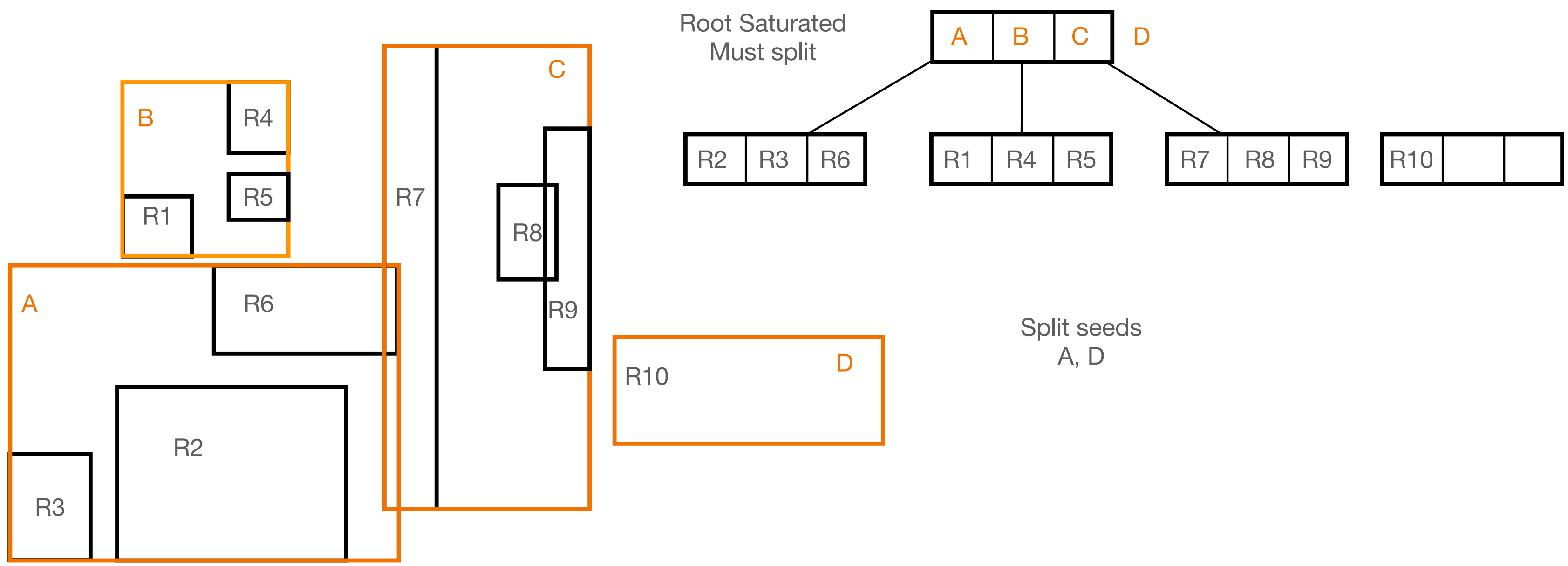


B
R4
R5
R1
C
R7
A
R6
R2
R3
A
B
C
R2
R3
R6
R1
R4
R5
R7

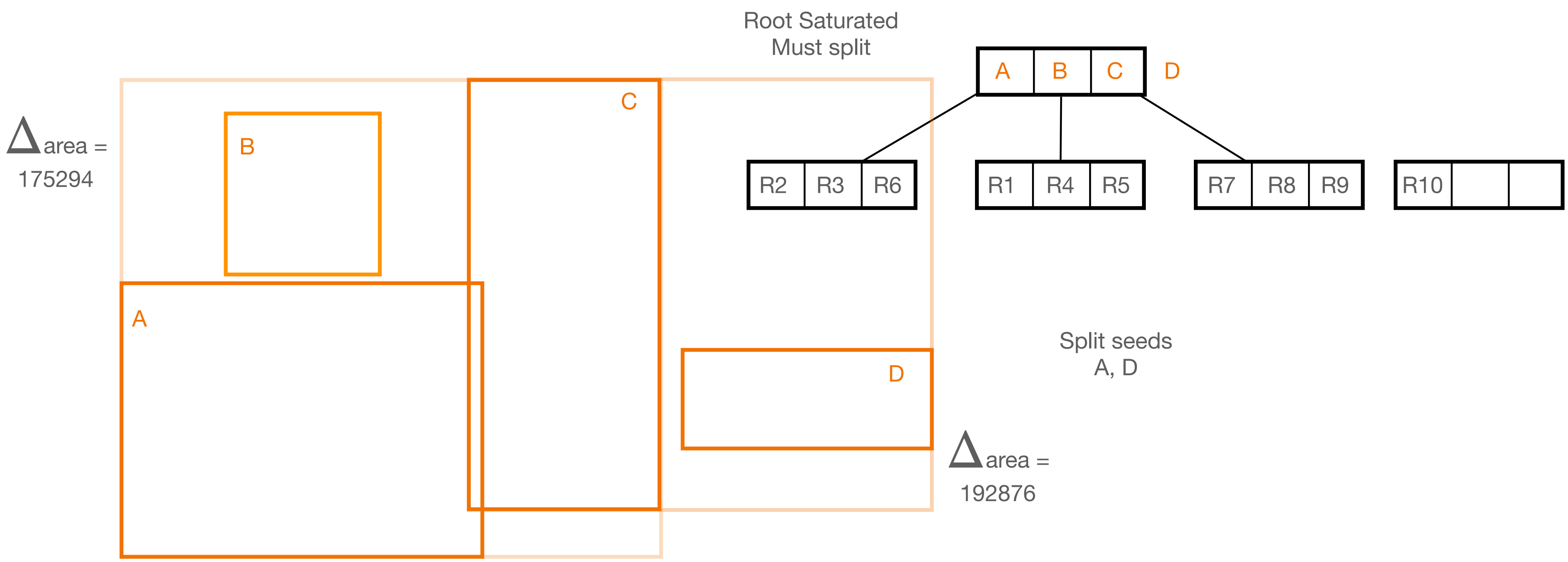


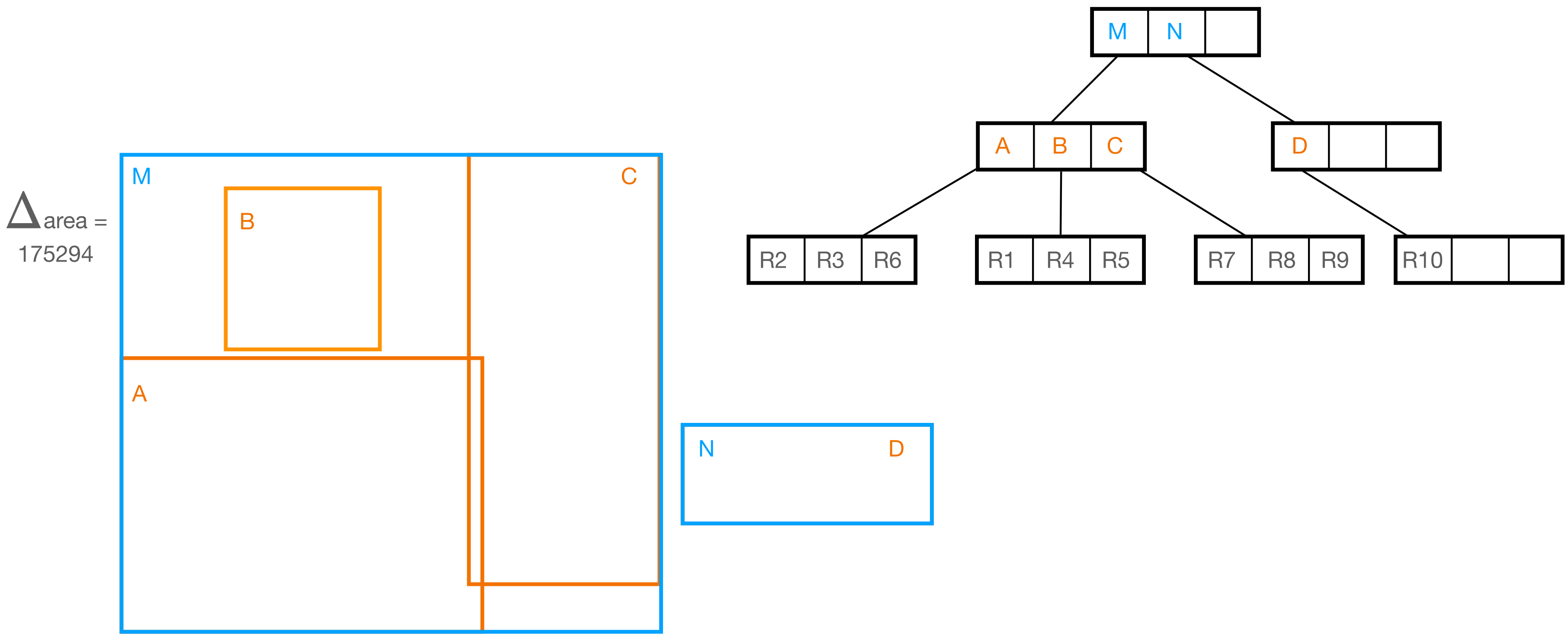
A
B
C
R2
R3
R6
R1
R4
R5
R7
R8
R9
R10
B
R4
R5
R1
A
R6
R2
R3
R7
C
R8
R9
R10
Split seeds
R7, R10
Δarea =
142252
Δarea =
52316


Root Saturated
Must split
A
B
C
D
R2
R3
R6
R1
R4
R5
R7
R8
R9
R10
B
R4
R5
R1
C
R7
R8
R9
A
R6
R2
R3
R10
D

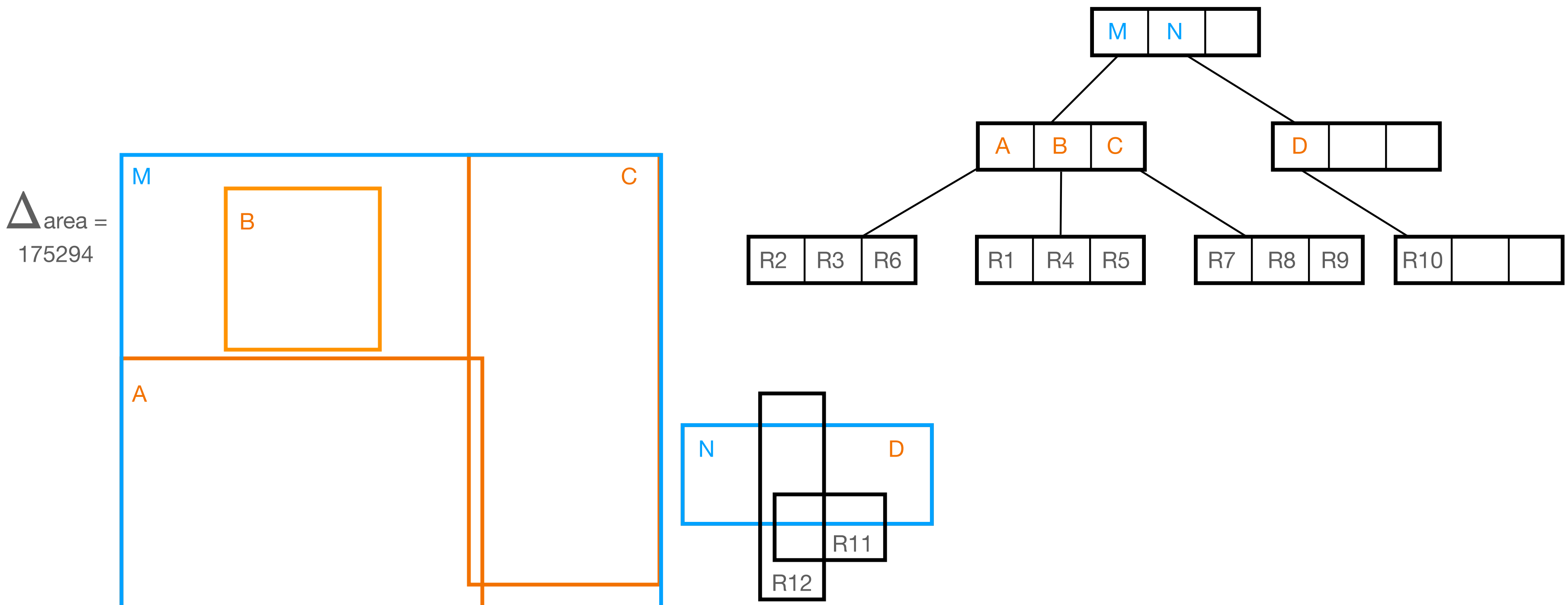


Root Saturated
Must split
A
B
C
D
R2
R3
R6
R1
R4
R5
R7
R8
R9
R10
B
R4
R5
R1
C
R7
R8
R9
A
R6
R2
R3
R10
D
Split seeds
A, D

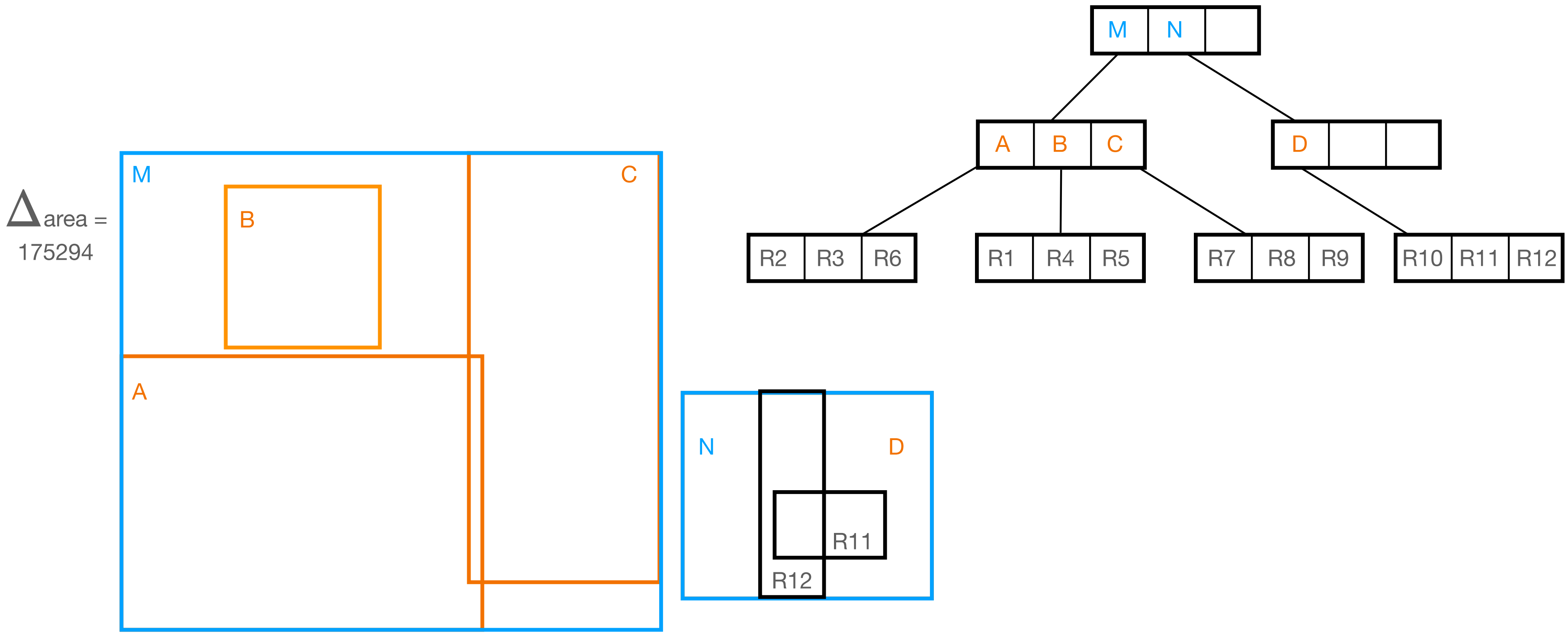

Root Saturated
Must split
A
B
C
D
R2
R3
R6
R1
R4
R5
R7
R8
R9
R10
Δ area =
175294
Split seeds
A, D
Δ area =
192876



Δarea =
175294
M
B
C
A
N
D
M
N
A
B
C
D
R2
R3
R6
R1
R4
R5
R7
R8
R9
R10



Δ area =
175294
M
B
C
A
N
D
R11
R12
M
N
A
B
C
D
R2
R3
R6
R1
R4
R5
R7
R8
R9
R10



Δarea =
175294
M
B
C
A
N
D
R11
R12
M N
A B C
D
R2 R3 R6
R1 R4 R5
R7 R8 R9
R10 R11 R12

A Theory of Computation

\

- While we have introduced many problems with polynomial-time algorithms…
  - We haven’t formally defined what this means
  - And not all problems enjoy fast computation
- Given a set I of problem instances, and a set S of problem solutions, an abstract problem is a binary relationship in I x S. That is, a set of pairs (i,s)
  - The problem shortest path associates each instance of a graph and an origin-destination with a sequence of vertices which connect the origin and destination
- D**ecision** problems have a yes or no solution. Abstract decision problem is a function which maps problem instances I into {yes, no}.
  - e.g. Is the shortest path in this graph between nodes 0 and nodes 30 have length > 10?

Decision Problem Instance

- If a machine is to solve an instance of a problem, the input must be specified in terms of a string of bits
  - We must encode problem instance I into a sequence of binary inputs
  - This helps understand the “size” of the problem instance
  - A concrete problem is an encoding of the set of problem instances to the set of binary strings
  - And not all problems enjoy fast computation

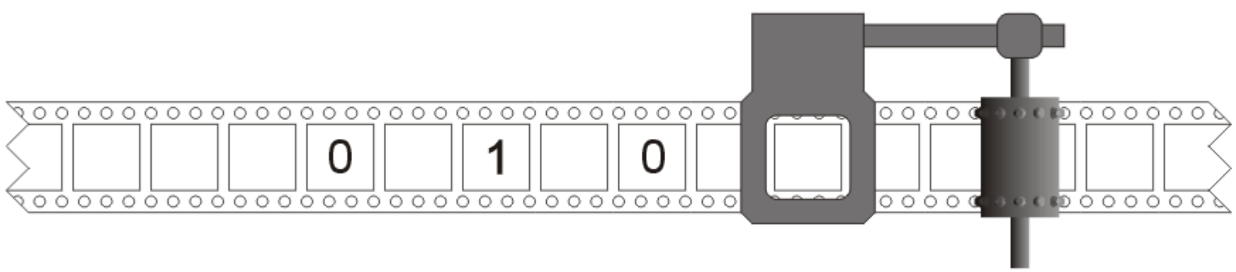
- An algorithm is a procedure that processes an concrete problem instance of size n and generates the correct answer
  - In what computer model?
  - Turing machine (1936) — Alan Turing (developed decoders for German coders (WW II))

Turing machine 1

\

The Turing machine has four components:

- An arbitrary-length tape
- **A head that can**
  - Read a symbol off the tape,
  - Write a symbol to the tape, and/or
  - Move to the next entry to the left or the right

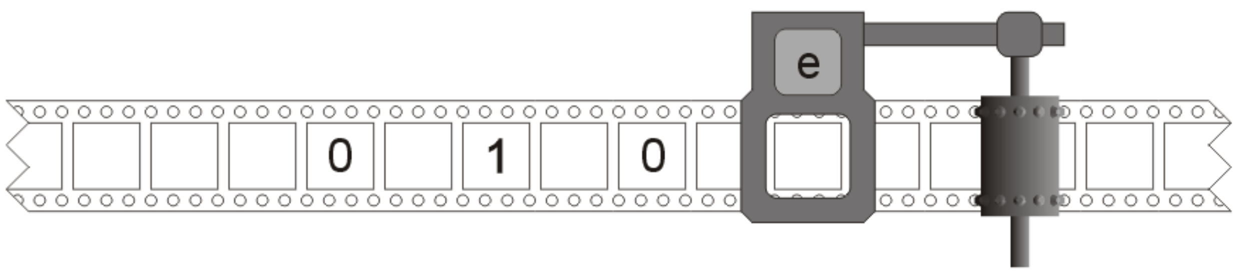


Turing machine 2

\

The Turing machine has four components:

- An arbitrary-length tape
- A head
- **A state**
  - The state is one of a finite set of symbols Q
  - In this example, $\mathrm{Q} = \{b, c, e, f\}$
  - The initial state of the machine is denoted $q_0 \in \mathrm{Q}$
  - Certain states may halt the computation



Turing machine 3

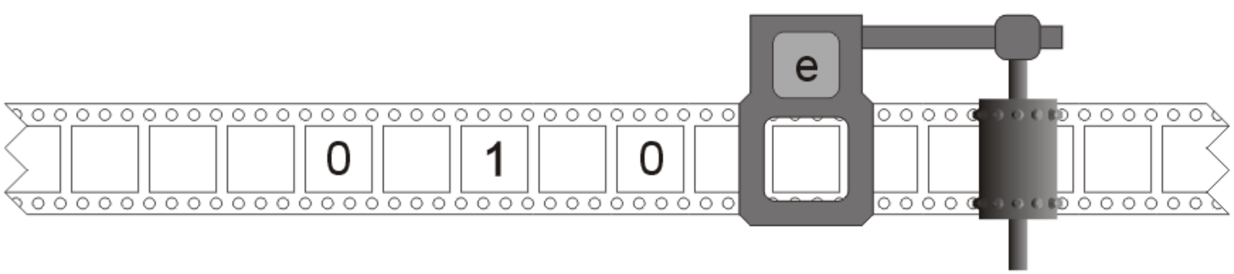
\

The Turing machine has four components:

- An arbitrary-length tape
- A head
- A state
- **A transition table** (this is your program!)
  - $Q \times \Gamma \to Q \times \Gamma \times \{\mathbf{L}, \mathbf{R}, \mathbf{N}\}$
  - **L** moves one entry to the left
  - **R** moves one entry to the right
  - **N** indicates no shift

Current		**New**		
State	Symbol read	State	Symbol to write	Direction
b	$\boldsymbol{B}$	c	0	**R**
c	$\boldsymbol{B}$	e	$\boldsymbol{B}$	**R**
e	$\boldsymbol{B}$	f	1	**R**
f	$\boldsymbol{B}$	b	$\boldsymbol{B}$	**R**

There is at most one entry in this table for each pair of current settings

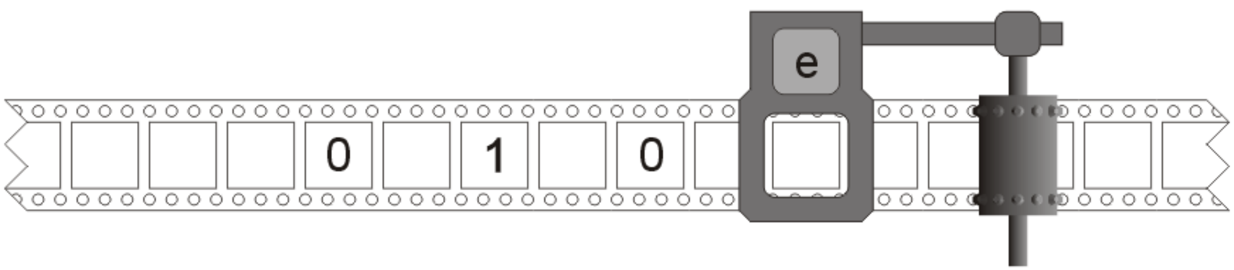


Example (Turing, '36)

A program to write 0 1 0 1 0 1 0 ⋯

Currently, the state is e and the symbol under the head is $\boldsymbol{B}$

Current		Next		
State	Symbol read	State	Symbol to write	Direction
b	$\boldsymbol{B}$	c	0	**R**
c	$\boldsymbol{B}$	e	$\boldsymbol{B}$	**R**
e	$\boldsymbol{B}$	f	1	**R**
f	$\boldsymbol{B}$	b	$\boldsymbol{B}$	**R**

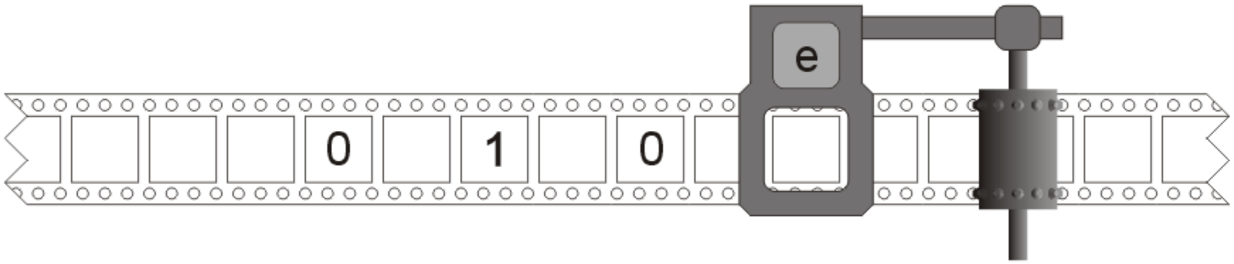


Example - 2

The transition table dictates that the machine must:

- The state is set to f
- Print symbol 1 onto the tape
- Move one entry to the right

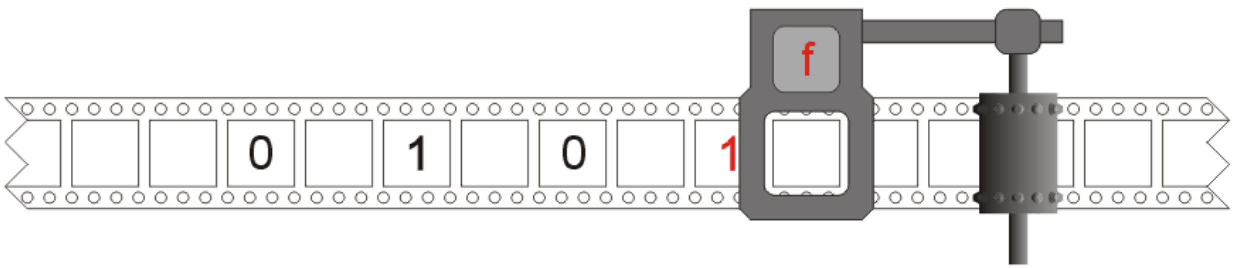
Current		Next		
State	Symbol read	State	Symbol to write	Direction
b	***B***	c	0	**R**
c	***B***	e	***B***	**R**
e	***B***	f	1	**R**
f	***B***	b	***B***	**R**



Example - 3

The state and symbol under the head have been updated

Current		Next		
State	Symbol read	State	Symbol to write	Direction
b	***B***	*c*	0	**R**
c	***B***	*e*	***B***	**R**
e	***B***	*f*	1	**R**
f	***B***	*b*	***B***	**R**



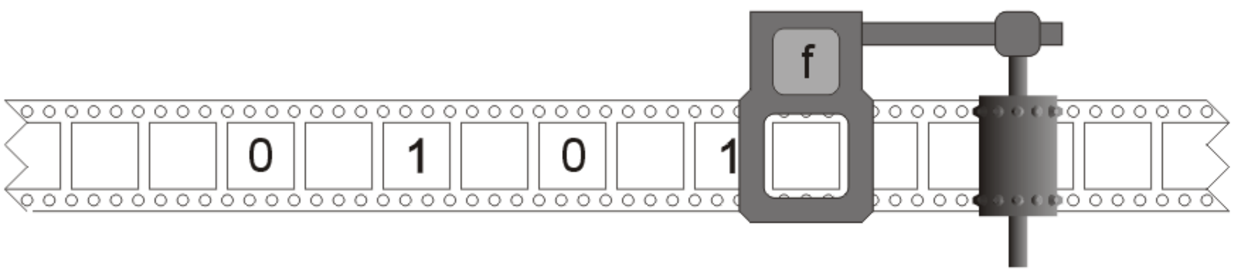
Example - 4

\

The state is f and the symbol under the head is the blank $\boldsymbol{B}$:t

- The state is set to b
- A blank is printed to the tape
- Move one entry to the right

Current		Next		
State	Symbol read	State	Symbol to write	Direction
b	$\boldsymbol{B}$	c	0	**R**
c	$\boldsymbol{B}$	e	$\boldsymbol{B}$	**R**
e	$\boldsymbol{B}$	f	1	**R**
f	$\boldsymbol{B}$	b	$\boldsymbol{B}$	**R**

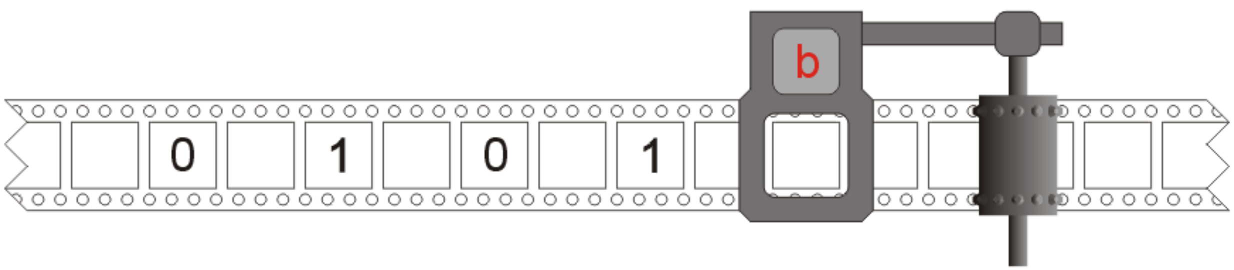


Example - 5

Again, the state is b, the symbol a blank, and therefore:

- Set the state to c
- Print the symbol 0 to the tape
- Move one entry to the right

Current		Next		
State	Symbol read	State	Symbol to write	Direction
b	$\boldsymbol{B}$	c	0	**R**
c	$\boldsymbol{B}$	e	$\boldsymbol{B}$	**R**
e	$\boldsymbol{B}$	f	1	**R**
f	$\boldsymbol{B}$	b	$\boldsymbol{B}$	**R**

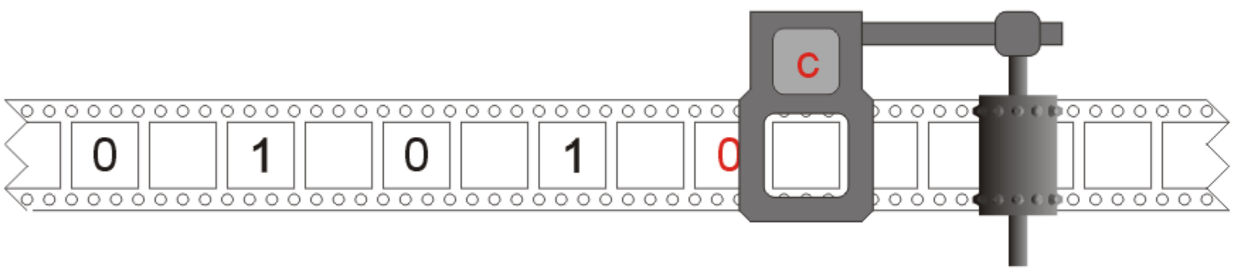


Example - 6

The result is the state c and a blank symbol is under the head:

- Set the state to e
- Write a blank to the tape
- Move one entry to the right

Current		Next		
State	Symbol read	State	Symbol to write	Direction
b	***B***	c	0	**R**
c	***B***	e	***B***	**R**
e	***B***	f	1	**R**
f	***B***	b	***B***	**R**



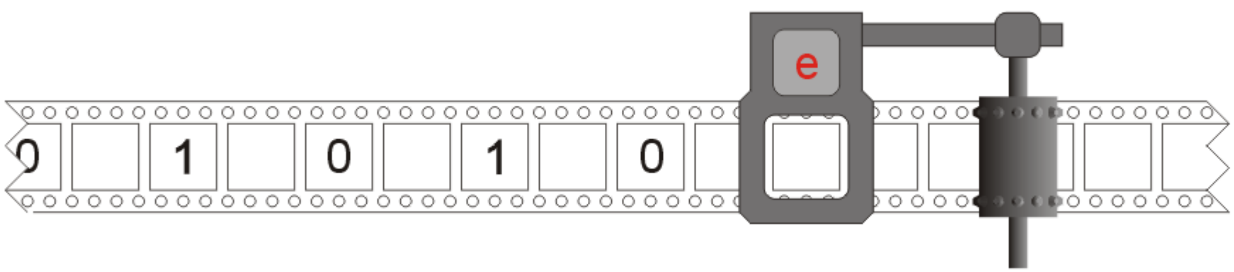
Example - 7

\

The result is the state e and a blank symbol $\mathbf{B}$ under the head

- This is the state we were in four steps ago
- This machine never halts...

Current		Next		
State	Symbol read	State	Symbol to write	Direction
b	$\boldsymbol{B}$	c	0	**R**
c	$\boldsymbol{B}$	e	$\boldsymbol{B}$	**R**
e	$\boldsymbol{B}$	f	1	**R**
f	$\boldsymbol{B}$	b	$\boldsymbol{B}$	**R**



Another Example

\

This Turing machine does what?

- Tape symbols: $\Gamma = \{\boldsymbol{B}, 1\}$
- States: $Q = \{a, b, c, d, e, \boldsymbol{H}\}$
- Initial state: $q_0 = a$
- Halting state: $\boldsymbol{H}$

Current		Next		
State	Symbol read	State	Symbol to write	Direction
a	$\boldsymbol{B}$	a	$\boldsymbol{B}$	**R**
a	1	b	1	**R**
b	$\boldsymbol{B}$	c	1	**R**
b	1	b	1	**R**
c	$\boldsymbol{B}$	d	$\boldsymbol{B}$	**L**
c	1	b	1	**R**
d	$\boldsymbol{B}$	d	$\boldsymbol{B}$	**L**
d	1	e	$\boldsymbol{B}$	**L**
e	$\boldsymbol{B}$	$\boldsymbol{H}$	$\boldsymbol{B}$	**R**
e	1	e	1	**L**

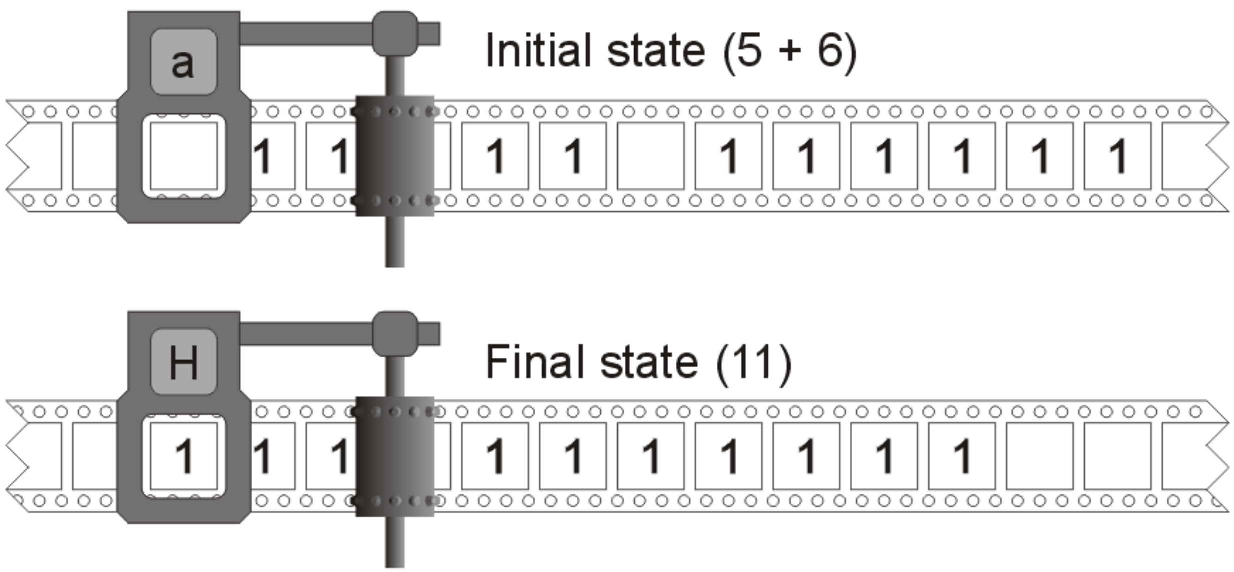
Note there is exactly one entry for each pair in $Q \setminus \{\boldsymbol{H}\} \times \Gamma$

- It may not be necessary to have one for each, but you cannot have more than one transition for a given state and symbol
- **Deterministic** program

Example 2

After 22 steps, a group of five ones and a group of six ones are merged into a single group of eleven ones

- This represents 5 + 6 = 11
- It is the simplest addition machine (no boolean representation of numbers)



Non-deterministic algorithms

A Turing machine is non-deterministic if the transition table can contain more than one entry per state-letter pair

- When more than one transition is possible, a non-deterministic Turing machine branches and creating a new sequence of computation for each possible transition

A non-deterministic algorithm can be implemented on a deterministic machine in one of three manners:

- Assuming execution along any branch ultimately stops, perform a depth-first traversal by choosing one of the two or more options and if one does not find a solution, continue with the next option
- Create a copy of the currently executing process and have each copy work on one of the next possible steps
  - These can be performed either on separate processors or as multiple processes or threads on a single processor
- Randomly choose one of the multiple options

Turing-Church Conjecture

\

Alan Turing and Alonzo Church (Turing’s PhD mentor at Princeton):

- For any algorithm which can be calculated given arbitrary amounts of time and storage, there is an equivalent Turing machine for that algorithm
- Formally: a function on the natural numbers can be calculated by an effective method if and only if it is computable by a Turing machine
  - 'Effective method’: each step of which is precisely predetermined and which is certain to produce the answer in a finite number of steps

A computational system is said to be *Turing complete* if it can compute every function computable on a Turing machine

- e.g., a programming language compiled into machine code and run on a processo

Decision Problem Instance

\

- An algorithm solves a concrete problem in time O(T(n)) if, when provided with a problem instance of size n bits, it produces the correct answer to the question in O(T(n)) time
- **Polynomially solvable** problems: $T(n) = n^k$ for some k > 0
- Size of the input:
  - Integers represented in binary
  - Sets represented in bits related to the number of elements in the set times the number of bits per element

P and NP

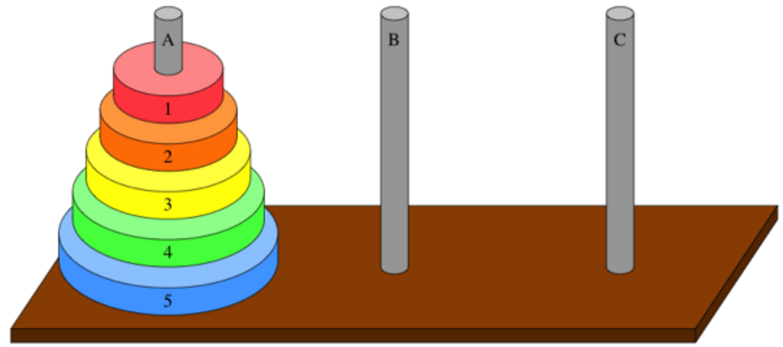
\

- A decision problem belongs to the class **P** if there is a algorithm solving the problem with a running time on a deterministic machine that is polynomial in the input size.
- A decision problem belongs to the class **NP (**non-deterministic polynomial) if:
  - Any solution y leading to ‘yes' can be encoded in polynomial space with respect to the size of the input x.
  - Checking whether a given solution leads to `yes' can be done in polynomial time with respect to the size of x and y
  - problem is solvable in polynomial time in a non-deterministic machine
    - Can explore all possible solutions in parallel
    - Oracle selects best possible solution to check

<u>Slightly more formal:</u>

- Problem Q belongs to the class **NP**, if there exists a polynomial-time 2-argument algorithm A, such that:
  - For each instance i, i is a yes-instance to Q, if and only if, there is a polynomial-size certificate c for which A(i,c) = true.

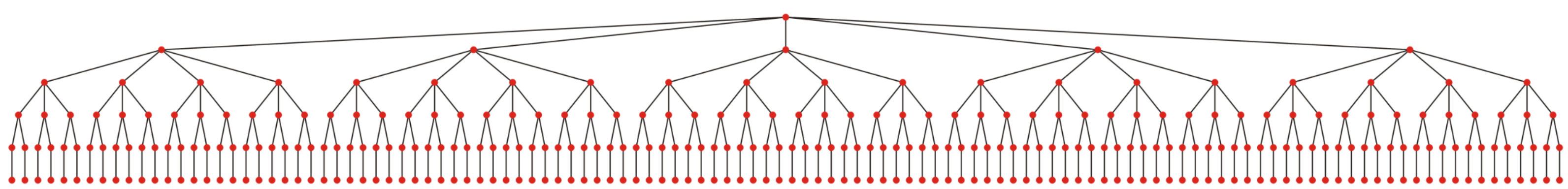
Examples



- Tower of Hanoi
  - Number of moves to solve is exponential in number of disks
  - Can't describe solution in polynomial number of moves
  - Not in NP
- Shortest path problem
  - Is the length of a shortest path from a to b less than a threshold?
  - Can answer in polynomial time in the size of the description of the network —> It is in **P**
- Traveling salesperson problem
  - Is the length of a complete traveling salesperson tour less than a threshold?
  - A tour can be described in polynomial space, and we can verify the length of the tour in polynomial time —> it is in **NP**

Non-deterministic polynomial-time algorithms

- The traveling salesman problem can solved non-deterministically:
  - At each step, spawn a thread for each possible path
  - As you finish, compare them and determine if any of them have length less than k
  - The run time is now $\Theta(|V|)$
  - This is a brute-force search



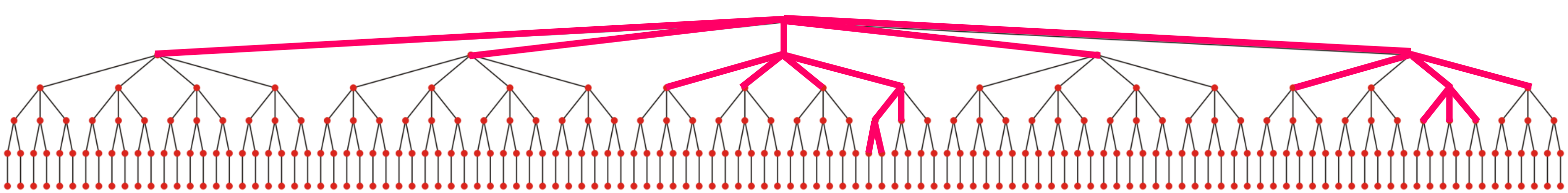
Non-deterministic polynomial-time algorithms

Consider the following decision problem:

"Is there a path between vertices a and b with weight no greater than K?"

Dijkstra's algorithm can answer this in polynomial time

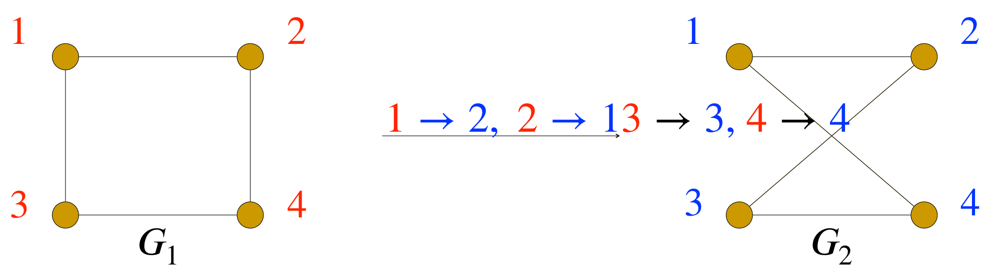
- Dijkstra's algorithm also solves the optimization problem



Examples of $\mathbf{NP}$ problems

- Factoring: factor a given number n.
- Decision version: Given (n, k), decide whether n has a factor less than k
- Factoring is in **NP**: For any candidate factor $m \le k$, it's easy to check whether $m \mid n$.

- Graph Isomorphism: Given two graphs G_1 and G_2, decide whether we can permute vertices of G_1 to get G_2.



- Easy to check: For any given permutation, easy to permute G_1 according to it and then compare to G_2.

Reduction and completeness

- Decision problem for language A is reducible to that for language B in time t if $\exists f : Domain(A) \to Domain(B)$ s.t. $\forall$ input instance x for A,
  1. $x \in A \Leftrightarrow f(x) \in B$, and
  2. one can compute $f(x)$ in time $t(|x|)$
- Thus to solve A, it is enough to solve B.
  - First compute $f(x)$
  - Run algorithm for B on $f(x)$.
  - If the algorithm outputs $f(x) \in B$, then output $x \in A$.

Reduction

- Reduction converts the solution of a problem to the solution of another problem

- Graphically, we may think of the following image:

$$\begin{tikzcd} \text{Problem A} \arrow[r] \arrow[d, "\text{reduction } \mathbf{O}(n^{k_1})"'] & \text{Solution to A} \\ \text{Problem B} \arrow[r] & \text{Solution to B} \arrow[u, "\text{reversion } \mathbf{O}(n^{k_2})"'] \end{tikzcd}$$

- To solve Problem A, we:
  - Reduce the problem to Problem B in polynomial time
  - Solve Problem B
  - Revert the solution back into a solution for Problem A

- We want the reduction and reversion algorithms to be of polynomial complexity: **polynomial reduction**

Example: Polynomial reduction

- Multiply two n digit decimal numbers:
  - Reduction: convert the two numbers into binary numbers
  - Multiply the two binary numbers
  - Reversion: convert the solution back into a decimal number

- Both the reduction and the reversion run in $\Theta(n)$ time

- Observe: if a decision problem is reduced to a decision problem, the corresponding reversion algorithm is trivial and is in $\Theta(1)$ time

Polynomial reduction

- Another example: Does a list have a duplicate element?
  - Reduction: Sort the list
  - Simpler problem: Does a sorted list have a duplicate element?
  - Reversion: Return `true` or `false`, as is

- Both the reduction and the reversion run in $\Theta(n)$ time
  - If a decision problem is reduced to a decision problem, the reversion is therefore $\Theta(1)$
    - Either the solution or its negation

Examples: Polynomial reduction

\

- Example: Does an n by n assignment problem have minimum cost less than K?
  - Polynomial Reduction: Reduce to the solution of n sequential shortest path problems in non-negative weight graphs with O(2n) vertices
  - Reversion: Convert the decision of the successive shortest path algorithm

- Example: Given two sequences $a_1, a_2, \ldots, a_n$ and $b_1, b_2, \ldots, b_n$, is there a permutation j(i) such that $\sum_{i=1}^{n} a_i b_{j(i)} \geq K$?
  - Polynomial Reduction: sort both sequences increasing order, multiply the sorted sequences and verify product is greater than or equal to K
  - Another polynomial reduction: convert to assignment problem!

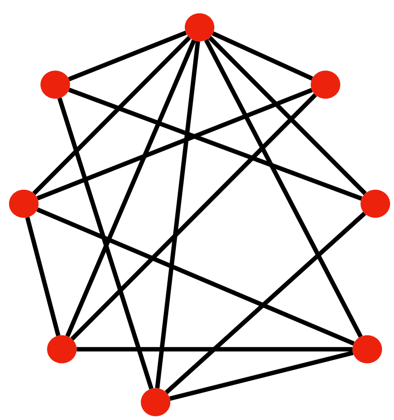
Polynomially Reducible

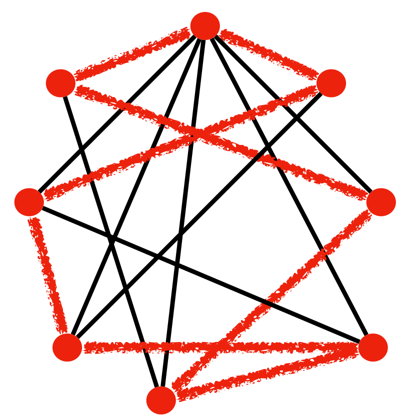
- Definition: Problem A is polynomially reducible to problem B if there exists an algorithm for solving problem A in polynomial time if we could solve arbitrary instances of problem B at unit cost
  - Written as $A \leq_P B$
  - If $A \leq_P B$, and $B \leq_P A$, then we write $A =_P B$

- If $A \in \mathbf{P}$ and $B \leq_P A$, then $B \in \mathbf{P}$
  - e.g. Shortest path problem is in **P**. Assignment problem is polynomially reducible to shortest path problem. Then Assignment problem is in **P**.

Polynomial Reduction

- Problem A: Traveling salesperson problem
  - Given a weighted directed graph, find a simple cycle that visits each vertex once and has total cost less than or equal to K
- Problem B: Hamiltonian cycle problem
  - Given a directed graph, does there exist a simple cycle that includes every vertex once?





Polynomial Reduction

- Claim: $B \leq_P A$

Proof: Let graph for B be G(V,E). We are going to construct a weighted graph G(V,E') with a weight function for Problem A, in polynomial time

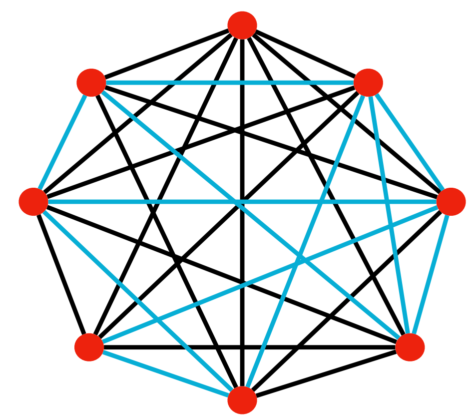
Let E' be the dense set of edge in $V \times V$. Assign weights as follows: w(e') = 0 if $e' \in E$, w(e') = 1 if $e' \notin E$

Reduction: $O(|V|^2)$ is polynomial

TSP problem: Does there exist a simple cycle that visits every vertex in V once, and has total cost less than or equal to 0

If yes, it is a cycle with all edges in E, and so it is a Hamiltonian cycle

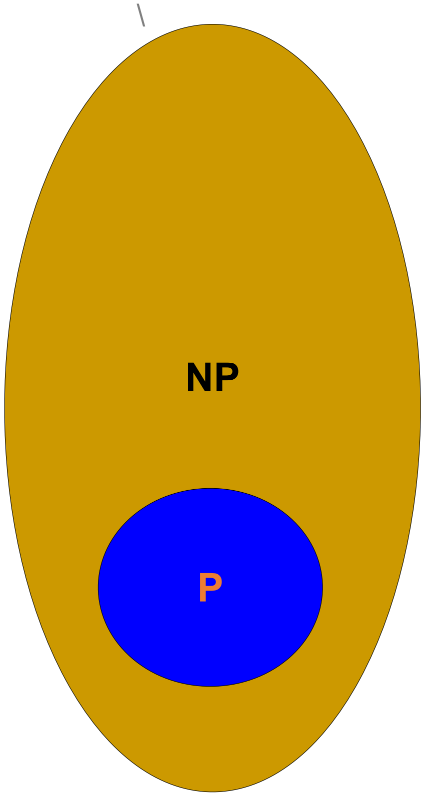
If no, no cycle exists with all edges in E, so no Hamiltonian cycle can be found



This shows Hamiltonian cycle $\leq_P$ Traveling Salesperson Problem

NP Complete

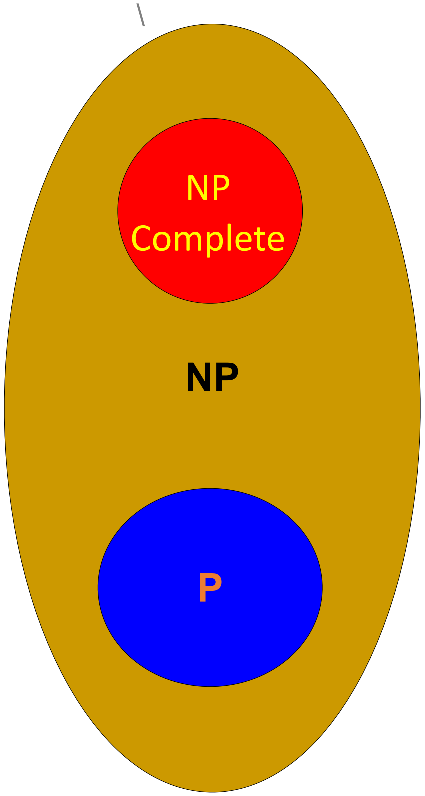
- We know class **P** is a subset of class **NP**
  - TSP is not known to be in class **P** but is in **NP**
  - It is a conjecture that **P** $\neq$ **NP**, but it has not been proven
  - We do know $\mathbf{P} \subset \mathbf{NP}$



- Definition: a problem A is NP-complete, if $A \in \mathbf{NP}$ and every problem $B \in \mathbf{NP}$ can be polynomially reduced to A. That is, $B \leq_P A$

- Do such problems exist? If so, **and** one of them was in class **P**, then every problem in class **NP** could be solved in polynomial time

Existence of NP-complete Problems



- Steven Cook and Leonid Levin (BU) proved in parallel that there exists an **NP** complete problem
  - Levin joined BU in 1980s
- Specifically: Boolean satisfiability (SAT)
  - Given a Boolean formula, is there an assignment of True and False to its variables so that the formula evaluates to True?
  - e.g. Can we find values of p, q, r, s so formula below is true?

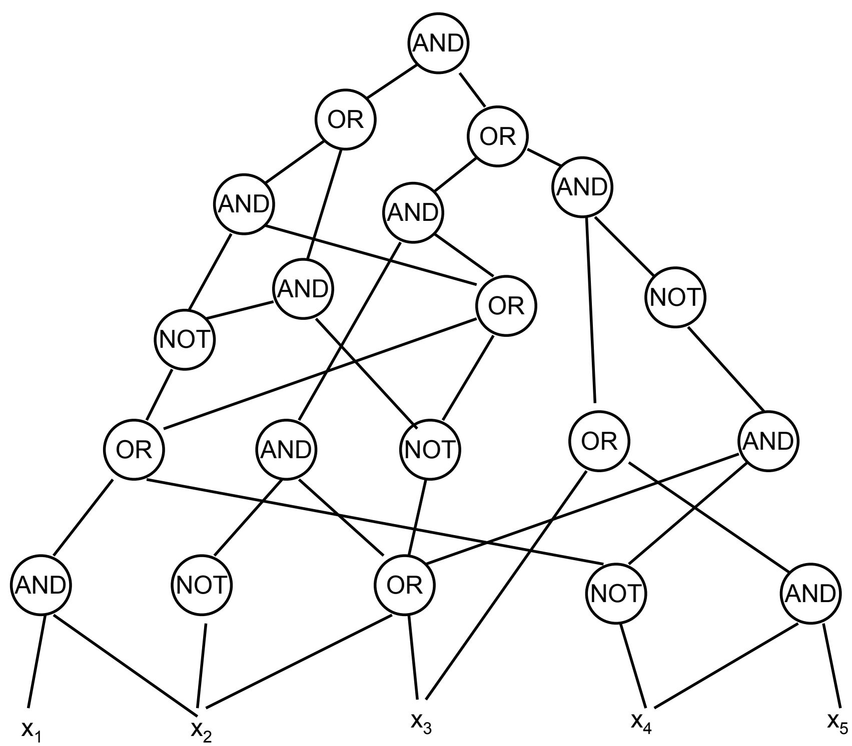
$$(p \vee q \vee r) \wedge (p \vee \neg q \wedge s) \wedge (\neg p) \wedge (\neg q \wedge s) \wedge (\neg s)$$

- Cook-Levin Theorem
  - If a polynomial-time deterministic algorithm can solve this problem, then polynomial-time deterministic algorithms can solve all NP problems

Equivalent Problem: Circuit SAT

\

- Find inputs to the circuit so the output is true
  - All the gates are ands, nots, ors



SAT and k-SAT

\

- SAT formula: AND of m clauses (Conjunctive Normal Form)
- n variables (taking values 0 and 1)
- a literal: a variable x_i or its negation $\bar{x}_i$
- m clauses, each being OR of some literals.
  - $(x_1 \vee x_2 \vee \bar{x}_3) \wedge (x_1 \vee \bar{x}_2 \vee x_4) \wedge (\bar{x}_4) \wedge (\bar{x}_2 \vee x_3) \wedge (\bar{x}_3)$
- SAT Problem: Is there an assignment of variables s.t. the formula evaluates to 1?
- k-SAT: same as SAT but each clause has at most k literals.
- SAT and k-SAT are in **NP**
  - Given any assignment, it's easy to check whether it satisfies all clauses.

The 1st **NP**-complete problem: SAT

\

- Consider an arbitrary problem class Y in **NP**
- Given a problem instance y, with description length n bits, there is an algorithm which is polynomial in n, p(n), which can verify whether the answer to this instance y is true
  - The input description to this verification algorithm is a set of bits: Boolean variables
  - Verification algorithm manipulates Boolean variables into a yes or no decision: a Boolean output
- The algorithm can be specified as a Boolean formula in conjunctive normal form, involving p(n) terms
- Finding a set of inputs to this formula that makes it true is equivalent to determining whether there original problem instance has answer true
  - Y $\leq_P$ SAT

How Do We Find Other NP-Complete Problems

- Reduction! We want to find if problem Y in **NP** is NP-complete
- If SAT $\leq_P$ Y, then Problem Y is NP-Complete! SAT is no easier than Y
- Note: to prove a problem A is in class **P**, we need to find a problem B in class **P** so that A $\leq_P$ B
- To prove a problem A is NP-complete class **P**, find an NP-complete problem so that B $\leq_P$ A
- Note the following: it is easy to show that SAT $=_P$ 3-SAT using standard logic transformations

Graph NP-Complete Problems

\

- Karp ('72) followed Cook's proof to show 3-D SAT can be reduced to 21 different graph problems
  - TSP, others
- Many others have continued to extend this to thousands of other problems
- Note: to prove a problem A is in class **P**, we need to find a problem B in class **P** so that A $\leq_P$ B
- To prove a problem A is NP-complete class **P**, find an NP-complete problem so that B $\leq_P$ A
- Note the following: it is easy to show that SAT $=_P$ 3-SAT using standard logic transformations

NP-complete problem 1: Clique

- Clique: Given a graph G and a number k, decide whether G has a clique of size $\geq k$.
  - Clique: a complete subgraph.
- Fact: Clique is in **NP**.
- Theorem: If one can solve Clique in polynomial time, then one can also solve 3-SAT in polynomial time.
  - So Clique is at least as hard as 3-SAT.
- Corollary: Clique is **NP**-complete.

Approach: Reduction

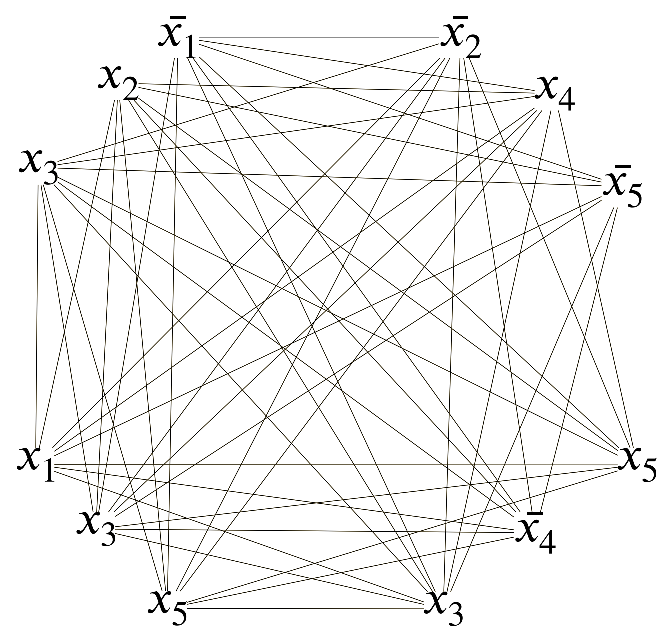
\

- Given a 3-SAT formula $\varphi = C_1 \wedge \ldots \wedge C_k$, with conjunctions C_k, we construct a graph G s.t.
  - if ϕ is satisfiable, then G has a clique of size k.
  - if ϕ is unsatisfiable, then G has no clique of size $\geq k$.
  - Note: k is the number of clauses of ϕ.

- If you can solve the Clique problem, then you can also solve the 3-SAT problem.

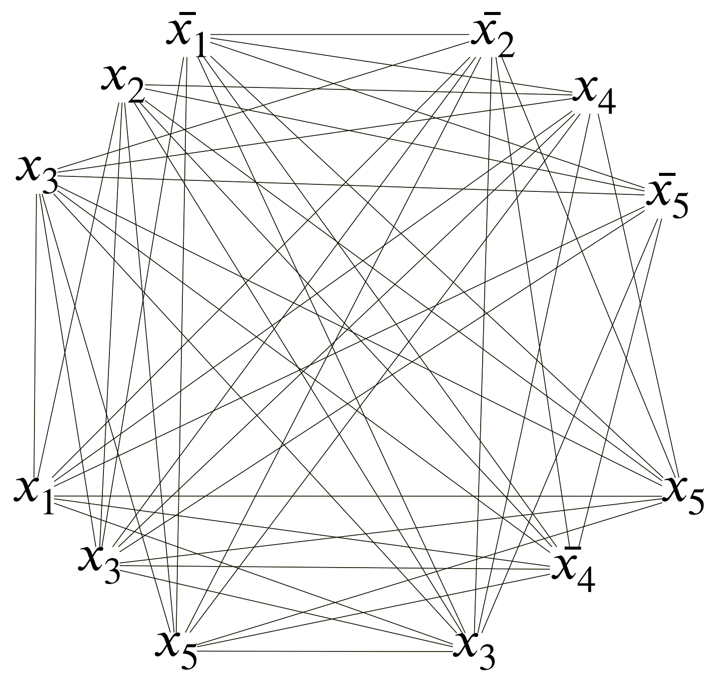
Construction

- Put each literal appearing in the formula as a vertex.
  - Literal: x_i and $\bar{x}_i$
  - e.g.
    $\phi = (\bar{x}_1 \vee x_2 \vee x_3) \wedge (\bar{x}_2 \vee x_3 \vee x_5) \wedge (x_1 \vee x_3 \vee x_5) \wedge (x_3 \vee \bar{x}_4 \vee x_5)$
- Literals from the same clause are not connected
- Two literals from different clauses are connected if they are not the negation of each other



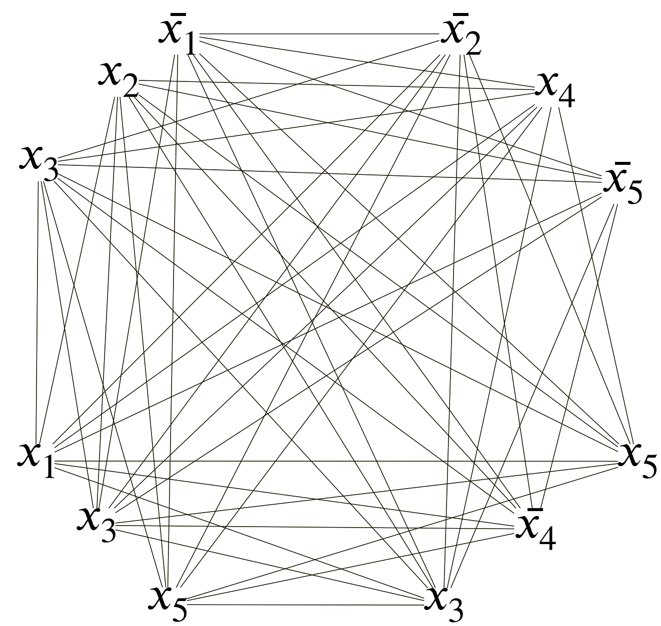
ϕ is satisfied $\Rightarrow G$ has a k-clique

- If ϕ is satisfied,
- then there is a satisfying assignment $x_1 \ldots x_n$ s.t. each clause has at least one literal being 1.
  - E.g. $x = 00111$, then pick $\bar{x}_1, x_4, x_3, x_5$
- And those literals (one from each clause) are consistent because they all evaluate to 1
- So the subgraph with these vertices is complete. --- A clique of size k.



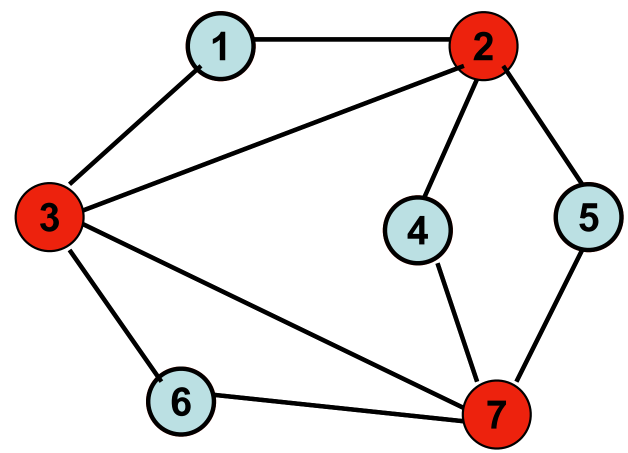
G has a k-clique ⇒ ϕ is satisfied

- If the graph has a clique of size k:
- It must be one vertex from each clause.
  - Vertices from the same clause don't connect.
- And these literals are consistent.
  - Otherwise they don't all connect.
- So we can pick the assignment by these vertices. It satisfies all clauses by satisfying at least one vertex in each clause
- Hence, it makes ϕ true



NP-complete problem 2: Vertex Cover

- Vertex Cover: Given a graph G and a number k, decide whether G has a vertex cover of size $\leq k$
  - V' is a vertex cover if all edges in G are "touched" by vertices from V'



- Vertex Cover is in **NP**
  - Given a candidate subset $S \subseteq V$, it is easy to check whether "$|S| \leq k$ and S touches all edges in E"

NP-complete

\

- To prove Vertex Cover is **NP**-complete, reduce Clique to Vertex Cover
- For any graph G, the complement of G is $\bar{G}$
  - If $G = (V, E)$, then $\bar{G} = (V, \bar{E})$
  - $\bar{E} = \{\{i, j\} : i \in V, j \in V, \{i, j\} \notin E\}$
  - Constructing $\bar{G}$ from G is a polynomial operation

- Theorem: G has a k-clique $\iff$ $\bar{G}$ has a vertex cover of size $n - k$
  - Will show In next slide

- Given this theorem, Clique can be reduced to Vertex Cover
- So Vertex Cover is **NP**-complete

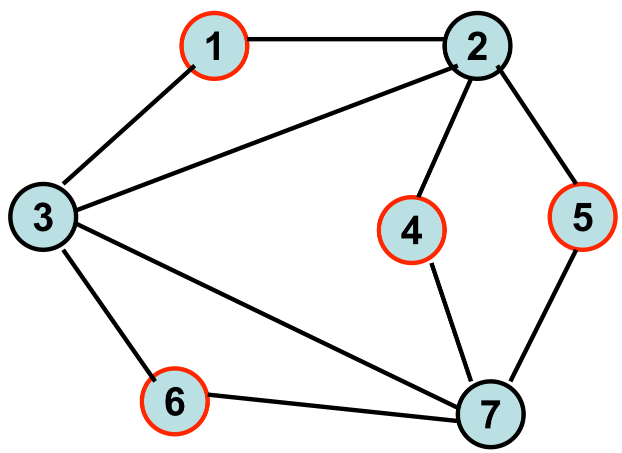
Proof of the theorem

\

- G has a k-clique $\iff$ $\exists V' \subset V, |V'| = k, V'$ is a clique in G
- Independent set in $\bar{G}$: For any u, v in V', $\{u, v\} \notin \bar{E}$
- V' is a clique in G $\iff$ V' is an independent set in $\bar{G}$
- V' is an independent set in $\bar{G}$ $\iff$ V/V' is a vertex cover of $\bar{G}$, because every edge in $\bar{E}$ must touch a vertex in V/V'
- Let $V'' = V/V'$. Then, $|V''| = n - k$, and V'' is a vertex cover of $\bar{G}$

Another NP-Complete Problem: Independent Set

- Independent Set: Decide whether a given graph has an independent set of size at least k
- The above argument shows that the Independent Set problem is also NP-Complete because Clique $\leq_P$ Independent set



NP-complete problem 3: Integer Programming (IP)

- Any 3-SAT formula can be expressed by integer programming.
- Consider a conjunctive clause, for example,
  - $\bar{x}_1 \vee x_2 \vee x_3, \quad x_1, x_2, x_3 \in \{0,1\}$
  - This is equivalent to $(1 - x_1) + x_2 + x_3 \geq 1, \quad x_1, x_2, x_3 \in \{0,1\}$

- Multiple conjunctive clauses:
  - Translate to multiple integer programming constraints which must be satisfied simultaneously
  - Finding a feasible solution to the set of integer programming inequalities would yield a feasible solution to 3-SAT

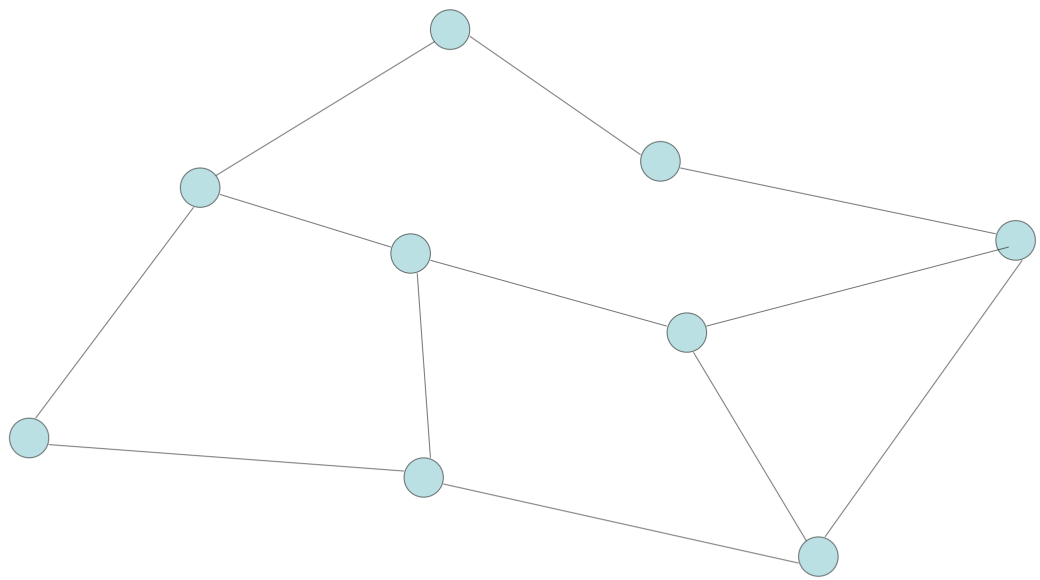
Integer Programming example

\

- 3-SAT formula
  $(\bar{x}_1 \vee x_2 \vee x_3) \wedge (\bar{x}_2 \vee x_4 \vee \bar{x}_5) \wedge (x_1 \vee x_3 \vee x_5) \wedge (x_3 \vee \bar{x}_4 \vee x_5)$
- Integer program inequalities
- $(1 - x_1) + x_2 + x_3 \geq 1,$
  $(1 - x_2) + x_4 + (1 - x_5) \geq 1,$
  $x_1 + x_3 + x_5 \geq 1,$
  $x_3 + (1 - x_4) + x_5 \geq 1,$
  $x_1, x_2, x_3, x_4, x_5 \in \{0,1\}$
- So if one can solve IP efficiently, then one can also solve 3-SAT efficiently
- 3-SAT $\leq_P$ Integer Programming

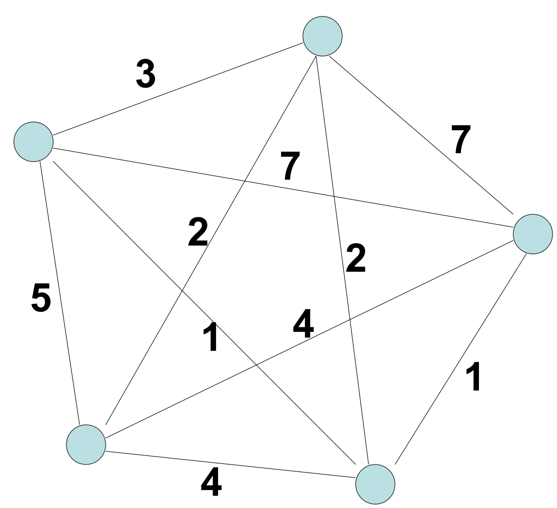
Hamiltonian Circuit Problem

- Hamiltonian Circuit – does there exist a simple cycle including all the vertices of the graph?
  - Can show Vertex Cover $\leq_P$ Hamiltonian Circuit (very complicated construction)



Traveling Salesman Problem

- Given a complete graph with edge weights, determine the shortest tour that includes all of the vertices (visit each vertex exactly once, and get back to the starting point)
  - Already proved Hamiltonian Circuit $\leq_P$ TSP



Find the minimum cost tour

Subset Sums

The subset sum problem

- Given n natural numbers $w_1, \ldots, w_n$ and an integer W, is there a subset that adds up to exactly W?

Ex. $\{ 215, 215, 275, 275, 355, 355, 420, 420, 580, 580, 655, 655 \}, \quad W = 1505$

Yes. $215 + 355 + 355 + 580 = 1505$.

- Subset sum is in NP
  - Easy to verify whether a subset sums up to W
- Important aspect: numbers are input in binary
  - Reduction must be polynomial in number of bits of input
  - Want to show 3-SAT $\leq_P$ Subset Sum

Subset Sum via Dynamic Programming

\

- Break up into smaller problems, build up to full problem
- F(k,s) = True if there exists a subset of $w_1, \ldots, w_k$ that adds up to s, else false
- Initialization: F(k,0) = True, k = 0, …, n; F(0, s) = False, s > 0
- Dynamic Programming recursion:

$$F(k,s) = \begin{cases} F(k-1,s) & \text{if } w_k > s \\ F(k-1,s) \vee F(k-1,s-w_k) & \text{if } w_k \le s \end{cases}$$

- Can fill table to get solution: $O(n \times W)$
- But, this is not polynomial: W is exponential in the size of the binary description of W in bits
  - Requires exponential space and time

SAT Reduction to Subset Sum

\

- Given 3-SAT instance with n variables and k clauses, form 2n + 2k decimal integers, each having n + k digits as follows
  - For each variable x_i, construct numbers t_i and f_i of n + m digits: the i-th digit of t_i and f_i is equal to 1, other digits from 1 to n are 0
  - For n+1≤j≤n+m, the j-th digit of t_i is equal to 1 if x_i is in clause c_{j-n}

    For n+1≤j≤n+m,t he j-th digit of f_i is equal to 1 if $\bar{x}_i$ is in clause c_{j-n}
  - All other digits of t_i and f_i are 0
  - Example: $(x_1 \vee x_2 \vee x_3) \wedge (\bar{x}_1 \vee \bar{x}_2 \vee x_3) \wedge (\bar{x}_1 \vee x_2 \vee \bar{x}_3) \wedge (x_1 \vee \bar{x}_2 \vee x_3)$

	i			j			
Number	1	2	3	1	2	3	4
t_1	1	0	0	1	0	0	1
f_1	1	0	0	0	1	1	0
t_2	0	1	0	1	0	1	0
f_2	0	1	0	0	1	0	1
t_3	0	0	1	1	1	0	1
f_3	0	0	1	0	0	1	0

SAT Reduction to Subset Sum - 2

\

- Given 3-SAT instance with n variables and k clauses, form 2n + 2k decimal integers, each having n + k digits as follows
  - For each clause c_j, construct numbers z_j and y_j of n + m digits: The (n+j)-th digit of z_j and y_j is equal to 1, all other digits of z_j and y_j are 0
  - Example: $(x_1 \vee x_2 \vee x_3) \wedge (\bar{x}_1 \vee \bar{x}_2 \vee x_3) \wedge$
    $(\bar{x}_1 \vee x_2 \vee \bar{x}_3) \wedge (x_1 \vee \bar{x}_2 \vee x_3)$
  - Finally, construct a sum number s of n + m digits:
    For 1 ≤ j ≤ n, the j-th digit of s is equal to 1
    For n + 1 ≤ j ≤ n + m, the j-th digit of s is equal to 3
  - e.g. 1113333
  -

	i			j			
Number	1	2	3	1	2	3	4
t_1	1	0	0	1	0	0	1
f_1	1	0	0	0	1	1	0
t_2	0	1	0	1	0	1	0
f_2	0	1	0	0	1	0	1
t_3	0	0	1	1	1	0	1
f_3	0	0	1	0	0	1	0
z_1	0	0	0	1	0	0	0
y_1	0	0	0	1	0	0	0
z_2	0	0	0	0	1	0	0
y_2	0	0	0	0	1	0	0
z_3	0	0	0	0	0	1	0
y_3	0	0	0	0	0	1	0
z_4	0	0	0	0	0	0	1
y_4	0	0	0	0	0	0	1
sum	1	1	1	3	3	3	3

SAT Reduction to Subset Sum - 3

\

- If we find subset of rows that sum to the total sum, then we can convert the rows to Boolean variables $x_i, i = 1, \ldots, n$ such that every clause $c_j, j = 1, \ldots, m$ is true
  - x_i true if t_i is in subset; otherwise, f_i must be in subset so x_i is false
  - Since the j columns sum to 3, at least one of the literals in c_j must be true because there are at most 2 in z_j, y_j
- If there exists a true assignment, then there is a set of rows that sums to sum
  - For clause c_j, add rows y_j if two or less literals are true, add z_j if only one literal is true
- Result: 3-SAT $\leq_P$ Subset Sum

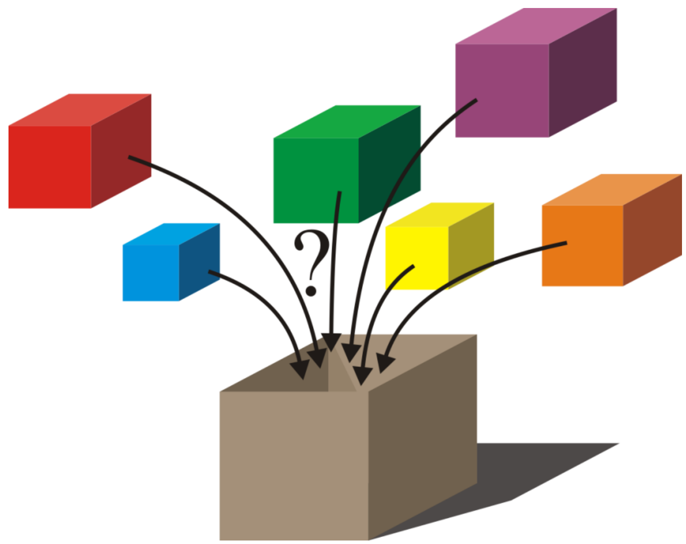
	i			j			
Number	1	2	3	1	2	3	4
t_1	1	0	0	1	0	0	1
f_1	1	0	0	0	1	1	0
t_2	0	1	0	1	0	1	0
f_2	0	1	0	0	1	0	1
t_3	0	0	1	1	1	0	1
f_3	0	0	1	0	0	1	0
z_1	0	0	0	1	0	0	0
y_1	0	0	0	1	0	0	0
z_2	0	0	0	0	1	0	0
y_2	0	0	0	0	1	0	0
z_3	0	0	0	0	0	1	0
y_3	0	0	0	0	0	1	0
z_4	0	0	0	0	0	0	1
y_4	0	0	0	0	0	0	1
sum	1	1	1	3	3	3	3

Knapsack problem

\

The knapsack problem

- Suppose a container can hold a maximum weight W
- Given a number of items with specified weights w_i and values v_i, is there some combination of items that has a total value greater than or equal to V without exceeding the maximum weight W?
-



Knapsack Decision Problem is NP-Complete \

- Show that Subset Sum $\leq_P$ Knapsack
- Given subset sum problem instance with $w_1, \ldots, w_n$ and total weight W, form following Knapsack problem:
  - Let weight of object i be w_i
  - Let value v_i of object i be $v_i = w_i$
- Solve the following knapsack decision problem: Is there a subset of items S such that the value is greater than or equal to W, while the total weight is less than or equal to W?
  - $\sum_{i \in S} v_i = \sum_{i \in S} w_i \geq W; \quad \sum_{i \in S} w_i \leq W;$
  - If answer is yes, then S is a subset with sum W. If no solution can be found for knapsack, then subset sum does not have a subset with sum W

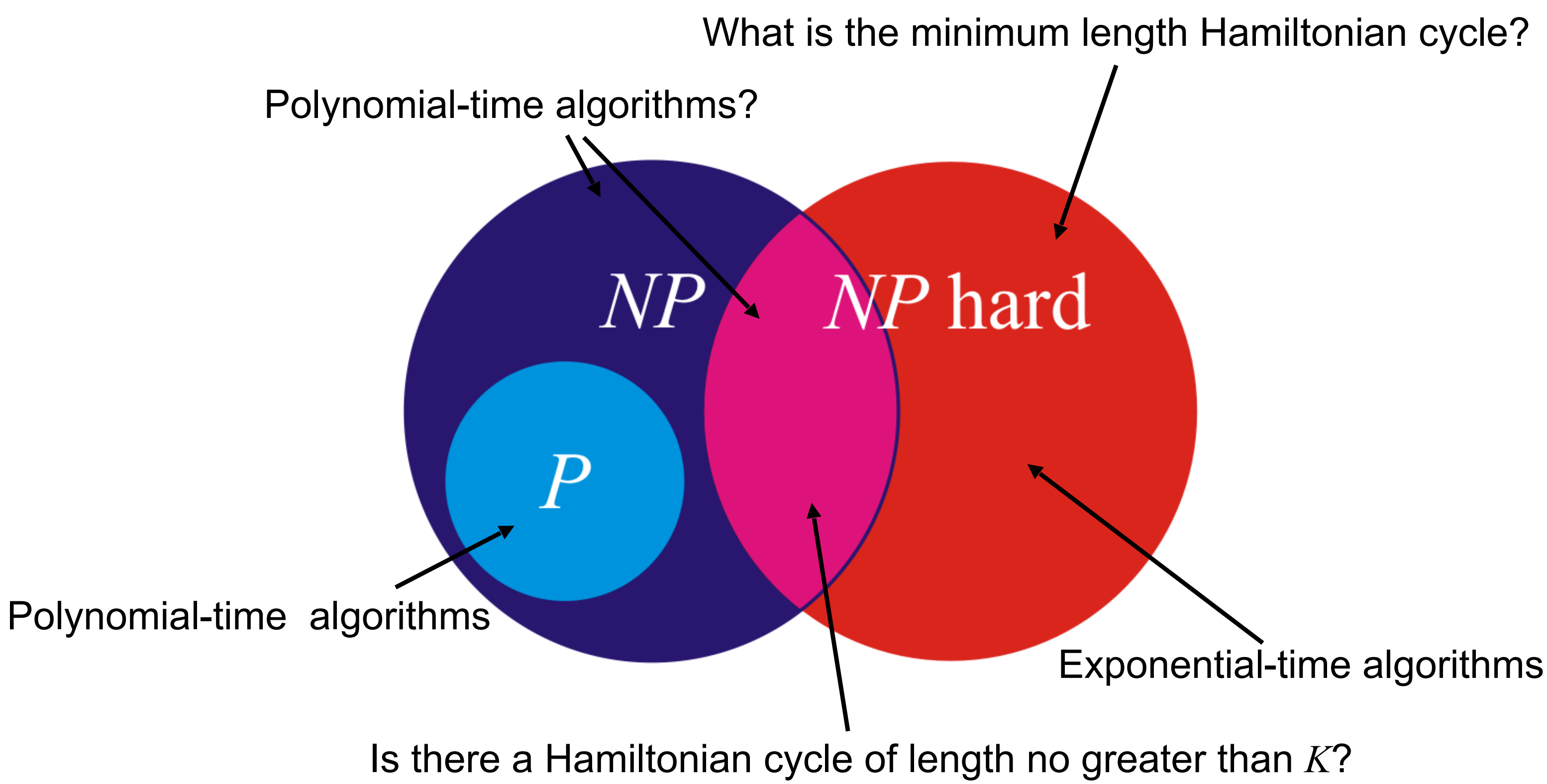
NP-Hard vs NP-Complete Problems

\

- The optimization version of NP-Complete problem is referred to as an **NP-Hard** problem
  - NP-complete problems are decision problems
  - E.g. finding the minimum length Hamiltonian circuit is NP-hard; finding whether there exists a Hamiltonian circuit with length less than L is NP-complete
  - Finding the maximum value that fits in a knapsack is NP-hard; finding whether we can fit value of at least V is NP-Complete
- Formally, a problem X is NP-hard, if there is an NP-complete problem Y, such that Y is reducible to X in polynomial time
  - But X does not have to be in NP, and does not have to be a decision problem

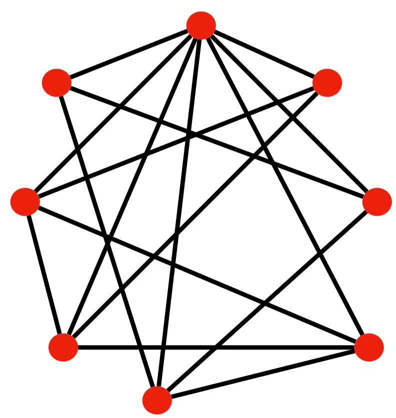
NP and NP Hard

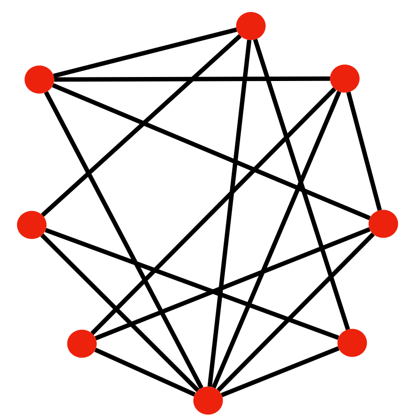
Consider this Venn diagram where $P \subseteq NP$



Problems in NP not in P or NP-Complete

- Graph Isomorphism
  - We don't have an algorithm to show problem is in **P**, and no reduction of an np-complete problem to it





Other Complexity Concepts

\

- **Pseudopolynomial** complexity: If K is the size of the largest number, and n is the size of the input, then the worst case complexity is polynomial in n and K. (e.g. Integer Knapsack)

- **Strongly Polynomial** complexity: worst case complexity is polynomial in input size, independent of largest value of number in input.

- **Strongly NP-complete** problems: If one restricts the size of the largest number in the problem to K, where K is a polynomial in the input size n, then the problem is still NP-complete.
  - e.g. Clique
  - Hard to find approximate solutions

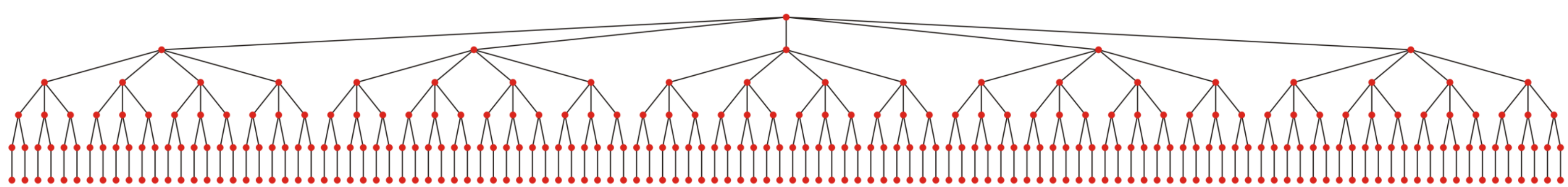
Deterministic polynomial-time algorithms

Consider the traveling salesman problem:

In a complete weighed graph, what is the least weight Hamiltonian cycle?

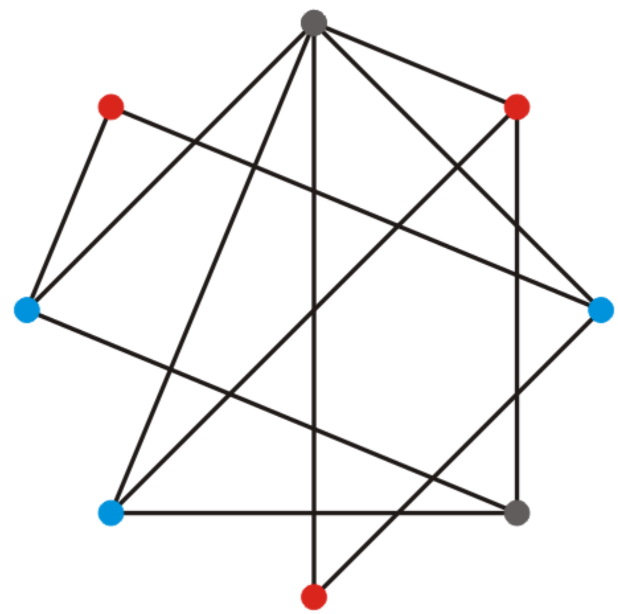
A deterministic algorithm to find a solution is to do tree traversal

- This tries every path
- The run time is $\Theta(|V|!)$
- The Held-Karp algorithm runs in $\Theta(|V|^2\ 2^{|V|})$ time
  - It uses dynamic programming
- To the best of our knowledge, this problem is not in P



NP-Hard: Graph colorings

- Given a graph and n colors, is it possible to assign one of the colors to each of the vertices so that no adjacent vertices have the same color?



  - Complete graphs requires $|V|$ colors
  - Finding the smallest number of colors for a graph is NP Hard

NP-Complete: Graph isomorphisms

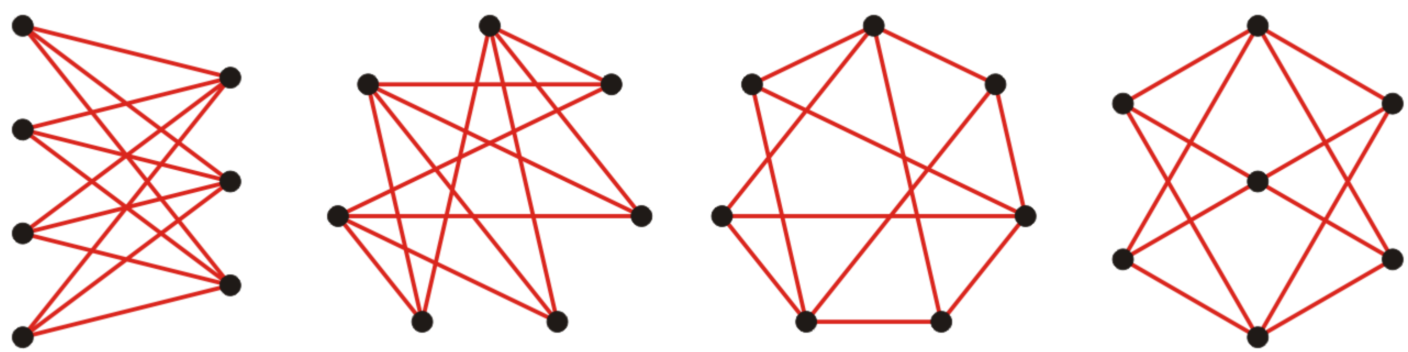
\

Recall the sub-graph isomorphism problem

- Given two graphs, is one graph isomorphic to a sub-graph of the other?

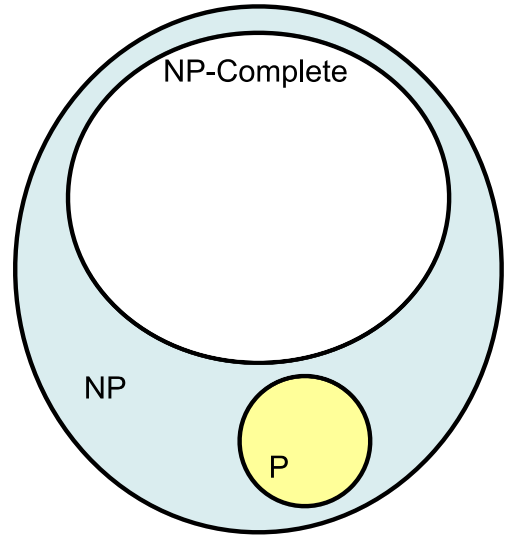
A possibly easier question is:

- Are two graphs isomorphic?



- So far, the best algorithm is $2^{O(\sqrt{n \ln(n)})}$

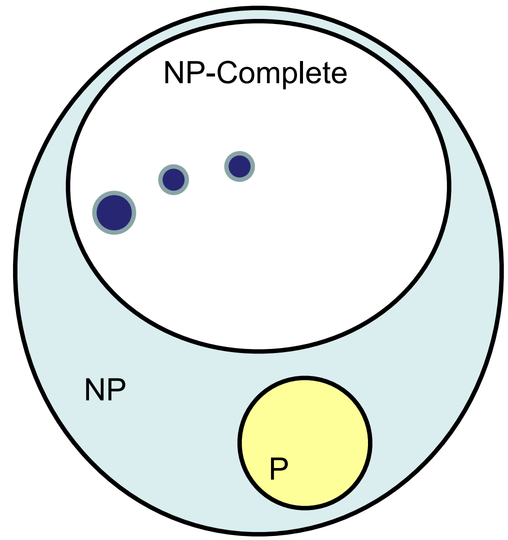
Populating the NP-Completeness Universe



- Circuit Sat $<_P$ 3-SAT
- 3-SAT $<_P$ Independent Set
- 3-SAT $<_P$ Vertex Cover
- Independent Set $<_P$ Clique
- 3-SAT $<_P$ Hamiltonian Circuit
- Hamiltonian Circuit $<_P$ Traveling Salesman
- 3-SAT $<_P$ Integer Linear Programming
- 3-SAT $<_P$ Graph Coloring
- 3-SAT $<_P$ Subset Sum
- Subset Sum $<_P$ Scheduling with Release times and deadlines

Populating the NP-Completeness Universe

- Circuit Sat $<_P$ 3-SAT
- 3-SAT $<_P$ Independent Set
- 3-SAT $<_P$ Vertex Cover
- Independent Set $<_P$ Clique
- 3-SAT $<_P$ Hamiltonian Circuit
- Hamiltonian Circuit $<_P$ Traveling Salesman
- 3-SAT $<_P$ Integer Linear Programming
- 3-SAT $<_P$ Graph Coloring
- 3-SAT $<_P$ Subset Sum
- Subset Sum $<_P$ Scheduling with Release times and deadlines



Exercise

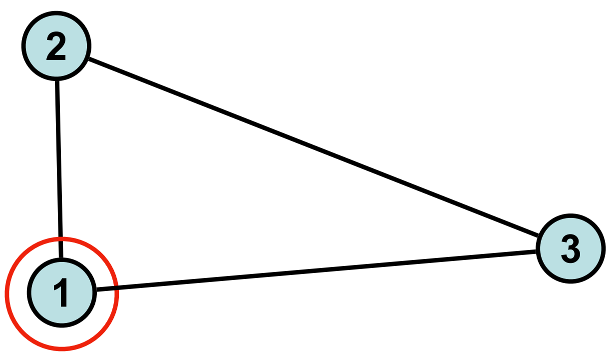
\

- *Dominating Set* problem: Given a graph $G = (V, E)$ and an integer K, decide whether G contains a dominating set of size at most K.
  - Dominating set: $S \subseteq V$ such that $\forall v \in V$, either $v \in S$ or v has a neighbor in S.
  - Namely, S and S's neighbors cover the entire V.
- Prove that Dominating Set is NP-complete.
- Hint: Reduction from Vertex Cover.

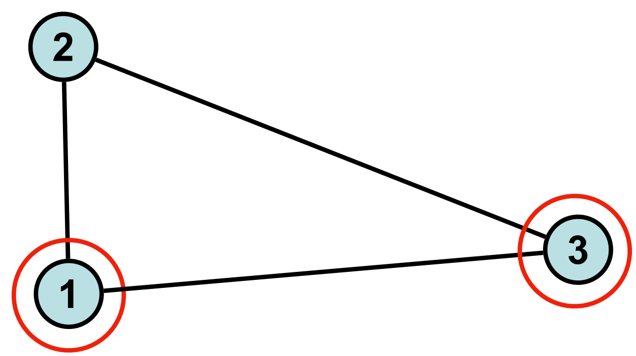
Vertex Cover vs Dominating Set

- Every vertex cover is a dominating set (if graph is connected)
- But, size of minimal vertex cover is different from size of minimal dominating set! Minimal dominating set can be smaller…

Dominating Set



Vertex Cover.



Reduction

- Add an extra vertex for every edge
  - Minimum dominating set = k $\Longleftrightarrow$ minimum vertex cover is $k - n_I$, where n_I is number of isolated vertices