Lower Bound

For each problem, we want to know the lower bound: the best possible algorithm's efficiency for a problem → $\Omega(.)$

Tight lower bound: we have found an algorithm in the this lower-bound efficiency class $\Theta(.)$.

Trivial lower bound: the problem's input/output size

- Too low
- Too high

Information-Theoretic Arguments

This approach seeks to establish a lower bound based on the amount of information it has to produce – an information-theoretic lower bound

Recall the problem of guess the number from 1...*n* by asking 'yes/no' questions

Fundamentally, it is a coding problem. If the input number can be encoded into *m* bits, each 'yes/no' question just resolve one bits and therefore, the lower bound is *m*

We know that $m=\log_2 n$

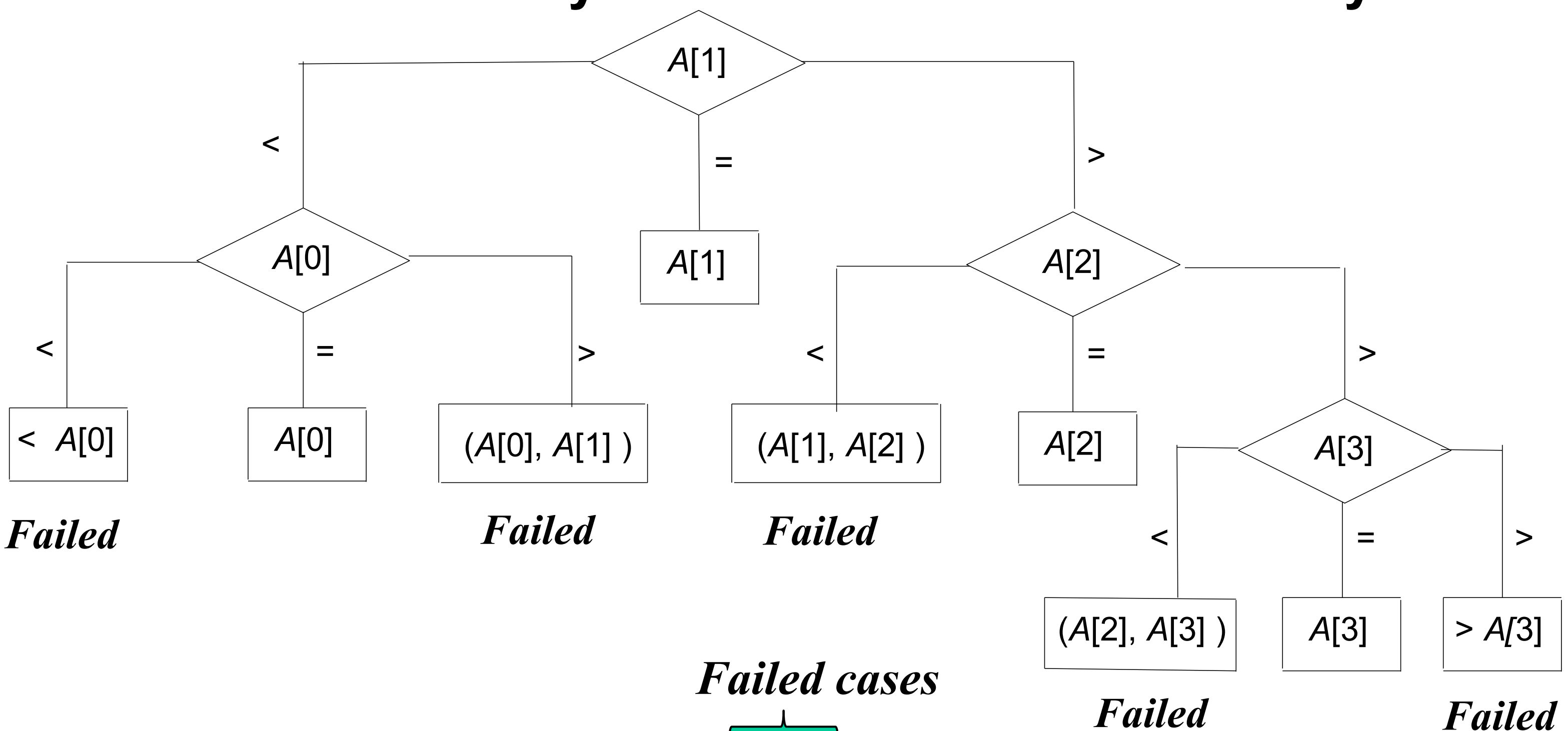
Solution: a decision tree. We will apply the decision tree to find the lower bound for several problems

Complexity for the worst case: the height of this decision tree

Given *L* leaves, the height of the binary tree is at least $\lceil \log_2 L \rceil$

Decision Tree for Searching a Sorted Array

Decision tree for binary search in a four-element array

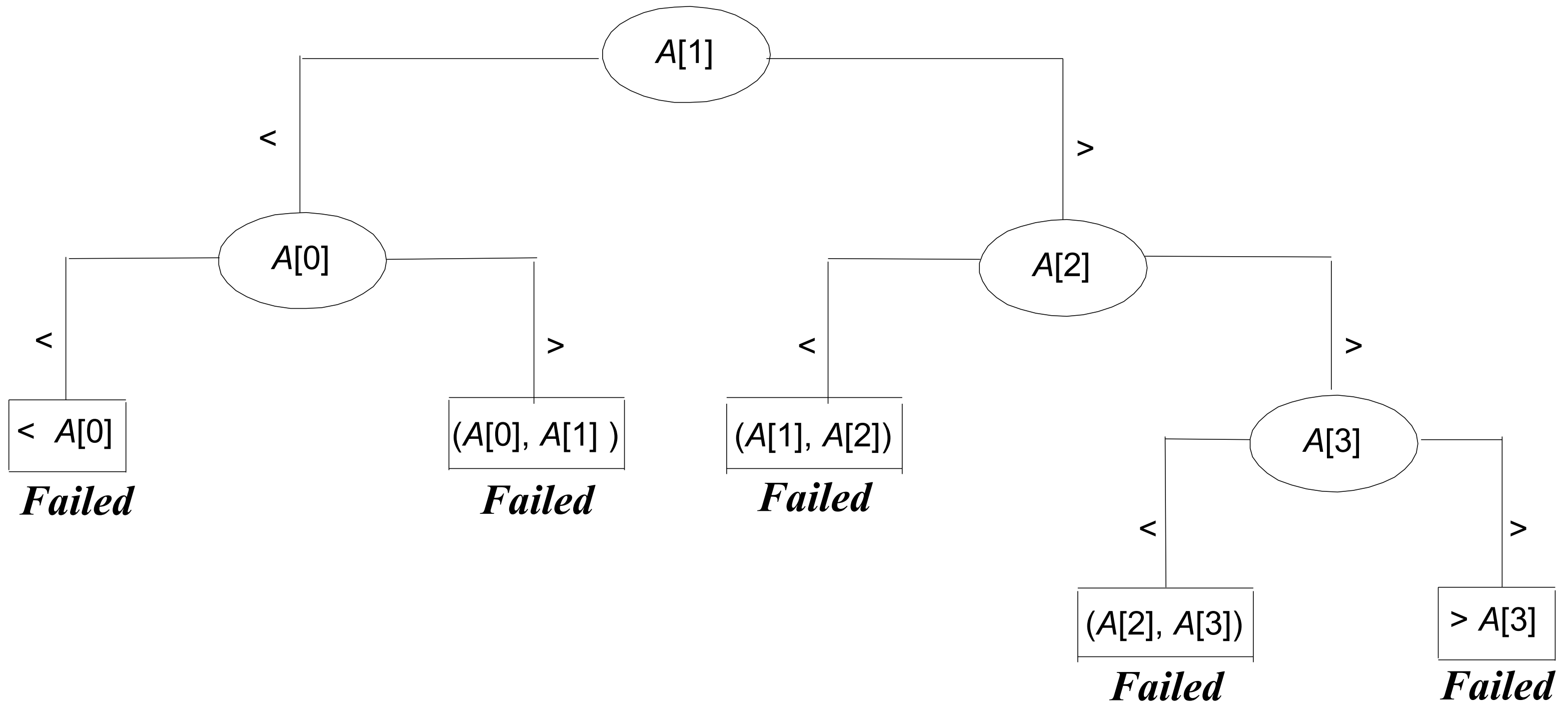


# of leaves for n elements → $n+\overbrace{n+1}^{\textit{Failed cases}}$

→lower bound for the worst case: $\lceil \log_3(2n+1) \rceil = \log_3 9 = 3$

→ Ω(logn)

Binary Search → Binary Decision Tree



Lower bound is then $\lceil \log_2(n+1) \rceil$ ➡ **A tight lower bound**

Leaves → unsuccessful search

Parent nodes → successful search

P, *NP*, and *NP*-Complete Problems

As we discussed, problems that can be solved in polynomial time are usually called tractable and the problems that cannot be solved in polynomial time are called intractable, now

Is there a polynomial-time algorithm that solves the problem?

Possible answers:

- yes
- no
  - because it can be proved that all algorithms take exponential time
  - because it can be proved that no algorithm exists at all to solve this problem
- don't know
- don't know, but if such algorithm were to be found, then it would provide a means of solving many other problems in polynomial time

Types of Problems

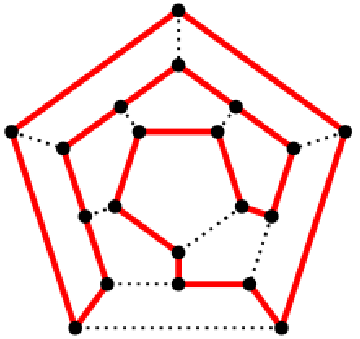
Two types of problems:

- ***Optimization problem:*** **construct a solution that maximizes or minimizes some objective function**
  - MST, all shortest paths, single source shortest paths, …
- ***Decision problem:*** **answer yes/no to a question**
  - Selection, searching, …

Many problems have BOTH decision and optimization versions.

Eg: Traveling Salesman Problem

- *optimization*: find Hamiltonian cycle of minimum weight
- *decision*: Is there a Hamiltonian cycle of weight < *k*



Hamiltonian Circuit*: a closed path in a graph that visits every node in the graph exactly once*

Deterministic VS Nondeterministic Algorithm

A *deterministic algorithm* is the algorithm we discussed before

- E.g., a math function: given a specific input, generate the same and unique output in different runs

A *nondeterministic algorithm* is the counterpart

- *May have different outputs in different runs*
- *It is a two-stage process:*
  - *Guessing stage:* generate a random string ***S*** as a candidate solution
  - *Verification stage:* using a ***deterministic*** algorithm which takes the original input ***I*** and ***S*** as input and determine if ***S*** is a solution to ***I***

Why becomes nondeterministic?

- *System noise*
- *random number generator*

Deterministic VS Nondeterministic Algorithm

A problem can have BOTH *deterministic* and *nondeterministic* algorithms

Example:

Shortest path problem: find the shortest path from ***a*** to ***b*** in a weighted graph

- **Deterministic algorithm:** searching the shortest path (e.g., brute force enumerating)
- **Nondeterministic algorithm:** generate a path ***P*** and decide whether ***P*** is a simple path (all vertices on the path are distinct) from ***a*** to ***b*** of length<= Threshold.

The Class P & *NP*

P: the class of decision *problems* that are solvable by deterministic algorithms in $O(p(n))$, where $p(n)$ is a polynomial on ***n***

NP: the class of decision *problems* that are solvable in polynomial time by *nondeterministic* algorithms

Thus *NP* can also be thought of as the class of problems

- whose solutions can be verified in polynomial time; or
- that can be solved in polynomial time on a machine that can pursue infinitely many paths of the computation in parallel

Note that *NP* stands for "Nondeterministic Polynomial-time"

All the problems in *P* can also be solved in this manner (but no guessing is necessary), so we have:

$$P \subseteq NP$$

Example: Conjunctive Normal Form (CNF) Satisfiability

Problem: Is a Boolean expression in its conjunctive normal form (CNF), i.e., are there "true" or "false" assignments of these variables that makes the Boolean expression true?

This problem is in *NP*.

Nondeterministic algorithm:

- Guess truth assignment
- Check assignment to see if it satisfies CNF formula

Example: (Boolean operation)

$$(a \vee \overline{b} \vee \overline{c}) \wedge (\overline{a} \vee b) \wedge (\overline{a} \vee \overline{b} \vee \overline{c})$$

$\vee$ ***is logic "or"***

$\wedge$ ***is logic "and" or "logical conjunction"***

Truth assignments: $a = true, b = true, c = \ false \Rightarrow$

$$\textbf{the entire expression} = true$$

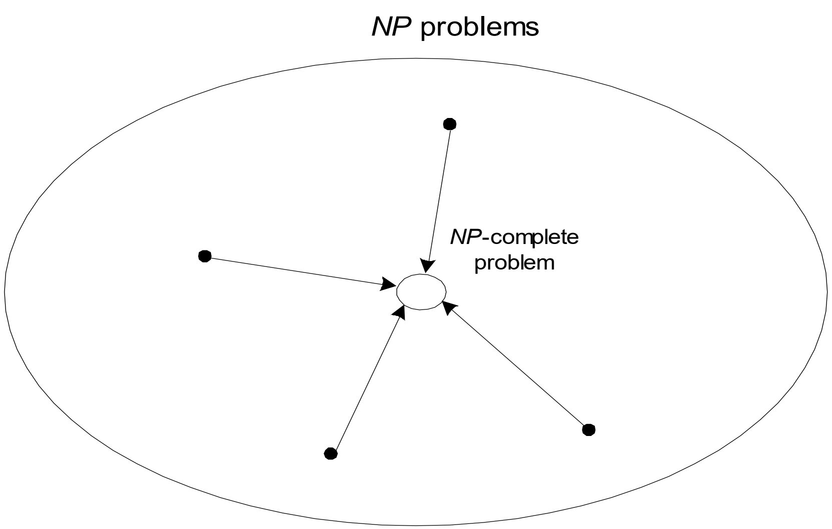
Checking phase: Θ(*n*)

NP-Complete problems

A decision problem *D* is <u>*NP*-complete</u> iff

1. $D \in NP$
2. every problem in *NP* is polynomial-time reducible to *D*

The class of *NP*-complete problems is denoted <u>*NPC*</u>



Polynomial Reductions

A decision problem D_1 is said to be polynomial reducible to a decision problem D_2 if there exists a function *f* that transforms instances of D_1 to instances of D_2 such that

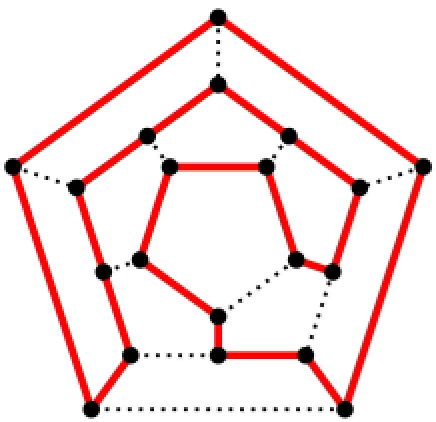
1. *f* maps all "yes" instances of D_1 to "yes" instances of D_2 and all "no" instances of D_1 to "no" instances of D_2
2. *f* is computable by a polynomial-time algorithm

If D_2 can be solved in polynomial time → D_1 can be solved in polynomial time

Polynomial Reductions

Example: Polynomial-time reduction of Hamiltonian Circuit to decision version of Traveling Salesman Problem (Is there a solution of TSP with total distance no larger than *k=n?*) given integer distance

Hamiltonian Circuit: a closed path in a graph that visits every node in the graph exactly once

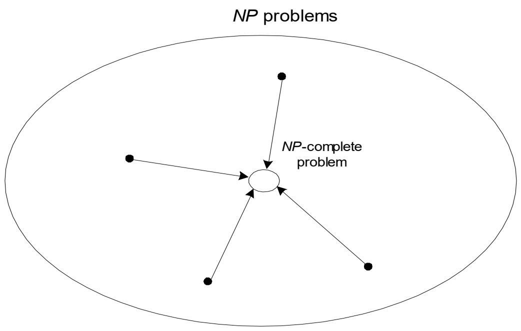


Traveling Salesman: find the shortest path that visits every city exact once and returns to the origin

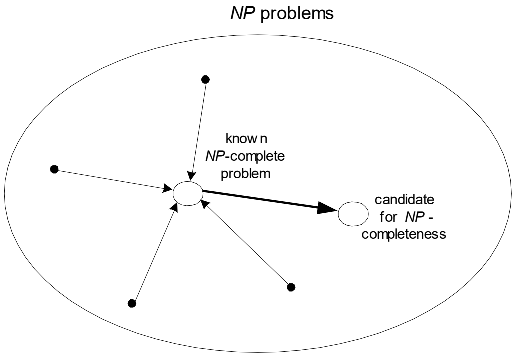
To Prove a Decision Problem is in NPC

1. Prove it is in *NP* (verification takes polynomial time)

2. Prove that all problems in *NP* is reducible to this problem



3. Or Prove that a known *NPC* problem is reducible to this problem



BIG problem: If we can prove any given NPC problem can be solve in polynomial time → P=NP

Chapter 12: Coping with the Limitations of Algorithm Power

There are two principal approaches to tackling NP-hard problems or other "intractable" problems:

- Use a strategy that guarantees solving the problem exactly but doesn't guarantee to find a solution in polynomial time

- Use an approximation algorithm that can find an approximate (sub-optimal) solution in polynomial time

Exact solutions

The exact solution approach includes the strategies:

- ***Exhaustive search* (brute force)**
  - useful only for small instances
- **Dynamic programming**
  - Applicable for some problems, e.g., knapsack problem, TSP
- ***Backtracking***
  - eliminates some cases from consideration
  - yields solutions in reasonable time for many instances but worst case is still exponential
- ***Branch-and-bound***
  - Only applicable for optimization problems
  - further cuts down on the search
  - fast solutions for most instances
  - worst case is still exponential

Need a state-space tree

Nodes: partial solutions
Edges: choices in completing solutions

Backtracking

Construct the *state-space tree*:

- nodes: partial solutions
- edges: choices in completing solutions

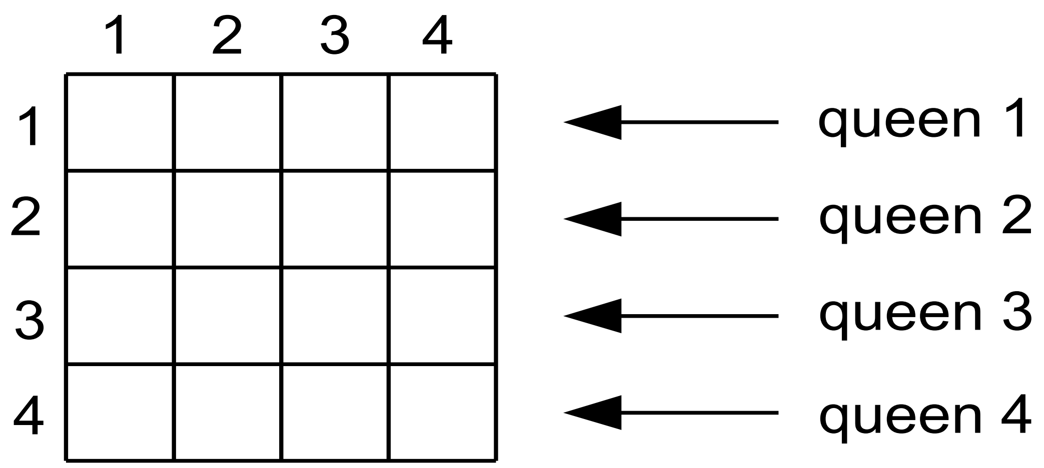
Explore the state space tree using depth-first search (DFS)

"Prune" non-promising subtrees

- DFS stops exploring subtree rooted at nodes leading to no solutions and
- "backtracks" to its parent node

The Most Popular Example: The *n*-Queen problem

Place n queens on an n-by-n chess board so that no two of them are in the same row, column, or diagonal. Solution exists for all natural numbers except n=2 and n=3.



Brute force algorithm: only allow one queen at each row $\Theta(n^n)$

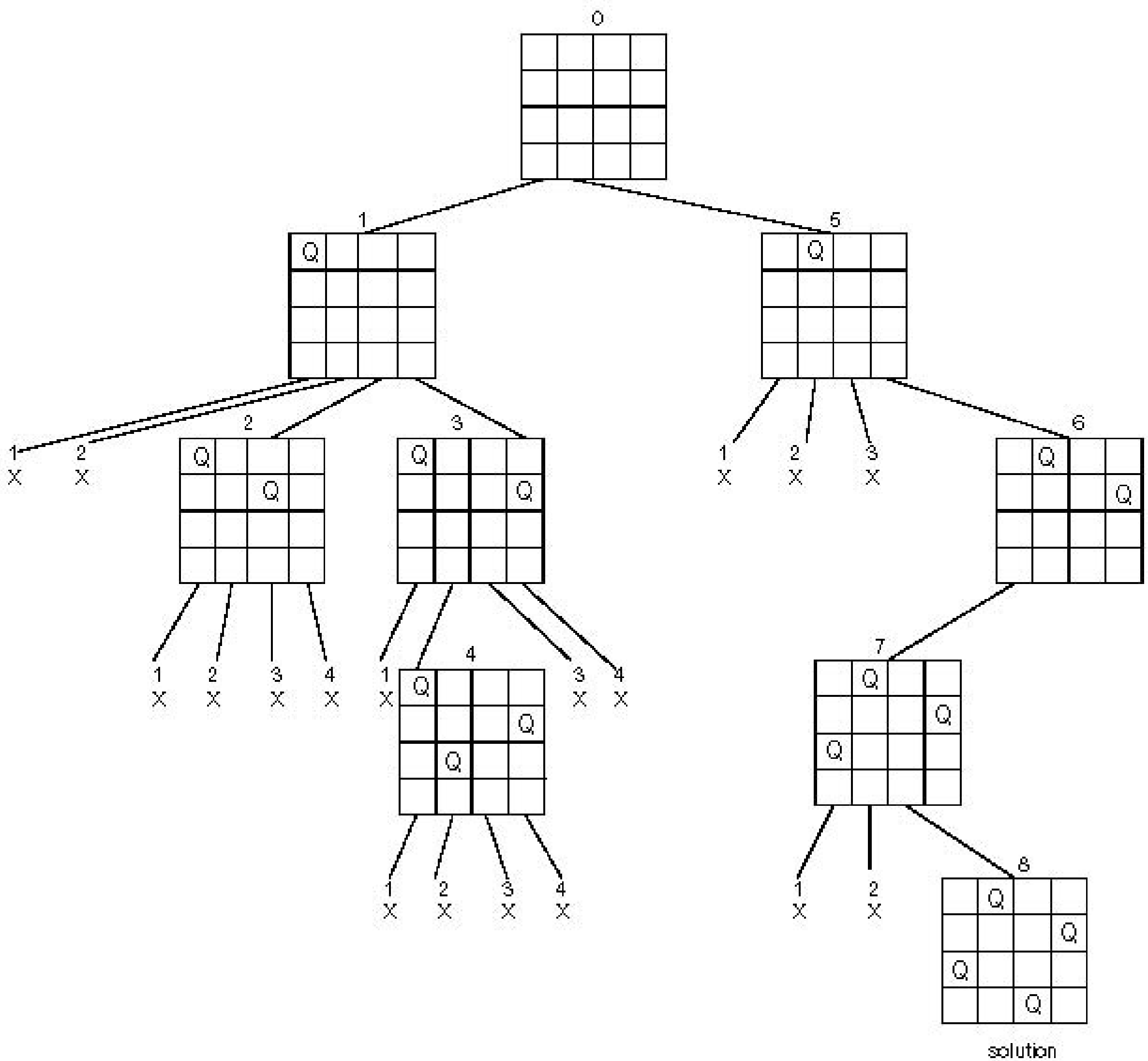
State-space of the four-queens problem

1st row

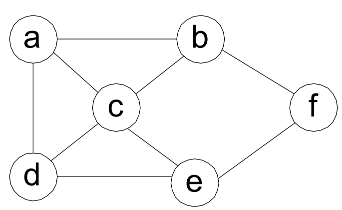
2nd row

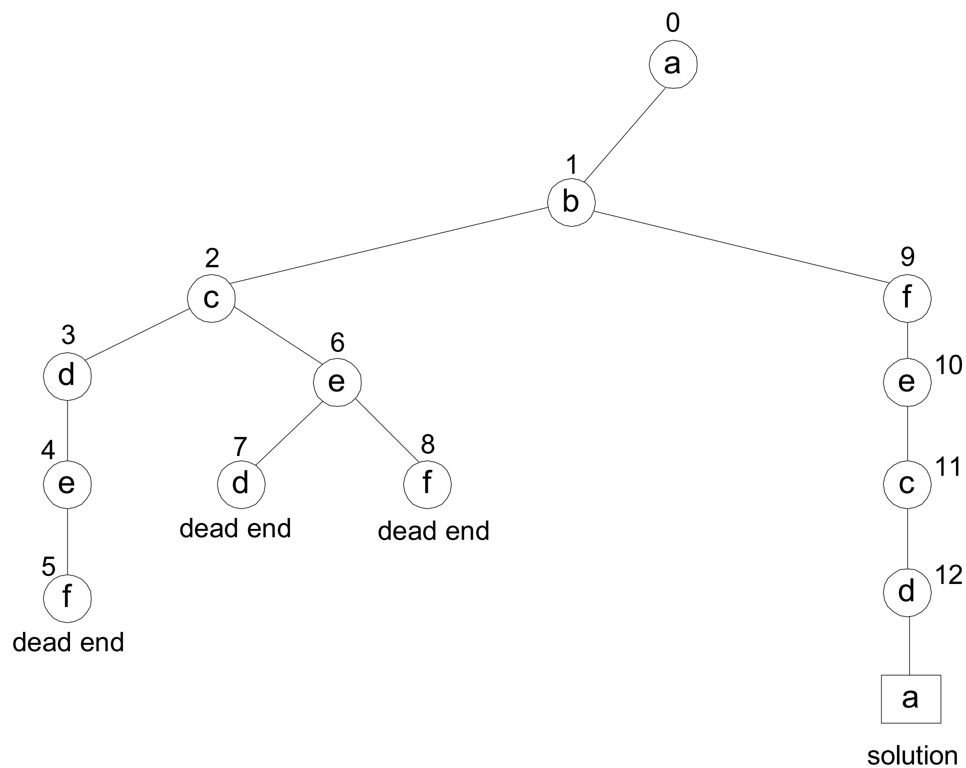
3rd row

4th row



Example: Hamiltonian Circuit Problem

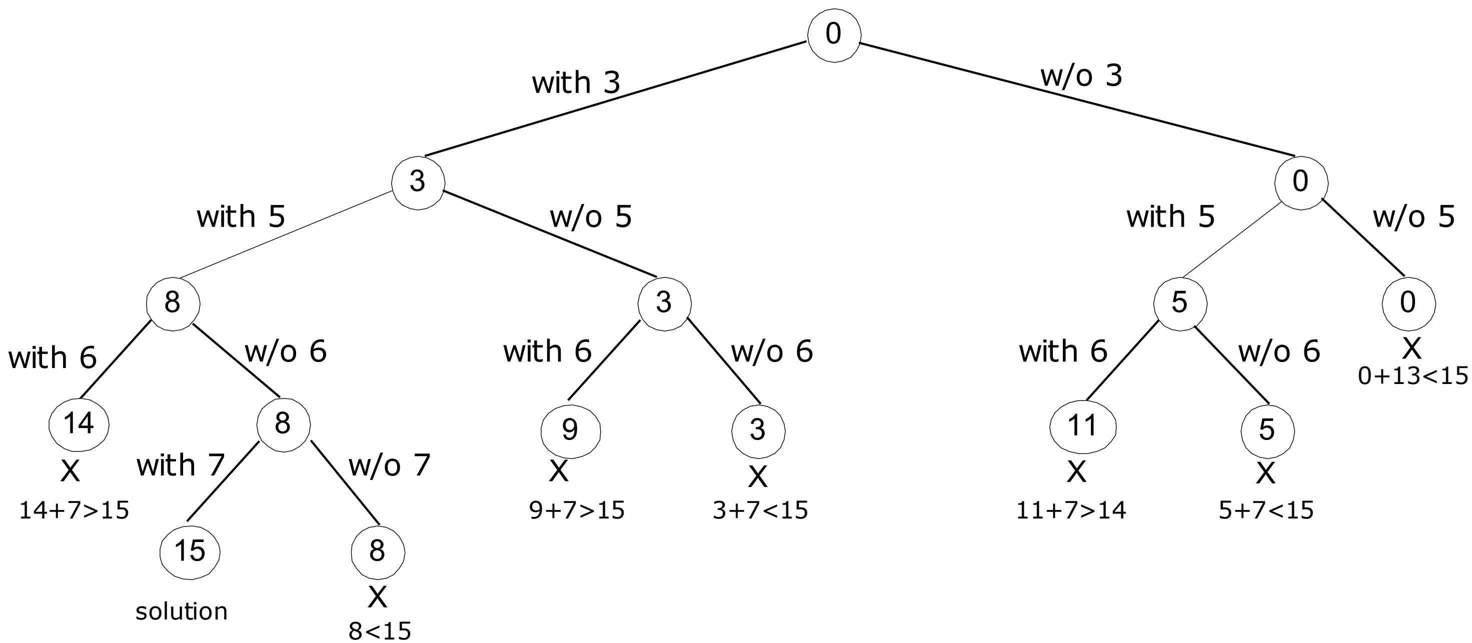




Subset-Sum Problem

Find a subset of a given set $S=\{s_1,s_2,\ldots,s_n\}$ of *n* positive integers whose sum is equal to a given positive integer *d*

For example: S={3,5,6,7} and d=15 → solutions {3,5,7}



Branch and Bound

An enhancement of backtracking.

Applicable to optimization problems

Uses a lower bound for the value of the objective function for each node (partial solution) to:

- no solution can beat the lower bound
- guide the search through state-space
- rule out certain branches as "unpromising" – do not explore these subtrees
- using a "best-first" rule

Example: The assignment problem

For example:

	Job 1	*Job 2*	*Job 3*	*Job 4*
Person a	*9*	*2*	*7*	*8*
Person b	*6*	*4*	*3*	*7*
Person c	*5*	*8*	*1*	*8*
Person d	*7*	*6*	*9*	*4*

Cost matrix

Select one element in each row of the cost matrix C so that:

- ***no two selected elements are in the same column; and***
- ***the sum is minimized***

If using exhaustive search, the permutation of n persons ⇨ $\Theta(n!)$

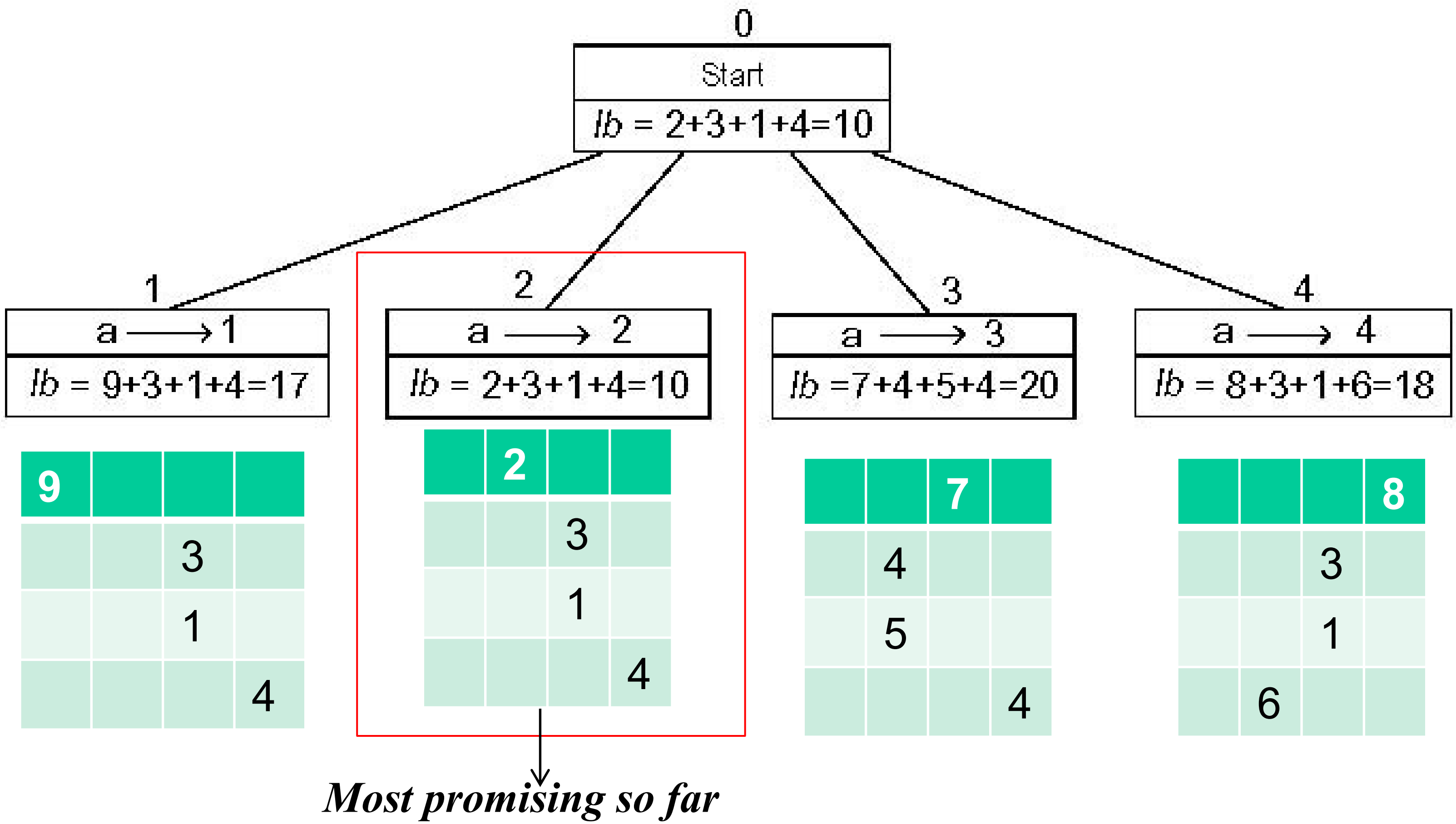
Example: The assignment problem

	Job 1	*Job 2*	*Job 3*	*Job 4*
Person a	*9*	*2*	*7*	*8*
Person b	*6*	*4*	*3*	*7*
Person c	*5*	*8*	*1*	*8*
Person d	*7*	*6*	*9*	*4*

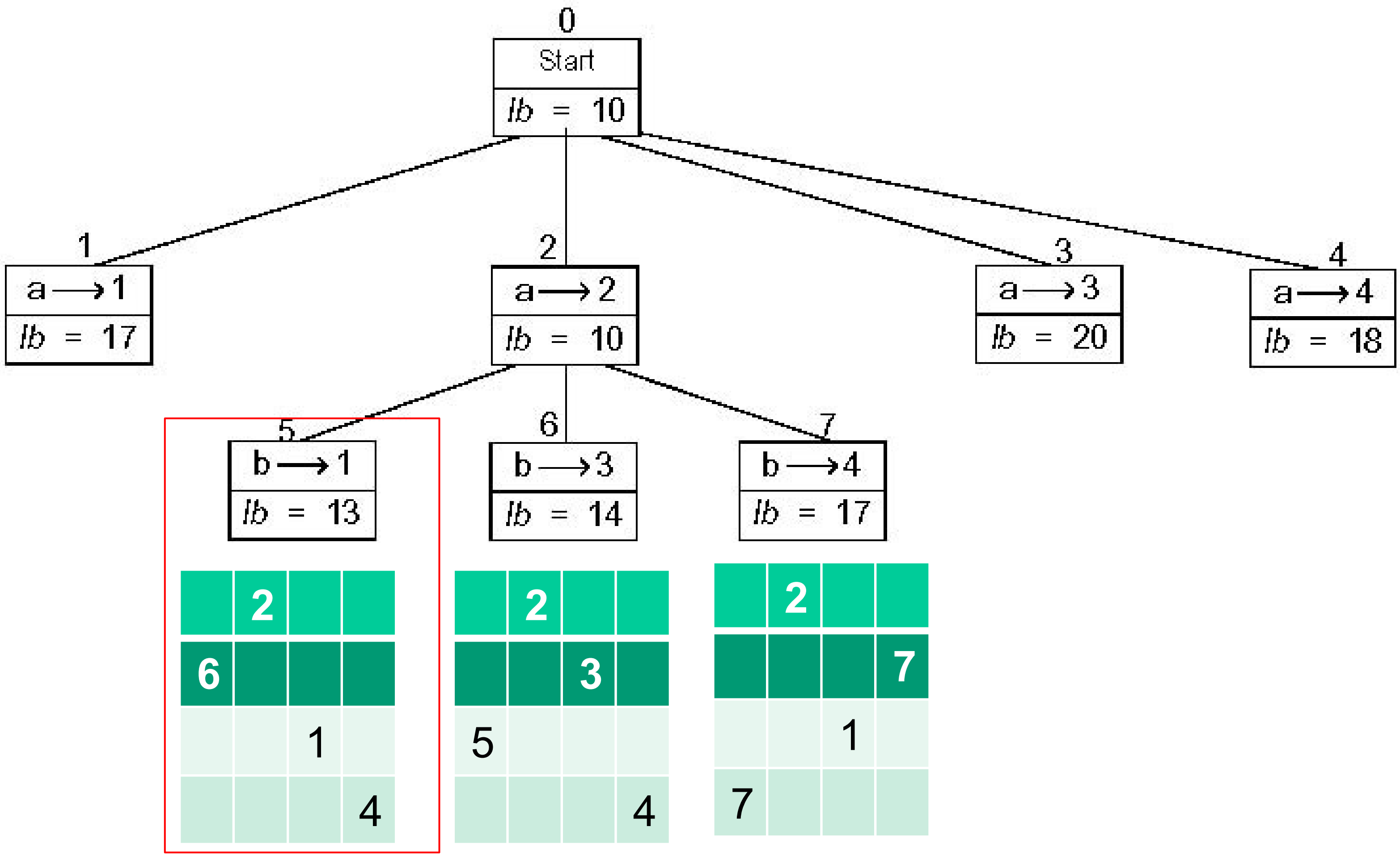
Lower bound*: Any solution to this problem will have total cost of *at least*:** ***The summation of the smallest elements in each row

No solution can beat the lower bound!

Assignment problem: lower bounds



State-space levels 0, 1, 2



Complete state-space