ANALYSIS AND DESIGN OF ALGORITHMS

UNIT-V

CHAPTER 11:

LIMITATIONS OF ALGORITHM POWER

OUTLINE

- **Limitations of Algorithm Power**
  - Decision Trees
    - Decision Trees for Sorting Algorithms
    - Decision Trees for Searching a Sorted Array
  - P, NP, and NP-complete Problems
    - P and NP Problems
    - NP-Complete Problems

DECISION TREES

- Performance of algorithms like searching, sorting can be studied with a device called the **decision tree**.

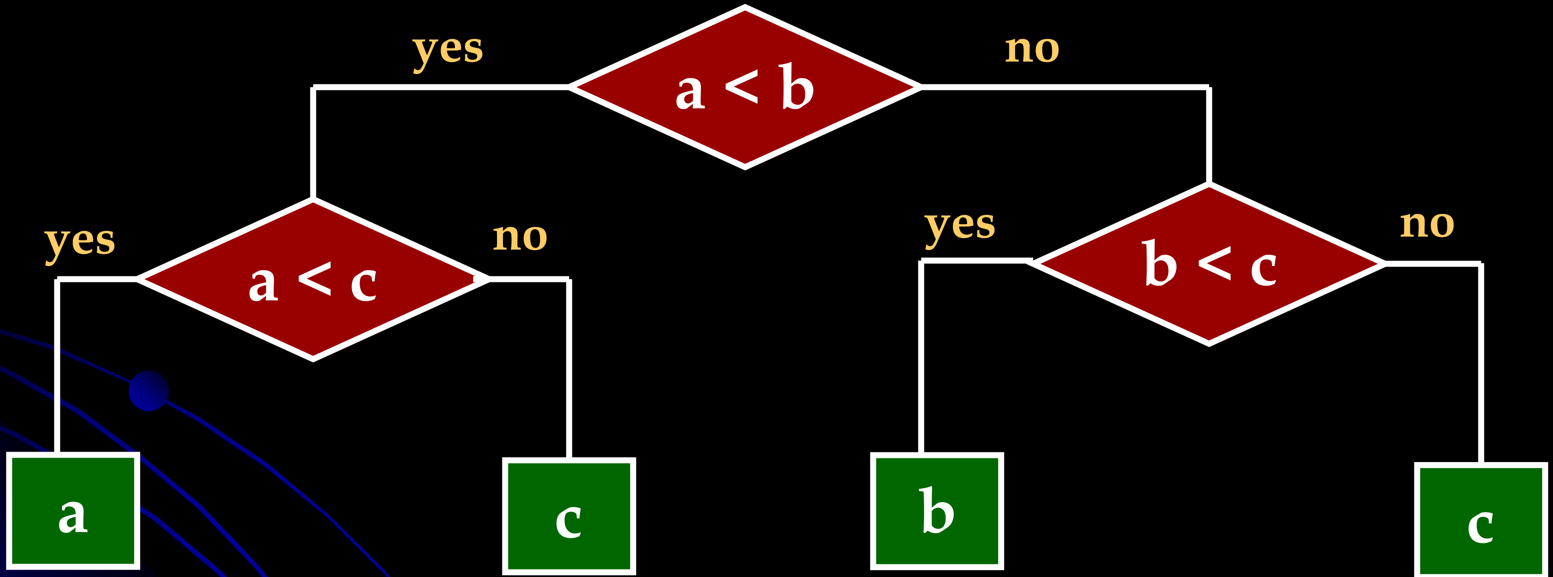

Figure: Decision tree for finding a minimum of three numbers.

- Each internal node of a binary decision tree represents a key comparison indicated in the node, e.g., $k < k'$.

DECISION TREES

- The **node's left subtree** contains the information about subsequent comparisons made if $k < k'$, while its **right subtree** does the same for the case of $k > k'$.

- Each **leaf represents a possible outcome** of the algorithm's run on some input of size *n*. The **number of leaves can be greater** than number of outcomes because, for some algorithms, the same outcome can be arrived through a different chain of comparisons.

- The algorithm's work on a particular input of size *n* can be traced by a **path from the root to a leaf in its decision tree**, and the number of **comparisons** made by the algorithm on such a run is equal to the **number of edges in this path**.

- Hence, the **number of comparisons in the worst case** is equal to the **height** of the algorithm's decision tree.

DECISION TREES

- For any binary tree with l leaves and height h, $h \geq \lceil \log_2 l \rceil$ … (1)
- Inequality (1) puts a lower bound on the heights of binary decision trees. Such a bound is called the ***information- theoretic lower bound***.

Decision Trees for Sorting Algorithms

- Most sorting algorithms are comparison-based, i.e., they work by comparing elements in a list to be sorted.
- Therefore, by studying properties of decision trees for comparison-based sorting algorithms, we can derive important lower bounds on the time efficiencies of such algorithms.

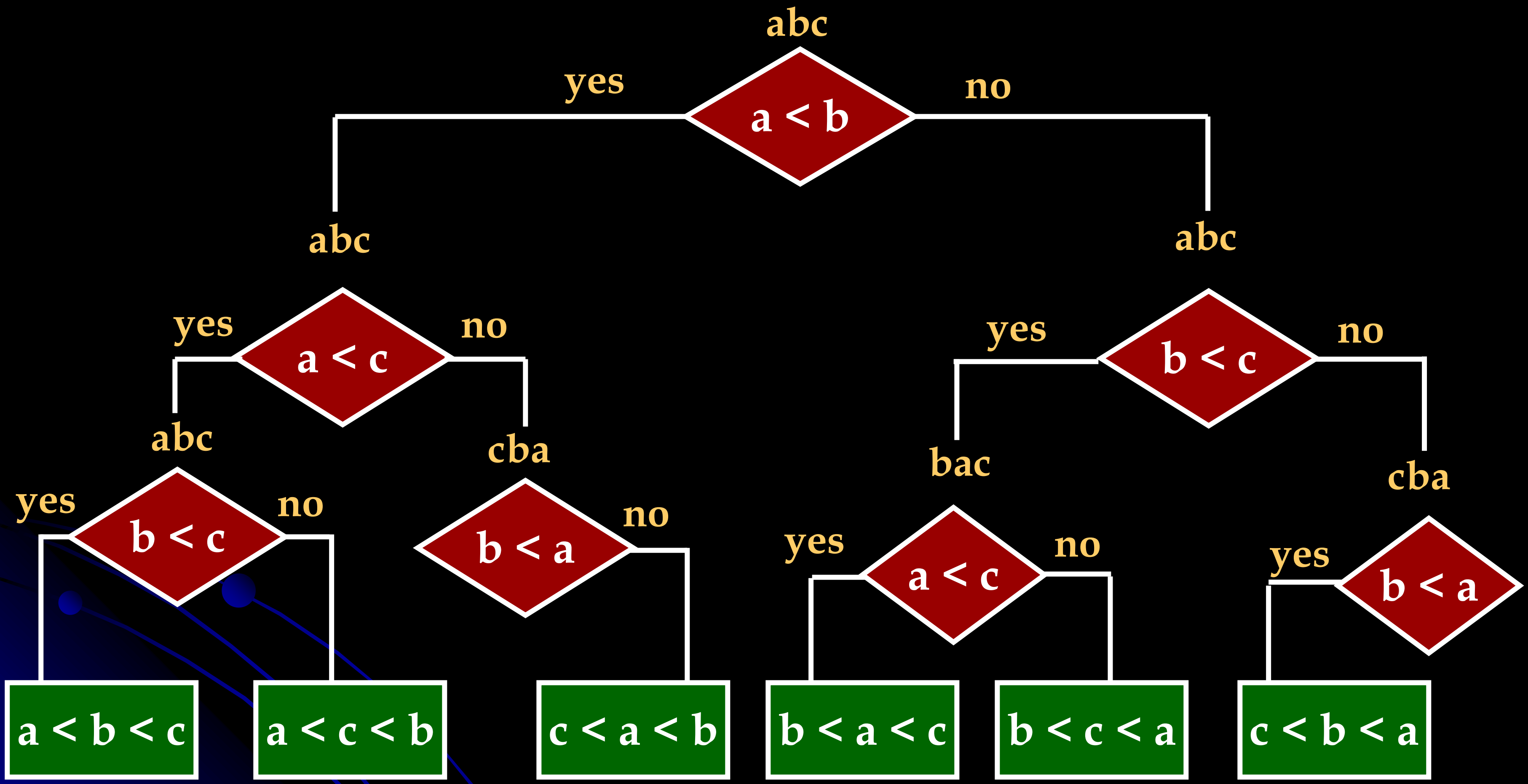

Figure: Decision tree for the three-element selection sort. A triple above a node indicates the state of the array being sorted. Note the two redundant comparisons b < a with a single possible outcome because of the results of some previously made comparisons.

Decision Trees for Sorting Algorithms

- We can interpret an outcome of a sorting algorithm **as finding a permutation** of the element indices of an input list that puts the list's elements in ascending order.

  For example, for the outcome a < c < b obtained by sorting a list a, b, c (refer previous slide), the permutation in question is 1, 3, 2.

- Hence, the number of possible outcomes for **sorting an arbitrary *n*-element list is equal to *n* !.**

Decision Trees for Sorting Algorithms

- The worst-case number of comparisons made by comparison-based sorting algorithm cannot be less than $\lceil \log_2 n! \rceil$:

$$C_{worst}(n) \geq \lceil \log_2 n! \rceil$$

Using Stirling's formula for $n!$, we get

$$\lceil \log_2 n! \rceil \approx \log_2 \sqrt{2\Pi n}(n/e)^n = n\log_2 n - n\log_2 e + \frac{\log_2 n}{2} + \frac{\log_2 2\Pi}{2} \approx n \log_2 n$$

- About **n log$_2$ n comparisons** are necessary to sort an arbitrary n-element list by any comparison-based sorting algorithm.

Decision Trees for Sorting Algorithms

- We can also use decision trees **for analyzing the average-case behavior** of a comparison-based sorting algorithm.

- We can compute the average number of comparisons for a particular algorithm as the **average depth of its decision tree's leaves**, i.e., as the average path length from the root to the leaves.

  For example, for the three-element insertion sort whose decision tree is given in next slide, this number is $(2 + 3 + 3 + 2 + 3 + 3)/6 = 2\frac{2}{3}$.

- Under the standard assumption that all $n!$ outcomes of sorting are equally likely, the following lower bound on the average number of comparisons C_{avg} made by any comparison-based algorithm in sorting an n-element list has been proved:

$$C_{avg}(n) \geq \log_2 n!$$

- This lower bound is about $\mathbf{n \log_2 n}$.

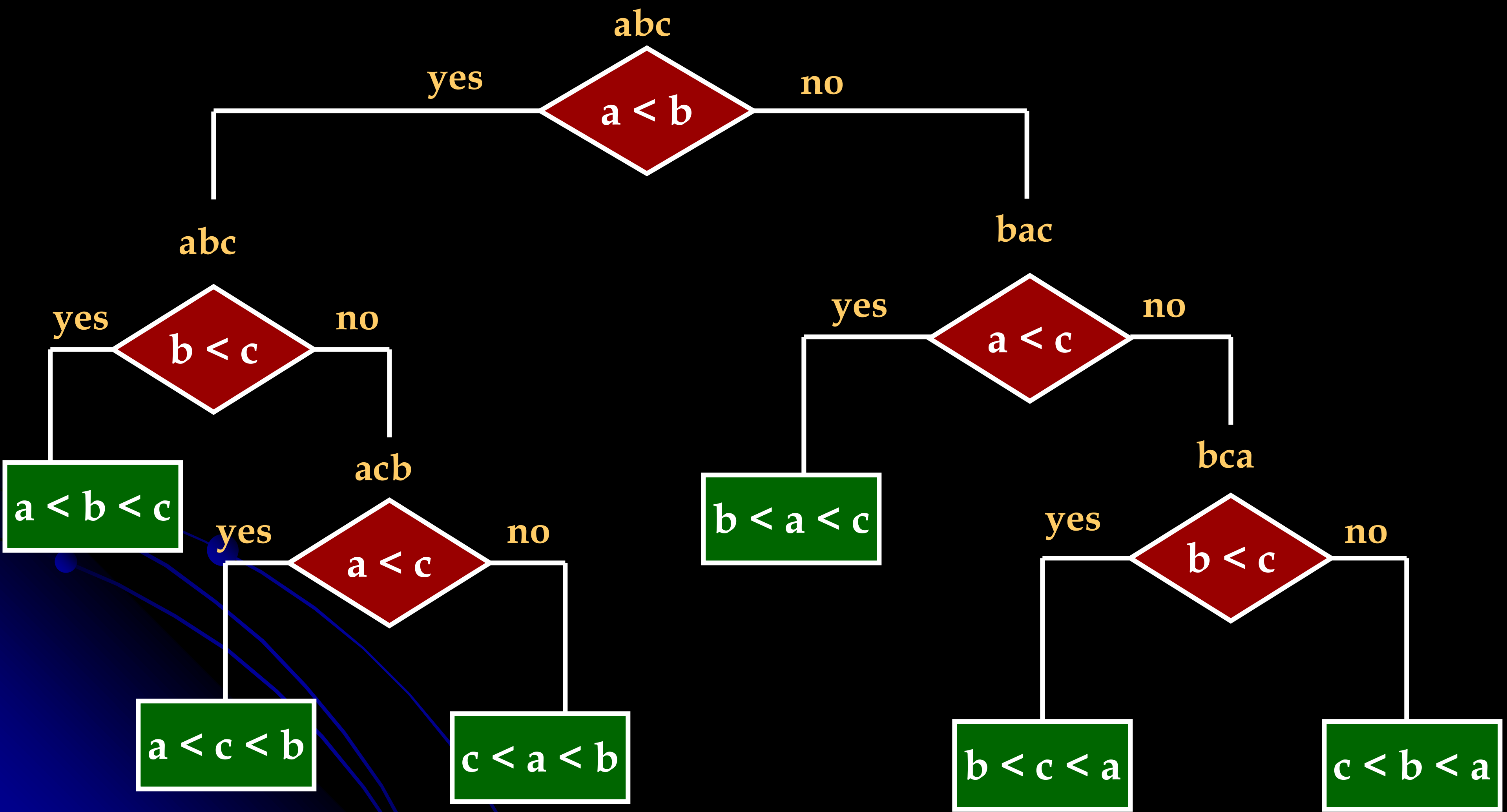

Figure: Decision tree for the three-element insertion sort.

Decision Trees for Searching a Sorted Array

- The decision trees can be used for establishing lower bounds on the number of key comparisons in searching a sorted array of n keys:
  $A[0] < A[1] < \ . \ . \ . < A[n-1]$. The principal algorithm for this problem is binary search.

- The number of comparisons made by **binary search in the worst case** is given by the formula

$$C_{worst}(n) = \lfloor \log_2 n \rfloor + 1 = \lceil \log_2 (n+1) \rceil .$$

  we will use decision trees to determine whether this is the smallest possible number of comparisons.

- Since we are dealing with three-way comparisons in which search key K is compared with some element A[i] to see whether K <A[i], K=A[i], or K > A[i], it is natural to try **using ternary decision trees**.

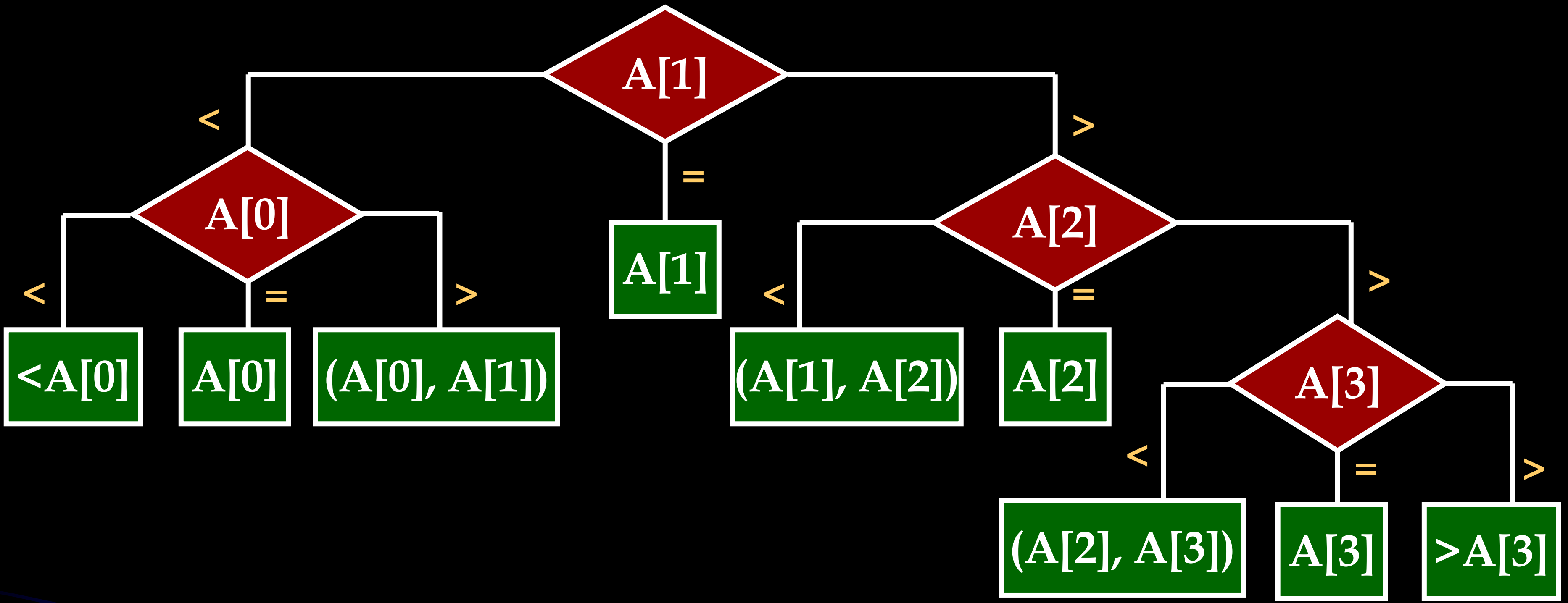

Figure: Ternary decision tree for binary search in a four-element array.

- The leaves indicate either a matching element in the case of a successful search or a found interval that the search key belongs to in the case of an unsuccessful search.

- For an array of n elements, all such decision trees will have $2n + 1$ leaves (n for successful searches and $n+1$ for unsuccessful ones).

Decision Trees for Searching a Sorted Array

- Since the minimum height h of a ternary tree with l leaves is $\lceil \log_3 l \rceil$, we get the following lower bound on the number of worst-case comparisons:

$$C_{worst}(n) \geq \lceil \log_3 (2n + 1) \rceil .$$

- To obtain a better lower bound, we should consider binary rather than ternary decision trees such as the one shown below:

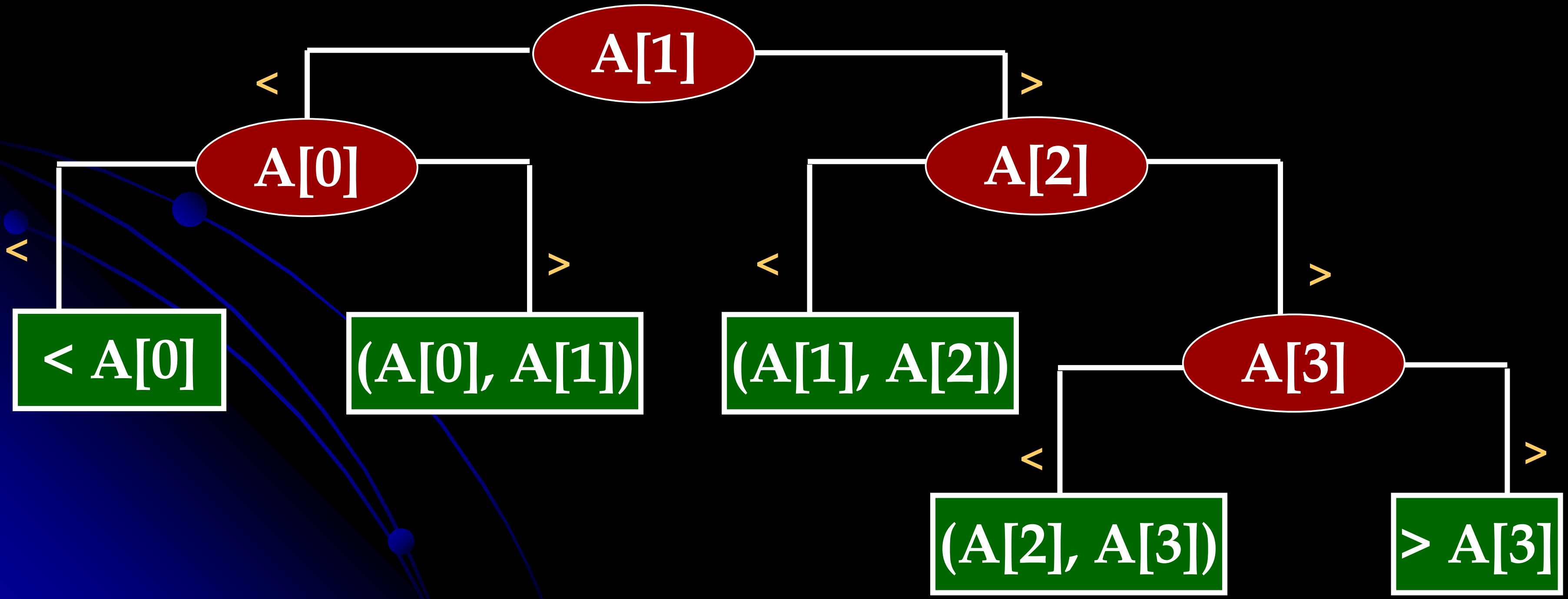

Figure: Binary decision tree for binary search in a four-element array.

Decision Trees for Searching a Sorted Array

- **Internal nodes** in such a tree (refer previous slide) correspond to the same three-way comparisons as before, but they also serve as terminal nodes for successful searches.
- **Leaves** therefore represent only unsuccessful searches, and there are $n + 1$ of them for searching an n-element array.
- The binary decision tree is simply the ternary decision tree with all the middle subtrees eliminated.
- For binary decision trees,

  $$C_{worst}(n) \geq \lceil \log_2 (n+1) \rceil .$$

  This inequality closes the gap between the lower bound and the number of worst-case comparisons made by binary search.

P, NP, AND NP-COMPLETE PROBLEMS

- We say that an algorithm solves a problem in **polynomial time** if its worst-case time efficiency belongs to O($p(n)$) where $p(n)$ is a polynomial of the problem's input size n.

- Problems that can be solved in polynomial time are called **tractable**.

- Problems that cannot be solved in polynomial time are called **intractable**.

- Problems like GCD of two integers, sorting, searching, checking connectivity, finding a minimum spanning tree, finding shortest paths in a weighted graph etc. can be solved in polynomial time.

P, NP, AND NP-COMPLETE PROBLEMS

- **Class P** is a class of decision problems that can be solved in polynomial time by (deterministic) algorithms. This class of problems is called polynomial.

- Some decision problems cannot be solved at all by any algorithm. Such problems are called **undecidable**.

  Example: Halting problem: given a computer program and an input to it, determine whether the program will halt on that input or continue working indefinitely on it.

- There are many important problems for **which no polynomial-time algorithm has been found**. Some of the best-known problems that fall into this category are:
  - ***Hamiltonian circuit:*** Determine whether a given graph has a Hamiltonian circuit (a path that starts and ends at the same vertex and passes through all the other vertices exactly once).

P, NP, AND NP-COMPLETE PROBLEMS

- ***Traveling salesman:*** Find the shortest tour through *n* cities with known positive integer distances between them (find the shortest Hamiltonian circuit in a complete graph with positive integer weights).
- ***Partition problem:*** Given *n* positive integers, determine whether it is possible to partition them into two disjoint subsets with the same sum.
- ***Graph coloring:*** For a given graph, find its chromatic number (the smallest number of colors that need to be assigned to the graph's vertices so that no two adjacent vertices are assigned the same color).

P, NP, AND NP-COMPLETE PROBLEMS

- A **nondeterministic algorithm** is a two-stage procedure that takes as its input an instance *I* of a decision problem and does the following:
  - **Nondeterministic ("guessing") stage:** An arbitrary string *S* is generated that can be thought of as a candidate solution to the given instance *I*.
  - **Deterministic ("verification") stage:** A deterministic algorithm takes both *I* and *S* as its input and outputs yes if *S* represents a solution to instance *I*. (If *S* is not a solution to instance *I*, the algorithm either returns no or is allowed not to halt at all.)

- **Class NP** is the class of decision problems that can be solved by nondeterministic polynomial algorithms. This class of problems is called **nondeterministic polynomial**.

- **P ⊆ NP .** This is true because, if a problem is in *P*, we can use the deterministic polynomial-time algorithm that solves it in the verification-stage of a nondeterministic algorithm that simply ignores string *S* generated in its nondeterministic (“guessing”) stage.

What problems are in *NP*?

- Hamiltonian circuit existence
- Traveling salesman problem
- Partition problem: Is it possible to partition a set of *n* integers into two disjoint subsets with the same sum?

P, NP, AND NP-COMPLETE PROBLEMS

- A decision problem D_1 is said to be **polynomially reducible** to a decision problem D_2 if there exists a function t that transforms instances of D_1 to instances of D_2 such that
  1. t 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. t is computable by a polynomial-time algorithm.

- This definition implies that if a problem D_1 is polynomially reducible to some problem D_2 that can be solved in polynomial time, then problem D_1 can also be solved in polynomial time.

NP-Complete

- A decision problem D is said to be ***NP*-complete** if
  1. it belongs to class ***NP;***
  2. every problem in NP is polynomially reducible to D.

- The definition of NP-completeness implies that if there exists a deterministic polynomial-time algorithm for just one NP-complete problem, then every problem in NP can be solved in polynomial time by a deterministic algorithm, and hence **P = NP**.

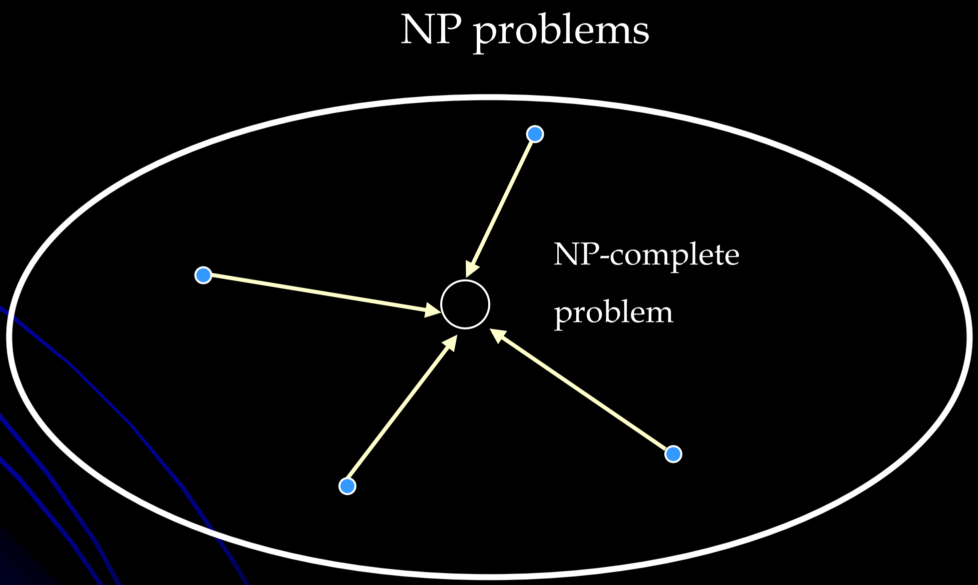

Figure: Notion of an NP-complete problem. Polynomial-time reductions of NP problems to an NP-complete problems are shown by arrows.

End of Chapter 11