Limitations of Algorithm Power

Limitations

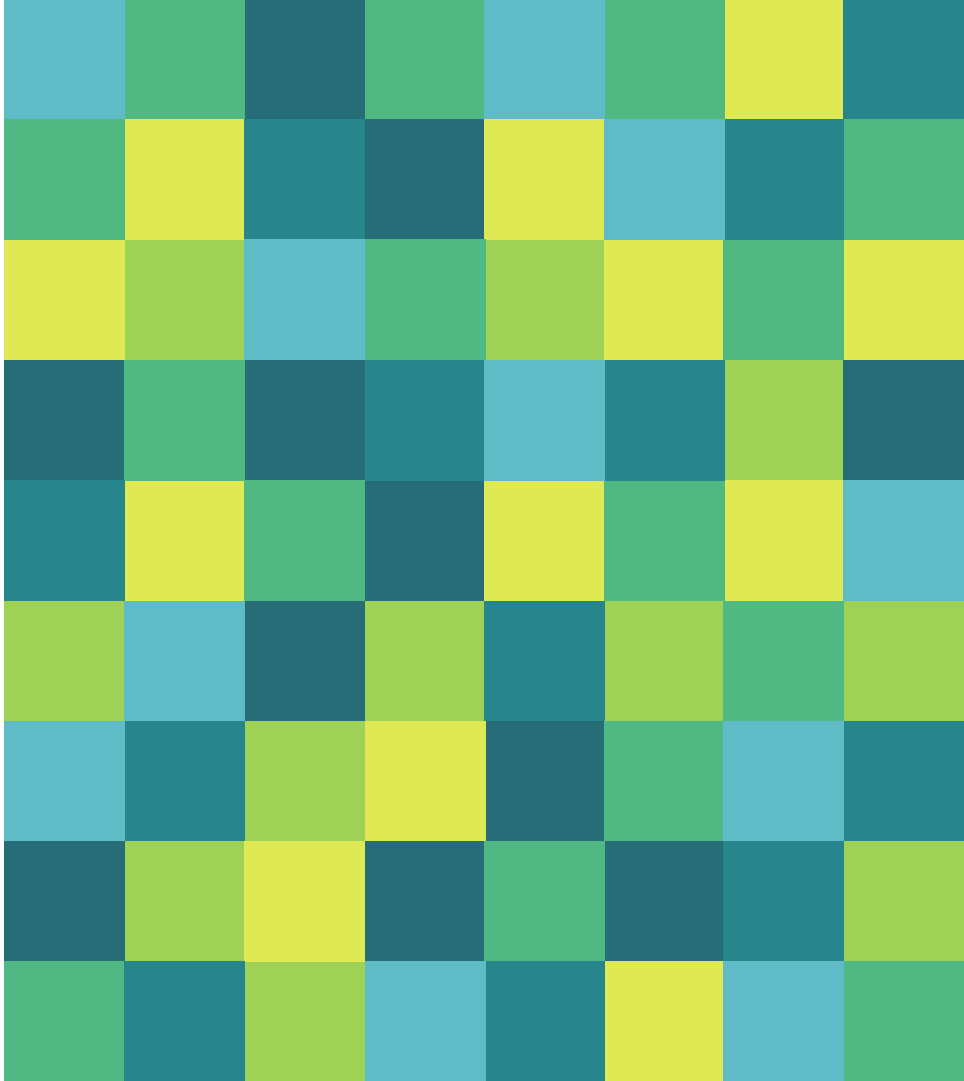
- In the course of the semester, we have come across many algorithms for solving different types of problems.
- But, algorithms have their own limitations.
- There are problems which do not have an algorithm to solve them.
- And, there are other problems which have algorithms to solve them but not in polynomial time.

Limitations

- In this chapter, we shall study the limitations of an algorithm.

1.

Lower Bound Arguments



Lower Bound Arguments

- To determine if an algorithm is practically useful or not, we need to look at its efficiency.
- The efficiency of an algorithm can be determined in two ways:
  - Asymptotic Efficiency
  - Lower Bound Arguments

Asymptotic Efficiency

- We can determine the asymptotic efficiency of an algorithm.
- Example:
  - Selection Sort has quadratic efficiency
  - Towers of Hanoi has exponential efficiency
- According to this comparison, selection sort has a better efficiency than Towers of Hanoi.

Asymptotic Efficiency

- But this comparison is similar to comparing apples with oranges.
- The fairer approach would be to compare the asymptotic efficiency of an algorithm with other algorithms solving the same problem.
- So, according to this concept, selection sort has slower efficiency because there are sorting algorithms which have n log n efficiency.

Lower Bound Arguments

- When trying to ascertain the practical applicability of an algorithm, it is desirable to know the best possible efficiency of an algorithm.
- Knowing such a ***lower bound*** can tell us how much improvement we can hope to achieve in our quest for a better algorithm for the problem in question.
- If such a bound is ***tight,*** i.e., we already know an algorithm in the same efficiency class as the lower bound, we can hope for a constant-factor improvement at best.

Lower Bound Arguments

- If there is a gap between the efficiency of the fastest algorithm and the best lower bound known, the door for possible improvement remains open.
- Either a faster algorithm matching the lower bound could exist or a better lower bound could be proved.

Trivial Lower Bounds

- The simplest method of obtaining a lower-bound class is based on counting the number of items in the problem's input that must be processed and the number of output items that need to be produced.
- Since any algorithm must at least "read“ all the items it needs to process and "write" all its outputs, such a count yields a ***trivial lower bound.***

Trivial Lower Bounds - Examples

- Generating Permutations – $\Omega(n!)$
- Polynomial Evaluation – $\Omega(n)$
- Matrix Multiplication – $\Omega(n^2)$
- Travelling Salesman Problem - $\Omega(n^2)$

Trivial Lower Bounds - Problems

- They are sometimes too low to be useful.
- It assumes that every element of input needs to be processed to solve the problem, which is not the case.
  - For example, in searching problem, we need not check every element of the input.

Adversary Arguments – Example 1

- Consider a game of guessing a number between 1 to n selected by somebody by asking that person yes / no questions.
- The minimum number of questions that need to be asked to solve this problem is floor($\log_2 n$).
- The adversary starts by considering each of the numbers between 1 and *n* as being potentially selected.

Adversary Arguments – Example 1

- After each question, the adversary gives an answer that leaves him with the largest set of numbers consistent with this and all the previously given answers. (This strategy leaves him with at least one half of the numbers he had before his last answer.)
- If an algorithm stops before the size of the set is reduced to one, the adversary can exhibit a number that could be a legitimate input the algorithm failed to identify.

Adversary Arguments – Example 1

- It is a simple technical matter now to show that one needs floor($\log_2 n$) iterations to shrink an n-element set to a one-element set by halving and rounding up the size of the remaining set.
- Hence, at least $\log_2 n$ questions need to be asked by any algorithm in the worst case.

Adversary Arguments – Example 1

- It is a simple technical matter now to show that one needs floor($\log_2 n$) iterations to shrink an n-element set to a one-element set by halving and rounding up the size of the remaining set.
- Hence, at least $\log_2 n$ questions need to be asked by any algorithm in the worst case.

Adversary Arguments

- Based on following the logic of a malevolent but honest adversary.
- The malevolence makes him push the algorithm down the most time-consuming path.
- While his honesty forces him to stay consistent with the choices already made.
- A lower bound is then obtained by measuring the amount of work needed to shrink a set of potential inputs to a single input along the most time-consuming path.

Adversary Arguments – Example 2

- Consider the problem of merging two sorted lists of size n:

  $$a_1 < a_2 < \ldots < a_n \textbf{ and } b_1 < b_2 < \ldots < b_n$$

  into a single sorted list of size $2n$.
- For simplicity, we assume that all the a's and b's are distinct, which gives the problem a unique solution.

Adversary Arguments – Example 2

- The adversary will employ the following rule: reply true to the comparison $a_i < b_j$ if and only if $i < j$.
- This will force any correct merging algorithm to produce the only combined list consistent with this rule:

$$b_1 < a_1 < b_2 < a_2 < \ldots < b_n < a_n$$

Adversary Arguments – Example 2

- To produce this combined list, any correct algorithm will have to explicitly compare $2n - 1$ adjacent pairs of its elements, i.e., b1 to *a1*, a1 to *b2*, and so on.
- If one of these comparisons has not been made, e.g., a1, has not been compared to *b2*, we can transpose these keys to get

$$b_1 < b_2 < a_1 < a_2 < \ldots < b_n < a_n$$

Adversary Arguments – Example 2

- Hence, $2n - 1$ is indeed a lower bound for the number of key comparison needed for any merging algorithm.

Problem Reduction

- To show that problem *P* is at least as hard as another problem *Q* with a known lower bound, we need to reduce *Q* to *P* (not *P* to Q!).
- In other words, we should show that an arbitrary instance of problem *Q* can be transformed (in a reasonably efficient fashion) to an instance of problem *P*, so any algorithm solving *P* would solve *Q* as well. Then a lower bound for *Q* will be a lower bound for *P*.

Important Problems used for Problem Reduction

Problem	Lower Bound	Tightness
sorting (comparison-based)	$\Omega(n \log n)$	yes
searching in a sorted array	$\Omega(\log n)$	yes
element uniqueness	$\Omega(n \log n)$	yes
n-digit integer multiplication	$\Omega(n)$	unknown
multiplication of n-by-n matrices	$\Omega(n^2)$	unknown

Problem Reduction - Example

- Consider the Euclidean minimum spanning tree problem: given *n* points in the Cartesian plane, construct a tree of minimum total length whose vertices are the given points.

Problem Reduction - Example

- As a problem with a known lower bound, we use the element uniqueness problem.
- We can transform any set x_1, x_2, … , x_n of n real numbers into a set of n points in the Cartesian plane by simply adding 0 as the points' y coordinate: $(x_1, 0)$, $(x_2, 0)$, … , $(x_n,0)$.

Problem Reduction - Example

- *Let T* be a minimum spanning tree found for this set of points.
- Since *T* must contain a shortest edge, checking whether *T* contains a zero length edge will answer the question about uniqueness of the given numbers.
- This reduction implies that $\Omega(n \log n)$ is a lower bound for the Euclidean minimum spanning tree problem, too.

Problem Reduction - Example

- The final results about the complexity of many problems are not known and hence the reduction technique is often used to compare the relative complexity of problems.
- The following formulae show that the problems of computing the product of two n-digit integers and squaring an n-digit integer belong to the same complexity class.

$$x \cdot y = \frac{(x+y)^2 - (x-y)^2}{4} \quad \text{and} \quad x^2 = x \cdot x$$

Problem Reduction - Example

- Multiplying two symmetric matrices turns out to be in the same complexity class as multiplying two arbitrary square matrices.
- The former problem is a special case of the latter one, but we also can reduce the problem of multiplying two arbitrary square matrices of order n, say A and B, to the problem of multiplying two symmetric matrices.

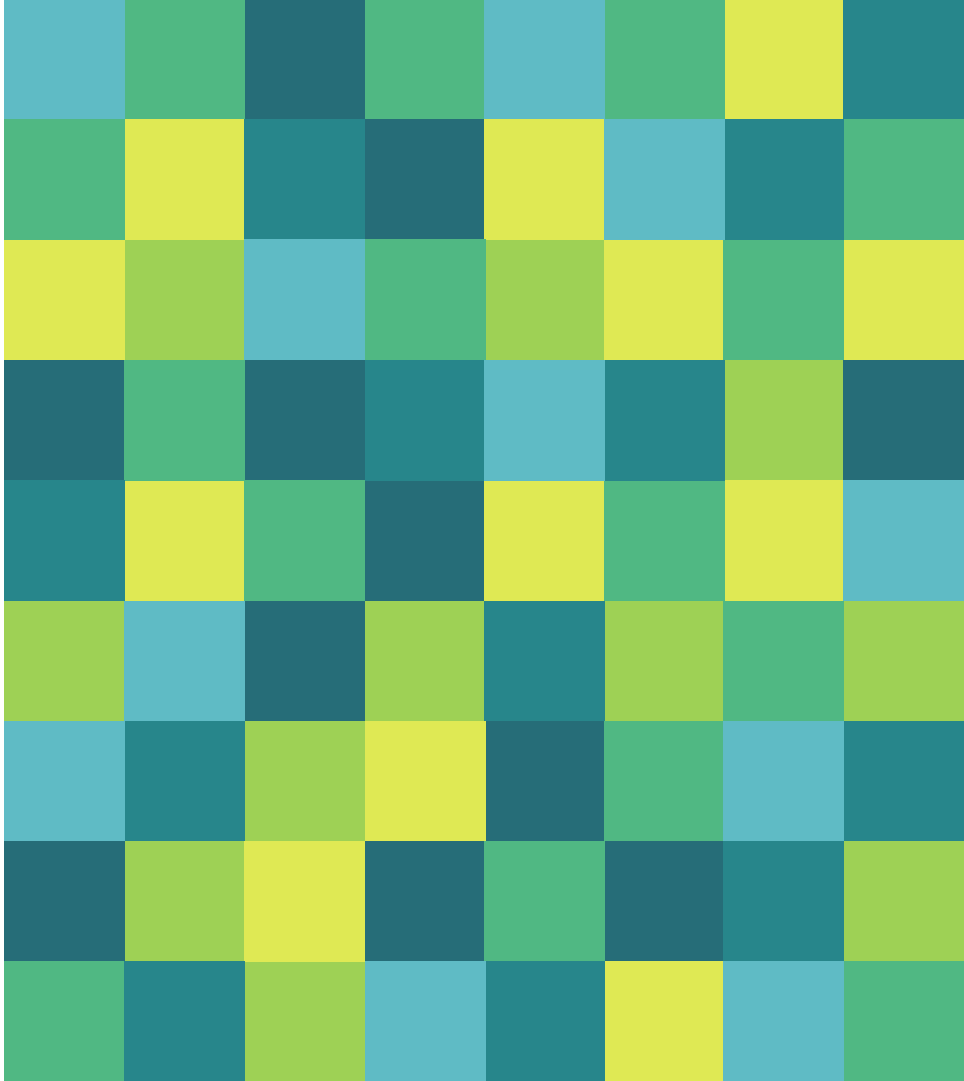
Problem Reduction - Example

$$X = \begin{bmatrix} 0 & A \\ A^T & 0 \end{bmatrix} \text{ and } Y = \begin{bmatrix} 0 & B^T \\ B & 0 \end{bmatrix},$$

$$XY = \begin{bmatrix} 0 & A \\ A^T & 0 \end{bmatrix} \begin{bmatrix} 0 & B^T \\ B & 0 \end{bmatrix} = \begin{bmatrix} AB & 0 \\ 0 & A^T B^T \end{bmatrix},$$

2.

Decision Trees



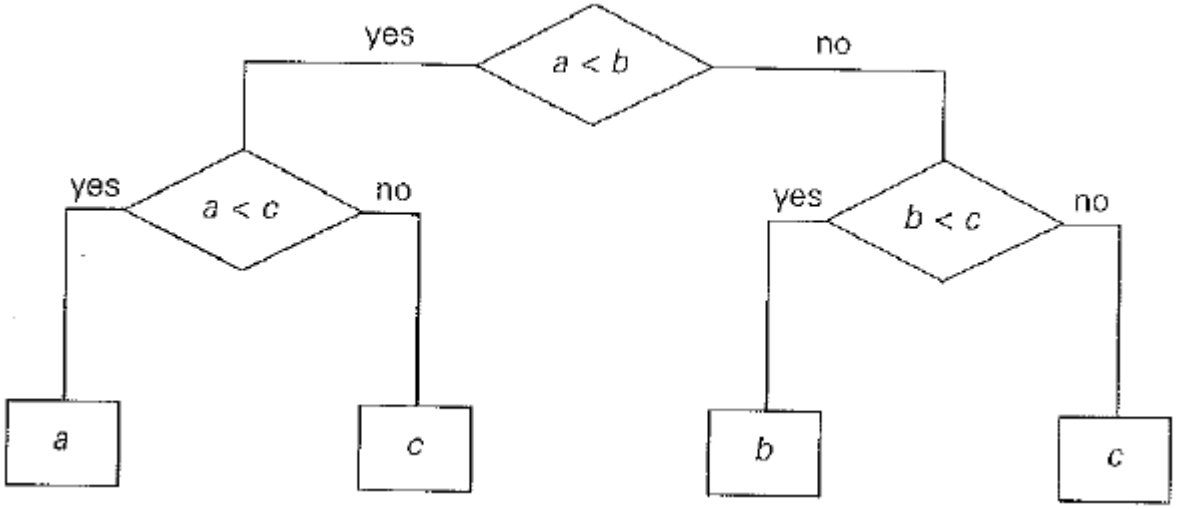
Introduction

- Many important algorithms, especially those for sorting and searching, work by comparing items of their inputs.
- We can study the performance of such algorithms with a device called the ***decision tree***.

Introduction

- Many important algorithms, especially those for sorting and searching, work by comparing items of their inputs.
- We can study the performance of such algorithms with a device called the ***decision tree***.

Decision Tree for finding a minimum of three numbers



Idea

- The central idea behind this model lies in the observation that a tree with a given number of leaves, which is dictated by the number of possible outcomes, has to be tall enough to have that many leaves.
- Specifically, it is not difficult to prove that for any binary tree with leaves and height h,

$$h \geq \lceil \log_2 l \rceil.$$

Idea

- A binary tree of height *h* with the largest number of leaves has all its leaves on the last level.
- Hence, the largest number of leaves in such a tree is 2^h.
- In other words, $2^h >= l$ which implies:

$$h \geq \lceil \log_2 l \rceil.$$

Decision Trees for Sorting Algorithms

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

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.
- The number of possible outcomes for sorting an arbitrary n-element list is equal to $n!$.

Decision Trees for Sorting Algorithms

- The height of a binary decision tree for any comparison-based sorting algorithm and hence the worst -case number of comparisons made by such an algorithm cannot be less than:

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

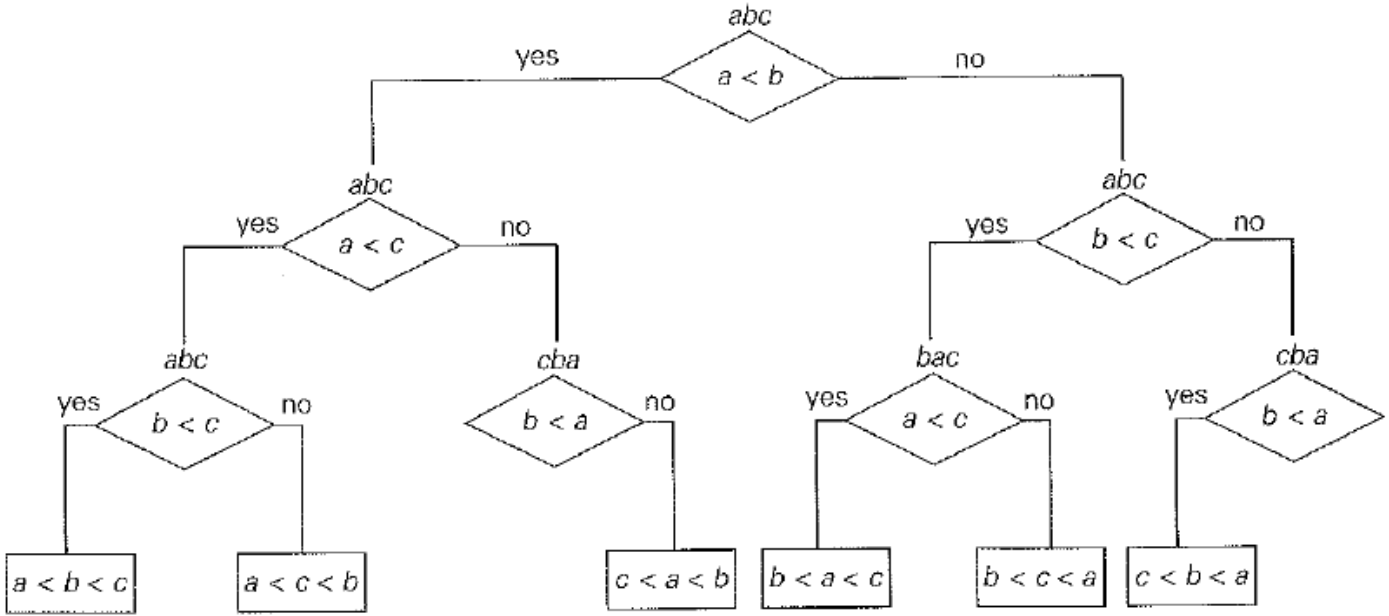
Decision Trees for Sorting Algorithms

- Using Stirling's formula:

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

- About $n \log_2 n$ comparisons are necessary to sort an arbitrary n-element list by any comparison-based sorting algorithm.

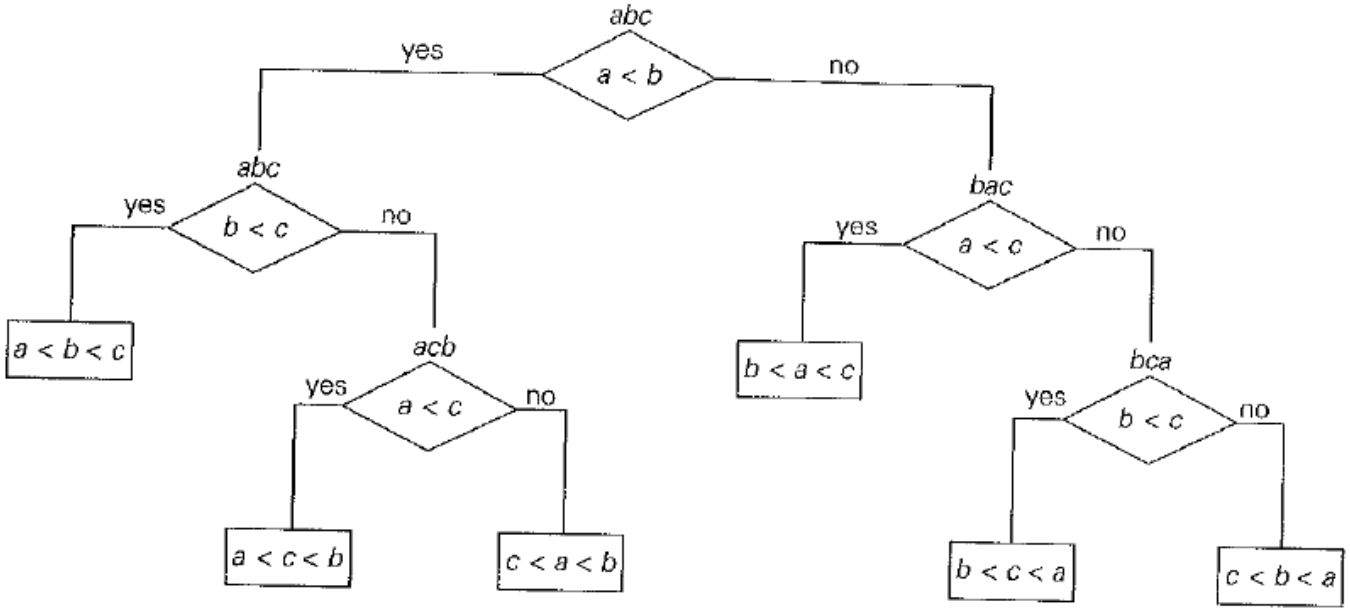
Decision Tree - Three Element Selection Sort



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.

Decision Trees for Sorting Algorithms



Decision Trees for Sorting Algorithms

- For example, for the three-element insertion sort this number is:
- (2 + 3 + 3 + 2 + 3 + 3)/6 = 2(2/3).

Decision Trees for Sorting Algorithms

- 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:

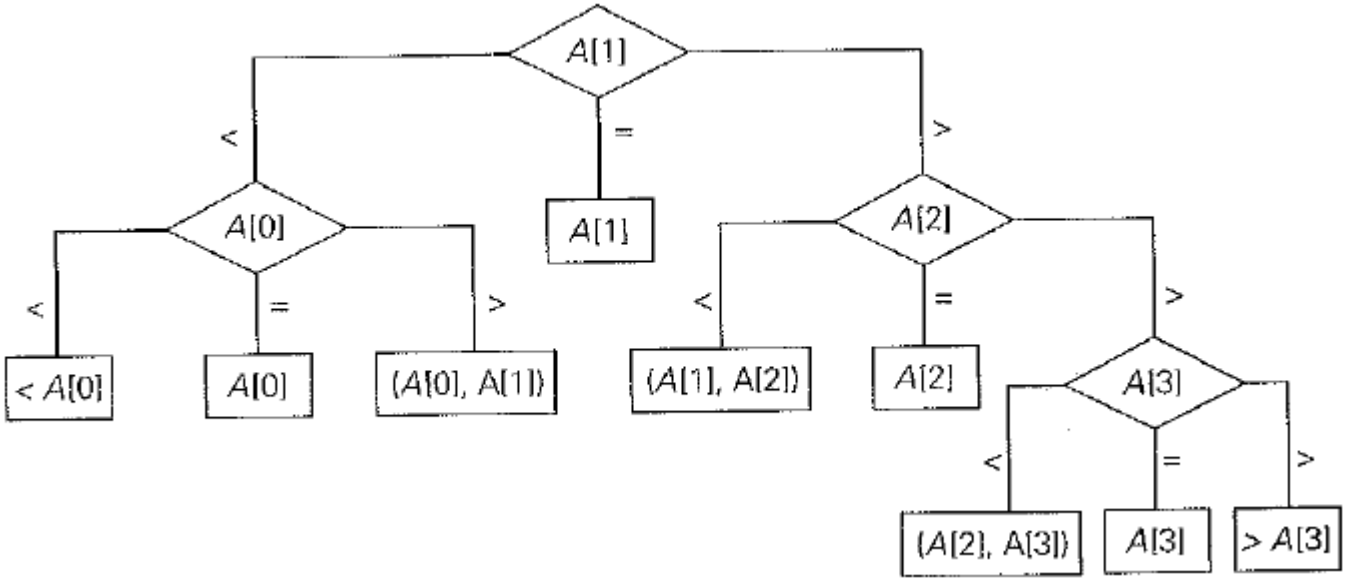
$$C_{avg}(n) \geq \log_2 n!.$$

Decision Trees for Searching Algorithms

- Decision trees can be used for establishing lower bounds on the number of key comparisons in searching a sorted array of *n* keys: *A[O]* < A[1] < ... < *A[n* -1].
- The number of comparisons made by binary search in the worst case:

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

Decision Trees for Searching Algorithms



Decision Trees for Searching Algorithms

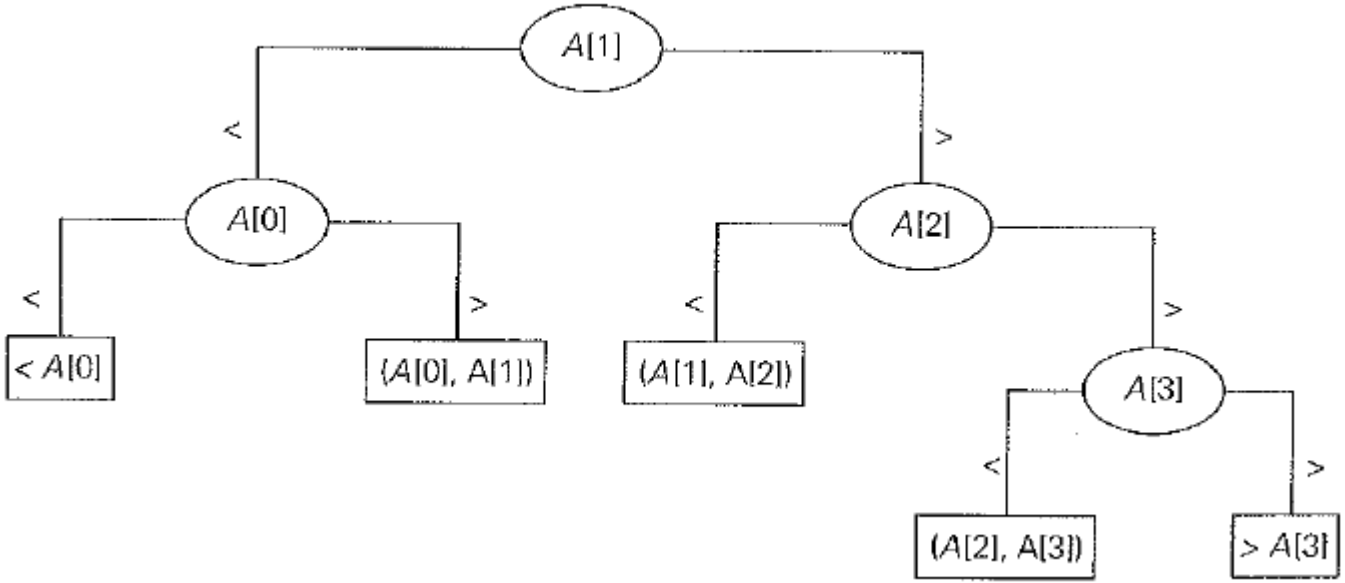
- For an array of *n* elements, all such decision trees will have *2n* +1 leaves (n for successful searches and *n* + 1 for unsuccessful ones).
- Since the minimum height *h* of a ternary tree with *l* leaves is floor($\log_3 l$), we get the following lower bound on the number of worst-case comparisons:

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

Decision Trees for Searching Algorithms

- This lower bound is smaller than ceil($\log_2(n + 1)$), the number of worst-case comparisons for binary search, at least for large values of *n* (and smaller than or equal to ceil($\log_2(n + 1)$) for every positive integer.
- Can we prove a better lower bound, or is binary search far from being optimal?

Decision Trees for Searching Algorithms

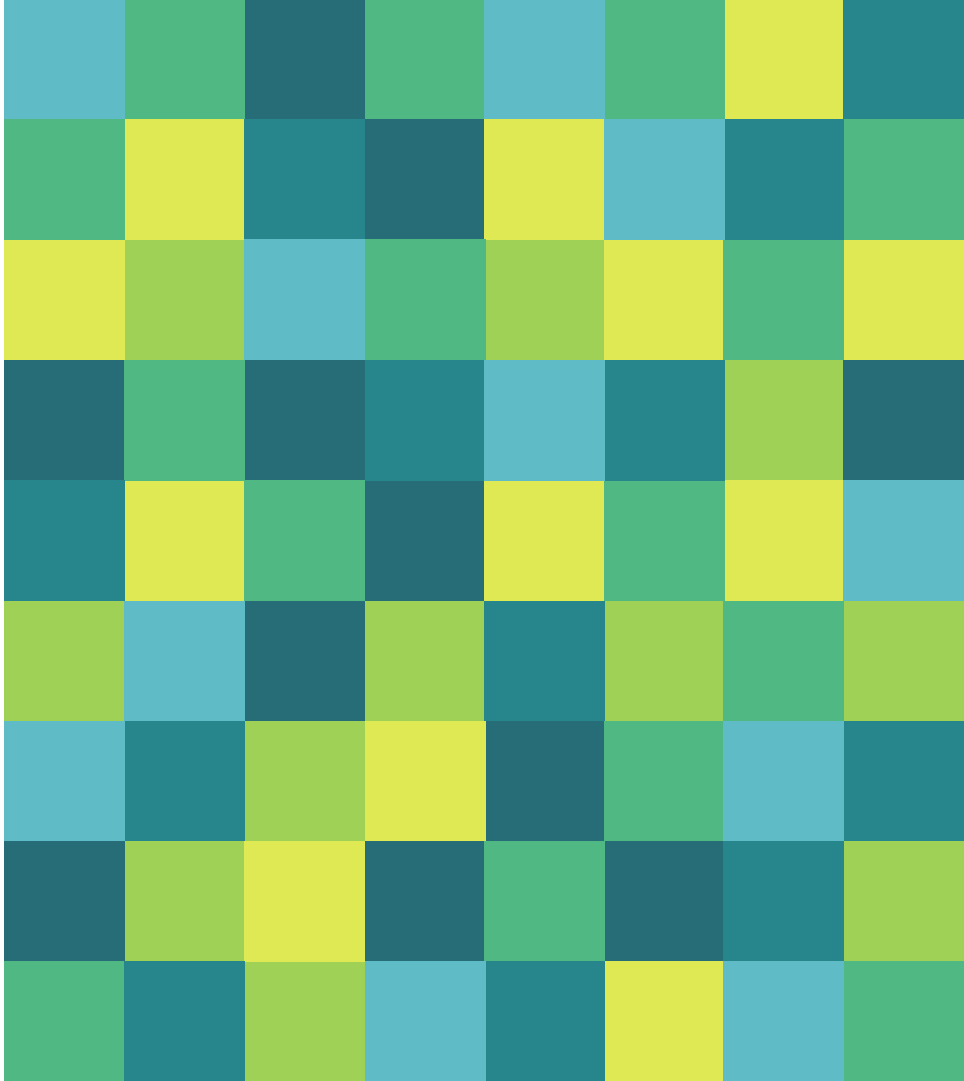


Decision Trees for Searching Algorithms

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

3.

P, NP and NP Complete Problems



DEFINITION 1
Polynomial Time Algorithms

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.

Note: Since, we are using big · oh notation here, problems solvable in, say logarithmic time are solvable in polynomial time as well.

Tractable and Intractable Problems

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

Why are problems that cannot be solved in polynomial time intractable?

- We cannot solve arbitrary instances of such problems in a reasonable amount of time.
- There are very few useful polynomial-time algorithms with the degree of a polynomial higher than three and do not usually involve extremely large coefficients.

Why are problems that cannot be solved in polynomial time intractable?

- Third, polynomial functions possess many convenient properties; in particular, both the sum and composition of two polynomials are always polynomials too.
- Fourth, the choice of this class has led to a development of an extensive theory called ***computational complexity,*** which seeks to classify problems according to their inherent difficulty.

DEFINITION 1
Class P

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.***.

The Restriction of P to Decision Problems

- First, it is sensible to exclude problems not solvable in polynomial time because of their exponentially large output.
- Second, many important problems that are not decision problems in their most natural formulation can be reduced to a series of decision problems that are easier to study.

Undecidable Problems

- Some decision problems cannot be solved at all by any algorithm. Such problems are called *undecidable.*
- *Example: Halting Problem given by Alan Turing in 1936.*

Best known problems with no polynomial – time algorithm

- Hamiltonian Circuit
- Traveling Salesman
- Knapsack Problem
- Partition Problem
- Bin Packing
- Graph Coloring
- Integer Linear Programming

DEFINITION
Non – Deterministic Algorithm

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*.

DEFINITION
Non – Deterministic Polynomial

A nondeterministic algorithm is said to be ***nondeterministic polynomial*** if the time efficiency of its verification stage is polynomial

DEFINITION
Class NP

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

DEFINITION
Class NP

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

Class NP

- Most decision problems are in *NP*. First of all, this class includes all the problems in *P*:

$$P \subseteq NP.$$

DEFINITION
Class NP

- 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.

DEFINITION
Class NP

- But *NP* also contains the Hamiltonian circuit problem, the partition problem, as well as decision versions of the traveling salesman, the knapsack, graph coloring and many hundreds of other difficult combinatorial optimization problems.
- The halting problem, on the other hand, is among the rare examples of decision problems that are known not to be in *NP.*

QUESTION

This leads to the most important open question of theoretical computer science:

"Is *P* a proper subset of *NP*, or are these two classes, in fact, the same?

We can put this symbolically as

$$P \stackrel{?}{=} NP.$$

POLYNOMIAL TIME REDUCIBILITY

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:

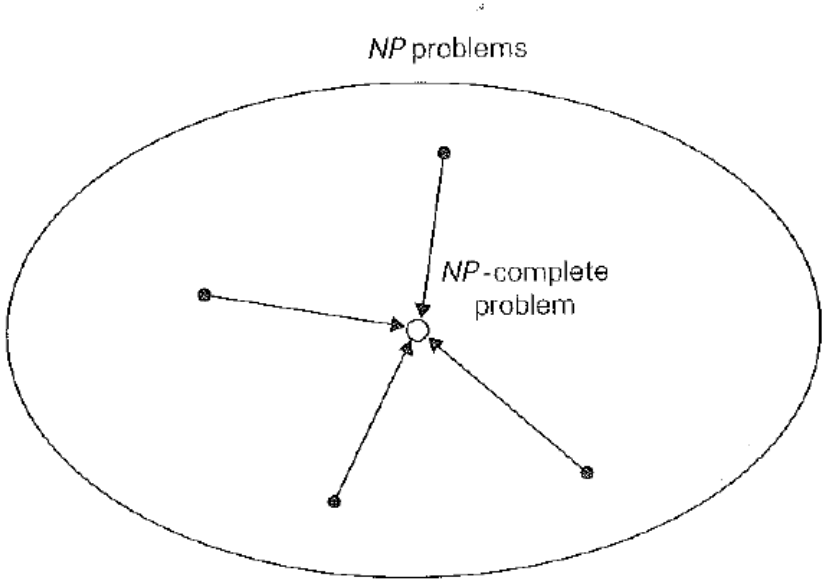
- t maps all yes instances of D_1 to yes instances of D_2 and all no instances of D1 to no instances of D2
- t is computable by a polynomial-time algorithm.

NP COMPLETE PROBLEM

A decision problem *D is* said to be *NP-complete* if:

- It belongs to class *NP*
- Every problem in *NP* is polynomially reducible to *D*

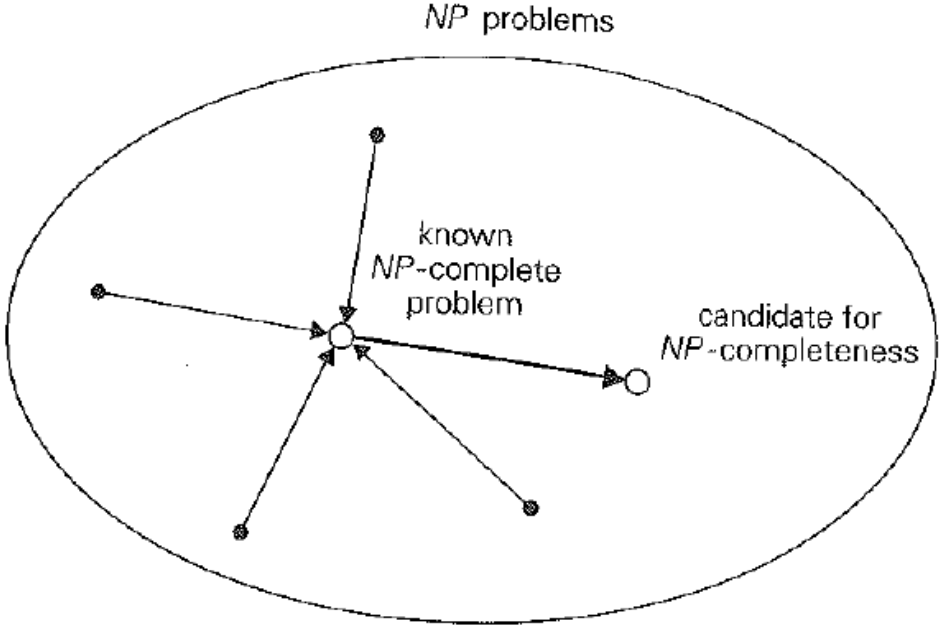
NP COMPLETE PROBLEM



NP – COMPLETE PROBLEMS

- The second step is to show that every problem in *NP* is reducible to the problem in question in polynomial time.
- Because of the transitivity of polynomial reduction, this step can be done by showing that a known NP-complete problem can be transformed to the problem in question in polynomial time.

NP – COMPLETE PROBLEMS



NP – COMPLETE PROBLEMS

- The definition of NP-completeness immediately implies that if there exists a deterministic polynomial-time algorithm for just one NP-complete problem, then every problem in *N P* can be solved in polynomial time by a deterministic algorithm, and hence *P = NP.*
- Such implications make most computer scientists believe that *P is not equal to NP*, although nobody has been successful so far in finding a mathematical proof of this intriguing conjecture.