Module 5

11. Limitations of Algorithmic Power:

- We have encountered dozens of algorithms for solving a variety of different problems. But the power of algorithms is not unlimited, it's limited.

1) Some problems cannot be solved by any algorithm - No solution.

2) Some problems can be solved algorithmically but in non-polynomial time.

3) Some problems can be solved in Polynomial time.

- There are three bounds of an algorithm:
  1. Lower bound - minimum amount of time
  2. Upper bound - maximum amount of time
  3. Tight bound

24/7/24

11.2 Decision Trees

- Performance of algorithms like searching and sorting, which work by comparing items of their inputs, can be studied with a device called Decision Tree.
- A decision tree is a flowchart like structure used to make decisions or predictions.

Structure of a Decision tree :

- Root node : Represents initial decision to be made.
- Internal nodes: Represents key comparisons. Eg: $K < K'$
- Branches : Represents the outcome of a decision, leading to another node
  - 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 $K > K'$.
- Leaf nodes : Represents the final decision or prediction no further splits occur at these nodes.
  - Each leaf represents a possible outcome of the algorithm 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 number of leaves must be at least as large as the number of possible outcomes.
- 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, & the number of comparisons made by the algorithm on such a run is equal to the length of this path.

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

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

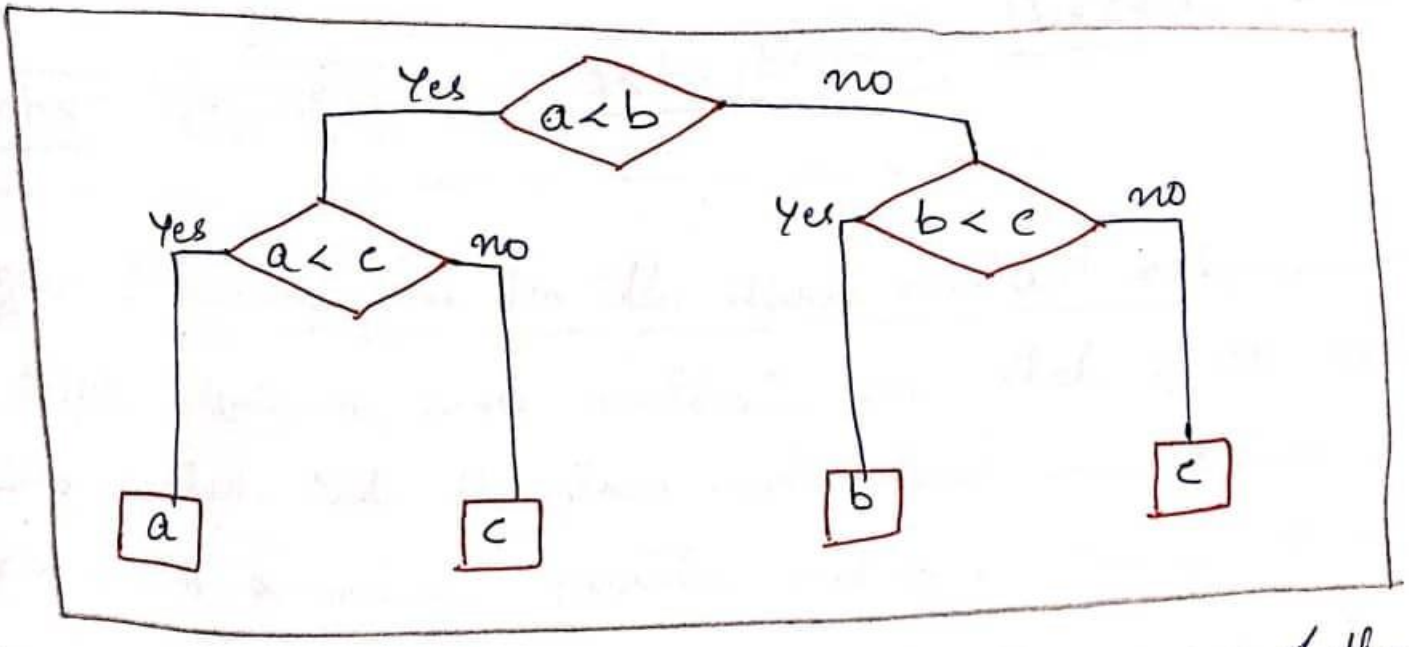


Figure: Decision tree for finding a minimum of three numbers

Decision trees for sorting:

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

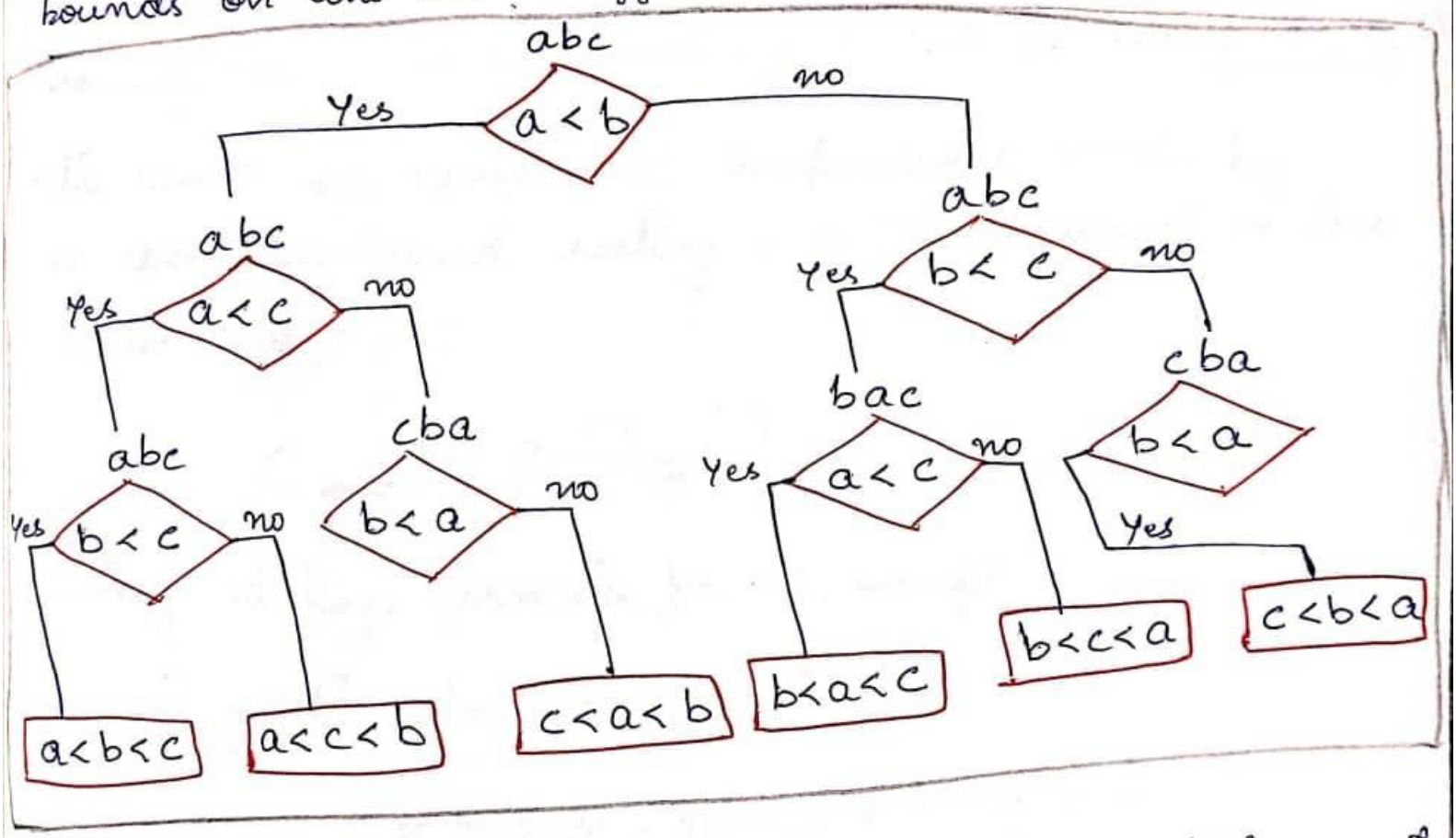


Figure: Decision tree for the three-element selection sort. (A triple above a node indicates the state of the array being sorted. Note the two redundant comparisons b<a with a single possible outcome because of the results of some previously made comparisons.)

- 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 eg. A three-element list a, b, c of orderable items such as real numbers or strings. For the outcome $a < c < b$ obtained by sorting this list, the permutation in question is 1, 3, 2.
- In general, the number of possible outcomes for sorting an arbitrary n-element list is equal to $n!$.
- The worst-case number of comparisons made by comparisons-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

$$\lceil \log_2 n! \rceil \approx \log_2 \sqrt{2\pi n}\,(n/e)^n$$

$$= n \log_2 n - n \log_2 e + \frac{\log_2 n}{2} + \frac{\log_2 2\pi}{2}$$

$$\approx n \log_2 n$$

- $n \log_2 n$ comparisons are necessary in the worst case to sor an arbitrary n-element list by any comparisons-based sorting algorithm.

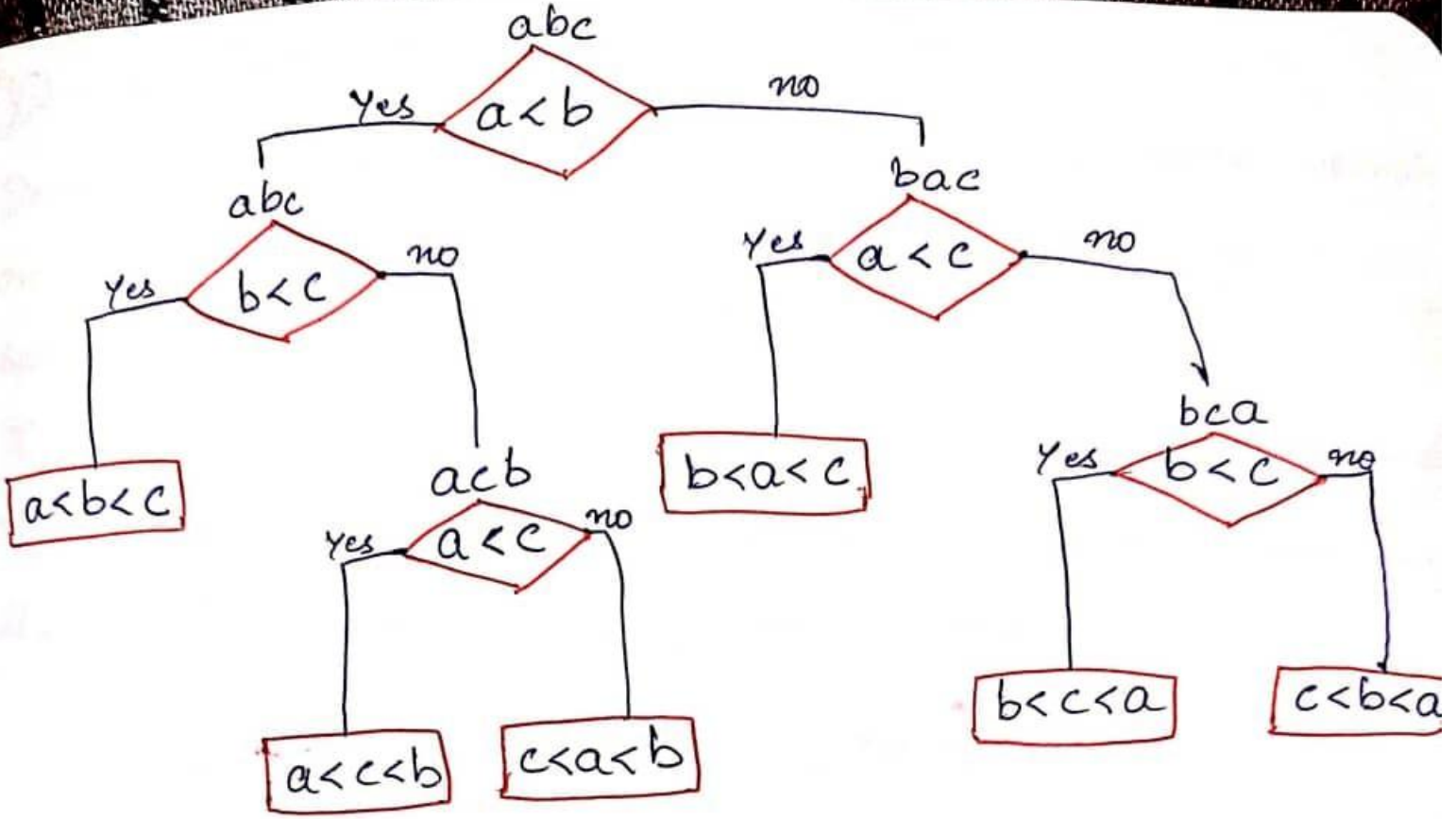


Figure : Decision tree for three-element insertion sort

- Decision trees are also used for analyzing the average-case efficiencies of comparison-based sorting algorithms. 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.

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

- The lower bounds for the average & worst cases are almost identical.

Decision trees for Searching a Sorted array:

- Decision trees can be used for establishing lower bounds on the number of key comparisons in searching a sorted array of n keys: $A[0] < A[1] < \dots < A[n-1]$.
- 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

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

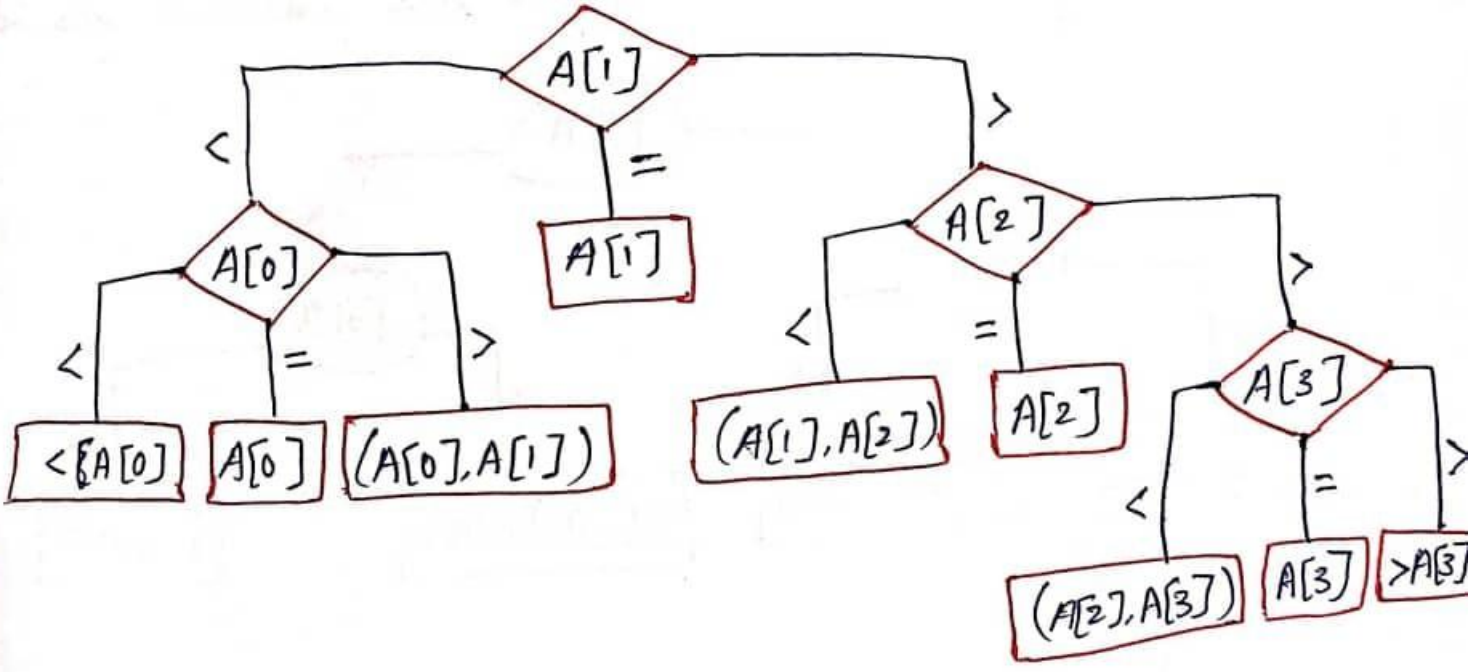


Figure: Ternary decision tree for binary search in a four-element array.

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

Search key K is compared with some element $A[i]$ to see whether $K < A[i]$, $K = A[i]$ or $K > A[i]$ using ternary decision trees.

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

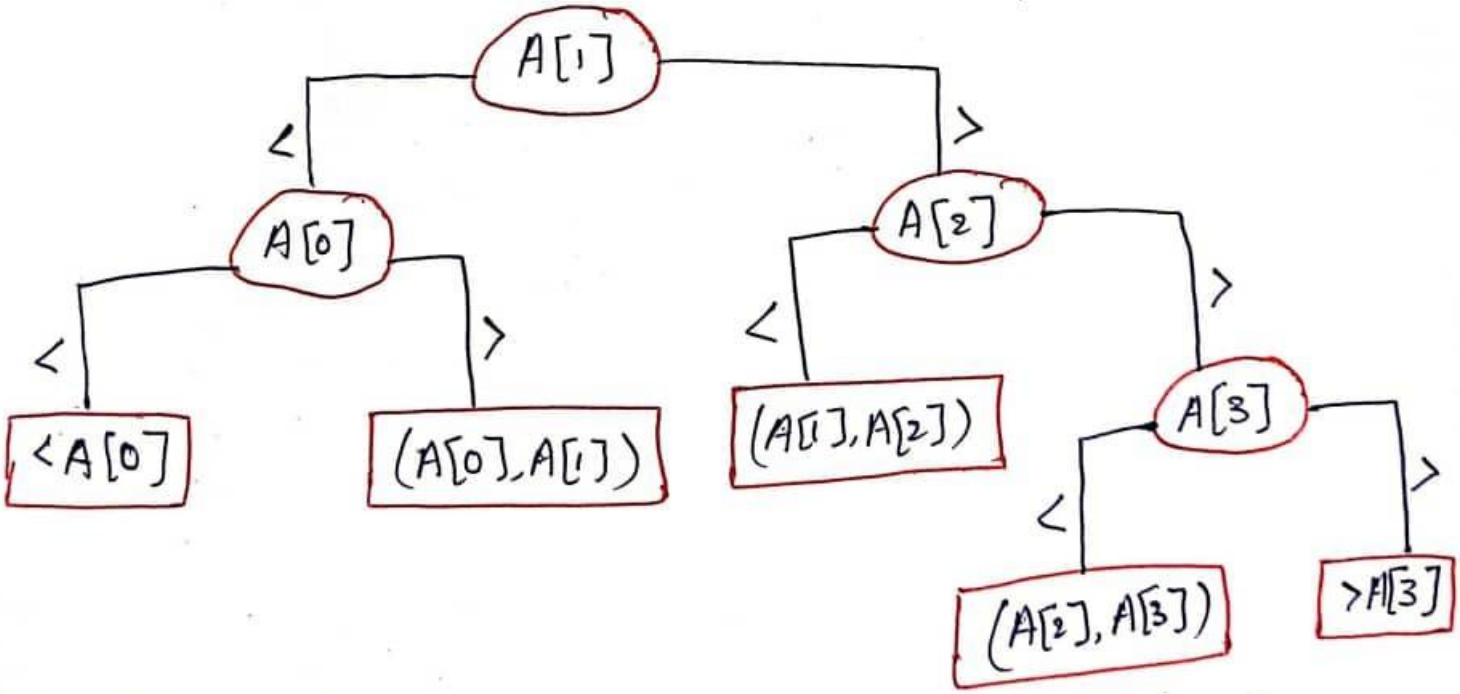


Figure: Binary decision tree for binary search in a four-element array.

- Internal nodes in Binary decision tree 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, & 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.
- Binary decision tree yields $C_{worst}(n) \geq \lceil \log_2(n+1) \rceil$, this lower bound is smaller than $\lceil \log_3(2n+1) \rceil$.
- The average number of comparisons made is about $\log_2 n - 1$ & $\log_2(n+1)$ for successful & unsuccessful searches respectively.

11.3 P, NP, and NP-complete Problems

A computational problem specifies an input-output relationship.

- Problem: We need to solve a computational problem.

Eg: Input - length in cm
Output - length in m

- Algorithm: specifies how to solve a problem

Eg: 1. Read cm
2. Calculate m = cm * 0.01
3. Print m

- Program: A computer executable description of an algorithm

Eg:
```
main()
{
    float cm, m;
    scanf("%f", &cm);
    m = cm * 0.01;
    Printf("%f", m);
}
```

- Types of Problems:

1) Tractable - Problems that can be solved in polynomial time
2) Intractable - Problems that cannot be solved in polynomial time
3) Decision - Problems that tries to answer a Yes/no question
4) Optimization - Problems that tries to find an optimal solution.

11.3 P, NP, and NP-complete Problems

- A computational problem specifies an input-output relationship.
- Problem: We need to solve a computational problem.

Eg: Input - length in cm
Output - length in m

- Algorithm: specifies how to solve a problem

Eg: 1. Read cm
2. Calculate m = cm * 0.01
3. Print m

- Program: A computer executable description of an algorithm

Eg:
```
main()
{
    float cm, m;
    scanf("%f", &cm);
    m = cm * 0.01;
    Printf("%f", m);
}
```

- Types of Problems:

1) Tractable - Problems that can be solved in polynomial time
2) Intractable - Problems that cannot be solved in polynomial time
3) Decision - Problems that tries to answer a Yes/no question
4) Optimization - Problems that tries to find an optimal solution.

· In computer science, there exist some problems whose solutions are not yet found, the problems are divided into classes known as complexity classes.

· A complexity class is a set of problems with related complexity. These classes help scientists to group problem based on how much time & space they require to solve problems & verify the solutions.

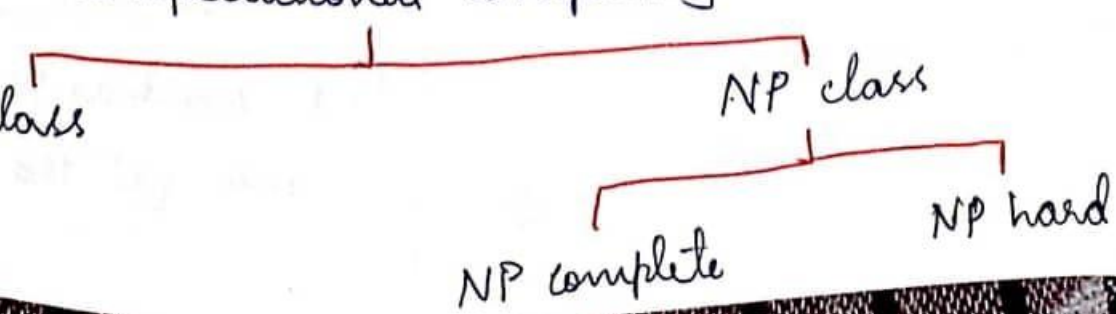
· Computational complexity - The amount of resources required to run an algorithm. The common resources are time and space.

· The time complexity of an algorithm is used to describe how much time the algorithm takes to solve a problem, but it can also be used to describe how long it takes to verify the answer.

· The space complexity of an algorithm describes how much memory is required for the algorithm to operate.

· Complexity classes are useful in organising similar types of problems.

· Depending on computational complexity of various problem the problems can be classified into different groups.



Polynomial time

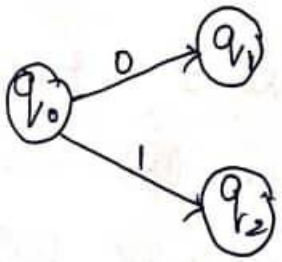
- Linear search - n
- Binary search - $\log n$
- Insertion sort - n^2
- Merge sort - $n \log n$
- Matrix multiplication - n^3

Exponential time

- 0/1 Knapsack - 2^n
- Travelling Salesman - 2^n
- Sum of subsets - 2^n
- Graph colouring - 2^n
- Hamilton cycle - 2^n

Deterministic Algorithm

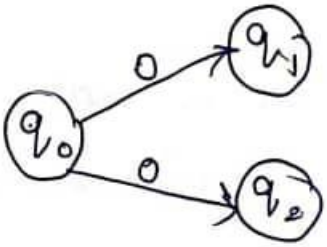
- Produces only a single output for the same input even on different runs.



- Can determine what is the next step.
- Can solve problem in Polynomial time

Non deterministic Algorithm

- Can provide different outputs for the same input on different runs.



- Cannot determine
- Cannot solve the problem in polynomial time

Undecidable problems

- Some decision problems cant be solved at all by an algorithm.

Decidable Problems

- Problems that can be solved by an algorithm.

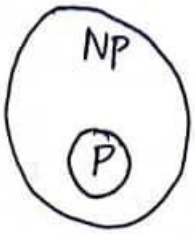
Halting Problem: Given a computer program & an input to it, determine whether the program will halt on that input or continue working indefinitely on it.

P class:

- The set of decision problems that can be solved in polynomial time using deterministic algorithms are called P class problems / Polynomial.
- The P class problems can be solved & verified in polynomial time.
- The P class problems are solvable & tractable.

NP class:

- The set of decision problems that can be solved in polynomial time using non-deterministic algorithms are called NP class problems.
- The NP class problems can be verified in polynomial time
- The solutions of the NP class are hard to find, but the solutions are easy to verify.
- Problems of NP can be verified by a Turing machine in polynomial time.
- $P \subseteq NP$



NP-hard class:

- A problem is NP-hard if all other problems in NP can be polynomially reduced to NP-hard.
- All NP-hard problems are not in NP.
- If a solution for an NP-hard problem is given then it takes a long time to check whether it is right or not
- Problem A is in NP-hard if, for every problem B in NP, there exists a polynomial-time reduction from B to A.

NP-Complete class:

- A problem is NP-Complete if it is both NP & NP hard
- Any problem in NP class can be transformed or reduced into NP-complete problems in polynomial time
- If one could solve an NP-complete problem in polynomial time, then one could also solve any NP problem in polynomial time.

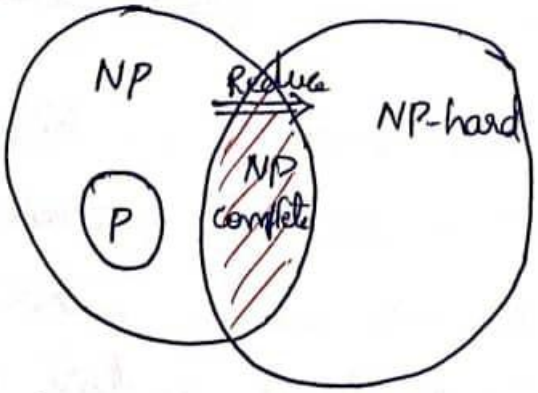
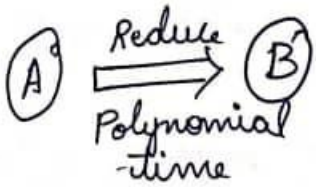


Figure - Relationship among P, NP, NP-hard, NP-complete

Reduction:

Let A & B are two problems.

Then problem 'A' reduces to problem 'B' iff there is a way to solve problem 'A' by deterministic algorithm the solve problem 'B' in polynomial time



- If A is reducible to B we denote it by $A \propto B$

- Properties of Reduction

- if $A \propto B$ & B in P ⟶ then A in P
- if $A \propto B$ & A is not in P ⟶ then B is not in P
- if A is decision problem & B is optimization problem ⟶ then it is possible $A \propto B$
- if $A \propto B$ & $B \propto C$ ⟶ then $A \propto C$ (transitive)
- if $A \propto B$ & $B \propto A$ ⟶ then A, B are polynomially equivalent
- if (satisfiability $\propto A$) ⟶ then A is NP-hard

- Satisfiability (SAT): The problem of determining if a formula is satisfiable or unsatisfiable.
- A formula is said to be satisfiable if variables can be assigned with values such that the formula turns out to be TRUE.

. A formula is said to be unsatisfiable if variables cannot be assigned with values such that the formula turns out to be TRUE.

(Ex): Consider a boolean formula

$$(x_1 \vee x_2 \vee \bar{x}_3) \wedge (\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_3) \wedge (x_1 \vee \bar{x}_2)$$

let $x_1 = 1,\ x_2 = 0,\ x_3 = 0$

Result of formula $= (1 \vee 0 \vee 1) \wedge (0 \vee 1 \vee 1) \wedge (1 \vee 1)$

$= 1 \wedge 1 \wedge 1$

$= 1$ (TRUE) $\therefore$ Satisfiable

(Ex): Non-deterministic Algorithm for satisfiability

```
Algorithm Eval (E, n)
{
    for i=1 to n do
        x_i = Choice (false, true);
    if E (x_1 ... x_n) then Success();
    else Failure();
}
```

(12) Coping with limitations of Algorithmic Power:

- There are problems that are difficult to solve algorithmically. There are two algorithm design techniques backtracking & branch-and-bound - that often make it possible to solve at least some large instances of difficult combinatorial problems. Both strategies are an improvement over exhaustive search.
- Both backtracking & branch-and-bound are based on the construction of a state-space tree
- State-space tree:
- constructing a tree of choices being made, i.e nodes reflect specific choices made for a solution's components
- Root - represents initial state before the search for a solution begins.
- All state nodes are either promising node or non-Promising node.
- Promising node - leads to complete solution
- Non promising node - dead end / no solution / doesn't lead to complete solution
- Leaves - either promising or non-promising nodes.
- State space tree is constructed using DFS (Depth First Search).

Backtracking - a method to solve problems by making a series of choices that we can return or backtrack to.

3 Keys - ① Our choice - what choice do we make at each call of the function?

② Our constraints - when do we stop following a certain path? when do we not even go one way?

③ Our goal - What's our target? What are we trying to find? (Base case comes from this)

- Algorithm Backtrack (X[1...i])

// Gives a template of a generic backtracking algorithm

// Input : X[1...i] specifies first i promising components of a solution

// Output : All the tuples representing the problems solution

if X[1...i] is a solution write X[1..i]

else

for each element $x \in S_{i+1}$ consistent with X[1..i] & the constraints do

X[i+1] ← x

Backtrack(X[1..i+1])

12.1 Backtracking

- The principal idea of backtracking is to construct solutions one component at a time & evaluate such partially constructed candidates as follows.
- If a partially constructed solution can be developed further without violating the problem's constraints it is done by taking the 1st remaining legitimate option for the next component.
- If there is no legitimate option for the next component no alternatives for any remaining component need to be considered.
- In this case, the algorithm backtracks to replace the last component of the partially constructed solution with its next option.
- A state space tree for a backtracking algorithm is constructed by DFS manner.
- If the current node is promising, the tree builds further
- If the current node is nonpromising, the algorithm backtracks to the parent node to consider next possible option, if no option available, it backtracks one more level up the tree & so on.
- Finally if the algorithm reaches a complete

solution to the problems, it either stops (if just one solution is required) or continues searching for other possible solutions.

1. n-Queens Problem:

- The problem is to place n queens on an $n \times n$ chessboard so that no two queens attack each other by being in the same row or in the same column or on the same diagonal.
- For $n=1$, the problem has a trivial solution, & for $n=2$ & $n=3$ there is no solution.
- So lets considers the four-queens problem & solve it by the backtracking Technique.
- Since each of the four queens has to be placed in its own row, assign a column for each queen on the board as the figure.

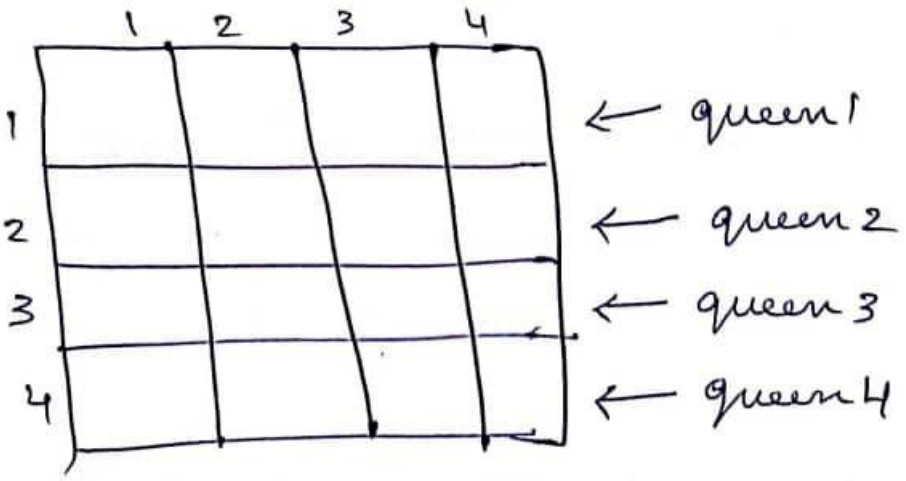


Figure: Board for the four-queens problem

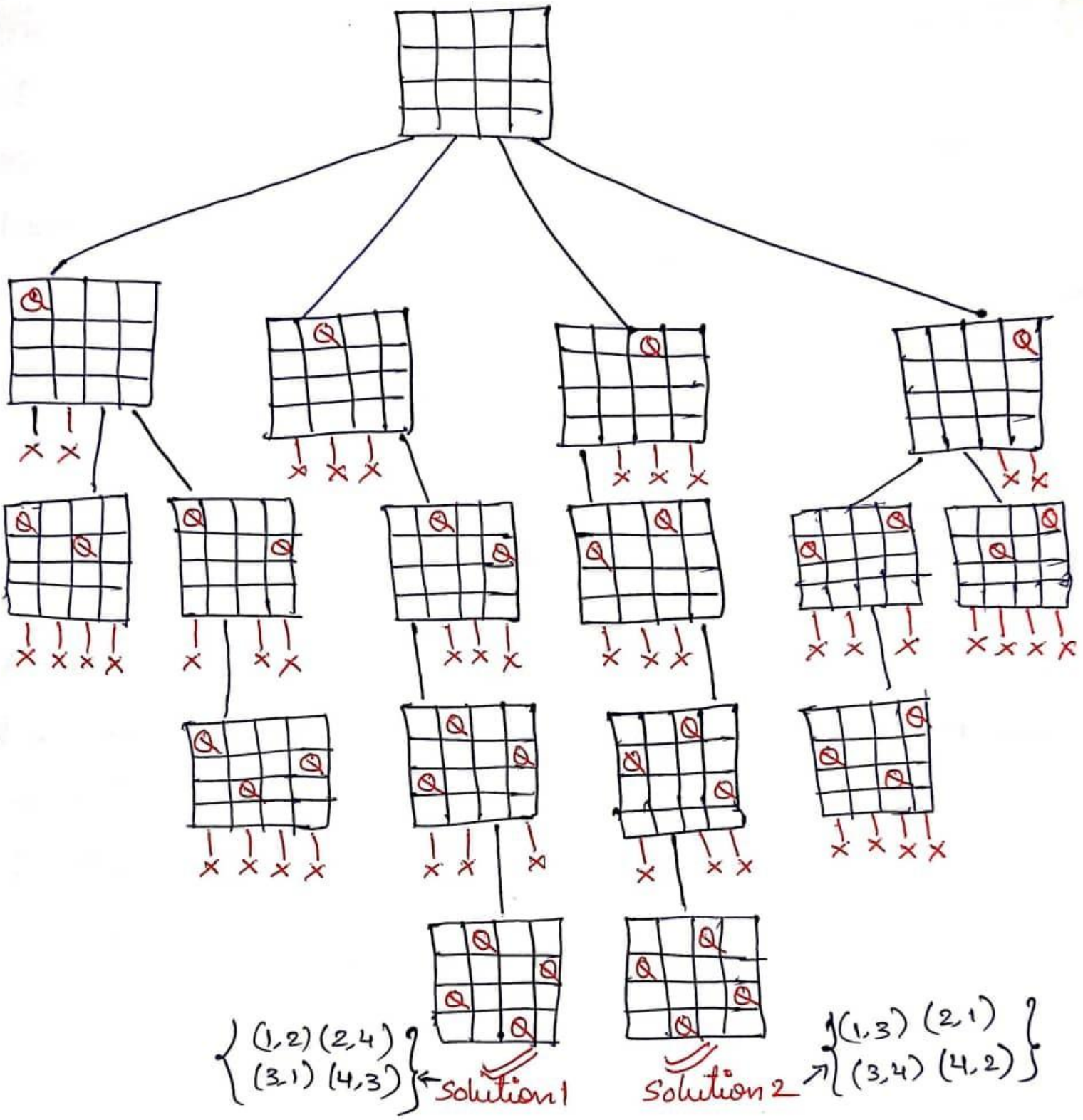


Figure : State space tree solving 4-queens problem by backtracking (× denotes an unsuccessful attempt to place a queen in the indicated column)

We start with an empty board & then place 1st queen in the 1st possible position of its row (1,1).

. Then place queen 2, columns 1 & 2 are unsuccessful, place in (2,3).

. Then placing queen 3 is deadend, the algorithm backtracks & queen 2 in next position (2,4).

· Then queen 3 is placed in (3,2), next its deadend. the algorithm backtracks all the way to queen 1 & moves it to (1,2).

· The queen 2 goes to (2,4), queen 3 to (3,1) & queen 4 to (4,3), which is a solution to the problem.

. Algorithm n-queens (K, n)
(K: initial value; n: no of queens)

// using backtracking the algorithm prints all possible solutions

```
for i ← 1 to n do
    if (Place(K, i))
        x[K] ← i
        if K == n
            print- x[1...n]
        else
            n-queens (K+1, n)
```

Algorithm place(k, i)

// returns True if Q is placed at k^{th} row & i^{th} column otherwise returns false

```
for j ← 1 to (k-1) do
    if (x[j] == i) or (abs(x[j]-k) == abs(j-k))
        return False
return True
```

Row attack, column attack, diagonal attack:

	1	2	3	4
x	3	1	4	2

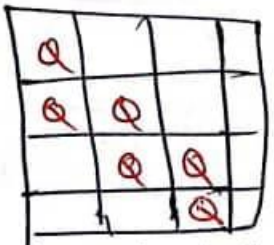
Solution 2

	1	2	3	4
x	2	4	1	3

Solution 1

n-queens array representation

- index of the array x – row
- value of the array x – column
- Its challenging to denote diagonal attack.



(1,1) (2,2) (3,3) (4,4)
(2,1) (3,2) (4,3)

Lets consider any 2 points

(3,2) → (i, j)
(4,3) → (k, l)

$i - j = k - l$
$3 - 2 = 4 - 3$
$1 = 1$
$j - l = i - k$ → $abs(j-l) = abs(k-l)$

(2,4) (3,3) (4,2)

Lets consider any 2 points

(3,3) → (i, j)
(4,2) → (k, l)

$i + j = k + l$
$3 + 3 = 4 + 2$
$6 = 6$
$j - l = k - i$ → $abs(j-l) = abs(k-l)$

2. Subset-Sum Problem :

Find a subset of a given set $A = \{a_1, \ldots a_n\}$ of n positive integers whose sum is equal to a given positive integer d. For ex, for $A = \{1, 2, 5, 6, 8\}$ & $d = 9$, there are 2 solutions: $\{1, 2, 6\}$ & $\{1, 8\}$. Some instances of this problem may have no solutions.

Sort the set's elements in increasing order $a_1 < a_2 < \ldots < a_n$

Ex: $A = \{3, 5, 6, 7\}$ and $d = 15$.

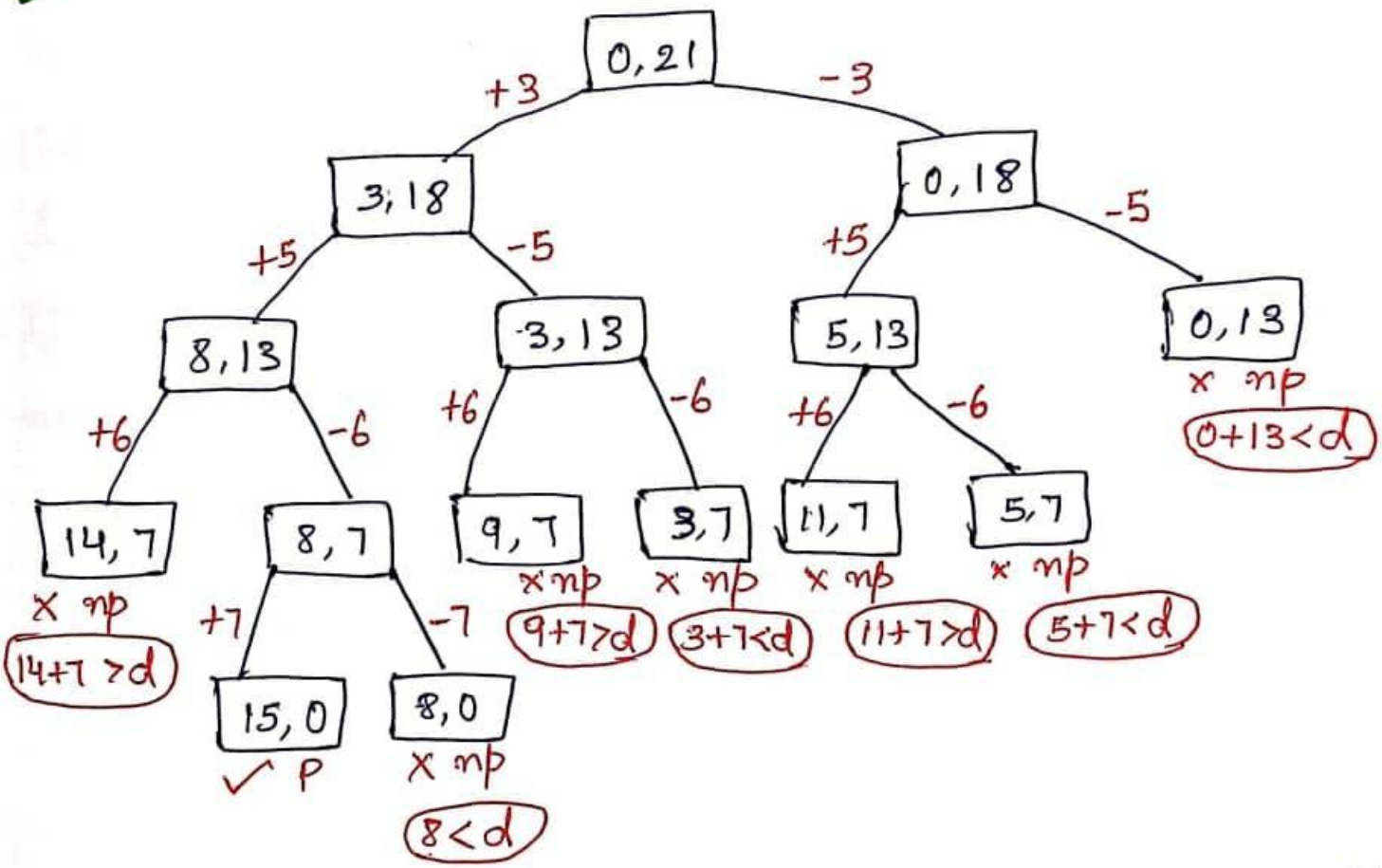


Figure : Complete state space tree of Subset-Sum problem using backtracking.

The root of the tree represents the starting point, with no decisions about the given elements made as yet. Its left & right children represent, respectively, inclusion &

exclusion of a_i in a set being sought.

- A path from the root to a node on the i^{th} level of the tree indicates which of the first i numbers have been included in the subsets represented by that node.
- We record the value of s, the sum of these numbers, in the node.
- If s is equal to d, $(s == d)$ we have a solution to the problem.
- We can either report this result & stop or, if all the solutions need to be found, continue by backtracking to the node's parent.
- If s is not equal to d, $(s \neq d)$, we can terminate the node as nonpromising if either of the following two inequalities holds:

$$s + a_{i+1} > d \quad \text{(the sum is too large)}$$

$$s + \sum_{j=i+1}^{n} a_j < d \quad \text{(the sum is too small)}$$

initial sum — index of element — total sum

Algorithm Subset_sum (s, K, γ)

// Input : $W[1 \ldots n]$ increasing order & d

// output : Subsets whose summation is d

$x[k] = 1$

if $(W[k] + s == d)$

```
        write (x[1...n])
else if (s + w[k] + w[k+1] <= d)
        Subset_sum (s+w[k], k+1, r-w[k])
if (s + r - w[k] >=d) and (s + w[k+1] < d)
        x[k] = 0
        Subset_sum (s, k+1, r-w[k])
```

Backtracking

1. Constructs state-space tree
2. Finds all feasible solutions
3. Uses DFS for construction of state space tree
4. No bounding function

Branch-and-Bound

1. Constructs state-space tree
2. Finds optimal solution
3. Uses DFS & BFS for construction of state space tree
4. Uses bounding function (upper bound or lower bound)

Eg: 1. TSP (Travelling Salesman Problem) - lower bound
2. Knapsack - upper bound

12.2 Branch-and-Bound

• Branch & Bound (BB, B&B) is a method for solving optimization problems by breaking them down into smaller sub-problems & using a bounding function to eliminate sub problems that cannot contain the optimal solution.

We terminate a search path at the current node in a state-space tree of a B&B algorithm for any one of the 3 reasons:

1.) The value of the node's bound is better than the value of the best solution seen so far.

2.) The node represents no feasible solutions because the constraints of the problem are already violated.

3.) The subset of feasible solutions represented by the node consists of a single point (& hence no further choices can be made), we compare the value of the objective function for this feasible solution with that of the best solution seen so far & update the latter with the former if the new solution is better.

Knapsack Problem (using Brand & Bound)

Given n items of known weights w_i & values v_i, $i = 1, 2, \ldots n$, & a knapsack of capacity W, find the most valuable subset of the items that fit in the knapsack.

- Order the items of a given instance in descending order by their value-to-weight ratios.

$$\frac{v_1}{w_1} \geq \frac{v_2}{w_2} \geq \ldots \geq \frac{v_n}{w_n}$$

- The 1st item gives the best payoff per weight unit & the last one gives the worst payoff per weight unit.
- Construct the state-space tree for this as binary tree.
- Each node on the i^{th} level of this tree, $0 \leq i \leq n$, represents all the subsets of n items that include a particular selection made from the first i ordered items.
- This particular selection is uniquely determined by the path from the root to the node: a branch going to the left indicates the inclusion of the next item, & a branch going right indicates its exclusion.
- Record the total weight w & the total value v of this selection in the node, along with some upper

bound ub on the value of any subset that can be obtained by adding zero or more items to this selection

- Compute the upper bound ub:

$$ub = v + (W - w)(v_{i+1}/w_{i+1})$$

Eg:

item	weight (w)	value (v)	value/weight (v/w)
1	4	$40	10 ⇒ (40/4) v_1/w_1
2	7	$42	6 ⇒ (42/7) v_2/w_2
3	5	$25	5 ⇒ (25/5) v_3/w_3
4	3	$12	4 ⇒ (12/3) v_4/w_4

W = 10 (Knapsack capacity)

- Calculate upperbound (ub):

- Initially, v = 0, w = 0, v_1/w_1

$$ub = 0 + (10 - 0) \times 10 \Rightarrow ub = 0 + (10 - 0)(v_1/w_1)$$

$$= 0 + 10 \times 10$$

$$ub = 100$$

- w = 4, v = 40, v_2/w_2 (with 1)

$$ub = 40 + (10 - 4) \times 6 \Rightarrow ub = 40 + (10 - 4)(v_2/w_2)$$

$$= 40 + 6 \times 6$$

$$ub = 76$$

- w = 0, v = 0, v_2/w_2 (without 1)

$$ub = 0 + (10 - 0) \times (v_2/w_2) = 0 + (10 - 0) \times 6 = 10 \times 6 = 60$$

$$ub = 60$$

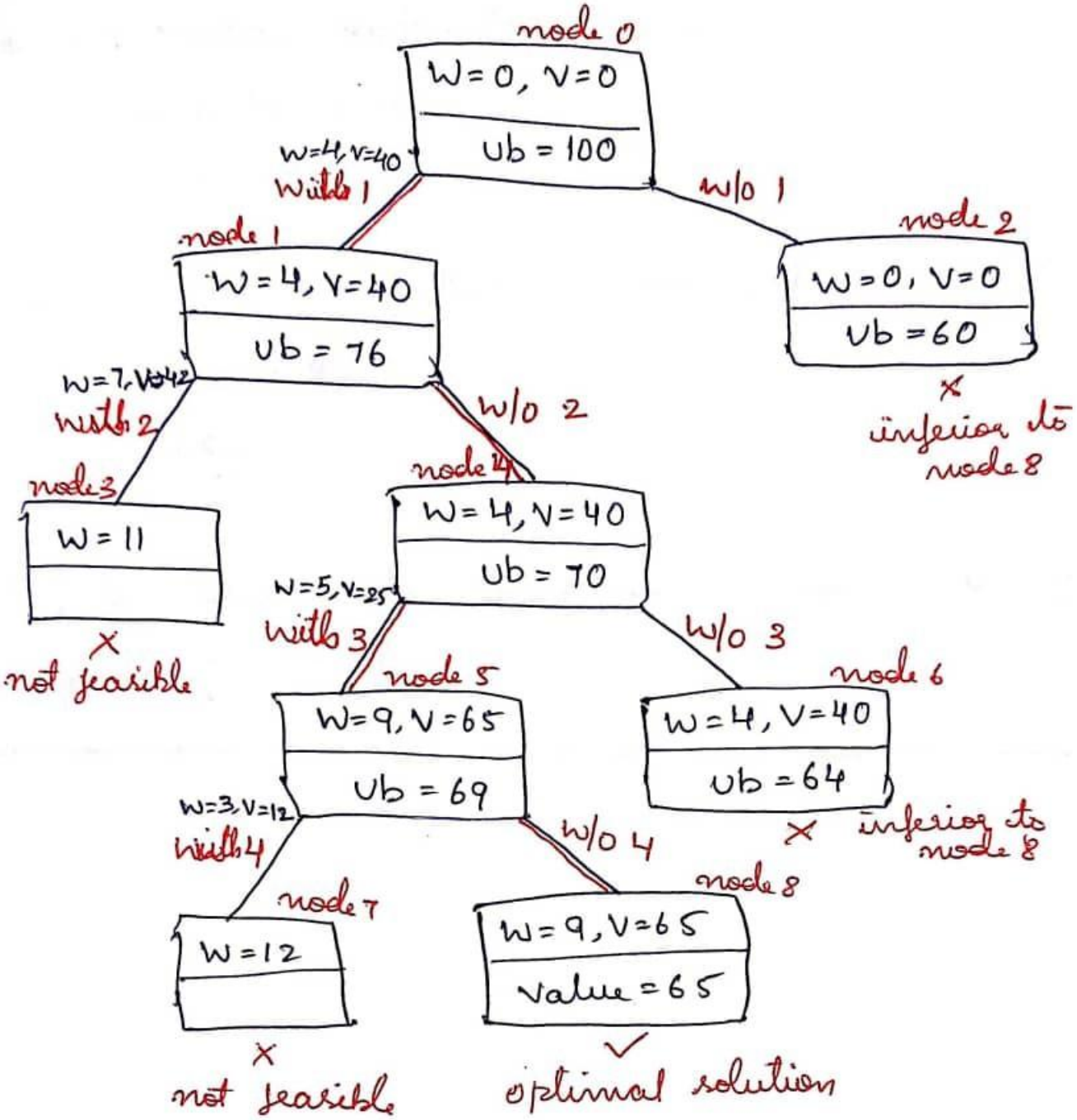


Figure: State-space tree of the best-first B&B algorithm for the instance of the knapsack problem.

- $W = 4, V = 40$, v_3/w_3 (without 2)

$Ub = 40 + (10-4)5$

$= 40 + 6 \times 5$

$\boxed{Ub = 70}$

- $W = 9, V = 65$, v_4/w_4 (with 3)

$Ub = 65 + (10-9)4 = 69$. $\boxed{Ub = 69}$

- $w = 4, v = 40$, v_4/w_4 (without 3)

$Ub = 40 + (10-4)\,4$

$\boxed{Ub = 64}$

- $w = 9, v = 65$, (without 4), no next item, $v_{i+1}/w_{i+1} = 1$

$Ub = 65 + (10-9) \cdot 1$

$\boxed{Ub = 65}$

- Objects / items to be placed in knapsack are 1st & 3rd

∴ Max profit = $\boxed{65}$.

12.3 Approximation Algorithms for NP-Hard Problems:

There are two versions of Knapsack problem.

1. 0/1 Knapsack problem: (Discrete)
   - items are indivisible (either take an item or not)
2. Fractional Knapsack problem: (continuous)
   - items are divisible (can take any fraction of an item

---

1. Greedy algorithm for the discrete Knapsack problem

Step 1: Compute the value-to-weight ratios $r_i = v_i/w_i$, $i = 1, \ldots n$, for the items given

Step 2: Sort the items in non-increasing order of the ratios computed in step 1.

Step 3: Repeat the following operation until no item is left in the sorted list: if the current item on the list fits into the knapsack, place it in the knapsack & proceed to the next item; otherwise, just proceed to the next item.

Eg:

item	weight	value	v_i/w_i
1	7	42	6
2	3	12	4
3	4	40	10
4	5	25	5

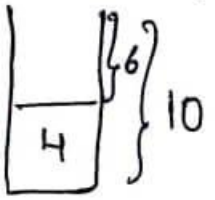
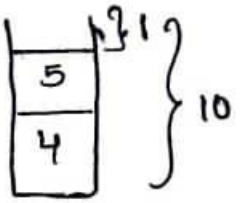
Step 1: Compute $r_i = v_i/w_i$

Step 2: Arrange in non increasing order

item	weight	value	v/w
1	4	40	10
2	7	42	6
3	5	25	5
4	3	12	4

Step 3:

Initially $W = 10$

	weight	value	
item 1	4	40 ✓	$\Longrightarrow$  After placing 1st item $W = 10 - 4 = 6$
item 2	7	42 ✗	$\Longrightarrow w_i > W$, i.e $7 > 6$ not feasible
item 3	5	25 ✓	$\Longrightarrow$  After placing 3rd item $W = 6 - 5 = 1$
item 4	3	12 ✗	$\Longrightarrow w_4 > W$, i.e $3 > 1$ not feasible

∴ items picked 1 & 3, profit = 40 + 25 = 65

---

2. Greedy algorithm for the continuous Knapsack problem

Step 1: Compute the value-to-weight ratios v_i / w_i, $i = 1 \ldots n$ for the items given.

Step 2: Sort the items in nonincreasing order of the ratios computed in step 1.

Step 3: Repeat the following operation until the knapsack is filled to its full capacity or no item is left

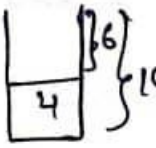
in the sorted list: if the current item on the list fits into the knapsack in its entirety, take it & proceed to the next item; otherwise, take its largest fraction to fill the knapsack to its full capacity & stop.

Eg: After step1 & step 2

item	weight	value	v/w
1	4	40	10
2	7	42	6
3	6	25	5
4	3	12	4

Step3:

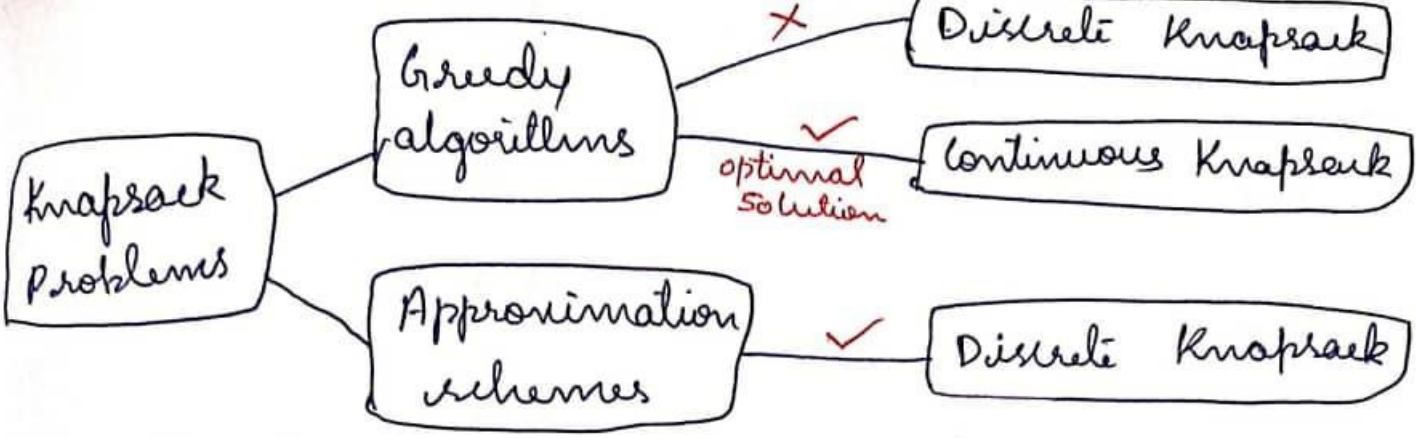
Initially W = 10

	Weight	value	
item1	4	40 ✓	⇒  After placing 1st item W = 10 − 4 = 6
item2	7	42 ✓ 6/7	⇒ Take the largest fraction to fill the knapsack. Select 6/7 of the item
item3	6	25 ✗	⇒ $w_i > W$, i.e 6 > 0 not feasible

After placing 2nd item W = 6 − 6 = 0

∴ items placed 1st item & (6/7) of 2nd item,

$$\text{Profit} = 40 + \left(42 * \frac{6}{7}\right) = 40 + 36 = \boxed{76}$$



Approximation schemes for Knapsack problem

- This algorithm generates all subsets of k items or less, & for each one that fits into the knapsack it adds the remaining items as the greedy algorithm would do (i.e, in nonincreasing order of their value-to weight ratios).
- The subset of the highest value obtained in this fashion is returned as the algorithm's output.

$$\frac{f(s^*)}{f(s_a^{(k)})} \leq 1 + 1/k$$

for any instance of size n. where k is an integer parameter in range $0 \leq k \leq n$.

Eg: Approximation schem with $k = 2$ $W = 10$

item	weight	value	v_i/w_i
1	4	40	10
2	7	42	6
3	5	25	5
4	1	4	4

W	subset	Possible added items	Value or Profit
4+5+1	ϕ	1, 3, 4	$ 69
4+5+1	{1}	3, 4	$ 69
7+1	{2}	4	$ 46
5+4+1	{3}	1, 4	$ 69
1+4+5	{4}	1, 3	$ 69
(4+7) 11 > W	{1, 2}	not feasible	
4+5+1	{1, 3}	4	$ 69
4+1+5	{1, 4}	3	$ 69
(7+5) 12 > W	{2, 3}	not feasible	
7+1	{2, 4}		$ 46
5+1	{3, 4}	1	$ 69

optimal solution is {1, 3, 4}, W = 10, Profit = 69

Algorithm efficiency is in $O(kn^{k+1})$