AVL

- **DEFINITION An *AVL tree is a binary search tree in which the balance factor of*** every node, which is defined as the difference between the heights of the node's left and right subtrees, is either 0 or +1 or −1. (The height of the empty tree is defined as−1. Of course, the balance factor can also be computed as the difference between the numbers of levels rather than the height difference of the node's left and right subtrees.)

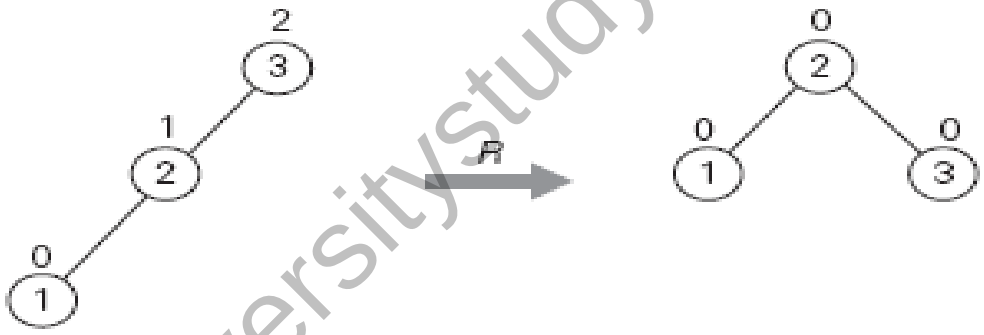
If an insertion of a new node makes an AVL tree unbalanced, we transform the tree by a rotation. A ***rotation in an AVL tree is a local transformation of its*** subtree rooted at a node whose balance has become either +2 or −2. If there are several such nodes, we rotate the tree rooted at the unbalanced node that is the closest to the newly inserted leaf.

single right rotation, or R-rotation.

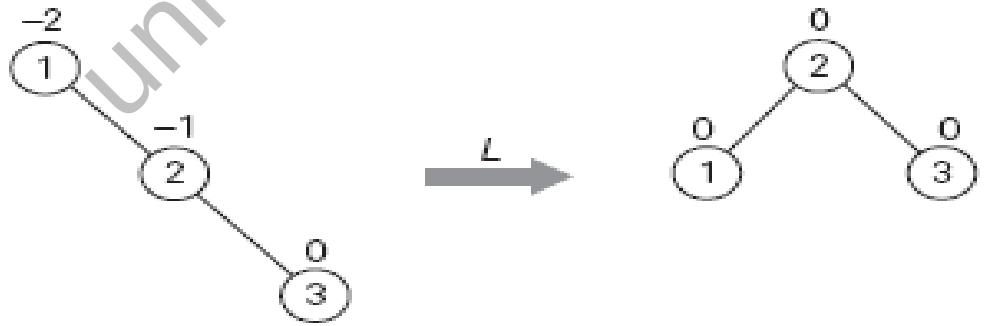
- this rotation is performed after a new key is inserted into the left subtree of the left child of a tree whose root had the balance of +1 before the insertion.

L-rotation

- The symmetric ***single left rotation, or L-rotation, is the mirror image of the*** single *R-rotation. It is performed after a new key is inserted into the right subtree* of the right child of a tree whose root had the balance of −1 before the insertion.



(a)

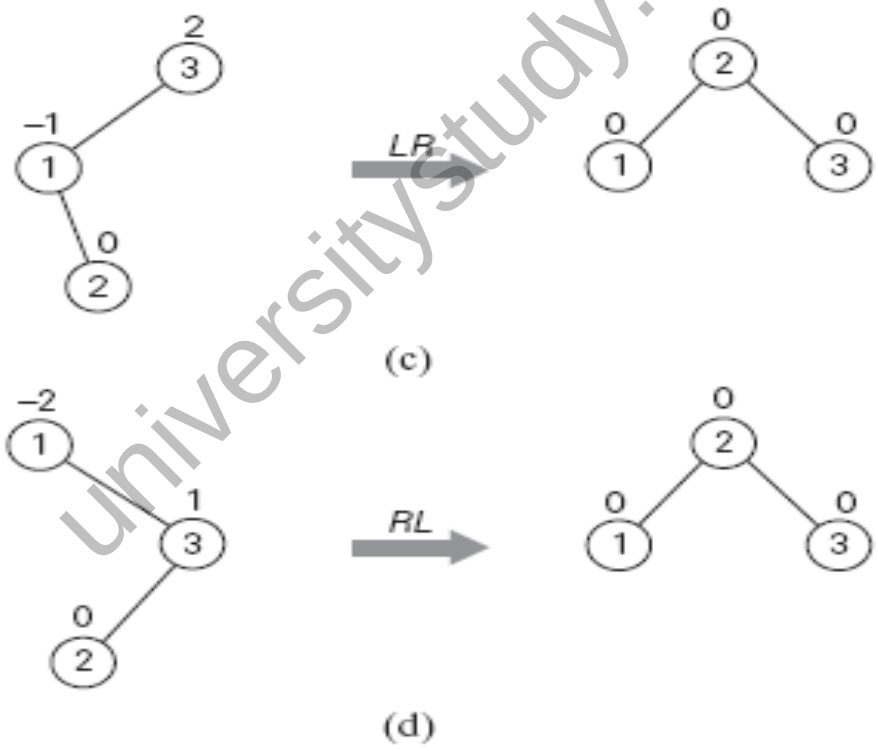


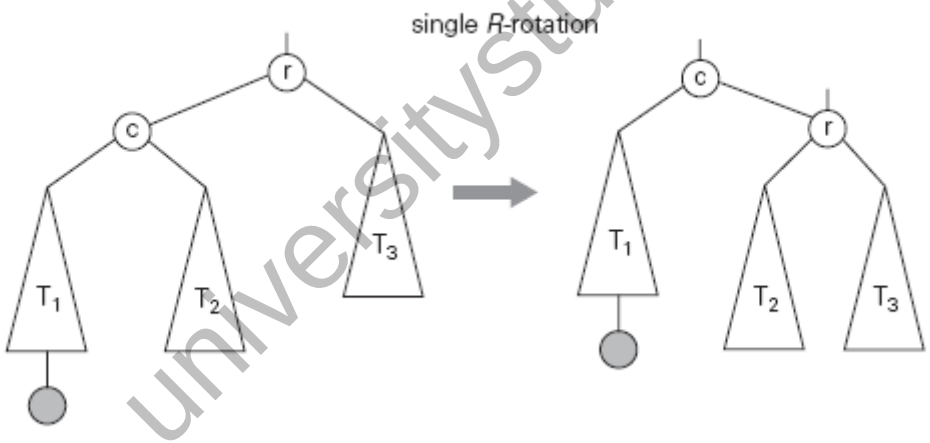
(b)

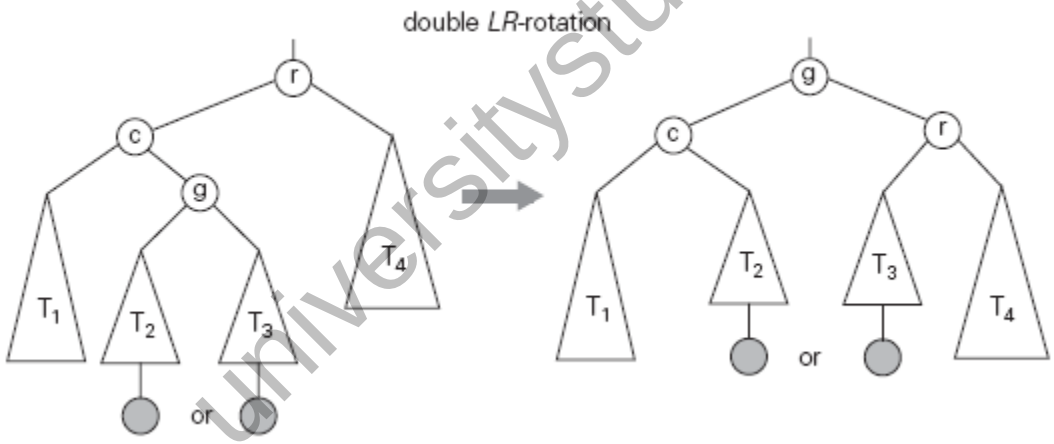
DOUBLE LR AND RL ROTATION

- It is performed after a new key is inserted into the right subtree of the left child of a tree whose root had the balance of +1 before the insertion.
- The ***double right-left rotation (RL-rotation) is the mirror image of the double*** *LR-rotation and is left for the exercises.*

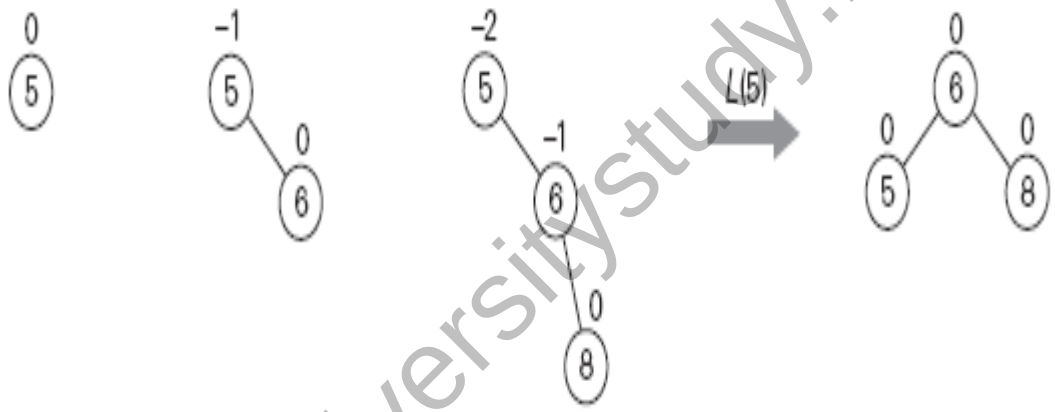
Double LR AND RL ROTATION

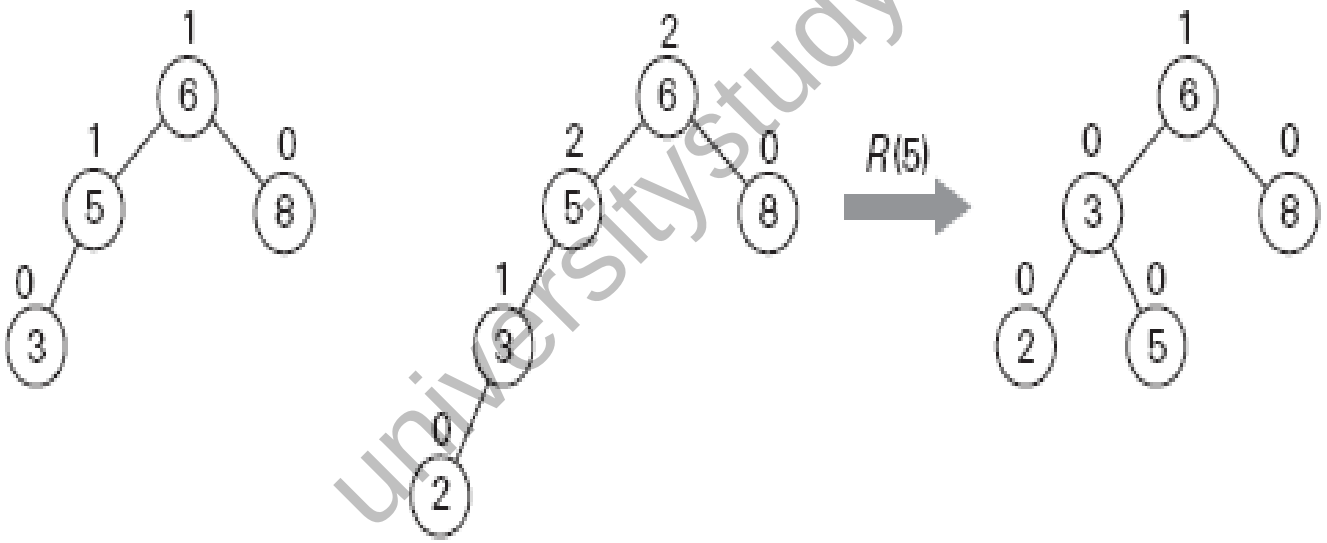


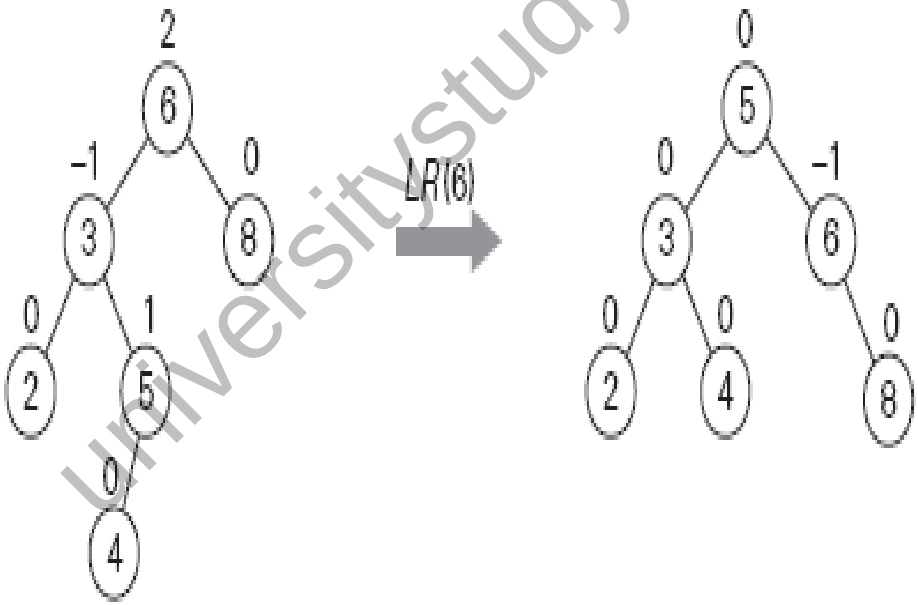


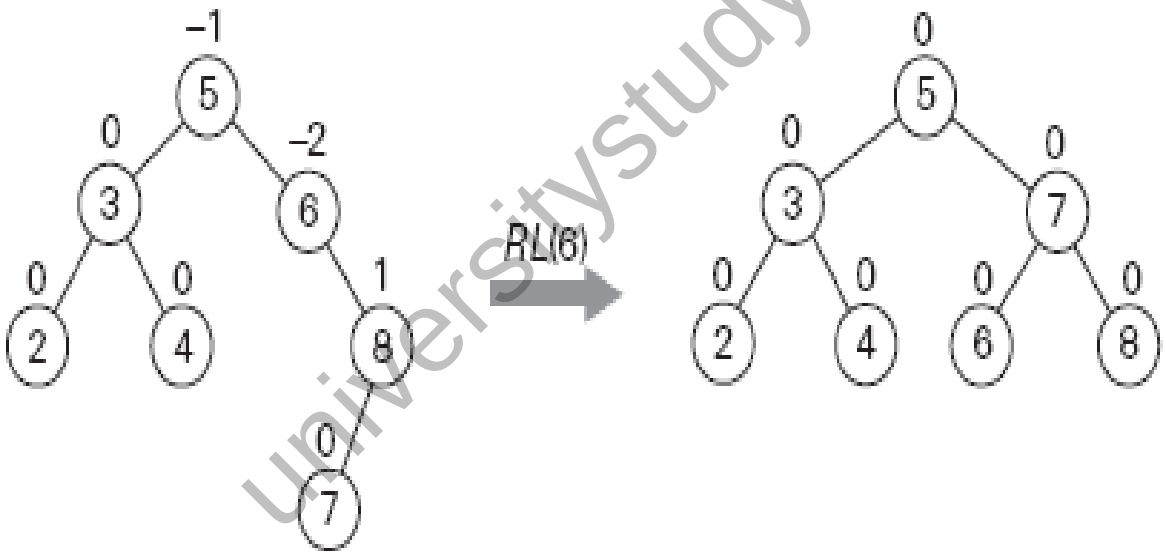


5,6,8,3,2,4,7









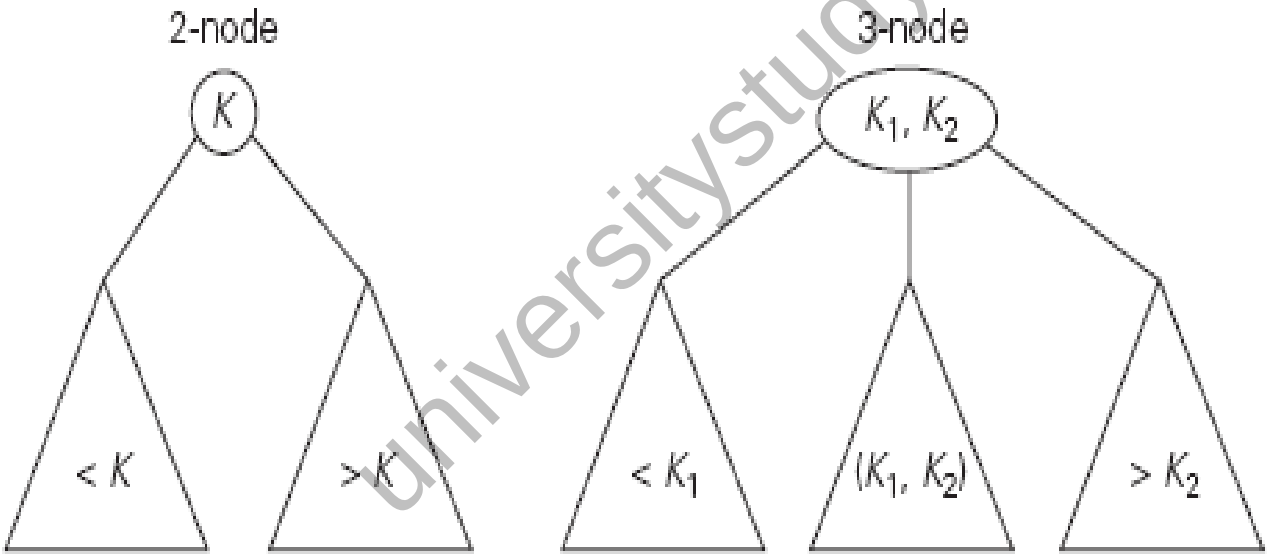
Complexity

- Theta (log n)

- A ***2-3 tree is a tree that can have nodes of two*** kinds: 2-nodes and 3-nodes.
- A ***2-node contains a single key K and has two children:*** the left child serves as the root of a subtree whose keys are less than *K, and the* right child serves as the root of a subtree whose keys are greater than *K.*

- A ***3-node contains two ordered keys K1 and K2 (K1<K2) and has three*** children.
- The leftmost child serves as the root of a subtree with keys less than *K1,*
- the middle child serves as the root of a subtree with keys between *K1 and K2,*
- and the rightmost child serves as the root of a subtree with keys greater than *K2*

- The last requirement of the 2-3 tree is that all its leaves must be on the same level.
- In other words, a 2-3 tree is always perfectly height-balanced:
- the length of a path from the root to a leaf is the same for every leaf. It is this property that we “buy” by allowing more than one key in the same node of a search tree.



As for any search tree, the efficiency of the dictionary operations depends on the tree's height. So let us first find an upper bound for it. A 2-3 tree of height h with the smallest number of keys is a full tree of 2-nodes (such as the final tree in Figure 6.8 for $h = 2$). Therefore, for any 2-3 tree of height h with n nodes, we get the inequality

$$n \geq 1 + 2 + \cdots + 2^h = 2^{h+1} - 1,$$

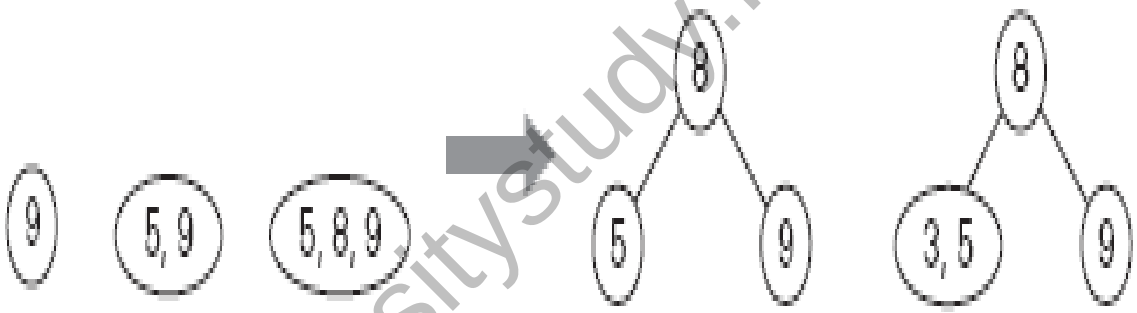
and hence

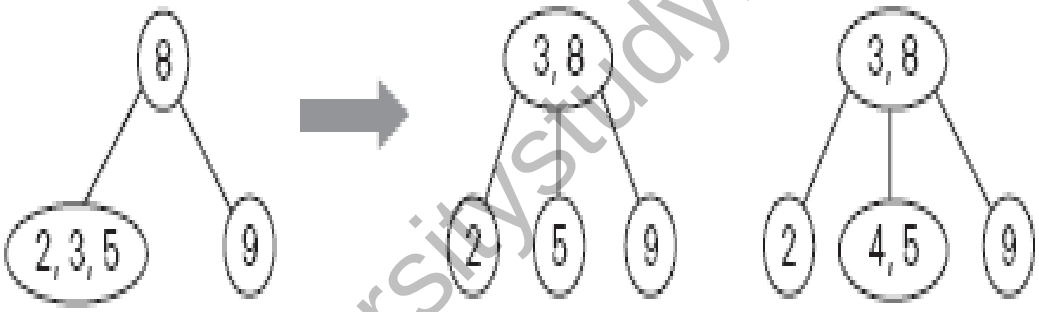
$$h \leq \log_2(n + 1) - 1.$$

On the other hand, a 2-3 tree of height h with the largest number of keys is a full tree of 3-nodes, each with two keys and three children. Therefore, for any 2-3 tree with n nodes,

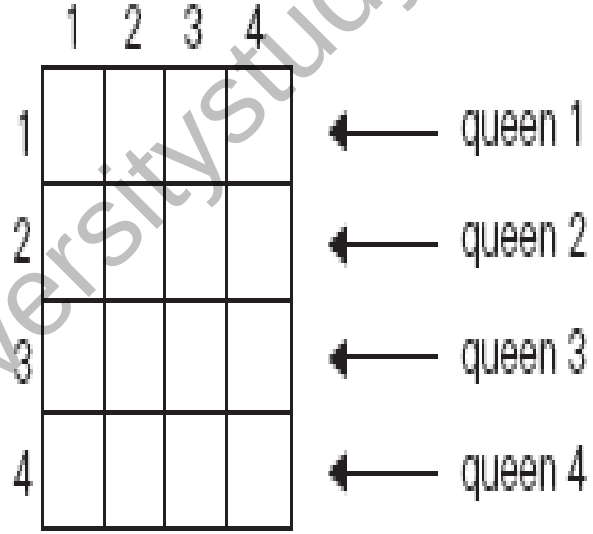
$$n \leq 2 \cdot 1 + 2 \cdot 3 + \cdots + 2 \cdot 3^h = 2(1 + 3 + \cdots + 3^h) = 3^{h+1} - 1$$

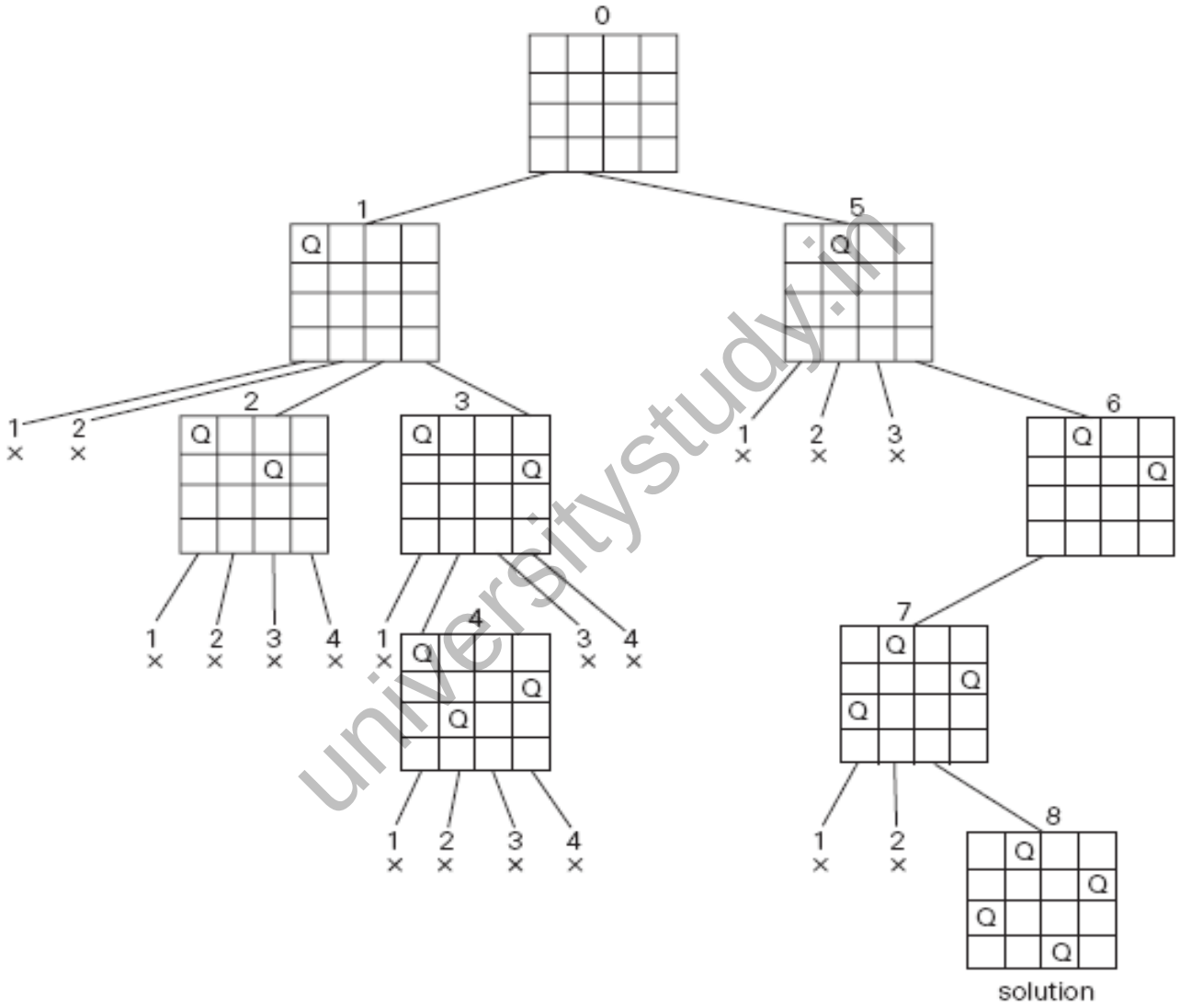
9,5,8,3,2,4,7





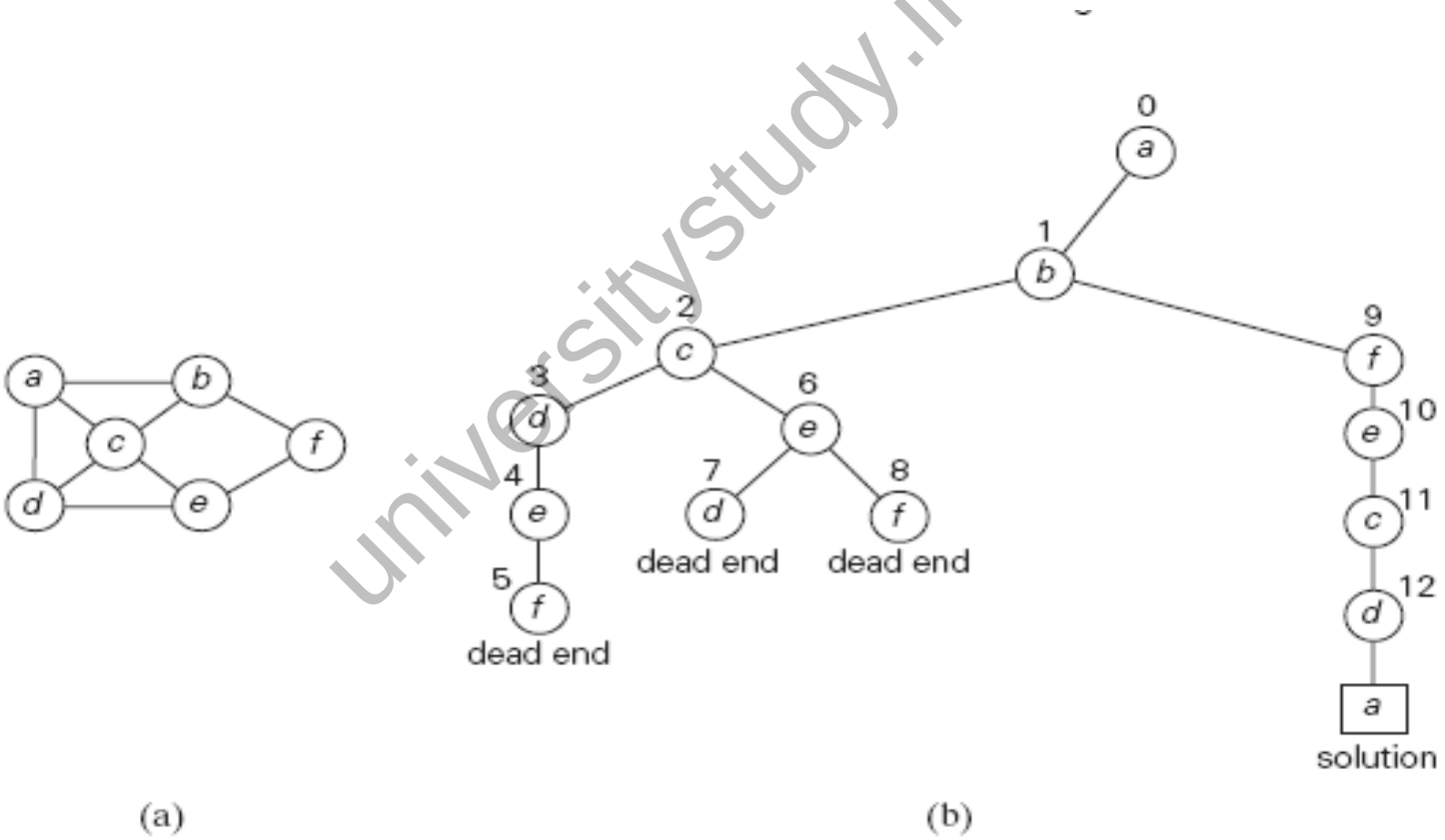
We start with the empty board and then place queen 1 in the first possible position of its row, which is in column 1 of row 1. Then we place queen 2, after trying unsuccessfully columns 1 and 2, in the first acceptable position for it, which is square (2, 3), the square in row 2 and column 3. This proves to be a dead end because there is no acceptable position for queen 3. So, the algorithm backtracks and puts queen 2 in the next possible position at (2, 4). Then queen 3 is placed at (3, 2), which proves to be another dead end. The algorithm then backtracks all the way to queen 1 and moves it to (1, 2). Queen 2 then goes to (2, 4), queen 3 to (3, 1), and queen 4 to (4, 3), which is a solution to the problem. The state-space tree of this search is shown in Figure 12.2.



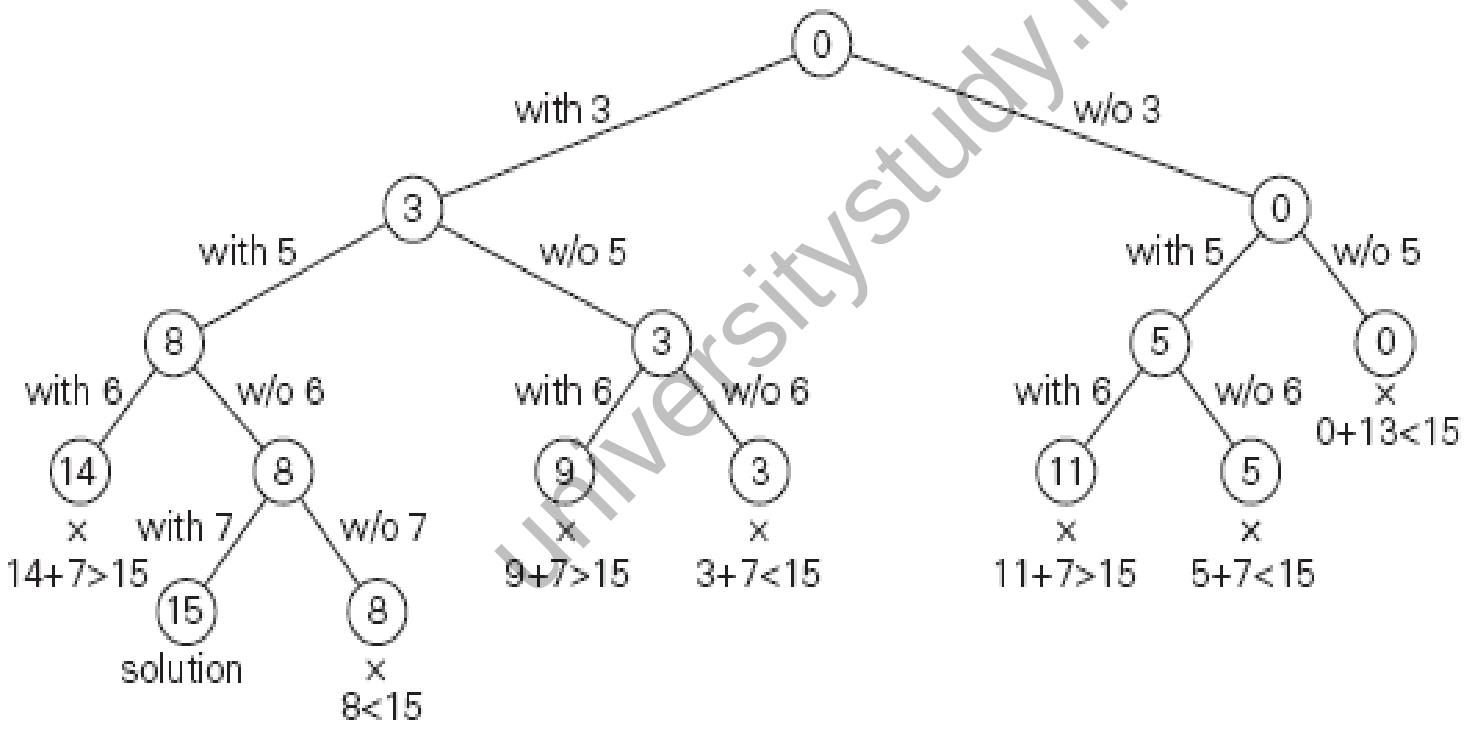


State-space tree of solving the four-queens problem by backtracking. × denotes an unsuccessful attempt to place a queen in the indicated column. The numbers above the nodes indicate the order in which the nodes are generated.

Hamiltonian Circuit Problem



Subset problem



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 problem's solutions
if X[1..i] is a solution write X[1..i]
else      //see Problem 9 in this section's exercises
    for each element x ∈ S_{i+1} consistent with X[1..i] and the constraints do
        X[i + 1] ← x
        Backtrack(X[1..i + 1])
```

Branch and Bound

Compared to backtracking, branch-and-bound requires two additional items:

- a way to provide, for every node of a state-space tree, a bound on the best value of the objective function[1] on any solution that can be obtained by adding further components to the partially constructed solution represented by the node
- the value of the best solution seen so far

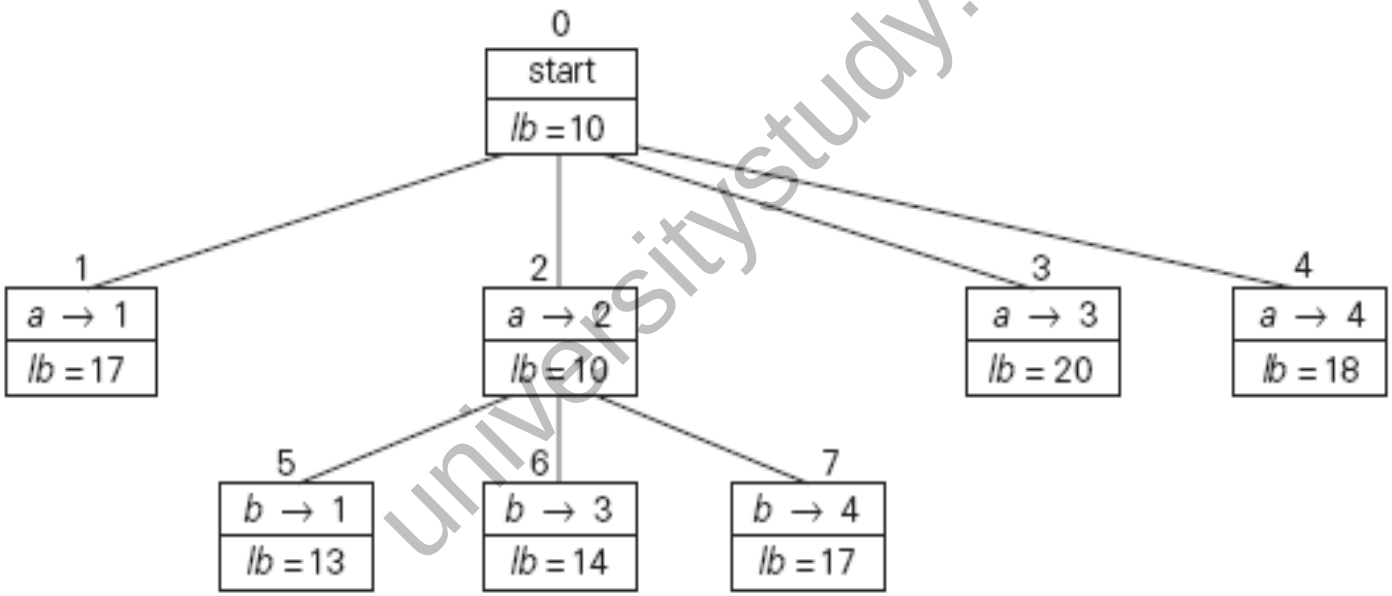
In general, we terminate a search path at the current node in a state-space tree of a branch-and-bound algorithm for any one of the following three reasons:

- The value of the node's bound is not better than the value of the best solution seen so far.
- The node represents no feasible solutions because the constraints of the problem are already violated.
- The subset of feasible solutions represented by the node consists of a single point (and hence no further choices can be made)—in this case, we compare the value of the objective function for this feasible solution with that of the best solution seen so far and update the latter with the former if the new solution is better.

Assignment problem

$$C = \begin{array}{c} \begin{matrix} \text{job 1} & \text{job 2} & \text{job 3} & \text{job 4} \end{matrix} \\ \begin{bmatrix} 9 & 2 & 7 & 8 \\ 6 & 4 & 3 & 7 \\ 5 & 8 & 1 & 8 \\ 7 & 6 & 9 & 4 \end{bmatrix} \end{array} \begin{matrix} \\ \text{person } a \\ \text{person } b \\ \text{person } c \\ \text{person } d \end{matrix}$$

Rather than generating a single child of the last promising node as we did in backtracking, we will generate all the children of the most promising node among nonterminated leaves in the current tree. (Nonterminated, i.e., still promising, leaves are also called *live*.) How can we tell which of the nodes is most promising? We can do this by comparing the lower bounds of the live nodes. It is sensible to consider a node with the best bound as most promising, although this does not, of course, preclude the possibility that an optimal solution will ultimately belong to a different branch of the state-space tree. This variation of the strategy is called the ***best-first branch-and-bound***.



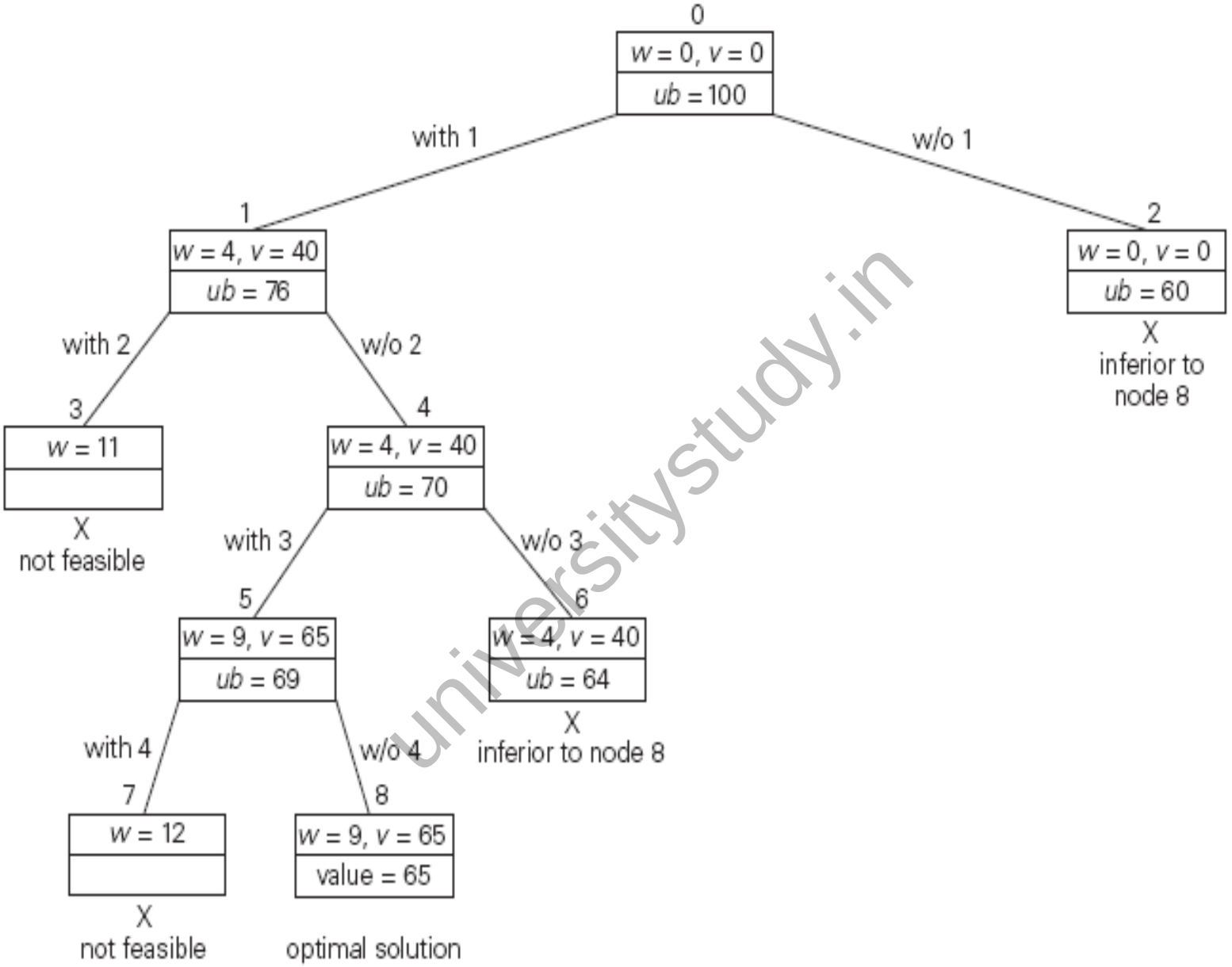
Knapsack problem

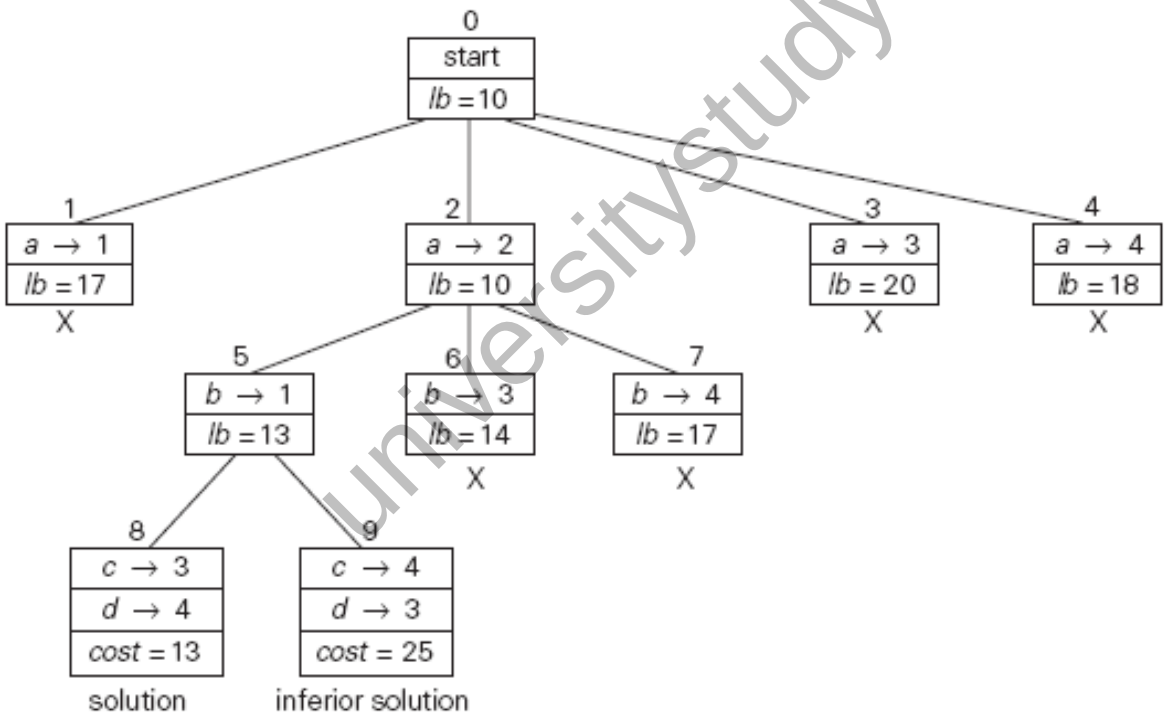
item	weight	value	$\frac{\text{value}}{\text{weight}}$
1	4	\$40	10
2	7	\$42	6
3	5	\$25	5
4	3	\$12	4

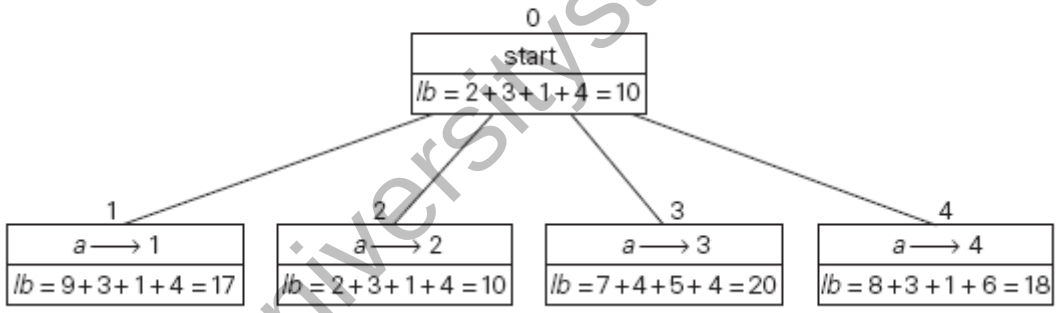
The knapsack's capacity W is 10.

A simple way to compute the upper bound ub is to add to v, the total value of the items already selected, the product of the remaining capacity of the knapsack $W - w$ and the best per unit payoff among the remaining items, which is v_{i+1}/w_{i+1}:

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







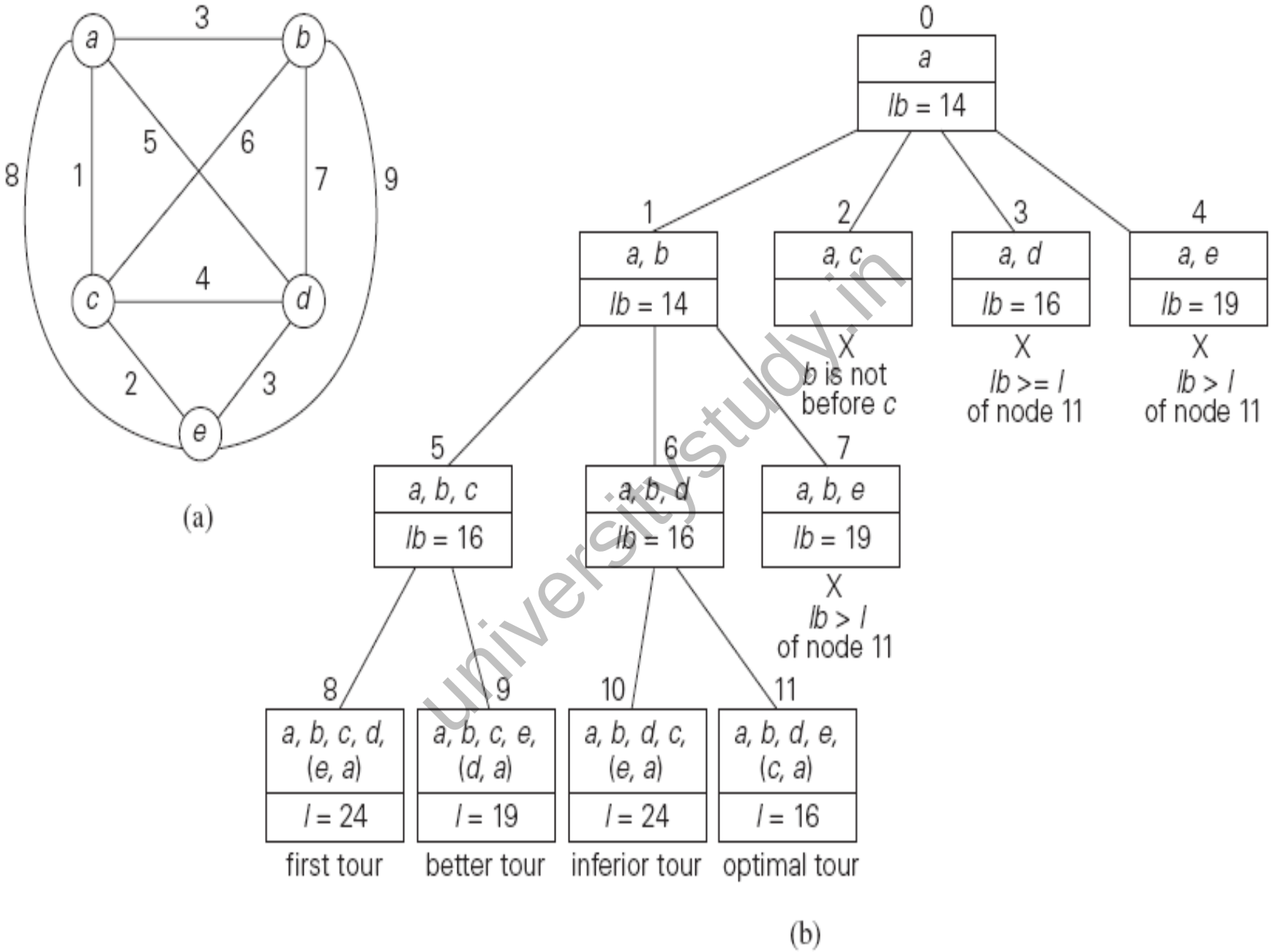
- For each city *i, 1≤ i ≤ n, find* the sum *si of the distances from city i to the two nearest cities; compute the sums of these n numbers, divide the result by 2, and, if all the distances are integers,*round up the result to the nearest integer:

$$lb = \lceil s/2 \rceil.$$

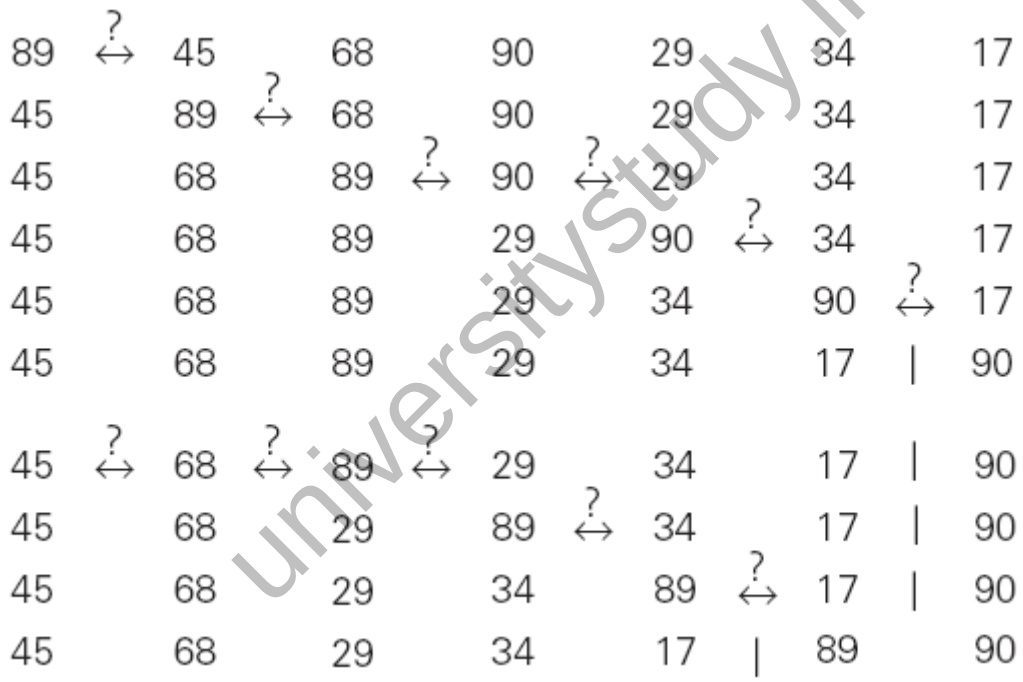
$$lb = \lceil [(1+3)+(3+6)+(1+2)+(3+4)+(2+3)]/2 \rceil = 14.$$

Edge(a,d) and (d,a)

$$\lceil[(1+5)+(3+6)+(1+2)+(3+5)+(2+3)]/2\rceil = 16.$$



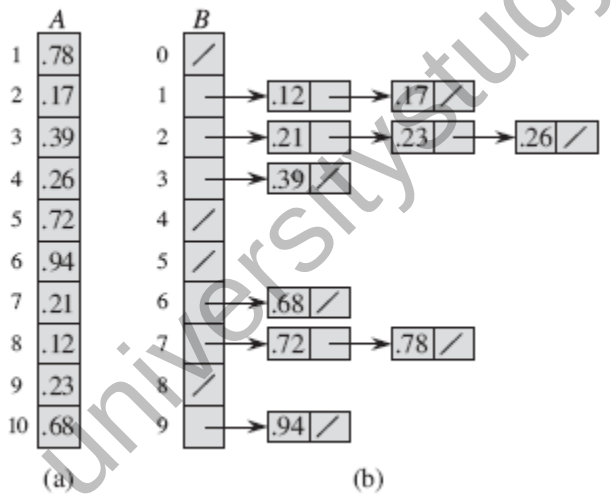
BUBBLE SORT



ALGORITHM *BubbleSort*$(A[0..n-1])$

```
//Sorts a given array by bubble sort
//Input: An array A[0..n − 1] of orderable elements
//Output: Array A[0..n − 1] sorted in nondecreasing order
for i ← 0 to n − 2 do
    for j ← 0 to n − 2 − i do
        if A[j + 1] < A[j] swap A[j] and A[j + 1]
```

BUCKET SORT



```
BUCKET-SORT(A)
1  let B[0..n − 1] be a new array
2  n = A.length
3  for i = 0 to n − 1
4      make B[i] an empty list
5  for i = 1 to n
6      insert A[i] into list B[⌊nA[i]⌋]
7  for i = 0 to n − 1
8      sort list B[i] with insertion sort
9  concatenate the lists B[0], B[1], ..., B[n − 1] together in order
```

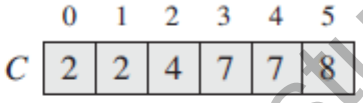
$$T(n) = \Theta(n) + \sum_{i=0}^{n-1} O(n_i^2) \ .$$

MINIMUM AND MAXIMUM

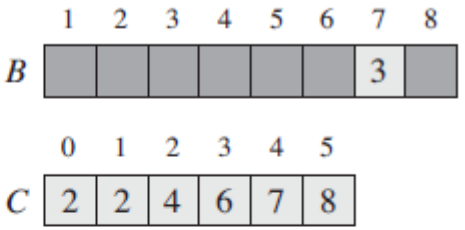
```
MINIMUM(A)
1  min = A[1]
2  for i = 2 to A.length
3       if min > A[i]
4            min = A[i]
5  return min
```

Count sort

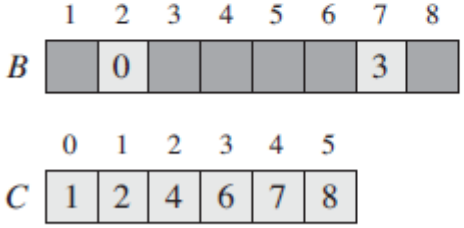
(a)



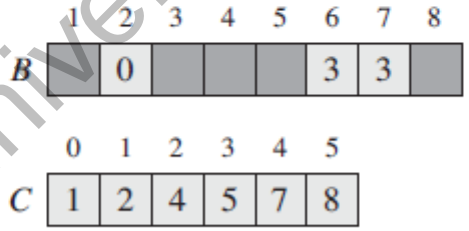
(b)



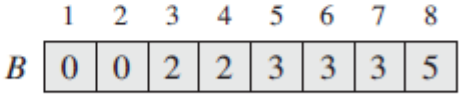
(c)



(d)



(e)



(f)

```
COUNTING-SORT(A, B, k)
 1  let C[0..k] be a new array
 2  for i = 0 to k
 3      C[i] = 0
 4  for j = 1 to A.length
 5      C[A[j]] = C[A[j]] + 1
 6  // C[i] now contains the number of elements equal to i.
 7  for i = 1 to k
 8      C[i] = C[i] + C[i − 1]
 9  // C[i] now contains the number of elements less than or equal to i.
10  for j = A.length downto 1
11      B[C[A[j]]] = A[j]
12      C[A[j]] = C[A[j]] − 1
```

CSE 326: Data Structures
Sorting in (kind of) linear time

Zasha Weinberg in lieu of Steve Wolfman

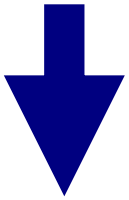
Winter Quarter 2000

BinSort (a.k.a. BucketSort)

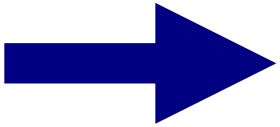
- If all keys are 1…K
- Have array of size K
- Put keys into correct bin (cell) of array

BinSort example

- K=5. list=(5,1,3,4,3,2,1,1,5,4,5)



Bins in array	
key = 1	1,1,1
key = 2	2
key = 3	3,3
key = 4	4,4
key = 5	5,5,5



Sorted list:
1,1,1,2,3,3,4,4,5,5,5

BinSort Pseudocode

```
procedure BinSort (List L,K)

LinkedList bins[1..K]
{ Each element of array bins is linked list.
Could also do a BinSort with array of arrays. }

For Each number x in L
      bins[x].Append(x)
End For
For i = 1..K
      For Each number x in bins[i]
            Print x
      End For
End For
```

BinSort Running time

BinSort Conclusion:

- K is a constant
  - BinSort is linear time
- K is variable
  - Not simply linear time
- K is large (e.g. 2^{32})
  - Impractical

BinSort is “stable”

- Stable Sorting algorithm.
  - Items in input with the same key end up in the same order as when they began.
  - Important if keys have associated values
  - Critical for RadixSort

RadixSort

- Radix = “The base of a number system” (Webster’s dictionary)
- History: used in 1890 U.S. census by Hollerith*
- Idea: BinSort on each digit, bottom up.

* Thanks to Richard Ladner for this fact, taken from Winter 1999 CSE 326 course web.

RadixSort – magic! It works.

- Input list:
  126, 328, 636, 341, 416, 131, 328
- BinSort on lower digit:
  341, 131, 126, 636, 416, 328, 328
- BinSort result on next-higher digit:
  416, 126, 328, 328, 131, 636, 341
- BinSort that result on highest digit:
  126, 131, 328, 328, 341, 416, 636

Not magic. It provably works.

- Keys
  - N-digit numbers
  - base B
- Claim: after i^{th} BinSort, least significant i digits are sorted.
  - e.g. B=10, i=3, keys are 1776 and 8234. 8234 comes before 1776 for last 3 digits.

Induction to the rescue!!!

- base case:
  - i=0. 0 digits are sorted (that wasn't hard!)

Induction is rescuing us…

- Induction step
  - assume for i, prove for i+1.
  - consider two numbers: X, Y. Say X_i is i^{th} digit of X (from the right)
    - $X_{i+1} < Y_{i+1}$ then $i+1^{th}$ BinSort will put them in order
    - $X_{i+1} > Y_{i+1}$, same thing
    - $X_{i+1} = Y_{i+1}$, order depends on last i digits. Induction hypothesis says already sorted for these digits. (Careful about ensuring that your BinSort preserves order aka "stable"…)

Paleontology fact

- Early humans had to survive without induction.

Running time of Radixsort

- How many passes?
- How much work per pass?
- Total time?

- Conclusion
  - Not truly linear if K is large.
- In practice
  - RadixSort only good for large number of items, relatively small keys
  - Hard on the cache, vs. MergeSort/QuickSort

What data types can you RadixSort?

- Any type T that can be BinSorted
- Any type T that can be broken into parts A and B,
  - You can reconstruct T from A and B
  - A can be RadixSorted
  - B can be RadixSorted
  - A is always more significant than B, in ordering

Example:

- 1-digit numbers can be BinSorted
- 2 to 5-digit numbers can be BinSorted without using too much memory
- 6-digit numbers, broken up into A=first 3 digits, B=last 3 digits.
  - A and B can reconstruct original 6-digits
  - A and B each RadixSortable as above
  - A more significant than B

RadixSorting Strings

- 1 Character can be BinSorted
- Break strings into characters
- Need to know length of biggest string (or calculate this on the fly).

RadixSorting Strings example

	5th pass	4th pass	3rd pass	2nd pass	1st pass
String 1	z	i	p	p	y
String 2	z	a	p		
String 3	a	n	t	s	
String 4	f	l	a	p	s

NULLs are just like fake characters

RadixSorting Strings running time

- N is number of strings
- L is length of longest string
- RadixSort takes O(N*L)

RadixSorting IEEE floats/doubles

- You can RadixSort real numbers, in most representations
- We do IEEE floats/doubles, which are used in C/C++.
- Some people say you can't RadixSort reals. In practice (like IEEE reals) you can.

Anatomy of a real number

Sign (positive or negative)

$-1.3892 * 10^{24}$

$+1.507 * 10^{-17}$

Exponent

Significand (a.k.a. mantissa)

IEEE floats in binary*

$-1.0110100111 * 2^{1011}$
$+1.101101001 * 2^{-1}$

- Sign: 1 bit
- Significand: always 1.*fraction*. *fraction* uses 23 bits
- Biased exponent: 8 bits.
  - Bias: represent –127 to +127 by adding 127 (so range is 0-254)

* okay, simplified to focus on the essential ideas.

Observations

- significand always starts with 1
  → only one way to represent any number
- Exponent always more significant than significand
- Sign is most significant, but in a weird way

Pseudocode

```
procedure RadixSortReals (Array[1..N])

RadixSort Significands in Array as unsigned ints
RadixSort biased exponents in Array as u-ints

Sweep thru Array,
      put negative #'s separate from positive #'s.
Flip order of negative #'s, & put them before
      the positive #'s.

Done.
```

Transform and Conquer

Transform and Conquer

- Algorithms based on the idea of transformation
  - Transformation stage
    - Problem instance is modified to be more amenable to solution
  - Conquering stage
    - Transformed problem is solved
- Major variations are for the transform to perform:
  - Instance simplification
  - Different representation
  - Problem reduction

Presorting

- Presorting is an old idea, you sort the data and that allows you to more easily compute some answer
  - Saw this with quickhull, closest point
- Some other simple presorting examples
  - Element Uniqueness
  - Computing the mode of *n* numbers

Element Uniqueness

- Given a list *A* of *n* orderable elements, determine if there are any duplicates of any element

Brute Force:

```
for each x ∈ A
  for each y ∈ {A – x}
      if x = y return not unique
return unique
```

Presorting:

```
Sort A
for i ← 1 to n-1
   if A[i] = A[i+1] return not unique
return unique
```

Runtime?

Computing a mode

- A mode is a value that occurs most often in a list of numbers
  - e.g. the mode of [5, 1, 5, 7, 6, 5, 7] is 5
  - If several different values occur most often any of them can be considered the mode
- "Count Sort" approach: (assumes all values > 0; what if they aren't?)

```
max ← max(A)
freq[1..max] ← 0
for each x ∈ A
   freq[x] +=1
mode ← freq[1]
for i ← 2 to max
   if freq[i] > freq[mode]  mode ← i
return mode
```

Runtime?

Presort Computing Mode

```
Sort A
i ← 0
modefrequency ← 0
while i ≤ n-1
   runlength ← 1;   runvalue ← A[i]
   while i+runlength ≤ n-1  and A[i+runlength] = runvalue
            runlength += 1
   if runlength > modefrequency
            modefrequency ← runlength
            modevalue ← runvalue
   i+= runlength

return modevalue
```

Gaussian Elimination

- This is an example of transform and conquer through representation change
- Consider a system of two linear equations:

  $A_{11}x + A_{12}y = B_1$

  $A_{21}x + A_{22}y = B_2$
- To solve this we can rewrite the first equation to solve for x:

  $x = (B_1 – A_{12}y) / A_{11}$
- And then substitute in the second equation to solve for y. After we solve for y we can then solve for x:

  $A_{21}(B_1 – A_{12}y) / A_{11} + A_{22}y = B_2$

Gaussian Elimination

- In many applications we need to solve a system of n equations with n unknowns, e.g.:

  $A_{11}x_1 + A_{12}x_2 + \ldots + A_{1n}x_n = B_1$

  $A_{21}x_1 + A_{22}x_2 + \ldots + A_{2n}x_n = B_2$

  …

  $A_{n1}x_1 + A_{n2}x_2 + \ldots + A_{nn}x_n = B_n$

- If n is a large number it is very cumbersome to solve these equations using the substitution method.
- Fortunately there is a more elegant algorithm to solve such systems of linear equations: Gaussian elimination
  - Named after Carl Gauss

Gaussian Elimination

The idea is to transform the system of linear equations into an equivalent one that eliminates coefficients so we end up with a triangular matrix.

$$
\begin{aligned}
&A_{11}x_1 + A_{12}x_2 + \ldots + A_{1n}x_n = B_1\\
&A_{21}x_1 + A_{22}x_2 + \ldots + A_{2n}x_n = B_2\\
&\ldots\\
&A_{n1}x_1 + A_{n2}x_2 + \ldots + A_{nn}x_n = B_n
\end{aligned}
\quad \Rightarrow \quad
\begin{aligned}
&A_{11}x_1 + A_{12}x_2 + \ldots + A_{1n}x_n = B_1\\
&0x_1 \quad + A_{22}x_2 + \ldots + A_{2n}x_n = B_2\\
&\ldots\\
&0x_1 \quad + 0x_2 \quad + \ldots + A_{nn}x_n = B_n
\end{aligned}
$$

In matrix form we can write this as: $Ax = B \rightarrow A'x = B'$

$$A = \begin{bmatrix} A_{11} & A_{12} & \ldots & A_{1n} \\ A_{21} & A_{22} & \ldots & A_{2n} \\ \ldots & & & \\ A_{n1} & A_{n2} & \ldots & A_{nn} \end{bmatrix} \quad B = \begin{bmatrix} B_1 \\ B_2 \\ \ldots \\ B_n \end{bmatrix}$$

Gaussian Elimination

- Why transform?

$$A_{11}x_1 + A_{12}x_2 + \ldots + A_{1n}x_n = B_1$$
$$0x_1 + A_{22}x_2 + \ldots + A_{2n}x_n = B_2$$
$$\ldots$$
$$0x_1 + 0x_2 + \ldots + A_{nn}x_n = B_n$$

- The matrix with zeros in the lower triangle (it is called an upper triangular matrix) is easier to solve.

  We can solve the last equation first, substitute into the second to last, etc. working our way back to the first one.

Gaussian Elimination Example

- Solve the following system:

$$2x_1 - x_2 + x_3 = 1$$
$$4x_1 + x_2 - x_3 = 5$$
$$x_1 + x_2 + x_3 = 0$$

$$\begin{bmatrix} 2 & -1 & 1 & 1 \\ 4 & 1 & -1 & 5 \\ 1 & 1 & 1 & 0 \end{bmatrix}$$

subtract 2*row1

subtract ½*row1

$$\begin{bmatrix} 2 & -1 & 1 & 1 \\ 0 & 3 & -3 & 3 \\ 0 & \frac{3}{2} & \frac{1}{2} & -\frac{1}{2} \end{bmatrix}$$

subtract ½*row2

$$\begin{bmatrix} 2 & -1 & 1 & 1 \\ 0 & 3 & -3 & 3 \\ 0 & 0 & 2 & -2 \end{bmatrix}$$

Gaussian Elimination

- In our example we replaced an equation with a sum or difference with a multiple of another equation
- We might also need to:
  - Exchange two equations
  - Replace an equation with its nonzero multiple
- Pseudocode:

```
for i ← 1 to n do A[i,n+1] ← B[i]
for i ← 1 to n – 1
   for j ← i+1 to n do
     for k ← i to n+1 do
        A[j,k] ← A[j,k] – A[i,k]*A[j,i] / A[i,i]
```

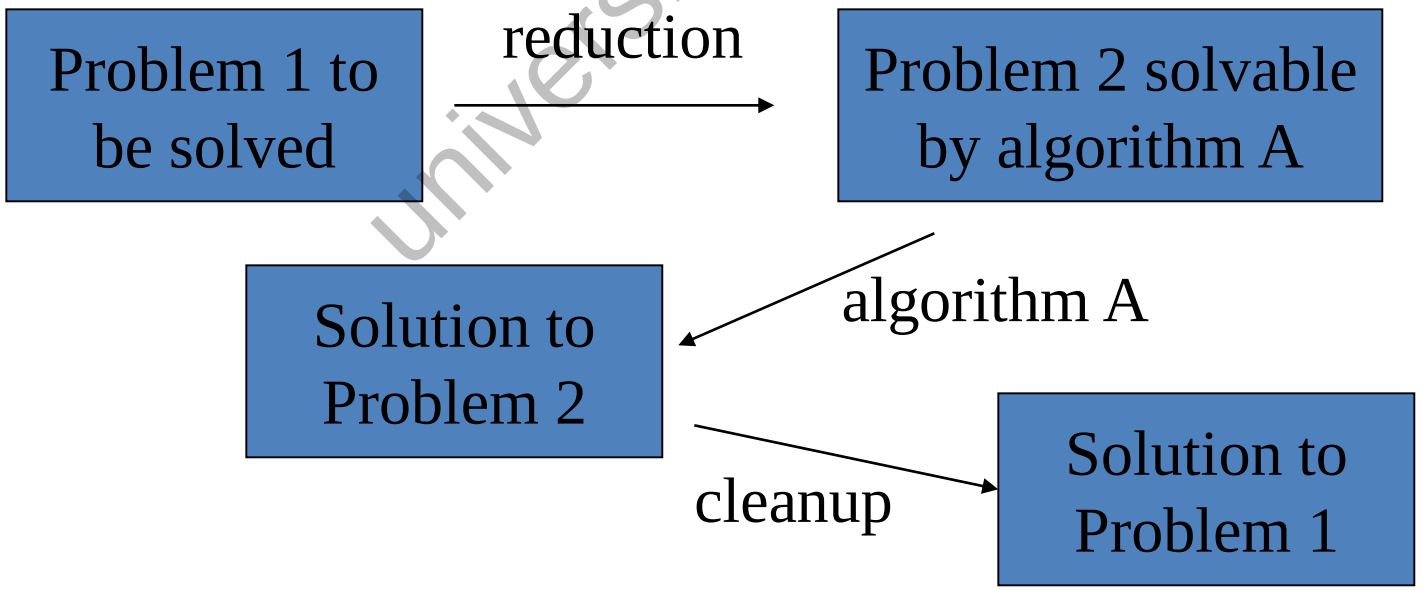
$O(n^3)$ algorithm

Textbook Chapter 6

- Skipping other matrix operations, balanced trees, heaps, binary exponentiation

Problem Reduction

- If you need to solve a problem, reduce it to another problem that you know how to solve; we saw this idea already with NPC problems



Linear Programming

- One more example of problem reduction; linear programming
- A Linear Program (LP) is a problem that can be expressed as follows (the so-called Standard Form):
  - minimize (or maximize) cx
  - subject to
    - Ax = b
    - x >= 0
- where x is the vector of variables to be solved for, A is a matrix of known coefficients, and c and b are vectors of known coefficients. The expression "cx" is called the objective function, and the equations "Ax=b" are called the constraints.

Linear Programming Example

- Wyndor Glass produces glass windows and doors
- They have 3 plants:
  - Plant 1: makes aluminum frames and hardware
  - Plant 2: makes wood frames
  - Plant 3: produces glass and makes assembly
- Two products proposed:
  - Product 1: 8’ glass door with aluminum siding (x1)
  - Product 2: 4’ x 6’ wood framed glass window (x2)
- Some production capacity in the three plants is available to produce a combination of the two products
- Problem is to determine the best product mix to maximize profits

Wyndor Glass Co. Data

Plant	Production time per batch (hr)		Production time available per week (hr)
	Product		
	1	2	
1	1	0	4
2	0	2	12
3	3	2	18
Profit per batch	**$3,000**	**$5,000**	

Formulation:

Maximize $z = 3x_1 + 5x_2$ (objective to maximize $$)

Subject to

$x_1 <= 4$ (Plant One)

$2x_2 <= 12$ (Plant Two)

$3x_1 + 2x_2 <= 18$ (Plant Three)

$x_1, x_2 >= 0$ (Non-negativity requirements)

Example 2: Spacing to Center Text

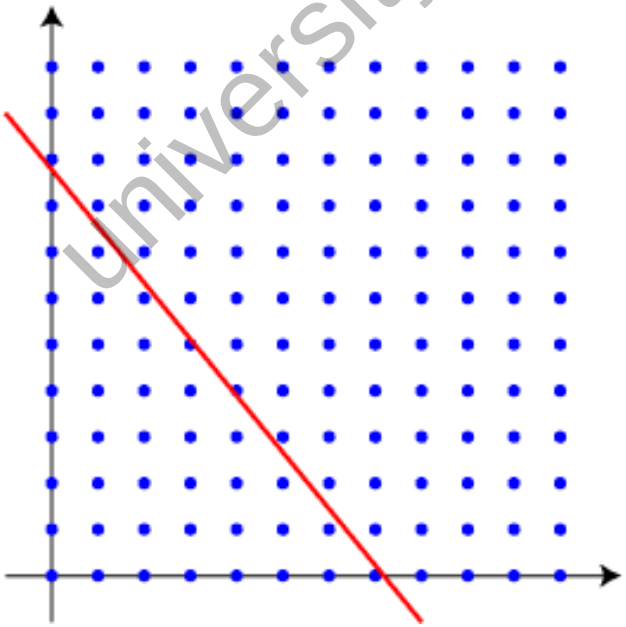
- To center text we need to indent it ourselves by using an appropriate number of space characters. The complication is that we have two types of spaces: the usual space and option-space (also known as non-breaking space). These two spaces are different widths.

- Given three numbers
  - a (the width of a normal space),
  - b (the width of an option-space), and
  - c (the amount we want to indent),
- Find two more numbers
  - x (the number of normal spaces to use), and
  - y (the number of option-spaces to use),

- So that ax+by is as close as possible to c.

Spacing Problem

- Visualize problem in 2 dimensions, say a=11, b=9, and c=79. Each blue dot in the picture represents the combination of x option-spaces and y spaces. The red line represents the ideal width of 79 pixels.
- We want to find a blue dot that's as close as possible to the red line.



Spacing Problem

- If we want to find the closest point below the line then our equations become:

  $x \geq 0$

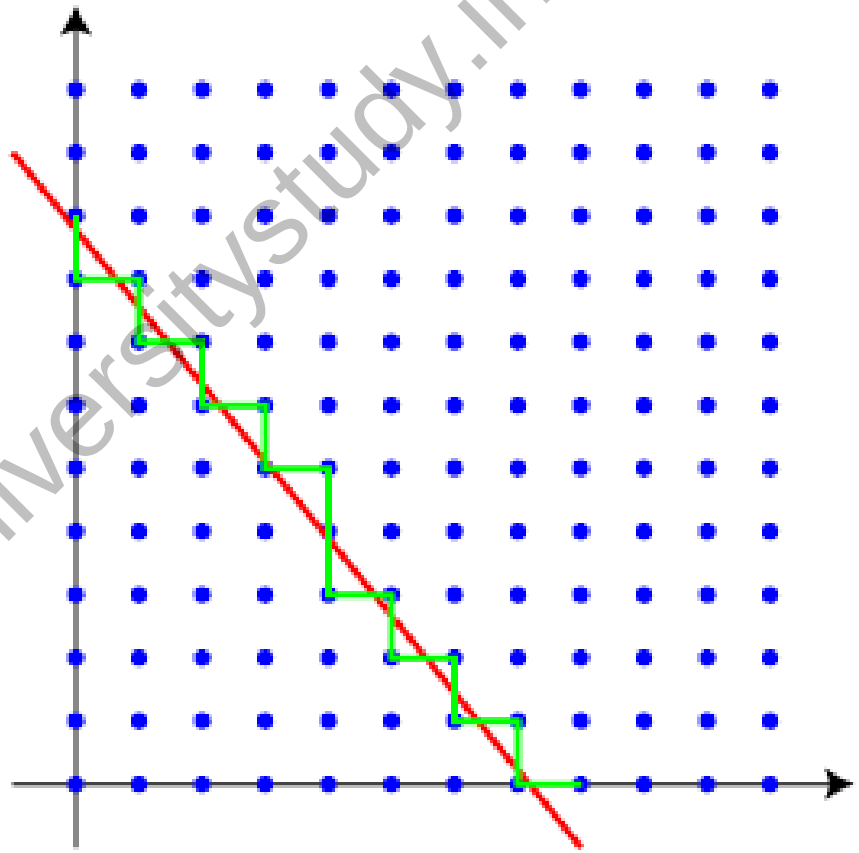
  $y \geq 0$

  $ax + by \leq c$

- The linear programming problem is to maximize $ax + by \leq c$ to find the closest point to the line

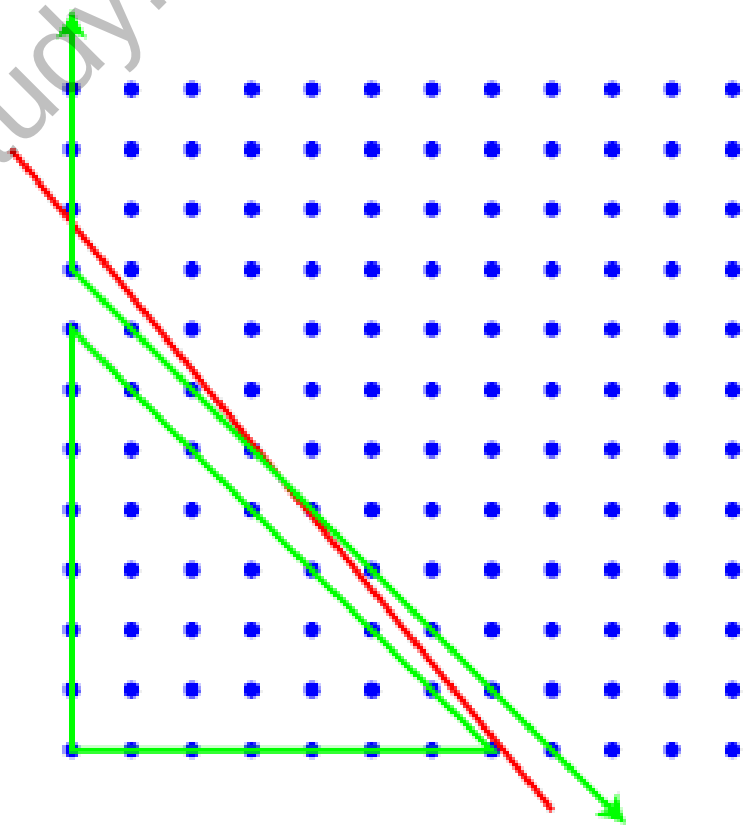
Possible Solutions

- Brute Force



Possible Solutions

- Simplex method
  - Consider only points along boundary of “feasible region”
  - Won’t go into the algorithm here, but it finds solutions in worst case exponential time but generally runs efficiently in polynomial time



Knapsack Problem

- We can reduce the knapsack problem to a solvable linear programming problem
- Discrete or 0-1 knapsack problem:
  - Knapsack of capacity W
  - n items of weights w_1, w_2 … w_n and values v_1, v_2 … v_n
  - Can only take entire item or leave it
- Reduces to:

$$\text{Maximize} \quad \sum_{i=1}^{n} v_i x_i \qquad \text{where } x_i = 0 \text{ or } 1$$

$$\text{Constrained by:} \quad \left(\sum_{i=1}^{n} w_i x_i \right) \leq W$$