RV College of Engineering®

Autonomous Institution Affiliated to Visvesvaraya Technological University, Belagavi

Approved by AICTE New Delhi

Transform and Conquer

UNIT 3

Prof. Rajesh R M

Dept. of AIML

RV College of Engineering

Bengaluru

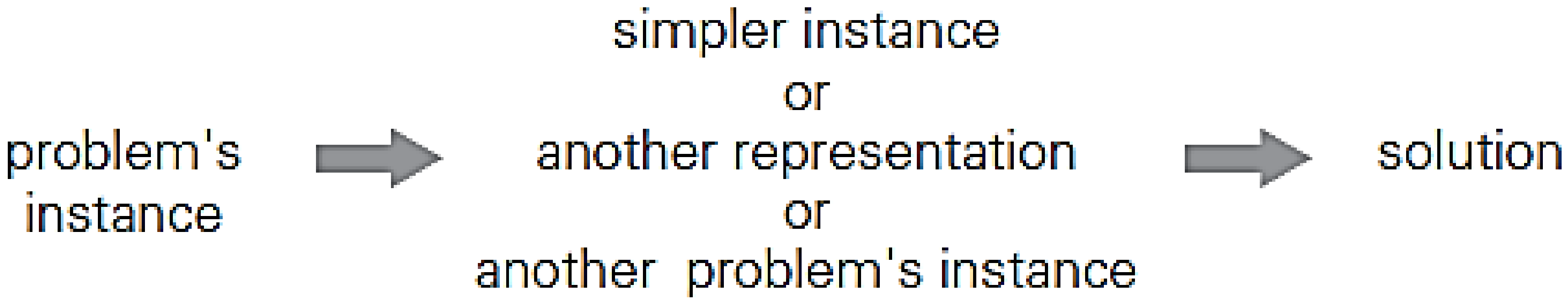
Go, Change the World

Transform and Conquer

Transform-and-conquer

Transform and conquer has two-stage procedures.

- First, in the transformation stage, the problem's instance is modified to be, for one reason or another, more amenable to solution.
- Then, in the second or conquering stage, it is solved.



Transform and Conquer

Transform-and-conquer Variants

- **Instance simplification**: Transformation to a simpler or more convenient instance of the same problem.

Ex: Pre-sorting, AVL tree

- **Representation change**: Transformation to a different representation of the same instance.

Ex: Heap Sort, 2-3 Tree

- **Problem reduction**: Transformation to an instance of a different problem for which an algorithm is already available.

- Ex: LCM(a,b)=$\frac{a*b}{\gcd(a,b)}$

EXAMPLE 1 *Checking element uniqueness in an array*

```
ALGORITHM UniqueElements(A[0..n − 1])
//Determines whether all the elements in a given array are distinct
//Input: An array A[0..n − 1]
//Output: Returns "true" if all the elements in A are distinct and "false"
otherwise
for i ←0 to n − 2 do
        for j ←i + 1 to n − 1 do
                if A[i]= A[j ]
                        return false
return true
```

If this element uniqueness problem looks familiar to you, it should; we considered a brute-force algorithm for the problem. The brute-force algorithm compared pairs of the array's elements until either two equal elements were found or no more pairs were left. Its worst-case efficiency was in (n^2).

Transform and Conquer

Pre-sorting

Its not a direct problem solving technique or a method, its an intermediate stage to solve the problem.

- So far, we have discussed three elementary sorting algorithms—selection sort, bubble sort, and insertion sort that are quadratic in the worst and average cases, and two advanced algorithm mergesort, which is always in (n log n), and quicksort, whose efficiency is also (n log n) in the average case but is quadratic in the worst case.

- If we apply pre-sort technique to find the element uniqueness in the array we can reduce the time complexity to n log n

Transform and Conquer

```
ALGORITHM PresortElementUniqueness(A[0..n − 1])
//Solves the element uniqueness problem by sorting the array first
//Input: An array A[0..n − 1] of orderable elements
//Output: Returns “true” if A has no equal elements, “false” otherwise

sort the array A
for i ←0 to n − 2 do
      if A[i]= A[i + 1] return false
return true
```

T (n) = Tsort(n) + Tscan(n) → (n log n) + (n) = (n log n)

Transform and Conquer

EXAMPLE 2 *Computing a mode* A ***mode*** is a value that occurs most often in a given list of numbers.

For example, for 5, 1, 5, 7, 6, 5, 7, the mode is 5. (If several different values occur most often, any of them can be considered a mode.)

The brute-force approach to computing a mode would scan the list and compute the frequencies of all its distinct values, then find the value with the largest frequency.

Transform and Conquer

EXAMPLE 2 *Computing a mode* A ***mode*** is a value that occurs most often in a given list of numbers.

For example, for 5, 1, 5, 7, 6, 5, 7, the mode is 5. (If several different values occur most often, any of them can be considered a mode.)

The brute-force approach to computing a mode would scan the list and compute the frequencies of all its distinct values, then find the value with the largest frequency.

$$C(n) = \sum_{i=1}^{n}(i-1) = 0 + 1 + \cdots + (n-1) = \frac{(n-1)n}{2} \in \Theta(n^2)$$

Transform and Conquer

```
ALGORITHM PresortMode(A[0..n − 1])
//Computes the mode of an array by sorting it first
//Input: An array A[0..n − 1] of orderable elements
//Output: The array's mode
sort the array A
i ←0 //current run begins at position i
modef requency ←0 //highest frequency seen so far
while i ≤ n − 1 do
    runlength←1
    runvalue←A[i]
    while i + runlength ≤ n − 1 and A[i + runlength]= runvalue
        runlength←runlength + 1
    if runlength > modef requency
        modef requency ←runlength
        modevalue←runvalue
    i ←i + runlength
return modevalue
```

$$T(n) = T_{sort}(n) + T_{search}(n) = \Theta(n \log n) + \Theta(\log n) = \Theta(n \log n)$$

2-3 tree

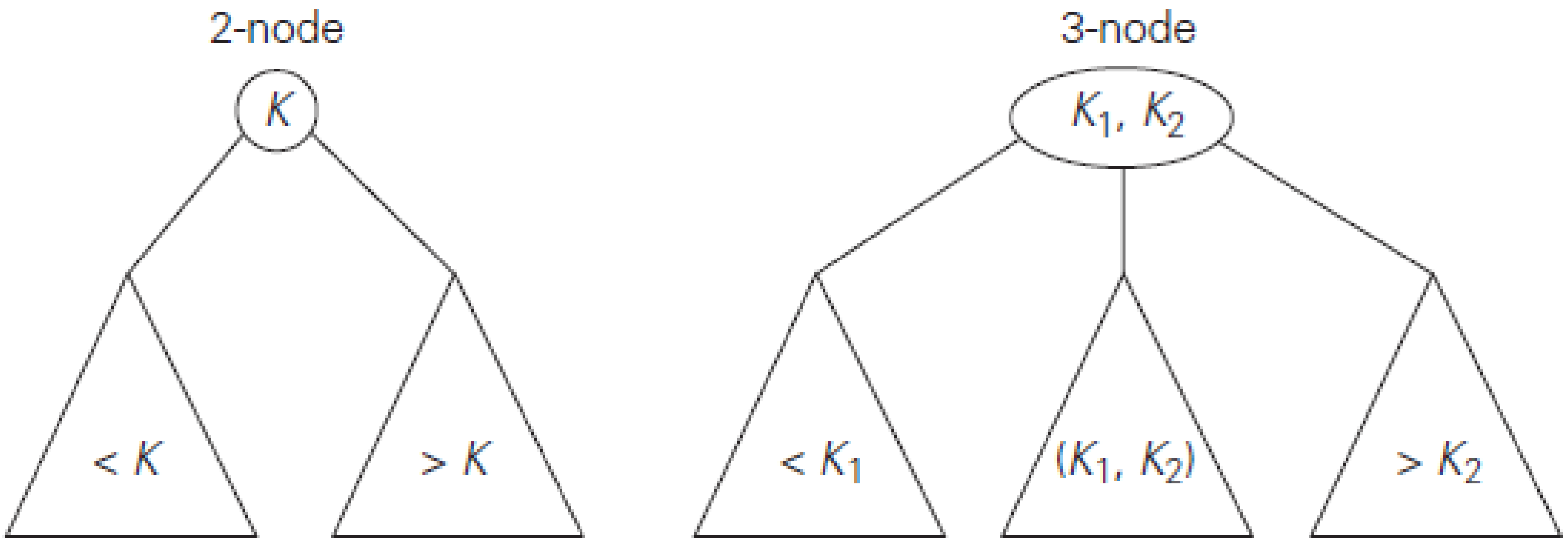
A ***2-3 tree*** is a tree that can have nodes of two kinds: 2-nodes and 3-nodes.

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

Transform and Conquer

2-3 tree

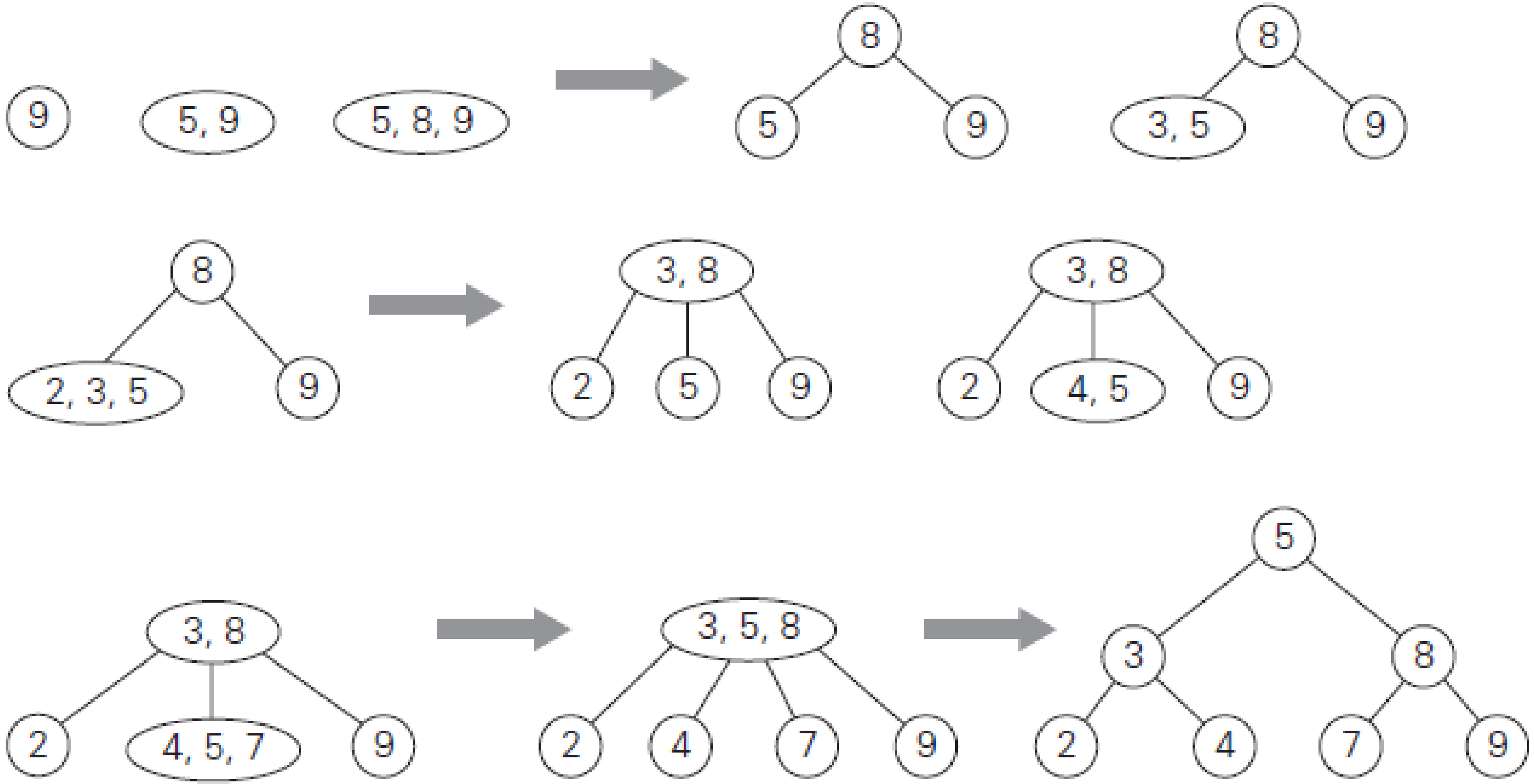


Two kinds of nodes of a 2-3 tree.

Transform and Conquer

2-3 tree

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



Transform and Conquer

Heap Sort

Space and time tradeoff

Time-Space Trade-Off

A tradeoff is a situation where one thing increases and another thing decreases. It is a way to solve a problem in:

- Either in less time and by using more space
- In very little space by spending a long amount of time.

Example: The fn fibonacci Numbers is defined by the recurrence relation:

$F_n = F_{n-1} + F_{n-2}$,

where, $F_0 = 0$ *and* $F_1 = 1$.

Space and time tradeoff

Time-Space Trade-Off

```
int Fibonacci(int N)
{
    // Base Case
    if (N < 2)

        return N;
    // Recursively computing the term
    // using recurrence relation
    return Fibonacci(N - 1) + Fibonacci(N - 2);
}
```

Sort by Counting

- This is no a in place sorting algorithm
- No comparison is done to sort the elements

```
ALGORITHM ComparisonCountingSort(A[0..n − 1])
//Sorts an array by comparison counting
//Input: An array A[0..n − 1] of orderable elements
//Output: Array S[0..n − 1] of A's elements sorted in nondecreasing order
for i ←0 to n − 1 do Count[i]←0
for i ←0 to n − 2 do
        for j ←i + 1 to n − 1 do
                if A[i]<A[j ]
                                Count[j ]←Count[j ]+ 1
                else Count[i]←Count[i]+ 1
for i ←0 to n − 1 do S[Count[i]]←A[i]
return S
```

$$C(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} [(n-1) - (i+1) + 1] = \sum_{i=0}^{n-2} (n-1-i) = \frac{n(n-1)}{2}.$$

Space and time tradeoff

The Naive String Matching Algorithm

- The naïve approach tests all the possible placement of Pattern P [1.......m] relative to text T [1......n].
- We try shift s = 0, 1.......n-m, successively and for each shift s. Compare T [s+1.......s+m] to P [1......m].
- The naïve algorithm finds all valid shifts using a loop that checks the condition

  P [1.......m] = T[s+1.......s+m] for each of the n - m +1 possible value of s.

```
NAIVE-STRING-MATCHER (T, P)
n ← length [T]
m ← length
for s ← 0 to n −m
      do if P [1.....m] = T [s + 1....s + m]
      then print "Pattern occurs with shift" s
```

Analysis: This for loop from 3 to 5 executes for n-m + 1(we need at least m characters at the end) times and in iteration we are doing m comparisons. So the total complexity is O (n-m+1).

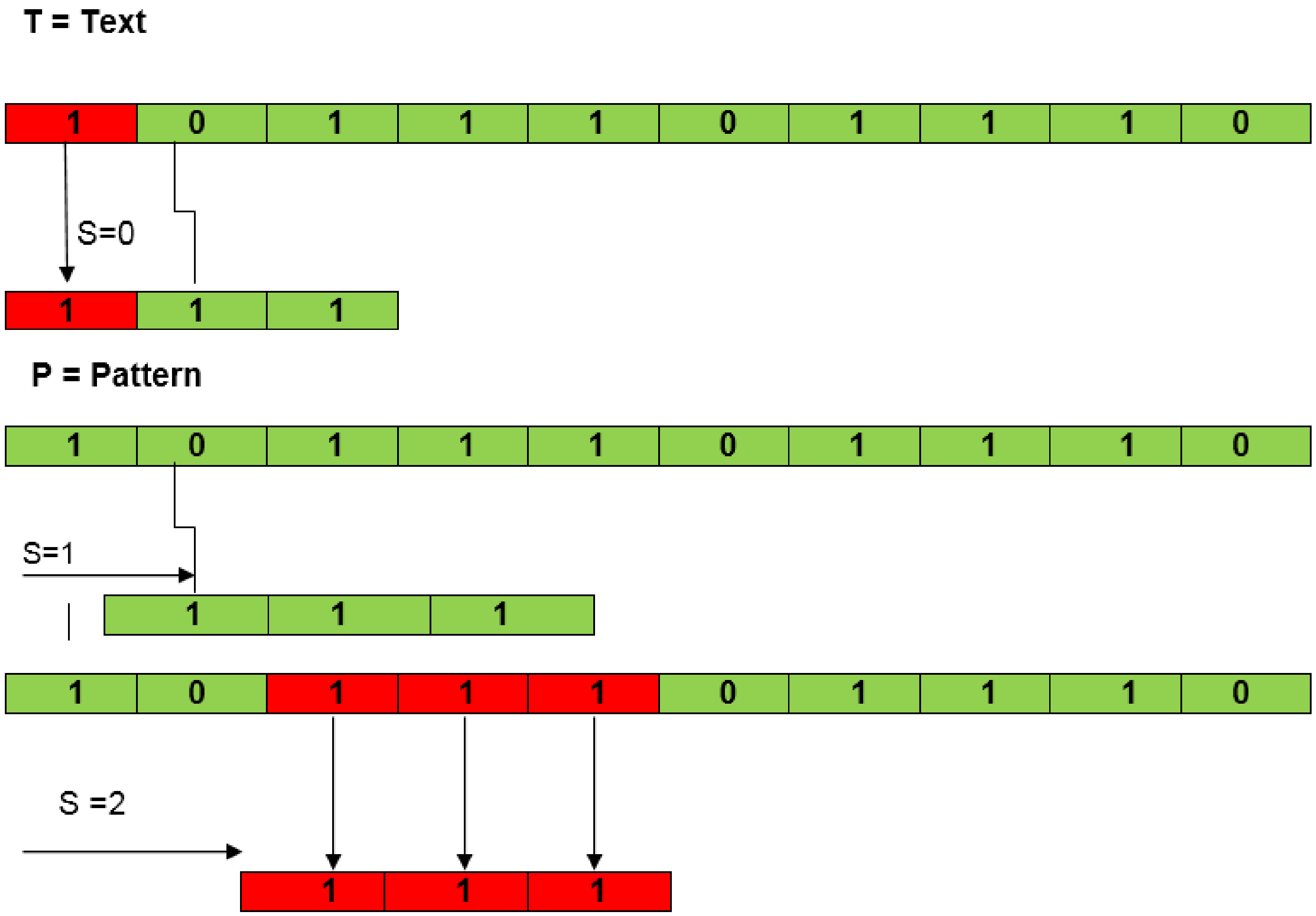
Space and time tradeoff

Example:

Suppose T = 1011101110

P = 111

Find all the Valid Shift



So, S=2 is a Valid Shift

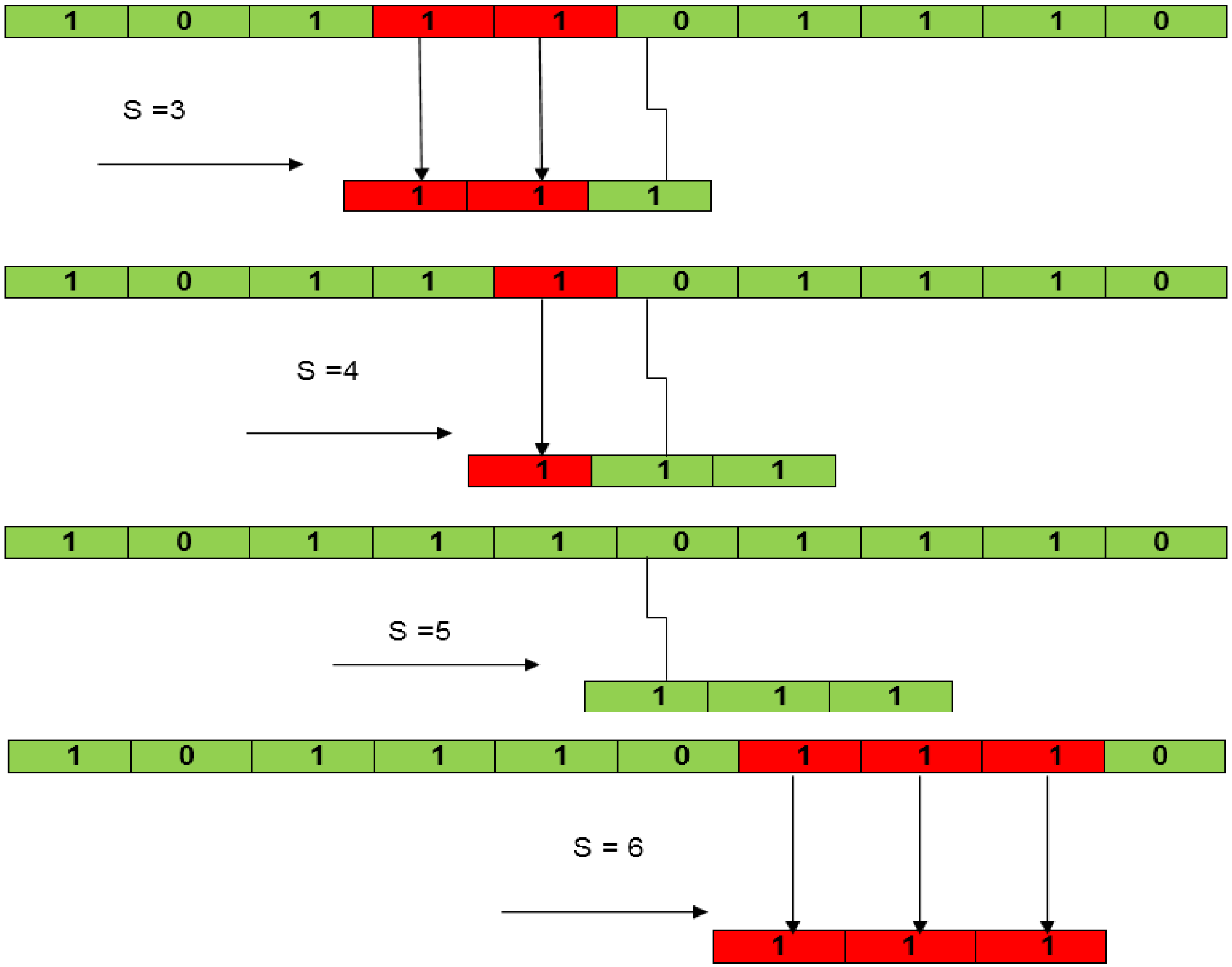
Space and time tradeoff

Example:

Suppose T = 1011101110

P = 111

Find all the Valid Shift



So, S=6 is a Valid Shift

Space and time tradeoff

Horspool String Matching Algorithm

- This algorithm is used to find he substring(pattern) in the given string.
- Here the shifts of the pattern is done more than one number of times in the shifting.

Shift Table

```
ALGORITHM ShiftTable(P [0..m − 1])
//Fills the shift table used by Horspool's and Boyer-Moore algorithms
//Input: Pattern P[0..m − 1] and an alphabet of possible characters
//Output: Table[0..size − 1] indexed by the alphabet's characters and
// filled with shift sizes computed by formula
for i ←0 to size − 1 do
      Table[i]←m
for j ←0 to m − 2 do
      Table[P[j ]]←m − 1− j
return Table
```

Space and time tradeoff

Horspool String Matching Algorithm

LEARNING Length=8

Shift Table of Learning

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	
5	8	8	8	6	8	8	8	2	8	8	7	8	1	8	8	8	4	8	8	8	8	8	8	8	8	8

M	A	C	H	I	N	E		L	E	A	R	N	I	N	G		T	E	C	H	N	I	Q	U	E	

L	E	A	R	N	I	N	G

Space and time tradeoff

Horspool String Matching Algorithm

R I N G Length=4

Shift Table of Learning

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	
4	4	4	8	4	4	4	4	2	4	4	4	4	1	4	4	4	3	4	4	4	4	4	4	4	4	4

C	O	M	P	U	T	E	R		S	C	I	E	N	C	E		A	N	D		E	N	G	I	N	E	E	R	I	N	G

Space and time tradeoff

Horspool String Matching Algorithm

```
ALGORITHM HorspoolMatching(P [0..m − 1], T [0..n − 1])
//Implements Horspool's algorithm for string matching
//Input: Pattern P[0..m − 1] and text T [0..n − 1]
//Output: The index of the left end of the first matching substring or −1 if
there are no matches ShiftTable(P [0..m − 1]) //generate Table of shifts

i ←m − 1 //position of the pattern's right end
while i ≤ n − 1 do
    k←0 //number of matched characters
    while k ≤ m − 1 and P[m − 1− k]= T [i − k] do
        k←k + 1
    if k = m
        return i − m + 1
    Else
        i ←i + Table[T [i]]
return −1
```

Space and time tradeoff

Boyers Moor Algorithm (String Matching)

We need to compute the Bad Shift Table and Good Suffix Table before doing the String matching.

Example for

B	A	O	B	A	B

Bad shift table is

B	A	O	_	Other
2	1	3	6	6

We need to find the value of d1

Where d1=t(c)-k

t(c) is the value of bad shift table

k is the number of matching character

Space and time tradeoff

Boyers Moor Algorithm (String Matching)

Good Suffix table

B	A	O	B	A	B

K	Pattern	No Shifts d2	
1	B A O B A B	2	B is present in the pattern
2	B A O B A B	5	AB is not present but B is Present, but A is suffix of B so take the next occurrence
3	B A O B A B	5	BAB is not present but AB is not Present, B is present, no suffix B is there so take B next occurrence

d2={max(t(c), 1}

Space and time tradeoff

Boyers Moor Algorithm (String Matching)

Bad Shift				
B	A	O	_	Other
2	1	3	6	6

Good Suffix Shift		
K	Pattern	d2
1	B A O B A B	2
2	B A O B A B	5
3	B A O B A B	5

Shift value=max(d2,d1)

B	E	S	S	_	K	N	E	W	_	A	B	O	U	T	_	B	A	O	B	A	B

B	A	O	B	A	B