DESIGN AND ANALYSIS OF ALGORITHMS (DAA)

Radhika Chapaneri

Sem V

Knapsack problem

Given some items, pack the knapsack to get the maximum total value. Each item has some weight and some value. Total weight that we can carry is no more than some fixed number W.
So we must consider weights of items as well as their values.

Item #	Weight	Value
1	1	8
2	3	6
3	5	5

Knapsack problem

There are two versions of the problem:

1. "0-1 knapsack problem"
   - Items are indivisible; you either take an item or not. Some special instances can be solved with *dynamic programming*

2. "Fractional knapsack problem"
   - Items are divisible: you can take any fraction of an item

0-1 Knapsack problem

- Given a knapsack with maximum capacity *W*, and a set *S* consisting of *n* items
- Each item *i* has some weight w_i and benefit value b_i (all w_i and *W* are integer values)
- Problem: How to pack the knapsack to achieve maximum total value of packed items?

0-1 Knapsack problem

- Problem, in other words, is to find

$$\max \sum_{i \in T} b_i \text{ subject to } \sum_{i \in T} w_i \leq W$$

- The problem is called a "*0-1*" problem, because each item must be entirely accepted or rejected.

Recursive Formula for subproblems (continued)

Recursive formula for subproblems:

$$V[k,w] = \begin{cases} V[k-1,w] & \text{if } w_k > w \\ \max\{V[k-1,w], V[k-1,w-w_k] + b_k\} & \text{else} \end{cases}$$

0-1 Knapsack Algorithm

```
for w = 0 to W
  V[0,w] = 0
for i = 1 to n
  V[i,0] = 0
for i = 1 to n
  for w = 0 to W
        if w_i <= w // item i can be part of the solution
              if b_i + V[i-1,w-w_i] > V[i-1,w]
                    V[i,w] = b_i + V[i-1,w- w_i]
              else
                    V[i,w] = V[i-1,w]
        else V[i,w] = V[i-1,w]  // w_i > w
```

Running time

```
for w = 0 to W
  V[0,w] = 0
for i = 1 to n
  V[i,0] = 0
for i = 1 to n
  for w = 0 to W
        < the rest of the code >
```

O(W)

Repeat *n* times

O(W)

What is the running time of this algorithm?

O(n*W)

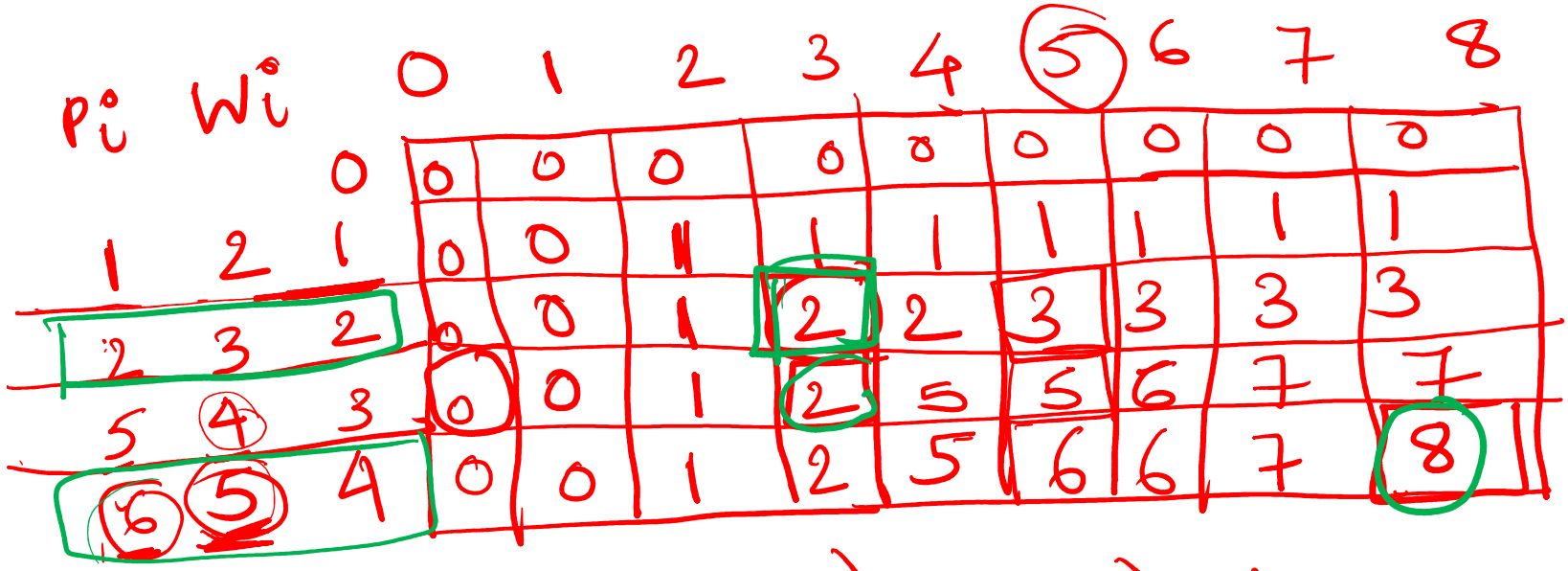
Remember that the brute-force algorithm takes $O(2^n)$

W/M = 8

N = 4

Pi / bi = {1, 2, 5, 6}

wi = {2, 3, 4, 5}



$V(k,w) = V(k-1,w)$ if $w_k > w$

$\max\{V(k-1,w), V(k-1, w-w_k) + b_k\}$ else

$V(4,8) = x_1\ x_2\ x_3\ x_4$

$= 0\ 1\ 0\ 1$

Max profit = 8

Remaining = 8 − 6 = 2

2 − 2 = 0

Example

Let's run our algorithm on the following data:

n = 4 (# of elements)
W = 5 (max weight)
Elements (weight, benefit):
(2,3), (3,4), (4,5), (5,6)

Example (18)

Items:

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	**7**

i=4

b_i=6

w_i=5

w= 5

w- w_i=0

if w_i <= w // item i can be part of the solution
    if b_i + V[i-1,w-w_i] > V[i-1,w]
        V[i,w] = b_i + V[i-1,w- w_i]
    else
        V[i,w] = V[i-1,w]
else V[i,w] = V[i-1,w] // w_i > w

Multistage graph

- Let G=(v,E) be a directed graph. In this we divide the problem into no. of stages or multiple stages then we try to solve whole problem.
- Multistage graph problem is to determine shortest path from source to destination. This can be solved by using either forward or backward approach.
- In forward approach we will find the path from destination to source, in backward approach we will find the path from source to destination.

Multistage graph

- A multistage graph G=(V,E) is a directed graph in which the vertices are partitioned into k>=2 disjoint sets Vi, i<=i<=k.
- The vertex s is source and t is the sink. Let c(i,j) be the cost of edge <i,j>.
- The cost of a path from s to t is the sum of costs of the edges on the path.
- The multistage graph problem is to find a minimum-cost path from s to t.

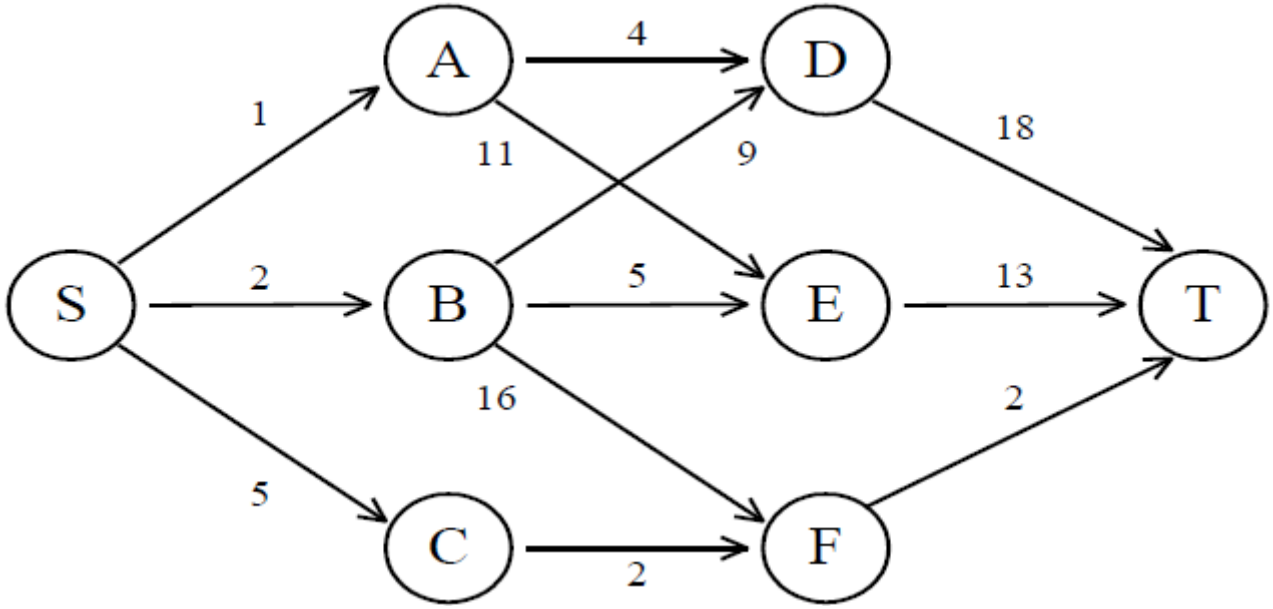
Multistage graph

- A dynamic programming formulation for a k-stage graph problem is obtained by first noticing that every s to t path is the result of a sequence of k-2 decisions.

- The ith decision invloves determining which vertex in Vi+1, 1<=i<=k-2, is on the path. It is easy to see that principal of optimality holds.
- Let p(i,j) be a minimum-cost path from vertex j in Vi to vertex t. Let cost(i,j) be the cost of this path.
- Using forward approach to find cost of the path

$$cost(i,j) = \min_{\substack{l \in V_{i+1} \\ \langle j,l \rangle \in E}} \{c(j,l) + cost(i+1,l)\} \qquad (5.5)$$

Multistage graph



- ■The greedy method can not be applied to this case:
- (S, A, D, T) 1+4+18 = 23.
- ■The real shortest path is:
- (S, C, F, T) 5+2+2 = 9.

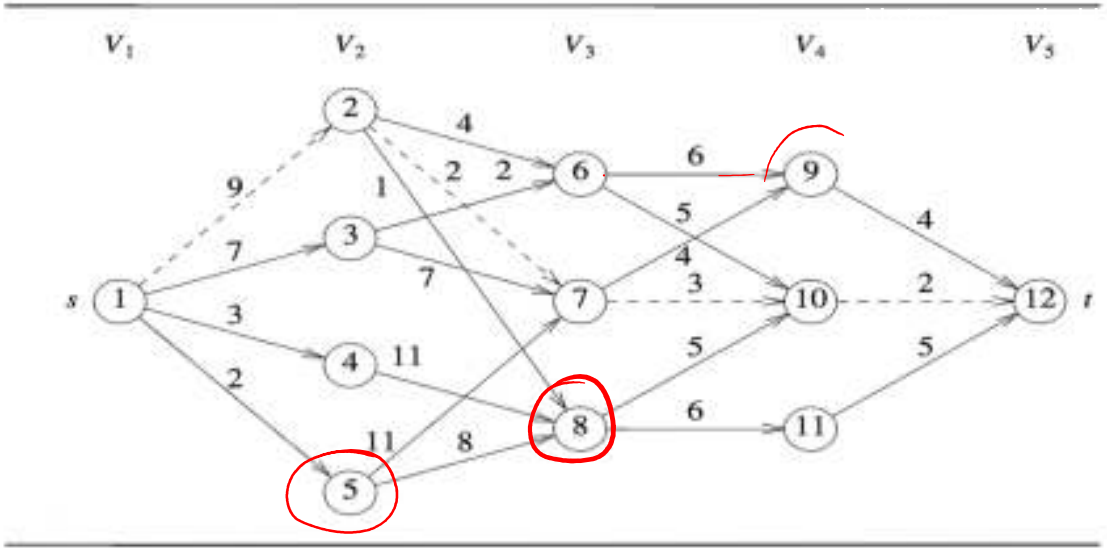


Figure 5.2 Five-stage graph

Stage, Vertex

$$cost(i,j) = \min_{\langle j,l\rangle \in E,\ l \in V_{i+1}} \{ c(j,l) + cost(i+1,l) \}$$

V	1	2	3	4	5	6	7	8	9	10	11	12
cost	16	7	9	18	15	7	5	7	4	2	5	0
d	2/3	7	6	8	8	10	10	10	12	12	12	12

$cost(5,12) = 0$

$cost(4,9) = 4$

$cost(4,10) = 2$

$cost(4,11) = 5$

$$cost(3,6) = \min \begin{cases} c(6,9) + cost(4,9) \\ c(6,10) + cost(4,10) \end{cases}$$

$$= \min(6+4, 5+2)$$

10, 7

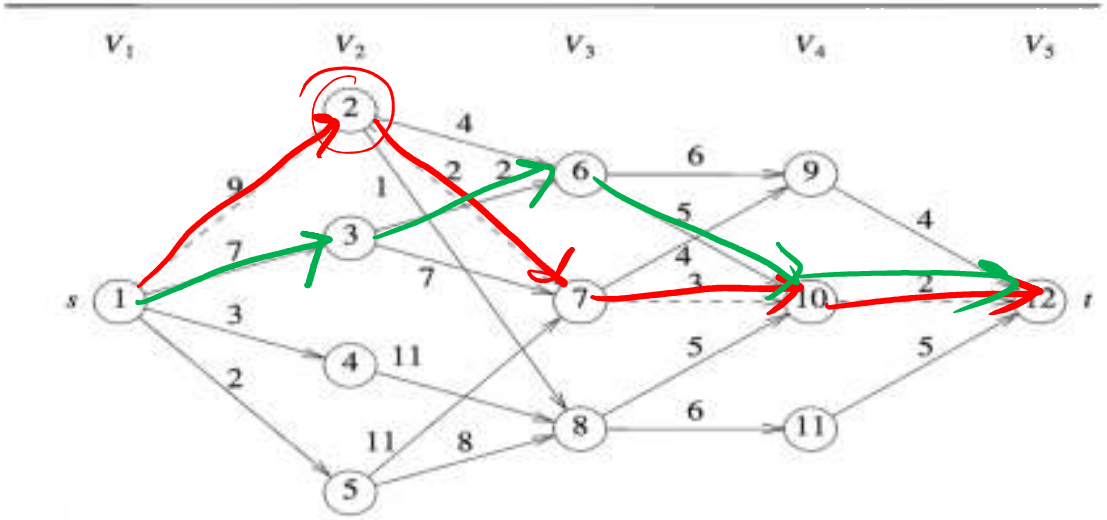


Figure 5.2 Five-stage graph

For finding path

$d(1,1) = 2$

$d(2,12) = 7$

$d(3,7) = 10$

$d(4,10) = 12$

$d(5,12) =$

$1-2-7-10-12$

$d(1,8) = 3$

$d(2,3) = 6$

$d(3,6) = 10$

$d(4,10) = 12$

Multistage graph

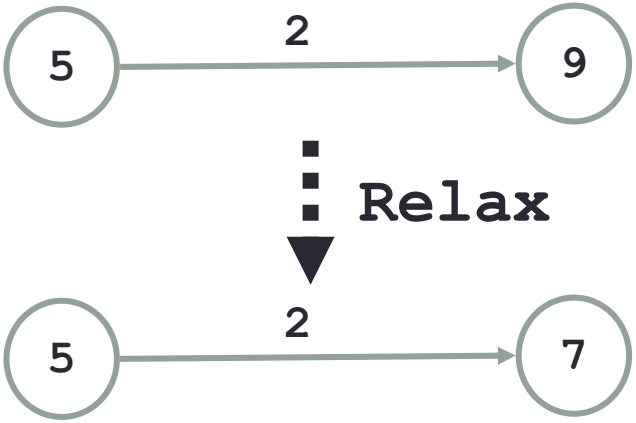
```
1   Algorithm FGraph(G, k, n, p)
2   // The input is a k-stage graph G = (V, E) with n vertices
3   // indexed in order of stages. E is a set of edges and c[i, j]
4   // is the cost of ⟨i, j⟩. p[1 : k] is a minimum-cost path.
5   {
6       cost[n] := 0.0;
7       for j := n − 1 to 1 step −1 do
8       { // Compute cost[j].
9           Let r be a vertex such that ⟨j, r⟩ is an edge
10          of G and c[j, r] + cost[r] is minimum;
11          cost[j] := c[j, r] + cost[r];
12          d[j] := r;
13      }
14      // Find a minimum-cost path.
15      p[1] := 1; p[k] := n;
16      for j := 2 to k − 1 do p[j] := d[p[j − 1]];
17  }
```

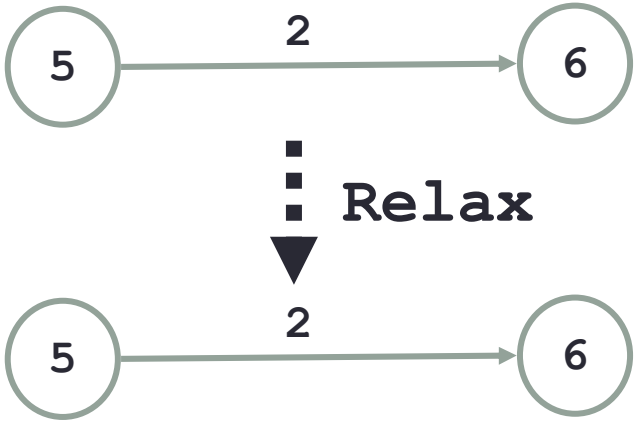
Algorithm 5.1 Multistage graph pseudocode corresponding to the forward approach

Bellman Ford Algorithm

- A key technique in shortest path algorithms is *relaxation*
  - Idea: for all *v*, maintain upper bound d[*v*] on δ(s,v)

```
Relax(u,v,w) {
     if (d[v] > d[u]+w) then d[v]=d[u]+w;
}
```





Bellman-Ford Algorithm

```
BellmanFord()
    for each v ∈ V
        d[v] = ∞;
    d[s] = 0;
    for i=1 to |V|-1
        for each edge (u,v) ∈ E
            Relax(u,v, w(u,v));
    for each edge (u,v) ∈ E
        if (d[v] > d[u] + w(u,v))
            return "no solution";
```

Initialize d[], which will converge to shortest-path value δ

Relaxation: Make |V|-1 passes, relaxing each edge

Test for solution
Under what condition do we get a solution?

```
Relax(u,v,w): if (d[v] > d[u]+w) then d[v]=d[u]+w
```

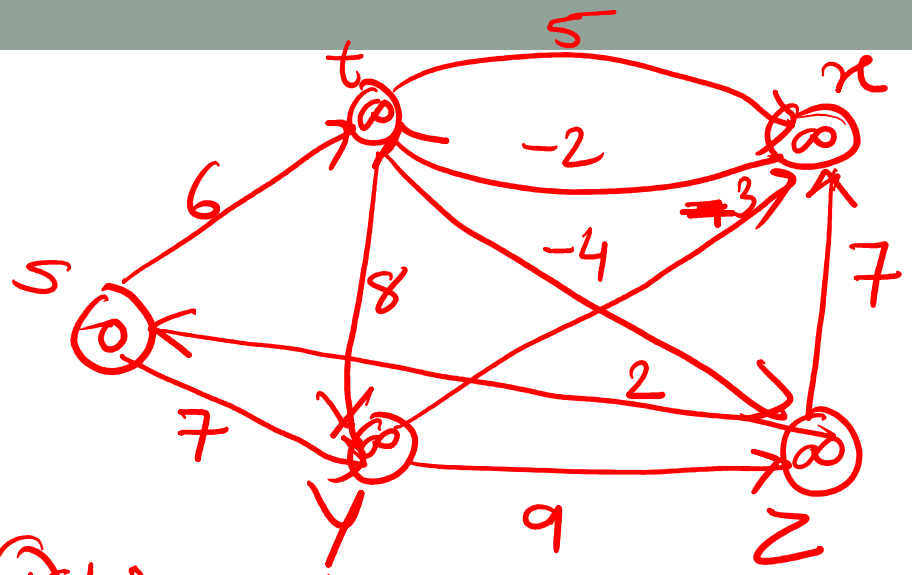
Bellman-Ford Algorithm

```
BellmanFord()
    for each v ∈ V
        d[v] = ∞;
    d[s] = 0;
    for i=1 to |V|-1
        for each edge (u,v) ∈ E
            Relax(u,v, w(u,v));
    for each edge (u,v) ∈ E
        if (d[v] > d[u] + w(u,v))
            return "no solution";
```

What will be the running time?

A: O(VE)

```
Relax(u,v,w): if (d[v] > d[u]+w) then d[v]=d[u]+w
```



$tx = 5$ $\quad zx = -7$

$ty = 8$ $\quad zs = 2$

$tz = -4$ $\quad st = 6$

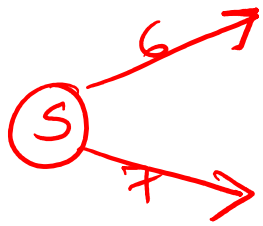
$xt = -2$ $\quad sy = 7$

$yx = -3$

$yz = 9$

1st iteration

	d	P
s	0	N
t	~~∞~~ 6	~~N~~ s
x	∞	N
y	~~∞~~ 7	~~N~~ s
z	∞	N

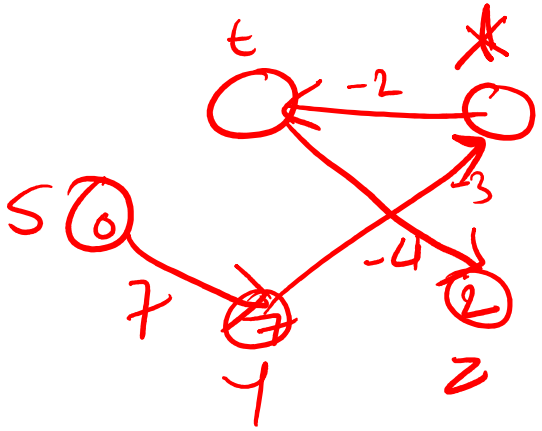


2nd iteration

	d	P
s	0	N
t	6	s
x	~~∞~~ ~~11~~ 4	~~N~~ y
y	7	s
z	~~∞~~ 2	~~N~~ t

3rd iteration

	d	P
s	0	N
t	~~6~~ 2	~~s~~ x
x	4	y
y	7	s
z	2	t



4th iteration

	d	P
s	0	~~x~~ N
t	2	x
x	4	y
y	7	s
z	z-2	t

sth iteration

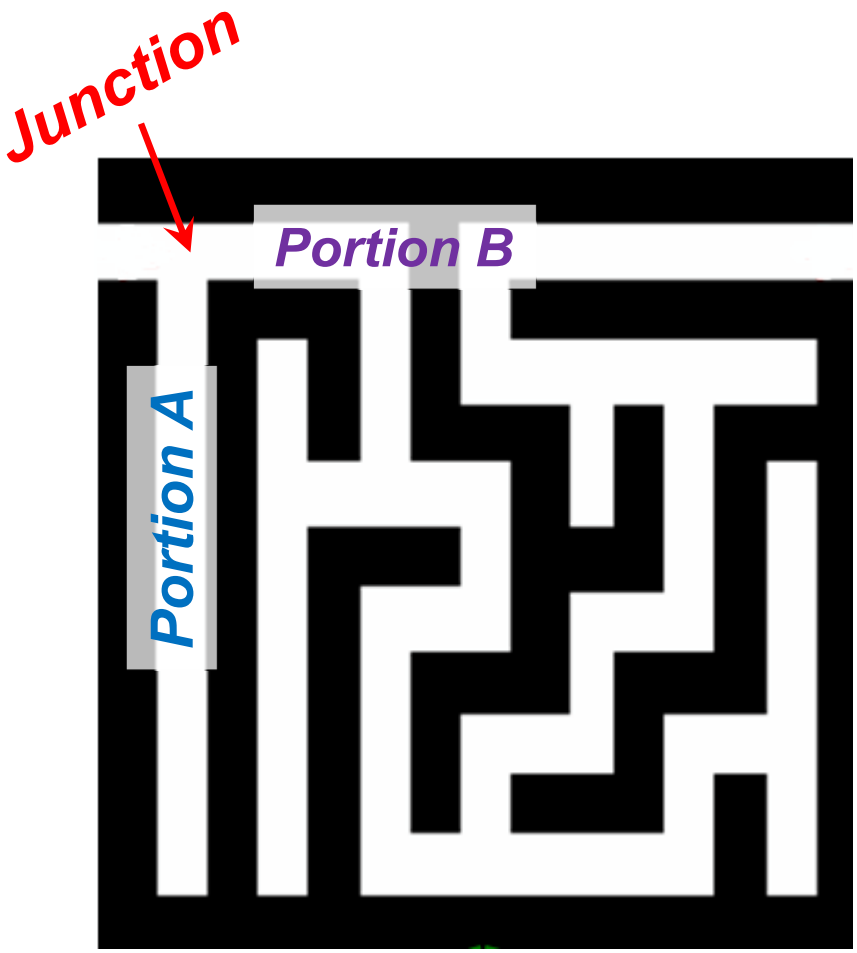
Backtracking

- Backtracking is a technique used to solve problems with a large search space, by systematically trying and eliminating possibilities.

- A standard example of backtracking would be going through a maze.
  - At some point in a maze, you might have two options of which direction to go:

Backtracking

- One strategy would be to try going through **Portion A** of the maze.
  - If you get stuck before you find your way out, then you ***"backtrack"*** to the junction.
- At this point in time you know that **Portion A** will ***NOT*** lead you out of the maze,
  - so you then start searching in **Portion B**

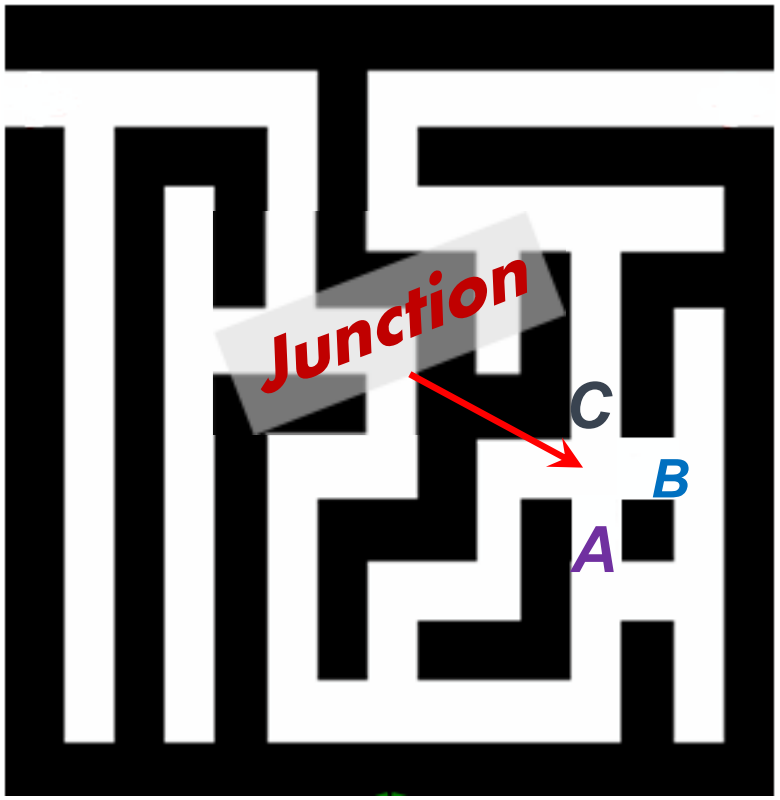


Backtracking

- Clearly, at a single junction you could have even more than 2 choices.

- The backtracking strategy says to try each choice, one after the other,
  - if you ever get stuck, ***"backtrack"*** to the junction and try the next choice.

- If you try all choices and never found a way out, then there IS no solution to the maze.

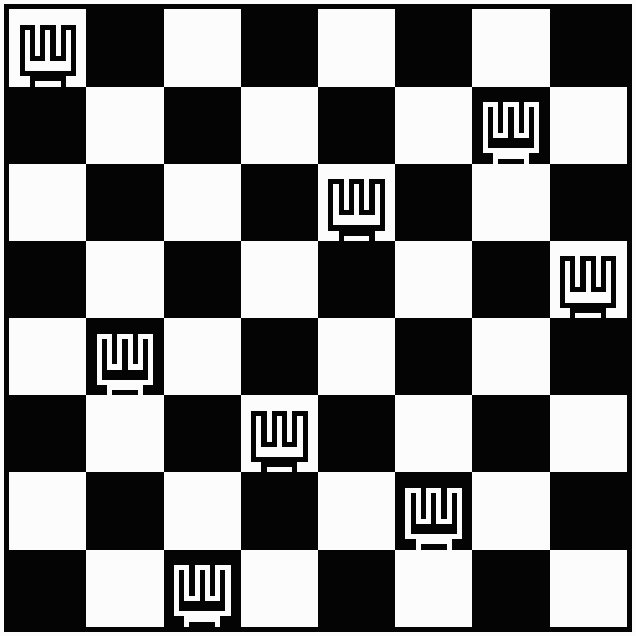


Backtracking – N Queens Problem

- Find an arrangement of **N** queens on a single chess board such that no two queens are attacking one another.

- In chess, queens can move all the way down any row, column or diagonal (so long as no pieces are in the way).

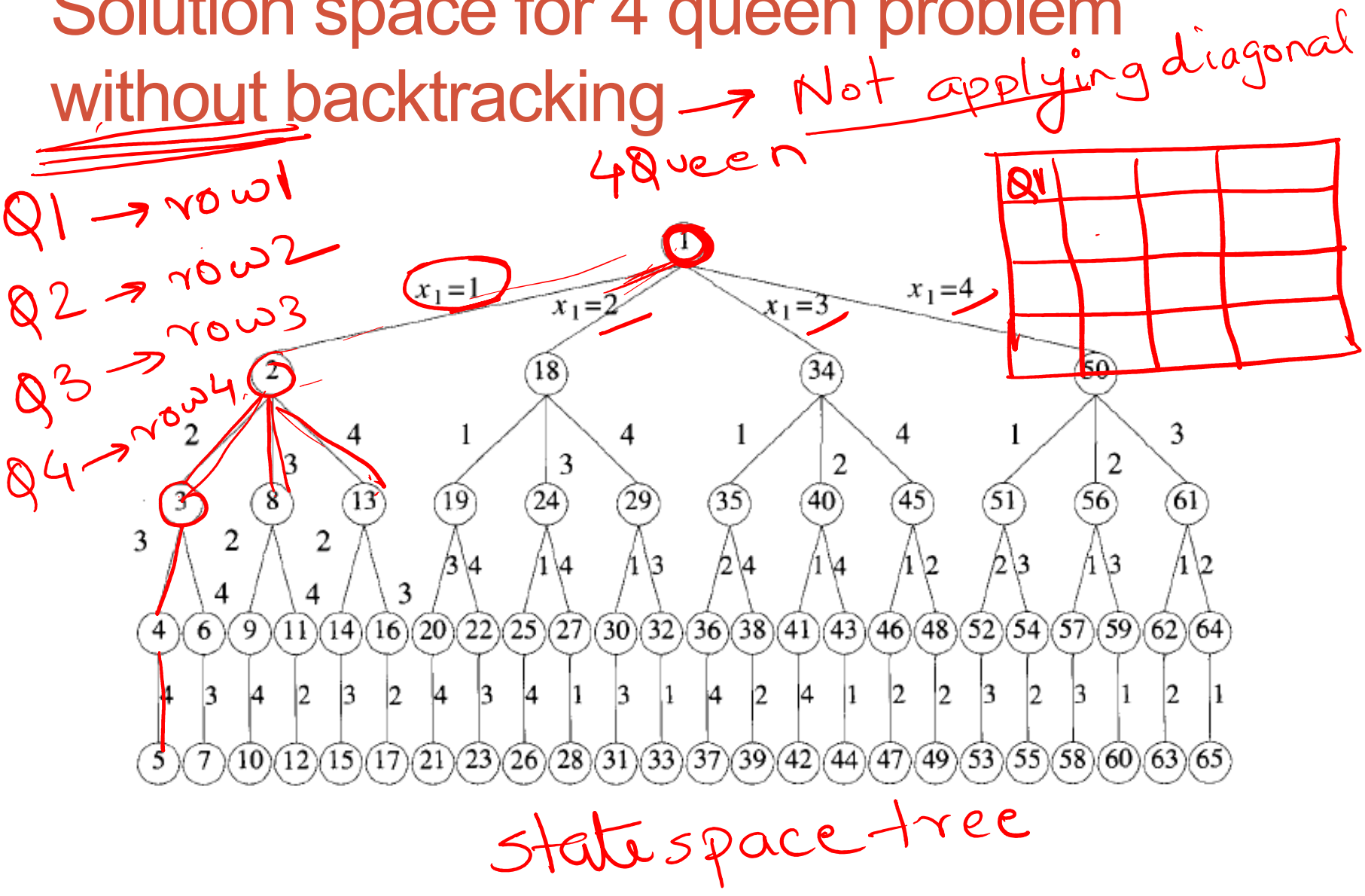
  - Due to this restrictions, it's clear that each row and column of the board will have exactly one queen.



N Queen Problem

- Let us number the rows and columns of the of the chessboard 1through n
- Since each queen is to be placed on different row we can assume queen i is to be placed on row i.
- All solutions to 8 queen problem can therefore be represented as (x1,x2…xn) where xi is the column on which queen i is to be placed
- For example for 4 queen problem solution will be represented as (x1,x2,x3,x4)

Solution space for 4 queen problem without backtracking

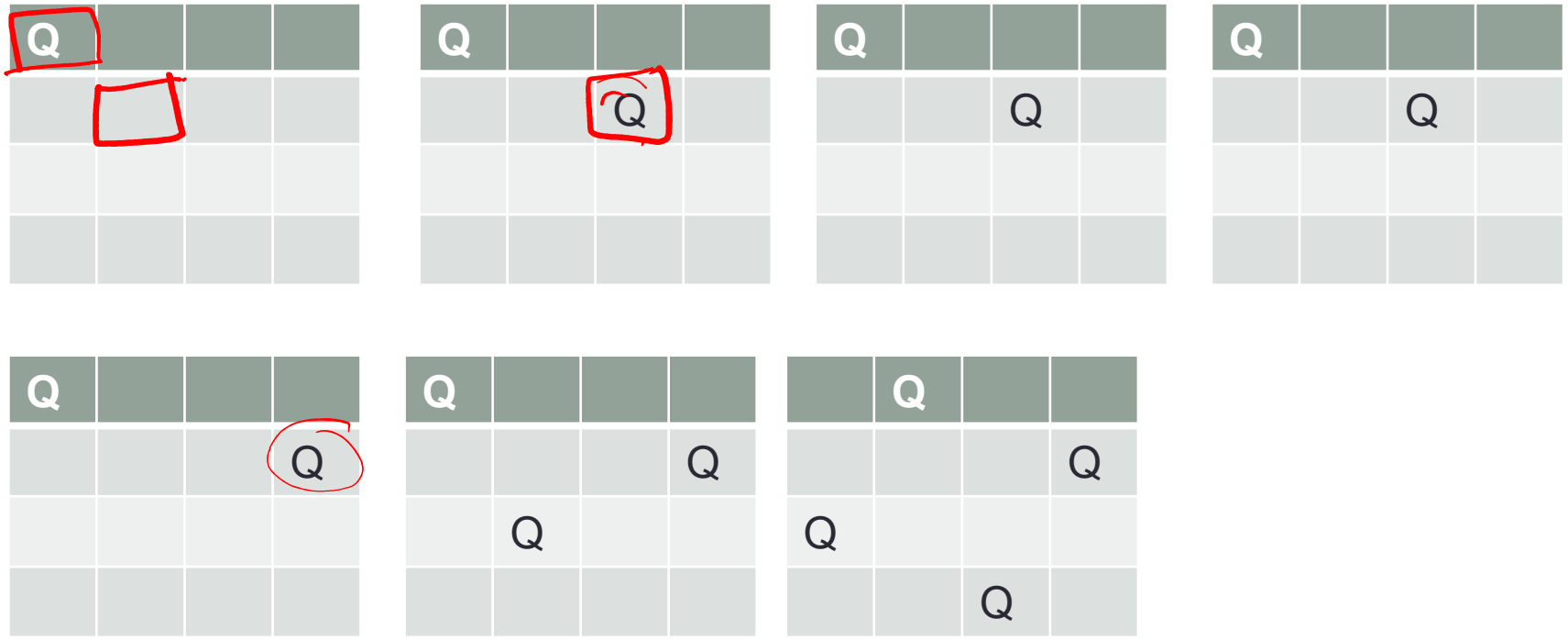


Backtracking – N Queens Problem

- The backtracking strategy is as follows:
  1) Place a queen on the first available square in row 1.
  2) Move onto the next row, placing a queen on the first available square there (that doesn't conflict with the previously placed queens).
  3) Continue in this fashion until either:
     a) you have solved the problem, or
     b) you get stuck.

     When you get stuck, remove the queens that got you there, until you get to a row where there is another valid square to try.

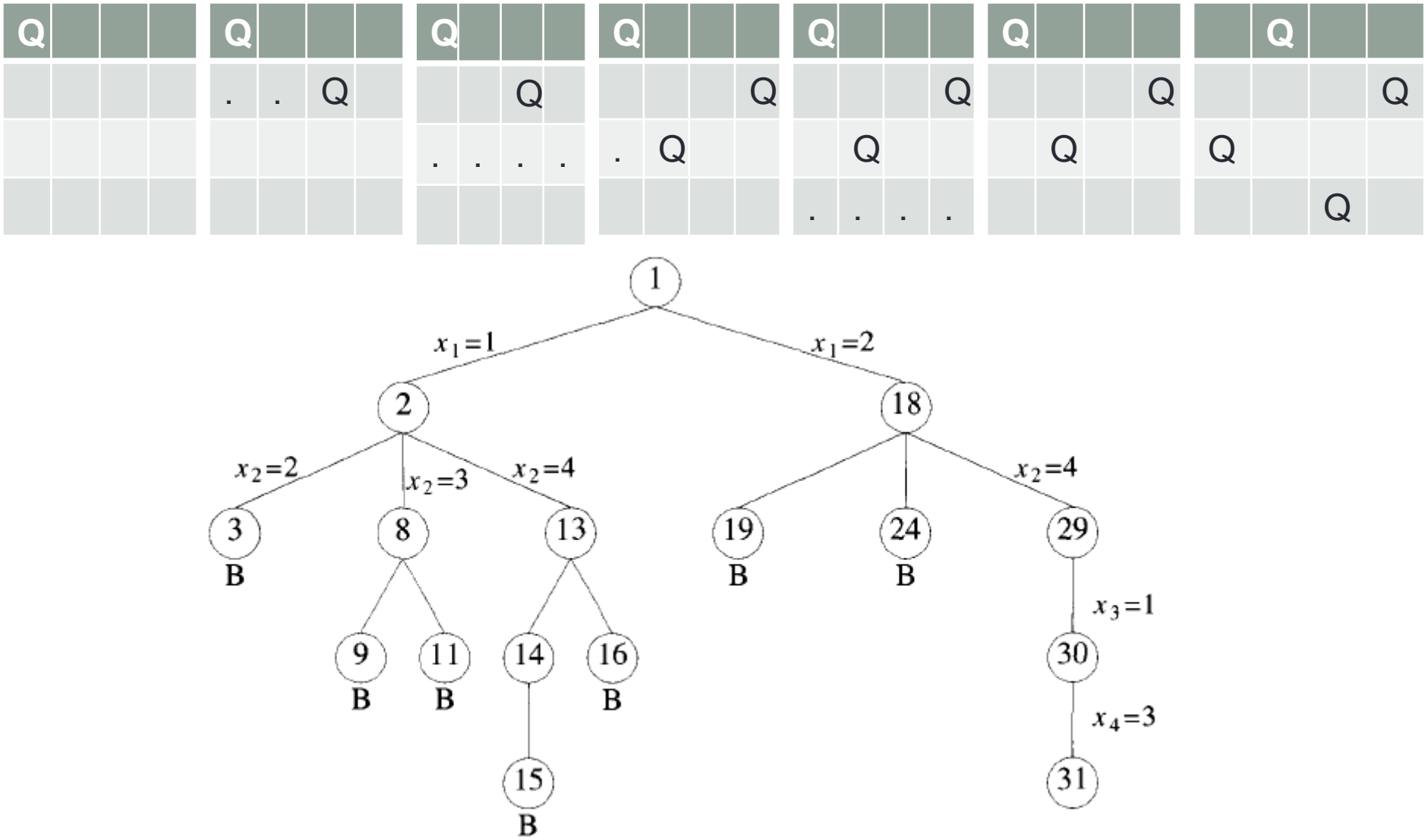
4 queen problem - Backtracking



- In the above tree edges are labelled by possible values of xi.
- Edges from level 1 to level 2 nodes specify the values for x1.
- Thus, the leftmost tree contain all solution with x1 = 1; its leftmost subttree contains all solutions with x1=1 and x2 =2 and so on
- The edges from level I to level i+1 are labelled with the values of xi.
- The solution space is defined by all paths from root to leaf node in the tree

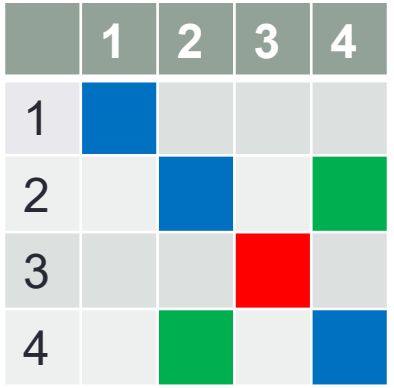
- The constraints for this problem are
- no two xi can be same i.e all queens must be on different column
- No two queen can be on the same diagonal.
- This realization reduces the solution space.

4 queen problem - Backtracking



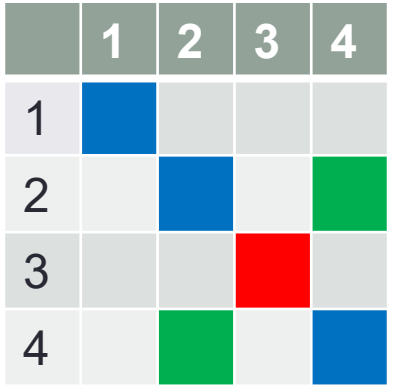
4 queen problem - Backtracking

- **Finding the diagonal**
- If we imagine the chessboard square as a two dimensional array a[ ][ ]
- Suppose two queens are placed at positions(i, j) and (k, l)
- Every element on the same diagonal that runs from upper left to lower right has same row – column value
- $i - j = k - l$
- $j - l = i - k$
- Every element on the same diagonal
- that runs from upper right to lower left
- has same row + column value



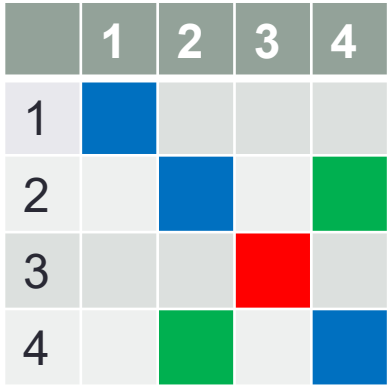
4 queen problem - Backtracking

- Every element on the same diagonal that runs from upper right to lower left has same row + column value
- i + j = k + l
- j – l = k – i
- Therefore two queens lie on the same diagonal if and only if |j – l| = |i-k|



- Place (k,i) returns a Boolean value that is true if the kth queen can be placed in column i.
- It tests both whether i is distinct from al previous values of x[1]…..,x[k-1]
- Whether there is no other queen on the same diagonal

```
1   Algorithm Place(k, i)
2   // Returns true if a queen can be placed in kth row and
3   // ith column. Otherwise it returns false. x[ ] is a
4   // global array whose first (k − 1) values have been set.
5   // Abs(r) returns the absolute value of r.
6   {
7       for j := 1 to k − 1 do
8           if ((x[j] = i) // Two in the same column
9               or (Abs(x[j] − i) = Abs(j − k)))
10                  // or in the same diagonal
11              then return false;
12      return true;
13  }
```



```
1    Algorithm NQueens(k, n)
2    // Using backtracking, this procedure prints all
3    // possible placements of n queens on an n × n
4    // chessboard so that they are nonattacking.
5    {
6        for i := 1 to n do
7        {
8            if Place(k, i) then
9            {
10               x[k] := i;
11               if (k = n) then write (x[1 : n]);
12               else NQueens(k + 1, n);
13           }
14       }
15   }
```

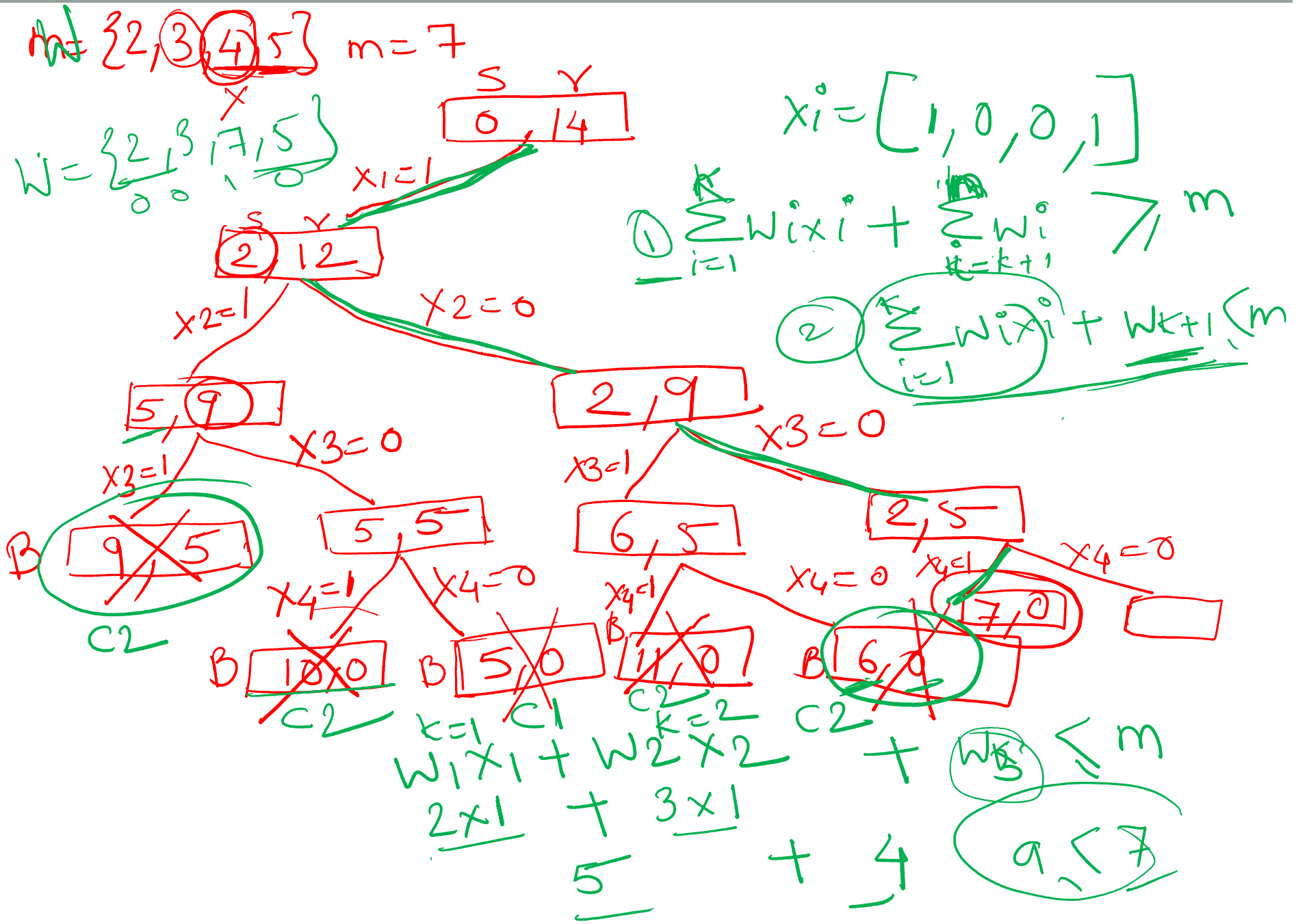
Sum of subset Problem

- Problem: Suppose we are given n distinct positive integers (usually called as weights) and we desire to find all combinations of these numbers whose sums are m
- For example:
- n = 4 , (w1, w2, w3 , w4) = (11,13,24, 7) and m = 31
- If the solution is represented by (x1, x2,…xn)
- Xi= 0 if wi is not chosen
- Xi =1 if wi is chosen
- The solution for the above problem is
- (1,1,0,1) = 11+13+7= 31
- (0,0,1,1) = 24+ 7 = 31

- The tree organization of solution is called

state space tree

Refer notebook for diagram


{2,3,4,5} m=7
W={2,3,7,5}
S Y
0 14
x1=1
2 12
x2=1
x2=0
5,9
2,9
x3=1
x3=0
9 5
5,5
6,5
2,5
x4=1
x4=0
10 0
5,0
11,0
7,0
6,0
C2
C1
B
k=1
k=2
W1X1 + W2X2
2x1 + 3x1
5
4
xi=[1,0,0,1]
① Σ Wixi + Σ Wi ≥ m
② Σ Wixi + Wk+1 < m
Wk ≤ m
9 < 7

- We assume that wi's are arranged in non decreasing order
- The bounding function or constraint (through this we will know when to use backtracking)are
- 1. $$\sum_{i=1}^{k} w_i x_i + \sum_{i=k+1}^{n} wi \geq m$$

Total sum still there

till now

The total sum + still there is not greater than m, we will never get the solution

- The second bounding function or constraint is

$$\sum_{i=1}^{k} w_i x_i + w_{k+1} < m$$

- Since in increasing order if addition of next element is giving sum greater than m,then there is no point in going further
- Sum is $s = \sum_{i=1}^{k-1} w_i x_i$
- Remaining is $r = \sum_{i=k}^{n} w_i$

Algorithm

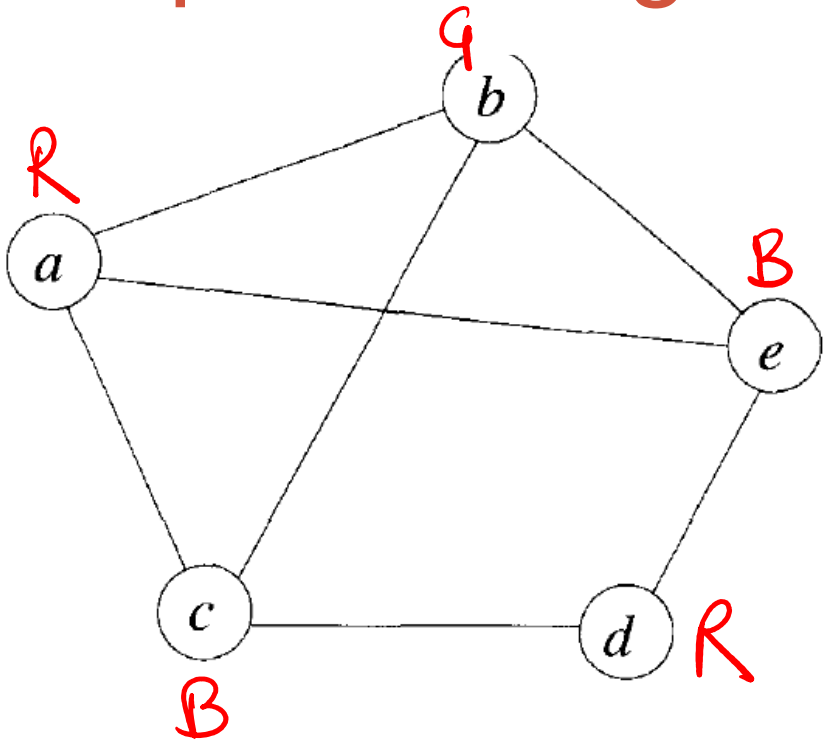
```
1   Algorithm SumOfSub(s, k, r)
2   // Find all subsets of w[1 : n] that sum to m. The values of x[j],
3   // 1 ≤ j < k, have already been determined. s = Σ_{j=1}^{k-1} w[j] * x[j]
4   // and r = Σ_{j=k}^{n} w[j]. The w[j]'s are in nondecreasing order.
5   // It is assumed that w[1] ≤ m and Σ_{i=1}^{n} w[i] ≥ m.
6   {
7       // Generate left child. Note: s + w[k] ≤ m since B_{k-1} is true.
8       x[k] := 1;
9       if (s + w[k] = m) then write (x[1 : k]); // Subset found
10          // There is no recursive call here as w[j] > 0, 1 ≤ j ≤ n.
11      else if (s + w[k] + w[k + 1] ≤ m)
12          then SumOfSub(s + w[k], k + 1, r − w[k]);
13      // Generate right child and evaluate B_k.
14      if ((s + r − w[k] ≥ m) and (s + w[k + 1] ≤ m)) then
15      {
16          x[k] := 0;
17          SumOfSub(s, k + 1, r − w[k]);
18      }
19  }
```

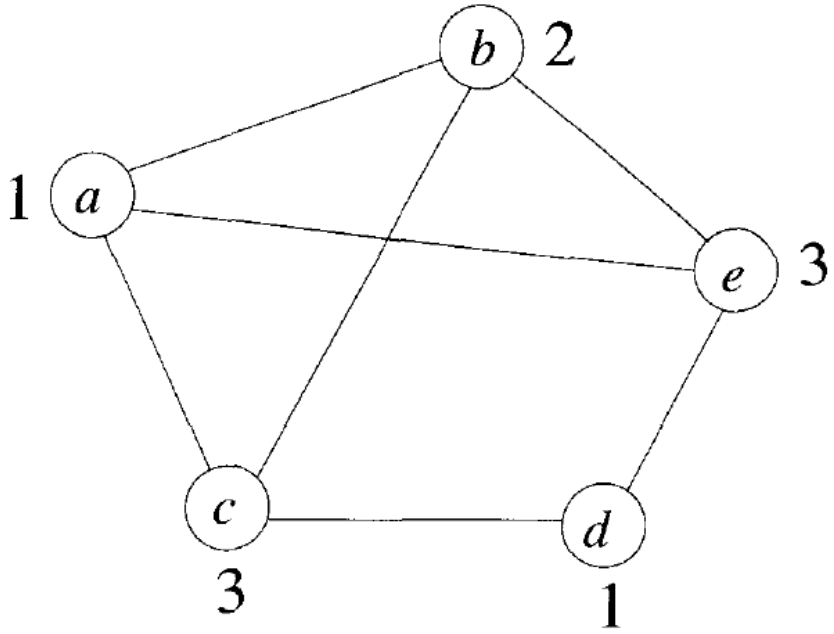
- For example wi ={2,3,4,5} m =7
- solution Tree for above problem refer example

Graph Coloring

- Let G be a graph and m be a given positive integer
- We want to discover whether the nodes of G can be colored in such a way that no two adjacent nodes have the same color yet only m colors are used.
- This is termed as m-colorability decision problem
- The m – colorabilty optimization problem asks for the smallest integer m for which the graph G can be colored
- This integer is referred to as the chromatic number of the graph.

Graph Coloring

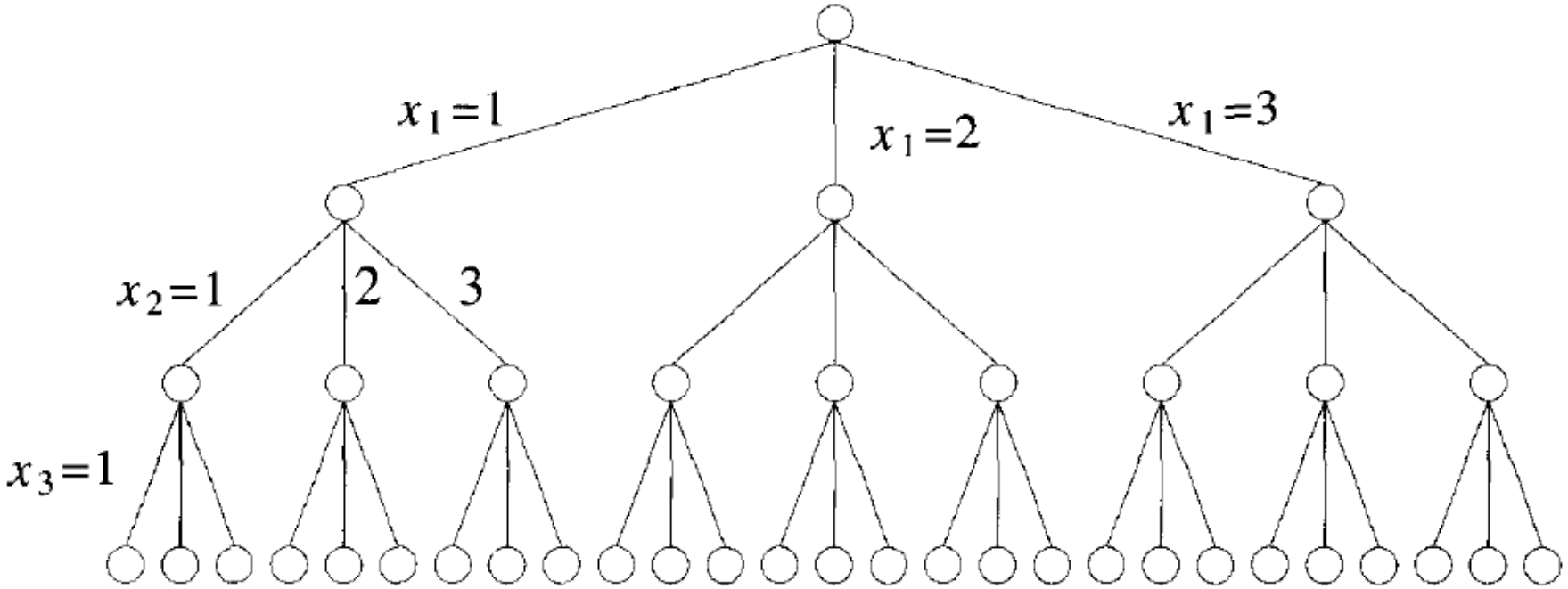




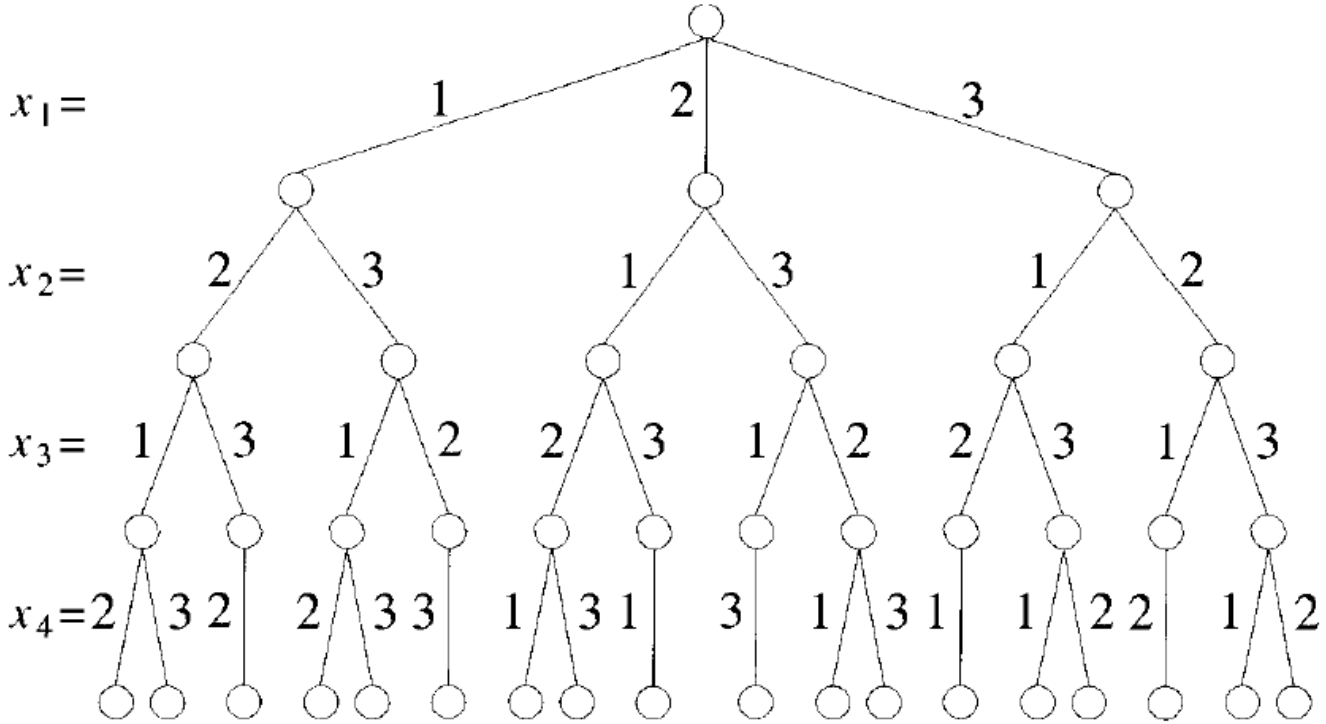
It can be seen that three colors are needed to color this graph and hence this graph's chromatic number is 3

- Suppose we represent a graph by adjacency matrix [1..n, 1..n] where G[I,j] = 1 if there is an edge of G and G[i,j]=0 otherwise.
- The colors are represented by the integers 1,2, … m and the solution is given by n tuple [x1, x2 …xn] where xi is the color of node i

Graph Coloring without backtracking

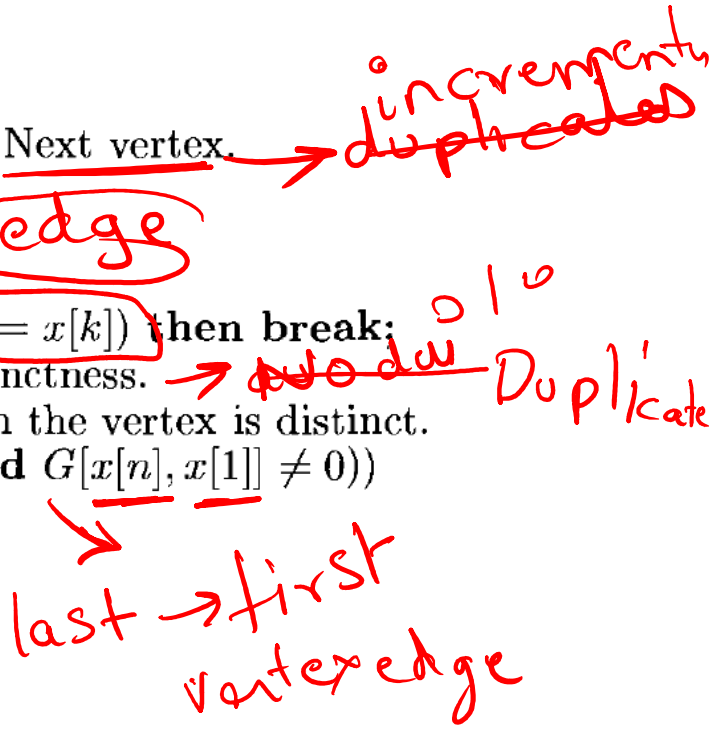


Graph Coloring with Backtracking



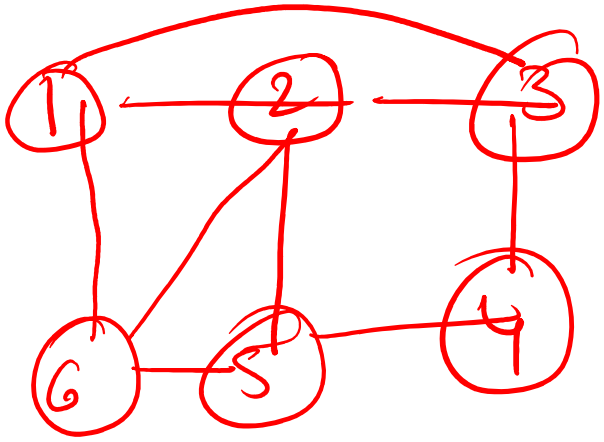
Hamiltonian cycle

```
1   Algorithm NextValue(k)
2   // x[1 : k − 1] is a path of k − 1 distinct vertices. If x[k] = 0, then
3   // no vertex has as yet been assigned to x[k]. After execution,
4   // x[k] is assigned to the next highest numbered vertex which
5   // does not already appear in x[1 : k − 1] and is connected by
6   // an edge to x[k − 1]. Otherwise x[k] = 0. If k = n, then
7   // in addition x[k] is connected to x[1].
8   {
9       repeat
10      {
11          x[k] := (x[k] + 1) mod (n + 1); // Next vertex.
12          if (x[k] = 0) then return;
13          if (G[x[k − 1], x[k]] ≠ 0) then
14          { // Is there an edge?
15              for j := 1 to k − 1 do if (x[j] = x[k]) then break;
16                                  // Check for distinctness.
17              if (j = k) then // If true, then the vertex is distinct.
18                  if ((k < n) or ((k = n) and G[x[n], x[1]] ≠ 0))
19                      then return;
20          }
21      } until (false);
22  }
```



```
1   Algorithm Hamiltonian(k)
2   // This algorithm uses the recursive formulation of
3   // backtracking to find all the Hamiltonian cycles
4   // of a graph. The graph is stored as an adjacency
5   // matrix G[1 : n, 1 : n]. All cycles begin at node 1.
6   {
7       repeat
8       { // Generate values for x[k].
9           NextValue(k); // Assign a legal next value to x[k].
10          if (x[k] = 0) then return;
11          if (k = n) then write (x[1 : n]);
12          else Hamiltonian(k + 1);
13      } until (false);
14  }
```

Hamiltonian cycle → Defined as a cycle that passes to all the vertices of a graph exactly once except the starting vertex & ending vertex that is same vertex



1 - 2 - 3 - 4 - 5 - 6 - 1

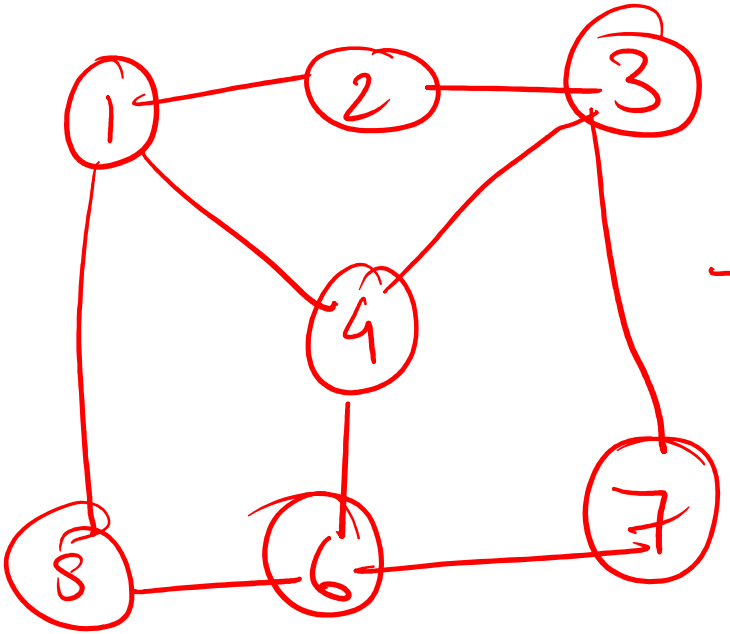
1 - 3 - 4 - 5 - 2 - 6 - 1

1 - 2 - 6 - 5 - 4 - 3 - 1

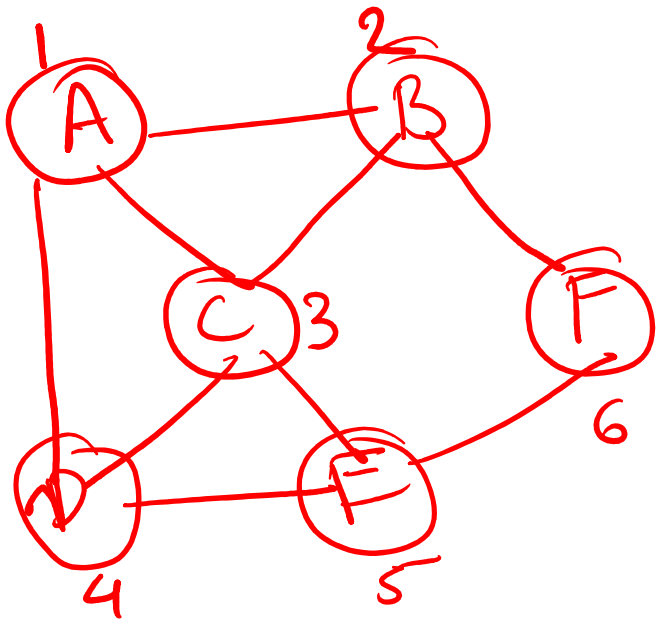
1 - 6 - ~~2 - 5 - 4 - 3~~ - 1

1 - 2 - ~~5 - 4 - 3 - 6~~ - 1 X

same cycle 2 - 3 - 4 - 5 - 6 - 1 - 2



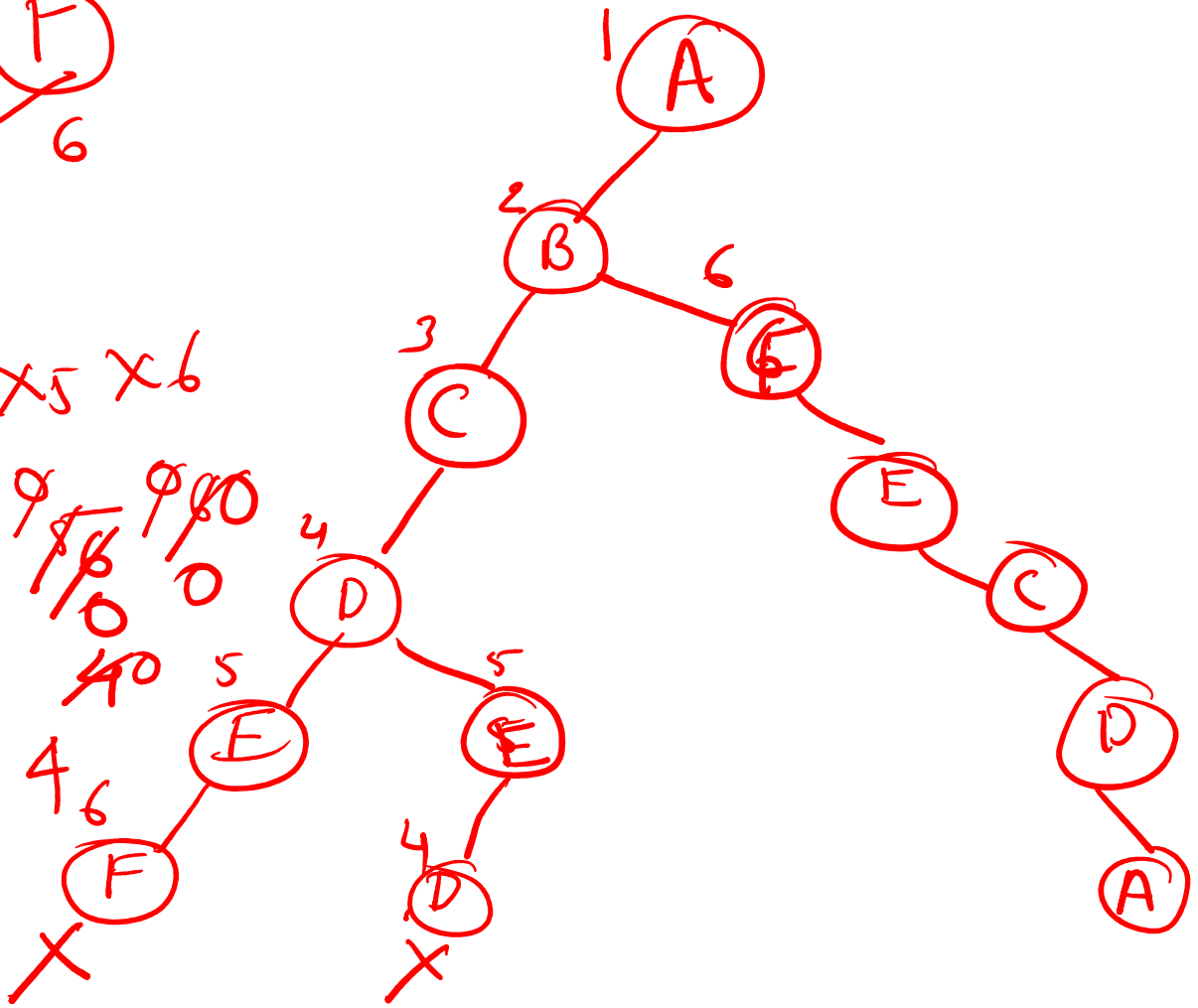
→ No hamiltonian cycle

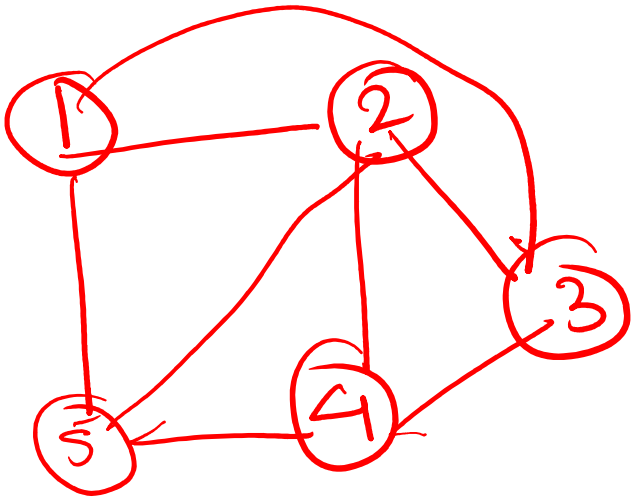


(1) No duplicate (2) Edge to previous vertex
(3) edge from last to 1st vertex

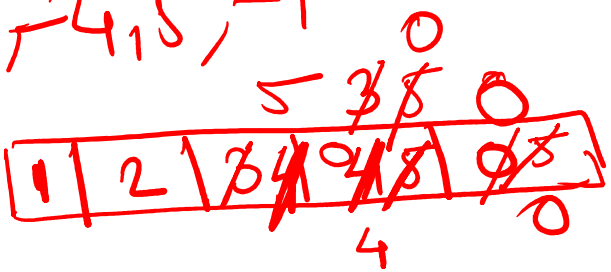
X_1 X_2 X_3 X_4 X_5 X_6

1 2 6 5 3 4





1-2-3-4,5,-1



1,2,5,4,3,1

0,1,2,3,4,5

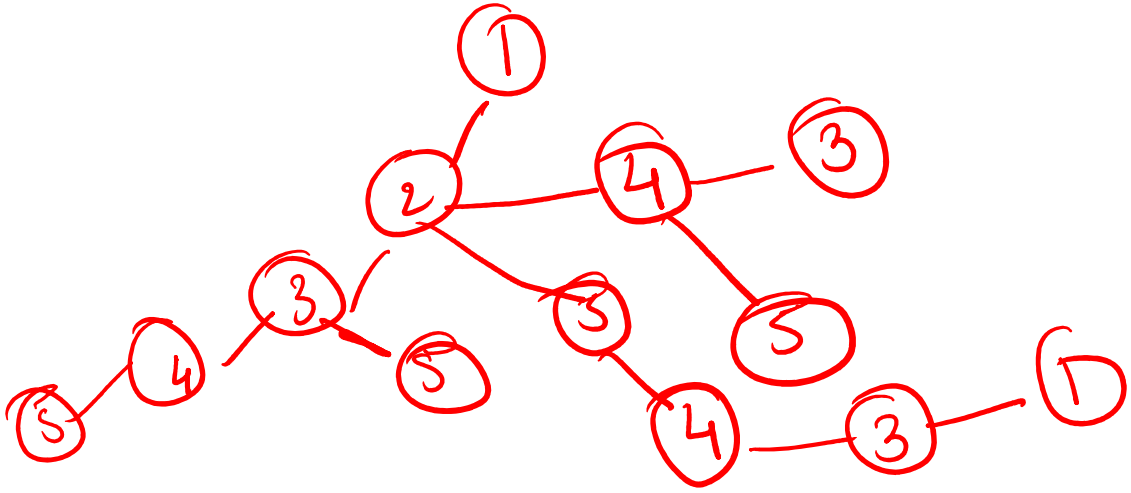
mod(6) mod(6+1)

① No duplicates

0,1,2,3,7

X

② Edge to previous vertex



③ last vertex to first vertex

Introduction

- What is *string matching*?
  - Finding all occurrences of a *pattern* in a given *text* (or *body of text*)
- Many applications
  - While using editor/word processor/browser
  - Login name & password checking
  - Virus detection
  - Header analysis in data communications
  - DNA sequence analysis

String-Matching Problem

- The *text* is in an array $T[1..n]$ of length n
- The *pattern* is in an array $P[1..m]$ of length m
- Elements of T and P are characters from a *finite alphabet* Σ
  - E.g., $\Sigma = \{0,1\}$ or $\Sigma = \{a, b, \ldots, z\}$
- Usually T and P are called *strings* of characters

String-Matching Problem …contd

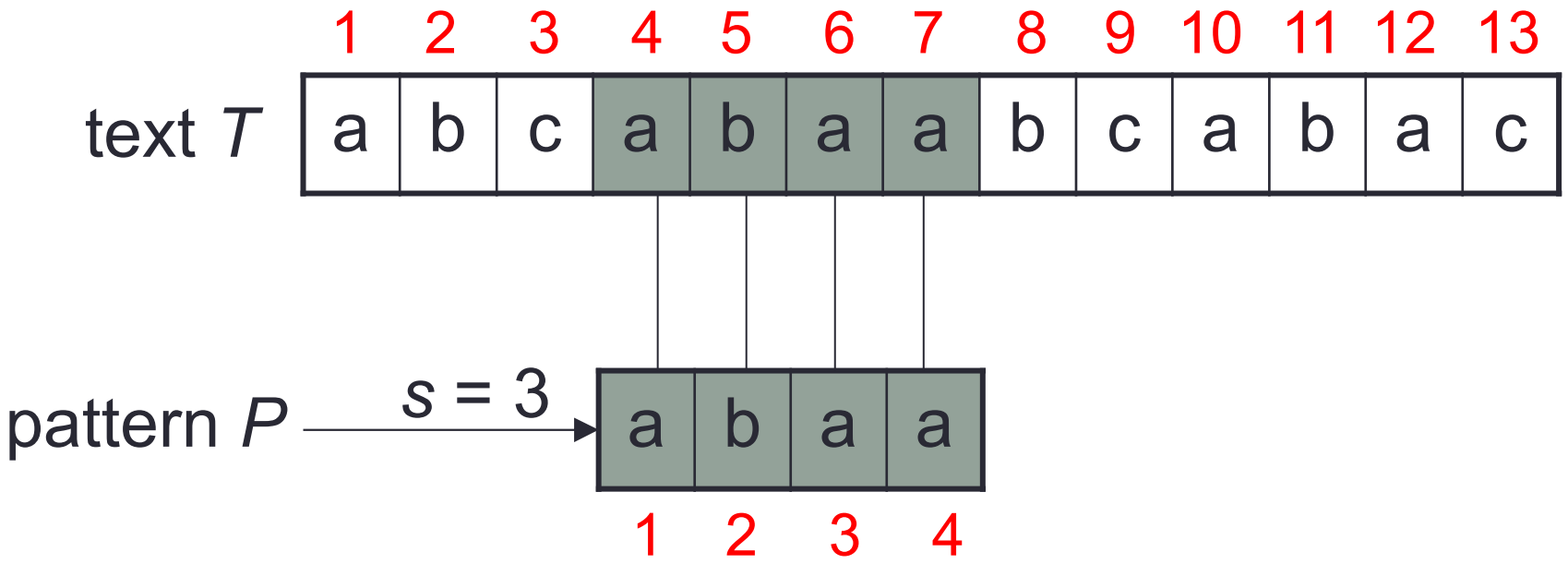
- We say that pattern *P occurs with shift s* in text *T* if:

  a) $0 \le s \le n-m$ and

  b) $T[(s+1)..(s+m)] = P[1..m]$

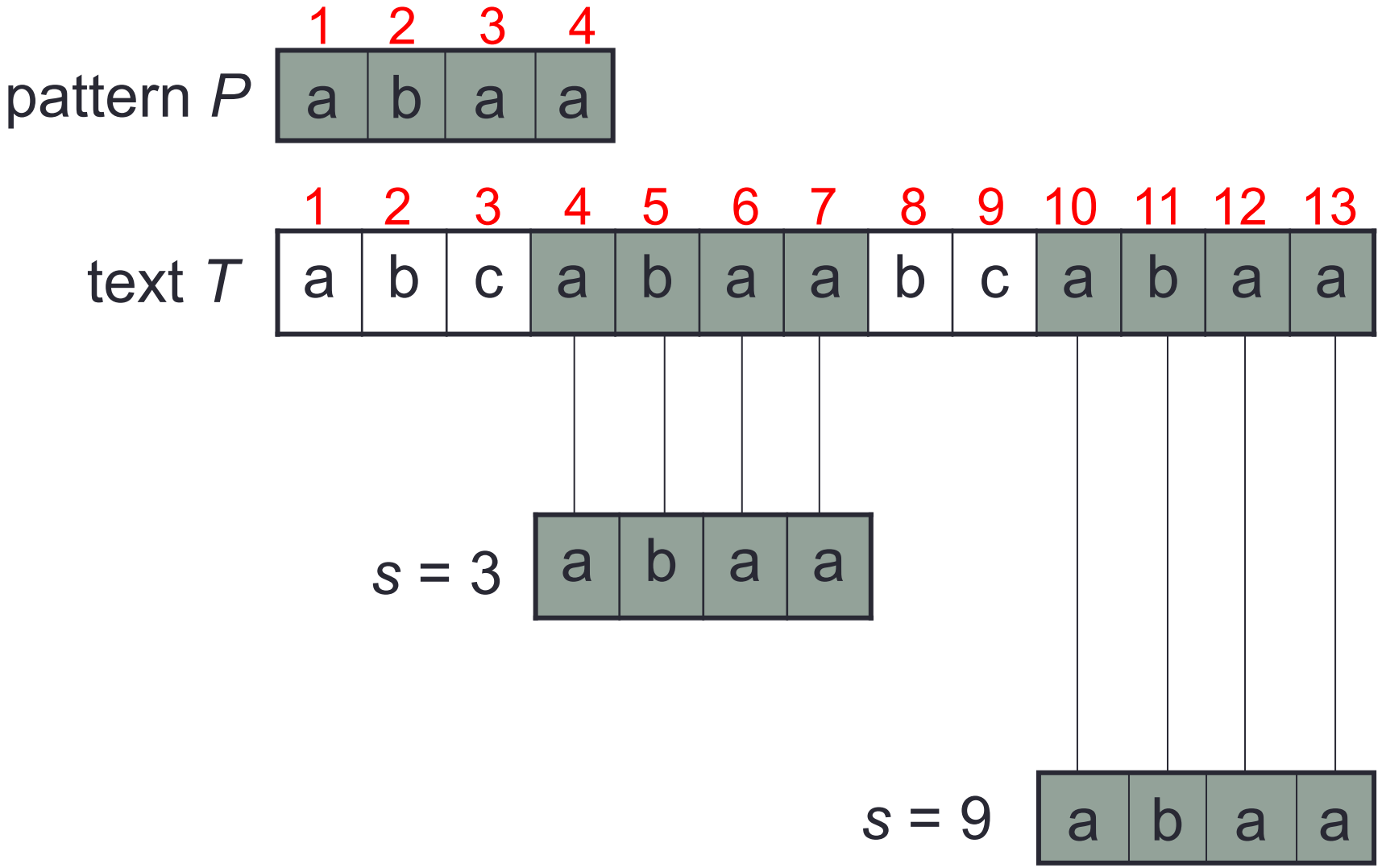
- If *P* occurs with shift *s* in *T*, then *s* is a *valid shift*, otherwise *s* is an *invalid shift*
- String-matching problem: finding all valid shifts for a given *T* and *P*

Example 1



shift $s = 3$ is a valid shift
(n=13, m=4 and $0 \leq s \leq n\text{-}m$ holds)

Example 2


1 2 3 4
pattern P
a b a a
1 2 3 4 5 6 7 8 9 10 11 12 13
text T
a b c a b a a b c a b a a
s = 3
a b a a
s = 9
a b a a

Naïve String-Matching Algorithm

- The naive 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 values of s.
- Naive string-matching procedure is like sliding a “template” containing the pattern over the text, noting for which shifts all of the characters on the template equal the corresponding characters in the text.
- The **for** loop of lines 3–5 considers each possible shift explicitly.
- The test in line 4 determines whether the current shift is valid; this test implicitly loops to check corresponding character positions until all positions match successfully or a mismatch is found.
- Line 5 prints out each valid shift s.

Naïve String-Matching Algorithm

Input: Text strings $T[1..n]$ and $P[1..m]$

Result: All valid shifts displayed

NAÏVE-STRING-MATCHER (T, P)

$n \leftarrow length[T]$

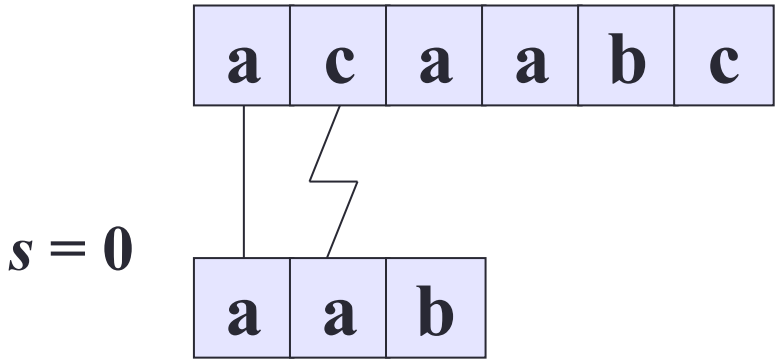
$m \leftarrow length[P]$

for $s \leftarrow 0$ **to** $n\text{-}m$

if $P[1..m] = T[(s+1)..(s+m)]$

print "pattern occurs with shift" s

Example: Naive String Matching Algorithm



$n \leftarrow length[T] = 6$

$m \leftarrow length[P] = 3$

for $s \leftarrow 0$ to n – m (6 - 3 = 3)

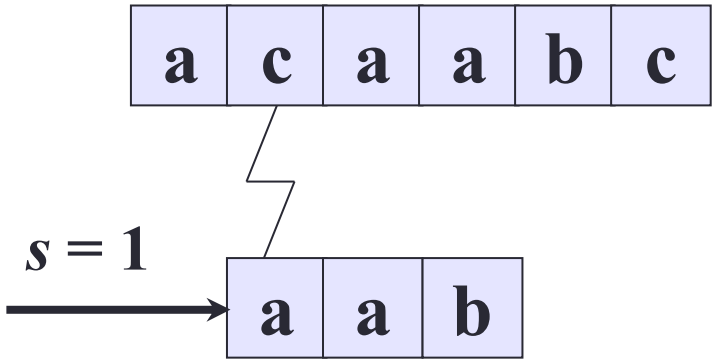
$P[1] = T[s + 1]$

$P[1] = T[1]$ (As a = a)

$P[2] = T[s + 2]$

But $P[2] \neq T[2]$ (As a ≠ c)

Example: Naive String Matching Algorithm

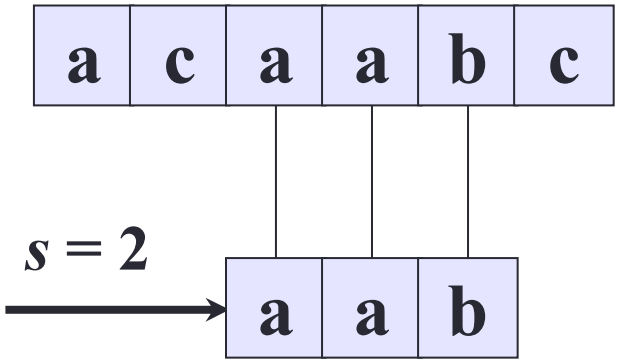


for $s \leftarrow 1$

$P[1] = T[s + 1]$

But $P[1]$? $T[2]$ (As a ? c)

Example: Naive String Matching Algorithm



for $s \leftarrow 2$

$P[1] = T[s + 1]$

$P[1] = T[3]$ (As a = a)

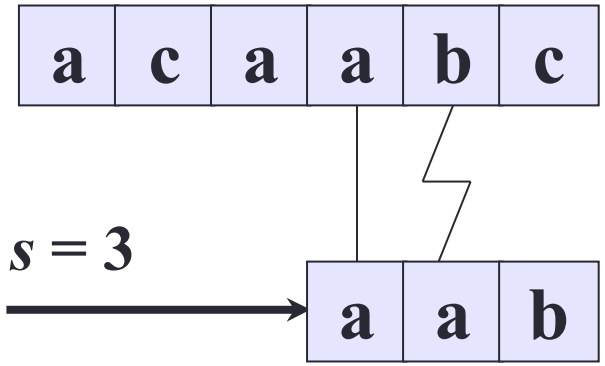
$P[2] = T[s + 2]$

$P[2] = T[4]$ (As a = a)

$P[3] = T[s + 3]$

$P[3] = T[5]$ (As b = b)

Example: Naive String Matching Algorithm



for $s \leftarrow 3$

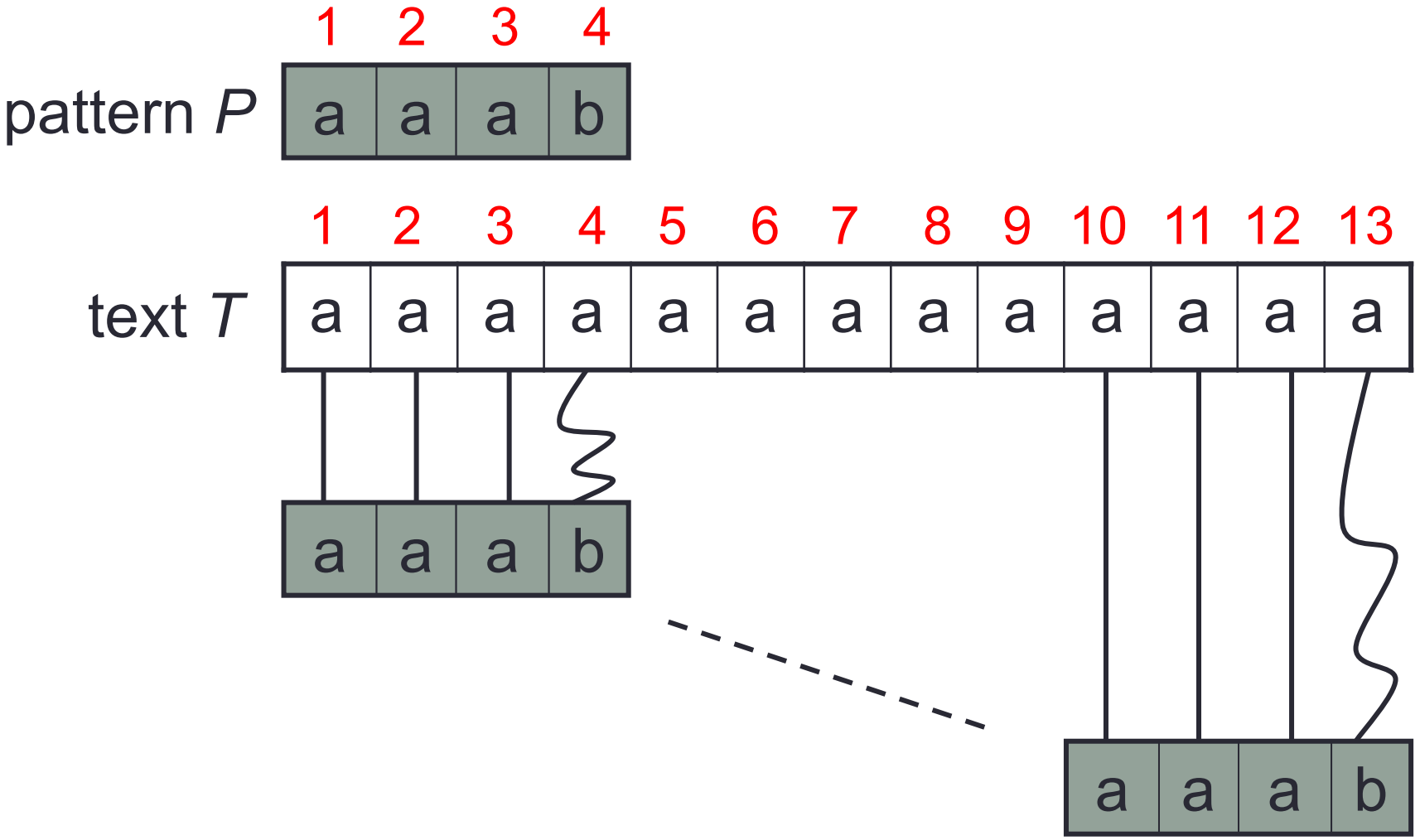
$P[1] = T[s + 1]$

$P[1] = T[4]$ (As a = a)

$P[2] = T[s + 2]$

But $P[2] \neq T[5]$ (As a ≠ b)

Analysis: Worst-case Example



Worst-case Analysis

- There are m comparisons for each shift in the worst case
- There are $n-m+1$ shifts
- So, the worst-case running time is $\Theta((n-m+1)m)$
  - In the example on previous slide, we have (13-4+1)4 comparisons in total
- Naïve method is inefficient because information from a shift is not used again

The Rabin-Karp Algorithm

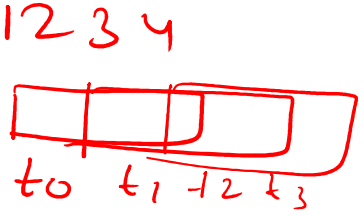
Notes compiled by Radhika Chapaneri

- Given a text $T[1 .. n]$ of length n, a pattern $P[1 .. m]$ of length $m \leq n$, both as arrays.
- Assume that elements of P and T are characters drawn from a finite set of alphabets Σ.
- Where $\Sigma = \{0, 1, 2, \ldots, 9\}$, so that each character is a decimal digit.
- Now our objective is "finding all valid shifts with which a given pattern P occurs in a text T".

Notations: The Rabin-Karp Algorithm

Let us suppose that

- p denotes decimal value of given a pattern $P[1 .. m]$
- t_s = decimal value of length-m substring $T[s + 1 .. s + m]$, of given text $T[1 .. n]$, for $s = 0, 1, ..., n - m$.



- It is very obvious that, $t_s = p$ if and only if $T[s + 1 .. s + m] = P[1 .. m]$;
  thus, s is a valid shift if and only if $t_s = p$.

- Now the question is how to compute p and t_s efficiently
- Answer is Horner's rule

Horner's Rule

Notes compiled by Radhika Chapaneri

Example: Horner's rule

$[3, 4, 5] = 5 + 10(4 + 10(3)) = 5 + 10(4 + 30) = 5\ 340 = 345$

$p = P[3] + 10\ (P[3 - 1] + 10(P[1]))$.

Formula

- We can compute p in time $\Theta(m)$ using this rule as

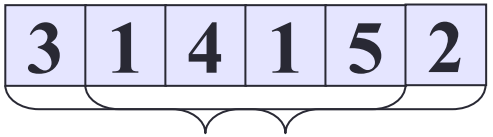
$p = P[m] + 10\ (P[m\text{-}1] + 10(P[m\text{-}2] + \ldots + 10(P[2] + 10P[1])\ ))$

- Similarly t_0 can be computed from $T\,[1\ ..\ m]$ in time $\Theta(m)$.

- To compute $t_1, t_2, \ldots, t_{n-m}$ in time $\Theta(n - m)$, it suffices to observe that t_{s+1} can be computed from t_s in constant time.

Computing t_{s+1} from t_s in constant time

- Text = [3, 1, 4, 1, 5, 2]; t_0 = 31415
- m = 5; Shift = 0



- Old higher-order digit = 3
- New low-order digit = 2

- $t_1 = 10.(31415 - 10^{4}.T(1)) + T(5+1)$
  $= 10.(31415 - 10^{4}.3) + 2$
  $= 10(1415) + 2 = 14152$

- $t_{s+1} = 10(t_s - T[s+1]\,10^{m-1}) + T[s+m+1])$
- $t_1 = 10(t_0 - T[1]\,10^{4}) + T[0+5+1])$

Procedure: Computing t_{s+1} from t_s

1. Subtract T[s + 1]10^{m-1} from t_s, removes high-order digit
2. Multiply result by 10, shifts the number left one position
3. Add $T[s + m + 1]$, it brings appropriate low-order digit.

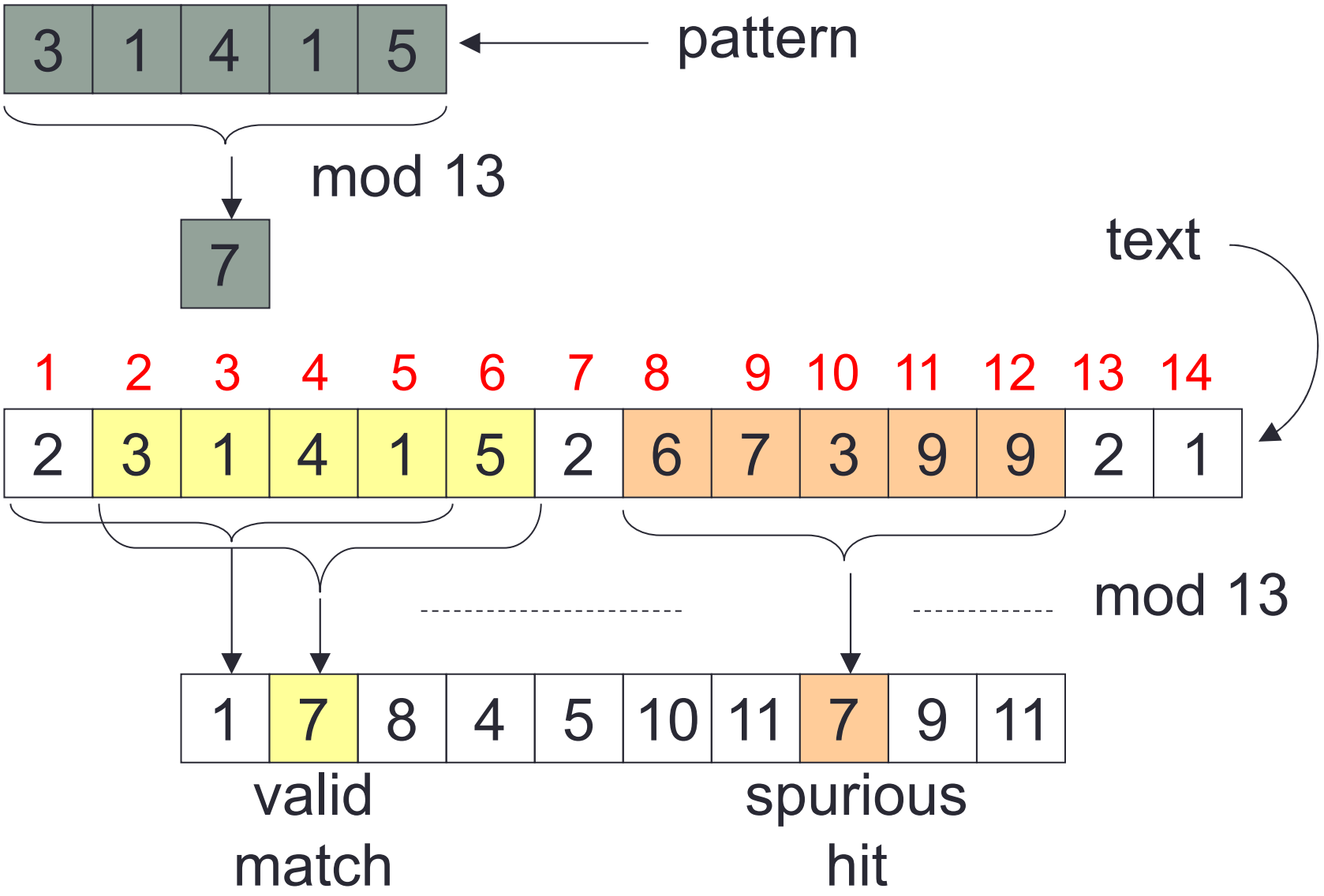
$$t_{s+1} = (10(t_s - T[s + 1]\ 10^{m-1}) + T[s + m + 1])$$

Another issue and its treatment

- The only difficulty with the above procedure is that p and t_s may be too large to work with conveniently.

- Fortunately, there is a simple cure for this problem, compute p and the t_s modulo a suitable modulus q.

Example


3 1 4 1 5
pattern
mod 13
7
text
1 2 3 4 5 6 7 8 9 10 11 12 13 14
2 3 1 4 1 5 2 6 7 3 9 9 2 1
mod 13
1 7 8 4 5 10 11 7 9 11
valid match
spurious hit

Problem of Spurious Hits

- $t_s \equiv p \pmod{q}$ does not imply that $t_s = p$
  - Modular equivalence does not necessarily mean that two integers are equal
- A case in which $t_s \equiv p \pmod{q}$ when $t_s \neq p$ is called a *spurious hit*
- On the other hand, if two integers are not modular equivalent, then they cannot be equal

Sequence of Steps Designing Algorithm

1. Compute the lengths of pattern P and text T
2. Compute p and t_s under modulo q using Horner's Rule
3. For any shift s for which $t_s \equiv p \pmod{q}$, must be tested further to see if s is really valid shift or a **spurious hit**.
4. This testing can be done by checking the condition: $P[1 .. m] = T[s + 1 .. s + m]$. If these strings are equal s is a valid shift otherwise spurious hit.
5. If for shift s, $t_s \equiv p \pmod{q}$ is false, compute t_{s+1} and replace it with t_s and repeat the step 3.

Note

- As $t_s \equiv p \pmod{q}$ does not imply that $t_s = p$, hence text comparison is required to find valid shift

The Rabin-Karp Algorithm

Generalization

- Given a text $T[1 .. n]$ of length n, a pattern $P[1 .. m]$ of length $m \leq n$, both as arrays.
- Assume that elements of P and T are characters drawn from a finite set of alphabets $\Sigma = \{0, 1, 2, \ldots, d\text{-}1\}$.
- Now our objective is "finding all valid shifts with which a given pattern P occurs in a text T".

Note

- $t_{s+1} = (d(t_s - T[s + 1]h) + T[s + m + 1]) \bmod q$

  where $h = d^{m-1} \pmod q$ is the value of the digit "1" in the high-order position of an m-digit text window.

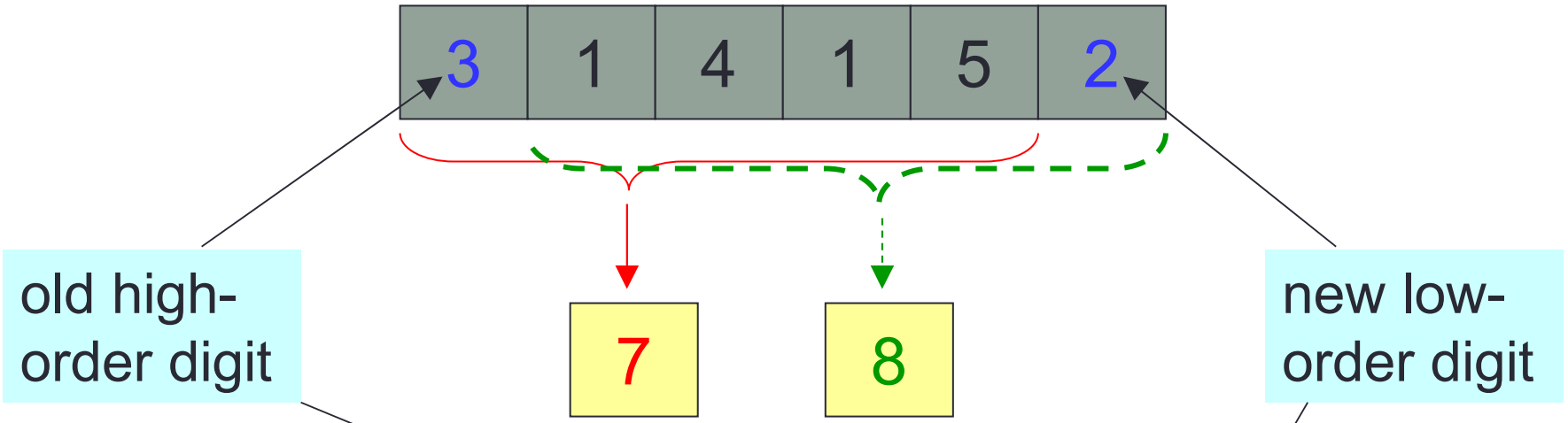
The Rabin-Karp Algorithm

Notes compiled by Radhika Chapaneri

RABIN-KARP-MATCHER(T, P, d, q)

```
1   n ← length[T]
2   m ← length[P]
3   h ← d^(m-1) mod q
4   p ← 0
5   t_0 ← 0
6   for i ← 1 to m                          ▯ Preprocessing.
7       do p ← (dp + P[i]) mod q
8          t_0 ← (dt_0 + T[i]) mod q
9   for s ← 0 to n - m                      ▯ Matching.
10      do if p = t_s
11            then if P[1 .. m] = T[s + 1 .. s + m]
12                    then print "Pattern occurs with shift" s
13         if s < n - m
14            then t_(s+1) ← (d(t_s - T[s + 1]h) + T[s + m + 1]) mod q
```

How values modulo 13 are computed



$$14152 \equiv (31415 - 3 \cdot 10000) \cdot 10 + 2 \pmod{13}$$

$$\equiv (7 - 3 \cdot 3) \cdot 10 + 2 \pmod{13}$$

$$\equiv 8 \pmod{13}$$

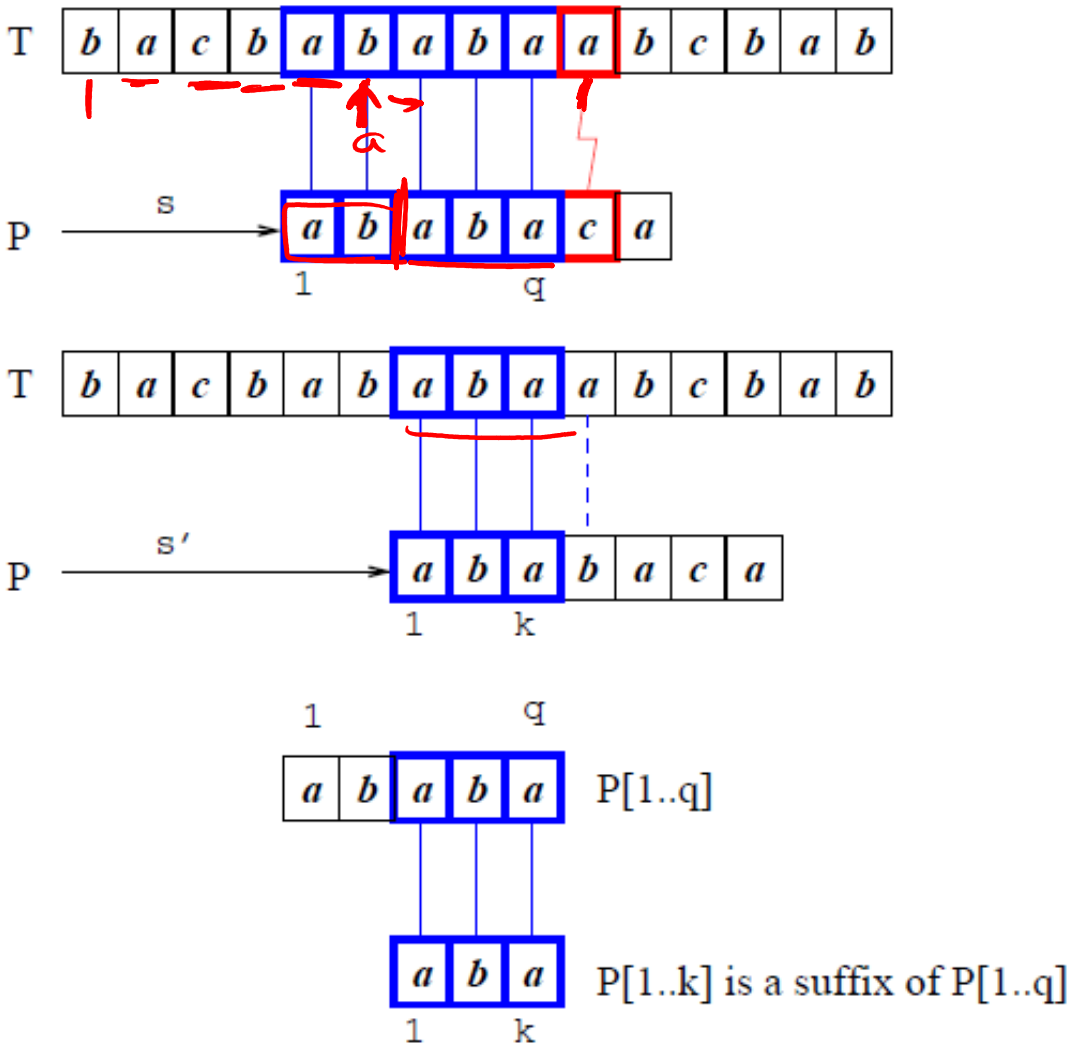
Analysis: The Rabin-Karp Algorithm

- Worst case Running Time
  - Preprocessing time: $\Theta(m)$
  - Matching time is $\Theta((n - m + 1)m)$

Knuth-Morris-Pratt String Matching

- Knuth, Morris and Pratt discovered first linear time string-matching algorithm by analysis of the naïve algorithm.
- It keeps the information that naive approach wasted gathered during the scan of the text. By avoiding this waste of information, it achieves a running time of O(m + n).
- The implementation of Knuth-Morris-Pratt algorithm is efficient because it minimizes the total number of comparisons of the pattern against the input string.

Knuth-Morris-Pratt String Matching

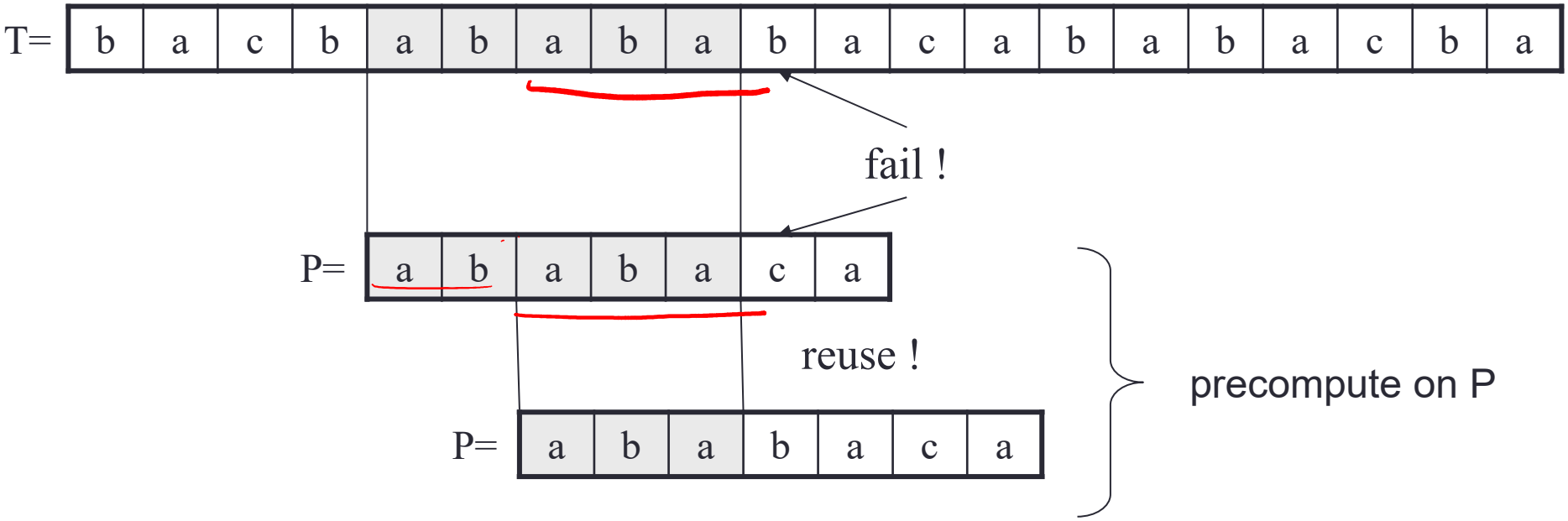


Knuth-Morris-Pratt String Matching

- The prefix-function ∏:
- It preprocesses the pattern to find matches of prefixes of the pattern with the pattern itself.
- It is defined as the size of the largest prefix of P[1..j − 1] that is also a suffix of P[2..j].
- It also indicates how much of the last comparison can be reused if it fails.

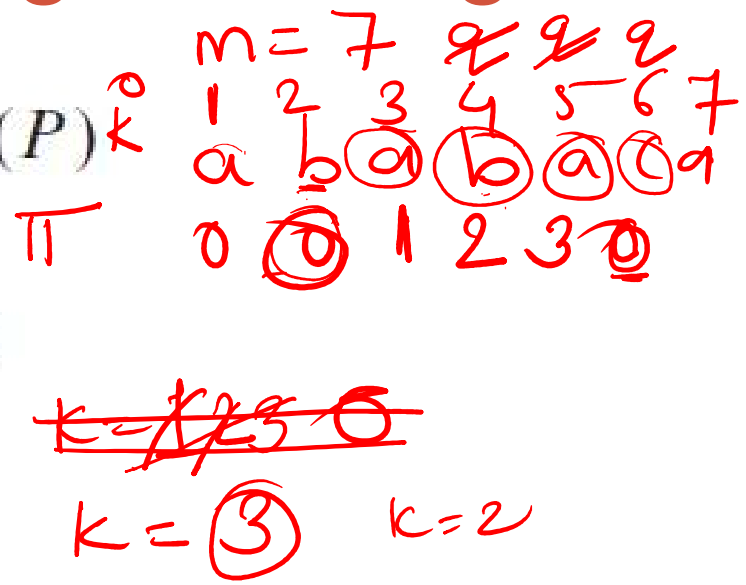
Knuth-Morris-Pratt String Matching

- main idea: reuse the work, after a failure



Knuth-Morris-Pratt String Matching

```
COMPUTE-PREFIX-FUNCTION(P)
 1  m = P.length
 2  let π[1..m] be a new array
 3  π[1] = 0
 4  k = 0
 5  for q = 2 to m
 6      while k > 0 and P[k + 1] ≠ P[q]
 7          k = π[k − 1]
 8      if P[k + 1] == P[q]
 9          k = k + 1
10      π[q] = k
11  return π
```



Knuth-Morris-Pratt String Matching

Now let us consider an example so that the algorithm can be clearly understood.

Text	b a c b a b a b a b a c a a b
Pattern	a b a b a c a

Let us execute the KMP algorithm to find whether ‘p’ occurs in ‘T’.

Knuth-Morris-Pratt String Matching

- Compute ∏ function on Pattern a b a b a c a

Pattern	a	b	a	b	a	c	a
∏	0	0	1	2	3	0	1

Knuth-Morris-Pratt String Matching

- Compute ∏ function on Pattern
- a b a b b a b b a b b a b a b b a b b
- 0 0 1 2 0 1 2 0 1 2 0 1 2 3 4 5 6 7 8

KMP Algorithm

Step 1: Initialize the input variables :
n = Length of the Text .
m = Length of the Pat tern .
∏ = Pref i x −function of pattern (p) .
q = Number of characters matched .

Step 2: Define the variable :
q=0 , the beginning of the match .

KMP Algorithm

Step 3: Compare the first character of the pattern with first character of text .
If match is not found , substitute the value of ⌈⌉[q] to q .
If match is found , then increment the value of q by 1.

Step 4: Check whether all the pattern elements are matched with the text elements .
If not , repeat the search process .
If yes , print the number of shifts taken by the pattern .

Step 5: look for the next match .

KMP Algorithm

```
KMP-MATCHER(T, P)
 1  n = T.length
 2  m = P.length
 3  π = COMPUTE-PREFIX-FUNCTION(P)
 4  q = 0                                  // number of characters matched
 5  for i = 1 to n                         // scan the text from left to right
 6      while q > 0 and P[q + 1] ≠ T[i]
 7          q = π[q]                       // next character does not match
 8      if P[q + 1] == T[i]
 9          q = q + 1                      // next character matches
10      if q == m                          // is all of P matched?
11          print "Pattern occurs with shift" i − m
12          q = π[q]                       // look for the next match
```

KMP Algorithm

- Text b a c b a b a b a b a c a a b
- Pattern a b a b a c a

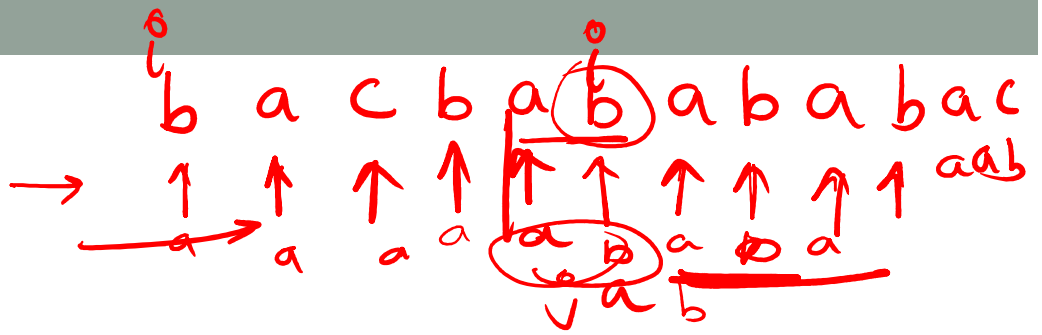
KMP Algorithm

- O(m) - It is to compute the prefix function values.
- O(n) - It is to compare the pattern to the text.
- Total of O(n + m) run time.

KMP Algorithm

- Advantages:
- The running time of the KMP algorithm is optimal (O(m + n)), which is very fast.
- The algorithm never needs to move backwards in the input text T. It makes the algorithm good for processing very large files.

Initially: n = size of S = 15;
m = size of p = 7



Step 1: i = 1, q = 0
comparing p[1] with S[1]

S	b	a	c	b	a	b	a	b	a	b	a	c	a	a	b

p	a	b	a	b	a	c	a

P[1] does not match with S[1]. 'p' will be shifted one position to the right.

Step 2: i = 2, q = 0
comparing p[1] with S[2]

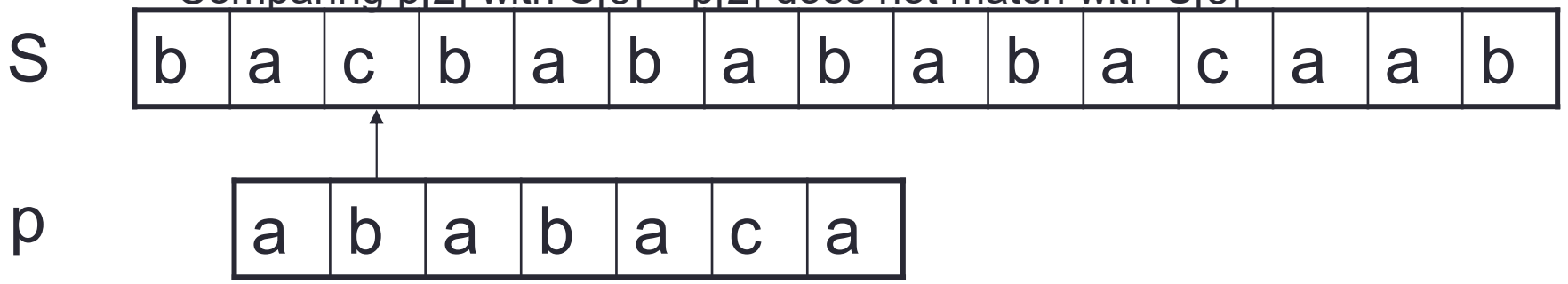
S	b	a	c	b	a	b	a	b	a	b	a	c	a	a	b

p	a	b	a	b	a	c	a

P[1] matches S[2]. Since there is a match, p is not shifted.

Step 3: i = 3, q = 1

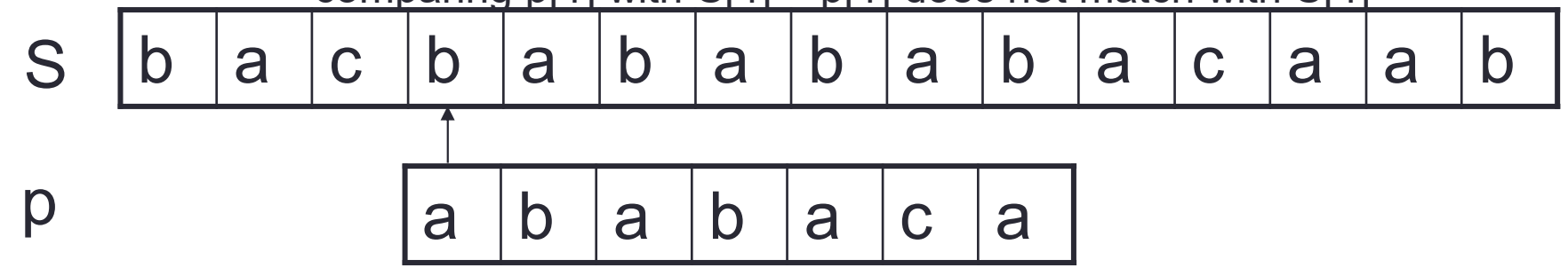
Comparing p[2] with S[3] p[2] does not match with S[3]



Backtracking on p, comparing p[1] and S[3]

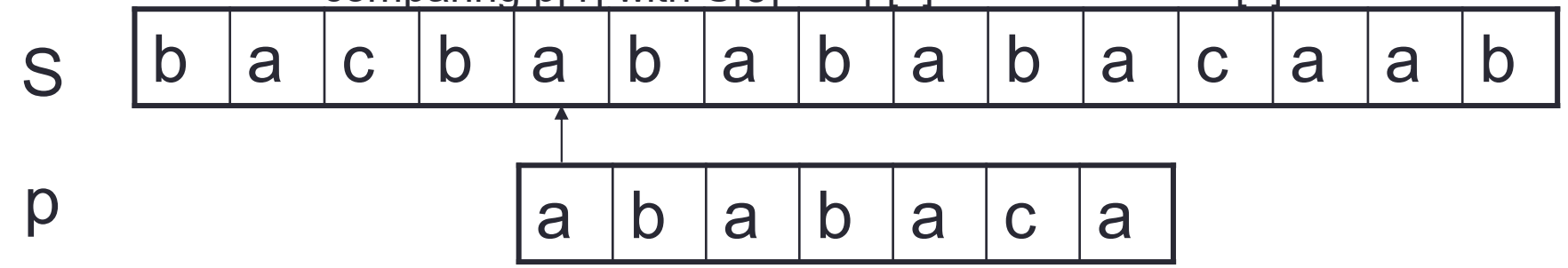
Step 4: i = 4, q = 0

comparing p[1] with S[4] p[1] does not match with S[4]



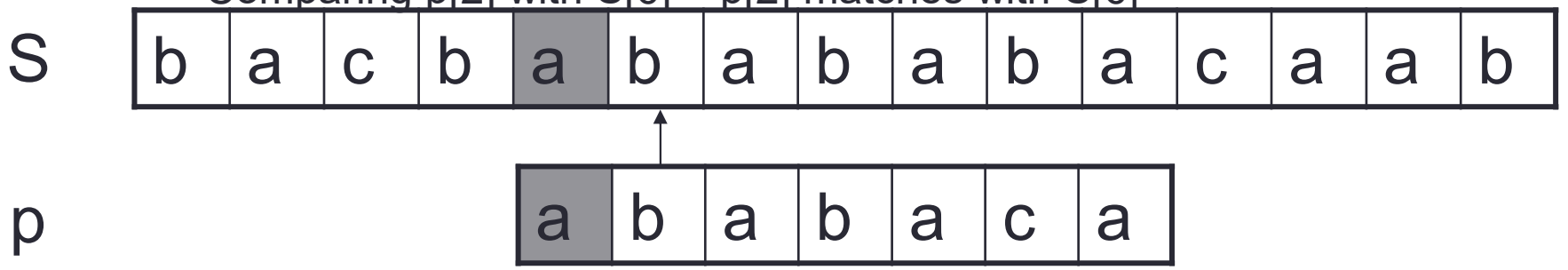
Step 5: i = 5, q = 0

comparing p[1] with S[5] p[1] matches with S[5]



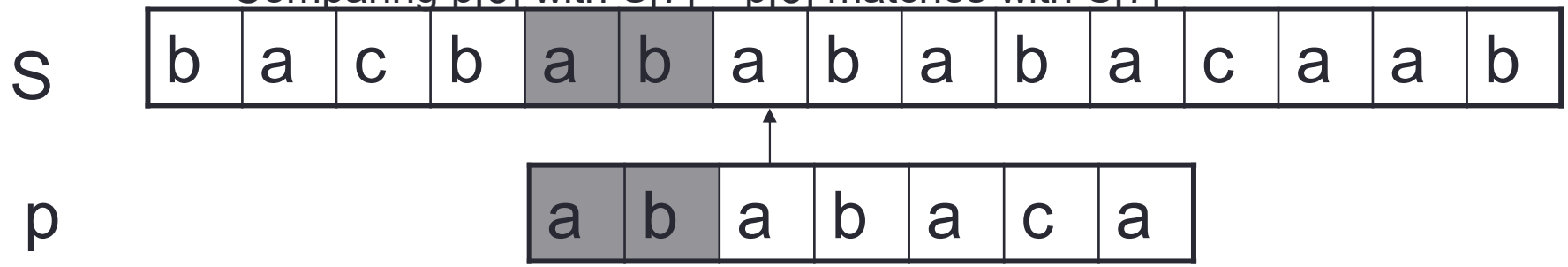
Step 6: i = 6, q = 1

Comparing p[2] with S[6] p[2] matches with S[6]



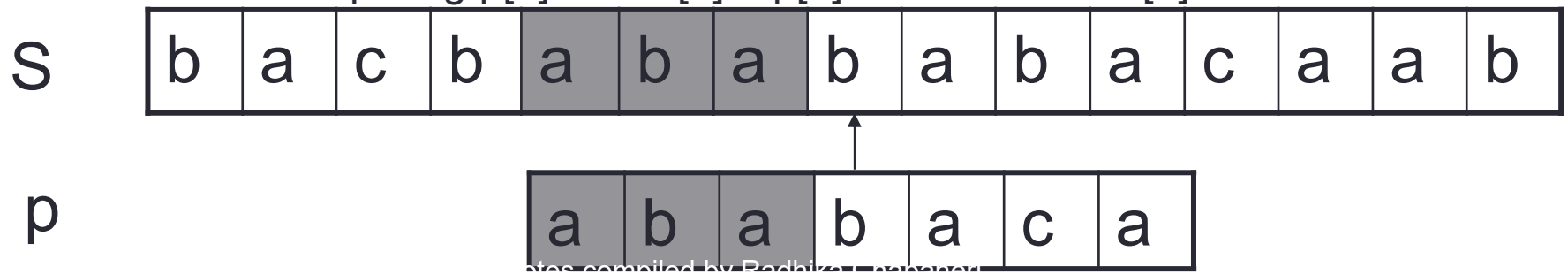
Step 7: i = 7, q = 2

Comparing p[3] with S[7] p[3] matches with S[7]



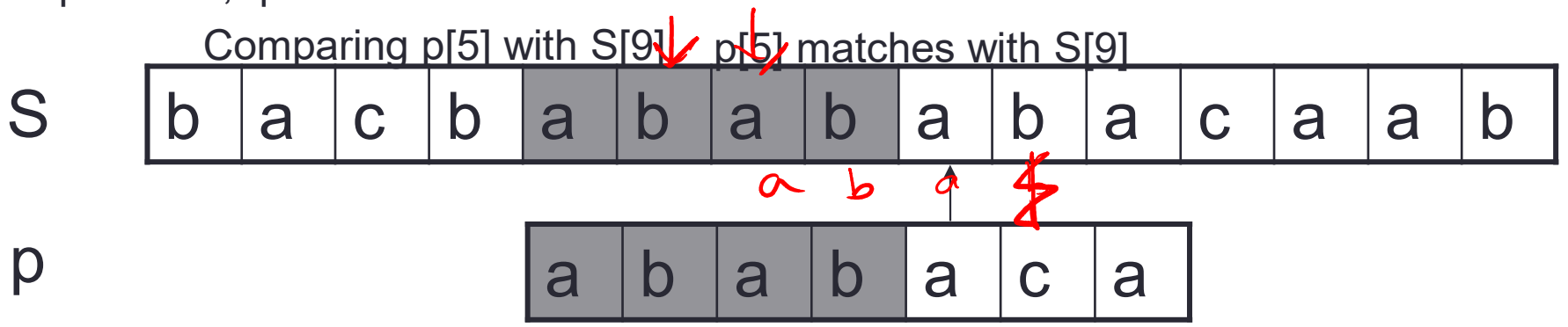
Step 8: i = 8, q = 3

Comparing p[4] with S[8] p[4] matches with S[8]



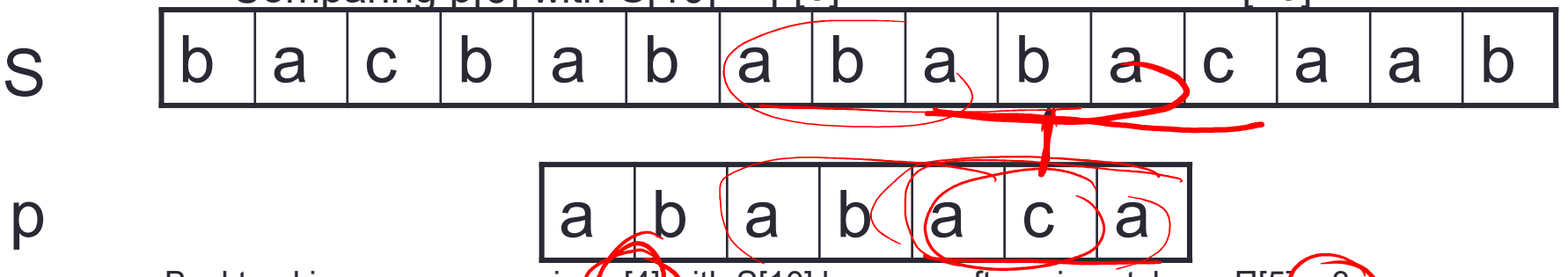
Step 9: i = 9, q = 4

Comparing p[5] with S[9] p[5] matches with S[9]



Step 10: i = 10, q = 5

Comparing p[6] with S[10] p[6] doesn't match with S[10]

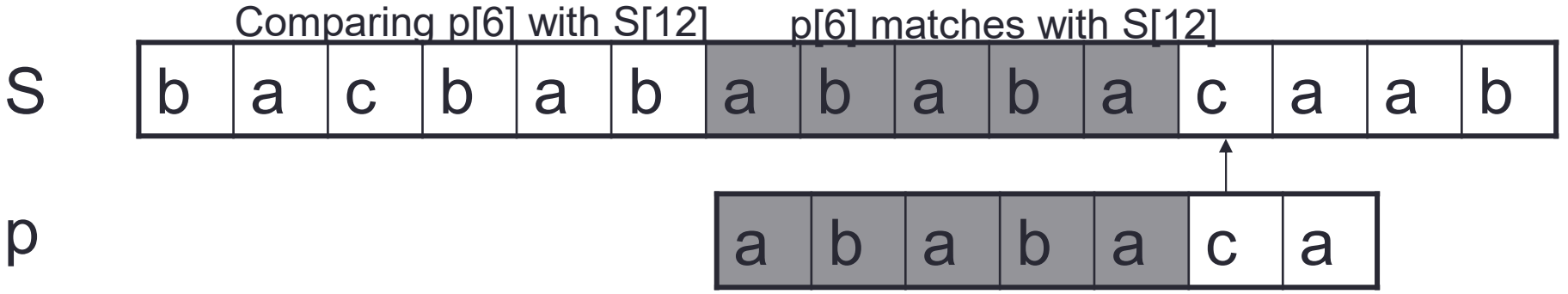


Backtracking on p, comparing p[4] with S[10] because after mismatch q = Π[5] = 3

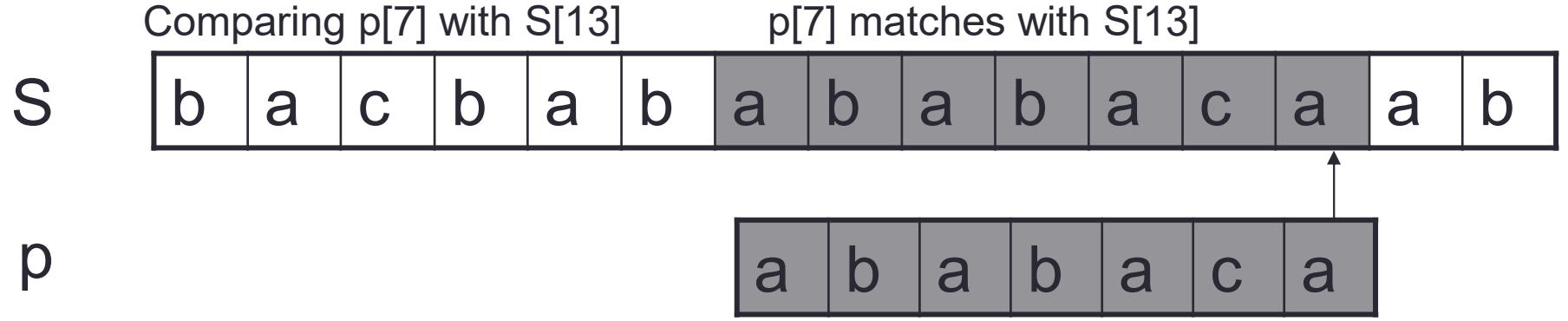
Step 11: i = 11, q = 4

Comparing p[5] with S[11] p[5] matches with S[11]

Step 12: i = 12, q = 5



Step 13: i = 13, q = 6



Pattern ‘p’ has been found to completely occur in string ‘S’. The total number of shifts that took place for the match to be found are: i – m = 13 – 7 = 6 shifts.

String Matching with Finite Automata

- A **finite automaton** M is a 5-tuple $(Q, q_0, A, \Sigma, \delta)$, where
  - Q is a finite set of **states**,
  - $q_0 \in Q$ is the **start state**,
  - A ⍰ Q is a distinguished set of **accepting states**,
  - Σ is a finite **input alphabet**,
  - δ is a function from $Q \times \Sigma$ into Q, called the **transition function** of M.

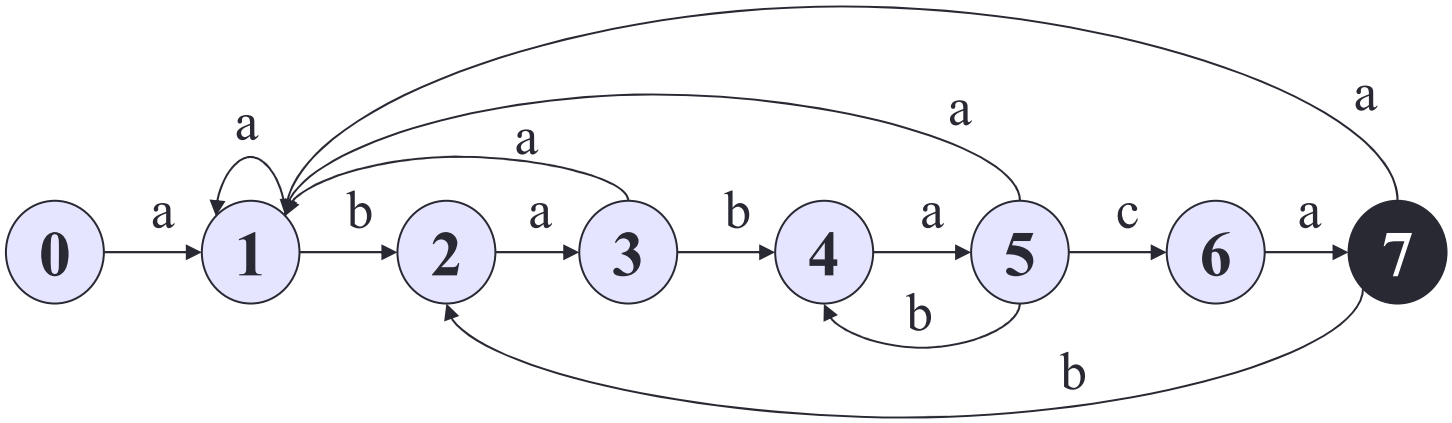
String Matching with Finite Automata

Notes compiled by Radhika Chapaneri

- The finite automation begins in state q and reads the characters of its input string one at a time.
- If the automation is in state q and reads input chrachetrer a it moves (“makes a transition”) from state q to state (q,a).
- Whenver its current state q is a member of A, the machine M has accepted the string so far.
- An input that is not accepted is rejected.
- String-matching automata are very efficient because it examines each character *exactly once*, taking constant time.
- The matching time used-after preprocessing the pattern to build the automaton-is therefore $\Theta(n)$.

String Matching Automata for given Pattern

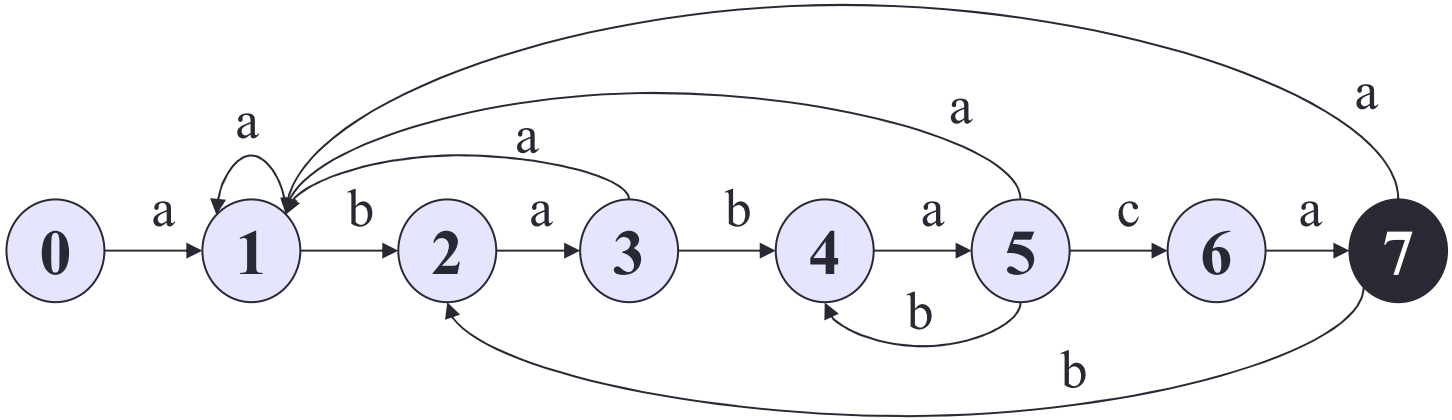
- construct finite automata for the given pattern



String Matching using Finite Automata

Finite Automata for Pattern P = ababaca

Text T = abababacaba.



i	--	1	2	3	4	5	6	7	8	9	10	11
$T[i]$	--	a	b	a	b	a	b	a	c	a	b	a
state $\varphi(T_i)$	0	1	2	3	4	5	4	5	6	7	2	3

String Matching with finite Automata


```
FINITE-AUTOMATON-MATCHER(T, δ, m)
1   n ← length[T]
2   q ← 0
3   for i ← 1 to n
4       do q ← δ(q, T[i])
5          if q = m
6   then print "Pattern occurs with shift" i - m
```

- Matching time on a text string of length n is $\Theta(n)$.
- Memory Usage: $O(m|\Sigma|)$,
- Preprocessing Time: Best case: $O(m|\Sigma|)$.

3. String Matching with Finite Automata


```
COMPUTE-TRANSITION-FUNCTION(P, Σ)
1   m ← length[P]
2   for q ← 0 to m
3       do for each character a ∈ Σ
4           do k ← min(m + 1, q + 2)
5              repeat k ← k - 1
6                 until P_k ⊐ P_q a
7              δ(q, a) ← k
8   return δ
```

Running Time = $O(m^3 |\Sigma|)$

Summary

Algorithm	Preprocessing Time	Matching Time
Naive	0	$O((n-m+1)m)$
Rabin-Karp	$\Theta(m)$	$O((n-m+1)m)$
Finite Automaton	$O(m\|\Sigma\|)$	$\Theta(n)$