Algorithms

String Processing

Ming-Hsien Tsai

Academia Sinica
National Taiwan University

Based on: U. Manber. Introduction to Algorithms - A Creative Approach. Addison-Wesley, 1989.
Also based on slides made by Professor Yih-Kuen Tsay.

Data Compression

- Assume a sequence of English letters
- Each of the 26 characters is represented by a unique string of bits, called the ***encoding*** of the character
- For example, in ASCII encoding
  - A is represented by the bit string 1000001
  - B is represented by the bit string 1000010
  - The word “AND” requires 21 bits

Data Compression (cont'd)

- If A appears very often, can we make the bit representation of A shorter?
- For example, the encoding of A is changed to 1001
- Observe that the encoding of M is 1001101
- What is the problem?

Data Compression (cont'd)

Prefix constraint: the prefixes of an encoding of one character must not be equal to a complete encoding of another character

Problem

Given a text (a sequence of characters), find an encoding for the characters that satisfies the prefix constraint and that minimizes the total number of bits needed to encode the text

Data Compression (cont'd)

- Denote the text by F
- Denote the characters by C_1, C_2, …, C_n and denote their ***frequencies*** in the text by f_1, f_2, …, f_n
- Given an encoding E in which a bit string S_i of length s_i represents C_i, the length of the text encoded by using E is:

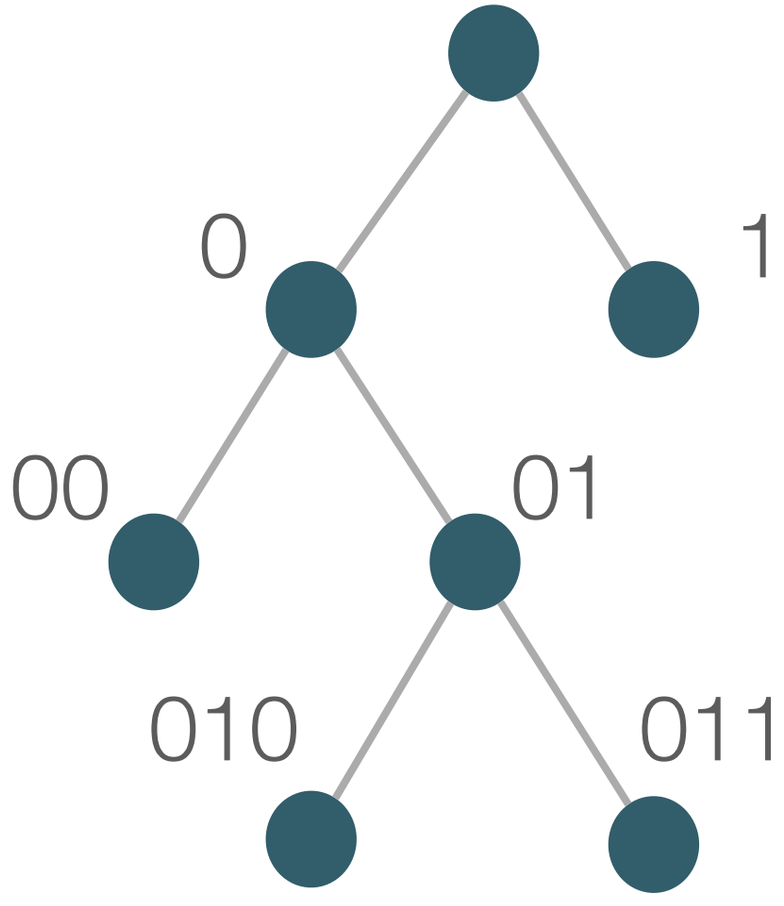
$$L(E, F) = \sum_{i=1}^{n} s_i \cdot f_i$$

- The goal is to find an encoding E that satisfies the prefix constraint and minimizes L(E, F)

Data Compression (cont'd)

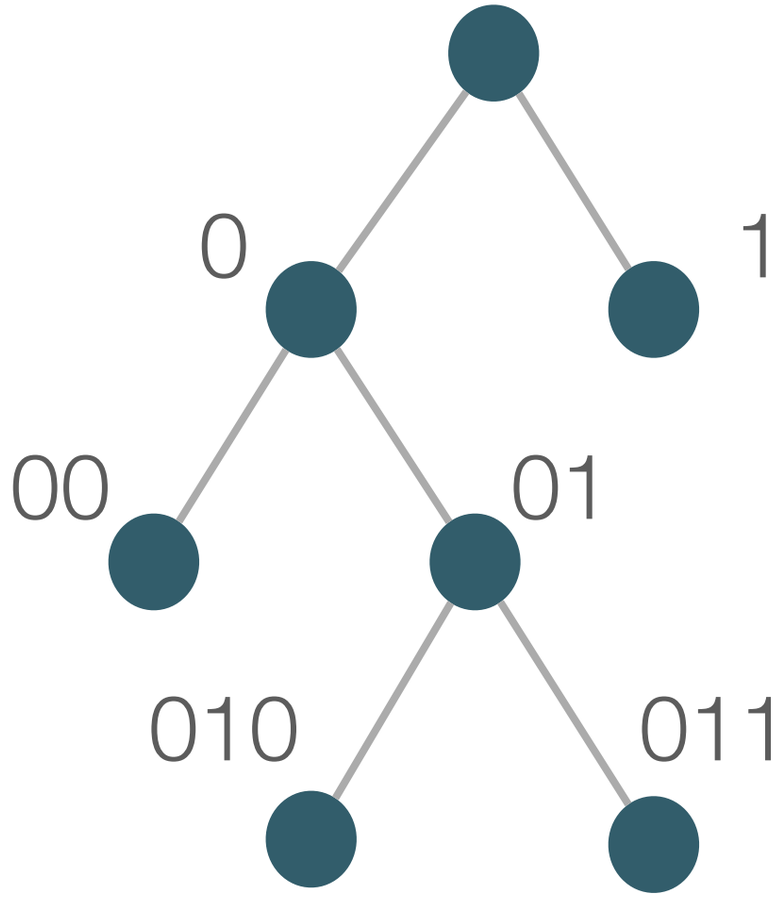
We can use the tree representation of encoding

- A character corresponds to a node
- A node corresponds to at most a character



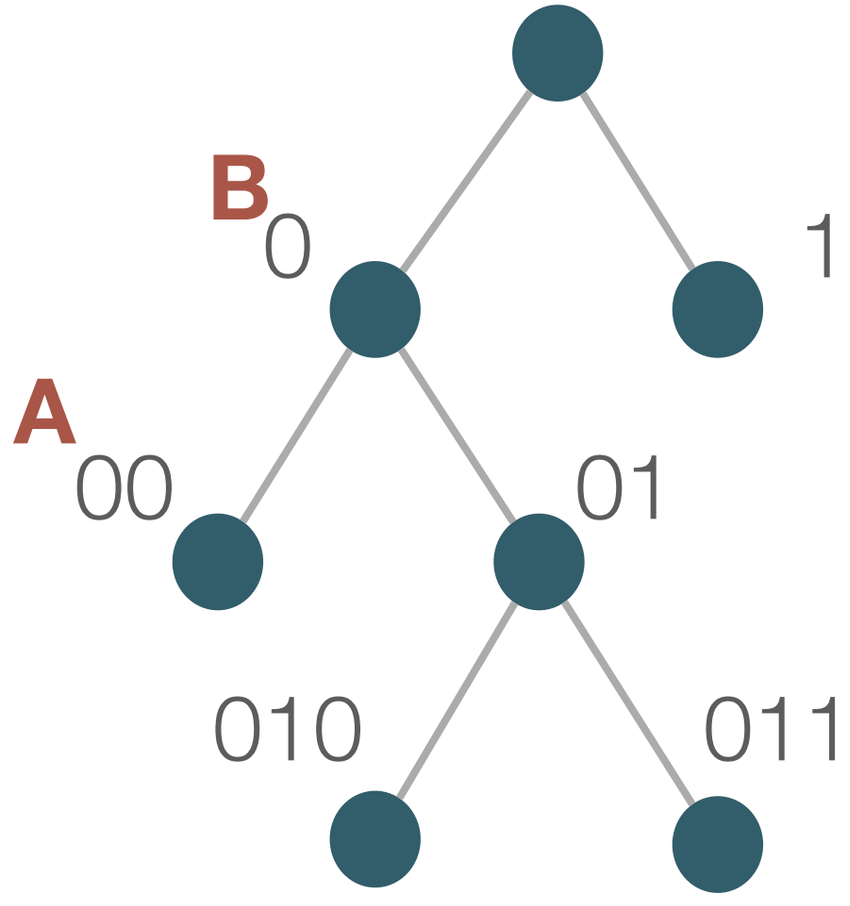
Data Compression (cont'd)

What kind of correspondence between characters and nodes will satisfy the prefix constraint?



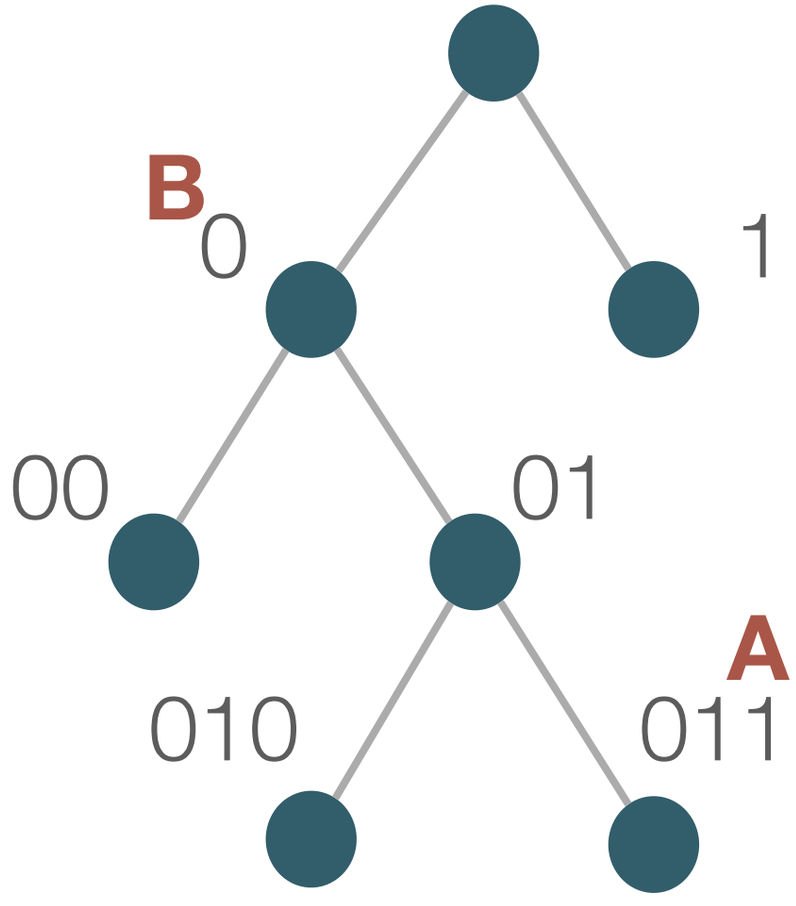
Data Compression (cont'd)

What kind of correspondence between characters and nodes will satisfy the prefix constraint?



Data Compression (cont'd)

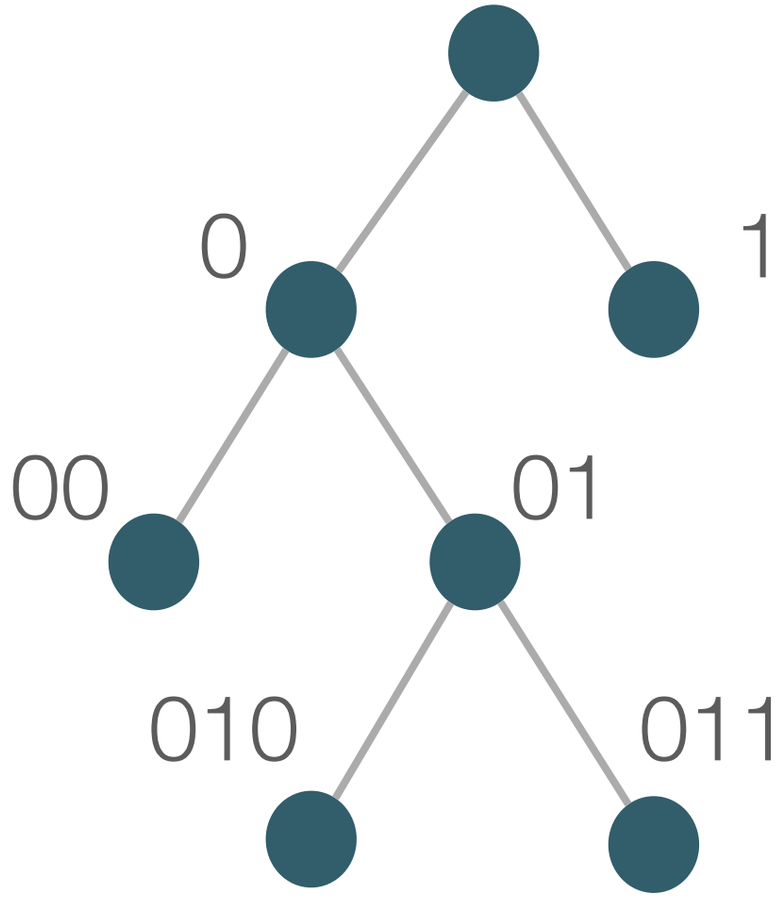
What kind of correspondence between characters and nodes will satisfy the prefix constraint?



Data Compression (cont'd)

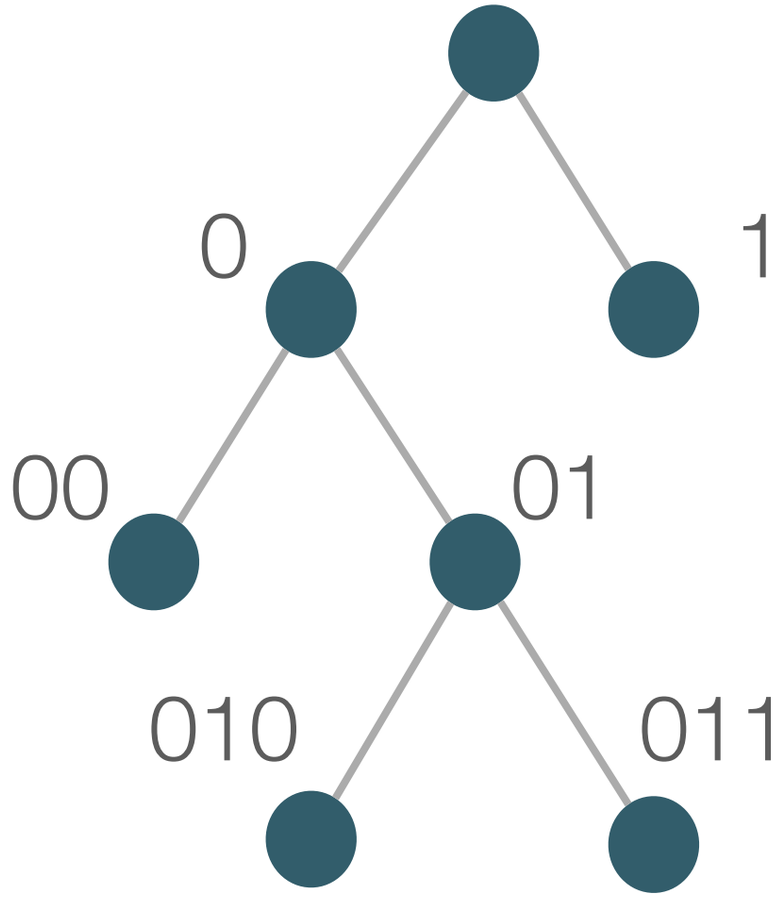
What kind of correspondence between characters and nodes will satisfy the prefix constraint?

All characters correspond to leaves



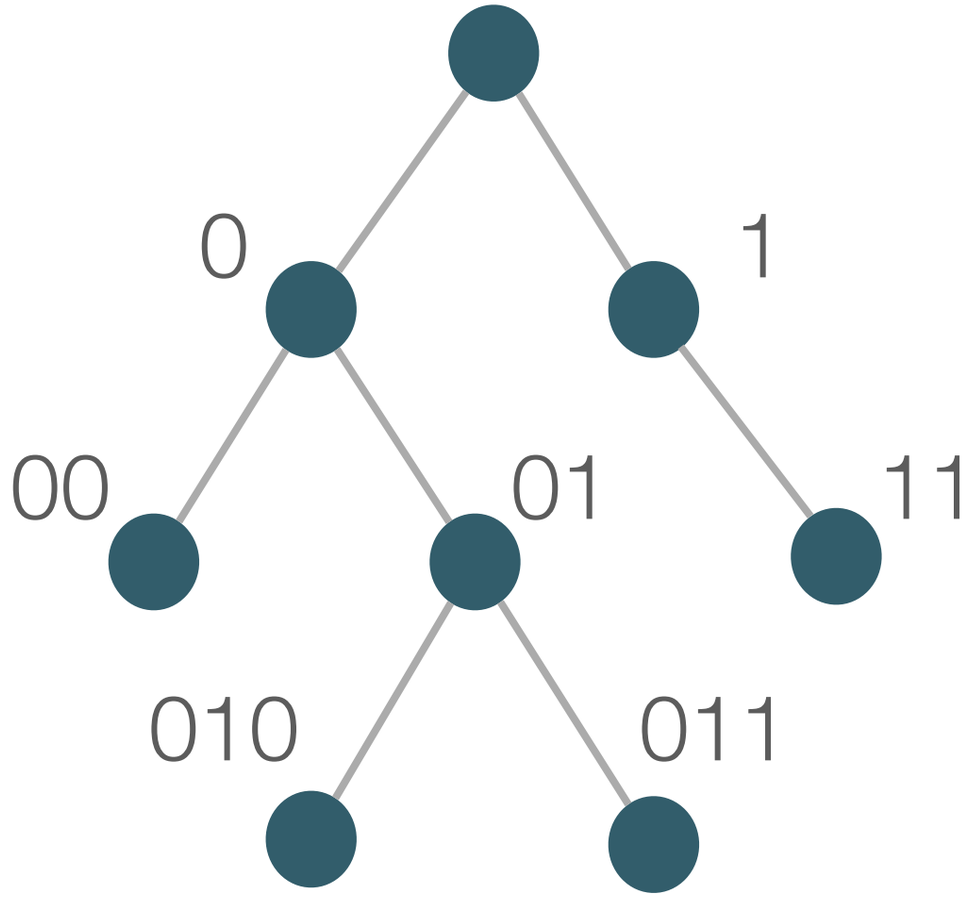
Data Compression (cont'd)

What other properties does the tree have if it minimizes L(E, F)?



Data Compression (cont'd)

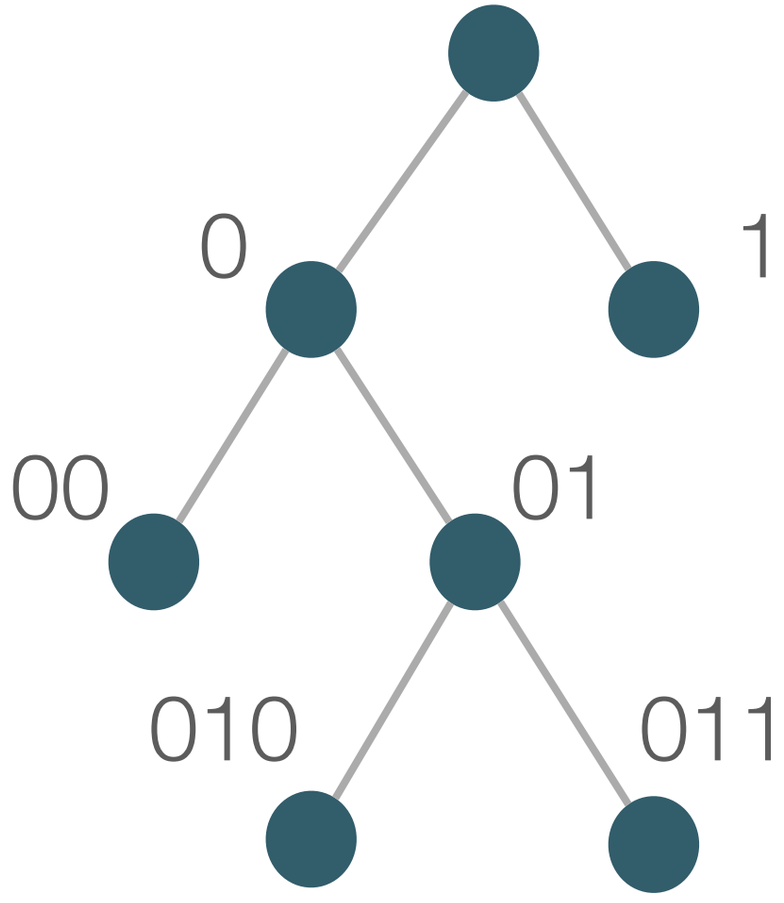
What other properties does the tree have if it minimizes L(E, F)?



Data Compression (cont'd)

What other properties does the tree have if it minimizes L(E, F)?

Each node has either two children or no children



Data Compression (cont'd)

We want to construct a tree that encodes n characters and minimizes L(E, F)

Induction Hypothesis

We know how to construct a tree that encodes n - 1 characters such that the tree minimizes L(E, F)

How to apply the induction hypothesis?

Option 1: Construct a tree for C_1, C_2, …, C_{n-1}, and then expand the tree for C_n

Option 2: Construct a tree for C_1, C_2, …, C_{i-1}, C_{i+1}, …, C_{j-1}, C_{j+1}, …, C_{n-2}, D_{ij} and then expand the node corresponding to D_{ij}

Data Compression (cont'd)

We want to construct a tree that encodes n characters and minimizes L(E, F)

Induction Hypothesis

We know how to construct a tree that encodes n - 1 characters such that the tree minimizes L(E, F)

How to apply the induction hypothesis?

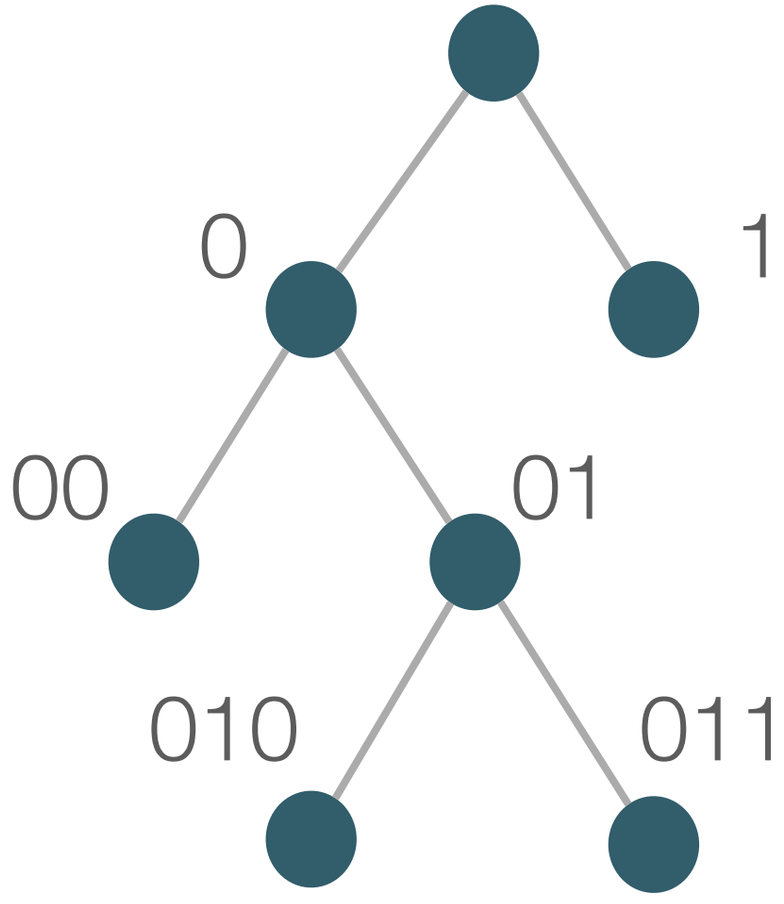
Option 1: Construct a tree for C_1, C_2, …, C_{n-1}, and then expand the tree for C_n ❌

Option 2: Construct a tree for C_1, C_2, …, C_{i-1}, C_{i+1}, …, C_{j-1}, C_{j+1}, …, C_{n-2}, D_{ij} and then expand the node corresponding to D_{ij} ⭕

Data Compression (cont'd)

What is the best choice of C_i and C_j to be merged as D_{ij}?

Note that after the expansion of the node corresponding to D_{ij}, the bit length of C_i and C_j will increase by 1



Data Compression (cont'd)

- The construction algorithm:
  - Base case (n = 1): trivial
  - Inductive step (n > 1):
    - Find two characters C_i and C_j with minimal frequencies f_i and f_j
    - Define a new character D_{ij} with frequency $(f_i + f_j)$
    - Construct a tree for D_{ij} and all the characters C_k where $1 \leq k \leq n$, $k \neq i$, and $k \neq j$ (by induction hypothesis)
    - Expand the node corresponding to D_{ij} in the tree such that one child corresponds to C_i and the other child corresponds to C_j

Data Compression (cont'd)

Huffman's encoding is one implementation

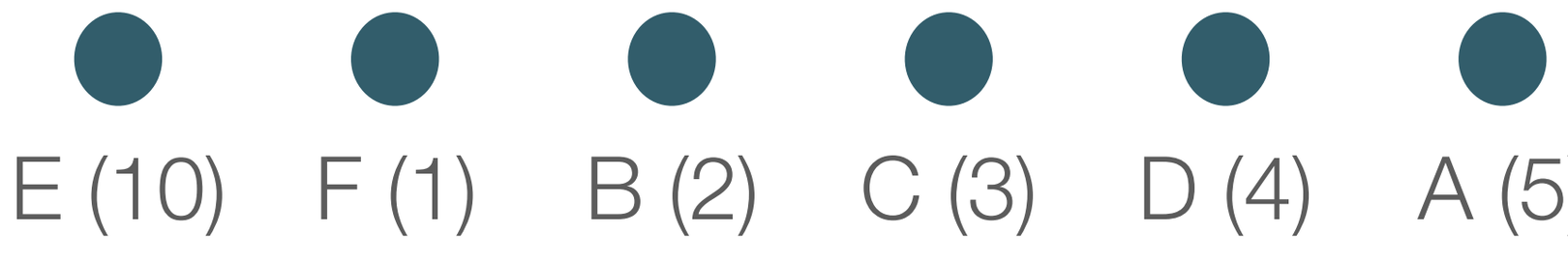
```
Algorithm Huffman_Encoding (S, f);
begin
    insert all characters appearing in S into a heap H according to
        their frequencies defined by f;
    while H is not empty do
        if H contains only one character X then
            make X the root of T
        else
            pick two characters X and Y with lowest frequencies and
                delete them from H;
            replace X and Y with a new character Z whose frequency is
                the sum of the frequencies of X and Y;
            insert Z to H;
            make X and Y children of Z in T {Z has no parent yet}
end
```

Complexity: O(n log n)

Data Compression (cont'd)

The tree built by Huffman's encoding is called a Huffman tree

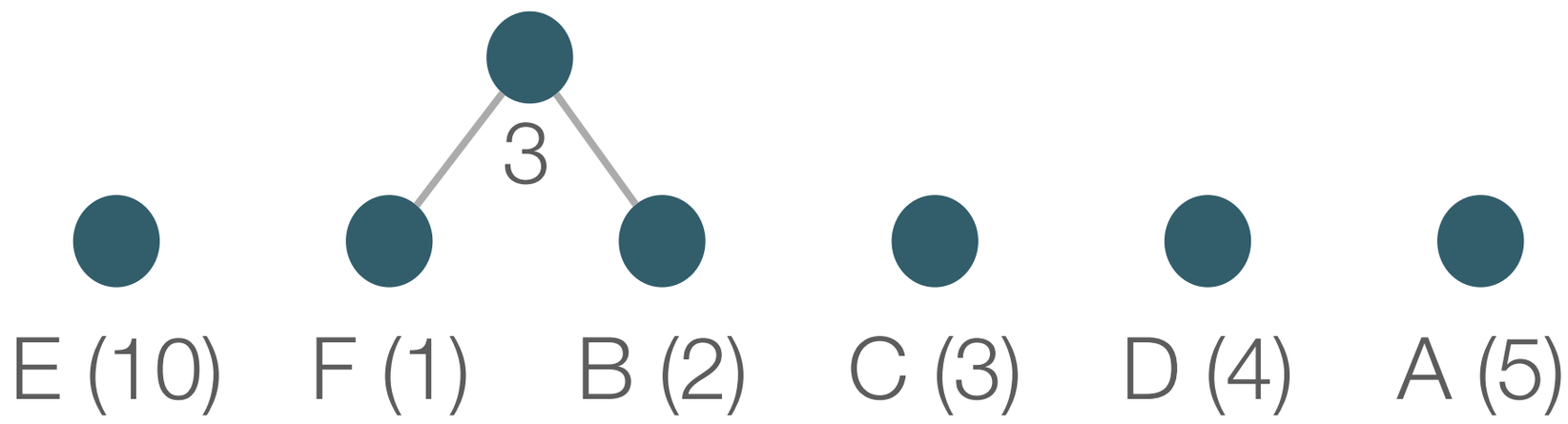
Character	A	B	C	D	E	F
Frequency	5	2	3	4	10	1



Data Compression (cont'd)

The tree built by Huffman's encoding is called a Huffman tree

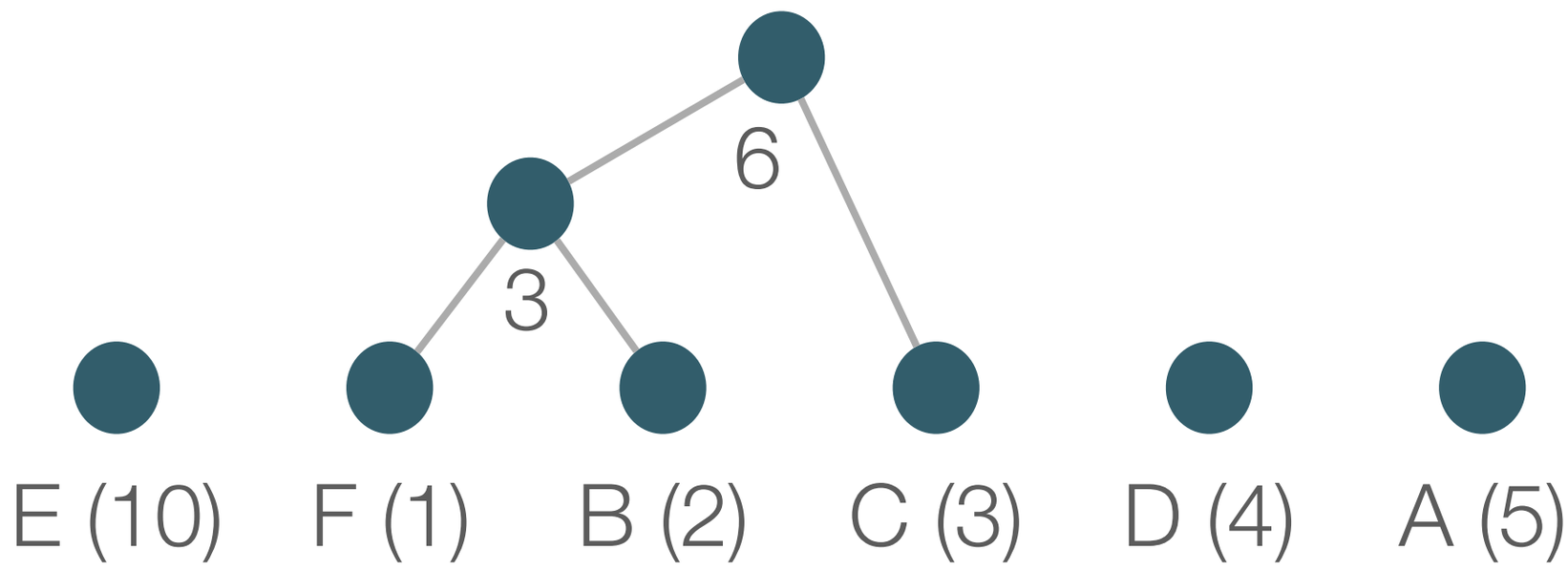
Character	A	B	C	D	E	F
Frequency	5	2	3	4	10	1



Data Compression (cont'd)

The tree built by Huffman's encoding is called a Huffman tree

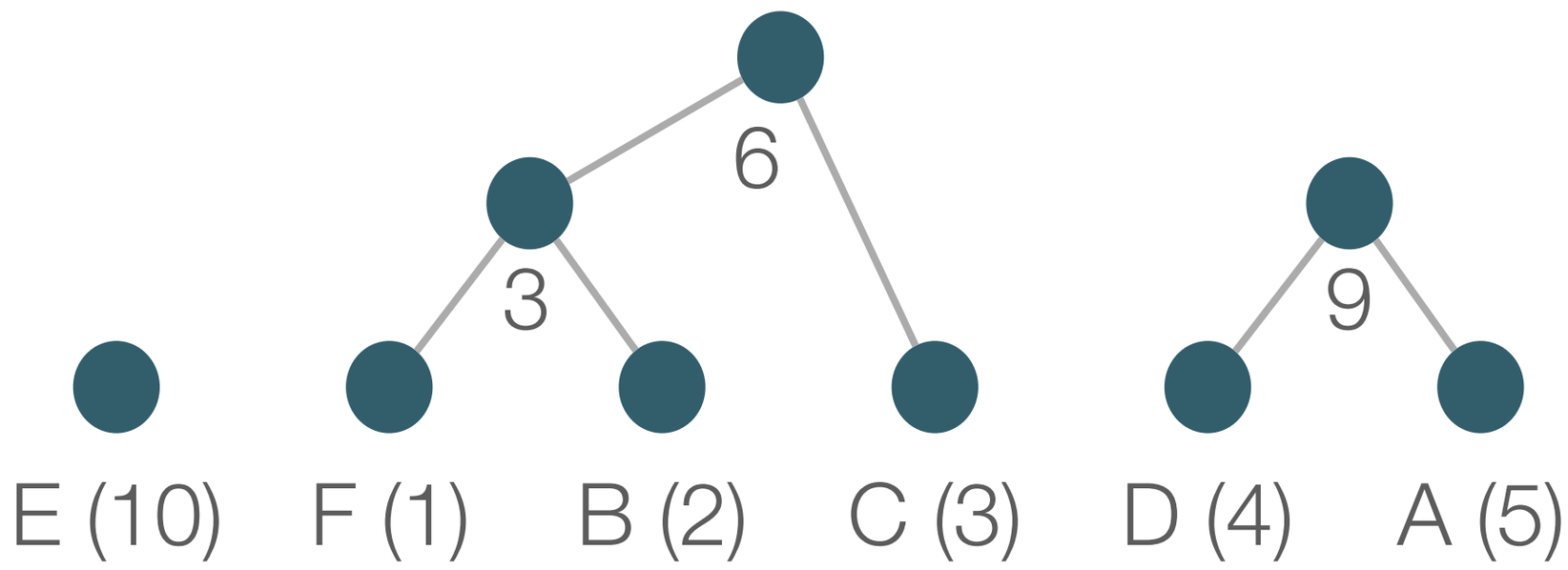
Character	A	B	C	D	E	F
Frequency	5	2	3	4	10	1



Data Compression (cont'd)

The tree built by Huffman's encoding is called a Huffman tree

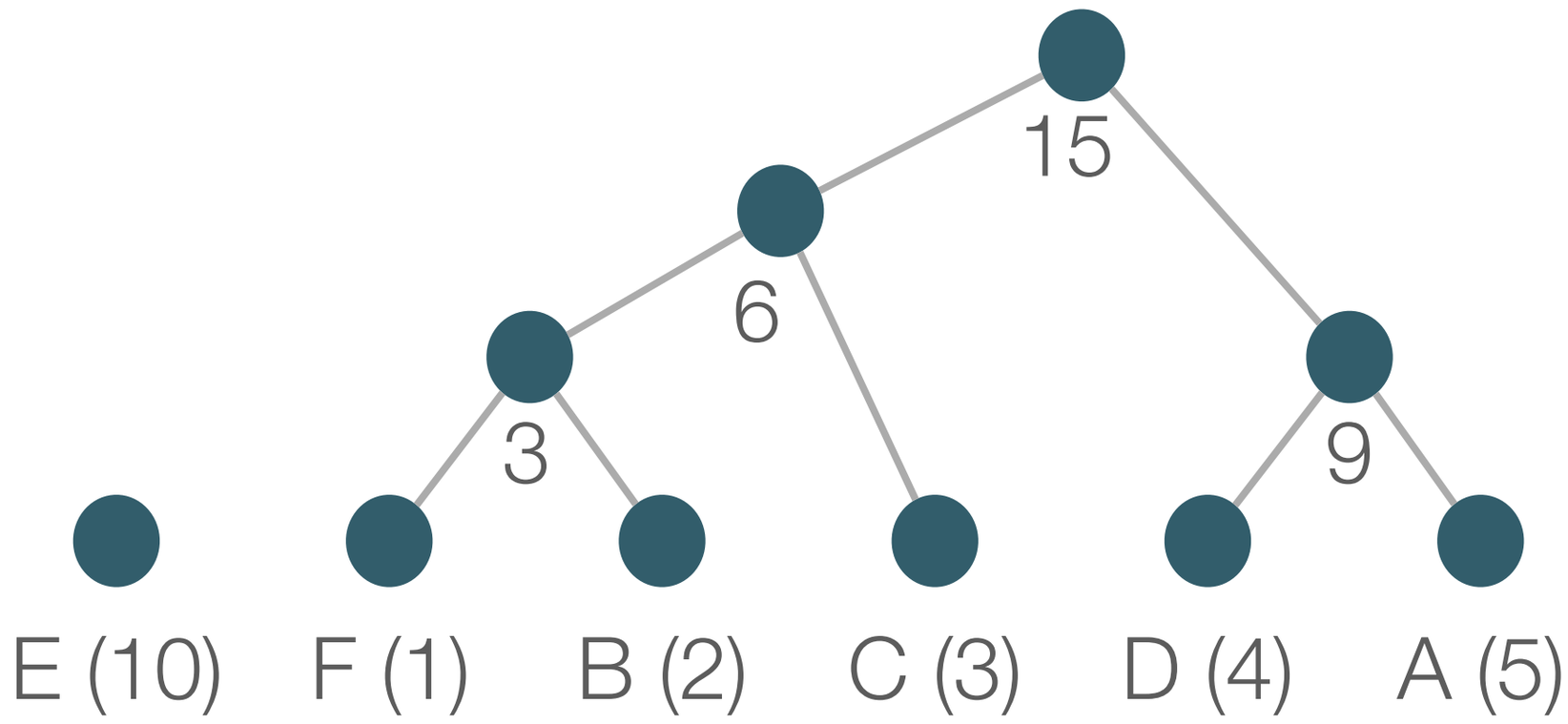
Character	A	B	C	D	E	F
Frequency	5	2	3	4	10	1



Data Compression (cont'd)

The tree built by Huffman's encoding is called a Huffman tree

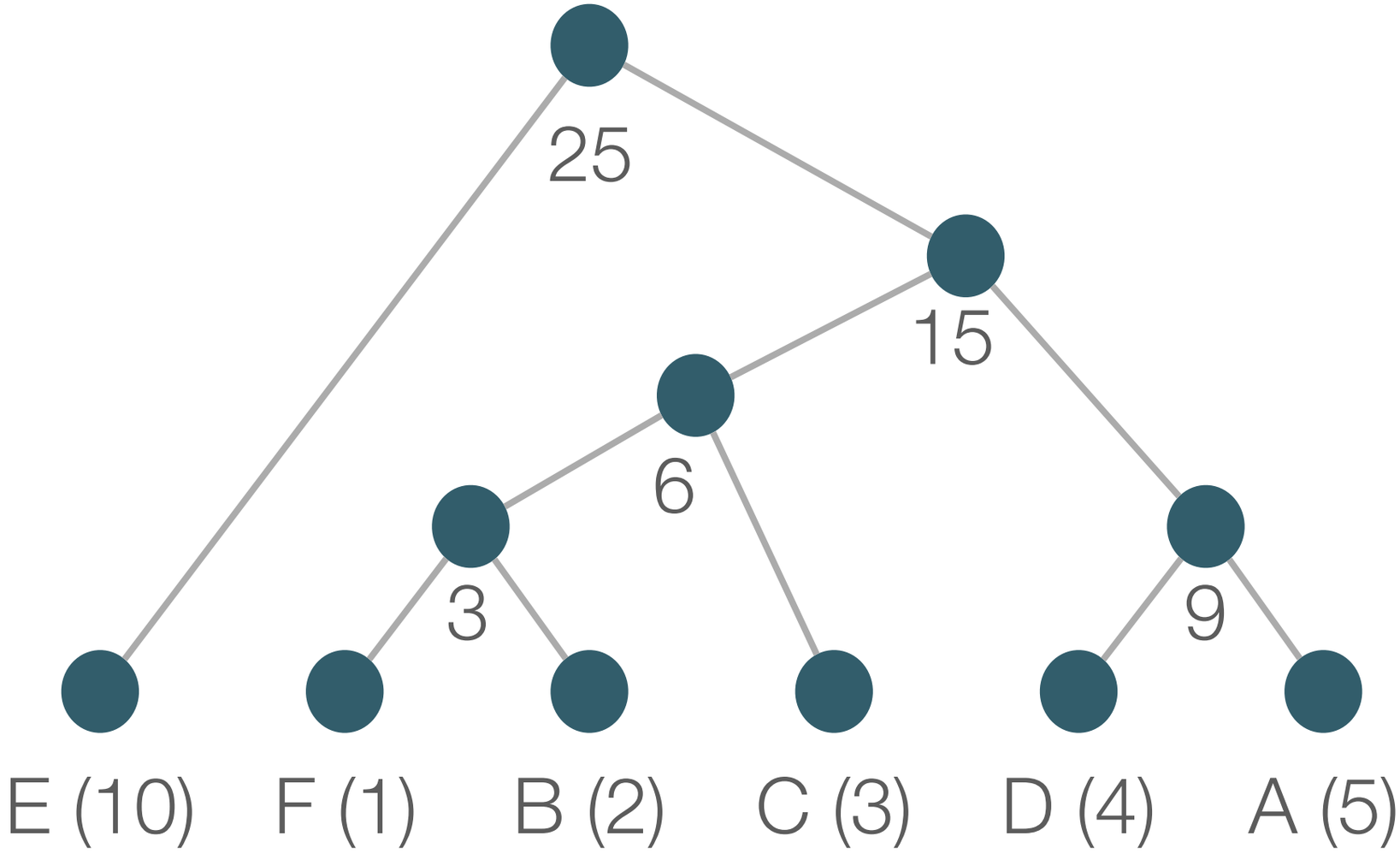
Character	A	B	C	D	E	F
Frequency	5	2	3	4	10	1



Data Compression (cont'd)

The tree built by Huffman's encoding is called a Huffman tree

Character	A	B	C	D	E	F
Frequency	5	2	3	4	10	1



String Matching

Problem

Given two strings $A = a_1a_2\cdots a_n$ and $B = b_1b_2\cdots b_m$ with $m \leq n$, find the first occurrence (if any) of B in A. In other words, find the smallest k such that, for all i, $1 \leq i \leq m$, we have $a_{k+i} = b_i$.

A (nonempty) ***substring*** of a string A is a consecutive sequence of characters $a_ia_{i+1}\cdots a_j$ $(i < j)$ from A

Given a string $A = a_1a_2\cdots a_n$, denote the prefix $a_1a_2\cdots a_i$ by A(i)

Straightforward String Matching

$A = xyxxyxyxyyxyxyxyyxyxyxx.$ $B = xyxyyxyxyxx.$

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
	x	y	x	x	y	x	y	x	y	y	x	y	x	y	x	y	y	x	y	x	y	x	x
1:	x	y	x	y	·	·	·																
2:		x	·	·	·																		
3:			x	y	·	·	·																
4:		x	y	x	y	y	·	·	·														
5:			x	·	·	·																	
6:				x	y	x	y	y	x	y	x	y	x	x									
7:					x	·	·	·															
8:						x	y	x	·	·	·												
9:							x	·	·	·													
10:								x	·	·	·												
11:									x	y	x	y	y	·	·	·							
12:										x	·	·	·										
13:													x	y	x	y	y	x	y	x	y	x	x

Figure 6.20 An example of a straightforward string matching.

Complexity: O(mn)

Straightforward String Matching

$A = xyxxyxyxyyxyxyxyyxyxyxx.$ $B = xyxyyxyxyxx.$

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
	x	y	x	x	y	x	y	x	y	y	x	y	x	y	x	y	y	x	y	x	y	x	x
1:	x	y	x	y	·	·	·																
2:		x	·	·	·																		
3:			x	y	·	·	·																
4:		x	y	x	y	y	·	·	·														
5:			x	·	·	·																	
6:				x	y	x	y	y	x	y	x	y	x										
7:					x	·	·	·															
8:						x	y	x	·	·	·												
9:							x	·	·	·													
10:								x	·	·	·												
11:									x	y	x	y	y	·	·	·							
12:										x	·	·	·										
13:													x	y	x	y	y	x	y	x	y	x	x

Figure 6.20 An example of a straightforward string matching.

Complexity: O(mn)

Matching Against Itself

B = x y x y y x y x y x 

Matching Against Itself

B = x y x y y x y x y x x

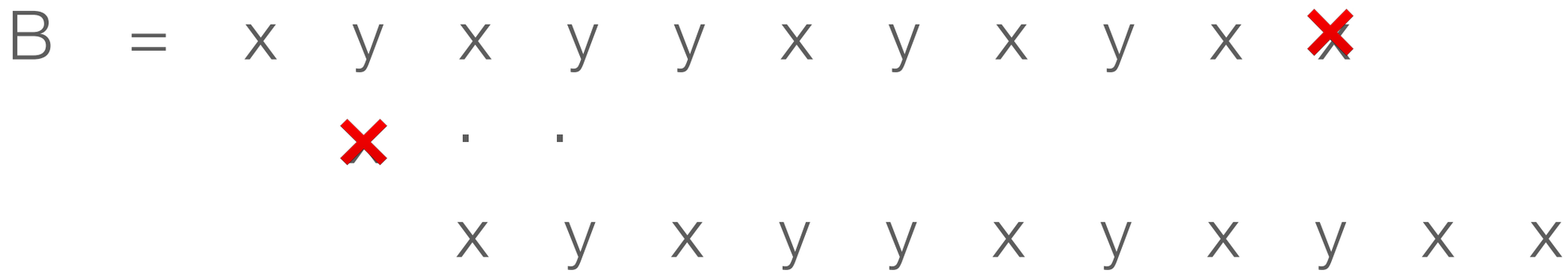
x y x y y x y x y x x

Matching Against Itself

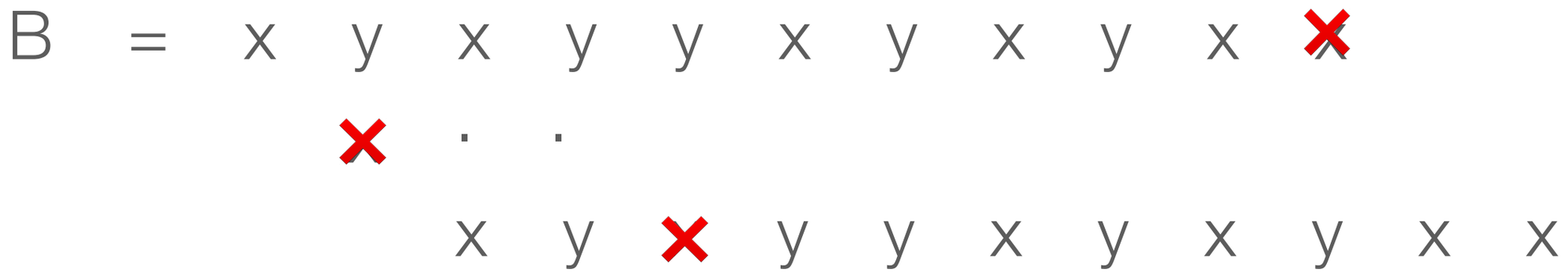
B = x y x y y x y x y x x

x y x y y x y x y x x

Matching Against Itself

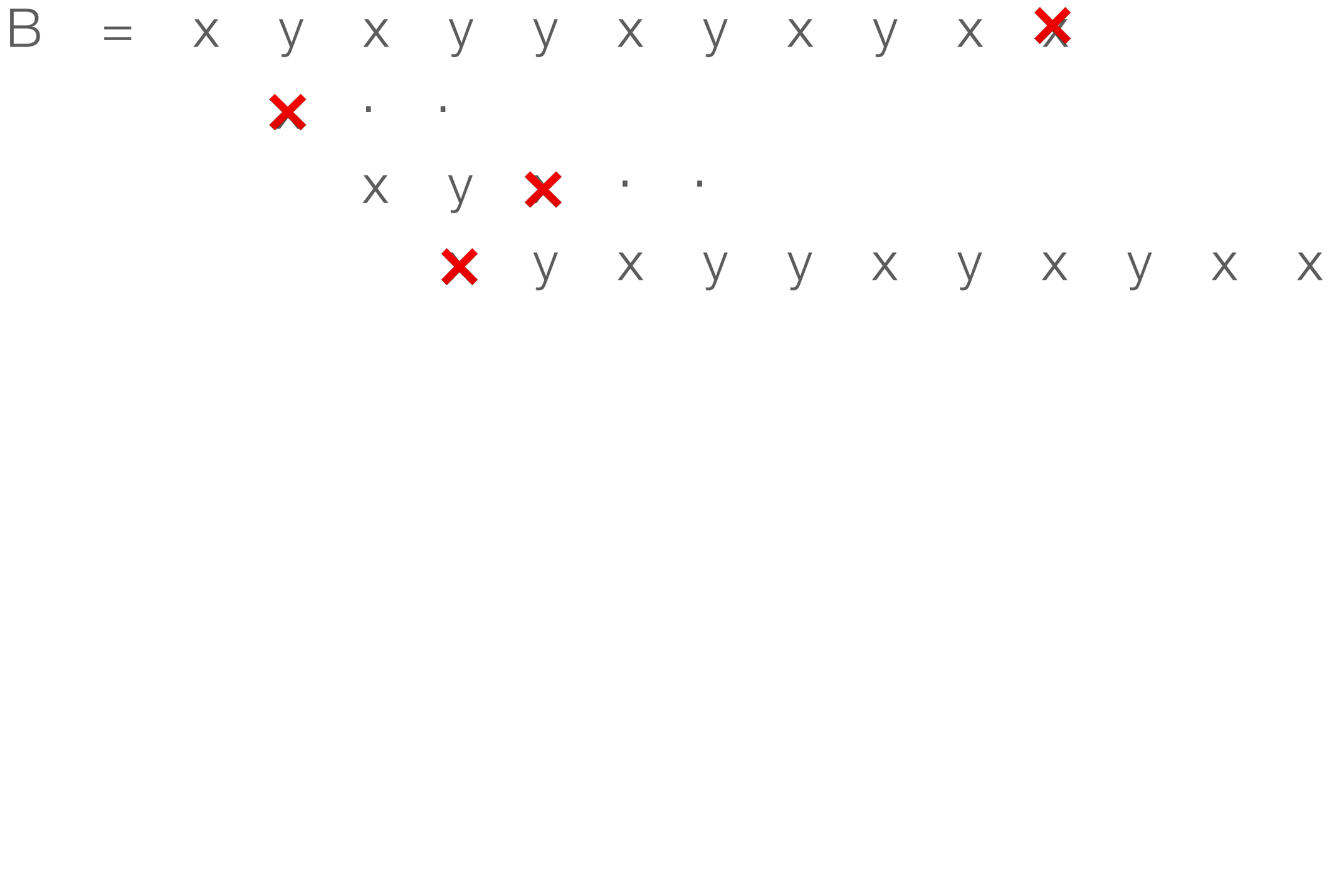


Matching Against Itself



Matching Against Itself

Matching Against Itself



Matching Against Itself

B = x y x y y x y x y x x

x . .

x y x . .

x . .

x y x y y x y x y x x

Matching Against Itself

B = x y x y y x y x y x x

. .

x y .

. .

x y x y y x y x y x x

Matching Against Itself

B = x y x y y x y x y x x

x . .

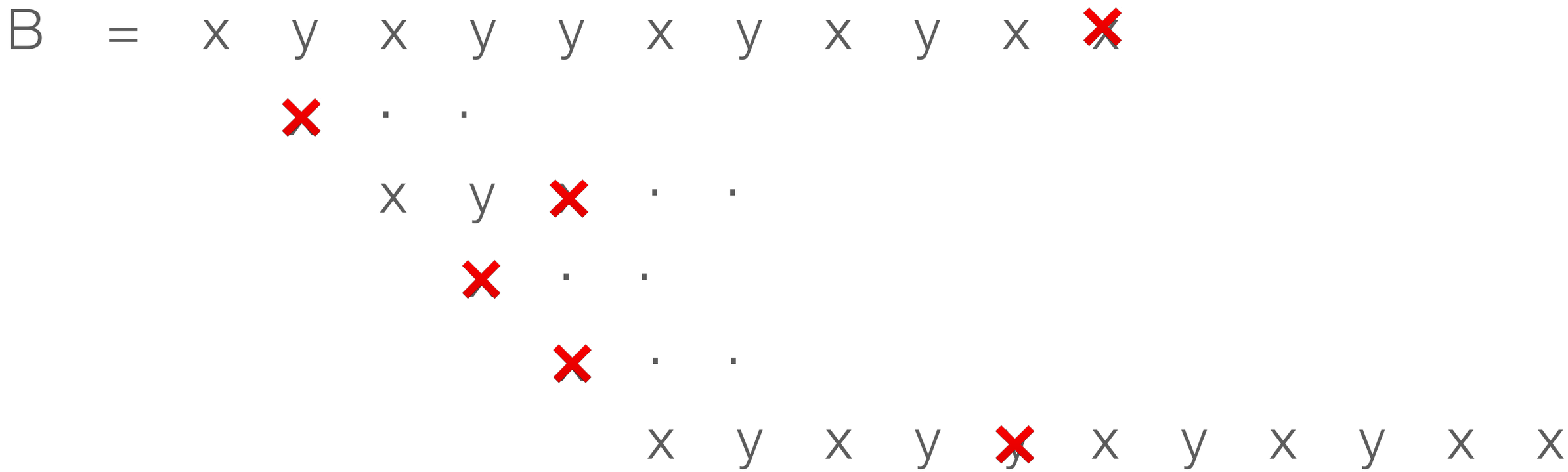
x y x . .

x . .

x . .

x y x y y x y x y x x

Matching Against Itself



Matching Against Itself

B = x y x y y x y x y x x

x . .

x y x . .

x . .

x y x y x

x y x y y x y x y x

Matching Against Itself

B = x y x y y x y x y x x

x . .

x y x . .

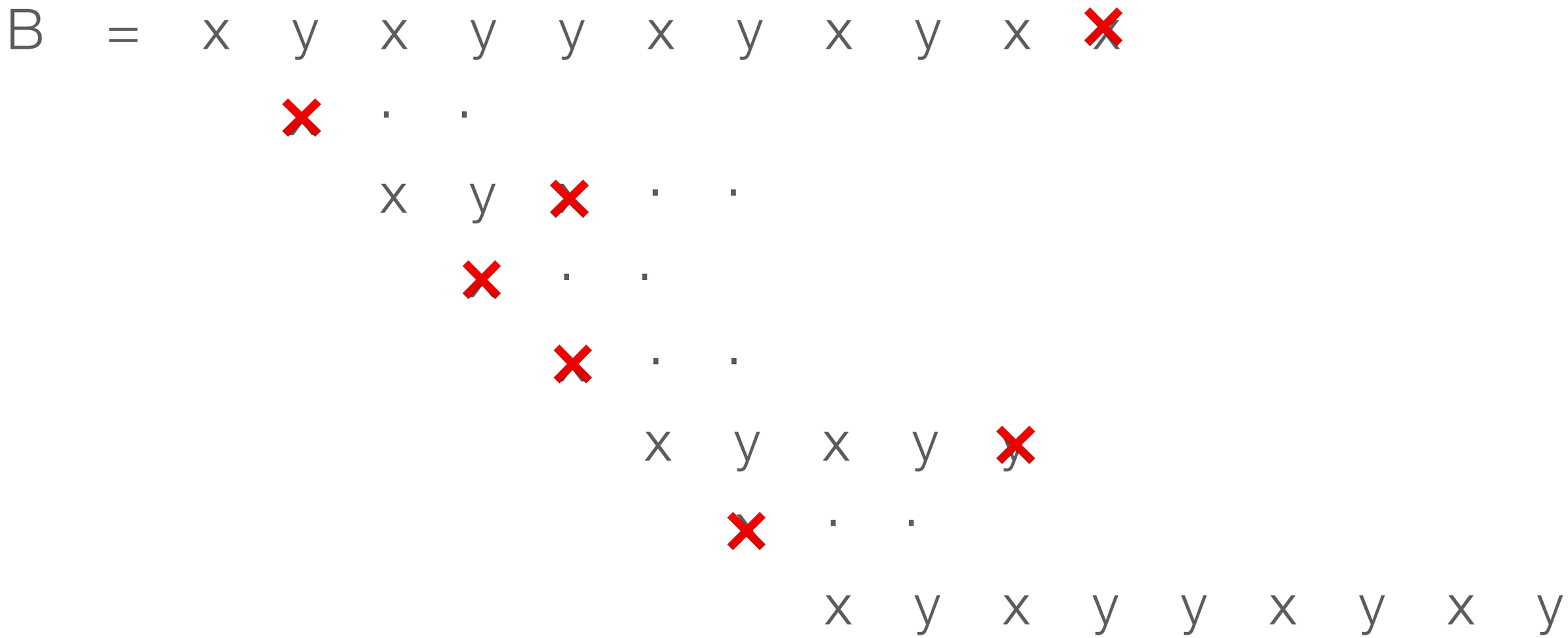
x y . .

x . .

x y x y x

x y x y y x y x y x

Matching Against Itself



Matching Against Itself

B = x y x y y x y x y x x

x . .

x y x . .

x . .

x y x y x

x . .

x y x

Matching Against Itself

B = x y x y y x y x y x x

x . .

x y x . .

x . .

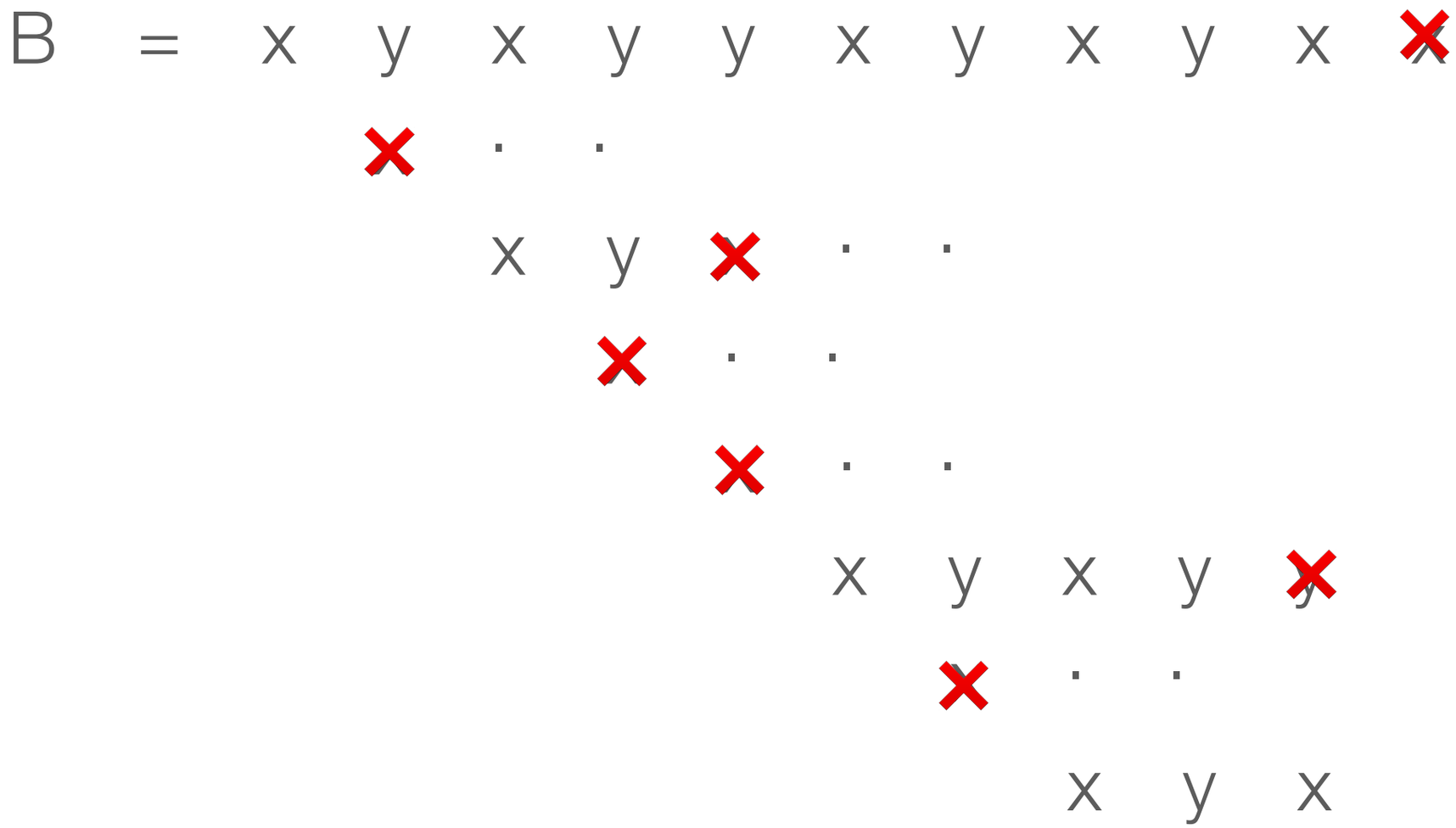
x y x y x

x . .

x y x

xyx is both the suffix and the prefix of B(10)

Matching Against Itself



xyx is both the suffix and the prefix of B(10)

The position of the next sliding can be precomputed

The Values of *next*

next(i) = the maximum j ($0 < j < i - 1$) such that $b_{i-j}b_{i-j+1}\cdots b_{i-1} = b_1b_2\cdots b_j$, and 0 if no such j exists

next(1) is set to -1 so that this unique case is easily differentiated

next(2) is always equal to 0

i	=	1	2	3	4	5	6	7	8	9	10	11
B	=	x	y	x	y	y	x	y	x	y	x	x
$next$	=	-1	0	0	1	2	0	1	2	3	4	3

The Values of *next* (cont'd)

- next(1) = -1, next(2) = 0
- For i > 2,
  - next(i) = next(i - 1) + 1 if $b_{i-1} = b_{next(i - 1) + 1}$
  - next(i) = next(next(i - 1) + 1) + 1 if $b_{i-1} \neq b_{next(i - 1) + 1}$ and $b_{i-1} = b_{next(next(i - 1) + 1) + 1}$
  - ...

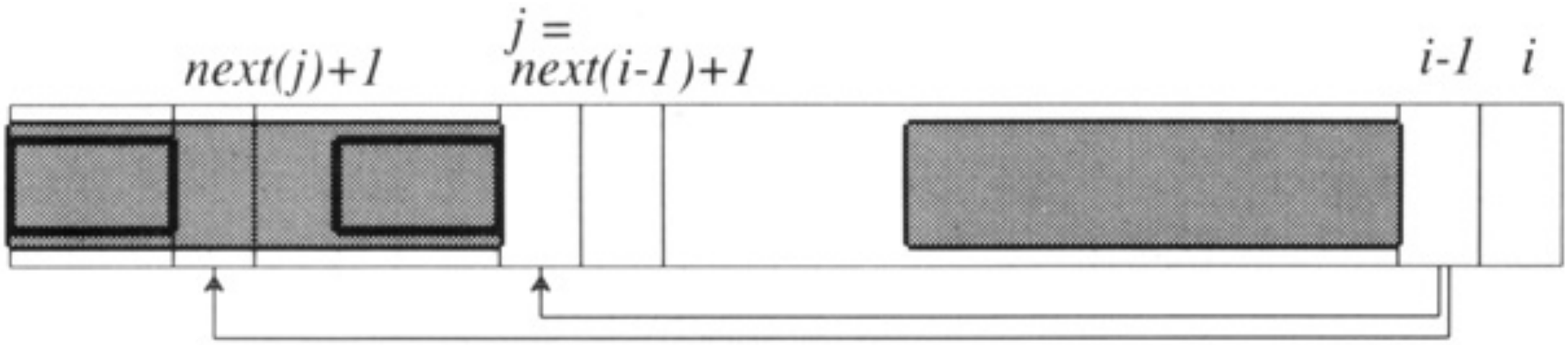


Figure 6.24 Computing next(i).

Source: [Manber 1989]

The Values of *next* (cont'd)

```
Algorithm Compute_Next (B, m);
begin
    next[1] := -1;
    next[2] := 0;
    for i := 3 to m do
        j := next(i - 1) + 1;
        while B[i - 1] ≠ B[j] and j > 0 do
            j := next[j] + 1;
        next[i] := j
end                                          Complexity: O(m)
```

The Values of *next* (cont'd)

- The complexity of Compute_Next is O(m)
  - Consider the relation between i and i - j
  - i ≥ i - j must hold
  - After one iteration of the for loop, i is incremented by 1
  - After one iteration of the while loop, j decreases
  - The initial value of i - j is 2
  - The maximum value of i is m

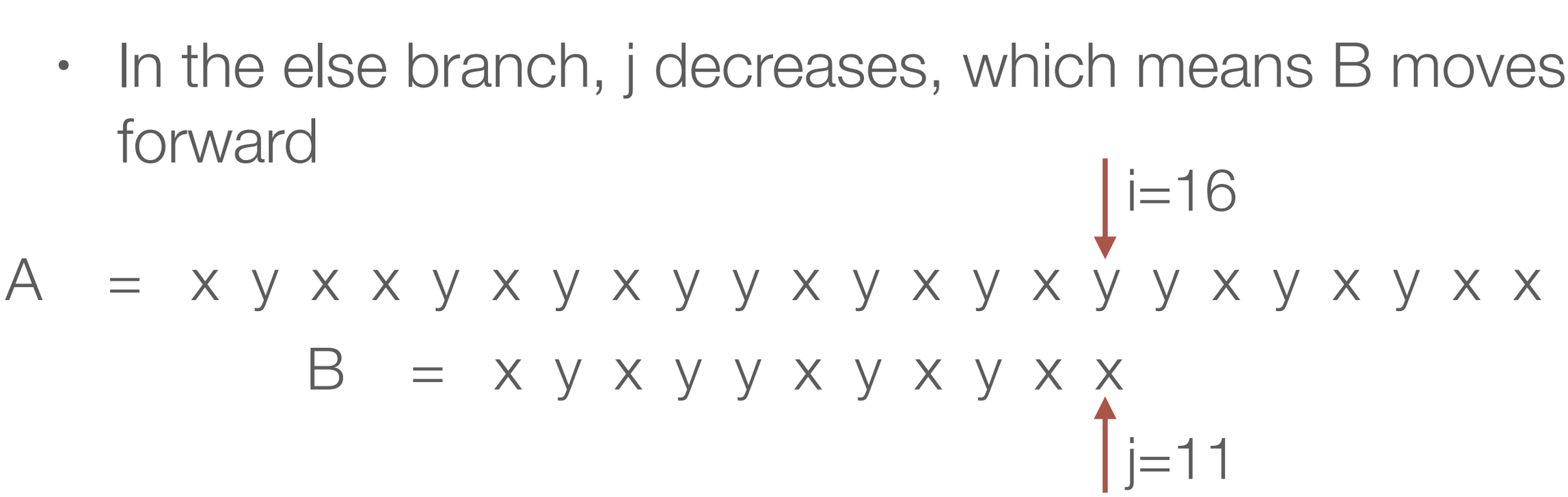
The KMP Algorithm

```
Algorithm String_Match (A, n, B, m);
begin
    j := 1; i := 1;
    Start := 0;
    while Start = 0 and i ≤ n do
        if B[j] = A[i] then
            j := j + 1;
            i := i + 1
        else
            j := next[j] + 1;
            if j = 0 then
                j := 1;
                i := i + 1;
        if j = m + 1 then Start := i - m
end
```

Complexity: O(n)

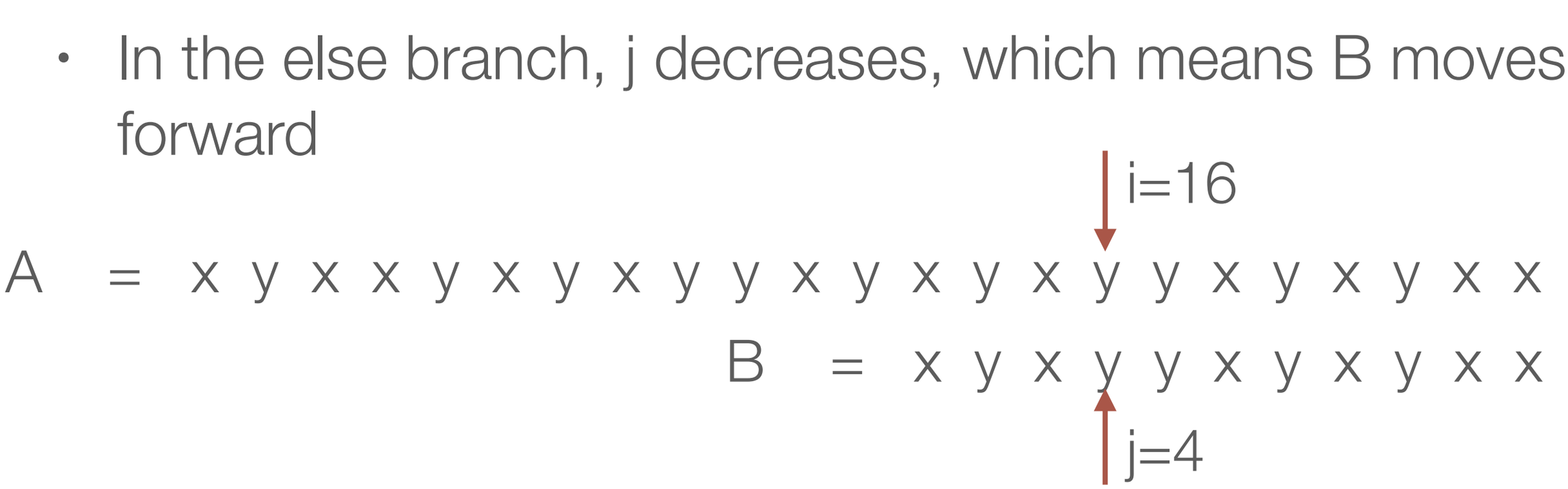
The KMP Algorithm

- The complexity of the KMP algorithm is O(n)
  - In the B[j] = A[i] branch, i is incremented by 1
  - In the else branch, j decreases, which means B moves forward



The KMP Algorithm

- The complexity of the KMP algorithm is O(n)
  - In the B[j] = A[i] branch, i is incremented by 1
  - In the else branch, j decreases, which means B moves forward



String Editing

Problem

Given two strings $A = a_1a_2\cdots a_n$ and $B = b_1b_2\cdots b_m$, find the minimum number of changes required to change A character by character such that it becomes equal to B

Three types of changes (or edit steps) allowed:

1. insert
2. delete
3. replace

String Editing (cont'd)

- How to change A(n) to B(m) by induction?
  - Change A(n - 1) to B(m) and then delete a_n
  - Change A(n - 1) to B(m - 1) and then change a_n to b_m
  - Change A(n) to B(m - 1) and then insert b_n
- Which one requires the minimum number of changes?
- We need to compute the cost

String Editing (cont'd)

Let C(i, j) denote the minimum cost of changing A(i) to B(j)

$$C(i,j) = min \begin{cases} C(i-1,j)+1 & (\text{deleting } a_i) \\ C(i,j-1)+1 & (\text{inserting } b_j) \\ C(i-1,j-1)+1 & (a_i \rightarrow b_j) \\ C(i-1,j-1) & (a_i = b_j) \end{cases}$$

String Editing (cont'd)

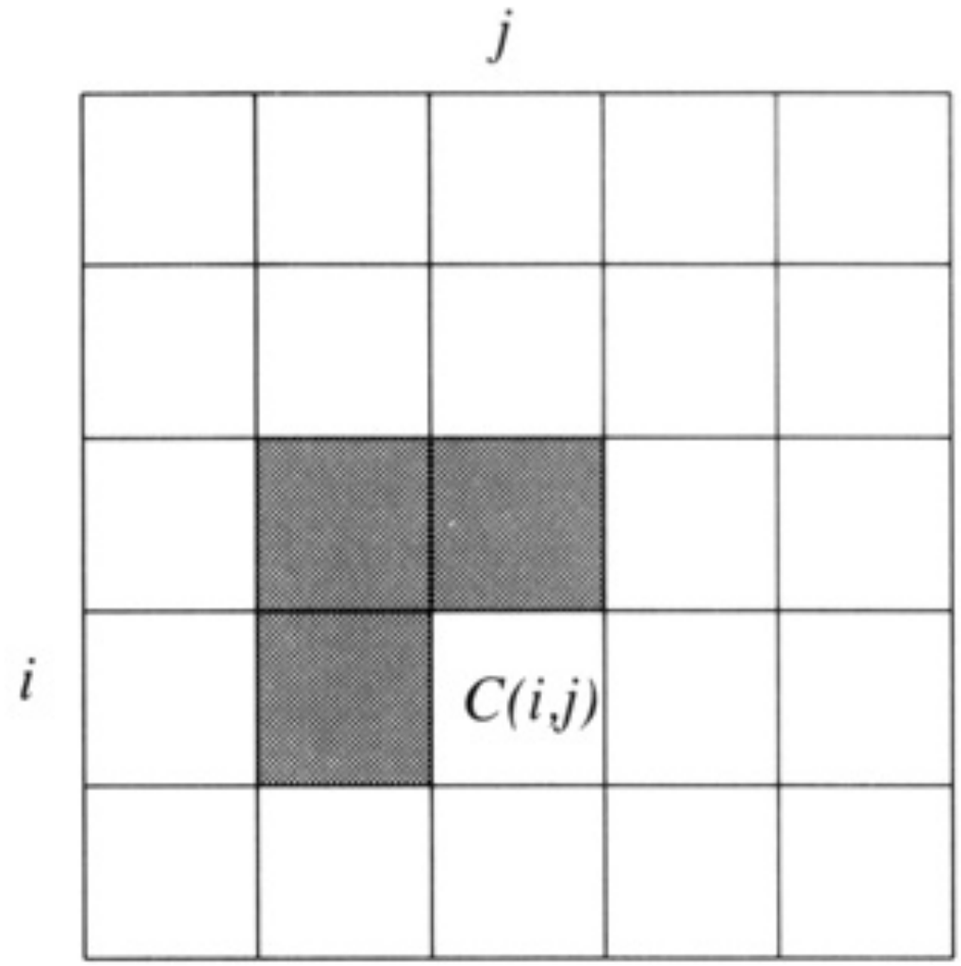


Figure 6.26 The dependencies of $C(i, j)$.

Source: [Manber 1989]

String Editing (cont'd)

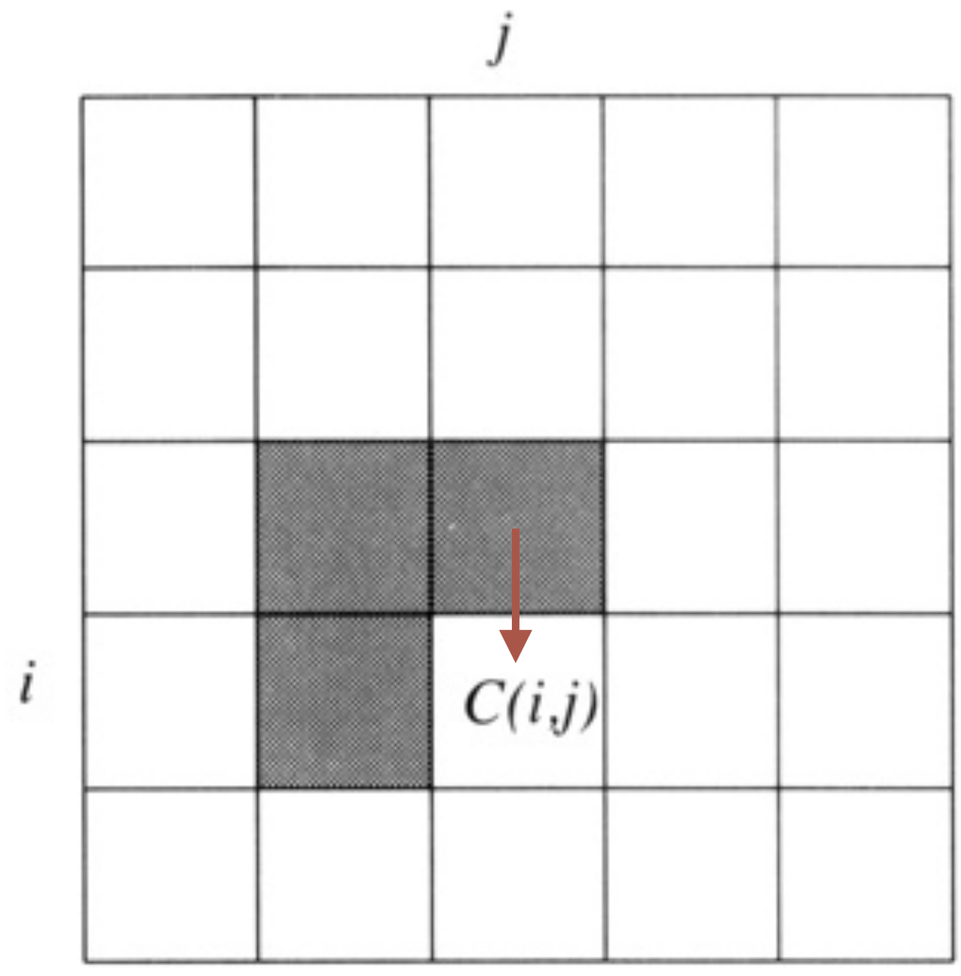


Figure 6.26 The dependencies of $C(i, j)$.

If C(i-1, j) is the minimum, the best way to change A(i) to B(j) is as follows.

1. Change A(i - 1) to B(j)
2. Delete a_i

Source: [Manber 1989]

String Editing (cont'd)

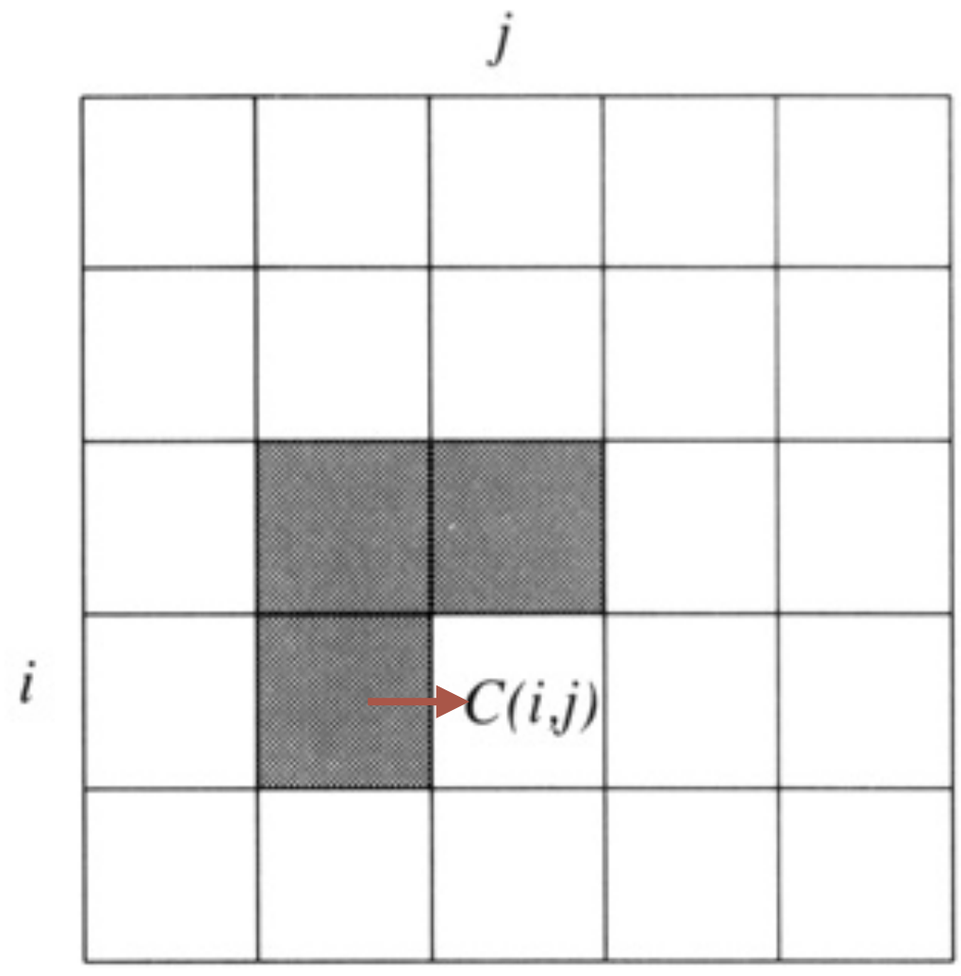


Figure 6.26 The dependencies of $C(i, j)$.

If C(i, j - 1) is the minimum, the best way to change A(i) to B(j) is as follows.

1. Change A(i) to B(j - 1)
2. Insert b_j

Source: [Manber 1989]

String Editing (cont'd)

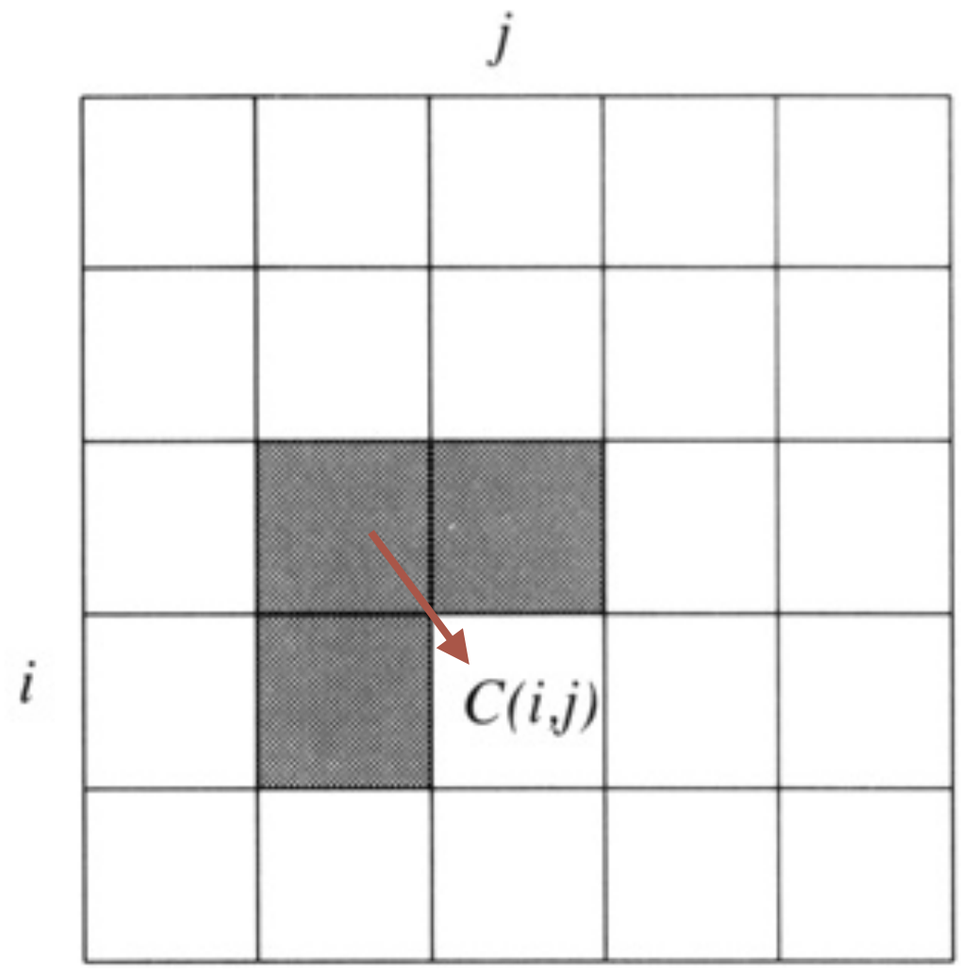


Figure 6.26 The dependencies of $C(i, j)$.

If C(i - 1, j - 1) + 1 is the minimum, the best way to change A(i) to B(j) is as follows.

1. Change A(i - 1) to B(j - 1)
2. Replace a_i by b_j

Source: [Manber 1989]

String Editing (cont'd)

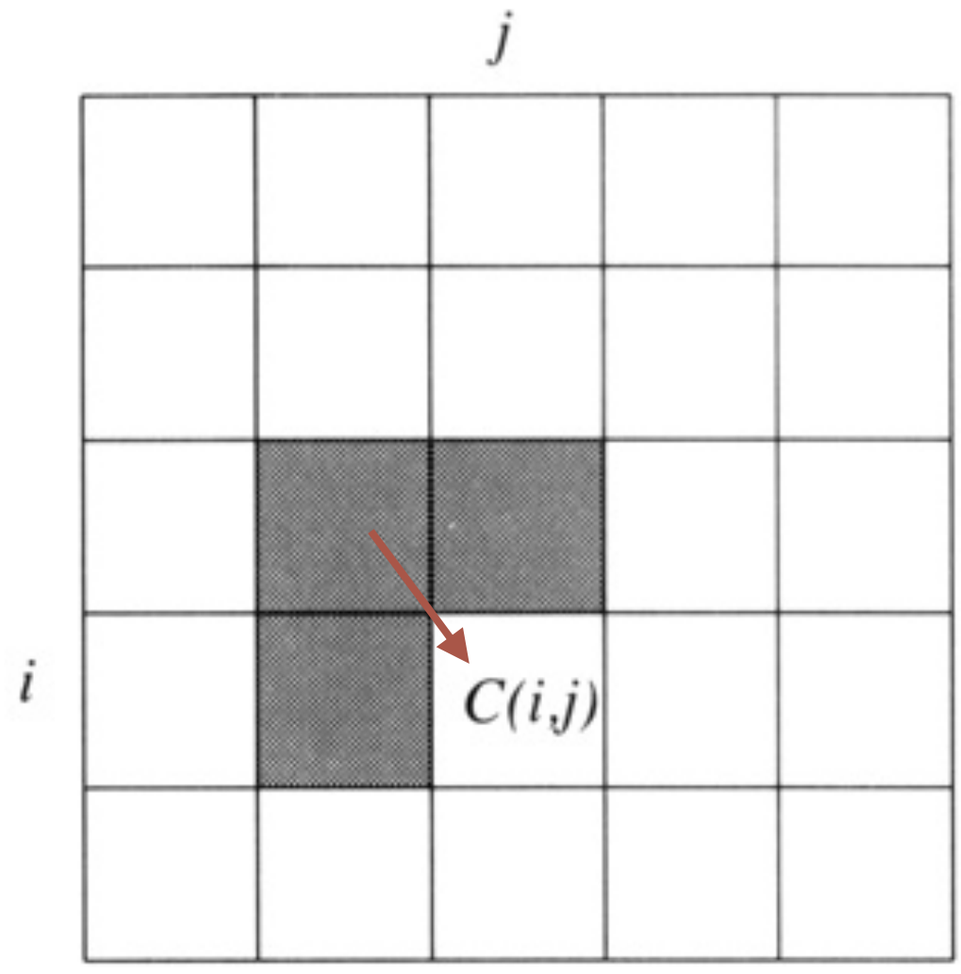


Figure 6.26 The dependencies of $C(i, j)$.

If $a_i = b_j$ and C(i - 1, j - 1) is the minimum, the best way to change A(i) to B(j) is as follows.

1. Change A(i - 1) to B(j - 1)

Source: [Manber 1989]

String Editing (cont'd)

```
Algorithm Minimum_Edit_Distance (A, n, B, m);
begin
    for i := 0 to n do C[i, 0] := i;
    for j := 1 to m do C[0, j] := j;
    for i := 1 to n do
        for j := 1 to m do
            x := C[i - 1, j] + 1
            y := C[i, j - 1] + 1;
            if a_i = b_j then z := C[i - 1, j - 1]
            else z := C[i - 1, j - 1] + 1
            C[i, j] := min(x, y, z)
end
```

Complexity: O(nm)