University of BRISTOL | Department of Computer Science

PROGRAMMING and ALGORITHMS II

INTRODUCTION TO

DYNAMIC PROGRAMMING

Dr Tilo Burghardt

Unit Code COMS10001

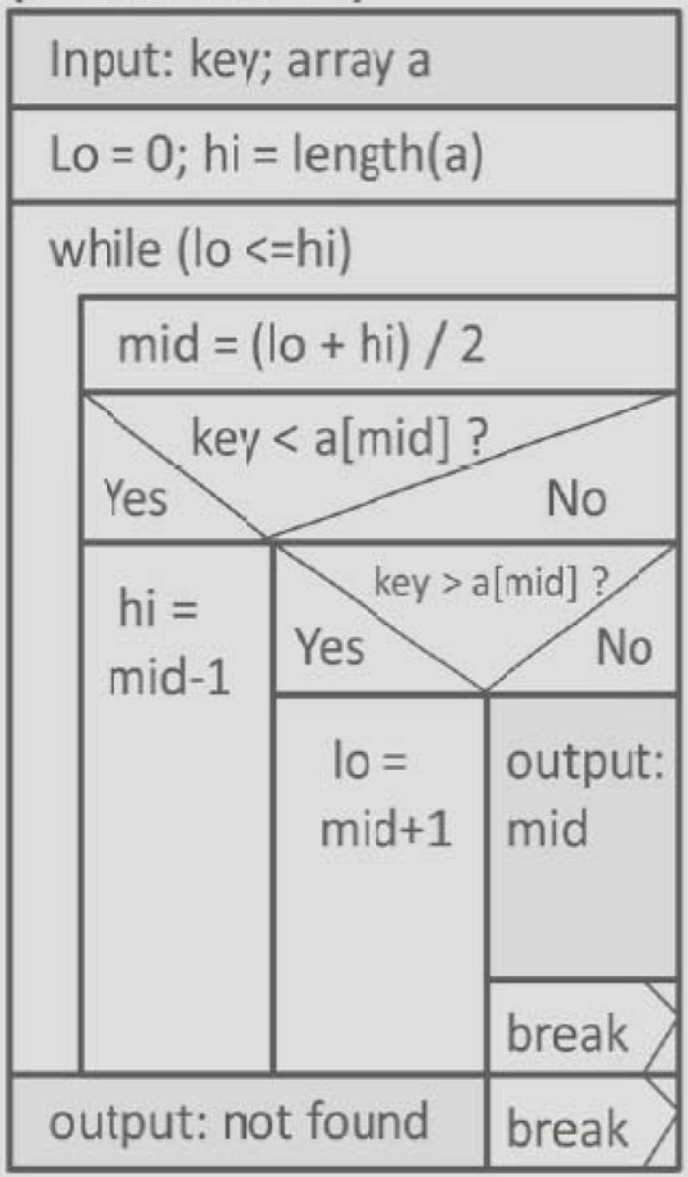
Recap: Simple Binary Search

University of BRISTOL | Department of Computer Science

Code Fragment (Java)

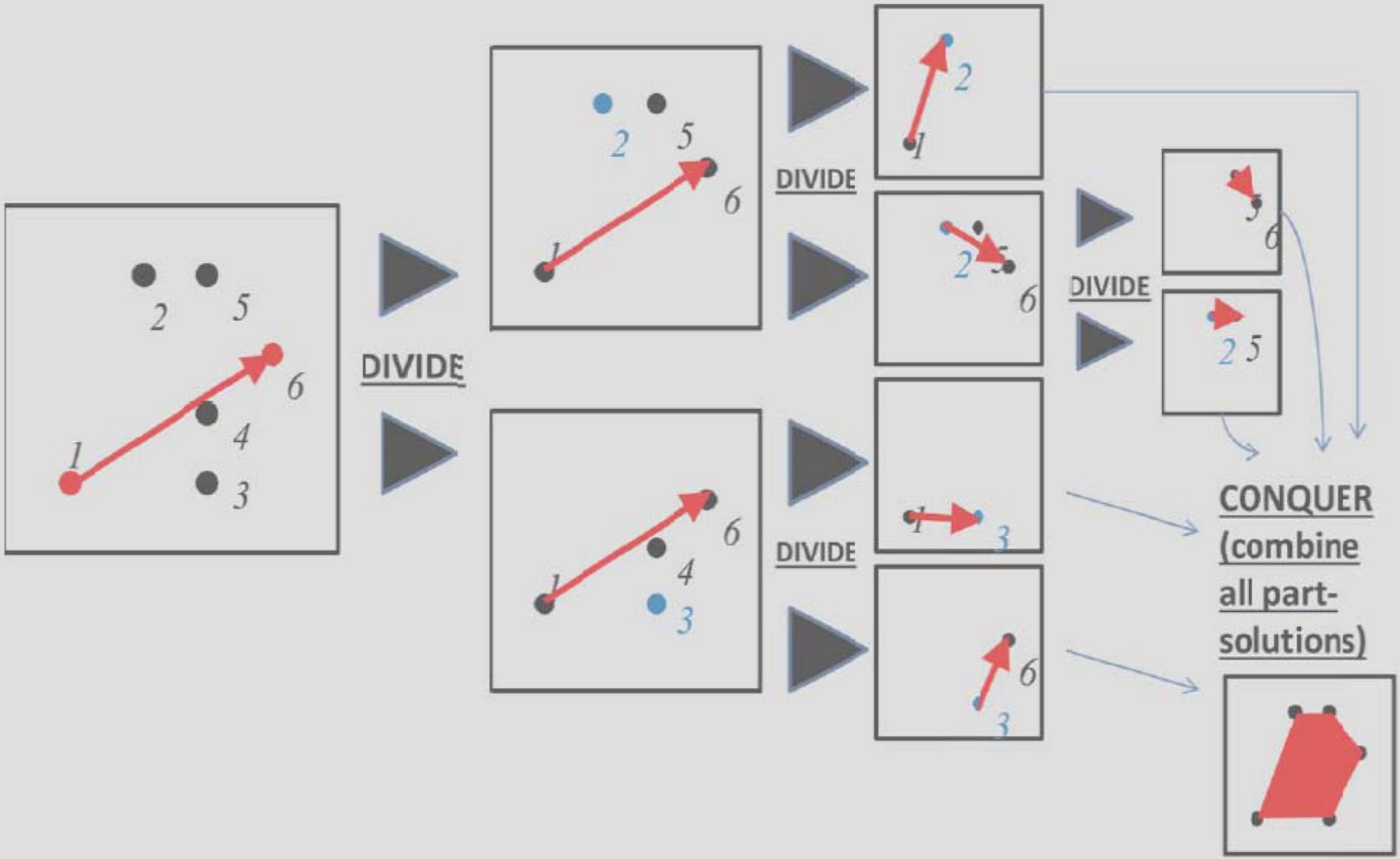
```
class Library {
  int[] a; //book keys
  ...
  int find(int key) {
    int lo = 0;
    int hi = a.length - 1;
    while (lo <= hi) {
      int mid = lo + (hi - lo) / 2;
      //discard upper array part
      if (key < a[mid]) hi = mid - 1;
      //discard lower array part
      else if (key > a[mid]) lo = mid + 1;
      //key found
      else return mid;
    }
    //key not in array
    return -1;
  }...}
```

Nassi-Shneiderman-Diagram (structure chart)



Recap: Quick Hull Algorithm

University of BRISTOL | Department of Computer Science

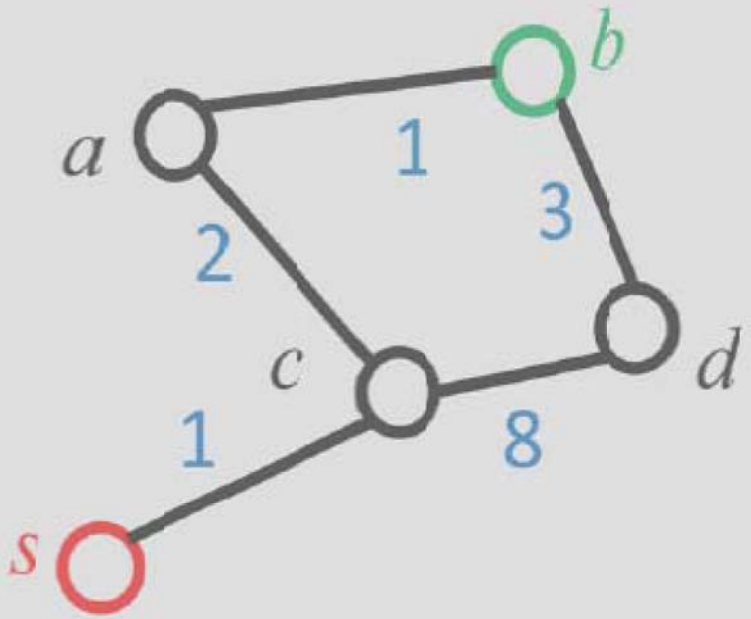


DIVIDE and CONQUER in four phases

1) **PREPARE**
   Transforming the problem into a form suitable for structured subdivision
2) **DIVIDE**
   Recursively breaking the problem into sub-problems that are similar to the original problem but smaller in size,
3) **SOLVE**
   Compute solutions for the sub-problems successively and independently,
4) **CONQUER**
   Combine these solutions to create one solution to the original, larger problem.

Recap: Dijkstra's Algorithm

University of BRISTOL | Department of Computer Science



CANDIDATE SET (INVERSE)

$$W = \{s, c, a, b\}$$

Evolving SOLUTION SET

$$D(a) = 3$$
$$D(b) = 4$$
$$D(c) = 1$$
$$D(d) = 7$$
$$D(s) = 0$$

Dynamic SELECTION & FEASIBILITY FUNCTION

SOLUTION FUNCTION

Exit if $V = W$?

Update Distances: $\forall_{v \in V \setminus W} : D(v) = \min(D(v), D(b) + d(b, v))$

GREEDY ALORITHMS in five parts

1) **CANDIDATE SET**
   ...from which candidates for a solution are sampled
2) **SOLUTION SET**
   ...which holds current elements which contribute to a solution
3) **SELECTION FUNCTION**
   ...which chooses current best candidate to potentially add to the solution
4) **FEASIBILITY FUNCTION**
   ...that is used to determine if a candidate can be used to contribute to a solution
5) **OBJECTIVE/SOLUTION FUNCTION**
   ...which assigns a fitness value to a current solution and determines when the solution is complete

A Knapsack

capacity

W = 11 *(kg)*

Important Things

University of BRISTOL | Department of Computer Science

Packing Important Things

University of BRISTOL | Department of Computer Science

Item i	Value v_i	Weight w_i
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

Greedy Approaches ...

capacity
W = 11

Item i	Value v_i	Weight w_i
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

a) highest value first

b) smallest weight first

c) highest value/weight ratio first

→ **non-optimal solutions**

Limitations of 'Greedy'

- for many problems greedy algorithms fail to produce optimal solutions
- in general, the greedy approach produces a `local' optimum which may be a bad case globally
  - e.g. gradient decent algorithms (tracing a function to lower and lower value down the `gradient') rarely provide the absolute global minimum
- however, if approximations are sufficient then greedy algorithms often deliver a fast way of computing results even for some of the toughest problems

Subproblem Structure

University of BRISTOL | Department of Computer Science

Item i	Value v_i	Weight w_i
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

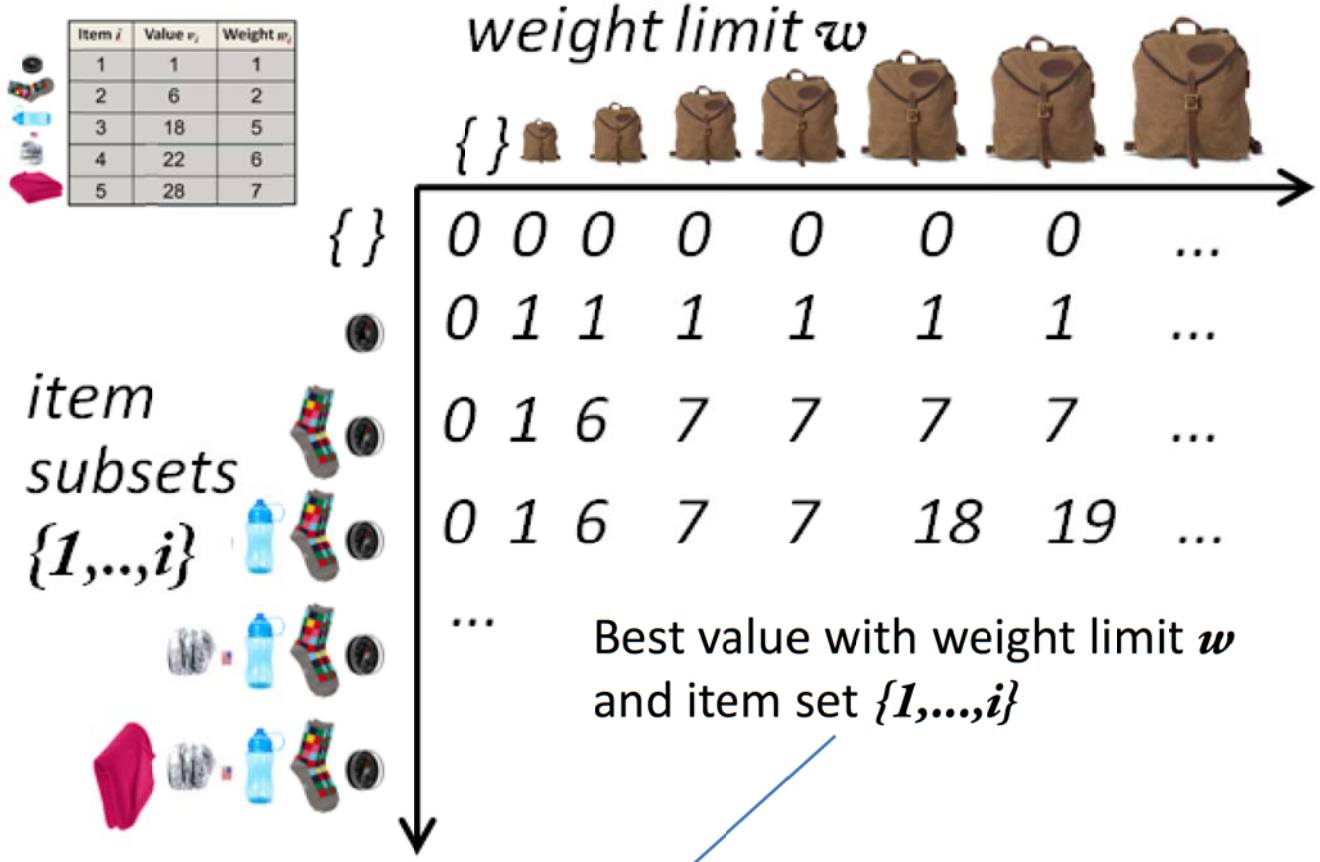
weight limit w

item subsets {1,..,i}

{ }	0	0	0	0	0	0	0	...
	0	1	1	1	1	1	1	...
	0	1	6	7	7	7	7	...
	0	1	6	7	7	18	19	...
...								

This array ***best(i,w)*** holds at each entry the best value with weight limit ***w*** and item set ***{1,...,i}***, for instance ***18*** is the best value for ***i=4*** and ***w=5***

Recurrence Relation



$$best(i,w) = \begin{cases} 0 & if\ (i = 0) \\ best(i-1,w) & if\ (w_i > w) \\ \max[best(i-1,w), v_i + best(i-1, w-w_i)] & otherwise \end{cases}$$

Bottom-Up Approach

Stepwise Result Calculation

University of BRISTOL | Department of Computer Science

Item i	Value v_i	Weight w_i
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

weight limit w

item subsets {1,..,i}	{ }	1	2	3	4	5	6	7	8	9	10	11
{ }	0	0	0	0	0	0	0	0	0	0	0	0
	0	1	1	1	1	1	1	1	1	1	1	1
	0	1	6	7	7	7	7	7	7	7	7	7
	0	1	6	7	7	18	19	24	25	25	25	25
	0	1	6	7	7	18	22	24	28	29	29	40
	0	1	6	7	7	18	22	28	29	34	34	40

$$best(i,w) = \begin{cases} 0 & if\,(i=0) \\ best(i-1,w) & if\,(w_i > w) \\ \max[best(i-1,w), v_i + best(i-1, w-w_i)] & otherwise \end{cases}$$

Tracing the Result

University of BRISTOL | Department of Computer Science

Item i	Value v_i	Weight w_i
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

weight limit w

item subsets {1,..,i}	{ }	1	2	3	4	5	6	7	8	9	10	11
{ }	0	0	0	0	0	0	0	0	0	0	0	0
	0	1	1	1	1	1	1	1	1	1	1	1
	0	1	6	7	7	7	7	7	7	7	7	7
	0	1	6	7	7	18	19	24	25	25	25	25
	0	1	6	7	7	18	22	24	28	29	29	40
	0	1	6	7	7	18	22	28	29	34	34	40

$$best(i,w) = \begin{cases} 0 & if\,(i=0) \\ best(i-1,w) & if\,(w_i > w) \\ \max[best(i-1,w), v_i + best(i-1, w-w_i)] & otherwise \end{cases}$$

Knapsack Algorithm

KNAPSACK $(n, W, w_1, \ldots, w_n, v_1, \ldots, v_n)$

FOR $w = 0$ TO W

  $best[0, w] \leftarrow 0$.

FOR $i = 1$ TO n

  FOR $w = 1$ TO W

  IF $(w_i > w)$ $best[i, w] \leftarrow best[i-1, w]$.

  ELSE $best[i, w] \leftarrow \max \{ best[i-1, w],\ v_i + best[i-1, w-w_i] \}$.

RETURN $best[n, W]$.

Historical Note

Richard Bellman pioneered the systematic study of dynamic programming during the 1950s.

THE THEORY OF DYNAMIC PROGRAMMING

RICHARD BELLMAN

1. **Introduction.** Before turning to a discussion of some representative problems which will permit us to exhibit various mathematical features of the theory, let us present a brief survey of the fundamental concepts, hopes, and aspirations of dynamic programming.

To begin with, the theory was created to treat the mathematical problems arising from the study of various multi-stage decision processes, which may roughly be described in the following way: We have a physical system whose state at any time t is determined by a set of quantities which we call state parameters, or state variables. At certain times, which may be prescribed in advance, or which may be determined by the process itself, we are called upon to make decisions which will affect the state of the system. These decisions are equivalent to transformations of the state variables, the choice of a decision being identical with the choice of a transformation. The outcome of the preceding decisions is to be used to guide the choice of future ones, with the purpose of the whole process that of maximizing some function of the parameters describing the final state.

Examples of processes fitting this loose description are furnished by virtually every phase of modern life, from the planning of industrial production lines to the scheduling of patients at a medical clinic; from the determination of long-term investment programs for universities to the determination of a replacement policy for machinery in factories; from the programming of training policies for skilled and unskilled labor to the choice of optimal purchasing and inventory policies for department stores and military establishments.

Richard Bellman
(1920 - 1984)

General Scheme of

Dynamic Programming

Step1: Determine Structure

Characterize the structure of an optimal solution by showing that it can be decomposed into optimal sub-problems.

Step2: Find Recurrence Relation

Recursively define the value of an optimal solution by expressing it in terms of optimal solutions for smaller problems.

Step 3: Bottom-up Computation

Compute the value of an optimal solution in a bottom-up fashion by using a table structure.

Step 4: Construction of optimal solution

Construct an optimal solution from computed information.

Definition of Grammars

A **Formal Grammar** G is a structure (4-tupel):

$$G = (\ V,\ T,\ P,\ S\ )$$

variables (=non-terminals)	terminal symbols	productions (=rules)	sentence symbol (=start symbol)
$\{A, B, C, \ldots\}$	$\{a, b, c, \ldots\}$	$\{p_1, p_2, \ldots\}$	$S \in V$

Disjunction: $V \cap T = \varnothing$

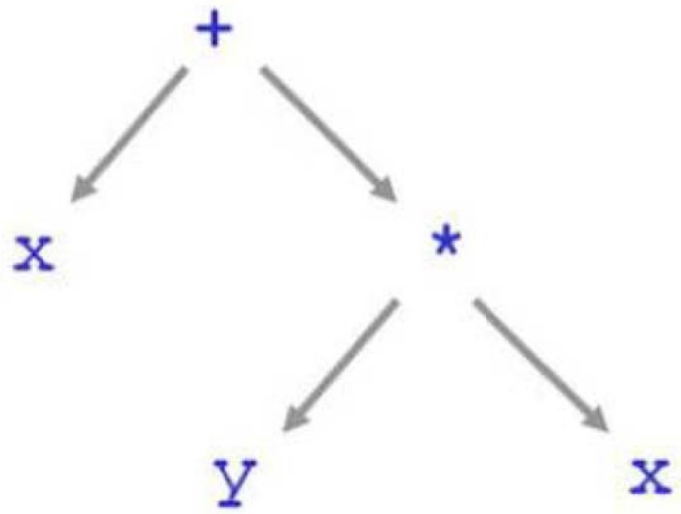
Created Language: $L(G) = \{\underline{w} \in T^* \mid S \Rightarrow_G \underline{w}\}$

Grammar Example

$$G = (\{E, M, F\}, \{x, y, (,), +, *\}, P, E)$$

$$P = \{E \to M,$$
$$E \to E + M,$$
$$M \to F,$$
$$M \to M * F,$$
$$F \to x,$$
$$F \to y,$$
$$F \to (E)\}$$

$$\underline{w} = x + y * x \in L(G)$$



Chomsky Hierarchy

TM

- Type 0: recursively enumerable
  $X \rightarrow Y$

LBA

- Type 1: context-sensitive
  $XAY \rightarrow XZY$

PDA

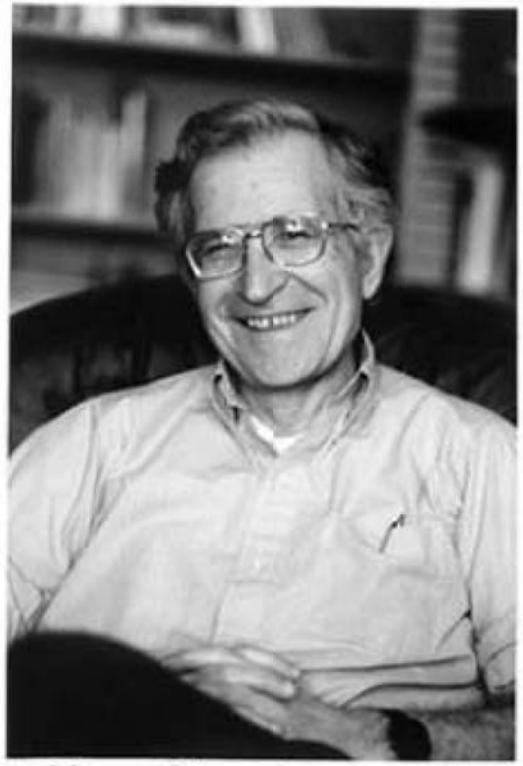
- Type 2: context-free
  $A \rightarrow X$

DFA

- Type 3: regular
  $A \rightarrow a \; ; A \rightarrow aB$

  ↓

  trivial



A Noam Chomsky

$X,Y,Z\ldots$ strings of terminals and non-terminals

$a,b,c\ \ldots$ terminals

$A,B,C\ldots$ non-terminals

CFL Membership Problem

Given:
A language $L(G)$ described by a Context-free Grammar (CFG) G in Chomsky Normal Form (CNF)

e.g. $S \rightarrow AB$
$A \rightarrow CC \mid a \mid c$
$B \rightarrow BC \mid b$
$C \rightarrow CB \mid BA \mid c$

Question:
Is string w element of the language $L(G)$?

e.g. w=abbc; Is $w \in L(G)$?

The CYK Algorithm

- algorithm from 1960s named after creators John Cocke, Daniel Younger and Tadao Kasami
- solves the CFL (given in CNF) membership problem in cubic time
- classic algorithm that utilises Dynamic Programming

Cocke

Kasami

CYK Algorithm - Idea

- **Observation:**
  the analysis of longer substrings is based on parsing shorter substrings

- **Example:**
  aab may be decomposed as a ab or aa b

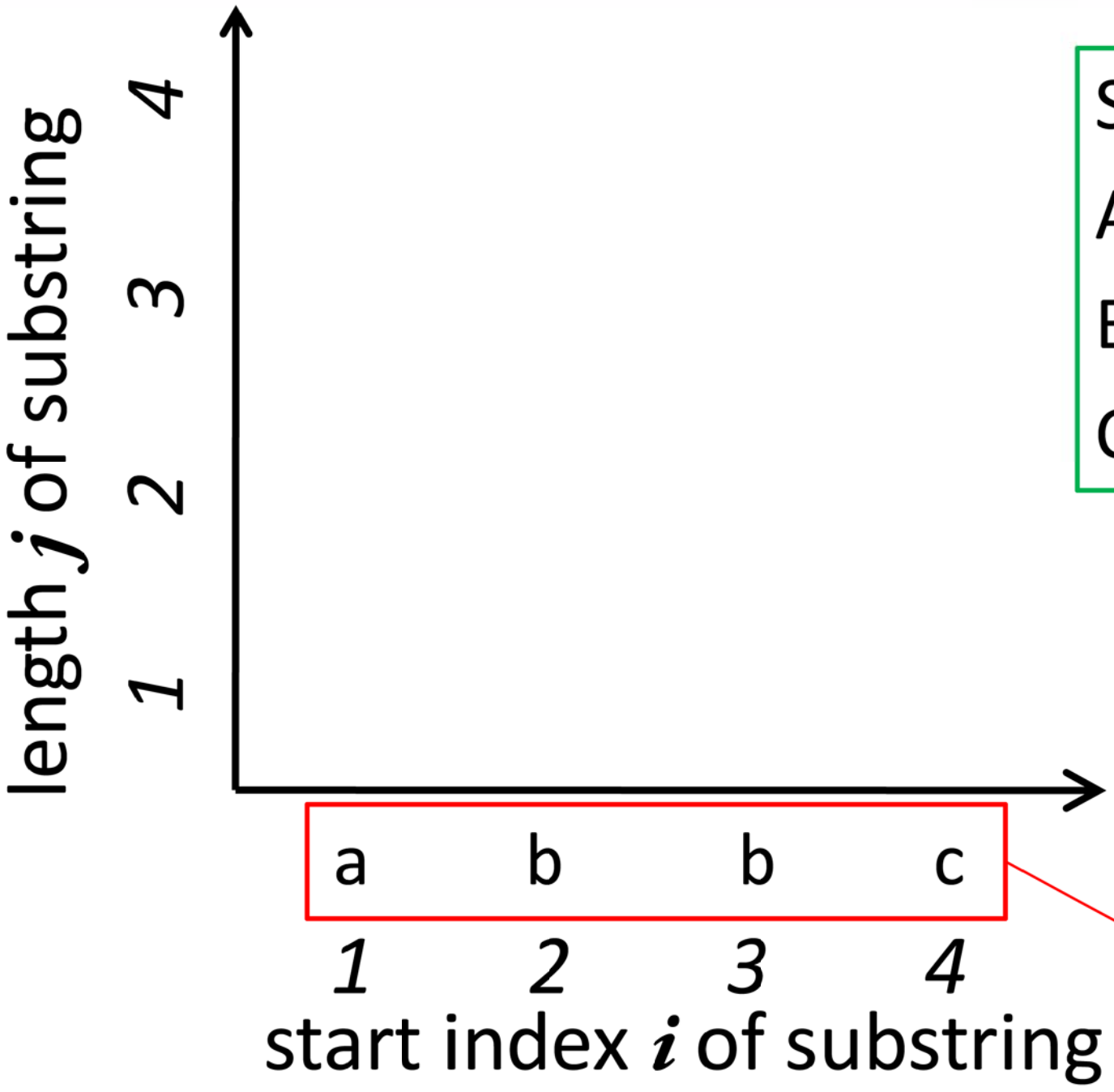
→ If we understand how to parse a and ab (and aa and b) then we can infer how to parse aab

CYK Algorithm - Approach

- **Approach:**
  we analyse the string in a bottom-up fashion
  in order of ascending substring complexity

Length-1 substrings	Length-2 substrings	Length-3 substrings	Length-4 substrings
abbc	abbc	abbc	abbc
abbc	abbc	abbc	
abbc	abbc		
abbc			

CYK Example



S → AB

A → CC | a | c

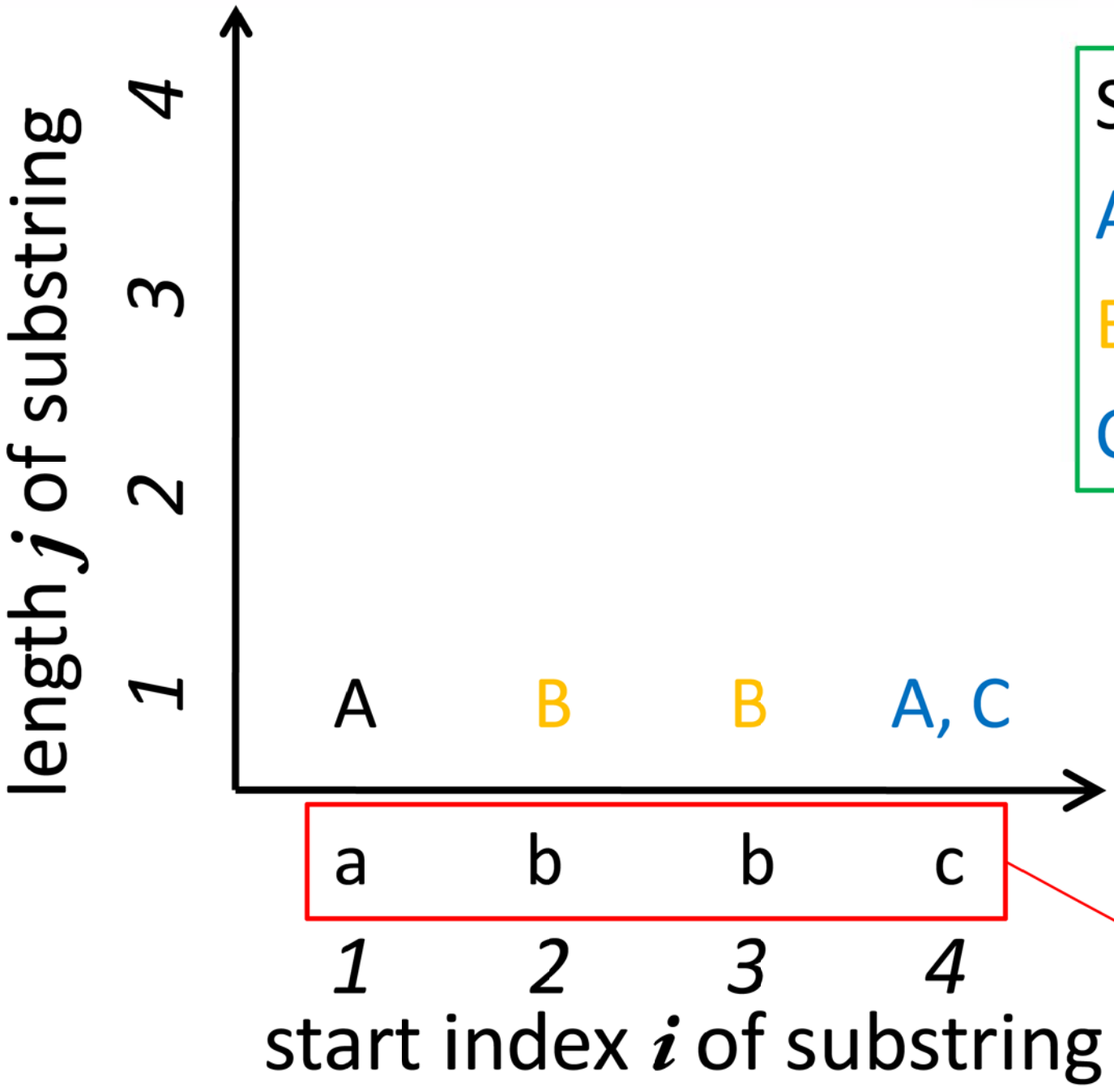
B → BC | b

C → CB | BA | c

language definition

string to test

CYK Example



S → AB

A → CC | a | c

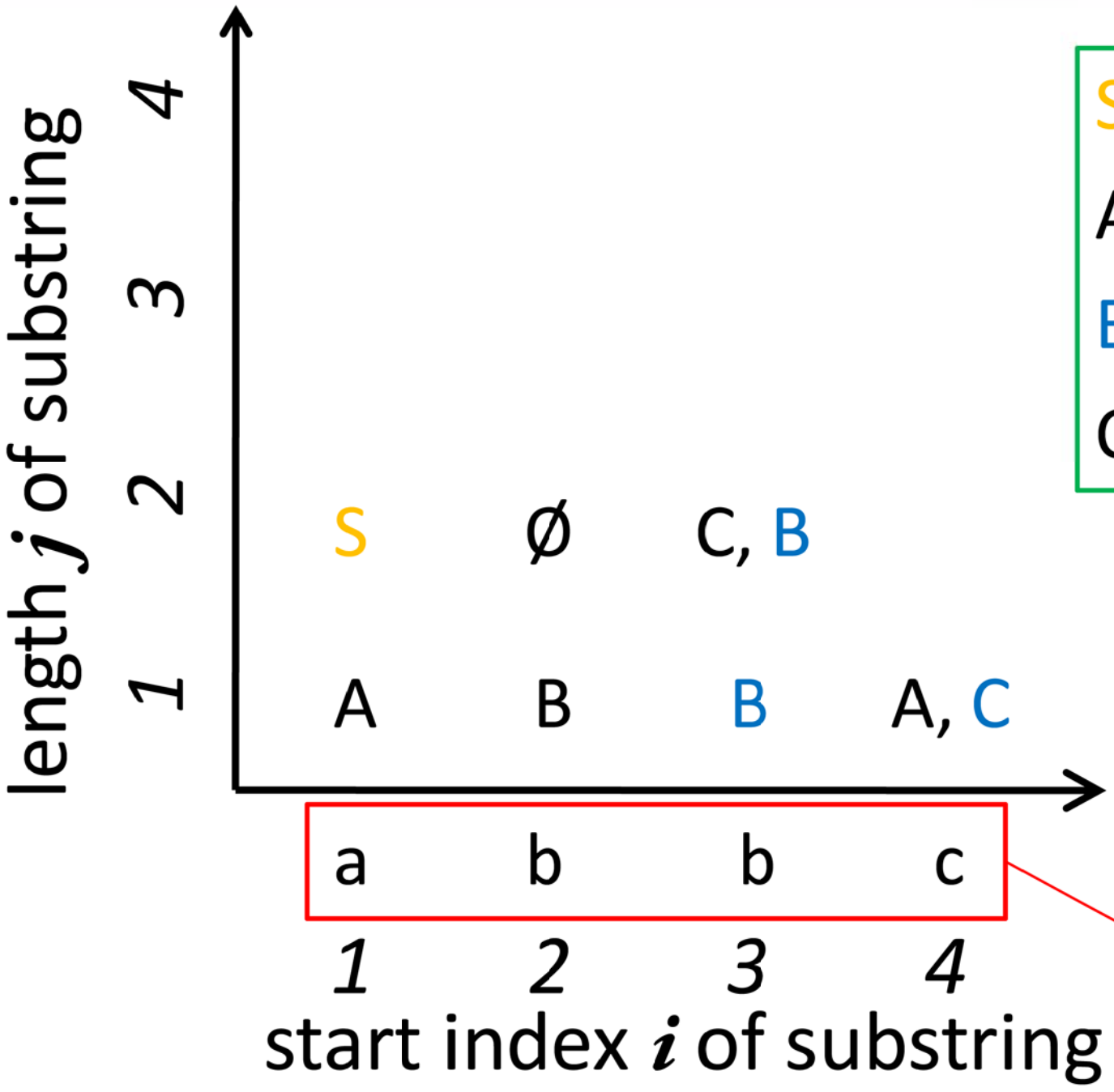
B → BC | b

C → CB | BA | c

language definition

string to test

CYK Example



S → AB

A → CC | a | c

B → BC | b

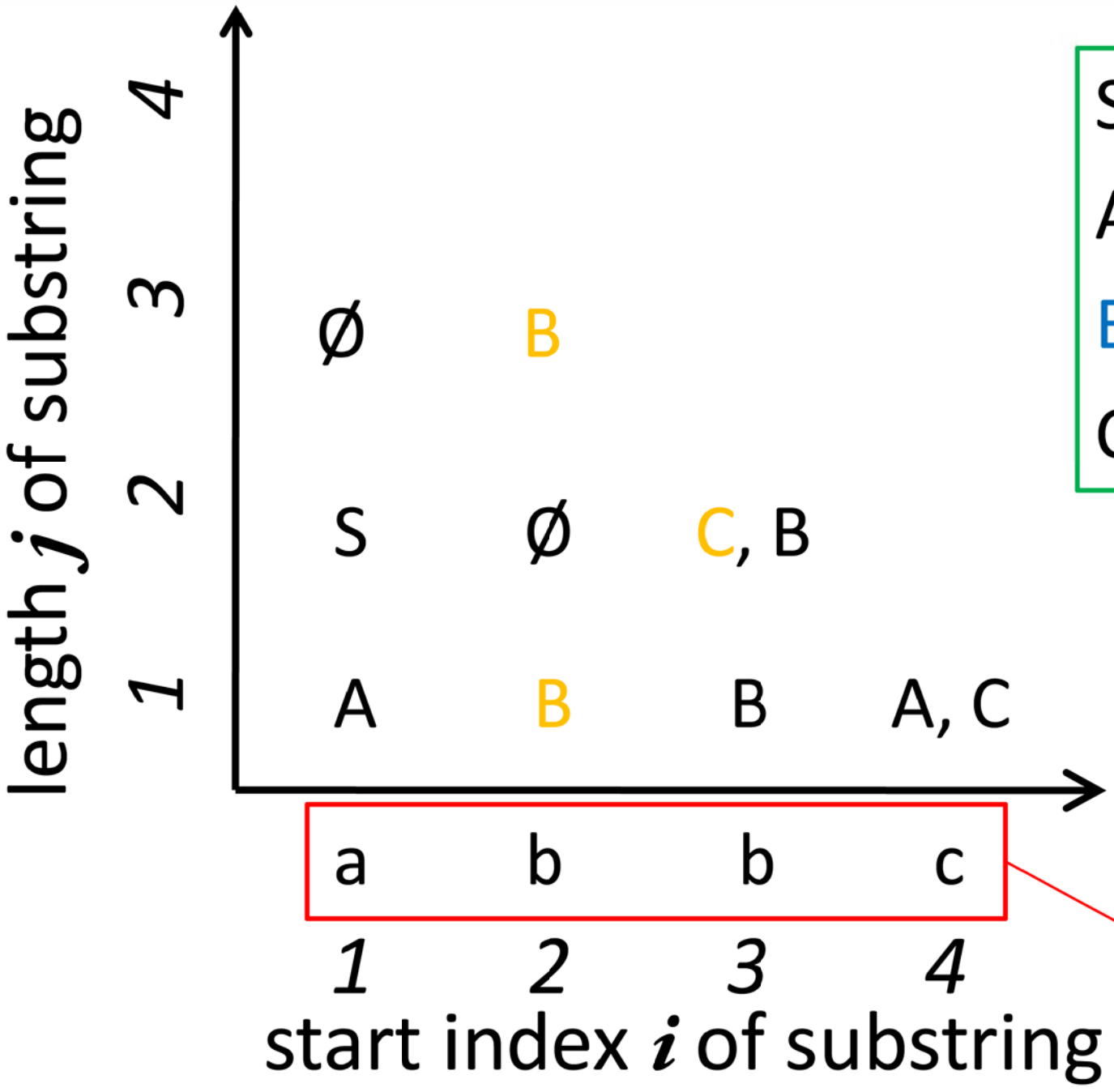
C → CB | BA | c

language definition

string to test

CYK Example

University of BRISTOL | Department of Computer Science



S → AB

A → CC | a | c

B → BC | b

C → CB | BA | c

language definition

string to test

CYK Example

University of BRISTOL | Department of Computer Science

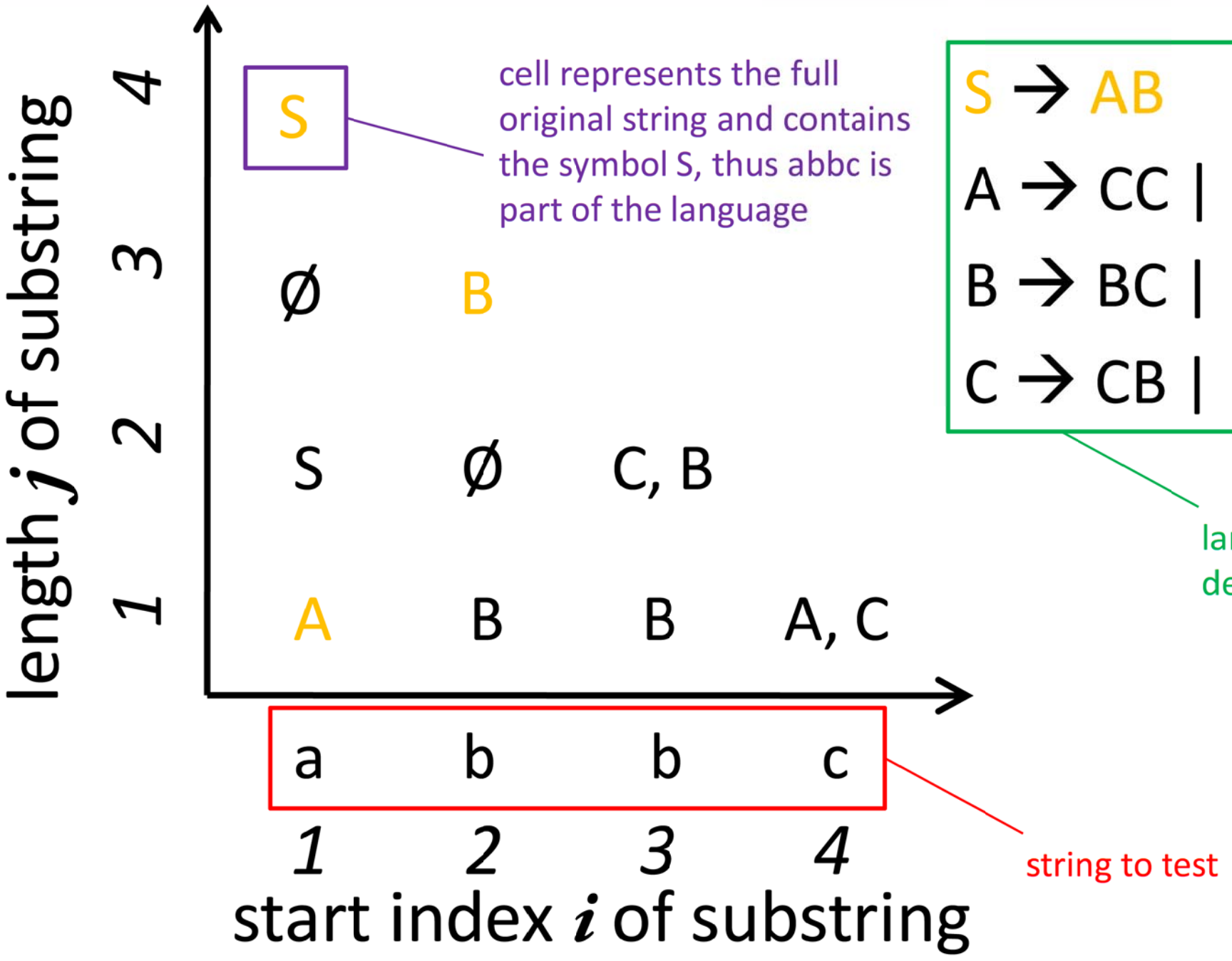
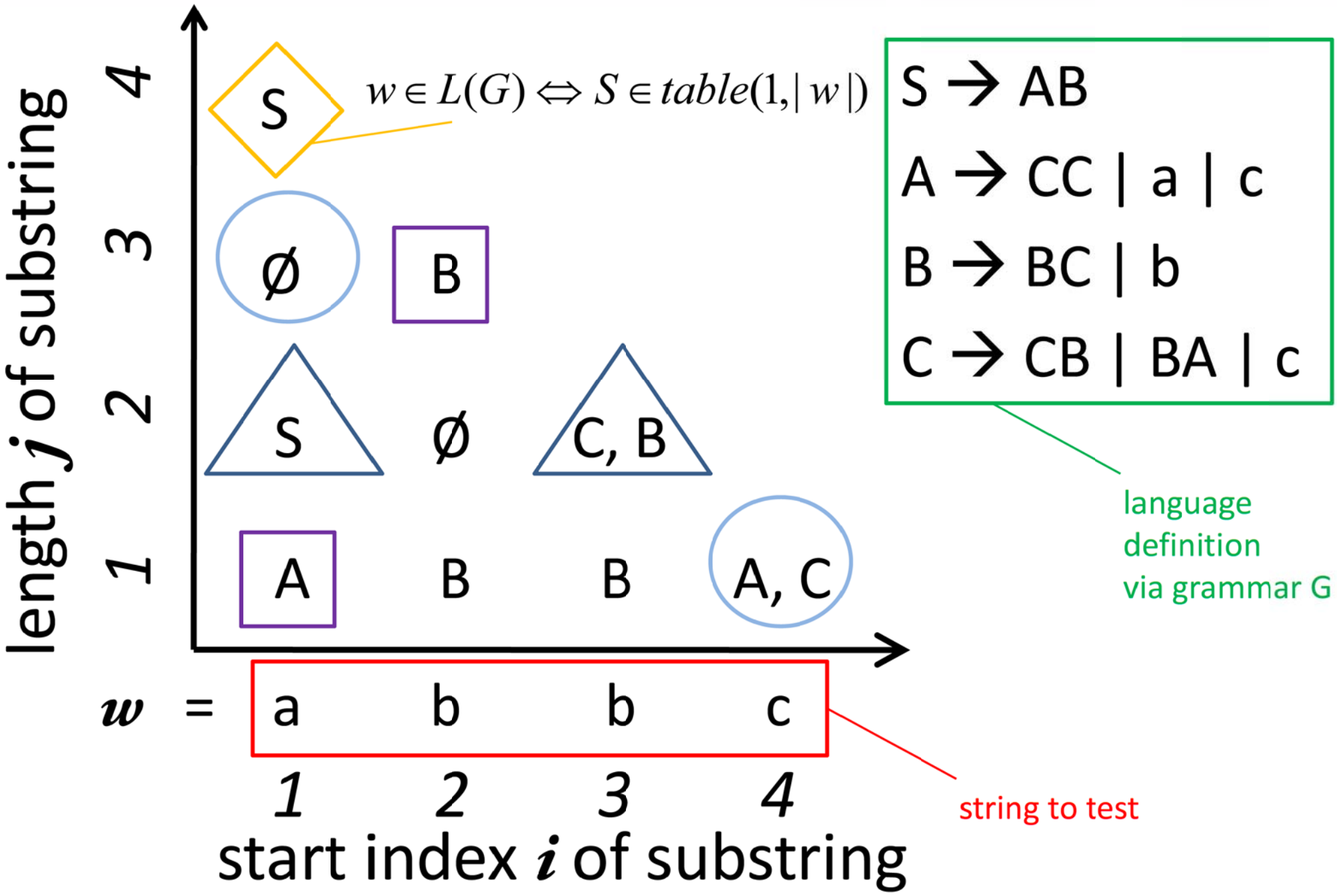


Table Evaluation Scheme

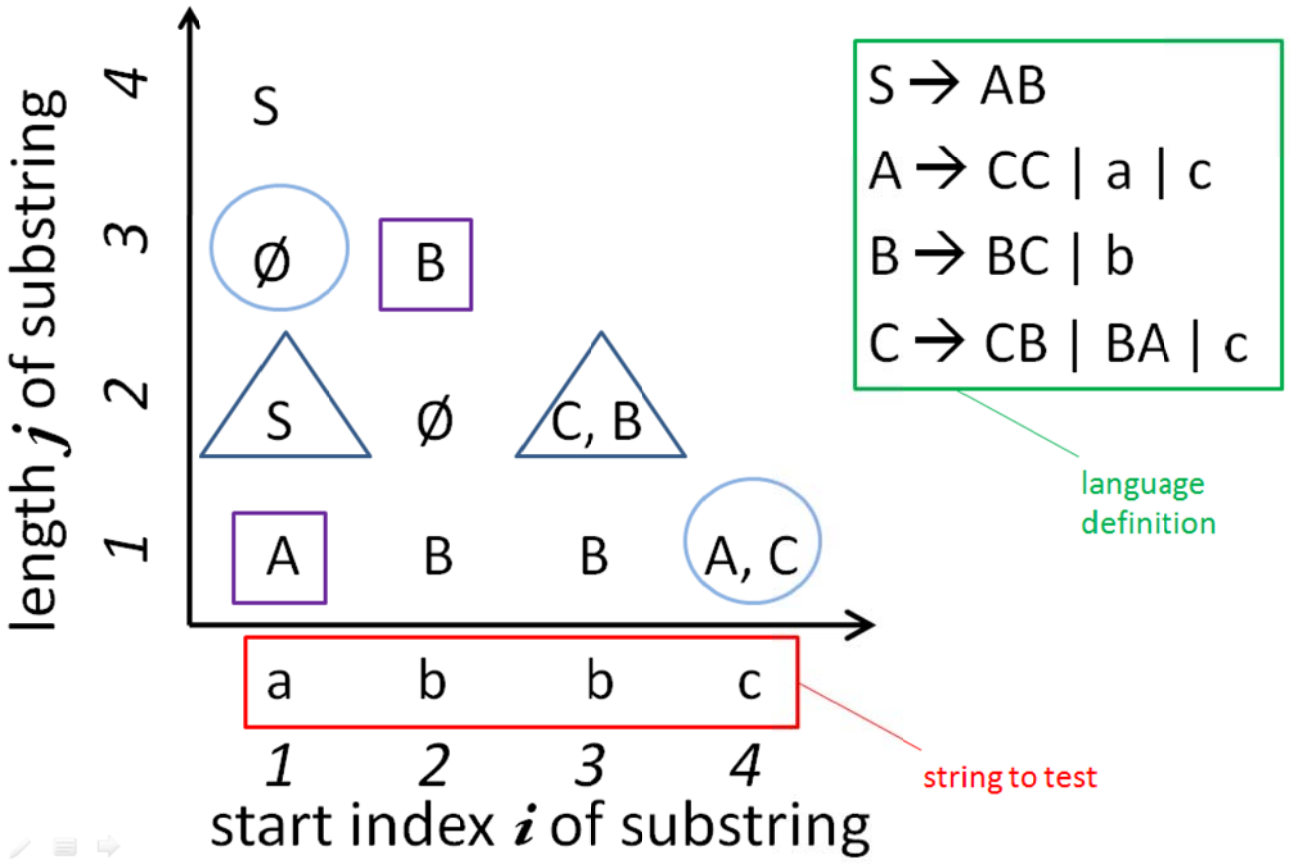
University of BRISTOL | Department of Computer Science



Recurrence Relation

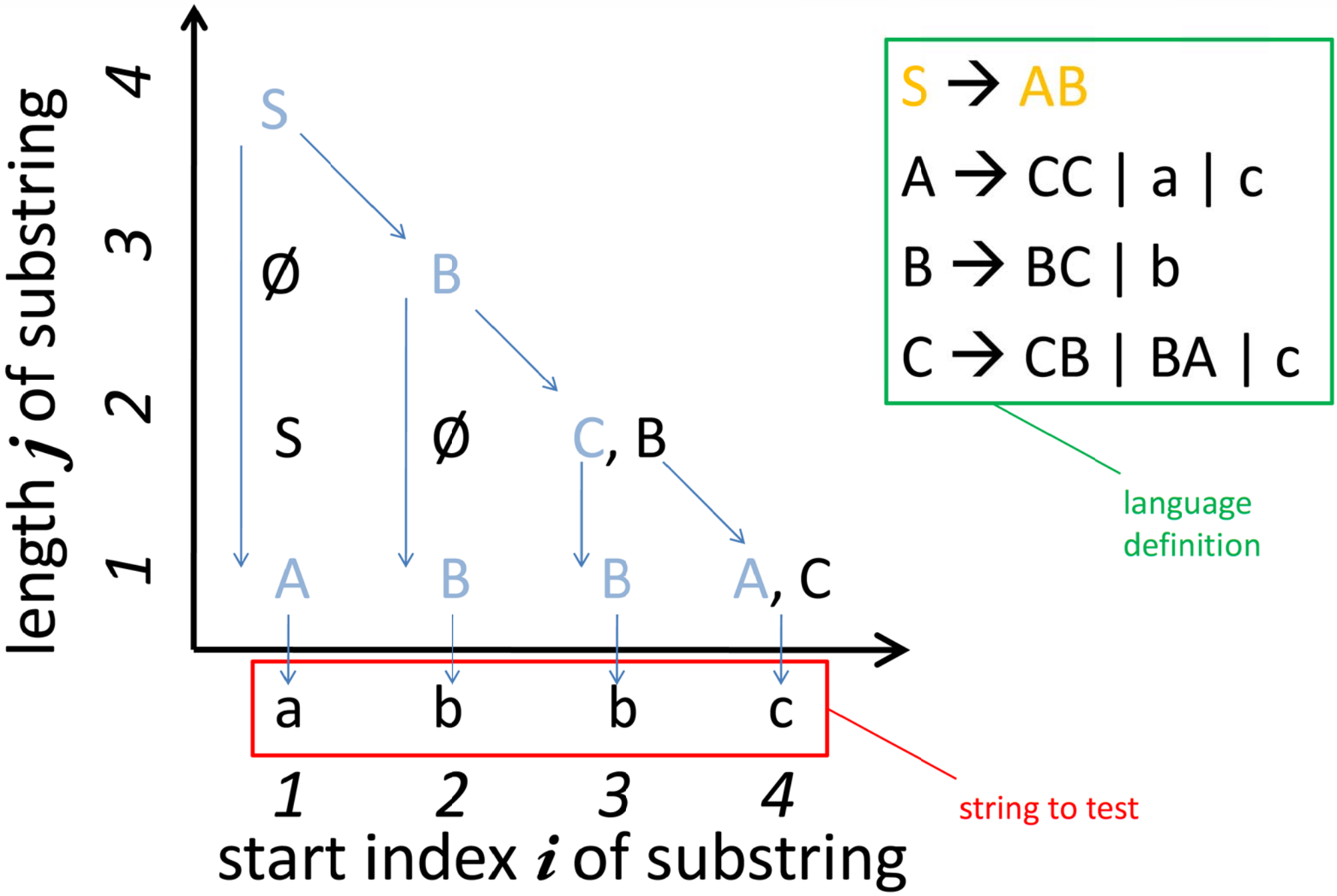
$$table(i,1) = \{R \mid \exists_{(R \to w[i]) \in P}\}$$

$$table(i,j) = \bigcup_{x=1}^{x=j-1} \{R \mid \exists_{(R \to AB) \in P} : A \in table(i,x) \wedge B \in table(i+x, j-x)\}$$



Reading the Parse Tree

University of BRISTOL | Department of Computer Science



Algorithm Design Paradigms

DIVIDE & CONQUER

Break down a problem into **independent** sub-problems of related type, solve them separately and combine the solutions.

GREEDY APPROACH

Use a sequence of **locally optimal** decisions incrementally to build up a solution.

DYNAMIC PROGRAMMING

Break down a problem into **overlapping** sub-problems of related type, build up solutions from larger and larger sub-solutions.