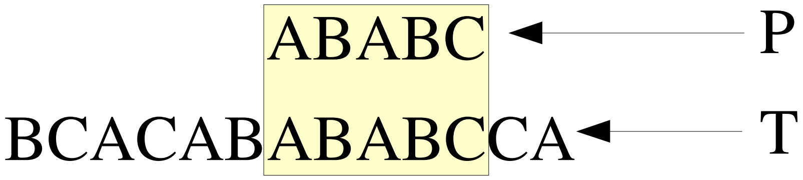
String Matching

(also called string searching)

The Problem

- Given a text T and a pattern P, find an occurrence of P inside T or return *no match*.
- T is of size t, P is of size p.
- Example:

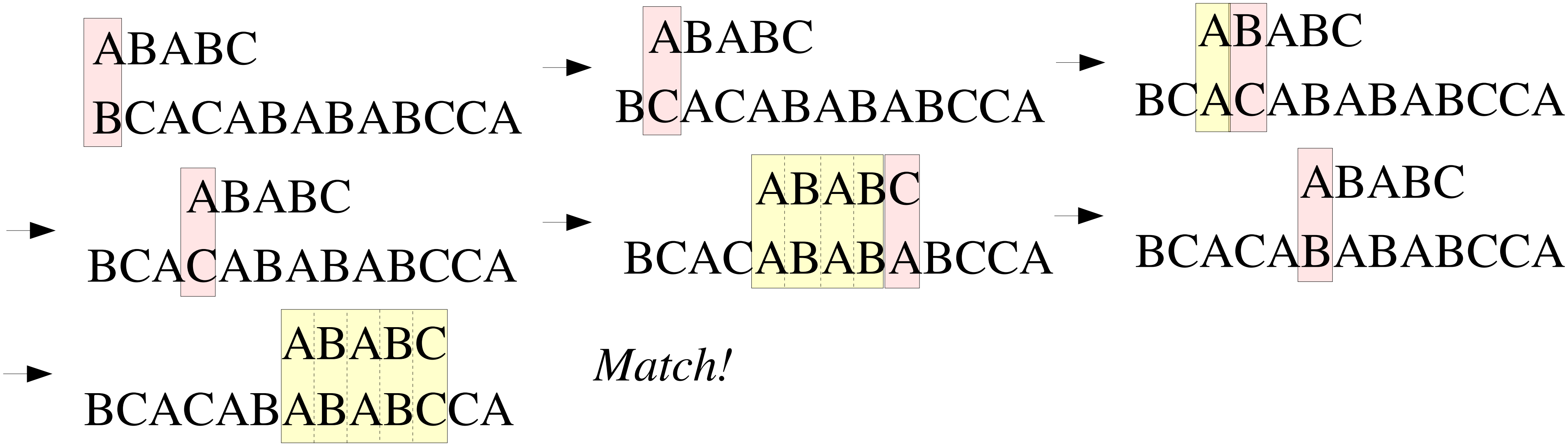


ABABC ← P

BCACABABABCCA ← T

Straight-Forward Solution

- Compare *P* to *T* starting at position 1
  - if mismatch, move *P* to the right and try again
  - if match, return current position
- Worst case: *(t-p+1)*p* comparisons, that is *O((t-p)*p)* and if *p=o(t)* we have *O(t*p)*.



ABABC
BCACABABABCCA

ABABC
BCACABABABCCA

ABABC
BCACABABABCCA

ABABC
BCACABABABCCA

ABABC
BCACABABABCCA

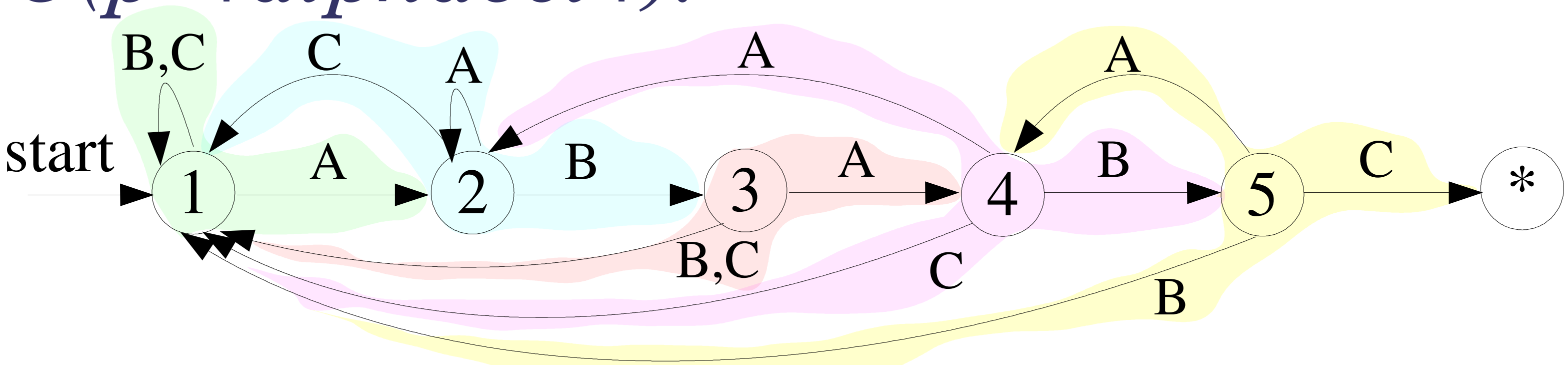
ABABC
BCACABABABCCA

ABABC
BCACABABABCCA

Match!

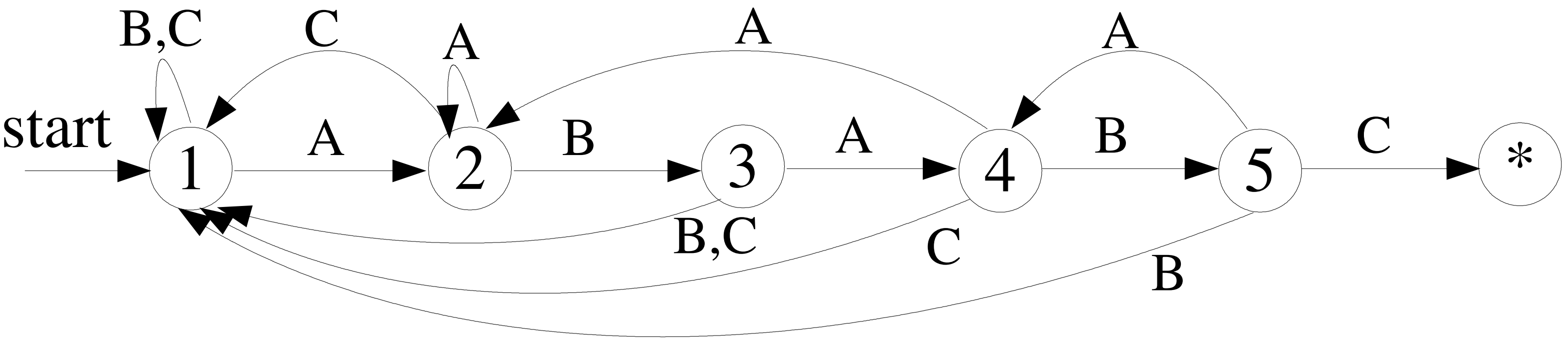
With Finite Automata

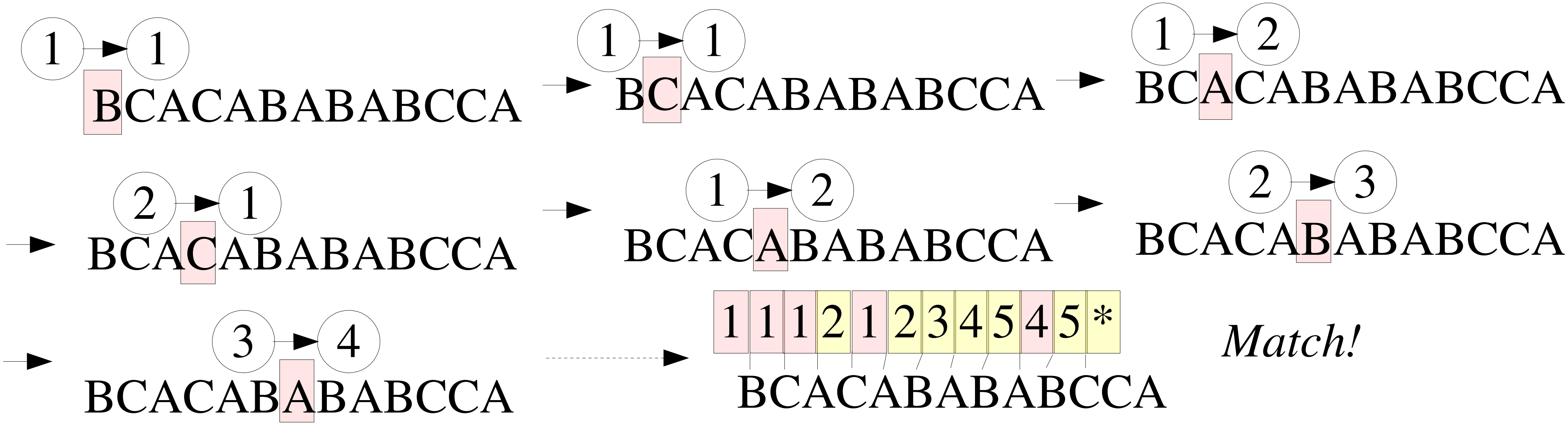
- Given *P*, it is possible to construct a finite automaton that is used to scan *T* in *O(t)*.
- Idea is to remember the last matched sub-string and to reuse the information.
- Match = reach *, no match = get stuck.
- Construction of the automaton: *O(p* | alphabet |)*.



With Finite Automata

- Algorithm: scan *T* and take transitions in the automaton. Success if reach *, failure if stuck.

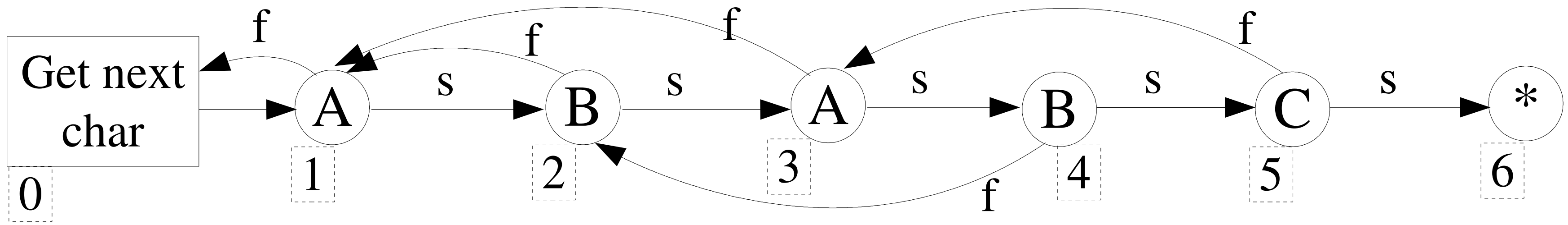




Match!

Knuth-Morris-Pratt Flowchart

- Given *P*, it is possible to construct a finite flowchart used to scan *T* in *O(t+p)*.
- Idea is to remember the maximum of matchable characters before the i^{th} position.
- Match = reach *, no match = get stuck.
- Construction of the flowchart: *O(p^2)*.

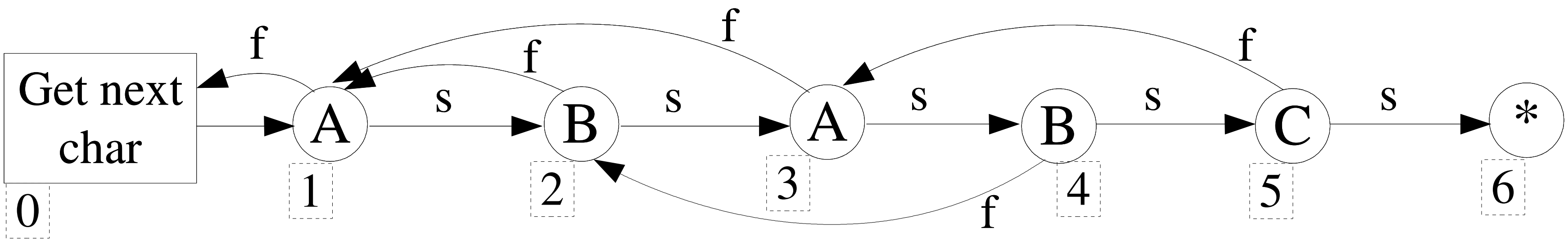


Next table:

A	B	A	B	C	-
0	1	1	2	3	

Knuth-Morris-Pratt Flowchart

- Algorithm: scan *T* and follow *P* according to the *next* table.



Next table:

A	B	A	B	C	-
0	1	1	2	3	

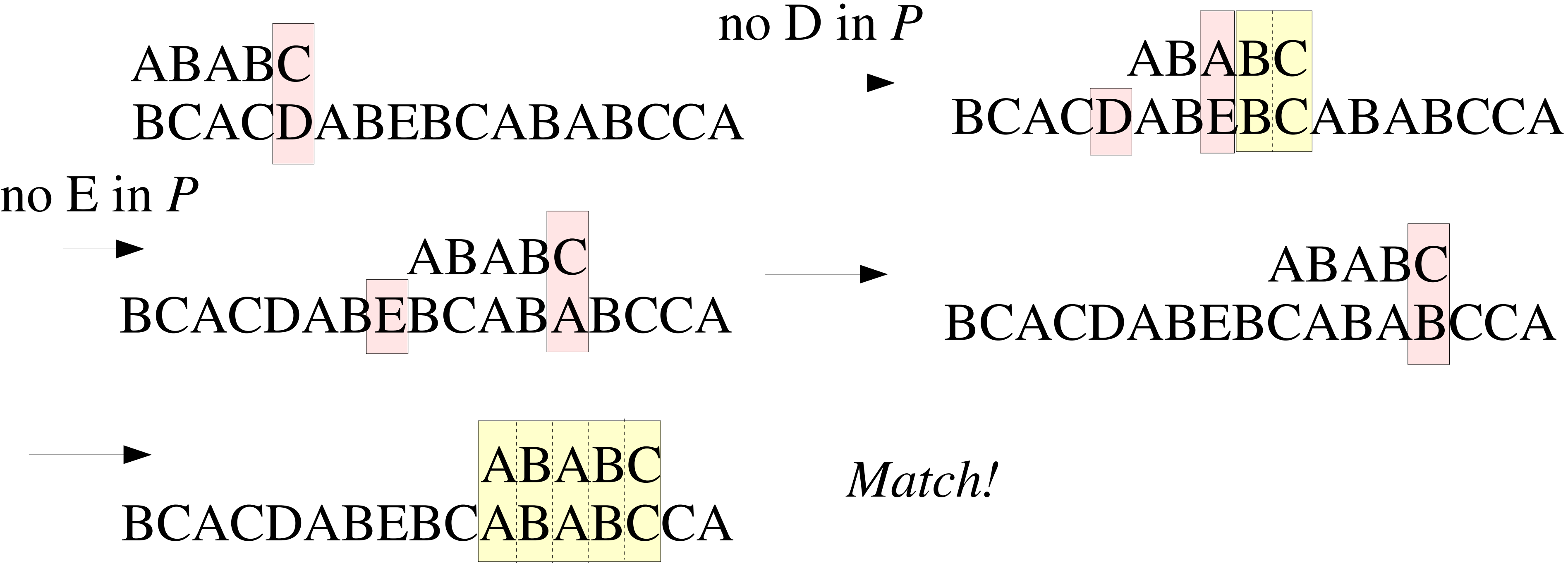
BCACABABABCCA

0	0	0	2	1	2	3	4	5	3	5	*
1	1	1		0					4		
				1							

Match!

Boyer-Moore Algorithm

- Idea is to skip text without checking it. Scan from right to left, use heuristics to decide how far to jump.
- Average running time *O(t / p)*, worst *O(t*p)*.



ABABC
BCACDABEBCABABCCA

no D in *P*

ABABC
BCACDABEBCABABCCA

no E in *P*

ABABC
BCACDABEBCABABCCA

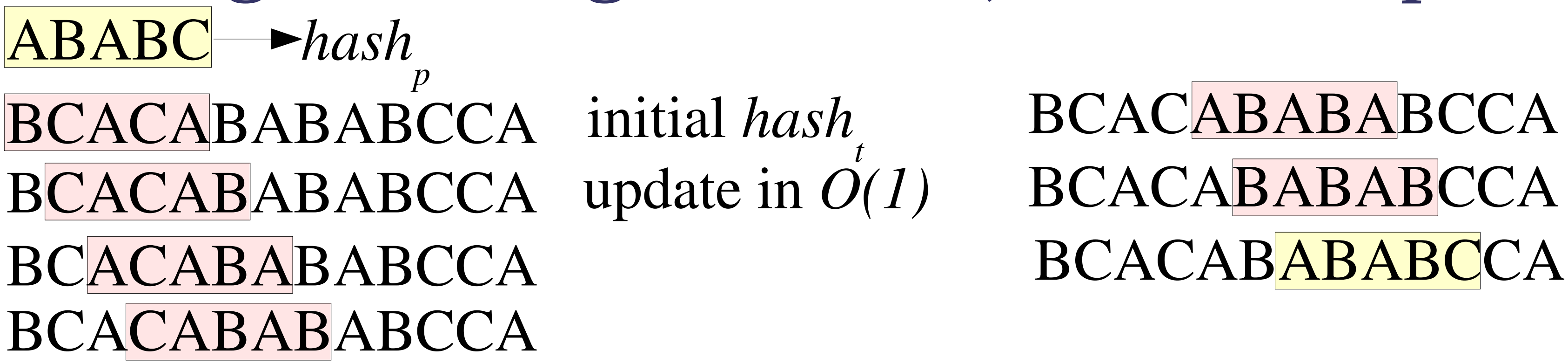
ABABC
BCACDABEBCABABCCA

ABABC
BCACDABEBCABABCCA

Match!

Rabin-Karp Algorithm

- Uses hash to identify equal strings! Very powerful for multi-pattern matching.
- *Trick*: use a *special* hash function. Treat the characters as number in some base, usually a “big” prime ⇒ compute next hash iteratively. Hopefully few collisions.
- Average running time *O(t)*, worst *O(t*p)*.



ABABC → $hash_p$

BCACABABABCCA

BCACABABABCCA

BCACABABABCCA

BCACABABABCCA

initial $hash_t$

update in $O(1)$

BCACABABABCCA

BCACABABABCCA

BCACABABABCCA

Rabin-Karp Algorithm

- Hash update: “shift” in the corresponding base.
- Also practical to use base 256 for characters (=1 byte) and a prime as the hash table size. Worse hash function, more collisions, but very fast to compute and performs well (when using xor).
- Useful for one of the *fun challenges*.