COMP90038
Algorithms and Complexity

Lecture 16: Time/Space Tradeoffs – String Search Revisited

(with thanks to Harald Søndergaard & Michael Kirley)

Andres Munoz-Acosta

munoz.m@unimelb.edu.au

Peter Hall Building G.83

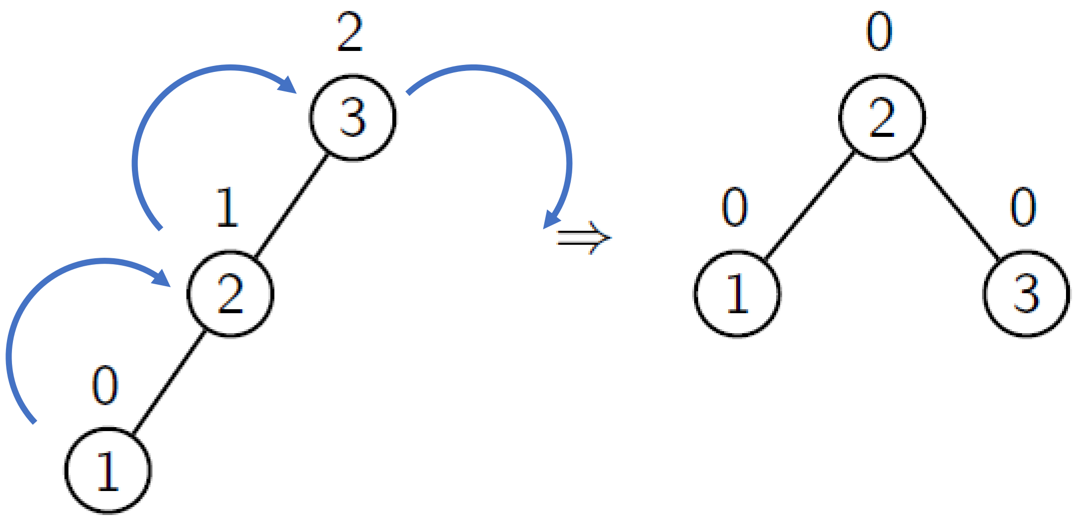
Recap

- BST have optimal performance when they are balanced.

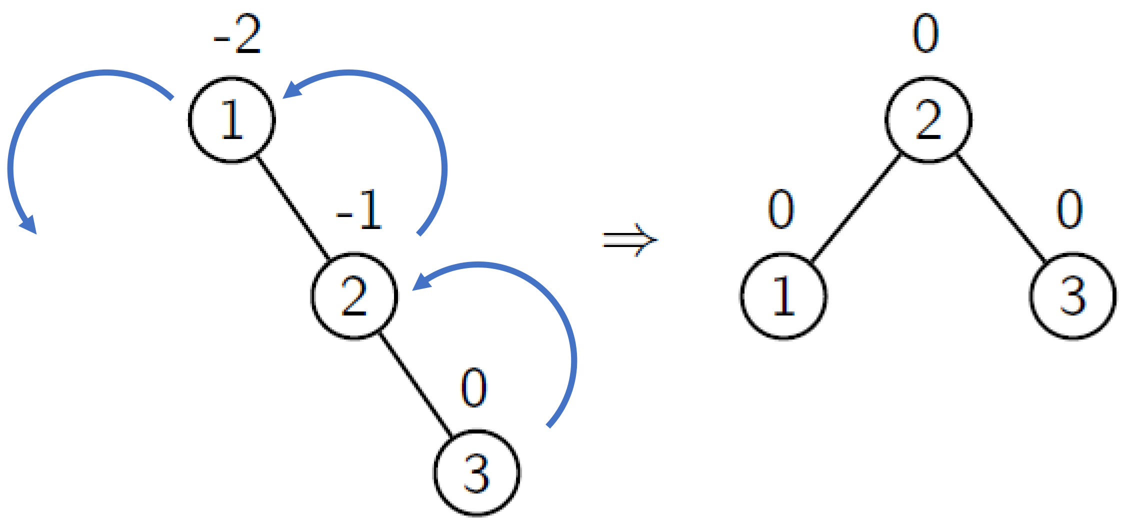
- AVL Trees:
  - Self-balancing trees for which the balance factor is -1, 0, or 1, for every sub-tree.
  - Rebalancing is achieved through rotations.
  - It guarantees depth of a tree with n nodes to be $\Theta(\log n)$

- 2–3 trees:
  - Trees that allow more than one item to be stored in a tree node.
  - This allows for a simple way of keeping search trees perfectly balanced.
  - Insertions, splits and promotions are used to grow and balance the tree.

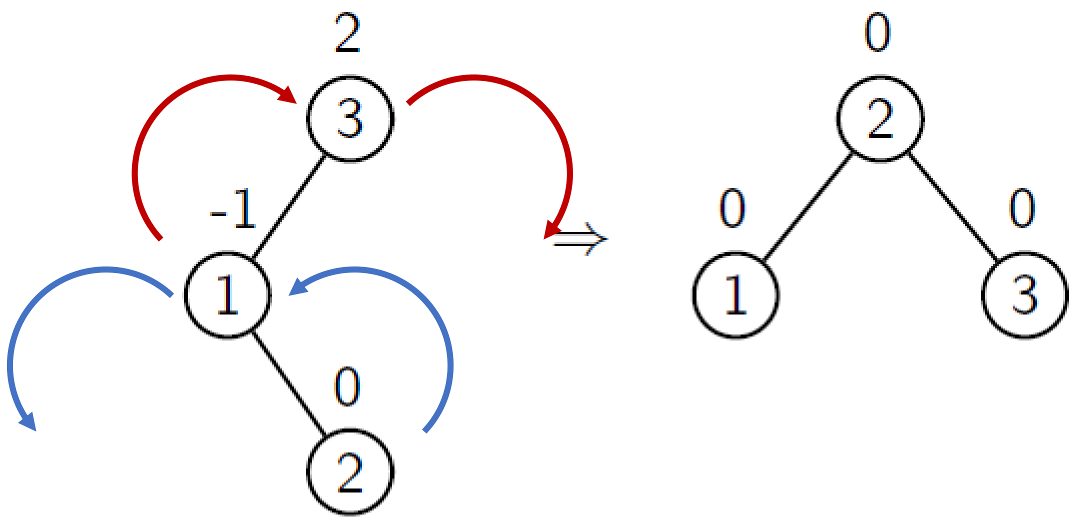
AVL Trees: R-Rotation



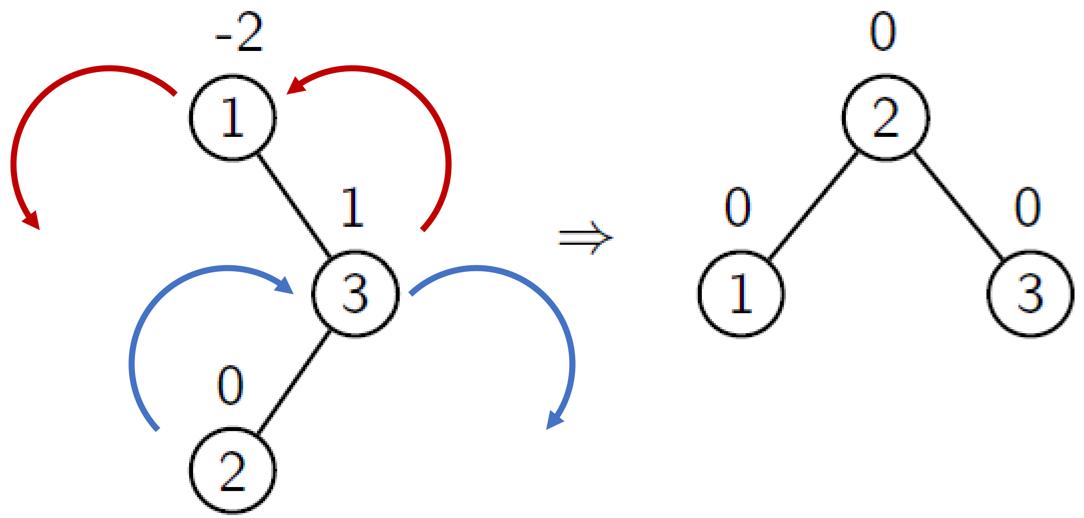
AVL Trees: L-Rotation



AVL Trees: LR-Rotation

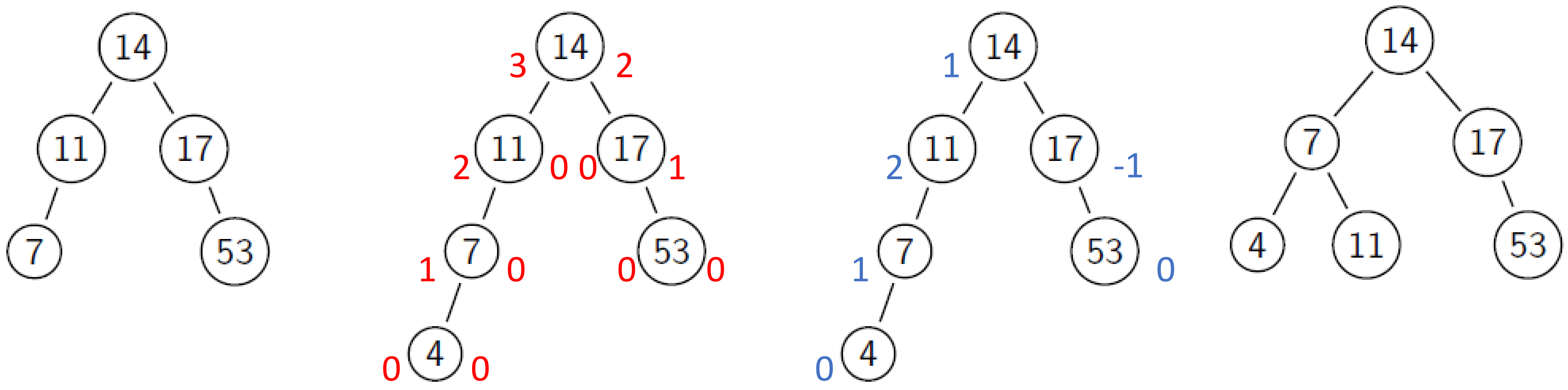


AVL Trees: RL-Rotation



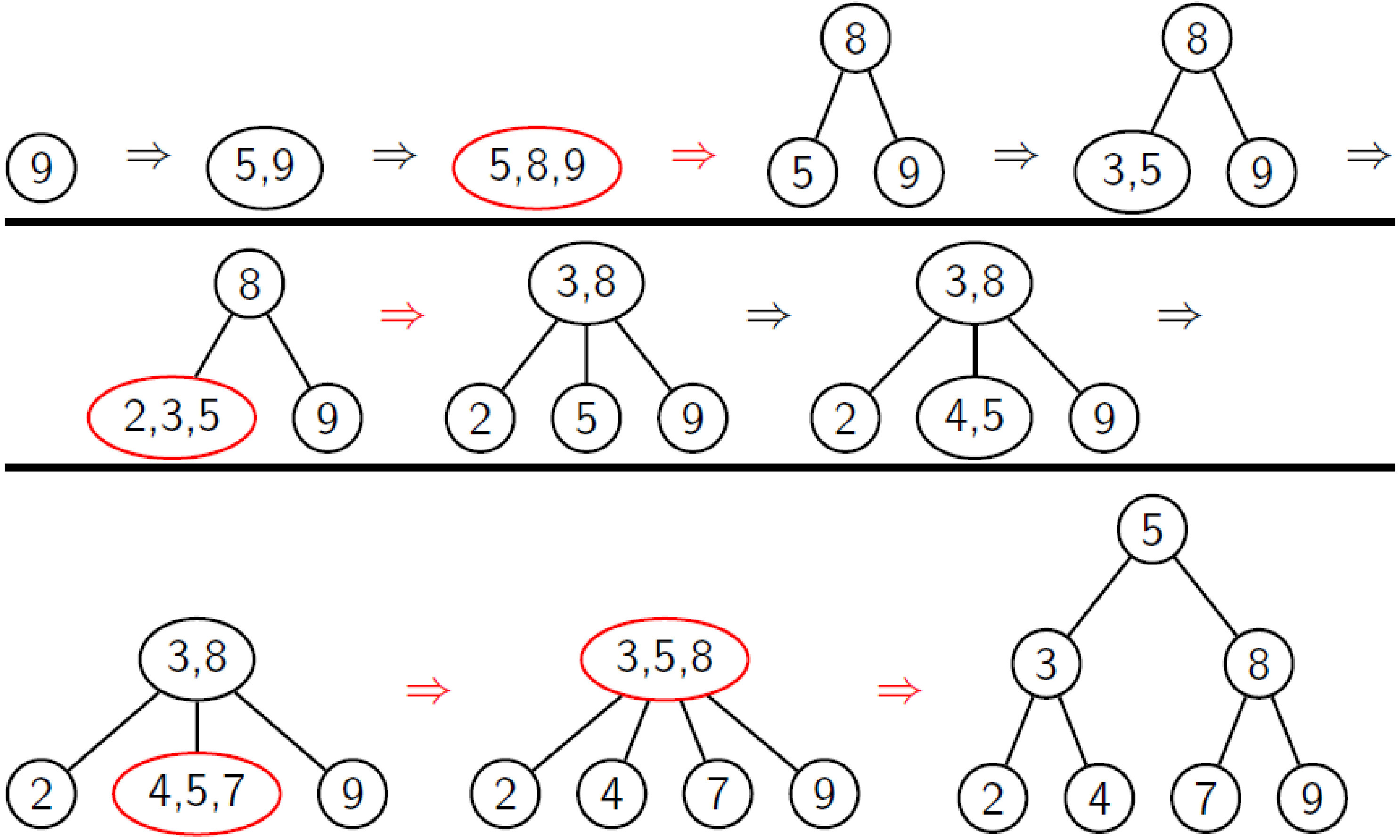
Example

- On the tree below, insert the elements {4, 13, 12}



- https://www.cs.usfca.edu/~galles/visualization/AVLtree.html

Example: Build a 2–3 Tree from {9, 5, 8, 3, 2, 4, 7}



2–3 Tree Analysis

- Worst case search time results when all nodes are 2-nodes. The relation between the number n of nodes and the height h is:

$$n = 1 + 2 + 4 + \ldots + 2^h = 2^{h+1} - 1$$

- That is, $\log_2(n+1) = h+1$.

- In the best case, all nodes are 3-nodes:

$$n = 2 + 2\times 3 + 2\times 3^2 + \ldots + 2\times 3^h = 3^{h+1} - 1$$

- That is, $\log_3(n+1) = h+1$.

- Hence we have $\log_3(n+1) - 1 \le h \le \log_2(n+1) - 1$.

- Useful formula: $\displaystyle\sum_{i=0}^{n} a^i = \frac{a^{n+1}-1}{a-1} \text{ for } a \neq 1$

Spending Space to Save Time

- Often we can find ways of decreasing the time required to solve a problem, by using additional memory in a clever way.

- For example, in **Lecture 6 (Recursion)** we considered the simple recursive way of finding the *n*-th Fibonacci number and discovered that the algorithm uses exponential time.

```
function FIB(n)
    if n = 0 then
        return 1
    if n = 1 then
        return 1
    return FIB(n − 1) + FIB(n − 2)
```

Spending Space to Save Time

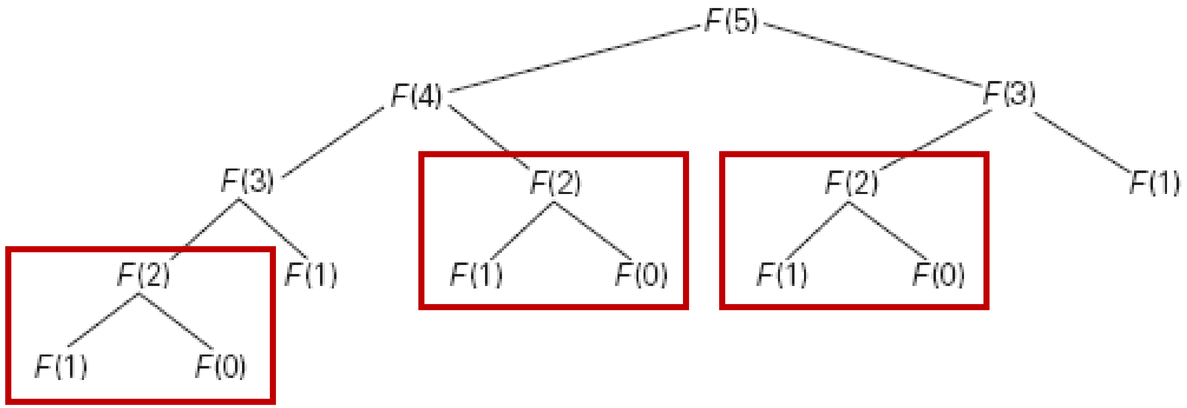


FIGURE 2.6 Tree of recursive calls for computing the 5th Fibonacci number by the definition-based algorithm.

Spending Space to Save Time

- However, suppose the same algorithm uses a table to **tabulate** the function $\text{F}\small{\text{IB}}()$ as we go: As soon as an intermediate result $\text{F}\small{\text{IB}}(i)$ has been found, it is not simply returned to the caller; the value is first placed in slot i of a table (an array). Each call to $\text{F}\small{\text{IB}}()$ first looks in this table to see if the required value is there, and only if it is not, the usual recursive process kicks in.

Fibonacci Numbers with Tabulation

- We assume that, from the outset, all entries of the table *F* are 0.

$$
\begin{array}{l}
\textbf{function } \textsc{Fib}(n) \\
\quad \textbf{if } n = 0 \textbf{ or } n = 1 \textbf{ then} \\
\quad\quad \textbf{return } 1 \\
\quad \mathit{result} \leftarrow F[n] \\
\quad \textbf{if } \mathit{result} = 0 \textbf{ then} \\
\quad\quad \mathit{result} \leftarrow \textsc{Fib}(n-1) + \textsc{Fib}(n-2) \\
\quad\quad F[n] \leftarrow \mathit{result} \\
\quad \textbf{return } \mathit{result}
\end{array}
$$

- (I show this code just so that you can see the principle; in **Lecture 6** we already discovered a different linear-time algorithm, so here we don't really need tabulation.)

Sorting by Counting

- Suppose we need to sort large arrays, but we know that they will hold keys taken from a **small**, **fixed** set (so lots of duplicate keys).

- For example, suppose all keys are single digits:

$$6\ \boxed{3}\ \boxed{3}\ 8\ 1\ 0\ 8\ 7\ 9\ 2\ 5\ \boxed{3}\ 5\ \boxed{3}\ 1\ 8\ 7\ 6\ 5\ 1\ 2\ 1\ 5\ \boxed{3}$$

- Then we can, in a single linear scan, count the occurrences of each key in array *A* and store the result in a small table:

key	0	1	2	3	4	5	6	7	8	9
Occ	1	4	2	$\boxed{5}$	0	4	2	2	3	1

- Now use a second linear scan to make the counts **cumulative**:

key	0	1	2	3	4	5	6	7	8	9
Occ	1	5	7	12	12	16	18	20	23	24

-

Sorting by Counting

- We can now create a sorted array *S*[1]...*S*[*n*] of the items by simply slotting items into pre-determined slots in *S* (a third linear scan).

6 3 3 8 1 0 8 7 9 2 5 3 5 3 1 8 7 6 5 1 2 1 5 3

key	0	1	2	3	4	5	6	7	8	9
Occ	1	5	7	12	12	16	18	20	23	24

- Place the last record (with key 3) in *S*[12] and decrement *Occ*[3] (so that the next `3' will go into slot 11), and so on.

for $i \leftarrow n$ **to** 1 **do**
$\quad S[Occ[A[i]]] \leftarrow A[i]$
$\quad Occ[A[i]] \leftarrow Occ[A[i]] - 1$

-

Sorting by Counting

- Note that this gives us a **linear-time** sorting algorithm (for the cost of some extra space).

- However, it only works in situations where we have a small range of keys, known in advance.

- The method never performs a key-to-key comparison.

- The time complexity of **key-comparison based sorting** has been proven to be in $\Omega(n \log n)$.

String Matching Revisited

- In **Lecture 5 (Brute Force Methods)** we studied an approach to string search.

for $i \leftarrow 0$ to $n - m$ **do**
    $j \leftarrow 0$
    while $j < m$ and $p[j] = t[i + j]$ **do**
        $j \leftarrow j + 1$
    if $j = m$ **then**
        return i
return -1

String Matching Revisited

N	O	B	O	D	Y	_	N	O	T	I	C	E	D	_	H	I	M
N	**O**	**T**															
	N	O	T														
		N	O	T													
			N	O	T												
				N	O	T											
					N	O	T										
						N	O	T									
							N	**O**	**T**								

FIGURE 3.3 Example of brute-force string matching. The pattern's characters that are compared with their text counterparts are in bold type.

String Matching Revisited

- “Strings” are usually built from a small, pre-determined alphabet.
- Most of the better algorithms rely on some pre-processing of strings before the actual matching process starts.
- The pre-processing involves the construction of a small table (of predictable size).
- Levitin refers to this as “input enhancement”.

Horspool's String Search Algorithm

- Comparing from right to left in the pattern.

- Very good for random text strings.

S T R I N G S E A R C H E X A M P
E X A M

- We can do better than just observing a mismatch here.

- Because the pattern has **no occurrence of I**, we might as well slide it 4 positions along.

- This decision is based only on knowing the pattern.

Horspool's String Search Algorithm

S	T	R	I	N	G	S	E	A	R	C	H	E	X	A	M	P
E	X	A	M													
				E	X	A	M									

- Here we can slide the pattern 3 positions, because the last occurrence of E in the pattern is its first position.

S	T	R	I	N	G	S	E	A	R	C	H	E	X	A	M	P
E	X	A	M													
				E	X	A	M									
							E	X	A	M						
											E	X	A	M		
												E	X	A	M	

Horspool's String Search Algorithm

- What happens when we have longer partial matches?

S	E	A	R	C	H	I	N	G		
B	I	R	C	H						
	B	I	R	C	H					
						B	I	R	C	H

- The shift is determined by the last character in the pattern.

- Note that this is the same as the character in the text that we first matched against. Hence the skip is **always** determined by that character, whether it matched or not.

Char	Shift
A	5
B	4
C	1
⋮	⋮
H	5
I	3
⋮	⋮
R	2
S	5
⋮	⋮
Z	5

Horspool's String Search Algorithm

- Building (calculating) the shift table is easy.
- We assume indices start from 0.
- Let *alphasize* be the size of the alphabet.

```
function FindShifts(P[·], m)          ▷ Pattern P has length m
    for i ← 0 to alphasize − 1 do
        Shift[i] ← m
    for j ← 0 to m − 2 do
        Shift[P[j]] ← m − (j + 1)
```

Horspool's String Search Algorithm

function HORSPOOL($P[\cdot], m, T[\cdot], n$)
  FINDSHIFTS(P, m)
  $i \leftarrow m - 1$
  while $i < n$ **do**
    $k \leftarrow 0$
    while $k < m$ **and** $P[m - 1 - k] = T[i - k]$ **do**
      $k \leftarrow k + 1$
    if $k = m$ **then** ▷ We have a match
      return $i - m + 1$ ▷ Start of the match
    else
      $i \leftarrow i + \mathit{Shift}[T[i]]$ ▷ Slide the pattern along
  return -1

Horspool's String Search Algorithm

- We can also consider posting a sentinel: Append the pattern *P* to the end of the text *T* so that a match is guaranteed.

function $\text{HORSPOOL}(P[\cdot], m, T[\cdot], n)$
    $\text{FINDSHIFTS}(P, m)$
    $i \leftarrow m - 1$
    while True **do**
        $k \leftarrow 0$
        while $k < m$ **and** $P[m - 1 - k] = T[i - k]$ **do**
            $k \leftarrow k + 1$
        if $k = m$ **then**
            if $i \geq n$ **then**
                return -1
            else
                return $i - m + 1$
        $i \leftarrow i + Shift[T[i]]$

Horspool's String Search Algorithm

- Unfortunately the worst-case behaviour of Horspool's algorithm is still $O(m \times n)$, like the brute-force method.
- However, in practice, for example, when used on English texts, it is linear-time, and fast.

Other Important String Search Algorithms

- Horspool's algorithm was inspired by the famous **Boyer-Moore** algorithm (**BM**), also covered in Levitin's book. The BM algorithm is very similar, but it has a more sophisticated shifting strategy, which makes it $O(m+n)$.

- Another famous string search algorithm is the **Knuth-Morris-Pratt** algorithm (**KMP**), explained in the remainder of these slides. KMP is very good when the alphabet is small, say, we need to search through very long bit strings.

- Also, we shall soon meet the **Rabin-Karp** algorithm (**RK**), albeit briefly.

- While very interesting, **the BM, KMP, and RK algorithms are not examinable.**

Knuth-Morris-Pratt (Not Examinable)

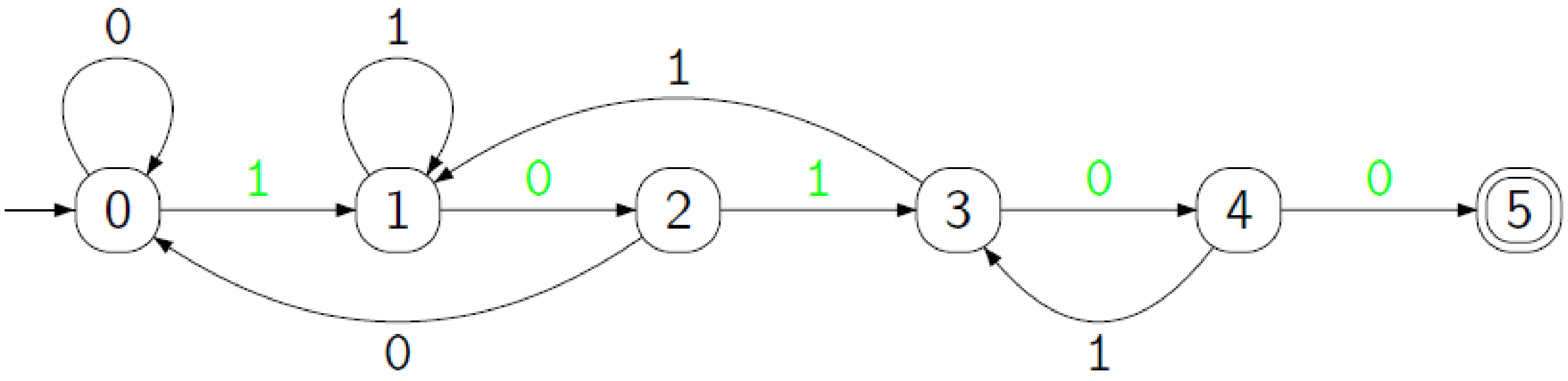
- Suppose we are searching in strings that are built from a small alphabet, such as the binary digits 0 and 1, or the nucleobases.
- Consider the brute-force approach for this example:

Text:	1 0 0 0 1 0 0 0 0
Pattern:	1 0 0 0 0

- Every “false start” contains a lot of information.
- Again, we hope to **pre-process** the pattern so as to find out when the brute-force method's index i can be incremented by more than 1.
- Unlike Horspool's method, KMP works by comparing from left to right in the pattern.

Knuth-Morris-Pratt as Running an FSA

- Given the pattern [1 0 1 0 0] we want to construct the following **finite-state automaton**:

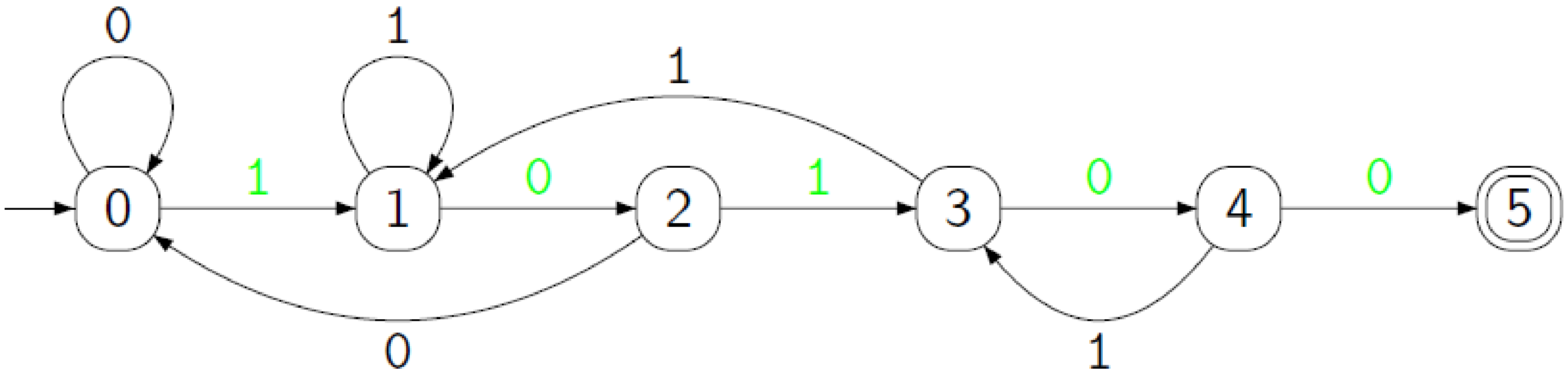


- We can capture the behaviour of this automaton in a table.

Knuth-Morris-Pratt Automaton

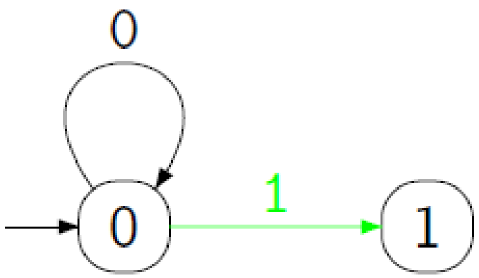
- We can represent the finite-state automaton as a 2-dimensional "transition" array *T*, where *T*[c][j] is the state to go to upon reading the character *c* in state *j*.

j	$T['0'][j]$	$T['1'][j]$
0	0	1
1	2	1
2	0	3
3	4	1
4	5	3

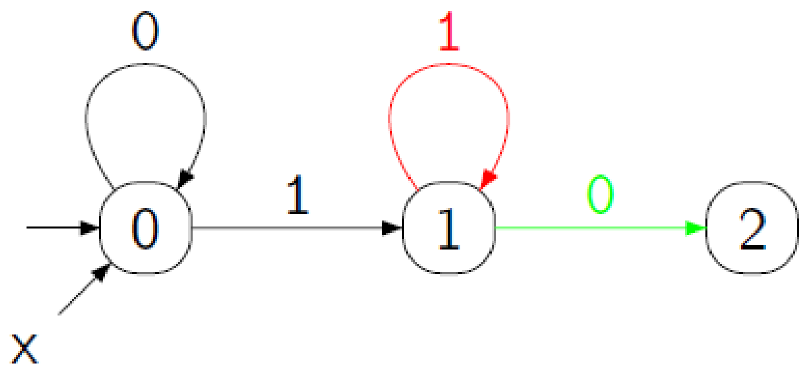


Constructing the Automaton

- The automaton (or the table *T*) can be constructed step-by-step:

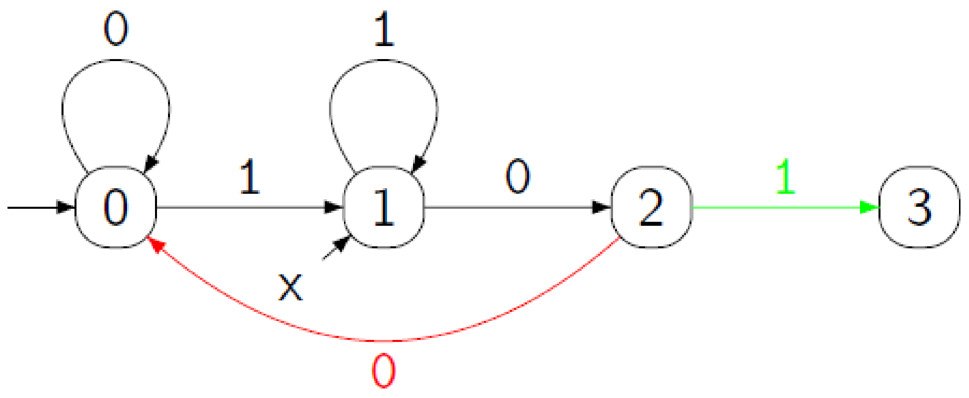


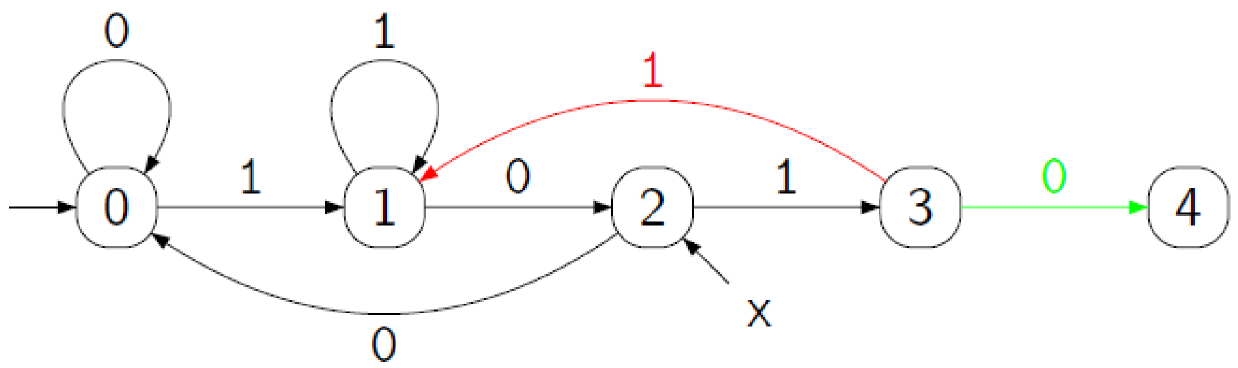
- Somewhat tricky but fast.
- *x* is a “backtrack point”.
- For next state *j*:
  - First *x*'s transitions are copied (in red).
  - Then the success arc is updated, determined by *P*[*j*] (in green).



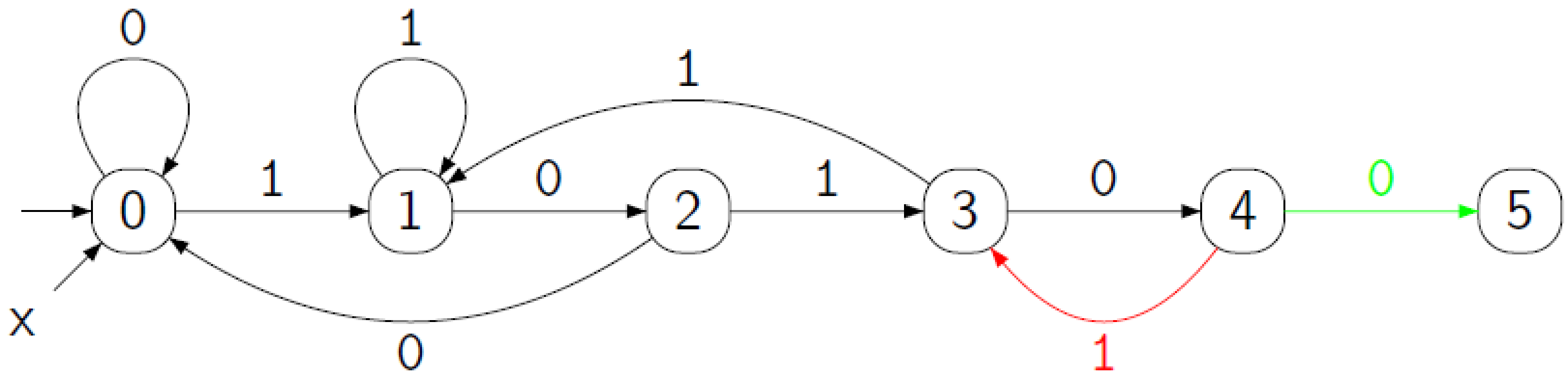
- Finally *x* is updated based on *P*[*j*].

Constructing the Automaton





Constructing the Automaton



Constructing the Automaton

$T['0'][0] \leftarrow 0$
$T['1'][0] \leftarrow 0$
$T[P[0]][0] \leftarrow 1$
$x \leftarrow 0$
$j \leftarrow 1$
while $j < m$ **do**
    $T['0'][j] \leftarrow T['0'][x]$
    $T['1'][j] \leftarrow T['1'][x]$
    $T[P[j]][j] \leftarrow j + 1$
    $x \leftarrow T[P[j]][x]$
    $j \leftarrow j + 1$

Pattern Compilation: Hard-Wiring the Pattern

- Even better, we can directly produce code that is specialised to find the given pattern. As a C program, for the example `p = 1 0 1 0 0`:

```
int kmp(char *s) {
        int i = -1;
        s0: i++; if (s[i] == '0') goto s0;
        s1: i++; if (s[i] == '1') goto s1;
        s2: i++; if (s[i] == '0') goto s0;
        s3: i++; if (s[i] == '1') goto s1;
        s4: i++; if (s[i] == '1') goto s3;
        s5: return i-4;
}
```

- Again, this assumes that we have posted a sentinel, that is, appended `p` to the end of `s` before running `kmp(s)`.

Next week

- We look at the hugely important technique of **hashing**, a standard way of implementing a "dictionary".

- Hashing is arguably the best example of how to gain speed by using additional space to great effect.