COMP90038
Algorithms and Complexity

Lecture 16: Input enhancement

(with thanks to Harald Søndergaard & Michael Kirley)

Andres Munoz-Acosta

munoz.m@unimelb.edu.au

Peter Hall Building G.83

On the previous lecture

- We talked about **balanced trees**, a group of ADTs that optimise search performance by maintaining the tree **"reasonably" balanced**
- We examined an instance simplification approach (**AVL trees**) and a representational change approach (**2–3 trees**)
  - Both of them guarantee a depth of $\Theta(\log n)$ for n nodes
- **AVL Trees** track the difference in height between subtrees through the **balance factor**, which must be at most ± 1
  - Rebalancing is achieved through **rotations**
- **2–3 trees** stores up to **two items** in each tree node
  - **Insertions, splits and promotions** are used to grow and balance the tree

Today's lecture

- We often find that we can **decrease the time** required to solve a problem, by using **additional memory space**

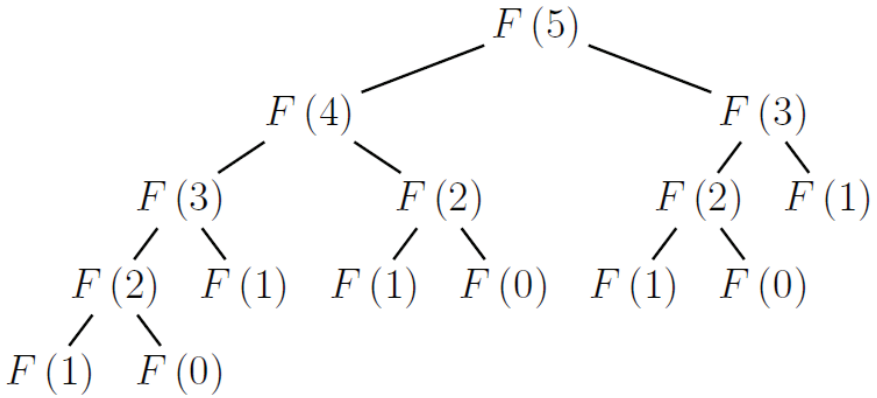
- **Input enhancement** is a technique that **pre-processes** the input, and stores **additional information** obtained in the form of a **table**

- Today we will talk about **three algorithms** that use **input enhancement** to solve a task
  - Distribution counting
  - Horspool's string matching algorithm
  - Knuth-Morris-Prat string matching algorithm

Input enhancement

- In **Lecture 6**, you discussed a simple recursive algorithm or finding the n-th Fibonacci number, with exponential complexity

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

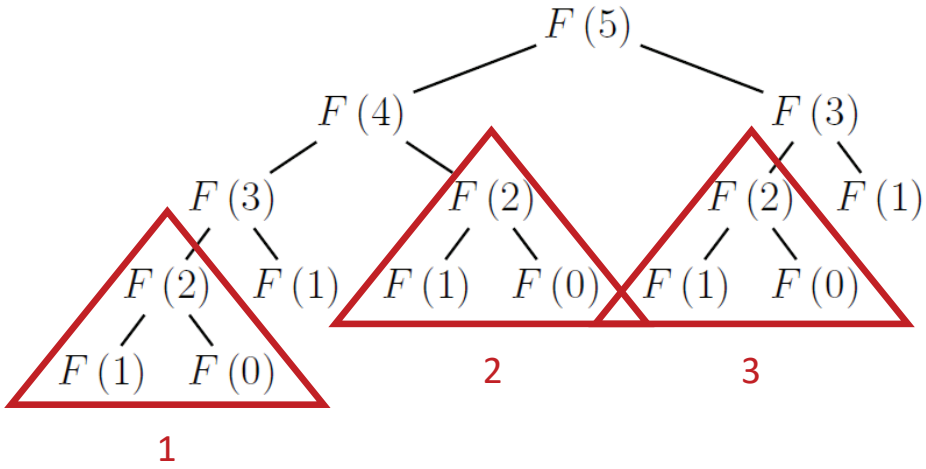


- What can we notice in the recursion tree?

Input enhancement

- In **Lecture 6**, you discussed a simple recursive algorithm or finding the n-th Fibonacci number, with exponential complexity

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

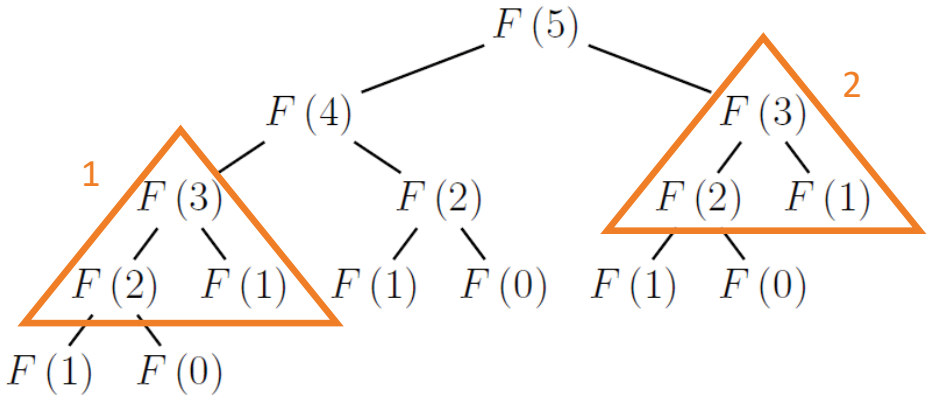


- What can we notice in the recursion tree?
  - There is a lot of redundancy

Input enhancement

- In **Lecture 6**, you discussed a simple recursive algorithm or finding the n-th Fibonacci number, with exponential complexity

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



- What can we notice in the recursion tree?
  - There is a lot of redundancy

Input enhancement

- Let's modify this algorithm to **store intermediate results** of $\text{FIB}(i)$
  - Let's use an array $F[0,\ldots,n]=0$ as a table
  - Once $\text{FIB}(i)$ has been calculated, its value is stored in $F[i]$
  - Before each call to $\text{FIB}(i)$, $F[i]$ is **checked first**. If $F[i] = 0$, the recursive process starts

These are the base cases, return 1 →

```
function FIB(n)
    if n = 0 or n = 1 then
        return 1
    x ← F[n]
    if x = 0 then
        x ← FIB(n − 1) + FIB(n − 2)
        F[n] ← x
    return x
```

Input enhancement

- Let's modify this algorithm to **store intermediate results** of $\text{FIB}(i)$
  - Let's use an array $F[0,\ldots,n]=0$ as a table
  - Once $\text{FIB}(i)$ has been calculated, its value is stored in $F[i]$
  - Before each call to $\text{FIB}(i)$, $F[i]$ is **checked first**. If $F[i] = 0$, the recursive process starts

Otherwise, we take the value from the table →

```
function FIB(n)
    if n = 0 or n = 1 then
        return 1
    x ← F[n]
    if x = 0 then
        x ← FIB(n − 1) + FIB(n − 2)
        F[n] ← x
    return x
```

Input enhancement

- Let's modify this algorithm to **store intermediate results** of $\text{FIB}(i)$
  - Let's use an array $F[0,\ldots,n]=0$ as a table
  - Once $\text{FIB}(i)$ has been calculated, its value is stored in $F[i]$
  - Before each call to $\text{FIB}(i)$, $F[i]$ is **checked first**. If $F[i] = 0$, the recursive process starts

Run the recursion, if the value is empty →

```
function FIB(n)
    if n = 0 or n = 1 then
        return 1
    x ← F[n]
    if x = 0 then
        x ← FIB(n − 1) + FIB(n − 2)
        F[n] ← x
    return x
```

Input enhancement

- Let's modify this algorithm to **store intermediate results** of $\text{FIB}(i)$
  - Let's use an array $F[0,\ldots,n]=0$ as a table
  - Once $\text{FIB}(i)$ has been calculated, its value is stored in $F[i]$
  - Before each call to $\text{FIB}(i)$, $F[i]$ is **checked first**. If $F[i] = 0$, the recursive process starts

Store the result in the table →

```
function FIB(n)
    if n = 0 or n = 1 then
        return 1
    x ← F[n]
    if x = 0 then
        x ← FIB(n − 1) + FIB(n − 2)
        F[n] ← x
    return x
```

Input enhancement

- Let's modify this algorithm to **store intermediate results** of $\text{FIB}(i)$
  - Let's use an array $F[0,\ldots,n]=0$ as a table
  - Once $\text{FIB}(i)$ has been calculated, its value is stored in $F[i]$
  - Before each call to $\text{FIB}(i)$, $F[i]$ is **checked first**. If $F[i] = 0$, the recursive process starts

```
function FIB(n)
    if n = 0 or n = 1 then
        return 1
    x ← F[n]
    if x = 0 then
        x ← FIB(n − 1) + FIB(n − 2)
        F[n] ← x
    return x
```

Return the value from the table →

Distribution counting

- Let's apply input enhancement to sorting. Suppose you need to sort a **large array** that holds **keys** from a **small, fixed set**, i.e., lots of duplicates

- For example, let's sort this array of **single digit integers**

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

- Let's count the **frequency** of each key in a table, using a linear scan

key	0	1	2	3	4	5	6	7	8	9
Frequency	0	0	0	0	0	0	0	0	0	0

Distribution counting

- Let's apply input enhancement to sorting. Suppose you need to sort a **large array** that holds **keys** from a **small, fixed set**, i.e., lots of duplicates

- For example, let's sort this array of **single digit integers**

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

- Let's count the **frequency** of each key in a table, using a linear scan

key	0	1	2	3	4	5	6	7	8	9
Frequency	0	0	0	0	0	0	1	0	0	0

Distribution counting

- Let's apply input enhancement to sorting. Suppose you need to sort a **large array** that holds **keys** from a **small, fixed set**, i.e., lots of duplicates

- For example, let's sort this array of **single digit integers**

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

- Let's count the **frequency** of each key in a table, using a linear scan

key	0	1	2	3	4	5	6	7	8	9
Frequency	0	0	0	1	0	0	1	0	0	0

Distribution counting

- Let's apply input enhancement to sorting. Suppose you need to sort a **large array** that holds **keys** from a **small, fixed set**, i.e., lots of duplicates

- For example, let's sort this array of **single digit integers**

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

- Let's count the **frequency** of each key in a table, using a linear scan

key	0	1	2	3	4	5	6	7	8	9
Frequency	0	0	0	2	0	0	1	0	0	0

Distribution counting

- Let's apply input enhancement to sorting. Suppose you need to sort a **large array** that holds **keys** from a **small, fixed set**, i.e., lots of duplicates

- For example, let's sort this array of **single digit integers**

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

- Let's count the **frequency** of each key in a table, using a linear scan

key	0	1	2	3	4	5	6	7	8	9
Frequency	0	0	0	2	0	0	1	0	1	0

Distribution counting

- Let's apply input enhancement to sorting. Suppose you need to sort a **large array** that holds **keys** from a **small, fixed set**, i.e., lots of duplicates

- For example, let's sort this array of **single digit integers**

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

- Let's count the **frequency** of each key in a table, using a linear scan

key	0	1	2	3	4	5	6	7	8	9
Frequency	0	1	0	2	0	0	1	0	1	0

Distribution counting

- Let's apply input enhancement to sorting. Suppose you need to sort a **large array** that holds **keys** from a **small, fixed set**, i.e., lots of duplicates

- For example, let's sort this array of **single digit integers**

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

- Let's count the **frequency** of each key in a table, using a linear scan

key	0	1	2	3	4	5	6	7	8	9
Frequency	1	1	0	2	0	0	1	0	1	0

Distribution counting

- Let's apply input enhancement to sorting. Suppose you need to sort a **large array** that holds **keys** from a **small, fixed set**, i.e., lots of duplicates

- For example, let's sort this array of **single digit integers**

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

- Let's count the **frequency** of each key in a table, using a linear scan

key	0	1	2	3	4	5	6	7	8	9
Frequency	1	1	0	2	0	0	1	0	2	0

Distribution counting

- Let's apply input enhancement to sorting. Suppose you need to sort a **large array** that holds **keys** from a **small, fixed set**, i.e., lots of duplicates

- For example, let's sort this array of **single digit integers**

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

- Let's count the **frequency** of each key in a table, using a linear scan

key	0	1	2	3	4	5	6	7	8	9
Frequency	1	1	0	2	0	0	1	1	2	0

Distribution counting

- Let's apply input enhancement to sorting. Suppose you need to sort a **large array** that holds **keys** from a **small, fixed set**, i.e., lots of duplicates

- For example, let's sort this array of **single digit integers**

- Let's count the **frequency** of each key in a table, using a linear scan

key	0	1	2	3	4	5	6	7	8	9
Frequency	1	4	2	5	0	4	2	2	3	1

Distribution counting

- Let's use a second linear scan to **accumulate** the counts

key	0	1	2	3	4	5	6	7	8	9
Frequency	1	4	2	5	0	4	2	2	3	1
Accumulation	0	0	0	0	0	0	0	0	0	0

Distribution counting

- Let's use a second linear scan to **accumulate** the counts

key	0	1	2	3	4	5	6	7	8	9
Frequency	1	4	2	5	0	4	2	2	3	1
Accumulation	1	0	0	0	0	0	0	0	0	0

Distribution counting

- Let's use a second linear scan to **accumulate** the counts

key	0	1	2	3	4	5	6	7	8	9
Frequency	1	4	2	5	0	4	2	2	3	1
Accumulation	1	5	0	0	0	0	0	0	0	0

Distribution counting

- Let's use a second linear scan to **accumulate** the counts

key	0	1	2	3	4	5	6	7	8	9
Frequency	1	4	2	5	0	4	2	2	3	1
Accumulation	1	5	7	0	0	0	0	0	0	0

Distribution counting

- Let's use a second linear scan to **accumulate** the counts

key	0	1	2	3	4	5	6	7	8	9
Frequency	1	4	2	5	0	4	2	2	3	1
Accumulation	1	5	7	12	0	0	0	0	0	0

Distribution counting

- Let's use a second linear scan to **accumulate** the counts

key	0	1	2	3	4	5	6	7	8	9
Frequency	1	4	2	5	0	4	2	2	3	1
Accumulation	1	5	7	12	12	0	0	0	0	0

Distribution counting

- Let's use a second linear scan to **accumulate** the counts

key	0	1	2	3	4	5	6	7	8	9
Frequency	1	4	2	5	0	4	2	2	3	1
Accumulation	1	5	7	12	12	16	0	0	0	0

Distribution counting

- Let's use a second linear scan to **accumulate** the counts

key	0	1	2	3	4	5	6	7	8	9
Frequency	1	4	2	5	0	4	2	2	3	1
Accumulation	1	5	7	12	12	16	18	0	0	0

Distribution counting

- Let's use a second linear scan to **accumulate** the counts

key	0	1	2	3	4	5	6	7	8	9
Frequency	1	4	2	5	0	4	2	2	3	1
Accumulation	1	5	7	12	12	16	18	20	0	0

Distribution counting

- Let's use a second linear scan to **accumulate** the counts

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

Distribution counting

- Let's use a second linear scan to **accumulate** the counts

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

Distribution counting

- Let's use a second linear scan to **accumulate** the counts

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

- Now, let's sort the data using a third scan. Note that **accumulation** value on the table represents the **index of the element to insert**

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
6	3	3	8	1	0	8	7	9	2	5	3	5	3	1	8	7	6	5	1	2	1	5	3

Distribution counting

- Let's use a second linear scan to **accumulate** the counts

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

- Now, let's sort the data using a third scan. Note that **accumulation** value on the table represents the **index of the element to insert**

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
6	3	3	8	1	0	8	7	9	2	5	3	5	3	1	8	7	6	5	1	2	1	5	3
											3												

Distribution counting

- Let's use a second linear scan to **accumulate** the counts

key	0	1	2	3	4	5	6	7	8	9
Frequency	1	4	2	5	0	4	2	2	3	1
Accumulation	1	5	7	11	12	16	18	20	23	24

- Now, let's sort the data using a third scan. Note that **accumulation** value on the table represents the **index of the element to insert**

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
6	3	3	8	1	0	8	7	9	2	5	3	5	3	1	8	7	6	5	1	2	1	5	3
											3												

Distribution counting

- Let's use a second linear scan to **accumulate** the counts

key	0	1	2	3	4	5	6	7	8	9
Frequency	1	4	2	5	0	4	2	2	3	1
Accumulation	1	5	7	11	12	16	18	20	23	24

- Now, let's sort the data using a third scan. Note that **accumulation** value on the table represents the **index of the element to insert**

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
6	3	3	8	1	0	8	7	9	2	5	3	5	3	1	8	7	6	5	1	2	1	5	3
											3				5								

Distribution counting

- Let's use a second linear scan to **accumulate** the counts

key	0	1	2	3	4	5	6	7	8	9
Frequency	1	4	2	5	0	4	2	2	3	1
Accumulation	1	5	7	11	12	15	18	20	23	24

- Now, let's sort the data using a third scan. Note that **accumulation** value on the table represents the **index of the element to insert**

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
6	3	3	8	1	0	8	7	9	2	5	3	5	3	1	8	7	6	5	1	2	1	5	3
											3				5								

Distribution counting

- Let's use a second linear scan to **accumulate** the counts

key	0	1	2	3	4	5	6	7	8	9
Frequency	1	4	2	5	0	4	2	2	3	1
Accumulation	1	5	7	11	12	15	18	20	23	24

- Now, let's sort the data using a third scan. Note that **accumulation** value on the table represents the **index of the element to insert**

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
6	3	3	8	1	0	8	7	9	2	5	3	5	3	1	8	7	6	5	1	2	1	5	3
				1							3				5								

Distribution counting

- Let's use a second linear scan to **accumulate** the counts

key	0	1	2	3	4	5	6	7	8	9
Frequency	1	4	2	5	0	4	2	2	3	1
Accumulation	1	4	7	11	12	15	18	20	23	24

- Now, let's sort the data using a third scan. Note that **accumulation** value on the table represents the **index of the element to insert**

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
6	3	3	8	1	0	8	7	9	2	5	3	5	3	1	8	7	6	5	1	2	1	5	3
				1							3				5								

Distribution counting

- Let's use a second linear scan to **accumulate** the counts

key	0	1	2	3	4	5	6	7	8	9
Frequency	1	4	2	5	0	4	2	2	3	1
Accumulation	1	4	7	11	12	15	18	20	23	24

- Now, let's sort the data using a third scan. Note that **accumulation** value on the table represents the **index of the element to insert**

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
6	3	3	8	1	0	8	7	9	2	5	3	5	3	1	8	7	6	5	1	2	1	5	3
				1		2					3				5								

Distribution counting

- Let's use a second linear scan to **accumulate** the counts

key	0	1	2	3	4	5	6	7	8	9
Frequency	1	4	2	5	0	4	2	2	3	1
Accumulation	1	4	6	11	12	15	18	20	23	24

- Now, let's sort the data using a third scan. Note that **accumulation** value on the table represents the **index of the element to insert**

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
6	3	3	8	1	0	8	7	9	2	5	3	5	3	1	8	7	6	5	1	2	1	5	3
				1		2					3				5								

Distribution counting

- Let's use a second linear scan to **accumulate** the counts

key	0	1	2	3	4	5	6	7	8	9
Frequency	1	4	2	5	0	4	2	2	3	1
Accumulation	1	4	6	11	12	15	18	20	23	24

- Now, let's sort the data using a third scan. Note that **accumulation** value on the table represents the **index of the element to insert**

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
6	3	3	8	1	0	8	7	9	2	5	3	5	3	1	8	7	6	5	1	2	1	5	3
			1	1		2					3				5								

Distribution counting

- Let's use a second linear scan to **accumulate** the counts

key	0	1	2	3	4	5	6	7	8	9
Frequency	1	4	2	5	0	4	2	2	3	1
Accumulation	1	3	6	11	12	15	18	20	23	24

- Now, let's sort the data using a third scan. Note that **accumulation** value on the table represents the **index of the element to insert**

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
6	3	3	8	1	0	8	7	9	2	5	3	5	3	1	8	7	6	5	1	2	1	5	3
			1	1		2					3				5								

Distribution counting

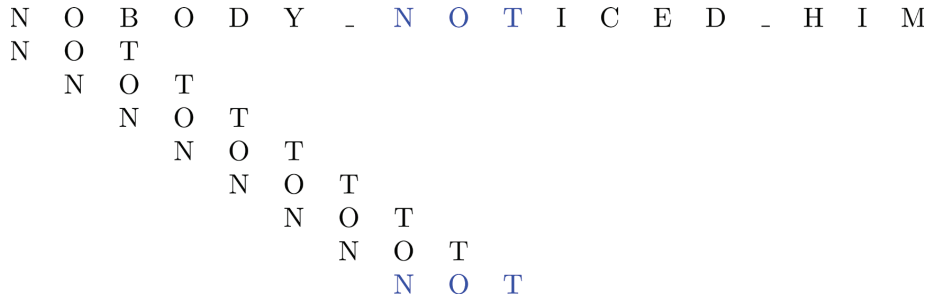
- This is a **linear-time** sorting algorithm, with $O(n+k)$ complexity, where n is the number of items, and k is the number of possible keys
  - Hence, for it to fulfill its guarantees k must be known and $k<<n$

- Because it does **not perform key-to-key comparisons**, the lower bound time complexity of a **comparison sort** of $\Omega(n \log n)$ **does not apply**

String matching revisited

- In **Lecture 5**, you studied a **brute-force** approach to string matching

```
for i ← 0 to n − m do
    j ← 0
    while j < m and p[j] = t[i + j] do
        j ← j + 1
    if j = m then
        return i
return −1
```



- Isn't this very inefficient?

String matching revisited

- **Strings** are usually built from a small, pre-determined **alphabet**
  - Most of the better algorithms rely on some form of **pre-processing** before the actual search
  - Pre-processing involves the construction of a **small table** of predictable size

- We will discuss these advanced string matching algorithms
  - Horspool's Algorithm
  - Knuth-Morris-Pratt Algorithm
  - Rabin-Karp Algorithm (Lecture 17)

Horspool's Algorithm

- One of the reasons the brute-force approach is so inefficient is because it compares the **first character** of the pattern

```
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
```

Horspool's Algorithm

- One of the reasons the brute-force approach is so inefficient is because it compares the **first character** of the pattern

```
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
```

- However, if we were to compare the **last character** of the pattern, we could quickly discard larger sections of the string

```
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
```

Horspool's Algorithm

- Did you noticed that sometimes we **do not shift the complete length** of the pattern?
- On the first two attempts, we knew that the pattern did not had $\{B,Y\}$; hence we could shift the whole pattern. However, in the next attempt, there is a partial match:

```
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
```

- To determine the **number of shifts**, we need to build a table

Horspool's Algorithm

- Let's work with an example with a small alphabet size ($\alpha = 4$)
  - The pattern is $P[.] = \text{TAACG}$ ($m = 5$)
  - The string is $T[.] = \text{GACCGCGTGAGATAACGTCA}$
  - The alphabet is [A T G C]

- We use this algorithm to create the shift table:

```
function FindShifts(P[·], m)
    for i ← 0 to α − 1 do
        Shift[i] ← m
    for j ← 0 to m − 2 do
        Shift[P[j]] ← m − (j + 1)
```

Horspool's Algorithm

- Let's work with an example with a small alphabet size ($\alpha = 4$)
  - The pattern is $P[.]$ = TAACG ($m = 5$)
  - The string is $T[.]$ = GACCGCGTGAGATAACGTCA
  - The alphabet is [A T G C]

- We use this algorithm to create the shift table:

1 → **function** FINDSHIFTS($P[\cdot], m$)
  for $i \leftarrow 0$ to $\alpha - 1$ **do**
    $Shift[i] \leftarrow m$
  for $j \leftarrow 0$ to $m - 2$ **do**
    $Shift[P[j]] \leftarrow m - (j + 1)$

Memory state (1)	
$P[.]$	$[T,\ A,\ A,\ C,\ G]$
m	5
α	4
i	
j	
$Shift[.]$	$[0,\ 0,\ 0,\ 0]$
$P[j]$	
$m - (j + 1)$	

Horspool's Algorithm

- Let's work with an example with a small alphabet size ($\alpha = 4$)
  - The pattern is $P[.]$ = TAACG ($m = 5$)
  - The string is $T[.]$ = GACCGCGTGAGATAACGTCA
  - The alphabet is [A T G C]

- We use this algorithm to create the shift table:

function FINDSHIFTS($P[\cdot], m$)
  for $i \leftarrow 0$ to $\alpha - 1$ **do**
    $Shift[i] \leftarrow m$
  for $j \leftarrow 0$ to $m - 2$ **do**
    $Shift[P[j]] \leftarrow m - (j + 1)$

2-5

Memory state (2)	
$P[.]$	$[T,\ A,\ A,\ C,\ G]$
m	5
α	4
i	**0**
j	
$Shift[.]$	[**5**, 0, 0, 0]
$P[j]$	
$m - (j + 1)$	

Horspool's Algorithm

- Let's work with an example with a small alphabet size ($\alpha = 4$)
  - The pattern is $P[.]$ = TAACG ($m = 5$)
  - The string is $T[.]$ = GACCGCGTGAGATAACGTCA
  - The alphabet is [A T G C]

- We use this algorithm to create the shift table:

function FINDSHIFTS($P[\cdot], m$)
  for $i \leftarrow 0$ to $\alpha - 1$ **do**
    $Shift[i] \leftarrow m$
  for $j \leftarrow 0$ to $m - 2$ **do**
    $Shift[P[j]] \leftarrow m - (j + 1)$

2-5 →

Memory state (3)	
$P[.]$	$[T,\ A,\ A,\ C,\ G]$
m	5
α	4
i	**1**
j	
$Shift[.]$	$[5,\ \mathbf{5},\ 0,\ 0]$
$P[j]$	
$m - (j + 1)$	

Horspool's Algorithm

- Let's work with an example with a small alphabet size ($\alpha = 4$)
  - The pattern is $P[.]$ = TAACG ($m = 5$)
  - The string is $T[.]$ = GACCGCGTGAGATAACGTCA
  - The alphabet is [A T G C]

- We use this algorithm to create the shift table:

function FINDSHIFTS($P[\cdot], m$)
  for $i \leftarrow 0$ to $\alpha - 1$ **do**
    $Shift[i] \leftarrow m$
  for $j \leftarrow 0$ to $m - 2$ **do**
    $Shift[P[j]] \leftarrow m - (j + 1)$

2-5 →

Memory state (4)	
$P[.]$	$[T,\ A,\ A,\ C,\ G]$
m	5
α	4
i	**2**
j	
$Shift[.]$	$[5,\ 5,\ \mathbf{5},\ 0]$
$P[j]$	
$m - (j + 1)$	

Horspool's Algorithm

- Let's work with an example with a small alphabet size ($\alpha = 4$)
  - The pattern is $P[.]$ = TAACG ($m = 5$)
  - The string is $T[.]$ = GACCGCGTGAGATAACGTCA
  - The alphabet is [A T G C]

- We use this algorithm to create the shift table:

function FINDSHIFTS($P[\cdot], m$)
  for $i \leftarrow 0$ to $\alpha - 1$ **do**
    $Shift[i] \leftarrow m$
  for $j \leftarrow 0$ to $m - 2$ **do**
    $Shift[P[j]] \leftarrow m - (j + 1)$

2-5 →

Memory state (5)	
$P[.]$	$[T,\ A,\ A,\ C,\ G]$
m	5
α	4
i	**3**
j	
$Shift[.]$	$[5,\ 5,\ 5,\ \mathbf{5}]$
$P[j]$	
$m - (j + 1)$	

Horspool's Algorithm

- Let's work with an example with a small alphabet size ($\alpha = 4$)
  - The pattern is $P[.]$ = TAACG ($m = 5$)
  - The string is $T[.]$ = GACCGCGTGAGATAACGTCA
  - The alphabet is [A T G C]

- We use this algorithm to create the shift table:

function FINDSHIFTS($P[\cdot], m$)
  for $i \leftarrow 0$ to $\alpha - 1$ **do**
    $Shift[i] \leftarrow m$
  for $j \leftarrow 0$ to $m - 2$ **do**
    $Shift[P[j]] \leftarrow m - (j + 1)$

6-9

Memory state (6)	
$P[.]$	$[T,\ A,\ A,\ C,\ G]$
m	5
α	4
i	0
j	**0**
$Shift[.]$	$[5,\ \mathbf{4},\ 5,\ 5]$
$P[j]$	**T**
$m - (j + 1)$	**4**

Horspool's Algorithm

- Let's work with an example with a small alphabet size ($\alpha = 4$)
  - The pattern is $P[.]$ = TAACG ($m = 5$)
  - The string is $T[.]$ = GACCGCGTGAGATAACGTCA
  - The alphabet is [A T G C]

- We use this algorithm to create the shift table:

function FINDSHIFTS($P[\cdot], m$)
  for $i \leftarrow 0$ to $\alpha - 1$ **do**
    $Shift[i] \leftarrow m$
  for $j \leftarrow 0$ to $m - 2$ **do**
    $Shift[P[j]] \leftarrow m - (j + 1)$

6-9 →

Memory state (7)	
$P[.]$	$[T,\ A,\ A,\ C,\ G]$
m	5
α	4
i	0
j	**1**
$Shift[.]$	[**3**, 4, 5, 5]
$P[j]$	**A**
$m - (j + 1)$	**3**

Horspool's Algorithm

- Let's work with an example with a small alphabet size ($\alpha = 4$)
  - The pattern is $P[.]$ = TAACG ($m = 5$)
  - The string is $T[.]$ = GACCGCGTGAGATAACGTCA
  - The alphabet is [A T G C]

- We use this algorithm to create the shift table:

function FINDSHIFTS($P[\cdot], m$)
  for $i \leftarrow 0$ to $\alpha - 1$ **do**
    $Shift[i] \leftarrow m$
  for $j \leftarrow 0$ to $m - 2$ **do**
    $Shift[P[j]] \leftarrow m - (j + 1)$

6-9

Memory state (8)	
$P[.]$	$[T,\ A,\ A,\ C,\ G]$
m	5
α	4
i	0
j	**2**
$Shift[.]$	[**2**, 4, 5, 5]
$P[j]$	**A**
$m - (j + 1)$	**2**

Horspool's Algorithm

- Let's work with an example with a small alphabet size ($\alpha = 4$)
  - The pattern is $P[.]$ = TAACG ($m = 5$)
  - The string is $T[.]$ = GACCGCGTGAGATAACGTCA
  - The alphabet is [A T G C]

- We use this algorithm to create the shift table:

```
function FindShifts(P[·], m)
    for i ← 0 to α − 1 do
        Shift[i] ← m
    for j ← 0 to m − 2 do
        Shift[P[j]] ← m − (j + 1)
```

6-9

Memory state (9)	
$P[.]$	$[T,\ A,\ A,\ C,\ G]$
m	5
α	4
i	0
j	**3**
$Shift[.]$	$[2,\ 4,\ 5,\ \mathbf{1}]$
$P[j]$	**C**
$m-(j+1)$	**1**

Horspool's Algorithm

- A **sentinel** is a **known pattern appended** at the end of the data to be processed
  - Sentinels are used to guarantee **algorithm completion** and avoid **memory overflow**

- To the right, we have version of Horspool's algorithm that **expects** a sentinel
  - The string is $T[.] =$ GACCGCGTGAGATAACGTCATAACG

```
function HORSPOOL(P[·], m, T[·], n)
    FINDSHIFTS(P, m)
    i ← m − 1
    while TRUE do
        k ← 0
        while k < m and P[m − 1 − k] = T[i − k] do
            k ← k + 1
        if k = m then
            if i ≥ n then
                return −1
            else
                return i − m + 1
        i ← i + Shift[T[i]]
```

Horspool's Algorithm

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
T[.]	G	A	C	C	G	C	G	T	G	A	G	A	T	A	A	C	G	T	C	A	T	A	A	C	G
FAILED (C!=A, SHIFT BY G)	T	A	A	C	G																				

```
function HorspoolMatching(P[·], m, T[·], n)
    FindShifts(P, m)
    i ← m − 1
    while true do
        k ← 0
1 →     while k < m and P[m − 1 − k] = T[i − k] do
            k ← k + 1
        if k = m then
            if i ≥ n then
                return −1
            else
                return i − m + 1
        i ← i + Shift[T[i]]
```

Memory state (1)	
$P[.]$	$[T,\ A,\ A,\ C,\ G]$
m	5
n	25
i	4
k	0
$k < m$	**true**
$P[4]$	**G**
$T[4]$	**G**
$Shift[.]$	$[2,\ 4,\ 5,\ 1]$

Horspool's Algorithm

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
T[.]	G	A	C	C	G	C	G	T	G	A	G	A	T	A	A	C	G	T	C	A	T	A	A	C	G
FAILED (C!=A, SHIFT BY G)	T	A	A	C	G																				

2 →

```
function HorspoolMatching(P[·], m, T[·], n)
    FindShifts(P, m)
    i ← m − 1
    while true do
        k ← 0
        while k < m and P[m − 1 − k] = T[i − k] do
            k ← k + 1
        if k = m then
            if i ≥ n then
                return −1
            else
                return i − m + 1
        i ← i + Shift[T[i]]
```

Memory state (2)	
$P[.]$	$[T,\ A,\ A,\ C,\ G]$
m	5
n	25
i	4
k	**1**
$k < m$	**true**
$P[3]$	**C**
$T[3]$	**C**
$Shift[.]$	$[2,\ 4,\ 5,\ 1]$

Horspool's Algorithm

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
T[.]	G	A	C	C	G	C	G	T	G	A	G	A	T	A	A	C	G	T	C	A	T	A	A	C	G
FAILED (C!=A, SHIFT BY G)	T	A	A	C	G																				

3 →

```
function HORSPOOL(P[·], m, T[·], n)
    FINDSHIFTS(P, m)
    i ← m − 1
    while TRUE do
        k ← 0
        while k < m and P[m − 1 − k] = T[i − k] do
            k ← k + 1
        if k = m then
            if i ≥ n then
                return −1
            else
                return i − m + 1
        i ← i + Shift[T[i]]
```

Memory state (3)	
$P[.]$	$[T,\ A,\ A,\ C,\ G]$
m	5
n	25
i	4
k	**2**
$k < m$	**true**
$P[2]$	**C**
$T[2]$	**A**
$Shift[.]$	$[2,\ 4,\ 5,\ 1]$

Horspool's Algorithm

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
T[.]	G	A	C	C	G	C	G	T	G	A	G	A	T	A	A	C	G	T	C	A	T	A	A	C	G
FAILED (C!=A, SHIFT BY G)	T	A	A	C	G																				

```
function HorspoolMatching(P[·], m, T[·], n)
    FindShifts(P, m)
    i ← m − 1
    while true do
        k ← 0
        while k < m and P[m − 1 − k] = T[i − k] do
            k ← k + 1
        if k = m then
            if i ≥ n then
                return −1
            else
                return i − m + 1
4 →     i ← i + Shift[T[i]]
```

Memory state (4)	
$P[.]$	$[T,\ A,\ A,\ C,\ G]$
m	5
n	25
i	**9**
k	2
$k < m$	true
$P[2]$	C
$T[2]$	A
$Shift[.]$	$[2,\ 4,\ 5,\ 1]$
$Shift[G]$	**5**

Horspool's Algorithm

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
T[.]	G	A	C	C	G	C	G	T	G	A	G	A	T	A	A	C	G	T	C	A	T	A	A	C	G
FAILED (C!=A, SHIFT BY G)	T	A	A	C	G																				
FAILED (A!=G, SHIFT BY A)						T	A	A	C	G															

function HORSPOOL($P[\cdot], m, T[\cdot], n$)
    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 $i - m + 1$
        $i \leftarrow i + \mathit{Shift}[T[i]]$

Horspool's Algorithm

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
T[.]	G	A	C	C	G	C	G	T	G	A	G	A	T	A	A	C	G	T	C	A	T	A	A	C	G
FAILED (C!=A, SHIFT BY G)	T	A	A	C	G																				
FAILED (A!=G, SHIFT BY A)						T	A	A	C	G															
FAILED (A!=G, SHIFT BY A)								T	A	A	C	G													

```
function HORSPOOL(P[·], m, T[·], n)
    FINDSHIFTS(P, m)
    i ← m − 1
    while TRUE do
        k ← 0
        while k < m and P[m − 1 − k] = T[i − k] do
            k ← k + 1
        if k = m then
            if i ≥ n then
                return −1
            else
                return i − m + 1
        i ← i + Shift[T[i]]
```

Horspool's Algorithm

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
T[.]	G	A	C	C	G	C	G	T	G	A	G	A	T	A	A	C	G	T	C	A	T	A	A	C	G
FAILED (C!=A, SHIFT BY G)	T	A	A	C	G																				
FAILED (A!=G, SHIFT BY A)						T	A	A	C	G															
FAILED (A!=G, SHIFT BY A)								T	A	A	C	G													
FAILED (A!=G, SHIFT BY A)										T	A	A	C	G											

```
function HORSPOOL(P[·], m, T[·], n)
    FINDSHIFTS(P, m)
    i ← m − 1
    while TRUE do
        k ← 0
        while k < m and P[m − 1 − k] = T[i − k] do
            k ← k + 1
        if k = m then
            if i ≥ n then
                return −1
            else
                return i − m + 1
        i ← i + Shift[T[i]]
```

Horspool's Algorithm

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
T[.]	G	A	C	C	G	C	G	T	G	A	G	A	T	A	A	C	G	T	C	A	T	A	A	C	G
FAILED (C!=A, SHIFT BY G)	T	A	A	C	G																				
FAILED (A!=G, SHIFT BY A)						T	A	A	C	G															
FAILED (A!=G, SHIFT BY A)								T	A	A	C	G													
FAILED (A!=G, SHIFT BY A)										T	A	A	C	G											
FAILED (C!=G, SHIFT BY C)												T	A	A	C	G									

function HORSPOOL($P[\cdot], m, T[\cdot], n$)
  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 $i - m + 1$
    $i \leftarrow i + \mathit{Shift}[T[i]]$

Horspool's Algorithm

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
T[.]	G	A	C	C	G	C	G	T	G	A	G	A	T	A	A	C	G	T	C	A	T	A	A	C	G
FAILED (C!=A, SHIFT BY G)	T	A	A	C	G																				
FAILED (A!=G, SHIFT BY A)						T	A	A	C	G															
FAILED (A!=G, SHIFT BY A)								T	A	A	C	G													
FAILED (A!=G, SHIFT BY A)										T	A	A	C	G											
FAILED (C!=G, SHIFT BY C)												T	A	A	C	G									
FOUND AT $i = 16$													T	A	A	C	G								

function HORSPOOL($P[\cdot], m, T[\cdot], n$)
  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 $i - m + 1$ ← RETURNS 12
    $i \leftarrow i + \mathit{Shift}[T[i]]$

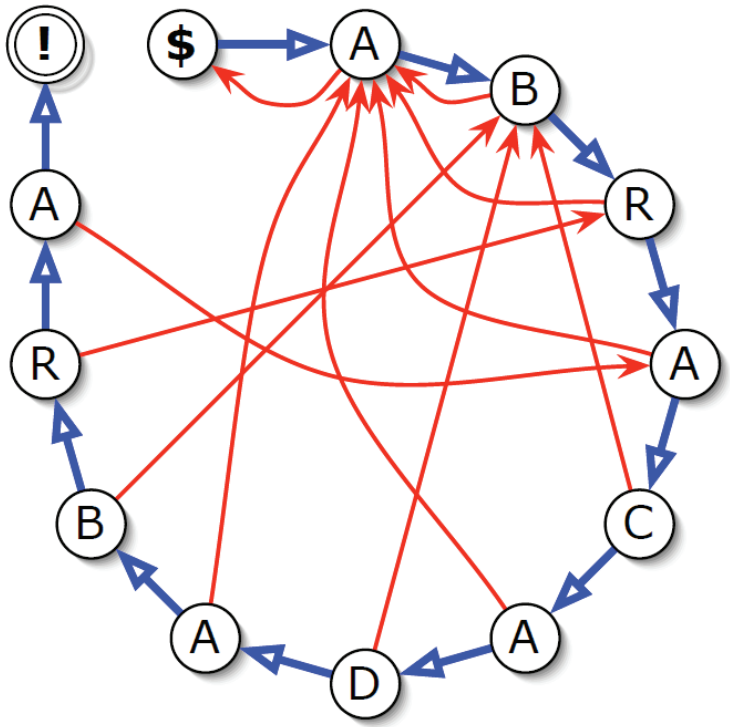
Horspool's Algorithm

- The **worst-case behavior** of the Horspool's algorithm is $O(m \times n)$, like the brute-force method

- In practice (for example, when analyzing English texts), it is closer to $O(n)$.

Knuth-Morris-Pratt Algorithm

- A string matching algorithm can be thought of feeding the text through a **finite-state machine (FSM)**, a directed graph that:
  - Its **nodes** are labelled by **each character** from the pattern
  - Each node has a **success** and **failure** edges
  - Success edges define a path **through** the characters of the pattern in order
  - Failure edges always point to **earlier** characters in the pattern



Knuth-Morris-Pratt Algorithm

- In practice, we only encode the failure edges as an **array**
- Let's work with this (slightly different) example ($\alpha = 4$)
  - The pattern is $P[1..m]$ = TAATG ($m = 5$)
  - The string is $T[1..n]$ = GACCGCGTGAGATAATGTCA
  - The alphabet is [A T G C]
  - P[0] = NULL

```
function ComputeFailure(P[·], m)
    j ← 0
    for i ← 1 to m do
1       if P[i] = P[j] then
            fail[i] ← fail[j]
        else
            fail[i] ← j
        while j > 0 and P[i] ≠ P[j] do
            j ← fail[j]
        j ← j + 1
```

Memory state (1)	
$P[.]$	$[T,\ A,\ A,\ T,\ G]$
m	5
i	1
j	0
$P[i]$	T
$P[j]$	NULL
$fail[.]$	[., ., ., ., .]

Knuth-Morris-Pratt Algorithm

- In practice, we only encode the failure edges as an **array**
- Let's work with this (slightly different) example ($\alpha = 4$)
  - The pattern is $P[1..m]$ = TAATG ($m = 5$)
  - The string is $T[1..n]$ = GACCGCGTGAGATAATGTCA
  - The alphabet is [A T G C]
  - P[0] = NULL

```
function ComputeFailure(P[·], m)
    j ← 0
    for i ← 1 to m do
        if P[i] = P[j] then
            fail[i] ← fail[j]
        else
            fail[i] ← j
        while j > 0 and P[i] ≠ P[j] do
            j ← fail[j]
        j ← j + 1
```

2 →

Memory state (2)	
$P[.]$	$[T,\ A,\ A,\ T,\ G]$
m	5
i	1
j	0
$P[i]$	T
$P[j]$	NULL
$fail[.]$	[0, ., ., ., .]

Knuth-Morris-Pratt Algorithm

- In practice, we only encode the failure edges as an **array**
- Let's work with this (slightly different) example ($\alpha = 4$)
  - The pattern is $P[1..m]$ = TAATG ($m = 5$)
  - The string is $T[1..n]$ = GACCGCGTGAGATAATGTCA
  - The alphabet is [A T G C]
  - P[0] = NULL

```
function ComputeFailure(P[·], m)
    j ← 0
    for i ← 1 to m do
        if P[i] = P[j] then
            fail[i] ← fail[j]
        else
            fail[i] ← j
        while j > 0 and P[i] ≠ P[j] do
            j ← fail[j]
3 →     j ← j + 1
```

Memory state (3)	
P [.]	[T, A, A, T, G]
m	5
i	1
j	1
P [i]	T
P [j]	NULL
$fail$ [.]	[0, ., ., ., .]

Knuth-Morris-Pratt Algorithm

- In practice, we only encode the failure edges as an **array**
- Let's work with this (slightly different) example ($\alpha = 4$)
  - The pattern is $P[1..m]$ = TAATG ($m = 5$)
  - The string is $T[1..n]$ = GACCGCGTGAGATAATGTCA
  - The alphabet is [A T G C]
  - P[0] = NULL

```
function ComputeFailure(P[·], m)
    j ← 0
    for i ← 1 to m do
4 ──→   if P[i] = P[j] then
            fail[i] ← fail[j]
        else
            fail[i] ← j
        while j > 0 and P[i] ≠ P[j] do
            j ← fail[j]
        j ← j + 1
```

Memory state (4)	
P [.]	[T, A, A, T, G]
m	5
i	**2**
j	1
P [i]	**A**
P [j]	**T**
$fail$ [.]	[0, ., ., ., .]

Knuth-Morris-Pratt Algorithm

- In practice, we only encode the failure edges as an **array**
- Let's work with this (slightly different) example ($\alpha = 4$)
  - The pattern is $P[1..m]$ = TAATG ($m = 5$)
  - The string is $T[1..n]$ = GACCGCGTGAGATAATGTCA
  - The alphabet is [A T G C]
  - P[0] = NULL

```
function ComputeFailure(P[·], m)
    j ← 0
    for i ← 1 to m do
        if P[i] = P[j] then
            fail[i] ← fail[j]
        else
            fail[i] ← j
        while j > 0 and P[i] ≠ P[j] do
            j ← fail[j]
        j ← j + 1
```

5 →

Memory state (5)	
P [.]	[T, A, A, T, G]
m	5
i	2
j	1
P [i]	A
P [j]	T
$fail$ [.]	[0, **1**, ., ., .]

Knuth-Morris-Pratt Algorithm

- In practice, we only encode the failure edges as an **array**
- Let's work with this (slightly different) example ($\alpha = 4$)
  - The pattern is $P[1..m]$ = TAATG ($m = 5$)
  - The string is $T[1..n]$ = GACCGCGTGAGATAATGTCA
  - The alphabet is [A T G C]
  - P[0] = NULL

```
function ComputeFailure(P[·], m)
    j ← 0
    for i ← 1 to m do
        if P[i] = P[j] then
            fail[i] ← fail[j]
        else
            fail[i] ← j
        while j > 0 and P[i] ≠ P[j] do
            j ← fail[j]
        j ← j + 1
```

6 →

Memory state (6)	
P [.]	[T, A, A, T, G]
m	5
i	2
j	**0**
P [i]	A
P [j]	T
$fail$ [.]	[0, 1, ., ., .]

Knuth-Morris-Pratt Algorithm

- In practice, we only encode the failure edges as an **array**
- Let's work with this (slightly different) example ($\alpha = 4$)
  - The pattern is $P[1..m]$ = TAATG ($m = 5$)
  - The string is $T[1..n]$ = GACCGCGTGAGATAATGTCA
  - The alphabet is [A T G C]
  - P[0] = NULL

```
function ComputeFailure(P[·], m)
    j ← 0
    for i ← 1 to m do
        if P[i] = P[j] then
            fail[i] ← fail[j]
        else
            fail[i] ← j
        while j > 0 and P[i] ≠ P[j] do
            j ← fail[j]
        j ← j + 1
```

7

Memory state (7)	
P [.]	[T, A, A, T, G]
m	5
i	2
j	1
P [i]	A
P [j]	T
$fail$ [.]	[0, 1, ., ., .]

Knuth-Morris-Pratt Algorithm

- In practice, we only encode the failure edges as an **array**
- Let's work with this (slightly different) example ($\alpha = 4$)
  - The pattern is $P[1..m]$ = TAATG ($m = 5$)
  - The string is $T[1..n]$ = GACCGCGTGAGATAATGTCA
  - The alphabet is [A T G C]
  - P[0] = NULL

```
function ComputeFailure(P[·], m)
    j ← 0
    for i ← 1 to m do
8       if P[i] = P[j] then
            fail[i] ← fail[j]
        else
            fail[i] ← j
        while j > 0 and P[i] ≠ P[j] do
            j ← fail[j]
        j ← j + 1
```

Memory state (8)	
P [.]	[T, A, A, T, G]
m	5
i	**3**
j	1
P [i]	**A**
P [j]	**T**
$fail$ [.]	[0, 1, ., ., .]

Knuth-Morris-Pratt Algorithm

- In practice, we only encode the failure edges as an **array**
- Let's work with this (slightly different) example ($\alpha = 4$)
  - The pattern is $P[1..m]$ = TAATG ($m = 5$)
  - The string is $T[1..n]$ = GACCGCGTGAGATAATGTCA
  - The alphabet is [A T G C]
  - P[0] = NULL

```
function ComputeFailure(P[·], m)
    j ← 0
    for i ← 1 to m do
        if P[i] = P[j] then
            fail[i] ← fail[j]
        else
            fail[i] ← j
        while j > 0 and P[i] ≠ P[j] do
            j ← fail[j]
        j ← j + 1
```

9 →

Memory state (9)	
P [.]	[T, A, A, T, G]
m	5
i	3
j	1
P [i]	A
P [j]	T
$fail$ [.]	[0, 1, **1**, ., .]

Knuth-Morris-Pratt Algorithm

- In practice, we only encode the failure edges as an **array**
- Let's work with this (slightly different) example ($\alpha = 4$)
  - The pattern is $P[1..m]$ = TAATG ($m = 5$)
  - The string is $T[1..n]$ = GACCGCGTGAGATAATGTCA
  - The alphabet is [A T G C]
  - P[0] = NULL

```
function ComputeFailure(P[·], m)
    j ← 0
    for i ← 1 to m do
        if P[i] = P[j] then
            fail[i] ← fail[j]
        else
            fail[i] ← j
        while j > 0 and P[i] ≠ P[j] do
            j ← fail[j]
        j ← j + 1
```

10

Memory state (10)	
P [.]	[T, A, A, T, G]
m	5
i	3
j	**0**
P [i]	A
P [j]	T
$fail$ [.]	[0, 1, 1, ., .]

Knuth-Morris-Pratt Algorithm

- In practice, we only encode the failure edges as an **array**
- Let's work with this (slightly different) example ($\alpha = 4$)
  - The pattern is $P[1..m]$ = TAATG ($m = 5$)
  - The string is $T[1..n]$ = GACCGCGTGAGATAATGTCA
  - The alphabet is [A T G C]
  - P[0] = NULL

```
function ComputeFailure(P[·], m)
    j ← 0
    for i ← 1 to m do
        if P[i] = P[j] then
            fail[i] ← fail[j]
        else
            fail[i] ← j
        while j > 0 and P[i] ≠ P[j] do
            j ← fail[j]
11 ⟶    j ← j + 1
```

Memory state (11)	
P [.]	[T, A, A, T, G]
m	5
i	3
j	1
P [i]	A
P [j]	T
$fail$ [.]	[0, 1, 1, ., .]

Knuth-Morris-Pratt Algorithm

- In practice, we only encode the failure edges as an **array**
- Let's work with this (slightly different) example ($\alpha = 4$)
  - The pattern is $P[1..m]$ = TAATG ($m = 5$)
  - The string is $T[1..n]$ = GACCGCGTGAGATAATGTCA
  - The alphabet is [A T G C]
  - P[0] = NULL

```
function ComputeFailure(P[·], m)
    j ← 0
    for i ← 1 to m do
12      if P[i] = P[j] then
            fail[i] ← fail[j]
        else
            fail[i] ← j
        while j > 0 and P[i] ≠ P[j] do
            j ← fail[j]
        j ← j + 1
```

Memory state (12)	
P [.]	[T, A, A, T, G]
m	5
i	**4**
j	1
P [i]	**T**
P [j]	**T**
$fail$ [.]	[0, 1, 1, ., .]

Knuth-Morris-Pratt Algorithm

- In practice, we only encode the failure edges as an **array**
- Let's work with this (slightly different) example ($\alpha = 4$)
  - The pattern is $P[1..m]$ = TAATG ($m = 5$)
  - The string is $T[1..n]$ = GACCGCGTGAGATAATGTCA
  - The alphabet is [A T G C]
  - P[0] = NULL

```
function ComputeFailure(P[·], m)
    j ← 0
    for i ← 1 to m do
        if P[i] = P[j] then
            fail[i] ← fail[j]
        else
            fail[i] ← j
        while j > 0 and P[i] ≠ P[j] do
            j ← fail[j]
        j ← j + 1
```

13 →

Memory state (13)	
P [.]	[T, A, A, T, G]
m	5
i	4
j	1
P [i]	T
P [j]	T
$fail$ [.]	[0, 1, 1, **0**, .]

Knuth-Morris-Pratt Algorithm

- In practice, we only encode the failure edges as an **array**
- Let's work with this (slightly different) example ($\alpha = 4$)
  - The pattern is $P[1..m]$ = TAATG ($m = 5$)
  - The string is $T[1..n]$ = GACCGCGTGAGATAATGTCA
  - The alphabet is [A T G C]
  - P[0] = NULL

```
function ComputeFailure(P[·], m)
    j ← 0
    for i ← 1 to m do
        if P[i] = P[j] then
            fail[i] ← fail[j]
        else
            fail[i] ← j
        while j > 0 and P[i] ≠ P[j] do
            j ← fail[j]
        j ← j + 1
```

14

Memory state (14)	
P [.]	[T, A, A, T, G]
m	5
i	4
j	**2**
P [i]	T
P [j]	T
$fail$ [.]	[0, 1, 1, 0, .]

Knuth-Morris-Pratt Algorithm

- In practice, we only encode the failure edges as an **array**
- Let's work with this (slightly different) example ($\alpha = 4$)
  - The pattern is $P[1..m]$ = TAATG ($m = 5$)
  - The string is $T[1..n]$ = GACCGCGTGAGATAATGTCA
  - The alphabet is [A T G C]
  - P[0] = NULL

```
function ComputeFailure(P[·], m)
    j ← 0
    for i ← 1 to m do
15      if P[i] = P[j] then
            fail[i] ← fail[j]
        else
            fail[i] ← j
        while j > 0 and P[i] ≠ P[j] do
            j ← fail[j]
        j ← j + 1
```

Memory state (15)	
P [.]	[T, A, A, T, G]
m	5
i	**5**
j	2
P [i]	**G**
P [j]	**A**
$fail$ [.]	[0, 1, 1, 0, .]

Knuth-Morris-Pratt Algorithm

- In practice, we only encode the failure edges as an **array**
- Let's work with this (slightly different) example ($\alpha = 4$)
  - The pattern is $P[1..m]$ = TAATG ($m = 5$)
  - The string is $T[1..n]$ = GACCGCGTGAGATAATGTCA
  - The alphabet is [A T G C]
  - P[0] = NULL

```
function ComputeFailure(P[·], m)
    j ← 0
    for i ← 1 to m do
        if P[i] = P[j] then
            fail[i] ← fail[j]
        else
            fail[i] ← j
        while j > 0 and P[i] ≠ P[j] do
            j ← fail[j]
        j ← j + 1
```

16 →

Memory state (16)	
P [.]	[T, A, A, T, G]
m	5
i	5
j	2
P [i]	G
P [j]	A
$fail$ [.]	[0, 1, 1, 0, **2**]

Knuth-Morris-Pratt Algorithm

This loop keeps scanning T →

```
function KnuthMorrisPratt(P[·], m, T[·], n)
    j ← 1
    for i ← 1 to n do
        while j > 0 and T[i] ≠ P[j] do
            j ← fail[j]
        if j = m then
            return i − m + 1
        j ← j + 1
```

Knuth-Morris-Pratt Algorithm

This loop decreases *j* depending of *fail* →

```
function KnuthMorrisPratt(P[·], m, T[·], n)
    j ← 1
    for i ← 1 to n do
        while j > 0 and T[i] ≠ P[j] do
            j ← fail[j]
        if j = m then
            return i − m + 1
        j ← j + 1
```

Knuth-Morris-Pratt Algorithm

This conditional detects success on the search 

```
function KnuthMorrisPratt(P[·], m, T[·], n)
    j ← 1
    for i ← 1 to n do
        while j > 0 and T[i] ≠ P[j] do
            j ← fail[j]
        if j = m then
            return i − m + 1
        j ← j + 1
```

Knuth-Morris-Pratt Algorithm

```
function KnuthMorrisPratt(P[·], m, T[·], n)
    j ← 1
    for i ← 1 to n do
        while j > 0 and T[i] ≠ P[j] do
            j ← fail[j]
        if j = m then
            return i − m + 1
        j ← j + 1
```

This line increases *j* by one position

Knuth-Morris-Pratt Algorithm

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
T[.]	G	A	C	C	G	C	G	T	G	A	G	A	T	A	A	T	G	T	C	A
FAILED (G!=T, SHIFT BY 1)	T	A	A	T	G															

Knuth-Morris-Pratt Algorithm

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
T[.]	G	A	C	C	G	C	G	T	G	A	G	A	T	A	A	T	G	T	C	A
FAILED (G!=T, SHIFT BY 1)	T	A	A	T	G															
FAILED (A!=T, SHIFT BY 1)		T	A	A	T	G														

Knuth-Morris-Pratt Algorithm

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
T[.]	G	A	C	C	G	C	G	T	G	A	G	A	T	A	A	T	G	T	C	A
FAILED (G!=T, SHIFT BY 1)	T	A	A	T	G															
FAILED (A!=T, SHIFT BY 1)		T	A	A	T	G														
FAILED (C!=T, SHIFT BY 1)			T	A	A	T	G													

Knuth-Morris-Pratt Algorithm

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
T[.]	G	A	C	C	G	C	G	T	G	A	G	A	T	A	A	T	G	T	C	A
FAILED (G!=T, SHIFT BY 1)	T	A	A	T	G															
FAILED (A!=T, SHIFT BY 1)		T	A	A	T	G														
FAILED (C!=T, SHIFT BY 1)			T	A	A	T	G													
FAILED (C!=T, SHIFT BY 1)				T	A	A	T	G												

Knuth-Morris-Pratt Algorithm

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
T[.]	G	A	C	C	G	C	G	T	G	A	G	A	T	A	A	T	G	T	C	A
FAILED (G!=T, SHIFT BY 1)	T	A	A	T	G															
FAILED (A!=T, SHIFT BY 1)		T	A	A	T	G														
FAILED (C!=T, SHIFT BY 1)			T	A	A	T	G													
FAILED (C!=T, SHIFT BY 1)				T	A	A	T	G												
FAILED (G!=T, SHIFT BY 1)					T	A	A	T	G											

Knuth-Morris-Pratt Algorithm

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
T[.]	G	A	C	C	G	C	G	T	G	A	G	A	T	A	A	T	G	T	C	A
FAILED (G!=T, SHIFT BY 1)	T	A	A	T	G															
FAILED (A!=T, SHIFT BY 1)		T	A	A	T	G														
FAILED (C!=T, SHIFT BY 1)			T	A	A	T	G													
FAILED (C!=T, SHIFT BY 1)				T	A	A	T	G												
FAILED (G!=T, SHIFT BY 1)					T	A	A	T	G											
FAILED (C!=T, SHIFT BY 1)						T	A	A	T	G										

Knuth-Morris-Pratt Algorithm

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
T[.]	G	A	C	C	G	C	G	T	G	A	G	A	T	A	A	T	G	T	C	A
FAILED (G!=T, SHIFT BY 1)	T	A	A	T	G															
FAILED (A!=T, SHIFT BY 1)		T	A	A	T	G														
FAILED (C!=T, SHIFT BY 1)			T	A	A	T	G													
FAILED (C!=T, SHIFT BY 1)				T	A	A	T	G												
FAILED (G!=T, SHIFT BY 1)					T	A	A	T	G											
FAILED (C!=T, SHIFT BY 1)						T	A	A	T	G										
FAILED (G!=T, SHIFT BY 1)							T	A	A	T	G									

Knuth-Morris-Pratt Algorithm

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
T[.]	G	A	C	C	G	C	G	T	G	A	G	A	T	A	A	T	G	T	C	A
FAILED (G!=T, SHIFT BY 1)	T	A	A	T	G															
FAILED (A!=T, SHIFT BY 1)		T	A	A	T	G														
FAILED (C!=T, SHIFT BY 1)			T	A	A	T	G													
FAILED (C!=T, SHIFT BY 1)				T	A	A	T	G												
FAILED (G!=T, SHIFT BY 1)					T	A	A	T	G											
FAILED (C!=T, SHIFT BY 1)						T	A	A	T	G										
FAILED (G!=T, SHIFT BY 1)							T	A	A	T	G									
FAILED (G!=A, SHIFT BY 1)								T	A	A	T	G								

Knuth-Morris-Pratt Algorithm

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
T[.]	G	A	C	C	G	C	G	T	G	A	G	A	T	A	A	T	G	T	C	A
FAILED (G!=T, SHIFT BY 1)	T	A	A	T	G															
FAILED (A!=T, SHIFT BY 1)		T	A	A	T	G														
FAILED (C!=T, SHIFT BY 1)			T	A	A	T	G													
FAILED (C!=T, SHIFT BY 1)				T	A	A	T	G												
FAILED (G!=T, SHIFT BY 1)					T	A	A	T	G											
FAILED (C!=T, SHIFT BY 1)						T	A	A	T	G										
FAILED (G!=T, SHIFT BY 1)							T	A	A	T	G									
FAILED (G!=A, SHIFT BY 1)								T	A	A	T	G								
FAILED (A!=T, SHIFT BY 2)										T	A	A	T	G						

Knuth-Morris-Pratt Algorithm

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
T[.]	G	A	C	C	G	C	G	T	G	A	G	A	T	A	A	T	G	T	C	A
FAILED (G!=T, SHIFT BY 1)	T	A	A	T	G															
FAILED (A!=T, SHIFT BY 1)		T	A	A	T	G														
FAILED (C!=T, SHIFT BY 1)			T	A	A	T	G													
FAILED (C!=T, SHIFT BY 1)				T	A	A	T	G												
FAILED (G!=T, SHIFT BY 1)					T	A	A	T	G											
FAILED (C!=T, SHIFT BY 1)						T	A	A	T	G										
FAILED (G!=T, SHIFT BY 1)							T	A	A	T	G									
FAILED (G!=A, SHIFT BY 1)								T	A	A	T	G								
FAILED (A!=T, SHIFT BY 2)										T	A	A	T	G						
FAILED (G!=T, SHIFT BY 1)											T	A	A	T	G					

Knuth-Morris-Pratt Algorithm

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
T[.]	G	A	C	C	G	C	G	T	G	A	G	A	T	A	A	T	G	T	C	A
FAILED (G!=T, SHIFT BY 1)	T	A	A	T	G															
FAILED (A!=T, SHIFT BY 1)		T	A	A	T	G														
FAILED (C!=T, SHIFT BY 1)			T	A	A	T	G													
FAILED (C!=T, SHIFT BY 1)				T	A	A	T	G												
FAILED (G!=T, SHIFT BY 1)					T	A	A	T	G											
FAILED (C!=T, SHIFT BY 1)						T	A	A	T	G										
FAILED (G!=T, SHIFT BY 1)							T	A	A	T	G									
FAILED (G!=A, SHIFT BY 1)								T	A	A	T	G								
FAILED (A!=T, SHIFT BY 2)										T	A	A	T	G						
FAILED (G!=T, SHIFT BY 1)											T	A	A	T	G					
FAILED (A!=T, SHIFT BY 1)												T	A	A	T	G				

Knuth-Morris-Pratt Algorithm

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
T[.]	G	A	C	C	G	C	G	T	G	A	G	A	T	A	A	T	G	T	C	A
FAILED (G!=T, SHIFT BY 1)	T	A	A	T	G															
FAILED (A!=T, SHIFT BY 1)		T	A	A	T	G														
FAILED (C!=T, SHIFT BY 1)			T	A	A	T	G													
FAILED (C!=T, SHIFT BY 1)				T	A	A	T	G												
FAILED (G!=T, SHIFT BY 1)					T	A	A	T	G											
FAILED (C!=T, SHIFT BY 1)						T	A	A	T	G										
FAILED (G!=T, SHIFT BY 1)							T	A	A	T	G									
FAILED (G!=A, SHIFT BY 1)								T	A	A	T	G								
FAILED (A!=T, SHIFT BY 2)										T	A	A	T	G						
FAILED (G!=T, SHIFT BY 1)											T	A	A	T	G					
FAILED (A!=T, SHIFT BY 1)												T	A	A	T	G				
FOUND AT $i = 13$													T	A	A	T	G			

Knuth-Morris-Pratt Algorithm

- The **worst-case behavior** of the Knuth-Morris-Pratt Algorithm is $O(n)$
  - At each character comparison, we either **increase** i and j by one, or we **decrease** j by one and leave i alone
  - We can **increment** i at most n-1, so there are at most n-1 **successful comparisons**
  - Since the number of times we **decrease** j cannot exceed the number of times we **increment** j, there can only be n-1 **unsuccessful comparisons**

Next week

- Hashing (Levitin Section 7.3)
  - Horner's rule (Levitin Section 6.5)

- Rabin-Karp Algorithm (http://jeffe.cs.illinois.edu/teaching/algorithms/notes/07-strings.pdf)