Strings

SFWRENG 2CO3: Data Structures and Algorithms

Jelle Hellings

Department of Computing and Software
McMaster University

Winter 2024

Strings over alphabets

An *alphabet* $\mathcal{A}$ is a finite set of distinct symbols.

A *string* over $\mathcal{A}$ is a sequence of symbols taken from $\mathcal{A}$.

Strings over alphabets

An *alphabet* $\mathcal{A}$ is a finite set of distinct symbols.

A *string* over $\mathcal{A}$ is a sequence of symbols taken from $\mathcal{A}$.

Examples

- A typical string over the roman alphabet {'a', . . . , 'z', '␣'}:
  "hello", "hello␣world", "strings␣over␣alphabets", and "".
- A *bit string* is a sequence over $\{0, 1\}$:
  "0011", "1101100", "", and "0".
- A *DNA String* is a sequence over $\{A, C, G, T\}$:
  "", "AACATG", "AGT", and "AAACCCAAATTT".
- A *Unicode string* is a sequence over the unicode code points
  (149 186 symbols and counting).
- A *byte string* is a sequence over *bytes*.

Operations on strings and alphabets

We assume the following basic operations:

- We can sequentially iterate over the symbols in a string.
- We can look up the i-th symbol in a string in $\Theta(1)$.
  (This can be hard in some practical settings: UTF-8 and UTF-16 strings do *not* support this).
- We assume that each alphabet $\mathcal{A}$ is an ordered list L of symbols.
- For each $\sigma \in \mathcal{A}$, we can determine its position in L.

BucketSort: A special-purpose sort

Assumption We have M distinct symbols with values in the range $0, \ldots, M-1$.

BucketSort: A special-purpose sort

Assumption We have M distinct symbols with values in the range $0, \ldots, M-1$.

Algorithm BucketSort(L):

```
buckets := [0 | 0 ≤ i ≤ M − 1].
for all v ∈ L do
    buckets[v] := buckets[v] + 1.
k := 0.
for all i := 0 upto M − 1 do
    for all j := 0 upto buckets[i] do
        L[k] := i.
        k := k + 1.
```

BucketSort: A special-purpose sort

Assumption We have M distinct symbols with values in the range $0, \ldots, M-1$.

Algorithm BucketSort(L):

```
buckets := [0 | 0 ≤ i ≤ M − 1].        ⎫
for all v ∈ L do                        ⎬ Θ(|L|)
    buckets[v] := buckets[v] + 1.       ⎭
k := 0.
for all i := 0 upto M − 1 do
    for all j := 0 upto buckets[i] do
        L[k] := i.
        k := k + 1.
```

BucketSort: A special-purpose sort

Assumption We have M distinct symbols with values in the range $0, \ldots, M-1$.

Algorithm BucketSort(L):

$buckets := [0 \mid 0 \leq i \leq M-1]$.
for all $v \in L$ **do**
  $buckets[v] := buckets[v] + 1$.   } $\Theta(|L|)$
$k := 0$.
for all $i := 0$ upto $M-1$ **do**
  for all $j := 0$ upto $buckets[i]$ **do**
    $L[k] := i$.   } $|L|$
    $k := k + 1$.

BucketSort: A special-purpose sort

Assumption We have M distinct symbols with values in the range $0, \ldots, M-1$.

Algorithm BucketSort(L):

$buckets := [0 \mid 0 \leq i \leq M-1]$.
for all $v \in L$ **do**
  $buckets[v] := buckets[v] + 1$.
$k := 0$.
for all $i := 0$ upto $M-1$ **do**
  for all $j := 0$ upto $buckets[i]$ **do**
    $L[k] := i$.
    $k := k + 1$.

Lines 1–3: $\Theta(|L|)$

Lines 6–8: $|L|$

Lines 5–8: $\Theta(M + |L|)$

BucketSort: A special-purpose sort

Assumption We have M distinct symbols with values in the range $0, \ldots, M-1$.

Algorithm BucketSort(L):

$buckets := [0 \mid 0 \leq i \leq M-1]$. } $\Theta(M)$
for all $v \in L$ **do**
    $buckets[v] := buckets[v] + 1$. } $\Theta(|L|)$
$k := 0$.
for all $i := 0$ upto $M-1$ **do**
    for all $j := 0$ upto $buckets[i]$ **do**
        $L[k] := i$.
        $k := k + 1$. } $|L|$ } $\Theta(M + |L|)$

BucketSort: A special-purpose sort

Assumption We have M distinct symbols with values in the range $0, \ldots, M-1$.

Algorithm BucketSort(L):

$buckets := [0 \mid 0 \leq i \leq M-1]$.
for all $v \in L$ **do**
  $buckets[v] := buckets[v] + 1$.
$k := 0$.
for all $i := 0$ upto $M-1$ **do**
  for all $j := 0$ upto $buckets[i]$ **do**
    $L[k] := i$.
    $k := k + 1$.

$\Theta(M)$ (first line)

$\Theta(|L|)$ (the first **for all** loop)

$|L|$ (the inner **for all** loop)

$\Theta(M + |L|)$ (the outer **for all** loop)

$\Theta(M + |L|)$ (the whole algorithm)

BucketSort: A special-purpose sort

Assumption We have M distinct symbols with values in the range $0, \ldots, M-1$.

Algorithm BucketSort(L):

```
buckets := [0 | 0 ≤ i ≤ M − 1].
for all v ∈ L do
    buckets[v] := buckets[v] + 1.
k := 0.
for all i := 0 upto M − 1 do
    for all j := 0 upto buckets[i] do
        L[k] := i.
        k := k + 1.
```

$L =$ "GAGGATATGTAG".

A:	0
C:	0
G:	0
T:	0

BucketSort: A special-purpose sort

Assumption We have M distinct symbols with values in the range $0, \ldots, M-1$.

Algorithm BucketSort(L):

```
buckets := [0 | 0 ≤ i ≤ M − 1].
for all v ∈ L do
    buckets[v] := buckets[v] + 1.
k := 0.
for all i := 0 upto M − 1 do
    for all j := 0 upto buckets[i] do
        L[k] := i.
        k := k + 1.
```

L = "GAGGATATGTAG".

A:	0
C:	0
G:	1
T:	0

BucketSort: A special-purpose sort

Assumption We have M distinct symbols with values in the range $0, \ldots, M-1$.

Algorithm BucketSort(L):

1: $buckets := [0 \mid 0 \leq i \leq M-1]$.
2: **for all** $v \in L$ **do**
3: $\quad buckets[v] := buckets[v] + 1$.
4: $k := 0$.
5: **for all** $i := 0$ upto $M-1$ **do**
6: $\quad$ **for all** $j := 0$ upto $buckets[i]$ **do**
7: $\quad\quad L[k] := i$.
8: $\quad\quad k := k + 1$.

$L =$ "GAGGATATGTAG".

A:	1
C:	0
G:	1
T:	0

BucketSort: A special-purpose sort

Assumption We have M distinct symbols with values in the range $0, \ldots, M-1$.

Algorithm BucketSort(L):

$buckets := [0 \mid 0 \leq i \leq M-1]$.
for all $v \in L$ **do**
    $buckets[v] := buckets[v] + 1$.
$k := 0$.
for all $i := 0$ upto $M-1$ **do**
    for all $j := 0$ upto $buckets[i]$ **do**
        $L[k] := i$.
        $k := k + 1$.

$L =$ "GAGGATATGTAG".

A:	1
C:	0
G:	2
T:	0

BucketSort: A special-purpose sort

Assumption We have M distinct symbols with values in the range $0, \dots, M-1$.

Algorithm BucketSort(L):

```
buckets := [0 | 0 ≤ i ≤ M − 1].
for all v ∈ L do
    buckets[v] := buckets[v] + 1.
k := 0.
for all i := 0 upto M − 1 do
    for all j := 0 upto buckets[i] do
        L[k] := i.
        k := k + 1.
```

L = "GAGGATATGTAG".

A:	1
C:	0
G:	3
T:	0

BucketSort: A special-purpose sort

Assumption We have M distinct symbols with values in the range $0, \ldots, M-1$.

Algorithm BucketSort(L):

```
buckets := [0 | 0 ≤ i ≤ M − 1].
for all v ∈ L do
    buckets[v] := buckets[v] + 1.
k := 0.
for all i := 0 upto M − 1 do
    for all j := 0 upto buckets[i] do
        L[k] := i.
        k := k + 1.
```

L = "GAGGATATGTAG".

A:	2
C:	0
G:	3
T:	0

BucketSort: A special-purpose sort

Assumption We have M distinct symbols with values in the range $0, \dots, M-1$.

Algorithm BucketSort(L):

```
buckets := [0 | 0 ≤ i ≤ M − 1].
for all v ∈ L do
    buckets[v] := buckets[v] + 1.
k := 0.
for all i := 0 upto M − 1 do
    for all j := 0 upto buckets[i] do
        L[k] := i.
        k := k + 1.
```

$L =$ "GAGGATATGTAG".

A:	2
C:	0
G:	3
T:	1

BucketSort: A special-purpose sort

Assumption We have M distinct symbols with values in the range $0, \ldots, M-1$.

Algorithm BucketSort(L):

```
buckets := [0 | 0 ≤ i ≤ M − 1].
for all v ∈ L do
    buckets[v] := buckets[v] + 1.
k := 0.
for all i := 0 upto M − 1 do
    for all j := 0 upto buckets[i] do
        L[k] := i.
        k := k + 1.
```

$L =$ “GAGGATATGTAG”.

A:	3
C:	0
G:	3
T:	1

BucketSort: A special-purpose sort

Assumption We have M distinct symbols with values in the range $0, \ldots, M-1$.

Algorithm BucketSort(L):

```
1: buckets := [0 | 0 ≤ i ≤ M − 1].
2: for all v ∈ L do
3:     buckets[v] := buckets[v] + 1.
4: k := 0.
5: for all i := 0 upto M − 1 do
6:     for all j := 0 upto buckets[i] do
7:         L[k] := i.
8:         k := k + 1.
```

L = "GAGGATATGTAG".

A:	3
C:	0
G:	3
T:	2

BucketSort: A special-purpose sort

Assumption We have M distinct symbols with values in the range $0, \ldots, M-1$.

Algorithm BucketSort(L):

```
buckets := [0 | 0 ≤ i ≤ M − 1].
for all v ∈ L do
    buckets[v] := buckets[v] + 1.
k := 0.
for all i := 0 upto M − 1 do
    for all j := 0 upto buckets[i] do
        L[k] := i.
        k := k + 1.
```

L = “GAGGATATGTAG”.

A:	3
C:	0
G:	4
T:	2

BucketSort: A special-purpose sort

Assumption We have M distinct symbols with values in the range $0, \ldots, M-1$.

Algorithm BucketSort(L):

```
buckets := [0 | 0 ≤ i ≤ M − 1].
for all v ∈ L do
    buckets[v] := buckets[v] + 1.
k := 0.
for all i := 0 upto M − 1 do
    for all j := 0 upto buckets[i] do
        L[k] := i.
        k := k + 1.
```

$L =$ “GAGGATATGTAG”.

A:	3
C:	0
G:	4
T:	3

BucketSort: A special-purpose sort

Assumption We have M distinct symbols with values in the range $0, \ldots, M-1$.

Algorithm BucketSort(L):

```
buckets := [0 | 0 ≤ i ≤ M − 1].
for all v ∈ L do
    buckets[v] := buckets[v] + 1.
k := 0.
for all i := 0 upto M − 1 do
    for all j := 0 upto buckets[i] do
        L[k] := i.
        k := k + 1.
```

L = "GAGGATATGTAG".

A:	4
C:	0
G:	4
T:	3

BucketSort: A special-purpose sort

Assumption We have M distinct symbols with values in the range $0, \ldots, M-1$.

Algorithm BucketSort(L):

```
buckets := [0 | 0 ≤ i ≤ M − 1].
for all v ∈ L do
    buckets[v] := buckets[v] + 1.
k := 0.
for all i := 0 upto M − 1 do
    for all j := 0 upto buckets[i] do
        L[k] := i.
        k := k + 1.
```

L = “GAGGATATGTAG”.

A:	4
C:	0
G:	5
T:	3

BucketSort: A special-purpose sort

Assumption We have M distinct symbols with values in the range $0, \ldots, M-1$.

Algorithm BucketSort(L):

$buckets := [0 \mid 0 \leq i \leq M-1]$.
for all $v \in L$ **do**
    $buckets[v] := buckets[v] + 1$.
$k := 0$.
for all $i := 0$ upto $M - 1$ **do**
    for all $j := 0$ upto $buckets[i]$ **do**
        $L[k] := i$.
        $k := k + 1$.

$L =$ "AAAAATATGTAG".

A:	4
C:	0
G:	5
T:	3

BucketSort: A special-purpose sort

Assumption We have M distinct symbols with values in the range $0, \ldots, M-1$.

Algorithm BucketSort(L):

```
buckets := [0 | 0 ≤ i ≤ M − 1].
for all v ∈ L do
    buckets[v] := buckets[v] + 1.
k := 0.
for all i := 0 upto M − 1 do
    for all j := 0 upto buckets[i] do
        L[k] := i.
        k := k + 1.
```

$L =$ "AAAAATATGTAG".

A:	4
C:	0
G:	5
T:	3

BucketSort: A special-purpose sort

Assumption We have M distinct symbols with values in the range $0, \ldots, M-1$.

Algorithm BucketSort(L):

```
buckets := [0 | 0 ≤ i ≤ M − 1].
for all v ∈ L do
    buckets[v] := buckets[v] + 1.
k := 0.
for all i := 0 upto M − 1 do
    for all j := 0 upto buckets[i] do
        L[k] := i.
        k := k + 1.
```

$L =$ “AAAAGGGGGTAG”.

A:	4
C:	0
G:	5
T:	3

BucketSort: A special-purpose sort

Assumption We have M distinct symbols with values in the range $0, \ldots, M-1$.

Algorithm BucketSort(L):

```
buckets := [0 | 0 ≤ i ≤ M − 1].
for all v ∈ L do
    buckets[v] := buckets[v] + 1.
k := 0.
for all i := 0 upto M − 1 do
    for all j := 0 upto buckets[i] do
        L[k] := i.
        k := k + 1.
```

$L =$ “AAAAGGGGGTTT”.

A:	4
C:	0
G:	5
T:	3

BucketSort: A special-purpose sort

Assumption We have M distinct symbols with values in the range $0, \ldots, M-1$.

Algorithm GBucketSort(L, r):

```
buckets := [[ ] | 0 ≤ i ≤ M − 1].
for all v ∈ L do
    Append v to buckets[r(v)].
k := 0.
for all i := 0 upto M − 1 do
    for all j := 0 upto |buckets[i]| do
        L[k] := buckets[i][j].
        k := k + 1.
```

Generalization

Assume we have values that "represent" $0, \ldots, M-1$ via some function r.

BucketSort: A special-purpose sort

Assumption We have M distinct symbols with values in the range $0, \ldots, M-1$.

Algorithm GBucketSort(L, r):

```
buckets := [[ ] | 0 ≤ i ≤ M − 1].
for all v ∈ L do
    Append v to buckets[r(v)].
k := 0.
for all i := 0 upto M − 1 do
    for all j := 0 upto |buckets[i]| do
        L[k] := buckets[i][j].
        k := k + 1.
```

Generalization

Assume we have values that "represent" $0, \ldots, M-1$ via some function r.

Notice that GBucketSort is *stable*.

RadixSort: A special-purpose sort for strings

Assumption. We have strings of length k over alphabet $\mathcal{A}$.

RadixSort: A special-purpose sort for strings

Assumption. We have strings of length k over alphabet $\mathcal{A}$.

Algorithm RadixSort(L):

```
for d := k − 1 downto 0 do
    Stable-sort L on the d-th string symbols.
```

RadixSort: A special-purpose sort for strings

Assumption. We have strings of length k over alphabet $\mathcal{A}$.

Algorithm RadixSort(L):

```
for d := k − 1 downto 0 do
    Stable-sort L on the d-th string symbols.
    GBucketSort(L, r_d) with r_d(S) = S[d].
```

RadixSort: A special-purpose sort for strings

Assumption. We have strings of length k over alphabet $\mathcal{A}$.

Algorithm RadixSort(L):

for $d := k - 1$ downto 0 **do**
  Stable-sort L on the d-th string symbols.
  GBucketSort(L, r_d) with $r_d(S) = S[d]$.

(lines 2–3: $\Theta(|L| + |\mathcal{A}|)$)

RadixSort: A special-purpose sort for strings

Assumption. We have strings of length k over alphabet $\mathcal{A}$.

Algorithm RadixSort(L):

for $d := k - 1$ downto 0 **do**
  Stable-sort L on the d-th string symbols.
  GBucketSort(L, r_d) with $r_d(S) = S[d]$.

$$\left.\begin{array}{l}\text{Stable-sort } L \text{ on the } d\text{-th string symbols.} \\ \text{GBucketSort}(L, r_d) \text{ with } r_d(S) = S[d].\end{array}\right\} \Theta(|L| + |\mathcal{A}|)$$

Whole loop: $\Theta(k(|L| + |\mathcal{A}|))$

RadixSort: A special-purpose sort for strings

Assumption. We have strings of length k over alphabet $\mathcal{A}$.

Algorithm RadixSort(L):

1: **for** $d := k - 1$ downto 0 **do**
2: Stable-sort L on the d-th string symbols.
3: GBucketSort(L, r_d) with $r_d(S) = S[d]$.

RadixSort: A special-purpose sort for strings

Assumption. We have strings of length k over alphabet $\mathcal{A}$.

Algorithm RadixSort(L):

1: **for** $d := k - 1$ downto 0 **do**
2: Stable-sort L on the d-th string symbols.
3: GBucketSort(L, r_d) with $r_d(S) = S[d]$.

L = ["AGCTCT",
"ATTAAC",
"GCGCGG",
"GGCGCG",
"TCTATG",
"TCACCG",
"AGCTGA",
"ATCTAA",
"GTCTGC",
"TGGACG"]

RadixSort: A special-purpose sort for strings

Assumption. We have strings of length k over alphabet $\mathcal{A}$.

Algorithm RadixSort(L):

1: **for** $d := k - 1$ downto 0 **do**
2: Stable-sort L on the d-th string symbols.
3: GBucketSort(L, r_d) with $r_d(S) = S[d]$.

L = ["AGCTCT", "ATTAAC", "GCGCGG", "GGCGCG", "TCTATG", "TCACCG", "AGCTGA", "ATCTAA", "GTCTGC", "TGGACG"]

$\longrightarrow$

L = ["AGCTGA", "ATCTAA", "GTCTGC", "ATTAAC", "GCGCGG", "GGCGCG", "TCTATG", "TCACCG", "TGGACG", "AGCTCT"]

RadixSort: A special-purpose sort for strings

Assumption. We have strings of length k over alphabet $\mathcal{A}$.

Algorithm RadixSort(L):

for $d := k - 1$ downto 0 **do**
  Stable-sort L on the d-th string symbols.
  GBucketSort(L, r_d) with $r_d(S) = S[d]$.

L = [“AGCTGA”, “ATCTAA”, “GTCTGC”, “ATTAAC”, “GCGCGG”, “GGCGCG”, “TCTATG”, “TCACCG”, “TGGACG”, “AGCTCT”]

$\longrightarrow$

L = [“ATCTAA”, “ATTAAC”, “GGCGCG”, “TCACCG”, “TGGACG”, “AGCTCT”, “AGCTGA”, “GTCTGC”, “GCGCGG”, “TCTATG”]

RadixSort: A special-purpose sort for strings

Assumption. We have strings of length k over alphabet $\mathcal{A}$.

Algorithm RadixSort(L):

1: **for** $d := k - 1$ downto 0 **do**
2: Stable-sort L on the d-th string symbols.
3: GBucketSort(L, r_d) with $r_d(S) = S[d]$.

L = [“ATCTAA”, “ATTAAC”, “GGCGCG”, “TCACCG”, “TGGACG”, “AGCTCT”, “AGCTGA”, “GTCTGC”, “GCGCGG”, “TCTATG”]

$\longrightarrow$

L = [“ATTAAC”, “TGGACG”, “TCTATG”, “TCACCG”, “GCGCGG”, “GGCGCG”, “ATCTAA”, “AGCTCT”, “AGCTGA”, “GTCTGC”]

RadixSort: A special-purpose sort for strings

Assumption. We have strings of length k over alphabet $\mathcal{A}$.

Algorithm RadixSort(L):

1: **for** $d := k - 1$ downto 0 **do**
2: Stable-sort L on the d-th string symbols.
3: GBucketSort(L, r_d) with $r_d(S) = S[d]$.

L = ["ATTAAC", "TGGACG", "TCTATG", "TCACCG", "GCGCGG", "GGCGCG", "ATCTAA", "AGCTCT", "AGCTGA", "GTCTGC"]

→

L = ["TCACCG", "GGCGCG", "ATCTAA", "AGCTCT", "AGCTGA", "GTCTGC", "TGGACG", "GCGCGG", "ATTAAC", "TCTATG"]

RadixSort: A special-purpose sort for strings

Assumption. We have strings of length k over alphabet $\mathcal{A}$.

Algorithm RadixSort(L):

for $d := k - 1$ downto 0 **do**
  Stable-sort L on the d-th string symbols.
  GBucketSort(L, r_d) with $r_d(S) = S[d]$.

L = ["TCACCG",		L = ["TCACCG",
"GGCGCG",		"GCGCGG",
"ATCTAA",		"TCTATG",
"AGCTCT",		"GGCGCG",
"AGCTGA",		"AGCTCT",
"GTCTGC",	→	"AGCTGA",
"TGGACG",		"TGGACG",
"GCGCGG",		"ATCTAA",
"ATTAAC",		"GTCTGC",
"TCTATG"]		"ATTAAC"]

RadixSort: A special-purpose sort for strings

Assumption. We have strings of length k over alphabet $\mathcal{A}$.

Algorithm RadixSort(L):

1: **for** $d := k - 1$ downto 0 **do**

2: Stable-sort L on the d-th string symbols.

3: GBucketSort(L, r_d) with $r_d(S) = S[d]$.

L = [“TCACCG”, “GCGCGG”, “TCTATG”, “GGCGCG”, “AGCTCT”, “AGCTGA”, “TGGACG”, “ATCTAA”, “GTCTGC”, “ATTAAC”]

→

L = [“AGCTCT”, “AGCTGA”, “ATCTAA”, “ATTAAC”, “GCGCGG”, “GGCGCG”, “GTCTGC”, “TCACCG”, “TCTATG”, “TGGACG”]

RadixSort: A special-purpose sort for strings

Assumption. We have strings of length k over alphabet $\mathcal{A}$.

Algorithm RadixSort(L):

1: **for** $d := k - 1$ downto 0 **do**
2: Stable-sort L on the d-th string symbols.
3: GBucketSort(L, r_d) with $r_d(S) = S[d]$.

L = ["AGCTCT",
"AGCTGA",
"ATCTAA",
"ATTAAC",
"GCGCGG",
"GGCGCG",
"GTCTGC",
"TCACCG",
"TCTATG",
"TGGACG"].

RadixSort: A special-purpose sort for strings

Assumption. We have strings of length k over alphabet $\mathcal{A}$.

Algorithm RadixSort(L):

```
for d := k − 1 downto 0 do
    Stable-sort L on the d-th string symbols.
    GBucketSort(L, r_d) with r_d(S) = S[d].
```

RadixSort: A special-purpose sort for strings

Assumption. We have strings of length k over alphabet $\mathcal{A}$.

Algorithm RadixSort(L):

for $d := k - 1$ downto 0 **do**
    Stable-sort L on the d-th string symbols.
    GBucketSort(L, r_d) with $r_d(S) = S[d]$.

Correctness

RadixSort: A special-purpose sort for strings

Assumption. We have strings of length k over alphabet $\mathcal{A}$.

Algorithm RadixSort(L):

1: **for** $d := k - 1$ downto 0 **do**
2: Stable-sort L on the d-th string symbols.
3: GBucketSort(L, r_d) with $r_d(S) = S[d]$.

Correctness

Invariant: In L, the suffix of the last $k - (d + 1)$ symbols is sorted.

RadixSort: A special-purpose sort for strings

Assumption. We have strings of length k over alphabet $\mathcal{A}$.

Algorithm RadixSort(L):

for $d := k - 1$ downto 0 **do**
    Stable-sort L on the d-th string symbols.
    GBucketSort(L, r_d) with $r_d(S) = S[d]$.

Correctness

Invariant: In L, the suffix of the last $k - (d + 1)$ *symbols is sorted*.

Generalization: strings with variable lengths up-to-k

RadixSort: A special-purpose sort for strings

Assumption. We have strings of length k over alphabet $\mathcal{A}$.

Algorithm RadixSort(L):

1: **for** $d := k - 1$ downto 0 **do**

2: Stable-sort L on the d-th string symbols.

3: GBucketSort(L, r_d) with $r_d(S) = S[d]$.

Correctness

Invariant: In L, the suffix of the last $k - (d + 1)$ symbols is sorted.

Generalization: strings with variable lengths up-to-k

Let S be a string of length $|S| < k$.

Interpret $S[|S|], \ldots, S[k-1]$ as symbols that come before all other symbols.

The book calls this *least-significant-digit string sort*.

RadixSort: A special-purpose sort for strings

Assumption. We have strings of length k over alphabet $\mathcal{A}$.

Algorithm RadixSort(L):

1: **for** $d := k - 1$ downto 0 **do**
2: Stable-sort L on the d-th string symbols.
3: GBucketSort(L, r_d) with $r_d(S) = S[d]$.

Is RadixSort worth it?

RadixSort: A special-purpose sort for strings

Assumption. We have strings of length k over alphabet $\mathcal{A}$.

Algorithm RadixSort(L):

1: **for** $d := k - 1$ downto 0 **do**
2: Stable-sort L on the d-th string symbols.
3: GBucketSort(L, r_d) with $r_d(S) = S[d]$.

Is RadixSort worth it?

- Optimal sorts perform $\Theta(|L| \log(|L|))$ *comparisons*.
- Comparing two strings of length k costs at-most $\Theta(k)$.

RadixSort: A special-purpose sort for strings

Assumption. We have strings of length k over alphabet $\mathcal{A}$.

Algorithm RadixSort(L):

```
for d := k − 1 downto 0 do
    Stable-sort L on the d-th string symbols.
    GBucketSort(L, r_d) with r_d(S) = S[d].
```

Is RadixSort worth it?

- Optimal sorts perform $\Theta(|L|\log(|L|))$ *comparisons*.
- Comparing two strings of length k costs at-most $\Theta(k)$.
- For $|L|$ *random strings*, comparisons are expected to cost $\Theta(\log_2(|L|))$.

RadixSort: A special-purpose sort for strings

Assumption. We have strings of length k over alphabet $\mathcal{A}$.

Algorithm RadixSort(L):

1: **for** $d := k - 1$ downto 0 **do**
2: Stable-sort L on the d-th string symbols.
3: GBucketSort(L, r_d) with $r_d(S) = S[d]$.

Is RadixSort worth it?

- Optimal sorts perform $\Theta(|L|\log(|L|))$ *comparisons*.
- Comparing two strings of length k costs at-most $\Theta(k)$.
- For $|L|$ *random strings*, comparisons are expected to cost $\Theta(\log_2(|L|))$.

$\Theta(k(|L| + |\mathcal{A}|))$ versus $\Theta(k|L|\log(|L|))$ (or $\Theta(|L|\log^2(|L|))$ expected).

Most-significant-digit string sort

RadixSort does not try to minimize the number of sorting rounds:
if k is the length of the longest string in L, then RadixSort "reorders" the list k times.

Most-significant-digit string sort

RadixSort does not try to minimize the number of sorting rounds:
if k is the length of the longest string in L, then RadixSort "reorders" the list k times.

Consider GBucketSort(L, r_0):

L = ["ATTAAC",		L = ["ATTAAC",
"GCGCGG",		"AGCTGA",
"GGCGCG",		"ATCTAA",
"TCTATG",		"GCGCGG",
"TCACCG",	→	"GGCGCG",
"AGCTGA",		"GTCTGC",
"ATCTAA",		"TCTATG",
"GTCTGC",		"TCACCG",
"TGGACG"]		"TGGACG"].

Most-significant-digit string sort

RadixSort does not try to minimize the number of sorting rounds:
if k is the length of the longest string in L, then RadixSort "reorders" the list k times.

Consider GBucketSort(L, r_0):

L = ["ATTAAC",		L = ["ATTAAC",
"GCGCGG",		"AGCTGA",
"GGCGCG",		"ATCTAA",
"TCTATG",		"GCGCGG",
"TCACCG",	$\rightarrow$	"GGCGCG",
"AGCTGA",		"GTCTGC",
"ATCTAA",		"TCTATG",
"GTCTGC",		"TCACCG",
"TGGACG"]		"TGGACG"].

Most-significant-digit string sort

RADIXSORT does not try to minimize the number of sorting rounds:
if k is the length of the longest string in L, then RADIXSORT "reorders" the list k times.

Consider GBUCKETSORT(L, r_0):

L = ["ATTAAC",		L = ["ATTAAC",
"GCGCGG",		"AGCTGA",
"GGCGCG",		"ATCTAA",
"TCTATG",		"GCGCGG",
"TCACCG",	→	"GGCGCG",
"AGCTGA",		"GTCTGC",
"ATCTAA",		"TCTATG",
"GTCTGC",		"TCACCG",
"TGGACG"]		"TGGACG"].

Most-significant-digit string sort

RadixSort does not try to minimize the number of sorting rounds:
if k is the length of the longest string in L, then RadixSort "reorders" the list k times.

Consider GBucketSort(L, r_0).

After *constructing* the buckets with respect to r_0,
we only need to resort the individual buckets.

Most-significant-digit string sort

RadixSort does not try to minimize the number of sorting rounds:
if k is the length of the longest string in L, then RadixSort "reorders" the list k times.

Consider GBucketSort(L, r_0).

After *constructing* the buckets with respect to r_0,
we only need to resort the individual buckets.

We will apply this idea *recursively*!

Most-significant-digit string sort

RadixSort does not try to minimize the number of sorting rounds:
if k is the length of the longest string in L, then RadixSort "reorders" the list k times.

Consider GBucketSort(L, r_0).

After *constructing* the buckets with respect to r_0,
we only need to resort the individual buckets.

We will apply this idea *recursively*!

Algorithm MSD-Sort(L, d):

1: **if** $d < k$ **and** $|L| > 1$ **then**
2: GBucketSort(L, r_d) with $r_d(S) = S[d]$,
during which we further sort each
individual bucket separately.

Most-significant-digit string sort

Algorithm MSD-Sort(L, d):

if $d < k$ **and** $|L| > 1$ **then**
    $buckets := [[\,] \mid 0 \le i \le |\mathcal{A}| - 1]$.
    for all $v \in L$ **do**
        Append v to $buckets[v[d]]$.
    $k := 0$.
    for all $i := 0$ upto $|\mathcal{A}| - 1$ **do**
        $k_{\text{start}} := k$.
        for all $j := 0$ upto $|buckets[i]|$ **do**
            $L[k] := buckets[i][j]$.
            $k := k + 1$.
        MSD-Sort($L[k_{\text{start}} \ldots k)$, $d + 1$).

Most-significant-digit string sort

Algorithm MSD-SORT(L, d):

```
if d < k and |L| > 1 then
    buckets := [[ ] | 0 ≤ i ≤ |𝒜| − 1].
    for all v ∈ L do
        Append v to buckets[v[d]].
    k := 0.
    for all i := 0 upto |𝒜| − 1 do
        k_start := k.
        for all j := 0 upto |buckets[i]| do
            L[k] := buckets[i][j].
            k := k + 1.
        MSD-SORT(L[k_start ... k), d + 1).
```

L = [“ATTAAC”,
“GCGCGG”,
“GGCGCG”,
“TCTATG”,
“TCACCG”,
“AGCTGA”,
“ATCTAA”,
“GTCTGC”,
“TGGACG”].

Most-significant-digit string sort

Algorithm MSD-SORT(L, d):

if $d < k$ **and** $|L| > 1$ **then**
  $buckets := [[\ ] \mid 0 \leq i \leq |\mathcal{A}| - 1]$.
  for all $v \in L$ **do**
    Append v to $buckets[v[d]]$.
  $k := 0$.
  for all $i := 0$ upto $|\mathcal{A}| - 1$ **do**
    $k_{\text{start}} := k$.
    for all $j := 0$ upto $|buckets[i]|$ **do**
      $L[k] := buckets[i][j]$.
      $k := k + 1$.
    MSD-SORT($L[k_{\text{start}} \ldots k)$, $d + 1$).

L = ["ATTAAC",
"AGCTGA",
"ATCTAA",
"TCTATG",
"TCACCG",
"AGCTGA",
"ATCTAA",
"GTCTGC",
"TGGACG"].

Most-significant-digit string sort

Algorithm MSD-SORT(L, d):

if $d < k$ **and** $|L| > 1$ **then**
  $buckets := [[\ ] \mid 0 \leq i \leq |\mathcal{A}| - 1]$.
  for all $v \in L$ **do**
    Append v to $buckets[v[d]]$.
  $k := 0$.
  for all $i := 0$ upto $|\mathcal{A}| - 1$ **do**
    $k_{\text{start}} := k$.
    for all $j := 0$ upto $|buckets[i]|$ **do**
      $L[k] := buckets[i][j]$.
      $k := k + 1$.
    MSD-SORT($L[k_{\text{start}} \ldots k)$, $d + 1$).

L = ["ATTAAC",
"AGCTGA",
"ATCTAA",
"TCTATG",
"TCACCG",
"AGCTGA",
"ATCTAA",
"GTCTGC",
"TGGACG"].

Most-significant-digit string sort

Algorithm MSD-SORT(L, d):

if $d < k$ **and** $|L| > 1$ **then**
  $buckets := [[\ ] \mid 0 \leq i \leq |\mathcal{A}| - 1]$.
  for all $v \in L$ **do**
    Append v to $buckets[v[d]]$.
  $k := 0$.
  for all $i := 0$ upto $|\mathcal{A}| - 1$ **do**
    $k_{\text{start}} := k$.
    for all $j := 0$ upto $|buckets[i]|$ **do**
      $L[k] := buckets[i][j]$.
      $k := k + 1$.
    MSD-SORT($L[k_{\text{start}} \ldots k)$, $d + 1$).

L = ["AGCTGA",
"AGCTGA",
"ATCTAA",
"TCTATG",
"TCACCG",
"AGCTGA",
"ATCTAA",
"GTCTGC",
"TGGACG"].

Most-significant-digit string sort

Algorithm MSD-SORT(L, d):

```
if d < k and |L| > 1 then
    buckets := [[ ] | 0 ≤ i ≤ |𝒜| − 1].
    for all v ∈ L do
        Append v to buckets[v[d]].
    k := 0.
    for all i := 0 upto |𝒜| − 1 do
        k_start := k.
        for all j := 0 upto |buckets[i]| do
            L[k] := buckets[i][j].
            k := k + 1.
        MSD-SORT(L[k_start ... k), d + 1).
```

L = ["AGCTGA",
"AGCTGA",
"ATCTAA",
"TCTATG",
"TCACCG",
"AGCTGA",
"ATCTAA",
"GTCTGC",
"TGGACG"].

Most-significant-digit string sort

Algorithm MSD-Sort(L, d):

- **if** $d < k$ **and** $|L| > 1$ **then**
  - $buckets := [[\ ] \mid 0 \leq i \leq |\mathcal{A}| - 1]$.
  - **for all** $v \in L$ **do**
    - Append v to $buckets[v[d]]$.
  - $k := 0$.
  - **for all** $i := 0$ upto $|\mathcal{A}| - 1$ **do**
    - $k_{\text{start}} := k$.
    - **for all** $j := 0$ upto $|buckets[i]|$ **do**
      - $L[k] := buckets[i][j]$.
      - $k := k + 1$.
    - MSD-Sort($L[k_{\text{start}} \ldots k)$, $d + 1$).

L = ["AGCTGA",
"AGCTGA",
"ATCTAA",
"TCTATG",
"TCACCG",
"AGCTGA",
"ATCTAA",
"GTCTGC",
"TGGACG"].

Most-significant-digit string sort

Algorithm MSD-Sort(L, d):

if $d < k$ **and** $|L| > 1$ **then**
    $buckets := [[\ ] \mid 0 \leq i \leq |\mathcal{A}| - 1]$.
    for all $v \in L$ **do**
        Append v to $buckets[v[d]]$.
    $k := 0$.
    for all $i := 0$ upto $|\mathcal{A}| - 1$ **do**
        $k_{\text{start}} := k$.
        for all $j := 0$ upto $|buckets[i]|$ **do**
            $L[k] := buckets[i][j]$.
            $k := k + 1$.
        MSD-Sort($L[k_{\text{start}} \ldots k)$, $d + 1$).

L = [“AGCTGA”,
“ATTAAC”,
“ATCTAA”,
“TCTATG”,
“TCACCG”,
“AGCTGA”,
“ATCTAA”,
“GTCTGC”,
“TGGACG”].

Most-significant-digit string sort

Algorithm MSD-SORT(L, d):

if $d < k$ **and** $|L| > 1$ **then**
    $buckets := [[\,] \mid 0 \leq i \leq |\mathcal{A}| - 1]$.
    for all $v \in L$ **do**
        Append v to $buckets[v[d]]$.
    $k := 0$.
    for all $i := 0$ upto $|\mathcal{A}| - 1$ **do**
        $k_{\text{start}} := k$.
        for all $j := 0$ upto $|buckets[i]|$ **do**
            $L[k] := buckets[i][j]$.
            $k := k + 1$.
        MSD-SORT($L[k_{\text{start}} \ldots k)$, $d + 1$).

L = ["AGCTGA",
"ATTAAC",
"ATCTAA",
"TCTATG",
"TCACCG",
"AGCTGA",
"ATCTAA",
"GTCTGC",
"TGGACG"].

Most-significant-digit string sort

Algorithm MSD-SORT(L, d):

if $d < k$ **and** $|L| > 1$ **then**
    $buckets := [[\,] \mid 0 \le i \le |\mathcal{A}| - 1]$.
    for all $v \in L$ **do**
        Append v to $buckets[v[d]]$.
    $k := 0$.
    for all $i := 0$ upto $|\mathcal{A}| - 1$ **do**
        $k_{\text{start}} := k$.
        for all $j := 0$ upto $|buckets[i]|$ **do**
            $L[k] := buckets[i][j]$.
            $k := k + 1$.
        MSD-SORT($L[k_{\text{start}} \ldots k)$, $d + 1$).

L = ["AGCTGA",
"ATCTAA",
"ATCTAA",
"TCTATG",
"TCACCG",
"AGCTGA",
"ATCTAA",
"GTCTGC",
"TGGACG"].

Most-significant-digit string sort

Algorithm MSD-SORT(L, d):

if $d < k$ **and** $|L| > 1$ **then**
    $buckets := [[\ ] \mid 0 \le i \le |\mathcal{A}| - 1]$.
    for all $v \in L$ **do**
        Append v to $buckets[v[d]]$.
    $k := 0$.
    for all $i := 0$ upto $|\mathcal{A}| - 1$ **do**
        $k_{\text{start}} := k$.
        for all $j := 0$ upto $|buckets[i]|$ **do**
            $L[k] := buckets[i][j]$.
            $k := k + 1$.
        MSD-SORT($L[k_{\text{start}} \ldots k)$, $d + 1$).

L = [“AGCTGA”,
“ATCTAA”,
“ATCTAA”,
“TCTATG”,
“TCACCG”,
“AGCTGA”,
“ATCTAA”,
“GTCTGC”,
“TGGACG”].

Most-significant-digit string sort

Algorithm MSD-Sort(L, d):

if $d < k$ **and** $|L| > 1$ **then**
  $buckets := [[\,] \mid 0 \le i \le |\mathcal{A}| - 1]$.
  for all $v \in L$ **do**
    Append v to $buckets[v[d]]$.
  $k := 0$.
  for all $i := 0$ upto $|\mathcal{A}| - 1$ **do**
    $k_{\text{start}} := k$.
    for all $j := 0$ upto $|buckets[i]|$ **do**
      $L[k] := buckets[i][j]$.
      $k := k + 1$.
    MSD-Sort($L[k_{\text{start}} \ldots k)$, $d + 1$).

L = ["AGCTGA",
"ATCTAA",
"ATCTAA",
"TCTATG",
"TCACCG",
"AGCTGA",
"ATCTAA",
"GTCTGC",
"TGGACG"].

Most-significant-digit string sort

Algorithm MSD-SORT(L, d):

if $d < k$ **and** $|L| > 1$ **then**
  $buckets := [[\,] \mid 0 \le i \le |\mathcal{A}| - 1]$.
  for all $v \in L$ **do**
    Append v to $buckets[v[d]]$.
  $k := 0$.
  for all $i := 0$ upto $|\mathcal{A}| - 1$ **do**
    $k_{\text{start}} := k$.
    for all $j := 0$ upto $|buckets[i]|$ **do**
      $L[k] := buckets[i][j]$.
      $k := k + 1$.
    MSD-SORT($L[k_{\text{start}} \ldots k)$, $d + 1$).

L = [“AGCTGA”,
“ATCTAA”,
“ATTAAC”,
“TCTATG”,
“TCACCG”,
“AGCTGA”,
“ATCTAA”,
“GTCTGC”,
“TGGACG”].

Most-significant-digit string sort

Algorithm MSD-Sort(L, d):

if $d < k$ **and** $|L| > 1$ **then**
  $buckets := [[\,] \mid 0 \leq i \leq |\mathcal{A}| - 1]$.
  for all $v \in L$ **do**
    Append v to $buckets[v[d]]$.
  $k := 0$.
  for all $i := 0$ upto $|\mathcal{A}| - 1$ **do**
    $k_{\text{start}} := k$.
    for all $j := 0$ upto $|buckets[i]|$ **do**
      $L[k] := buckets[i][j]$.
      $k := k + 1$.
    MSD-Sort($L[k_{\text{start}} \ldots k)$, $d + 1$).

L = [“AGCTGA”,
“ATCTAA”,
“ATTAAC”,
“TCTATG”,
“TCACCG”,
“AGCTGA”,
“ATCTAA”,
“GTCTGC”,
“TGGACG”].

Most-significant-digit string sort

Algorithm MSD-SORT(L, d):

```
if d < k and |L| > 1 then
    buckets := [[ ] | 0 ≤ i ≤ |𝒜| − 1].
    for all v ∈ L do
        Append v to buckets[v[d]].
    k := 0.
    for all i := 0 upto |𝒜| − 1 do
        k_start := k.
        for all j := 0 upto |buckets[i]| do
            L[k] := buckets[i][j].
            k := k + 1.
        MSD-Sort(L[k_start ... k), d + 1).
```

L = [“AGCTGA”,
“ATCTAA”,
“ATTAAC”,
“TCTATG”,
“TCACCG”,
“AGCTGA”,
“ATCTAA”,
“GTCTGC”,
“TGGACG”].

Most-significant-digit string sort

Algorithm MSD-Sort(L, d):

```
if d < k and |L| > 1 then
    buckets := [[ ] | 0 ≤ i ≤ |𝒜| − 1].
    for all v ∈ L do
        Append v to buckets[v[d]].
    k := 0.
    for all i := 0 upto |𝒜| − 1 do
        k_start := k.
        for all j := 0 upto |buckets[i]| do
            L[k] := buckets[i][j].
            k := k + 1.
        MSD-Sort(L[k_start ... k), d + 1).
```

L = [“AGCTGA”,
“ATCTAA”,
“ATTAAC”,
“GCGCGG”,
“GGCGCG”,
“GTCTGC”,
“ATCTAA”,
“GTCTGC”,
“TGGACG”].

Most-significant-digit string sort

Algorithm MSD-Sort(L, d):

if $d < k$ **and** $|L| > 1$ **then**
  $buckets := [[\,]\mid 0 \leq i \leq |\mathcal{A}| - 1]$.
  for all $v \in L$ **do**
    Append v to $buckets[v[d]]$.
  $k := 0$.
  for all $i := 0$ upto $|\mathcal{A}| - 1$ **do**
    $k_{\text{start}} := k$.
    for all $j := 0$ upto $|buckets[i]|$ **do**
      $L[k] := buckets[i][j]$.
      $k := k + 1$.
    MSD-Sort($L[k_{\text{start}} \ldots k)$, $d + 1$).

L = ["AGCTGA",
"ATCTAA",
"ATTAAC",
"GCGCGG",
"GGCGCG",
"GTCTGC",
"ATCTAA",
"GTCTGC",
"TGGACG"].

Most-significant-digit string sort

Algorithm MSD-Sort(L, d):

```
if d < k and |L| > 1 then
    buckets := [[ ] | 0 ≤ i ≤ |𝒜| − 1].
    for all v ∈ L do
        Append v to buckets[v[d]].
    k := 0.
    for all i := 0 upto |𝒜| − 1 do
        k_start := k.
        for all j := 0 upto |buckets[i]| do
            L[k] := buckets[i][j].
            k := k + 1.
        MSD-Sort(L[k_start ... k), d + 1).
```

L = [“AGCTGA”,
“ATCTAA”,
“ATTAAC”,
“GCGCGG”,
“GGCGCG”,
“GTCTGC”,
“TGGACG”,
“TCTATG”,
“TCACCG”].

Most-significant-digit string sort

Algorithm MSD-SORT(L, d):

```
if d < k and |L| > 1 then
    buckets := [[ ] | 0 ≤ i ≤ |𝒜| − 1].
    for all v ∈ L do
        Append v to buckets[v[d]].
    k := 0.
    for all i := 0 upto |𝒜| − 1 do
        k_start := k.
        for all j := 0 upto |buckets[i]| do
            L[k] := buckets[i][j].
            k := k + 1.
        MSD-SORT(L[k_start ... k), d + 1).
```

L = ["AGCTGA",
"ATCTAA",
"ATTAAC",
"GCGCGG",
"GGCGCG",
"GTCTGC",
"TCACCG",
"TCTATG",
"TGGACG"].

Most-significant-digit string sort

Algorithm MSD-Sort(L, d):

if $d < k$ **and** $|L| > 1$ **then**
  $buckets := [[\,] \mid 0 \leq i \leq |\mathcal{A}| - 1]$.
  for all $v \in L$ **do**
    Append v to $buckets[v[d]]$.
  $k := 0$.
  for all $i := 0$ upto $|\mathcal{A}| - 1$ **do**
    $k_{\text{start}} := k$.
    for all $j := 0$ upto $|buckets[i]|$ **do**
      $L[k] := buckets[i][j]$.
      $k := k + 1$.
    MSD-Sort($L[k_{\text{start}} \ldots k)$, $d + 1$).

Finetuning

We end up with many arrays *buckets* that each hold $|\mathcal{A}|$ lists!

Most-significant-digit string sort

Algorithm MSD-SORT(L, d):

```
if |L| ≤ |𝒜| then
    Use another algorithm to sort L.
else if d < k and |L| > 1 then
    buckets := [[ ] | 0 ≤ i ≤ |𝒜| − 1].
    for all v ∈ L do
        Append v to buckets[v[d]].
    k := 0.
    for all i := 0 upto |𝒜| − 1 do
        k_start := k.
        for all j := 0 upto |buckets[i]| do
            L[k] := buckets[i][j].
            k := k + 1.
        MSD-Sort(L[k_start ... k), d + 1).
```

Finetuning

We end up with many arrays *buckets* that each hold $|\mathcal{A}|$ lists!

Most-significant-digit string sort

Algorithm MSD-SORT(L, d):

if $|L| \leq |\mathcal{A}|$ **then**
    Use another algorithm to sort L.
else if $d < k$ **and** $|L| > 1$ **then**
    $buckets := [[\ ] \mid 0 \leq i \leq |\mathcal{A}| - 1]$.
    for all $v \in L$ **do**
        Append v to $buckets[v[d]]$.
    $k := 0$.
    for all $i := 0$ upto $|\mathcal{A}| - 1$ **do**
        $k_{\text{start}} := k$.
        for all $j := 0$ upto $|buckets[i]|$ **do**
            $L[k] := buckets[i][j]$.
            $k := k + 1$.
        MSD-SORT($L[k_{\text{start}} \ldots k)$, $d + 1$).

Finetuning

We end up with many arrays *buckets* that each hold $|\mathcal{A}|$ lists!

Complexity

At-most $\Theta(k(|L| + |\mathcal{A}|))$.

Sorting: Best practices

So which sort algorithm is the best?

Sorting: Best practices

So which sort algorithm is the best?

Depends on the to-be-sorted input.

Sorting: Best practices

So which sort algorithm is the best?

Depends on the to-be-sorted input.

Often, your standard sort algorithm will be sufficient.

Tries: special-purpose sets and dictionaries

Assumption We have an alphabet $\mathcal{A}$ with $M = |\mathcal{A}|$ symbols.

A *Trie* is a set representation that can hold strings over $\mathcal{A}$ such that:

- strings of length N can be *added* in $\Theta(N)$;
- strings of length N can be *removed* in $\Theta(N)$;
- *checking* whether a string of length N is in the set costs $\Theta(N)$;
- one can efficiently *iterate* over all strings in the set (in sorted order).

Tries: special-purpose sets and dictionaries

Assumption We have an alphabet $\mathcal{A}$ with $M = |\mathcal{A}|$ symbols.

A *Trie* is a set representation that can hold strings over $\mathcal{A}$ such that:

- strings of length N can be *added* in $\Theta(N)$;
- strings of length N can be *removed* in $\Theta(N)$;
- *checking* whether a string of length N is in the set costs $\Theta(N)$;
- one can efficiently *iterate* over all strings in the set (in sorted order).

We have seen tries with $\mathcal{A} = \{0, 1\} \rightarrow$ BSSet in Example Assignment 3.

Tries: special-purpose sets and dictionaries

Assumption We have an alphabet $\mathcal{A}$ with $M = |\mathcal{A}|$ symbols.

Each node n in a *trie* T over $\mathcal{A} = \{\sigma_1, \dots, \sigma_M\}$ has

- a flag $n.end$ that is `true` if the node n represents a string in T;
- at-most M edges to children labeled $\sigma_1, \dots, \sigma_M$.

Tries: special-purpose sets and dictionaries

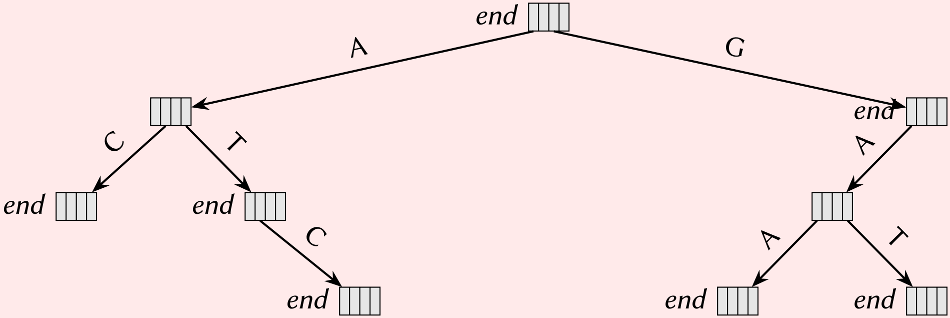
Assumption We have an alphabet $\mathcal{A}$ with $M = |\mathcal{A}|$ symbols.

Each node n in a *trie* T over $\mathcal{A} = \{\sigma_1, \dots, \sigma_M\}$ has

- a flag $n.end$ that is `true` if the node n represents a string in T;
- at-most M edges to children labeled $\sigma_1, \dots, \sigma_M$.

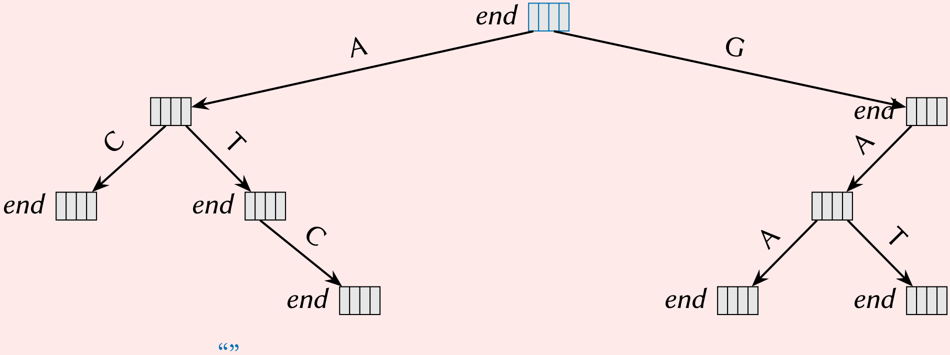
Tries: special-purpose sets and dictionaries

Assumption We have an alphabet $\mathcal{A}$ with $M = |\mathcal{A}|$ symbols.



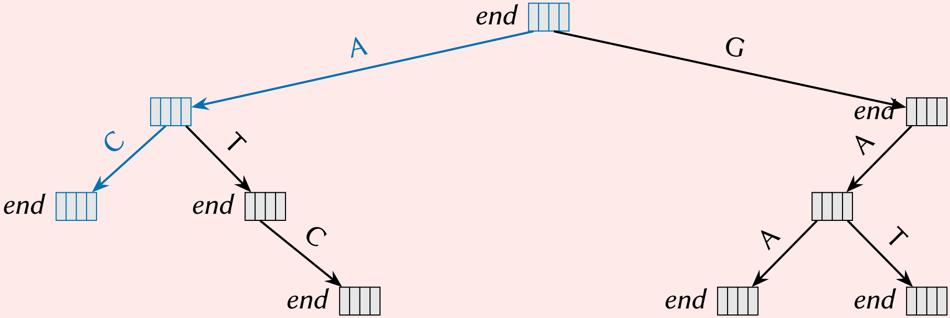
Tries: special-purpose sets and dictionaries

Assumption We have an alphabet $\mathcal{A}$ with $M = |\mathcal{A}|$ symbols.



Tries: special-purpose sets and dictionaries

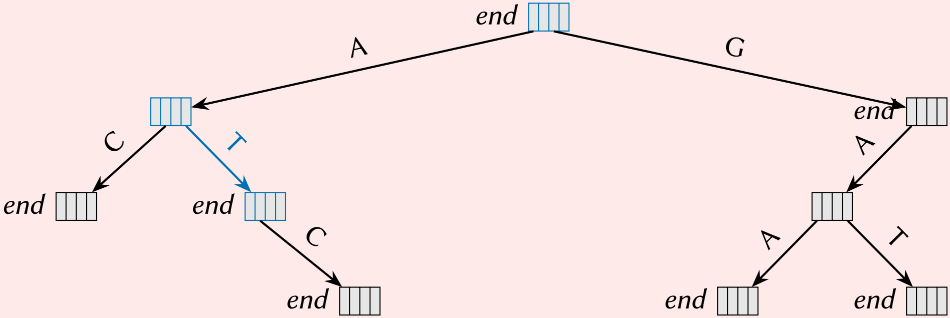
Assumption We have an alphabet $\mathcal{A}$ with $M = |\mathcal{A}|$ symbols.



“”, “AC”

Tries: special-purpose sets and dictionaries

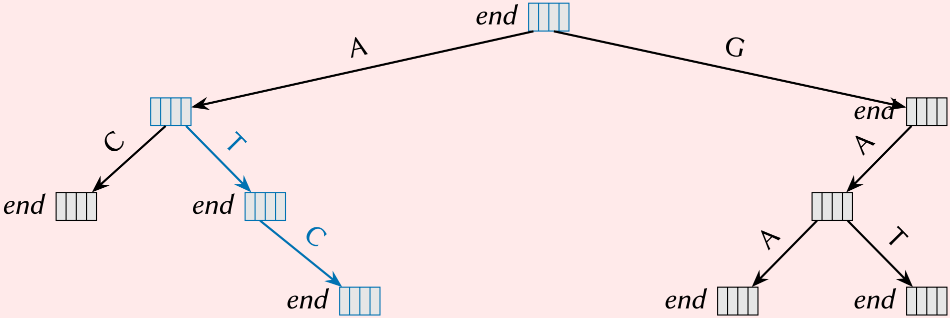
Assumption We have an alphabet $\mathcal{A}$ with $M = |\mathcal{A}|$ symbols.



"", "AC", "AT"

Tries: special-purpose sets and dictionaries

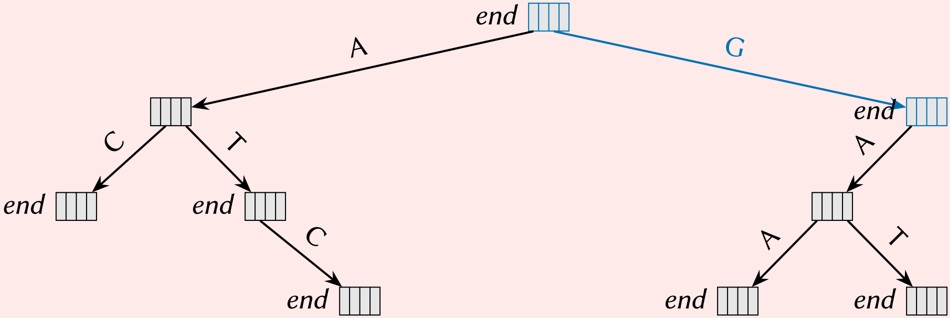
Assumption We have an alphabet $\mathcal{A}$ with $M = |\mathcal{A}|$ symbols.



“”, “AC”, “AT”, “ATC”

Tries: special-purpose sets and dictionaries

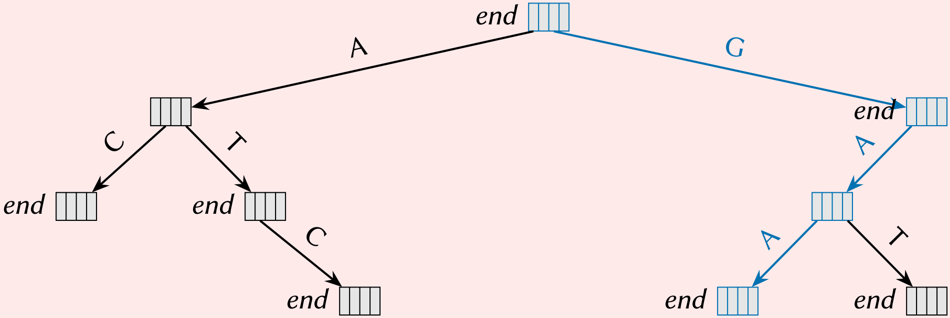
Assumption We have an alphabet $\mathcal{A}$ with $M = |\mathcal{A}|$ symbols.



"", "AC", "AT", "ATC", "G"

Tries: special-purpose sets and dictionaries

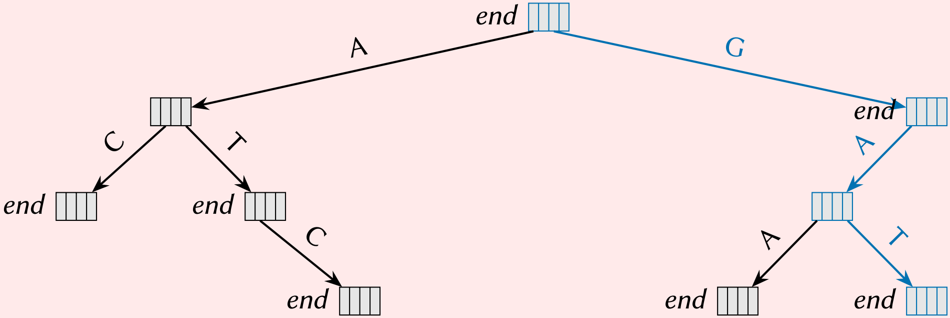
Assumption We have an alphabet $\mathcal{A}$ with $M = |\mathcal{A}|$ symbols.



"", "AC", "AT", "ATC", "G", "GAA"

Tries: special-purpose sets and dictionaries

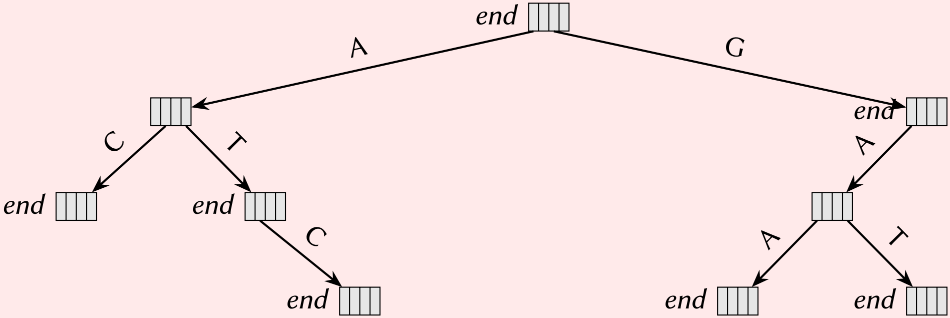
Assumption We have an alphabet $\mathcal{A}$ with $M = |\mathcal{A}|$ symbols.



"", "AC", "AT", "ATC", "G", "GAA", "GAT".

Tries: special-purpose sets and dictionaries

Assumption We have an alphabet $\mathcal{A}$ with $M = |\mathcal{A}|$ symbols.

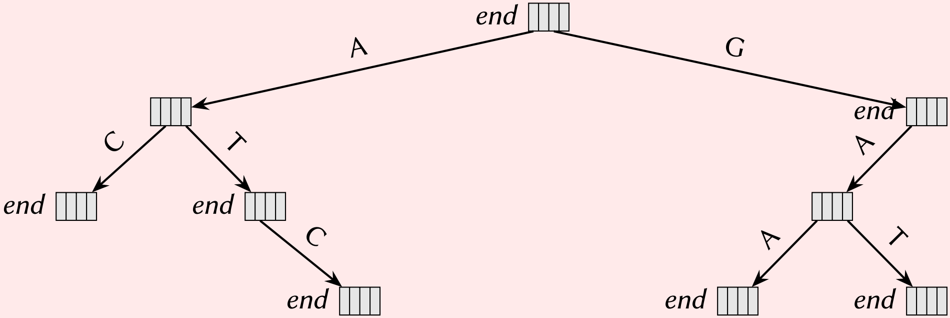


Adding a string

- Follow-or-make a path according to the string symbols.
- Set *n.end* on the last node *n* on this path.

Tries: special-purpose sets and dictionaries

Assumption We have an alphabet $\mathcal{A}$ with $M = |\mathcal{A}|$ symbols.

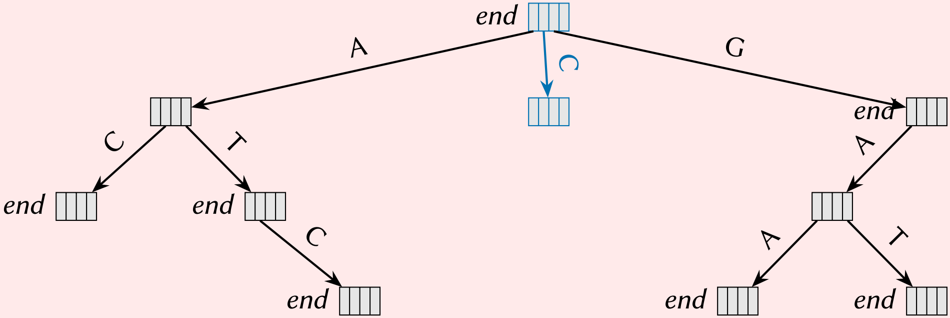


Adding a string "CAT"

- Follow-or-make a path according to the string symbols.
- Set *n.end* on the last node *n* on this path.

Tries: special-purpose sets and dictionaries

Assumption We have an alphabet $\mathcal{A}$ with $M = |\mathcal{A}|$ symbols.

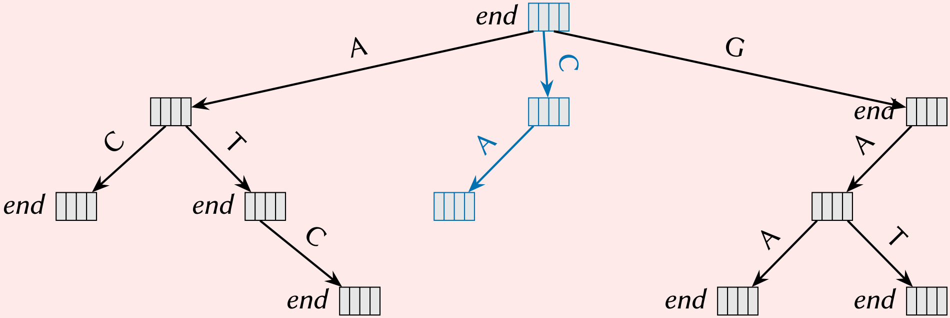


Adding a string "CAT"

- Follow-or-make a path according to the string symbols.
- Set *n.end* on the last node *n* on this path.

Tries: special-purpose sets and dictionaries

Assumption We have an alphabet $\mathcal{A}$ with $M = |\mathcal{A}|$ symbols.

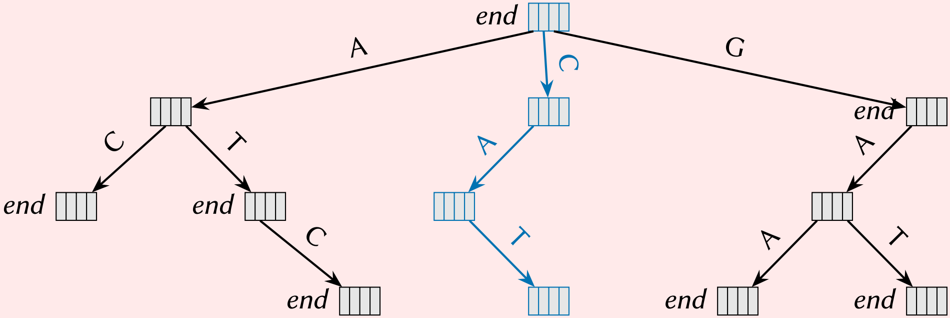


Adding a string "CAT"

- Follow-or-make a path according to the string symbols.
- Set *n.end* on the last node *n* on this path.

Tries: special-purpose sets and dictionaries

Assumption We have an alphabet $\mathcal{A}$ with $M = |\mathcal{A}|$ symbols.

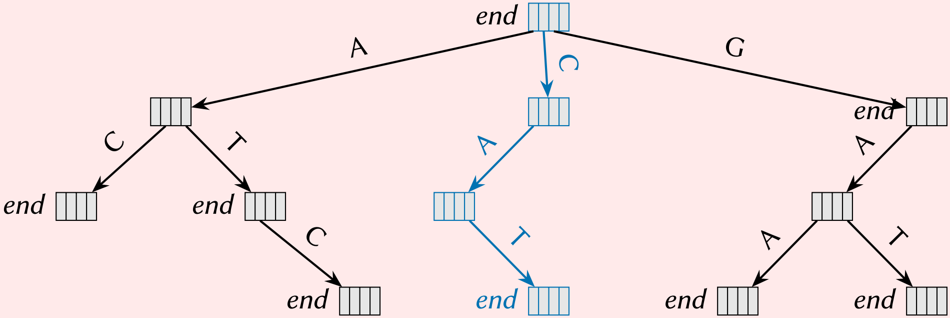


Adding a string "CAT"

- Follow-or-make a path according to the string symbols.
- Set *n.end* on the last node *n* on this path.

Tries: special-purpose sets and dictionaries

Assumption We have an alphabet $\mathcal{A}$ with $M = |\mathcal{A}|$ symbols.

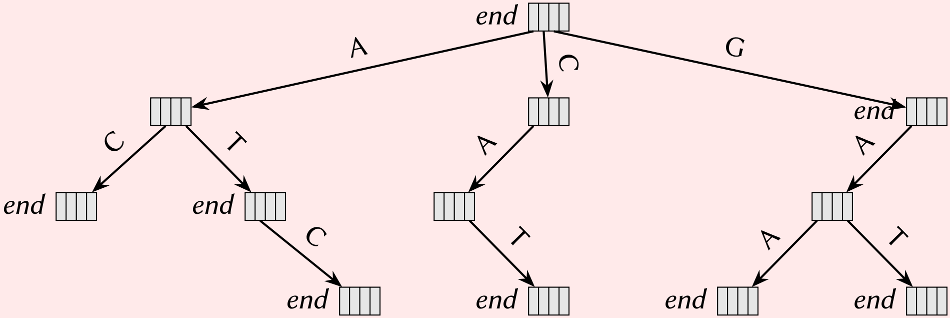


Adding a string "CAT"

- Follow-or-make a path according to the string symbols.
- Set *n.end* on the last node *n* on this path.

Tries: special-purpose sets and dictionaries

Assumption We have an alphabet $\mathcal{A}$ with $M = |\mathcal{A}|$ symbols.

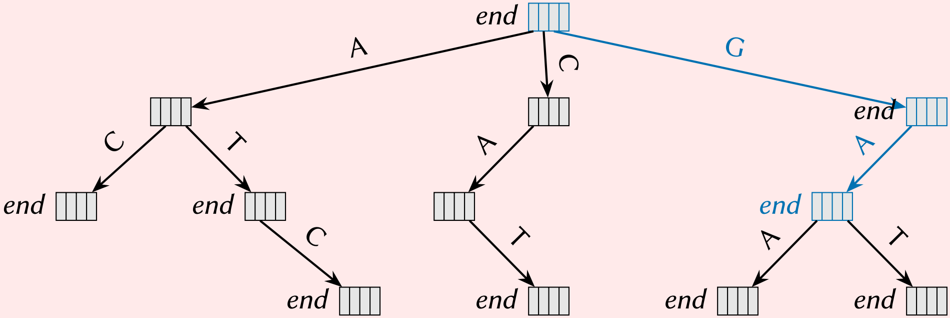


Adding a string "GA"

- Follow-or-make a path according to the string symbols.
- Set *n.end* on the last node *n* on this path.

Tries: special-purpose sets and dictionaries

Assumption We have an alphabet $\mathcal{A}$ with $M = |\mathcal{A}|$ symbols.

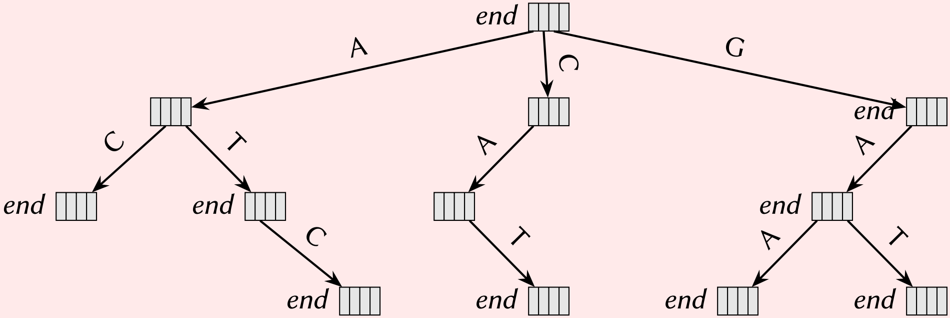


Adding a string "GA"

- Follow-or-make a path according to the string symbols.
- Set *n.end* on the last node *n* on this path.

Tries: special-purpose sets and dictionaries

Assumption We have an alphabet $\mathcal{A}$ with $M = |\mathcal{A}|$ symbols.

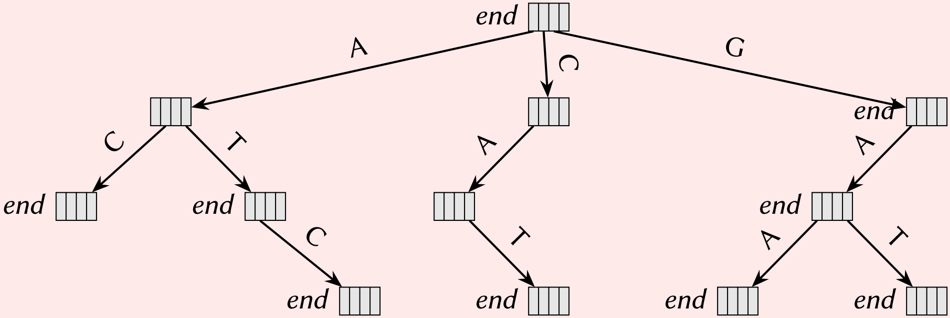


Removing a string

- Follow a path according to the string symbols to node n and unset $n.end$.
- Remove n if n has no children.
- Recurse to the ancestors m: remove m if m has no children and $m.end$ is unset.

Tries: special-purpose sets and dictionaries

Assumption We have an alphabet $\mathcal{A}$ with $M = |\mathcal{A}|$ symbols.

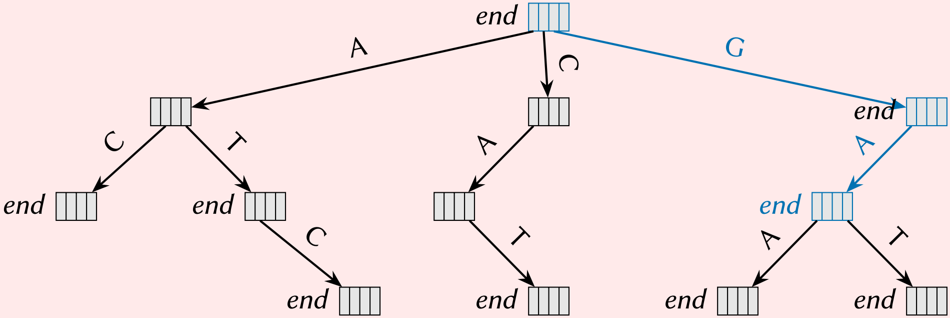


Removing a string "GA"

- Follow a path according to the string symbols to node n and unset $n.end$.
- Remove n if n has no children.
- Recurse to the ancestors m: remove m if m has no children and $m.end$ is unset.

Tries: special-purpose sets and dictionaries

Assumption We have an alphabet $\mathcal{A}$ with $M = |\mathcal{A}|$ symbols.

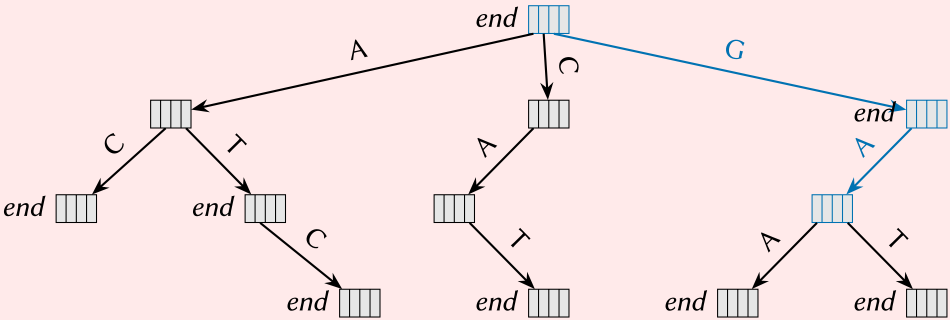


Removing a string "GA"

- Follow a path according to the string symbols to node n and unset $n.end$.
- Remove n if n has no children.
- Recurse to the ancestors m: remove m if m has no children and $m.end$ is unset.

Tries: special-purpose sets and dictionaries

Assumption We have an alphabet $\mathcal{A}$ with $M = |\mathcal{A}|$ symbols.

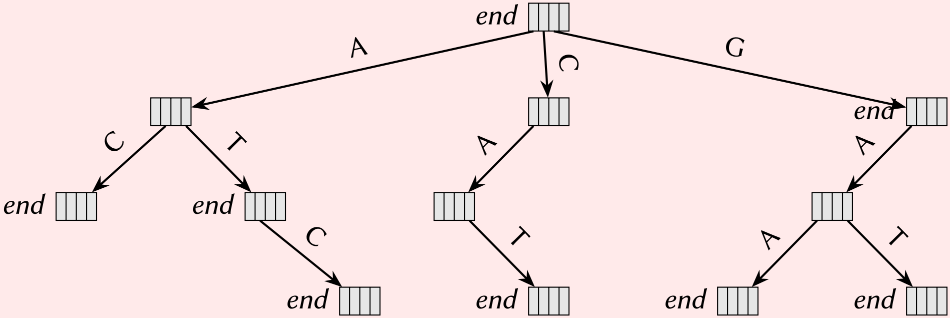


Removing a string "GA"

- Follow a path according to the string symbols to node n and unset $n.end$.
- Remove n if n has no children.
- Recurse to the ancestors m: remove m if m has no children and $m.end$ is unset.

Tries: special-purpose sets and dictionaries

Assumption We have an alphabet $\mathcal{A}$ with $M = |\mathcal{A}|$ symbols.

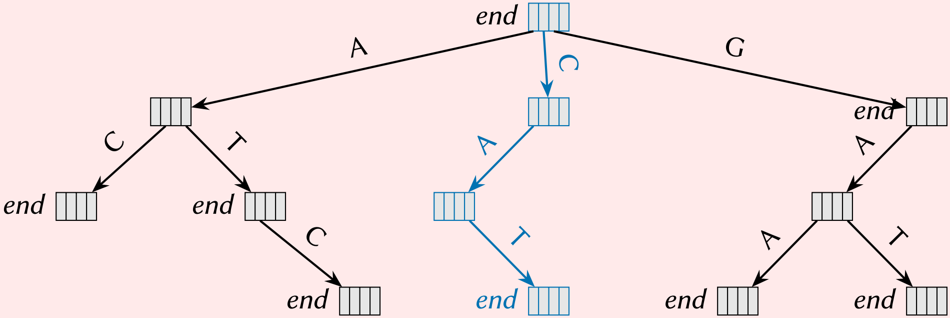


Removing a string "CAT"

- Follow a path according to the string symbols to node *n* and unset *n.end*.
- Remove *n* if *n* has no children.
- Recurse to the ancestors *m*: remove *m* if *m* has no children and *m.end* is unset.

Tries: special-purpose sets and dictionaries

Assumption We have an alphabet $\mathcal{A}$ with $M = |\mathcal{A}|$ symbols.

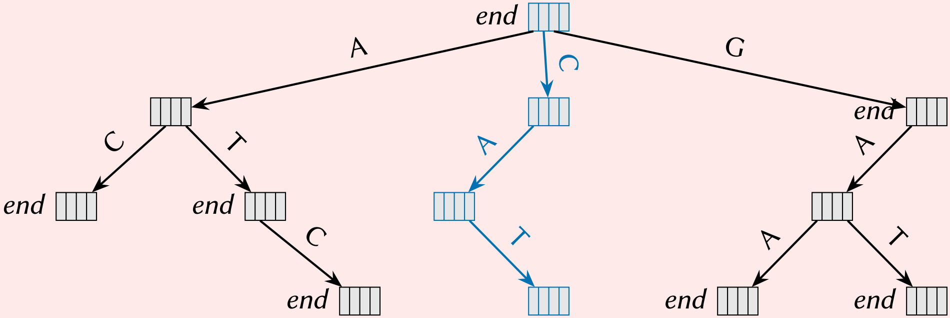


Removing a string "CAT"

- Follow a path according to the string symbols to node n and unset $n.end$.
- Remove n if n has no children.
- Recurse to the ancestors m: remove m if m has no children and $m.end$ is unset.

Tries: special-purpose sets and dictionaries

Assumption We have an alphabet $\mathcal{A}$ with $M = |\mathcal{A}|$ symbols.

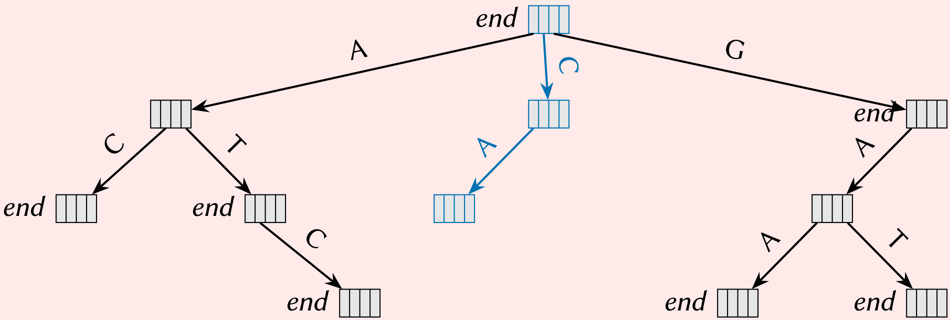


Removing a string "CAT"

- Follow a path according to the string symbols to node n and unset $n.end$.
- Remove n if n has no children.
- Recurse to the ancestors m: remove m if m has no children and $m.end$ is unset.

Tries: special-purpose sets and dictionaries

Assumption We have an alphabet $\mathcal{A}$ with $M = |\mathcal{A}|$ symbols.

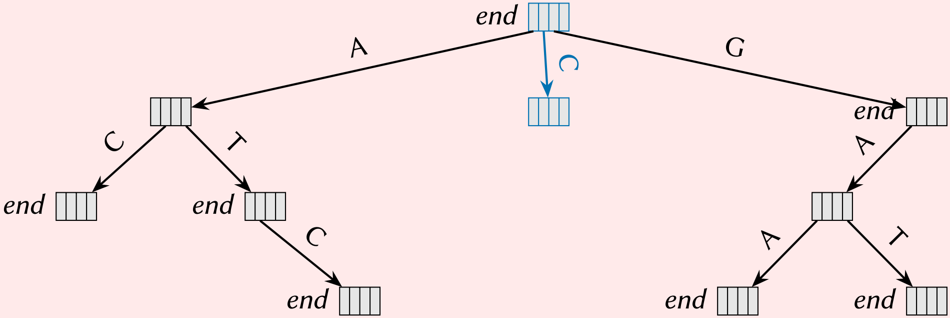


Removing a string "CAT"

- Follow a path according to the string symbols to node *n* and unset *n.end*.
- Remove *n* if *n* has no children.
- Recurse to the ancestors *m*: remove *m* if *m* has no children and *m.end* is unset.

Tries: special-purpose sets and dictionaries

Assumption We have an alphabet $\mathcal{A}$ with $M = |\mathcal{A}|$ symbols.

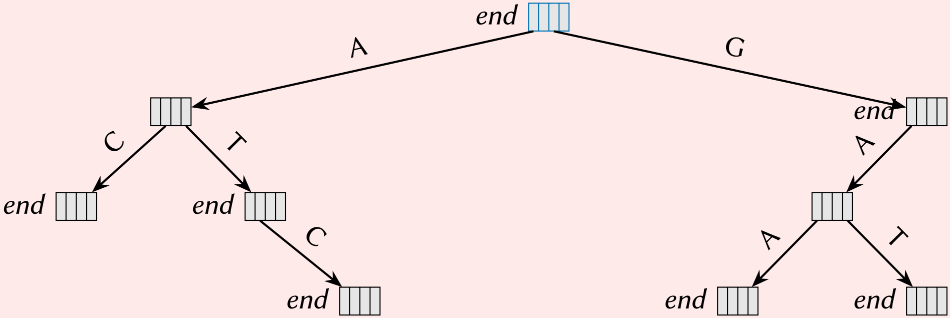


Removing a string "CAT"

- Follow a path according to the string symbols to node *n* and unset *n.end*.
- Remove *n* if *n* has no children.
- Recurse to the ancestors *m*: remove *m* if *m* has no children and *m.end* is unset.

Tries: special-purpose sets and dictionaries

Assumption We have an alphabet $\mathcal{A}$ with $M = |\mathcal{A}|$ symbols.

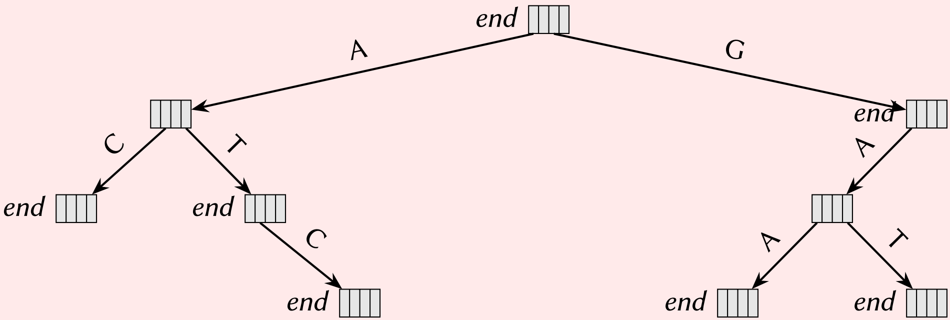


Removing a string "CAT"

- Follow a path according to the string symbols to node *n* and unset *n.end*.
- Remove *n* if *n* has no children.
- Recurse to the ancestors *m*: remove *m* if *m* has no children and *m.end* is unset.

Tries: special-purpose sets and dictionaries

Assumption We have an alphabet $\mathcal{A}$ with $M = |\mathcal{A}|$ symbols.



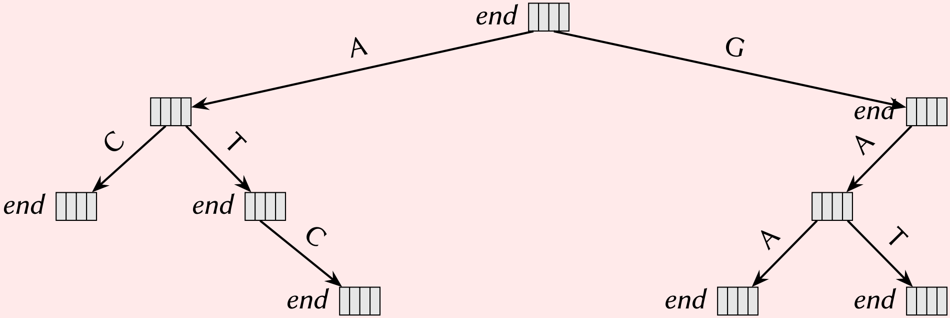
Printing all strings in-order

Perform a pre-order traversal starting at the root. For each node n:

- print the path from root to node n if $n.end$ is set;
- pre-order traverse all children in-order of alphabet symbols.

Tries: special-purpose sets and dictionaries

Assumption We have an alphabet $\mathcal{A}$ with $M = |\mathcal{A}|$ symbols.

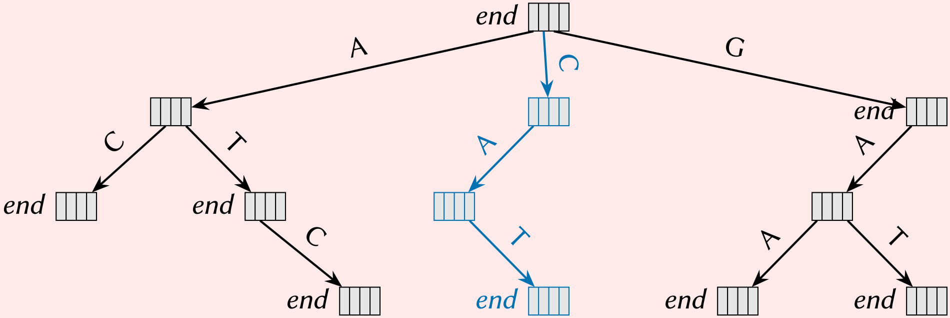


Printing all strings in-order with prefix W

- Follow a path according to the string symbols of W to node m.
- Perform a pre-order traversal starting at the node m.

Tries: special-purpose sets and dictionaries

Assumption We have an alphabet $\mathcal{A}$ with $M = |\mathcal{A}|$ symbols.

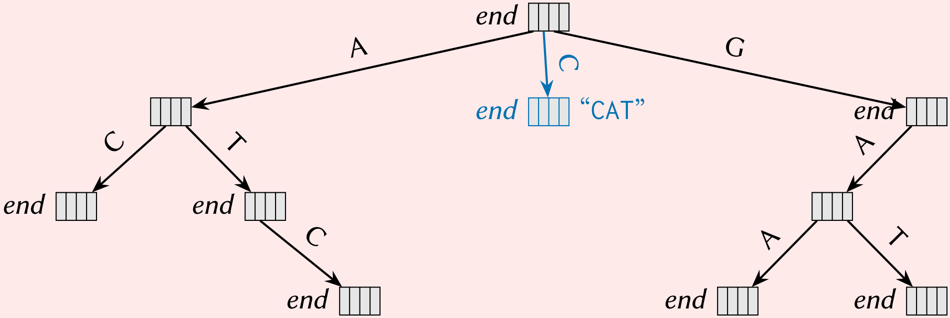


Finetuning

- To deal with big alphabets:
  use a dictionary with $\mathcal{A}$-symbols as keys at each node to store all edges.

Tries: special-purpose sets and dictionaries

Assumption We have an alphabet $\mathcal{A}$ with $M = |\mathcal{A}|$ symbols.



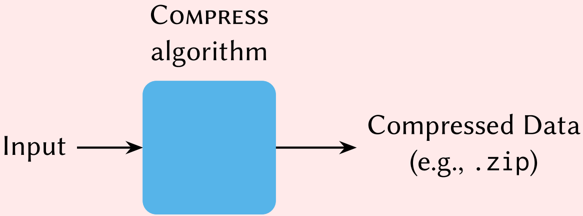
Finetuning

- To deal with big alphabets:
  use a dictionary with $\mathcal{A}$-symbols as keys at each node to store all edges.
- To *compress* non-branching paths: nodes can represent strings of symbols.

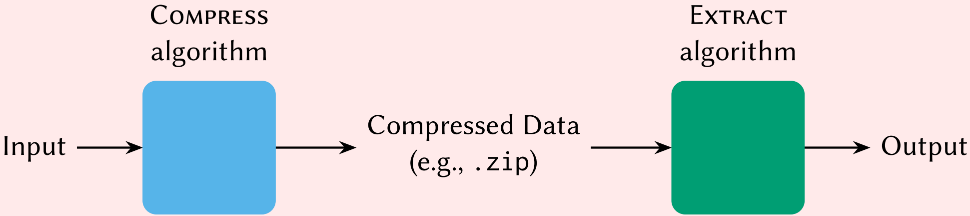
Data compression

Input

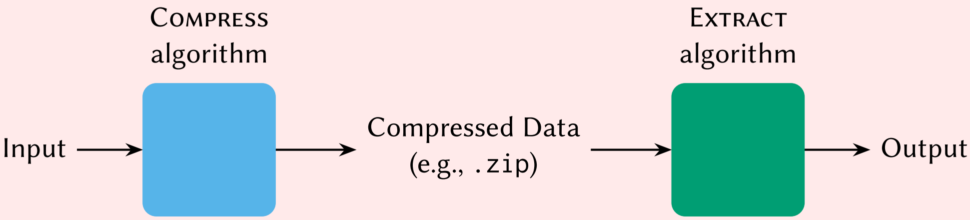
Data compression



Data compression



Data compression



Lossless compression: The **input** must be equivalent to the **output**!

Limits of compression

Theorem

No algorithm A can compress every input I.

Limits of compression

Theorem

No algorithm A can compress every input I into compressed data $A(I)$ with $|A(I)| < |I|$.

Limits of compression

Theorem

No algorithm A can compress every input I into compressed data $A(I)$ with $|A(I)| < |I|$.

Proof

Limits of compression

Theorem

No algorithm A can compress every input I into compressed data $A(I)$ with $|A(I)| < |I|$.

Proof

Intuituin 1 If A can compress every *input*, then $|A(A(I))| < |A(I)|$.

Limits of compression

Theorem

No algorithm A can compress every input I into compressed data A(I) with $|A(I)| < |I|$.

Proof

Intuituin 1 If A can compress every *input*, then $|\mathrm{A}(\mathrm{A}(I))| < |\mathrm{A}(I)|$.
Repeaded application of A on the *input* will eventually lead to zero bits!

Limits of compression

Theorem

No algorithm A can compress every input I into compressed data A(I) with $|A(I)| < |I|$.

Proof

Intuituin 1 If A can compress every *input*, then $|\mathrm{A}(\mathrm{A}(I))| < |\mathrm{A}(I)|$.
Repeaded application of A on the *input* will eventually lead to zero bits!

Intuition 2 Consider all possible *inputs* of N bits: we have 2^N distinct values.

Limits of compression

Theorem

No algorithm A can compress every input I into compressed data A(I) with $|A(I)| < |I|$.

Proof

Intuituin 1 If A can compress every *input*, then $|\mathrm{A}(\mathrm{A}(I))| < |\mathrm{A}(I)|$.
Repeaded application of A on the *input* will eventually lead to zero bits!

Intuition 2 Consider all possible *inputs* of N bits: we have 2^N distinct values.
We need at-least 2^N distinct *outputs* of A on these inputs.

Limits of compression

Theorem

No algorithm A can compress every input I into compressed data $A(I)$ with $|A(I)| < |I|$.

Proof

Intuituin 1 If A can compress every *input*, then $|\mathrm{A}(\mathrm{A}(I))| < |\mathrm{A}(I)|$.
Repeaded application of A on the *input* will eventually lead to zero bits!

Intuition 2 Consider all possible *inputs* of N bits: we have 2^N distinct values.
We need at-least 2^N distinct *outputs* of A on these inputs.

- We can compress at-most 2^M values to a size of $M < N$ bits.

Limits of compression

Theorem

No algorithm A can compress every input I into compressed data A(I) with $|A(I)| < |I|$.

Proof

Intuituin 1 If A can compress every *input*, then $|\mathrm{A}(\mathrm{A}(I))| < |\mathrm{A}(I)|$.
Repeaded application of A on the *input* will eventually lead to zero bits!

Intuition 2 Consider all possible *inputs* of N bits: we have 2^N distinct values.
We need at-least 2^N distinct *outputs* of A on these inputs.

- We can compress at-most 2^M values to a size of $M < N$ bits.
- We can compress *a small fraction* $\frac{2^M}{2^N} = 2^{M-N}$ of all inputs to M bits.
  E.g., we can compress at-most 2^{32} out of 2^{64} values from 8 B to 4 B.

Limits of compression

Theorem

No algorithm A can compress every input I into compressed data $A(I)$ with $|A(I)| < |I|$.

Proof

Intuituin 1 If A can compress every *input*, then $|\mathrm{A}(\mathrm{A}(I))| < |\mathrm{A}(I)|$.
Repeaded application of A on the *input* will eventually lead to zero bits!

Intuition 2 Consider all possible *inputs* of N bits: we have 2^N distinct values.
We need at-least 2^N distinct *outputs* of A on these inputs.

- We can compress at-most 2^M values to a size of $M < N$ bits.
- We can compress *a small fraction* $\frac{2^M}{2^N} = 2^{M-N}$ of all inputs to M bits.
  E.g., we can compress at-most 2^{32} out of 2^{64} values from 8 B to 4 B.
  Doing so will require more-than 8 B for some other inputs.

Limits of compression

Theorem

No algorithm A can compress every input I into compressed data A(I) with $|A(I)| < |I|$.

Proof

Intuituin 1 If A can compress every *input*, then $|A(A(I))| < |A(I)|$.
Repeaded application of A on the *input* will eventually lead to zero bits!

Intuition 2 Consider all possible *inputs* of N bits: we have 2^N distinct values.
We need at-least 2^N distinct *outputs* of A on these inputs.

- We can compress at-most 2^M values to a size of $M < N$ bits.
- We can compress *a small fraction* $\frac{2^M}{2^N} = 2^{M-N}$ of all inputs to M bits.
  E.g., we can compress at-most 2^{32} out of 2^{64} values from 8 B to 4 B.
  Doing so will require more-than 8 B for some other inputs.

Conceptually: We need *structure* in the input to be able to reliably compress that input!

A simple structure: Small alphabets

Consider DNA strings over the alphabet $\mathcal{A} = \{A, C, T, G\}$.

An usual DNA string $\mathcal{S}$ represented by N characters takes up $N\text{B} = 8N\text{bit}$.

How many bytes do we need to represent $\mathcal{S}$?

A simple structure: Small alphabets

Consider DNA strings over the alphabet $\mathcal{A} = \{A, C, T, G\}$.

An usual DNA string $\mathcal{S}$ represented by N characters takes up $N\text{B} = 8N\text{bit}$.

How many bytes do we need to represent $\mathcal{S}$?

- We have $|\mathcal{A}| = 4$ distinct values.

A simple structure: Small alphabets

Consider DNA strings over the alphabet $\mathcal{A} = \{A, C, T, G\}$.

An usual DNA string $\mathcal{S}$ represented by N characters takes up $N\mathrm{B} = 8N\mathrm{bit}$.

How many bytes do we need to represent $\mathcal{S}$?

- We have $|\mathcal{A}| = 4$ distinct values.
- We can represent 4 distinct values with 2 bit: “00”, “01”, “10”, and “11”.

A simple structure: Small alphabets

Consider DNA strings over the alphabet $\mathcal{A} = \{A, C, T, G\}$.

An usual DNA string $\mathcal{S}$ represented by N characters takes up $N\text{B} = 8N\text{bit}$.

How many bytes do we need to represent $\mathcal{S}$?

- We have $|\mathcal{A}| = 4$ distinct values.
- We can represent 4 distinct values with 2 bit: "00", "01", "10", and "11".
- A single byte holds 8 bit.

A simple structure: Small alphabets

Consider DNA strings over the alphabet $\mathcal{A} = \{A, C, T, G\}$.

An usual DNA string $\mathcal{S}$ represented by N characters takes up $N\text{B} = 8N\text{bit}$.

How many bytes do we need to represent $\mathcal{S}$?

- We have $|\mathcal{A}| = 4$ distinct values.
- We can represent 4 distinct values with 2 bit: “00”, “01”, “10”, and “11”.
- A single byte holds 8 bit.

Hence, we need at-most $\frac{8N}{4} = 2N\text{bit}$.
A compression ratio of 0.25.

A simple structure: Small alphabets

Consider DNA strings over the alphabet $\mathcal{A} = \{A, C, T, G\}$.

An usual DNA string $\mathcal{S}$ represented by N characters takes up NB = $8N$bit.

How many bytes do we need to represent $\mathcal{S}$?

- We have $|\mathcal{A}| = 4$ distinct values.
- We can represent 4 distinct values with 2 bit: “00”, “01”, “10”, and “11”.
- A single byte holds 8 bit.

Hence, we need at-most $\frac{8N}{4} = 2N$bit.
A compression ratio of 0.25.

From *bits* to *bytes*

We can store *four DNA characters* in one byte. Can we store $\mathcal{S}$ in $\frac{2N}{8}$B using the above?

A simple structure: Small alphabets

Consider DNA strings over the alphabet $\mathcal{A} = \{A, C, T, G\}$.

An usual DNA string $\mathcal{S}$ represented by N characters takes up $N\text{B} = 8N\text{bit}$.

How many bytes do we need to represent $\mathcal{S}$?

- We have $|\mathcal{A}| = 4$ distinct values.
- We can represent 4 distinct values with 2 bit: “00”, “01”, “10”, and “11”.
- A single byte holds 8 bit.

Hence, we need at-most $\frac{8N}{4} = 2N\text{bit}$.
A compression ratio of 0.25.

From *bits* to *bytes*

We can store *four DNA characters* in one byte. Can we store $\mathcal{S}$ in $\frac{2N}{8}\text{B}$ using the above?

No! Where in the last byte would our string end?
E.g., “ACTGA” takes 10 bit (1.25 B).

A common structure: Repetition

Consider the following string of bits:

`00000000000000011111110000000011111111111000000000011`

A common structure: Repetition

Consider the following string of bits:

```
00000000000000011111110000000011111111111000000000011
```

15 zeros | 7 ones | 8 zeros | 11 ones | 10 zeros | 2 ones

A common structure: Repetition

Consider the following string of bits:

000000000000000111111100000000111111111110000000000 11

15 zeros 7 ones 8 zeros 11 ones 10 zeros 2 ones

Number	(in 4-bit binary)
15	1111
7	0111
8	1000
11	1011
10	1010
2	0010

A common structure: Repetition

Consider the following string of bits:

```
111101111100010111101000010
```

$\underbrace{\texttt{1111}}_{\texttt{0s}}\underbrace{\texttt{0111}}_{\texttt{1s}}\ldots$

0s 1s 0s 1s 0s 1s

Number	(in 4-bit binary)
15	1111
7	0111
8	1000
11	1011
10	1010
2	0010

From $15 + 7 + 8 + 11 + 10 + 2 = 53$ bit to $6 \cdot 4 = 24$ bit.

A common structure: Repetition

Consider the following string of bits:

`000000000000000001111111000000001111111111100000000001 1`

17 zeros | 7 ones | 8 zeros | 11 ones | 10 zeros | 2 ones

Number	(in 4-bit binary)
15	1111
7	0111
8	1000
11	1011
10	1010
2	0010

A common structure: Repetition

Consider the following string of bits:

`111100000001001111000101110100010`

`0s 1s 0s 1s 0s 1s 0s 1s`

Number	(in 4-bit binary)
15	`1111`
7	`0111`
8	`1000`
11	`1011`
10	`1010`
2	`0010`

From $17 + 7 + 8 + 11 + 10 + 2 = 55$ bit to $8 \cdot 4 = 32$ bit.

A common structure: Repetition

Consider the following string of bits:

`11110000001001111000101110100010`

`0s 1s 0s 1s 0s 1s 0s 1s`

Number	(in 4-bit binary)
15	1111
7	0111
8	1000
11	1011
10	1010
2	0010

Run-length encoding: *simple idea* with good results on *bitmaps*.

Another common structure: Using symbol frequencies

Consider *simple* text written in the English language.

Another common structure: Using symbol frequencies

Consider *simple* text written in the English language.

The text uses the following 66 symbols "frequently":

- Digits `0123456789`: 10 symbols.
- Lower-case letters "a"–"z": 26 symbols.
- upper-case letters "A"–"Z": 26 symbols.
- Punctuation " ", ".", ",", "!": 4 symbols.

Another common structure: Using symbol frequencies

Consider *simple* text written in the English language.

The text uses the following 66 symbols "frequently":

- Digits `0123456789`: 10 symbols.
- Lower-case letters "a"–"z": 26 symbols.
- upper-case letters "A"–"Z": 26 symbols.
- Punctuation " ", ".", ",", "!": 4 symbols.

Stored normally, *each* symbol occupies 1 B = 8 bit.

Another common structure: Using symbol frequencies

Consider *simple* text written in the English language.

The text uses the following 66 symbols "frequently":

- Digits `0123456789`: 10 symbols.
- Lower-case letters "a"–"z": 26 symbols.
- upper-case letters "A"–"Z": 26 symbols.
- Punctuation " ", ".", ",", "!": 4 symbols.

Stored normally, *each* symbol occupies 1 B = 8 bit.

Even in these "frequent" symbols, some are much rarer than others: "x" versus "e".

Another common structure: Using symbol frequencies

Consider *simple* text written in the English language.
The text uses the following 66 symbols "frequently":

- Digits `0123456789`: 10 symbols.
- Lower-case letters "a"–"z": 26 symbols.
- upper-case letters "A"–"Z": 26 symbols.
- Punctuation " ", ".", ",", "!": 4 symbols.

Stored normally, *each* symbol occupies 1 B = 8 bit.

Even in these "frequent" symbols, some are much rarer than others: "x" versus "e".

Idea. Use fewer bits for frequent characters, more for rare characters.

Another common structure: Using symbol frequencies

Consider the string “anna can scan a can!”.

Another common structure: Using symbol frequencies

Consider the string "anna can scan a can!".

Symbol	Count	Bit pattern
'a'	6	
'n'	5	
' '	4	
'c'	3	
's'	1	
'!'	1	

The string has *6 distinct symbols*: at-least 3 bits if *all the same length*.

Another common structure: Using symbol frequencies

Consider the string "`anna can scan a can!`".

Symbol	Count	Bit pattern
'a'	6	`000`
'n'	5	`001`
' '	4	`010`
'c'	3	`011`
's'	1	`100`
'!'	1	`101`

The string has *6 distinct symbols*: at-least 3 bits if *all the same length*.

Another common structure: Using symbol frequencies

Consider the string "anna can scan a can!".

Symbol	Count	Bit pattern
'a'	6	0
'n'	5	1
' '	4	00
'c'	3	01
's'	1	10
'!'	1	11

Attempt 1. The most-frequent symbols get the shortest bit patterns.

```
a n n a    c  a n    s  c  a n    a    c  a n !
0 1 1 0 00 01 0 1 00 10 01 0 1 00 0 00 01 0 1 11
```

Another common structure: Using symbol frequencies

Consider the string "`anna can scan a can!`".

Symbol	Count	Bit pattern
'a'	6	0
'n'	5	1
' '	4	00
'c'	3	01
's'	1	10
'!'	1	11

Attempt 1. The most-frequent symbols get the shortest bit patterns.

```
a n n a    c  a n    s  c  a n    a    c  a n !
0 1 1 0 00 01 0 1 00 10 01 0 1 00 0 00 01 0 1 11
```

```
0110000101001001010000001011 1
```

Another common structure: Using symbol frequencies

Consider the string "`anna can scan a can!`".

Symbol	Count	Bit pattern
'a'	6	`0`
'n'	5	`1`
' '	4	`00`
'c'	3	`01`
's'	1	`10`
'!'	1	`11`

Attempt 1. The most-frequent symbols get the shortest bit patterns.

`01100001010010010100000010111` ← 29 bit *instead of at-least* 60 bit.

Another common structure: Using symbol frequencies

Consider the string "`anna can scan a can!`".

Symbol	Count	Bit pattern
'a'	6	`0`
'n'	5	`1`
' '	4	`00`
'c'	3	`01`
's'	1	`10`
'!'	1	`11`

Attempt 1. The most-frequent symbols get the shortest bit patterns.

`01100001010010010100000010111` ← 29 bit *instead of at-least* 60 bit.

a?

Another common structure: Using symbol frequencies

Consider the string "`anna can scan a can!`".

Symbol	Count	Bit pattern
'a'	6	`0`
'n'	5	`1`
' '	4	`00`
'c'	3	`01`
's'	1	`10`
'!'	1	`11`

Attempt 1. The most-frequent symbols get the shortest bit patterns.

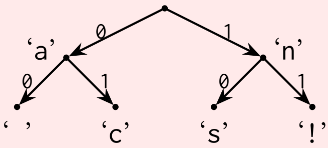
`01100001010010010100000010111` ← 29 bit *instead of at-least* 60 bit.

$\underbrace{\texttt{01}}_{c?}$

Another common structure: Using symbol frequencies

Consider the string "`anna can scan a can!`".

Symbol	Count	Bit pattern
'a'	6	`0`
'n'	5	`1`
' '	4	`00`
'c'	3	`01`
's'	1	`10`
'!'	1	`11`



Attempt 1. The most-frequent symbols get the shortest bit patterns.

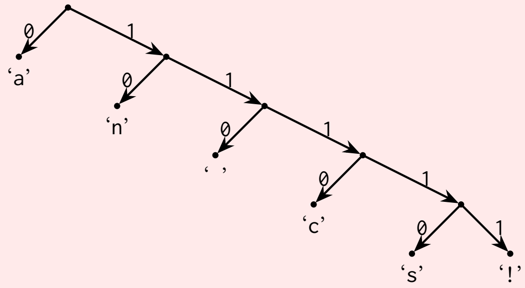
`01100001010010010100000010111` ⟵ 29 bit *instead of at-least* 60 bit.

Issue. The bit pattern of one symbol (e.g., a) is a *prefix* of other symbols!

Another common structure: Using symbol frequencies

Consider the string "`anna can scan a can!`".

Symbol	Count	Bit pattern
'a'	6	`0`
'n'	5	`10`
' '	4	`110`
'c'	3	`1110`
's'	1	`11110`
'!'	1	`11111`

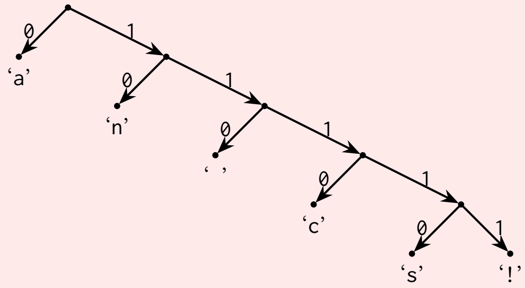


Attempt 2. The most-frequent symbols get the shortest *prefix-free* bit patterns.

Another common structure: Using symbol frequencies

Consider the string "anna can scan a can!".

Symbol	Count	Bit pattern
'a'	6	0
'n'	5	10
' '	4	110
'c'	3	1110
's'	1	11110
'!'	1	11111



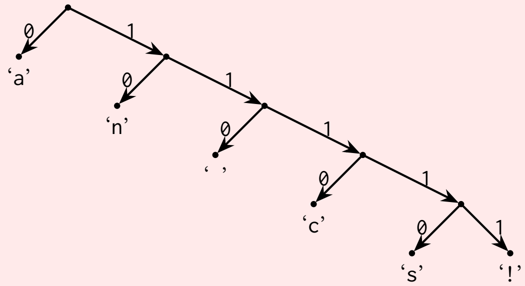
Attempt 2. The most-frequent symbols get the shortest *prefix-free* bit patterns.

```
a n  n  a     c    a n      s     c    a n      a     c    a n  !
0 10 10 0 110 1110 0 10 110 11110 1110 0 10 110 0 110 1110 0 10 11111
```

Another common structure: Using symbol frequencies

Consider the string “`anna can scan a can!`”.

Symbol	Count	Bit pattern
‘a’	6	`0`
‘n’	5	`10`
‘ ’	4	`110`
‘c’	3	`1110`
‘s’	1	`11110`
‘!’	1	`11111`



Attempt 2. The most-frequent symbols get the shortest *prefix-free* bit patterns.

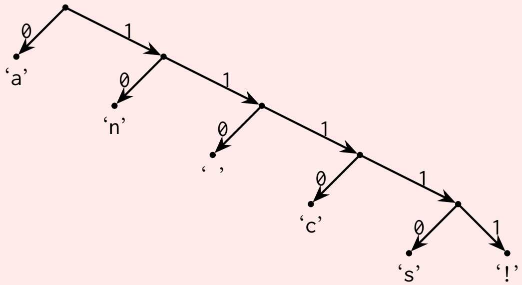
```
a n  n  a     c    a n      s     c    a n     a     c    a n  !
0 10 10 0 110 1110 0 10 110 11110 1110 0 10 110 0 110 1110 0 10 11111
```

`01010011011100101101111011100101100110111001011111` ← 50 bit.

Another common structure: Using symbol frequencies

Consider the string “anna can scan a can!”.

Symbol	Count	Bit pattern
‘a’	6	0
‘n’	5	10
‘ ’	4	110
‘c’	3	1110
‘s’	1	11110
‘!’	1	11111



Attempt 2. The most-frequent symbols get the shortest *prefix-free* bit patterns.

01010011011100101101111011100101100110111001011111 ← 50 bit.

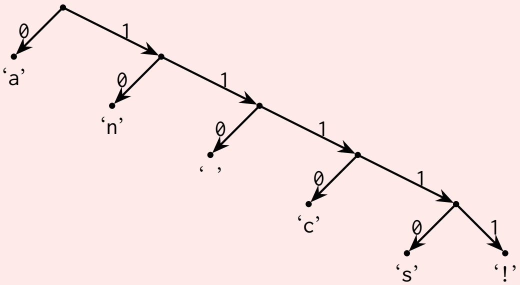
⋎

a

Another common structure: Using symbol frequencies

Consider the string “`anna can scan a can!`”.

Symbol	Count	Bit pattern
‘a’	6	0
‘n’	5	10
‘ ’	4	110
‘c’	3	1110
‘s’	1	11110
‘!’	1	11111



Attempt 2. The most-frequent symbols get the shortest *prefix-free* bit patterns.

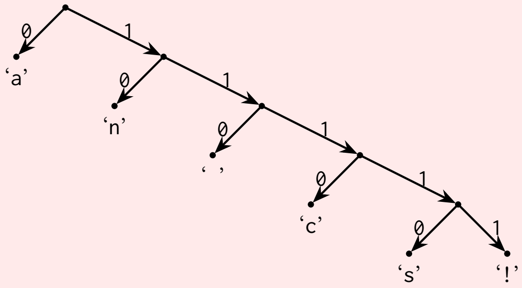
`01010011011100101101111011100101100110111001011111` ← 50 bit.

`a n n a ‘ ’ c a n ‘ ’ s c a n ‘ ’ a ‘ ’ c a n ‘!’`

Another common structure: Using symbol frequencies

Consider the string “`anna can scan a can!`”.

Symbol	Count	Bit pattern
‘a’	6	0
‘n’	5	10
‘ ’	4	110
‘c’	3	1110
‘s’	1	11110
‘!’	1	11111



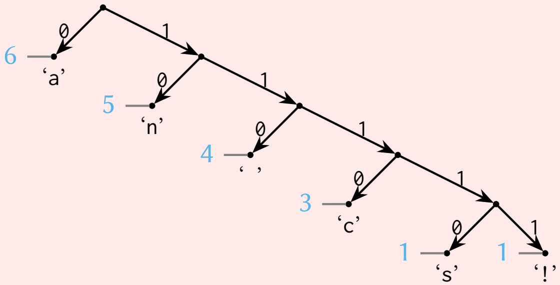
Attempt 2. The most-frequent symbols get the shortest *prefix-free* bit patterns.

`01010011011100101101111011100101100110111001011111` ← 50 bit.

Another common structure: Using symbol frequencies

Consider the string "anna can scan a can!".

Symbol	Count	Bit pattern
'a'	6	0
'n'	5	10
' '	4	110
'c'	3	1110
's'	1	11110
'!'	1	11111

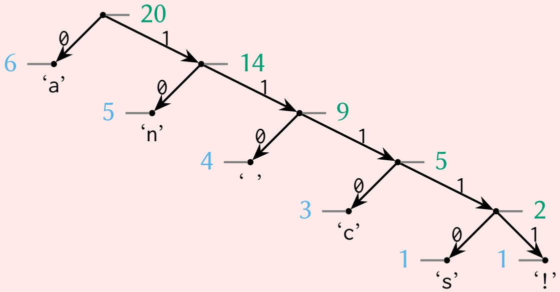


Questions. How to construct the bit patterns and are these patterns optimal?

Another common structure: Using symbol frequencies

Consider the string “anna can scan a can!”.

Symbol	Count	Bit pattern
‘a’	6	0
‘n’	5	10
‘ ’	4	110
‘c’	3	1110
‘s’	1	11110
‘!’	1	11111



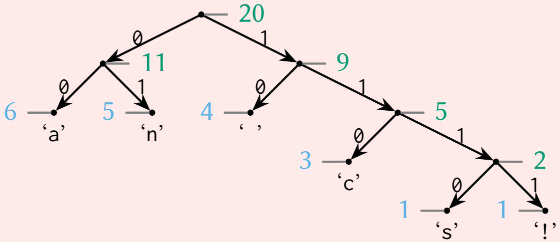
Questions. How to construct the bit patterns and are these patterns optimal?

$$6 \cdot 1 + 5 \cdot 2 + 4 \cdot 3 + 3 \cdot 4 + 1 \cdot 5 + 1 \cdot 5 = 50.$$

Another common structure: Using symbol frequencies

Consider the string “anna can scan a can!”.

Symbol	Count	Bit pattern
‘a’	6	00
‘n’	5	01
‘ ’	4	10
‘c’	3	110
‘s’	1	1110
‘!’	1	1111



Questions. How to construct the bit patterns and are these patterns optimal?

$$6 \cdot 2 + 5 \cdot 2 + 4 \cdot 2 + 3 \cdot 3 + 1 \cdot 4 + 1 \cdot 4 = 47.$$

Huffman coding

Problem

Given an alphabet $\mathcal{A}$ and symbol-frequencies $f : \mathcal{A} \to [0, 1]$,
Produce *prefix-free bit patterns* for all symbols in $\mathcal{A}$ such that these patterns are optimal.

Huffman coding

Problem

Given an alphabet $\mathcal{A}$ and symbol-frequencies $f : \mathcal{A} \to [0, 1]$,
Produce *prefix-free bit patterns* for all symbols in $\mathcal{A}$ such that these patterns are optimal.

Optimal No other bit patterns will compress strings $\mathcal{S}$ over $\mathcal{A}$ more
Assuming symbols counts in $\mathcal{S}$ agree with f.

Huffman coding

Problem

Given an alphabet $\mathcal{A}$ and symbol-frequencies $f : \mathcal{A} \rightarrow [0, 1]$,
Produce *prefix-free bit patterns* for all symbols in $\mathcal{A}$ such that these patterns are optimal.

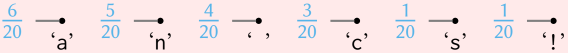
Optimal No other bit patterns will compress strings $\mathcal{S}$ over $\mathcal{A}$ more
Assuming symbols counts in $\mathcal{S}$ agree with f.

Algorithm HuffmanPFTrie(f):

$Q :=$ a min-priority queue.
for all $\sigma \in \mathcal{A}$ **do**
  Make a leaf-node n labeled σ.
  Add $(n, f(\sigma))$ to Q with priority $f(\sigma)$.
while $|Q| \geq 2$ **do**
  $(n_0, p_0) :=$ DelMin(Q), $(n_1, p_1) :=$ DelMin(Q).
  Create a node n with children n_0 labeled `0`, n_1 labeled `1`.
  Add node $(n, p_0 + p_1)$ to Q with priority $p_0 + p_1$.
return n with $(n, p) :=$ DelMin(Q).

Huffman coding

Symbol	Count	Frequency
'a'	6	$\frac{6}{20}$
'n'	5	$\frac{5}{20}$
' '	4	$\frac{4}{20}$
'c'	3	$\frac{3}{20}$
's'	1	$\frac{1}{20}$
'!'	1	$\frac{1}{20}$

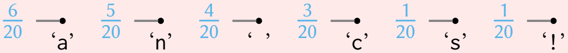


Algorithm HuffmanPFTrie(f):

Q := a min-priority queue.
for all $\sigma \in \mathcal{A}$ **do**
    Make a leaf-node n labeled σ.
    Add $(n, f(\sigma))$ to Q with priority $f(\sigma)$.
…

Huffman coding

Symbol	Count	Frequency
‘a’	6	$\frac{6}{20}$
‘n’	5	$\frac{5}{20}$
‘ ’	4	$\frac{4}{20}$
‘c’	3	$\frac{3}{20}$
‘s’	1	$\frac{1}{20}$
‘!’	1	$\frac{1}{20}$



Algorithm HUFFMANPFTRIE(f):

...

while $|Q| \geq 2$ **do**

  $(n_0, p_0) :=$ DELMIN(Q), $(n_1, p_1) :=$ DELMIN(Q).

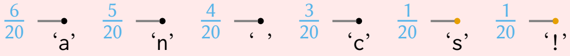
  Create a node n with children n_0 labeled 0, n_1 labeled 1.

  Add node $(n, p_0 + p_1)$ to Q with priority $p_0 + p_1$.

return n with $(n, p) :=$ DELMIN(Q).

Huffman coding

Symbol	Count	Frequency
‘a’	6	$\frac{6}{20}$
‘n’	5	$\frac{5}{20}$
‘ ’	4	$\frac{4}{20}$
‘c’	3	$\frac{3}{20}$
‘s’	1	$\frac{1}{20}$
‘!’	1	$\frac{1}{20}$

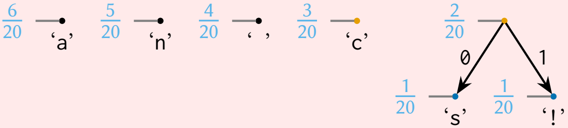


Algorithm HuffmanPFTrie(f):

...
while $|Q| \geq 2$ **do**
  $(n_0, p_0) :=$ DelMin(Q), $(n_1, p_1) :=$ DelMin(Q).
  Create a node n with children n_0 labeled `0`, n_1 labeled `1`.
  Add node $(n, p_0 + p_1)$ to Q with priority $p_0 + p_1$.
return n with $(n, p) :=$ DelMin(Q).

Huffman coding

Symbol	Count	Frequency
‘a’	6	$\frac{6}{20}$
‘n’	5	$\frac{5}{20}$
‘ ’	4	$\frac{4}{20}$
‘c’	3	$\frac{3}{20}$
‘s’	1	$\frac{1}{20}$
‘!’	1	$\frac{1}{20}$



Algorithm HuffmanPFTrie(f):

...

while $|Q| \geq 2$ **do**

  $(n_0, p_0) :=$ DelMin(Q), $(n_1, p_1) :=$ DelMin(Q).

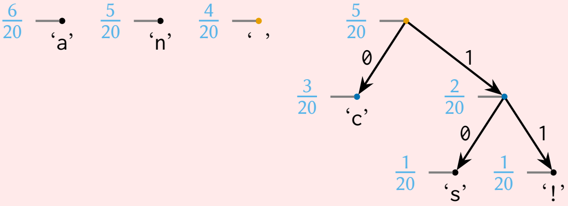
  Create a node n with children n_0 labeled 0, n_1 labeled 1.

  Add node $(n, p_0 + p_1)$ to Q with priority $p_0 + p_1$.

return n with $(n, p) :=$ DelMin(Q).

Huffman coding

Symbol	Count	Frequency
‘a’	6	$\frac{6}{20}$
‘n’	5	$\frac{5}{20}$
‘ ’	4	$\frac{4}{20}$
‘c’	3	$\frac{3}{20}$
‘s’	1	$\frac{1}{20}$
‘!’	1	$\frac{1}{20}$



Algorithm HuffmanPFTrie(f):

...

while $|Q| \geq 2$ **do**

  $(n_0, p_0) := \text{DelMin}(Q)$, $(n_1, p_1) := \text{DelMin}(Q)$.

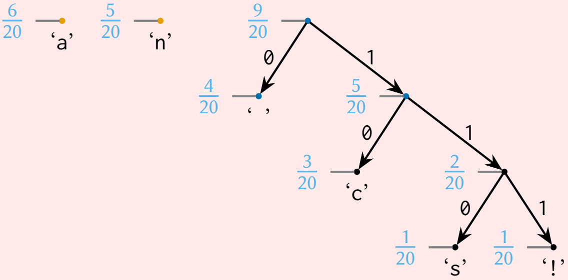
  Create a node n with children n_0 labeled 0, n_1 labeled 1.

  Add node $(n, p_0 + p_1)$ to Q with priority $p_0 + p_1$.

return n with $(n, p) := \text{DelMin}(Q)$.

Huffman coding

Symbol	Count	Frequency
‘a’	6	$\frac{6}{20}$
‘n’	5	$\frac{5}{20}$
‘ ’	4	$\frac{4}{20}$
‘c’	3	$\frac{3}{20}$
‘s’	1	$\frac{1}{20}$
‘!’	1	$\frac{1}{20}$

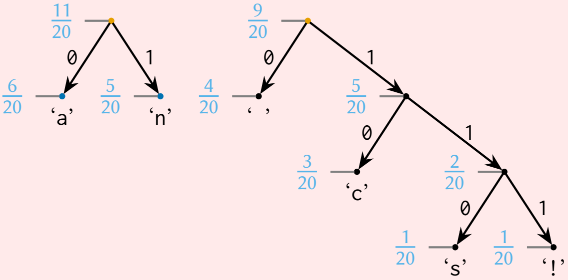


Algorithm HuffmanPFTrie(f):

…
while $|Q| \geq 2$ **do**
    (n_0, p_0) := DelMin(Q), (n_1, p_1) := DelMin(Q).
    Create a node n with children n_0 labeled 0, n_1 labeled 1.
    Add node $(n, p_0 + p_1)$ to Q with priority $p_0 + p_1$.
return n with (n, p) := DelMin(Q).

Huffman coding

Symbol	Count	Frequency
‘a’	6	$\frac{6}{20}$
‘n’	5	$\frac{5}{20}$
‘ ’	4	$\frac{4}{20}$
‘c’	3	$\frac{3}{20}$
‘s’	1	$\frac{1}{20}$
‘!’	1	$\frac{1}{20}$



Algorithm HuffmanPFTrie(f):

…

while $|Q| \geq 2$ **do**

 $(n_0, p_0) := \textsc{DelMin}(Q)$, $(n_1, p_1) := \textsc{DelMin}(Q)$.

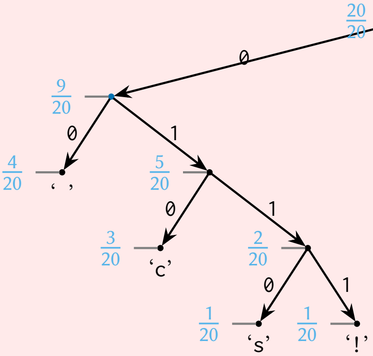
 Create a node n with children n_0 labeled `0`, n_1 labeled 1.

 Add node $(n, p_0 + p_1)$ to Q with priority $p_0 + p_1$.

return n with $(n, p) := \textsc{DelMin}(Q)$.

Huffman coding

Symbol	Count	Frequency
‘a’	6	$\frac{6}{20}$
‘n’	5	$\frac{5}{20}$
‘ ’	4	$\frac{4}{20}$
‘c’	3	$\frac{3}{20}$
‘s’	1	$\frac{1}{20}$
‘!’	1	$\frac{1}{20}$

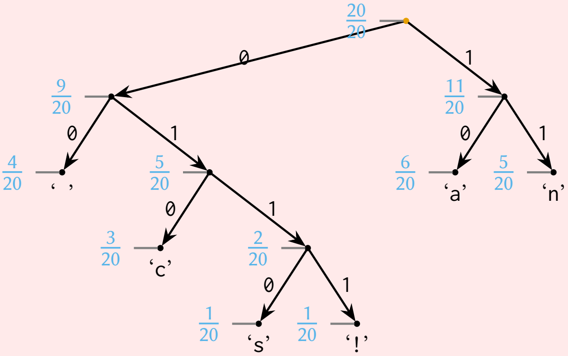


Algorithm HUFFMANPFTRIE(f):

…
while $|Q| \geq 2$ **do**
    $(n_0, p_0) :=$ DELMIN(Q), $(n_1, p_1) :=$ DELMIN(Q).
    Create a node n with children n_0 labeled 0, n_1 labeled 1.
    Add node $(n, p_0 + p_1)$ to Q with priority $p_0 + p_1$.
return n with $(n, p) :=$ DELMIN(Q).

Huffman coding

Symbol	Count	Frequency
‘a’	6	$\frac{6}{20}$
‘n’	5	$\frac{5}{20}$
‘ ’	4	$\frac{4}{20}$
‘c’	3	$\frac{3}{20}$
‘s’	1	$\frac{1}{20}$
‘!’	1	$\frac{1}{20}$



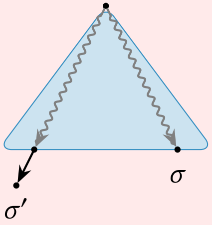
Algorithm HUFFMANPFTRIE(f):

…
while $|Q| \geq 2$ **do**
  $(n_0, p_0) :=$ DELMIN(Q), $(n_1, p_1) :=$ DELMIN(Q).
  Create a node n with children n_0 labeled `0`, n_1 labeled 1.
  Add node $(n, p_0 + p_1)$ to Q with priority $p_0 + p_1$.
return n with $(n, p) :=$ DELMIN(Q).

Huffman coding

Property 1

Let $\sigma \in \mathcal{A}$ be the symbol with lowest frequency f.
Any optimal prefix-free trie for $\mathcal{A}, f$ can be changed such that the path to σ is longest.

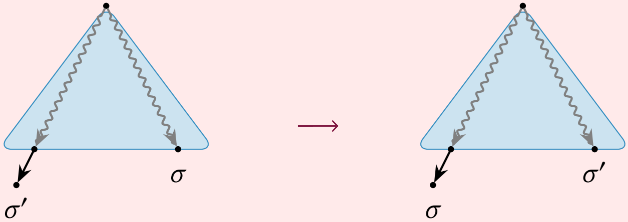


Huffman coding

Property 1

Let $\sigma \in \mathcal{A}$ be the symbol with lowest frequency f.

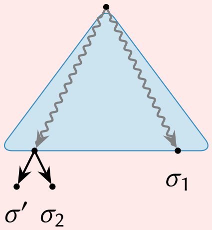
Any optimal prefix-free trie for $\mathcal{A}, f$ can be changed such that the path to σ is longest.



Huffman coding

Property 2

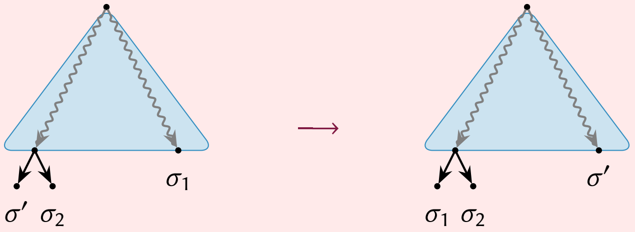
Let $\sigma_1, \sigma_2 \in \mathcal{A}$ be the symbols with lowest frequency f.
Any optimal prefix-free trie for $\mathcal{A}, f$ can be changed such that symbols σ_1, σ_2 are children of the same node.



Huffman coding

Property 2

Let $\sigma_1, \sigma_2 \in \mathcal{A}$ be the symbols with lowest frequency f.
Any optimal prefix-free trie for $\mathcal{A}, f$ can be changed such that symbols σ_1, σ_2 are children of the same node.



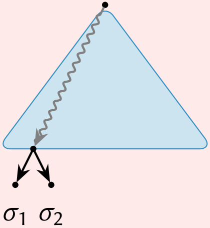
Huffman coding

Property 3

Let $\sigma_1, \sigma_2 \in \mathcal{A}$ be symbols represented by children n_0, n_1 of node n in trie T.
Let T' be the prefix-free trie for $\mathcal{A}', f'$ with

- $\mathcal{A}' = \mathcal{A} \setminus \{\sigma_2\}$;
- $f' = \{\sigma \mapsto f(\sigma) \mid \sigma \in \mathcal{A} \setminus \{\sigma_1, \sigma_2\}\} \cup \{\sigma_1 \mapsto f(\sigma_1) + f(\sigma_2)\}$; and
- leafs n_0, n_1 removed from n and n made to represent σ_1.

The trie T is optimal for $\mathcal{A}, f$ if and only if T' is optimal for $\mathcal{A}', f'$.



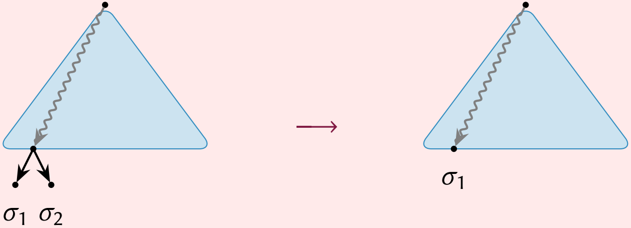
Huffman coding

Property 3

Let $\sigma_1, \sigma_2 \in \mathcal{A}$ be symbols represented by children n_0, n_1 of node n in trie T.
Let T' be the prefix-free trie for $\mathcal{A}', f'$ with

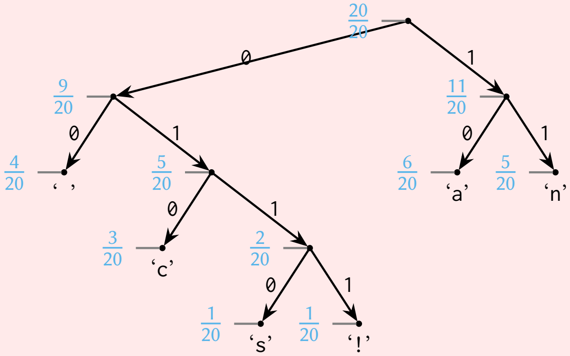
- $\mathcal{A}' = \mathcal{A} \setminus \{\sigma_2\}$;
- $f' = \{\sigma \mapsto f(\sigma) \mid \sigma \in \mathcal{A} \setminus \{\sigma_1, \sigma_2\}\} \cup \{\sigma_1 \mapsto f(\sigma_1) + f(\sigma_2)\}$; and
- leafs n_0, n_1 removed from n and n made to represent σ_1.

The trie T is optimal for $\mathcal{A}, f$ if and only if T' is optimal for $\mathcal{A}', f'$.



Huffman coding

Symbol	Count	Frequency
‘a’	6	$\frac{6}{20}$
‘n’	5	$\frac{5}{20}$
‘ ’	4	$\frac{4}{20}$
‘c’	3	$\frac{3}{20}$
‘s’	1	$\frac{1}{20}$
‘!’	1	$\frac{1}{20}$



Theorem

The HuffmanPFTrie *algorithm builds an optimal prefix-free code.*

Proof (sketch)

HuffmanPFTrie follows Property 1–3.

Beyond Huffman: Frequent strings

Huffman looks at frequent symbols from an alphabet.

- We can generalize these ideas to *frequent* sequences of symbols.
- Tries can be used to efficiently manage *frequency* data for substrings in an input.
- Challenge: which substrings to consider?
  E.g., fixed length, maximum length, ...

Beyond Huffman: Frequent strings

Huffman looks at frequent symbols from an alphabet.

- We can generalize these ideas to *frequent* sequences of symbols.
- Tries can be used to efficiently manage *frequency* data for substrings in an input.
- Challenge: which substrings to consider?
  E.g., fixed length, maximum length, ...

Many variations of this idea used in practice, e.g., `.zip`, `.gif`, ...

Basic substring search: Searching a needle in a haystack

Problem

Given strings $\mathcal{S}$ (the haystack) and P (the needle or pattern), return the first position in $\mathcal{S}$ at which P occurs (if any).

Basic substring search: Searching a needle in a haystack

Problem

Given strings $\mathcal{S}$ (the haystack) and P (the needle or pattern), return the first position in $\mathcal{S}$ at which P occurs (if any).

Algorithm BasicStringSearch($\mathcal{S}$, P):

1: **for** $i := 0$ upto $|\mathcal{S}| - 1$ **do**
2:     **if** pattern P starts at $\mathcal{S}[i]$ **then**
3:         **return** i.

Basic substring search: Searching a needle in a haystack

Problem

Given strings $\mathcal{S}$ (the haystack) and P (the needle or pattern), return the first position in $\mathcal{S}$ at which P occurs (if any).

Algorithm BasicStringSearch($\mathcal{S}$, P):

1: **for** $i := 0$ upto $|\mathcal{S}| - 1$ **do**
2:     **if** pattern P starts at $\mathcal{S}[i]$ **then**
3:         **return** i.

Searching P = "*example*"

```
“an example of words”
 example
```

Basic substring search: Searching a needle in a haystack

Problem

Given strings $\mathcal{S}$ (the haystack) and P (the needle or pattern), return the first position in $\mathcal{S}$ at which P occurs (if any).

Algorithm BasicStringSearch($\mathcal{S}$, P):

```
1: for i := 0 upto |S| − 1 do
2:     if pattern P starts at S[i] then
3:         return i.
```

Searching P = "*example*"

```
“ a n   e x a m p l e   o f   w o r d s ”
    e x a m p l e
```

Basic substring search: Searching a needle in a haystack

Problem

Given strings $\mathcal{S}$ (the haystack) and P (the needle or pattern),
return the first position in $\mathcal{S}$ at which P occurs (if any).

Algorithm BASICSTRINGSEARCH($\mathcal{S}$, P):

for $i := 0$ upto $|\mathcal{S}| - 1$ **do**
    if pattern P starts at $\mathcal{S}[i]$ **then**
        return i.

Searching P = "*example*"

```
“a n   e x a m p l e   o f   w o r d s”
         e x a m p l e
```

Basic substring search: Searching a needle in a haystack

Problem

Given strings $\mathcal{S}$ (the haystack) and P (the needle or pattern), return the first position in $\mathcal{S}$ at which P occurs (if any).

Algorithm BasicStringSearch($\mathcal{S}$, P):

1: **for** $i := 0$ upto $|\mathcal{S}| - 1$ **do**
2:     **if** pattern P starts at $\mathcal{S}[i]$ **then**
3:         **return** i.

Searching P = "*example*"

```
“an example of words”
    example
```

Basic substring search: Searching a needle in a haystack

Problem

Given strings $\mathcal{S}$ (the haystack) and P (the needle or pattern), return the first position in $\mathcal{S}$ at which P occurs (if any).

Algorithm BasicStringSearch($\mathcal{S}$, P):

```
for i := 0 upto |S| − 1 do
    if pattern P starts at S[i] then
        return i.
```

Searching P = "*example*"

```
“ a n   e x a m p l e   o f   w o r d s ”
        e x a m p l e
```

Basic substring search: Searching a needle in a haystack

Problem

Given strings $\mathcal{S}$ (the haystack) and P (the needle or pattern), return the first position in $\mathcal{S}$ at which P occurs (if any).

Algorithm BasicStringSearch($\mathcal{S}$, P):

for $i := 0$ upto $|\mathcal{S}| - 1$ **do**
    if pattern P starts at $\mathcal{S}[i]$ **then**
        return i.

Searching $P =$ "*great*"

```
“ a n   e x a m p l e   o f   w o r d s ”
                         g r e a t
```

Basic substring search: Searching a needle in a haystack

Problem

Given strings $\mathcal{S}$ (the haystack) and P (the needle or pattern), return the first position in $\mathcal{S}$ at which P occurs (if any).

Algorithm BasicStringSearch($\mathcal{S}$, P):

1: **for** $i := 0$ upto $|\mathcal{S}| - 1$ **do**
2:     **if** pattern P starts at $\mathcal{S}[i]$ **then**
3:         **return** i.

Searching P = "*great*"

```
“an example of words”
              great
```

Basic substring search: Searching a needle in a haystack

Problem

Given strings $\mathcal{S}$ (the haystack) and P (the needle or pattern), return the first position in $\mathcal{S}$ at which P occurs (if any).

Algorithm BasicStringSearch($\mathcal{S}$, P):

for $i := 0$ upto $|\mathcal{S}| - 1$ **do**
  if pattern P starts at $\mathcal{S}[i]$ **then**
    return i.

Searching $P =$ “*great*”

```
“ a n   e x a m p l e   o f   w o r d s ”
                             g r e a t
```

Basic substring search: Searching a needle in a haystack

Problem

Given strings $\mathcal{S}$ (the haystack) and P (the needle or pattern), return the first position in $\mathcal{S}$ at which P occurs (if any).

Algorithm BasicStringSearch($\mathcal{S}$, P):

```
for i := 0 upto |S| − 1 do
    if pattern P starts at S[i] then
        return i.
```

Searching P = "*great*"

```
“an example of words”
              great
```

Basic substring search: Searching a needle in a haystack

Problem

Given strings $\mathcal{S}$ (the haystack) and P (the needle or pattern), return the first position in $\mathcal{S}$ at which P occurs (if any).

Algorithm BASICSTRINGSEARCH($\mathcal{S}$, P):

1: **for** $i := 0$ upto $|\mathcal{S}| - 1$ **do**
2:     **if** pattern P starts at $\mathcal{S}[i]$ **then**
3:         **return** i.

Searching P = "*great*"

```
“an example of words”
               great
```

Basic substring search: Searching a needle in a haystack

Problem

Given strings $\mathcal{S}$ (the haystack) and P (the needle or pattern), return the first position in $\mathcal{S}$ at which P occurs (if any).

Algorithm BasicStringSearch($\mathcal{S}$, P):

1: **for** $i := 0$ upto $|\mathcal{S}| - 1$ **do**
2:     **if** pattern P starts at $\mathcal{S}[i]$ **then**
3:         **return** i.

Searching P = "*great*"

```
“an example of words”
               great
```

Basic substring search: Searching a needle in a haystack

Problem

Given strings $\mathcal{S}$ (the haystack) and P (the needle or pattern), return the first position in $\mathcal{S}$ at which P occurs (if any).

Algorithm BasicStringSearch($\mathcal{S}$, P):

1: **for** $i := 0$ upto $|\mathcal{S}| - |P|$ **do**
2: **if** pattern P starts at $\mathcal{S}[i]$ **then**
3: **return** i.

Searching P = "*great*"

```
"an example of words"
                great
```

Basic substring search: Searching a needle in a haystack

Problem

Given strings $\mathcal{S}$ (the haystack) and P (the needle or pattern), return the first position in $\mathcal{S}$ at which P occurs (if any).

Algorithm BasicStringSearch($\mathcal{S}$, P):

```
for i := 0 upto |S| − |P| do
    if pattern P starts at S[i] then
        return i.
```

Complexity

Basic substring search: Searching a needle in a haystack

Problem

Given strings $\mathcal{S}$ (the haystack) and P (the needle or pattern), return the first position in $\mathcal{S}$ at which P occurs (if any).

Algorithm BasicStringSearch($\mathcal{S}$, P):

for $i := 0$ upto $|\mathcal{S}| - |P|$ **do**
  if MatchString($\mathcal{S}$, P, i) **then**
    return i.

Algorithm MatchString($\mathcal{S}$, P, i):

for $j := 0$ upto $|P| - 1$ **do**
  if $\mathcal{S}[i + j] \neq P[j]$ **then**
    return `false`.
return `true`.

Complexity

Basic substring search: Searching a needle in a haystack

Problem

Given strings $\mathcal{S}$ (the haystack) and P (the needle or pattern), return the first position in $\mathcal{S}$ at which P occurs (if any).

Algorithm BasicStringSearch($\mathcal{S}$, P):

for $i := 0$ upto $|\mathcal{S}| - |P|$ **do**
    if MatchString($\mathcal{S}$, P, i) **then**
        return i.

Algorithm MatchString($\mathcal{S}$, P, i):

for $j := 0$ upto $|P| - 1$ **do**
    if $\mathcal{S}[i + j] \neq P[j]$ **then**
        return `false`.
return `true`.

} $\Theta(|P|)$

Complexity

Basic substring search: Searching a needle in a haystack

Problem

Given strings $\mathcal{S}$ (the haystack) and P (the needle or pattern), return the first position in $\mathcal{S}$ at which P occurs (if any).

Algorithm BasicStringSearch($\mathcal{S}$, P):

1: **for** $i := 0$ upto $|\mathcal{S}| - |P|$ **do**
2:     **if** MatchString($\mathcal{S}$, P, i) **then**
3:         **return** i.

$\Big\} \; \Theta((|\mathcal{S}| - |P|)|P|)$

Complexity

Basic substring search: Searching a needle in a haystack

Problem

Given strings $\mathcal{S}$ (the haystack) and P (the needle or pattern), return the first position in $\mathcal{S}$ at which P occurs (if any).

Algorithm BasicStringSearch($\mathcal{S}$, P):

```
for i := 0 upto |S| − |P| do
    if MatchString(S, P, i) then
        return i.
```

$\Theta((|\mathcal{S}| - |P|)|P|) = \Theta(|\mathcal{S}||P|)$

Complexity

Substring search: Room for improvement

Searching $P =$ “*string*”

“a strong string”

Substring search: Room for improvement

Searching P = "*string*"

```
“a  strong  string”
 string
```

Substring search: Room for improvement

Searching $P =$ "*string*"

```
“a strong string”
  string
```

Substring search: Room for improvement

Searching $P =$ "*string*"

```
" a   s t r o n g   s t r i n g "
      s t r i n g
```

Substring search: Room for improvement

Searching P = "*string*"

```
“ a   s t r o n g   s t r i n g ”
      s t r i n g
```

Substring search: Room for improvement

Searching $P =$ "*string*"

```
“a strong string”
  string
```

Substring search: Room for improvement

Searching $P =$ "*string*"

```
“a strong string”
  string
```

Substring search: Room for improvement

Searching $P =$ "*string*"

“ a   s t r o n g   s t r i n g ”

s t r i n g

Substring search: Room for improvement

Searching P = "*string*"

```
“a  strong  string”
         string
```

Substring search: Room for improvement

Searching $P =$ "*string*"

```
" a   s t r o n g   s t r i n g "
            s t r i n g
```

Substring search: Room for improvement

Searching P = "*string*"

```
" a   s t r o n g   s t r i n g "
                    s t r i n g
```

Substring search: Room for improvement

Searching $P =$ "*string*"

```
“ a   s t r o n g   s t r i n g ”
                    s t r i n g
```

Substring search: Room for improvement

Searching $P =$ "*string*"

```
“a strong string”
          string
```

Substring search: Room for improvement

Searching P = "$ACACGT$"

" A C A C A C A C G T "

Substring search: Room for improvement

Searching $P =$ "*ACACGT*"

" A C A C A C A C G T "

A C A C G T

Substring search: Room for improvement

Searching $P =$ "$ACACGT$"

“ A C A C A C A C G T ”

A C A C G T

Substring search: Room for improvement

Searching $P =$ "$ACACGT$"

" A C A C A C A C G T "

A C A C G T

Substring search: Room for improvement

Searching P = "*ACACGT*"

```
" A C A C A C A C G T "
  A C A C G T
```

Substring search: Room for improvement

Searching $P =$ "$ACACGT$"

" A C A C A C A C G T "

A C A C G T

Substring search: Room for improvement

Searching $P =$ "$ACACGT$"

```
" A C A C A C A C G T "
  A C A C G T
```

Substring search: Room for improvement

Searching $P =$ "$ACACGT$"

```
" A C A C A C A C G T "
      A C A C G T
```

Substring search: Room for improvement

Searching $P =$ "$ACACGT$"

```
" A C A C A C A C G T "
      A C A C G T
```

Substring search: Room for improvement

Searching $P =$ "$ACACGT$"

```
" A C A C A C A C G T "
        A C A C G T
```

Substring search: Room for improvement

Searching $P =$ "*ACACGT*"

```
" A C A C A C A C G T "
          A C A C G T
```

Substring search: Room for improvement

Searching $P =$ "$ACACGT$"

“ A C A C A C A C G T ”
A C A C G T

Substring search: Room for improvement

Searching $P =$ "$ACACGT$"

```
“ A C A C A C A C G T ”
          A C A C G T
```

Substring search: Room for improvement

Searching $P = $ "$ACACGT$"

" A C A C A C A C G T "

A C A C G T

Substring search: Room for improvement

Searching $P =$ "$ACACGT$"

" A C A C A C A C G T "

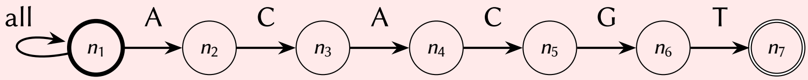
A C A C G T

Encode search patterns as an automaton

Finite automata

A finite automaton is a *graph* with

- a single *initial* node;
- zero-or-more *final* nodes;
- edges labeled with *symbols*.

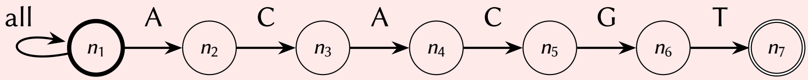


Encode search patterns as an automaton

Finite automata

A finite automaton is a *graph* with

- a single *initial* node;
- zero-or-more *final* nodes;
- edges labeled with *symbols*.



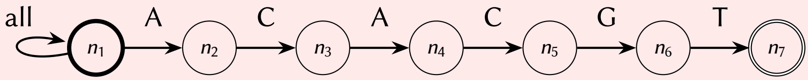
Typically, we refer to nodes as *states* and edges as *transitions*.

Encode search patterns as an automaton

Finite automata

A finite automaton is a *graph* with

- a single *initial* node;
- zero-or-more *final* nodes;
- edges labeled with *symbols*.



Running a finite automaton

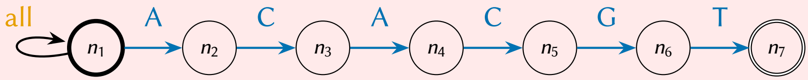
We can use a string as *input* to the automaton to decide which path to follow.

Encode search patterns as an automaton

Finite automata

A finite automaton is a *graph* with

- a single *initial* node;
- zero-or-more *final* nodes;
- edges labeled with *symbols*.



Running a finite automaton

We can use a string as *input* to the automaton to decide which path to follow.

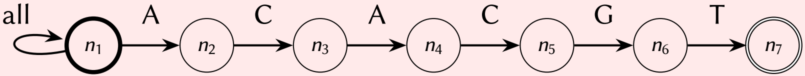
"ACACACACGT".

Encode search patterns as an automaton

Finite automata

A finite automaton is a *graph* with

- a single *initial* node;
- zero-or-more *final* nodes;
- edges labeled with *symbols*.

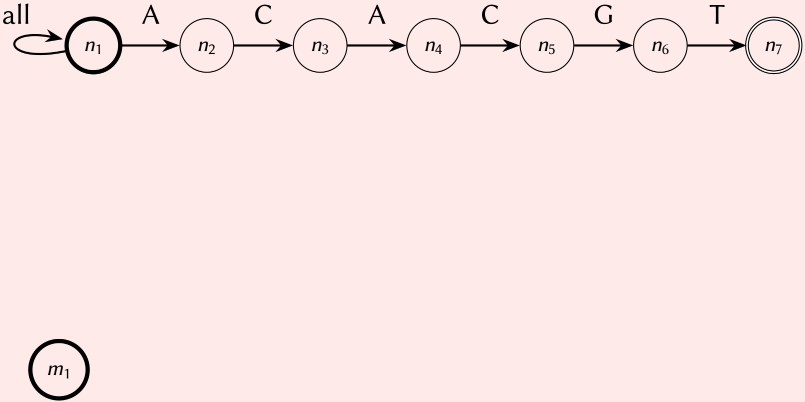


Running a finite automaton

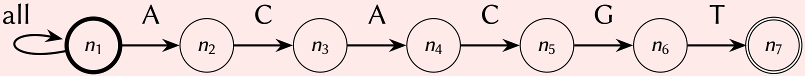
We can use a string as *input* to the automaton to decide which path to follow.

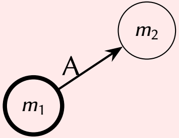
For efficiency: we want a *deterministic* automaton: an automaton without choices!

Encode search patterns as an automaton

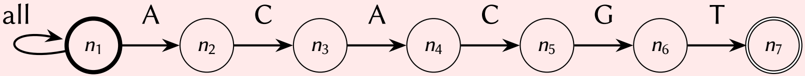


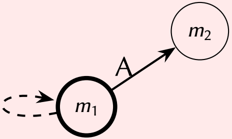
Encode search patterns as an automaton



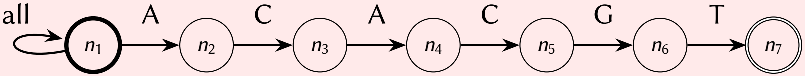


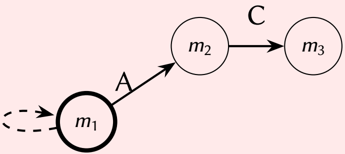
Encode search patterns as an automaton



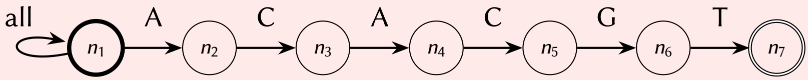


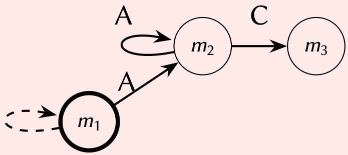
Encode search patterns as an automaton



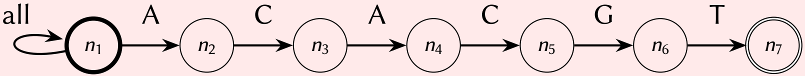


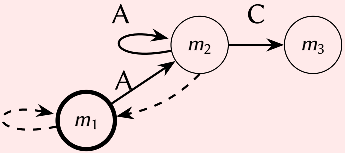
Encode search patterns as an automaton



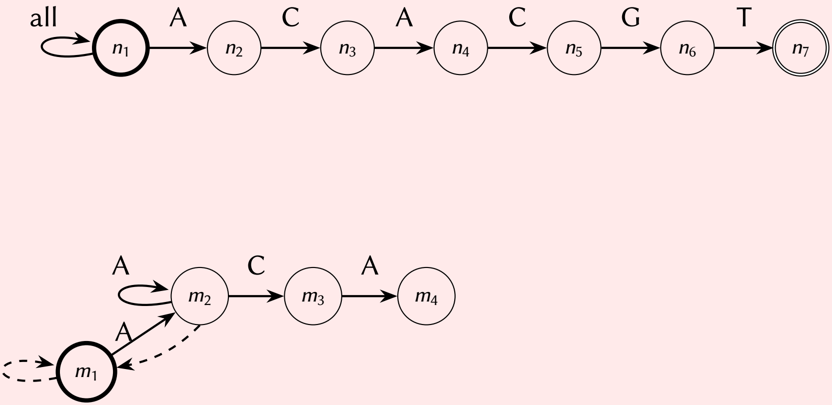


Encode search patterns as an automaton

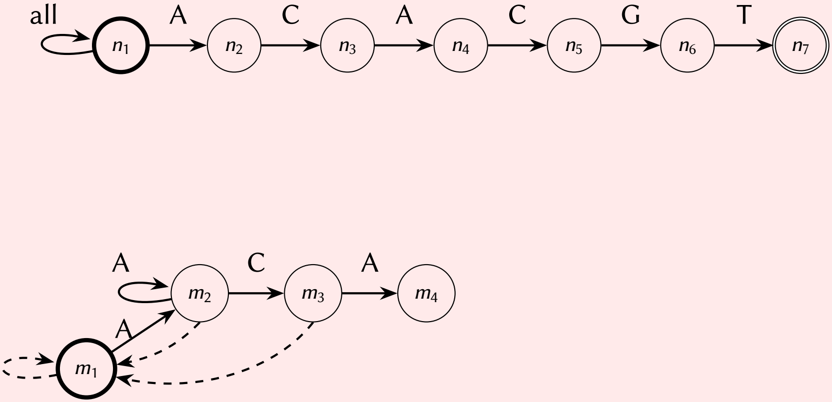




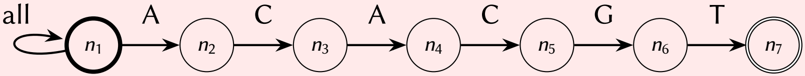
Encode search patterns as an automaton

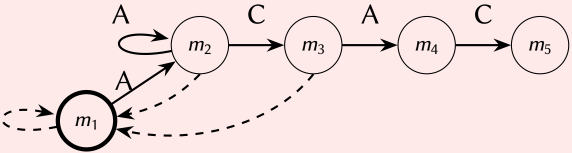


Encode search patterns as an automaton

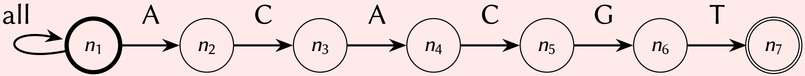


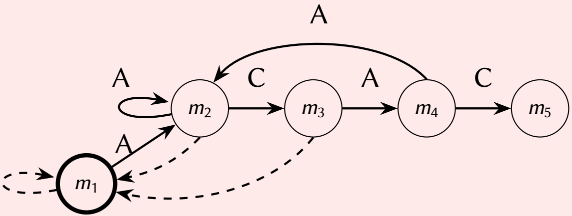
Encode search patterns as an automaton



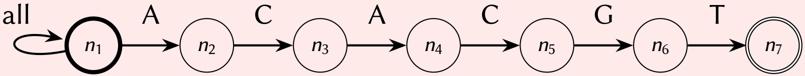


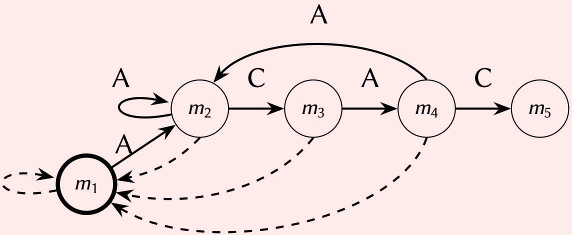
Encode search patterns as an automaton



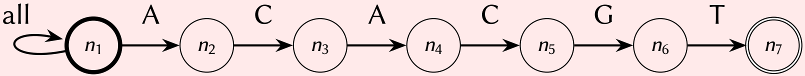


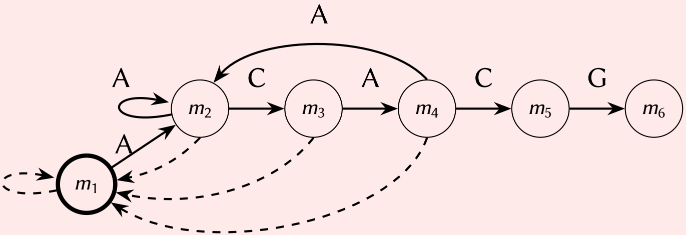
Encode search patterns as an automaton



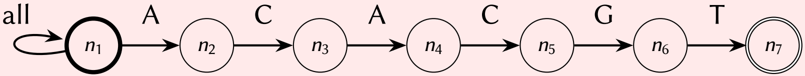


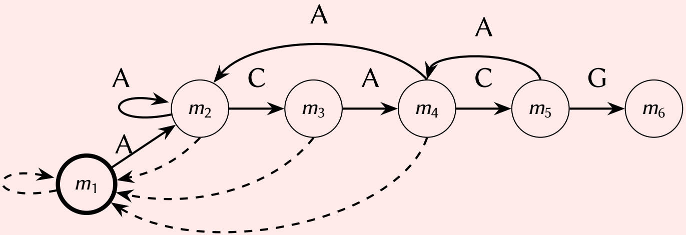
Encode search patterns as an automaton



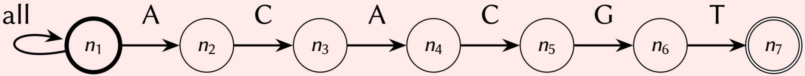


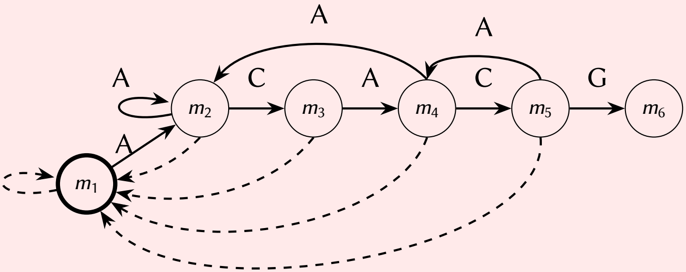
Encode search patterns as an automaton



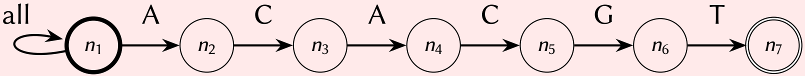


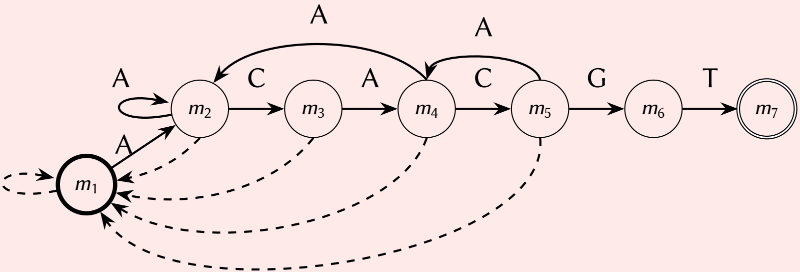
Encode search patterns as an automaton



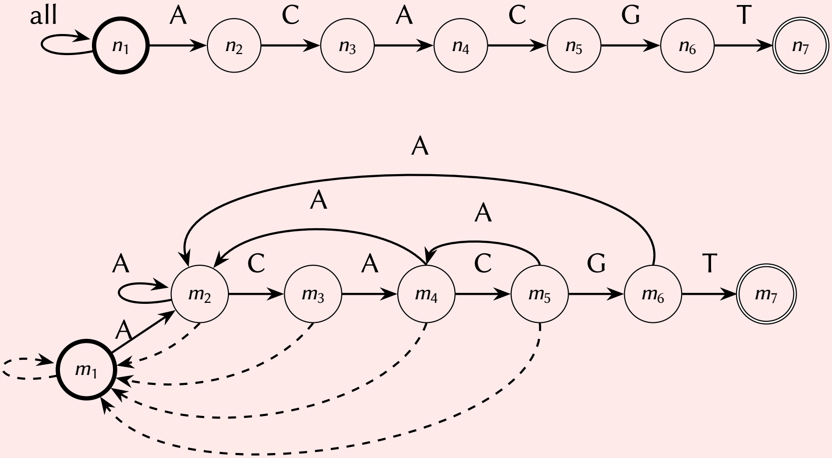


Encode search patterns as an automaton

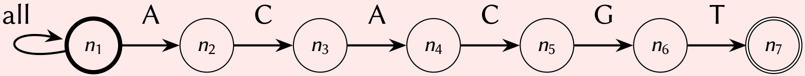


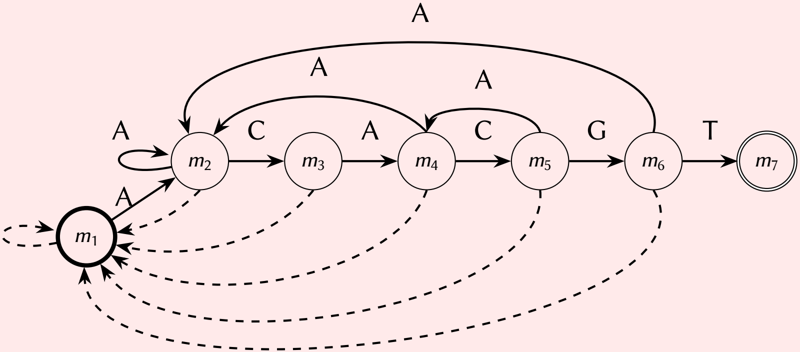


Encode search patterns as an automaton

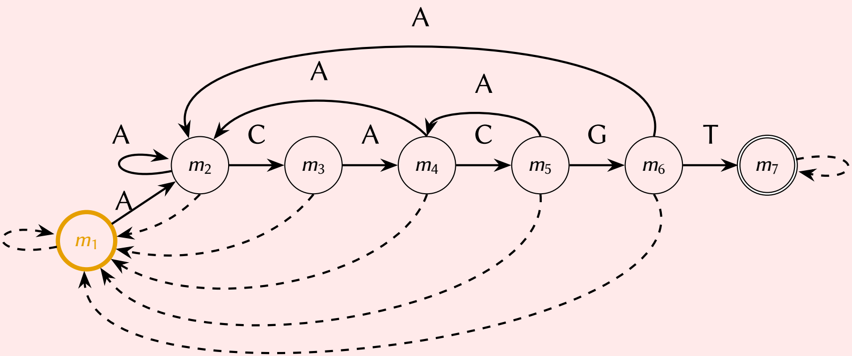


Encode search patterns as an automaton





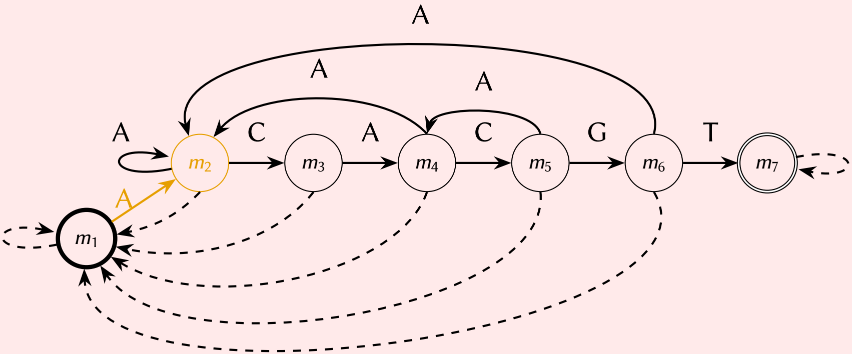
Encode search patterns as an automaton



Searching P = "*ACACGT*"

" A C A C A C A C G T "

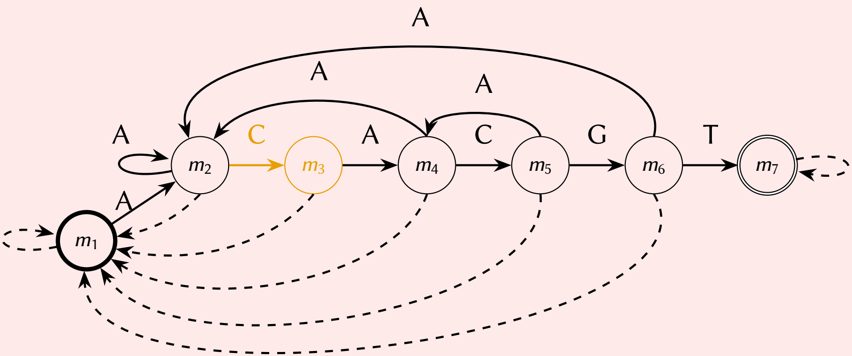
Encode search patterns as an automaton



Searching P = "*ACACGT*"

" A C A C A C A C G T "

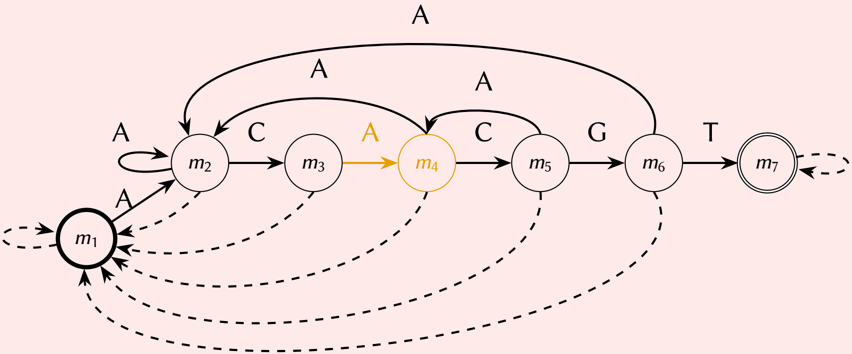
Encode search patterns as an automaton



Searching P = “*ACACGT*”

“ A C A C A C A C G T ”

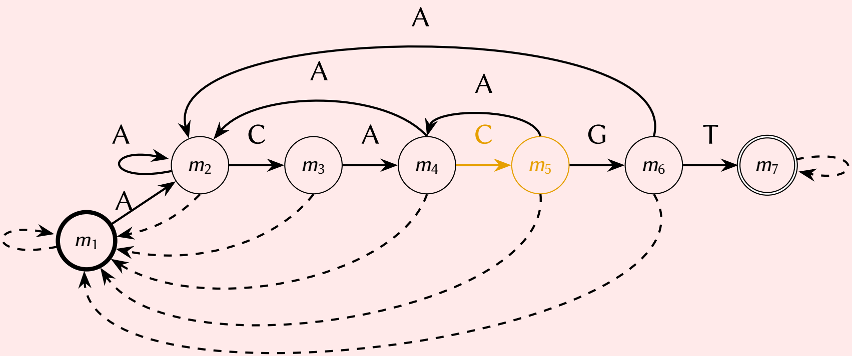
Encode search patterns as an automaton



Searching P = "*ACACGT*"

" A C A C A C A C G T "

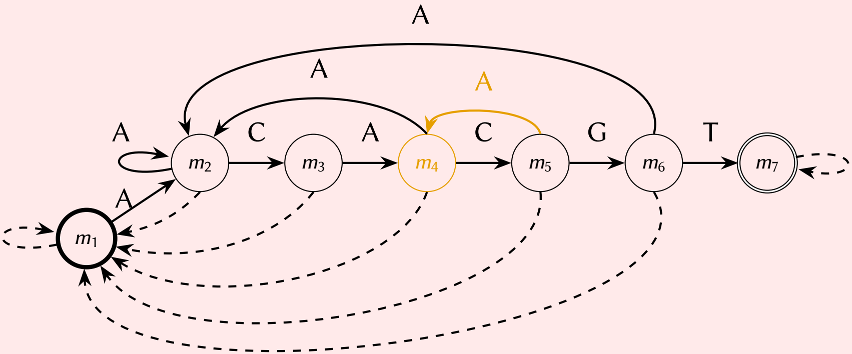
Encode search patterns as an automaton



Searching P = "*ACACGT*"

" A C A C A C A C G T "

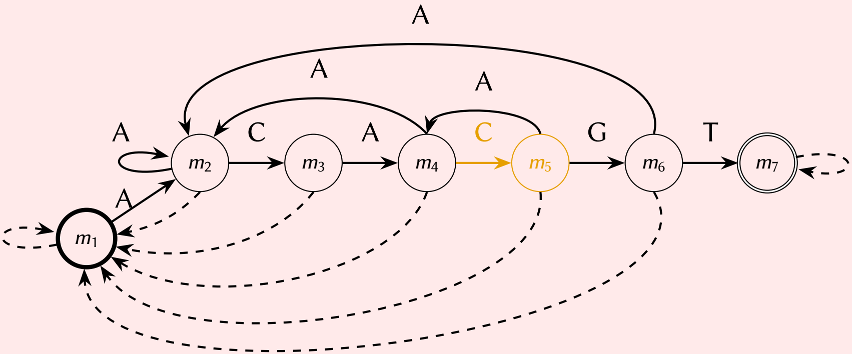
Encode search patterns as an automaton



Searching P = "*ACACGT*"

" A C A C A C A C G T "

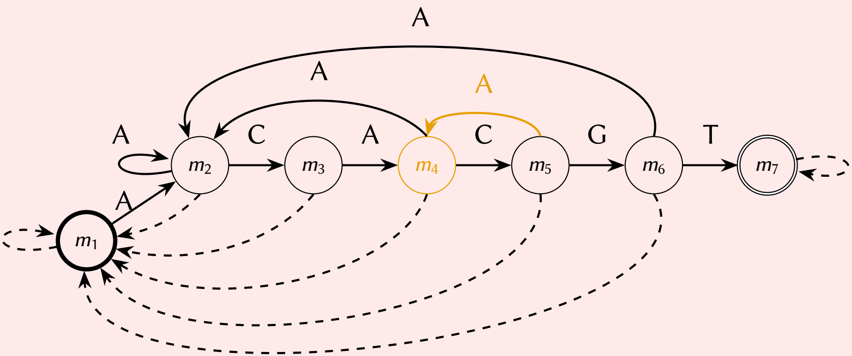
Encode search patterns as an automaton



Searching P = "*ACACGT*"

" A C A C A C A C G T "

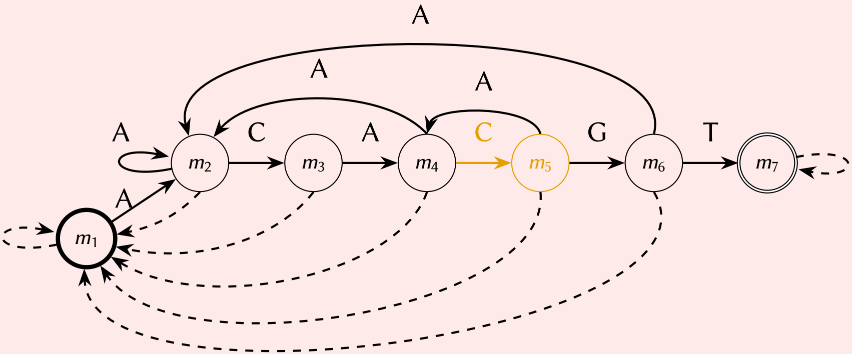
Encode search patterns as an automaton



Searching P = “*ACACGT*”

“ A C A C A C A C G T ”

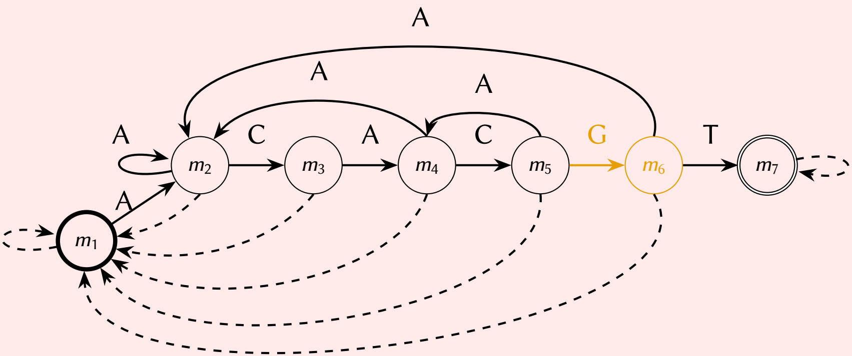
Encode search patterns as an automaton



Searching P = “*ACACGT*”

“ A C A C A C A C G T ”

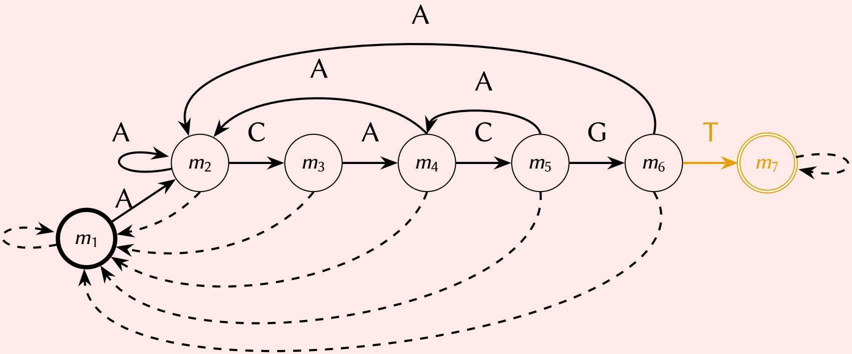
Encode search patterns as an automaton



Searching P = "*ACACGT*"

" A C A C A C A C G T "

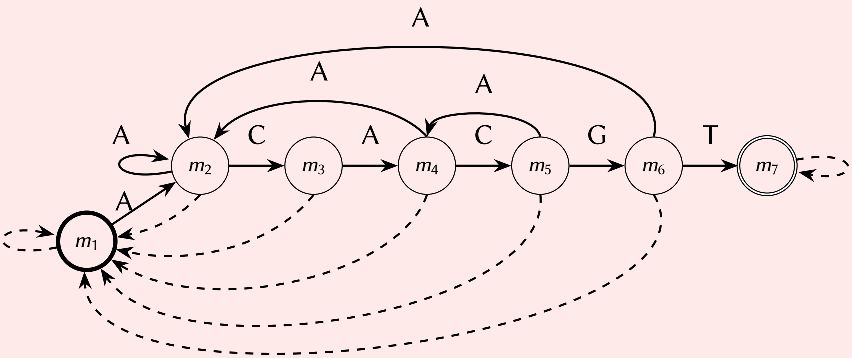
Encode search patterns as an automaton



Searching P = “$ACACGT$”

“ A C A C A C A C G T ”

Encode search patterns as an automaton



Complexity of running a *deterministic* finite automaton with input $\mathcal{S}$

- Always in exactly *one* state.
- We perform at-most $\Theta(|\mathcal{S}|)$ *state transitions* in the automaton.
- Need efficient representation of the *transitions* (per state): e.g., hash table.

More general search patterns: Regular expressions

A *regular expression* describes a set of *strings*.

More general search patterns: Regular expressions

A *regular expression* describes a set of *strings*.

- $\emptyset$ describes $\emptyset$.
- $\sigma \in \mathcal{A}$ describes $\{\sigma\}$.

More general search patterns: Regular expressions

A *regular expression* describes a set of *strings*.

- $\emptyset$ describes $\emptyset$.
- $\sigma \in \mathcal{A}$ describes $\{\sigma\}$.

Now let e, e_1, e_2 be regular expressions describing sets R, R_1, R_2.

- (e) desribes R.
- $e_1 e_2$ describes $\{\textsc{Concatenate}(\mathcal{S}_1, \mathcal{S}_2) \mid \mathcal{S}_1 \in R_1 \wedge \mathcal{S}_2 \in R_2\}$.
- $e_1 \mid e_2$ describes $R_1 \cup R_2$.
- e^* describes any sequence of strings in R.

More general search patterns: Regular expressions

A *regular expression* describes a set of *strings*.

- $\emptyset$ describes $\emptyset$.
- $\sigma \in \mathcal{A}$ describes $\{\sigma\}$.

Now let e, e_1, e_2 be regular expressions describing sets R, R_1, R_2.

- (e) desribes R.
- $e_1 e_2$ describes $\{\textsc{Concatenate}(\mathcal{S}_1, \mathcal{S}_2) \mid \mathcal{S}_1 \in R_1 \land \mathcal{S}_2 \in R_2\}$.
- $e_1 \mid e_2$ describes $R_1 \cup R_2$.
- e^* describes any sequence of strings in R.

Examples

$$\text{moose} \mid \text{mouse}$$

$$\text{sub}^*\text{section}$$

More general search patterns: Regular expressions

A *regular expression* describes a set of *strings*.

- $\emptyset$ describes $\emptyset$.
- $\sigma \in \mathcal{A}$ describes $\{\sigma\}$.

Now let e, e_1, e_2 be regular expressions describing sets R, R_1, R_2.

- (e) desribes R.
- e_1e_2 describes $\{\text{Concatenate}(\mathcal{S}_1, \mathcal{S}_2) \mid \mathcal{S}_1 \in R_1 \wedge \mathcal{S}_2 \in R_2\}$.
- $e_1 \mid e_2$ describes $R_1 \cup R_2$.
- e^* describes any sequence of strings in R.

Examples

$$\text{moose} \mid \text{mouse}$$

$$\text{sub}^*\text{section}$$

Claim: Every regular expression is equivalent to a deterministic finite automaton

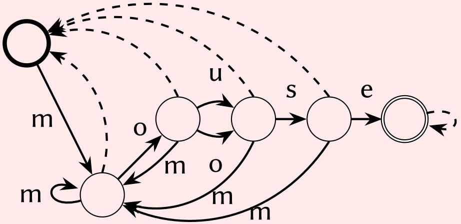
See SFWRENG 2FA3: Discrete Mathematics with Applications II.

More general search patterns: Regular expressions

A *regular expression* describes a set of *strings*.

Claim: Every regular expression is equivalent to a deterministic finite automaton

See SFWRENG 2FA3: Discrete Mathematics with Applications II.

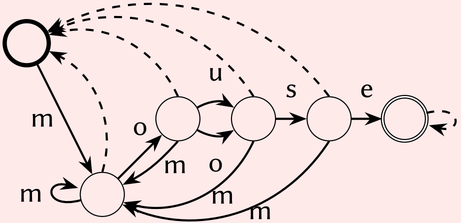


More general search patterns: Regular expressions

A *regular expression* describes a set of *strings*.

Claim: Every regular expression is equivalent to a deterministic finite automaton

See SFWRENG 2FA3: Discrete Mathematics with Applications II.



Deterministic finite automata can grow *very* large

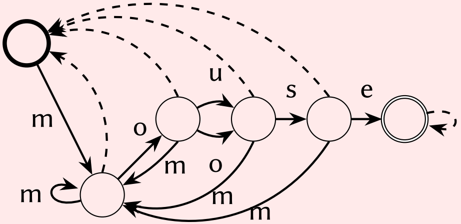
Alternatively, you can run a *nondeterminstic* finite automaton: an automaton with choices!

More general search patterns: Regular expressions

A *regular expression* describes a set of *strings*.

Claim: Every regular expression is equivalent to a deterministic finite automaton

See SFWRENG 2FA3: Discrete Mathematics with Applications II.



Deterministic finite automata can grow *very* large

Alternatively, you can run a *nondeterminstic* finite automaton: an automaton with choices!

Lower costs to construct the automaton, higher costs to run them.

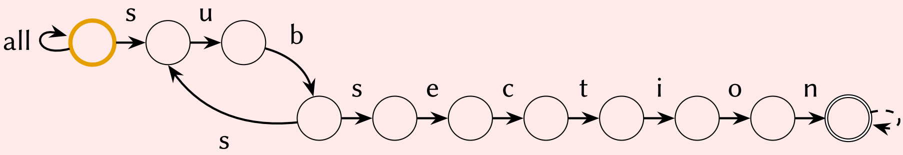
More general search patterns: Regular expressions

A *regular expression* describes a set of *strings*.

Deterministic finite automata can grow *very* large

Alternatively, you can run a *nondeterminstic* finite automaton: an automaton with choices!

Lower costs to construct the automaton, higher costs to run them.



Searching P = “sub*section”

“ a s u b s t r i n g i n a s u b s e c t i o n ”

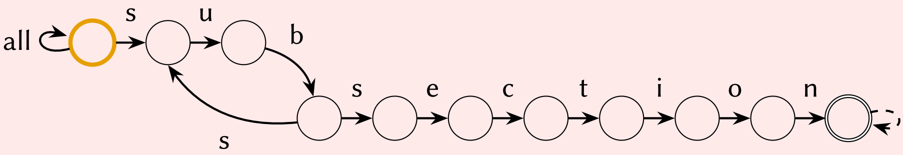
More general search patterns: Regular expressions

A *regular expression* describes a set of *strings*.

Deterministic finite automata can grow *very* large

Alternatively, you can run a *nondeterminstic* finite automaton: an automaton with choices!

Lower costs to construct the automaton, higher costs to run them.



Searching $P =$ "sub*section"

“ a   s u b s t r i n g   i n   a   s u b s e c t i o n ”

Idea. Keep track of the *set of states* we can be in while walking to the string.

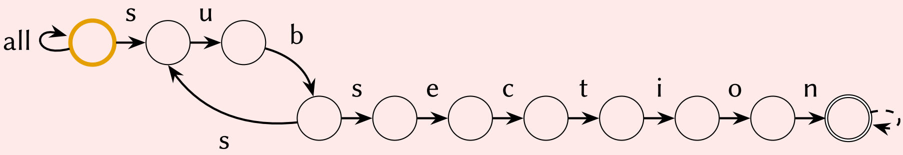
More general search patterns: Regular expressions

A *regular expression* describes a set of *strings*.

Deterministic finite automata can grow *very* large

Alternatively, you can run a *nondeterminstic* finite automaton: an automaton with choices!

Lower costs to construct the automaton, higher costs to run them.



Searching P = "sub*section"

“ a s u b s t r i n g i n a s u b s e c t i o n ”

Idea. Keep track of the *set of states* we can be in while walking to the string.

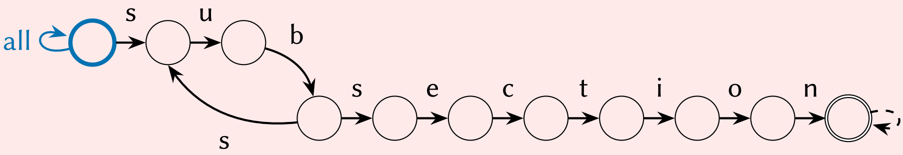
(A *powerset construction* guided by the string one is searching in).

More general search patterns: Regular expressions

A *regular expression* describes a set of *strings*.

Deterministic finite automata can grow *very* large

Alternatively, you can run a *nondeterminstic* finite automaton: an automaton with choices!
Lower costs to construct the automaton, higher costs to run them.



Searching P = "sub*section"

“ a   s u b s t r i n g   i n   a   s u b s e c t i o n ”

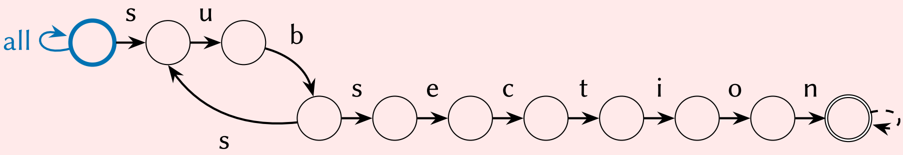
Idea. Keep track of the *set of states* we can be in while walking to the string.
(A *powerset construction* guided by the string one is searching in).

More general search patterns: Regular expressions

A *regular expression* describes a set of *strings*.

Deterministic finite automata can grow *very* large

Alternatively, you can run a *nondeterminstic* finite automaton: an automaton with choices!
Lower costs to construct the automaton, higher costs to run them.



Searching P = "sub*section"

“ a s u b s t r i n g i n a s u b s e c t i o n ”

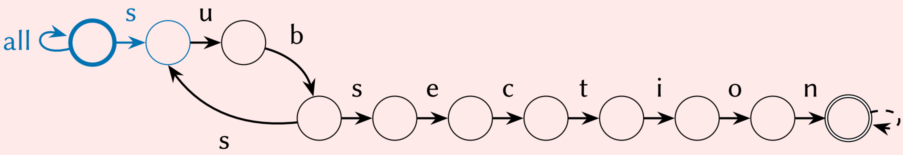
Idea. Keep track of the *set of states* we can be in while walking to the string.
(A *powerset construction* guided by the string one is searching in).

More general search patterns: Regular expressions

A *regular expression* describes a set of *strings*.

Deterministic finite automata can grow *very* large

Alternatively, you can run a *nondeterminstic* finite automaton: an automaton with choices!
Lower costs to construct the automaton, higher costs to run them.



Searching $P =$ "sub*section"

“ a s u b s t r i n g i n a s u b s e c t i o n ”

Idea. Keep track of the *set of states* we can be in while walking to the string.
(A *powerset construction* guided by the string one is searching in).

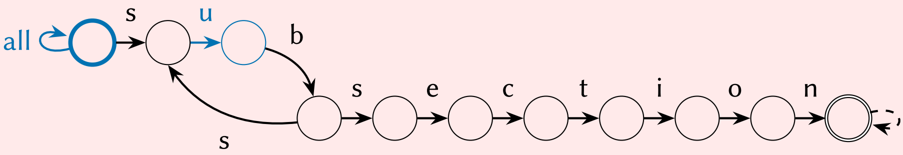
More general search patterns: Regular expressions

A *regular expression* describes a set of *strings*.

Deterministic finite automata can grow *very* large

Alternatively, you can run a *nondeterminstic* finite automaton: an automaton with choices!

Lower costs to construct the automaton, higher costs to run them.



Searching $P =$ "sub*section"

" a s u b s t r i n g i n a s u b s e c t i o n "

Idea. Keep track of the *set of states* we can be in while walking to the string.

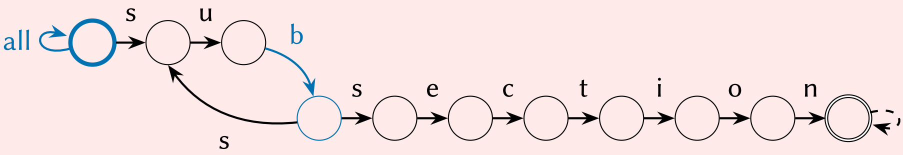
(A *powerset construction* guided by the string one is searching in).

More general search patterns: Regular expressions

A *regular expression* describes a set of *strings*.

Deterministic finite automata can grow *very* large

Alternatively, you can run a *nondeterminstic* finite automaton: an automaton with choices!
Lower costs to construct the automaton, higher costs to run them.



Searching $P =$ "sub*section"

“ a s u b s t r i n g i n a s u b s e c t i o n ”

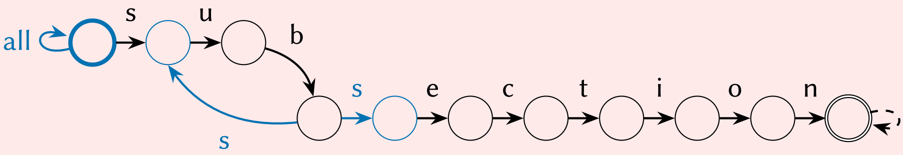
Idea. Keep track of the *set of states* we can be in while walking to the string.
(A *powerset construction* guided by the string one is searching in).

More general search patterns: Regular expressions

A *regular expression* describes a set of *strings*.

Deterministic finite automata can grow *very* large

Alternatively, you can run a *nondeterminstic* finite automaton: an automaton with choices!
Lower costs to construct the automaton, higher costs to run them.



Searching $P =$ "sub*section"

“ a s u b s t r i n g i n a s u b s e c t i o n ”

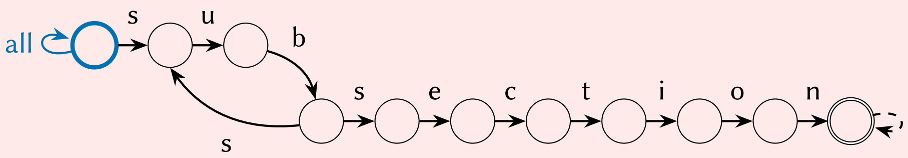
Idea. Keep track of the *set of states* we can be in while walking to the string.
(A *powerset construction* guided by the string one is searching in).

More general search patterns: Regular expressions

A *regular expression* describes a set of *strings*.

Deterministic finite automata can grow *very* large

Alternatively, you can run a *nondeterminstic* finite automaton: an automaton with choices!
Lower costs to construct the automaton, higher costs to run them.



Searching $P =$ “sub*section”

“ a s u b s t r i n g i n a s u b s e c t i o n ”

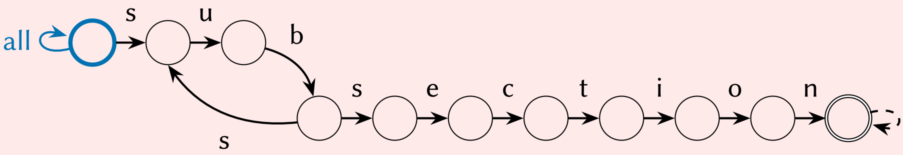
Idea. Keep track of the *set of states* we can be in while walking to the string.
(A *powerset construction* guided by the string one is searching in).

More general search patterns: Regular expressions

A *regular expression* describes a set of *strings*.

Deterministic finite automata can grow *very* large

Alternatively, you can run a *nondeterminstic* finite automaton: an automaton with choices!
Lower costs to construct the automaton, higher costs to run them.



Searching $P =$ "sub*section"

“ a s u b s t r i n g i n a s u b s e c t i o n ”

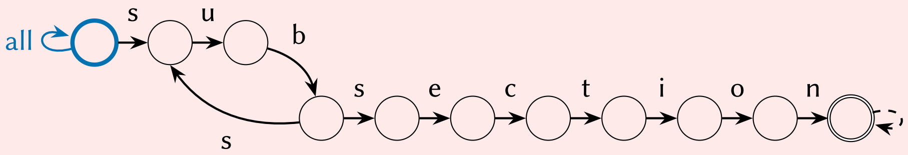
Idea. Keep track of the *set of states* we can be in while walking to the string.
(A *powerset construction* guided by the string one is searching in).

More general search patterns: Regular expressions

A *regular expression* describes a set of *strings*.

Deterministic finite automata can grow *very* large

Alternatively, you can run a *nondeterminstic* finite automaton: an automaton with choices!
Lower costs to construct the automaton, higher costs to run them.



Searching $P =$ "sub*section"

“ a s u b s t r i n g i n a s u b s e c t i o n ”

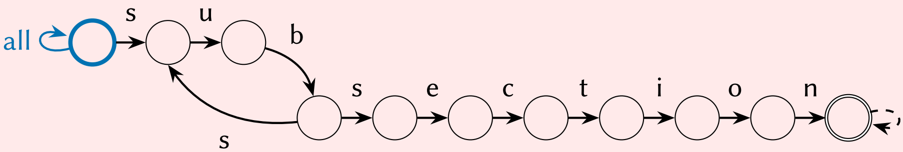
Idea. Keep track of the *set of states* we can be in while walking to the string.
(A *powerset construction* guided by the string one is searching in).

More general search patterns: Regular expressions

A *regular expression* describes a set of *strings*.

Deterministic finite automata can grow *very* large

Alternatively, you can run a *nondeterminstic* finite automaton: an automaton with choices!
Lower costs to construct the automaton, higher costs to run them.



Searching $P =$ "sub*section"

" a s u b s t r i n g i n a s u b s e c t i o n "

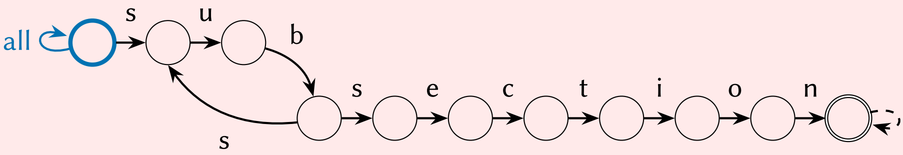
Idea. Keep track of the *set of states* we can be in while walking to the string.
(A *powerset construction* guided by the string one is searching in).

More general search patterns: Regular expressions

A *regular expression* describes a set of *strings*.

Deterministic finite automata can grow *very* large

Alternatively, you can run a *nondeterminstic* finite automaton: an automaton with choices!
Lower costs to construct the automaton, higher costs to run them.



Searching P = "sub*section"

“ a s u b s t r i n g i n a s u b s e c t i o n ”

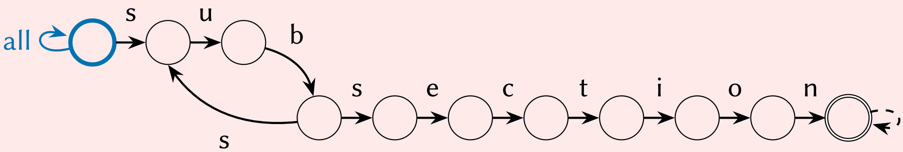
Idea. Keep track of the *set of states* we can be in while walking to the string.
(A *powerset construction* guided by the string one is searching in).

More general search patterns: Regular expressions

A *regular expression* describes a set of *strings*.

Deterministic finite automata can grow *very* large

Alternatively, you can run a *nondeterminstic* finite automaton: an automaton with choices!
Lower costs to construct the automaton, higher costs to run them.



Searching $P =$ "sub*section"

“ a   s u b s t r i n g   i n   a   s u b s e c t i o n ”

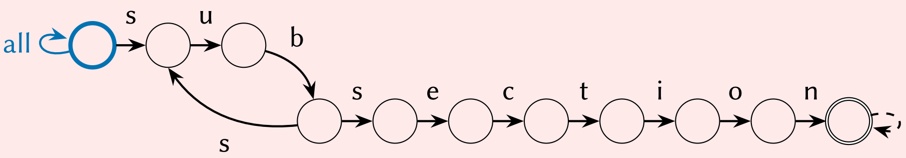
Idea. Keep track of the *set of states* we can be in while walking to the string.
(A *powerset construction* guided by the string one is searching in).

More general search patterns: Regular expressions

A *regular expression* describes a set of *strings*.

Deterministic finite automata can grow *very* large

Alternatively, you can run a *nondeterminstic* finite automaton: an automaton with choices!
Lower costs to construct the automaton, higher costs to run them.



Searching $P =$ "sub*section"

“ a s u b s t r i n g i n a s u b s e c t i o n ”

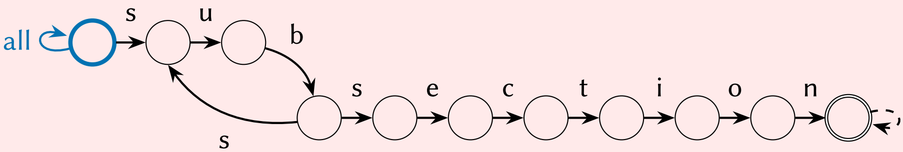
Idea. Keep track of the *set of states* we can be in while walking to the string.
(A *powerset construction* guided by the string one is searching in).

More general search patterns: Regular expressions

A *regular expression* describes a set of *strings*.

Deterministic finite automata can grow *very* large

Alternatively, you can run a *nondeterminstic* finite automaton: an automaton with choices!
Lower costs to construct the automaton, higher costs to run them.



Searching $P =$ "sub*section"

" a s u b s t r i n g i n a s u b s e c t i o n "

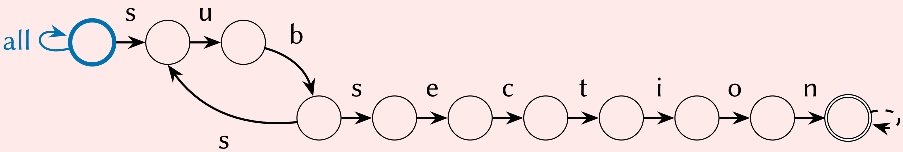
Idea. Keep track of the *set of states* we can be in while walking to the string.
(A *powerset construction* guided by the string one is searching in).

More general search patterns: Regular expressions

A *regular expression* describes a set of *strings*.

Deterministic finite automata can grow *very* large

Alternatively, you can run a *nondeterminstic* finite automaton: an automaton with choices!
Lower costs to construct the automaton, higher costs to run them.



Searching $P =$ "sub*section"

" a s u b s t r i n g i n a s u b s e c t i o n "

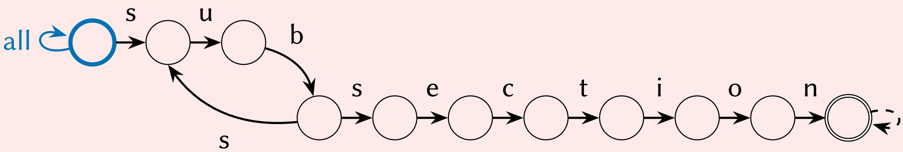
Idea. Keep track of the *set of states* we can be in while walking to the string.
(A *powerset construction* guided by the string one is searching in).

More general search patterns: Regular expressions

A *regular expression* describes a set of *strings*.

Deterministic finite automata can grow *very* large

Alternatively, you can run a *nondeterminstic* finite automaton: an automaton with choices!
Lower costs to construct the automaton, higher costs to run them.



Searching P = "sub*section"

“a substring in a subsection”

Idea. Keep track of the *set of states* we can be in while walking to the string.
(A *powerset construction* guided by the string one is searching in).

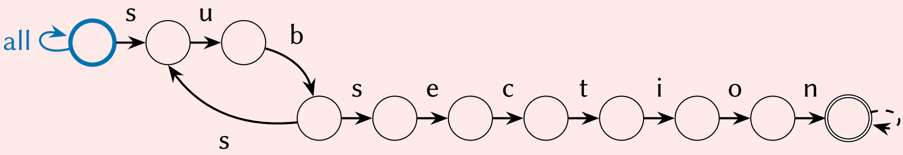
More general search patterns: Regular expressions

A *regular expression* describes a set of *strings*.

Deterministic finite automata can grow *very* large

Alternatively, you can run a *nondeterminstic* finite automaton: an automaton with choices!

Lower costs to construct the automaton, higher costs to run them.



Searching P = "sub*section"

“ a   s u b s t r i n g   i n   a   s u b s e c t i o n ”

Idea. Keep track of the *set of states* we can be in while walking to the string.

(A *powerset construction* guided by the string one is searching in).

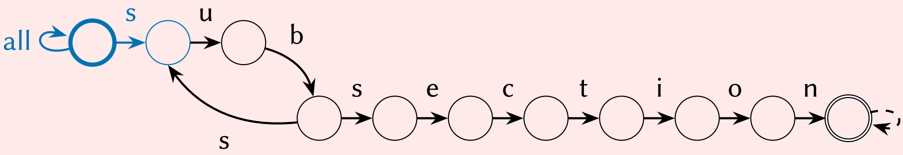
More general search patterns: Regular expressions

A *regular expression* describes a set of *strings*.

Deterministic finite automata can grow *very* large

Alternatively, you can run a *nondeterminstic* finite automaton: an automaton with choices!

Lower costs to construct the automaton, higher costs to run them.



Searching $P =$ "sub*section"

“ a s u b s t r i n g i n a s u b s e c t i o n ”

Idea. Keep track of the *set of states* we can be in while walking to the string.

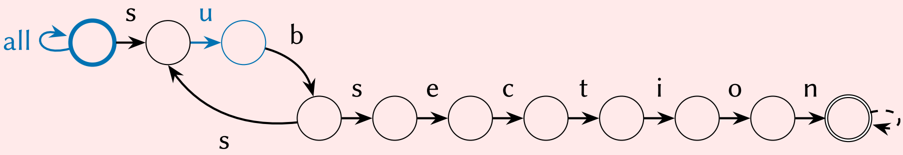
(A *powerset construction* guided by the string one is searching in).

More general search patterns: Regular expressions

A *regular expression* describes a set of *strings*.

Deterministic finite automata can grow *very* large

Alternatively, you can run a *nondeterminstic* finite automaton: an automaton with choices!
Lower costs to construct the automaton, higher costs to run them.



Searching P = "sub*section"

“ a s u b s t r i n g i n a s u b s e c t i o n ”

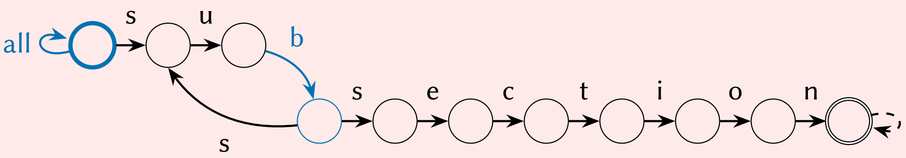
Idea. Keep track of the *set of states* we can be in while walking to the string.
(A *powerset construction* guided by the string one is searching in).

More general search patterns: Regular expressions

A *regular expression* describes a set of *strings*.

Deterministic finite automata can grow *very* large

Alternatively, you can run a *nondeterminstic* finite automaton: an automaton with choices!
Lower costs to construct the automaton, higher costs to run them.



Searching $P =$ "sub*section"

“ a   s u b s t r i n g   i n   a   s u b s e c t i o n ”

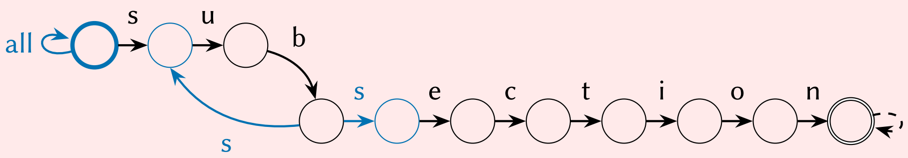
Idea. Keep track of the *set of states* we can be in while walking to the string.
(A *powerset construction* guided by the string one is searching in).

More general search patterns: Regular expressions

A *regular expression* describes a set of *strings*.

Deterministic finite automata can grow *very* large

Alternatively, you can run a *nondeterminstic* finite automaton: an automaton with choices!
Lower costs to construct the automaton, higher costs to run them.



Searching $P =$ "sub*section"

“ a   s u b s t r i n g   i n   a   s u b s e c t i o n ”

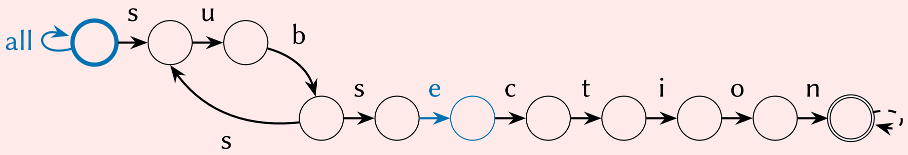
Idea. Keep track of the *set of states* we can be in while walking to the string.
(A *powerset construction* guided by the string one is searching in).

More general search patterns: Regular expressions

A *regular expression* describes a set of *strings*.

Deterministic finite automata can grow *very* large

Alternatively, you can run a *nondeterminstic* finite automaton: an automaton with choices!
Lower costs to construct the automaton, higher costs to run them.



Searching P = "sub*section"

“ a s u b s t r i n g i n a s u b s e c t i o n ”

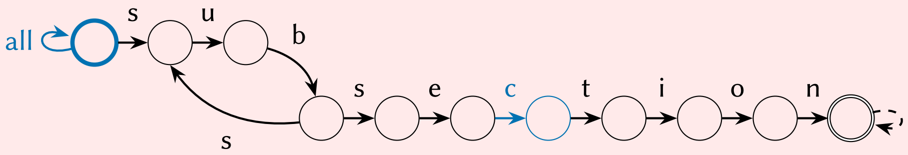
Idea. Keep track of the *set of states* we can be in while walking to the string.
(A *powerset construction* guided by the string one is searching in).

More general search patterns: Regular expressions

A *regular expression* describes a set of *strings*.

Deterministic finite automata can grow *very* large

Alternatively, you can run a *nondeterminstic* finite automaton: an automaton with choices!
Lower costs to construct the automaton, higher costs to run them.



Searching $P =$ "sub*section"

"a substring in a subsection"

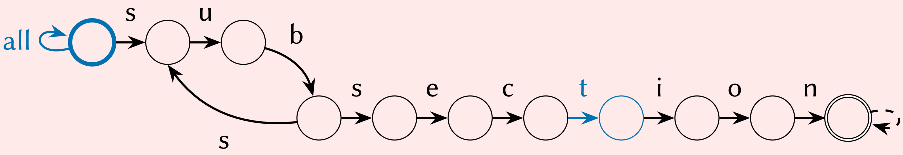
Idea. Keep track of the *set of states* we can be in while walking to the string.
(A *powerset construction* guided by the string one is searching in).

More general search patterns: Regular expressions

A *regular expression* describes a set of *strings*.

Deterministic finite automata can grow *very* large

Alternatively, you can run a *nondeterminstic* finite automaton: an automaton with choices!
Lower costs to construct the automaton, higher costs to run them.



Searching $P =$ "sub*section"

" a   s u b s t r i n g   i n   a   s u b s e c t i o n "

Idea. Keep track of the *set of states* we can be in while walking to the string.
(A *powerset construction* guided by the string one is searching in).

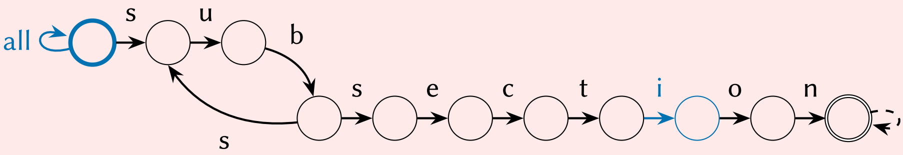
More general search patterns: Regular expressions

A *regular expression* describes a set of *strings*.

Deterministic finite automata can grow *very* large

Alternatively, you can run a *nondeterminstic* finite automaton: an automaton with choices!

Lower costs to construct the automaton, higher costs to run them.



Searching P = "sub*section"

“ a s u b s t r i n g i n a s u b s e c t i o n ”

Idea. Keep track of the *set of states* we can be in while walking to the string.
(A *powerset construction* guided by the string one is searching in).

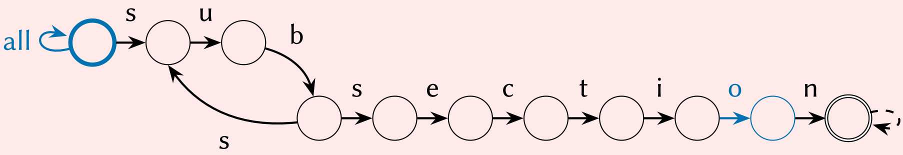
More general search patterns: Regular expressions

A *regular expression* describes a set of *strings*.

Deterministic finite automata can grow *very* large

Alternatively, you can run a *nondeterminstic* finite automaton: an automaton with choices!

Lower costs to construct the automaton, higher costs to run them.



Searching P = "sub*section"

" a s u b s t r i n g i n a s u b s e c t i o n "

Idea. Keep track of the *set of states* we can be in while walking to the string.

(A *powerset construction* guided by the string one is searching in).

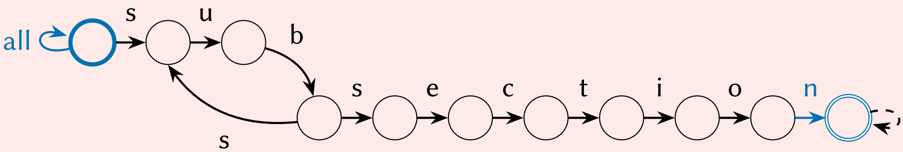
More general search patterns: Regular expressions

A *regular expression* describes a set of *strings*.

Deterministic finite automata can grow *very* large

Alternatively, you can run a *nondeterminstic* finite automaton: an automaton with choices!

Lower costs to construct the automaton, higher costs to run them.



Searching P = “sub*section”

“ a s u b s t r i n g i n a s u b s e c t i o n ”

Idea. Keep track of the *set of states* we can be in while walking to the string.

(A *powerset construction* guided by the string one is searching in).

More general search patterns: Regular expressions

A *regular expression* describes a set of *strings*.

Final remarks on regular expressions

- Running an N-state deterministic finite automaton on $\mathcal{S}$ can be done in $\Theta(|\mathcal{S}|)$.
- Running an N-state nondeterministic finite automaton on $\mathcal{S}$ can be done in $\Theta(N|\mathcal{S}|)$.

More general search patterns: Regular expressions

A *regular expression* describes a set of *strings*.

Final remarks on regular expressions

- Running an N-state deterministic finite automaton on $\mathcal{S}$ can be done in $\Theta(|\mathcal{S}|)$.
- Running an N-state nondeterministic finite automaton on $\mathcal{S}$ can be done in $\Theta(N|\mathcal{S}|)$.
- For single words, a deterministic finite automaton can be easily constructed: see the Knuth-Morris-Pratt algorithm in the book.

More general search patterns: Regular expressions

A *regular expression* describes a set of *strings*.

Final remarks on regular expressions

- Running an N-state deterministic finite automaton on $\mathcal{S}$ can be done in $\Theta(|\mathcal{S}|)$.
- Running an N-state nondeterministic finite automaton on $\mathcal{S}$ can be done in $\Theta(N|\mathcal{S}|)$.
- For single words, a deterministic finite automaton can be easily constructed: see the Knuth-Morris-Pratt algorithm in the book.
- Automata-based searching in string can be very fast: core component in lexers, parsers, and compilers.

More general search patterns: Regular expressions

A *regular expression* describes a set of *strings*.

Final remarks on regular expressions

- Running an N-state deterministic finite automaton on $\mathcal{S}$ can be done in $\Theta(|\mathcal{S}|)$.
- Running an N-state nondeterministic finite automaton on $\mathcal{S}$ can be done in $\Theta(N|\mathcal{S}|)$.
- For single words, a deterministic finite automaton can be easily constructed: see the Knuth-Morris-Pratt algorithm in the book.
- Automata-based searching in string can be very fast: core component in lexers, parsers, and compilers.
- Many *RegExp* libraries are *regular expression like*: they support non-regular features. Consequently, many such libraries use *shamefully slow* backtracking algorithms: Worst-case exponential complexity, even when searching for *simple patterns*.

Substring search in less-than $\Theta(|\mathcal{S}|)$ steps

Searching $P =$ "*great*"

```
“a n   e x a m p l e   o f   w o r d s”
 g r e a t
```

How can I skip checking most letters in $\mathcal{S}$?

Substring search in less-than $\Theta(|\mathcal{S}|)$ steps

Searching $P =$ "*great*"

```
“ a n   e x a m p l e   o f   w o r d s ”
  g r e a t
```

How can I skip checking most letters in $\mathcal{S}$?

Assume we checked up-till position i.

Observation. If we compare the last symbol from our pattern with $S[i + |P| - 1]$, then

- We see a symbol that is not even in P: *P cannot occur in $S[i \ldots i + |P|)$*.

Substring search in less-than $\Theta(|\mathcal{S}|)$ steps

Searching $P =$ "*great*"

```
“ a n   e x a m p l e   o f   w o r d s ”
              g r e a t
```

How can I skip checking most letters in $\mathcal{S}$?

Assume we checked up-till position i.

Observation. If we compare the last symbol from our pattern with $S[i + |P| - 1]$, then

- We see a symbol that is not even in P: *P cannot occur in $S[i \dots i + |P|)$*.
- We see a symbol that is in P: *P could have started somewhere in $S[i \dots i + |P|)$*.

Substring search in less-than $\Theta(|\mathcal{S}|)$ steps

Searching $P =$ "*great*"

```
“ a n   e x a m p l e   o f   w o r d s ”
                    g r e a t
```

How can I skip checking most letters in $\mathcal{S}$?

Assume we checked up-till position i.

Observation. If we compare the last symbol from our pattern with $S[i + |P| - 1]$, then

- We see a symbol that is not even in P: *P cannot occur in $S[i \ldots i + |P|)$*.
- We see a symbol that is in P: *P could have started somewhere in $S[i \ldots i + |P|)$*.
  Based on the symbol, decide where to start looking.

Substring search in less-than $\Theta(|\mathcal{S}|)$ steps

Searching $P = $ "*great*"

```
“an example of words”
            great
```

How can I skip checking most letters in $\mathcal{S}$?

Assume we checked up-till position i.

Observation. If we compare the last symbol from our pattern with $S[i + |P| - 1]$, then

- We see a symbol that is not even in P: *P cannot occur in $S[i \dots i + |P|)$*.
- We see a symbol that is in P: *P could have started somewhere in $S[i \dots i + |P|)$*.
  Based on the symbol, decide where to start looking.

Substring search in less-than $\Theta(|\mathcal{S}|)$ steps

Searching $P =$ "*great*"

```
“an example of words”
              great
```

How can I skip checking most letters in $\mathcal{S}$?

Assume we checked up-till position i.

Observation. If we compare the last symbol from our pattern with $S[i + |P| - 1]$, then

- We see a symbol that is not even in P: *P cannot occur in $S[i \dots i + |P|)$*.
- We see a symbol that is in P: *P could have started somewhere in $S[i \dots i + |P|)$*.
  Based on the symbol, decide where to start looking.

Substring search in less-than $\Theta(|\mathcal{S}|)$ steps

Searching $P =$ "*great*"

```
“ a n   e x a m p l e   o f   w o r d s ”
                                g r e a t
```

How can I skip checking most letters in $\mathcal{S}$?

Assume we checked up-till position i.

Observation. If we compare the last symbol from our pattern with $S[i + |P| - 1]$, then

- We see a symbol that is not even in P: *P cannot occur in $S[i \dots i + |P|)$*.
- We see a symbol that is in P: *P could have started somewhere in $S[i \dots i + |P|)$*.
  Based on the symbol, decide where to start looking.
- If the symbols match (never in this example): we might have found pattern P.
  Inspect from the end-of-P to the begin to check.

Substring search in less-than $\Theta(|\mathcal{S}|)$ steps

Searching $P =$ "*great*"

```
“ a n   e x a m p l e   o f   w o r d s ”
                                g r e a t
```

How can I skip checking most letters in $\mathcal{S}$?

Assume we checked up-till position i.

Observation. If we compare the last symbol from our pattern with $S[i + |P| - 1]$, then

- We see a symbol that is not even in P: *P cannot occur in $S[i \dots i + |P|)$*.
- We see a symbol that is in P: *P could have started somewhere in $S[i \dots i + |P|)$*.
  Based on the symbol, decide where to start looking.
- If the symbols match (never in this example): we might have found pattern P.
  Inspect from the end-of-P to the begin to check.
  When it does not matches: another opportunity to jump!

Substring search in less-than $\Theta(|\mathcal{S}|)$ steps

Searching $P =$ "*great*"

```
“ a n   e x a m p l e   o f   w o r d s ”
                                g r e a t
```

How can I skip checking most letters in $\mathcal{S}$?

With some preprocessing on P one can precompute how to jump around optimally: the Boyer-Moore algorithm.