Sparse Suffix Sorting

Tomohiro I

Suffix sorting

- Sorting suffixes of a string T is a fundamental task having a lot of applications such as text indexing.

Suffix sorting

- Sorting suffixes of a string T is a fundamental task having a lot of applications such as text indexing.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
T =	l	i	o	n	l	y	i	n	g	o	n	l	y	o	n	l	y	o	n	$

Suffix sorting

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
$T =$	l	i	o	n	l	y	i	n	g	o	n	l	y	o	n	l	y	o	n	$

```
lionlyingonlyonlyon$
 ionlyingonlyonlyon$
  onlyingonlyonlyon$
   nlyingonlyonlyon$
    lyingonlyonlyon$
     yingonlyonlyon$
      ingonlyonlyon$
       ngonlyonlyon$
        gonlyonlyon$
         onlyonlyon$
          nlyonlyon$
           lyonlyon$
            yonlyon$
             onlyon$
              nlyon$
               lyon$
                yon$
                 on$
                  n$
                   $
```

sort

```
20 $
 9 gonlyonlyon$
 7 ingonlyonlyon$
 2 ionlyingonlyonlyon$
 1 lionlyingonlyonlyon$
 5 lyingonlyonlyon$
16 lyon$
12 lyonlyon$
19 n$
 8 ngonlyonlyon$
 4 nlyingonlyonlyon$
15 nlyon$
11 nlyonlyon$
18 on$
 3 onlyingonlyonlyon$
14 onlyon$
10 onlyonlyon$
 6 yingonlyonlyon$
17 yon$
13 yonlyon$
```

all suffixes of T

Suffix sorting

1	2	3	4	5	6	7	8	9	10	11	12	13	…
T =	l	i	o	n	l	y	i	n	g	o	n	l	…

Suffix Array of T

```
lionlyingonlyonlyon$
 ionlyingonlyonlyon$
  onlyingonlyonlyon$
   nlyingonlyonlyon$
    lyingonlyonlyon$
     yingonlyonlyon$
      ingonlyonlyon$
       ngonlyonlyon$
        gonlyonlyon$
         onlyonlyon$
          nlyonlyon$
           lyonlyon$
            yonlyon$
             onlyon$
              nlyon$
               lyon$
                yon$
                 on$
                  n$
                   $
```

sort

20	$
9	gonlyonlyon$
7	ingonlyonlyon$
2	ionlyingonlyonlyon$
1	lionlyingonlyonlyon$
5	lyingonlyonlyon$
16	lyon$
12	lyonlyon$
19	n$
8	ngonlyonlyon$
4	nlyingonlyonlyon$
15	nlyon$
11	nlyonlyon$
18	on$
3	onlyingonlyonlyon$
14	onlyon$
10	onlyonlyon$
6	yingonlyonlyon$
17	yon$
13	yonlyon$

all suffixes of T

Suffix sorting

	1	2	3	4	5	6	7	8	9	10	11	12	13	…
$T =$	l	i	o	n	l	y	i	n	g	o	n	l	y	…

Suffix Array of T

```
lionlyingonlyonlyon$
 ionlyingonlyonlyon$
  onlyingonlyonlyon$
   nlyingonlyonlyon$
    lyingonlyonlyon$
     yingonlyonlyon$
      ingonlyonlyon$
       ngonlyonlyon$
        gonlyonlyon$
         onlyonlyon$
          nlyonlyon$
           lyonlyon$
            yonlyon$
             onlyon$
              nlyon$
               lyon$
                yon$
                 on$
                  n$
                   $
```

20	$
9	gonlyonlyon$
7	ingonlyonlyon$
2	ionlyingonlyonlyon$
1	lionlyingonlyonlyon$
5	lyingonlyonlyon$
16	lyon$
12	lyonlyon$
19	n$
8	ngonlyonlyon$
4	nlyingonlyonlyon$
15	nlyon$
11	nlyonlyon$
18	on$
3	onlyingonlyonlyon$
14	onlyon$
10	onlyonlyon$
6	yingonlyonlyon$
17	yon$
13	yonlyon$

Suffix sorting

Suffix Array of T

1	2	3	4	5	6	7	8	9	10	11	12	13	…							
T =	l	i	o	n	l	y	i	n	g	o	n	l	y	o	n	l	y	o	n	$

```
lionlyingonlyonlyon$
 ionlyingonlyonlyon$
  onlyingonlyonlyon$
   nlyingonlyonlyon$
    lyingonlyonlyon$
     yingonlyonly
      ingonlyon
       ngonlyon
        gonlyonly
         onlyonlyon$
          nlyonlyon$
           lyonlyon$
            yonlyon$
             onlyon$
              nlyon$
               lyon$
                yon$
                 on$
                  n$
                   $
```

20	$
9	gonlyonlyon$
7	ingonlyonlyon$
2	ionlyingonlyonlyon$
1	lionlyingonlyonlyon$
5	lyingonlyonlyon$
16	lyon$
12	lyonlyon$
19	n$
8	ngonlyonlyon$
4	nlyingonlyonlyon$
15	nlyon$
11	nlyonlyon$
18	on$
3	onlyingonlyonlyon$
14	onlyon$
10	onlyonlyon$
6	yingonlyonlyon$
17	yon$
13	yonlyon$

Suffix sorting

Suffix Array of T

1	2	3	4	5	6	7	8	9	10	11	12	13	…
T =	l	i	o	n	l	y	i	n	g	o	n	l	y	o	n	l	y	o	n	$

I found lyon in 3 steps!

20	$
9	gonlyonlyon$
7	ingonlyonlyon$
2	ionlyingonlyonlyon$
1	lionlyingonlyonlyon$
5	lyingonlyonlyon$
16	lyon$
12	lyonlyon$
19	n$
8	ngonlyonlyon$
4	nlyingonlyonlyon$
15	nlyon$
11	nlyonlyon$
18	on$
3	onlyingonlyonlyon$
14	onlyon$
10	onlyonlyon$
6	yingonlyonlyon$
17	yon$
13	yonlyon$

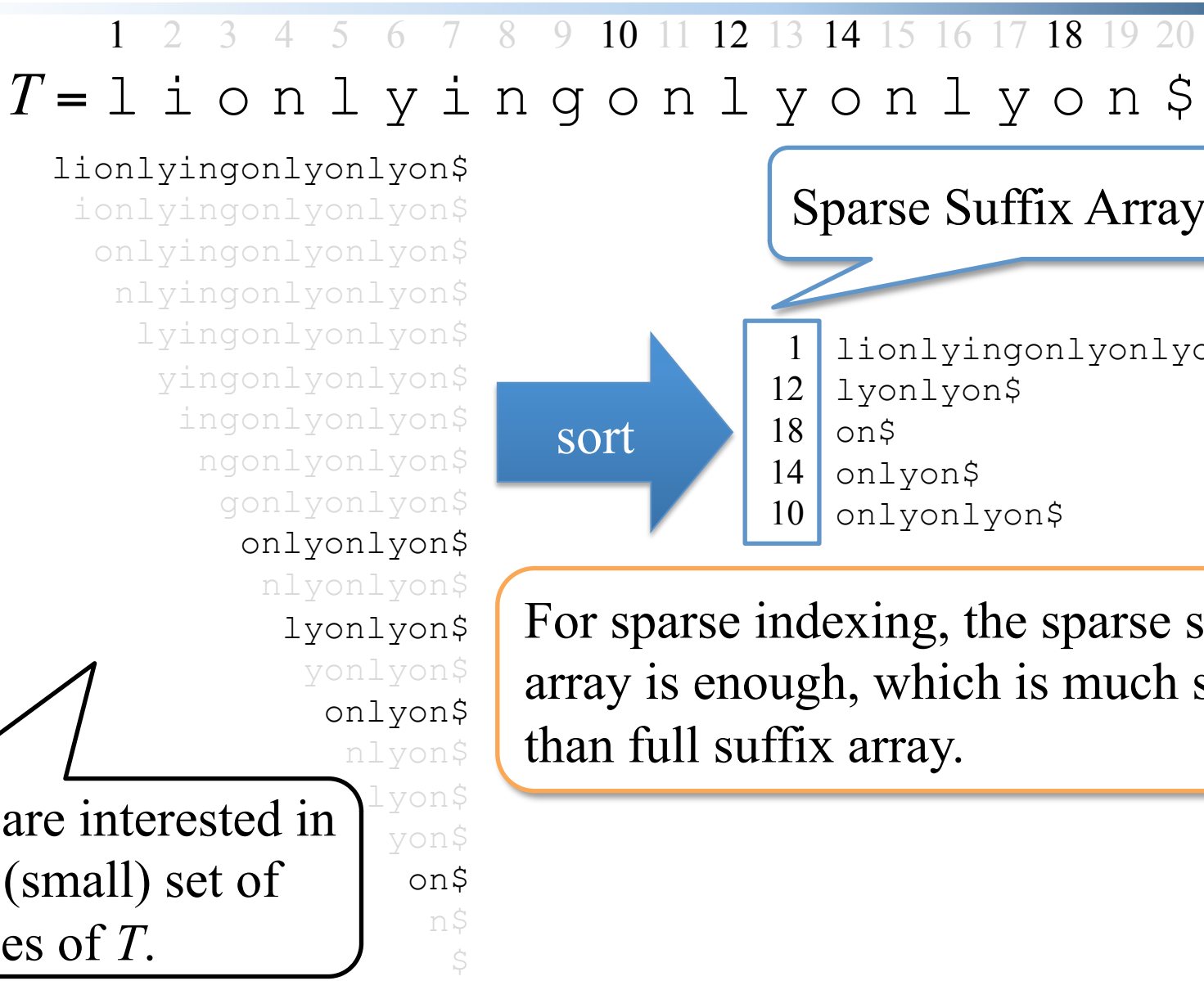
Sparse suffix sorting

```
                1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
T = l i o n l y i n g o n l y o n l y o n $
```

```
lionlyingonlyonlyon$
 ionlyingonlyonlyon$
  onlyingonlyonlyon$
   nlyingonlyonlyon$
    lyingonlyonlyon$
     yingonlyonlyon$
      ingonlyonlyon$
       ngonlyonlyon$
        gonlyonlyon$
         onlyonlyon$
          nlyonlyon$
           lyonlyon$
            yonlyon$
             onlyon$
              nlyon$
               lyon$
                yon$
                 on$
                  n$
                   $
```

If we are interested in some (small) set of suffixes of T.

Sparse suffix sorting



Problem

Given T of length n and set of b positions,
sort the designated suffixes.

- It can be solved in $O(n)$ time by computing full suffix array and make it sparse, but it needs $O(n)$ space in addition to the text.
- An alternative way is to use any sorting algorithm for a set of strings (not for suffixes) that uses $O(b)$ additional space, but it takes $\Omega(bn)$ time.

Question

Using $O(s)$ additional space for some $s \in [b, n]$,
how fast can we sort sparse suffixes?

Monte-Carlo Algorithm

Randomized algorithms that may return a wrong output with low probability.

We can win the gamble with high probability!!

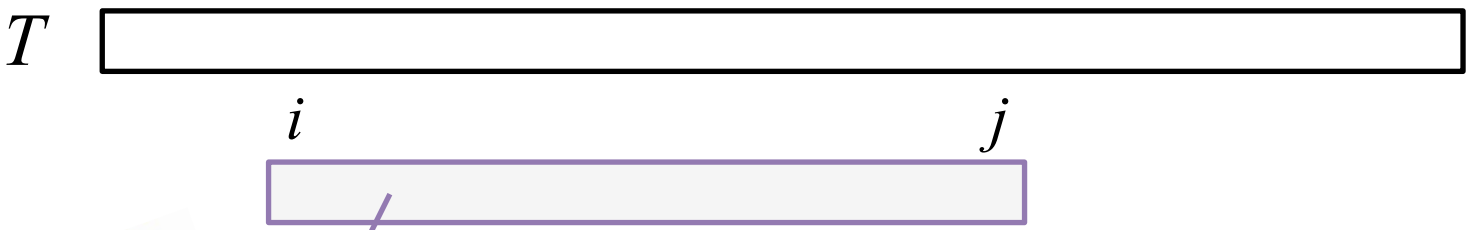
Monte-Carlo Algorithm

Randomized algorithms that may return a wrong output with low probability.

We can win the gamble with high probability!!

- Monte-Carlo algorithm using Karp-Rabin fingerprints.
  - Time-space tradeoffs: for any $s \in [b, n]$ $O(n+(bn/s)\log s))$ time using $O(s)$ additional space.
  - When $s = b$, the time complexity is $O(n \log b)$.
  - When $s = b \log b$, the time complexity is $O(n)$.

Karp-Rabin fingerprints (rolling hash) [Karp and Rabin, `87]

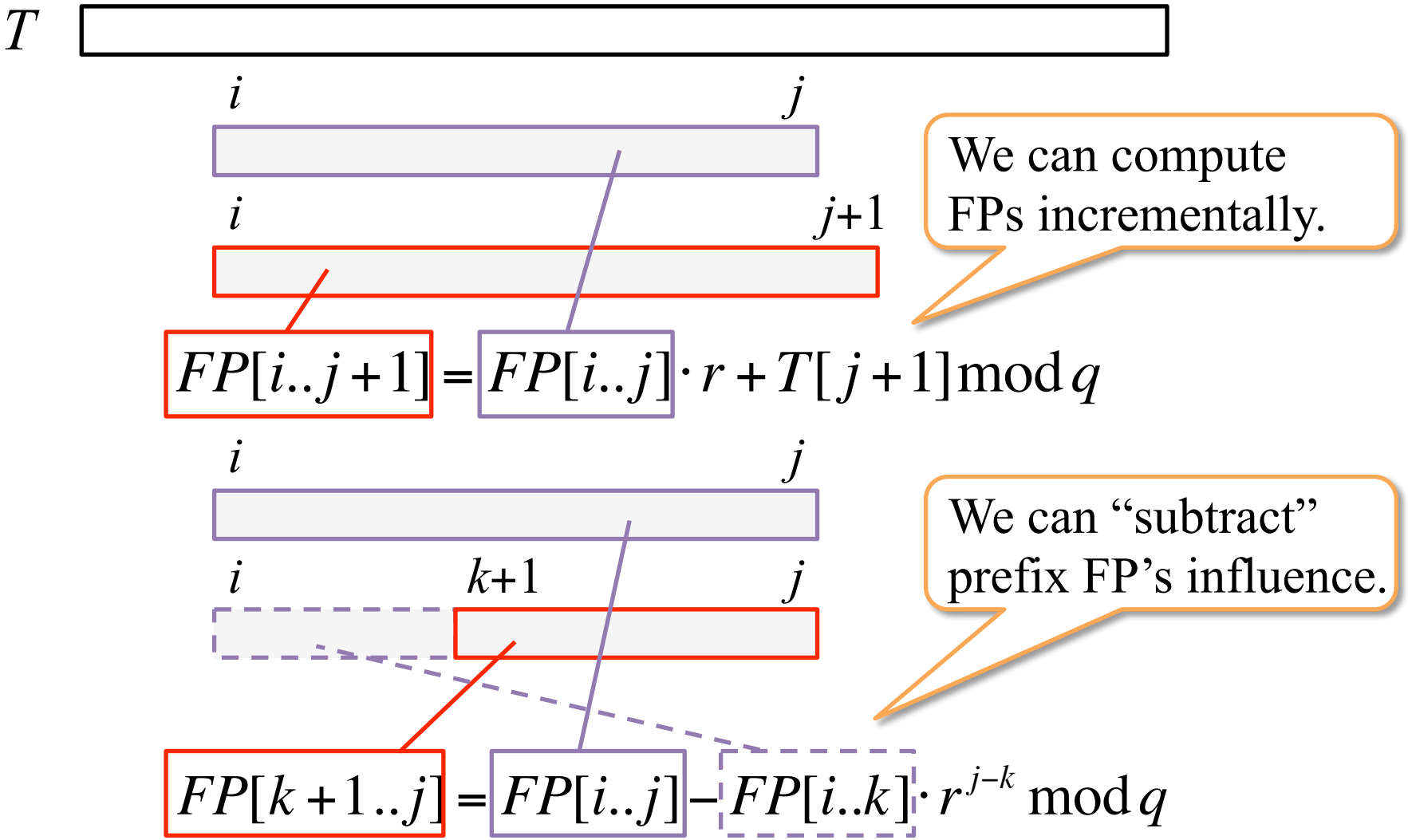


$$FP[i..j] = \sum_{k=i}^{j} T[k] \cdot r^{j-k} \bmod q$$

- q is a prime number.
- r is chosen from [0, q-1] uniformly at random.

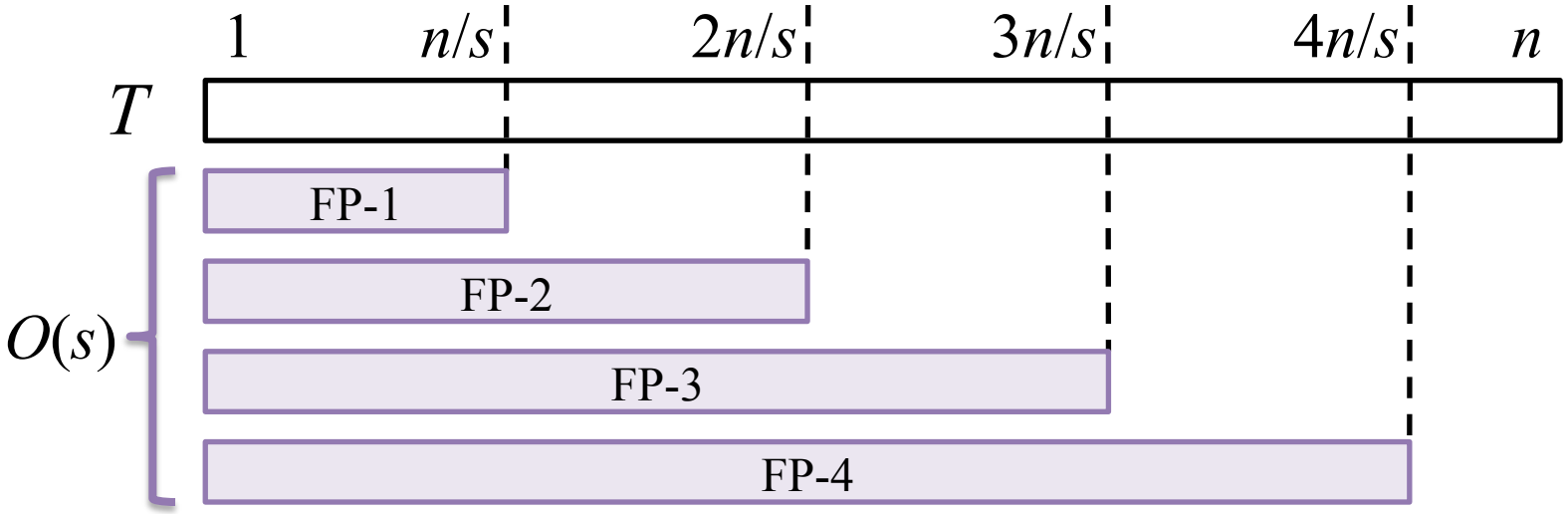
For sufficiently large q, the hash function becomes perfect with high probability, i.e., $T[i..i+l] = T[i'..i'+l]$ iff $FP[i..i+l] = FP[i'..i'+l]$.

Karp-Rabin fingerprints (rolling hash)
[Karp and Rabin, `87]



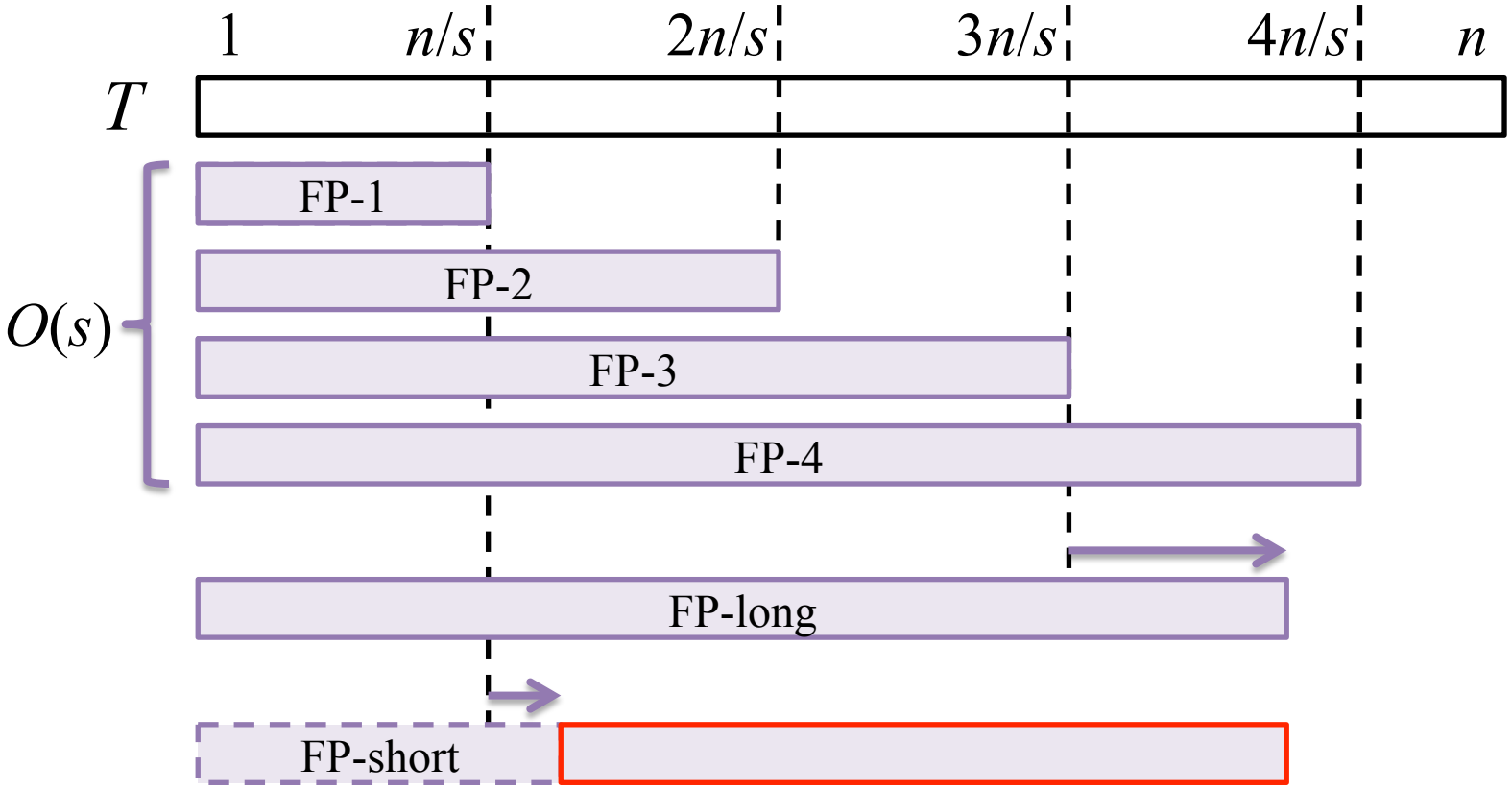
Lemma

We can preprocess T in $O(n)$ time and $O(s)$ space so that the fingerprint for any substring of length l can be computed in $O(\min\{l, n/s\})$ time.



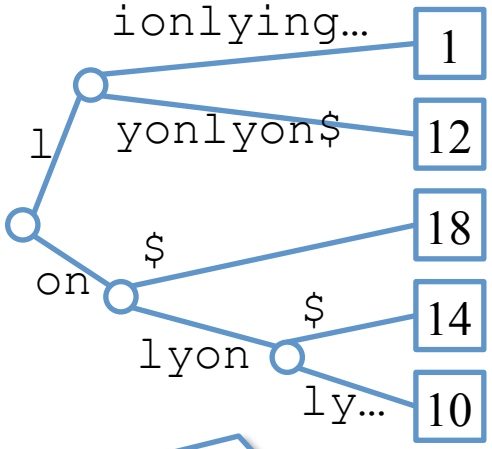
Lemma

We can preprocess T in $O(n)$ time and $O(s)$ space so that the fingerprint for any substring of length l can be computed in $O(\min\{l, n/s\})$ time.



Monte-Carlo algorithm

- The algorithm constructs sparse suffix tree.



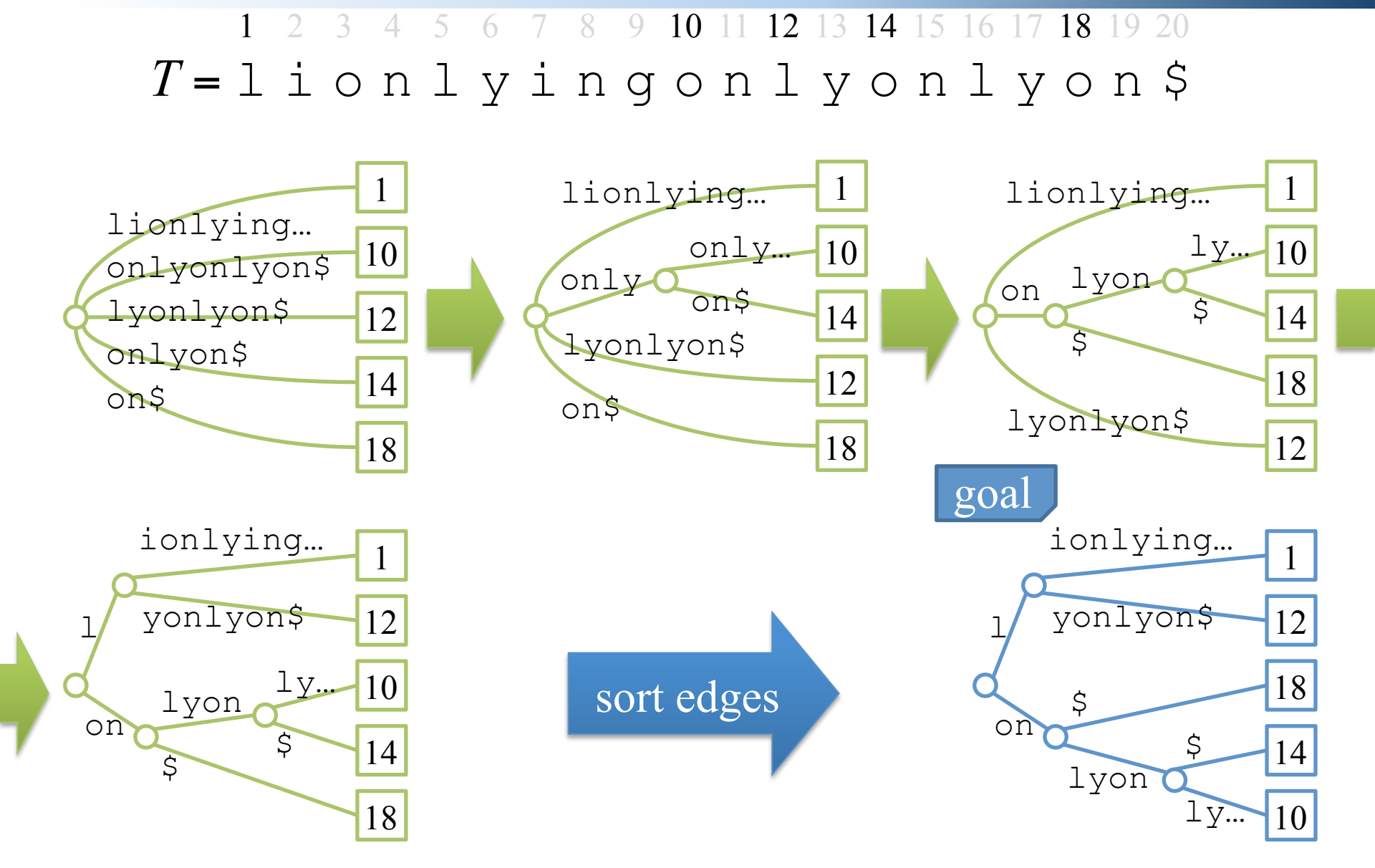
Sparse Suffix Array

1	lionlyingonlyonlyon$
12	lyonlyon$
18	on$
14	onlyon$
10	onlyonlyon$

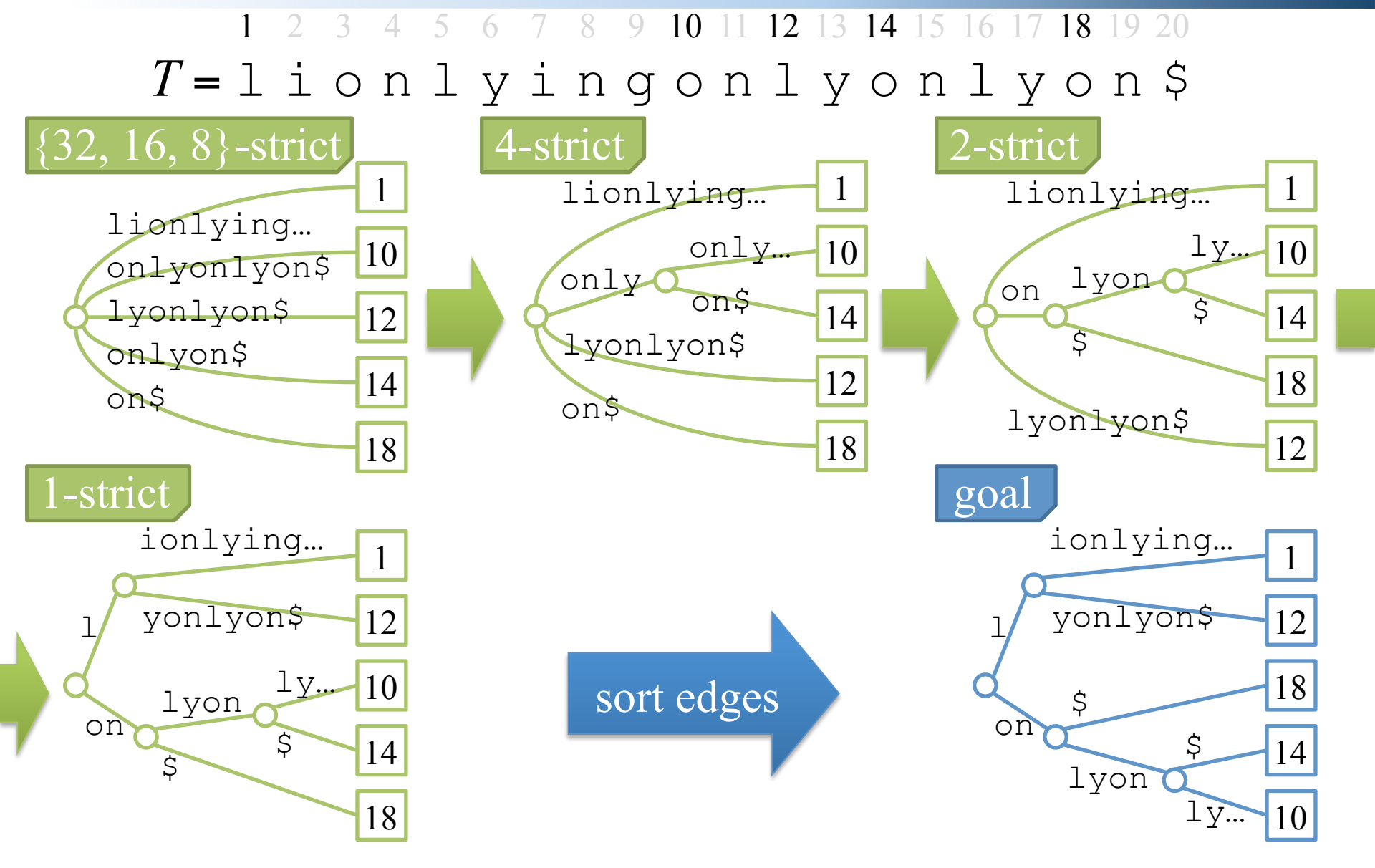
Sparse Suffix Tree:

- Compacted trie representing all designated suffixes.
- For each node, child edges are sorted.
- Since edge labels can be encoded by pointers to T, sparse suffix tree can be represented in $O(b)$ space.

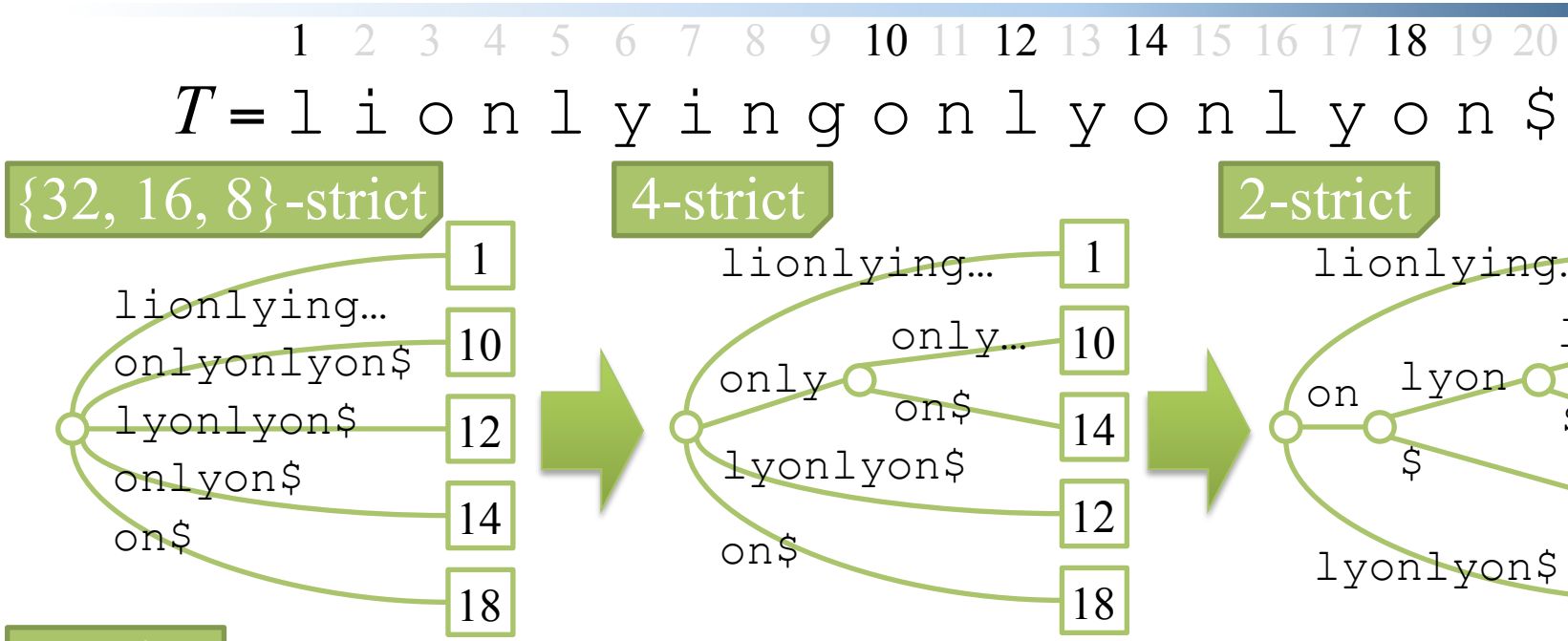
Overview: gradual refinement



l-strict (unsorted) sparse suffix trees



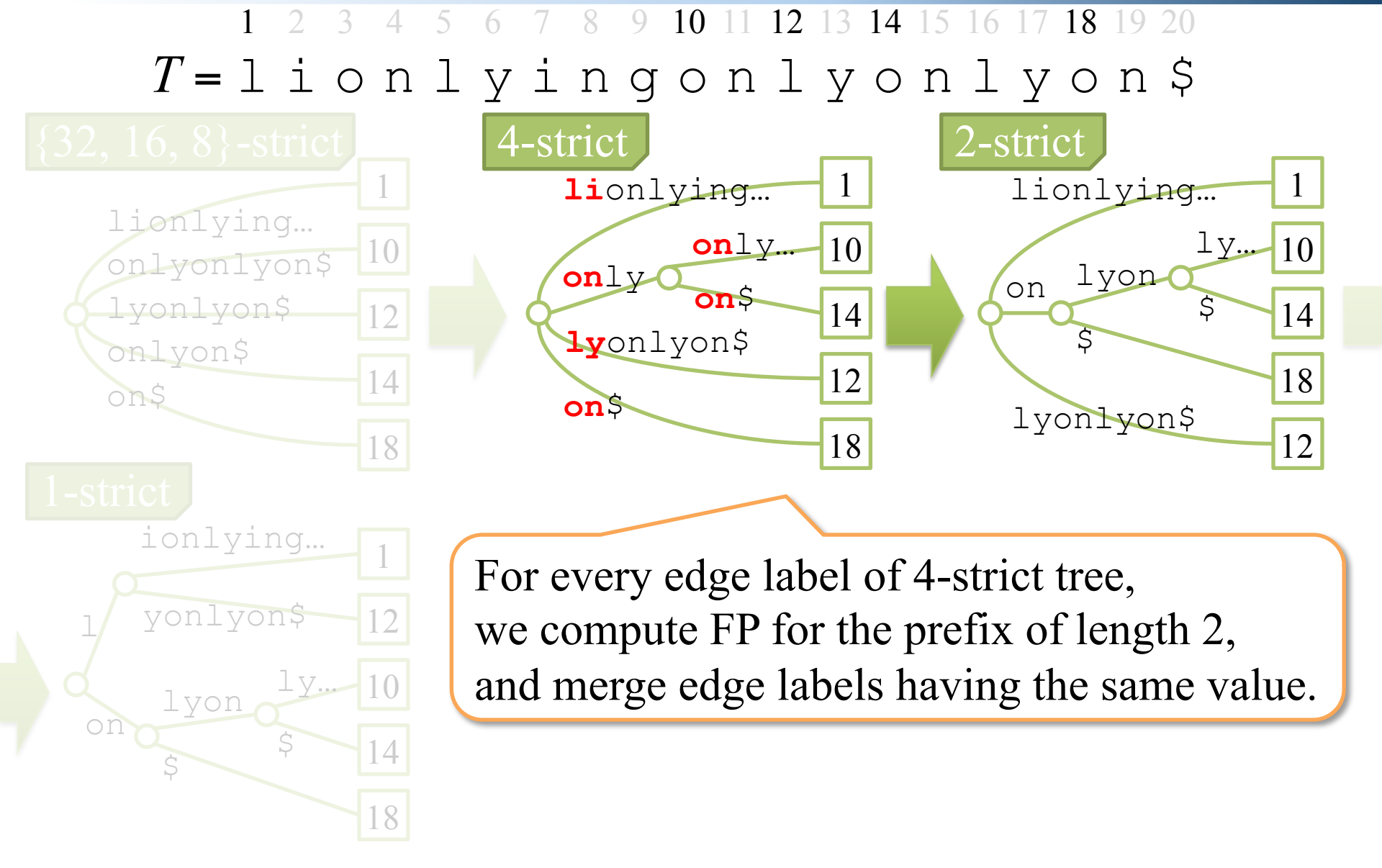
l-strict (unsorted) sparse suffix trees



Condition of l-strict tree

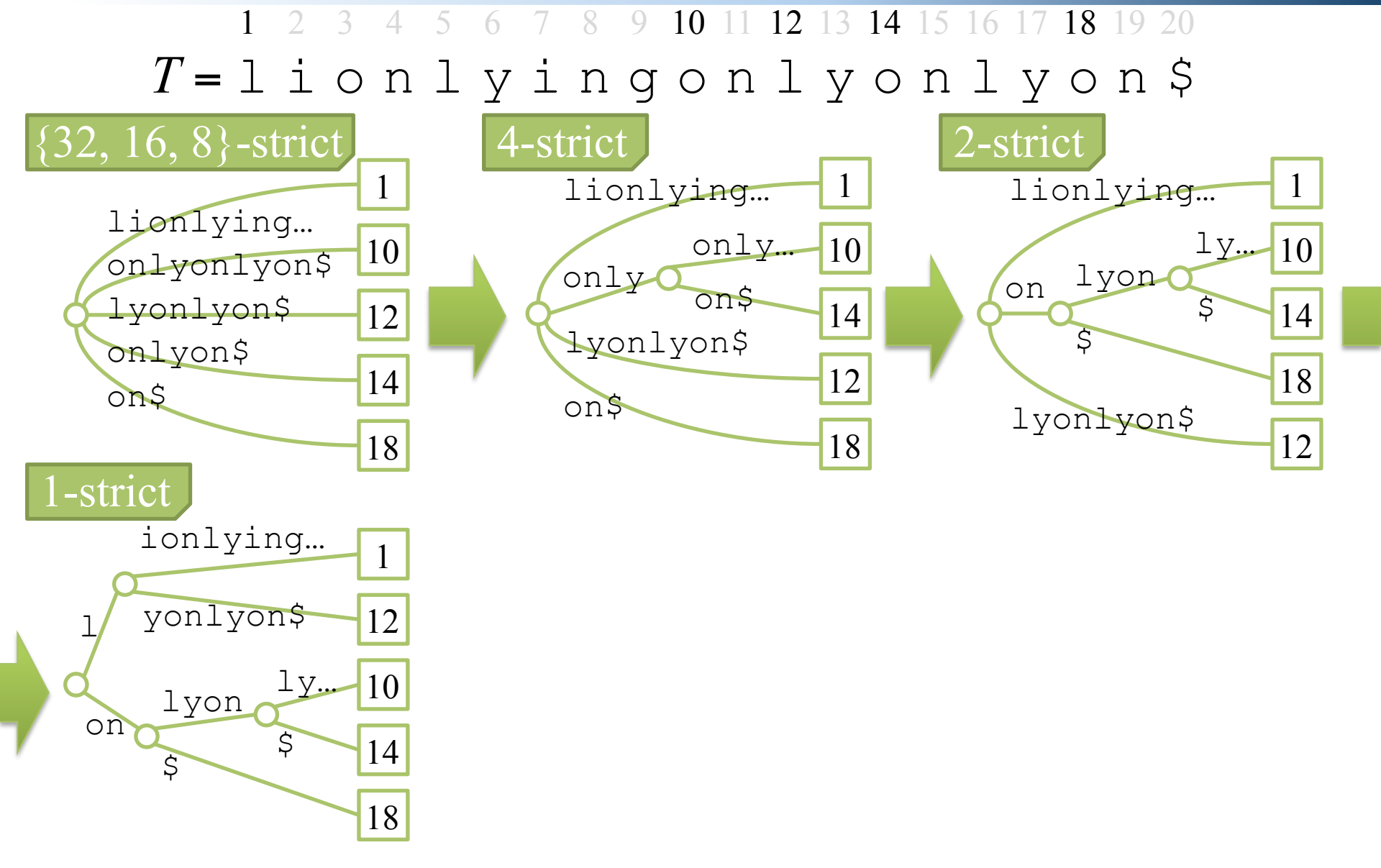
For any internal node of l-strict tree, any pair of child edge labels has common prefix of length less than l.

From 4-strict tree to 2-strict tree



For every edge label of 4-strict tree,
we compute FP for the prefix of length 2,
and merge edge labels having the same value.

Gradual refinement



The total cost for gradual refinement

The total cost for FP computation is $O((bn/s)\log s)$.

Proof.

- For each refinement step, the FPs can be computed in $O(b \min\{l, n/s\})$ time.
- For the first $\log s$ steps, the cost is $O((bn/s)\log s)$.
- After $\log s$ steps, since $l < n/2^{\log s} = n/s$, the cost is $bn/s + bn/2s + bn/4s + \ldots + b = O(bn/s)$

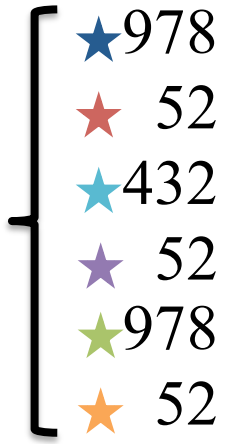
The total cost for gradual refinement

The total cost for merging edge labels is
$O(b \log n \log_s n) = O(b \log^2 n / \log s) = O((bn/s)\log s)$.

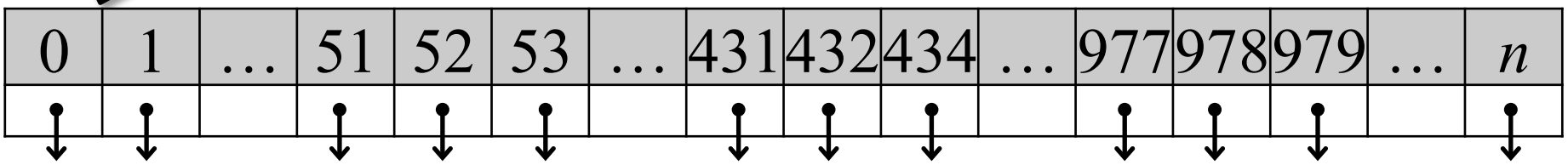
Proof.
- For each refinement step, we conduct radix grouping of $O(b)$ integer keys with radix s, which can be done in $O(b \log_s n)$ time.

Grouping by distribute-and-collect

k elements associated with keys (integers in [0, *n*])



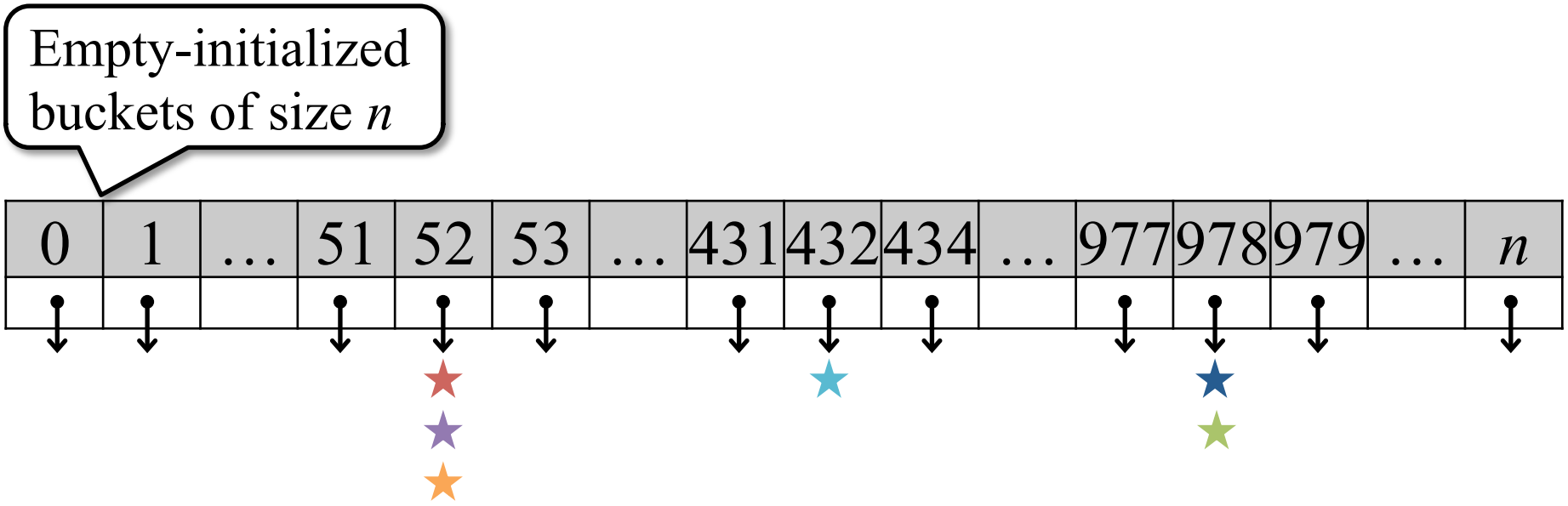
Empty-initialized buckets of size *n*



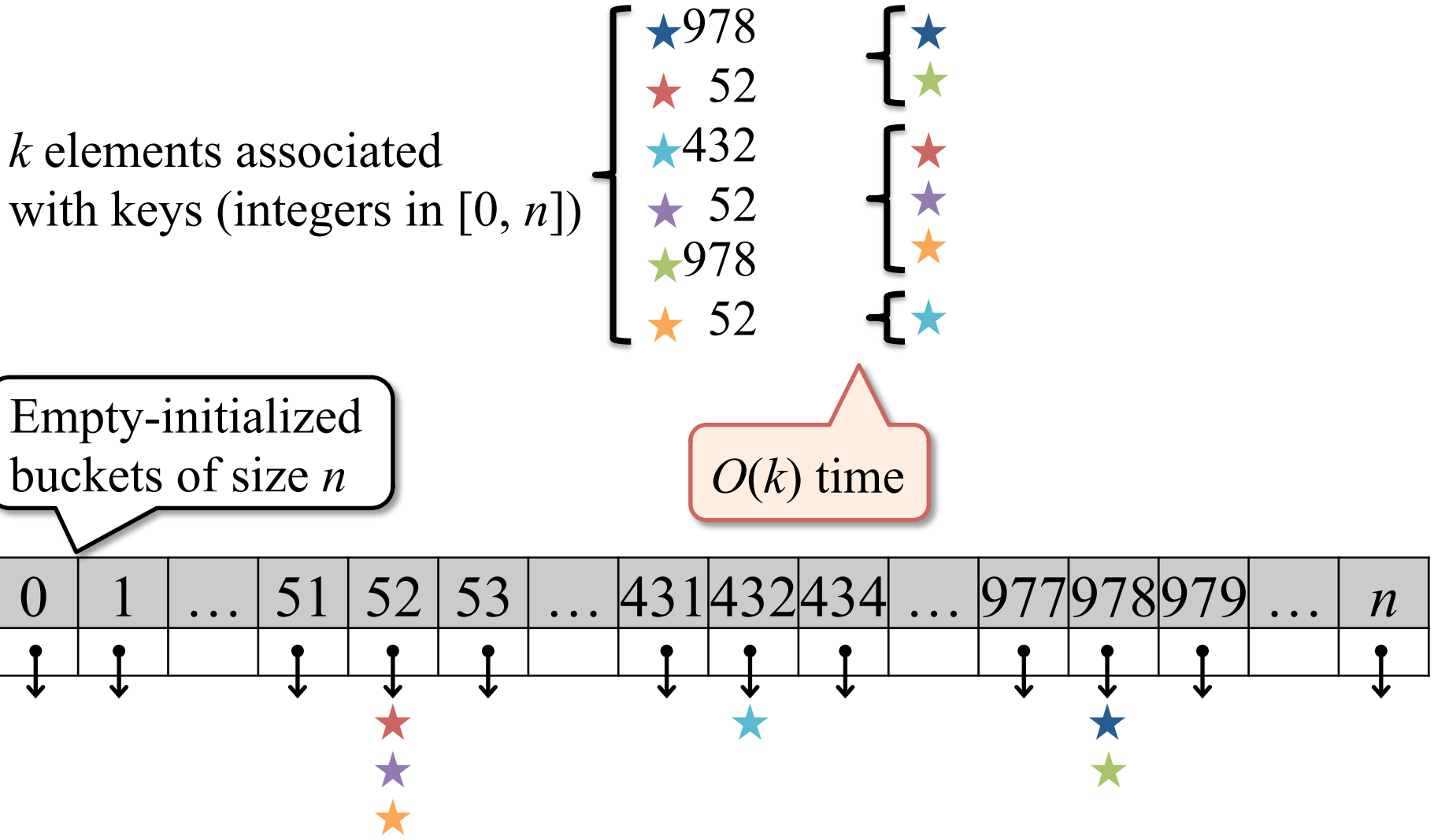
Grouping by distribute-and-collect

k elements associated with keys (integers in [0, *n*])

978
52
432
52
978
52

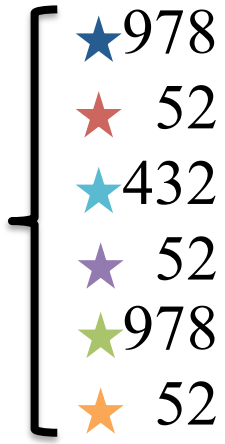


Grouping by distribute-and-collect

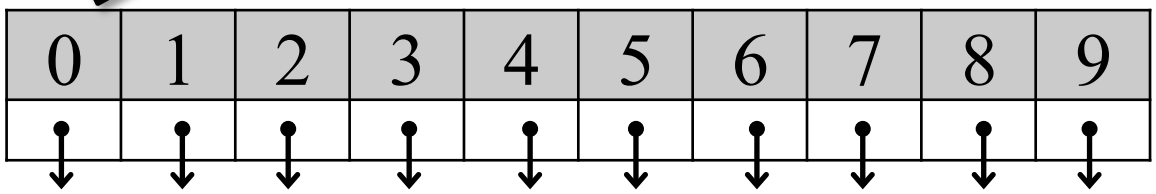


Grouping by distribute-and-collect

k elements associated with keys (integers in [0, *n*])



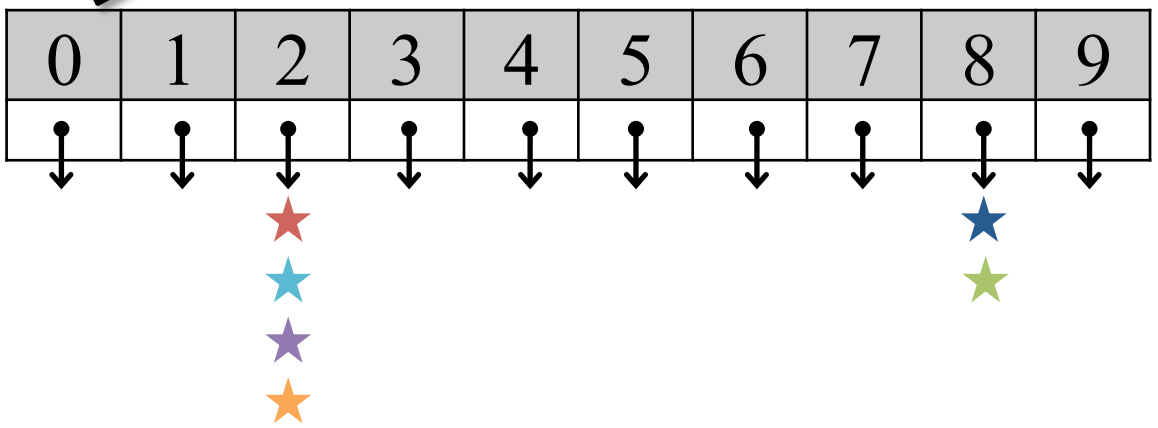
Empty-initialized buckets of size 10



Grouping by distribute-and-collect

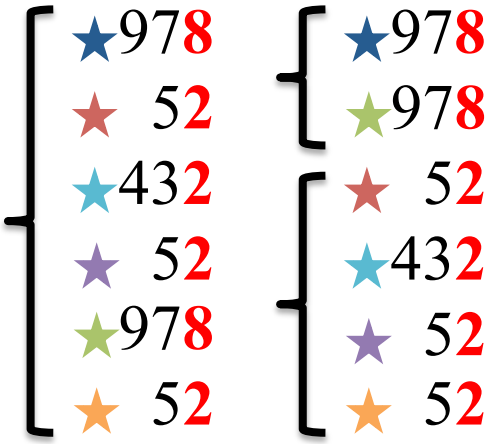
k elements associated with keys (integers in [0, *n*])

Empty-initialized buckets of size 10

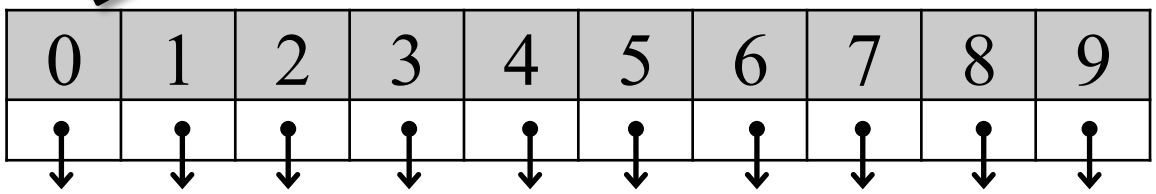


Grouping by distribute-and-collect

k elements associated with keys (integers in [0, *n*])

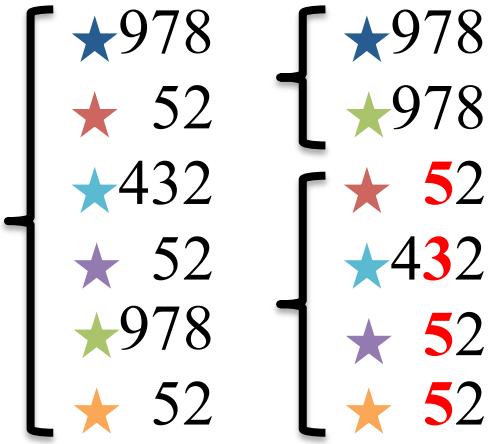


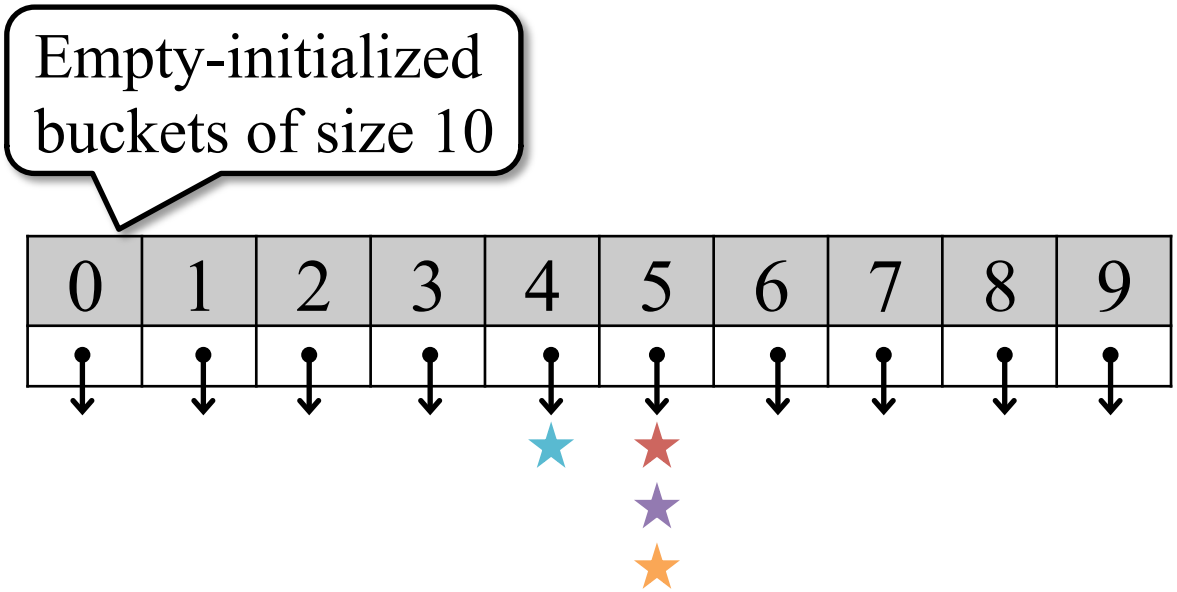
Empty-initialized buckets of size 10



Grouping by distribute-and-collect

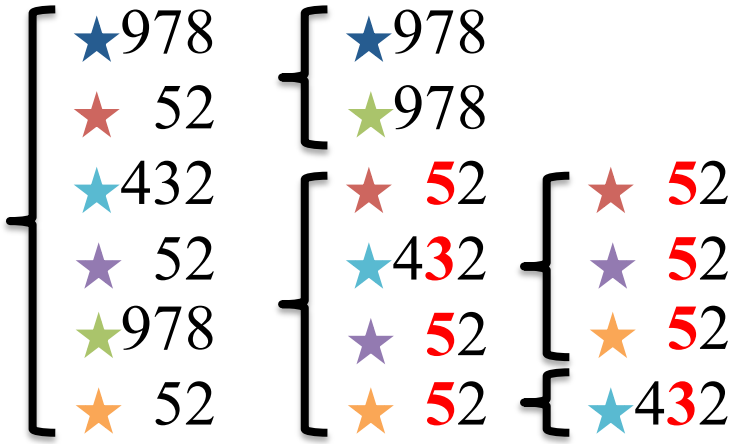
k elements associated with keys (integers in [0, *n*])



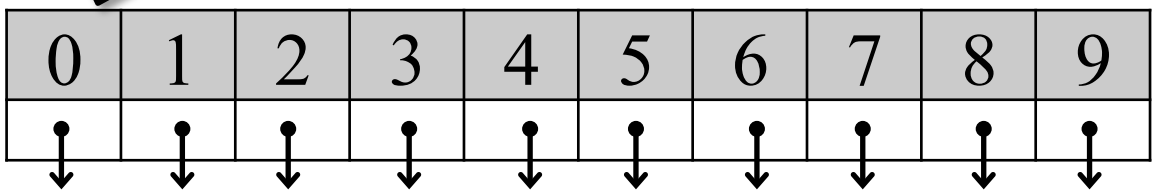


Grouping by distribute-and-collect

k elements associated with keys (integers in [0, *n*])

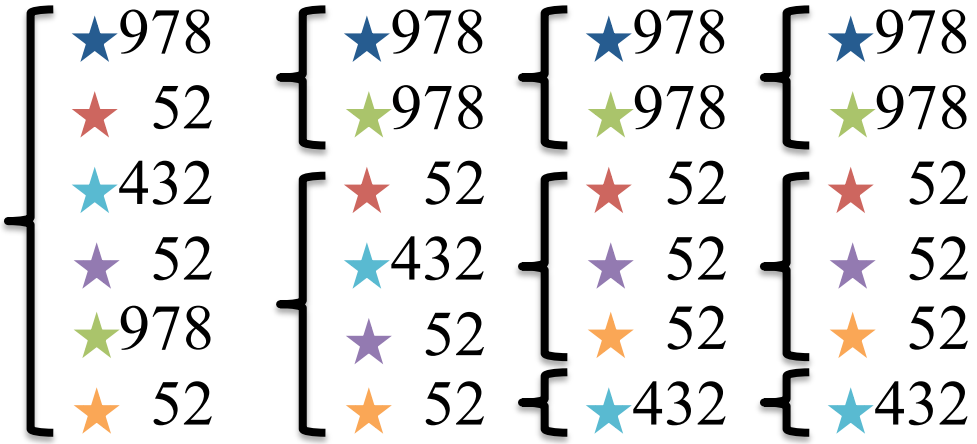


Empty-initialized buckets of size 10

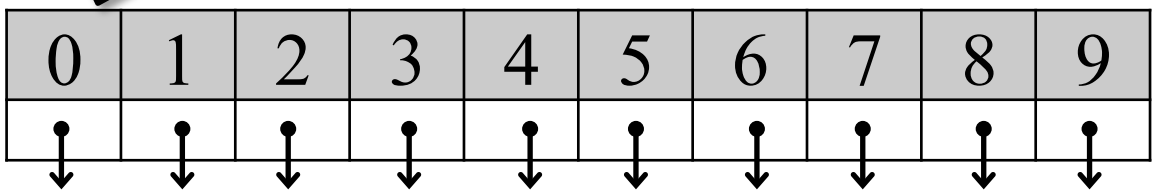


Grouping by distribute-and-collect

k elements associated with keys (integers in $[0, n]$)



Empty-initialized buckets of size 10

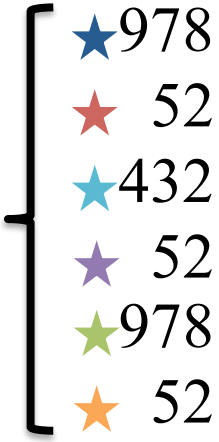


$O(k \log_{10} n)$ time

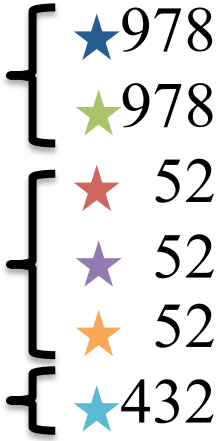
Grouping by distribute-and-collect

Radix grouping with radix *s*

k elements associated with keys (integers in [0, *n*])

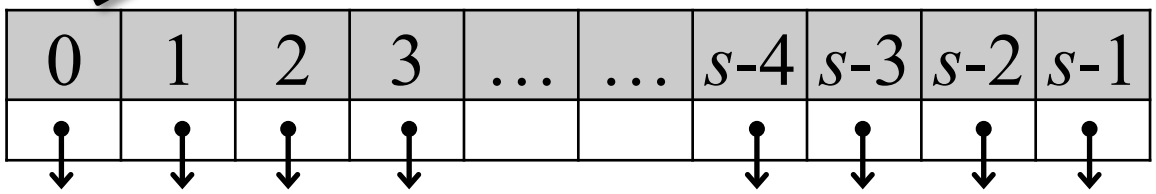


$\log_s n$ steps



$O(k \log_s n)$ time

Empty-initialized buckets of size ***s***



The total cost for gradual refinement

The total cost for merging edge labels is
$O(b \log n \log_s n) = O(b \log^2 n / \log s) = O((bn/s)\log s)$.

Proof.
- For each refinement step, we conduct radix grouping of $O(b)$ integer keys with radix s, which can be done in $O(b \log_s n)$ time.

Monte-Carlo algorithm

The total cost for FP computation is $O((bn/s)\log s)$.

The total cost for merging edge labels is
$O(b \log n \log_s n) = O(b \log^2 n / \log s) = O((bn/s)\log s)$.

Theorem

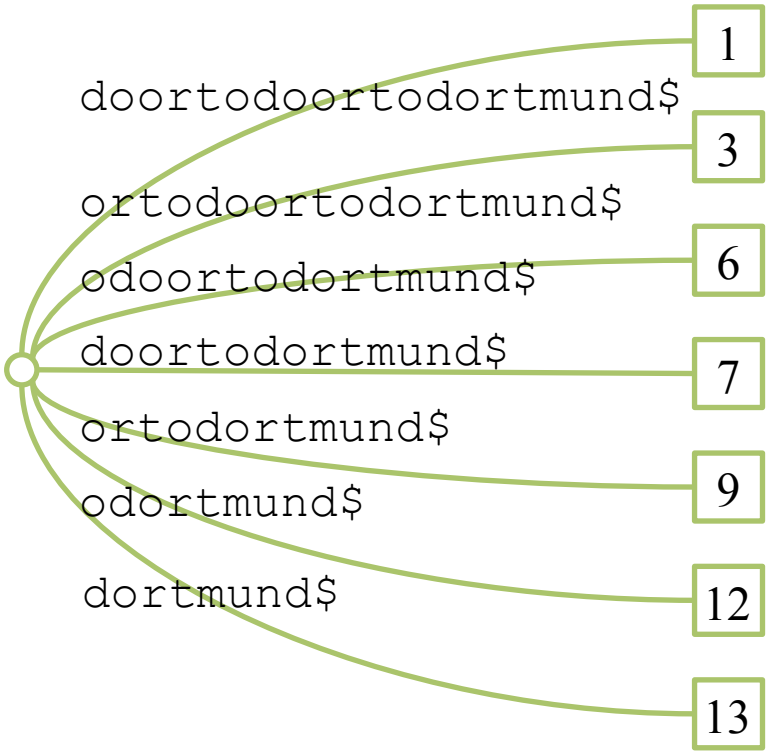
For any $s \in [b, n]$, we can construct sparse suffix tree correctly with high probability in $O(n+(bn/s)\log s))$ time using $O(s)$ additional space.

- When $s = b$, the time complexity is $O(n \log b)$.
- When $s = b \log b$, the time complexity is $O(n)$.

Exercise

- Follow the gradual refinement steps for this instance.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
$T =$	d	o	o	r	t	o	d	o	o	r	t	o	d	o	r	t	m	u	n	d	$



?