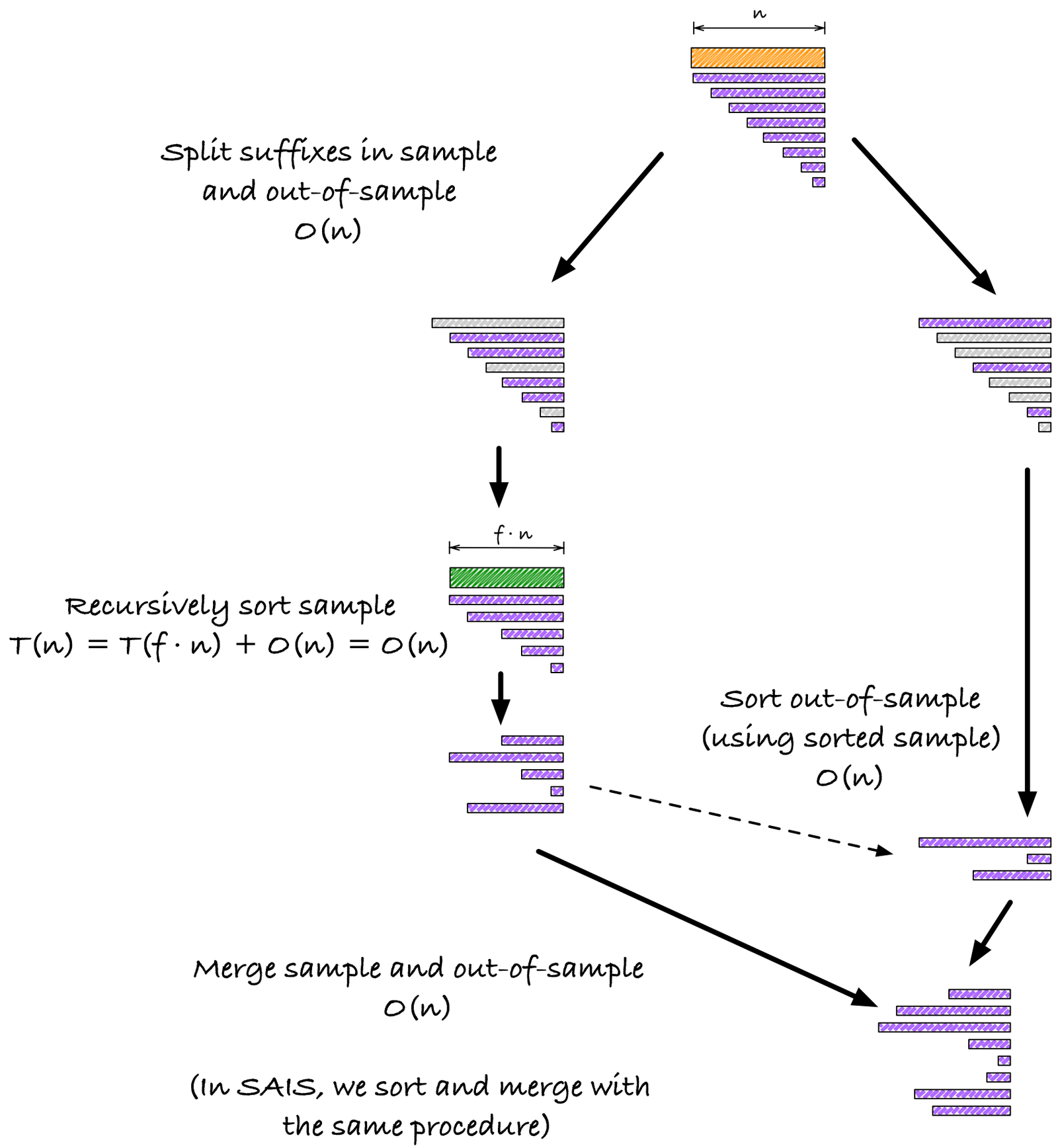
The SAIS algorithm

Suffix Array - Induced Sorting: Another linear time suffix array construction algorithm

Sampling approach

Same general approach as skew

Practically all the details differ



Why see both skew and SAIS?

- Skew is a classic and, I believe, the first sampling approach to SA construction
    - (Perhaps despite appearance) it is fairly easy to understand the approach in skew
    - Understanding the general approach makes it easier to understand similar algorithms
    - It’s the training wheels for the other algorithms (like tries and suffix trees)

Why see both skew and SAIS?

- SAIS involves more (conceptually) complex ideas
  - However, it is one of the fastest SA construction algorithms around (so should be your first choice when building SAs)
- It has a low memory overhead (O(n) bits*) in addition to input and output)
- It is easier to implement than skew is (just harder to fully understand)

*) Under some probabilistic assumptions such as random strings

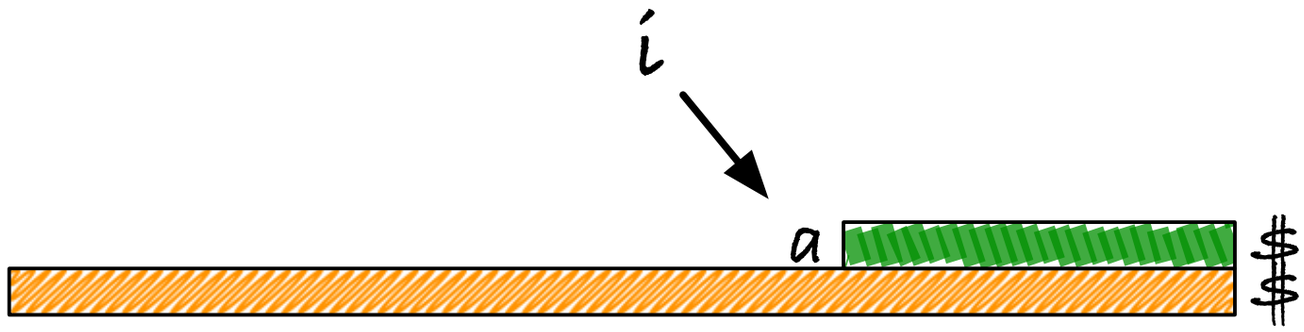
Samples

The samples are so-called "LMS suffixes".
They are not found at fixed offsets, but identified by the content of the string.

```
         *  *  *   *
x = mississippi$

         ississippi$
LMS         issippi$
suffixes       ippi$
                   $
```

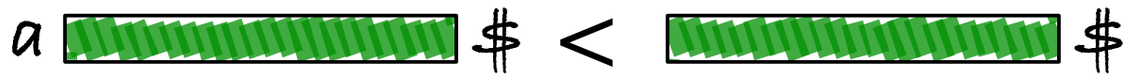
Suffix classes

We define “smaller” (S) and “larger” (L) classes by how suffix x[i:] compares to x[i+1:]

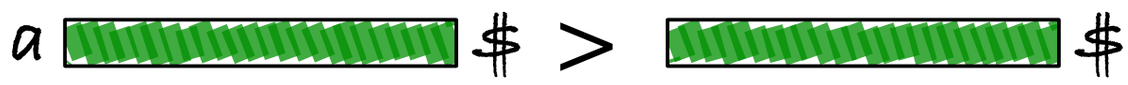


t[n] = S (x[n:] = “$” is always small)

t[i] = S if x[i:] < x[i+1:]



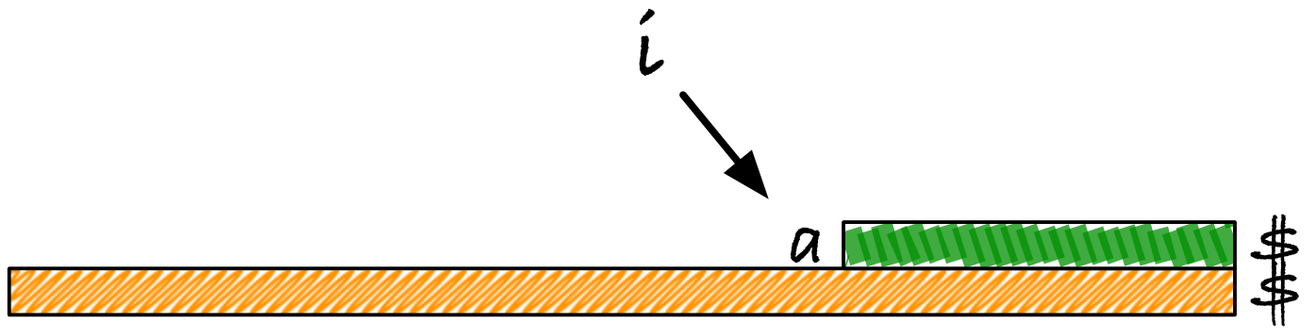
t[i] = L if x[i:] > x[i+1:]



We use a terminal sentinel in this algorithm (it simplifies things a lot)

Suffix classes

We define “smaller” (S) and “larger” (L) classes by how suffix x[i:] compares to x[i+1:]



t[n] = S (x[n:] = “$” is always small)

This is a special case (the empty string is really smaller than “$”) but we need the last index to be S.

Having this one special case for the sentinel prevents a lot of special cases in the algorithm (for example having to index past the last index). It is a major reason why we have $ in this algo.

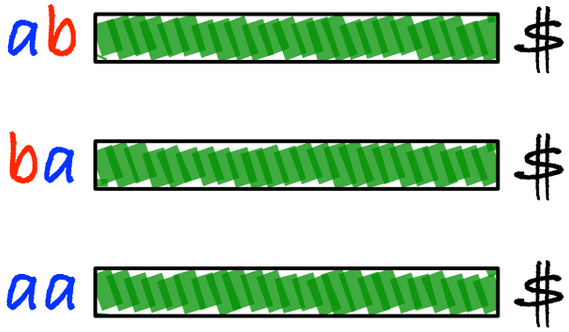
Computing classes

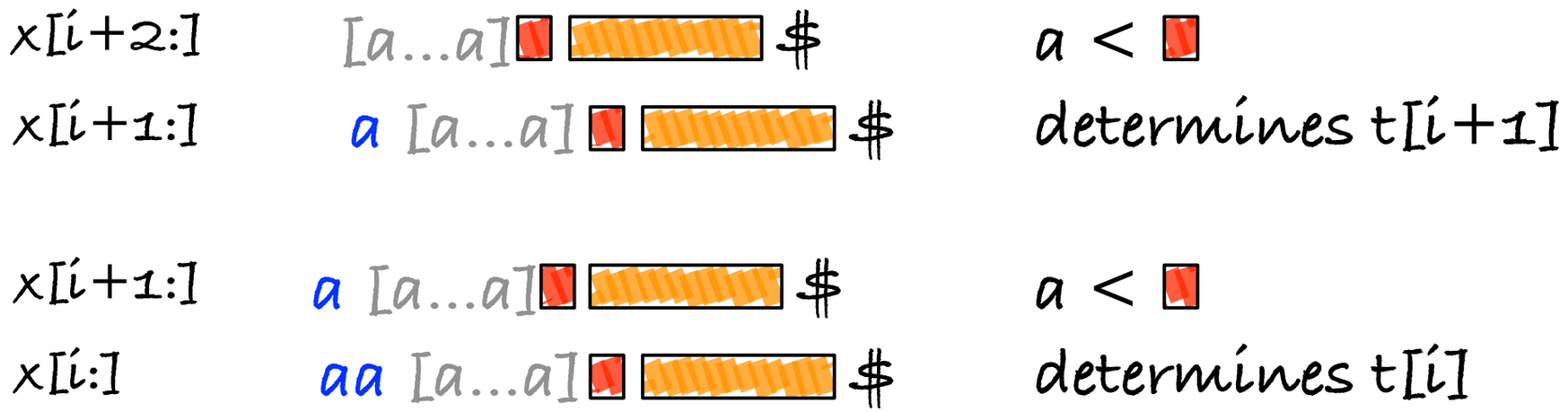
t[n] = S (x[n:] = "$" is always small)

t[i] = S if x[i] < x[i+1]

t[i] = L if x[i] > x[i+1]

t[i] = t[i+1] if x[i] = x[i+1]





You can build the type array in O(n) scanning right to left.

LMS suffixes

LMS indices are "**L**eft-**M**ost **S**mall" indices.
LMS suffixes are suffixes that start at an LMS index.

```
          *   *   *     *
          LSLLSLLSLLLS
x = mississippi$

lms[i] = t[i] == S and t[i-1] == L
```

LMS suffixes

```
ississippi$
 issippi$
  ippi$
   $
```

Magical recursion: building u

```
          *  *  *   *
x = mississippi$

u = ississippi$
```

(encoded in new alphabet)

u = [issi][issi][ippi$][$]

Strings between LMS indices we will call LMS strings (different from LMS suffixes). They will be our alphabet.

"$" is also an LMS string ("between" will include from the last index to the end in this definition)

Magical recursion: building u

- With a dynamic alphabet, we can’t assume that σ is constant
- We avoid problems the same was as with skew:
  - we only use the alphabet size when sorting in a “bucket-like” way, in time O(n + σ), and
  - σ is in O(n) because you can’t have more than n LMS strings

Magical recursion: sorting sample

```
LMS indices:    1   4   7   11
          x = mississippi$

    0      1      2       3
u = [issi][issi][ippi$][$]
```

p = [1, 4, 7, 11]
maps from indices in u to indices in x
(but you don't need it explicitly)

1-1 correspondence between
suffixes in u and LMS suffixes
in x (and they come in the same order)

```
u[sa_u[0]:] = u[3:] = [$]
u[sa_u[1]:] = u[2:] = [ippi$][$]
u[sa_u[2]:] = u[1:] = [issi][ippi$][$]
u[sa_u[3]:] = u[0:] = [issi][issi][ippi$][$]

x[p[sa_u[0]]:] = x[p[3]:] = x[11:] = $
x[p[sa_u[1]]:] = x[p[2]:] = x[7:]  = ippi$
x[p[sa_u[2]]:] = x[p[1]:] = x[4:]  = issippi$
x[p[sa_u[3]]:] = x[p[0]:] = x[1:]  = ississippi$
```

Sorting suffixes of u sorts
the LMS suffixes in x

(provided the alphabet
preserves the LMS string
order)

Why the redundancy?

```
                  *   *   *    *
          x = mississippi$

u = [issi][issi][ippi$][$]
```

```
u = [iss][iss][ippi][$]?
```

Not including the redundancy would also work for sorting.

[aac][abb] < [aac][bbb] iff [aaca][abb] < [aacb][bbb]

Pulling a character forward doesn't change the order.

The alphabet would be smaller, and we would need to look
at more character to determine the order of two strings, that is all.

Why the redundancy?

```
                    *   *   *    *
                x = mississippi$

u = [issi][issi][ippi$][$]
```

```
u = [iss][iss][ippi][$]?
```

We do it this way because the process for creating the alphabet sorts the overlapping LMS strings, and we might as well exploit that.

You could change that and build u from the other alphabet if you wanted to, but there is no good reason for it.

What is the length of u?

- There is a letter in u for each interval between LMS indices (+1 for the sentinel)
- You cannot have consecutive LMS indices [why?]
- Thus |u| ≦ ½n
- T(n) = T(½n) + O(n) in O(n)

$$\sum_{i=0}^{\infty} \frac{1}{2^i} = 2$$

Terminating the recursion

- What is the base case for the recursion? For what u do we return immediately instead of recursing?

- If all letters are unique, we can bucket sort (not unlike the stop criteria in the skew algorithm)

Terminating the recursion

- What is the base case for the recursion? For what u do we return immediately instead of recursing?

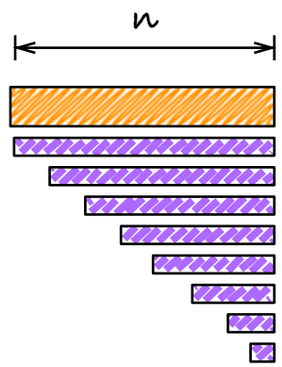
- If all letters are unique, we can bucket sort (not unlike the stop criteria in the skew algorithm)

- If the letters are numbered 0, 1, … σ - 1, this is particularly simple:

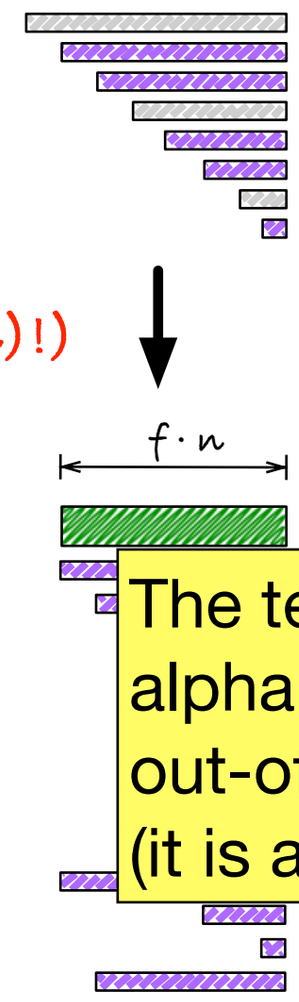
```
for i, a in enumerate(x):
    sa[a] = i
```

- You can easily ensure this when building alphabets, but it must also be true for all input, so be careful

Status on magical recursion…



Classify suffixes as S or L
Your sample is the LMS suffixes



Build u from the first LMS suffix
(we need to build the alphabet and u in O(n)!)

Extracting order of LMS suffixes from sa_u is straightforward (they correspond 1-1 in the order they appear)

The technique for building the alphabet will also solve sorting the out-of-sample and merging…
(it is a pretty cool idea)

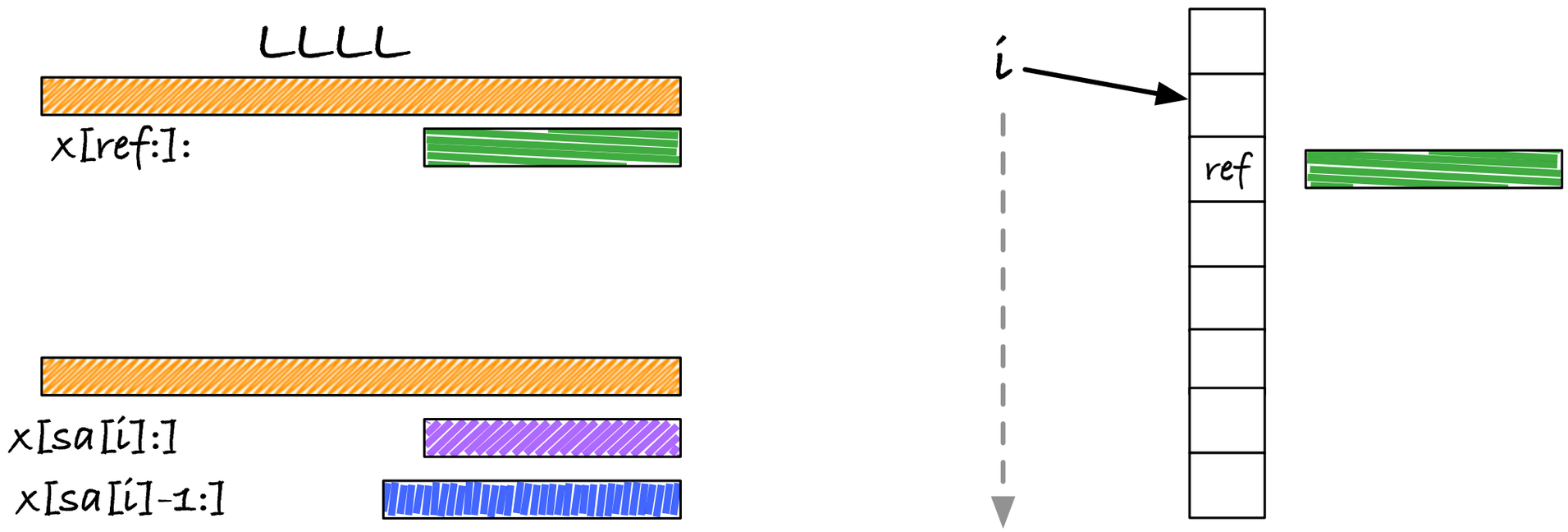
Induced sorting

- Creating the alphabet for u requires that we sort the LMS strings (the strings between LMS indices)

- The technique we use is similar to bucket/radix sort

- When the LMS suffixes are sorted, we can sort the remaining strings with the same approach

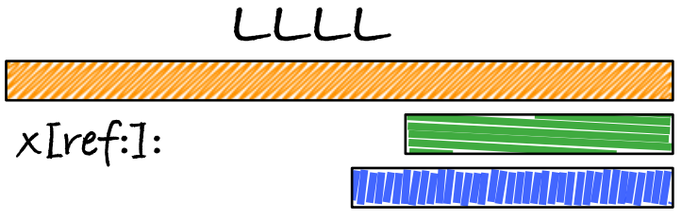
Inserting runs of L

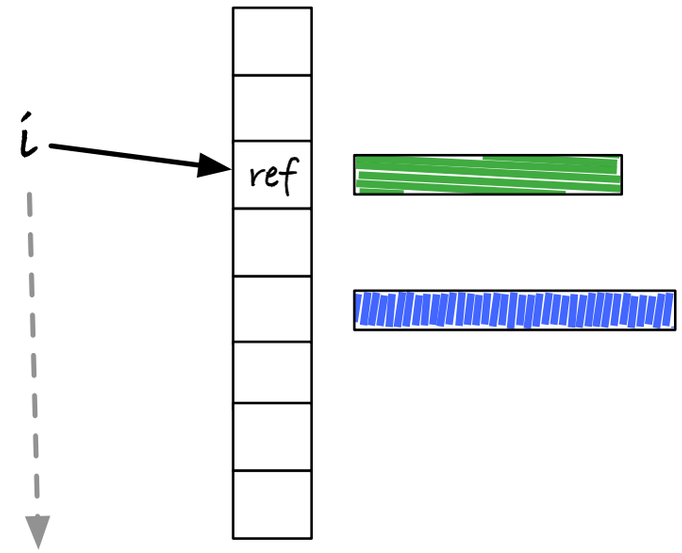



```
for i = 0, ..., n+1:
   if sa[i] defined and t[sa[i]-1] = L:
      bin sa[i]-1
```

When binning, fill from the top
of the bin. That way, if you insert two strings
in the same bin, the first sorts before the second

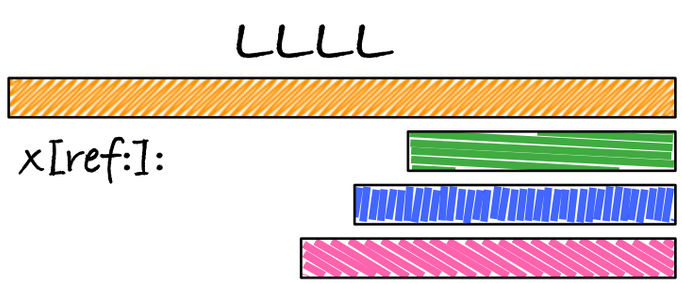
Inserting runs of L



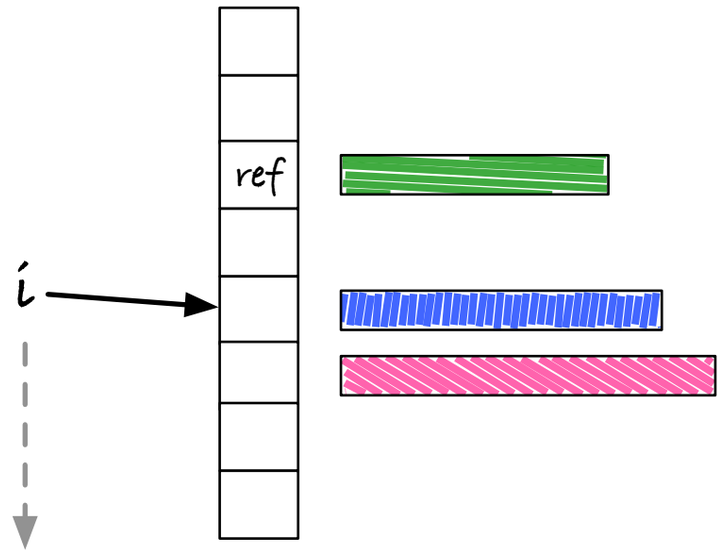


When we see ref we insert
ref - 1. Since x[ref-1:] is L
it goes later in the array
(maybe same bin, maybe later bin)

Inserting runs of L

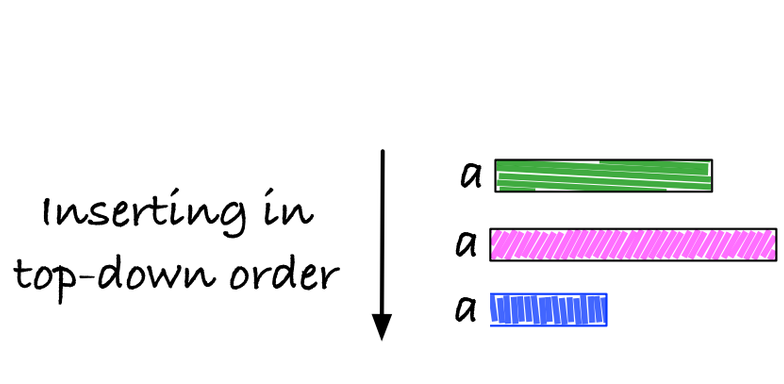


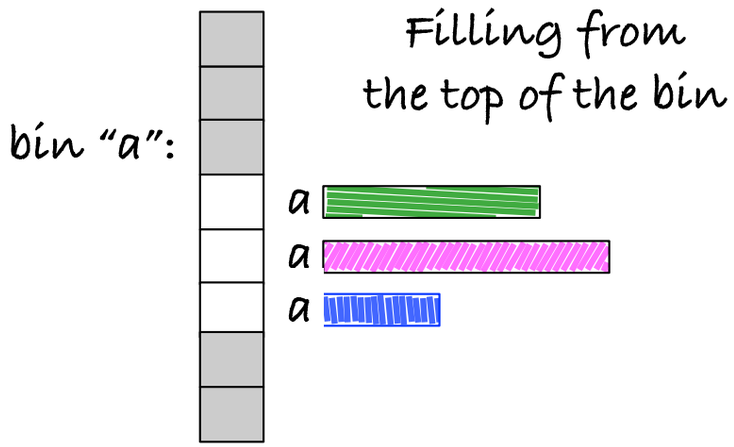
When we see ref - 1 we insert ref - 2. Since x[ref-2:] is L it goes later in the array (maybe same bin, maybe later bin)

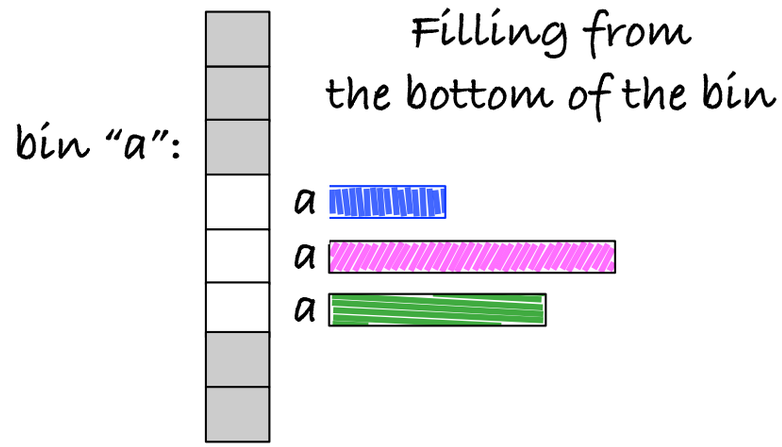


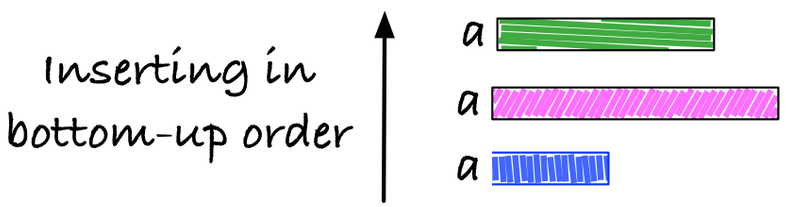
When we are done, we have inserted all the L-suffixes before ref in sorted order.

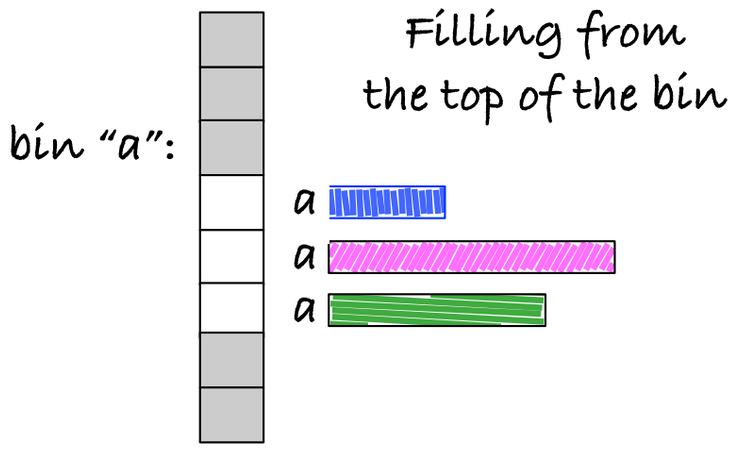
Order of binning

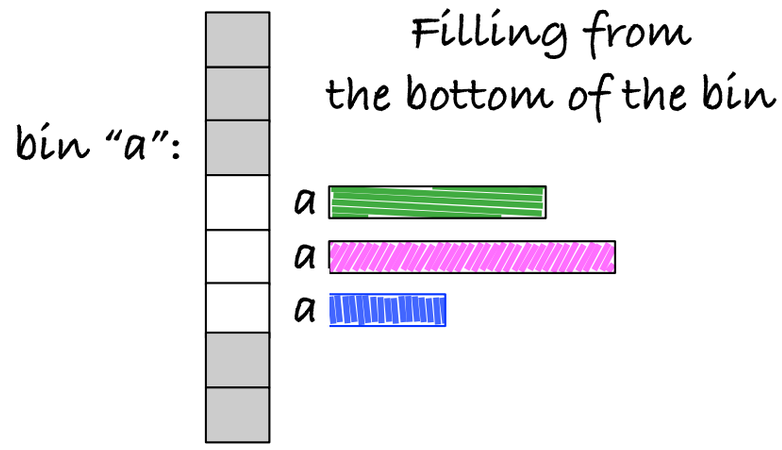




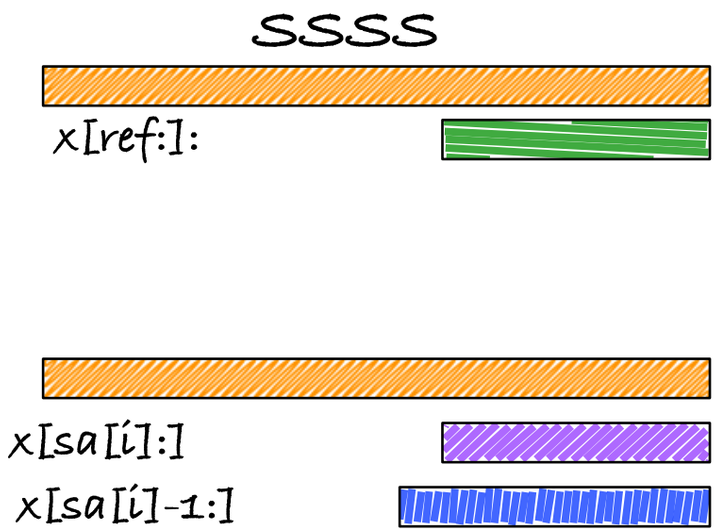


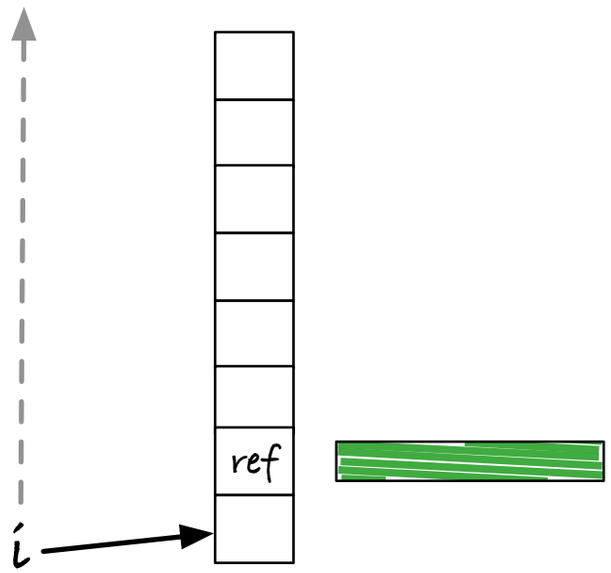






Inserting runs of S

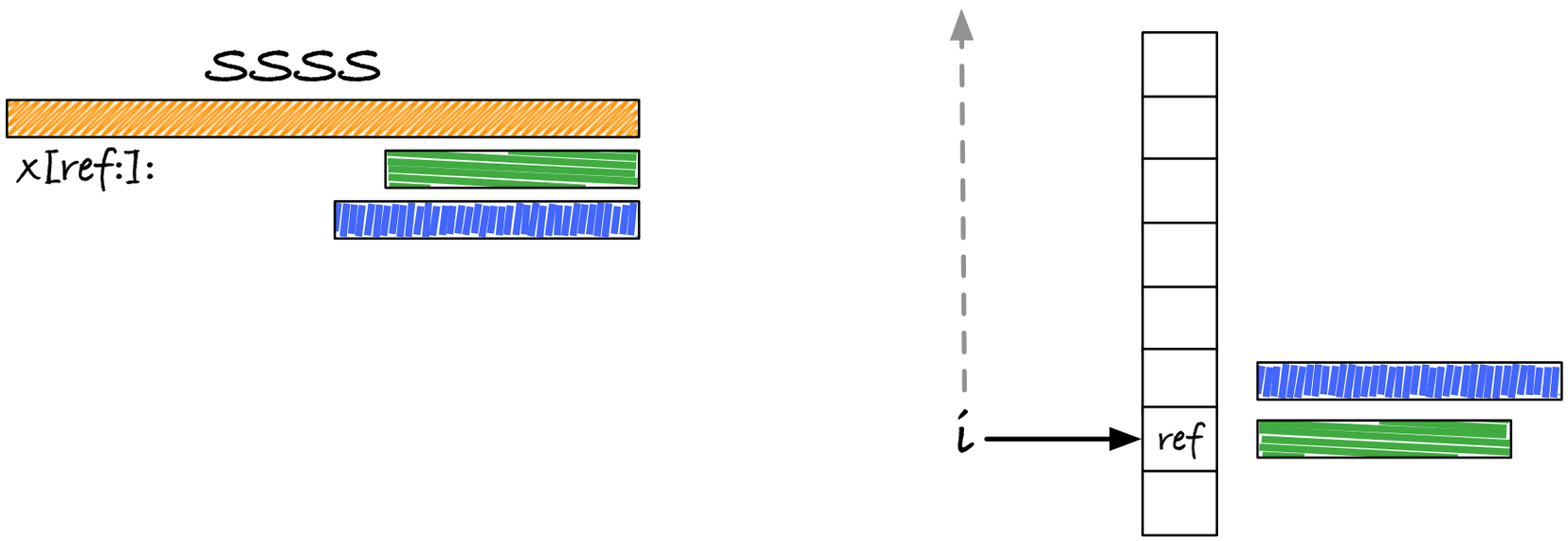





```
for i = n+1, ..., 0:
    if sa[i] defined and t[sa[i]-1] = s:
        bin sa[i]-1
```

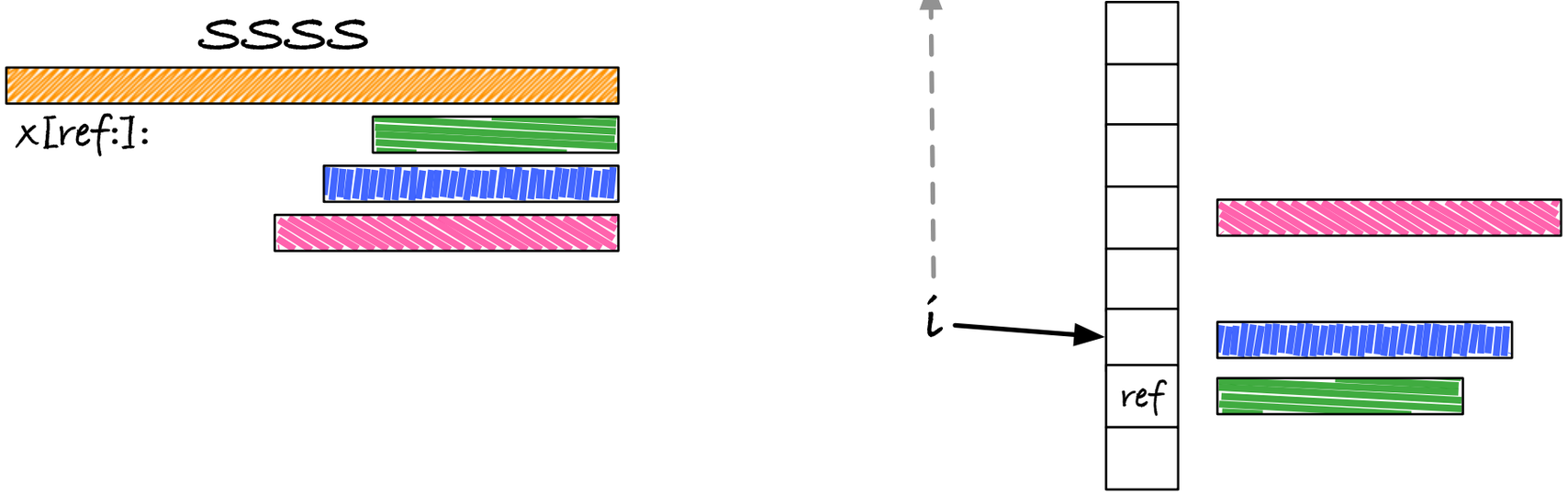
When binning, fill from the bottom
of the bin. That way, if you insert two strings
in the same bin, the first sorts after the second

Inserting runs of S



When we see ref - 1 we insert
ref - 2. Since x[ref-2:] is S
it goes earlier in the array
(maybe same bin, maybe later bin)

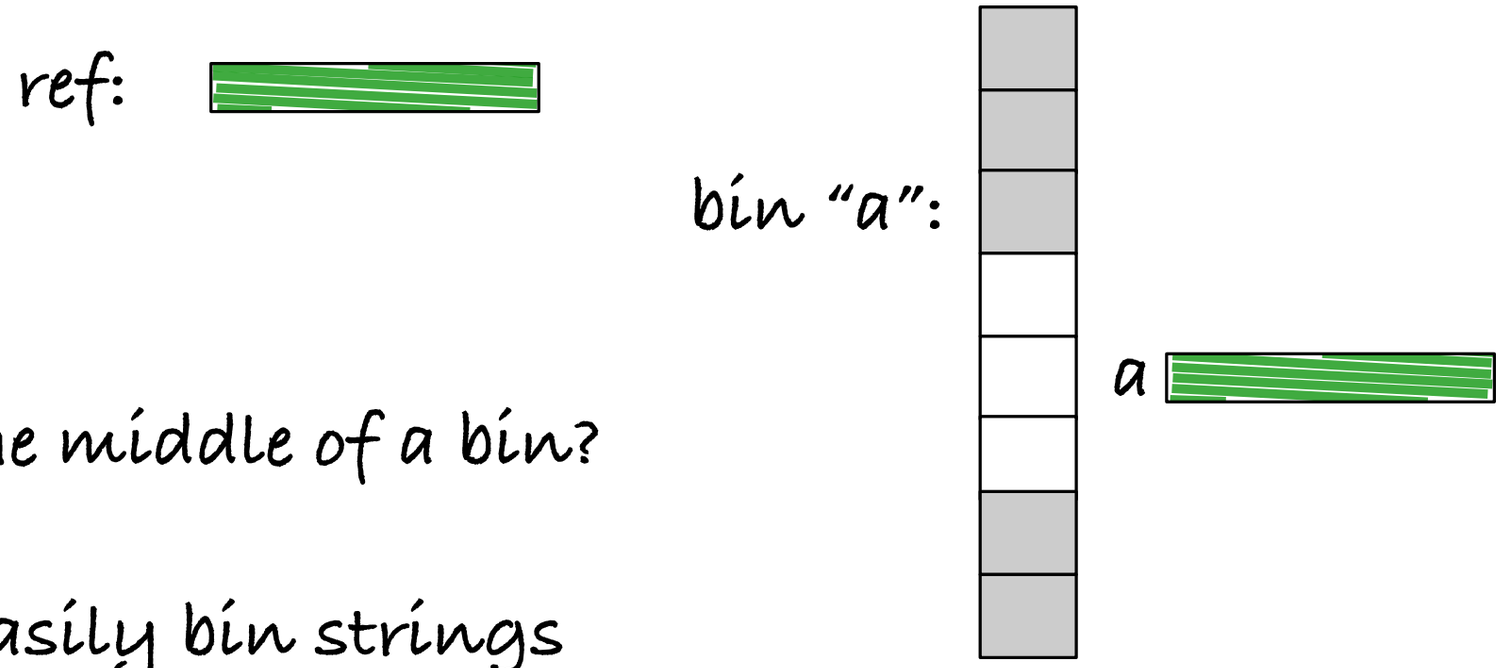
Inserting runs of S



When we see ref we insert
ref - 1. Since x[ref-1:] is s
it goes earlier in the array
(maybe same bin, maybe later bin)

When we are done, we have inserted all the S-suffixes before ref in sorted order.

What about the placement of the reference string?



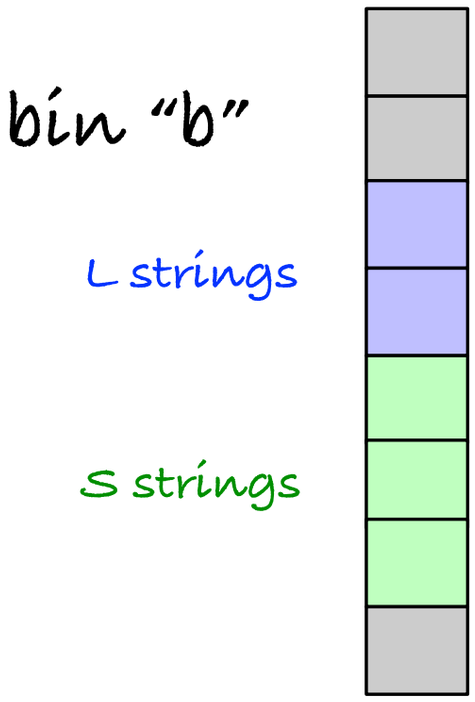
What if ref is in the middle of a bin?

Then we cannot easily bin strings
[so we should do something to avoid
that ref can sit in a position we need]

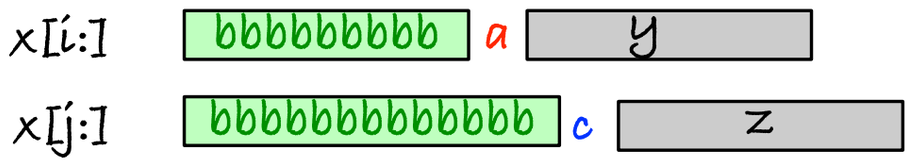
Bucket structure

L string in bin "b": $x[i:] = b^k ay$, $t[i] = L$, so $a < b$

S string in bin "b": $x[j:] = b^l cz$, $t[j] = S$, so $b < c$

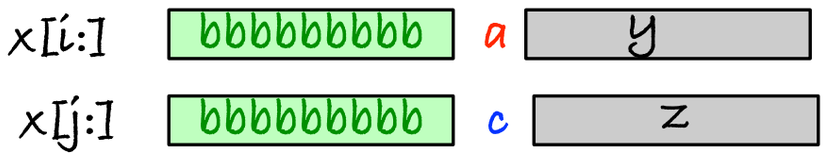


A) $l > k$



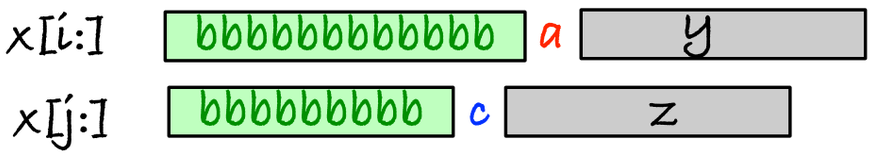
$a < b \rightarrow x[i:] < x[j:]$

B) $l = k$



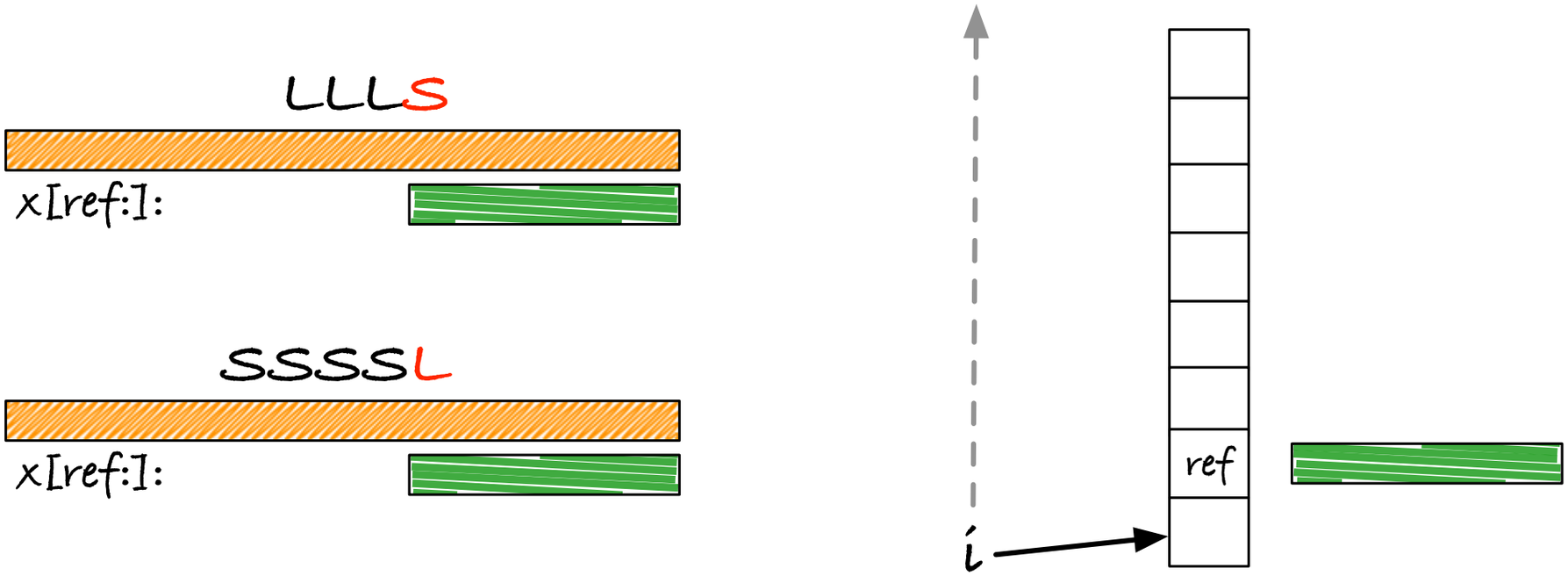
$a < b < c \rightarrow x[i:] < x[j:]$

C) $l < k$



$b < c \rightarrow x[i:] < x[j:]$

Computing runs…



If x[ref:] is an S string we can put it at the bottom of its bin and sort the L-run before it safely (top-down/left-right)

Likewise, if x[ref:] is an L string we can put it at the top of its bin and sort the S-run before it (bottom-up/right-left)

Can we sort using these runs?

- Scanning through the array takes time O(n), so we can only do it a constant number of times

- There are as many runs of L or S as there are LMS indices

- That is O(n), so sorting run by run would take $O(n^2)$ [and we don't even know how to merge the results…]

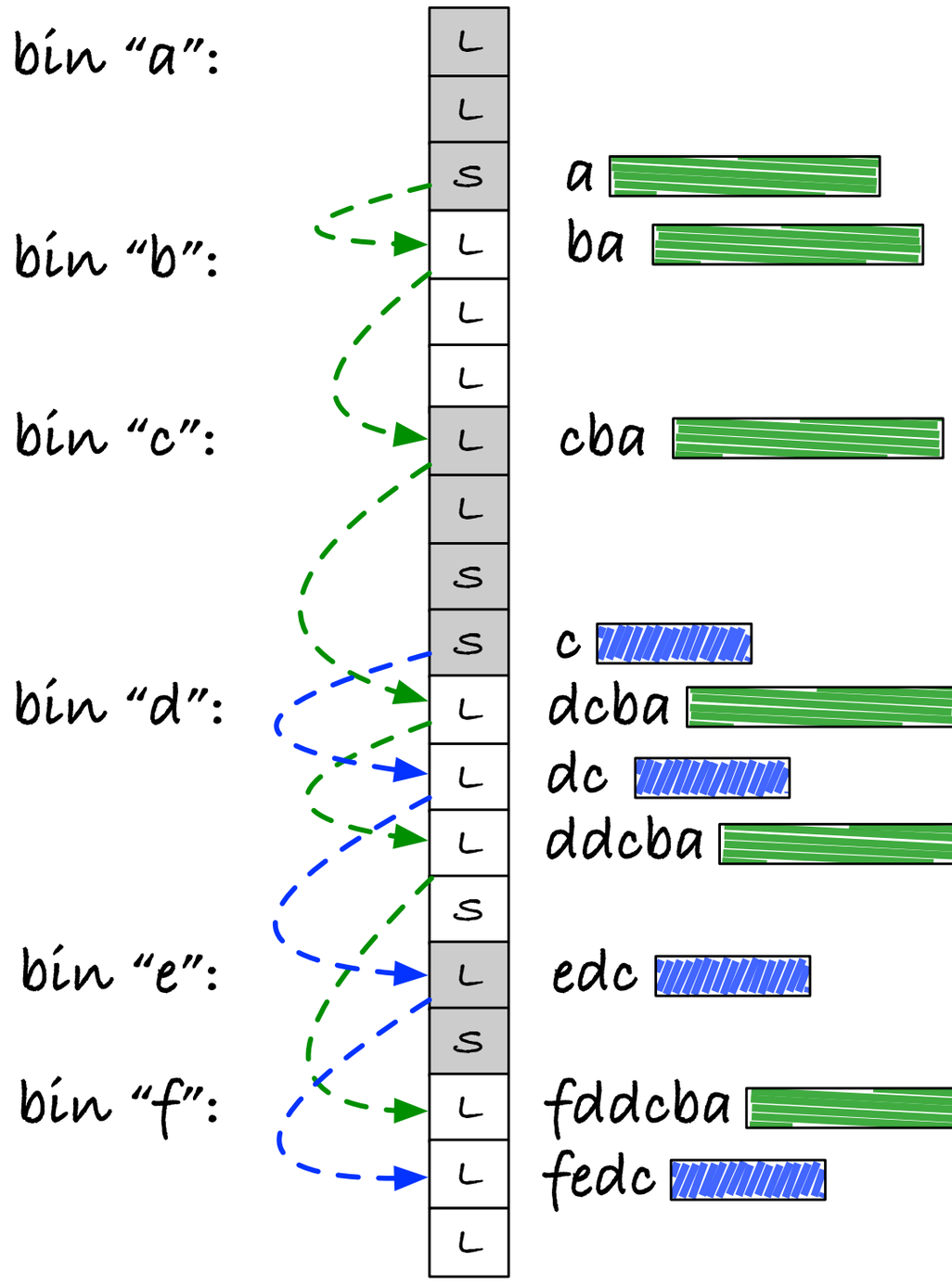
Sorting in parallel

You can, however, use more than one reference.

If you start with two references, like here
you insert the run before both of them in one scan
(top-to-bottom for L strings in this example).

You can start with any number of refs and sort the preceding L or S runs [depending on the refs].

Such a parallel run takes time O(n), but does it sort all the strings in the runs?



The strings are sorted

(sort of)

- Consider two strings in the array after the procedure.
- If they start with different letters, they are sorted (the bins are trivially sorted)
- Otherwise, call them ay and az, and assume that they are out of order.
- Both could be references, and then they are out of order because the references are
- One could be a reference and the other not, but then they have different classes, and the structure of bins ensure they are ordered
- Otherwise, y and z are in the array and out of order. Redo the analysis on them.
- **Conclude:** Either they are sorted, or the references are out of order.

The strings are sorted
(sort of)

- Otherwise, call them ay and az, and assume that they are out of order.
    - Both could be references, and then they are out of order because the references are
    - One could be a reference and the other not, but then they have different classes, and the structure of bins ensure they are ordered
    - Otherwise, y and z are in the array and out of order. Redo the analysis on them.
    - **Conclude:** Either they are sorted, or the references are out of order.

The strings are sorted

(sort of)

- Otherwise, call them ay and az, and assume that they are out of order.
    - Both could be references, and then they are out of order because the references are
    - One could be a reference and the other not, but then they have different classes, and the structure of bins ensure they are ordered
    - Otherwise, y and z are in the array and out of order. Redo the analysis on them.
- **Concl** order.

You can make this proof more formal with contradiction + induction, but it boils down to the same. If ay and az are out of order, y and z must be as well.

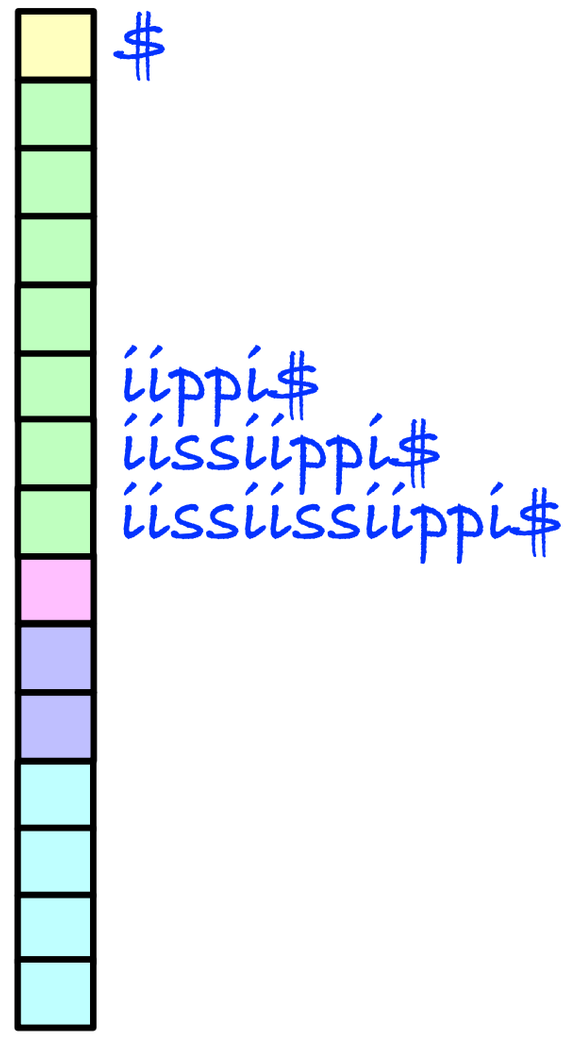
This is great!

- This tells us how to sort all the non-LMS suffixes once we have the order of the LMS suffixes from the recursion (the sort and merge bit)

Induced sorting

```
 *   *   *    *
LSSLLSSLLSSLLLS
miissiissiippi$
```

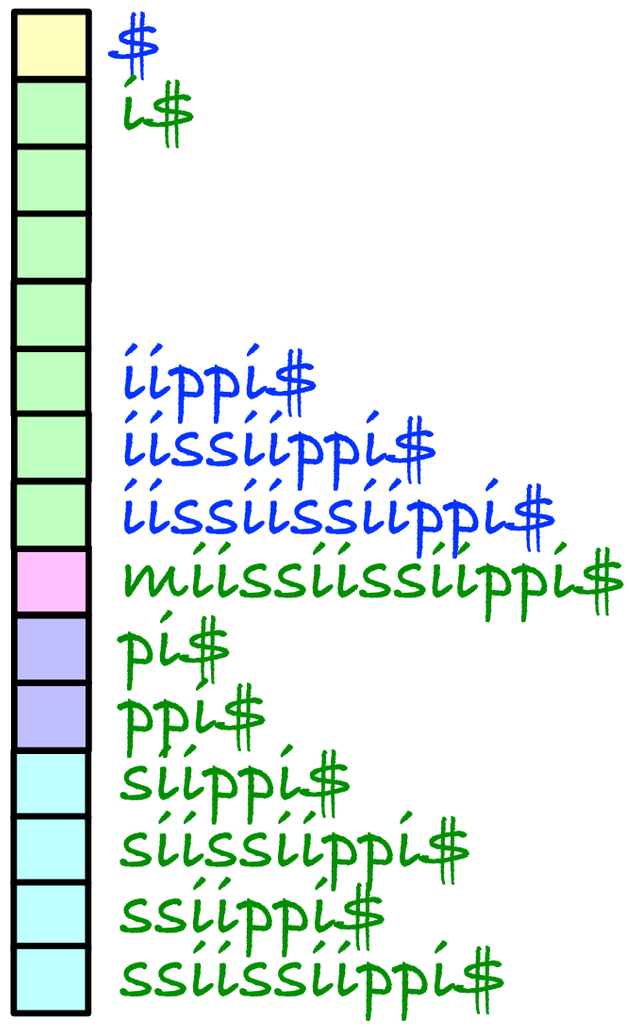
After sorting LMS-suffix,
put them in the S-part of their bins.
Don't worry about the right positions
(we don't know them anyway),
just get them in the right order.



Induced sorting

```
 *   *   *    *
LSSLLSSLLSSLLLS
miissiissiippi$
```

Do an L scan to sort the runs of L before the LMS suffixes. After that, all L suffixes are sorted and in their right position.



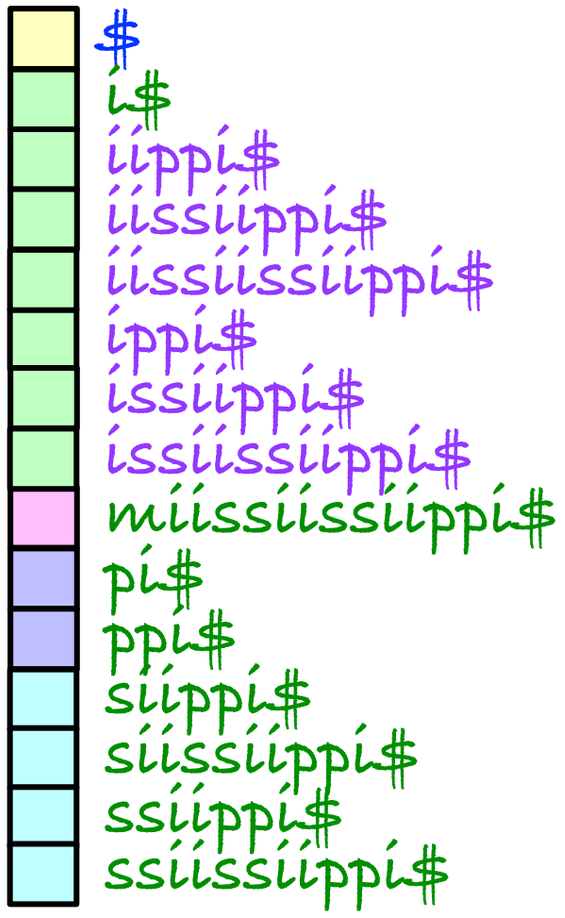
Induced sorting

✪ ✪ ✪

LSSLLSSLLSSLLLS

miissiissiippi$

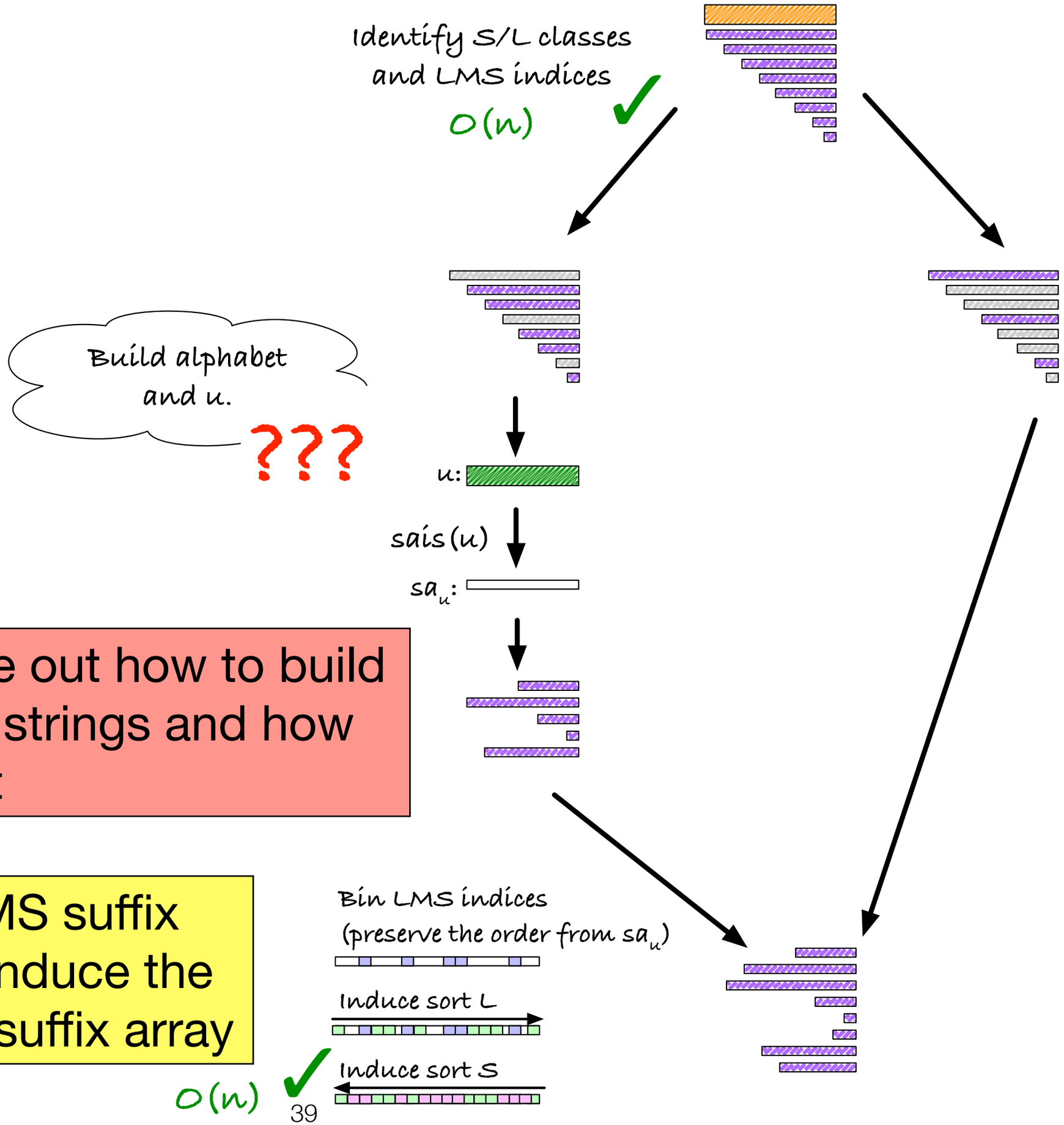
Do an S scan to sort the runs of S before the leftmost Ls.



Just keep all the strings in the array. The procedure skips past those we don't need, leaving them in the right positions. But reset the bucket pointers so we can fill the S segments of the bins.

The LMS strings might move here, as they are sorted to their correct positions.

Current status



We still need to figure out how to build the alphabet of LMS strings and how to construct u from it

If we can sort the LMS suffix recursively, we can induce the sorting of the entire suffix array

Building in u

```
x[10:]: i$$
x[7:]:  ipp i
x[1:]:  iss iss ipp i
x[4:]:  iss ipp i
x[8:]:  ppi
x[2:]:  ssi ssi ppi
x[5:]:  ssi ppi
```

- In skew we sorted triplets
- Build a table for the triplets
- Used lookups when building u

```
alpha = {
 “#”: 0,
 “i$$”: 1,
 “ipp”: 2,
 “iss”: 3,
 “ppi”: 4,
 “ssi”: 5
}
```

```
u[k] = alpha[x[i:i+3]]
```

Building in u

```
x[10:]: i$$
x[7:]:  ipp i
x[1:]:  iss iss ipp i
x[4:]:  iss ipp i
x[8:]:  ppi
x[2:]:  ssi ssi ppi
x[5:]:  ssi ppi
```

- In skew we sorted triplets
- Build a table for the triplets
- Used lookups when building u

We need to sort LMS strings. We don’t have a bound on their length, so we cannot use radix sort.

```
“ipp”: 2,
“iss”: 3,
“ppi”: 4,
“ssi”: 5
}
```

A simple table with LMS strings as keys will not work (we must compare keys for lookups, and we don’t have a bound on that)

```
u[k] = alpha[x[i:i+3]]
```

Sorting LMS strings in linear time…

Sorting strings (again)

Consider two "sorted" strings:

LLLLLS
a α

LLLS
b β

where the refs are:

ref: α

ref: β

When we sort L-runs, we sort prefixes down to (and including) the first S

When we sort S-runs, we sort prefixes down to (and including) the first L

We don't know the order of all strings

but we know the order of their prefixes

Cases:

a α
b β

Ordered if a ≠ b.
If a = b then ordered if suffixes are ordered

α
β

a α
β

α

Always ordered

b β

α
β

Ordered if α ≠ β. Otherwise, we do not know (we only know the ordering of the chars used for binning).

LLLLLS
a α

LLLS
b β

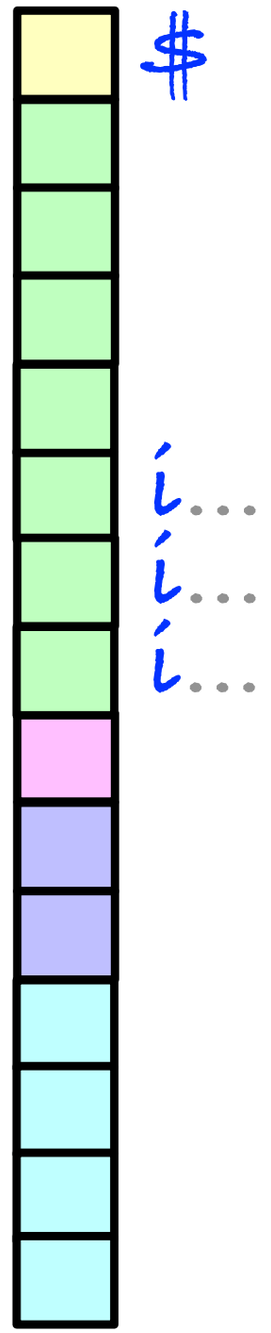
LLLLLS
a α

LLLS
b β

Sorting LMS strings

```
  *    *    *     *
LSSLLSSLLSSLLLS
miissiissiippi$
```

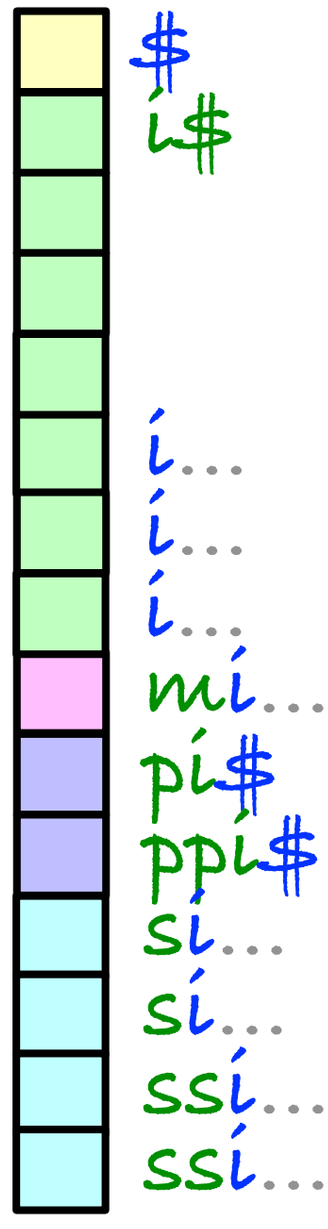
Put LMS suffixes in their bins,
but don't worry about their order
(we don't know their order yet).



Sorting LMS strings

```
  *   *   *     *
LSSLLSSLLSSLLLS
miissiissiippi$
```

Do an L scan to sort the runs of L before the LMS suffixes. After that, all L suffixes are sorted wrt to their prefix up to the LMS character.



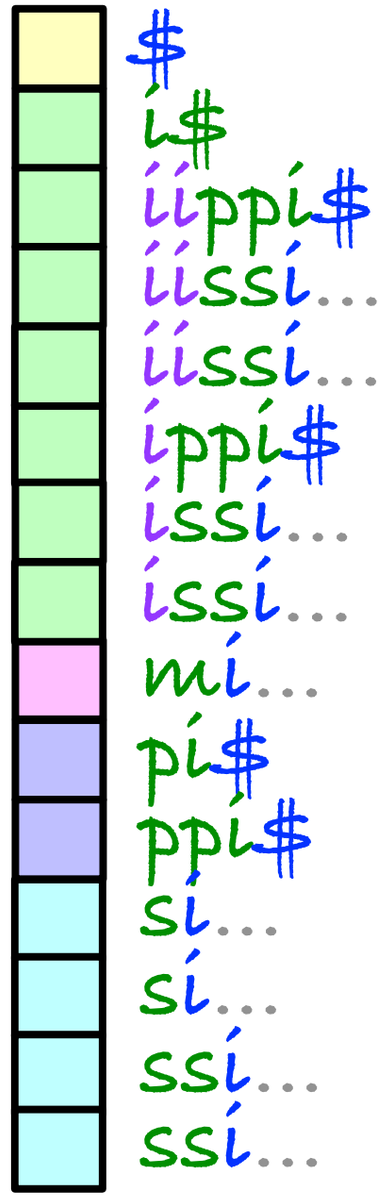
Sorting LMS strings

LSSLLSSLLSSLLLS

miissiissiippi$

Do an S scan to sort the runs of S before the leftmost Ls.

Now the strings are sorted with the prefix of S down to the first L, and from the first L to the next S.



Sorting LMS strings

* * * *

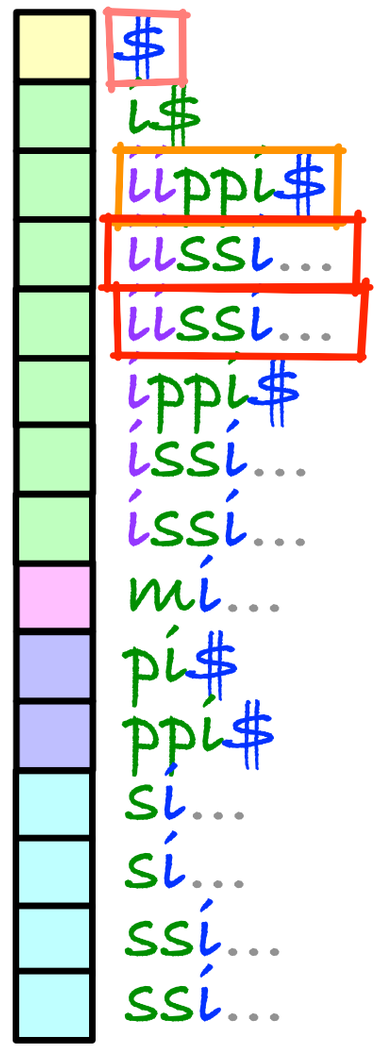
LSSLLSSLLSSLLLS

miissiissiippi$

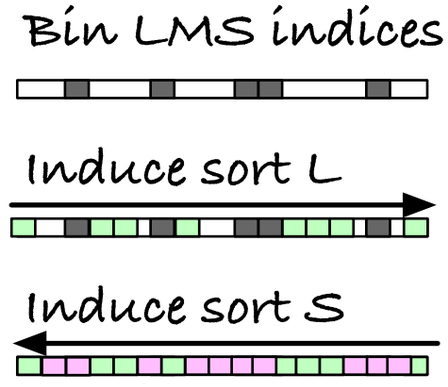
Do an S scan to sort the runs of S before the leftmost Ls.

Now the strings are sorted with the prefix of S down to the first L, and from the first L to the next S.

All LMS strings are sorted!



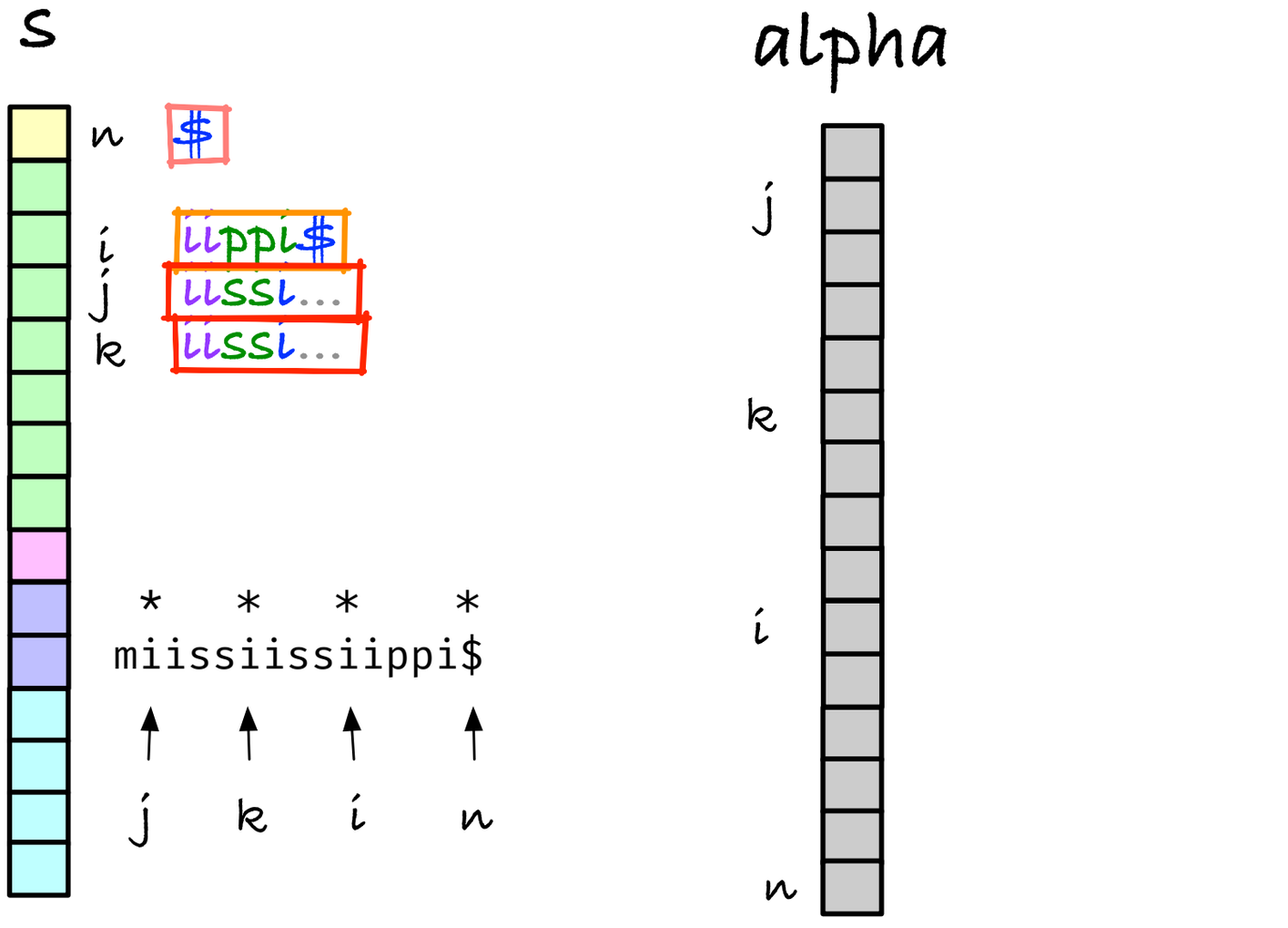
Building in u



- Sorting LMS strings ✓
- Build a table for the ~~triplets~~LMS strings
- Used lookups when building u

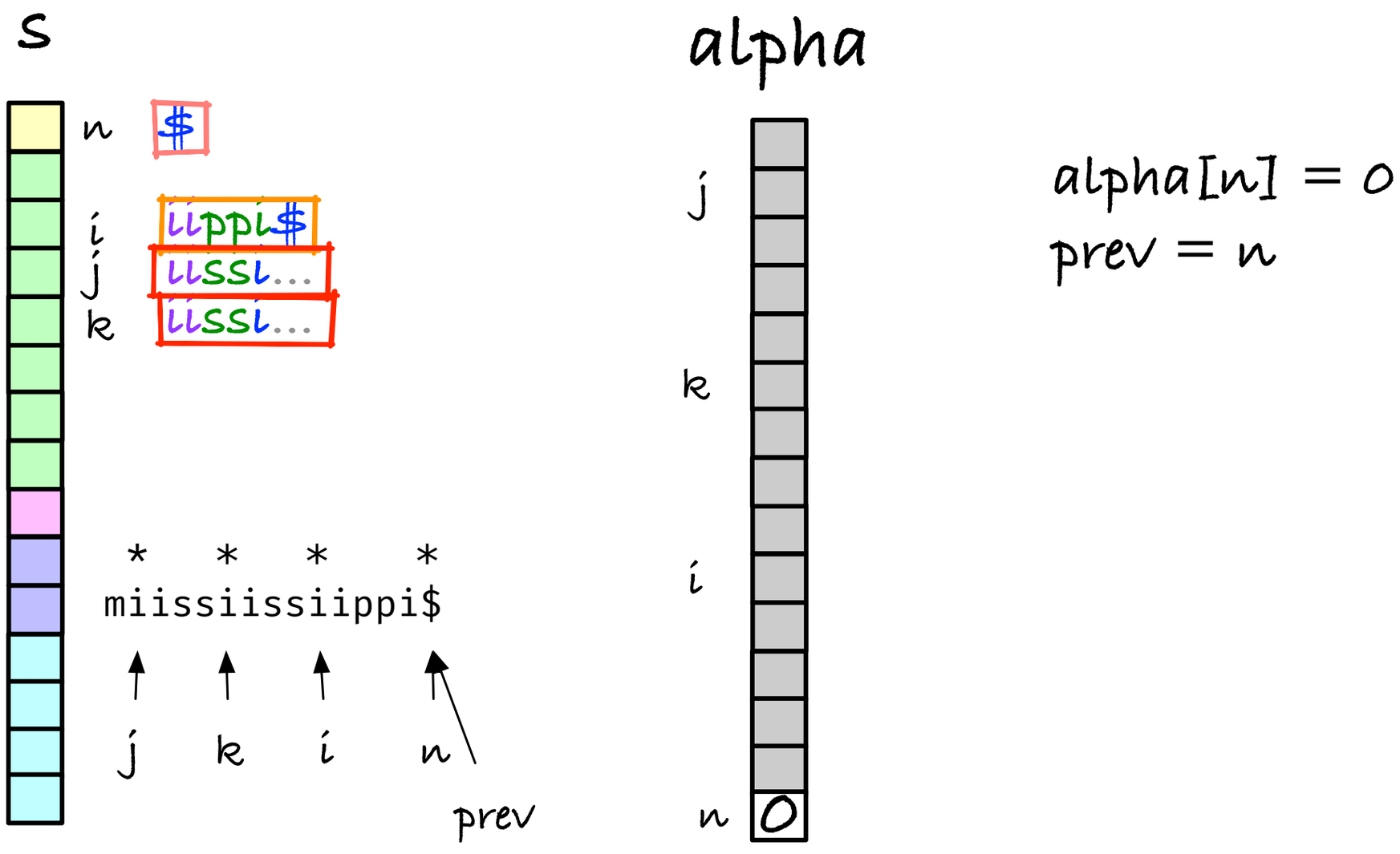
Building the alphabet table (in linear time)

Building table

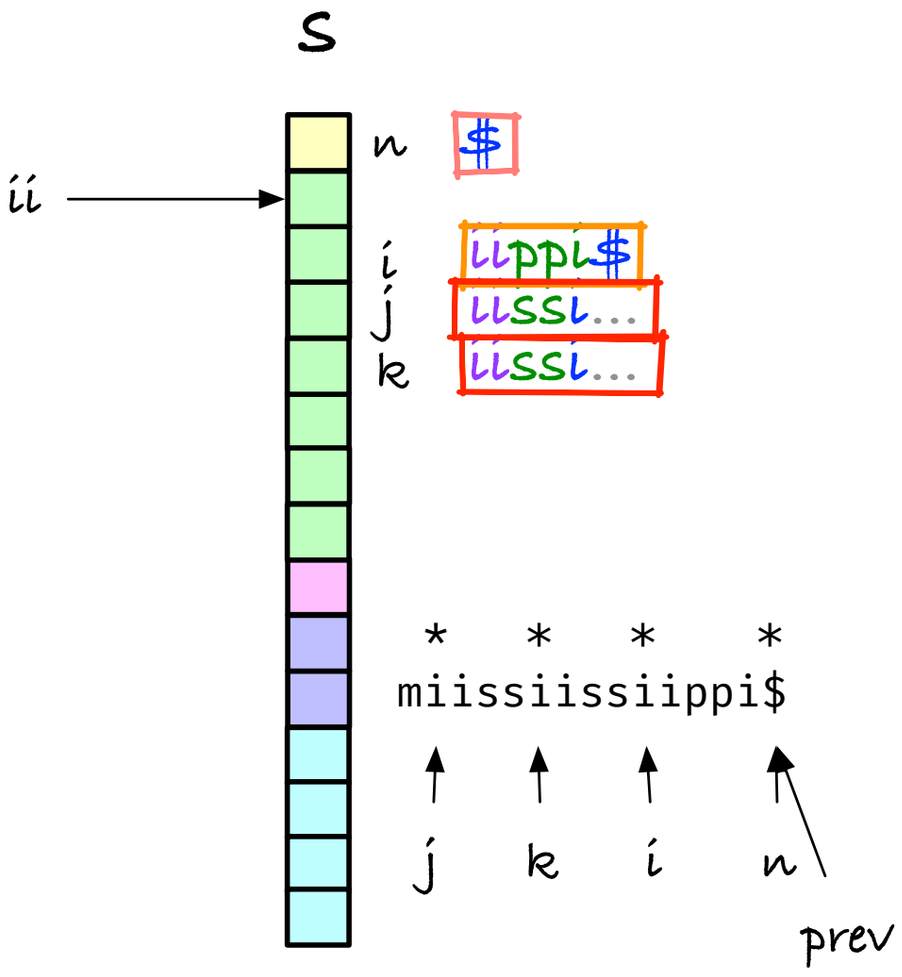


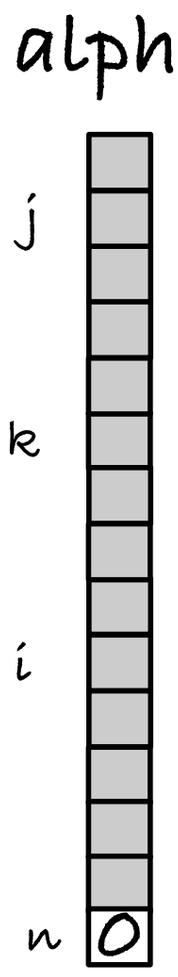
alpha an integer array
(selected indices on the left)

Building table



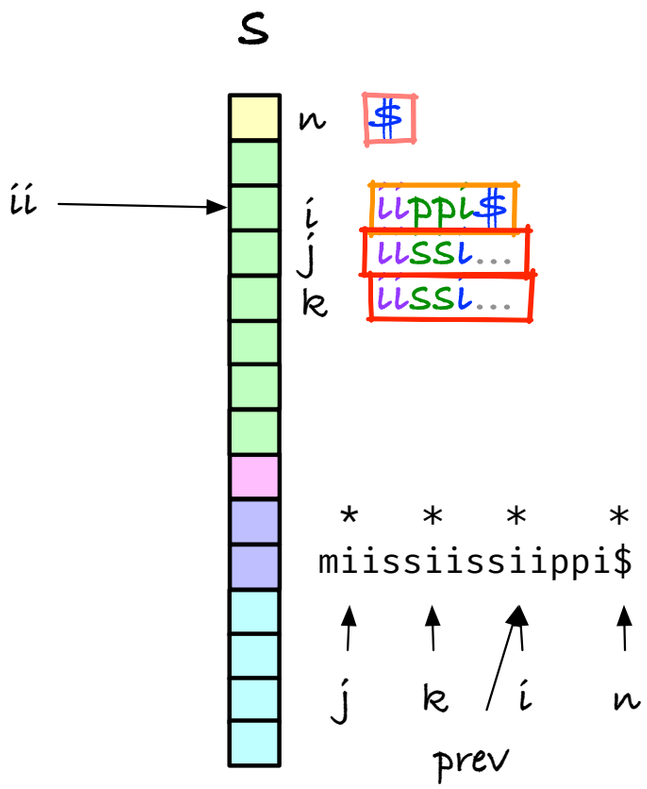
Building table

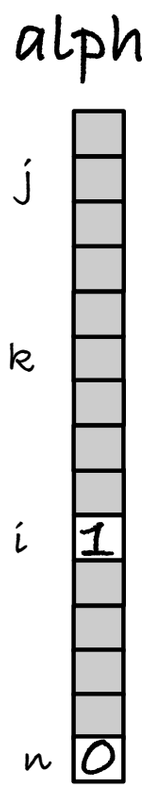





```
a = 0
Scan ii down from 1 to n+1

if is_lms[s[ii]]:
   if not lms_eq(s[ii],prev):
      a += 1
  alpha[s[ii]] = a
  prev = s[ii]
```

Building table

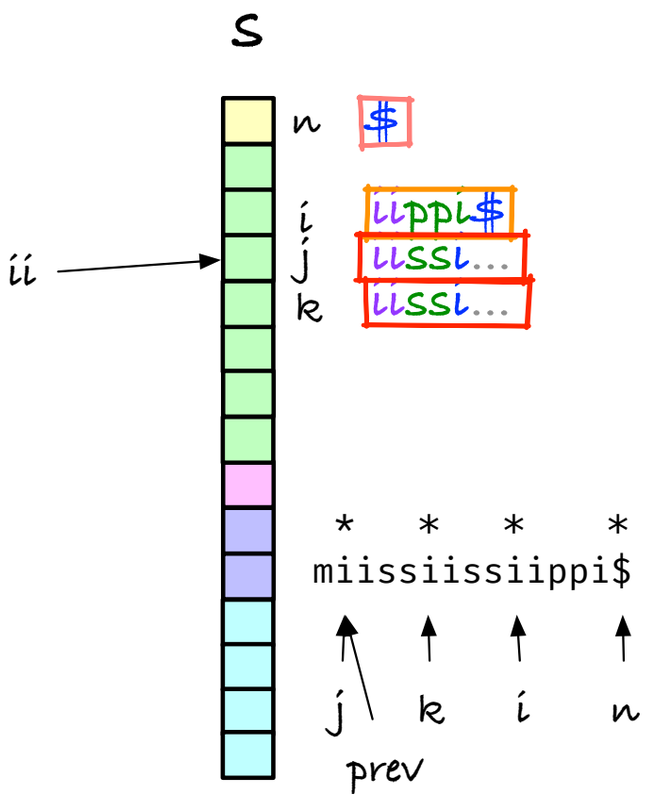


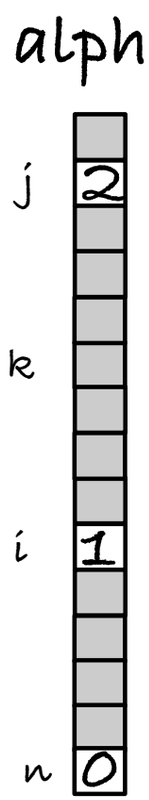



```
a = 0
Scan ii down from 1 to n+1

if is_lms[s[ii]]:
   if not lms_eq(s[ii],prev):
      a += 1
   alpha[s[ii]] = a
   prev = s[ii]
```

not lms_eq(liippi$, $)

Building table

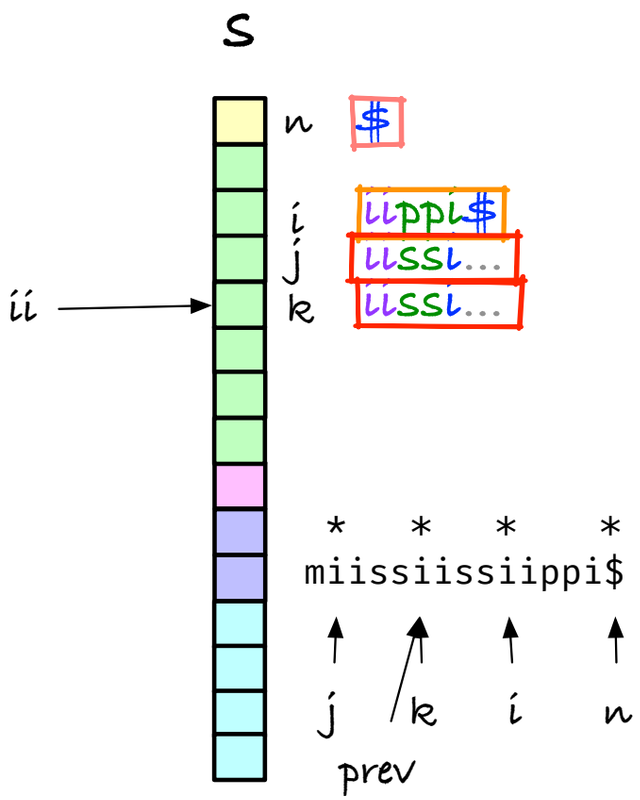


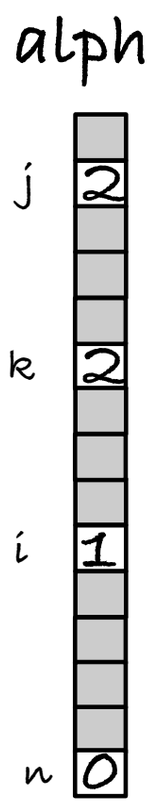



```
a = 0
Scan ii down from 1 to n+1

if is_lms[s[ii]]:
   if not lms_eq(s[ii],prev):
      a += 1
   alpha[s[ii]] = a
   prev = s[ii]
```

not lms_eq(llssi... , liippi$)

Building table





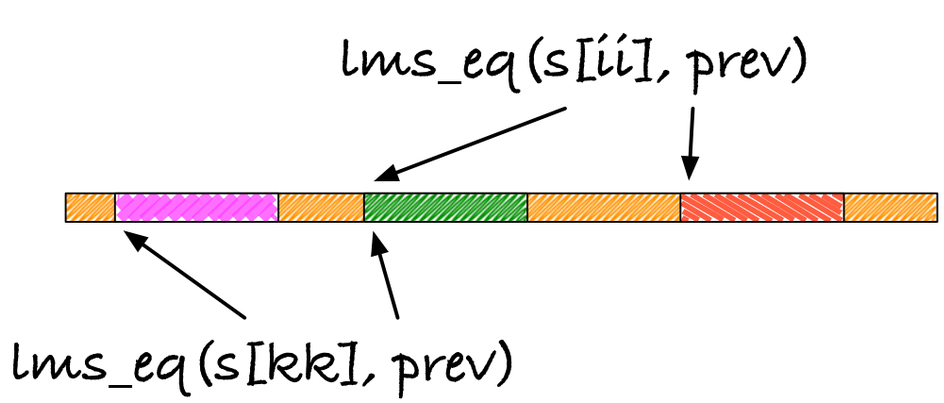

```
a = 0
Scan ii down from 1 to n+1

if is_lms[s[ii]]:
   if not lms_eq(s[ii],prev):
      a += 1
   alpha[s[ii]] = a
   prev = s[ii]
```

lms_eq(issi... , issi...)

Building table

`is_lms[i] = t[i] == L && t[i-1] == L`

`lms_eq(i,j) = string comparison that stops at the first LMS`

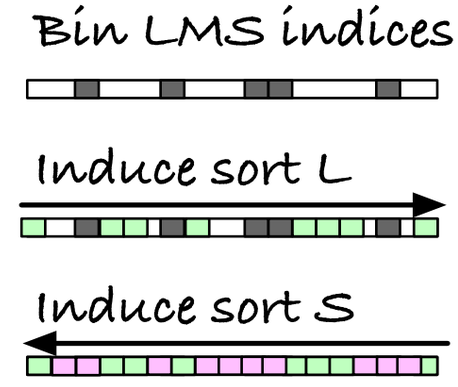


Each LMS string is part of at most two comparisons (one as s[ii] and one as prev)

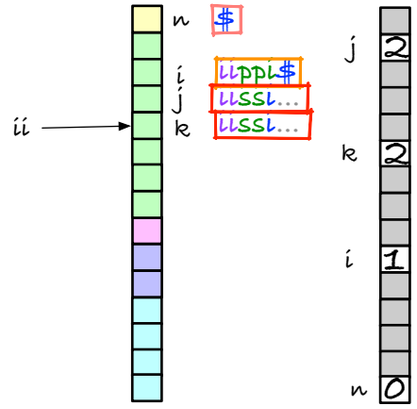
Total comparison = 2 · total LMS string length
= O(n)

Total building time:
O(n + comparison) = O(n)

Building in u



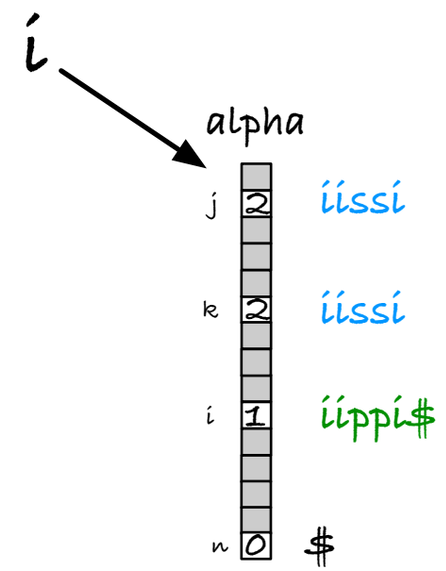
- Sorting LMS strings ✓
- Build a table for LMS strings ✓
- Used lookups when building u



Constructing the actual string (in linear time)

Constructing u

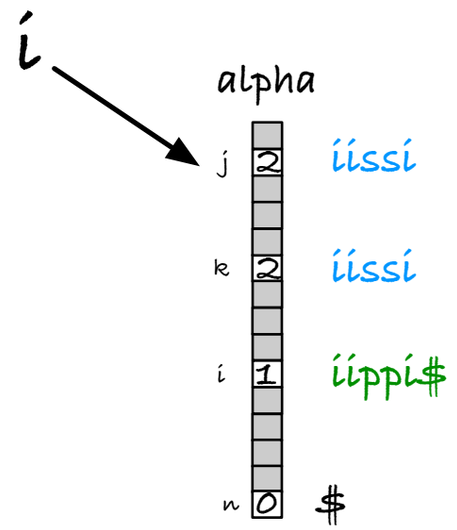
```
for (i = 0; i ⩽ n; i++) {
    if (is_lms[i]) {
        u[k++] = alpha[i];
    }
}
```

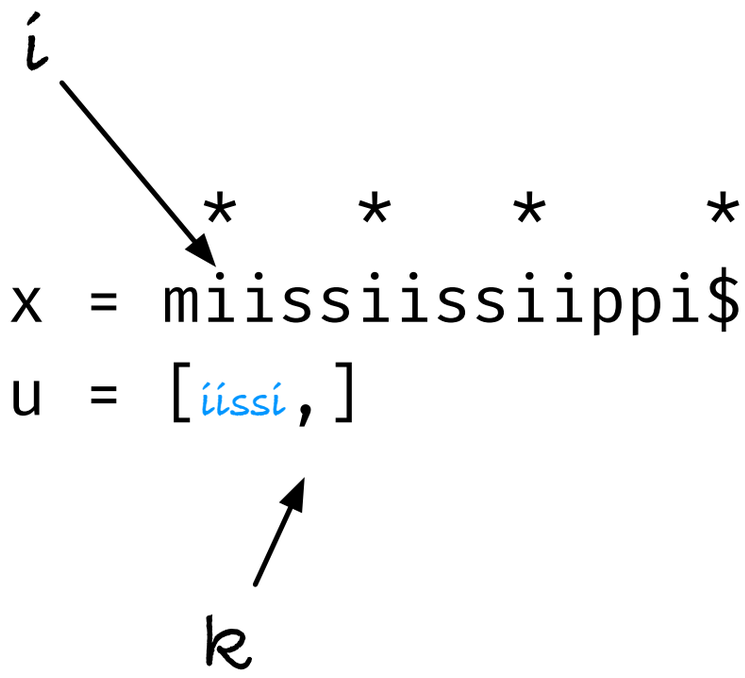



```
i
      *   *   *    *
x = miissiissiippi$
u = []
    k
```

Constructing u

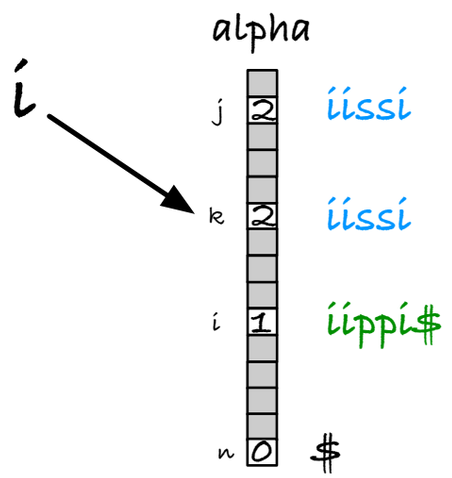
```
for (i = 0; i ⩽ n; i++) {
    if (is_lms[i]) {
        u[k++] = alpha[i];
    }
}
```





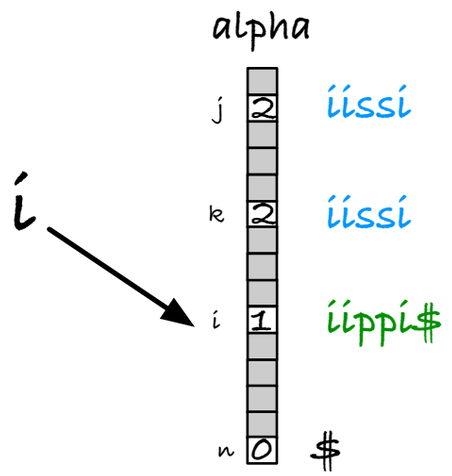
Constructing u

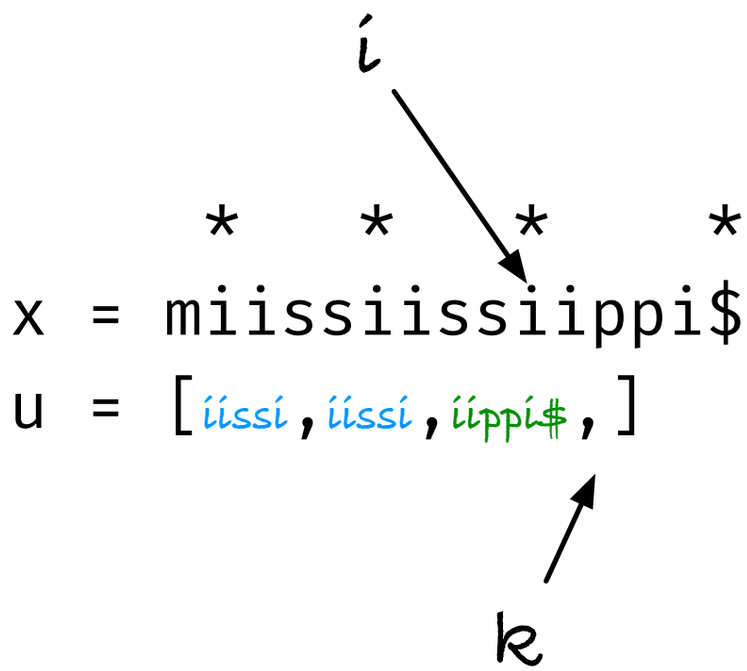
```
for (i = 0; i ⩽ n; i++) {
    if (is_lms[i]) {
        u[k++] = alpha[i];
    }
}
```



Constructing u

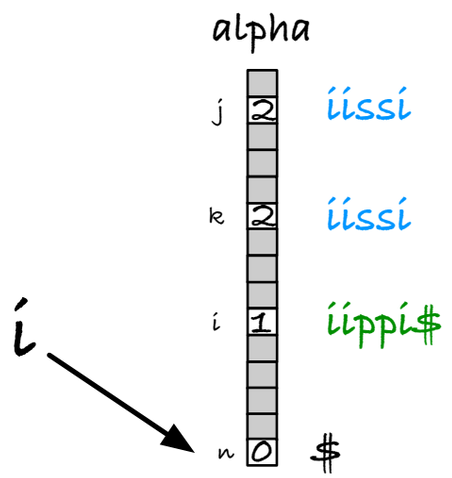
```
for (i = 0; i ⩽ n; i++) {
    if (is_lms[i]) {
        u[k++] = alpha[i];
    }
}
```





Constructing u

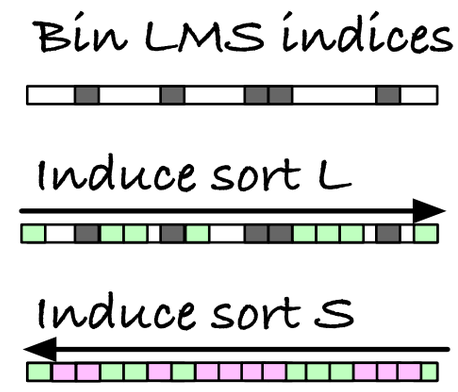
```
for (i = 0; i ⩽ n; i++) {
    if (is_lms[i]) {
        u[k++] = alpha[i];
    }
}
```

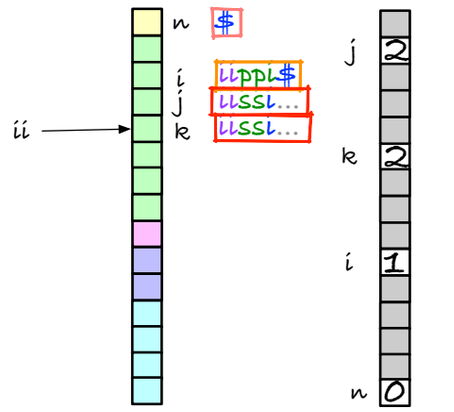


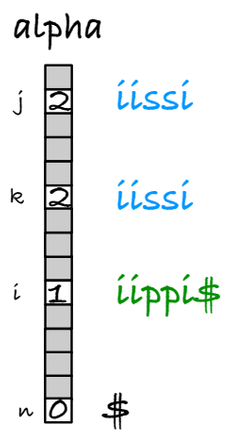
We don’t actually look at x at all here. We just extract the defined values in alpha, in the order they appear there. You can also exploit that to build u from alpha alone (if you have marked the non-used entries as undefined).

Building in u

- Sorting LMS strings ✓
- Build a table for LMS strings ✓
- Construct u without looking up LMS-strings ✓



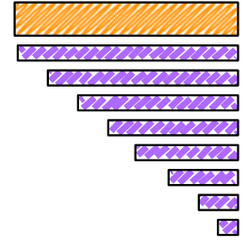


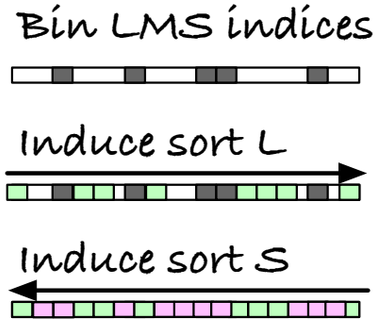


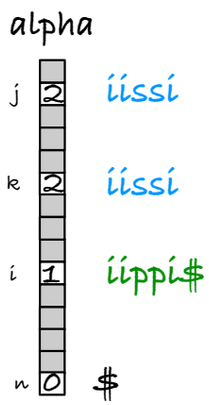

```
        *   *   *    *
x = miissiissiippi$
u = [iissi,iissi,iippi$,$]
```

Identify S/L classes and LMS indices

O(n) ✓

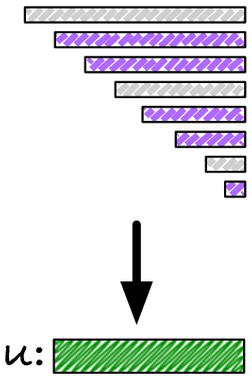




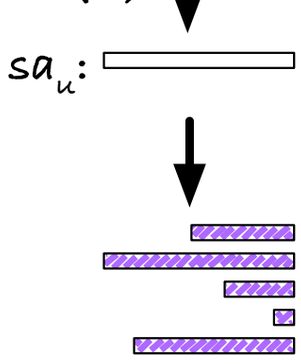


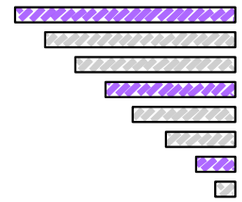
Sort LMS strings using induced sorting. Then get table and build u.

O(n)



sais(u)





Bin LMS indices
(preserve the order from sa_u)

Induce sort L

Induce sort S

O(n) ✓

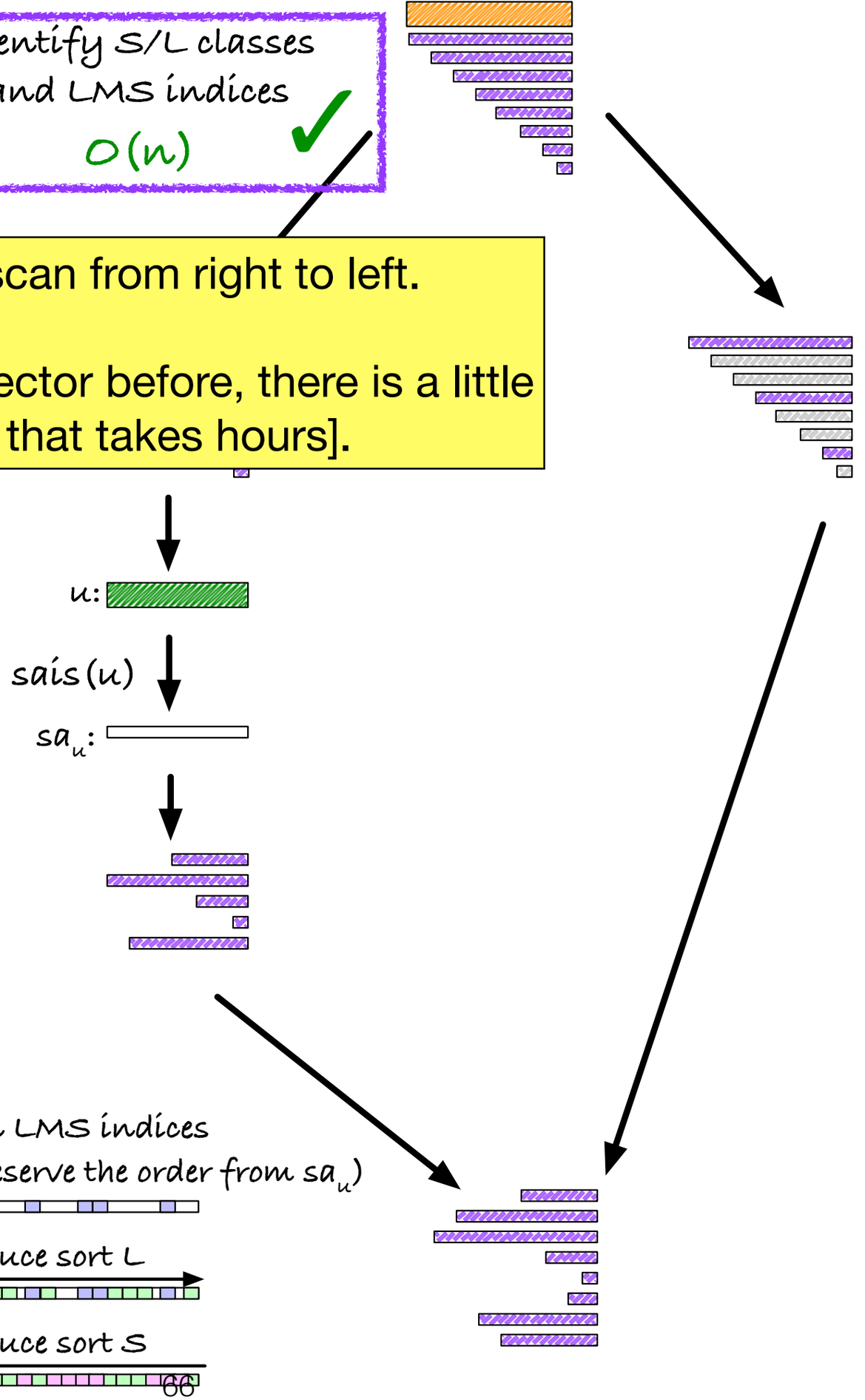
Identify S/L classes and LMS indices

O(n) ✓

Classifying the indices is a simple scan from right to left.
It is easy to implement and fast.
[If you haven't implemented a bit-vector before, there is a little more work, but it is not something that takes hours].

Sort LMS strings using induced sorting. Then get table and build u.

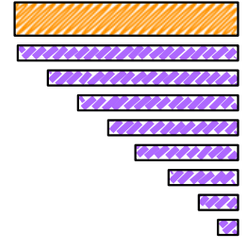
O(n)

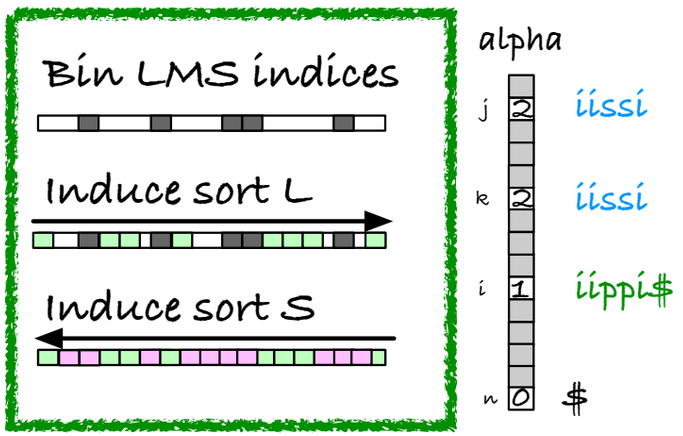


O(n) ✓

Identify S/L classes and LMS indices

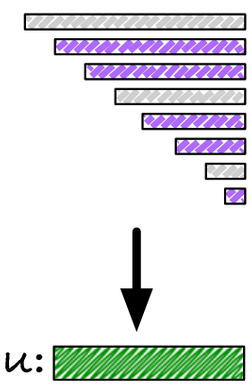
O(n)





Sort LMS strings using induced sorting. Then get table and build u.

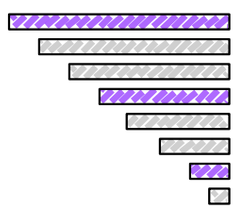
O(n)



u:

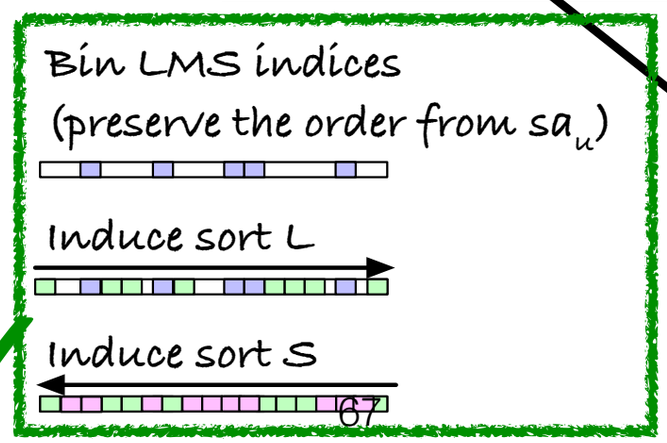
sais(u)

sa_u:

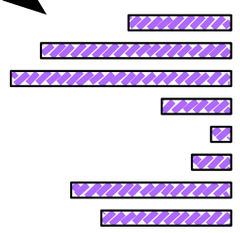


Induced sorting is almost trivial to implement.
You bin into arrays (in two directions) and that is all.

It is also insanely fast!

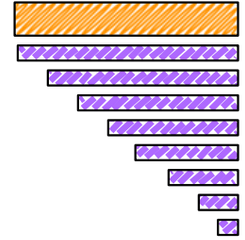


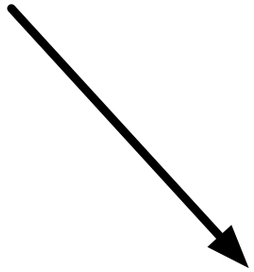
O(n)

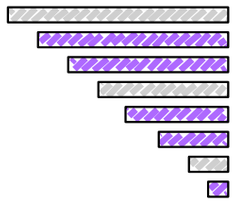


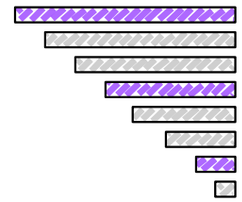
Identify S/L classes and LMS indices

O(n) ✓



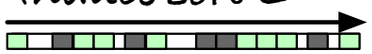




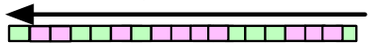


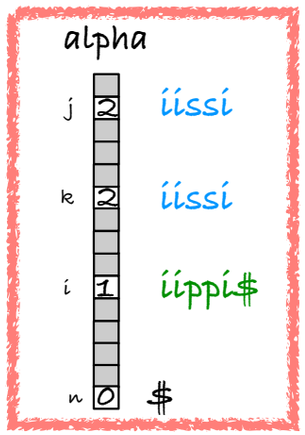
Bin LMS indices

Induce sort L



Induce sort S





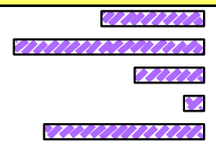
u:

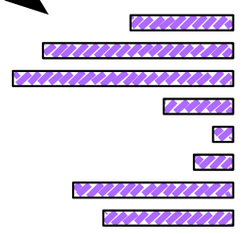
sais(u)

Sort LMS strings using induced sorting. Then get table and build u.

O(n)

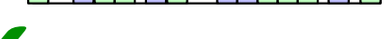
The hardest thing in building u is lms_eq(), and that is also pretty easy…



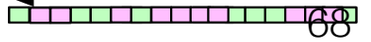


Bin LMS indices
(preserve the order from sa_u)

Induce sort L



Induce sort S



O(n) ✓

SAIS vs Skew

- SAIS is almost entirely induced sorting
  - It is a little harder to grok how it works
    - but it is not harder to implement than radix sorting
- Skew might be easier to understand than SAIS, but SAIS is easier to implement
- SAIS is also *much* faster in practice

Memory usage…

- As explained, each level in the recursion uses:
  - Bit-array for classes
  - Buckets
  - Arrays for induced sorting
  - Alphabet array
  - u and sa_u
  - Pointers for u to x for LMS indices

Memory usage…

- As explained, each level in the recursion uses:
  - Bit-array for classes
  - Buckets
  - Arrays for i
  - Alphabet a
  - u and sa_u
  - Pointers fo

You can do most of these computations in a single integer array and you can reuse this array between recursions.

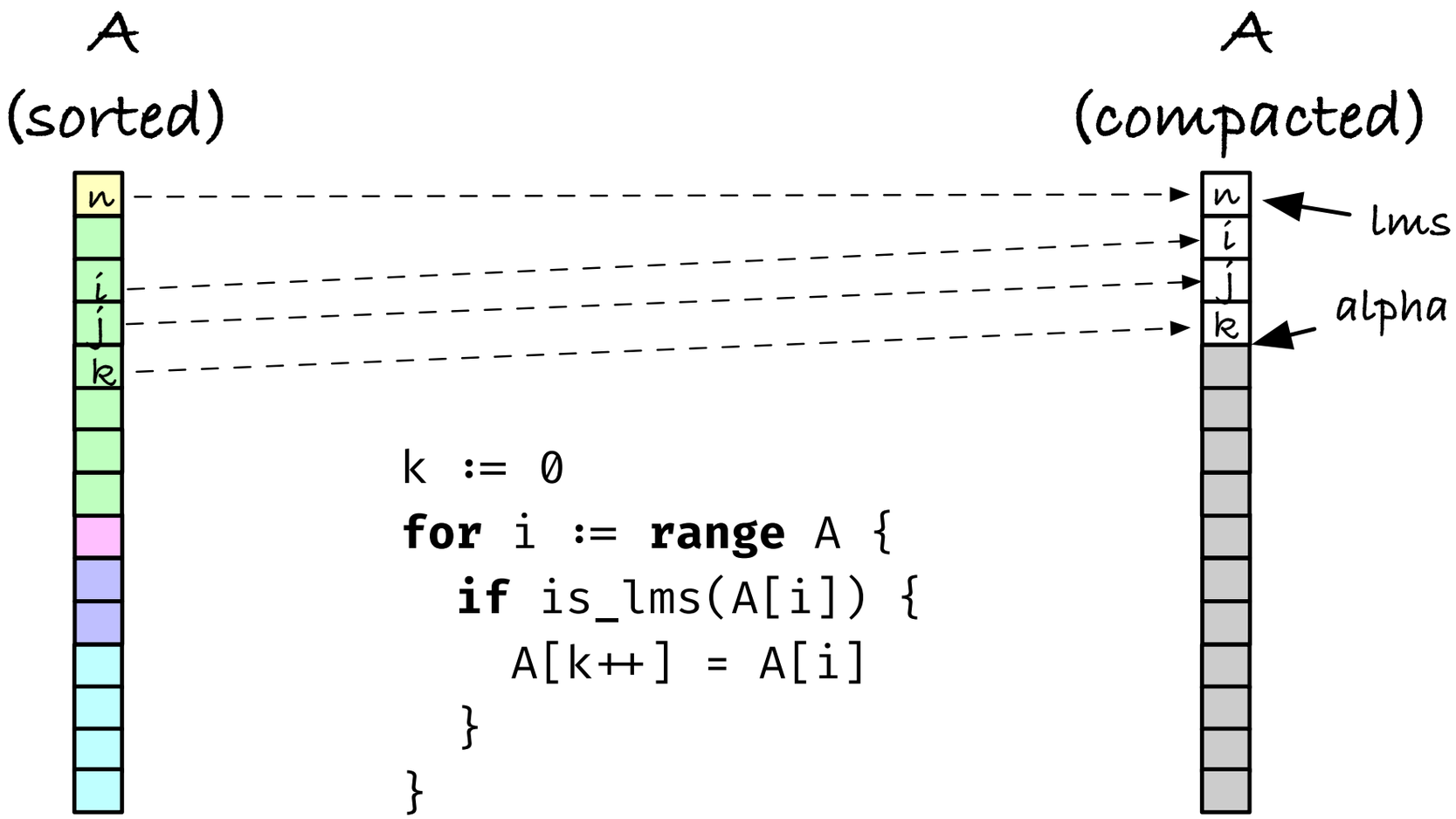
You only need the bit-vector and buckets as extra overhead. You can reuse the bit-vector between recursions (you can recompute it after the recursive call).

Then the overhead is O(n) **bits** and O(σ) for buckets.

For random strings, σ is O(1) because the expected length of LMS strings is bounded.

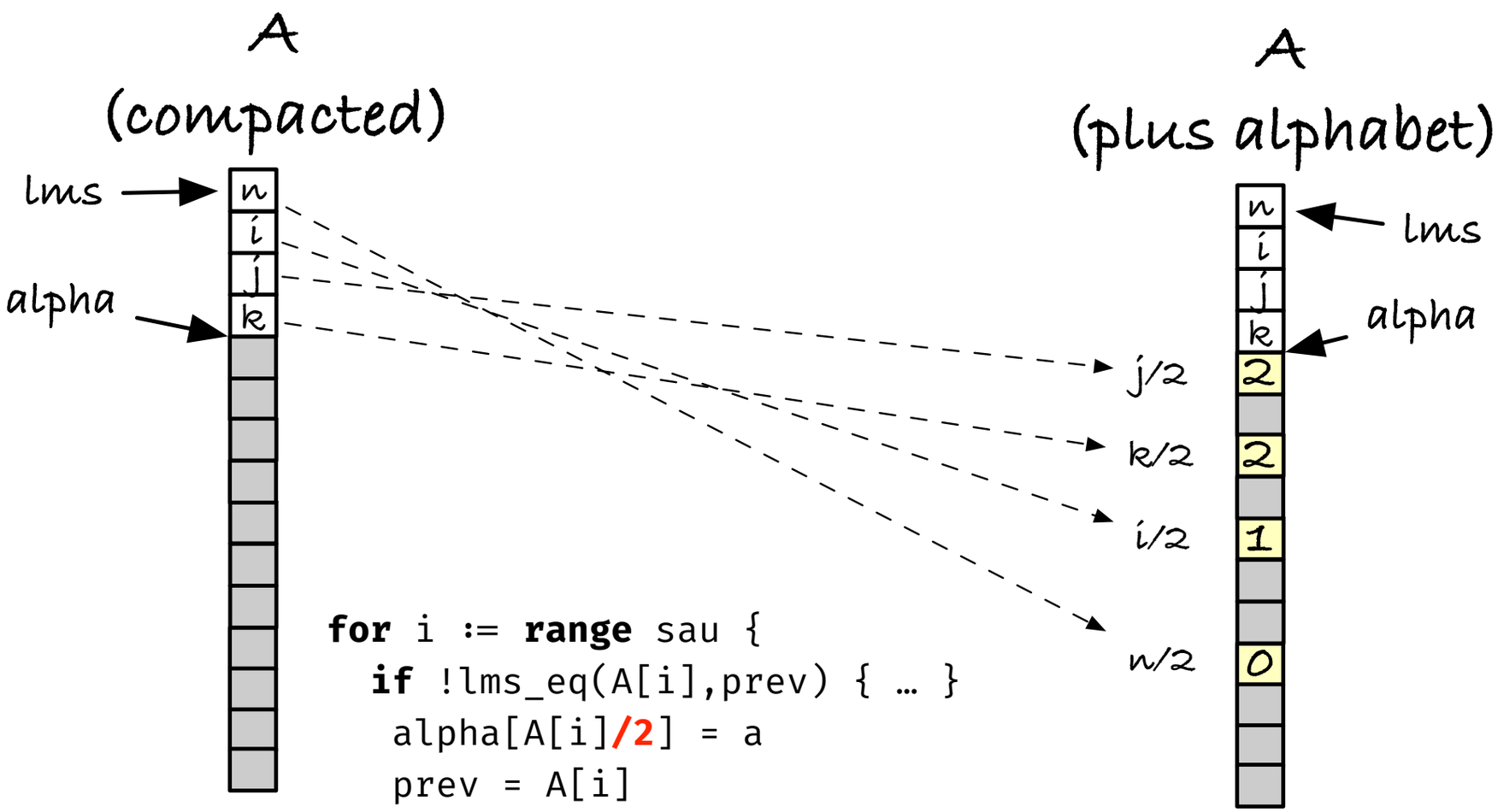
Most cases (probabilistic), the memory overhead is O(n) **bits** (in addition to x and sa).

Saving memory

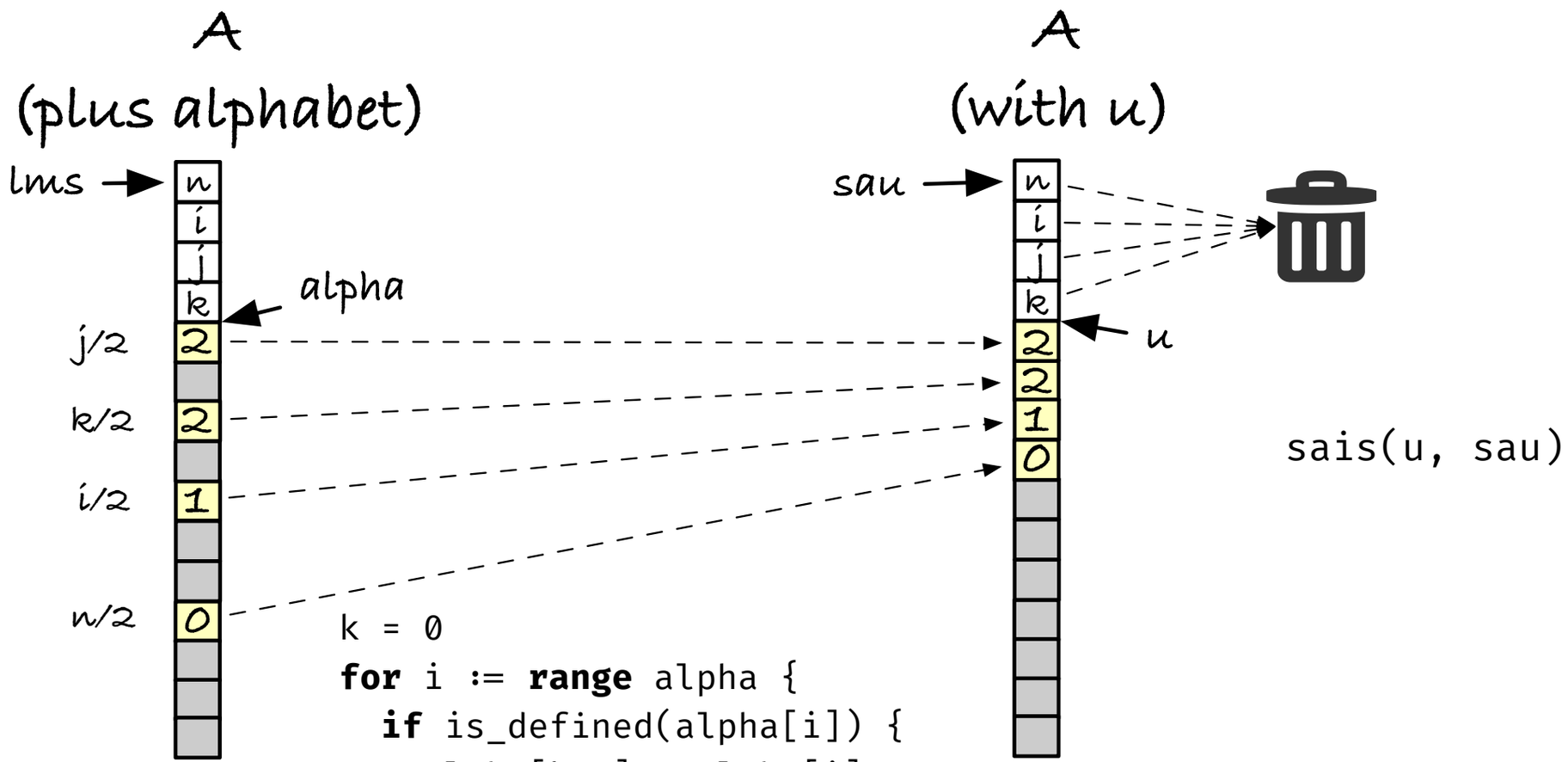



```
k := 0
for i := range A {
  if is_lms(A[i]) {
    A[k++] = A[i]
  }
}
lms := A[:k]
alpha := A[k:]
alpha[:] = undefined
```

Saving memory

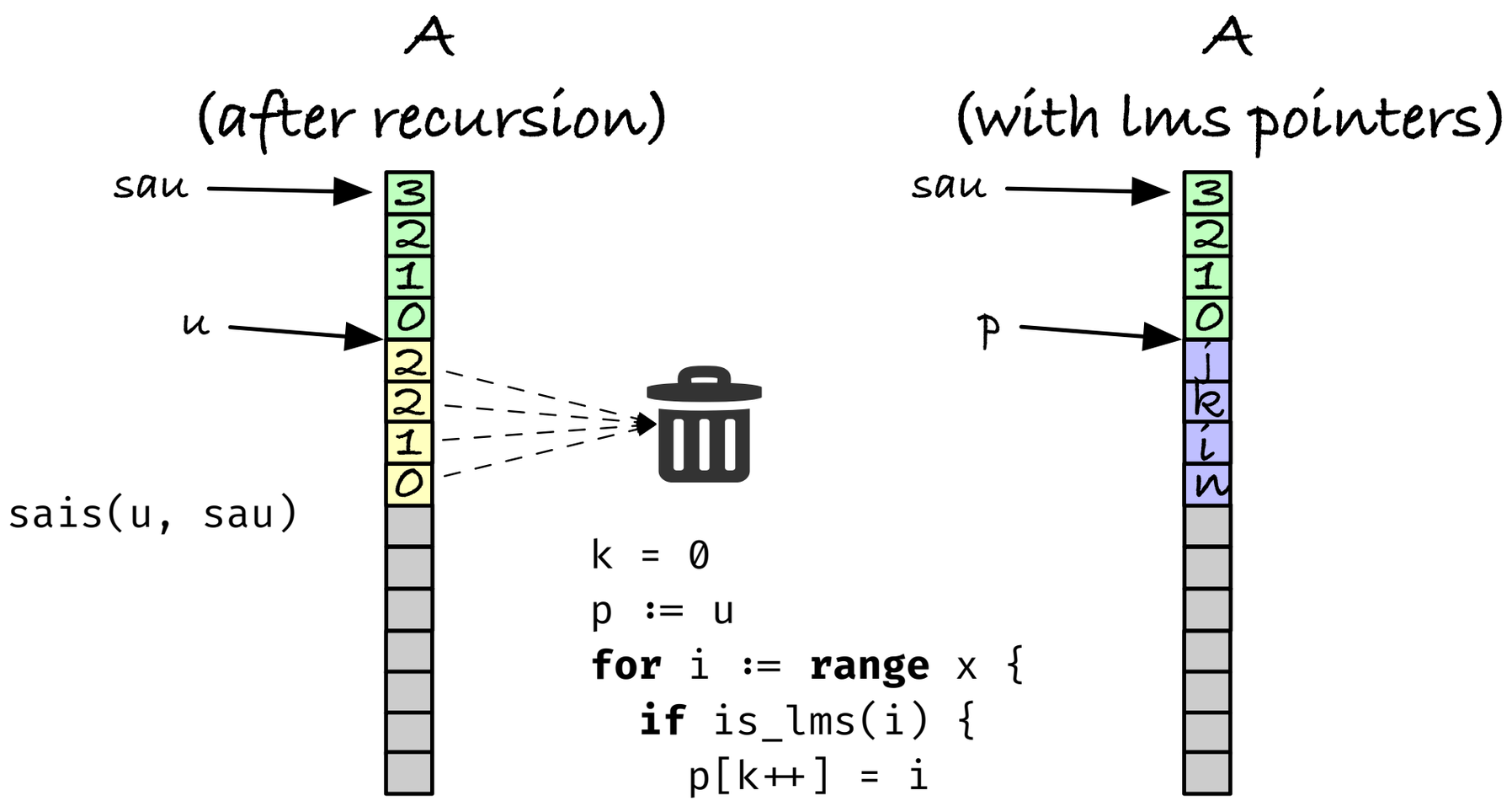


Saving memory




```
k = 0
for i := range alpha {
  if is_defined(alpha[i]) {
    alpha[k++] = alpha[i]
  }
}
u := alpha[:k]
sau := lms
```

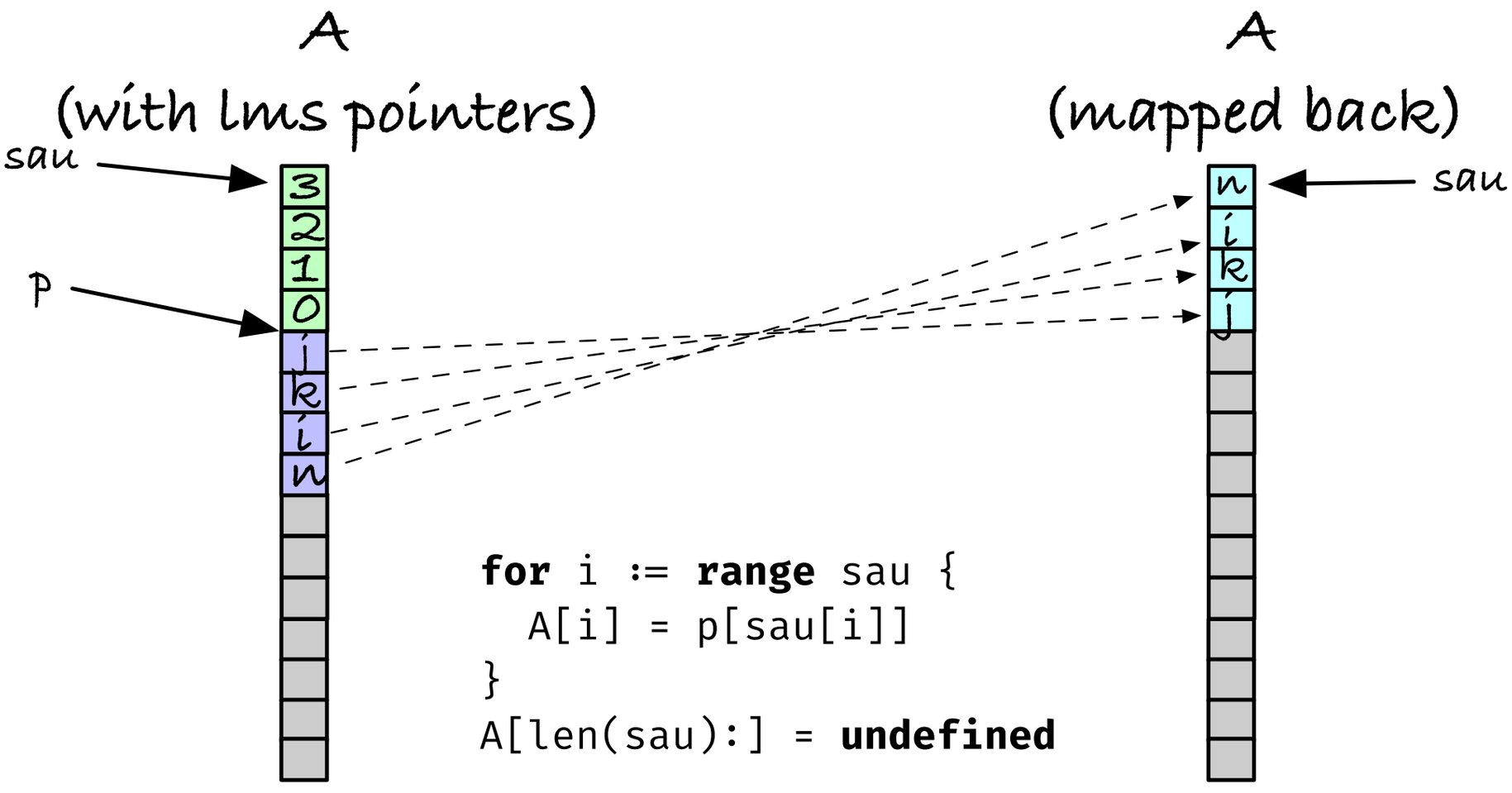
Saving memory



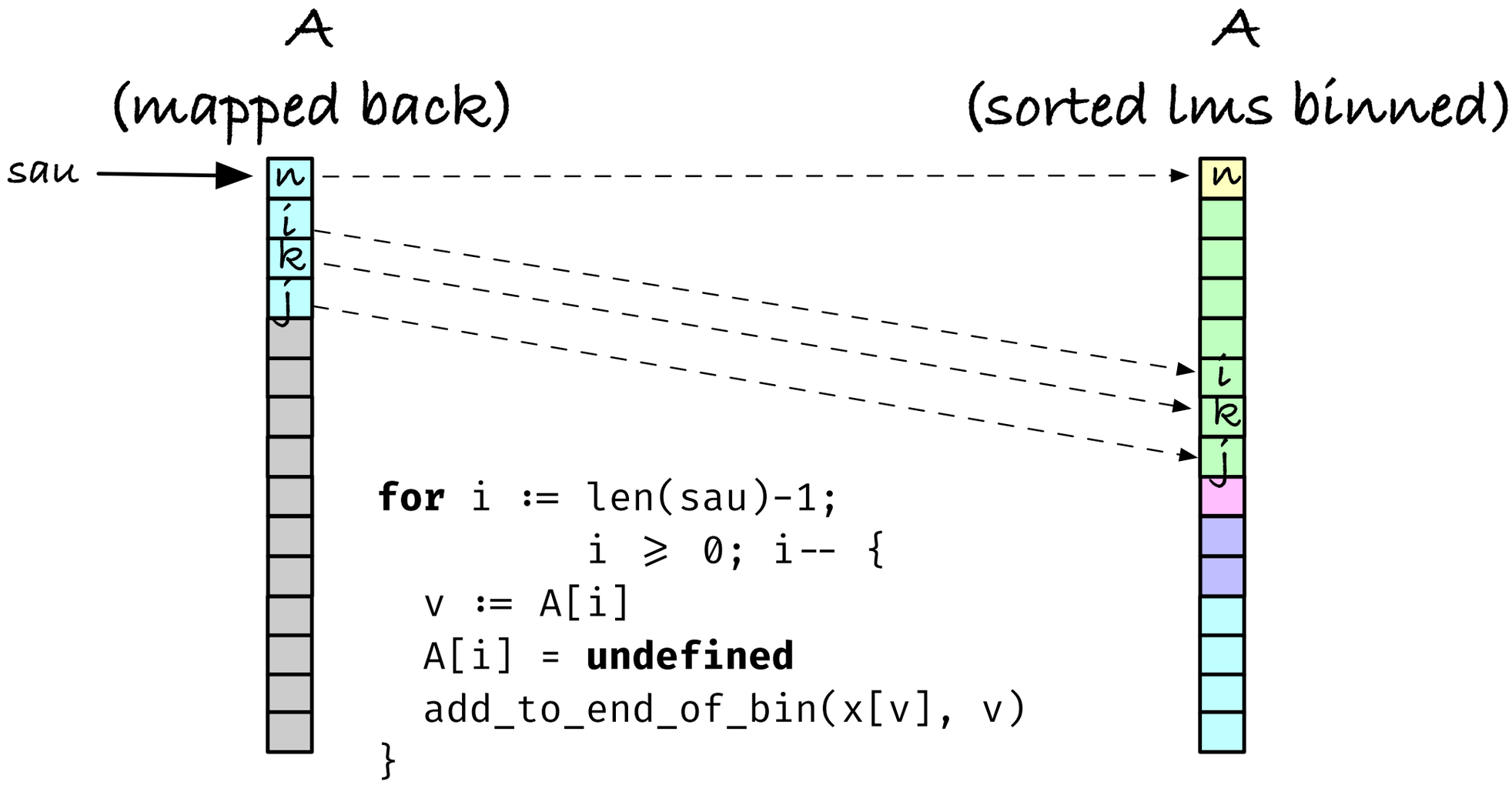

```
sais(u, sau)
```

```
k = 0
p := u
for i := range x {
  if is_lms(i) {
    p[k++] = i
  }
}
```

Saving memory



Saving memory




```
for i := len(sau)-1;
         i ⩾ 0; i-- {
  v := A[i]
  A[i] = undefined
  add_to_end_of_bin(x[v], v)
}
```

Saving memory

- The total memory you need is
  - The array A [O(n) integers]
    - This is always needed as it is also the output SA, so we usually don’t count this as space usage
  - The L/S bit-array [O(n) bits]
    - You can reuse the same bit-array in each recursion if you recompute it after the recursive call
  - The array for buckets [O(???) but expected O(1)]

Saving memory

- The total memory you need is
  - The array A [O(n) integers]
    - This is always needed as it is also the output SA, so we usually don’t count this as space usage
  - The L[ ] [O(n) integers]
    - You can reuse the same bit-array in each recursion if you recompute it after the recursive call
  - The array for buckets [O(???) but expected O(1)]

If you use strings as input, but need to translate them into integer arrays, then there is a real O(n) integers overhead that you can’t get rid of.


That's all Folks!