Suffix array construction

The “skew” algorithm

Sorting suffixes

mississippi$
ississippi$
ssissippi$
sissippi$
issippi$
ssippi$
sippi$
ippi$
ppi$
pi$
i$
$

Sorting suffixes...

$
i$
ippi$
issippi$
ississippi$
mississippi$
pi$
ppi$
sippi$
sissippi$
ssippi$
ssissippi$

How long would it take with a standard sort algorithm (like quick sort?)

O(n log n) comparisons; O(n) per comparison. O(n² log n) in total.

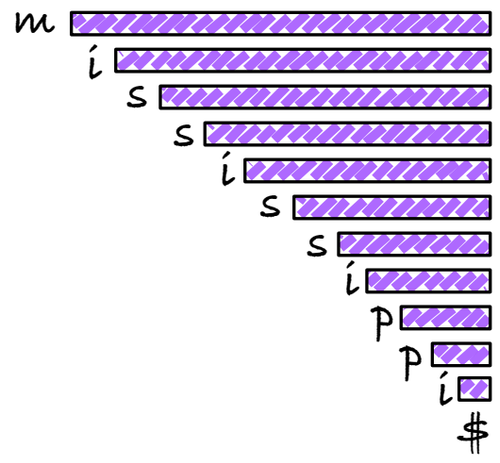
Bucket sort

Counts: $ i m p s

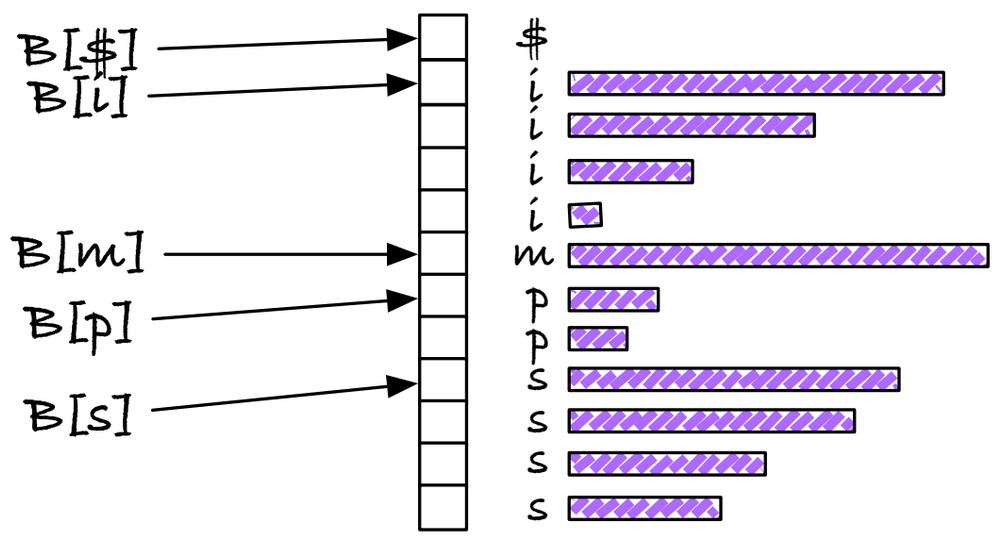
1 4 1 2 4

Buckets: $ i m p s

(acc sum of counts) 0 1 5 6 8

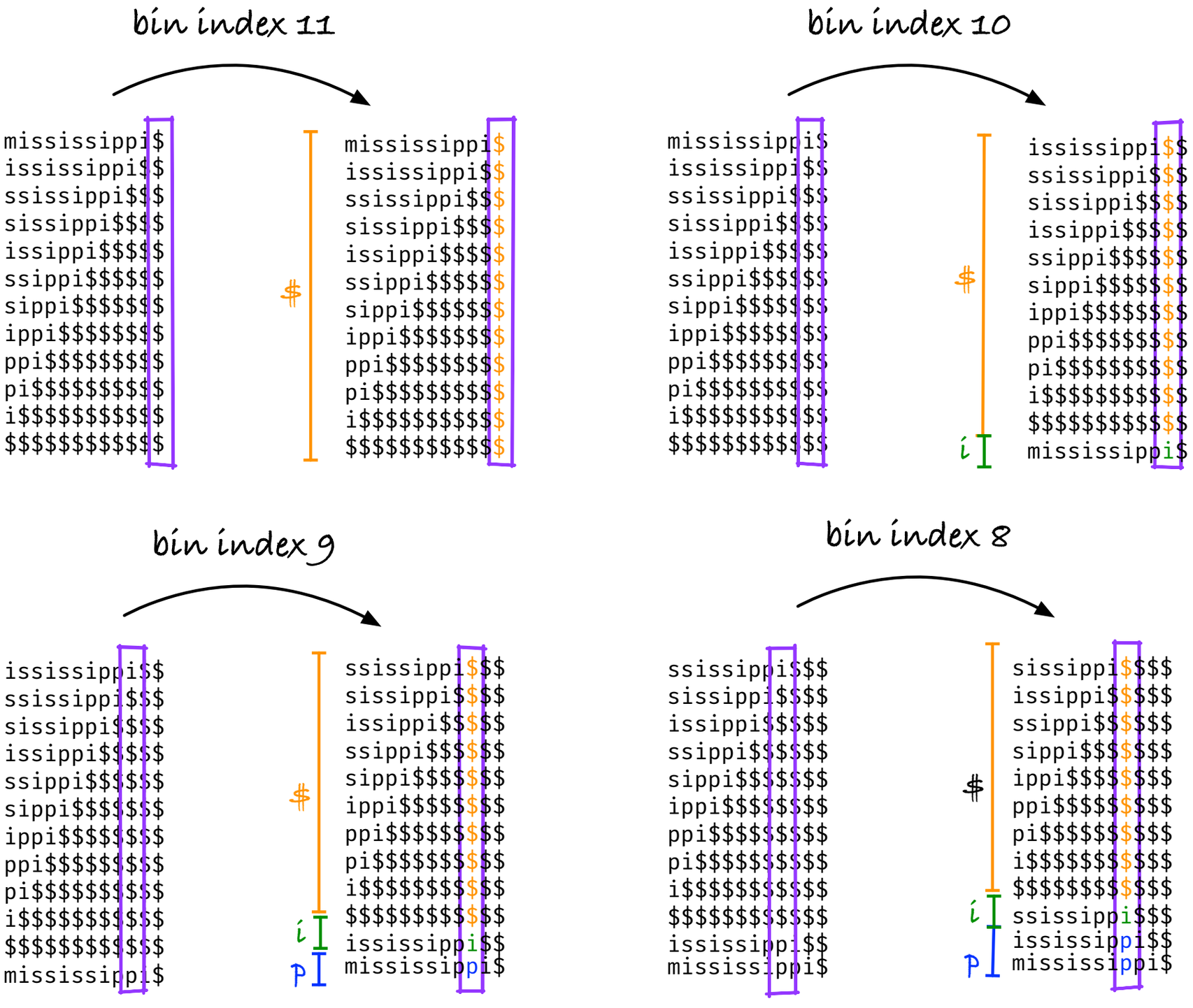


$$A[B[x[i]]++] = i$$

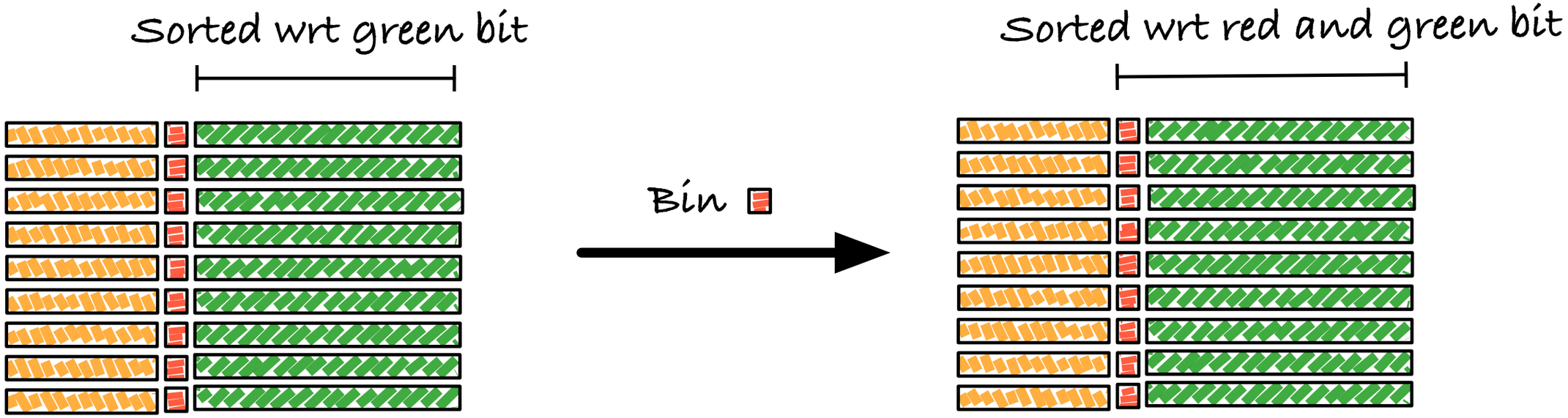


Running time: $O(n + \sigma) = O(n)$

Radix sort


bin index 11
mississippi$
ississippi$$
ssissippi$$$
sissippi$$$$
issippi$$$$$
ssippi$$$$$$
sippi$$$$$$$
ippi$$$$$$$$
ppi$$$$$$$$$
pi$$$$$$$$$$
i$$$$$$$$$$$
$$$$$$$$$$$$
$
mississippi$
ississippi$$
ssissippi$$$
sissippi$$$$
issippi$$$$$
ssippi$$$$$$
sippi$$$$$$$
ippi$$$$$$$$
ppi$$$$$$$$$
pi$$$$$$$$$$
i$$$$$$$$$$$
$$$$$$$$$$$$
bin index 10
mississippi$
ississippi$$
ssissippi$$$
sissippi$$$$
issippi$$$$$
ssippi$$$$$$
sippi$$$$$$$
ippi$$$$$$$$
ppi$$$$$$$$$
pi$$$$$$$$$$
i$$$$$$$$$$$
$$$$$$$$$$$$
$
i
ississippi$$
ssissippi$$$
sissippi$$$$
issippi$$$$$
ssippi$$$$$$
sippi$$$$$$$
ippi$$$$$$$$
ppi$$$$$$$$$
pi$$$$$$$$$$
i$$$$$$$$$$$
$$$$$$$$$$$$
mississippi$
bin index 9
ississippi$$
ssissippi$$$
sissippi$$$$
issippi$$$$$
ssippi$$$$$$
sippi$$$$$$$
ippi$$$$$$$$
ppi$$$$$$$$$
pi$$$$$$$$$$
i$$$$$$$$$$$
$$$$$$$$$$$$
mississippi$
$
i
p
ssissippi$$$
sissippi$$$$
issippi$$$$$
ssippi$$$$$$
sippi$$$$$$$
ippi$$$$$$$$
ppi$$$$$$$$$
pi$$$$$$$$$$
i$$$$$$$$$$$
$$$$$$$$$$$$
ississippi$$
mississippi$
bin index 8
ssissippi$$$
sissippi$$$$
issippi$$$$$
ssippi$$$$$$
sippi$$$$$$$
ippi$$$$$$$$
ppi$$$$$$$$$
pi$$$$$$$$$$
i$$$$$$$$$$$
$$$$$$$$$$$$
ississippi$$
mississippi$
$
i
p
sissippi$$$$
issippi$$$$$
ssippi$$$$$$
sippi$$$$$$$
ippi$$$$$$$$
ppi$$$$$$$$$
pi$$$$$$$$$$
i$$$$$$$$$$$
$$$$$$$$$$$$
ssissippi$$$
ississippi$$
mississippi$

Radix sort



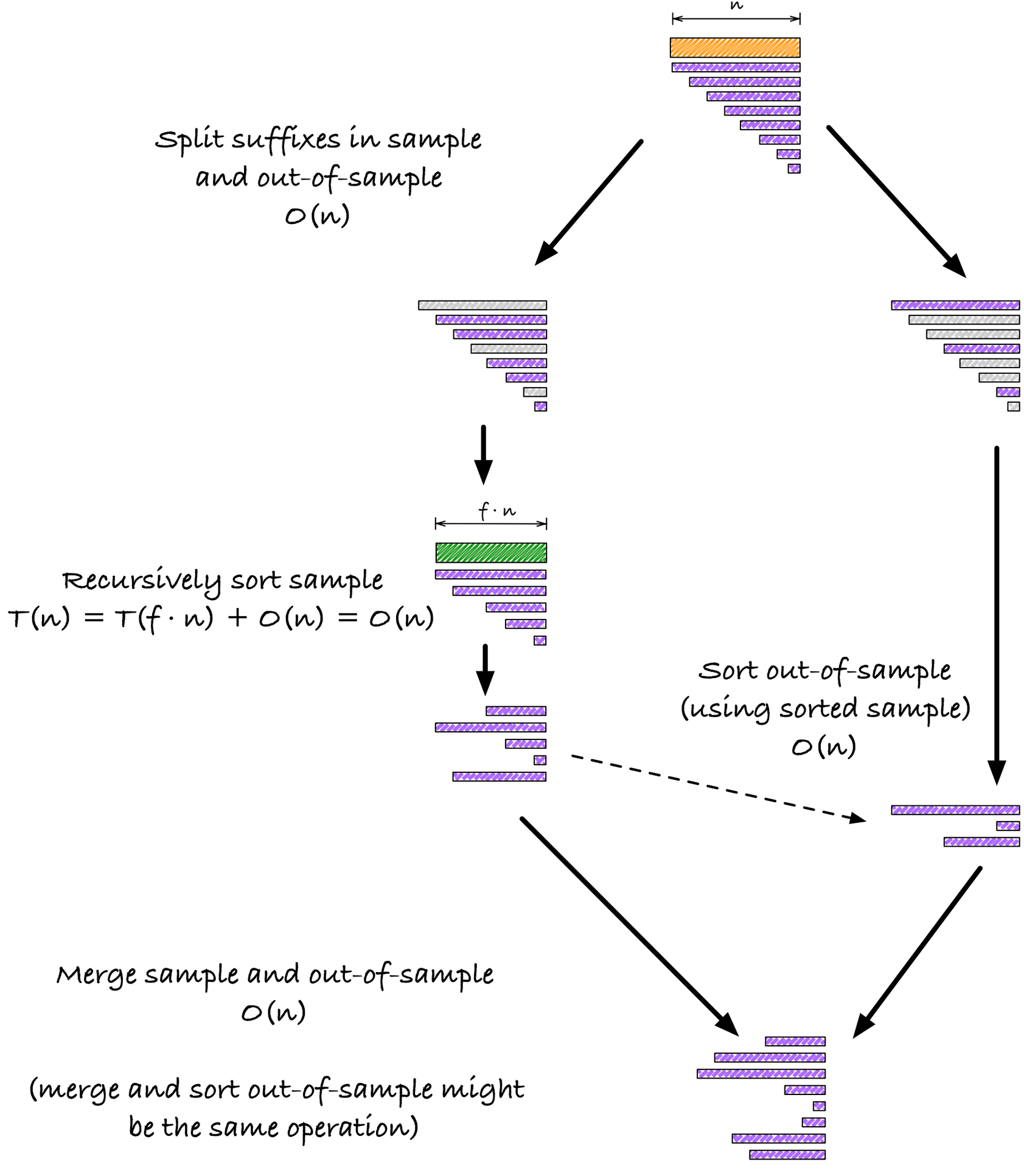
O(n) per bucket sort.

$O(n^2)$ for full radix sort.

Radix sort

- You can also radix sort from left to right by handling bins recursively
- Then you can terminate early when bins have size one
- but there is a little more overhead to it
- We won’t be radix sorting long strings, so you can just use the simplest approach when you need to sort…

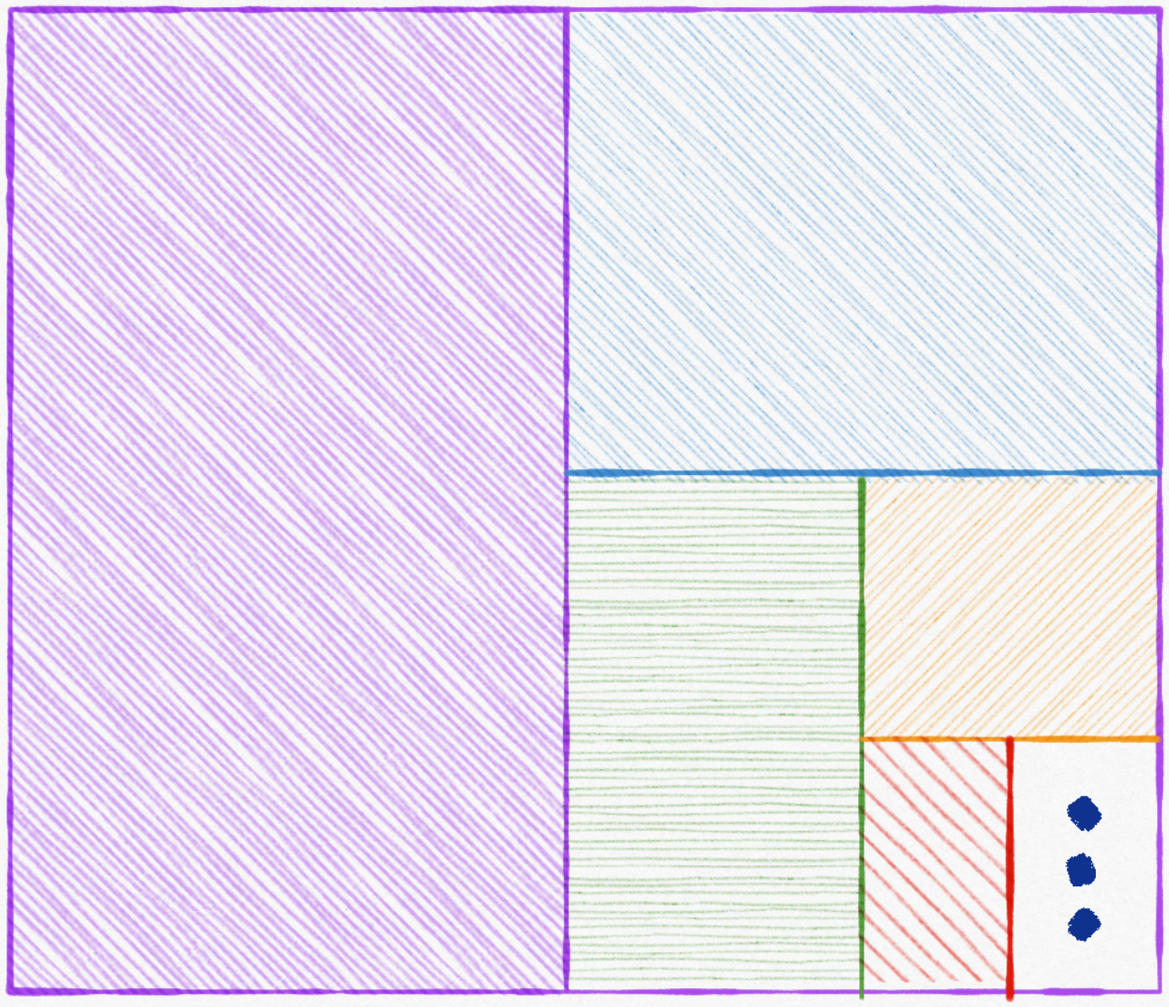
Sampling approach (to get to O(n))



Recurrence

$$T(n) = T(f \cdot n) + O(n)$$

$$T(n) \in O\left(n \sum_{i=0} f^i\right)$$



$$n + n/2 + n/4 + n/8 + n/16 + \cdots$$

$$\leq 2n$$

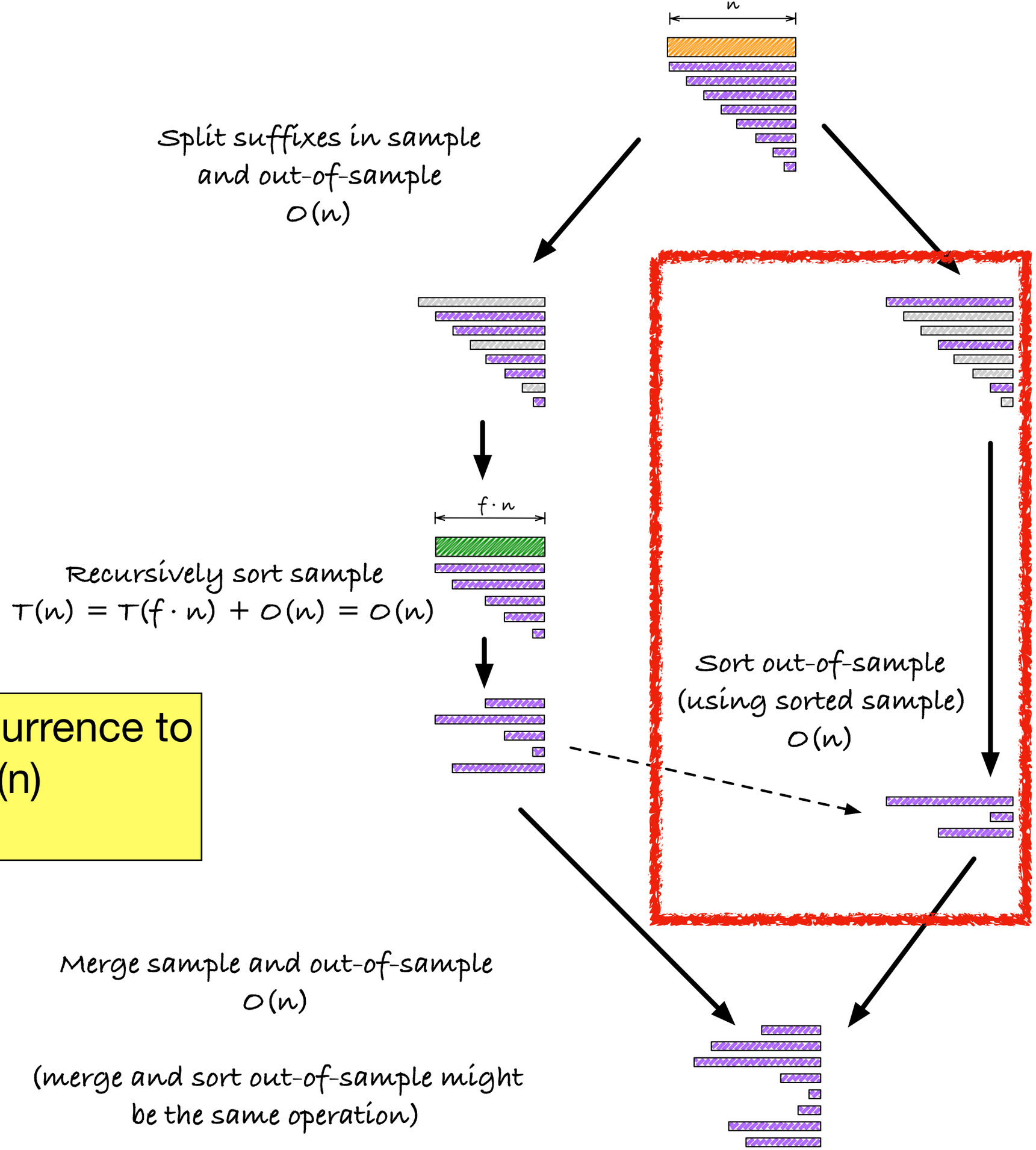
Recurrence

$$T(n) = T(f \cdot n) + O(n)$$

$$T(n) \in O\left(n \sum_{i=0} f^i \right)$$

$$\sum_{k=0}^{\infty} f^k = \frac{1}{1-f} \text{ for } 0 \leq f < 1$$

Why don't we sort out-of-sample recursively?



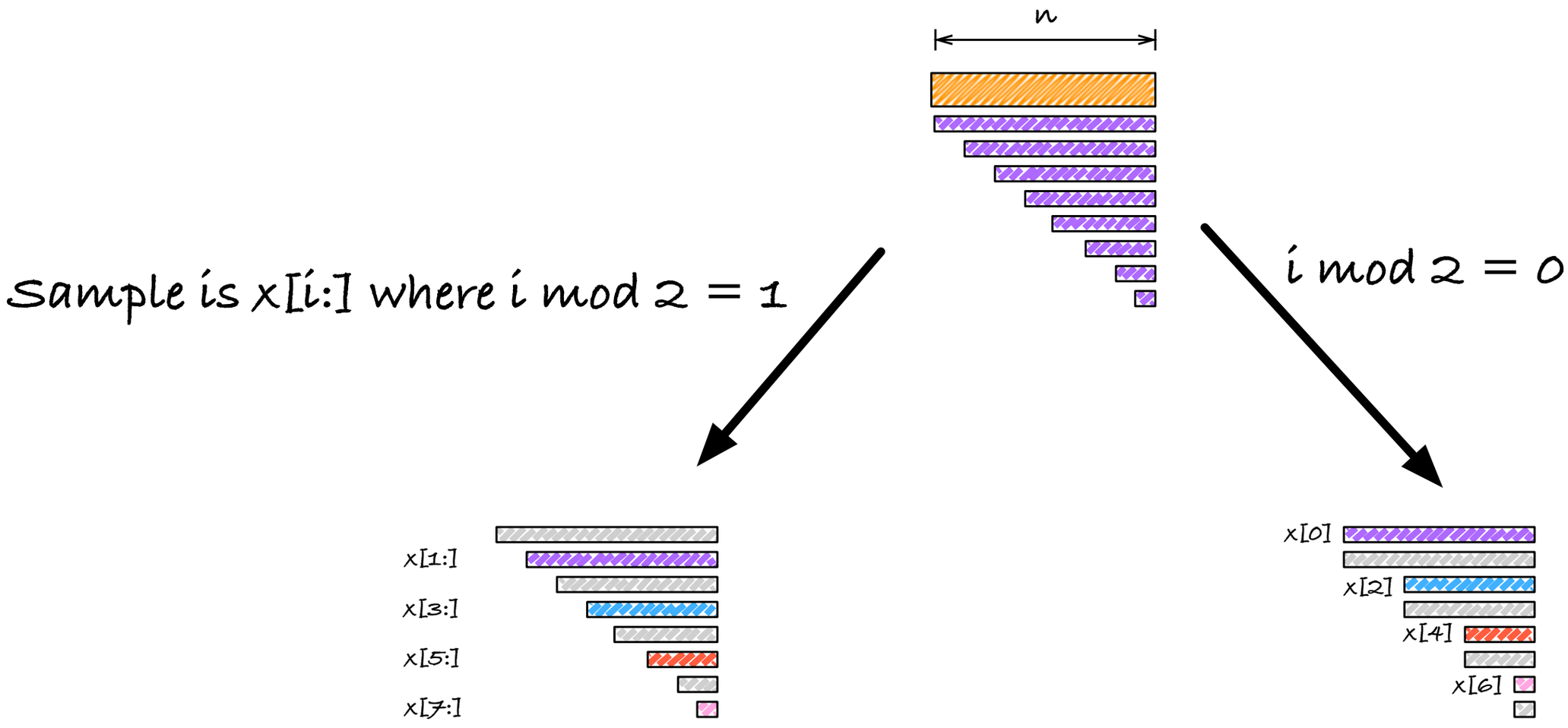
That would change the recurrence to
T(n) = T(f·n) + T((1-f)·n) + O(n)
which isn't in O(n).

First attempt

- The first attempt is the obvious choice for selecting samples

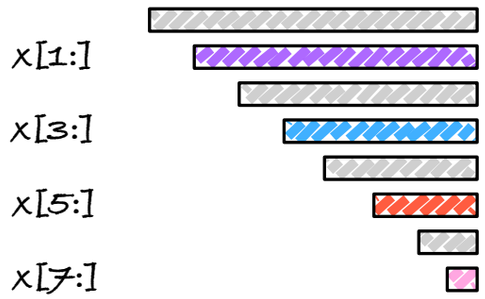
- It won’t work, but the way it doesn’t work will tell us why the skew algorithm doesn’t do the obvious (and why it does what it does)

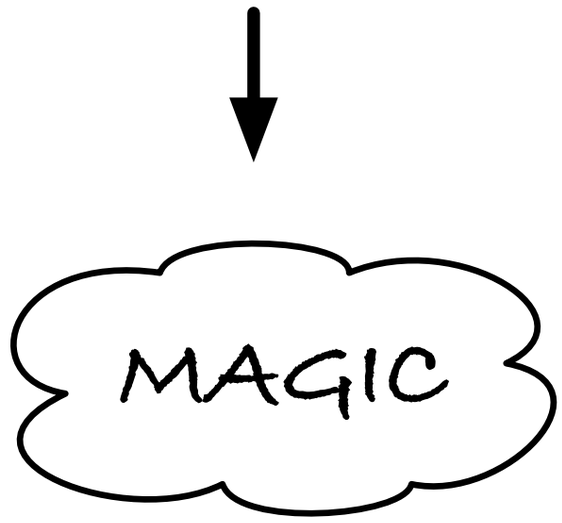
First attempt

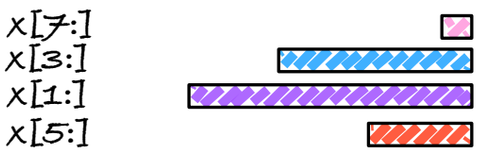


Easily done in O(n) since it is just a scan through the input string...

First attempt

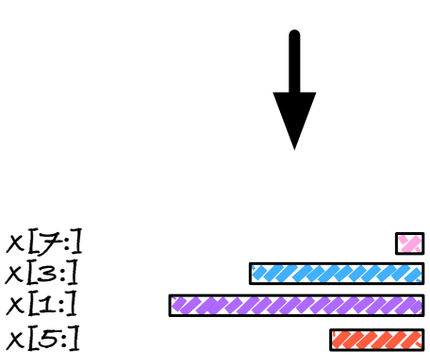






You can get this to work, but I won't show it, because we will run into a problem shortly. I will show you how to sort the sample in the fixed version instead.

First attempt

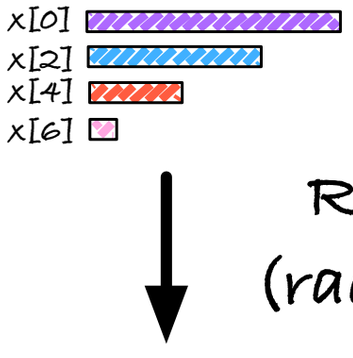


Rearrange:

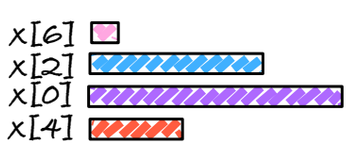
$sa_0[i] = sa_1[i] - 1$ for i index in sa_1.

Special case:

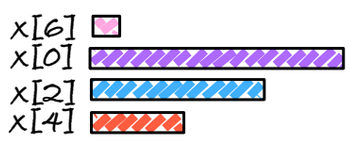
If last index in class mod 2 = 0, then this won't include it. Put it at the top, the bucket sort will fix it into the top of the right bucket.



Reorder according to sample
(radix sort up to first character)
O(n)



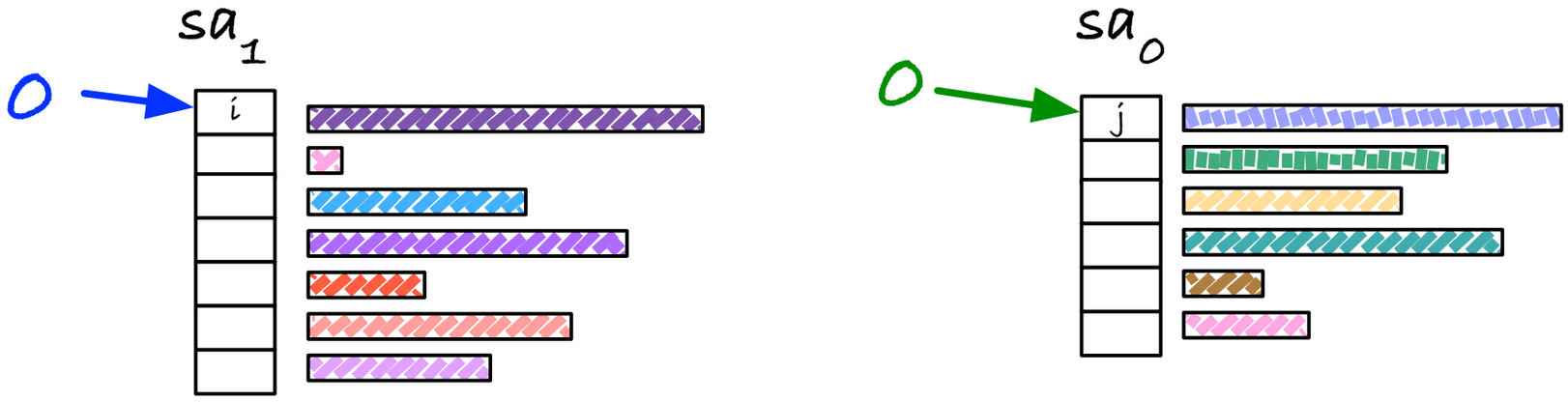
Bucket sort on first character
O(n)



First attempt

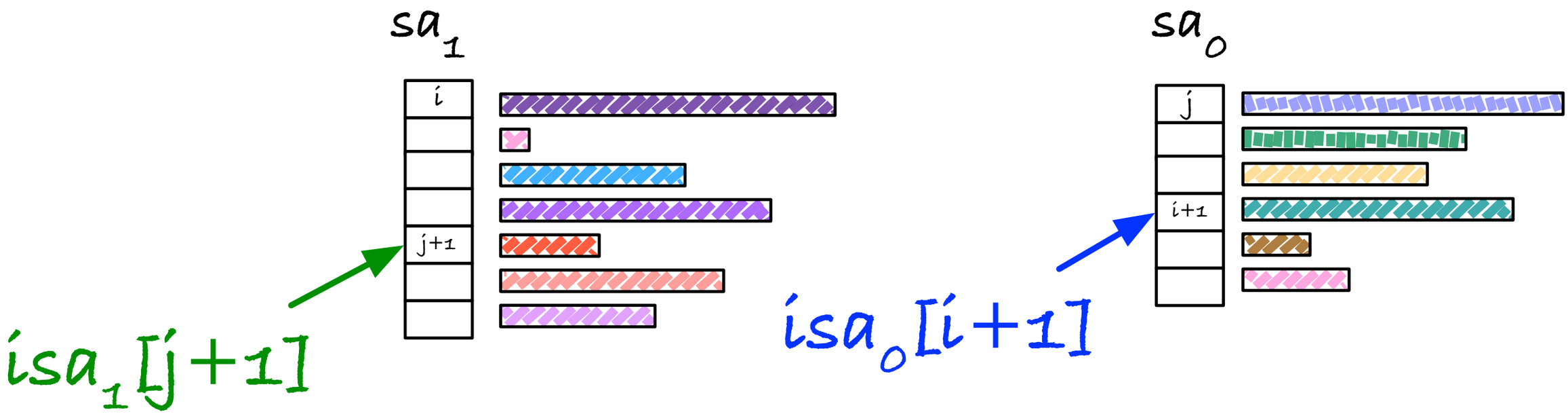
We can merge two arrays/lists in O(c·n) where c is the comparison time.

Can we compare these strings in O(1)?



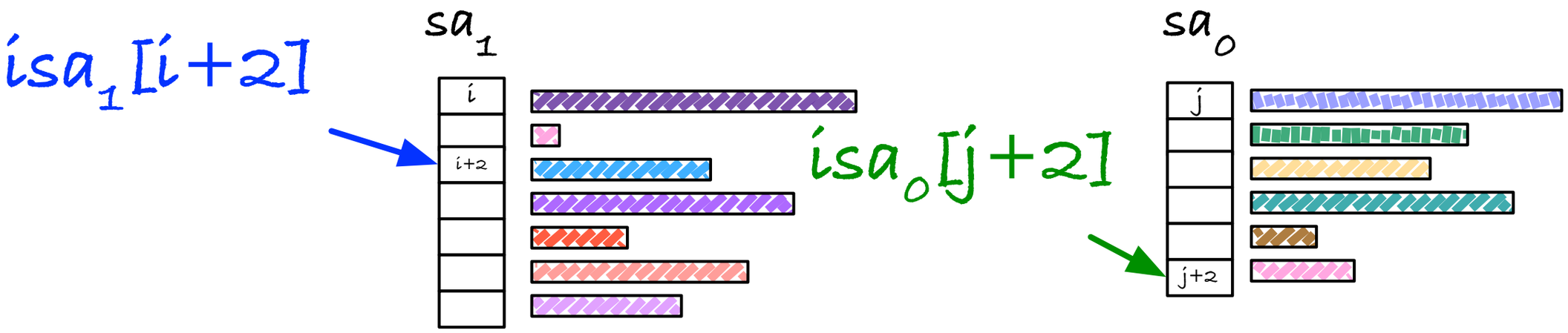

```
def less(x[i:],x[j:]):
    if x[i] < x[j]: return True
    if x[i] > x[j]: return False
    else: return less(x[i+1:],x[j+1:])
```

First attempt




```
def less(x[i:],x[j:]):
    if x[i] < x[j]: return True
    if x[i] > x[j]: return False
    else: return less(x[i+1:],x[j+1:])
```

First attempt

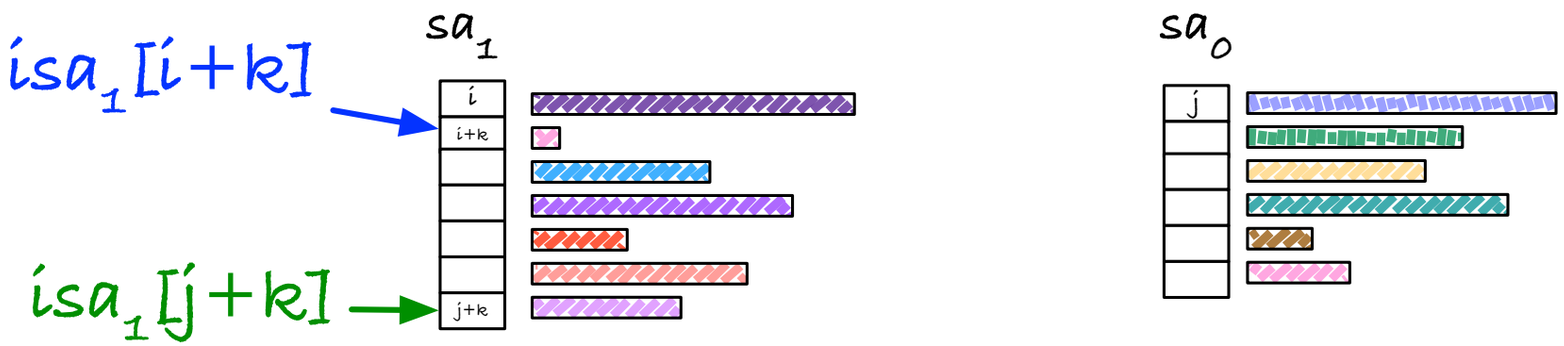


If we keep going, comparing identical letters, the comparison takes O(n) time, so the merge takes $O(n^2)$.

Not cool!!!

First attempt

Wouldn't it be nice if we could get both indices in the same array?
Then we could determine their order in O(1)

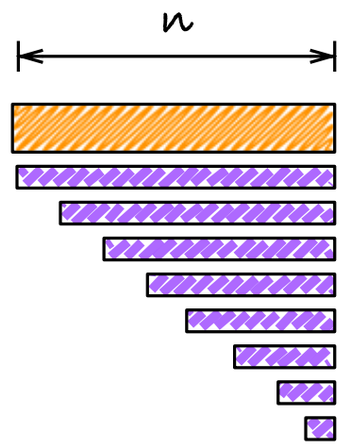


$$less(x[isa_1[i+k]:], x[isa_1[j+k]:]) = isa_1[i+k] < isa_1[j+k])$$

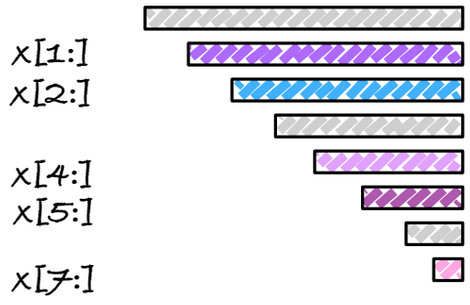
Well, we can't. The isa pointers will keep swapping array and never end up on the same side.

Not unless we make the sample and out-of-sample different sizes!

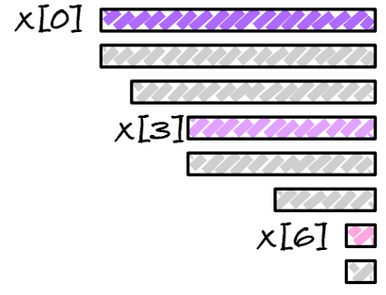
Skew algorithm



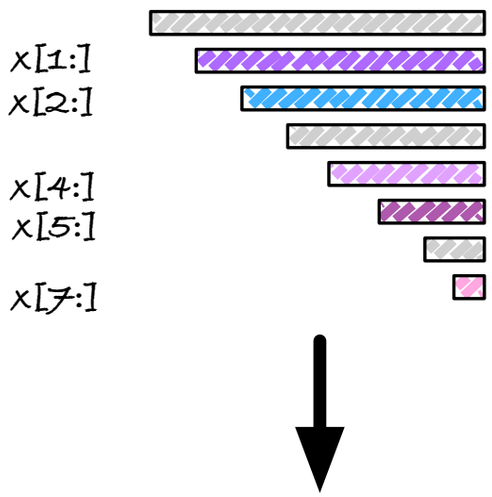
Sample is x[i:] where i mod 3 ≠ 0



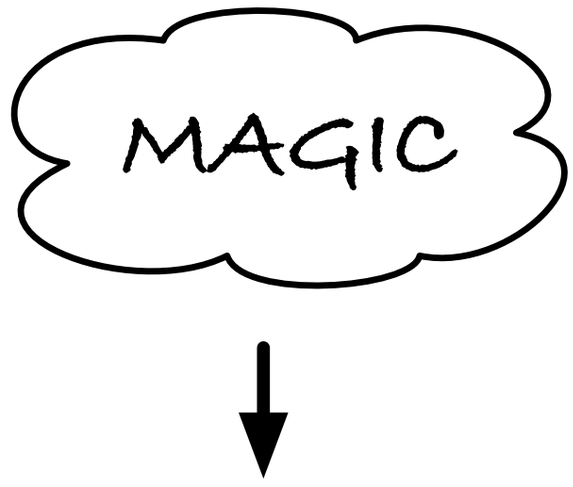
i mod 3 = 0

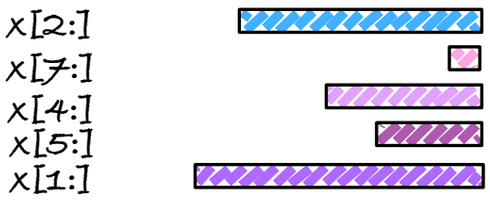


Skew algorithm

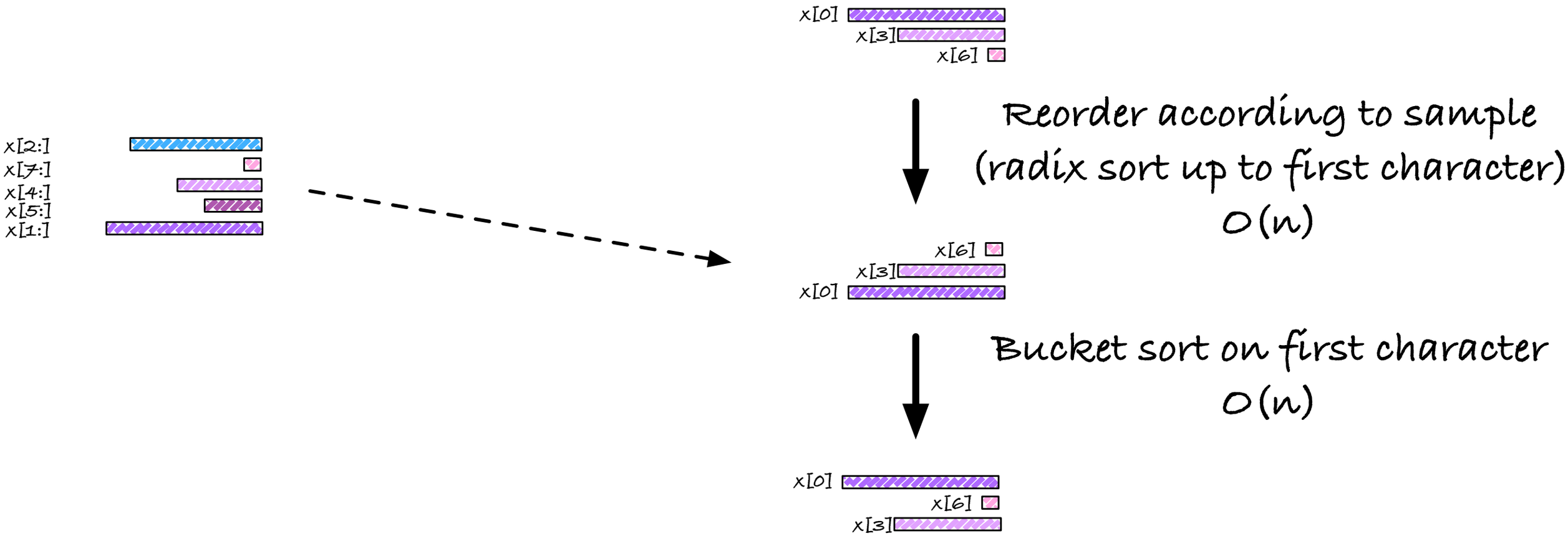


I will tell you how to do magic when we are sure that we can merge…





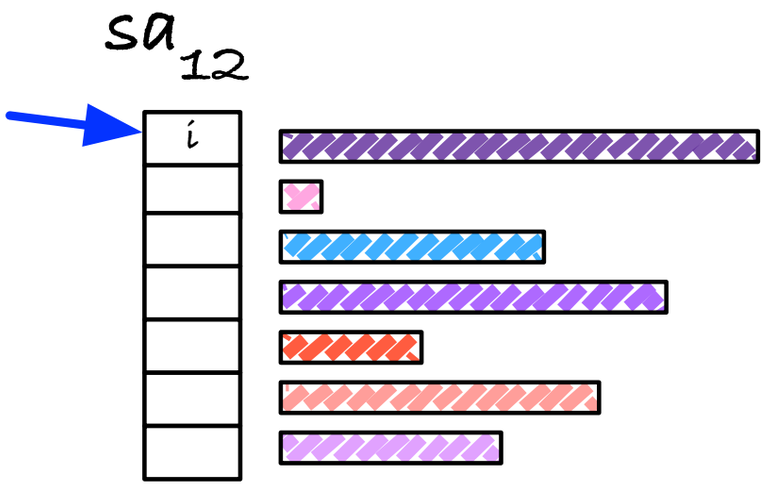
Skew algorithm

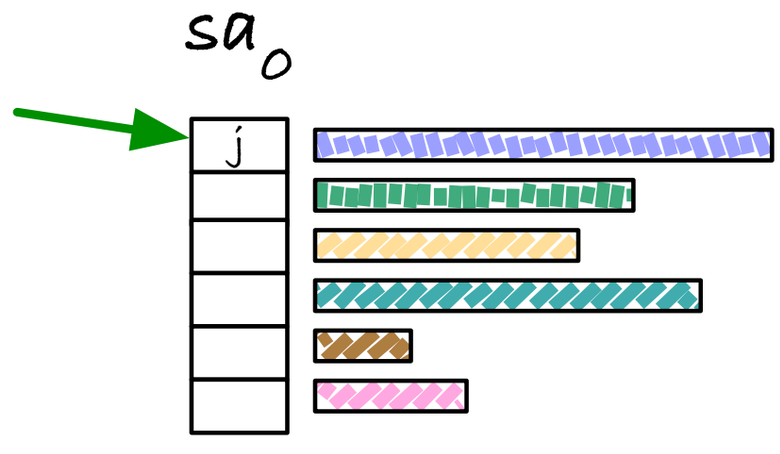


[i-1 for i in sa12 if i % 3 == 1]
(plus special case if n % 3 == 0)

Skew algorithm

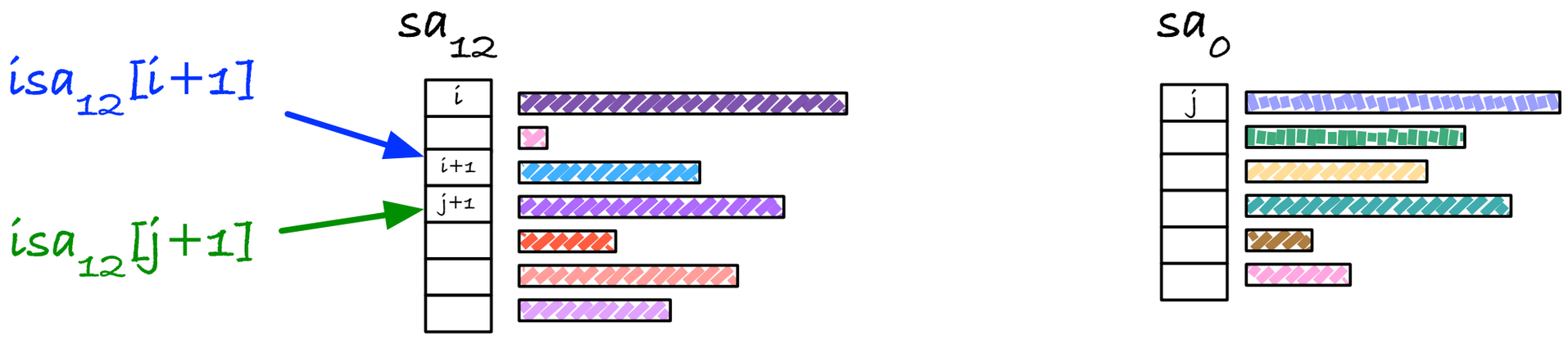
```
def less(x[i:],x[j:]):
    if x[i] < x[j]: return True
    if x[i] > x[j]: return False
    else: return less(x[i+1:],x[j+1:])
```





Skew algorithm

```
def less(x[i:],x[j:]):
    if x[i] < x[j]: return True
    if x[i] > x[j]: return False
    else: return less(x[i+1:],x[j+1:])
```

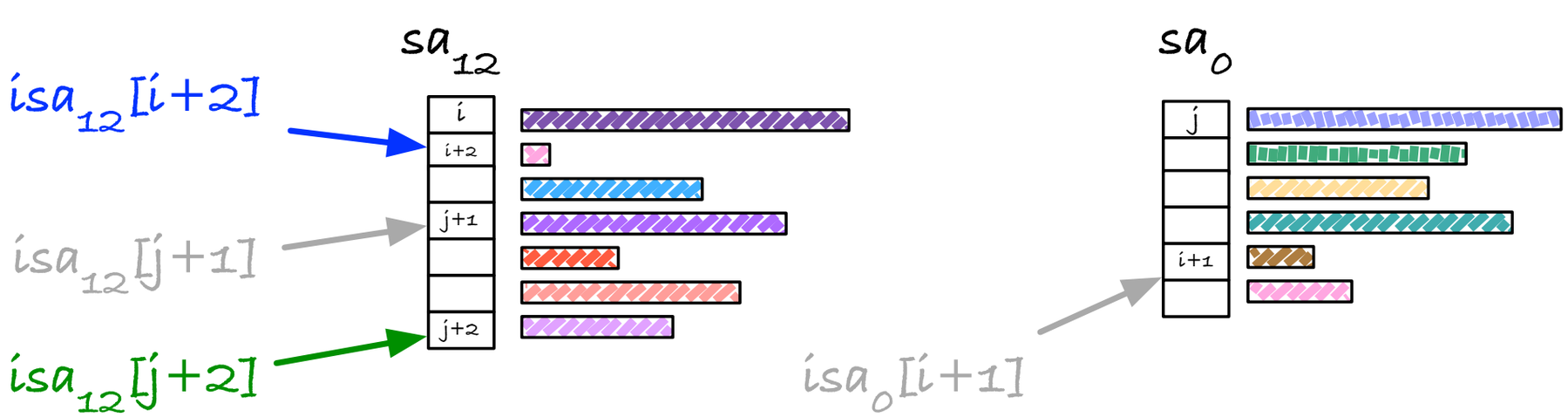


$(j+1) \bmod 3 = 1$

if $i \bmod 3 = 1$ then $(i+1) \bmod 3 = 2$

Skew algorithm

```
def less(x[i:],x[j:]):
    if x[i] < x[j]: return True
    if x[i] > x[j]: return False
    else: return less(x[i+1:],x[j+1:])
```



$(j+1) \bmod 3 = 1$ and $(j+2) \bmod 3 = 2$

if $i \bmod 3 = 2$ then $(i+1) \bmod 3 = 0$

and $(i+2) \bmod 3 = 1$

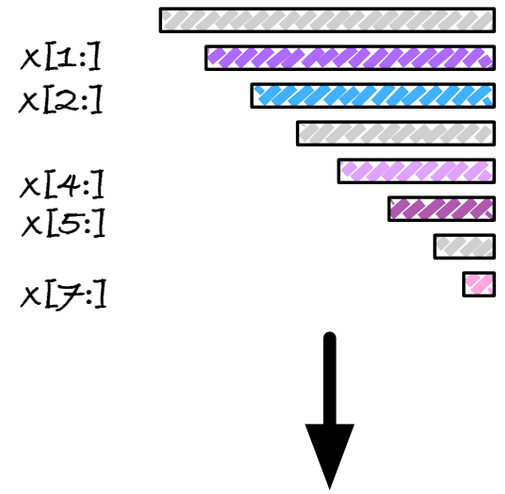
Skew algorithm

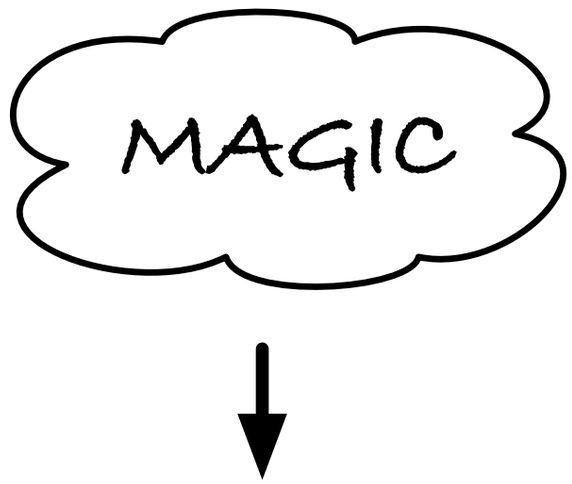
- In no more than two steps, you can determine less(x[i:],x[j:]), so comparison is in O(1)

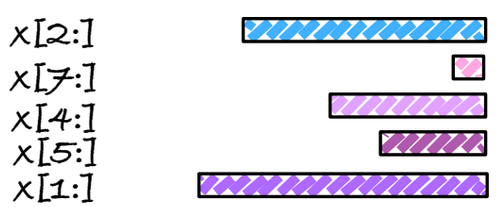
- Therefore, the merge is in O(n)

MAGIC!!!

- **What do we need for the magic step?**
  - We must create a new string, u, from the samples (the algorithm takes a string as input, so we can only recurse with a string)
  - We must do that in O(n) (or the recurrence doesn't work)
  - The suffix array for u must give us the order of the samples







Constructing u

This is going to look a bit weird, but we will get to the good place in the end…

```
u = x[1:] # x[2:]

if x = “mississippi” then
u = “ississippi#ssissippi”
```

`u = x[2:]#x[1:]` works just as well.

(you need to change some indices later in the algorithm, but which of `x[1:]` or `x[2:]` you put first doesn’t matter)

Constructing u

This is going to look a bit weird, but we will get to the good place in the end…

The # character is the sentinel (smaller than all other characters)

```
u = x[1:] # x[2:]
```

It is a different symbol than $ because it serves another purpose
(the “central” versus “terminal” sentinel)
but it *is* the same sentinel when you implement it.

In the skew algorithm you don’t need the terminal sentinel in the input
(although we use it implicitly when radix sorting)

So, we don’t need to add $ to the end of u here.

Constructing u

This is going to look a bit weird, but we will get to the good place in the end…

```
u = x[1:] # x[2:]
```

```
if x = “mississippi” then
u = “ississippi#ssissippi”
```

The length of u is not smaller than x!

$|u| = 2n-2$

Recursing on a *longer* string is folly!

Constructing u

This is going to look a bit weird, but we will get to the good place in the end…

```
u = x[1:] # x[2:]

if x = “mississippi” then
u = “ississippi#ssissippi”

Group in triplets:
u = [iss] [iss] [ipp] [i$$] [#] [ssi] [ssi] [ppi]
```

Constructing u

This is going to look a bit weird, but we will get to the good place in the end…

If we can create an alphabet of triplets, and create u as a string with this alphabet, all in O(n), then this u has length ⅔n and then we can recurse with T(n)=T(⅔n)+O(n).

u = x[1

if x = “miss

u = “ississi

Put a pin in this, but remember that we need to create a triplet alphabet.

Let’s first figure out what we can do with u, though, to see if it is worthwhile to do it.

Group in triplets:

u = [iss] [iss] [ipp] [i$$] [#] [ssi] [ssi] [ppi]

Constructing u

Consider the suffixes of u…

	u:	[iss] [iss] [ipp] [i$#] [#] [ssi] [ssi] [ppi]
u[0:]	x[1:]:	[iss] [iss] [ipp] [i$#] [#] [ssi] [ssi] [ppi]
u[1:]	x[4:]:	[iss] [ipp] [i$#] [#] [ssi] [ssi] [ppi]
u[2:]	x[7:]:	[ipp] [i$#] [#] [ssi] [ssi] [ppi]
u[3:]	x[10:]:	[i$#] [#] [ssi] [ssi] [ppi]
u[4:]		[#] [ssi] [ssi] [ppi]
u[5:]	x[2:]:	[ssi] [ssi] [ppi]
u[6:]	x[5:]:	[ssi] [ppi]
u[7:]	x[8:]:	[ppi]

Constructing u

Consider the suffixes of u…

	u:	[iss] [iss] [ipp] [i$#] [#] [ssi] [ssi] [ppi]
u[0:]	x[1:]:	[iss] [iss] [ipp] [i$#] [#] [ssi] [ssi] [ppi]
u[1:]	x[4:]:	[iss] [ipp] [i$#] [#] [ssi] [ssi] [ppi]
u[2:]	x[7:]:	[ipp] [i$#] [#] [ssi] [ssi] [ppi]
u[3:]	x[10:]:	[i$#] [#] [ssi] [ssi] [ppi]
u[4:]		[#] [ssi] [ssi] [ppi]
u[5:]	x[2:]:	[ssi] [ssi] [ppi]

When you compare u[i:] with u[j:] for some i,j, you never compare past the # sentinel (they always differ there, with # being the smallest letter).

The order of suffixes u[i:] does not depend on the strings past the sentinel. The x[i:] # x[2:] (i mod 3 = 1) strings are in effect just the x[i:] strings.

Constructing u

However! u is over a different alphabet. All the triplets are really different symbols (in practise, different integers)

[xyz] = ξ

u[2:] = ξω = x[2:] = [ipp] [i$$]

u[2:] x[7:]: [ipp] [i$$]

u[3:]

u[4:]

u[5:]

u[6:] x[5:]:

u[7:] x[8:]:

If the order of u suffixes should match the order of the corresponding x suffixes, then the alphabet must be order-preserving, i.e., [xyz] < [abc] iff “xyz” < “abc”.

Creating the alphabet

- We have to create an alphabet of triplets
  - I.e. each triplet must map to a unique integer
- The mapping must be order-preserving, [xyz] < [abc] iff “xyz” < “abc”
  - (This is usually the tricky part, since it is easy to build a table of triplets, but sorting them might not be)
- We only have O(n) time to build the alphabet in, or the running time breaks

Creating the alphabet

Take the i mod 3 ≠ 0 suffixes and radix sort them on the first three letters. Then you have the sorted triplets.

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

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

Runs in O(n + σ)

But we need O(n)… is O(n + σ) in O(n)?

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

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

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

Worries about the alphabet size?

- In this algorithm (and others) we create new alphabets. They are not constant sized! So, we need to consider the alphabet size as part of the time analysis.

- However, we can't have more than O(n) triplets if we get them from ⅔n different strings. So the alphabet size, σ, is in O(n)

- This is a far cry from O(1), but we only see σ in the bucket sorts in this algorithm, which runs in O(n + σ), which is then in O(n).

- We are okay in this algorithm, even if the alphabet size isn't constant.

Creating the alphabet

{“#”: 0}

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

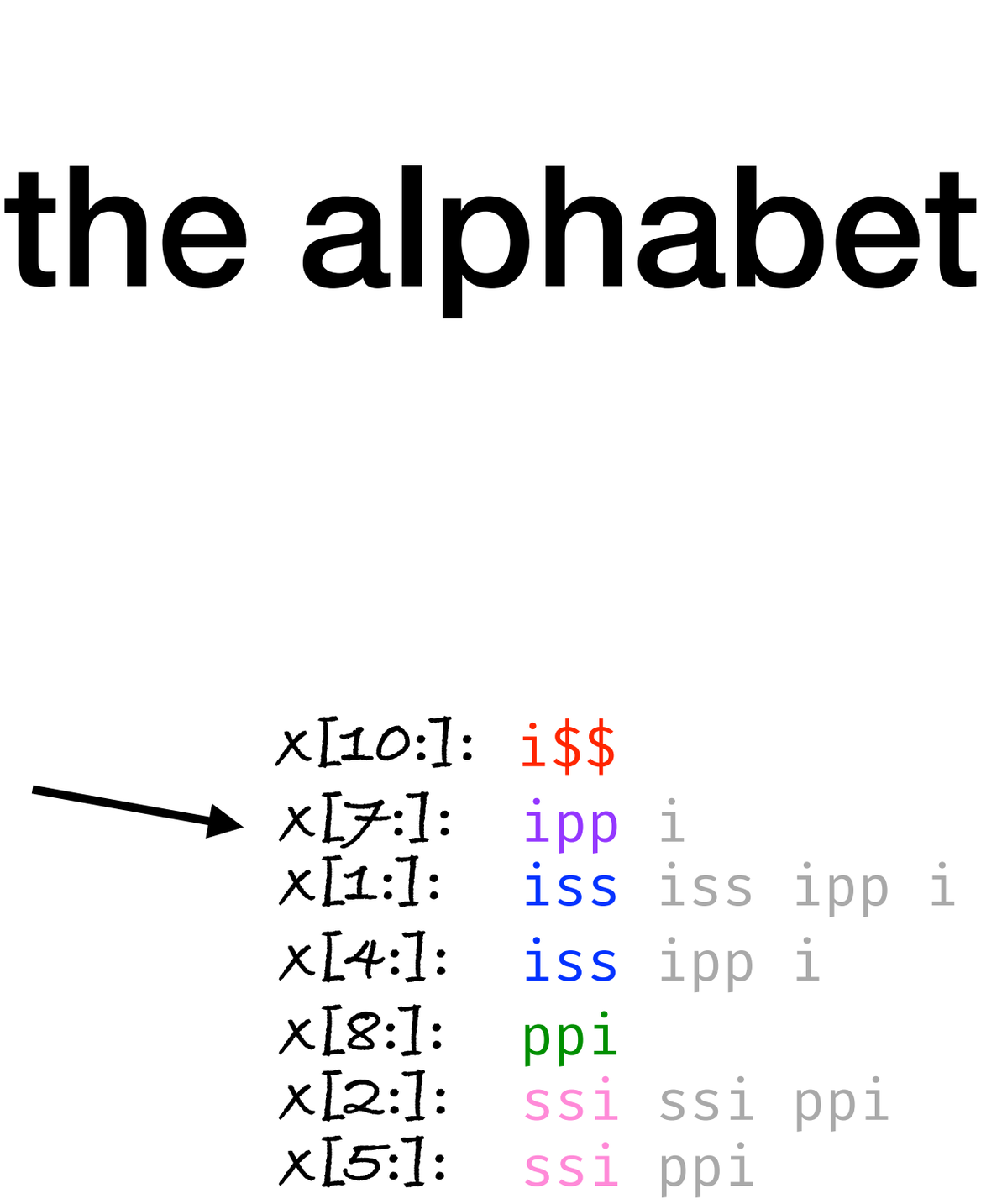
Creating the alphabet

```
{“#”: 0,
 “i$$”: 1,
}
```

```
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
```

Creating the alphabet

```
{“#”: 0,
 “i$$”: 1,
 “ipp”: 2
}
```



Creating the alphabet

```
{“#”: 0,
 “i$$”: 1,
 “ipp”: 2,
 “iss”: 3
}
```



```
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
```

Creating the alphabet

```
{“#”: 0,
 “i$$”: 1,
 “ipp”: 2,
 “iss”: 3
}
```



```
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
```

Creating the alphabet

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

```
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
```

Creating the alphabet

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

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

Creating the alphabet

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

```
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
```

Creating the alphabet

- Because we sort the triplets with radix sort, we can preserve the order in O(n).
- If you just created the table and sorted the keys, then you would pay for the sorting, typically O(n log n)
    - O(n log n) compassions with comparisons of triplets taking constant time

You have the alphabet, now what?

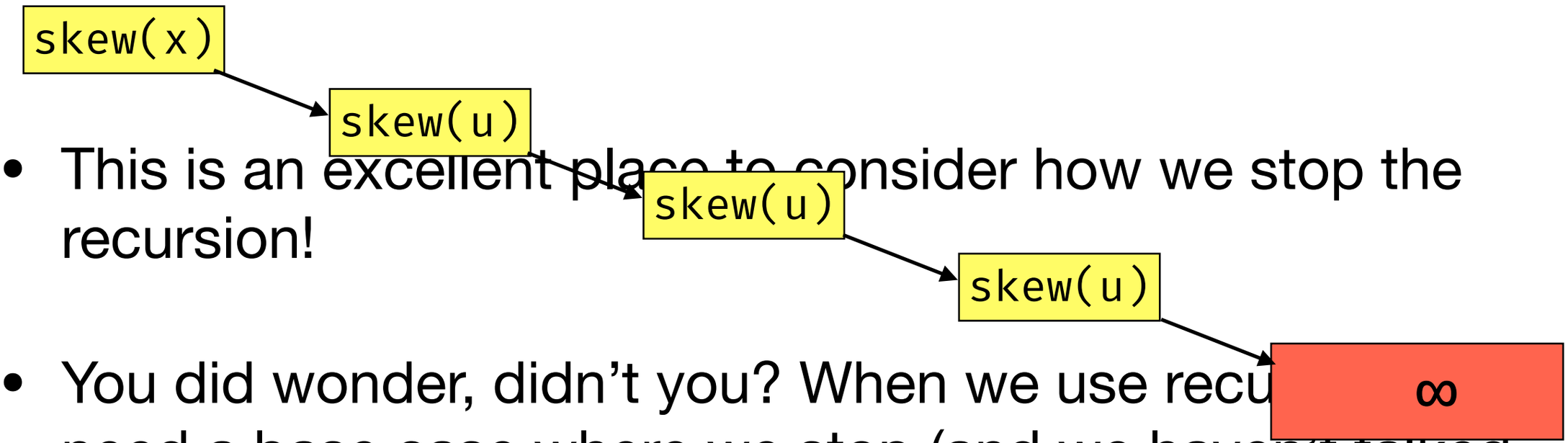
- This is an excellent place to consider how we stop the recursion!

You have the alphabet, now what?

- This is an excellent place to consider how we stop the recursion!

- You did wonder, didn’t you? When we use recursion, we need a base case where we stop (and we haven’t talked about that at all)

You have the alphabet, now what?



- This is an excellent place to consider how we stop the recursion!
- You did wonder, didn't you? When we use recu need a base case where we stop (and we haven't talked about that at all)

Base case

- The purpose of building u is to sort suffixes i mod 3 ≠ 0
- If all leading triplets are unique, they are already sorted
  - We can determine the order of strings from the prefix until they differ
  - If they differ in the first three letters, sorting on the first three letter gives us the correct order
  - Then we don’t need to recurse and can stop here!

Constructing u

- Easy way to build u is to put the triplets and their numbers in a table, then straight-forward scan through x[1+i:1+i+3] and then x[2+i:2+i+3] in jumps of 3

```
[*(alpha[triplet(x, i)] for i in range(1, len(x), 3)),
 SENTINEL,
 *(alpha[triplet(x, i)] for i in range(2, len(x), 3))]
```

Recurse, and then what?

- You know how to map between indices in x[i:] i mod 3 ≠ 0 and u[j:] because you created u in a specific way.

- Use that to map u[j:] back to x[i:]

sa_u

```
u[4:]    [#] [ssi] [ssi] [ppi]
u[3:]    [i$$] [#] [ssi] [ssi] [ppi]
u[2:]    [ipp] [i$$] [#] [ssi] [ssi] [ppi]
u[1:]    [iss] [ipp] [i$$] [#] [ssi] [ssi] [ppi]
u[0:]    [iss] [iss] [ipp] [i$$] [#] [ssi] [ssi] [ppi]
u[7:]    [ppi]
u[6:]    [ssi] [ppi]
u[5:]    [ssi] [ssi] [ppi]
```

```
          0     1     2     3       4 5     6     7
u = x[1...]x[4...]x[7...]x[10...] # x[2...]x[5...]x[8...]
```

```
x[10:]:  i
x[7:]:   ippi
x[4:]:   issippi
x[1:]:   ississippi
x[8:]:   ppi
x[5:]:   ssippi
x[2:]:   ssissippi
```

Recurse, and then what?

- You know how to map between indices in x[i:] i mod 3 ≠ 0 and u[j:] because you created u in a specific way.

- Use that to map u[j:] back to x[i:]

sa_u

u[4:] [#] [ssi] [ssi] [ppi]
u[3:] [i$$] [#] [ssi] [ssi] [ppi]
u[2:] [ipp] [i$$] [#] [ssi] [ssi] [ppi]
u[1:] [iss] [ipp] [i$$] [#] [ssi] [ssi] [ppi]
u[0:] [iss] [iss] [ipp] [i$$] [#] [ssi] [ssi] [ppi]
u[7:] [ppi]
u[6:] [ssi] [ppi]
u[5:] [ssi] [ssi] [ppi]

0 1 2 3 4 5 6 7
u = x[1...]x[4...]x[7...]x[10...] # x[2...]x[5...]x[8...]

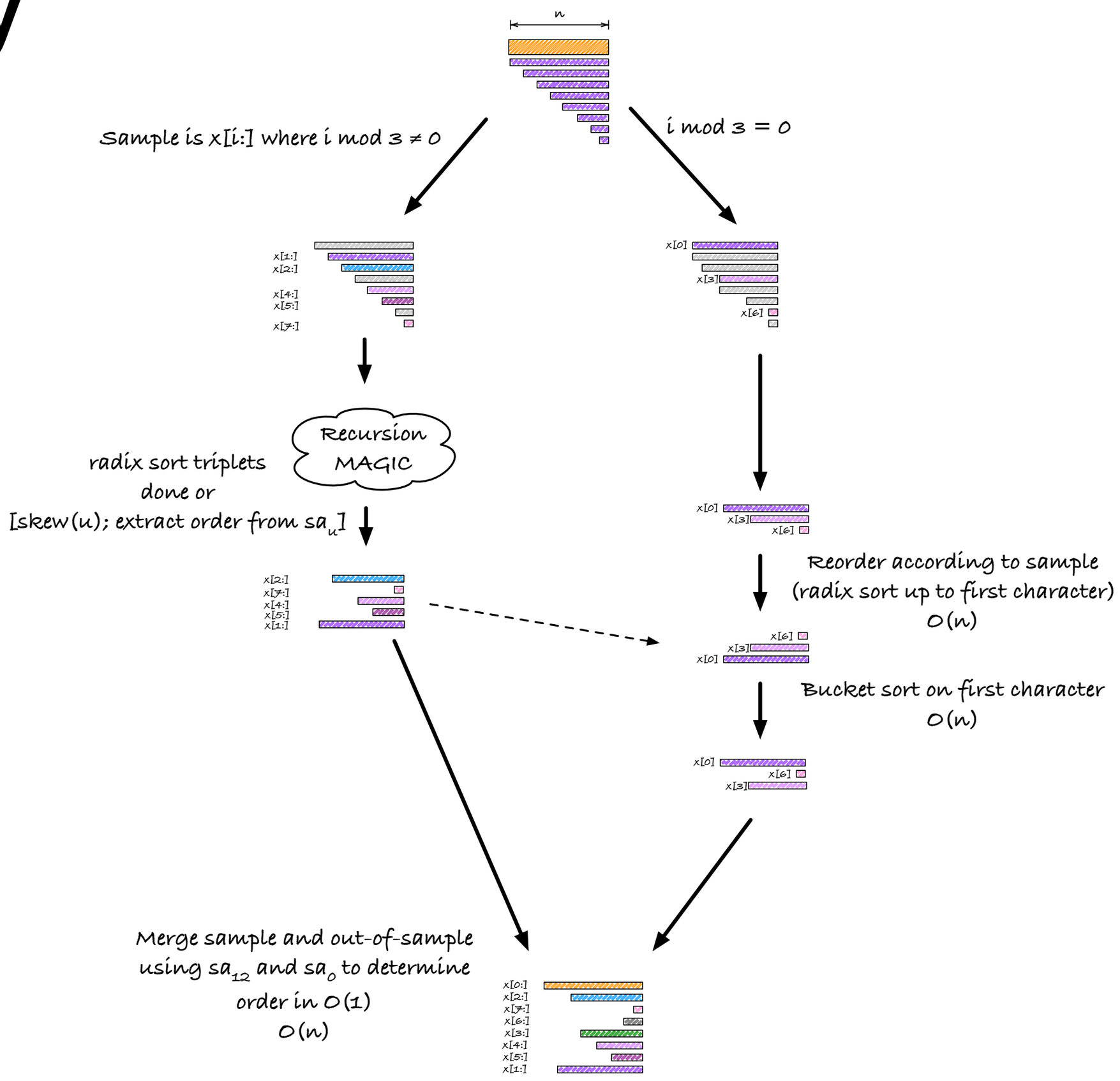
x[10:]: i
x[7:]: ippi

The exact mapping from u[j:] to x[i:] depends on how you construct u (u = x[1:] # x[2:] or x[2:] # x[1:]) or whether you include the central sentinel (you can avoid it if you start with a terminal $), but the mapping is always just a bit of arithmetic.

Summary of magic

- Radix sort the leading triplet of the suffixes i mod 3 ≠ 0
  - If the triplets are unique, we have sorted the suffixes
  - Otherwise,
    - construct u from triplets in `x[1:] # x[2:]`
    - call `skew(u)`
    - map the indices in sa_u back to the suffixes i mod 3 ≠ 0 (this depends on how we constructed u, but is straightforward)

Summary of skew



Final words on sentinels

- The central sentinel # when we construct u separates the i mod 3 = 1 from the i mod 3 = 2 strings. It must be there, or the algorithm fails.

- However, if the input, x, has a terminal sentinel, then the triplets in the alphabet we generate from x will contain a symbol that functions as #.

  - If we create u = x[1:]x[2:] or u = x[2:]x[1:] (just concatenate the two classes without the # sentinel), the center triplet will be unique and function as #.

  - This is a tedious proof, essentially just case analysis, and I won’t show it.

That's all Folks!