Space/Time Trade-Offs

Murat Osmanoglu

Space/Time Trade-Offs

- solve a problem in less time by using more storage space, or in little space by spending a long time

Space/Time Trade-Offs

- solve a problem in less time by using more storage space, or in little space by spending a long time

- you may have a large amount of space but not infinite, or you can wait a little more but not forever

Space/Time Trade-Offs

- solve a problem in less time by using more storage space, or in little space by spending a long time

- you may have a large amount of space but not infinite, or you can wait a little more but not forever

Space-for-Time Tradeoffs

- we cover two varieties of space-for-time algorithms:

Space/Time Trade-Offs

- solve a problem in less time by using more storage space, or in little space by spending a long time

- you may have a large amount of space but not infinite, or you can wait a little more but not forever

Space-for-Time Tradeoffs

- we cover two varieties of space-for-time algorithms:
  - input enhancement; preprocess the input to store the additional information to be used later to improve the time efficiency

Space/Time Trade-Offs

- solve a problem in less time by using more storage space, or in little space by spending a long time

- you may have a large amount of space but not infinite, or you can wait a little more but not forever

Space-for-Time Tradeoffs

- we cover two varieties of space-for-time algorithms:
  - input enhancement; preprocess the input to store the additional information to be used later to improve the time efficiency
  - pre-structuring; preprocess the input to make accessing its element easier

Input Enhancement

Sorting Problem

	Best Case	Average Case	Worst Case	Space
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	1
Bubble Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	1
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$	1
Merge Sort	$O(n\log n)$	$O(n\log n)$	$O(n\log n)$	$O(n)$
Quicksort	$O(n\log n)$	$O(n\log n)$	$O(n^2)$	$O(\log n)$

Input Enhancement

Sorting Problem

	Best Case	Average Case	Worst Case	Space
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	1
Bubble Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	1
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$	1
Merge Sort	$O(n\log n)$	$O(n\log n)$	$O(n\log n)$	$O(n)$
Quicksort	$O(n\log n)$	$O(n\log n)$	$O(n^2)$	$O(\log n)$

- What is common among all of these algorithms?

Input Enhancement

Sorting Problem

	Best Case	Average Case	Worst Case	Space
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	1
Bubble Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	1
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$	1
Merge Sort	$O(nlogn)$	$O(nlogn)$	$O(nlogn)$	$O(n)$
Quicksort	$O(nlogn)$	$O(nlogn)$	$O(n^2)$	$O(logn)$

- What is common among all of these algorithms?

  the sorted order they determine is only based on comparisons between the input elements

Input Enhancement

Sorting Problem

	Best Case	Average Case	Worst Case	Space
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	1
Bubble Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	1
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$	1
Merge Sort	$O(n\log n)$	$O(n\log n)$	$O(n\log n)$	$O(n)$
Quicksort	$O(n\log n)$	$O(n\log n)$	$O(n^2)$	$O(\log n)$

- What is common among all of these algorithms?

  the sorted order they determine is only based on comparisons between the input elements

- Can we establish a lower bound on the number of comparisons for the worst case of comparison-based sorting algorithms ?

Input Enhancement

Sorting Problem

- Let's use a decision tree to get the intuition

↓

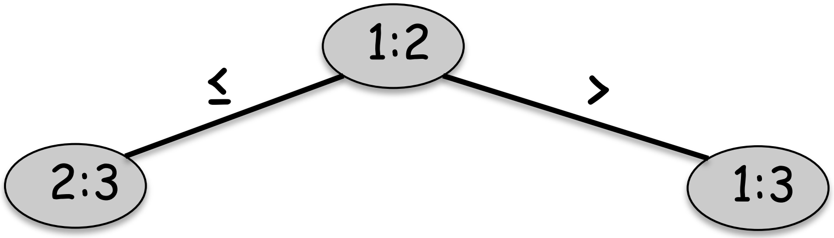
a full binary tree that represents the comparisons between elements that are performed by the algorithm

Input Enhancement

Sorting Problem

- Let's use a decision tree to get the intuition

a full binary tree that represents the comparisons between elements that are performed by the algorithm

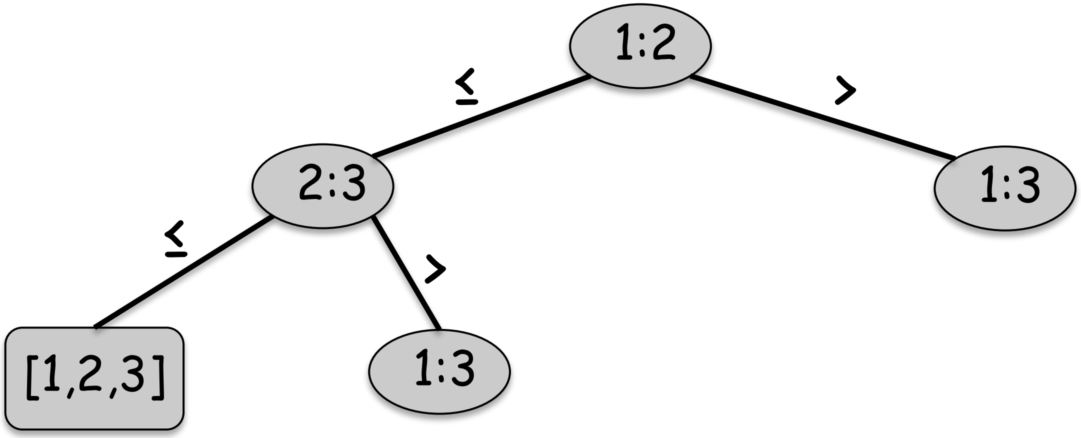


Input Enhancement

Sorting Problem

- Let's use a decision tree to get the intuition

a full binary tree that represents the comparisons between elements that are performed by the algorithm

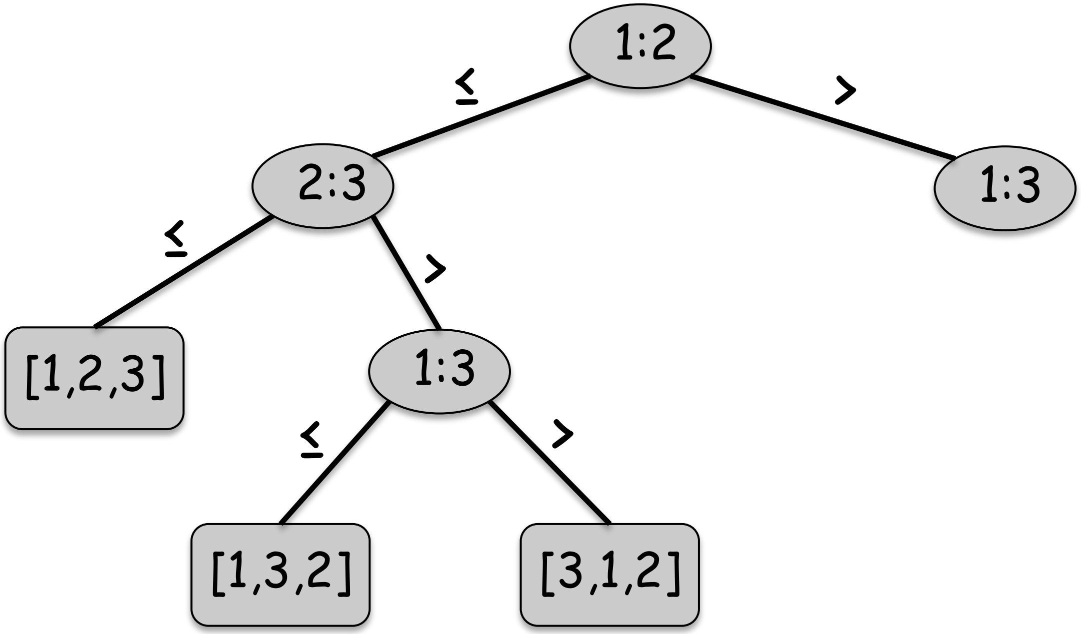


Input Enhancement

Sorting Problem

- Let's use a decision tree to get the intuition

a full binary tree that represents the comparisons between elements that are performed by the algorithm



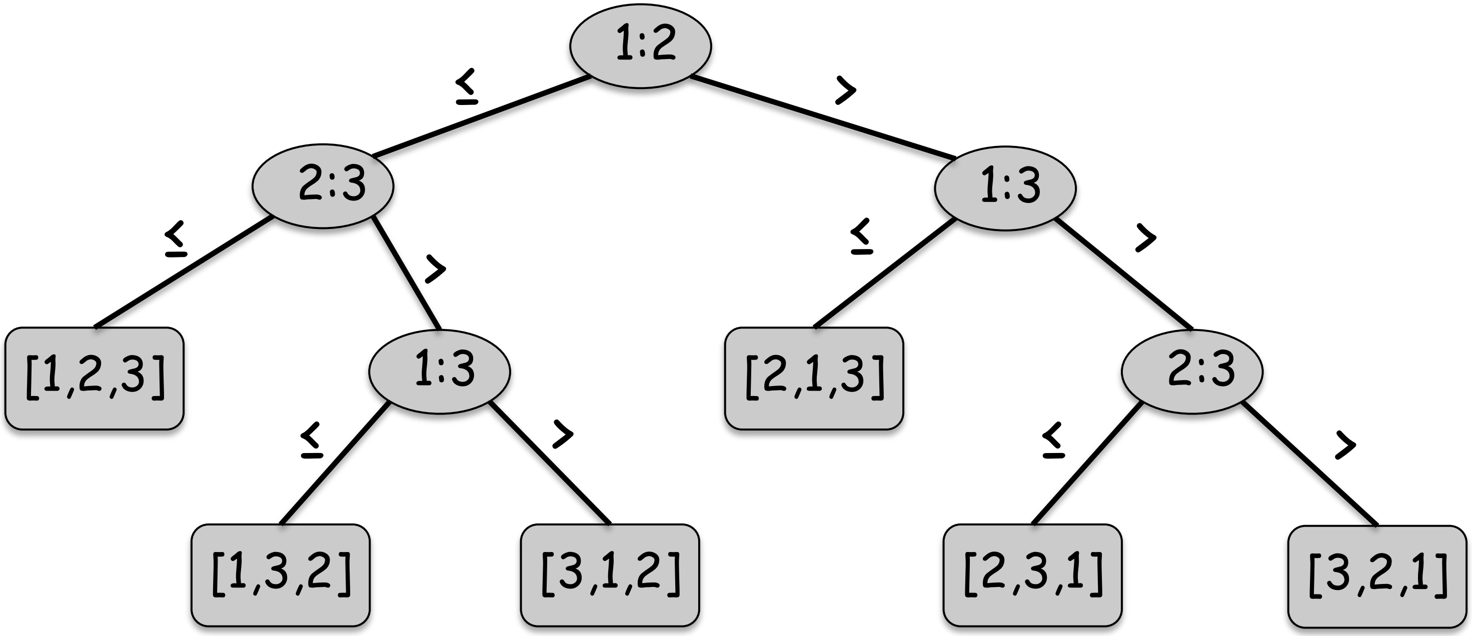
Input Enhancement

Sorting Problem

- Let's use a decision tree to get the intuition

↓

a full binary tree that represents the comparisons between elements that are performed by the algorithm

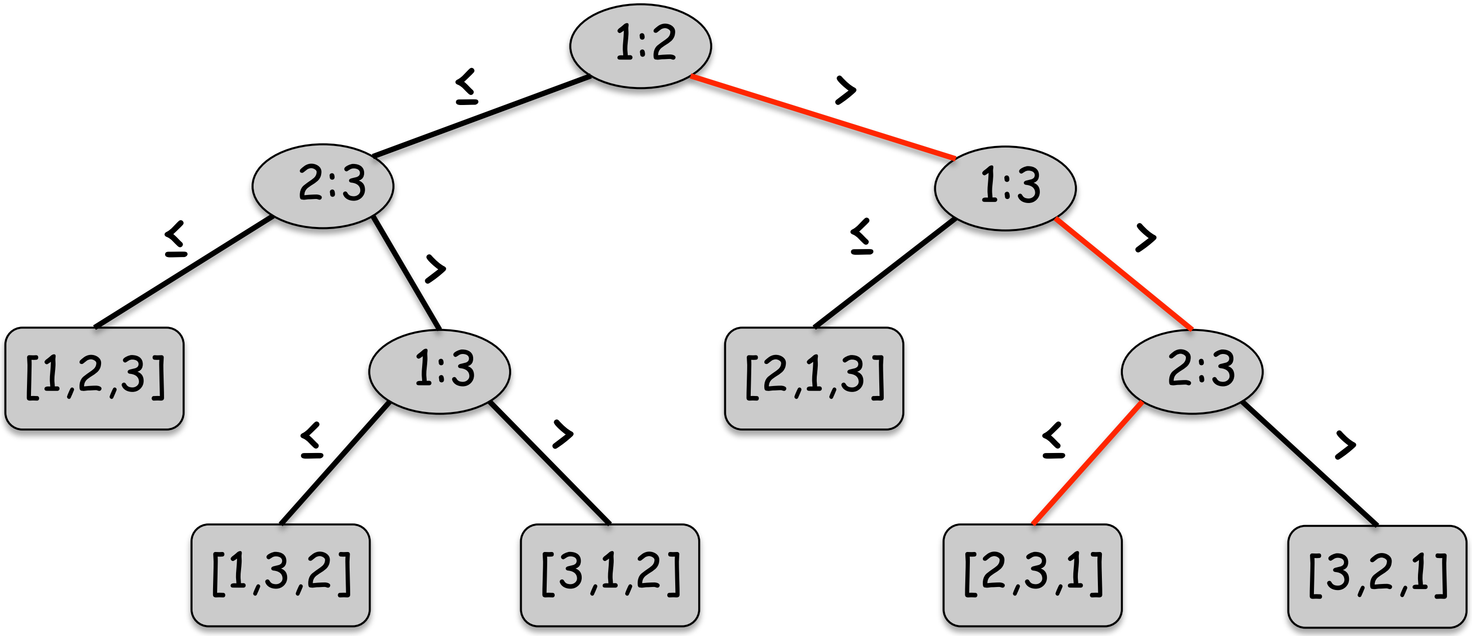


Input Enhancement

Sorting Problem

- Let's use a decision tree to get the intuition

a full binary tree that represents the comparisons between elements that are performed by the algorithm



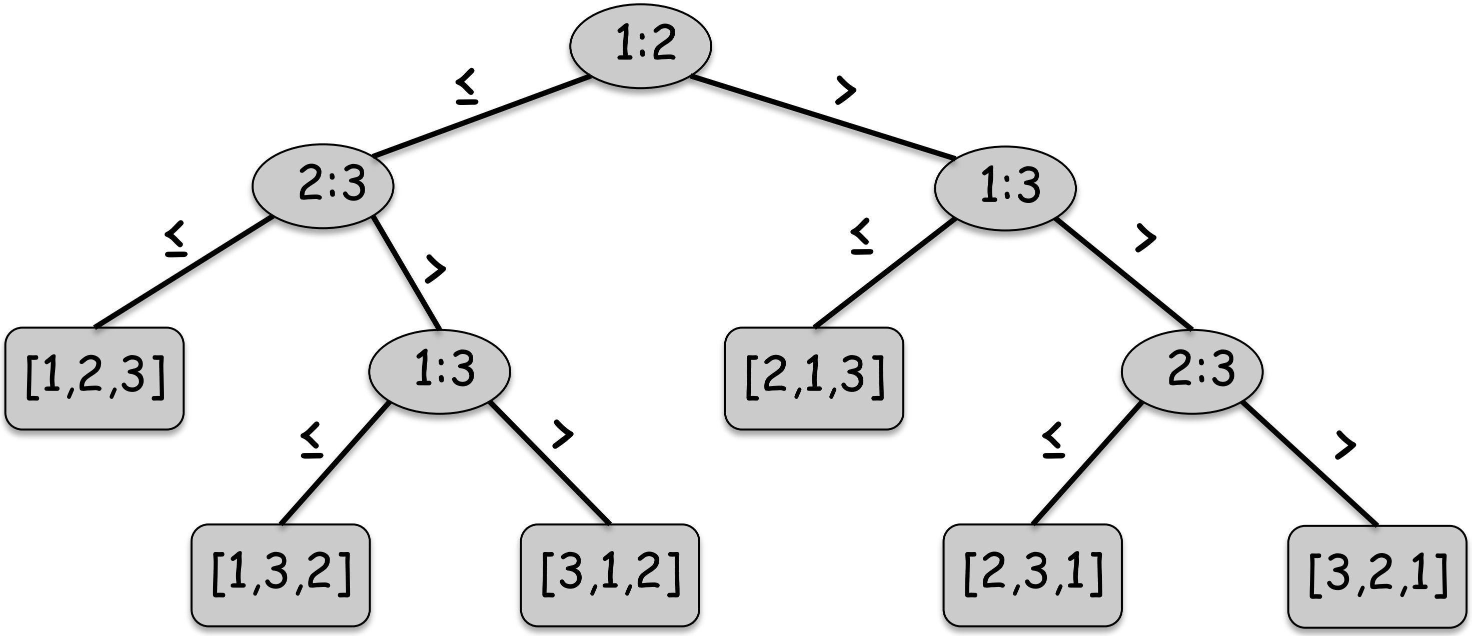
- for the input (4, 9, 6), the red path indicates the decision made

Input Enhancement

Sorting Problem

- Let's use a decision tree to get the intuition

- each permutation appears as one of the leaves in the tree



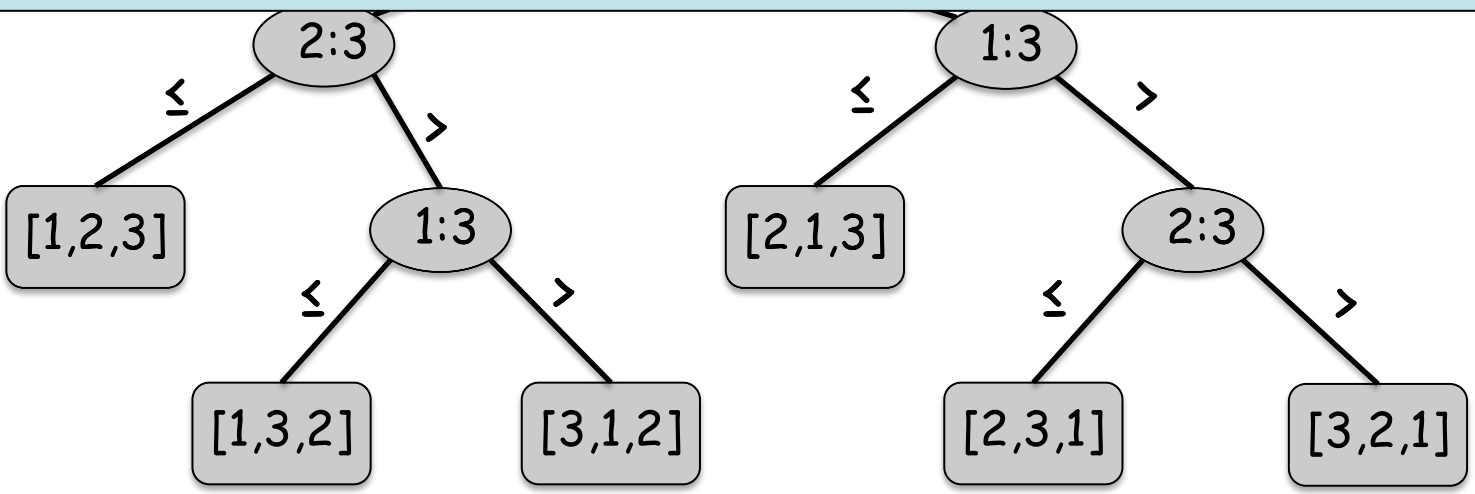
- for the input (4, 9, 6), the red path indicates the decision made

Input Enhancement

Sorting Problem

- Let's use a decision tree to get the intuition

- each permutation appears as one of the leaves in the tree
- the depth of a particular node will be the number of comparisons the comparison-based sorting algorithm performs



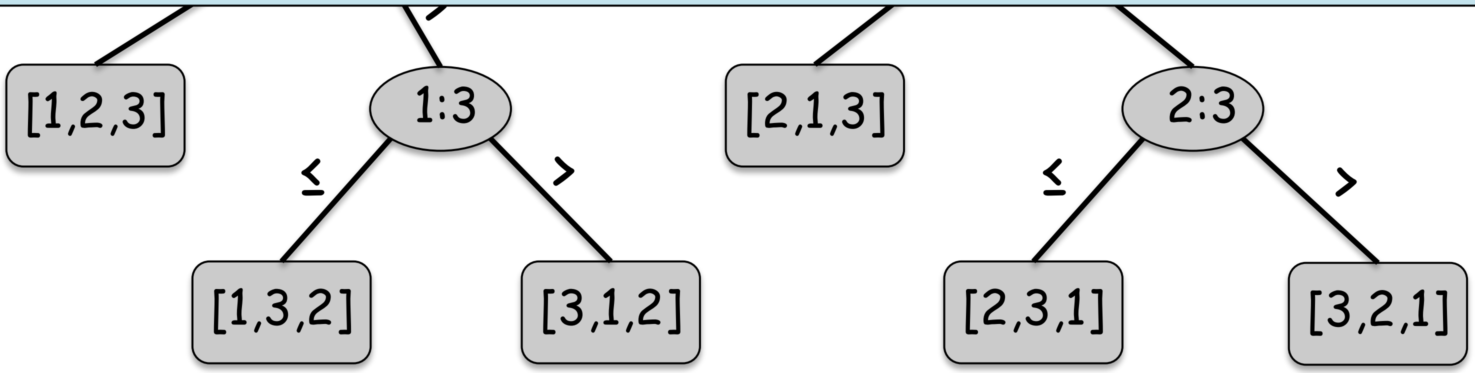
- for the input (4, 9, 6), the red path indicates the decision made

Input Enhancement

Sorting Problem

- Let’s use a decision tree to get the intuition

- each permutation appears as one of the leaves in the tree
- the depth of a particular node will be the number of comparisons the comparison-based sorting algorithm performs
- the height of the tree then will equal to the worst-case number of comparisons

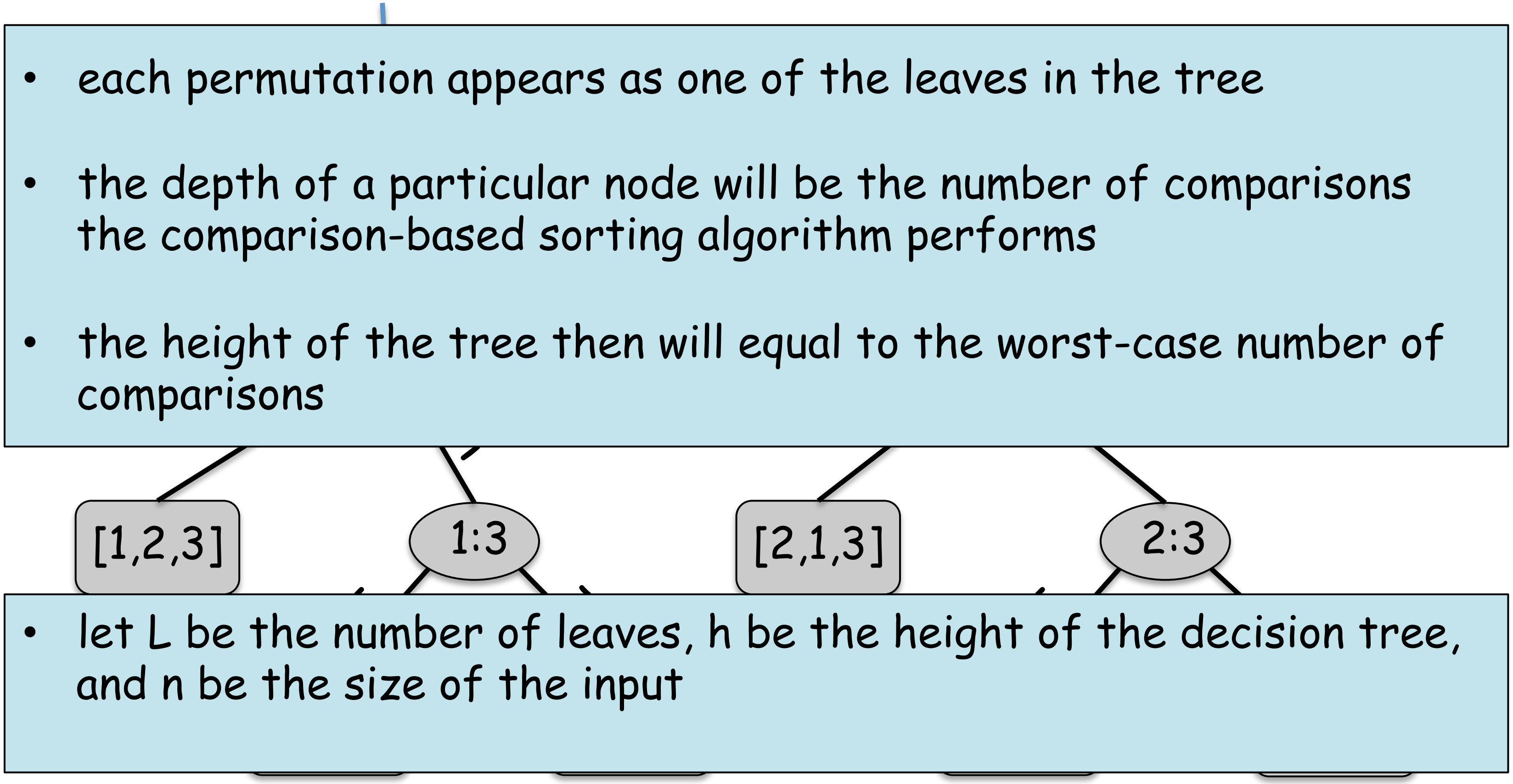


- for the input (4, 9, 6), the red path indicates the decision made

Input Enhancement

Sorting Problem

- Let’s use a decision tree to get the intuition



- each permutation appears as one of the leaves in the tree
- the depth of a particular node will be the number of comparisons the comparison-based sorting algorithm performs
- the height of the tree then will equal to the worst-case number of comparisons

- let L be the number of leaves, h be the height of the decision tree, and n be the size of the input

- for the input (4, 9, 6), the red path indicates the decision made

Input Enhancement

Sorting Problem

- Let’s use a decision tree to get the intuition

- each permutation appears as one of the leaves in the tree
- the depth of a particular node will be the number of comparisons the comparison-based sorting algorithm performs
- the height of the tree then will equal to the worst-case number of comparisons

- let L be the number of leaves, h be the height of the decision tree, and n be the size of the input

$L = n!, \; L \leq 2^h$

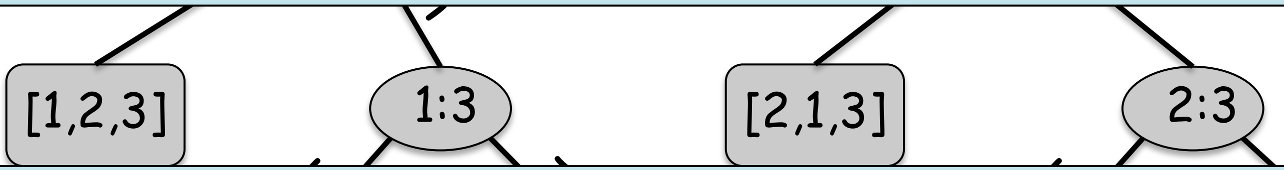
- for the input (4, 9, 6), the red path indicates the decision made

Input Enhancement

Sorting Problem

- Let's use a decision tree to get the intuition

- each permutation appears as one of the leaves in the tree
- the depth of a particular node will be the number of comparisons the comparison-based sorting algorithm performs
- the height of the tree then will equal to the worst-case number of comparisons



- let L be the number of leaves, h be the height of the decision tree, and n be the size of the input

  $L = n!$, $L \leq 2^h$, thus $h \geq \log(n!)$

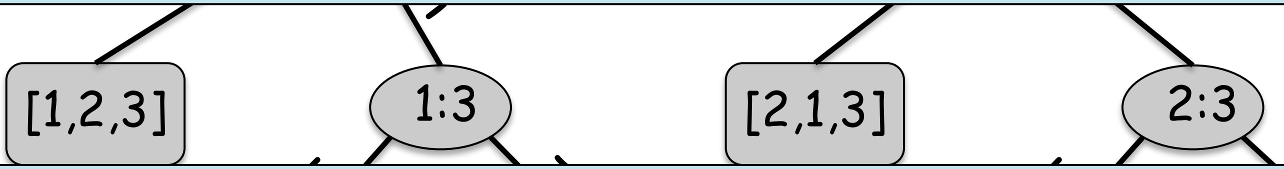
- for the input (4, 9, 6), the red path indicates the decision made

Input Enhancement

Sorting Problem

- Let’s use a decision tree to get the intuition

- each permutation appears as one of the leaves in the tree
- the depth of a particular node will be the number of comparisons the comparison-based sorting algorithm performs
- the height of the tree then will equal to the worst-case number of comparisons



- let L be the number of leaves, h be the height of the decision tree, and n be the size of the input

$L = n!$, $L \le 2^h$, thus $h \ge \log(n!)$, $h = \Omega(n\log n)$

- for the input (4, 9, 6), the red path indicates the decision made

Input Enhancement

Sorting by Counting

- given an array of n orderable items $[a_1, a_2, \ldots, a_n]$, reorder the items as $[a_1', a_2', \ldots, a_n']$ such that $a_1' \le a_2' \le \ldots \le a_n'$

Input Enhancement

Sorting by Counting

- given an array of n orderable items $[a_1, a_2, \ldots, a_n]$, reorder the items as $[a_1', a_2', \ldots, a_n']$ such that $a_1' \le a_2' \le \ldots \le a_n'$
- for each element of the array, count the total number of elements smaller than this number, and record the results in a table

Input Enhancement

Sorting by Counting

- given an array of n orderable items $[a_1, a_2, \ldots, a_n]$, reorder the items as $[a_1', a_2', \ldots, a_n']$ such that $a_1' \leq a_2' \leq \ldots \leq a_n'$
- for each element of the array, count the total number of elements smaller than this number, and record the results in a table
- this count determines the position of the element in the final sorted array

  if the count is 5 for some element, then the element should be placed in the sixth position in the sorted array

Input Enhancement

Sorting by Counting

- given an array of n orderable items $[a_1, a_2, \ldots, a_n]$, reorder the items as $[a_1', a_2', \ldots, a_n']$ such that $a_1' \le a_2' \le \ldots \le a_n'$
- for each element of the array, count the total number of elements smaller than this number, and record the results in a table
- this count determines the position of the element in the final sorted array

  if the count is 5 for some element, then the element should be placed in the sixth position in the sorted array

CountingSort-1(X[1,n])

```
let C[1, n] and B[1, n] be new arrays
for i = 1 to n
    C[i] ← 0
for i = 1 to n - 1
    for j = i+1 to n
        if X[i] < X[j]
            C[j] ← C[j] + 1
        else
            C[i] ← C[i] + 1
for i = 1 to n
    B[C[i]+1] ← X[i]
return B
```

Input Enhancement

Sorting by Counting

- given an array of n orderable items $[a_1, a_2, \ldots, a_n]$, reorder the items as $[a_1', a_2', \ldots, a_n']$ such that $a_1' \le a_2' \le \ldots \le a_n'$

- for each element of the array, count the total number of elements smaller than this number, and record the results in a table

- this count determines the position of the element in the final sorted array

  if the count is 5 for some element, then the element should be placed in the sixth position in the sorted array

CountingSort-1(X[1,n])

```
let C[1, n] and B[1, n] be new arrays
for i = 1 to n
    C[i] ← 0
for i = 1 to n - 1
    for j = i+1 to n
        if X[i] < X[j]
            C[j] ← C[j] + 1
        else
            C[i] ← C[i] + 1
for i = 1 to n
    B[C[i]+1] ← X[i]
return B
```

X	12	8	9	17	5	10

Input Enhancement

Sorting by Counting

- given an array of n orderable items $[a_1, a_2, \dots, a_n]$, reorder the items as $[a_1', a_2', \dots, a_n']$ such that $a_1' \leq a_2' \leq \dots \leq a_n'$
- for each element of the array, count the total number of elements smaller than this number, and record the results in a table
- this count determines the position of the element in the final sorted array

  if the count is 5 for some element, then the element should be placed in the sixth position in the sorted array

CountingSort-1(X[1,n])

```
let C[1, n] and B[1, n] be new arrays
for i = 1 to n
    C[i] ← 0
for i = 1 to n - 1
    for j = i+1 to n
        if X[i] < X[j]
            C[j] ← C[j] + 1
        else
            C[i] ← C[i] + 1
for i = 1 to n
    B[C[i]+1] ← X[i]
return B
```

X	12	8	9	17	5	10
C						
B						

Input Enhancement

Sorting by Counting

- given an array of n orderable items [a_1, a_2,…, a_n], reorder the items as [a_1', a_2',…, a_n'] such that $a_1' \leq a_2' \leq \ldots \leq a_n'$
- for each element of the array, count the total number of elements smaller than this number, and record the results in a table
- this count determines the position of the element in the final sorted array

  if the count is 5 for some element, then the element should be placed in the sixth position in the sorted array

CountingSort-1(X[1,n])

```
let C[1, n] and B[1, n] be new arrays
for i = 1 to n
    C[i] ← 0
for i = 1 to n - 1
    for j = i+1 to n
        if X[i] < X[j]
            C[j] ← C[j] + 1
        else
            C[i] ← C[i] + 1
for i = 1 to n
    B[C[i]+1] ← X[i]
return B
```

X	12	8	9	17	5	10
C	0	0	0	0	0	0
B						

Input Enhancement

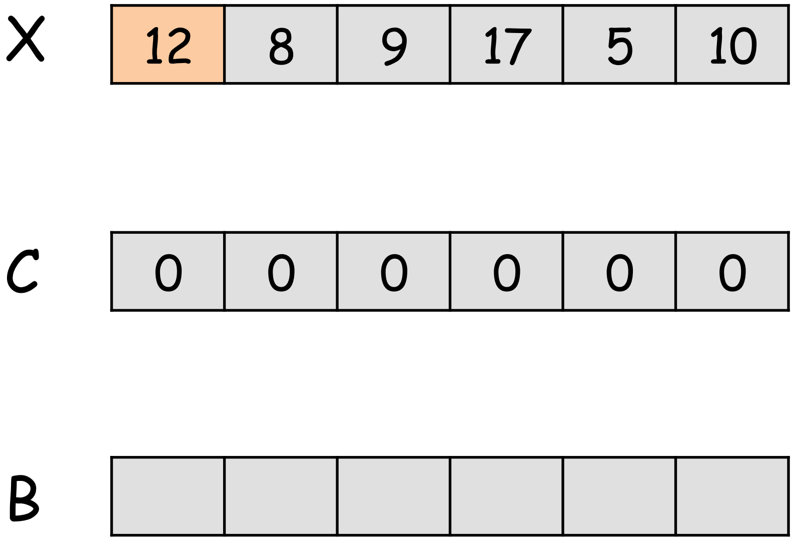
Sorting by Counting

- given an array of n orderable items $[a_1, a_2, \ldots, a_n]$, reorder the items as $[a_1', a_2', \ldots, a_n']$ such that $a_1' \leq a_2' \leq \ldots \leq a_n'$
- for each element of the array, count the total number of elements smaller than this number, and record the results in a table
- this count determines the position of the element in the final sorted array

  if the count is 5 for some element, then the element should be placed in the sixth position in the sorted array

CountingSort-1(X[1,n])

```
let C[1, n] and B[1, n] be new arrays
for i = 1 to n
    C[i] ← 0
for i = 1 to n - 1
    for j = i+1 to n
        if X[i] < X[j]
            C[j] ← C[j] + 1
        else
            C[i] ← C[i] + 1
for i = 1 to n
    B[C[i]+1] ← X[i]
return B
```



X	12	8	9	17	5	10

C	0	0	0	0	0	0

B						

Input Enhancement

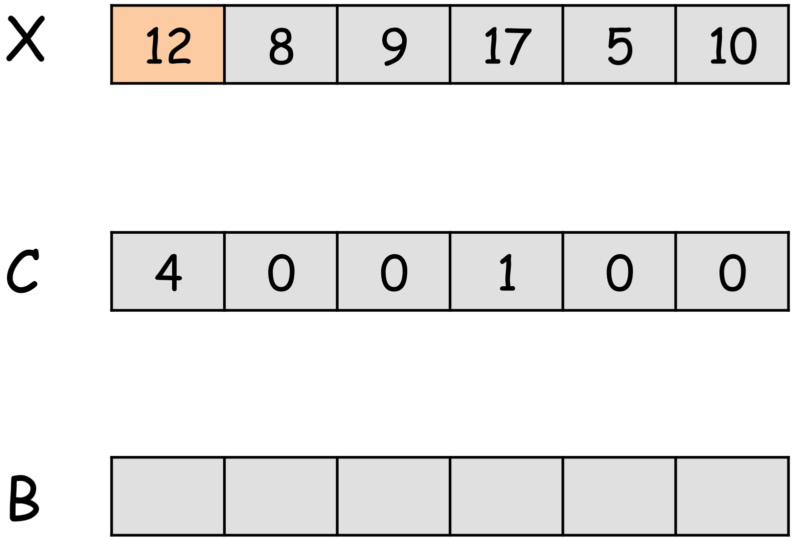
Sorting by Counting

- given an array of n orderable items [a_1, a_2,…, a_n], reorder the items as [a_1', a_2',…, a_n'] such that $a_1' \leq a_2' \leq \ldots \leq a_n'$
- for each element of the array, count the total number of elements smaller than this number, and record the results in a table
- this count determines the position of the element in the final sorted array

  if the count is 5 for some element, then the element should be placed in the sixth position in the sorted array

CountingSort-1(X[1,n])

```
let C[1, n] and B[1, n] be new arrays
for i = 1 to n
    C[i] ← 0
for i = 1 to n - 1
    for j = i+1 to n
        if X[i] < X[j]
            C[j] ← C[j] + 1
        else
            C[i] ← C[i] + 1
for i = 1 to n
    B[C[i]+1] ← X[i]
return B
```



X	12	8	9	17	5	10
C	4	0	0	1	0	0
B						

Input Enhancement

Sorting by Counting

- given an array of n orderable items $[a_1, a_2,\dots, a_n]$, reorder the items as $[a_1', a_2',\dots, a_n']$ such that $a_1' \le a_2' \le \dots \le a_n'$
- for each element of the array, count the total number of elements smaller than this number, and record the results in a table
- this count determines the position of the element in the final sorted array

  if the count is 5 for some element, then the element should be placed in the sixth position in the sorted array

CountingSort-1(X[1,n])

```
let C[1, n] and B[1, n] be new arrays
for i = 1 to n
    C[i] ← 0
for i = 1 to n - 1
    for j = i+1 to n
        if X[i] < X[j]
            C[j] ← C[j] + 1
        else
            C[i] ← C[i] + 1
for i = 1 to n
    B[C[i]+1] ← X[i]
return B
```

X	12	8	9	17	5	10
C	4	0	0	1	0	0
B						

Input Enhancement

Sorting by Counting

- given an array of n orderable items $[a_1, a_2, \dots, a_n]$, reorder the items as $[a_1', a_2', \dots, a_n']$ such that $a_1' \le a_2' \le \dots \le a_n'$
- for each element of the array, count the total number of elements smaller than this number, and record the results in a table
- this count determines the position of the element in the final sorted array

  if the count is 5 for some element, then the element should be placed in the sixth position in the sorted array

CountingSort-1(X[1,n])

```
let C[1, n] and B[1, n] be new arrays
for i = 1 to n
    C[i] ← 0
for i = 1 to n – 1
    for j = i+1 to n
        if X[i] < X[j]
            C[j] ← C[j] + 1
        else
            C[i] ← C[i] + 1
for i = 1 to n
    B[C[i]+1] ← X[i]
return B
```

X	12	8	9	17	5	10
C	4	1	1	2	0	1
B						

Input Enhancement

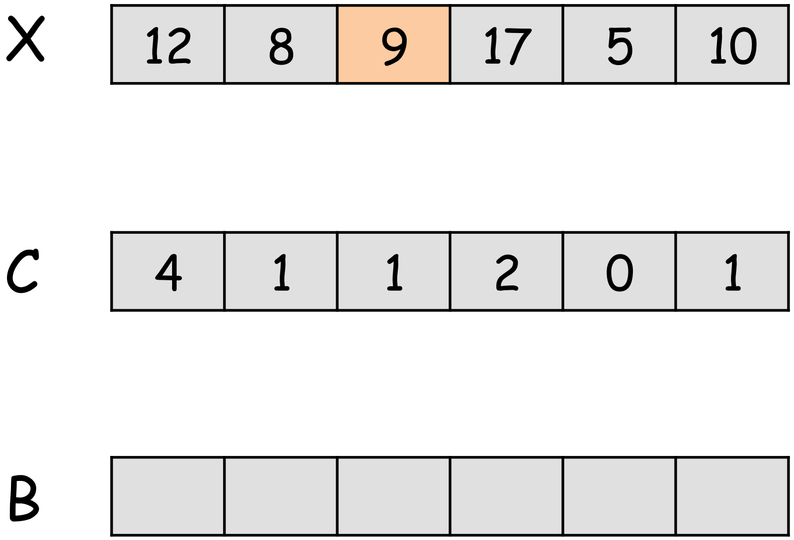
Sorting by Counting

- given an array of n orderable items $[a_1, a_2, \ldots, a_n]$, reorder the items as $[a_1', a_2', \ldots, a_n']$ such that $a_1' \le a_2' \le \ldots \le a_n'$
- for each element of the array, count the total number of elements smaller than this number, and record the results in a table
- this count determines the position of the element in the final sorted array

  if the count is 5 for some element, then the element should be placed in the sixth position in the sorted array

CountingSort-1(X[1 ,n])

```
let C[1, n] and B[1, n] be new arrays
for i = 1 to n
    C[i] ← 0
for i = 1 to n - 1
    for j = i+1 to n
        if X[i] < X[j]
            C[j] ← C[j] + 1
        else
            C[i] ← C[i] + 1
for i = 1 to n
    B[C[i]+1] ← X[i]
return B
```



X	12	8	9	17	5	10
C	4	1	1	2	0	1
B						

Input Enhancement

Sorting by Counting

- given an array of n orderable items $[a_1, a_2, \dots, a_n]$, reorder the items as $[a_1', a_2', \dots, a_n']$ such that $a_1' \leq a_2' \leq \dots \leq a_n'$
- for each element of the array, count the total number of elements smaller than this number, and record the results in a table
- this count determines the position of the element in the final sorted array

  if the count is 5 for some element, then the element should be placed in the sixth position in the sorted array

CountingSort-1(X[1 ,n])

```
let C[1, n] and B[1, n] be new arrays
for i = 1 to n
    C[i] ← 0
for i = 1 to n - 1
    for j = i+1 to n
        if X[i] < X[j]
            C[j] ← C[j] + 1
        else
            C[i] ← C[i] + 1
for i = 1 to n
    B[C[i]+1] ← X[i]
return B
```

X	12	8	9	17	5	10
C	4	1	2	3	0	2
B						

Input Enhancement

Sorting by Counting

- given an array of n orderable items [a_1, a_2,…, a_n], reorder the items as [a_1', a_2',…, a_n'] such that $a_1' \leq a_2' \leq \ldots \leq a_n'$
- for each element of the array, count the total number of elements smaller than this number, and record the results in a table
- this count determines the position of the element in the final sorted array

  if the count is 5 for some element, then the element should be placed in the sixth position in the sorted array

CountingSort-1(X[1,n])

```
let C[1, n] and B[1, n] be new arrays
for i = 1 to n
    C[i] ← 0
for i = 1 to n - 1
    for j = i+1 to n
        if X[i] < X[j]
            C[j] ← C[j] + 1
        else
            C[i] ← C[i] + 1
for i = 1 to n
    B[C[i]+1] ← X[i]
return B
```

X	12	8	9	17	5	10
C	4	1	2	3	0	2
B						

Input Enhancement

Sorting by Counting

- given an array of n orderable items [a_1, a_2,…, a_n], reorder the items as [a_1', a_2',…, a_n'] such that $a_1' \leq a_2' \leq … \leq a_n'$
- for each element of the array, count the total number of elements smaller than this number, and record the results in a table
- this count determines the position of the element in the final sorted array

  if the count is 5 for some element, then the element should be placed in the sixth position in the sorted array

CountingSort-1(X[1,n])

```
let C[1, n] and B[1, n] be new arrays
for i = 1 to n
    C[i] ← 0
for i = 1 to n – 1
    for j = i+1 to n
        if X[i] < X[j]
            C[j] ← C[j] + 1
        else
            C[i] ← C[i] + 1
for i = 1 to n
    B[C[i]+1] ← X[i]
return B
```

X	12	8	9	17	5	10
C	4	1	2	5	0	2
B						

Input Enhancement

Sorting by Counting

- given an array of n orderable items $[a_1, a_2, \dots, a_n]$, reorder the items as $[a_1', a_2', \dots, a_n']$ such that $a_1' \le a_2' \le \dots \le a_n'$
- for each element of the array, count the total number of elements smaller than this number, and record the results in a table
- this count determines the position of the element in the final sorted array

  if the count is 5 for some element, then the element should be placed in the sixth position in the sorted array

CountingSort-1(X[1,n])

```
let C[1, n] and B[1, n] be new arrays
for i = 1 to n
    C[i] ← 0
for i = 1 to n - 1
    for j = i+1 to n
        if X[i] < X[j]
            C[j] ← C[j] + 1
        else
            C[i] ← C[i] + 1
for i = 1 to n
    B[C[i]+1] ← X[i]
return B
```

X	12	8	9	17	5	10
C	4	1	2	5	0	2
B						

Input Enhancement

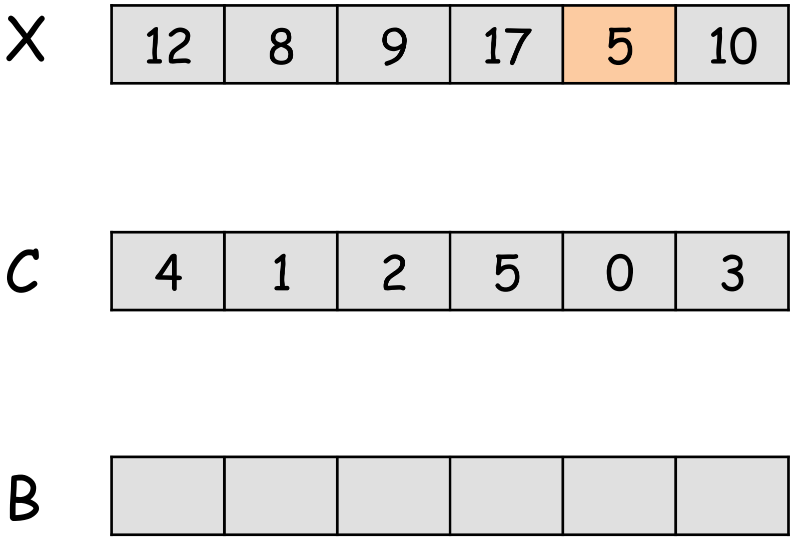
Sorting by Counting

- given an array of n orderable items [a_1, a_2,…, a_n], reorder the items as [a_1', a_2',…, a_n'] such that $a_1' \leq a_2' \leq \ldots \leq a_n'$
- for each element of the array, count the total number of elements smaller than this number, and record the results in a table
- this count determines the position of the element in the final sorted array

  if the count is 5 for some element, then the element should be placed in the sixth position in the sorted array

CountingSort-1(X[1,n])

```
let C[1, n] and B[1, n] be new arrays
for i = 1 to n
    C[i] ← 0
for i = 1 to n - 1
    for j = i+1 to n
        if X[i] < X[j]
            C[j] ← C[j] + 1
        else
            C[i] ← C[i] + 1
for i = 1 to n
    B[C[i]+1] ← X[i]
return B
```



X	12	8	9	17	5	10
C	4	1	2	5	0	3
B						

Input Enhancement

Sorting by Counting

- given an array of n orderable items $[a_1, a_2, \ldots, a_n]$, reorder the items as $[a_1', a_2', \ldots, a_n']$ such that $a_1' \leq a_2' \leq \ldots \leq a_n'$
- for each element of the array, count the total number of elements smaller than this number, and record the results in a table
- this count determines the position of the element in the final sorted array

  if the count is 5 for some element, then the element should be placed in the sixth position in the sorted array

CountingSort-1(X[1,n])

```
let C[1, n] and B[1, n] be new arrays
for i = 1 to n
    C[i] ← 0
for i = 1 to n - 1
    for j = i+1 to n
        if X[i] < X[j]
            C[j] ← C[j] + 1
        else
            C[i] ← C[i] + 1
for i = 1 to n
    B[C[i]+1] ← X[i]
return B
```

X	12	8	9	17	5	10
C	4	1	2	5	0	3
B						

Input Enhancement

Sorting by Counting

- given an array of n orderable items [$a_1, a_2, \dots, a_n$], reorder the items as [$a_1', a_2', \dots, a_n'$] such that $a_1' \le a_2' \le \dots \le a_n'$
- for each element of the array, count the total number of elements smaller than this number, and record the results in a table
- this count determines the position of the element in the final sorted array

  if the count is 5 for some element, then the element should be placed in the sixth position in the sorted array

CountingSort-1(X[1,n])

```
let C[1, n] and B[1, n] be new arrays
for i = 1 to n
    C[i] ← 0
for i = 1 to n - 1
    for j = i+1 to n
        if X[i] < X[j]
            C[j] ← C[j] + 1
        else
            C[i] ← C[i] + 1
for i = 1 to n
    B[C[i]+1] ← X[i]
return B
```

X	12	8	9	17	5	10
C	4	1	2	5	0	3
B					12	

Input Enhancement

Sorting by Counting

- given an array of n orderable items $[a_1, a_2, \ldots, a_n]$, reorder the items as $[a_1', a_2', \ldots, a_n']$ such that $a_1' \leq a_2' \leq \ldots \leq a_n'$
- for each element of the array, count the total number of elements smaller than this number, and record the results in a table
- this count determines the position of the element in the final sorted array

  if the count is 5 for some element, then the element should be placed in the sixth position in the sorted array

CountingSort-1(X[1,n])

```
let C[1, n] and B[1, n] be new arrays
for i = 1 to n
    C[i] ← 0
for i = 1 to n - 1
    for j = i+1 to n
        if X[i] < X[j]
            C[j] ← C[j] + 1
        else
            C[i] ← C[i] + 1
for i = 1 to n
    B[C[i]+1] ← X[i]
return B
```

X	12	8	9	17	5	10
C	4	1	2	5	0	3
B		8			12	

Input Enhancement

Sorting by Counting

- given an array of n orderable items $[a_1, a_2, \ldots, a_n]$, reorder the items as $[a_1', a_2', \ldots, a_n']$ such that $a_1' \le a_2' \le \ldots \le a_n'$
- for each element of the array, count the total number of elements smaller than this number, and record the results in a table
- this count determines the position of the element in the final sorted array

  if the count is 5 for some element, then the element should be placed in the sixth position in the sorted array

CountingSort-1(X[1,n])

```
let C[1, n] and B[1, n] be new arrays
for i = 1 to n
    C[i] ← 0
for i = 1 to n - 1
    for j = i+1 to n
        if X[i] < X[j]
            C[j] ← C[j] + 1
        else
            C[i] ← C[i] + 1
for i = 1 to n
    B[C[i]+1] ← X[i]
return B
```

X	12	8	9	17	5	10
C	4	1	2	5	0	3
B		8	9		12	

Input Enhancement

Sorting by Counting

- given an array of n orderable items $[a_1, a_2, \ldots, a_n]$, reorder the items as $[a_1', a_2', \ldots, a_n']$ such that $a_1' \leq a_2' \leq \ldots \leq a_n'$
- for each element of the array, count the total number of elements smaller than this number, and record the results in a table
- this count determines the position of the element in the final sorted array

  if the count is 5 for some element, then the element should be placed in the sixth position in the sorted array

CountingSort-1(X[1,n])

```
let C[1, n] and B[1, n] be new arrays
for i = 1 to n
    C[i] ← 0
for i = 1 to n - 1
    for j = i+1 to n
        if X[i] < X[j]
            C[j] ← C[j] + 1
        else
            C[i] ← C[i] + 1
for i = 1 to n
    B[C[i]+1] ← X[i]
return B
```

X	12	8	9	17	5	10
C	4	1	2	5	0	3
B		8	9		12	17

Input Enhancement

Sorting by Counting

- given an array of n orderable items $[a_1, a_2, \dots, a_n]$, reorder the items as $[a_1', a_2', \dots, a_n']$ such that $a_1' \le a_2' \le \dots \le a_n'$
- for each element of the array, count the total number of elements smaller than this number, and record the results in a table
- this count determines the position of the element in the final sorted array

  if the count is 5 for some element, then the element should be placed in the sixth position in the sorted array

CountingSort-1(X[1,n])

```
let C[1, n] and B[1, n] be new arrays
for i = 1 to n
    C[i] ← 0
for i = 1 to n - 1
    for j = i+1 to n
        if X[i] < X[j]
            C[j] ← C[j] + 1
        else
            C[i] ← C[i] + 1
for i = 1 to n
    B[C[i]+1] ← X[i]
return B
```

X	12	8	9	17	5	10
C	4	1	2	5	0	3
B	5	8	9		12	17

Input Enhancement

Sorting by Counting

- given an array of n orderable items $[a_1, a_2, \ldots, a_n]$, reorder the items as $[a_1', a_2', \ldots, a_n']$ such that $a_1' \le a_2' \le \ldots \le a_n'$
- for each element of the array, count the total number of elements smaller than this number, and record the results in a table
- this count determines the position of the element in the final sorted array

  if the count is 5 for some element, then the element should be placed in the sixth position in the sorted array

CountingSort-1(X[1,n])

```
let C[1, n] and B[1, n] be new arrays
for i = 1 to n
    C[i] 🡐 0
for i = 1 to n - 1
    for j = i+1 to n
        if X[i] < X[j]
            C[j] 🡐 C[j] + 1
        else
            C[i] 🡐 C[i] + 1
for i = 1 to n
    B[C[i]+1] 🡐 X[i]
return B
```

X	12	8	9	17	5	10

C	4	1	2	5	0	3

B	5	8	9	10	12	17

Input Enhancement

Sorting by Counting

- given an array of n orderable items $[a_1, a_2, \ldots, a_n]$, reorder the items as $[a_1', a_2', \ldots, a_n']$ such that $a_1' \le a_2' \le \ldots \le a_n'$
- for each element of the array, count the total number of elements smaller than this number, and record the results in a table
- this count determines the position of the element in the final sorted array

  if the count is 5 for some element, then the element should be placed in the sixth position in the sorted array

CountingSort-1(X[1,n])

```
let C[1, n] and B[1, n] be new arrays
for i = 1 to n
    C[i] ← 0
for i = 1 to n - 1
    for j = i+1 to n
        if X[i] < X[j]
            C[j] ← C[j] + 1
        else
            C[i] ← C[i] + 1
for i = 1 to n
    B[C[i]+1] ← X[i]
return B
```

Input Enhancement

Sorting by Counting

- given an array of n orderable items $[a_1, a_2, \dots, a_n]$, reorder the items as $[a_1', a_2', \dots, a_n']$ such that $a_1' \le a_2' \le \dots \le a_n'$
- for each element of the array, count the total number of elements smaller than this number, and record the results in a table
- this count determines the position of the element in the final sorted array

  if the count is 5 for some element, then the element should be placed in the sixth position in the sorted array

CountingSort-1(X[1,n])

```
let C[1, n] and B[1, n] be new arrays
for i = 1 to n
    C[i] ← 0                        → O(n)
for i = 1 to n - 1
    for j = i+1 to n
        if X[i] < X[j]
            C[j] ← C[j] + 1         → $O(n^2)$
        else
            C[i] ← C[i] + 1
for i = 1 to n
    B[C[i]+1] ← X[i]                → O(n)
return B
```

Input Enhancement

Sorting by Counting

- given an array of n orderable items $[a_1, a_2, \dots, a_n]$, reorder the items as $[a_1', a_2', \dots, a_n']$ such that $a_1' \leq a_2' \leq \dots \leq a_n'$
- for each element of the array, count the total number of elements smaller than this number, and record the results in a table
- this count determines the position of the element in the final sorted array

  if the count is 5 for some element, then the element should be placed in the sixth position in the sorted array

CountingSort-1(X[1,n])

```
let C[1, n] and B[1, n] be new arrays
for i = 1 to n
    C[i] ← 0                          ──────────→  O(n)
for i = 1 to n - 1
    for j = i+1 to n
        if X[i] < X[j]
            C[j] ← C[j] + 1           ──────────→  O(n^2)     time complexity : O(n^2)
        else
            C[i] ← C[i] + 1
for i = 1 to n
    B[C[i]+1] ← X[i]                  ──────────→  O(n)
return B
```

Input Enhancement

Sorting by Counting

- given an array of n orderable items $[a_1, a_2, \dots, a_n]$, reorder the items as $[a_1', a_2', \dots, a_n']$ such that $a_1' \le a_2' \le \dots \le a_n'$
- for each element of the array, count the total number of elements smaller than this number, and record the results in a table
- this count determines the position of the element in the final sorted array

  if the count is 5 for some element, then the element should be placed in the sixth position in the sorted array

CountingSort-1(X[1,n])

```
let C[1, n] and B[1, n] be new arrays  →  space complexity : O(n)
for i = 1 to n
    C[i] ← 0                           →  O(n)
for i = 1 to n – 1
    for j = i+1 to n
        if X[i] < X[j]
            C[j] ← C[j] + 1            →  O(n^2)      time complexity : O(n^2)
        else
            C[i] ← C[i] + 1
for i = 1 to n
    B[C[i]+1] ← X[i]                   →  O(n)
return B
```

Input Enhancement

Sorting by Counting

- given an array of n orderable items [a_1, a_2,…, a_n], reorder the items as [a_1', a_2',…, a_n'] such that $a_1' \le a_2' \le \ldots \le a_n'$
- for each element of the array, count the total number of elements smaller than this number, and record the results in a table
- this count determines the position of the element in the final sorted array

  if the count is 5 for some element, then the element should be placed in the sixth position in the sorted array

CountingSort-1(X[1,n])

```
let C[1, n] and B[1, n] be new arrays
for i = 1 to n
    C[i] ← 0
for i = 1 to n - 1
    for j = i+1
        if X[i
            C[j
        else
            C[i] ← C[i] + 1
for i = 1 to n
    B[C[i]+1] ← X[i]
return B
```

space complexity : $O(n)$

$O(n)$

: $O(n^2)$

$O(n)$

we can achieve a Counting Sort algorithm with $O(n)$ running time if each of the input elements is an integer in the range $[0,k]$ where $k = O(n)$

Input Enhancement

Sorting by Counting

- given an array of n orderable items $[a_1, a_2, \ldots, a_n]$, reorder the items as $[a_1', a_2', \ldots, a_n']$ such that $a_1' \le a_2' \le \ldots \le a_n'$

CountingSort-2(X[1,n],k)

```
let C[1, k] and B[1, n] be new arrays
for i = 0 to k
    C[i] ← 0
for j = 1 to n
    C[X[j]] ← C[X[j]] + 1
for i = 1 to k
    C[i] ← C[i] + C[i-1]
for j = n to 1
    B[C[X[j]]] ← X[j]
    C[X[j]] ← C[X[j]] – 1
return B
```

Input Enhancement

Sorting by Counting

- given an array of n orderable items $[a_1, a_2, \ldots, a_n]$, reorder the items as $[a_1', a_2', \ldots, a_n']$ such that $a_1' \leq a_2' \leq \ldots \leq a_n'$

CountingSort-2(X[1,n],k)

```
let C[1, k] and B[1, n] be new arrays
for i = 0 to k
    C[i] ← 0
for j = 1 to n
    C[X[j]] ← C[X[j]] + 1
for i = 1 to k
    C[i] ← C[i] + C[i-1]
for j = n to 1
    B[C[X[j]]] ← X[j]
    C[X[j]] ← C[X[j]] – 1
return B
```

X	4	2	0	6	2	0	3	2

Input Enhancement

Sorting by Counting

- given an array of n orderable items $[a_1, a_2, \ldots, a_n]$, reorder the items as $[a_1', a_2', \ldots, a_n']$ such that $a_1' \le a_2' \le \ldots \le a_n'$

CountingSort-2(X[1,n],k)

```
let C[1, k] and B[1, n] be new arrays
for i = 0 to k
    C[i] ← 0
for j = 1 to n
    C[X[j]] ← C[X[j]] + 1
for i = 1 to k
    C[i] ← C[i] + C[i-1]
for j = n to 1
    B[C[X[j]]] ← X[j]
    C[X[j]] ← C[X[j]] - 1
return B
```

X

4	2	0	6	2	0	3	2

C

0	1	2	3	4	5	6

B

Input Enhancement

Sorting by Counting

- given an array of n orderable items $[a_1, a_2, \ldots, a_n]$, reorder the items as $[a_1', a_2', \ldots, a_n']$ such that $a_1' \le a_2' \le \ldots \le a_n'$

```
CountingSort-2(X[1,n],k)
let C[1, k] and B[1, n] be new arrays
for i = 0 to k
    C[i] ← 0
for j = 1 to n
    C[X[j]] ← C[X[j]] + 1
for i = 1 to k
    C[i] ← C[i] + C[i-1]
for j = n to 1
    B[C[X[j]]] ← X[j]
    C[X[j]] ← C[X[j]] - 1
return B
```

X

4	2	0	6	2	0	3	2

C

0	1	2	3	4	5	6
0	0	0	0	0	0	0

B

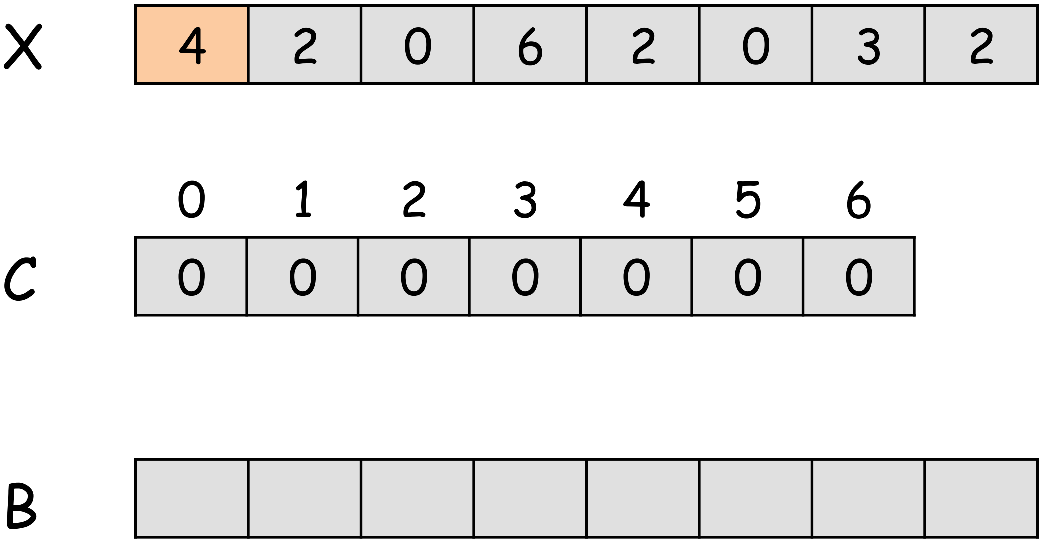
Input Enhancement

Sorting by Counting

- given an array of n orderable items $[a_1, a_2,\dots, a_n]$, reorder the items as $[a_1', a_2',\dots, a_n']$ such that $a_1' \le a_2' \le \dots \le a_n'$

CountingSort-2(X[1,n],k)

```
let C[1, k] and B[1, n] be new arrays
for i = 0 to k
    C[i] ← 0
for j = 1 to n
    C[X[j]] ← C[X[j]] + 1
for i = 1 to k
    C[i] ← C[i] + C[i-1]
for j = n to 1
    B[C[X[j]]] ← X[j]
    C[X[j]] ← C[X[j]] - 1
return B
```



X	4	2	0	6	2	0	3	2

	0	1	2	3	4	5	6
C	0	0	0	0	0	0	0

B								

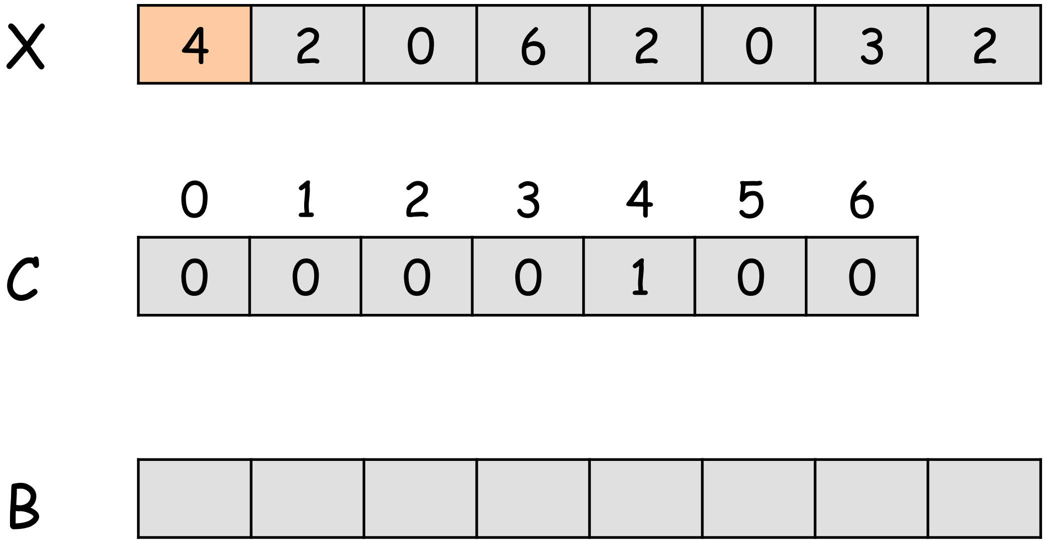
Input Enhancement

Sorting by Counting

- given an array of n orderable items $[a_1, a_2, \ldots, a_n]$, reorder the items as $[a_1', a_2', \ldots, a_n']$ such that $a_1' \leq a_2' \leq \ldots \leq a_n'$

CountingSort-2(X[1,n],k)

```
let C[1, k] and B[1, n] be new arrays
for i = 0 to k
    C[i] ← 0
for j = 1 to n
    C[X[j]] ← C[X[j]] + 1
for i = 1 to k
    C[i] ← C[i] + C[i-1]
for j = n to 1
    B[C[X[j]]] ← X[j]
    C[X[j]] ← C[X[j]] - 1
return B
```



X	4	2	0	6	2	0	3	2

	0	1	2	3	4	5	6
C	0	0	0	0	1	0	0

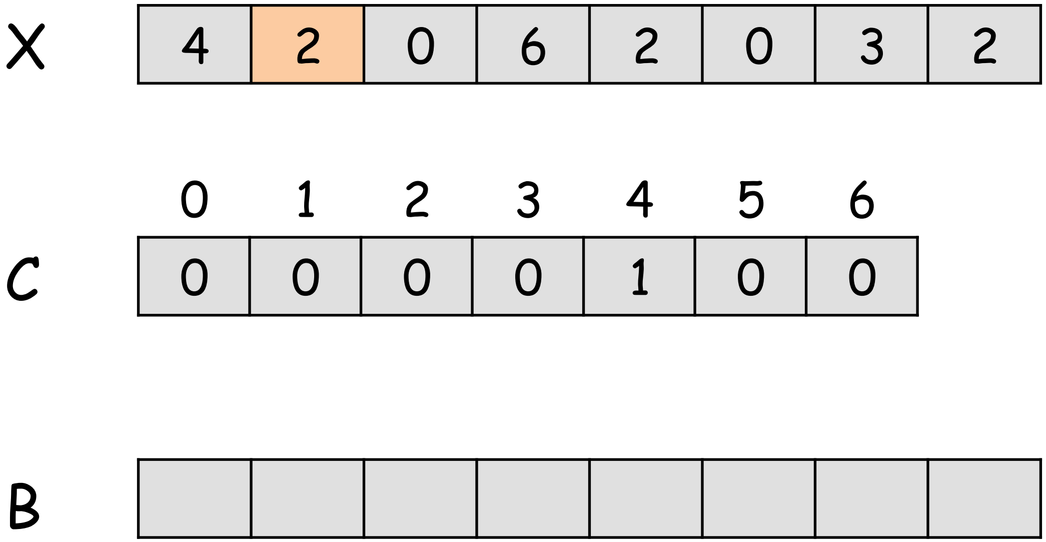
B								

Input Enhancement

Sorting by Counting

- given an array of n orderable items [a_1, a_2,…, a_n], reorder the items as [a_1', a_2',…, a_n'] such that $a_1' \le a_2' \le \ldots \le a_n'$

```
CountingSort-2(X[1,n],k)
let C[1, k] and B[1, n] be new arrays
for i = 0 to k
    C[i] ← 0
for j = 1 to n
    C[X[j]] ← C[X[j]] + 1
for i = 1 to k
    C[i] ← C[i] + C[i-1]
for j = n to 1
    B[C[X[j]]] ← X[j]
    C[X[j]] ← C[X[j]] - 1
return B
```



X	4	2	0	6	2	0	3	2

	0	1	2	3	4	5	6
C	0	0	0	0	1	0	0

B								

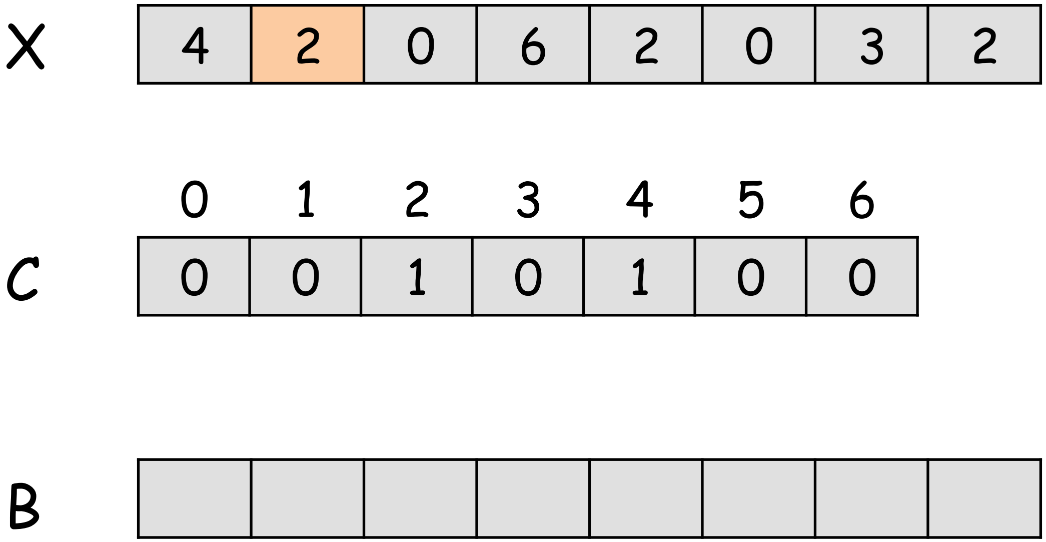
Input Enhancement

Sorting by Counting

- given an array of n orderable items $[a_1, a_2, \ldots, a_n]$, reorder the items as $[a_1', a_2', \ldots, a_n']$ such that $a_1' \le a_2' \le \ldots \le a_n'$

CountingSort-2(X[1,n],k)

```
let C[1, k] and B[1, n] be new arrays
for i = 0 to k
    C[i] ← 0
for j = 1 to n
    C[X[j]] ← C[X[j]] + 1
for i = 1 to k
    C[i] ← C[i] + C[i-1]
for j = n to 1
    B[C[X[j]]] ← X[j]
    C[X[j]] ← C[X[j]] - 1
return B
```



X	4	2	0	6	2	0	3	2

	0	1	2	3	4	5	6
C	0	0	1	0	1	0	0

B								

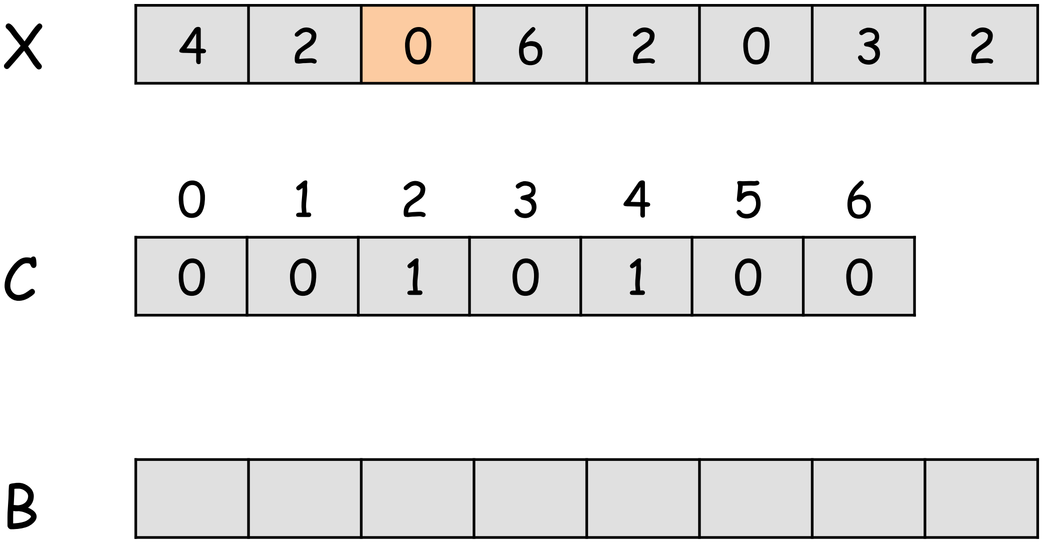
Input Enhancement

Sorting by Counting

- given an array of n orderable items [a_1, a_2,…, a_n], reorder the items as [a_1', a_2',…, a_n'] such that $a_1' \leq a_2' \leq \ldots \leq a_n'$

CountingSort-2(X[1,n],k)

```
let C[1, k] and B[1, n] be new arrays
for i = 0 to k
    C[i] ← 0
for j = 1 to n
    C[X[j]] ← C[X[j]] + 1
for i = 1 to k
    C[i] ← C[i] + C[i-1]
for j = n to 1
    B[C[X[j]]] ← X[j]
    C[X[j]] ← C[X[j]] - 1
return B
```



X

4	2	0	6	2	0	3	2

C

0	1	2	3	4	5	6
0	0	1	0	1	0	0

B

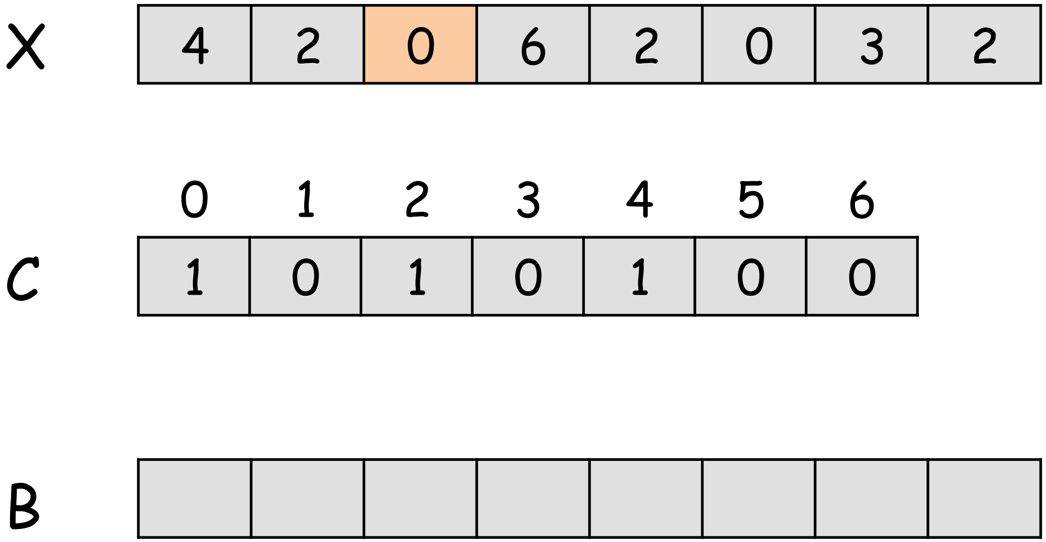
Input Enhancement

Sorting by Counting

- given an array of n orderable items $[a_1, a_2, \ldots, a_n]$, reorder the items as $[a_1', a_2', \ldots, a_n']$ such that $a_1' \leq a_2' \leq \ldots \leq a_n'$

CountingSort-2(X[1,n],k)

```
let C[1, k] and B[1, n] be new arrays
for i = 0 to k
    C[i] ← 0
for j = 1 to n
    C[X[j]] ← C[X[j]] + 1
for i = 1 to k
    C[i] ← C[i] + C[i-1]
for j = n to 1
    B[C[X[j]]] ← X[j]
    C[X[j]] ← C[X[j]] - 1
return B
```



X	4	2	0	6	2	0	3	2

	0	1	2	3	4	5	6
C	1	0	1	0	1	0	0

B								

Input Enhancement

Sorting by Counting

- given an array of n orderable items [a_1, a_2,…, a_n], reorder the items as [a_1', a_2',…, a_n'] such that $a_1' \leq a_2' \leq \ldots \leq a_n'$

```
CountingSort-2(X[1,n],k)
let C[1, k] and B[1, n] be new arrays
for i = 0 to k
    C[i] ← 0
for j = 1 to n
    C[X[j]] ← C[X[j]] + 1
for i = 1 to k
    C[i] ← C[i] + C[i-1]
for j = n to 1
    B[C[X[j]]] ← X[j]
    C[X[j]] ← C[X[j]] - 1
return B
```

X

4	2	0	6	2	0	3	2

C

0	1	2	3	4	5	6
2	0	3	1	1	0	1

B

Input Enhancement

Sorting by Counting

- given an array of n orderable items [a_1, a_2,…, a_n], reorder the items as [a_1', a_2',…, a_n'] such that $a_1' \leq a_2' \leq \ldots \leq a_n'$

```
CountingSort-2(X[1,n],k)
let C[1, k] and B[1, n] be new arrays
for i = 0 to k
    C[i] ← 0
for j = 1 to n
    C[X[j]] ← C[X[j]] + 1
for i = 1 to k
    C[i] ← C[i] + C[i-1]
for j = n to 1
    B[C[X[j]]] ← X[j]
    C[X[j]] ← C[X[j]] - 1
return B
```

X	4	2	0	6	2	0	3	2

	0	1	2	3	4	5	6
C	2	0	3	1	1	0	1

B								

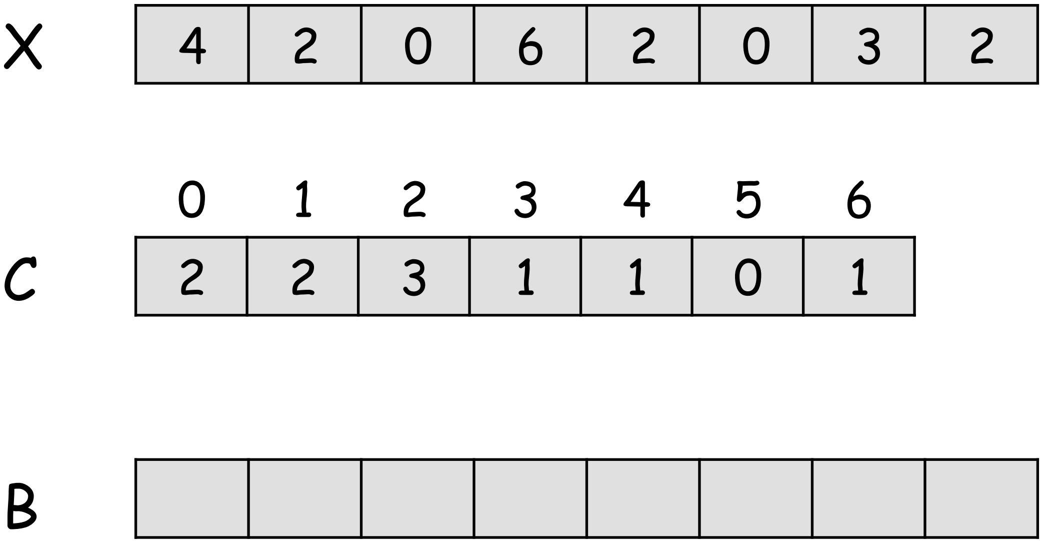
Input Enhancement

Sorting by Counting

- given an array of n orderable items [a_1, a_2,…, a_n], reorder the items as [a_1', a_2',…, a_n'] such that $a_1' \le a_2' \le \ldots \le a_n'$

CountingSort-2(X[1,n],k)

```
let C[1, k] and B[1, n] be new arrays
for i = 0 to k
    C[i] ← 0
for j = 1 to n
    C[X[j]] ← C[X[j]] + 1
for i = 1 to k
    C[i] ← C[i] + C[i-1]
for j = n to 1
    B[C[X[j]]] ← X[j]
    C[X[j]] ← C[X[j]] - 1
return B
```



X							
4	2	0	6	2	0	3	2

C	0	1	2	3	4	5	6
	2	2	3	1	1	0	1

B							

Input Enhancement

Sorting by Counting

- given an array of n orderable items $[a_1, a_2, \ldots, a_n]$, reorder the items as $[a_1', a_2', \ldots, a_n']$ such that $a_1' \le a_2' \le \ldots \le a_n'$

CountingSort-2(X[1,n],k)

```
let C[1, k] and B[1, n] be new arrays
for i = 0 to k
    C[i] ← 0
for j = 1 to n
    C[X[j]] ← C[X[j]] + 1
for i = 1 to k
    C[i] ← C[i] + C[i-1]
for j = n to 1
    B[C[X[j]]] ← X[j]
    C[X[j]] ← C[X[j]] - 1
return B
```

X	4	2	0	6	2	0	3	2

	0	1	2	3	4	5	6
C	2	2	5	1	1	0	1

B								

Input Enhancement

Sorting by Counting

- given an array of n orderable items $[a_1, a_2, \dots, a_n]$, reorder the items as $[a_1', a_2', \dots, a_n']$ such that $a_1' \le a_2' \le \dots \le a_n'$

```
CountingSort-2(X[1,n],k)
let C[1, k] and B[1, n] be new arrays
for i = 0 to k
    C[i] ← 0
for j = 1 to n
    C[X[j]] ← C[X[j]] + 1
for i = 1 to k
    C[i] ← C[i] + C[i-1]
for j = n to 1
    B[C[X[j]]] ← X[j]
    C[X[j]] ← C[X[j]] - 1
return B
```

X	4	2	0	6	2	0	3	2

	0	1	2	3	4	5	6
C	2	2	5	6	1	0	1

B								

Input Enhancement

Sorting by Counting

- given an array of n orderable items $[a_1, a_2, \dots, a_n]$, reorder the items as $[a_1', a_2', \dots, a_n']$ such that $a_1' \le a_2' \le \dots \le a_n'$

```
CountingSort-2(X[1,n],k)
let C[1, k] and B[1, n] be new arrays
for i = 0 to k
    C[i] ← 0
for j = 1 to n
    C[X[j]] ← C[X[j]] + 1
for i = 1 to k
    C[i] ← C[i] + C[i-1]
for j = n to 1
    B[C[X[j]]] ← X[j]
    C[X[j]] ← C[X[j]] - 1
return B
```

X	4	2	0	6	2	0	3	2

	0	1	2	3	4	5	6
C	2	2	5	6	7	0	1

B								

Input Enhancement

Sorting by Counting

- given an array of n orderable items [a_1, a_2,…, a_n], reorder the items as [a_1', a_2',…, a_n'] such that $a_1' \le a_2' \le … \le a_n'$

```
CountingSort-2(X[1,n],k)
let C[1, k] and B[1, n] be new arrays
for i = 0 to k
    C[i] ← 0
for j = 1 to n
    C[X[j]] ← C[X[j]] + 1
for i = 1 to k
    C[i] ← C[i] + C[i-1]
for j = n to 1
    B[C[X[j]]] ← X[j]
    C[X[j]] ← C[X[j]] - 1
return B
```

X

4	2	0	6	2	0	3	2

C

0	1	2	3	4	5	6
2	2	5	6	7	7	1

B

Input Enhancement

Sorting by Counting

- given an array of n orderable items $[a_1, a_2, \dots, a_n]$, reorder the items as $[a_1', a_2', \dots, a_n']$ such that $a_1' \le a_2' \le \dots \le a_n'$

CountingSort-2(X[1,n],k)

```
let C[1, k] and B[1, n] be new arrays
for i = 0 to k
    C[i] ← 0
for j = 1 to n
    C[X[j]] ← C[X[j]] + 1
for i = 1 to k
    C[i] ← C[i] + C[i-1]
for j = n to 1
    B[C[X[j]]] ← X[j]
    C[X[j]] ← C[X[j]] - 1
return B
```

X

4	2	0	6	2	0	3	2

C

0	1	2	3	4	5	6
2	2	5	6	7	7	8

B

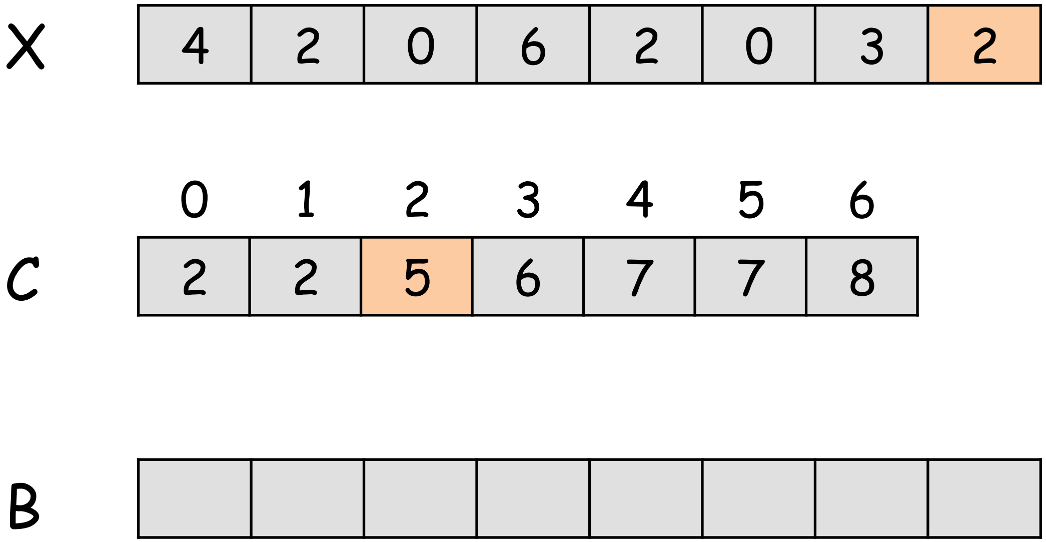
Input Enhancement

Sorting by Counting

- given an array of n orderable items $[a_1, a_2, \ldots, a_n]$, reorder the items as $[a_1', a_2', \ldots, a_n']$ such that $a_1' \leq a_2' \leq \ldots \leq a_n'$

CountingSort-2(X[1,n],k)

```
let C[1, k] and B[1, n] be new arrays
for i = 0 to k
    C[i] ← 0
for j = 1 to n
    C[X[j]] ← C[X[j]] + 1
for i = 1 to k
    C[i] ← C[i] + C[i-1]
for j = n to 1
    B[C[X[j]]] ← X[j]
    C[X[j]] ← C[X[j]] - 1
return B
```



X

4	2	0	6	2	0	3	2

C

0	1	2	3	4	5	6
2	2	5	6	7	7	8

B

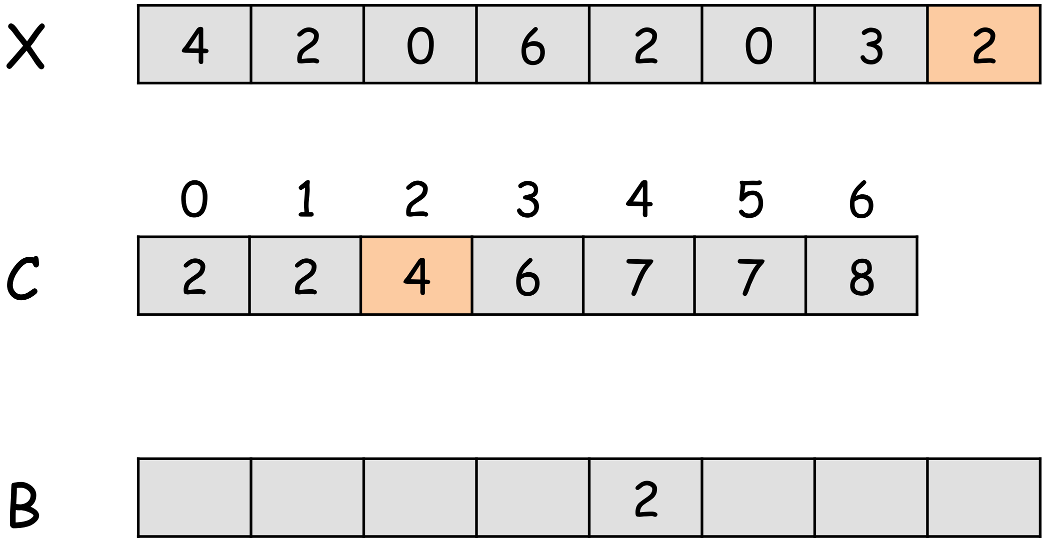
Input Enhancement

Sorting by Counting

- given an array of n orderable items $[a_1, a_2, \ldots, a_n]$, reorder the items as $[a_1', a_2', \ldots, a_n']$ such that $a_1' \le a_2' \le \ldots \le a_n'$

CountingSort-2(X[1,n],k)

```
let C[1, k] and B[1, n] be new arrays
for i = 0 to k
    C[i] ← 0
for j = 1 to n
    C[X[j]] ← C[X[j]] + 1
for i = 1 to k
    C[i] ← C[i] + C[i-1]
for j = n to 1
    B[C[X[j]]] ← X[j]
    C[X[j]] ← C[X[j]] - 1
return B
```



X	4	2	0	6	2	0	3	2

	0	1	2	3	4	5	6
C	2	2	4	6	7	7	8

B					2			

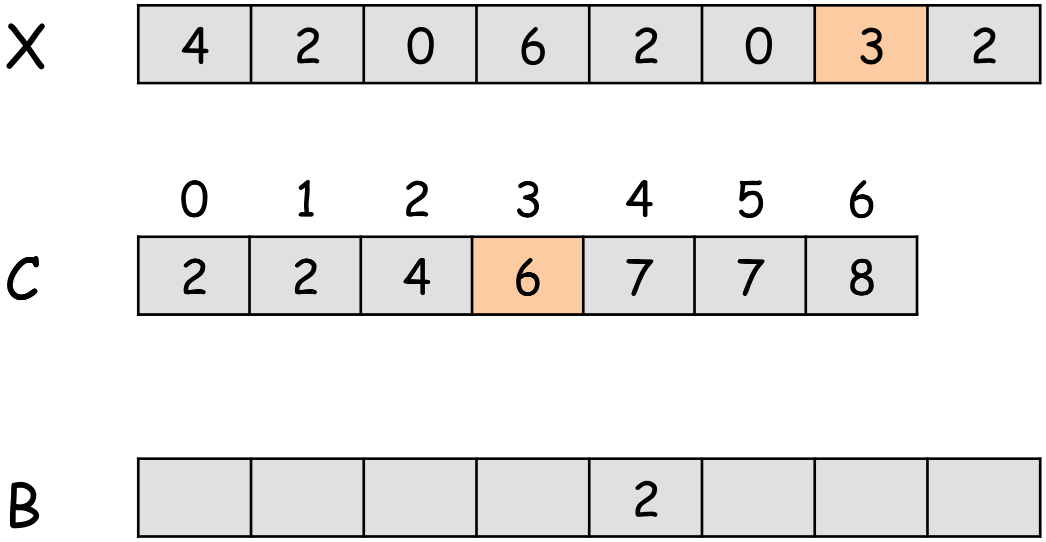
Input Enhancement

Sorting by Counting

- given an array of n orderable items $[a_1, a_2, \ldots, a_n]$, reorder the items as $[a_1', a_2', \ldots, a_n']$ such that $a_1' \le a_2' \le \ldots \le a_n'$

CountingSort-2(X[1,n],k)

```
let C[1, k] and B[1, n] be new arrays
for i = 0 to k
    C[i] ← 0
for j = 1 to n
    C[X[j]] ← C[X[j]] + 1
for i = 1 to k
    C[i] ← C[i] + C[i-1]
for j = n to 1
    B[C[X[j]]] ← X[j]
    C[X[j]] ← C[X[j]] - 1
return B
```



X	4	2	0	6	2	0	3	2

	0	1	2	3	4	5	6
C	2	2	4	6	7	7	8

B					2			

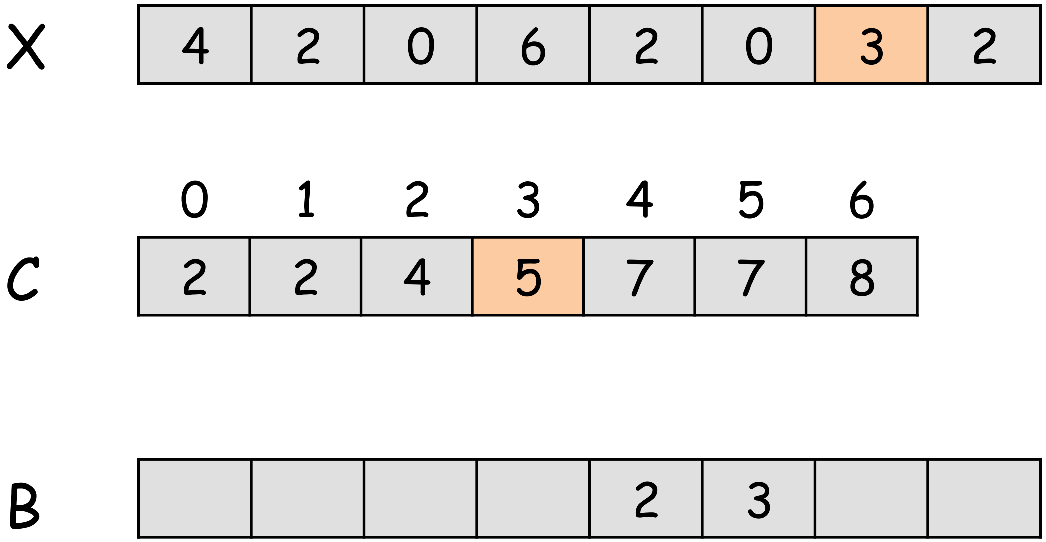
Input Enhancement

Sorting by Counting

- given an array of n orderable items $[a_1, a_2, \ldots, a_n]$, reorder the items as $[a_1', a_2', \ldots, a_n']$ such that $a_1' \leq a_2' \leq \ldots \leq a_n'$

CountingSort-2(X[1,n],k)

```
let C[1, k] and B[1, n] be new arrays
for i = 0 to k
    C[i] ← 0
for j = 1 to n
    C[X[j]] ← C[X[j]] + 1
for i = 1 to k
    C[i] ← C[i] + C[i-1]
for j = n to 1
    B[C[X[j]]] ← X[j]
    C[X[j]] ← C[X[j]] - 1
return B
```



X

4	2	0	6	2	0	3	2

C

0	1	2	3	4	5	6
2	2	4	5	7	7	8

B

				2	3		

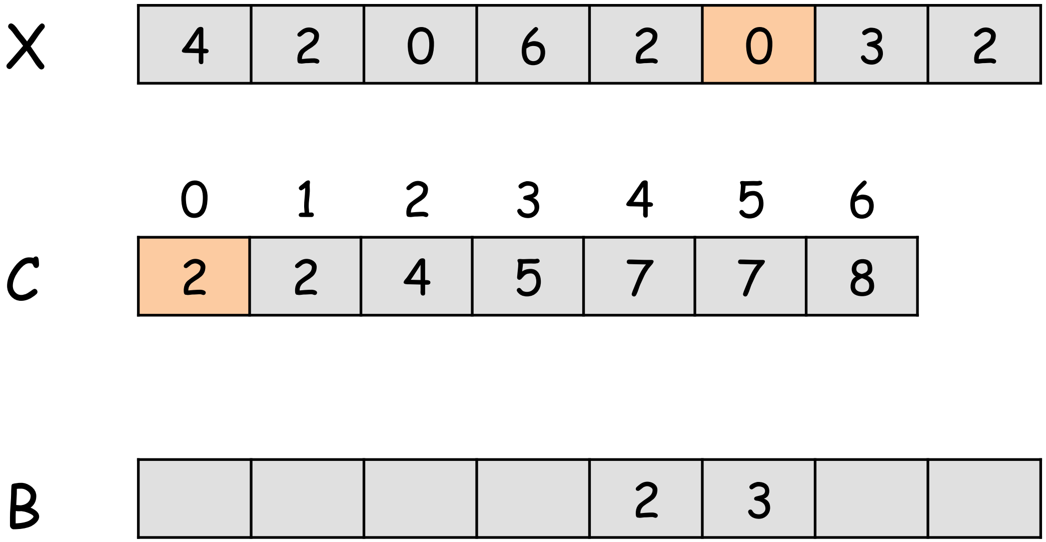
Input Enhancement

Sorting by Counting

- given an array of n orderable items $[a_1, a_2, \ldots, a_n]$, reorder the items as $[a_1', a_2', \ldots, a_n']$ such that $a_1' \leq a_2' \leq \ldots \leq a_n'$

CountingSort-2(X[1,n],k)

```
let C[1, k] and B[1, n] be new arrays
for i = 0 to k
    C[i] ← 0
for j = 1 to n
    C[X[j]] ← C[X[j]] + 1
for i = 1 to k
    C[i] ← C[i] + C[i-1]
for j = n to 1
    B[C[X[j]]] ← X[j]
    C[X[j]] ← C[X[j]] - 1
return B
```



X	4	2	0	6	2	0	3	2

	0	1	2	3	4	5	6
C	2	2	4	5	7	7	8

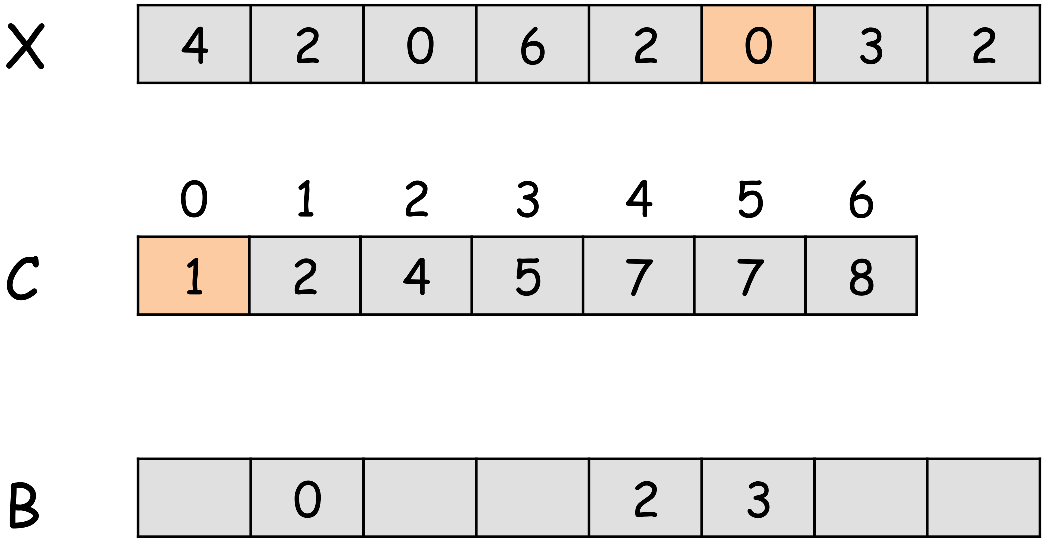
B					2	3		

Input Enhancement

Sorting by Counting

- given an array of n orderable items $[a_1, a_2,\ldots, a_n]$, reorder the items as $[a_1', a_2',\ldots, a_n']$ such that $a_1' \leq a_2' \leq \ldots \leq a_n'$

```
CountingSort-2(X[1,n],k)
let C[1, k] and B[1, n] be new arrays
for i = 0 to k
    C[i] ← 0
for j = 1 to n
    C[X[j]] ← C[X[j]] + 1
for i = 1 to k
    C[i] ← C[i] + C[i-1]
for j = n to 1
    B[C[X[j]]] ← X[j]
    C[X[j]] ← C[X[j]] - 1
return B
```



X

4	2	0	6	2	0	3	2

C

0	1	2	3	4	5	6
1	2	4	5	7	7	8

B

	0			2	3		

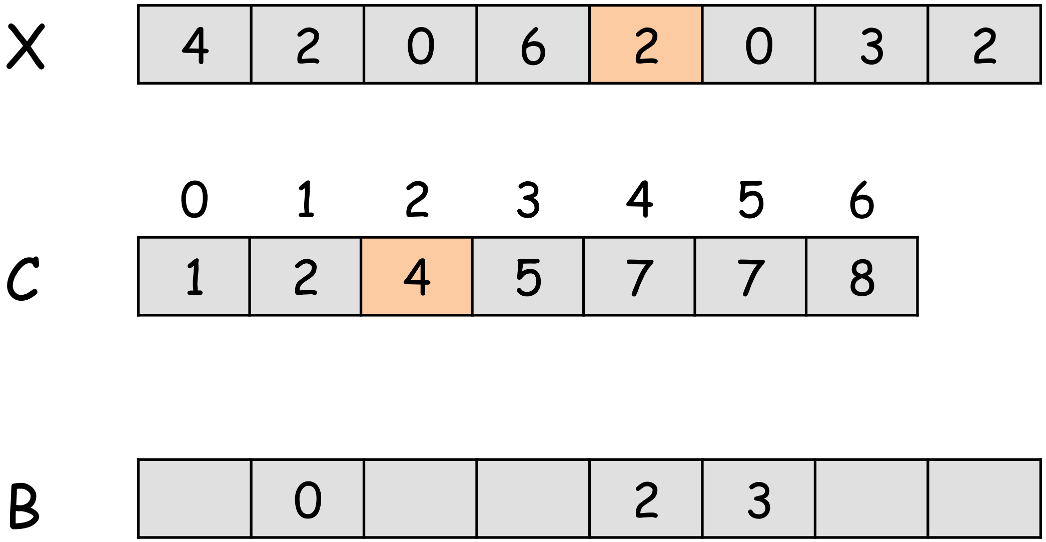
Input Enhancement

Sorting by Counting

- given an array of n orderable items $[a_1, a_2, \ldots, a_n]$, reorder the items as $[a_1', a_2', \ldots, a_n']$ such that $a_1' \le a_2' \le \ldots \le a_n'$

CountingSort-2(X[1,n],k)

```
let C[1, k] and B[1, n] be new arrays
for i = 0 to k
    C[i] ← 0
for j = 1 to n
    C[X[j]] ← C[X[j]] + 1
for i = 1 to k
    C[i] ← C[i] + C[i-1]
for j = n to 1
    B[C[X[j]]] ← X[j]
    C[X[j]] ← C[X[j]] - 1
return B
```



X

4	2	0	6	2	0	3	2

C

0	1	2	3	4	5	6
1	2	4	5	7	7	8

B

	0			2	3		

Input Enhancement

Sorting by Counting

- given an array of n orderable items $[a_1, a_2, \ldots, a_n]$, reorder the items as $[a_1', a_2', \ldots, a_n']$ such that $a_1' \le a_2' \le \ldots \le a_n'$

CountingSort-2(X[1,n],k)

```
let C[1, k] and B[1, n] be new arrays
for i = 0 to k
    C[i] ← 0
for j = 1 to n
    C[X[j]] ← C[X[j]] + 1
for i = 1 to k
    C[i] ← C[i] + C[i-1]
for j = n to 1
    B[C[X[j]]] ← X[j]
    C[X[j]] ← C[X[j]] - 1
return B
```

X

4	2	0	6	2	0	3	2

C

0	1	2	3	4	5	6
1	2	3	5	7	7	8

B

	0		2	2	3		

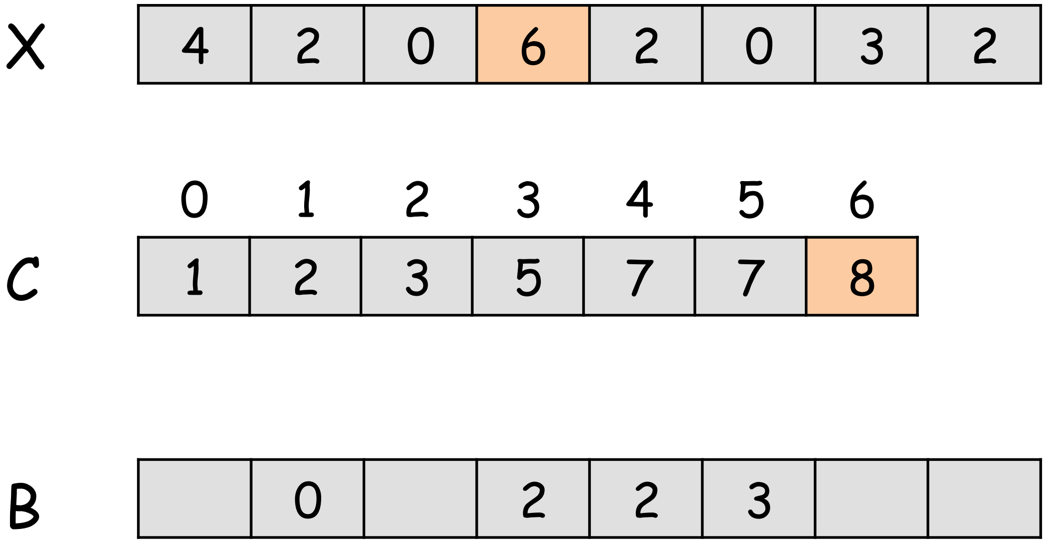
Input Enhancement

Sorting by Counting

- given an array of n orderable items $[a_1, a_2, \dots, a_n]$, reorder the items as $[a_1', a_2', \dots, a_n']$ such that $a_1' \le a_2' \le \dots \le a_n'$

CountingSort-2(X[1,n],k)

```
let C[1, k] and B[1, n] be new arrays
for i = 0 to k
    C[i] ← 0
for j = 1 to n
    C[X[j]] ← C[X[j]] + 1
for i = 1 to k
    C[i] ← C[i] + C[i-1]
for j = n to 1
    B[C[X[j]]] ← X[j]
    C[X[j]] ← C[X[j]] - 1
return B
```



X								
	4	2	0	6	2	0	3	2

	0	1	2	3	4	5	6
C	1	2	3	5	7	7	8

B								
		0		2	2	3		

Input Enhancement

Sorting by Counting

- given an array of n orderable items [a_1, a_2,…, a_n], reorder the items as [a_1', a_2',…, a_n'] such that $a_1' \le a_2' \le \ldots \le a_n'$

CountingSort-2(X[1,n],k)

```
let C[1, k] and B[1, n] be new arrays
for i = 0 to k
    C[i] ← 0
for j = 1 to n
    C[X[j]] ← C[X[j]] + 1
for i = 1 to k
    C[i] ← C[i] + C[i-1]
for j = n to 1
    B[C[X[j]]] ← X[j]
    C[X[j]] ← C[X[j]] - 1
return B
```

X

4	2	0	6	2	0	3	2

C

0	1	2	3	4	5	6
1	2	3	5	7	7	7

B

	0		2	2	3		6

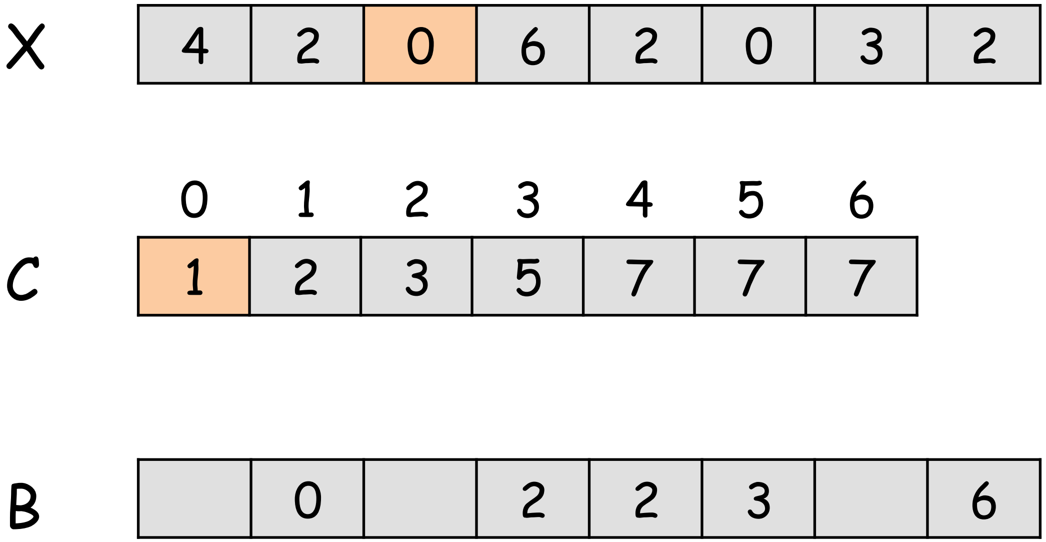
Input Enhancement

Sorting by Counting

- given an array of n orderable items $[a_1, a_2, \dots, a_n]$, reorder the items as $[a_1', a_2', \dots, a_n']$ such that $a_1' \le a_2' \le \dots \le a_n'$

CountingSort-2(X[1,n],k)

```
let C[1, k] and B[1, n] be new arrays
for i = 0 to k
    C[i] ← 0
for j = 1 to n
    C[X[j]] ← C[X[j]] + 1
for i = 1 to k
    C[i] ← C[i] + C[i-1]
for j = n to 1
    B[C[X[j]]] ← X[j]
    C[X[j]] ← C[X[j]] - 1
return B
```



X	4	2	0	6	2	0	3	2

	0	1	2	3	4	5	6
C	1	2	3	5	7	7	7

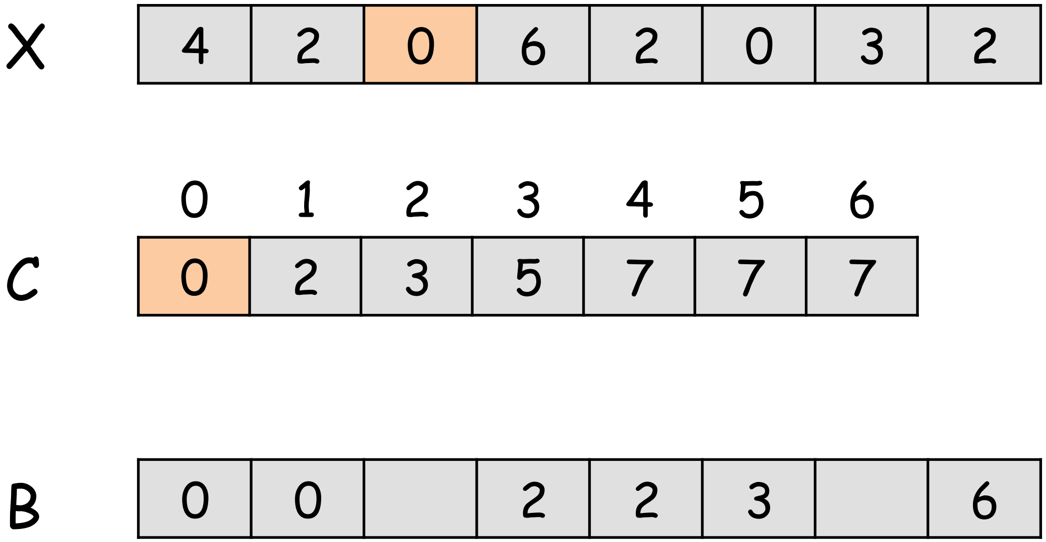
B		0		2	2	3		6

Input Enhancement

Sorting by Counting

- given an array of n orderable items $[a_1, a_2, \ldots, a_n]$, reorder the items as $[a_1', a_2', \ldots, a_n']$ such that $a_1' \le a_2' \le \ldots \le a_n'$

```
CountingSort-2(X[1,n],k)
let C[1, k] and B[1, n] be new arrays
for i = 0 to k
    C[i] ← 0
for j = 1 to n
    C[X[j]] ← C[X[j]] + 1
for i = 1 to k
    C[i] ← C[i] + C[i-1]
for j = n to 1
    B[C[X[j]]] ← X[j]
    C[X[j]] ← C[X[j]] - 1
return B
```



X

4	2	0	6	2	0	3	2

C

0	1	2	3	4	5	6
0	2	3	5	7	7	7

B

0	0		2	2	3		6

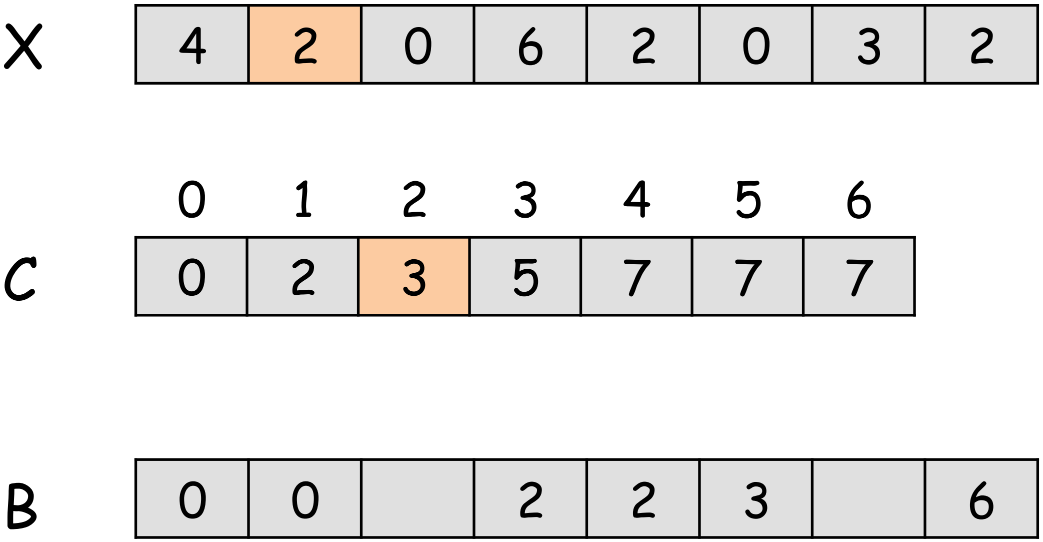
Input Enhancement

Sorting by Counting

- given an array of n orderable items $[a_1, a_2, \dots, a_n]$, reorder the items as $[a_1', a_2', \dots, a_n']$ such that $a_1' \le a_2' \le \dots \le a_n'$

CountingSort-2(X[1,n],k)

```
let C[1, k] and B[1, n] be new arrays
for i = 0 to k
    C[i] ← 0
for j = 1 to n
    C[X[j]] ← C[X[j]] + 1
for i = 1 to k
    C[i] ← C[i] + C[i-1]
for j = n to 1
    B[C[X[j]]] ← X[j]
    C[X[j]] ← C[X[j]] - 1
return B
```



X

4	2	0	6	2	0	3	2

C

0	1	2	3	4	5	6
0	2	3	5	7	7	7

B

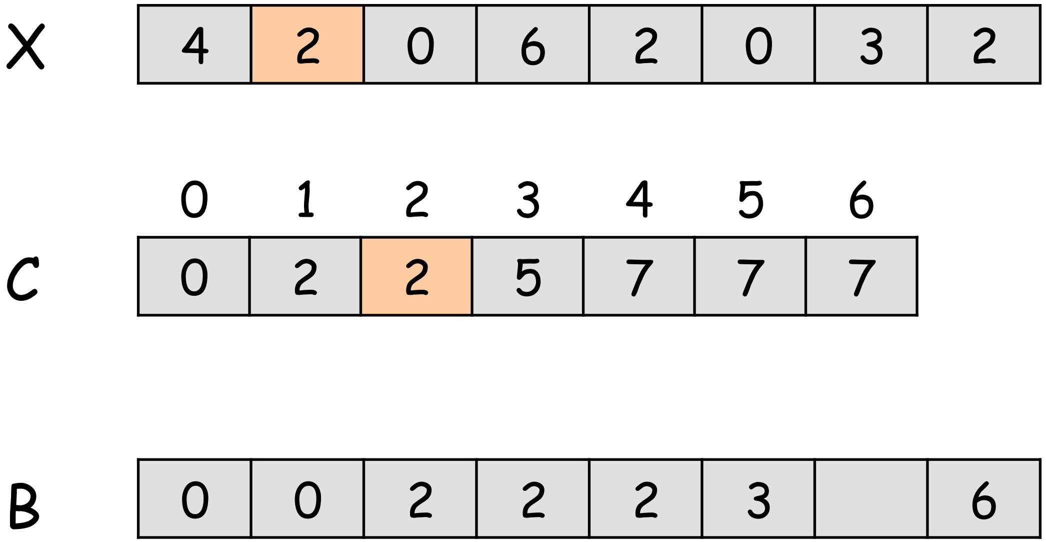
0	0		2	2	3		6

Input Enhancement

Sorting by Counting

- given an array of n orderable items $[a_1, a_2, \ldots, a_n]$, reorder the items as $[a_1', a_2', \ldots, a_n']$ such that $a_1' \leq a_2' \leq \ldots \leq a_n'$

```
CountingSort-2(X[1,n],k)
let C[1, k] and B[1, n] be new arrays
for i = 0 to k
    C[i] ← 0
for j = 1 to n
    C[X[j]] ← C[X[j]] + 1
for i = 1 to k
    C[i] ← C[i] + C[i-1]
for j = n to 1
    B[C[X[j]]] ← X[j]
    C[X[j]] ← C[X[j]] - 1
return B
```



X	4	2	0	6	2	0	3	2

	0	1	2	3	4	5	6
C	0	2	2	5	7	7	7

B	0	0	2	2	2	3		6

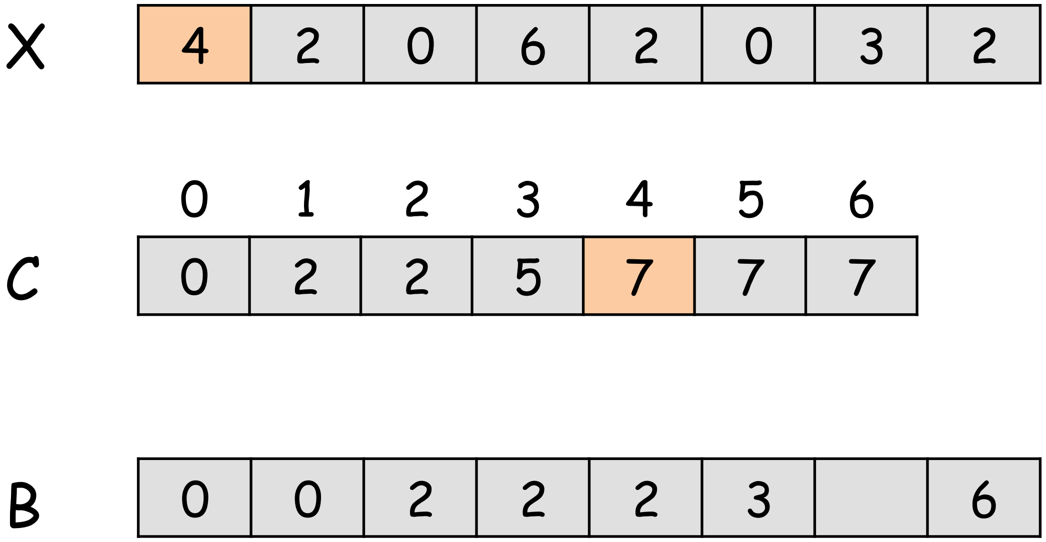
Input Enhancement

Sorting by Counting

- given an array of n orderable items [a_1, a_2,…, a_n], reorder the items as [a_1', a_2',…, a_n'] such that $a_1' \le a_2' \le … \le a_n'$

CountingSort-2(X[1,n],k)

```
let C[1, k] and B[1, n] be new arrays
for i = 0 to k
    C[i] ← 0
for j = 1 to n
    C[X[j]] ← C[X[j]] + 1
for i = 1 to k
    C[i] ← C[i] + C[i-1]
for j = n to 1
    B[C[X[j]]] ← X[j]
    C[X[j]] ← C[X[j]] - 1
return B
```



X	4	2	0	6	2	0	3	2

	0	1	2	3	4	5	6
C	0	2	2	5	7	7	7

B	0	0	2	2	2	3		6

Input Enhancement

Sorting by Counting

- given an array of n orderable items $[a_1, a_2, \ldots, a_n]$, reorder the items as $[a_1', a_2', \ldots, a_n']$ such that $a_1' \le a_2' \le \ldots \le a_n'$

```
CountingSort-2(X[1,n],k)
let C[1, k] and B[1, n] be new arrays
for i = 0 to k
    C[i] ← 0
for j = 1 to n
    C[X[j]] ← C[X[j]] + 1
for i = 1 to k
    C[i] ← C[i] + C[i-1]
for j = n to 1
    B[C[X[j]]] ← X[j]
    C[X[j]] ← C[X[j]] - 1
return B
```

X	4	2	0	6	2	0	3	2

	0	1	2	3	4	5	6
C	0	2	2	5	6	7	7

B	0	0	2	2	2	3	4	6

Input Enhancement

Sorting by Counting

- given an array of n orderable items $[a_1, a_2,\dots, a_n]$, reorder the items as $[a_1', a_2',\dots, a_n']$ such that $a_1' \le a_2' \le \dots \le a_n'$

```
CountingSort-2(X[1,n],k)
let C[1, k] and B[1, n] be new arrays
for i = 0 to k
    C[i] ← 0
for j = 1 to n
    C[X[j]] ← C[X[j]] + 1
for i = 1 to k
    C[i] ← C[i] + C[i-1]
for j = n to 1
    B[C[X[j]]] ← X[j]
    C[X[j]] ← C[X[j]] - 1
return B
```

Input Enhancement

Sorting by Counting

- given an array of n orderable items $[a_1, a_2, \ldots, a_n]$, reorder the items as $[a_1', a_2', \ldots, a_n']$ such that $a_1' \le a_2' \le \ldots \le a_n'$

CountingSort-2(X[1,n],k)

```
let C[1, k] and B[1, n] be new arrays
for i = 0 to k
    C[i] ← 0                          → O(k)
for j = 1 to n
    C[X[j]] ← C[X[j]] + 1             → O(n)
for i = 1 to k
    C[i] ← C[i] + C[i-1]              → O(k)
for j = n to 1
    B[C[X[j]]] ← X[j]
    C[X[j]] ← C[X[j]] - 1             → O(n)
return B
```

Input Enhancement

Sorting by Counting

- given an array of n orderable items [a_1, a_2,…, a_n], reorder the items as [a_1', a_2',…, a_n'] such that $a_1' \leq a_2' \leq \ldots \leq a_n'$

CountingSort-2(X[1,n],k)

```
let C[1, k] and B[1, n] be new arrays
for i = 0 to k
    C[i] ← 0                          → O(k)
for j = 1 to n
    C[X[j]] ← C[X[j]] + 1             → O(n)
for i = 1 to k
    C[i] ← C[i] + C[i-1]              → O(k)
for j = n to 1
    B[C[X[j]]] ← X[j]
    C[X[j]] ← C[X[j]] – 1             → O(n)
return B
```

time complexity : O(k + n)

Input Enhancement

Sorting by Counting

- given an array of n orderable items $[a_1, a_2,\dots, a_n]$, reorder the items as $[a_1', a_2',\dots, a_n']$ such that $a_1' \leq a_2' \leq \dots \leq a_n'$

```
CountingSort-2(X[1,n],k)
let C[1, k] and B[1, n] be new arrays
for i = 0 to k
    C[i] ← 0                        → O(k)
for j = 1 to n
    C[X[j]] ← C[X[j]] + 1           → O(n)
for i = 1 to k
    C[i] ← C[i] + C[i-1]            → O(k)
for j = n to 1
    B[C[X[j]]] ← X[j]
    C[X[j]] ← C[X[j]] – 1           → O(n)
return B
```

time complexity : $O(k + n)$

space complexity : $O(k + n)$

Input Enhancement

String Matching

- given a string of n characters (text) and a string of m characters (pattern), determine whether the text has a substring that matches the pattern

```
StringMatching($T = t_1t_2 \ldots t_n$, $P = p_1p_2 \ldots p_m$)

for i = 1 to n – m + 1
    j ← 1
    while j < m and $p_j = t_{i+j}$
        j ← j+1
    if j = m
        return i
return 0
```

text : FEDERICOFELLINI

ERIC

Input Enhancement

String Matching

- given a string of n characters (text) and a string of m characters (pattern), determine whether the text has a substring that matches the pattern

```
StringMatching($T = t_1t_2...t_n$, $P = p_1p_2...p_m$)

for $i = 1$ to $n - m + 1$
    $j \leftarrow 1$
    while $j < m$ and $p_j = t_{i+j}$
        $j \leftarrow j+1$
    if $j = m$
        return i
return 0
```

text : FEDERICOFELLINI

ERIC

- when a mismatch occurs, shift the pattern one position to right

Input Enhancement

String Matching

- given a string of n characters (text) and a string of m characters (pattern), determine whether the text has a substring that matches the pattern

```
StringMatching($T = t_1t_2...t_n$, $P = p_1p_2...p_m$)

for i = 1 to n - m + 1
    j ← 1
    while $j < m$ and $p_j = t_{i+j}$
        j ← j+1
    if j = m
        return i
return 0
```

text : F E D E R I C O F E L L I N I

E R I C

- when a mismatch occurs, shift the pattern one position to right

Input Enhancement

String Matching

- given a string of n characters (text) and a string of m characters (pattern), determine whether the text has a substring that matches the pattern

```
StringMatching(T = $t_1 t_2 \ldots t_n$, P = $p_1 p_2 \ldots p_m$)

for i = 1 to n - m + 1
    j ← 1
    while j < m and $p_j = t_{i+j}$
        j ← j+1
    if j = m
        return i
return 0
```

text : FEDERICOFELLINI

ERIC

- when a mismatch occurs, shift the pattern one position to right

Input Enhancement

String Matching

- given a string of n characters (text) and a string of m characters (pattern), determine whether the text has a substring that matches the pattern

StringMatching($T = t_1t_2…t_n$, $P = p_1p_2…p_m$)

```
for $i = 1$ to $n - m + 1$
    $j \leftarrow 1$
    while $j < m$ and $p_j = t_{i+j}$
        $j \leftarrow j+1$
    if $j = m$
        return $i$
return 0
```

text : F E D E R I C O F E L L I N I

E R I C

- when a mismatch occurs, shift the pattern one position to right

Input Enhancement

String Matching

- given a string of n characters (text) and a string of m characters (pattern), determine whether the text has a substring that matches the pattern

StringMatching($T = t_1t_2 \ldots t_n, P = p_1p_2 \ldots p_m$)

```
for $i = 1$ to $n - m + 1$
    $j \leftarrow 1$
    while $j < m$ and $p_j = t_{i+j}$
        $j \leftarrow j+1$
    if $j = m$
        return $i$
return 0
```

```
text : F E D E R I C O F E L L I N I
         E R I C
```

- when a mismatch occurs, shift the pattern one position to right

Input Enhancement

String Matching

- given a string of n characters (text) and a string of m characters (pattern), determine whether the text has a substring that matches the pattern

```
StringMatching($T = t_1 t_2 \ldots t_n$, $P = p_1 p_2 \ldots p_m$)

for i = 1 to n - m + 1
    j ← 1
    while $j < m$ and $p_j = t_{i+j}$
        j ← j+1
    if j = m
        return i
return 0
```

```
text : F E D E R I C O F E L L I N I
           E R I C
```

- when a mismatch occurs, shift the pattern one position to right

Input Enhancement

String Matching

- given a string of n characters (text) and a string of m characters (pattern), determine whether the text has a substring that matches the pattern

```
StringMatching(T = $t_1t_2…t_n$, P = $p_1p_2…p_m$)

for i = 1 to n - m + 1
    j ← 1
    while j < m and $p_j = t_{i+j}$
        j ← j+1
    if j = m
        return i
return 0
```

text : F E D E R I C O F E L L I N I

E R I C

- when a mismatch occurs, shift the pattern one position to right

Input Enhancement

String Matching

- given a string of n characters (text) and a string of m characters (pattern), determine whether the text has a substring that matches the pattern

```
StringMatching(T = $t_1t_2…t_n$, P = $p_1p_2…p_m$)

for i = 1 to n – m + 1
    j ← 1
    while j < m and $p_j = t_{i+j}$
        j ← j+1
    if j = m
        return i
return 0
```

text : F E D E R I C O F E L L I N I

E R I C

- when a mismatch occurs, shift the pattern one position to right

Input Enhancement

String Matching

- given a string of n characters (text) and a string of m characters (pattern), determine whether the text has a substring that matches the pattern

```
StringMatching($T = t_1t_2\ldots t_n$, $P = p_1p_2\ldots p_m$)

for $i = 1$ to $n - m + 1$
    $j \leftarrow 1$
    while $j < m$ and $p_j = t_{i+j}$
        $j \leftarrow j+1$
    if $j = m$
        return $i$
return 0
```

text : FEDERICOFELLINI

ERIC

- when a mismatch occurs, shift the pattern one position to right

Input Enhancement

String Matching

- given a string of n characters (text) and a string of m characters (pattern), determine whether the text has a substring that matches the pattern

```
StringMatching($T = t_1t_2...t_n$, $P = p_1p_2...p_m$)

for i = 1 to n – m + 1
    j ← 1
    while j < m and $p_j = t_{i+j}$
        j ← j+1
    if j = m
        return i
return 0
```

text : F E D E R I C O F E L L I N I

E R I C

- when a mismatch occurs, shift the pattern one position to right

- worst-case : O(nm)
- average-case : O(n + m)
  (for random natural-language texts, just a few comparisons expected before a shift)

Input Enhancement

String Matching

- given a string of n characters (text) and a string of m characters (pattern), determine whether the text has a substring that matches the pattern

StringMatching($T = t_1t_2…t_n$, $P = p_1p_2…p_m$)

```
for i = 1 to n − m + 1
    j ← 1
    while j ≤ m and p = t
        j
    if j =
        ret
return 0
```

text : F E D E R I C O F E L L I N I

E R I C

shift the

ght

- if a mismatch occurs, make a shift to right as large as possible

- worst-case : O(nm)
- average-case : O(n + m)
  (for random natural-language texts, just a few comparisons expected before a shift)

Input Enhancement

String Matching

- given a string of n characters (text) and a string of m characters (pattern), determine whether the text has a substring that matches the pattern

StringMatching($T = t_1t_2…t_n$, $P = p_1p_2…p_m$)

```
for i = 1 to n – m + 1
    j ← 1
    while j
        j
    if j =
        ret
return 0
```

text : F E D E R I C O F E L L I N I

- if a mismatch occurs, make a shift to right as large as possible
- there is a risk that you can miss a matching substring when you shift too much

shift the
ght

- worst-case : O(nm)
- average-case : O(n + m)
  (for random natural-language texts, just a few comparisons expected before a shift)

Input Enhancement

String Matching

- given a string of n characters (text) and a string of m characters (pattern), determine whether the text has a substring that matches the pattern

```
StringMatching($T = t_1t_2 \ldots t_n$, $P = p_1p_2 \ldots p_m$)

for i = 1 to n – m + 1
    j ← 1
    while
        j
    if j =
        ret
return 0
```

text : FEDERICOFELLINI

shift the

ght

- if a mismatch occurs, make a shift to right as large as possible
- there is a risk that you can miss a matching substring when you shift too much
- How do we determine the size of such shift?

- average-case : O(n + m)
  (for random natural-language texts, just a few comparisons expected before a shift)

Input Enhancement

Horspool's Algorithm

- use the pattern to make a 'Bad Match Table' that stores shift sizes of all possible characters that can be encountered in the text

Input Enhancement

Horspool's Algorithm

- use the pattern to make a 'Bad Match Table' that stores shift sizes of all possible characters that can be encountered in the text
- compare pattern with text, starting from the rightmost character in the pattern

Input Enhancement

Horspool's Algorithm

- use the pattern to make a 'Bad Match Table' that stores shift sizes of all possible characters that can be encountered in the text
- compare pattern with text, starting from the rightmost character in the pattern
- if a mismatch occurs, shift the pattern to right corresponding to the value in the table

Input Enhancement

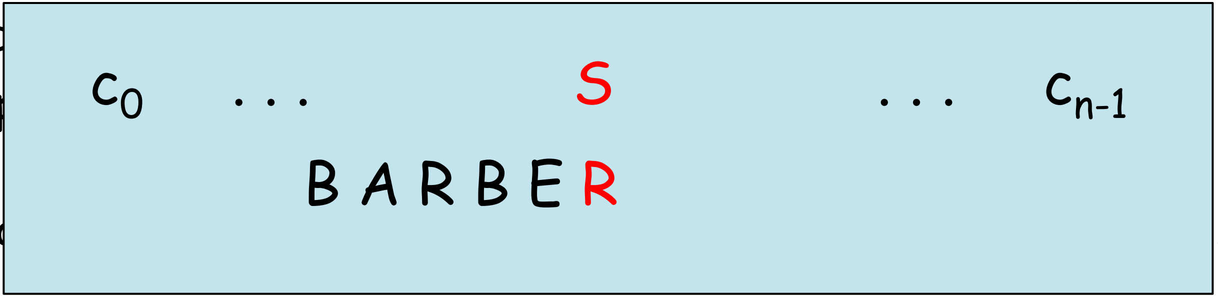
Horspool's Algorithm

- use the pattern to make a 'Bad Match Table' that stores shift sizes of all possible characters that can be encountered in the text
- compare pattern with text, starting from the rightmost character in the pattern
- if a mismatch occurs, shift the pattern to right corresponding to the value in the table
- first, let's analyze the following cases (to determine the shift size, we look at the character that is aligned against the last character of the pattern) :

Input Enhancement

Horspool's Algorithm

- use the pattern to make a 'Bad Match Table' that stores shift sizes of all possible c
- compare p in the pattern
- if a mism he value in the table

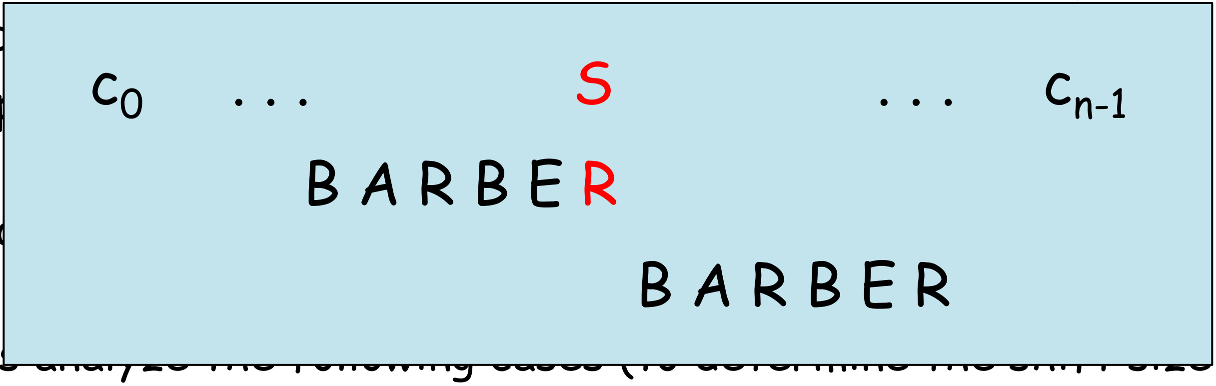


- first, let's analyze the following cases (to determine the shift size, we look at the character that is aligned against the last character of the pattern) :
  - if there is no such character in the pattern, we can safely shift the pattern by its entire length

Input Enhancement

Horspool's Algorithm

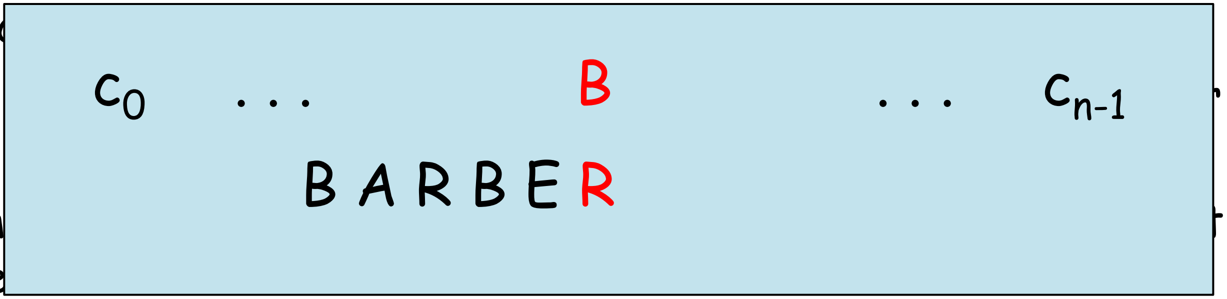
- use the pattern to make a 'Bad Match Table' that stores shift sizes of all possible c
- compare p in the pattern
- if a mism he value in the table
- first, let's analyze the following cases (to determine the shift size, we look at the character that is aligned against the last character of the pattern) :
  - if there is no such character in the pattern, we can safely shift the pattern by its entire length



Input Enhancement

Horspool's Algorithm

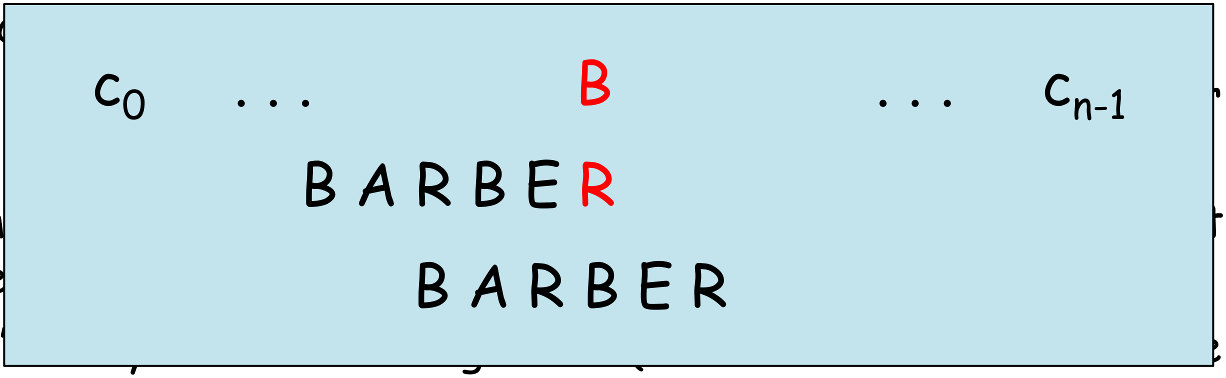
- use the pattern to make a 'Bad Match Table' that stores shift sizes of all possible
- compare … in the pattern
- if a mism… the value in the table



- first, let's analyze the following cases (to determine the shift size, we look at the character that is aligned against the last character of the pattern) :
  - if there is no such character in the pattern, we can safely shift the pattern by its entire length
  - if the character contained in the pattern but it is not the last one, the shift should align the rightmost occurrence of the character in the pattern with the character in the text

Input Enhancement

Horspool's Algorithm



c_0 . . . B . . . c_{n-1}

B A R B E R

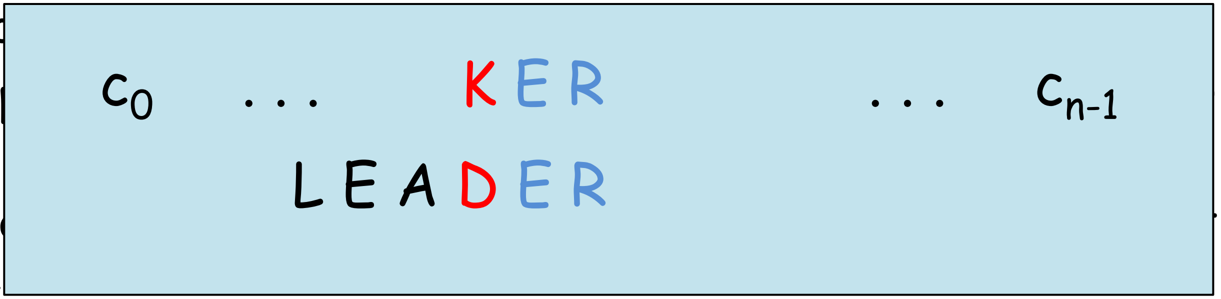
B A R B E R

- use the pattern to make a 'Bad Match Table' that stores shift sizes of all possible
- compare … in the pattern
- if a mism … the value in the table
- first, let' … , we look at the character that is aligned against the last character of the pattern) :
  - if there is no such character in the pattern, we can safely shift the pattern by its entire length
  - if the character contained in the pattern but it is not the last one, the shift should align the rightmost occurrence of the character in the pattern with the character in the text

Input Enhancement

Horspool's Algorithm

- use the pattern to make a 'Bad Match Table' that stores shift sizes of all possible c
- compare p in the pattern
- if a mism he value in the table



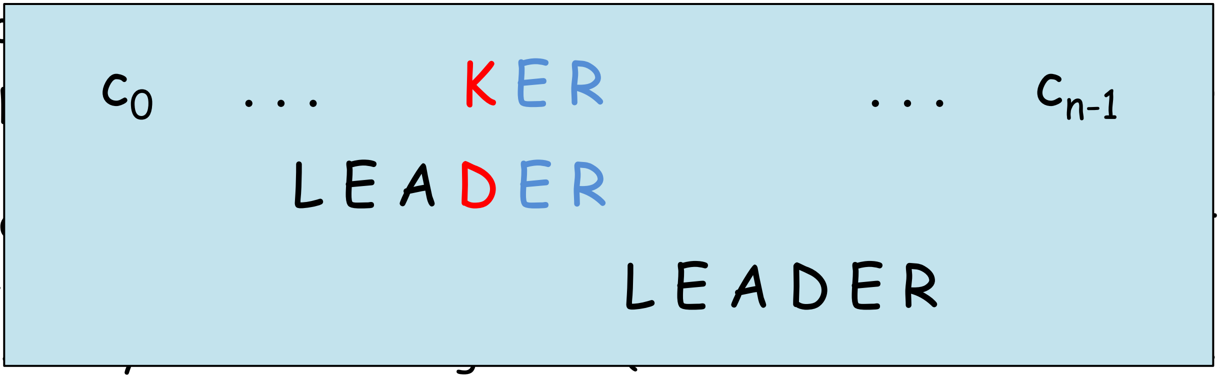
c_0 . . . KER . . . c_{n-1}

LEADER

- first, let's analyze the following cases (to determine the shift size, we look at the character that is aligned against the last character of the pattern) :
  - if there is no such character in the pattern, we can safely shift the pattern by its entire length
  - if the character contained in the pattern but it is not the last one, the shift should align the rightmost occurrence of the character in the pattern with the character in the text
  - if the character equals to the last one in the pattern but there is no same character among others, we can safely shift the pattern by its entire length

Input Enhancement

Horspool's Algorithm



c_0 . . . KER . . . c_{n-1}

LEADER

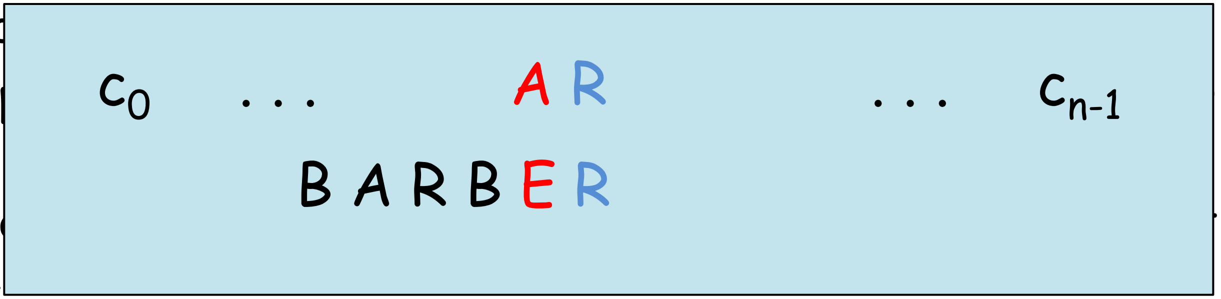
LEADER

- use the pattern to make a 'Bad Match Table' that stores shift sizes of all possible c
- compare p in the pattern
- if a mism he value in the table
- first, let' , we look at the character that is aligned against the last character of the pattern) :
  - if there is no such character in the pattern, we can safely shift the pattern by its entire length
  - if the character contained in the pattern but it is not the last one, the shift should align the rightmost occurrence of the character in the pattern with the character in the text
  - if the character equals to the last one in the pattern but there is no same character among others, we can safely shift the pattern by its entire length

Input Enhancement

Horspool's Algorithm

- use the pattern to make a 'Bad Match Table' that stores shift sizes of all possible c
- compare p in the pattern
- if a mism he value in the table



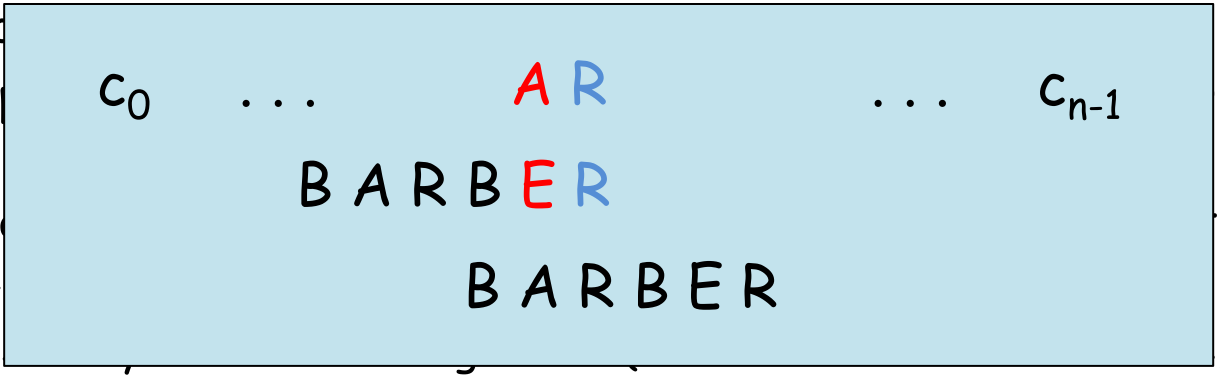
c_0 . . . A R . . . c_{n-1}

B A R B E R

- first, let's analyze the following cases (to determine the shift size, we look at the character that is aligned against the last character of the pattern) :
  - if there is no such character in the pattern, we can safely shift the pattern by its entire length
  - if the character contained in the pattern but it is not the last one, the shift should align the rightmost occurrence of the character in the pattern with the character in the text
  - if the character equals to the last one in the pattern but there is no same character among others, we can safely shift the pattern by its entire length
  - if the character is the last one in the pattern and the character also equals some other character in the pattern, the shift should align the rightmost occurrence of the character in the pattern with the character in the text

Input Enhancement

Horspool's Algorithm



- use the pattern to make a 'Bad Match Table' that stores shift sizes of all possible c
- compare p in the pattern
- if a mism he value in the table
- first, let' , we look at the character that is aligned against the last character of the pattern) :
  - if there is no such character in the pattern, we can safely shift the pattern by its entire length
  - if the character contained in the pattern but it is not the last one, the shift should align the rightmost occurrence of the character in the pattern with the character in the text
  - if the character equals to the last one in the pattern but there is no same character among others, we can safely shift the pattern by its entire length
  - if the character is the last one in the pattern and the character also equals some other character in the pattern, the shift should align the rightmost occurrence of the character in the pattern with the character in the text

Input Enhancement

Horspool's Algorithm

$$\text{Table}(C) = \begin{cases} \text{if } C \text{ is not among the first m – 1 characters of the pattern,} \\ \text{return m} \\ \text{otherwise, return the distance from the rightmost } C \text{ among} \\ \text{the first m – 1 characters of the pattern to its last character} \end{cases}$$

Input Enhancement

Horspool's Algorithm

Table(*C*) =

- if *C* is not among the first m – 1 characters of the pattern, return m
- otherwise, return the distance from the rightmost *C* among the first m – 1 characters of the pattern to its last character

BadMatchTable(P[0,m-1])

input *: a pattern and an alphabet of possible characters*
output *: a bad match table whose equals to the size of the alphabet*

```
for i = 0 to s – 1
    Table[i] 🡐 m
for j = 0 to m – 2
    Table[P[j]] 🡐 m – 1 – j
return Table
```

Input Enhancement

Horspool's Algorithm

Table(*C*) =

- if *C* is not among the first m – 1 characters of the pattern, return m
- otherwise, return the distance from the rightmost *C* among the first m – 1 characters of the pattern to its last character

BadMatchTable(P[0,m-1])

input *: a pattern and an alphabet of possible characters*
output *: a bad match table whose equals to the size of the alphabet*

```
for i = 0 to s – 1
    Table[i] ← m
for j = 0 to m – 2
    Table[P[j]] ← m – 1 – j
return Table
```

- assume the pattern is BARBER and the alphabet is
  Σ = { A, B, …, Z, _ }

Input Enhancement

Horspool's Algorithm

$$\text{Table}(C) = \begin{cases} \text{if } C \text{ is not among the first } m - 1 \text{ characters of the pattern, return } m \\ \text{otherwise, return the distance from the rightmost } C \text{ among the first } m - 1 \text{ characters of the pattern to its last character} \end{cases}$$

BadMatchTable(P[0,m-1])

input *: a pattern and an alphabet of possible characters*
output *: a bad match table whose equals to the size of the alphabet*

```
for i = 0 to s – 1
    Table[i] ← m
for j = 0 to m – 2
    Table[P[j]] ← m – 1 – j
return Table
```

- assume the pattern is BARBER and the alphabet is
  $\Sigma = \{ A, B, \ldots, Z, \_ \}$

	A	B	E	R	*
Table(C)					

Input Enhancement

Horspool’s Algorithm

Table(*C*) =

- if *C* is not among the first m – 1 characters of the pattern, return m
- otherwise, return the distance from the rightmost *C* among the first m – 1 characters of the pattern to its last character

BadMatchTable(P[0,m-1])

input *: a pattern and an alphabet of possible characters*
output *: a bad match table whose equals to the size of the alphabet*

```
for i = 0 to s - 1
    Table[i] ← m
for j = 0 to m - 2
    Table[P[j]] ← m - 1 - j
return Table
```

- assume the pattern is BARBER and the alphabet is
  $\Sigma = \{ A, B, \ldots, Z, \_ \}$

	A	B	E	R	*
Table(C)	6	6	6	6	6

Input Enhancement

Horspool's Algorithm

$$\text{Table}(C) = \begin{cases} \text{if } C \text{ is not among the first } m - 1 \text{ characters of the pattern, return } m \\ \text{otherwise, return the distance from the rightmost } C \text{ among the first } m - 1 \text{ characters of the pattern to its last character} \end{cases}$$

BadMatchTable(P[0,m-1])

input *: a pattern and an alphabet of possible characters*
output *: a bad match table whose equals to the size of the alphabet*

```
for i = 0 to s – 1
    Table[i] ← m
for j = 0 to m – 2
    Table[P[j]] ← m – 1 – j
return Table
```

- assume the pattern is BARBER and the alphabet is $\Sigma = \{ A, B, \dots, Z, \_ \}$

	A	B	E	R	*
Table(C)	6	5	6	6	6

Input Enhancement

Horspool's Algorithm

$$\text{Table}(C) = \begin{cases} \text{if } C \text{ is not among the first } m - 1 \text{ characters of the pattern, return } m \\ \text{otherwise, return the distance from the rightmost } C \text{ among the first } m - 1 \text{ characters of the pattern to its last character} \end{cases}$$

BadMatchTable(P[0,m-1])

input *: a pattern and an alphabet of possible characters*
output *: a bad match table whose equals to the size of the alphabet*

```
for i = 0 to s – 1
    Table[i] ← m
for j = 0 to m – 2
    Table[P[j]] ← m – 1 – j
return Table
```

- assume the pattern is BARBER and the alphabet is $\Sigma = \{ A, B, \ldots, Z, \_ \}$

	A	B	E	R	*
Table(C)	4	5	6	6	6

Input Enhancement

Horspool's Algorithm

Table(*C*) =

- if *C* is not among the first m – 1 characters of the pattern, return m
- otherwise, return the distance from the rightmost *C* among the first m – 1 characters of the pattern to its last character

BadMatchTable(P[0,m-1])

input *: a pattern and an alphabet of possible characters*
output *: a bad match table whose equals to the size of the alphabet*

```
for i = 0 to s – 1
    Table[i] ← m
for j = 0 to m – 2
    Table[P[j]] ← m – 1 – j
return Table
```

- assume the pattern is BARBER and the alphabet is
  $\Sigma = \{ A, B, \ldots, Z, \_ \}$

	A	B	E	R	*
Table(C)	4	5	6	3	6

Input Enhancement

Horspool's Algorithm

$$\text{Table}(C) = \begin{cases} \text{if } C \text{ is not among the first } m - 1 \text{ characters of the pattern, return } m \\ \text{otherwise, return the distance from the rightmost } C \text{ among the first } m - 1 \text{ characters of the pattern to its last character} \end{cases}$$

BadMatchTable(P[0,m-1])

input *: a pattern and an alphabet of possible characters*
output *: a bad match table whose equals to the size of the alphabet*

```
for i = 0 to s – 1
     Table[i] ← m
for j = 0 to m – 2
     Table[P[j]] ← m – 1 – j
return Table
```

- assume the pattern is BARBER and the alphabet is
  Σ = { A, B, …, Z, _ }

	A	B	E	R	*
Table(C)	4	2	6	3	6

Input Enhancement

Horspool's Algorithm

Table(*C*) =

- if *C* is not among the first m – 1 characters of the pattern, return m
- otherwise, return the distance from the rightmost *C* among the first m – 1 characters of the pattern to its last character

BadMatchTable(P[0,m-1])

input *: a pattern and an alphabet of possible characters*
output *: a bad match table whose equals to the size of the alphabet*

```
for i = 0 to s – 1
      Table[i] ← m
for j = 0 to m – 2
      Table[P[j]] ← m – 1 – j
return Table
```

- assume the pattern is BARBER and the alphabet is
  Σ = { A, B, …, Z, _ }

	A	B	E	R	*
Table(C)	4	2	1	3	6

Input Enhancement

Horspool's Algorithm

```
HorspoolMatching(P[0,m-1], T[0,n-1])

Table[0,s-1] ← BadMatchTable(P[0,m-1])
i ← m - 1
while i ≤ n - 1
        k ← 0
        while k ≤ m-1 and P[m-1-k]=T[i-k]
                k ← k + 1
        if k = m
            return i - m + 1
        else
            i ← i + Table[T[i]]
return -1
```

Input Enhancement

Horspool's Algorithm

```
HorspoolMatching(P[0,m-1], T[0,n-1])

Table[0,s-1] ← BadMatchTable(P[0,m-1])
i ← m - 1
while i ≤ n - 1
        k ← 0
        while k ≤ m-1 and P[m-1-k]=T[i-k]
                k ← k + 1
        if k = m
            return i - m + 1
        else
            i ← i + Table[T[i]]
return -1
```

JIM_SAW_ME_IN_A_BARBERSHOP

BARBER

Input Enhancement

Horspool's Algorithm

```
HorspoolMatching(P[0,m-1], T[0,n-1])

Table[0,s-1] ← BadMatchTable(P[0,m-1])
i ← m - 1
while i ≤ n - 1
        k ← 0
        while k ≤ m-1 and P[m-1-k]=T[i-k]
                k ← k + 1
        if k = m
            return i - m + 1
        else
            i ← i + Table[T[i]]
return -1
```

n = 26 JIM_SAW_ME_IN_A_BARBERSHOP

m = 6 BARBER

Input Enhancement

Horspool's Algorithm

```
HorspoolMatching(P[0,m-1], T[0,n-1])

Table[0,s-1] ← BadMatchTable(P[0,m-1])
i ← m - 1
while i ≤ n - 1
        k ← 0
        while k ≤ m-1 and P[m-1-k]=T[i-k]
                k ← k + 1
        if k = m
            return i - m + 1
        else
            i ← i + Table[T[i]]
return -1
```

	A	B	E	R	*
Table(C)	4	2	1	3	6

n = 26 JIM_SAW_ME_IN_A_BARBERSHOP

m = 6 BARBER

Input Enhancement

Horspool's Algorithm

```
HorspoolMatching(P[0,m-1], T[0,n-1])

Table[0,s-1] ← BadMatchTable(P[0,m-1])
i ← m - 1
while i ≤ n - 1
        k ← 0
        while k ≤ m-1 and P[m-1-k]=T[i-k]
                k ← k + 1
        if k = m
            return i - m + 1
        else
            i ← i + Table[T[i]]
return -1
```

	A	B	E	R	*
Table(C)	4	2	1	3	6

n = 26 JIM_SAW_ME_IN_A_BARBERSHOP

m = 6 BARBER

i = 5 k = 0

Input Enhancement

Horspool's Algorithm

```
HorspoolMatching(P[0,m-1], T[0,n-1])

Table[0,s-1] ← BadMatchTable(P[0,m-1])
i ← m - 1
while i ≤ n - 1
        k ← 0
        while k ≤ m-1 and P[m-1-k]=T[i-k]
                k ← k + 1
        if k = m
            return i - m + 1
        else
            i ← i + Table[T[i]]
return -1
```

	A	B	E	R	*
Table(C)	4	2	1	3	6

n = 26　　JIM_SAW_ME_IN_A_BARBERSHOP

m = 6　　BARBER

i = 5　　k = 0

Input Enhancement

Horspool's Algorithm

```
HorspoolMatching(P[0,m-1], T[0,n-1])

Table[0,s-1] ← BadMatchTable(P[0,m-1])
i ← m - 1
while i ≤ n - 1
        k ← 0
        while k ≤ m-1 and P[m-1-k]=T[i-k]
                k ← k + 1
        if k = m
            return i - m + 1
        else
            i ← i + Table[T[i]]
return -1
```

	A	B	E	R	*
Table(C)	4	2	1	3	6

n = 26 JIM_SAW_ME_IN_A_BARBERSHOP

m = 6 BARBER

i = 5 k = 0

Input Enhancement

Horspool's Algorithm

```
HorspoolMatching(P[0,m-1], T[0,n-1])

Table[0,s-1] ← BadMatchTable(P[0,m-1])
i ← m - 1
while i ≤ n - 1
        k ← 0
        while k ≤ m-1 and P[m-1-k]=T[i-k]
                k ← k + 1
        if k = m
            return i - m + 1
        else
            i ← i + Table[T[i]]
return -1
```

	A	B	E	R	*
Table(C)	4	2	1	3	6

n = 26 J I M _ S A W _ M E _ I N _ A _ B A R B E R S H O P

m = 6 B A R B E R

i = 9 k = 0

Input Enhancement

Horspool's Algorithm

```
HorspoolMatching(P[0,m-1], T[0,n-1])

Table[0,s-1] ← BadMatchTable(P[0,m-1])
i ← m - 1
while i ≤ n - 1
        k ← 0
        while k ≤ m-1 and P[m-1-k]=T[i-k]
                k ← k + 1
        if k = m
            return i - m + 1
        else
            i ← i + Table[T[i]]
return -1
```

	A	B	E	R	*
Table(C)	4	2	1	3	6

n = 26 JIM_SAW_ME_IN_A_BARBERSHOP

m = 6 BARBER

i = 9 k = 0

Input Enhancement

Horspool's Algorithm

```
HorspoolMatching(P[0,m-1], T[0,n-1])

Table[0,s-1] ← BadMatchTable(P[0,m-1])
i ← m - 1
while i ≤ n - 1
        k ← 0
        while k ≤ m-1 and P[m-1-k]=T[i-k]
                k ← k + 1
        if k = m
            return i - m + 1
        else
            i ← i + Table[T[i]]
return -1
```

	A	B	E	R	*
Table(C)	4	2	1	3	6

n = 26 JIM_SAW_ME_IN_A_BARBERSHOP

m = 6 BARBER

i = 9 k = 0

Input Enhancement

Horspool's Algorithm

```
HorspoolMatching(P[0,m-1], T[0,n-1])

Table[0,s-1] ← BadMatchTable(P[0,m-1])
i ← m - 1
while i ≤ n - 1
        k ← 0
        while k ≤ m-1 and P[m-1-k]=T[i-k]
                k ← k + 1
        if k = m
            return i - m + 1
        else
            i ← i + Table[T[i]]
return -1
```

	A	B	E	R	*
Table(C)	4	2	1	3	6

n = 26 JIM_SAW_ME_IN_A_BARBERSHOP

m = 6 BARBER

i = 10 k = 0

Input Enhancement

Horspool's Algorithm

```
HorspoolMatching(P[0,m-1], T[0,n-1])

Table[0,s-1] ← BadMatchTable(P[0,m-1])
i ← m - 1
while i ≤ n - 1
        k ← 0
        while k ≤ m-1 and P[m-1-k]=T[i-k]
                k ← k + 1
        if k = m
            return i - m + 1
        else
            i ← i + Table[T[i]]
return -1
```

	A	B	E	R	*
Table(C)	4	2	1	3	6

n = 26 JIM_SAW_ME_IN_A_BARBERSHOP

m = 6 BARBER

i = 10 k = 0

Input Enhancement

Horspool's Algorithm

```
HorspoolMatching(P[0,m-1], T[0,n-1])

Table[0,s-1] ← BadMatchTable(P[0,m-1])
i ← m - 1
while i ≤ n - 1
        k ← 0
        while k ≤ m-1 and P[m-1-k]=T[i-k]
                k ← k + 1
        if k = m
            return i - m + 1
        else
            i ← i + Table[T[i]]
return -1
```

	A	B	E	R	*
Table(C)	4	2	1	3	6

n = 26 JIM_SAW_ME_IN_A_BARBERSHOP

m = 6 BARBER

i = 10 k = 0

Input Enhancement

Horspool's Algorithm

```
HorspoolMatching(P[0,m-1], T[0,n-1])

Table[0,s-1] ← BadMatchTable(P[0,m-1])
i ← m - 1
while i ≤ n - 1
        k ← 0
        while k ≤ m-1 and P[m-1-k]=T[i-k]
                k ← k + 1
        if k = m
            return i - m + 1
        else
            i ← i + Table[T[i]]
return -1
```

	A	B	E	R	*
Table(C)	4	2	1	3	6

n = 26 JIM_SAW_ME_IN_A_BARBERSHOP

m = 6 BARBER

i = 16 k = 0

Input Enhancement

Horspool's Algorithm

```
HorspoolMatching(P[0,m-1], T[0,n-1])

Table[0,s-1] ← BadMatchTable(P[0,m-1])
i ← m - 1
while i ≤ n - 1
        k ← 0
        while k ≤ m-1 and P[m-1-k]=T[i-k]
                k ← k + 1
        if k = m
            return i - m + 1
        else
            i ← i + Table[T[i]]
return -1
```

	A	B	E	R	*
Table(C)	4	2	1	3	6

n = 26   J I M _ S A W _ M E _ I N _ A _ B A R B E R S H O P

m = 6   B A R B E R

i = 16   k = 0

Input Enhancement

Horspool's Algorithm

```
HorspoolMatching(P[0,m-1], T[0,n-1])

Table[0,s-1] ← BadMatchTable(P[0,m-1])
i ← m - 1
while i ≤ n - 1
        k ← 0
        while k ≤ m-1 and P[m-1-k]=T[i-k]
                k ← k + 1
        if k = m
            return i - m + 1
        else
            i ← i + Table[T[i]]
return -1
```

	A	B	E	R	*
Table(C)	4	2	1	3	6

n = 26 J I M _ S A W _ M E _ I N _ A _ B A R B E R S H O P

m = 6 B A R B E R

i = 16 k = 0

Input Enhancement

Horspool's Algorithm

```
HorspoolMatching(P[0,m-1], T[0,n-1])

Table[0,s-1] ← BadMatchTable(P[0,m-1])
i ← m - 1
while i ≤ n - 1
        k ← 0
        while k ≤ m-1 and P[m-1-k]=T[i-k]
                k ← k + 1
        if k = m
            return i - m + 1
        else
            i ← i + Table[T[i]]
return -1
```

	A	B	E	R	*
Table(C)	4	2	1	3	6

n = 26 JIM_SAW_ME_IN_A_BARBERSHOP

m = 6 BARBER

i = 18 k = 0

Input Enhancement

Horspool's Algorithm

```
HorspoolMatching(P[0,m-1], T[0,n-1])

Table[0,s-1] ← BadMatchTable(P[0,m-1])
i ← m - 1
while i ≤ n - 1
        k ← 0
        while k ≤ m-1 and P[m-1-k]=T[i-k]
                k ← k + 1
        if k = m
            return i - m + 1
        else
            i ← i + Table[T[i]]
return -1
```

	A	B	E	R	*
Table(C)	4	2	1	3	6

n = 26 JIM_SAW_ME_IN_A_BARBERSHOP

m = 6 BARBER

i = 18 k = 1

Input Enhancement

Horspool's Algorithm

```
HorspoolMatching(P[0,m-1], T[0,n-1])

Table[0,s-1] ← BadMatchTable(P[0,m-1])
i ← m - 1
while i ≤ n - 1
        k ← 0
        while k ≤ m-1 and P[m-1-k]=T[i-k]
                k ← k + 1
        if k = m
            return i - m + 1
        else
            i ← i + Table[T[i]]
return -1
```

	A	B	E	R	*
Table(C)	4	2	1	3	6

n = 26 J I M _ S A W _ M E _ I N _ A _ B A R B E R S H O P

m = 6 B A R B E R

i = 18 k = 1

Input Enhancement

Horspool's Algorithm

```
HorspoolMatching(P[0,m-1], T[0,n-1])

Table[0,s-1] ← BadMatchTable(P[0,m-1])
i ← m - 1
while i ≤ n - 1
        k ← 0
        while k ≤ m-1 and P[m-1-k]=T[i-k]
                k ← k + 1
        if k = m
            return i - m + 1
        else
            i ← i + Table[T[i]]
return -1
```

	A	B	E	R	*
Table(C)	4	2	1	3	6

n = 26 JIM_SAW_ME_IN_A_BARBERSHOP

m = 6 BARBER

i = 18 k = 1

Input Enhancement

Horspool's Algorithm

```
HorspoolMatching(P[0,m-1], T[0,n-1])

Table[0,s-1] ← BadMatchTable(P[0,m-1])
i ← m - 1
while i ≤ n - 1
        k ← 0
        while k ≤ m-1 and P[m-1-k]=T[i-k]
                k ← k + 1
        if k = m
            return i - m + 1
        else
            i ← i + Table[T[i]]
return -1
```

	A	B	E	R	*
Table(C)	4	2	1	3	6

n = 26 JIM_SAW_ME_IN_A_BARBERSHOP

m = 6 BARBER

i = 21 k = 0

Input Enhancement

Horspool's Algorithm

```
HorspoolMatching(P[0,m-1], T[0,n-1])

Table[0,s-1] ← BadMatchTable(P[0,m-1])
i ← m - 1
while i ≤ n - 1
        k ← 0
        while k ≤ m-1 and P[m-1-k]=T[i-k]
                k ← k + 1
        if k = m
            return i - m + 1
        else
            i ← i + Table[T[i]]
return -1
```

	A	B	E	R	*
Table(C)	4	2	1	3	6

n = 26 JIM_SAW_ME_IN_A_BARBERSHOP

m = 6 BARBER

i = 21 k = 1

Input Enhancement

Horspool's Algorithm

```
HorspoolMatching(P[0,m-1], T[0,n-1])

Table[0,s-1] ← BadMatchTable(P[0,m-1])
i ← m - 1
while i ≤ n - 1
        k ← 0
        while k ≤ m-1 and P[m-1-k]=T[i-k]
                k ← k + 1
        if k = m
            return i - m + 1
        else
            i ← i + Table[T[i]]
return -1
```

	A	B	E	R	*
Table(C)	4	2	1	3	6

n = 26 J I M _ S A W _ M E _ I N _ A _ B A R B E R S H O P

m = 6 B A R B E R

i = 21 k = 2

Input Enhancement

Horspool's Algorithm

```
HorspoolMatching(P[0,m-1], T[0,n-1])

Table[0,s-1] ← BadMatchTable(P[0,m-1])
i ← m - 1
while i ≤ n - 1
        k ← 0
        while k ≤ m-1 and P[m-1-k]=T[i-k]
                k ← k + 1
        if k = m
            return i - m + 1
        else
            i ← i + Table[T[i]]
return -1
```

	A	B	E	R	*
Table(C)	4	2	1	3	6

n = 26 JIM_SAW_ME_IN_A_BARBERSHOP

m = 6 BARBER

i = 21 k = 3

Input Enhancement

Horspool's Algorithm

```
HorspoolMatching(P[0,m-1], T[0,n-1])

Table[0,s-1] ← BadMatchTable(P[0,m-1])
i ← m - 1
while i ≤ n - 1
        k ← 0
        while k ≤ m-1 and P[m-1-k]=T[i-k]
                k ← k + 1
        if k = m
            return i - m + 1
        else
            i ← i + Table[T[i]]
return -1
```

	A	B	E	R	*
Table(C)	4	2	1	3	6

n = 26 JIM_SAW_ME_IN_A_BARBERSHOP

m = 6 BARBER

i = 21 k = 4

Input Enhancement

Horspool's Algorithm

```
HorspoolMatching(P[0,m-1], T[0,n-1])

Table[0,s-1] ← BadMatchTable(P[0,m-1])
i ← m - 1
while i ≤ n - 1
        k ← 0
        while k ≤ m-1 and P[m-1-k]=T[i-k]
                k ← k + 1
        if k = m
            return i - m + 1
        else
            i ← i + Table[T[i]]
return -1
```

	A	B	E	R	*
Table(C)	4	2	1	3	6

n = 26 JIM_SAW_ME_IN_A_BARBERSHOP

m = 6 BARBER

i = 21 k = 5

Input Enhancement

Horspool's Algorithm

```
HorspoolMatching(P[0,m-1], T[0,n-1])

Table[0,s-1] ← BadMatchTable(P[0,m-1])
i ← m - 1
while i ≤ n - 1
        k ← 0
        while k ≤ m-1 and P[m-1-k]=T[i-k]
                k ← k + 1
        if k = m
            return i - m + 1
        else
            i ← i + Table[T[i]]
return -1
```

	A	B	E	R	*
Table(C)	4	2	1	3	6

n = 26   J I M _ S A W _ M E _ I N _ A _ B A R B E R S H O P

m = 6   B A R B E R

i = 21   k = 6

Input Enhancement

Horspool's Algorithm

```
HorspoolMatching(P[0,m-1], T[0,n-1])

Table[0,s-1] ← BadMatchTable(P[0,m-1])
i ← m - 1
while i ≤ n - 1
        k ← 0
        while k ≤ m-1 and P[m-1-k]=T[i-k]
                k ← k + 1
        if k = m
            return i - m + 1
        else
            i ← i + Table[T[i]]
return -1
```

	A	B	E	R	*
Table(C)	4	2	1	3	6

n = 26 J I M _ S A W _ M E _ I N _ A _ B A R B E R S H O P

m = 6 B A R B E R

i = 21 k = 6 index 19

Input Enhancement

Horspool's Algorithm

```
HorspoolMatching(P[0,m-1], T[0,n-1])

Table[0,s-1] ← BadMatchTable(P[0,m-1])
i ← m - 1
while i ≤ n - 1
        k← 0
        while k ≤ m-1 and P[m-1-k]=T[i-k]
                k ← k + 1
        if k = m
            return i - m + 1
        else
            i ← i + Table[T[i]]
return -1
```

- worst-case time complexity :

Input Enhancement

Horspool's Algorithm

```
HorspoolMatching(P[0,m-1], T[0,n-1])

Table[0,s-1] ← BadMatchTable(P[0,m-1])
i ← m - 1
while i ≤ n - 1
        k← 0
        while k ≤ m-1 and P[m-1-k]=T[i-k]
                k ← k + 1
        if k = m
            return i - m + 1
        else
            i ← i + Table[T[i]]
return -1
```

- worst-case time complexity :

text : AAA…AAA (length n)
pattern : BAA…AAA (length m)

Input Enhancement

Horspool's Algorithm

```
HorspoolMatching(P[0,m-1], T[0,n-1])

Table[0,s-1] ← BadMatchTable(P[0,m-1])
i ← m - 1
while i ≤ n - 1
        k← 0
        while k ≤ m-1 and P[m-1-k]=T[i-k]
                k ← k + 1
        if k = m
            return i - m + 1
        else
            i ← i + Table[T[i]]
return -1
```

- worst-case time complexity : O(nm)

text : AAA…AAA (length n)
pattern : BAA…AAA (length m)

Input Enhancement

Horspool's Algorithm

```
HorspoolMatching(P[0,m-1], T[0,n-1])

Table[0,s-1] ← BadMatchTable(P[0,m-1])
i ← m - 1
while i ≤ n - 1
        k ← 0
        while k ≤ m-1 and P[m-1-k]=T[i-k]
                k ← k + 1
        if k = m
            return i - m + 1
        else
            i ← i + Table[T[i]]
return -1
```

- worst-case time complexity : $O(nm)$

    text : AAA…AAA (length n)
    pattern : BAA…AAA (length m)

- average-case time complexity :

    $O(n/\min(m,|\Sigma|)) \approx O(n)$

Input Enhancement

Horspool's Algorithm

```
HorspoolMatching(P[0,m-1], T[0,n-1])

Table[0,s-1] ← BadMatchTable(P[0,m-1])
i ← m - 1
while i ≤ n - 1
        k ← 0
        while k ≤ m-1 and P[m-1-k]=T[i-k]
                k ← k + 1
        if k = m
            return i - m + 1
        else
            i ← i + Table[T[i]]
return -1
```

- worst-case time complexity : O(nm)

  text : AAA…AAA (length n)
  pattern : BAA…AAA (length m)

- average-case time complexity :

  $O(n/\min(m,|\Sigma|)) \approx O(n)$

- space complexity : $O(|\Sigma|)$

Pre-Structuring

Hashing

- a very efficient way of implementing dictionaries,

Pre-Structuring

Hashing

- a very efficient way of implementing dictionaries,

- a dictionary is an abstract data type supporting three operations : searching, insertion, and deletion.
- elements in a dictionary can be of an arbitrary nature: numbers, characters strings of some alphabet, etc.
- each element consists of a number of fields so that each of them keeps a particular type of information
- at least one of fields corresponds to 'key', used to identify the elements dictionaries

Pre-Structuring

Hashing

- a very efficient way of implementing dictionaries
- distributes the elements based their keys among a one-dimensional array H[0,m-1], called Hash Table

Pre-Structuring

Hashing

- a very efficient way of implementing dictionaries
- distributes the elements based their keys among a one-dimensional array H[0,m-1], called Hash Table
- distribution performed through a function, called hash function,

Pre-Structuring

Hashing

- a very efficient way of implementing dictionaries
- distributes the elements based their keys among a one-dimensional array H[0,m-1], called Hash Table
- distribution performed through a function, called hash function, that maps keys of the elements (large data sets) to some value (smaller data set) in [0,m-1], called hash address

Pre-Structuring

Hashing

- a very efficient way of implementing dictionaries
- distributes the elements based their keys among a one-dimensional array H[0,m-1], called Hash Table
- distribution performed through a function, called hash function, that maps keys of the elements (large data sets) to some value (smaller data set) in [0,m-1], called hash address
- the operations 'searching, insertion, and deletion' take constant time in average (when hash table properly implemented)

Pre-Structuring

Hash Function

- A hash table is an array H[0,m-1]

Pre-Structuring

Hash Function

- A hash table is an array H[0,m-1]
- A hash function h then

$$h : U \rightarrow \{0, 1, \ldots, m-1\}$$

(an item x hashes to the slot H[h(x)])

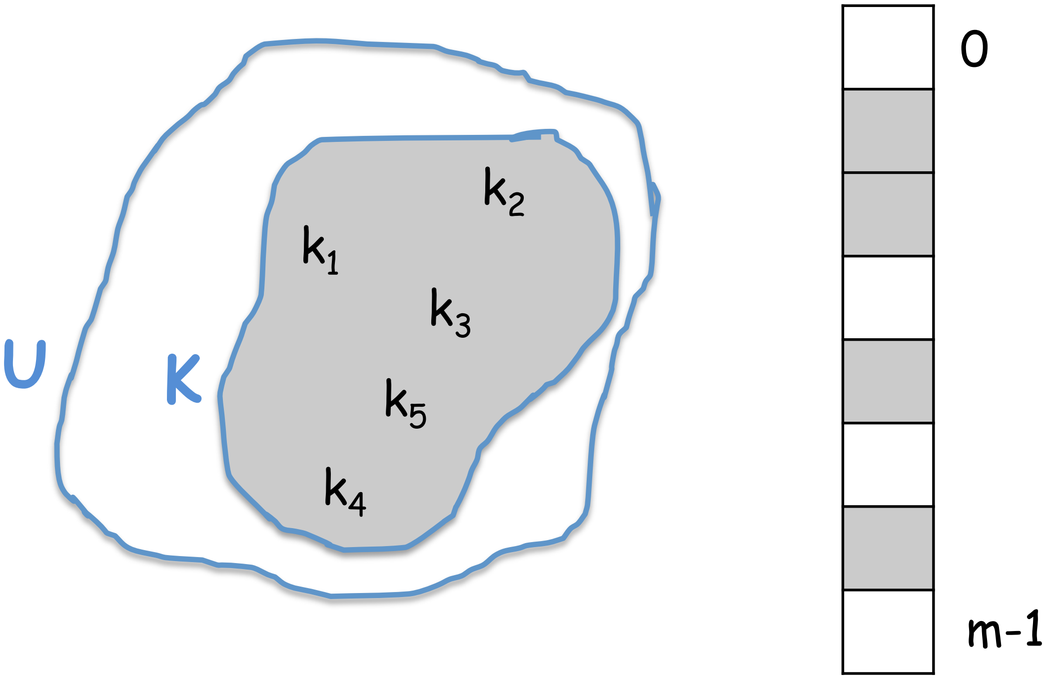
Pre-Structuring

Hash Function

- A hash table is an array H[0,m-1]
- A hash function h then

$$h : U \rightarrow \{0, 1, \ldots, m-1\}$$

(an item x hashes to the slot H[h(x)])



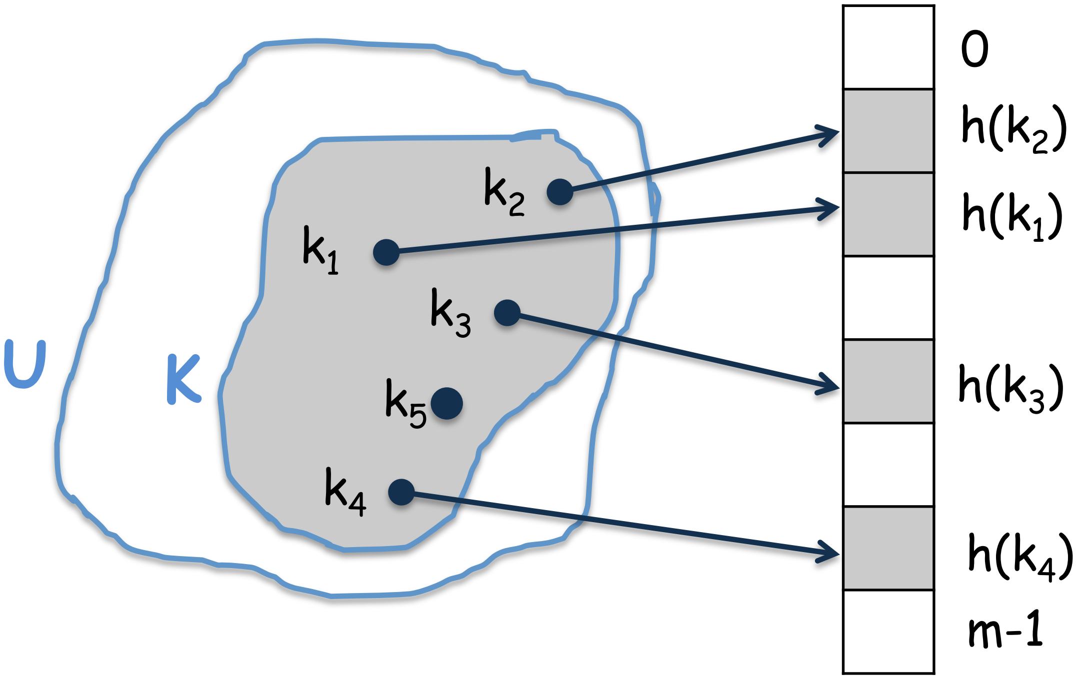
Pre-Structuring

Hash Function

- A hash table is an array H[0,m-1]
- A hash function h then

$$h : U \rightarrow \{0, 1, \ldots, m-1\}$$

(an item x hashes to the slot H[h(x)])



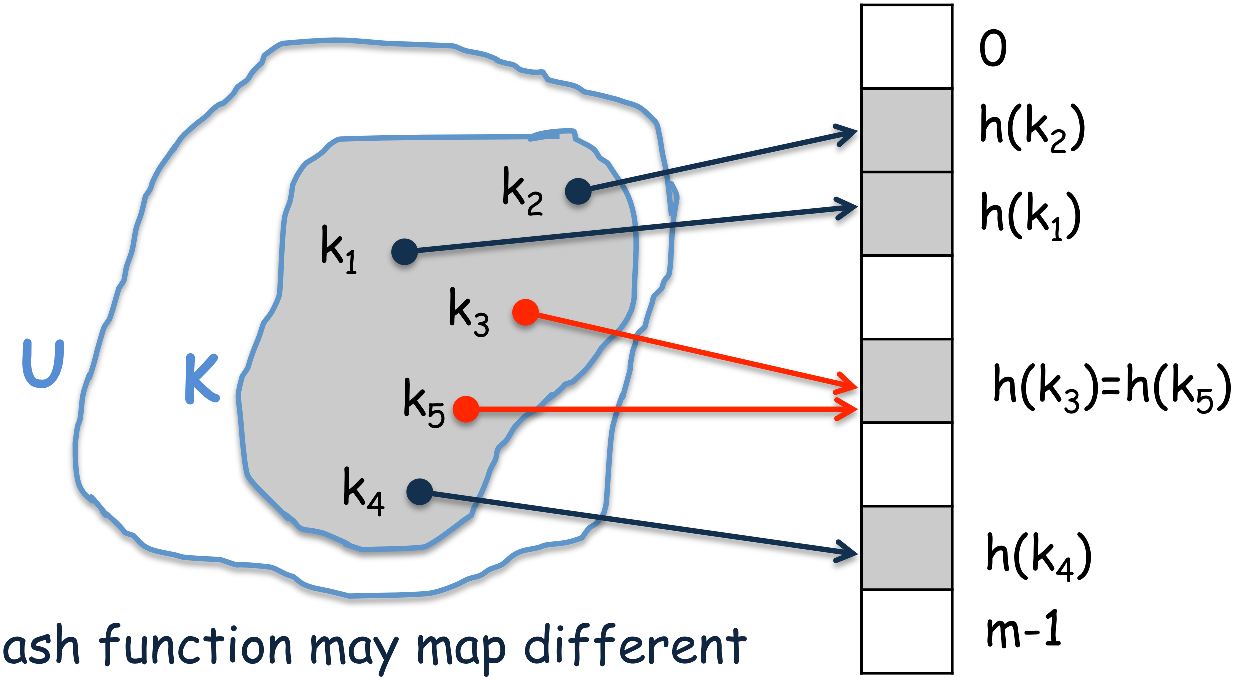
Pre-Structuring

Hash Function

- A hash table is an array H[0,m-1]
- A hash function h then

$$h : U \rightarrow \{0, 1, \ldots, m-1\}$$

(an item x hashes to the slot H[h(x)])



- Collusion : a hash function may map different keys to same slot (many-to-one mapping)

Pre-Structuring

Hash Function

- A hash table is an array H[0,m-1]
- A hash function h then

$$h : U \rightarrow \{0, 1, \ldots, m-1\}$$

(an item x hashes to the slot H[h(x)])

Pre-Structuring

Hash Function

- A hash table is an array H[0,m-1]
- A hash function h then

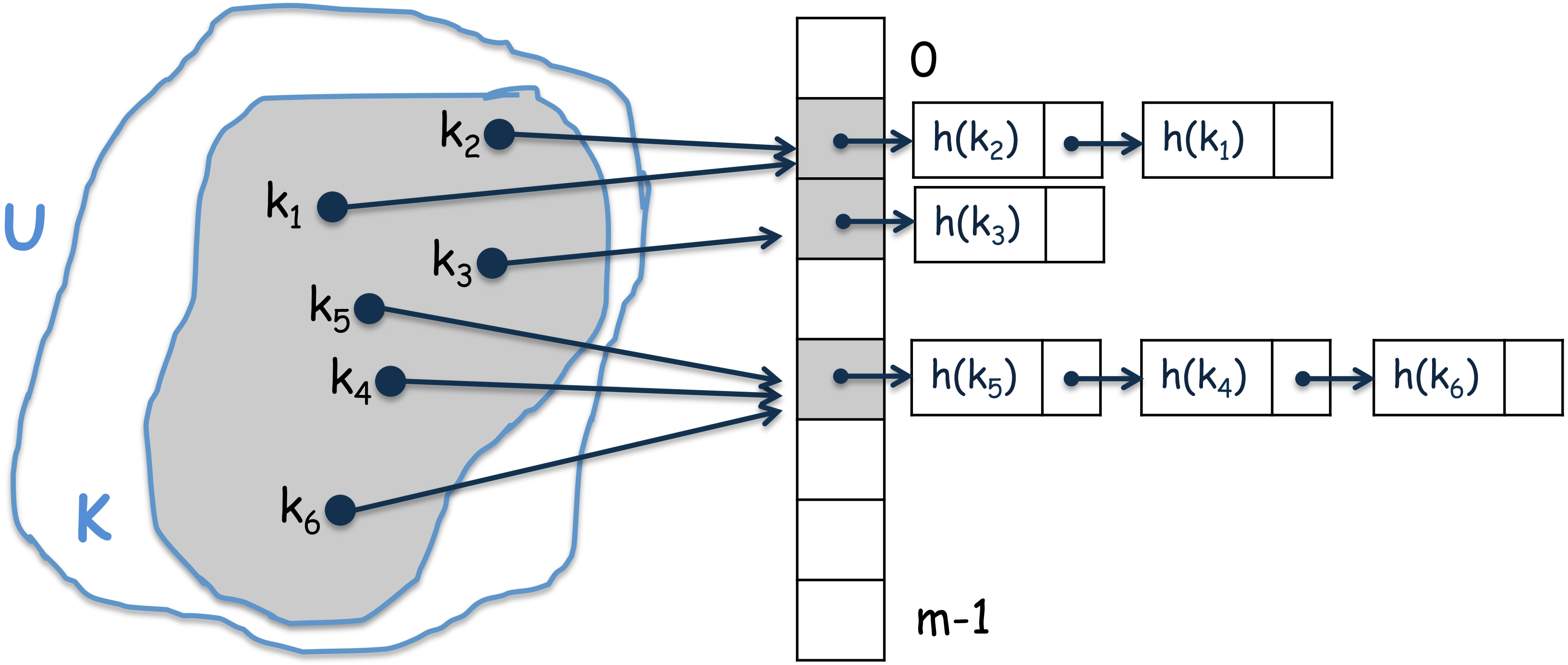
$$h : U \rightarrow \{0, 1, \ldots, m-1\}$$

(an item x hashes to the slot H[h(x)])

- A good hash function should :
  - be a easy to compute
  - distribute the keys evenly through the hash table
  - avoid collisions as much as possible
  - use less space (or slots)

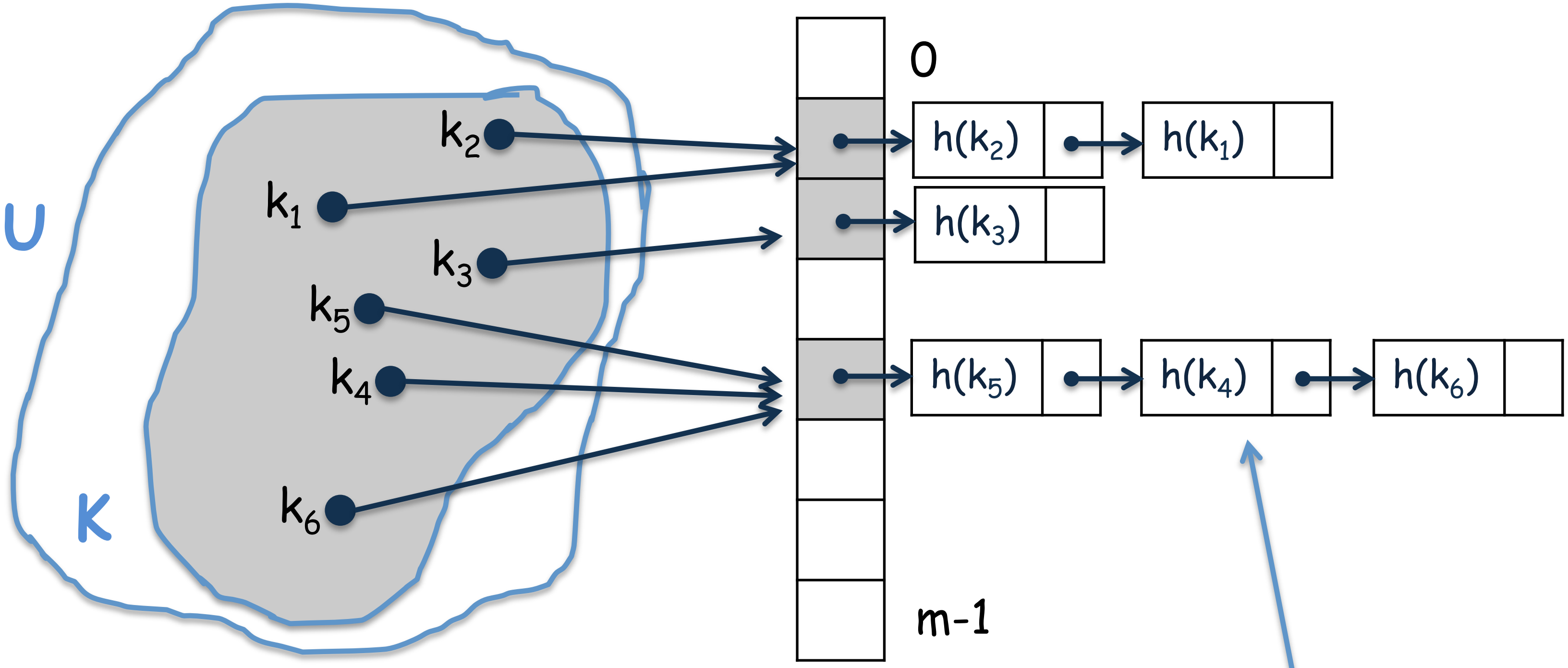
Pre-Structuring

Open Hashing (Separate Chaining)



Pre-Structuring

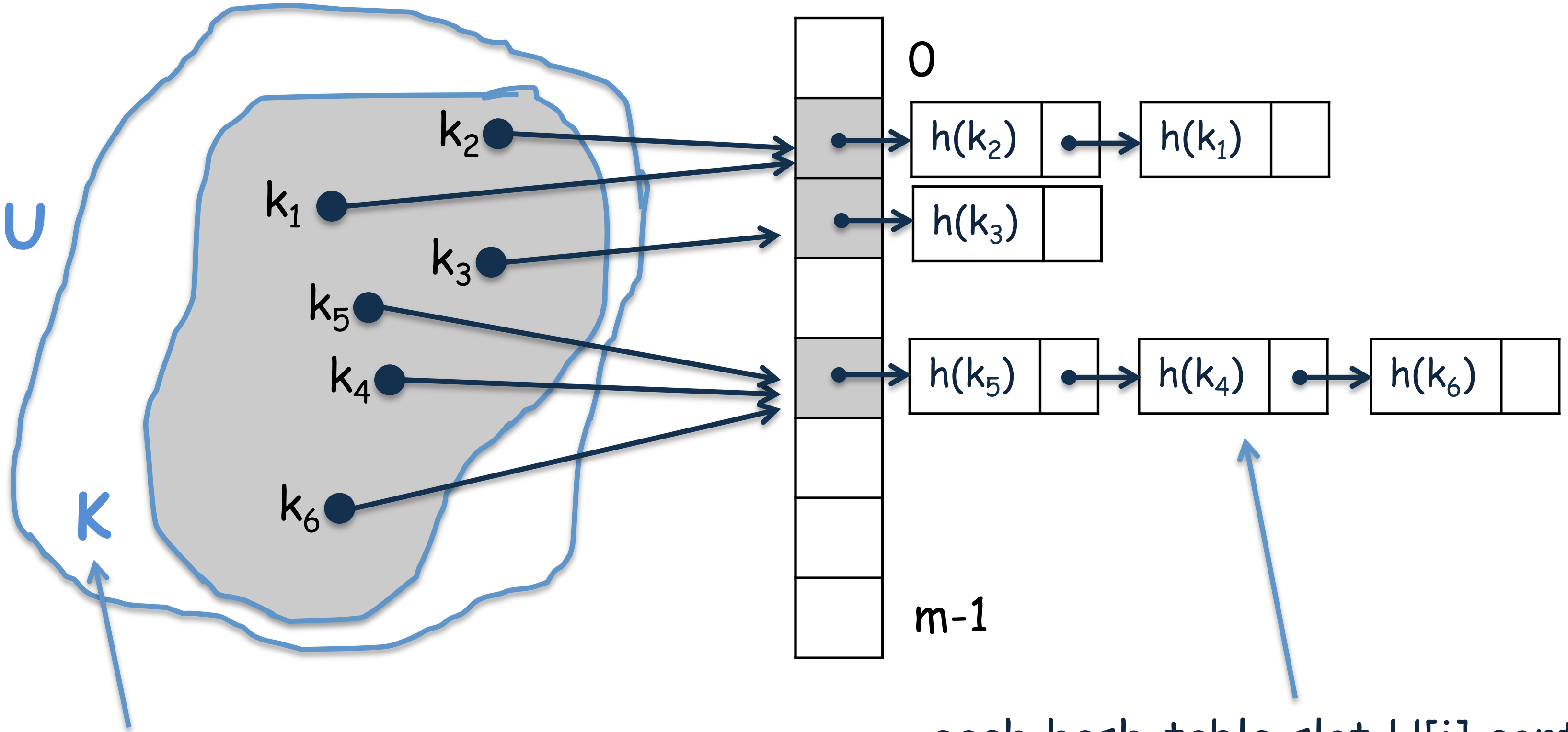
Open Hashing (Separate Chaining)



each hash-table slot H[i] contains a linked list of all the keys whose hash value is i

Pre-Structuring

Open Hashing (Separate Chaining)

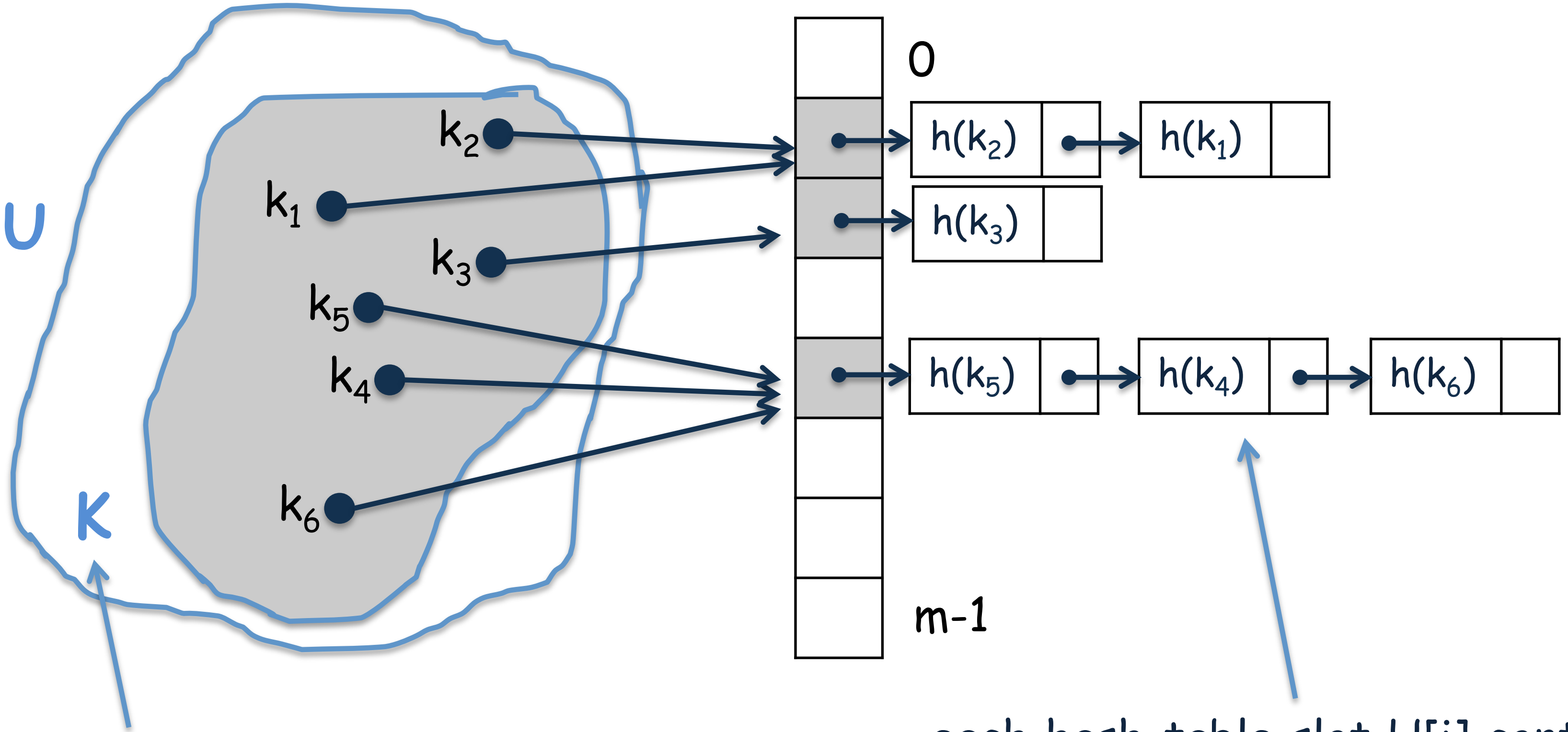


N keys to be stored and m slots in hash table; average list length is N/m (this fraction is called load factor)

each hash-table slot H[i] contains a linked list of all the keys whose hash value is i

Pre-Structuring

Open Hashing (Separate Chaining)

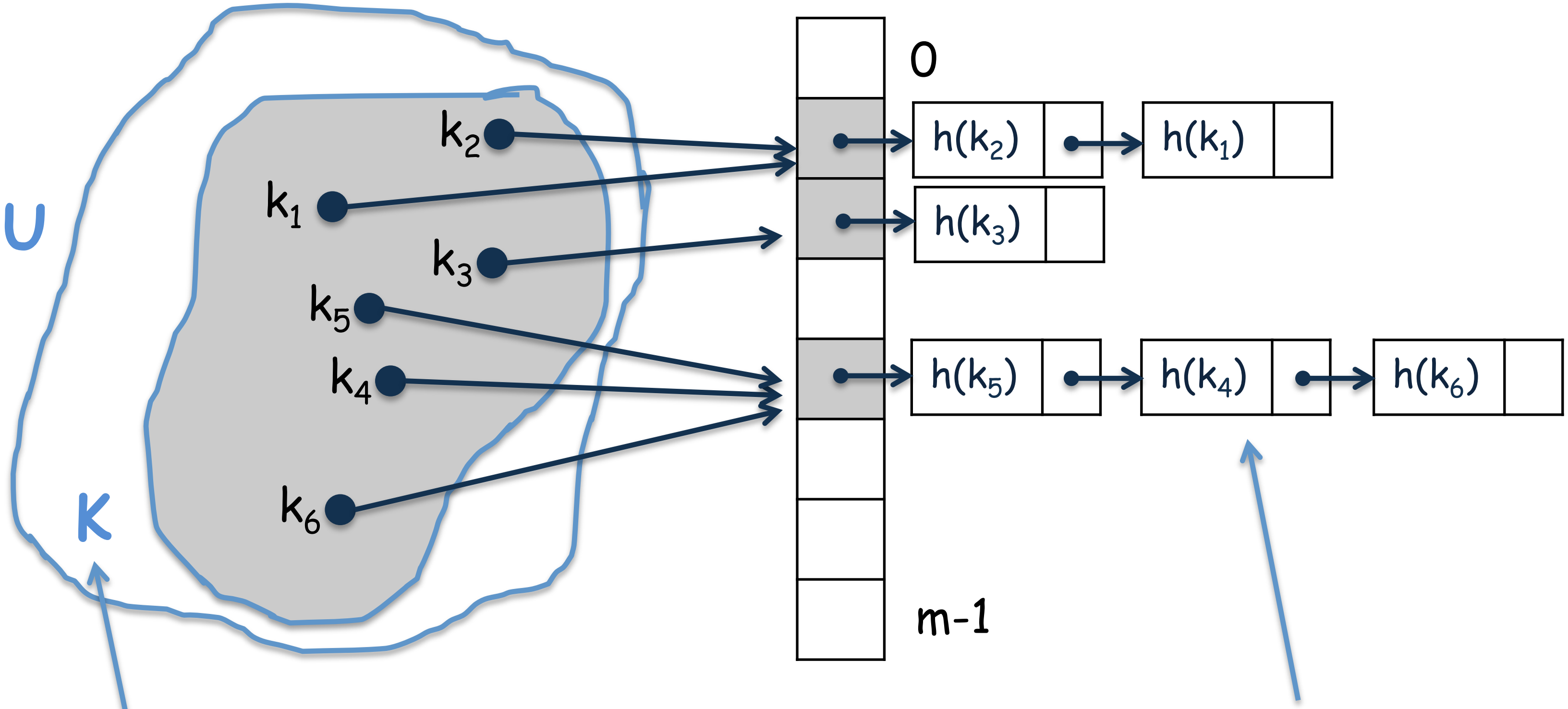


N keys to be stored and m slots in hash table; average list length is N/m (this fraction is called load factor)
worst case?

each hash-table slot H[i] contains a linked list of all the keys whose hash value is i

Pre-Structuring

Open Hashing (Separate Chaining)



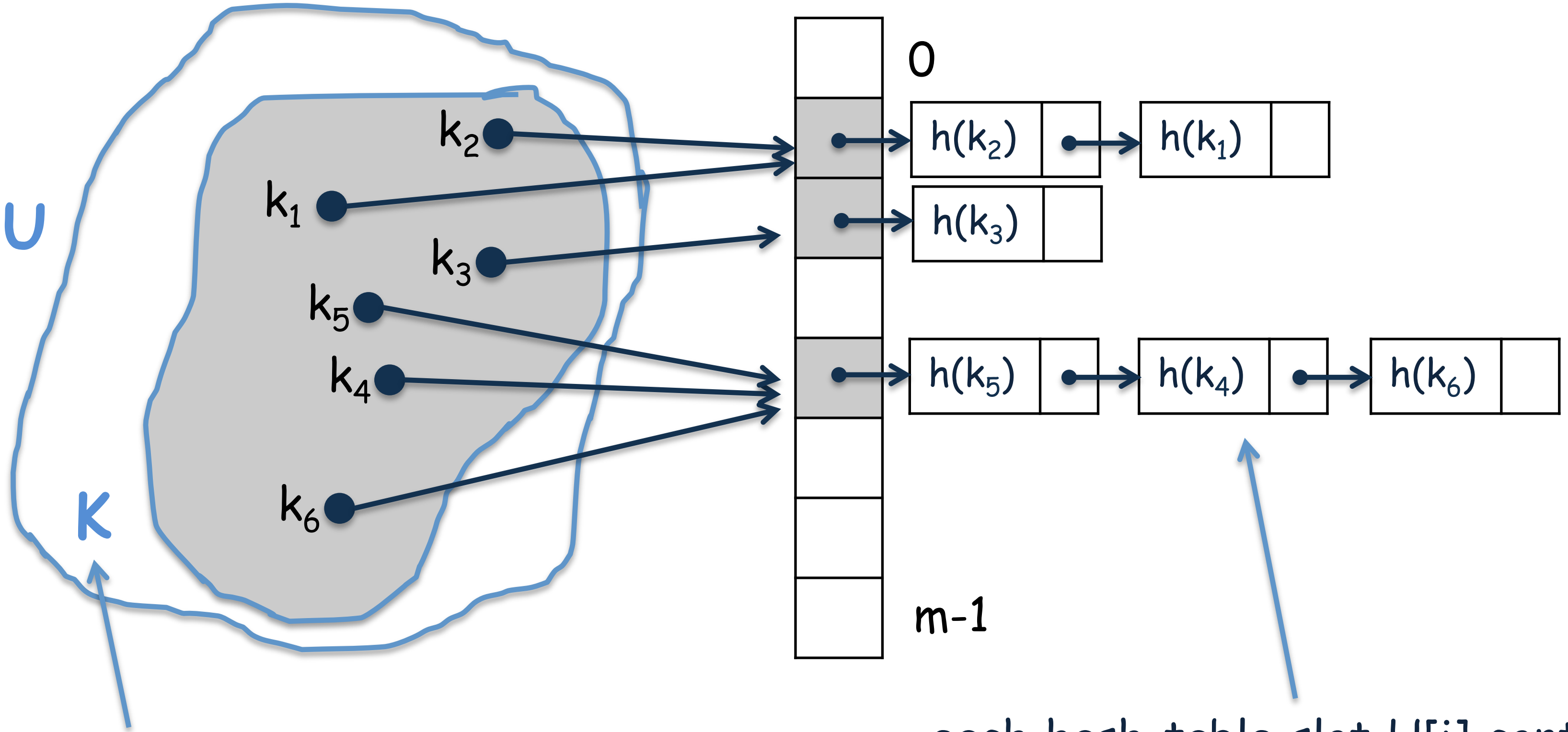
N keys to be stored and m slots in hash table; average list length is N/m (this fraction is called load factor)

each hash-table slot H[i] contains a linked list of all the keys whose hash value is i

m is too large → too many empty arrays entry

Pre-Structuring

Open Hashing (Separate Chaining)



N keys to be stored and m slots in hash table; average list length is N/m (this fraction is called load factor)

each hash-table slot H[i] contains a linked list of all the keys whose hash value is i

m is too large ➔ too many empty arrays entry
m is too small ➔ list will be too long

Pre-Structuring

Open Hashing (Separate Chaining)

Consider the following example :

A FOOL ARE SOON

Pre-Structuring

Open Hashing (Separate Chaining)

Consider the following example :

A FOOL ARE SOON

- Let’s define a hash function as : add the positions of the letters in the alphabet and compute the remainder of the division of the sum by 13

Pre-Structuring

Open Hashing (Separate Chaining)

Consider the following example :

A FOOL ARE SOON

- Let's define a hash function as : add the positions of the letters in the alphabet and compute the remainder of the division of the sum by 13
- h(A) = 1 mod 13 = 1
  h(FOOL) = (6 + 15 + 15 + 12) mod 13 = 9
  h(ARE) = (1 + 18 + 5) mod 13 = 11
  h(SOON) = (19 + 15 + 15 + 14) mod 13 = 11

Pre-Structuring

Open Hashing (Separate Chaining)

Consider the following example :

A FOOL ARE SOON

- Let's define a hash function as : add the positions of the letters in the alphabet and compute the remainder of the division of the sum by 13
- h(A) = 1 mod 13 = 1
  h(FOOL) = (6 + 15 + 15 + 12) mod 13 = 9
  h(ARE) = (1 + 18 + 5) mod 13 = 11
  h(SOON) = (19 + 15 + 15 + 14) mod 13 = 11
- ARE and SOON are stored in same linked list

Pre-Structuring

Open Hashing (Separate Chaining)

Consider the following example :

A FOOL ARE SOON

- Let's define a hash function as : add the positions of the letters in the alphabet and compute the remainder of the division of the sum by 13
- h(A) = 1 mod 13 = 1
  h(FOOL) = (6 + 15 + 15 + 12) mod 13 = 9
  h(ARE) = (1 + 18 + 5) mod 13 = 11
  h(SOON) = (19 + 15 + 15 + 14) mod 13 = 11
- ARE and SOON are stored in same linked list
- How do we search in the hash table?

Pre-Structuring

Open Hashing (Separate Chaining)

Consider the following example :

A FOOL ARE SOON

- Let's define a hash function as : add the positions of the letters in the alphabet and compute the remainder of the division of the sum by 13
- h(A) = 1 mod 13 = 1
  h(FOOL) = (6 + 15 + 15 + 12) mod 13 = 9
  h(ARE) = (1 + 18 + 5) mod 13 = 11
  h(SOON) = (19 + 15 + 15 + 14) mod 13 = 11
- ARE and SOON are stored in same linked list
- How do we search in the hash table?
  -- search whether the table contains KID or not
  -- compute h(KID) = 11
  -- search corresponding linked-list which includes ARE and SOON

Pre-Structuring

Open Hashing (Separate Chaining)

Consider the following example :

A FOOL ARE SOON

- Let's define a hash function as : add the positions of the letters in the alphabet and compute the remainder of the division of the sum by 13
- h(A) = 1 mod 13 = 1
  h(FOOL) = (6 + 15 + 15 + 12) mod 13 = 9
  h(ARE) = (1 + 18 + 5) mod 13 = 11
  h(SOON) = (19 + 15 + 15 + 14) mod 13 = 11
- ARE and SOON are stored in same linked list
- How do we search in the hash table?
  - -- search whether the table contains KID or not
  - -- compute h(KID) = 11
  - -- search corresponding linked-list which includes ARE and SOON
- In Separate Chaining, a search takes $O(1+\pi)$ time in average $\pi = N/m$

Pre-Structuring

Open Hashing (Separate Chaining)

Consider the following example :

A FOOL ARE SOON

- Let's define a hash function as : add the positions of the letters in the alphabet and compute the remainder of the division of the sum by 13
- h(A) = 1 mod 13 = 1
  h(FOOL) = (6 + 15 + 15 + 12) mod 13 = 9
  h(ARE) = (1 + 18 + 5) mod 13 = 11
  h(SOON) = (19 + 15 + 15 + 14) mod 13 = 11
- ARE and SOON are stored in same linked list
- How do we search in the hash table?
  -- search whether the table contains KID or not
  -- compute h(KID) = 11
  -- search corresponding linked-list which includes ARE and SOON
- In Separate Chaining, a search takes $O(1+\pi)$ time in average $\pi = N/m$
- the average number of cells examined in a successful search, S (U for unseccessful) : $S \approx 1 + \pi/2$ and $U \approx \pi$

Pre-Structuring

Closed Hashing (Open Addressing)

Linear Probing

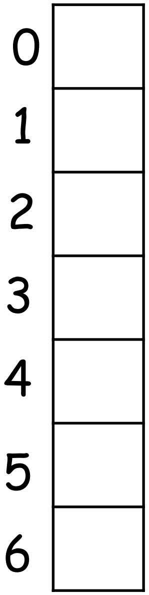
- $h(k,i) = (h'(k) + i) \bmod m$ where $h' : U \{0,1,\ldots,m-1\}$ is an ordinary hash function

Pre-Structuring

Closed Hashing (Open Addressing)

Linear Probing

- $h(k,i) = (h'(k) + i) \bmod m$ where $h' : U \{0,1,\ldots,m-1\}$ is an ordinary hash function

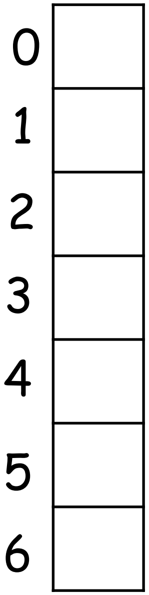


Pre-Structuring

Closed Hashing (Open Addressing)

Linear Probing

- $h(k,i) = (h'(k) + i) \bmod m$ where $h' : U \{0,1,\ldots,m-1\}$ is an ordinary hash function



insert(18)

Pre-Structuring

Closed Hashing (Open Addressing)

Linear Probing

- h(k,i) = (h′(k) + i) mod m where h′ : U {0,1,…,m-1} is an ordinary hash function

0	
1	
2	
3	
4	18
5	
6	

insert(18)

h(18) = 18 mod 7

h(18) = 4

Pre-Structuring

Closed Hashing (Open Addressing)

Linear Probing

- $h(k,i) = (h'(k) + i) \bmod m$ where $h' : U \{0,1,\ldots,m-1\}$ is an ordinary hash function

0	14
1	
2	
3	
4	18
5	
6	

insert(18)

h(18) = 18 mod 7

h(18) = 4

insert(14)

h(14) = 14 mod 7

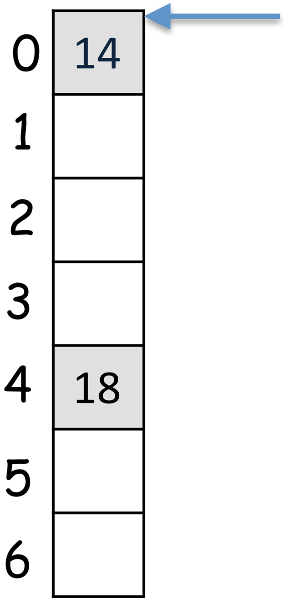
h(14) = 0

Pre-Structuring

Closed Hashing (Open Addressing)

Linear Probing

- h(k,i) = (h′(k) + i) mod m where h′ : U {0,1,…,m-1} is an ordinary hash function



insert(18)

h(18) = 18 mod 7

h(18) = 4

insert(21)

h(21) = 21 mod 7

h(21) = 0

insert(14)

h(14) = 14 mod 7

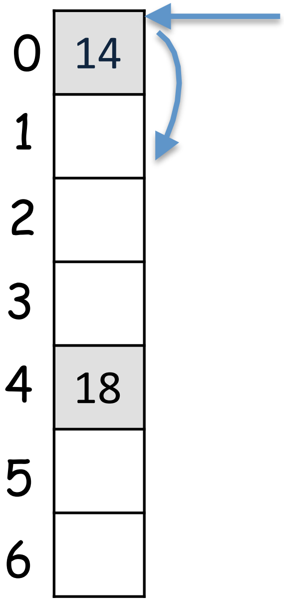
h(14) = 0

Pre-Structuring

Closed Hashing (Open Addressing)

Linear Probing

- $h(k,i) = (h'(k) + i) \bmod m$ where $h' : U \{0,1,\ldots,m-1\}$ is an ordinary hash function



insert(18)

$h(18) = 18 \bmod 7$

$h(18) = 4$

insert(14)

$h(14) = 14 \bmod 7$

$h(14) = 0$

insert(21)

$h(21) = 21 \bmod 7$

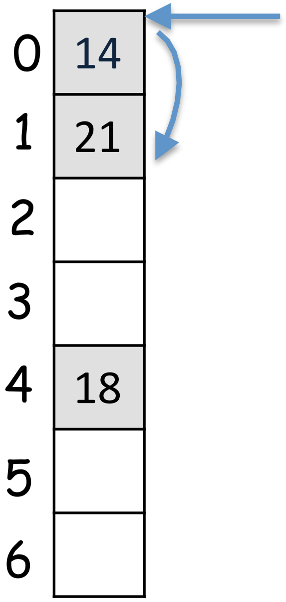
$h(21) = 0$

Pre-Structuring

Closed Hashing (Open Addressing)

Linear Probing

- $h(k,i) = (h'(k) + i) \bmod m$ where $h' : U \{0,1,\ldots,m-1\}$ is an ordinary hash function



0	14
1	21
2	
3	
4	18
5	
6	

insert(18)

$h(18) = 18 \bmod 7$

$h(18) = 4$

insert(21)

$h(21) = 21 \bmod 7$

$h(21) = 0$

insert(14)

$h(14) = 14 \bmod 7$

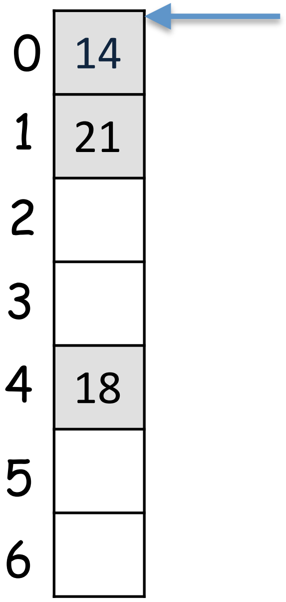
$h(14) = 0$

Pre-Structuring

Closed Hashing (Open Addressing)

Linear Probing

- $h(k,i) = (h'(k) + i) \bmod m$ where $h' : U \{0,1,\ldots,m-1\}$ is an ordinary hash function



insert(18)

h(18) = 18 mod 7

h(18) = 4

insert(21)

h(21) = 21 mod 7

h(21) = 0

insert(14)

h(14) = 14 mod 7

h(14) = 0

insert(35)

h(35) = 35 mod 7

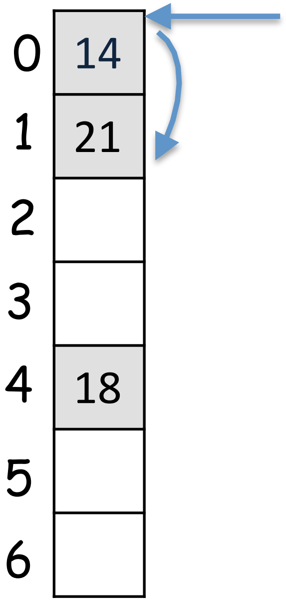
h(35) = 0

Pre-Structuring

Closed Hashing (Open Addressing)

Linear Probing

- $h(k,i) = (h'(k) + i) \bmod m$ where $h' : U \{0,1,\ldots,m-1\}$ is an ordinary hash function



0	14
1	21
2	
3	
4	18
5	
6	

insert(18)

$h(18) = 18 \bmod 7$

$h(18) = 4$

insert(21)

$h(21) = 21 \bmod 7$

$h(21) = 0$

insert(14)

$h(14) = 14 \bmod 7$

$h(14) = 0$

insert(35)

$h(35) = 35 \bmod 7$

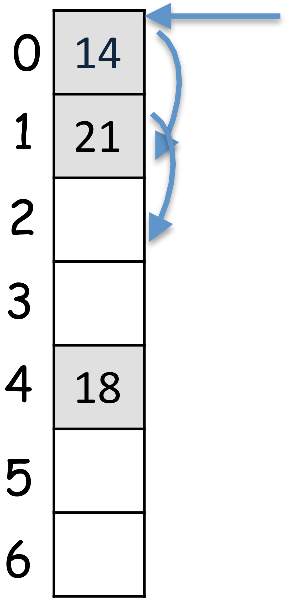
$h(35) = 0$

Pre-Structuring

Closed Hashing (Open Addressing)

Linear Probing

- $h(k,i) = (h'(k) + i) \bmod m$ where $h' : U \{0,1,\ldots,m-1\}$ is an ordinary hash function



<u>insert(18)</u>
$h(18) = 18 \bmod 7$
$h(18) = 4$

<u>insert(21)</u>
$h(21) = 21 \bmod 7$
$h(21) = 0$

<u>insert(14)</u>
$h(14) = 14 \bmod 7$
$h(14) = 0$

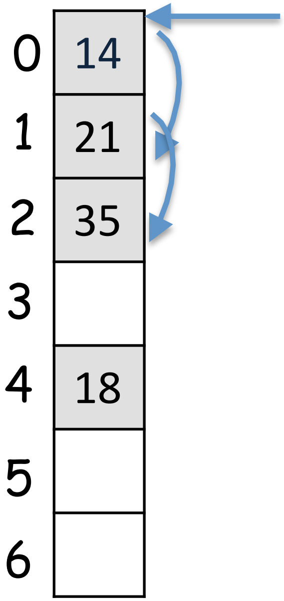
<u>insert(35)</u>
$h(35) = 35 \bmod 7$
$h(35) = 0$

Pre-Structuring

Closed Hashing (Open Addressing)

Linear Probing

- h(k,i) = (h'(k) + i) mod m where h' : U {0,1,…,m-1} is an ordinary hash function



insert(18)
h(18) = 18 mod 7
h(18) = 4

insert(21)
h(21) = 21 mod 7
h(21) = 0

insert(14)
h(14) = 14 mod 7
h(14) = 0

insert(35)
h(35) = 35 mod 7
h(35) = 0

Pre-Structuring

Closed Hashing (Open Addressing)

Linear Probing

- $h(k,i) = (h'(k) + i) \bmod m$ where $h' : U \{0,1,\ldots,m-1\}$ is an ordinary hash function

0	14
1	21
2	35
3	
4	18
5	
6	

insert(18)

$h(18) = 18 \bmod 7$

$h(18) = 4$

insert(21)

$h(21) = 21 \bmod 7$

$h(21) = 0$

insert(14)

$h(14) = 14 \bmod 7$

$h(14) = 0$

insert(35)

$h(35) = 35 \bmod 7$

$h(35) = 0$

insert(8)

$h(8) = 8 \bmod 7$

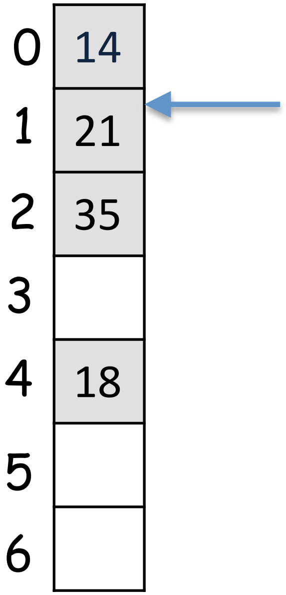
$h(8) = 1$

Pre-Structuring

Closed Hashing (Open Addressing)

Linear Probing

- h(k,i) = (h'(k) + i) mod m where h' : U {0,1,…,m-1} is an ordinary hash function



insert(18)
h(18) = 18 mod 7
h(18) = 4

insert(21)
h(21) = 21 mod 7
h(21) = 0

insert(14)
h(14) = 14 mod 7
h(14) = 0

insert(35)
h(35) = 35 mod 7
h(35) = 0

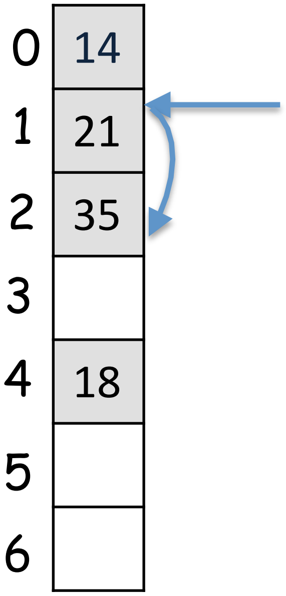
insert(8)
h(8) = 8 mod 7
h(8) = 1

Pre-Structuring

Closed Hashing (Open Addressing)

Linear Probing

- $h(k,i) = (h'(k) + i) \bmod m$ where $h' : U \{0,1,\ldots,m-1\}$ is an ordinary hash function



0	14
1	21
2	35
3	
4	18
5	
6	

insert(18)
h(18) = 18 mod 7
h(18) = 4

insert(14)
h(14) = 14 mod 7
h(14) = 0

insert(21)
h(21) = 21 mod 7
h(21) = 0

insert(35)
h(35) = 35 mod 7
h(35) = 0

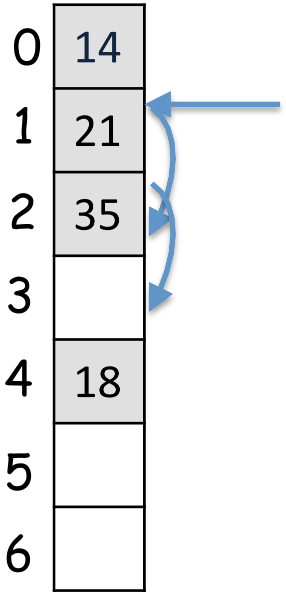
insert(8)
h(8) = 8 mod 7
h(8) = 1

Pre-Structuring

Closed Hashing (Open Addressing)

Linear Probing

- $h(k,i) = (h'(k) + i) \bmod m$ where $h' : U \{0,1,\ldots,m-1\}$ is an ordinary hash function



0	14
1	21
2	35
3	
4	18
5	
6	

insert(18)

$h(18) = 18 \bmod 7$

$h(18) = 4$

insert(14)

$h(14) = 14 \bmod 7$

$h(14) = 0$

insert(21)

$h(21) = 21 \bmod 7$

$h(21) = 0$

insert(35)

$h(35) = 35 \bmod 7$

$h(35) = 0$

insert(8)

$h(8) = 8 \bmod 7$

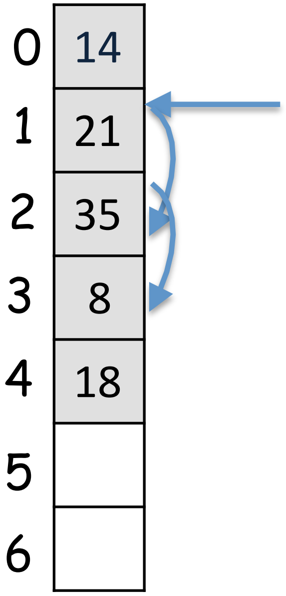
$h(8) = 1$

Pre-Structuring

Closed Hashing (Open Addressing)

Linear Probing

- $h(k,i) = (h'(k) + i) \bmod m$ where $h' : U \{0,1,\dots,m-1\}$ is an ordinary hash function



insert(18)

$h(18) = 18 \bmod 7$

$h(18) = 4$

insert(21)

$h(21) = 21 \bmod 7$

$h(21) = 0$

insert(14)

$h(14) = 14 \bmod 7$

$h(14) = 0$

insert(35)

$h(35) = 35 \bmod 7$

$h(35) = 0$

insert(8)

$h(8) = 8 \bmod 7$

$h(8) = 1$

Pre-Structuring

Closed Hashing (Open Addressing)

Linear Probing

- $h(k,i) = (h'(k) + i) \bmod m$ where $h' : U\ \{0,1,\ldots,m-1\}$ is an ordinary hash function

0	14
1	21
2	35
3	8
4	18
5	
6	

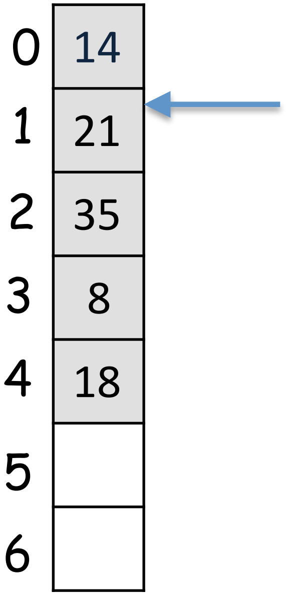
find(8)

Pre-Structuring

Closed Hashing (Open Addressing)

Linear Probing

- $h(k,i) = (h'(k) + i) \bmod m$ where $h' : U \{0,1,\ldots,m-1\}$ is an ordinary hash function



0	14
1	21
2	35
3	8
4	18
5	
6	

find(8)

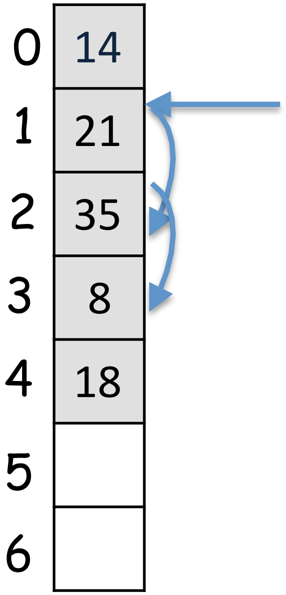
$h(8)=1$

Pre-Structuring

Closed Hashing (Open Addressing)

Linear Probing

- $h(k,i) = (h'(k) + i) \bmod m$ where $h' : U \{0,1,\ldots,m-1\}$ is an ordinary hash function



find(8)

h(8)=1

after two probes

Pre-Structuring

Closed Hashing (Open Addressing)

Linear Probing

- $h(k,i) = (h'(k) + i) \bmod m$ where $h' : U \{0,1,\ldots,m-1\}$ is an ordinary hash function

0	14
1	21
2	35
3	8
4	18
5	
6	

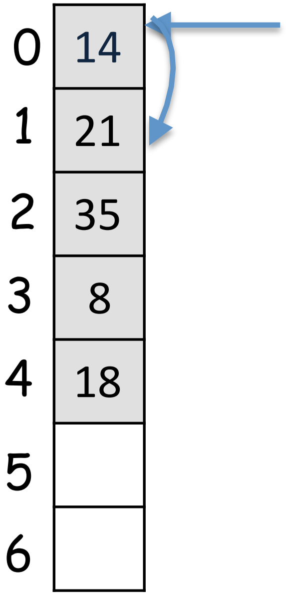
delete(21)

Pre-Structuring

Closed Hashing (Open Addressing)

Linear Probing

- h(k,i) = (h'(k) + i) mod m where h' : U {0,1,…,m-1} is an ordinary hash function



0	14
1	21
2	35
3	8
4	18
5	
6	

delete(21)

h(21)=0

Pre-Structuring

Closed Hashing (Open Addressing)

Linear Probing

- $h(k,i) = (h'(k) + i) \bmod m$ where $h' : U \{0,1,\ldots,m-1\}$ is an ordinary hash function

0	14
1	
2	35
3	8
4	18
5	
6	

delete(21)

$h(21)=0$

Pre-Structuring

Closed Hashing (Open Addressing)

Linear Probing

- $h(k,i) = (h'(k) + i) \bmod m$ where $h' : U \{0,1,\dots,m-1\}$ is an ordinary hash function

0	14
1	
2	35
3	8
4	18
5	
6	

find(35)

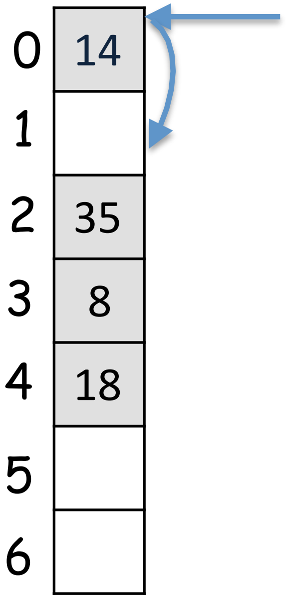
h(35)=0

Pre-Structuring

Closed Hashing (Open Addressing)

Linear Probing

- $h(k,i) = (h'(k) + i) \bmod m$ where $h' : U \{0,1,\ldots,m-1\}$ is an ordinary hash function



find(35)

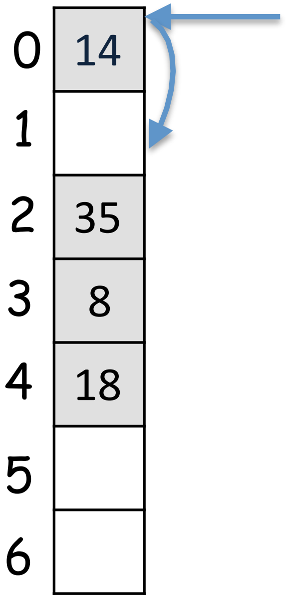
h(35)=0

Pre-Structuring

Closed Hashing (Open Addressing)

Linear Probing

- $h(k,i) = (h'(k) + i) \bmod m$ where $h' : U \{0,1,\ldots,m-1\}$ is an ordinary hash function



0	14
1	
2	35
3	8
4	18
5	
6	

find(35)

h(35)=0

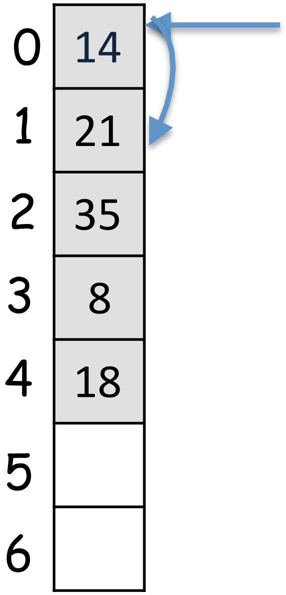
put some indicator when you delete

Pre-Structuring

Closed Hashing (Open Addressing)

Linear Probing

- $h(k,i) = (h'(k) + i) \bmod m$ where $h' : U \{0,1,\ldots,m-1\}$ is an ordinary hash function



0	14
1	21
2	35
3	8
4	18
5	
6	

delete(21)

h(21)=0

Pre-Structuring

Closed Hashing (Open Addressing)

Linear Probing

- $h(k,i) = (h'(k) + i) \bmod m$ where $h' : U \{0,1,\ldots,m-1\}$ is an ordinary hash function

0	14
1	X
2	35
3	8
4	18
5	
6	

delete(21)

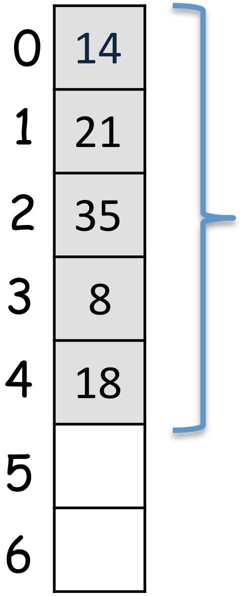
$h(21)=0$

Pre-Structuring

Closed Hashing (Open Addressing)

Linear Probing

- $h(k,i) = (h'(k) + i) \bmod m$ where $h' : U \{0,1,\ldots,m-1\}$ is an ordinary hash function



0	14
1	21
2	35
3	8
4	18
5	
6	

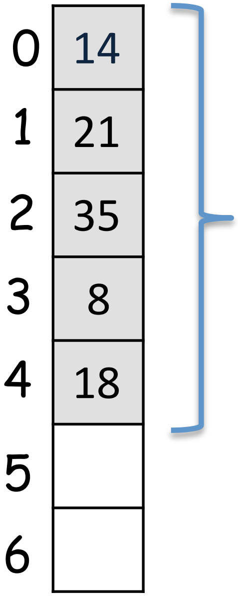
A cluster is collection of consecutive occupied slots

Pre-Structuring

Closed Hashing (Open Addressing)

Linear Probing

- $h(k,i) = (h'(k) + i) \bmod m$ where $h' : U \{0,1,\ldots,m-1\}$ is an ordinary hash function



0	14
1	21
2	35
3	8
4	18
5	
6	

A cluster is collection of consecutive occupied slots

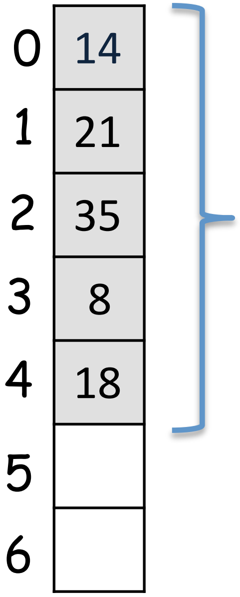
Linear Probing can create large clusters that increases the running time of find-insert-delete operations

Pre-Structuring

Closed Hashing (Open Addressing)

Linear Probing

- $h(k,i) = (h'(k) + i) \bmod m$ where $h' : U \{0,1,\ldots,m-1\}$ is an ordinary hash function



A cluster is collection of consecutive occupied slots

Linear Probing can create large clusters that increases the running time of find-insert-delete operations

the average number of cells examined in a successful search, S (U for unseccessful) :

$$S \approx 1/2\ (1 + 1/(1-\pi)) \text{ and } U \approx 1/2\ (1 + 1/(1-\pi)^2)$$

Pre-Structuring

Quadratic Probing

Closed Hashing (Open Addressing)

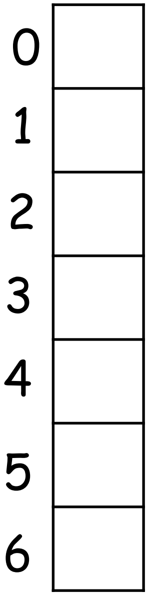
- $h(k,i) = (h'(k) + ci + ci^2) \bmod m$ where $h' : U \rightarrow \{0,1,\ldots,m-1\}$ is an ordinary hash function

Pre-Structuring

Quadratic Probing

Closed Hashing (Open Addressing)

- $h(k,i) = (h'(k) + ci + ci^2) \bmod m$ where $h' : U \rightarrow \{0,1,\dots,m-1\}$ is an ordinary hash function



simply use the following form

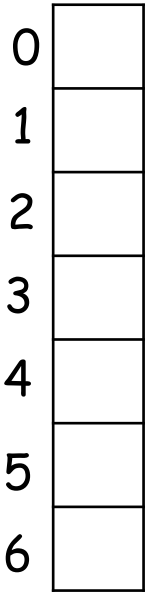
$(h'(k) + 1) \bmod m$
$(h'(k) + 4) \bmod m$
$(h'(k) + 9) \bmod m$
…

Pre-Structuring

Quadratic Probing

Closed Hashing (Open Addressing)

- $h(k,i) = (h'(k) + ci + ci^2) \bmod m$ where $h' : U \rightarrow \{0,1,\ldots,m-1\}$ is an ordinary hash function



insert(8)

simply use the following form

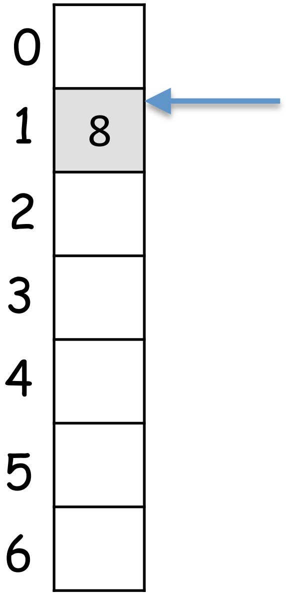
$(h'(k) + 1) \bmod m$
$(h'(k) + 4) \bmod m$
$(h'(k) + 9) \bmod m$
…

Pre-Structuring

Quadratic Probing

Closed Hashing (Open Addressing)

- $h(k,i) = (h'(k) + ci + ci^2) \bmod m$ where $h' : U \rightarrow \{0,1,\ldots,m-1\}$ is an ordinary hash function



insert(8)

$h(8)=8 \bmod 7$

$h(8)=1$

simply use the following form

$(h'(k) + 1) \bmod m$

$(h'(k) + 4) \bmod m$

$(h'(k) + 9) \bmod m$

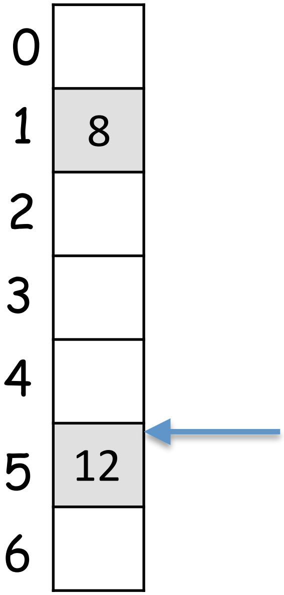
…

Pre-Structuring

Quadratic Probing

Closed Hashing (Open Addressing)

- $h(k,i) = (h'(k) + ci + ci^2) \bmod m$ where $h' : U \rightarrow \{0,1,\ldots,m-1\}$ is an ordinary hash function



insert(8)

$h(8)=8 \bmod 7$

$h(8)=1$

insert(12)

$h(12)=12 \bmod 7$

$h(12)=5$

simply use the following form

$(h'(k) + 1) \bmod m$

$(h'(k) + 4) \bmod m$

$(h'(k) + 9) \bmod m$

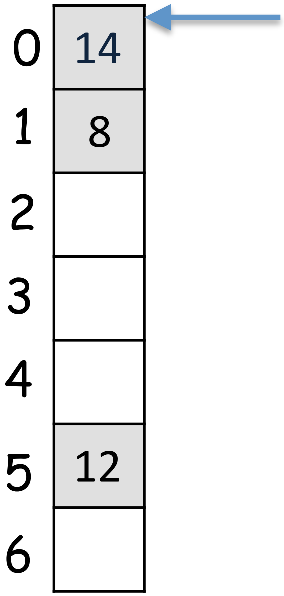
…

Pre-Structuring

Quadratic Probing

Closed Hashing (Open Addressing)

- $h(k,i) = (h'(k) + ci + ci^2) \bmod m$ where $h' : U \rightarrow \{0,1,\ldots,m-1\}$ is an ordinary hash function



0	14
1	8
2	
3	
4	
5	12
6	

insert(8)
h(8)=8 mod 7
h(8)=1

insert(12)
h(12)=12 mod 7
h(12)=5

insert(14)
h(14)=14 mod 7
h(14)=0

simply use the following form

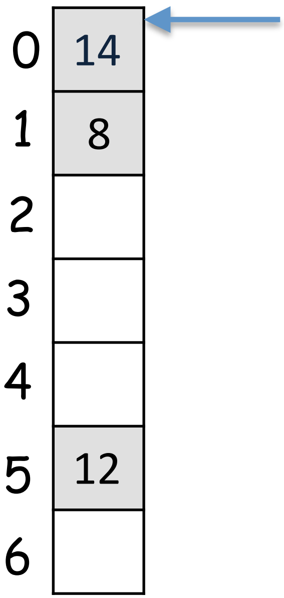
(h'(k) +1) mod m
(h'(k) +4) mod m
(h'(k) +9) mod m
…

Pre-Structuring

Quadratic Probing

Closed Hashing (Open Addressing)

- $h(k,i) = (h'(k) + ci + ci^2) \bmod m$ where $h' : U \rightarrow \{0,1,\dots,m-1\}$ is an ordinary hash function



0	14
1	8
2	
3	
4	
5	12
6	

insert(8)
h(8)=8 mod 7
h(8)=1

insert(14)
h(14)=14 mod 7
h(14)=0

insert(12)
h(12)=12 mod 7
h(12)=5

insert(21)
h(21)=21 mod 7
h(21)=0

simply use the following form

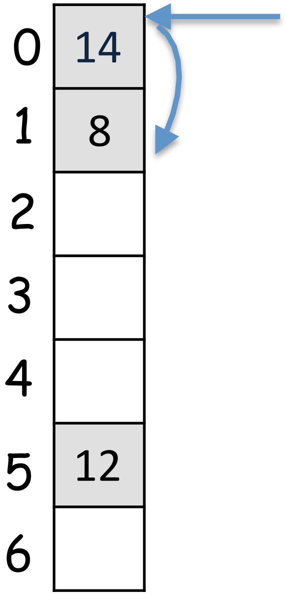
$(h'(k) + 1) \bmod m$
$(h'(k) + 4) \bmod m$
$(h'(k) + 9) \bmod m$
…

Pre-Structuring

Quadratic Probing

Closed Hashing (Open Addressing)

- $h(k,i) = (h'(k) + ci + ci^2) \bmod m$ where $h' : U \rightarrow \{0,1,\ldots,m-1\}$ is an ordinary hash function



insert(8)
h(8)=8 mod 7
h(8)=1

insert(14)
h(14)=14 mod 7
h(14)=0

insert(12)
h(12)=12 mod 7
h(12)=5

insert(21)
h(21)=21 mod 7
h(21)=0

simply use the following form

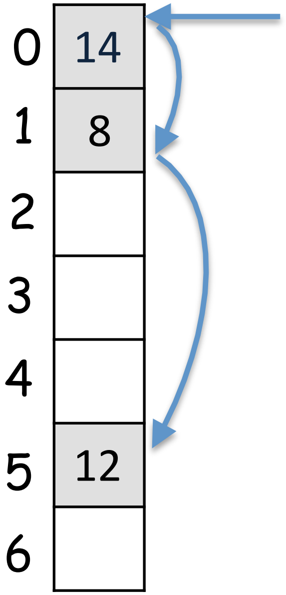
(h'(k) +1) mod m
(h'(k) +4) mod m
(h'(k) +9) mod m
…

Pre-Structuring

Quadratic Probing

Closed Hashing (Open Addressing)

- h(k,i) = (h'(k) +ci+ci^2) mod m where h' : U ➔ {0,1,…,m-1} is an ordinary hash function



insert(8)
h(8)=8 mod 7
h(8)=1

insert(14)
h(14)=14 mod 7
h(14)=0

insert(12)
h(12)=12 mod 7
h(12)=5

insert(21)
h(21)=21 mod 7
h(21)=0

simply use the following form

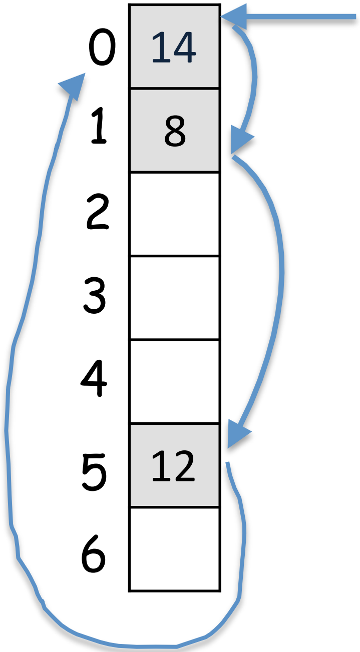
(h'(k) +1) mod m
(h'(k) +4) mod m
(h'(k) +9) mod m
…

Pre-Structuring

Quadratic Probing

Closed Hashing (Open Addressing)

- $h(k,i) = (h'(k) + ci + ci^2) \bmod m$ where $h' : U \rightarrow \{0,1,\ldots,m-1\}$ is an ordinary hash function



0	14
1	8
2	
3	
4	
5	12
6	

insert(8)

h(8)=8 mod 7

h(8)=1

insert(14)

h(14)=14 mod 7

h(14)=0

insert(12)

h(12)=12 mod 7

h(12)=5

insert(21)

h(21)=21 mod 7

h(21)=0

simply use the following form

(h'(k) +1) mod m

(h'(k) +4) mod m

(h'(k) +9) mod m

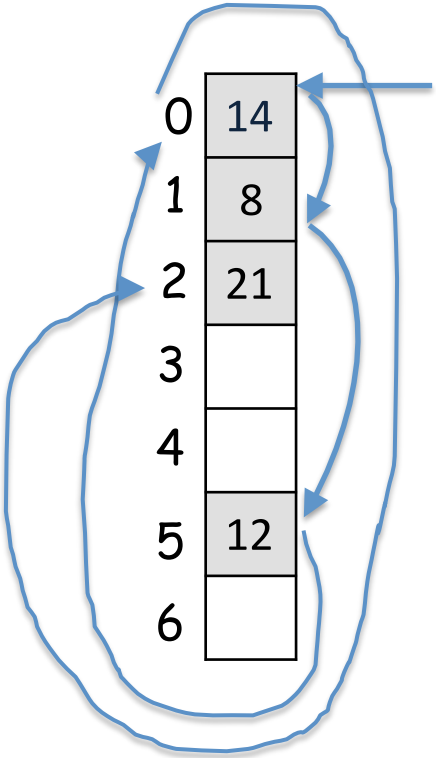
…

Pre-Structuring

Closed Hashing (Open Addressing)

Quadratic Probing

- $h(k,i) = (h'(k) + ci + ci^2) \bmod m$ where $h' : U \rightarrow \{0,1,\ldots,m-1\}$ is an ordinary hash function



0	14
1	8
2	21
3	
4	
5	12
6	

insert(8)
h(8)=8 mod 7
h(8)=1

insert(14)
h(14)=14 mod 7
h(14)=0

insert(12)
h(12)=12 mod 7
h(12)=5

insert(21)
h(21)=21 mod 7
h(21)=0

simply use the following form

(h'(k) +1) mod m
(h'(k) +4) mod m
(h'(k) +9) mod m
…