CSCI 3230 Data Structures

Sorting

Weitian Tong, Ph.D.

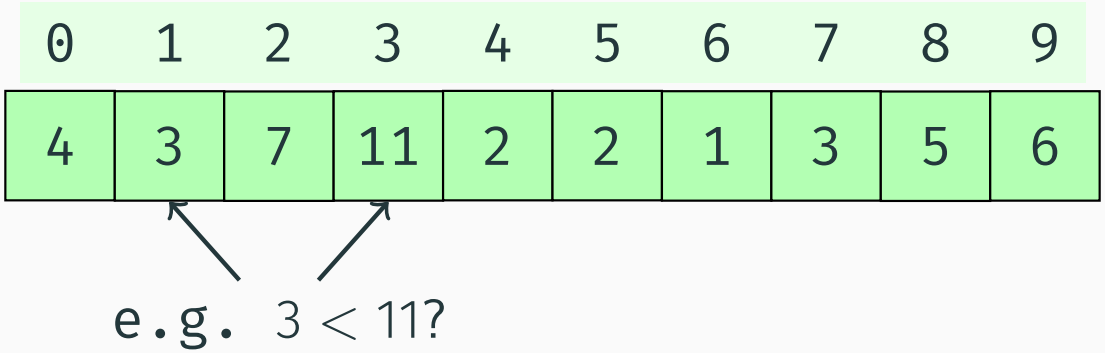
Department of Computer Science
Georgia Southern University
Website: www.weitiantong.com
Email:wtong@georgiasouthern.edu

Table of contents

Comparison Sort

Comparison Sort

Sort the input by successive comparison of pairs of input elements.



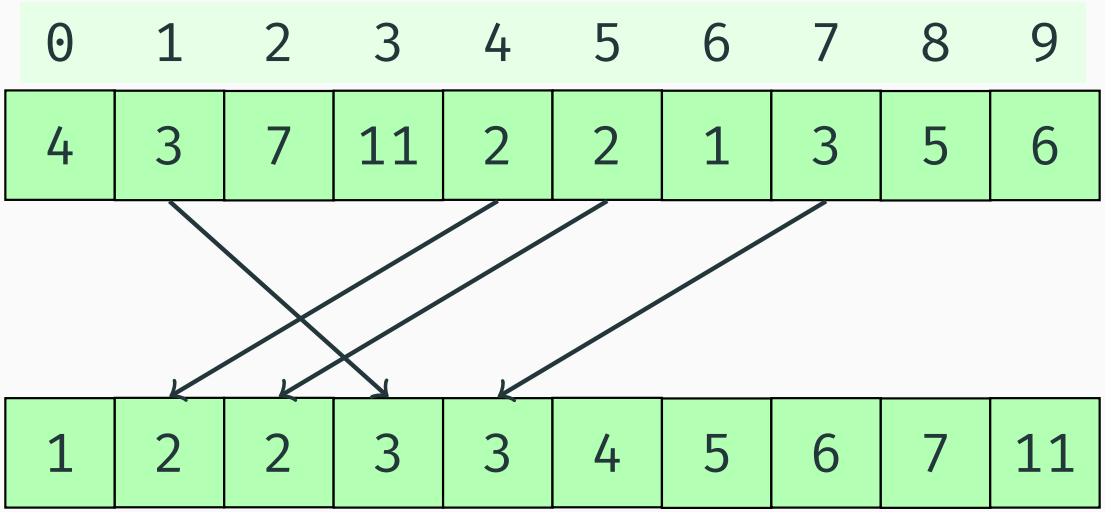
Comparison Sort algorithms are very general: they make no assumptions about the values of the input elements.

Sort in place, Stable sort

Sort in place: require only $O(1)$ additional memory.

e.g. sort by swapping elements within the input array

A sorting algorithm is said to be **stable** if the ordering of identical keys in the input is preserved in the output.



The stable sort property is important, for example, when entries with identical keys are already ordered by another criterion.

Comparison Sort

Selection Sort

Selection Sort

Selection Sort operates by

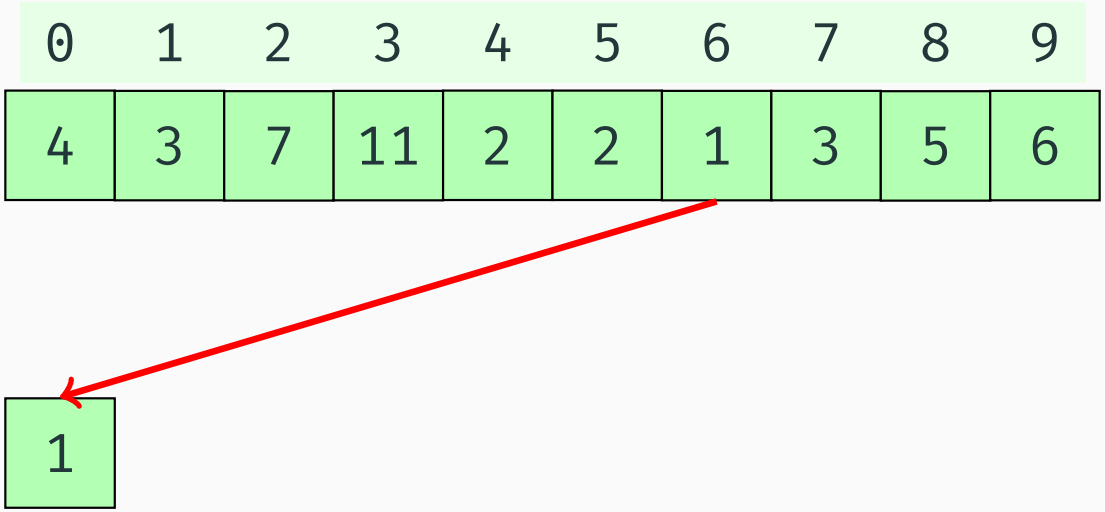
- first finding the smallest element in the input list, and moving it to the output list;
- then finding the next smallest value and does the same;
- continuing in this way until all the input elements have been selected and placed in the output list in the correct order.

0	1	2	3	4	5	6	7	8	9
4	3	7	11	2	2	1	3	5	6

Selection Sort

Selection Sort operates by

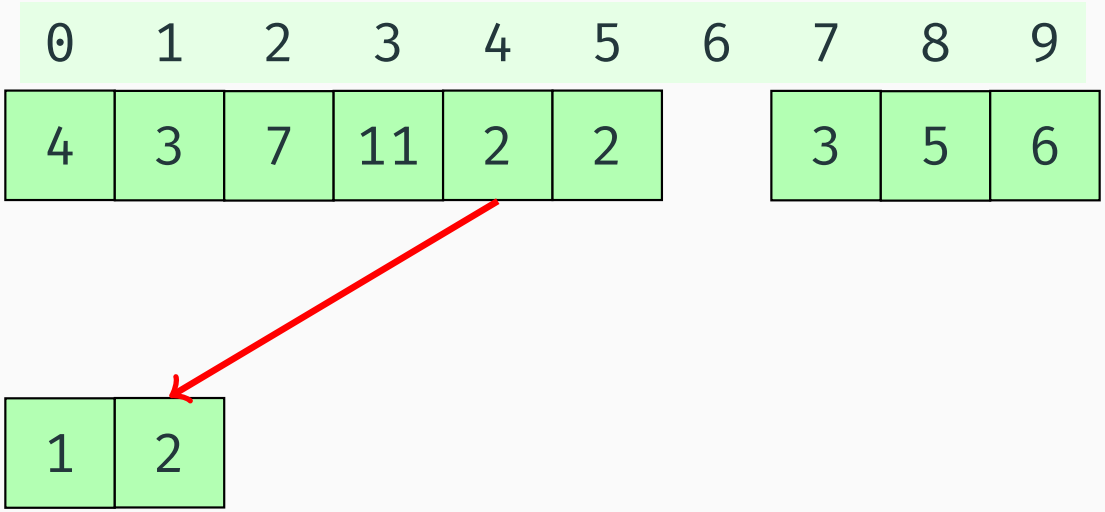
- first finding the smallest element in the input list, and moving it to the output list;
- then finding the next smallest value and does the same;
- continuing in this way until all the input elements have been selected and placed in the output list in the correct order.



Selection Sort

Selection Sort operates by

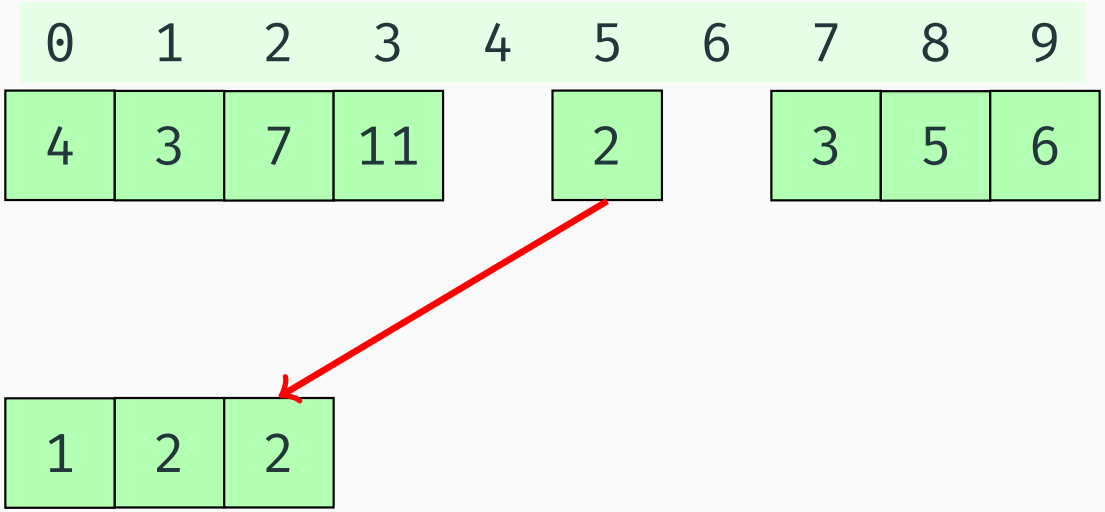
- first finding the smallest element in the input list, and moving it to the output list;
- then finding the next smallest value and does the same;
- continuing in this way until all the input elements have been selected and placed in the output list in the correct order.



Selection Sort

Selection Sort operates by

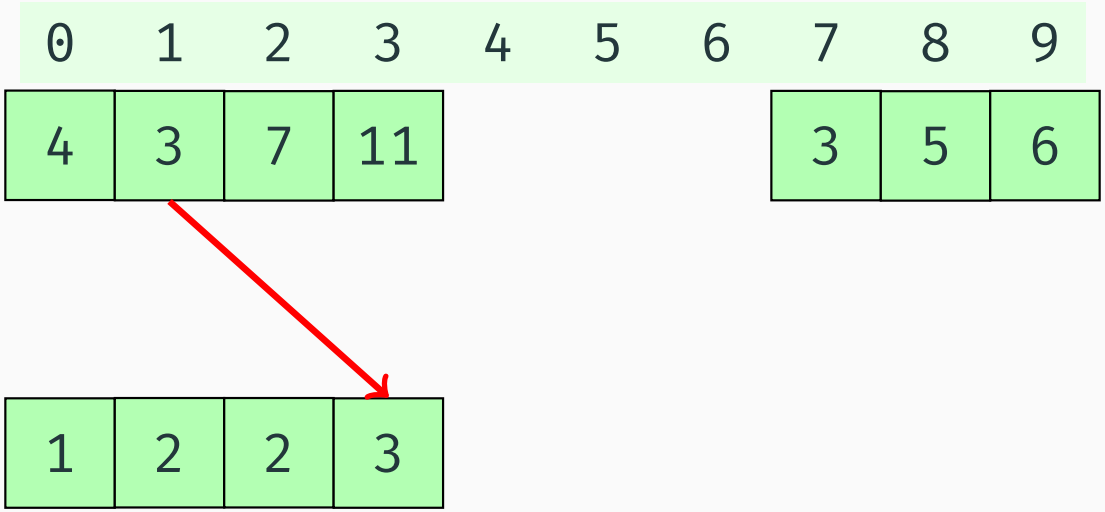
- first finding the smallest element in the input list, and moving it to the output list;
- then finding the next smallest value and does the same;
- continuing in this way until all the input elements have been selected and placed in the output list in the correct order.



Selection Sort

Selection Sort operates by

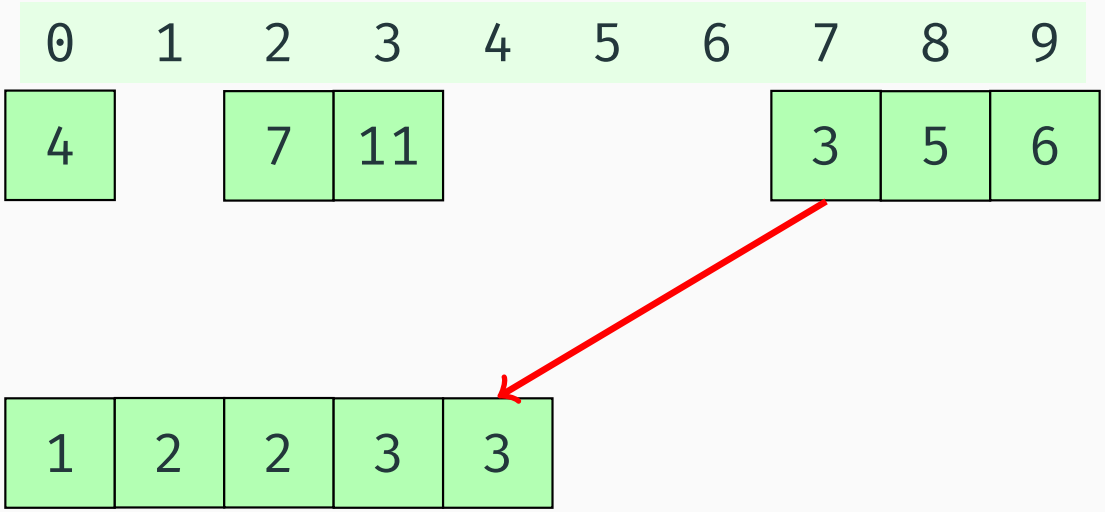
- first finding the smallest element in the input list, and moving it to the output list;
- then finding the next smallest value and does the same;
- continuing in this way until all the input elements have been selected and placed in the output list in the correct order.



Selection Sort

Selection Sort operates by

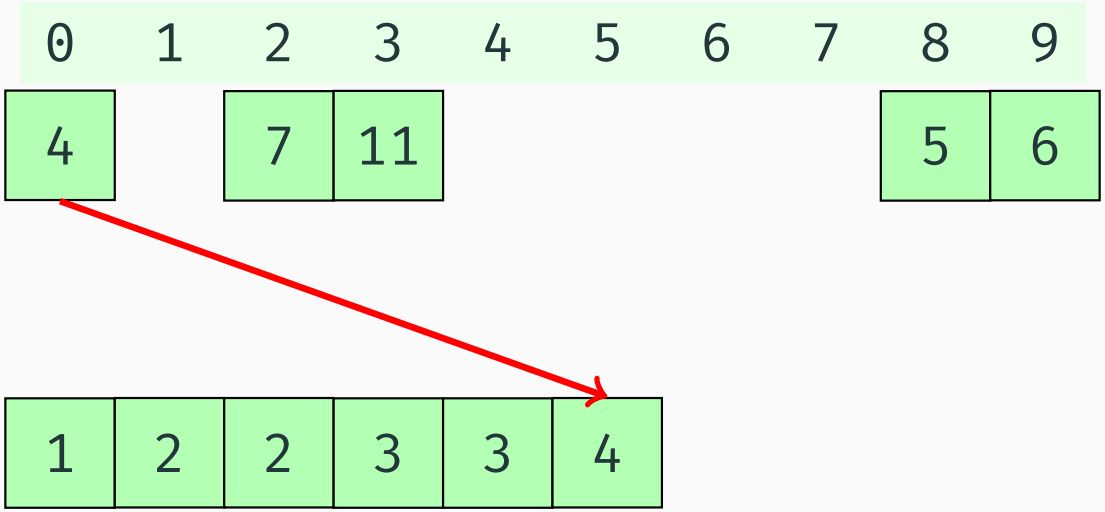
- first finding the smallest element in the input list, and moving it to the output list;
- then finding the next smallest value and does the same;
- continuing in this way until all the input elements have been selected and placed in the output list in the correct order.



Selection Sort

Selection Sort operates by

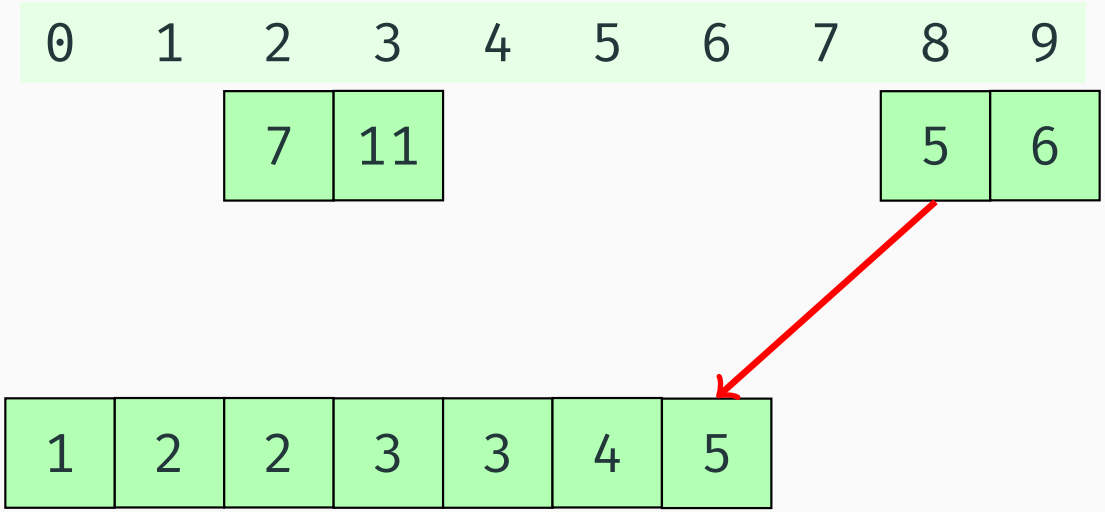
- first finding the smallest element in the input list, and moving it to the output list;
- then finding the next smallest value and does the same;
- continuing in this way until all the input elements have been selected and placed in the output list in the correct order.



Selection Sort

Selection Sort operates by

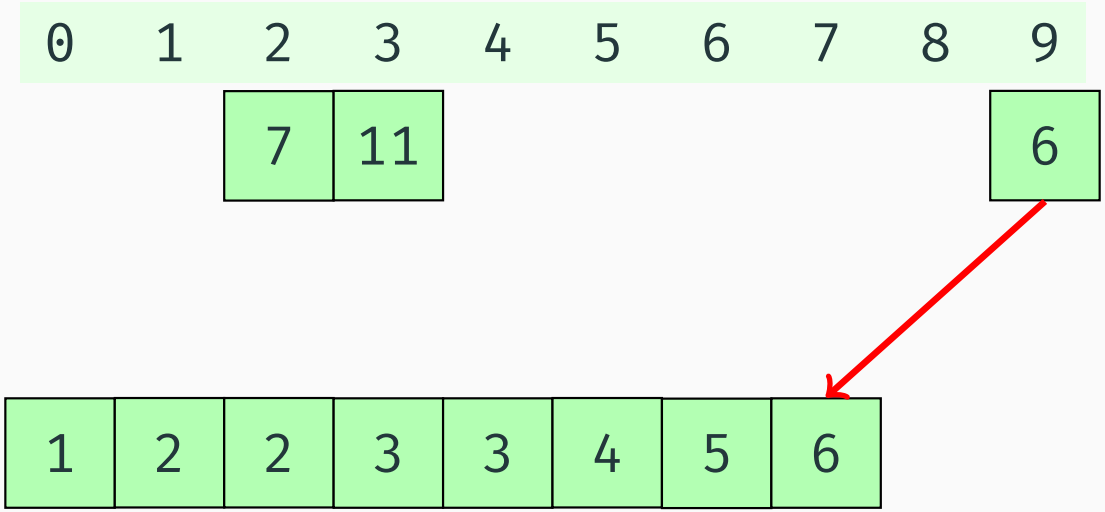
- first finding the smallest element in the input list, and moving it to the output list;
- then finding the next smallest value and does the same;
- continuing in this way until all the input elements have been selected and placed in the output list in the correct order.



Selection Sort

Selection Sort operates by

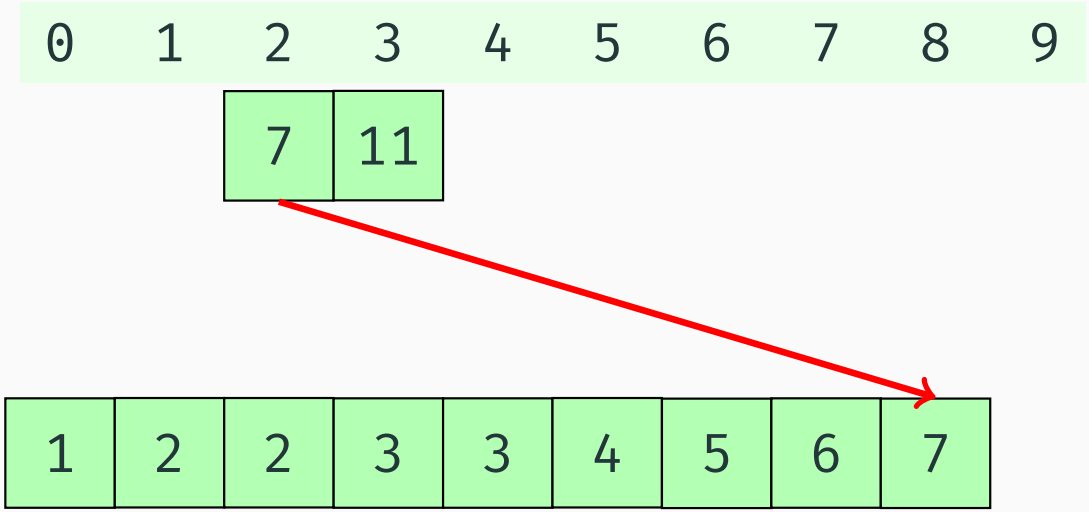
- first finding the smallest element in the input list, and moving it to the output list;
- then finding the next smallest value and does the same;
- continuing in this way until all the input elements have been selected and placed in the output list in the correct order.



Selection Sort

Selection Sort operates by

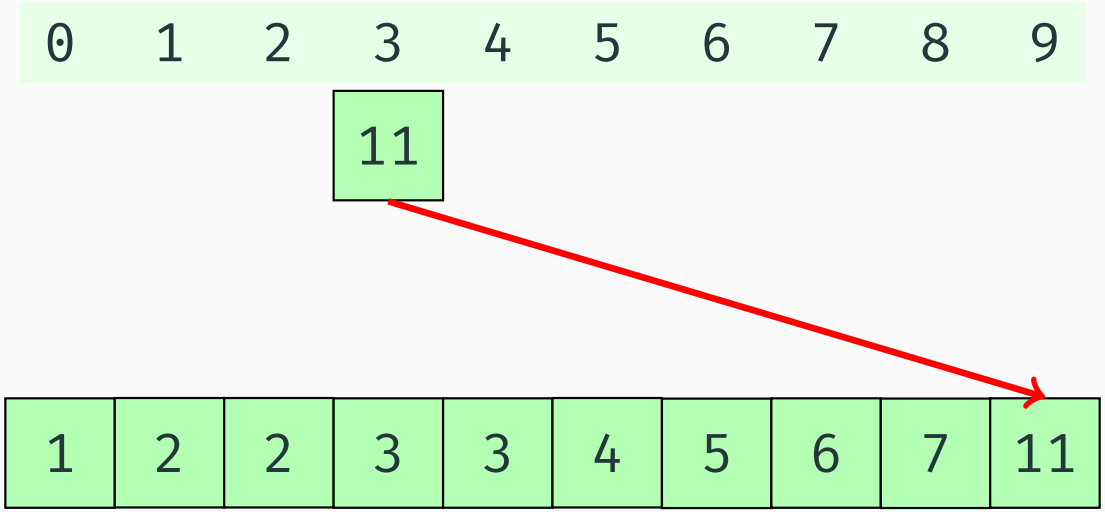
- first finding the smallest element in the input list, and moving it to the output list;
- then finding the next smallest value and does the same;
- continuing in this way until all the input elements have been selected and placed in the output list in the correct order.



Selection Sort

Selection Sort operates by

- first finding the smallest element in the input list, and moving it to the output list;
- then finding the next smallest value and does the same;
- continuing in this way until all the input elements have been selected and placed in the output list in the correct order.

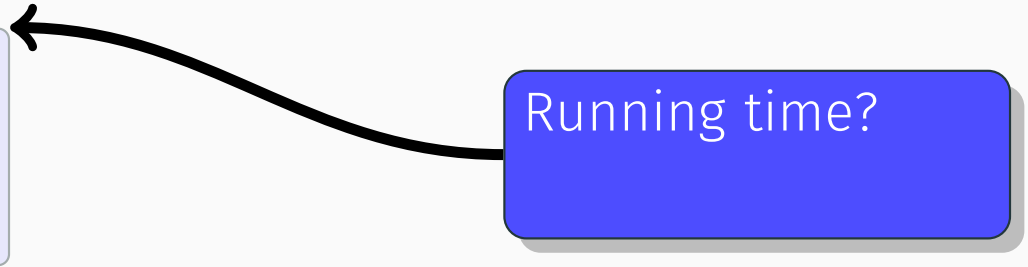


Selection Sort

```
1  for i = 0 to n-1
2  // A[0,...,i-1]: i smallest keys in sorted order.
3  // A[i,...,n-1]:s the remaining keys
4  jmin = i
5  for j = i+1 to n-1
6    if A[j] < A[jmin]
7      jmin = j
8  swap A[i] with A[jmin]
```

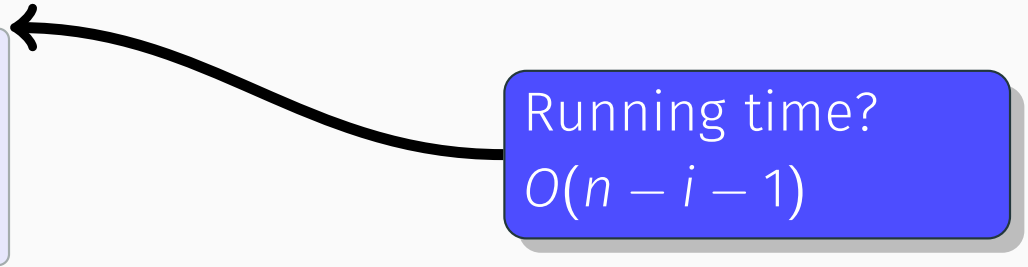
Selection Sort

```
1  for i = 0 to n-1
2  // A[0,...,i-1]: i smallest keys in sorted order.
3  // A[i,...,n-1]:s the remaining keys
4  jmin = i
5  for j = i+1 to n-1
6    if A[j] < A[jmin]
7      jmin = j
8  swap A[i] with A[jmin]
```



Selection Sort

```
1  for i = 0 to n-1
2  // A[0,...,i-1]: i smallest keys in sorted order.
3  // A[i,...,n-1]:s the remaining keys
4  jmin = i
5  for j = i+1 to n-1
6    if A[j] < A[jmin]
7      jmin = j
8  swap A[i] with A[jmin]
```



Selection Sort

```
1  for i = 0 to n-1
2  // A[0,...,i-1]: i smallest keys in sorted order.
3  // A[i,...,n-1]:s the remaining keys
4  jmin = i
5  for j = i+1 to n-1
6    if A[j] < A[jmin]
7      jmin = j
8  swap A[i] with A[jmin]
```

Running time?
$O(n - i - 1)$

Total running time:

$$T(n) = \sum_{i=0}^{n-1}(n - i - 1) = \sum_{i=0}^{n-1} i = O(n^2)$$

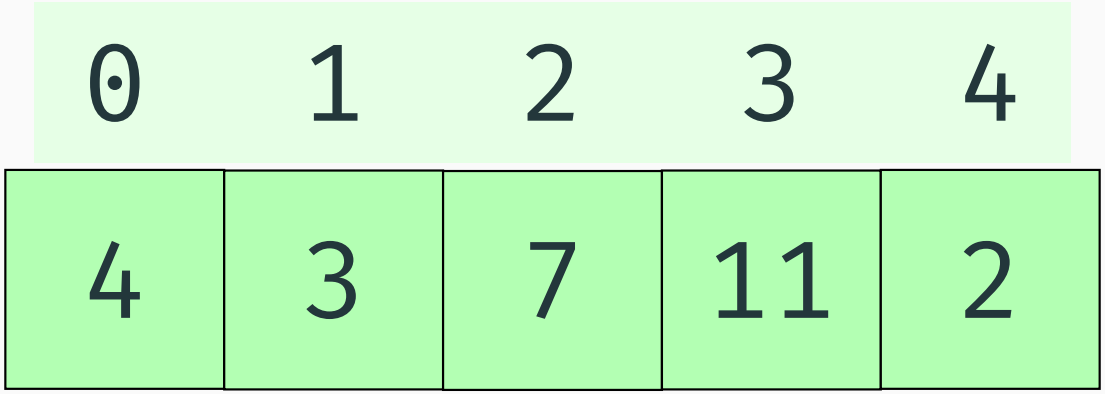
Comparison Sort

Bubble Sort

Bubble Sort

Bubble Sort operates:

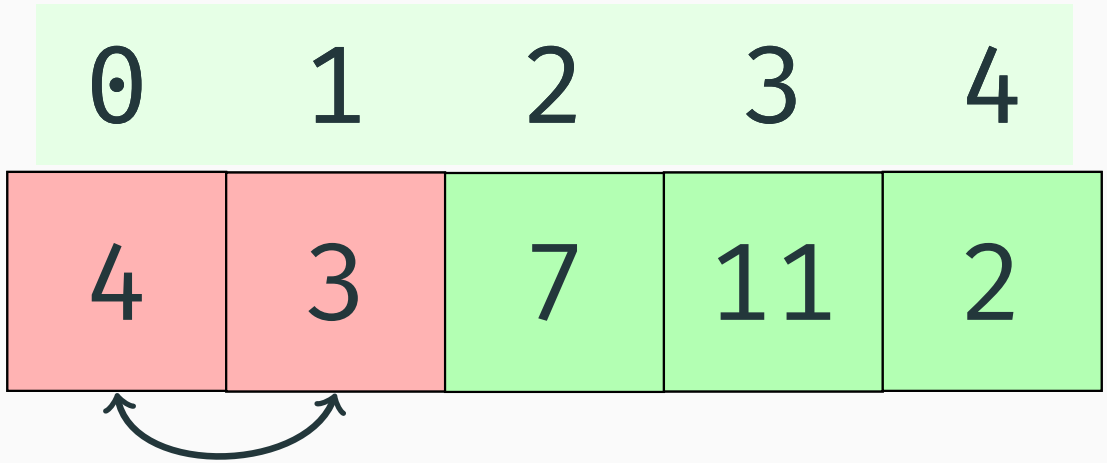
- Successively comparing adjacent elements, swap them if they are out of order;
- After the 1st pass, the largest element is in the correct position;
- continuing in this way until sort the entire array.



Bubble Sort

Bubble Sort operates:

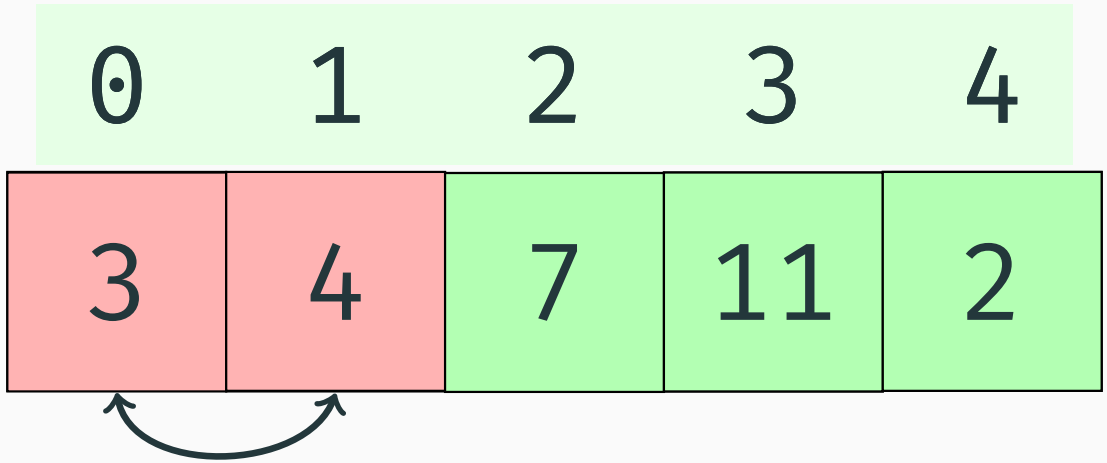
- Successively comparing adjacent elements, swap them if they are out of order;
- After the 1st pass, the largest element is in the correct position;
- continuing in this way until sort the entire array.



Bubble Sort

Bubble Sort operates:

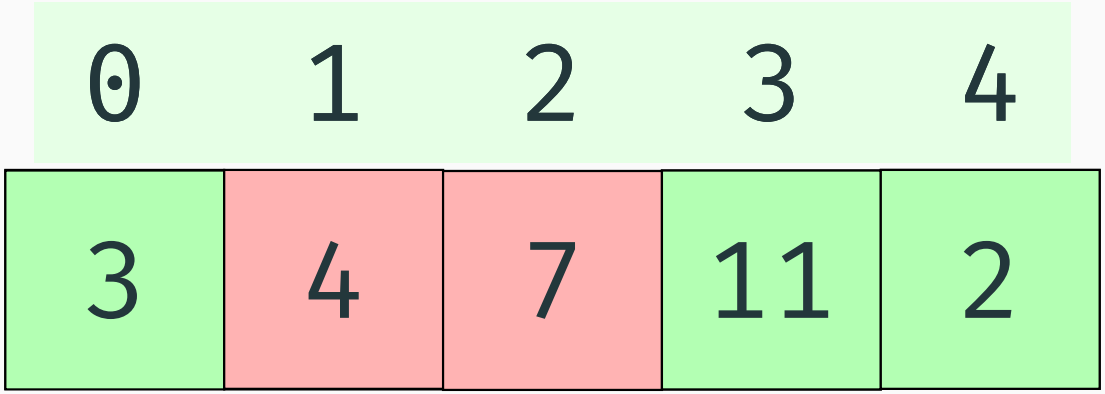
- Successively comparing adjacent elements, swap them if they are out of order;
- After the 1st pass, the largest element is in the correct position;
- continuing in this way until sort the entire array.



Bubble Sort

Bubble Sort operates:

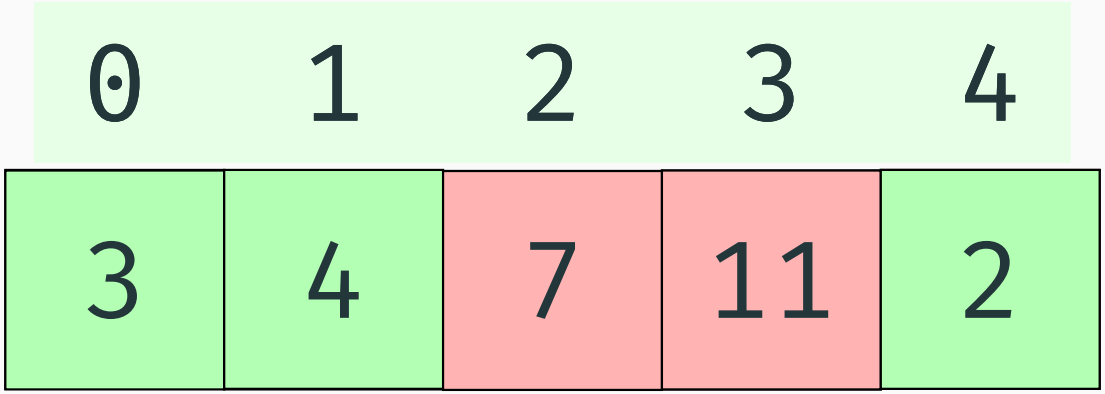
- Successively comparing adjacent elements, swap them if they are out of order;
- After the 1st pass, the largest element is in the correct position;
- continuing in this way until sort the entire array.



Bubble Sort

Bubble Sort operates:

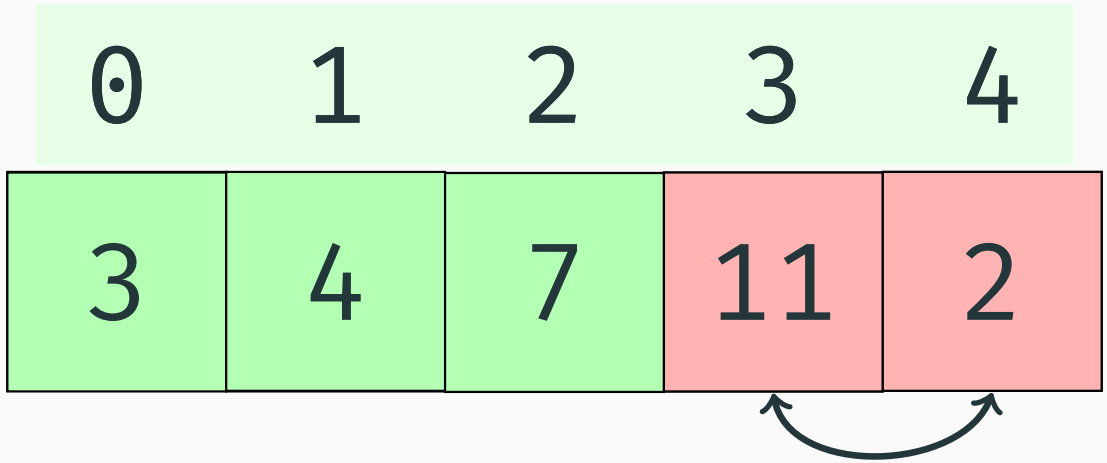
- Successively comparing adjacent elements, swap them if they are out of order;
- After the 1st pass, the largest element is in the correct position;
- continuing in this way until sort the entire array.



Bubble Sort

Bubble Sort operates:

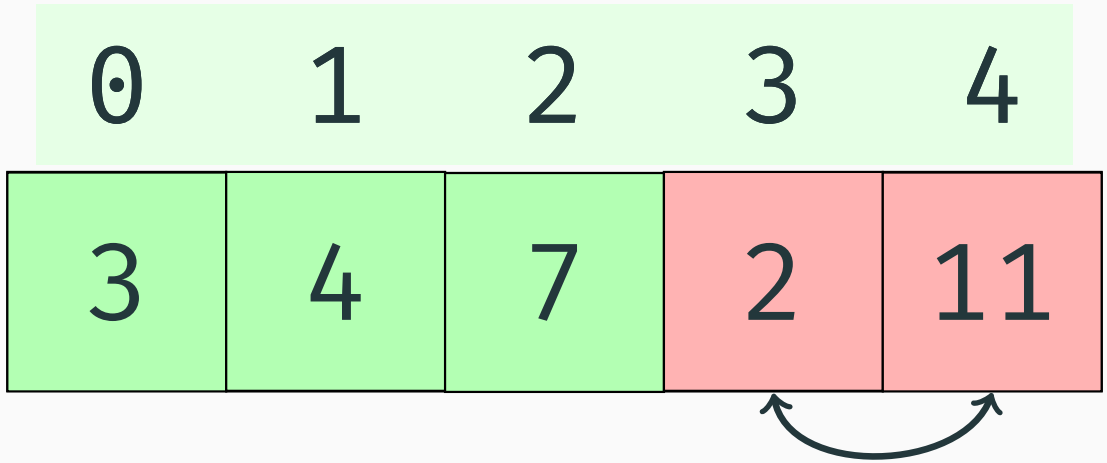
- Successively comparing adjacent elements, swap them if they are out of order;
- After the 1st pass, the largest element is in the correct position;
- continuing in this way until sort the entire array.



Bubble Sort

Bubble Sort operates:

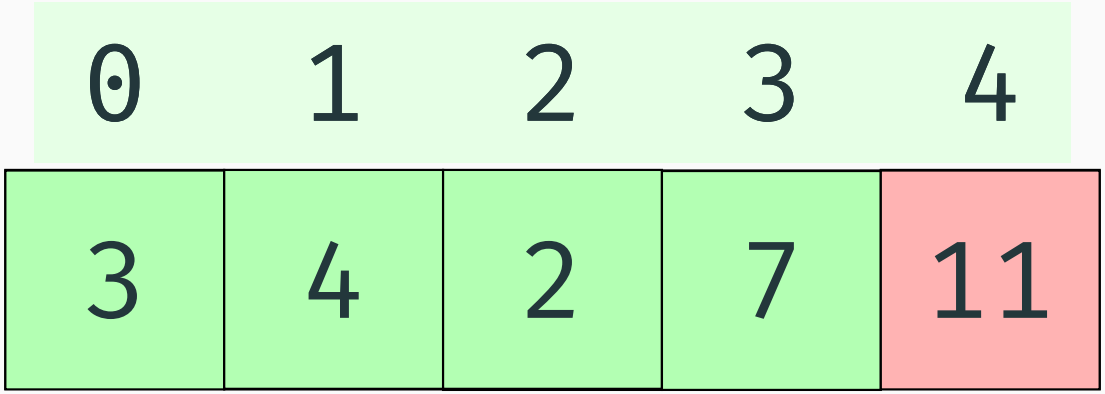
- Successively comparing adjacent elements, swap them if they are out of order;
- After the 1st pass, the largest element is in the correct position;
- continuing in this way until sort the entire array.



Bubble Sort

Bubble Sort operates:

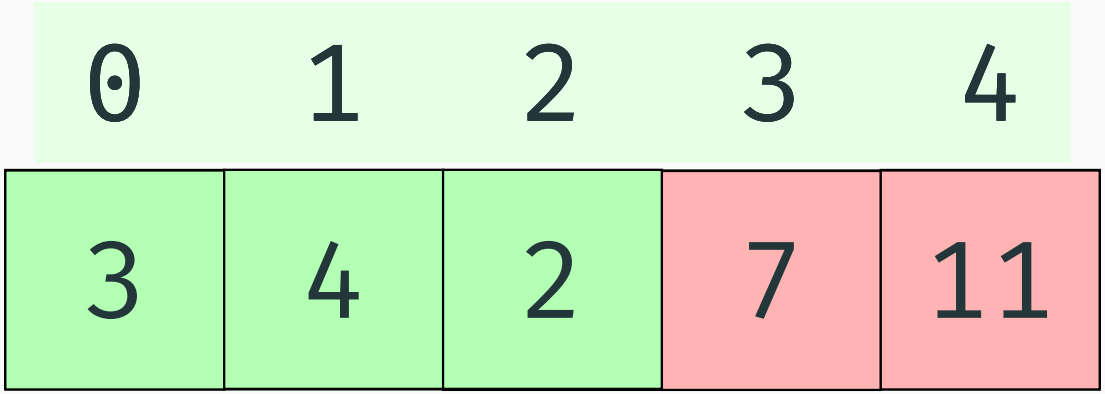
- Successively comparing adjacent elements, swap them if they are out of order;
- After the 1st pass, the largest element is in the correct position;
- continuing in this way until sort the entire array.



Bubble Sort

Bubble Sort operates:

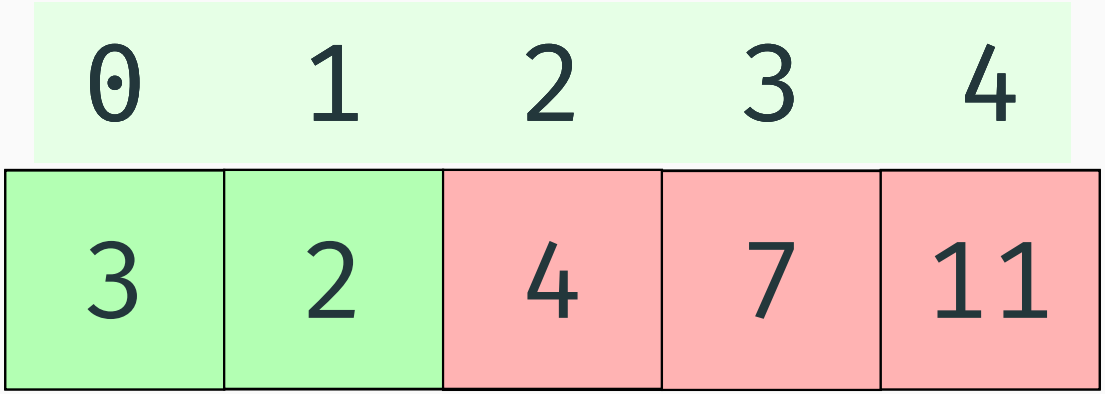
- Successively comparing adjacent elements, swap them if they are out of order;
- After the 1st pass, the largest element is in the correct position;
- continuing in this way until sort the entire array.



Bubble Sort

Bubble Sort operates:

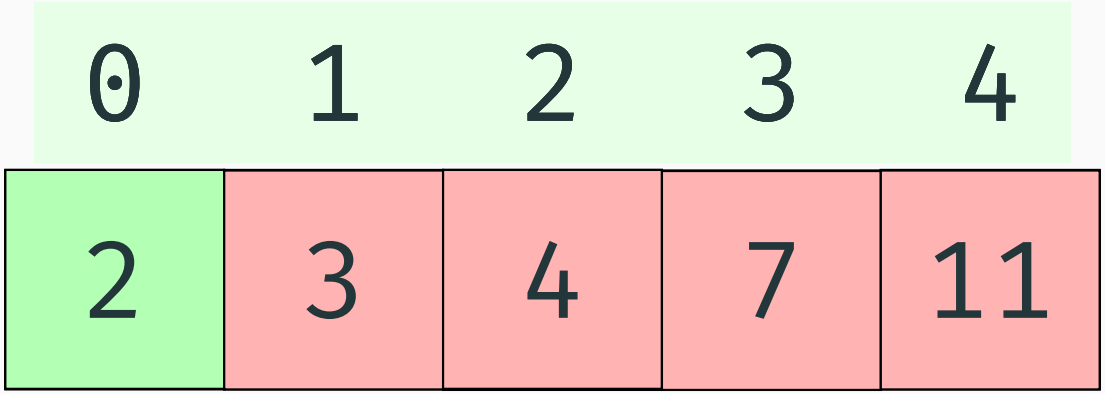
- Successively comparing adjacent elements, swap them if they are out of order;
- After the 1st pass, the largest element is in the correct position;
- continuing in this way until sort the entire array.



Bubble Sort

Bubble Sort operates:

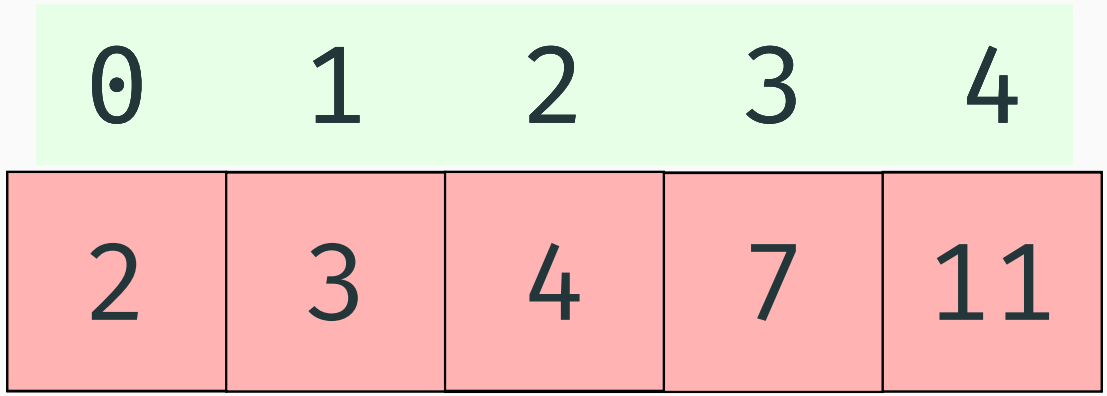
- Successively comparing adjacent elements, swap them if they are out of order;
- After the 1st pass, the largest element is in the correct position;
- continuing in this way until sort the entire array.



Bubble Sort

Bubble Sort operates:

- Successively comparing adjacent elements, swap them if they are out of order;
- After the 1st pass, the largest element is in the correct position;
- continuing in this way until sort the entire array.

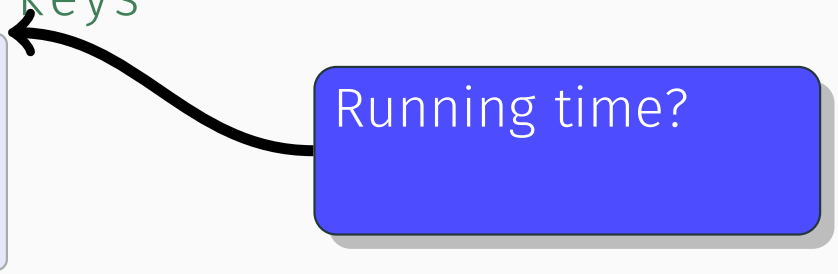


Bubble Sort

```
1  for i = n-1 downto 1
2  // A[i+1,...,n-1]:
3  //       n-i-1 largest keys in sorted order.
4  // A[0,...,i]: the remaining keys
5    for j = 0 to i-1
6      if A[j] > A[j + 1]
7        swap A[j] and A[j + 1]
```

Bubble Sort

```
1  for i = n-1 downto 1
2  // A[i+1,...,n-1]:
3  //       n-i-1 largest keys in sorted order.
4  // A[0,...,i]: the remaining keys
5    for j = 0 to i-1
6      if A[j] > A[j + 1]
7        swap A[j] and A[j + 1]
```



Bubble Sort

```
1  for i = n-1 downto 1
2  // A[i+1,...,n-1]:
3  //       n-i-1 largest keys in sorted order.
4  // A[0,...,i]: the remaining keys
5    for j = 0 to i-1
6      if A[j] > A[j + 1]
7        swap A[j] and A[j + 1]
```

Running time?
$O(i)$

Bubble Sort

```
1 for i = n-1 downto 1
2 // A[i+1,...,n-1]:
3 //       n-i-1 largest keys in sorted order.
4 // A[0,...,i]: the remaining keys
5   for j = 0 to i-1
6     if A[j] > A[j + 1]
7       swap A[j] and A[j + 1]
```

Running time?
$O(i)$

Total running time:

$$T(n) = \sum_{i=0}^{n-1} i = O(n^2)$$

Comparison Sort

Insertion Sort

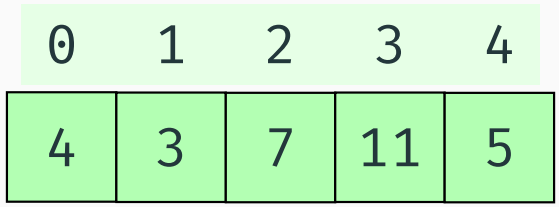
Insertion Sort

Like Selection Sort, Insertion Sort maintains two sublists:

- A left sublist containing sorted keys
- A right sublist containing the remaining unsorted keys

Unlike Selection Sort, the keys in the left sublist are

- not the smallest keys in the input list,
- but the first sorted keys in the input list.



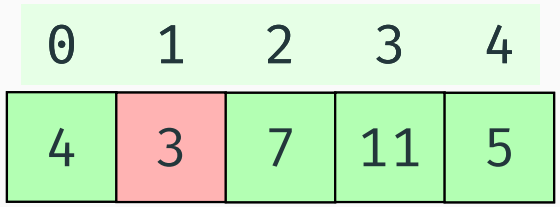
Insertion Sort

Like Selection Sort, Insertion Sort maintains two sublists:

- A left sublist containing sorted keys
- A right sublist containing the remaining unsorted keys

Unlike Selection Sort, the keys in the left sublist are

- not the smallest keys in the input list,
- but the first sorted keys in the input list.




```
key = 3
```

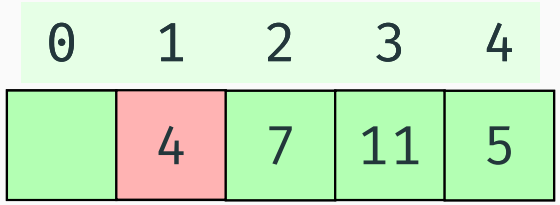
Insertion Sort

Like Selection Sort, Insertion Sort maintains two sublists:

- A left sublist containing sorted keys
- A right sublist containing the remaining unsorted keys

Unlike Selection Sort, the keys in the left sublist are

- not the smallest keys in the input list,
- but the first sorted keys in the input list.



`key = 3`

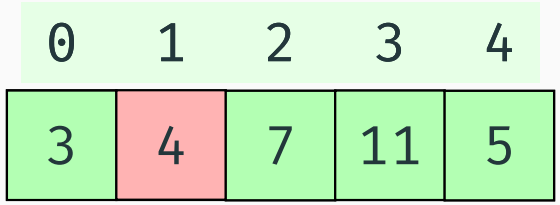
Insertion Sort

Like Selection Sort, Insertion Sort maintains two sublists:

- A left sublist containing sorted keys
- A right sublist containing the remaining unsorted keys

Unlike Selection Sort, the keys in the left sublist are

- not the smallest keys in the input list,
- but the first sorted keys in the input list.




```
key = 3
```

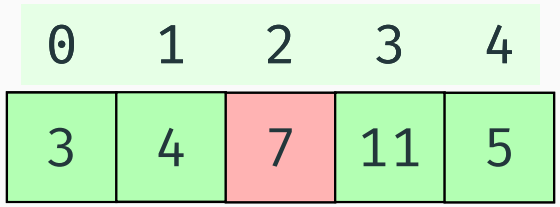
Insertion Sort

Like Selection Sort, Insertion Sort maintains two sublists:

- A left sublist containing sorted keys
- A right sublist containing the remaining unsorted keys

Unlike Selection Sort, the keys in the left sublist are

- not the smallest keys in the input list,
- but the first sorted keys in the input list.



`key = 7`

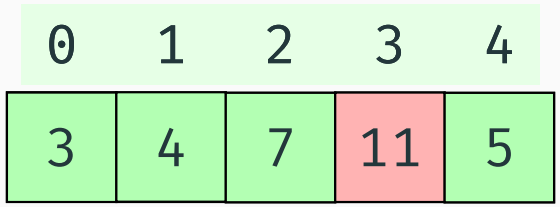
Insertion Sort

Like Selection Sort, Insertion Sort maintains two sublists:

- A left sublist containing sorted keys
- A right sublist containing the remaining unsorted keys

Unlike Selection Sort, the keys in the left sublist are

- not the smallest keys in the input list,
- but the first sorted keys in the input list.



`key = 11`

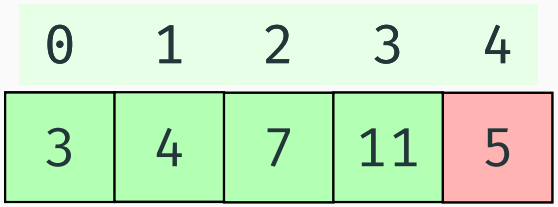
Insertion Sort

Like Selection Sort, Insertion Sort maintains two sublists:

- A left sublist containing sorted keys
- A right sublist containing the remaining unsorted keys

Unlike Selection Sort, the keys in the left sublist are

- not the smallest keys in the input list,
- but the first sorted keys in the input list.



`key = 5`

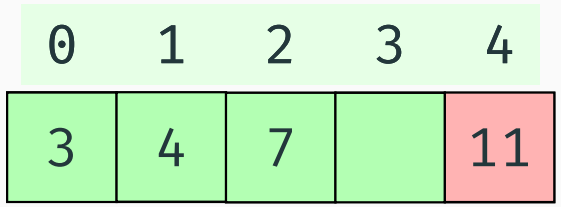
Insertion Sort

Like Selection Sort, Insertion Sort maintains two sublists:

- A left sublist containing sorted keys
- A right sublist containing the remaining unsorted keys

Unlike Selection Sort, the keys in the left sublist are

- not the smallest keys in the input list,
- but the first sorted keys in the input list.



`key = 5`

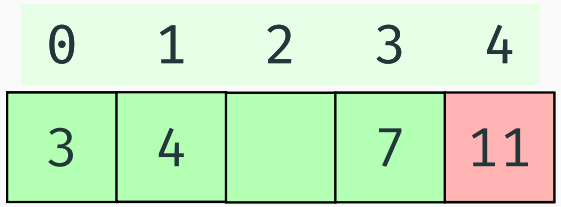
Insertion Sort

Like Selection Sort, Insertion Sort maintains two sublists:

- A left sublist containing sorted keys
- A right sublist containing the remaining unsorted keys

Unlike Selection Sort, the keys in the left sublist are

- not the smallest keys in the input list,
- but the first sorted keys in the input list.



`key = 5`

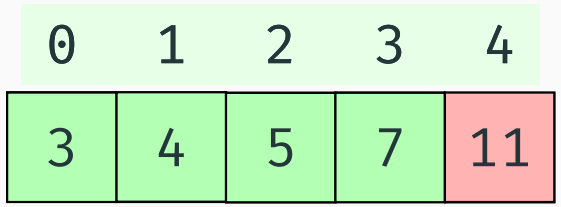
Insertion Sort

Like Selection Sort, Insertion Sort maintains two sublists:

- A left sublist containing sorted keys
- A right sublist containing the remaining unsorted keys

Unlike Selection Sort, the keys in the left sublist are

- not the smallest keys in the input list,
- but the first sorted keys in the input list.



`key = 5`

Insertion Sort

```
1  for i = 1 to n-1
2  // A[0,...,i-1]:
3  //      first i keys of the input in sorted order.
4  // A[i,...,n-1]: the remaining keys
5    key = A[i]
6    j = i
7    while j > 0 & A[j-1] > key
8      A[j] = A[j-1]
9      j = j-1
10   A[j] = key
```

Insertion Sort

```
1  for i = 1 to n-1
2  // A[0,...,i-1]:
3  //      first i keys of the input in sorted order.
4  // A[i,...,n-1]: the remaining keys
5    key = A[i]
6    j = i
7    while j > 0 & A[j-1] > key
8      A[j] = A[j-1]
9      j = j-1
10   A[j] = key
```

Running time?

Insertion Sort

```
1  for i = 1 to n-1
2  // A[0,...,i-1]:
3  //      first i keys of the input in sorted order.
4  // A[i,...,n-1]: the remaining keys
5    key = A[i]
6    j = i
7    while j > 0 & A[j-1] > key
8      A[j] = A[j-1]
9      j = j-1
10   A[j] = key
```

Running time?
$O(i)$

Insertion Sort

```
1  for i = 1 to n-1
2  // A[0,...,i-1]:
3  //      first i keys of the input in sorted order.
4  // A[i,...,n-1]: the remaining keys
5    key = A[i]
6    j = i
7    while j > 0 & A[j-1] > key
8      A[j] = A[j-1]
9      j = j-1
10   A[j] = key
```

Running time?
$O(i)$

Total running time:

$$T(n) = \sum_{i=0}^{n-1} i = O(n^2)$$

Selection sort vs Bubble sort vs Insertion sort

Selection Sort, Bubble Sort, Insertion Sort have $O(n^2)$ running time.

However, all **can** also easily be designed to

- Sort in place
- Stable sort

Comparison Sort

Merge Sort

Recursive Sorts

Divide-and-conquer is a general algorithm design paradigm:

- **Divide**: divide the input data S in two disjoint subsets S_1 and S_2
- **Recur**: solve the subproblems associated with S_1 and S_2
- **Conquer**: combine the solutions for S_1 and S_2 to solve S

The base case for the recursion is a subproblem of size 0 or 1

Recursive Sorts: Given list of objects to be sorted

- **Split** the list into two sublists.
- **Recursively** have two friends sort the two sublists.
- **Combine** the two sorted sublists into one entirely sorted list.

Examples: Merge Sort, Quick Sort, ...

Merge Sort

Merge-sort is a sorting algorithm based on the divide-and-conquer paradigm

Merge Sort was invented by *John von Neumann*, one of the pioneers of computing, in 1945.

```
1  Algorithm mergeSort(S)
2  // Input: sequence S with n elements
3  // Output: sequence S sorted
4  if S.size() > 1
5    (S1, S2) = partition(S, n/2) // Divide
6    mergeSort(S1)                // Recur
7    mergeSort(S2)                // Recur
8    S = merge(S1, S2)            // Conquer
```

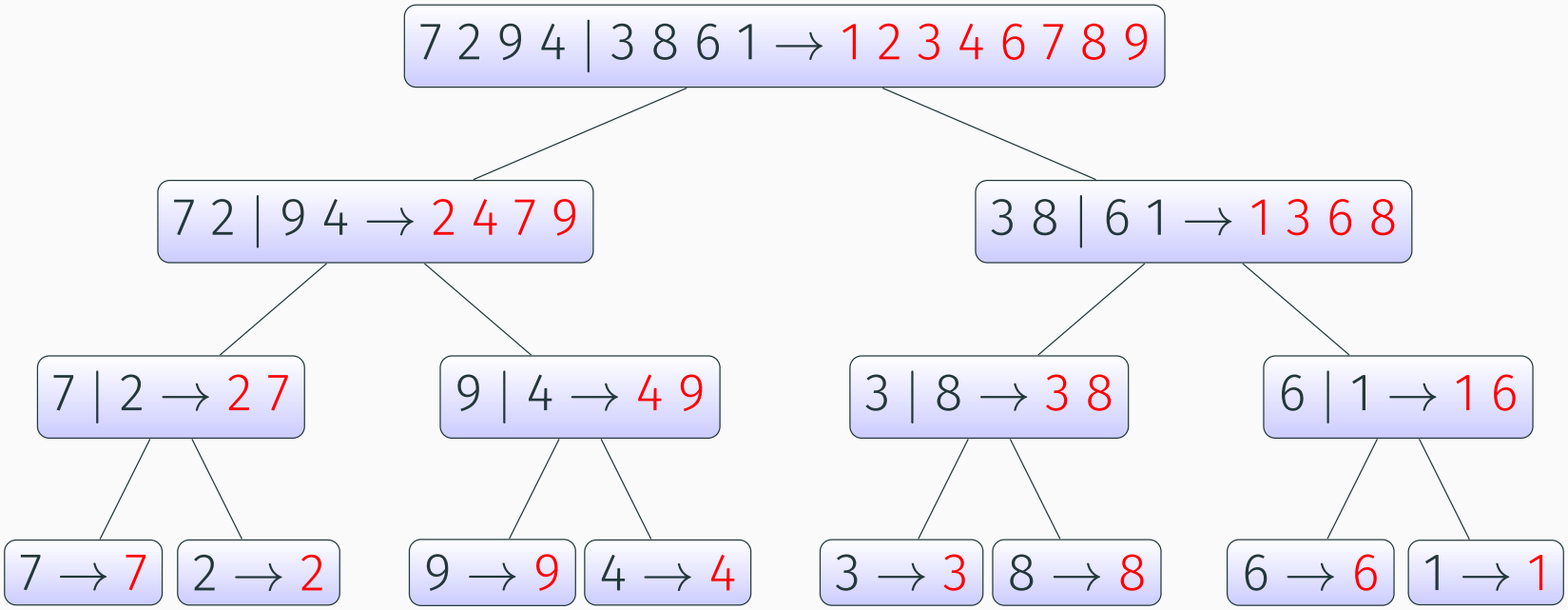
Merging Two Sorted Sequences

```
1  Algorithm merge(A, B)
2  // Input sequences A and B with n/2 elements each
3  // Output sorted sequence of A  B
4  S = empty sequence
5  while !A.isEmpty() and !B.isEmpty()
6    if A.first().element() < B.first().element()
7      S.addLast(A.remove(A.first()))
8    else
9      S.addLast(B.remove(B.first()))
10
11 while !A.isEmpty()
12     S.addLast(A.remove(A.first()))
13 while !B.isEmpty()
14     S.addLast(B.remove(B.first()))
15 return S
```

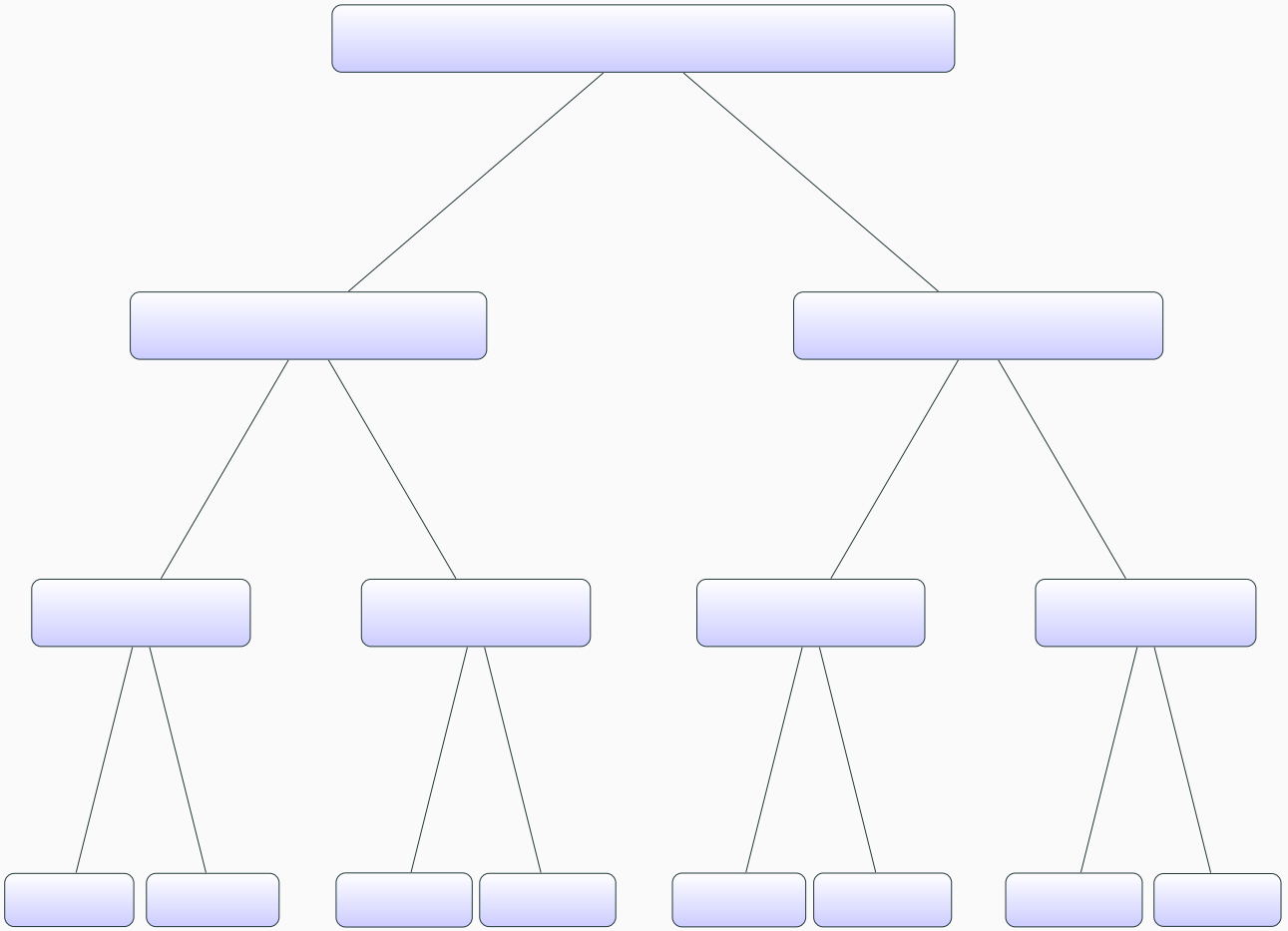
Merging Sort Tree

Merge-Sort Tree:

- each node represents a recursive call of merge-sort and stores unsorted sequence before the execution and its partition sorted sequence at the end of the execution
- the root is the initial call
- the leaves are calls on subsequences of size 0 or 1

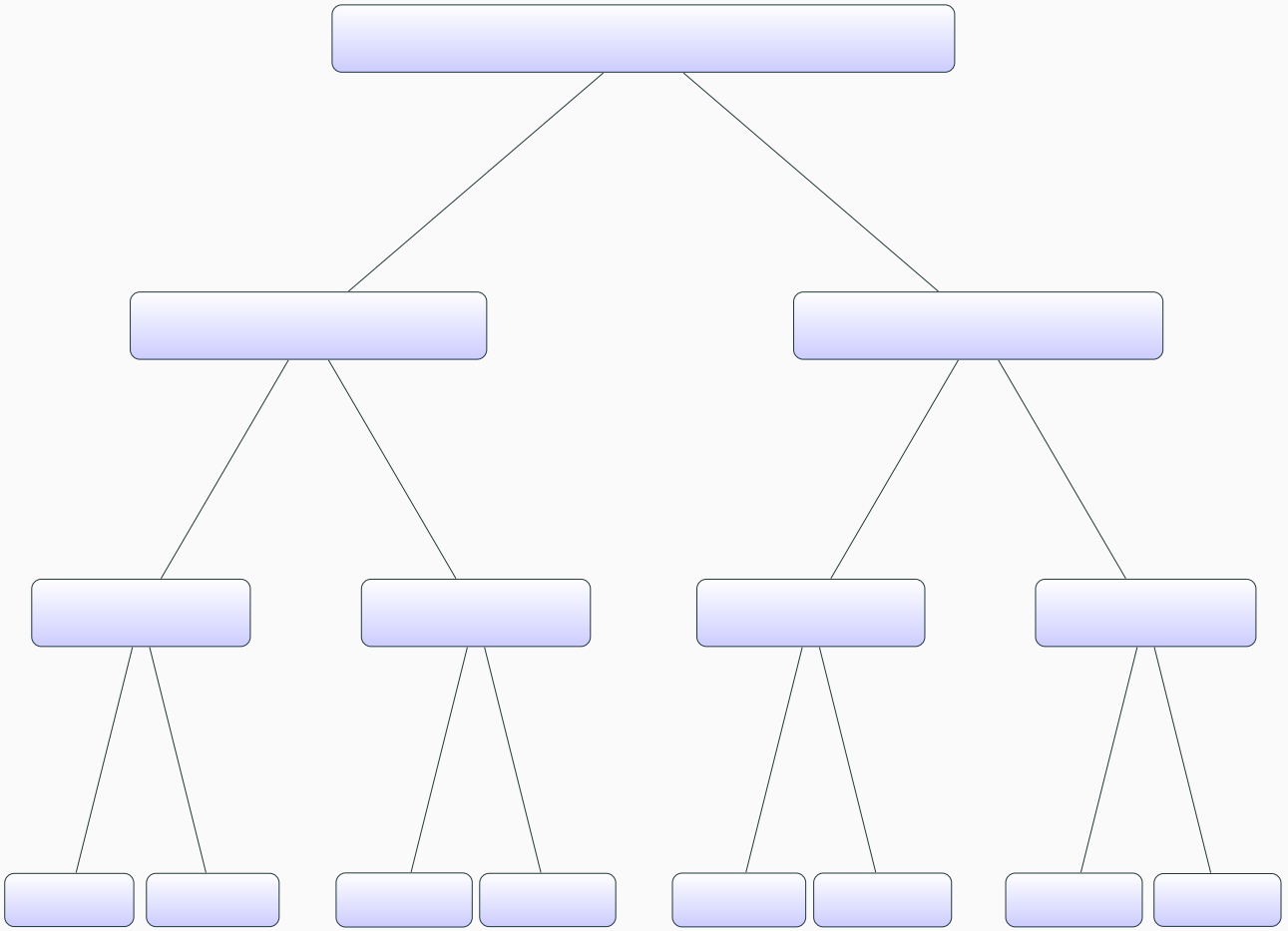


Analysis of Merge-Sort



depth	♯seq	size
0	1	n
1	2	$n/2$
i	2^i	$n/2^i$
...	...	...

Analysis of Merge-Sort

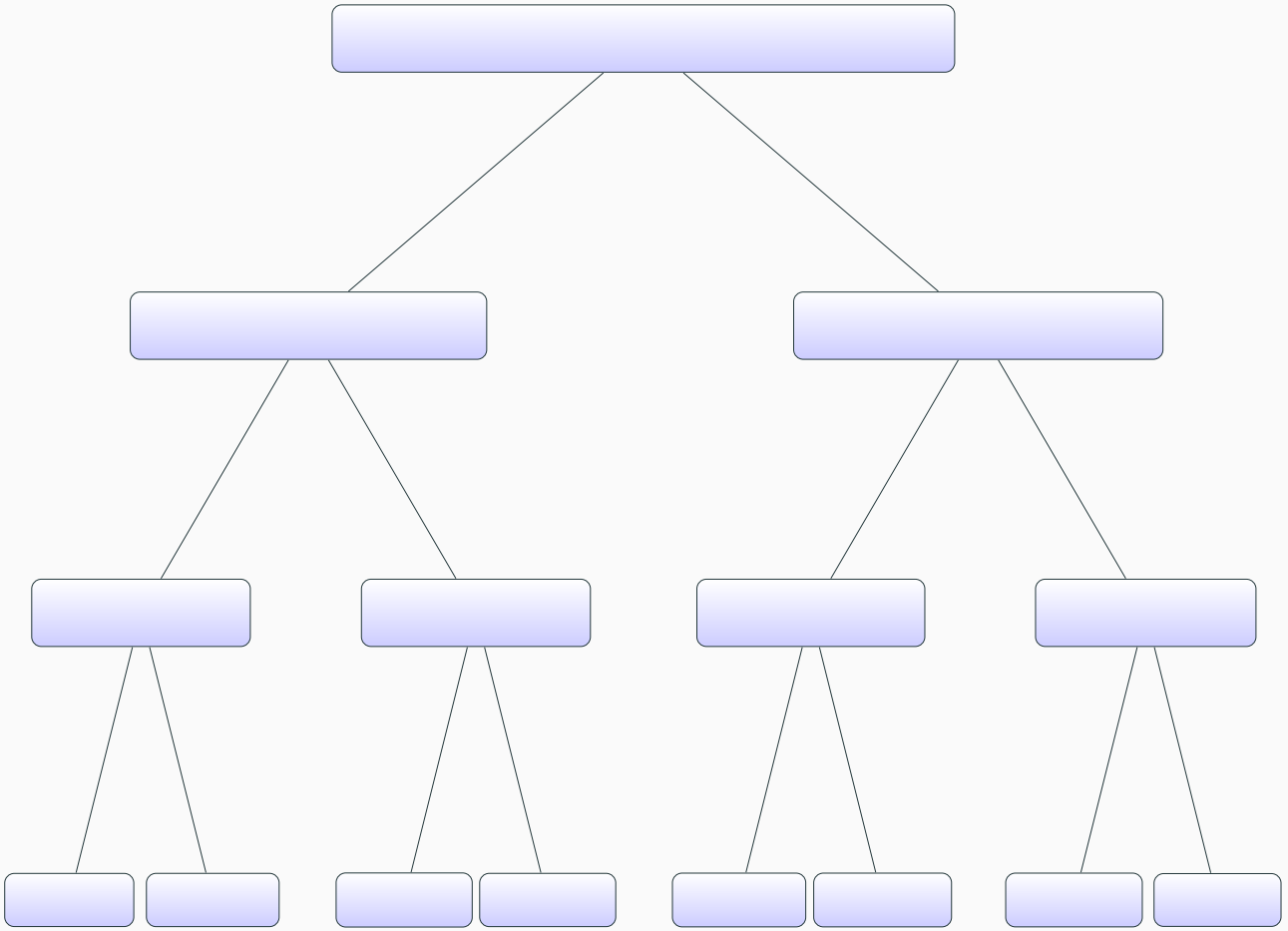


depth	♯seq	size
0	1	n
1	2	$n/2$
i	2^i	$n/2^i$
...	...	...

The height h of the merge-sort tree is $O(\log n)$

- at each recursive call we divide the sequence in half.

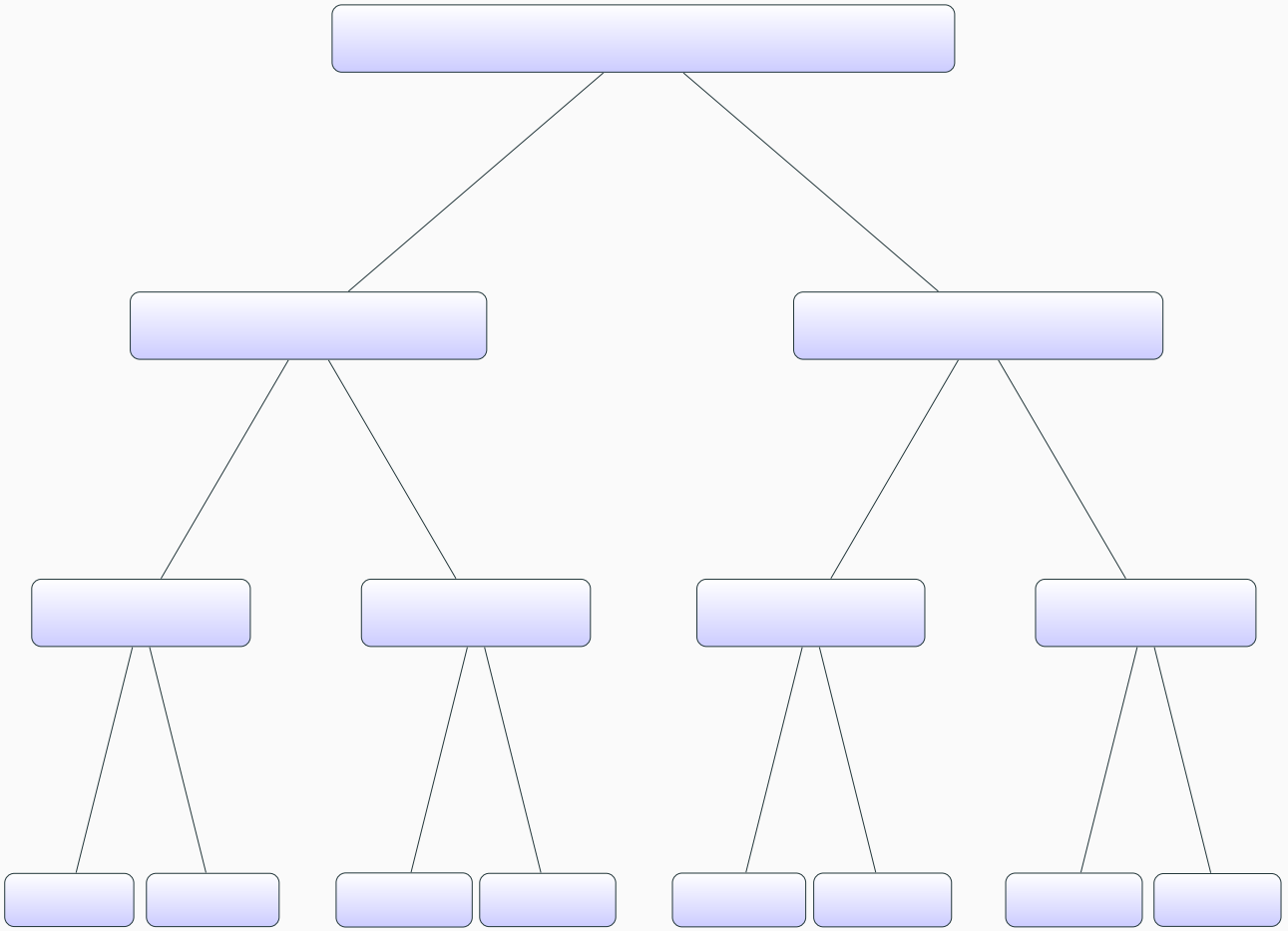
Analysis of Merge-Sort



The overall amount or work done at the nodes of depth i is $O(n)$

- we partition and merge 2^i sequences of size $n/2^i$

Analysis of Merge-Sort



depth	♯seq	size
0	1	n
1	2	$n/2$
i	2^i	$n/2^i$
...	...	...

Thus, the total running time of merge-sort is $O(n \log n)$!

More Discussion on Merge Sort

Sort in place?

More Discussion on Merge Sort

Sort in place?

YES

Normally, merging is not in-place: new memory must be allocated to hold S. It is possible to do in-place merging using linked lists.

- Code is more complicated
- Only changes memory usage by a constant factor

More Discussion on Merge Sort

Sort in place?

YES

Normally, merging is not in-place: new memory must be allocated to hold S. It is possible to do in-place merging using linked lists.

- Code is more complicated
- Only changes memory usage by a constant factor

Stable?

More Discussion on Merge Sort

Sort in place?

YES

Normally, merging is not in-place: new memory must be allocated to hold S. It is possible to do in-place merging using linked lists.

- Code is more complicated
- Only changes memory usage by a constant factor

Stable?

YES

Comparison Sort

Heap Sort

Heap Sort

Invented by *Williams & Floyd* in 1964

Heap Sort:

- Build an array-based (max) heap
- Iteratively call removeMax() to extract the keys in descending order

Heap Sort

Invented by *Williams & Floyd* in 1964

Heap Sort:

- Build an array-based (max) heap
- Iteratively call removeMax() to extract the keys in descending order

Running time?:

Heap Sort

Invented by *Williams & Floyd* in 1964

Heap Sort:

- Build an array-based (max) heap
- Iteratively call removeMax() to extract the keys in descending order

Running time?:

$$O(n \log n) \text{ worst case}$$

Heap Sort

Invented by *Williams & Floyd* in 1964

Heap Sort:

- Build an array-based (max) heap
- Iteratively call removeMax() to extract the keys in descending order

Running time?:

$$O(n \log n) \text{ worst case}$$

Sorts in place?:

Heap Sort

Invented by *Williams & Floyd* in 1964

Heap Sort:

- Build an array-based (max) heap
- Iteratively call removeMax() to extract the keys in descending order

Running time?:

$$O(n \log n) \text{ worst case}$$

Sorts in place?:

YES

Heap Sort

Invented by *Williams & Floyd* in 1964

Heap Sort:

- Build an array-based (max) heap
- Iteratively call removeMax() to extract the keys in descending order

Running time?:

$$O(n \log n) \text{ worst case}$$

Sorts in place?:

YES

Stable?:

Heap Sort

Invented by *Williams & Floyd* in 1964

Heap Sort:

- Build an array-based (max) heap
- Iteratively call removeMax() to extract the keys in descending order

Running time?:

$$O(n \log n) \text{ worst case}$$

Sorts in place?:

YES

Stable?:

NO as heap operations may disorder ties

Heap Sort

Invented by *Williams & Floyd* in 1964

Heap Sort:

- Build an array-based (max) heap
- Iteratively call removeMax() to extract the keys in descending order

Running time?:

$$O(n \log n) \text{ worst case}$$

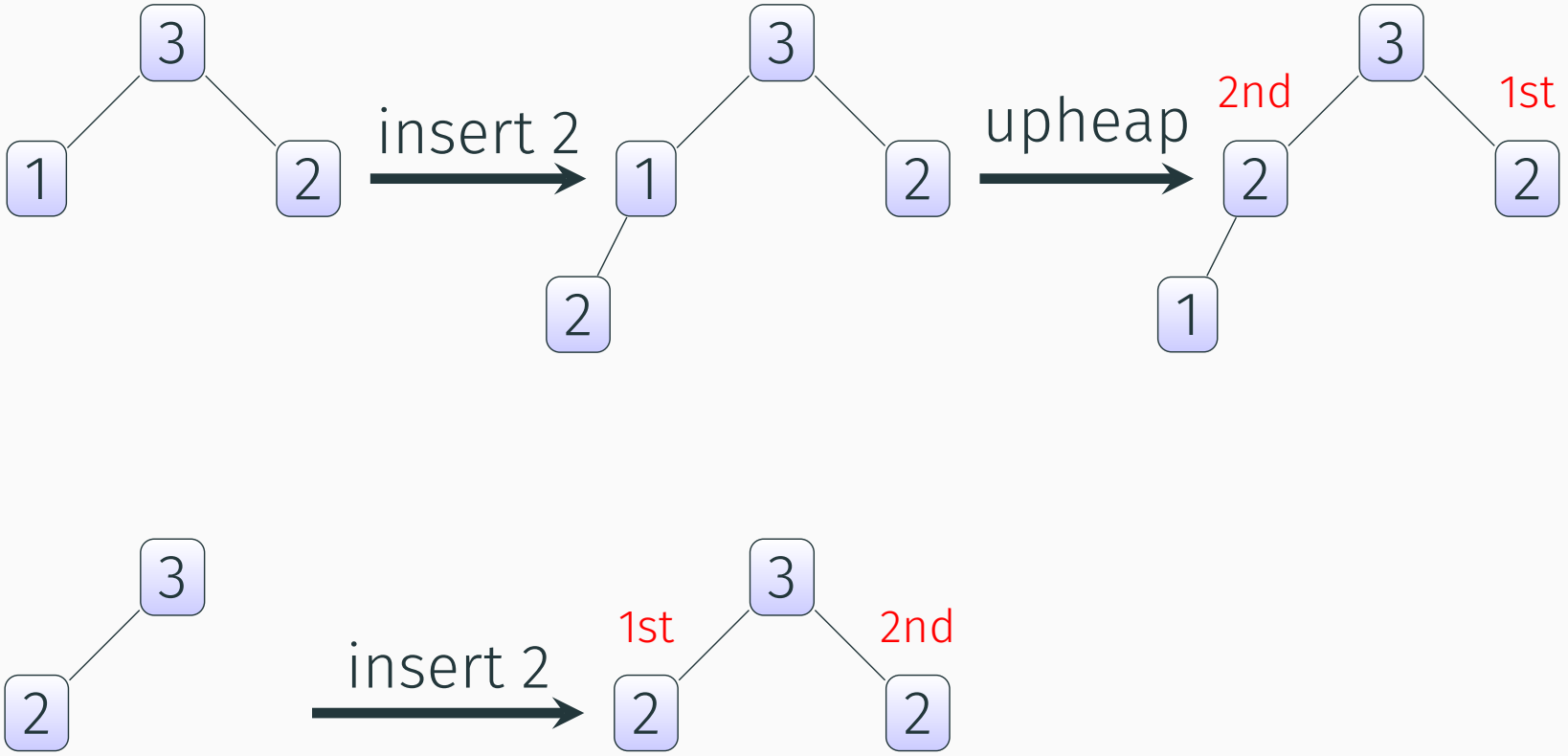
Sorts in place?:

YES

Stable?:

NO as heap operations may disorder ties

Heapsort is Not Stable



When we call the method removeMax(), which side should we go?

Comparison Sort

Quick Sort

Quick Sort

Invented by *C.A.R. Hoare* in 1960

Quick-sort is a divide-and-conquer algorithm

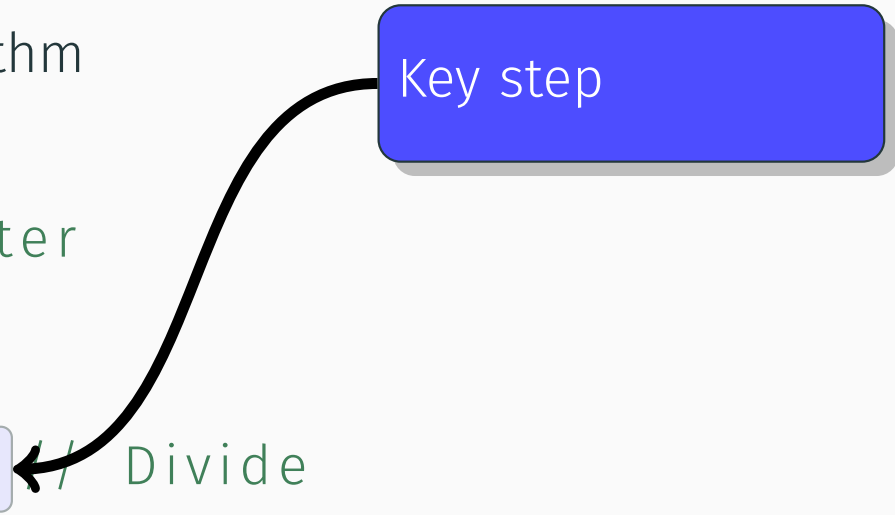
```
1  Algorithm QuickSort(S)
2  // L: less; E: equal; G: greater
3  // p: pivot or position
4  if S.size() > 1
5    (L, E, G) = Partition(S, p) // Divide
6    QuickSort(L)  //Recur: Small elements are sorted
7    QuickSort(G)  //Recur: Large elements are sorted
8    S = (L, E, G) //Conquer: Thus input is sorted
```

Quick Sort

Invented by *C.A.R. Hoare* in 1960

Quick-sort is a divide-and-conquer algorithm

```
1  Algorithm QuickSort(S)
2  // L: less; E: equal; G: greater
3  // p: pivot or position
4  if S.size() > 1
5    (L, E, G) = Partition(S, p) // Divide
6    QuickSort(L)  //Recur: Small elements are sorted
7    QuickSort(G)  //Recur: Large elements are sorted
8    S = (L, E, G) //Conquer: Thus input is sorted
```

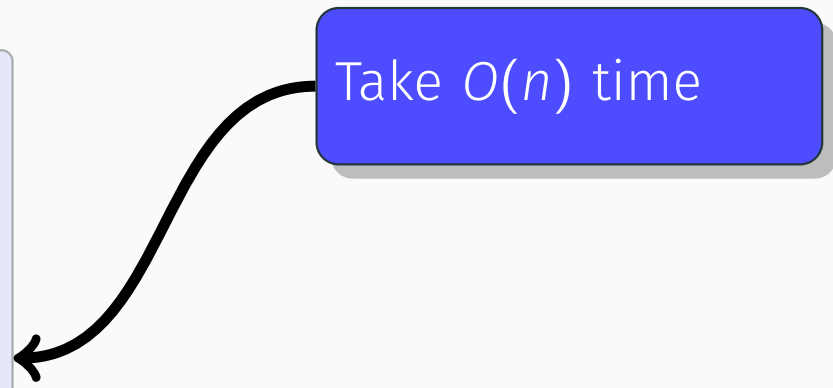


Partition

```
1  Algorithm partition(S, p)
2  //  Input: sequence S, position p of pivot
3  //  Output: subsequences L, E, G of the elements of S
       less than, equal to, or greater than the pivot,
       resp.
4  L, E, G <- empty sequences
5  while !S.isEmpty()
6    y <- S.remove(S.first())
7    if y < p
8      L.addLast(y)
9    else if y = p
10     E.addLast(y)
11   else // y > p
12     G.addLast(y)
13 return L, E, G
```

Partition

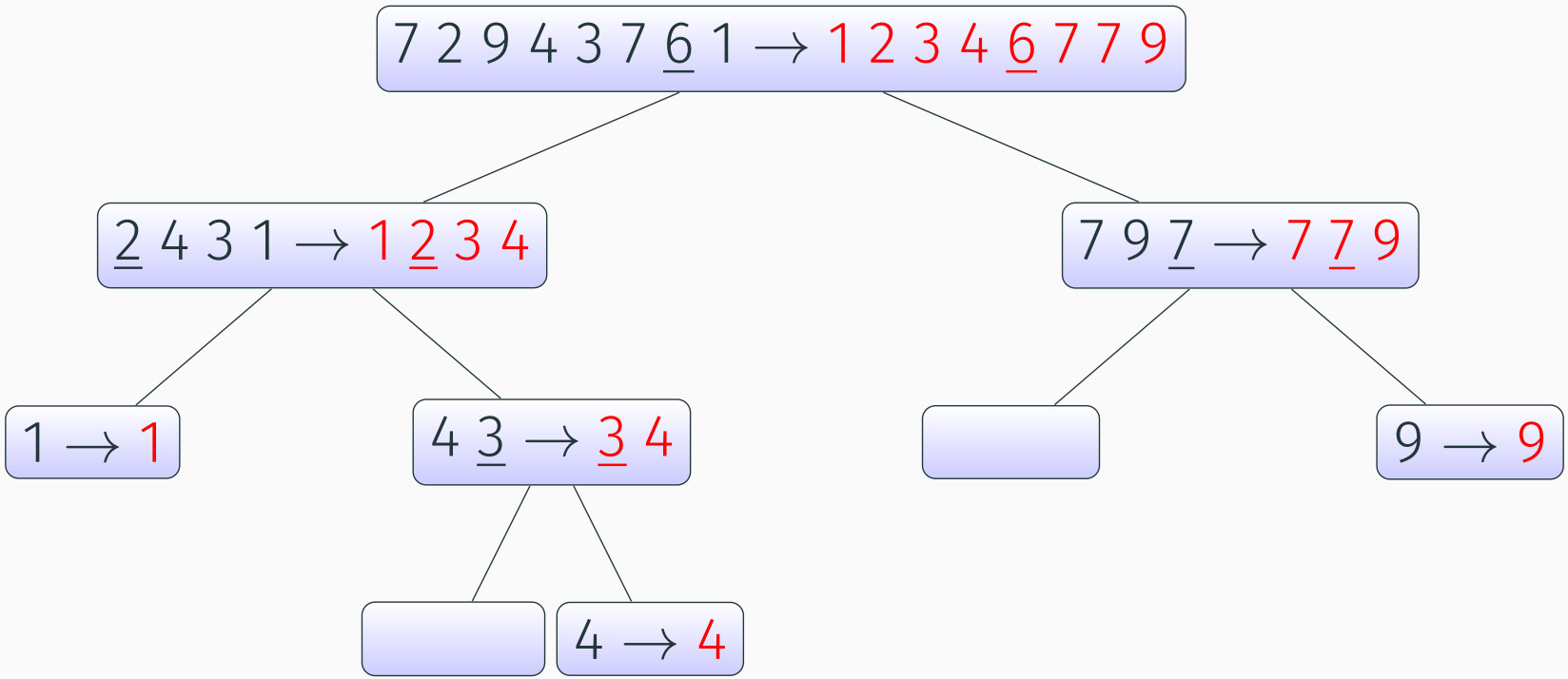
```
1  Algorithm partition(S, p)
2  //  Input: sequence S, position p of pivot
3  //  Output: subsequences L, E, G of the elements of S
       less than, equal to, or greater than the pivot,
       resp.
4  L, E, G <- empty sequences
5  while !S.isEmpty()
6    y <- S.remove(S.first())
7    if y < p
8      L.addLast(y)
9    else if y = p
10     E.addLast(y)
11   else // y > p
12     G.addLast(y)
13 return L, E, G
```



Merging Sort Tree

Quick-Sort Tree:

- each node represents a recursive call of quick-sort and stores
  - unsorted sequence before the execution and its partition
  - sorted sequence at the end of the execution
- the root is the initial call
- the leaves are calls on subsequences of size 0 or 1



Running Time of Quick Sort

The worst case for quick-sort occurs when the pivot is the unique minimum or maximum element

One of L and G has size $n - 1$ and the other has size 0

The running time is proportional to the sum

$$n + (n - 1) + \ldots + 2 + 1$$

Thus, the **worst-case** running time of Quick Sort is $O(n^2)$

Running Time of Quick Sort

The worst case for quick-sort occurs when the pivot is the unique minimum or maximum element

One of L and G has size $n - 1$ and the other has size 0

The running time is proportional to the sum

$$n + (n - 1) + \ldots + 2 + 1$$

Thus, the **worst-case** running time of Quick Sort is $O(n^2)$

If the pivot is selected randomly, the **average-case** running time for Quick Sort is $O(n \log n)$.

More Discussion on Quick Sort

Sort in place?

More Discussion on Quick Sort

Sort in place?

YES

More Discussion on Quick Sort

Sort in place?

YES

Stable?

More Discussion on Quick Sort

Sort in place?

YES

Stable?

YES

More Discussion on Quick Sort

Sort in place?

YES

Stable?

YES

But can not be both!!!

More Discussion on Quick Sort

Sort in place?

YES

Stable?

YES

But can not be both!!!

The algorithm just described is stable. However it does not sort in place: $O(n)$ new memory is allocated for L, E and G

More Discussion on Quick Sort

Sort in place?

YES

Stable?

YES

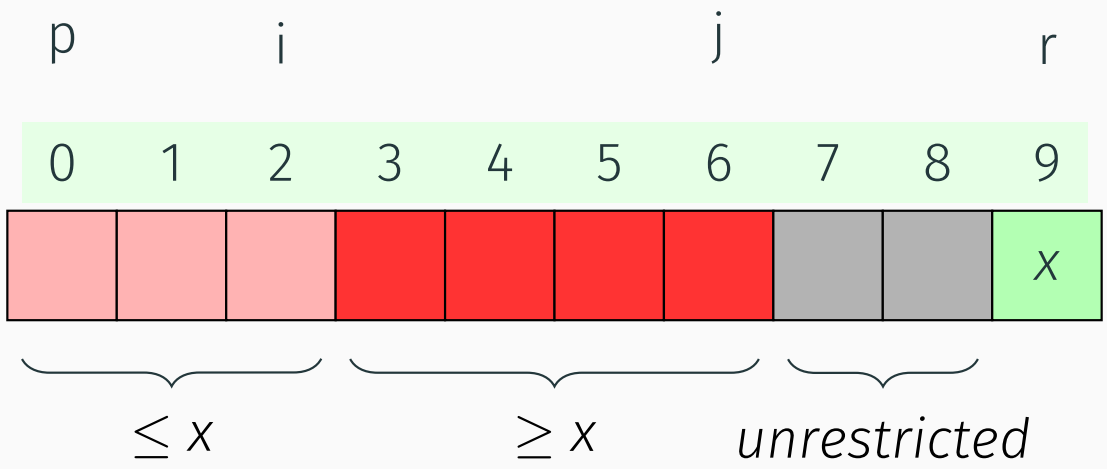
But can not be both!!!

Is there an in-place quick-sort?

In-Place Quick Sort

3 subsets are maintained

- One containing values less than or equal to the pivot
- One containing values greater than the pivot
- One containing values yet to be processed

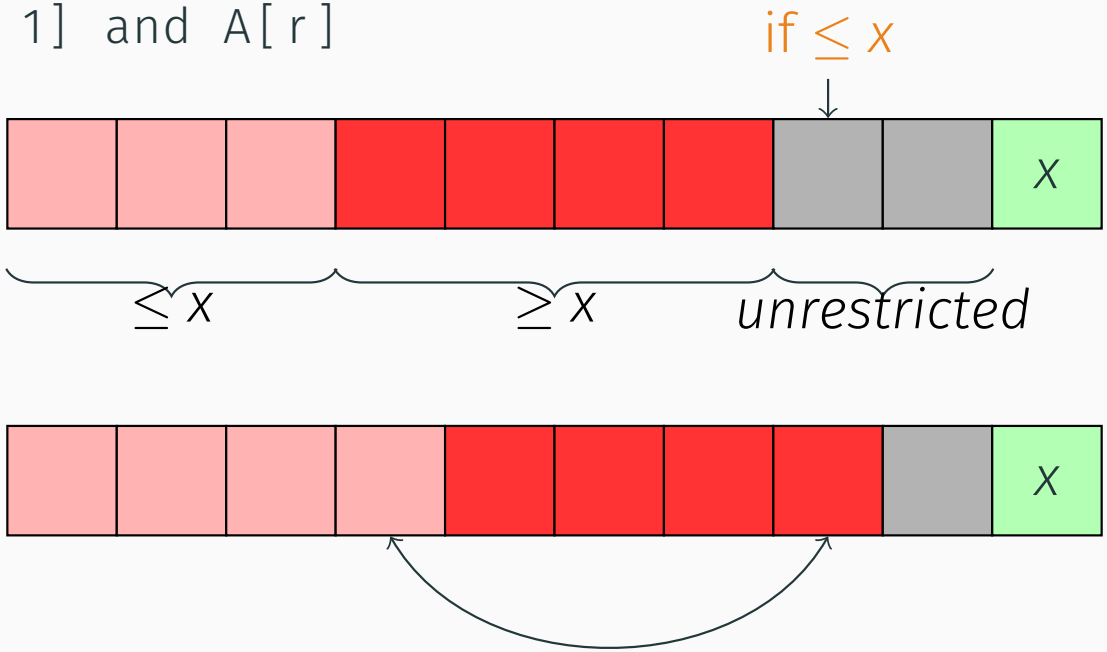


In-Place Quick Sort

```
1  Algorithm QuickSort(A, p, r)
2  if p < r
3    q = Partition(A, p, r)
4    QuickSort(A, p, q - 1)
5    //Small elements are sorted
6    QuickSort(A, q + 1, r)
7    //Large elements are sorted
8    //Thus input is sorted
```

In-Place Quick Sort

```
1  inPlacePartition(A, p ,r)
2  x = A[r]
3  i = p - 1
4  for j = p to r - 1
5     if A[j] <= x
6        i = i + 1
7        swap A[j] and A[i]
8  swap A[i + 1] and A[r]
```

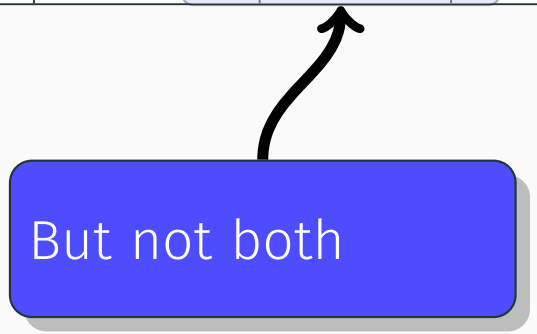


Summary of Comparison Sorts

Algorithm	Best	Worst	Average	In Place	Stable	Comments
Selection	n^2	n^2		Yes	Yes	
Bubble	n	n^2		Yes	Yes	Must count swaps for linear best case running time.
Insertion	n	n^2		Yes	Yes	Good if often almost sorted
Merge	$n \log n$	$n \log n$		Yes	Yes	Good for very large datasets that require swapping to disk
Heap	$n \log n$	$n \log n$		Yes	No	Best if guaranteed $n \log n$ required
Quick	$n \log n$	n^2	$n \log n$	Yes	Yes	Usually fastest in practice

Summary of Comparison Sorts

Algorithm	Best	Worst	Average	In Place	Stable	Comments
Selection	n^2	n^2		Yes	Yes	
Bubble	n	n^2		Yes	Yes	Must count swaps for linear best case running time.
Insertion	n	n^2		Yes	Yes	Good if often almost sorted
Merge	$n \log n$	$n \log n$		Yes	Yes	Good for very large datasets that require swapping to disk
Heap	$n \log n$	$n \log n$		Yes	No	Best if guaranteed $n \log n$ required
Quick	$n \log n$	n^2	$n \log n$	Yes	Yes	Usually fastest in practice

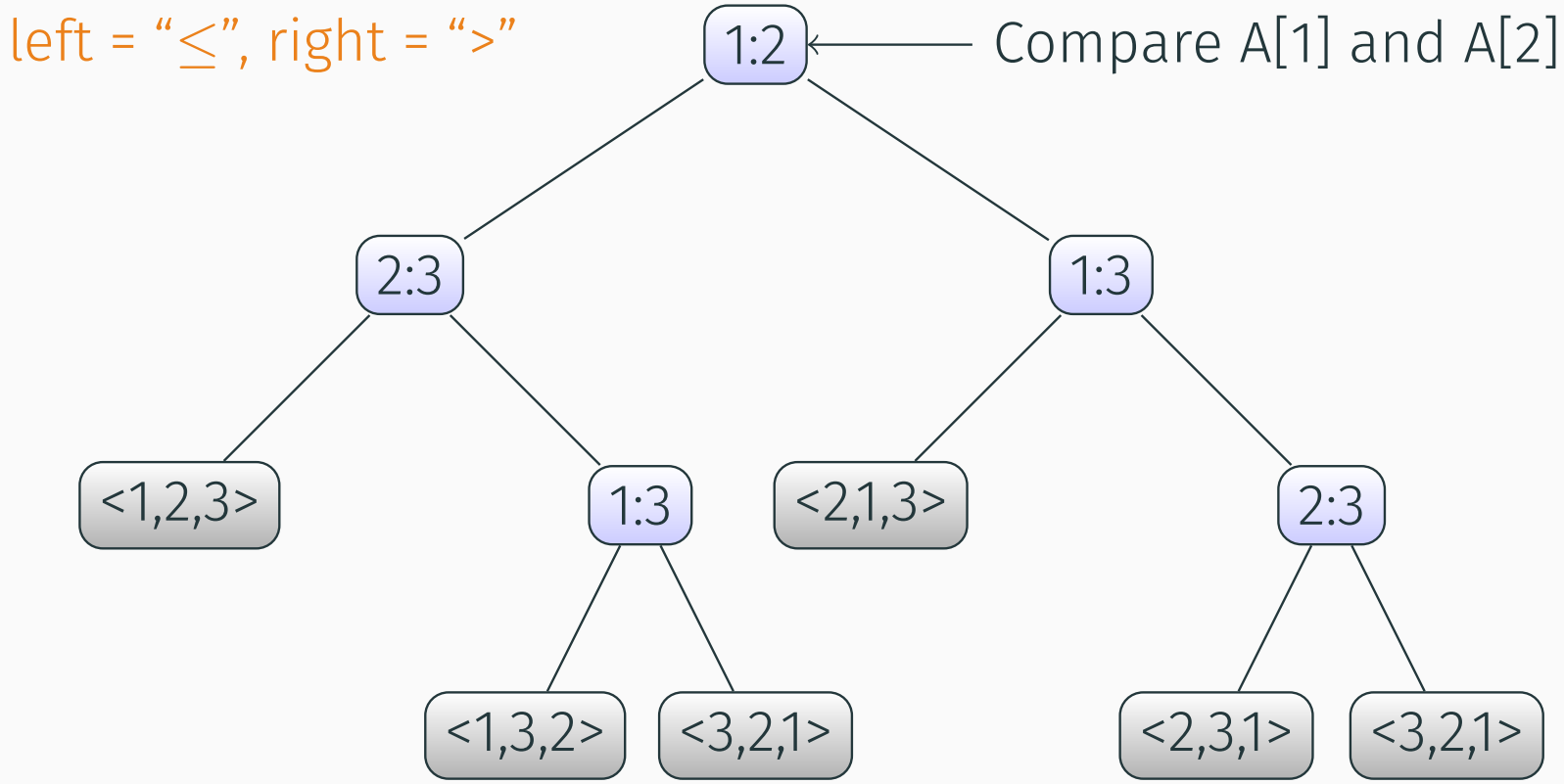


Comparison Sort

Lower Bound on Comparison Sorting (Optional)

Comparison Sort: Decision Trees

- For a 3-element array, there are 6 external nodes.
- For an n-element array, there are $n!$ external nodes.



Comparison Sort: Decision Trees

- For a 3-element array, there are 6 external nodes.
- For an n-element array, there are $n!$ external nodes.

To store $n!$ external nodes, a decision tree must have a height of at least $\lceil \log(n!) \rceil$

Worst-case time is equal to the height of the binary decision tree.

$$T(n) \in \Omega(\log n!) = \Omega(n \log n)$$

Thus Merge Sort and Heap Sort are asymptotically optimal.

Linear Sorting

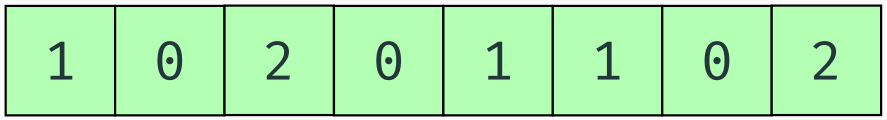
Linear Sorting

Counting Sort

Counting Sort

Invented by *Harold Seward* in 1954.

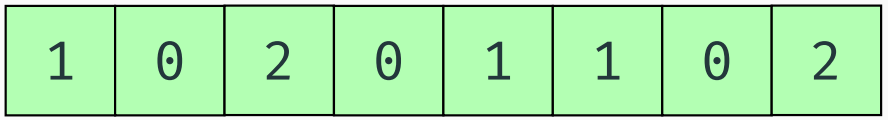
Consider the following case: the elements to be sorted come from a finite (and preferably small) set $[1, \ldots, k]$

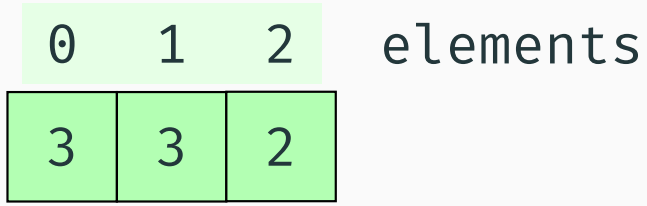


Counting Sort

Invented by *Harold Seward* in 1954.

Consider the following case: the elements to be sorted come from a finite (and preferably small) set $[1, \ldots, k]$

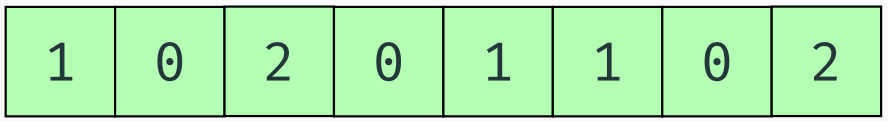


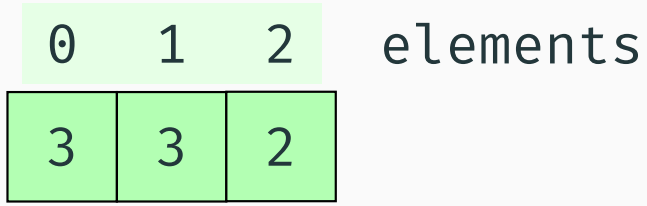


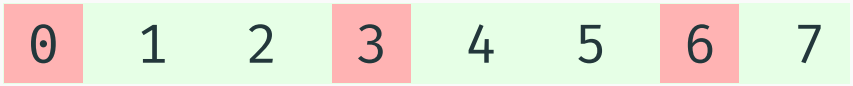
Counting Sort

Invented by *Harold Seward* in 1954.

Consider the following case: the elements to be sorted come from a finite (and preferably small) set $[1, \ldots, k]$



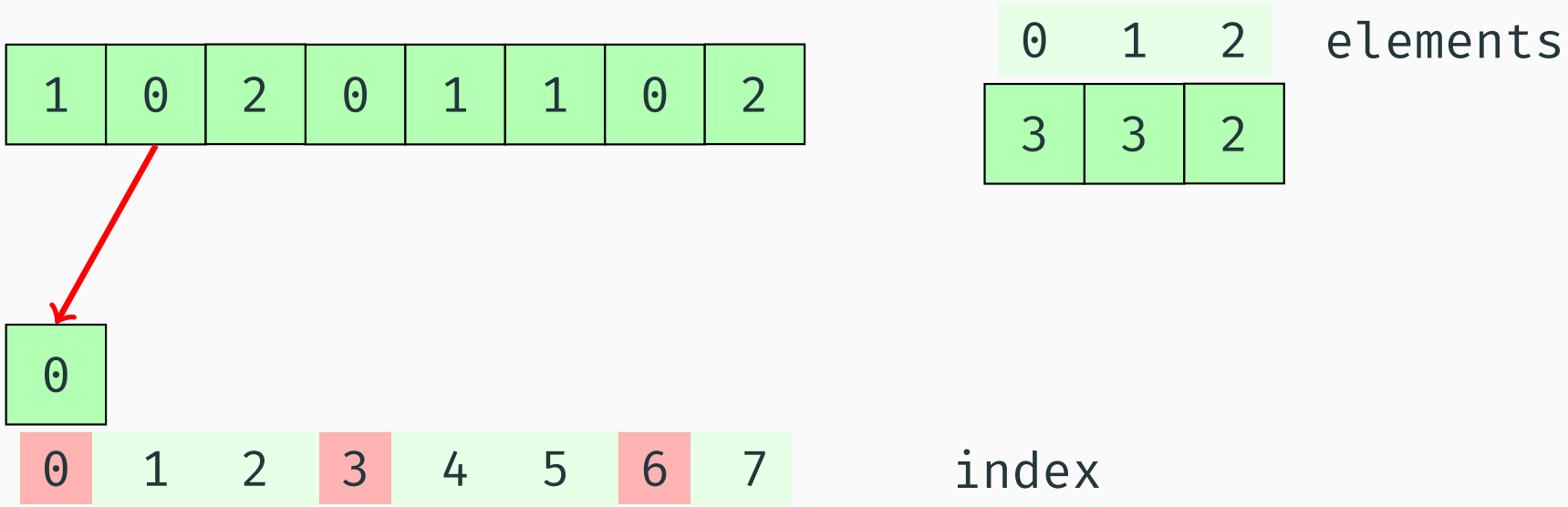




Counting Sort

Invented by *Harold Seward* in 1954.

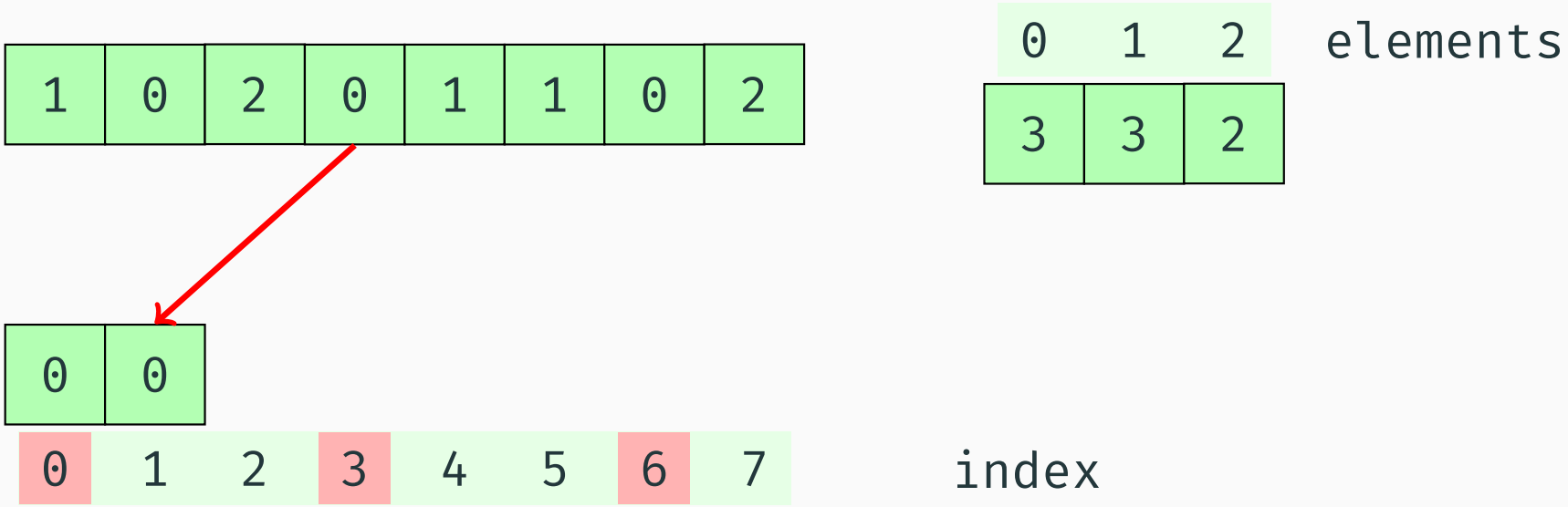
Consider the following case: the elements to be sorted come from a finite (and preferably small) set $[1, \ldots, k]$



Counting Sort

Invented by *Harold Seward* in 1954.

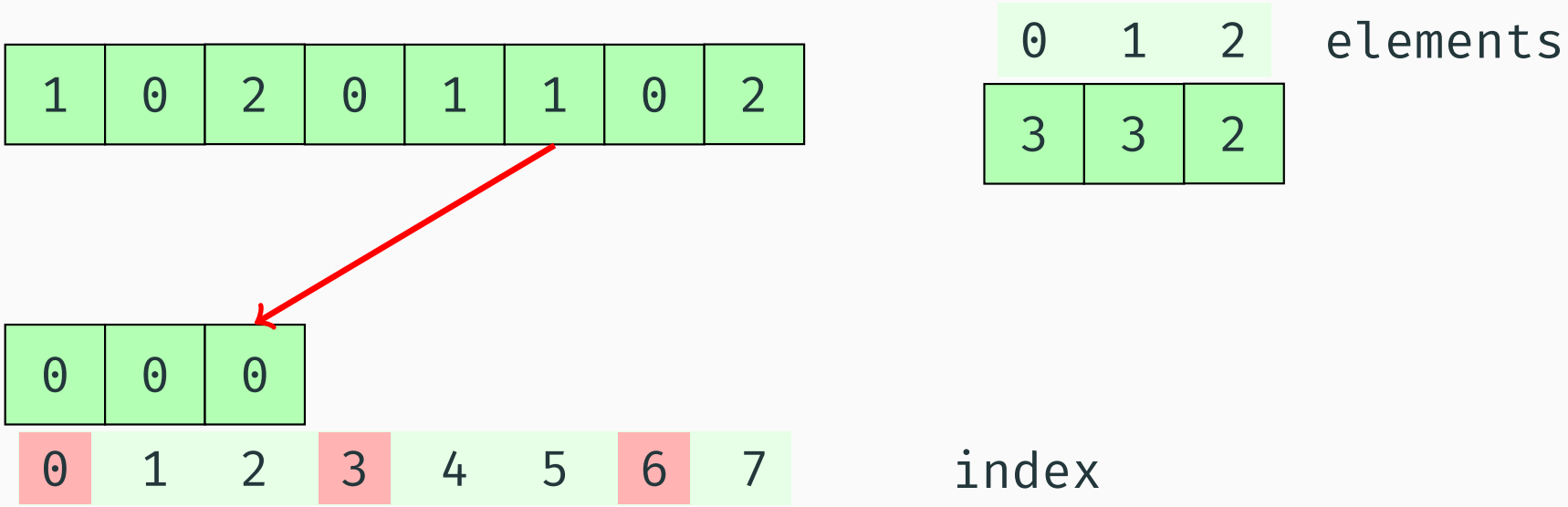
Consider the following case: the elements to be sorted come from a finite (and preferably small) set $[1, \ldots, k]$



Counting Sort

Invented by *Harold Seward* in 1954.

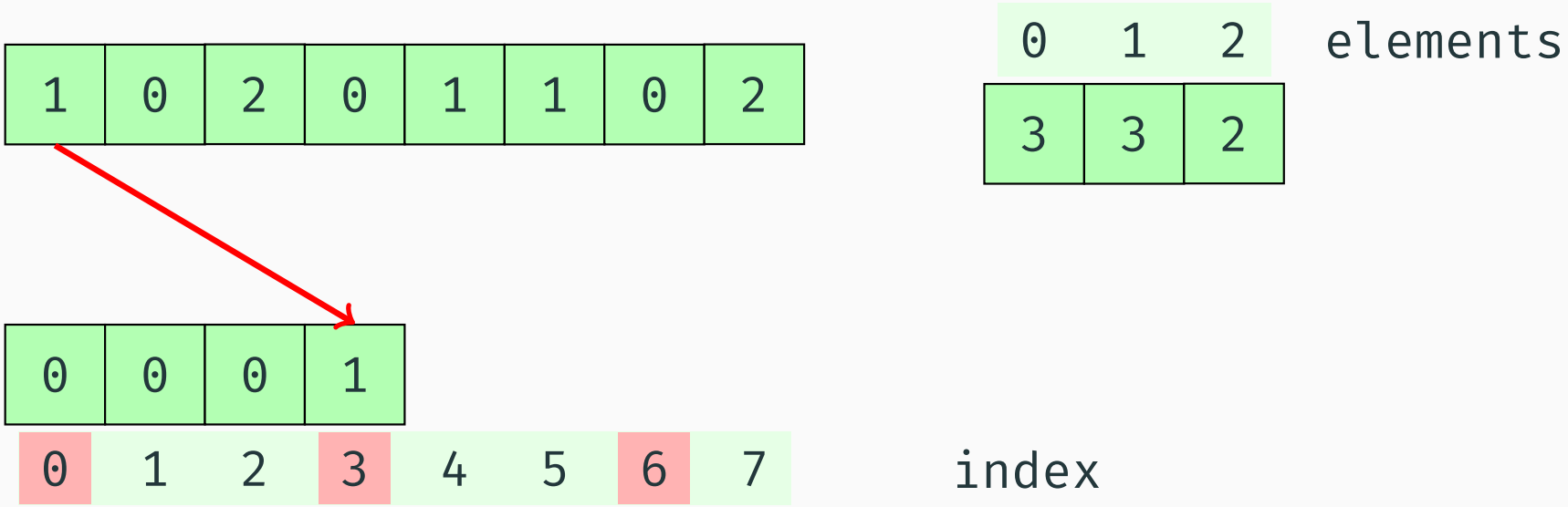
Consider the following case: the elements to be sorted come from a finite (and preferably small) set $[1, \ldots, k]$



Counting Sort

Invented by *Harold Seward* in 1954.

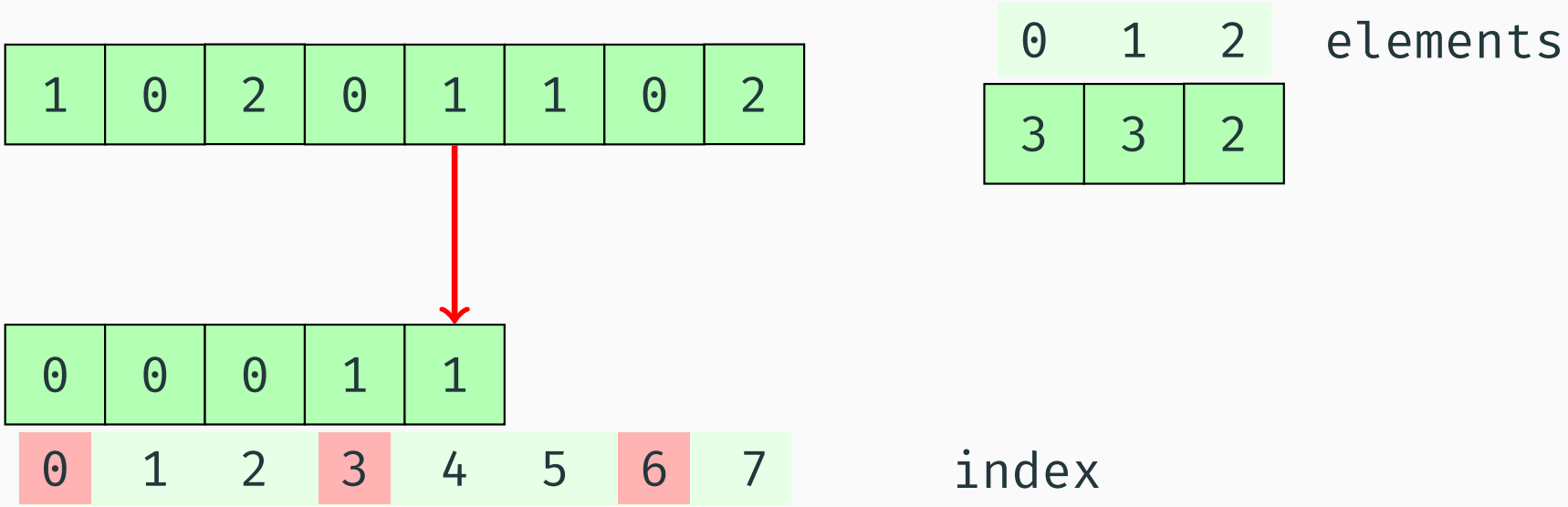
Consider the following case: the elements to be sorted come from a finite (and preferably small) set $[1, \ldots, k]$



Counting Sort

Invented by *Harold Seward* in 1954.

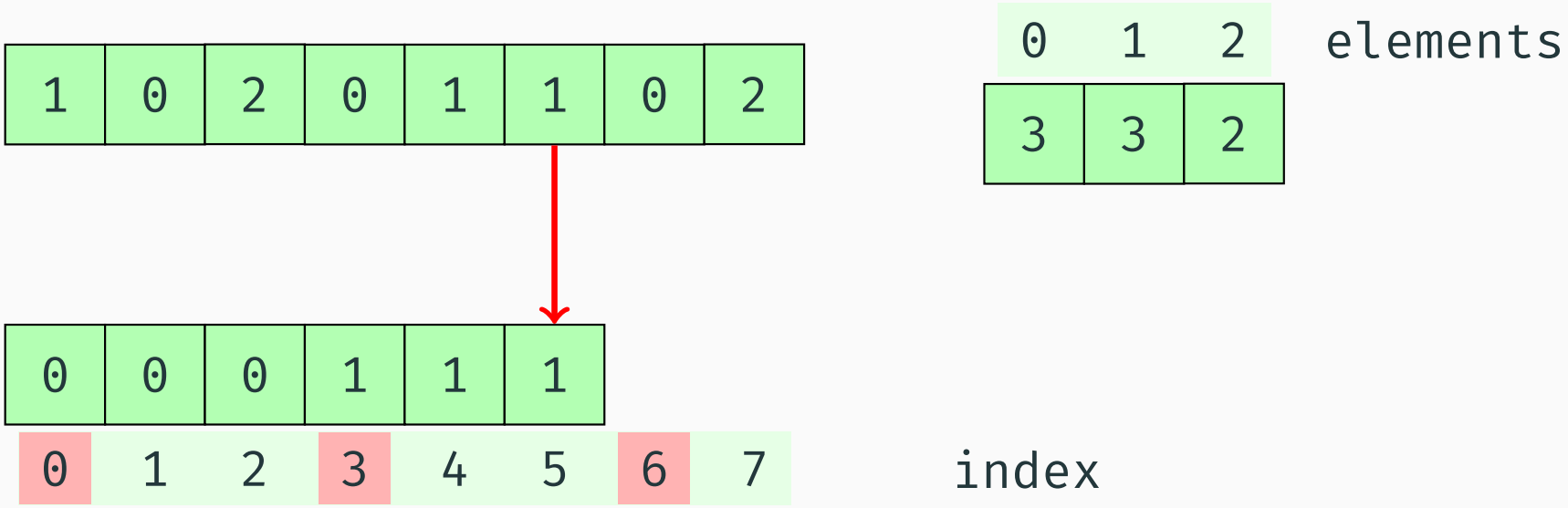
Consider the following case: the elements to be sorted come from a finite (and preferably small) set $[1, \ldots, k]$



Counting Sort

Invented by *Harold Seward* in 1954.

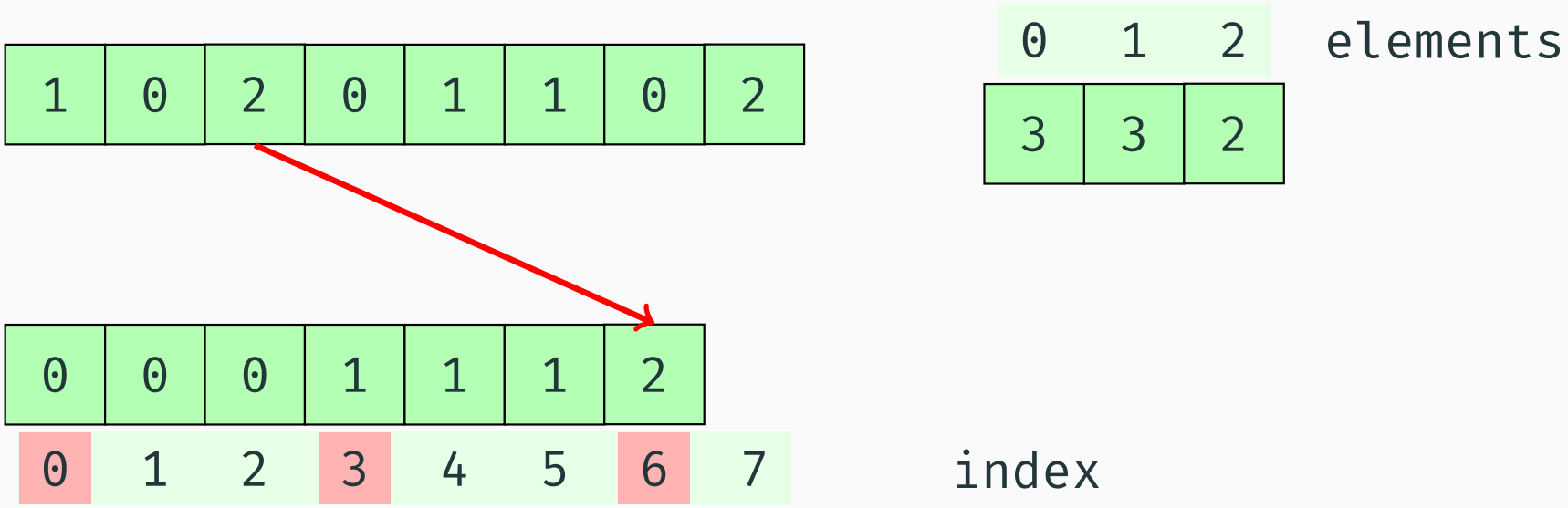
Consider the following case: the elements to be sorted come from a finite (and preferably small) set $[1, \ldots, k]$



Counting Sort

Invented by *Harold Seward* in 1954.

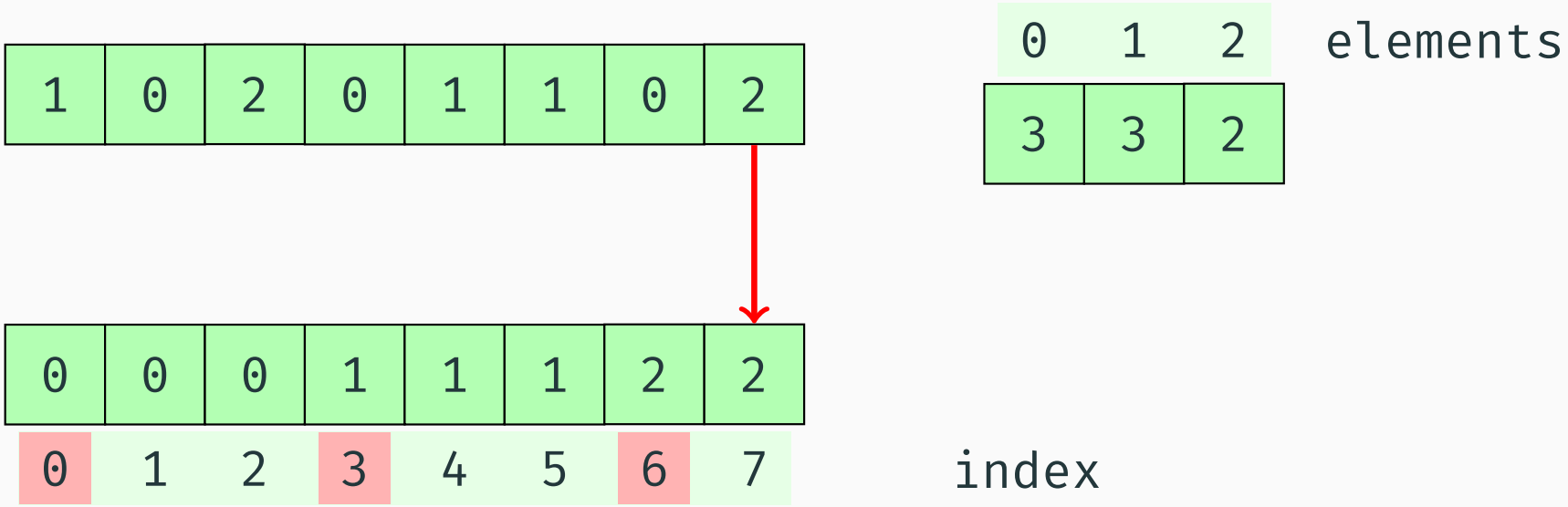
Consider the following case: the elements to be sorted come from a finite (and preferably small) set $[1, \ldots, k]$



Counting Sort

Invented by *Harold Seward* in 1954.

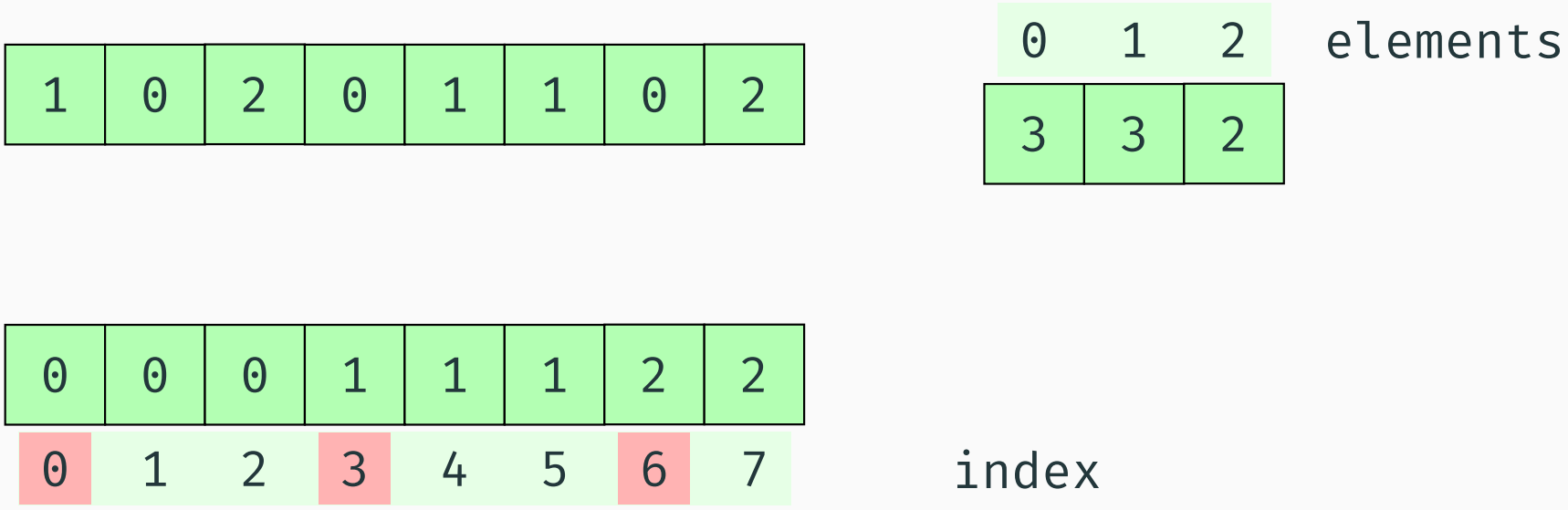
Consider the following case: the elements to be sorted come from a finite (and preferably small) set $[1, \ldots, k]$



Counting Sort

Invented by *Harold Seward* in 1954.

Consider the following case: the elements to be sorted come from a finite (and preferably small) set $[1, \ldots, k]$



Running time: $O(n + k)$

Linear Sorting

Radix Sort

Radix Sort

Suppose input satisfies:

- An array of n numbers
- Each number contains d digits
- Each digit between $[0, \ldots, k-1]$

344 125 333 134 224 334 143 225 325 243

Radix Sort

Suppose input satisfies:

- An array of n numbers
- Each number contains d digits
- Each digit between $[0, \ldots, k-1]$

344 125 333 134 224 334 143 225 325 243

Main idea:

- Select one digit
- Separate numbers into k piles based on selected digit
- Apply some stable sort algorithm only based on selected digit

Radix Sort

344	3	4	4
125	1	2	5
333	3	3	3
134	1	3	4
224	2	2	4
334	3	3	4
143	1	4	3
225	2	2	5
325	3	2	5
243	2	4	3

Radix Sort: left first

3	4	4
1	2	5
3	3	3
1	3	4
2	2	4
3	3	4
1	4	3
2	2	5
3	2	5
2	4	3

Radix Sort: left first

3	4	4
1	2	5
3	3	3
1	3	4
2	2	4
3	3	4
1	4	3
2	2	5
3	2	5
2	4	3

1	2	5
1	3	4
1	4	3
2	2	4
2	2	5
2	4	3
3	4	4
3	3	3
3	3	4
3	2	5

Radix Sort: left first

3	4	4
1	2	5
3	3	3
1	3	4
2	2	4
3	3	4
1	4	3
2	2	5
3	2	5
2	4	3

1	2	5
1	3	4
1	4	3
2	2	4
2	2	5
2	4	3
3	4	4
3	3	3
3	3	4
3	2	5

1	2	5
2	2	4
2	2	5
3	2	5
1	3	4
3	3	3
3	3	4
1	4	3
2	4	3
3	4	4

Wrong!

Radix Sort: right first

3	4	4
1	2	5
3	3	3
1	3	4
2	2	4
3	3	4
1	4	3
2	2	5
3	2	5
2	4	3

Radix Sort: right first

3	4	4
1	2	5
3	3	3
1	3	4
2	2	4
3	3	4
1	4	3
2	2	5
3	2	5
2	4	3

3	3	3
1	4	3
2	4	3
3	4	4
1	3	4
2	2	4
3	3	4
1	2	5
2	2	5
3	2	5

Radix Sort: right first

3	4	4
1	2	5
3	3	3
1	3	4
2	2	4
3	3	4
1	4	3
2	2	5
3	2	5
2	4	3

3	3	3
1	4	3
2	4	3
3	4	4
1	3	4
2	2	4
3	3	4
1	2	5
2	2	5
3	2	5

2	2	4
1	2	5
2	2	5
3	2	5
3	3	3
1	3	4
3	3	4
1	4	3
2	4	3
3	4	4

Radix Sort: right first

3	4	4
1	2	5
3	3	3
1	3	4
2	2	4
3	3	4
1	4	3
2	2	5
3	2	5
2	4	3

3	3	3
1	4	3
2	4	3
3	4	4
1	3	4
2	2	4
3	3	4
1	2	5
2	2	5
3	2	5

2	2	4
1	2	5
2	2	5
3	2	5
3	3	3
1	3	4
3	3	4
1	4	3
2	4	3
3	4	4

1	2	5
1	3	4
1	4	3
2	2	4
2	2	5
2	4	3
3	2	5
3	3	3
3	3	4
3	4	4

Radix Sort: right first

3	4	4
1	2	5
3	3	3
1	3	4
2	2	4
3	3	4
1	4	3
2	2	5
3	2	5
2	4	3

3	3	3
1	4	3
2	4	3
3	4	4
1	3	4
2	2	4
3	3	4
1	2	5
2	2	5
3	2	5

2	2	4
1	2	5
2	2	5
3	2	5
3	3	3
1	3	4
3	3	4
1	4	3
2	4	3
3	4	4

1	2	5
1	3	4
1	4	3
2	2	4
2	2	5
2	4	3
3	2	5
3	3	3
3	3	4
3	4	4

```
1  RadixSort(A, d)
2  for i = 1 to d
3    apply a stable sort to A on digit i
```

Radix Sort: right first

3	4	4
1	2	5
3	3	3
1	3	4
2	2	4
3	3	4
1	4	3
2	2	5
3	2	5
2	4	3

3	3	3
1	4	3
2	4	3
3	4	4
1	3	4
2	2	4
3	3	4
1	2	5
2	2	5
3	2	5

2	2	4
1	2	5
2	2	5
3	2	5
3	3	3
1	3	4
3	3	4
1	4	3
2	4	3
3	4	4

1	2	5
1	3	4
1	4	3
2	2	4
2	2	5
2	4	3
3	2	5
3	3	3
3	3	4
3	4	4

```
1 RadixSort(A, d)
2 for i = 1 to d
3     apply a stable sort to A on digit i
```

Running time: $O(d(n+k))$

Linear Sorting

Bucket Sort

Bucket Sort

Suppose input is constrained to finite interval, *e.g.*, real numbers in the range $[0, 1)$.

Bucket Sort

Suppose input is constrained to finite interval, *e.g.*, real numbers in the range $[0, 1)$.

```
1  BucketSort(A, d)
2  for i = 1 to n
3    insert A[i] to list B[⌊n · A[i]⌋]
4  for i = 0 to n - 1
5    sort list B[i] with Insert Sort
6        // average running time is O(1)
7  Concatenate lists B[0], B[1], ... , B[n-1]
8  return concatenated list
```

Bucket Sort

Suppose input is constrained to finite interval, *e.g.*, real numbers in the range $[0, 1)$.

```
1  BucketSort(A, d)
2  for i = 1 to n
3    insert A[i] to list B[⌊n · A[i]⌋]
4  for i = 0 to n - 1
5    sort list B[i] with Insert Sort
6        // average running time is O(1)
7  Concatenate lists B[0], B[1], ... , B[n-1]
8  return concatenated list
```

If input is random and uniformly distributed, expected running time is $O(n)$.

Thank you!

Questions?