Practical program analysis (continued)

Example analysis (2)

```
1  for (int i = 0; i < n; i++)
2      for (int j = 0; j < i; j++) // Previous example was j<n
3          if (A[i] == A[j])
4              duplicates++;
```

- As before, lines 3–4 run in time $O(1)$.

Example analysis (2)

```
1  for (int i = 0; i < n; i++)
2      for (int j = 0; j < i; j++) // Previous example was j<n
3          if (A[i] == A[j])
4              duplicates++;
```

- As before, lines 3–4 run in time $O(1)$.
- `for (j...)` runs lines 3–4 `i` times, for each `i` $\in \{0, \ldots, n-1\}$.

Example analysis (2)

```
1  for (int i = 0; i < n; i++)
2      for (int j = 0; j < i; j++) // Previous example was j<n
3          if (A[i] == A[j])
4              duplicates++;
```

- As before, lines 3–4 run in time $O(1)$.
- `for (j...)` runs lines 3–4 `i` times, for each `i` $\in \{0, \ldots, n-1\}$.
- Total number of times is $1 + 2 + \cdots + (n-1) = \frac{1}{2}n(n-1) = O(n^2)$.

Example analysis (2)

```
1  for (int i = 0; i < n; i++)
2      for (int j = 0; j < i; j++) // Previous example was j<n
3          if (A[i] == A[j])
4              duplicates++;
```

- As before, lines 3–4 run in time $O(1)$.
- `for (j...)` runs lines 3–4 `i` times, for each `i` $\in \{0, \ldots, n-1\}$.
- Total number of times is $1 + 2 + \cdots + (n-1) = \frac{1}{2}n(n-1) = O(n^2)$.
- So the total running time is $O(n^2)$, as before.
- In fact, in both cases, it is $\Theta(n^2)$.

Example analysis (3)

```
1  int log = 0;
2  for (int i = 0; i < n; i *= 2)
3      log++;
```

- Lines 1 and 3 run in time $O(1)$ – simple statements.

Example analysis (3)

```
1  int log = 0;
2  for (int i = 0; i < n; i *= 2)
3      log++;
```

- Lines 1 and 3 run in time $O(1)$ – simple statements.
- `for (i...)` runs $\log_2 n$ times.

Example analysis (3)

```
1  int log = 0;
2  for (int i = 0; i < n; i *= 2)
3      log++;
```

- Lines 1 and 3 run in time $O(1)$ – simple statements.
- `for (i...)` runs $\log_2 n$ times.
- Total running time is $\Theta(\log n)$.

Sorting algorithms

Selection sort

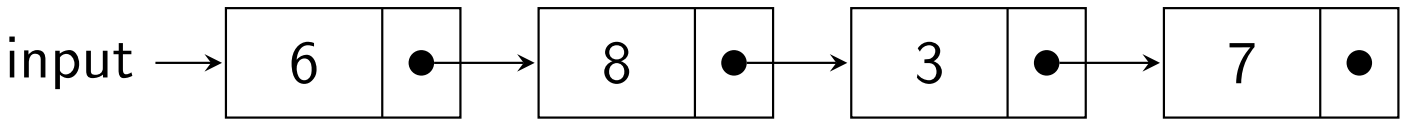
To sort a list:

```
output = empty list;
while (input list is not empty) {
    find smallest item in input;
    delete it from input;
    append it to output;
}
```

Selection sort

To sort a list:

```
output = empty list;
while (input list is not empty) {
    find smallest item in input;
    delete it from input;
    append it to output;
}
```

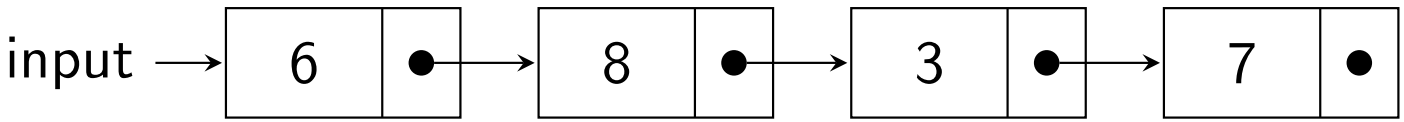


output

Selection sort

To sort a list:

```
    output = empty list;
▷   while (input list is not empty) {
        find smallest item in input;
        delete it from input;
        append it to output;
    }
```

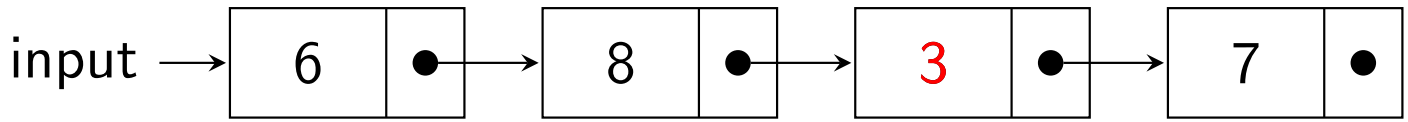


output

Selection sort

To sort a list:

```
output = empty list;
while (input list is not empty) {
    find smallest item in input;
    delete it from input;
    append it to output;
}
```

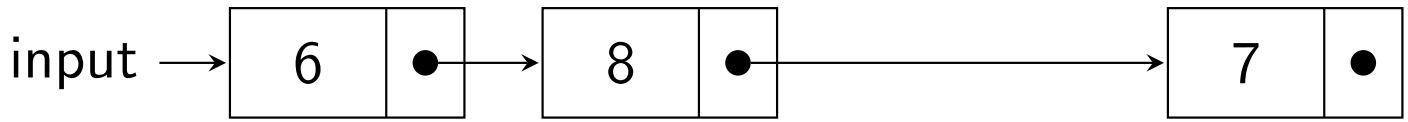


output

Selection sort

To sort a list:

```
output = empty list;
while (input list is not empty) {
    find smallest item in input;
    delete it from input;
    append it to output;
}
```

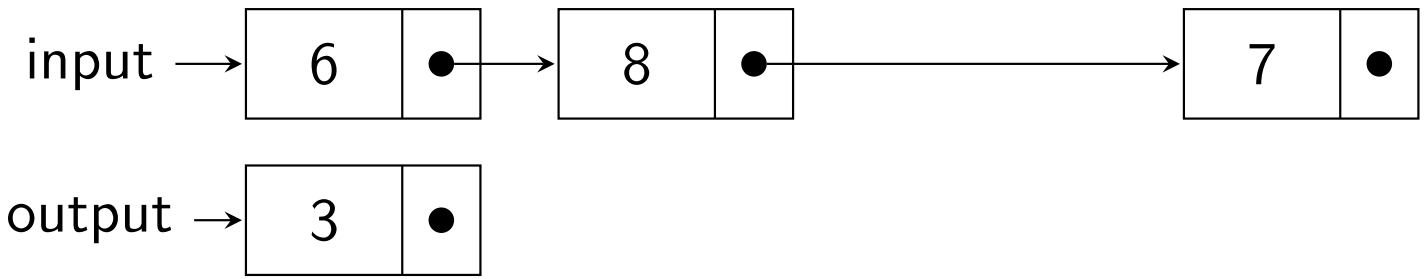


output

Selection sort

To sort a list:

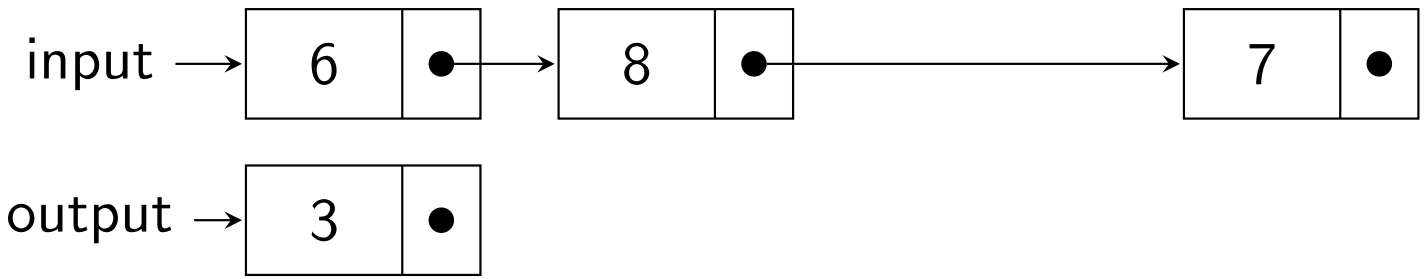
```
output = empty list;
while (input list is not empty) {
    find smallest item in input;
    delete it from input;
▷   append it to output;
}
```



Selection sort

To sort a list:

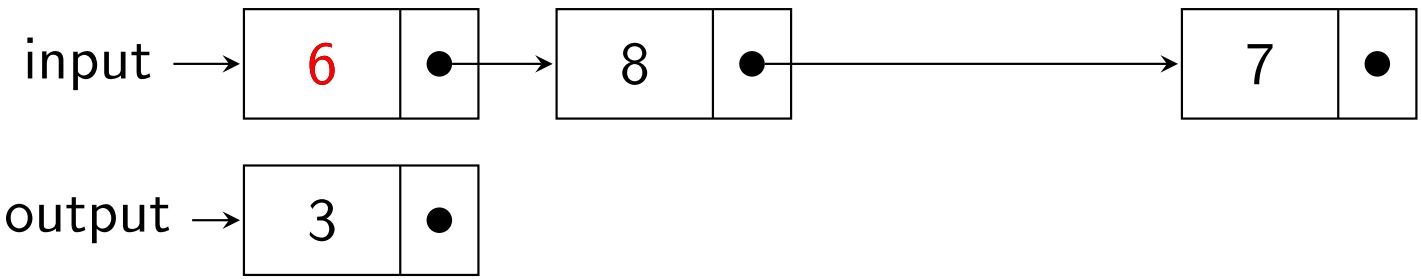
```
    output = empty list;
▷   while (input list is not empty) {
        find smallest item in input;
        delete it from input;
        append it to output;
    }
```



Selection sort

To sort a list:

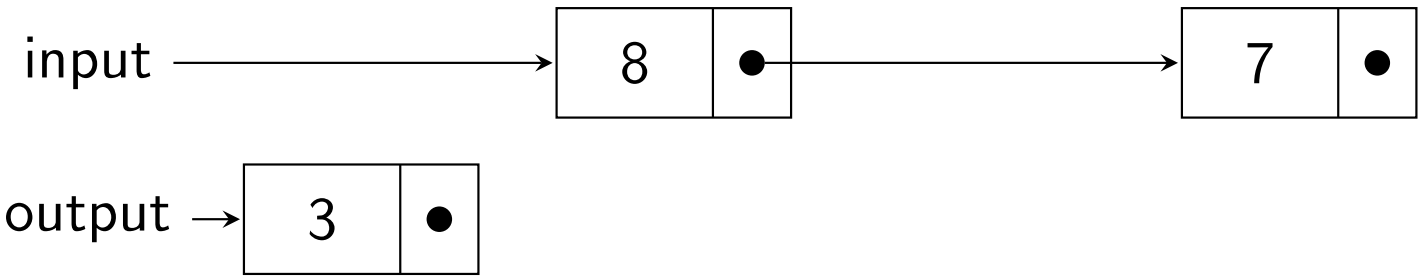
```
output = empty list;
while (input list is not empty) {
    find smallest item in input;
    delete it from input;
    append it to output;
}
```



Selection sort

To sort a list:

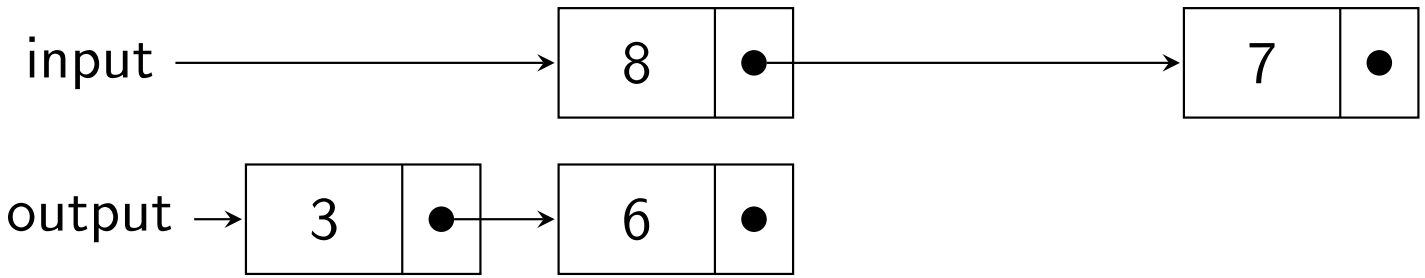
```
output = empty list;
while (input list is not empty) {
    find smallest item in input;
    delete it from input;
    append it to output;
}
```



Selection sort

To sort a list:

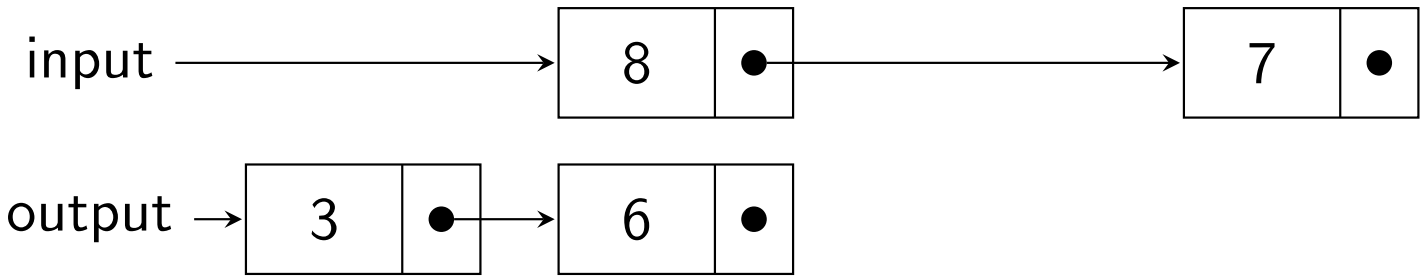
```
output = empty list;
while (input list is not empty) {
    find smallest item in input;
    delete it from input;
▷   append it to output;
}
```



Selection sort

To sort a list:

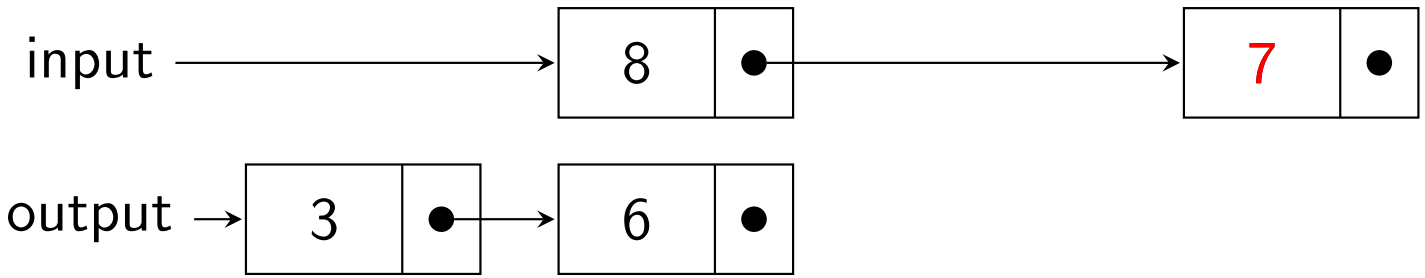
```
    output = empty list;
▷   while (input list is not empty) {
        find smallest item in input;
        delete it from input;
        append it to output;
    }
```



Selection sort

To sort a list:

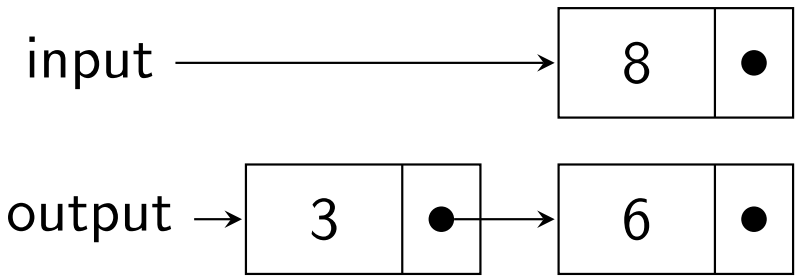
```
output = empty list;
while (input list is not empty) {
    find smallest item in input;
    delete it from input;
    append it to output;
}
```



Selection sort

To sort a list:

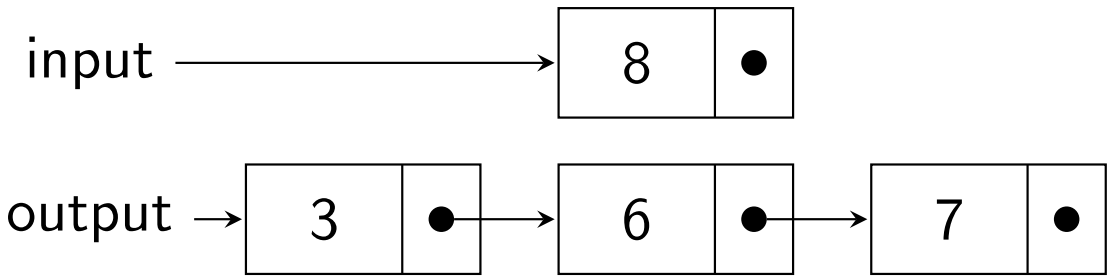
```
output = empty list;
while (input list is not empty) {
    find smallest item in input;
    delete it from input;
    append it to output;
}
```



Selection sort

To sort a list:

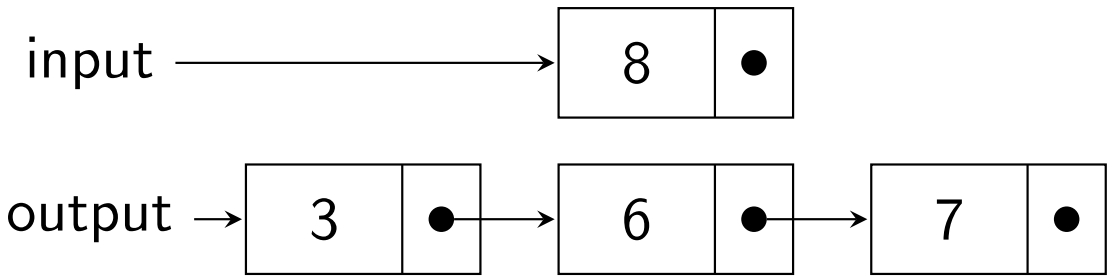
```
output = empty list;
while (input list is not empty) {
    find smallest item in input;
    delete it from input;
▷   append it to output;
}
```



Selection sort

To sort a list:

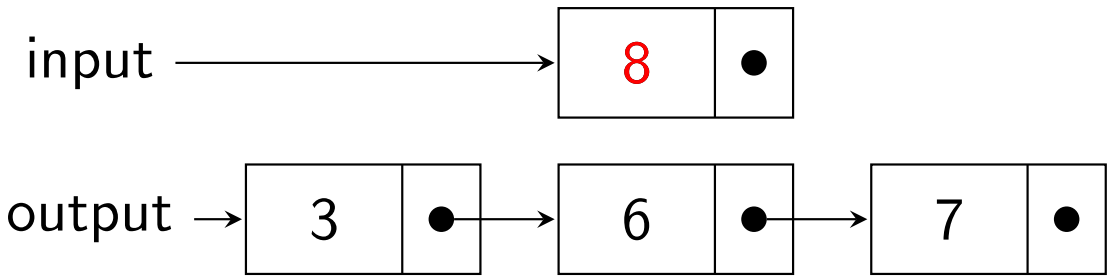
```
    output = empty list;
▷   while (input list is not empty) {
        find smallest item in input;
        delete it from input;
        append it to output;
    }
```



Selection sort

To sort a list:

```
output = empty list;
while (input list is not empty) {
    find smallest item in input;
    delete it from input;
    append it to output;
}
```

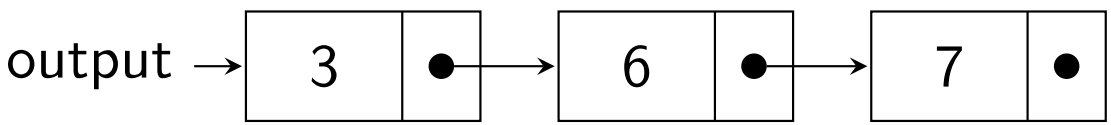


Selection sort

To sort a list:

```
output = empty list;
while (input list is not empty) {
    find smallest item in input;
    delete it from input;
    append it to output;
}
```

input

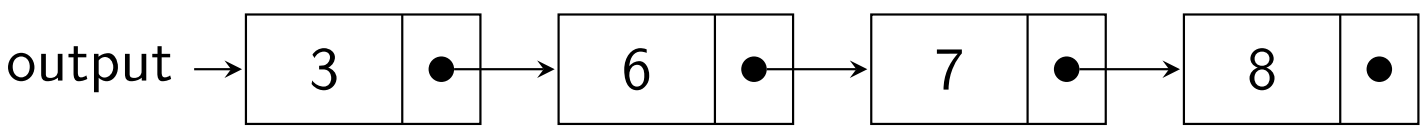


Selection sort

To sort a list:

```
output = empty list;
while (input list is not empty) {
    find smallest item in input;
    delete it from input;
▷   append it to output;
}
```

input

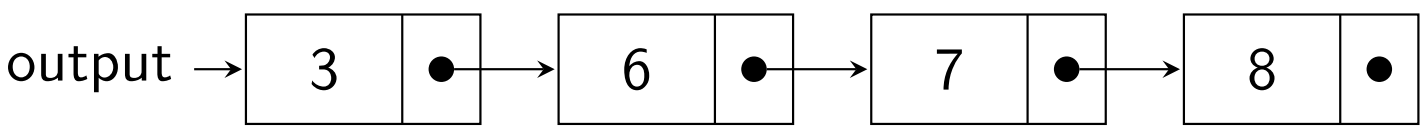


Selection sort

To sort a list:

```
    output = empty list;
▷   while (input list is not empty) {
        find smallest item in input;
        delete it from input;
        append it to output;
    }
```

input



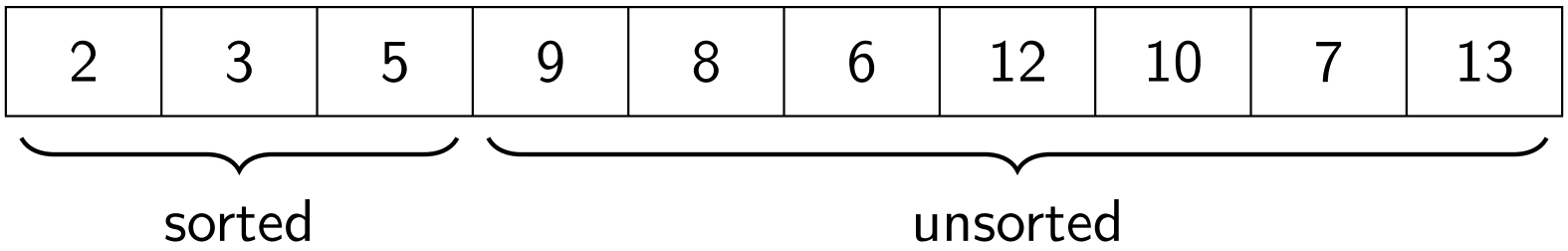
Selection sort for arrays

Suppose we've somehow got here:

2	3	5	9	8	6	12	10	7	13

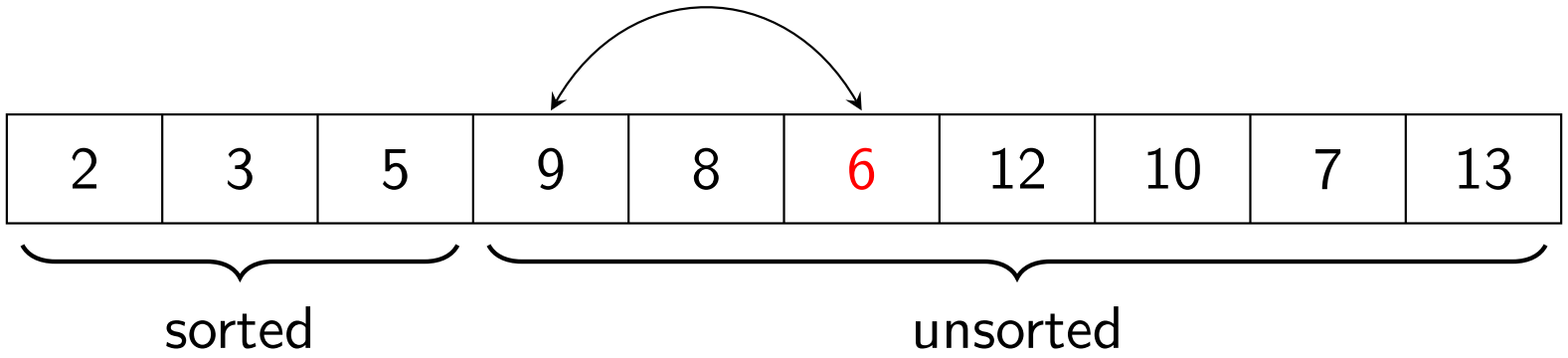
Selection sort for arrays

Suppose we've somehow got here:



Selection sort for arrays

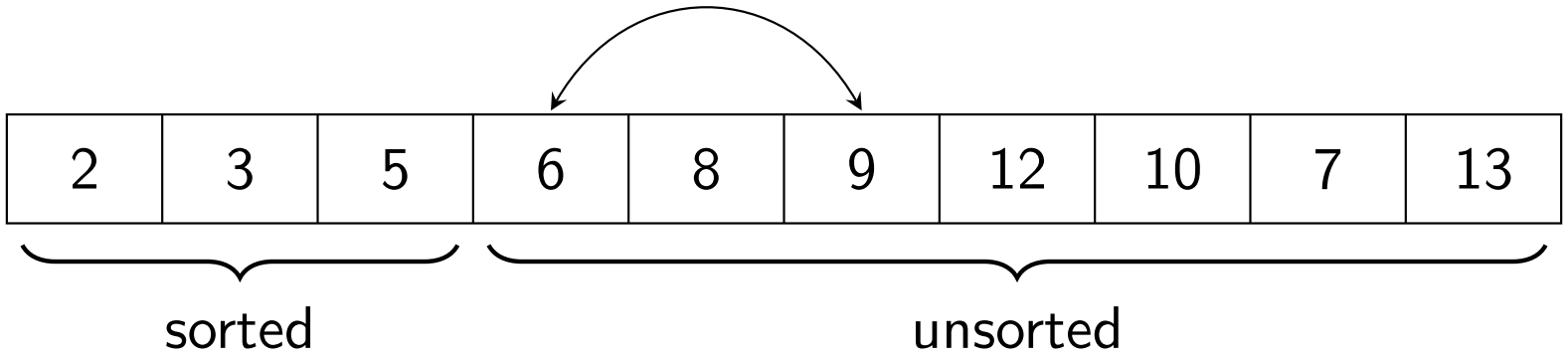
Suppose we've somehow got here:



We can extend the sorted portion of the array by finding the smallest item in the unsorted portion and moving it to the front.

Selection sort for arrays

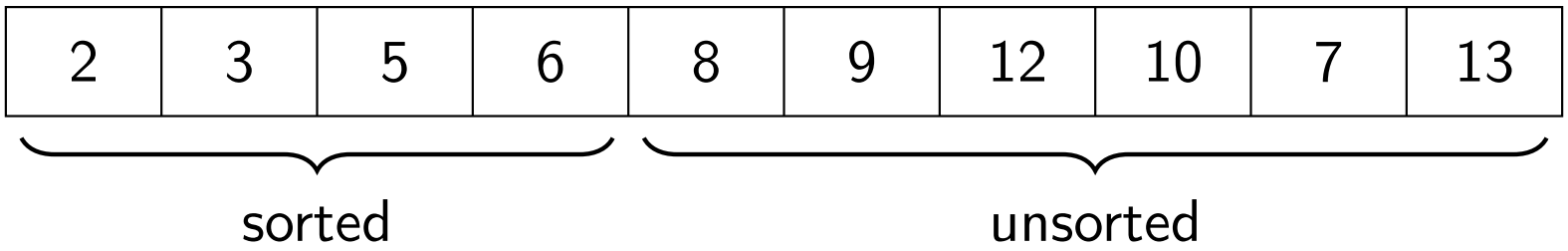
Suppose we've somehow got here:



We can extend the sorted portion of the array by finding the smallest item in the unsorted portion and moving it to the front.

Selection sort for arrays

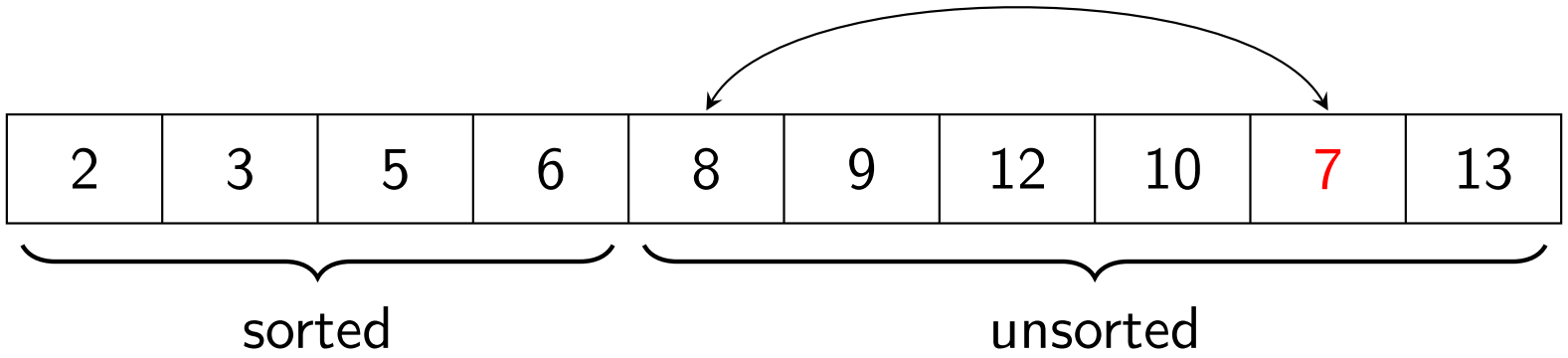
Suppose we've somehow got here:



We can extend the sorted portion of the array by finding the smallest item in the unsorted portion and moving it to the front.

Selection sort for arrays

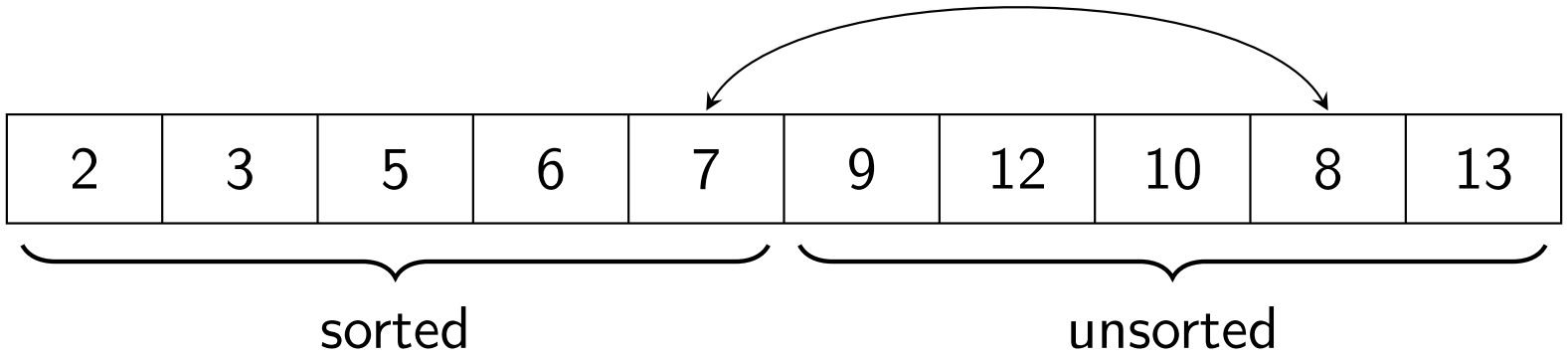
Suppose we've somehow got here:



We can extend the sorted portion of the array by finding the smallest item in the unsorted portion and moving it to the front.

Selection sort for arrays

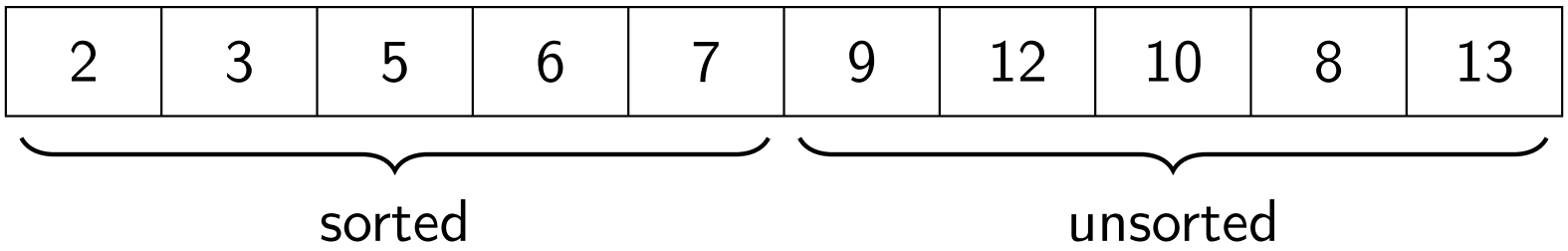
Suppose we've somehow got here:



We can extend the sorted portion of the array by finding the smallest item in the unsorted portion and moving it to the front.

Selection sort for arrays

Suppose we've somehow got here:



We can extend the sorted portion of the array by finding the smallest item in the unsorted portion and moving it to the front.

Start by saying the sorted portion has length zero.

Selection sort in Java

```
public static void sort (int[] ints) {
    for (int i = 0; i < ints.length; i++) {
        int minIndex = i; // Index of min unsorted item
        for (int j = i+1; j < ints.length; j++)
            if (ints[j] < ints[minIndex])
                minIndex = j;
        if (minIndex != i)
            Util.swap (ints, i, minIndex);
    }
}
```

Selection sort in Java

```
public static void sort (int[] ints) {
    for (int i = 0; i < ints.length; i++) {
        int minIndex = i; // Index of min unsorted item
        for (int j = i+1; j < ints.length; j++)
            if (ints[j] < ints[minIndex])
                minIndex = j;
        if (minIndex != i)
            Util.swap (ints, i, minIndex);
    }
}
```

This is an **in-place** sort:

- sorts the array in its original memory location;
- only uses $O(1)$ additional memory (three `ints`, plus one more in `swap()`.

Selection sort running time

```
public static void sort (int[] ints) {
    for (int i = 0; i < ints.length; i++) {
        int minIndex = i;
        for (int j = i+1; j < ints.length; j++)
            if (ints[j] < ints[minIndex])
                minIndex = j;
        if (minIndex != i)
            Util.swap (ints, i, minIndex);
    }
}
```

Consider an array of length n.

Selection sort running time

```
public static void sort (int[] ints) {
    for (int i = 0; i < ints.length; i++) {
        int minIndex = i;
        for (int j = i+1; j < ints.length; j++)
            if (ints[j] < ints[minIndex])
                minIndex = j;
        if (minIndex != i)
            Util.swap (ints, i, minIndex);
    }
}
```

Consider an array of length n.

- For each value of `i`, the inner loop runs $n - \texttt{i} - 1$ times.

Selection sort running time

```
public static void sort (int[] ints) {
    for (int i = 0; i < ints.length; i++) {
        int minIndex = i;
        for (int j = i+1; j < ints.length; j++)
            if (ints[j] < ints[minIndex])
                minIndex = j;
        if (minIndex != i)
            Util.swap (ints, i, minIndex);
    }
}
```

Consider an array of length n.

- For each value of `i`, the inner loop runs $n - \mathtt{i} - 1$ times.
- Total number of steps is $(n-1) + (n-2) + \cdots + 0 = \Theta(n^2)$.

Selection sort – summary

- Extends sorted region by repeatedly adding the smallest unsorted element.
- Can sort arrays and lists.
- Running time is $\Theta(n^2)$.

Insertion sort

Similar idea to selection sort:

- Consider the input as having a sorted part and an unsorted part.

Insertion sort

Similar idea to selection sort:

- Consider the input as having a sorted part and an unsorted part.
- Initially, the sorted part is empty.
- Grow the sorted part by taking the first unsorted item and inserting it into the right place.

Insertion sort running time

If the sorted part has length s,

- for an array, inserting the new item requires moving up to s existing items to make space;
- for a list, inserting requires searching through up to s positions to find the right place.

Running time is proportional to $1 + 2 + \cdots + (n-1) = \Theta(n^2)$.

Mergesort

Divide and conquer:

- divide a problem into sub-problems;
- solve the sub-problems (usually by recursion);
- combine the solutions to solve the main problem.

Mergesort sorts a list by:

- splitting the list into two halves;
- recursively sorting the halves;
- merging the two halves into a single sorted list.

Mergesort example

Input:	9	3	6	5	2	4	8	7
Sublist 1:								
Sublist 2:								
Output:								

- Divide the input into two sublists.

Mergesort example

Input:	3	6	5	2	4	8	7
Sublist 1:	9						
Sublist 2:							
Output:							

- Divide the input into two sublists.

Mergesort example

Input:	6	5	2	4	8	7
Sublist 1:	9					
Sublist 2:	3					
Output:						

- Divide the input into two sublists.

Mergesort example

Input:	5	2	4	8	7
Sublist 1:	9	6			
Sublist 2:	3				
Output:					

- Divide the input into two sublists.

Mergesort example

Input:	2	4	8	7
Sublist 1:	9	6		
Sublist 2:	3	5		
Output:				

- Divide the input into two sublists.

Mergesort example

Input:				
Sublist 1:	9	6	2	8
Sublist 2:	3	5	4	7
Output:				

- Divide the input into two sublists.

Mergesort example

Input:	
Sublist 1:	Sort by
Sublist 2:	recursion
Output:	

- Divide the input into two sublists.
- Sort the sublists by recursion.

Mergesort example

Input:				
Sublist 1:	2	6	8	9
Sublist 2:	3	4	5	7
Output:				

- Divide the input into two sublists.
- Sort the sublists by recursion.
- Merge the sublists.

Mergesort example

Input:					
Sublist 1:	6	8	9		
Sublist 2:	3	4	5	7	
Output:	2				

- Divide the input into two sublists.
- Sort the sublists by recursion.
- Merge the sublists.

Mergesort example

Input:			
Sublist 1:	6	8	9
Sublist 2:	4	5	7
Output:	2	3	

- Divide the input into two sublists.
- Sort the sublists by recursion.
- Merge the sublists.

Mergesort example

Input:			
Sublist 1:	6	8	9
Sublist 2:	5	7	
Output:	2	3	4

- Divide the input into two sublists.
- Sort the sublists by recursion.
- Merge the sublists.

Mergesort example

Input:				
Sublist 1:	6	8	9	
Sublist 2:	7			
Output:	2	3	4	5

- Divide the input into two sublists.
- Sort the sublists by recursion.
- Merge the sublists.

Mergesort example

Input:					
Sublist 1:	8	9			
Sublist 2:	7				
Output:	2	3	4	5	6

- Divide the input into two sublists.
- Sort the sublists by recursion.
- Merge the sublists.

Mergesort example

Input:						
Sublist 1:	8	9				
Sublist 2:						
Output:	2	3	4	5	6	7

- Divide the input into two sublists.
- Sort the sublists by recursion.
- Merge the sublists.

Mergesort example

Input:							
Sublist 1:	9						
Sublist 2:							
Output:	2	3	4	5	6	7	8

- Divide the input into two sublists.
- Sort the sublists by recursion.
- Merge the sublists.

Mergesort example

Input:								
Sublist 1:								
Sublist 2:								
Output:	2	3	4	5	6	7	8	9

- Divide the input into two sublists.
- Sort the sublists by recursion.
- Merge the sublists.

Mergesort in Java (1)

```
public static void mergesort (DoublyLinkedList list) {
    if (list.length() <= 1) return;

    DoublyLinkedList one = new DoublyLinkedList();
    DoublyLinkedList two = new DoublyLinkedList();

    // Split into sublists.
    while (!list.isEmpty()) {
        one.addToTail(extractHead (list));
        if (!list.isEmpty())
            two.addToTail(extractHead (list));
    }
    ...
```

`extractHead()` is a helper method that returns the first element of the list and deletes it.

Mergesort in Java (2)

```
        // Recursively sort the sublists.
        mergesort (one);
        mergesort (two);

        // Merge the sublists.
        while (!(one.isEmpty() && two.isEmpty())) {
            if (two.isEmpty())
                list.addToTail (extractHead (one));
            else if (one.isEmpty())
                list.addToTail (extractHead (two));
            else if (one.get(0).compareTo(two.get(0)) < 0)
                list.addToTail (extractHead (one));
            else
                list.addToTail (extractHead (two));
        }
    }
```

Mergesort running time (1)

Suppose we run `mergesort` on a list of length n.

- Splitting walks along the list once: time $\Theta(n)$.

Mergesort running time (1)

Suppose we run `mergesort` on a list of length n.

- Splitting walks along the list once: time $\Theta(n)$.
- Recursive calls: time ??

Mergesort running time (1)

Suppose we run `mergesort` on a list of length n.

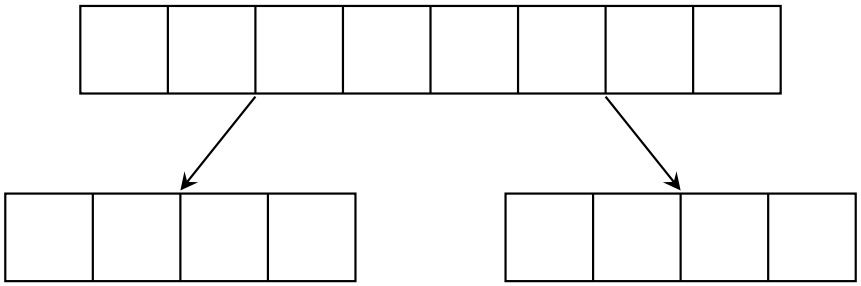
- Splitting walks along the list once: time $\Theta(n)$.
- Recursive calls: time ??
- Merging constructs the output list one element at a time: time $\Theta(n)$.

Mergesort running time (2)

The recursion when we sort a list of length n:

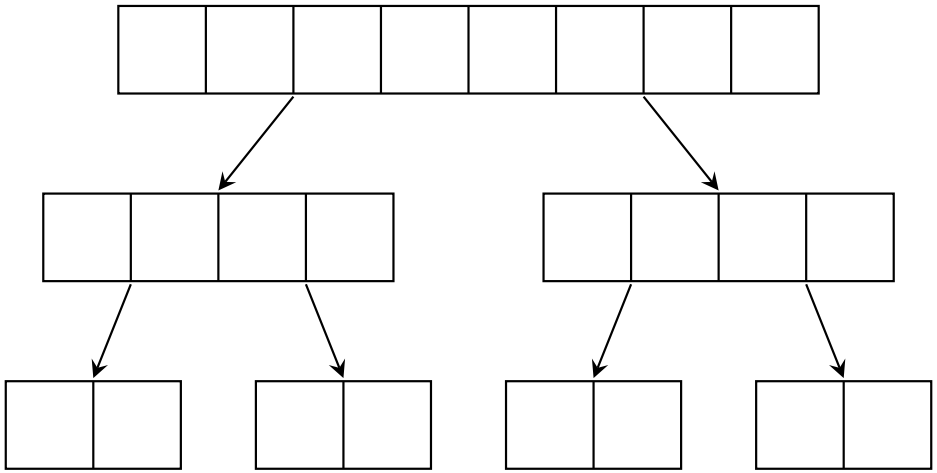
Mergesort running time (2)

The recursion when we sort a list of length n:



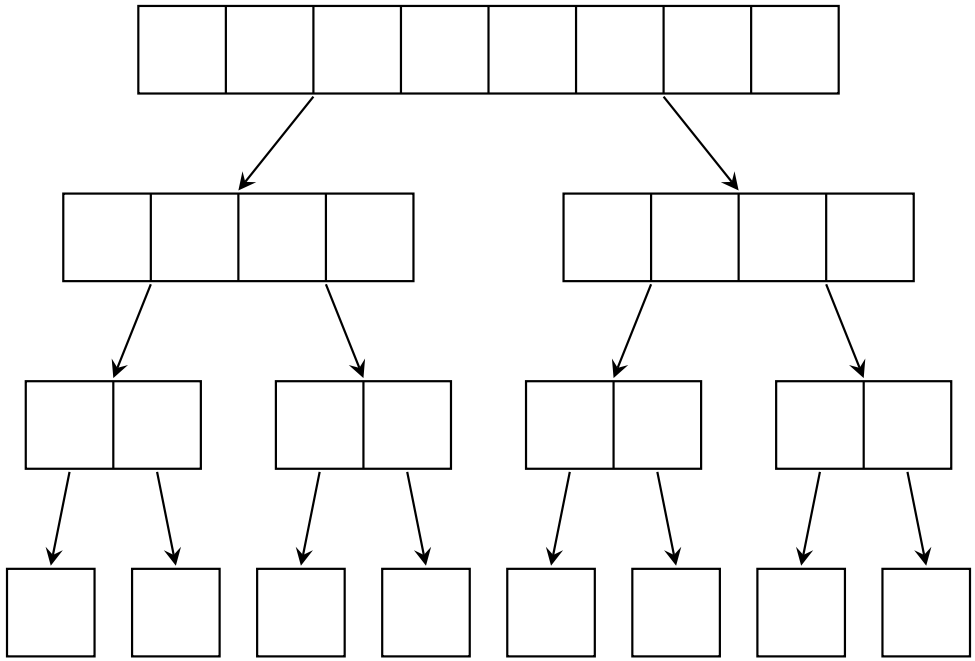
Mergesort running time (2)

The recursion when we sort a list of length n:



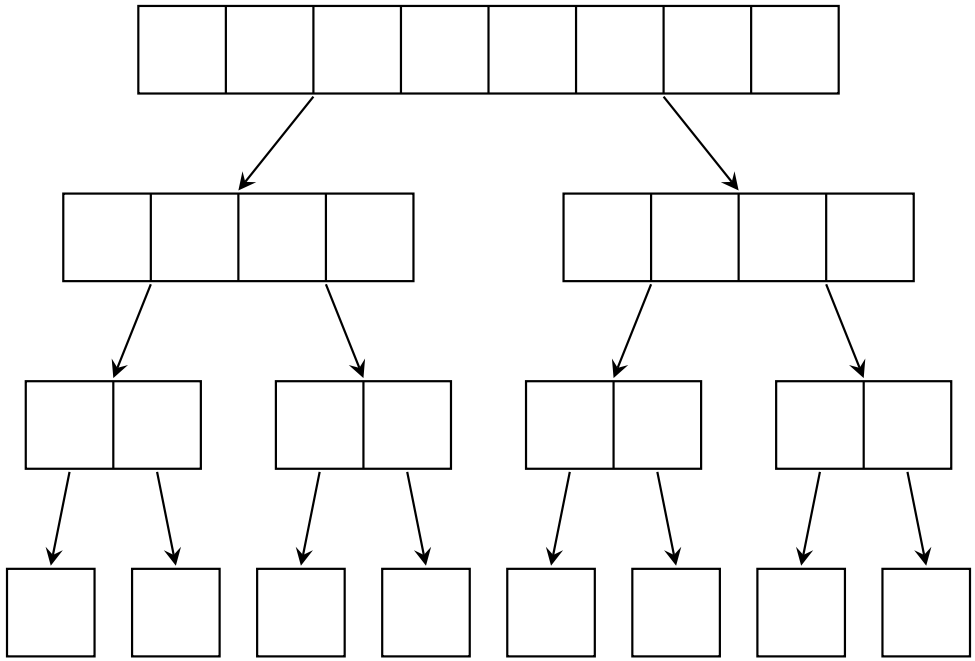
Mergesort running time (2)

The recursion when we sort a list of length n:



Mergesort running time (2)

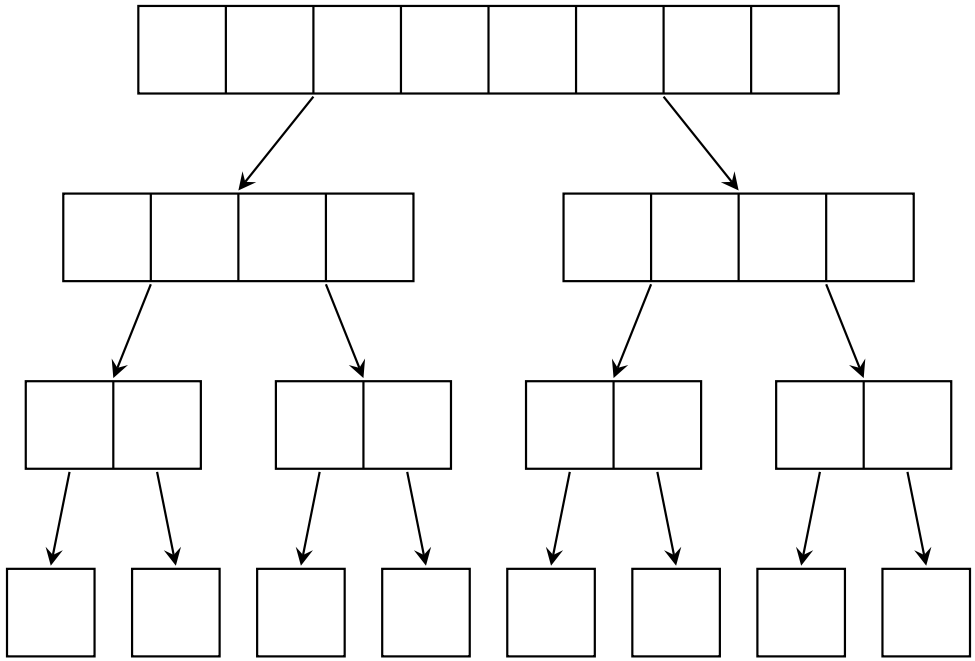
The recursion when we sort a list of length n:



There are $\log_2 n$ levels of recursion (halve list until length 1).

Mergesort running time (2)

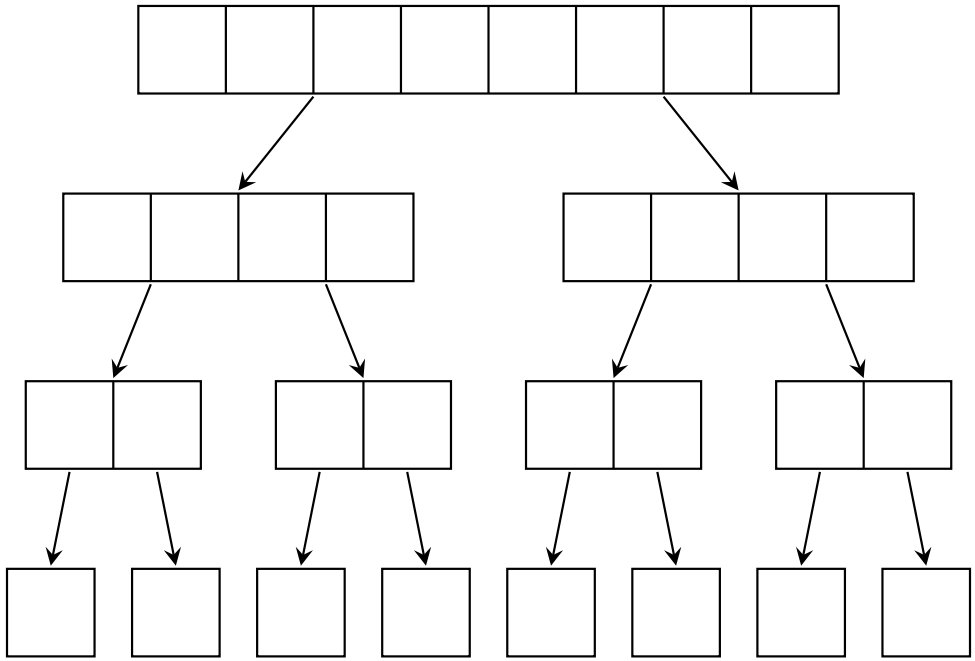
The recursion when we sort a list of length n:



There are $\log_2 n$ levels of recursion (halve list until length 1).
Each level processes lists of total length n so takes time $\Theta(n)$.

Mergesort running time (2)

The recursion when we sort a list of length n:



There are $\log_2 n$ levels of recursion (halve list until length 1).
Each level processes lists of total length n so takes time $\Theta(n)$.
Total running time is $\Theta(n \log n)$.

Mergesort – summary

- Sorts by recursively splitting into sublists and combining.
- Mostly used on lists (can be adapted to arrays).
- Running time is $\Theta(n \log n)$.
- No sorting algorithm based on comparing pairs of items can run faster than $\Theta(n \log n)$.