SIM GLOBAL EDUCATION

UNIVERSITY OF WOLLONGONG AUSTRALIA

CSCI203 - Algorithms and Data Structures

Sorting Algorithms

Sionggo Japit

sjapit@uow.edu.au

2 January 2023

Lecture Outline

Sorting:

- Bubble sort
- Selection sort
- Insertion sort
- Shell sort
- Merge sort
- Quick sort
- Heap sort

Sorting

Sorting

- Some popular sorting algorithms
  - Bubble sort
  - Selection sort
  - Insertion sort
  - Shell sort
  - Merge sort
  - Quick sort
  - Heap sort

- http://www.eecs.harvard.edu/~ellard/Q-97/Demos/SortDemo/sorts.html

Bubble Sort

Unsorted List

2	6	4	13	11	15

swap swap swap swap

2	13	6	4	15	11
2	6	13	4	15	11
2	6	4	13	15	11
2	6	4	13	15	11
2	6	4	13	11	15

End of Pass 1

Bubble Sort

Bubble Sort

Bubble Sort

Method:

1. Stepping through the list to be sorted
2. Comparing two items at a time, swap them if they are in the wrong order
3. Repeat step 1 until no swaps are needed

Bubble Sort

Unsorted List

13	2	6	4	15	11

Start pass 1:

13	2	6	4	15	11

compare

2	13	6	4	15	11

swapped

2	13	6	4	15	11

compare

2	6	13	4	15	11

swapped

2	6	13	4	15	11

compare

2	6	4	13	15	11

swapped

2	6	4	13	15	11

compare

2	6	4	13	15	11

no swap

2	6	4	13	15	11

compare

2	6	4	13	11	15

swapped

2	6	4	13	11	**15**

End pass 1:

Start pass 2:

2	6	4	13	11	**15**

compare

2	6	4	13	11	**15**

no swap

2	6	4	13	11	**15**

compare

2	4	6	13	11	**15**

swapped

2	4	6	13	11	**15**

compare

2	4	6	13	11	**15**

no swap

2	4	6	13	11	**15**

compare

2	4	6	11	13	**15**

swapped

2	4	6	11	**13**	**15**

End pass 2:

Start pass 3:

2	4	6	11	**13**	**15**

compare

2	4	6	11	**13**	**15**

no swap

2	4	6	11	**13**	**15**

compare

2	4	6	11	**13**	**15**

no swap

2	4	6	11	**13**	**15**

compare

2	4	6	11	**13**	**15**

no swap

2	4	6	**11**	**13**	**15**

End pass 3:

2	4	6	**11**	**13**	**15**

Since there is no swap operation in the current pass, this indicates the list is in sorted order now. The Bubble sort algorithm terminates.

Bubble Sort

Bubble sort

```
procedure bubble(T[1..n])
 sorted = false
 i = 0
 while i < n and not sorted do
    sorted = true
    i = i + 1
    for j = 1 to n – i do
     if T[j] > T[j + 1]
        t = T[j]
        T[j] = T[j + 1]
        T[j + 1] = t
        sorted = false
```

While the list is still not sorted, and the number of unsorted element is smaller than n, continue to sort.

Bubble Sort

Bubble sort

```
procedure bubble(T[1..n])
 sorted = false
 i = 0
 while i < n and not sorted do
    sorted = true
    i = i + 1
    for j = 1 to n – i do
      if T[j] > T[j + 1]
        t = T[j]
        T[j] = T[j + 1]
        T[j + 1] = t
        sorted = false
```

While the list is still not sorted, and the number of unsorted element is smaller than n, continue to sort.

Swap T[j] and T[j+1]

Bubble Sort

Bubble sort

```
procedure bubble(T[1..n])
 sorted = false
 i = 0
 while i < n and not sorted do
    sorted = true
    i = i + 1
    for j = 1 to n – i do
     if T[j] > T[j + 1]
        t = T[j]
        T[j] = T[j + 1]
        T[j + 1] = t
        sorted = false
```

While the list is still not sorted, and the number of unsorted element is smaller than n, continue to sort.

Swap T[j] and T[j+1]

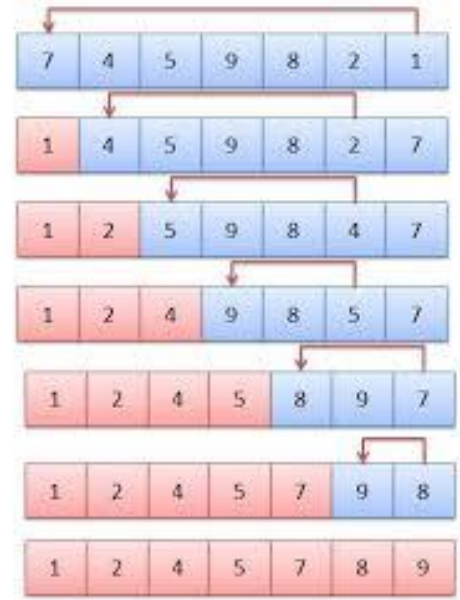
If there is some swapping, the data is not sorted yet. Set a flag to false.

Bubble Sort

Bubble sort's efficiency:

- At most $n + 1$ times through the outer loop.
- $n - i$ times through each inner loop.
- Algorithm is in $\Theta(n^2)$.
- Usually more swaps than selection or insertion sort.
- Good if only a few items out of order.

Note: While simple, this algorithm is highly inefficient and is rarely used except in education.



Selection Sort

Selection Sort

Selection Sort

Method:

1. Find the minimum value in the list
2. Swap it with the value in the first position
3. Repeat the steps above for the remainder of the list

Selection Sort

Unsorted List

13	2	6	4	15	11

Start processing:

13	2	6	4	15	11

Scan the list for smallest value.

2	13	6	4	15	11

swap

2	13	6	4	15	11

Scan the list for smallest value.

2	4	6	13	15	11

swap

2	4	6	13	15	11

Scan the list for smallest value.

2	4	6	13	15	11

swap

2	4	6	13	15	11

Scan the list for smallest value.

2	4	6	11	15	13

swap

2	4	6	11	15	13

Scan the list for smallest value.

2	4	6	11	13	15

swap

End processing:

Selection Sort

Selection sort

```
Procedure select(T[1..n])
  for i = 1 to n-1 do
    minIndex = i;
    minValue = T[i];
    for j = i+1 to n do
      if T[j] < minValue then
        minIndex = j;
        minValue = T[j];
    T[minIndex] = T[i];
    T[i] = minValue
```

For 1 to n-1 elements.

Selection Sort

Selection sort

```
Procedure select(T[1..n])
  for i = 1 to n-1 do
    minIndex = i;
    minValue = T[i];
    for j = i+1 to n do
      if T[j] < minValue then
        minIndex = j;
        minValue = T[j];
    T[minIndex] = T[i];
    T[i] = minValue
```

For 1 to n-1 elements.

Find the minimum element. Store the index in minIndex and store the value in minValue.

Selection Sort

Selection sort

```
Procedure select(T[1..n])
  for i = 1 to n-1 do
     minIndex = i;
     minValue = T[i];
     for j = i+1 to n do
        if T[j] < minValue then
           minIndex = j;
           minValue = T[j];
     T[minIndex] = T[i];
     T[i] = minValue
```

For 1 to n-1 elements.

Find the minimum element. Store the index in minIndex and store the value in minValue.

Swap the minimum element with the first element in the unsorted part of the list

Selection Sort

Selection sort's efficiency:

- $n - 1$ times through the outer loop.
- $n - i$ times through each inner loop.
- On average, ${}^{n}/_{2}$ times through each inner loop.
- $\frac{n \times (n-1)}{2}$ total inner loops.
- Algorithm is in $\Theta(n^2)$
- Algorithm is in $O(n^2)$
- Algorithm is in $\Omega(n^2)$

Insertion Sort

Insertion Sort

Insertion sort

```
procedure insert(T[1..n])
  for i = 2 to n do
    x = T[i];
    j = i – 1
    while j > 0 and x < T[j] do
        T[j + 1] = T[j]
        j = j – 1
    T[j + 1] = x
```

From the 2nd element onwards.

Take the first element in the remaining unsorted elements.

If T[j] is smaller than the taken value, x, move T[j] backwards 1 position.

Insert value into the correct position.

Insertion Sort

1. Every iteration of insertion sort the first element of the unsorted sublist from the input data is transferred to the sorted sublist by inserting it into the correct position.
2. Step 1 is repeated until no input element remain.
3. The resulting list after k iteration has the property where the first k entries are sorted.

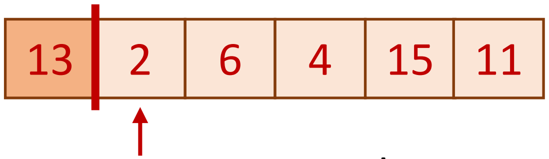
Insertion Sort

Unsorted List

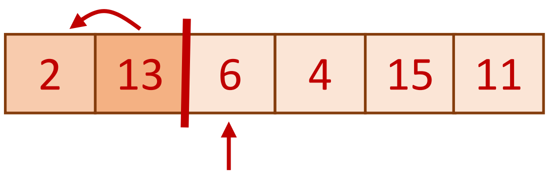
Sorted List

13	2	6	4	15	11

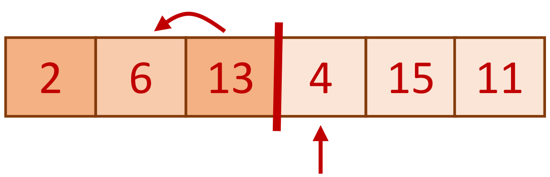
Start processing:



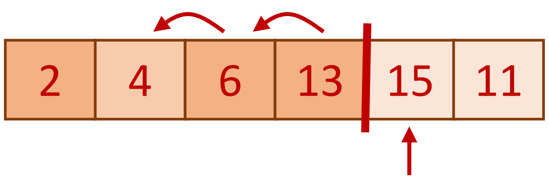
Insert new item to the sorted list.



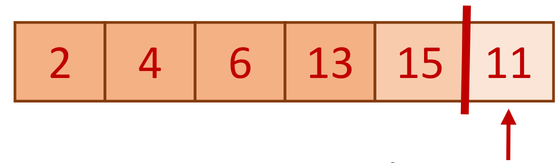
Insert new item to the sorted list.



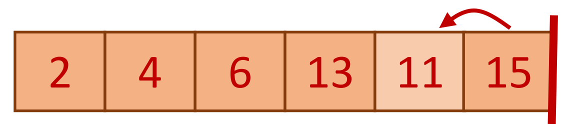
Insert new item to the sorted list.



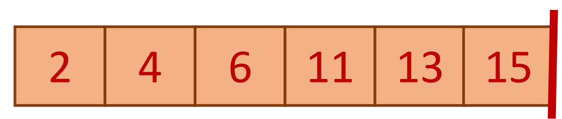
Insert new item to the sorted list.



Insert new item to the sorted list.



Insert new item to the sorted list.



The list is sorted.

End processing:

Insertion Sort

Insertion sort

```
procedure insert(T[1..n])
    for i = 2 to n do
        x = T[i];
        j = i – 1
        while j > 0 and x < T[j] do
            T[j + 1] = T[j]
            j = j – 1
        T[j + 1] = x
```

From the 2nd element onwards.

Insertion Sort

Insertion sort

```
procedure insert(T[1..n])
    for i = 2 to n do
        x = T[i];
        j = i – 1
        while j > 0 and x < T[j] do
            T[j + 1] = T[j]
            j = j – 1
        T[j + 1] = x
```

From the 2nd element onwards.

Take the first element in the remaining unsorted elements.

Insertion Sort

Insertion sort

```
procedure insert(T[1..n])
    for i = 2 to n do
        x = T[i];
        j = i – 1
        while j > 0 and x < T[j] do
            T[j + 1] = T[j]
            j = j – 1
        T[j + 1] = x
```

From the 2nd element onwards.

Take the first element in the remaining unsorted elements.

If T[j] is smaller than the taken value, *x*, move T[j] backwards 1 position.

Insertion Sort

Insertion sort

```
procedure insert(T[1..n])
    for i = 2 to n do
        x = T[i];
        j = i – 1
        while j > 0 and x < T[j] do
            T[j + 1] = T[j]
            j = j – 1
        T[j + 1] = x
```

From the 2nd element onwards.

Take the first element in the remaining unsorted elements.

If T[j] is smaller than the taken value, x, move T[j] backwards 1 position.

Insert value into the correct position.

Insertion Sort

Insertion sort's efficiency:

- $n - 1$ times through the outer loop.
- Variable times through each inner loop.
- On average $\frac{(n-i)}{2}$ times through each inner loop.
- $(n-1)\left(\frac{n-1}{2}\right)$ total inner loops.
- Algorithm is in $\Theta(n^2)$
- Algorithm is in $O(n^2)$
- Algorithm is in $\Omega(n^2)$

Shell Sort

Shell Sort

Shell Sort

- The Shell sort is an improved version of the insertion sort in which diminishing partitions are used to sort the data.
- Shell sort improves insertion sort with two observations:
  - Insertion sort is efficient if the input is "almost sorted", and
  - Insertion sort is typically inefficient because it moves values just one position at a time.

Shell Sort

Method:

- The algorithm makes multiple passes through the list
- At each pass the algorithm sorts a number of segments, a sublist that contains a minimum of N/k elements, using Insertion sort. N is the total number of elements in the list and k is a parameter called *increment.*

Shell Sort

- The value of k is decrease at each pass and thus the size of the set to be sorted gets larger. The process is repeated until the value of k is 1; i.e., the set consists of the entire list.
- Note that as the size of the set increases, the number of sets to be sorted decreases. This sets the insertion sort up for an almost-best case run each iteration with a complexity that approaches $O(n)$.

Shell Sort

Initial list:

A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]	A[9]

Shell Sort

Initial list:

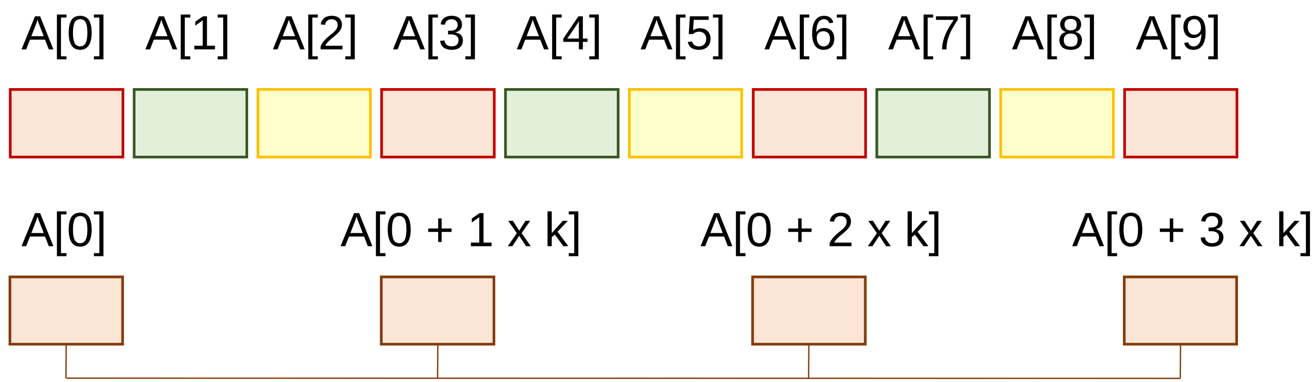
A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]	A[9]

Increment = 3

Shell Sort

Initial list:

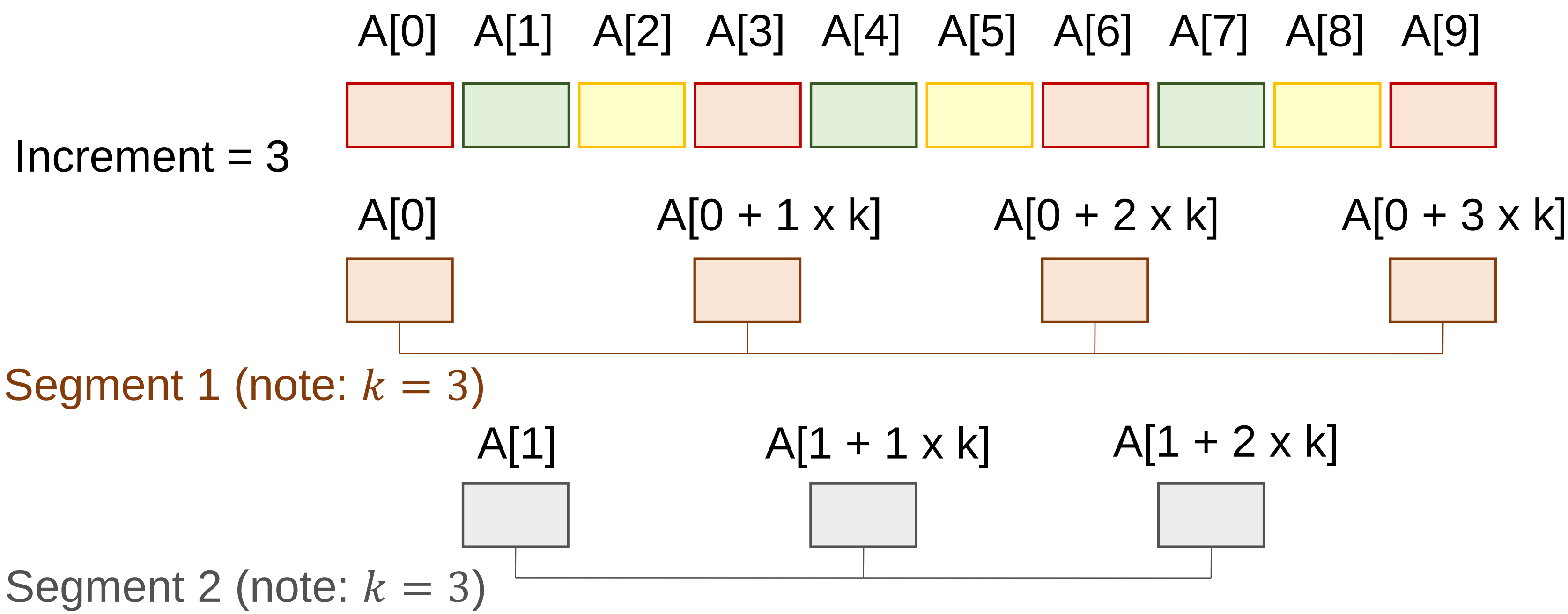
Increment = 3



Segment 1 (note: $k = 3$)

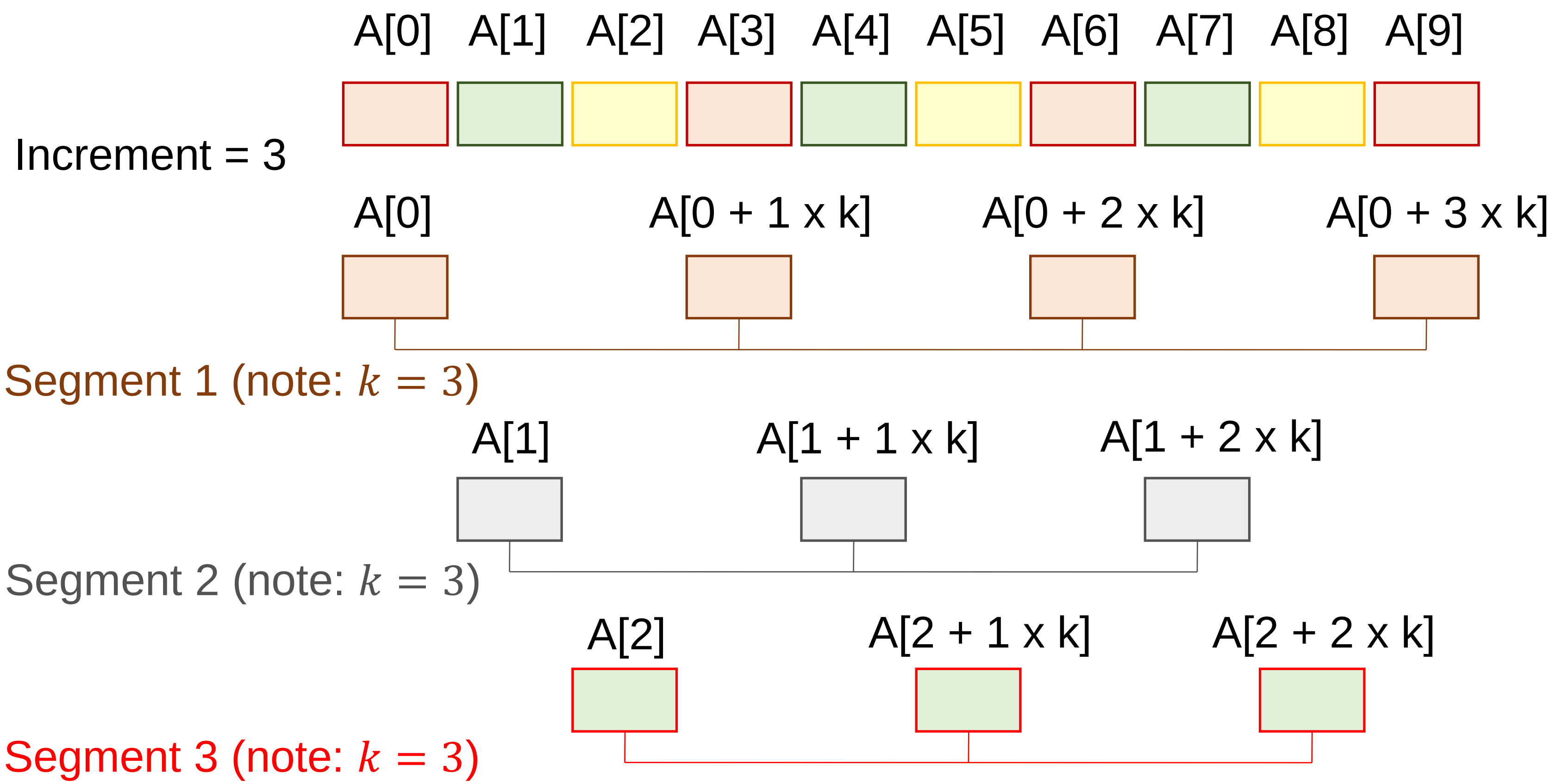
Shell Sort

Initial list:



Shell Sort

Initial list:



Shell Sort

First Increment: k = 5

A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]	A[9]
77	62	14	9	30	21	80	25	70	55

Shell Sort

First Increment: k = 5

A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]	A[9]
77	62	14	9	30	21	80	25	70	55
21	62	14	9	30	77	80	25	70	55

Shell Sort

First Increment: k = 5

A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]	A[9]
77	62	14	9	30	21	80	25	70	55
21	62	14	9	30	77	80	25	70	55
21	62	14	9	30	77	80	25	70	55

Shell Sort

First Increment: k = 5

A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]	A[9]
77	62	14	9	30	21	80	25	70	55
21	62	14	9	30	77	80	25	70	55
21	62	14	9	30	77	80	25	70	55
21	62	14	9	30	77	80	25	70	55

Shell Sort

First Increment: k = 5

A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]	A[9]
77	62	14	9	30	21	80	25	70	55
21	62	14	9	30	77	80	25	70	55
21	62	14	9	30	77	80	25	70	55
21	62	14	9	30	77	80	25	70	55
21	62	14	9	30	77	80	25	70	55

Shell Sort

First Increment: k = 5

A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]	A[9]
77	62	14	9	30	21	80	25	70	55
21	62	14	9	30	77	80	25	70	55
21	62	14	9	30	77	80	25	70	55
21	62	14	9	30	77	80	25	70	55
21	62	14	9	30	77	80	25	70	55
21	62	14	9	30	77	80	25	70	55

Shell Sort

Second Increment: k = 2

A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]	A[9]
21	62	14	9	30	77	80	25	70	55

Shell Sort

Second Increment: k = 2 (Segment 1)

A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]	A[9]
21	62	14	9	30	77	80	25	70	55
14	62	21	9	30	77	80	25	70	55

Shell Sort

Second Increment: k = 2 (Segment 1)

A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]	A[9]
21	62	14	9	30	77	80	25	70	55
14	62	21	9	30	77	80	25	70	55
14	62	21	9	30	77	80	25	70	55

Shell Sort

Second Increment: k = 2 (Segment 1)

A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]	A[9]
21	62	14	9	30	77	80	25	70	55
14	62	21	9	30	77	80	25	70	55
14	62	21	9	30	77	80	25	70	55
14	62	21	9	30	77	80	25	70	55

Shell Sort

Second Increment: k = 2 (Segment 1)

A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]	A[9]
21	62	14	9	30	77	80	25	70	55
14	62	21	9	30	77	80	25	70	55
14	62	21	9	30	77	80	25	70	55
14	62	21	9	30	77	80	25	70	55
14	62	21	9	30	77	70	25	80	55

Shell Sort

Second Increment: k = 2 (Segment 2)

A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]	A[9]
14	62	21	9	30	77	70	25	80	55

Shell Sort

Second Increment: k = 2 (Segment 2)

A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]	A[9]
14	62	21	9	30	77	70	25	80	55
14	9	21	62	30	77	70	25	80	55

Shell Sort

Second Increment: k = 2 (Segment 2)

A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]	A[9]
14	62	21	9	30	77	70	25	80	55
14	9	21	62	30	77	70	25	80	55
14	9	21	62	30	77	70	25	80	55

Shell Sort

Second Increment: k = 2 (Segment 2)

A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]	A[9]
14	62	21	9	30	77	70	25	80	55
14	9	21	62	30	77	70	25	80	55
14	9	21	62	30	77	70	25	80	55
14	9	21	25	30	62	70	77	80	55

Shell Sort

Second Increment: k = 2 (Segment 2)

A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]	A[9]
14	62	21	9	30	77	70	25	80	55
14	9	21	62	30	77	70	25	80	55
14	9	21	62	30	77	70	25	80	55
14	9	21	25	30	62	70	77	80	55
14	9	21	25	30	55	70	62	80	77

Shell Sort

Third Increment: k = 1

14	9	21	25	30	55	70	62	80	77

Shell Sort

Third Increment: k = 1

14	9	21	25	30	55	70	62	80	77

9	14	21	25	30	55	70	62	80	77

Shell Sort

Third Increment: k = 1

14	9	21	25	30	55	70	62	80	77
9	14	21	25	30	55	70	62	80	77
9	14	21	25	30	55	70	62	80	77

Shell Sort

Third Increment: k = 1

14	9	21	25	30	55	70	62	80	77
9	14	21	25	30	55	70	62	80	77
9	14	21	25	30	55	70	62	80	77
9	14	21	25	30	55	70	62	80	77

Shell Sort

Third Increment: k = 1

14	9	21	25	30	55	70	62	80	77
9	14	21	25	30	55	70	62	80	77
9	14	21	25	30	55	70	62	80	77
9	14	21	25	30	55	70	62	80	77
9	14	21	25	30	55	70	62	80	77

Shell Sort

Third Increment: k = 1

14	9	21	25	30	55	70	62	80	77
9	14	21	25	30	55	70	62	80	77
9	14	21	25	30	55	70	62	80	77
9	14	21	25	30	55	70	62	80	77
9	14	21	25	30	55	70	62	80	77
9	14	21	25	30	55	70	62	80	77

Shell Sort

Third Increment: k = 1

14	9	21	25	30	55	70	62	80	77
9	14	21	25	30	55	70	62	80	77
9	14	21	25	30	55	70	62	80	77
9	14	21	25	30	55	70	62	80	77
9	14	21	25	30	55	70	62	80	77
9	14	21	25	30	55	70	62	80	77
9	14	21	25	30	55	70	62	80	77

Shell Sort

Third Increment: k = 1

9	14	21	25	30	55	70	62	80	77

Shell Sort

Third Increment: k = 1

9	14	21	25	30	55	70	62	80	77
9	14	21	25	30	55	62	70	80	77

Shell Sort

Third Increment: k = 1

9	14	21	25	30	55	70	62	80	77
9	14	21	25	30	55	62	70	80	77
9	14	21	25	30	55	62	70	80	77

Shell Sort

Third Increment: k = 1

9	14	21	25	30	55	70	62	80	77
9	14	21	25	30	55	62	70	80	77
9	14	21	25	30	55	62	70	80	77
9	14	21	25	30	55	62	70	77	80

Shell Sort

```
procedure shellsort(T[1..n])
 inc = n
 while inc > 1 do
     inc = inc ÷ 2
     for j = 1 to inc do
         k = j + inc
         while k ≤ n do
             done = false
             x = T[k]
             current = k; previous = current – inc
```

Shell Sort

Insertion Sort

```
while previous ≥ j and not done do
    if x < T[previous] then
        T[current] = T[previous]
        current = previous
        previous = previous – inc
    else
        done = true
        T[current] = x
        k = k + inc
```

Shell Sort

Shell sort's efficiency:

- Analysis of Shell sort is difficult.
- Intervals $\frac{n}{2}, \frac{n}{4}, \ldots, 1$ are not optimal but are easy to compute.
- With these intervals, worst case is in $O(n^2)$
- Better intervals are $1, 3, 7, \ldots, 2^m - 1$ (Hibbard), where $m = 2,3,4, \ldots$
- With these intervals, worst case is in $O\left(n^{3/2}\right)$

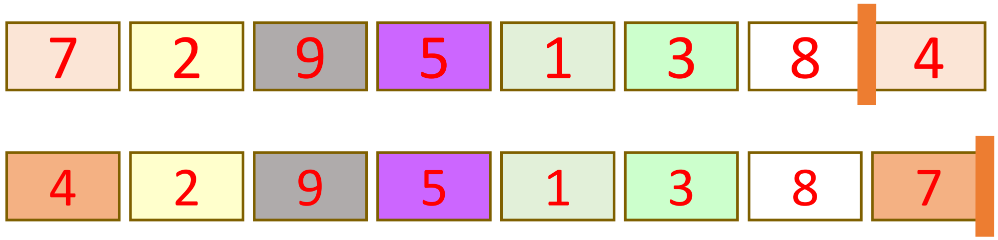
Shell Sort – Hibbard Interval

First Increment: k = 7

7	2	9	5	1	3	8	4

Shell Sort – Hibbard Interval

First Increment: k = 7



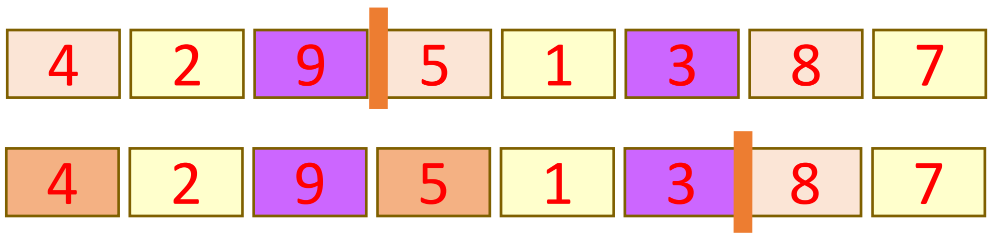
Shell Sort – Hibbard Interval

Second Increment: k = 3 (segment 1)

4	2	9	5	1	3	8	7

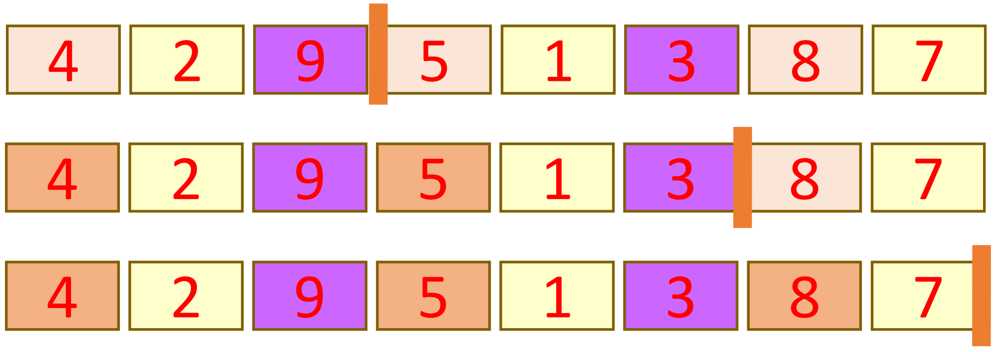
Shell Sort – Hibbard Interval

Second Increment: k = 3 (segment 1)



Shell Sort – Hibbard Interval

Second Increment: k = 3 (segment 1)



Shell Sort – Hibbard Interval

Second Increment: k = 3 (segment 2)

4	2	9	5	1	3	8	7

Shell Sort – Hibbard Interval

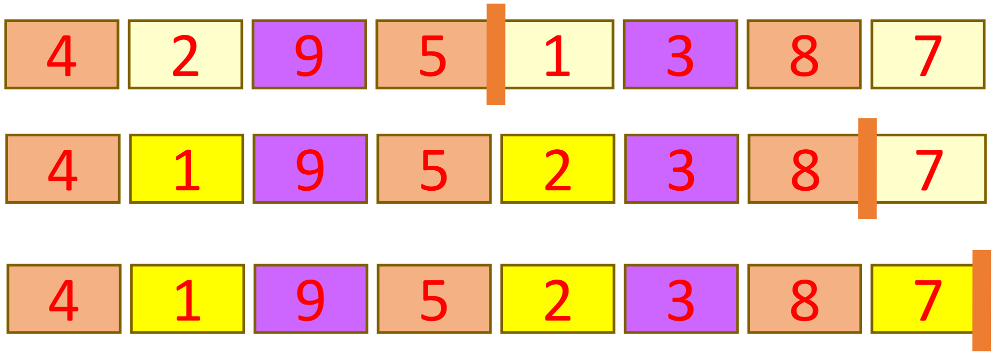
Second Increment: k = 3 (segment 2)

4	2	9	5	1	3	8	7

4	1	9	5	2	3	8	7

Shell Sort – Hibbard Interval

Second Increment: k = 3 (segment 2)



Shell Sort – Hibbard Interval

Second Increment: k = 3 (segment 3)

4	1	9	5	2	3	8	7

Shell Sort – Hibbard Interval

Second Increment: k = 3 (segment 3)

4	1	9	5	2	3	8	7

4	1	3	5	2	9	8	7

Shell Sort – Hibbard Interval

Third Increment: k = 1

4	1	3	5	2	9	8	7

Shell Sort – Hibbard Interval

Third Increment: k = 1

4	1	3	5	2	9	8	7

<u>1</u>	4	3	5	2	9	8	7

Shell Sort – Hibbard Interval

Third Increment: k = 1

4	1	3	5	2	9	8	7
1	4	3	5	2	9	8	7
1	**3**	4	5	2	9	8	7

Shell Sort – Hibbard Interval

Third Increment: k = 1

4	1	3	5	2	9	8	7
1	4	3	5	2	9	8	7
1	3	4	5	2	9	8	7
1	3	4	**<u>5</u>**	2	9	8	7

Shell Sort – Hibbard Interval

Third Increment: k = 1

4	1	3	5	2	9	8	7
1	4	3	5	2	9	8	7
1	3	4	5	2	9	8	7
1	3	4	5	2	9	8	7
1	**<u>2</u>**	3	4	5	9	8	7

Shell Sort – Hibbard Interval

Third Increment: k = 1

4	1	3	5	2	9	8	7
1	4	3	5	2	9	8	7
1	3	4	5	2	9	8	7
1	3	4	5	2	9	8	7
1	2	3	4	5	9	8	7
1	2	3	4	5	**<u>9</u>**	8	7

Shell Sort – Hibbard Interval

Third Increment: k = 1

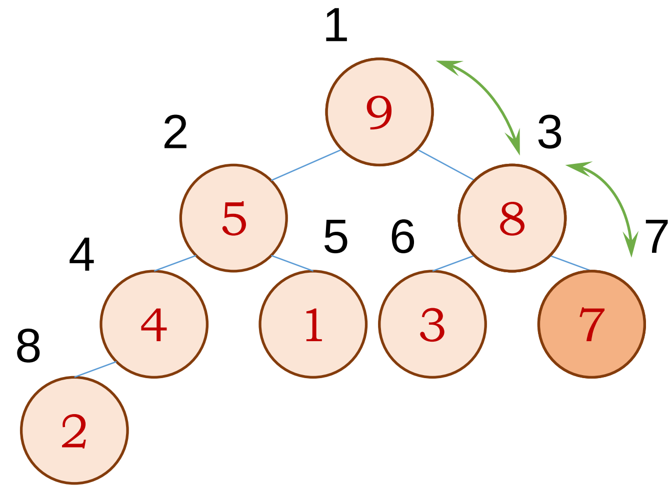
4	1	3	5	2	9	8	7
1	4	3	5	2	9	8	7
1	3	4	5	2	9	8	7
1	3	4	5	2	9	8	7
1	2	3	4	5	9	8	7
1	2	3	4	5	9	8	7
1	2	3	4	5	**8**	9	7

Shell Sort – Hibbard Interval

Third Increment: k = 1

4	1	3	5	2	9	8	7
1	4	3	5	2	9	8	7
1	3	4	5	2	9	8	7
1	3	4	5	2	9	8	7
1	2	3	4	5	9	8	7
1	2	3	4	5	9	8	7
1	2	3	4	5	8	9	7
1	2	3	4	5	**7**	8	9

Heap Sort



Heap Sort

Heap Sort

Heap Sort

- The heap sort algorithm is an improved version of the selection sort in which the largest element (the root) is selected and exchanged with the last element in the unsorted list.
- The inefficiency of selection sort results from the procedure used to select the largest elements (assuming a descending order result) from an unsorted list.

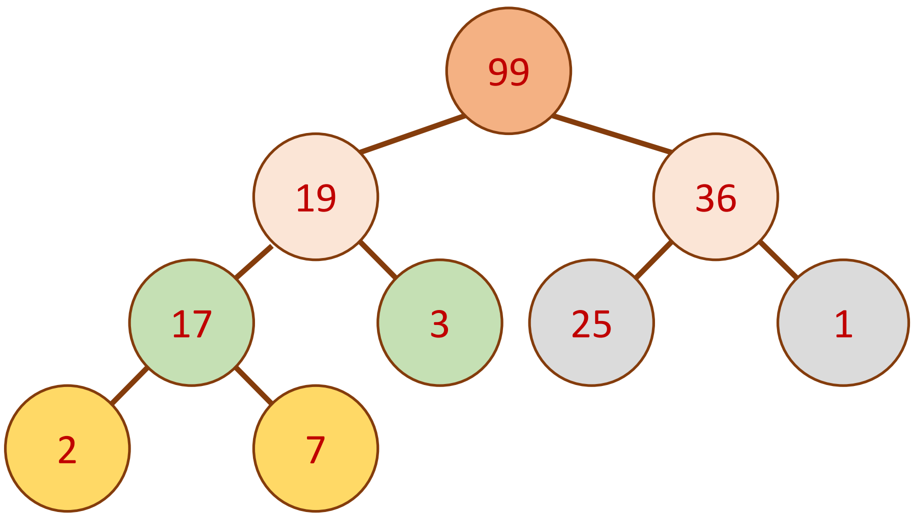
Heap Sort

- The heap sort algorithm emulates the selection sort except that the unsorted list of elements is maintained as a heap structure.
- Since heap structure is a tree structure with the root having the largest element (maximum heap), we do not need to scan the entire tree to locate the largest element; we re-heap and thus reduce the number of steps in locating the largest element.

Heap Sort

Recall that a heap structure is:

- *a binary tree with the following properties:*
  - *The tree is complete or nearly complete*
  - *The key value of each node is greater than or equal to the key value in each of its descendents.*



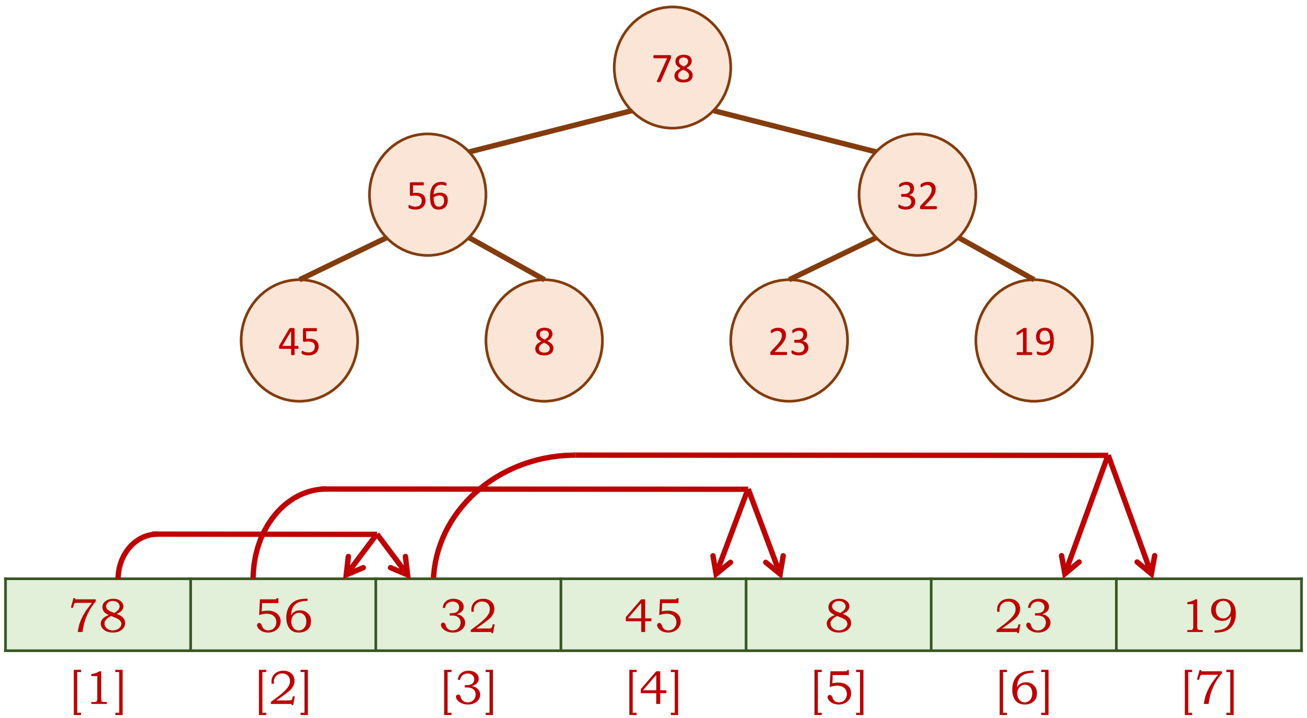
Heap Sort

- A heap can be implemented using an array where
  - $T[1]$ is the root of the tree
  - $T[i]$ is the parent of $T[2i]$ and $T[2i+1]$
  - $T[i] \geq T[2i]$ and $T[i] \geq T[2i+1]$

Heap Sort

- T[1] is the parent of T[2] and T[3]
- T[1] > T[2}, T[1] > T[3} …
- T[2] is the parent of T[4] and T[5]

…



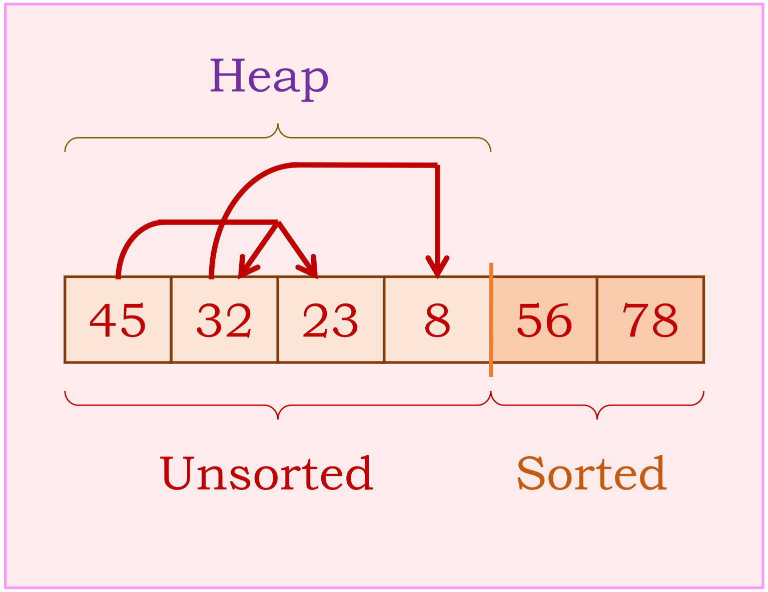
Heap Sort

Heap Sort

Method

1. Turn the array to be sorted into a heap structure (We call this makeheap.)
2. Exchange the root, which is the largest element in the heap, with the last element in the unsorted list, resulted in the largest element being added to the beginning of the sorted list.
3. Re-heap (We call this siftdown or heapify) the unsorted array.
4. Repeat step 2 and 3 until the entire list is sorted.

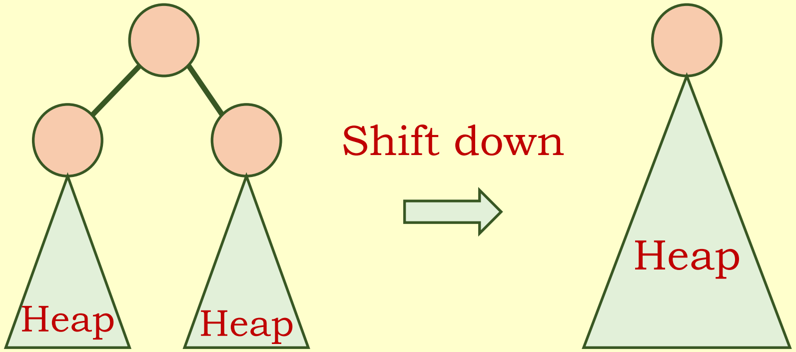
Heap Sort



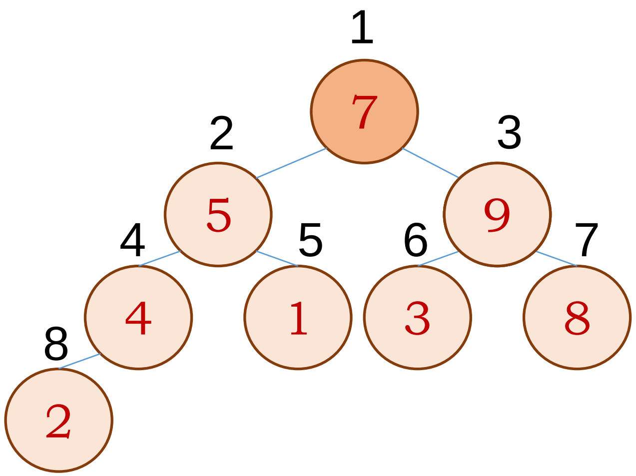
siftdown

Idea:

- *Compare the root with the left sub-tree's root and the right sub-tree's root*
- *Swap root with the greater node (Max Heap)*
- *Repeat until value is placed at the correct position*



Heap Sort



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

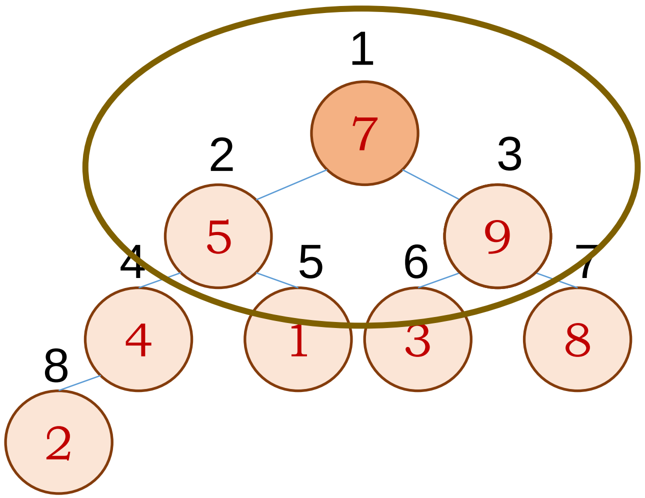
Siftdown:

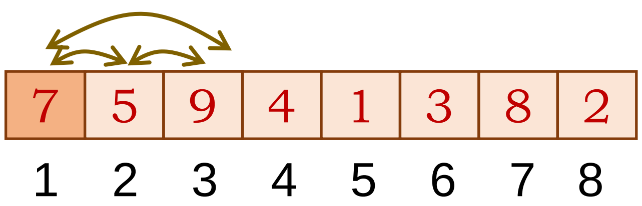
We want to shift **7** down the tree to its correct position.

Idea:

- *Compare the root with the left sub-tree's root and the right sub-tree's root*
- *Swap root with the greater node (Max Heap)*
- *Repeat until 7 is placed at the correct position*

Heap Sort





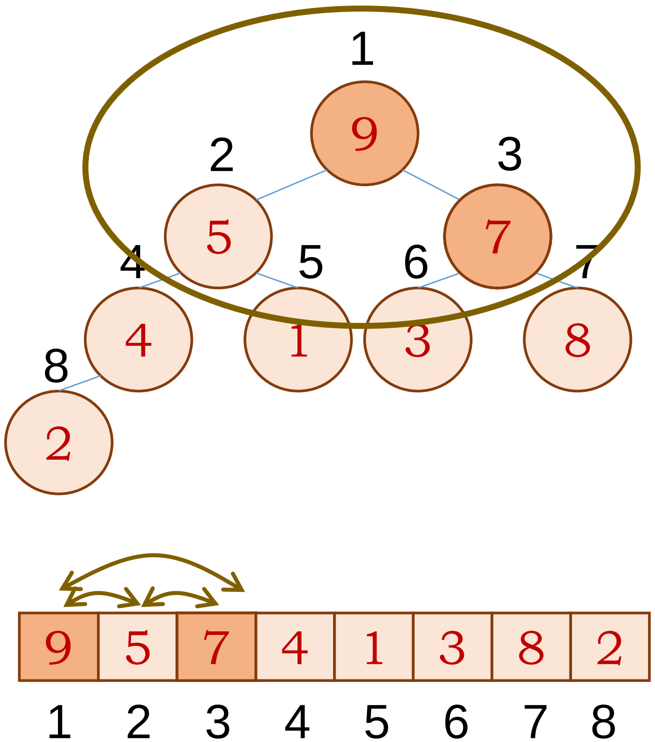
Siftdown:

We want to shift **7** down the tree to its correct position.

Idea:

- *Compare the root with the left sub-tree's root and the right sub-tree's root*
- *Swap root with the greater node (Max Heap)*
- *Repeat until 7 is placed at the correct position*

Heap Sort

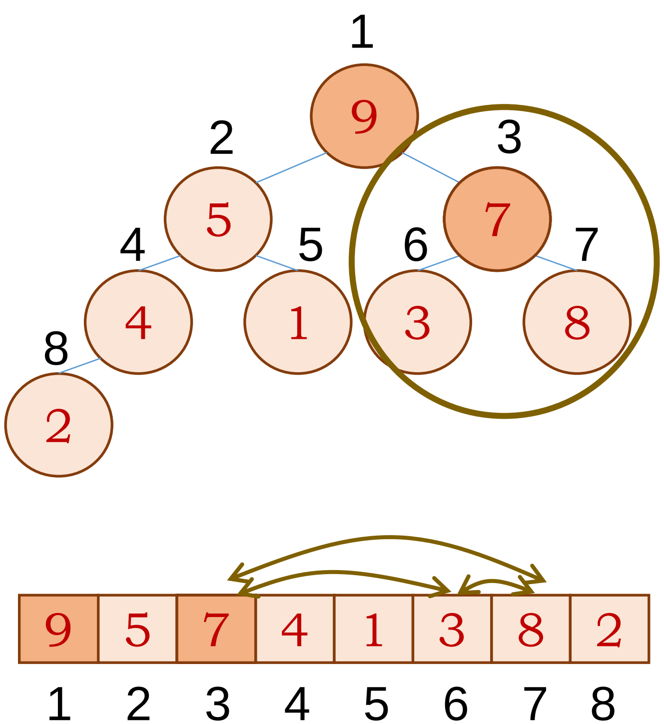


Siftdown:
We want to shift **7** down the tree to its correct position.

Idea:

- *Compare the root with the left sub-tree's root and the right sub-tree's root*
- *Swap root with the greater node (Max Heap)*
- *Repeat until 7 is placed at the correct position*

Heap Sort



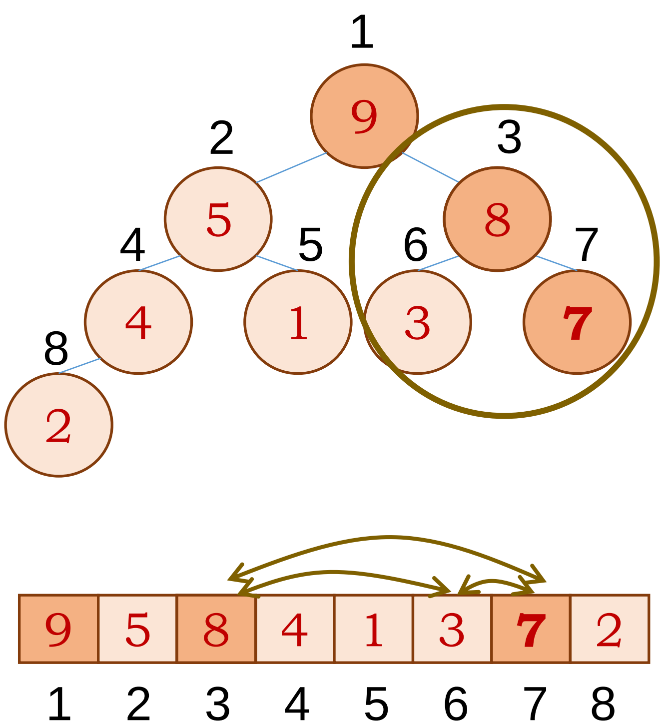
Siftdown:

We want to shift **7** down the tree to its correct position.

Idea:

- *Compare the root with the left sub-tree's root and the right sub-tree's root*
- *Swap root with the greater node (Max Heap)*
- *Repeat until 7 is placed at the correct position*

Heap Sort



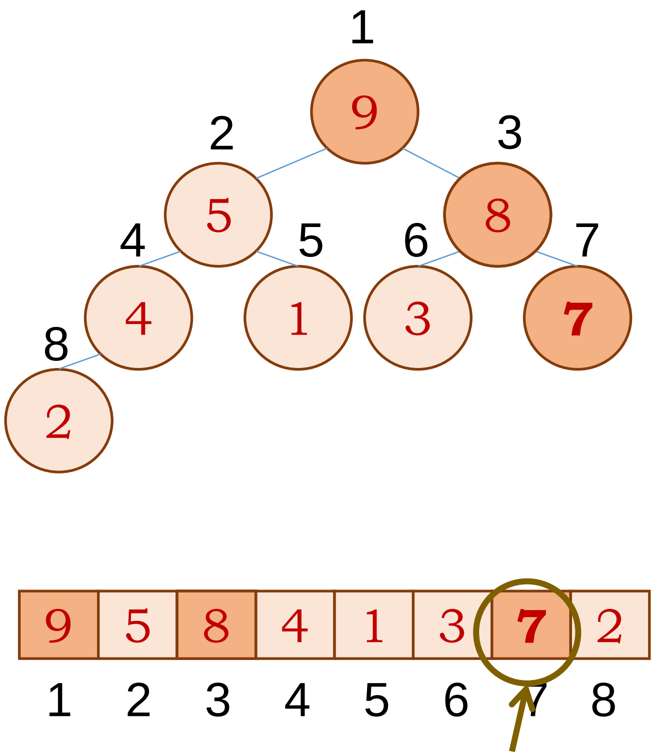
Siftdown:

We want to shift **7** down the tree to its correct position.

Idea:

- *Compare the root with the left sub-tree's root and the right sub-tree's root*
- *Swap root with the greater node (Max Heap)*
- *Repeat until 7 is placed at the correct position*

Heap Sort



Siftdown:

We want to shift **7** down the tree to its correct position.

Idea:

- *Compare the root with the left sub-tree's root and the right sub-tree's root*
- *Swap root with the greater node (Max Heap)*
- *Repeat until 7 is placed at the correct position*

7 is shifted down to the correct position.

Heap Sort

```
Procedure siftdown(T[1 .. n], i)
 k = i
 repeat
    j = k
    if 2j <= n and T[2j] > T[k] then k = 2j
    if 2j + 1 <= n and T[2j + 1] > T[k] then k = 2j + 1
    swap T[j] and T[k]
 until j = k
```

Heap Sort

```
Procedure siftdown(T[1 .. n], i)
 k = i
 repeat
    j = k
    if 2j <= n and T[2j] > T[k] then k = 2j
    if 2j + 1 <= n and T[2j + 1] > T[k] then k = 2j + 1
    swap T[j] and T[k]
 until j = k
```

If there's **left child and** its value is greater than root's value

Heap Sort

```
Procedure siftdown(T[1 .. n], i)
 k = i
 repeat
   j = k
   if 2j <= n and T[2j] > T[k] then k = 2j
   if 2j + 1 <= n and T[2j + 1] > T[k] then k = 2j + 1
   swap T[j] and T[k]
 until j = k
```

If there's **left child and** its value is greater than root's value

If there is **right child and** its value is greater than root's value

Heap Sort

```
Procedure siftdown(T[1 .. n], i)
 k = i
 repeat
    j = k
    if 2j <= n and T[2j] > T[k] then k = 2j
    if 2j + 1 <= n and T[2j + 1] > T[k] then k = 2j + 1
    swap T[j] and T[k]
 until j = k
```

If there's **left child and** its value is greater than root's value

If there is **right child and** its value is greater than root's value

Swap root value with the bigger child's value

Heap Sort

```
Procedure siftdown(T[1 .. n], i)
 k = i
 repeat
   j = k
   if 2j <= n and T[2j] > T[k] then k = 2j
   if 2j + 1 <= n and T[2j + 1] > T[k] then k = 2j + 1
   swap T[j] and T[k]
 until j = k
```

If there's **left child and** its value is greater than root's value

If there is **right child and** its value is greater than root's value

Swap root value with the bigger child's value

If the root has been swapped with one of its child, k will change; thus in this case, we need to repeat siftdown with the sub-tree

Heap Sort

The idea:

- Create a maximum heap.
- Repeat until sorted
  - Exchange the root with the last node.
  - Detach the last node from the heap
  - Reconstruct the heap.

Heapsort

procedure heap($T[1..n]$)

  makeheap(T)

  for $i = n$ to 2 step -1 do

    swap $T[1]$ and $T[i]$

    siftdown($T[1..i-1]$, 1)

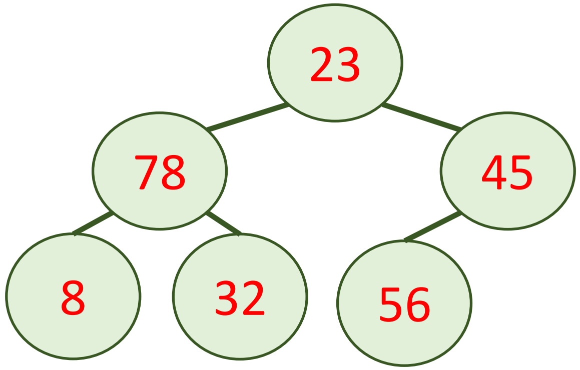
23	78	45	8	32	56

Heap Sort

```
Procedure heap(T[1..n])
        makeheap(T)
        for i = n to 2 step –1do
            swap T[1] and T[i]
            siftdown(T[1 .. I – 1], 1)
```

Note: Assuming we are making a maximum heap.

23	78	45	8	32	56

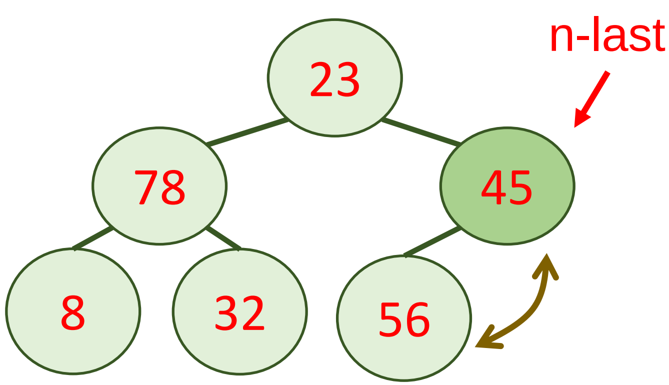


Heap Sort

```
Procedure heap(T[1..n])
        makeheap(T)
        for i = n to 2 step –1do
            swap T[1] and T[i]
            siftdown(T[1 .. I – 1], 1)
```

Note: Assuming we are making a maximum heap.

1. Start from n-last until the root, check for heap-value constraint (order property).



23	78	45	8	32	56

n-last is the parent of the last node of a heap structure.

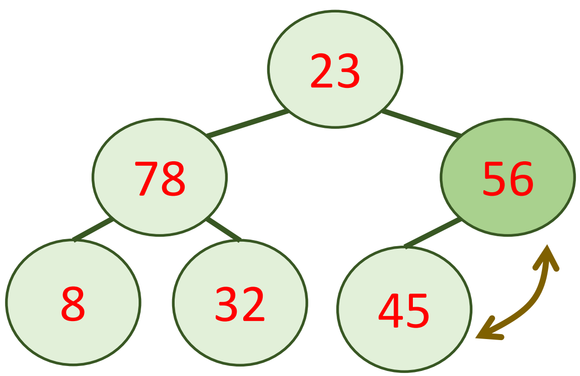
Heap Sort

```
Procedure heap(T[1..n])
        makeheap(T)
        for i = n to 2 step –1do
            swap T[1] and T[i]
            siftdown(T[1 .. I – 1], 1)
```

Note: Assuming we are making a maximum heap.

1. Start from n-last until the root, check for heap-value constraint (order property).

23	78	56	8	32	45

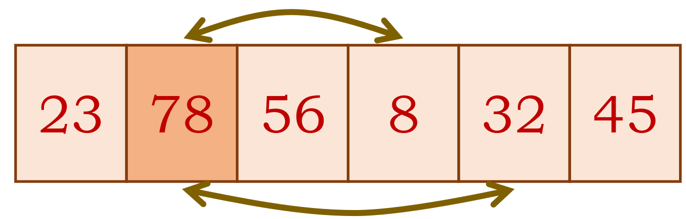


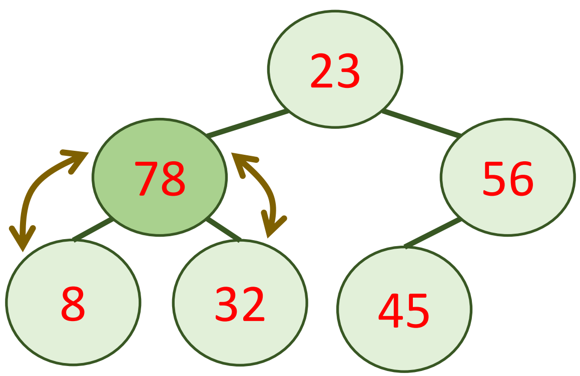
2. Heap-value constraint is violated. Swap the value to correct it.

Heap Sort

```
Procedure heap(T[1..n])
        makeheap(T)
        for i = n to 2 step –1do
            swap T[1] and T[i]
            siftdown(T[1 .. I – 1], 1)
```

Note: Assuming we are making a maximum heap.

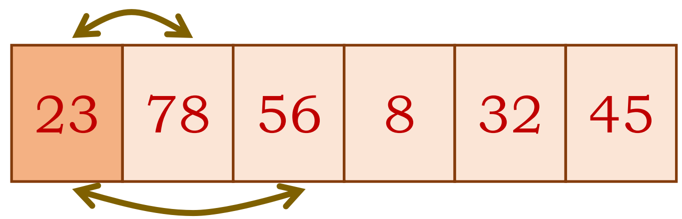


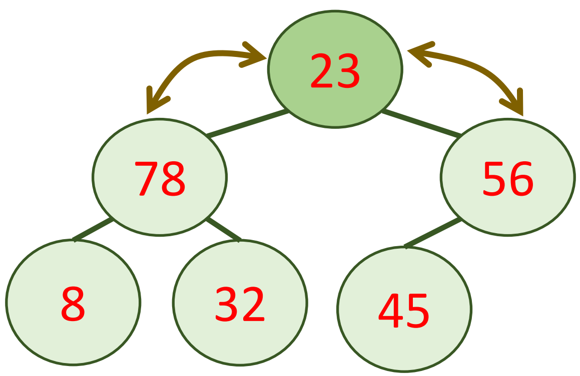


Heap Sort

Procedure heap(T[1..n])
    makeheap(T)
    for i = n to 2 step –1do
        swap T[1] and T[i]
        siftdown(T[1 .. I – 1], 1)

Note: Assuming we are making a maximum heap.

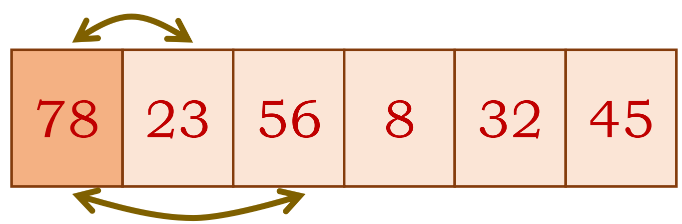


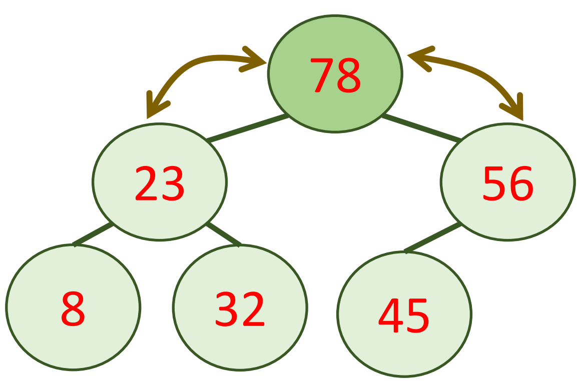


Heap Sort

```
Procedure heap(T[1..n])
    makeheap(T)
    for i = n to 2 step –1do
        swap T[1] and T[i]
        siftdown(T[1 .. I – 1], 1)
```

Note: Assuming we are making a maximum heap.

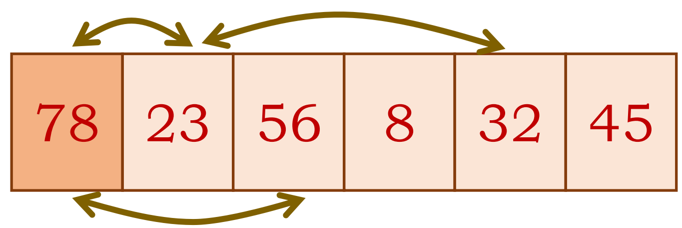


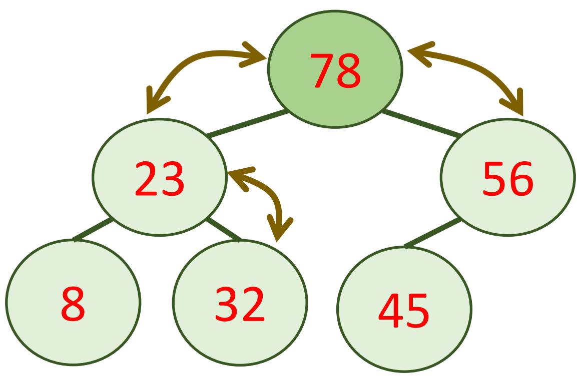


Heap Sort

```
Procedure heap(T[1..n])
    makeheap(T)
    for i = n to 2 step –1do
        swap T[1] and T[i]
        siftdown(T[1 .. I – 1], 1)
```

Note: Assuming we are making a maximum heap.

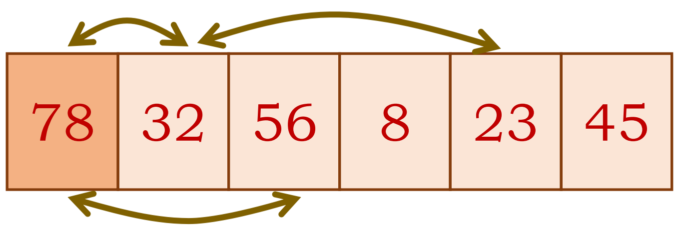


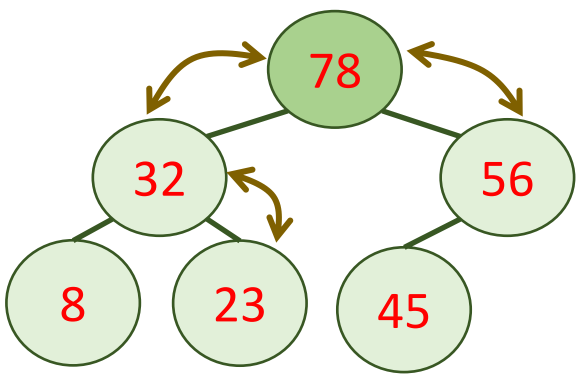


Heap Sort

```
Procedure heap(T[1..n])
        makeheap(T)
        for i = n to 2 step –1do
            swap T[1] and T[i]
            siftdown(T[1 .. I – 1], 1)
```

Note: Assuming we are making a maximum heap.



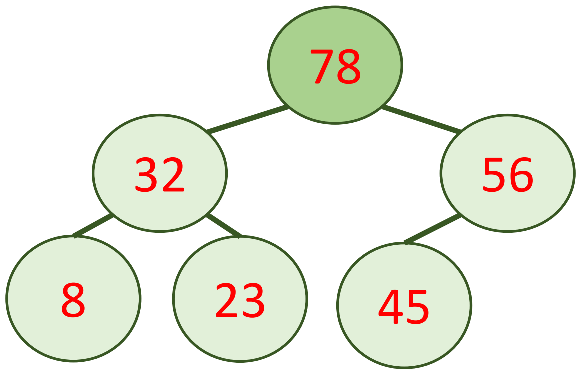


Heap Sort

```
Procedure heap(T[1..n])
        makeheap(T)
        for i = n to 2 step –1do
            swap T[1] and T[i]
            siftdown(T[1 .. I – 1], 1)
```

Note: Assuming we are making a maximum heap.

78	32	56	8	23	45

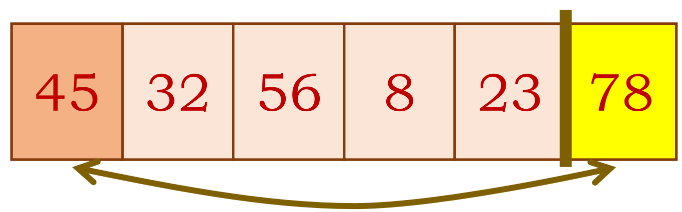


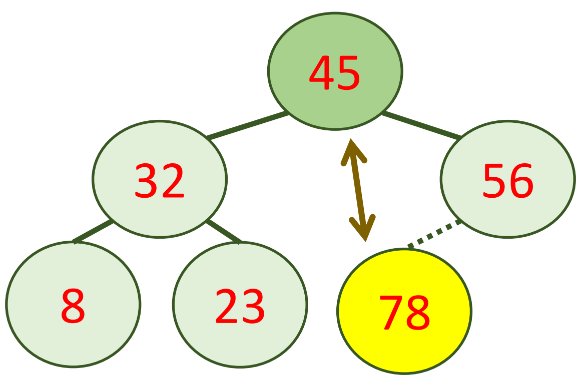
A maximum heap is constructed.

Heap Sort

```
Procedure heap(T[1..n])
        makeheap(T)
        for i = n to 2 step –1do
            swap T[1] and T[i]
            siftdown(T[1 .. I – 1], 1)
```

Note: Assuming we are making a maximum heap.



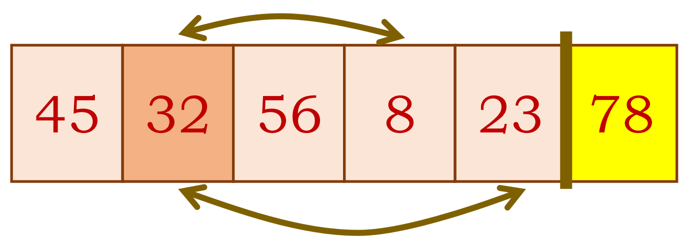


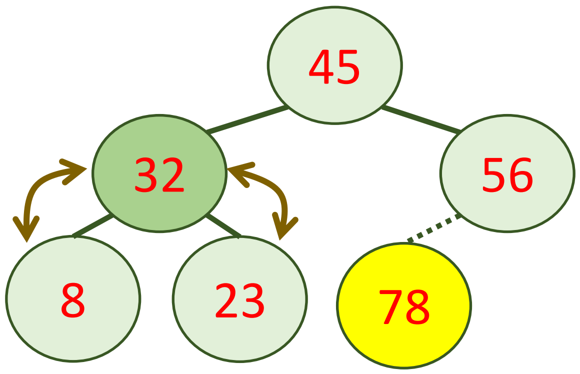
Swap the root and the last element of the Heap, and detach it (logically) from the heap.

Heap Sort

Procedure heap(T[1..n])
    makeheap(T)
    for i = n to 2 step –1do
        swap T[1] and T[i]
        siftdown(T[1 .. I – 1], 1)

Note: Assuming we are making a maximum heap.



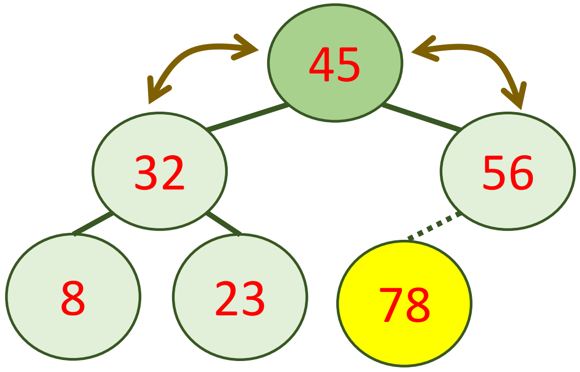


Heap Sort

```
Procedure heap(T[1..n])
        makeheap(T)
        for i = n to 2 step –1do
           swap T[1] and T[i]
           siftdown(T[1 .. I – 1], 1)
```

Note: Assuming we are making a maximum heap.

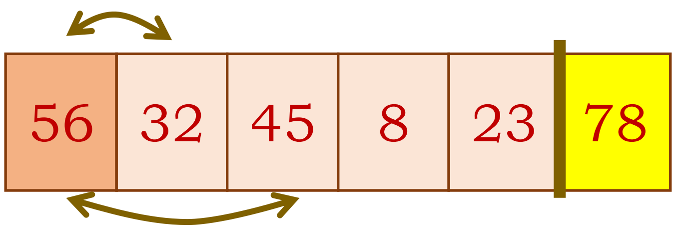
45	32	56	8	23	78

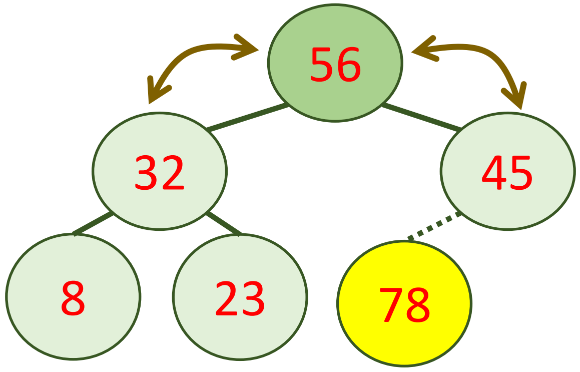


Heap Sort

Procedure heap(T[1..n])
makeheap(T)
for i = n to 2 step –1do
swap T[1] and T[i]
siftdown(T[1 .. I – 1], 1)

Note: Assuming we are making a maximum heap.



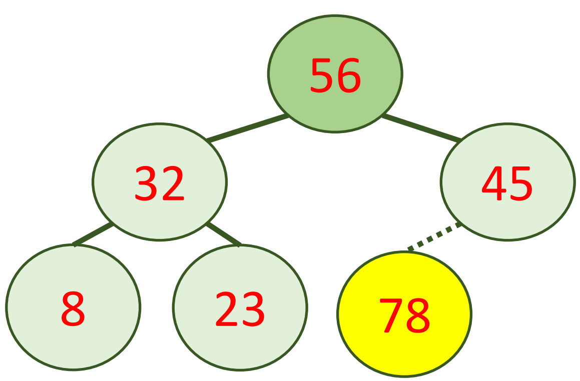


Heap Sort

```
Procedure heap(T[1..n])
        makeheap(T)
        for i = n to 2 step –1do
           swap T[1] and T[i]
           siftdown(T[1 .. I – 1], 1)
```

Note: Assuming we are making a maximum heap.

56	32	45	8	23	78

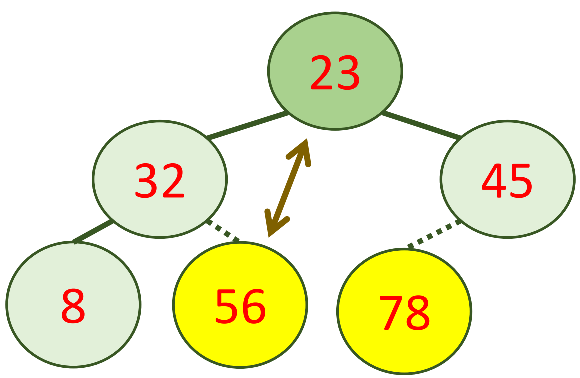


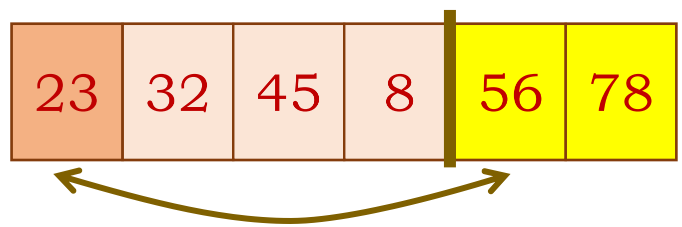
A maximum heap is constructed.

Heap Sort

```
Procedure heap(T[1..n])
        makeheap(T)
        for i = n to 2 step –1do
           swap T[1] and T[i]
           siftdown(T[1 .. I – 1], 1)
```

Note: Assuming we are making a maximum heap.





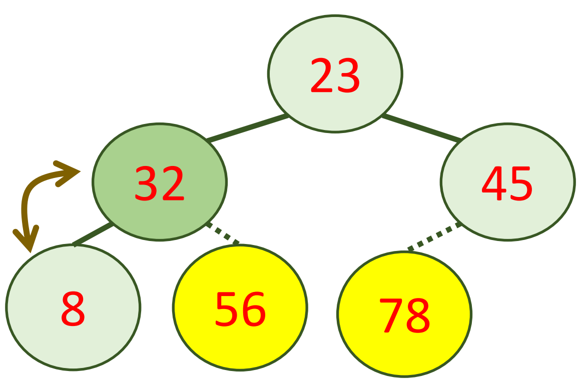
Swap the root and the last element of the Heap, and detach it (logically) from the heap.

Heap Sort

```
Procedure heap(T[1..n])
        makeheap(T)
        for i = n to 2 step –1do
            swap T[1] and T[i]
            siftdown(T[1 .. I – 1], 1)
```

Note: Assuming we are making a maximum heap.

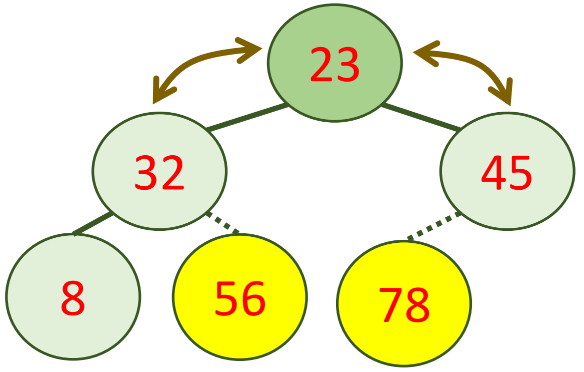
23	32	45	8	56	78

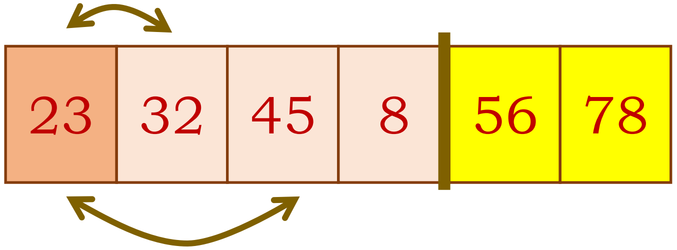


Heap Sort

Procedure heap(T[1..n])
makeheap(T)
for i = n to 2 step –1do
swap T[1] and T[i]
siftdown(T[1 .. I – 1], 1)

Note: Assuming we are making a maximum heap.

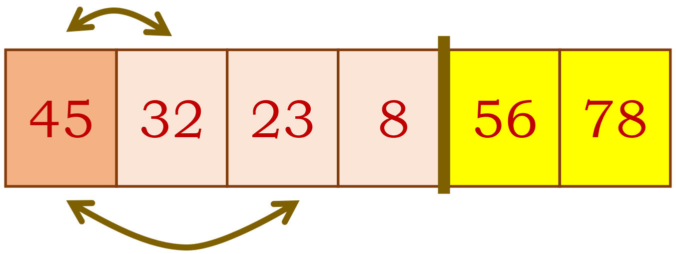


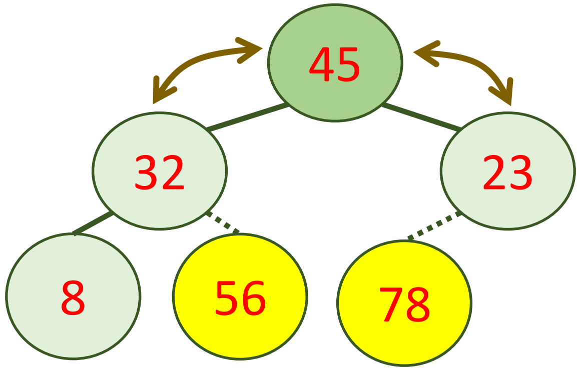


Heap Sort

```
Procedure heap(T[1..n])
        makeheap(T)
        for i = n to 2 step –1do
           swap T[1] and T[i]
           siftdown(T[1 .. I – 1], 1)
```

Note: Assuming we are making a maximum heap.



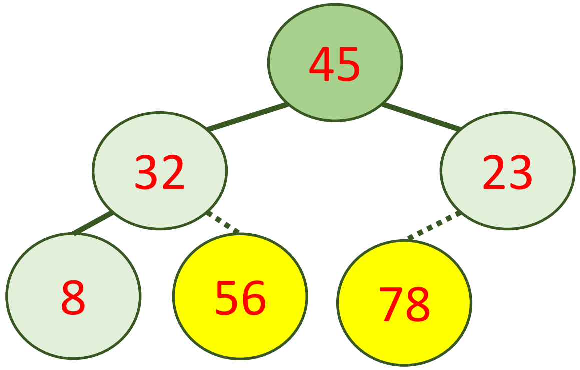


Heap Sort

```
Procedure heap(T[1..n])
        makeheap(T)
        for i = n to 2 step –1do
           swap T[1] and T[i]
           siftdown(T[1 .. I – 1], 1)
```

Note: Assuming we are making a maximum heap.

45	32	23	8	56	78

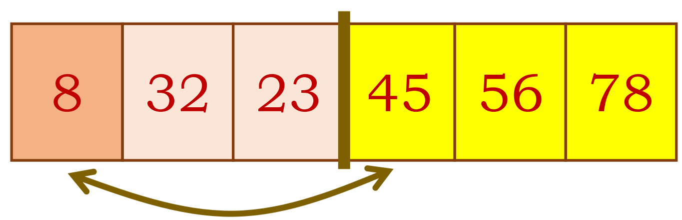


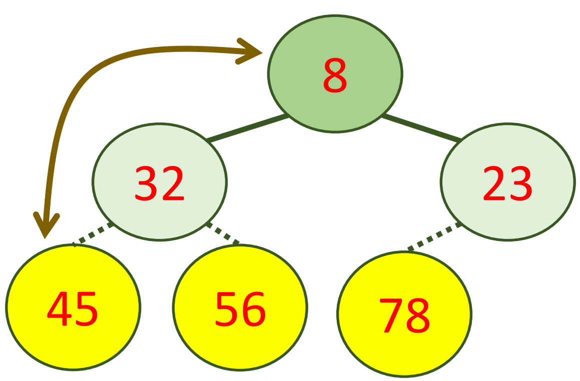
A maximum heap is constructed.

Heap Sort

```
Procedure heap(T[1..n])
        makeheap(T)
        for i = n to 2 step –1do
   →       swap T[1] and T[i]
           siftdown(T[1 .. I – 1], 1)
```

Note: Assuming we are making a maximum heap.



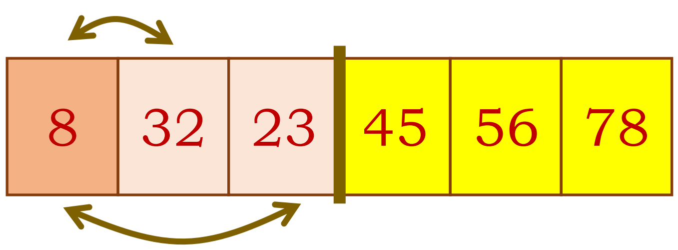


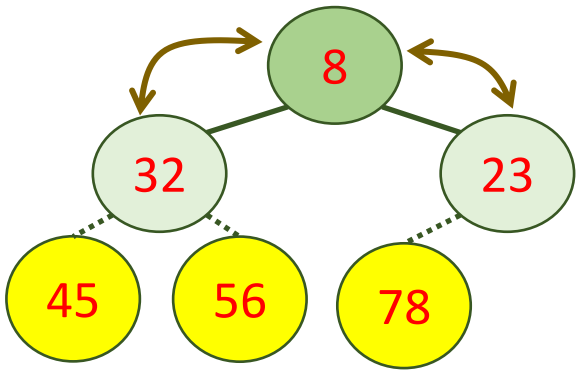
Swap the root and the last element of the Heap, and detach it (logically) from the heap.

Heap Sort

```
Procedure heap(T[1..n])
        makeheap(T)
        for i = n to 2 step –1do
           swap T[1] and T[i]
           siftdown(T[1 .. I – 1], 1)
```

Note: Assuming we are making a maximum heap.

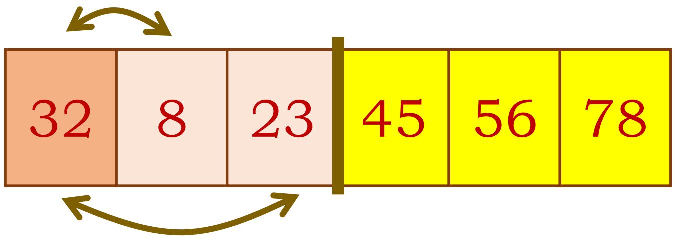


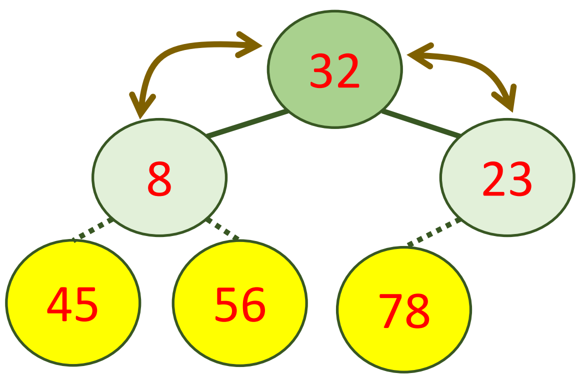


Heap Sort

Procedure heap(T[1..n])
    makeheap(T)
    for i = n to 2 step –1do
        swap T[1] and T[i]
        siftdown(T[1 .. I – 1], 1)

Note: Assuming we are making a maximum heap.



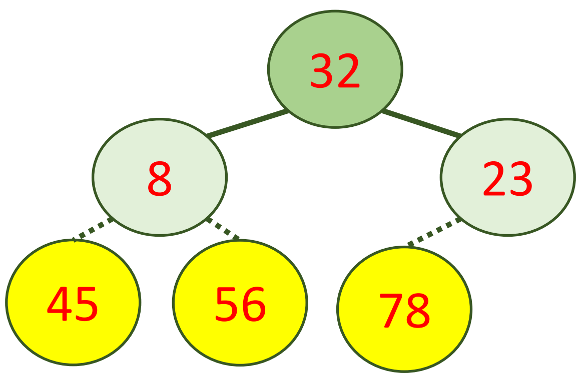


Heap Sort

```
Procedure heap(T[1..n])
        makeheap(T)
        for i = n to 2 step –1do
           swap T[1] and T[i]
           siftdown(T[1 .. I – 1], 1)
```

Note: Assuming we are making a maximum heap.

32	8	23	45	56	78

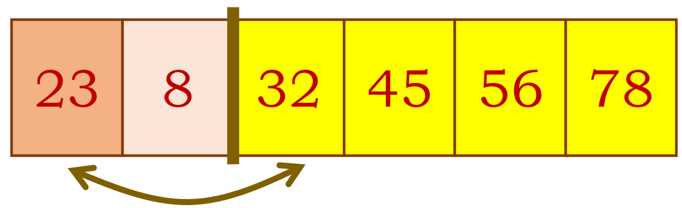


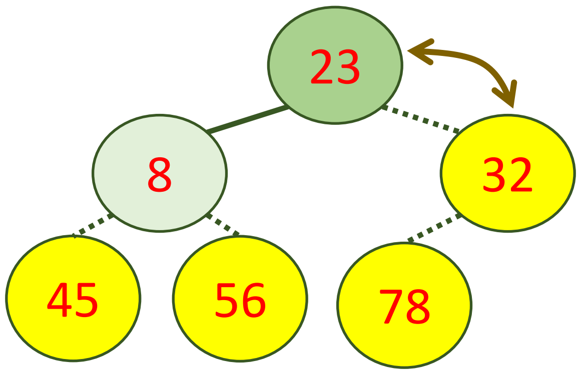
A maximum heap is constructed.

Heap Sort

Procedure heap(T[1..n])
makeheap(T)
for i = n to 2 step –1do
swap T[1] and T[i]
siftdown(T[1 .. I – 1], 1)

Note: Assuming we are making a maximum heap.





Swap the root and the last element of the Heap, and detach it (logically) from the heap.

Heap Sort

Procedure heap(T[1..n])

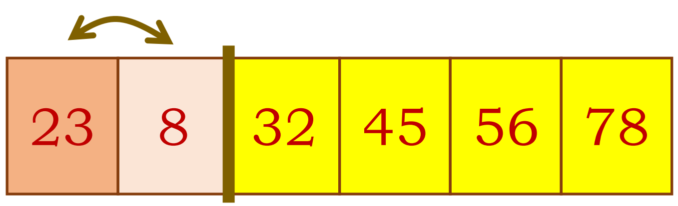
makeheap(T)

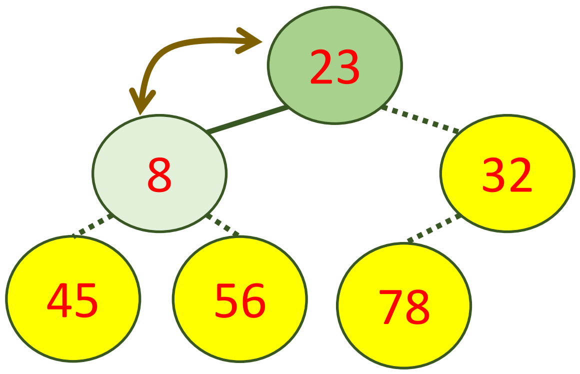
for i = n to 2 step –1do

swap T[1] and T[i]

siftdown(T[1 .. I – 1], 1)

Note: Assuming we are making a maximum heap.

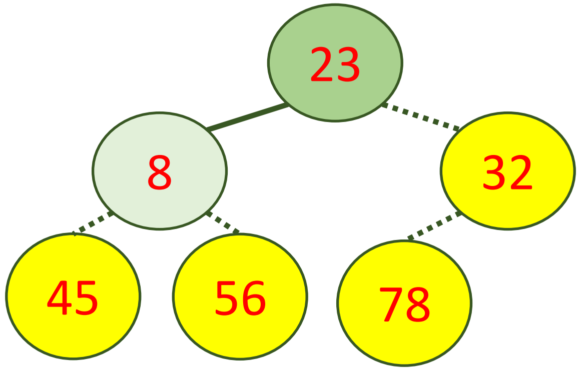


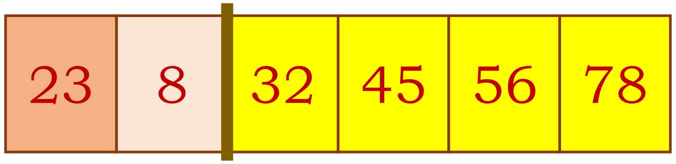


Heap Sort

```
Procedure heap(T[1..n])
        makeheap(T)
        for i = n to 2 step –1do
           swap T[1] and T[i]
           siftdown(T[1 .. I – 1], 1)
```

Note: Assuming we are making a maximum heap.



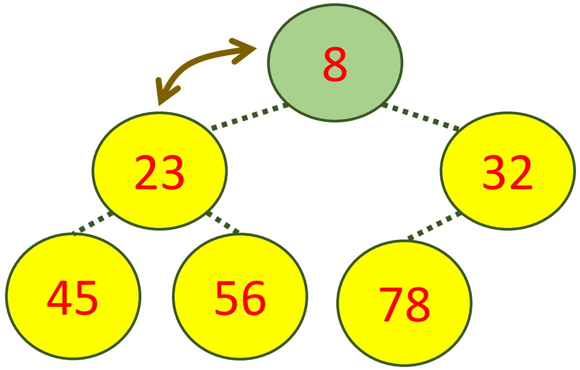


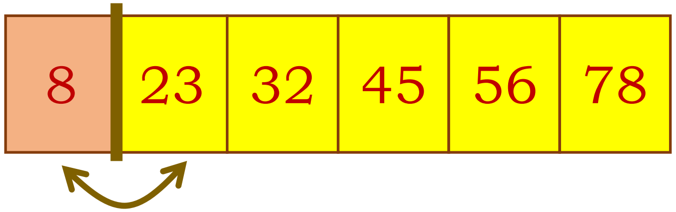
A maximum heap is constructed.

Heap Sort

```
Procedure heap(T[1..n])
        makeheap(T)
        for i = n to 2 step –1do
           swap T[1] and T[i]
           siftdown(T[1 .. I – 1], 1)
```

Note: Assuming we are making a maximum heap.





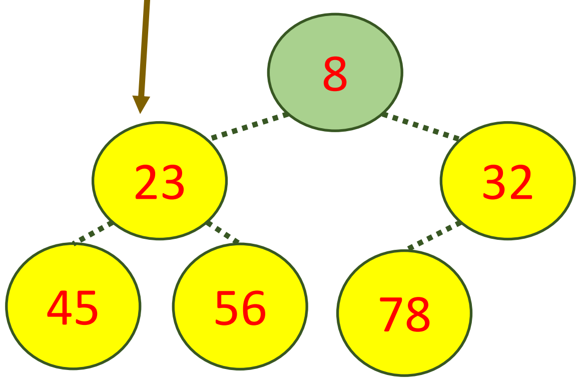
Swap the root and the last element of the Heap, and detach it (logically) from the heap.

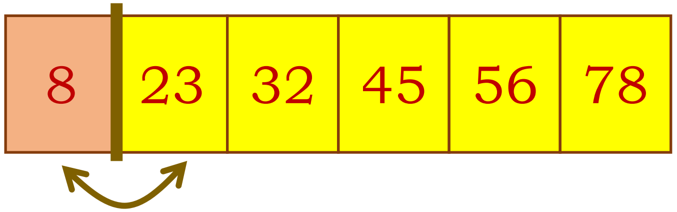
Heap Sort

```
Procedure heap(T[1..n])
        makeheap(T)
        for i = n to 2 step –1do
           swap T[1] and T[i]
           siftdown(T[1 .. I – 1], 1)
```

Note: Assuming we are making a maximum heap.

Reach end of loop.



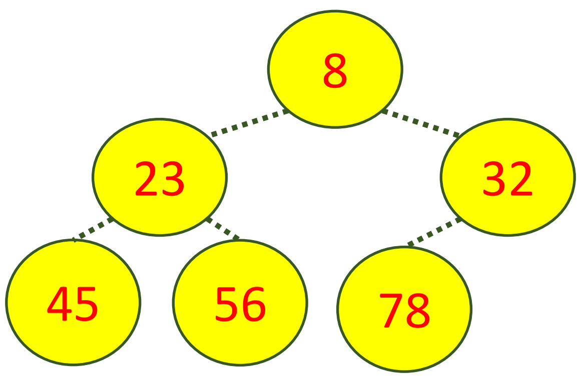


Heap Sort

```
Procedure heap(T[1..n])
        makeheap(T)
        for i = n to 2 step –1do
            swap T[1] and T[i]
            siftdown(T[1 .. I – 1], 1)
```

Note: Assuming we are making a maximum heap.

8	23	32	45	56	78



The list is now sorted.

Heap Sort

Heapsort

procedure heap($T[1..n]$)

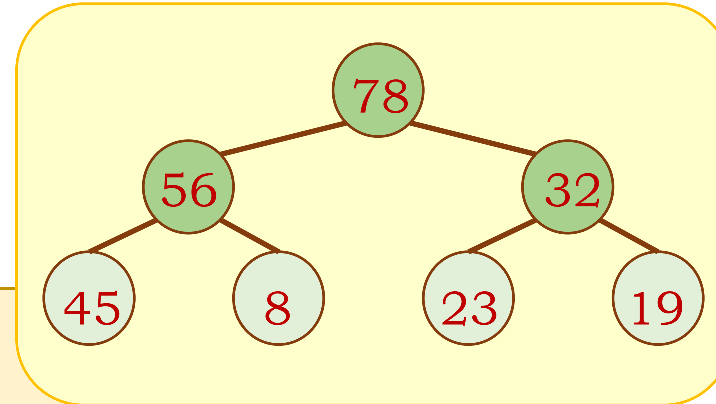
    makeheap(T)

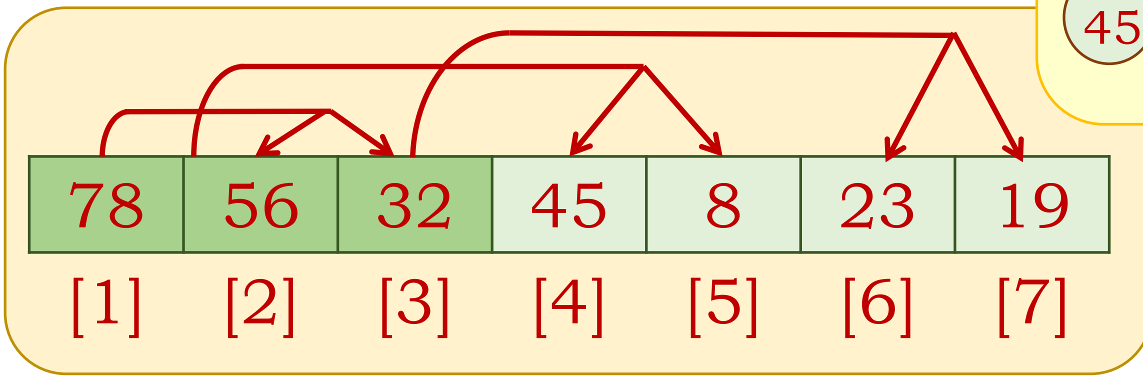
    for $i = n$ to 2 step -1 do

        swap $T[1]$ and $T[i]$

        siftdown($T[1..i-1]$, 1)

Heap Sort





All $T[i], i = 1$ to $n/2$, are internal nodes, i.e. root of some sub-tree.

Idea:

- Start from the sub-tree whose root is $T[n/2]$
- Rearrange the sub-tree so that it is a heap
- Repeat with all the sub-trees from $T[n/2]$ to $T[1]$

Heap Sort

Makeheap

```
Procedure makeheap(T[1..n])
 for i = n ÷ 2 to 1 step –1 do
    siftdown(T, i)

Procedure siftdown(T[1 .. n], i)
 k = i
 repeat
    j = k
    if 2j <= n and T[2j] > T[k] then k = 2j
    if 2j + 1 <= n and T[2j + 1] > T[k] then k = 2j + 1
    swap T[j] and T[k]
 until j = k
```

Heap Sort

Heapsort

procedure heap($T[1..n]$)

makeheap(T)

for $i = n$ to 2 step -1 do

swap $T[1]$ and $T[i]$

siftdown($T[1..i-1]$, 1)

Makeheap

T = [7, 2, 9, 5, 1, 3, 8, 4] Unsorted list

T = [**7, 2, 9, 5**, 1, 3, 8, 4]

Siftdown **5**:

[**7, 2, 9, *5***, 1, 3, 8, ***4***] siftdown 5 - 0 swaps

Siftdown **9**:

[**7, 2, *9***, 5, 1, ***3***, ***8***, 4] siftdown 9 - 0 swaps

Siftdown **2**:

[**7**, ***2***, *9*, ***5***, ***1***, 3, 8, 4] siftdown 2

[**7**, ***5***, *9*, ***2***, ***1***, 3, 8, ***4***] siftdown 2 – 1st swap

[**7**, ***5***, *9*, ***4***, ***1***, 3, 8, ***2***] siftdown 2 – 2nd swap

Heap Sort

Heapsort

procedure heap($T[1..n]$)

makeheap(T)

for $i = n$ to 2 step -1 do

swap $T[1]$ and $T[i]$

siftdown($T[1..i-1]$, 1)

Makeheap

T = [**7**, 5, 9, 4, 1, 3, 8, 2] (...continune)

Siftdown **7**:

[***7***, ***5***, ***9***, 4, 1, 3, 8, 2] siftdown 7

[***9***, ***5***, <u>***7***</u>, 4, 1, ***3***, ***8***, 2] siftdown 7 – 1st swap

[***9***, ***5***, ***8***, 4, 1, ***3***, <u>***7***</u>, 2] siftdown 7 – 2nd swap

[9, 5, 8, 4, 1, 3, 7, 2] heap complete

Heap Sort

- Heapsort

  T = [7, 2, 9, 5, 1, 3, 8, 4]

Unsorted list

heap

[9, 5, 8, 4, 1, 3, 7, 2] after makeheap

Swap 9 and 2: [9, 5, 8, 4, 1, 3, 7, 2]

[2, 5, 8, 4, 1, 3, 7, **9**]

Siftdown 2: [***2***, 5, 8, 4, 1, 3, 7, **9**]

[8, 5, ***2***, 4, 1, 3, 7, **9**] - 1st swap

[8, 5, ***7***, 4, 1, 3, ***2***, **9**] - 2nd swap

Heapsort

procedure heap($T[1..n]$)

makeheap(T)

for $i = n$ to 2 step -1 do

swap $T[1]$ and $T[i]$

siftdown($T[1..i-1]$, 1)

Heap Sort

Heapsort

procedure heap($T[1..n]$)

makeheap(T)

for $i = n$ to 2 step -1 do

swap $T[1]$ and $T[i]$

siftdown($T[1..i-1]$, 1)

(...continue)

Swap 8 and 2: [8, 5, 7, 4, 1, 3, 2, **9**]

[2, 5, 7, 4, 1, 3, **8, 9**]

Siftdown 2: [2, 5, 7, 4, 1, 3, **8, 9**]

[7, 5, ***2***, 4, 1, 3, **8, 9**] - 1st swap

[7, 5, 3, 4, 1, ***2***, **8, 9**] - 2nd swap

Swap 7 and 2: [7, 5, 3, 4, 1, ***2***, **8, 9**]

[2, 5, 3, 4, 1, **7, 8, 9**]

Siftdown 2: [2, 5, 3, 4, 1, **7, 8, 9**]

[5, ***2***, 3, 4, 1, **7, 8, 9**] - 1st swap

[5, 4, 3, ***2***, 1, **7, 8, 9**] - 2nd swap

Heap Sort

Heapsort

procedure heap($T[1..n]$)

makeheap(T)

for $i = n$ to 2 step -1 do

swap $T[1]$ and $T[i]$

siftdown($T[1..i-1]$, 1)

(...continue)

Swap 5 and 1: [5, 4, 3, 2, 1, **7, 8, 9**]

[1, 4, 3, 2, **5, 7, 8, 9**]

Siftdown 1: [1, 4, 3, 2, **5, 7, 8, 9**]

[4, ***1***, 3, 2, **5, 7, 8, 9**] - 1st swap

[4, 2, 3, ***1***, **5, 7, 8, 9**] - 2nd swap

Swap 4 and 1: [4, 2, 3, ***1***, **5, 7, 8, 9**]

[1, 2, 3, **4, 5, 7, 8, 9**]

Siftdown 1: [1, 2, 3, **4, 5, 7, 8, 9**]

[3, 2, ***1***, **4, 5, 7, 8, 9**] - 1st swap

Heap Sort

Heapsort

procedure heap($T[1..n]$)

makeheap(T)

for $i = n$ to 2 step -1 do

swap $T[1]$ and $T[i]$

siftdown($T[1..i-1]$, 1)

(...continue)

Swap 5 and 1: [3, 2, ***1***, **4, 5, 7, 8, 9**]

[1, 2, **3, 4, 5, 7, 8, 9**]

Siftdown 1: [1, 2, **3, 4, 5, 7, 8, 9**]

[2, ***1***, **3, 4, 5, 7, 8, 9**] - 1st swap

Swap 2 and 1: [2, ***1***, **3, 4, 5, 7, 8, 9**]

[1, **2, 3, 4, 5, 7, 8, 9**]

[**1, 2, 3, 4, 5, 7, 8, 9**] - Sorted list

Heap Sort

Heapsort's efficiency

```
Procedure heap(T[1..n])
    makeheap(T)
    for i = n to 2 step –1do
        swap T[1] and T[i]
        siftdown(T[1 .. I – 1], 1)
```

Heap Sort

Heapsort's efficiency

Procedure heap(T[1..n])

makeheap(T)

for i = n to 2 step –1do

swap T[1] and T[i]

siftdown(T[1 .. I – 1], 1)

This process calls (½*n*) time of siftdown. Siftdown has requires log_2n times. Thus makeheap needs $nlog_2n$ times.

Heap Sort

Heapsort's efficiency

Procedure heap(T[1..n])

makeheap(T)

This process calls (½*n*) time of siftdown. Siftdown has requires $log_2 n$ times. Thus makeheap needs $nlog_2 n$ times.

```
for i = n to 2 step –1do
    swap T[1] and T[i]
    siftdown(T[1 .. I – 1], 1)
```

This loop starts at the end of the array and moves through the heap one element at a time until it reaches the second element. Thus approximately ***n*** times.

Heap Sort

Heapsort's efficiency

Procedure heap(T[1..n])

makeheap(T)

for i = n to 2 step –1do

swap T[1] and T[i]

siftdown(T[1 .. I – 1], 1)

This process calls (½*n*) time of siftdown. Siftdown has requires log_2n times. Thus makeheap needs $nlog_2n$ times.

This loop starts at the end of the array and moves through the heap one element at a time until it reaches the second element. Thus approximately ***n*** times.

Complexity for Swap is constant.

Heap Sort

Heapsort's efficiency

Procedure heap(T[1..n])

makeheap(T)

for i = n to 2 step –1do

swap T[1] and T[i]

siftdown(T[1 .. I – 1], 1)

This process calls (½*n*) time of siftdown. Siftdown has requires log_2n times. Thus makeheap needs $nlog_2n$ times.

This loop starts at the end of the array and moves through the heap one element at a time until it reaches the second element. Thus approximately ***n*** times.

Complexity for Swap is constant.

This process contains a recursive loop. The loop follows a branch down a binary tree from the root to a leaf or until the parent and child data are in heap order. This requires $\boldsymbol{log_2n}$.

Heap Sort

Heapsort's efficiency

Procedure heap(T[1..n])

makeheap(T)

for i = n to 2 step –1do

swap T[1] and T[i]

siftdown(T[1 .. I – 1], 1)

This process calls (½*n*) time of siftdown. Siftdown has requires $log_2 n$ times. Thus makeheap needs $nlog_2 n$ times.

This loop starts at the end of the array and moves through the heap one element at a time until it reaches the second element. Thus approximately ***n*** times.

Complexity for Swap is constant.

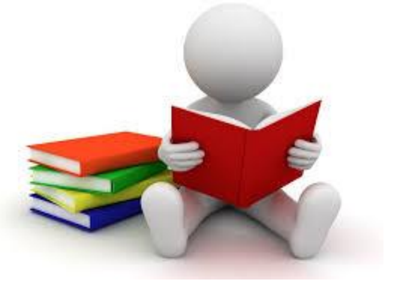
This process contains a recursive loop. The loop follows a branch down a binary tree from the root to a leaf or until the parent and child data are in heap order. This requires $\boldsymbol{log_2 n}$.

$\boldsymbol{n\ log_2 n}$

Heap Sort

Heapsort's efficiency:

- Makeheap is $O(n \lg n)$.
- Siftdown is $O(\lg n)$.
- Heapsort is $O(n \lg n) + (n-1)O(\lg n) = O(n \lg n)$.



Quick Sort

Quick Sort

Method

- Divide-and-conquer
- Pick an element (pivot) from the list
  - Pivot is arbitrarily chosen
  - Normally, the first element is selected
- Partition the list into two halves such that:
  - All the elements in the first half is smaller than the pivot
  - All the elements in the second half is greater than the pivot.

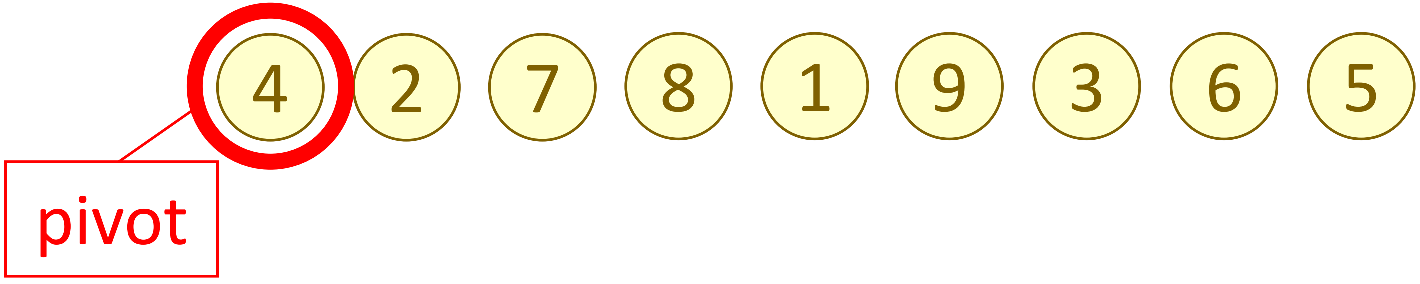
Quick Sort

- After the rearrangement, the pivot element (pivot) occupies a proper position in a sorting of the list.
- Recursively
  - Quick-sort the 1st half
  - Quick-sort the 2nd half

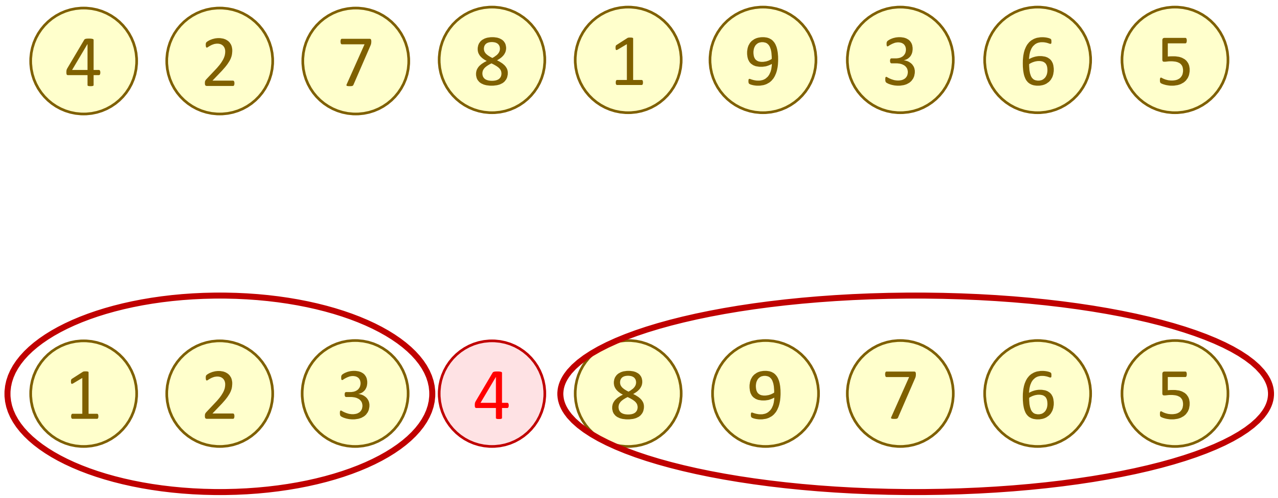
Quick Sort

4 2 7 8 1 9 3 6 5

Quick Sort



Quick Sort



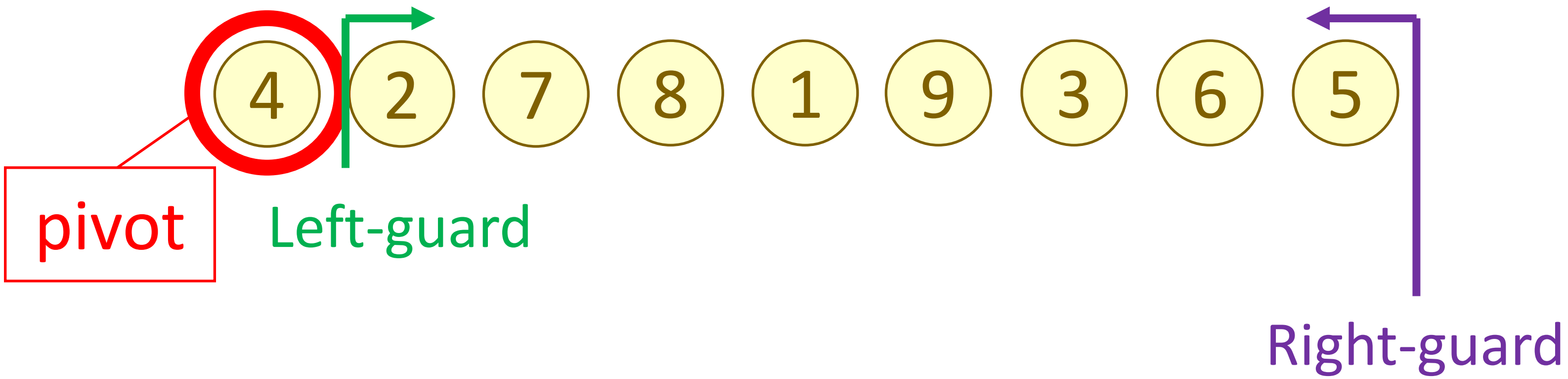
Elements that are smaller than the pivot value are placed on the left-hand-side of the pivot value.

Elements that are greater than or equal to the pivot value are placed on the right-hand-side of the pivot value.

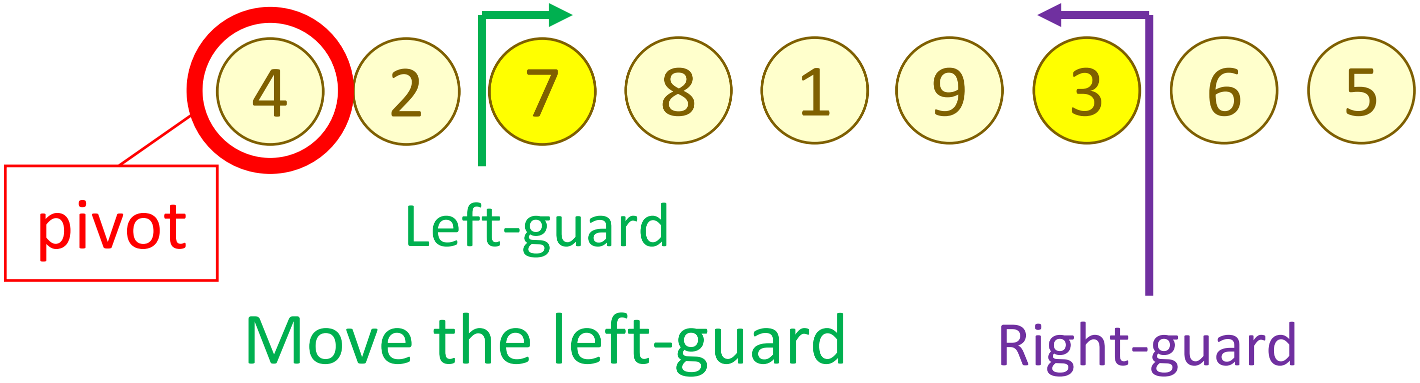
How to divide the list into two sub-lists?

4 2 7 8 1 9 3 6 5

How to divide the list into two sub-lists?



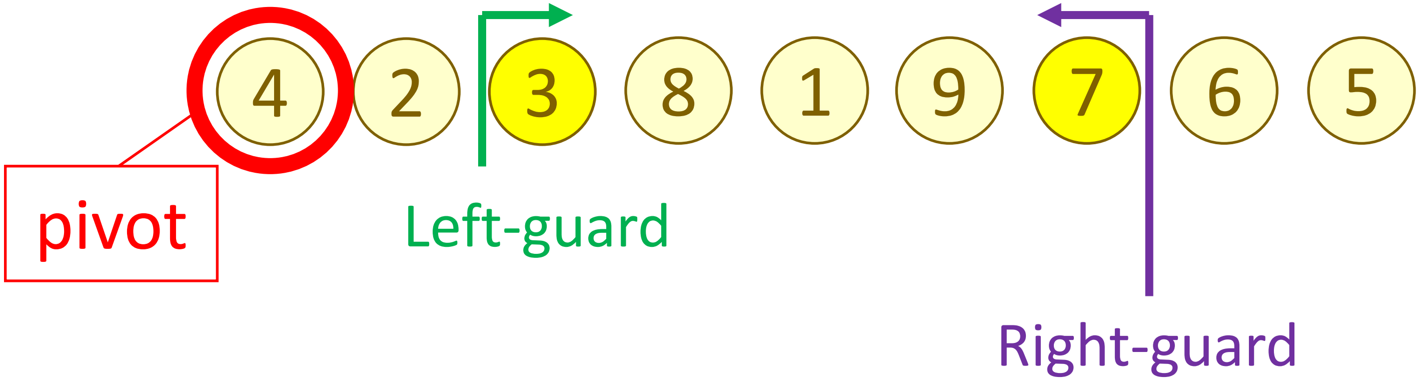
How to divide the list into two sub-lists?



Move the left-guard toward the right until an element that is greater or equal to the pivot value.

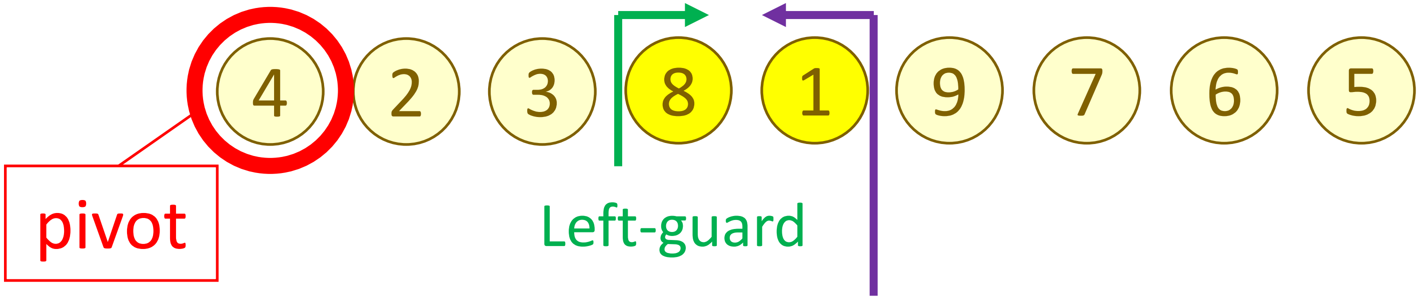
Move the right-guard toward the left until an element that is smaller than pivot value.

How to divide the list into two sub-lists?



If the right-guard is on the right of left-guard, swap the two elements that blocks the left-guard and right-guard.

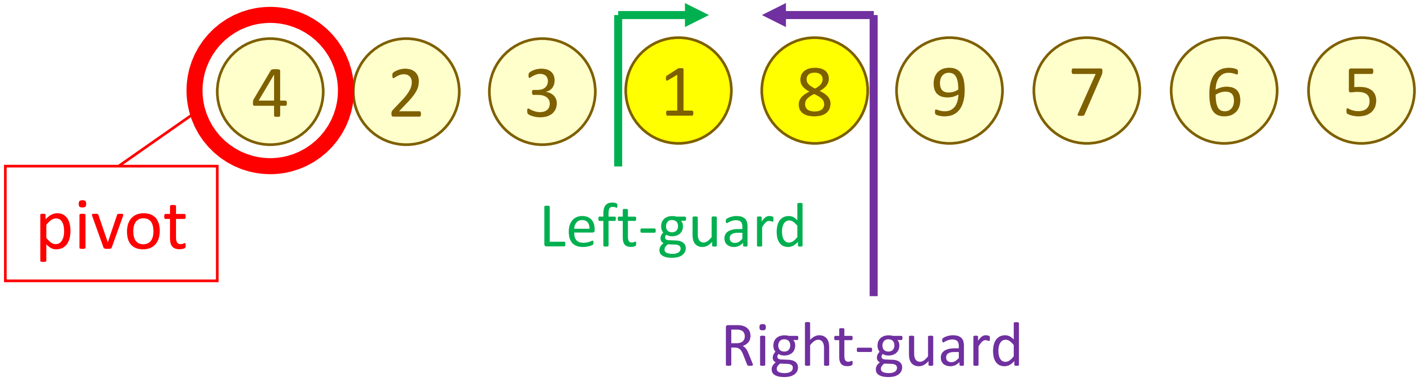
How to divide the list into two sub-lists?



Continue moving the left-guard toward the right until an element that is greater or equal to the pivot value.

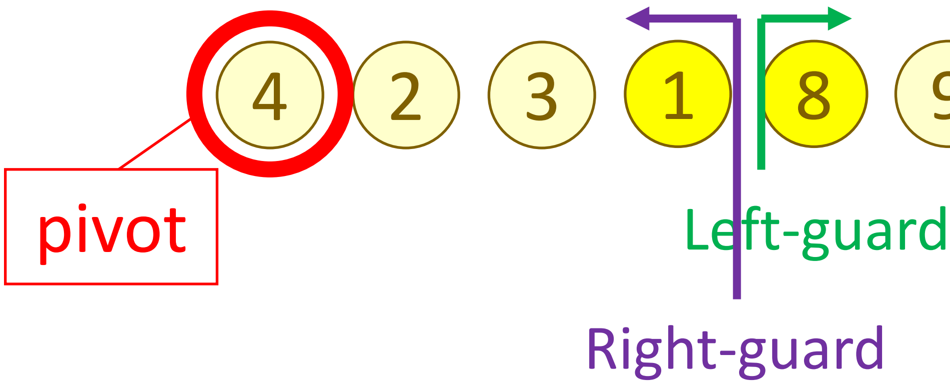
Continue moving the right-guard toward the left until an element that is smaller than pivot value.

How to divide the list into two sub-lists?



If the right-guard is on the right of left-guard, swap the two elements that blocks the left-guard and right-guard.

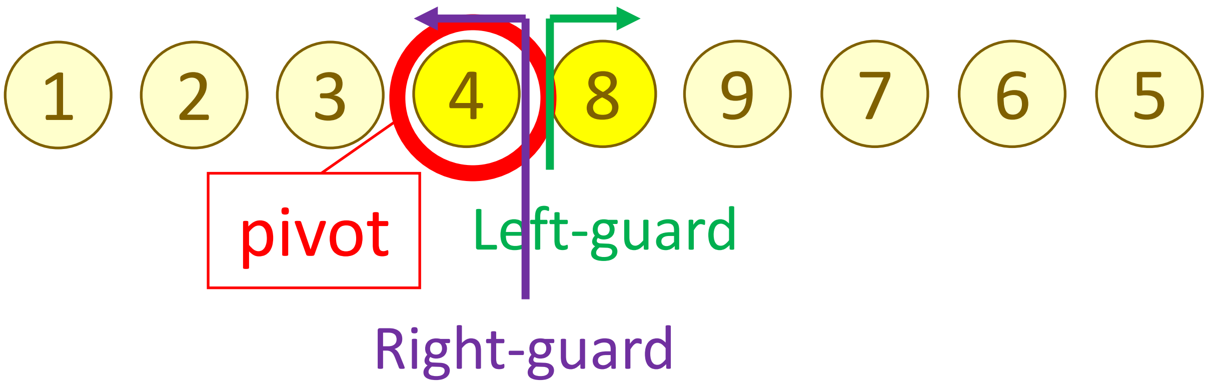
How to divide the list into two sub-lists?



Continue moving the right-guard toward the left until an element that is smaller than pivot value.

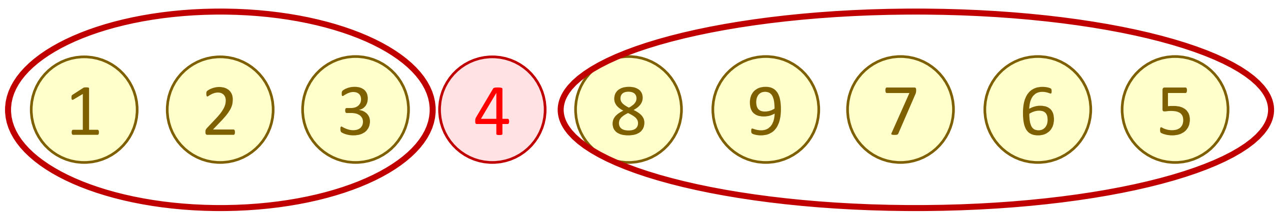
Continue moving the left-guard toward the right until an element that is greater or equal to the pivot value.

How to divide the list into two sub-lists?



If the right-guard is on the right of left-guard, swap the two elements that blocks the left-guard and right-guard, otherwise, swap the element on the left of right-guard (element that blocks the right-guard) with the pivot value.

The list is now split into two sub-lists, the left sub-list consisting of the elements 1, 2, and 3, and the right sub-list consisting of the elements 8, 9, 7, 6, and 5.



Elements that are smaller than the pivot value are placed on the left-hand-side of the pivot value.

Elements that are greater than or equal to the pivot value are placed on the right-hand-side of the pivot value.

How to divide the list into two sub-lists?

1 2 3 4 8 9 7 6 5

After the division, the pivot value is **guaranteed** to be in the position, where the value is supposed to be, when the list is sorted.

Recursive process

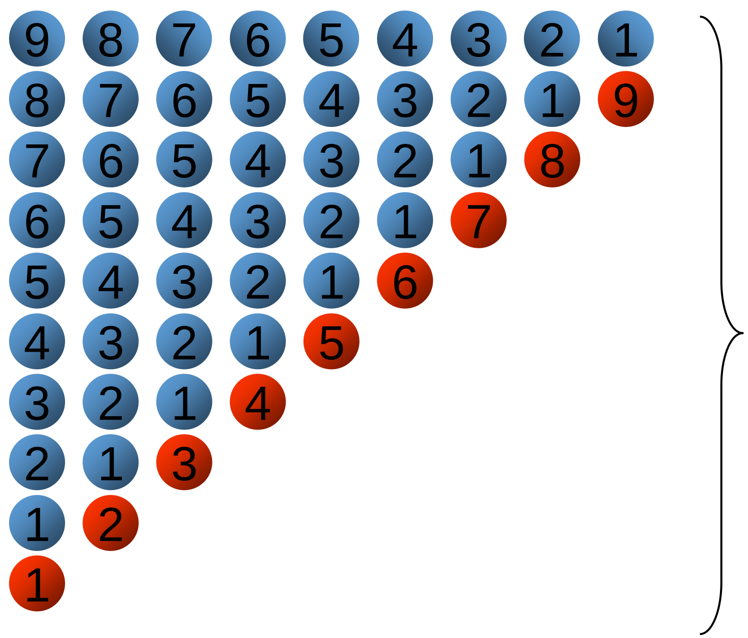
- The algorithm will recursively sort the left sub-list and right sub-list respectively, until all the pivot values are placed on their respective position.

Quick Sort

```
Procedure quicksort(L[Low .. High])
if  High > Low
    pivot(L[Low .. High], Position)
    quicksort(L[Low .. Position - 1])
    quicksort(L[Position + 1 .. High])
endif
```

Critical step. Efficiency of the algorithm depends on the choice of pivot.

A bad case

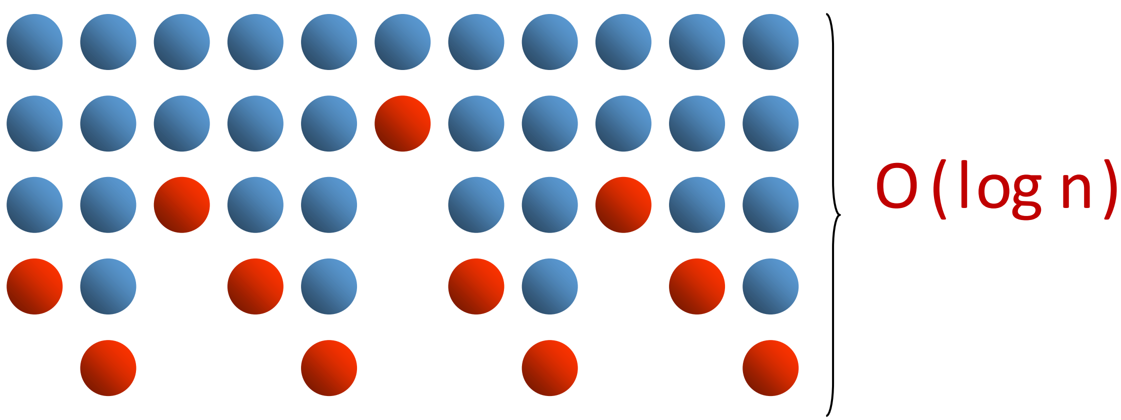


9 8 7 6 5 4 3 2 1
8 7 6 5 4 3 2 1 9
7 6 5 4 3 2 1 8
6 5 4 3 2 1 7
5 4 3 2 1 6
4 3 2 1 5
3 2 1 4
2 1 3
1 2
1

(n)

A good case

- Each x divides X evenly into Y and Z.
- The running time is $O(n \log n)$.

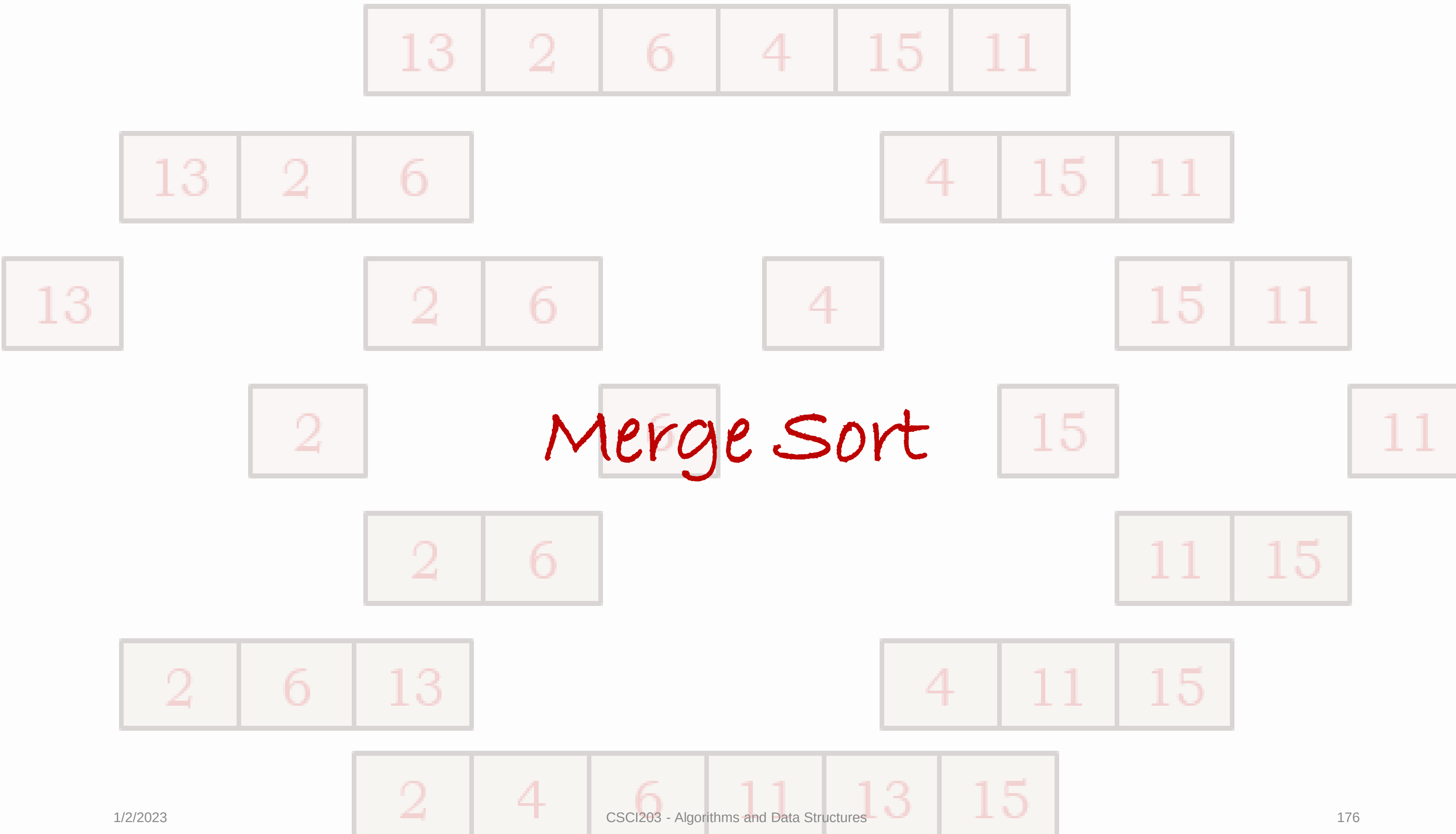


Quick Sort

Quicksort's efficiency:

- On average, each partition halves the size of the array to be sorted.
- On average each partition swaps half the elements.
- Algorithms is in $O(n \lg n)$ in average case.
- Worst case, algorithm is in $O(n^2)$. This scenario occurs when a list is in descending order, and it is to be sorted in ascending order or vice versa.

Merge Sort



Merge Sort

Merge Sort

Merge Sort

- Merge sort is a sorting algorithm based on the divide-and-conquer design strategy.
- The algorithm divides the unsorted list into two nearly equal size sub-lists:
  - Sort each sub-list recursively by applying merge sort
  - **Merge the two sub-lists back into one sorted list**

Merge Sort

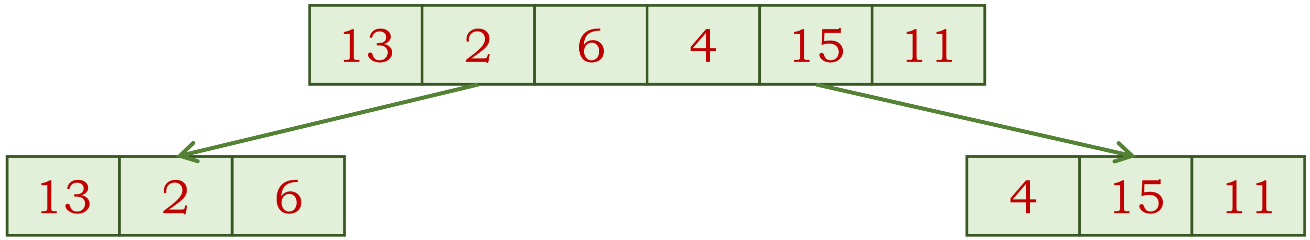
Method

- If the list is of length 0 or 1, then it is already sorted
- Otherwise
  - Divide the unsorted list into two about equal size sub-lists
  - Sort each sub-list recursively by applying merge sort
- **Merge the two sub-lists** back into one sorted list

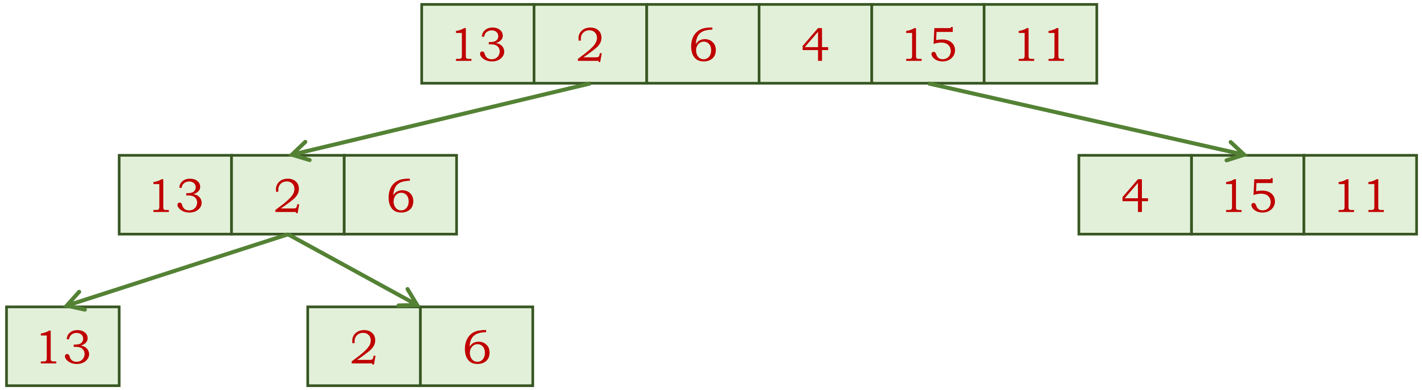
Merge sort

13	2	6	4	15	11

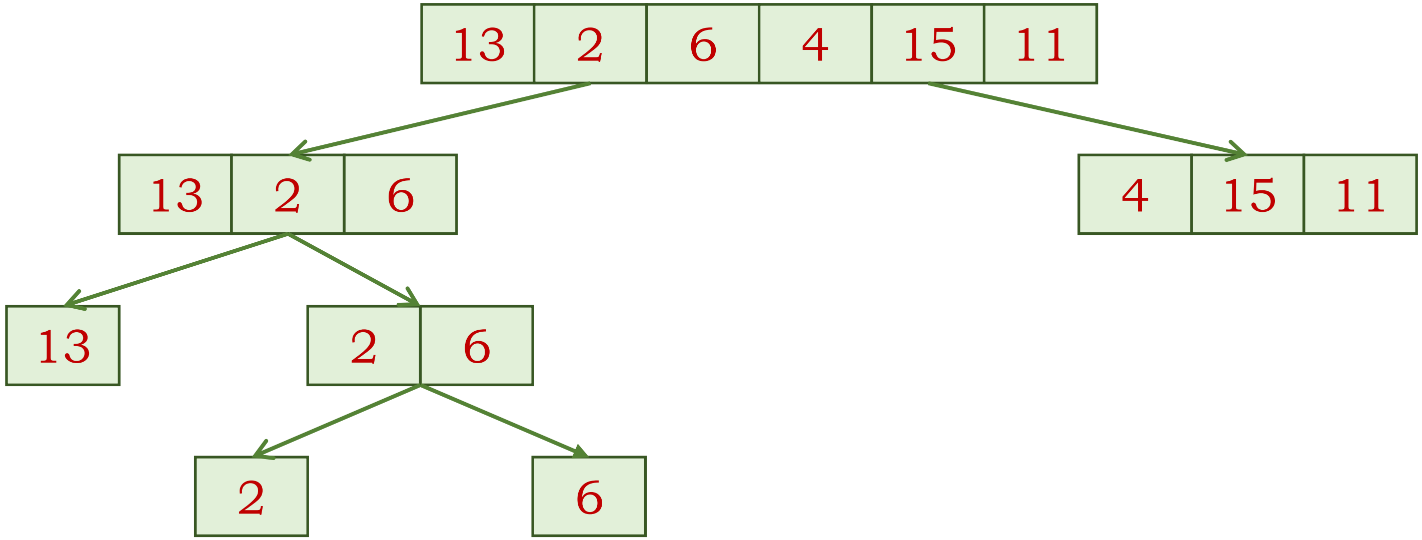
Merge sort



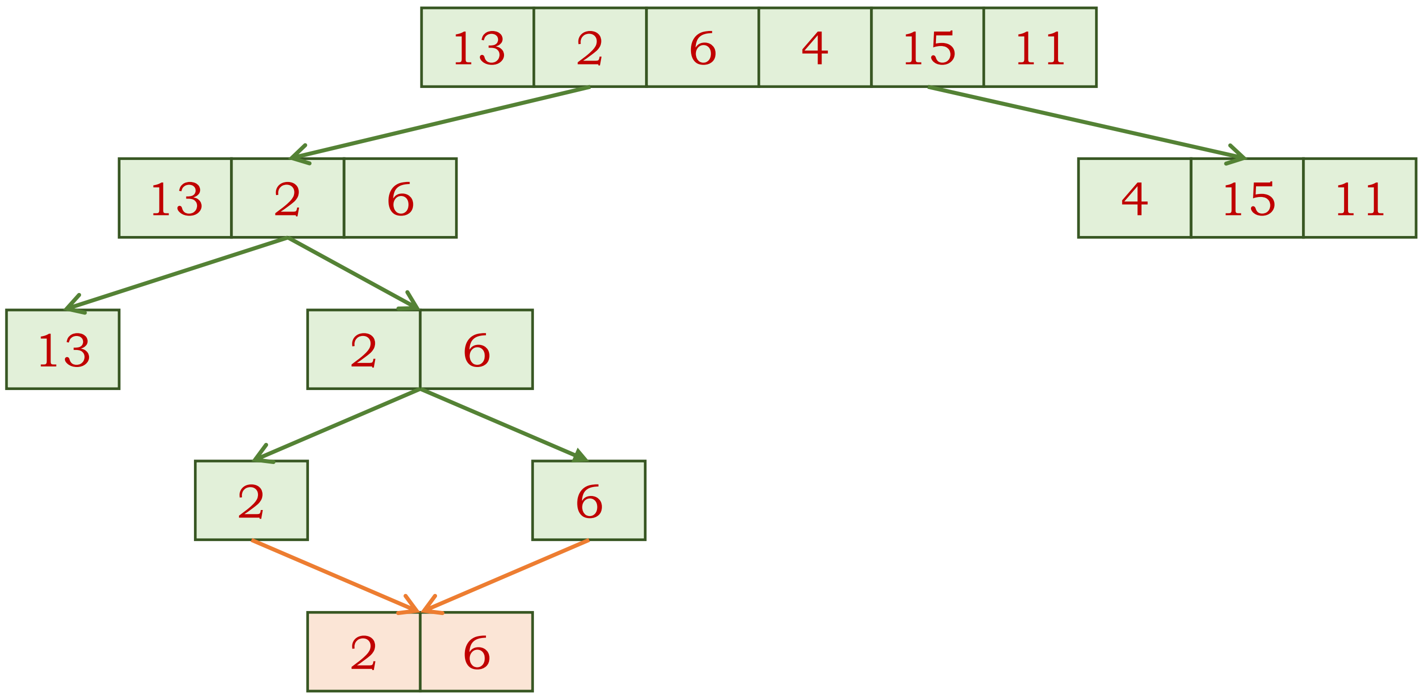
Merge sort



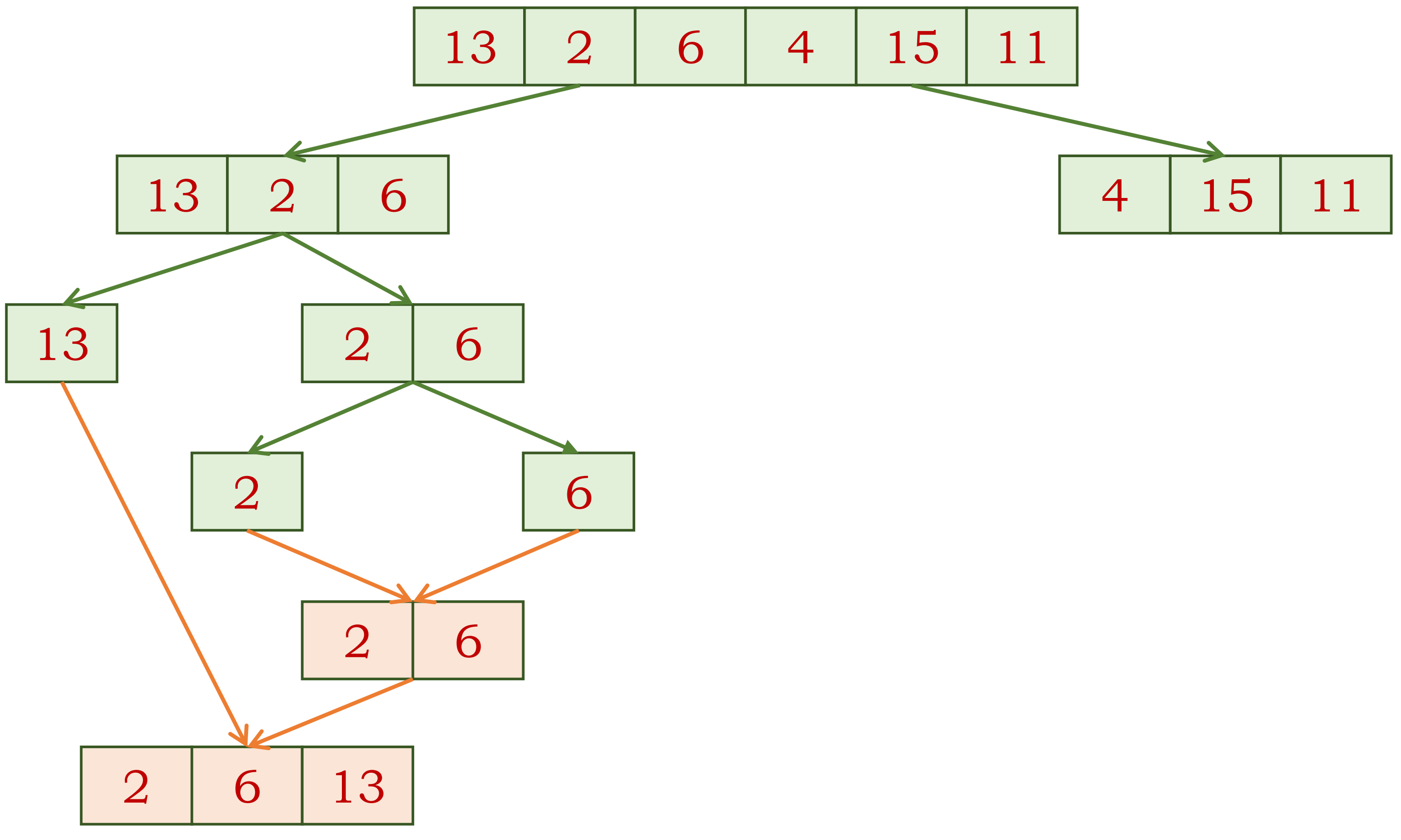
Merge sort



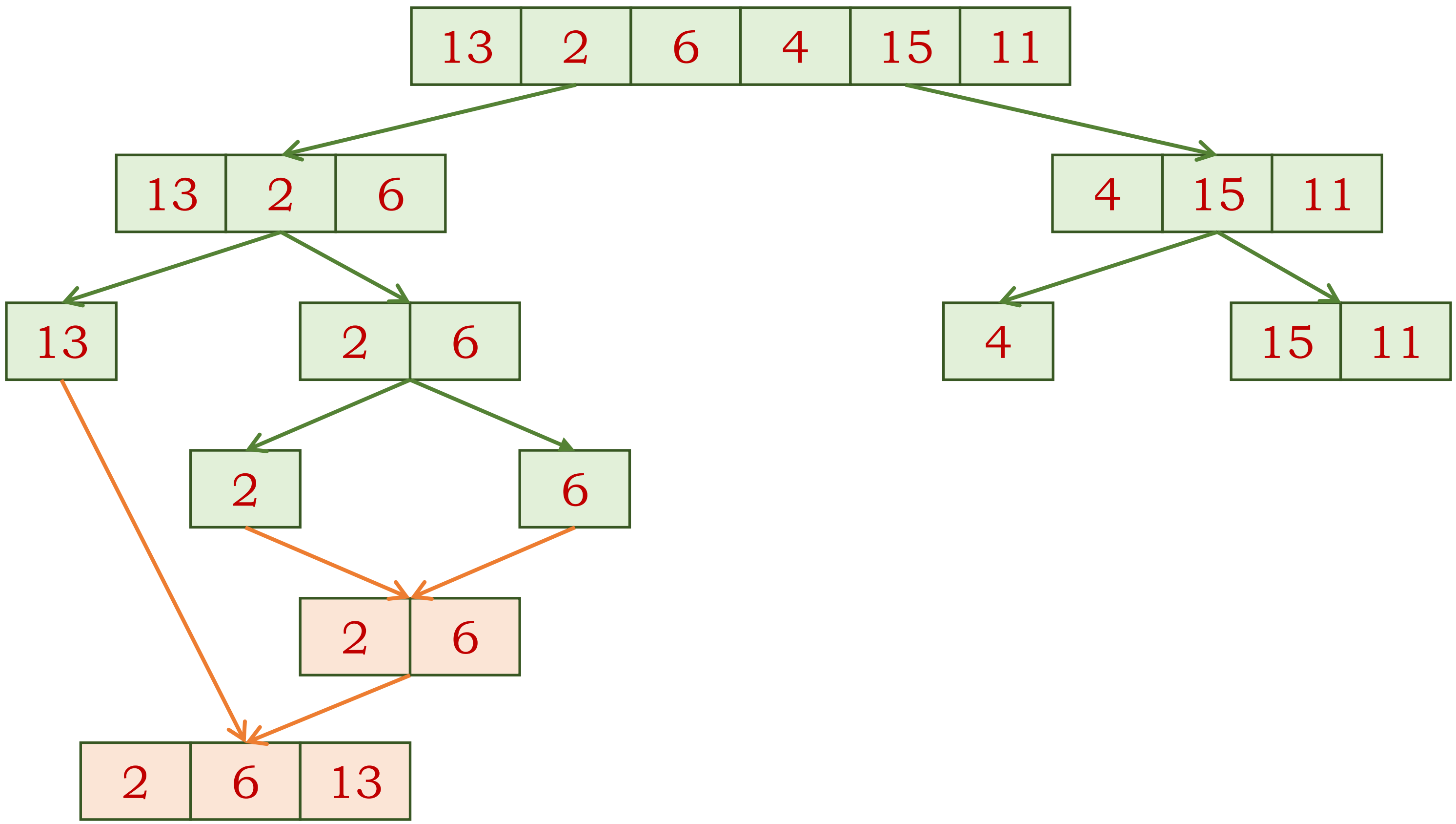
Merge sort



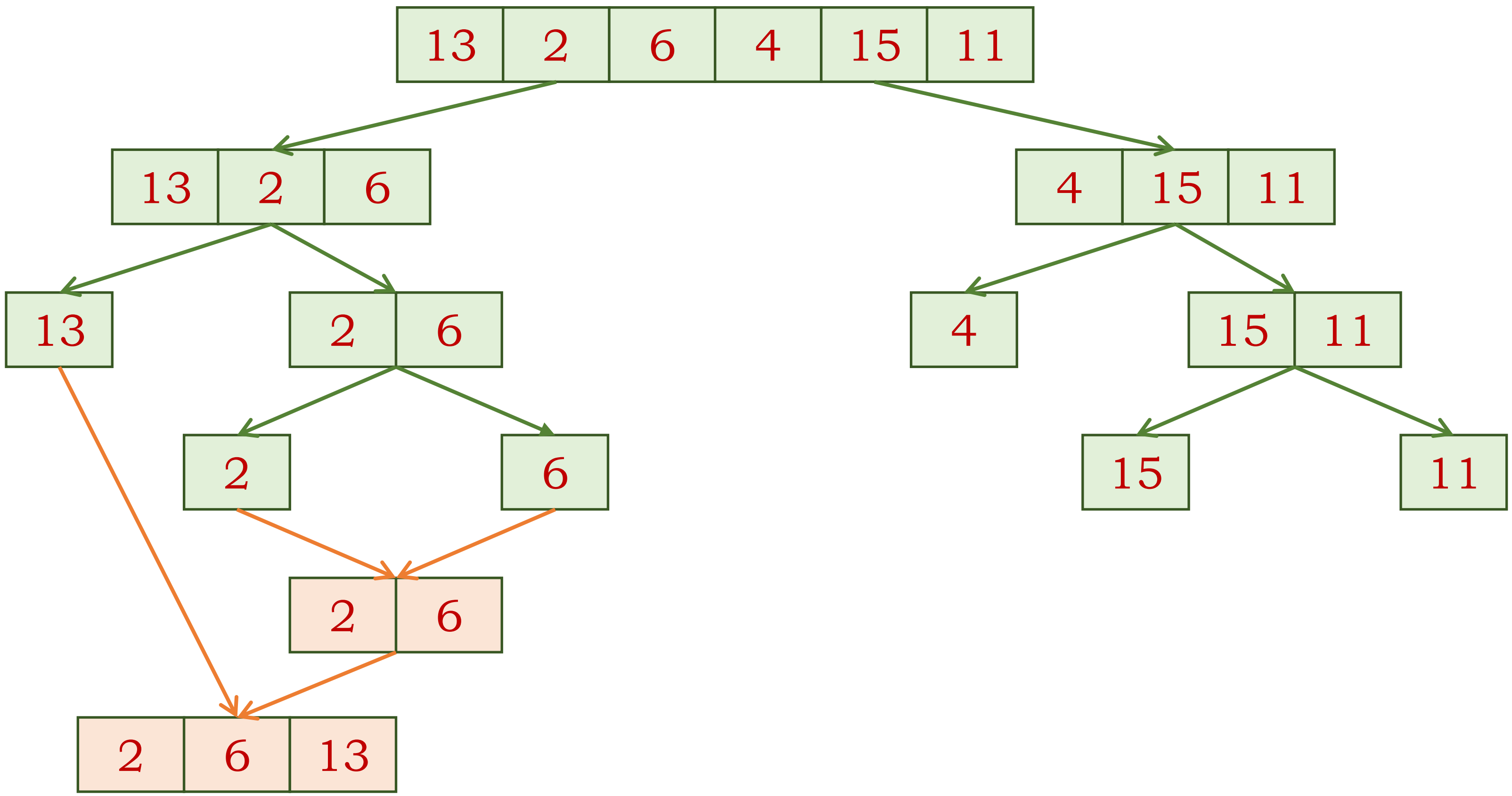
Merge sort



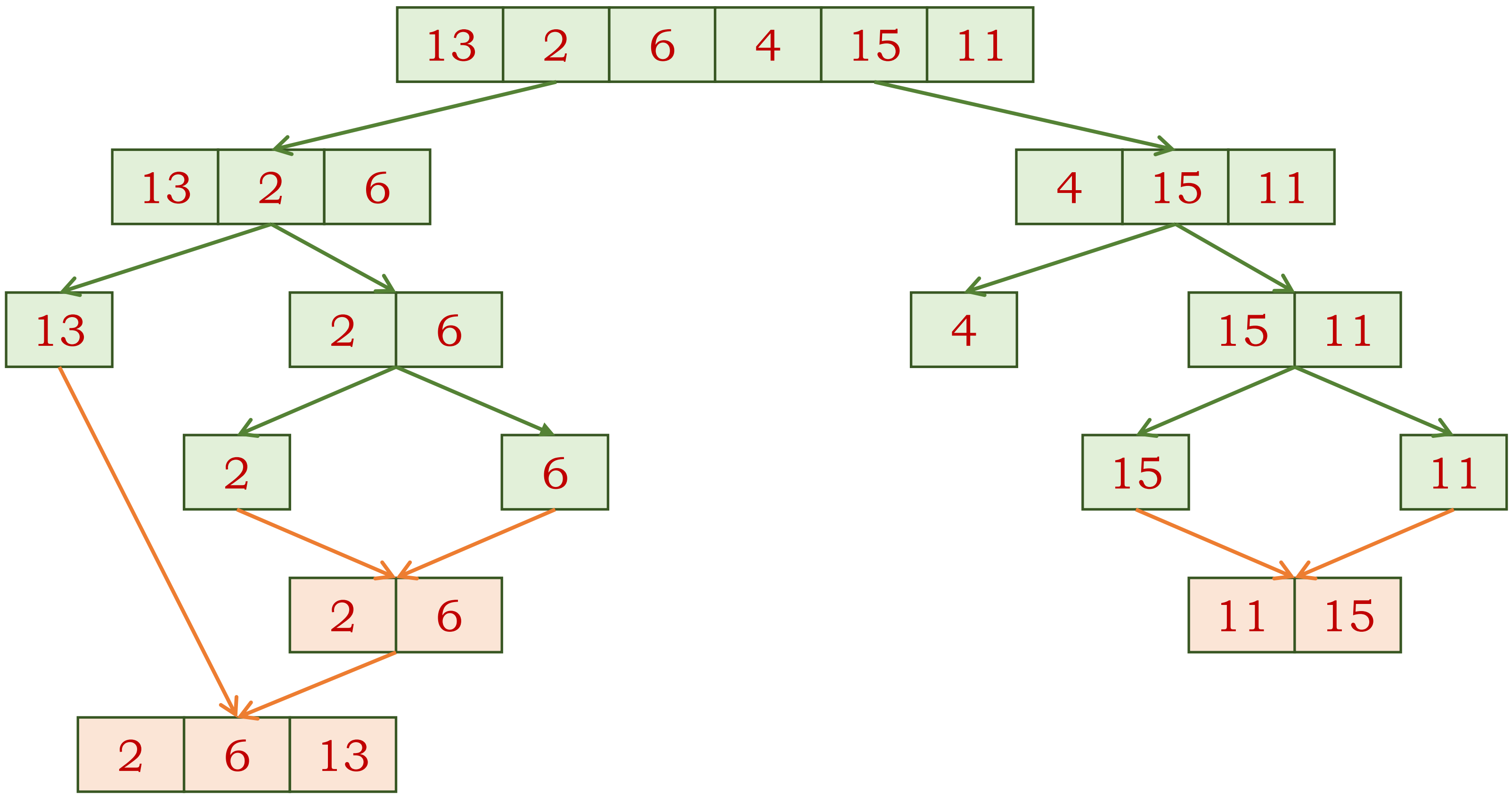
Merge sort



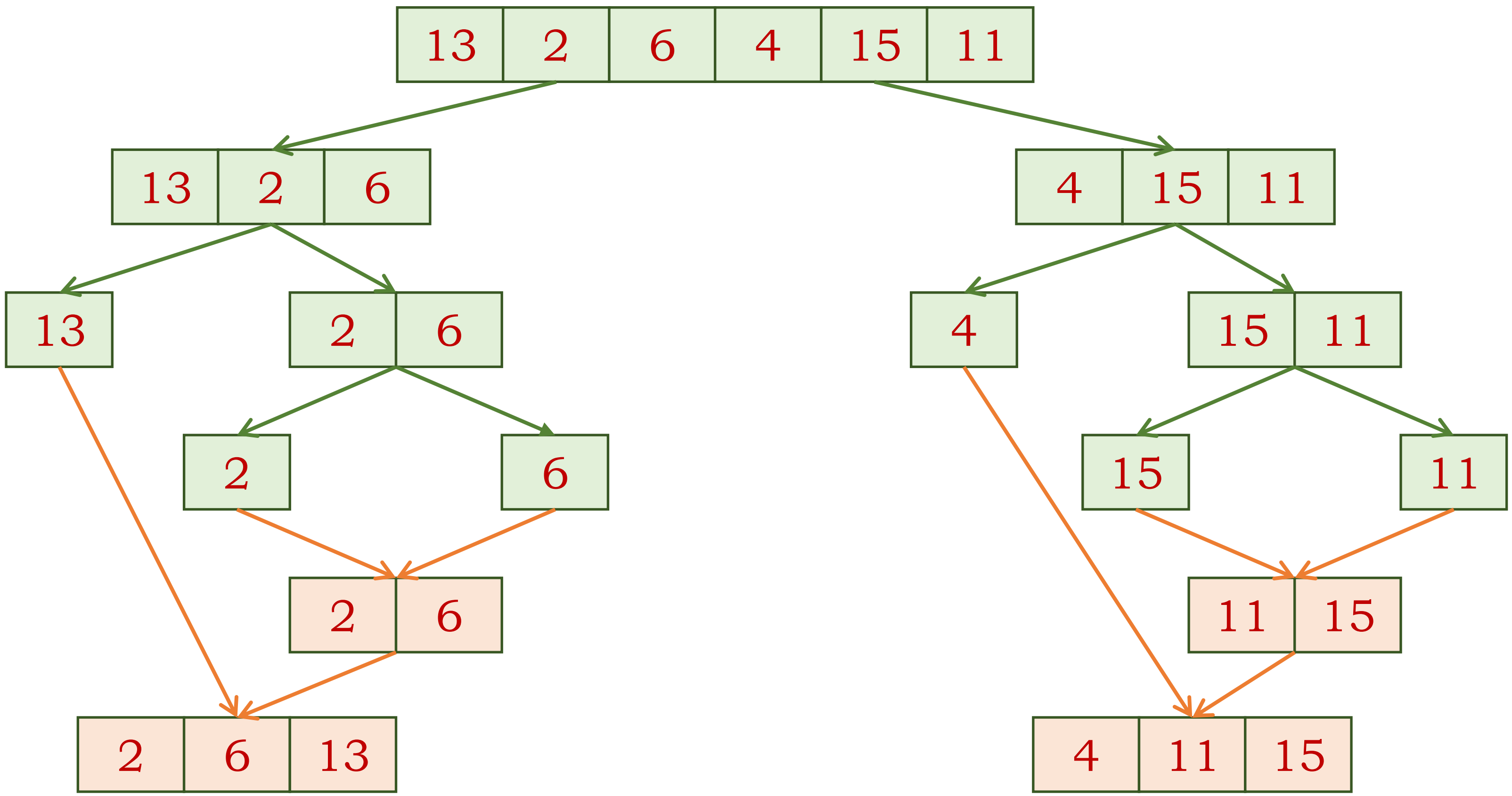
Merge sort



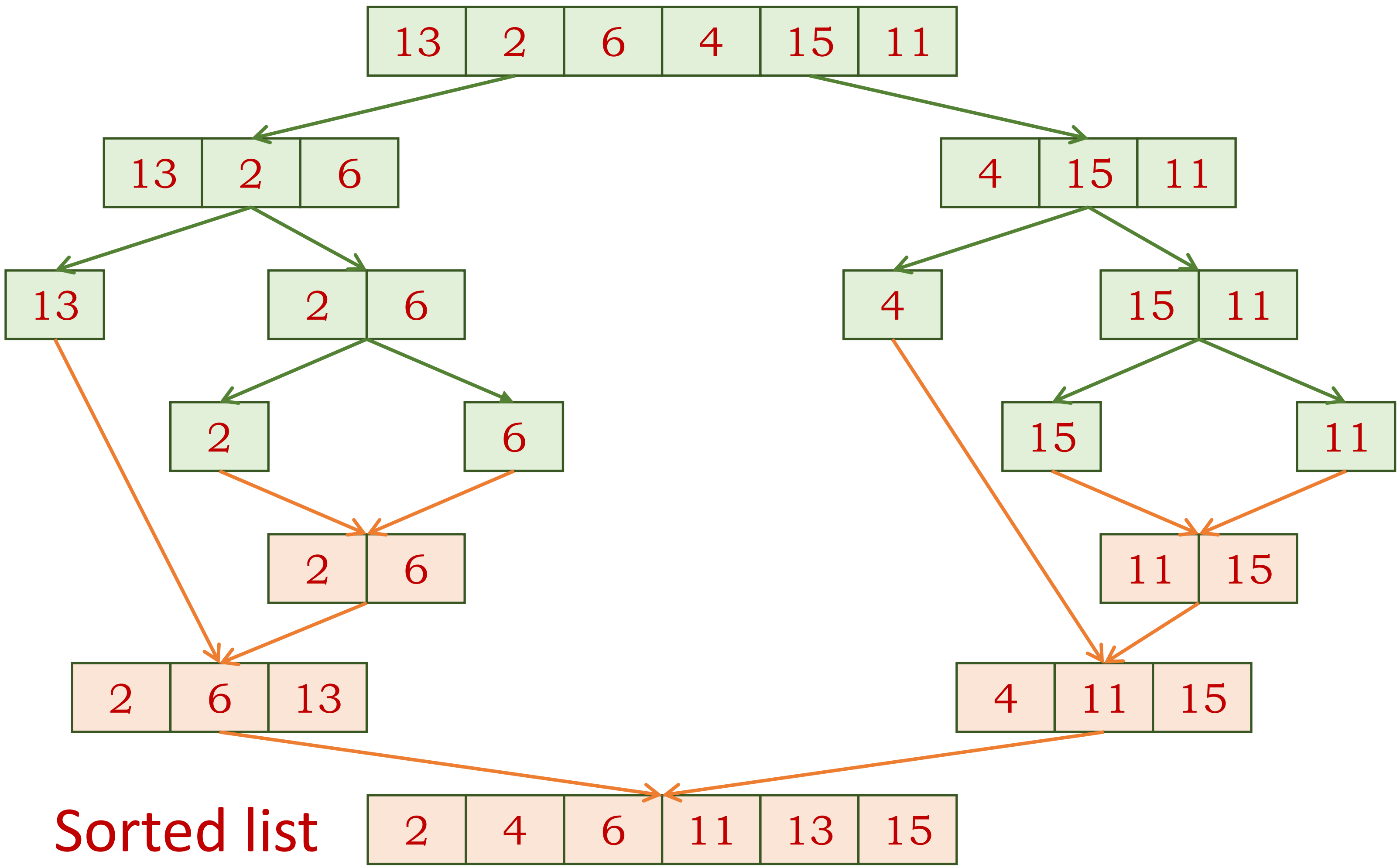
Merge sort



Merge sort



Merge sort



13 2 6 4 15 11
13 2 6
4 15 11
13
2 6
4
15 11
2
6
15
11
2 6
11 15
2 6 13
4 11 15
Sorted list
2 4 6 11 13 15

Merge Sort

```
Mergesort
global X[1..n] // temporary array used in merge
 procedure
procedure mergesort(T[left..right])
  if left < right then
    centre = (left + right) ÷ 2
    mergesort(T[left..centre])
    mergesort(T[centre+1..right])
    merge(T[left..centre],
        T[centre+1..right],T[left..right])
```

Merge Sort

```
Mergesort
global X[1..n] // temporary array used in merge
 procedure
procedure mergesort(T[left..right])
  if left < right then
    centre = (left + right) ÷ 2
    mergesort(T[left..centre])
    mergesort(T[centre+1..right])
    merge(T[left..centre],
        T[centre+1..right],T[left..right])
```

Merge sort the left half

Merge Sort

```
Mergesort
global X[1..n] // temporary array used in merge
 procedure
procedure mergesort(T[left..right])
  if left < right then
    centre = (left + right) ÷ 2
    mergesort(T[left..centre])
    mergesort(T[centre+1..right])
    merge(T[left..centre],
          T[centre+1..right],T[left..right])
```

Merge sort the left half

Merge sort the right half

Merge Sort

```
Mergesort
global X[1..n] // temporary array used in merge
 procedure
procedure mergesort(T[left..right])
  if left < right then
    centre = (left + right) ÷ 2
    mergesort(T[left..centre])
    mergesort(T[centre+1..right])
    merge(T[left..centre],
        T[centre+1..right],T[left..right])
```

Merge sort the left half

Merge sort the right half

Merge the two sorted sub-lists.

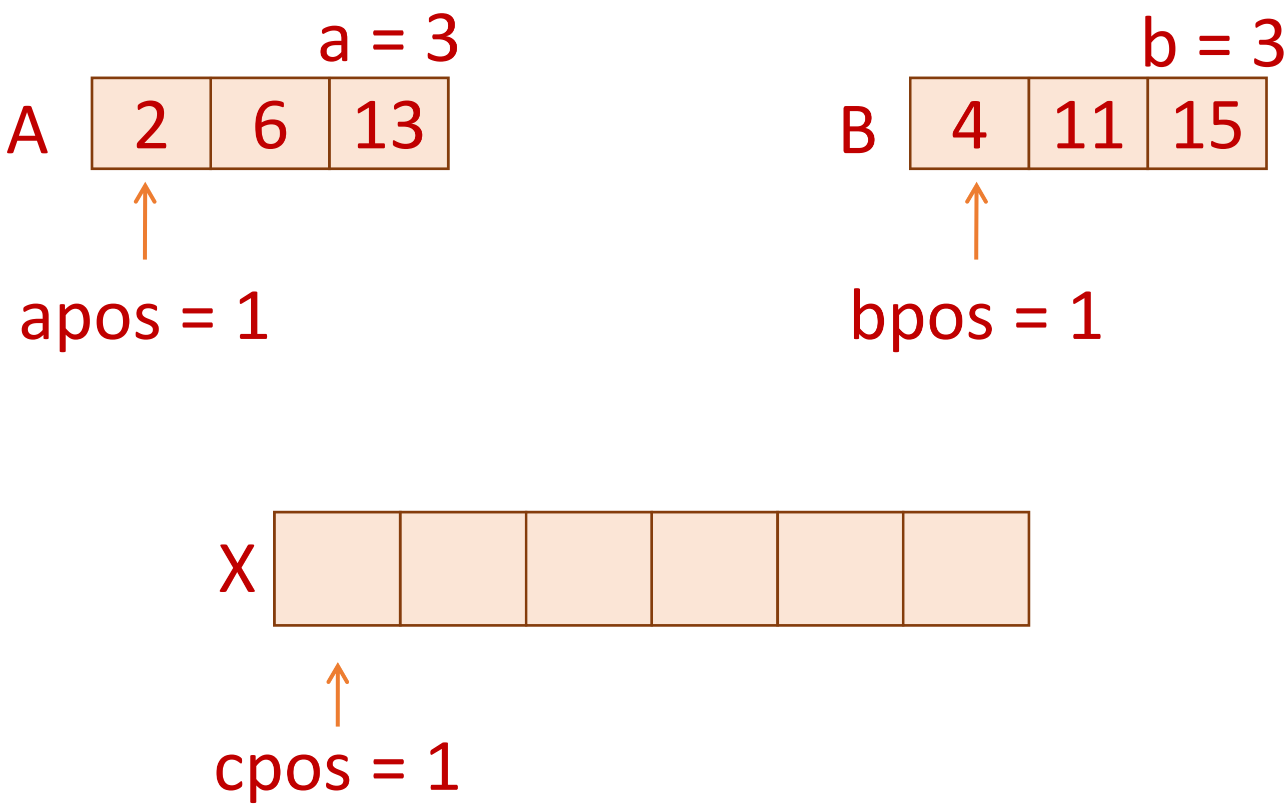
How to merge multiple sub-lists?

```
procedure merge(A[1..a], B[1..b], C[1..a + b])
  apos = 1; bpos = 1; cpos = 1
  while apos ≤ a and bpos ≤ b do
    if A[apos] ≤ B[bpos] then
      X[cpos] = A[apos];
      apos = apos + 1; cpos = cpos + 1
    Else
      X[cpos] = B[bpos];
      bpos = bpos + 1; cpos = cpos + 1
```

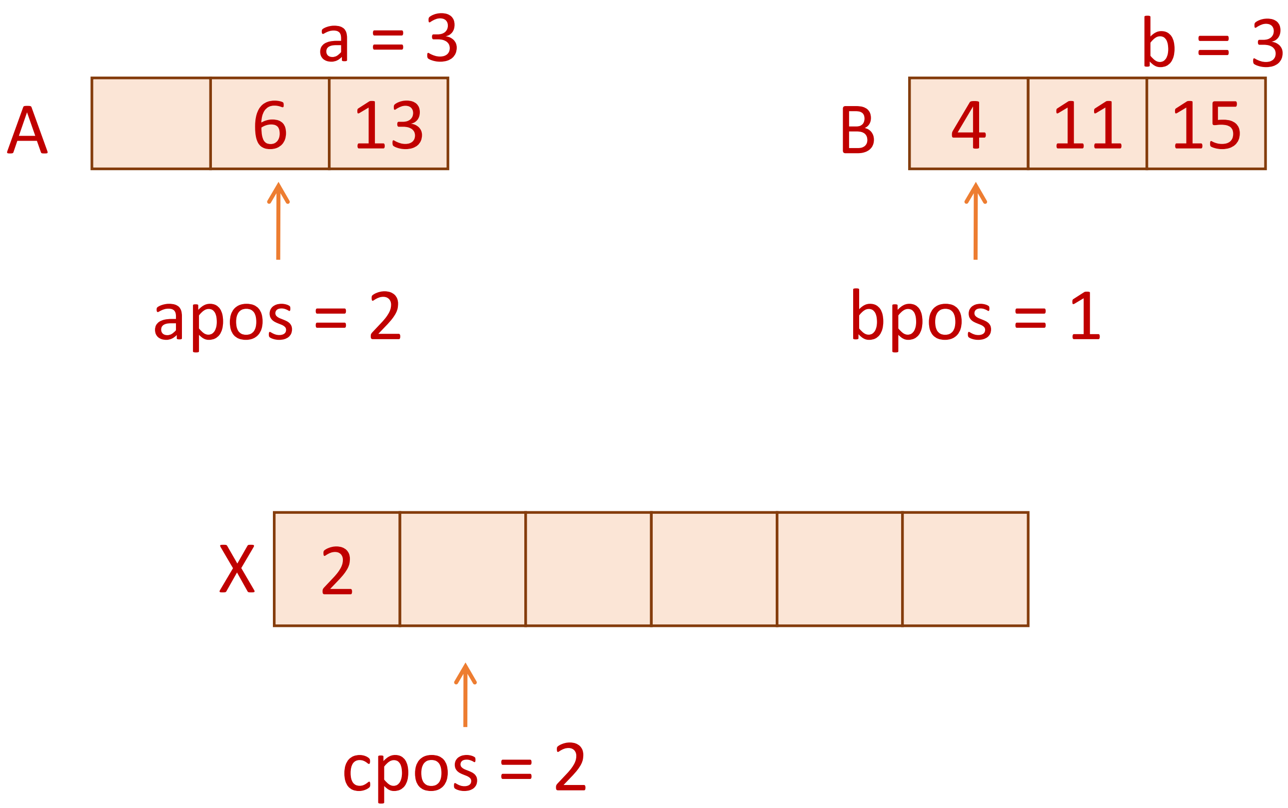
Merge Sort

```
while apos ≤ a do
    X[cpos] = A[apos];
    apos = apos + 1; cpos = cpos + 1;
while bpos ≤ b do
    X[cpos] = B[bpos];
    bpos = bpos + 1; cpos = cpos + 1;
for cpos = 1 to a + b do
        C[cpos] = X[cpos]
```

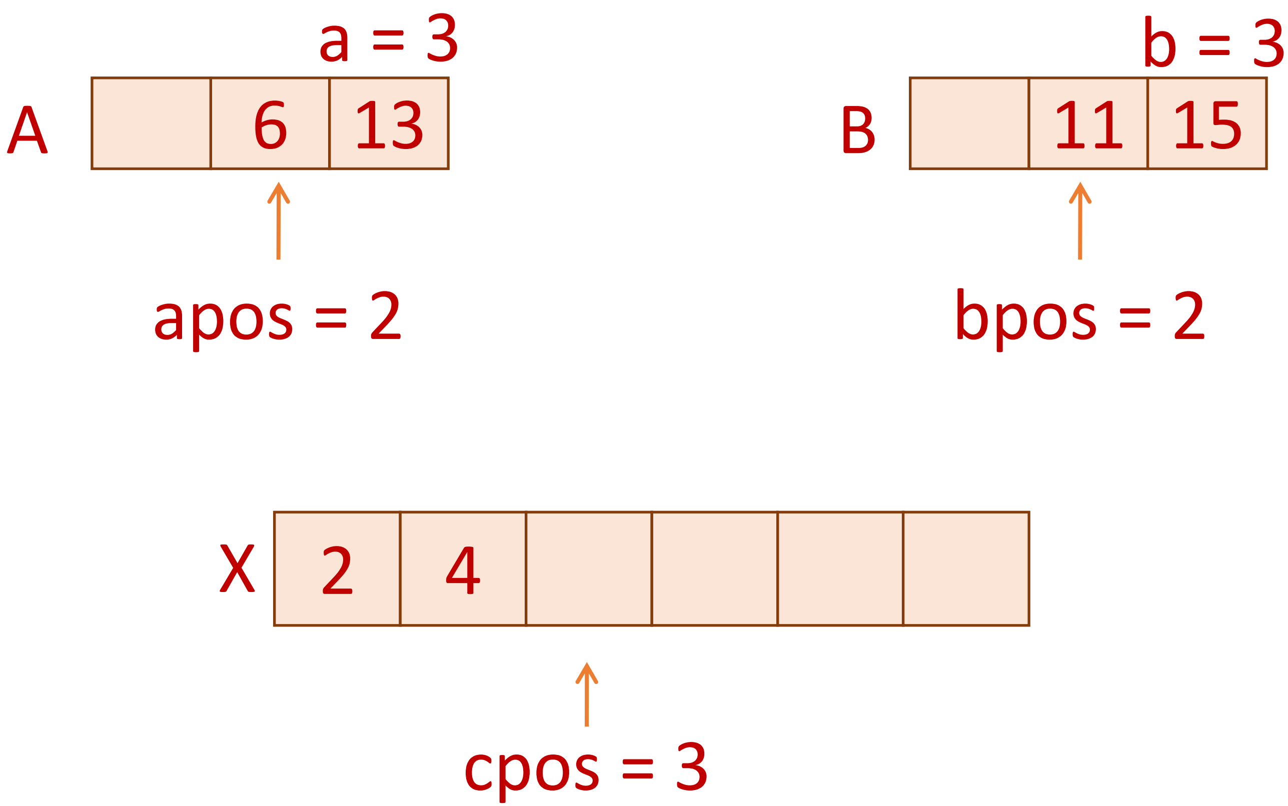
Merge Sort



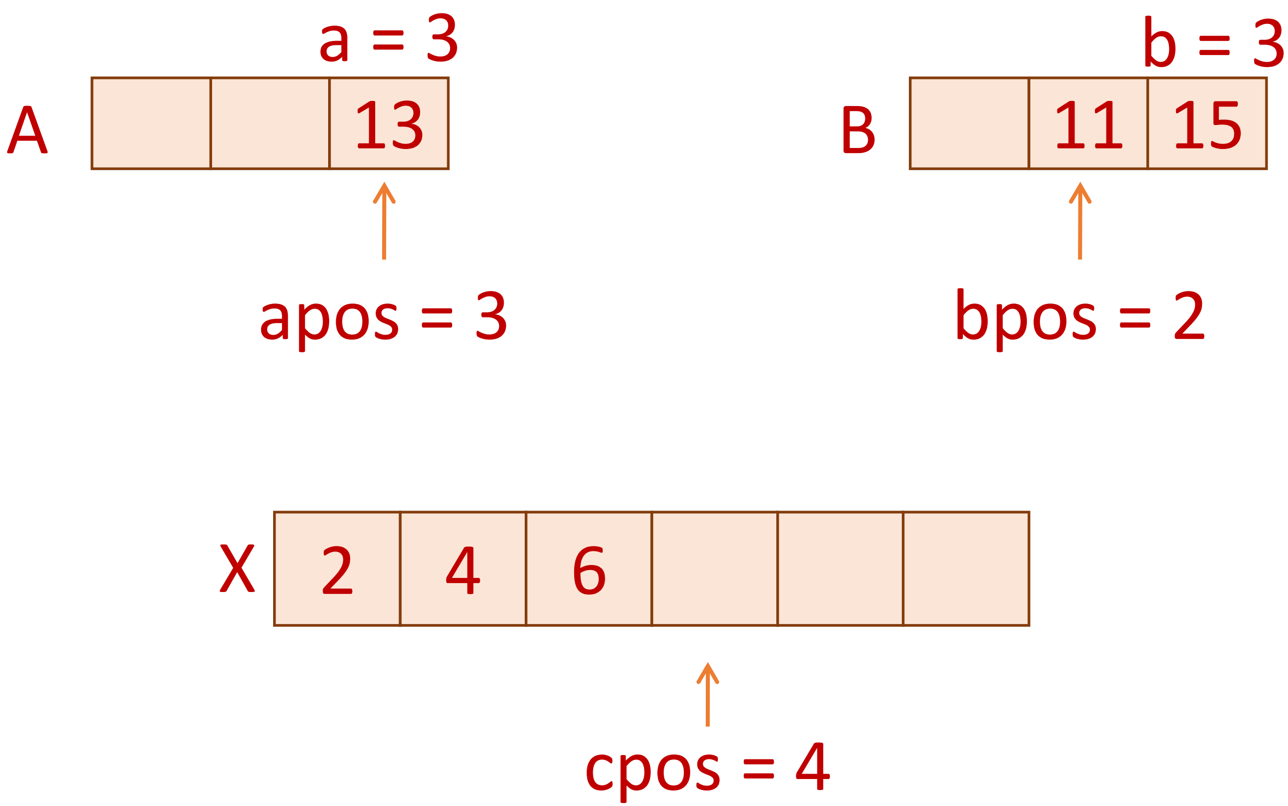
Merge Sort



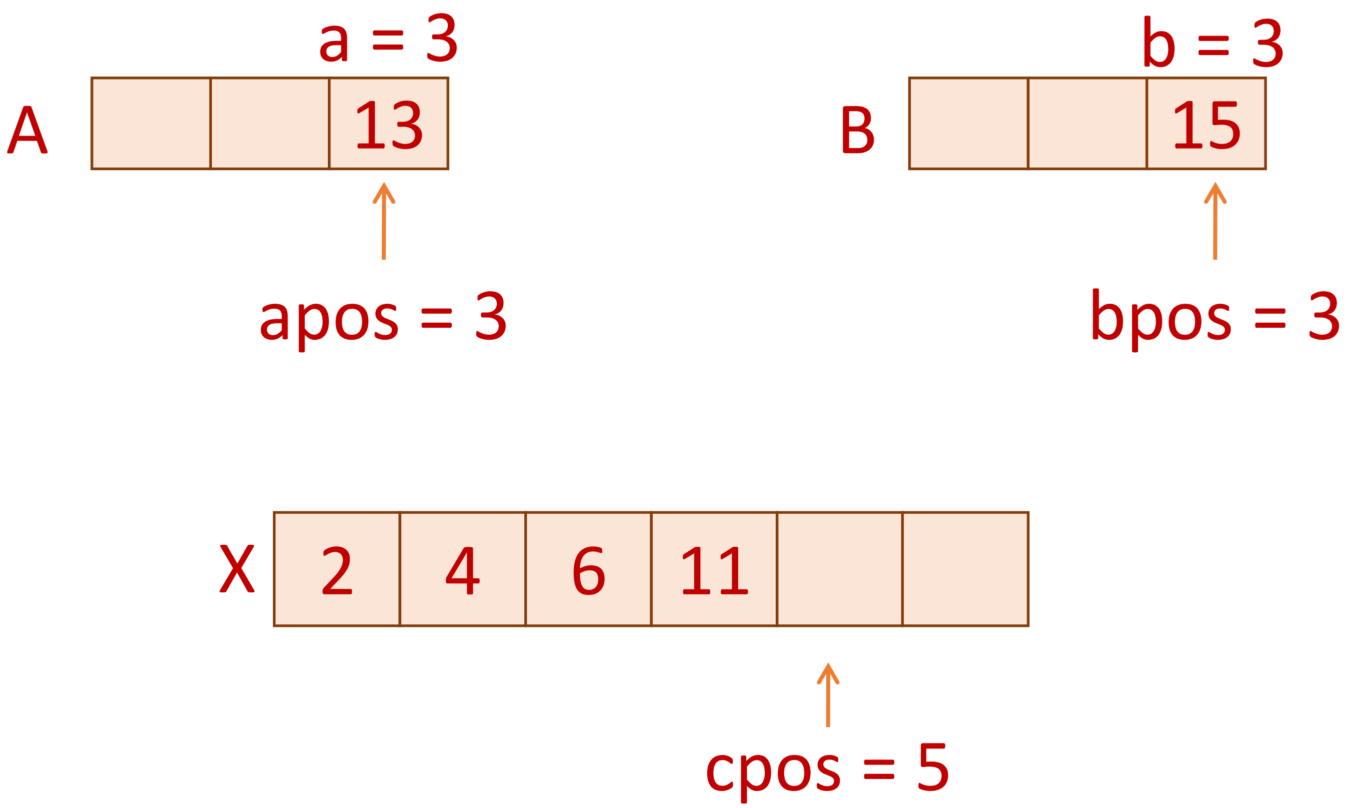
Merge Sort



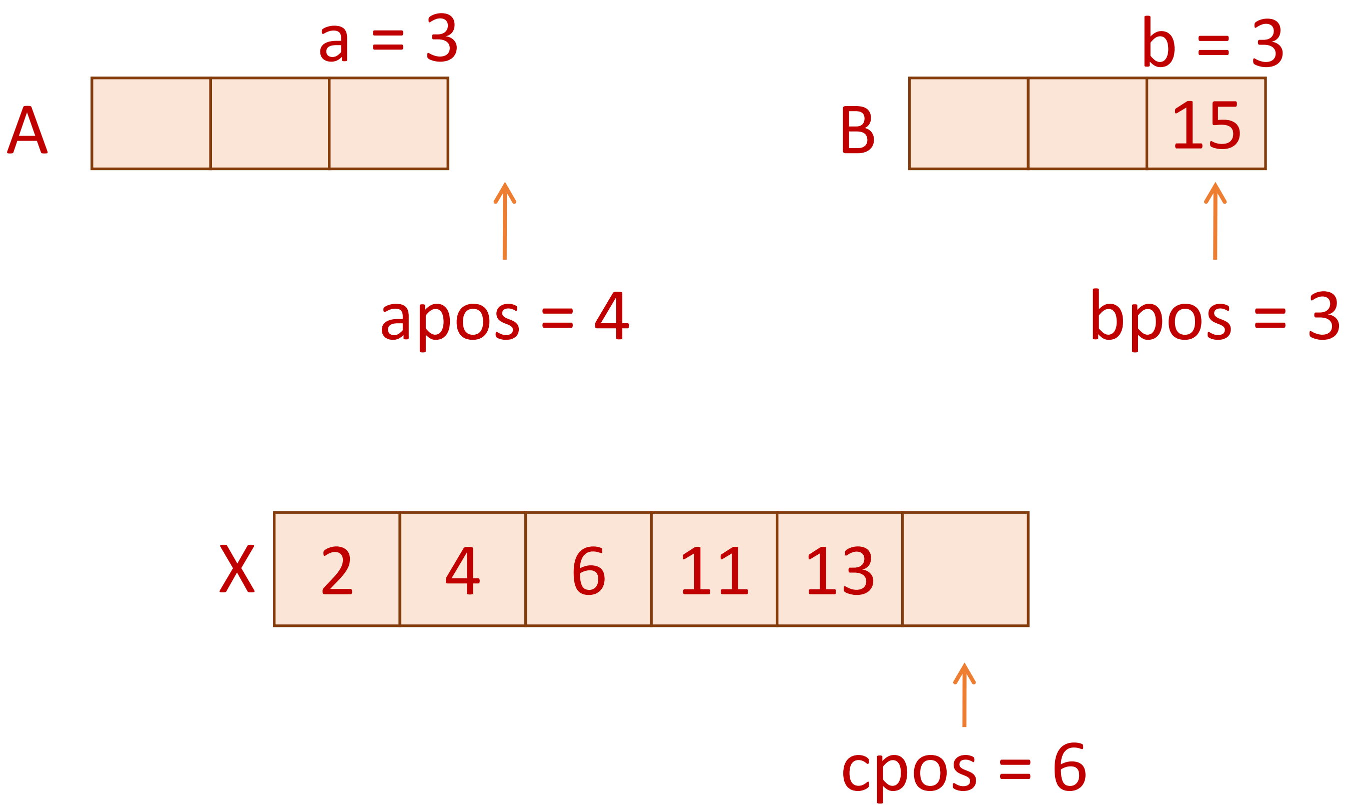
Merge Sort



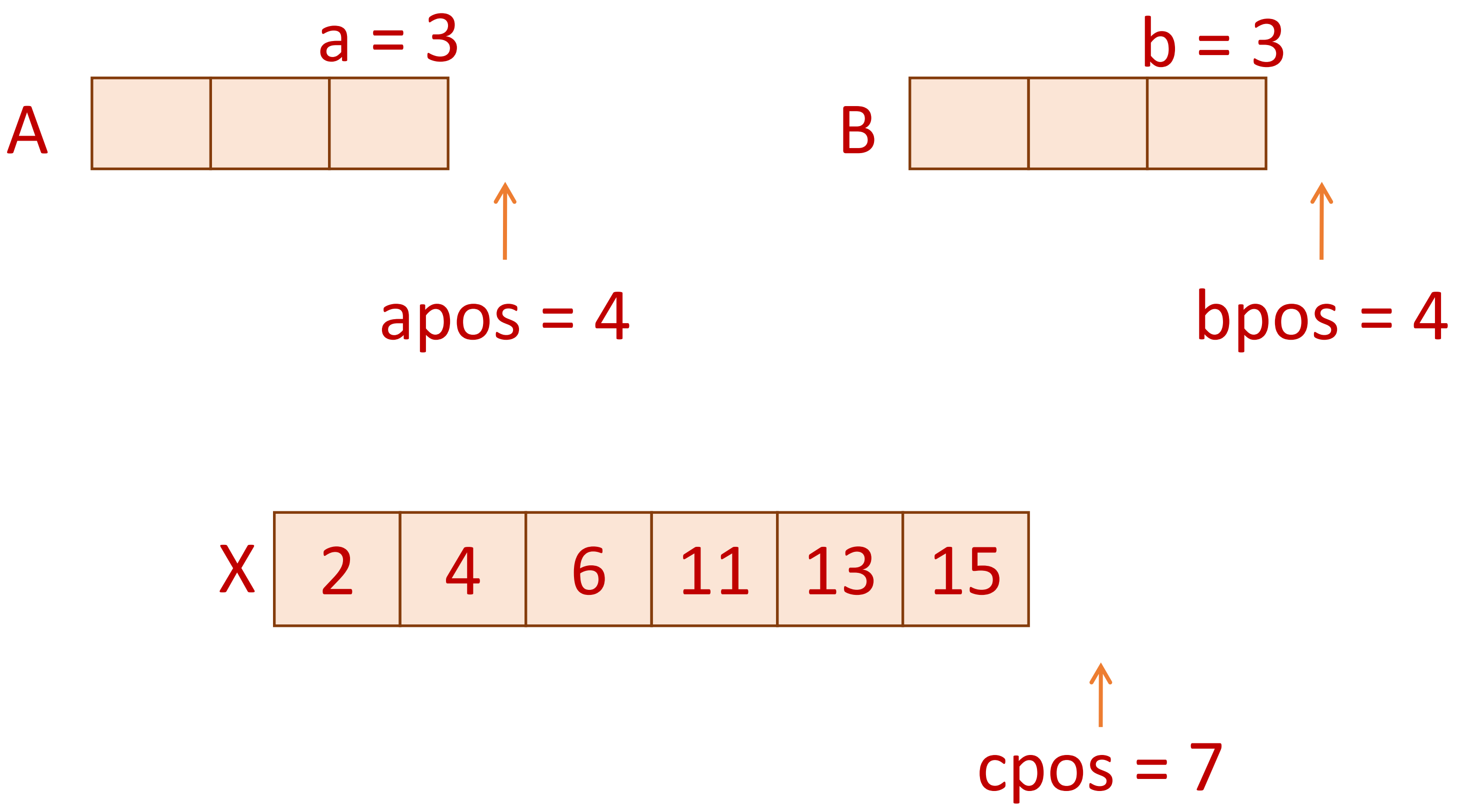
Merge Sort



Merge Sort



Merge Sort



Merge Sort

Merge Sort

Mergesort's efficiency:

- Merging efficiency is in $O(n)$.
- Merge operation is called $O(\log_2 n)$ time recursively.
- Hence, Mergesort complexity is in $O(n \log_2 n)$.
- Note: Mergesort uses an additional array $x[1..n]$. If x was local to merge, much more storage would be used because of recursive calls.

Other Issues With Sorting

Other issues with sorting

- Sorting strings
- Sorting structures
- A good general strategy in sorting things that you do not want to swap is to construct an array of pointers to the objects and sort that.
- If $A[P[i]] \; > \; A[P[j]]$ then

$$tmp \; = \; P[i]$$
$$P[i] = P[j]$$
$$P[j] \; = \; tmp$$

Other Issues With Sorting

Other issues with sorting

- Comparisons between data types do not have to use < or >
- Any sort can be programmed to use a pointer to an external comparison procedure
- Instead of

    if T[i] > T[j] then swap T[i] and T[j]

  use

    if not ordered(T[i], T[j]) then swap T[i] and T[j]