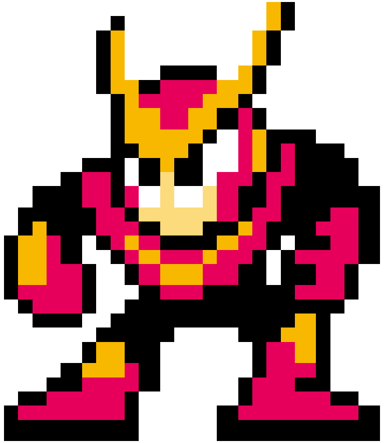


Lecture 3 (Sorting 1)

Insertion Sort and Merge sort

Analysis of Algorithms

- The theoretical study of computer-program **performance** and **resource usage**.

- You can’t understand it unless you analyze it.
- There are analytical and computational methods.

  Both verify each other.

- In this course we are interested of studying analytical methods.
- It requires a notable mathematical background.
- Sorting problem is an abstract model of analysis of algorithms.

Why study algorithms and performance?

- Algorithms help us to understand scalability.
- Performance often draws the line between what is feasible and what is impossible.
- Algorithmic mathematics provides a language for talking about program behavior.
- Performance is the currency of computing.
- The lessons of program performance generalize to other computing resources.
- Speed is fun!

Naive Insertion Sort

Insertion Sort

- **Naive Insertion Sort**
- In-Place Insertion Sort
- Insertion Sort Runtime

Merge sort

- Analyzing Merge sort
- Merge Sort Runtime

Insertion Sort

General strategy:

- Starting with an empty output sequence.
- Add each item from input, inserting into output at right point.

Naive approach, build entirely new output: Demo (Link)

Input:

32	15	2	17	19	26	41	17	17

Output:

Insertion Sort

General strategy:

- Starting with an empty output sequence.
- Add each item from input, inserting into output at right point.

Naive approach, build entirely new output:

Input:	32	15	2	17	19	26	41	17	17

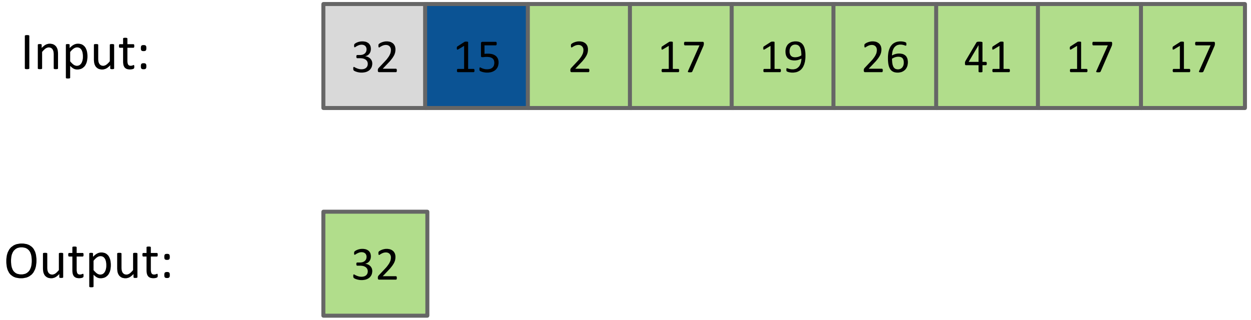
Output:

Insertion Sort

General strategy:

- Starting with an empty output sequence.
- Add each item from input, inserting into output at right point.

Naive approach, build entirely new output:



Input: 32 15 2 17 19 26 41 17 17

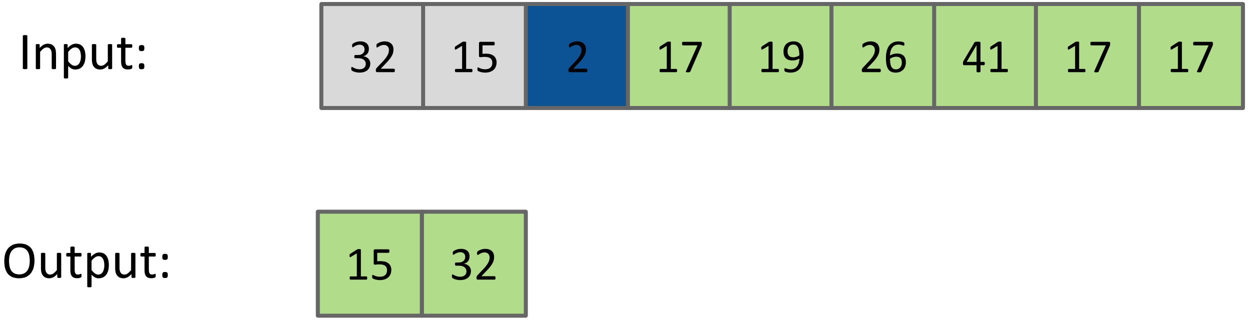
Output: 32

Insertion Sort

General strategy:

- Starting with an empty output sequence.
- Add each item from input, inserting into output at right point.

Naive approach, build entirely new output:



Input:	32	15	2	17	19	26	41	17	17

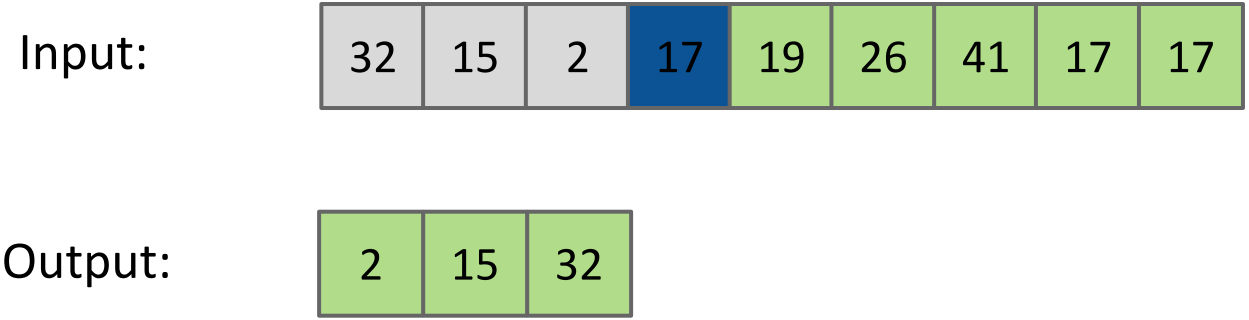
Output:	15	32

Insertion Sort

General strategy:

- Starting with an empty output sequence.
- Add each item from input, inserting into output at right point.

Naive approach, build entirely new output:



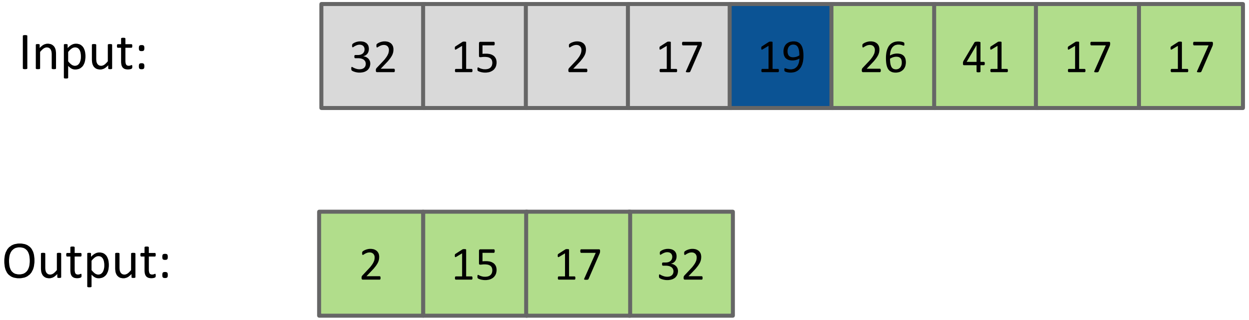
Input:	32	15	2	17	19	26	41	17	17
Output:	2	15	32						

Insertion Sort

General strategy:

- Starting with an empty output sequence.
- Add each item from input, inserting into output at right point.

Naive approach, build entirely new output:



Input:	32	15	2	17	19	26	41	17	17

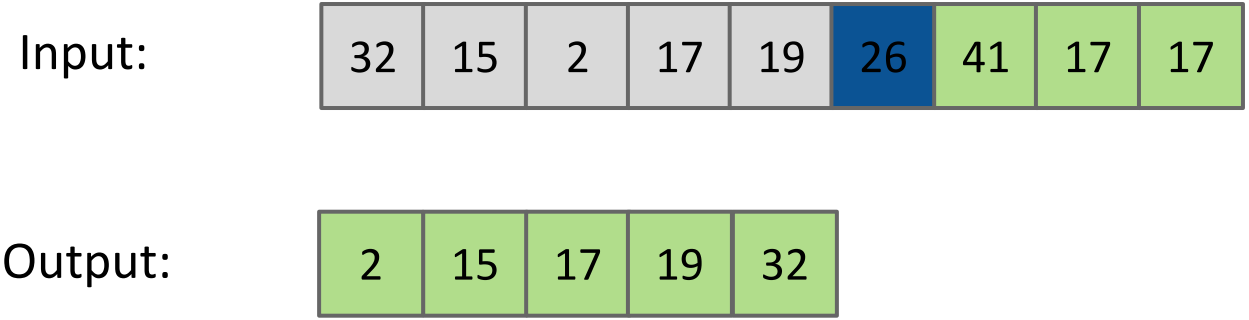
Output:	2	15	17	32

Insertion Sort

General strategy:

- Starting with an empty output sequence.
- Add each item from input, inserting into output at right point.

Naive approach, build entirely new output:



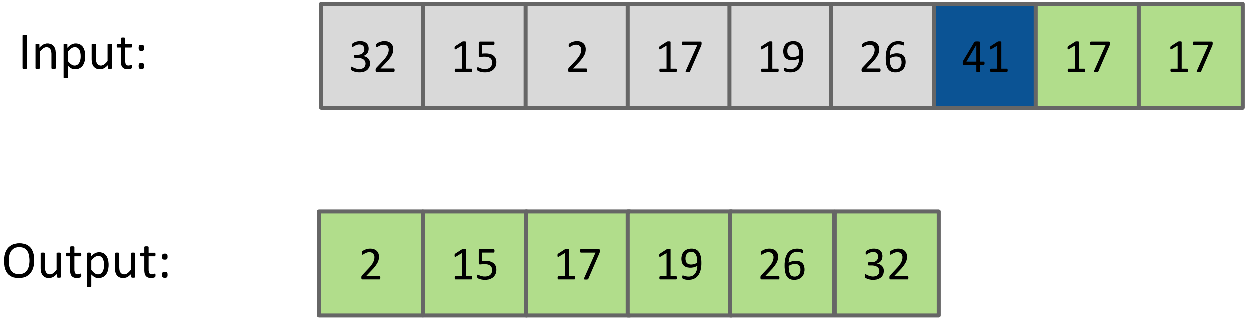
Input:	32	15	2	17	19	26	41	17	17
Output:	2	15	17	19	32				

Insertion Sort

General strategy:

- Starting with an empty output sequence.
- Add each item from input, inserting into output at right point.

Naive approach, build entirely new output:



Input: 32 15 2 17 19 26 41 17 17

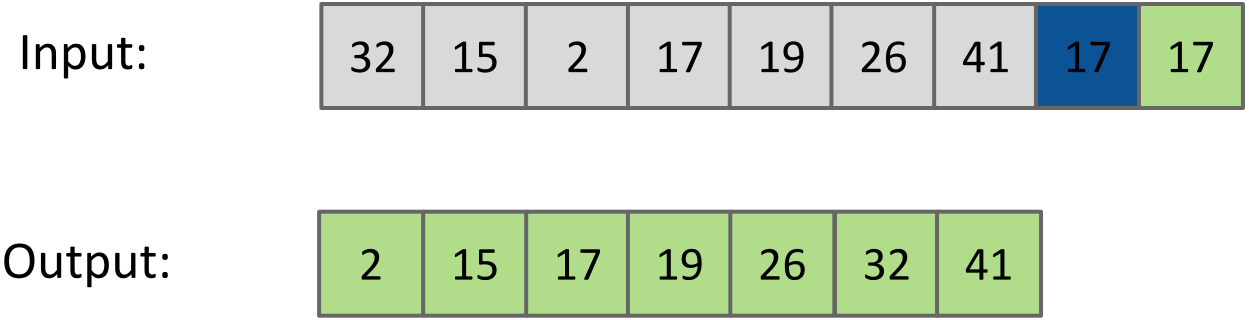
Output: 2 15 17 19 26 32

Insertion Sort

General strategy:

- Starting with an empty output sequence.
- Add each item from input, inserting into output at right point.

Naive approach, build entirely new output:



Input: 32 15 2 17 19 26 41 17 17

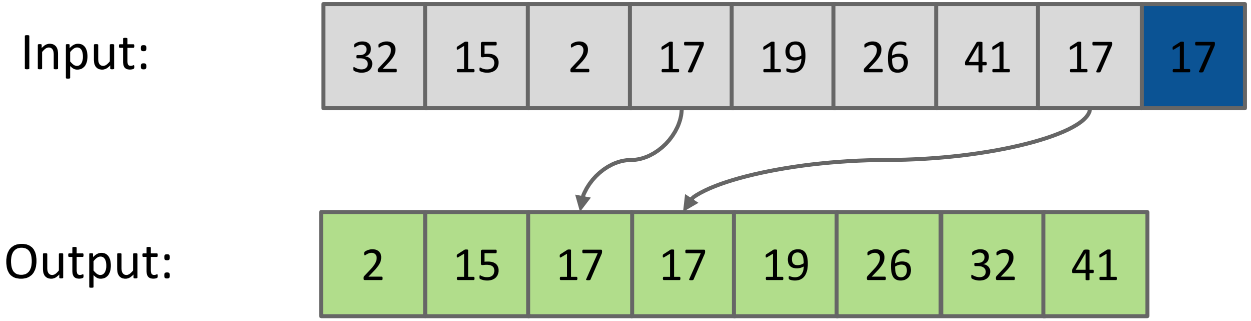
Output: 2 15 17 19 26 32 41

Insertion Sort

General strategy:

- Starting with an empty output sequence.
- Add each item from input, inserting into output at right point.

Naive approach, build entirely new output:



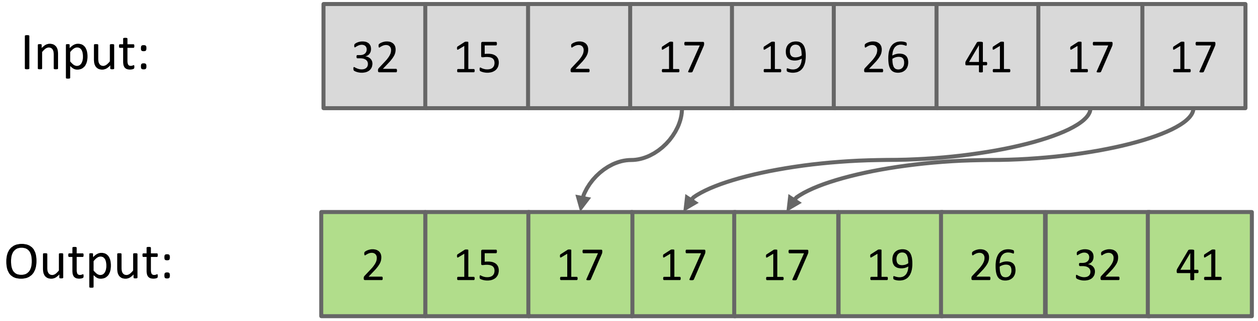
Input:	32	15	2	17	19	26	41	17	17
Output:	2	15	17	17	19	26	32	41	

Insertion Sort

General strategy:

- Starting with an empty output sequence.
- Add each item from input, inserting into output at right point.

Naive approach, build entirely new output:



Input:	32	15	2	17	19	26	41	17	17
Output:	2	15	17	17	17	19	26	32	41

In-Place Insertion Sort

Insertion Sort

- Naive Insertion Sort
- **In-Place Insertion Sort**
- Insertion Sort Runtime

Merge sort

- Analyzing Merge sort
- Merge Sort Runtime

In-place Insertion Sort

General strategy:

- Repeat for i = 0 to N - 1:
  - Designate item i as the traveling item.
  - Swap item backwards until traveller is in the right place among all previously examined items.

Input:	32	15	2	17	19	26	41	17	17

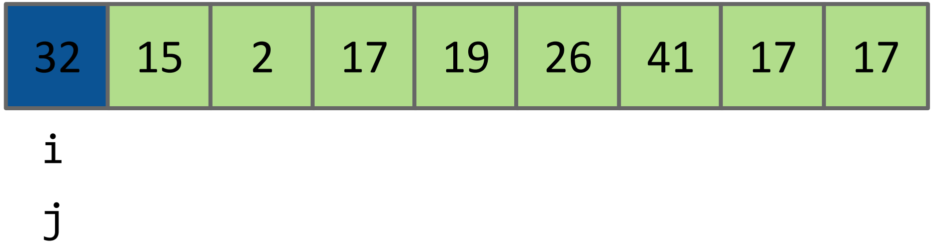
In example above: Use j pointer to track current spot of traveling item.

In-place Insertion Sort

General strategy:

- Repeat for i = 0 to N - 1:
  - **Designate item i as the traveling item.**
  - Swap item backwards until traveller is in the right place among all previously examined items.

Input:



32	15	2	17	19	26	41	17	17
i								
j								

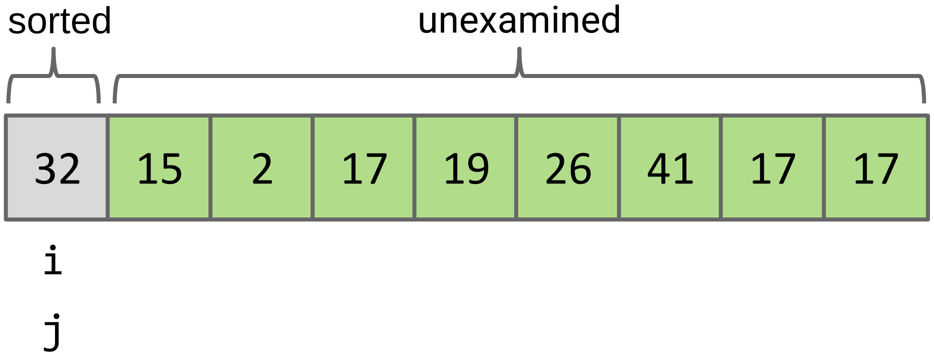
In example above: Use j pointer to track current spot of traveling item.

In-place Insertion Sort

General strategy:

- Repeat for i = 0 to N - 1:
  - Designate item i as the traveling item.
  - **Swap item backwards until traveller is in the right place among all previously examined items.**

Input:



32	15	2	17	19	26	41	17	17

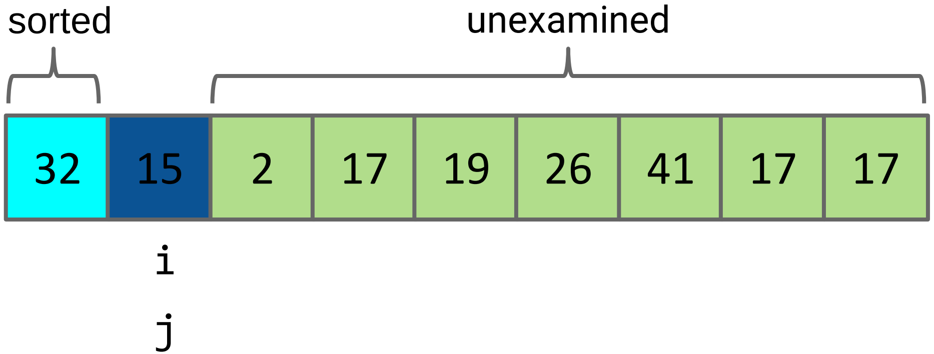
In example above: Use j pointer to track current spot of traveling item.

In-place Insertion Sort

General strategy:

- Repeat for i = 0 to N - 1:
  - **Designate item i as the traveling item.**
  - Swap item backwards until traveller is in the right place among all previously examined items.

Input:

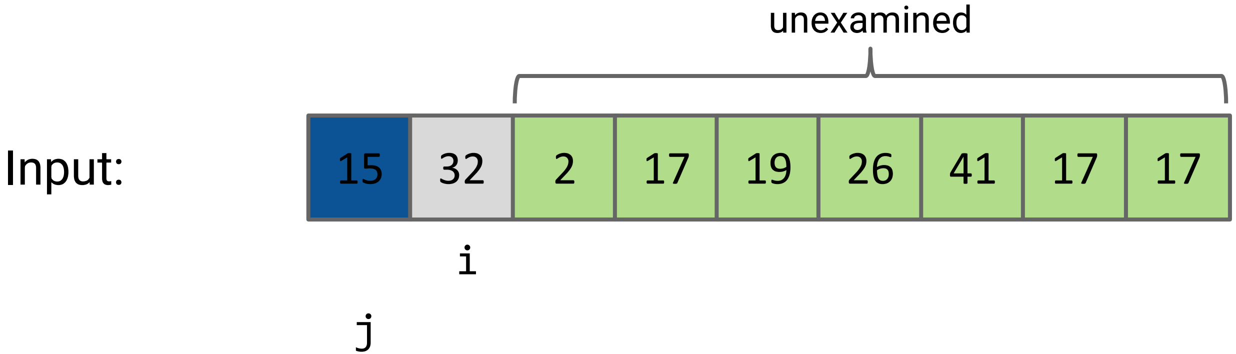


In example above: Use j pointer to track current spot of traveling item.

In-place Insertion Sort

General strategy:

- Repeat for i = 0 to N - 1:
  - Designate item i as the traveling item.
  - **Swap item backwards until traveller is in the right place among all previously examined items.**



unexamined

Input:

15	32	2	17	19	26	41	17	17

i

j

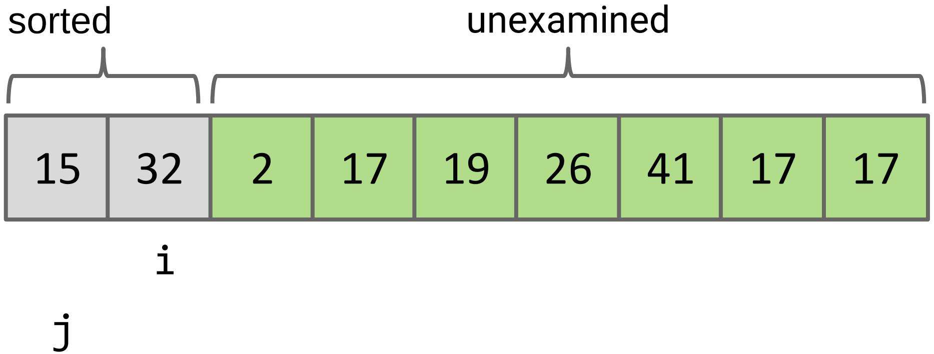
In example above: Use j pointer to track current spot of traveling item.

In-place Insertion Sort

General strategy:

- Repeat for i = 0 to N - 1:
    - Designate item i as the traveling item.
    - **Swap item backwards until traveller is in the right place among all previously examined items.**

Input:



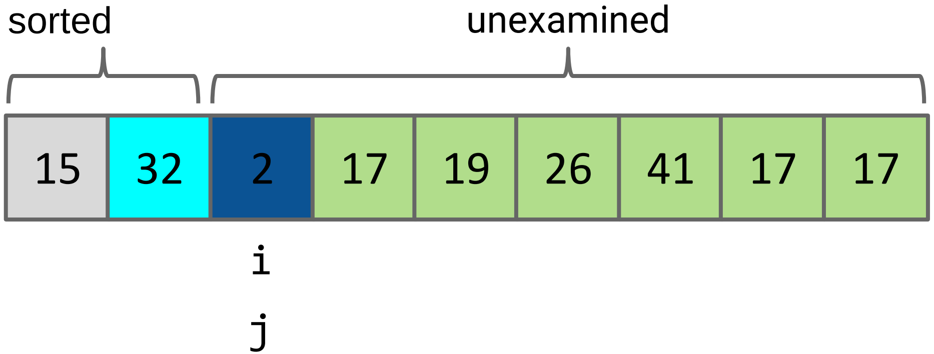
In example above: Use j pointer to track current spot of traveling item.

In-place Insertion Sort

General strategy:

- Repeat for i = 0 to N - 1:
  - **Designate item i as the traveling item.**
  - Swap item backwards until traveller is in the right place among all previously examined items.

Input:



15	32	2	17	19	26	41	17	17

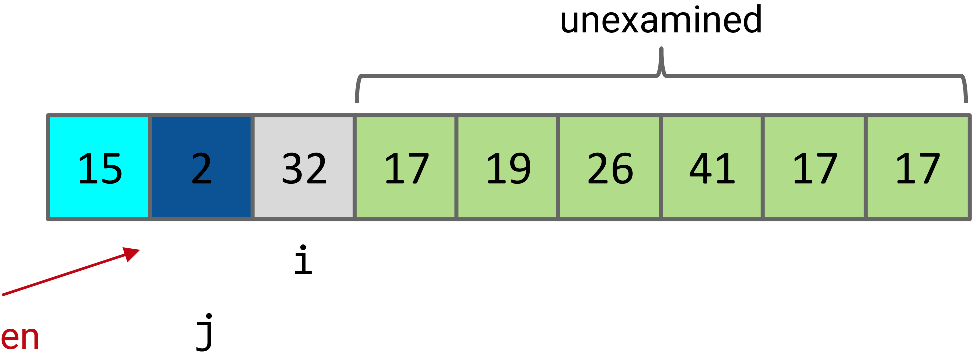
In example above: Use j pointer to track current spot of traveling item.

In-place Insertion Sort

General strategy:

- Repeat for i = 0 to N - 1:
  - Designate item i as the traveling item.
  - **Swap item backwards until traveller is in the right place among all previously examined items.**

Input:



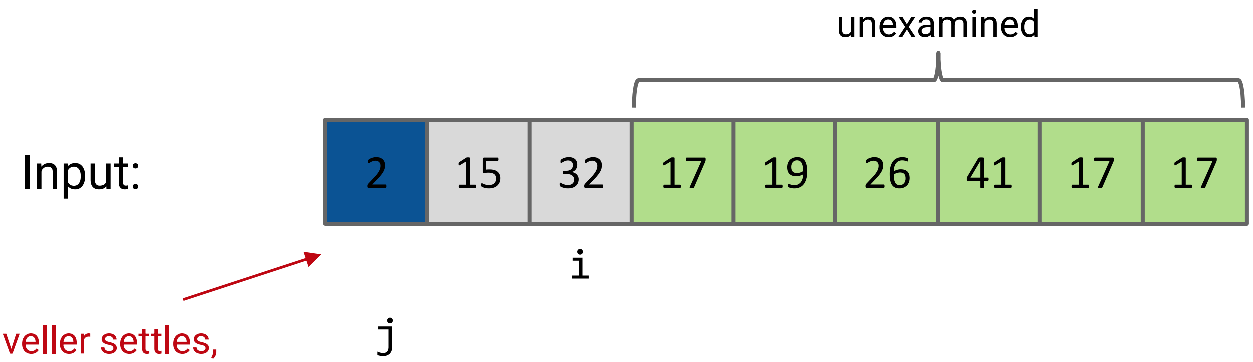
Note: We’ve temporarily broken our invariant that the items up through item i should be sorted!

In example above: Use j pointer to track current spot of traveling item.

In-place Insertion Sort

General strategy:

- Repeat for i = 0 to N - 1:
  - Designate item i as the traveling item.
  - **Swap item backwards until traveller is in the right place among all previously examined items.**



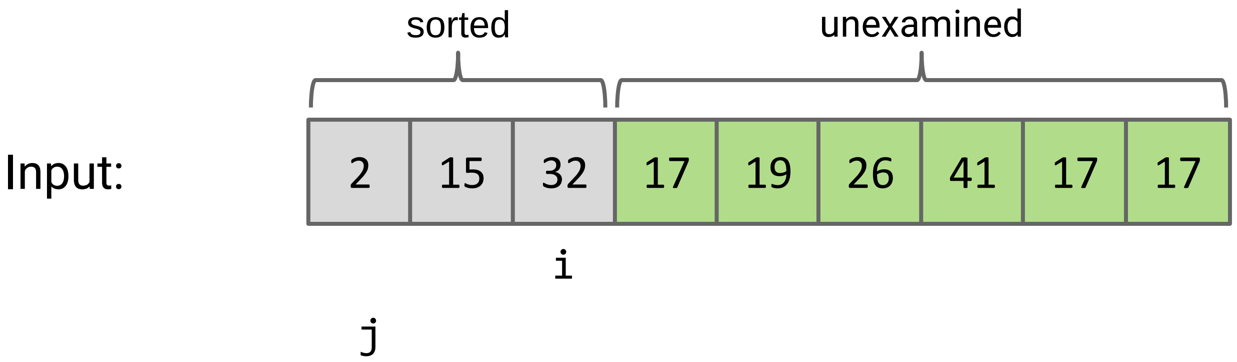
Once the traveller settles, the invariant is restored.

In example above: Use j pointer to track current spot of traveling item.

In-place Insertion Sort

General strategy:

- Repeat for i = 0 to N - 1:
    - Designate item i as the traveling item.
    - **Swap item backwards until traveller is in the right place among all previously examined items.**



2	15	32	17	19	26	41	17	17

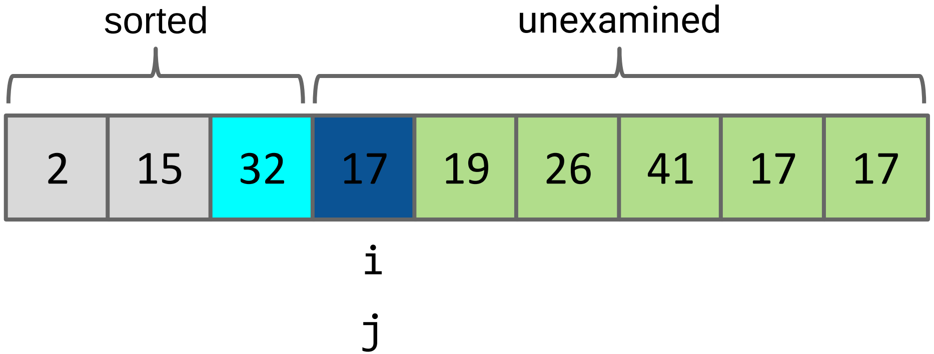
In example above: Use j pointer to track current spot of traveling item.

In-place Insertion Sort

General strategy:

- Repeat for i = 0 to N - 1:
  - **Designate item i as the traveling item.**
  - Swap item backwards until traveller is in the right place among all previously examined items.

Input:



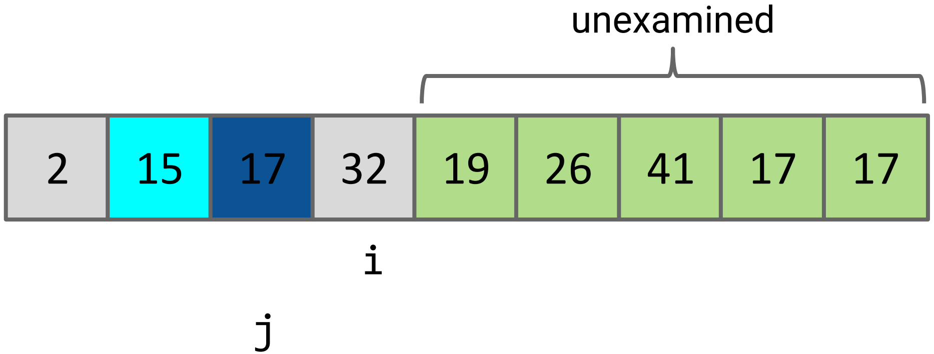
In example above: Use j pointer to track current spot of traveling item.

In-place Insertion Sort

General strategy:

- Repeat for i = 0 to N - 1:
    - Designate item i as the traveling item.
    - **Swap item backwards until traveller is in the right place among all previously examined items.**

Input:

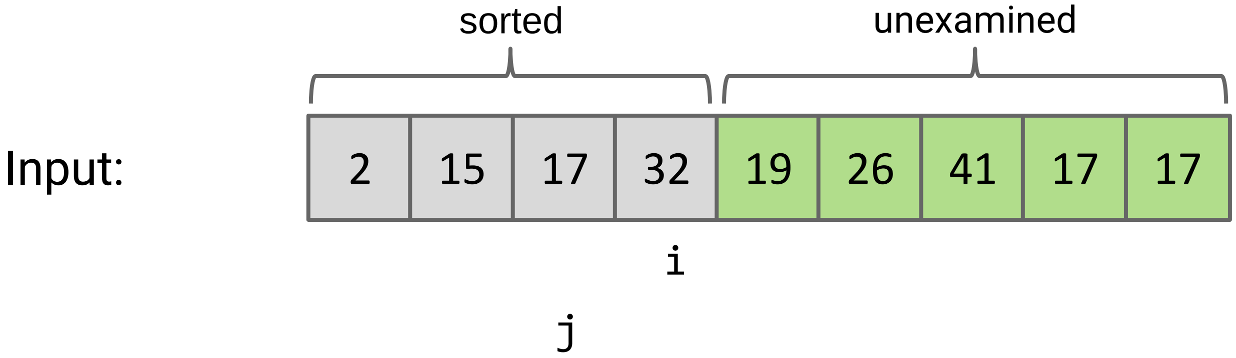


In example above: Use j pointer to track current spot of traveling item.

In-place Insertion Sort

General strategy:

- Repeat for i = 0 to N - 1:
    - Designate item i as the traveling item.
    - **Swap item backwards until traveller is in the right place among all previously examined items.**



Input:

2	15	17	32	19	26	41	17	17

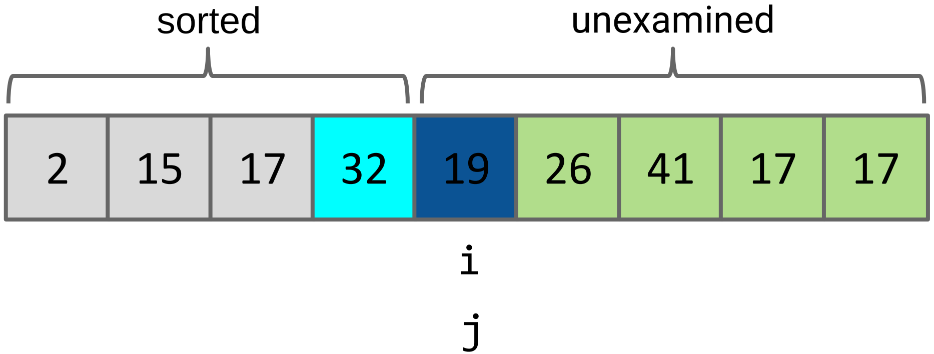
In example above: Use j pointer to track current spot of traveling item.

In-place Insertion Sort

General strategy:

- Repeat for i = 0 to N - 1:
    - **Designate item i as the traveling item.**
    - Swap item backwards until traveller is in the right place among all previously examined items.

Input:



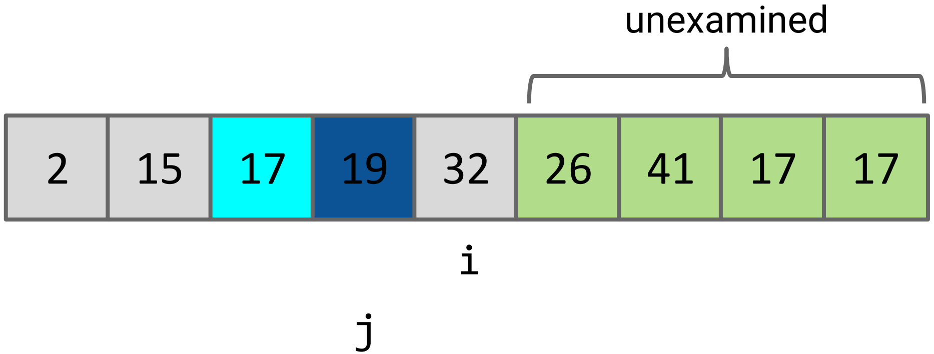
In example above: Use j pointer to track current spot of traveling item.

In-place Insertion Sort

General strategy:

- Repeat for i = 0 to N - 1:
  - Designate item i as the traveling item.
  - **Swap item backwards until traveller is in the right place among all previously examined items.**

Input:

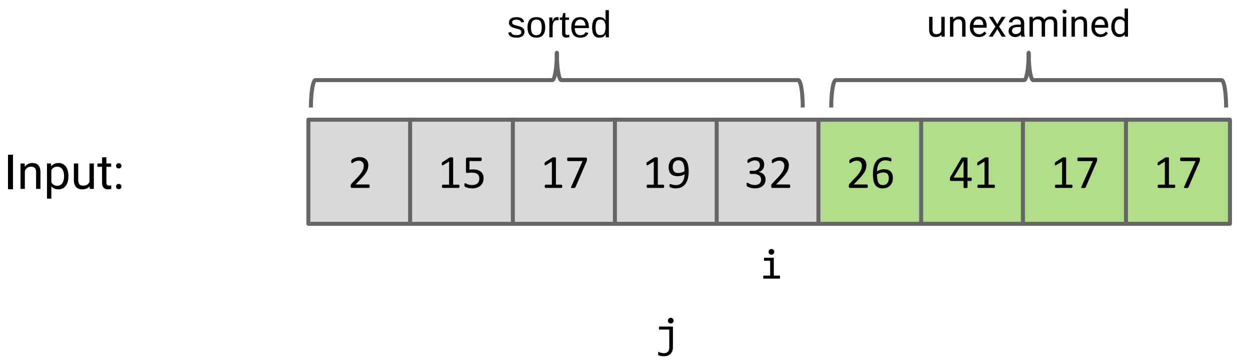


In example above: Use j pointer to track current spot of traveling item.

In-place Insertion Sort

General strategy:

- Repeat for i = 0 to N - 1:
  - Designate item i as the traveling item.
  - **Swap item backwards until traveller is in the right place among all previously examined items.**



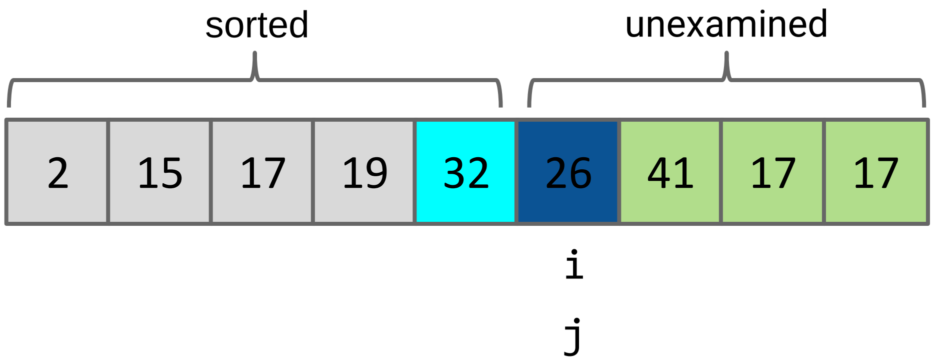
In example above: Use j pointer to track current spot of traveling item.

In-place Insertion Sort

General strategy:

- Repeat for i = 0 to N - 1:
    - **Designate item i as the traveling item.**
    - Swap item backwards until traveller is in the right place among all previously examined items.

Input:



2	15	17	19	32	26	41	17	17

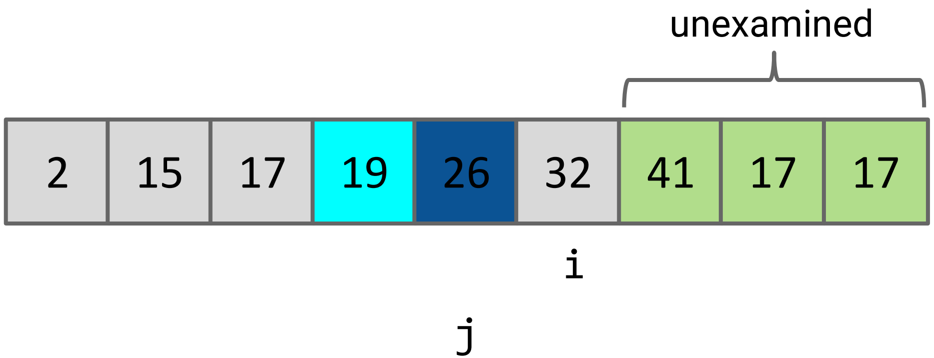
In example above: Use j pointer to track current spot of traveling item.

In-place Insertion Sort

General strategy:

- Repeat for i = 0 to N - 1:
  - Designate item i as the traveling item.
  - **Swap item backwards until traveller is in the right place among all previously examined items.**

Input:



2	15	17	19	26	32	41	17	17

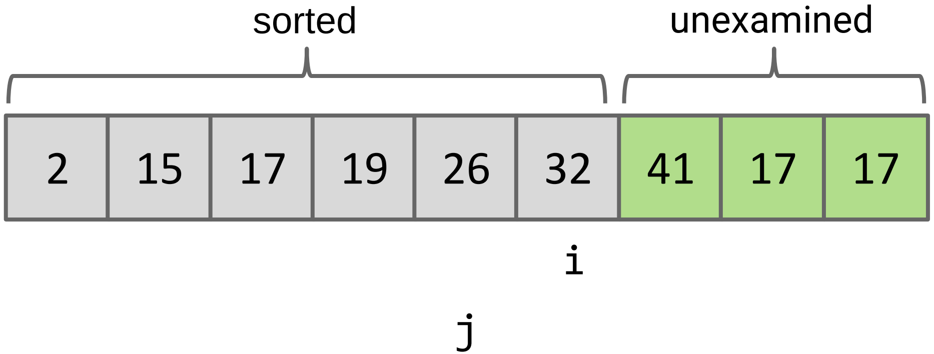
In example above: Use j pointer to track current spot of traveling item.

In-place Insertion Sort

General strategy:

- Repeat for i = 0 to N - 1:
  - Designate item i as the traveling item.
  - **Swap item backwards until traveller is in the right place among all previously examined items.**

Input:



2	15	17	19	26	32	41	17	17

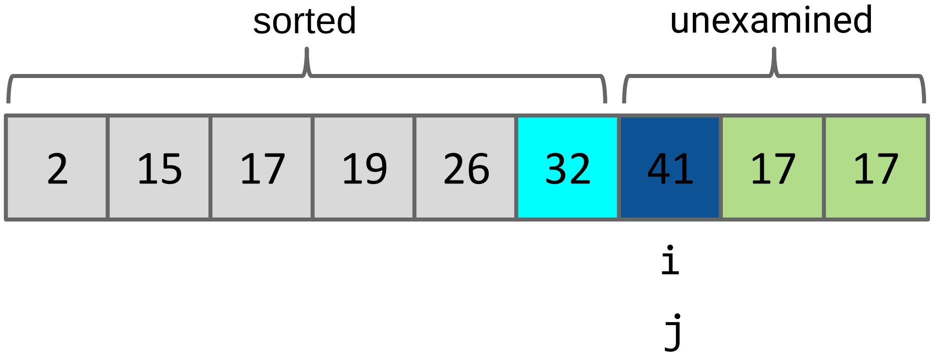
In example above: Use j pointer to track current spot of traveling item.

In-place Insertion Sort

General strategy:

- Repeat for i = 0 to N - 1:
  - **Designate item i as the traveling item.**
  - Swap item backwards until traveller is in the right place among all previously examined items.

Input:



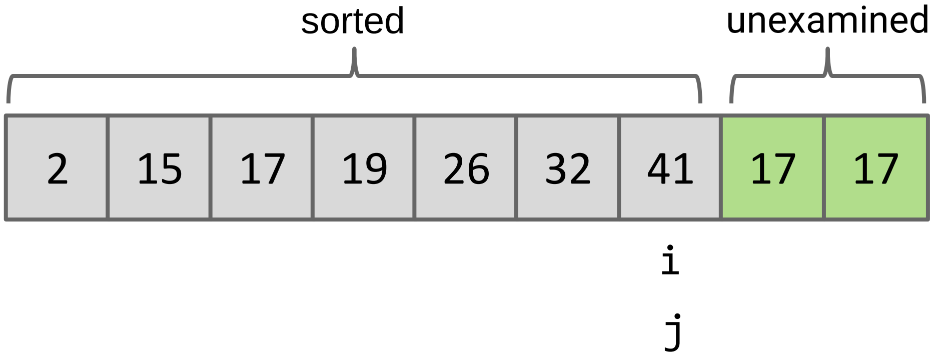
In example above: Use j pointer to track current spot of traveling item.

In-place Insertion Sort

General strategy:

- Repeat for i = 0 to N - 1:
    - Designate item i as the traveling item.
    - **Swap item backwards until traveller is in the right place among all previously examined items.**

Input:



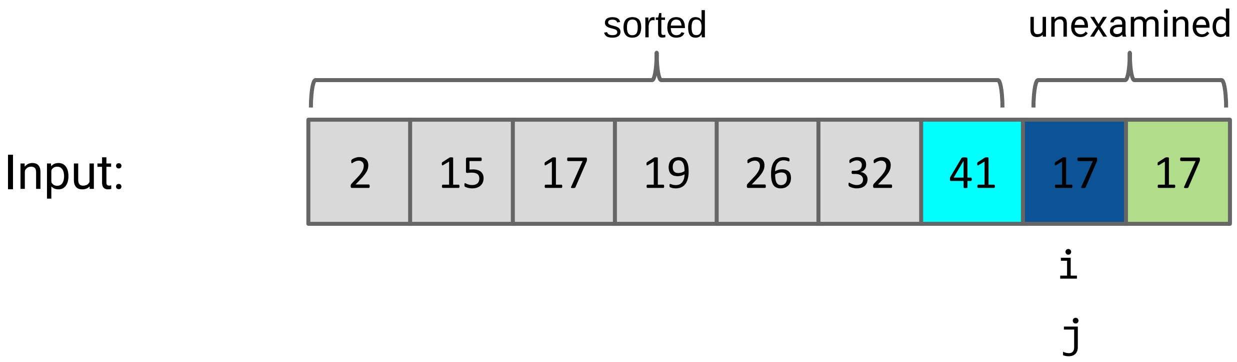
2	15	17	19	26	32	41	17	17

In example above: Use j pointer to track current spot of traveling item.

In-place Insertion Sort

General strategy:

- Repeat for i = 0 to N - 1:
  - **Designate item i as the traveling item.**
  - Swap item backwards until traveller is in the right place among all previously examined items.



sorted

unexamined

Input:

2 15 17 19 26 32 41 17 17

i

j

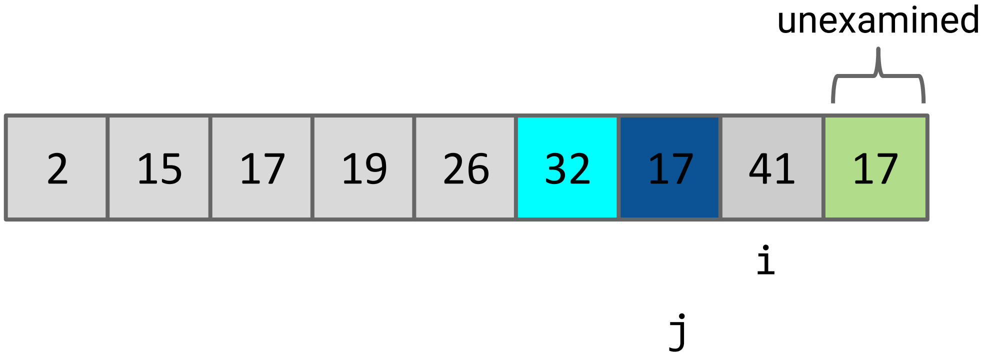
In example above: Use j pointer to track current spot of traveling item.

In-place Insertion Sort

General strategy:

- Repeat for i = 0 to N - 1:
  - Designate item i as the traveling item.
  - **Swap item backwards until traveller is in the right place among all previously examined items.**

Input:



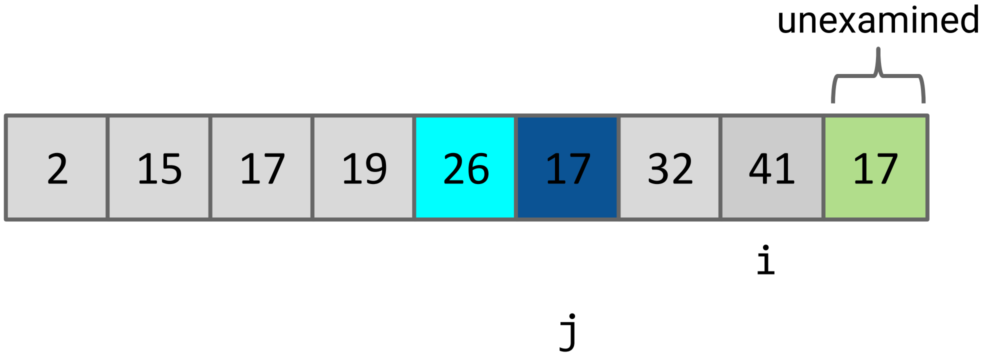
In example above: Use j pointer to track current spot of traveling item.

In-place Insertion Sort

General strategy:

- Repeat for i = 0 to N - 1:
    - Designate item i as the traveling item.
    - **Swap item backwards until traveller is in the right place among all previously examined items.**

Input:



unexamined

2	15	17	19	26	17	32	41	17

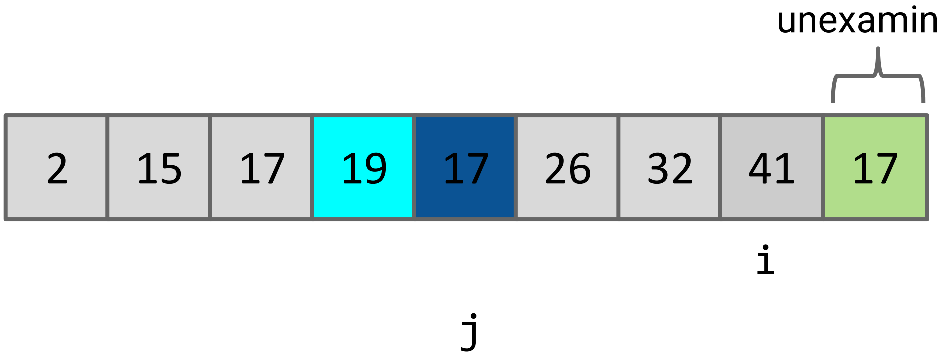
In example above: Use j pointer to track current spot of traveling item.

In-place Insertion Sort

General strategy:

- Repeat for i = 0 to N - 1:
    - Designate item i as the traveling item.
    - **Swap item backwards until traveller is in the right place among all previously examined items.**

Input:



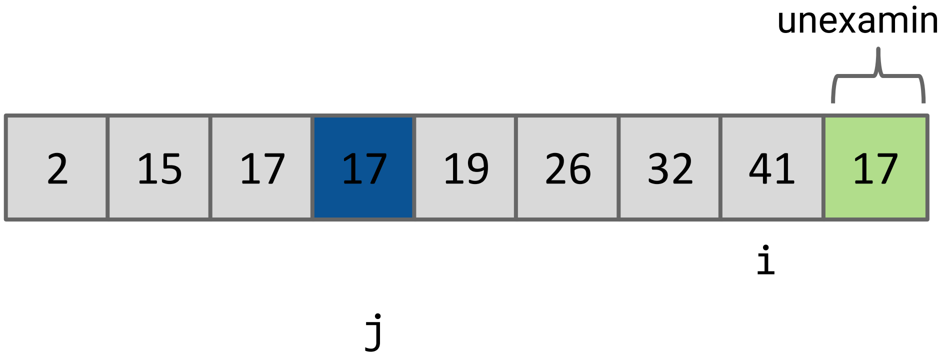
In example above: Use j pointer to track current spot of traveling item.

In-place Insertion Sort

General strategy:

- Repeat for i = 0 to N - 1:
  - Designate item i as the traveling item.
  - **Swap item backwards until traveller is in the right place among all previously examined items.**

Input:

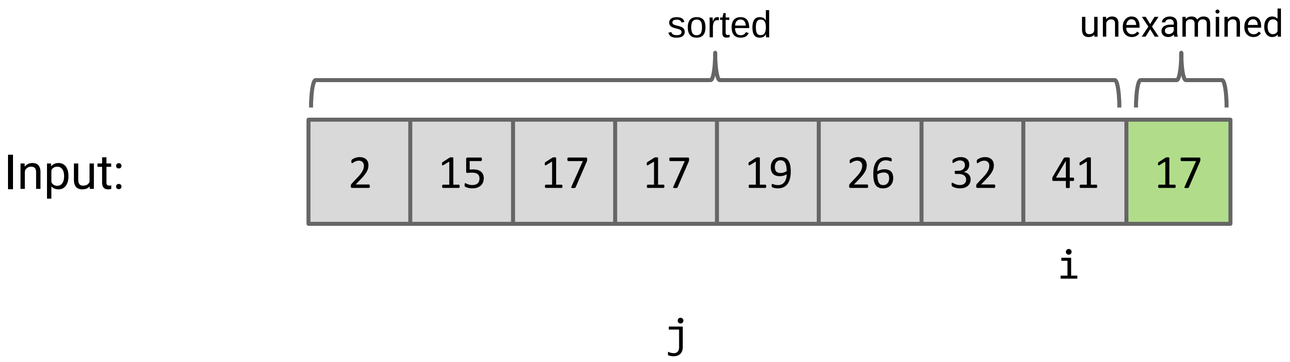


In example above: Use j pointer to track current spot of traveling item.

In-place Insertion Sort

General strategy:

- Repeat for i = 0 to N - 1:
  - Designate item i as the traveling item.
  - **Swap item backwards until traveller is in the right place among all previously examined items.**



2	15	17	17	19	26	32	41	17

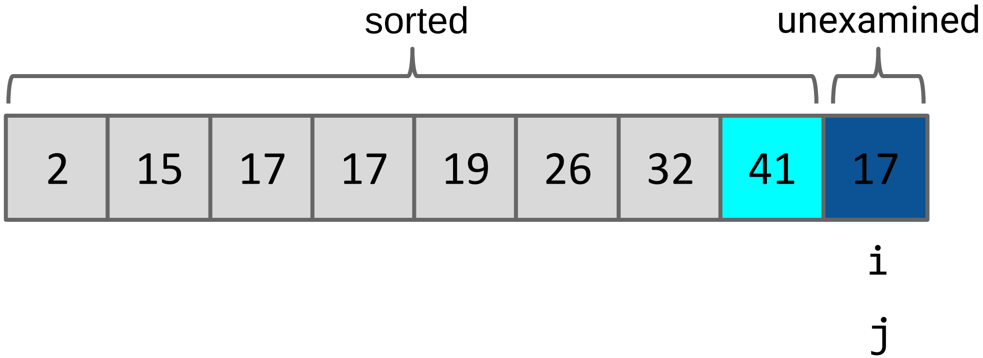
In example above: Use j pointer to track current spot of traveling item.

In-place Insertion Sort

General strategy:

- Repeat for i = 0 to N - 1:
  - **Designate item i as the traveling item.**
  - Swap item backwards until traveller is in the right place among all previously examined items.

Input:



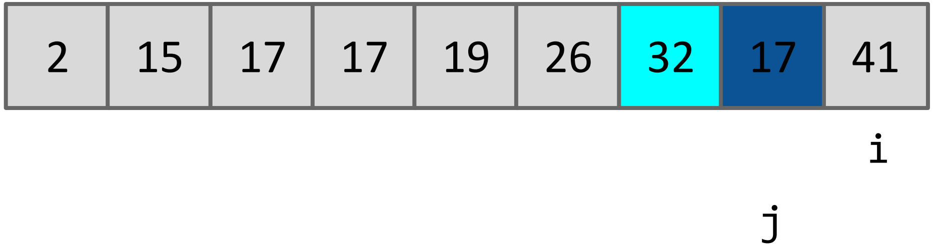
In example above: Use j pointer to track current spot of traveling item.

In-place Insertion Sort

General strategy:

- Repeat for i = 0 to N - 1:
  - Designate item i as the traveling item.
  - **Swap item backwards until traveller is in the right place among all previously examined items.**

Input:



2	15	17	17	19	26	32	17	41

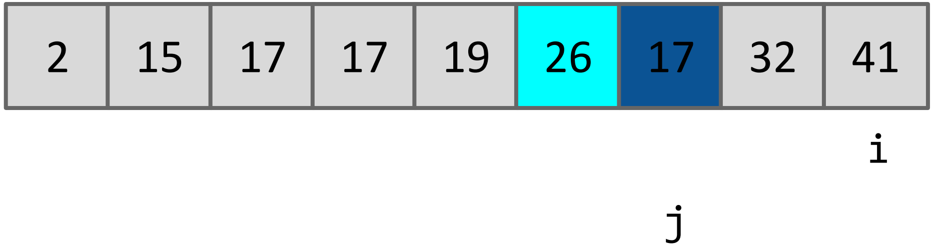
In example above: Use j pointer to track current spot of traveling item.

In-place Insertion Sort

General strategy:

- Repeat for i = 0 to N - 1:
  - Designate item i as the traveling item.
  - **Swap item backwards until traveller is in the right place among all previously examined items.**

Input:



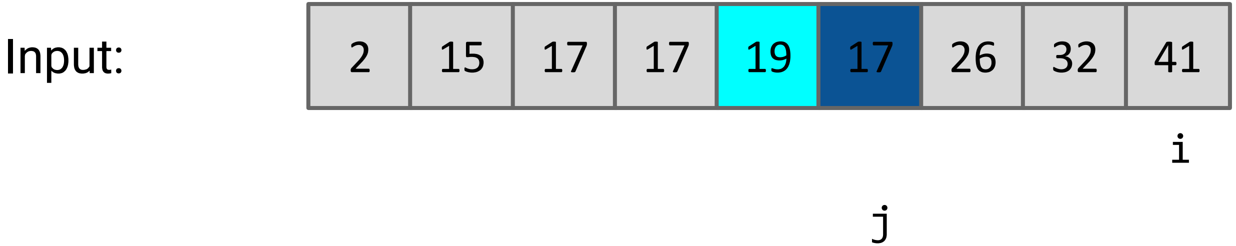
2	15	17	17	19	26	17	32	41

In example above: Use j pointer to track current spot of traveling item.

In-place Insertion Sort

General strategy:

- Repeat for i = 0 to N - 1:
    - Designate item i as the traveling item.
    - **Swap item backwards until traveller is in the right place among all previously examined items.**



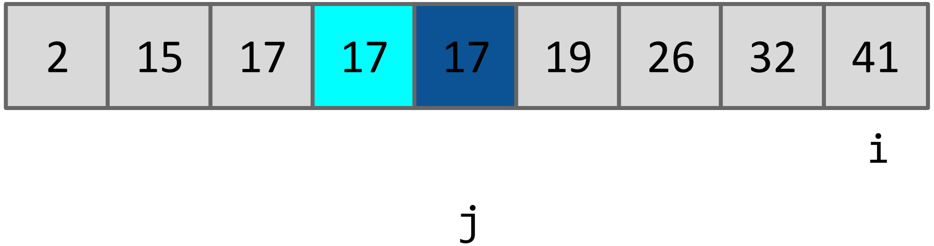
In example above: Use j pointer to track current spot of traveling item.

In-place Insertion Sort

General strategy:

- Repeat for i = 0 to N - 1:
  - Designate item i as the traveling item.
  - **Swap item backwards until traveller is in the right place among all previously examined items.**

Input:



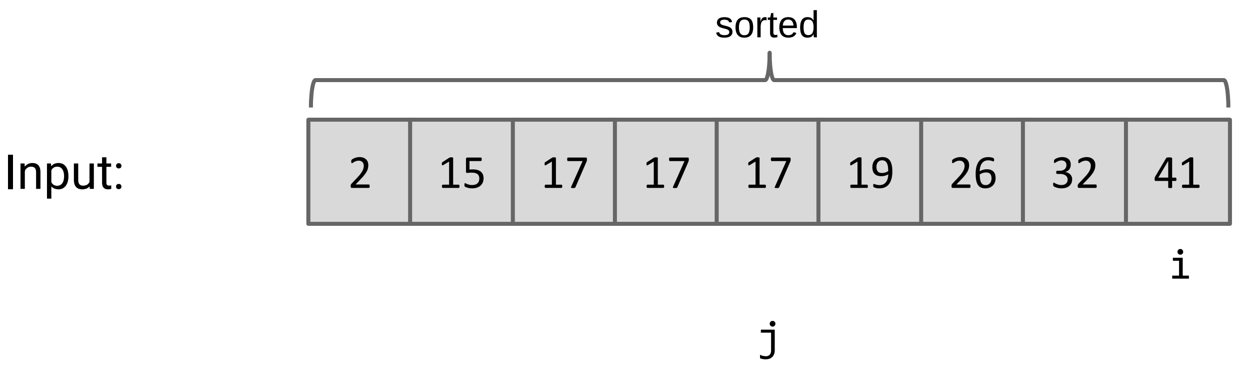
2	15	17	17	17	19	26	32	41
								i
				j				

In example above: Use j pointer to track current spot of traveling item.

In-place Insertion Sort

General strategy:

- Repeat for i = 0 to N - 1:
  - Designate item i as the traveling item.
  - **Swap item backwards until traveller is in the right place among all previously examined items.**



Input:

2	15	17	17	17	19	26	32	41

In example above: Use j pointer to track current spot of traveling item.

In-Place Insertion Sort Algorithm

Insertion Sort

- Naive Insertion Sort
- **In-Place Insertion Sort**
- Insertion Sort Runtime

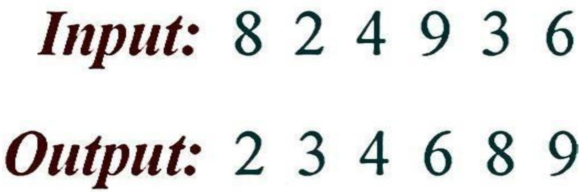
Merge sort

- Analyzing Merge sort
- Merge Sort Runtime

The problem of sorting

- **Input:** sequence ⟨a1, a2, …, an⟩ of numbers.
- **Output:** permutation ⟨a'1, a'2, …, a'n⟩ such that

a'1 ≤ a'2 ≤ … ≤ a'n.



Input: 8 2 4 9 3 6

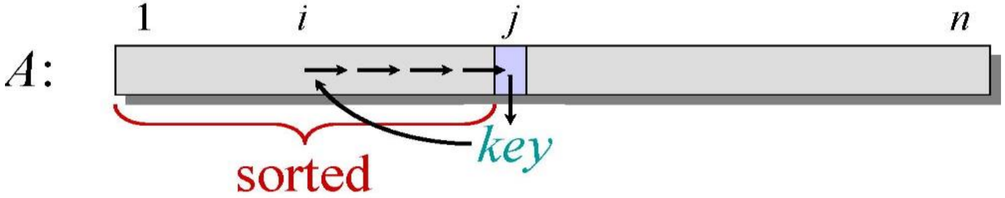
Output: 2 3 4 6 8 9

- What is the difference between sequence and permutation?
- Does this difference affect your formulation and solution?

Insertion Sort

“pseudocode”

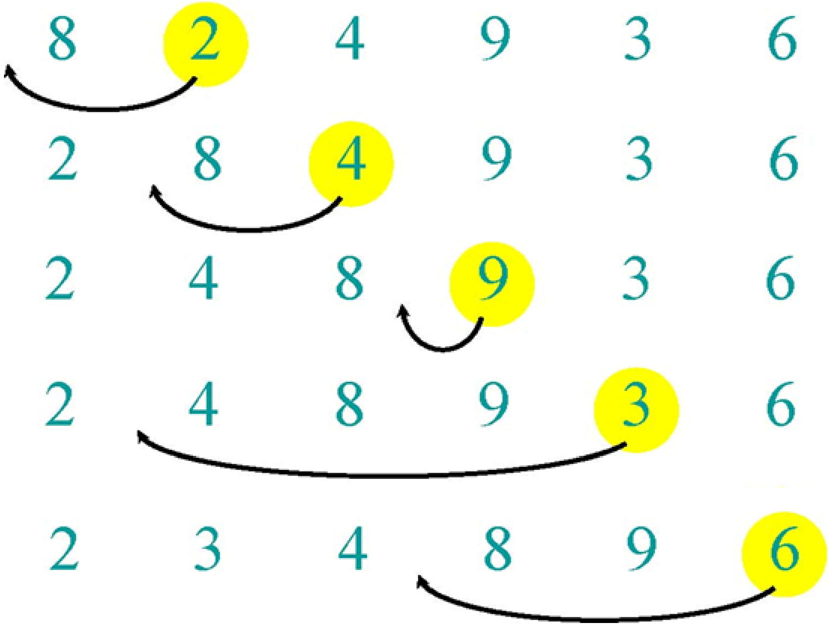
```
INSERTION-SORT $(A, n)$        ▷ $A[1 \,.\,.\, n]$
    for $j \leftarrow 2$ to $n$
        do $key \leftarrow A[j]$
           $i \leftarrow j - 1$
           while $i > 0$ and $A[i] > key$
               do $A[i+1] \leftarrow A[i]$
                  $i \leftarrow i - 1$
           $A[i+1] = key$
```



Example of insertion sort

```
INSERTION-SORT (A, n)    ▷ A[1 . . n]
  for j ← 2 to n
      do key ← A[j]
         i ← j – 1
         while i > 0 and A[i] > key
             do A[i+1] ← A[i]
                i ← i – 1
         A[i+1] = key
```

Input: 8 2 4 9 3 6



8 2 4 9 3 6

2 8 4 9 3 6

2 4 8 9 3 6

2 4 8 9 3 6

2 3 4 8 9 6

Output: 2 3 4 6 8 9

Insertion Sort Runtime

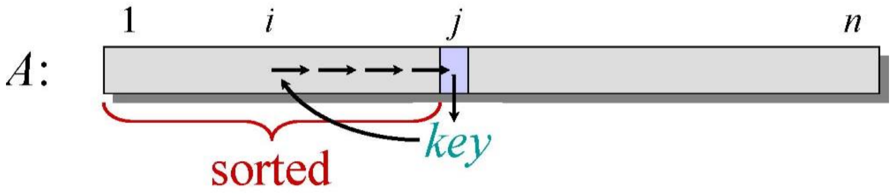
Insertion Sort

- Naive Insertion Sort
- In-Place Insertion Sort
- **Insertion Sort Runtime**

Merge sort

- Analyzing Merge sort
- Merge Sort Runtime

Insertion Sort Time



```
INSERTION-SORT(A)                                                    cost    times
for $j \leftarrow 2$ to $n$                                          $c_1$   $n$
    do $key \leftarrow A[j]$                                         $c_2$   $n-1$
        ▷ Insert $A[j]$ into the sorted sequence $A[1..j-1]$.        0       $n-1$
        $i \leftarrow j-1$                                           $c_4$   $n-1$
        while $i > 0$ and $A[i] > key$                              $c_5$   $\sum_{j=2}^{n} t_j$
            do $A[i+1] \leftarrow A[i]$                              $c_6$   $\sum_{j=2}^{n}(t_j-1)$
                $i \leftarrow i-1$                                   $c_7$   $\sum_{j=2}^{n}(t_j-1)$
        $A[i+1] \leftarrow key$                                      $c_8$   $n-1$
```

Running time

- Parameterize the running time by the size of the input, since short The running time depends on the input: an already sorted
- sequences are easier to sort than long ones.
- Generally, we seek upper bounds on the running time,

  because everybody likes a guarantee.

Kinds of analyses

- **Worst-case:** (usually)

  T(n) =maximum time of algorithm on any input of size n.

- **Average-case:** (sometimes)

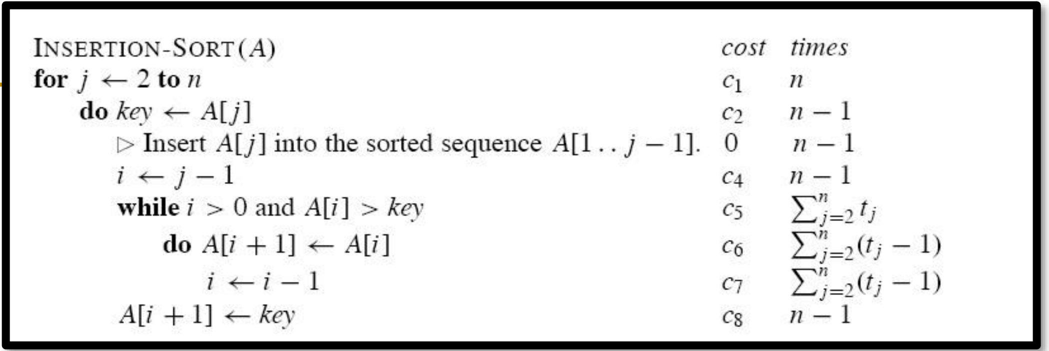
  T(n) =expected time of algorithm over all inputs of size n.

- Need assumption of statistical distribution of inputs.

  Best-case: (do not care !)

- We care about average-case and worst-case analysis (similar in many cases.)

Best Case Analysis



```
INSERTION-SORT(A)                                                  cost    times
for $j \leftarrow 2$ to $n$                                        $c_1$   $n$
    do $key \leftarrow A[j]$                                       $c_2$   $n-1$
        ▷ Insert $A[j]$ into the sorted sequence $A[1..j-1]$.      $0$     $n-1$
        $i \leftarrow j-1$                                         $c_4$   $n-1$
        while $i > 0$ and $A[i] > key$                             $c_5$   $\sum_{j=2}^{n} t_j$
            do $A[i+1] \leftarrow A[i]$                            $c_6$   $\sum_{j=2}^{n} (t_j - 1)$
                $i \leftarrow i-1$                                 $c_7$   $\sum_{j=2}^{n} (t_j - 1)$
        $A[i+1] \leftarrow key$                                    $c_8$   $n-1$
```

- The array is already sorted.

- Always find that A[i] ≤ key upon the first time the while loop test is run (when i = j − 1).

- All t_j are 1.

- Running time is : $\mathbf{T\,(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1)}$

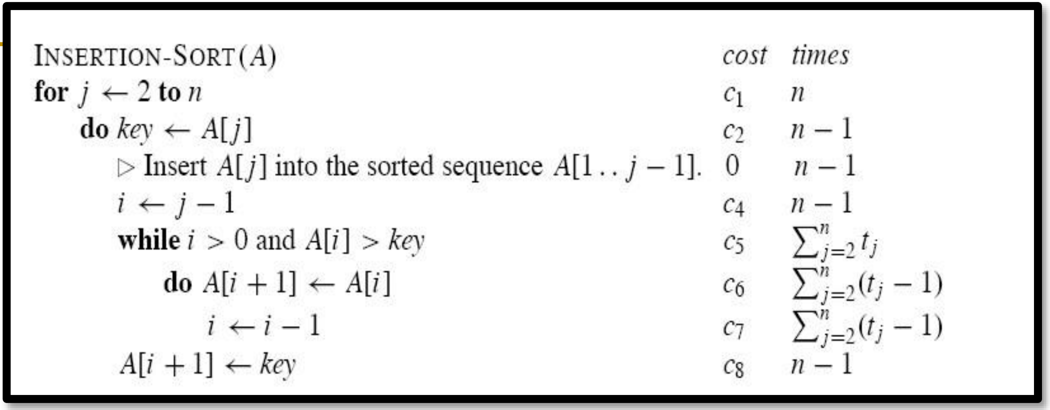
  $\mathbf{= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8)}$.

- Can express T (n) as **an +b** for constants a and b (that depend on the statement costs c_i)⇒ T (n) is a **linear function** of n.

Worst Case Analysis

- The array is in reverse sorted order.
- Always find that A[i] > key in while loop test.
- Have to compare key with all elements to the left of the j th position ⇒ compare with (j – 1) elements.
  - $\sum_{j=2}^{n} t_j = \sum_{j=2}^{n} j$ and $\sum_{j=2}^{n} (t_j - 1) = \sum_{j=2}^{n} (j - 1)$.
  - $\sum_{j=1}^{n} j$ is known as an ***arithmetic series***, it equals $\frac{n(n+1)}{2}$.

Worst Case Analysis



```
INSERTION-SORT(A)                                        cost   times
for $j \leftarrow 2$ to $n$                              $c_1$  $n$
    do $key \leftarrow A[j]$                             $c_2$  $n-1$
        ▷ Insert $A[j]$ into the sorted sequence $A[1..j-1]$.  0   $n-1$
        $i \leftarrow j-1$                               $c_4$  $n-1$
        while $i > 0$ and $A[i] > key$                   $c_5$  $\sum_{j=2}^{n} t_j$
            do $A[i+1] \leftarrow A[i]$                  $c_6$  $\sum_{j=2}^{n} (t_j - 1)$
                $i \leftarrow i-1$                       $c_7$  $\sum_{j=2}^{n} (t_j - 1)$
        $A[i+1] \leftarrow key$                          $c_8$  $n-1$
```

- Running time is

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) \\ &\quad + c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1) \\ &= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right)n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right)n \\ &\quad - (c_2 + c_4 + c_5 + c_8)\ . \end{aligned}$$

- Can express $T(n)$ as $an^2 + bn + c$ for constants a, b, c (that again depend on statement costs) $\Rightarrow T(n)$ is a *quadratic function* of n.

Order of growth

- Another abstraction to ease analysis and focus on the important features.
- Look only at the leading term of the formula for running time.
  - Drop lower-order terms.
  - Ignore the constant coefficient in the leading term.
- Example: For insertion sort:
- The worst-case running time is $\mathbf{an^2 + bn + c}$.
- Drop lower-order terms⇒ an^2.
- Ignore constant coefficient ⇒ n^2.

Order of growth

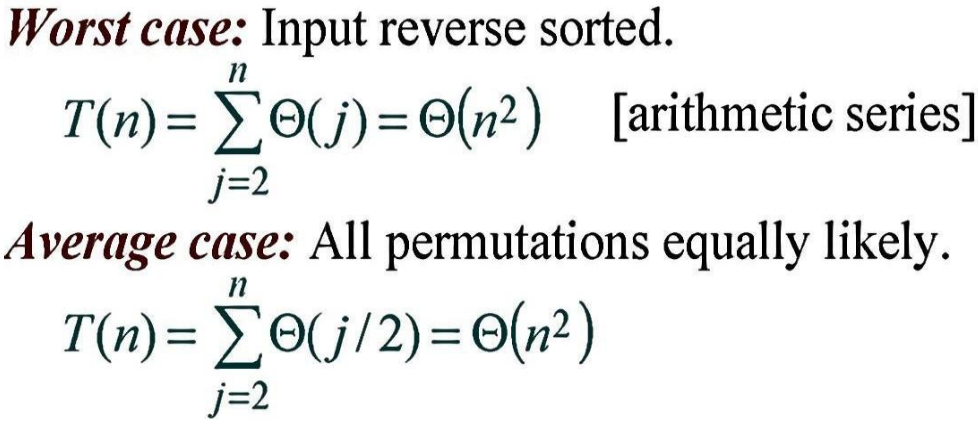
- But we can’t say that the worst-case running time T(n) equals n^2.
- **It grows like n^2. But it doesn't equal n^2.**
- We say that the running time is (n^2) to capture the notion that the order of growth is n^2.
- We usually consider one algorithm to be **more efficient** than another if its worst case running time has a **smaller order of growth**.
- Notice that the justification of the n^2 time of insertion sort can be explained by the existence of two nested loops (do you notice?!)

Insertion sort analysis

- Is insertion sort a fast sorting algorithm?

- Moderately so, for small n.

- Not at all, for large n.



Worst case: Input reverse sorted.

$$T(n) = \sum_{j=2}^{n} \Theta(j) = \Theta(n^2) \quad \text{[arithmetic series]}$$

Average case: All permutations equally likely.

$$T(n) = \sum_{j=2}^{n} \Theta(j/2) = \Theta(n^2)$$

Merge sort

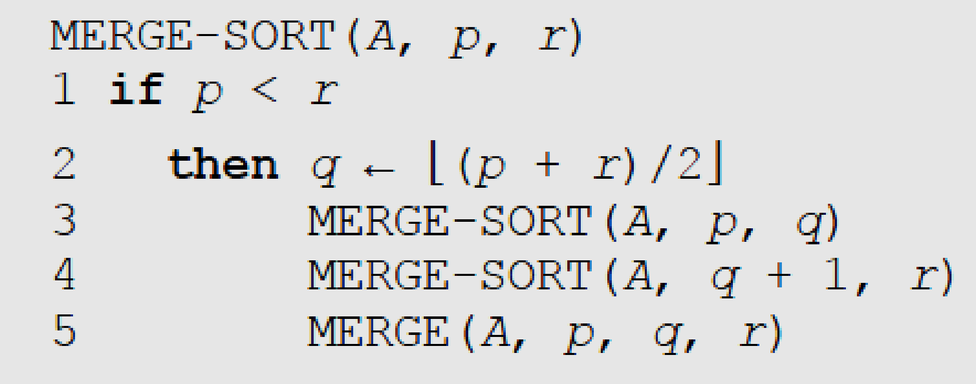
Insertion Sort

- Naive Insertion Sort
- In-Place Insertion Sort
- Insertion Sort Runtime

Merge sort

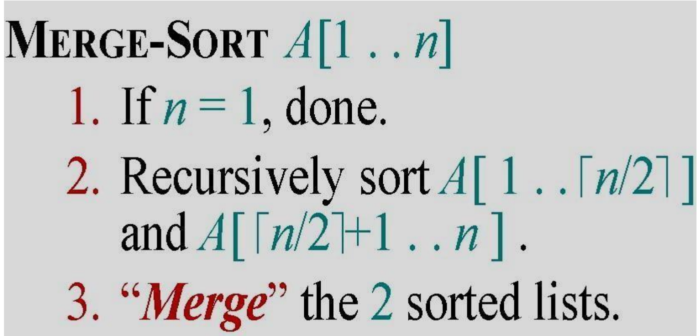
- **Analyzing Merge sort**
- Merge Sort Runtime

Merge sort



```
MERGE-SORT(A, p, r)
if $p < r$
    then $q \leftarrow \lfloor (p + r)/2 \rfloor$
        MERGE-SORT(A, p, q)
        MERGE-SORT(A, q + 1, r)
        MERGE(A, p, q, r)
```

May be better understood as follows:



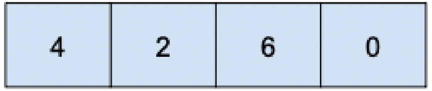
MERGE-SORT $A[1\,..\,n]$

1. If $n = 1$, done.
2. Recursively sort $A[\,1\,..\lceil n/2\rceil\,]$ and $A[\,\lceil n/2\rceil{+}1\,..\,n\,]$.
3. "*Merge*" the 2 sorted lists.

Branching continues until p<r becomes FALSE

Merge Sort

Merge sort splits the list in half, applies merge sort to each half, and then merges the two halves together in a zipper fashion.

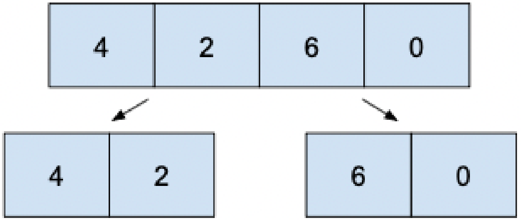


4	2	6	0

Runtime: Θ(NlogN)

Merge Sort

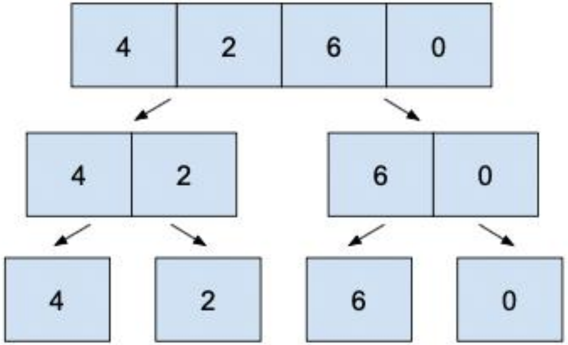
Merge sort splits the list in half, applies merge sort to each half, and then merges the two halves together in a zipper fashion.



Runtime: Θ(NlogN)

Merge Sort

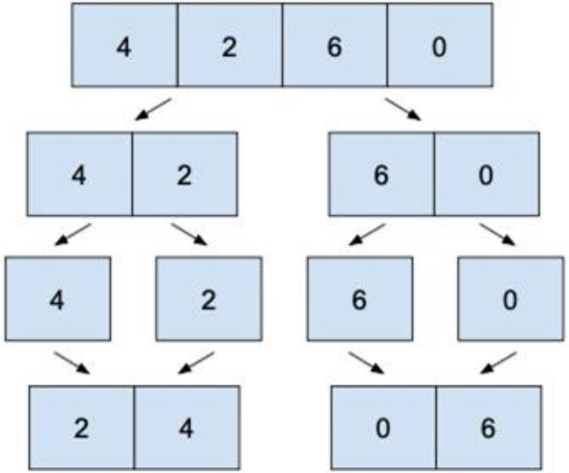
Merge sort splits the list in half, applies merge sort to each half, and then merges the two halves together in a zipper fashion.



Runtime: Θ(NlogN)

Merge Sort

Merge sort splits the list in half, applies merge sort to each half, and then merges the two halves together in a zipper fashion.

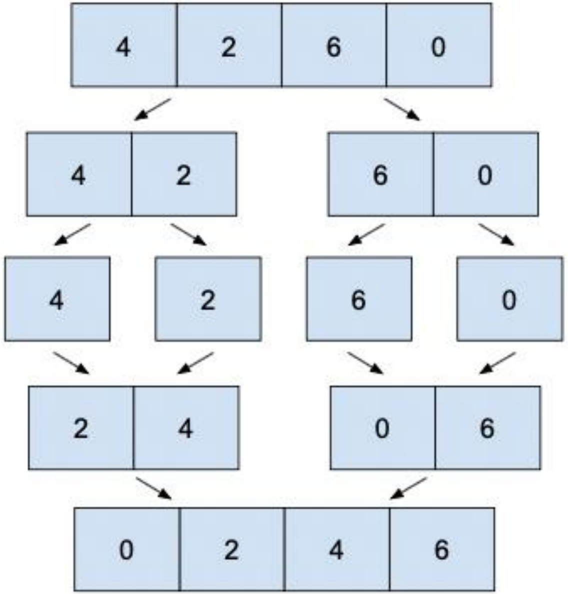


Runtime: Θ(NlogN)

Merge Sort

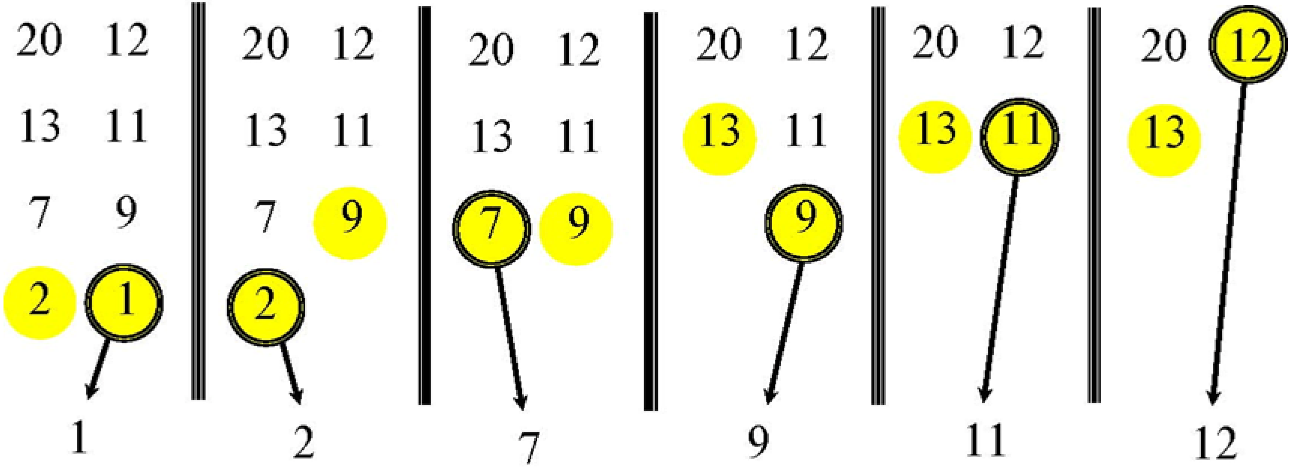
Merge sort splits the list in half, applies merge sort to each half, and then merges the two halves together in a zipper fashion.

Runtime: Θ(NlogN)

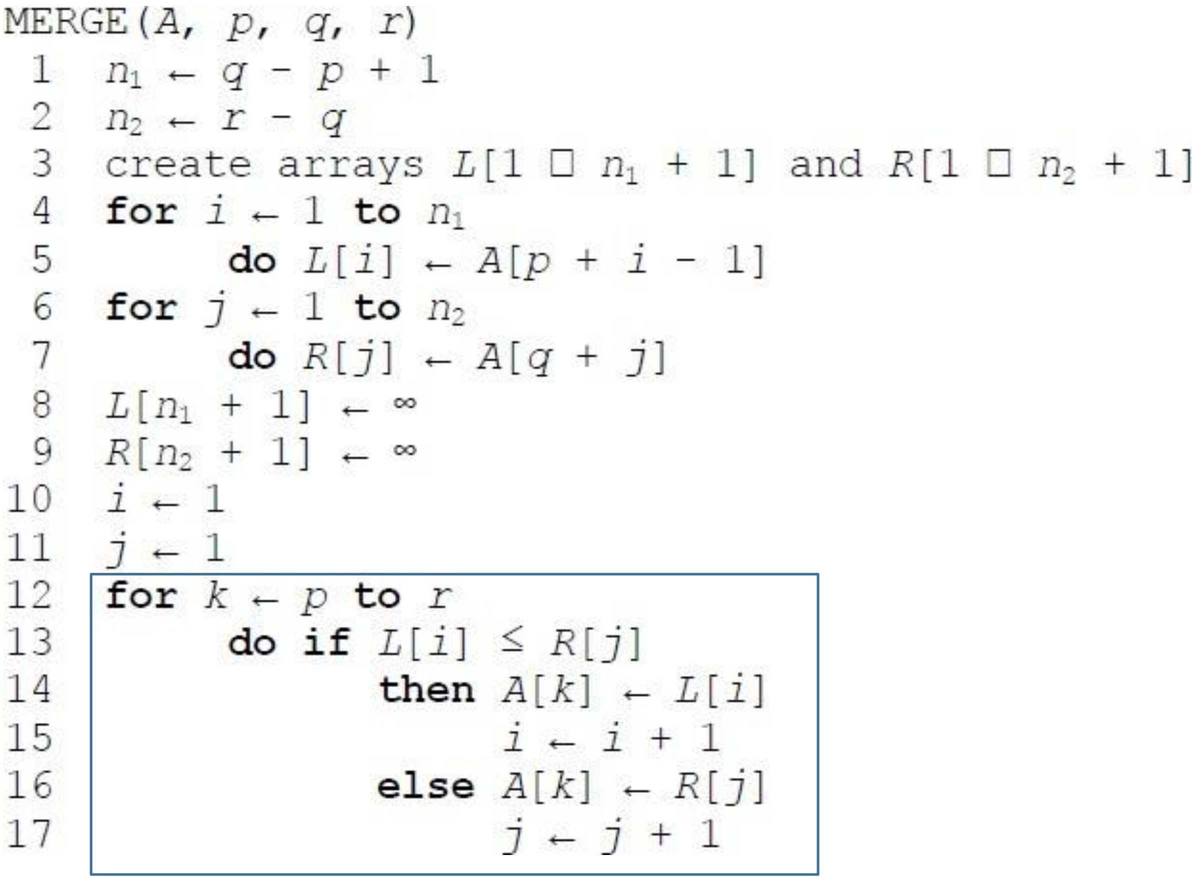


Merging two sorted arrays

- Time = Θ(n) to merge a total of n elements (linear time).



Pseudo-code for Merging



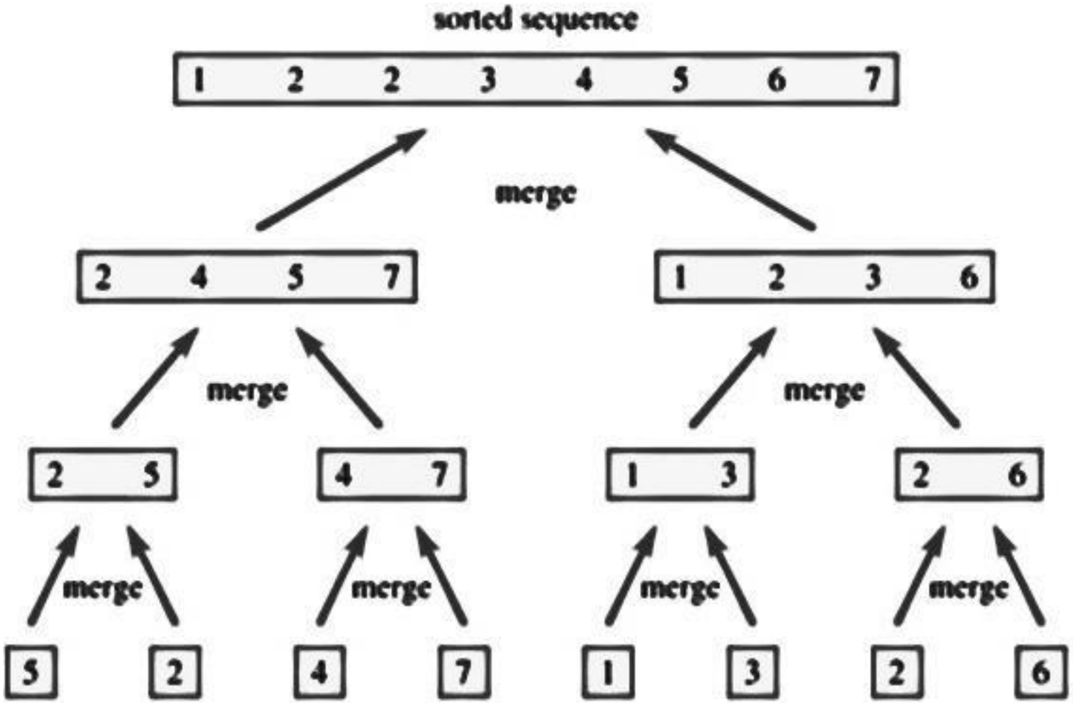
```
MERGE($A$, $p$, $q$, $r$)
$n_1 \leftarrow q - p + 1$
$n_2 \leftarrow r - q$
create arrays $L[1$ □ $n_1 + 1]$ and $R[1$ □ $n_2 + 1]$
for $i \leftarrow 1$ to $n_1$
        do $L[i] \leftarrow A[p + i - 1]$
for $j \leftarrow 1$ to $n_2$
        do $R[j] \leftarrow A[q + j]$
$L[n_1 + 1] \leftarrow \infty$
$R[n_2 + 1] \leftarrow \infty$
$i \leftarrow 1$
$j \leftarrow 1$
for $k \leftarrow p$ to $r$
        do if $L[i] \le R[j]$
              then $A[k] \leftarrow L[i]$
                   $i \leftarrow i + 1$
              else $A[k] \leftarrow R[j]$
                   $j \leftarrow j + 1$
```

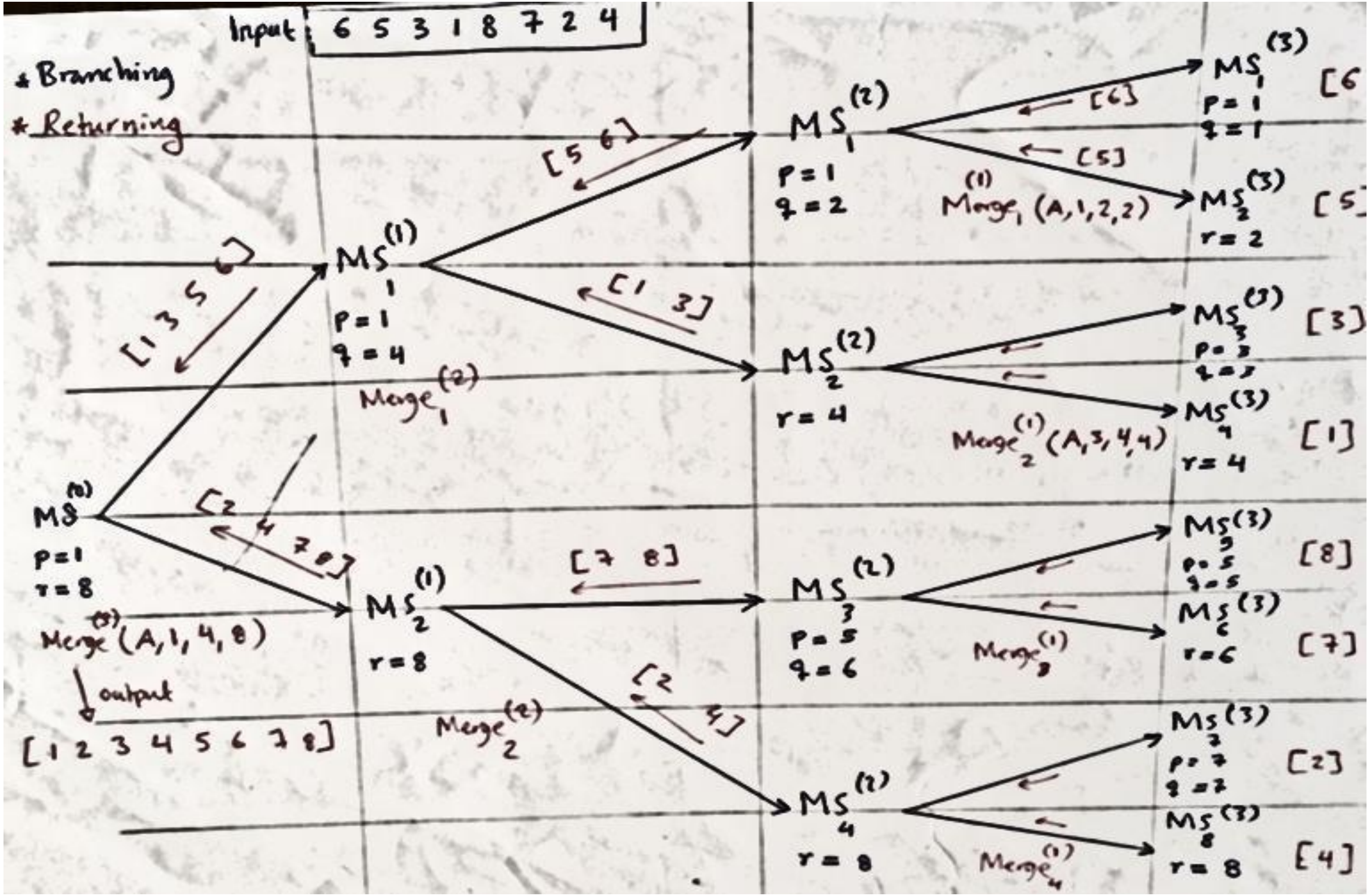
Pseudo-code for Merging

- The procedure assumes that the subarrays A[p…q] and A[q + 1…r] are in sorted order.
- Output: a single sorted subarray that replaces the current subarray A[p…r].
- MERGE procedure takes time Θ(n), where n = r - p + 1
- Using ∞ just to copy all elements of the other array to the output one. (we can get ride of it)
- Lines from 12 to 17 do the task.

Returning-back

until the initial copy of Merge-Sort()




Input: 6 5 3 1 8 7 2 4
* Branching
* Returning
MS(0) p=1 r=8
Merge(3)(A,1,4,8)
output
[1 2 3 4 5 6 7 8]
[1 3 5 6]
[2 4 7 8]
MS1(1) p=1 q=4
Merge1(2)
MS2(1) r=8
Merge2(2)
[5 6]
[1 3]
[7 8]
[2 4]
MS1(2) p=1 q=2
Merge1(1)(A,1,2,2)
MS2(2) r=4
Merge2(1)(A,3,4,4)
MS3(2) p=5 q=6
Merge3(1)
MS4(2) r=8
Merge4(1)
[6]
[5]
MS1(3) p=1 q=1
MS2(3) r=2
MS3(3) p=3 q=3
MS4(3) r=4
MS5(3) p=5 q=5
MS6(3) r=6
MS7(3) p=7 q=7
MS8(3) r=8
[6
[5
[3]
[1]
[8]
[7]
[2]
[4]

Divide-and-conquer approach

- Merge sort belongs to this family.

1. Divide the problem into a number of subproblems.
2. Conquer the subproblems by solving them recursively.
3. Combine the solutions to the subproblems into the solution for the original problem.

- The main advantage is to avoid comparing all elements with each other. If you notice well, you can see that some elements aren’t compared to other elements in the merging step.

Merge sort

Insertion Sort

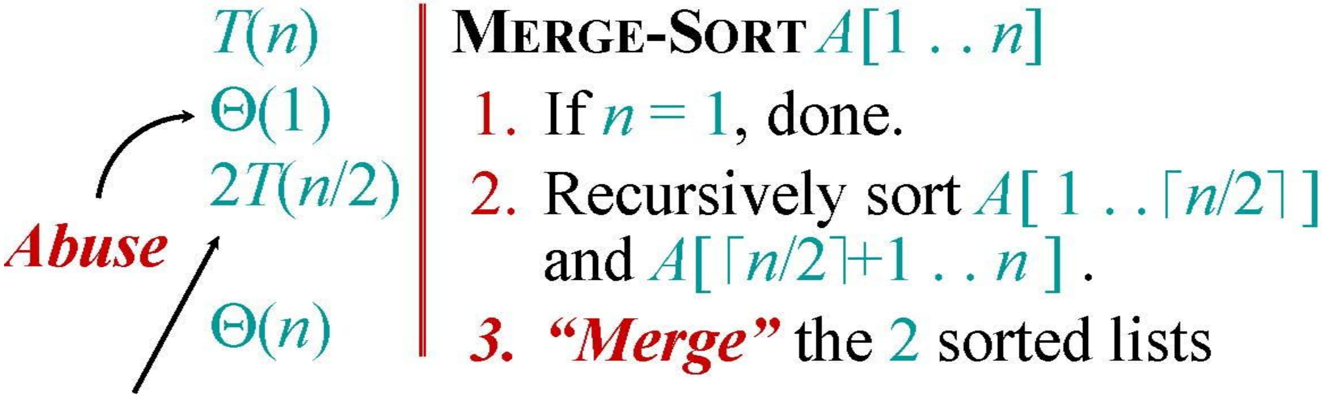
- Naive Insertion Sort
- In-Place Insertion Sort
- Insertion Sort Runtime

Merge sort

- Analyzing Merge sort
- **Merge Sort Runtime**

Analyzing Merge Sort

- **Sloppiness:** Should be T(⌈n/2⌉) + T(⌊n/2⌋) , but it turns out not to matter asymptotically. (That means that the array is recursively divided into two equal parts changing between odd and even sizes).


$T(n)$
$\Theta(1)$
$2T(n/2)$
Abuse
$\Theta(n)$

```
MERGE-SORT $A[1 \,..\, n]$
  1. If $n = 1$, done.
  2. Recursively sort $A[\,1 \,..\, \lceil n/2 \rceil\,]$
     and $A[\,\lceil n/2 \rceil + 1 \,..\, n\,]$ .
  3. "Merge" the 2 sorted lists
```

Recurrence for merge sort

- We shall usually omit stating the base case when T(n) = Θ(1) for sufficiently small n, but only when it has no effect on the asymptotic solution to the recurrence.

- There are several ways to find a good upper bound on T(n), (recurrence for example)

$$T(n) = \begin{cases} \Theta(1) \text{ if } n = 1; \\ 2T(n/2) + \Theta(n) \text{ if } n > 1. \end{cases}$$

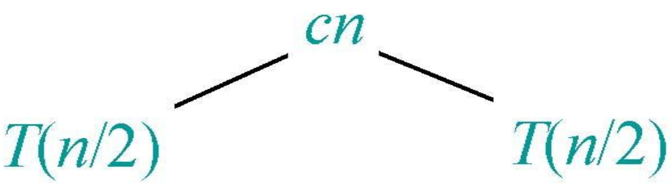
Recursion Tree

- Solve T(n) = 2T(n/2) + cn, where c > 0is constant.

$$T(n)$$

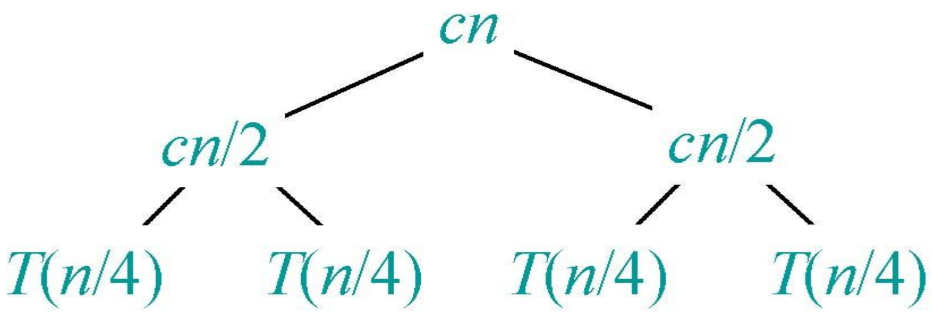
Recursion Tree

- Solve T(n) = 2T(n/2) + cn, where c > 0is constant.



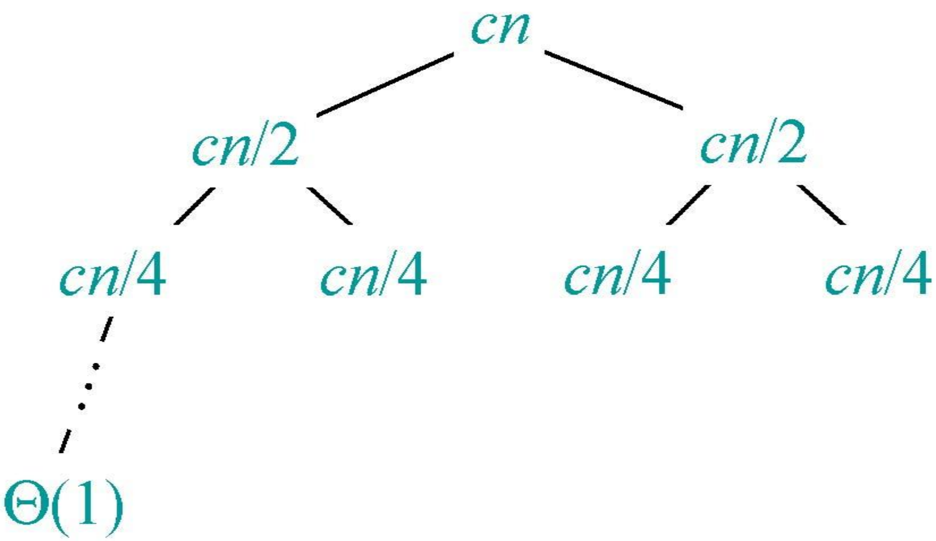
Recursion Tree

- Solve T(n) = 2T(n/2) + cn, where c > 0is constant.



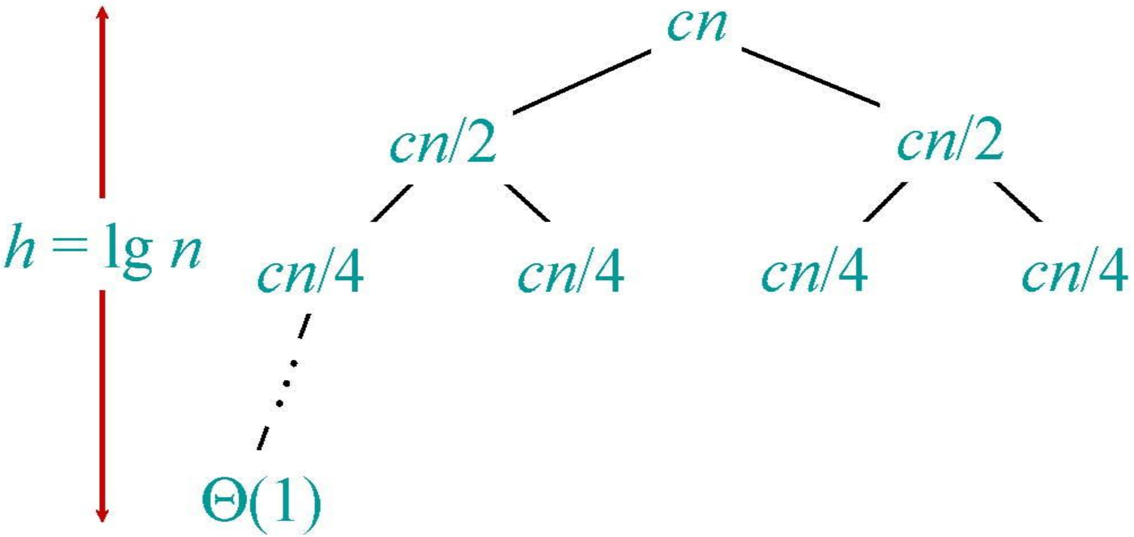
Recursion Tree

- Solve T(n) = 2T(n/2) + cn, where c > 0is constant.



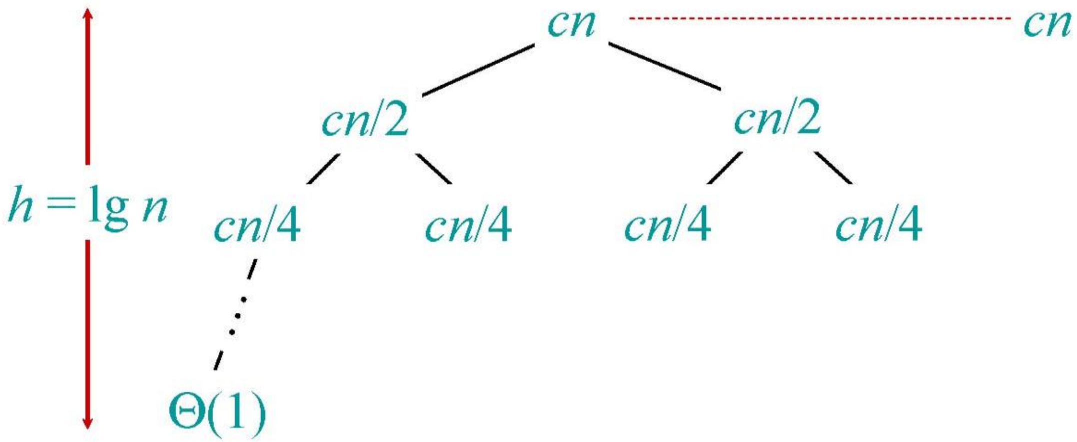
Recursion Tree

- Solve T(n) = 2T(n/2) + cn, where c > 0is constant.



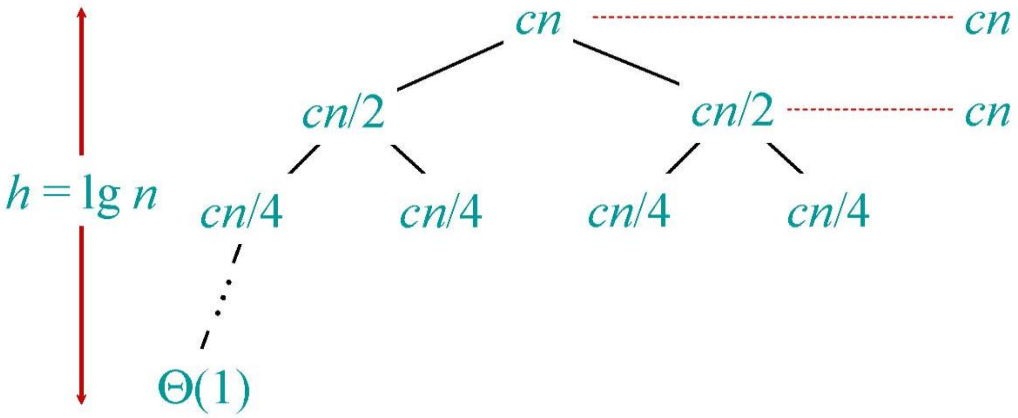
Recursion Tree

- Solve T(n) = 2T(n/2) + cn, where c > 0is constant.



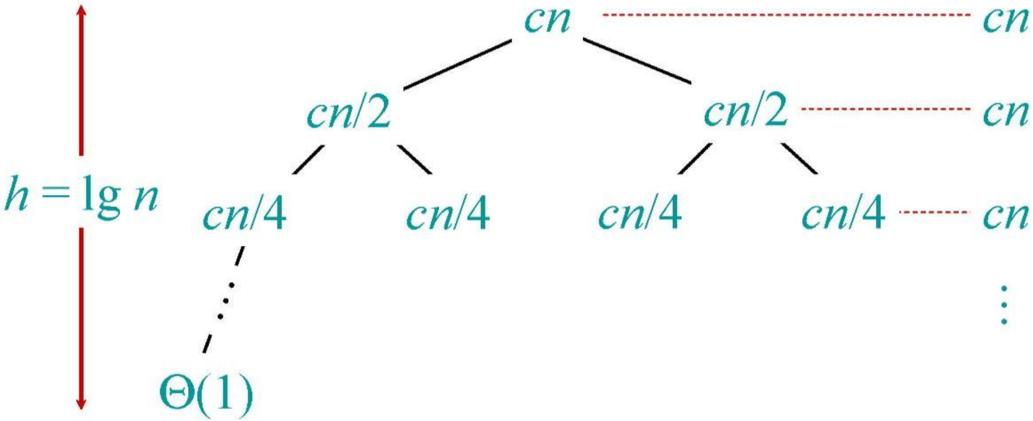
Recursion Tree

- Solve T(n) = 2T(n/2) + cn, where c > 0is constant.



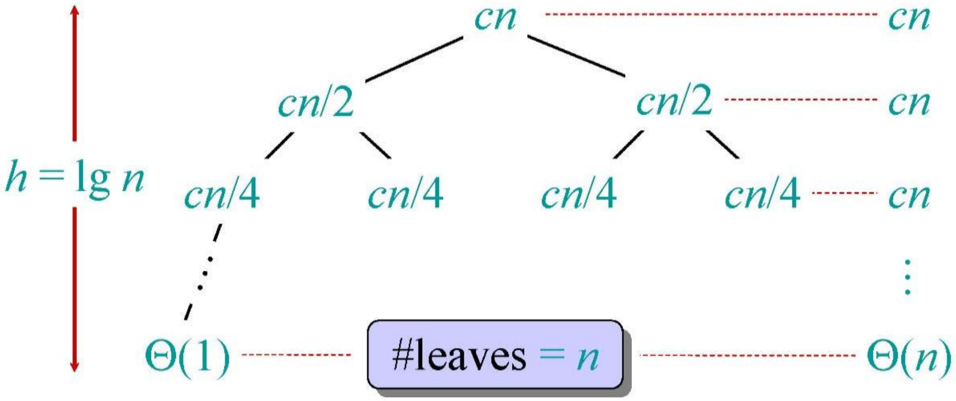
Recursion Tree

- Solve T(n) = 2T(n/2) + cn, where c > 0is constant.



Recursion Tree

- Solve T(n) = 2T(n/2) + cn, where c > 0is constant.



Merge sort

- Merge sort running time is Θ(n lg n).
- Θ(n lg n)grows more slowly than Θ(n^2).
- Therefore, merge sort asymptotically beats insertion sort in the worst case.
- In practice, merge sort beats insertion sort for n> 30 or so.
- Try coding both algorithms:
  - Choose different sizes for n (i.e., 5, 10, 50, 100, 1000).
  - Count the number of comparisons.
  - Compare the count against theoretical results.

Insertion Sort

General strategy:

- Starting with an empty output sequence.
- Add each item from input, inserting into output at right point.

For naive approach, if output sequence contains k items, worst cost to insert a single item is k.

- Might need to move everything over.

More efficient method:

- Do everything in place using swapping.

Sorts So Far

	Best Case Runtime	Worst Case Runtime	Space	Demo	Notes
Mergesort	$\Theta(N \log N)$	$\Theta(N \log N)$	$\Theta(N)$	Link	Fastest of these.
Insertion Sort (in place)	$\Theta(N)$	$\Theta(N^2)$	$\Theta(1)$	Link	Best for small N or almost sorted.