CSE214 – Analysis of Algorithms

PhD Furkan Gözükara, Toros University

https://github.com/FurkanGozukara/Analysis-of-Algorithms-2019

Lecture 1

Introduction to Analysis of Algorithms

Based on Cevdet Aykanat's and Mustafa Ozdal's Lecture Notes - Bilkent

Logaritma Nedir? Logaritma Formülleri Özellikleri

Logaritma Tanımı: a, b ∈ R^+ ve a≠1 olmak üzere $a^x = b$ denklemini sağlayan x sayısına $\log_a b$ denir ve b'nin a tabanında logaritması diye okunur.

1) $\log_a x = b$ ise $x = a^b$ $log_2 8 = 3 \mid 8 = 2^3$

2) $\log_a(A.B) = \log_a A + \log_a B$ $log_2(4 * 8) = log_2(32) = 5 = log_2(4) + log_2(8) = 2 + 3$

3) $\log_a(A/B) = \log_a A - \log_a B$ $log_2(16/4) = log_2(4) = 2 = log_2(16) - log_2(4) = 4 - 2 = 2$

4) $\log_a A^n = n.\log_a A$ $log_2 8^2 = log_2 64 = 6 = 2 * log_2 8 = 2 * 3 = 6$

5) $\log_{a^m} A^n = \frac{n}{m} \log_a A$ $log_{2^3} 8^2 = log_8 64 = 2 = (2/3) * log_2 8 = 2/3 * 3 = 2$

6) $\log_{(a^n)} x = \frac{1}{n} .\log_a x$ $log_{2^3} 8 = log_8 8 = 1 = (1/3) * log_2 8 = 1/3 * 3 = 1$

7) $\log_a x = (\log_b x)/(\log_b a)$ [taban değiştirme] $log_4 16 = 2 = log_2 16 - log_2 4 = 4 - 2 = 2$

8) $a^{\log_a x} =$ $2^{log_2 8} = 2^3 = 8 = 8$

9) $\log_a \sqrt[n]{A} = \frac{1}{n} \log_a A$ $log_2 \sqrt[3]{8} = log_2 2 = 1 = \left(\frac{1}{3}\right) * log_2 8 = \frac{1}{3} * 3 = 1$

10) $\log_{1/a} x = -\log_a x$ $log_{1/2} 8 = -log_2 8 = -3 \mid 8 = \left(\frac{1}{2}\right)^{-3}$

11) $\log_a b.\log_b c.\log_c d = \log_a d$ $log_2 4 * log_4 16 * log_{16} 256 = 2 * 2 * 2 = 8 = log_2 256$

12) $\log_a b = 1/\log_b a$ veya $\log_a b.\log_b a = 1$

Algorithm Definition

- Algorithm: A sequence of computational steps that transform the input to the desired output
- Procedure vs. algorithm
  - *An algorithm* ***must halt within finite time*** *with the right output meanwhile procedure may not halt*

- Example:

a sequence of n numbers → Sorting Algorithm → **sorted order of input sequence**

Many Real World Applications

- Bioinformatics
  - Determine/compare DNA sequences
- Internet
  - Manage/manipulate/route data
- Information retrieval
  - Search and access information in large data
- Security
  - Encode & decode personal/financial/confidential data
- Electronic design automation
  - Minimize human effort in chip-design process

Course Objectives

- Learn basic algorithms
- Gain skills to design new algorithms
- Focus on efficient algorithms
- Design algorithms that
  - are fast
  - use as little memory as possible
  - are correct!

Outline of Lecture 1

- Study two sorting algorithms as examples
  - Insertion sort: *Incremental* algorithm
  - Merge sort: *Divide-and-conquer*
- Introduction to runtime analysis
  - Best vs. worst vs. average case
  - Asymptotic analysis

Sorting Problem

Input: Sequence of numbers

$$\langle a_1, a_2, \ldots, a_n \rangle$$

Output: A sorted list

$$\Pi = \langle \Pi(1), \Pi(2), \ldots, \Pi(n) \rangle$$

such that

$$a_{\Pi(1)} \leq a_{\Pi(2)} \leq \ldots \leq a_{\Pi(n)}$$

Insertion Sort

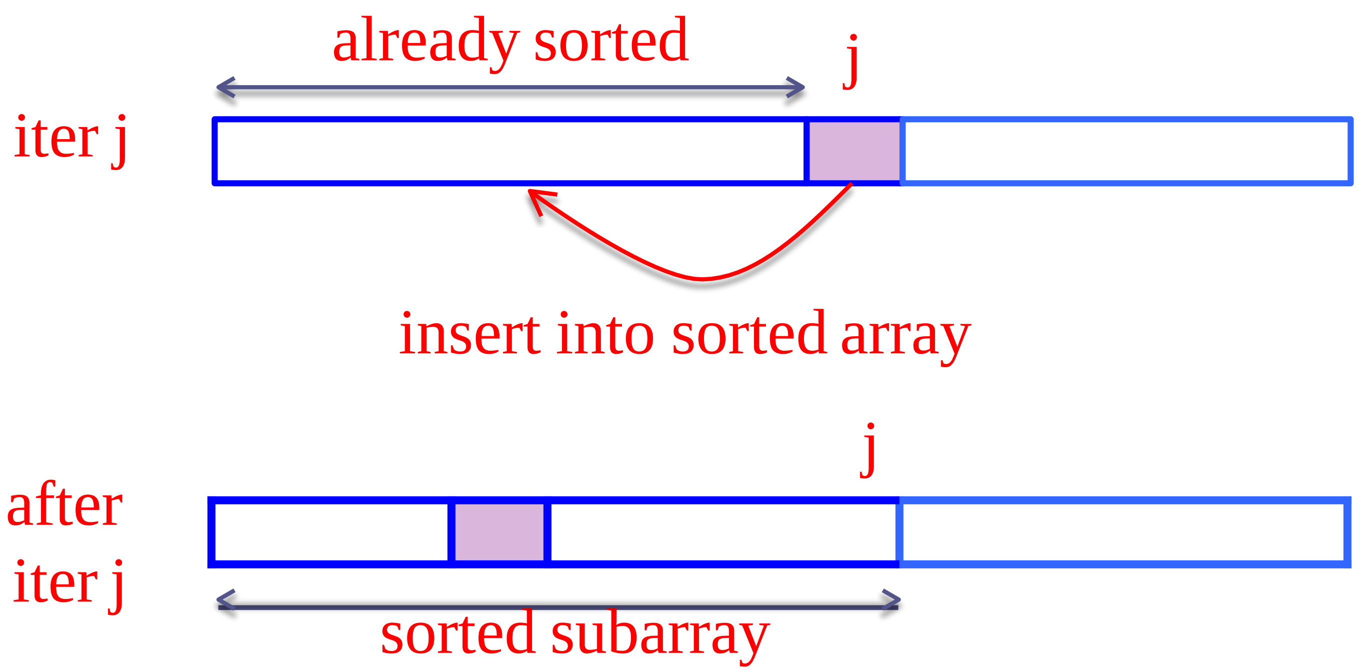
Insertion sort is an incremental algorithm

An incremental algorithm is given a sequence of input, and finds a sequence of solutions that build incrementally while adapting to the changes in the input

Incremental computation, is a software feature which, whenever a piece of data changes, attempts to save time by only recomputing those outputs which depend on the changed data

Insertion Sort: Basic Idea

- Assume input array: A[1..n]
- Iterate j from 2 to n



Pseudo-code notation

- Objective: Express algorithms to humans in a clear and concise way

- Liberal use of English

- Indentation for block structures

- Omission of error handling and other details
  → *needed in real programs*

Algorithm: Insertion Sort

Insertion-Sort (A)

```
1. for j ← 2 to n do
2.     key ← A[j];
3.     i ← j - 1;
4.     while i > 0 and A[i] > key
         do
5.         A[i+1] ← A[i];
6.         i ← i - 1;
       endwhile
7.     A[i+1] ← key;
     endfor
```

Algorithm: Insertion Sort

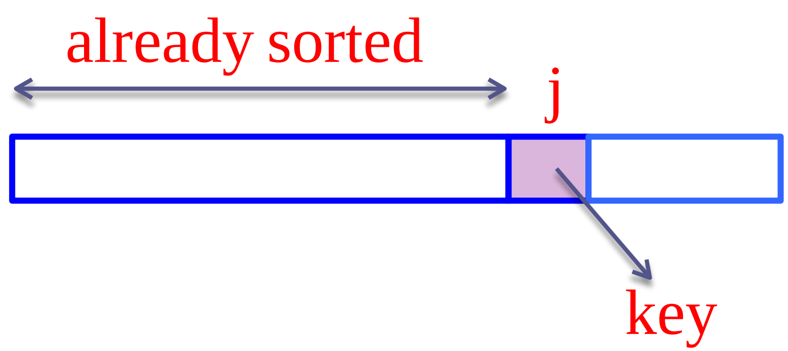
<u>Insertion-Sort</u> (A)

```
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;
    endwhile
    A[i+1] ← key;
  endfor
```

Iterate over array

<u>Loop invariant</u>:

The subarray A[1..j-1] is always sorted

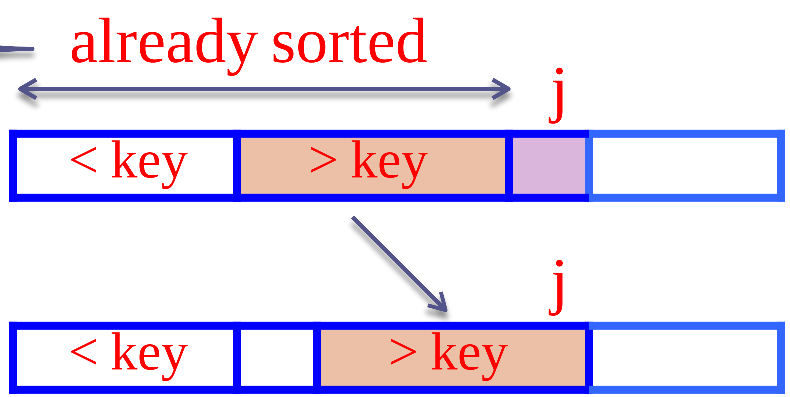


Algorithm: Insertion Sort

<u>Insertion-Sort</u> (A)

```
1. for j ← 2 to n do
2.      key ← A[j];
3.      i ← j - 1;
4. while i > 0 and A[i] > key
        do
5.          A[i+1] ← A[i];
6.          i ← i - 1;
        endwhile
7.      A[i+1] ← key;
    endfor
```

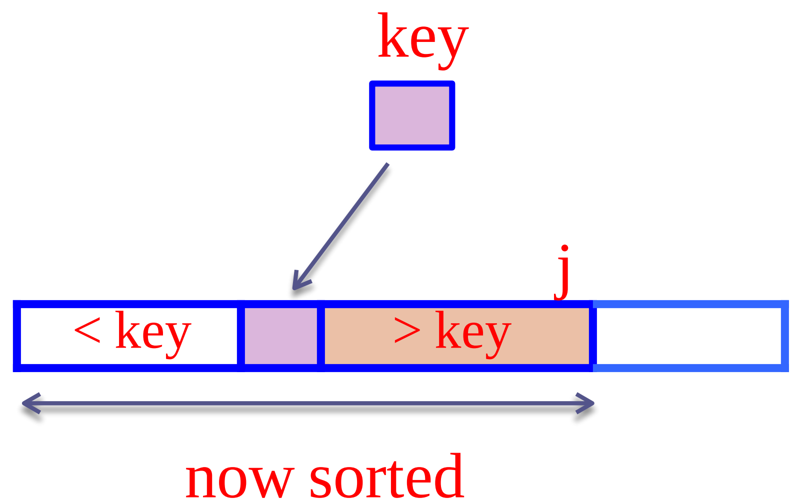
Shift right the entries in A[1..j-1] that are > key



Algorithm: Insertion Sort

Insertion-Sort (A)

```
1. for j ← 2 to n do
2.      key ← A[j];
3.      i ← j - 1;
4. while i > 0 and A[i] > key
        do
5.          A[i+1] ← A[i];
6.          i ← i - 1;
        endwhile
7.      A[i+1] ← key;
   endfor
```



Insert key to the correct location

End of iter j: A[1..j] is sorted

Insertion Sort - Example

Insertion-Sort (A)

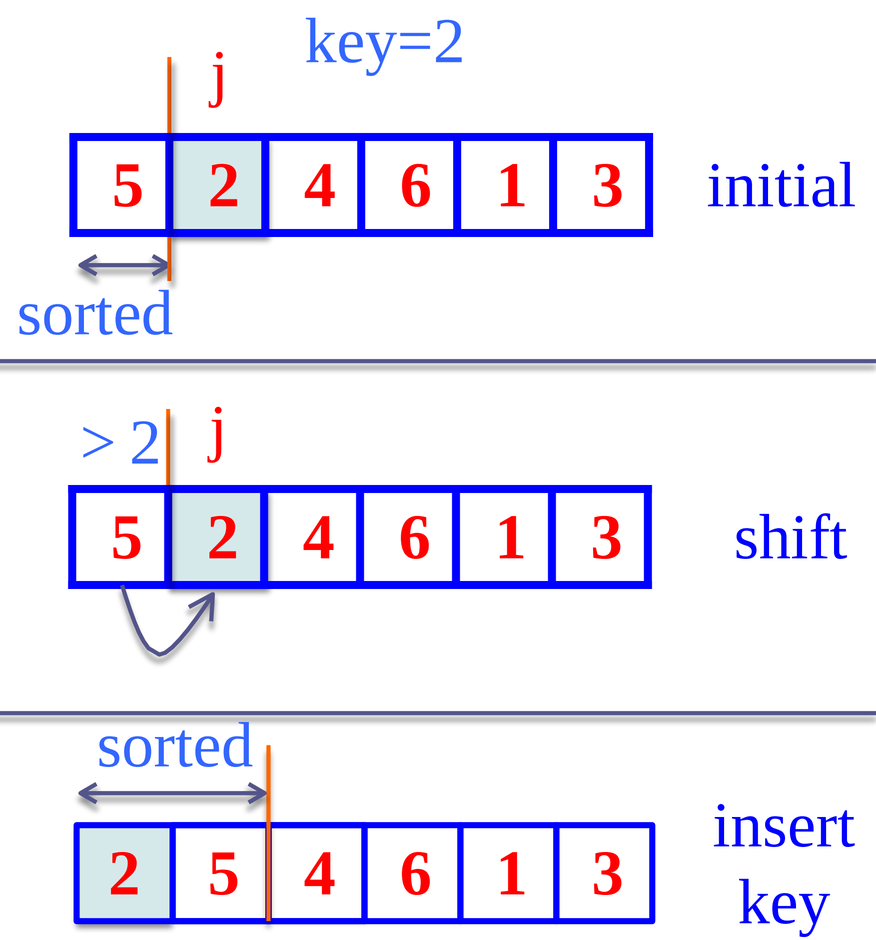
```
1. for j ← 2 to n do
2.      key ← A[j];
3.      i ← j - 1;
4. while i > 0 and A[i] > key
      do
5.          A[i+1] ← A[i];
6.          i ← i - 1;
      endwhile
7.      A[i+1] ← key;
   endfor
```

5	2	4	6	1	3

Insertion Sort - Example: Iteration j=2

Insertion-Sort (A)

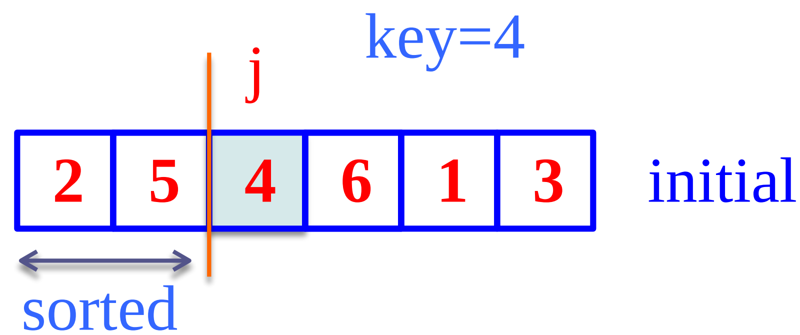
```
1. for j ← 2 to n do
2.       key ← A[j];
3.       i ← j - 1;
4. while i > 0 and A[i] > key
         do
5.             A[i+1] ← A[i];
6.             i ← i - 1;
         endwhile
7.       A[i+1] ← key;
     endfor
```



Insertion Sort - Example: Iteration j=3

Insertion-Sort (A)

```
1. for j ← 2 to n do
2.       key ← A[j];
3.       i ← j - 1;
4. while i > 0 and A[i] > key
         do
5.             A[i+1] ← A[i];
6.             i ← i - 1;
         endwhile
7.       A[i+1] ← key;
   endfor
```



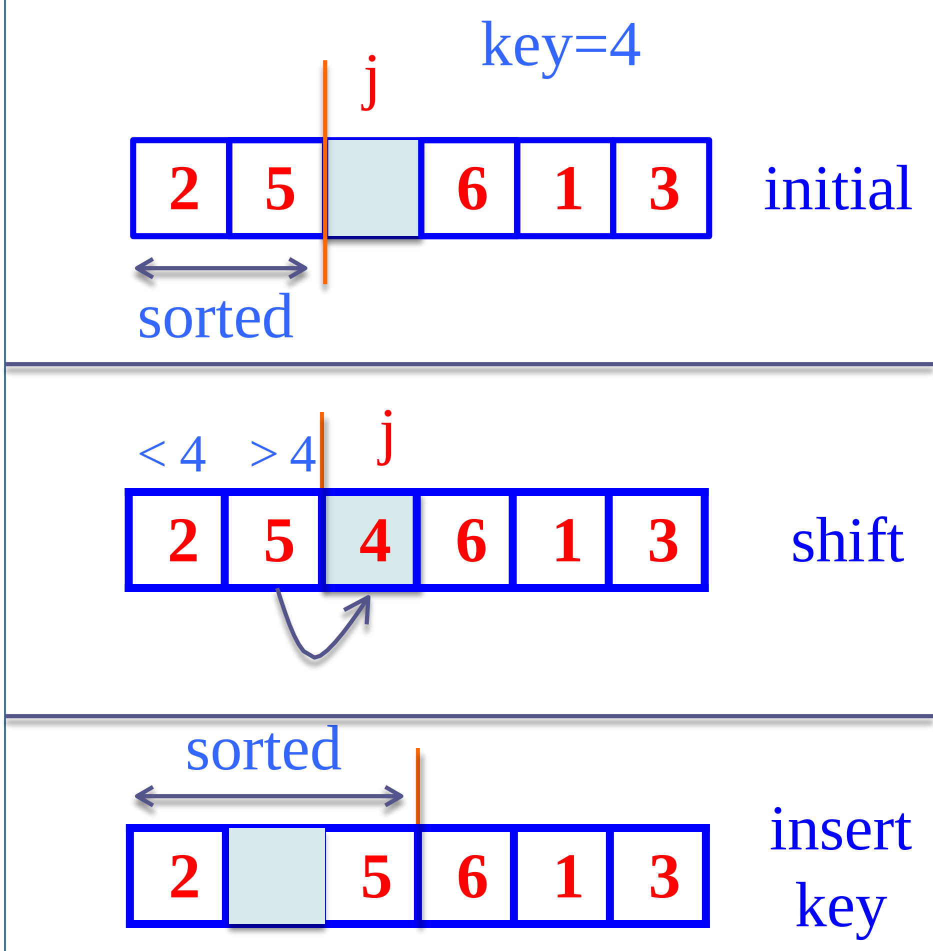
What are the entries at the end of iteration j=3?

?	?	?	?	?	?

Insertion Sort - Example: Iteration j=3

Insertion-Sort (A)

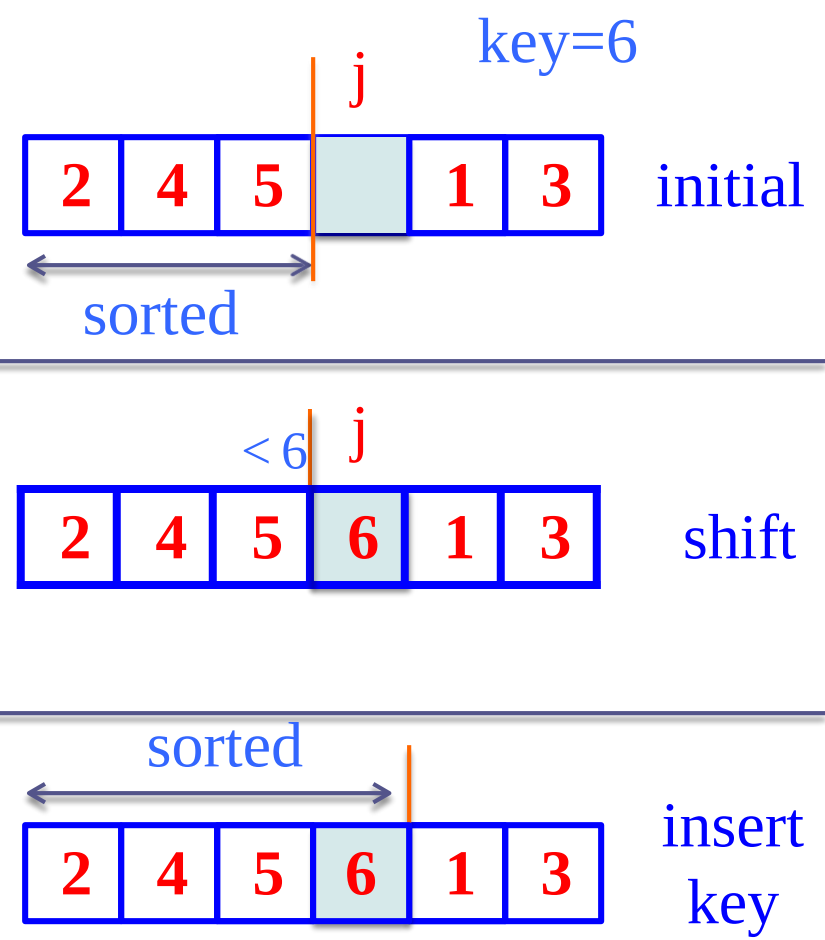
```
1. for j ← 2 to n do
2.       key ← A[j];
3.       i ← j - 1;
4. while i > 0 and A[i] > key
         do
5.             A[i+1] ← A[i];
6.             i ← i - 1;
         endwhile
7.       A[i+1] ← key;
     endfor
```



Insertion Sort - Example: Iteration j=4

Insertion-Sort (A)

```
1. for j ← 2 to n do
2.       key ← A[j];
3.       i ← j - 1;
4. while i > 0 and A[i] > key
         do
5.            A[i+1] ← A[i];
6.            i ← i - 1;
         endwhile
7.       A[i+1] ← key;
   endfor
```



Insertion Sort - Example: Iteration j=5

Insertion-Sort (A)

```
1. for j ← 2 to n do
2.      key ← A[j];
3.      i ← j - 1;
4. while i > 0 and A[i] > key
        do
5.          A[i+1] ← A[i];
6.          i ← i - 1;
        endwhile
7.      A[i+1] ← key;
    endfor
```

key=1

j

2	**4**	**5**	**6**	**1**	**3**

initial

sorted

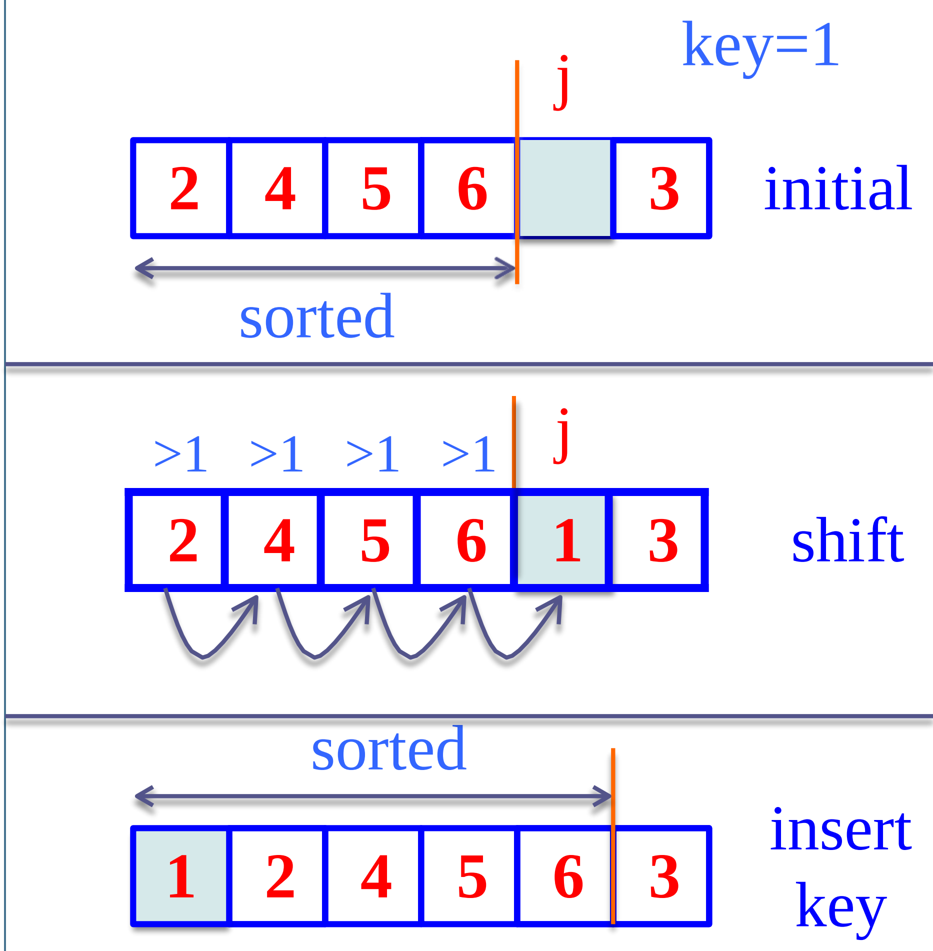
What are the entries at the end of iteration j=5?

?	**?**	**?**	**?**	**?**	**?**

Insertion Sort - Example: Iteration j=5

<u>Insertion-Sort</u> (A)

```
1. for j ← 2 to n do
2.       key ← A[j];
3.       i ← j - 1;
4. while i > 0 and A[i] > key
         do
5.             A[i+1] ← A[i];
6.             i ← i - 1;
         endwhile
7.       A[i+1] ← key;
     endfor
```



Insertion Sort - Example: Iteration j=6

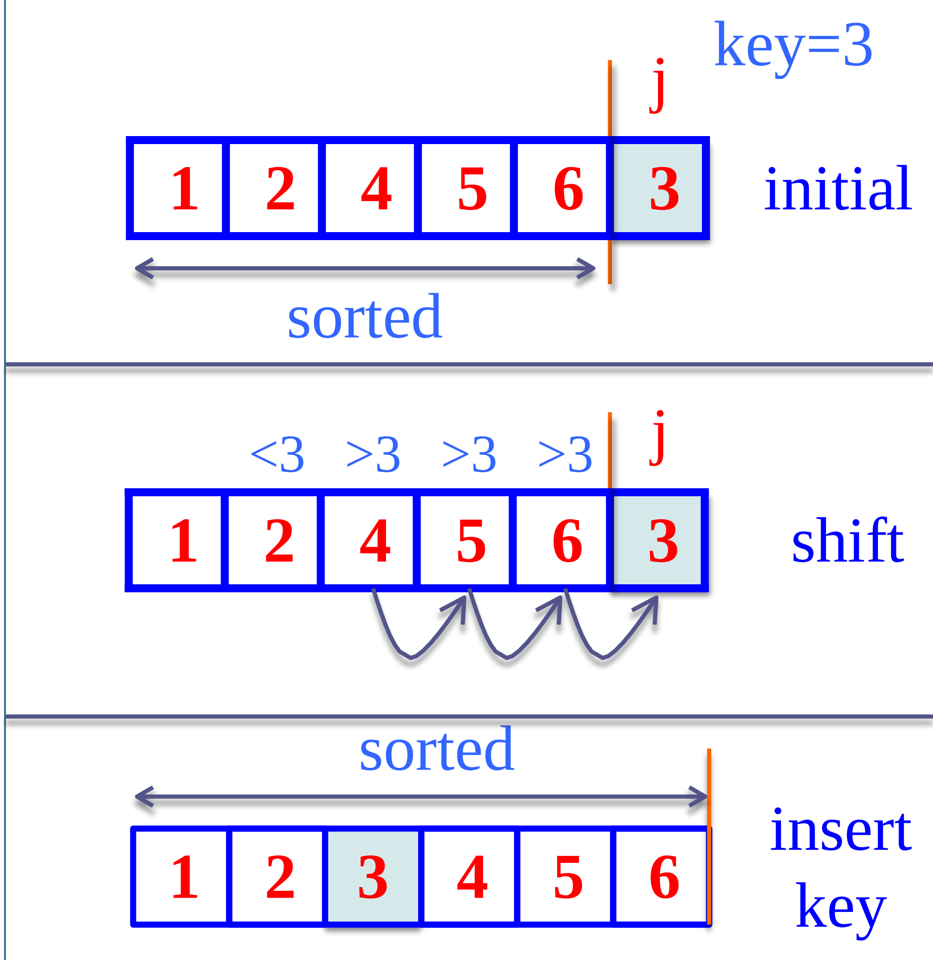
Insertion-Sort (A)

1. **for** j ← 2 **to** n **do**
2. key ← A[j];
3. i ← j - 1;
4. **while** i > 0 **and** A[i] > key **do**
5. A[i+1] ← A[i];
6. i ← i - 1;

   endwhile

7. A[i+1] ← key;

endfor



Insertion Sort Algorithm - Notes

- Items sorted in-place
  - Elements rearranged within array
  - At most constant number of items stored outside the array at any time (e.g. the variable *key*)
  - Input array A contains sorted output sequence when the algorithm ends
- Incremental approach
  - Having sorted A[1..j-1], place A[j] correctly so that A[1..j] is sorted

Running Time

- Depends on:
  - Input size (e.g., 6 elements vs 6,000,000 elements)
  - Input itself (e.g., partially sorted)

- Usually want *upper bound*

Kinds of running time analysis

- Worst Case *(Usually)*

  $T(n)$ = max time on any input of size n

- Average Case (*Sometimes*)

  $T(n)$ = average time over all inputs of size n

  Assumes statistical distribution of inputs

- Best Case (*Rarely*)

  $T(n)$ = min time on any input of size n

  BAD*: Cheat with slow algorithm that works fast on some inputs

  GOOD: Only for showing bad lower bound

*Can modify any algorithm (almost) to have a low best-case running time

- Check whether input constitutes an output at the very beginning of the algorithm

Running Time

- For Insertion-Sort, what is its worst-case time?
  - Depends on speed of primitive operations
    - Relative speed (on same machine)
    - Absolute speed (on different machines)

- Asymptotic analysis
  - Ignore machine-dependent constants
  - Look at growth of $T(n)$ as $n \to \infty$

Θ Notation

- Drop low order terms
- Ignore leading constants

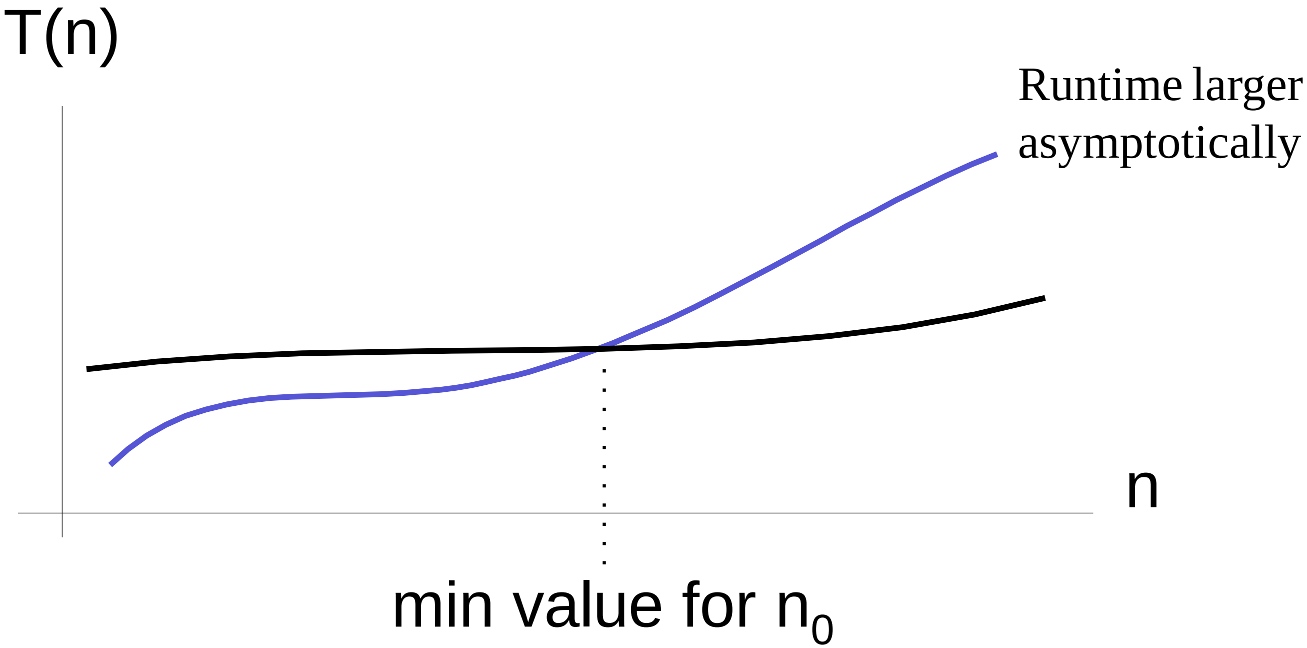
e.g.

$$2\boldsymbol{n}^2+5n+3 = \Theta(n^2)$$

$$3\boldsymbol{n}^3+90n^2-2n+5 = \Theta(n^3)$$

- *Formal explanations in the next lecture.*

- As n gets large, a $\Theta(n^2)$ algorithm runs faster than a $\Theta(n^3)$ algorithm



Insertion Sort – Runtime Analysis

Cost	Insertion-Sort (A)
c_1	1. **for** j ← 2 **to** n **do**
c_2	2. key ← A[j];
c_3	3. i ← j - 1;
c_4	4. **while** i >= 0 **and** A[i] > key **do**
c_5	5. A[i+1] ← A[i];
c_6	6. i ← i - 1;
	endwhile
c_7	7. A[i+1] ← key;
	endfor

t_j: The number of times while loop test is executed for j

How many times is each line executed?

# times	Insertion-Sort (A)
n	1. **for** j ← 2 **to** n **do**
$n-1$	2. key ← A[j];
$n-1$	3. i ← j - 1;
k_4	4. **while** i >= 0 **and** A[i] > key **do**
k_5	5. A[i+1] ← A[i];
k_6	6. i ← i - 1;
	endwhile
$n-1$	7. A[i+1] ← key;
	endfor

$$k_4 = \sum_{j=2}^{n} t_j$$

$$k_5 = \sum_{j=2}^{n} (t_j - 1)$$

$$k_6 = \sum_{j=2}^{n} (t_j - 1)$$

Insertion Sort – Runtime Analysis

- Sum up costs:

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{j=2}^{n} t_j +$$
$$c_5 \sum_{j=2}^{n} (t_j - 1) + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7(n-1)$$

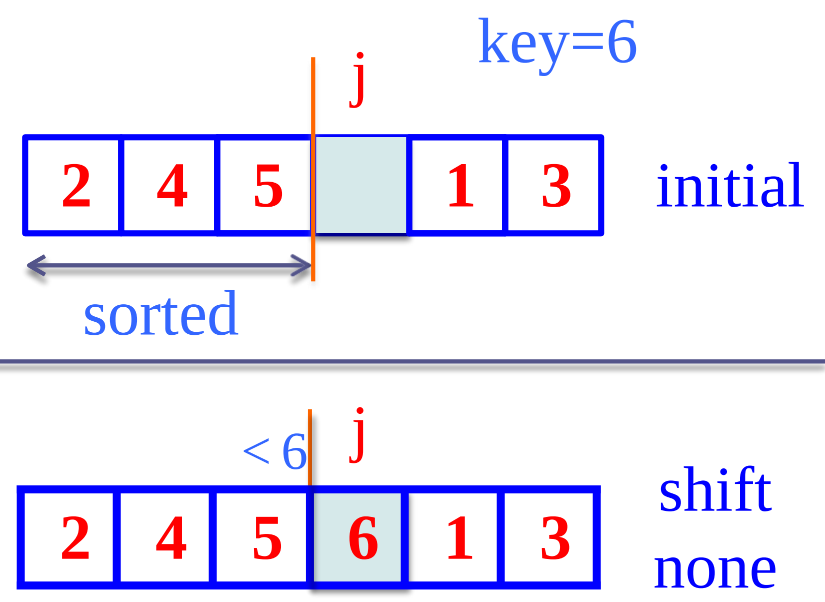
- What is the best case runtime?

- What is the worst case runtime?

Question: If A[1...j] is already sorted, t_j = ?

Insertion-Sort (A)

```
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;
    endwhile
    A[i+1] ← key;
endfor
```



$t_j = 1$

Insertion Sort – Best Case Runtime

- Original function:

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{j=2}^{n} t_j + c_5 \sum_{j=2}^{n} (t_j - 1) + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7(n-1)$$

- Best-case: Input array is already sorted

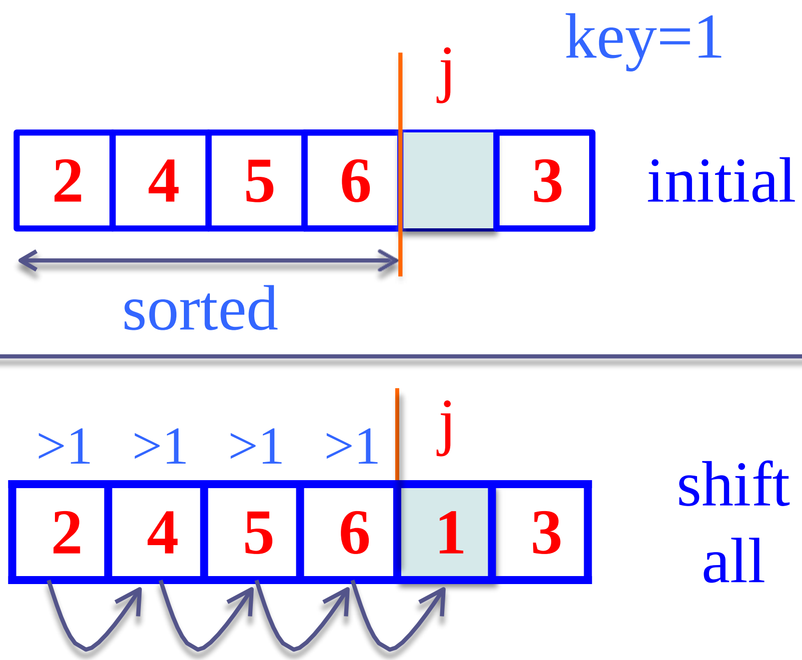
$t_j = 1$ for all j

$$T(n) = (c_1 + c_2 + c_3 + c_4 + c_7)n - (c_2 + c_3 + c_4 + c_7)$$

Q: If A[j] is smaller than every entry in A[1..j-1], $t_j = ?$

Insertion-Sort (A)

```
1. for j ← 2 to n do
2.       key ← A[j];
3.       i ← j - 1;
4. while i >= 0 and A[i] > key
         do
5.             A[i+1] ← A[i];
6.             i ← i - 1;
         endwhile
7.       A[i+1] ← key;
     endfor
```



$t_j = j$

Insertion Sort – Worst Case Runtime

- Worst case: The input array is reverse sorted

  $t_j = j$ for all j

- After derivation, worst case runtime:

$$T(n) = \tfrac{1}{2}(c_4 + c_5 + c_6)n^2 + (c_1 + c_2 + c_3 + \tfrac{1}{2}(c_4 - c_5 - c_6) + c_7)n - (c_2 + c_3 + c_4 + c_7)$$

$$1 + 2 + 3 + \ldots + n = \frac{n.(n+1)}{2}$$

Asymptotic Notation

This will be explained in further lessons

Just for now, it simply means that how our algorithm’s run time grows as the number of inputs grows to the infinity

Insertion Sort – Asymptotic Runtime Analysis

<u>Insertion-Sort</u> (A)

```
1. for j ← 2 to n do
2.      key ← A[j];                         }
3.      i ← j - 1;                          }  Θ(1)
4. while i >= 0 and A[i] > key
        do
5.          A[i+1] ← A[i];                  }
6.          i ← i - 1;                      }  Θ(1)
        endwhile
7.      A[i+1] ← key;                       }  Θ(1)
   endfor
```

Asymptotic Runtime Analysis of Insertion-Sort

- Worst-case (input reverse sorted)
  - *Inner loop is Θ(j)*

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

- Average case (all permutations equally likely)
  - *Inner loop is Θ(j/2)*

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

  - Often, average case not much better than worst case
- Is this a fast sorting algorithm?
  - Yes, for small *n*. No, for large *n*.

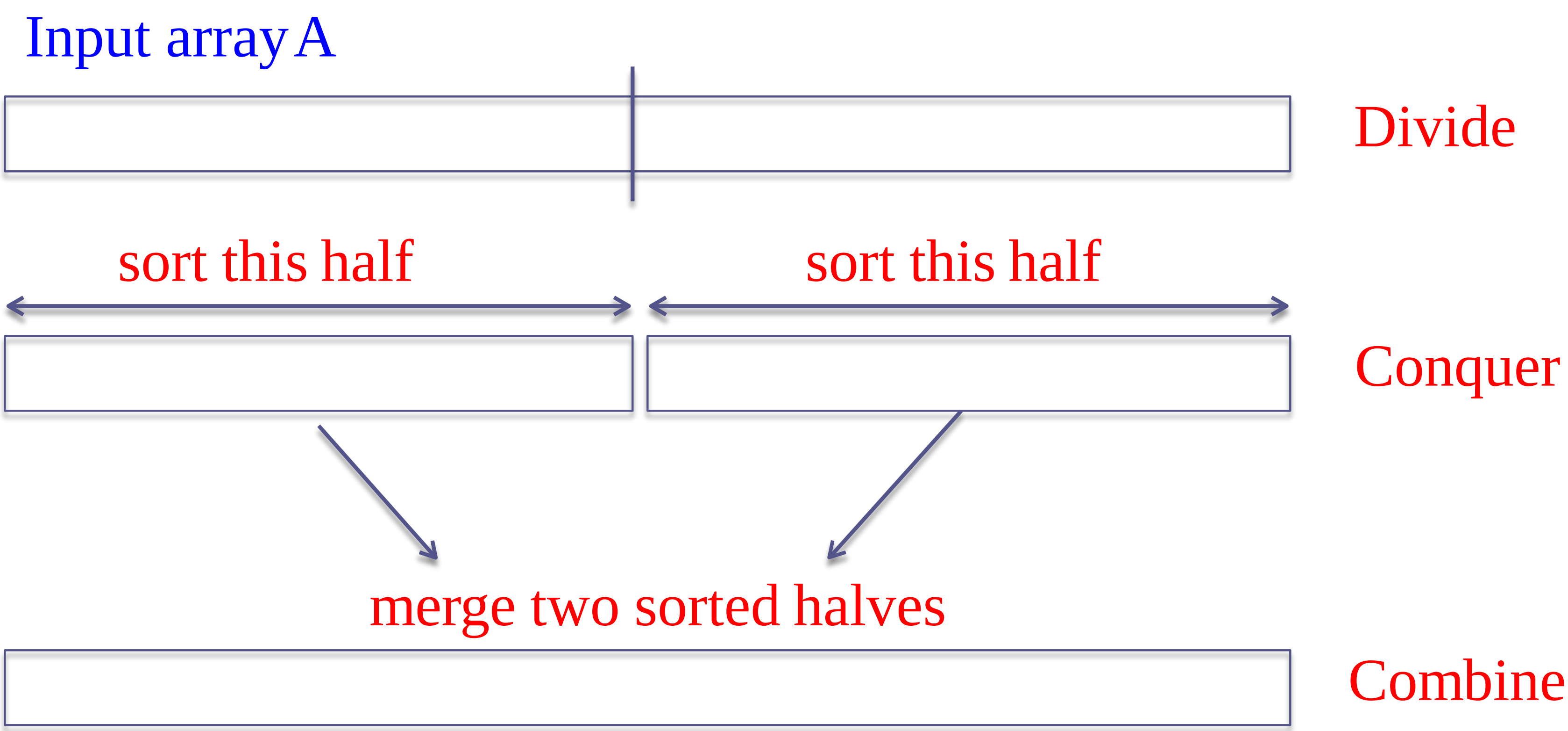
Merge Sort

Merge Sort is a divide and conquer type algorithm

Divide and Conquer basically works in three steps:

1. **Divide** - It first divides the problem into small chunks or sub-problems
2. **Conquer** - It then solve those sub-problems recursively so as to obtain a separate result for each sub-problem
3. **Combine** - It then combine the results of those sub-problems to arrive at a final result of the main problem

Merge Sort: Basic Idea



Merge-Sort (A, p, r)

if p = r **then return;**

else

q ← ⌊ (p+r)/2⌋; *(Divide)*

Merge-Sort (A, p, q); *(Conquer)*

Merge-Sort (A, q+1, r); *(Conquer)*

Merge (A, p, q, r); *(Combine)*

endif

- Call Merge-Sort(A,1,n) to sort A[1..n]
- Recursion bottoms out when subsequences have length 1

Merge Sort: Example

Merge-Sort (A, p, r)

if p = r **then**

return

else

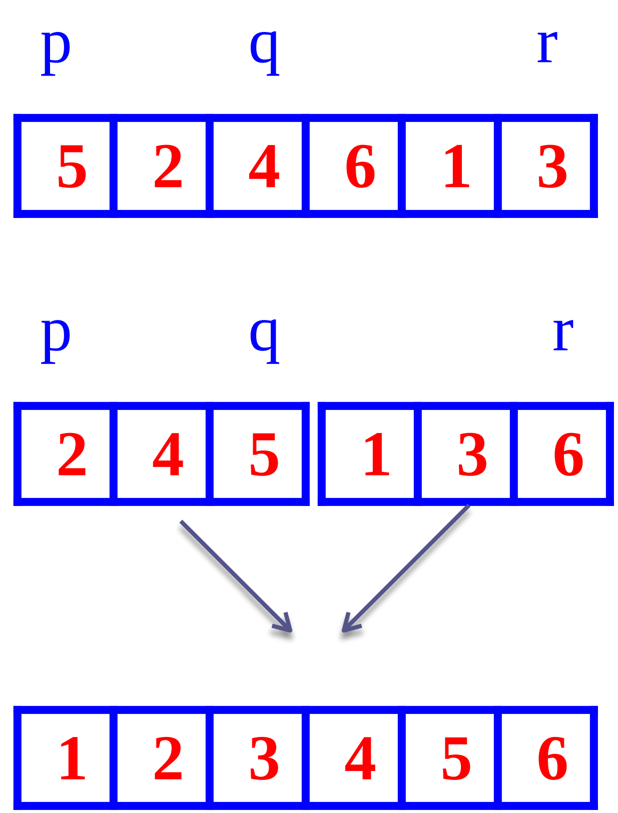
q ← ⌊(p+r)/2⌋

Merge-Sort (A, p, q)

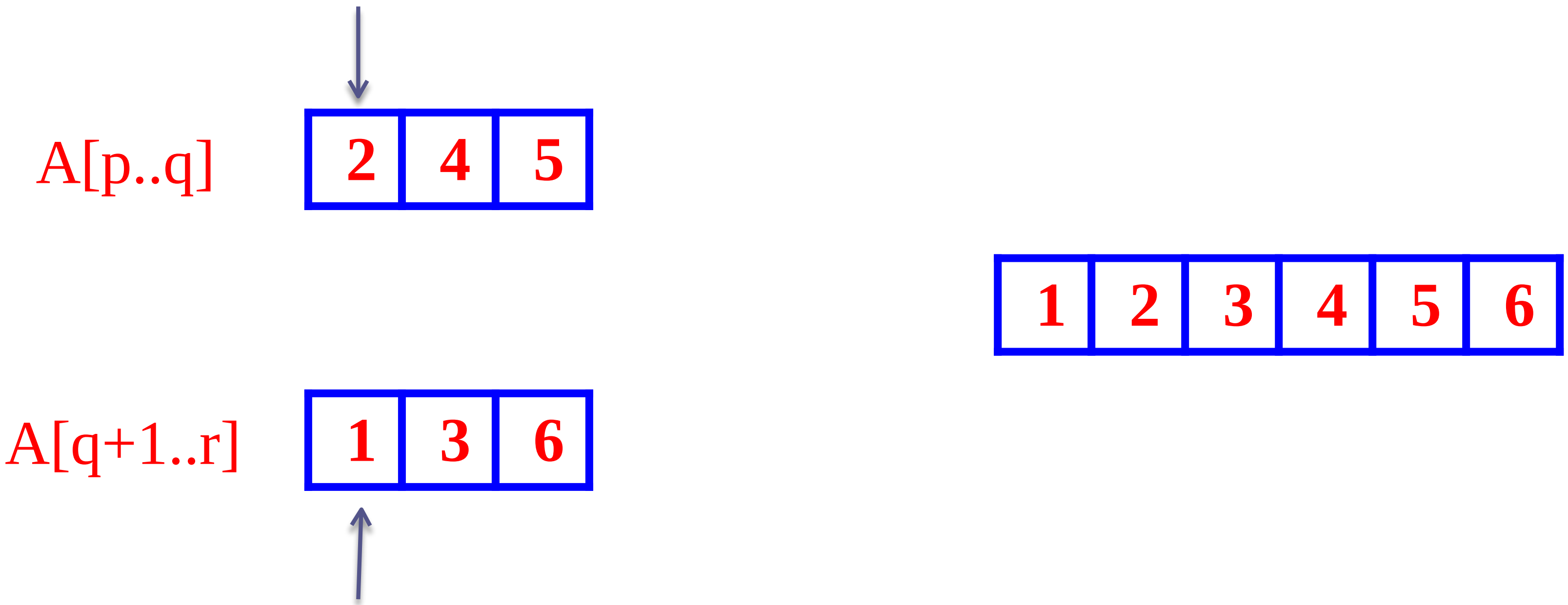
Merge-Sort (A, q+1, r)

Merge(A, p, q, r)

endif



How to merge 2 sorted subarrays?



- What is the complexity of this step? $\Theta(n)$

Merge Sort: Complexity

Merge-Sort (A, p, r)	$\rightarrow$ T(n)
if p = r **then** **return**	$\rightarrow$ $\Theta(1)$
else	
$q \leftarrow \lfloor (p+r)/2 \rfloor$	$\rightarrow$ $\Theta(1)$
Merge-Sort (A, p, q)	$\rightarrow$ T(n/2)
Merge-Sort (A, q+1, r)	$\rightarrow$ T(n/2)
Merge(A, p, q, r)	$\rightarrow$ $\Theta(n)$
endif	

Merge Sort – Recurrence

- Describe a function recursively in terms of itself
- To analyze the performance of recursive algorithms

- For merge sort:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n=1 \\ 2T(n/2) + \Theta(n) & \text{otherwise} \end{cases}$$

How to solve for T(n)?

$$T(n) = \begin{cases} \Theta(1) & \text{if } n=1 \\ 2T(n/2) + \Theta(n) & \text{otherwise} \end{cases}$$

- Generally, we will assume $T(n) = \Theta(1)$ for sufficiently small n
- The recurrence above can be rewritten as:

$$T(n) = 2\,T(n/2) + \Theta(n)$$

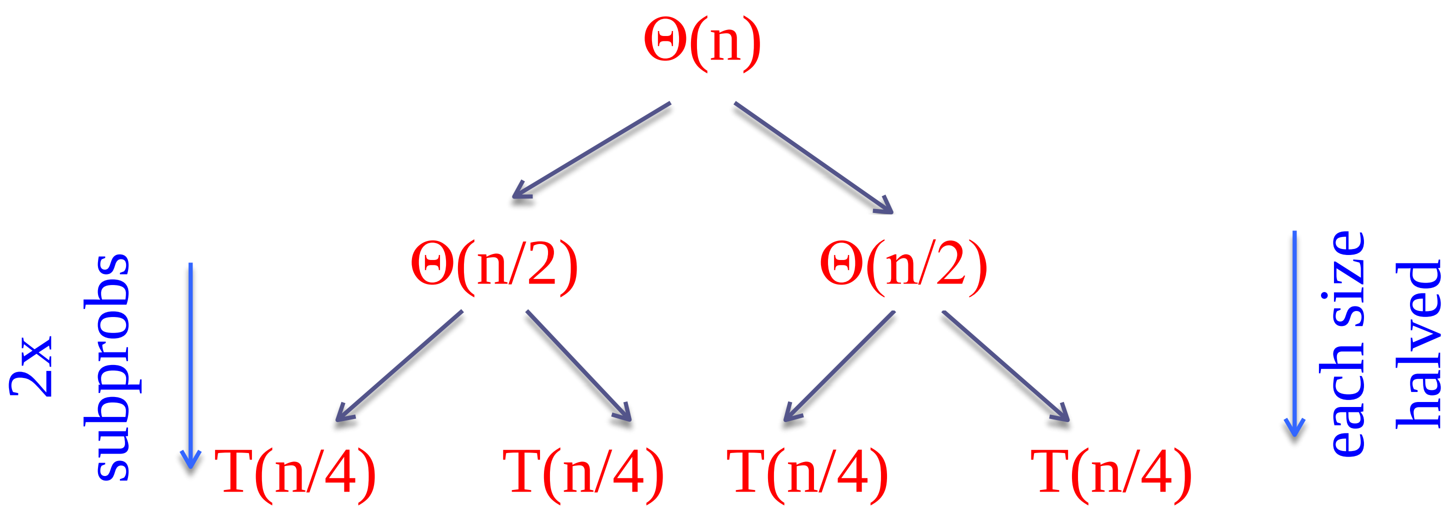
- How to solve this recurrence?

Solve Recurrence: $T(n) = 2T(n/2) + \Theta(n)$

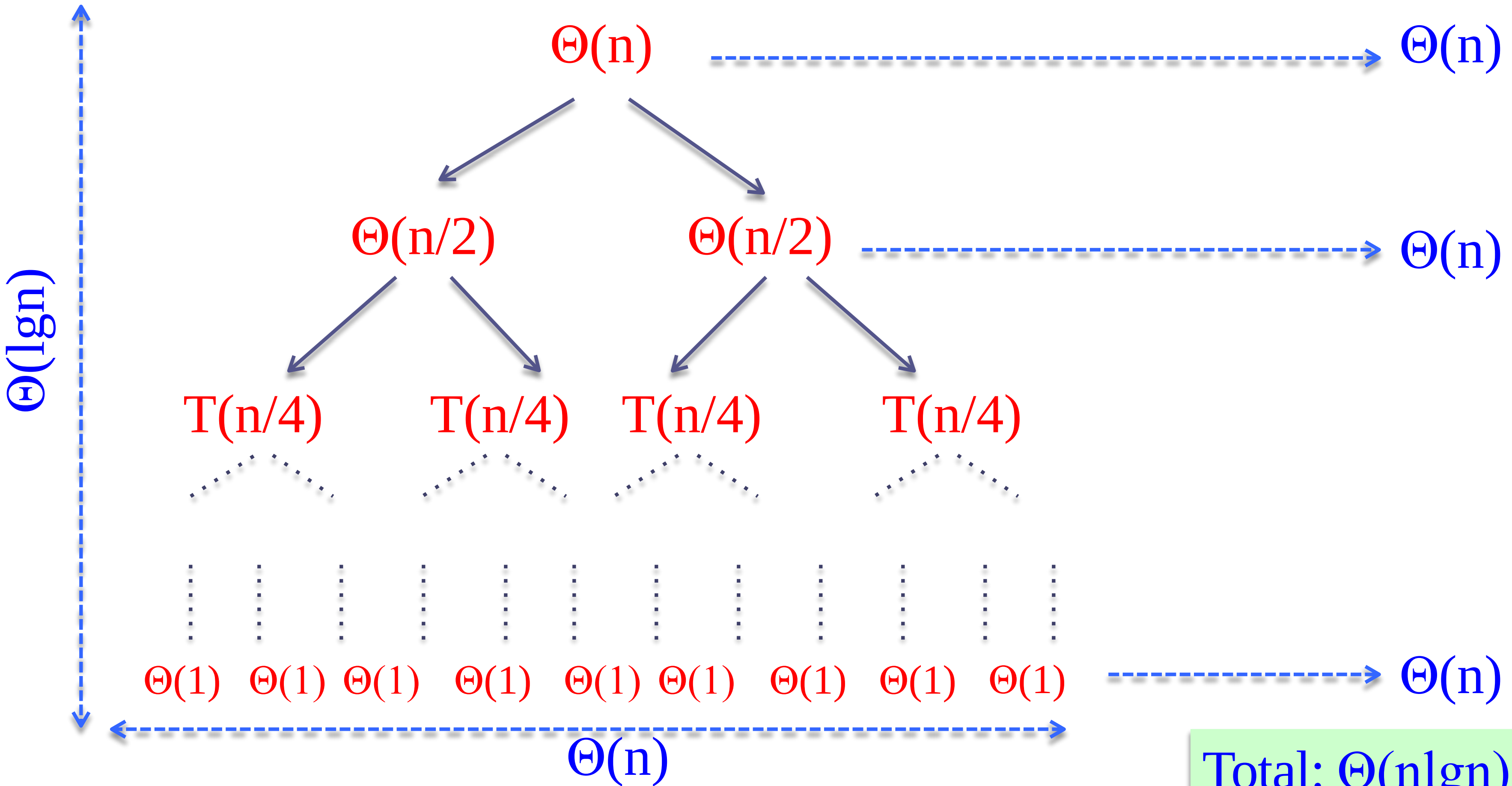
$\Theta(n)$

$T(n/2)$ $T(n/2)$

Solve Recurrence: $T(n) = 2T(n/2) + \Theta(n)$



Solve Recurrence: $T(n) = 2T(n/2) + \Theta(n)$



Total: $\Theta(n\lg n)$

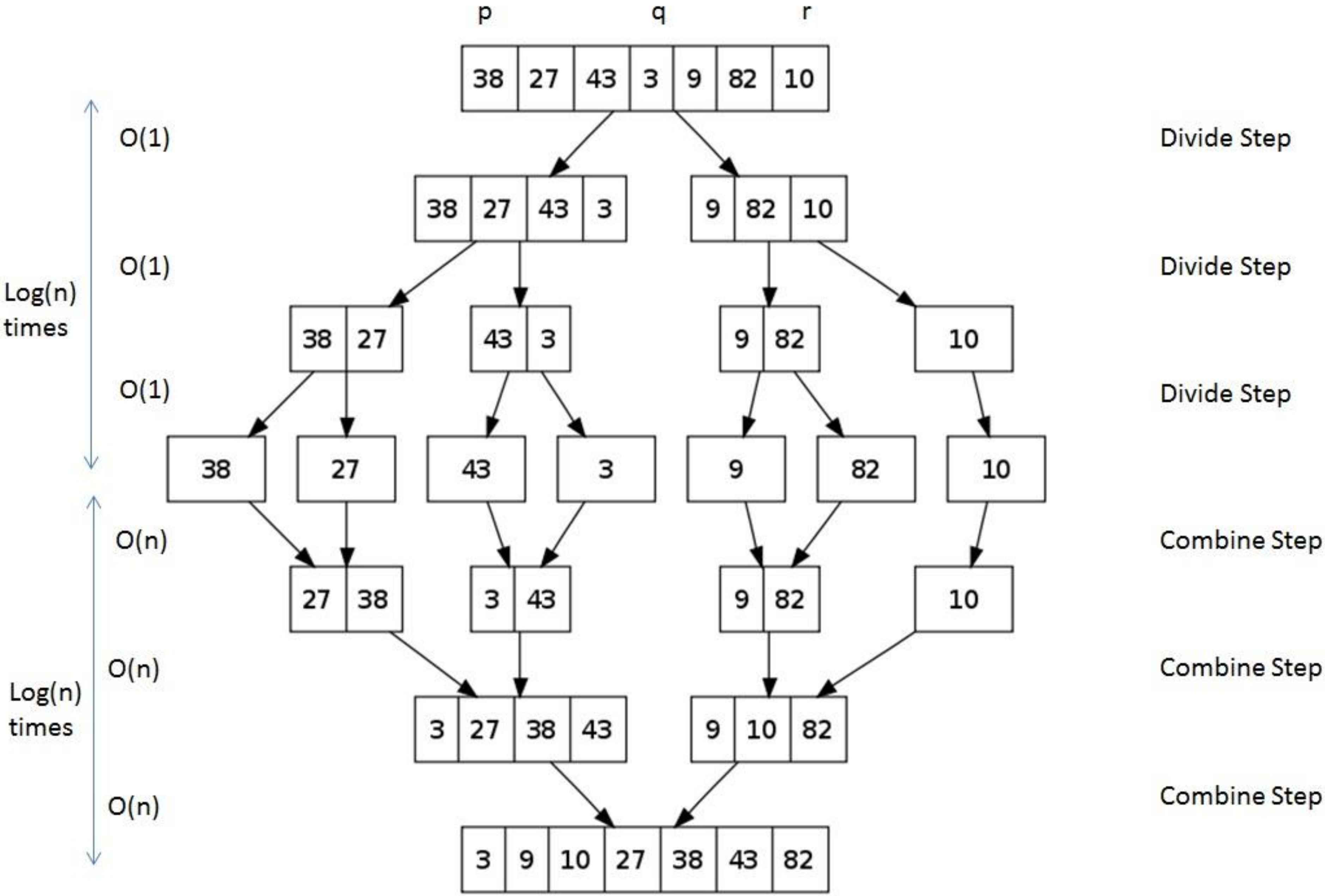
Merge Sort Complexity

- Recurrence:

  $$T(n) = 2T(n/2) + \Theta(n)$$

- Solution to recurrence:

  $$T(n) = \Theta(n\lg n)$$



Total Runtime = Total time required in Divide + Total time required in Combine

$= 1 * \text{Log}(n) + n * \text{Log}(n) = n\,\text{Log}(n).$

Conclusions: Insertion Sort vs. Merge Sort

- $\Theta(n\lg n)$ grows more slowly than $\Theta(n^2)$
- *E.g. n=1,000 > Merge sort = 1000*log(1000)= 9,965 | Insertion Sort = 1000*1000=1,000,000*
- Therefore Merge-Sort beats Insertion-Sort in the worst case
- In practice, Merge-Sort beats Insertion-Sort for $n>30$ or so.