UNIT-II

Prepared by :- Sumitra Jakhete

DIVIDE AND CONQUER AND GREEDY

UNIT-II

- Divide & Conquer: General method, Control abstraction, Merge sort, Quick Sort – Worst, Best and average case. Binary search, Large integer Multiplication, Strassen's Matrix multiplication. (for all above algorithms analysis to be done with recurrence)

- Greedy Method: General method and characteristics, Prim's method for MST , Kruskal method for MST (using nlogn complexity), Dijkstra's Algorithm, Huffman Trees (nlogn complexity), Fraction Knapsack problem, Job Sequencing

DIVIDE AND CONQUER

Divide-and-Conquer

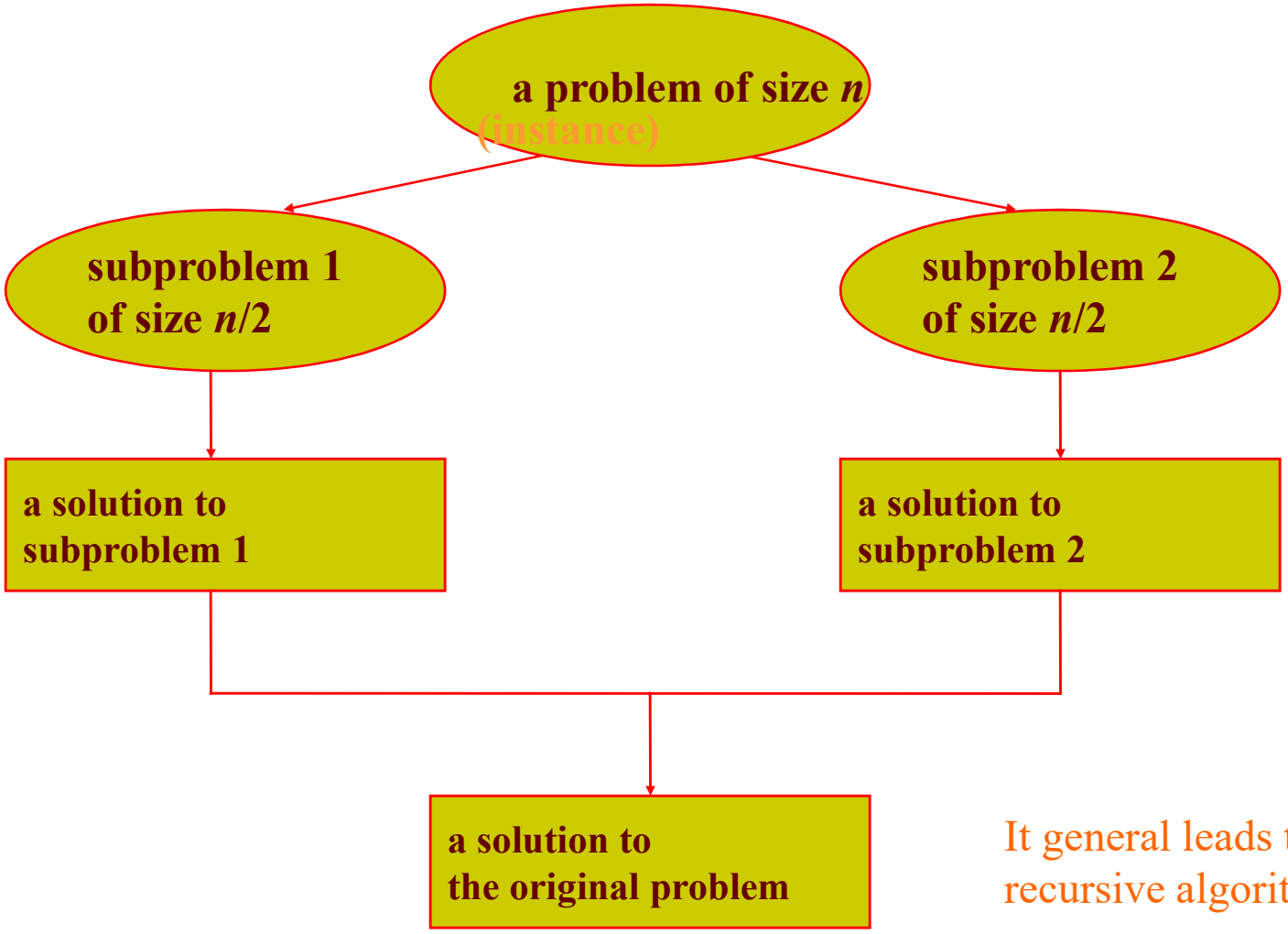
The most-well known algorithm design strategy:

1. Divide instance of problem into two or more smaller instances(sub problems)

1. Solve smaller instances recursively

1. Obtain solution to original (larger) instance by combining these solutions

Divide-and-Conquer Technique



It general leads to a recursive algorithm!

Divide-and-Conquer Examples

- Sorting: merge sort and quicksort
- Min-Max Algorithm
- Binary tree traversals
- Binary search
- Multiplication of large integers
- Matrix multiplication: Strassen's algorithm

General Divide-and-Conquer Recurrence

$T(n) = aT(n/b) + f(n)$ where $f(n) \in \Theta(n^d)$, $d \geq 0$

Master Theorem: If $a < b^d$, $T(n) \in \Theta(n^d)$

If $a = b^d$, $T(n) \in \Theta(n^d \log n)$

If $a > b^d$, $T(n) \in \Theta(n^{\log_b a})$

Note: The same results hold with O instead of Θ.

Examples: $T(n) = 4T(n/2) + n \Rightarrow T(n) \in ?$

$T(n) = 4T(n/2) + n^2 \Rightarrow T(n) \in ?$

$T(n) = 4T(n/2) + n^3 \Rightarrow T(n) \in ?$

Θ(n^2)

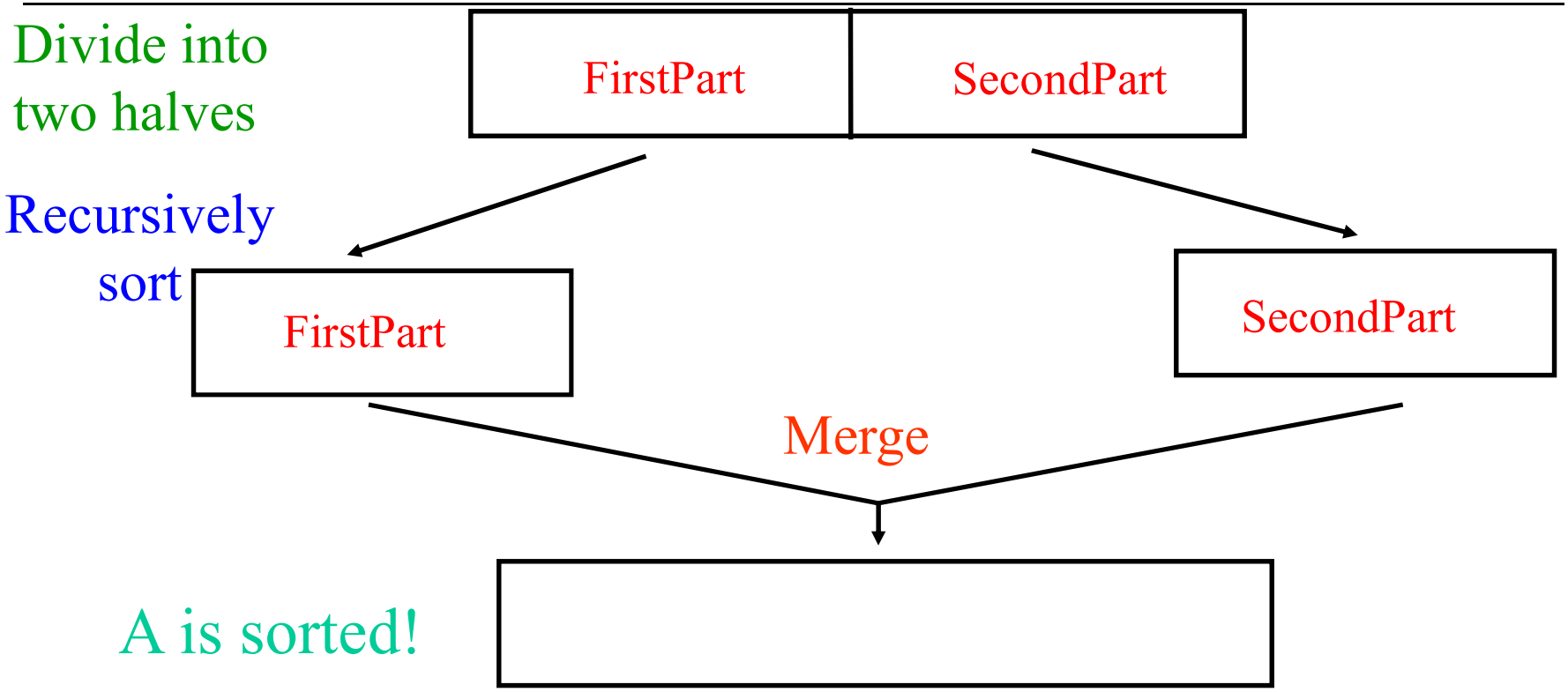
Θ(n^2log n)

Θ(n^3)

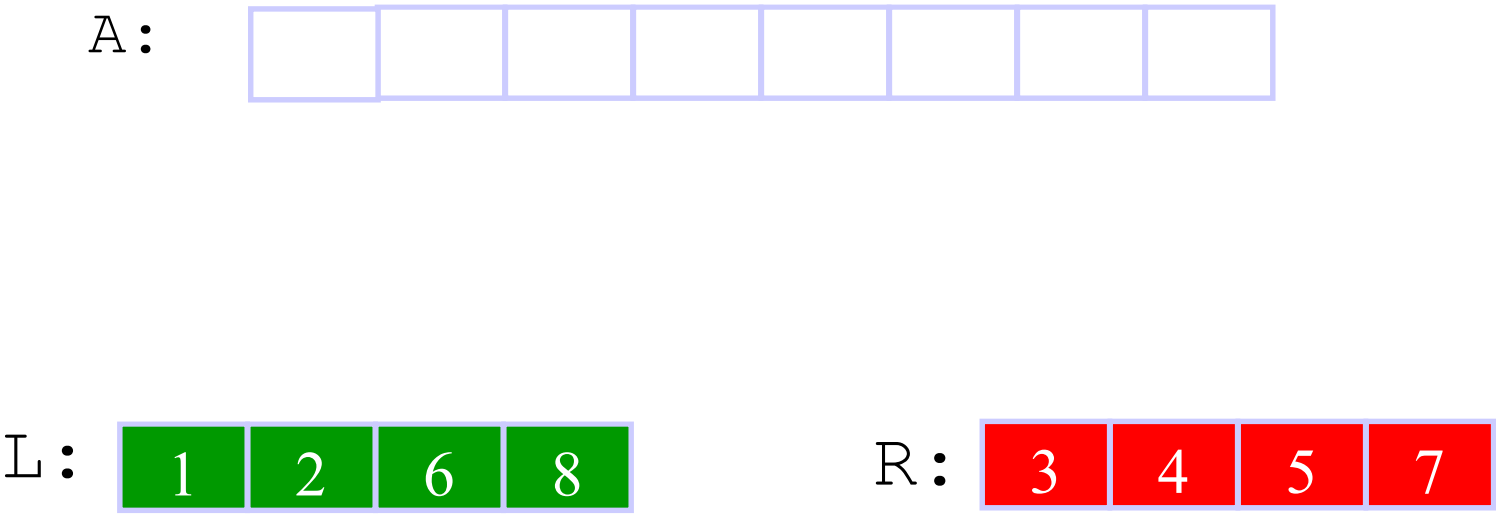
Mergesort

- Split array A[0..*n*-1] into about equal halves and make copies of each half in arrays B and C
- Sort arrays B and C recursively
- Merge sorted arrays B and C into array A as follows:
  - Repeat the following until no elements remain in one of the arrays:
    - compare the first elements in the remaining unprocessed portions of the arrays
    - copy the smaller of the two into A, while incrementing the index indicating the unprocessed portion of that array
  - Once all elements in one of the arrays are processed, copy the remaining unprocessed elements from the other array into A.

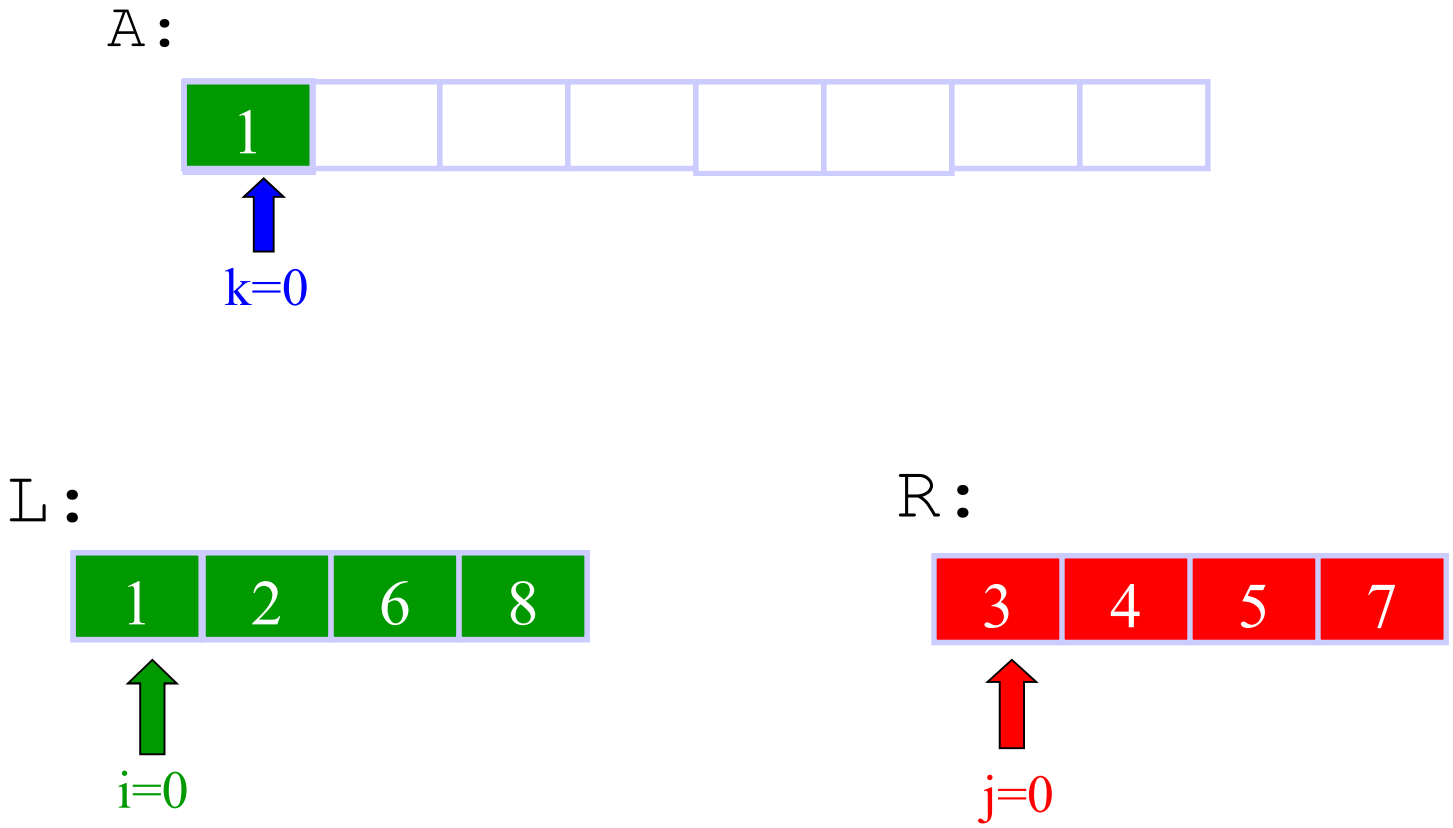
Merge Sort: Idea



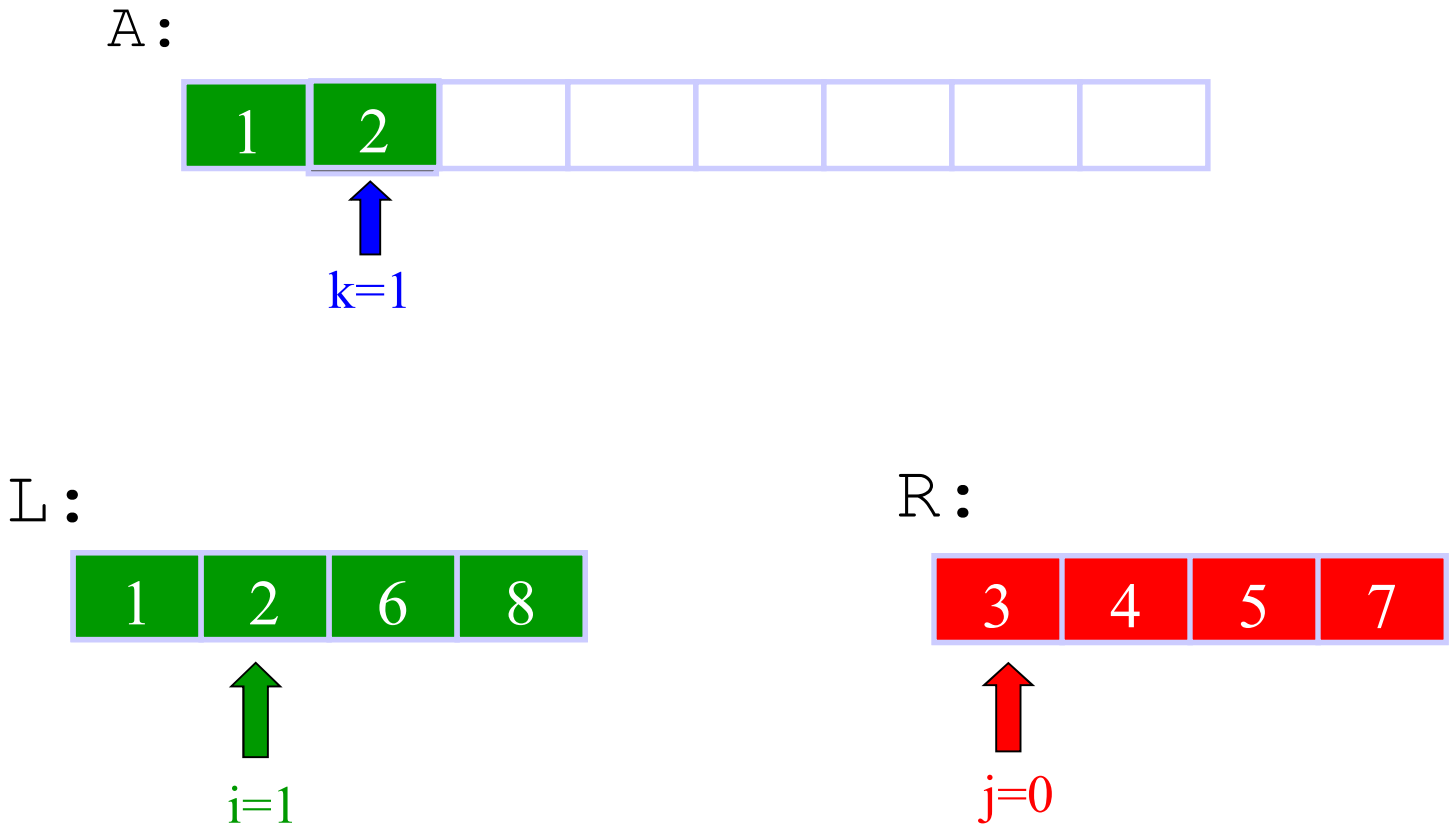
Merge-Sort: Merge Example



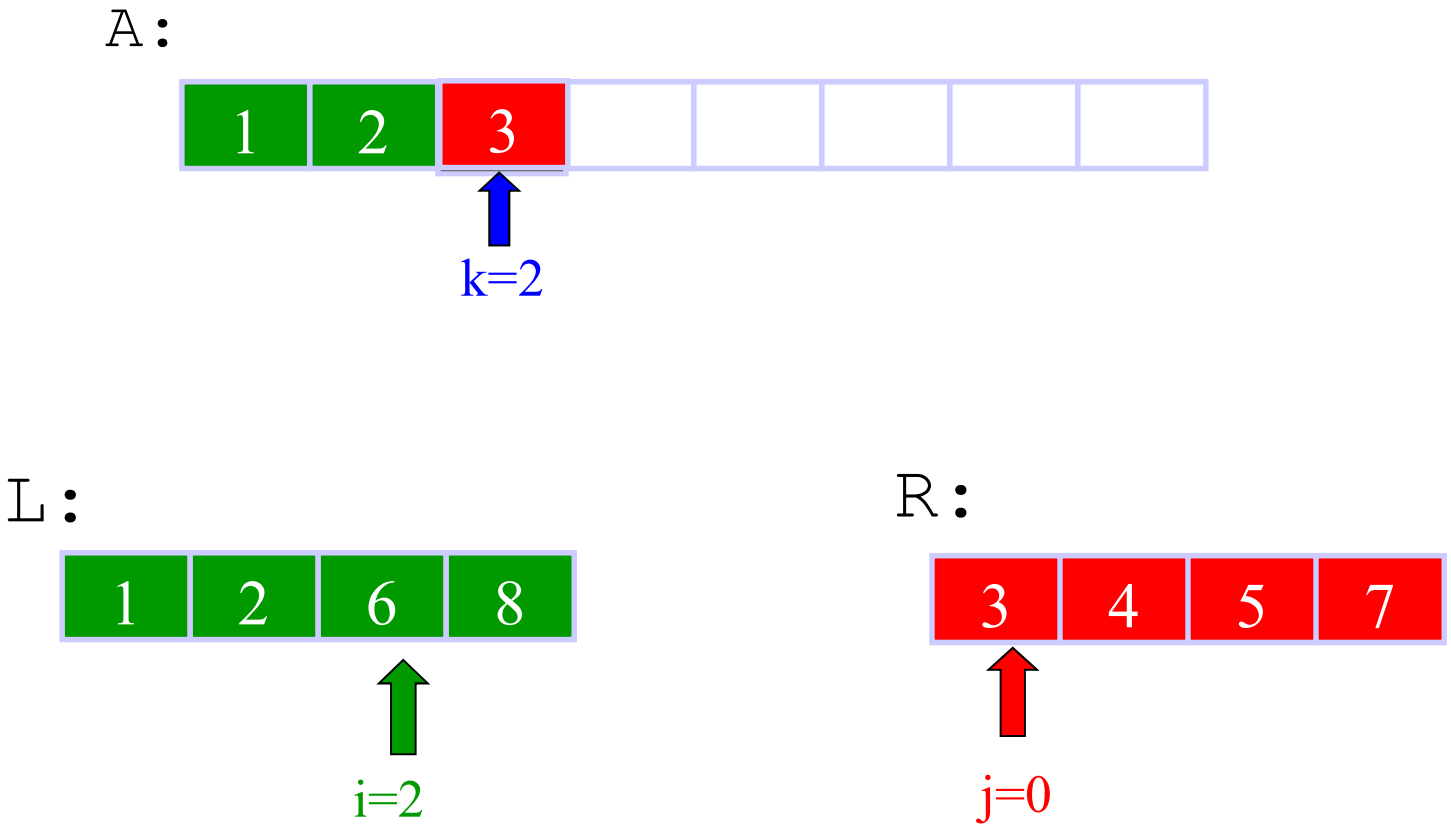
Merge-Sort: Merge Example



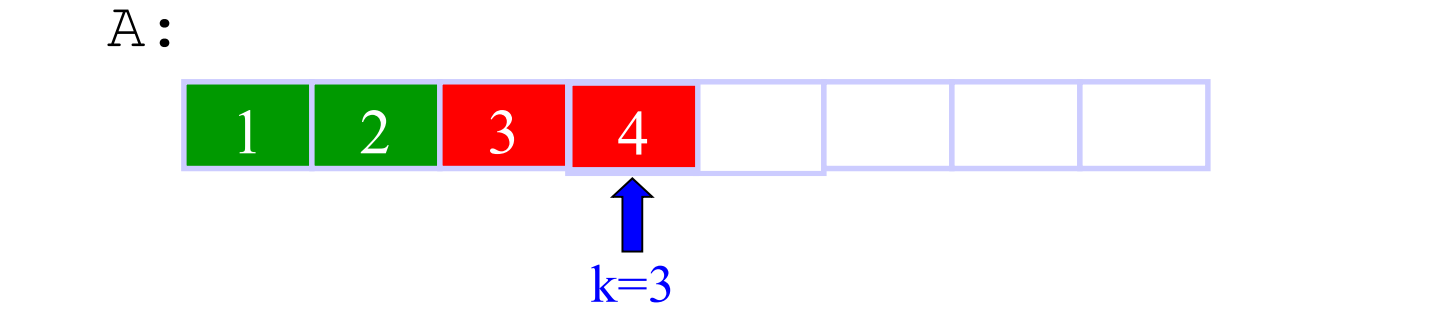
Merge-Sort: Merge Example

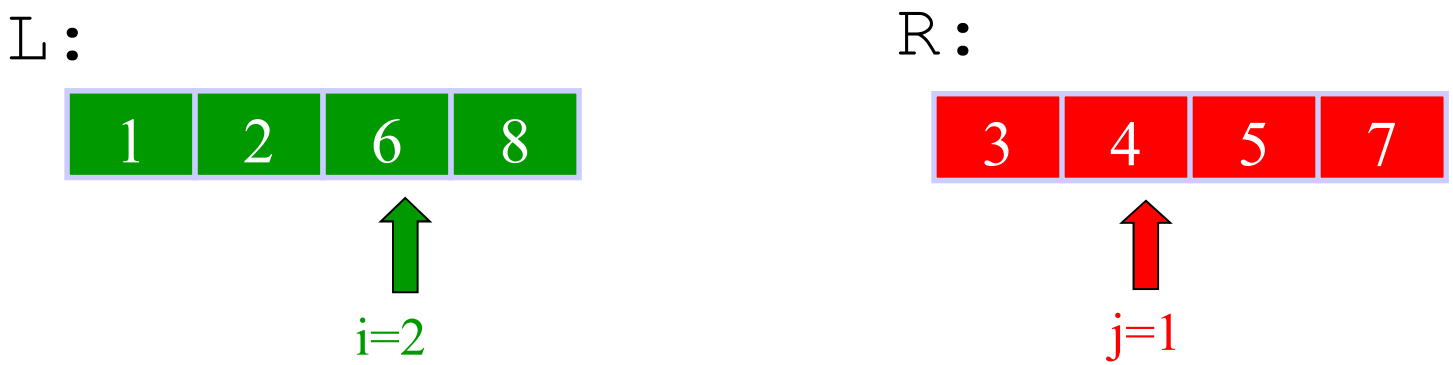


Merge-Sort: Merge Example

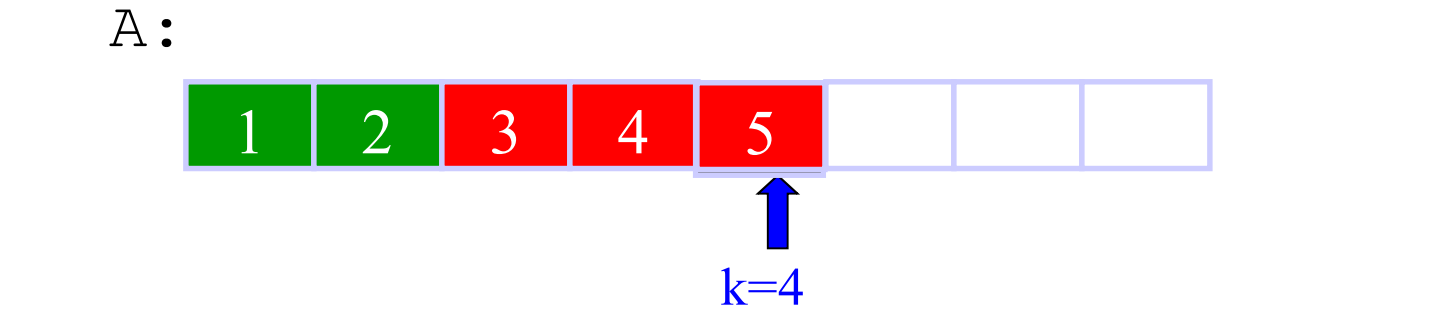


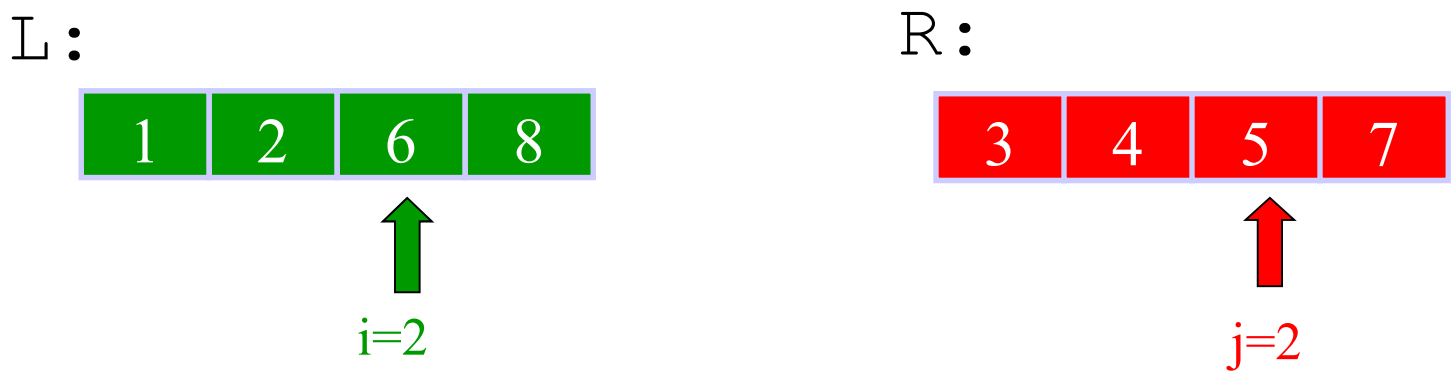
Merge-Sort: Merge Example



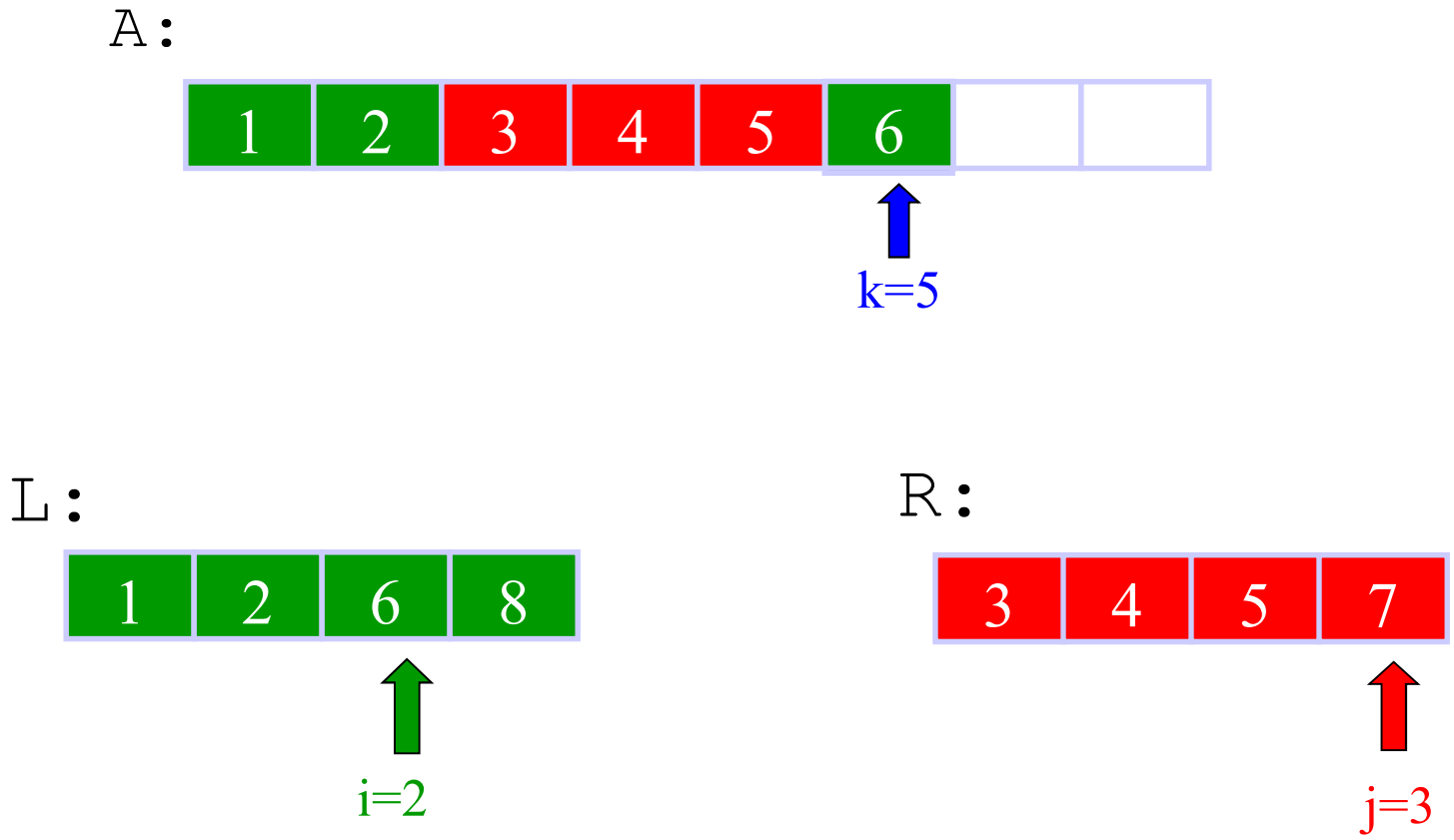


Merge-Sort: Merge Example

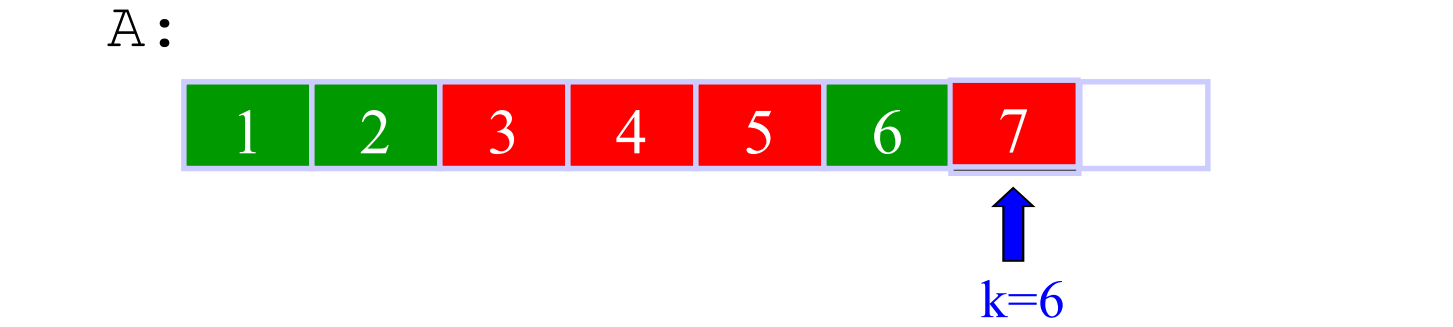


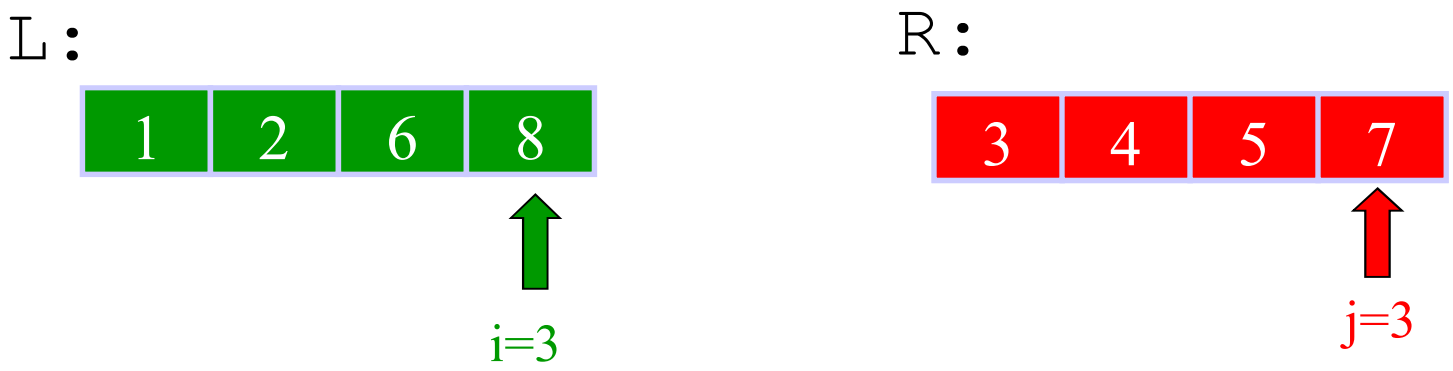


Merge-Sort: Merge Example

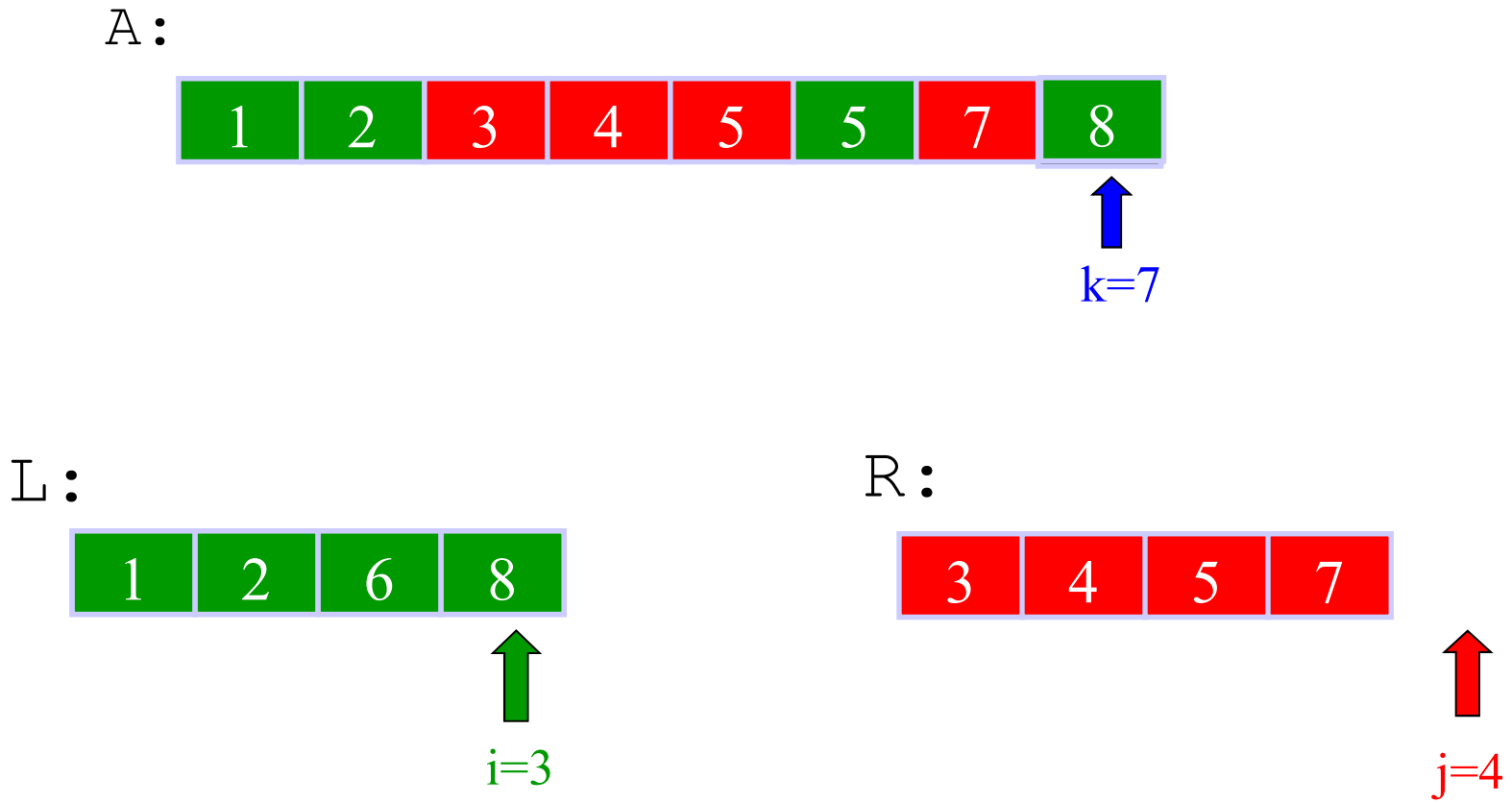


Merge-Sort: Merge Example

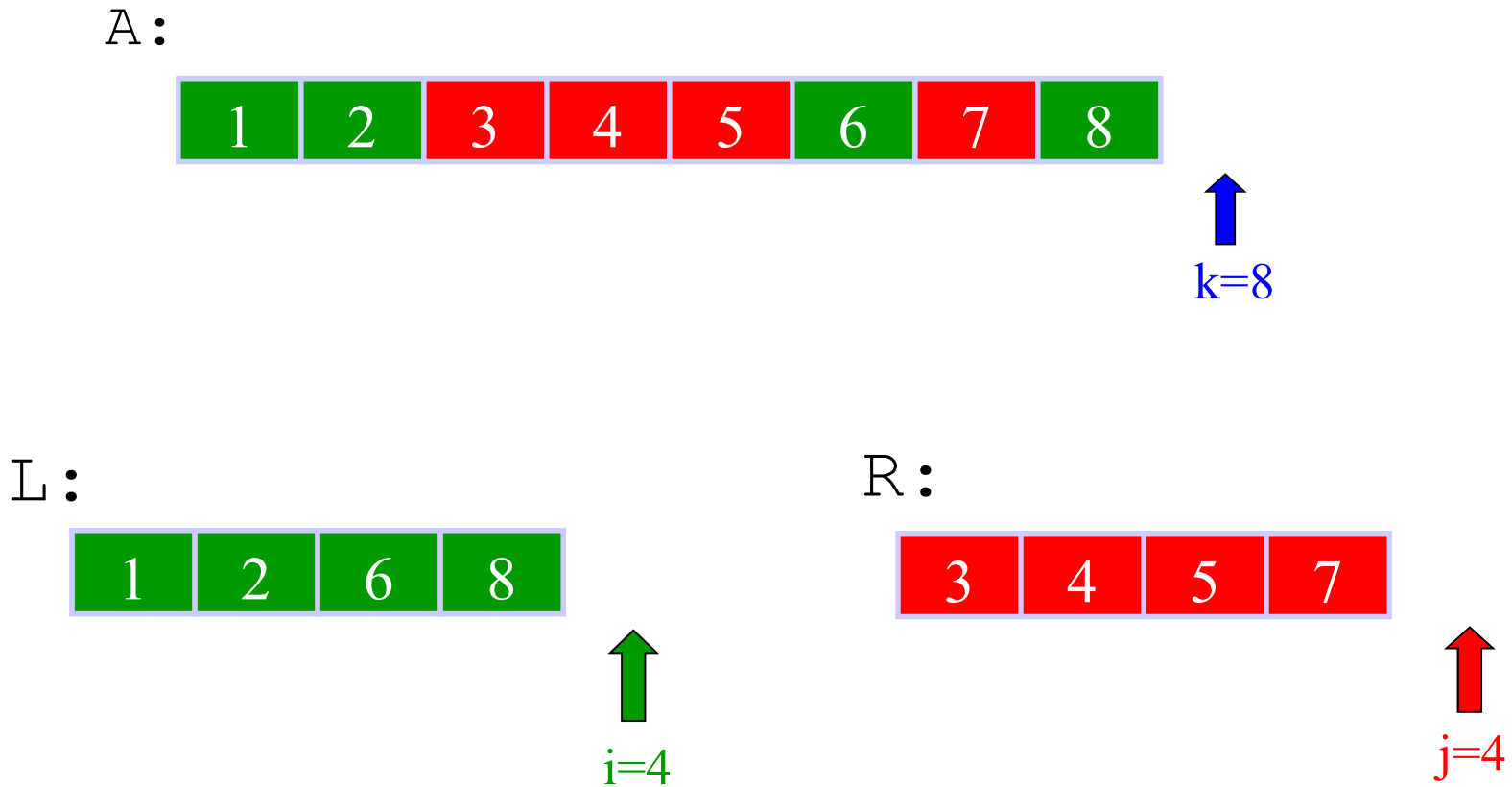




Merge-Sort: Merge Example

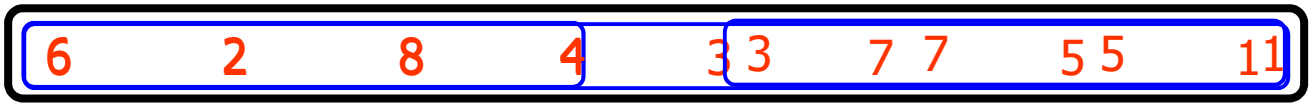


Merge-Sort: Merge Example



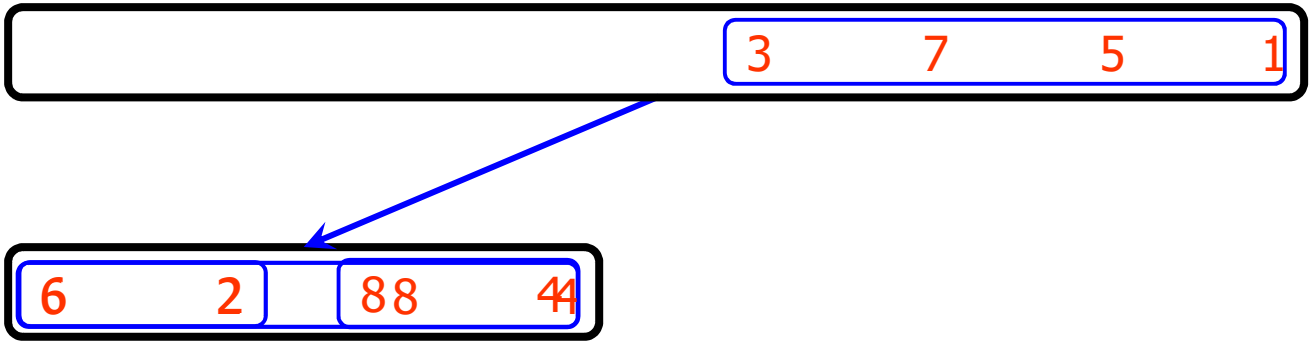
Merge-Sort Execution Example

Divide



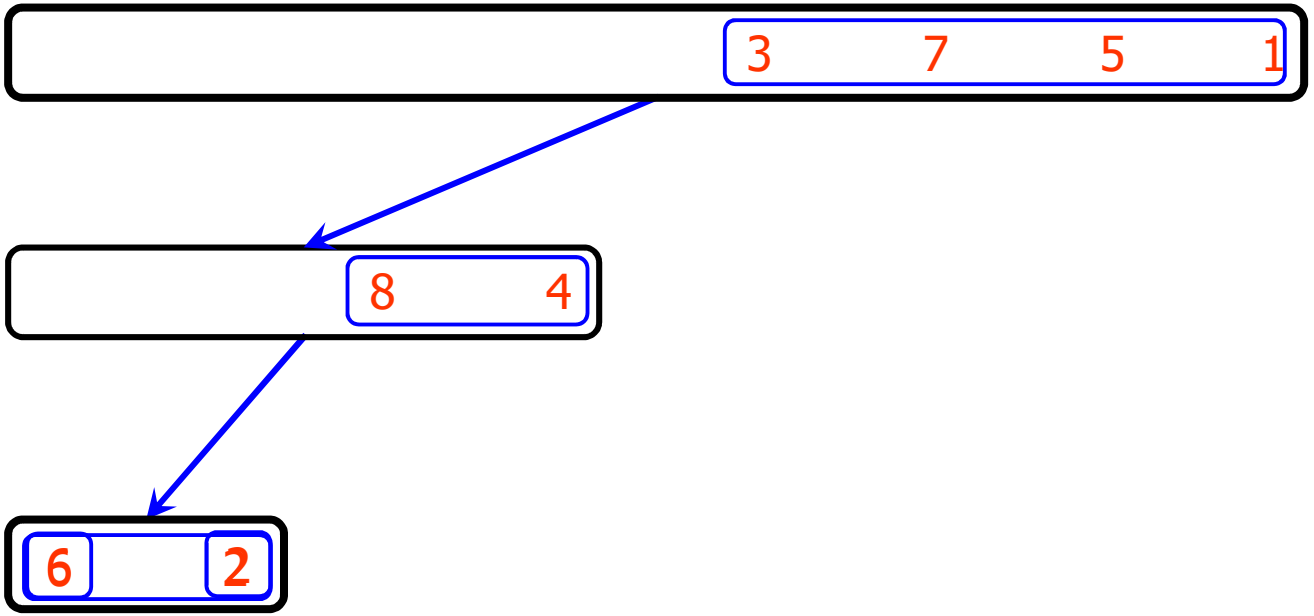
Merge-Sort Execution Example

Recursive call , divide



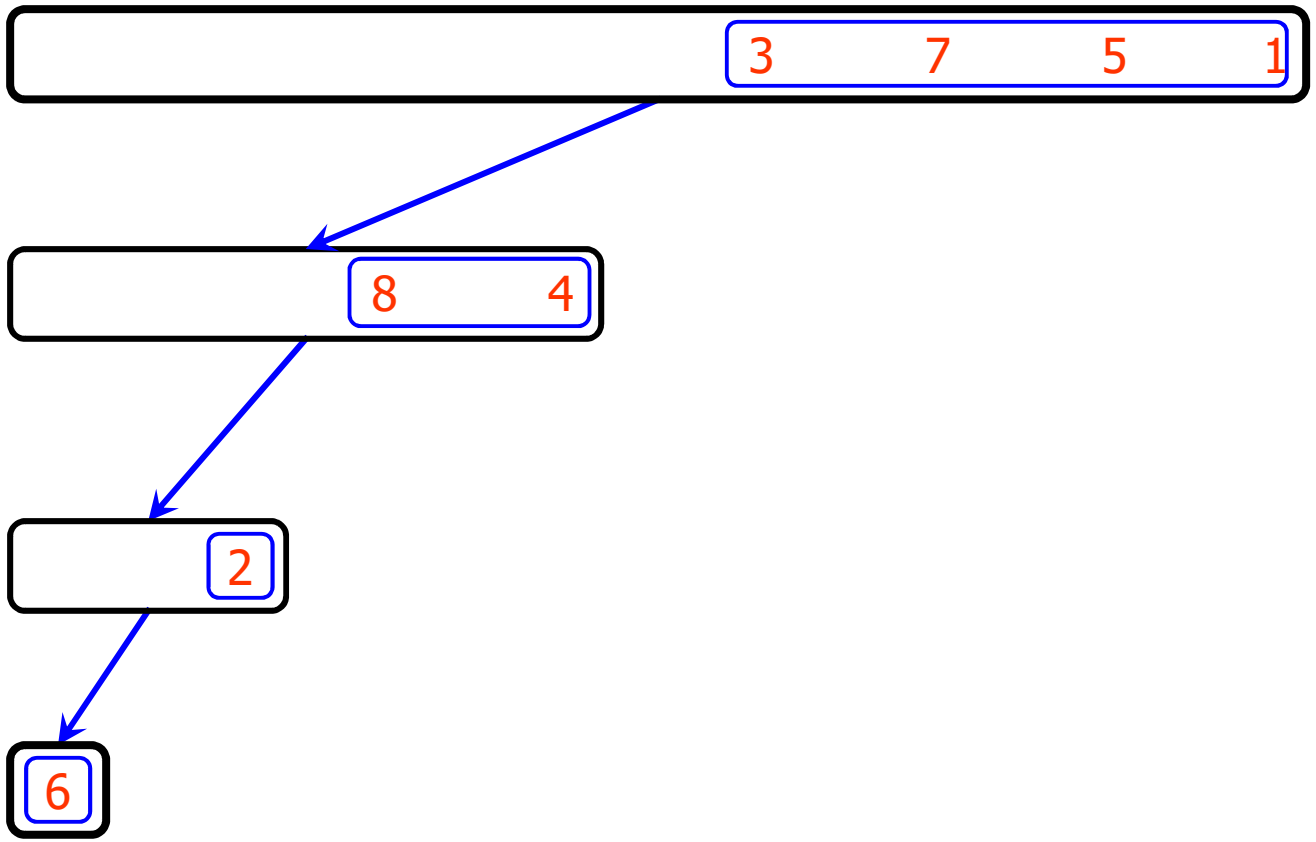
Merge-Sort Execution Example

Recursive call , divide



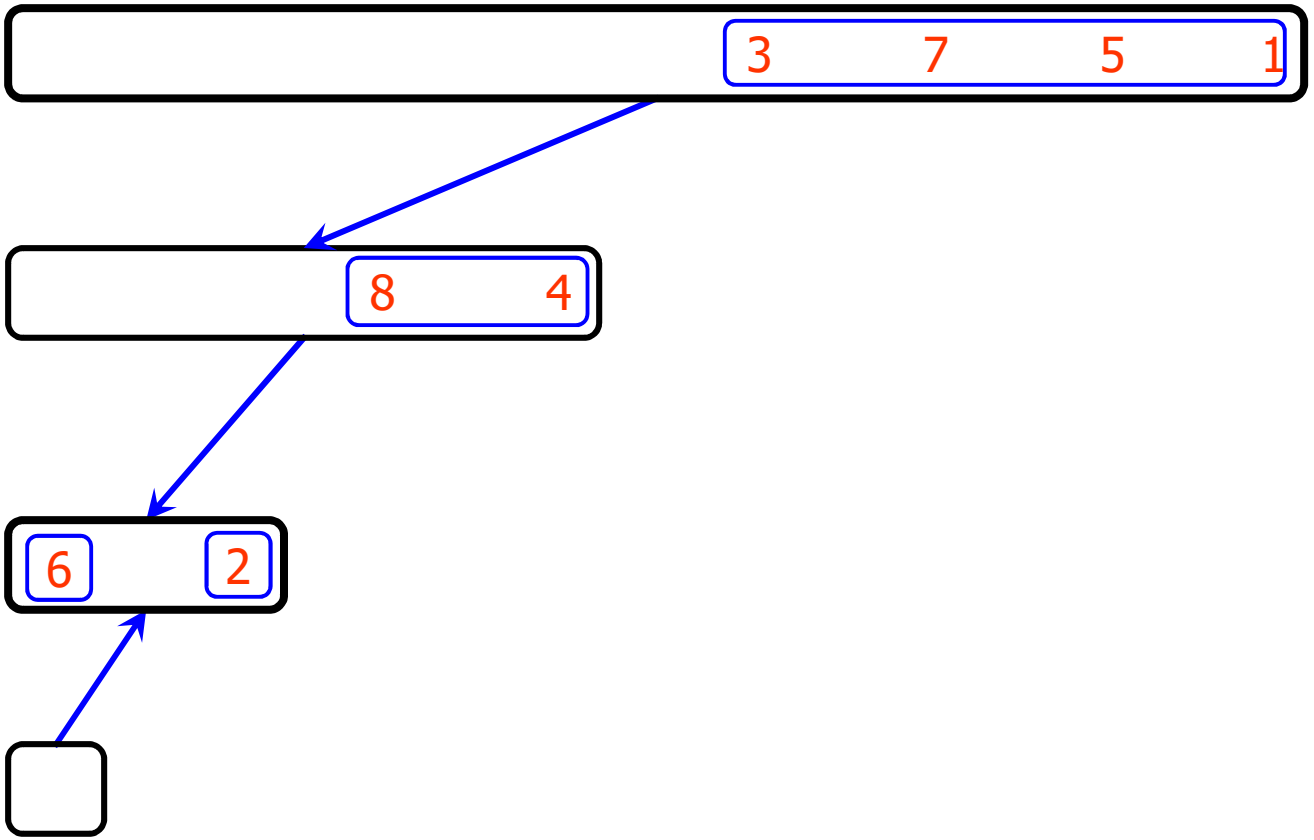
Merge-Sort Execution Example

Recursive call , base case



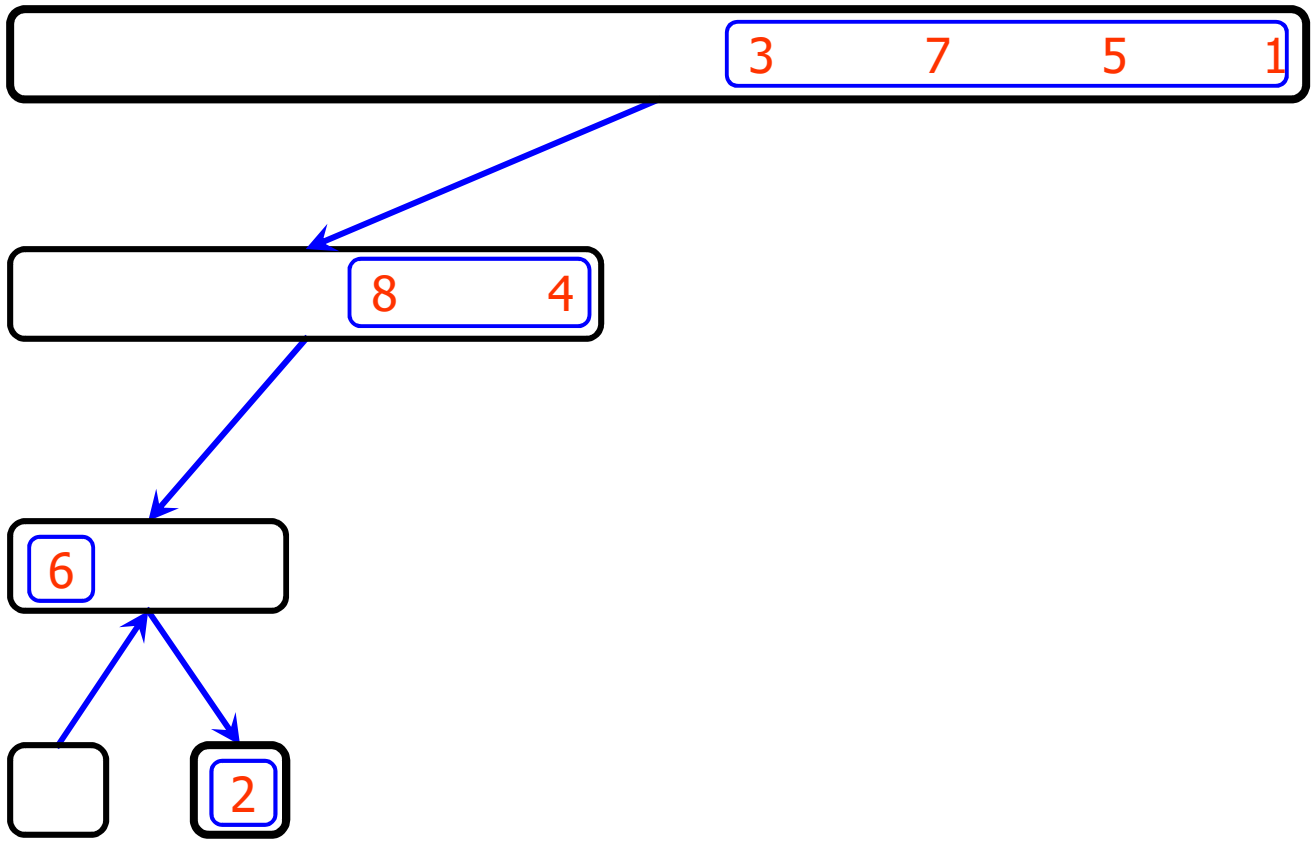
Merge-Sort Execution Example

Recursive call return



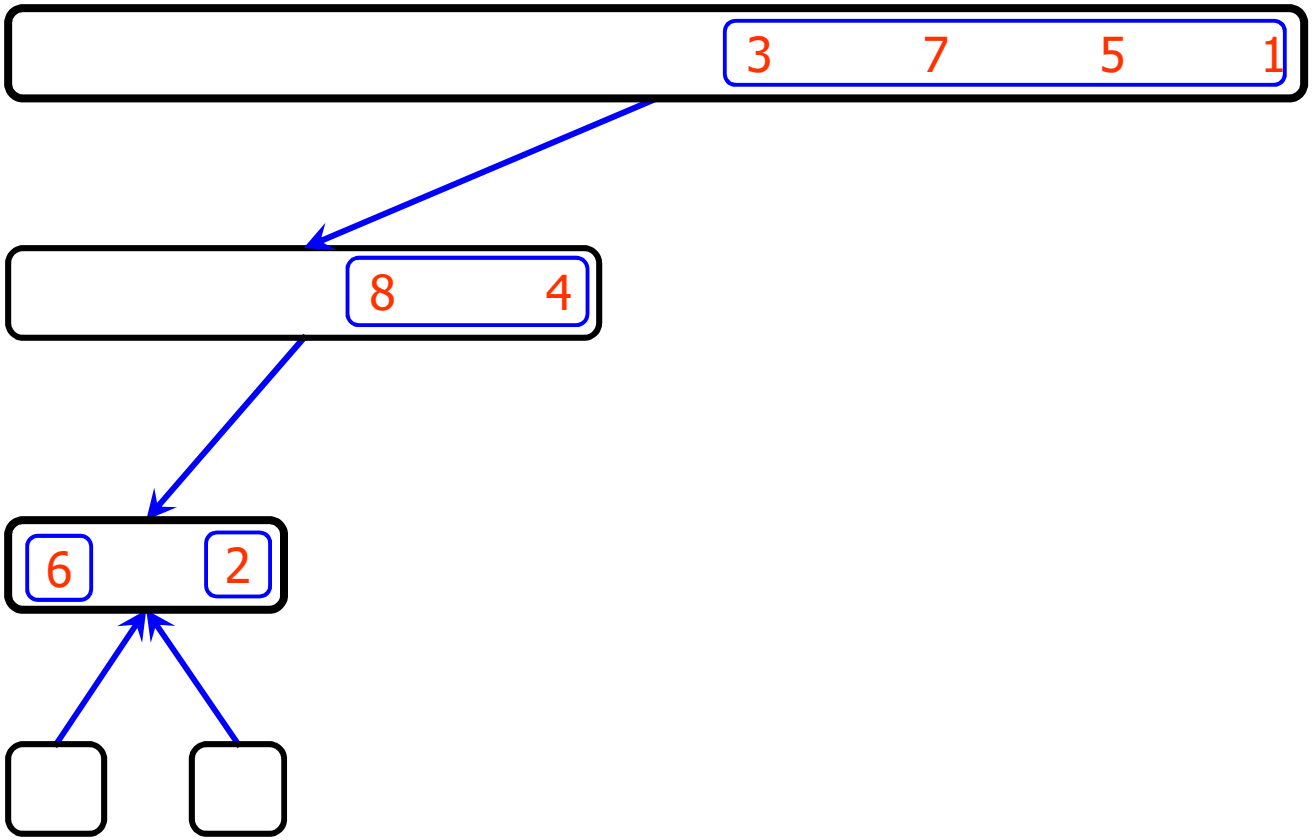
Merge-Sort Execution Example

Recursive call , base case



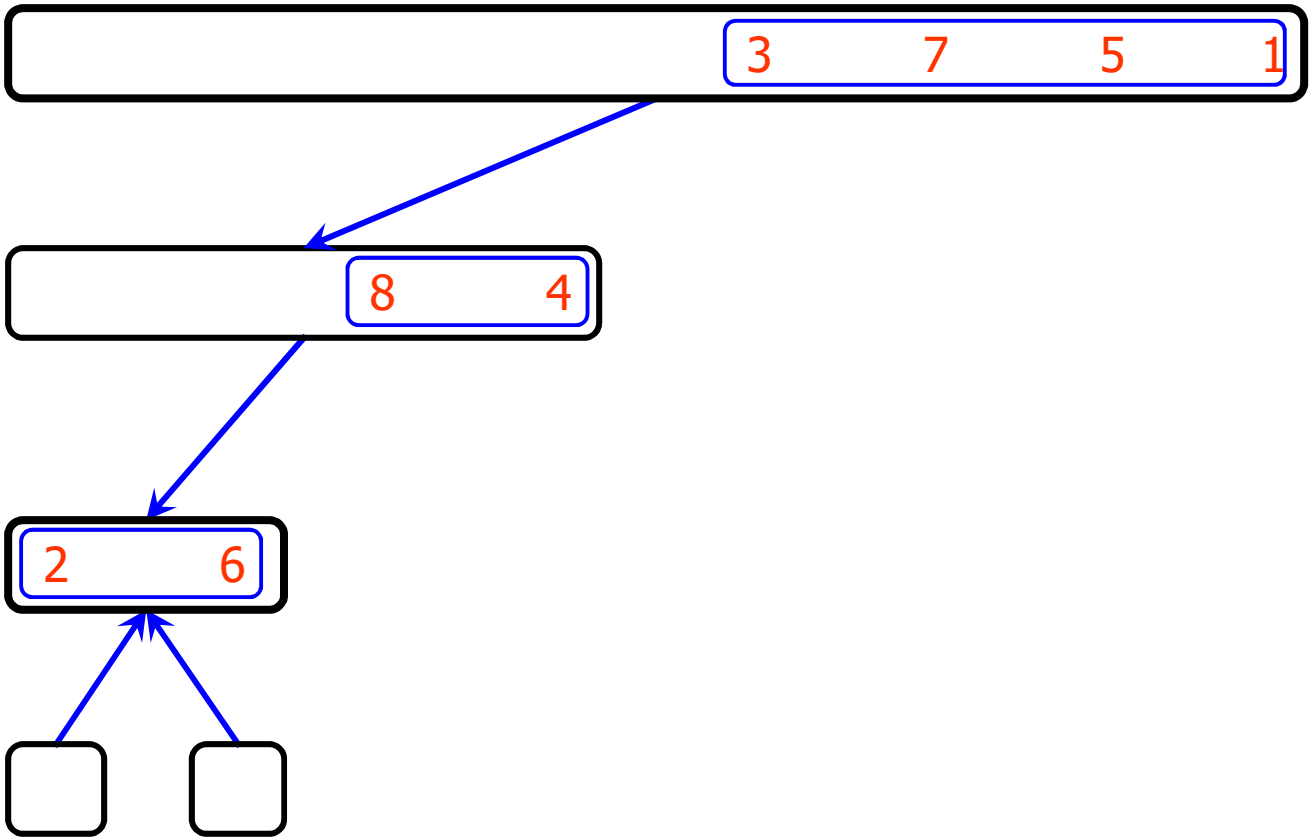
Merge-Sort Execution Example

Recursive call return



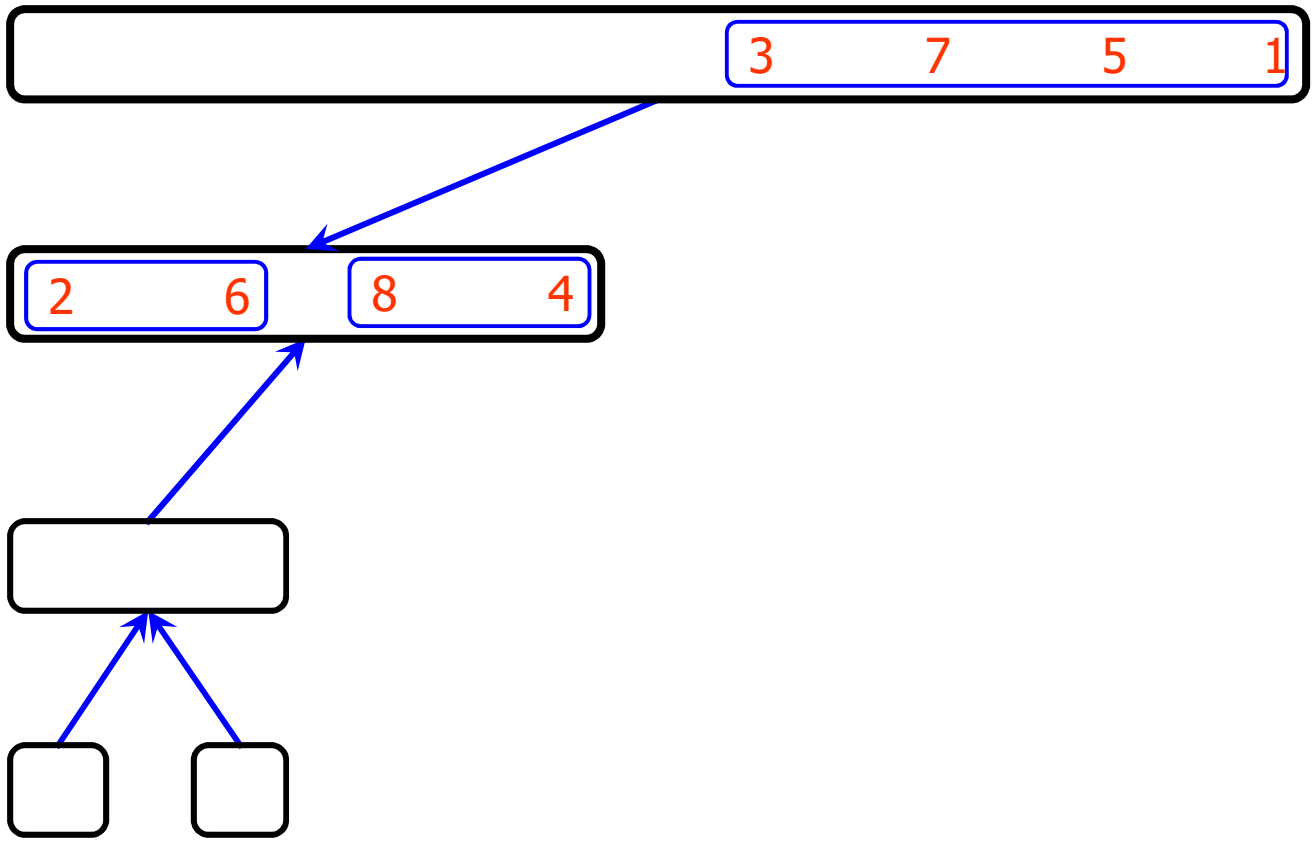
Merge-Sort Execution Example

Merge



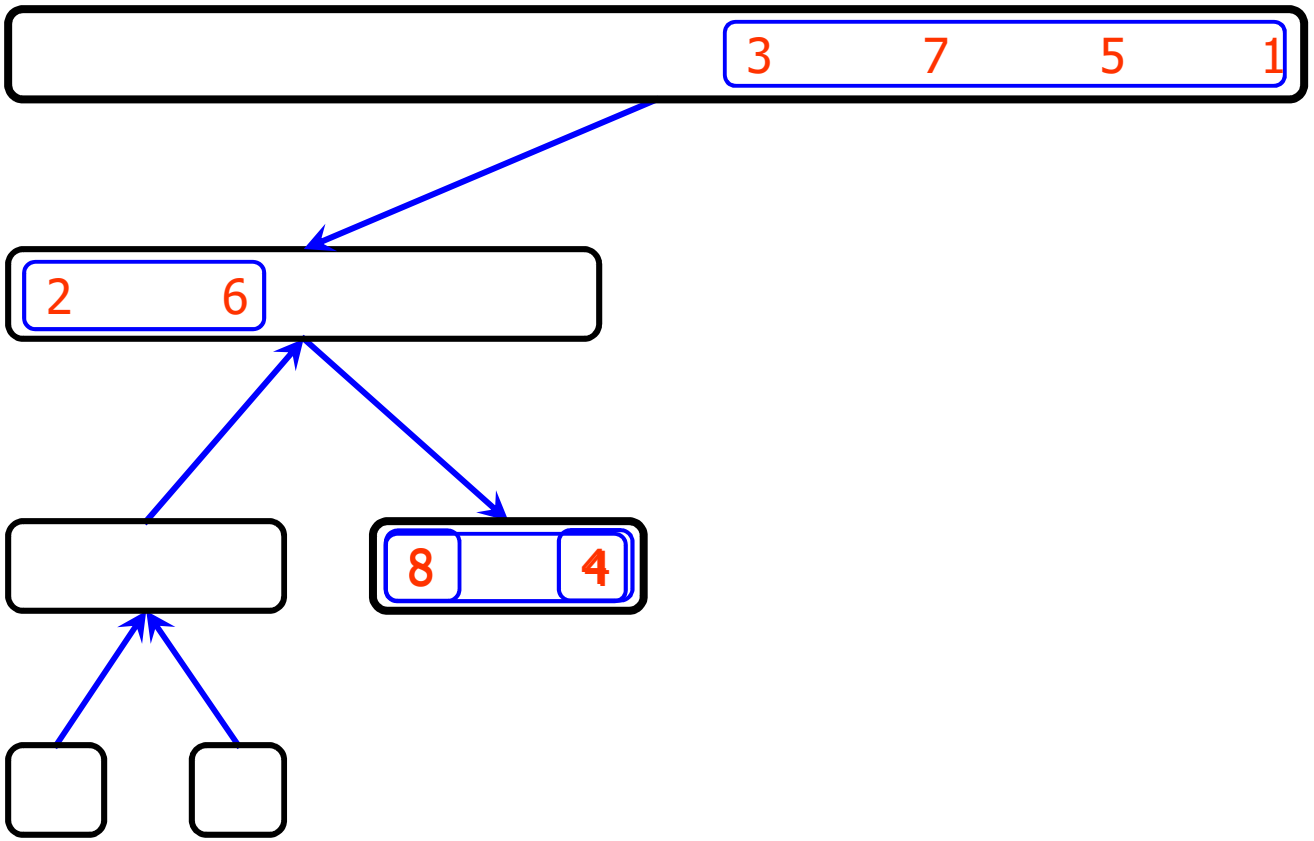
Merge-Sort Execution Example

Recursive call return



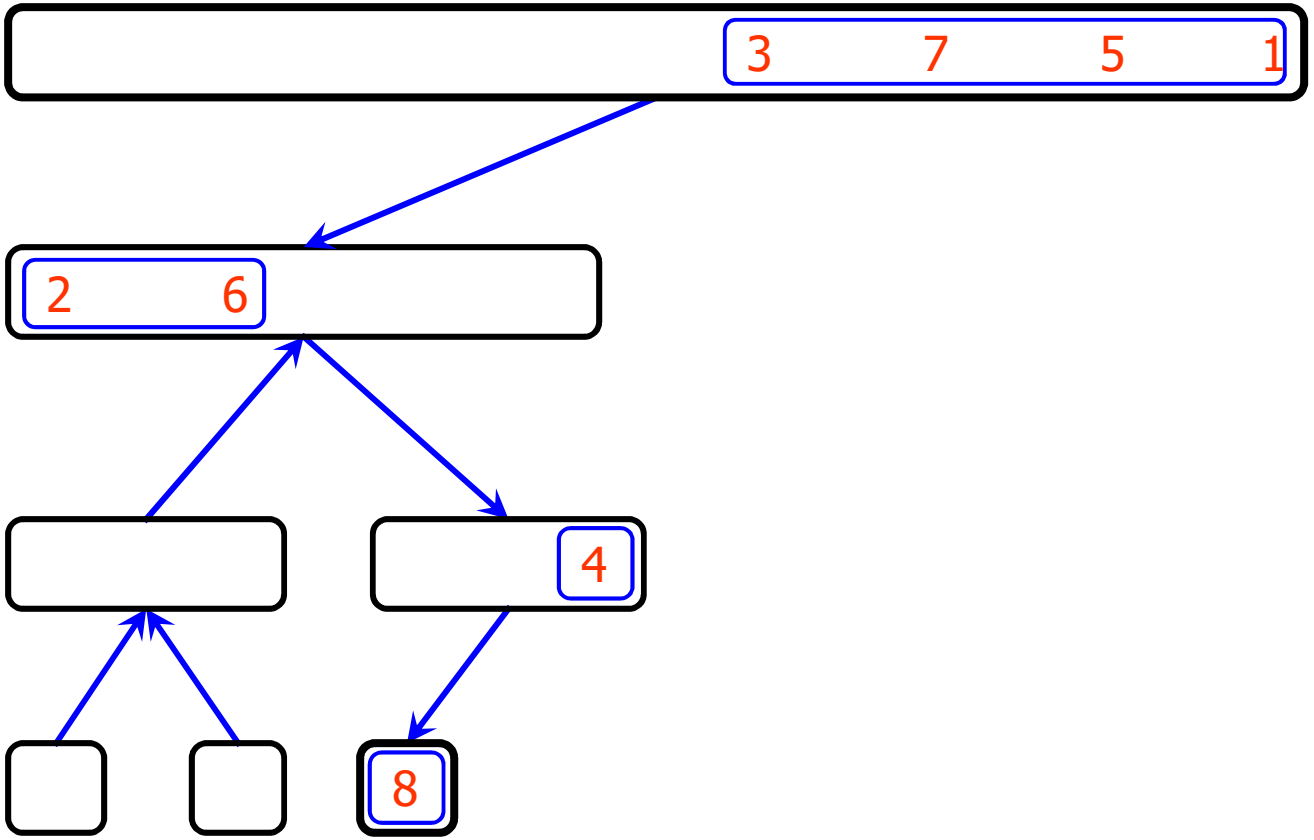
Merge-Sort Execution Example

Recursive call , divide



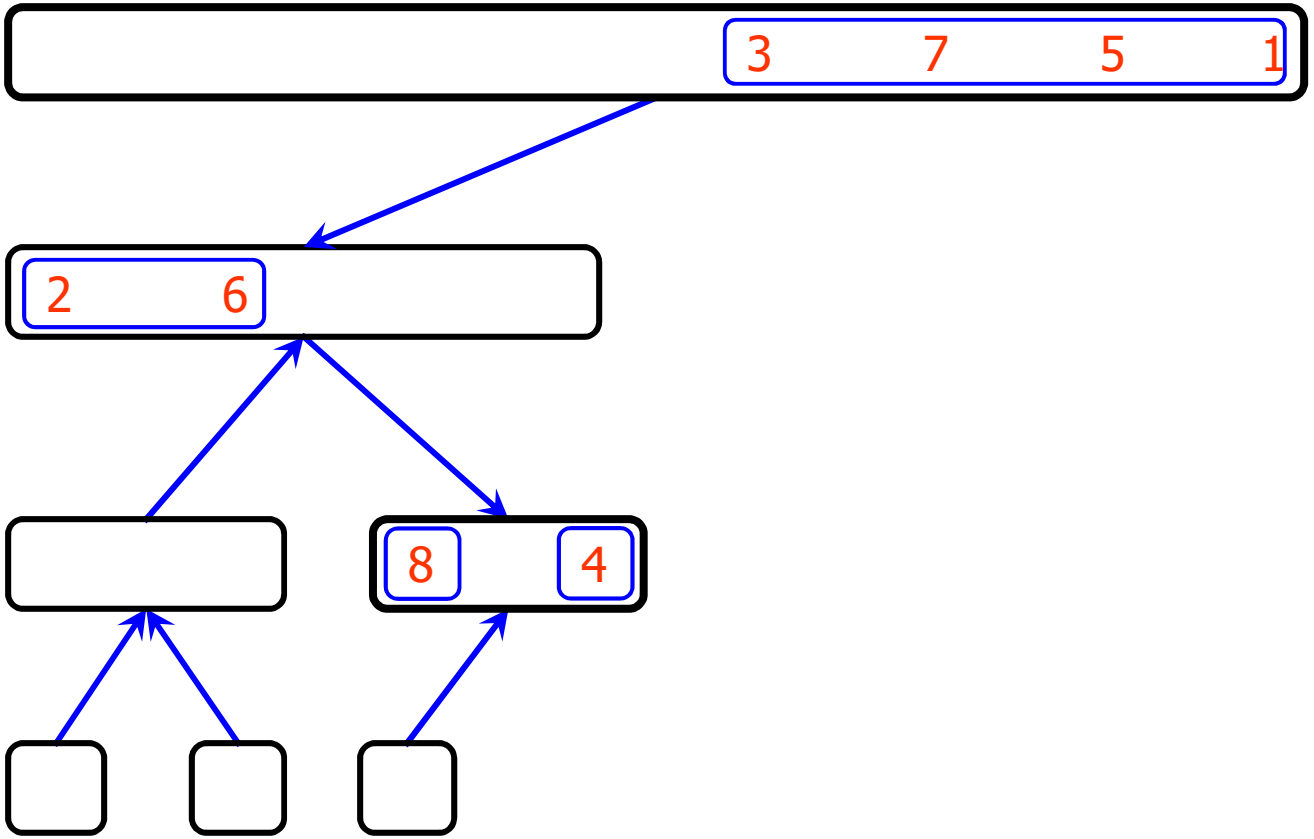
Merge-Sort Execution Example

Recursive call, base case



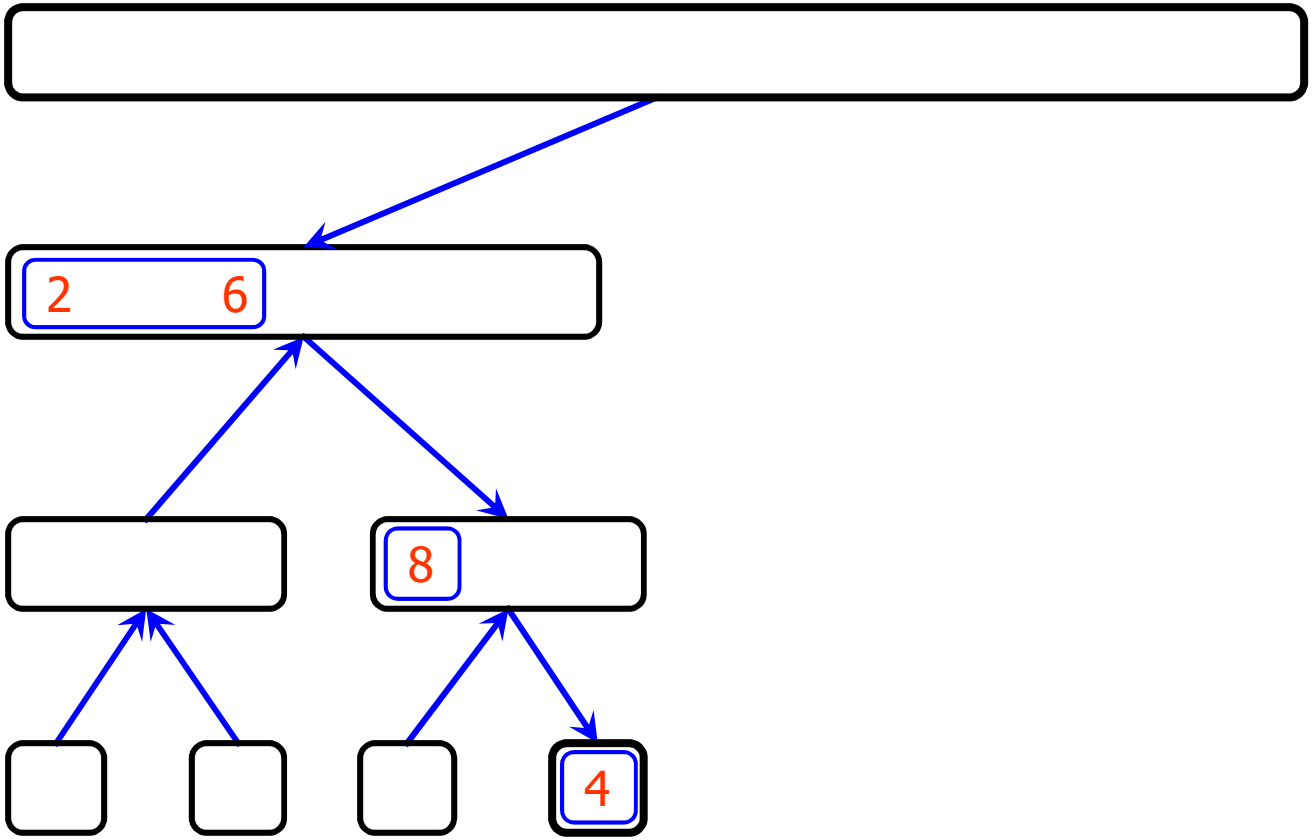
Merge-Sort Execution Example

Recursive call return



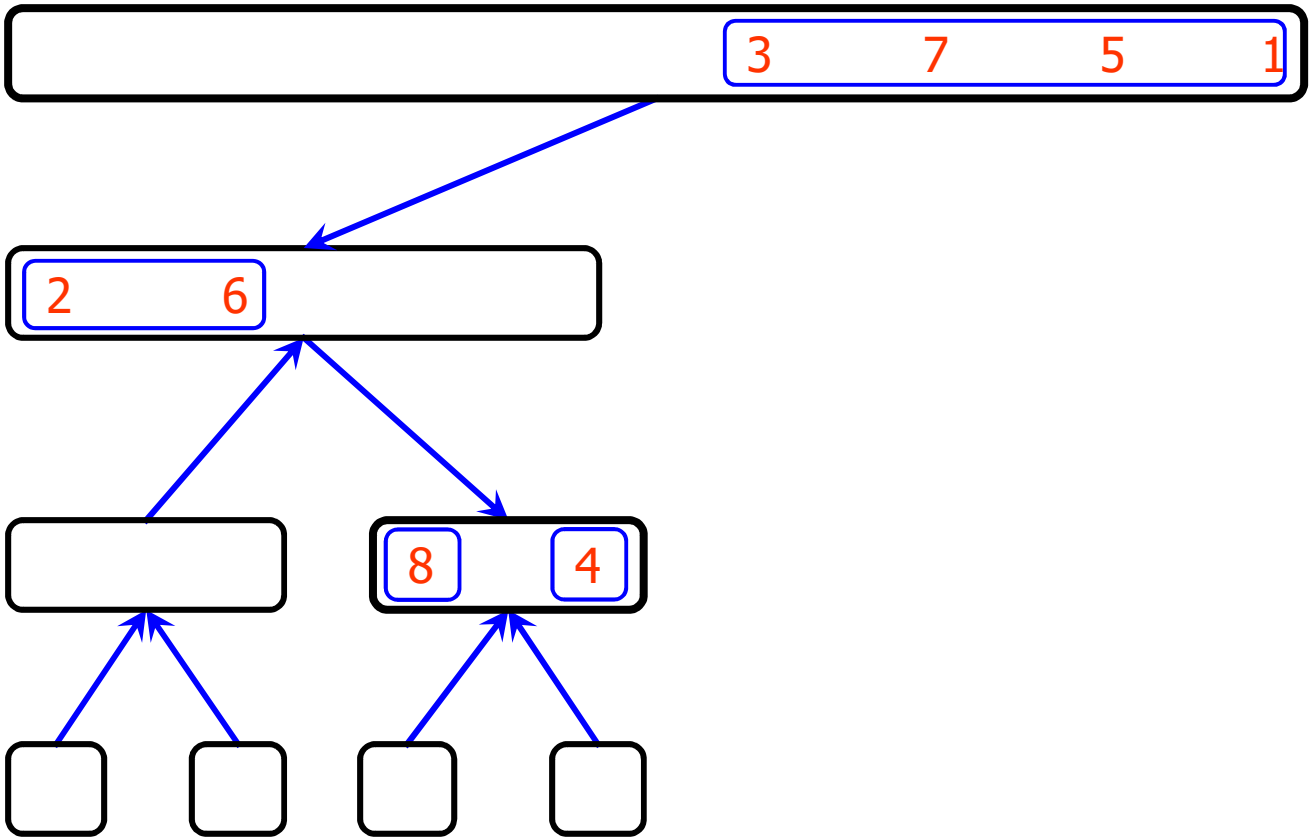
Merge-Sort Execution Example

Recursive call, base case



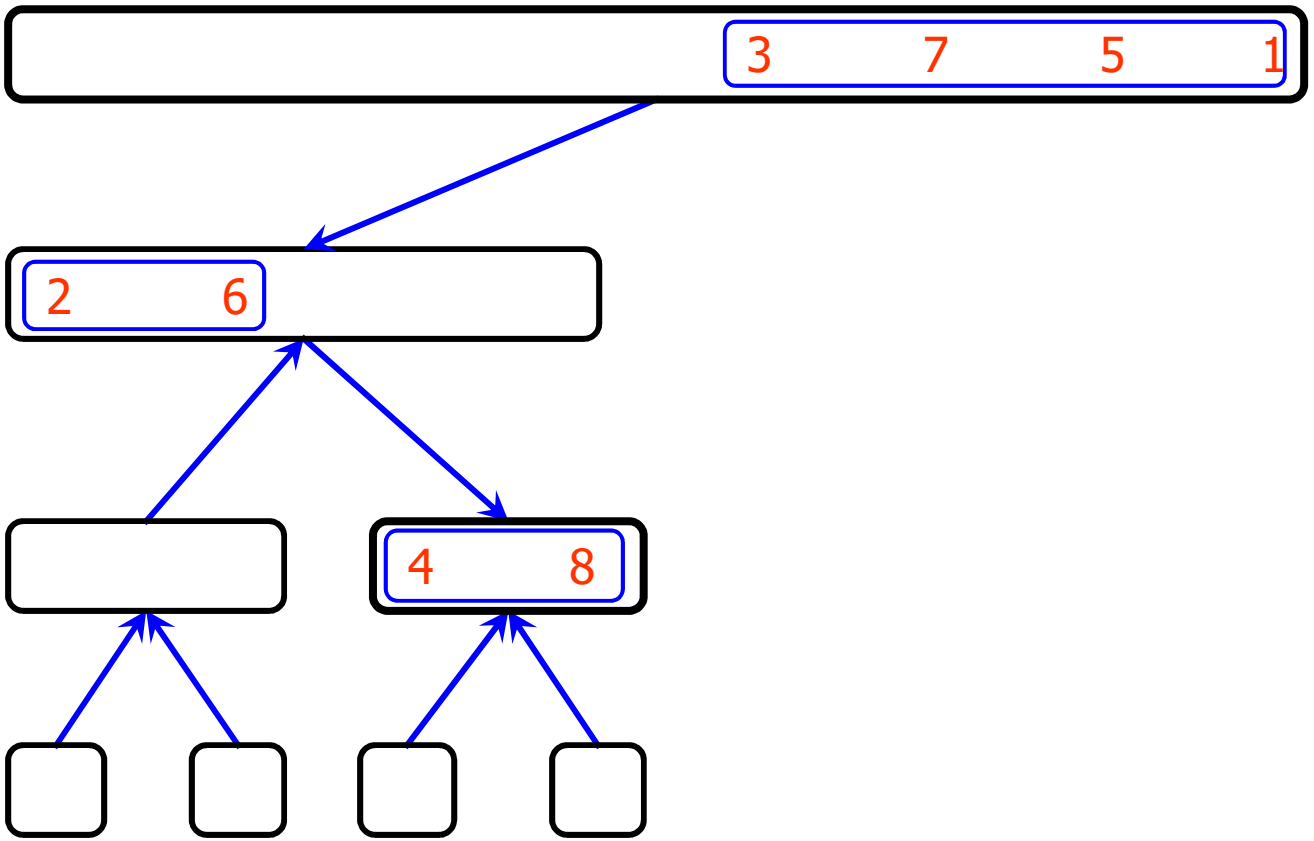
Merge-Sort Execution Example

Recursive call return



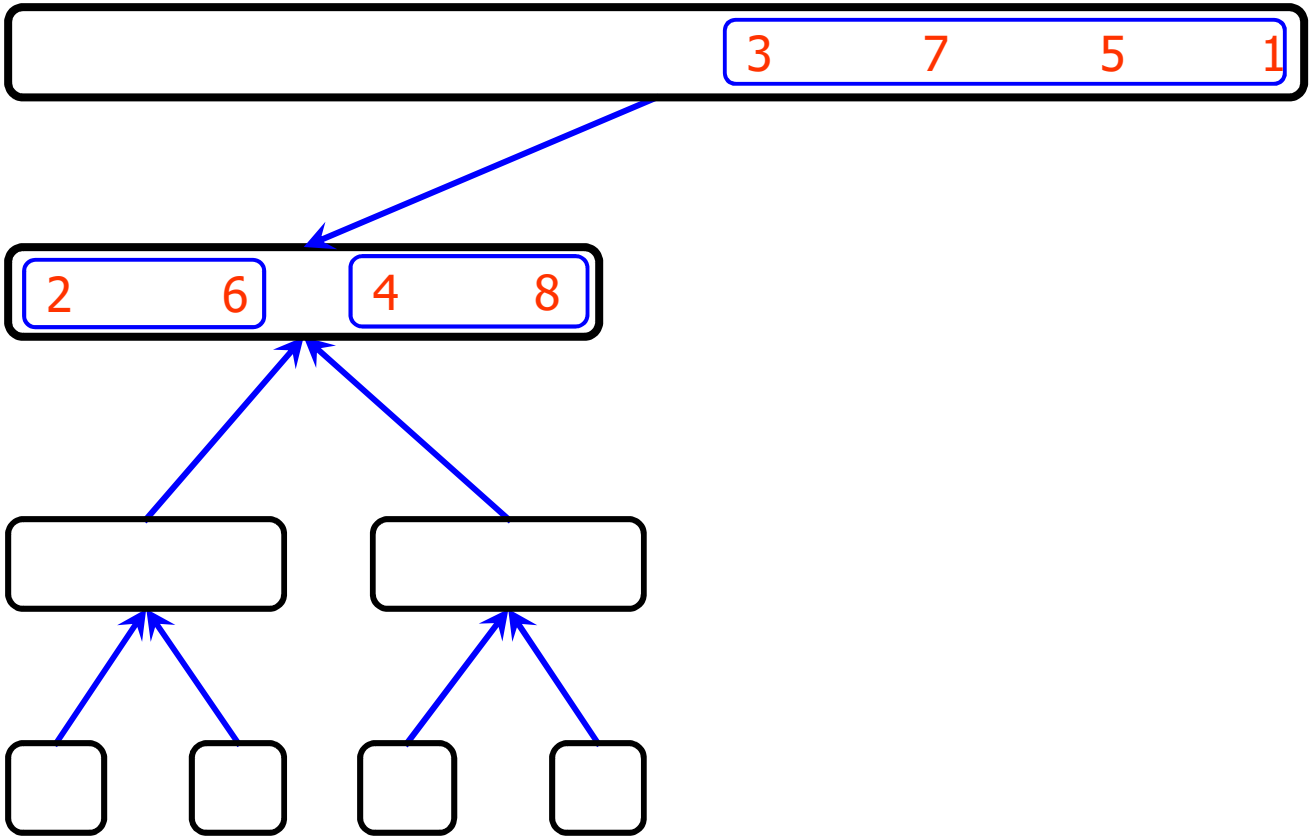
Merge-Sort Execution Example

merge



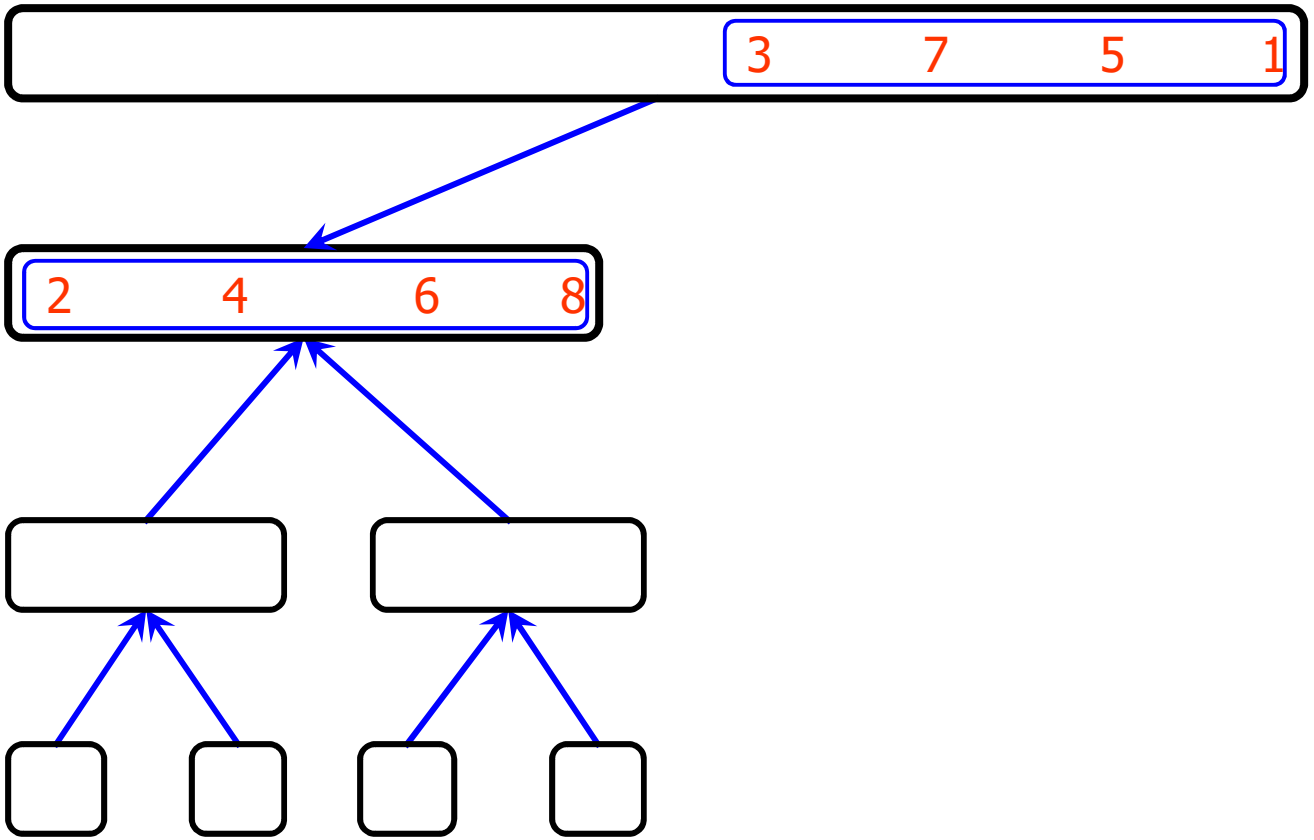
Merge-Sort Execution Example

Recursive call return



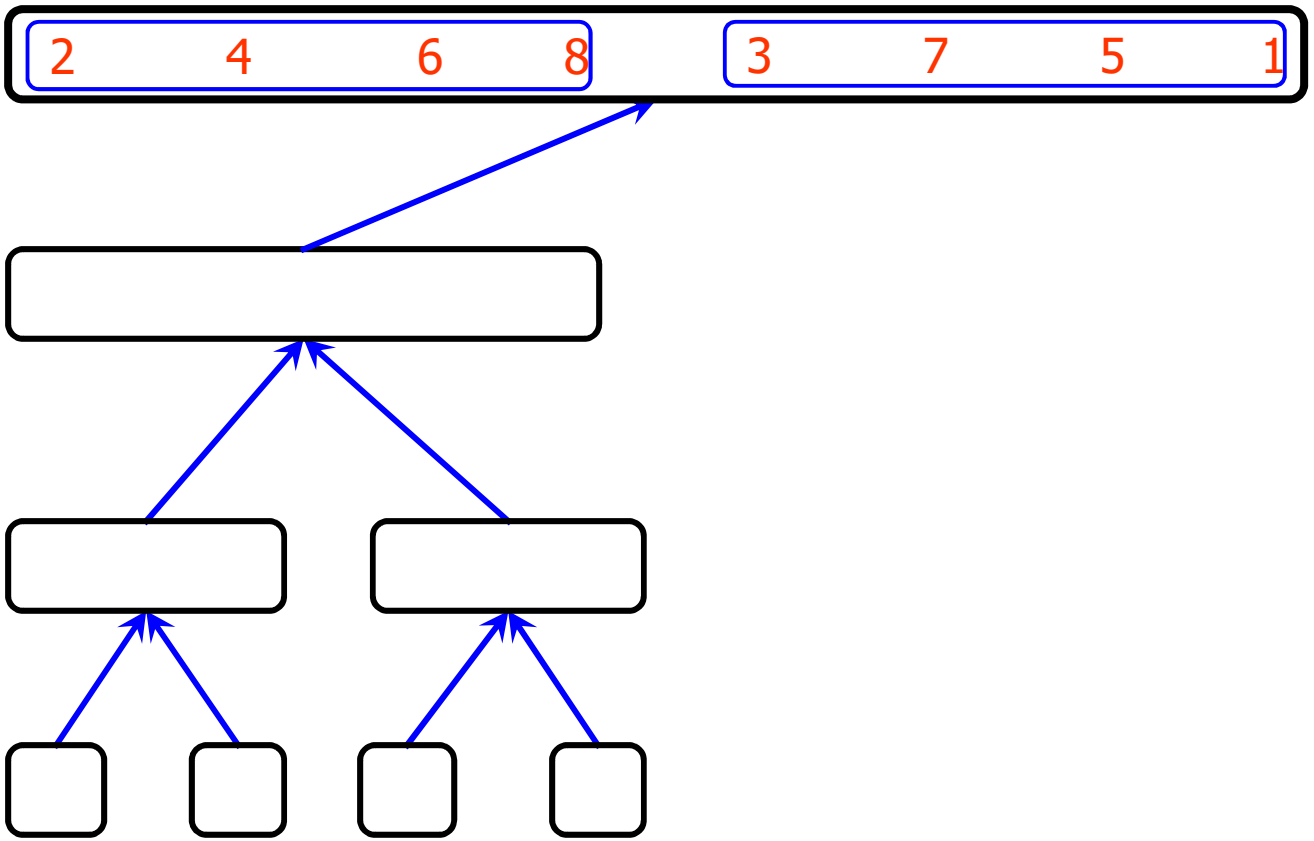
Merge-Sort Execution Example

merge



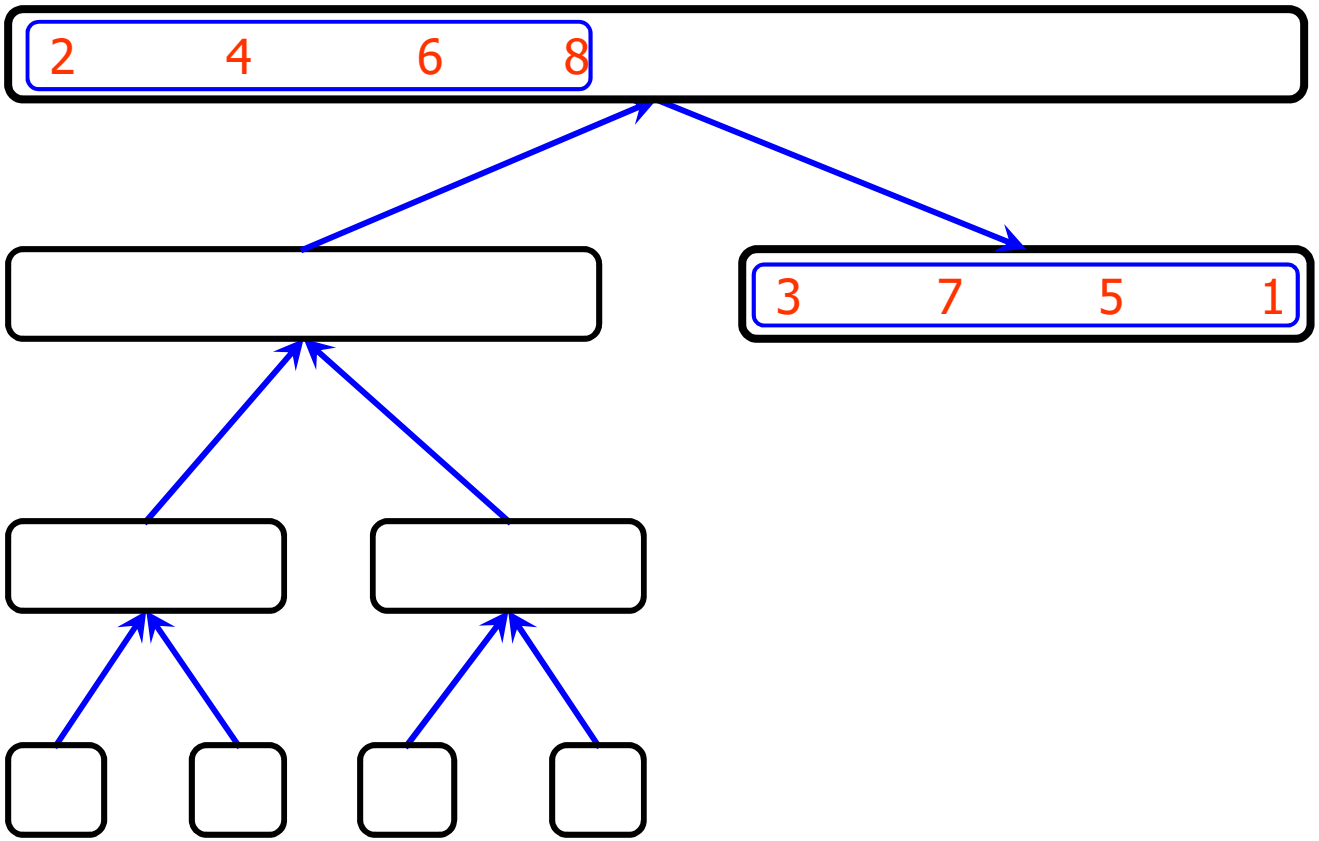
Merge-Sort Execution Example

Recursive call return

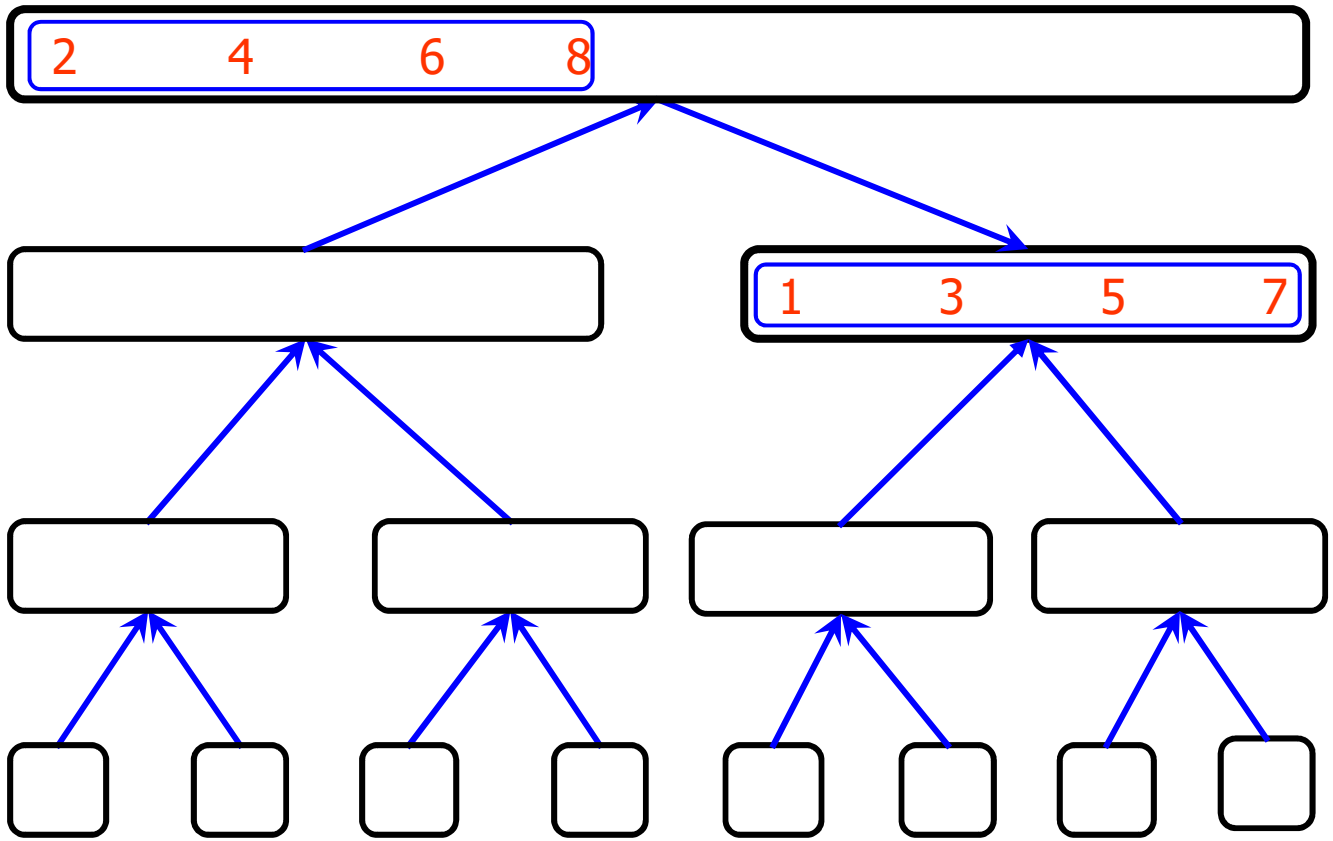


Merge-Sort Execution Example

Recursive call

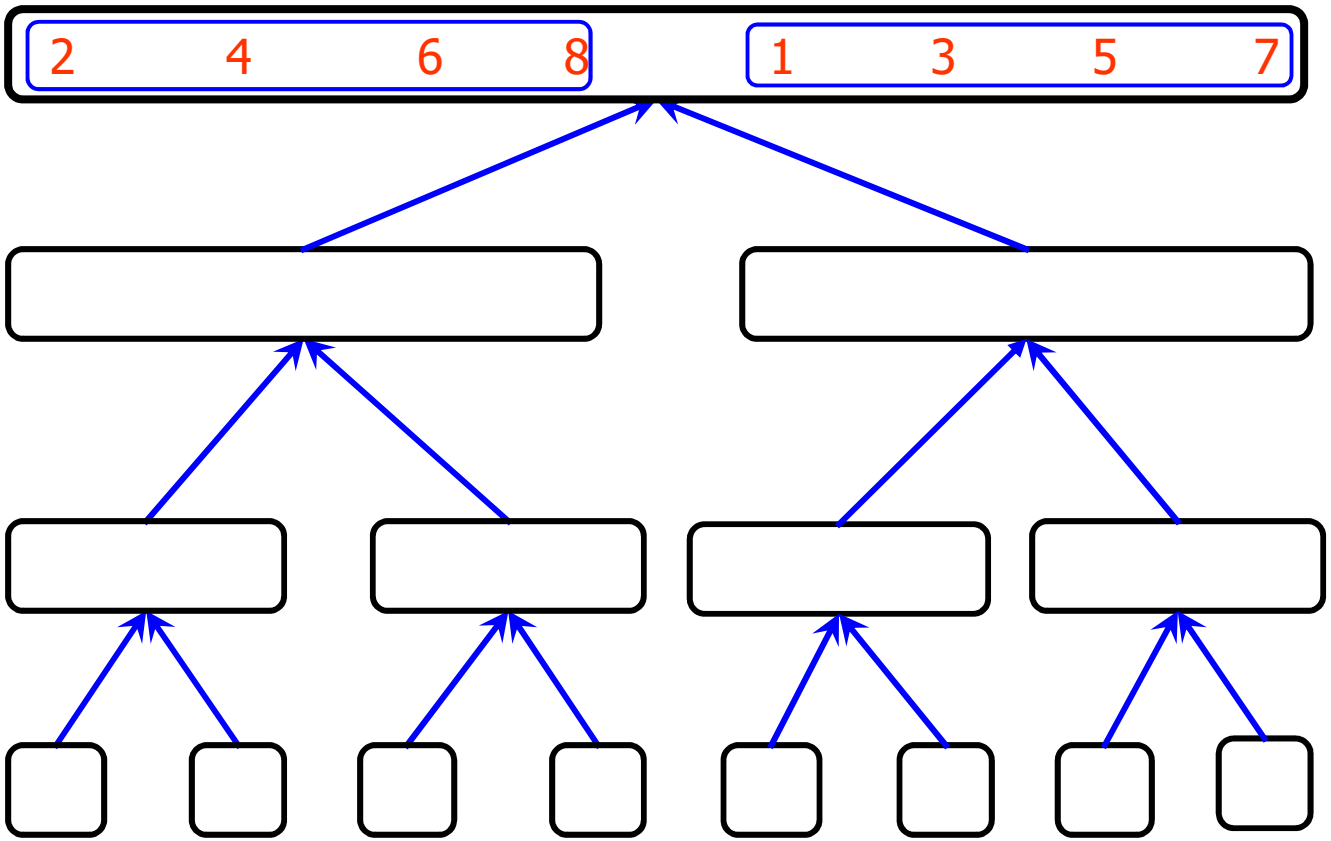


Merge-Sort Execution Example



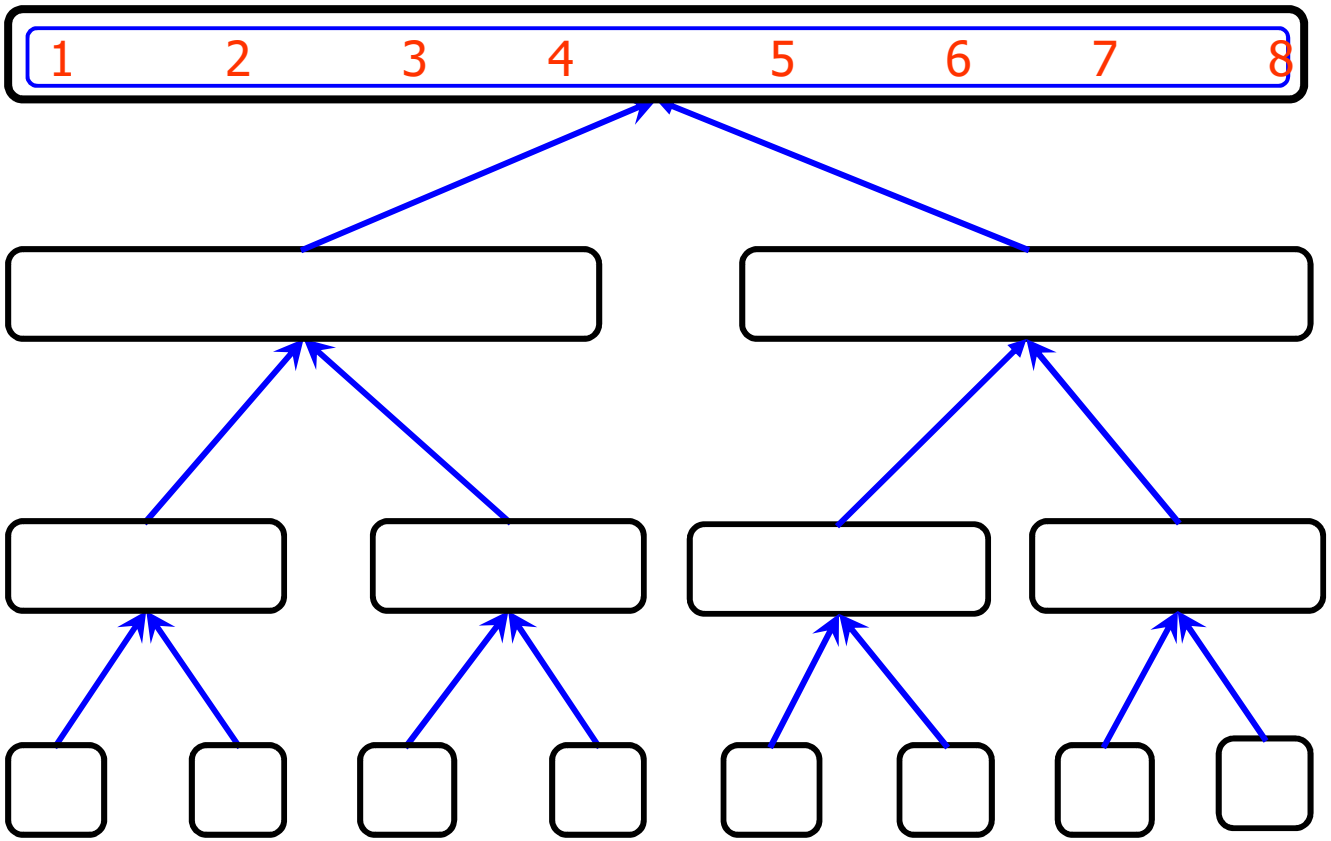
Merge-Sort Execution Example

Recursive call return



Merge-Sort Execution Example

merge



MERGE SORT ALGORITHM

PROCEDURE MERGE_SORT(A,LOW,HIGH,N): Vector A contains maximum N elements. Integer variables LOW and HIGH refers to first and last element of vector A respectively.

1. [Divide the List in smallest part]

If LOW<HIGH

MID ← (LOW + HIGH) / 2

MERGE_SORT(A,LOW,MID,N)

MERGE_SORT(A,MID+1,HIGH,N)

MERGE(A,LOW,MID,HIGH)

2. [Finished]

RETURN

PROCEDURE MERGE(A,LOW,MID,HIGH): **LOW and MID are integer variables containing indexes of two sub arrays. The lists pointed by LOW and MID are merged and pointer to the beginning of the merged list is returned.**

---

1. **[Initialize variables]**

I ← LOW

J ← MID

K ← LOW

2. **[Compare corresponding elements and output the smallest]**

```
WHILE I < MID AND J<=HIGH
        IF ( A[I] <= A[J] )
    TEMP[K] ← A[I]
                I ← I + 1
                K ← K + 1
        ELSE
   TEMP[K] ← A[J]
                J ← J + 1
                K ← K + 1
```

3. [Copy remaining unprocessed elements in output area]

```
IF I >= MID
 WHILE J<=HIGH
        {TEMP[K] ← A[J]
        J ← J + 1
        K ← K + 1 }
ELSE
 WHILE I < MID
        {TEMP[K] ← A[I]
        I ← I + 1
        K ← K + 1}
```

4. [Copy elements of temporary vector into original vector]

```
FOR I = LOW TO HIGH
    A[ I ] ← TEMP[ I ]
```

5. [Finished]

```
RETURN
```

Analysis of mergesort

Let T(N) denote the worst-case running time of mergesort to sort N numbers.

Assume that N is a power of 2.

- Divide step: O(1) time
- Conquer step: 2 T(N/2) time
- Combine step: O(N) time

Recurrence equation:

$T(1) = 1$ if $N=1$

$T(N) = 2T(N/2) + N$ if $N>1$

Analysis: solving recurrence (Substitution)

$$T(N) = 2T(\frac{N}{2}) + N$$

$$= 2(2T(\frac{N}{4}) + \frac{N}{2}) + N$$

$$= 4T(\frac{N}{4}) + 2N$$

$$= 4(2T(\frac{N}{8}) + \frac{N}{4}) + 2N$$

$$= 8T(\frac{N}{8}) + 3N = \square$$

$$= 2^k T(\frac{N}{2^k}) + kN$$

$$T(N) = 2^k T(\frac{N}{2^k}) + kN$$

$$= N + N\log N$$

$$= O(N\log N)$$

Solve the following recurrence relation using Master's theorem-

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

- **Solution-**
- We write the given recurrence relation as $T(n) = 2T(n/2) + n$.
- This is because in the general form, we have θ for function f(n) which hides constants in it.
- Now, we can easily apply Master's theorem.
- We compare the given recurrence relation with $T(n) = aT(n/b) + \theta (n^k \log^p n)$.
- Then, we have- $a = 2$ $b = 2$ $k = 1$ $p = 0$
- Now, $a = 2$ and $b^k = 2^1 = 2$.
- Clearly, $a = b^k$.
- So, we follow case-02.
- Since $p = 0$, so we have-
- $T(n) = \theta (n^{\log_b a}.\log^{p+1} n)$
- $T(n) = \theta (n^{\log_2 2}.\log^{0+1} n)$
- $T(n) = \theta (n^1.\log^1 n)$
- Thus, **$T(n) = \theta (n \log n)$**

Quicksort Algorithm

Given an array of *n* elements (e.g., integers):

- If array only contains one element, return
- Else
  - pick one element to use as *pivot.*
  - Partition elements into two sub-arrays:
    - Elements less than or equal to pivot
    - Elements greater than pivot
  - Quicksort two sub-arrays
  - Return results

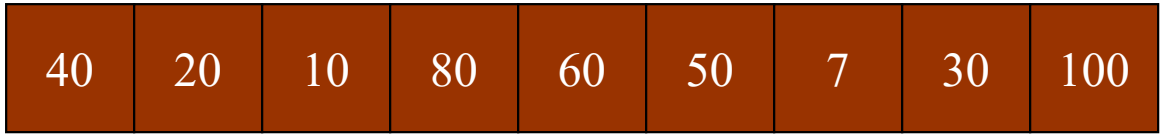
Quicksort

Partition the array between indices low and high – select a pivot and move all values that are less than or equal to the pivot to the left of the pivot index and all values greater than the pivot to the right of the pivot index.

1. Step 1 – randomly choose a pivot index between low and high. Alternatively select low as the pivot index. (It is as good as any choice)
2. Step 2 – move the pivot to position A[low]
3. Step 3 – set partition indices i = low + 1, j = high - 1

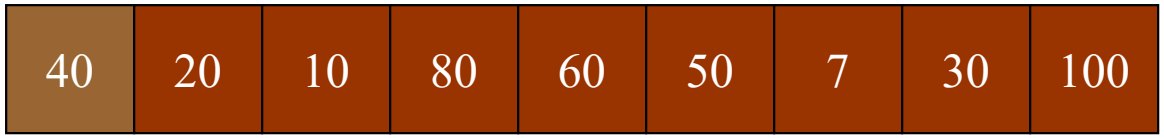
Example

We are given array of n integers to sort:



Pick Pivot Element

There are a number of ways to pick the pivot element. In this example, we will use the first element in the array:

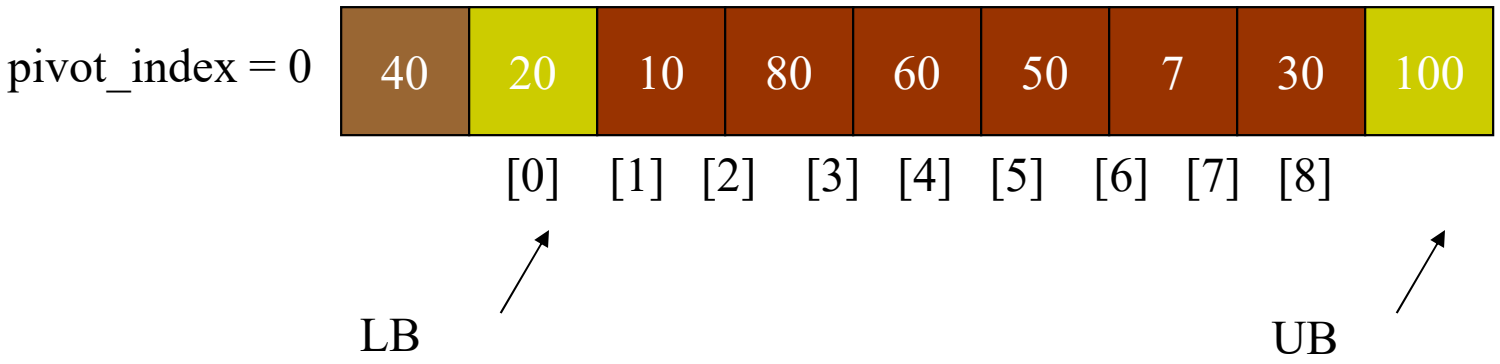


Partitioning Array

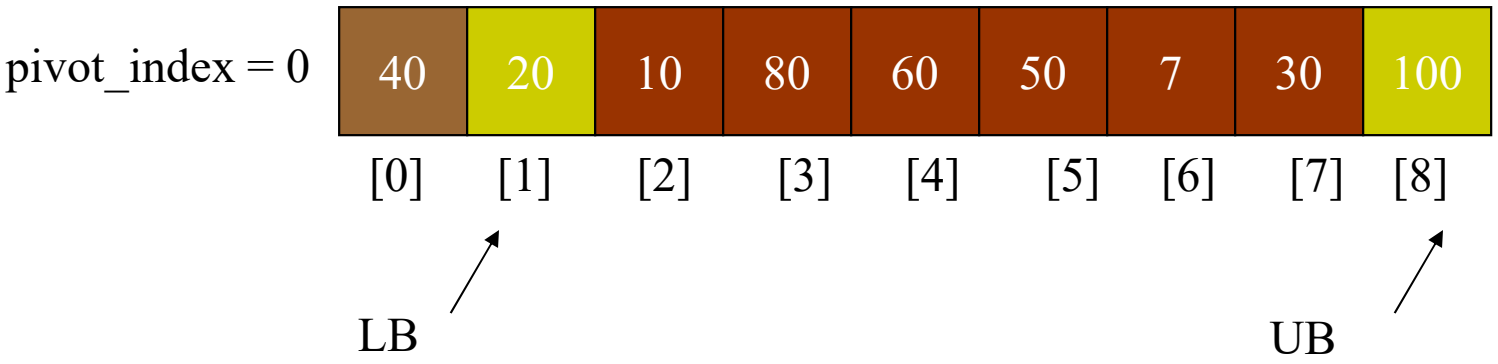
Given a pivot, partition the elements of the array such that the resulting array consists of:

1. One sub-array that contains elements >= pivot
2. Another sub-array that contains elements < pivot

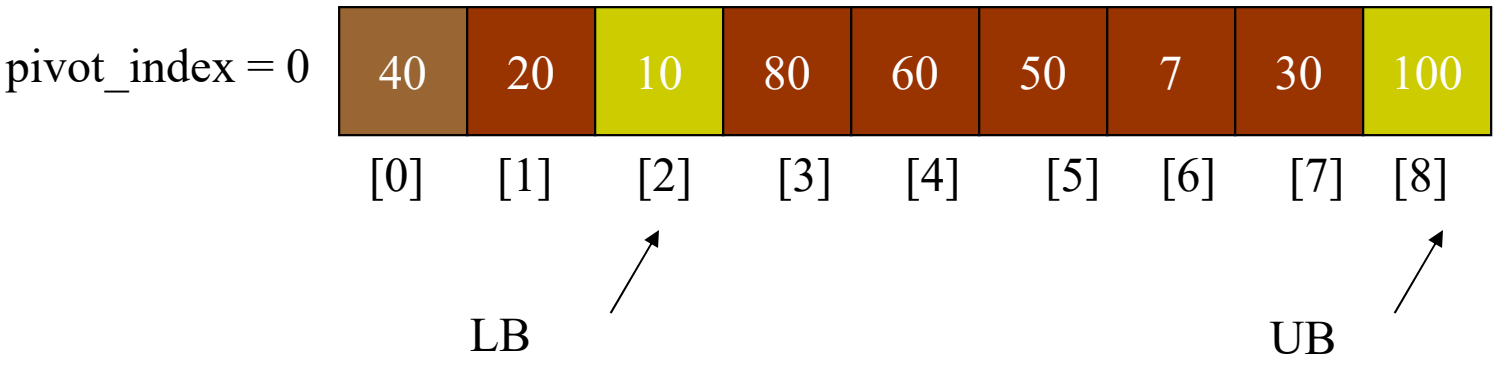
The sub-arrays are stored in the original data array.


pivot_index = 0
40
20
10
80
60
50
7
30
100
[0]
[1]
[2]
[3]
[4]
[5]
[6]
[7]
[8]
LB
UB

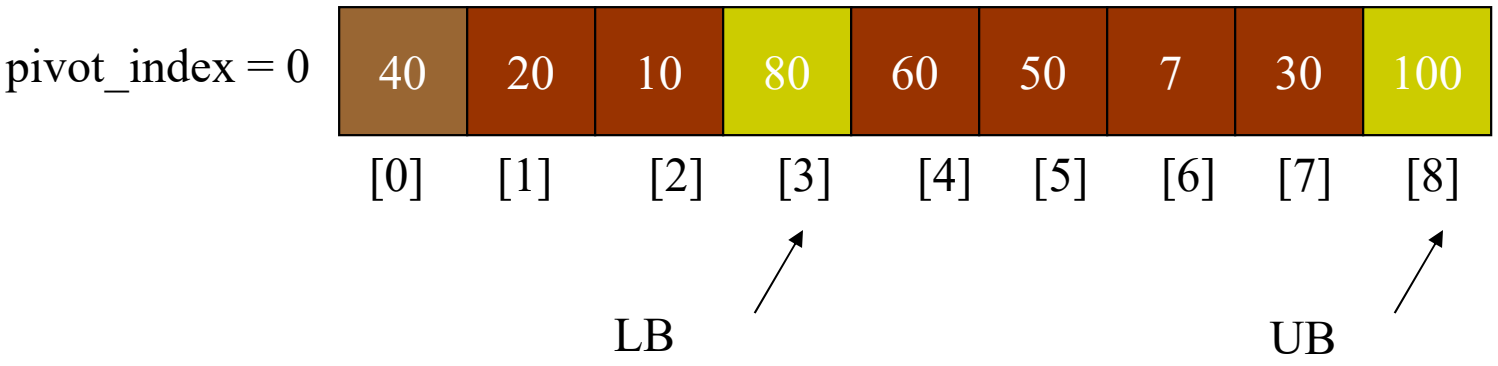
1. While data[LB] <= data[pivot]
   ++LB



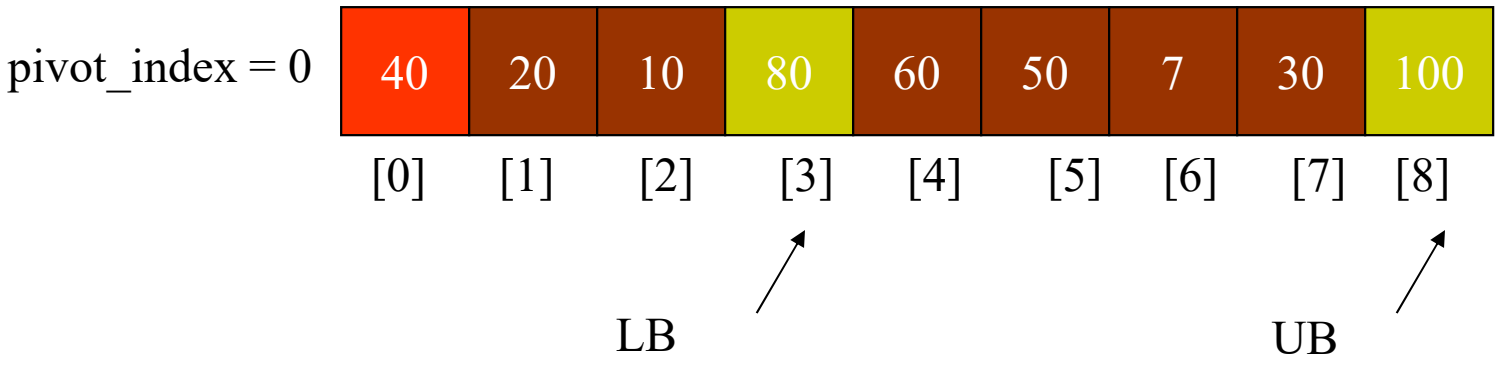
1. While data[LB] <= data[pivot]
   ++LB



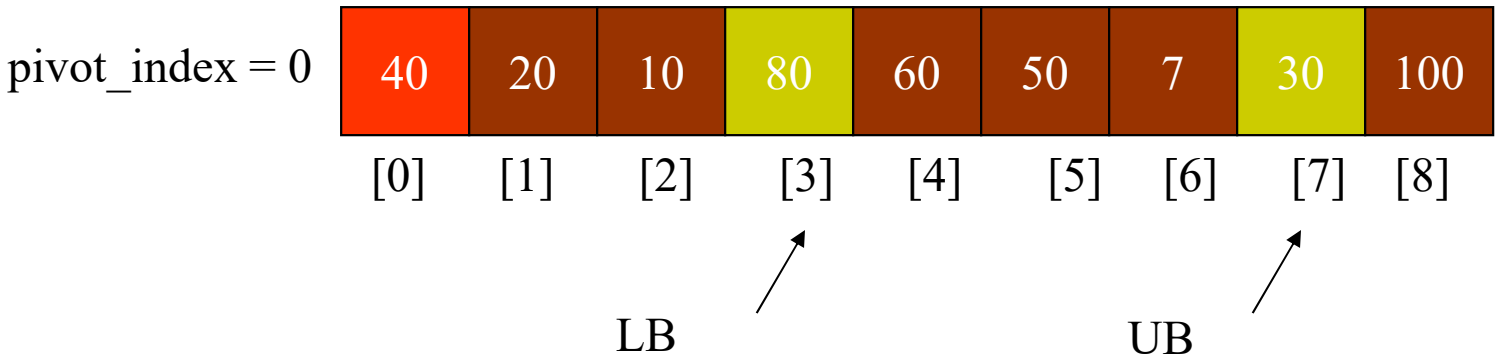
1. While data[LB] <= data[pivot]
   ++LB

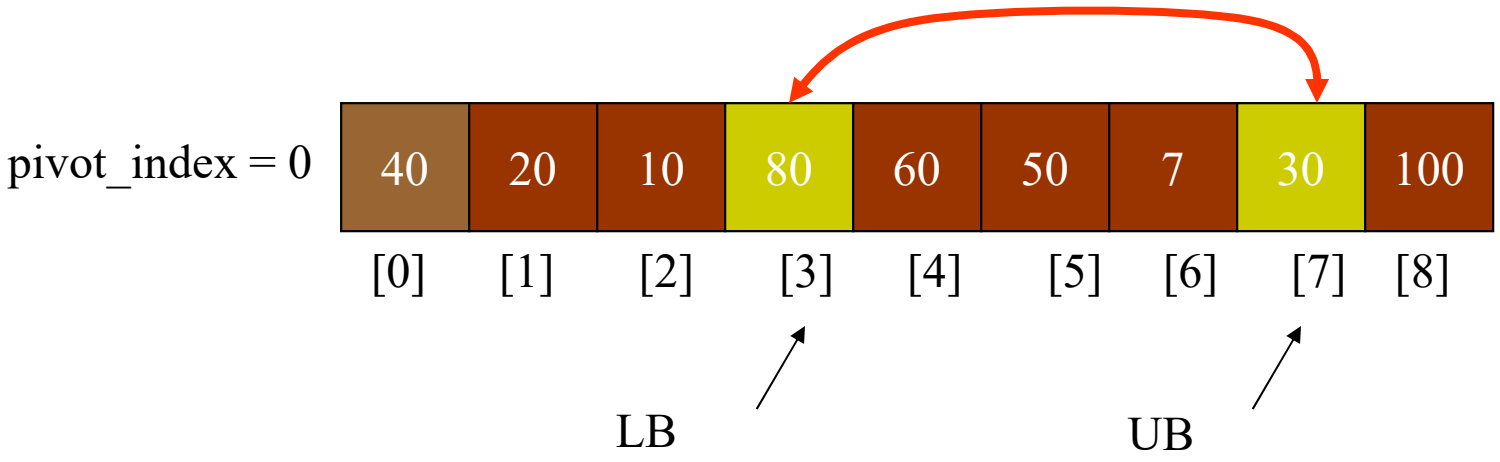


1. While data[LB] <= data[pivot]
   ++LB
2. While data[UB] > data[pivot]
   --UB

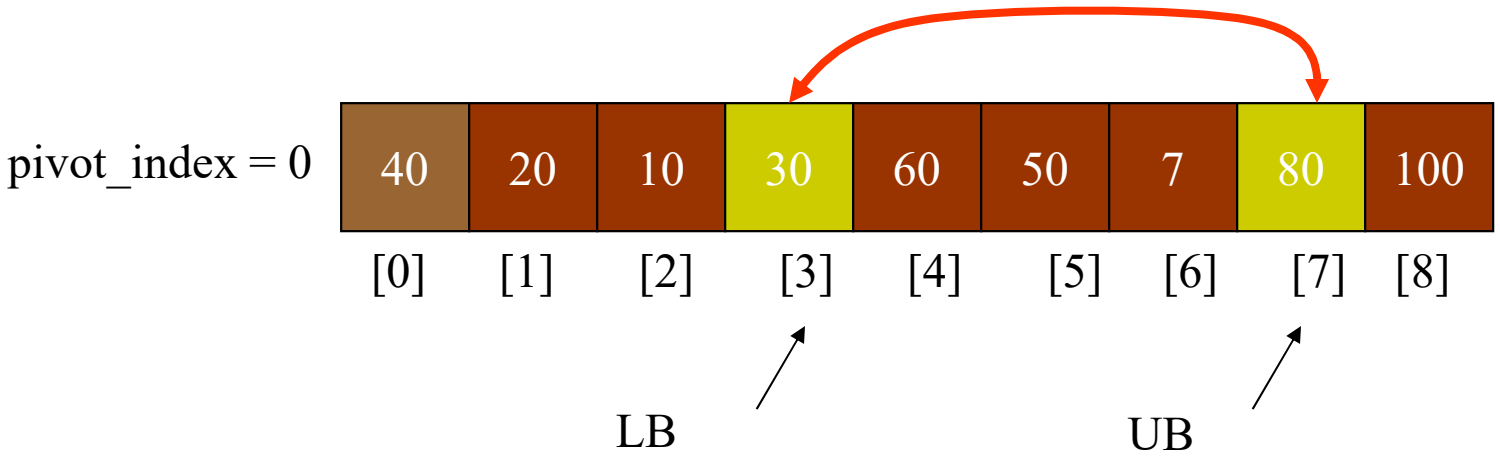


1. While data[LB] <= data[pivot]
   ++LB
2. While data[UB] > data[pivot]
   -- UB

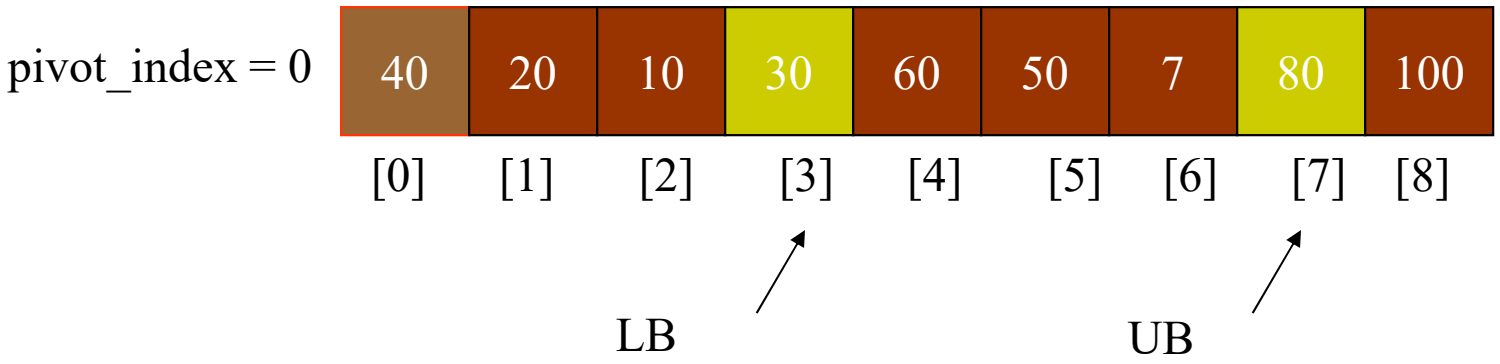



pivot_index = 0
40
20
10
80
60
50
7
30
100
[0]
[1]
[2]
[3]
[4]
[5]
[6]
[7]
[8]
LB
UB

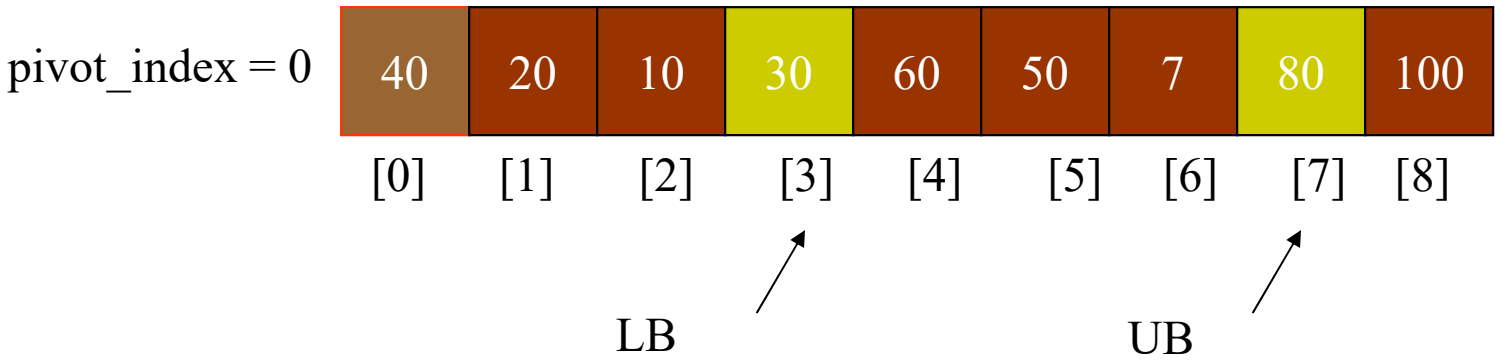
1. While data[LB] <= data[pivot]
   ++LB
2. While data[UB] > data[pivot]
   --UB
3. If LB< UB
   swap data[LB] and data[UB]



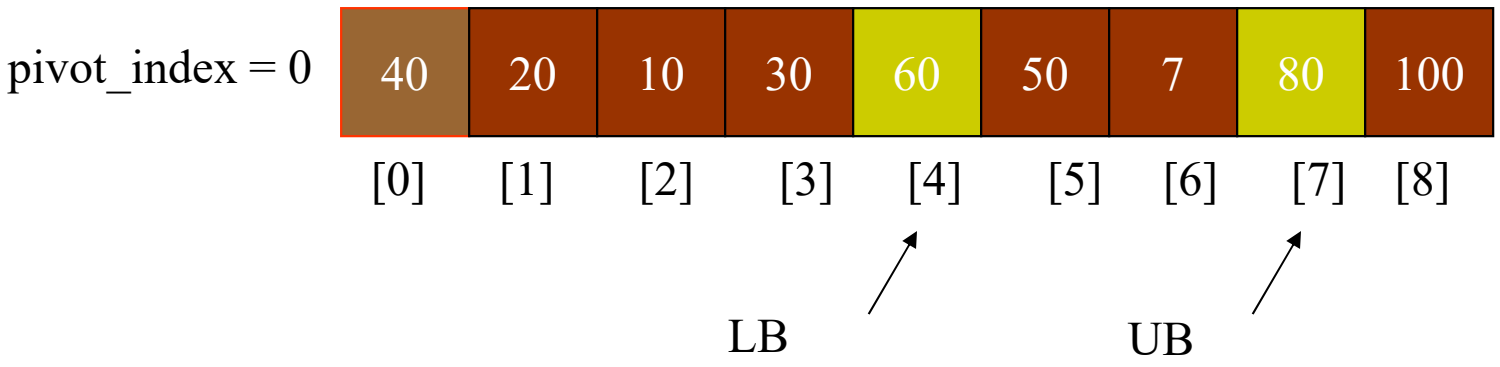
1. While data[LB] <= data[pivot]
   ++LB
2. While data[UB] > data[pivot]
   --UB
3. If LB < UB
   swap data[LB] and data[UB]
4. While LB > UB, go to 1.



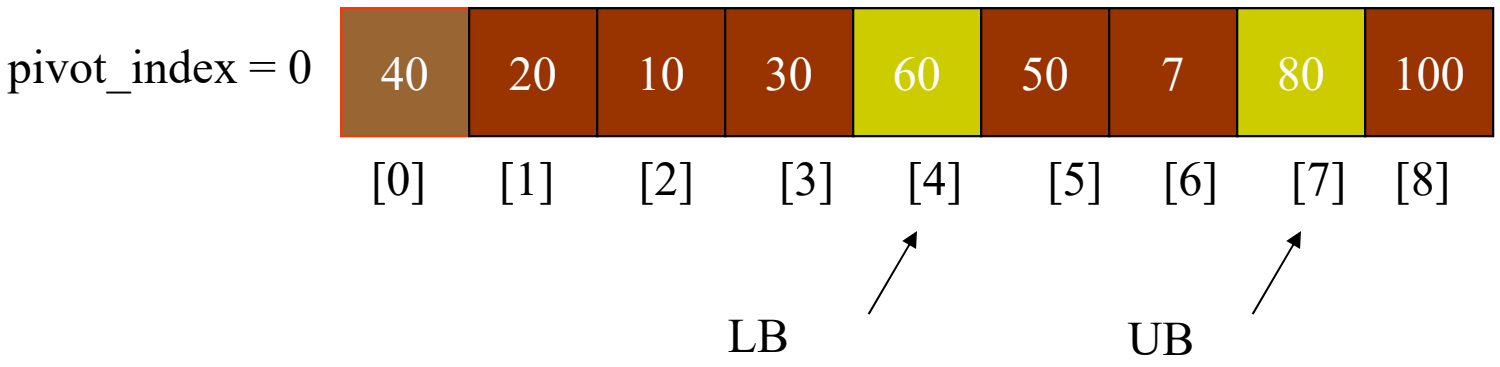
1. While data[LB] <= data[pivot]
   ++LB
2. While data[UB] > data[pivot]
   --UB
3. If LB < UB
   swap data[LB] and data[UB]
4. While LB > UB, go to 1.

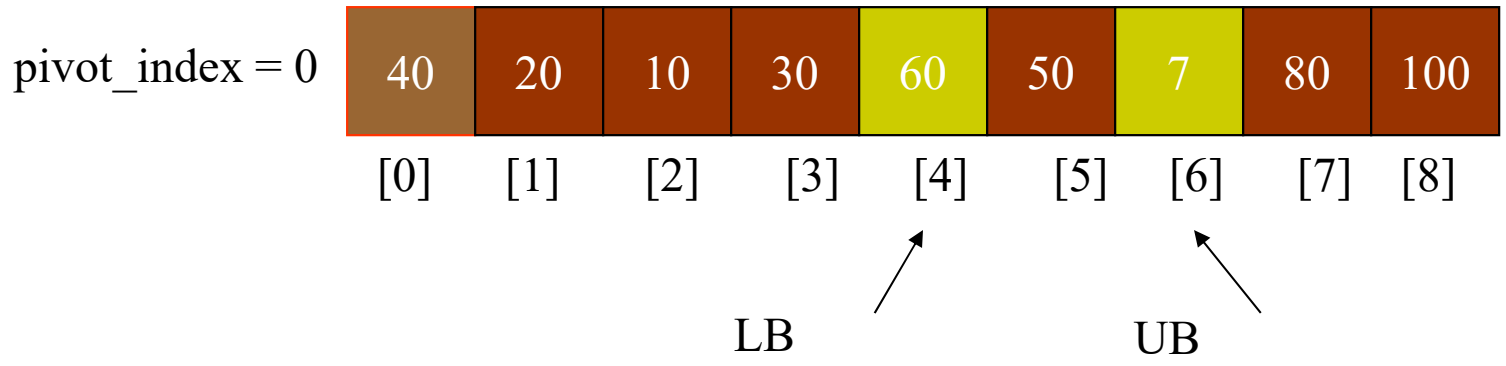


1. While data[LB] <= data[pivot]
   ++LB
2. While data[UB] > data[pivot]
   --UB
3. If LB < UB
   swap data[LB] and data[UB]
4. While LB > UB, go to 1.

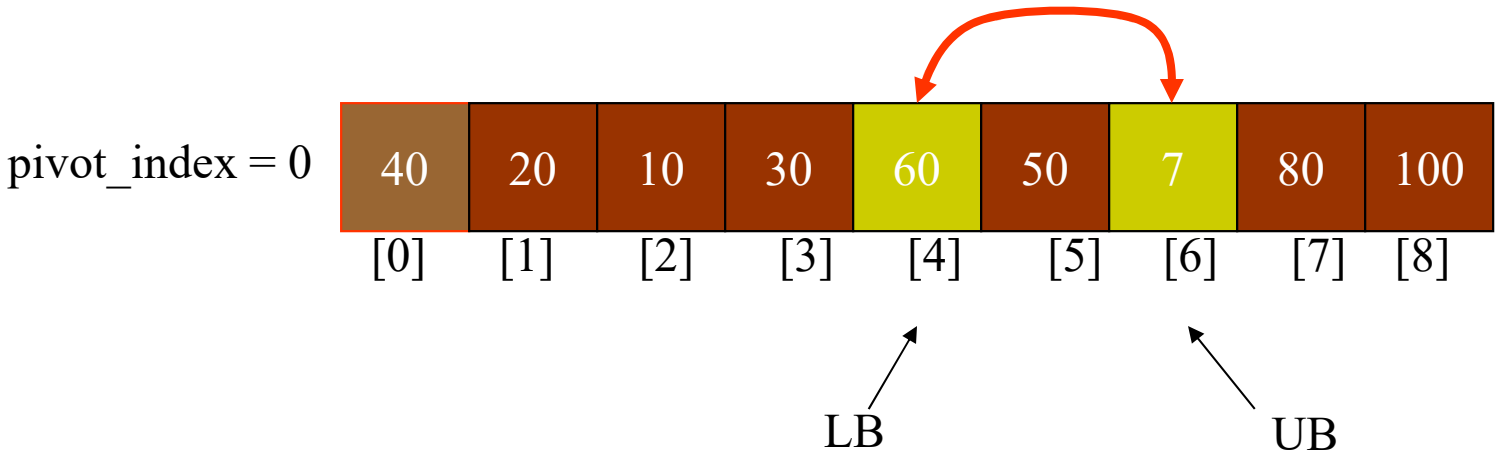


1. While data[LB] <= data[pivot]
   ++LB
2. While data[UB] > data[pivot]
   --UB
3. If LB < UB
   swap data[LB] and data[UB]
4. While LB > UB, go to 1.

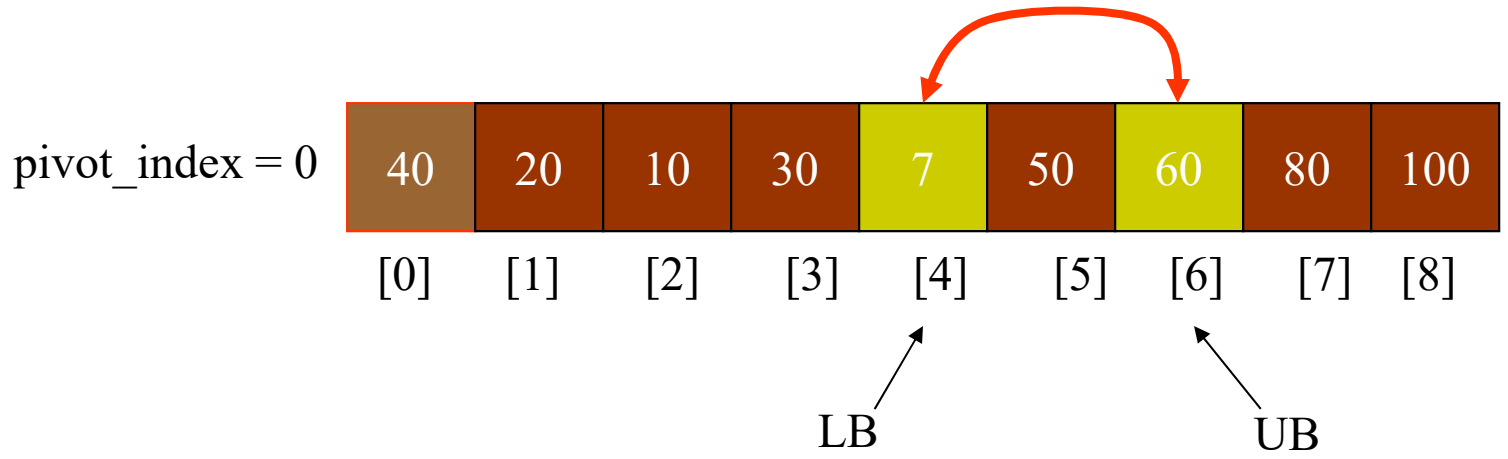



pivot_index = 0
40
20
10
30
60
50
7
80
100
[0]
[1]
[2]
[3]
[4]
[5]
[6]
[7]
[8]
LB
UB

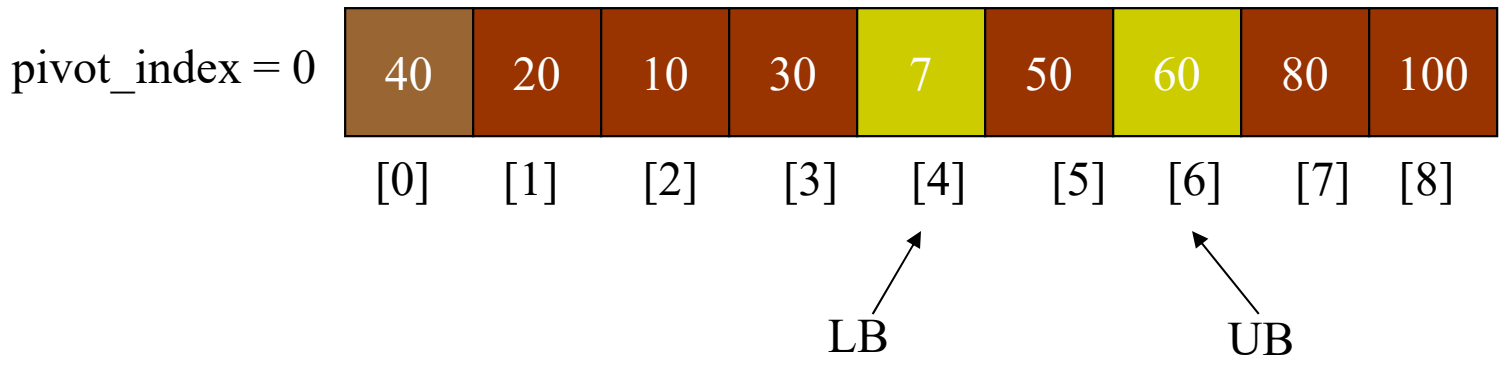
1. While data[LB] <= data[pivot]
   ++LB
2. While data[UB] > data[pivot]
   --UB
3. If LB < UB
   swap data[LB] and data[UB]
4. While LB > UB, go to 1.



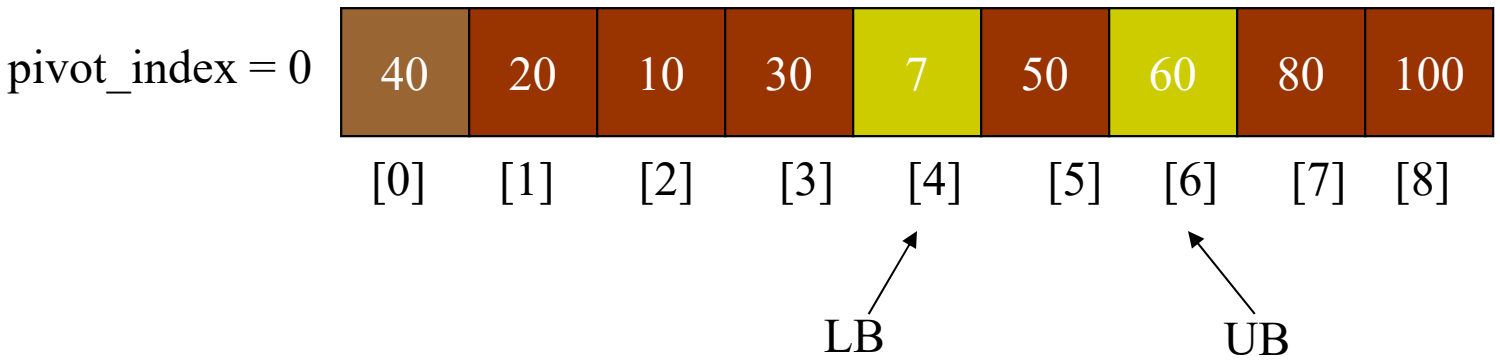
1. While data[LB] <= data[pivot]
   ++LB
2. While data[UB] > data[pivot]
   --UB
3. If LB < UB
   swap data[LB] and data[UB]
4. While LB > UB, go to 1.



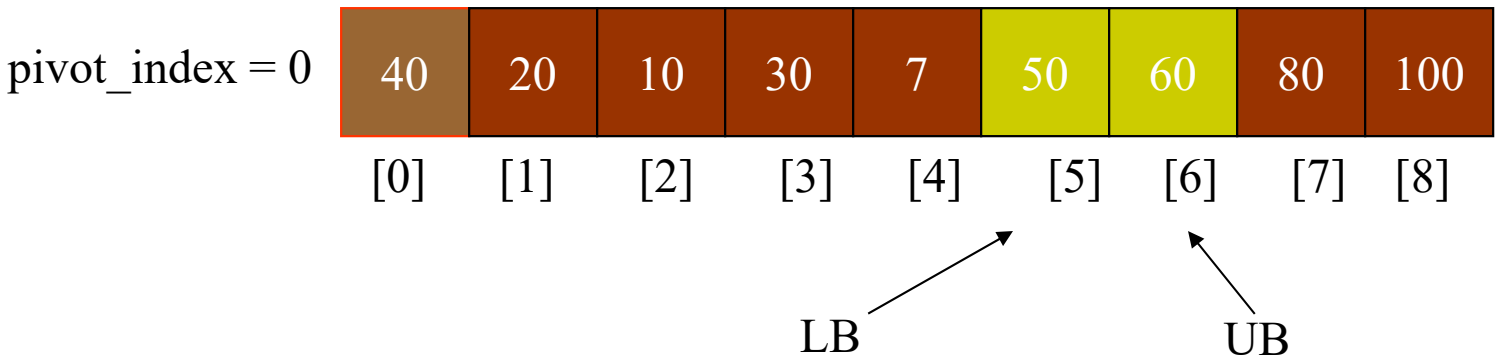
1. While data[LB] <= data[pivot]
   ++LB
2. While data[UB] > data[pivot]
   --UB
3. If LB < UB
   swap data[LB] and data[UB]
4. While LB > UB, go to 1.



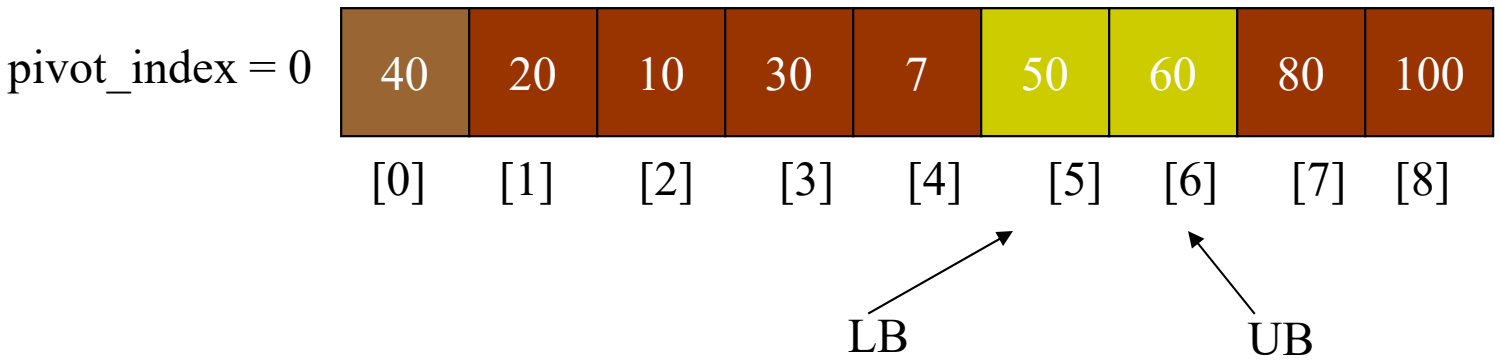
1. While data[LB] <= data[pivot]
   ++LB
2. While data[UB] > data[pivot]
   --UB
3. If LB < UB
   swap data[LB] and data[UB]
4. While LB > UB, go to 1.



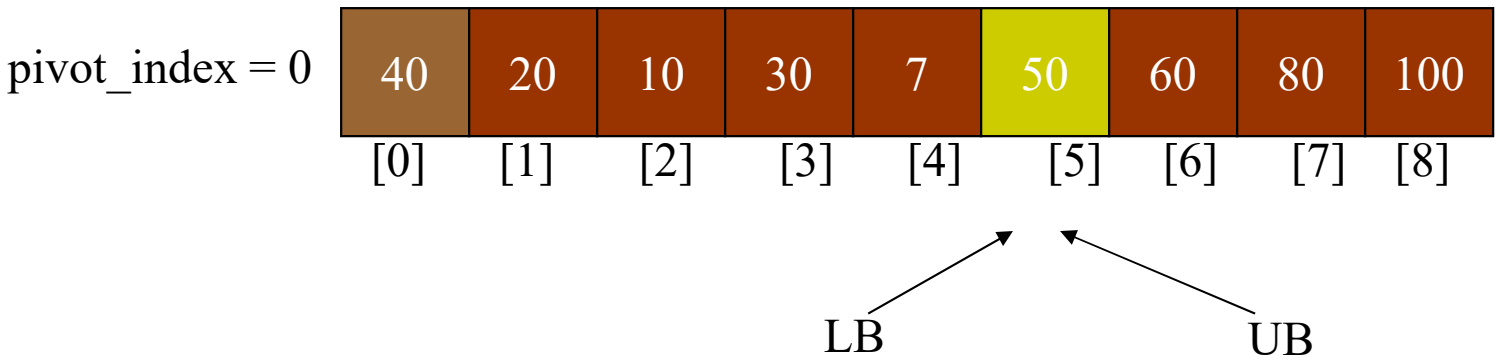
1. While data[LB] <= data[pivot]
   ++LB
2. While data[UB] > data[pivot]
   --UB
3. If LB < UB
   swap data[LB] and data[UB]
4. While LB > UB, go to 1.



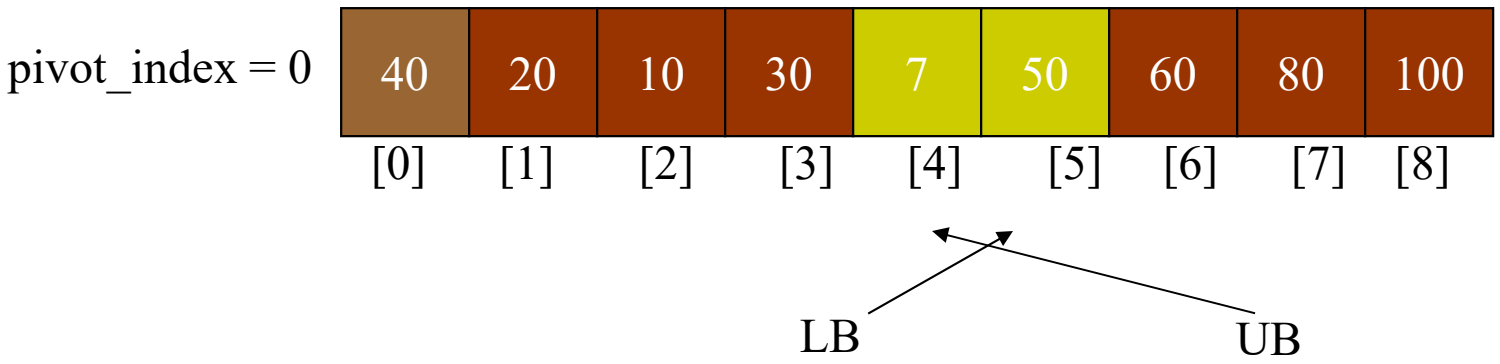
1. While data[LB] <= data[pivot]
   ++LB
2. While data[UB] > data[pivot]
   --UB
3. If LB < UB
   swap data[LB] and data[UB]
4. While LB > UB, go to 1.



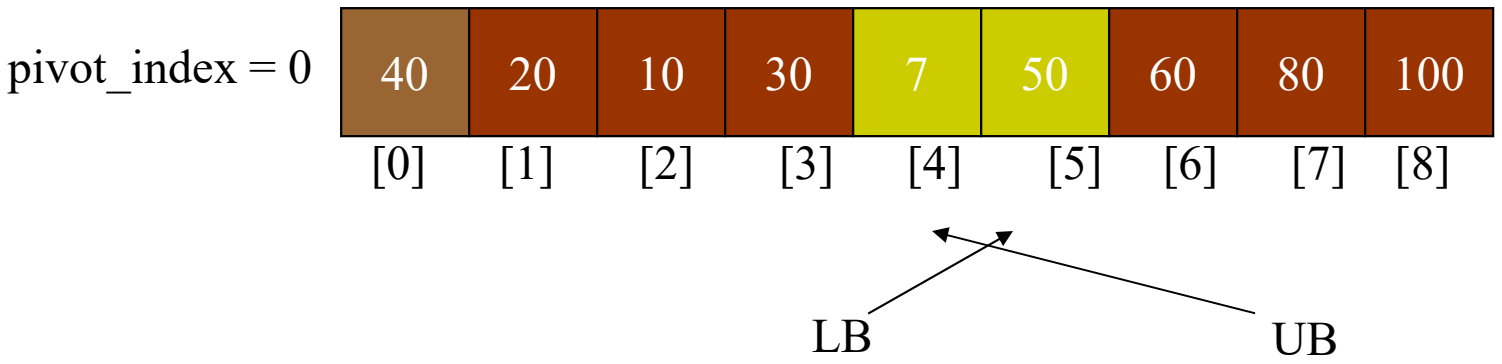
1. While data[LB] <= data[pivot]
   ++LB
2. While data[UB] > data[pivot]
   --UB
3. If LB < UB
   swap data[LB] and data[UB]
4. While LB > UB, go to 1.



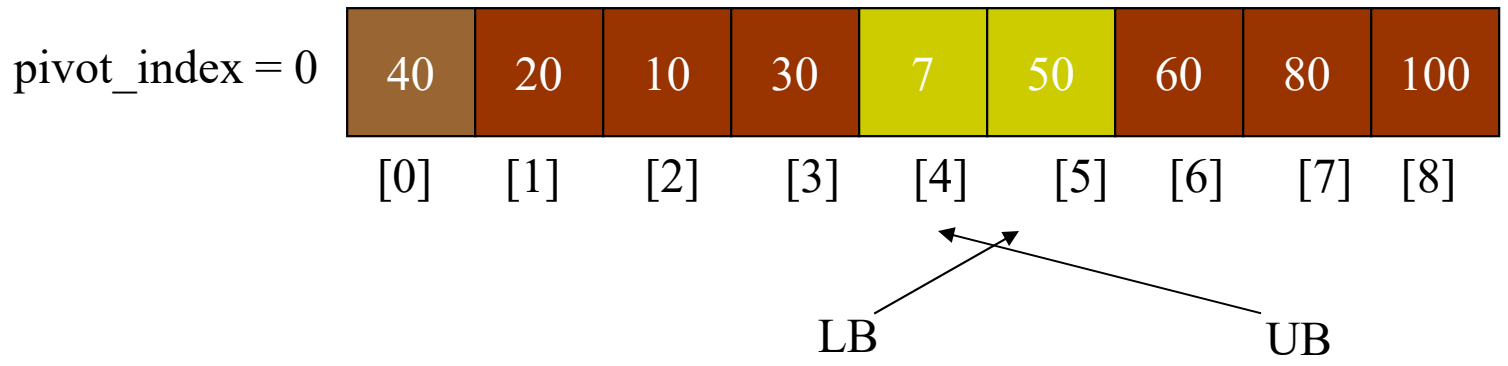
1. While data[LB] <= data[pivot]
   ++LB
2. While data[UB] > data[pivot]
   --UB
3. If LB < UB
   swap data[LB] and data[UB]
4. While LB > UB, go to 1.



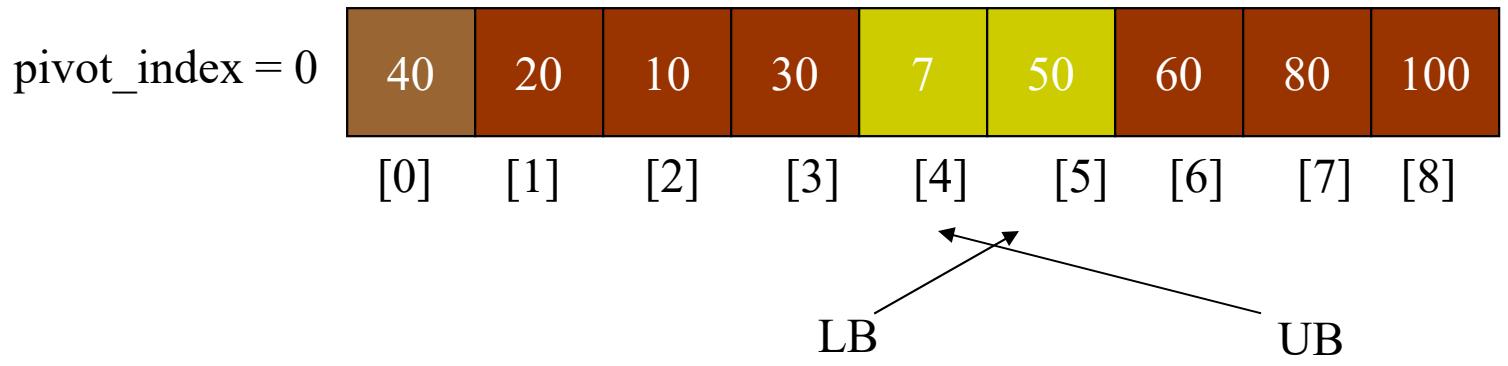
1. While data[LB] <= data[pivot]
   ++LB
2. While data[UB] > data[pivot]
   --UB
3. If LB < UB
   swap data[LB] and data[UB]
4. While LB > UB, go to 1.



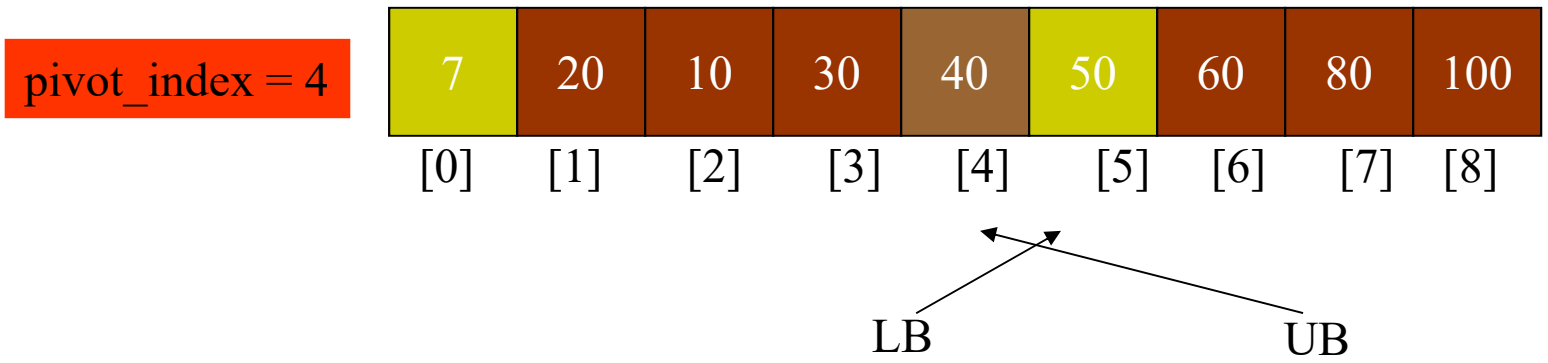
1. While data[LB] <= data[pivot]
   ++LB
2. While data[UB] > data[pivot]
   --UB
3. If LB < UB
   swap data[LB] and data[UB]
4. While LB > UB, go to 1.



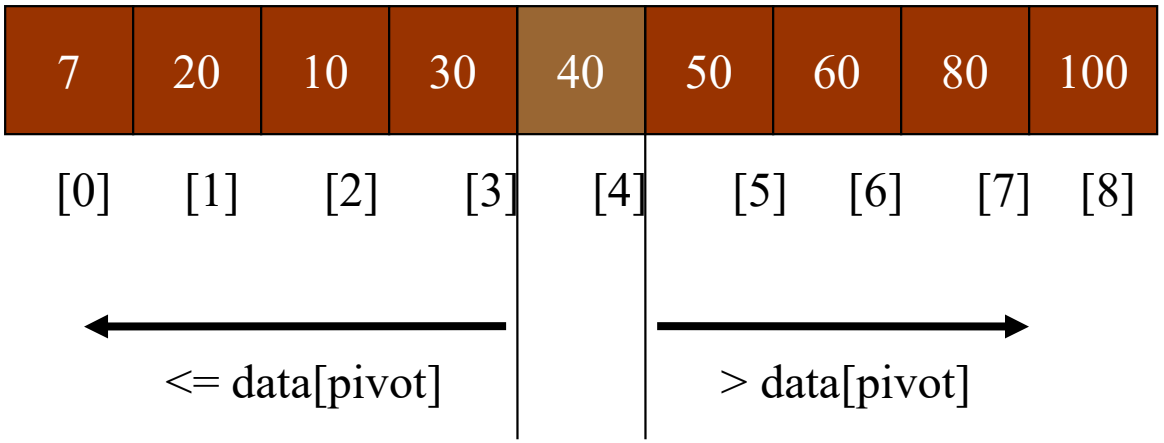
1. While data[LB] <= data[pivot]
   ++LB
2. While data[UB] > data[pivot]
   --UB
3. If LB < UB
   swap data[LB] and data[UB]
4. While LB > UB, go to 1.
5. Swap data[UB] and data[pivot_index]



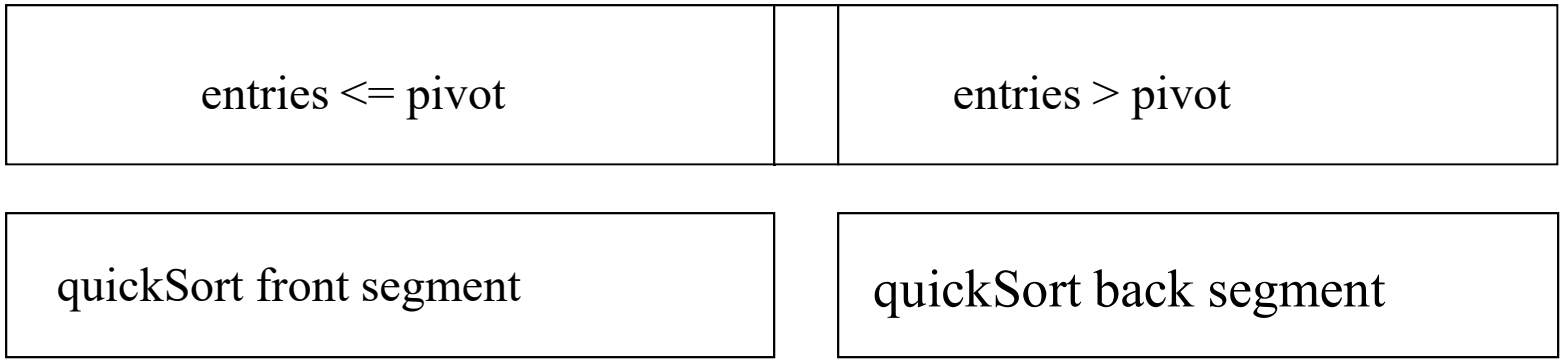
1. While data[LB] <= data[pivot]
   ++LB
2. While data[UB] > data[pivot]
   --UB
3. If LB < UB
   swap data[LB] and data[UB]
4. While LB > UB, go to 1.
5. Swap data[UB] and data[pivot_index]



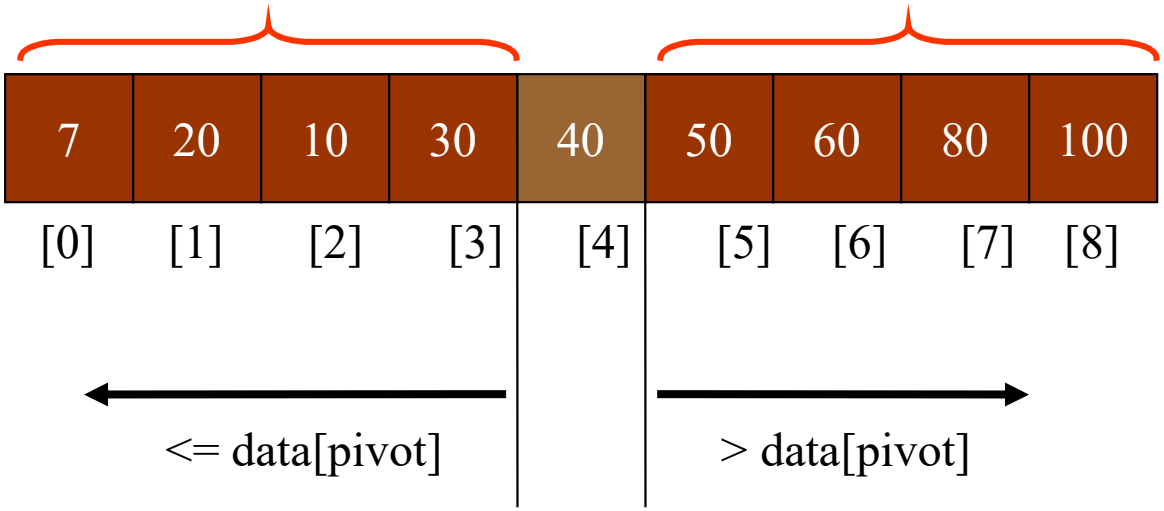
Partition Result

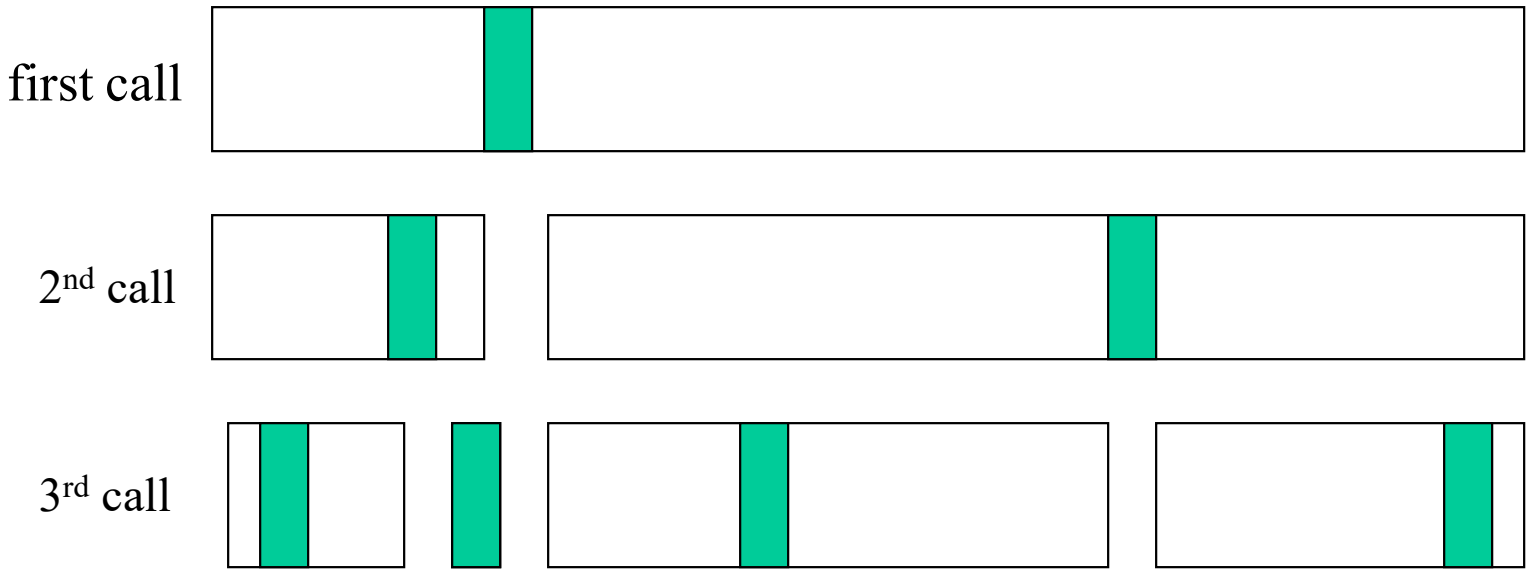


When a call to partition terminates, the pivot separates all the entries that are less than or equal to the pivot from those entries greater than the pivot.



Recursion: Quicksort Sub-arrays





Each time the (sub) list is partitioned, exactly one value (the pivot) is placed in its correct position.

If the list is equally divided during each partitioning, it will require about lg (n) calls to quickSort to produce a sorted list.

ALGORITHM-QUICK SORT

Procedure QUICK_SORT (K,LB,UB):

This recursive procedure sorts vector K of N elements in ascending order. integer variable LB & UB denotes the lower and upper bound of the sub table being processed. I & J are used to select certain keys during the processing of each subtable. PIVOT contains the pivot point which is being placed in its final position within subtable. FLAG is a logical variable. When FLAG becomes false the input subtable has been partitioned into two disjointed parts.

1. IF LB<UB
THEN
I=split(K,LB,UB)
CALL QUICK_SORT(K , LB, I-1) (Sort first subtable)
CALL QUICK_SORT(K , I+1,UB) (Sort second subtable)

3. [Finished]
RETURN

Function split(K,LB,UB):

1. [Initialize]
   PIVOT := K[LB]
   I := LB
   J := UB
   FLAG:=True
2. [Perform Sort]
   Repeat thru step While FLAG
3. WHILE K[I] < PIVOT AND I<=UB
   I ← I + 1
4. WHILE K[J] > PIVOT
   J ← J – 1
5. IF I < J
   swap(K[I] , K[J]) (Interchange records)
   ELSE
   FLAG ← False
6. swap(K[LB], K[J]) (Interchange records)

7. [Finished]
   RETURN J

Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- What is best case running time?

Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- What is best case running time?
  - Recursion:
    1. Partition splits array in two sub-arrays of size n/2
    2. Quicksort each sub-array

Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- What is best case running time?
  - Recursion:
    1. Partition splits array in two sub-arrays of size n/2
    2. Quicksort each sub-array
  - Depth of recursion tree?

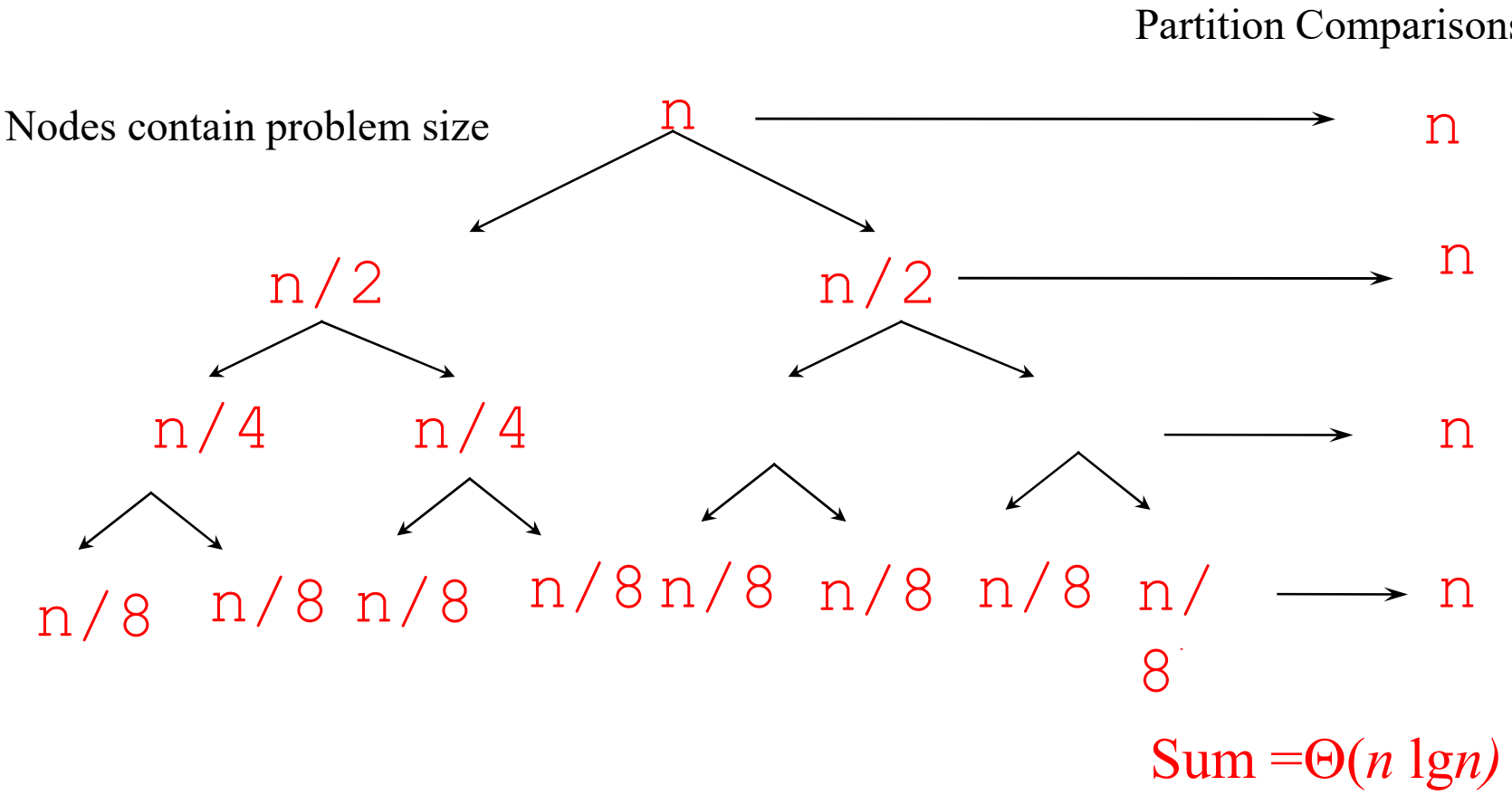
Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- What is best case running time?
  - Recursion:
    1. Partition splits array in two sub-arrays of size n/2
    2. Quicksort each sub-array
  - Depth of recursion tree? $O(\log_2 n)$

Quicksort Analysis

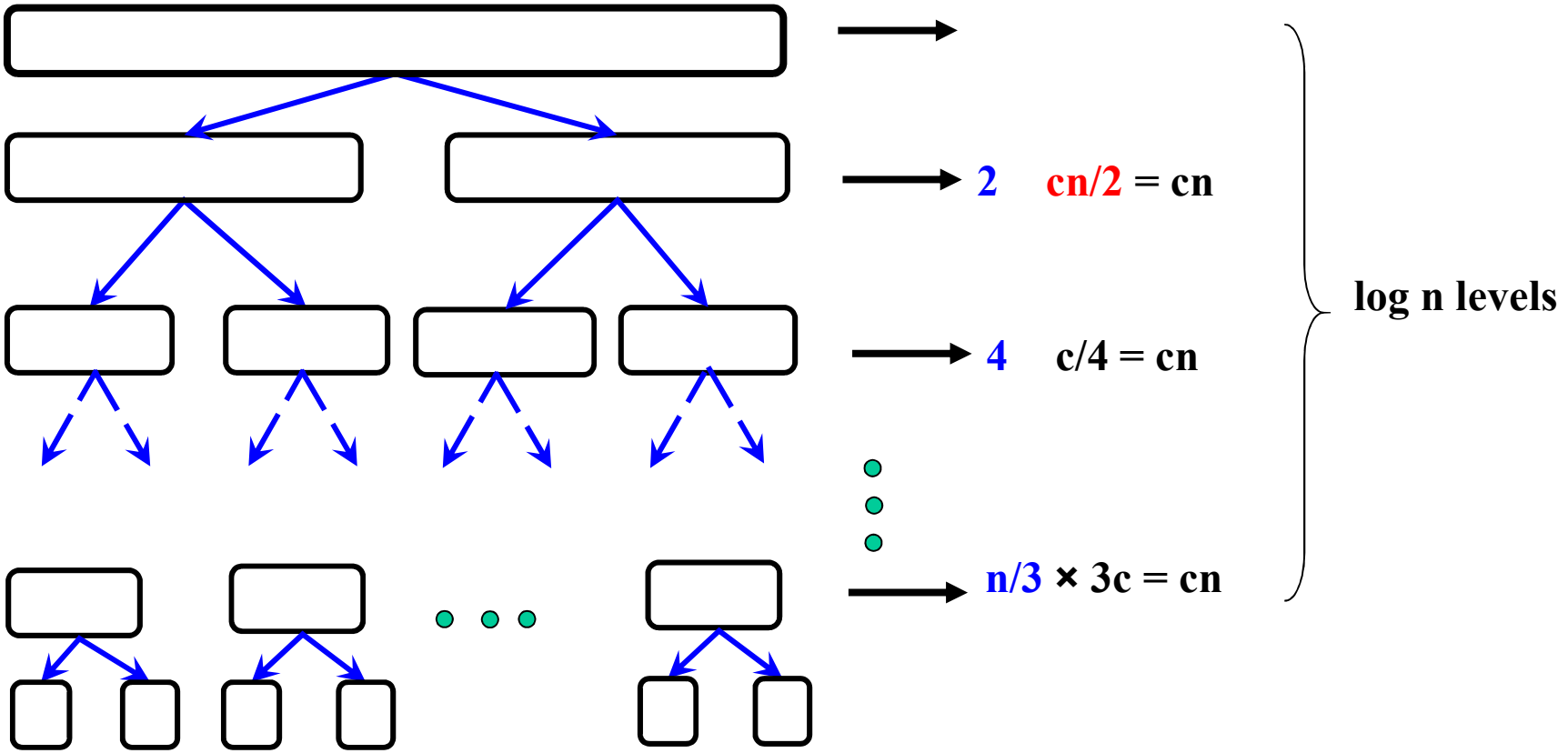
- Assume that keys are random, uniformly distributed.
- What is best case running time?
  - Recursion:
    1. Partition splits array in two sub-arrays of size n/2
    2. Quicksort each sub-array
  - Depth of recursion tree? $O(\log_2 n)$
  - Number of accesses in partition?

Recursion Tree for Best Case



Quick-Sort: Best Case

- Even Partition



Total time: O(nlogn)

Best Case

Recurrence equation:

$$T(N) = 2T(N/2) + N$$

Which can be solved using master's theorem or substitution.

Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- What is best case running time?
  - Recursion:
    1. Partition splits array in two sub-arrays of size n/2
    2. Quicksort each sub-array
  - Depth of recursion tree? $O(\log_2 n)$
  - Number of accesses in partition? O(n)

Quicksort Analysis

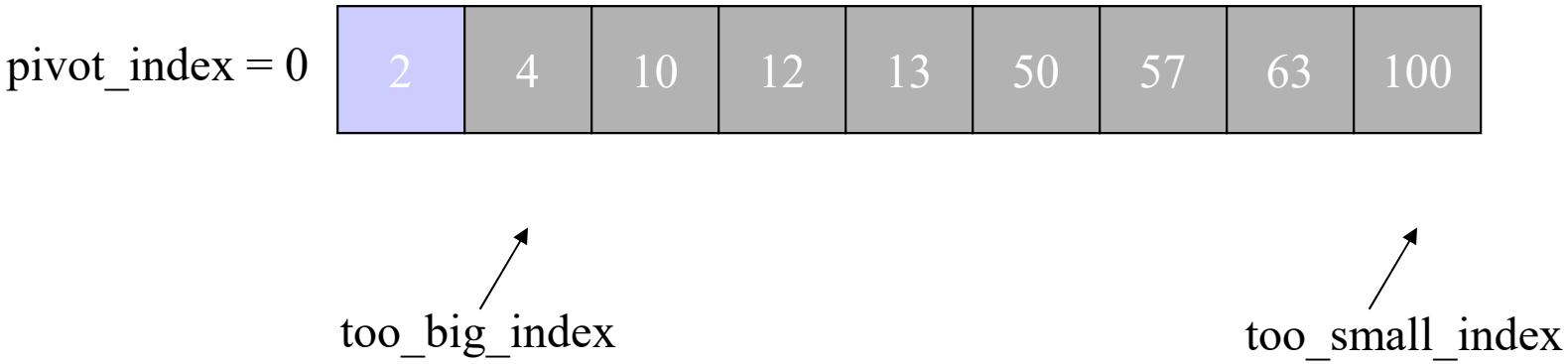
- Assume that keys are random, uniformly distributed.
- Best case running time: $O(n \log_2 n)$

Quicksort Analysis

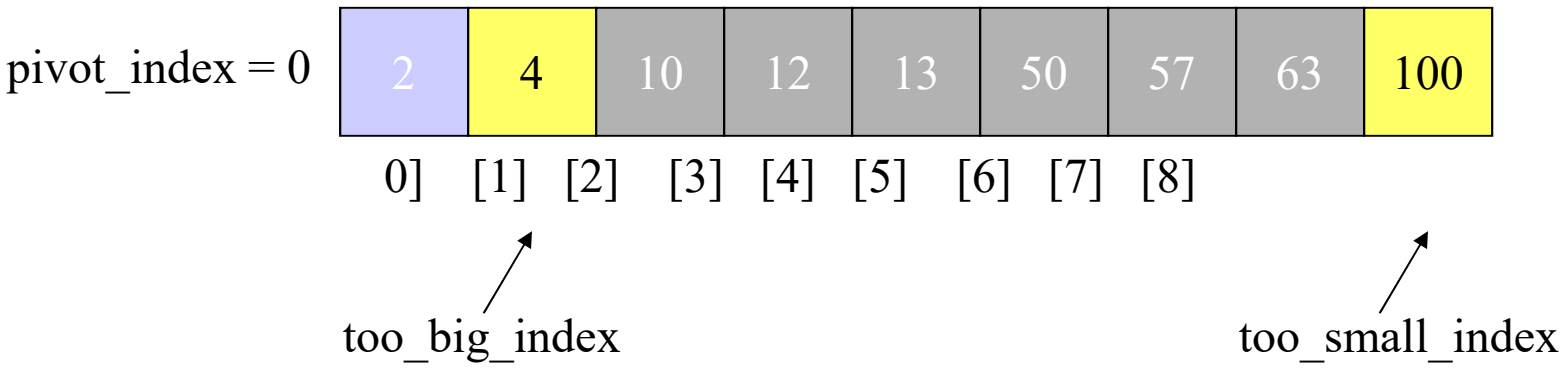
- Assume that keys are random, uniformly distributed.
- Best case running time: $O(n \log_2 n)$
- Worst case running time?
  - **Worst case**: when the pivot does not divide the sequence in two

Quicksort: Worst Case

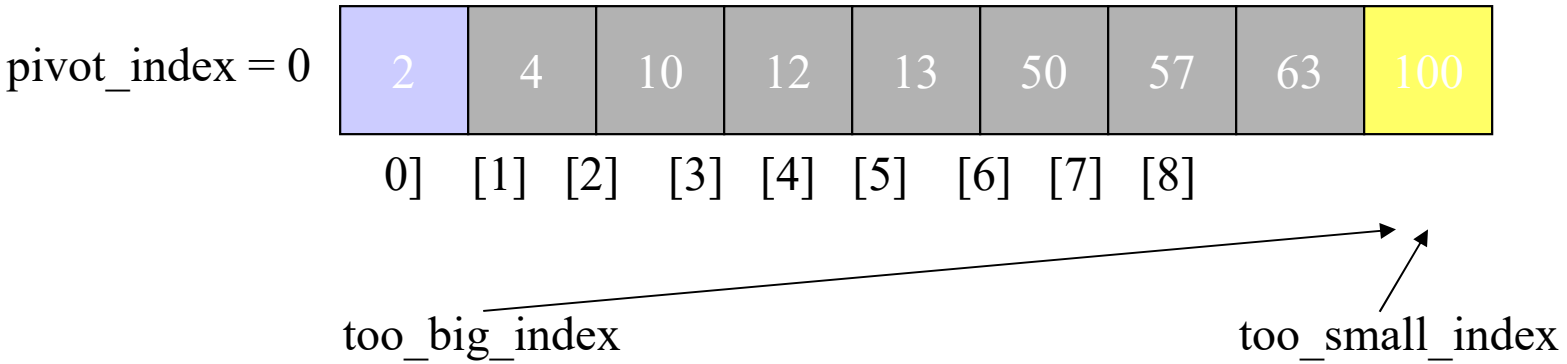
- Assume first element is chosen as pivot.
- Assume we get array that is already in order:



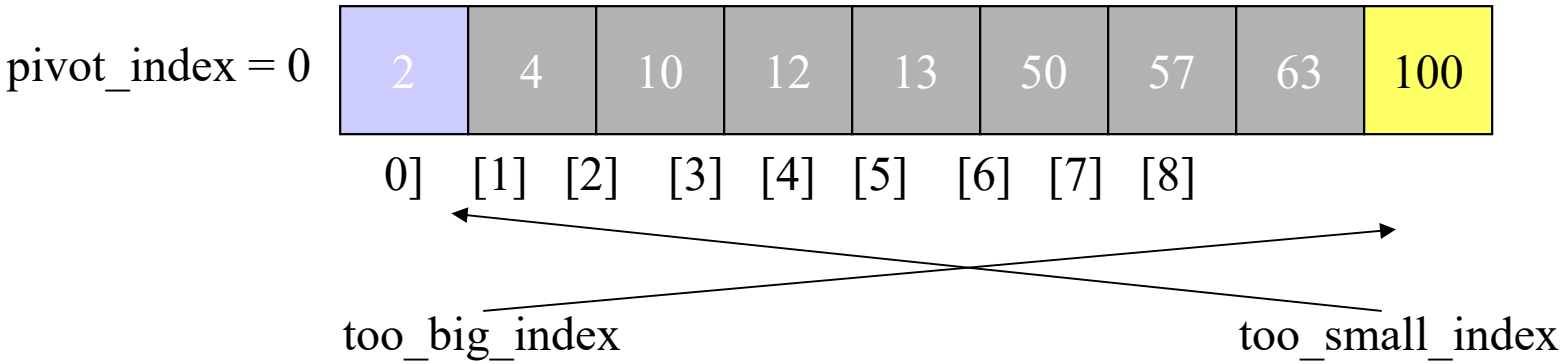
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.
5. Swap data[too_small_index] and data[pivot_index]



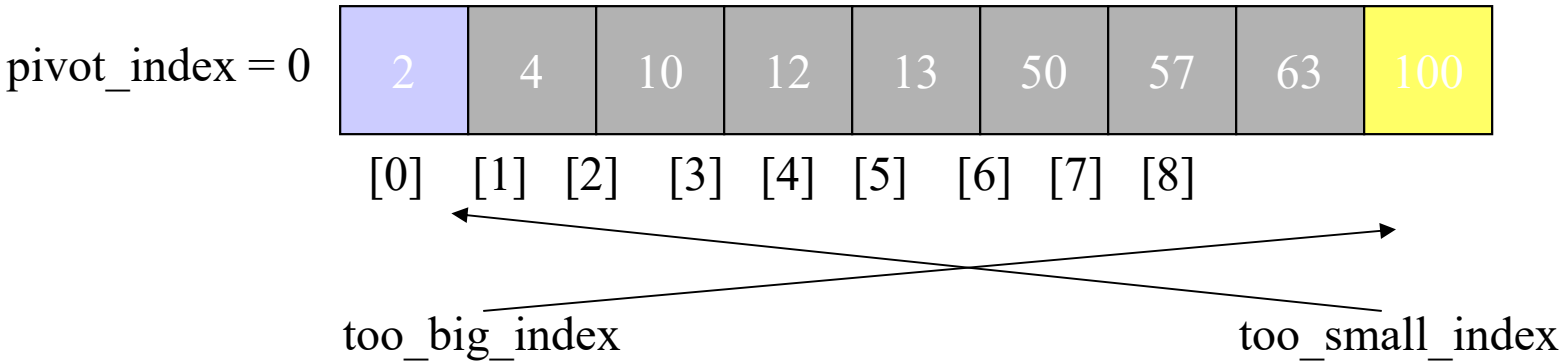
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.
5. Swap data[too_small_index] and data[pivot_index]



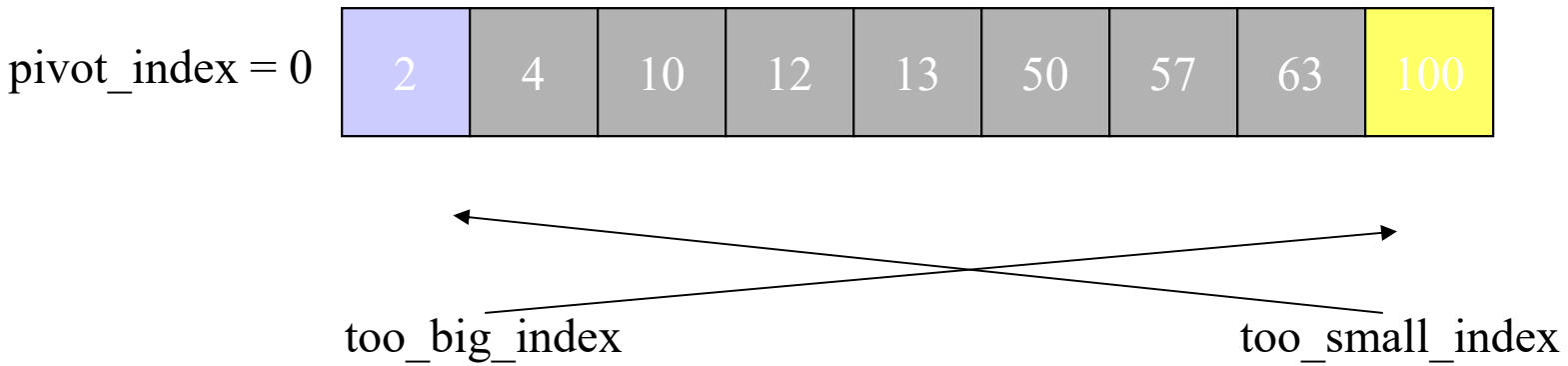
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.
5. Swap data[too_small_index] and data[pivot_index]



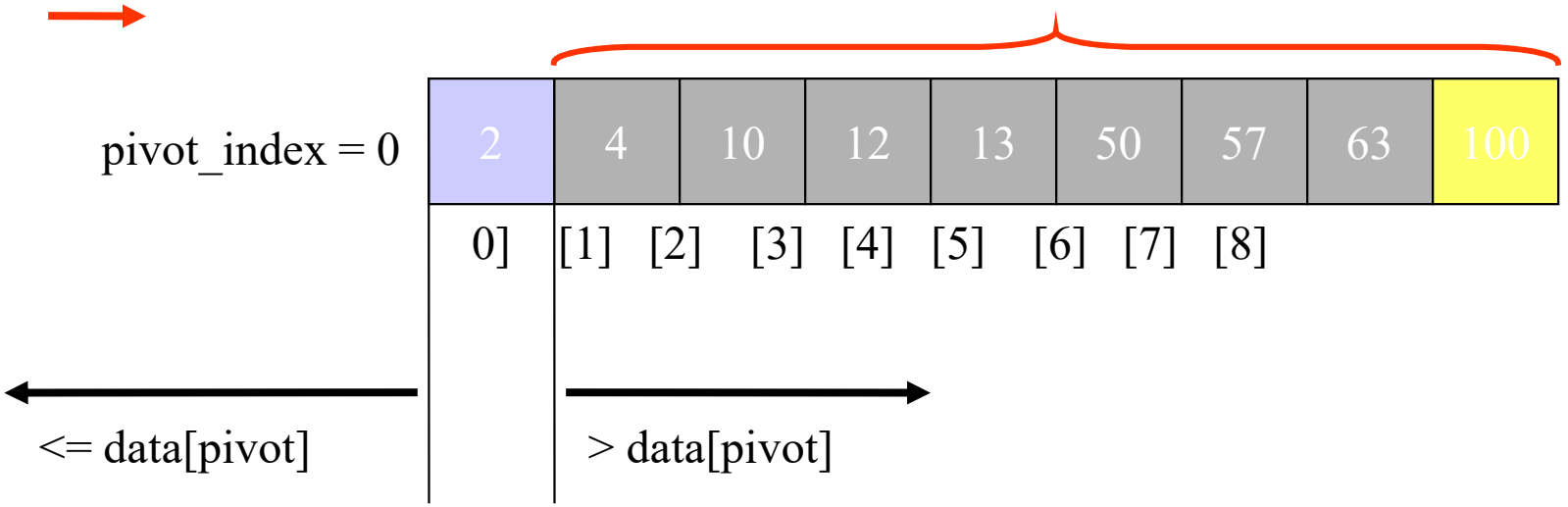
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.
5. Swap data[too_small_index] and data[pivot_index]



1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.
5. Swap data[too_small_index] and data[pivot_index]



1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.
5. Swap data[too_small_index] and data[pivot_index]



Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- Best case running time: $O(n \log_2 n)$
- Worst case running time?
  - Recursion:
    1. Partition splits array in two sub-arrays:
       - one sub-array of size 0
       - the other sub-array of size n-1
    2. Quicksort each sub-array
  - Depth of recursion tree?

Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- Best case running time: $O(n \log_2 n)$
- Worst case running time?
  - Recursion:
    1. Partition splits array in two sub-arrays:
       - one sub-array of size 0
       - the other sub-array of size n-1
    2. Quicksort each sub-array
  - Depth of recursion tree? O(n)

Quicksort Analysis

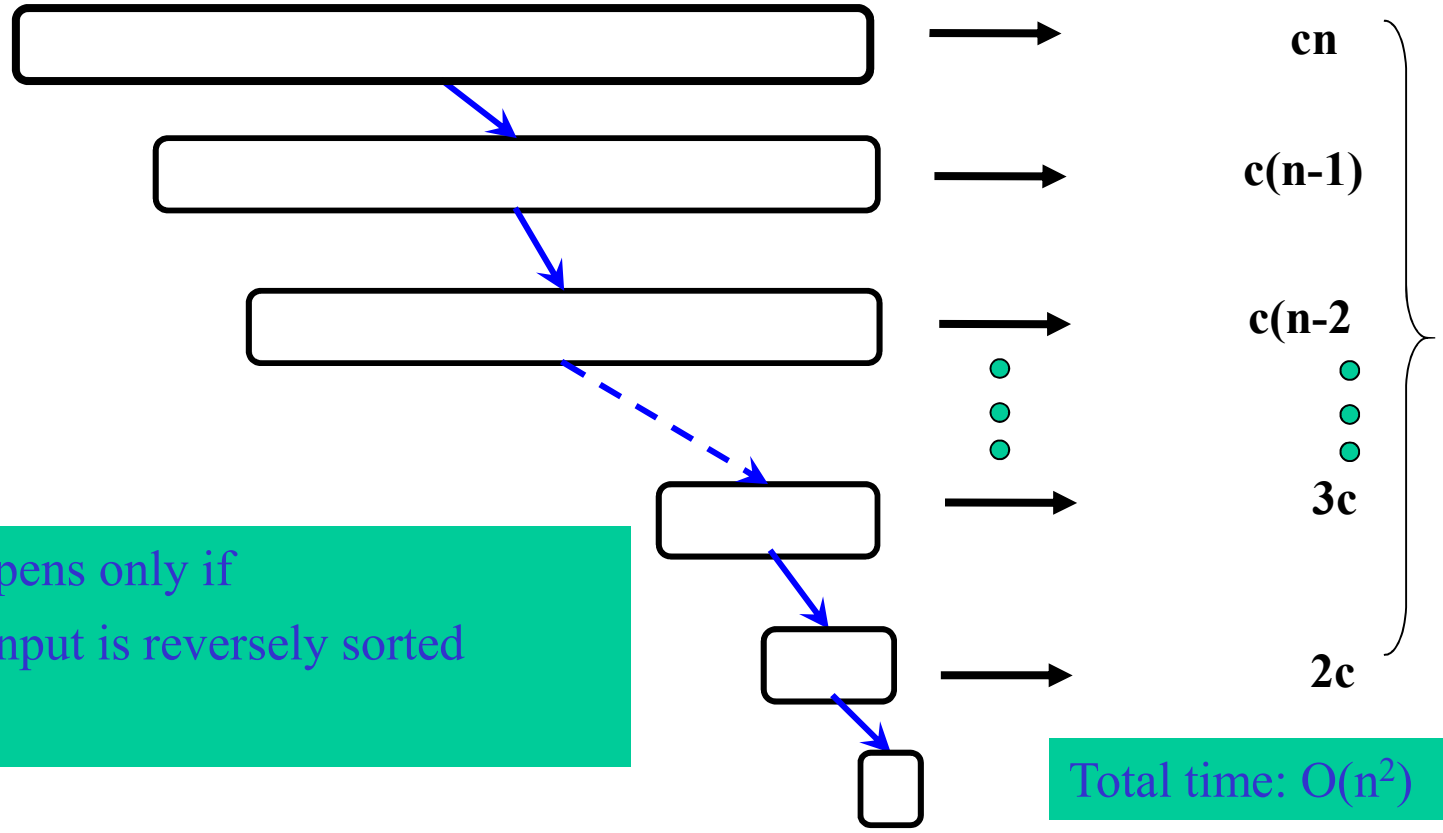
- Assume that keys are random, uniformly distributed.
- Best case running time: $O(n \log_2 n)$
- Worst case running time?
  - Recursion:
    1. Partition splits array in two sub-arrays:
       - one sub-array of size 0
       - the other sub-array of size n-1
    2. Quicksort each sub-array
  - Depth of recursion tree? O(n)
  - Number of accesses per partition?

Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- Best case running time: $O(n \log_2 n)$
- Worst case running time?
  - Recursion:
    1. Partition splits array in two sub-arrays:
       - one sub-array of size 0
       - the other sub-array of size n-1
    2. Quicksort each sub-array
  - Depth of recursion tree? O(n)
  - Number of accesses per partition? O(n)

Quick-Sort: Worst Case

- Unbalanced Partition



Happens only if

- input is reversely sorted

Total time: $O(n^2)$

Worst Case

Recurrence equation:

$$T(N) = T(N-1) + N$$

Solve by substitution method:

$T(N)=T(N-K)+(N-K+1)+(N-K+2)+….+N$

Now, put K=N then

$T(0)+1+2+3+….+N$

As we know that $=1+2+3+……+N$

$= N(N+1)/2$

$=N^2$

Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- Best case running time: $O(n \log_2 n)$
- Worst case running time: $O(n^2)$!!!

Quicksort Code

```
Quicksort(A, p, r)
{
     if (p < r)
     {
          q = Partition(A, p, r);
          Quicksort(A, p, q);
          Quicksort(A, q+1, r);
     }
}
```

Partition Code

```
Partition(A, p, r)
    x = A[p];
    i = p - 1;
    j = r + 1;
    while (TRUE)
        repeat
            j--;
        until A[j] <= x;
        repeat
            i++;
        until A[i] >= x;
        if (i < j)
            Swap(A, i, j);
        else
            return j;
```

Illustrate on
A = {5, 3, 2, 6, 4, 1, 3, 7};

What is the running time of `partition()`?

Analyzing Quicksort: Average Case

- For simplicity, assume:
  - All inputs distinct (no repeats)
  - Slightly different **`partition()`** procedure
    - partition around a random element, which is not included in subarrays
    - all splits (0:n-1, 1:n-2, 2:n-3, … , n-1:0) equally likely
- *What is the probability of a particular split happening?*
- Answer: 1/n

Analyzing Quicksort: Average Case

- So partition generates splits
  (0:n-1, 1:n-2, 2:n-3, … , n-2:1, n-1:0)
  each with probability 1/n
- If T(n) is the expected running time,

$$T(n) = \frac{1}{n}\sum_{k=0}^{n-1}\left[T(k) + T(n-1-k)\right] + \Theta(n)$$

- *What is each term under the summation for?*
- *What is the Θ(n) term for?*

Analyzing Quicksort: Average Case

- So…

$$T(n) = \frac{1}{n}\sum_{k=0}^{n-1}\left[T(k) + T(n-1-k)\right] + \Theta(n)$$

$$= \frac{2}{n}\sum_{k=0}^{n-1} T(k) + \Theta(n)$$

Write it on the board

  - Note: this is just like the book's recurrence (p166), except that the summation starts with k=0
  - We'll take care of that in a second

Analyzing Quicksort: Average Case

- We can solve this recurrence using the dreaded substitution method
  - Guess the answer
  - Assume that the inductive hypothesis holds
  - Substitute it in for some value < n
  - Prove that it follows for n

Analyzing Quicksort: Average Case

- We can solve this recurrence using the dreaded substitution method
  - Guess the answer
    - *What's the answer?*
  - Assume that the inductive hypothesis holds
  - Substitute it in for some value < n
  - Prove that it follows for n

Analyzing Quicksort: Average Case

- We can solve this recurrence using the dreaded substitution method
  - Guess the answer
    - $T(n) = O(n \lg n)$
  - Assume that the inductive hypothesis holds
  - Substitute it in for some value $< n$
  - Prove that it follows for n

Analyzing Quicksort: Average Case

- We can solve this recurrence using the dreaded substitution method
  - Guess the answer
    - $T(n) = O(n \lg n)$
  - Assume that the inductive hypothesis holds
    - *What's the inductive hypothesis?*
  - Substitute it in for some value < n
  - Prove that it follows for n

Analyzing Quicksort: Average Case

- We can solve this recurrence using the dreaded substitution method
  - Guess the answer
    - $T(n) = O(n \lg n)$
  - Assume that the inductive hypothesis holds
    - $T(n) \leq an \lg n + b$ for some constants a and b
  - Substitute it in for some value < n
  - Prove that it follows for n

Analyzing Quicksort: Average Case

- We can solve this recurrence using the dreaded substitution method
  - Guess the answer
    - $T(n) = O(n \lg n)$
  - Assume that the inductive hypothesis holds
    - $T(n) \leq an \lg n + b$ for some constants a and b
  - Substitute it in for some value < n
    - *What value?*
  - Prove that it follows for n

Analyzing Quicksort: Average Case

- We can solve this recurrence using the dreaded substitution method
  - Guess the answer
    - $T(n) = O(n \lg n)$
  - Assume that the inductive hypothesis holds
    - $T(n) \leq an \lg n + b$ for some constants a and b
  - Substitute it in for some value < n
    - The value k in the recurrence
  - Prove that it follows for n

Analyzing Quicksort: Average Case

- We can solve this recurrence using the dreaded substitution method
  - Guess the answer
    - $T(n) = O(n \lg n)$
  - Assume that the inductive hypothesis holds
    - $T(n) \leq an \lg n + b$ for some constants a and b
  - Substitute it in for some value < n
    - The value k in the recurrence
  - Prove that it follows for n
    - Grind through it…

Analyzing Quicksort: Average Case

$$T(n) = \frac{2}{n}\sum_{k=0}^{n-1} T(k) + \Theta(n)$$

The recurrence to be solved

$$\text{¿}\ \frac{2}{n}\sum_{k=0}^{n-1} (ak \lg k + b) + \Theta(n)$$

Plug in inductive hypothesis

$$\text{¿}\ \frac{2}{n}\left[b + \sum_{k=1}^{n-1} (ak \lg k + b)\right] + \Theta(n)$$

Expand out the k=0 case

$$\text{¿}\ \frac{2}{n}\sum_{k=1}^{n-1} (ak \lg k + b) + \frac{2b}{n} + \Theta(n)$$

2b/n is just a constant, so fold it into Θ(n)

$$\text{¿}\ \frac{2}{n}\sum_{k=1}^{n-1} (ak \lg k + b) + \Theta(n)$$

Note: leaving the same recurrence as the book

Analyzing Quicksort: Average Case

$$T(n) = \frac{2}{n}\sum_{k=1}^{n-1}(ak\lg k + b) + \Theta(n)$$

The recurrence to be solved

$$= \frac{2}{n}\sum_{k=1}^{n-1} ak\lg k + \frac{2}{n}\sum_{k=1}^{n-1} b + \Theta(n)$$

Distribute the summation

$$= \frac{2a}{n}\sum_{k=1}^{n-1} k\lg k + \frac{2b}{n}(n-1) + \Theta(n)$$

Evaluate the summation:
b+b+…+b = b (n-1)

$$= \frac{2a}{n}\sum_{k=1}^{n-1} k\lg k + 2b + \Theta(n)$$

Since n-1<n, 2b(n-1)/n < 2b

This summation gets its own set of slides later

Analyzing Quicksort: Average Case

$$T(n) \le \frac{2a}{n}\sum_{k=1}^{n-1} k \lg k + 2b + \Theta(n)$$

The recurrence to be solved

$$\le \frac{2a}{n}\left(\frac{1}{2}n^2 \lg n - \frac{1}{8}n^2\right) + 2b + \Theta(n)$$

We'll prove this later

$$\le an \lg n - \frac{a}{4}n + 2b + \Theta(n)$$

Distribute the (2a/n) term

$$\le an \lg n + b + \left(\Theta(n) + b - \frac{a}{4}n\right)$$

Remember, our goal is to get T(n) ≤ an lg n + b

$$\le an \lg n + b$$

Pick a large enough that an/4 dominates Θ(n)+b

Analyzing Quicksort: Average Case

- So $T(n) \leq an \lg n + b$ for certain a and b
  - Thus the induction holds
  - Thus T(n) = O(n lg n)
  - Thus quicksort runs in O(n lg n) time on average

Binary Search

- BinSearch(A,v):

```
return BinSearchHelper(A,0,n-1,v)
```

- BinSearchHelper(A,low,high,v):

```
if low > high then
    return -1
mid ← (low+high)/2
if A[mid] = v then
  return mid
else if A[mid] < v then
  return BinSearchHelper(A,mid+1,high,v)
else
  return BinSearchHelper(A,low,mid-1,v)
```

Binary search analysis

Best Case:

When element is in the middle position

One comparision required

O(1)

Binary search analysis

Worst Case:

When element is in the not present

Recurrence relation is:

$T(n)=T(n/2)+1$

Time required—O(log n)[prove by substitution or master theorem)]

Binary search analysis

- Average case analysis

make some assumptions:

1) The value we are searching for is in the array.

2) Each value is equally likely to be in the array.

3) The size of the array is $n = 2^k-1$, where k is a positive integer.

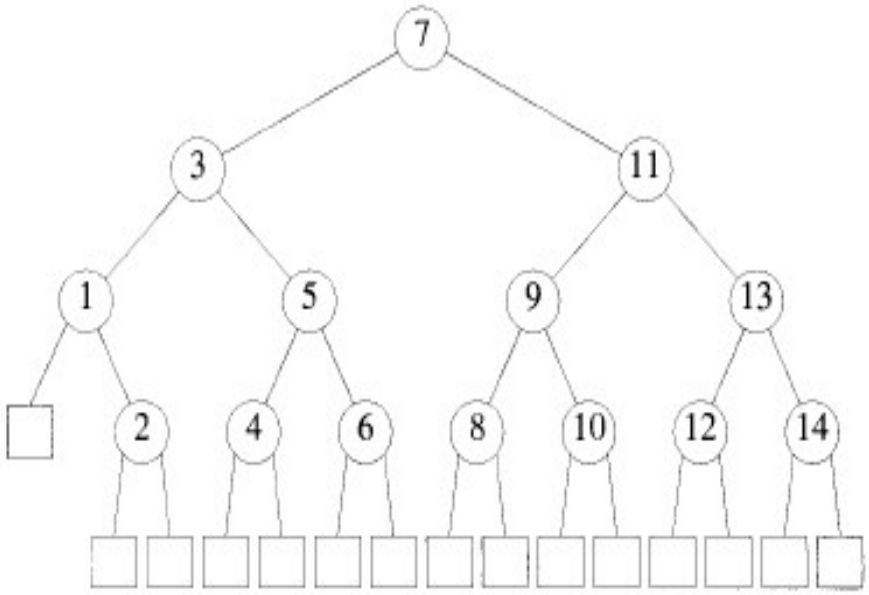
Decision Tree

−15, −6, 0, 7, 9, 23, 54, 82, 101, 112, 125, 131, 142, 151

$x = 151$	*low*	*high*	*mid*
	1	14	7
	8	14	11
	12	14	13
	14	14	14
			found

$x = -14$	*low*	*high*	*mid*
	1	14	7
	1	6	3
	1	2	1
	2	2	2
	2	1	not found

$x = 9$	*low*	*high*	*mid*
	1	14	7
	1	6	3
	4	6	5
			found



Decision Tree

Take the following array of 15 elements as an example:

5 15 25 35 45 55 65 75 85 95 105 115 125 135 145

First, we note that using 1 comparison, we can find 1 element. If we use two comparisons exactly, there are 2 possible elements we can find. In general, after using k comparisons, we can find 2^{k-1} elements. (To see this, consider doing a binary search on the array 2, 5, 6, 8, 12, 17, 19. 8 would be found in 1 comparison, 5 and 17 in two, and 1, 6, 12 and 19 would be found in 3 comparisons.)

The expected number of comparisons we make when running the algorithm would be a sum over the number of comparisons necessary to find each individual element multiplied by the probability we are searching for that element. Let p(j) represent the number of comparisons it would take to find element j, then the sum we have is:

$$\sum_{j=1}^{n} \frac{1}{n} p(j) = \frac{1}{n} \sum_{j=1}^{n} p(j)$$

Now, the trick will be to determine that sum. BUT, we have already out lined that p(j) will be 1 for one value of j, 2 for 2 values of j, 3 for 4 values of j, etc. Since $n=2^k-1$, we can formulate the sum as follows:

$$\sum_{j=1}^{n} \frac{1}{n} p(j) = \frac{1}{n} \sum_{j=1}^{n} p(j) = \frac{1}{n} \sum_{j=1}^{k} j 2^{j-1}$$

This is because the value j appears exactly 2^{j-1} times in the original sum.

$$\sum_{j=1}^{k} j 2^{j-1} = 1(2^0) + 2(2^1) + \ldots\ldots\ldots\ldots\ldots + k(2^{k-1})$$

$$-2\sum_{j=1}^{k} j 2^{j-1} = 1(2^1) + 2(2^2) + \ldots + (k-1)(2^{k-1}) + k(2^k)$$

Subtracting the bottom equation from the top, we get the follo

$$-\sum_{j=1}^{k} j 2^{j-1} = 2^0 + 2^1 + 2^2 + \ldots + 2^{k-1} - k 2^k$$

$$-\sum_{j=1}^{k} j 2^{j-1} = 2^k - 1 - k 2^k$$

$$\sum_{j=1}^{k} j 2^{j-1} = -2^k + 1 + k 2^k$$

$$\sum_{j=1}^{k} j 2^{j-1} = (k-1)2^k + 1$$

Thus, the average run-time of the binary search is

$$\frac{(k-1)2^k+1}{n} = \frac{(k-1)2^k+1}{2^k-1} \approx k-1 = O(\log n)$$

Example: Counting money

- Suppose you want to count out a certain amount of money, using the fewest possible bills and coins
- A greedy algorithm would do this would be: At each step, take the largest possible bill or coin that does not overshoot
  - Example: To make $6.39, you can choose:
    - a $5 bill
    - a $1 bill, to make $6
    - a 25¢ coin, to make $6.25
    - A 10¢ coin, to make $6.35
    - four 1¢ coins, to make $6.39

Some Greedy Algorithms

- Kruskal's MST algorithm
- Prim's MST algorithm
- Dijkstra's SSSP algorithm
- fractional knapsack algorithm
- Huffman codes
- ...

Minimum Spanning Tree

A spanning tree that has minimum total weight is called a **minimum spanning tree** for the graph.

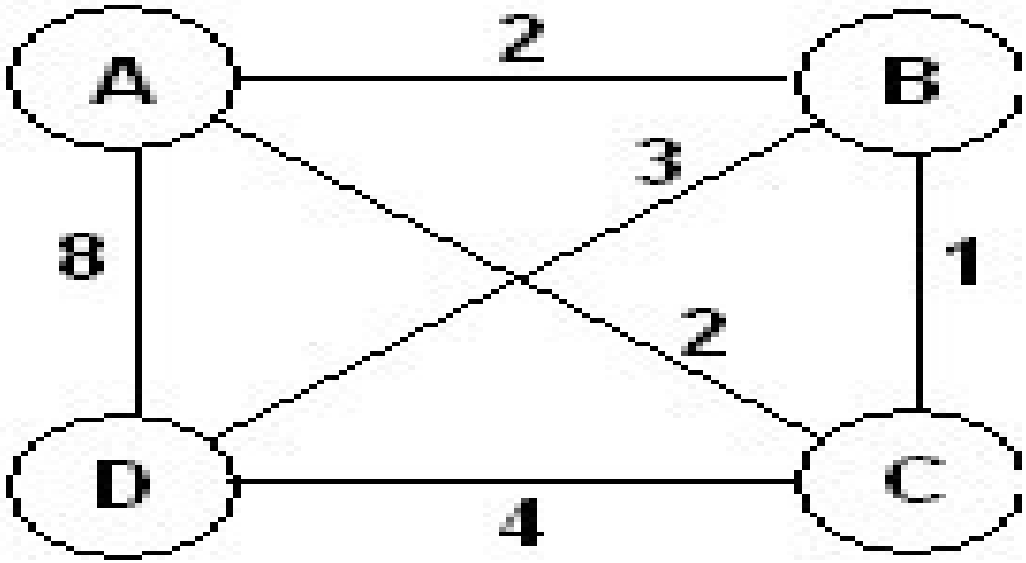
Technically it is a minimum-weight spanning tree.

If all edges have the same weight, breadth-first search or depth-first search will yield minimum spanning trees.

For the rest of this discussion, we assume the edges have weights associated with them.

Minimum Spanning Tree

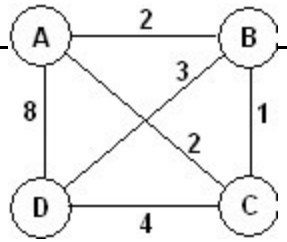
Consider this graph.



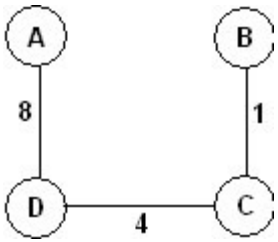
It has 16 spanning trees.

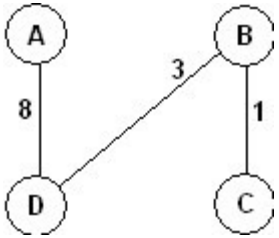
Minimum Spanning Tree

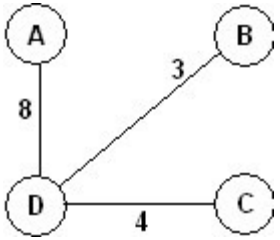
- Consider this graph.

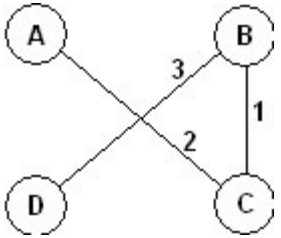


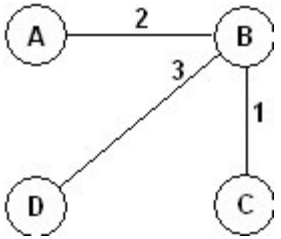
- It has 16 spanning trees. Some are:



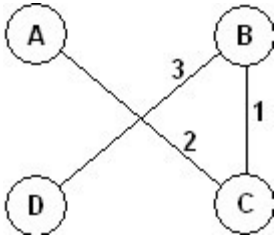


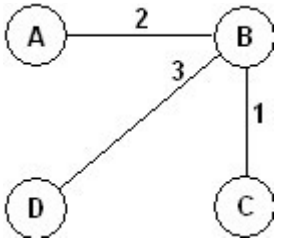






- There are two minimum-cost spanning trees, each with a cost of 6:





Minimum Spanning Tree

There are many approaches to compute a minimum spanning tree. We could try to detect cycles and remove edges, but the two algorithms we will study build them from the bottom-up in a *greedy* fashion.

1. **Prim's Algorithm –** ***starts with a single vertex*** and then adds the edge with the minimum weight to the spanning tree.

2. **Kruskal's Algorithm –** ***starts with a forest of single node trees*** and then adds the edge with the minimum weight to connect two components.

Prim's Algorithm

- Prim's algorithm finds a minimum cost spanning tree by selecting edges from the graph one-by-one as follows:

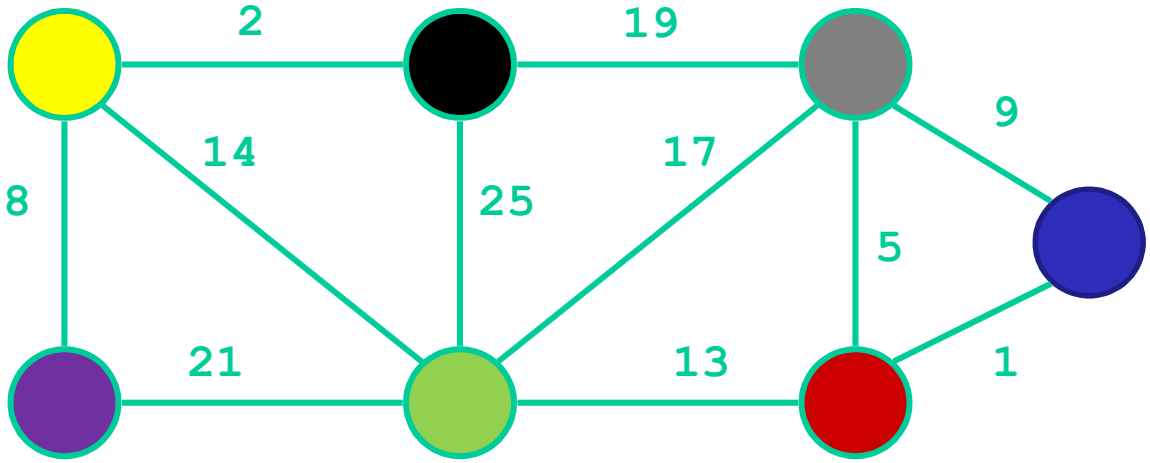
1. It starts with a tree, T, consisting of a single starting vertex, x.

2. Then, it finds the shortest edge emanating from x that connects T to the rest of the graph (i.e., a vertex not in the tree T).

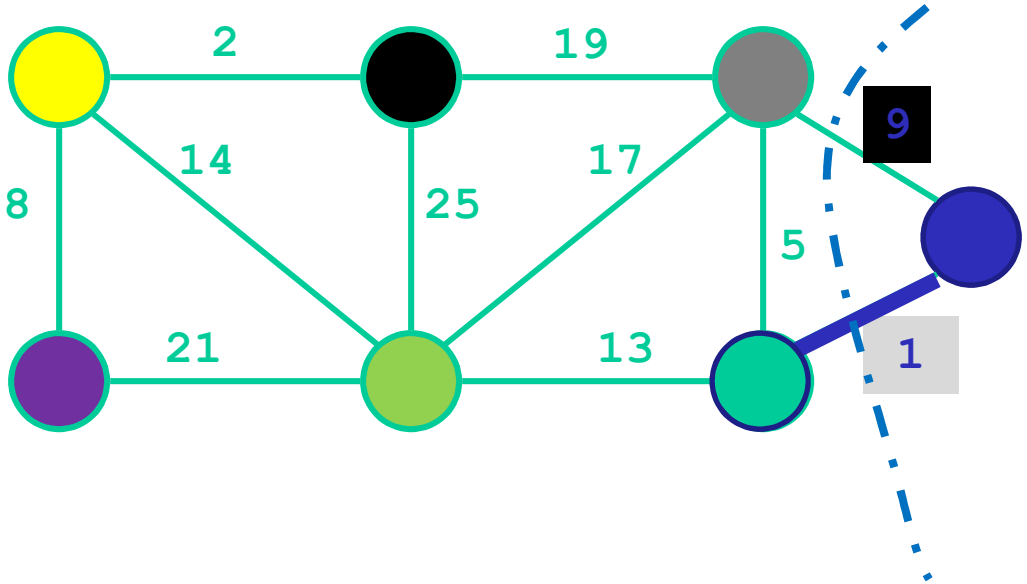
3. It adds this edge and the new vertex to the tree T.

4. It then picks the shortest edge emanating from the revised tree T that also connects T to the rest of the graph and repeats the process.

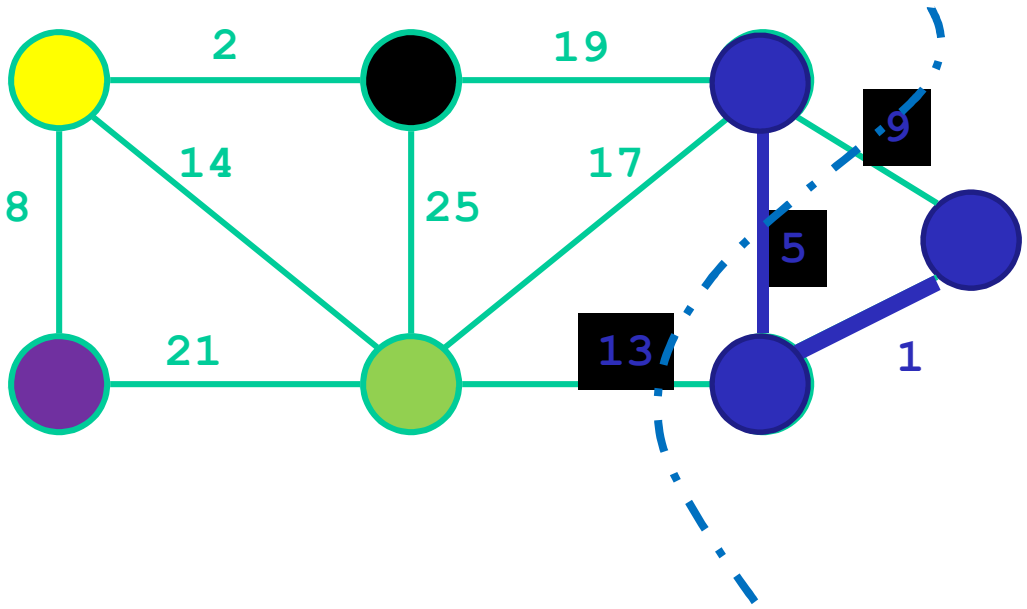
Prim's Algorithm



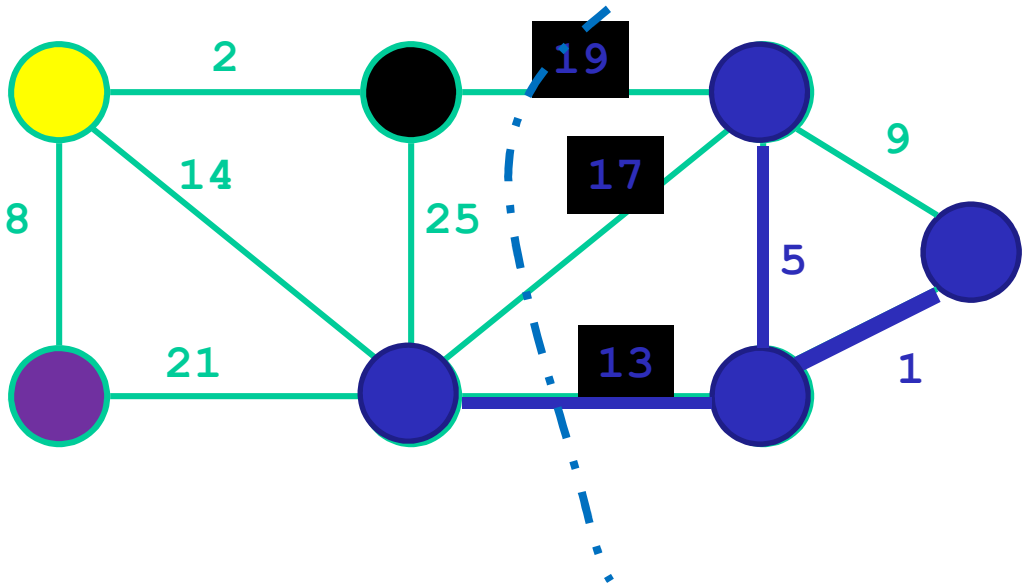
Prim's Algorithm



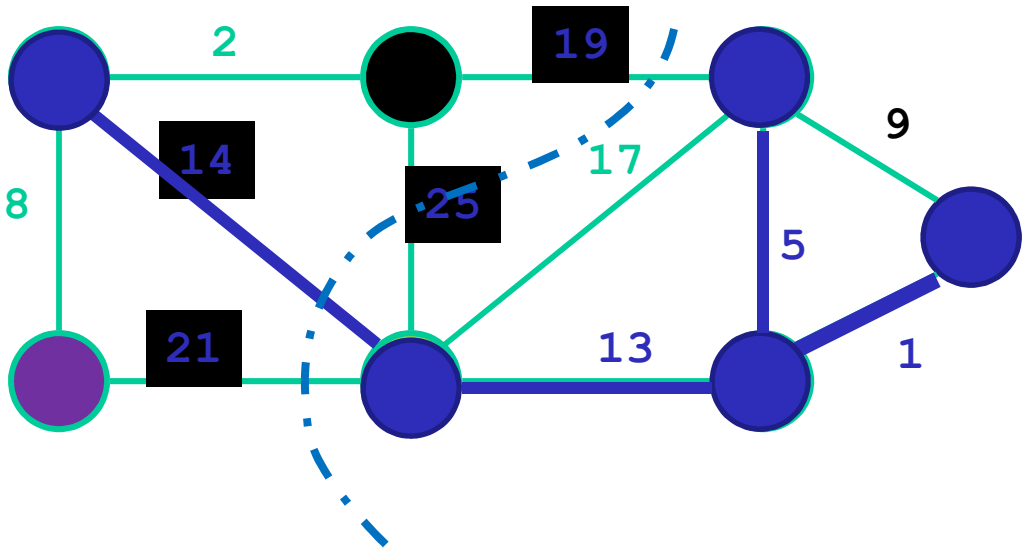
Prim's Algorithm



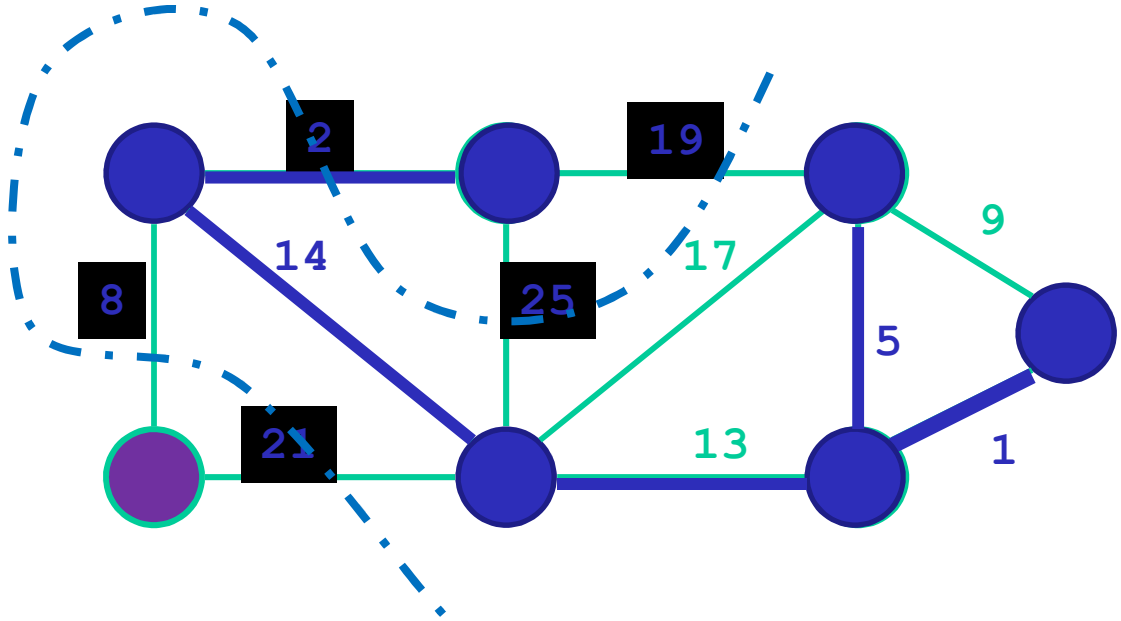
Prim's Algorithm



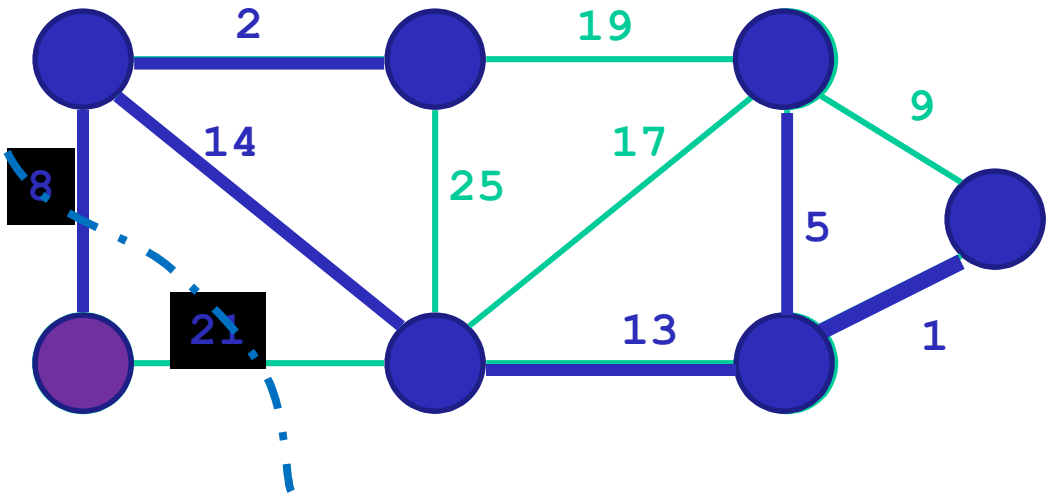
Prim's Algorithm



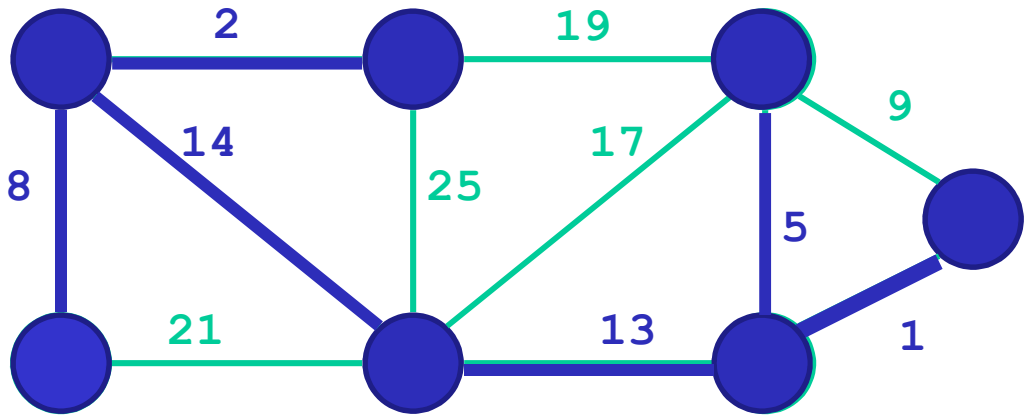
Prim's Algorithm



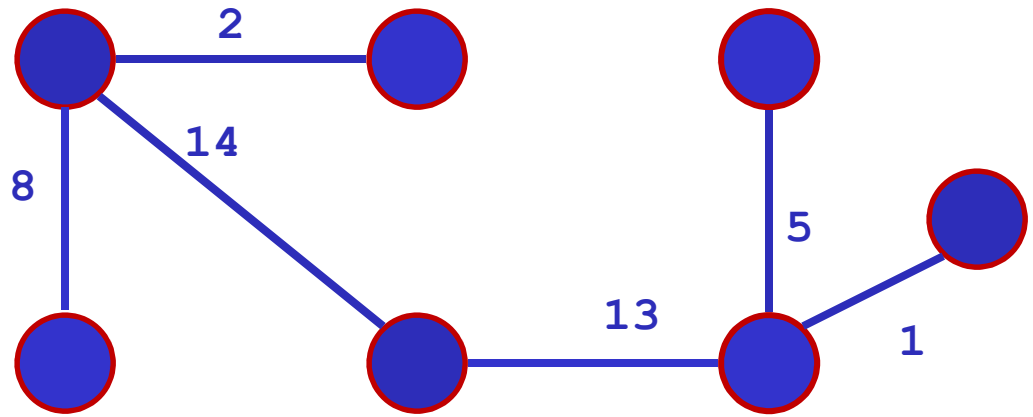
Prim's Algorithm



Prim's Algorithm



Resultant MST



Prim's Algorithm

```
TV={0};
For(T=0;T contains fewer than n-1 edges;add(u,v) to T)
{
  Let (u,v) be the least-cost edge such that u belongs  to TV and v
    doesn't belongs to TV;
  If(there is no such edge)
        break;
  Add v to TV;
}
If(T contains fewer than n-1 edges)
  cout<<"no spanning tree";
```

Prim's Algorithm With adjacency Matrix

Function PRIMS(A , M, N, V): A is a two dimensional array having max. M rows and N columns. VISITED ,D , P are two dimensional vectors.

1. **Repeat for I = 1 to V**

   VISITED[I] = P[I] = 0

   D[I]= 32767

2. **CURRENT =1**

   TOTV=0

   VISITED[CURRENT]=1

3. **Repeat thru step 6 while TOTV != V**

4. **MINCOST = 32767**

   Repeat for I =1 to V

   If (A[CURRENT][I] != 0 AND VISITED[I] = 0)

   If (D[I] >= A[CURRENT][I])

   D[I] = A[CURRENT][I]

   P[I] =CURRENT

Prim's Algorithm contd..

```
5.      Repeat for I =1 to V
        If (VISITED[I] = 0 AND D[I] <= MINCOST)
              MINCOST= D[I]
               CURRENT = I

6.      VISITED[CURRENT] = 1
        TOTV=TOTV + 1

7.      MINCOST = 0
        Repeat for I = 1 to V
            WRITE( I ,   P[I])
            If D[I] != 32767
                 MINCOST = MINCOST + D[I]

8.      RETURN MINCOST
```

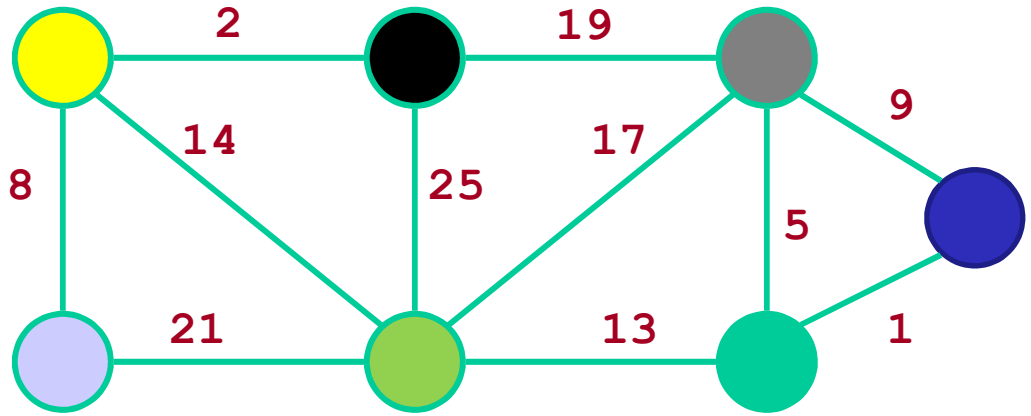
```
1   Algorithm Prim(E, cost, n, t)
2   // E is the set of edges in G. cost[1 : n, 1 : n] is the cost
3   // adjacency matrix of an n vertex graph such that cost[i, j] is
4   // either a positive real number or ∞ if no edge (i, j) exists.
5   // A minimum spanning tree is computed and stored as a set of
6   // edges in the array t[1 : n − 1, 1 : 2]. (t[i, 1], t[i, 2]) is an edge in
7   // the minimum-cost spanning tree. The final cost is returned.
8   {
9       Let (k, l) be an edge of minimum cost in E;
10      mincost := cost[k, l];
11      t[1, 1] := k; t[1, 2] := l;
12      for i := 1 to n do  // Initialize near.
13          if (cost[i, l] < cost[i, k]) then near[i] := l;
14          else near[i] := k;
15      near[k] := near[l] := 0;
16      for i := 2 to n − 1 do
17      { // Find n − 2 additional edges for t.
18          Let j be an index such that near[j] ≠ 0 and
19          cost[j, near[j]] is minimum;
20          t[i, 1] := j; t[i, 2] := near[j];
21          mincost := mincost + cost[j, near[j]];
22          near[j] := 0;
23          for k := 1 to n do // Update near[ ].
24              if ((near[k] ≠ 0) and (cost[k, near[k]] > cost[k, j]))
25                  then near[k] := j;
26      }
27      return mincost;
28  }
```

Kruskal's Algorithm

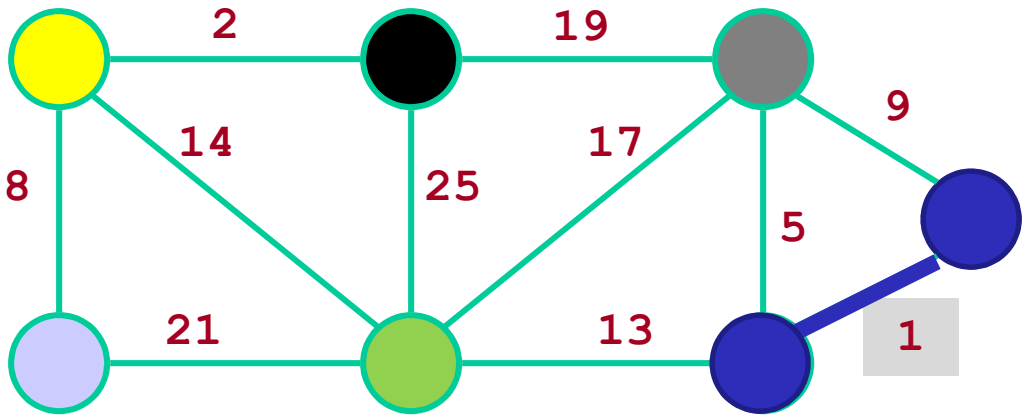
Greedy algorithm to choose the edges as follows.

Step 1	**First edge: choose any edge with the minimum weight.**
Step 2	**Next edge: choose any edge with minimum weight from *those not yet selected*. (The subgraph can look disconnected at this stage.)**
Step 3	**Continue to choose edges of minimum weight from those not yet selected, except *do not select any edge that creates a cycle* in the subgraph.**
Step 4	**Repeat step 3 until the subgraph connects all vertices of the original graph.**

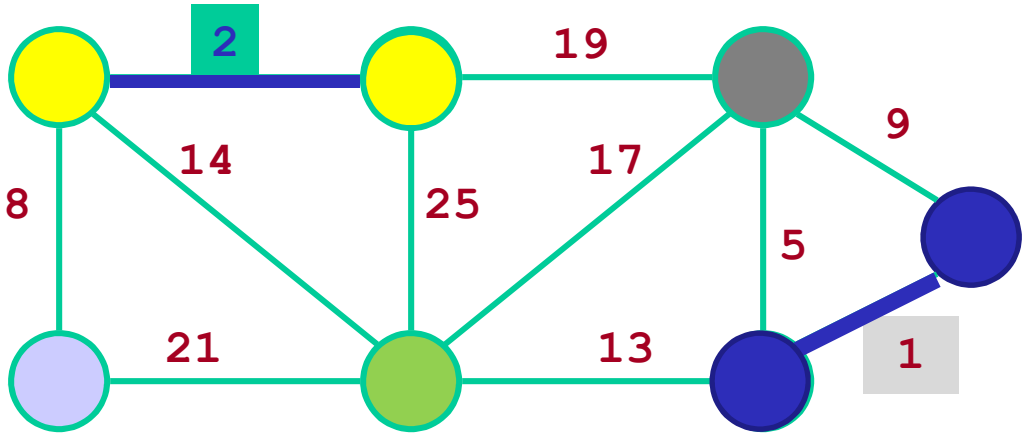
Kruskal's Algorithm



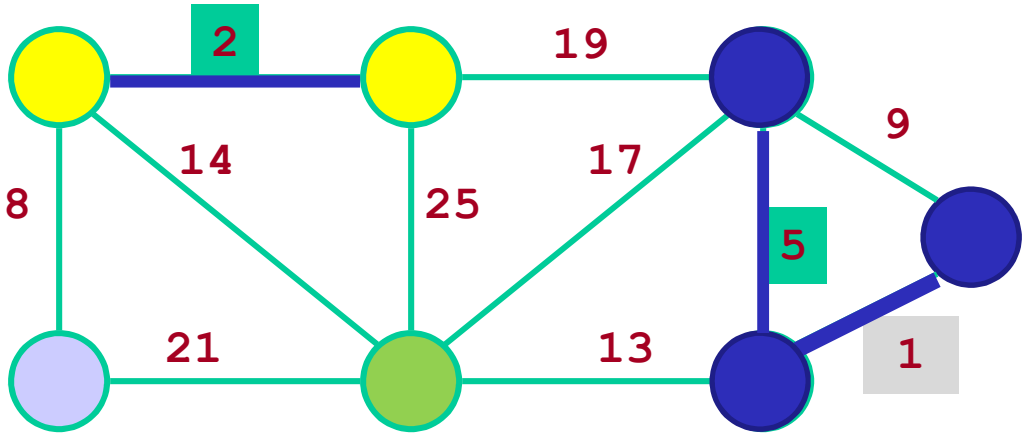
Kruskal's Algorithm



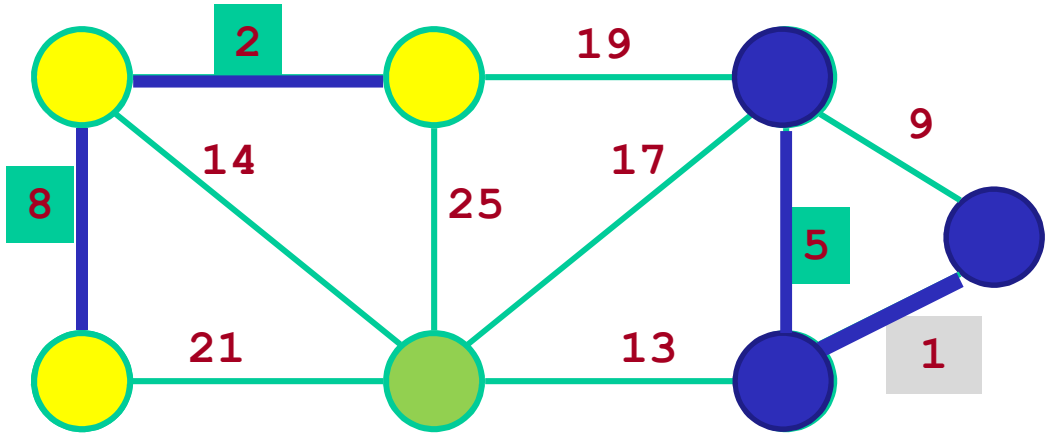
Kruskal's Algorithm



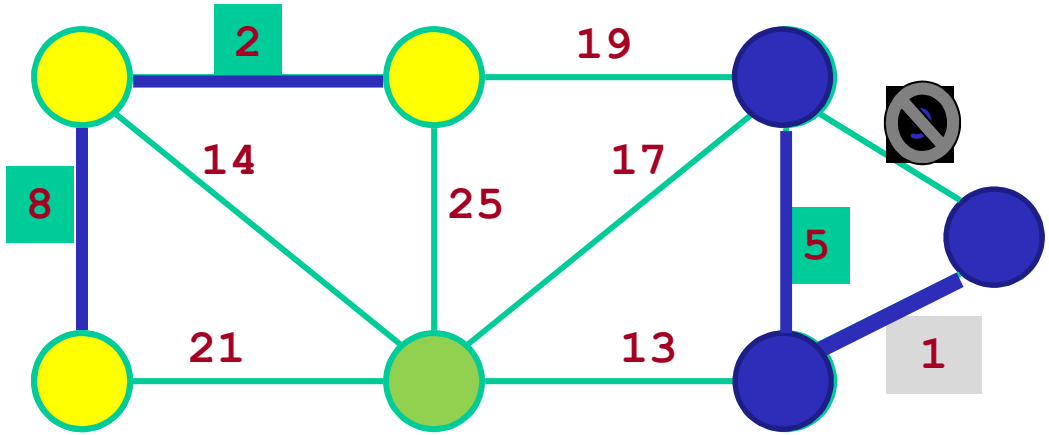
Kruskal's Algorithm



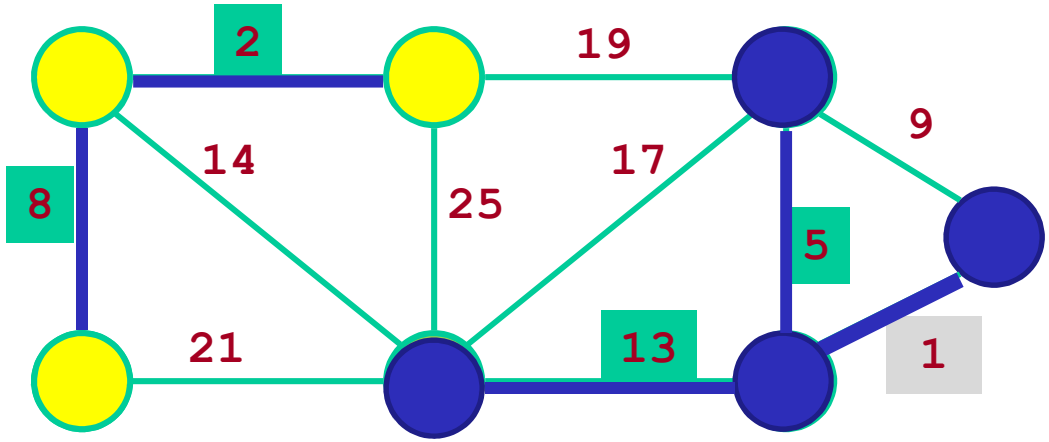
Kruskal's Algorithm



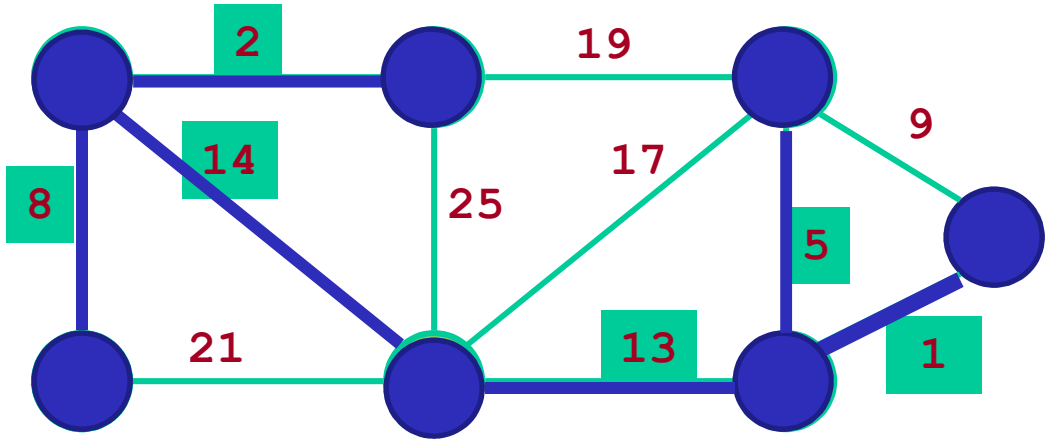
Kruskal's Algorithm



Kruskal's Algorithm

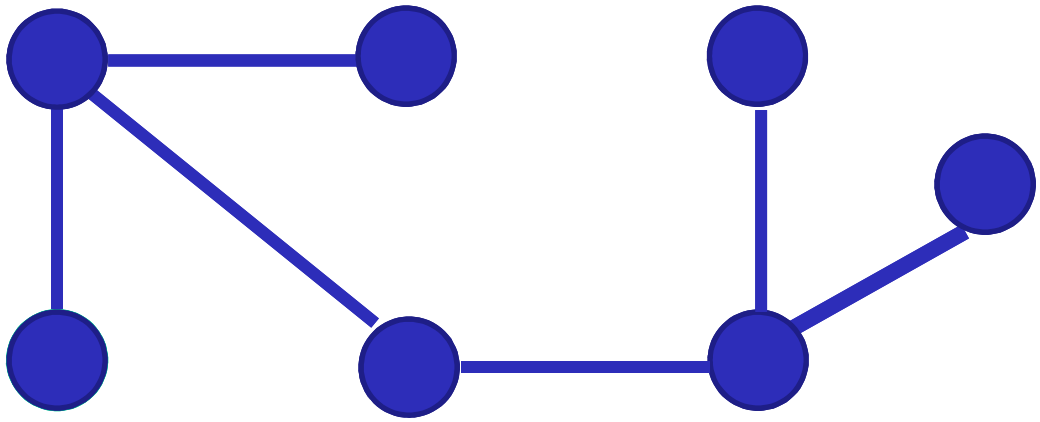


Kruskal's Algorithm



MST using Krushkal's Algorithm

Total Weight --- 43



Krushkal's Algorithm

```
T=0;
While ((T contains less than n-1 edges)&&(E is not empty)
{
  choose an edge (v,w) from E of lowest-cost;
  Delete (v,w) from E;
  If ((v,w) doesn't create a cycle in T)
        Add (v,w) to T;
  Else
          discard(v,w);
}
If(T contains fewer than n-1 edges)
  cout<<"no spanning tree";
```

Kruskal's Algorithm

- Cycle determination
  - If two vertices have the same representative, they're already connected and adding a further connection between them is pointless
  - Procedure:
    - For each end-point of the edge that you're going to add,
    1. follow the lists and find its representative
    2. if the two representatives are equal, then the edge will form a cycle

Kruskal's Algorithm

What will affect the running time?

```
Kruskal()
{
    T = ∅;
    for each v ∈ V
        MakeSet(v);
    sort E by increasing edge weight w
    for each (u,v) ∈ E (in sorted order)
        if FindSet(u) ≠ FindSet(v)
            T = T U {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

Kruskal's Algorithm

```
Kruskal()
{
    T = ∅;
    for each v ∈ V
        MakeSet(v);
    sort E by increasing edge weight w
    for each (u,v) ∈ E (in sorted order)
        if FindSet(u) ≠ FindSet(v)
            T = T U {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

What will affect the running time?

1 Sort

O(V) MakeSet() calls

O(E) FindSet() calls

O(V) Union() calls

(Exactly how many Union()s?)

Kruskal's Algorithm: Running Time

- To summarize:
  - Sort edges: O(E lg E)
  - O(V) MakeSet()'s
  - O(E) FindSet()'s
  - O(V) Union()'s
- Upshot:
  - Best disjoint-set union algorithm makes above 3 operations take $O(E \cdot \alpha(E,V))$, α almost constant
  - Overall thus O(E lg E), almost linear w/o sorting

```
1   Algorithm Kruskal(E, cost, n, t)
2   // E is the set of edges in G. G has n vertices. cost[u, v] is the
3   // cost of edge (u, v). t is the set of edges in the minimum-cost
4   // spanning tree. The final cost is returned.
5   {
6       Construct a heap out of the edge costs using Heapify;
7       for i := 1 to n do parent[i] := -1;
8       // Each vertex is in a different set.
9       i := 0; mincost := 0.0;
10      while ((i < n - 1)  and (heap not empty)) do
11      {
12          Delete a minimum cost edge (u, v) from the heap
13          and reheapify using Adjust;
14          j := Find(u); k := Find(v);
15          if (j ≠ k) then
16          {
17              i := i + 1;
18              t[i, 1] := u; t[i, 2] := v;
19              mincost := mincost + cost[u, v];
20              Union(j, k);
21          }
22      }
23      if (i ≠ n - 1) then write ("No spanning tree");
24      else return mincost;
25  }
```