Data Structures and Algorithms

Dr. L. Rajya Lakshmi

Comparison based sorting: Lower bound

- Insertion sort, merge sort, and quick sort
- Primitive operation used is: Comparison
- Consider a sequence S = (a_1, a_1, . . ., a_n) of distinct elements
- Compare two elements, say a_i and a_j (outcome: “yes” or “no”)
  - Perform few steps
  - Compare another pair of elements

Comparison based sorting: Lower bound

- Insertion sort:
  - Compares the first element (say y) from the unsorted part to the last element (say x) in the sorted part
  - Change the position of the last element in the sorted if y < x
  - Compares y to the previous element of x
- Merge sort:
  - Compares the first two elements in the sorted sub-sequences
  - Appends the smallest to the end of the result sequence
- Quick sort:
  - Picks the pivot and place “i“ and “j” at appropriate positions
  - Increments i, and compares the element at ith position to the pivot
  - Decrements j, and compares the element at jth position to the pivot
  - Performs required actions

Comparison based sorting: Lower bound

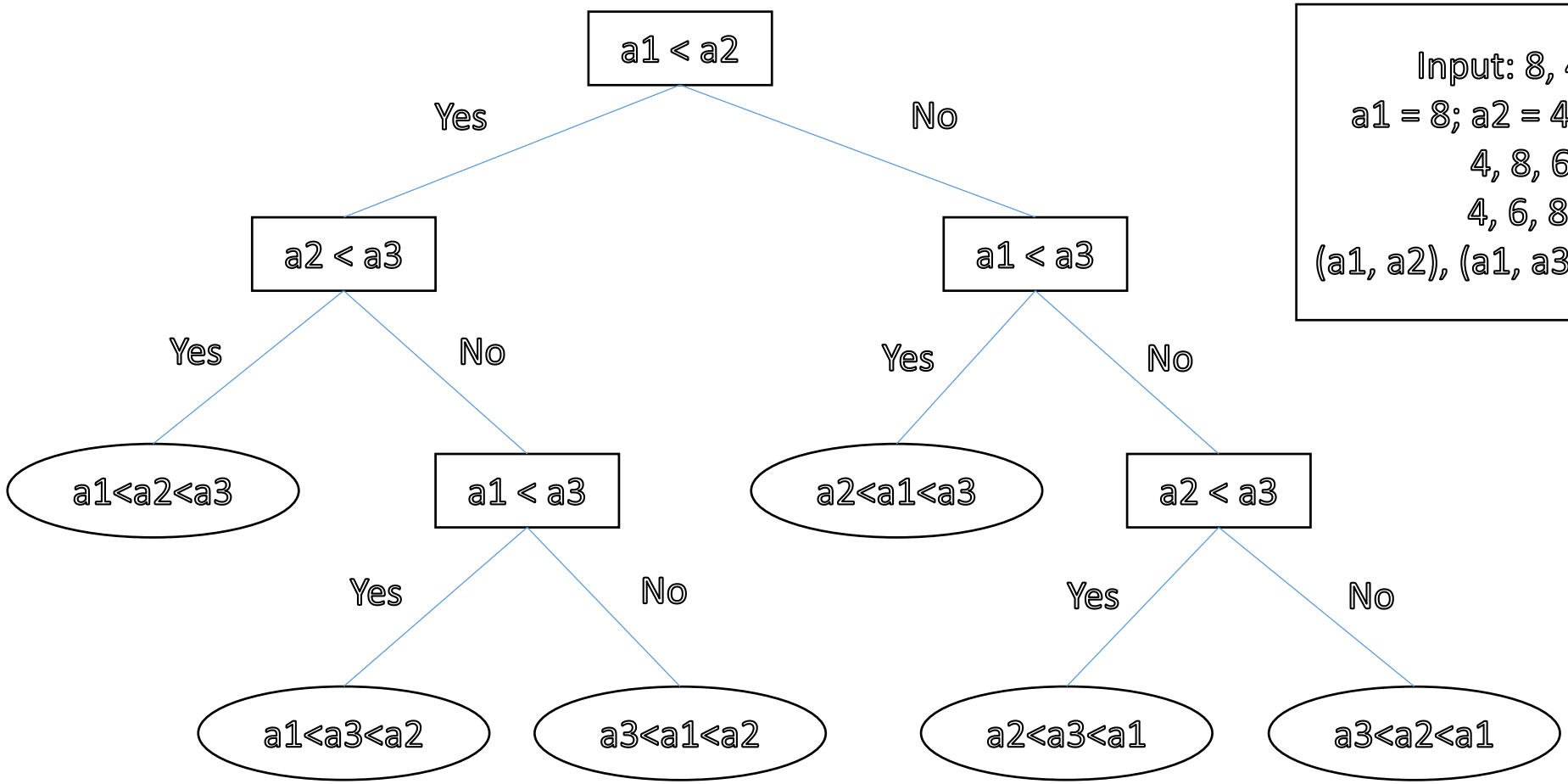
- Consider an input array (8, 4, 6)
- (4, 8, 6)
- (4, 6, 8)
- The comparisons performed are: (4, 8), (6, 8), (4, 6)
- Consider another permutation of input array, say (8, 6, 4)
- (6, 8, 4)
- (4, 6, 8)
- The comparisons performed are: (6, 8), (4, 8), (4, 6)

Comparison based sorting: Lower bound

- A comparison based sorting algorithm can be represented in the form of a decision tree
- All possible sequences of comparisons performed on an n-element sequence (a_1, a_2, . . ., a_n)
- Each internal node represents a comparison
- Associate with each external node v the permutation that causes the sorting algorithm to end up in v
- Each possible input permutation causes the algorithm to execute a series of comparisons (traverse a path) and end up at some external node

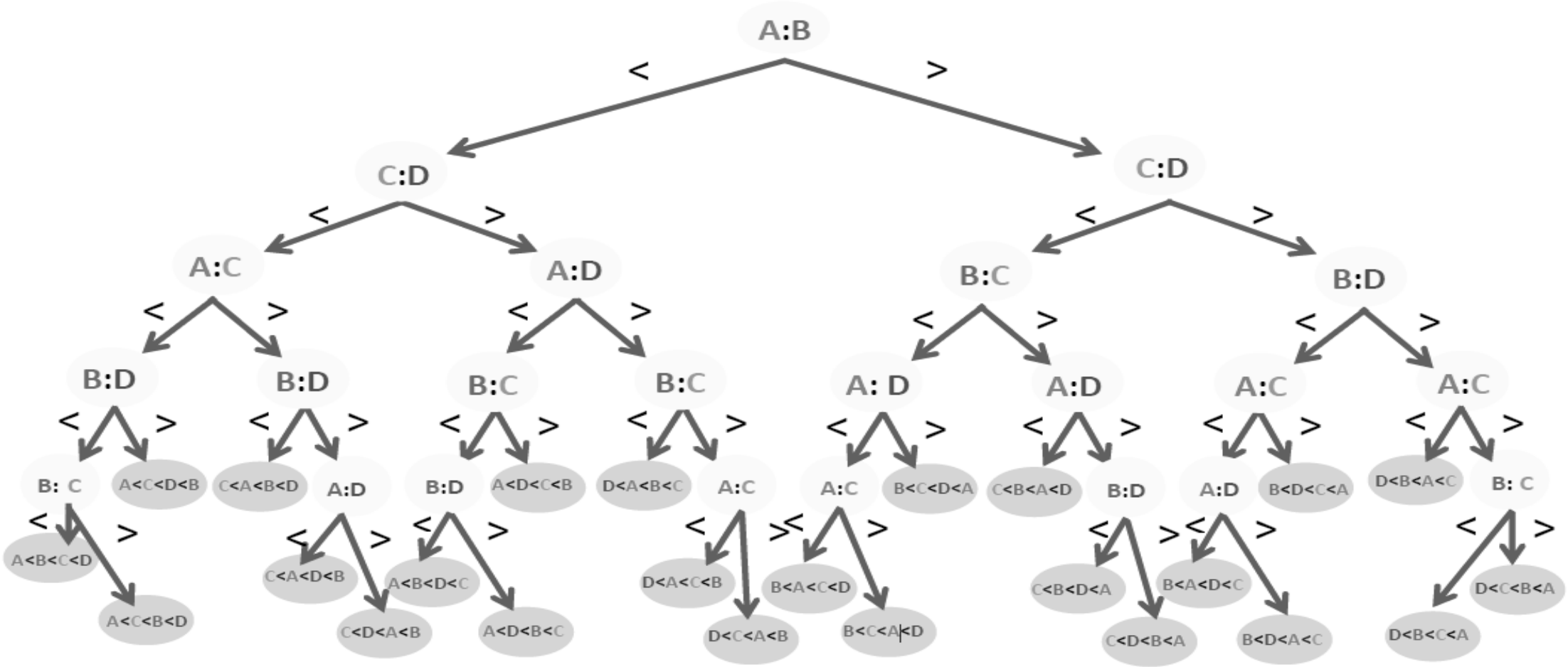
Comparison based sorting: Lower bound

- Consider a general sequence of three elements (a1, a2, a3)



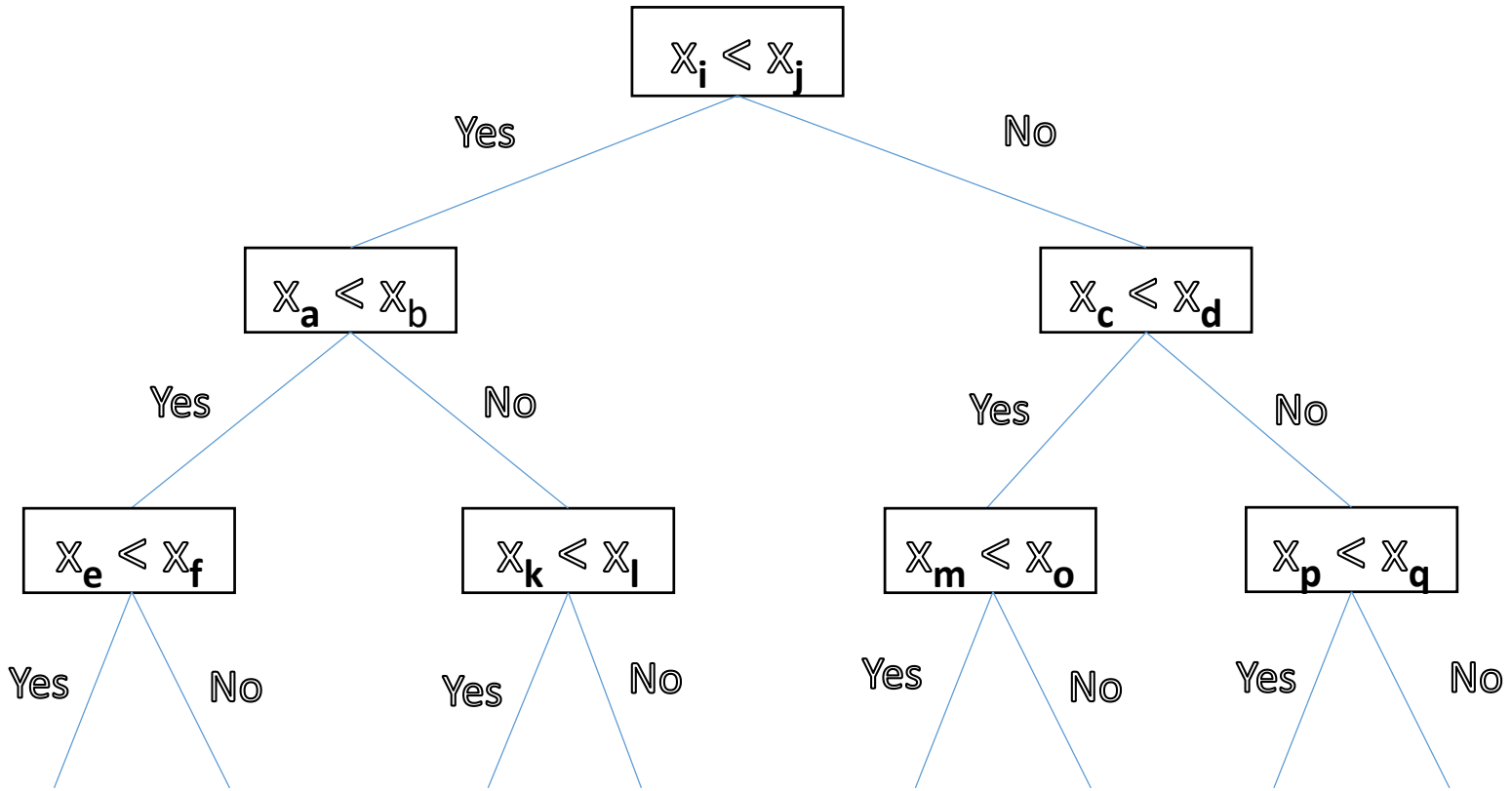
Comparison based sorting: Lower bound

- Consider a general sequence of four elements (A, B, C, D):



Comparison based sorting: Lower bound

- Consider a general sequence (S) of n elements ($x_1, x_2, \ldots, x_n$)



Comparison based sorting: Lower bound

- There exists a 1-1 correspondence between input permutations and external nodes
- Consider two permutations, P1 and P2, of $S = (x_1, x_2, \ldots, x_n)$
- Assume that P1 and P2 end up at the eternal node v
- x_i appears before x_j in P1 and x_i appears after x_j in P2
- The permutation associated with v is P3 of S where x_i appears either before or after x_j

Comparison based sorting: Lower bound

- **Theorem:**

  The running time of any comparison-based algorithm for sorting some n-element input sequence ($S = (x_1, x_2, \ldots, x_n)$) is $\Omega(n \log n)$ in the worst-case

- **Proof:**

  The running time must be greater than or equal to the height of its decision tree

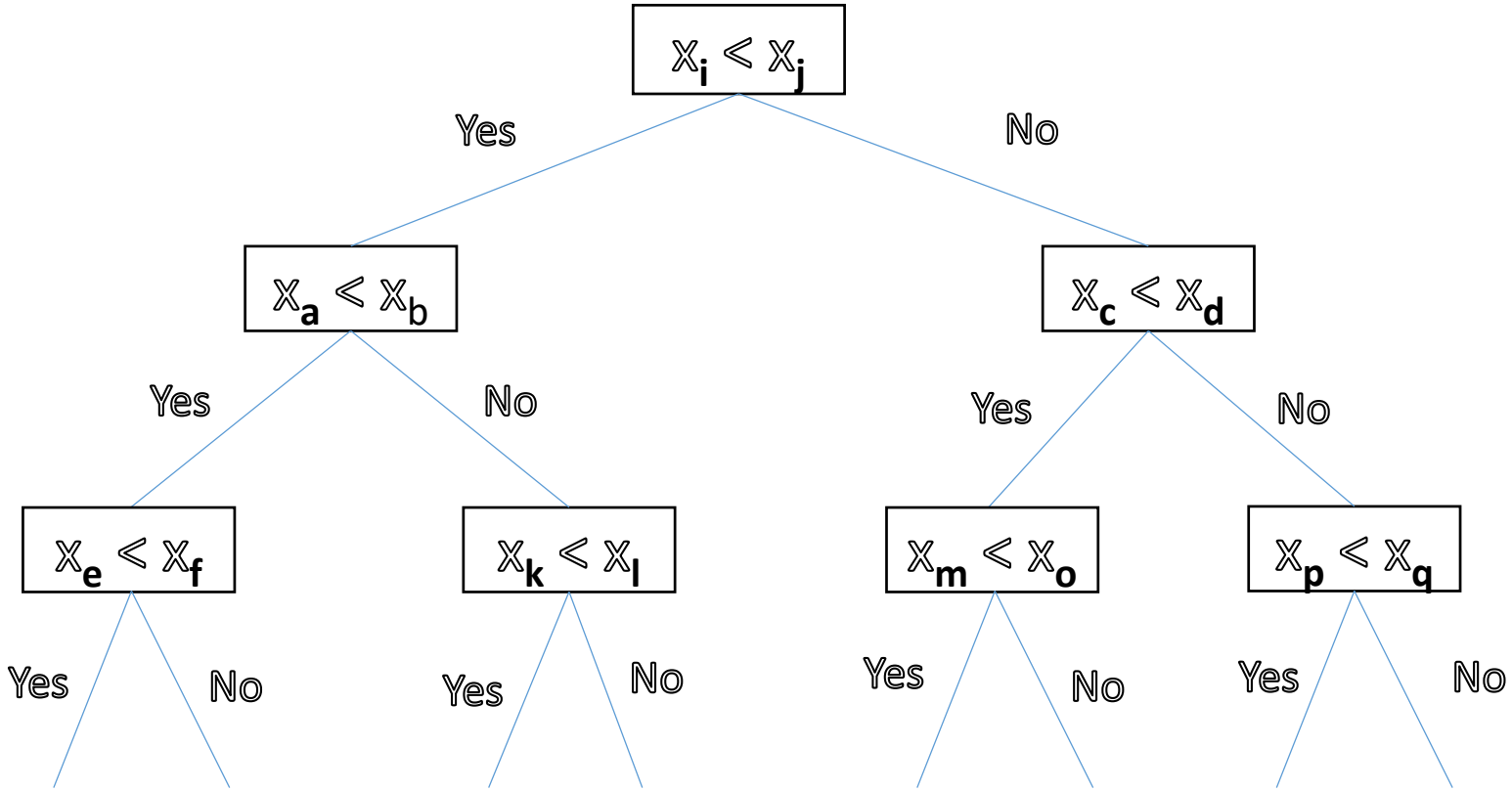
  Each external node must be associated with one permutation of the input

  Each permutation must result in a different external node

  The number of permutations of S are: n!

  There exists “n!” external nodes in the decision tree

Comparison based sorting: Lower bound



n!

Comparison based sorting: Lower bound

- **Proof (Contd):**

Can we say any thing about the height of a binary tree in term of number of leaves?

$h \geq \lceil \log(m) \rceil$ (m is the number of leaves)

Height of decision tree is: log(n!)

$\log(n!) \geq \log(n/2)^{n/2}$

$= n/2 \log(n/2)$

is $\Omega(n \log n)$

Bucket Sort

- Can run asymptotically faster than O(n log n), but with special assumptions
- Consider a sequence S of n items
- Item: (key, element)
- The keys are in the range [0, . . ., N-1] for some $N \geq 2$
- S should be sorted according to the keys
- If N is O(n), then we can sort S in O(n) time
- Restrictive assumption about the format of elements

Bucket sort

- Not based on comparisons
- Uses an array (of size N) of buckets to categorize the items based on keys
- Keys are used as indices (0, . . ., N-1) into the bucket array B
- An item with key “k” is placed in bucket B[k] (sequence of items with key k)
- Place the items back into S in sorted order, how?
- Enumerate the items in the buckets B[0], B[1], . . ., B[N-1] in order

Bucket sort

Algorithm Bucket_Sort(S)

Input: Sequence S of items with integer keys in the range 0, . . ., N-1

Output: Sequence S sorted in nondecreasing order of the keys

```
let B be an array of N sequences, each of which is initially empty
for each item x in S do
        let k be the key of x
        remove x from S and insert x at the end of B[k]
for i ← 0 to N-1 do
        for each item x in bucket B[i] do
                remove x from B[i] and insert it at the end of S
```

Bucket sort

- Time complexity?
- Space complexity?
- What happens as N increases compared to n?

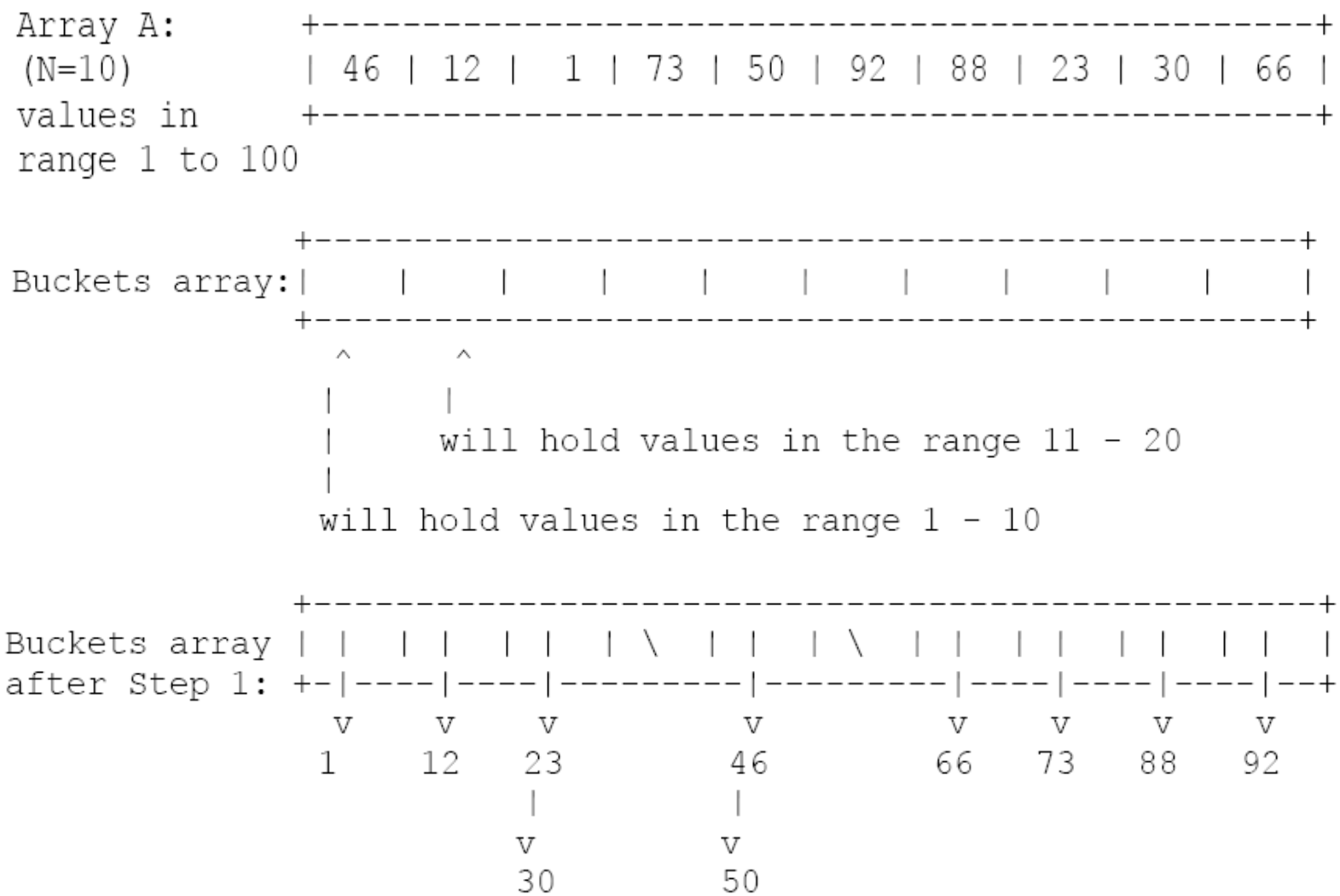
Bucket sort

- How to sort integers using bucket sort?
- The values to be sorted are evenly distributed in some range **min** to **max**
- It is possible to divide the range into N equal parts, each of size k
- Given a value, it is possible to tell in which part of the range it is in

Bucket sort

- Use an array (of size N) of buckets
- The range of values is divided into N equal parts (of size k)
- The first bucket (the first array element) will hold values in the first part of the range (min to min + k -1)
- The second bucket will hold the values in the second part of the range
- Step1:
  - Go through given array of elements once
  - Put each values in its appropriate bucket (maintain each bucket in sorted order)
- Go through buckets B[0] to B[N-1] and put the values back into the original array

Bucket sort


Array A:
(N=10)
values in
range 1 to 100
46 | 12 | 1 | 73 | 50 | 92 | 88 | 23 | 30 | 66
Buckets array:
will hold values in the range 11 - 20
will hold values in the range 1 - 10
Buckets array
after Step 1:
1 12 23 30 46 50 66 73 88 92

Bucket sort

- What is the complexity if the buckets are maintained using linked list?
- What is the time complexity in the best-case scenario?