Fundamental of Computer Science
CS1FC16: Lecture 04

Sorting

Dr Varun Ojha
Department of Computer Science

University of Reading

Learning Objectives

On completion of this lecture, you will be able to

- Understand sorting algorithms and the importance of a faster sorting
- Evaluate time order of sorting algorithms
- Apply knowledge to solve numeric examples
- Create programs to sort given data structure.

Content of this lecture

- Part – I: Simple techniques
  - Insertion sort
  - Bubble sort
- Part – II: Divide and conquer techniques
  - Merge Sort
  - Quick Sort
- Part – III: Non-comparison techniques
  - Radix / Bucket sort
- Part – IV:
  - Exercises

Sorting

`sort(datastructure) -> (datastructure)`

- A sorting algorithm takes a `datastructure` and sorts its elements into *ascending* or else *descending* order
- In general, the elements are records which are sorted in terms of one or more of their fields – the key(s)
- Repetitions of an element are usually allowed so that the sorted list may be in partial order not total order
- A stable sort keeps repeated elements in the order the repetitions were given. An unstable sort may move them

Fundamental of Computer Science
CS1FC16: Lecture 04, Part – I

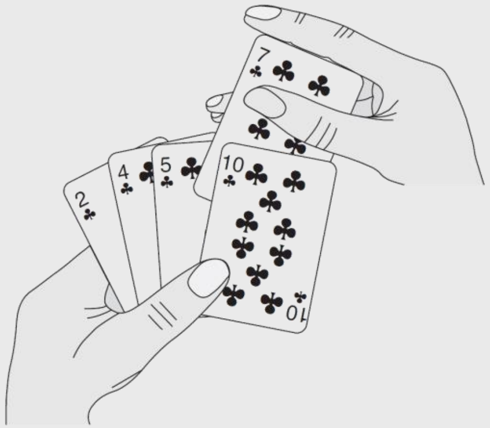
Simple Algorithms $O(n^2)$

Dr Varun Ojha

Department of Computer Science

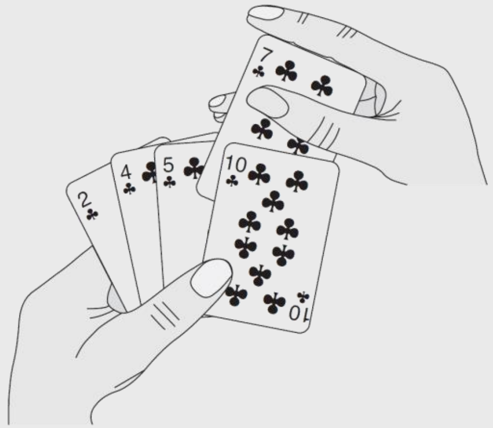
University of Reading

Insertion Sort



- Insert each element into its final position in the sorted list
- Start with an empty output list
- Put the first element into the list
- Put the second element before or after the element that is already in the list
- In general, given a list of n elements, produce a list of $n + 1$ sorted elements by placing the $n + 1\text{th}$ element in its final position in the sorted list

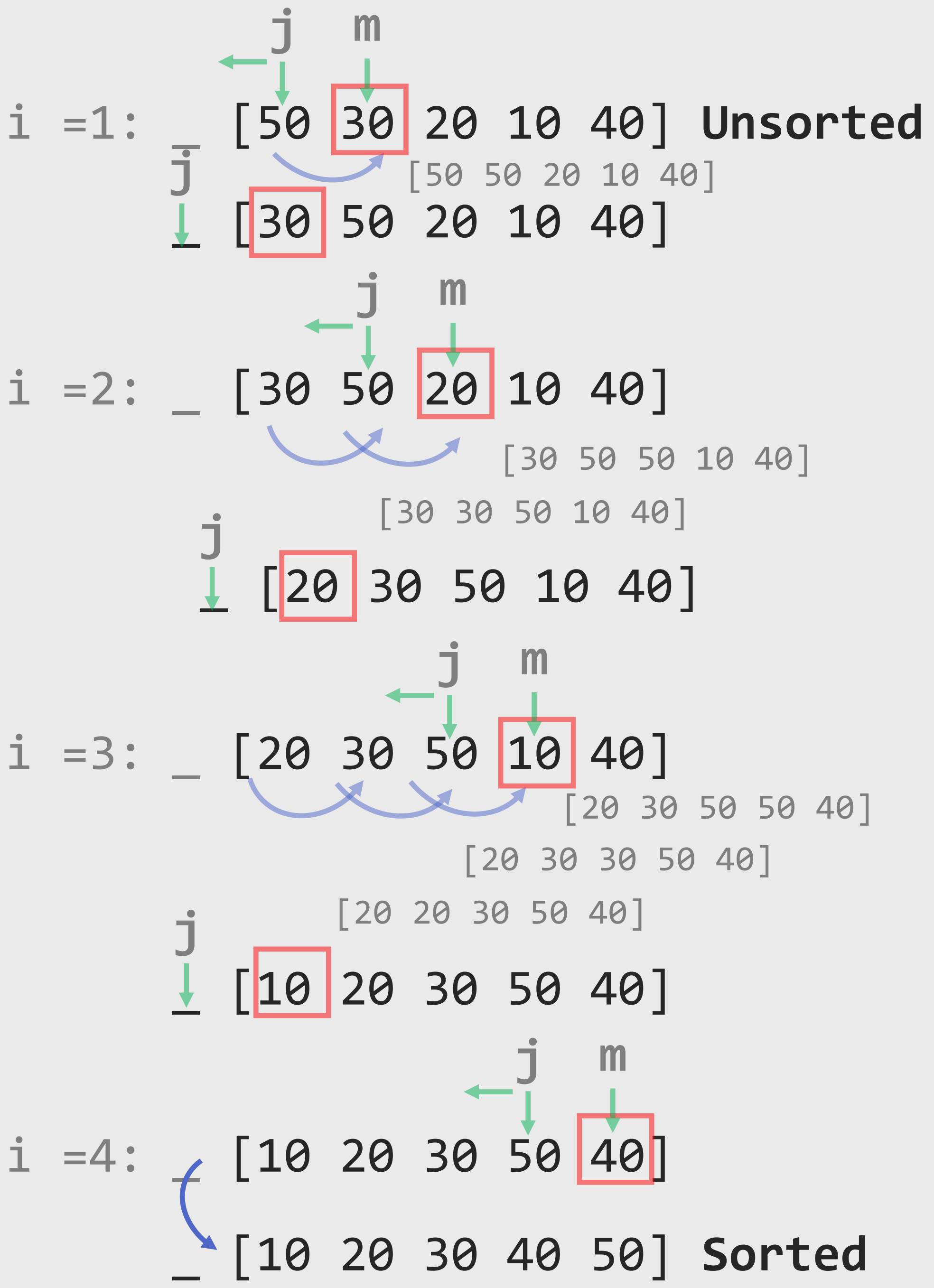
Insertion Sort



```
input: data /* unsorted array */
output: data /* sorted array */
insertionSort(data)
for i from 2 to length(data) do
        m := data(i) /* pick an element for insertion */
        j := i - 1
        while j >= 1 and data(j) > m do
                data(j + 1) := data(j) /* move element to next position */
                i = i – 1 /* take a key */
        data(j+1) = m /* insert at j+1th position */
```

Insertion Sort

```
input: data /* unsorted */
output: data /* sorted */
insertionSort(data)
for i from 2 to length(data) do
        m := data(i) /* pick */
        j := i - 1
        while j >= 1 and data(j) > m do
                data(j + 1) := data(j)
                j := j - 1
        data(j+1) := m /* insert */
```



Insertion Sort Complexity

```
input: data /* unsorted */
output: data /* sorted */
insertionSort(data)
for i from 2 to length(data) do
      m := data(i) /* pick */
      j := i - 1
      while j >= 1 and data(j) > m do
            data(j + 1) := data(j)
            j := j - 1
      data(j+1) := m /* insert */
```

- Compute the worst-case time, $T(n)$, to sort n elements
- The for-loop is executed at most $\max(0, n-1)$ times. In general, this is $n-1$ times
- The while-loop is executed at most i times on the ith iteration.
- The for-loop, together with the while-loop, is not executed more than i times on the ith iteration.
- Therefore, we have

$$T(n) = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} \Rightarrow O(n^2)$$

Bubble Sort

```
input: data /* unsorted array */

output: data /* sorted array */

bubbleSort(data)

for i from 1 to length(data)-1 do
    /* by virtue of swapping, the last element is already sorted*/
    for j from 1 to length(data)-1 do
        if data(j) > data(j+1) do /*if next element is small swap */
            temp := data(j) /* preserve data(j) temporarily */
            data(j) := data(j+1)/* place element data(j+1) to data(j) */
            data(j+1) := temp /* place preserve element to data(j+1) */
```

Bubble Sort

```
input: data /* unsorted array */

output: data /* sorted array */

bubbleSort(data)

for i from 1 to length(data)-1 do
    for j from 1 to length(data)-1 do
        if data(j) > data(j+1) do
            temp := data(j)
            data(j) := data(j+1)
            data(j+1) := temp
```

First Pass: i = 1; run for j from 1 to 4

j = 1 [5 3 1 2 4] Here, bubble sort compares the first two elements, and swaps 5 and 3 since 5 > 3.
j = 2 [3 5 1 2 4] Swap since 5 > 1
j = 3 [3 1 5 2 4] Swap since 5 > 2
j = 4 [3 1 2 5 4] Swap since 5 > 4.

Second Pass: i = 2; run for j from 1 to 4

j = 1 [3 1 2 4 5] Swap since 3 > 1
j = 2 [1 2 3 4 5] Swap since 3 > 2
j = 3 [1 2 3 4 5] Do not swap
j = 3 [1 2 3 4 5] Do not swap

Algorithm will still run since it does not know if elements have been sorted. The bubble sort needs one whole pass without any swap to know it is sorted.

Third Pass: i = 3 run for j from 1 to 4

j = 1 [1 2 3 4 5]
j = 2 [1 2 3 4 5]
j = 3 [1 2 3 4 5]
j = 4 [1 2 3 4 5]

Exercise: How can you make this algorithm little more efficient?

Fundamental of Computer Science
CS1FC16: Lecture 04, Part – II

Divide and conquer techniques $O(n \log n)$

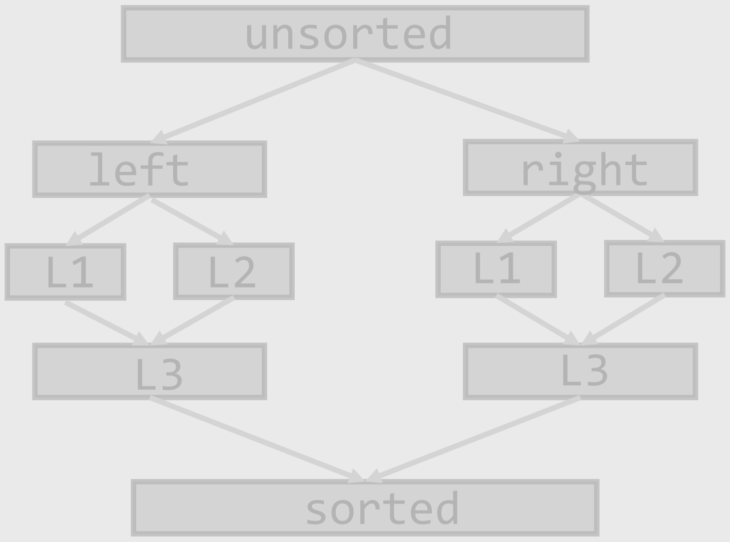
Dr Varun Ojha
Department of Computer Science

University of Reading

Merge Sort

- Sort a list of elements
- If the list has zero or one element, then stop
- If the list has more than one element, then divide the list into two equal or nearly equal parts until all lists have at most one element
  - Recursively sorts the sub lists
  - Recursively merge the results
- Why is this much faster than an insertion sort?

Merge Sort

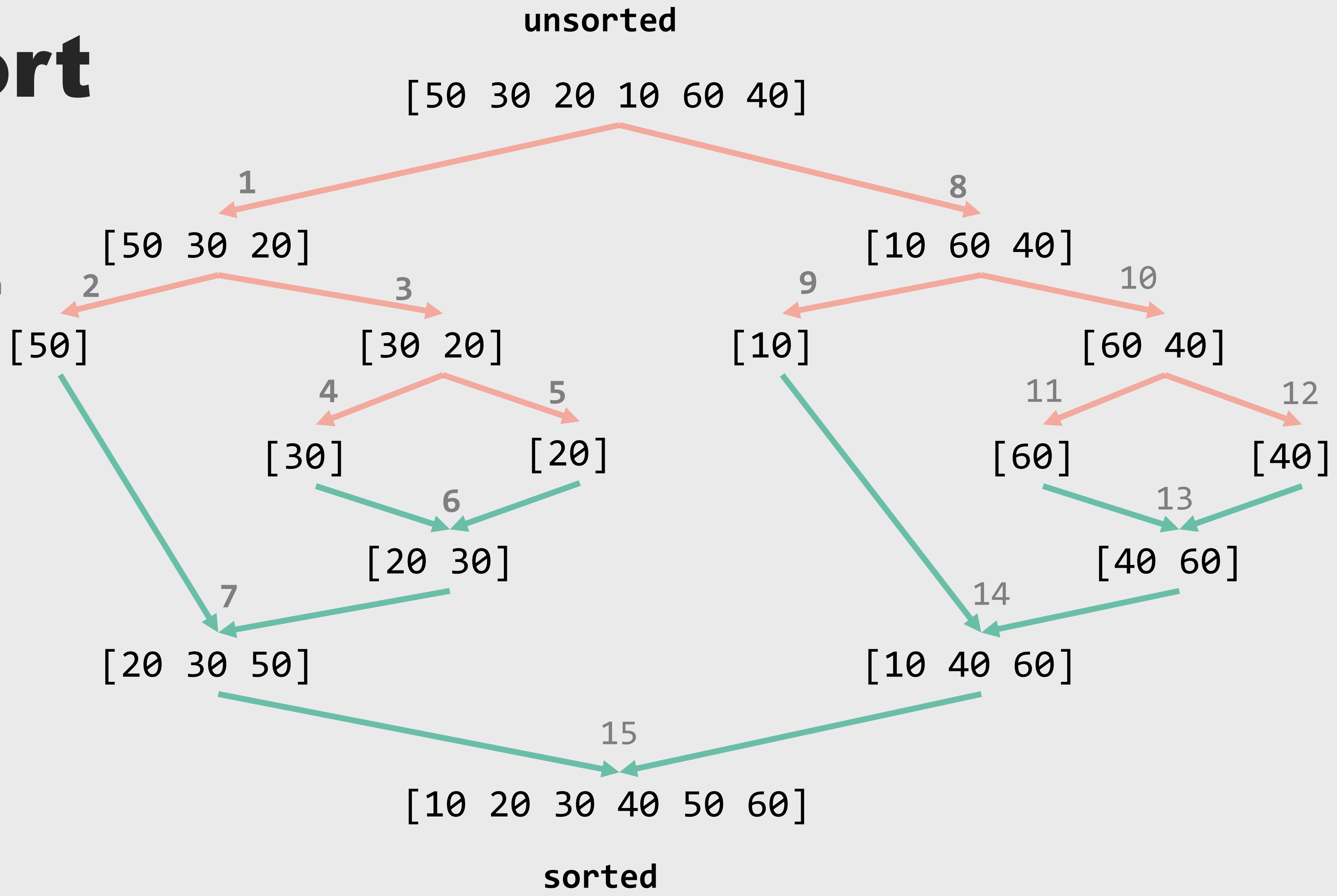



```
input: list /* unsorted array */
output: list /* sorted array */
mergesort(list)-> (list)
    if length(list) > 1 then
        split(list) -> (left, right)
        merge(mergesort(left), mergesort(right))
end
merge(L1, L2) -> (L3)
    L3 := EmptyList
    while L1, L2 are both non-empty
        remove the smaller of the first element of L1, L2 from the list it
        is in and add it on the right of L3

        if removal of this element makes one list empty then remove all of
        the elements from the other list and append them to L3
end
split(L1) -> (L2, L3)
    transcribe the first floor(length(L1)/2) elements of L1 into L2
    transcribe the remaining elements of L1 into L3
end
```

Marge Sort

```
input: list /* unsorted */
output: list /* sorted */
mergesort(list)-> (list)
    if length(list) > 1 then
        split(list)->(
        left, right)
    merge(
        mergesort(left),
        mergesort(right)
    )
end
```



Merge Sort: Complexity

```
input: list /* unsorted array */
output: list /* sorted array */
mergesort(list)-> (list)
    if length(list) > 1 then
        split(list) -> (left,
        right)
        merge(
              mergesort(left),
              mergesort(right)
        )
end
```

$T(n)$ ← mergesort(list)-> (list)

$T(1)$ ← split(list) -> (left, right)

$T(n)$ ← merge(

$T(n/2)$ ← mergesort(left),

and

$T(n/2)$ ← mergesort(right)

- **Divide** (split) takes constant time $T(1)$
- **Concur** (mergesort) operates on two sub lists of length $n/2$, hence it takes $2T(n/2)$.
- **Combine** (merge) operation need to compare n elements $T(n)$.

Merge Sort Complexity

```
input: list /* unsorted array */
output: list /* sorted array */
mergesort(list)-> (list)
    if length(list) > 1 then
        split(list) -> (left, right)
        merge(  mergesort(left),
                mergesort(right)
             )
end
```

- Compute the worst-case time-order, $\boldsymbol{T(n)}$, of merge sort $\boldsymbol{n}$ elements
- To make the analysis easy, assume $\boldsymbol{n = 2^k}$
- Now $\boldsymbol{k = \log_2 n}$
- We have recurrence relation (expression) for merge sort as:

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

T(n) = 2T(n/2) + n -> O(nlog n)?

Revisit Lecture 02: substitution method

We have:

$$T(n) = \begin{cases} 1 & n = 1 \\ 2T(n/2) + n & n > 1 \end{cases}$$

We want to solve:

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

Substitute $T(n/2)$ in Eq. (1)

$$T(n) = 2[2T(n/2^2) + n] + n$$

$$T(n) = 2^2T(n/2^2) + 2n \qquad (2)$$

Substitute $T(n/2^2)$ in Eq. (2)

$$T(n) = 2^2[2T(n/2^3) + n] + 2n$$

$$T(n) = 2^3T(n/2^3) + 3n \qquad (3)$$

Substitute $T(n/2^3)$ in Eq. (3) and so on up to $k-1$

:

We will have

$$T(n) = 2^{k-1}[2T(n/2^k) + n] + (k-1)n$$

$$T(n) = 2^kT(n/2^k) + kn \qquad (k)$$

Find $T(n/2)$ value

Since we have

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

Therefore,

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

Find $T(n/2^2)$ value

$$T(n/2^2) = 2T(n/2^2/2) + n$$
$$= 2T(n/2^3) + n$$

Assume $2^k = n$ in Eq. (k), for this recurrence comes to a halt.

$$T(n) = nT(n/n) + kn$$
$$= nT(1) + kn$$
$$= n + nk$$
$$= n + n\log n$$

/* we ignore n because n log n is much higher term */

If $2^k = n$, then $k = \log n$

$$T(n) = O(n\log n)$$

Quick Sort

```
input: list /* unsorted array */
output: list /* sorted array */
quicksort(list, low, high)-> (list)
    if low > high then
        partition(list, low, high) -> (pivotIndex) /* list[pivot] is now at right place */
        quicksort(list, low, pivotIndex - 1) /* all elements left to pivot is < list[pivot] */
        quicksort(list, pivotIndex + 1, high) /* all elements right to pivot is >= list[pivot] */
end
partition(list, low, high) -> (pivotIndex)
    pivotElement = list[high] /* pivot is partitioning index, which is to be placed at right place */
    i = low – 1 /* index of smaller element */
    for j from  low to high -1 do
        if list[j] <= pivotElement
            i = i + 1 /* check next smaller index */
            Swap list[i] with list[j]
    Swap list[i+1] and a[high]
    return i+1 /* pivot is partitioning index */
end
```

Quick Sort

Partitioning Algorithm Illustration

Example from Cormen T. (Ch 7. 2009)

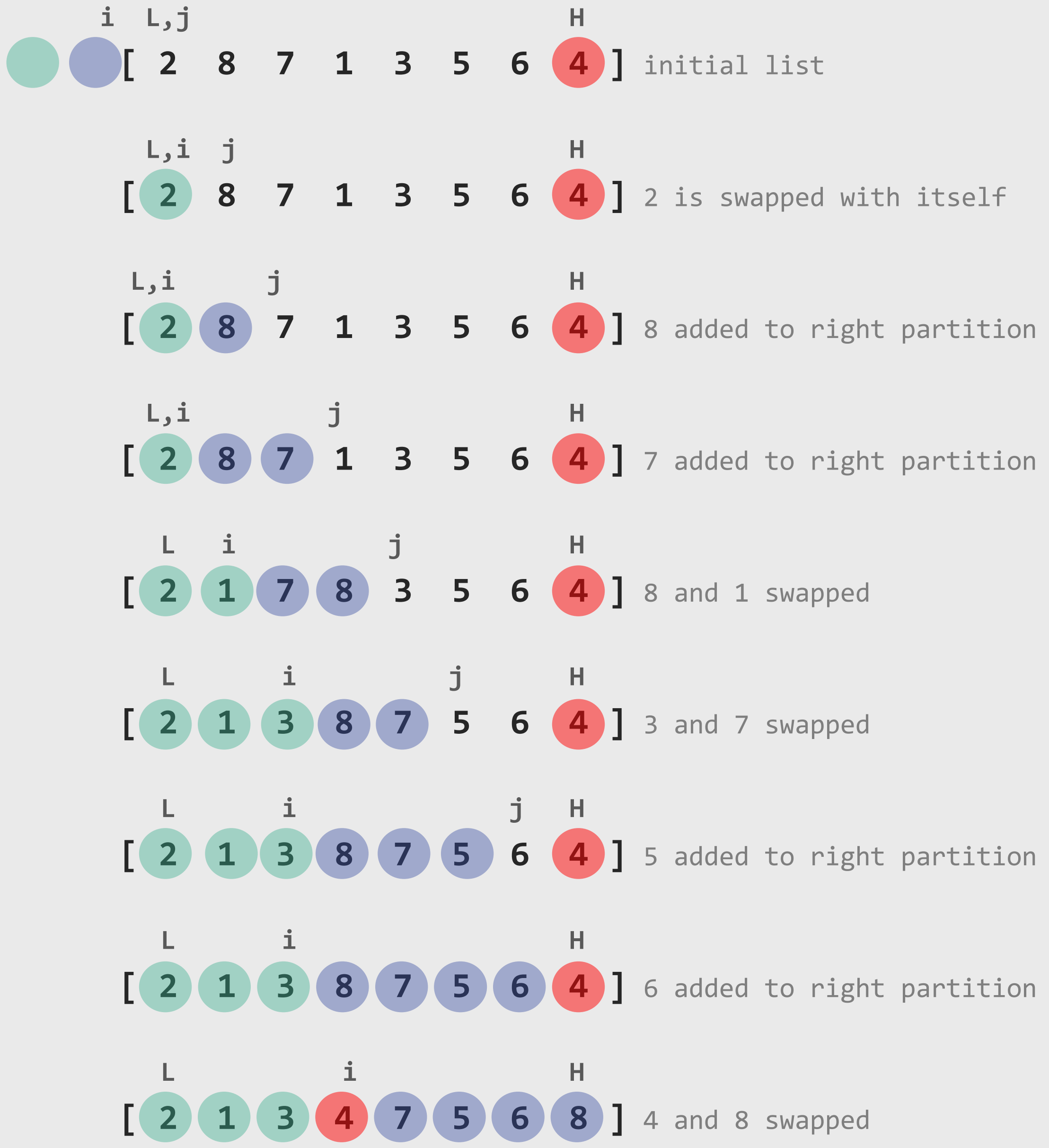
```
partition(list, L, H) -> (pivotIndex)
    pivotElement = list[high]
    i = low - 1 /* index of smaller element */
    for j from  low to high -1 do
        if list[j] <= pivotElement
            i = i + 1 /* next smaller index */
            Swap list[i] with list[j]
    Swap list[i+1] and a[high]
    return i+1 /* pivot index */
end
```

Pivot Element, list[hight]

Left partition

Right partition

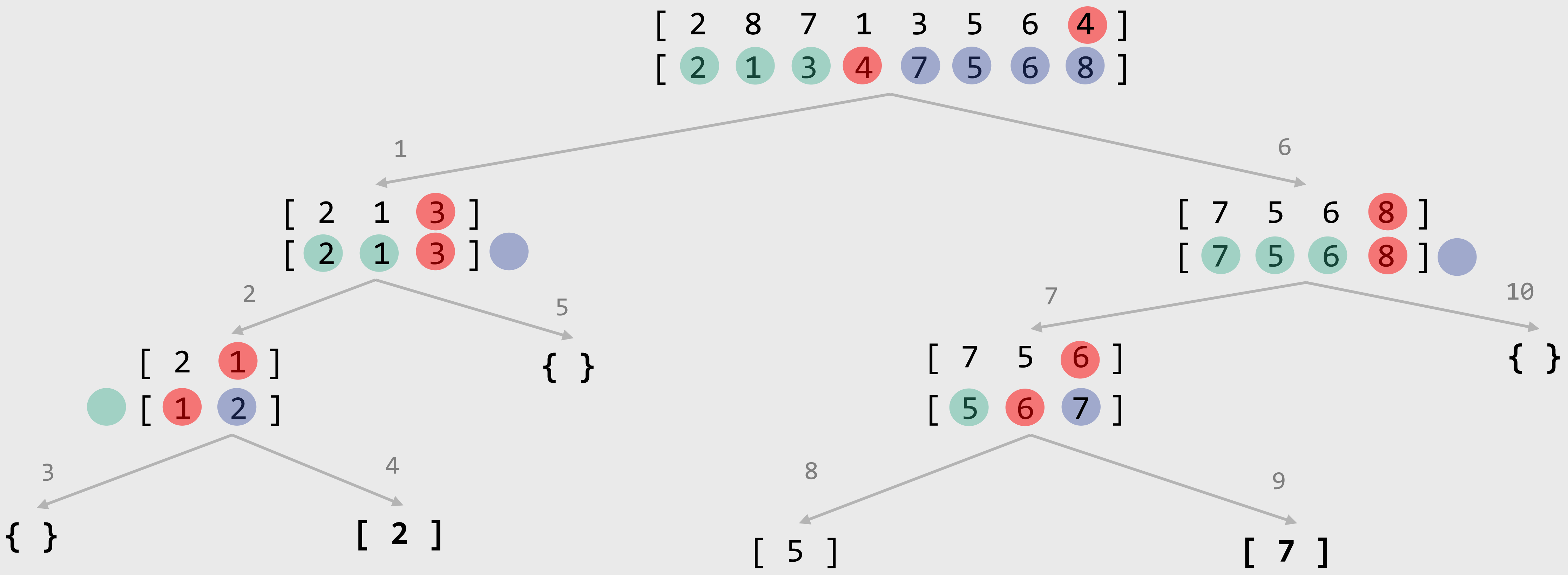
L Low index
H High index
i Pivot index



Quick Sort

Partitioning Algorithm Illustration

Example from Cormen T. (Ch 7. 2009)



Quick Sort: Complexity

```
input: list /* unsorted array */
output: list /* sorted array */
quicksort(list, low, high)-> (list)
    if low > high then
        partition(list, low, high) ->
        (pivotIndex) /* list[pivot] is now
        at right place */
        quicksort(list, low, pivotIndex - 1)
        /* all elements left to pivot is <
        list[pivot] */
        quicksort(list, pivotIndex + 1,
        high)
        /* all elements right to pivot is >=
        list[pivot] */
end
```

$T(n)$ ← partition

$T(n/2)$ ← quicksort(list, low, pivotIndex - 1)

$T(n/2) - 1$ ← quicksort(list, pivotIndex + 1, high)

- **Divide** (partition) takes constant time $T(n)$
- **Concur** (sort) operates on two sublists of length $\frac{n}{2}$. Hence, it takes $2T(n/2)$.
- **Combine** (sub-arrays already sorted) No operations to done here $T(0)$.
- Thus, the average time order is:

$$T(n) = 2T\left(\frac{n}{2}\right) + n$$

$$= O(n \log n)$$

Exercise: What is worst-case time complexity?

Fundamental of Computer Science
CS1FC16: Lecture 04, Part – III

Non-comparison techniques $O(n)$

Dr Varun Ojha
Department of Computer Science
University of Reading

Radix/Bucket Sort

- The time complexity of sorting depends on the number of comparisons
- Radix sort uses the key to sort elements in one shot, without comparing pairs of elements
- This has time $O(n)$ which is utterly stupendous!
- Radix sort uses buckets (arrays or lists of data) so it is often called *bucket sort*

Radix Sort: Example

- **Input:** (310,213,023,130,013,301,222,032,201,111,323,002,330,102,231,120)
- **Pass 1: units**

Bucket	Content
0	310, 130, 330, 120
1	301, 201, 111, 231
2	222, 032, 002, 102
3	213, 023, 013, 323

- **Output:** (310,130,330,120,301,201,111,231,222,032,002,102,213,023,013,323)

Radix Sort: Example

- **Input** (comes from pass 1)**:** (310,130,330,120,301,201,111,231,222,032,002,102,213,023,013,323)
- **Pass 2: tens**

Bucket	Content
0	301, 201, 002, 102
1	310, 111, 213, 013
2	120, 222, 023, 323
3	130, 330, 231, 032

- **Output:** (301,201,002,102,310,111,213,013,120,222,023,323,130,330,231,032)

Radix Sort: Example

- **Input** (comes from pass 2)**:** (301,201,002,102,310,111,213,013,120,222,023,323,130,330,231,032)
- **Pass 3: hundreds**

Bucket	Content
0	002, 013, 023, 032
1	102, 111, 120, 130
2	201, 213, 222, 231
3	301, 310, 323, 330

- **Output:** (002,013,023,032,102,111,120,130,201,213,222,231,301,310,323,330)

Time Order of Radix Sort

- Let n be the number of elements in the list
- Let k be the number of digits in the key
- Each digit of the key is examined once per list, so radix/bucket sort is $O(kn)$
- However, $k \ll n$ in practical cases and, in any case, k is a constant, so time is $O(n)$

Summary (1/3)

- If there is a small number of elements, then insertion sort or bubble sort may be the quickest algorithms, because they are so simple
- Insertion sort and bubble sort are particularly quick if the elements are almost sorted
- Merge sort and Quick sort are effective algorithms to use in a divide and conquer algorithms
- If there is a moderate or large number of elements, then a divide and conquer algorithm, such as merge sort or quicksort may be highly effective, but quicksort is very slow on sorted data!

Summary (2/3)

- Bucket sort is exceptionally fast, but it requires a lot of memory and is suitable only if data are encoded with short keys
- There is no best algorithm – everything depends on circumstances
- Commercial sorting packages compute statistics on the elements before selecting a sorting algorithm

Summary (3/3)

- Simple sorting algorithms have time $O(n^2)$. They are usually only useful for small numbers of data or data that are highly sorted
- Divide and conquer algorithms have time $O(n \log n)$. They are usually only useful for moderate to large numbers of data
- Bucket algorithms have time $O(n)$. They are usually only useful for very large numbers of data indexed by short keys, but they do consume a lot of memory

Fundamental of Computer Science
CS1FC16: Lecture 04, Part – IV

Exercises

Dr Varun Ojha
Department of Computer Science

University of Reading

Exercise

- Write a C++ program of insertion sort and selection sort and Compare how insertion sort is different from selection sort?
- How can you make babble sort of this lecture algorithm little more efficient?
- Complexity of bubble sort is $O(n^2)$. Show that this statement is true.