Sorting Algorithms

Merge Sort

Quick Sort

1

Data Structure and Algorithms

Deepali Londhe
Information Technology,
PICT, Pune

2

Agenda

- Searching and sorting
- Concept of internal and external sorting
- Sort stability
- Sorting methods: Bubble, insertion, Quick, Merge, shell and comparison of all sorting methods.
- Case Studies Set Operation, String Operation
- Fibonacci Series.

3

Sorting Algorithms

- Elementary Techniques
  - Bubble Sort
  - Insertion Sort
  - Shell Sort
- Two classic Algorithms- Divide and Conquer
  - Quick Sort
  - Merge Sort

4

Overview

- Divide and Conquer
- Merge Sort
- Quick Sort

5

5

Divide and Conquer

1. **Base Case**, solve the problem **directly** if it is small enough
2. **Divide** the problem into two or more **similar and smaller** subproblems
3. **Recursively** solve the subproblems
4. **Combine** solutions to the subproblems

6

6

Divide and Conquer - Sort

Problem:

- Input: A[left..right] – **unsorted** array of integers
- Output: A[left..right] – **sorted** in non-decreasing order

7

7

Divide and Conquer - Sort

1. **Base case**
   at most one element (left ≥ right), return
2. **Divide** A into two subarrays: FirstPart, SecondPart
   Two Subproblems:
   sort the FirstPart
   sort the SecondPart
3. **Recursively**
   sort FirstPart
   sort SecondPart
4. **Combine** sorted FirstPart and sorted SecondPart

8

8

Overview

- Divide and Conquer
- Merge Sort
- Quick Sort

9

9

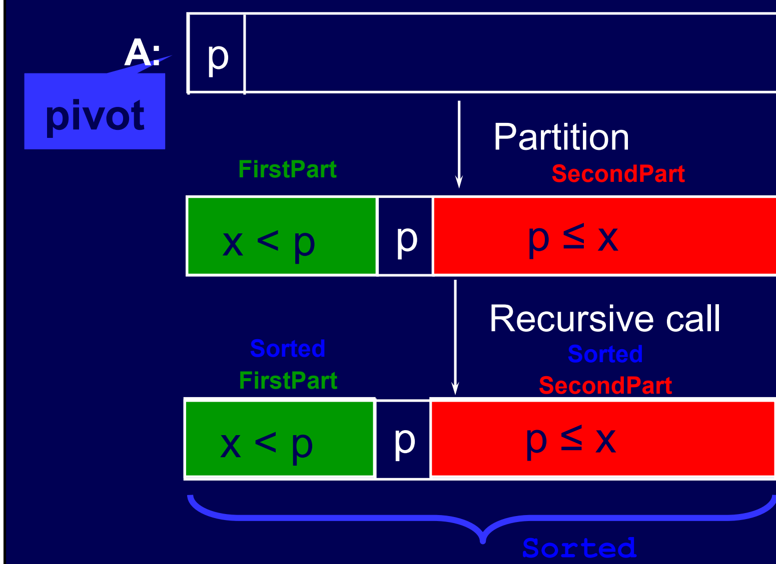
Quick Sort

- Divide:
  - Pick any element p as the pivot, e.g, the first element
  - Partition the remaining elements into
    - FirstPart, which contains all elements < p
    - SecondPart, which contains all elements ≥ p
- Recursively sort the FirstPart and SecondPart
- Combine: no work is necessary since sorting is done in place

10

10

Quick Sort



11

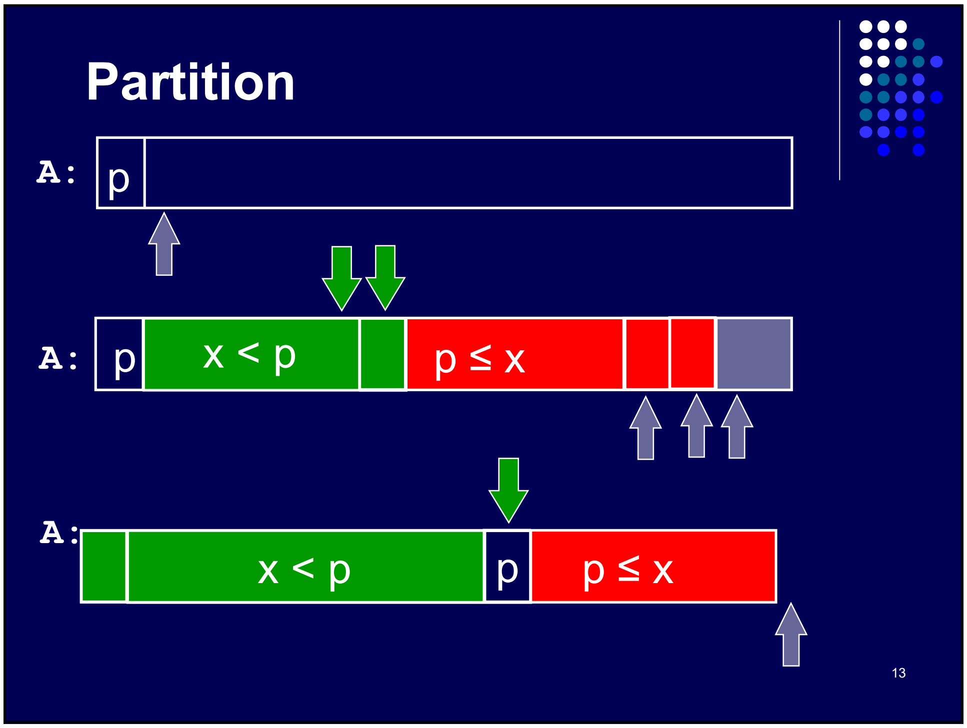
11

Quick Sort

```
Quick-Sort(A, left, right)
  if    left ≥ right  return
  else
        middle ← Partition(A, left, right)
        Quick-Sort(A, left, middle-1 )
        Quick-Sort(A, middle+1, right)
  end if
```

12

12



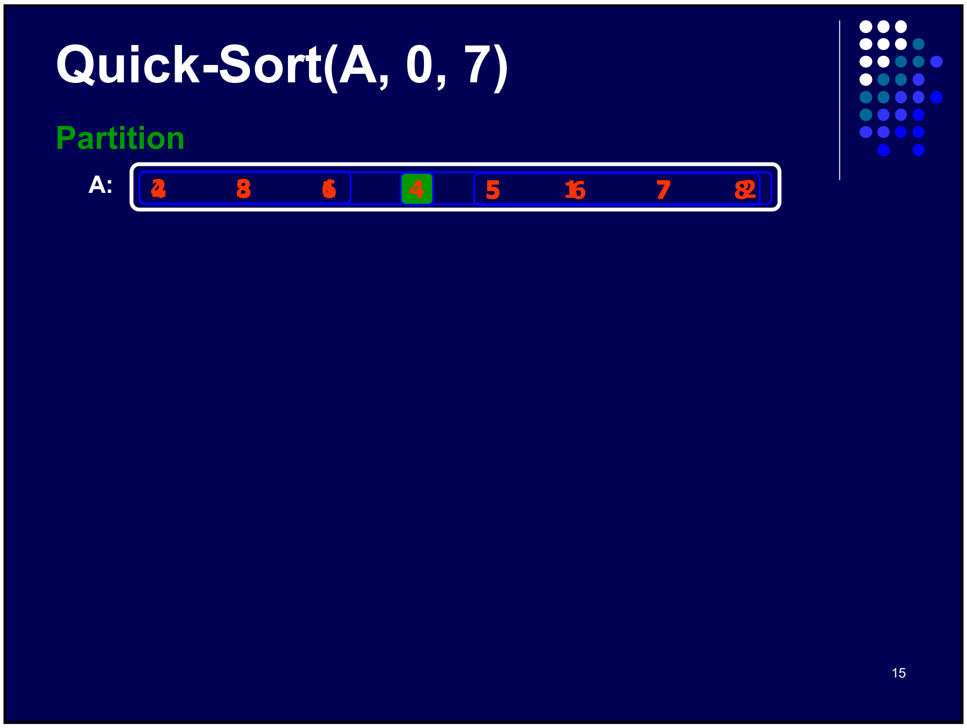
13

```
Partition(A, left, right)
    P ← A[right]
    i ← left-1
    for j ← left to right
        if A[j] < P then
            i ← i + 1
            swap(A[i], A[j])
        end if
    end for j
    swap(A[i+1], A[right])
    return i+1
```

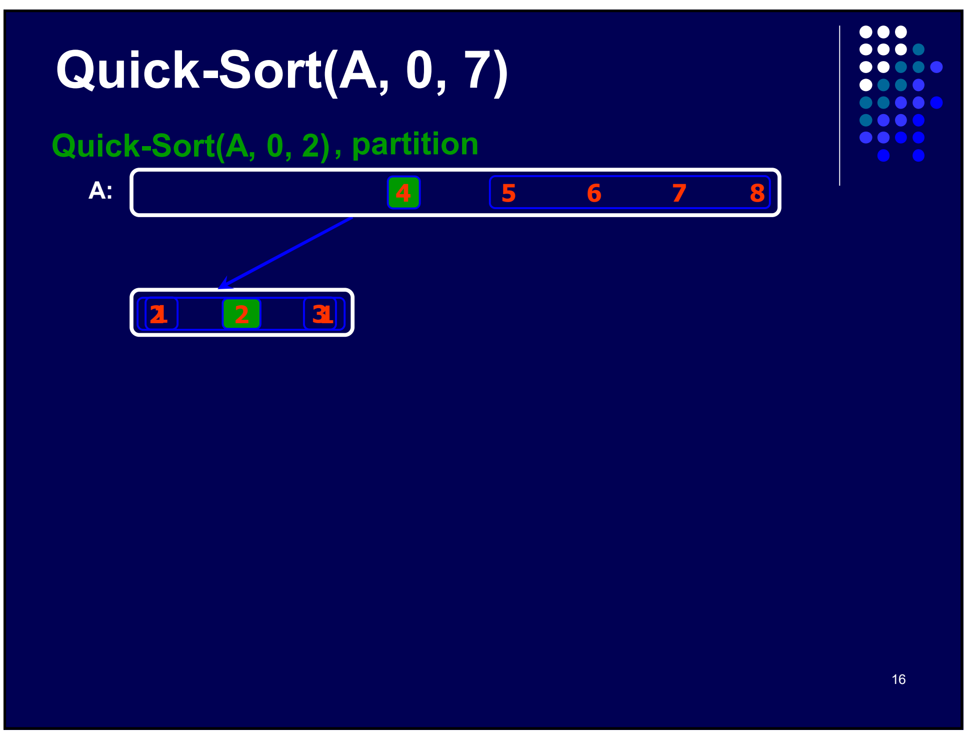
n = right – left +1
Time: cn for some constant c
Space: constant

14

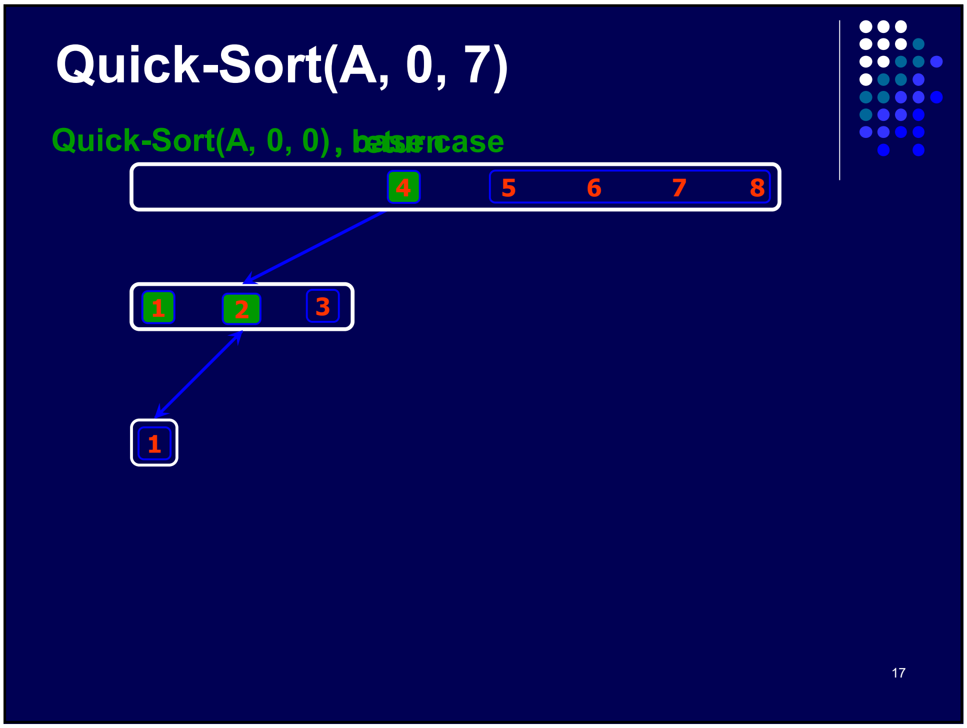
14


Quick-Sort(A, 0, 7)
Partition
A:
15

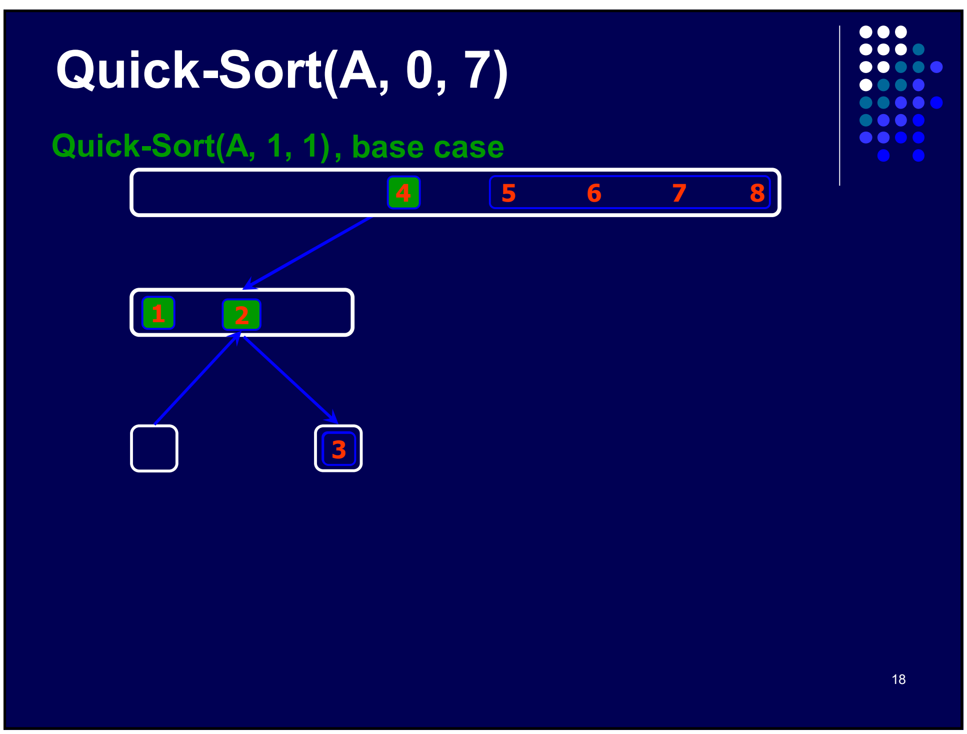
15


Quick-Sort(A, 0, 7)
Quick-Sort(A, 0, 2), partition
A:
4 5 6 7 8
16

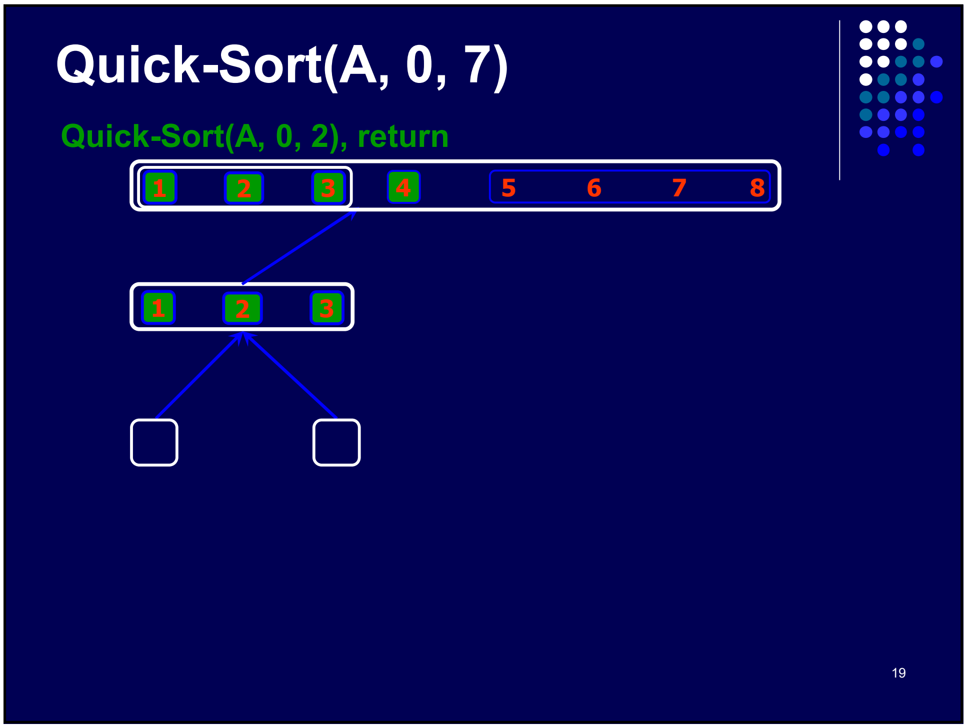
16


Quick-Sort(A, 0, 7)
Quick-Sort(A, 0, 0), base case
4
5 6 7 8
1 2 3
1
17

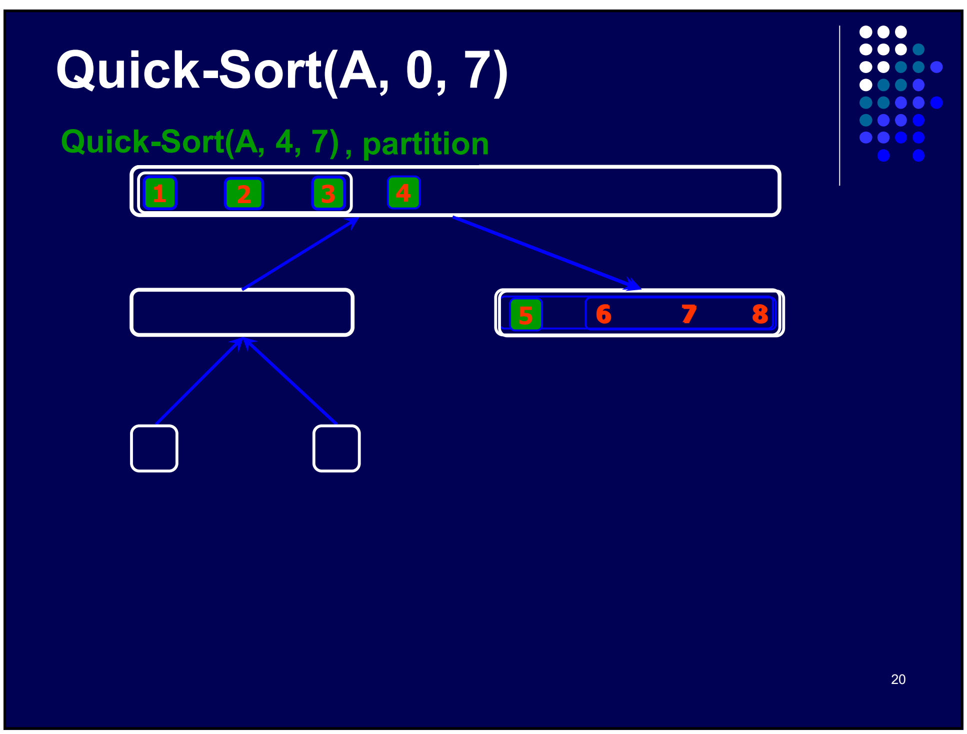
17


Quick-Sort(A, 0, 7)
Quick-Sort(A, 1, 1), base case
4
5 6 7 8
1 2
3
18

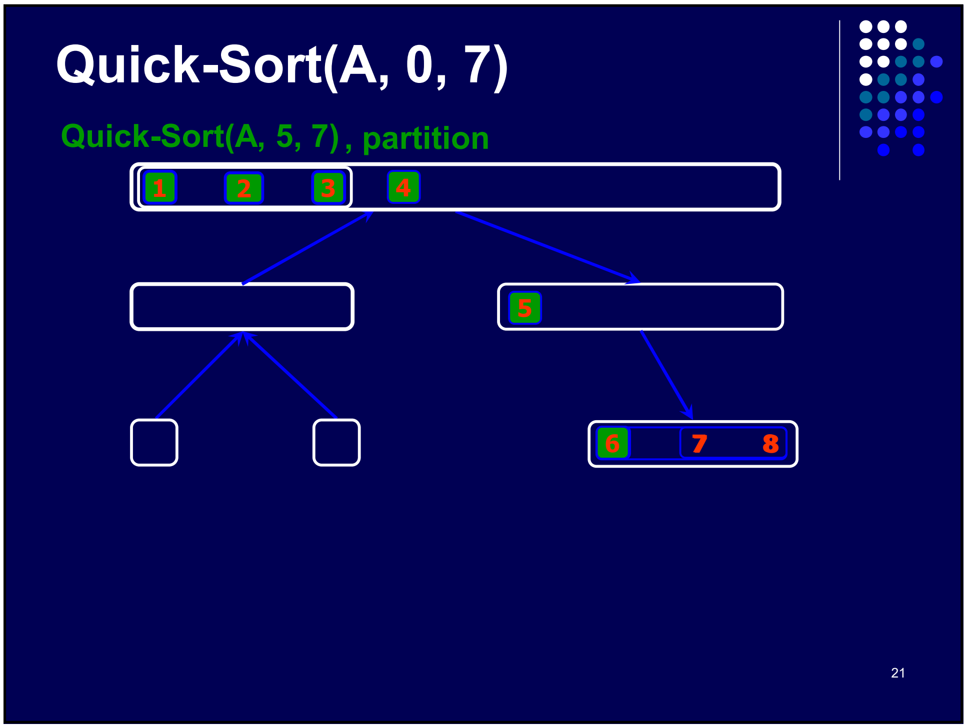
18


Quick-Sort(A, 0, 7)
Quick-Sort(A, 0, 2), return
1 2 3 4 5 6 7 8
1 2 3
19

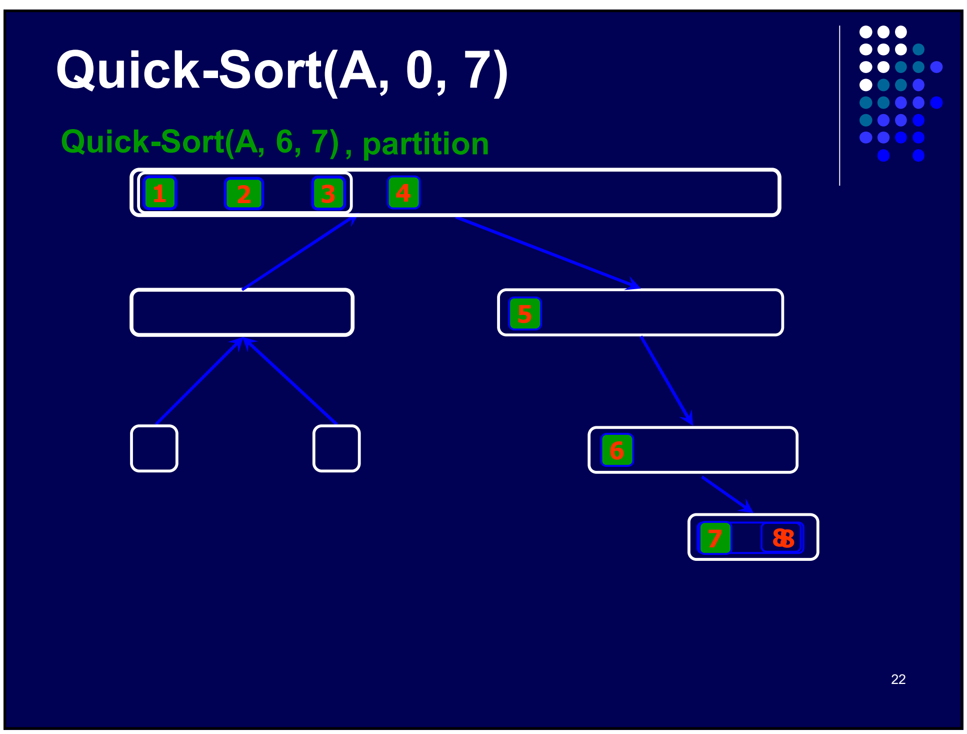
19


Quick-Sort(A, 0, 7)
Quick-Sort(A, 4, 7) , partition
1 2 3 4
5 6 7 8
20

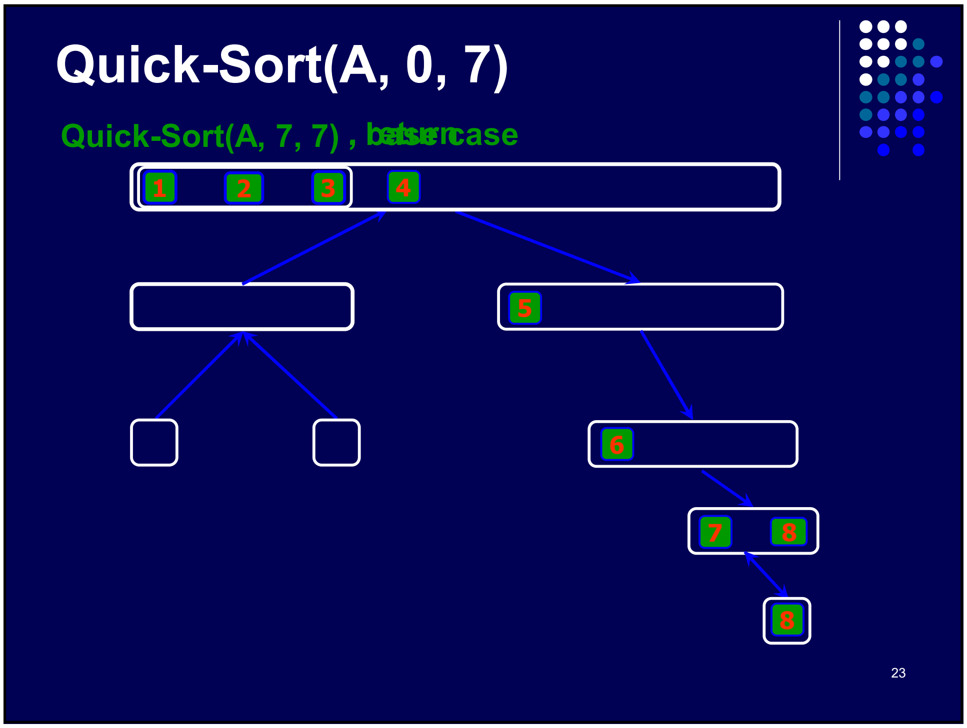
20


Quick-Sort(A, 0, 7)
Quick-Sort(A, 5, 7) , partition
1 2 3 4
5
6 7 8
21

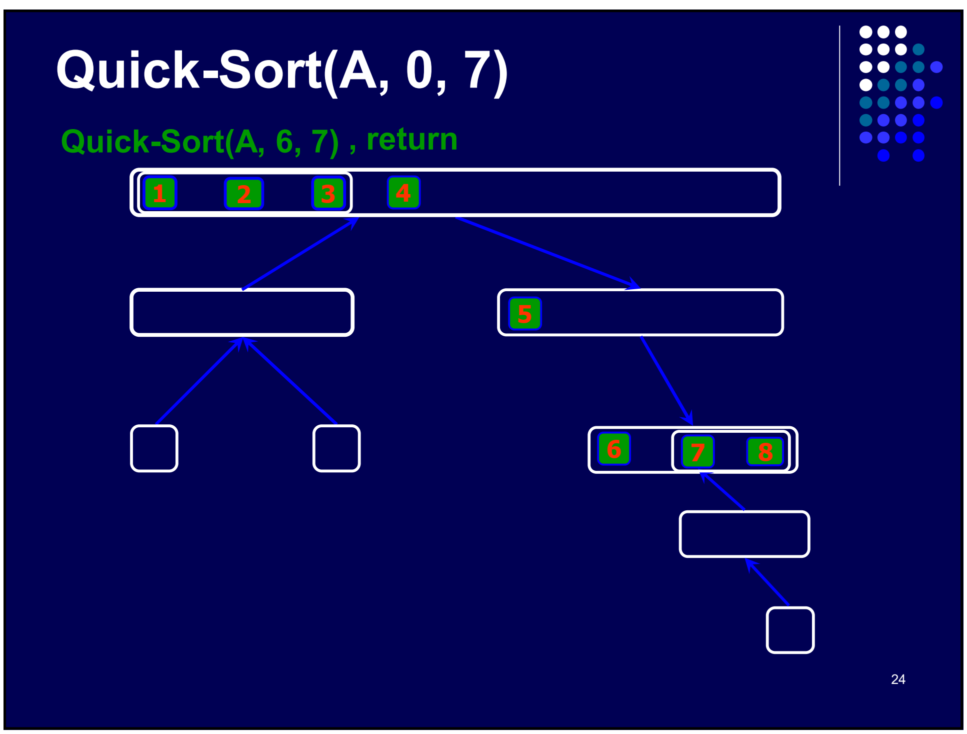
21


Quick-Sort(A, 0, 7)
Quick-Sort(A, 6, 7) , partition
1 2 3 4
5
6
7 8
22

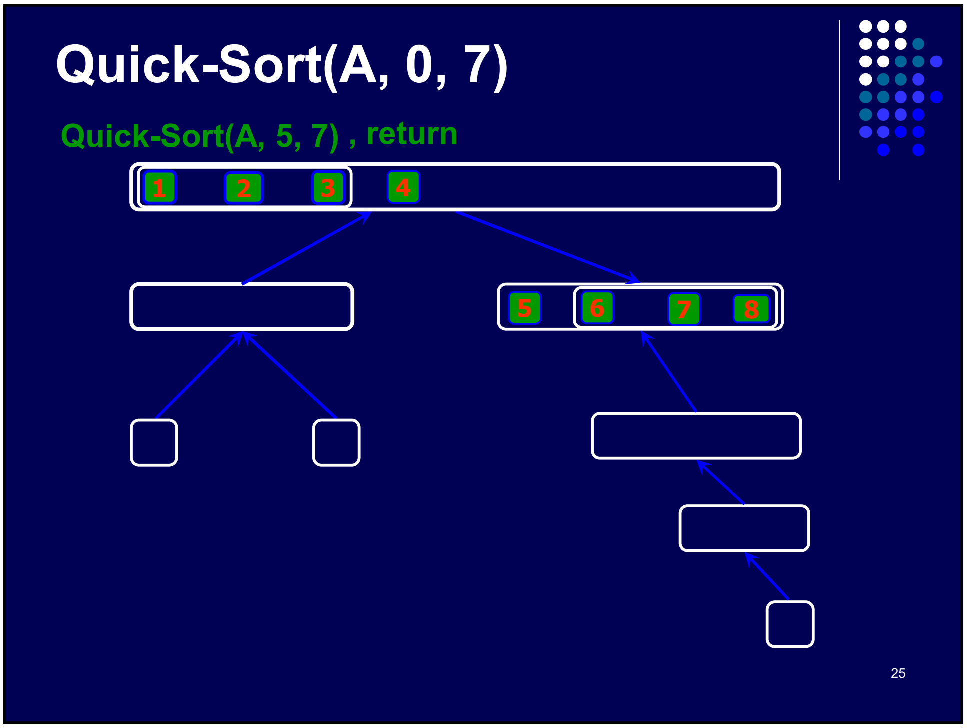
22


Quick-Sort(A, 0, 7)
Quick-Sort(A, 7, 7) , base case
1 2 3 4
5
6
7 8
8
23

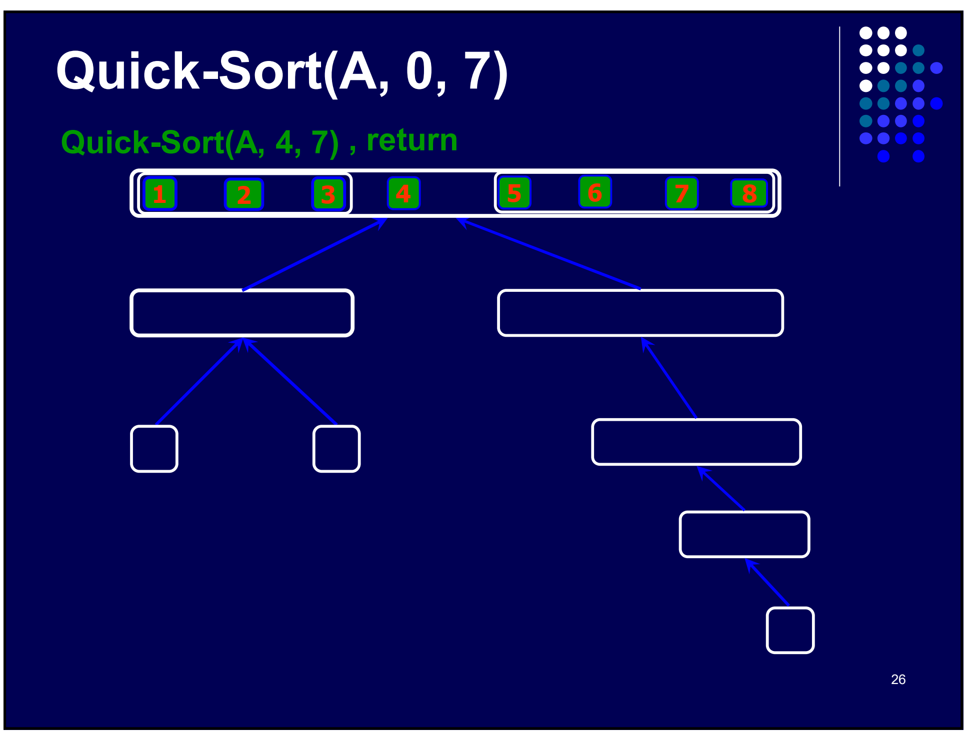
23


Quick-Sort(A, 0, 7)
Quick-Sort(A, 6, 7) , return
1 2 3 4
5
6 7 8
24

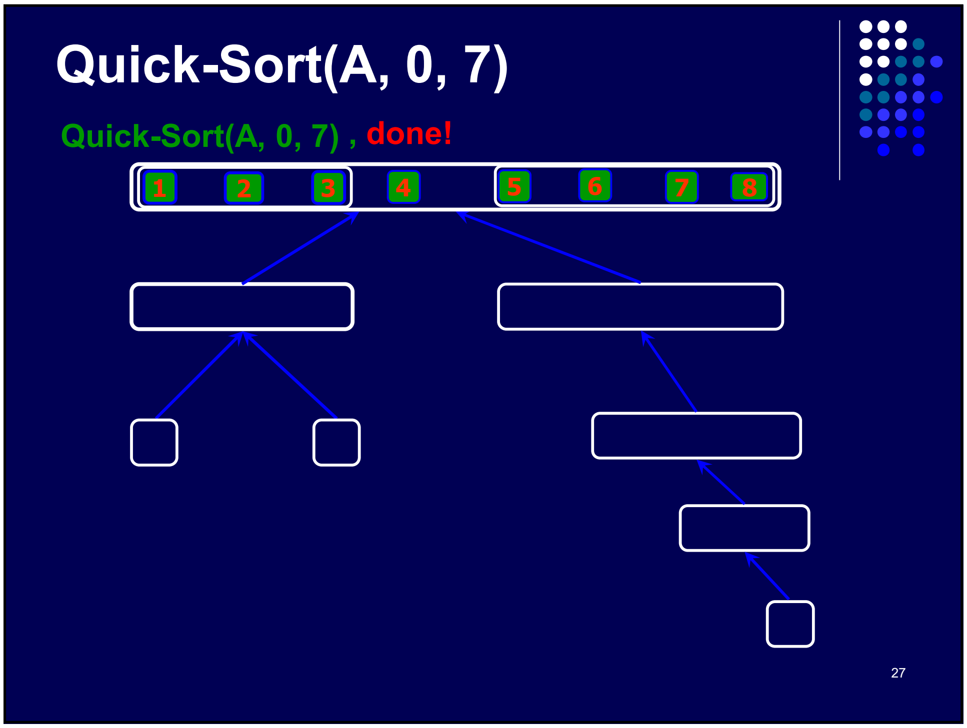
24


Quick-Sort(A, 0, 7)
Quick-Sort(A, 5, 7) , return
1
2
3
4
5
6
7
8
25

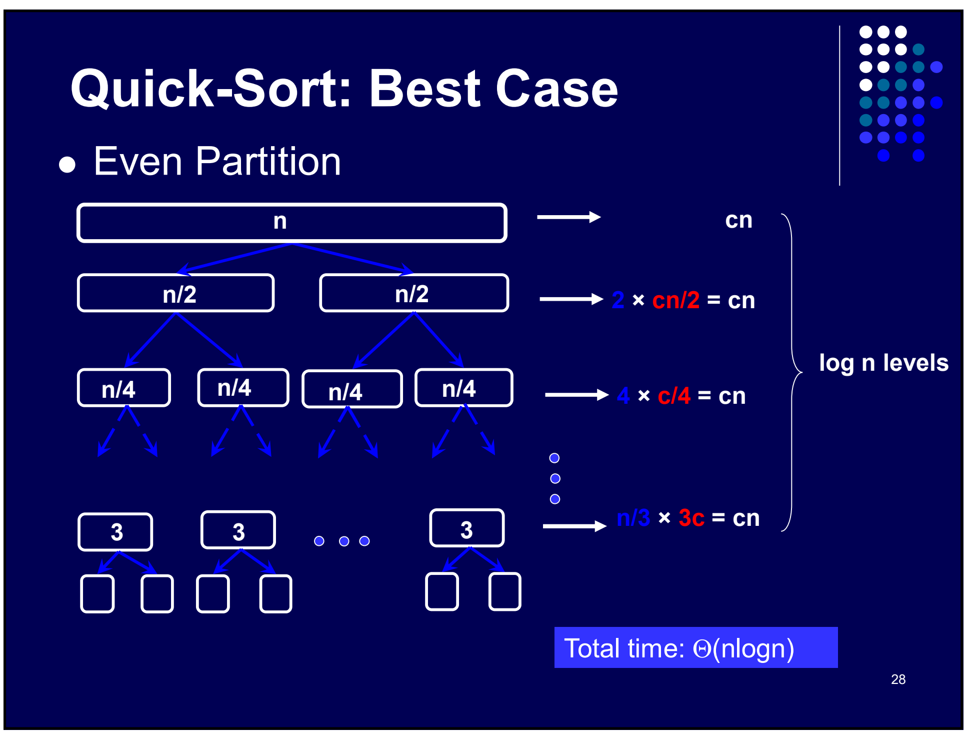
25


Quick-Sort(A, 0, 7)
Quick-Sort(A, 4, 7) , return
1
2
3
4
5
6
7
8
26

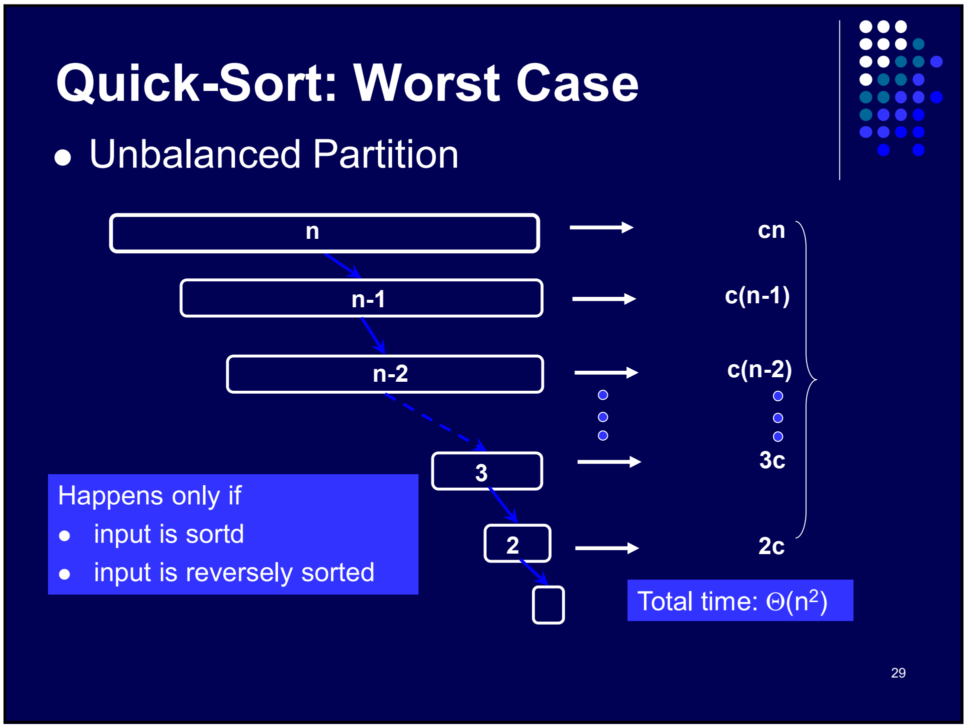
26


Quick-Sort(A, 0, 7)
Quick-Sort(A, 0, 7) , done!
1
2
3
4
5
6
7
8
27

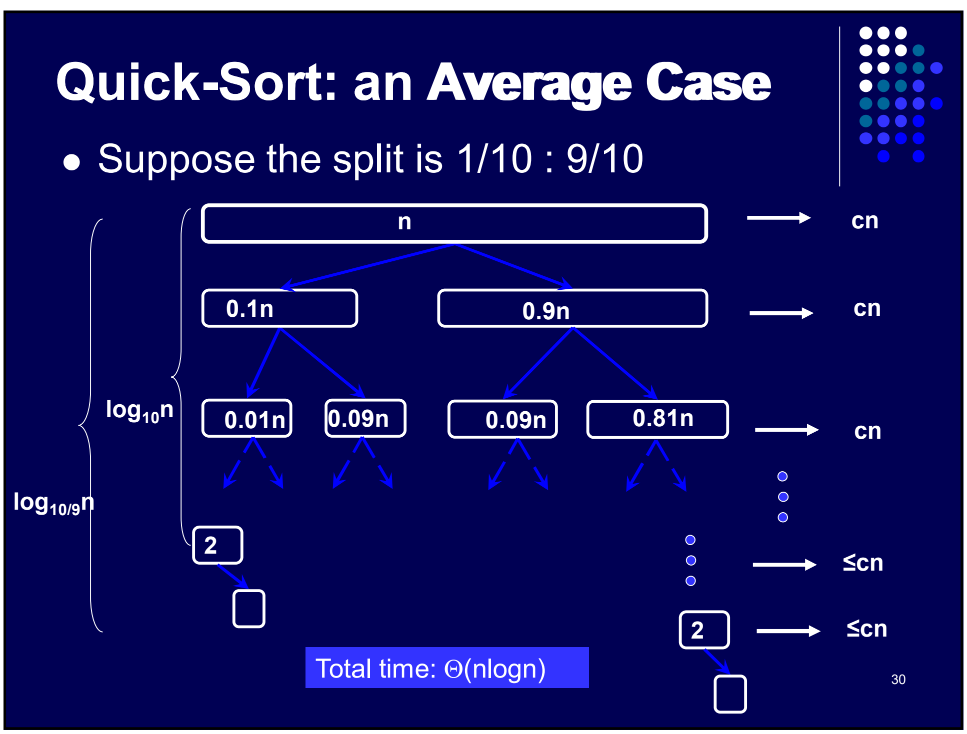
27


Quick-Sort: Best Case
Even Partition
n
n/2
n/2
n/4
n/4
n/4
n/4
3
3
3
cn
2 × cn/2 = cn
4 × c/4 = cn
n/3 × 3c = cn
log n levels
Total time: Θ(nlogn)
28

28


Quick-Sort: Worst Case
Unbalanced Partition
n
n-1
n-2
3
2
cn
c(n-1)
c(n-2)
3c
2c
Happens only if
input is sortd
input is reversely sorted
Total time: Θ(n2)
29

29


Quick-Sort: an Average Case
Suppose the split is 1/10 : 9/10
n
0.1n
0.9n
0.01n
0.09n
0.09n
0.81n
2
2
log10n
log10/9n
cn
cn
cn
≤cn
≤cn
Total time: Θ(nlogn)
30

30

Quick Sort Simplified

```
qsort(int a[],int l,int r)
    if (l>=r)   return;
    i=l;
    j=r+1;
    p=a[l];
while(1)
{        do{  i++;  } while(a[i]<p);
         do{  j--;  } while(a[j]>p);
         if(i>=j)  break;
         Swap(a[i],a[j])              }
a[l]=a[j];
a[j]=p;
qsort(a,l,j-1);
qsort(a,j+1,r);
```

31

31

Quick-Sort Summary

- Time
  - Most of the work done in partitioning.
  - Average case takes **Θ(n log(n))** time.
  - Worst case takes **Θ(n^2)** time

Space
  - Sorts in-place, i.e., does not require additional space

32

32

Summary

- Divide and Conquer
- Merge-Sort
  - Most of the work done in Merging
  - **Θ(n log(n))** time
  - **Θ(n) space**
- Quick-Sort
  - Most of the work done in partitioning
  - Average case takes **Θ(n log(n))** time
  - Worst case takes **Θ(n²)** time
  - **Θ(1) space**

33

33

- **Stability**
- We say that a sorting algorithm is *stable* if, when two records have the same key, they stay in their original order. This property will be important for extending bucket sort to an algorithm that works well when k is large. But first, which of the algorithms we've seen is stable?
- Bucket sort? Yes. We add items to the lists Y[i] in order, and concatenating them preserves that order.
- Heap sort? No. The act of placing objects into a heap (and heapifying them) destroys any initial ordering they might have.
- Merge sort? Maybe. It depends on how we divide lists into two, and on how we merge them. For instance if we divide by choosing every other element to go into each list, it is unlikely to be stable. If we divide by splitting a list at its midpoint, and break ties when merging in favor of the first list, then the algorithm can be stable.
- Quick sort? Again, maybe. It depends on how you do the partition step.
- Any comparison sorting algorithm can be made stable by modifying the comparisons to break ties according to the original positions of the objects, but only some algorithms are automatically stable.

34

34

- **Comparison sorts:** A comparison sort examines elements with a comparison operator, which usually is the less than or equal to operator(≤). Comparison sorts include:
  - Bubble sort
  - Insertion sort
  - Selection sort
  - Shell sort
  - Heapsort
  - Mergesort
  - Quicksort.
- **Non-Comparison sorts:** these use other techniques to sort data, rather than using comparison operations. These include:
  - Radix sort (examines individual bits of keys)
  - Bucket sort (examines bits of keys)
  - Counting sort (indexes using key values)

35

35

References

- **Books**
- D. E. Knuth, *The Art of Computer Programming: Vol. 3: Sorting and Searching, 2d ed., Addison-*Wesley, Reading, Mass., 1998.
- SORTING AND SEARCHING ALGORITHMS: A COOKBOOK BY THOMAS NIEMANN
- Robert Sedgewick, Kevin Wayne, “Algorithms”, 4th edition, Addison-Wesley Professional
- Samanta Debasis, “**CLASSIC DATA STRUCTURES”, PHI, 2nd ed.**
- Ellis Horowitz and Sartaj Sahni , “Fundamentals of Data Structures”, Computer Science Press, 1983
- R. Gilberg, B. Forouzan, "Data Structures: A pseudo Code Approach with C++", Cengage Learning, ISBN 9788131503140.
- E. Horowitz, S. Sahni, D. Mehta, "Fundamentals of Data Structures in C++", Galgotia Book Source, New Delhi, 1995, ISBN 16782928
- Dinesh P. Shah, Sartaj Sahani , “Handbook of DATA STRUCTURES and APPLICATIONS”, CHAPMAN & HALL/CRC
- Bayer B. et al. (2015) Electro-Mechanical Brake Systems. In: Winner H., Hakuli S., Lotz F., Singer C. (eds) Handbook of Driver Assistance Systems. Springer, Cham
- **Web**
  - http://statmath.wu.ac.at/courses/data-analysis/itdtHTML/node55.html
  - https://en.wikipedia.org/wiki/Persistent_data_structure

36

Thank You !!!

37

Quick sort videos

- https://www.youtube.com/watch?v=cNB5JCG3vts
- https://www.youtube.com/watch?v=ywWBy6J5gz8

38

38