# M4. Array
# (ADT/DS, Searching, and Sorting)

Instructor: Manikandan Narayanan

Weeks 3-4

CS2700 (PDS) Moodle: https://courses.iitm.ac.in/course/view.php?id=4892

# Acknowledgment of Sources

- Slides based on content from related
  - Courses:
    - IITM – Profs. **Rupesh**/Krishna(S)/Prashanth/Kartik's PDS (Thy/Lab) offerings (slides, quizzes, notes, lab assignments, etc. for instance from Rupesh's Jul 2019 offering - www.cse.iitm.ac.in/~rupesh/teaching/pds/jul19/ )
    - *Most slides are based on Rupesh Nasre's slides – we thank him and acknowledge by marking **[RN]** in the bottom right of these slides.*
    - *Array ADT vs. DS view from brilliant.org:*

    *https://brilliant.org/wiki/arrays-adt/* *https://brilliant.org/wiki/abstract-data-types/*

  - Books:
    - **Main textbook:** *"Data Structures and Algorithm Analysis in C++"* by ***Weiss*** (content, figures, slides, exercises/questions, etc.). – cited as [WeissBook]
    - Additional/optional book: *"Practice of Programming"* by *Kernighan and Pike* (style of programming, programming exercises/questions, etc.) – cited as [KPBook]

# Outline for Module M4

- M4 Array
  - **M4.1 Array ADT vs. DS (incl. Matrix and Applications)**
  - M4.2 Searching algorithms (linear and binary search, 2D binary search)
  - M4.3 Sorting algorithms (bubble/insertion/selection sort, quicksort, mergesort/heapsort algos.)
  - M4.4 Comparison-based sorting model (running time lower bound)
  - M4.5 Other sorting models/algorithms (bucket sort algo. and cousins)

# ADT

- Abstract Data Type
- Defines the interface of the functionality provided by the data structure.
- Hides implementation details.
  - Defines *what* and hides *how*.
- Makes software modular.
- Allows easy change of implementation.

[RN]

# Array (aka Vector) as an ADT

Array: ordered collection of items (of the same primitive or complex data type) accessible by an integer index

```
class Array {
public:
  Array();
  //minimal reqd. functionality:
  void set(int i, Element v);
  Element get(int i);

  //optional:
  int find(Element e);
  void print();
  int size();
};
```

Key Property: Store/retrieve elements using an integer index (position).

What are the complexities of these operations?

[RN]

# Other ADTs

- Queue ADT (FIFO)
  - enqueue, dequeue
- Set ADT
  - union, find, intersection, complement, etc.
- Fan regulator
  - IncSpeed, decSpeed, getSpeed, getCompanyName
- Integer
  - size, isSigned, getValue, setValue, add, sub
- Student
  - getRollNo, getHostel, getFavGame, setHostel, getSlots, setCGPA

[RN]

# This course is…

…all about such **ADTs**, their implementations (**DS**), and their **prog. applications**!

cse.iitm.ac.in/course_details.php?arg=ODg=

New

- Design correct programs to solve problems.
- Choose efficient data structures and apply them to solve problems.
- Analyze the efficiency of programs based on time complexity.
- Prove the correctness of a program using loop invariants, pre-conditions and post-conditions in programs.

Course Contents:

- Review of Problem Solving using computers, Abstraction, Elementary Data Types. Algorithm design- Correctness via Loop invariants as a way of arguing correctness of programs, preconditions, post conditions associated with a statement. (3 lectures)
- Complexity and Efficiency via model of computation (notion of time and space), mathematical preliminaries, Elementary asymptotics (big-oh, big-omega, and theta notations). (3 lectures)
- ADT Array -- searching and sorting on arrays:Linear search, binary search on a sorted array. Bubble sort, Insertion sort, Merge Sort and analysis; Emphasis on the comparison based sorting model. Counting sort, Radix sort, bucket sort. (6 lectures)
- ADT Linked Lists, Stacks, Queues:List manipulation, insertion, deletion, searching a key, reversal of a list, use of recursion to reverse/search. Doubly linked lists and circular linked lists. (3 lectures)
- Stacks and queues as dynamic data structures implemented using linked lists. Analyse the ADT operations when implemented using arrays. (3 lectures)
- ADT Binary Trees:Tree representation, traversal, application of binary trees in Huffman coding. Introduction to expression trees: traversal vs post/pre/infix notation. Recursive traversal and other tree parameters (depth, height, number of nodes etc.) (4 lectures)
- ADT Dictionary: Binary search trees, balanced binary search trees - AVL Trees. Hashing - collisions, open and closed hashing, properties of good hash functions. (3+3 lectures)
- ADT Priority queues: Binary heaps with application to in-place sorting (5 lectures)
- Graphs: Representations (Matrix and Adjacency List), basic traversal techniques: Depth First Search + Breadth First Search (Stacks and Queues) (5 lectures)
  (**Note :** The ADTs will be taught using C++, introducing its syntax as required to explain the concepts (such as objects, classes, encapsulation, operator overloading, polymorphism and basic STL such as string and vector).

# ADT (interface) vs. DS (impl.)

| **Abstract Data Type (ADT)** | **Other Common Names** | **Commonly Implemented with (DS)** |
|---|---|---|
| *Array* | *Vector* | *Array (static/dynamic)* |
| List | Sequence | Array, Linked List |
| Queue | | Array, Linked List |
| Double-ended Queue | Dequeue, Deque | Array, Doubly-linked List |
| Stack | | Array, Linked List |
| Associative Array | Dictionary, Hash Map, Hash, Map** | Hash Table |
| Set | | Red-black Tree, Hash Table |
| Priority Queue | Heap | Heap |
| *Binary Tree…* | … | … |
| *Graph…* | … | … |

[From Abstract Data Types. *Brilliant.org*. Retrieved 06:42, August 13, 2024, from https://brilliant.org/wiki/abstract-data-types/ ]

# Implementing Array ADT using Array DS (contiguous memory locations)

```
class Array {
public:
  Array();
  //minimal reqd. functionality
  void set(int i, Element v);
  Element get(int i);

  //optional:
  int find(Element e);
  void print();
  int size();
};
```

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 4 | 2 | 7 | 2 | 9 | | | |

**Design decisions**

- Size of the array? (Static vs. Dynamic)
- Maintain size separately or use a sentinel?
- On overflow: error or realloc?
- On underflow: error message or exit or silent?
- Are duplicates allowed? If so, what should find return?
- ...

[RN]

# Array ADT using Array DS
## (static array of some max size)

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 4 | 2 | 7 | 2 | 9 | | | |

```
class Array {
public:
  Array();
  //minimal reqd. functionality
  void set(int i, Element v);
  Element get(int i);

  //optional:
  int find(Element e);
  void print();
  int size();
};
```

**With certain design decisions:**

**O(1)**

**O(1)**

**O(N)**

**O(N)**

**O(1)**

[RN]

# Implementation Details: Array DS (**contiguous memory** locations)

- Simplest data structure
  - Acts as aggregate over primitives or other aggregates
  - May have multiple dimensions
- Contiguous storage
- Random access in O(1)
- Languages such as C use type system to index appropriately
  - e.g., a[i] and a[i + 1] refer to locations (memory address) based on type
- Storage space:
  - Fixed for arrays
  - Dynamically allocatable but fixed on system’s stack or heap
  - Variable for vectors (internally, reallocation and copying)

[RN]

# Implementation Details: 2D Array (aka Matrix) DS

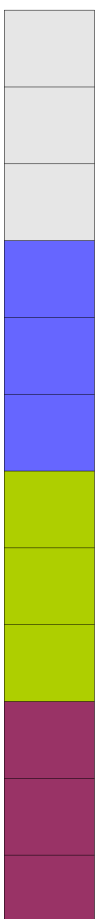
- Typically, 2D arrays stored as a **single 1D array** in contiguous memory locations…
  - …in row-major order in C/C++

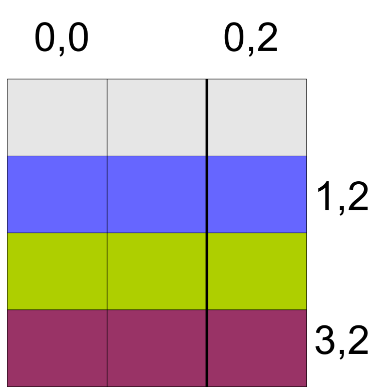
- Sometimes, 2D arrays stored as an **array of arrays** like so:
  - int *arr[Nrows]; … //(or)
  - vector< vector<int> > arr(Nrows);
    //for (int i=0; i < Nrows; i++) { arr[i].resize(Ncols); }

[RN]

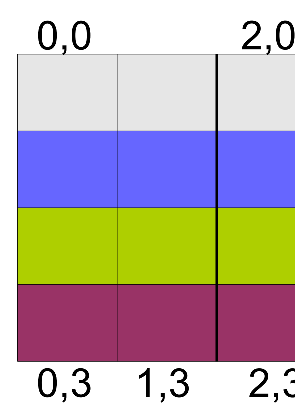
# Impl. Details (contd.): nD Array DS (as a **single 1D array**)

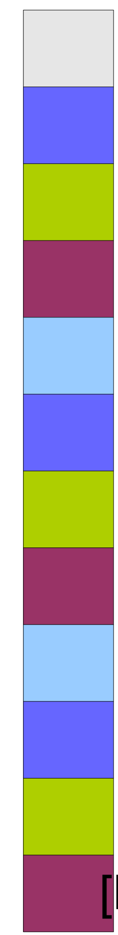
- In C, C++, Java, we use *row-major* storage.
  - All elements of a row are stored together.






- In Fortran, we use *column-major* storage.
  - each column is stored together.



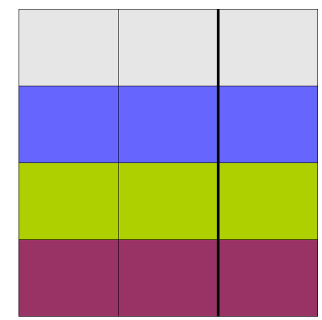
[RN]

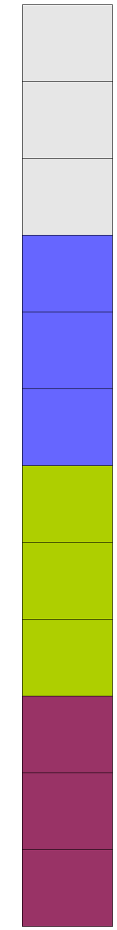
# Impl. Details (contd.): nD Array DS (as a **single 1D array** in C/C++)

```
void fun(int a[ ][ ]) {
        a[0][0] = 20;
}
void main() {
        int a[5][10];
        fun(a);
        printf("%d\n", a[0][0]);
}
```

**ERROR: type of formal parameter 1 is incomplete**

We view an array to be a D-dimensional matrix. However, for the hardware, it is simply single dimensional.





For declaration int a[w4][w3][w2][w1]:

- What is the address of a[i][j][k][l]?
  - (i * w3 * w2 * w1 + j * w2 * w1 + k * w1 + l) * 4
- How to optimize the computation?
  - Use **Horner's rule**: (((i * w3 + j) * w2 + k) * w1 + l) * 4

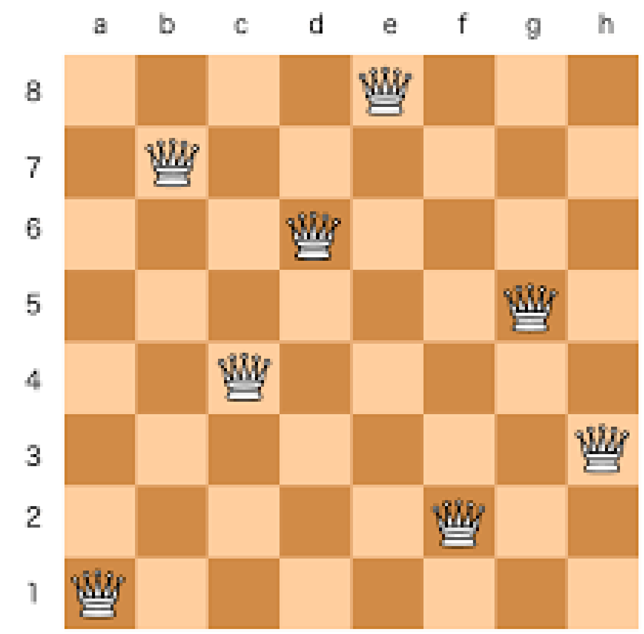
[RN]

# Array Applications: Programming Homework

- Merge two sorted arrays
  - In a third array
  - *In situ* (later also check with linked lists)
- For a given data, create a histogram
  - Numbers of students in [0..10), [10, 20), ..., [90, 100].
- Given two arrays of sizes N1 and N2, find a product matrix (P[i][j] = A[i] * B[j]).
  - Can this be done in O(N1 + N2) time?
  - or O(N1 log N2)?
- Given an unsorted array of (positive and negative) integers, the task is to find the smallest positive number missing from the array (in O(N) time?).

[RN]

# Matrix Appn.: 8-Queens Problem (Homework)




Given a chess-board,
can you place 8 queens
in non-attacking positions?
(no two queens in the same row
or same column or same diagonal)

- Does a solution exist for 2x2, 3x3, 4x4?

Image source: leetcode.com

[RN]

# Matrix Appn.: Knight Tour (Homework)

- Start from a corner.
- Visit all 64 squares without visiting a square twice.
- The only moves allowed are valid knight's moves.
- Cannot wrap-around the board.

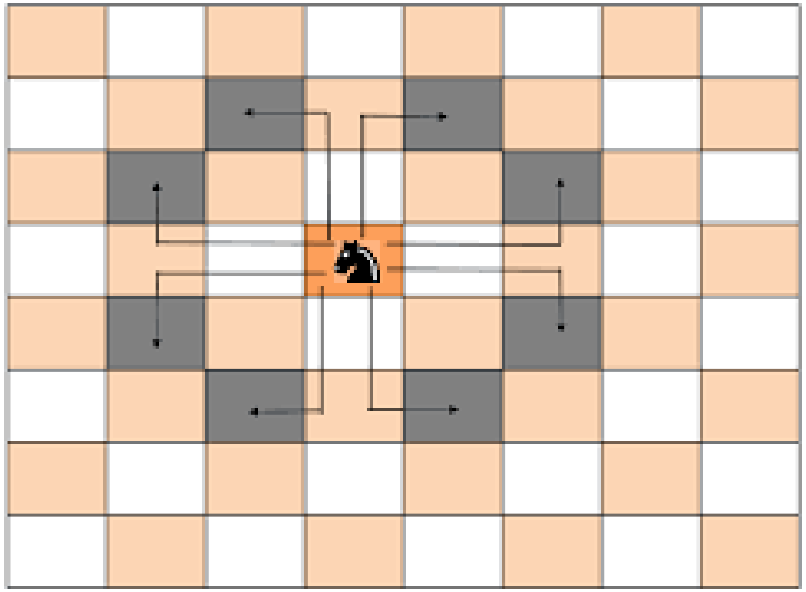


Image source: tutorialhorizon.com

Image source: leetcode.com

[RN]

# Outline for Module M4

- M4 Array
  - M4.1 Array ADT vs. DS (incl. Matrix and Applications)
  - **M4.2 Searching algorithms (linear and binary search, 2D binary search)**
  - M4.3 Sorting algorithms (bubble/insertion/selection sort, quicksort, mergesort/heapsort algos.)
  - M4.4 Comparison-based sorting model (running time lower bound)
  - M4.5 Other sorting models/algorithms (bucket sort algo. and cousins)

# Search

- Linear: O(N)
- Binary: O(log N)
  - $T(N) = T(N/2) + c$

**How about Ternary search?**

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | ... | 40 | 50 | ... | 91 | 95 | 98 | 99 |

mid1 mid2

```
int bsearch(int a[], int N, int val) {
        int low = 0, high = N - 1;

        while (low <= high) {
                int mid = (low + high) / 2;
                if (a[mid] == val) return 1;
                if (a[mid] > val) high = mid - 1;
                else low = mid + 1;
        }
        return 0;
}
```

[RN]

# From 1D to 2D search – does binary search work in 2D?

- If a matrix is sorted left-to-right and top-to-bottom, can we apply binary search?

[RN]

# Search in a Sorted Matrix[M][N]

| | | | | |
|---|---|---|---|---|
| 3 | 5 | 9 | 20 | 39 |
| 4 | 6 | 11 | 21 | 40 |
| 7 | 10 | 12 | 23 | 45 |
| 8 | 13 | 22 | 27 | 46 |
| 19 | 29 | 41 | 43 | 49 |
| 24 | 30 | **44** | 50 | 52 |
| 25 | 31 | 47 | 51 | 55 |
| 28 | 33 | 48 | 53 | 61 |
| 32 | 42 | 54 | 56 | 66 |
| 35 | 57 | 60 | 62 | 69 |

Focus on **44**.
Check where all values < 44 appear.
Check where all values > 44 appear.

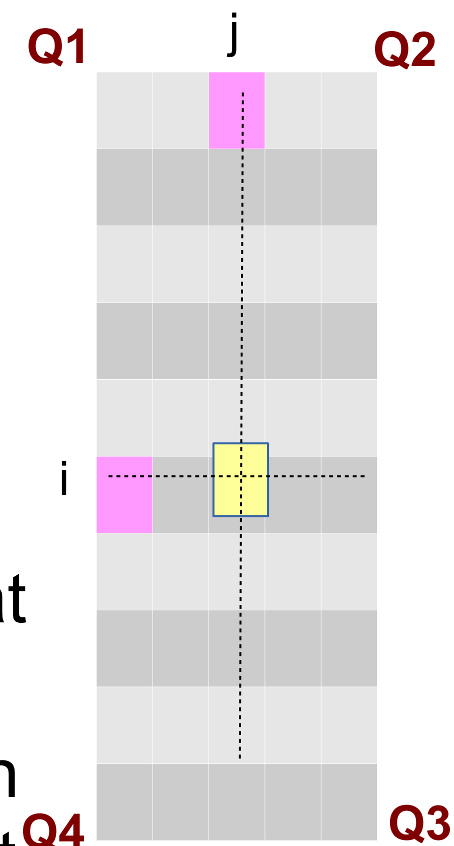
**Classwork**: Devise a method to search for an element in this matrix.

For now, let's assume that all values are unique.

[RN]

# Search in a Sorted Matrix[M][N]

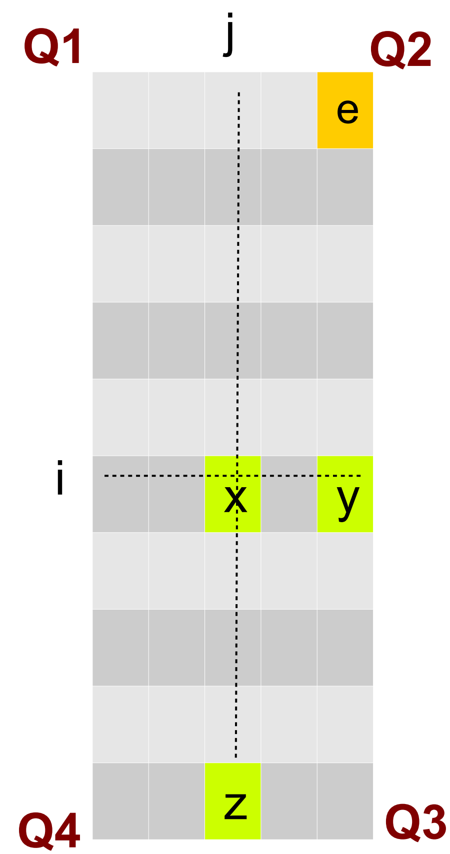
- Approach 1: Divide and Conquer
  - < i, 0 and < 0, j → Q1
  - < i, 0 and > 0, j → Q1, Q2
  - > i, 0 and < 0, j → Q1, Q4
  - > i, 0 and > 0, j → Q1, Q2, Q3, Q4
  - $T(M, N) = 4T(M/2, N/2) + c = O(\min(M, N)^2 \log \max(M,N))$
  - This complexity is almost same as that for the linear search.
  - To improve complexity, we need to reduce at least one quadrant.
  - Note: A number in Q1 is always smaller than [i,j]. But a number smaller than [i, j] need not be in Q1.




[RN]

# Search in a Sorted Matrix[M][N]

- Approach 2: Divide and Conquer
  - Use the corner points of Q1, Q2, Q3, Q4 to decide the quadrant.
  - > y and > z → Q3
  - Else → Q1, Q2, Q4
  - $T(M, N) = 3T(M/2, N/2) + c = O(\min(M, N)^{1.585} \log \max(M,N))$
- Approach 3: Elimination
  - Consider e: [0, N-1].
  - If key == e, found the element
  - If key < e, eliminate that column
  - If key > e, eliminate that row
  - $O(M + N)$
  - What other corner points I can start with?




[RN]

# Surprise Quiz

- What is *Triskaidekaphobia*?
- What is *Paraskevidekatriaphobia*?


Stall numbers at Santa Anita Park progress from 12 to 12A to 14.


Numbers in a lift

Source: wikipedia

[RN]

# Outline for Module M4

- M4 Array
  - M4.1 Array ADT vs. DS (incl. Matrix and Applications)
  - M4.2 Searching algorithms (linear and binary search, 2D binary search)
  - **M4.3 Sorting algorithms (bubble/insertion/selection sort, quicksort, mergesort/heapsort algos.)**
  - M4.4 Comparison-based sorting model (running time lower bound)
  - M4.5 Other sorting models/algorithms (bucket sort algo. and cousins)

# Sorting

- A fundamental operation
- Elements need to be stored in increasing order.
  - Some methods would work with duplicates.
  - Algorithms that maintain relative order of duplicates from input to output are called stable.
- Comparison-based methods
  - Insertion, (Shell), Selection, Quick, Merge, Heap
- Other methods
  - Radix, Bucket, Counting

[RN]

# Sorting Algorithms at a Glance

| Algorithm | Worst case complexity | Average case complexity |
|---|---|---|
| Bubble | $O(n^2)$ | $O(n^2)$ |
| Insertion | $O(n^2)$ | $O(n^2)$ |
| Shell | $O(n^2)$ | Depends on increment sequence |
| Selection | $O(n^2)$ | $O(n^2)$ |
| Heap | $O(n \log n)$ | $O(n \log n)$ |
| Quick | $O(n^2)$ | $O(n \log n)$ depending on partitioning |
| Merge | $O(n \log n)$ | $O(n \log n)$ |
| Bucket | $O(n\ \alpha \log \alpha)$ | Depends on $\alpha$ |

[RN]

# Bubble Sort

- Compare **adjacent** values and swap, if required.
- How many times do we need to do it?
- What is the invariant?
  - After $i^{th}$ iteration, i largest numbers are at their final places.
  - An element may move *away* from its final position in the intermediate stages (e.g., check the 2 element of a reverse-sorted array).
- **Best** case: Sorted sequence
- **Worst** case: Reverse sorted (n-1 + n-2 + ... + 1 + 0)
- **Homework**: Write the code.

[RN]

# Bubble Sort

```
for (ii = 0; ii < N; ++ii)
    for (jj = 0; jj < N - 1; ++jj)
        if (arr[jj] > arr[jj + 1]) swap(jj, jj + 1);
```

Not using ii

```
for (ii = 0; ii < N - 1; ++ii)
    for (jj = 0; jj < N – ii - 1; ++jj)
        if (arr[jj] > arr[jj + 1]) swap(jj, jj + 1);
```

$O(n^2)$

- **Best** case: Sorted sequence
- **Worst** case: Reverse sorted (n-1 + n-2 + ... + 1 + 0)
- What do we measure?
  - Number of comparisons
  - Number of swaps (bounded by comparisons)
- Number of comparisons remains the same!

[RN]

# Insertion Sort

- Consider $i^{th}$ element and insert it at its place w.r.t. the first i elements.
  - Resembles insertion of a playing card.
- Invariant: Keep the first i elements sorted.
- **Note:** Insertion is in a sorted array.
- Complexity: O(n log n)?
  - Yes, binary search is O(log n).
  - But are we doing more work?
  - Best case, Worst case?
- **Homework**: Write the code.

[RN]

# Insertion Sort

```
for (ii = 1 ; ii < N; ++ii) {
        int key = arr[ii];
        int jj = ii - 1;

        while (jj >= 0 && key < arr[jj]) {
                arr[jj + 1] = arr[jj];
                --jj;
        }
        arr[jj + 1] = key;
}
```

$i^{th}$ element

Shift elements
0 + 1 + 2 + ... n-1

At its place

- **Best** case: Sorted: while loop is O(1)
- **Worst** case: Reverse sorted: $O(n^2)$

[RN]

# Selection Sort

- Approach: Choose the minimum element, and push it to its final place.
- What is the invariant?
  - First i elements are at their final places after i iterations.
- **Homework:**

```
for (ii = 0 ; ii < N - 1; ++ii) {
    int iimin = ii;

    for (jj = ii + 1; jj < N; ++jj)
        if (arr[jj] < arr[iimin])
            iimin = jj;
    swap(iimin, ii);
}
```

Find min.

[RN]

# Heapsort

Given N elements,
build a heap and
then perform N deleteMax,
store each element into an array.

N storage

O(N) time

O(N log N) time

O(N) time and N space

O(N log N) time and 2N space

```
for (int ii = 0; ii < nelements; ++ii) {
        h.hide_back(h.deleteMax());
}
h.printArray(nelements);
```

**Source:** heap-sort.cpp

**Can we avoid the second array?**

[RN]

# Quicksort

- Approach:
  - Choose an arbitrary element (called pivot).
  - Place the pivot at its final place.
  - Make sure all the elements smaller than the pivot are to the left of it, and ... (called partitioning)
  - Divide-and-conquer.

```
void quick(int start, int end) {
        if (start < end) {
                int iipivot = partition(start, end);
                quick(start, iipivot - 1);
                quick(iipivot + 1, end);
        }
}
```

Crucially decides the complexity.

[RN]

# Merge Sort

- Divide-and-Conquer
  - Divide the array into two halves
  - Sort each array separately
  - Merge the two sorted sequences
- Worst case complexity: O(n log n)
  - Not efficient in practice due to array copying.
- **Homework:**

```
void mergeSort(int start, int end) {
        if (start < end) {
                int mid = (start + end) / 2;
                mergeSort(start, mid);
                mergeSort(mid + 1, end);
                merge(start, mid, end);
        }
}
```

[RN]

# Outline for Module M4

- M4 Array
  - M4.1 Array ADT vs. DS (incl. Matrix and Applications)
  - M4.2 Searching algorithms (linear and binary search, 2D binary search)
  - M4.3 Sorting algorithms (bubble/insertion/selection sort, quicksort, mergesort/heapsort algos.)
  - **M4.4 Comparison-based sorting model (running time lower bound)**
  - M4.5 Other sorting models/algorithms (bucket sort algo. and cousins)

# The 20-Questions-game

How many YES/NO qns are reqd. to identify an object from a set of M objects?

(Information-theoretic lower bound)

(lower bounds are typically harder to prove than upper bounds on running time of a class of algorithms, but info. theory offers techniques to prove lower bounds)

# Abstract representation of any comparison-based sorting algorithm

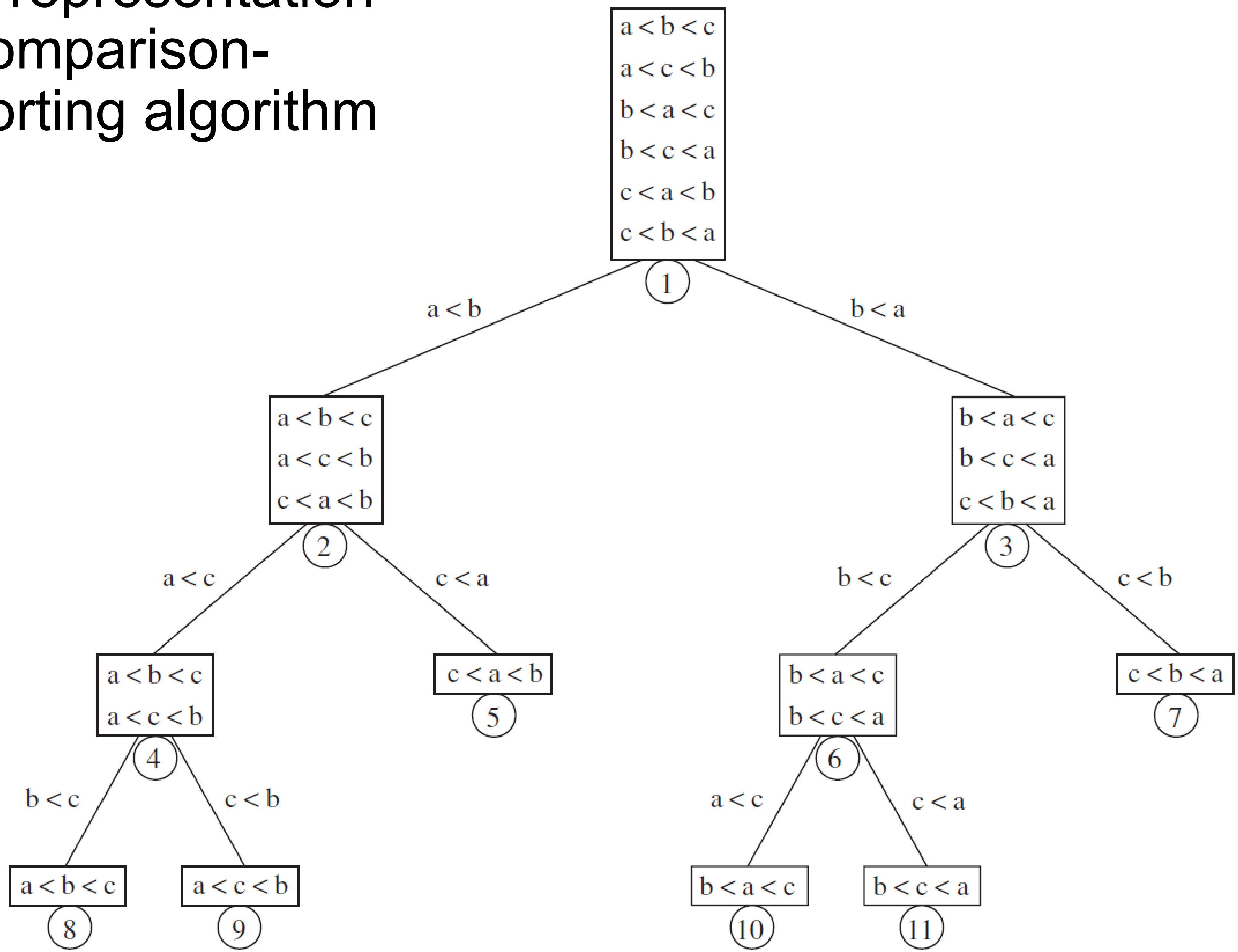



**Figure 7.20** A decision tree for three-element sort

[from WeissBook]

# The class of comparison-based sorting algorithms – running time lower bound

- Array consists of n distinct elements.
- Number of orderings/permutations = n!
- A sorting algorithm must distinguish between these permutations.
- The number of yes/no qns. necessary to distinguish n! permutations is log(n!).
  - Also called information theoretic lower bound
- Given: $N! >= (n/2)^{n/2}$
- $\log(N!) >= n/2 \log(n/2)$ which is $\Omega(n \log n)$
- Comparison-based sort needs 1 qn. per comparison (two numbers). Hence it must require at least n log n time.
  - For each comparison-based sorting algorithm, there exists an input for which it would take n log n comparisons.
  - Heapsort, mergesort are theoretically asymptotically optimal (subject to constants)

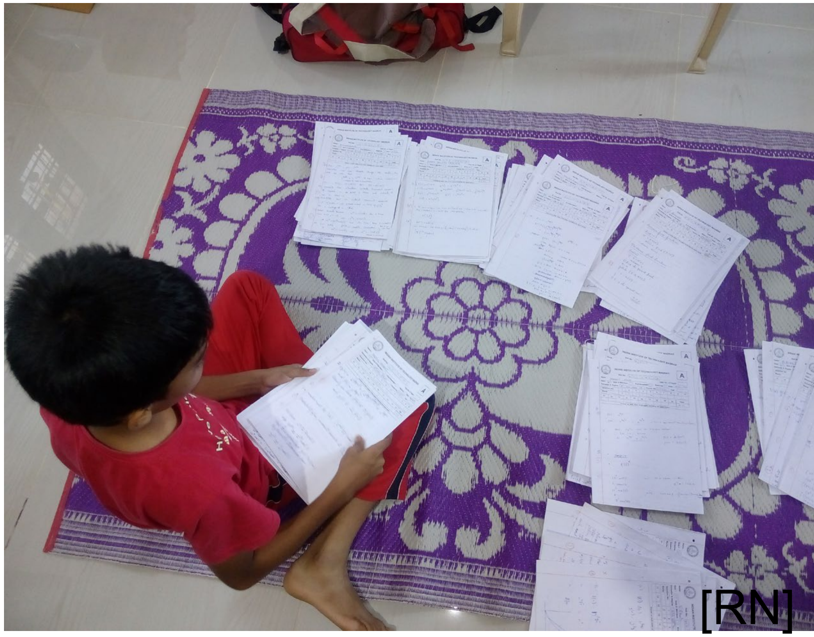
[RN]

# Outline for Module M4

- M4 Array
  - M4.1 Array ADT vs. DS (incl. Matrix and Applications)
  - M4.2 Searching algorithms (linear and binary search, 2D binary search)
  - M4.3 Sorting algorithms (bubble/insertion/selection sort, quicksort, mergesort/heapsort algos.)
  - M4.4 Comparison-based sorting model (running time lower bound)
  - **M4.5 Other sorting models/algorithms (bucket sort algo. and cousins)**

# Bucket Sort

- Hash / index each element into a bucket.
- Sort each bucket.
  - use other sorting algorithms such as insertion sort.
- Output buckets in increasing order.
- Special case when number of buckets >= maximum element value.
- Unsuitable for arbitrary types.



[RN]

# Counting Sort

- Bucketize elements.
- Find count of elements in each bucket.
- Perform prefix sum.
- Copy elements from buckets to original array.

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Original array | 4 | 1 | 4 | 9 | 11 | 7 | 5 | 1 | 3 | 4 |
| Buckets | 1, 1 | | 3 | 4, 4, 4 | 5 | | 7 | | 9 | 11 |
| Bucket sizes | 2 | 0 | 1 | 3 | 1 | 0 | 1 | 0 | 1 0 | 1 |
| Starting index | 0 | 2 | 2 | 3 | 6 | 7 | 7 | 8 | 8 9 | 9 |
| Output array | 1 | 1 | 3 | 4 | 4 | 4 | 7 | 8 | 9 | 11 |

[RN]

# Radix Sort

**O(P * (N + B))**
P = passes
N = elements
B = buckets

- Generalization of bucket sort.
- Radix sort sorts using different digits.
- At every step, elements are moved to buckets based on their $i^{th}$ digits, starting from the least significant digit.
- **Homework 1**: 33, 453, 124, 225, 1023, 432, 2232
- **Homework 2:** bat, gym, cat, rat, dim, cub

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 64 | 8 | 216 | 512 | 27 | 729 | 0 | 1 | 343 | 125 |
| 0 | 1 | 512 | 343 | 64 | 125 | 216 | 27 | 8 | 729 |
| 00, 01, 08 | 512, 216 | 125, 27, 729 | | 343 | | 64 | | | |
| 000, 001, 008, 027, 064 | 125 | 216 | 343 | | 512 | | 729 | | |

[RN]

# Merge vs. Radix Sort: An exercise for the road!

You are given a set of $m$ strings, with each string being of maximum length $k$. These strings are words from the English alphabet $\Sigma$ with $|\Sigma| = 26$.

- What is the running time of sorting these strings using merge sort?
  (Beware: Each comparison is not O(1).)

- What is the running time of sorting these strings using radix sort?
  (Note: How many buckets are being used here? How many rounds/passes?)

- Express above running times in terms of $k$, $m$ and $|\Sigma|$, so that you can answer the following questions:
  - Which of the above two algorithms is better for English alphabet?
  - Which of the two algorithms is better if your language has a very large alphabet, i.e., will your answer change if $|\Sigma|$ is not a constant?