Arrays

Rupesh Nasre.
rupesh@cse.iitm.ac.in

July 2022

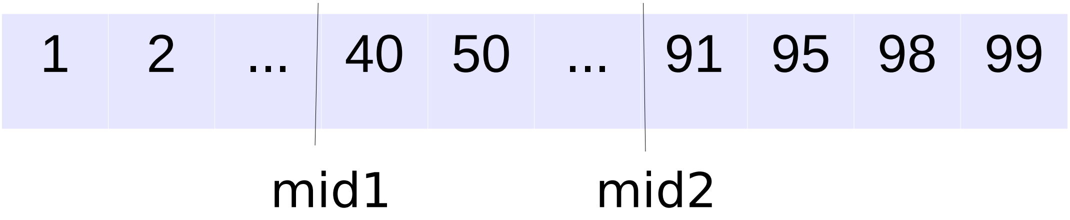
Properties

- 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 based on type
- Storage space:
  - Fixed for arrays
  - Dynamically allocatable but fixed on stack and heap
  - Variable for vectors (internally, reallocation and copying)

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;
}
```

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

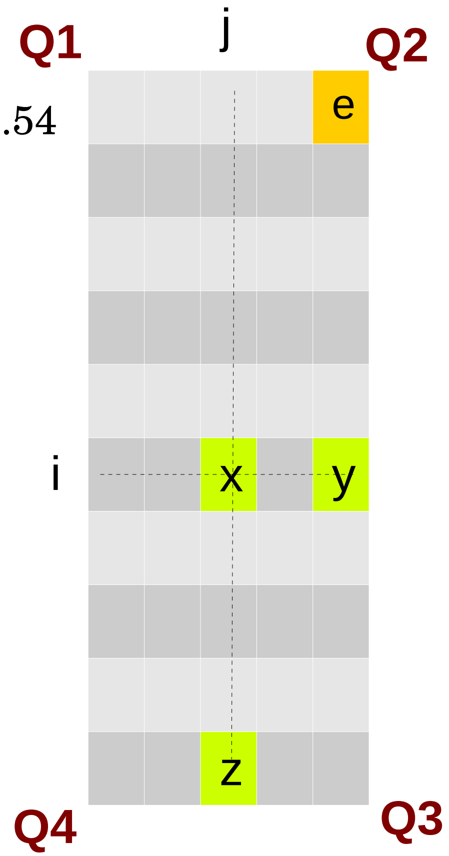
Focus on **44**.
Check where all values < 44 appear.
Check where all values > 44 appear.

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

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

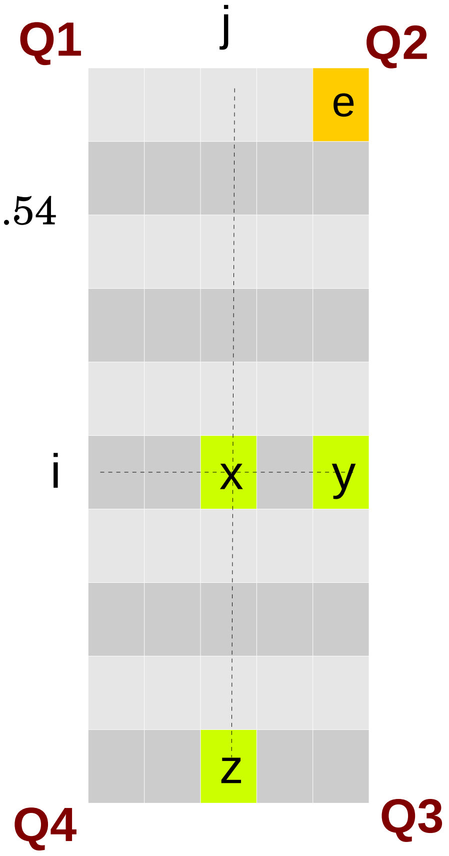
Search in a Sorted Matrix[M][N]

- 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
  - $T(M, N) = 3T(M/2, N/2) + c = O(\min(M, N))^{1.54}$
- 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?



Search in a Sorted Matrix[M][N]

- Approach 4: Divide and Conquer
  - Reduce at least one quadrant
  - > x → Q2, Q3, Q4 (eliminate Q1)
  - < x → Q1, Q2, Q4 (eliminate Q3)
  - == x → *eureka*
  - $T(M, N) = 3T(M/2, N/2) + c = O(\min(M, N))^{1.54}$



Problem: Negative then Positive.

```
int arr[N] = {53, 33, 0, -4, 43, 9, 58, 22, -59, 4, -7, 74, 55, -9, 23, 8, 2, -3};
```

```
-3 -9 -7 -4 -59 9 58 22 43 4 0 74 55 33 23 8 2 53
```

Given a list of numbers (boys+girls / CS+nonCS / Mahanadi+Ganga / Negative+Positive), move all negatives to the left (in any order).

Problem: Merge sorted arrays

```
int A[] = {-3, 0, 43, 58, 64, 79, 93};
int B[] = {-5, 4, 59, 70, 74, 75, 81, 88, 92};
int NA = sizeof(A) / sizeof(A[0]);
int NB = sizeof(B) / sizeof(B[0]);

int C[NA + NB]; // variable length array, allowed from ANSI C99 standard.
```

```
-5 -3 0 4 43 58 59 64 70 74 75 79 81 88 92 93
```

```
C[indexC] = A[indexA];
indexA++;
indexC++;
```

Extend the program to perform in-situ merge.
Array A has two sorted sequences.

C = A merge B, with A and B are sorted.
C is also sorted.

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, Bubble, Selection, Shell, Quick, Merge
- Other methods
  - Radix, Bucket, Counting

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 α

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 2nd element of a reverse-sorted array).
- **Best** case: Sorted sequence
- **Worst** case: Reverse sorted (n-1 + n-2 + ... + 1 + 0)
- **Classwork**: Write the code.

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!

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?
- **Classwork**: Write the code.

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;
}
```

ith element

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

At its place

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

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.
- **Classwork:**

```
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.

Heapsort

Given N elements, N storage

build a heap and O(N) time

then perform N deleteMax, O(N log N) time

store each element into an array. 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);
```

Can we avoid the second array?

Source: heap-sort.cpp

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.
- Best case, worst case?
- **Classwork:** Write the code.

6	2	4	9	11	7	8	1	3	5

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.
- **Classwork:** Write the code (reuse the merge function already written).

6	2	4	9	11	7	8	1	3	5

Comparison-based Sorts

- Array consists of n distinct elements.
- Number of permutations = n!
- A sorting algorithm must distinguish between these permutations.
- The number of yes/no bits 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 Ω (n log n)
- Comparison-based sort needs 1 bit 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)

Bucket Sort

- Hash / index each element into a bucket, based on its value (specific hash function).
- 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.

6	2	4	9	11	7	8	1	3	5

Counting Sort

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

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

Radix Sort

- Generalization of bucket sort.
- Radix sort sorts using different digits.
- At every step, elements are moved to buckets based on their ith digits, starting from the least significant digit.
- **Classwork**: 33, 453, 124, 225, 1023, 432, 2232

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		

Practice Problem

Knight’s tour

- Start from a corner.
- Visit all 64 squares without visiting a square twice.
- The only moves allowed are 2.5 places.
- Cannot wrap-around the board.

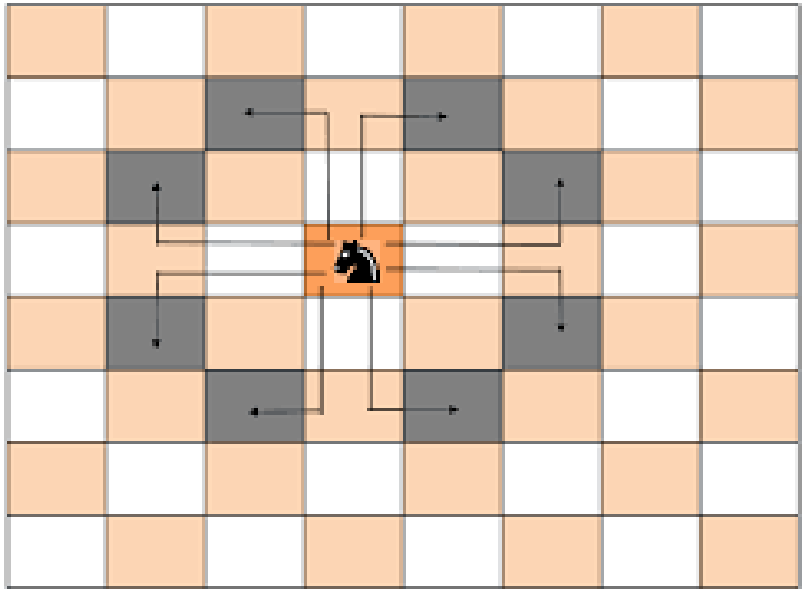


Image source: tutorialhorizon.com

8-Queens Problem

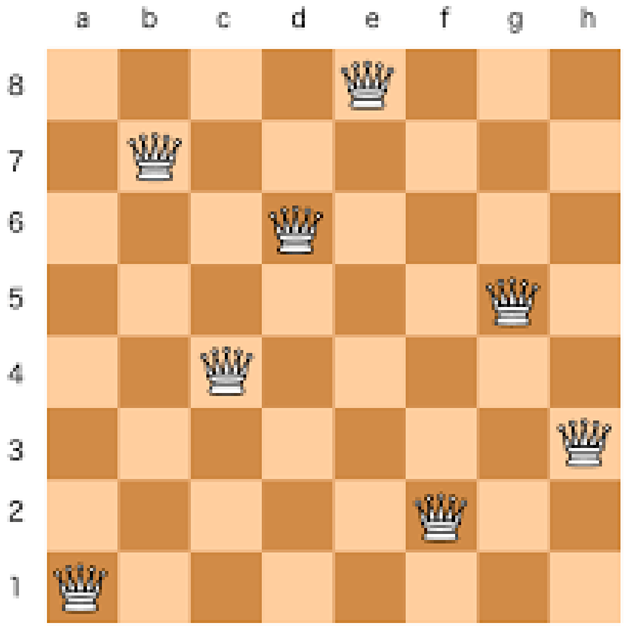
Given a chess-board,

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?
- Have you seen similar constraints somewhere?

Image source: leetcode.com