CS202 – Data Structures

LECTURE-24

Sorting – II

More on Sorting algorithms

Dr. Maryam Abdul Ghafoor
Assistant Professor
Department of Computer Science, SBASSE

Agenda

- Sorting Algorithms
  - Quick Sort
  - External Sort
  - Linear Sorting Algorithms

Quicksort

Quick Sort

- Choose a "pivot" element
- Group your elements into three groups:
  - Less than pivot
  - Equal to pivot
  - Greater than pivot
- Recursively sort (quick sort) the less than and greater than groups
- Concatenate the three sorted groups back together again

Algorithm

1. Start with list I of n items
2. Choose a pivot item v from I
3. Partition I into 2 unsorted lists I_1 and I_2 around the pivot v
   - I_1: All keys smaller than v
   - I_2: All keys larger than v
   - Items equal to v can go in either list and pivot is not part of I_1 or I_2
4. Sort I_1 and I_2 recursively, yielding sorted lists S_1 and S_2
5. Concatenate S_1, v, S_2 yielding the sorted list S

Base case: list of size 0 or 1 (already sorted!)

Quicksort Example

5	3	9	4	8	2	1	6

Suppose we pick the first element (i.e., 5) as the pivot

Another Example

0	1	3	4	5	7	9

Suppose we pick the first element (i.e., 0) as the pivot

What will be the resulting complexity?

Given an array of integers, which number/integer should be selected as a pivot?

Nobody has responded yet.

Hang tight! Responses are coming in.

What are some ways to pick the pivot?

- Always pick the first element
- Randomly pick an element from the list (Randomized QS)
- Randomly pick 3 elements and then choose the median of these 3 elements as the pivot (median-of-3 strategy)
- Note: With 2 and 3, the expected running time is in Θ(n logn)

What is the ideal pivot?

- Ideal pivot: median

How can we efficiently find the Median?

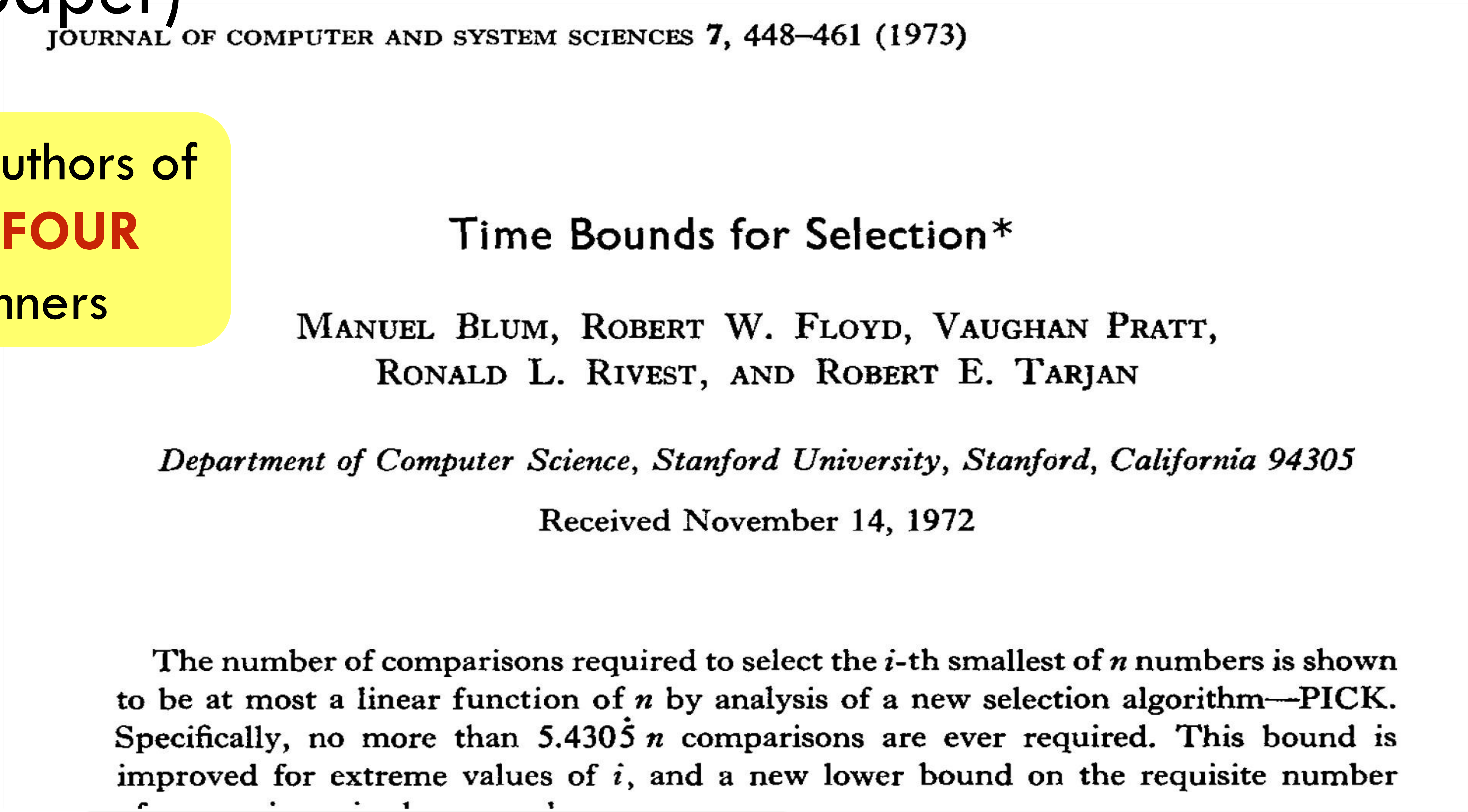
Pivot Selection

How can we efficiently find the Median?

How can we efficiently find the median?

We can find the median in $O(n)$ time using BFPRT (called PICK in original paper)

Historical note: The authors of this paper include **FOUR** Turing Award winners



JOURNAL OF COMPUTER AND SYSTEM SCIENCES 7, 448–461 (1973)

Time Bounds for Selection*

MANUEL BLUM, ROBERT W. FLOYD, VAUGHAN PRATT, RONALD L. RIVEST, AND ROBERT E. TARJAN

Department of Computer Science, Stanford University, Stanford, California 94305

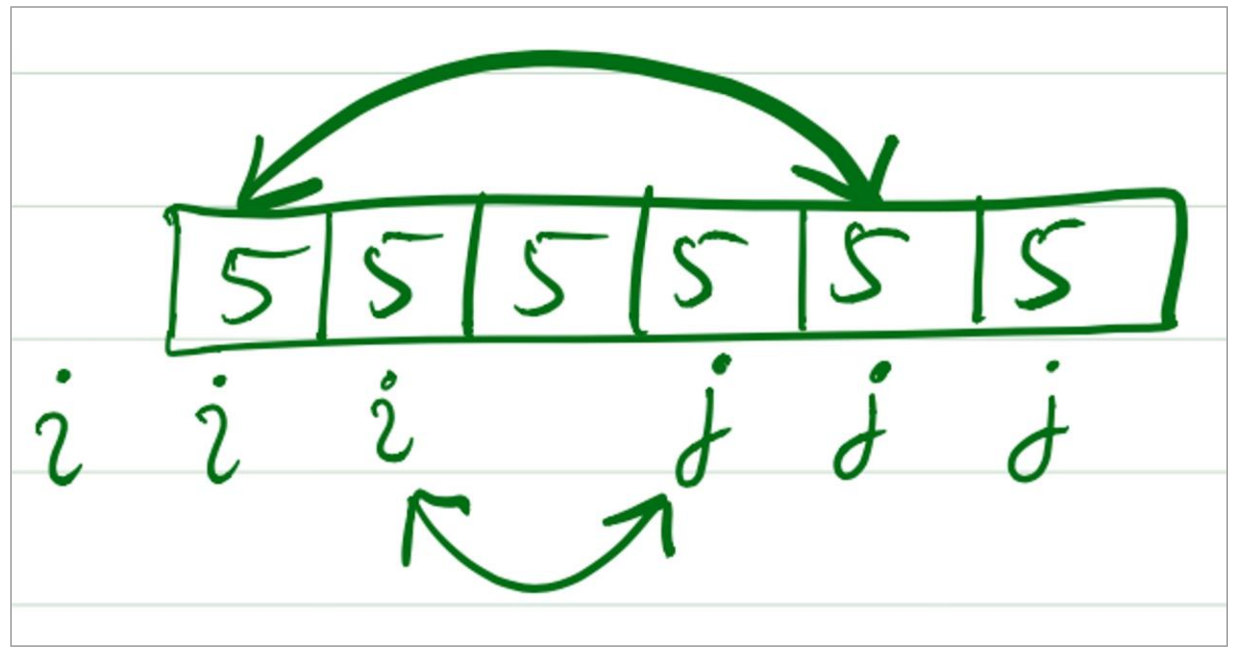
Received November 14, 1972

The number of comparisons required to select the i-th smallest of n numbers is shown to be at most a linear function of n by analysis of a new selection algorithm—PICK. Specifically, no more than $5.430\dot{5}\, n$ comparisons are ever required. This bound is improved for extreme values of i, and a new lower bound on the requisite number

Quicksort if input is …

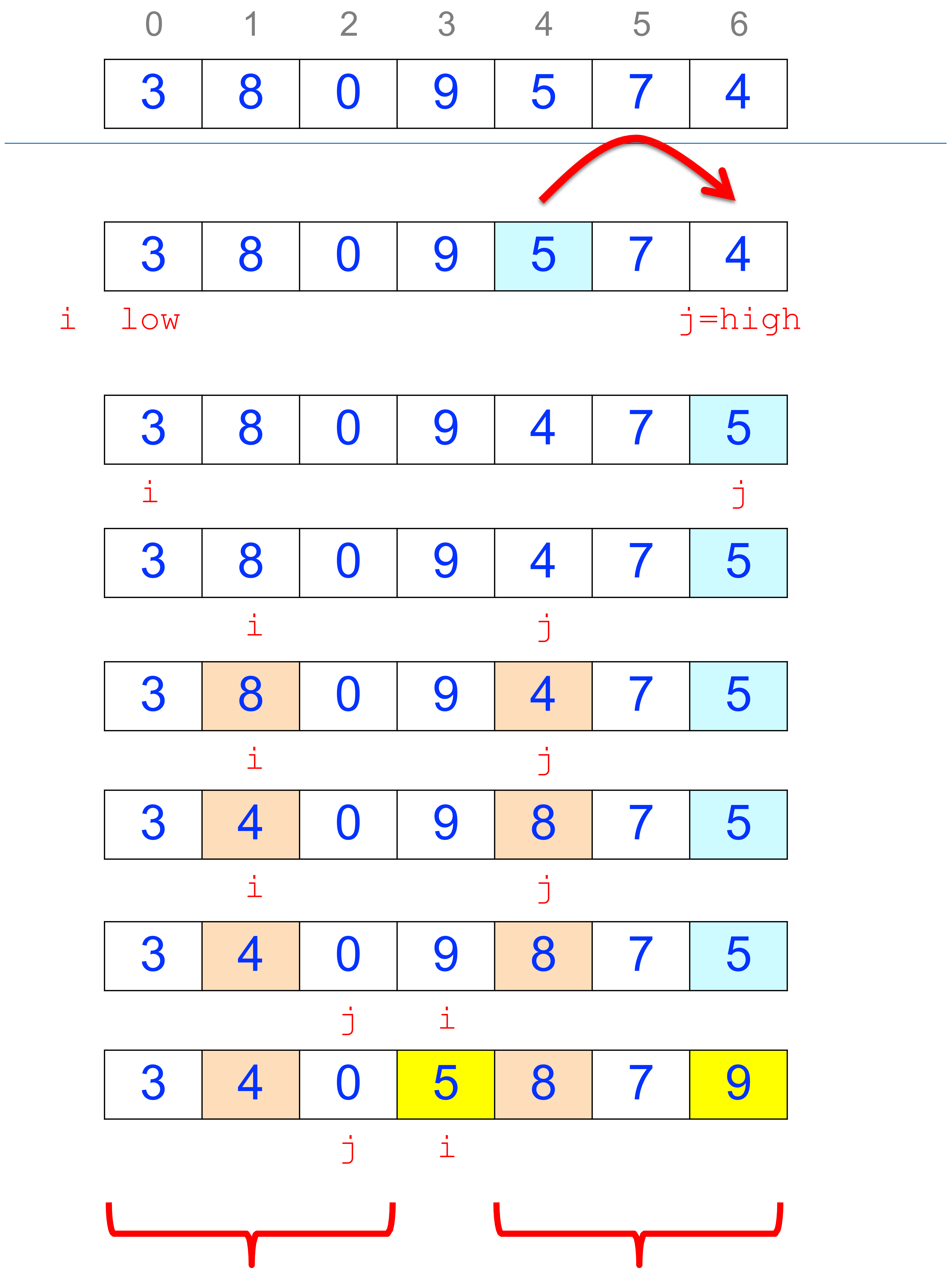
- If we pick the pivot randomly
  - Array: $\Theta(1)$ to read the value
  - Linked List: $\Theta(n)$ to read the random pivot

Invariants & Duplicate Values



- All items left of i are less than the pivot
- All items right of i are greater than the pivot
- Handling keys equal to the pivot with arrays
  - The solution is to make sure each index, i and j, stops whenever it reaches a key equal to the pivot and participates in a swap
  - If all items have the same key, half go into $I1$ and half into $I2$, well- balanced recursion tree, $O(nlogn)$ time

```
void quicksort(int a[], int low, int high) {

  // If there are fewer than two items, do nothing.
  if (low < high) {

    // Generate a random number in between low .. high
    srand(time(NULL));
    int pivotIndex = low + rand() % (high - low);
    int pivot = a[pivotIndex];

    // Swap pivot with last item (since this is in-place)
    a[pivotIndex] = a[high];
    a[high] = pivot;

    int i = low - 1;
    int j = high;
    int tmp; // temporary variable for swapping

    do {
      do { i++; } while (a[i] < pivot);
      do { j--; } while ((a[j] > pivot) && (j > low));
      if (i < j) {
        // swap a[i] and a[j]
        tmp = a[i];
        a[i] = a[j];
        a[j] = tmp;
      }
    } while (i < j);

    a[high] = a[i];
    a[i] = pivot; // Put pivot in the middle where it belongs

    cout<<"privot = "<< pivot << ", (low,high) = (" << low << "," << high << "): ";
    printArray(a, array_size);

    quicksort(a, low, i - 1); // Recursively sort left list
    quicksort(a, i + 1, high); // Recursively sort right list
  }
}
```



Quick Sort

Algorithm inPlaceQuickSort(S, a, b):

Input: An array S of distinct elements; integers a and b

Output: Array S with elements originally from indices from a to b, inclusive, sorted in nondecreasing order from indices a to b

```
if a ≥ b then return            {at most one element in subrange}
p ← S[b]            {the pivot}
l ← a            {will scan rightward}
r ← b − 1            {will scan leftward}
while l ≤ r do
   {find an element larger than the pivot}
   while l ≤ r and S[l] ≤ p do
      l ← l + 1
   {find an element smaller than the pivot}
   while r ≥ l and S[r] ≥ p do
      r ← r − 1
   if l < r then
      swap the elements at S[l] and S[r]
{put the pivot into its final place}
swap the elements at S[l] and S[b]
{recursive calls}
inPlaceQuickSort(S, a, l − 1)
inPlaceQuickSort(S, l + 1, b)
{we are done at this point, since the sorted subarrays are already consecutive}
```

Code Fragment 11.6: In-place quick-sort for an input array S.

Quicksort

- Fastest known comparison-based sorting for arrays
  - in the average-case [$\Theta(n\log n)$]
  - Very widely used!
  - … but worst-case is in $\Theta(n^2)$

Quicksort

- Like mergesort, quicksort is also a divide and conquer algorithm, however, they have important differences

	Dividing	Merging
Mergesort	Simple	all the work
Quicksort	all the work	Simple (Concatenation)

Sorting Algorithms

	Worst-case	Best-case	In-place	Stable
Insertion	$O(n^2)$	$O(n)$	✓	✓
Selection	$O(n^2)$	$O(n^2)$	✓	✗
Merge	$O(n \log n)$	$O(n \log n)$	✗	✓
Quick	$O(n^2)$	$O(n \log n)$	✓	✗

Heap Sort

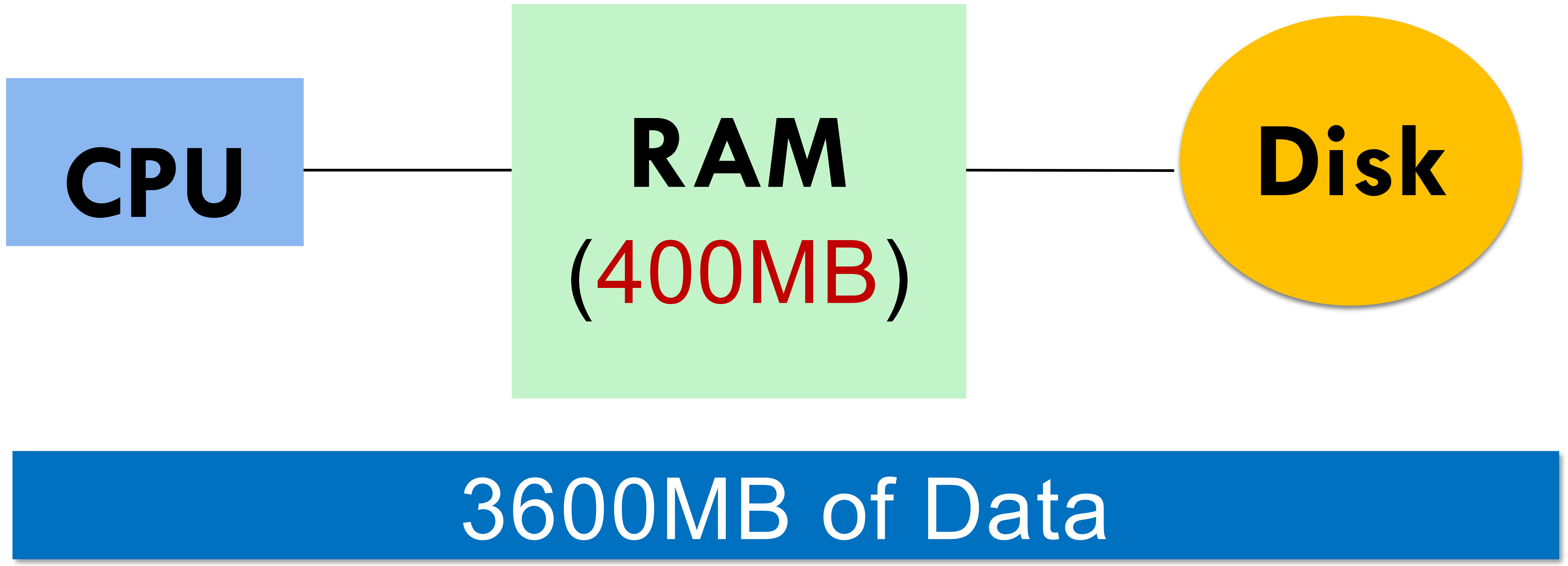
Heap Sort

- Sorting algorithm that uses heaps
- Given an array of unsorted numbers
  - Build a heap (Floyd's method)
  - Call removeMin( ) n times
  - We get a sorted keys!
- Total time = O(n log n)

External Sort

Sorting Big Data

- Suppose we'd like to sort **3600MB** of numbers using **400MB RAM**
- Which sorting algorithm would you use? What would you do to minimize disk accesses?



Sorting Big Data

Suppose we'd like to sort **3600MB** of numbers using **400MB RAM**

How many reads are required?

How many MB can be sorted in one read?

How to merge and store sorted results?

External Sort – High Level Idea

- **Step 1:** Divide the input into smaller chunks that fit into memory.
- **Step 2:** Sort each chunk individually.
- **Step 3:** Merge the sorted chunks into a single sorted output.

External Sorting

- External Sorting is a technique used to sort large amounts of data that cannot fit into memory all at once.

- It involves reading chunks of data into memory, sorting them, and then writing them back to disk.

- Commonly used in sorting large scale datasets stored on the disk.

External Sort

- Read 400MB of data into memory
  - Sort using a conventional method (e.g., quicksort)
  - Write sorted 400MB to a temporary file on disk
  - Repeat until all data is in sorted chunks (3600/400 = 9 chunks in total)

- Read first 40MB of each sorted chunk, and merge using remaining 40MB
  - Read and write to the disk as necessary
  - Single 9-way merge is used

Sorting Massive Data – Summary

- Need sorting algorithms that minimize disk accesses
- Quicksort and Heapsort both jump all over the array, leading to expensive random disk accesses
- Merge sort scans linearly through arrays, leading to (relatively) efficient sequential disk access
  - Merge sort is the basis of massive sorting
  - Merge sort can leverage multiple disks

Lower Bound for Comparison-based Sorting

$O(n^2)$ ⟷ $O(n \log n)$

- insertion sort
- selection sort

- heap sort
- mergesort
- quicksort
- BST/AVL sort

Can we do better? Sort faster than $O(n \log n)$?

Lower Bound for Comparison-based Sorting

- If the computational primitive is comparison (e.g., <, >, ≥, ≤) then $\Omega(n\ logn)$ is the worst-case lower bound on sorting

- This means that no comparison-based sorting algorithm can ever run faster than $\Omega\ (n\ logn)$ on arbitrary input!
  - Proof: see book! (11.3.1)

Question

PollEv

We are given an Array A of n non-negative integers in the range of 0 to k. What is the most suitable approach?

For non-negative integers in a range from 0 to k, which of the following is most suitable method for sorting?

Divide and conquer strategy — 0%

Comparison based in-place algorithm — 0%

Using an array, where each slot stores a count of a number corresponding to its index — 0%

External sort — 0%

None of the above — 0%

Question

We are given an Array A of n non-negative integers in the range of 0 to k. We wish to create another Array C of size k + 1 such that C[x] is equal to the number of times x appears in the original Array A.

Example:

A = 2, 5, 3, 0, 2, 3, 0, 3

then C = 2, 0, 2, 3, 0, 1

Count Sort

Example

3	5	2	3	1	0	2

0	1	2	3	4	5	6
			1			

3	5	2	3	1	0	2

0	1	2	3	4	5	6
			1		1	

3	5	2	3	1	0	2

0	1	2	3	4	5	6
		1	1		1	

3	5	2	3	1	0	2

0	1	2	3	4	5	6
		1	2		1	

3	5	2	3	1	0	2

0	1	2	3	4	5	6
	1	1	2		1	

3	5	2	3	1	0	2

0	1	2	3	4	5	6
1	1	1	2		1	

3	5	2	3	1	0	2

0	1	2	3	4	5	6
1	1	2	2		1	

Sorting

- How can we get the sorted output using C?

- What is the complexity of sorting?

Range Search

- How can you use the output array for range search?

- What is the complexity of this approach?

Counting Sort – Limitation

- Requires a limited range of values
- Not suitable for large or unbounded integers
- Can not sort negative numbers without modification
- Not an in-place sorting algorithm
- Not be used for floating numbers or string
- Inefficient for small data sets – Wastes memory

Bucket Sort

Bucket Sort

- Non-comparative sorting algorithm that distributes elements into multiple "buckets" and sorts them individually
- Sorting numbers with a known range, especially floating-point numbers and uniformly distributed data.
- Takes O(n), if elements are evenly distributed across buckets.

Bucket Sort (a.k.a. Integer Sorting)

- Uses buckets to sort integers in the range 0 to k − 1

- High level Idea
  - Keep an array of q queues (or buckets)
  - Walk through the array, enqueue each key i in queue i
  - Last step: Concatenate queues in order

- Works well when the keys are in a small range(q is in $O(n)$)
  - n is the number of integers to sort

Bucket Sort

- How to sort numbers [0.42, 0.32, 0.23, 0.52, 0.25, 0.47, 0.51]?

- High level Idea
  – Create buckets
  – Each number is placed in the corresponding bucket
  – Sort each bucket
  – Concatenate sorted buckets

Bucket Sort

- How to sort numbers [0.42, 0.32, 0.23, 0.25, 0.47, 0.50]?

Create buckets

Buckets	
0	0.0-0.1
1	0.1-0.2
2	0.2-0.3
3	0.3-0.4
4	0.4-0.5

Insert elements

Buckets	Elements
0	
1	
2	0.23,0.25
3	0.32
4	0.42,0.47,0.5

Bucket Sort

- How to sort numbers [0.42, 0.32, 0.23, 0.25, 0.47, 0.50]?

Sort elements

Buckets	Elements
0	
1	
2	0.23,0.25
3	0.32
4	0.42,0.47,0.5

Concatenate Buckets

0.23,0.25,0.32,0.42,0.47,0.5

This is exactly a hash table!

Bucket Sort – Time Complexity

- Data is uniformly distributed – O(n)

- $O(q + n)$
  - Linear in n but also linear in q
  - time to initialize and concatenate q queues O(q)
  - time to put items in queues O(n)

- O(n^2) when all elements are in a single bucket

Radix Sort

Radix Sort – Time Complexity

- Sorts numbers digit by digit, starting either from the least significant digit (LSD) or from the most significant digit (MSD).

- Data is uniformly distributed – O(n)

- $O(q + n)$
  - Linear in n but also linear in q
  - time to initialize and concatenate q queues O(q)
  - time to put items in queues O(n)

- Worst case time complexity when working with huge numbers. $O(n^2)$

Radix Sort

- Works well with
  - Integer
  - Fixed length string

- Limitations
  - Memory requirement
  - Data doesn’t fit into buckets
  - Sparse data with outliers (when elements have huge digits)

Sorting Algorithms

- https://www.toptal.com/developers/sorting-algorithms
- https://www.cs.usfca.edu/~galles/visualization/ComparisonSort.html
- https://www.youtube.com/watch?v=kPRA0W1kECg

Questions