Sorting algorithms

- Sorting Overview
- Elementary Sorting Algorithms
  - Insertion sort (recap.)
  - Bubble sort
  - Selection sort
- Merge Sort
- Quick Sort

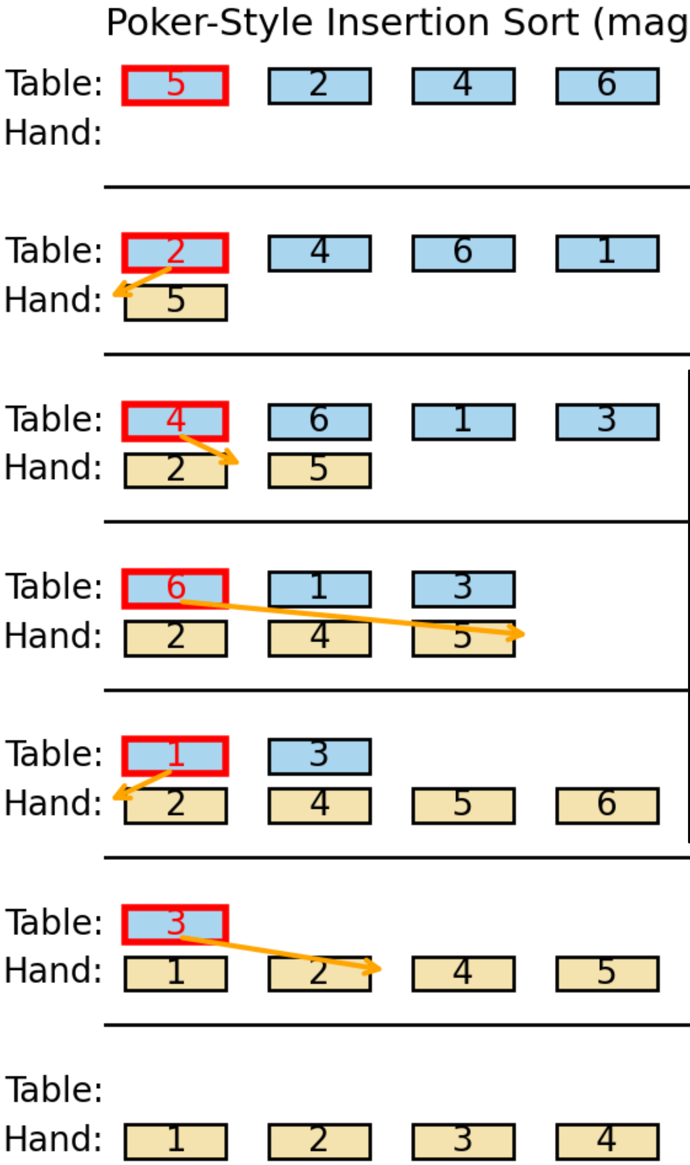
Yanlin Zhang & Wei Wang | DSAA 2043 Spring 2025

The Problem of Sorting

- Input
  - A sequence of n numbers < a1, a2, …, an>
- Output
  - Permutation < a'1, a'2, …, a'n> such that a'1 ≤ a'2 ≤ … ≤ a'n
- Example
  - Input: 8 2 4 9 3 6
  - Output: 2 3 4 6 8 9
- Sorting is a fundamental operation:
  - Searching (Binary Search requires sorted arrays)
  - Data Processing (Efficient indexing in databases)
  - Graph Algorithms (Kruskal’s algorithm for MST)
  - Bioinformatics (Sorting datasets before analysis)

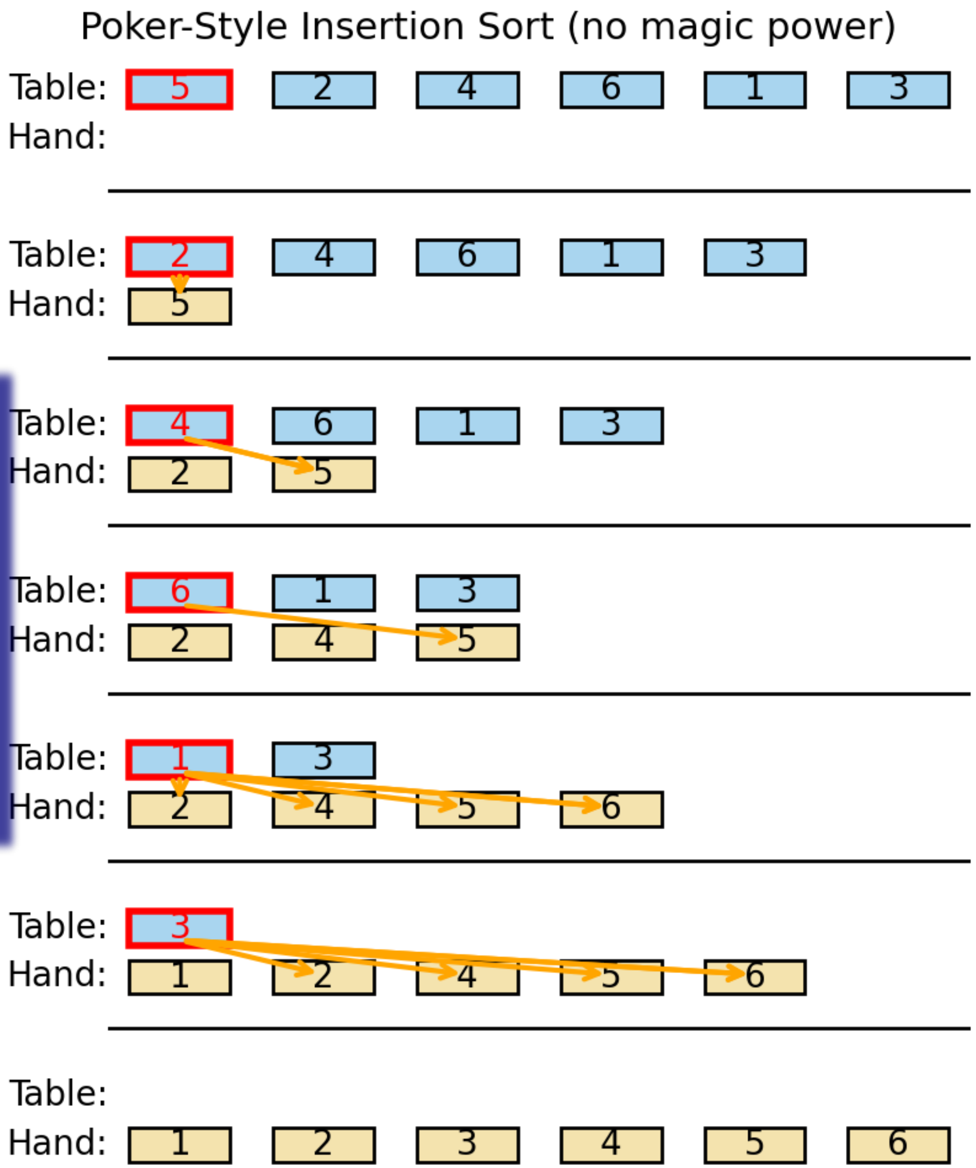
Elementary Sorting Algorithms

Insertion Sort (Recap.)



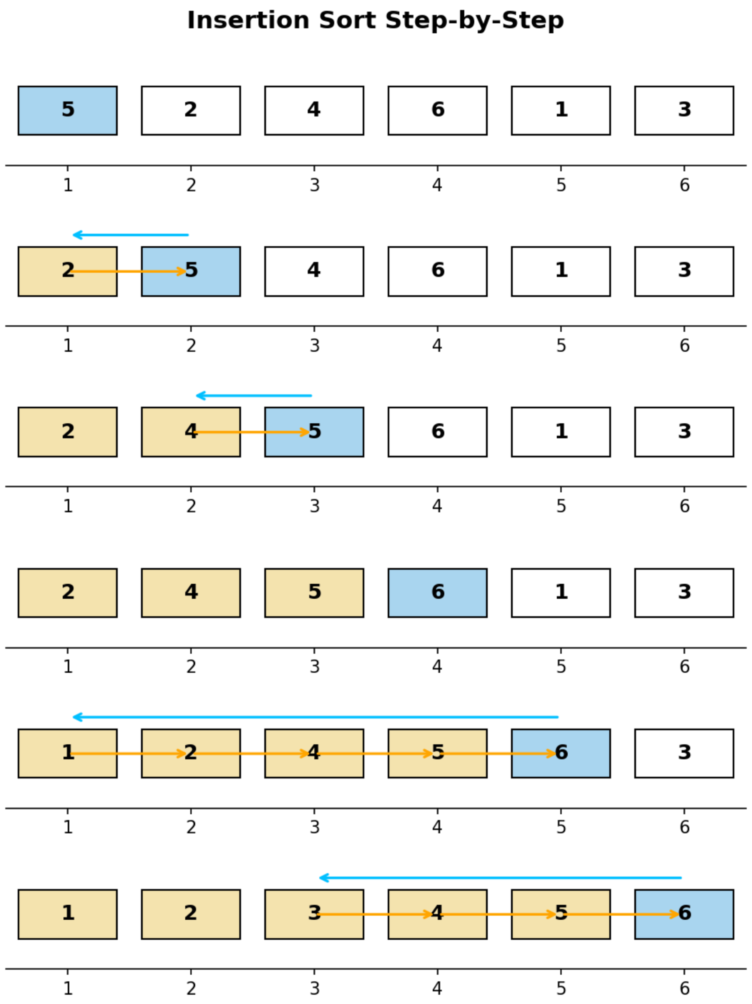
Strategy

- Start "empty handed"
- Insert a card in the right position of the already sorted hand
- Continue until all cards are inserted/sorted



Insertion Sort (Recap.)


```
Algorithm InsertionSort(A)
Input: An array A with n elements
Output: A sorted in non-decreasing order

for i ← 2 to length(A) do
    key ← A[i]  // Current element to be inserted
    j ← i - 1   // Start comparing from the previous element
    while j > 0 and A[j] > key do
        A[j + 1] ← A[j]  // Shift element to the right
        j ← j - 1
    end while
    A[j + 1] ← key  // Place key in the correct position
end for

return A  // Sorted array
```



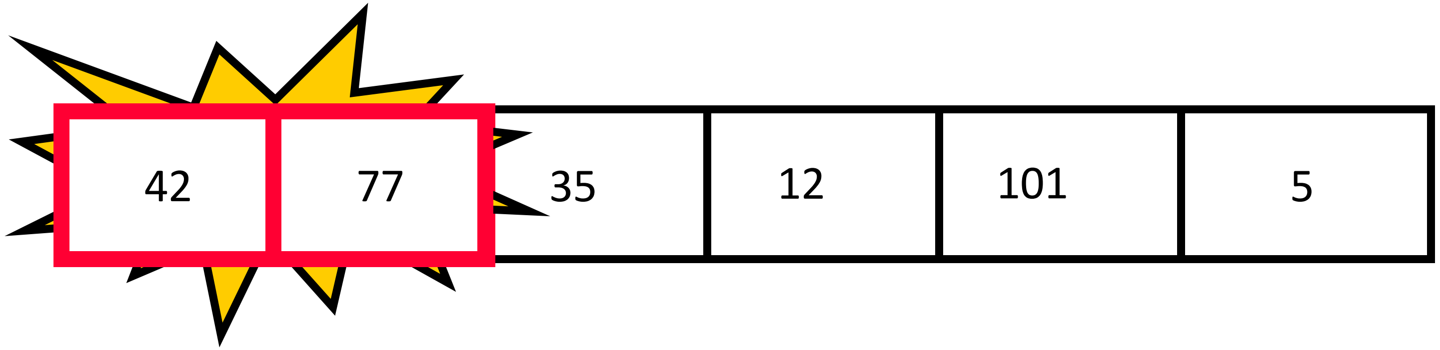
Bubble Sort: "Bubbling Up" the Largest Element

- Traverse a collection of elements
- Move from the front to the end
- “Bubble” the largest value to the end using pair-wise comparisons and swapping

77	42	35	12	101	5

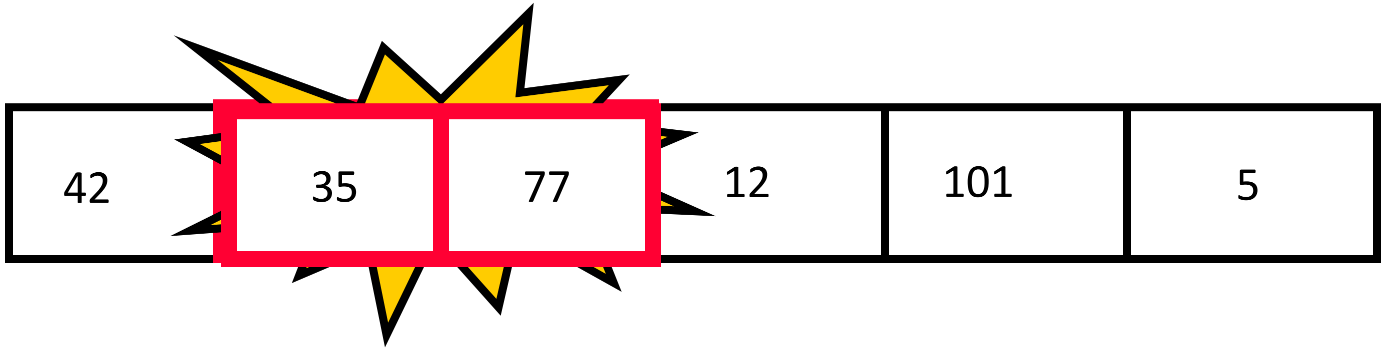
Bubble Sort: "Bubbling Up" the Largest Element

- Traverse a collection of elements
- Move from the front to the end
- “Bubble” the largest value to the end using pair-wise comparisons and swapping



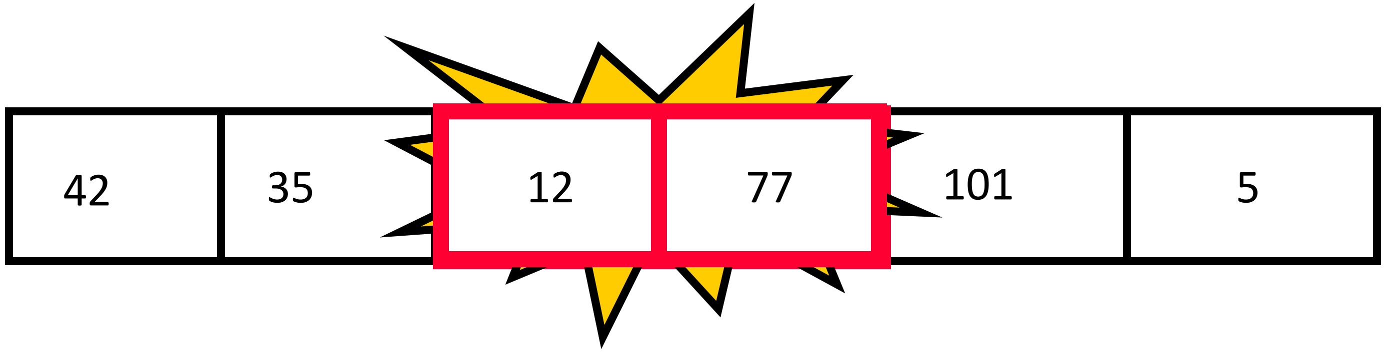
Bubble Sort: "Bubbling Up" the Largest Element

- Traverse a collection of elements
- Move from the front to the end
- “Bubble” the largest value to the end using pair-wise comparisons and swapping



Bubble Sort: "Bubbling Up" the Largest Element

- Traverse a collection of elements
- Move from the front to the end
- “Bubble” the largest value to the end using pair-wise comparisons and swapping



Bubble Sort: "Bubbling Up" the Largest Element

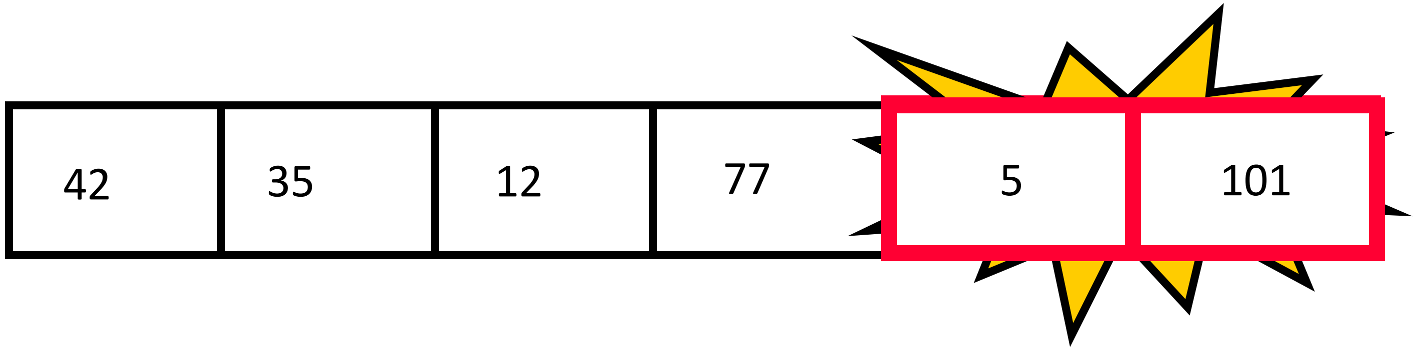
- Traverse a collection of elements
- Move from the front to the end
- "Bubble" the largest value to the end using pair-wise comparisons and swapping

42	35	12	77	101	5

No need to swap

Bubble Sort: "Bubbling Up" the Largest Element

- Traverse a collection of elements
- Move from the front to the end
- “Bubble” the largest value to the end using pair-wise comparisons and swapping



Bubble Sort: "Bubbling Up" the Largest Element

- Traverse a collection of elements
- Move from the front to the end
- "Bubble" the largest value to the end using pair-wise comparisons and swapping

42	35	12	77	5	101

Largest value correctly placed

The "Bubble Up" Algorithm

```
index ← 1
last_compare_at ← n – 1

loop
  exitif(index > last_compare_at)
  if(A[index] > A[index + 1]) then
    Swap(A[index], A[index + 1])
  endif
  index ← index + 1
endloop
```

Items of Interest

- Notice that only the largest value is correctly placed
- All other values are still out of order
- So we need to repeat this process

42	35	12	77	5	101

Largest value correctly placed

Repeat “Bubble Up” How Many Times?

- If we have N elements …
- And if each time we bubble an element, we place it in its correct location …
- Then we repeat the “bubble up” process N – 1 times
- This guarantees we’ll correctly place all N elements

"Bubbling" All the Elements

N - 1

42	35	12	77	5	101
35	12	42	5	77	101
12	35	5	42	77	101
12	5	35	42	77	101
5	12	35	42	77	101

Reducing the Number of Comparisons

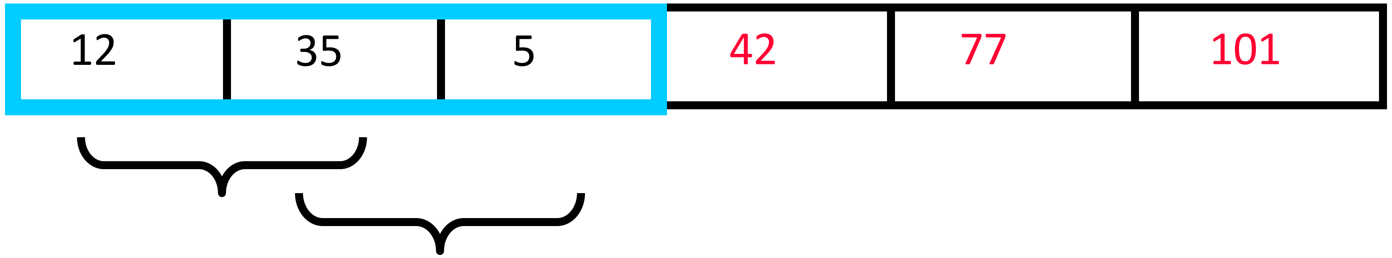
77	42	35	12	101	5
42	35	12	77	5	101
35	12	42	5	77	101
12	35	5	42	77	101
12	5	35	42	77	101

Reducing the Number of Comparisons

- On the Nth "bubble up", we only need to do MAX – N comparisons

For example:

- This is the 4th "bubble up"
- MAX is 6
- Thus we have 2 comparisons to do



Putting It All Together

```
procedure Bubblesort(A)
  to_do, index isoftype Num
  to_do ← N – 1

  loop
    exitif(to_do = 0)
    index ← 1
    loop
      exitif(index >= to_do)
      if(A[index] > A[index + 1]) then
        Swap(A[index], A[index + 1])
      endif
      index ← index + 1
    endloop
    to_do ← to_do - 1
  endloop
endprocedure                        # Bubblesort
```

Inner loop

Outer loop

Already Sorted Collections?

- What if the collection was already sorted?
- What if only a few elements were out of place and after a couple of “bubble ups,” the collection was sorted?
- We want to be able to detect this and “stop early”!

5	12	35	42	77	101

Using a Boolean “Flag”

- We can use a boolean variable to determine if any swapping occurred during the “bubble up”
- If no swapping occurred, then we know that the collection is already sorted!
- This boolean “flag” needs to be reset after each “bubble up”

Pseudo-Code

```
did_swap: Boolean
did_swap ← true

loop
  exitif((to_do = 0) OR NOT(did_swap))
  index ← 1
  did_swap ← false
  loop
    exitif(index >= to_do)
    if(A[index] > A[index + 1]) then
      Swap(A[index], A[index + 1])
      did_swap ← true
    endif
    index ← index + 1
  endloop
  to_do ← to_do - 1
endloop
```

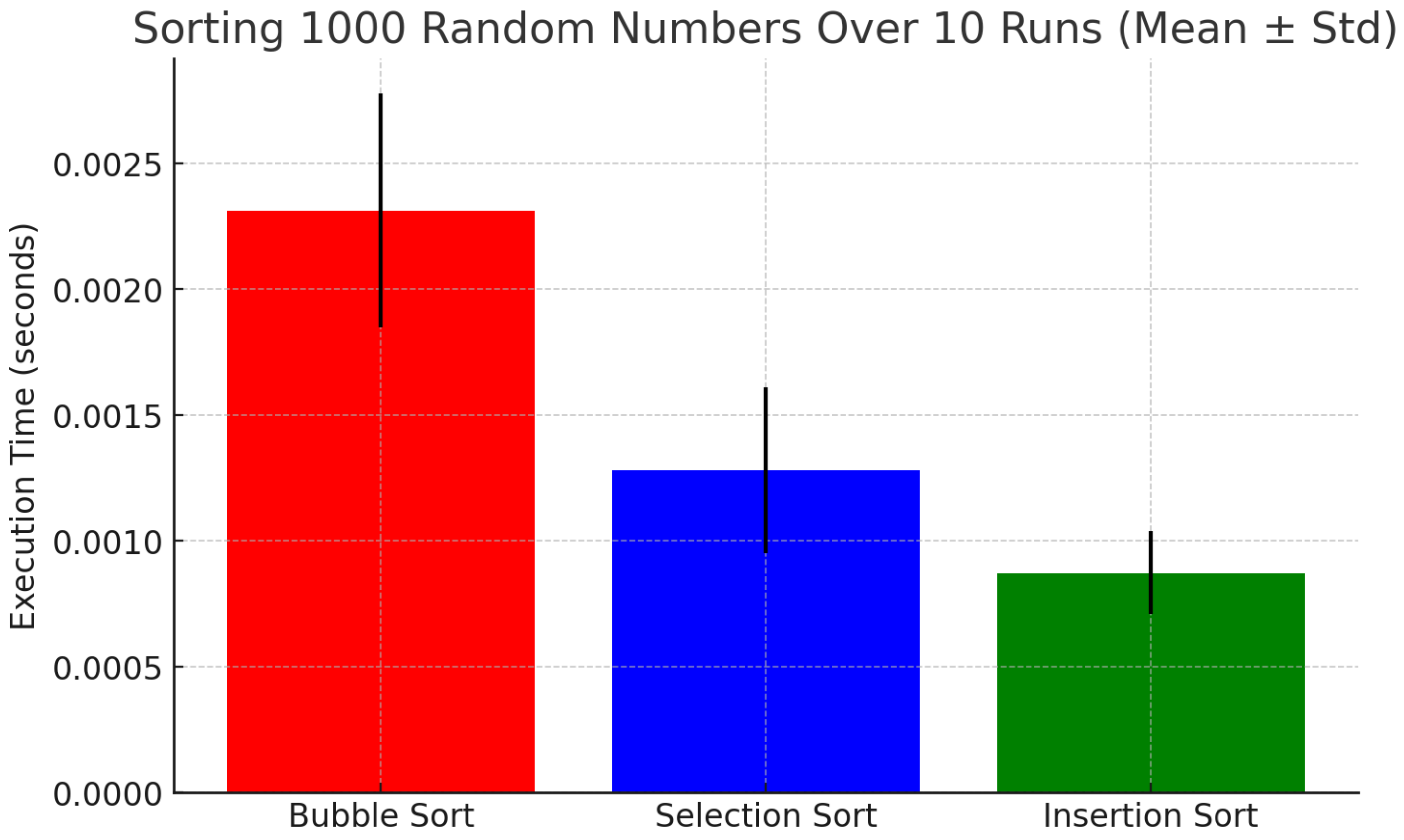
Selection sort

- Continuously finds the smallest element from the unsorted part and swaps it with the first unsorted position.

```
Input: An array A of size n (1-based index)
Output: A sorted array A in non-decreasing order
Algorithm SelectionSort(A, n):
    for i ← 1 to n do
        min_index ← i
        for j ← i+1 to n do
            if A[j] < A[min_index] then
                min_index ← j
        swap A[i] and A[min_index]
    return A
```

Pass	Unsorted Part	Min Element	Swap	Updated Array
1	**[6, 3, 8, 5, 2]**	`2` (at index 5)	Swap `A[1] ↔ A[5]`	**[2, 3, 8, 5, 6]**
2	**[3, 8, 5, 6]**	`3` (at index 2)	No swap needed	**[2, 3, 8, 5, 6]**
3	**[8, 5, 6]**	`5` (at index 4)	Swap `A[3] ↔ A[4]`	**[2, 3, 5, 8, 6]**
4	**[8, 6]**	`6` (at index 5)	Swap `A[4] ↔ A[5]`	**[2, 3, 5, 6, 8]**
5	**[8]**	No need to swap	Done	**[2, 3, 5, 6, 8]**

Comparison of elementary sorting algorithms



Comparison of elementary sorting algorithms

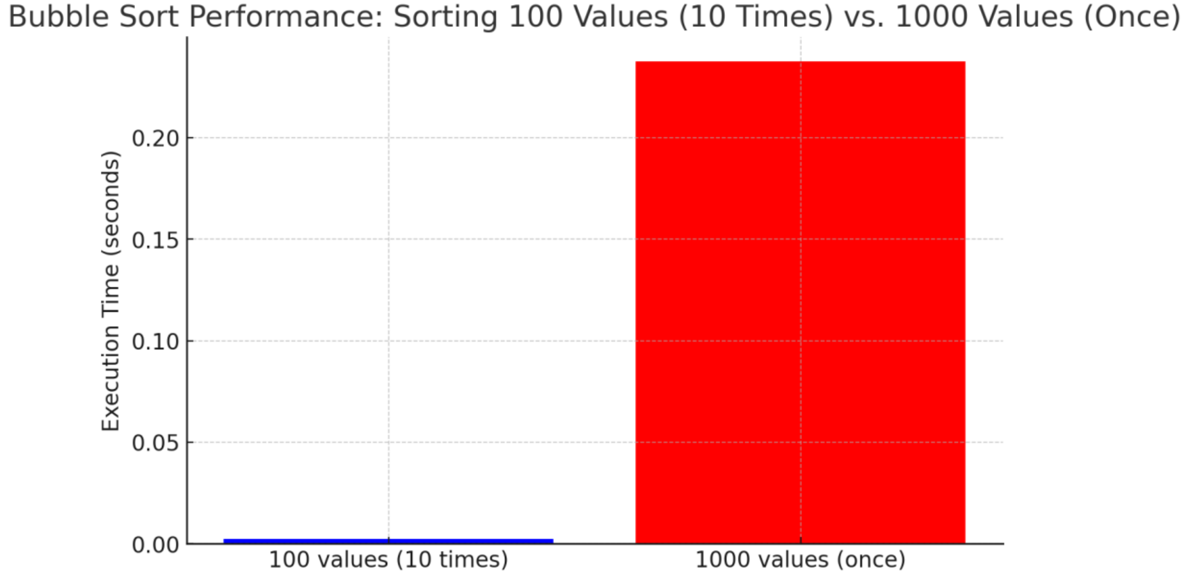
- Number of Comparisons
  - Bubble Sort: $O(n^2)$ comparisons (compares adjacent elements in every pass).
  - Selection Sort: $O(n^2)$ comparisons (finds the minimum element in each pass).
  - Insertion Sort: $O(n^2)$ comparisons (worst case), but $O(n)$ for nearly sorted arrays.
- Number of Swaps
  - Bubble Sort: $O(n^2)$ swaps (every adjacent swap is performed).
  - Selection Sort: $O(n)$ swaps (only one swap per pass).
  - Insertion Sort: $O(n)$ swaps (only shifts elements when needed); fewer swaps in nearly sorted cases.
- Best Case vs. Worst Case
  - Bubble Sort: Best case $O(n)$ (if already sorted, it can stop early).
  - Selection Sort: Always $O(n^2)$ (even if sorted, it always scans the full array).
  - Insertion Sort: Best case $O(n)$ (if sorted, only checks each element once).
- Stability and Adaptability
  - Bubble Sort and Insertion Sort are stable (maintain the order of equal elements); and adaptive (take advantage of partially sorted data).
  - Selection sort always scans the full array, and may reorder equal elements.

Merge Sort

The Problem with $O(n^2)$ Sorting Algorithms

Algorithm	Comparisons (Worst Case)	Swaps (Worst Case)
Bubble Sort	$O(n^2)$	$O(n^2)$
Selection Sort	$O(n^2)$	$O(n)$
Insertion Sort	$O(n^2)$	$O(n)$

- We need a sorting method that reduces swaps and comparisons.

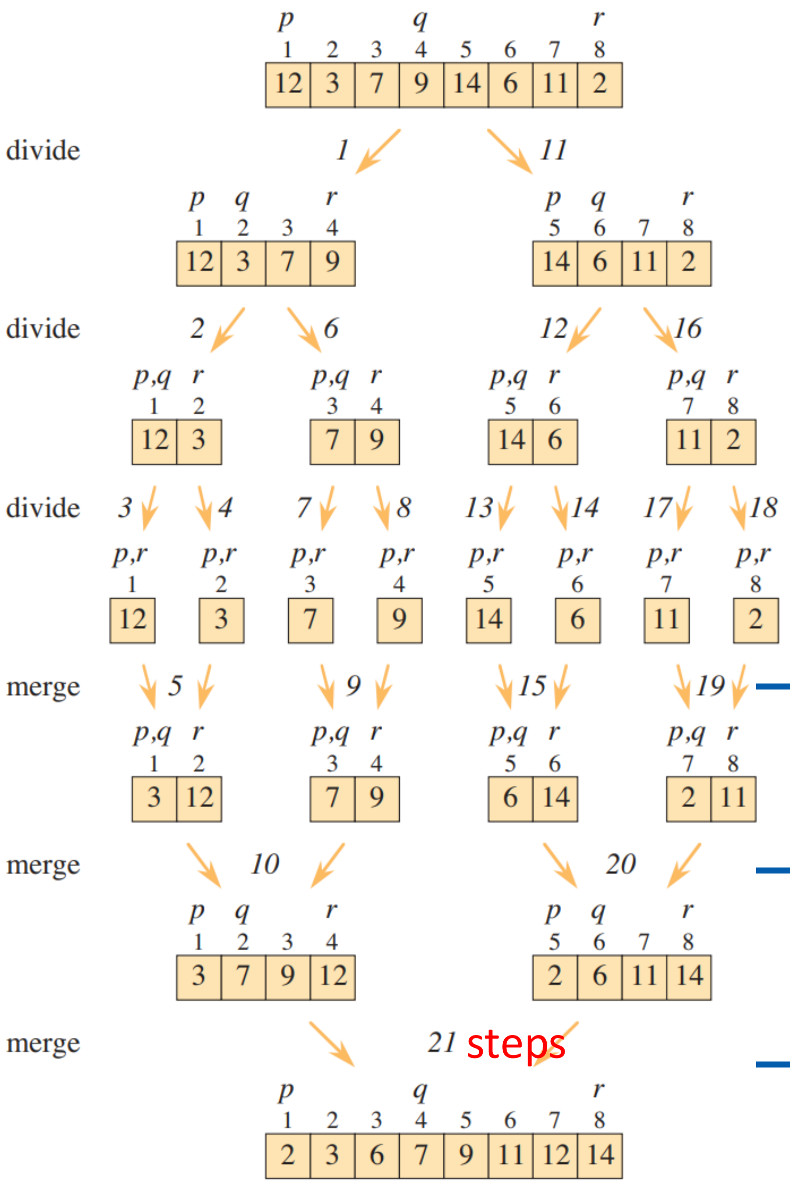


How Would You Sort 1000 Pieces of Paper?

香港科技大学(广州)
THE HONG KONG UNIVERSITY OF SCIENCE AND TECHNOLOGY (GUANGZHOU)

- Method A: Uses Insertion Sort strategy (picking one paper at a time and inserting it in order).
- Method B: Uses Merge-like strategy (first sorts small sections, then merges them together).
  - How many comparisons are needed for merging two sorted sections?
- Result: Method B finishes much faster!
- Lesson: Sorting small sections first, then merging, is faster than moving elements one by one.
  - Fewer swaps and comparisons = faster sorting!
  - Merging sorted sections is easier than sorting everything from scratch.

Merge-Sort Example



→ 1 comparison for each merge; 4 merges

→ 3 comparisons for each merge; 2 merges

→ 7 comparisons for each merge; 1 merge

17 comparisons

Merge Sort


```
Algorithm MergeSort(A, left, right):
    Input: An array A with indices left to right
    Output: A sorted array A[left:right]

    if left < right then
        mid ← (left + right) / 2
        MergeSort(A, left, mid)
        MergeSort(A, mid + 1, right)
        Merge(A, left, mid, right)
```

```
Algorithm Merge(A, left, mid, right):
    Create two temporary arrays: Left[] and Right[]
    Copy elements from A[left:mid] into Left[]
    Copy elements from A[mid+1:right] into Right[]
    i ← 1, j ← 1, k ← left
    while i ≤ length(Left) and j ≤ length(Right) do
        if Left[i] ≤ Right[j] then
            A[k] ← Left[i]
            i ← i + 1
        else
            A[k] ← Right[j]
            j ← j + 1
        k ← k + 1
    while i ≤ length(Left) do
        A[k] ← Left[i]
        i ← i + 1
        k ← k + 1
    while j ≤ length(Right) do
        A[k] ← Right[j]
        j ← j + 1
        k ← k + 1
```

Merging Two Sorted Arrays (Algo. Merge)

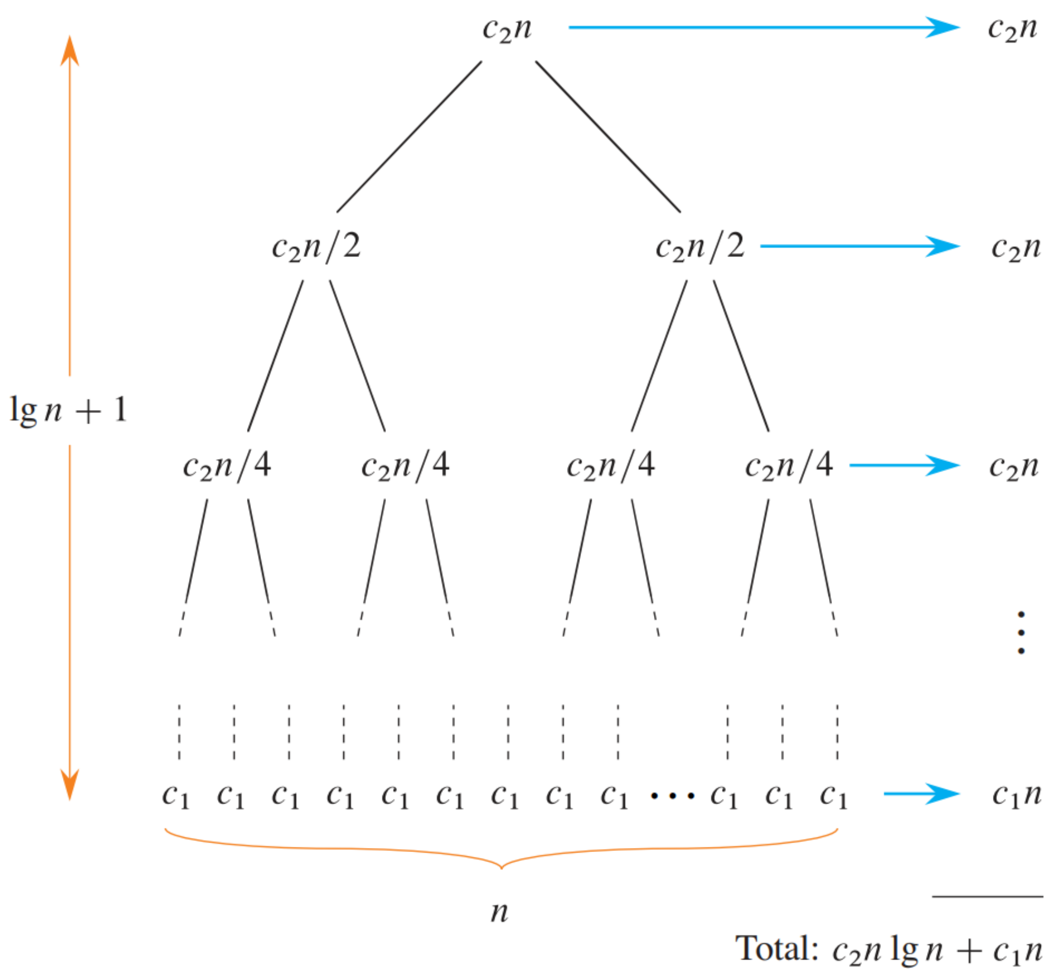
20	12
13	11
7	9
2	1

An O(nlogn) sorting algorithm

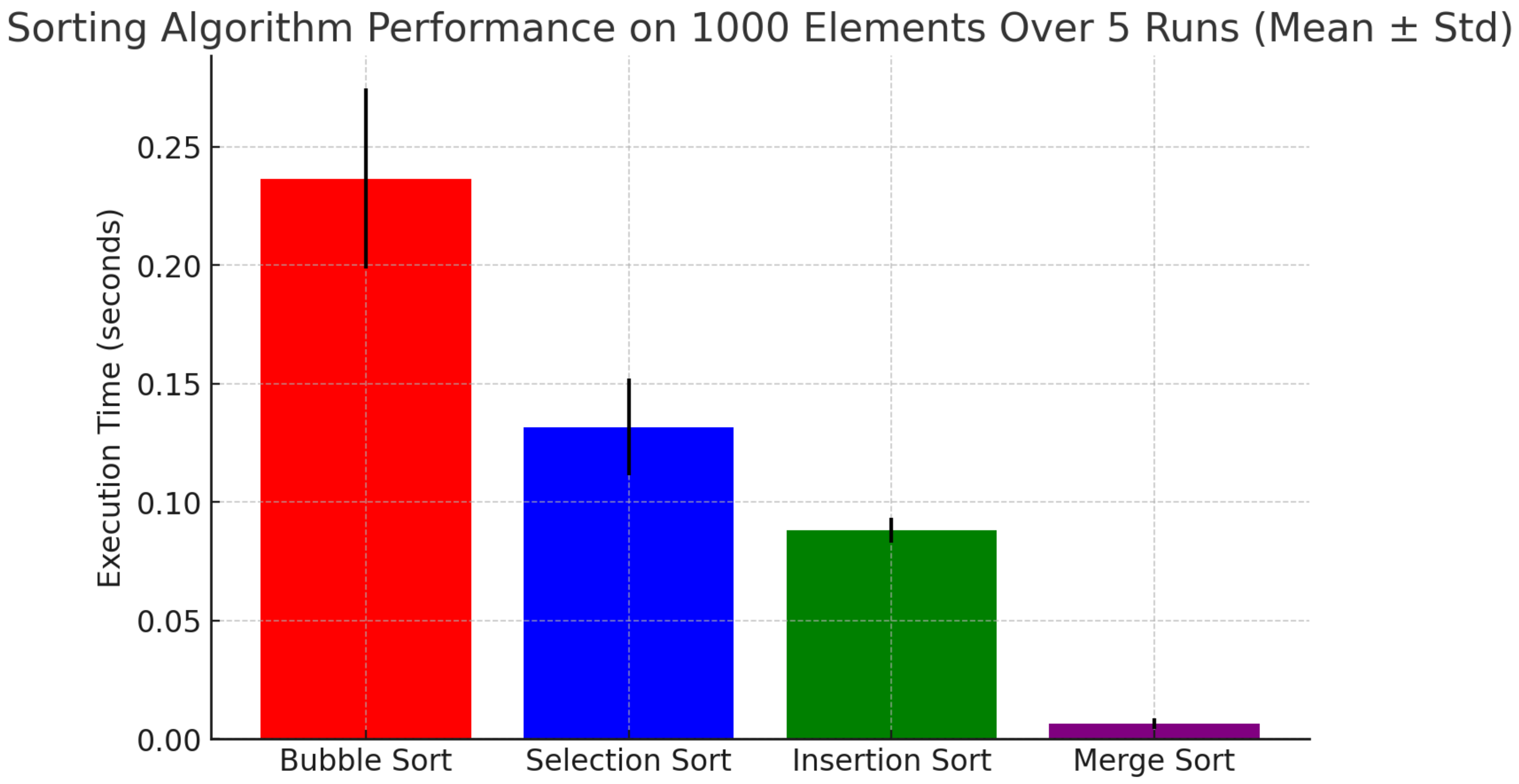

```
Algorithm MergeSort(A, left, right):
    Input: An array A with indices left to right
    Output: A sorted array A[left:right]

    if left < right then
        mid ← (left + right) / 2
        MergeSort(A, left, mid)
        MergeSort(A, mid + 1, right)
        Merge(A, left, mid, right)
```

Analysis

- Dividing the array into two halves takes O(1).
- Recursively sorting each half takes 2T(n/2).
- Merging the two sorted halves, which takes O(n).
- Thus, we have : T(n)=2T(n/2)+O(n)
- We have a total of logn+1 levels of merges. Each level costs O(n). → T(n) = O(nlogn)



Sort Algorithms Performance



Quick Sort

Merge Sort Weaknesses

- Requires extra space.
- Slower in practice due to copying.

→ We need in-place sorting

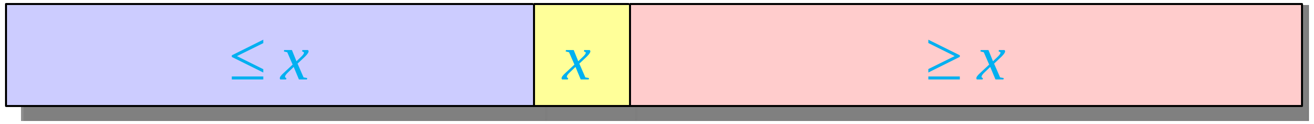
```
Algorithm Merge(A, left, mid, right):
    Create two temporary arrays: Left[] and Right[]
    Copy elements from A[left:mid] into Left[]
    Copy elements from A[mid+1:right] into Right[]
    i ← 1, j ← 1, k ← left
    while i ≤ length(Left) and j ≤ length(Right) do
        if Left[i] ≤ Right[j] then
            A[k] ← Left[i]
            i ← i + 1
        else
            A[k] ← Right[j]
            j ← j + 1
        k ← k + 1
    while i ≤ length(Left) do
        A[k] ← Left[i]
        i ← i + 1
        k ← k + 1
    while j ≤ length(Right) do
        A[k] ← Right[j]
        j ← j + 1
        k ← k + 1
```

Sorting Papers on a Table Revisits

- Imagine sorting 1000 papers on a tiny table.
- Merge Sort Approach:
  - Split into smaller piles, sort them separately, then merge.
  - Problem: Needs extra space for temporary piles.
- Quick Sort Approach:
  - Pick a pivot (e.g., middle paper).
  - Move smaller papers to the left, larger papers to the right.
  - Repeat sorting within the same space.

Quick Sort

- A popular sorting algorithm discovered by C.A.R. Hoare in 1962
  - In many situations, it's the fastest, in $O(n \log n)$ time (for in-memory sorting)
- Basic scheme
  - Partition: partition an array into two subarrays around a pivot x such that elements in left subarray $\leq x \leq$ the elements



  - Recursion: recursively apply quicksort to each of the two subarrays

Quick Sort (Pseudo-Code)

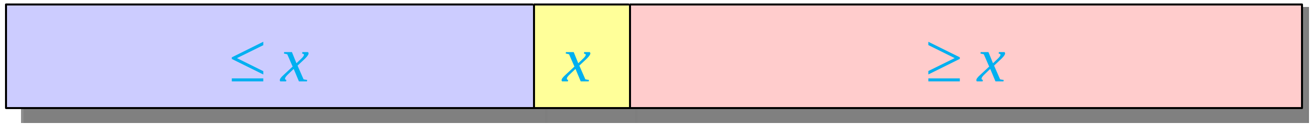
QUICKSORT(A, p, r)
    if $p < r$
        $q \leftarrow$ PARTITION(A, p, r)
        QUICKSORT$(A, p, q-1)$ //recursively sort the low side
        QUICKSORT$(A, q+1, r)$ //recursively sort the high side

Initial call: QUICKSORT$(A, 1, n)$

Partition

Divide data into two groups, such that:

- All items with a value higher than a specified amount (the pivot) are in one group
- All items with a lower value are in another



Partitioning

- Say I have 12 values:
  - **175 192 95 45 115 105 20 60 185 5 90 180**
- I pick a pivot=104, and partition (NOT sorting yet):
  - **95 45 20 60 5 90 | 175 192 115 105 185 180**
  - Note: In the future the pivot will be an actual element
  - Also: Partitioning need not maintain order of elements and usually won't, although I did in this example
- The partition is the leftmost item in the right array:
  - **95 45 20 60 5 90 | 175 192 115 105 185 180**
- Which we return to designate where the division is located

Partitioning

- The partition process (two indexs)
  - Start with two pointers: *leftIndex* initialized to one position to the left of the first cell; *rightIndex* to one position to the right of the last cell
  - *leftIndex* moves to the right; *rightIndex* moves to the left

- Stopping and Swapping
  - When *leftIndex* encounters an item smaller than the pivot, it keeps going; when it finds a larger item, it stops
  - When *rightIndex* encounters an item larger than the pivot, it keeps going; when it finds a smaller item, it stops
  - When the two *indexs* eventually meet, the process is complete
  - When the two *indexs* stop, swap the two elements

Efficiency: Partitioning

- O(n) time
  - left starts at 0 and moves one-by-one to the right
  - right starts at n-1 and moves one-by-one to the left
  - When left and right cross, we stop.
    - So we'll hit each element just once
- Number of comparisons is n+1
- Number of swaps is worst case n/2
  - Worst case, we swap every single time
  - Each swap involves two elements
  - Usually, it will be less than this
    - Since in the random case, some elements will be on the correct side of the pivot

Modified Partitioning

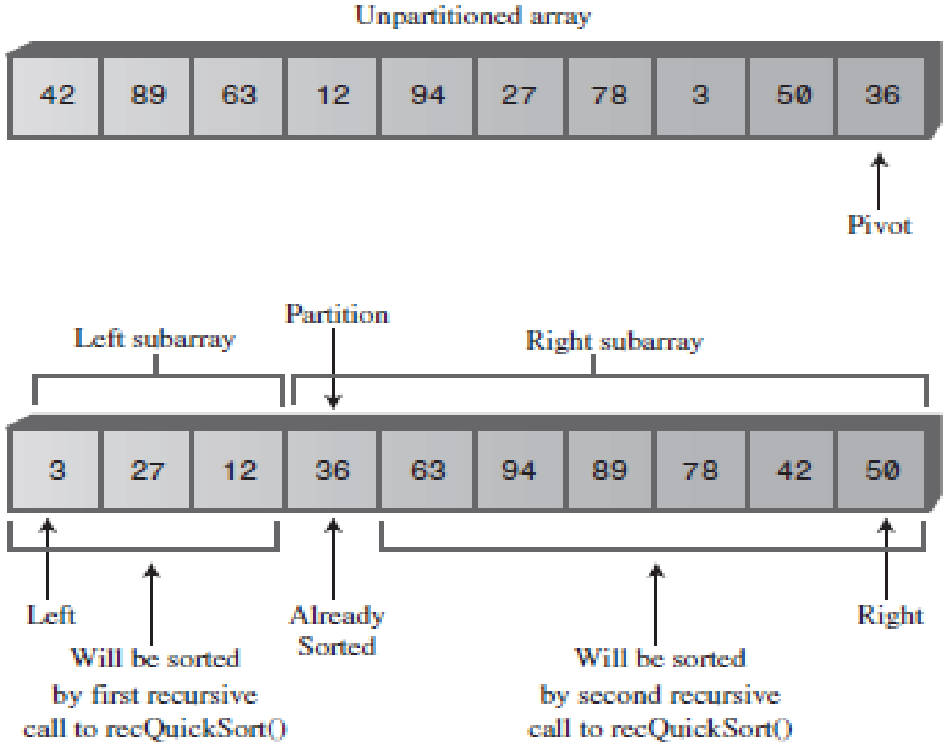
- In preparation for Quicksort:
  - Choose our pivot value to be the rightmost element
  - Partition the array around the pivot
  - Ensure the pivot is at the location of the partition
    - Meaning, the pivot should be the leftmost element of the right subarray
- Example: Unpartitioned **42 89 63 12 94 27 78 3 50 36**
- Partitioned around Pivot: **3 27 12 36 63 94 89 78 42 50**
- What does this imply about the pivot element after the partition?

Placing the PIVOT

- Goal: Pivot must be in the leftmost position in the right subarray
  - **3 27 12 36 63 94 89 78 42 50**
- Our algorithm does not do this currently
- It currently will not touch the pivot
  - left increments till it finds an element > pivot
  - right decrements till it finds an element < pivot
  - So the pivot itself won’t be touched, and will stay on the right:
  - **3 27 12 63 94 89 78 42 50 36**

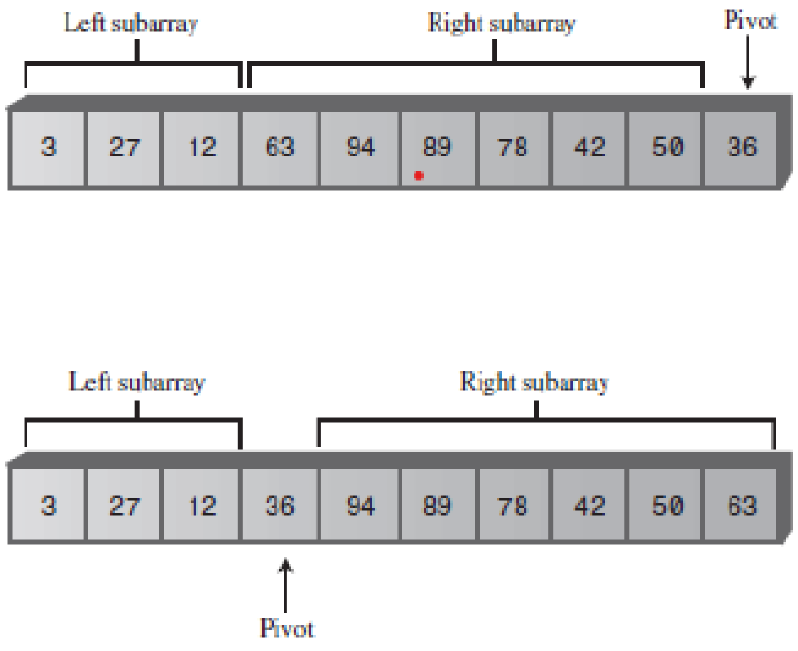
Shifting the PIVOT

- We have this:
  - 3 27 12 63 94 89 78 42 50 36
- Our goal is the position of 36
- Shift every element in the right suba
  - 3 27 12 36 63 94 89 78 42 50



Recursive calls sort subarrays.

Swapping the PIVOT



Swapping the pivot.

- Just swap the leftmost with the pivot! Better
  - 3 27 12 36 **94 89 78 42 50 63**
  - We can do this because the right subarray is not in any particular order
- Just takes one more line to our Python method
  - Basically, a single call to swap()
  - Swaps A[end-1] (the pivot) with A[left]

    (the partition index)

Partition

```
Algorithm Partition(A, left, right):
    Input: Array A, starting index left, ending index right
    Output: Index of the pivot after rearrangement

    pivot ← A[left]   // Choose first element as pivot
    leftIndex ← left + 1
    rightIndex ← right
    while true do:
        // Move leftIndex to the right until finding an element >= pivot
        while leftIndex ≤ right and A[leftIndex] < pivot do:
            leftIndex ← leftIndex + 1
        // Move rightIndex to the left until finding an element <= pivot
        while rightIndex ≥ left and A[rightIndex] > pivot do:
            rightIndex ← rightIndex - 1
        if leftIndex ≥ rightIndex then:
            break  // Indices have crossed, partitioning is complete
        swap A[leftIndex] and A[rightIndex]  // Swap elements
    swap A[left] and A[rightIndex]  // Move pivot to correct position
    return rightIndex  // Return final position of pivot
```

Shall We Try It On An ARRAY?

1, 7, 5, 3, 6, 9, 0, 4, 8, 2

Let’s go step-by-step via Quick Sort

BEST Case…

- We partition the array each time into two equal subarrays
- Say we start with array of size n = 2^i
- We recurse until the base case, 1 element

- Draw the tree
  - First call -> Partition n elements, n operations
  - Second calls -> Each partition n/2 elements, 2(n/2)=n operations
  - Third calls -> Each partition n/4, 4(n/4) = n operations
  - …
  - (i+1)th calls -> Each partition n/2^i = 1, 2^i(1) = n(1) = n ops
- Total: (i+1)*n = (log n + 1)*n -> O(n log n)

The Very BAD Case....

- If the array is sorted
- Let’s see the problem:
  - **0 10 20 30 40 50 60 70 80 90**
- What happens after the partition? This:
  - **0 10 20 30 40 50 60 70 80 90**
- This is sorted, but the algorithm doesn’t know it.
- It will then call itself on an array of zero size (the left subarray) and an array of n-1 size (the right subarray).
- Producing:
  - **0 10 20 30 40 50 60 70 80 90**

The Very BAD Case....

- In the worst case, we partition every time into an array of n-1 elements and an array of 0 elements
- This yields O(n^2) time:
  - First call: Partition n elements, n operations
  - Second calls: Partition n-1 and 0 elements, n-1 operations
  - Third calls: Partition n-2 and 0 elements, n-2 operations
  - Draw the tree
- Yielding: Operations = n + n-1 + n-2 + … + 1 = n(n+1)/2 -> O(n^2)

Choosing Pivot

- What caused the problem was “blindly” choosing the pivot from the right end.
- In the case of a reverse sorted array, this is not a good choice at all
- Can we improve our choice of the pivot? Let’s choose the middle of three values

Median-Of-Three Partitioning

- Every time you partition, choose the median value of the left, center and right element as the pivot
- Example:
  - **44 11 55 33 77 22 00 99 101 66 88**
- Pivot: Take the median of the leftmost, middle and rightmost
  - **44 11 55 33 77 22 00 99 101 66 88 - Median: 44**
- Then partition around this pivot:
  - **11 00 33 22 44 77 55 99 101 66 88**
- Increases the likelihood of an equal partition
  - Also, it cannot possibly be the worst case

How This Fixes The WORST Case?

- Here's our array:
  - **0 10 20 30 40 50 60 70 80 90**

- Let's see on the board how this fixes things
- In fact in a perfectly sorted array, we choose the middle element as the pivot!
  - Which is optimal
  - We get $O(N\log N)$

- Vast majority of the time, if you use QuickSort with a Median-Of-Three partition, you get $O(N\log N)$ behavior

One Final Optimization…

- After a certain point, just doing insertion sort is faster than partitioning small arrays and making recursive calls

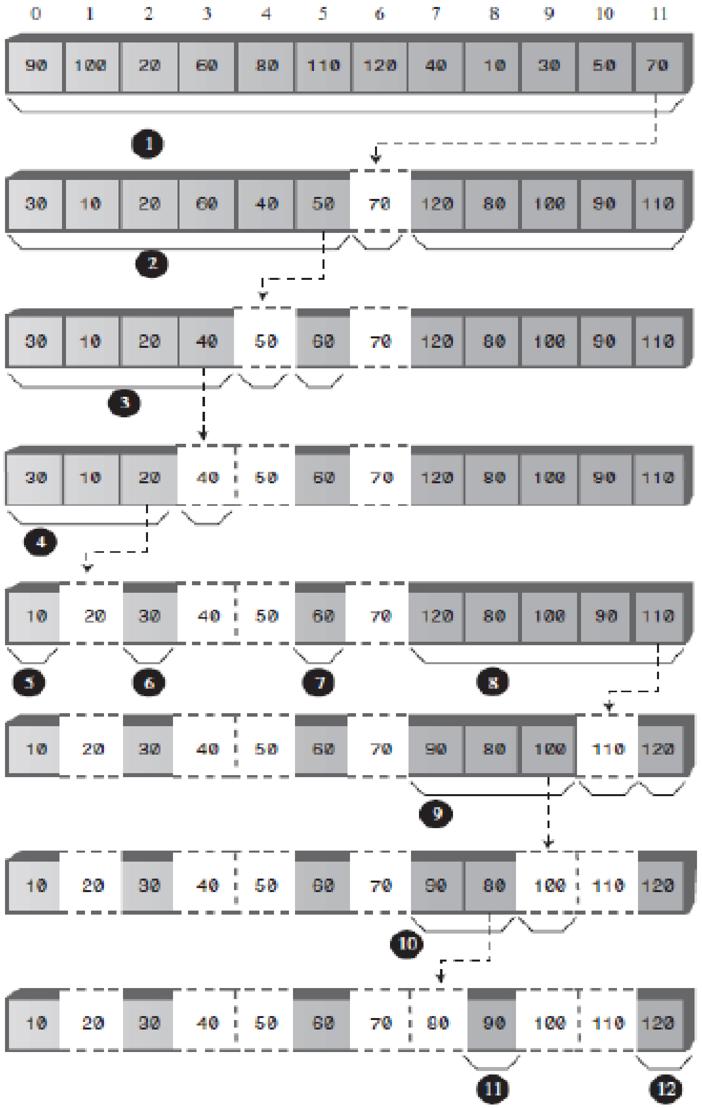
- Once you get to a very small subarray, you can just sort with insertion sort
- You can experiment a bit with 'cutoff' values
  - Knuth: n=9

Operation Count Estimates

- For QuickSort
- n=8: 30 comparisons, 12 swaps
- n=12: 50 comparisons, 21 swaps
- n=16: 72 comparisons, 32 swaps
- n=64: 396 comparisons, 192 swaps
- n=100: 678 comparisons, 332 swaps
- n=128: 910 comparisons, 448 swaps

- The only competitive algorithm is MergeSort
  - But, takes much more memory like we said

Summary of Quicksort


The quicksort process.

- Quick sort operates in $O(N*\log N)$ time (except when the simpler version is applied to already-sorted data).
- Subarrays smaller than a certain size (the cutoff) can be sorted by a method other than quicksort.
- The insertion sort is commonly used to sort subarrays smaller than the cutoff.
- The insertion sort can also be applied to the entire array, after it has been sorted down to a cutoff point by quicksort.

Swaps and Comparisons in Quicksort

N	8	12	16	64	100	128
$\log_2 N$	3	3.59	4	6	6.65	7
$N*\log_2 N$	24	43	64	384	665	896
Comparisons: $(N+2)*\log_2 N$	30	50	72	396	678	910
Swaps: fewer than $N/2*\log_2 N$	12	21	32	192	332	448

*The $\log_2 N$ quantity used in the table is true only in the best-case scenario, where each subarray is partitioned exactly in half. For random data, it is slightly greater.

Sort algorithm performance