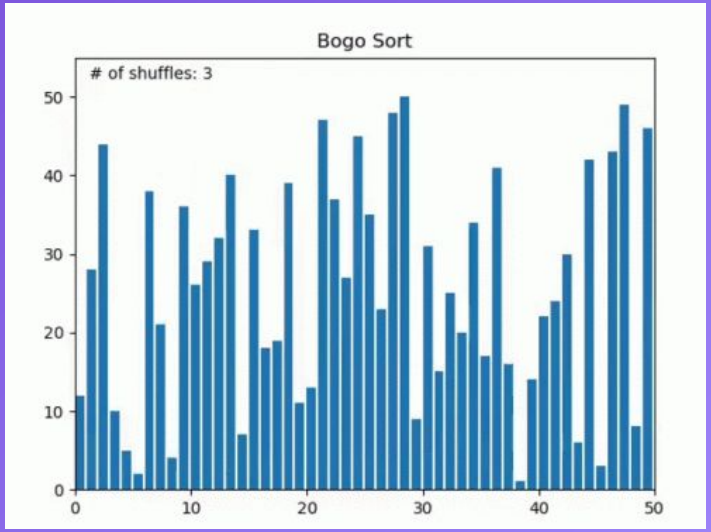
Sorting

CSE 332

Slides by James Richie Sulaeman

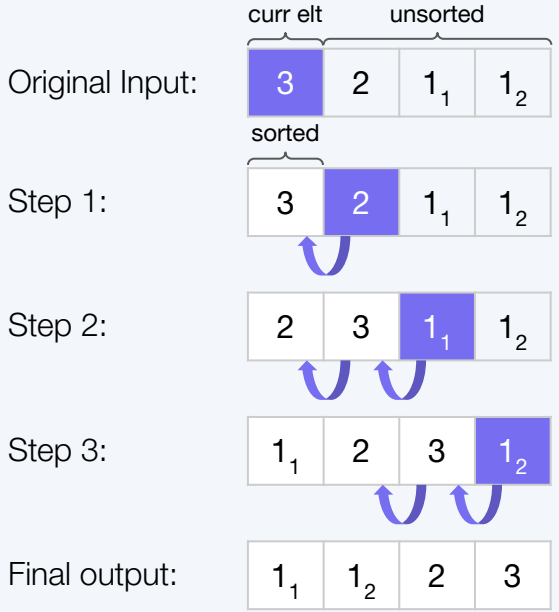


Comparison-based Sorting

Insertion Sort

Builds a sorted subarray at the front of the original array. Takes the first element of the unsorted subarray and inserts it into the sorted subarray.

- In-place and stable.
- Best-case runtime: $\mathcal{O}(n)$
- Worst-case runtime: $\mathcal{O}(n^2)$

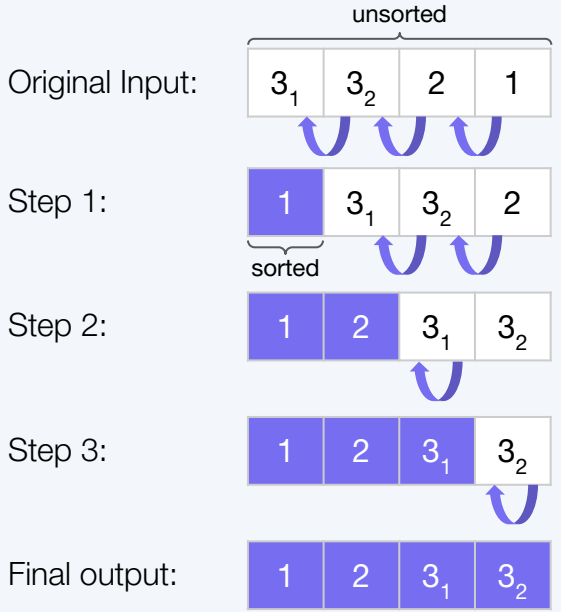


Selection Sort

Builds a sorted subarray at the front of the original array. Takes the smallest element of the unsorted subarray and **swaps it into the correct location** at the end of the sorted subarray.

- In-place and **not usually stable**
  - Can be made stable if you **shift** instead of swapping and break ties by selecting the left-most element (see image at right)
- Best-case runtime: $\mathcal{O}(n^2)$
- Worst-case runtime: $\mathcal{O}(n^2)$

A stable version of Selection Sort

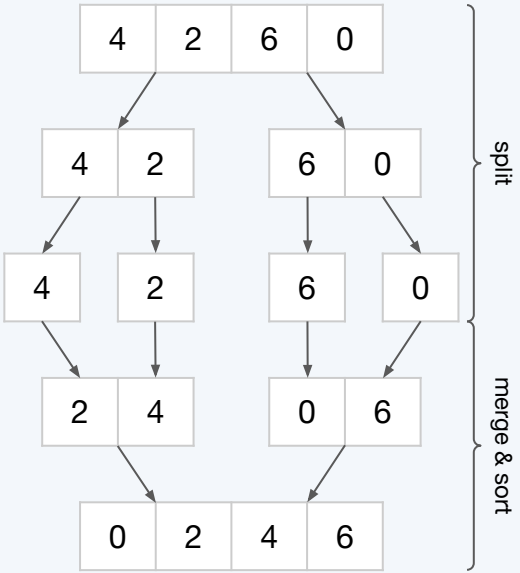


Merge Sort

```
mergeSort(input) -> sorted input:
1. sortedLeft = mergeSort(left half of input)
2. sortedRight = mergeSort(right half of input)
3. return (merged 'sortedLeft' and 'sortedRight')
```

Recursively splits an array into two halves, sorts them, and then merges them back together to obtain a sorted array.

- Not in-place but stable.
- Best-case runtime: $\mathcal{O}(n \log n)$
- Worst-case runtime: $\mathcal{O}(n \log n)$

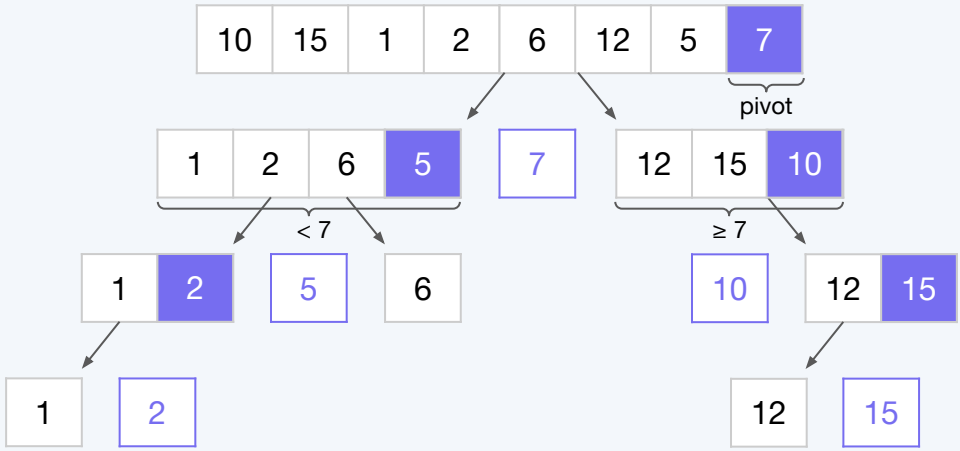


Quick Sort

```
quickSort(input) -> void:
1. pick pivot
2. partition input into 'lessThanPivot' and
   'greaterThanPivot' parts
3. quickSort(lessThanPivot)
4. quickSort(greaterThanPivot)
```

Recursively partitions an array based on a pivot element, sorts the subarrays on either side of the pivot, and merges them back together to obtain a sorted array.

- In-place but not stable.
- Best-case runtime: $\mathcal{O}(n \log n)$
- Worst-case runtime: $\mathcal{O}(n^2)$



Problem 0

Problem 0

Suppose we sort an array of numbers, but it turns out every element of the array is the same (e.g. `[17, 17, 17, ..., 17]`).

What is the asymptotic runtime of the following sorting algorithms?

Sorting Algorithm	Asymptotic Runtime	Explanation
Insertion Sort		
Selection Sort		
Merge Sort		
Quick Sort		

Problem 0

Suppose we sort an array of numbers, but it turns out every element of the array is the same (e.g. `[17, 17, 17, ..., 17]`).

What is the asymptotic runtime of the following sorting algorithms?

Sorting Algorithm	Asymptotic Runtime	Explanation
Insertion Sort	$\mathcal{O}(n)$	Insertion Sort will traverse the array, but since it is already 'sorted', no extra computation is necessary
Selection Sort	$\mathcal{O}(n^2)$	Selection Sort always has $\mathcal{O}(n^2)$ runtime since it has to find the smallest item in the unsorted subarray n times
Merge Sort	$\mathcal{O}(n \log n)$	Merge Sort always has $\mathcal{O}(n \log n)$ runtime. You can prove this using recurrences!
Quick Sort	$\mathcal{O}(n^2)$	This is the worst case for Quick Sort. Since all the elements are the same, all items will end up on the same side of each partition.

Non-Comparison Sorting

Bucket Sort

```
bucketSort(input) -> sorted input:
1. create array of size B
2. put each element into their corresponding bucket
3. generate sorted array by iterating through the
   buckets in order
```

Distributes elements into their corresponding buckets. Buckets have an inherent ordering and are merged together in this ordering to produce the sorted array.

- Not in-place but stable.
- Runtime: $\mathcal{O}(n + B)$
  - Need to iterate over n elements and B buckets.
  - Good when $B \ll n$ or $B \approx n$.
  - Bad when $B \gg n$.

B: 5

Original Input:

$[5_1, 1_1, 3_1, 4_1, 3_2, 2, 1_2, 1_3, 5_2, 4_2, 5_3]$

Bucket Array	
1	$1_1, 1_2, 1_3$
2	2
3	$3_1, 3_2$
4	$4_1, 4_2$
5	$5_1, 5_2, 5_3$

Final output:

$[1_1, 1_2, 1_3, 2, 3_1, 3_2, 4_1, 4_2, 5_1, 5_2, 5_3]$

Radix Sort

```
radixSort(input) -> sorted input:
1. create array of size b
2. for each significant digit:
   3. run bucket sort on the elements using their
      corresponding significant digit values
```

Repeatedly runs bucket sort on the elements for each significant digit, from least significant to most significant.

Original Input: `[478, 537, 9, 721, 3, 38, 143, 67]` b: 10

Bucket Sort on 1's Digit

0	1	2	3	4	5	6	7	8	9
	721		3, 143				537, 67	478, 38	9

Bucket Sort on 10's Digit

0	1	2	3	4	5	6	7	8	9
3, 9		721	537, 38	143		67	478		

Radix Sort

```
radixSort(input) -> sorted input:
1. create array of size b
2. for each significant digit:
   3. run bucket sort on the elements using their
      corresponding significant digit values
```

Repeatedly runs bucket sort on the elements for each significant digit, from least significant to most significant.

Original Input: `[478, 537, 9, 721, 3, 38, 143, 67]` b: 10

Bucket Sort on 10's Digit

0	1	2	3	4	5	6	7	8	9
3, 9		721	537, 38	143		67	478		

- Notice how elements are now sorted with respect to their last two digits.
- By running bucket sort from the least to the most significant digit, the order of the more significant digits take precedence over the less significant digits.

Radix Sort

```
radixSort(input) -> sorted input:
1. create array of size b
2. for each significant digit:
   3. run bucket sort on the elements using their
      corresponding significant digit values
```

Repeatedly runs bucket sort on the elements for each significant digit, from least significant to most significant.

Original Input: `[478, 537, 9, 721, 3, 38, 143, 67]` b: 10

Bucket Sort on 10's Digit

0	1	2	3	4	5	6	7	8	9
3, 9		721	537, 38	143		67	478		

Bucket Sort on 100's Digit

0	1	2	3	4	5	6	7	8	9
3, 9, 38, 67	143			478	537		721		

Radix Sort

```
radixSort(input) -> sorted input:
1. create array of size b
2. for each significant digit:
   3. run bucket sort on the elements using their
      corresponding significant digit values
```

Repeatedly runs bucket sort on the elements for each significant digit, from least significant to most significant.

Original Input: `[478, 537, 9, 721, 3, 38, 143, 67]` b: 10

Bucket Sort on 100's Digit

0	1	2	3	4	5	6	7	8	9
3, 9, 38, 67	143			478	537		721		

Output: `[3, 9, 38, 67, 143, 478, 537, 721]`

- Not in-place but stable.
- Runtime: $\mathcal{O}(p(n+b))$
  - Bucket sort has $\mathcal{O}(n+b)$ runtime, and we're running it $p = \log_b(\max(n))$ times.

Thank You!