

# CSC 226

## Algorithms and Data Structures: II Sorting Revisited

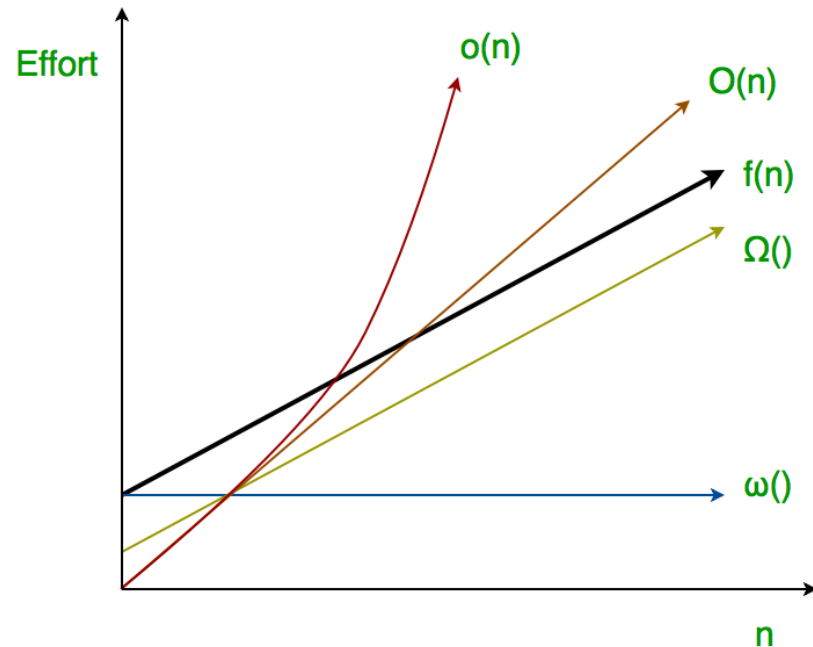
Tianming Wei

[twei@uvic.ca](mailto:twei@uvic.ca)

ECS 466

# Previous: Asymptotic Notation Review

- Big-Oh
- Big-Omega
- Big-Theta
- Little-oh
- Little-omega



# Previous: Asymptotic Notation Review

Which one is FALSE in the following?

A.  $n^2 \in O(n \log^2 n)$

B.  $n \log n \in O(n^2)$

C.  $f(n) \in O(g(n)) \Rightarrow f(n) \in o(g(n))$

D.  $f(n) \in o(g(n)) \Rightarrow f(n) \in O(g(n))$

# Previous: Asymptotic Notation Review

Which one is FALSE in the following?

A.  $n^2 \in O(n \log^2 n)$  FALSE

B.  $n \log n \in O(n^2)$  True

C.  $f(n) \in O(g(n)) \Rightarrow f(n) \in o(g(n))$  FALSE

D.  $f(n) \in o(g(n)) \Rightarrow f(n) \in O(g(n))$  True

# Previous: Asymptotic Notation Review

Big-O Notation	Comparison Notation	Limit Definition
$f \in o(g)$	$f \oslash g$	$\lim_{x \rightarrow \infty} \frac{f(x)}{g(x)} = 0$
$f \in O(g)$	$f \oslash\leq g$	$\lim_{x \rightarrow \infty} \frac{f(x)}{g(x)} < \infty$
$f \in \Theta(g)$	$f \oslash\equiv g$	$\lim_{x \rightarrow \infty} \frac{f(x)}{g(x)} \in \mathbb{R}_{>0}$
$f \in \Omega(g)$	$f \oslash\geq g$	$\lim_{x \rightarrow \infty} \frac{f(x)}{g(x)} > 0$
$f \in \omega(g)$	$f \oslash> g$	$\lim_{x \rightarrow \infty} \frac{f(x)}{g(x)} = \infty$

# Sorting revisited

- Overview of sorting algorithms

# Different types of Sorting Algorithms

- Comparison Based Sorting
  - sorting algorithm that sorts based only on comparisons
  - elements to be sorted must satisfy total order properties
- Integer Sorting
  - sorting algorithm that sorts a collection of data values by numeric keys, each of which is an integer



# Selection Sort

**Algorithm** selectionSort(A,n) :

**Input:** Array A of size n

**Output:** Array A sorted

```
for k ← 0 to n-2 do
  min ← k
  for j ← k+1 to n-1 do
    if A[j] < A[min] then
      min ← j
    end
  end
  swap(A[k], A[min])
end
end
```



# Properties of Selection Sort

- *Comparisons*: Worst-case = best-case = average-case =  $O(n^2)$
- exactly  $n - 1$  swaps
- Simple implementation
- *Not adaptive*: The input order has no effect on the efficiency of the algorithm
- *In-place*: only requires a constant amount of (additional) memory

# Linear Insertion/Insertion Sort (Pseudocode)

**Algorithm** insertionSort(A, n) :

**Input:** Array A of size n

**Output:** Array A sorted

**for**  $k \leftarrow 1$  **to**  $n-1$  **do**

$val \leftarrow A[k]$

$j \leftarrow k-1$

**while**  $j \geq 0$  **and**  $A[j] > val$  **do**

$A[j+1] \leftarrow A[j]$

$j \leftarrow j - 1$

**end**

$A[j+1] = val$

**end**

**end**

# Complexity of Insertion Sort

- *Comparisons:*
  - Worst-case:  $O(n^2)$
  - Best-case:  $O(n)$
  - Average-case:  $O(n^2)$
- $O(n)$  space
- *Swaps:*
  - Worst-case:  $O(n^2)$
  - Best-case:  $O(n)$

# Some Properties of Insertion Sort

- Simple implementation
- Efficient for
  - (quite) small data sets
  - data sets of already substantially sorted sequences: more efficient in practice than most other simple quadratic algorithms (*adaptive*)
- *In-place*: only requires a constant amount of (additional) memory
- *Online*: updates sort as elements are received
- *Natural*: people often use an insertion like sorting in practice

# Quicksort

**Algorithm** *quickSort*(*A*, *a*, *b*)

**Input** Array *A*, ranks *a* and *b*

**Output** *A* with the elements ordered between *a* and *b*

**if** *a*  $\geq$  *b* **then return**

*p*  $\leftarrow$  *A*[*b*]

*l*  $\leftarrow$  *a*

*r*  $\leftarrow$  *b* - 1

**while** *l*  $\leq$  *r* **do**

**while** *l*  $\leq$  *r* **and** *A*[*l*]  $\leq$  *p* **do**

*l*  $\leftarrow$  *l* + 1

**while** *r*  $\geq$  *l* **and** *A*[*r*]  $\geq$  *p* **do**

*r*  $\leftarrow$  *r* - 1

**if** *l* < *r* **then**

        swap(*A*[*l*], *A*[*r*])

swap(*A*[*l*], *A*[*b*])

*quickSort*(*S*, *a*, *l* - 1)

*quickSort*(*S*, *l* + 1, *b*)

# Properties of Quicksort

- Worst case time complexity:  $O(n^2)$
- Expected:  $O(n \log(n))$ 
  - since, when picked at random, the pivot is likely good and splits up the sequence pretty evenly ...
- Best-case:  $O(n \log n)$
- Mergesort has  $O(n \log(n))$  worst case time complexity, but quicksort is better in practice
- Usually the best choice unless the recursion gets too deep or stability is needed

# Mergesort (Pseudocode)

**Algorithm** mergeSort( $S$ ) :

**input:** Sequence  $S$

**output:** Sequence  $S$  sorted

**if**  $S.size() < 2$  **then**

**return**  $S$

$S_1, S_2 \leftarrow \text{divide}(S)$

$S_1 \leftarrow \text{mergeSort}(S_1)$

$S_2 \leftarrow \text{mergeSort}(S_2)$

$S \leftarrow \text{merge}(S_1, S_2)$

**return**  $S$

# Merge (Pseudocode)

```
Algorithm merge( $S_1$ ,  $S_2$ ) :  
while not ( $S_1.isEmpty()$  or  $S_2.isEmpty()$ ) do  
    if  $S_1.first().key() < S_2.first().key()$  then  
         $S.insertLast(S_1.removeFirst())$   
    else  
         $S.insertLast(S_2.removeFirst())$   
    end  
end  
while not ( $S_1.isEmpty()$ ) do  
     $S.insertLast(S_1.removeFirst())$   
end  
while not ( $S_2.isEmpty()$ ) do  
     $S.insertLast(S_2.removeFirst())$   
end  
return  $S$ 
```



# Properties of Mergesort

- Worst-case = best-case = average case =  $O(n \log n)$
- $O(n)$  extra space

# Heapsort (Pseudocode)

**Algorithm** heapSort(Array A, int n)

**input:** array A of size n

**output:** array A sorted

heapify(A, n) //make A a maxHeap

end  $\leftarrow$  n-1

**while** end > 0 **do**

    swap(A[end], A[0])

    end  $\leftarrow$  end - 1

    bubbleDown(A, 0, end)

# Properties of Heapsort

- Worst-case = best-case = average case =  $O(n \log n)$
- In-place (constant extra space)
- Quick sort still outperforms it in most cases, why?

# Radixsort LSD

- Take the least significant digit (or group of bits) of each key.
- Group the keys based on that digit into buckets, but otherwise keep the original order of keys.
- Concatenate the buckets together in order.
- Repeat the grouping process with each more significant digit.

# Radixsort MSD

- Take the most significant digit of each key.
- Sort the list of elements based on that digit, grouping elements with the same digit into one bucket.
- Recursively sort each bucket, starting with the next digit to the right.
- Concatenate the buckets together in order.

# Properties of Radixsort

- Repeated sorting by means of Bucket Sort on  $n$  keys
  - For each component of the key perform one Bucket Sort
- Implement buckets ( $N$  of them) as queues
- Let the number of components per key be  $d$
- **Theorem. The time complexity of Radix Sort is  $O(d(n + N))$  or  $O(dn)$  for large  $n$ .**

# Shellsort (Pseudocode)

**Algorithm** Shellsort ( $A, n$ )

**input:** array  $A$  of size  $n$

**output:**  $A$  sorted

$h \leftarrow 1$

**while**  $h < n/3$  **do**  $h \leftarrow 3h + 1$

**while**  $h \geq 1$  **do**

**for**  $i = h$  **to**  $n-1$  **do**

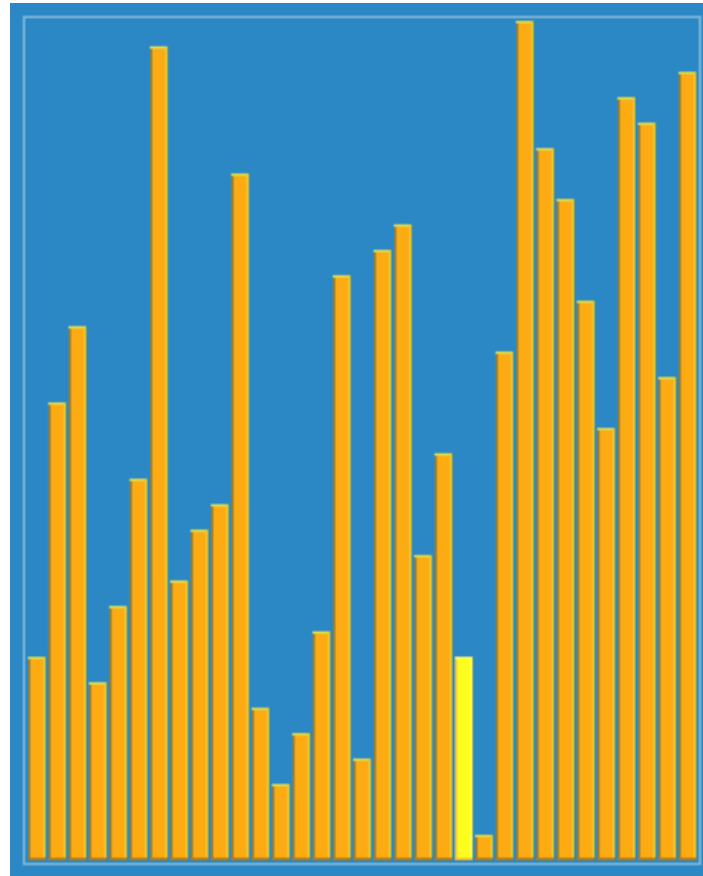
$j \leftarrow i$

**while**  $j \geq h$  and  $A[j] < A[j-h]$  **do**

            swap( $A[j], A[j-h]$ )

$h \leftarrow h/3$

# Shellsort (Animation)





# Properties of Shellsort

- The analysis of Shellsort is dependent on the numeric properties of  $n$ , the size of  $h$  and the step size and in some cases is still an open question
- This algorithm decreases  $h$  by the sequence  $(3^k - 1)/2$  which has a worst-case time of  $O(n^{3/2})$
- Is this better than  $O(n \log n)$ ?

# Bubble Sort (Pseudocode)

**Algorithm** bubbleSort( $A, n$ ) :

**Input:** Array  $A$  of size  $n$

**Output:** Array  $A$  sorted

**repeat**

    swapped  $\leftarrow$  false

**for**  $i = 1$  **to**  $n-1$  **do**

**if**  $A[i-1] > A[i]$  **then**

            swap( $A[i-1], A[i]$ )

            swapped  $\leftarrow$  true

$n \leftarrow n-1$

**until** not swapped

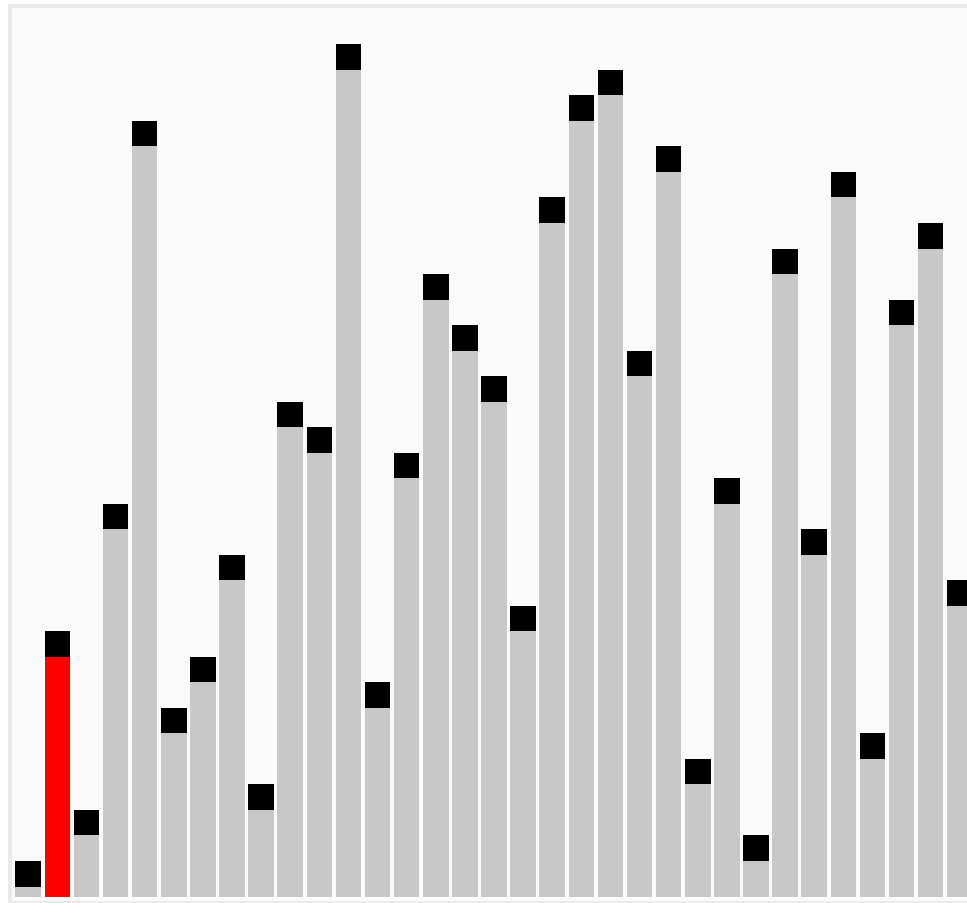
# Complexity of Bubble Sort

- Worst-case = average-case =  $O(n^2)$
- Best-case =  $O(n)$

# Shakersort

- Variation of bubble sort: sorts in both directions instead of just one as in bubble sort
- slightly better performance than bubble sort
- worse than insertion sort in practice

# Shakersort



	Type of Sorting Algorithm	Worst Case Time	Best Case Performance	Average Case Performance	Properties
Insertion Sort	Comparison Based Sorting	$O(n^2)$	$O(n)$	$O(n^2)$	adaptive, in place, stable, online
Bubblesort	Comparison Based Sorting	$O(n^2)$	$O(n)$	$O(n^2)$	in place
Selection Sort	Comparison Based Sorting	$O(n^2)$	$O(n^2)$	$O(n^2)$	in place
Binary Insertion	Comparison Based Sorting	$O(n^2)$	$O(n)$	$O(n^2)$	adaptive, in place
Shakersort	Comparison Based Sorting	$O(n^2)$	$O(n)$	$O(n^2)$	stable, in place
Shellsort	Comparison Based Sorting	$O(n^2)$	$O(n \log n)$		in place
Quicksort	Comparison Based Sorting	$O(n^2)$	$O(n \log n)$	$O(n \log n)$	in place
Heapsort	Comparison Based Sorting	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	in place
Mergesort	Comparison Based Sorting	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	not in place
Bucketsort	Integer Sorting	$O(n+k)$		$O(n+k)$	can be implemented such that stable
Radixsort	Integer Sorting	$O(dn)$			stable

# Sorting out Sorting

- Visualization and Comparison of Sorting Algorithms
- <https://www.youtube.com/watch?v=ZZuD6iUe3Pc>