



Data Structure and Algorithms [CO2003]

Chapter 10 - Sort

Lecturer: Duc Dung Nguyen, PhD.

Contact: nddung@hcmut.edu.vn

Faculty of Computer Science and Engineering
Hochiminh city University of Technology

1. Sorting concepts

2. Insertion Sort

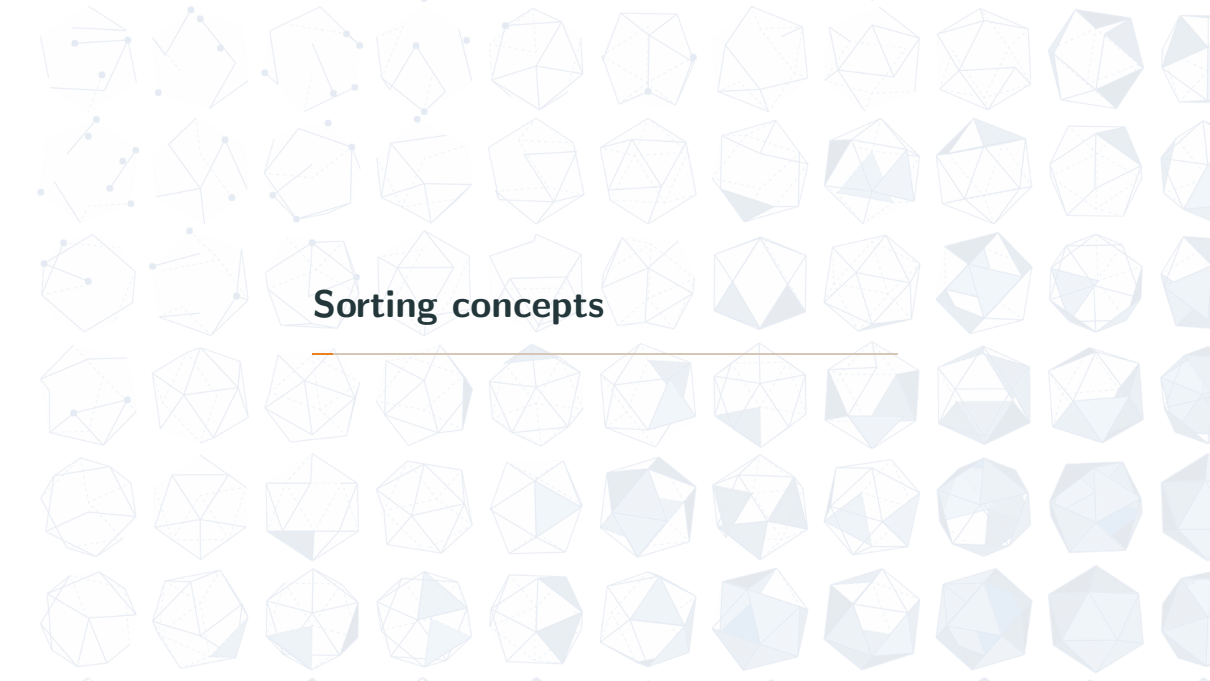
3. Selection Sort

4. Exchange Sort

5. Divide-and-Conquer

- **L.O.6.1** - Depict the working steps of sorting algorithms step-by-steps.
- **L.O.6.2** - Describe sorting algorithms by using pseudocode.
- **L.O.6.3** - Implement sorting algorithms using C/C++ .
- **L.O.6.4** - Analyze the complexity and develop experiment (program) to evaluate sorting algorithms.
- **L.O.6.5** - Use sorting algorithms for problems in real-life.
- **L.O.8.4** - Develop recursive implementations for methods supplied for the following structures: list, tree, heap, searching, and graphs.
- **L.O.1.2** - Analyze algorithms and use Big-O notation to characterize the computational complexity of algorithms composed by using the following control structures: sequence, branching, and iteration (not recursion).

Sorting concepts



- One of the most **important concepts** and **common applications** in computing.

23	78	45	8	32	56
----	----	----	---	----	----



8	23	32	45	56	78
---	----	----	----	----	----

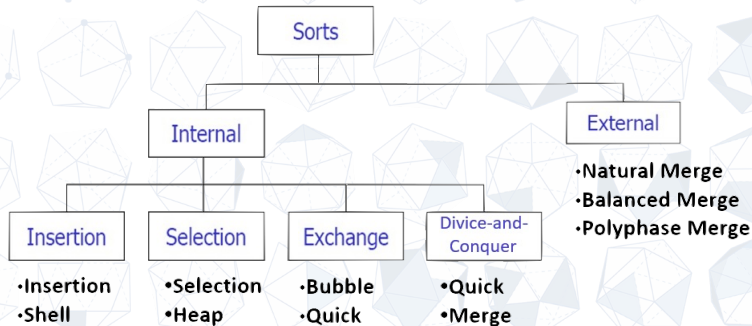
Sort stability: data with equal keys maintain their relative input order in the output.

78	8	45	8	32	56
----	---	----	---	----	----

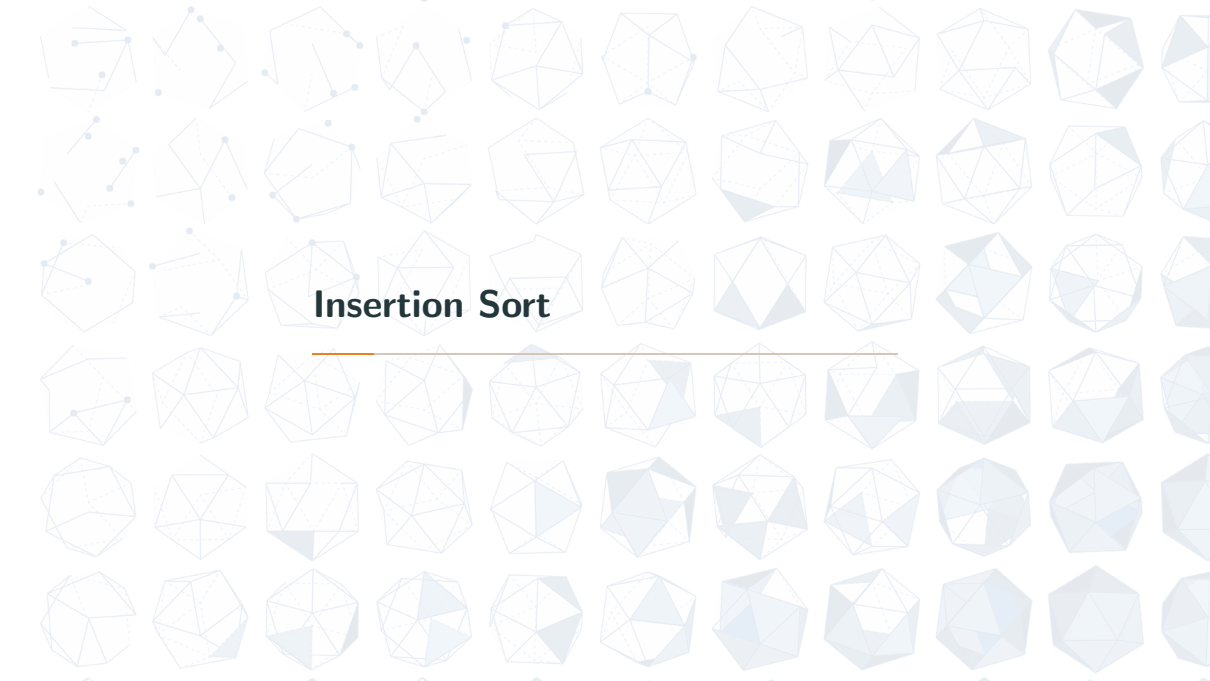


8	8	32	45	56	78
---	---	----	----	----	----

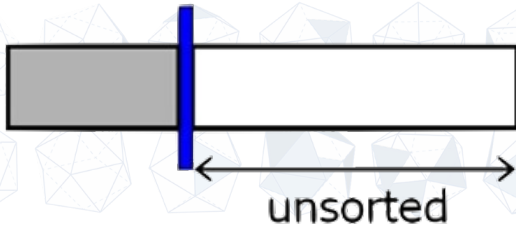
Sort efficiency: a measure of the relative efficiency of a sort = number of **comparisons** + number of **moves**.



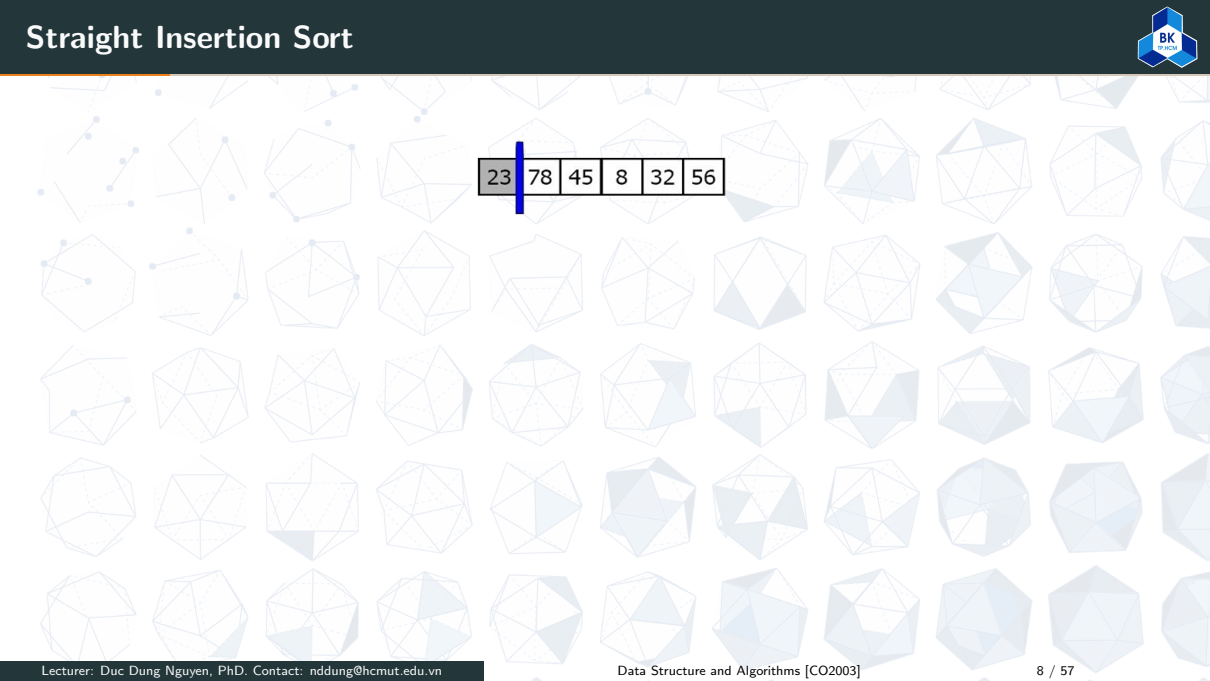
Insertion Sort



- The list is divided into two parts: **sorted** and **unsorted**.
- In each pass, the first element of the unsorted sublist is **inserted** into the sorted sublist.

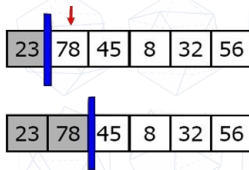


Straight Insertion Sort

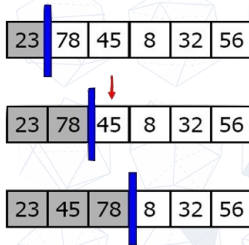
The background of the slide is a repeating pattern of various polyhedrons, including cubes, octahedrons, and dodecahedrons, in light blue and white.

23	78	45	8	32	56
----	----	----	---	----	----

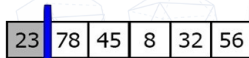
Straight Insertion Sort



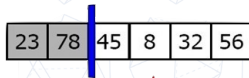
Straight Insertion Sort



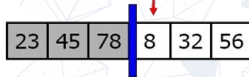
Straight Insertion Sort



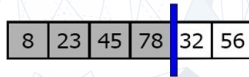
23	78	45	8	32	56
----	----	----	---	----	----



23	78	45	8	32	56
----	----	----	---	----	----




23	45	78	8	32	56
----	----	----	---	----	----




8	23	45	78	32	56
---	----	----	----	----	----


Straight Insertion Sort




23	78	45	8	32	56
----	----	----	---	----	----




23	78	45	8	32	56
----	----	----	---	----	----



23	45	78	8	32	56
----	----	----	---	----	----



8	23	45	78	32	56
---	----	----	----	----	----



8	23	32	45	78	56
---	----	----	----	----	----

Straight Insertion Sort

23 | 78 | 45 | 8 | 32 | 56

23 | 78 | 45 | 8 | 32 | 56

23 | 45 | 78 | 8 | 32 | 56

8 | 23 | 45 | 78 | 32 | 56

8 | 23 | 32 | 45 | 78 | 56

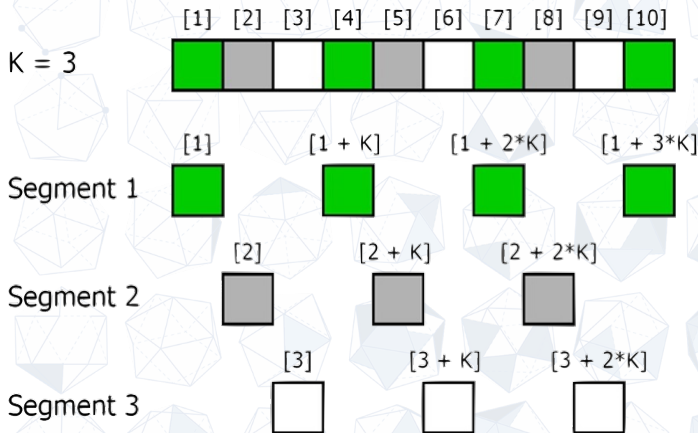
8 | 23 | 32 | 45 | 56 | 78

Algorithm InsertionSort()

Sorts the contiguous list using straight insertion sort.

```
if count > 1 then
  current = 1
  while current < count do
    temp = data[current]
    walker = current - 1
    while walker >= 0 AND temp.key < data[walker].key do
      data[walker+1] = data[walker]
      walker = walker - 1
    end
    data[walker+1] = temp
    current = current + 1
  end
end
```

- Named after its creator Donald L. Shell (1959).
- Given a list of N elements, the list is divided into K segments (K is called the **increment**).
- Each segment contains N/K or more elements.
- Segments are dispersed throughout the list.
- Also is called **diminishing-increment sort**.





- For the value of K in each iteration, sort the K segments.
- After each iteration, K is reduced until it is 1 in the final iteration.

Example of Shell Sort

Unsorted

Tim
Dot
Eva
Roy
Tom
Kim
Guy
Amy
Jon
Ann
Jim
Kay
Ron
Jan

Sublists incr. 5

Tim Dot
↓ ↓
Kim Guy
↓ ↓
Jim Kay
↓ ↓
Ron Jan

Eva Roy
↓ ↓
Guy Amy
↓ ↓
Jon Ann

5-Sorted

Jim Dot
↓ ↓
Kim Guy
↓ ↓
Tim Kay
↓ ↓
Ron Roy

Amy Jan
↓ ↓
Eva Jon
↓ ↓
Tom Ann

Recombined

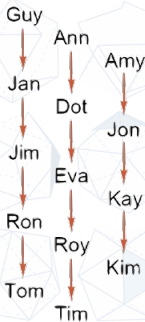
Jim
Dot
Amy
Jan
Ann
Kim
Guy
Eva
Jon
Tom
Tim
Kay
Ron
Roy

Example of Shell Sort

Sublists incr. 3



3-Sorted



List incr. 1



Sorted



- From more of the comparisons, it is better when we can receive more new information.
- Incremental values should not be multiples of each other, other wise, the same keys compared on one pass would be compared again at the next.
- The final incremental value must be 1.

- Incremental values may be:

1, 4, 13, 40, 121, ...

$$k_t = 1$$

$$k_{i-1} = 3 * k_i + 1$$

$$t = \lceil \log_3 n \rceil - 1$$

- or:

1, 3, 7, 15, 31, ...

$$k_t = 1$$

$$k_{i-1} = 2 * k_i + 1$$

$$t = \lceil \log_2 n \rceil - 1$$

Algorithm ShellSort()

Sorts the contiguous list using Shell sort.

```
k = first_incremental_value
while k >= 1 do
  segment = 1
  while segment <= k do
    SortSegment(segment)
    segment = segment + 1
  end
  k = next_incremental_value
end
End ShellSort
```

Algorithm SortSegment(val segment <int>, val k <int>)

Sorts the segment beginning at segment using insertion sort, step between elements in the segment is k.

```
current = segment + k
```

```
while current < count do
```

```
    temp = data[current]
```

```
    walker = current - k
```

```
    while walker >= 0 AND temp.key < data[walker].key do
```

```
        data[walker + k] = data[walker]
```

```
        walker = walker - k
```

```
    end
```

```
    data[walker + k] = temp
```

```
    current = current + k
```

```
end
```

```
End SortSegment
```

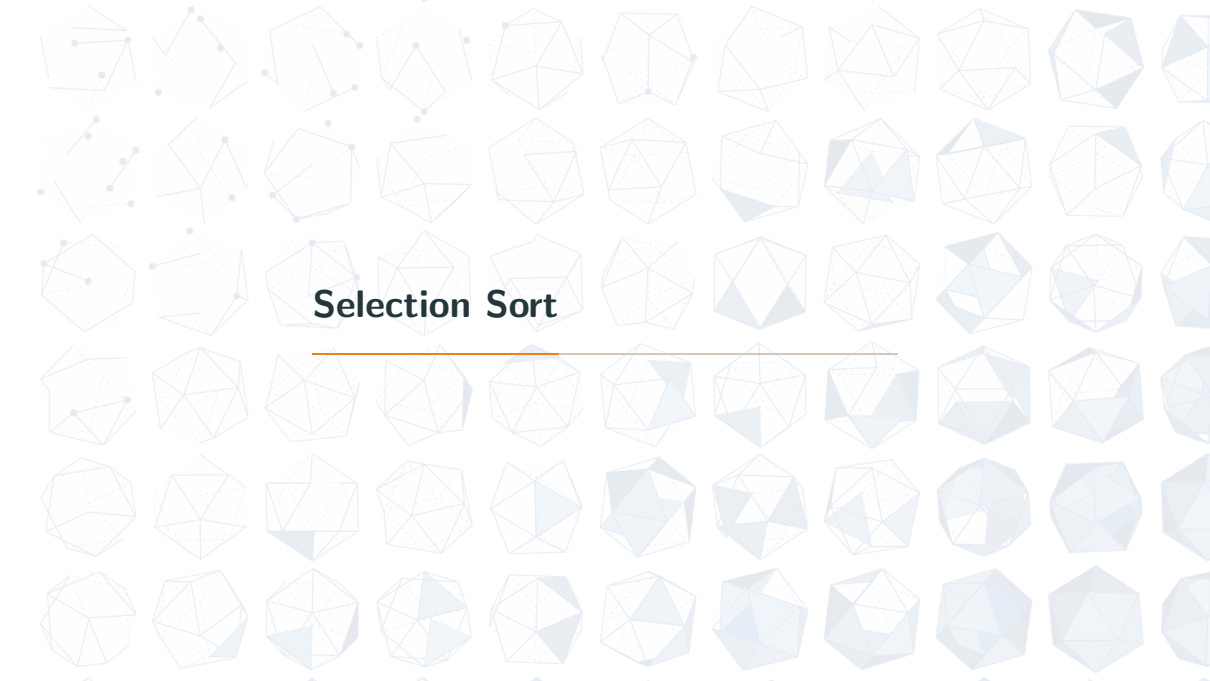
- Straight insertion sort:

$$f(n) = n(n+1)/2 = O(n^2)$$

- Shell sort:

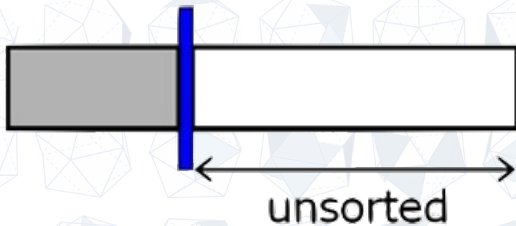
$$O(n^{1.25}) \text{ (Empirical study)}$$

Selection Sort



In each pass, the smallest/largest item is **selected** and placed in a sorted list.

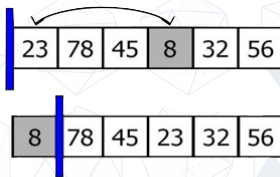
- The list is divided into two parts: **sorted** and **unsorted**.
- In each pass, in the unsorted sublist, the smallest element is **selected** and **exchanged** with the first element.



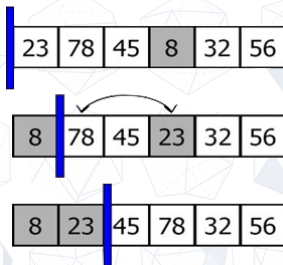
Straight Selection Sort



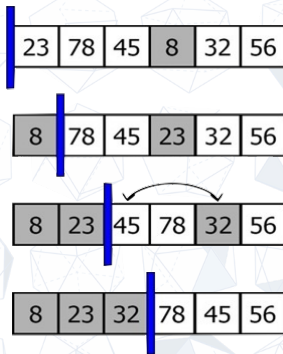
23	78	45	8	32	56
----	----	----	---	----	----



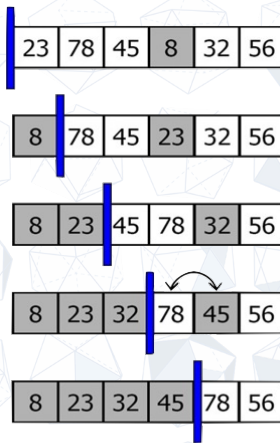
Straight Selection Sort



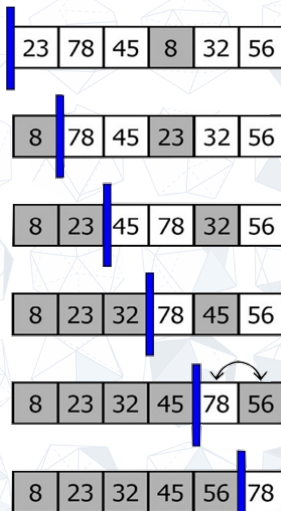
Straight Selection Sort



Straight Selection Sort



Straight Selection Sort



Algorithm SelectionSort()

Sorts the contiguous list using straight selection sort.

```
current = 0
```

```
while current < count - 1 do
```

```
    smallest = current
```

```
    walker = current + 1
```

```
    while walker < count do
```

```
        if data [walker].key < data [smallest].key then
```

```
            | smallest = walker
```

```
        end
```

```
        walker = walker + 1
```

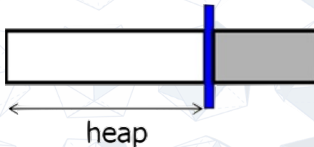
```
    end
```

```
    swap(current, smallest)
```

```
    current = current + 1
```

```
end
```

- The unsorted sublist is organized into a **heap**.
- In each pass, in the unsorted sublist, the largest element is **selected** and **exchanged** with the last element.
- The the heap is **reheaped**.



Heap Sort

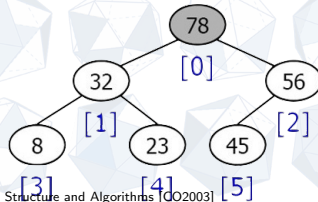
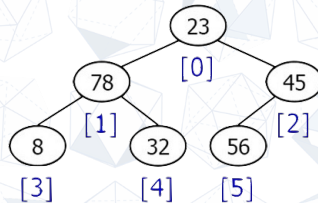
23	78	45	8	32	56
----	----	----	---	----	----

23	78	56	8	32	45
----	----	----	---	----	----

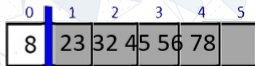
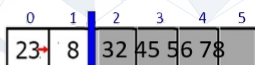
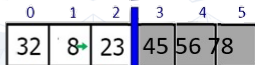
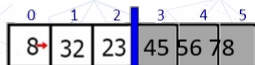
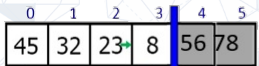
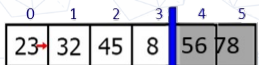
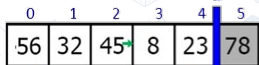
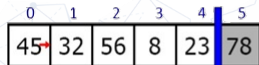
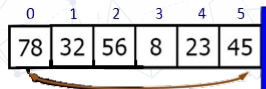
23	78	56	8	32	45
----	----	----	---	----	----

78	23	56	8	32	45
----	----	----	---	----	----

78	32	56	8	23	45
----	----	----	---	----	----



Heap Sort



Algorithm HeapSort()

Sorts the contiguous list using heap sort.

position = count/2 - 1

while position >= 0 **do**

 ReheapDown(position, count - 1)

 position = position - 1

end

last = count - 1

while last > 0 **do**

 swap(0, last)

 last = last - 1

 ReheapDown(0, last - 1)

end

End HeapSort

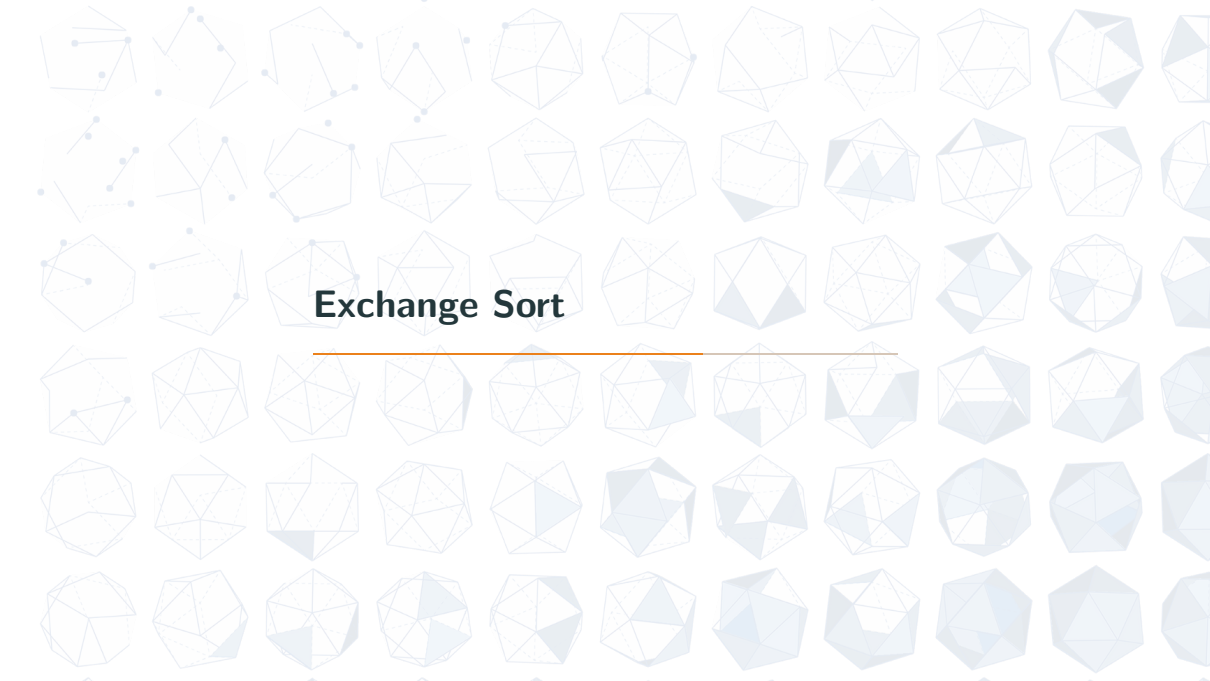
- Straight selection sort:

$$O(n^2)$$

- Heap sort:

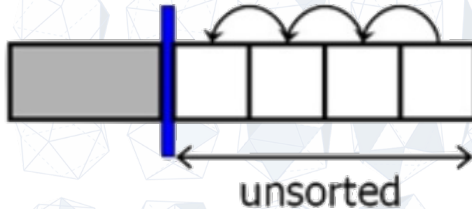
$$O(n \log_2 n)$$

Exchange Sort

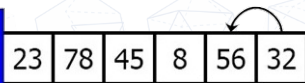


- In each pass, elements that are out of order are **exchanged**, until the entire list is sorted.
- **Exchange** is extensively used.

- The list is divided into two parts: **sorted** and **unsorted**.
- In each pass, the smallest element is **bubbled** from the unsorted sublist and moved to the sorted sublist.



23 78 45 8 56 32



23 78 45 8 32 56



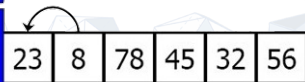
23 78 45 8 32 56

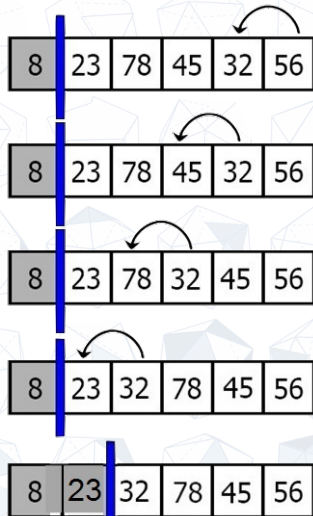


23 78 8 45 32 56



23 8 78 45 32 56





Algorithm BubbleSort()

Sorts the contiguous list using bubble sort.

current = 0, flag = False

while *current* < *count* **AND** *flag* = *False* **do**

 walker = count - 1

 flag = True

while *walker* > *current* **do**

if *data* [*walker*].*key* < *data* [*walker*-1].*key* **then**

 flag = False

 swap(*walker*, *walker* - 1)

end

walker = *walker* - 1

end

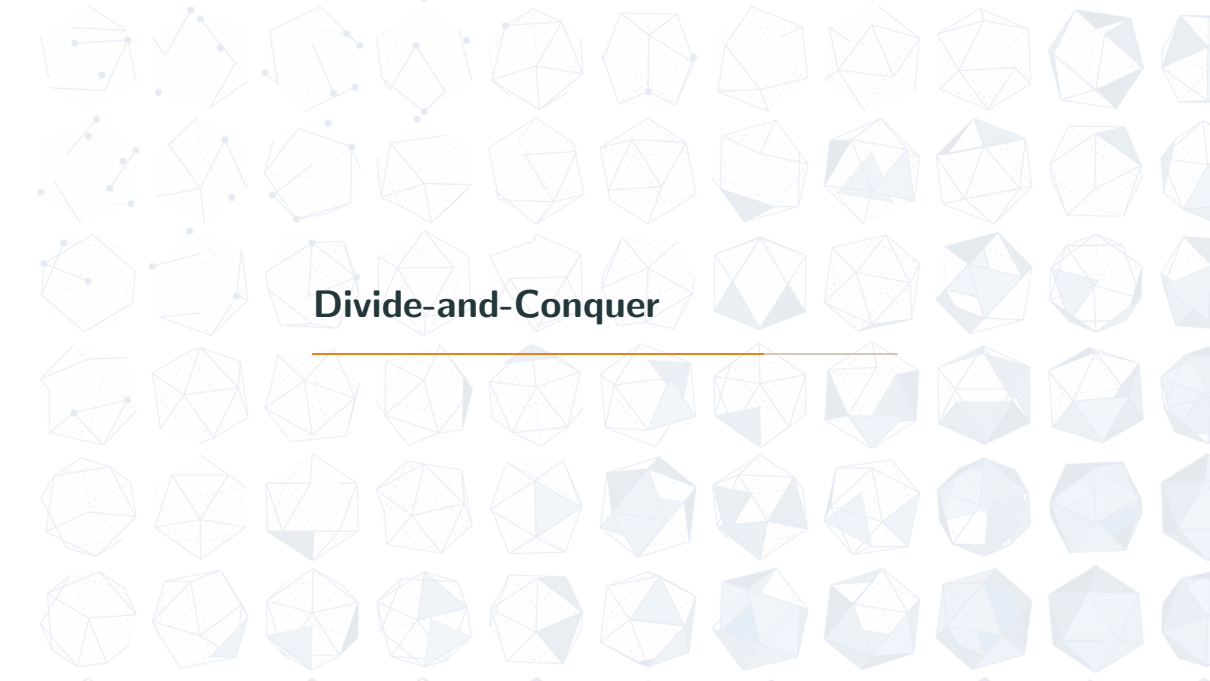
current = *current* + 1

end

- Bubble sort:

$$f(n) = n(n+1)/2 = O(n^2)$$

Divide-and-Conquer



Algorithm DivideAndConquer()
if *the list has length* > 1 **then**
 partition the list into lowlist and highlist
 lowlist.DivideAndConquer()
 highlist.DivideAndConquer()
 combine(lowlist, highlist)
end
End DivideAndConquer

Divide-and-Conquer Sort



	Partition	Combine
Merge Sort	easy	hard
Quick Sort	hard	easy

Algorithm QuickSort()

Sorts the contiguous list using quick sort.

recursiveQuickSort(0, count - 1)

End QuickSort

Algorithm recursiveQuickSort(val left <int>, val right <int>)

Sorts the contiguous list using quick sort.

Pre: left and right are valid positions in the list

Post: list sorted

if left < right **then**

 pivot_position = Partition(left, right)

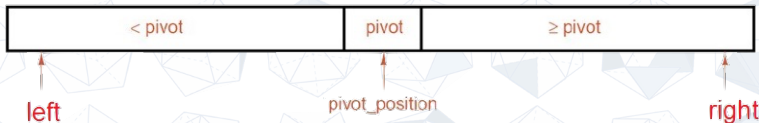
 recursiveQuickSort(left, pivot_position - 1)

 recursiveQuickSort(pivot_position + 1, right)

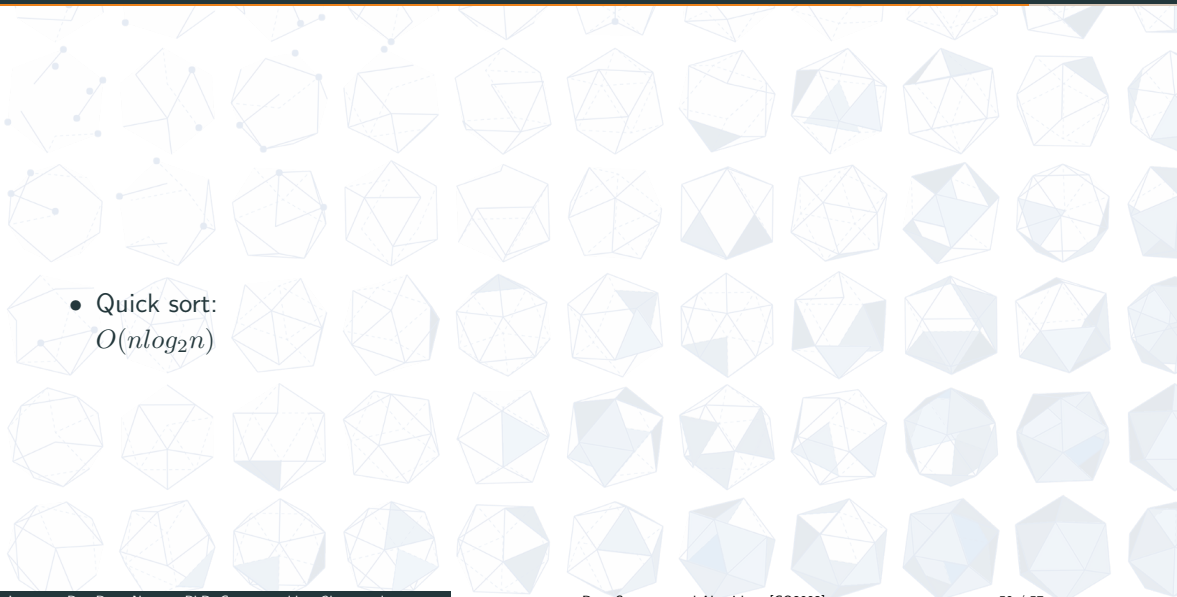
end

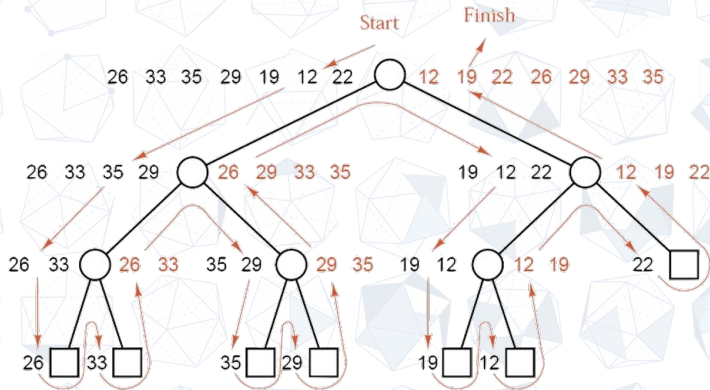
End recursiveQuickSort

Given a pivot value, the partition rearranges the entries in the list as the following figure:



- Quick sort:
 $O(n \log_2 n)$





Algorithm MergeSort()

Sorts the linked list using merge sort.

recursiveMergeSort(head)

End MergeSort

Algorithm recursiveMergeSort(ref sublist <pointer>)

Sorts the linked list using recursive merge sort.

if *sublist is not NULL AND sublist->link is not NULL* **then**

 Divide(sublist, second_list)

 recursiveMergeSort(sublist)

 recursiveMergeSort(second_list)

 Merge(sublist, second_list)

end

End recursiveMergeSort

Algorithm Divide(val sublist <pointer>, ref second_list <pointer>)

Divides the list into two halves.

midpoint = sublist

position = sublist->link

while *position is not NULL* **do**

 position = position->link

if *position is not NULL* **then**

 midpoint = midpoint->link

 position = position->link

end

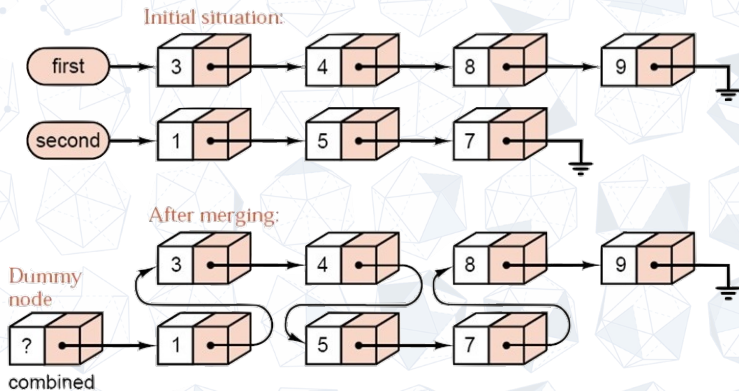
end

second_list = midpoint->link

midpoint->link = NULL

End Divide

Merge two sublists



Algorithm Merge(ref first <pointer>, ref second <pointer>)

Merges two sorted lists into a sorted list.

lastSorted = address of combined

while *first is not NULL AND second is not NULL* **do**

if *first->data.key <= second->data.key* **then**

 lastSorted->link = first

 lastSorted = first

 first = first->link

else

 lastSorted->link = second

 lastSorted = second

 second = second->link

end

end

Merge two sublists

// ...

if *first* is *NULL* **then**

 lastSorted->link = second

 second = *NULL*

else

 lastSorted->link = first

end

first = combined.link

End Merge