COMP 2140 - Data Structures

Shahin Kamali

Topic 5 - Sorting
University of Manitoba

Based on notes by S. Durocher.
Overview

- Review: Insertion Sort
- Merge Sort
- Quicksort
- Heapsort
- Counting Sort

For further reading, refer to Open Data Structures Book (Chapter 11)
Sorting

- **Input:**
  - a sequence of \( n \) objects: \( A[0], \ldots, A[n - 1] \) (typically an array or a linked list)
  - a comparison predicate, \( \leq \), that defines a total order on \( A \)

- **Output:**
  - an ordered representation of the objects in \( A \)
Sorting

**Input:**
- a sequence of \( n \) objects: \( A[0], \ldots, A[n-1] \) (typically an array or a linked list)
- a comparison predicate, \( \leq \), that defines a total order on \( A \)

**Output:**
- an ordered representation of the objects in \( A \)

Many sorting algorithms exist:
- bubble sort, insertion sort, merge sort, heapsort, radix sort, bucket sort, quicksort, etc.
Insertion Sort

- Go through the items in the array (list) one by one
- For each item \( x \) at index \( i \):
  - We know the sub-array \( A[0] \ldots A[i-1] \) is sorted
  - **Insert** \( x \) in its correct position in the sub-array \( A[i] \ldots A[i] \).

\[
\begin{align*}
4 & \quad 3 & \quad 2 & \quad 10 & \quad 12 & \quad 1 & \quad 5 & \quad 6
\end{align*}
\]
Insertion Sort

Go through the items in the array (list) one by one

For each item $x$ at index $i$:

- We know the sub-array $A[0] \ldots A[i-1]$ is sorted
- **Insert** $x$ in its correct position in the sub-array $A[i] \ldots A[i]$. 

```
 4 3 2 10 12 1 5 6
4 3 2 10 12 1 5 6
```
**Insertion Sort**

- Go through the items in the array (list) one by one
- For each item $x$ at index $i$:
  - We know the sub-array $A[0] \ldots A[i-1]$ is sorted
  - **Insert** $x$ in its correct position in the sub-array $A[i] \ldots A[i]$. 
Insertion Sort

- Go through the items in the array (list) one by one
- For each item $x$ at index $i$:
  - We know the sub-array $A[0] \ldots A[i - 1]$ is sorted
  - Insert $x$ in its correct position in the sub-array $A[i] \ldots A[i]$. 

![Insertion Sort Diagram]

```plaintext
<table>
<thead>
<tr>
<th>4</th>
<th>3</th>
<th>2</th>
<th>10</th>
<th>12</th>
<th>1</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>3</td>
<td>2</td>
<td>10</td>
<td>12</td>
<td>1</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>2</td>
<td>10</td>
<td>12</td>
<td>1</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>4</td>
<td>10</td>
<td>12</td>
<td>1</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>
```
Insertion Sort

- Go through the items in the array (list) one by one
- For each item \( x \) at index \( i \):
  - We know the sub-array \( A[0] \ldots A[i-1] \) is sorted
  - **Insert** \( x \) in its correct position in the sub-array \( A[i] \ldots A[i] \).
Insertion Sort

- Go through the items in the array (list) one by one
- For each item $x$ at index $i$:
  - We know the sub-array $A[0] \ldots A[i - 1]$ is sorted
  - **Insert** $x$ in its correct position in the sub-array $A[i] \ldots A[i]$. 

![Insertion Sort Diagram]
Insertion Sort

- Go through the items in the array (list) one by one
- For each item \( x \) at index \( i \):
  - We know the sub-array \( A[0] \ldots A[i - 1] \) is sorted
  - **Insert** \( x \) in its correct position in the sub-array \( A[i] \ldots A[i] \).
Insertion Sort

- Go through the items in the array (list) one by one
- For each item $x$ at index $i$:
  - We know the sub-array $A[0] \ldots A[i-1]$ is sorted
  - **Insert** $x$ in its correct position in the sub-array $A[i] \ldots A[i]$. 

![Insertion Sort Diagram](image)
Insertion Sort

- Go through the items in the array (list) one by one
- For each item $x$ at index $i$:
  - We know the sub-array $A[0] \ldots A[i-1]$ is sorted
  - Insert $x$ in its correct position in the sub-array $A[i] \ldots A[i]$. 

![Insertion Sort Diagram]
/* Function to sort an array using insertion sort */
void insertionSort(int arr[], int n)
{
    int i, key, j;
    for (i = 1; i < n; i++)
    {
        key = arr[i];
        j = i - 1;

        /* Move elements of arr[0..i-1], that are greater than key, to one position ahead of their current position */
        while (j >= 0 && arr[j] > key)
        {
            arr[j+1] = arr[j];
            j = j - 1;
        }
        arr[j+1] = key;
    }
}
Insertion Sort Summary

- **One Iteration of the Insertion Sort Algorithm:**
  - After the $i$th iteration, $A[0..i]$ is sorted.
  - Insert item $A[i + 1]$ in its proper place in $A[0..i]$. 

![Diagram](image)
Insertion Sort Summary

- One Iteration of the Insertion Sort Algorithm:
  - After the $i$th iteration, $A[0..i]$ is sorted.
  - Insert item $A[i+1]$ in its proper place in $A[0..i]$.
  - In the worst case, $i$ items are moved in the $i + 1$th iteration!

\[
\begin{array}{c}
\text{sorted} \\
0 \quad i \quad i+1 \\
\leq A[i+1] \quad > A[i+1] \quad A[i+1] \\
\end{array}
\]
Insertion Sort Analysis

- In the worst case the array is sorted backwards.

```
<table>
<thead>
<tr>
<th>n</th>
<th>n-1</th>
<th>n-2</th>
<th>...</th>
<th>3</th>
<th>2</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>5</td>
<td>...</td>
<td>n</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>4</td>
<td>...</td>
<td>n-1</td>
<td>n</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>...</td>
<td>n-2</td>
<td>n-1</td>
<td>n</td>
</tr>
</tbody>
</table>
```
Insertion Sort Analysis

- In the worst case the array is sorted backwards.

\[
\begin{array}{cccccc}
 n & n-1 & n-2 & \ldots & 3 & 2 & 1 \\
 \hline
 \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\
 3 & 4 & 5 & \ldots & n & 2 & 1 \\
 2 & 3 & 4 & \ldots & n-1 & n & 1 \\
 1 & 2 & 3 & \ldots & n-2 & n-1 & n
\end{array}
\]

- The total number of moved items:

\[
1 + 2 + \ldots + n - 1 = n(n - 1)/2 \in (n^2)
\]
The worst-case running time of insertion sort is $O(n^2)$. 
The **worst-case** running time of insertion sort is $O(n^2)$.

As it turns out, the **average-case** running time is also $O(n^2)$.

Faster sorting algorithms exist. These include:

<table>
<thead>
<tr>
<th></th>
<th>worst case</th>
<th>average case</th>
</tr>
</thead>
<tbody>
<tr>
<td>Quicksort</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Merge Sort</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Heapsort</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The **worst-case** running time of insertion sort is $O(n^2)$.

As it turns out, the **average-case** running time is also $O(n^2)$.

Faster sorting algorithms exist. These include:

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>worst case</th>
<th>average case</th>
</tr>
</thead>
<tbody>
<tr>
<td>Quicksort</td>
<td>$O(n^2)$ (random pivot)</td>
<td>$O(n \log n)$</td>
</tr>
<tr>
<td>Merge Sort</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
</tr>
<tr>
<td>Heapsort</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
</tr>
</tbody>
</table>
Insertion Sort Time Complexity

- The **worst-case** running time of insertion sort is $O(n^2)$.
- As it turns out, the **average-case** running time is also $O(n^2)$.
- Faster sorting algorithms exist. These include:

<table>
<thead>
<tr>
<th></th>
<th>worst case</th>
<th>average case</th>
</tr>
</thead>
<tbody>
<tr>
<td>Quicksort</td>
<td>$O(n^2)$ (random pivot)</td>
<td>$O(n \log n)$</td>
</tr>
<tr>
<td>Merge Sort</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
</tr>
<tr>
<td>Heapsort</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
</tr>
</tbody>
</table>

- The lower bound on the worst-case time complexity of any comparison-based sorting algorithm is also $n \log n$. 
Merge Sort

- Merge sort is an example of a **divide-and-conquer** algorithm.
  - **Divide** the input into two or more disjoint subsets.
  - Recursively solve each sub-problem.
  - Combine solutions to the sub-problems to give the solution to the original problem.
**Merge Sort Algorithm**

- **Input:** an array \( A[0..n-1] \) of comparable elements.
  - **Divide** \( A \) into two subarrays \( A[0..\lfloor n/2 \rfloor] \) and \( A[\lfloor n/2 \rfloor + 1, n-1] \)
  - **Recursively** sort each sub-array
  - **Combine** the two subarray via merging them

In practice, when the length of sub-array is less than 100, selection sort is applied.
Merge Sort Algorithm

- **Input:** an array $A[0..n-1]$ of comparable elements.
  - Divide $A$ into two subarrays $A[0..\lfloor n/2 \rfloor]$ and $A[\lceil n/2 \rceil + 1, n-1]$
  - Recursively sort each sub-array
  - Combine the two subarray via merging them

- The base of recursion is an array of size 1 which is sorted
  - In practice, when the length of sub-array is less than 100, selection sort is applied.
Merge Sort Scheme
Merge Sort Scheme
Merge Sort Scheme

sort recursively
Merge Sort Scheme

sort recursively
Merge Sort Scheme

sort recursively

sort recursive
Merge Sort Scheme

sort recursively

merge

sort recursive
Merge Sort Scheme

sort recursively

merge

sort recursive
Merge Sort Example

MS(A[0..7])

\[ 13 \ -4 \ 7 \ 5 \ 6 \ 9 \ 2 \ 1 \]
Merge Sort Example

MS(A[0..7])

\[
\begin{array}{ccccccc}
13 & -4 & 7 & 5 & 6 & 9 & 2 & 1 \\
\end{array}
\]

MS(A[0..3])

\[
\begin{array}{cccc}
13 & -4 & 7 & 5 \\
\end{array}
\]
Merge Sort Example

13 -4 7 5 6 9 2 1

13 -4 7 5

13 -4
Merge Sort Example

```
| Call | | return |
|------| |--------|
```

```
MS(A[0..7]) | 13 | -4 | 7 | 5 | 6 | 9 | 2 | 1 |
|------------|----|----|----|----|----|----|----|----|
MS(A[0..3]) | 13 | -4 | 7 | 5 |    |    |    |    |
|------------|----|----|----|----|----|----|----|----|
MS(A[0..1]) | 13 | -4 |    |    |    |    |    |    |
|------------|----|----|----|----|----|----|----|----|
MS(A[0..0]) | 13 |    |    |    |    |    |    |    |
```

```
Merge Sort Example

Call

return
Merge Sort Example

MS(A[0..7])

13 -4 7 5 6 9 2 1

MS(A[0..3])

13 -4 7 5

MS(A[0..1])

13 -4

MS(A[0..0])

13

MS(A[0..1])

-4 13

Call

return
Merge Sort Example

13  -4  7  5  6  9  2  1

13  -4  7  5

13  -4

13  -4

Call

return
Merge Sort Example

The image illustrates the merge sort algorithm visually, showing the division and merging of arrays to sort them. The process is depicted with a tree-like structure, where each node represents a sub-array and the edges indicate the merge operations.
Merge Sort Example

13  -4  7  5  6  9  2  1

MS(A[0..7])

13  -4  7  5

MS(A[0..3])

13
-4

MS(A[0..1])

13
-4

MS(A[0..0])

-4
13

MS(A[0..1])

7

MS(A[1..1])

7

MS(A[2..2])

5

MS(A[2..3])

13  -4  7  5

MS(A[0..3])

Call
return
Merge Sort Example

```
13 -4 7 5 6 9 2 1
```

```
13 -4 7 5
```

```
13 -4
```

```
13
```

```
-4 13
```

```
-4
```

```
7
```

```
7 5
```

```
7
```

```
5
```

```
Merge Sort Example
```

```
Call
```

```
return
```
Merge Sort Example
Merge Sort Example

```plaintext
13  -4  7  5  6  9  2  1
```

```
13  -4  7  5
13  -4
13  13
-4  13
-4  13
-4  13
-4  13

13  -4
13  -4
13  13
-4  13
-4  13
-4  13
-4  13

7  5
7  5
7  5
7  5
7  5
7  5
7  5
7  5

5  7
5  7
5  7
5  7
5  7
5  7
5  7
5  7

-4  5  7  13
-4  5  7  13
-4  5  7  13
-4  5  7  13
-4  5  7  13
-4  5  7  13
-4  5  7  13
-4  5  7  13
```

Call return
Merge Sort Example

```
13  -4  7  5  6  9  2  1
```

```
13  -4  7  5
```

```
13  -4
```

```
13  13
```

```
-4  13
```

```
-4  13
```

```
-4  5  7  13
```

```
-4  5  7  13
```

Call

return
Merge Sort Example

MS(A[0..7])

MS(A[0..3])

MS(A[0..1])

MS(A[0..0])

MS(A[0..1])

MS(A[1..1])

MS(A[2..2])

MS(A[2..3])

MS(A[3..3])

MS(A[0..3])

MS(A[0..7])

Call

return
Merge Sort Example
Merge Sort Example

```
13  -4  7  5  6  9  2  1
  /    /    /    /    /    /
 MS(A[0..7]) MS(A[0..6]) MS(A[2..3]) MS(A[3..4]) MS(A[4..5])
  /    /    /    /    /    /
 MS(A[0..5]) MS(A[0..4]) MS(A[1..3]) MS(A[2..2]) MS(A[3..1])
  /    /    /    /    /    /
 MS(A[0..3]) MS(A[0..2]) MS(A[0..1]) MS(A[0..1]) MS(A[0..0])
  /    /    /    /    /    /
 MS(A[0..1]) MS(A[0..0]) MS(A[0..0]) MS(A[0..0])
  /    /    /    /    /    /
 MS(A[0..0]) MS(A[0..0]) MS(A[0..0])
```

Call
return
Merge Sort Example
Merge Sort Example

```
MS(A[0..7])

13  -4  7  5  6  9  2  1

MS(A[0..3])

13  -4  7  5

MS(A[0..1])

13  -4

MS(A[0..0])

13

-4  13

MS(A[0..1])

-4

MS(A[0..0])

-4

MS(A[0..3])

-4  5  7  13

MS(A[4..7])

6  9  2  1

MS(A[4..5])

6  9

MS(A[4..4])

6

MS(A[5..5])

9

MS(A[2..3])

7  5

MS(A[2..2])

7

MS(A[2..1])

7

MS(A[2..0])

7

MS(A[3..3])

5  7

MS(A[3..2])

5

MS(A[3..1])

5

MS(A[3..0])

5

MS(A[4..7])

6  9

MS(A[4..6])

6

MS(A[5..6])

9

MS(A[6..7])

2  1

MS(A[6..5])

2

MS(A[6..4])

2

MS(A[6..3])

2

MS(A[6..2])

2

MS(A[6..1])

2

MS(A[6..0])

2

Call

return
```
Merge Sort Example
Merge Sort Example

```
| 13 | -4 | 7 | 5 | 6 | 9 | 2 | 1 |
|------------------|
| 13 | -4 | 7 | 5 |
| 13 | -4 |
| 13 |
| -4 | 13 |
| -4 | 13 |
| 5 | 7 |
| 5 | 7 |
| 6 | 9 |
| 6 | 9 |
| 2 | 1 |
| 2 | 1 |
| 2 | 1 |
```

Call

return
Merge Sort Example

```
13  -4  7  5  6  9  2  1
```

```
13  -4  7  5
```

```
13  -4
```

```
13
```

```
-4  13
```

```
-4  13
```

```
-4  5  7  13
```

```
-4  5  7  13
```

```
-4  5  7  13
```

```
-4  5  7  13
```

```
-4  5  7  13
```

```
-4  5  7  13
```

```
-4  5  7  13
```

```
-4  5  7  13
```

```
-4  5  7  13
```

```
-4  5  7  13
```

```
-4  5  7  13
```

```
-4  5  7  13
```

```
-4  5  7  13
```

```
-4  5  7  13
```

```
-4  5  7  13
```

```
-4  5  7  13
```

```
-4  5  7  13
```

```
-4  5  7  13
```

```
-4  5  7  13
```

```
-4  5  7  13
```

```
-4  5  7  13
```

```
-4  5  7  13
```

```
-4  5  7  13
```

```
-4  5  7  13
```

```
-4  5  7  13
```

```
-4  5  7  13
```

```
-4  5  7  13
```

```
-4  5  7  13
```

```
-4  5  7  13
```

```
-4  5  7  13
```

```
-4  5  7  13
```

```
-4  5  7  13
```

```
-4  5  7  13
```

```
-4  5  7  13
```

```
-4  5  7  13
```

```
-4  5  7  13
```

```
-4  5  7  13
```

```
-4  5  7  13
```

```
-4  5  7  13
```

```
-4  5  7  13
```

```
-4  5  7  13
```

```
-4  5  7  13
```

```
-4  5  7  13
```

```
-4  5  7  13
```

```
-4  5  7  13
```

```
-4  5  7  13
```

```
-4  5  7  13
```

```
-4  5  7  13
```

```
-4  5  7  13
```

```
-4  5  7  13
```

```
-4  5  7  13
```

```
-4  5  7  13
```

```
-4  5  7  13
```

```
-4  5  7  13
```

```
-4  5  7  13
```

```
-4  5  7  13
```

```
-4  5  7  13
```

```
-4  5  7  13
```

```
-4  5  7  13
```

```
-4  5  7  13
```

```
-4  5  7  13
```

```
-4  5  7  13
```

```
-4  5  7  13
```

```
-4  5  7  13
```

```
-4  5  7  13
```

```
-4  5  7  13
```

```
-4  5  7  13
```

```
-4  5  7  13
```

```
-4  5  7  13
```

```
-4  5  7  13
```

```
-4  5  7  13
```

```
-4  5  7  13
```

```
-4  5  7  13
```

```
-4  5  7  13
```

```
-4  5  7  13
```

```
-4  5  7  13
```

```
-4  5  7  13
```

```
-4  5  7  13
```

```
-4  5  7  13
```

```
-4  5  7  13
```

```
-4  5  7  13
```

```
-4  5  7  13
```

```
-4  5  7  13
```

```
-4  5  7  13
```

```
-4  5  7  13
```

```
-4  5  7  13
```

```
-4  5  7  13
```

```
-4  5  7  13
```

```
-4  5  7  13
```

```
-4  5  7  13
```

```
-4  5  7  13
```

```
-4  5  7  13
```

```
-4  5  7  13
```

```
-4  5  7  13
```

```
-4  5  7  13
```

```
-4  5  7  13
```

```
-4  5  7  13
```

```
-4  5  7  13
```

```
-4  5  7  13
```

```
-4  5  7  13
```

```
-4  5  7  13
```

```
-4  5  7  13
```

```
-4  5  7  13
```

```
-4  5  7  13
```

```
-4  5  7  13
```

```
-4  5  7  13
```

```
-4  5  7  13
```

```
-4  5  7  13
```
Merge Sort Example
Merge Sort Example
Merge Sort Example
Merge Sort Example

![Merge Sort Tree Example](image-url)
Given two sorted arrays $A$ and $B$ of size $n$ and $m$, merge them into array $C$ of size $m + n$.

- $i$, $j$, and $k$ are three indices moving on $A$, $B$, and $C$.
- They are initially 0.

At each step, copy the smaller of $A[i]$ and $B[j]$ to $C[k]$.

- Increment $k$
  - If $A[i]$ is copied, increment $i$; otherwise increment $j$.

If one array ends ($i = n$ or $j = m$), copy the remaining items of the other array to $C$. 

---

**Merging Sorted Sub-arrays**
Merge Example

length $n$  A \[\begin{array}{c}
\text{i} \\
\text{j} \\
\end{array}\]

-5 -1 10 21 25 30

length $m$  B

-3 -2 0 15 50

length $n + m$  C


Merge Example

length $n$

$$a_i = [-5, -1, 10, 21, 25, 30]$$

length $m$

$$b_j = [-3, -2, 0, 15, 50]$$

length $n + m$

$$c_k = [-5, \ldots]$$
### Merge Example

<table>
<thead>
<tr>
<th>length $n$</th>
<th>A</th>
<th>i</th>
</tr>
</thead>
<tbody>
<tr>
<td>-5 -1 10 21 25 30</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>length $m$</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>-3 -2 0 15 50</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>length $n + m$</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>-5</td>
<td></td>
</tr>
</tbody>
</table>
### Merge Example

<table>
<thead>
<tr>
<th>Length $n$</th>
<th>$A$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>-5</td>
</tr>
<tr>
<td>$i$</td>
<td></td>
</tr>
<tr>
<td>$j$</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Length $m$</th>
<th>$B$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>-3</td>
</tr>
<tr>
<td>$k$</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Length $n + m$</th>
<th>$C$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>-5</td>
</tr>
</tbody>
</table>
Merge Example

length $n$ A

\[\begin{array}{c}
-5 \quad -1 \quad 10 \quad 21 \quad 25 \quad 30 \\
\end{array}\]

length $m$ B

\[\begin{array}{c}
-3 \quad -2 \quad 0 \quad 15 \quad 50 \\
\end{array}\]

length $n + m$ C

\[\begin{array}{c}
-5 \quad -3 \quad -2 \\
\end{array}\]
Merge Example

length \( n \)  
A \( \overline{-5 -1 10 21 25 30} \)

length \( m \)  
B \( \overline{-3 -2 0 15 50} \)

length \( n + m \)  
C \( \overline{-5 -3 -2 -1 \cdots} \)
Merge Example

length \( n \)

A

\[-5 \quad -1 \quad 10 \quad 21 \quad 25 \quad 30\]

j

length \( m \)

B

\[-3 \quad -2 \quad 0 \quad 15 \quad 50\]

length \( n + m \)

C

\[-5 \quad -3 \quad -2 \quad -1 \quad 0\]

k
Merge Example

length \( n \)  
\[ \begin{array}{c}
A \quad -5 & -1 & 10 & 21 & 25 & 30 \\
\end{array} \]

length \( m \)  
\[ \begin{array}{c}
B \quad -3 & -2 & 0 & 15 & 50 \\
\end{array} \]

length \( n + m \)  
\[ \begin{array}{c}
C \quad -5 & -3 & -2 & -1 & 0 & 10 \\
\end{array} \]
Merge Example

Length $n$  
$A$: \[-5, -1, 10, 21, 25, 30\]

Length $m$  
$B$: \[-3, -2, 0, 15, 50\]

Length $n + m$  
$C$: \[-5, -3, -2, -1, 0, 10, 15\]
Merge Example

length $n$

| A | -5 | -1 | 10 | 21 | 25 | 30 |

length $m$

| B | -3 | -2 | 0  | 15 | 50 |

length $n + m$

| C | -5 | -3 | -2 | -1 | 0  | 10 | 15 | 21 |
Merge Example

Length $n$

| A   | -5 | -1 | 10 | 21 | 25 | 30 |

Length $m$

| B   | -3 | -2 | 0  | 15 | 50 |

Length $n + m$

| C   | -5 | -3 | -2 | -1 | 0  | 10 | 15 | 21 | 25 |
Merge Example

length $n$  A: \(-5, -1, 10, 21, 25, 30\)

length $m$  B: \(-3, -2, 0, 15, 50\)

length $n + m$  C: \(-5, -3, -2, -1, 0, 10, 15, 21, 25, 30\)
Merge Example

length $n$  

| A | -5 | -1 | 10 | 21 | 25 | 30 |

length $m$  

| B | -3 | -2 | 0  | 15 | 50 |

length $n + m$  

| C | -5 | -3 | -2 | -1 | 0  | 10 | 15 | 21 | 25 | 30 | 50 |
// Merge A[0..n] and B[0..m] into C[0..n+m-1]
void mergeArrays(int arr1[], int arr2[], int n, int m, int arr3[])
{
    int i = 0, j = 0, k = 0;
    while (i < n && j < m)
    {
        if (A[i] < B[j])
        {
            C[k] = A[i]; i++; k++;
        }
        else
        {
            C[k] = B[j]; j++; k++;
        }
    }
    while (i < n)
    {
        C[k] = A[i]; i++; k++;
    }
    while (j < m)
    {
        C[k] = B[j]; j++; k++;
    }
}
Merge Sort Summary

- Recursively sort the left half of the input array *A*
- Recursively sort the left half of the input array *B*
- Merge the two sub-arrays into a new one
  - note that merging requires a new array, that is, it cannot be done in place.
What is the time complexity of merge-sort?

\[ T(n) = \begin{cases} 
1, & n \leq 1 \\
2T(n/2) + O(n) & n \geq 2 
\end{cases} \]

We solve this using replacement method to get \( T(n) \in O(n \log n) \).
Analysis of Merge Sort

\[ T(n) = \begin{cases} 
1, & n \leq 1 \\
2T(n/2) + O(n) & n \geq 2 
\end{cases} \]

Since merging two subarrays of size \( n/2 \) takes \( O(n) \), the merge time is at most \( cn \) for some constant value of \( c \) (think of \( c \) as \( M \) in the definition of big-Oh). So we have:

\[
T(n) \leq 2T(n/2) + c \cdot n \\
\leq 2(2T(n/4) + c \cdot n/2) + c \cdot n = 4T(n/4) + 2cn \\
\leq 4(2T(n/8) + c \cdot n/4) + 2c \cdot n = 8T(n/8) + 3cn \\
\leq \ldots \\
\leq 2^k T\left(\frac{n}{2^k}\right) + kcn \\
\ldots \\
\leq 2^{\log n} T\left(\frac{n}{2^{\log n}}\right) + \log ncn = nT(1) + cn \log n \in O(n \log n).
\]
Should we use merge sort?

- Unlike insertion sort, merge sort does not work in place.
- Merging two arrays requires temporary storage.
Quick Sort

- Like insertion sort, quicksort works **in place**.
- Like merge sort, quicksort employs a divide and conquer strategy.
- Quicksort is usually implemented as a randomized algorithm.
Quick Sort

- Select an arbitrary element in the array as a **pivot**.
- Partition the array such that elements less than or equal to the pivot appear to its left and elements greater than or equal to the pivot appear to its right.
- Recursively sort each partition.
- In the base case we have an array of size 1.
Quick Sort

- Select an arbitrary element in the array as a **pivot**.
- Partition the array such that elements less than or equal to the pivot appear to its left and elements greater than or equal to the pivot appear to its right.
- Recursively sort each partition.
- In the base case we have an array of size 1.
Quick Sort

- Select an arbitrary element in the array as a **pivot**.
- Partition the array such that elements less than or equal to the pivot appear to its left and elements greater than or equal to the pivot appear to its right.
- Recursively sort each partition.
- In the base case we have an array of size 1.
Quick Sort

- Select an arbitrary element in the array as a **pivot**.
- Partition the array such that elements less than or equal to the pivot appear to its left and elements greater than or equal to the pivot appear to its right.
- Recursively sort each partition.
- In the base case we have an array of size 1.
Quick Sort

- Select an arbitrary element in the array as a **pivot**.
- Partition the array such that elements less than or equal to the pivot appear to its left and elements greater than or equal to the pivot appear to its right.
- Recursively sort each partition.
- In the base case we have an array of size 1.
Quick Sort

- Select an arbitrary element in the array as a pivot.
- Partition the array such that elements less than or equal to the pivot appear to its left and elements greater than or equal to the pivot appear to its right.
- Recursively sort each partition.
- In the base case we have an array of size 1.
Quick Sort

- Select an arbitrary element in the array as a **pivot**.
- Partition the array such that elements less than or equal to the pivot appear to its left and elements greater than or equal to the pivot appear to its right.
- Recursively sort each partition.
- In the base case we have an array of size 1.
Quick Sort

- Select an arbitrary element in the array as a **pivot**.
- Partition the array such that elements less than or equal to the pivot appear to its left and elements greater than or equal to the pivot appear to its right.
- Recursively sort each partition.
- In the base case we have an array of size 1.
Quick Sort

- Select an arbitrary element in the array as a **pivot**.
- Partition the array such that elements less than or equal to the pivot appear to its left and elements greater than or equal to the pivot appear to its right.
- Recursively sort each partition.
- In the base case we have an array of size 1.
Quick Sort

- Select an arbitrary element in the array as a pivot.
- Partition the array such that elements less than or equal to the pivot appear to its left and elements greater than or equal to the pivot appear to its right.
- Recursively sort each partition.
- In the base case we have an array of size 1.
Quick Sort

- Select an arbitrary element in the array as a pivot.
- Partition the array such that elements less than or equal to the pivot appear to its left and elements greater than or equal to the pivot appear to its right.
- Recursively sort each partition.
- In the base case we have an array of size 1.
Quick Sort

- Any element in the array can be selected as the pivot.
- Typically, the pivot is selected randomly.
- Elements are partitioned into those less than or equal to the pivot and those greater than the pivot.
- In general, elements within a partition are not initially sorted.
- **Partitioning can be performed in place.**
Partition Algorithm

First, swap pivot with the first element.

Store elements $\leq$ the pivot at the front of the array and elements $\geq$ the pivot at the back of the array.

1. Scan the array starting from the front until an element is found that is $> \text{ the pivot}$.
2. Scan the array starting from the back until an element is found that is $< \text{ the pivot}$.
3. Swap these two items.
4. Continue until the entire array has been partitioned.
// Quicksort Partition Java Code
public static int partition(int [] A, int lo, int hi, int pivot) {
    int finalPivot;
    int left = lo + 1;
    int right = hi;
    swap(A, lo, pivot);
    while (left < right) {
        while (left < right && A[lo] >= A[left])
            left++;
        while (left < right && A[lo] <= A[right])
            right--;
        swap(A, left, right);
    }
    if (A[right] <= A[lo])
        finalPivot = right;
    else
        finalPivot = right - 1;
    swap(A, lo, finalPivot);
}
Quicksort Example

- Sort the following array using quicksort.

```
6  -3  5  1  2  -4  3  7
```
Quick sort Example

- Sort the following array using quick sort.

\[
\begin{array}{cccccccc}
6 & -3 & 5 & 1 & 2 & -4 & 3 & 7 \\
\end{array}
\]

- **Step 1.** Select a pivot element and swap it with the first element.

\[
\begin{array}{cccccccc}
6 & -3 & 5 & 1 & 2 & -4 & 3 & 7 \\
\end{array}
\]

- Note that you could select any element as the pivot.
Quicksort Example

- We now partition the array such that elements to the left of the pivot are $\leq$ than the pivot and element to the right of the pivot are $\geq$ the pivot.
  - Initialize left to the leftmost element after the pivot and right to the rightmost element.

- **step 2:**
  Increment left until we find an element that is greater than the pivot.
Quicksort Example

- **Step 3:** Decrement right until we find an element that is less than the pivot.

- **Step 4:** Swap the elements at positions left and right.
Quicksort Example

- **Step 5:** Repeat until left $\geq$ right.

- **Step 6:** If the element at position right is less than the pivot, then swap it with the pivot. Otherwise, swap the element at position right $-1$ with the pivot.
Quicksort Example

- The pivot element is now in the correct position in the array.
- Elements to the left of the pivot are $\leq$ than it and elements to the right of the pivot are $\geq$ than it.
Quicksort Example

- The pivot element is now in the correct position in the array.
- Elements to the left of the pivot are \(\leq\) than it and elements to the right of the pivot are \(\geq\) than it.

\[
\begin{array}{cccccccc}
1 & -3 & -4 & 2 & 6 & 5 & 3 & 7 \\
\end{array}
\]

- Quicksort is called recursively on the left and right partitions.

\[
\begin{array}{cccccccc}
-4 & -3 & 1 & 2 & 3 & 5 & 6 & 7 \\
\end{array}
\]
The pivot element is now in the correct position in the array.

Elements to the left of the pivot are \( \leq \) than it and elements to the right of the pivot are \( \geq \) than it.

Quicksort is called recursively on the left and right partitions.

Once the left subarray is recursively sorted and the right subarray is recursively sorted, the entire array is sorted.

The base case is reached when the array has size \( \leq 1 \).