ST. CHARLES COMMUNITY COLLEGE

CPT-281 - Introduction to Data Structures with C++

Module 12

Sorting Algorithms

Dayu Wang

- **Sorting Algorithms**
  - → A **sorting algorithm** sorts the elements in an vector in **non-decreasing (increasing)** order.
  - → Some sorting algorithms use **comparison** to sort the vectors; others do **not** use comparison.
  - → In this class, you are only required to understand comparison sorting algorithms.
- **Comparison Sort**
  - → **Only** the **comparison result** can be used to sort the vector.

    For example, for two elements `a` and `b` in the input vector, you **cannot** use the magnitude of `a` or `b`, but you can use their **comparison result**, either `a < b`, `a > b`, or `a == b`.
  - → Comparison properties:
    - **Totalness**: for all `a` and `b`, either $a \le b$ or $a \ge b$.
    - **Transitivity**: if $a \le b$ and $b \le c$, then $a \le c$.

- **In-Place Sort**

  → **If a sorting algorithm can directly manipulate the elements in the input vector without making additional copies of the input vector, the sorting algorithm is in-place.**

  → **In-place sorting algorithms have space complexity of $O(1)$; they only use constant extra memory to sort the input vector.**

- **Stable Sort**

  → **If `a == b` in the input vector, theoretically, either `a` or `b` may appear before the other in the sorted vector.**

  If a sorting algorithm can guarantee that for any `a == b` in the input vector, the relative order of `a` and `b` in the sorted vector is the same as in the input vector, the sorting algorithm is stable.

- **Selection Sort**

  → **In the first iteration, you put the minimum value in the vector in the first place; in the second iteration, you put the second minimum value in the vector in the second place; and so on.**

- **Demo of Selection Sort**

35	65	30	60	20	Swap 35 and 20.
20	65	30	60	35	Swap 65 and 30.
20	30	65	60	35	Swap 65 and 35.
20	30	35	60	65	No swap
20	30	35	60	65	No swap
20	30	35	60	65	Sorted

- **C++ Implementation of Selection Sort**

```
void selection_sort(vector<int>& vec) {
    for (size_t i = 0; i < vec.size(); i++) {
        // Stores the index of the min value in the rest of the vector.
        size_t min = i;

        // Find the min value in the rest of the vector.
        for (size_t j = i + 1; j < vec.size(); j++) {
            if (vec.at(j) < vec.at(min)) { min = j; }
        }

        // Swap vec.at(min) with vec.at(i) if they are not the same.
        if (min != i) { swap(vec.at(i), vec.at(min)); }
    }
}
```


- **Performance of Selection Sort**

Number of Comparisons		
Best Case	**Worst Case**	**Average Case**
$O(n^2)$	$O(n^2)$	$O(n^2)$

Number of Swaps	
Best Case	**Worst Case**
$O(1)$	$O(n)$

→ **Selection sort is an in-place sorting algorithm.**

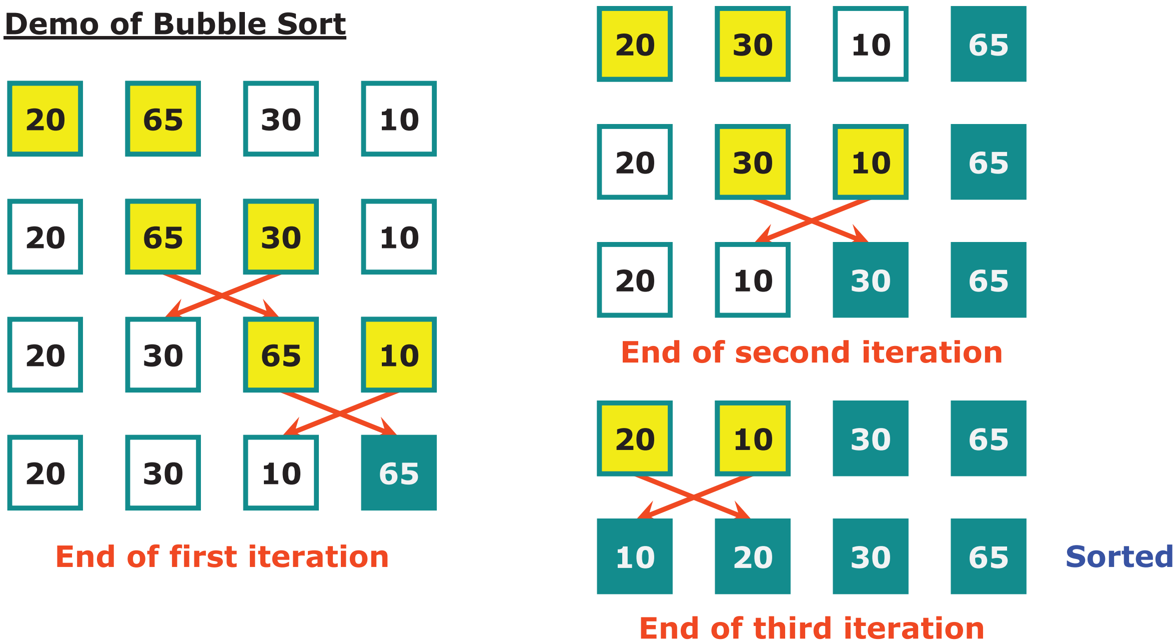
→ **Selection sort is a stable sorting algorithm.**

- **Bubble Sort**

→ Compare the adjacent pairs (e.g., vec[0] and vec[1], vec[1] and vec[2], vec[2] and vec[3], and so on) of elements. If they are out of order, swap them.

In the first iteration, you place the largest item; in the second iteration, you place the second largest item; and so on.

- **Demo of Bubble Sort**



- **C++ Implementation of Bubble Sort**

```cpp
void bubble_sort_I(vector<int>& vec) {
    for (size_t i = 0; i < vec.size(); i++) {
        for (size_t j = 1; j < vec.size(); j++) {
            if (vec.at(j) < vec.at(j - 1)) {  // Out of order
                swap(vec.at(j), vec.at(j - 1));
            }
        }
    }
}
```

→ Can we improve this solution (code)?

1) We do not need to compare values that are already placed (at the end of the vector).

2) If in any iteration, no swaps were ever occurred, it means ________?

   It means that the vector is already sorted.

   Therefore, at the end of an iteration, if there were no swaps occurred in the iteration, then we can stop any further processing of the vector, since it is already sorted.

- **C++ Implementation of Bubble Sort (Improved)**

```
void bubble_sort_II(vector<int>& vec) {
    for (size_t i = 0; i < vec.size(); i++) {
        // Stores whether a swap occurs in this iteration.
        bool swapped = false;

        for (size_t j = 1; j < vec.size(); j++) {
            if (vec.at(j) < vec.at(j - 1)) {  // Out of order
                swap(vec.at(j), vec.at(j - 1));
                swapped = true;
            }
        }

        // If no swap occurred in this iteration,
        // then the vector is already sorted.
        if (!swapped) { return; }
    }
}
```


- **Performance of Bubble Sort**

Number of Comparisons		
Best Case	Worst Case	Average Case
$O(n)$	$O(n^2)$	$O(n^2)$

Number of Swaps	
Best Case	Worst Case
$O(1)$	$O(n^2)$

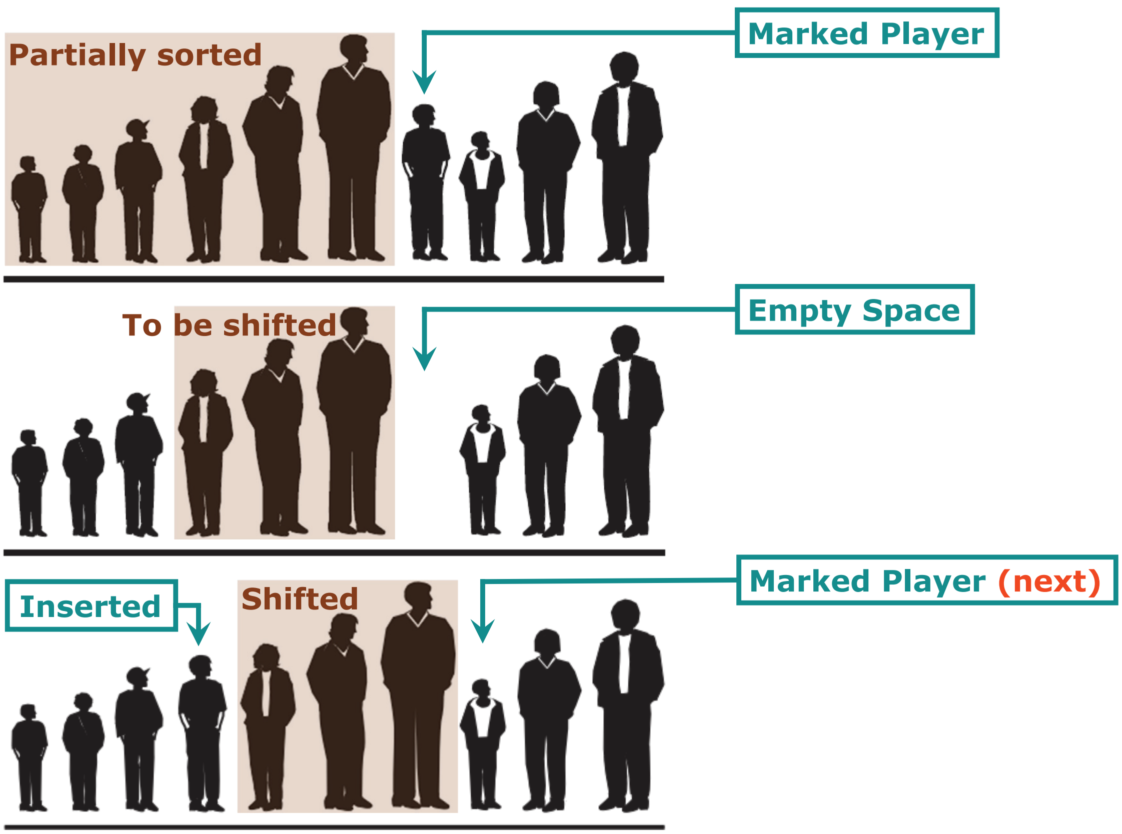
→ **Bubble sort is an in-place sorting algorithm.**

→ **Bubble sort is a stable sorting algorithm.**

→ **In general, bubble sort is a bad sorting algorithm (slow).**

- **It uses up to $O(n^2)$ swaps.**
- **Swap is a slow operation.**

- **Insertion Sort**



- **C++ Implementation of Insertion Sort**

```cpp
void insertion_sort(vector<int>& vec) {
    for (size_t mark = 1; mark < vec.size(); mark++) {
        int key = vec.at(mark), j;
        for (j = mark - 1; j >= 0 && vec.at(j) > key; j--) {
            vec.at(j + 1) = vec.at(j);  // Data shift
        }
        vec.at(j + 1) = key;
    }
}
```

- **Performance of Insertion Sort**

Number of Comparisons		
Best Case	Worst Case	Average Case
$O(n)$	$O(n^2)$	$O(n^2)$

Number of Shifts	
Best Case	Worst Case
$O(1)$	$O(n^2)$

→ **Insertion sort is an in-place sorting algorithm.**

→ **Insertion sort is a stable sorting algorithm.**

- **Summary of Quadratic Sorting Algorithms**

  → Selection sort, bubble sort, and insertion sort are in-place sorting algorithms.

  → Selection sort, bubble sort, and insertion sort are stable sorting algorithms.

  → Quadratic: $O(n^2)$ time complexity

- **External Learning Resource**

  - https://www.toptal.com/developers/sorting-algorithms
  - https://www.geeksforgeeks.org/sorting-algorithms

- **Merge Sort**

  → Merge sort contains two parts:

  1) Merge operation
  2) Merge sort algorithm

- **Merge Operation**

  → How to merge two already sorted vectors into a single sorted vector?

	i	i	i	i	i
Array 1:	23	47	81	95	

Output:	7	14	23	39	47	55	62	74	81	95

	j	j	j	j	j	j	j
Array 2:	7	14	39	55	62	74	

- **C++ Implementation of the Merge Operation**

```cpp
/** Merges two sorted vectors into a single sorted vector.
    @param vec_1: first sorted vector to merge
    @param vec_2: second sorted vector to merge
    @param vec_3: merged vector
*/
void merge(const vector<int>& vec_1, const vector<int>& vec_2,
           vector<int>& vec_3) {
    size_t i = 0, j = 0, k = 0;
    while (i < vec_1.size() && j < vec_2.size()) {
        if (vec_1.at(i) <= vec_2.at(j)) {
            vec_3.at(k++) = vec_1.at(i++);
        } else {
            vec_3.at(k++) = vec_2.at(j++);
        }
    }
    while (i < vec_1.size()) { vec_3.at(k++) = vec_1.at(i++); }
    while (j < vec_2.size()) { vec_3.at(k++) = vec_2.at(j++); }
}
```

→ **Time complexity of the merge operator:** $O(m + n)$

m, n are the size of the two input vectors.

If m and n are (almost) equal, then the time complexity can be simply written as $O(n)$.

- **Merge Sort**

→ **Basic Idea (Algorithm):**

1) **Divide the entire vector into two halves, the left half and right half.**
2) **Sort the left half.**
3) **Sort the right half.**
4) **Merge the sorted left half and right half to form sorted whole vector.**

→ **[Important] How to "sort the left half"? How to "sort the right half"?**

Merge sort is a recursive algorithm.

→ **Algorithm**

1) **If the size of the vector is less than 2, then return (base case).**
2) **Copy the left half of the vector into another vector (denoted as `left_half`).**
3) **Copy the right half of the vector into another vector (denoted as `right_half`).**
4) **Recursively sort `left_half`.**
5) **Recursively sort `right_half`.**
6) **Merge `left_half` and `right_half`.**

- **C++ Implementation of Merge Sort**

```cpp
void merge_sort(vector<int>& vec) {
    // Base case
    if (vec.size() < 2) { return; }

    // Copy the left and right half of the vector into 2 other vectors.
    vector<int> left, right;
    for (size_t i = 0; i < vec.size(); i++) {
        if (i < vec.size() / 2) { left.push_back(vec.at(i)); }
        else { right.push_back(vec.at(i)); }
    }

    // Sort "left" and "right" recursively.
    merge_sort(left);
    merge_sort(right);

    // Merge the sorted left and right half.
    merge(left, right, vec);
}
```


- **Performance of Merge Sort**

→ **Merge:** $\mathrm{O}(n)$

→ **Merge sort:** $T(n) = 2T\left(\frac{n}{2}\right) + O(n) = O(n\log n)$ **(Master's theorem)**

Number of Comparisons		
Best Case	**Worst Case**	**Average Case**
$\mathrm{O}(n\log n)$	$\mathrm{O}(n\log n)$	$\mathrm{O}(n\log n)$

→ **Merge sort is not an in-place sorting algorithm.**

We need extra memory space to store the left and right halves of the input vector.

→ **Merge sort is not a quadratic sorting algorithm.**

Time complexity of merge sort is $\mathrm{O}(n\log n)$, instead of $\mathrm{O}(n^2)$.

→ **Merge sort is a stable sorting algorithm.**

- **Best time complexity for comparison sorting algorithms**

→ **For comparison sorting algorithms, $O(n \log n)$ is already the optimal time complexity.**

In other words, no comparison sorting algorithm can do better than $O(n \log n)$ (can be mathematically proved).

→ **Merge sort is very fast since it reaches the optimal time complexity.**

- **Example of Non-Comparison Sorting Algorithm**

→ **Radix Sort**

3487	1090	1128	1090	1090
2873	2932	2932	1128	1128
4378	2873	3439	3287	2873
1090	8743	8743	4378	2932
8743	3487	2873	3439	3287
1128	3287	4378	3487	3439
3439	4378	3487	8743	3487
3287	1128	3287	2873	3888
2932	3888	3888	3888	4378
3888	3439	1090	2932	8743

→ **Other non-comparison sorting algorithms (not required):**

- **Counting sort**
- **Bucket sort**