정렬

물론입니다! 아래는 C++에서 **STL을 사용한 경우**와 **STL 없이 직접 구현한 경우**로, 버블, 선택, 삽입, 퀵, 병합, 기수, 카운터 정렬에 대한 **코드**, **정렬 방법 설명**, 그리고 각 정렬 알고리즘의 **시간 복잡도**, **공간 복잡도**를 정리한 내용입니다.

📚 정렬 알고리즘 정리 (C++ 기준)

정렬 알고리즘	STL 사용 여부	시간 복잡도 (최선/평균/최악)	공간 복잡도	특징
버블 정렬 (Bubble Sort)	✅ STL	O(n²) / O(n²) / O(n²)	O(1)	인접한 원소를 비교하며 순서를 정함. 안정 정렬.
선택 정렬 (Selection Sort)	✅ STL	O(n²) / O(n²) / O(n²)	O(1)	가장 작은 원소를 찾아 위치 변경. 안정 정렬 아님.
삽입 정렬 (Insertion Sort)	✅ STL	O(n²) / O(n²) / O(n²)	O(1)	부분적으로 정렬된 데이터에 효율적. 안정 정렬.
퀵 정렬 (Quick Sort)	✅ STL	O(n log n) (평균) / O(n²) (최악)	O(log n) (재귀 스택)	분할 기반. 평균적으로 매우 빠름.
병합 정렬 (Merge Sort)	✅ STL	O(n log n) / O(n log n) / O(n log n)	O(n)	분할-병합 구조. 안정 정렬.
기수 정렬 (Radix Sort)	✅ STL	O(d × (n + k))	O(n + k)	자릿수 기반 정렬. 정수/문자에 적합.
카운터 정렬 (Counting Sort)	✅ STL	O(n + k)	O(k)	범위가 제한된 정수 정렬. 최적 O(n) 가능.

🔍 각 정렬 알고리즘 설명 및 코드

---

1. 버블 정렬 (Bubble Sort)

설명: 인접한 두 원소를 비교하여 큰 값을 뒤로 이동. 반복적으로 순서를 정함.

```cpp
// STL 사용 (std::vector + std::sort)
#include <vector>
#include <algorithm>
std::vector<int> arr = {64, 34, 25, 12, 22, 11, 90};
```

```cpp
std::sort(arr.begin(), arr.end()); // O(n²) 최악

// STL 없이 직접 구현
void bubbleSort(std::vector<int>& arr) {
    int n = arr.size();
    for (int i = 0; i < n-1; ++i) {
        for (int j = 0; j < n-i-1; ++j) {
            if (arr[j] > arr[j+1]) {
                std::swap(arr[j], arr[j+1]);
            }
        }
    }
}
```

---

2. 선택 정렬 (Selection Sort)

설명: 가장 작은 원소를 찾아 위치에 삽입. 전체 순회 중 최소값을 찾음.

```cpp
// STL
std::sort(arr.begin(), arr.end());

// 직접 구현
void selectionSort(std::vector<int>& arr) {
    int n = arr.size();
    for (int i = 0; i < n-1; ++i) {
        int min_idx = i;
        for (int j = i+1; j < n; ++j) {
            if (arr[j] < arr[min_idx]) {
                min_idx = j;
            }
        }
        std::swap(arr[i], arr[min_idx]);
    }
}
```

---

3. 삽입 정렬 (Insertion Sort)

설명: 부분적으로 정렬된 데이터에 매우 효율적. 각 원소를 정확한 위치에 삽입.

```cpp
// STL
std::sort(arr.begin(), arr.end());

// 직접 구현
```

```cpp
void insertionSort(std::vector<int>& arr) {
    int n = arr.size();
    for (int i = 1; i < n; ++i) {
        int key = arr[i];
        int j = i - 1;
        while (j >= 0 && arr[j] > key) {
            arr[j+1] = arr[j];
            --j;
        }
        arr[j+1] = key;
    }
}
```

---

4. 퀵 정렬 (Quick Sort)

설명: 피봇 기반 분할. 평균적으로 매우 빠르며, 최악 경우 $O(n^2)$ 발생 가능.

```cpp
// STL
std::sort(arr.begin(), arr.end());

// 직접 구현 (재귀 버전)
void quickSort(std::vector<int>& arr, int low, int high) {
    if (low < high) {
        int pivot = partition(arr, low, high);
        quickSort(arr, low, pivot - 1);
        quickSort(arr, pivot + 1, high);
    }
}

int partition(std::vector<int>& arr, int low, int high) {
    int pivot = arr[high];
    int i = low - 1;
    for (int j = low; j < high; ++j) {
        if (arr[j] <= pivot) {
            ++i;
            std::swap(arr[i], arr[j]);
        }
    }
    std::swap(arr[i+1], arr[high]);
    return i + 1;
}
```

---

5. 병합 정렬 (Merge Sort)

설명: 분할-병합 구조. 안정 정렬이며 O(n log n) 보장.

```
// STL
std::sort(arr.begin(), arr.end());

// 직접 구현
void merge(std::vector<int>& arr, int left, int mid, int right) {
    std::vector<int> temp;
    int i = left, j = mid + 1;
    while (i <= mid && j <= right) {
        temp.push_back(arr[i] <= arr[j] ? arr[i++] : arr[j++]);
    }
    while (i <= mid) temp.push_back(arr[i++]);
    while (j <= right) temp.push_back(arr[j++]);
    for (int k = 0; k < temp.size(); ++k) arr[left + k] = temp[k];
}

void mergeSort(std::vector<int>& arr, int left, int right) {
    if (left < right) {
        int mid = left + (right - left) / 2;
        mergeSort(arr, left, mid);
        mergeSort(arr, mid + 1, right);
        merge(arr, left, mid, right);
    }
}
```

---

6. 기수 정렬 (Radix Sort)

설명: 자릿수 기반 정렬. 정수 또는 문자열에 적합. O(d × (n + k)) (d: 자릿수, k: 범위)

```
// STL
std::sort(arr.begin(), arr.end());

// 직접 구현 (10진수 기반)
void radixSort(std::vector<int>& arr) {
    int max = *std::max_element(arr.begin(), arr.end());
    for (int exp = 1; max / exp > 0; exp *= 10) {
        countingSortByDigit(arr, exp);
    }
}

void countingSortByDigit(std::vector<int>& arr, int exp) {
    std::vector<int> output(arr.size());
    std::vector<int> count(10);
    for (int num : arr) {
        int digit = (num / exp) % 10;
```

```
        count[digit]++;
    }
    for (int i = 1; i < 10; ++i) {
        count[i] += count[i-1];
    }
    for (int i = arr.size()-1; i >= 0; --i) {
        int digit = (arr[i] / exp) % 10;
        output[count[digit]-1] = arr[i];
        count[digit]--;
    }
    for (int i = 0; i < arr.size(); ++i) {
        arr[i] = output[i];
    }
}
```

---

7. 카운터 정렬 (Counting Sort)

설명: 범위가 제한된 정수에 최적. O(n + k) 시간 복잡도.

```
// STL
std::sort(arr.begin(), arr.end());

// 직접 구현
void countingSort(std::vector<int>& arr, int max_val) {
    std::vector<int> count(max_val + 1, 0);
    for (int num : arr) {
        count[num]++;
    }
    int idx = 0;
    for (int i = 0; i <= max_val; ++i) {
        while (count[i] > 0) {
            arr[idx++] = i;
            count[i]--;
        }
    }
}
```

📌 정리 요약

알고리즘	STL 사용 여부	시간 복잡도 (평균)	공간 복잡도	안정성	적합한 경우
버블	✅	O(n²)	O(1)	✅	작은 데이터, 교육용
선택	✅	O(n²)	O(1)	❌	작은 데이터

알고리즘	STL 사용 여부	시간 복잡도 (평균)	공간 복잡도	안정성	적합한 경우
삽입	✅	O(n²)	O(1)	✅	부분 정렬된 데이터
퀵	✅	O(n log n)	O(log n)	❌	대부분의 경우
병합	✅	O(n log n)	O(n)	✅	안정 정렬 필요, 대용량
기수	✅	O(d×(n+k))	O(n+k)	✅	정수, 문자열, 정해진 범위
카운터	✅	O(n+k)	O(k)	✅	범위 제한된 정수

> 💡 **STL 사용 시** `std::sort` 는 일반적으로 **퀵 정렬 또는 인터넷 정렬 기반**으로 구현되어 있으며, **평균적으로 매우 빠르고** 대부분의 경우 최적의 성능을 보임.
> 그러나 **정렬 방식이 정해져 있어야** (예: 정수, 정렬 가능) `std::sort` 가 최적임.

✅ 추가 팁

- **STL을 사용하면** `std::sort` 는 대부분의 경우 최적의 성능을 보장.
- **자체 구현**은 학습 및 알고리즘 이해에 유용.
- **안정 정렬**이 필요한 경우 (예: 동일한 값의 순서 보존) → **병합, 삽입, 기수, 카운터**가 적합.
- **대용량 정렬** → **병합 또는 퀵** (STL 기반) 추천.

이 정리가 도움이 되셨나요? 더 구체적인 예제나 특정 알고리즘에 대한 설명이 필요하시면 알려주세요! 😊