Interchange

Interchange Sort

Interchange Sort là thuật toán sắp xếp duyệt qua các cặp số trong mảng nếu số có thứ tự nhỏ hơn có giá trị lớn hơn thì đổi chỗ hai số đó. Ta có code của **Interchange Sort** như sau:

```
for(int i = 1; i <= n - 1; i++)
    for(int j = i + 1; j <= n; j++)
        if (a[i] > a[j]) swap(a[i],a[j]);
```

Độ phức tạp thuật toán: O(n^2)

Quick Sort

Quick Sort là thuật toán cải tiến của **Interchange Sort**. Áp dụng kỹ thuật chia để trị, **Quick Sort** sẽ chọn một phần tử làm **key** sau đó hoán đổi vị trí sao cho các phần tử nhỏ hơn **key** sẽ nằm bên trái nó, các phần tử lớn hơn sẽ nằm bên phải rồi tiếp tục sử dụng đệ quy với 2 phần bên trái và phải của **key**. Ta có code của **Quick Sort** như sau:

```
void QuickSort(int l, int r) {
    if (l >= r) return;

    int key = a[(l + r) >> 1];

    int i = l, j = r;
        while (i < j) {
                while (a[i] < key) i++;
                while (a[j] > key) j--;
                if (i <= j){
                        swap(a[i],a[j]);
                        i++;
                        j--;
                }
        }
    QuickSort(l,j);
    QuickSort(i,r);
}
```

Độ phức tạp trung bình của **Quick Sort** là O(nlogn), tệ nhất có thể lên tới O(n^2)

Bubble Sort

Bubble Sort được xây dựng tương tự như việc nổi bọt trong thực tế. Cụ thể, thuật toán này hoạt động bằng cách so sánh từng cặp phần tử liền kề và hoán đổi chúng nếu chúng không theo đúng thứ tự. Quá trình này lặp lại nhiều lần cho đến khi mảng được sắp xếp hoàn toàn. Thuật toán được gọi là **nổi bọt** vì các phần tử lớn dần nổi lên phía cuối mảng qua từng vòng lặp. Ta có code của **Bubble Sort** như sau:

```
for(int i = 1; i <= n; i++)
    for(int j = 1; j <= n - i; j++)
        if (a[j] > a[j + 1]) swap(a[j],a[j + 1]);
```

Độ phức tạp thuật toán: O(n^2)

Shake Sort

Shake Sort là một biến thể cải tiến của **Bubble Sort**.

Điểm khác biệt chính là thay vì chỉ duyệt theo một chiều (**Bubble Sort** chỉ đẩy phần tử lớn lên cuối), **Shake Sort** duyệt theo cả hai chiều trong mỗi vòng lặp, giúp giảm số lần duyệt cần thiết.

Ta có code của **Shake Sort** như sau:

```
int l = 1, r = n;

while(l < r) {
    for(int i = l; i <= r - 1; i++)
        if (a[i] > a[i + 1]) swap(a[i],a[i + 1]);
    r--;
    for(int i = r; i >= l + 1; i--)
        if (a[i] > a[i + 1]) swap(a[i],a[i +1]);
    l++;
}
```

Độ phức tạp thuật toán: O(n ^ 2)

Insertion

Insertion Sort

Insertion Sort chia mảng thành hai phần đã sắp xếp và chưa sắp xếp, chèn từng phần tử vào vị trí phù hợp.

Ta có code của **Insertion Sort** như sau:

```
for(int i = 1; i <= n; i++)
    cur = a[i];
    j = i;
    while(j > 0 && cur < a[j - 1]) {
        a[j] = a[j - 1];
        j--;
    }
    a[j] = cur;
```

Độ phức tạp thuật toán: tốt nhất là O(n), tệ nhất là O(n^2).

Binary Insertion Sort

Binary Insertion Sort là thuật toán cải tiến từ **Insertion Sort** bằng cách dùng tìm kiếm nhị phân để tìm vị trí chèn, giảm số phép so sánh nhưng vẫn phải dịch chuyển phần tử. Ta có code của như sau:

```
for (int i = 1; i < n; i++) {
        int value = a[i];
        int L = 0, R = i - 1, pos = i;

        while (L <= R) {
            int mid = (L + R) / 2;
            if (a[mid] <= value) {
                pos = mid + 1;
                L = mid + 1;
            } else R = mid - 1;

        }
        for (int j = i; j > pos; j--) a[j] = a[j - 1];
        a[pos] = value;
}
```

Độ phức tạp: tốt nhất là O(nlogn), tệ nhất là O(n^2)

Shell Sort

Shell Sort là thuật toán cải tiến từ **Insertion Sort**, nhưng thay vì so sánh các phần tử kề nhau, ta so sánh và hoán đổi các phần tử cách nhau một khoảng gap(k), giảm số lần dịch chuyển phần tử xa hơn. Ta có code của **Shell Sort** như sau:

```
for (int k = n / 2; k >= 1; k /= 2) {
        for (int i = k; i < n; i++) {
            int value = a[i];
            int j = i;

            while (j >= k && a[j - k] > value) {
                a[j] = a[j - k];
                j -= k;
            }
            a[j] = value;
        }
    }
```

Độ phức tạp thuật toán: Tốt nhất là O(nlogn), tệ nhất là O(n^2)

Selection

Selection Sort

Selection Sort là thuật toán tìm phần tử nhỏ nhất và hoán đổi với phần tử đầu tiên, lặp lại cho phần còn lại. Ta có code của **Selection Sort** như sau:

```
for (int i = 0; i < n - 1; i++) {
        int min_index = i;

        for (int j = i + 1; j < n; j++) {
            if (a[j] < a[min_index]) {
                min_index = j;
            }
        }
        swap(a[i], a[min_index]);
    }
```

Độ phức tạp thuật toán: O(n^2)

Heap Sort

Heap Sort là thuật toán được cải tiến từ **Selection Sort**. Dùng cấu trúc dữ liệu **Heap** để tìm phần tử nhỏ nhất/lớn nhất nhanh chóng.

```
priority_queue<int> pq;
for (int i = 0; i < n; i++) {
    pq.push(a[i]);
}

for (int i = n - 1; i >= 0; i--) {
    a[i] = pq.top();
    pq.pop();
}
```

Độ phức tạp thuật toán: O(nlogn)

Merge

Merge Sort

Merge Sort sử dụng phương pháp **chia để trị**, chia nhỏ mảng và trộn các mảng con đã được sắp xếp. Ta có code **Merge Sort** như sau:

```
if (left >= right) return; // Điều kiện dừng đệ quy

    int mid = left + (right - left) / 2;

    // Gọi đệ quy sắp xếp hai nửa
    mergeSort(a, left, mid);
    mergeSort(a, mid + 1, right);

    // Trộn hai mảng đã sắp xếp
    int n1 = mid - left + 1, n2 = right - mid;
    int L[n1], R[n2];

    for (int i = 0; i < n1; i++) L[i] = a[left + i];
    for (int i = 0; i < n2; i++) R[i] = a[mid + 1 + i];

    int i = 0, j = 0, k = left;
    while (i < n1 && j < n2) {
        a[k++] = (L[i] <= R[j]) ? L[i++] : R[j++];
    }

    while (i < n1) a[k++] = L[i++];
    while (j < n2) a[k++] = R[j++];
```

Độ phức tạp thuật toán: O(nlogn)

Natural Merge Sort

Natural Merge Sort là một biến thể của **Merge Sort**, nhưng thay vì chia mảng theo kích thước cố định, nó tận dụng các dãy con đã được sắp xếp sẵn trong mảng để giảm số lần chia nhỏ mảng.

Ý tưởng chính:

- Tìm các dãy con đã sắp xếp sẵn trong mảng.
- Trộn các dãy con đó lại với nhau cho đến khi toàn bộ mảng được sắp xếp.

Ta có code **Natural Merge Sort** như sau:

```
void naturalMergeSort(vector<int>& arr) {
    int n = arr.size();
    while (true) {
        vector<pair<int, int>> runs;
        int start = 0;

        while (start < n) {
            int end = start;
            while (end + 1 < n && arr[end] <= arr[end + 1]) end++;
            runs.push_back({start, end});
            start = end + 1;
        }

        if (runs.size() == 1) break;

        for (size_t i = 0; i + 1 < runs.size(); i += 2) {
            int left = runs[i].first, mid = runs[i].second, right = runs[i + 1].seco
            vector<int> temp;
            int l = left, r = mid + 1;

            while (l <= mid && r <= right)
                temp.push_back(arr[l] <= arr[r] ? arr[l++] : arr[r++]);
            while (l <= mid) temp.push_back(arr[l++]);
            while (r <= right) temp.push_back(arr[r++]);

            for (size_t j = 0; j < temp.size(); j++)
                arr[left + j] = temp[j];
        }
    }
}
```

Độ phức tạp thuật toán: Tốt nhất là O(n), tệ nhất là O(nlogn)

K-way Merge Sort

K-Way Merge Sort là một biến thể của **Merge Sort**, thay vì chia mảng thành 2 phần, thuật toán này chia thành K phần nhỏ hơn rồi trộn lại bằng cách sử dụng cấu trúc dữ liệu **hàng đợi ưu tiên (priority queue/min-heap)**.

Ý tưởng chính:

- Chia mảng thành K phần nhỏ hơn và sắp xếp từng phần.
- Sử dụng hàng đợi ưu tiên (min-heap) để trộn K dãy đã sắp xếp lại thành một mảng hoàn chỉnh.

Ta có code **K-Way Merge Sort** như sau:

```
void kWayMergeSort(vector<int>& arr, int K) {
    int n = arr.size();
    if (n <= 1) return;

    vector<vector<int>> subArrays(K);
    for (int i = 0; i < n; i++)
        subArrays[i % K].push_back(arr[i]);

    for (int i = 0; i < K; i++)
        sort(subArrays[i].begin(), subArrays[i].end());

    priority_queue<pair<int, pair<int, int>>, vector<pair<int, pair<int, int>>>, gre

    for (int i = 0; i < K; i++) {
        if (!subArrays[i].empty())
            minHeap.push({subArrays[i][0], {i, 0}});
    }

    int index = 0;
    while (!minHeap.empty()) {
        auto [val, pos] = minHeap.top();
        minHeap.pop();

        int i = pos.first, j = pos.second;
        arr[index++] = val;

        if (j + 1 < subArrays[i].size())
            minHeap.push({subArrays[i][j + 1], {i, j + 1}});
    }
}
```

Độ phức tạp thuật toán: O(nlogK)

Specific

Counting Sort

Counting Sort là thuật toán sắp xếp không dựa trên so sánh, hoạt động bằng cách đếm số lần xuất hiện của mỗi phần tử, sau đó sử dụng thông tin này để sắp xếp mảng. Ta có code của **Counting Sort** như sau:

```

void countingSort(vector<int>& arr) {
    if (arr.empty()) return;

    int maxVal = *max_element(arr.begin(), arr.end());
    vector<int> count(maxVal + 1, 0);

    for (int num : arr) count[num]++;

    for (int i = 1; i <= maxVal; i++) count[i] += count[i - 1];

    vector<int> sorted(arr.size());
    for (int i = arr.size() - 1; i >= 0; i--) {
        sorted[count[arr[i]] - 1] = arr[i];
        count[arr[i]]--;
    }

    arr = sorted;
}
```

Độ phức tạp thuật toán: O(n + k)

Radix Sort

Radix Sort là thuật toán sắp xếp không dựa trên so sánh, hoạt động bằng cách sắp xếp các chữ số theo từng vị trí (đơn vị, chục, trăm, ...) bằng **Counting Sort**.

Ý tưởng chính:

- Xác định số có nhiều chữ số nhất (số lớn nhất trong mảng).
- Sắp xếp từng chữ số từ hàng đơn vị đến hàng cao nhất bằng Counting Sort.
- Lặp lại quá trình cho các hàng tiếp theo.

```
void radixSort(vector<int>& arr) {
    if (arr.empty()) return;

    int maxVal = *max_element(arr.begin(), arr.end());

    for (int exp = 1; maxVal / exp > 0; exp *= 10) {
        vector<int> count(10, 0), sorted(arr.size());

        for (int num : arr) count[(num / exp) % 10]++;

        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;
            sorted[count[digit] - 1] = arr[i];
            count[digit]--;
        }

        arr = sorted;
    }
}
```

Độ phức tạp thuật toán: O(n * k)